Tru64 UNIX

Security Programming

Part Number: AA-RSFDA-TE

September 2002

Product Version: Tru64 UNIX Version 5.1B or higher

This manual describes how to write programs that include the use of Tru64 UNIX security features.

Hewlett-Packard Company
Palo Alto, California

Contents

About This Manual

This manual describes how to write programs that include the use of HP Tru64 UNIX security features.

Audience

This manual is intended for programmers who are modifying or creating security-relevant programs and are familiar with programming in C on UNIX systems.

Organization

The manual is organized as follows:

Chapter 1	Describes the approach to examples used throughout this guide and provides information about the trusted computing base.
Chapter 2	Provides specific techniques for designing trusted programs, such as whether the program is to be a directly executed command or a daemon.
Chapter 3	Describes the structure of the authentication database and the techniques for querying it.
Chapter 4	Describes the various user and group identities of the operating system and how you should use them, particularly the audit ID that is not a part of traditional UNIX systems. It also describes the contents of the enhanced (protected) password database.
Chapter 5	Describes guidelines for when trusted programs should make entries in the audit logs and the mechanisms for doing so.
Chapter 6	Describes the Security Integration Architecture (SIA) programming interface.
Chapter 7	Describes the use of access control lists (ACLs) in applications that run on Tru64 UNIX.
Chapter 8	Describes the GSS-API standard and security fundamentals and GSS Application Security SDK function calls with best practices and portability concerns for using them.
Appendix A	Provides coding examples for trusted Tru64 UNIX systems.

Appendix B	Contains the default auditable events (`/etc/sec/audit_events`) and the default audit-event aliases (`/etc/sec/event_aliases`) files.
Appendix C	Describes how to use GSS-APIs to secure an application using C-programming language example code. It also explains the sample programs provided Application Security SDK.

Related Documentation

The following documents provide important information that supplements the information in certain chapters:

- The *Security Administration* manual describes how to perform common Tru64 UNIX administrative tasks.
- The *Release Notes* might contain important undocumented information about security.

Reader's Comments

HP welcomes any comments and suggestions you have on this and other Tru64 UNIX manuals.

You can send your comments in the following ways:

- Fax: 603-884-0120 Attn: UBPG Publications, ZKO3-3/Y32
- Internet electronic mail: `readers_comment@zk3.dec.com`

  A Reader's Comment form is located on your system in the following location:

  `/usr/doc/readers_comment.txt`

Please include the following information along with your comments:

- The full title of the manual and the order number. (The order number appears on the title page of printed and PDF versions of a manual.)
- The section numbers and page numbers of the information on which you are commenting.
- The version of Tru64 UNIX that you are using.
- If known, the type of processor that is running the Tru64 UNIX software.

The Tru64 UNIX Publications group cannot respond to system problems or technical support inquiries. Please address technical questions to your local system vendor or to the appropriate HP technical support office. Information provided with the software media explains how to send problem reports to HP.

Conventions

This manual uses the following typographical conventions:

\	A backslash at the end of a line in an example indicates continuation.
#	A number sign represents the system prompt when you are logged in to a Tru64 UNIX system using the root user account.
net stop	Bold courier type indicates user input.
>>>	The console subsystem prompt is three right angle brackets.
file	Italic (slanted) type indicates variable values, placeholders, and function argument names.
[|] { | }	In syntax definitions, brackets indicate items that are optional and braces indicate items that are required. Vertical bars separating items inside brackets or braces indicate that you choose one item from among those listed.
. . .	In syntax definitions, a horizontal ellipsis indicates that the preceding item can be repeated one or more times.
cat(1)	A cross-reference to a reference page includes the appropriate section number in parentheses. For example, cat(1) indicates that you can find information on the cat command in Section 1 of the reference pages.
[Ctrl/x]	This symbol indicates that you hold down the first named key while pressing the key or mouse button that follows the slash. In examples, this key combination is enclosed in a box (for example, [Ctrl/C]).

1 Introduction for Programmers

This chapter describes the implication of running trusted applications on a trusted Tru64 UNIX system. Libraries, header files, the standard trusted system directories and the trusted computing base (TCB) are discussed. This chapter and the ones that follow use partial and complete C programs to illustrate basic ideas. Although some of these can be used without modification, they are not a collection of routines from which you can assemble trusted programs.

In addition to the programming techniques described in this manual, you can also use the following programming techniques:

- Common Data Security Architecture (CDSA)

  CDSA is a multiplatform, industry standard security infrastructure. It provides a standards-based, stable programming interface that applications can use to access operating system security services, allowing developers to create cross-platform, security-enabled applications. Applications request security services, such as cryptography and other public key operations, through a dynamically extensible application programming interface (API). These requests are serviced by a set of plug-in security service modules (SPIs), which can be supplemented or changed as business needs and technologies evolve.

  See `cdsa`(3) for more information about CDSA.

- Secure Socket Layer (SSL)

  SSL is a commonly-used protocol for managing the security of a message transmission on the Internet. SSL has recently been succeeded by Transport Layer Security (TLS), which is based on SSL. SSL uses a program layer located between the Hypertext Transfer Protocol (HTTP) and Transport Control Protocol (TCP) layers. The provided OpenSSL library implements the Secure Sockets Layer (SSL v2/v3) and Transport Layer Security (TLS v1) protocols.

  See `ssl`(3) for more information about SSL.

This chapter contains the following information:

- Security programming overview
- Libraries and header files
- Standard trusted system directories

- Security-relevant system calls and library routines

This chapter and the ones that follow use partial and complete C programs to illustrate basic ideas. Although some of these can be used without modification, they are not a collection of routines from which you can assemble trusted programs.

1.1 Security Programming Overview

You must protect the trusted computing base (TCB) from unintended modification. To do this, you first define which of your programs and data files are a part of the TCB. The following list describes the components of the TCB:

- *Trusted Programs:* Any program that could subvert a security rule must be considered a trusted program. This includes programs that make direct security decisions, and those that do not, but could subvert security if they contained errors or malicious code. Consider a program trusted if the program file has its user ID set to root (SUID).
- *Indirect Programs:* A program is trusted if another trusted program invokes it or otherwise interacts with it and depends upon its actions for security decisions. A program is also trusted if it modifies a data file or other object upon which another trusted program depends.
- *Program Files:* Executable files that contain a trusted program are considered a part of the TCB.
- *Object Code and Libraries:* All object (binary) code modules and their files, whether statically or dynamically linked, that are included in a trusted program are part of the TCB. This includes the standard C library routines and interfaces, which are frequently used by trusted programs.
- *Data Files:* The TCB includes any file that contains data used by a trusted program to make a security decision, for example, the `ttys` database.
- *Shell Scripts:* A shell script is a data file that a shell program interprets, performing the shell commands in the file. A shell script is considered part of the TCB if it performs a function on behalf of a trusted program or if it is needed for correct operation of the system. You can determine if a shell script is security relevant if removing or replacing the script would cause the system to perform improperly (for example, removing some of the `rc` startup scripts) or provide an opportunity for a security breach (installing a different `cron` startup file). Shell script files should be protected as carefully as object code program files. Note that a shell script must be readable to be executed.
- *Antecedent Directories:* Consider all parent directories of TCB files a part of the TCB and protect them accordingly. If malicious users can remove

and redefine links in these directories, then they can create new, phony files that might cause a trusted program to make an incorrect security decision.

1.1.1 Protecting TCB Files

Each of the following mechanisms presents a way to protect the files and directories of the TCB:

- *Discretionary Access Control (DAC):* Discretionary access control (the owner, group, mode bits, and access control lists (ACLs)) is the most important protection for TCB files. It must prevent untrusted users and groups from modifying these files, although they might be allowed to read the files. It is common to create pseudousers and pseudogroups for this purpose.

  Existing programs may copy only the mode bits when replicating a file and therefore accidentally delete the ACL. This removes the protection offered by the ACL. HP recommends that you use restrictive traditional permissions, such as `other::---` and `group::---`, and then grant access to individual users with user entries. Using this approach, if an ACL is lost, unintended access is not allowed. See Chapter 7 for information on programming with ACLs.
- *Read-Only File Systems:* You can place all files that only need to be read on a separate file system and mount that file system as read only. This ensures that no program, no matter how privileged, can alter those files (at least short of remounting the file system). You can, of course, remount the file system as read/write if you need to alter the files. This is somewhat drastic but offers good protection against corruption of security data. You can also physically set a read only locking tab on many kinds of removable media.
- *Sticky Bit:* Tru64 UNIX includes the sticky bit on directories. The sticky bit restricts the removal of directory entries (links) to those owned by the requesting user or the owner of the directory. Without this protection, programs only need write access to the directory. Use the sticky bit where appropriate; for example, when a program needs to store files owned by different users in a single directory.

1.2 Libraries and Header Files

Your system documentation contains reference pages for all security system calls (section 2) and routines (section 3).

The `libsecurity.so`, `libaud.a`, `libaud.so`, `libpacl.a`, and the `libpacl.so` libraries hold the enhanced security interface binaries. Use the `–lsecurity` compilation option to link these into your program, for example:

```
$ cc ... –lsecurity –ldb –lm –laud ...
```

Your programs need to include several header files that hold definitions (constants, macros, structures, library interfaces, and so forth) necessary to use the Tru64 UNIX security interfaces. Following traditional UNIX practice, all Tru64 UNIX system call and library reference pages denote the header files that you need to use their routines. You are likely to use the following individual header files, in the order listed:

`<sys/secdefines.h>`	Defines compilation constants that determine the security configuration of your system. You always need to include this file first.
`<sys/security.h>`	Holds general definitions. You almost always need to include this file.
`<sys/acl.h>`	For access control lists. You need this if you manipulate access control lists.
`<prot.h>`	Defines the authentication databases and Tru64 UNIX protected subsystems. You need these if your program accesses any of the authentication databases.
`<sys/audit.h>`	Defines the audit subsystem constants for security audit interfaces. You need this if you generate or process audit records.
`<protcmd.h>`	Provides a few miscellaneous definitions for trusted commands that are delivered with Tru64 UNIX. You seldom need these.
`<sia.h>`	SIA constants, structures, and macro definitions
`<siad.h>`	SIA constants, structures, and macro definitions internally used by the interfaces and security mechanisms

1.3 Standard Trusted System Directories

Tru64 UNIX defines several directories to hold its security information. You can review the reference pages for a description of these files and directories, primarily the section 4 reference pages.

You may need to create new files and directories in the standard trusted system directories. Generally, you should create new directories for the files you place in these trees. Do not simply insert new files in existing directories unless that directory was explicitly created for such files. Table 1–1 lists the directories you might use:

Table 1–1: Standard Trusted System Directories

Directory	Contents
`/tcb/bin, /usr/tcb/bin`	Contains directly executed trusted commands and daemons.
`/tcb/lib`	Contains programs that are run by other trusted programs but are never invoked from the command line.
`/tcb/files`	Contains control files, databases, and scripts used by the trusted computing base (TCB). You can define a subdirectory of this directory for your protected subsystem, if necessary.
`/var/tcb`	Alternative to the `/tcb` directory.

1.4 Security-Relevant System Calls and Library Routines

The tables in the following sections list many of the Tru64 UNIX system calls and library routines that have security implications for programmers.

Note that some system calls and library routines not covered in these sections might also have implicit security concerns.

The misuse of a system call or library routine that does not seem to have any security concerns could threaten the security of a computer system. For example, all system calls bypass file access permissions when called by a privileged process. Ultimately, programmers are responsible for the security implications of their programs.

1.4.1 System Calls

Table 1–2 lists the system calls that have security relevance for programmers.

Table 1–2: Security-Relevant System Calls

Category	System Calls
File control	`creat`, `open`, `fcntl`, `read`, `mknod`[a], `write`
Process control	`fork`, `sigpause`, `execve`, `sigsetmask`, `setpgrp`[a], `sigvec`, `sigblock`
File attributes	`access`, `chroot`[a], `chmod`[a], `stat`, `chown`[a], `umask`
User and group ID	`getegid`, `getuid`, `getgid`, `setgroups`[a], `geteuid`, `setreuid`[a]
Auditing	`audcntl`[a], `audgen`[a]
General	`syscall`

[a] These system calls can be called only by a privileged process or they may behave differently when called by a nonprivileged process. See the associated reference pages for more information.

1.4.2 Library Routines

Library routines are system services that programs can call. Many library routines use system calls. Table 1–3 lists Tru64 UNIX library routines that have security implications.

Table 1–3: Security-Relevant Library Routines

Category	Library Routines
File control	`fopen, popen`
Password handling	`getpass, putpwent, getpwnam, setpwent, getpwent, endpwent, getpwuid, passlen, pw_mapping, randomword, time_lock`
Process control	`signal`

2 Trusted Programming Techniques

This chapter presents specific techniques for designing trusted programs. This chapter contains the following information:

- Writing SUID and SGID programs
- Handling errors
- Protecting files
- Specifying a secure search path
- Responding to signals
- Using open file descriptors with child processes
- Security concerns in the X environment

2.1 Writing SUID and SGID Programs

SUID (set user ID) and SGID (set group ID) programs change the effective UID or GID of a process to the UID or GID of the program. They are a solution to the problem of providing controlled access to system-level files and directories, because they give a process the access rights of the files' owner.

The potential for security abuse is higher for programs in which the user ID is set to `root` or the group ID is set to any group that provides write access to system-level files. Do not write a program that sets the user ID to `root` unless there is no other way to accomplish the task.

The `chown` system call automatically removes any SUID or SGID bits on a file, unless the RUID of the executing process is set to zero. This prevents the accidental creation of SUID or SGID programs owned by the `root` account. For more information, see `chown`(2).

The following list provides suggestions for creating more secure SUID and SGID programs:

- Verify all user-provided pathnames with the `access` system call.
- Trap all relevant signals to prevent core dumps.
- Test for all error conditions, such as system call return values and buffer overflow.

When possible, create SGID programs rather than SUID programs. One reason is that file access is generally more restrictive for a group than for a user. If your SGID program is compromised, the restrictive file access reduces the range of actions available to the attacker.

Another reason is that it is easier to access files owned by the user executing the SGID program. When a user executes an SUID program, the original effective UID is no longer available for use for file access. However, when a user executes an SGID program, the user's primary GID is still available as part of the group access list. Therefore, the SGID process still has group access to the files that the user could access.

The stack of all SUID programs is not executable by default. User applications that rely on the stack being executable will fail. If absolutely necessary, you can change the default setting. This allows the stack of SUID programs to be executable. To change from default of zero (not executable) to executable, use the following command:

```
# sysconfig -r proc executable_stack=1
```

To ensure that the change persists across reboots, use the `sysconfigdb` command to add the entry to the `/etc/sysconfigtab` file.

2.2 Handling Errors

Most system calls and library routines return an integer return code, which indicates the success or failure of the call. Always check the return code to make sure that a routine succeeded. If the call fails, test the global variable `errno` to find out why it failed.

The `errno` variable is set when an error occurs in a system call. You can use this value to obtain a more detailed description of the error condition. You can use this information to determine how your program will respond or to produce helpful diagnostic messages. This error code corresponds to an error name in `<errno.h>`. For more information, see `errno`(2).

The following `errno` values indicate a possible security breach:

EPERM	Indicates an attempt by someone other than the owner to modify a file in a way reserved to the file owner or superuser. It can also mean that a user attempted to do something that is reserved for a superuser.
EACCES	Indicates an attempt to access a file for which the user does not have permission.

EROFS	Indicates an attempt to access a file on a mounted file system when that permission has been revoked.

If your program makes a privileged system call but the resulting executable program does not have superuser privilege, it will fail when it tries to execute the privileged system call. If the security administrator has set up the audit system to log failed attempts to execute privileged system calls, the failure will be audited.

If your program detects a possible security breach, do not have it display a diagnostic message that could help an attacker defeat the program. For instance, do not display a message that indicates the program is about to exit because the attacker's real user ID (UID) did not match a UID in an access file, or even worse, provide the name of the access file. Restrict this information by using the `audgen()` routine for SUID root programs and using `syslog` for other programs. In addition, you could program a small delay before issuing a message to prevent programmed attempts to penetrate your program by systematically trying various inputs.

2.3 Protecting Files

If your program uses any permanent files (for example, a database), make sure these files have restrictive permissions and that your program provides controlled access. These precautions also apply to shared memory segments, semaphores, and interprocess communication mechanisms; set restrictive permissions on all of these objects.

Programs sometimes create temporary files to store data while the program is running. Follow these precautions when you use temporary files:

- Be sure your program deletes temporary files before it exits.
- Avoid storing sensitive information in temporary files, unless the information has been encrypted.
- Give only the owner of the temporary file read and write permission. Set the file creation mask to 077 by using the `umask()` system call at the beginning of the program.
- Create temporary files in private directories that are writable only by the owner or in `/tmp`. The `/tmp` directory has the sticky bit set (mode 1777), so that files in it can be deleted only by the file owner, the owner of the directory, or the superuser.

A common practice is to create a temporary file, then unlink the file while it is still open. This limits access to any processes that had the file open before the unlink; when the processes exit, the inode is released.

Note that this use of `unlink` on an NFS-mounted file system takes a slightly different action. The client kernel renames the file and the unlink is sent to NFS only when the process exits. You cannot guarantee that the file will be inaccessible to someone else, but you can be reasonably sure that the file will be inaccessible when the process exits. In any case, always explicitly ensure that no temporary files remain after the process exits.

2.4 Specifying a Secure Search Path

If you use the `popen`, `system`, or `exec*p` routines, which execute `/bin/sh` or `/sbin/sh`, be careful when specifying a pathname or defining the shell `PATH` variable. The `PATH` variable is a security-sensitive variable because it specifies the search path for executing commands and scripts on your system. For more information, see `environ`(5), `popen`(3), and `system`(3).

The following list describes how to create a secure search path:

- Specify absolute pathnames for the `PATH` variable.
- Do not include public or temporary directories, other users' directories, or the current working directory in your search path. Including these directories increases the possibility of inadvertently executing the wrong program or of being trapped by a malicious program.
- Be sure that system directories appear before user directories in the list. This prevents you from mistakenly executing a program that might have the same name as a system program.
- Analyze your path-list syntax, especially your use of nulls, decimal points, and colons. A null entry or decimal point entry in a path list specifies the current working directory and a colon is used to separate entries in the path list. For this reason, the first entry following an equal sign should never begin with a colon.
- If a path list ends with a colon, certain shells and `exec*p` routines search the current working directory last. To avoid having various shells interpret this trailing colon in different ways, use the decimal point rather than a null entry to reference the current working directory.

You might want to use the `execve` system call rather than any of the `exec*p` routines because `execve` requires that you specify the pathname. For more information, see `execve`(2).

2.5 Responding to Signals

The Tru64 UNIX operating system generates signals in response to certain events. The event could be initiated by a user at a terminal (such as quit, interrupt, or stop), by a program error (such as a bus error), or by another program (such as kill).

By default, most signals terminate the receiving process; however, some signals only stop the receiving process. Many signals, such as SIGQUIT or SIGTRAP, write the core image to a file for debugging purposes. A core image file might contain sensitive information, such as passwords.

To protect sensitive information in core image files and protect programs from being interrupted by input from the keyboard, write programs that capture signals such as SIGQUIT, SIGTRAP, or SIGTSTP.

Use the `signal` routine to cause your process to change its response to a signal. This routine enables a process to ignore a signal or call a subroutine when the signal is delivered. (The SIGKILL and SIGSTOP signals cannot be caught, ignored, or blocked. They are always passed to the receiving process.) For more information, see `signal`(2) and `sigvec`(2).

Also, be aware that child processes inherit the signal mask that the parent process sets before calling `fork`. The `execve` system call resets all caught signals to the default action; ignored signals remain ignored. Therefore, be sure that processes handle signals appropriately before you call `fork` or `execve`. For more information, see the `fork`(2) and `execve`(2) reference pages.

2.6 Using Open File Descriptors with Child Processes

A child process can inherit all the open file descriptors of its parent process and therefore can have the same type of access to files. This relationship creates a security concern.

For example, suppose you write a set user ID (SUID) program that does the following:

- Allows users to write data to a sensitive, privileged file
- Creates a child process that runs in a nonprivileged state

Because the parent SUID process opens a file for writing, the child (or any user running the child process) can write to that sensitive file.

To protect sensitive, privileged files from users of a child process, close all file descriptors that are not needed by the child process before the child is created. An efficient way to close file descriptors before creating a child process is to use the `fcntl` system call. You can use this call to set the `close-on-exec` flag on the file after you open it. File descriptors that have this flag set are automatically closed when the process starts a new program with the `exec` system call.

For more information, see the `fcntl`(2) reference page.

2.7 Security Concerns in the X Environment

The following sections discuss several ways to increase security in the X programming environment:

- Restrict access control
- Protect keyboard input
- Block keyboard and mouse events
- Protect device-related events

2.7.1 Protect Keyboard Input

Users logged into hosts listed in the access control list can call the `XGrabKeyboard` function to take control of the keyboard. When a client has called this function, the X server directs all keyboard events only to that client. Using this call, an attacker could grab the input stream from a window and direct it to another window. The attacker could return simulated keystrokes to the window to fool the user running the window. Thus, the user might not realize that anything was wrong.

The ability of an attacker to capture a user’s keystrokes threatens the confidentiality of the data stored on the workstation.

The X Windows System provides a secure keyboard mode that directs everything a user types at the workstation keyboard to a single, secure window. Users can set this mode by selecting the Secure Keyboard item from the Commands menu in an X window.

Include a secure keyboard mode in programs that deal with sensitive data. This precaution is especially important if your program prompts a user for a password.

Some guidelines for implementing secure keyboard mode follow:

- Use the `XGrabKeyboard` call to the `Xlib` library.
- Use a visual cue to let the user know that secure keyboard mode has been set; for example, reverse video on the screen.
- Use the `XUngrabKeyboard` function to release the keyboard grab when the user reduces the window to an icon. Releasing the keyboard frees the user to direct keystrokes to another window.

2.7.2 Block Keyboard and Mouse Events

Hosts listed in the access control list can send events to any window if they know its ID. The `XSendEvent` call enables the calling application to send keyboard or mouse events to the specified window. An attacker could use this call to send potentially destructive data to a window. For example, this data could execute the `rm -rf *` command or use a text editor to change the contents of a sensitive file. If the terminal was idle, a user might not notice these commands being executed.

The ability of an attacker to send potentially destructive data to a workstation window threatens the integrity of the data stored on the workstation.

The X Windows System blocks keyboard and mouse events sent from another client if the `allowSendEvents` resource is set to `False` in the `.Xdefaults` file.

You can write programs that block events sent from other clients. The `XSendEvent` call sends an event to the specified window and sets the `send_event` flag in the event structure to `True`. Test this flag for each keyboard and mouse event that your program accepts. If the flag is set to `False,` the event was initiated by the keyboard and is safe to accept.

2.7.3 Protect Device-Related Events

Device-related events, such as keyboard and mouse events, propagate upward from the source window to ancestor windows until one of the following conditions is met:

- An X client selects the event for a window by setting its event mask
- An X client rejects the event by including that event in the `do-not-propagate` mask

You can use the `XReparentWindow` function to change the parent of a window. This call changes a window's parent to another window on the same screen. All you need to know to change a window's parent is the window ID. With the window ID of the child, you can discover the window ID of its parent.

The misuse of the `XReparentWindow` call can threaten security in a windowing system. The new parent window can select any event that the child window does not select.

Take these precautions to protect against this type of abuse:

- Have the child window select the device events that it needs. This precaution prevents the new parent from intercepting events that propagated upward from the child. Parent windows that centralize event handling for child windows are at greater security risk. An attacker can change the parent and intercept the events intended for the children. Therefore, it is safer for each child window to handle its own device events. Events that the child explicitly selects never propagate.

- Have the child window specify that device events will not propagate further in the window hierarchy by setting the `do-not-propagate` mask. This precaution prevents any device event from propagating to the parent window, regardless of whether the child requested the event.
- Have the child window ask to be notified when its parent window is changed by setting the `StructureNotify` or `SubstructureNotify` bit in the child window's event mask. For information on setting these event masks, see the *X Window System: The Complete Reference to Xlib, X Protocol, ICCCM, XLFD*.

2.8 Protecting Shell Scripts

When you write a shell script that handles sensitive data, set and export the `PATH` variable before writing the body of the script. Do not make shell scripts SUID or SGID.

3 Authentication Database

This chapter contains the following information:

- Authentication database overview
- Authentication database components
- Accessing the authentication database

3.1 Authentication Database Overview

The authentication database is a set of databases that store all Tru64 UNIX security information when enhanced security is enabled. The following databases comprise the authentication database:

- Device assignment
- File control
- System default
- Protected password
- Terminal control

The trusted programs (that is, any program that could subvert a security rule) you create specifically for systems with enhanced security enabled need to use the information in these databases. Except for a few specialized cases, system administrators maintain these databases using the Tru64 UNIX administrative interfaces. Therefore your programs usually only read them.

The following sections describe the authentication databases. The `authcap`(4) reference page contains general information on the file format.

3.1.1 Device Assignment Database (devassign)

The device assignment database contains device attributes for devices on the system. There are two kinds of devices included in the `devassign` database:

- Terminals
- X displays

The name of a device entry is used in the device-related commands. This name is independent of the names of the device files that represent the device.

System administrators maintain the device assignment database; your programs should not modify its contents.

The logical entries for this database have dynamic sizes (are not self-contained). For this reason, you must use the `copyesdvent()` routine to make a working copy of a structure that contains one of its entries. See the `getesdvent`(3) reference page for details.

The text file `/etc/auth/system/devassign` holds the entire device assignment database.

3.1.2 File Control Database

The file control database helps to assure that your security-sensitive files have the correct protection attributes (owner, mode bits, and so forth). It contains the absolute pathname and the correct attributes for each file (or directory). These attributes include any combination of the following:

- File type (regular, block special, character special, directory, fifo, socket)
- Owner
- Group
- Permission mode bits
- Access control list (if the system is configured for access control lists)

Your programs should not read from or write to the file control database other than to use its entries for newly created files through the `create_file_securely()` interface. However, you should add all new security-sensitive files and directories to the database. Include all of the attributes that do not change. This ensures that these attributes are regularly checked and corrected.

You can use the `create_file_securely()` routine to create files with the attributes specified in the file control database. This routine can only be used to create a new file. You should create new versions of files in a different file. (The Tru64 UNIX convention is to append a `:t` to a pathname for the file's new contents.) Then rename the new file (using the `rename`() system call) to the existing file name.

The file control database is a text file: `/etc/auth/system/files`. See the `files`(4) reference page for a definition of the format of this file. The system administrator can use the `edauth –df` command to add or remove entries from this database. See the `edauth`(8) reference page for more information.

3.1.3 System Default Database

The system default database, `/etc/auth/system/default`, is a text file that contains fields that are to be used when the corresponding fields are left undefined in other databases. Specifically, this database contains default information for the enhanced (protected) password, device assignment, and terminal control databases. (Note that all fields in each of the authentication databases may be left undefined, but all fields do not have system default values.)

The system default database also contains fields for miscellaneous system parameters. Your programs should not need this miscellaneous information.

System administrators maintain this database and your programs should never have to modify it. The access routines for other databases also return the system default values. See Section 3.1.5, for an example of how to access and use the information in the system default database.

The entire system default database has only one entry, the `default` entry.

3.1.4 Enhanced (Protected) Password Database

The enhanced password database (`/tcb/files/auth.db` and `/var/tcb/files/auth.db`) are `dbm` files that hold a set of user authentication profiles. User authentication profiles can also be distributed between Tru64 UNIX systems using the NIS `prpasswd` map. Each authentication profile entry is named with a user name (a name that a user supplies during login). The authentication profile has many fields that govern the user's login session. Chapter 4 describes these fields in detail.

An authentication profile is associated with the account whose presence is indicated by a line in the traditional `/etc/passwd` file or NIS `passwd` map. The encrypted password has been moved from the `/etc/passwd` file to the authentication profile.

The system assigns the traditional meanings for the other fields in the `/etc/passwd` database. Each entry in `/etc/passwd` corresponds to exactly one authentication profile in the protected password database with the same user ID and name. (Both must be present for an account to be considered valid.) The `/etc/passwd` entry contains a dummy encrypted password field; the authentication profile holds the real one.

The traditional UNIX interfaces for querying the `/etc/passwd` file is `getpwent()`. The interfaces' functions are unchanged and always fetch their information from the `/etc/passwd` file or NIS map. Note however, that the encrypted password that is returned is a dummy value. (The routine is not modified to retrieve the encrypted password from the authentication profile.)

Your programs should not modify the enhanced (protected) password database. However, many trusted programs need to read the information from the user authentication profiles.

3.1.5 Terminal Control Database

The terminal control database, `/etc/auth/system/ttys.db`, is a `dbm` file that contains fields used primarily during login that apply to the login terminal, as opposed to the user who is logging in. This database consists of an entry for each terminal upon which users may log in including X terminals.

Each entry in the database has a name of the terminal that matches a name in the file used to specify login ports (`/etc/inittab`). The entries in the device assignment database correspond to each entry in the terminal control database. Most trusted programs (for example, `login`) do not provide their services if there is no corresponding entry in the device assignment database.

Each terminal control database entry contains the following fields:

- The name of the terminal.
- The user ID and time of the last unsuccessful login attempt. Because the user ID is stored in the database, an unsuccessful login attempt that specifies a user name that does not map to a user ID does not produce a valid user ID in this database. If the user name maps to a valid ID, that ID is placed in the appropriate field.
- The user ID and time of the last successful login.
- The number of unsuccessful login attempts since the last successful login.
- Whether the terminal is locked.
- The number of unsuccessful attempts that the system allows before locking the terminal.
- The enforced time delay after a failed login attempt (enforced by the login program).
- The number of seconds after which the login, once started, times out if there is no keyboard input. Upon timeout, the login program terminates the login attempt.

System administrators maintain the entries in the terminal control database, although the Tru64 UNIX login programs modify many fields. Your programs do not usually modify this database. Although it is unlikely, trusted programs may need to read this database.

The file `/etc/auth/system/ttys.db` holds the entire terminal control database.

3.2 Authentication Database Components

Each database consists of a set of named entries. Programs primarily use the name of the entry to request a specific entry from a database, although a program can also sequentially search through the entries in a database.

Each entry contains a set of fields. Each field has an identifier, used to access the field and a value. Each database has an allowed set of fields in one of its entries. Individual fields are optional and can be omitted from an entry. There are several types of fields including string, integer, and Boolean.

The general format for an entry is as follows:

```
entry_name:string_field=value:integer_field#value:\
                        :boolean_field_true:boolean_field_false@:chkent:
```

In general, library routines read or write an entry as a whole. A C structure holds all possible fields for a given entry of the database. This structure is always accompanied by a flags structure which holds a mask designating which fields are to be read or written.

Your programs should take appropriate action when a field is undefined. In many cases, the undefined fields should be fetched from the system defaults database, as described in Section 3.2.1. Structures for each database include system default fields and flags for that database. Thus, it is easy to retrieve the system default values associated with a particular field because the system default values are available from the same structure that stores values for the individual entry.

3.2.1 Database Form

In general, you will not have to deal with the physical format of the authentication databases. All databases have the same logical form and similar access libraries. For example, the terminal control database consists of an entry for each controlled terminal. The following `ttys` file sample physical format entry for `tty01` and the associated table illustrate the database file format.

```
tty01:t_devname=tty01:t_uid#44:t_logtime#772479074:\
 :t_login_timeout#20:t_failures#3:t_lock@:\
 :chkent:
```

Meaning	Field	Value	Description
Name	t_devname	tty01	Terminal 1
User of last login	t_uid	44	UID of 44
Time of last login	t_logtime	772479074	Fri Jun 24 13:31:13 EDT 1994

Meaning	Field	Value	Description
Login timeout	t_login_timeout#20	20	Login timeout in seconds
Attempts since last login	t_failures#3	3	Failed login attempts since last successful login
Account status	t_lock	@	Unlocked (false)
Check entry	:chkent:<EOL>	chkent	End of entry

The following C structure is used for fetching an entry from the `ttys` database (see the include file `<prot.h>`):

```
struct es_term  {
    struct estc_field  *ufld;     /* fields for this entry */
    struct estc_flag   *uflg;     /* flags for this entry */
    struct estc_field  *sfld;     /* system default fields */
    struct estc_flag   *sflg;     /* system default flags */
};
```

The `estc_field` holds the data for the fields of the entry, and `estc_flag` holds the flags that designate which fields in `estc_field` are present or are set. The following is the `estc_field` structure:

```
struct estc_field  {
    char    *fd_devname;      /* terminal name */
    uid_t   fd_uid;           /* uid of last successful login */
    time_t  fd_slogin;        /* time of last successful login*/
    ushort  fd_nlogins;       /* consecutive failed attempts */
    char    fd_lock;          /* terminal lock status */
    ushort  fd_login_timeout; /* login timeout value */
};
```

```
struct estc_flag  {
unsigned short
    fg_devname        :1,     /* name present? */
    fg_uid            :1,     /* uid present? */
    fg_slogin         :1,     /* time present? */
    fg_nlogins        :1,     /* failed attempts present? */
    fg_lock           :1,     /* lock status present? */
    fg_login_timeout  :1      /* login timeout present? */


};
```

The `getestcent`(3) reference page defines the library routines that you can use to access the terminal control database. The access routines return or set the fields for a specific entry `ufld` and `uflg` and for the system defaults (`sfld` and `sflg`). For each database whose fields have system defaults, the system defaults are returned in addition to the fields for that entry.

3.2.2 Reading and Writing a Database

Each database is owned by a user/group, to which your program must have discretionary access. Your program can be installed in two ways:

- SGID to the appropriate group as a standard program of the subsystem
- SUID 0 as a standard program of the subsystem

The library routines automatically enforce one database writer at a time. However, the database is locked only for the duration of the time the database is being rewritten. There is no way to lock an entry against access across a retrieval and write operation. You should complete your writes as quickly as possible.

3.2.2.1 Buffer Management

You must understand how the system allocates and returns buffers for database entries to properly code programs that retrieve, replace, and add database entries. All database routines are patterned after the `getpwent()` routines in that they return pointers to static storage that is reused on each call. You must save the buffer contents if you are going to retrieve another entry and need to refer again to the previous entry, or if you need to rewrite an existing entry or add a new entry. You cannot read a database entry, change one or more field and flag values, and submit the same buffer to the routine that modifies the database.

The logical form for some database entry fields is self-contained. Other fields contain pointers to variable length data.

The `devassign` database logical form contains some fields that are pointers to variable length data. The `getesdvent`(3) reference page describes the `copyesdvent()` routine that allocates a structure to hold a device assignment database entry and copies the contents of a buffer returned from `getesdvent()` or `getesdvnam()` into it.

You can save an entry for a self-contained database by simple structure assignment, as follows:

```
struct es_passwd *pr;       /* returned value */
struct es_passwd *pwcopy;   /* buffer for saved values */

/* Retrieve john’s protected password database entry */

pr = getesnam("john");

/* store values of john’s entry to a local buffer */

pwcopy = copyespwent(pr);
if (!pwcopy) abort();
```

```
/* Change the password minimum change time to two weeks */

pwcopy->uflg->fg_min = 1;
pwcopy->ufld->fd_min = 14 * 24 * 60 * 60;

/* Rewrite john's protected password database entry */

if (!putespwnam("john", pwcopy))
    errmsg("Could not write protected password entry\n");
free(pwcopy);
```

3.2.2.2 Reading an Entry by Name or ID

You can read database entries by specifying their name or, in some databases, some other identifying value. For example, you can fetch entries from the enhanced (protected) password database by the entry name (the user's name) or the user ID. The following code reads the entry associated with the name `tty44` from the terminal control database:

```
  .
  .
  .
struct es_term   *entry;
  .
  .
  .
if ((entry = getestcnam("tty44")) == NULL)
            errmsg ("Entry not found");
```

Note that `getestcnam()` allocates the data structure for the returned entry. Hence, `entry` is only a pointer to an `es_term` structure that is reused the next time any of the `prtc()` or `estc()` routines is called.

3.2.2.3 Reading Entries Sequentially

You can also read database entries sequentially as illustrated in the following code:

```
  .
  .
  .
struct es_term   *entry;
  .
  .
  .
setprtcent();                            /* rewind the database*/
while ((entry = getestcent()) != NULL){  /* read next entry   */
  .                                      /* process the entry */
  .
  .
}
endprtcent ();                          /* close */
```

Note that `getestcent()` also allocates the data structure for the entry. You can restart the search from the beginning using `setprtcent()`.

3.2.2.4 Using System Defaults

A system default is a field that is used when the corresponding field in an entry is not defined. The system default database contains defaults for the other databases. The following databases contain information for which there are system defaults:

- Protected password
- Terminal control
- Device assignment

Note that only certain fields in these databases are allowed to have defaults.

When your program reads a logical entry, the library routine returns both the fields for that entry (`ufld` and `uflg`) and for the system default (`sfld` and `sflg`). If the entry does not contain the field you need, use the system default. In some cases if the system default is also undefined, your program should generate audit data to report the error and execute a failure path. In other cases, you can safely define a default value.

For example, if you need to determine the timeout value for the terminal `tty14`, your code might look like this:

```
struct es_term  *entry;        /* the entry for the terminal */
ushort          time_out;      /* final timeout value */
   .
   .
   .

/*--- fetch the entry by name ---*/

if ((entry = getestcnam ("tty14")) == NULL)
      errmsg ("Entry not found");

/*--- if defined for the terminal, use it ---*/

if (entry->uflg->fg_login_timeout)
      time_out = entry->ufld->fd_login_timeout;

/*--- else if system default is defined, use it ---*/

else if (entry->sflg->fg_login_timeout)
      time_out = entry->sfld->fd_login_timeout;
```

```
/*--- otherwise, assume a value of 0 ---*/

else time_out = 0;
```

3.2.2.5 Writing an Entry

Your program should seldom have to modify a database, and even more rarely a system default. However, if this is necessary, place the new fields in `ufld` and set the corresponding flags in `uflg`, and then call the appropriate library routine. For example, to set a new timeout value for the terminal `tty14` to 20, your code might look like this:

```
struct es_term  *entry, *ecopy;
 .
 .
 .

/*--- fetch the entry by name ---*/

if ((entry = getestcnam("tty14")) == NULL)
      errmsg ("Entry not found");

/*--- change the desired field(s) ---*/

ecopy = copyestcent(entry); if (! ecopy) abort():
ecopy->ufld->fd_login_timeout = 20;  /* set timeout value */
ecopy->uflg->fg_login_timeout =  1;  /* set flag to show the
                                          field has been set */
/*--- update the database ---*/

if (!putestcnam("tty14", ecopy))
      errmsg ("Could not update database");
free(ecopy);
```

Note

You must call the appropriate `copyes*()` routine to save the data for later use.

The `copyes*()` routines return pointers to a `malloc()` storage area that the caller must clear.

You can only set system defaults using the `putesdfnam`() interface for the system default database. You cannot, for example, set the `sfld` and `sflg` fields in an `es_term` entry and then call `putestcnam`() to set system defaults.

