Trusted Solaris Developer’s Guide

Sun Microsystems, Inc.
901 San Antonio Road
Palo Alto, CA 94303-4900
U.S.A.

Part No: 816–1042–10
November 2001

Adobe PostScript™

010928@2471

Contents

Tables

Figures

Preface

The *Trusted Solaris Developer's Guide* describes how to use the programming interfaces to write new trusted applications for the Trusted Solaris™ operating environment. Readers should know UNIX® programming and understand security policy concepts.

Related Books

The Trusted Solaris documentation set is supplemental to the Solaris™ 8 4/01 documentation set. You should obtain a copy of both sets for a complete understanding of the Trusted Solaris environment.

In this book, system administration duties are referenced to give context for how to set up aspects of the environment in which third-party applications operate. The Trusted Solaris environment contains various administrative roles, and these references to system administrator duties are general and do not refer to a specific administrative role.

The *Trusted Solaris Administrator's document set* describes how system administration duties are divided among different roles. See the *Trusted Solaris Roadmap* for a description of the books in the documentation set.

How This Book is Organized

The first two chapters present an overview of the Trusted Solaris programming interfaces, how security policy is enforced, how to retrieve security attribute information for file systems and processes, and how to use the Trusted Solaris security mechanisms. An overview of security policy and interprocess communications is presented in Chapter 10.

Chapter 1 presents an overview of the Trusted Solaris application programming interfaces and how security policy is enforced in the system.

Chapter 2 contains short example programs showing how to retrieve security attribute information for file system and process objects, and how to use the security mechanisms provided in the Trusted Solaris environment.

Chapter 3 describes the data types and programming interfaces for managing file and process privileges. This chapter also describes how privileges are used in programs, presents guidelines for using privileges, and has a section of code examples.

Chapter 4 describes the data types and programming interfaces for managing labels on process, file system, and device objects. This chapter also describes how a process acquires a CMW label, when label operations require privilege, and presents guidelines for handling labels.

Chapter 5 presents example code showing how to use the programming interfaces.

Chapter 6 describes the data types and programming interfaces for managing the process clearance. This chapter also describes how a process acquires a clearance, which privileges bypass the restrictions placed on a process by the process clearance, and has a section of code examples.

Chapter 7 describes the data types and programming interfaces for getting information on multilevel and single-level directories. There chapter has a section of code examples.

Chapter 8 describes the data types and programming interfaces for generating audit records from a third-party application. There chapter also describes privilege and has a section of code examples.

Chapter 9 describes the data types and programming interfaces for reading the security information in the user databases. This chapter has a section of code examples.

Chapter 10 presents an overview of how security policy is applied to process-to-process communications within the same workstation and across the network.

Chapter 11 describes the data types and programming interfaces for managing labels on System V IPC™ objects. This chapter has a section of code examples.

Chapter 12 describes the data types and programming interfaces for handling security attribute information on messages transmitted across the network. This chapter has a section of code examples.

Chapter 13 describes data types and programming interfaces for remote procedure calls (RPC). This chapter has a section of code examples.

Chapter 14 describes the data types and programming interfaces that allow administrative applications to access and modify security-related X Window System information. This chapter has a section of code examples.

Chapter 15 describes the data types and programming interfaces for creating a graphical user interface for building labels and clearances. This chapter has a section of code examples.

Appendix A provides information on accessing man pages, shared libraries, header files, abbreviations used in data type and interface names, and preparing an application for release.

Appendix B provides listings of the programming interfaces including parameter and return value declarations.

Ordering Sun Documents

Fatbrain.com, the Internet's most comprehensive professional bookstore, stocks select product documentation from Sun Microsystems, Inc.

For a list of documents and how to order them, visit the Sun Documentation Center on Fatbrain.com at `http://www1.fatbrain.com/documentation/sun`.

Accessing Sun Documentation Online

The docs.sun.com^{SM} Web site enables you to access Sun technical documentation online. You can browse the docs.sun.com archive or search for a specific book title or subject. The URL is `http://docs.sun.com`.

Typographic Changes and Symbols

The following table describes the type changes and symbols used in this book.

TABLE P–1 Typographic Conventions

Typeface or Symbol	Meaning	Example
`AaBbCc123`	The names of commands, files, and directories; on-screen computer output	Edit your `.login` file. Use `ls -a` to list all files. `system% You have mail.`

`AaBbCc123`	What you type, contrasted with on-screen computer output	`system%` **`su - janez`**
		`Password::`

AaBbCc123	Command-line placeholder or variable name. Replace with a real name or value	To delete a file, type `rm` *filename*. The *errno* variable is set.
AaBbCc123	Book titles, new words or terms, or words to be emphasized	Read Chapter 6 in *User's Guide*. These are called *class* options. You *must* be root to do this.
Code samples are in `code font` and may display the following:		
`%`	C shell prompt	`system%`
`$`	Profile shell prompt	`system$`
`#`	root role prompt	`system#`

CHAPTER 1

Introduction to the API and Security Policy

The Trusted Solaris environment provides an application programming interface (API) for accessing and handling security-related information from within third-party applications. This chapter summarizes the API functionality and introduces you to the Trusted Solaris security policy.

- "Operating Environment Features" on page 23
- "Data Objects" on page 24
- "Application Programming Interfaces" on page 26
- "Security Policy" on page 33

Operating Environment Features

The Trusted Solaris environment is based on the Solaris operating environment, and provides enhanced security while maintaining the following Solaris operating environment features:

- ANSI C language specification.
- Application Programming Interface (API).
- SPARC™ Architecture Manual Version 8 application binary interfaces (ABIs), and System V Release 4 ABI.
- Executable file formats:
  - `a.out`.
  - Executable and linking format (ELF).
  - Interpreted files.
- Device programming interfaces:
  - Device Driver Interface (DDI).

    - Device Kernel Interface (DKI).
- File systems and file system objects.
- User and system administration commands.
- Common Desktop Environment (CDE) specification.
- The Trusted Solaris X window system is based on and generally compatible with the X11R5-based window system in the Solaris operating environment.
- Motif and OpenLook Interface Toolkit (OLIT).
- Level 1, 2, and 3 internationalization.
- Solaris pluggable authentication module (PAM) functionality.

  Note that the password generation algorithm can be replaced by installing a new shared object called `/usr/lib/security/pam_rw.so` which implements the replacement function. The file must be at `ADMIN_LOW`, with the permissions `rwxr-xr-x root sys`. The function must conform to the syntax and symantics described in the `randomword`(3TSOL) man page.

Data Objects

Applications use Solaris and Trusted Solaris APIs to work on data in the types of objects described here. The Trusted Solaris environment implements security policy by imposing constraints on security-related operations applications perform on these objects. "Security Policy" on page 33 describes Trusted Solaris security policy as it applies to applications.

File System Objects

File system objects reside in a file system where they can be read, written to, searched, and executed according to file system security policy. File system objects are the following:

- Directories.
- Regular data files.
- Executable files.
- Symbolic links.
- Mapped memory.
- Device objects – Device special (character and block) files for device drivers to printers, workstations, tape drives, and floppy drives.

X11 Windows Objects

X Window System objects handle data input and output through a special file system interface. Although the data in these special files is not accessed the way the data in file system objects is accessed, these files are protected by file system security policy, while the X Window Server and the X Window System objects are protected by X Window System security policy.

Process Objects

A process can access data in another process or in lightweight processes (independently scheduled threads of execution). All process to process communications is protected by either process, network, or interprocess communications (IPC) security policy. If the communication involves a special file, the file is protected by file system security policy.

IPC Objects

Interprocess communication (IPC) objects are the following.

- Unnamed pipes.
- Named pipes (FIFOs).
- Mapped Memory.
- System V IPC objects (message queues, semaphore, and shared memory).
- Pseudo-Terminal Devices (PTYs).
- Signals.
- Process Tracing.

Network Communication Endpoints

Network communication endpoints are sockets and transport layer interface (TLI) endpoints.

- INET Domain Sockets bind to a port.
- UNIX Domain Socket Rendezvous bind to a file.
- INET Domain TLI bind to a port.
- UNIX Domain TLI bind to a file.
- Remote Procedure Calls (RPC) bind to a port.

STREAMS Objects

STREAMS objects form the basis for networking software and are protected by network security policy. Security attribute information carried on STREAMS is accessed through the IPC and networking APIs described in detail in this guide. “Trusted Streams” on page 270 lists interfaces that let you access the security attribute information on a Stream directly; however, no conceptual information or code examples is currently provided for these interfaces.

Application Programming Interfaces

The Trusted Solaris API provides access to the following security features. These features are listed here, briefly introduced in this chapter, and covered in detail in the remaining chapters of this guide.

- Security mechanisms:
  - Privileges
  - User authorizations
  - CMW labels
  - Process clearances
  - Multilevel directories
  - Application auditing
- User and Rights Profile database security information
- System security configuration settings
- Security attribute information:
  - File system security attributes and flags
  - Process security attribute flags
  - Network security attributes
  - X11 Windows security attributes
- Process to object communications:
  - Secure interprocess communications with CMW labels
  - Secure file system communications with CMW labels and file system security attributes
  - Secure network communications with CMW labels, multilevel ports, multilevel mappings (RPC only), and network security attributes
  - Secure transfer of data between X11 Windows with CMW labels and windows security attributes

- Label builder – APIs that let you create a graphical user interface for your application that takes end user input and builds a valid label for the system

Privileges

Privileges let a process perform tasks that are normally prohibited by the system security policy. In the Solaris operating environment, processes with the effective User ID of 0 (superuser) can bypass the system security policy, and processes at any other user ID have limited powers. In the Trusted Solaris environment, there is no superuser. A process with any user ID can be assigned specific privileges to give it a defined set of security-related powers. See `priv_desc`(4) for a list of privileges and the tasks they allow a process to perform.

Most applications do not use privileges because they do not need security-related powers to run. An application using privileges is called a Trusted Computing Base (TCB) application and should be carefully coded to not make information available in inappropriate ways. "Security Policy" on page 33 provides guidelines to help you know when privileges might be needed, and Chapter 3 provides information and guidelines for coding privileged programs.

- Get and set the file and process privilege sets.
- Set the effective, permitted, and inheritable process privilege sets.
- Convert privilege IDs between numeric and text.
- Get privilege text for a privilege ID.

User Authorizations

The Trusted Solaris environment provides authorizations to control login, files and file management, devices, labels, and system administration activities. Applications can check a user's authorizations before performing certain tasks on behalf of that user if the tasks require user authorization. The tasks might be privileged administrative tasks or privileged non-administrative tasks. A good coding practice is to identify the authorization to be checked, identify the user or role performing the task, and check whether that user or role has the authorization to perform the task before turning privileges on in the application. If the task requires privilege (it usually does), authorizations should be checked before the process asserts the privilege.

Authorizations are administratively assigned and control user access to specific tasks. Authorizations are stored in `/etc/security/auth_attr` database. For a description of the file, see `auth_attr`(4). See `getauthattr`(3SECDB) for information on the family of routines for accessing and manipulating authorizations.

CMW Labels

CMW Labels control access to and maintain the classification of data. All processes and objects have a CMW label with two portions: the sensitivity label portion for mandatory access control (MAC) decisions, and the information label portion to identify the true sensitivity of the data.

Chapter 4 describes programming interfaces that do the following.

- Get and set file and process labels.
- Get file system label ranges.
- Initialize labels.
- Find the greatest lower bound or least upper bound between two levels.
- Compare levels for dominance and equality.
- Check and set binary label types.
- Convert labels between binary and text or hexadecimal.
- Check that a sensitivity label is valid and within the system or user accreditation range.
- Get information from the `label_encodings`(4) file. This file is set up and maintained by the system administrator and contains the label definitions for the system.

Process Clearance

When a user starts an application from a workspace, the user's session clearance is set on the process and called the process clearance. The process clearance sets the upper bound to which the process can change an object's CMW label and to which the process can write data. Chapter 6 describes programming interfaces that do the following:

- Get and set the process clearance.
- Initialize a binary clearance.
- Find the greatest lower bound or least upper bound between two levels.
- Compare levels for dominance and equality.
- Check and set binary label types.
- Convert clearances between binary and text or hexadecimal.
- Check that a clearance is valid.

Multilevel Directories

Multilevel directories (MLDs) enable a program that runs at different sensitivity labels to use a common directory and access files at the sensitivity label at which the process is currently running. An MLD contains only single-level directories (SLDs), and each SLD stores files at the sensitivity label of the SLD. Within one MLD, several files with the same name can be stored in different SLDs. Each instance of the same file contains data appropriate to the sensitivity label of the SLD where it is stored. This is called polyinstantiation of directories and files. Chapter 7 describes programming interfaces that do the following:

- Get single-level or multilevel directory names.
- Get attribute information for a single-level or multilevel directory.
- Using single-level or multilevel directory names in system calls.

Application Auditing

Third-party applications can generate audit records to monitor user actions to detect suspicious or abnormal patterns of system usage. Chapter 8 describes third-party application auditing.

User and Rights Profile Database Access

The user and profile databases contain information on users, roles, and profiles that can be accessed by an application. Chapter 9 describes programming interfaces that access this data.

Interprocess Communications

The Trusted Solaris environment supports labeled interprocess communications (IPC) with access and ownership checks. It supports the transfer of security attribute information for network endpoint objects.

Labeled endpoint communications can be single-level, multilevel, or polyinstantiated:

- Single-level port connection – Two unprivileged processes communicate at the same sensitivity label.
- Multilevel port connections – A privileged server communicates with any number of unprivileged clients running at different sensitivity labels.
- Polyinstantiated port connection (UNIX address family only) – A single-level connection using files of the same name residing in different single-level directories (SLDs) within a multilevel directory (MLD). Polyinstantiated port connections

create multiple independent parallel binds.

See the following chapters for information: Chapter 10, Chapter 11, Chapter 12, and Chapter 13.

Trusted X Window System

The Trusted X Window System, Version 11, server starts at login and handles the workstation windowing system using a trusted interprocess communication (IPC) path. Windows, properties, selections, and Tooltalk™ sessions are created at multiple sensitivity labels (polyinstantiated) as separate and distinct objects. Applications created with Motif widgets, Xt Intrinsics, Xlib, and CDE interfaces run within the security policy constraints enforced by extensions to the X11 protocols.

Appendix B describes the extensions for developers who need to create a X11 trusted IPC path. Chapter 14 describes programming interfaces to access security attribute information and translate binary labels and clearances to text by a specified width and font list for display in the X Window System.

Application User Interface

The Common Desktop Environment (CDE) 1.1.1 window system is the user interface for all interaction with the Trusted Solaris distributed operating system. User interfaces for new applications should use CDE APIs, Motif widgets 1.2, Xt Intrinsics, or XLib. The Trusted Solaris environment supports OpenWindows™ applications (based on the XView™ and Open Look Interface Toolkit (OLIT)) so trusted and untrusted applications that use OLIT for their user interface will run on the Trusted Solaris environment.

Label Builder

The Trusted Solaris environment provides Motif-based programming interfaces for adding a general label building user interface to an application. The label building interface lets a user interactively build valid CMW labels, sensitivity labels, or clearances. See Chapter 15 for information on the programming interfaces.

System Security Configuration Settings

The system administrator sets system variables in the `/etc/security` file to configure the system to handle certain security attributes at a site. Chapter 2 describes the programming interface for accessing Trusted Solaris system security variables that do the following:

- Enable privilege debugging for testing a privileged application. When privilege debugging is on, an application succeeds even when it does not have all the privileges it needs and the missing privileges are printed to the command line and to a file for your information. See *Trusted Solaris Administrator's Procedures* or "Privilege Debugging" on page 253 for information on enabling and using privilege debugging.
- Hide file names of files that have had their sensitivity labels upgraded by a privileged processes.

Security Attributes

Security attributes define security information for file systems, processes, data packets, communication endpoints, and X Window System objects.

File System Security Attributes and Flags

File systems store the Solaris and Trusted Solaris security attributes listed below as a security attribute set accessible by the programming interfaces described in Chapter 2. Chapter 3 describes how to access file privileges.

Solaris Attributes	Trusted Solaris Attributes
Access Control Lists (ACLs)	CMW label
DAC permission bits	File system label range
file user ID	Forced and allowed privilege sets
file group ID	Audit preselection attributes
	Attribute flags
	Multilevel directory prefix

Process Security Attributes and Flags

User processes receive the Solaris and Trusted Solaris security attributes listed below from the user or role that started them and the workspace where they were started.

- Chapter 2 describes how to access process attribute flags.
- Chapter 3 describes how to access process privilege sets.
- Chapter 4 describes how to access labels on processes.
- Chapter 6 describes how to access the process clearance.

Process ID	Process clearance
Real and effective user ID	CMW label
Real and effective group ID	Process attribute flags
Supplementary group list	Process privilege sets
User audit ID	
Audit session ID	
umask (defines permission bits for files created by the process)	

Endpoint Communications Security Attributes

The Trusted Security Information eXchange (TSIX) library provides access to the Trusted Solaris security attributes on data packets and communication endpoints. TSIX is based on Berkeley sockets and supports transport layer interface (TLI). Chapter 12 describes how to access security attributes on data packets and communication endpoints.

Effective user ID	Sensitivity label
Effective group ID	Audit information
Process ID	Process clearance
Network session ID	Effective privilege set
Supplementary group ID	Process attribute flags
Audit ID	

Trusted X Window System Security Attributes

The Trusted X Window System stores the security attributes listed below. Chapter 14 describes how to access X Window System security attributes.

Window Server owner ID	Sensitivity label
User ID	Internet address
Group ID	X Window Server clearance
Process ID	X Window Server minimum label
Session ID	Trusted Path window
Audit ID	

The Trusted Path flag means the window is a trusted path window. The trusted path window is always the top-most window (such as the screen stripe or log in window), and protects the system against access by untrusted programs.

Security Policy

The laws, rules, and practical guidelines by which the Trusted Solaris environment regulates how sensitive information is protected, managed, and distributed is called security policy. Trusted Solaris applications differ from Solaris applications in that they are subject to mandatory access control (MAC) and cannot run with all the powers of superuser. Solaris applications by contrast are subject to discretionary access control (DAC) only and can run with all the powers of superuser.

The Trusted Solaris environment provides privileges so processes can override mandatory read, write, and search restrictions; discretionary read, write, execute, and search restrictions; and perform special security-related tasks that would normally be reserved for superuser.

Discretionary Access Policy

The Trusted Solaris environment supports discretionary read, write, execute, and search permission using user, group, and other permission bits; and access control lists (ACLs). Controlling access with DAC and ACLs is part of the Solaris operating environment and not described in great detail in this guide, although retrieving ACLs as a file system security attribute is described in Chapter 2 and DAC policy is summarized in “Discretionary Access” on page 37

Mandatory Access Policy

The Trusted Solaris environment supports mandatory search, read, and write operations. MAC is enforced by comparing the sensitivity label and clearance of a process with the sensitivity label of the object to which the process is seeking access and determining whether the access is allowed or denied according to the MAC policy enforced on the object and the outcome of the comparison.

The outcome states the relationship between the process sensitivity label and object sensitivity label and is described as one dominating the other or equaling the other. The relationships of dominance and equality are covered in Chapter 4, and summarized here:

- Dominates – Has a higher or equal position in the classifications hierarchy, as defined in the `label_encodings`(4) file
- Equals – Has the same position in the hierarchy.

The outcome also states the relationship between the process clearance and the object sensitivity label as one of dominance or equality. If the access operation attempts to change the CMW label of the object, the clearance sets the highest level to which the sensitivity label portion can be changed. If the access operation is a write-up (see “Write Access” on page 35 below), the clearance sets the highest level to which the process may write.

The Trusted Solaris environment supports the following mandatory read and write operations on interactions between unprivileged processes and the objects they access. See “Policy Enforcement” on page 36 for information on how these operations apply to objects.

Read Access

The Trusted Solaris definition of mandatory read access includes read-equal and read-down:

- Read-Equal – An unprivileged process can read from an object only when the process sensitivity label is equal to the object sensitivity label.
- Read-Down – An unprivileged process can read from an object of a lower sensitivity label only when the process sensitivity label dominates the object sensitivity label and the labels are not equal.

Write Access

The Trusted Solaris definition of mandatory write access includes write-equal and write-up:

- Write-Equal – An unprivileged process can write to an object only when the process sensitivity label is equal to the object sensitivity label.
- Write-Up – An unprivileged process can write to an object of a higher sensitivity label only when the process sensitivity label is dominated by the object sensitivity label and the labels are not equal.

When to Use Privileges

To know if your application can run without privilege, you need to know what tasks use which privileges and when those privileges are needed. The following guidelines are to help you determine what privileges (if any) an application might need.

- Applications that perform no special tasks and operate within the mandatory access, discretionary access, and ownership controls of the system do not require privilege.
- Application tasks that require read, write, execute, or search access to an object require privilege when the process does not have discretionary or mandatory access. If a process does not have the access or the needed privilege, the external variable *errno* is set to `EACCES` or `ESRCH`. The privileges to correct the error are listed under the `EACCES` or `ESRCH` errors on the man page.
- Application tasks that modify an object in a way that only the owning process can modify it require privilege if the modifying process does not own the object. If a process does not own the object or have the proper privilege, the external variable *errno* is set to `EPERM`. The privileges to correct the error are listed in the Description section and under the `EPERM` error on the man page.
- Some application tasks always require privilege even when discretionary and mandatory access are allowed. Setting privileges on an executable file or redirecting console output to another device are two examples of such tasks. If a process does not have the privilege for such a task, the external variable *errno* is set to `EPERM`. The privileges to correct the error are listed in the Description section and under the `EPERM` error on the man page.

See Appendix A for information on how to access man pages to obtain information on privileges and privilege descriptions.

Administrative and User Applications

Administrative applications run at the administrative sensitivity labels of `ADMIN_HIGH` or `ADMIN_LOW`. At `ADMIN_HIGH`, the application can read down to any object to which it has discretionary access, and at `ADMIN_LOW`, the application can write up to any object to which it has discretionary access. An administrator will generally launch an application at `ADMIN_HIGH` to perform read-down operations, and launch the same application at `ADMIN_LOW` to perform write-up operations. In these cases, no privileges are needed as long as the application has discretionary access.

See "Initialize Binary Labels and Check Types" on page 100 in Chapter 5 for definitions of and information on initializing labels to `ADMIN_HIGH` and `ADMIN_LOW`.

Users generally launch an application at a given sensitivity label and access objects at that same sensitivity label. If the user keeps data at another sensitivity label, he or she will usually change the workspace sensitivity label and launch the application at the new sensitivity label. In this case, no privileges are needed as long as the application also has discretionary access.

If a user application is designed to access objects at sensitivity labels different from the sensitivity label at which the application is running, the application might need privilege to complete its tasks if mandatory access is denied.

See "Label Guidelines" on page 88 in Chapter 4 for guidance on the use of privileges to bypass mandatory access controls or to change a process or object sensitivity label.

Policy Enforcement

In UNIX all input and output is performed through a file interface, which means that file system security policy applies throughout the Trusted Solaris environment. For this reason, file system security policy is described in detail here.

File system security policy is stated in terms of the following:

- Mandatory and discretionary access checks between the process and the path name preceding the final object.
- Mandatory and discretionary access checks between the process and the final object.

Security policy for interprocess communications (IPC) is stated in terms of mandatory read and write access checks between the accessing process and the process being accessed. Some IPC mechanisms and X Window System objects use files, and file system security policy as described in this section applies to those operations. Some IPC mechanisms have the read-down and write-up security policy, while other IPC mechanisms have the more restrictive read-equal and write-equal policy. The X

Window system has the write-equal and read-down policy. See the following chapters for specific security policy information on these topics:

- Chapter 10 covers security policy for process-to-process communications on the same host and over the network.
- Chapter 14 covers security policy for accessing X11 windows property and resource data.

File System Security Policy

This section describes mandatory and discretionary access checks for the following file system objects:

- Directories – Regular directories and multilevel directories.
- Files – Regular files, executable files, device special files, and symbolic links.

Discretionary Access

The owner of the process must have discretionary search (execute) access to all directories in the path preceding the final object. Once the final object is reached, access operations can be performed as follows.

- Read from a file or list the contents of a directory – Discretionary read access is allowed when a process has discretionary search (execute) access to all directories in the object's path and discretionary read access to the object.
- Write to a file, create a file or directory, or delete a file or directory – Discretionary write access is allowed when the process has discretionary search (execute) access to all directories in the object's path and discretionary write access to the object.
- Execute a file – Discretionary execute access is allowed when the process has discretionary search (execute) access to all directories in the file's path and discretionary execute access to the file.

Mandatory Access

In addition to passing the DAC checks, mandatory search access is required to all directories in the path preceding the final file. Mandatory search access to a directory is allowed when the process sensitivity label dominates the sensitivity label of all directories in the path. Once the final file is reached, access operations can be performed as follows.

- Read from a file, execute a file, list the contents of a directory, view file security attributes, or view file security attribute flags – Mandatory read access is allowed when the process has mandatory search access to all directories in the path and the process sensitivity label dominates the sensitivity label of the final object. If the final object is a device special file, the process sensitivity label must equal the device sensitivity label.

- Write to a file, modify file security attributes, modify file security attribute flags, or delete a file – Mandatory write access is allowed when the process has discretionary and mandatory search access to all directories in the path and the file's sensitivity label dominates the process sensitivity label. If the final object is a device special file, the process sensitivity label must equal the device sensitivity label.
- Create a file or directory – Create access is write-equal. When a process creates a file, directory, or symbolic link the process sensitivity label must equal the sensitivity label of the file or directory.

File System Access Privileges

When a discretionary or mandatory access check fails on a file system object, the process can assert privilege to bypass security policy, or raise an error if the task should not be allowed at the current label or for that user.

Discretionary access is enabled as follows:

- Search access to all directories in the path preceding the final file system object is enabled when the process asserts the `file_dac_search` privilege.
- Read access to the final object is enabled when the process asserts the `file_dac_read` privilege.
- Write access to the final object is enabled when the process asserts the `file_dac_write` privilege.
- Execute access to the final object is enabled when the process asserts the `file_dac_execute` privilege.

Mandatory access is enabled as follows:

- Search access to all directories in the path preceding the final file system object is enabled when the process asserts the `file_mac_search` privilege.
- Read access (including execute access) to the final object is enabled when the process asserts the `file_mac_read` privilege.
- Write access to the final object is enabled when the process asserts the `file_mac_write` privilege.
- Create access to the final object is enabled when the process asserts the `file_mac_write` privilege.

When Access Checks are Performed

Mandatory and discretionary access checks are performed on the path name at the time a file system object is opened. No further access checks are performed when the file descriptor is used in other system calls, except as follows:

- A file is opened for writing and the descriptor is later used with the `fstat`(2) system call for a read. In this case, there are access checks for the read and privilege may be required if the access is denied.
- A file is opened for reading and the descriptor is later used with the `fchmod`(2) system call for a write. In this case, there are access checks for the write access and privilege may be required if the access is denied.

File System Policy Examples

The examples in this section illustrate the kinds of things you need to think about when a process accesses a file system object for read, write, search, and execute operations.

The process accesses `/export/home/heartyann/somefile` for reading and writing, and `/export/home/heartyann/filetoexec` for execution. These files are both protected at Confidential. The process sensitivity label is Secret and the process clearance is Top Secret. Confidential is lower than Secret and Secret is lower than Top Secret.

Sensitivity Labels

As shown in the following figure, the path `/export/home` has a sensitivity label of `ADMIN_LOW` and the `heartyann` directory and `somefile` have a sensitivity label of Confidential.

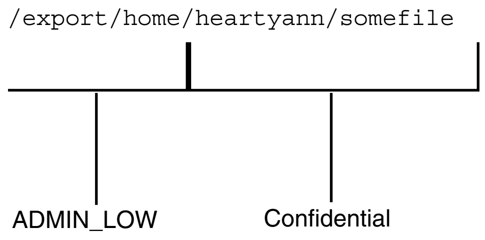

FIGURE 1–1 Accessing a File System Object

- The process does not own `somefile` or the directories in `somefile`'s path.
- Discretionary access permissions on `/export` allow the owner and group read, write, and search access; and allow others read and search access.

- Discretionary access permission on `/export/home` allow the owner read, write, and search access; and allow the group and others read and search access.
- Discretionary access permissions on `/export/home/heartyann` allow the owner and group read, write, and search access; and allow others read and search access.
- Discretionary access permissions on `somefile` allow the owner read and write access; and the group and others read access only.
- Discretionary access permissions on `filetoexec` allow the owner read, write, and execute access; and allow the group and others read and execute access.

If the process fails a mandatory or discretionary access check, the program needs to assert an error or the proper privilege if the program is intended to run with privilege.

See Chapter 4 in "Label Guidelines" on page 88 for information on handling sensitivity labels when privileges are used to bypass access controls.

Open the File

The Secret process opens `somefile` for reading, performs a read operation, and closes the file. The fully adorned pathname is used so `somefile` in the Confidential `/export/home/heartyann` single-level directory is accessed.

A fully adorned pathname uses the multilevel directory adornment and specifies precisely which single-level directory is wanted. If a regular pathname was used instead, the Secret single-level directory would be accessed because the process is running at Secret.

See "Adorned Names" on page 137 for a discussion on fully adorned pathnames. Chapter 7 presents interfaces for handling multilevel and single-level directories so fully adorned pathnames are not hardcoded the way they have been for clarity in these examples.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

main()
{
    int filedes, retval;
    ssize_t size;
    char readbuf[1024];
    char *buffer = "Write to File.";
    char *file = "/export/home/.MLD.heartyann/.SLD.1/filetoexec";
    char *argv[10] = {"filetoexec"};

    filedes = open("/export/home/.MLD.heartyann/.SLD.1/somefile", O_RDONLY);
    size = read(filedes, readbuf, 29);
    retval = close(filedes);
```

- Mandatory access checks on the open(2) system call – The process needs mandatory search access to /export/home/heartyann, and mandatory read access to somefile. The process running at Secret passes both mandatory access checks.
- Discretionary access checks on the open(2) system call – The process needs discretionary search access to /export/home/heartyann, and discretionary read access to somefile. The permission bits for other on the directory path and somefile allow the required discretionary search and read access.
- Mandatory access checks on the read(2) system call – The mandatory access checks were performed when somefile opened. No other access checks are performed.
- Discretionary access checks on the read(2) system call – The discretionary access checks were performed when somefile was opened. No other access checks are performed.

Write to the File

The Secret process opens somefile for writing in the Confidential /export/home/heartyann single-level directory, performs a write operation, and closes the file.

```
filedes = open("/export/home/.MLD.heartyann/.SLD.1/somefile", O_WRONLY);
size = write(filedes, buffer, 14);
retval = close(filedes);
```

- Mandatory access checks on the open(2) system call – The process needs mandatory search access to /export/home/heartyann, and mandatory write access to somefile. The process running at Secret passes the mandatory search access check, but does not pass the mandatory write access check. For mandatory write access, somefile's sensitivity label must dominate the process sensitivity label and it does not (Confidential does not dominate Secret). The process can assert the file_mac_write privilege to override this restriction or assert an error.
- Discretionary access checks on the open(2) system call – The process needs discretionary search access to /export/home/heartyann, and discretionary write access to somefile. The permission bits for other on the directory path and somefile allow the discretionary search access, but do not pass the discretionary write access check. The process can assert the file_dac_write privilege to override this restriction or assert an error.
- Mandatory access checks on the write(2) system call – The mandatory access checks were performed when somefile opened. No other access checks are performed.
- Discretionary access checks on the write(2) system call – The discretionary access checks were performed when somefile was opened. No other access checks are performed.

Execute a File

The Secret process executes an executable file in the Confidential `/export/home/heartyann` single-level directory.

```
retval = execv(file, argv);
```

- Mandatory access checks on the `execv`(2) system call – The process needs mandatory search access to `/export/home/heartyann`, and mandatory read access to `file`. Mandatory read access to a file is needed to execute the file. The process running at Secret passes both of these mandatory access checks.
- Discretionary access checks on the `execv`(2) system call – The process needs discretionary search access to `/export/home/heartyann`, and discretionary execute access to `file`. The permission bits on the directory path and on `file` allow discretionary search and execute access to `file`.

CHAPTER 2

Getting Started

This chapter contains short code examples to introduce you to some of the Trusted Solaris programming interfaces. The first section shows how to query the system security configuration, and how to query and set security attribute information for file systems and processes. The second section presents a short overview of Trusted Solaris security mechanisms. The major topics in this chapter are:

- "System Security Configuration and Attribute Information" on page 43
- "Trusted Solaris Security Mechanisms" on page 52
- "User and Rights Profile Databases" on page 58

System Security Configuration and Attribute Information

System security configuration variables provide system-wide information on the system configuration. Some applications query system variables before taking actions that might be affected by the status of the system's security configuration.

File system security attributes and flags provide security-related information for specified local and mounted file systems. Applications might need to know the status of file system security attributes and flags. For example, an application can query the file system default access control list (ACL) before performing a directory operation, or can find out if a directory is a multilevel directory before creating a new file in it.

Process security attribute flags provide information on the calling process. Applications might need to know the status of a process security attribute flag to, for example, know whether the process was started from an administrative role (trusted path flag set) or by a normal user (trusted path flag not set).

Programming Interfaces

The programming interfaces and code examples to check system security configuration and security attribute information are provided here. Descriptions of the data handled by these calls are in the appropriate chapter. For example, Chapter 4 covers labels and Chapter 3 covers privileges.

In cases where there is one set of interfaces to access a file using the pathname and another to access a file by the file descriptor, the examples that follow show the pathname only because the syntax is nearly identical.

All examples in this section compile with the `-ltsol` library.

System Security Configuration

This system call gets information on the system security configuration. Refer to the `secconf`(2) man page.

```
long secconf(int name);
```

File System Security Attributes

These system calls get information on file system security attributes using a path name or file descriptor. Refer to the `getfsattr`(2) man page.

```
int getfsattr(char *path, u_long type,
    void *buf_P, int len);
int fgetfsattr(int fd, u_long type, void *buf_P);
```

File System Security Attribute Flags

These system calls get information on file system security attribute flags using a path name or file descriptor. Refer to the `getfattrflag`(2) man page.

```
int fgetfattrflag(const char *path, secflgs_t *flags);
int setfattrflag(const char *path, secflgs_t which,
    secflgs_t flags);
int fsetfattrflag(int fildes, secflgs_t *flags);
int getfattrflag(int fildes, secflgs_t *flags);
int mldgetfattrflag(const char *path, secflgs_t *flags)
int mldsetfattrflag(const char * path, secflgs_t which,
    secflgs_t flags)
```

Process Security Attribute Flags

These system calls get and set process security attribute flags. Refer to the `getpattr`(2) man page.

```
int getpattr(pattr_type_t type, pattr_flag_t *value);
int setpattr(pattr_type_t type, pattr_flag_t value);
```

Query System Security Configuration

System variables provide information on how the system is configured. The system variables are initialized at system start up, and when there is no entry in `system`(4), default values are used. An application can query the system variables with the `secconf`(2) system call. The following variables are defined in `/etc/system` and have the default values listed:

`_TSOL_HIDE_UPGRADED_NAMES` – When a directory contains a file or subdirectory that has had its sensitivity label upgraded by a privileged process, this variable determines whether or not those upgraded files or subdirectories can be listed or obtained by system call requests such as `getdents`(2). Default is off. When off, names of upgraded files and subdirectories are visible when listing directories. When on, names of upgraded files or subdirectories are hidden.

`_TSOL_PRIVS_DEBUG` – Enable privilege debugging. Default is off. See *Trusted Solaris Administrator's Procedures* or "Privilege Debugging" on page 253 for information on how to enable and use privilege debugging.

This code queries the system variables to show their current values.

```
#include <tsol/secconf.h>

main()
{

long retval;


    retval = secconf(_TSOL_HIDE_UPGRADED_NAMES);
    printf("Hide Names = %d\n", retval);

    retval = secconf(_TSOL_PRIVS_DEBUG);
    printf("Priv Debug = %d\n", retval);
}
```

The `printf` statements print the following. A *retval* of 1 means the variable is on; 0 means off; and -1 means an error has occurred. *errno* is set only when the input variable is invalid.

```
Hide Names = 0
Priv Debug = 0
```

Query File System Security Attributes

File system security attributes fill in absent security attributes on local and mounted file system objects that were not assigned a full set of security attributes by the system administrator or did not acquire them from their creating process. You can get file system security attributes from the `vfstab`(4) and `vfstab_adjunct`(4) files, or from the file or directory inode.

Get Attributes from Adjunct File

The `vfstab_adjunct`(4) file contains remote mount points and their related security information. This file is set up and maintained by the system administrator so that file systems mounted to local workstations from remote workstations have the correct security attributes.

This example retrieves and displays lines from `vfstab_adjunct`(4). The `getvfsaent`(3TSOL) routine first reads the top line of the file and with each subsequent call reads the next lines one-by-one. The `getvfsaent`(3TSOL) routine reads the line for the mount point specified by the input file.

Note – Be sure to include `stdio.h` as shown in the example code below.

```
#include <stdio.h>
#include <tsol/vfstab_adjunct.h>

main()
{
    struct vfsaent *entry;
    char *vfsfile = "/etc/security/tsol/vfstab_adjunct";
    char *file = "/shark/doc";
    int retval;
    FILE *fp;

    fp = fopen(vfsfile, "r");
    if (fp == NULL) {
         printf("Can't open %s\n", vfsfile);
         exit(1);
    }

/* Step through file line-by-line. */
    retval = getvfsaent(fp, &entry);
    if (retval == 0) {
         printf("Mount Point is %s \n Security Info is %s\n",
         entry->vfsa_fsname, entry->vfsa_attr);
```

```
            free(entry);
        }
        else
            printf("No entries!\n");

        fseek(fp, 0, 0);

/* Retrieve specific mount point. */
        retval = getvfsafile(fp, &entry, file);
        if (retval == 0) {
            printf("Mount Point is %s \nSecurity Info is %s\n",
            entry->vfsa_fsname, entry->vfsa_attr);
            free(entry);
        }
        else
            printf("Mount point not found.\n");
        fclose(fp);
}
```

The `printf` statements print the following. There is only one entry in this `vfstab_adjunct` file for the `/opt/SUNWspro` mount point:

```
Mount Point is /opt/SUNWspro
Security Info is slabel=[C]:allowed all
Mount Point not found
```

Get Attributes from inode

The following code gets the CMW label `(FSA_LABEL)` of *file* and returns it in *buffer*.

```
#include <tsol/fsattr.h>
#include <tsol/label.h>

main()
{
        char *file = "/export";
        char buffer [3*1024], *string = (char *)0;
        int length, retval;

        length = sizeof(buffer);
        retval = getfsattr(file, FSA_LABEL, buffer, length);
        retval = bcltos((bclabel_t *)buffer, &string, 0, VIEW_INTERNAL);
        printf("/export CMW label = %s \n", buffer);
}
```

The `printf` statement prints the following:

```
/export CMW label = [ADMIN_LOW]
```

Manifest Constant Values

Manifest constant values can be any one of the following:

`FSA_ACLCNT` – File system access Access Control List (ACL) count.

`FSA_ACL` – File system access ACL.

`FSA_APRIV` – File system allowed privilege set.

`FSA_FPRIV` – File system forced privilege set.

`FSA_LABEL` – File system CMW label.

`FSA_AFLAGS` – File system attribute flags as described in "Get and Set File System Security Attribute Flags" on page 49.

`FSA_LBLRNG` – File system label range.

`FSA_MLDPFX` – File system MLD prefix string.

`FSA_APSACNT` – Number of classes in the process audit preselection mask.

`FSA_APSA` – Classes in the process audit preselection mask. The process needs the `file_audit` privilege in its effective set to get this information. See "Privileges and Authorizations" on page 52 for more information.

Manifest Constant Descriptions

The programming interfaces for accessing CMW labels, file system label ranges, file privileges, and multilevel directories are described briefly in "Trusted Solaris Security Mechanisms" on page 52 and in more detail in their respective chapters in this guide.

- ACLs – Because ACLs are part of the Solaris operating environment, they are not described in this guide.
- Audit preselection attributes – Audit preselection attributes are specified for a file system from the command line by the system administrator with `setfsattr`(1M). File system audit preselection attributes specify auditing on file permission bits. A file system can be configured so its files and directories are audited when access (read, write, or execute) succeeds or fails.

  Audit preselection attributes are specified for a process from the command line by the system administrator with `auditconfig`(1M). File system preselection attributes override the process preselection attributes. For example, a process that is audited for reads on files is not audited for reads on files that have file system preselection audit attributes that specify not to audit reads. See Trusted Solaris Audit Administration for more information.

Get and Set File System Security Attribute Flags

This example sets the public attribute flag on a regular directory and gets the MLD flag of a multi-level directory. The process needs the `file_owner` and `file_audit` privileges for this example to work. Use `setfpriv`(1) to set the privileges as follows. The `file_setpriv` privilege is required with `setfpriv`(1) so this command must be executed from the profile shell with this privilege.

```
phoenix% setfpriv -s -a file_owner,file_audit executable
```

```
#include <tsol/secflgs.h>
main()
{
    secflgs_t value;
    char *file = "/opt/SUNWspro"; /* Not MLD */
    char *file1 = "/export/home/zelda"; /* MLD */
    int retval;

    retval = setfattrflag(file, FAF_PUBLIC, FAF_PUBLIC);
    retval = getfattrflag(file, &value);
    printf("Public Attribute Flag = %d\n", value);

    retval = mldgetfattrflag(file1, &value);
    printf("MLD Attribute Flag = %d\n", value);
}
```

The `printf` statements print the following where 1 equals True and 0 equals false.

```
Public Attribute Flag = 0
MLD Attribute Flag = 1
```

`FAF_MLD` – Directory is a multi-level directory. `FAF_MLD` may be set without privilege if the directory is empty, the effective user ID of the process matches the directory owner, and the process has mandatory write access.

`FAF_SLD` – Directory is a single-level directory. This flag cannot be set programmatically.

If an adorned pathname is passed to `getfattrflag`(1), `FAF_MLD` is returned if the directory is an MLD. If an unadorned pathname is passed and if the directory is an MLD, `FAF_SLD` is returned.

If an adorned pathname is passed to `mldgetfattrflag`(2), `FAF_SLD` is returned if the directory is an MLD. If an unadorned pathname is passed and if the directory is an MLD, `FAF_MLD` is returned.

Adorned names are described in Chapter 7.

`FAF_PUBLIC` – File or directory is public. Audit records are not generated for read operations on public files and directories even when the read operations are part of a preselected audit class. This applies to the following read operations: access(2), fstatvfs(2), lstat(2), open(2) (read only), pathconf(2), readlink(2), stat(2), and statvfs(2).

Note – If the AUE_MAC or AUE_UPRIV audit pseudo events are in a preselected audit class, an audit record for those events is always generated regardless of the public attribute flag setting. See *Trusted Solaris Audit Administration* for more information on these pseudo audit events.

The process needs the file_audit and file_owner privileges in its effective set to get or set the public attribute flag for a file or directory. See “Privileges and Authorizations” on page 52 for more information. This flag can also be administratively set as described in *Trusted Solaris Administrator’s Procedures*.

FAF_ALL – The directory is a public MLD.

Get and Set Process Security Attribute Flags

Use getpattr(2) to query the attribute flags of the calling process.

```
#include <tsol/pattr.h>

main()
{
    int retval;
    pattr_flag_t value;

    retval = getpattr(PAF_TRUSTED_PATH, &value);
    printf("Trusted Path Value = %d\n", value);

    retval = getpattr(PAF_PRIV_DBG, &value);
    printf("Priv Debug value = %d\n", value);

    retval = getpattr(PAF_TOKMAPPER, &value);
    printf("Trusted Network Value = %d\n", value);

    retval = getpattr(PAF_DISKLESS_BOOT, &value);
    printf("Diskless Boot Value = %d\n", value);

    retval = getpattr(PAF_SELAGNT, &value);
    printf("Bypass Selection Agent Value = %d\n", value);

    retval = getpattr(PAF_PRINT_SYSTEM, &value);
    printf("Print System Value = %d\n", value);

    retval = getpattr(PAF_LABEL_VIEW, &value);
    printf("Label View Value = %d\n", value);
    retval = getpattr(PAF_LABEL_XLATE, &value);
    printf("Label Translate Value = %x\n", value);

    retval = getpattr(PAF_AUTOMOUNT, &value);
    printf("Automounter Value = %x\n", value);
}
```

The `printf` statements print the following where a *value* of 0 means the flag is off, and a value of 1 means it is on. The label translation value is 0 when *off* and a hexadecimal value representing the label translation flags when *on*. See "Manifest Constant Values" on page 51 for a description of the process attribute flags.