3.3 Accessing the Authentication Databases

Tru64 UNIX includes a set of library routines to access each database. The following reference pages describe the form and use of these databases; you should read them with this chapter.

Subject	Database	Reference Page
Device Assignment	`devassign`	`getesdvent`(3)
File Control	`file`	`getesfient`(3)
System Default	`default`	`getesdfent`(3)
Protected Password	`auth.db/prpasswd (NIS)`	`getespwent`(3)
Terminal Control	`ttys.db`	`getestcent`(3)

The library routines defined on these reference pages hide the actual file format of the databases. Trusted programs do not need to know the format; they simply use these library routines.

4 Identification and Authentication

This chapter contains the following information:

- The audit ID
- Identity support libraries
- Using daemons
- Using the enhanced (protected) password database

4.1 The Audit ID

Tru64 UNIX preserves all traditional process user and group identities. Additionally, it provides the per-process audit ID (AUID), which is unique to Tru64 UNIX. The AUID is similar in principle to the real user ID, except that it remains unchanged even in cases where the real user ID changes.

The audit ID is associated with all audit records and establishes the user identity even in those cases where the real and effective user IDs have been changed from their values at login.

The audit ID can be set only once in a line of process descendants, regardless of any process privileges. The audit ID is set at login to the authenticated user (the same as the real and effective user IDs) and is inherited from parent to child when a process forks using the `fork()` system call.

Programs that are created from startup scripts or that are created as a result of *respawn* entries in the `inittab` file are created with an unset audit ID. Such programs are normally authentication programs (`getty/login` sequences, window managers, trusted path managers) that set the AUID based on the user that authenticates through that interface.

Programs started through startup scripts typically receive requests for service on behalf of users and spawn a process to service that request. Such programs typically set the audit ID in the child service process based on the requesting process's effective identity. If you are writing this type of program, you should use the SIA routines. The SIA routines properly set up the user's environment in the child process regardless of the security mechanisms in use on the system (BASE, enhanced, DCE, and so forth).

The `getluid`() and `setluid`() system calls read and set the audit ID. See their reference pages for details.

4.2 Identity Support Libraries

The Tru64 UNIX operating system provides several library routines for managing user and group identities. For example, the `set_auth_parameters()` routine is required by some routines used by enhanced security. It stores the initial user and group IDs that can later be queried or tested by the other routines. If you are writing a program or routine that will be used with the enhanced security option, you must call `set_auth_parameters()` at the beginning of your program's `main()` routine.

Several of the enhanced security routines for querying the authentication database require the program to have previously called `set_auth_parameters()` before changing any of the user or group IDs, or the command arguments *`argc`* and *`argv`*.

See the `identity(3)` reference page for more information.

To keep your code portable between security mechanisms, use the SIA session routines.

4.3 Using Daemons

Whenever a daemon performs an operation at the request of a user program (the client), it acts in one of two ways:

- It can run under its own identities, authorizations, and privileges, making its own decisions about what actions the requesting program may or may not perform. In this case, it does not need to change any of its own user identities.
- It can have the underlying operating system enforce operations as if the daemon had the client's security attributes (user IDs, authorizations, and so forth).

In the latter case, the daemon needs to establish a set of security attributes. The preferred technique is to fork a process, set the identities and privileges using SIA, and then either perform the actions directly or execute a program to perform them.

4.4 Using the Enhanced (Protected) Password Database

Although the enhanced (protected) password database is intended mainly for Tru64 UNIX programs, your programs may need to use the fields described in the following list. (These fields are also described in the `getespwent(3)` and `prpasswd(4)` reference pages, the `prot.h` include file, and the administrative part of this document.)

- User name (`u_name`) and ID (`u_id`) — These fields correspond to the user name and ID in `/etc/passwd`.

- Encrypted password (`u_pwd`) — This field is the real encrypted password.
- Retired status (`u_retired`) — This field indicates whether the authentication profile is valid. If not valid, login sessions are not allowed. Once retired, an account should never again be reused.
- Login session priority (`u_priority`) — The process priority assigned to programs of the user login session using `setpriority()`.
- User audit mask (`u_auditmask`) and control flags (`u_audcntl`) — This mask and its control flags, with the system audit mask, designate the events audited during the login session. The `login` program assigns a mask to the user's login shell. Audit masks and the control flags are inherited across `exec()` and `fork()` calls. See `auditmask`(8) for more information.
- Password parameters — The following parameters describe the login password and its generation:
  - Maximum length in characters for passwords chosen by the user (`u_maxchosen`)
  - Password expiration interval (`u_exp`)
  - Minimum password lifetime (`u_minchg`)
  - Password lifetime (`u_life`)
  - Time and date of last successful password change (`u_succhg`)
  - Time and date of last unsuccessful password change attempt (`u_unsucchg`)
  - User who last changed the password (`u_pwchanger`)
  - Password generation parameters (`u_genpwd`)
  - User generated password generation parameters (`u_pickpw`)
- Login password requirements (`u_nullpw`) — This is sometimes called the "null password option" and controls attempts to set a null password. Most administrators do not allow this option.
- Times during which a user may login (`u_tod`) — This field is formated like the UUCP `systems` file. (The `systems` file describes when a remote system can be contacted for file transfer.) It determines the valid times for a user to log in.
- Time and date of last login (`u_suclog`) — Expressed as a canonical UNIX time (in seconds since 1970).
- Terminal used during last login (`u_suctty`) — The terminal name is a cross-reference to the device assignment and terminal control databases.

- Number of unsuccessful login attempts since last login (`u_numunsuclog`) — This value is used to compute whether the terminal is disabled due to too many unsuccessful attempts.
- Number of unsuccessful login attempts allowed before disabling (`u_maxtries`) — This value is the user-specific limit for the number of unsuccessful attempts allowed until the account is disabled.
- Lock status (`u_lock`) — Whether or not the administrator has locked the account. A locked profile cannot be used for login or other services. Only an explicit request from the system administrator should unlock an authentication profile, and only programs that handle such requests should reset the locked field. A common programming error is to assume that the lock indicates all lock conditions. This indicator shows only the status of the administrative lock. An account may appear to be locked due to being disabled by password lifetime expiration or exceeding the number of unsuccessful attempts allowed for the account.

Your program can assume that with enhanced security enabled, the user name and ID in the enhanced (protected) password database is maintained by the system to have a corresponding entry in the `/etc/passwd` file.

4.4.1 Example: Password Expiration Program

The program named `myexpire` in Example 4–1 is a program for use with enhanced security that prints the user's password expiration time as defined in the enhanced (protected) password database. This program is part of the authentication protected subsystem and runs in the set group ID (SGID) mode, setting the GID to `auth`.

Example 4–1: Password Expiration Program

```
#include <sys/types.h>
#include <stdio.h>
#include <sys/security.h>
#include <prot.h>

main (argc, argv)
int        argc;
char      *argv[];
{
   struct es_passwd  *acct;
   time_t expire_time;
   time_t expire_date;

   /*--- Standard initialization ---*/

   set_auth_parameters(argc, argv);
   initprivs();
```

Example 4–1: Password Expiration Program (cont.)

```
   /*--- fetch account information using audit ID ---*/

   if ((acct = getespwuid(getluid())) == NULL)
      errmsg("Internal error");

  /*-- test if personal or system default applies and print --*/

   if (acct->uflg->fg_expire)
      expire_time = acct->ufld->fd_expire;
   else if (acct->sflg->fg_expire)
      expire_time = acct->sfld->fd_expire;
   else {
      audit_db_error(acct);     /* audit (externally defined) */
      errmsg("No user-specific or system default \
                                          expiration time.");
   }

   if (!acct->ufld->fg_schange) {
      audit_db_error(acct);      /* audit (externally defined) */
      errmsg("Account does not have successful change time");
   }

   expire_date = acct->ufld->fd_schange + expire_time;

   if (acct->uflg->fg_psw_chg_reqd && \
                 acct->ufld->fd_psw_chg_reqd) \
                          expire_date = time((time_t *) NULL);

   audit_action(acct->ufld->fd_name, expire_date);
   exit(0);
}
```

Note

The enhanced (protected) password database files are accessible only to processes in the `auth` group. Programs that need to read the enhanced password database files must set the group ID to `auth`. (See the `setgid`(2) reference page.) To write this information you must set the UID to 0 or to a user ID and have a group ID of `auth`.

5 Audit Record Generation

This chapter contains the following information:

- Audit record overview
- Audit events
- Audit records and tokens
- Audit flag and masks
- Disabling auditing for the current process
- Modifying auditing for the current process
- Application-specific audit records
- Site-defined events
- Creating your own audit logs
- Parsing audit logs

5.1 Audit Record Overview

Trusted programs often generate their own audit records because:

- Auditing at the application level avoids generating a large amount of system-level audit records.
- When examining an audit report, it is difficult to deduce a user's intention solely from system-level audit records.

Trusted programs can use the `audgen()` system call, the `audgenl()` library routine, or the `audgen` command to generate audit records; `audgenl()` is a front-end to `audgen()`. For arguments, the program supplies an audit event followed by audit data consisting of audit tokens and values.

The following code fragment shows how a program that checks boot authentication can call `audgenl()` to audit authentication failure:

```
if(audgenl(AUTH_EVENT,  1
           AUD_T_LOGIN, pr->ufld.fd_name,  2
           AUD_T_UID, pr->ufld.fd_uid,
           AUD_T_CHARP, "boot authentication failed"),0)== -1)
perror("audgenl");
```

Notes:

1 `AUTH_EVENT` is the record event name.

2 `AUD_T_LOGIN`, `AUD_T_UID`, and `AUD_T_CHARP` are tokens, each with a corresponding value.

These identifiers are defined in `<sys/audit.h>`. See Section 5.2 and Section 5.3 for descriptions of events and tokens.

5.2 Audit Events

Each audit record has an audit event associated with it. The system automatically adds the event when generating system call audit records. Self-auditing application programs pass the event as an argument to `audgen()` or `audgenl()` when generating audit records. There are two types of audit events available to application programs:

- Trusted events, which are defined in `<sys/audit.h>` with values between `MIN_TRUSTED_EVENT` and `(MIN_TRUSTED_EVENT + N_TRUSTED_EVENTS -1)`. For example, the `LOGIN` event.
- Site-defined events, which are defined in `/etc/sec/audit_events` with values between `MIN_SITE_EVENT` and 1048576. The default range for site-defined events is 64. For information on defining site events, see Section 5.8.

5.3 Audit Records and Tokens

The audit subsystem has no fixed record type. Instead, an audit record is a series of `tuples` (data objects containing two or more components). Each tuple consists of an audit `token` and its corresponding `value`; depending on the token type, the tuple might contain a `length` field.

The following sections describe the two types of tokens: `public tokens` and `private tokens`. Application programs use public tokens.

5.3.1 Public Tokens

Public tokens are available to application programs that generate audit records using `audgen()` and `audgenl()`. Public tokens are defined in `<sys/audit.h>` and begin with `AUD_T_`; for example, `AUD_T_CHARP`.

There are three basic types of public tokens:

pointer	Used to represent data strings or structures as pointers. AUD_T_CHARP (character string) and AUD_T_HOMEDIR (home directory) are two pointer-type tokens.
iovec	Used to represent data as iovec-formatted data. AUD_T_OPAQUE, and AUD_T_INTARRAY are two iovec-type tokens. (Look for the iovec comments in <sys/audit.h>. The iovec structure is defined in <sys/uio.h>. For information about iovec, see the readv(2) and writev(2) reference pages.)
fixed length	Used to represent data as a 32- or 64-bit quantity. (AUD_T_RESULT and AUD_TP_LONG are 64–bit; others are 32-bit.) Most tokens use fixed-length data. AUD_T_AUID (audit ID), AUD_T_UID (user ID), and AUD_T_PID (process ID) are examples of fixed-length tokens.

The following example generates an audit record using iovec-formatted data:

```
#define AUD_COMPAT
#include <sys/audit.h>
#include <sys/uio.h>

main()
{
     char buf[100];
     int i;
     struct iovec iov;

     for (i = 0; i < sizeof(buf); i++)
          buf[i] = i;

     iov.iov_len = sizeof(buf);
     iov.iov_base = buf;

     if (audgenl (AUDGEN8,
                 AUD_T_CHARP, "opaque data test",
                 AUD_T_OPAQUE, &iov,
                 0 ) ==  -1 )
         perror ("audgenl");
}
```

5.3.2 Private Tokens

Private tokens are used by the kernel; they are not available to application programs. The `audgen()` system call rejects any attempts by application programs to write records that contain private tokens. Private tokens are defined in `<sys/audit.h>` and begin with `AUD_TP_`; for example `AUD_TP_AUID`.

The kernel uses the private tokens when creating audit records. For example, the kernel encapsulates each audit record with `AUD_TP_LENGTH` tuples whose value is the length of the audit record. Another example is the `audgen()` or `audgenl()` *`event`* argument, from which the kernel creates a `AUD_TP_EVENT` tuple.

5.4 Audit Flag and Masks

Whether an audit event actually results in the generation of an audit record depends on the following flag and mask settings:

- Process audit control flag
- Process audit mask
- System audit mask

The process audit control flag has four exclusive states:

AUDIT_OR	An audit record is generated if either the system audit mask or the process audit mask indicates such an event should be audited.
AUDIT_AND	An audit record is generated if both the system audit mask and the process audit mask indicate such an event should be audited.
AUDIT_OFF	No audit records are generated for the current process.
AUDIT_USR	An audit record is generated if the process audit mask indicates such an event should be audited.

The process audit control flag also has two nonexclusive states:

AUDIT_SYSCALL_OFF	Turns off system call record generation for the process.
AUDIT_HABITAT_USR	Turns on the habitat system calls in the user mask for the process even if system calls are turned off for the

system mask. The habitat system calls are: System V – `unlink()` and `open()`; real time – `memlk()`, `memunlk()`, `psx4_time_drift()`, and `rt_setprio()`. These habitat system calls are turned on or off as a group. See Appendix B for the habitat events.

The system administrator can establish a default audit level for users, while retaining the ability to audit any individual user at whatever level the administrator deems appropriate. (See Security Administration for information on configuring and administering the audit subsystem.)

From a programmer's perspective, a privileged process can set its audit level (specify what gets audited), either as an absolute mask or in relation to the system audit mask. See Section 5.6 for an example showing how to set a process's audit mask. See `audcntl`(2) and `auditmask`(8) for more information.

5.5 Disabling System-Call Auditing for the Current Process

Controlling which events are audited is an important step in fine-tuning the amount of audit data collected. System calls can generate large amounts of audit data, but this data is not necessarily useful information. In general, actively auditing the modification of fields in a security-relevant database or auditing a specific security-relevant action provides more usable information than trying to derive this information from a multitude of system-call audit records. For example, the login process executes thousands of system calls, but a single informative audit record written by the login process uses less system resources and is easier to understand.

Application programs can disable system-call auditing but still allow trusted-event auditing. The following code fragment shows how to use the `audcntl()` system call to set `AUDIT_SYSCALL_OFF`:

```
/* OR the AUDIT_SYSCALL_OFF bit into the audcntl flag */
if ((cntlflag = audcntl(GET_PROC_ACNTL,
                        NULL, 0, 0, 0, 0)) == -1)
    perror("audcntl");
else
    audcntl(SET_PROC_ACNTL, NULL, 0,
            cntlflag|AUDIT_SYSCALL_OFF, 0, 0);
```

5.6 Modifying System-Call Auditing for the Current Process

A process can control what is audited for itself or another process by modifying the target process's `auditmask` and `audcntl` flags. You can modify the current process's audit mask as follows:

```
/* ex. set the process's auditmask to audit only LOGIN
   events and successful setgroups calls
*/
#include <sys/audit.h>
#include <sys/syscall.h>
char buf[AUDIT_MASK_LEN];
⋮
bzero (buf, sizeof(buf));
A_PROCMASK_SET (buf, LOGIN, 1, 1);
A_PROCMASK_SET (buf, SYS_setgroups, 1, 0);
if (audcntl (SET_PROC_AMASK, buf,
             AUDIT_MASK_LEN, 0, 0, 0) == -1)
    perror ("audcntl");
```

The A_PROCMASK_SET macro, defined in `<sys/audit.h>`, takes the following arguments:

buf	The buffer containing the mask.
event name	The `<sys/audit.h>` header file contains trusted event names. The `<sys/*syscall.h>` header files contain system call names.
succeed	Indicates whether to audit success; a `1` means audit event success.
fail	Indicates whether to audit failure; a `1` means audit event failure.

See `audcntl`(2) for more information.

5.7 Application-Specific Audit Records

An application program provides application-specific audit data as arguments to `audgen()` or `audgenl()`.

The following code fragment sends an audit record to the kernel when the specified *event* occurs. The *event* is either a trusted event from `<sys/audit.h>` or a site-defined event from `/etc/sec/site_events`. (Whether the kernel actually writes an audit record to the audit log depends on the events audited for this process.)

```
/* If bad_thing occurs, generate an event of type event_num,
 * with string "bad thing happened", and a result of 66.
 */

#include <sys/audit.h>

if (bad_thing) {
   if (audgenl (event_num,
                AUD_T_CHARP, "bad thing happened",
                AUD_T_RESULT, 66, 0 ) == -1)
      perror ("audgenl");
}
```

In general, an application-generated audit record does not have to include data for the tokens listed in Table 5–1. The kernel automatically adds this information to each audit record. However, the audit subsystem does not prevent you from putting any of the public token tuples in an audit record; for example, you can add an `AUD_T_AUID` tuple to an audit record even though the system will later add an `AUD_TP_AUID` to the record. Both tuples are written to the audit log.

5.8 Site-Defined Events

A site can define its own set of audit events, called **site-defined events**, in the locally created and maintained file `/etc/sec/site_events`. The file contains one entry for each site event.

The potential range for site event numbers is `MIN_SITE_EVENT` (defined in `<sys/audit.h>`) to 1048576. The default range is 64. To change this value, set `audit-site-events` in `/etc/sysconfigtab` and reboot. For example, to allow for up to 128 site-defined events:

```
sec:
    audit-site-events=128
```

Each site-event entry can contain up to `INT_MAX` subevents. There is no default range defined for subevents.

The maximum length for an event or subevent name is `AUD_MAXEVENT_LEN`, defined in `<sys/audit.h>`.

Application programs can generate records containing both site-defined events and the trusted events defined in `<sys/audit.h>` (`MIN_TRUSTED_EVENT` to `MAX_TRUSTED_EVENT`).

The `auditmask` utility supports preselection for site-defined events, and the `audit_tool` utility supposts postselection for site-defined events and subevents.

5.8.1 Sample site_events File

The syntax for a site-defined audit event entry is:

[*event_name event_number* [, *subevent_name subevent_number* ...] ;]

The following entries in a sample /etc/sec/site_events file demonstrate how to create site-defined events and subevents:

```
essence 2048,     1
    ess_read 0,   2
    ess_write 1;  3
rdb 2049,
    rdb_open 0,
    rdb_close 1,
    rdb_read 2,
    rdb_write 3;
decinspect 2050;
```

Notes:

1. essence is the event; 2048 is the event number. Note that 2048 is MIN_SITE_EVENT, the lowest number available for site-defined events.
2. ess_read is the first subevent; 0 is the first subevent number.
3. ess_write is the second subevent; 1 is the second subevent number.

See aud_sitevent(3) for more information on site-defined events.

5.8.2 Example: Generating an Audit Record for a Site-Defined Audit Event

The following code fragment uses audgenl() to generate audit data for an rdb_close event:

```
int event_num, subevent_num;

/* translate event name(s) into event numbers */
if (aud_sitevent_num ("rdb", "rdb_close",
                      &event_num, &subevent_num ))
    printf ("aud_sitevent_num failed");

/* generate audit data */
else if (audgenl (event_num,
                  AUD_T_SUBEVENT, subevent_num,
                  AUD_T_CHARP, "Trusted RDB V1.0 Close",
                  0) == -1)
    perror ("audgenl");
```

You should include an AUD_T_CHARP, *event name* argument pair with audgenl() when generating a record for a site-defined event. Doing so simplifies the task of analyzing audit data on a system that does not have a copy of the local `site_events` file.

See `aud_sitevent`(3) and `audgenl`(3) for more information.

5.9 Creating Your Own Audit Logs

You can use the `audgen()` system call to create your own audit log. If the *`size`* argument to `audgen()` is a nonzero value, audit data is copied to the *`userbuff`* specified in `audgen()` rather than written to the system audit logs. A trusted application can then write the data in *`userbuff`* to a unique log file. See `audgen`(2) for more information.

You can use the `audit_tool` utility to read the new audit log. More detailed information can be read from the log using the information in Section 5.10.

5.10 Parsing an Audit Log

Most people use `audit_tool` or `dxaudit` to read audit logs. The `audit_tool` utility is a sophisticated program that converts audit data into useful information, formats output, and handles audit records that span audit log files. When `audit_tool` first reads an audit log, it creates a corresponding `.hdr` file to maintain state information. This state information reduces the time needed for subsequent reads of the audit logs. Also, if an audit record spans audit logs, `audit_tool` opens both log files and creates a complete record in the header file.

The following sections describe the format and construction of audit logs; they provide:

- A description of an audit log plus a list of the token types generally found in all audit records.
- The binary record format with examples showing an octal dump of a record and its formatted output.
- A table of token/tuple byte descriptions, which lists the data types and format for each public and private token.
- Sample macros for parsing tuples.

These sections do not provide the design information needed to create a program similar to `audit_tool`; they do provide the basic information required to parse an audit log into records and tuples.

5.10.1 Overview of Audit Log Format and List of Common Tuples

Audit logs are regular UNIX data files that contain audit records.
An audit record consists of a series of tuples whose format is either *`token:value`* or *`token:length:value`*. Each record starts and ends with

an AUD_TP_LENGTH tuple. (The audit_tool utility uses AUD_TP_LENGTH to determine whether an audit record is valid. If the actual length of the record does not match the AUD_TP_LENGTH value, audit_tool discards the record and provides a warning.) Table 5–1 shows the default tuples generally used for audit records.

Table 5–1: Default Tuples Common to Most Audit Records

Tuple	Comment	Tuple	Comment
AUD_TP_LENGTH		AUD_TP_VERSION	
AUD_TP_AUID		AUD_TP_RUID	
AUD_TP_HOSTADDR		AUD_TP_EVENTP	if habitat
AUD_TP_HABITAT	if habitat	AUD_TP_EVENT	
AUD_TP_UID		AUD_TP_PID	
AUD_TP_PPID		AUD_TP_DEV	if device is associated with a process
AUD_TP_NCPU		AUD_TP_TV_USEC	
AUD_TP_SET_UIDS	if uid change	AUD_TP_TID	if AUDIT_USR flag is set

5.10.2 Token/Tuple Byte Descriptions

Table 5–2 lists public and private tokens with their octal values. For each tuple, the third column lists the sequence in which tuple data is written to an audit log by the kernel. Token is a 1–byte quantity; length is a 4–byte quantity. Sample Parse Macro refers to the macro that audit_tool uses to parse the tuple. These macros are provided, for reference purposes only, in Section 5.10.3.

Table 5–2: Token/Tuple Byte Descriptions

Token	Octal Value	Tuple Format and Sample Parse Macro
AUD_T_CHARP	001	token:length:null-terminated ASCII string. PARSE_DEF3
AUD_T_SOCK	003	token:length:struct sockaddr (4.3 style (u_short); if family is > UCHAR_MAX, assume 4.4 style sockaddr of length (byte) then family (byte)). PARSE_DEF3
AUD_T_LOGIN	004	token:length:null-terminated ASCII string. PARSE_DEF3

Table 5–2: Token/Tuple Byte Descriptions (cont.)

Token	Octal Value	Tuple Format and Sample Parse Macro
AUD_T_HOMEDIR	005	token:length:null-terminated ASCII string. PARSE_DEF3
AUD_T_SHELL	006	token:length:null-terminated ASCII string. PARSE_DEF3
AUD_T_DEVNAME	007	token:length:null-terminated ASCII string. PARSE_DEF3
AUD_T_SERVICE	010	token:length:null-terminated ASCII string. (reserved for future use)
AUD_T_HOSTNAME	011	token:length:null-terminated ASCII string. PARSE_DEF3
AUD_T_INTP	012	token:length:int (First element is number of elements in array; note that AUD_T_INTARRAY is the preferred tuple when generating an audit record.) PARSE_DEF3
AUD_T_LSOCK	016	
AUD_T_RSOCK	017	
AUD_T_LHOSTNAME	020	
AUD_T_OPAQUE	030	token:length:value. (proplist or truly opaque; check for proplist name/value pairs else dump as hex) PARSE_DEF6
AUD_T_INTARRAY	031	token:length:int. PARSE_DEF3
AUD_T_GIDSET	032	token:length:int1, int2, ... (unaligned). PARSE_DEF3
AUD_T_XDATA	033	token:struct aud_xdata (See `<sys/audit.h>`.) PARSE_DEF8
AUD_T_AUID	040	token:int. PARSE_DEF2
AUD_T_RUID	041	token:int. PARSE_DEF2
AUD_T_UID	042	token:int. PARSE_DEF2
AUD_T_PID	043	token:int. PARSE_DEF2
AUD_T_PPID	044	token:int. PARSE_DEF2
AUD_T_GID	045	token:unsigned int. PARSE_DEF2
AUD_T_EVENT	046	token:int. PARSE_DEF2
AUD_T_SUBEVENT	047	token:int. PARSE_DEF2
AUD_T_DEV	050	token:int (Parse using the `major()`/`minor()` macros from `<sys/types.h>`.) PARSE_DEF2

Table 5–2: Token/Tuple Byte Descriptions (cont.)

Token	Octal Value	Tuple Format and Sample Parse Macro
AUD_T_ERRNO	051	token:int. PARSE_DEF1
AUD_T_RESULT	052	token:long. PARSE_DEF4
AUD_T_MODE	053	token:unsigned int. PARSE_DEF2
AUD_T_HOSTADDR	054	token:unsigned int. PARSE_DEF2
AUD_T_INT	055	token:int. PARSE_DEF2
AUD_T_DESCRIP	056	token:int (file descriptor). PARSE_DEF2
AUD_T_HOSTID	057	token:int. PARSE_DEF1
AUD_T_X_ATOM	060	token:unsigned int. PARSE_DEF2
AUD_T_X_CLIENT	061	token:int. PARSE_DEF2
AUD_T_X_PROPERTY	062	token:int. PARSE_DEF2
AUD_T_X_RES_CLASS	063	token:unsigned int. PARSE_DEF2
AUD_T_X_RES_TYPE	064	token:unsigned int. PARSE_DEF2
AUD_T_X_RES_ID	065	token:unsigned int. PARSE_DEF2
AUD_T_LHOSTNAME	066	
AUD_T_SECEVENT	177	token:int. PARSE_DEF2
AUD_TP_ACCRGHT	201	token:length:cmsg_data (fd1, fd2, ... – See `<sys/socket.h>`.) PARSE_DEF3
AUD_TP_MSGHDR	202	token:length:msghdr->msg_name. (See `<sys/socket.h>`.) PARSE_DEF3
AUD_TP_EVENTP	203	token:length:string. PARSE_DEF3
AUD_TP_HABITAT	204	token:length:string. PARSE_DEF3
AUD_TP_ADDRVEC	205	token:length:struct sockaddr. (See `socket.h`.) PARSE_DEF3
AUD_TP_INTP	206	token:length:int. PARSE_DEF3
AUD_TP_AUID	241	token:int. PARSE_DEF1
AUD_TP_RUID	0242	token:int. PARSE_DEF1
AUD_TP_UID	0243	token:int. PARSE_DEF1
AUD_TP_PID	0244	token:int. PARSE_DEF1
AUD_TP_PPID	0245	token:int. PARSE_DEF1
AUD_TP_HOSTADDR	246	token:unsigned int. PARSE_DEF1
AUD_TP_EVENT	247	token:int. PARSE_DEF1

Table 5–2: Token/Tuple Byte Descriptions (cont.)

Token	Octal Value	Tuple Format and Sample Parse Macro
AUD_TP_SUBEVENT	250	token:int (Reserved for future use.) PARSE_DEF1
AUD_TP_NCPU	251	token:int. PARSE_DEF1
AUD_TP_DEV	252	token:int (Parse using the `major()`/`minor()` macros from `sys/types.h`.) PARSE_DEF1
AUD_TP_LENGTH	253	token:int. PARSE_DEF1
AUD_TP_IPC_GID	254	token:unsigned int (ipc|msg|shm_perm.gid). PARSE_DEF2
AUD_TP_IPC_MODE	255	token:unsigned int (ipc|msg|shm_perm.mode). PARSE_DEF2
AUD_TP_IPC_UID	256	token:int (ipc|msg|shm_perm.uid). PARSE_DEF2
AUD_TP_TV_SEC	257	token:timeval.tv_sec (See `<sys/time.h>`.) PARSE_DEF1
AUD_TP_TV_USEC	260	token:timeval.tv_usec (See `<sys/time.h>`.) PARSE_DEF1
AUD_TP_SHORT	261	token:short. PARSE_DEF2
AUD_TP_LONG	262	token:long. PARSE_DEF5
AUD_TP_VNODE_DEV	263	token:int. PARSE_DEF2
AUD_TP_VNODE_ID	264	token:unsigned int. PARSE_DEF2
AUD_TP_VN-ODE_MODE	265	token:unsigned int. PARSE_DEF2
AUD_TP_VERSION	266	token:unsigned int. (See `<sys/audit.h>`.) (AUD_VERSION | AUD_VERS_LONG). PARSE_DEF1
AUD_TP_SET_UIDS	267	token:int. PARSE_DEF2
AUD_TP_CONT	270	token:unsigned int (A unique int for each component of a record.) PARSE_DEF1
AUD_TP_TID	271	token:long. PARSE_DEF4
AUD_TP_PRIV	272	token:unsigned short. PARSE_DEF1

5.10.3 Parsing Tuples

The algorithm for reading a stream of audit records is as follows:

1. Open the audit log.

2. Find the first audit record (starts and ends with AUD_TP_LENGTH tuples).
3. Check that the record length matches the value in the AUD_TP_LENGTH tuple. (If the length does not match, discard the record.)
4. Retrieve the first tuple following the AUD_TP_LENGTH tuple.
5. If the tuple length is variable, determine the size of the data.
6. Extract the data.
7. Retrieve the next tuple, check the length if necessary, and extract the data.
8. Repeat until no more records.
9. Close the audit log.

The following macros illustrate how audit_tool parses tuples. The macros are provided for reference purposes only; they illustrate one approach. Note that *indx* values are maintained and used by audit_tool; they are not part of the audit record tuple.

```
/* fixed length scalar value */
#define PARSE_DEF1(tokentype,field) \
  bcopy (&rec_ptr[i+sizeof token], &field, sizeof(field)); \
  i += (sizeof token + sizeof(field)); \
  break;

/* fixed length field in array */
#define PARSE_DEF2(tokentype,field,indx) \
  if (indx < AUD_NPARAM) \
      bcopy (&rec_ptr[i+sizeof token], &field[indx++], sizeof(field[0])); \
  i += (sizeof token + sizeof(field[0])); \
  break;

/* array of strings */
#define PARSE_DEF3(tokentype,len,field,indx) \
  bcopy (&rec_ptr[i+sizeof token], &j, sizeof(int)); \
  if (j >= rec_len) j = 0; \
  if (indx < AUD_NPARAM) { \
      len[indx] = j; \
      field[indx++] = (char *)&rec_ptr[i+(sizeof token)+(sizeof *intp)]; \
  } \
  i += (sizeof token + sizeof *intp + j); \
  break;

/* fixed length scalar value whose size is h/w dependent (32 or 64-bit) */
#define PARSE_DEF4(tokentype,field) \
  bzero (field.val, sizeof(field.val)); \
  j = af->version & AUD_VERS_LONG ? sizeof(int)*2 : sizeof(int); \
  bcopy (&rec_ptr[i+sizeof token], field.val, j); \
  i += (sizeof token + j); \
  break;

/* fixed length field in array whose size is h/w dependent (32 or 64-bit) */
#define PARSE_DEF5(tokentype,field,indx) \
  bzero (field[indx].val, sizeof(field[indx].val)); \
  j = af->version & AUD_VERS_LONG ? sizeof(int)*2 : sizeof(int); \
  if (indx < AUD_NPARAM) \
      bcopy (&rec_ptr[i+sizeof token], field[indx++].val, j); \
  i += (sizeof token + j); \
  break;

/* array of opaque data streams */
#define PARSE_DEF6 PARSE_DEF3

/* iovec element in array */
#define PARSE_DEF7(tokentype,field,indx) \
  j = sizeof(field[0]); \
  if (indx < AUD_NPARAM) { \
      bcopy (&rec_ptr[i+sizeof token], &j, sizeof(int)); \
      if (j > rec_len ) j = 0; \
      bcopy (&rec_ptr[i+sizeof token+sizeof(int)], &field[indx++], j); \
  } \
  i += (sizeof token + sizeof(int) + j); \
  break;

/* array of iovec elements with variable length components */
#define PARSE_DEF8(tokentype,field,ptr,indx) \
  j = sizeof(field[0]); \
  if (indx < AUD_NPARAM) { \
      bcopy (&rec_ptr[i+sizeof token], &j, sizeof(int)); \
      if (j > rec_len) j = 0; \
      bcopy (&rec_ptr[i+sizeof token+sizeof(int)], &field[indx], j); \
      ptr[indx++] = ((struct aud_xdata *) \
                    &rec_ptr[i+sizeof token+sizeof(int)])->xdata; \
  } \
  i += (sizeof token + sizeof(int) + j); \
  break;
```

6 Using the SIA Interface

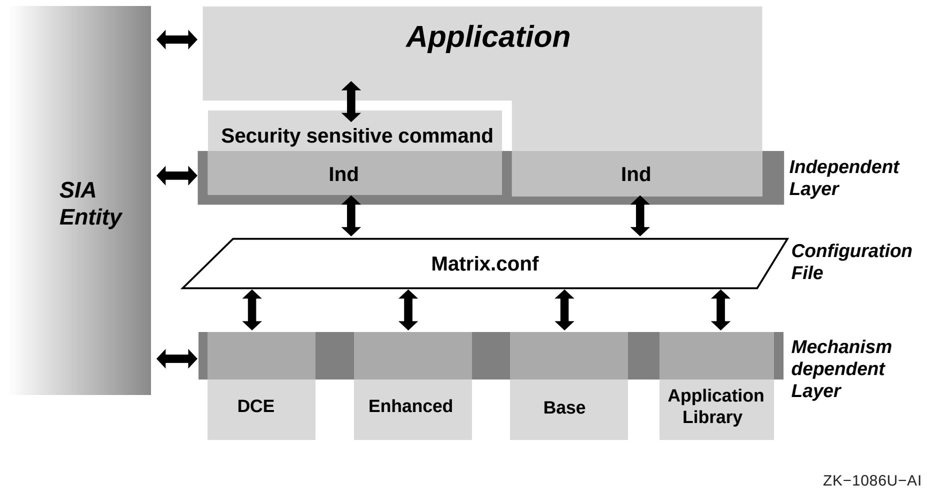
This chapter contains the following information:

- SIA overview
- SIA architecture
- SIA system initialization
- SIAENTITY structure
- SIA parameter collection
- Maintaining state
- SIA return values
- SIA debugging and logging
- Integrating security mechanisms
- SIA session processing
- Changing secure information
- Accessing Security Information
- Session parameter collection
- Packaging products for the SIA
- Security mechanism-dependent interface
- Single-user mode
- Symbol preemption for SIA routines

6.1 SIA Overview

The Security Integration Architecture (SIA) allows the layering of local and distributed security authentication mechanisms onto the Tru64 UNIX operating system. The SIA configuration framework isolates security-sensitive commands from the specific security mechanisms. The Tru64 UNIX security-sensitive commands have been modified to call a set of mechanism-dependent routines. By providing a library with a unique set of routines, developers can change the behavior of security-sensitive commands, without changing the commands themselves. The SIA defines the security mechanism-dependent interfaces (`siad_*()` routines) required for SIA configurability. Figure 6–1 illustrates the relationship of the components that make up the SIA.

Figure 6–1: SIA Layering



The security-sensitive commands are listed in Table 6–1.

Table 6–1: Security-Sensitive Operating System Commands

Command	Description
`chfn`	Changes finger information
`chsh`	Changes login shell information
`dnascd`	Spans DECnet
`ftpd`	Serves the Internet File Transfer Protocol
`login`	Authenticates users
`passwd`	Creates or changes user passwords
`rshd`	Serves remote execution
`su`	Substitutes a user ID

Table 6–2 and Table 6–3 list the SIA porting routines.

Table 6–2: SIA Mechanism-Independent Routines

SIA Routine	Description
`sia_init()`	Initializes the SIA configuration
`sia_chk_invoker()`	Checks the calling application for privileges
`sia_collect_trm()`	Collects parameters

Table 6–2: SIA Mechanism-Independent Routines (cont.)

SIA Routine	Description
`sia_chg_finger()`	Changes finger information
`sia_chg_password()`	Changes the user's password
`sia_chg_shell()`	Changes the login shell
`sia_ses_init()`	Initializes SIA session processing
`sia_ses_authent()`	Authenticates an entity
`sia_ses_reauthent()`	Revalidates a user's password
`sia_ses_suauthent()`	Processes the `su` command
`sia_ses_estab()`	Establishes the context for a session
`sia_ses_launch()`	Logs session startup and any TTY conditioning
`sia_ses_release()`	Releases resources associated with session
`sia_make_entity_pwd()`	Provides the password structure for SIAENTITY
`sia_audit()`	Generates the audit records
`sia_chdir()`	Changes the current directory safely (NFS-safe)
`sia_timed_action()`	Calls with a time limit and signal protection
`sia_become_user()`	`su` routine
`sia_validate_user()`	Validate a user's password
`sia_get_groups()`	Gets groups

Table 6–3: SIA Mechanism-Dependent Routines

SIA Routine	Description
`siad_init()`	Initializes processing once per reboot
`siad_chk_invoker()`	Verifies the calling program privileges
`siad_ses_init()`	Initializes the session
`siad_ses_authent()`	Authenticates the session
`siad_ses_estab()`	Checks resources and licensing
`siad_ses_launch()`	Logs the session startup
`siad_ses_suauthent()`	Processes the `su` command
`siad_ses_reauthent()`	Revalidates a user's password
`siad_ses_release()`	Releases session resources
`siad_chg_finger()`	Processes the `chfn` command

Table 6–3: SIA Mechanism-Dependent Routines (cont.)

SIA Routine	Description
`siad_chg_password()`	Invokes a function to change passwords
`siad_chg_shell()`	Processes the `chsh` command
`siad_getpwent()`	Processes `getpwent()` and `getpwent_r()`
`siad_getpwuid()`	Processes `getpwuid()` and `getpwuid_r()`
`siad_getpwnam()`	Processes `getpwnam()` and `getpwnam_r()`
`siad_setpwent()`	Initializes a series of `getpwent()` calls
`siad_endpwent()`	Releases resources after a series of `getpwent()` calls
`siad_getgrent()`	Processes `getgrent()` and `getgrent_r()`
`siad_getgrgid()`	Processes `getgrgid()` and `getgrgid_r()`
`siad_getgrnam()`	Processes `getgrnam()` and `getgrnam_r()`
`siad_setgrent()`	Initializes a series of `getgrent()` calls
`siad_endgrent()`	Closes series of `getgrent()` calls
`siad_chk_user()`	Determines if a mechanism can change the requested information
`siad_get_groups()`	Fills in the array of a user's supplementary groups

The SIA establishes a layer between the security-sensitive commands and the security mechanisms that deliver the security mechanism-dependent functions. Each of the security-dependent SIA routines can be configured to use up to four security mechanisms, called in varying orders.

The selection and order of the calls to the different security mechanisms is established by a switch table file, `/etc/sia/matrix.conf` (see Security Administration), similar to the way `/etc/svc.conf` is used to control `libc get*` functions. However, the calling mechanism is distinctly different.

The SIA calling mechanism looks up the addresses of routines in the shared libraries and calls them to access the specific security mechanism routine. SIA provides alternative control and configuration for the `getpw*` and `getgr*` functions in Tru64 UNIX.

SIA layering establishes internationalized message catalog support and thread-safe porting interfaces for new security mechanisms and new security-sensitive commands that need transparency. The thread safety is provided by a set of locks pertaining to types of SIA interfaces. However, because SIA is a layer between utilities and security mechanisms, it is the responsibility of the layered security mechanisms to provide reentrancy in their implementations.

The primary focus for SIA is to provide transparent interfaces for security-sensitive commands like `login`, `su`, and `passwd` that are sufficiently flexible and extensible to suit future security requirements. Any layered product on Tru64 UNIX that is either creating a new security mechanism or includes security-sensitive commands requires SIA integration to preserve these transparent interfaces.

The SIA components consist of only user-level modules. The components resolve the configuration issues with respect to the security-sensitive command's utilization of multiple security mechanisms. The SIA components do not resolve any kernel issues pertaining to the configuration and utilization of multiple security mechanisms.

6.2 SIA Architecture

The layering architecture introduced by SIA in Tru64 UNIX consists of the following two groups of interface routines:

`sia_*`()	The security mechanism-independent interface used by security-sensitive commands.
`siad_*`()	The security mechanism-dependent interface supplied by each specific security mechanism.

Each security mechanism delivers a shared library containing the `siad_*()` routines and provides a unique security mechanism name to satisfy the configuration. The one word security mechanism name and the library name are used as keys in the `matrix.conf` file to specify which mechanisms to call and in what order.

The Tru64 UNIX security-sensitive commands have been modified to use the mechanism-independent `sia_*`() routines. These routines are used by the commands and utilities to access security functions yet remain isolated from the specific security technologies. Each `sia_*`() routine calls the associated mechanism-dependent `siad_*`() routines, depending on the selected configuration specified in the `matrix.conf` file. See Security Administration for a more detailed discussion of the file.

The mechanism-dependent `siad_*`() interface routines are defined by SIA as callouts to security mechanism-dependent functions provided by the security mechanisms. The `matrix.conf` file is used to determine which security mechanisms are called and in what order they are called for each SIA function.

The process of calling a particular module within a specified security mechanism and passing the required state is done by the mechanism-independent layer. The calling process uses shared library functions to access and look up specific module addresses within specified shared libraries provided by the security mechanisms.

The naming of the security mechanism-dependent modules, `siad_*`() routines, is fixed to alleviate name conflicts and to simplify the calling sequence. Tru64 UNIX uses the `dlopen`() and `dlsym`() shared library interfaces to open the specified security-mechanism shared library and look up the `siad_*`() function addresses. If you need to preempt the `siad_*`() routines, your names must be of the form `__siad_*` in your library and the library must be linked ahead of `libc`. See Section 6.17 for more information on the naming and preempting requirements.

6.2.1 Libraries

SIA security mechanisms are configured as separate shared libraries with entry points that are SIA defined names. Each mechanism is required to have a unique mechanism identifier. The actual entry points in the shared library provided by the security mechanism are the same for each mechanism, `siad_*`() form entry points.

The default security configuration is the BASE security mechanism contained in `libc`. The default BASE security mechanism uses the `/etc/passwd` file, or a hashed database version, as the user database and the `/etc/group` file as the group's database. The default BASE mechanism also uses the Network Information Service (NIS) if it is configured. In single-user mode or during installation, the BASE security mechanism is in effect.

6.2.2 Header Files

The SIA interfaces and structures are defined in the `/usr/include/sia.h` and `/usr/include/siad.h` files. The `sia*.h` files are part of the program development subsets.

6.3 SIA System Initialization

The SIA provides a callout to each security mechanism on each reboot of the system. This callout is performed by the `/usr/sbin/siainit` program, which calls each of the configured security mechanisms at their `siad_init()` entry point. This allows the security mechanisms to perform a reboot initialization. A SIADFAIL response from the `siad_init()` call causes the system to not reboot and an SIA INITIALIZATION FAILURE message to be sent to the console. Consequently, only problems that would cause a security risk or would not allow `root` to log in should warrant a SIADFAIL response from the `siad_init`() call.

6.4 SIAENTITY Structure

The SIAENTITY structure contains session processing parameters and is used to transfer session state between the session processing stages. Example 6–1 shows the SIAENTITY structure.

Example 6–1: The SIAENTITY Structure

```
typedef struct siaentity {
    char *name;            /* collected name                */
    char *password;        /* entered or collected password */
    char *acctname;        /* verified account name         */
    char **argv;           /* calling command argument list */
    int  argc;             /* number of arguments           */
    uid_t  suid;           /* starting ruid                 */
    char *hostname;        /* requesting host NULL=>local   */
    char *tty;             /* pathname of local tty         */
    int can_collect_input; /* 1 => yes, 0 => no input       */
    int error;             /* error message value           */
    int authcount;         /* Number of consecutive         */
                           /* failed authent attempts       */
    int authtype;          /* Type of last authent          */
    struct passwd *pwd;    /* pointer to passwd struct      */
    char *gssapi;          /* for gss_api prototyping       */
    char *sia_pp;          /* for passport prototyping      */
    int *mech[SIASWMAX];   /* pointers to mech-specific data  */
                           /* allocated by mechanisms indexed */
                           /* by the mechind argument         */
} SIAENTITY;
```

6.5 SIA Parameter Collection

The SIA provides parameter collection callback capability so that any graphical user interface (GUI) can provide a callback. The `sia_collect_trm`() routine is used for terminal parameter collection. Commands calling the `sia_*`() routines pass as an argument to the appropriate collection routine pointer, thus allowing the security mechanism to prompt the user for specific input. If the collection routine argument is NULL, the security mechanism assumes that no collection is allowed and that the other arguments must be used to satisfy the request. The NULL case is used for noninteractive commands. For reliability, use a collection routine whenever possible.

The `can_collect_input` argument is included in the session processing and disables the collection facility for input while allowing the output of warnings or error messages. Collection routines support simple form and menu data collection. Some field verification is supported to check parameter lengths and content (alphanumeric, numeric only, letters only, and invisible). The collection routine supplied by the security-sensitive command or utility is responsible for providing the appropriate display characteristics.

The parameter collection capability provided by SIA uses the `sia_collect_trm`()interface which is defined in `sia.h`. See Example 6–2.

Example 6–2: The sia.h Definition for Parameter Collection

```
int sia_collect_trm(timeout, rendition, title,
     num_prompts, prompts);

 int timeout              /* number of seconds to wait */
                          /* 0 => wait forever */
 int rendition
    SIAMENUONE      1      /* select one of the choices given */
    SIAMENUANY      2      /* select any of the choices given */
    SIAFORM         3      /* fill out the form               */
    SIAONELINER     4      /* One question with one answer    */
    SIAINFO         5      /* Information only                */
    SIAWARNING      6      /* ERROR or WARNING message        */

  unsigned char *title      /* pointer to a title string. */
                          /* NULL => no title */

  int num_prompts           /* Number of prompts in collection */
  prompt_t *prompts         /* pointer to prompts */

  typedef struct prompt_t
    {
    unsigned char *prompt;
    unsigned char *result;
    int max_result_length; /* in chars */
    int min_result_length; /* in chars */
    int control_flags;
    } prompt_t;

  control_flags
   SIARESINVIS  0x2   result is invisible
   SIARESANY    0x10  result can contain any ASCII chars
   SIAPRINTABLE 0x20  result can contain only printable chars
   SIAALPHA     0x40  result can contain only letters
   SIANUMBER    0x80  result can contain only numbers
   SIAALPHANUM  0x100 result can contain only letters and numbers
```

See the `sia_collect_trm`(3) reference page for more information on parameter collection.

6.6 Maintaining State

Some commands require making multiple calls to `sia_*()` routines and maintaining state across those calls. The state is always associated with a particular user (also called an entity). SIA uses the term entity to mean a user, program, or system which can be authenticated. The entity identifier is the user ID (UID). All security mechanisms which are ported to Tru64 UNIX must be administered such that a particular UID maps equivalently across each mechanism. This constraint allows for the interaction and coexistence of multiple security mechanisms. If a security mechanism has an alternative identifier for a user, it must provide a mapping to a unique UID for other mechanisms to properly interoperate and provide synchronized security information.

A pointer to the SIAENTITY structure (see Section 6.4) is used as an argument containing intermediate state identifying the entity requesting a security session function. The SIAENTITY structure also allows for the sharing of state between security mechanisms while processing a session.

The `libc` library provides for the allocating and freeing of primitives for SIAENTITY structures. The allocation of the SIAENTITY structures occurs as part of the session initialization routine, `sia_ses_init`(). The deallocation of the SIAENTITY structure occurs in the call to the session release `sia_ses_release`() routine. If errors occur during session processing (such as in the `sia_ses_*authent`() routines) and you give up instead of retrying, `sia_ses_release`() must be called to clean or free up the SIAENTITY structure related to the session. If errors occur during an `sia_ses_estab()` or `sia_ses_launch`() routine causing failure status to be returned, the routines call `sia_ses_release`().

6.7 SIA Return Values

SIA supports the passing of a success or failure response back to the calling command or utility. The SIAENTITY structure has a reserved error code field (*`error`*), which is available for finer error definition.

The `siad_ses_*()` routines return bitmapped values that indicate the following status:

SIADFAIL	Indicates conditional failure. Lowest bit set to 0. Continue to call subsequent security mechanisms.

SIADSUCCESS	Indicates conditional success. Lowest bit set to 1.
SIADSTOP	Modifies the return to be unconditional. Second lowest bit set to 1. Included with either SIADFAIL or SIADSUCCESS.

6.8 SIA Debugging and Logging

SIA supports a debugging and logging capability that allows appending data to the `/var/adm/sialog` file. The SIA logging facility supports the following three log-item types:

EVENT	Success cases within the SIA processing
ERROR	Failures within the SIA processing
ALERT	Security configuration or security risks within the SIA interfaces

The `sia_log()` logging routine is available to security mechanisms and accepts formatting strings compatible to `printf()` format. Each log entry is time stamped. Example 6–3 is a typical `/var/adm/sialog` file.

Example 6–3: Typical /var/adm/sialog File

```
SIA:EVENT Wed Feb  3 05:21:31 1999
Successful SIA initialization
SIA:EVENT Wed Feb  3 05:22:08 1999
Successful session authentication for terry on :0
SIA:EVENT Wed Feb  3 05:22:08 1999
Successful establishment of session
SIA:ERROR Wed Feb  3 05:22:47 1999
Failure to authenticate session for root on :0
SIA:ERROR Wed Feb  3 05:22:52 1999
Failure to authenticate session for root on :0
SIA:EVENT Wed Feb  3 05:22:59 1999
Successful session authentication for root on :0
SIA:EVENT Wed Feb  3 05:22:59 1999
Successful establishment of session
SIA:EVENT Wed Feb  3 05:23:00 1999
Successful launching of session
SIA:EVENT Wed Feb  3 05:24:40 1999
Successful authentication for su from root to terry
SIA:EVENT Wed Feb  3 05:25:46 1999
Successful password change for terry
```

The `sia_log`() routine is for debugging only. The `_ses_*` routines use `audgen`() for audit logging.

6.9 SIA Integrating Security Mechanisms

Depending on the class or type of SIA processing being requested, the selection and order of security mechanisms may vary. A typical set of security mechanisms might include a local mechanism (one that is only concerned with the local system security) and a distributed security mechanism (one that is concerned with aspects of security that span several systems). SIA layering allows these two security mechanisms to either coexist or be better integrated.

An example of security mechanism integration is the log in or session processing. SIA layering passes state (SIAENTITY) between the various security mechanisms during the session processing. This state contains collected names and passwords and the current state of session processing. The local security mechanism can be designed to trust the authentication process of a previously run security mechanism, thus allowing authentication vouching. In this case, if a user is successfully authenticated by the distributed mechanism, the local mechanism can accept or trust that authentication and continue with session processing.

SIA also allows the local mechanism to not accept vouching. In this case, the local mechanism would be forced to do its own authentication process regardless of previous authentication outcomes. This typically results in the user being asked for several sets of user names and passwords. Although SIA allows any ordering of security mechanisms, it makes sense that those mechanisms that accept vouching should be ordered after those that do not.

Notes

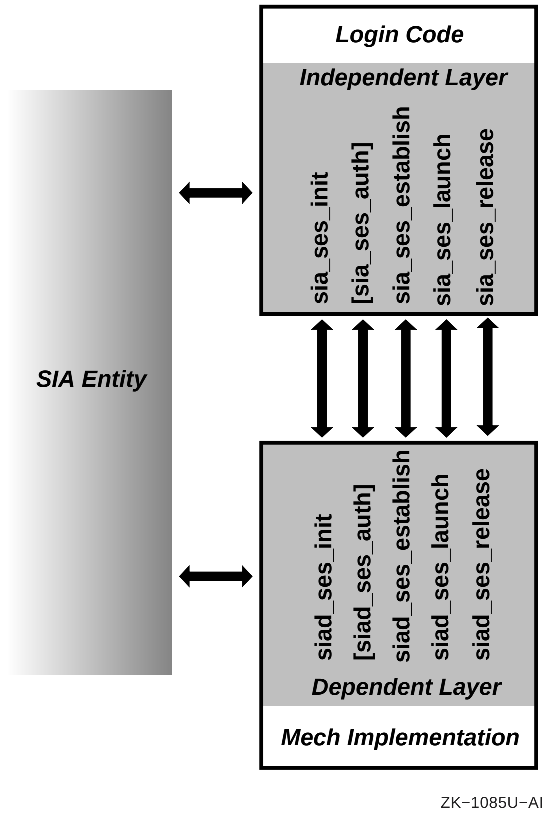
The default security mechanism, BASE, accepts authentication vouching.

The SIA layer deals with the isolation of security mechanisms from the commands' specific user interface preferences. To accomplish this isolation, the calling command provides a pointer to a parameter collection routine as an argument to the `sia_*`() routines. The collection routine must support simple form and menu type processing. The definitions or the requirements of the collection routine are defined in `sia.h`. This separation of user interface from the security mechanisms allows the flexibility to change the user interface to suit any workstation or dumb terminal model.

6.10 SIA Session Processing

The session processing interfaces are associated with the process of a utility or command that needs to become or act as some other entity. Figure 6–2 illustrates the SIA routines and their relationship in a typical login session.

Figure 6–2: SIA Login Session Processing



The session processing interfaces to the security mechanism-dependent routines (`siad_*()`) use the same returns to determine the state of the session and whether it should continue. The returns are as follows:

SIADFAIL	A SIADFAIL response from a security mechanism `siad_*()` routine indicates that the security mechanism has failed but that processing should continue.
SIAD-FAIL | SIASTOP	A SIADFAIL | SIADSTOP response from a security mechanism `siad_*()` routine indicates that the security mechanism has failed and that the session

	processing should be stopped. This return is used if some major security problem or risk is found. Such an event should be sent to the `sialog` file as an ALERT.
SIADSUCCESS	The final response is SIADSUCCESS, which indicates that the security mechanism has successfully completed that phase of session processing. Under some conditions, a return of SIADSUCCESS | SIADSTOP is also useful.

Not all security mechanisms have processing required in each phase of the session processing. In general, the default response is SIADFAIL to force the other configured security mechanisms to produce the required SIADSUCCESS response. The only exceptions to this are the first and last stages of session processing. If a security mechanism has nothing to do in either session initialization or session release, it should return a SIADSUCCESS response. For all other phases of session processing, a SIADFAIL response is the default.

The session processing interfaces are typically called in the following order:

`sia_ses_init()`	Initialize the session.
`sia_ses_authent()`	Authenticate the session. Can be recalled on failure for retries.
`sia_ses_estab()`	Establish the session. On failure, calls `sia_ses_release()`.
`sia_ses_launch()`	Launch the session. On failure, calls `sia_ses_release()`.
`sia_ses_release()`	Release the session.

The session routines must all have the same number and order of mechanisms to keep the mechanism index (`mechind`) consistent.

Example 6–4 is a code fragment that shows session processing for the `login` command.

Example 6–4: Session Processing Code for the login Command

```
   .
   .
   .
/* SIA LOGIN PROCESS BEGINS */

/* Logging of failures to sia_log is done within the libsia */
/* Logging to syslog is responsibility of calling routine   */

if((sia_ses_init(&entity, oargc, oargv, hostname, loginname, \
   ttyn, 1, NULL)) == SIASUCCESS) {

/***** SIA SESSION AUTHENTICATION *****/

   if(!fflag) {
      for(cnt=5; cnt; cnt--) {
         if((authret=sia_ses_authent(sia_collect,NULL,entity)) \
   == SIASUCCESS)
           break;
         else    if(authret & SIASTOP)
           break;
         fputs(MSGSTR(INCORRECT, "Login incorrect\n"), stderr);
      }
      if(cnt <= 0 || (authret & SIASTOP)) {
         sia_ses_release(&entity);
         exit(1);
      }
   }

/***** SIA SESSION ESTABLISHMENT *****/

   if(sia_ses_estab(sia_collect,entity) == SIASUCCESS) {
      /****** set up environment   *******/
      /* destroy environ. unless user requested preservation */
      if (!pflag) {
        pp = getenv("TERM");
        if (pp)
        strncpy(term, pp, sizeof term);
        clearenv();
   }
   (void)setenv("HOME", entity->pwd->pw_dir, 1);
      if(entity->pwd->pw_shell && *entity->pwd->pw_shell)
        strncpy(shell, entity->pwd->pw_shell, sizeof shell);
   (void)setenv("SHELL", shell, 1);
   if (term[0] == ’ ’)
      (void)strncpy(term, stypeof(tty), sizeof(term));
   (void)setenv("TERM", term, 0);
   (void)setenv("USER", entity->pwd->pw_name, 1);
   (void)setenv("LOGNAME", entity->pwd->pw_name, 1);
```

Example 6–4: Session Processing Code for the login Command (cont.)

```
   (void)setenv("PATH", _PATH_DEFPATH, 0);

/***** SIA LAUNCHING SESSION *****/

   if(sia_ses_launch(sia_collect,entity) == SIASUCCESS) {
   /* 004 - start */
   if ((entity -> pwd           != NULL) &&
       (entity -> pwd -> pw_dir != NULL) &&
       (entity -> pwd -> pw_dir [0] != 0))
          sprintf (hush_path, "%s/%s",
             entity -> pwd -> pw_dir,
             _PATH_HUSHLOGIN);
   else    strcpy (hush_path, _PATH_HUSHLOGIN);
   quietlog = access(hush_path, F_OK) == 0;
   /* 004 - end */
   if(!quietlog)
      quietlog = !*entity->pwd->pw_passwd && \
   !usershell(entity->pwd->pw_shell);
         if (!quietlog) {
            struct stat st;
            motd();
            (void)sprintf(tbuf, "%s/%s", _PATH_MAILDIR, \
    entity->pwd->pw_name);
            if (stat(tbuf, &st) == 0 && st.st_size != 0)
               (void)printf(MSGSTR(MAIL, "You have %smail.\n"),
                      (st.st_mtime > st.st_atime) ? MSGSTR(NEW, \
     "new ") : );
         }
         sia_ses_release(&entity);

/******* Setup default signals **********/

         (void)signal(SIGALRM, SIG_DFL);
         (void)signal(SIGQUIT, SIG_DFL);
         (void)signal(SIGINT, SIG_DFL);
         (void)signal(SIGTSTP, SIG_IGN);

         tbuf[0] = '-';
         (void)strcpy(tbuf + 1, (p = rindex(shell, '/')) ?
                         p + 1 : shell);

/****** Nothing left to fail *******/

         if(setreuid(geteuid(),geteuid()) < 0) {
            perror("setreuid()");
            exit(3);
         }
```

Example 6–4: Session Processing Code for the login Command (cont.)

```
         execlp(shell, tbuf, 0);
         (void)fprintf(stderr, MSGSTR(NO_SHELL, \
  "login: no shell: %s.\n"), strerror(errno));
         exit(0);
      }
/***** SIA session launch failure *****/
    }
/***** SIA session establishment failure *****/
  }
  logerror(entity);
  exit(1);
}

logerror(entity)
SIAENTITY *entity;
{
  if(entity != NULL)
    {
    sia_ses_release(&entity);
    }
  syslog(LOG_ERR, MSGSTR(FAILURE3," LOGIN FAILURE "));
}
  .
  .
  .
```

6.10.1 Session Initialization

Session initialization is performed by the `sia_ses_init()` routine. The `sia_ses_init()` routine calls each configured security mechanism's `siad_ses_init()` entry point to do any processing associated with the start of a session processing sequence. The session initialization stage is responsible for setting up the SIAENTITY structure, which is used to maintain state though the different stages of session processing.

6.10.2 Session Authentication

The authentication stage of session processing is responsible for proving the identity for the session. This stage of the processing must determine the entity associated with the session. If the entity cannot be determined, the authentication fails. If the authentication is successful, an entity is derived.

The top level SIA session authentication routine, `sia_ses_authent()`, calls the security mechanism-dependent `siad_ses_authent()` routines according to the configured sequence stored in the `matrix.conf` file. As the multiple authentication routines are called, the SIAENTITY structure is used to hold precollected parameters like the name, password, and eventually the associated `/etc/passwd` entry of the entity.

By using precollected arguments, the security mechanisms avoid recollecting arguments. An example is when `root` attempts to log in to a system configured to first call the DCE `siad_ses_authent()` routine followed by the local ENHANCED (enhanced security) `siad_ses_authent()` routine.

It is likely that the DCE authentication process will not be capable of authenticating `root`. However, it is capable of asking the user for a name and password, which are then passed to the ENHANCED `siad_ses_authent()` routine using the SIAENTITY structure. This allows the ENHANCED mechanism to verify the `root` name and password, thus authenticating `root`. As soon as the session authentication stage is complete, the password field is cleared.

Each security mechanism-dependent authentication routine must have the ability to determine and set the entity on a successful authentication. If a security mechanism has its own private interpretation of the entity, it must provide a translation to the common SIA entity, user name and UID. Without this restriction there is no way to synchronize security mechanisms with respect to a common entity.

At the successful completion of the session authentication stage, the SIAENTITY structure must contain the user name and UID of the authenticated entity. If the session authentication fails, the calling command or program can call `sia_ses_authent()` again to retry the authentication process. Certain mechanisms may allow other mechanisms to vouch for this stage of session processing. This usually occurs when local mechanisms default their authentication process to other distributed mechanisms.

6.10.3 Session Establishment

The session establishment stage is invoked with `sia_ses_estab()` following a successful session authentication stage. The `sia_ses_estab()` routine is configured to call multiple security mechanism's `siad_ses_estab()` routines in the order defined in the `matrix.conf` file. The session establishment stage of session processing is responsible for checking mechanism resources and licensing to determine whether this session can be successfully launched. The determination of the `passwd struct` entry and any other required security context must occur in this stage. At the successful completion of the session establishment stage, the system is prepared to grant the session launching.

6.10.4 Session Launch

The session launch stage is responsible for the logging and the accounting of the session startup. The local mechanism is additionally responsible for setting the `wtmp` and `utmp` entries and for setting the effective UID to the UID associated with the entity. The processing by the `setgid()` and `initgroup()` routines as well as `lastlog` updating are also done by the local mechanism. Only catastrophic errors should be able to stop the session from continuing.

6.10.5 Session Release

The last stage of the session processing sequence (either successful or failed) is the call to the `sia_ses_release()` routine. This routine frees all session processing resources, such as the SIAENTITY structure. Each configured mechanism is called to release any resources which are no longer required for the session.

6.10.6 Specific Session Processing

The following sections describe specific session processing for the `login`, `rshd`, and `rlogind` commands.

6.10.6.1 The login Process

The most common case of session processing is when the login process becomes the entity associated with a user. The entity is the unique SIA identifier for any person or process that can be authenticated and authorized. The code in Example 6–4 is part of the `login` command.

6.10.6.2 The rshd Process

Session processing for `/usr/sbin/rshd` differs from `login`. The `rshd` process calls `ruserok()` to check the `.rhosts` and `host.equiv` files for authorization. If `ruserok()` fails, the `rshd` fails.

6.10.6.3 The rlogind Process

The `rlogind`, program executes the `login` command with the `–f` flag if its call to `ruserok()` is successful, and without the `–f` flag if the call to `ruserok()` is unsuccessful. If `login` is executed without the `–f` flag, `sia_ses_authent()` is called, which prompts for a user name and password, if required.

6.11 Changing Secure Information

The routines described in this section handle the changing of the traditional /etc/passwd entry information. This class of routines could be extended to handle other types of common secure information. Only the traditional passwd, chfn, and chsh types of command processing are specified. Each of these routines follows the same operational model. When a user requests a change, the routines in this class check each mechanism that was configured by calling siad_chk_user() to determine whether the user is registered with the mechanism. Once it is determined that the user is registered with more than one security mechanism, the user is given a menu selection by the collection routine to choose which mechanism is targeted for the change. If only one mechanism is configured to handle the request, then that mechanism is called directly.

6.11.1 Changing a User's Password

To change a password, the sia_chg_password() routine calls the configured mechanisms by using the siad_chg_password() routine. To determine which mechanisms support a particular user, the siad_chk_user() call is made to all mechanisms configured for the siad_chg_passwd() routine. When multiple mechanisms claim registry of a user, the user is given a selection to choose from. If the user is only registered with one mechanism, then that mechanism is called.

6.11.2 Changing a User's Finger Information

The sia_chg_finger() routine calls the configured mechanisms by the siad_chg_finger() routine to change finger information. To determine which mechanisms support a particular user, the siad_chk_user() call is made to all mechanisms configured for the siad_chg_finger() routine. When multiple mechanisms claim registry of the user, the user is given a selection menu to choose one from. If the user is only registered with one mechanism, then that mechanism is called.

6.11.3 Changing a User's Shell

The sia_chg_shell() routine calls the configured mechanisms by the siad_chg_shell() routine to change a user's login shell. To determine which mechanisms support a particular user, the siad_chk_user() call is made to all mechanisms configured for the siad_chg_shell() routine. When multiple mechanisms claim registry of the user, the user is given a selection menu from which to choose a mechanism. If the user is only registered with one mechanism, then that mechanism is called.

6.12 Accessing Security Information

The SIA interfaces described in the following sections handle the access to the traditional UNIX `/etc/passwd` and `/etc/group` information. You can create routines to handle the access of other common secure information. Mechanism-dependent security information access should not be handled by the SIA interfaces unless nearly all mechanisms support the type of information being accessed.

The `sia_context` and `mech_contexts` structures, defined in `sia.h`, are used to maintain state across mechanisms. The structures are as follows:

```
struct mech_contexts {
        void *value;
        void (*destructor)();
};

struct sia_context {
        FILE *fp;
        union {
        struct group *group;
        struct passwd *pass;
        } value;
        int pkgind;
        unsigned buflen;
        char *buffer;
        struct mech_contexts mech_contexts[SIASWMAX];
};
```

Because the `getgr*()` and the `getpw*()` routines have SIA interfaces, security mechanisms need provide only one routine for both reentrant and nonreentrant, threadsafe applications. This is accomplished by the `sia_getpasswd()` and `sia_getgroup()` routines which encapsulate the arguments in a common form for the security mechanism's `siad_*()` routines.

6.12.1 Accessing /etc/passwd Information

Access to traditional `/etc/passwd` entries is accomplished by the `getpw*()` routines in `libc` and `libc_r`. The `sia_getpasswd()` routine in the SIA layer preserves the calling semantics of the current `getpw*()` routines and converts them into one common routine used for both single and multithreaded processes. By doing this conversion, security mechanisms need only support one set of `getpw*()` routines. The processing of the `getpwent()` routine is accomplished by calling each configured security mechanism in the predefined order until all entries have been exhausted.

6.12.2 Accessing /etc/group Information

Access to traditional `/etc/group` entries is accomplished by the `getgr*()` routines in `libc` and `libc_r`. The `sia_getgroup()` routine in the SIA layer preserves the calling semantics of the current `getgr*()` routines and converts them into one common routine used for both single and multithreaded processes. The conversion to a single routine eases the security mechanism port by reducing the number of routines required. The processing of the `getgrent()` routine is accomplished by calling each configured security mechanism in the predefined order until all group entries have been exhausted.

6.13 Session Parameter Collection

The SIA session interfaces and the interfaces that change secure information use a predefined parameter collection capability. The calling application passes the address to a parameter collection routine through the SIA to the `siad_*()` routines. The collection routine allows different security mechanisms to prompt the user for different parameters without having to be aware of the user interface details.

This capability isolates the SIA security mechanisms from the user interface and the ability to do simple forms and menus. This collection capability is sufficiently limited to allow ease of implementation by different user-interface packages or windowing systems. However, the collection routines must support simple (up to eight item) menu or form styles of processing. On dumb terminals, forms processing becomes a set of one line questions. Without this capability, the application needs to be modified to support new security questions.

6.14 Packaging Products for the SIA

The SIA defines the security mechanism components that are required to port to the Tru64 UNIX system. These components are as follows:

- A shared library containing the mechanism-dependent (`siad_*()`) routines used as an interface to commands and utilities
- A default `/etc/sia/matrix.conf` file, which is installed to use the security mechanism through SIA

The shared library must contain all of the `siad_*()` routines described in Table 6–3. The default dummy routine for any `siad_*()` routine always returns the SIADFAIL failure response. If a security mechanism is supplying dummy routines, these routines should not be configured into the `matrix.conf` file.

The /etc/sia/matrix.conf file contains one line for each siad_*() routine. This line contains the mechanism identifiers (called mech_types) and the actual path to the security mechanism library. The sia_*() routines use this set of keys to call mechanisms in a right to left ordering. See Security Administration for the default matrix.conf settings for Tru64 UNIX.

If the DCE security mechanism is to be called first followed by the BASE (BSD) security mechanism, the configuration line for siad_init() might look like the following:

```
siad_init=(DCE,/usr/shlib/libdcesia.so)(BSD,libc.so)
```

Layered security products must deliver pretested matrix.conf files on their kits. The modification of an SIA matrix.conf file must be followed by a reboot. System administrators must never be required to edit a live matrix.conf file hand.

See Security Administration for a more detailed discussion of the matrix.conf file.

6.15 Security Mechanism-Dependent Interface

Security mechanisms are required to provide all of the siad_*() entry points. (See Table 6–3.) The default stub routine should return SIADFAIL. With the exception of the session routines, no stubs should ever be called in the /etc/sia/matrix.conf file. The session routines must all have the same number and order of mechanism to keep the mechanism index (mechind) consistent. However, if an error in configuration occurs, the stub routines deliver the appropriate SIADFAIL response.

The order of security mechanisms in the /etc/sia/matrix.conf file is the same for each class of interfaces. Therefore, if a security mechanism supports session processing, it is called in the same order for all the session related interfaces.

The layered security mechanism should provide a set of private entry points prefixed by *mechanism_name*__ for each of the siad_*() entries used for internal calls within the mechanism to siad_*() routines. An example of this is in the BASE mechanism in libc. To assure that the BASE mechanism is calling its own siad_getpwuid() routine, a separate entry point is created and called from the siad_getpwuid() entry as follows:

```
int siad_getpwuid(uid_t uid, struct passwd *result, \
                                   char *buffer, int buflen)
        {
        return(bsd_siad_getpwuid(uid,result,buffer,buflen));
        }
```

```
static int bsd_siad_getpwuid(uid_t uid, struct passwd *result, \
                                       char *buffer, int buflen)
        {
        /* The BSD security mechanism siad_getpwuid() routine */
        }
```

If the convention of supplying internal names is used for all of the `siad_*()` entry points, a layered security mechanism can then produce a separate library containing all the security mechanism-dependent code. This leaves the configured shared library with only stubs that call the other library.

Security mechanisms generally fall into two categories: local and distributed. The local security mechanism is responsible for establishing all of the local context required to establish a session on the local system. There are two local security mechanisms in Tru64 UNIX: the BASE mechanism and the ENHANCED mechanism.

Distributed mechanisms, like DCE, are more concerned with establishing distributed session context like Kerberos tickets. However, the distributed security mechanism may provide some local context that can be used by the local security mechanism. The distributed security mechanism may also provide a sufficiently strong authentication to allow a local mechanism to trust it for authentication. This notion of one mechanism trusting another is called vouching and allows the user to be authenticated only once to establish a login session. Local mechanisms should always be configured last in the calling sequences.

All of the SIA capabilities listed in this section can be configured to use multiple security mechanisms.

6.16 Single-User Mode

If you want to have your own single-user security mode, you need to rebuild and replace the commands and utilities affected, such as any statically linked binaries found in `/sbin`. This can be accomplished by providing an `siad_*()` routine library to precede `libc` in the link order for the affected commands.

The new routines need to override the `__siad_*()` routines, as opposed to the `siad_*()` routines. The `siad_*()` naming convention is the weak symbol entry point, while the `__siad_*()` convention is the strong symbol entry point that is actually used. See Section 6.17 for more information about routine-naming conventions.

6.17 Symbol Preemption for SIA Routines

This section describes the naming convention for routines (added by developers) that must be followed to stay in compliance with ANSI C routine naming rules.

6.17.1 Overview of the Symbol Preemption Problem

Overriding the symbols used by the SIA routines in `libc` is not as simple as providing routines named the same as the SIA routines (such as, `siad_ses_init()`) in a library loaded before `libc.a`. This is because of the ANSI C convention for `libc` routine names and the symbols that must be reserved to the user.

A conflict exists between the requirements of ANSI C and the expectations of the application developers regarding what entry points can exist in the `libc.a` and `libc.so` libraries. The ANSI C standard lists the symbols allowed, and the only other symbols that are valid must be of the "reserved-to-vendor" form. That is, they must start with two underscores, or one underscore and a capital letter. This set of symbols is limited, and does not meet the expectations of the general user community.

6.17.2 The Tru64 UNIX Solution

To satisfy both ANSI C and developer expectations, Tru64 UNIX uses "strong" and "weak" symbols to provide the additional names. If a routine such as `bcopy()` is not allowed by ANSI C, it has a weak symbol named `bcopy()` and a strong symbol named `__bcopy()`.

The weak symbol can be preempted by the user with no effect on the `bcopy()` routine within `libc`, because the library uses the strong symbols for these "namespace-protected" routines.

For the SIA routines, this means that there is a weak symbol for `siad_ses_init` which is normally bound to the strong symbol `__siad_ses_init()`. If other code already uses the symbol `siad_ses_init()`, only the binding of the weak symbol is affected.

The SIA code in `libc` references the strong symbol `__siad_ses_init()` for its own uses. Thus, to override the default BASE security mechanism for single-user mode, it is necessary to provide a replacement for the `__siad_ses_init()` routine.

For a library that is only dynamically loaded under the control of the SIA routines and the `/etc/sia/matrix.conf` file, it is only necessary to provide the `siad_ses_init()` form of the symbol name. If the dynamically loaded library is only used through the `matrix.conf` file, it is acceptable to provide both forms of symbols. This simplifies the code, but is not safe if the library usage ever changes to require that the library be linked against, not just dynamically loaded.

6.17.3 Replacing the Single-User Environment

Example 6–5 shows the code to use if a security mechanism library developer needs to replace the single-user environment as well as provide a normal shared library for `matrix.conf`.

Example 6–5: Preempting Symbols in Single-User Mode

```
/* preempt libc.a symbols in single-user mode */
#ifdef SINGLE_USER
# pragma weak siad_ses_init = __siad_ses_init
# define siad_ses_init __siad_ses_init
#endif
#include <sia.h>
#include <siad.h>
```

The single-user (static) library modules are then compiled as follows:

```
% cc -DSINGLE_USER ...
```

This keeps the shared library from interfering with the `libc.so` symbols, but allows the preemption of the `libc.a` symbols for the nonshared images used in single-user mode. The nonshared images are then built with the replacement mechanism library supplied to the linker before `libc.a` as in the following example:

```
% cc -non_shared -o passwd passwd.o -ldemo_mech
```

The shared library is built in the normal fashion.

7 Programming with ACLs

This chapter contains the following information:

- ACL overview
- ACL data representations
- ACL library routines
- ACL rules
- ACL creation example
- ACL inheritance example

7.1 ACL Overview

Tru64 UNIX access control lists (ACLs) are an optional extension to the discretionary access control (DAC) traditionaly provided on a UNIX system. Traditional UNIX DAC is the traditional UNIX permission bits; ACLs are an extension of the UNIX permission bits. A file or directory that has only the permission bits may be considered an object with an ACL containing only the three required or base entries that correspond to the `usr`, `group`, and `other` permission bits.

There are two types of ACLs:

- An **access ACL** is associated with a file or directory, and is used to determine if a process may access the file or directory.
- **Default ACLs** are associated with a directory. Default ACLs are used to determine the ACLs applied to new files and subdirectories created in the given directory. See Section 7.6 for more information.

The Tru64 UNIX ACL implementation is based on Draft 13 with some Draft 15 extensions of the POSIX P1003.6 standard.

ACLs can be applied to any file or directory on a file system that supports property lists. The file systems that support property lists are:

- UFS
- AdvFS
- NFS (between Tru64 UNIX systems)

ACLs can be applied even if ACL processing is not enabled on the system; however, ACL access checks and default ACL inheritance do not take place.

See Security Administration for a more detailed description of using and administering ACLs. See the `acl`(4) reference page for more information on using and programming with ACLs. See the `proplist`(4) reference page for more information on property lists.

7.2 ACL Data Representations

An ACL has an internal and an external representation. The external representation consists of text and is used to enter and display ACLs. Library routines manipulate ACLs in working storage in an internal representation that is only indirectly accessible to the calling routine. This internal representation can be interpreted with the `acl.h` header file.

7.2.1 Internal Data Representation

The ACL routines manipulate the working storage representation, which is a set of opaque data structures for ACLs and ACL entries. Your program should operate on these data structures only through the defined routines. Because the working storage data structures are subject to change, the interface is the only reliable way to access the data.