```
Trusted Path Value = 0
Priv Debug Value = 0
Trusted Network Value = 0
Diskless Boot value = 0
Bypass Selection Agent Value = 0
Print System Value = 0
Label View Value = 1
Label Translate Value = 1
Automounter Value = 1
```

Manifest Constant Values

`PAF_TRUSTED_PATH:` The trusted path flag is set for all administrative roles. Any process started from an administrative role has this flag set to 1. All other processes have this flag set to 0. This flag can be queried and cleared, but not set.

`PAF_PRIV_DEBUG:` The privilege debug flag is set to 1 when the process is started in privilege debugging mode. This flag can be queried by any process, but set only by a trusted path process. Enabling and using privilege debugging mode is described in *Trusted Solaris Administrator's Procedures* and "Privilege Debugging" on page 253 in Appendix A.

`PAF_NO_TOKMAP:` The trusted computing base network flag is set to 1 only on trusted computing base applications that send packets without security attributes to workstations that expect packets with security attributes.

`PAF_DISKLESS_BOOT`: The diskless boot flag supports diskless boot servers. When this flag is set to 1, the security attribute information in network packet headers is not sent.

`PAF_SELAGNT`: The selection agent flag when set to 1 permits a process to bypass the Selection Manager when moving data from one window to another. See "Moving Data Between Windows" on page 221 for more information.

`PAF_PRINT_SYSTEM`: The print system flag when set to 1 identifies a client process as a member of the printing subsystem.

`PAF_LABEL_VIEW`: When a user or role starts a process, this flag is set according to the label view specification in the `label_encodings` file or user label view setting in the `/etc/security/tsol/tsoluser` file. The label view applies to how the `ADMIN_HIGH` and `ADMIN_LOW` administrative labels are viewed in the system by users. The setting in the `tsoluser` file (if one exists) takes precedence over the setting in the `label_encodings` file.

A value of zero indicates the external view is in use and a value of 1 indicates the internal view is in use. Regardless of the value of this flag, a text to binary label translation can request the text string output for an administrative label to use the internal or external name. See Chapter 5 in "Binary to Text Label Translation Routines" on page 114 for details.

- Internal view – Show `ADMIN_HIGH` and `ADMIN_LOW`.
- External view – Set `ADMIN_LOW` to the next lowest label and `ADMIN_HIGH` to the next highest label as defined in `label_encodings`(4) .

`PAF_LABEL_XLATE`: The label translation flag when set to 1 indicates the `flags= keyword` option is in use in the `label_encodings`(4) file. This optional flag setting specifies which of 15 flags are associated with the word using this optional flag. Flags are not used by the system, but can be used by applications specifically written to use them to do such things as define certain words that appear only in printer banner labels (not in normal labels), or to define certain words that appear only in labels embedded in formal message traffic. This flag can be queried and set by a trusted path process only.

Trusted Solaris Security Mechanisms

This section provides short examples of the Trusted Solaris security mechanisms to give you an idea of how they are used. Every example in this section has a corresponding chapter, and the interface declarations can be found in the chapters. All examples compile with the `-ltsol` library, and in some cases, other libraries are also needed as noted with the example.

Privileges and Authorizations

Privileges let a process perform security-related tasks normally prohibited by the system security policy. Authorizations let a user perform privileged tasks not allowed to all users. Every authorization maps to a privileged task. Always check a user's authorizations before allowing a privileged task to take place.

Caution – The development, testing, and debugging of privileged applications should always be on an isolated development machine to prevent bugs and incomplete code from compromising security policy on the main system.

Privileges distribute security-related powers so a process has enough power to perform a task and no more. Likewise, authorizations distribute security-related powers so each user or role has enough power to perform a task and no more.

The system administrator assigns authorizations to users and roles through an execution profile. The `chkauth`(3TSOL) routine accepts a valid user name and authorization as parameters and returns true if the authorization is assigned to that user. During development, privileges can be assigned to the executable file and/or inherited from the user's or role's executable profile at run time.

To know if a program performs tasks that require privilege and user authorization checks, ask these questions:

- Does the task require privilege?
  - Information on privileges for system calls is on the `Intro`(2) man page and the man page for the particular system call.
  - Information on privileges for library routines is on the man page for the library routine or the man page for the underlying system call if there is an underlying system call. Check the See Also section of the library routine man page for a list of system calls where you can find privilege information if there is no information on the library routine man page.
  - The `priv_desc`(4) man page provides a list of Trusted Solaris privileges and a description of the tasks they enable.
  - Refer to Chapter 3 for information to help you decide if the privileges should be assigned to the file, inherited, or both.
  - Use privilege debugging mode as described in *Trusted Solaris Administrator's Procedures* or "Privilege Debugging" on page 253 in Appendix A" to find out what privileges an application needs.
- Does the task have an authorization? – The Trusted Solaris authorizations and their descriptions are in `/etc/security/auth_attr`.

This example checks the process permitted set for the `file_downgrade_sl` privilege, and the user authorization `solaris.label.file.downgrade` for user ID zelda before performing a task that involves downgrading the sensitivity label on a file. If the privilege is in the permitted set and if zelda has the authorization, the code turns the `file_downgrade_sl` privilege on in the effective set (makes the privilege effective) and performs the task. When the task completes, `file_downgrade_sl` is turned off (is no longer effective).

The example compiles with the following libraries:

```
-lsecdb -lnsl -lcmd -ltsol
```

Note – The permitted set contains the privileges the process can potentially use during execution, and the effective set contains the privileges the process is actually using at a given time. Turning effective privileges on and off is called privilege bracketing and is discussed in Chapter 3.

```
#include <tsol/priv.h>
#include <tsol/auth.h>

main()
{
    char *zelda = “zelda”;
    priv_set_t priv_set;

/* Retrieve the permitted privilege set */
    getppriv(PRIV_PERMITTED, &priv_set);

    if(PRIV_ISASSERT(&priv_set, PRIV_FILE_DOWNGRADE_SL) &&
        chkauthattr(solaris.label.file.downgrade, zelda)) {
        set_effective_priv(PRIV_ON, 1, PRIV_FILE_DOWNGRADE_SL);
        /* Downgrade sensitivity label on file*/
        set_effective_priv(PRIV_OFF, 1, PRIV_FILE_DOWNGRADE_SL);
    }
    else {/* Raise Errors */}
}
```

CMW Labels and Clearances

When a process writes to a file with a higher sensitivity label or changes the CMW label of an object, the system checks that the file sensitivity label dominates the process sensitivity label and the process clearance dominates the file sensitivity label. If your application writes to files at different sensitivity labels, you might want to perform these checks in the code to catch errors or to turn privileges on in the effective set as needed.

This code performs the following tasks:

- Retrieves the binary file CMW label, process CMW label, and process clearance.
- Retrieves the sensitivity label portion of the file CMW label and process CMW label.
- Checks for dominance by comparing the process sensitivity label to the file sensitivity label, and the process clearance to the file sensitivity label.

  If the comparisons return 0 (process sensitivity label and clearance do not dominate the file sensitivity label), the operation to change the file CMW label or write up to the file requires privilege. See “Privileges and Authorizations” on page 52 for information on privileges.

Chapter 4 and Chapter 6 describe the programming interfaces for translating a binary label or clearance to text so they can be handled like a string.

```
#include <tsol/label.h>
main()
{
    int retval, retvalclearance, retvalsens;
    bclabel_t filecmwlabel, processcmwlabel;
    bslabel_t filesenslabel, processsenslabel;
    bclear_t processclearance;
    char *file = "/export/home/labelfile";

/* Get CMW label of file */
    retval = getcmwlabel(file, &filecmwlabel);

/* Get Process CMW label */
    retval = getcmwplabel(&processcmwlabel);

/* Get sensitivity label portion of CMW labels */
    getcsl(&filesenslabel, &filecmwlabel);
    getcsl(&processsenslabel, &processcmwlabel);

/* Get process clearance */
    retval = getclearance(&processclearance);

/* See if process label dominates file label (retvalclearance > 0) */
    retvalclearance = bldominates(&processsenslabel, &filesenslabel);

/* See if process clearance dominates file label (retvalsens > 0) */
    retvalsens = bldominates(&processclearance, &filesenslabel);

/* Test results */
    if(retvalclearance && retvalsens > 0)
        { /* Change file CMW label or write-up to file */}
    else if (retvalclearance == 0)
        { /* Turn on error message or make appropriate privilege
             effective */}
    else if (retvalsens == 0)
        { /* Turn on error message or make appropriate privilege
             effective*/}
}
```

Multilevel Directories

Multilevel directories (MLDs) enable an application to run at different sensitivity labels and access data in the single-level directory (SLD) at the sensitivity label at which its process was launched. This example shows how to get the name for the Confidential SLD in the `zelda` `MLD` by translating a text string to binary with `stobsl`(3TSOL) and passing the binary label to `getsldname`(1). The `/export/home/zelda` MLD is at `ADMIN_LOW` and the process is running at

Confidential. The process needs no privileges because it has mandatory read access to the MLD and the process sensitivity label dominates the SLD sensitivity label.

```
#include <tsol/mld.h>

char *file = "/export/home/zelda";
char buffer[3*1024];
bslabel_t senslabel;
int length, flags, retval, error;

main()
{
/* Get the Confidential SLD name */
    retval = stobsl("CONFIDENTIAL", &senslabel, NEW_LABEL, &error);
    length = sizeof(buffer);
    retval = getsldname(file, &senslabel, buffer, length);
    printf("SLD Name = %s\n", buffer);
}
```

The `printf` statement prints the name of the SLD at `ADMIN_LOW`. See Chapter 7 for the meaning of the SLD name.

```
SLD Name = .SLD.2
```

Note – You can get file attribute information for an MLD or symbolic link that is an MLD with the `mldstat`(3TSOL) and `mldlstat`(3TSOL) system calls. See also the `stat`(2) man page and Chapter 7.

Application Auditing

An application can log its own third-party audit events with the `auditwrite`(3TSOL) library routine. This example creates a user audit record in one call to `auditwrite`(). The audit event logged is `AUE_su` with the text "successful login at console". Normally, `auditwrite`() logs application-level audit events. This example logs a Trusted Solaris user event to show how the routine is used. Chapter 8 shows third-party audit events.

The process executing this program needs the `proc_audit_tcb` privilege in its effective set because `AUE_su` is a Trusted Computing Base (TCB) audit event. The code comments indicate where privilege bracketing as described in Chapter 3 should take place. The `aw_strerror`(3TSOL) routine converts `auditwrite` error messages (`aw_errno`) to strings. The parameters passed to `auditwrite`() are as follows:

- `AW_EVENT` specifies the audit event to be written to the audit log. `AW_EVENT` is a user event string name as defined in `audit_event`. There can be only one event written to a single audit record.
- `AW_TEXT` is a null-terminated string placed in the audit record to provide additional information on the audit event.

- AW_WRITE writes the event and its associated text to the audit trail.
- AW_END tells auditwrite() to stop parsing information.

```
#include <bsm/auditwrite.h>
#include <types.h>
#include <unistd.h>

main()
{
    char *aw_string;
    int retval, errno;

/* Turn proc_audit_tcb on in the effective set */
    retval = auditwrite( AW_EVENT, "AUE_su", AW_TEXT,
        "Successful login at console", AW_WRITE, AW_END);
/* Turn the proc_audit_tcb privilege off */

    aw_string = aw_strerror(aw_errno);
    printf("Retval = %d AW_ERROR = %s ERRNO = %d\n", retval,
        aw_string, errno);
}
```

To run the program and view the audit record, do the following:

1. **Assume an administrative role, open a terminal at** ADMIN_HIGH**, and execute the following command where** lo **is the class to which** AUE_su **belongs and** pid **is the process ID of the terminal.**

   ```
   #auditconfig -setpmask pid lo
   ```

2. **Assume an administrative role, open a second viewing terminal at** ADMIN_HIGH**, and use** praudit**(1M) to read the** not_terminated **(most recent and not yet closed) audit log file by typing the command and options shown:**

   Note – This syntax works when there is only one *not_terminated* file. If there are others, delete the older ones before executing this command.

   ```
   phoenix% tail -0f *not_terminated* | praudit
   ```

3. **Compile and run the code from the first terminal window.**

   These libraries are needed for the example to successfully compile. -lbsm -lnsl -lintl -lsocket -ltsol

   The process needs the proc_audit_tcb privilege for this example to work. Use setfpriv(1) to set the privileges as follows. The file_setpriv privilege is required with setfpriv(1)so this command must be executed from the profile shell with this privilege. phoenix% setfpriv -s -a proc_audit_tcb executable

   The printf statement prints the following in the first terminal window:

   ```
   Retval = 0, AW_ERROR = No error, ERRNO = 0
   ```

   The viewing window shows the following audit record:

```
header, 129,2,su,,Wed Jun 26 14:50:19 1996, +698 msec
text, Successful login at console
subject,zelda,zelda,staff,zelda,staff,1050,853,24,7 phoenix
slabel,Confidential
return,success,0
```

The audit record consists of a sequence of tokens. Each line starts with a token followed by the token value. In the example, the tokens for audit event `AUE_su` are header, text, subject, slabel, and return; and the token values are the information following the tokens until the next token is encountered. Trusted Solaris Audit Administration describes the tokens in detail.

User and Rights Profile Databases

The information in the `user_attr`(4), `prof_attr`(4), and `exec_attr`(4) databases is accessible through library routines (see `getuserattr`(3SECDB), `getprofattr`(3SECDB), and `getexecattr`(3SECDB). User information is put into the databases by the system administrator through the Users Tool set in the Solaris Management Console.

CHAPTER 3

Privileges

Privileges organize security-related powers into discrete pieces where each piece (or privilege) maps to a single security-related task. Privileges enable a program to perform specific tasks normally prohibited by the system security policy. Prohibited tasks are such things as accessing a file or directory to which the program does not have the appropriate mandatory or discretionary access.

A program turns on (makes effective) one or more privileges to perform one security-related task. For example, if the program does not have mandatory write access to a file, it turns on the `file_mac_write` privilege. If the program does not have discretionary write access either, it also turns on the `file_dac_write` privilege. However, if the program has both mandatory and discretionary write access, it needs no privileges. Most programs do not use privileges because they operate within the bounds of the system security policy.

This chapter describes the programming interfaces for handling privileges.

Types of Privileges

The Trusted Solaris environment allows up to 128 different privileges. The total includes the following types of Trusted Solaris privileges and site-defined privileges. See `priv_desc`(4) for a description of the Trusted Solaris privileges.

- File system privileges override file system restrictions on user and group IDs, access permissions, labeling, ownership, and file privilege sets.
- System V Interprocess Communication (IPC) privileges override restrictions on message queues, semaphore sets, or shared memory regions.
- Network privileges override restrictions on reserved port binding, multilevel port binding, sending broadcast messages, or specifying security attributes on messages or communication endpoints.
- Process privileges override restrictions on process auditing, labeling, covert channel delays, ownership, clearance, user IDs, or group IDs.
- System privileges override restrictions on system auditing, workstation booting, workstation configuration management, console output redirection, device management, file systems, creating hard links to directories, increasing message queue size, increasing processes, workstation network configuration, third-party loadable modules, or label translation.
- X Window System privileges override restrictions on reading to and writing from windows, input devices, labeling, font paths, moving data between windows, X server resource management, or direct graphics access (DGA).

Privilege Sets

Privileges are organized into file privilege sets and process privilege sets.

File Privilege Sets

Executable files, interpreted files, and CDE actions have file privilege sets assigned through the File Manager, with `setfpriv`(1), or by another privileged program. The file privilege sets are the forced set and the allowed set.

Allowed Set

The allowed set contains the privileges that will be assigned to the executable file (forced file set) or inherited and used by the executing process. When a process inherits a privilege from another process, it cannot use that privilege unless the privilege is in the allowed set of its executable file.

Allowed privileges provide Trojan horse protection because they protect against an untrusted process entering the system and inheriting privileges from another process. See "Inheritable Set" on page 61 for more information on inheriting privileges.

Forced Set

The forced set contains the privileges a program must have when it begins execution for security-related tasks performed by any user. Commands with forced privileges can be invoked from any shell, and CDE actions with forced privileges can be invoked from any workspace. The forced set must always be equal to or a subset of the allowed set, and so, every privilege in the forced set is also in the allowed set.

Interpreted Files

Interpreted files are scripts that begin with `#!` and go through an interpreter to be executed. The script file can have forced and allowed privilege sets and the interpreter can have forced and allowed privilege sets. The final forced set is the combination of the forced set assigned to the script and the forced set assigned to the interpreter restricted by the allowed set of the interpreter. The allowed set of the script does not restrict the final forced set.

Process Privilege Sets

Executing processes have process privilege sets computed from algorithms based on the contents of the file sets and any privileges inherited from the calling process. The process privilege sets are the inheritable, saved, permitted, and effective sets.

Inheritable Set

The inheritable set contains the privileges (if any) received from the parent process. A process passes its inheritable set to a new program during an `exec`(1) or a new process during a `fork`(2). The inheritable set of the new program or process always equals the inheritable set of the calling process. The new process or program can use only those inherited privileges that are also in the allowed set of its executable file, but passes all inheritable privileges to a new program or process. A program can clear its inheritable set and add any privileges in its permitted set to the inheritable set prior to a `fork`() or `exec`().

The system administrator can assign an inheritable set to a CDE action or command in an execution profile. The privileges are inherited when the user or role to which the execution profile is assigned starts the CDE action or executes a command from the profile shell.

Note – If a forced privilege is in the process's permitted set, that process can set the forced privilege in its own inheritable set and pass the forced privilege to a new process or program.

Saved Set

The saved set is a copy of the inherited privileges the process is allowed to use. The saved set equals the inheritable set restricted by the allowed set. Those privileges in the inheritable set also in the allowed set are put in the saved set. There are no interfaces for changing the saved set.

A program can query its saved set to determine the origination of a privilege. If the privilege is in the saved set, it is inherited for the current program invocation. If the privilege is not in the saved set, it is forced for the current program invocation.

A process may take a more limited (workstation-wide) action on a security-related task when started by a normal user (forced privilege), and a wider (network-wide) action on the same security-related task when started by an authorized user in an administrative role (inherited privilege).

Permitted Set

The permitted set contains the forced and inherited privileges a process can use. The permitted set is the forced set plus the inheritable set restricted by the allowed set. Those privileges in the inheritable set also in the allowed set are combined with the forced set and placed in the permitted set. A privileged process is a process with a permitted set not equal to zero.

Privileges can be removed from the permitted set, but not added. Once a permitted privilege is removed, it cannot be added back, it cannot be added to the inheritable set, and is removed from the inheritable set if it was added to the inheritable set prior to being removed from the permitted set.

As a security precaution, you can remove the privileges from the permitted set the program never uses. This way a program can never make use of an allowed privilege incorrectly assigned to its executable file or accidentally inherited.

Effective Set

Effective privileges are those permitted privileges a process uses for a single security-related task. By default, the effective set is initially equal to the permitted set, but a program should turn the effective set off at the beginning of execution to prepare for privilege bracketing.

Privilege bracketing is the practice of turning the effective privilege set off, then turning on (making effective) only those privileges needed for a specific security-related task, and turning them off as soon as they are no longer needed. See "Bracketing Effective Privileges" on page 76.

Change in User ID

Privilege-unaware programs change their UIDs either to gain or give up rights associated with the new UID. To simulate that action in a privilege-based system rather than a UID-based system, the effective and saved privilege sets are modified across setuid calls. If the `setuid`(2), `setreuid`(2), or `seteuid`(2) system call is called, the effective privilege set is copied to the saved set and the effective set is cleared. If you need the effective set, copy it back from the saved set or turn the effective privileges you need back on. If you need the original saved set (to determine the origination of a privilege), do the tests first or make a copy of the saved set.

The effective set is cleared based on the principle that a process cannot use privileges granted to the original caller while the user ID is changed. A `setuid` program can still manipulate privileges from the permitted set by putting them into the effective set. When a set UID program changes from its saved UID ID to the calling user ID, it gives up its privilege. When it changes back to the saved UID ID, it regains privilege.

Since set UID programs may not be aware of privileges, their privilege bracketing (see "Use Privilege Bracketing" on page 65) is tracked in the privilege sets.

Types of Privileged Applications

All privileged applications are part of the Trusted Computing Base (TCB). Some privileged applications have one or more forced privileges and might or might not inherit privileges. These applications are the Trusted Solaris equivalent of `setuid` applications in standard UNIX systems.

Other privileged applications have no forced privileges and always inherit privileges from the calling process. These applications are always called by a privileged process.

Privilege Names and Descriptions

The `priv_desc`(4) man page lists privilege names, manifest constant names, and description text for all system privileges.

Privileged Operations

The system calls that get and set file privilege sets require mandatory access and discretionary access to the file and may require privilege if access is denied. See the `fgetfpriv`(2) man page for specific details.

Setting File Privilege Sets

The `file_setpriv` privilege is required to set file privilege sets with the `setfpriv`(1) and `fsetfpriv`(2) system calls.

Keeping File Privilege Sets on an Executable File

When a process writes information to an executable file, the `file_setpriv` privilege is needed to prevent the file's forced and allowed privilege sets being set to `none`.

Core Files

The `proc_dumpcore` privilege must be effective for a privileged process to create a core file because the core file from a privileged process is likely to contain sensitive information. If this privilege is not effective, the process will not create a core file when it dies. For debugging purposes (only), you could make this privilege effective at the beginning of execution and leave it effective until the process dies.

Setting IDs

The calling process needs the `proc_setid` privilege in its effective set to change its user ID, group ID, or supplemental group ID.

Privilege Guidelines

Privileged applications should be developed in an isolated, protected environment separate from an operational Trusted Solaris system. Unfinished privileged applications are inherently untrustworthy and should not have an opportunity to compromise the security of a functioning system. The following additional practices are recommended for all privileged applications.

See Appendix B for information on secure application packaging.

Use Privilege Bracketing

When an application uses privilege, system security policy is being breached. Privileged tasks should be bracketed and carefully controlled to ensure that sensitive information is not compromised. See "Bracketing Effective Privileges" on page 76 for information on how to bracket privileges.

Avoid Shell Escapes

Shell escapes in an application can enable an end user to violate trust. For example, some mail applications interpret the `!command` line as a command and execute it. If a mail application is a trusted process, it runs with privileges. The end user can use this feature to create a script to take advantage of the mail application privileges. Applications should have this capability removed when they run in a trusted environment.

Avoid Command Line Execution

Running applications directly from the command line should be avoided if the application has been given privileges because the end user can take advantage of the privileges. For example, many application allow the end user to enter a command to execute followed by a document name. If the application has been given the privilege to override mandatory access controls (if the application needs to write down to an outside application), this could result in the end user opening a document that he or she does not ordinarily have the privileges to see.

Eliminate Covert Channels

Covert channels in privileged applications should be sought out and eliminated. A covert channel is an unintended path through which information can be transmitted in ways not protected by mandatory access controls. For example, in a privileged multilabel client/server application, the server has a queue of service requests. If unprivileged clients can add and remove requests from the queue and the queue has a finite size, the information on the full or not-full state of the queue can be exploited as a covert channel.

Data Types, Header Files, and Libraries

To use the programming interfaces described in this chapter, you need the following header file.