The working storage representation is not contiguous in memory. Also, a program cannot determine the sizes of ACL entries and ACL descriptors. The working storage data structures contain internal pointer references and are therefore meaningless if passed between processes or stored in a file. A program can convert the working storage representation of an ACL to other representations of an ACL.

The two types most commonly used to access the opaque data are `acl_t`, a pointer to type `acl` (the ACL structure), and `acl_entry_t`, a pointer to an ACL entry structure.

Note

The structures in the following sections are opaque internal data structures that are subject to change. Always use the defined types and the supplied library routines to access these structures.

The internal representation uses the following basic types and data structures.

7.2.1.1 typedef struct acl *acl_t;

The `acl_t` type is used to specify an internal (working storage) format ACL.

```
struct acl {
  int         acl_magic;      /* validation member */
  int         acl_num;        /* number of actual acl entries */
  int         acl_alloc_size; /* size available in the acl */
  acl_entry_t  acl_current;    /* pointer to current entry in */
  acl_entry_t  acl_first;      /* pointer to ACL linked list */
  attribute_t  *attr_data;     /* Pointer to the attr data */
};
```

7.2.1.2 typedef struct acl_entry *acl_entry_t;

The `acl_entry_t` type is used to specify an entry within an ACL.

```
struct acl_entry{
   acle_t            *entry;
   void              *head;
   struct acl_entry  *next;
   struct acl_entry  *prev;
   int               acl_magic;
   int               size;
};
```

7.2.1.3 typedef uint_t acl_type_t;

The ACL types supported are as follows:

```
#define  ACL_TYPE_ACC  0
#define  ACL_TYPE_ACCESS  ACL_TYPE_ACC
 /* The ACL is an access ACL.  The property list
    entry name for an access ACL is "DEC_ACL_ACC" */

#define  ACL_TYPE_DEF  1
#define ACL_TYPE_DEFAULT  ACL_TYPE_DEF
 /* The ACL is a default access ACL. The property list
    entry name for a default access ACL is "DEC_ACL_ACC" */
#define  ACL_TYPE_DEF_DIR  2
#define ACL_TYPE_DEFAULT_DIR  ACL_TYPE_DEF_DIR
 /* The ACL is a default directory ACL. The property list
    entry name for a default directory ACL is "DEC_ACL_DEF_DIR" */
```

`acl_type_t` is used to specify the ACL type.

7.2.1.4 typedef uint acl_tag_t;

The `acl_tag_t` type is used to specify the tag (the type) of an ACL entry. ACL entries with a tag type of `ACL_USER` or `ACL_GROUP` also have an associated tag qualifier. The tag qualifier is the ID of the user or group. The ACL entry tag types supported are:

```
#define  ACL_USER_OBJ  0
 /* entry that equates to the owning user permission bits. */
#define  ACL_GROUP_OBJ  1
 /* entry that equates to the owning group permission bits. */
#define  ACL_OTHER  2
#define  ACL_OTHER_OBJ  ACL_OTHER
 /* entry that equates to the other permission bits. */
#define  ACL_USER  23
 /* entry specifying permissions for a given user.  */
#define  ACL_GROUP  24
 /*  entry specifying permissions for a given group.  */
```

7.2.1.5 typedef uint_t acl_perm_t;

The `acl_perm_t` permission bit definitions are as follows:

```
#define  ACL_EXECUTE  0X001
#define  ACL_WRITE    0X002
#define  ACL_READ     0X004
```

7.2.1.6 typedef acl_perm_t *acl_permset_t;

The `acl_permset_t` type is used to point to the permissions assigned to an ACL entry.

7.2.1.7 Contiguous Internal Representation ACL

There is also a contiguous persistent data type for an ACL. This representation should be used only when the internal format ACL must persist between processes.

7.2.2 External Representation

The human-readable external representation of an ACL consists of a sequence of lines, each of which is terminated by a new-line character. The POSIX routines use the external representation when converting between the working storage representation and the text package.

The external representation is described in Security Administration. Table 7–1 shows the structure of individual entries.

Table 7–1: ACL Entry External Representation

Entry Type	`acl_tag_t` Value	Entry
base user	`USER_OBJ`	user::*perms*
base group	`GROUP_OBJ`	group::*perms*
base other	`OTHER_OBJ`	other::*perms*
user	`USER`	user:*user_name*:*perms*
group	`GROUP`	group:*group_name*:*perms*

7.3 ACL Library Routines

The ACL routines are contained in the `libpacl.a` library. The ACL library routines are based on Draft 13 of the POSIX P1003.6 standard. See the reference page for each individual routine for detailed information.

The following routines are used to get, set, and validate ACLs:

`acl_valid()`	Checks the specified internal representation ACL for valid format.
`acl_delete_def_fd()`	Deletes the default access ACL from the designated directory using the file descriptor.
`acl_delete_def_file()`	Deletes the default access ACL from the designated directory.
`acl_get_fd()`	Retrieves the internal representation of the specified ACL type associated with the specific file or directory using the file descriptor.
`acl_get_file()`	Retrieves the internal representation of the specified ACL type associated with the specific file or directory.
`acl_set_fd()`	Sets the specified ACL type on the given file or directory to the specified ACL internal representation using the file descriptor.

`acl_set_file()`	Sets the specified ACL type on the given file or directory to the specified ACL internal representation.

The following routines retrieve and manipulate ACL entries:

`acl_copy_entry()`	Copies an ACL entry into the memory provided.
`acl_create_entry()`	Creates an empty ACL entry for the given ACL, allocating memory as necessary.
`acl_delete_entry()`	Deletes the designated ACL entry from an ACL.
`acl_first_entry()`	Resets the current ACL entry so that the next call to `acl_get_entry()` returns the first entry.
`acl_get_entry()`	Returns a pointer to the next ACL entry of the given ACL.

The following routines retrieve and manipulate fields in an ACL entry:

`acl_add_perm()`	Adds a permission to a set of permissions belonging to an ACL entry.
`acl_clear_perm()`	Clears a permission in a given ACL entry.
`acl_delete_perm()`	Removes permissions from a set of permissions belonging to an ACL entry.
`acl_get_permset()`	Copies the permissions from a given ACL entry to the location provided.
`acl_get_qualifier()`	Returns a pointer to the tag qualifier (ID) associated with a given ACL entry.
`acl_get_tag_type()`	Copies the tag (type) from the given ACL entry to the location provided.
`acl_set_permset()`	Sets the permissions in a given ACL entry to the given permissions.

`acl_set_qualifier()`	Sets the tag qualifier (ID) of the specified ACL entry to the given UID or GID.
`acl_set_tag_type()`	Sets the tag (type) of the specified ACL entry to the given type.

The following routines manage working storage for the ACL manipulation:

`acl_free()`	Releases all working storage associated with the given ACL.
`acl_free_qualifier()`	Releases working storage associated with the given tag qualifier.
`acl_free_text()`	Releases the buffer associated with the given external representation (text) ACL.
`acl_init()`	Allocates and initializes ACL internal representation working storage.
`acl_copy_ext()`	Copies the working storage internal format ACL data to the contiguous persistent ACL format.
`acl_copy_int()`	Copies contiguous persistent ACL data to working storage.
`acl_dup()`	Creates a copy of the designated ACL. The copy is independent of the original entry.
`acl_size()`	Calculates the size of the given ACL.

The following routines convert ACLs between external and internal representations:

`acl_from_text()`	Creates an internal representation ACL from the given external representation (text) ACL.
`acl_to_text()`	Creates an external representation (text) ACL from the given internal representation ACL.

7.4 ACL Rules

Some interactions between the ACL and the UNIX permissions are subtle. Unless you understand the interaction between ACL routines and the system calls that manipulate UNIX DAC attributes, you might get different permissions than you intended.

The following sections describe rules for programs that handle ACLs.

7.4.1 Object Creation

If ACLs are enabled and are supported on the file system, the `open()`, `creat()`, and `mkdir()` functions perform ACL inheritance when creating a file or directory. See for a description of ACL inheritance.

When ACL inheritance is performed, the permissions on a created file come from the mode you provide and the inherited ACL, not the `umask`. Therefore, your program must set the mode when creating files and directories. The program must not depend on `umask` to protect the files and directories.

When copying one file to another, it is a common practice for a program to create a new file and propagate the owner, group, and mode. If the source file has an ACL, your program should propagate that ACL to the target file in all cases where the mode is propagated.

7.4.2 ACL Replication

Programs that replicate permissions must preserve the ACL. The discretionary protection of a file or directory is no longer described by the owner, group, and permissions; it includes the ACL which is a superset of the permissions. Neglecting to copy the ACL could allow unintended access to the file or directory.

7.4.3 ACL Validity

Any ACL you create must be valid according to the following POSIX ACL rules:

- It must have at least the three base entries
- The user entries must have unique valid qualifiers
- The group entries must have unique valid qualifiers
- The user and group identifiers must be valid

You can use the `acl_valid()` routine to check your ACLs.

7.5 ACL Creation Example

Assume that you want to set a file's access ACL to the following permissions:

```
user::rwx
user:june:r-x
user:sally:r-x
group::rwx
group:mktg:rwx
other::r-x
```

The following code takes a tabular form of the ACL, creates a working storage representation of the ACL, and applies it to a file. If you extract the following code into a file named `acl_example.c`, use the following command to compile it:

```
# cc -o acl_example -lpacl -lsecurity acl_example.c
```

```
#include <unistd.h>
#include <sys/types.h>
#include <prot.h>
#include <errno.h>
#include <sys/acl.h>

struct entries {
    acl_tag_t       tag_type;
    char            *qualifier;
    acl_perm_t      perms;
} table[] = {
    /* An ACL must (at a minimum) have the three base */
    /* entries that correspond to the permission bits */
    { ACL_USER_OBJ,  NULL,    ACL_READ | ACL_WRITE | ACL_EXECUTE },
    { ACL_USER,      "june",  ACL_READ | ACL_EXECUTE },
    { ACL_USER,      "sally", ACL_READ | ACL_EXECUTE },
    { ACL_GROUP_OBJ, NULL,    ACL_READ | ACL_WRITE | ACL_EXECUTE },
    { ACL_GROUP,     "mktg",  ACL_READ | ACL_WRITE | ACL_EXECUTE },
    { ACL_OTHER_OBJ, NULL,    ACL_READ | ACL_EXECUTE },
};

#define TABLE_ENTRIES (sizeof(table)/sizeof(table[0]))

main ( argc, argv )
int argc;
char *argv[];
{
    acl_t acl_p;
    acl_entry_t entry_p;
    acl_entry_t check_p;
    int     i, ret;
    uid_t   uid;
    gid_t   gid;
```

```
 /* Did the user enter a filename? */
 if (argc != 2) {
      printf("Usage: %s filename\n",argv[0]);
      exit(1);
 }

 /* Check to see if ACLs are supported and enabled for filename */
 ret = pathconf(argv[1], _PC_ACL_EXTENDED);              /*1*/
 if (ret == 1) {
      printf("  ACLs are enabled for file %s\n",argv[1]);
 }
 else if (ret == 0) {
      printf("  ACLs are supported for file %s,\n",argv[1]);
      printf("  but ACLs are not currently enabled on \n");
      printf("  the system\n");
 }
 else if ((ret == -1) && (errno == EINVAL)) {
      printf("  ACLs not supported on filesystem\n");
      exit(1);
 }
 else {
      printf("  Error checking file ACL status for \n");
      printf("  file %s, exiting...\n",argv[1]);
      perror("pathconf");
      exit(1);
 }


 /* Allocate an ACL */
 acl_p = acl_init(1024);                           /* 2 */

 /* Walk through the table creating corresponding ACL entries */
 for(i=0;i<TABLE_ENTRIES;i++) {

 /* Initialize the entry */
 entry_p = acl_create_entry(&acl_p);           /* 3 */

  /* Set the permissions for the entry */
acl_set_permset(entry_p,&table[i].perms);       /* 4 */

/* Set the user or group information for the entry */
switch(table[i].tag_type) {

   case ACL_USER:

       /* Get the uid from the user name */
       uid=pw_nametoid(table[i].qualifier);        /* 5 */
       if (uid == (uid_t) -1) {
           printf("  No translation for user name %s\n",
           table[i].qualifier);
           printf("  Exiting...\n");
           exit(1);
       }

       /* Specify this is a "USER:" entry */
       acl_set_tag_type(entry_p,table[i].tag_type);    /* 6 */

       /* Set the uid (entry qualifier) */
       acl_set_qualifier(entry_p,(void *)&uid);    /* 7 */
       break;

   case ACL_GROUP:

       /* Get the gid from the group name */
       gid=gr_nametoid(table[i].qualifier);        /* 8 */
       if (gid == (gid_t) -1) {
           printf("  No translation for group name %s\n",
           table[i].qualifier);
           printf("  Exiting...\n");
           exit(1);
       }

       /* Specify this is a "GROUP:" entry */
       acl_set_tag_type(entry_p,table[i].tag_type);

       /* Set the gid (entry qualifier) */
       acl_set_qualifier(entry_p,(void *) &gid);
       break;

   default:

       /* The three entries corresponding to the */
       /* Permission bits don’t have qualifiers */
       acl_set_tag_type(entry_p,table[i].tag_type);
       acl_set_qualifier(entry_p,NULL);
       break;
   }
}

/* Is the created ACL valid? */
if (acl_valid(acl_p, &check_p) < 0) {            /* 9 */
    printf("  Not Valid ACL\n");
    if (check_p) printf("  Duplicate entries\n");
    printf("  Exiting...\n");
    exit(1);
}

/* Set the ACL on the file */
if (acl_set_file(argv[1],ACL_TYPE_ACCESS, acl_p) < 0)
     perror("acl_set_file");
    /* Free the storage allocated for the ACL */
    acl_free(acl_p);
}
```

1. The _PC_ACL_ENABLED attribute of `pathconf`() returns the status of ACL processing for the given file.

2. This demonstrates the use of the initialization call for a working storage representation of the ACL. If the storage allocated is not big enough for all of the entries in the completed ACL, the `acl_create_entry`() routine will allocate more memory.

3. A new ACL entry is allocated with this call. The tag type, qualifier, and permissions in this new entry are unspecified.

4. The `acl_set_permset`() routine sets the permissions for the ACL entry.

5. The `pw_nametoid`() routine is an optimized mapping from user name to user ID and works with either Base or Enhanced security enabled. The `pw_nametoid`() routine is described in the `pw_mapping`(3) reference page.

6. The `acl_set_tag_type`() function sets the type for the given ACL entry. The current tag types are: ACL_USER_OBJ (owner permission bits), ACL_GROUP_OBJ (group permission bits), ACL_OTHER_OBJ (other permission bits), ACL_USER (permissions for specified user), ACL_GROUP (permissions for specified group).

7. The `acl_set_qualifier`() function sets the ID for the ACL_USER and ACL_GROUP tag types. This specifies which user or group the entry refers to. The other tag types do not require an ID.

8. The `gr_grouptoid`() routine provides an optimized mapping from group name to group ID and works with either Base or Enhanced security enabled. It is described in the `pw_mapping`(3) reference page.

9. The `acl_valid`() routine checks for missing and duplicate entries.

7.6 ACL Inheritance Example

This section shows how a program can specify a default access ACL on a directory and then describes what happens when a file and a directory are created in that directory. There is another type of default ACL called a default directory ACL. ACLs are inherited differently if a directory has a default directory ACL in addition to or in place of a default access ACL. See Security Administration for a complete description of the ACL inheritance rules.

Assume that directory /usr/john/acl_dir has the following access and default access ACLs:

```
% getacl /usr/john/acl_dir

# file: /usr/john/acl_dir
# owner: john
# group: prog
#
user::rwx
user:june:r-x
user:fred:r-x
group::rwx
group:mktg:rwx
other::r-x

% getacl -d /usr/john/acl_dir

# file: /usr/john/acl_dir
# owner: john
# group: prog
#
user::rwx
user:june:r-x
user:sally:r-x
group::rwx
group:mktg:rwx
other::rwx
```

The following program can be used to update the default access ACL on a directory to remove read and write permissions from a group entry and then create a regular file and a directory in the given directory. If you extract the following code into a file named acl_inheritance.c, it can be compiled with the following command:

```
% cc -o acl_inheritance -lpacl -lsecurity acl_inheritance.c
```

```
#include <unistd.h>
#include <sys/types.h>
#include <prot.h>
#include <errno.h>
#include <sys/acl.h>

#define REGULAR_FILE "regular"
#define DIRECTORY_FILE "dir"

main (argc, argv)
int argc;
char *argv[];
{
    acl_permset_t  acl_permset;
```

```
gid_t          *qualifier = NULL;
acl_tag_t      tag_type;
acl_t          acl;
acl_entry_t    acl_entry;
gid_t          my_gid;
int            ret;
char           pathname[PATH_MAX + 1];
int            fd;

/* Did the user enter a directory name and group name? */
if (argc != 3) {
   printf("Usage: %s directory group\n",argv[0]);
   exit(1);
}

/* Map the group name to a gid */
my_gid = gr_nametoid(argv[2]);
if (my_gid == (gid_t) -1) {
    printf("No translation for group %s\n",argv[2]);
    exit(1);
}

/* Read the default ACL from the directory */
acl = acl_get_file(argv[1], ACL_TYPE_DEFAULT);
if (!acl) {
  if (errno) {
     perror("acl_get_file");
  }
  else {
     printf("No default ACL found on %s\n", argv[1]);
  }
  exit(1);
}

ret = acl_first_entry(acl);
if (ret) {
   perror("acl_first_entry");
   exit(1);
}

/* Scan the ACL looking for the entry */
while (acl_entry = acl_get_entry(acl)) {

   /* retrieve the entry type */
   ret = acl_get_tag_type(acl_entry, &tag_type);
   if (ret) {
      perror("acl_get_tag_type");
      exit(1);
   }
   if (tag_type != ACL_GROUP) continue;

   qualifier = (gid_t *)acl_get_qualifier(acl_entry);
   if (!qualifier) {
      perror("acl_get_qualifier");
      exit(1);
   }

   /* Check for appropriate entry */
   if (*qualifier != my_gid) continue;

   ret = acl_get_permset(acl_entry, &acl_permset);
   if (ret) {
      perror("acl_get_permset");
      exit(1);
   }

   *acl_permset = *acl_permset & ~(ACL_READ | ACL_WRITE);

   ret = acl_set_permset(acl_entry, acl_permset);
   if (ret) {
      perror("acl_set_permset");
      exit(1);
   }

   ret = acl_set_file(argv[1], ACL_TYPE_DEFAULT, acl);
   if (ret) {
      perror("acl_set_file");
      exit(1);
   }

   break;

}

if (!acl_entry) {
   if (errno) {
      perror("acl_get_entry");
   }
   else {
      printf("ACL entry for %s not found\n", argv[2]);
   }
   exit(1);
}

/* Create the regular file */
sprintf(pathname, "%s/%s", argv[1], REGULAR_FILE);

fd = creat(pathname, 0644);
if (fd == -1) {
        perror("creat");
        exit(1);
    }
    close(fd);

    /* Create the directory */
    sprintf(pathname, "%s/%s", argv[1], DIRECTORY_FILE);

    ret = mkdir(pathname, 0700);
    if (ret == -1) {
        perror("mkdir");
        exit(1);
    }
}
```

When you run the previous example program, it removes the read and write permissions from the `mktg` group in the default ACL shown above. The program then creates a regular file and a directory in that directory to demonstrate ACL inheritance. Enter the following command to execute the example program:

```
% ./acl_inheritance /usr/john/acl_dir mktg
```

When the example program is executed, the access ACL on the newly created file and the access and default access ACLs on the newly created directory are as follows:

```
% getacl /usr/john/acl_dir/regular
# file: /usr/john/acl_dir/regular
# owner: john
# group: prog
#
user::rw-
user:june:r-x
user:sally:r-x
group::r--
group:mktg:--x
other::r--
```

Note that the permissions for the owning user, the owning group, and other are set to the logical AND of the default access ACL and the mode specified with the `creat()` call. The `umask` is not used when ACL inheritance takes place. The other entries are taken from the default access ACL of the parent directory, not the access ACL.

```
% getacl /usr/john/acl_dir/dir
# file: /usr/john/acl_dir/dir
# owner: john
# group: prog
#
user::rwx
user:june:r-x
user:sally:r-x
group::---
group:mktg:--x
```

```
other::---
```

Note that the access ACL inheritance rules for a subdirectory created in a directory that has a default access ACL are the same as those for a file. This is true only if there is not a default directory ACL on the parent directory in addition to the default access ACL.

The following command line displays the default access ACL:

```
% getacl -d /usr/john/acl_dir/dir
# file: /usr/john/acl_dir/dir
# owner: john
# group: prog
#
user::rwx
user:june:r-x
user:sally:r-x
group::rwx
group:mktg:--x
other::rwx
```

Note that the default ACL is inherited from the directory's parent.

8 GSS-API

The Generic Security Service Application Program Interface (GSS-API) functions enable applications in a distributed network environment to use the following security services on the network:

- Authentication — The application can verify the identity of a user or service.
- Integrity — The application can detect message tampering or corruption when it receives the message.
- Confidentiality — A message can be encrypted to render it unintelligible to eavesdroppers during transmission.

This chapter contains the following information:

- GSS-API overview
- Application security SDK
- Application security SDK functions
- Best practices
- Building a portable application

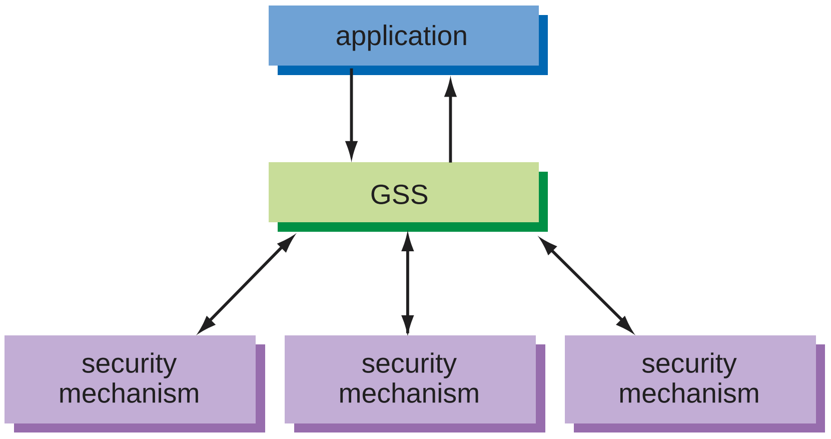
8.1 GSS-API Overview

The GSS-API is a standard programming interface that defines a set of generic C functions that can be used to secure distributed applications. The GSS-API has two primary design goals which are fundamental to its operation:

- Security mechanism independence
- Transport protocol independence

Because it is an open standard, the GSS-API has been designed to be generic so that as security and network technologies evolve, the API does not have to change.

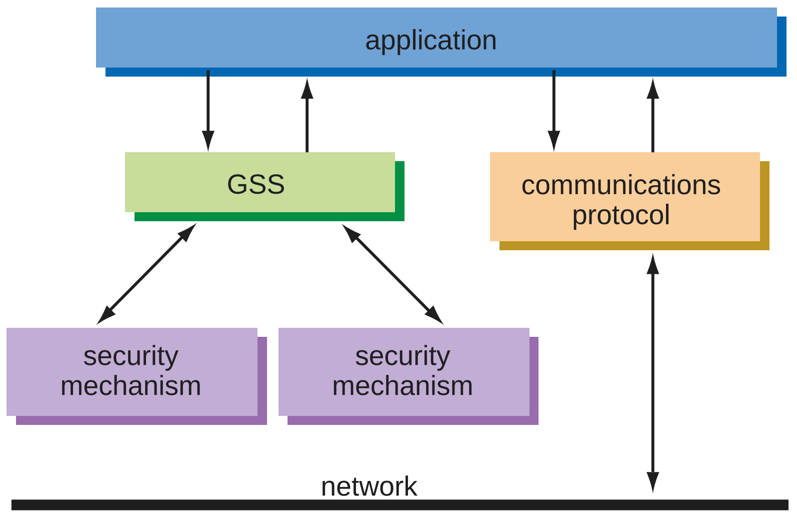
The GSS-API supports a wide range of underlying security mechanisms and technologies using the following architecture.



ZK-1825U-AI

A *security mechanism* is a method of providing security (such as Kerberos or public-key encryption). It is not only the cryptographic technology used, but also the syntax and semantics of the data that the technology employs. An application secured using the GSS-API standard may use one or more security mechanisms.

The GSS-API can be used in a broad range of network environments (for example, TCP/IP, SNA, and DECnet). The standard was not designed to provide a transport mechanism. Rather, the design provides security over an arbitrary network transport. The transport must be provided by the application. The communications protocol can be an interprocess communications path or a series of networks.



ZK-1826U-AI

GSS-API functions return information to the application, which then sends the information across the communications protocol in use. The other side of the distributed application passes the information to the GSS-API library.

For a developer securing an application using the GSS-API standard, these design goals of mechanism and transport independence provide a consistent interface that is independent of the underlying hardware and software platform — a one-time programming investment. The investment in modifications to secure an application remains constant even as the technologies evolve.

8.1.1 GSS-API Assumptions

The GSS-API standard makes the following assumptions:

- The application is distributed.

  The GSS-API standard assumes that the application is a distributed network application or divided into two parts using a peer-to-peer or a initiator-acceptor relationship.

- The source code can be modified.

  The GSS-API standard assumes that you can incorporate GSS-API functions in the application source code.

- The application guarantees token delivery.

  A token is an opaque data object returned by the GSS-API that an application needs to communicate with its peer. The GSS-API standard assumes that your application can deliver tokens generated during context establishment and context termination in the order in which they are generated.

- The application deallocates its data objects.

  The GSS-API standard assumes that if an application allocated a data object, it is responsible for deallocating it. If a data object is returned by a GSS-API function, the application must use the corresponding GSS-API function to release the object, thereby deallocating it. Otherwise, there can be a memory leak or memory fault in the application. Failure to use the proper functions for deallocation may create a situation where the security network may be compromised.

8.1.2 Further Information

The GSS-API is an industry standard created as part of the ongoing Internet RFC (Request For Comments) process.

Application Security SDK is based on the following RFCs:

- RFC 2078 “Generic Security Service Application Program Interface, Version 2, Update 1” September 3, 1998
- “Generic Security Service API Version 2: C-bindings,” August 7, 1998
- RFC 1964 “The Kerberos Version 5 GSS-API Mechanism,” June 1996
- RFC 1510 “The Kerberos Network Authentication Service (V5),” September 1993
- Internet Draft “Public Key Cryptography for Initial Authentication in Kerberos” (updates RFC 1510)

  File name: `draft-ietf-cat-kerberos-pk-init-07.txt`

The GSS-API standard is monitored by the Common Authentication Technology Internet Engineering Task Force (CAT-IETF) working group. For more information on the GSS-API, see the IETF web site at **`www.ietf.org`**.

8.2 Application Security SDK

HP implements the GSS-API functions through the Application Security Service Developers Kit (SDK). The Application Security SDK is an implementation of the GSS-API version 2.0 that provides several unique contributions to the GSS-API standard.

- Application Security SDK supports the Kerberos 5 mechanism. The sample code and documentation provide clear instructions for securing distributed applications using Kerberos 5 with the GSS-API, complete with considerations that are specific to this security mechanism.

- Application Security SDK has many additional functions to enhance the capabilities of the GSS-API. These functions, termed *extensions*, permit additional Kerberos 5 security features, including:
  - complete support for Triple DES (DES3), including encryption of authentication tokens and user data. HP recommends that all data be encrypted using DES3, which provides significantly increased security in comparison to DES encryption.
  - programmatic support for acquiring the initial credentials needed before the initiator application can establish a security context. Using the HP extension function eliminates the need to use kinit, *Active***TRUST** SignOn (available from CyberSafe Corporation), or HP Single SignOn to obtain these initial credentials.
  - recognition of hardware authentication devices, such as token cards. This adds an extra level of security at the client level during authentication requests.
  - ability to renew Kerberos 5 credentials. This prolongs the lifetime of an existing valid credential, and reduces the number of times the number of times the user is required to provide their username and password.

8.3 Application Security SDK Functions

Application Security SDK functions can be grouped into categories, of which only a subset is required to secure a distributed application. The functions can be divided according to these categories:

- Name Management Functions — Describes functions used to manipulate the internal and external names used by the GSS-API.
- Credential Management Functions — Describes functions used to acquire, query, and release credentials. This category also contains the HP extension functions included in Application Security SDK that are not part of the GSS-API standard. These functions are used to implement initial authentication, and include support for DES3 encryption, hardware authentication, and credential renewal.
- Security Context Management Functions — Describes functions used to initiate, accept, export, import, query and delete security contexts.
- Message Functions — Describes functions used to protect messages by ensuring data integrity, data origin, and, optionally, confidentiality.
- Miscellaneous Functions — Describes other support functions used to display status, release buffers, and manipulate object identifier (OID) sets.

- V1 Compliance Functions — Describes the functions that are supported for GSS-API Version 1 interoperability. These functions have been replaced for GSS-API Version 2.

8.3.1 Name Management Functions

The GSS-API uses names to identify entities in a network, just as a person uses a username to log in to a system on the network. The username identifies the person according to the system's naming structure. In the GSS-API standard, names identify an application or person using the application.

In the Kerberos 5 mechanism, names translate to principals. A *principal* is any user, client, network service, application, or system that has shared a secret with Kerberos (usually a password). Principals must have unique names within a realm, as well as an associated key.

The GSS-API utilizes four forms of names: external, exported, internal, and mechanism names. Functions are provided to convert between the various name forms.

- External names are text strings that are used as input to the GSS-API standard.
- Exported names are octet strings in a standard format, which are produced by a GSS-API function for use in name comparisons outside the standard.
- Internal names are opaque. This means the name cannot be displayed in an error message, for example. This type of name is used for all purposes internal to the GSS-API. An example of the usage with the Kerberos 5 mechanism is locating a credential belong to a specific username.
- Mechanism names are a special case of internal names that are mechanism-specific; meaning they are specific to a single mechanism. For single mechanism support, as in the case of Application Security SDK, the internal name and mechanism name are identical.

In the GSS-API, names are used to:

- Acquire credentials — Each side of an application imports its name into the GSS-API internal format for acquiring credentials.
- Establish a security context — The initiator identifies by name the acceptor with which it wants to establish a security context.
- Compare names — After the security context is established, name comparison may be required. This can be inside or outside the GSS-API standard, provided the proper name format is used.

– Inside the GSS-API standard, the internal name can be used for comparisons (using `gss_compare_name( )`).

– Outside the GSS-API standard, the exported name should be used for comparisons. The acceptor, for example, may want to compare the initiator's name against an access control list (ACL). There is no function call provided in the API to perform this comparison.

The name of the application or person using the application does not follow a syntax, structure, or convention defined by the GSS-API. The application's name is dependent on the application itself. Because the GSS-API is generic, it is possible to provide names in different formats. These different methods to represent names are called name types, and are passed as object identifiers (OIDs).

The application can also use the GSS-API default for its name. This eliminates the requirement to provide the particular name, and avoids the need to pass the OID. In this case, the name is dependent on the GSS-API implementation.

GSS-API Function	Description
`gss_canonicalize_name( )`	Convert an internal name to a mechanism specific name.
`gss_compare_name( )`	Compares two internal form names for equality.
`gss_display_name( )`	Translates a name to human readable or printable form. The name's format is specific to the GSS-API implementation. Therefore, do not compare the printable form of names returned from `gss_display_name( )` with other names. Use the function `gss_export_name( )` to generate the names required for comparison purpose. Do not import this name using `gss_import_name( )`. The printable form generated by this function is placed in a buffer that must be released by calling `gss_release_buffer( )`.
`gss_duplicate_name( )`	Creates a copy of an existing internal form name.
`gss_export_name( )`	Converts an internal name to an export form (a string), which is in a standard format.

GSS-API Function	Description
`gss_import_name( )`	Converts the application's name (external form) into the GSS-API internal form. The application passes in an object identifier (OID) that specifies how to parse its name. For example, the name could be a UNIX *`uid`* or login name. The name is returned in a structure that must be released by calling `gss_release_name( )`. You cannot call `gss_import_name( )` and specify the default name. You can specify the default name only when acquiring credentials with `gss_acquire_cred( )`.
`gss_inquire_mechs_for_name( )`	Identifies the mechanism that supports the indicated name type. In Application Security SDK, this is always Kerberos 5.
`gss_inquire_names_for_mech( )`	Lists the name types supported by the specified mechanism. In Application Security SDK, this is always Kerberos 5 supported name types.
`gss_release_name( )`	Frees storage of the GSS-API name allocated when `gss_import_name( )` is called.

8.3.1.1 Default Names and Syntax

Kerberos principal names are of the following form: *`name/instance@REALM`*, where the *`/instance`* and *`@REALM`* parts are optional. If the *`/instance`* is omitted, an empty instance is assumed. If the *`@REALM`* is omitted, the default realm is assumed. Multiple instances are allowed, each separated by a `/`.

If the application uses the GSS-API *default* for its name when acquiring credentials using the Kerberos 5 mechanism, the name used is the *default principal* from the default credential in the default credentials cache, if the cache exists. If it does not exist, the default name is determined according to the following rules:

- The default principal for a user (that is, a person) is the user's login name
- The default principal for a service is `host/`*`fqdn@REALM`*, where *`fqdn`* is the fully qualified domain name of the system on which the service is running and *`REALM`* is the system's default realm.

  HP recommends that for additional security, a service have its own principal name rather than share the use of the system's default `host/` principal name. For example, a secure FTP service might use the principal name `ftp/`*`fqdn@REALM`* or `ftp`*`@REALM`*, (skipping the *`fqdn`* instance); a secure telnet service might use the principal name `telnet/`*`fqdn@REALM`* or `telnet`*`@REALM`*.

When an application calls `gss_import_name( )` and does not use the default input format, it passes in an object identifier (OID) that specifies how to parse the application's name. For example, the `GSS_KRB5_NT_HOSTBASED_SERVICE_NAME` OID specifies the parsing syntax for a name of the format *`service@host`*. The `NT` in this parameter stands for name type.

8.3.2 Credential Management Functions

Credentials are used by an application to prove its identity. Each side of a distributed application needs to obtain credentials, which are used to initiate and accept a security context.

In Kerberos 5, there are three types of credentials:

- initial tickets — This type of credential is received from the Kerberos security server after initial authentication of a user principal. An initial ticket is properly termed a *ticket-granting-ticket* or TGT, and is required by the initiator application.
- service tickets — This ticket permits a user principal to gain access to a protected application. A TGT must be acquired before a service ticket can be received.
- service key table entry — This is a copy of the secret key associated with the principal in the principal database. This key must be extracted from the principal database to the service key table file using a database administration program. The key in the service key table file is used by an acceptor application instead of acquiring initial tickets.

For more information on Kerberos 5 security terms and infrastructure, see Security Primer.

When using the Kerberos 5 mechanism with the GSS-API standard, initial tickets are held and presented by a Kerberos principal as proof that it is what or who it claims to be. These credentials are used by the principal to prove that the trusted KDC has been contacted for initial authentication. Typically, initial tickets provided by the security server are placed in a central storage location called the credential cache.

In the GSS-API, initial tickets are retrieved from storage and used to establish a security context. In the standard GSS-API with the Kerberos 5 mechanism, the initiator application must already have obtained an initial ticket (TGT) prior to calling `gss_acquire_cred( )`. A user can obtain an initial ticket by running `kinit` or an equivalent application (for example, Single SignOn). The acceptor application will use the key stored in the service key table file to verify its identity.

The Application Security SDK provides additional functions called the HP extension functions that are used to perform a variety of tasks, including:

- Obtaining initial credentials before initiating a context. This can be done either through a program prompt to obtain secret information from the user, or by allowing the initiator application to also use a secret key stored in a service key table file on the initiator's host.
- Retrieving public-key credentials from a smart card device to obtain a TGT.
- Requesting initial credentials with specific options such as forwardable or renewable credentials.
- Renewing credentials before expiration to prolong the usable lifetime.
- Releasing buffers associated with credentials after they are no longer needed.

Using these special HP extension functions, the initial credential is received by the calling application, and placed in a credentials cache. The initial credential then can be retrieved using GSS-API functions, and be used to obtain the service tickets required for establishing the security context.

GSS-API Function	Description
`gss_acquire_cred( )`	Retrieves initial credentials from a storage location for use by an application. This step is required before a context can be established. The credential must exist before the function can be called.
`gss_add_cred( )`	Constructs credentials incrementally. This function is used when multiple mechanisms are supported. For single-mechanism implementations, HP recommends using the function `gss_acquire_cred( )`.
`gss_inquire_cred( )`	Returns information about credentials (for example, the list of mechanisms supported by this credential, credential type, and the lifetime of the credential). For Kerberos 5, this information is about the initial credential.

GSS-API Function	Description
`gss_inquire_cred_by_mech( )`	Obtains information about an existing credential specific to a particular mechanism. For Application Security SDK, information is obtained from a TGT obtained from the Kerberos security server.
`gss_release_cred( )`	Releases credentials from the application after use. Any credentials acquired by calling `gss_acquire_cred( )` or `gss_add_cred( )` must be released using this function. For Kerberos 5, this does not affect the initial credential entry in the credentials cache.

These functions may block pending exchanges between network entities such as authentication servers or network file systems. Blocking depends on the characteristics of the operating system and the security mechanism being used. In the Kerberos 5 mechanism, the credential management functions may block pending exchanges between other network entities such as directories or authentication servers.

8.3.2.1 Acquiring Initial Credentials

In the GSS-API standard before attempting to establish a context, the initiator application is required to have obtained initial credentials, and placed them in a central storage location called the credentials cache. The credentials are then retrieved from the cache using the function `gss_acquire_cred( )`.

8.3.2.1.1 Initiator Applications

Application Security SDK provides a special function to perform initial authentication for initiator applications, and places credentials into the cache for retrieval using the GSS-API standard. The function `csf_gss_acq_user( )` can be used as a replacement for the authentication process controlled by Single Sign-On, Credentials Manager, or `kinit`.

Parameters specified for this function indicate the type of secret that is being retrieved from the principal in order to prove their identity. In the case of a user principal, the secret is typically in the form of a principal name and password that are provided at a prompt, though it can be a service key table entry.

The initial tickets or TGTs can be obtained with selected Kerberos ticket flags, indicating such attributes as forwardable, or proxiable tickets, and designating the ticket lifetime and renew time.

8.3.2.1.2 Acceptor Applications

An acceptor service principal requires the use of a service key table file. This file is used to store the service principal's secret, which is in the form of a random key that was initially generated and provided by a Kerberos database administration program. This type of initial authentication is performed using the GSS-API function `gss_acquire_cred( )`.

8.3.2.1.3 DES3

Application Security SDK provides support for DES3 encryption. Using DES3 provides greatly enhanced security protection over DES encryption.

The following conditions must be met before DES3 encryption can used to encrypt messages:

- *Active***TRUST** Security Server must be configured for DES3.
- The principals for the initiating and accepting applications must have a DES3 key stored in the principal database.
- The initiator principal must obtain a TGT using DES3.
- The initiating application must indicate the use of DES3 when initiating the security context.

Note

Multiple encryption systems for a single security context are not allowed. You must use one or the other.

8.3.2.2 Credential Attributes

The function `gss_inquire_cred( )` is used to determine the properties or attributes associated with a particular credential. Some of the Kerberos information returned by this function includes the credential lifetime, the mechanisms supported, and the credential usage. The attributes assigned to an initial credential depend upon the attributes requested when acquiring the TGT. Service tickets obtained with a particular TGT inherit the attributes assigned to the TGT.

Credentials may have a defined lifetime, after which they expire. The `gss_inquire_cred( )` function can be used to determine if the credential has expired.

In some mechanism implementations, an application can no longer use credentials after they have expired. The application must release the credentials and acquire credentials again before it can continue. In the Kerberos 5 mechanism, the lifetime of the credentials is defined as the Kerberos ticket lifetime. After the credentials have expired, an application can no longer use them to establish a new security context. The application must release the credentials and acquire new ones using `csf_gss_acq_user( )` and then `gss_acquire_cred( )`.

Note

The GSS-API standard does not provide a function for renewing credentials beyond their initially established lifetime. However, you can use the HP extension function `csf_gss_renew_cred( )` to renew credentials if they have not already expired. Note that in order to renew a credential, the initial ticket must be requested from the security server with the renewable attribute flag.

8.3.2.3 Credentials Storage Location

For Kerberos 5, credentials are assumed to be stored on disk or other persistent storage medium on the local host.

For user principals, the credentials cache default location is assumed (unless the location is overridden by environment variables). The location is platform-dependent.

The default UNIX user's credentials cache file is located in `krb5/tmp/cc/krb5cc_`*`uid`* (where *`uid`* is the user identification number retrieved from the password file) unless the `CSFC5CCNAME` environment variable is set to an alternate path name.

For service principals, the default service key table file is contained in the platform-dependent locations; for UNIX systems, the default file is `/krb5/v5srvtab`, unless the CSFC5KTNAME environment variable is set to an alternate path name.

For more information on these credentials cache and service key table file settings, see the SSO *Installation and Administration Guide*.

8.3.2.4 Managing Credential Resources

Any credentials acquired with `gss_acquire_cred( )` must be released by calling `gss_release_cred( )`. The exact behavior that occurs when the credentials are released is dependent on the security mechanism used. In some cases, the internal buffers may be the only aspects of the credentials which are released. In other cases, the credentials stored on disk may be released as well. In the Kerberos 5 mechanism, only the internal buffers are released with this function.

Before deciding when to release credentials, the application must determine if it is going to be initiating or accepting security contexts more than once during the application's execution. If so, the application needs to acquire credentials and reuse them later with the initiate or accept security context functions, rather than releasing them immediately after this first use. In Kerberos 5 implementations, this provides improved performance.

8.3.3 Security Context Management Functions

A security context, once established, defines a unique conversation between each side of an application in a distributed environment. A security context is initiated by one side of the application, and then is accepted by the other side. A security context must be established prior to exchanging secured messages.

For Kerberos 5 implementations of the GSS-API standard, when a context is established, the Kerberos processes required for managing service tickets are performed. This includes obtaining, transferring and verification of these tickets. Once a context is established, it contains keys and other security parameters. The contents of a security context are used directly to apply protection to messages.

GSS-API Function	Description
`csf_gss_get_context_options ( )`	Obtains information about an existing security context, including verifying the type of encryption being used. This is an HP extension function included in Application Security SDK.
`gss_accept_sec_context( )`	Accepts a request to establish a security context from an initiator application.
`gss_context_time( )`	Indicates how much time remains in the security context's life. How this lifetime is determined and what it means is mechanism-dependent.
`gss_delete_sec_context( )`	Destroys a context when it is no longer needed. Each security context that was initiated or accepted must be destroyed using this function.
`gss_export_sec_context( )`	Transfers a security context to another process.
`gss_import_sec_context( )`	Imports a previously exported security context.

GSS-API Function	Description
`gss_init_sec_context( )`	Initiates a security context with a context acceptor application.
`gss_inquire_context( )`	Obtains information about a security context.

These functions may block pending exchanges between applications and security servers.

8.3.3.1 Identifying a Mechanism

The GSS-API standard is designed to be used in an environment where there are multiple security mechanisms available. This means that the targeted security mechanism must be identified programmatically.

One parameter passed to `gss_init_sec_context( )` is the security mechanism identifier (or `mech_type`) that the initiator desires to use. The mechanism identifier is passed as an *object identifier* (OID). Alternatively, the initiator can use the default mechanism identifier. The default that would be selected is dependent on the GSS-API implementation. For Application Security SDK, the default mechanism is Kerberos 5.

8.3.3.2 Token Exchange

During the `gss_init_sec_context( )` call, Application Security SDK searches the user's credentials cache for a service ticket. If the service ticket is not in the cache, Application Security SDK fetches a service ticket from the KDC. The returned service ticket is stored in the user's credentials cache. The GSS-API encodes the service ticket, along with other information, and returns it in a token to be sent to the other side of the application.

During the `gss_accept_sec_context( )` call, the GSS-API decodes the token and extracts and verifies the service ticket, using its secret key from the service key table file. If verification was successful and mutual authentication was requested, the GSS-API generates a token in response. If mutual authentication was not requested, no return token is generated. If an error occurred, the GSS-API may generate an error token.

8.3.3.3 Optional Security Measures

When initiating a context, the application can identify a number of optional security measures. The GSS-API implementation of these optional security measures is dependent on the security mechanism in use.

The following sections introduce the optional security measures:

- Channel Bindings

- Confidentiality and Integrity
- Replay Detection
- Out-of-Sequence Message Detection
- Mutual Authentication
- Encryption Type: DES vs. DES3
- Credentials Delegation

8.3.3.3.1 Channel Bindings

Channel bindings are used to strengthen the quality of the authentication between two sides of a distributed application during context establishment, by limiting the scope within which an intercepted context establishment token can be reused by an imposter. Channel bindings accomplish this by adding additional items of data (specified by the distributed application) into the security context.

Channel bindings are composed of two main parts:

- Address information
- Optional application data

The address information consists of an *address type* and an *address value* for both the initiator and the acceptor. The address type specifies the protocol type contained in the address value buffer. The address value is the actual address of the initiator or the acceptor (in a format that corresponds to the address type).

In Kerberos 5, when the KDC builds a service ticket, it encodes the client address into the ticket. When the initiator presents that ticket to the acceptor and channel bindings are passed to `gss_accept_sec_context( )`, the GSS-API verifies that the client address in the ticket is identical to the initiator address in the channel bindings. However, this check is not performed when using a TCP connection.

Note

The only address format supported is `GSS_C_AF_INET`.

The other component of the channel bindings data structure is application data, which the initiator can send to the other side of the application to be compared by the acceptor. Application data, if used, is determined by the application. The data supplied when the context is initiated must exactly match on the acceptor side. For example, the application may insert its "version number" in the application data, thereby prohibiting establishment of a security context with another application that is not a compatible version.

Channel bindings are encoded into the token generated by `gss_init_sec_context( )`, which is then sent to the acceptor. The channel bindings are verified by `gss_accept_sec_context( )`. The application initiating the security context must determine the channel binding values before calling `gss_init_sec_context( )`, and both applications must provide consistent values to the security context functions. The GSS-API checks the channel bindings on the accepting side of the application.

When channel bindings are passed to `gss_init_sec_context( )`, a hash value is computed using the channel bindings. The hash value is encoded into the token. During the call to `gss_accept_sec_context( )`, the GSS-API computes a hash value using the channel bindings passed to it and compares the computed hash value against the hash value in the token passed to it.

8.3.3.3.2 Confidentiality and Integrity

The initiator can specify that it wants message confidentiality or integrity, or both, to be available when sending and receiving messages. The acceptor can query to see if message confidentiality or integrity, or both, is required for this context. This information is then passed back to the initiator.

Note

The initiator can request confidentiality, but if the acceptor cannot supply it, confidentiality is not used.

Use the `GSS_C_CONF_FLAG` and the `GSS_C_INTEG_FLAG` of the `gss_init_sec_context( )` function to specify the confidentiality and integrity requirements.

8.3.3.3.3 Replay Detection

Replay detection means one side of the application checks to see if the token or message has been sent to it previously.

The initiator can specify that it wants messages to be checked against a message replay cache. This is used to determine if the message is a replay of a previous message.

The replay cache expands as required, allocating more resources for the cache as more messages are cached. Note that there is an upper limit on the cache size. Once a replay cache reaches a certain size, adding a new entry will be accompanied by freeing an older entry. The resources associated with the entire cache are freed when the security context is deleted.

Note

There are two different replay caches supported in Kerberos 5. The message replay cache is a memory-based cache that is controlled by parameters set during context establishment. The authentication replay cache is a file-based cache with settings controlled by UNIX environment variables.

8.3.3.3.4 Out-of-Sequence Message Detection

The initiator can specify that it wants messages to be sequenced (or numbered) to verify that all messages are received and that they are received in the order sent. Message sequencing also provides replay detection.

8.3.3.3.5 Mutual Authentication

The initiator can specify that it wants the acceptor to also perform authentication to the initiator. This way, both sides of the application can be assured that they are talking to the genuine other side of the application.

8.3.3.3.6 Encryption Type: DES vs. DES3

The initiator can specify the DES3 encryption standard during initial authentication. The initiator can also request DES3 encryption when initiating a security context. DES3 encryption offers significantly enhanced security over DES encryption.

8.3.3.3.7 Credentials Delegation

For Kerberos 5 implementations, the initiator can set the `GSS_C_DELEG_FLAG` delegation flag to specify that it wants initial tickets (TGTs) to be forwarded from one network host to another. Note that a TGT can be forwarded only if it has the Kerberos forwardable TGT attribute.

8.3.3.4 Identifying the Targeted Security Measures

The initiator calls `gss_init_sec_context( )` and identifies the security measures that it wants the security mechanism to use. Flags are returned to the application, indicating which security measures are provided by the security mechanism. Not all security measures defined by the GSS-API standard may be provided by a security mechanism.

When the initiator specifies the security measures to use when establishing a security context, the acceptor must accept them. There is no room for negotiation at this time. For example, if the initiator specifies sequencing or replay detection when it calls `gss_init_sec_context( )`, the acceptor must provide it.

If channel bindings are specified when the initiator calls `gss_init_sec_context( )`, the acceptor must supply the exact same channel bindings in its call to `gss_accept_sec_context( )`.

8.3.4 Message Functions

The GSS-API message functions perform protection on a per-message basis. They can be invoked by either side of the application. When the application calls these functions, it specifies the *quality of protection* (QOP), which identifies the cryptographic algorithm that will be used to process the message. The values of the QOP are mechanism-dependent. The key used to apply the cryptographic algorithm is taken from the security context.

GSS-API Function	Description
`gss_get_mic( )`	Generates a signature from a message, ensuring data integrity and data origin authentication; a token (the signature) is returned. The original message is not encapsulated into the token.
`gss_unwrap( )`	Deciphers and validates the token to determine its integrity, decrypts the message if needed, and returns the message.
`gss_verify_mic( )`	Validates the message against its signature to ensure it was not tampered with during transmission.
`gss_wrap( )`	Encapsulates a message into a token, encrypting it if confidentiality is specified, and includes a signature to ensure that the message was not tampered with during transit and to provide data origin authentication.
`gss_wrap_size_limit( )`	Determine the message size limit for the `gss_wrap( )` on a context, given a token size maximum that can be sent by your network.

These four functions are related to each other as pairs:

- `gss_get_mic( )` and `gss_verify_mic( )`
- `gss_wrap( )` and `gss_unwrap( )`

An application may use `gss_get_mic( )` and `gss_verify_mic( )` if it wants to send the signature separately from the data. For example, in PGP mail (Pretty Good Privacy), the message is sent first, followed by the signature. The signature is verified and the results are displayed, indicating the message was not modified in transit.

Note

These functions are new in GSS-API V2. The previous release used the functions `gss_sign( )`, `gss_verify( )`, `gss_seal( )`, and `gss_unseal( )`.

8.3.4.1 Quality of Protection

When the application calls these functions, it specifies the QOP, which identifies the integrity or confidentiality algorithm that will be applied to the message. Six algorithms are defined for the quality of protection in Kerberos 5:

- DES-MAC-MD5
- DES3-MAC-MD5
- MD2.5
- DES-MAC
- DES-CBC
- DES3-CBC

Note

Given the same quality of protection, `gss_get_mic( )` and `gss_wrap( )` calculate the signature the same way.

8.3.5 Miscellaneous Support Functions

This section lists miscellaneous support functions that may be used when securing applications.

GSS-API Function	Description
`gss_add_oid_set_member( )`	Adds an object identifier (OID) to a set.
`gss_create_empty_oid_set( )`	Creates a set containing no object identifiers.
`gss_display_status( )`	Converts a GSS-API status code to a printable text form. An application may need to call this function multiple times to receive the multiple status messages some errors indicate. The application releases the buffers by calling `gss_release_buffer( )`. This function operates on both major and minor status codes.

GSS-API Function	Description
gss_indicate_mechs( )	Returns the mechanism identifiers supported on the local system. The structure must be released by the application using gss_release_oid_set( ).
gss_release_buffer( )	Frees storage of printable names, buffers, and tokens. Used after calls to gss_display_status( ), gss_display_name( ), gss_export_name( ), gss_export_sec_context( ), gss_init_sec_context( ), gss_accept_sec_context( ), csf_gss_acq_user( ), gss_wrap( ), gss_get_mic( ), and gss_unwrap( ).
gss_release_oid_set( )	Frees storage of OID set objects.
gss_test_oid_set_member( )	Determine if an object identifier is member of a set.

In addition, there are some HP extension functions.

Application Security SDK Function	Description
cs_oid_cmp( )	Compares two OIDs.
cs_oid_dup( )	Duplicates an OID.
cs_oid_free( )	Frees resources associated with an OID.
cs_oid_in_set( )	Determines if an OID is included in an OID set.
cs_oid_set_cmp( )	Compares two OID sets.
cs_oid_set_dup( )	Duplicates an OID set.
cs_oid_set_free( )	Frees resources associated with an OID set.
cs_oid_set_insert( )	Inserts an OID into an OID set. The function gss_add_oid_set_member( ) is preferred for GSS-API v2 compliance.
cs_oid_set_isect( )	Creates a new set that is an intersection of two OID sets.
cs_oid_set_union( )	Creates a new set that is an union of two OID sets.
csf_gss_get_OidAddress( )	Retrieves the address of the built-in OID set.
csf_gss_get_RfcOidSet( )	Retrieves the address of the built-in Kerberos OID set.
csfgss_pPtr( )	Retrieves the a pointer to an OID mechanism or OID mechanism set.

8.3.5.1 OID and OID sets

In order to understand OIDs (Object Identifiers) and OID sets, it is useful to review some background information.

8.3.5.1.1 OSI

OSI, described in CCITT X.200, is an internationally standardized architecture that governs the interconnection of computers from the physical layer up to the user application layer. Objects at higher layers are defined abstractly and intended to be implemented with objects at lower layers.

8.3.5.1.2 ASN.1

OSI's method of specifying abstract objects is called ASN.1 (Abstract Syntax Notation One, defined in CCITT X.208), which defines a set of rules for representing such objects as strings of ones and zeros. ASN.1 is a flexible notation that allows the definition of a variety of data types, including simple types such as integers and bit strings, structured types such as sets and sequences, and complex types defined in terms of other types.

8.3.5.1.3 Object Identifiers

An OID (Object Identifier) is one such data type, being a simple data type used to identify objects such as algorithms and attribute types. OIDs are composed of a sequence of integer components, values that are given meanings by registration authorities.

In the GSS-API, OIDs are used to identify parameters such as name types and mechanisms. The following examples shows how they work.

Example 8–1: Constant Pointing to a Structure Containing a String

The constant `GSS_C_NT_USER_NAME` is a `gss_OID` type initialized to point to a `gss_OID_desc` structure containing the following OID string:

```
{10, (void *)"\x2a\x86\x48\x86\xf7\x12\x01\x02\x01\x01"}
```

This can be converted to the decimal object identifier values:

```
{ 1 2 840 113554 1 2 1 1 }
```

which in turn represent the following hierarchy of objects:

```
{iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2)
generic(1) user_name(1)}
```

Example 8–2: Constant Pointing to a String

The constant `rfc_krb5_c_OID` is a `gss_OID` type which points to the OID string for:

```
{iso(1) member-body(2) United States(840) mit(113554) infosys(1) gssapi(2)
krb5v2(3)}
```

8.3.5.1.4 OID Sets

An OID set is a GSS-API data structure that contains a set of OIDs, and a integer representing the number of OIDs in the set. Some GSS-API functions use sets to input and output a large number and/or a large variety of mechanism and attribute objects. For example, when you call `gss_acquire_cred( )`, an OID set is returned that contains object identifiers for all supported mechanisms. You can use `gss_test_oid_set_member( )` to see if a certain OID is in the set.

OID sets can also be created using the functions `gss_create_empty_oid_set( )` and `gss_add_oid_set_member( )`.

8.3.6 V1 Compliance Functions

There are several functions included in Application Security SDK that are not supported in the GSS-API Version 2. HP continues to support these functions in this release for v1 interoperability. New applications being written using GSS-API Version 2 should not use the following v1 functions; instead use the v2 replacement.

GSS-API v1 Function	GSS-API v2 Replacement
`gss_open( )`	Action no longer required
`gss_close( )`	Action no longer required
`gss_process_context_token( )`	Action no longer required
`gss_seal( )`	`gss_wrap( )`
`gss_sign( )`	`gss_get_mic( )`
`gss_unseal( )`	`gss_unwrap( )`
`gss_verify( )`	`gss_verify_mic( )`

8.4 Best Practices

The following sections provide information regarding security considerations and recommendations that should be followed when you are using Application Security SDK to create a custom application.

8.4.1 Multi-threading

The following functions are not thread-safe:

- `gss_acquire_cred( )`
- `gss_add_cred( )`
- `csf_gss_acq_user( )`
- `csf_gss_inq_user( )`
- `csf_gss_release_user( )`
- `csf_gss_renew_cred( )`

All of these functions pertain to the acquisition of initial tickets, and should therefore not be used in the multi-thread portion of your application.

Special care has been taken to ensure that `gss_init_sec_context( )` and `gss_accept_sec_context( )` are both multi-thread safe. All other function calls are thread-safe, provided that each thread uses a unique security context.

8.4.2 Cache Management

Cache sharing between principals is a security risk. One principal that shares the cache may impersonate another principal that also uses the cache. This can result in exploitation of the principals or the servers that trust the principals, or the passing of replayed messages.

HP recommends that you use a unique credentials cache for each initiator, and a unique key table entry for each acceptor. Ideally, each acceptor application should have a separate key table file.

8.4.3 Encryption Types

Application Security SDK supports both DES and DES3 encryption types. HP recommends using DES3 encryption as it provides significantly enhanced protection over DES.

Note that using DES3 restricts portability. Applications that use DES3 encryption can only be used with Application Security SDK implementations.

8.4.4 Exported Security Contexts

The `gss_export_sec_context( )` function creates a token that can be used to pass a security context between processes. This token contains the context's secret key, and therefore is extremely sensitive data. These exported security contexts cannot be secured by the GSS-API, and must be protected by the application.

Applications that use this GSS-API feature must be sure to pass an exported security context between processes in a secure manner. Exported security contexts should be kept within the boundaries of the host that created the security context.

8.4.5 Key Management with GSS and Kerberos 5

Secret keys are used with encryption algorithms and keyed checksum algorithms, and should be changed periodically. How often depends on the algorithms used and the amount of data that is protected by them. One way to decrease the frequency is by using a stronger encryption algorithm, such as DES3 instead of DES.

There are five different types of keys that are used:

- A user principal has a secret key (password) stored in the principal database. This key is used to acquire ticket-granting-tickets (TGTs). If a user principal acquires many TGTs with the same password, the password may need to be changed in advance of its expiration time.
- Each TGT contains a session key. This key is used to acquire service tickets and to renew tickets. If a user principal, or a service principal acting as a client to other services, acquires many service tickets with the same TGT, you may need to acquire a new TGT in advance of its expiration time.

  You can also shorten the lifetime of the TGT. When specifying the lifetime with `csf_gss_acq_user( )`, the function treats it as the minimum lifetime. To ensure that you don't use a TGT with a greater lifetime, use the `CSF_GSS_C_ACQ_USER_OPT_ALWAYS_FETCH` option which will force `csf_gss_acq_user( )` to obtain a new TGT with the desired lifetime from the KDC.
- Each service ticket contains a session key. This key is used when establishing a security context. If a user principal, or a service principal acting as a client to another service, establishes many security contexts using the same service ticket, you may need to replace the service ticket before it expires. Service tickets generally expire at the same time as the TGT used to obtain them. The only way to replace a service ticket before it expires is to acquire a new TGT, which will re-initialize the credentials cache in which the tickets are stored.
- A service principal has a secret key stored in the principal database and also in a service key table file. When accepting a security context, this key is used to encrypt data. Each service ticket that a user principal acquires for the service principal contains the encrypted data.

  Also, a service principal that acts as a client to other services uses its secret key to acquire TGTs. A service principal's secret key is typically used often and therefore needs to be changed often. This is one reason why HP recommends against sharing of a service principal amongst server applications.

- Each security context contains a session key. This key is used when protecting messages. Security contexts do not expire. If you use one security context to exchange a lot of data or if you have a long-lived conversation, you may need to delete the security context and establish a new one periodically.

Since all but the security context session keys are stored in either a credentials cache or a service key table file, credentials caches and service key table files need to be protected.

8.4.6 Multi-threaded Functions

The following Application Security SDK functions are not thread-safe:

- `gss_acquire_cred( )`
- `gss_add_cred( )`
- `csf_gss_acq_user( )`
- `csf_gss_inq_user( )`
- `csf_gss_release_user( )`
- `csf_gss_renew_cred( )`

All of these functions pertain to the acquisition of initial tickets, and should therefore not be used in a multi-thread environment. If credentials are fetched before a program creates threads, the credential may be passed into the threads for a multi-threaded acceptor or initiator.

Special care has been taken to ensure that `gss_init_sec_context( )` and `gss_accept_sec_context( )` are both multi-thread safe. All other Application Security SDK calls are thread-safe, provided that each thread uses a unique security context.

8.4.7 Mutual Authentication

Mutual authentication means that both the initiator and acceptor principals are required to verify their identities before the security context can be established. The `GSS_C_MUTUAL_FLAG` option in the `req_flags` parameter of the function `gss_init_sec_context( )` indicates whether mutual authentication is required.

If principals are not required to perform mutual authentication, there is a risk of getting a replayed message. If mutual authentication is not performed, message replay detection should be enabled using the GSS_C_REPLAY_FLAG of the req_flags parameter.

8.4.8 Protecting Passwords

HP recommends that all dialog boxes or forms used to obtain password information from principals should have the keyboard echo turned-off. This prevents other persons in the room from reading the user's password on the computer display.

Buffers used to store passwords should be set to zero immediately after use.

8.4.9 Replay Protection

The gss_init_sec_context( ) function contains a flag (GSS_C_REPLAY_FLAG of the req_flags parameter) that indicates if message replay detection should be enabled. Enabling message replay detection is important if applications are not performing mutual authentication during context establishment, as indicated by the GSS_C_MUTUAL_FLAG in the req_flags parameter being set to FALSE.

Message replay detection requires that a replay cache be used to store message received from the other application. There is a cache located on both the initiator and acceptor hosts. These caches are memory-based, and can expand in size to an upper limit. Consequently, replay detection will have an impact on the size of the process. New replay cache entries are allocated during gss_unwrap( ) and gss_verify_mic( ), giving the appearance of a memory leak in the application. This is normal behavior. When a security context that uses message replay detection is deleted, the entire message replay cache is also deleted.

There is a different type of replay cache that is used for authentication replay detection. This is by default a file-based cache, as controlled by the UNIX CSFC5RCNAME environment variables or the Windows RepCache registry entries. This cache should never be configured to be a memory-based cache as that would introduce a security vulnerability. This cache type is always active, and is not controlled by the initiating application.

8.4.10 Refreshing Credentials

There are two security parameters containing cryptographic keys that do not expire. One is the service key table entry. The other is the security context. HP advises that these parameters should be refreshed periodically to minimize the potential for cryptographic exploitation of these keys.

8.4.11 Resource Management

Many functions create strings, buffers or other resources that are used to store security information for use by the application. It is very important that these resources are properly freed or released after use by calling the appropriate function included in the SDK. Failure to do so may compromise your security. A mismanaged resource may be able to be retrieved and read in a security attack.

In the Reference section of the SDK Manuals, information is provided on the appropriate resource management call that is used to free or release the contents of each function that creates a sensitive resource. Be sure to read the Reference sections of the individual SDK manuals thoroughly to determine the correct procedures to avoid this security exposure.

8.4.12 Service Key Table Files

The service key table file contains the credentials used by the acceptor application to verify its identity. Access to this file should be very restricted, allowing no user permission to read the contents of the file. Only trusted services should have the permissions to access the file.

The credentials in the service key table file do not expire. HP recommends that these credentials be refreshed periodically. To refresh an entry in the service key table file, a new random key should be generated by the security server, and extracted to the service host using a principal database administration program.

8.4.13 Ticket Attributes

Kerberos 5 initial tickets can be requested with a variety of attributes, as specified by flags in the `csf_gss_acq_user( )` function. The following sections describe the attributes that can be requested.

8.4.13.1 Forwardable Tickets

Forwarding is a mechanism that sends a TGT from one host to another. The forwarded TGT can be used to generate a new service ticket on the second host on behalf of the initial principal named in the TGT. Tickets should only be forwarded to trusted principals.

8.4.13.2 Preauthentication

Preauthentication means that additional encrypted data is sent with a TGT request. The additional encrypted data is in the form of the Encrypted Timestamp. This added information provides extra security during the authentication process. If the `CSF_GSS_C_ACQ_USER_OPT_NOPREAUTH` is indicated, preauthentication is not required.

HP does not recommend skipping preauthentication, unless there is some need to do so unrelated to security, and therefore recommends that you avoid using this option unless absolutely necessary. Specifically for the Kerberos 5 security mechanism, preauthentication during acquisition of initial tickets prevents an impostor from getting tickets written in your name. In normal circumstances these tickets are only usable by the principal that is named in the ticket, as only they possess the secret key to decrypt it. However in the event there is some cryptographic exploitation capability held by an adversary, preauthentication prevents them from getting information from the security server that they can exploit. It also guarantees freshness of the ticket request.

8.4.13.3 Ticket Lifetime

The requested ticket lifetime is designated by the `CSF_GSS_ACQ_USER_OPT_LIFETIME` flag. The security server may return an initial ticket with a shorter lifetime than requested. This is because the maximum lifetime for a ticket issued to a principal in a given realm is controlled by the settings for the special principal `krbtgt/`*`REALM@REALM`*.

There are also ticket lifetime limits imposed by the Kerberos version. The maximum ticket lifetime for a Kerberos 4 ticket is 21 hours and 15 minutes. If you permit a maximum lifetime of 1 day, a Kerberos 5 ticket will last a full 24 hours, but a Kerberos 4 ticket expires after 21 hours and 15 minutes.

8.4.13.4 Ticket Renew Time

A ticket can be requested with a renewable attribute using the `CSF_GSS_C_ACQ_USER_OPT_RENEWABLE` flag. Note that the security server may return an initial ticket with a shorter renew time than requested. This is because the maximum renew time for a ticket issued to a principal in a given realm is controlled by the settings for the special principal `krbtgt/`*`REALM@REALM`*.

Once a renewable ticket is received, the renew time is controlled by a set of rules. When a principal initially requests a TGT, lifetime and renew time are not related. The granted renewable time has no relationship to the lifetime when the original ticket is issued. However, the two times do interact when a ticket is renewed.

8.4.13.4.1 General Rules for Lifetime and Renew Settings

When you renew a ticket, the resulting lifetime is determined according to the following rules:

- A ticket can only be renewed if it has a renewable ticket attribute.
- Renewable tickets may be renewed at any point during the ticket's lifetime.
- An expired ticket cannot be renewed.
- A renewable ticket can be renewed multiple times, until the renew time limit has expired.
- A renewed ticket will have the same lifetime as the original lifetime of the renewable ticket. As an example, assume a ticket has a lifetime of 30 minutes but is renewable for 50 minutes. After 15 minutes, the ticket is renewed. The renewed ticket lifetime is also 30 minutes.
- The rule is modified when a ticket is renewed if the remaining renew time is shorter than the original ticket lifetime. The renewed ticket expiration time cannot be later than the renew expiration time when the original ticket was issued. Referring to the example above, suppose the ticket is renewed before the ticket expires, say at 25 minutes after it is originally obtained as a renewable ticket. The renewed ticket will be valid for 25 minutes---not 30 minutes (25 + 25 = 50 minutes). The total time that the renewable and renewed ticket are valid must be less than or equal to the renew time limit set on the original ticket.
- Any proxy (service) tickets obtained with a renewable TGT inherit the renewable option and the renew time, and are renewed along with the TGT. This follows the general idea that service tickets inherit the attributes (when possible) of the TGT used to obtain them.
- Renewing tickets does not require a password.

8.5 Building a Portable Application

The GSS-API standard is designed to be generic, and to allow applications that incorporate the GSS-API to be source-level compatible. However, if you use GSS-API implementations from different vendors, you may encounter some portability problems between these vendors.

The following sections provide recommendations to help minimize this impact and make your GSS-API applications created using Application Security SDK more portable across platforms.

8.5.1 Using Printable Names and Comparing Names

The `gss_display_name( )` function converts an internal name (i.e., a name returned by `gss_import_name( )`) to a text form suitable for printing. Since the GSS-API standard does not require that the printable name and name-type OID returned from `gss_display_name( )` be suitable for input to `gss_import_name( )`, the printable name format is implementation-specific and operating system-dependent. The format of printable names may differ across GSS-API implementations, whether the implementations are cross-vendor, cross-platform, or incremental vendor releases. Consequently, printable names are not reliable as input to name comparison functions outside the GSS-API standard. You should use the export name format created with the `gss_export_name( )` function for this purpose. Examples where the export name may be used include ACL (access control list) functions, accounting functions, or diagnostic aids.

8.5.2 Specifying Mechanisms

Passing an OID set to `gss_acquire_cred( )` or `csf_gss_acq_user( )` and passing an OID to `gss_init_sec_context( )` and `gss_import_name( )` reduce the portability of an application as technology changes. For example, your enterprise may start out using the Kerberos security mechanism, but may later change to a public-key security mechanism. If application developers hard-coded OID sets specifying Kerberos, those applications would have to be modified. Also, mechanism OIDs can change.

To increase portability, an application should pass default OID and OID set values to functions, thus permitting the GSS-API implementation to choose whatever mechanisms are available and appropriate.

8.5.3 Specifying a Quality of Protection (QOP)

The `gss_wrap( )` and `gss_get_mic( )` functions accept a QOP parameter that specifies the algorithm to be applied to the message. The QOP parameter value is mechanism-specific. Specifying anything other than the default value (in which case the GSS-API implementation chooses a value) hard-codes the application to a mechanism.

Consequently, should you change the mechanism used, or the mechanism you currently use changes functionally, application source code modifications may be required.

The `gss_unwrap( )` and `gss_verify_mic( )` functions and their inverse functions of `gss_wrap( )` and `gss_get_mic( )` are supposed to return the actual QOP used to protect the message. If an application specifies the default QOP when it encodes a message, the QOP returned by the inverse function is mechanism-dependent.

To increase portability, an application should specify the default QOP value when calling `gss_wrap( )` and `gss_get_mic( )` and should not rely on the value returned by their inverse functions (`gss_unwrap( )` and `gss_verify_mic( )`, respectively).

The default integrity QOPs are as follows:

- `CSF_GSS_KRB5_INTEG_C_QOP_DES3_MD5` for DES3.
- `GSS_KRB5_INTEG_C_QOP_DES_MD5` for DES

The confidentiality QOPs are as follows:

- `CSF_GSS_KRB5_CONF_C_QOP_DES3` for DES3
- `GSS_KRB5_CONF_C_QOP_DES` for DES

The default QOP used depends on the encryption type selected when the security context was established.

Note

The DES3 QOP is not portable; designating this QOP renders the application to be Kerberos 5 mechanism-specific.

8.5.4 Default Names

Some GSS-API functions accept a parameter that selects a default name. However, the GSS-API standard does not define how default names are formed; it is mechanism-dependent and operating system-dependent. For example, one GSS-API implementation may form a default name by interrogating an environment variable, whereas another may form it from the user's ID (for example, a UNIX UID). There are also GSS-API implementations that form a default name based on indirect selection criteria and mechanism common sense. For example, Application Security SDK Kerberos mechanism implementation forms the default name *`user@REALM`* for credentials of usage `GSS_C_INITIATE` and the default `host/`*`hostname@REALM`* for credentials of usage `GSS_C_ACCEPT`.

HP recommends that to increase portability, you do not use mechanism-dependent default names.

8.6 Quick Reference

Application Security SDK consists of standard and proprietary functions described in that are used to secure applications:

- Function calls beginning with the letters `gss` correspond to the GSS-API standard.
- Function calls beginning with the letters `cs` or `csf` are HP-specific extensions.

Function	Description
`cs_oid_cmp( )`	Compare two OIDs.
`cs_oid_dup( )`	Duplicate an OID.
`cs_oid_free( )`	Free an OID.
`cs_oid_in_set( )`	Determine if an OID is in an OID set.
`cs_oid_set_cmp( )`	Compare two OID sets.
`cs_oid_set_dup( )`	Duplicate an OID set.
`cs_oid_set_free( )`	Free an OID set.
`cs_oid_set_insert( )`	Insert an OID into an OID set.
`cs_oid_set_isect( )`	Create a new OID set that is an intersection of two existing OID sets.
`cs_oid_set_union( )`	Create a new OID set that is a union of two existing OID sets.
`csf_gss_acq_user( )`	Acquire a user prior to initiating a security context.
`csf_gss_get_context_op-tions( )`	Verify the type of encryption being used.
`csf_gss_get_OidAddress( )`	Returns address of built-in OID.
`csf_gss_get_RfcOidSet( )`	Returns address of built-in Kerberos OID set.
`csf_gss_inq_user( )`	Obtain information about a user.
`csfgss_pPtr( )`	Get a pointer to an OID mechanism or OID mechanism set.
`csf_gss_release_user( )`	Delete a user when no longer needed.
`csf_gss_renew_cred( )`	Renew credentials.
`gss_accept_sec_context( )`	Accept a security context initiated by a peer application.
`gss_acquire_cred( )`	Retrieve stored credentials for use by an application.
`gss_add_cred( )`	Construct credentials incrementally.
`gss_add_oid_set_member( )`	Add an object identifier (OID) to a set.
`gss_canonicalize_name( )`	Convert an internal name to a mechanism-specific name (MN).
`gss_close( )`	Support for this function is for v1 interoperability.
`gss_compare_name( )`	Compare two internal form names.
`gss_context_time( )`	Determine how long a security context remains valid.

Function	Description
gss_cre-ate_empty_oid_set( )	Create a set containing no object identifiers.
gss_delete_sec_context( )	Delete a security context.
gss_display_name( )	Convert an internal form name to text.
gss_display_status( )	Convert a GSS-API status code to text.
gss_duplicate_name( )	Create a copy of an internal form name.
gss_export_name( )	Convert a mechanism specific name to an export form.
gss_export_sec_context( )	Transfer a security context to another process.
gss_get_mic( )	Calculate a signature, called a message integrity code (MIC), for a message.
gss_import_name( )	Convert a text name to internal form.
gss_import_sec_context( )	Import a previously exported security context.
gss_indicate_mechs( )	Determine the available security mechanisms.
gss_init_sec_context( )	Initiate a security context with a peer application.
gss_inquire_context( )	Obtain information about a security context.
gss_inquire_cred( )	Obtain information about credentials.
gss_in-quire_cred_by_mech( )	Obtain per-mechanism information about credentials
gss_in-quire_mechs_for_name( )	List the mechanisms that support the name type.
gss_in-quire_names_for_mech( )	List the name types supported by the specified mechanism.
gss_oid_to_str( )	Display OID as string.
gss_open( )	Support for this function is for v1 interoperability.
gss_process_context_to-ken( )	Process the token on a security context from a peer application. Support for this function is for v1 interoperability.
gss_release_buffer ( )	Delete a buffer.
gss_release_cred( )	Delete credentials after use.
gss_release_name( )	Delete an internal form name.
gss_release_oid( )	Free storage of OID object.
gss_release_oid_set( )	Delete a set of object identifiers.
gss_seal( )	Replaced by `gss_wrap( )`.
gss_sign( )	Replaced by `gss_get_mic( )`.

Function	Description
`gss_str_to_oid( )`	Construct OID from string.
`gss_test_oid_set_member( )`	Determine whether an object identifier is a member of a set.
`gss_unseal( )`	Replaced by `gss_unwrap( )`.
`gss_unwrap( )`	Verify the signature attached to a message and decrypt the message content if necessary.
`gss_verify( )`	Replaced by `gss_verify_mic( )`.
`gss_verify_mic( )`	Verify the signature for a received message.
`gss_wrap( )`	Attach a signature to a message and, optionally, encrypt the message content.
`gss_wrap_size_limit( )`	Determine the message size limit for `gss_wrap( )` on a context, given a maximum token size dictated by the network.

8.6.1 Reference Page Conventions

Each function is described in a reference page using the following categories:

Purpose

A brief description of the function purpose.

Syntax

A complete listing of permitted parameters and command line arguments.

Input parameters are those parameters passed from the application to Application Security SDK.

Output parameters are those parameters, passed from Application Security SDK to the application.

If an input parameter is optional, the application can supply a default value for that parameter.

If an output parameter is optional, Application Security SDK does not need to return a value to the application because the application does not need it. Use `NULL` in the output parameter to indicate this.

Note

To avoid memory leaks, the application must always release the storage associated with returned values after use.

Parameters

A list of variables passed to and returned from the function. For binary values:

- Logical 1 represents true.
- Logical 0 represents false.

Parameters initialized to `NULL` may also be zero.

Description

The definition, purpose, and usage tips for a function.

Portability Considerations

Concerns for porting an HP secured application to another environment.

Return Values

A list of major status values returned by the function.

See Also

A list of related function calls.

8.7 Constants

The header files included with Application Security SDK include many constants. The constant definitions from the header files are included here for reference.