```
#include <tsol/priv.h>
```

The examples in this chapter compile with the following library:

```
-ltsol
```

Single Privileges

One privilege is represented by the `priv_t` type definition. You initialize a variable of type `priv_t` with a privilege ID that can be either the constant name or numeric ID. The constant name is preferred because it makes your code easier to read.

```
priv_t priv_id = PRIV_FILE_DAC_WRITE;
```

Privilege Set Structure

Privilege sets are represented by the `priv_set_t` data structure. You initialize variables of type `priv_set_t` with the `str_to_priv_set`(3TSOL) routine or the `PRIV_ASSERT` macro depending on whether you want to assert one privilege at a time using its privilege ID (PRIV_ASSERT) or convert a string of one or more privileges into a privilege set using a single interface (str_to_priv_set).

File Privilege Sets

The type of file privilege set to be worked on is represented by the priv_ftype_t type definition. Values are PRIV_ALLOWED and PRIV_FORCED.

Process Privilege Sets

The type of process privilege set to be worked on is represented by the priv_ptype_t type definition. Values are PRIV_EFFECTIVE, PRIV_INHERITABLE, PRIV_PERMITTED, and PRIV_SAVED.

Operations on File and Process Sets

The type of operation performed on a file or process privilege set is represented by the priv_op_t type definition. Not all operations are valid for every type of privilege set. Read the privilege set descriptions in “Privilege Sets” on page 60 for details.

Values are the following:

- PRIV_ON – Turn the privileges asserted in the priv_set_t structure on in the specified file or process privilege set.
- PRIV_OFF – Turn the privileges asserted in the priv_set_t structure off in the specified file or process privilege set.
- PRIV_SET – Set the privileges in the specified file or process privilege set to the privileges asserted in the priv_set_t structure. If the structure is initialized to empty, PRIV_SET clears (sets to none) the privilege set.

Privilege Macros

The privilege macros operate on single privileges and privilege sets. They are described on the priv_macros(5) man page. The macros do not directly change the privilege sets associated with files or processes, but manipulate variables of type priv_set_t.

Privilege Macro	Description
`PRIV_ASSERT(priv_set, priv_id)`	Put the privilege (*priv_id*) into the set (`priv_set`).
`PRIV_ISASSERT(priv_set, priv_id)`	Return non-zero if the privilege (*priv_id*) is asserted in (*priv_set*).
`PRIV_EQUAL(priv_set_a, Priv_set_b)`	Return non-zero if the sets are identical.
`PRIV_EMPTY(priv_set)`	Initialize the set to empty.
`PRIV_FILL(priv_set)`	Fill the set with all privileges.
`PRIV_ISEMPTY(priv_set)`	Return non-zero if the set is empty, and 0 if not empty.
`PRIV_ISFULL(priv_set)`	Return non-zero if the privilege contains all privileges defined for the system, and 0 otherwise.
`PRIV_CLEAR(priv_set, priv_id)`	Remove the privilege (*priv_id*) from set (*priv_set*).
`PRIV_INTERSECT(priv_set_a, priv_set_b)`	Store the intersection of *set_a* and *set_b* in *set_b*.
`PRIV_INVERSE(priv_set)`	Stores the inverse of *priv_set* in priv_set.
`PRIV_UNION(priv_set_a, priv_set_b)`	Store the union of *set_a* and *set_b* in *set_b*.
`PRIV_XOR(priv_set_a, priv_set_b,)`	Store the exclusive or of *set_a* and *set_b* in *set_b* .
`PRIV_ISSUBSET(priv_set_a, priv_set_b)`	Returns non-zero when all privileges asserted in *priv_set_a* are also asserted in *priv_set_b*, and 0 otherwise.
PRIV_TEST(priv_id, errno)	Test whether *priv_id* is in the effective set, and sets *errno* to 1 if True and 0 if False.

Interface Declarations

The following interfaces are available for handling file and process privilege sets. Where there is one set of interfaces to access a file using the pathname and another to access a file by the file descriptor, the examples use the pathname interfaces only because the syntax is almost identical.

System Calls

These system calls get and set file and process privilege sets.

File Sets

These system calls get and set the file privilege set using the full path name of the file. Refer to the `getfpriv`(2) man page.

```
int getfpriv( char *path,
    priv_ftype_t type,
    priv_set_t *priv_set);

int setfpriv( char *path,
    priv_op_t op,
    priv_ftype_t type,
    priv_set_t *priv_set);
```

These system calls get and set file privilege set using a file descriptor. Refer to the `getfpriv`(2) man page.

```
int fgetfpriv(int fd,
    priv_ftype_t type,
    priv_set_t *priv_set);

int fsetfpriv(int fd,
    priv_op_t op,
    priv_ftype_t type,
    priv_set_t *priv_set);
```

Process Sets

These system calls get and set process privilege sets. Refer to the `getppriv`(2) man page.

```
int getppriv(priv_ptype_t type, priv_set_t *priv_set);

int setppriv(priv_op_t op,
    priv_ptype_t type,
    priv_set_t *priv_set);
```

Note – You can also use the library routines below to access process privilege sets. The syntax is a little different, but the semantics are the same.

Library Routines

These library routines get process privilege sets, convert a privilege ID or privilege set between binary and text, and get the privilege description text for a specified privilege ID.

Process Privilege Sets

These library routines set the effective, permitted, and inheritable privilege sets on a process. Refer to the `set_effective_priv`(3TSOL) man page.

```
int set_effective_priv(priv_op_t op, int privno, priv_t priv_id);

int set_permitted_priv(priv_op_t op, int privno, priv_t priv_id);

int set_inheritable_priv(priv_op_t op, int privno, priv_t priv_id);
```

Note – You can also use `setppriv`(2) and `getppriv`(2) to access process privilege sets. The syntax is a little different, but the semantics are the same.

Binary and Text Privilege Translation

These library routines translate a privilege ID or a privilege set between binary and text. Refer to the `priv_to_str`(3TSOL) man page.

```
char* priv_to_str(const priv_t priv_id);

priv_t str_to_priv(const char *priv_name);

char* priv_set_to_str(priv_set_t *priv_set,
    const char sep,
    char *buf, int *blen);

char* str_to_priv_set(const char *priv_names,
    priv_set_t *priv_set,
    const char *sep);
```

Privilege Description Text

These library routines get the privilege text for a specified privilege ID. Refer to the `priv_to_str`(3TSOL) man page.

```
char* get_priv_text(const priv_t priv_id);
```

Translating Privileges

These library routines convert the specified privilege ID to its corresponding external name or numeric ID and back. These routines read the privilege names database file described on the `priv_name`(4) man page to translate between the *priv_id* and **string*.

Privilege ID to String

In this example, *priv_id* is initialized to the manifest constant name `PRIV_FILE_DAC_WRITE` and passed to `priv_to_str`(3TSOL) routine to convert it to the external name.

The header files and declarations for the code segments in this section are provided in the first program.

```
#include <tsol/priv.h>

main()
{
    priv_t priv_id = PRIV_FILE_DAC_WRITE;
    char *string;

    string = priv_to_str(priv_id);
    printf("Priv string = %s\n", string);
}
```

The `printf` statement prints the following:

```
Priv string = file_dac_write
```

String to Privilege ID

In the next example, the `string` returned from the `priv_to_str`(3TSOL) routine is passed to the `str_to_priv`(3TSOL) routine to convert the string to the numeric ID.

```
    priv_id = str_to_priv(string);
    printf("Priv ID = %d\n", priv_id);
```

The `printf` statement prints the following:

```
Priv ID = 6
```

Get Description Text for Privilege ID

The `get_priv_text`(3TSOL) routine returns the description text for the specified *priv_id*. The `priv_name`(4) man page lists the description text for all privileges in the system.

```
    string = get_priv_text(priv_id);
    printf("%s\n", string);
```

The `printf` statement prints the following:

```
Allows a process to write a file or directory whose
permission bits or ACL do not allow the process write permission.
```

Setting and Getting File Privilege Sets

The Trusted Solaris environment provides the user commands and programming interfaces described here for setting and getting the privilege sets of an executable file. If no forced and allowed privileges are set, by default the forced and allowed privilege sets contain `none`.

Note – If you set file privilege sets prior to execution, the new privilege sets take effect immediately and are used to compute the process privilege sets for the current execution. If you set file privilege sets during execution, they do not take effect until the next execution and have no effect on the process privilege sets for the current execution.

Commands for File Sets

To set and get the file privilege sets from the command line, use `setfpriv`(1) and `getfpriv`(1). The `file_setpriv` privilege is required with `setfpriv`(1) so this command must be executed from the profile shell with this privilege. See "Assigning File Privileges using a Script" on page 254 for information on using `setfpriv`(1) in a script.

This command line sets the file privilege sets on `executable` for the examples in this chapter. When you specify more than one privilege, the names are separated by commas with no spaces. If you want to use spaces, enclose the privilege names in double quotes ("privilege1, privilege2").

```
phoenix% setfpriv -s -f file_setpriv \
-a file_mac_write,proc_setid,file_setpriv executable
```

This command line produces output to verify the file privilege sets were set:

```
phoenix% getfpriv executable
executable FORCED: file_setpriv
ALLOWED: file_mac_write,file_setpriv,proc_setid
```

Programming Interfaces for File Sets

The privilege macros and system calls described in this section get and set file privilege sets. The program below has the header files and variable declarations for the entire series of examples for this chapter. It also contains code to set and get the file privilege sets for `execfile`, which will be `exec'd` later to show what happens to process sets during an `exec`.

The `setfpriv`(1) system call sets the forced and allowed privilege sets on `execfile` and requires the `file_setpriv` privilege. The `file_setpriv` privilege is in the forced set for `executable` to make it available in the permitted set during execution. By default, the effective set equals the permitted set, and all effective privileges are on until explicitly turned off in preparation for privilege bracketing. The use of `file_setpriv` in this code does not follow security guidelines until privilege bracketing is put into effect as described in "Bracketing Effective Privileges" on page 76.

```
/* cc priv.c -o executable -ltsol */

#include <tsol/priv.h>
#include <sys/types.h>
#include <errno.h>
#include <stdio.h>

/* Global Variables*/
extern int errno;
char buffer [3*1024];

main()
{
    char *priv_names = "file_mac_write,proc_setid";
    char *string;
    char *privilege;
    char *file = "/export/home/zelda/executable";
    char *execfile = "/export/home/zelda/execfile";
    priv_set_t priv_set, priv_get, permitted_privs, saved_privs;
    int length = sizeof(buffer);
    int retval;
    pid_t pid;

/* To use with exec() later */
    char *argv[8] = {"execfile"};

/* Initialize privilege set data structures */

    PRIV_EMPTY(&priv_get);
    PRIV_EMPTY(&priv_set);

/* Turn allowed privileges off. See text for discussion. */

    retval = setfpriv(execfile, PRIV_SET, PRIV_ALLOWED, &priv_get);

/* Assert the privileges in priv_names in a privilege set */
/* structure and assign to execfile. See text below for discussion */
/* on methods for asserting privileges */

    if((string = str_to_priv_set(priv_names, &priv_set, ",")) != NULL)
    printf("string = %s errno = %d\n", string, errno);
    retval = setfpriv(execfile,PRIV_ON, PRIV_ALLOWED, &priv_set);

/* Check that the allowed privilege set contains the privileges */

    retval = getfpriv(execfile, PRIV_ALLOWED, &priv_get);
    priv_set_to_str(&priv_get, ',', buffer, &length);
    printf("execfile Allowed = %s\n", buffer);

/* Initialize privilege set data structures */

    PRIV_EMPTY(&priv_set);
    PRIV_EMPTY(&priv_get);
/* Assert file_mac_write in a privilege set structure */

    PRIV_ASSERT(&priv_set, PRIV_FILE_MAC_WRITE);

/* Set the forced privilege set on execfile */

    retval = setfpriv(execfile, PRIV_ON, PRIV_FORCED, &priv_set);

/* Check that the forced privilege set contains the privilege */

    retval = getfpriv(execfile, PRIV_FORCED, &priv_get);
    priv_set_to_str(&priv_get, ‘,’, buffer, &length);
    printf(“execfile Forced =%s\n”, buffer);
}
```

The `printf` statements print the file privilege sets for `execfile` as follows:

```
execfile Allowed = file_mac_write,proc_setid
execfile Forced = file_mac_write
```

The output uses a comma (“,”) to separate the allowed privileges. The separator is specified in the calls to `priv_set_to_str`(3TSOL). The separator is not used when there is only one privilege in the set.

Turn Allowed Privileges Off

The forced set is a subset of the allowed set. Any privileges in the forced set are cleared when the allowed set is cleared. The allowed set is `none` by default, but it is a good practice to clear it first so you know you are starting from zero. Always clear and set the allowed set before you set the forced set. After the following code executes, the allowed and forced sets are both `none`.

```
PRIV_EMPTY(&priv_set);
retval = setfpriv(execfile, PRIV_SET, PRIV_ALLOWED, &priv_set);
```

Assert Privileges in Privilege Set Structure

You can use the `PRIV_ASSERT` macro or the `str_to_priv_set`(3TSOL) routine to assert privileges in a privilege set structure. `str_to_priv_set`() works well when you have two or more privileges to assert because you can do it in one statement; whereas, `PRIV_ASSERT` must be called for each privilege asserted in the set. This code uses the `str_to_priv_set`() routine for the allowed set and `PRIV_ASSERT` for the forced set. The `str_to_priv`() routine returns `NULL` on success and the string passed to it in `priv_names` on failure.

```
if((string = str_to_priv_set(priv_names, &priv_set, “,”)) != NULL)
    printf(“string = %s errno = %d\n”, string, errno);
```

```
PRIV_EMPTY(&priv_set);
PRIV_ASSERT(&priv_set, PRIV_FILE_MAC_WRITE);
```

Contents of Privilege Sets

The next examples operate on the process sets. It might be helpful to see the of file and process privilege sets before any operations. The process sets are calculated from the algorithms in "Process Privilege Sets" on page 61.

```
executable Allowed = file_mac_write,file_setpriv,proc_setid
executable Forced = file_setpriv
Permitted = file_mac_write,file_setpriv,proc_setid
Effective = file_mac_write,file_setpriv,proc_setid
Saved = file_mac_write,proc_setid
Inheritable = file_mac_write&file_setpriv,proc_setid
```

Bracketing Effective Privileges

Privilege bracketing involves turning the effective privileges off (they are on and equal the permitted set by default), then turning on (making effective) only those permitted privileges needed for a given interface call, and turning them off when the privileged call completes.

- A privileged process cannot be exploited by making privileges available to another process.
- A bug in the application code is less likely to cause misuse of a privilege if the privilege is turned off when not needed.
- The principle of least privilege is enforced because the process uses only the privileges it needs for the interfaces it is currently calling.
- The evaluation of a trusted application is easier because privilege bracketing shows the person evaluating the code exactly where privileges are used.

When you analyze which privileges are needed for an interface, look at what the interface does and the purpose of the privileges described on the man page for that interface. Some privileges have broader effects than others and should be treated with greater scrutiny.

- Privileges with broad effects are those that override mandatory access control or discretionary access control policies.
- Privileges with narrower effects are those that allow access to a restricted operation such as mounting a file system.

For example, it is relatively easy to examine a segment of code to see that it uses a privilege with the mount(1M) system call and tell whether the use of that privilege can be exploited in any way. It is more difficult to tell if the use of a privilege to override the mandatory or discretionary access policy to access a restricted file can be exploited.

It is up to you to perform privilege bracketing in your code and to do it correctly. Always remember that all privileges override some policy that is not allowed to untrusted processes, and handle your use of privileges with the needed care.

Procedure for Bracketing Privileges

The procedure for bracketing the setfpriv(1) system call and the effects it has on the effective set are summarized here. The code is shown in the next headings.

At the start of execution before bracketing, the permitted and effective sets contain these privileges:

```
Permitted = file_mac_write,file_setpriv,proc_setid
Effective = file_mac_write,file_setpriv,proc_setid
```

1. Clear the effective set at the beginning of the application.

   ```
   Permitted = file_mac_write,file_setpriv,proc_setid
   Effective = none
   ```

2. Bracket the setfpriv() system call.

   a. Turn the file_setpriv privilege on in the effective set right before you call the setfpriv() system call.

      ```
      Permitted = file_mac_write,file_setpriv,proc_setid
      Effective = file_setpriv
      ```

   b. Turn off the effective set immediately after the setfpriv() system call.

      ```
      Permitted = file_mac_write,file_setpriv,proc_setid
      Effective = none
      ```

Clear Effective Set

The example uses set_effective_priv(3TSOL) to clear the effective set at the beginning of the application. The PRIV_SET parameter clears the effective privilege set, and the zero (0) indicates there is no parameter list of privilege IDs.

```
if (set_effective_priv(PRIV_SET, 0) == -1)
    perror("Cannot clear effective privileges");
```

Continue Application Code

Turning the entire effective privilege set off is followed by application code until a privilege is needed.

Bracketing the Call

The example uses `set_effective_priv`(3TSOL) to bracket. The first call turns `the file_setpriv privilege` on (asserts it) in the effective set; the second call turns it off. The 1 indicates the privilege parameter list has one privilege constant (`PRIV_FILE_SETPRIV`) in it.

```
/* Turn file_setpriv on in effective set */
    if (set_effective_priv(PRIV_ON, 1, PRIV_FILE_SETPRIV) == -1)
        perror("Cannot assert PRIV_FILE_SETPRIV");

/* Make interface call */
    retval = setfpriv(execfile, PRIV_SET, PRIV_ALLOWED, &priv_get);

/* Turn the file_setpriv privilege off */
    if (set_effective_priv(PRIV_OFF, 1, PRIV_FILE_SETPRIV) == -1)
        perror("Cannot clear PRIV_FILE_SETPRIV");

/* Continue application code ...*/
```

Bracketing in Example

This next example shows the body of the example application code with comments indicating the places where `setfpriv`(1) should be bracketed.

```
    PRIV_EMPTY(&priv_get);
    PRIV_EMPTY(&priv_set);

/* Turn file_setpriv on in the effective set */
    retval = setfpriv(execfile, PRIV_SET, PRIV_ALLOWED, &priv_get);
/* Turn the file_setpriv privilege off */

    if((string = str_to_priv_set(priv_names, &priv_set, ",")) != NULL)
        printf("string = %s errno = %d\n", string, errno);

/* Turn file_setpriv on in the effective set */
    retval = setfpriv(execfile,PRIV_ON, PRIV_ALLOWED, &priv_set);
/* Turn the file_setpriv privilege off */

    retval = getfpriv(execfile, PRIV_ALLOWED, &priv_get);
    priv_set_to_str(&priv_get, ',', buffer, &length);
    printf("execfile Allowed = %s\n", buffer);
    PRIV_EMPTY(&priv_set);
    PRIV_EMPTY(&priv_get);
    PRIV_ASSERT(&priv_set, PRIV_FILE_MAC_WRITE);

/* Turn file_setpriv on in the effective set */
    retval = setfpriv(execfile, PRIV_ON, PRIV_FORCED, &priv_set);
/* Turn the file_setpriv privilege off */

    retval = getfpriv(execfile, PRIV_FORCED, &priv_get);
    priv_set_to_str(&priv_get, ',', buffer, &length);
    printf("execfile Forced =%s\n", buffer);
```

Checking and Modifying Privileges

Applications can notify users that privileges are missing, and establish the privilege sets for a program.

Check Permitted Privileges

An application can check the permitted privilege set to be sure the application has all privileges it needs to function. This way, if an application is missing a privilege, it can issue an error message to that effect. Continuing without all the needed privileges typically produces error messages that are more difficult to interpret.

The following example gets the permitted set and checks for `PRIV_FILE_MAC_WRITE`, `PRIV_PROC_SETID`, and `PRIV_FILE_SETPRIV`. The `PRIV_ISSUBSET` macro provides another way (not shown) to check if one privilege set contains all the privileges in another privilege set from within your source code.

```
/* Initialize privilege set data structure */
 PRIV_EMPTY(&permitted_privs);

/* Test for privileges in permitted set. */

 if (getppriv(PRIV_PERMITTED, &permitted_privs) == -1)
    perror("Cannot get list of permitted privileges\n");

 if (!PRIV_ISASSERT(&permitted_privs, PRIV_FILE_MAC_WRITE))
    fprintf(stderr, "Need: file_mac_write.\n");

 if (!PRIV_ISASSERT(&permitted_privs, PRIV_PROC_SETID))
    fprintf(stderr, "Need: proc_setid.\n");

 if (!PRIV_ISASSERT(&permitted_privs, PRIV_FILE_SETPRIV))
    fprintf(stderr, "Need: file_setpriv.\n");
```

Remove a Permitted Privilege

You can remove privileges from the permitted set, but once a privilege is removed it cannot be added back. Only privileges in the permitted set can be in the inheritable set so do not remove a permitted privilege that needs to be in the inheritable set. This example removes the `file_mac_write` privilege from the permitted set. The 1 indicates the parameter list has one privilege constant.

```
if(set_permitted_priv(PRIV_OFF, 1, PRIV_FILE_MAC_WRITE) == -1)
    perror ("Cannot remove file_mac_write from permitted set");
```

Before this call the permitted set contains these privileges:

```
executable Permitted = file_mac_write,file_setpriv,proc_setid
```

After this call the permitted set contains these privileges:

```
executable Permitted = file_setpriv,proc_setid
```

Check Saved Privileges

An application can check the saved privilege set to determine the origin of a privilege to take action based on the findings. This example gets the saved set and checks for `PRIV_PROC_SETID` and `PRIV_FILE_SETPRIV` and finds that the `file_setpriv` privilege is not inherited, but the `proc_setid` privilege is inherited.

```
PRIV_EMPTY(&saved_privs);

 if (getppriv(PRIV_SAVED, &saved_privs) == -1)
    perror("Cannot get list of saved privileges\n");
 if (!PRIV_ISASSERT(&saved_privs, PRIV_PROC_SETID))
    fprintf(stderr, "proc_setid not in saved set. \n");

 if (!PRIV_ISASSERT(&saved_privs, PRIV_FILE_SETPRIV))
    fprintf(stderr, "file_setpriv not in saved set.\n");
```

Clear and Set the Inheritable Set

To set the privileges that will be active after a new program is started using `exec`(2), first clear the inheritable set of the process, then initialize it with the privileges that you want the program to inherit.

This example clears the inheritable privilege set. The `PRIV_SET` parameter clears the inheritable privilege set, and the zero (0) parameter indicates there is no parameter list of privilege IDs.

```
if (set_inheritable_priv(PRIV_SET, 0) == -1)
    perror("Cannot clear inheritable privileges");
```

Before this call the inheritable set contains these privileges:

```
Inheritable = file_mac_write,file_setpriv,proc_setid
```

After this call the inheritable set contains this privilege:

```
Inheritable = none
```

The following example sets the `proc_setid` privilege in the inheritable privilege set. Any privilege in the permitted set can be placed in the inheritable set and placing any other privilege in the inheritable set results in an `Invalid Argument` error. Because the `proc_setid` privilege is in the permitted set for `executable`, it can be placed in the inheritable set. Because it is also in the allowed set for `execfile`, it can be used by the new program when `execfile` is `exec'd` in "Execute a File" on page 82.

```
if (set_inheritable_priv(PRIV_ON, 1, PRIV_PROC_SETID) == -1)
    perror("Cannot set proc_setid privilege in inheritable set");
```

After this call the inheritable set contains this privilege:

```
Inheritable = proc_setid
```

Fork a Process

When a child process is created by fork, its process sets are identical to the parent's process sets. This can be proven by querying the process privilege sets, `forking` a process, and querying the child process privilege sets:

Parent Process Privilege Sets

Before the `fork`, the `parent process has` the following privileges:

```
Forked Inheritable = proc_setid
Forked Saved = file_setpriv,proc_setid
Forked Permitted = file_setpriv,proc_setid
Forked Effective = none
```

System Call and Code

```
pid = fork();
if (pid > 0)
    exit(0);

PRIV_EMPTY(&priv_get);
retval = getppriv(PRIV_INHERITABLE, &priv_get);
printf("retval = %d errno = %d\n", retval, errno);
priv_set_to_str(&priv_get, ',', buffer, &length);
printf("Forked Inheritable = %s\n", buffer);

PRIV_EMPTY(&priv_get);
retval = getppriv(PRIV_SAVED, &priv_get);
printf("retval = %d errno = %d\n", retval, errno);
priv_set_to_str(&priv_get, ',', buffer, &length);
printf("Forked Saved = %s\n", buffer);

PRIV_EMPTY(&priv_get);
retval = getppriv(PRIV_PERMITTED, &priv_get);
printf("retval = %d errno = %d\n", retval, errno);
priv_set_to_str(&priv_get, ',', buffer, &length);
printf("Forked Permitted = %s\n", buffer);

PRIV_EMPTY(&priv_get);
retval = getppriv(PRIV_EFFECTIVE, &priv_get);
printf("retval = %d errno = %d\n", retval, errno);
priv_set_to_str(&priv_get, ',', buffer, &length);
printf("Forked Effective = %s\n", buffer);
```

New Process Privilege Sets

After the `fork`(2) system call, the `printf` statements print the following:

```
Forked Inheritable = proc_setid
Forked Saved = file_setpriv,proc_setid
Forked Permitted = file_setpriv,proc_setid
Forked Effective = none
```

Execute a File

When a file is `exec'd`, the process sets are computed based on the algorithms described in "Process Privilege Sets" on page 61.

Privilege Sets

The execfile for the new program has the following file privilege sets, which were set by the exec'ing process's application code:

```
execfile Allowed = file_mac_write,proc_setid
execfile Forced = file_mac_write
```

The exec'ing process has the following process sets:

```
Exec'd Inheritable = proc_setid
Exec'd Saved = file_setpriv,proc_setid
Exec'd Permitted = file_setpriv,proc_setid
Exec'd Effective = none
```

System Call

```
retval = execv(execfile, argv);
```

New Process Privilege Sets

After the exec(2) system call, the process sets are as follows.

```
execfile Allowed = file_mac_write,proc_setid
execfile Forced = file_mac_write
Exec'd Inheritable = proc_setid
Exec'd Saved = proc_setid
Exec'd Permitted = file_mac_write,proc_setid
Exec'd Effective = file_mac_write,proc_setid
```

Set User ID

The exec'd program's effective privileges are on by default. Because the new program has the proc_setid privilege in its effective set, you can call setuid(2) to see how the effective and saved sets change when the User ID changes. See "Change in User ID" on page 63 for the discussion.

```
retval = setuid(0);

PRIV_EMPTY(&priv_get);
retval = getppriv(PRIV_EFFECTIVE, &priv_get);
priv_set_to_str(&priv_get, ',', buffer, &length);
printf("Executable setuid effective = %s\n", buffer);
```

```
PRIV_EMPTY(&priv_get);
retval = getppriv(PRIV_SAVED, &priv_get);
priv_set_to_str(&priv_get, ’,’, buffer, &length);
printf(“Executable setuid saved = %s\n”, buffer);
```

The `printf` statements print the following:

```
Executable setuid effective = none
Executable setuid saved = file_mac_write,proc_setid
```

CHAPTER 4

Labels

The Trusted Solaris environment uses two types of labels: CMW labels and sensitivity labels (SLs).

This chapter describes the programming interfaces for performing general label operations such as initializing labels, retrieving portions of a CMW label, and comparing labels. It also describes the programming interfaces for accessing CMW labels on processes and file system objects. Chapter 5 provides code examples for the programming interfaces described in this chapter.

Clearances have the same construction as sensitivity labels, but perform a different function. Because of the similarity, some of the interfaces in this chapter accept clearances as parameters and some families of interfaces include an interface to handle clearances. Because clearances have a different function, however, all interfaces for managing clearances are described in Chapter 6 with code examples that use clearances.

CMW Label Description

A CMW label is a construct for labeling all processes and objects. It combines a sensitivity label with an information label so the labels can be programmatically translated and manipulated as a combined unit, or accessed individually.

Sensitivity Label

A sensitivity label has an ID field, one hierarchical classification (also called a level), and a set of one or more non-hierarchical compartments (also called categories). The classification represents a single level within a hierarchy, while the compartments represent distinct areas of information in a system. Compartments limit access to only those who need to know the information in a particular area. For example, persons with a Secret classification have access to the secret information specified by the compartment list and no other secret information. The sensitivity label classification and compartments together represent the sensitivity level of a process or object.

Comparing sensitivity labels means that the sensitivity label portion of the process CMW label is compared to the sensitivity label portion of the target CMW label and access is either granted or denied to the process based on whether the sensitivity level of the process dominates the sensitivity level of the target. The relationships of equality and dominance are described in "Test Label Relationships" on page 106.

CMW Label Display

CMW labels appear throughout the Trusted Solaris user interface as a single sensitivity label.

Acquiring CMW labels

Labels are acquired from workspaces and other processes. A user can start a process only at the current sensitivity label of the workspace in which he or she is working.

Process CMW Label

When a process is started from the workspace, the process CMW label inherits the sensitivity value of the workspace CMW label.

When a new process is created using `fork`(2), the new process inherits the CMW label values of its calling process.

When a new program is started with `exec`(1), the `exec'ing` process must have both discretionary and mandatory access to the new program's file.

The `setcmwplabel`(2) system call programmatically sets the process CMW label. You would use this call after `forking` or `exec'ing` a new process that should operate at another CMW label from the calling process. Privileges may be required. See "Privileged Operations" on page 87.

Object CMW Label

When an object is created by a process, the object inherits the CMW label values of its calling process.

When a privileged process writes down to an object, the system changes the sensitivity label of the object to be the same as the sensitivity label of the process. This protects the information written from the process at the higher sensitivity label from being accessed by other processes running at lower sensitivity labels.

The `setcmwlabel`(2) system call programmatically sets the CMW label on a file system object.

The File Manager lets an authorized user change the sensitivity label on an existing file's CMW label.

Privileged Operations

The system calls that get and set process and file system object CMW labels require mandatory and discretionary access to the process or file system object and may require privilege if access is denied by the system security policy. See "System Calls" on page 93 for a list of system calls.

Translating Binary Labels

The calling process needs the `sys_trans_label` privilege in its effective set to translate a label between binary and text if the label being translated is not dominated by the process's sensitivity label. This privilege is also required to check if a label is valid when the process sensitivity label does not dominate the label being checked.

Setting Process Labels

The calling process needs the `proc_setsl` privilege in its effective set to set its own sensitivity label to another label not equal to the current sensitivity label.

Downgrading and Upgrading Sensitivity Labels

The calling process needs the `file_owner` privilege in its effective set to downgrade the sensitivity label on a file not owned by the calling process.

Downgrading Sensitivity Labels

A process can set the sensitivity label on a file system object to a new sensitivity label that does not dominate the object's existing sensitivity label with the `file_downgrade_sl` privilege in its effective set.

Upgrading Sensitivity Labels

A process can set the sensitivity label on a file system object to a new sensitivity label that dominates the object's existing sensitivity label with the `file_upgrade_sl` privilege in its effective set.

Label Guidelines

This section provides guidelines for you to follow when your program must use privileges to bypass access controls or change the sensitivity label.

Sensitivity Labels

Most applications do not use privileges to bypass access controls because they operate in one of the following ways:

- An application is launched by one user or many users at one sensitivity label and accesses data in objects at that same sensitivity label.
- An application is launched by one user or many users at one sensitivity label and accesses data in objects at other sensitivity labels, but the mandatory access

operations are allowed by the system security policy as described in "Security Policy" on page 33.

- An application is launched by one user or many users at different sensitivity labels and accesses data in objects at that same sensitivity label by way of multilevel directories. Multilevel directories are described in Chapter 7.

If an application accesses data at sensitivity labels other than the sensitivity label of its process and access is denied, the process needs privilege to gain access. Privileges let the application bypass mandatory or discretionary access controls (`file_mac_read`, `file_dac_read`, `file_mac_write`, `file_dac_write`, `file_mac_search` or `file_dac_search`), change the process sensitivity label so mandatory access is granted (`proc_setsl`), or upgrade or downgrade the sensitivity label of the data (file_upgrade_sl, `file_downgrade_sl`). No matter how access is obtained, the application design must abide by the guidelines presented here to not compromise the classification of data accessed.

Bypassing Mandatory Access Controls

If you use privileges to bypass mandatory access restrictions, be careful your application does not write data out at a lower sensitivity label than the label at which it read the data. Also, your application design should not allow the accidental downgrading of data due to program errors.

Upgrading or Downgrading Sensitivity Labels

Follow these guidelines when your application changes its own sensitivity label or the sensitivity label of another object.

- Upgrade a sensitivity label whenever possible.

  A program that upgrades a sensitivity label is safer than a program that downgrades a sensitivity label because application errors that cause information leaks upgrade the data, rather than downgrade it. Upgrading data results in the over classification of the data, but is not a security breach. You can use privileges to downgrade a sensitivity label, but use these privileges very carefully.
- Never change a process sensitivity label more than once. Changes to the process sensitivity label increase the possibility of accidentally transmitting data between different levels. Any change to the process sensitivity label is an upgrade or downgrade of the information in the process address space.
- Close all file descriptors when changing a file or process sensitivity label so sensitive data is not available to other processes.

Creating a Process at Another Sensitivity Label

Instead of changing the process sensitivity label, `fork()` a new process and change the sensitivity label of the forked process so tasks can be performed at another level separate from the data in the forking process. The forked process should either return information to the forking process or send the information to another process.

Information returned by a forked process at a changed sensitivity label should provide no more information than absolutely necessary. For example, provide the success or failure of a computation, and not the actual data. Returning or passing specific information keeps the data used to make the computation secure and prevents data at one level from mixing with data at another level.

Data Types, Header Files, and Libraries

To use the programming interfaces described in this chapter, you need the following header file.

```
#include <tsol/label.h>
```

The examples in this chapter compile with the following library:

```
-ltsol
```

CMW label

The data structure `bclabel_t` represents a binary CMW label. Interfaces accept and return a binary CMW label in a structure of type `bclabel_t`.

Setting Flag

The `setting_flag` type definition defines CMW label flag values as follows:

`SETCL_SL` – Set the sensitivity label portion of the CMW label. `SETCL_ALL` – Set the entire CMW label.

Sensitivity Label

The `bslabel_t` type definition represents the sensitivity label portion of a binary CMW label. Interfaces accept as parameters and return binary sensitivity labels in a variable of type `bslabel_t`. The `bslabel_t` type definition is compatible with the `blevel_t` structure.

Binary Levels

The `blevel_t` structure represents a binary level, which is a classification and set of compartments in a sensitivity label or clearance. Interfaces accept and return binary levels in a structure of type `blevel_t`.

Type Compatibility

Any variable of type `bclear_t` or `bslabel_t` can be passed to a function that accepts a parameter of type `blevel_t`.

Range of Sensitivity Labels

The `brange_t` data structure represents a range of sensitivity labels. The structure holds a minimum label and a maximum label. The structure fields are referred to as `variable.lower_bound` and `variable.upper_bound`.

Accreditation Range

The `set_id` data structure currently accepts the following values: `SYSTEM_ACCREDITATION_RANGE; USER_ACCREDITATION_RANGE`.

Label Information

The `label_info` structure contains length specifications of items in the `label_encodings` file. The structure is returned by `labelinfo`(3TSOL).

Field	Description
`slabel_len`	Maximum sensitivity label length.

Field	Description
`clabel_len`	Maximum CMW label length.
`clear_len`	Maximum clearance label length.
`vers_len`	Version string length.
`header_len`	Maximum length of the printer banner.
`protect_as_len`	Maximum length of a printer banner page header string returned by `bcltobanner`(3TSOL).
`caveats_len`	Maximum length of the printer banner page string returned by `bcltobanner`(3TSOL).
`channels_len`	Maximum length of a printer banner page channels string.

Banner Fields

The `banner_fields` structure contains the translated text labels and strings for display on printer banner and trailer pages and at the top and bottom of document body page. The structure is returned by `bcltobanner`(3TSOL). The first five fields consist of pointers to character strings, and the second five consist of short integer lengths of memory preallocated to the corresponding string pointer.

Field	Description
`header`	String appears on top and bottom of the banner and trailer pages.
`protect_as`	String appears in protect as banner page section.
`caveats`	String appears in the caveats banner page section.
`channels`	String appears in the handling channels section.
`header_len`	Preallocated string memory length for header.
`protect_as_len`	Preallocated string memory length for protect as section.
`caveats_len`	Preallocated string memory length for caveats section.
`channels_len`	Preallocated string memory length for channels section.

Programming Interface Declarations

The following programming interfaces are available for general label operations and accessing labels on processes and file system objects.

System Calls

These system calls get and set a file or process CMW label, or get the file system label range.

Caution – Every process that sets a label on another process or file system object must set a valid label as defined in the `label_encodings` file, and must pass the correct binary form of the label. The text to binary translation functions correct the label as much as possible to ensure a correct binary label results from the translation. However, you might still use the `bslvalid`(3TSOL) routine to check that the label is valid. A correctly constructed binary label can be invalid for a given system or user and should be checked that it falls within the system or user accreditation range with the `blinset`(3TSOL) routine.

File CMW Label

These system calls get and set the file CMW label by the path name or file descriptor. Refer to the `setcmwlabel`(2) and `getcmwlabel`(2) man pages.

```
int setcmwlabel(const char *path,
    const bclabel_t *label, const setting_flag_t flag);
int getcmwlabel(const char *path, const bclabel_t *label);

int fsetcmwlabel(const int fd, const bclabel_t *label,
    const setting_flag_t flag);
int fgetcmwlabel(const int fd, bclabel_t *label);

int lsetcmwlabel(const int fd,
    const bclabel_t *label, const setting_flag_t flag);
int lgetcmwlabel(const int fd, bclabel_t *label);
```

Process CMW Label

These system calls get and set the process CMW label. Refer to the `setcmwplabel`(2) and `getcmwplabel`(2) man pages.

```
int setcmwplabel(const bclabel_t *label, const setting_flag_t flag);
int getcmwplabel(const bclabel_t *label);
```

File System Label Range

These system calls get the file system label range. Refer to the `getcmwfsrange`(2) man page.

```
int getcmwfsrange(char *path, brange_t *range);
int fgetcmwfsrange(int fd, brange_t *range);
```

Library Routines

These library routines access, initialize, compare, translate, and verify labels. Library routines also obtain information on `label_encodings`(4).

CMW Label Initialization

These routines initialize a CMW label to `ADMIN_HIGH`, `ADMIN_LOW`, or undefined (similar to `NULL`). Refer to the `blmanifest`(3TSOL) man page.

```
void bclhigh(bclabel_t *label);
void bcllow(bclabel_t *label);
void bclundef(bclabel_t *label);
```

CMW Label Portions

These routines access the sensitivity label portion of a CMW label. Refer to the `blportion`(3TSOL) man page.

```
void getcsl(bslabel_t *destination_label, const bclabel_t *source_label);
void setcsl(bclabel_t *destination_label, const bslabel_t *source_label);
bslabel_t *bcltosl(bclabel_t *label);
```

Sensitivity Label Initialization

These routines initialize a sensitivity label to `ADMIN_HIGH`, `ADMIN_LOW`, or undefined. Refer to the `blmanifest`(3TSOL) man page.

```
void bslhigh(bslabel_t *label);
void bsllow(bslabel_t *label);
void bslundef(bslabel_t *label);
```

Level Comparison

These routines compare two levels to see if *level1* equals, dominates, or strictly dominates *level2*. A level is a classification and set of compartments in a sensitivity label or clearance.

A returned non-zero is true and 0 is false. Refer to the `blcompare`(3TSOL) man page.

```
int blequal(const blevel_t *level1, const blevel_t *level2);
int bldominates(const blevel_t *level1, const blevel_t *level2);
int blstrictdom(const blevel_t *level1, const blevel_t *level2);
int blinrange(const blevel_t *level, const brange_t *range);
```

Label Types

These routines check or set label type. A label can be a defined or undefined CMW label or sensitivity label. Refer to the `bltype`(3TSOL) man page.

```
int bltype(const void *label, const unsigned char type);
void setbltype(void *label, const unsigned char type);
```

Level Bounds

These routines compare two levels to find the sensitivity level that represents the greatest lower bound (`blminimum`(3TSOL)) or least upper bound (`blmaximum`(3TSOL)) of the range bounded by the two levels. A level is a classification and set of compartments in a sensitivity label or clearance. Refer to the `blcompare`(3TSOL) man page.

```
void blmaximum(blevel_t *maximum_label, const blevel_t *bounding_label);
void blminimum(blevel_t *minimum_label, const blevel_t *bounding_label);
```

Label Encodings File

The `label_encodings` file is a text file maintained by the system administrator that contains site-specific label definitions and constraints. This file is kept in `/etc/security/tsol/label_encodings`. See *Trusted Solaris Label Administration* and *Compartmented Mode Workstation Labeling: Encodings Format* for information on the `label_encodings` file.

These routines return information specified in the `label_encodings` file on maximum string lengths, version of `label_encodings` file in use, and text color name for the specified binary level.

- Maximum string lengths. Refer to the `labelinfo`(3TSOL) man page.

```
int labelinfo(struct label_info *info);
```

- Version in use. Refer to the `labelvers`(3TSOL) man page.

```
int labelvers(char **version, const int length);
```

- Text color name for a binary level. Refer to the `bltocolor`(3TSOL) man page.

```
char bltocolor(const blevel_t *label);
char bltocolor_t(const blevel_t *label, const int size, char * color_name);
```

Valid Sensitivity Label

This routine checks whether the specified sensitivity label is valid for the system (is defined in the `label_encodings` file for the system). Refer to the `blvalid`(3TSOL) man page.

```
int bslvalid(const bslabel_t *senslabel);
```

Accreditation range

This routine checks whether the sensitivity label falls within the system accreditation range as set in the `label_encodings` file for the system. Refer to the `blinset`(3TSOL) man page.

```
int blinset(const blevel_t *senslabel, const set_id *id);
```

Binary Translation

These routines translate a binary CMW label or sensitivity label from binary to text and back again. When translating from a string to binary, the string can be text or hexadecimal when *flag* is `NEW_LABEL` or `NO_CORRECTION`. Refer to the `bltos`(3TSOL) and `stobl`(3TSOL) man pages.

Note – See Chapter 14 for interfaces that translate binary labels to text, clip the final label according to a specified width, and use a font list for display in Motif-based graphical user interfaces (GUIs).

- CMW Label and text

```
int bcltos(const bclabel_t *label,
    char **string,    const int length,
    const int flags);
int stobcl(const char *string,
    bclabel_t *label,
    const int flags,
    int *error);

/* Translate and Clip string to length */
```

```
char *sbcltos(const bclabel_t *label,
    const int length);

/* Translate for inclusion on printer banner and header pages */
char *bcltobanner(const bclabel_t *label,
    struct banner_fields *fields,
    const int flags);
```

- Binary Sensitivity Label and text

```
int bsltos(const bslabel_t *label,
    char **string,
    const int length,
    const int flags);
int stobsl(const char *string,
    bslabel_t *label,
    const int flags,
    int *error);

/* Translate and clip string to length */
char *sbsltos(const bslabel_t *label,
    const int length);
```

Binary and Hexadecimal Translation

These routines translate a binary CMW label or sensitivity label from binary to hexadecimal and back again. Refer to the `btohex`(3TSOL) and `hextob`(3TSOL) man pages.

- Allocate and Free Memory for reentrant functions.

```
char h_alloc(const unsigned char id);
void h_free(char *hex);
```

- Translate CMW label between binary and Hexadecimal.

```
char *bcltoh(const bclabel_t *label);
char *bcltoh_r(const bclabel_t *label, char *hex);
int htobcl(const char *hex, bclabel_t *label);
```

- Translate sensitivity label between binary and Hexadecimal.