Context Flags	Definition
`GSS_C_DELEG_FLAG`	1
`GSS_C_MUTUAL_FLAG`	2
`GSS_C_REPLAY_FLAG`	4
`GSS_C_SEQUENCE_FLAG`	8
`GSS_C_CONF_FLAG`	16
`GSS_C_INTEG_FLAG`	32
`GSS_C_ANON_FLAG`	64
`GSS_C_PROT_READY_FLAG`	128
`GSS_C_TRANS_FLAG`	256
`CSF_GSS_C_DES_FLAG`	268435456
`CSF_GSS_C_DES3_FLAG`	536870912

Credentials Usage	Definitions
GSS_C_BOTH	0
GSS_C_INITIATE	1
GSS_C_ACCEPT	2

Status Code Types	Definitions
GSS_C_GSS_CODE	1
GSS_C_MECH_CODE	2

Address Types	Definitions
GSS_C_AF_UNSPEC	0
GSS_C_AF_LOCAL	1
GSS_C_AF_INET	2
GSS_C_AF_IMPLINK	3
GSS_C_AF_PUP	4
GSS_C_AF_CHAOS	5
GSS_C_AF_NS	6
GSS_C_AF_NBS	7
GSS_C_AF_ECMA	8
GSS_C_AF_DATAKIT	9
GSS_C_AF_CCITT	10
GSS_C_AF_SNA	11
GSS_C_AF_DECnet	12
GSS_C_AF_DLI	13
GSS_C_AF_LAT	14
GSS_C_AF_HYLINK	15
GSS_C_AF_APPLETALK	16
GSS_C_AF_BSC	17
GSS_C_AF_DSS	18
GSS_C_AF_OSI	19
GSS_C_AF_X25	21
GSS_C_AF_NULLADDR	255

Various NULL Values	Definitions
GSS_C_NO_NAME	NULL
GSS_C_NO_BUFFER	NULL
GSS_C_NO_OID	NULL
GSS_C_NO_OID_SET	NULL
GSS_C_NO_CONTEXT	NULL
GSS_C_NO_CREDENTIAL	NULL
GSS_C_NO_CHANNEL_BINDINGS	NULL
CSF_GSS_C_NO_USER	NULL
CSF_GSS_C_ACQ_USER_OPT_NONE	NULL

QOPs	Definitions
GSS_C_QOP_DEFAULT	0
GSS_KRB5_CONF_C_QOP_DES	0
GSS_KRB5_INTEG_C_QOP_MD5	1
GSS_KRB5_INTEG_C_QOP_DES_MD5	2
GSS_KRB5_INTEG_C_QOP_DES_MAC	3
CSF_GSS_KRB5_INTEG_C_QOP_DES3_MD5	5341
CSF_GSS_KRB5_CONF_C_QOP_DES3	5342

User Options	Definitions
CSF_GSS_C_ACQ_USER_OPT_LIFETIME	1
CSF_GSS_C_ACQ_USER_OPT_RENEWABLE	2
CSF_GSS_C_ACQ_USER_OPT_CCNAME	4
CSF_GSS_C_ACQ_USER_OPT_KTNAME	8
CSF_GSS_C_ACQ_USER_OPT_SVCKEY	16
CSF_GSS_C_ACQ_USER_OPT_ALWAYS_FETCH	256
CSF_GSS_C_ACQ_USER_OPT_FORWARDABLE	32
CSF_GSS_C_ACQ_USER_OPT_PROXIABLE	64
CSF_GSS_C_ACQ_USER_OPT_NOPREAUTH	128

Encryption Types	Definitions
CSF_GSS_C_ENCTYPE_DES_CBC_CRC	1
CSF_GSS_C_ENCTYPE_DES_CBC_MD5	3
CSF_GSS_C_ENCTYPE_DES3_CBC_MD5	5

Preauthentication Types	Definitions
CSF_GSS_C_PREAUTH_NONE	0
CSF_GSS_C_PREAUTH_ENC_TIMESTAMP	2
CSF_GSS_C_PREAUTH_ENC_UNIX_TIME	5

Challenge States	Definitions
CSF_GSS_C_USER_STATE_NULL	0
CSF_GSS_C_USER_STATE_PASSWORD_NOECHO	1
CSF_GSS_C_USER_STATE_CHALLENGE_ECHO	2
CSF_GSS_C_USER_STATE_OTP_ECHO	3
CSF_GSS_C_USER_STATE_PASSWORD_ECHO	4
CSF_GSS_C_USER_STATE_CHAL-LENGE_NOECHO	5
CSF_GSS_C_USER_STATE_OTP_NOECHO	6

Miscellaneous	Definitions
GSS_C_INDEFINITE	0xFFFFFFFF
CSF_GSS_C_PURGE_FLAG	1

8.8 Data Structures

Application Security SDK uses the following data structures to pass data to and from function calls:

- `gss_channel_bindings_t` — identifies the communications channel for a security context
- `gss_buffer_t` — specifies input or output data for a function call
- `csf_gss_opts_t` — specifies options for a user context

8.8.1 gss_channel_bindings_t

`gss_channel_bindings_t` is a pointer to the data structure containing information that identifies the communications channel for a security context. Channel bindings explains how channel bindings work.

The structure is defined in the `gssapi.h` header file as follows:

```
typedef struct _gss_channel_bindings_struct {
OM_uint32     initiator_addrtype;
gss_buffer_desc  initiator_address;
OM_uint32     acceptor_addrtype;
gss_buffer_desc  acceptor_address;
gss_buffer_desc  application_data;
} gss_channel_bindings_desc, *gss_channel_bindings_t;
```

The `initiator_addrtype` and `acceptor_addrtype` fields denote the type of addresses contained in the `initiator_address` and `acceptor_address` buffers. Application Security SDK supports the address format `GSS_C_AF_INET`.

The following function calls use `channel_bindings_t` structures:

```
gss_accept_sec_context( )
gss_init_sec_context( )
```

8.8.2 gss_buffer_t

`gss_buffer_t` is a pointer to a data structure that passes data to and from function calls. The actual data structure, a buffer descriptor, consists of a length field that contains the total number of data bytes and a value field that contains a pointer to the actual data.

The structure is defined in the `gssapi.h` header file as follows:

```
typedef struct gss_buffer_desc_struct {
size_t     length;
void      *value
} gss_buffer_desc, *gss_buffer_t;
```

Storage for the data returned to an application by a function call using this data structure is allocated by the function. The application must free this storage after use by invoking `gss_release_buffer( )`. Unused buffers may be initialized to the value of `GSS_C_EMPTY_BUFFER`.

The following function calls use options structures:

```
csf_gss_acq_user( )
gss_accept_sec_context( )
gss_delete_sec_context( )
gss_display_name( )
gss_display_status( )
gss_export_name( )
gss_export_sec_context( )
gss_get_mic( )
gss_import_name( )
gss_import_sec_context( )
gss_init_sec_context( )
gss_process_context_token( )
gss_release_buffer( )
gss_unwrap( )
gss_verify_mic( )
gss_wrap( )
```

8.8.3 csf_gss_opts_t

`csf_gss_opts_t` is a pointer to the data structure that stores optional parameters for a user context.

The structure used to store a single option is defined in the `ext.h` header file as follows:

```
typedef struct csf_gss_mech_opt_desc_struct {
gss_OID     mechOID;   /* Mech to specify option for */
OM_uint32    id;     /* Identifier of option */
void *     val;     /* Value associated with option if appropriate */
} csf_gss_mech_opt_desc, *csf_gss_opts_t;
```

An array of these structures, one for each option desired, must be constructed. Initialize the `mechOID` field of the last record of the array to `GSS_C_NO_OID`.

The following function calls use options structures:

`csf_gss_acq_user( )`

8.9 Return Values

The following information is provided to aid in debugging and troubleshooting applications.

8.9.1 Status Codes Defined

Many Application Security SDK functions have two return values: major status and minor status.Both status codes are returned in a common structure.

If the major status is zero, then the function was executed successfully.If the major status was non-zero, then it indicates what error occurred in Application Security SDK or that there was a failure in the underlying Kerberos 5 mechanism. When a non-zero major status is returned, see the minor status for a more detailed error code regarding the failure.

To programmatically check for error conditions and to recover from the error, see the header files included as part of Application Security SDK.

Application Security SDK also provides macros that evaluate major status values.

8.9.2 Error Processing Macros

Use the following macros to process return values.

Name	Description
GSS_ERROR( )	Indicates either a routine or calling error has occurred.
GSS_CALLING_ERROR( )	Indicates a calling error has occurred.
GSS_ROUTINE_ERROR( )	Indicates a routine error has occurred.
GSS_SUPPLEMENTARY_INFO( )	Indicates whether any supplementary information bits are set.

8.9.2.1 GSS_ERROR( )

Syntax

```
GSS_ERROR(major_status)
```

Parameters

`major_status`	Major status being examined for an error. It uses the `OM_uint32` type.

Description

The `GSS_ERROR( )` macro evaluates to true if the major status indicates a routine error or a calling error.

See Also

`GSS_CALLING_ERROR( )`, `GSS_ROUTINE_ERROR( )`

8.9.2.2 GSS_CALLING_ERROR( )

Syntax

```
GSS_CALLING_ERROR(major_status)
```

Parameters

`major_status`	Major status being examined for an error. It uses the `OM_uint32` type.

Description

The `GSS_CALLING_ERROR( )` macro evaluates to logical one (1) if a calling bit representing a calling error is set; logical zero (0) if no bits are set.

Calling Error Bits

GSS_S_CALL_INACCESSIBLE_READ	01xxxxxx
GSS_S_CALL_INACCESSIBLE_WRITE	02xxxxxx
GSS_S_CALL_BAD_STRUCTURE	03xxxxxx

See Also

GSS_ERROR( ), GSS_ROUTINE_ERROR( )

8.9.2.3 GSS_ROUTINE_ERROR( )

Syntax

```
GSS_ROUTINE_ERROR(major_status)
```

Parameters

major_status	Major status being examined for an error. It uses the OM_uint32 type.

Description

The GSS_ROUTINE_ERROR( ) macro evaluates to logical one (1) if a calling bit representing a routine error is set; logical zero (0) if no bits are set.

Calling Error Bits

GSS_S_BAD_BINDINGS	xx04xxxx
GSS_S_BAD_STATUS	xx05xxxx
GSS_S_BAD_SIG	xx06xxxx
GSS_S_BAD_MECH	xx01xxxx
GSS_S_BAD_NAME	xx02xxxx
GSS_S_BAD_NAMETYPE	xx03xxxx
GSS_S_CONTEXT_EXPIRED	xx0Cxxxx
GSS_S_CREDENTIALS_EXPIRED	xx0Bxxxx
GSS_S_DEFECTIVE_CREDENTIAL	xx0Axxxx
GSS_S_DEFECTIVE_TOKEN	xx09xxxx
GSS_S_FAILURE	xx0Dxxxx
GSS_S_NO_CRED	xx07xxxx
GSS_S_NO_CONTEXT	xx08xxxx

See Also

GSS_CALLING_ERROR( ), GSS_ERROR( )

8.9.2.4 GSS_SUPPLEMENTARY_INFO( )

Syntax

```
GSS_SUPPLEMENTARY_ERROR(major_status)
```

Parameters

`major_status`	Major status being examined for supplementary information bits. It uses the `OM_uint32` type.

Description

The `GSS_SUPPLEMENTARY_INFO( )` macro evaluates to logical 1 if any supplementary information bit is set; logical 0 if no bits are set. To test whether a specific bit is set, the application must AND the major status with the macro of the information bit being tested.

Supplementary Information Bits

GSS_S_CONTINUE_NEEDED	xxxx0001
GSS_S_DUPLICATE_TOKEN	xxxx0002
GSS_S_OLD_TOKEN	xxxx0004
GSS_S_UNSEQ_TOKEN	xxxx0008

See Also

`GSS_ERROR( )`

8.9.3 Major Status

Major status codes returned by Application Security SDK function calls are listed in numerical order.

Hex value	Return Code	Description
00000000	GSS_S_COMPLETE	The function call completed successfully.
01xxxxxx	GSS_S_CALL_INACCESSI-BLE_READ	A required input parameter could not be read.
02xxxxxx	GSS_S_CALL_INACCESSI-BLE_WRITE	A required output parameter could not be written.
03xxxxxx	GSS_S_CALL_BAD_STRUCTURE	A parameter was malformed.

Hex value	Return Code	Description
xx01xxxx	GSS_S_BAD_MECH	The requested mechanism is unavailable.
xx02xxxx	GSS_S_BAD_NAME	An invalid name was supplied.
xx03xxxx	GSS_S_BAD_NAMETYPE	A supplied name was of an unsupported type.
xx04xxxx	GSS_S_BAD_BINDINGS	Incorrect channel bindings were supplied.
xx05xxxx	GSS_S_BAD_STATUS	An invalid status code was supplied.
xx06xxxx	GSS_S_BAD_SIG or GSS_S_BAD_MIC	A token had an invalid MIC
xx07xxxx	GSS_S_NO_CRED	No credentials were supplied, or were unavailable or inaccessible
xx08xxxx	GSS_S_NO_CONTEXT	No context has been established.
xx09xxxx	GSS_S_DEFECTIVE_TOKEN	A token was invalid.
xx0Axxxx	GSS_S_DEFECTIVE_CREDENTIAL	A credential was invalid.
xx0Bxxxx	GSS_S_CREDENTIALS_EXPIRED	The referenced credentials have expired.
xx0Cxxxx	GSS_S_CONTEXT_EXPIRED	The security context has expired.
xx0Dxxxx	GSS_S_FAILURE	An error has occurred at the GSS-API level for unspecified reasons. The minor_status parameter contains more information.
xx0Exxxx	GSS_S_BAD_QOP	The requested quality of protection could not be provided by the context.
xx0Fxxxx	GSS_S_UNAUTHORIZED	The operation is forbidden by local security policy.

Hex value	Return Code	Description
xx10xxxx	GSS_S_UNAVAILABLE	The operation or option is unavailable. GSS_S_CONTINUE_NEEDED is a supplementary information bit flag. Test for GSS_S_CONTINUE_NEEDED by first testing the major status for an error using GSS_ERROR( ) and, if no error is indicated, bitwise ANDing the major status with GSS_S_CONTINUE_NEEDED.
xx11xxxx	GSS_S_DUPLICATE_ELEMENT	The requested credential element already exists.
xx12xxxx	GSS_S_NAME_NOT_MN	The provided name provided was not a mechanism name.
xxxx0001	GSS_S_CONTINUE_NEEDED	A token from the peer application is required to complete the security context establishment. Either gss_init_sec_context( ) or gss_accept_sec_context( ) must be called again to produce the token.
xxxx0002	GSS_S_DUPLICATE_TOKEN	The token was a duplicate of an earlier token.
xxxx0004	GSS_S_OLD_TOKEN	The validity period for the token has expired.
xxxx0008	GSS_S_UNSEQ_TOKEN	A later token has already been processed.
xxxx0010	GSS_S_GAP_TOKEN	An expected per-message token was not received.

8.9.4 Minor Status

Hybrid GSS\Kerberos error codes are listed below.

Hex value	Return Code	Description
087F979E4	GSS_KRB5_S_KG_CTX_BIND-INGS	Peer context binding validation failed
087F979E5	GSS_KRB5_S_KG_BAD_AU-THENTICATOR	Bad GSS Authenticator
087F979E6	GSS_KRB5_S_KG_NO_MECH	No mechanism
087F979E7	GSS_KRB5_S_KG_KRB_ERR	Token is KRB-ERR
087F979E8	GSS_KRB5_S_KG_RECV_ADDR	Decoding requires sender address
087F979E9	GSS_KRB5_S_KG_SENDER_ADDR	Encoding requires sender address
087F979EA	GSS_KRB5_S_KG_BAD_MSG	Token not private or safe message
087F979EB	GSS_KRB5_S_KG_CTX_IN-COMPLETE	Attempt to use incomplete security context
087F979EC	GSS_KRB5_S_KG_BAD_LENGTH	Invalid field length in token
087F979ED	GSS_KRB5_S_KG_BAD_QOP_US-AGE	QOP usage invalid or not supported
087F979EE	GSS_KRB5_S_KG_CONTEXT_ES-TABLISHED	Context is already fully established
087F979EF	GSS_KRB5_S_KG_NO_SUBKEY	Authenticator has no subkey
087F979F0	GSS_KRB5_S_KG_TGT_MISSING	Credential cache has no TGT
087F979F1	GSS_KRB5_S_KG_KEYTAB_NO-MATCH	No principal in keytab matches desired name
087F979F2	GSS_KRB5_S_KG_CCACHE_NO-MATCH	Principal in credential cache does not match desired name
087F979F3	GSS_KRB5_S_G_BAD_KEYGEN	Unable to generate subkey
087F979F4	GSS_KRB5_S_G_BAD_STATUS	Unknown display status code
087F979F5	GSS_KRB5_S_G_UNKNOWN_QOP	Unknown quality of protection specified
087F979F6	GSS_KRB5_S_G_BAD_USAGE	Credential usage type is unknown
087F979F7	GSS_KRB5_S_G_WRONG_SIZE	Buffer is the wrong size

Hex value	Return Code	Description
087F979F8	GSS_KRB5_S_G_BAD_MSG_CTX	Message context invalid
087F979F9	GSS_KRB5_S_G_BUFFER_ALLOC	Couldn't allocate gss_buffer_t data
087F979FA	GSS_KRB5_S_G_VALI-DATE_FAILED	Validation error
087F979FB	GSS_KRB5_S_G_NOSRVC	No SERVICE: prefix in name string
087F979FC	GSS_KRB5_S_G_UID_LEN	Buffer has wrong length for uid_t
087F979FD	GSS_KRB5_S_G_NOUSER	UID does not resolve to user name
087F979FE	GSS_KRB5_S_G_BAD_UID_NO-SUPP	UID not supported on this operating system
087F979FF	GSS_KRB5_S_G_BAD_SER-VICE_NAME	No @ in SERVICE-NAME name string

8.9.5 Kerberos-specific Codes

The Kerberos-specific error codes are minor error codes.

To obtain a text representations of status codes, use the function `gss_display_status( )`.

To see the complete list of error codes that can be returned, see the header files included in the `...\include\csf\sts` and the `...\include\csfc5\sts` directories.

A

Coding Examples

The examples in this appendix illustrate how to use some of the routines in the trusted Tru64 UNIX system.

A.1 Source Code for a Reauthentication Program (sia-reauth.c)

Example A–1 is a program that performs password checking.

Example A–1: Reauthentication Program

```
#include <sia.h>
#include <siad.h>

#ifndef NOUID
#define NOUID ((uid_t) -1)
#endif

main (argc, argv)
int argc;
char **argv;
{
        int i;
 SIAENTITY *entity = NULL;
 int (*sia_collect)() = sia_collect_trm;
 char uname[32];
 struct passwd *pw;
 uid_t myuid;

 myuid = getluid();
 if (myuid == NOUID)
  myuid = getuid(); /* get ruid */
 pw = getpwuid(myuid);
 if (!pw || !pw->pw_name || !*pw->pw_name) {
  sleep(3); /* slow down attacks */
  (void) fprintf(stderr, "sorry");
  return 1;
 }
 (void) strcpy(uname, pw->pw_name);
 i = sia_ses_init(&entity, argc, argv, NULL, uname, \
                                             NULL, TRUE, NULL);
 if (i != SIASUCCESS) {
  sleep(3); /* slow down attacks */
  (void) fprintf(stderr, "sorry");
  return 1;
 }
 i = sia_ses_reauthent(sia_collect, entity);
 if (i != SIASUCCESS) {
  (void) sia_ses_release(&entity);
  sleep(3); /* slow down attacks */
  (void) fprintf(stderr, "sorry");
```

Example A–1: Reauthentication Program (cont.)

```
  return 1;
 }
 i = sia_ses_release(&entity);
 if (i != SIASUCCESS) {
  sleep(3); /* slow down attacks */
  (void) fprintf(stderr, "sorry");
  return 1;
 }

 (void) fprintf(stderr, "Ok");

 return 0;
}
```

A.2 Source Code for a Superuser Authentication Program (sia-suauth.c)

Example A–2 is a program that allows root to become a user to run daemons (such as `crontab` or `sendmail`) for the user.

Example A–2: Superuser Authentication Program

```
#include <sia.h>
#include <siad.h>

main (argc, argv)
int argc;
char **argv;
{
        int i;

        i = sia_auth(getuid());
        printf("result is %d", i);

}

int  sia_auth(uid)
int uid;
{

        char uname[32];
        static SIAENTITY *entity=NULL;
        static int oargc = 1;
        static char *oargv[1] = { "siatest" };
        static int (*sia_collect)()=sia_collect_trm;
 struct passwd *pw;

 pw = getpwuid(uid);
 if (!pw) {
     printf("getpwuid failure");
     return 8;
 }
 (void) strcpy(uname, pw->pw_name);
        printf("SIA authentication for uid: %d, uname: %s ", \
                                                 uid, uname);
        if (sia_ses_init(&entity,oargc,oargv,NULL,uname,NULL, \
```

Example A–2: Superuser Authentication Program (cont.)

```
                                        FALSE, NULL) == SIASUCCESS) {
                printf( "sia_ses_init successful");
                entity->authtype = SIA_A_SUAUTH;
  if (sia_make_entity_pwd(pw, entity) == SIASUCCESS) {
      printf("sia_make_entity_pwd successful");
  }
  else {
      printf("sia_make_entity_pwd un-successful");
  }
                if ((sia_ses_launch(NULL, entity)) == SIASUCCESS) {
                        printf( "sia_ses_launch successful");
                }
                else {
                        printf( "sia_ses_launch un-successful");
   entity = NULL;
                }
                if ((sia_ses_release(&entity)) == SIASUCCESS) {
                        printf( "sia_ses_release successful");
                }
                else {
                        printf( "sia_ses_release un-successful");
                        return(4);
                }

        }
        else {
                printf( "sia_ses_init un-successful");
                return(5);
        }
        printf( "sia  **** successful");
        return(6);
}
```

B
Auditable Events and Aliases

This appendix contains the default auditable events (`/etc/sec/audit_events`) and the default audit event aliases (`/etc/sec/event_aliases`) as they as delivered on Tru64 UNIX.

B.1 Default Auditable Events File

The following is the default `/etc/sec/audit_events` file:

```
!  Audited system calls:
exit                            succeed  fail
fork                            succeed  fail
old open                        succeed  fail
close                           succeed
old creat                       succeed  fail
link                            succeed  fail
unlink                          succeed  fail
execv                           succeed  fail
chdir                           succeed  fail
fchdir                          succeed  fail
mknod                           succeed  fail
chmod                           succeed  fail
chown                           succeed  fail
getfsstat                       succeed  fail
mount                           succeed  fail
unmount                         succeed  fail
setuid                          succeed  fail
exec_with_loader                succeed  fail
ptrace                          succeed  fail
nrecvmsg                        succeed  fail
nsendmsg                        succeed  fail
nrecvfrom                       succeed  fail
naccept                         succeed  fail
access                          succeed  fail
kill                            succeed  fail
old stat                        succeed  fail
setpgid                         succeed  fail
old lstat                       succeed  fail
dup                             succeed  fail
pipe                            succeed  fail
open                            succeed  fail
setlogin                        succeed  fail
acct                            succeed  fail
```

```
classcntl                       succeed  fail
ioctl                           succeed  fail
reboot                          succeed  fail
revoke                          succeed  fail
symlink                         succeed  fail
readlink                        succeed  fail
execve                          succeed  fail
chroot                          succeed  fail
old fstat                       succeed  fail
vfork                           succeed  fail
stat                            succeed  fail
lstat                           succeed  fail
mmap                            succeed  fail
munmap                          succeed  fail
mprotect                        succeed  fail
old vhangup                     succeed  fail
kmodcall                        succeed  fail
setgroups                       succeed  fail
setpgrp                         succeed  fail
table                           succeed  fail
sethostname                     succeed  fail
dup2                            succeed  fail
fstat                           succeed  fail
fcntl                           succeed  fail
setpriority                     succeed  fail
socket                          succeed  fail
connect                         succeed  fail
accept                          succeed  fail
bind                            succeed  fail
setsockopt                      succeed  fail
recvmsg                         succeed  fail
sendmsg                         succeed  fail
settimeofday                    succeed  fail
fchown                          succeed  fail
fchmod                          succeed  fail
recvfrom                        succeed  fail
setreuid                        succeed  fail
setregid                        succeed  fail
rename                          succeed  fail
truncate                        succeed  fail
ftruncate                       succeed  fail
setgid                          succeed  fail
sendto                          succeed  fail
shutdown                        succeed  fail
socketpair                      succeed  fail
mkdir                           succeed  fail
rmdir                           succeed  fail
utimes                          succeed  fail
adjtime                         succeed  fail
sethostid                       succeed  fail
```

```
old killpg                      succeed  fail
setsid                          succeed  fail
pid_unblock                     succeed  fail
getdirentries                   succeed  fail
statfs                          succeed  fail
fstatfs                         succeed  fail
setdomainname                   succeed  fail
exportfs                        succeed  fail
getmnt                          succeed  fail
alternate setsid                succeed  fail
swapon                          succeed  fail
msgctl                          succeed  fail
msgget                          succeed  fail
msgrcv                          succeed  fail
msgsnd                          succeed  fail
semctl                          succeed  fail
semget                          succeed  fail
semop                           succeed  fail
lchown                          succeed  fail
shmat                           succeed  fail
shmctl                          succeed  fail
shmdt                           succeed  fail
shmget                          succeed  fail
utc_adjtime                     succeed  fail
security                        succeed  fail
kloadcall                       succeed  fail
priocntlset                     succeed  fail
sigsendset                      succeed  fail
msfs_syscall                    succeed  fail
sysinfo                         succeed  fail
uadmin                          succeed  fail
fuser                           succeed  fail
proplist_syscall                succeed  fail
ntp_adjtime                     succeed  fail
audcntl                         succeed  fail
setsysinfo                      succeed  fail
swapctl                         succeed  fail
memcntl                         succeed  fail
SystemV/unlink                  succeed  fail
SystemV/open                    succeed  fail
RT/memlk                        succeed  fail
RT/memunlk                      succeed  fail
RT/psx4_time_drift              succeed  fail
RT/rt_setprio                   succeed  fail

!  Audited trusted events:
audit_start                     succeed  fail
audit_stop                      succeed  fail
audit_setup                     succeed  fail
audit_suspend                   succeed  fail
```

```
audit_log_change                succeed  fail
audit_log_creat                 succeed  fail
audit_xmit_fail                 succeed  fail
audit_reboot                    succeed  fail
audit_log_overwrite             succeed  fail
audit_daemon_exit               succeed  fail
login                           succeed  fail
logout                          succeed  fail
auth_event                      succeed  fail
audgen8                         succeed  fail
net_tcp_stray_packet            succeed  fail
net_tcp_syn_timeout             succeed  fail
net_udp_stray_packet            succeed  fail
net_tcp_rejected_conn           succeed  fail

!  Audited mach traps:
lw_wire                         succeed  fail
lw_unwire                       succeed  fail
init_process                    succeed  fail
host_priv_self                  succeed  fail
semop_fast                      succeed  fail

!  Audited mach ipc events:
task_create                     succeed  fail
task_terminate                  succeed  fail
task_threads                    succeed  fail
thread_terminate                succeed  fail
vm_allocate                     succeed  fail
vm_deallocate                   succeed  fail
vm_protect                      succeed  fail
vm_inherit                      succeed  fail
vm_read                         succeed  fail
vm_write                        succeed  fail
vm_copy                         succeed  fail
vm_region                       succeed  fail
task_by_unix_pid                succeed  fail
bind_thread_to_cpu              succeed  fail
task_suspend                    succeed  fail
task_resume                     succeed  fail
task_get_special_port           succeed  fail
task_set_special_port           succeed  fail
thread_create                   succeed  fail
thread_suspend                  succeed  fail
thread_resume                   succeed  fail
thread_set_state                succeed  fail
thread_get_special_port         succeed  fail
thread_set_special_port         succeed  fail
port_allocate                   succeed  fail
port_deallocate                 succeed  fail
port_insert_send                succeed  fail
```

```
port_extract_send              succeed  fail
port_insert_receive            succeed  fail
port_extract_receive           succeed  fail
host_processors                succeed  fail
processor_start                succeed  fail
processor_exit                 succeed  fail
processor_set_default          succeed  fail
xxx_processor_set_default_priv succeed  fail
processor_set_tasks            succeed  fail
processor_set_threads          succeed  fail
host_processor_set_priv        succeed  fail
host_processors_name           succeed  fail
host_processor_priv            succeed  fail
```

B.2 Sample Event Aliases File

The following is the sample `/etc/sec/event_aliases` file provided with the Tru64 UNIX system:

```
# This is a SAMPLE alias list.  Your alias list should be built to
# satisfy your site’s requirements.

obj_creat: "old open" "old creat" link mknod open symlink mkdir SystemV/open

obj_delete: unlink truncate ftruncate SystemV/unlink rmdir

exec: execv exec_with_loader execve

obj_access: access "old stat" "old lstat" "old open" open statfs fstatfs \
    readlink "old fstat" stat lstat fstat close:1:0 dup dup2 fcntl \
    "old creat" mmap munmap mprotect memcntl SystemV/open

obj_modify: chmod chown fchown fchmod lchown utimes rename

ipc: recvmsg nrecvmsg recvfrom nrecvfrom sendmsg nsendmsg sendto accept \
    naccept connect socket bind shutdown socketpair pipe sysV_ipc \
    kill "old killpg" setsockopt sigsendset net_tcp_rejected_conn \
    net_udp_stray_packet

sysV_ipc: msgctl msgget msgrcv msgsnd shmat shmctl shmdt shmget semctl \
    semget semop

proc: exit fork chdir fchdir setuid ptrace setpgid setlogin chroot vfork \
    setgroups setpgrp setpriority setreuid setregid setgid audcntl \
    RT/rt_setprio setsid "alternate setsid" priocntlset

system: getfsstat mount unmount acct reboot table sethostname settimeofday \
    adjtime sethostid setdomainname exportfs getmnt swapon utc_adjtime \
    audcntl setsysinfo kloadcall getdirentries revoke "old vhangup" kmodcall \
    security sysinfo uadmin swapctl

misc: ioctl msfs_syscall fuser

trusted_event: login logout auth_event audgen8

all: obj_creat obj_delete exec obj_access obj_modify ipc proc system misc \
    trusted_event
```

```
#++++++++++++++++++++++++++++++++++++++++++++++++++++++

# adjtime is being called once a sec?

profile_audit: audit_start:1:1 audit_stop:1:1 audit_setup:1:1 audit_log_creat:1:
1 audit_xmit_fail:1:1 \
audit_reboot:1:1 audit_log_overwrite:1:1 audit_daemon_exit:1:1 \
settimeofday:1:1 ntp_adjtime:1:1 utc_adjtime:1:1

profile_net: connect:1:1 accept:1:1 bind:1:1 net_udp_stray_packet:1:1 net_tcp_re
jected_conn:1:1

profile_netmon: net_tcp_rejected_conn:1:1 net_tcp_syn_timeout:1:1 net_tcp_stray_
packet:1:1 net_udp_stray_packet:1:1

profile_auth: login:1:1 logout:1:1 auth_event:1:1

profile_filesys: mount:1:1 unmount:1:1

profile_creat: "old creat" link mknod symlink mkdir

profile_proc: setuid setgid setlogin chroot \
    setsid "alternate setsid"

# Definition of catagories
#=================================================================
# Desktop:
#  Provides suggested minimal auditing configuration for a single user system. C
onfiguration provides
# monitoring of tusted audit events, no monitoring of files, or network related
events.
# ---------------------------------------------------------------------------
#   This alias assumes:
#   - Local access is primarily interactive login, generally limited to one user
#   at a time, activity tracked and controlled by the system.
#   - Individual accountability is primarily maintained by the system.
#   - User related file area access is only limited  by file owner choice.
#   Browsing is unrestricted.
#   - System related file areas are mostly readonly. Browsing is unrestricted.
#   - Login uid is converted to username.
#   - Access to the network is monitored.
#   - Access to controlled files are unmonitored.
Desktop: \
profile_audit \
profile_auth

# Servers:
#  Provides suggested auditing configuration for a system which is used as a ser
ver for networked based
# applications (such as databases, web server, etc.). Configuration provides mon
itoring of trusted
# events, system files, network related files, and network related events.
# ---------------------------------------------------------------------------
#   This alias assumes:
#   -Network access is restricted to application (mail, db server, firewall,
#   etc.) controlled access through network mechanisms (tcp/ip reserved port,
#   DECnet objects, etc.) with the application being responsible for tracking
#   activity. Interactive access is strictly controlled by the system, activity
#   is tracked by the system. Application primarily handle access control,
#   system control is secondary.
#   - Local access logins are strictly controlled, activity is tracked by the
#   system.
#   - Individual accountability is primarily maintained by the applications.
#   - User related file area access is strictly limited to application related
#   files. Browsing is controlled.
#   - system related file areas are at most readonly for user aplication related
#   functions. Browsing is controlled by applications.
#   - Login uid is converted to username.
#   - Access to the network is monitored.
#   - Access to controlled files are monitored.
Server: \
profile_audit \
profile_auth \
profile_net \
profile_filesys \
profile_proc \
profile_creat obj_delete obj_modify


# Timesharing:
#  Provides suggested minimal auditing configuration for a system which is used
to support multiple
# interactive users. Configuration provides monitoring of trusted events, no mon
itoring of system
# files,or network related events or files.
# ---------------------------------------------------------------------------
#   This alias assumes:
#   - Local access is primarily interactive login, activity is tracked and
#   controlled by the system.
#   - Individual accountability is primarily maintained by the system.
#   - Interactive logins are generally unrestricted.
#   - User related file area access is only limited  by file owner choice.
#   Browsing is unrestricted.
#   - System related file areas are mostly readonly. Browsing is unrestricted.
#   - Login uid is converted to username.
#   - Access to the network is unmonitored.
#   - Access to controlled files is unmonitored.
Timesharing: \
profile_audit \
profile_auth

# Timesharing_extended_audit:
#  Provides suggested auditing configuration for a system which is used to suppo
rt multiple interactive
# users. Configuration provides monitoring of trusted events, system files, and
no monitoring of network
# related events or files.
# ---------------------------------------------------------------------------
#   This alias assumes:
#   - Local access is primarily interactive login, activity is tracked and
#   controlled by the system.
#   - Individual accountability is primarily maintained by the system.
#   - Interactive logins are generally unrestricted.#   - User related file area
 access is only limited  by file owner choice.
#   Browsing is unrestricted.
#   - System related file areas are mostly readonly. Browsing is unrestricted.
#   - Access to the network is monitored.
#   - Access to controlled files is monitored.
Timesharing_extended_audit: \
profile_audit \
profile_auth \
profile_filesys \
profile_proc \
profile_creat obj_delete obj_modify

# Networked_system:
#  Provides suggested auditing configuration for a system which has networking e
nabled. Should be used in
```

```
# conjuction with Desktop, Timesharing, or Timesharing_extended_audit templates.
 Configuration provides
# monitoring of trusted events, network related files and network related events
.
# ---------------------------------------------------------------------------
#   This alias assumes:
#   - Network access is through application (mail, pinter, etc.) controlled
#   network mechanisms (tcp/ip reserved port, DECnet objects, etc.) which are
#   responsible tracking activity and controlling access, and Interative login
#   with the system tracking activity and controlling access.
#   - Access to the network is monitored.
#   - Access to controlled files is monitored.
Networked_system: \
profile_audit \
profile_auth \
profile_net \
profile_creat obj_delete obj_modify


# NIS_server:
#  Provides suggested auditing configuration for a system used as a NIS server.
Should be used in
# conjuction with Desktop, Timesharing, or Timesharing_extended_audit templates.
 Configuration provides
# monitoring of trusted events, NIS related files and network related events.
# ---------------------------------------------------------------------------
#   This alias assumes:
#   - Network access is through application (mail, pinter, etc.) controlled
#   network mechanisms (tcp/ip reserved port, DECnet objects, etc.) which are
#   responsible tracking activity and controlling access, and Interative login
#   with the system tracking activity and controlling access. NIS is enabled.
#   - Access to the network is monitored.
#   - Access to controlled files is monitored.
NIS_server: \
profile_audit \
profile_net \
profile_creat obj_delete obj_modify
```

C
GSS-API Tutorial

This appendix shows you how to use GSS-APIs to secure an application using C-programming language example code. It also explains the sample programs provided Application Security SDK. This appendix contains the following information:

- Security primer
- Getting started
- Using Basic GSS-API Functions
- Step 1: Getting Names
- Step 2: Acquiring Credentials
- Step 3: Establishing a Security Context
- Step 4: Exchanging Messages
- Step 5: Terminating the Security Context
- Advanced Concepts
- Status Codes for GSS-API Functions
- Sample Programs

C.1 Security Primer

Read this section if you are unfamiliar with network authentication using Kerberos. The following information is contained in this section:

- Fundamental Concepts
- Kerberos Security Model

C.1.1 Fundamental Concepts

Start with the fundamental concepts explained below:

- Identification — Who are you? A secure system answers this question based on your system identity and a unique personal characteristic such as a smart card or a password.
- Authentication — Can you prove who you are? The authentication process verifies that you are who you say you are, generally through a

third party such as a security server. Authentication is also used to prove the origin of a message.

- Authorization — What are you allowed to do on the system? Authorization depends on authentication. Once the system knows you are who you say you are, your access privileges are allocated based on your identity.
- Confidentiality — Is the message protected from unauthorized disclosure? Encryption is a method for implementing confidentiality. Only people who know how to decrypt the information can access it. Encryption and decryption is usually provided by some kind of key (secret) and an algorithm (public).
- Integrity — Was a message corrupted in transit? Integrity is usually achieved by comparing checksums computed on the message before and after transmission. Message stream integrity is also provided by features such as replay detection, message sequencing, and so forth.

C.1.2 Kerberos Security Model

Kerberos is a two-way third-party authentication and key distribution system that was developed at the Massachusetts Institute of Technology.

C.1.2.1 Definitions

Client—A program or process that makes use of a network service on a user's behalf. Note that in some cases a server may itself be a client of some other server (for example, a print server may be a client of a file server).

User—A person who uses a client program.

Application Servers and Services—A host system which provides services from one side of a distributed application. Users run client programs to access services running on application servers.

Principal—Means for identifying entities using the HP security system, that is, any user, client, service, application, or host system that has stored a secret key with the security server.

Key Distribution Center (KDC)—The security server that contains the principal database, authentication service, and ticket-granting service. These KDC components provide principals with credentials that allow the principals to prove their identity and subsequently exchange messages:

- The *principal database* holds the secret keys for all principals used in network authentication.
- The *Authentication Service* verifies the identity of a principal and issues *Ticket-Granting Tickets* (TGT).

- The *Ticket-Granting Service* distributes service tickets after validating the TGT and the authenticator.

Principal Database—A database (usually located on the same machine as the KDC) that holds the shared secrets (that is, secret keys) used in Kerberos authentication. Each user has a principal in the database.

Realm—A name for a collection of principals in a network served by one or more KDCs. All principal names within a realm must be unique. Within a realm, the administration policy is the same for all principals.

Service—An application running on a server that is available to clients on a network.