```
char *bsltoh(const bslabel_t *label);
char *bsltoh_r(const bslabel_t *label, char *hex);
int htobsl(const char *hex, bslabel_t *label);
```

CHAPTER 5

Label Code Examples

This chapter presents example code showing how to use the programming interfaces discussed in Chapter 4.

Retrieving Version String

The components of sensitivity labels and clearances; and the handling caveats that appear on printer output are specified in a site-specific `label_encodings`(4) file. Some of the programming interfaces described in this chapter access these specifications in the `label_encodings` file, and therefore, their outputs vary depending on the `label_encodings` file in use for a particular site.

This example gets the version string of the `label_encodings` file accessed in some of the code examples in this chapter, and prints the version string to standard out.

```
#include <tsol/label.h>

main()
{
    int retval, length = 0;
    char *version = (char *)0;

    retval = labelvers(&version, length);
    if(retval > 0)
        printf("Version string = %s\n", version);
}
```

The `printf` statement prints the following:

```
Version string = TRUSTED SOLARIS MULTI-LABEL SAMPLE VERSION -
    5.5 00/07/19
```

Initialize Binary Labels and Check Types

These interfaces initialize a label to `ADMIN_HIGH`, `ADMIN_LOW`, and undefined. `ADMIN_HIGH` represents the highest possible classification number including all compartments and all markings. `ADMIN_HIGH` strictly dominates every other label in the system. Normal users cannot read or write files at `ADMIN_HIGH`.

`ADMIN_LOW` represents a classification of zero with no compartments. All users can read or execute files with a sensitivity label of `ADMIN_LOW`. No normal user can write files at `ADMIN_LOW`. Every other label in the system strictly dominates `ADMIN_LOW`. `ADMIN_LOW` is assigned to publicly accessible system files and commands.

Undefined is similar to `NULL` and represents an invalid label. A sensitivity label is undefined when the ID field is initialized to `SUN_SL_UN`. An undefined label is invalid. CMW labels do not have an undefined state, only the sensitivity portion has an undefined state.

A CMW label or sensitivity label, is defined when the ID field in the label structure is initialized to `SUN_CMW_ID` or `SUN_SL_ID`.

This example initializes a label to `ADMIN_HIGH` and `ADMIN_LOW`, and then and checks and prints the label type.

```
#include <tsol/label.h>

main()
{
    int retval;
    bslabel_t psenslabel;
    bclabel_t pCMWlabel;

/* initialize labels*/
    bclundef(&pCMWlabel);
    bslhigh(&psenslabel);

/* Check label types */
    retval = bltype(&psenslabel, SUN_SL_ID);
    printf("Is sensitivity label defined? %d\n", retval);
}
```

The `printf` statements print the following. Non-zero is True and 0 is False.

```
Is sensitivity label defined? 1
```

Get Process CMW Label

You can get the process CMW label and perform operations on it as a unit, or extract the sensitivity label portion and perform independent operations on it. This example gets the process CMW label, extracts the sensitivity label portion, translates the process CMW label to a text string, and prints the process CMW label.

```
#include <tsol/label.h>

main()
{
    int retval, length = 0;
    bclabel_t pCMWlabel;
    bslabel_t psenslabel;
    char *string;

/* Get process CMW label */
    retval = getcmwplabel(&pCMWlabel);

/* Get sensitivity label portion */
    getcsl(&psenslabel, &pCMWlabel);

/* Translate the process CMW label to text and print */
    retval = bcltos(&pCMWlabel, &string, length, LONG_CLASSIFICATION);
    printf("Process CMW label = %s\n", string);
}
```

The `printf` statement prints the following where `ADMIN_LOW` is the information label and `[C]` is the sensitivity label. This CMW label means the process is running at a sensitivity level of Confidential ([C]) with an information label of `ADMIN_LOW`. The CMW label is inherited from the workspace in which the program is run.

```
Process CMW label = ADMIN_LOW [C]
```

The text output depends on the flag parameter to bcltos(3TSOL) and specifications in label_encodings(4). See "Binary to Text Label Translation Routines" on page 114 for information on flag parameter values.

Set SL Portion of Process CMW Label

This example gets the calling process's CMW label, and sets the sensitivity portion to TOP SECRET (upgrades the label). The altered CMW label is set on the privileged process. The calling process needs the proc_setsl privilege in its effective set to change its sensitivity label. The code comments indicate where privilege bracketing as described in Chapter 3 should occur. (Note that this example will work only if the process clearance dominates TOP SECRET.)

```
#include <tsol/label.h>

main()
{
    int retval, error, length = 0;
    bclabel_t pCMWlabel;
    bslabel_t psenslabel;
    char *string = "TOP SECRET",  *string1 = (char *)0;

/* Create new sensitivity value and set CMW label to the value */
    retval = stobsl(string, &psenslabel, NEW_LABEL, &error);
    setcsl(&pCMWlabel, &psenslabel);

/* Set process CMW label with new CMW label */
/* Turn proc_setsl on in the effective set */
    retval = setcmwplabel(&pCMWlabel, SETCL_SL);
 /* Turn proc_setsl off */

}
```

The printf statement prints the following where ADMIN_LOW is the information label and [TS] is the sensitivity label.

```
Process CMW label = ADMIN_LOW [TS]
```

The text output depends on the flag parameter to bcltos(3TSOL) and specifications in label_encodings(4). See "Binary to Text Label Translation Routines" on page 114 for information on flag parameter values.

The SETCL_SL value passed to setcmwplabel(2) sets the sensitivity label portion of the CMW label.

Get File CMW Label

You can get a file CMW label and perform operations on it as a unit, or extract the SL portion and perform independent operations on the SL portion.

This example gets the file CMW label and extracts the sensitivity label portion. The `fgetcmwlabel`(2) and `lgetcmwlabel`(2) routines are used the same way, but operate on a file descriptor or symbolic link.

```
#include <tsol/label.h>

main()
{
    int retval, length = 0;
    bclabel_t fileCMWlabel;
 bslabel_t fsenslabel;
    char *string = (char *)0;

/* Get file CMW label */
    retval = getcmwlabel("/export/home/zelda/afile", &fileCMWlabel);

/* Get sensitivity label portion */
    getcsl(&fsenslabel, &fileCMWlabel);

/* Translate fileCMWlabel to text and print */
    retval = bcltos(&fileCMWlabel, &string, length, LONG_CLASSIFICATION);
    printf("File CMW label = %s\n", string);
}
```

```
File CMW label = [CONFIDENTIAL]
```

Set SL Portion of File CMW Label

In this example, the process is running at Confidential with a Top Secret clearance. The process upgrades the sensitivity label portion of a file's CMW label to Top Secret and needs the `file_upgrade_sl` privileges because a label upgrade is a task that always requires privilege. The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

A process cannot upgrade an object's sensitivity label to a higher level than its own clearance. "Find Greatest Level and Lowest Level" on page 128" describes how to check the process clearance against a sensitivity label.

If the system administrator has configured the system in the /etc/system file to not show file names when a file's CMW label has been upgraded, the upgraded file in this example will not be visible to a user who logs in at Confidential and lists the directory. See "Query System Security Configuration" on page 45 for information on querying the system variables.

Note – In the character-coded to binary translation, a new label is created with the NEW_LABEL flag parameter. See "Text to Binary and Hexadecimal Label Translation Routines" on page 116 for information on the text to binary label translation and the flag parameter.

The SETCL_SL value passed to the setcmwlabel(2) system call indicates that the sensitivity portion is to be set. The new sensitivity label must be in the containing file system's label range, and the required privileges must be effective.

```
#include <tsol/label.h>

main()
{
    int retval, error;
    bclabel_t fileCMWlabel;
    bslabel_t fsenslabel;
    char *string = "TOP SECRET",
      *string1 = "TOP SECRET";

/* Create new sensitivity label value */
/* Turn sys_trans_label on in the effective set */
    retval = stobsl(string, &fsenslabel, NEW_LABEL, &error);
/* Turn sys_trans_label off */

/* Set sensitivity label portion of CMW label to new value */
    setcsl(&fileCMWlabel, &fsenslabel);

/* Set file CMW label */
/* Turn file_upgrade_sl privilege on in the effective set */
    retval = setcmwlabel("/export/home/zelda/afile",
      &fileCMWlabel, SETCL_SL);
 /* Turn file_upgrade_sl off */
 }
```

Use getlabel(1) to check the change in the file label. Before the program above runs, the CMW label for afile is as follows:

```
phoenix%  getlabel afile
afile: [CONFIDENTIAL]
```

After the program runs, the CMW label is as follows. Be aware that if you use the getlabel(1) command at Confidential, you will need the sys_trans_label privilege to read the label on a Top Secret file.

```
phoenix%  getlabel afile
afile: [TOP SECRET]
```

File System Label Range

The file system label range specifies the upper and lower bounds to the sensitivity of data contained in the file system. The `getcmwfsrange()` and `fgetcmwfsrange()` system calls return a structure that contains the upper and lower bound of the file system sensitivity label range.

- Variable file system –
  - When the upper and lower bounds are not equal, the file system has a label range and is a multilabel file system. A multilabel file system supports all security attributes distinctly for every file system object.
  - When the upper and lower bounds are equal, the file system is a single-label file system. This type of file system supports all security attributes distinctly for every file system object.
- Fixed file system – When the upper and lower bounds are equal, the file system is a single-label file system. The file system's system sensitivity label comes from the mount specified in `vfstab_adjunct(4)`. A single-label file system supports security attributes for the file system, but not for every file system object.

How to query the file system security attributes in the inode or in the `vfstab_adjunct(4)` is described in "Query File System Security Attributes" on page 46 in Chapter 2.

The following sections describe two situations where a program might get the file system label range and test a sensitivity label against it before taking further action.

Test Range Before Changing File CMW Label

Before upgrading a file CMW label (as was done in the previous example), it is a good idea to test the file system label range to be sure the file's new sensitivity label is within the sensitivity label range of the file.

This example converts text strings to a new binary sensitivity label, gets the file system label range, and checks if the new sensitivity label is within the file system's label range.

```
#include <tsol/label.h>
```

```
main()
{
    int retval, error;
    bclabel_t fileCMWlabel;
    bslabel_t fsenslabel;
    brange_t range;
    char *string = "TOP SECRET";

/* Create new sensitivity label value */
    retval = stobsl(string, &fsenslabel, NEW_LABEL, &error);

/* Get file system label range */
    retval = getcmwfsrange("/export/home/zelda/afile", range);

/* Test new sensitivity label against label range */
    retval = blinrange(&fsenslabel, range);
    if(retval > 0)
        {/* Proceed with file CMW label upgrade. */}
}
```

Test Range before Routing Data to Device

Always check the label range on a device special file before using the Trusted Solaris interfaces to allocate a device and route input to the device. The input routed to the device should be within the label range of the device-special file.

Test Label Relationships

If your application accesses data at different sensitivity labels, you can perform checks in your code to be sure the process label has the correct relationship to the data label before you allow an access operation to take place. You check the sensitivity label to find out if access will be allowed by the system or if privilege is required to override access restrictions.

These examples show how to test two sensitivity labels for equality, dominance, and strict dominance. The Trusted Solaris environment checks the process clearance when the process changes the sensitivity label on any object or writes to an object of a higher sensitivity label. "Find Relationships Between Two Levels" on page 127 describes how to test for the relationship between a clearance and a sensitivity label.

Find Relationship Between Two Levels

A level is a classification and set of compartments for a sensitivity label or clearance; and is represented by the data type `blevel_t`. Two levels can be equal, one can dominate the other, or one can strictly dominate the other.

- Equals – One level is equal to another when its classification is arithmetically equal to the other's classification (by means of its place in the classifications hierarchy), and its compartments contain all the other's compartments and no additional compartments.
- Dominates – One level dominates another when its classification is arithmetically greater than or equal to the other's classification (by means of its place in the classifications hierarchy), and its compartments contain all the other's compartments.
- Strictly dominates – Level one is said to strictly dominate level two when level one dominates level two, but is not equal to level two.

This example tests the process sensitivity label against a file's sensitivity label. The code for getting the process and file CMW label and extracting the sensitivity label portion is not shown. See "Get Process CMW Label" on page 101 and "Get File CMW Label" on page 103 for example code to perform these operations.

In this example, the process sensitivity label is Confidential and the file sensitivity label is Confidential. The labels are equal, the process label dominates the file label, but does not strictly dominate the file label.

```
#include <tsol/label.h>

main()
{
    int equal, dominate, strictdom, retval;
    bslabel_t *plabel, *filelabel;
    bclabel_t fileCMWlabel, pCMWlabel;

/* Get file and process CMW labels */
    retval = getcmwlabel("/export/home/zelda/afile", &fileCMWlabel);
    retval = getcmwplabel(&pCMWlabel);

/* Get sensitivity labels */
    plabel = bcltosl(&plabel);
    filelabel = bcltosl(&filelabel);

/* Once have both labels, test for equality */
    equal = blequal(plabel, filelabel);
    printf("Process label equals file label? %d\n", equal);

/* Test for dominance */
    dominate = bldominates(plabel, filelabel);
    printf("Process label dominates file label? %d\n", dominate);

/* Test for strict dominance */
```

```
    strictdom = blstrictdom(plabel, filelabel);
    printf("Process label strictly dominates file label? %d\n", strictdom);
}
```

The `printf` statement prints the following where any value greater than zero is true and zero is false.

```
Process label equals file label? 1
Process label dominates file label? 1
Process label strictly dominates file label? 0
```

Accessing CMW Label Portions

The procedure "Get Process CMW Label" on page 101 uses the `getcsl`(3TSOL) and `setcsl`(3TSOL) routines to get and set the sensitivity portion of a process and file CMW label. This example uses routines to return a pointer to the sensitivity label portion of a CMW label.

```
#include <tsol/label.h>

main()
{
    bslabel_t *senslabel;
    bclabel_t pCMWlabel;
    int retval;

    retval = getcmwplabel(pCMWlabel);

/* Get a pointer to the sensitivity label portion of cmwlabel */
    senslabel = bcltosl(&pCMWlabel);

}
```

Finding Binary Level Bounds

The next two examples find the greatest and lowest values between two variables of type `blevel_t`. These interfaces let you compare two levels to find the level that represents the greatest lower bound (`blminimum`(3TSOL) routine) or least upper bound (`blmaximum`(3TSOL) routine) bounded by the two levels. A level can be a sensitivity label or a clearance.

In the example, *senslabel* is ADMIN_LOW and *plabel* is Confidential. The code finds the greatest lower bound and least upper bound of the range created by these two levels. The first example finds the greater of the classifications and the greater of all the compartments of the two variables passed to the blmaximum() routine and puts that value into the first parameter. This operation is called finding the least upper bound because it finds the lowest level that dominates both the original parameter values passed to the routine.

```
#include <tsol/label.h>

main()
{
    int retval, length = 0;
    char *string = (char *)0, *string1 = (char *)0;
    bslabel_t senslabel, plabel;
    bclabel_t pCMWlabel;

/* Initialize a label to ADMIN_LOW */
    bsllow(&senslabel);

/* Get process sensitivity label */
    retval = getcmwplabel(&pCMWlabel);
    getcsl(&plabel, &pCMWlabel);

    blmaximum(&senslabel, &plabel);
    retval = bsltos(&senslabel, &string, length, LONG_WORDS);
    printf("Maximum = %s\n", string);
```

The printf statements print the following where Confidential is the lowest level that dominates Confidential and ADMIN_LOW.

```
Maximum = CONFIDENTIAL
```

This part of the example finds the lower of the classifications and the lower of only those compartments contained in both parameters passed to the blminimum() routine, and puts that value into the first parameter. This operation is called finding greatest lower bound because it finds the greatest level dominated by both of the original parameter values passed to the routine.

```
    bsllow(&senslabel);

    blminimum(&senslabel, &plabel);
    retval = bsltos(&senslabel, &string1, length, LONG_WORDS);
    printf("Minimum = %s\n", string1);
}
```

The printf statements print the following where ADMIN_LOW is the highest level dominated by ADMIN_LOW and Confidential.

```
Minimum = ADMIN_LOW
```

Check Accreditation Range

Use the `blinset()` routine to check whether a sensitivity label is within the system or user accreditation range. The system accreditation range is all the labels valid for the system including `ADMIN_HIGH` and `ADMIN_LOW`. The classification and compartments of all sensitivity labels processed by a system must dominate the minimum sensitivity label of the system accreditation range and be dominated by the maximum sensitivity label of the system accreditation range. The system administrator defines the system accreditation range in the `label_encodings`(4) file.

The user accreditation range is all the sensitivity labels valid for a user and never includes `ADMIN_HIGH` or `ADMIN_LOW`. The classification and compartments of all sensitivity labels assigned to a user must dominate the minimum sensitivity label of the system accreditation range and be dominated by the maximum sensitivity label of the system accreditation range. The system administrator assigns the sensitivity label range (user accreditation range) to users and roles through the administrative user interface.

In this example the sensitivity label is checked against the system accreditation range (id.type = `SYSTEM_ACCREDITATION_RANGE`) and user accreditation range (id.type = `USER_ACCREDITATION_RANGE`).

```
#include <tsol/label.h>

main()
{
    char *string = "CONFIDENTIAL", *string1 = "UNCLASSIFIED";
    int sysval, userval, error, retval;
    bslabel_t senslabel;
    set_id id;

    retval = stobsl(string, &senslabel, NEW_LABEL, &error);
    id.type = SYSTEM_ACCREDITATION_RANGE;
    sysval = blinset(&senslabel, &id);
    id.type = USER_ACCREDITATION_RANGE;
    userval = blinset(&senslabel, &id);

    printf("System Range? = %d User Range? %d\n", sysval, userval);
}
```

The `printf` statement prints the following where 1 indicates the sensitivity label is within range, and 0 indicates one of the following: the sensitivity label is not a valid label, not in the specified range, or the calling process's sensitivity label does not dominate the sensitivity label and the calling process does not have the `sys_trans_label` privilege in its effective set.

```
System Range? = 1 User Range? = 1
```

Validating Labels

A valid label is a label defined in the `label_encodings` file. You can use the `bslvalid`(3TSOL) routine to check if a sensitivity label is valid. The sensitivity label of the calling process must dominate the sensitivity label being checked or the calling process needs the `sys_trans_label` privilege in its effective set for this operation to succeed.

```
#include <tsol/label.h>

main()
{
    int retval, error;
    bslabel_t senslabel;

    char *string = "CONFIDENTIAL";

    retval = stobsl(string, &senslabel, NEW_LABEL, &error);
    retval = bslvalid(&senslabel);
    printf("Valid Sensitivity Label? = %d\n", retval);
}
```

The `printf` statement prints the following where 1 indicates the label is valid; -1 indicates the `label_encodings` file is inaccessible; and 0 indicates the label is not valid, or the process sensitivity label does not dominate the clearance and the process does not have the `sys_trans_label` privilege in its effective set:

```
Valid Sensitivity Label? = 1
```

Getting Character-Coded Color Names

This example uses the `bltocolor`(3TSOL) call to get the character-coded color name associated with a sensitivity label of a particular level. The character-coded color names are specified in the `label_encodings`(4) file.

This example inquires about the character-coded color name associated with Confidential sensitivity labels. The process is running at Confidential so no privileges are needed for the inquiry. The calling process needs the `sys_trans_label` privilege in its effective set to inquire about labels that dominate the current process's sensitivity label.

```
#include <tsol/label.h>
```

```
main()
{
    int retval, error;
    bslabel_t senslabel;
    char *string = “CONFIDENTIAL”;
    char *string1;

    retval = stobsl(string, &senslabel, NEW_LABEL, &error);

    string1 = bltocolor(&senslabel);
    printf(“Confidential label color = %s\n”, string1);
}
```

The `printf` statement prints the following:

```
Confidential label color = BLUE
```

Label Encodings Information

The `labelinfo`(3TSOL) routine returns maximum length values as short integers for various character data fields from the label library. An application laying out a field that contains label information might use these lengths. The length values change depending on the actual contents of the `label_encodings`(4) file.

```
#include <tsol/label.h>
main()
{
    int retval;
    struct label_info info;

    retval = labelinfo(&info);
    printf(“Max sensitivity label length = %d\n”, info.slabel_len);
    printf(“Max CMW label length = %d\n”, info.clabel_len);
    printf(“Max clearance length = %d\n”, info.clear_len);
    printf(“Max version string length = %d\n”, info.vers_len);
    printf(“Max banner and trailer string length = %d\n”, info.header_len);
    printf(“Max protect as section string length = %d\n”,
        info.protect_as_len);
    printf(“Max caveats section string length = %d\n”, info.caveats_len);
    printf(“Max handling channels string length = %d\n”, info.channels_len);
}
```

The `printf` statements print the following lengths:

```
Max sensitivity label length = 45
Max CMW label length = 259
Max clearance length = 76
Max Version String length = 56
Max Banner and trailer page string length = 13
Max Protect as section string length = 256
Max Caveats section string length = 62
Max Handling channels section string length = 81
```

Translating Labels

All labels can be represented in binary, text, or hexadecimal. Within the kernel all labels are stored in binary form, and binary is the form used for labels passed to and received from programming interfaces.

- Binary Labels – Classifications are stored as an integer and compartments are stored as bit vectors using 0's and 1's.
- Character-Coded Labels – Human-readable labels that display classifications, and compartments using the names defined in the `label_encodings`(4) file.
- Hexadecimal Labels – The character-coded representation of the hexadecimal number that represents the same bit pattern as the corresponding binary label. The label has plain text characters but does not reveal the classification or compartment names. A process can store a label in plain text form when it will be read by processes at arbitrary labels.

Note – If label names are stored in files at a sensitivity label lower than the sensitivity level of the label names, or in files where users without the proper permissions or authorization could access them, store the label names in either binary or hexadecimal format to make them unreadable.

Binary and Text Label Translation

Labels can be translated from binary to text and back again. The calling process needs the `sys_trans_label` privilege in its effective set to translate any label not dominated by the process's sensitivity label.

- Text characters can be input in any combination of upper and lowercase letters, but they are always output all uppercase.
- Text label input and output formats consist of classifications and words defined in the `label_encodings` file. Classification names and words may contain embedded blanks or punctuation if they are defined that way in the `label_encodings` file.

Binary to Text Label Translation Routines

These examples translate binary labels to text. The translation uses the keyword settings in `label_encodings`(4) and the flag parameter value. Not all flag values make sense for every label, although nothing stops you from using any flag with any type of label. The descriptions state the label type a flag is to be used with. Settings that apply to sensitivity labels also apply to CMW labels.

- `LONG_WORDS`—Translate a binary label using the long names for words.
- `SHORT_WORDS`—Translate a binary label using the short names for words.
- `LONG_CLASSIFICATION`—Translate a binary label using long names for the classification.
- `SHORT_CLASSIFICATION`—Translate a binary label using short names for the classification.
- `NO_CLASSIFICATION`—Do not include the classification in the translation of a binary sensitivity label label
- `VIEW_INTERNAL`—Use internal names for the highest and lowest sensitivity labels in the system: `ADMIN_HIGH` and `ADMIN_LOW`.
- `VIEW_EXTERNAL`—Demote an `ADMIN_HIGH` sensitivity label to the next highest label, and promote an `ADMIN_LOW` label to the lowest label defined in `label_encodings`(4).

Note – The label view process attribute described in "Get and Set Process Security Attribute Flags" on page 50 contains the status of the label view.

CMW Labels

The text output form for CMW labels is as follows:

```
[SENSITIVITY LABEL]
```

This example initializes a CMW label to `ADMIN_LOW [ADMIN_HIGH]` and prints out the internal and external views. The process runs at `ADMIN_HIGH` and does not need privileges to translate the `ADMIN_LOW [ADMIN_HIGH]` label.

```
#include <tsol/label.h>

main()
{
    int retval, length = 0;
    char *string1 = (char *)0, *string2 = (char *)0;
    bclabel_t cmwlabel;

    bclhigh(&cmwlabel);
    retval = bcltos(&cmwlabel, &string1, length, VIEW_INTERNAL);
    printf("View Internal = %s\n", string1);
    retval = bcltos(&cmwlabel, &string2, length, VIEW_EXTERNAL);
    printf("View External = %s\n", string2);
}
```

The `printf` statements print the following:

```
View Internal = ADMIN_LOW [ADMIN_HIGH]
View External = UNCLASSIFIED [TS A B SA SB CC]
```

Note – Although `bclhigh()` and the other functions in the `bclmanifest()` family allow you to manipulate to manipulate the value of the information label of the CMW label, you cannot set this value on an object. The information label for all objects is `ADMIN_LOW` by default.

Sensitivity and Information Labels

The text forms of sensitivity labels and information labels output by interfaces are separated by spaces and formatted as follows where the curly brackets indicate optional items and the ellipses indicate repeated words. In a sensitivity label, words represent compartments, and in an information label words represent compartments and markings.

```
CLASSIFICATION {WORD}...
```

The following code example translates a binary sensitivity label to text using different flags. The process runs at `TS A B` and needs the `sys_trans_label` privilege for the translation after the call to `bslhigh`(3TSOL). The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

```
#include <tsol/label.h>
main()
{
    int retval, length = 0;
    char *string1 = (char *)0, *string2 = (char *)0,
        *string3 = (char *)0, *string4 = (char *)0,
        *string5 = (char *)0, *string6 = (char *)0,
        *string7 = (char *)0;
    bclabel_t cmwlabel;
    bslabel_t senslabel;

    retval = getcmwplabel(&cmwlabel);
    getcsl(&senslabel, &cmwlabel);

    retval = bsltos(&senslabel, &string1, length, LONG_WORDS);
    printf("Retval1 = %d Long Words = %s\n", retval, string1);

    retval = bsltos(&senslabel, &string2, length, SHORT_WORDS);
    printf("Retval2 = %d Short Words = %s\n", retval, string2);
    retval = bsltos(&senslabel, &string3, length, LONG_CLASSIFICATION);
    printf("Retval3 = %d Long Classifications = %s\n", retval, string3);

    retval = bsltos(&senslabel, &string4, length, SHORT_CLASSIFICATION);
    printf("Retval4 = %d Short Classifications = %s\n", retval, string4);

    retval = bsltos(&senslabel, &string5, length, NO_CLASSIFICATION);
    printf("Retval5 = %d No Classification = %s\n", retval, string5);

    bslhigh(&senslabel);
/* Turn sys_trans_label on in the effective set */
    retval = bsltos(&senslabel, &string6, length, VIEW_INTERNAL);
/* sys_trans_label off. */
    printf("Retval6 = %d View Internal = %s\n", retval, string6);

    retval = bsltos(&senslabel, &string7, length, VIEW_EXTERNAL);
    printf("Retval7 = %d View External = %s\n", retval, string7);
}
```

The `printf` statements print the following.

```
Long Words = TS A B
Short Words = TS A B
Long Classifications = TOP SECRET A B
Short Classifications = TS A B
No Classification = A B
View Internal = ADMIN_HIGH
View External = TS A B SA SB CC
```

Text to Binary and Hexadecimal Label Translation Routines

This example translates text strings to a binary CMW label or sensitivity label using the following flag values:

- `NEW_LABEL` – Create a new label and correct the string as much as possible so the binary label is a complete and valid label for the system as defined in `label_encodings`(4). If the correction cannot be made, an error is returned. The string can be a text or hexadecimal string.
- `NO_CORRECTION` – Create a new label, but do not correct the construction of the string. If the string is not a complete and valid label for the system, an error is returned. The string can be a text hexadecimal string.

CMW Labels

Text CMW labels are accepted in the following form: `[sensitivity_label]`.

Sensitivity and Information Labels

Text sensitivity and information labels are accepted in the following forms. Input items can be separated by white space, commas, or slashes (/). Short and long forms of classification names and words are interchangeable.

```
{+} {classification} {{+|-}{word}...
```

- The vertical bar (|) indicates a choice between two items. Leading and trailing white space is ignored.
- The plus and minus signs can be used to modify an existing label to turn on or off the compartments and markings associated with the words.
- Curly braces indicate optional items and ellipses indicate repeated words. In a sensitivity label, the words represent compartments.

Code Examples

This example translates text strings to a binary CMW label and sensitivity label and back again using the `NEW_LABEL` flag. An example of translating a sensitivity label to a specified length (clipping) is also given. If the process runs at `[TS A B]` or higher, the `sys_trans_label` privilege is not needed for the label translations.

```
#include <tsol/label.h>

main()
{
    int retval, error, length = 0;
    char *cmwstring =”SECRET A B [TOP SECRET A B]”;
    char *sensstring = "TOP SECRET A B";
    char *string1 = (char *)0, *string2 = (char *)0,
        *string3 = (char *)0;
    bclabel_t cmwlabel;
    bslabel_t senslabel;


    retval = stobcl(cmwstring, &cmwlabel, NEW_LABEL, &error);
    retval = bcltos(&cmwlabel, &string1, length, ALL_ENTRIES);
    retval = stobsl(sensstring, &senslabel, NEW_LABEL, &error);
    retval = bsltos(&senslabel, &string2, length, ALL_ENTRIES);
    string3 = sbsltos(&senslabel, 4);

    printf(“CMW label = %s\nSens label = %s\nClipped label = %s\n’’,
        string1, string2, string3);
}
```

The `printf` statement prints the following. In the clipped label, the arrow <-indicates the sensitivity label name has clipped letters.

```
CMW label = [TS A B]
Sens label = TS A B
```

```
Clipped label = TS<-
```

Binary and Hexadecimal Label Translation

There are two types of binary to hexadecimal routines: regular and reentrant. Both types of routines return a pointer to a string that contains the result of the translation or `NULL` if the label being translated is not a binary label.

- Binary labels – Classifications are stored as integer values and compartments are stored as bit vectors of 0's and 1's.
- Hexadecimal labels – The text representation of the hexadecimal number that represents the same bit pattern as the corresponding binary label.

Binary and Hexadecimal Label Translation Routines

This example converts a binary CMW label to hexadecimal and back again. Converting a sensitivity label is similar.

```
#include <tsol/label.h>
#include <stdio.h>

main()
{
    int retval;
    bclabel_t hcmwlabel, hexcmw;
    char *string;

    getcmwplabel(&hcmwlabel);
    if((string = bcltoh(&hcmwlabel)) != NULL)
        printf("Hex string = %s\n", string);

    retval = htobcl(string, &hexcmw);
    printf("Return Value = %d\n", retval);
}
```

The first `printf` statements print the binary CMW label in the following hexadecimal format:

```
Hex string = ADMIN_LOW [0xsensitivity label value]
```

The second `printf` statement prints the following where non-zero indicates a successful translation:

```
Return Value = 1
```

Reentrant Binary and Hexadecimal Label Translation Routines

The reentrant (`MT-SAFE`) routine `bcltoh_r`(3TSOL) requires the allocation and freeing of memory for a variable of the specified type. This example allocates memory, translates the binary CMW label to hexadecimal, and frees the memory at the end. Converting a sensitivity label to hexadecimal and back is a similar process.

```
#include <tsol/label.h>
#include <stdio.h>

main()
{
    int retval;
    bclabel_t hcmwlabel, hexcmw;
    char *string, *hex;

    getcmwplabel(&hcmwlabel);
    hex = h_alloc(SUN_CMW_ID);
    if((string = bcltoh_r(&hcmwlabel, hex)) != NULL)
        printf("Hex string = %s\n", string);

    retval = htobcl(string, &hexcmw);
    printf("Return Value = %d\n", retval);
    h_free(hex);
}
```

The `printf` statement prints the binary clearance in the following hexadecimal format:

```
Hex string =0x00000000000000000000000000000000000000000000
00000000000000000000000000000000000000000000000000000000000
0000000000000000000000000000[0x00040c0000000000000000000000
000000000000000000000003ffffffffffff0000]

Return Value = 1
```

Printer Banner Information

The `bcltobanner`(3TSOL) routine translates a binary CMW label into text coded labels and strings to appear on the printer banner page, trailer page, and document pages of print jobs. The labels and strings are computed from information in the `label_encodings`(4) file. This routine is used internally by the Trusted Solaris print system, and for most applications, this translation is unnecessary. However, it can be used in a print server application or in an application that needs the external character string representation used by the print system.

In this example, the CMW label is `ADMIN_LOW [TS]`. The first five fields of *banner_fields* are character pointers. If you preallocate memory for the character pointers, the second five fields contain short integer values indicating the length of the memory allocated. If you initialize the first five character pointers to `(char *)0` as in the example, the short integer fields do not need initialization.

```
#include <tsol/label.h>
main()
{
    int retval;
    bclabel_t cmwlabel;

    static struct banner_fields banner = {(char *)0, (char *)0, (char *)0,
        (char *)0, (char *)0};

    getcmwplabel(&cmwlabel);

    retval = bcltobanner(&cmwlabel, &banner, SHORT_WORDS);

    printf("Top and bottom banner/trailer header = %s\n", banner.header);
    printf("Protect as section of banner page = %s\n", banner.protect_as);
    printf("Inf. label/top and bottom body pages = %s\n", banner.ilabel);
    printf("Caveats section of printer banner page = %s\n", banner.caveats);
    printf("Handling channels section of banner page = %s\n",
        banner.channels);
}
```

The text in the `printf` statement indicates where on the banner, trailer, and document pages the various strings appear. The caveats string is empty because no caveats are provided in the printer banner section of the `label_encodings`(4) file. See *Trusted Solaris Label Administration* and *Compartmented Mode Workstation Labeling: Encodings Format* for information on how the strings are computed.

```
Top and bottom banner/trailer header = TOP SECRET
Protect as section of banner page = TOP SECRET A B
Inf. label/top and bottom body pages = UNCLASSIFIED
Caveats section of printer banner page =
Handling channels section of banner page = HANDLE
         VIA (CH B)/(CH A) CHANNELS JOINTLY
```

CHAPTER 6

Process Clearance

This chapter describes the programming interfaces for getting and managing the process clearance. The interfaces for reading user clearance information in the `user_attr` database are described in Chapter 9.

Use of Process Clearance

When an application starts from the workspace, the user's session clearance is set on the process and called the process clearance. If the application `forks` a process, the new process's clearance is set to the calling process's clearance. If the application `exec's` a program the new program's clearance is set to the calling process's clearance.

The session clearance is selected at login. It sets the least upper bound at which the user can work during that login session and is dominated by the user clearance. The user clearance is assigned by the system administrator and determines the highest sensitivity label at which the user can work during any login session.

When users start applications from the workspace, the process CMW label is set from the values in the workspace CMW label. Because the process gets the session clearance and the workspace CMW label, the process clearance is always greater than or equal to the sensitivity label portion of the process CMW label. There is no privilege to change this rule.

A clearance has a classification and set of one or more compartments like the sensitivity label portion of a CMW label. A clearance is not a sensitivity label, but used in addition to the process's sensitivity label in the following ways.

- When a process changes its sensitivity label, the process clearance determines the highest level to which the sensitivity label can be changed. A process cannot make its sensitivity label higher than its clearance. There is no privilege to change this rule.
- When a process writes to an object at a higher sensitivity label (write-up), the process clearance determines the highest level to which the process may write up. A process cannot write above its own clearance.

Privileged Operations

The process needs `proc_setclr` privilege in its effective set to change its process clearance so it is not *equal* to its current clearance.

The process needs the `sys_trans_label` privilege in its effective set to translate a binary clearance to text when the process sensitivity label does not dominate the clearance to be translated. This privilege is also needed to check if a clearance is valid when the process sensitivity label does not dominate the clearance.

Data Types, Header Files, and Libraries

To use the programming interfaces described in this chapter, you need the following header file.

```
#include <tsol/label.h>
```

The examples in this chapter compile with the following library:

```
-ltsol
```

Process Clearances

Interfaces accept as parameters and return binary process clearances in a variable of type `bclear_t`.

Binary Levels

A level is a classification and a set of compartments in a sensitivity label or clearance. Interfaces accept as parameters and return binary levels in a structure of type `blevel_t`.

Type Compatibility

Any variable of type `bclear_t` or `bslabel_t` can be passed to a function that accepts a parameter of type `blevel_t`.

Programming Interface Declarations

The following programming interfaces are available for managing process clearances.

System Calls

These system calls get and set the clearance of the calling process. Refer to the `getclearance`(2) and `setclearance`(2) man pages.

Caution – Every process that sets a clearance is responsible for setting a valid clearance as specified in the `label_encodings`(4) file, and must pass the correct binary form of the clearance. The text to binary translation functions correct the clearance as much as possible to ensure a correct binary clearance results from the translation. However, you might use the `bclearvalid`(3TSOL) routine to check that the clearance is valid.

```
int    getclearance(bclear_t *clearance);
int    setclearance(bclear_t *clearance);
```

Library Routines

Library routines are available to initialize, compare, translate and verify the process clearance.

Initialization

These routines initialize a clearance to ADMIN_HIGH, ADMIN_LOW, or undefined (similar to NULL). Refer to the blmanifest(3TSOL) man page.

```
void  bclearhigh(bclear_t *clearance);
void  bclearlow(bclear_t *clearance);
void  bclearundef(bclear_t *clearance);
```

Comparisons

These routines compare two levels to see if *level1* equals, dominates, or strictly dominates *level2*. A level is a classification and set of compartments in a sensitivity label or clearance.

A returned non-zero is true and 0 is false. Refer to the blcompare(3TSOL) man page.

```
int  blequal(const blevel_t *level1, const blevel_t *level2);
int  bldominates(const blevel_t *level1, const blevel_t *level2);
int  blstrictdom(const blevel_t *level1, const blevel_t *level2);
int  blinrange(const blevel_t *level, const brange_t *range);
```

Clearance Type

The bltype(3TSOL) routine checks the clearance type, and the setbltype(3TSOL) routine sets the clearance type. A clearance can be defined or undefined. Refer to the bltype(3TSOL) man page.

```
int  bltype(const void *clearance, const unsigned char type);
void  setbltype(void *clearance, const unsigned char type);
```

Level Bounds

These routines compare two levels to find the sensitivity level that represents the greatest lower bound (blminimum(3TSOL)) or least upper bound (blmaximum(3TSOL)) of the range bounded by the two levels. A level is a classification and set of compartments in a sensitivity label or clearance. Refer to the blminmax(3TSOL) man page.

```
void  blmaximum(blevel_t *maximum_label,
                const blevel_t *bounding_label);
void  blminimum(blevel_t *minimum_label,
                const blevel_t *bounding_label);
```

Valid Clearance

This routine tests whether the specified clearance is valid for the system. Refer to the `blvalid`(3TSOL) man page.

```
int  bclearvalid(const bclear_t *clearance);
```

Binary and Text Translation

These routines translate a clearance from binary to text and back again. Refer to the `stobl`(3TSOL) man page.

Note – See Chapter 14 for Interfaces that translate binary labels to text and clip the final label according to the specified width and font list for display in Motif-based graphical user interfaces (GUIs).

```
int  bcleartos(const bclear_t *clearance,
        char **string,
        const int len,
        const int flags);

int  stobclear(const char *string,
        bclear_t *clearance,
        const int flags, int *error);

char*  sbcleartos(const bclear_t *clearance,
        const int len);
```

Binary and Hexadecimal Translation

These routines translate a clearance from binary to hexadecimal and back again. Refer to the `btohex`(3TSOL) man page.

```
char   *h_alloc(const unsigned char id);
void   h_free(char *hex);

char   *bcleartoh_r(const bclear_t *clearance, char *hex);
char   *bcleartoh(const bclear_t *clearance);
int    htobclear(const char *s, bclear_t *clearance);
```

Process Clearance Operations

A program must get its process clearance before it can perform an operation on the clearance. This short program gets the process clearance of the calling process.

```
#include <tsol/label.h>

main()
{
    int          retval;
    bclear_t     pclear;

    retval = getclearance(&pclear);
    printf("Retval = %d\n", retval);
}
```

The `printf` statement prints the following:

```
Retval = 0
```

Set Process Clearance

The process needs the `proc_setclr` privilege to set the process clearance to another value if the new value is not equal to the sensitivity label portion of the process's own CMW label. A new process clearance is set with the `setclearance`(2) system call. This example initializes a clearance structure to `ADMIN_HIGH` and passes it to the `setclearance`(2) system call.

```
#include <tsol/label.h>

main()
{
    int          retval;
    bclear_t     hiclear, undef, loclear;

    bclearhigh(&hiclear);

/* Turn proc_setclr on in the effective set */
    retval = setclearance(&hiclear);
/* Turn off the proc_setclr privilege */

    printf("Retval = %d\n", retval);
}
```

The `printf`(1) statement prints the following:

```
Retval = 0
```

Initialize Clearance Structure

A clearance can be initialized to ADMIN_LOW or ADMIN_HIGH and have its type checked. This example initializes *undef* to undefined (similar to NULL) and *loclear* to ADMIN_LOW. It then checks the type on *loclear*, sets the type to undefined, and checks it again. A clearance is undefined when its ID field is initialized to SUN_CLR_UN. An undefined clearance is invalid. A clearance is defined when the ID field in the label structure is initialized to SUN_CLR_ID.

```
#include <tsol/label.h>

main()
{
    int          retval;
    bclear_t     loclear, undef;

    bclearlow(&loclear);
    bclearundef(&undef);

    retval = bltype(&loclear, SUN_CLR_ID);
    printf("Is clearance defined? %d\n", retval);

    setbltype(&loclear, SUN_CLR_UN);
    retval = bltype(&loclear, SUN_CLR_ID);
    printf("Is clearance defined? %d\n", retval);
}
```

The printf(1) statement prints the following where non-zero is True and 0 is False.

```
Is clearance defined? 1
Is clearance defined? 0
```

Find Relationships Between Two Levels

A level is a classification and set of compartments for a sensitivity label, information label, or clearance; and is represented by the blevel_t data type. Two levels can be equal, one can dominate the other, or one can strictly dominate the other.

- Equal – One level is equal to another when its classification is arithmetically equal to the other's classification (by means of its place in the classifications hierarchy), and its compartments contain all the other's compartments and no additional compartments.
- Dominates – One level dominates another when its classification is arithmetically greater than or equal to the other's (by means of its place in the classifications hierarchy), and its compartments contain all the other's compartments.
- Strictly dominates – Level one is said to strictly dominate level two when level one dominates level two, but is not equal to level two.

This example checks the process clearance against the sensitivity label portion of a file CMW label to find their relationship (equal, dominate, or strictly dominate). The process clearance is `TOP SECRET A B`, the sensitivity label portion of the file CMW label is Confidential.

```
#include <tsol/label.h>

main()
{
    int         retval;
    bclear_t    pclear;
    bclabel_t   cmwlabel;
    bslabel_t   senslabel;

    retval = getclearance(&pclear);
    retval = getcmwlabel("/export/home/zelda/afile", &cmwlabel);
    getcsl(&senslabel, &cmwlabel);

    retval = blequal(&pclear, &senslabel);
    printf("Clearance equals sensitivity label? %d\n", retval);

    retval = bldominates(&pclear, &senslabel);
    printf("Clearance dominates sensitivity label? %d\n", retval);

    retval = blstrictdom(&pclear, &senslabel);
    printf("Clearance strictly dominates sensitivity label? %d\n", retval);
}
```

The `printf`(1) statements print the following. Non-zero is True and 0 is False:

```
Clearance equals sensitivity label? 0
Clearance dominates sensitivity label? 1
Clearance strictly dominates sensitivity label? 1
```

Find Greatest Level and Lowest Level

The next example finds the greatest and lowest values between two variables of type `blevel_t`. These interfaces let you compare two levels to find the level that represents the greatest lower bound (with the `blminimum`(3TSOL) routine) or least upper bound (with the `blmaximum`(3TSOL) routine) bounded by the two levels. A level can be a sensitivity label or clearance.

The example code finds the greatest lower bound and least upper bound of the range created by a process clearance of `TS A B` and a sensitivity label of `ADMIN_LOW`. The process runs at Confidential.

The first part of the example finds the greater of the classifications and the greater of all the compartments of the two levels and puts that value into the first parameter. This operation is called finding the least upper bound because it finds the lowest level that dominates both original parameter values passed.

The process sensitivity level does not dominate the process clearance so the process needs the sys_trans_label privilege for the translation. The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

```
#include <tsol/label.h>
#include <tsol/priv.h>
main()
{
    int        retval, length = 0;
    char       *string = (char *)0, *string1 = (char *)0;
    bclear_t   clear;
    bslabel_t  senslabel;
    bsllow(&senslabel;);
    retval = getclearance(&clear;);
    blmaximum(&senslabel;, &clear;);
    /* Turn the sys_trans_label privilege on in the effective set */
    set_effective_priv(PRIV_ON, 1, PRIV_SYS_TRANS_LABEL);
    retval = bsltos(&senslabel;, &string;, length, LONG_WORDS);
    printf("Maximum = %s\n", string);
```

The printf statements print the following where TS ABLE BAKER is the lowest level that dominates TS A B and ADMIN_LOW.