Key—A secret value such as a password that is used during the process of encryption and decryption.

Ticket—A credential used to authenticate one principal to another. Only a KDC can issue a Kerberos ticket.

C.1.2.2 Concepts and Processes

The following sections describe Kerberos concepts and processes.

C.1.2.2.1 A Shared Secret

Two principals that wish to communicate with each other securely first need to share a secret key. This secret key, called a session key, is generated randomly and given to the two principals and no one else. The session key is used by both principals for authentication and encryption.

C.1.2.2.2 Trusted Third Party Arbitration

Each principal has a secret key, which remains unchanged until the principal changes it. The session key, on the other hand, is random and exists only for as long as a session is needed between the two principals.

A service doesn’t trust a client, and a client doesn’t trust a service. The client and service trust only the Key Distribution Center, which generates and distributes session keys.

C.1.2.2.3 The Kerberos Network

A Kerberos network is divided into security domains, called realms. Each realm has its own authentication server (KDC) and implements its own security policy. This allows organizations implementing Kerberos to have different levels of security for different information classes within the organization.

A realm can accept authentication from other realms or not accept them without a re-authentication if the information security policy requires re-authentication. This is called inter-realm authentication.

Realms are hierarchical. That is, each realm may have child realms, and each realm may have a parent. This structure allows realms that have no direct contact to share authentication information.

C.1.2.2.4 Three Phases to Authentication

There are three phases to the Kerberos authentication process:

1. The client obtains an initial credential (ticket-granting ticket or TGT) for use in requesting access to other services by using their password.

   The client and the KDC exchange Authentication Service messages when acquiring initial credentials.
2. The client requests authentication to a specific service.

   The client and the KDC exchange Ticket-Granting Service messages when acquiring a service ticket.
3. The client presents a service ticket to the service.

   The client and the requested service exchange application messages to do this.

C.1.2.2.5 Authentication Service Message Exchange

The Authentication Service (AS) exchange provides the client with its initial credentials and a session key (shared secret) to be used with the Ticket-Granting Service (TGS). These credentials can be presented to the Ticket-Granting Service to request credentials (that is, a service ticket) that prove the client's identity to a specific service.

This message exchange consists of a request generated by the client and the reply generated by the KDC Authentication Service. At the end of this exchange, the client has initial credentials (the Ticket-Granting Ticket) that allow it to authenticate itself to the Ticket-Granting Service.

When the Authentication Service receives the request message from the client, it checks to see if it knows the client. If it does, it generates a random session key (which becomes a shared secret between the client and the Ticket-Granting Service) and the TGT (which is used by the client to authenticate itself to the Ticket-Granting Service).

The Authentication Service sends both the session key and the TGT as part of a reply message. Since the Authentication Service doesn't know if the client is authentic, it encrypts part of the reply message using the client's secret key, knowing that only the authentic client will be able to decrypt it.

C.1.2.2.6 Ticket-Granting Service Message Exchange

A ticket is only good for a single service; this includes the TGT, which is a ticket to use the services of the Ticket-Granting Service. As such, it is necessary to obtain a separate service ticket for each service the client wants to use. The client obtains service tickets from the ticket-granting service by sending a message to the KDC Ticket-Granting Service. No prompt for a username and password is required in this case since the TGT is used to prove identity.

When a client presents a ticket (including the TGT) to a service, it also presents an authenticator. The authenticator contains the client's name encrypted in the session key of the ticket. In this way, if the service can decrypt the authenticator using the ticket's session key, and the client's name matches the one in the ticket, it can be guaranteed that the client is authentic. It certifies a client's knowledge of a session key and aids in detection of an unauthorized replay of credentials. Authenticators can only be used once.

After receiving the ticket request message, the Ticket-Granting Service decrypts the TGT. The session key is then used to decrypt the authenticator. If the client name matches in both the authenticator and the TGT, the client has successfully authenticated itself to the Ticket-Granting Service and can be granted a service ticket.

The service ticket is sent to the client in the reply message encrypted in the service's secret key. The encrypted part of the message is encrypted using the TGT session key instead of the client's secret key. This way, there is no need for the user to enter her/his password again.

After receiving the reply message, the client decrypts the encrypted part. It then stores the service ticket and the corresponding session key in the credentials cache.

C.1.2.2.7 Application Message Exchange

After a client obtains a service ticket for a service, it can authenticate and exchange messages with the service securely. The client sends the service ticket along with an authenticator to the service for verification.

C.1.2.3 Credential Attributes

The client can request that the credentials issued to it by the KDC carry certain characteristics:

- LIFETIME — credentials are issued for a specific length of time. This time is usually determined by the local security policy.

- RENEWABLE — an attribute that allows the client to renew its credentials before they expire without entering a principal name and password.
- POSTDATED — the client can request a POSTDATED ticket for later use. For example, this ticket could be used by applications to process batch jobs.
- FORWARDABLE — these credentials are used when it is necessary for a client to allow a service to perform an operation on its behalf using the client's ticket.

C.2 Getting Started

Initial decisions you need to think about before using Application Security SDK to secure a distributed application are listed below:

1. Before the application is started, will an initial credential (TGT) already exist for the application user?

   If no, the application will need to establish a *user context* by calling `csf_gss_acq_user( )`. This function fetches a initial credential (TGT) from the security server for the application user. These credentials are then used by the initiating application during subsequent processes, and are frequently referred to in this documentation as the initiator's credentials.

   If DES3 encryption or public-key credentials are being used, the application will need to pass these attributes to `csf_gss_acq_user( )`.
2. Will the application require mutual authentication?

   Mutual authentication is recommended when the initiating application needs to verify the identify of the accepting application, such as in the case of submitting sensitive data like a credit card number to the accepting application.

   If yes, the initiating application will need to request mutual authentication during context establishment when it calls `gss_init_sec_context( )`.
3. What messages (if any) need to be secured between the application and its peer?

   In some cases, authentication is the only needed service required by an application.
4. In the case where messages need to be protected, what kind of protection is required?

   Types of protection fall into two basic categories: confidentiality and integrity.

- Confidentiality (or privacy) protects a message from being viewed by an intruder.
- Integrity protects a message from being tampered with by an intruder. For example, during a financial transaction, integrity services would be used to protect the currency amount from being altered, or from the account number from being changed.

5. What Quality of Protection (QOP) is required for message protection?

   Lowest quality (DES encryption) will be the fastest but less secure; highest quality (DES3 encryption) will be the slowest but most secure. If the application is using DES3 encryption, there are several prerequisites that must be met.

C.3 Using Basic GSS-API Functions

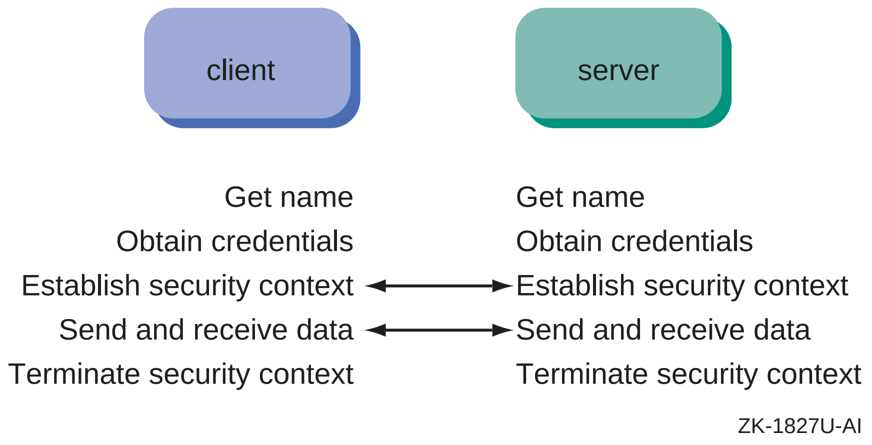
In the GSS-API world, a security context is either *initiated* or *accepted*. The role of the application determines whether it is initiating or accepting the security context. In a client-server application, typically the client initiates the security context and the server accepts the security context. A server may also act as a client of another secure service.

The process of implementing security between an initiator and an acceptor using GSS-API functions entails five major steps:

1. Getting names
2. Acquiring credentials
3. Establishing a security context
4. Exchanging messages
5. Terminating the security context

Both the initiating and the accepting applications should incorporate the steps in the same order. However, not all the steps may be necessary. For example, acquiring credentials can be omitted if default credentials are to be used.

Both applications must execute these steps in the order shown below. For example, each application must acquire credentials before a context can be established between them.



In this explanation of basic GSS functions, it is assumed that the initiating application (or initiator), is a user and the accepting application (or acceptor) is a service.

This explanation further assumes that both applications already possess initial credentials, that is, a TGT or a service key, from a security server. If you want to acquire the initial credentials at runtime, you must use the HP GSS-API extension `csf_gss_acq_user( )` to establish a user context.

The functions described for each of the steps in this section are a subset of the available GSS-API functions. They represent the minimum calls necessary to secure an application with Application Security SDK.

C.4 Step 1: Getting Names

Before you can secure an application, you must get names for the parties involved. This step demonstrates how to import a user name for the initiator and a service name for the acceptor. These names are used to acquire credentials and establish a security context.

Use the `gss_import_name( )` function to convert a name to an internal format which other GSS-API functions can use. There are many formats, or name types, from which to choose for the conversion.

An internal form name is unintelligible if printed out but you can convert it into a human-readable form using `gss_display_name( )`. This function is handy for displaying the principal associated with an existing credential if the default credential was acquired.

The same process occurs when either an initiator or the acceptor imports and displays a principal name, either a user name or a service name.

GSS-API Name Management Functions describes what you can do with other naming functions.

Note

The call by the initiator to `gss_import_name( )` is optional. If `gss_import_name( )` was not called by the initiator for itself, then the default principal name is retrieved from the credentials cache when the `gss_acquire_cred( )` or `gss_init_sec_context( )` call is made if default credentials are requested. The initiator must still import the name of the acceptor as it is required by the `gss_init_sec_context( )` call.

The call by the acceptor to `gss_import_name( )` is also optional. If `gss_import_name( )` was not called by the acceptor for itself, then the default service principal name (`host/`*`hostname@REALM`*) is created when the `gss_acquire_cred( )` or `gss_accept_sec_context( )` call is made.

The `gss_import_name( )` function includes the following parameters:

Input Parameters	
`input_name_buffer`	Text name to be converted
`input_name_type`	Type of name — public key, Kerberos, and generic name types are supported
Output Parameters	
`minor_status`	Kerberos 5 error code
`output_name`	Returned name in internal form

The following excerpt from the sample programs illustrates importing a name for the initiator and converting it to human-readable form. The name type specifies the Kerberos 5 default which, in this case, is `GSS_C_NT_USER_NAME`, a name type that is commonly used for user principals. This name type converts a user name user into the form *`user@LOCAL_REALM`*.

```
/***************************************************************/

char            szUserName[]      = "walth@APPSEC.CYBERSAFE.COM";
OM_uint32       gMaj, gMin        = 0;
gss_buffer_desc input_name        = GSS_C_EMPTY_BUFFER;
gss_buffer_desc display_name      = GSS_C_EMPTY_BUFFER;
gss_name_t      principal_name    = GSS_C_NO_NAME;
gss_OID         display_name_type = GSS_C_NO_OID;

/* Copy the principal name into the GSS buffer structure */
```

```
input_name.length = strlen( szUserName );
input_name.value  = strdup( szUserName );

/* Convert the user principal name into GSS internal form */
gMaj = gss_import_name( &gMin,
                        &input_name,
                        GSS_C_NT_USER_NAME,
                        &principal_name );

/* Convert the imported name to human-readable form */
gMaj = gss_display_name( &gMin,
                         principal_name,
                         &display_name,
                         &display_name_type );

/****************************************************************/
```

The following excerpt from the sample programs illustrates importing a name for the acceptor and converting it to human-readable form. The name type specifies the Kerberos 5 default which, in this case, is `GSS_KRB5_NT_HOSTBASED_SERVICE_NAME`, a name type that is often used for an unattended host. This name type converts the Kerberos host service principal *`service@host`* into the form *`service/fqdn@REALM`*. As an example, the converts `ftp@fuji` to `ftp/fuji.company.com@COMPANY.COM`.

```
/****************************************************************/

char            szHostName[]      = "host@dev007.cybersafe.com";
OM_uint32       gMaj, gMin        = 0;
gss_buffer_desc input_name        = GSS_C_EMPTY_BUFFER;
gss_buffer_desc display_name      = GSS_C_EMPTY_BUFFER;
gss_name_t      principal_name    = GSS_C_NO_NAME;
gss_OID         display_name_type = GSS_C_NO_OID;

/* Copy the host service principal name
   into the GSS buffer structure */
input_name.length = strlen( szHostName );
input_name.value  = strdup( szHostName );

/* Convert the host service principal name
   into GSS internal form */
gMaj = gss_import_name( &gMin,
                        &input_name,
                        GSS_KRB5_NT_HOSTBASED_SERVICE_NAME,
                        &principal_name );

/* Convert the imported name to human-readable form */
gMaj = gss_display_name( &gMin,
                         principal_name,
                         &display_name,
                         &display_name_type );

/****************************************************************/
```

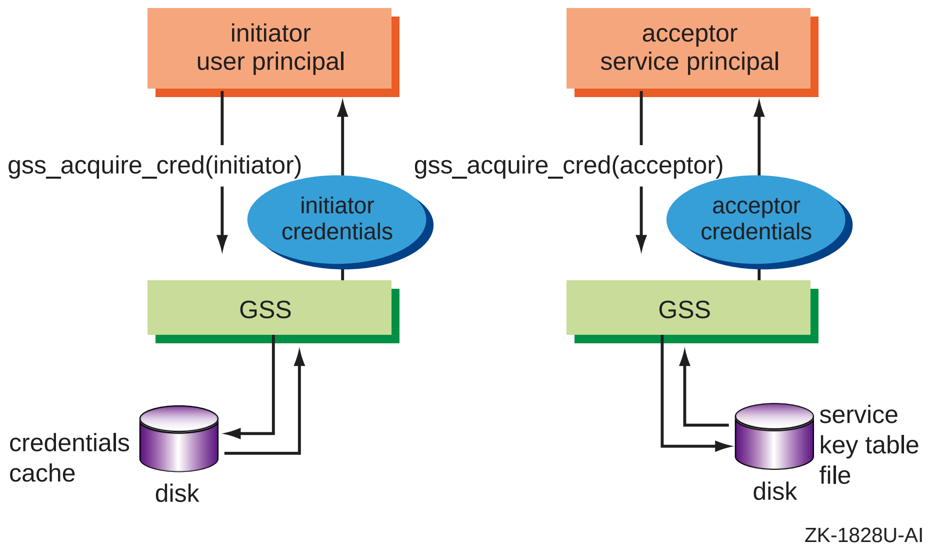
C.5 Step 2: Acquiring Credentials

This step demonstrates how credentials can be acquired for the initiator and the acceptor. The credentials contain TGTs and service key table entries from the security server that the initiator and the acceptor use to authenticate themselves to each other. Therefore, both the initiator and the acceptor must obtain credentials before a security context can be established.

Three types of credentials may be requested:

- `GSS_C_INITIATE`— for an application that is going to initiate a security context by calling `gss_init_sec_context( )`.
- `GSS_C_ACCEPT` — for an application that is going to accept a security context by calling `gss_accept_sec_context( )`.
- `GSS_C_BOTH` — for an application that is going to be both initiating and accepting security contexts, in other words, acting as both a client and a server.

As shown below, acquiring credentials is a local process. No communication with a peer application is needed.



The `gss_acquire_cred( )` function uses an imported name to look for a valid ticket associated with the user or service name. The credentials must already exist in the HP credentials cache for the initiator or the service key table for the acceptor. An exception is an acceptor application that also initiates security contexts with other acceptor applications. A common example is a proxy. In this case, both a credentials cache and a service key table file entry is required to support either role.

The service key table file is named `v5srvtab` by default on the Windows and UNIX platforms.

When `gss_acquire_cred( )` is called by an *initiator* requesting `GSS_C_INITIATE` credentials, the function verifies that a principal name was passed in and looks for a corresponding TGT in the credentials cache. If a principal name was *not* specified, the function looks for a valid TGT in the credentials cache using a default name which is the same as default principal name of the credential cache. (User login name is *not* used.)

When `gss_acquire_cred( )` is called by an *acceptor* requesting `GSS_C_ACCEPT` credentials, the function verifies that a principal name was passed in and looks for a corresponding entry in the service key table file. If a principal name was not specified, the function looks for a service key table file entry corresponding to the default acceptor principal name of `host/`*`fqdn@REALM`*.

An application that is going to be both initiating and accepting security contexts should have both a credentials cache and a service key table. In this case, when `gss_acquire_cred( )` is called by the application requesting `GSS_C_INITIATE` credentials, the function verifies that a principal name was passed in and looks for a valid TGT for the principal name in the credentials cache. If a TGT is not found in the credentials cache, the function looks for an appropriate entry in the service key table. If a service key table entry is found, the function fetches a TGT from the KDC using the key from the table file.

Note

The initiator's call to `gss_acquire_cred( )` is optional. If the initiator does not call `gss_acquire_cred( )` before the `gss_init_sec_context( )` call is made, credentials with the default name and type `GSS_C_INITIATE` are acquired.

The acceptor's call to `gss_acquire_cred( )` is also optional. In this case, `gss_accept_sec_context( )` obtains credentials with the default name and type `GSS_C_ACCEPT`. You will have improved performance if the accepting application acquires the credentials with `gss_acquire_cred( )`, and then re-uses them as required.

The following parameters are used by `gss_acquire_cred( )`:

Input Parameters	
`desired_name`	Principal or service name requiring credentials. If unspecified, the default principal is used.

Input Parameters	
`time_req`	Credential lifetime requested (ignored). You cannot specify the lifetime of the credentials because the credentials already exist. The lifetime is dictated by the circumstances under which the credentials were created.
`desired_mechs`	Security mechanism desired (Kerberos 5)
`cred_usage`	Credentials usage (initiate, accept, or both)
Output Parameters	
`minor_status`	Kerberos 5 error code
`output_cred_handle`	Credentials acquired
`actual_mechs`	Security mechanism used (Kerberos 5)
`time_rec`	Credential lifetime received. This is only supported for initiators; acceptor lifetimes are indefinite.

The following excerpt from the sample programs illustrates acquiring `GSS_C_INITIATE` credentials for the initiator and `GSS_C_ACCEPT` credentials for the acceptor. The principal names were imported in Step 1.

```
/*****************************************************************\


OM_uint32     gMaj, gMin              = 0;
OM_uint32     cred_lifetime_requested = 600;
OM_uint32     cred_lifetime_received  = 0;
i_32          cred_usage              = GSS_C_INITIATE;
gss_cred_id_t cred_handle             = GSS_C_NO_CREDENTIAL;
gss_OID_set   actual_mechs            = GSS_C_NO_OID_SET;

/* Get an Initiator credential from the cred cache */
gMaj = gss_acquire_cred( &gMin,
                         principal_name,
                         cred_lifetime_requested,
                         rfc_krb5_c_OID_set,
                         cred_usage,
                         &cred_handle,
                         &actual_mechs,
                         &cred_lifetime_received );
     :
     :
     :

OM_uint32     gMaj, gMin             = 0;
OM_uint32     cred_lifetime_received = 0;
i_32          cred_usage             = GSS_C_ACCEPT;
gss_cred_id_t cred_handle            = GSS_C_NO_CREDENTIAL;
gss_OID_set   actual_mechs           = GSS_C_NO_OID_SET;

/* Get an Acceptor credential from the cred cache
   or from the v5srvtab key table file */
gMaj = gss_acquire_cred( &gMin,
                         principal_name,
                         GSS_C_INDEFINITE,
                         GSS_C_NO_OID_SET,
                         cred_usage,
                         &cred_handle,
                         &actual_mechs,
                         &cred_lifetime_received );

/**************************************************************\
```

After the credentials are acquired, it is necessary to free storage allocated using the `gss_release_cred( )` function. The sample programs contain examples.

In general, if a buffer was locally allocated by the user (using `malloc` or `strdup`, for example), the application must free it. If a buffer was allocated by a GSS-API function, it must be freed using `gss_release_xxx( )` functions. These actions prevent memory leaks.

Credential Management Functions shows you what you can do with other credentials functions.

C.6 Step 3: Establishing a Security Context

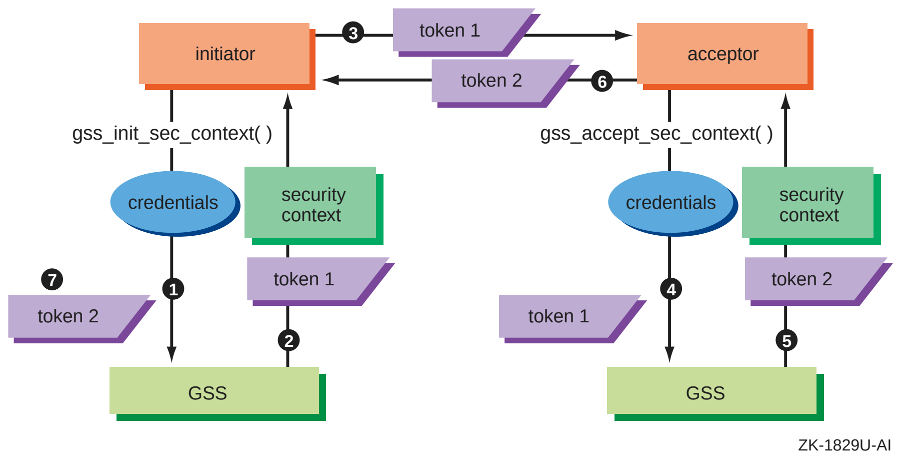
Once an application and its peer have credentials, a security context can be established. A security context is essentially a set of security parameters used to secure a conversation between the application and its peer.

Several security contexts may exist simultaneously between an application and its peer. However, one context should not be used for more than one conversation. For example, in a client-server application, if ten clients were communicating with a server application, each client would have a unique security context and the server would manage ten security contexts, one for each client.

When initiating a context, an application can select various security options which are discussed further in Security Context Management Functions:

- Channel bindings
- Confidentiality and integrity
- Replay detection
- Out-of-sequence message detection
- Mutual authentication
- Encryption algorithm (DES or DES3)
- Ticket forwarding (credentials delegation)

The sequence of steps taken when an application establishes a security context is described below. Several invocations of the function calls used by the initiator and the acceptor may be necessary before the context is fully established.



1. The initiator starts a security context with the acceptor by calling `gss_init_sec_context( )`, passing the credentials it previously acquired and the identity used by the acceptor.
2. The `gss_init_sec_context( )` function returns a partially-built security context plus a token and a status code.

   The token contains GSS-API information that must be sent to the acceptor. The token is opaque to the initiating application, although the application knows the length of the token so it can transfer the token to the acceptor.

   The status code indicates whether the function will need to be called again. If the function is called again, a return token must be received from the acceptor and passed to it.

   A token may even be returned when an error status is returned. Regardless of the status code returned, the token must be transferred to the other application.
3. The initiator sends the token across the communications protocol to the acceptor, which is waiting for a context to be established.

   How tokens are transferred is application dependent. However, the GSS-API standard requires that tokens be sent in the sequence in which they are generated during context establishment.
4. The acceptor reads the token from the communications protocol and calls `gss_accept_sec_context( )`, passing the credentials it previously acquired and the token it received.
5. The `gss_accept_sec_context( )` function returns a status code and may return a token.

The status code indicates whether the function will need to be called again. If the function is called again, a return token must be received from the initiator and passed to it.

A token may even be returned when an error status is returned. Regardless of the returned status code, the token must be transferred to the other application.

6. If gss_accept_sec_context( ) returned a token, the acceptor must send it across the communications protocol to the initiator, which is waiting for this token to continue the security context establishment.
7. If the initiator received a return token, it calls gss_init_sec_context( ) and passes the token to it.

Steps 1 through 5 show the minimum set of exchanges between the initiator and the acceptor. Steps 6 and 7 illustrate multiple invocations of the gss_init_sec_context( ) and gss_accept_sec_context( ) functions. These functions loop when the GSS_S_CONTINUE major status is returned. When a token is returned from gss_init_sec_context( ), it must be sent to the acceptor. If GSS_S_CONTINUE is returned, a token is expected back from the acceptor and must be provided as input to the next invocation of gss_init_sec_context( ). This behavior typically occurs when Kerberos performs mutual authentication.

When a GSS_S_CONTINUE status is returned as the major status code and any token returned by the call is sent to the peer application, the security context is fully established on the local application.

The gss_init_sec_context( ) function uses the following parameters:

Input Parameters	
initiator_cred_handle	Credentials for the initiator (or default credentials from the credentials cache).
target_name	Name of the acceptor
mech_type	Security mechanism desired (Kerberos 5)
req_flags	Service options such as DES3/DES encryption, credentials delegation, mutual authentication, and detection of replayed or out-of-sequence messages
time_req	Context lifetime requested (ignored)
input_chan_bindings	Channel bindings
input_token	Token returned from the acceptor
Output Parameters	
minor_status	Kerberos 5 error code

Input Parameters	
`context_handle`	Security context to be initiated
`actual_mech_type`	Security mechanism used (Kerberos 5)
`output_token`	Token to be sent to the acceptor
`ret_flags`	Flags identifying the service options supported by the context
`time_rec`	Context lifetime received (always indefinite)

The following excerpt from the sample programs illustrates how to use the `gss_init_sec_context( )` call to establish a security context with the following options:

- Mutual authentication
- Replay detection
- Out-of-sequence message detection

Since no encryption method is specified, it defaults to that assigned to the credentials in the principal database.

```
/***************************************************************/

char            szHostName[]      = "host@dev007.cybersafe.com";
OM_uint32       gMaj, gMin        = 0;
OM_uint32       req_flags         = GSS_C_MUTUAL_FLAG |
                                    GSS_C_REPLAY_FLAG |
                                    GSS_C_SEQUENCE_FLAG;
OM_uint32       req_time          = 3600;
OM_uint32       ret_flags         = 0;
OM_uint32       ret_time;         = 0;
gss_buffer_desc input_name        = GSS_C_EMPTY_BUFFER;
gss_buffer_desc input_token       = GSS_C_EMPTY_BUFFER;
gss_buffer_desc output_token      = GSS_C_EMPTY_BUFFER;
gss_buffer_t    token_handle      = GSS_C_NO_BUFFER;
gss_ctx_id_t    context_handle    = GSS_C_NO_CONTEXT;
gss_name_t      service_name      = GSS_C_NO_NAME;
gss_OID         actual_mech_type  = GSS_C_NO_OID;

/* Copy the host service principal name
   into the GSS buffer structure */
input_name.length = strlen( szHostName );
input_name.value  = strdup( szHostName );

/* Convert the host service principal name
   into GSS internal form */
gMaj = gss_import_name( &gMin,
                        &input_name,
                        GSS_KRB5_NT_HOSTBASED_SERVICE_NAME,
                        &service_name );

/* Establish a context. The default claimant credential is
   specified. In that case, the credential is pulled from the
   credential cache. Alternatively, the claimant credential
   could have been acquired via gss_acquire_cred and passed
   to gss_init_sec_context. */
do {
    token_handle = input_token.length ?
                   &input_token : GSS_C_NO_BUFFER;

    /* Initiate a security context with the Acceptor */
    gMaj = gss_init_sec_context( &gMin,
                                 GSS_C_NO_CREDENTIAL,
                                 &context_handle,
                                 service_name,
                                 GSS_C_NO_OID,
                                 req_flags,
                                 req_time,
                                 GSS_C_NO_CHANNEL_BINDINGS,
                                 token_handle,
                                 &actual_mech_type,
                                 &output_token,
                                 &ret_flags,
                                 &ret_time);


    /* If there is a token to send, regardless of
       error condition, then send it to the peer */
    if ( output_token.length != 0 ) {
        SendToken( &output_token );
    }

    /* Clear the buffers for the next iteration */
    Local_Release_Buffer( &input_token );
    gss_release_buffer( &tmpMinor, &output_token );

    /* Now is the proper time to check the status code */
    if ( GSS_ERROR( gMaj ) {
        /* Process error and exit */
    }

    /* If context establishment requires a token in response,
       then fetch it from the peer */
    if ( gMaj & GSS_S_CONTINUE_NEEDED ) {
        RecvToken( &input_token );
    }

} while ( gMaj & GSS_S_CONTINUE_NEEDED );

/*****************************************************************/
```

After the context is no longer needed, it should be destroyed using the `gss_delete_sec_context( )` function. The sample programs contain examples.

The `gss_accept_sec_context( )` function uses the following parameters:

Input Parameters	
`context_handle`	Security context being established
`acceptor_cred_handle`	Credentials for the acceptor
`input_token_buffer`	Token from the initiator
`input_chan_bindings`	Channel bindings
Output Parameters	

`minor_status`	Kerberos 5 error code
`src_name`	Name of the initiator
`mech_type`	Security mechanism used (Kerberos 5)
`output_token`	Token to be sent to the initiator
`ret_flags`	Flags identifying the service options supported by the context
`time_rec`	Context lifetime received (always indefinite)
`delegated_cred_handle`	Delegated credentials from the context initiator

The following excerpt from the sample programs illustrates how to use the `gss_accept_sec_context( )` call to establish a security context.

```
*****************************************************************

OM_uint32       gMaj, gMin       = 0;
OM_uint32       ret_flags        = 0;
OM_uint32       ret_time         = 0;
OM_uint32       ctx_flags        = 0;
gss_buffer_desc input_token      = GSS_C_EMPTY_BUFFER;
gss_buffer_desc output_token     = GSS_C_EMPTY_BUFFER;
gss_cred_id_t   delegated_cred   = GSS_C_NO_CREDENTIAL;
gss_ctx_id_t    context_handle   = GSS_C_NO_CONTEXT;
gss_name_t      src_name         = GSS_C_NO_NAME;
gss_OID         actual_mech_type = GSS_C_NO_OID;

/* Loop based on the example in
   "draft-ietf-cat-gssv2-cbind-07.txt" */
do {
    OM_uint32  tmpMinor = 0;

    /*  Fetch next security context token from the Initiator */
    RecvToken( &input_token );

    /*  Wait for the Initiator to try to establish a security
        context. The acceptor will use a previously acquired
        acceptor credential. */
    gMaj = gss_accept_sec_context( &gMin,
                                   &context_handle,
                                   cred_handle,
                                   &input_token,
                                   GSS_C_NO_CHANNEL_BINDINGS,
                                   &src_name,
                                   &actual_mech_type,
                                   &output_token,
                                   &ret_flags,
                                   &ret_time,
                                   &delegated_cred );

    /* If the authentication routine forms a token to send to
       the Initiator, then send it now -- before function error
       checking, so that if an error token is generated, the
       Initiator will receive it. */
    if ( output_token.length != 0 ) {
        SendToken( &output_token );
    }

    /* Clear the buffers for the next iteration */
    Local_Release_Buffer( &input_token );
    gss_release_buffer( &tmpMinor, &output_token );

    /* Now is the proper time to check the status code */
    if ( GSS_ERROR( gMaj ) ) {
        /* Process error and exit */
    }

} while ( gMaj & GSS_S_CONTINUE_NEEDED );

*****************************************************************
```

After the context is no longer needed, it should be destroyed using the `gss_delete_sec_context( )` function. The sample programs contain examples.

C.7 Step 4: Exchanging Messages

After credentials are acquired and the security context is established between an application and its peer, messages can be secured and sent. This step explains the options for securing messages that are available in Application Security SDK.

Version 2 of the GSS-API standard has two pairs of functions for securing data:

- `gss_get_mic( )` and `gss_verify_mic( )` provide message integrity.
- `gss_wrap( )` and `gss_unwrap( )` provide message integrity and, optionally, confidentiality.

When a sending application calls one of these functions, a token is returned by the function. The token contains a signature computed on the message being sent using an integrity algorithm. In the case of `gss_wrap( )`, the token may also contain the message itself which, optionally, can be encrypted for confidentiality.

Application Security SDK provides four integrity algorithms:

- DES3 MD5
- DES MAC
- DES MD5
- MD5

Two encryption algorithms are provided: DES3 and DES.

The application then uses a communications protocol such as TCP/IP to send the token (and the message if it is not included in the token) to the receiving application. The receiving application calls the complementary function to perform the reciprocal action on the data passed to it.

These functions offer three options for securing data:

1. Message and signature are sent separately.

   With this option, the token contains a signature created by the sending application using the `gss_get_mic( )` function. The message is not part of the token.

   When the receiving application gets the token and the message, it calls the `gss_verify_mic( )` function which verifies the signature to ensure that the message and the signature were not modified in transit.

2. Message and signature sent together.

   With this option, the token contains a signature created by the sending application using the `gss_wrap( )` function. The message is encapsulated in the token.

   When the receiving application gets the token, it calls the `gss_unwrap( )` function, which verifies the signature to ensure that the message and the signature were not modified in transit.

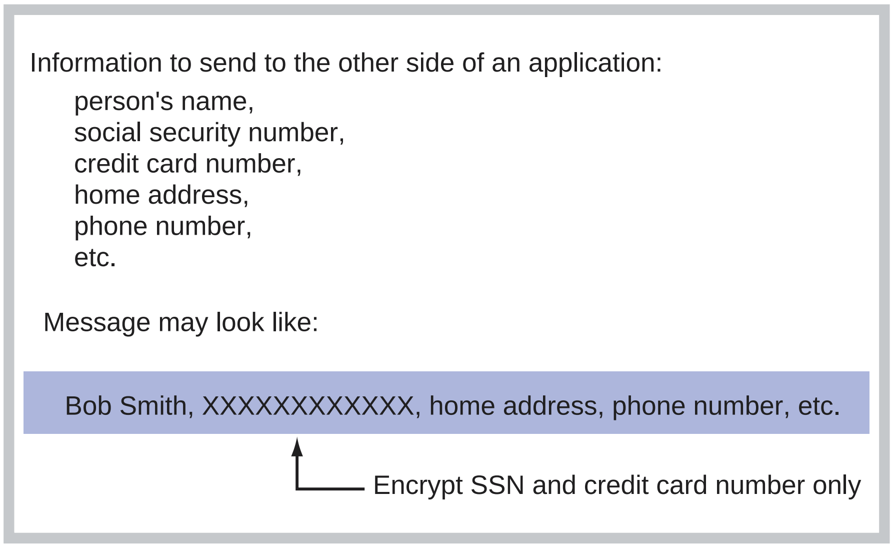
3. Encrypted message and signature sent together.

   This option is the same as option 2 except that the message and the signature are encrypted by the `gss_wrap( )` function and decrypted by the `gss_unwrap( )` function.

Each option serves a specific use when securing distributed application data.

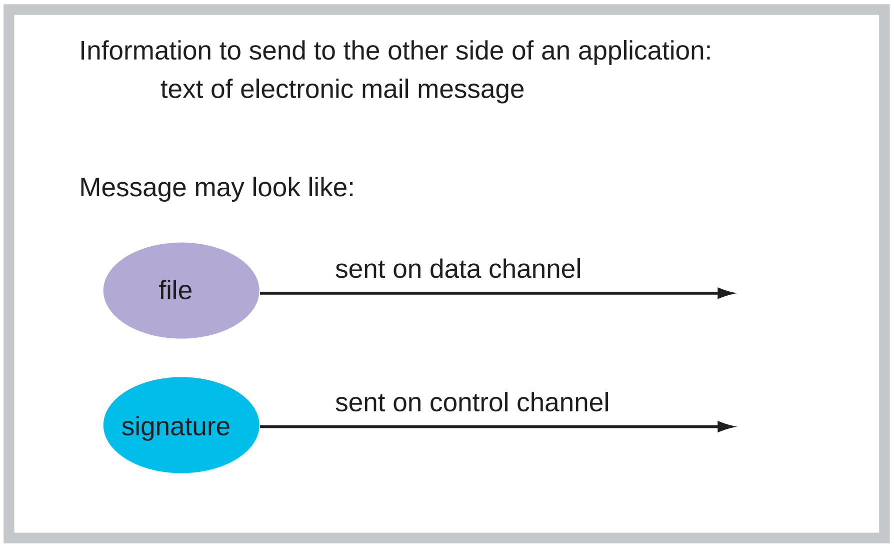
For example, if an application has some data that needs to be secured and other data that does not need to be secured, the application can choose to secure only the data that must be secured. These considerations are application dependent, based on the length of the data and the speed, efficiency, and strength of the encryption algorithm. If the data length is large, it may be efficient to secure only a piece of the message; however, it is not unusual to secure the entire message.

An example of when an application might use `gss_wrap( )` with confidentiality is below.


Information to send to the other side of an application:
person's name,
social security number,
credit card number,
home address,
phone number,
etc.
Message may look like:
Bob Smith, XXXXXXXXXXXX, home address, phone number, etc.
Encrypt SSN and credit card number only

ZK-1830U-AI

In another example, if an application wanted to ensure that a message's integrity was preserved during transmission, and the application requires that data be sent on one channel and checksum on another, out-of-band (control), channel such as ftp, the application needs to use the `gss_get_mic( )` function as shown below.

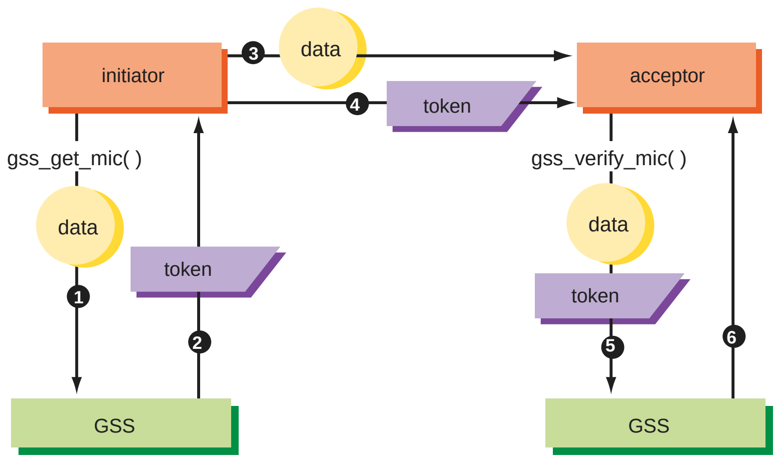

Information to send to the other side of an application:
text of electronic mail message
Message may look like:
file
sent on data channel
signature
sent on control channel

ZK-1831U-AI

C.7.1 Using gss_get_mic( ) and gss_verify_mic( )

Calling `gss_get_mic( )` allows an application to create a signature for data that can be verified by a peer application using `gss_verify_mic( )`. This ensures message integrity, that is, that the message and its signature were not modified in transit.

The sequence of steps for using these functions to send a message from an initiator to an acceptor follows:



ZK-1832U-AI

1. The initiator calls `gss_get_mic( )`, passing the data for which it wants to have a signature generated.
2. The function returns a token, which contains a signature for the data, but not the data itself.
3. The initiator sends the data to the acceptor.

   The token and data can be sent in any order, as separate messages, within the same message, or any other option.
4. The initiator sends the token to the acceptor.
5. The acceptor calls `gss_verify_mic( )`, passing the data and the token it received from the initiator.
6. The `gss_verify_mic( )` function returns a status indicating whether the signature was successfully verified against the data.

The `gss_get_mic( )` function includes the following parameters:

Input Parameters	
`context_handle`	Security context being used
`qop_req`	Quality of protection (QOP) requested.
`message_buffer`	Message to be protected
Output Parameters	
`minor_status`	Kerberos 5 error code
`message_token`	Token containing a signature

The following excerpt from the sample programs illustrates sending a message with this function call. The default QOP is specified which, in this case, is determined by the encryption type specified for the security context.

```
/****************************************************************/

char             szMessage[]   = "This is a test message";
OM_uint32        gMaj, gMin    = 0;
gss_buffer_desc  input_buffer  = GSS_C_EMPTY_BUFFER;
gss_buffer_desc  output_buffer = GSS_C_EMPTY_BUFFER;
gss_qop_t        qop_req       = GSS_C_QOP_DEFAULT;

/* Stuff plaintext message into the input buffer*/
input_buffer.value  = strdup( szMessage );
input_buffer.length = strlen( szMessage );

/* Create a message integrity code (MIC) from the message */
gMaj = gss_get_mic( &gMin,
                    context_handle,
                    qop_req,
                    &input_buffer,
                    &output_buffer );

/* Send the MIC token */
SendToken( &output_buffer );

/****************************************************************/
```

Storage associated with the token should be freed after use using the `gss_release_buffer( )` function to avoid memory leaks.

The `gss_verify_mic( )` function includes the following parameters:

Input Parameters	
`context_handle`	Security context being used
`message_buffer`	Message to be verified
`token_buffer`	Token containing a signature
Output Parameters	
`minor_status`	Kerberos 5 error code
`qop_state`	Quality of protection used

When the `gss_verify_mic( )` function returns `GSS_S_COMPLETE`, the signature was verified successfully.

The following excerpt from the sample programs illustrates receiving a message with this function call.

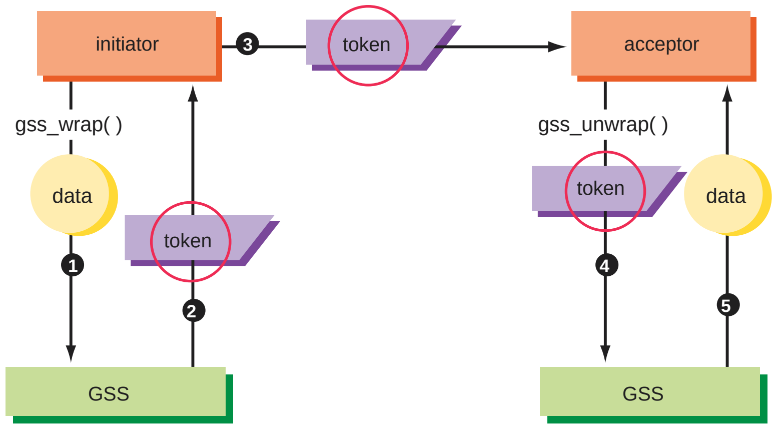
```
/****************************************************************/

char             szMessage[]    = "This is a test message";
OM_uint32        gMaj, gMin     = 0;
gss_buffer_desc  input_buffer   = GSS_C_EMPTY_BUFFER;
gss_buffer_desc  message_buffer = GSS_C_EMPTY_BUFFER;
gss_qop_t        qop_state      = 0;

/* Stuff plaintext message into the message buffer*/
message_buffer.value  = strdup( szMessage );
message_buffer.length = strlen( szMessage );

/* Receive the MIC token */
RecvToken( &input_buffer );

/* Verify the message against the message integrity code (MIC) */
gMaj = gss_verify_mic( &gMin,
                       context_handle,
                       &message_buffer,
                       &input_buffer,
                       &qop_state );

/****************************************************************/
```

C.7.2 Using gss_wrap( ) and gss_unwrap( )

Calling `gss_wrap( )` allows an application to wrap data that can be verified by the peer application using `gss_unwrap( )`. This ensures that the message has not been modified in transit (message integrity).

Additionally, the application can encrypt the data being sent for confidentiality, ensuring that no one can view the data. Encapsulating data into a token creates a secure envelope for data that can be sent over the network.

The sequence of steps for using these functions to send a message from an initiator to an acceptor follows:



ZK-1833U-AI

1. The initiator calls `gss_wrap( )` with data that it wants to wrap into a token. Confidentiality may be requested at the same time.
2. A token is returned containing the data and the signature of the data. If confidentiality was requested, both are encrypted.
3. The initiator sends the token to the acceptor.
4. The acceptor calls `gss_unwrap( )`, passing the token it received from the initiator.
5. The `gss_unwrap( )` function returns both a status code, verifying the signature, and the original data. The data is also decrypted if confidentiality was originally requested.

The `gss_wrap( )` function includes the following parameters:

Input Parameters	
`context_handle`	Security context being used
`conf_req_flag`	Confidentiality request (true or false)
`qop_req`	Quality of protection (QOP) requested
`input_message_buffer`	Message to be protected
Output Parameters	
`minor_status`	Kerberos 5 error code
`conf_state`	Confidentiality confirmation (true or false)
`conf_state`	Protected message

The following excerpt from the sample programs illustrates sending an encrypted message with this function call. The default QOP is specified which, in this case, is determined by the encryption type specified for the security context.

```
/**********************************************************/
char             szMessage[]   = "This is a test message";
OM_uint32        gMaj, gMin    = 0;
int              conf_req_flag = 1;
i_32             conf_state    = 0;
gss_buffer_desc  input_buffer  = GSS_C_EMPTY_BUFFER;
gss_buffer_desc  output_buffer = GSS_C_EMPTY_BUFFER;
gss_qop_t        qop_req       = GSS_C_QOP_DEFAULT;

/* Stuff plaintext message into the input buffer*/
input_buffer.value  = strdup( szMessage );
input_buffer.length = strlen( szMessage );

/* Wrap (Seal) the message into a token */
gMaj = gss_wrap( &gMin,
                 context_handle,
                 conf_req_flag,
                 qop_req,
                 &input_buffer,
                 &conf_state,
                 &output_buffer );

/* Send the message token */
SendToken( &output_buffer );
/**********************************************************/
```

Storage associated with the protected message should be freed after use using the `gss_release_buffer( )` function to avoid memory leaks.

The `gss_unwrap( )` function includes the following parameters:

Input Parameters	
`context_handle`	Security context being used
`input_message_buffer`	Message to be unwrapped, verified, and, optionally, decrypted
Output Parameters	
`minor_status`	Kerberos 5 error code
`output_message_buffer`	Unwrapped message
`conf_state`	Confidentiality confirmation (true or false)
`qop_state`	Quality of protection used

When the `gss_unwrap( )` function returns `GSS_S_COMPLETE`, the signature was verified successfully.

The following excerpt from the sample programs illustrates receiving a message with this function call.

```
/****************************************************************/

char             szMessage[256];
OM_uint32        gMaj, gMin    = 0;
i_32             conf_state    = 0;
gss_buffer_desc  input_buffer  = GSS_C_EMPTY_BUFFER;
gss_buffer_desc  output_buffer = GSS_C_EMPTY_BUFFER;
gss_qop_t        qop_state     = 0;

/* Receive the message token */
RecvToken( &input_buffer );

/* Unwrap (Unseal) the message token */
gMaj = gss_unwrap( &gMin,
                   context_handle,
                   &input_buffer,
                   &output_buffer,
                   &conf_state,
                   &qop_state );

/* Extract plaintext message from the output buffer*/
if ( output_buffer.value != NULL ) {
    strcpy( szMessage, output_buffer.value );
}

/****************************************************************/
```

Storage associated with the unwrapped message should be freed after use using the `gss_release_buffer( )` function to avoid memory leaks.

C.8 Step 5: Terminating the Security Context

When either application has completed communications with its peer, it may terminate the security context. The GSS-API standard allows a security context to be terminated by the initiating and the accepting application.

The process for terminating a security context is the same for both the initiator and the acceptor. If several security contexts coexist between an application and its peer, each context must be terminated separately.

The `gss_delete_sec_context( )` function includes the following parameters:

Input Parameters	
context_handle	Security context being deleted
Output Parameters	
minor_status	Kerberos 5 error code
output_token	Specify GSS_C_NO_BUFFER to request local deletion. This is recommended by HP.

The following excerpt from the sample programs illustrates terminating a security context and releasing credentials acquired earlier.

```
/***************************************************************/

/* The V2 GSS_API specs recommend setting the output token
   parameter to NULL to signify that no token is to be
   returned. */

gMaj = gss_delete_sec_context( &gMin,
                               &context_handle,
                               GSS_C_NO_BUFFER );

context_handle = GSS_C_NO_CONTEXT;
     :
     :
     :
..gMaj = gss_release_cred( &gMin,
                         &cred_handle );

cred_handle = GSS_C_NO_CREDENTIAL;

/***************************************************************/
```

C.9 Advanced Concepts

HP extensions to the GSS-API standard give applications better control of their security environment. These extensions allow you to:

- Obtain initial credentials for a user
- Set the required time synchronization between initiators, acceptors, and security servers
- Use DES3 encryption for improved confidentiality

These extensions are mechanism and implementation specific. If you use them, the portability of your application may be affected. For example, the HP implementation of DES3 will not interoperate with other GSS-API vendors offering DES3.

C.9.1 Obtaining Initial Credentials

When the standard GSS-API functions are used with Kerberos 5, the application must already have obtained a ticket-granting ticket (TGT) prior to running the GSS-API secured application. A user can obtain a TGT by running kinit or an equivalent application (for example, HP Single Sign-On).

In some situations it may be desirable to have the user execute only a single program that incudes the acquisition of the TGT. Therefore, Application Security SDK includes extension functions that can be used to obtain initial credentials from a security server.

The `csf_gss_acq_user( )` function is an HP extension that acquires initial credentials from within an application using secret key or public key authentication, optionally, with DES3 encryption.

When this function establishes a user context, it constructs a credentials cache for the user if it does not exist. If the credentials cache does exist but contains credentials that do not match those requested, the cache is re-initialized. In this case, the application must ensure that no other application is using the credentials cache.

This function also provides prompts and labels that the application must display. The application must collect the resulting responses and return them to GSS.

The `csf_gss_acq_user( )` function includes the following parameters:

Input Parameters	
`user_name`	Name of user or service needing initial credentials. If not specified, a default principal or login name can be used.
`desired_mechs`	Security mechanism desired (Kerberos 5)
`options`	Options that determine the protocol key, attributes (such as lifetime, renewable, forwardable, pre-authenticated and proxiable), and encryption method (DES or DES3) of the initial credentials.
`user_response`	Passes a user response to an application prompt
Output Parameters	
`minor_status`	Kerberos 5 error code
`user`	User context being acquired
`user_prompt`	Passes prompts to the application
`user_label`	Passes prompting information to the application
`prompt_state`	Passes prompting display hints to the application
`prompting_mech`	Security mechanism used (Kerberos 5)
`pwd_exp_time`	Expiration time of user's password

To use the `csf_gss_acq_user( )` function:

- Specify Kerberos 5 as the security mechanism.
- Select the desired options for the TGT.
- Respond to prompts. The function returns parameters that identify how the prompt should be processed and any user input needed.

This function is a looping function, that is, it may need to be called more than once:

- If it returns `GSS_S_COMPLETE`, the function has completed its task.

- If it returns `GSS_S_CONTINUE_NEEDED`, one or more prompts from Kerberos 5 must be satisfied.

Other special HP functions are available for managing initial credentials, for example, `csf_gss_inq_user( )` for querying the TGT. To release the storage associated with a user context after it is no longer needed, use `csf_gss_release_user( )`.

The following excerpt from the sample programs illustrates establishing a user context for initial credentials with the following options:

- Forwardable
- Lifetime of 10 hours
- Renew time of 48 hours
- DES3 encryption

It also checks for prompts of a password, challenge, and one-time password (OTP). The principal name was imported previously. Calls releasing storage no longer needed are not shown.

```
/*****************************************************************/

OM_uint32            gMaj           = 0;
OM_uint32            gMin           = 0;
i_32                 done           = 0;
i_32                 prompt_state   = CSF_GSS_C_USER_STATE_NULL;
int                  optCount       = 0;
gss_buffer_desc      user_prompt1   = GSS_C_EMPTY_BUFFER;
gss_buffer_desc      user_label1    = GSS_C_EMPTY_BUFFER;
gss_buffer_desc      response1      = GSS_C_EMPTY_BUFFER;
gss_buffer_t         user_prompt    = &user_prompt1;
gss_buffer_t         user_label     = &user_label1;
gss_buffer_t         response       = GSS_C_NO_BUFFER;
csf_gss_user_t       gss_user       = CSF_GSS_C_NO_USER;
gss_OID              prompting_mech = GSS_C_NO_OID;
csf_gss_mech_opt_desc opts[10];
char                 buff[256];
time_t               pwd_exp_time;

/* Set up the csf_gss_acq_user options */
opts[optCount].mechOID = rfc_krb5_c_OID;
opts[optCount].id  = CSF_GSS_C_ACQ_USER_OPT_FORWARDABLE;
++optCount;

opts[optCount].mechOID = rfc_krb5_c_OID;
opts[optCount].id  = CSF_GSS_C_ACQ_USER_OPT_LIFETIME;
opts[optCount].val = "10h";
++optCount;

opts[optCount].mechOID = rfc_krb5_c_OID;
opts[optCount].id  = CSF_GSS_C_ACQ_USER_OPT_RENEWABLE;
opts[optCount].val = "48h";
++optCount;

opts[optCount].mechOID = rfc_krb5_c_OID;
opts[optCount].id      = CSF_GSS_C_ACQ_USER_OPT_DES3;
++optCount;
```

```
opts[optCount].mechOID = GSS_C_NO_OID;

/* Acquire the user context */
do {
    gMaj = csf_gss_acq_user( &gMin,
                             principal_name,
                             rfc_krb5_c_OID_set,
                             opts,
                             response,
                             &gss_user,
                             user_prompt,
                             user_label,
                             &prompt_state,
                             &prompting_mech,
                             &pwd_exp_time );


    /* Check whether user needs to provide more information */
    if ( gMaj & GSS_S_CONTINUE_NEEDED ) {


        switch ( prompt_state )
        {
            case CSF_GSS_C_USER_STATE_PASSWORD_NOECHO:
            case CSF_GSS_C_USER_STATE_PASSWORD_ECHO:
            printf( "Password information is requested...\n");
            break;

            case CSF_GSS_C_USER_STATE_CHALLENGE_NOECHO:
            case CSF_GSS_C_USER_STATE_CHALLENGE_ECHO:
            printf( "Challenge information is requested...\n");
            break;

            case CSF_GSS_C_USER_STATE_OTP_NOECHO:
            case CSF_GSS_C_USER_STATE_OTP_ECHO:
            printf( "OTP information is requested...\n");
            break;

            default:
            printf( "Unrecognized prompt state.\n");
        }

        printf( "%s\n", user_label->value);
        printf( "Enter %s", user_prompt->value);
        fflush(stdout);

        response = &response1;

        fgets(buff, 256, stdin);
        response->value  = buff;
        response->length = strlen(buff);
        buff[response->length - 1] = ’\0’;
    }
    else {
        done = 1;
    }

} while (!done);

/*************************************************************/
```

C.9.2 Required time synchronization

The host for an accepting application must be synchronized within five minutes of the security server granting credentials. If the host for an initiating application is synchronized to within five minutes of the security server, it must also be synchronized within five minutes of the accepting application. If the clock skew is greater than five minutes, authentication will fail. When credentials are being forwarded from an application to its peer, the clocks on all systems must be synchronized to within five minutes.

As a consequence of the allowed five minute variation in host clocks, the security server will reject tickets before the actual expiration time if the remaining lifetime is less than the time skew limit. The actual valid lifetime in minutes required by the time skew limit depends on whether the initiator's clock is ahead or lags the security server's clock, but ranges from 1 to 5 minutes. As an example of a GSS failure related to this time skew limit, the function `gss_init_sec_context( )` will fail and set the context_handle parameter to `GSS_C_NO_CONTEXT` if the initiator's Kerberos credentials are within the time skew limit of expiration. If this occurs, the function returns the Kerberos error "Credentials Not Found" (hex value `C0001507`). In this case, the initiator application must acquire a new initial credential (TGT) from the security server before a context can be established with an acceptor application using `gss_init_sec_context( )`.

C.9.3 Using DES3 Encryption

Application Security SDK supports both DES and DES3 encryption. However, multiple encryption systems for a single security context are not allowed.

The following conditions must be met before DES3 encryption can used to encrypt messages:

- TrustBroker Security Server must be configured for DES3 when its database is created.
- The principals for the initiating and accepting applications must be DES3 enabled in the principal database. This means that if a user is running the initiating application, that user's principal must be DES3 enabled.
- The initiating application must obtain a TGT using DES3. The preceding excerpt contains an example using `csf_gss_acq_user( )`. The acceptor must have a key table entry for DES3.
- The initiating application must use the DES3 flag when initiating the security context.

It is not necessary to use HP extensions to enable DES3. However, you should always make sure the requested encryption is being used. A context can be downgraded from DES3 to DES if the above conditions are not met. After the context is established, check the flags returned with `csf_gss_get_context_options( )` to determine whether DES or DES3 was used.

C.10 Status Codes for GSS-API Functions

GSS-API functions have two return status codes: major status and minor status. It is important for an application to efficiently handle errors returned from the GSS-API using these status codes.

The *major status* is the GSS-API function error code. The major status is composed of three bit fields: the routine (function) error, the calling error, and supplementary information.

The *minor status* is a mechanism-specific error code. It may provide additional error information when `GSS_ERROR( )` indicates there was an error with the Kerberos 5 mechanism.

The presence of an error code is determined by sending a major status to the `GSS_ERROR( )` macro and by testing the result obtained. If zero is returned, there was no error; any other value indicates an error.

Warning

You must use the `GSS_ERROR( )` macro to test for the presence of an error; do not compare the status to the number zero.

The `GSS_ERROR( )` macro indicates errors by testing whether a code is present in the routine (function), calling, or supplementary information fields. The following code excerpt shows an example of this usage.

```
/***************************************************************/

OM_uint32       status_value = <--- value passed in to test
OM_uint32       majErr, minErr  = 0;
OM_uint32       message_context = 0;
gss_buffer_desc status_string   = GSS_C_EMPTY_BUFFER;

if ( GSS_ERROR(status_value) ||
     GSS_SUPPLEMENTARY_INFO(status_value) ) {

    /* First process the Major status code */
    do {
        /* Get the status string associated
           with the Major (GSS=API) status code */
        majErr = gss_display_status( &minErr,
                                     status_value,
                                     GSS_C_GSS_CODE,
                                     GSS_C_NO_OID,
                                     &message_context,
                                     &status_string );
        /* Print the status string */
        printf( "Major status string: %s\n",
        (char*)status_string.value );

        /* Free the status string buffer */
        gss_release_buffer( &minErr, &status_string );

    } while( message_context && !GSS_ERROR( majErr ) );

    /* Then process the Minor status code */
    do {
        /* Get the status string associated
           with the Minor (mechanism) status code */
        majErr = gss_display_status( &minErr,
                                     status_value,
                                     GSS_C_MECH_CODE,
                                     GSS_C_NO_OID,
                                     &message_context,
                                     &status_string );

        /* Print the status string */
        printf( "Minor status string: %s\n",
        (char*)status_string.value );

        /* Free the status string buffer */
        gss_release_buffer( &minErr, &status_string );

    } while( message_context && !GSS_ERROR( majErr ) );
}

/****************************************************************/
```

The supplemental information field provides additional function information. For example, if the `gss_init_sec_context( )` function returned a major status code indicating no error, the supplemental information field may indicate that `gss_init_sec_context( )` requires a response token from the other side of the application. In another example, if the function `gss_unwrap( )` returned a major status code indicating an error, the supplemental information field indicates whether the error is because the token is out of sequence, a replay, or old.

Supplemental information values are tested by ANDing the major status code with a named identifier. The example code fragment below shows the processing that occurs if `gss_unwrap( )` detects a replay.

```
/****************************************************************/

major_status = gss_unwrap(. . .);
if (GSS_ERROR(major_status)) {
   /* An error has occurred. */
   . . .
   if (major_status & GSS_S_DUPLICATE_TOKEN) {
     /* The token is a replay of a previous token. */
     printf ("Warning: Replay detected\n");
   }
}

/****************************************************************/
```

C.10.1 Minor Error Codes

The minor error codes for Kerberos mechanism are located in the `.hs` files in the `include/csf/sts` and `include/csfc5/sts` directories.

Any of these error codes may be referenced by your application, and your code may branch as needed, depending on the Kerberos error conditions that result. Applications need to refer to the macro that defines the value (the symbol after the `#define`).

Typically a minor error indicates a Kerberos configuration issue or a problem with a principal in the principal database.

C.11 Sample Programs

Application Security SDK includes a client/server sample application. The programs demonstrate the use of GSS-API functions. The sample program uses the following GSS-API functions:

- `gss_accept_sec_context( )`
- `gss_acquire_cred( )`
- `gss_delete_sec_context( )`
- `gss_display_name( )`
- `gss_display_status( )`
- `gss_export_sec_context( )`
- `gss_get_mic( )`
- `gss_import_name( )`
- `gss_import_sec_context( )`
- `gss_init_sec_context( )`
- `gss_inquire_context( )`
- `gss_inquire_names_for_mech( )`
- `gss_oid_to_str( )`
- `gss_release_buffer( )`
- `gss_release_cred( )`
- `gss_release_name( )`
- `gss_release_oid( )`
- `gss_release_oid_set( )`
- `gss_str_to_oid( )`
- `gss_unwrap( )`

- `gss_verify_mic( )`
- `gss_wrap( )`

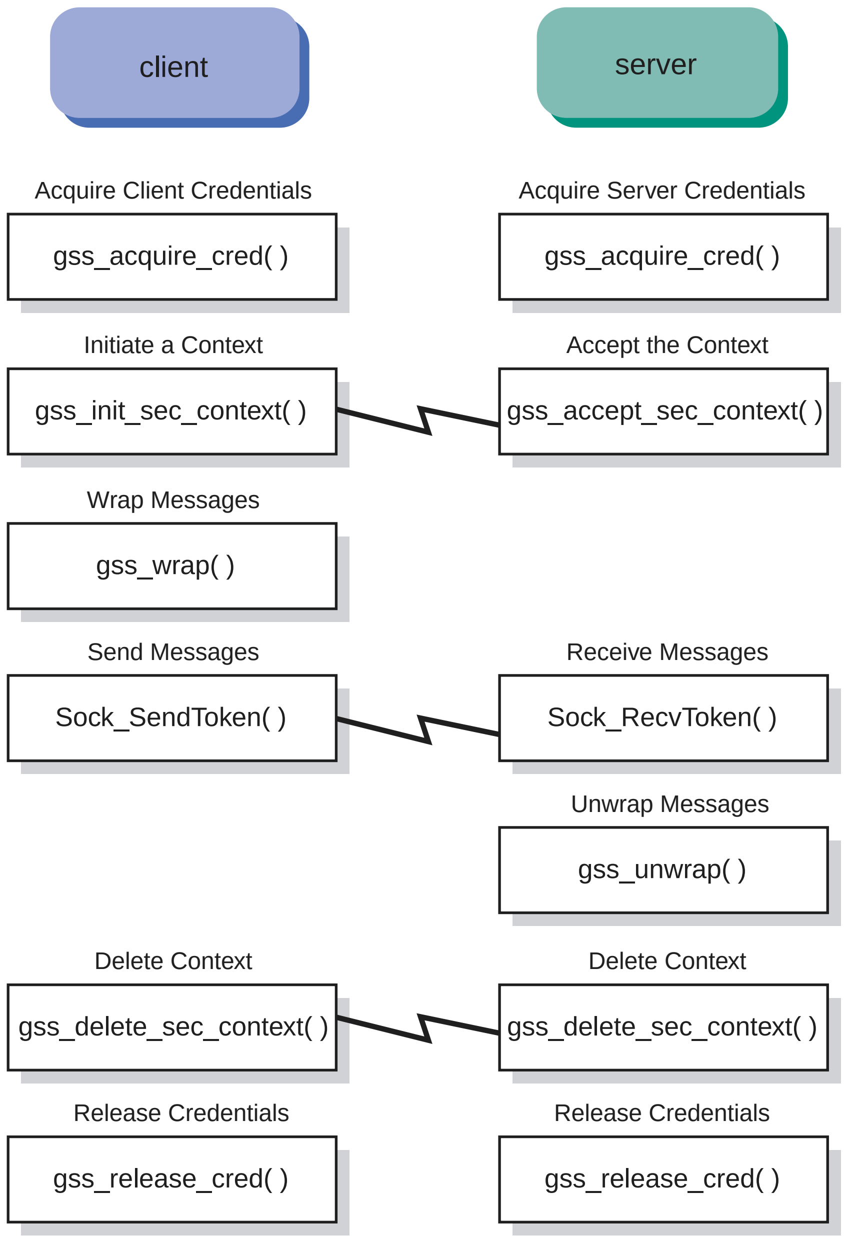
The client program initiates a security context with the server program. After a security context is established, the client and server exchange protected messages. The client and server programs can be distrubuted across any two systems in the same realm, or across two systems in different realms (which have been set up for interrealm authentication).

Each time the client is invoked, it performs one or more exchanges with the server. Each exchange with the server consists primarily of the following steps:

1. A TCP/IP connection is established.
2. (Optional, on by default:) The client and server establish a GSS-API context, and the server prints the identity of the client.
3. The client sends a message to the server. The message may be plaintext, cryptographically “signed” but not encrypted, or encrypted (default).
4. The server decrypts the message (if necessary), verifies its signature (if there is one) and prints it.
5. The server sends either a signature block (the default) or an empty token back to the client to acknowledge the message.
6. If the server sent a signature block, the client verifies it and prints a message indicating that it was verified.
7. The client sends an empty block to the server to tell it that the exchange is finished.
8. The client and server close the TCP/IP connection and destroy the GSS-API context.

Source Code	Description
`gss-server.c`	Main server source code
`gss-client.c`	Main client source code
`gss-misc.h`	Header file for miscellaneous tasks
`gss-misc.c`	Performs miscellaneous tasks
`Makefile_CPQ`	Build file
`README`	Information file

The following diagram shows how the GSS functions are implemented in the sample programs.


client
server
Acquire Client Credentials
gss_acquire_cred( )
Acquire Server Credentials
gss_acquire_cred( )
Initiate a Context
gss_init_sec_context( )
Accept the Context
gss_accept_sec_context( )
Wrap Messages
gss_wrap( )
Send Messages
Sock_SendToken( )
Receive Messages
Sock_RecvToken( )
Unwrap Messages
gss_unwrap( )
Delete Context
gss_delete_sec_context( )
Delete Context
gss_delete_sec_context( )
Release Credentials
gss_release_cred( )
Release Credentials
gss_release_cred( )

ZK-1834U-AI

C.11.1 Building the Sample Programs

HP recommends that you copy the sample code to a new location, so the original code is preserved in case you need to make changes to it. A makefile (`Makefile_CPQ`) is provided that needs little or no modification. Debugging is disabled by default.

To build the executables, use the make command as follows:

- To build the executables with the default settings, enter:

  ```
  # make -f Makefile_CPQ
  ```

- To enable debugging, enter:

  ```
  # make -f Makefile_CPQ OPTDEBUG=-g2
  ```

- To remove objects and executables, enter:

  ```
  # make -f Makefile_CPQ clean
  ```

C.11.2 Running the Sample Programs

This section describes:

- Prerequisites that must be met before the sample programs can run
- How to start the sample programs
- Sample server program commands
- Sample client program commands
- Typical sample program output
- Troubleshooting guidelines

C.11.2.1 Prerequisites

You must satisfy the following requirements before you can run the sample programs.

1. Define a principal for the client.

   The client program authenticates using the principal `host/host_name@REALM`, where `host_name` is the acceptor host's fully qualified domain name and `REALM` is the acceptor host's default realm. Therefore, the principal `host/host_name@REALM` must be registered with the security server (KDC).

2. Extract the server principal.

   The host principal for the server must be present in the service key table file (called `v5srvtab`), and the service key table file must be readable by the server program. If the service key table file is not present on the security server, it can be created by extracting the host principal.

3. Add a principal for the client to the principal database.

   To run the server on a host, you need to make sure that the principal corresponding to `service_name` is in the default keytab on the server host (`/krb5/v5srvtab`), and that the gss-server process can read the keytab. For example, the service name "`sample@server`" corresponds to the Kerberos principal "`sample/KDChost.domain.com@REALM`."

C.11.2.2 Starting the Sample Programs

Always run the sample server program first, then start the sample client program. Run each program in a separate shell so that you can see the sample output separately. The client and server may be run on the same machine or separate machines.

The client/server programs accept mandatory command line arguments and all switches are optional.

The server program requires the following arguments:

- `service_name`—The GSS-API service name of the form "*`service@host`*."

  For example:

  ```
  # gss-server sample@REALM
  ```

The client program requires the following arguments:

`host`	The host running the server.
`service_name`	The service name that the server will use to establish connections. If gss-server is running on a different machine from gss-client, and you do not specify the host name in the service name when running gss-client, you must specify the server's host name in the service name you specify to gss-client.
`msg`	The message. For more than one word you must enclose the message within double quotes; for example: `# gss-client unix1 sample@REALM "Hello Kerberos"`

C.11.2.3 Server Command Line Switches (Optional)

The server command line switches are defined below. All are optional.

`-port`	The TCP port on which to accept connections.
`-once`	Tells the server to exit after a single exchange, rather than persisting.
`-inetd`	Tells the server that it is running out of `inetd`, so it should interact with the client on `stdin` rather than binding to a network port. Implies `-once`.
`-export`	Tells the server to test the `gss_export_sec_context( )` function after establishing a context with a client.
`-logfile`	The file to which the server should append its output, rather than sending it to `stdout`.

C.11.2.4 Client Command Line Switches (Optional)

The client command line switches are defined below. All are optional.

`-port`	The TCP port to which to connect. Default is 4444.
`-mech`	The OID of the GSS-API mechanism to use.
`-d`	Tells the client to delegate credentials to the server. For the Kerberos GSS-API mechanism, this means that a forwardable TGT will be sent to the server, which will put it in its credential cache (you must have acquired your tickets with `kinit -f` for this to work).
`-f`	Tells the client that the `msg` argument is actually the name of a file whose contents should be used as the message.
`-q`	Tells the client to be quiet, i.e., to only print error messages.
`-ccount`	Specifies how many sessions the client should initiate with the server (the “connection count”).
`-mcount`	Specifies how many times the message should be sent to the server in each session (the “message count”).
`-na`	Tells the client not to do any authentication with the server. Implies `-nw`, `-nx` and `-nm`.
`-nw`	Tells the client not to “wrap” messages. Implies `-nx`.
`-nx`	Tells the client not to encrypt messages.
`-nm`	Tells the client not to ask the server to send back a cryptographic checksum (“MIC”).

The client/server sample usage message will be displayed if no arguments are specified or when the program is not used correctly:

```
# gss-server
Usage: gss-server [-port port] [-verbose] [-once]
```

```
       [-inetd] [-export] [-logfile file] [service_name]

# gss-client
Usage: gss-client [-port port] [-mech mechanism] [-d]
       [-f] [-q] [-ccount count] [-mcount count]
       [-na] [-nw] [-nx] [-nm] host service msg
```

C.11.3 Sample Program Output

This section contains output from the sample client and server programs, demonstrating the client passing a text file as the message and the server authenticating the client and printing out the message.

```
# gss-server -verbose -once sample@REALM
Server waiting:
Received token (size=600):
60 82 02 54 06 05 2b 05 01 05 02 01 00 6e 82 02
47 30 82 02 43 a0 03 02 01 05 a1 03 02 01 0e a2
07 03 05 00 20 00 00 00 a3 82 01 56 61 82 01 52
30 82 01 4e a0 03 02 01 05 a1 15 1b 13 43 59 42
45 52 4e 54 2e 5a 4b 33 2e 44 45 43 2e 43 4f 4d
a2 28 30 26 a0 03 02 01 03 a1 1f 30 1d 1b 06 73
61 6d 70 6c 65 1b 13 63 79 62 65 72 6e 74 2e 7a
6b 33 2e 64 65 63 2e 63 6f 6d a3 82 01 04 30 82
01 00 a0 03 02 01 05 a1 03 02 01 01 a2 81 f3 04
81 f0 8e 80 d6 1d b4 11 51 61 75 24 f4 e4 96 c5
db 1b ba 94 be c1 60 11 1a 24 c3 13 ae 22 50 90
e2 ba 18 7b 9e 8f 6a 42 ca fa 96 b8 cf f0 8c 3f
b8 ea 33 e9 21 24 ab ab e2 7e 4a 90 ee 75 93 11
99 4e 15 ad 2e 47 83 5b de 74 eb b7 92 ad 86 e6
27 fd c0 02 13 6d 56 38 3d 7e 80 dc ea a0 1c 79
37 34 6f fa 4b dc 79 ed 2b ee ef 93 37 ae 6c 1f
2c 83 62 c1 7a 0b 5a aa 10 47 e4 70 1d 31 9c 2c
24 ee 8e 69 4a e2 c2 cd 7e 52 d4 ff 19 d6 d4 77
e9 ee 78 19 e8 2d 5b 31 5c a0 89 28 f5 b2 e2 ef
bf 2a 12 b5 1b 88 e3 e8 a3 21 5b d5 93 a1 21 44
77 e8 e1 71 f3 2f b8 e1 06 20 fe 21 42 9a f5 bf
e3 7b 55 76 a2 54 34 05 43 14 0b aa 2e d2 da 91
31 02 8d 65 53 fe 4d 32 b6 b6 31 6f b2 7c d4 77
bb 87 0f 85 6d 61 8d 2c 21 e8 be 3d fb fd a7 72
bd 71 a4 81 d3 30 81 d0 a0 03 02 01 05 a1 03 02
01 00 a2 81 c3 04 81 c0 aa b1 ae d7 0a 9b 75 c9
ce ca b4 b1 1d a4 a4 4b bc 3a 73 4d b8 9e c0 fb
51 44 8a 67 8d ad 25 87 6e 66 ed bf d7 fb fc b1
6b 38 89 74 2b f8 eb 04 be 76 70 03 e3 2f db 7d
15 53 53 9e 8d e4 f1 a1 60 b6 01 33 42 90 60 a5
4c 75 dd af af 64 75 86 5a f8 25 57 c1 22 bc 12
b1 54 c9 c1 0b a6 27 c0 44 2a 84 48 a6 6a 00 62
f8 4d d7 27 9e 35 a5 74 2c 8f 04 c0 a8 c4 48 b6
d9 52 84 40 5b 19 fe 4d bf eb 07 d0 e0 f9 46 8c
```

```
72 5b bc df d0 0e 55 4a e2 e8 39 10 83 63 9f 02
cd 07 8b 00 f9 d6 46 77 29 6d 13 64 e4 3c b3 53
f7 46 12 9c 88 8b 5d e6 9e 05 f2 f7 6d 4e d8 15
dd eb 9b a6 37 28 77 9a
Sending accept_sec_context token (size=107):
60 69 06 05 2b 05 01 05 02 02 00 6f 5e 30 5c a0
03 02 01 05 a1 03 02 01 0f a2 50 30 4e a0 03 02
01 05 a1 03 02 01 00 a2 42 04 40 b1 61 8f 22 03
93 d2 d4 df 6f 6e 19 99 99 68 64 78 53 27 22 8a
c4 e8 42 b3 b5 23 3f 5d 0d 6a 6d 87 42 7b ca 8b
10 13 ff 34 66 23 cf 75 a0 24 e9 4a a7 d6 cd 9c
e2 a5 1b 98 55 02 ba 85 e8 3c b3
context flag: GSS_C_MUTUAL_FLAG
context flag: GSS_C_REPLAY_FLAG
context flag: GSS_C_CONF_FLAG
context flag: GSS_C_INTEG_FLAG
Accepted connection using mechanism OID { 1 2 840 113554 1 2 2 }.
Accepted connection: "user@REALM"
Message token (flags=228):
60 57 06 05 2b 05 01 05 02 02 01 dd fa de fa ff
ff e7 ad 91 9a ca e2 08 aa 32 27 86 fa 8a 0c 17
44 18 9c 8c 7b f4 65 8c 63 88 6f be 70 f0 a8 ef
95 17 da 92 48 44 e6 70 1c 4a 80 97 c0 f3 d3 34
39 1f 03 b3 55 df 50 75 f0 40 9d 8a 9b 5d 0f aa
3d 6b a0 c6 6d 34 42 29 58
Received message: "
Hello Kerberos.
I am a text file.
"
NOOP token
#
```

The following is the output from the client, using a test file called `test`:

```
# gss-client -f unix1 sample@REALM "test"
Sending init_sec_context token (size=600)...continue needed...
context flag: GSS_C_MUTUAL_FLAG
context flag: GSS_C_REPLAY_FLAG
context flag: GSS_C_CONF_FLAG
context flag: GSS_C_INTEG_FLAG
"user@REALM" to "sample/host.domain@REALM", lifetime -1,
flags 1b6, locally initiated, open
Name type of source name is { 1 2 840 113554 1 2 2 1 }.
Mechanism { 1 2 840 113554 1 2 2 } supports 7 names
  0: { 1 2 840 113554 1 2 1 1 }
  1: { 1 2 840 113554 1 2 1 2 }
  2: { 1 2 840 113554 1 2 1 3 }
  3: { 1 2 840 113554 1 2 1 4 }
  4: { 0 18 18 18 18 18 18 18 18 18 18 }
  5: { 1 2 840 113554 1 2 2 1 }
  6: { 1 2 840 113554 1 2 2 2 }
```

```
Signature verified.
#
```

C.11.4 Troubleshooting Guidelines

Most problems associated with running the sample programs are related to the setup and administration of the security server. However, if you have trouble getting the sample program to run correctly, there are several things to check:

- A non-root user typically runs the server program. However, by default, the service key table file is readable only by root.

  Make sure the server program can read the service key table file and that the file contains an entry for `sample/KDC_host_name@REALM`.
- The Kerberos library for Application Security SDK supports two environment variables that specify the location of the service key table file: `CSFC5KTNAME` and `KRB5KTNAME`. If both variables are set, the former variable has precedence. Make sure these variables properly specify where the service key table file is located, and ensure that you have a valid TGT (`klist`).
- The sample programs work across realms only if those realms are set up for interrealm authentication.

Remember that Application Security SDK overlays the Kerberos mechanism, so, if anything goes wrong, it's usually a failure in the underlying mechanism. The sample program is merely passing on the error. For example, if the wrong password is entered, Kerberos generates a message such as "Decrypt integrity check failed" and sends it to Application Security SDK. In this case, the sample program simply reports the error, cleans up, and exits.

Index