```
Maximum = TS A B
```

The second part of the example finds the lower of the classifications and only those compartments contained in both parameters, and puts that value in the first parameter. This operation finds the greatest lower bound because it finds the greatest level dominated by both original parameter values passed.

```
bsllow(&senslabel;);
blminimum(&senslabel;, &clear;);
retval = bsltos(&senslabel;, &string1;, length, LONG_WORDS);
printf("Minimum = %s\n", string1);
/* Turn sys_trans_label off */
set_effective_priv(PRIV_OFF, 1, PRIV_SYS_TRANS_LABEL);

}
```

The printf statements print the following where ADMIN_LOW is the highest level that is dominated by TS A B and ADMIN_LOW.

```
Minimum = ADMIN_LOW
```

Valid Clearance

A valid clearance is a clearance defined in the label_encodings(4) file. Call the bclearvalid(3TSOL) routine to check if a clearance is valid. The process running at TS A B equals the clearance and needs no privilege for this operation.

```
#include <tsol/label.h>
main()
{
    int         retval, error;
    bclear_t    bclear;
    char        *string = "TS ABLE BAKER";

    retval = stobclear(string, &bclear, NEW_LABEL, &error);
    retval = bclearvalid(&bclear);
    printf("Return value = %d\n", retval);
}
```

The `printf` statement prints the following where 1 means the clearance is valid; -1 means the `label_encodings` file is inaccessible; and 0 means the label is not valid or the process sensitivity label does not dominate the clearance and the `sys_trans_label` privilege is not effective:

```
Return value = 1
```

Translating Process Clearances

Clearances (like labels) can be represented in binary, text, or hexadecimal. Within the kernel all clearances are stored in binary form, and binary is the form used for clearances passed to and received from programming interfaces.

- Binary Clearances – Classifications are stored as an integer and compartments are stored as bit vectors using 0's and 1's.
- Text Clearance – Human-readable clearances that display classifications, and compartments using the names defined in the `label_encodings`(4) file.
- Hexadecimal Clearances – The text representation of the hexadecimal number that represents the same bit pattern as the corresponding binary clearance. The clearance has text characters but does not reveal the classification or compartment names. A process can store a clearance in text when it will be read by processes at arbitrary clearances.

Binary and Text

This example translate a binary clearance to text using long words. The process running at `TS A B` equals the clearance and needs no privilege.

Note – The text input and output formats, rules, and flags are presented in "Binary and Text Label Translation" on page 113.

```
#include <tsol/label.h>
main()
{
    int          retval, length = 0;
    bclear_t     pclear;
    char         *string = (char *)0;

    retval = getclearance(&pclear);
    retval = bcleartos(&pclear, &string, length, LONG_WORDS);
    printf("Process clearance = %s\n", string);
}
```

The `printf`(1) statement prints the following:

```
Process clearance = TS ABLE BAKER
```

This example clips the process label to five characters. The clipping occurs when the number of characters in *pclear* is greater than the specified length.

```
#include <tsol/label.h>

main()
{
    int          retval;
    bclear_t     pclear;
    char         *string = (char *)0;

    retval = getclearance(&pclear);
    string = sbcleartos(&pclear, 5);
    printf("Clipped process clearance = %s\n", string);
}
```

The `printf` statement prints the following. The left arrow is a clipped indicator to show the name has been clipped. The number of characters to which the name is clipped includes two characters for the clipped indicator.

```
Clipped process clearance = TS<-
```

This example translates a text string to a binary clearance.

```
#include <tsol/label.h>

main()
{
    int          retval, error;
    bclear_t     bclear;
    char         *labelstring = "TS ABLE BAKER";

    retval = stobclear(labelstring, &bclear, NEW_LABEL, &error);
    if (retval == 0)
        printf("Error = %d\n", error);
    else
        printf("Retval = %d\n", retval);
}
```

The `printf`(1) statement prints the following:

```
Retval = 1
```

Binary and Hexadecimal

There are two types of binary to hexadecimal routines: regular and reentrant. Both types of routines return a pointer to a string that contains the result of the translation or `NULL` if the clearance passed in is not type `bclear_t`.

Regular

This example translates the binary process clearance to hexadecimal and back.

```
#include <tsol/label.h>

main()
{
    int          retval;
    bclear_t     hclear;
    char         *string ;

    retval = getclearance(&hclear);

    if((string = bcleartoh(&hclear)) != 0)
        printf("Hex string = %s\n", string);

    retval = htobclear(string, &hclear);
    printf("Return Value = %d\n", retval);
}
```

The first `printf` statement prints the binary clearance in the following hexadecimal format:

```
Hex string = 0xClearance hexadecimal value
```

The second `printf` statement prints the following where non-zero indicates a successful translation:

```
Return Value = 1
```

Reentrant

The reentrant (MT-SAFE) routine `bcleartoh_r`(3TSOL) requires the allocation and freeing of memory for the value returned. The `h_alloc`(3TSOL) routine is used to allocate this memory, sizing it appropriately for the type of label (in this case hexadecimal) to be converted.

type where *type* is a hexadecimal value that indicates that a defined clearance (`SUN_CLR_ID`) is translated to hexadecimal.

This example allocates memory for the translation type, translates the binary process clearance to hexadecimal, and frees the memory at the end.

```
#include <tsol/label.h>

main()
{
    bclear_t     hclear;
    char         *string, *hex;

    getclearance(&hclear);
    hex = h_alloc(SUN_CLR_ID);
    if((string = bcleartoh_r(&hclear, hex)) != 0);
        printf("Hex string = %s\n", string);

    h_free(hex);
}
```

The `printf` statement prints the binary clearance in the following hexadecimal format:

```
Hex string = 0x0006cc0000000000000000000000000000000000
000000003ffffffffffff0000
```

CHAPTER 7

Multilevel Directories

The Trusted Solaris environment supports regular UNIX directories and multilevel directories (MLDs). MLDs enable a program that runs at different sensitivity labels to use a common directory and access files at the sensitivity label at which the program is currently running. An MLD contains only single-level directories (SLDs), and each SLD stores files at the sensitivity label of the SLD. Within one MLD, several files with the same name can be stored in different SLDs. Each instance of the same file contains data appropriate to the sensitivity label of the SLD where it is stored. This is called polyinstantiation of directories and files.

Directory Structure

The `tmp` directory and all home directories are automatically MLDs at `ADMIN_LOW` when set up for users in the User Manager by the system administrator. Additionally, `mkdir`(1) has an option for creating an MLD. Figure 7–1 shows the directory structure of Zelda's home directory where the MLD is `ADMIN_LOW` with three SLDs at Top Secret, Secret, and Confidential.

- An MLD cannot contain another MLD.
- An SLD cannot contain an MLD or an SLD.
- An SLD can contain regular UNIX directories and all types of files.

SLDs are created as needed during pathname lookup, and by the `getsldname`(2) and `fgetsldname`(2) system calls. The SLD sensitivity label is always a valid sensitivity label for the system.

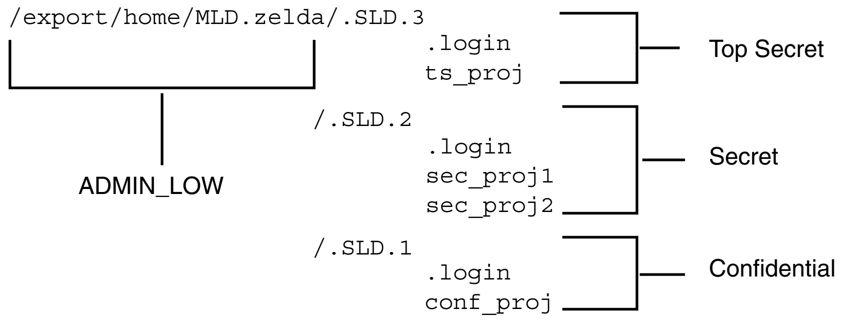

FIGURE 7–1 Multilevel Directories

An application running at Secret dominates the `ADMIN_LOW` directory path `/home/export/.MLD.zelda`, dominates the SLDs at Secret and Confidential, but does not dominate the SLD at Top Secret. Without privilege and with discretionary access, a process running at Secret has the following access:

- Read, Write, and Create access to the Secret SLD.
- The ability to read down to the Confidential SLD using the fully adorned name `/export/home/.MLD.zelda/.SLD.1`. See "Adorned Names" on page 137 and "Using Path Names with Adornments" on page 144.
- The ability to write up to the Top Secret SLD using the fully adorned name `/export/home/.MLD.zelda/.SLD.3` if the process clearance dominates the Top Secret SLD. See "Adorned Names" on page 137 and "Using Path Names with Adornments" on page 144.

A process running at Confidential would have access to the following files assuming the directory structure in Figure 7–1.

```
.login
conf_proj
```

A process running at Secret would have access to the following files assuming the directory structure in Figure 7–1.

```
.login
secret_proj1
secret_proj2
```

A process running at Top Secret would have access to the following files assuming the directory structure in Figure 7–1.

```
.login
ts_proj
```

Temporary Directory

Many applications create files in the `/tmp` directory. If `/tmp` is a regular UNIX directory at some sensitivity label, unprivileged processes running at other sensitivity labels cannot create files in `/tmp`. The Trusted Solaris environment makes `/tmp` an MLD so applications can create files in the SLD that corresponds to the sensitivity label of the process.

Symbolic Links

Symbolic links can be used in combination with MLDs. For example, a symbolic link whose target path name is in an MLD points to a different target file at each sensitivity label. Symbolic links in an SLD can point to a target path name in a regular directory to have a path name in an MLD refer to the same file when referenced at different sensitivity labels.

Adorned Names

When a process refers to an MLD in a pathname, the system transparently extends the reference to include the SLD that corresponds to the process sensitivity label. This operation is called pathname translation. If a process running at Confidential references `/export/home/zelda`, it accesses the SLD in `/export/home/zelda` at Confidential. Because pathname translation is transparent, the process does not explicitly reference the SLD.

All MLDs have an adornment. The adornment is `.MLD.` unless it was changed by the system administrator. The adornment lets a process refer directly to the MLD rather than transparently to the SLD that has the same sensitivity label as the process. A process would use the `ls`(1) command to reference the adorned name to do the following.

- List the SLDs within an MLD. Without the adornment, the contents of the SLD with the same he sensitivity label as the process are listed instead.

```
% ls /.MLD.tmp
```

- Refer explicitly to an SLD by using the adorned MLD name.

```
% ls /.MLD.tmp/.SLD.3
```

Privileged Operations

Mandatory and discretionary access is required to get information on an MLD or SLD, and to access objects within an SLD with the fully adorned path name.

When considering the mandatory and discretionary access rules presented in Chapter 1, the SLD is a component in the path name leading to the final file system object. The calling process needs mandatory and discretionary search access to the SLD and the appropriate access to the final object. Privileges may be required if access is denied.

To get the SLD name for a specified sensitivity label within an MLD, the calling process needs the following privileges in the following situations:

- The calling process needs the `file_upgrade_sl` privilege in its effective set if the process sensitivity label is strictly dominated by the SLD sensitivity label.
- The calling process needs the `file_downgrade_sl` privilege in its effective set if the SLD sensitivity label dominates the process's sensitivity label.

Data Types, Header Files, and Libraries

To use the programming interfaces described in this chapter, you need the following header file.

```
#include <tsol/mld.h>
```

The examples in this chapter compile with the following library:

```
-ltsol
```

Sensitivity Label

The bslabel_t type definition represents the sensitivity label portion of a binary CMW label. The `getsldname`(2) system call accepts a variable of type `bslabel_t`.

Status

The stat structure contains information on a specified MLD, SLD, or symbolic link. The structure is returned by the mldstat(3TSOL) and mldlstat(3TSOL) system calls.

Type	Field	Description	Default
mode_t	st_mode	File type and permissions.	0
nlink_t	st_nlink	Number of hard links.	1
uid_t	st_uid	User ID of owner.	0
gid_t	st_gid	Group Id of owner.	0
time_t	st_atime	Last access time in seconds.	Current time
time_t	st_mtime	Last modify time in seconds.	Current time
time_t	st_ctime	Last inode change time in seconds.	Current time

Programming Interface Declarations

The following programming interfaces are available for getting information on MLDs and SLDs.

System Calls

System calls are available to get the SLD name, get MLD adornment, and get SLD or MLD file attribute information.

Get SLD Name

The getsldname(2) system call gets the SLD name for *path_name* at the specified *slabel*. Refer to the getsldname(2) man page. The fgetsldname(2) system call uses a file descriptor.

```
int getsldname(const char *path_name,
    const bslabel_t *slabel,
    char *name_buf,
    const int len);
```

```
int fgetsldname(const int fd,
    const bslabel_t *slabel_p,
    char *name_buf,
    const int len);
```

Get MLD Adornment

The `getmldadorn`(2) system call gets the fully adorned path name for *path_name*. The `fgetmldadorn`(2) system call uses a file descriptor. Refer to the `getmldadorn`(2) man page.

```
int getmldadorn(const char *path_name, char *adorn_buf[MLD_ADORN_MAX]);
int fgetmldadorn(const int fd, char adorn_buf[MLD_ADORN_MAX]);
```

Get Attribute Information for SLD or MLD

The `mldstat`(3TSOL) system call returns file attribute information on the MLD specified by *path_name*. The `mldlstat`(3TSOL) system call returns information on the MLD symbolic link.

```
int mldstat(const char *path_name, struct stat *stat_buf);
int mldlstat(const char *path_name, struct stat *stat_buf);
```

Get MLD Attribute Flags

These system calls are described in "Get and Set File System Security Attribute Flags" on page 49 in Chapter 2. Also, refer to the `getfattrflag`(2) man page.

```
int mldgetfattrflag(const char *path, secflgs_t *flags);
int mldsetfattrflag(const char * path, secflgs_t which, secflgs_t flags);
```

Library Routines

Library routines are available to get the pathname of the current working directory and display a pathname with adornments.

Get Current Working Directory

This routine gets the fully adorned path name for the current working directory. Refer to the `mldgetcwd`(3TSOL) man page.

```
char* mldgetcwd(char *buf, size_t size);
```

Get Adorned Name

This routine gets the adorned name for the MLD specified in *path_name*. Refer to the `adornfc`(3TSOL) man page.

```
int adornfc(const char *path_name, char *adorned_name);
```

Find the Real Path Name

These routines take the path name supplied in *path_name*, expand all symbolic links, resolve dot references to the current directory and dot-dot references to the parent directory, remove extra slash characters, add the correct MLD and SLD adornments, and store the final result in *resolved_path*. The result is for the SLD at which the process is running, or at the specified SLD. Refer to the `mldrealpath`(3TSOL) man page.

```
char* mldrealpath(const char *path_name,
    char *resolved_path);

char *mldrealpathl(const char *path_name,
    char *resolved_path, const bslabel_t *senslabel);
```

Query MLD and SLD Name

The following code queries the MLD adornment with the `getmldadorn`(2) system call and queries the SLD name for the Top Secret SLD with the `getsldname`(2) system call. In this example, the Top Secret SLD does not already exist, so the call to `getsldname`(2) will create it.

The process is running at Confidential with a clearance of Top Secret. The process needs the `sys_trans_label` privilege to translate the Top Secret label, the `file_upgrade_sl` privilege to create the Top Secret SLD, and the `file_mac_search` and `file_mac_read` privileges to access the Top Secret SLD information.

```
#include <tsol/label.h>

main()
{
    int retval, error, length;
    bslabel_t label;
    char *buffer[1025], *buf[1025], *string = "TOP SECRET";
    char *file = “/export/home/zelda”;

    retval = getmldadorn(file, buffer);
    printf("MLD adornment = %s\n", buffer);
/* Turn sys_trans_label on in the effective set */
    retval = stobsl(string, &label, NEW_LABEL, &error);
/* Turn sys_trans_label off */

    length = sizeof(buf);

/* Turn file_upgrade_sl, file_mac_search, and file_mac_read on */
    retval = getsldname(file, &label, buf, length);
/* Turn file_upgrade_sl, file_mac_search, and file_mac_read off*/

    printf("SLD name = %s\n", buf);
}
```

The `printf`(1) statements print the following:

```
MLD adornment = .MLD.
SLD name = .SLD.3
```

This example queries the current working directory (MLD plus current SLD) with the `mldgetcwd`(3TSOL) routine, gets the adorned name for the MLD with the `adornfc`(1) routine, and finds the real path with the `mldrealpath`(1) routine by removing the extra slash in the path name stored in *resolvefile*. The process is running at Confidential.

```
#include <tsol/label.h>
#include <sys/types.h>

main()
{
    int retval;
    char *buffer[1025];
    char *file = "/export/home/zelda";
    char *string2, *name[1025], *string3, *resolved[1025];
    size_t size;

/* Character string with errors to be resolved */
    char *resolvefile = "./";

    size = sizeof(buffer);
    string2 = (char *)mldgetcwd(buffer, size);
    printf("Current working directory = %s\n", buffer);

    retval = adornfc(file, name);
    printf("Adorned name = %s\n", name);

    string3 = (char *)mldrealpath(resolvefile, resolved);
    printf("Real path = %s\n", resolved);
}
```

The `printf` statements print the following:

Note – If the SLD name is included in the file parameter to the `adornfc`(1) routine, the adorned name is returned with the SLD appended in the form `/export/home/zelda/.MLD..SLD.1`.

```
Current working directory = /export/home/.MLD.zelda/.SLD.2
Adorned name = /export/home/.MLD.zelda
Real path = /export/home/.MLD.zelda/.SLD.2
```

This example gets attribute information for the `/export/home/zelda` MLD. In the `printf`(1) statements, the `stat`(2) system call macros test whether the MLD is a directory or regular file, and the time returned in seconds is converted to a human-readable time with the `ctime`(3C) routine.

```
#include <tsol/label.h>
#include <sys/stat.h>

main()
{
    int retval;
    struct stat statbuf;
    char *file = “/export/home/zelda”;

    retval = mldstat(file, &statbuf);

    printf("Is file system object a directory? = %d\n",
        S_ISDIR(statbuf.st_mode));

    printf("Is file system object a regular file? = %d\n",
        S_ISREG(statbuf.st_mode));

    printf("Number of links = %d\n", statbuf.st_nlink);
    printf("Owner’s user ID = %d\n", statbuf.st_uid);
    printf("Owner’s group Id = %d\n", statbuf.st_gid);
    printf("Last access time = %s\n", ctime(&statbuf.st_atime));
    printf("Last modify time = %s\n", ctime(&statbuf.st_mtime));
    printf("Last status change = %s\n", ctime(&statbuf.st_ctime));
}
```

The `printf` statements print the following:

```
Is file system object a directory? = 1
Is file system object a regular file? = 0
Number of links = 6
Owner’s user ID = 29378
Owner’s group Id = 10
Last access time = Wed May 28 10:58:25 1999
Last modify time = Wed May 28 09:39:18 1999
Last status change = Wed May 28 09:39:18 1999
```

Using Path Names with Adornments

UNIX system calls that accept a path name such as `open`(2) and `creat`(2) go to the SLD at the same sensitivity label as the process unless the fully adorned path name is passed instead of a regular path name. The fully adorned path name includes the MLD adornment and the SLD directory name as shown in the code example. Note that a process cannot create files or directories in either an MLD or SLD with the `mkdir`(1) system call.

The mandatory access and discretionary access controls described in "Security Policy" on page 33 apply.

Open a File

In this example, the process is running at Confidential with a clearance of Top Secret. The Confidential process needs the `file_mac_search` privilege in its effective set to access the SLD at Top Secret. Because the file is opened for writing and a write-up is allowed by the security policy, no other privileges are needed assuming the operation passes all discretionary access checks.

```
#include <tsol/label.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
    int filedes;

/* Open a file in the SLD at which the process is running */
    filedes = open("/export/home/zelda/afile", O_WRONLY);
    printf("File descriptor for regular path = %d\n", filedes);

/* Open a file in the Top Secret SLD */
/* Turn file_mac_search on in the effective set */
    filedes = open("/export/home/.MLD.zelda/.SLD.3/afile", O_WRONLY);
/* Turn file_mac_search off */

    printf("File descriptor for adorned path = %d\n", filedes);
}
```

The `printf` statements print the following.

```
File descriptor for regular path = 3
File descriptor for adorned path = 4
```

Create a file

In this example, the process is running at Confidential with a clearance of Top Secret. The Confidential process needs the `file_mac_search` privilege in its effective set to access the SLD at Top Secret. If `afile` does not already exist in the Top Secret SLD, the process needs the `file_mac_write` privilege because the process sensitivity label does not equal the SLD sensitivity label. If `afile` already exists, the `file_mac_write` privilege is not needed.

```
#include <tsol/label.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

main()
{
    int filedes;

/* Create a file in the SLD at which the process is running */
    filedes = creat("/export/home/zelda/afile", 660);

    printf("File descriptor for regular path = %d\n", filedes);

/* Create a file in the Top Secret SLD */
/* Turn file_mac_search on in the effective set */
    filedes = creat("/export/home/.MLD.zelda/.SLD.3/afile", 660);
/* Turn file_mac_search off */

    printf("File descriptor for adorned path = %d\n", filedes);
}
```

The `printf` statements print the following.

```
File descriptor for regular path = 3
File descriptor for adorned path = 4
```

CHAPTER 8

Application Auditing

Auditing enables administrators to monitor user actions to detect suspicious or abnormal patterns of system usage. Auditing concepts, terminology, and administration procedures are fully covered in *Trusted Solaris Audit Administration*. This chapter describes how to use the `auditwrite`(3TSOL) routine in a third-party application to create and log third-party user audit events.

Third-Party User Activities

Third-party applications audit user activities by creating third-party audit events and audit classes specific to the application and generating audit records with those events using the `auditwrite`(3TSOL) routine.

- Third-party audit event – An event created and added to the `/etc/security/audit_event` file. The `audit_event`(4) man page describes how this file stores event definitions in the numbers 32768 to 65535 and specifies audit event to audit class mappings.

- Third-party audit class – A logical grouping of audit events defined in the `/etc/security/audit_class` file (`audit_class`(4)), and used for preselection and postselection (see `audit_control`(4) man page).

The application programmer defines the third-party audit events and classes used in third-party applications, and the system administrator at the site using the application sets up the above-referenced files to recognize the new events and classes.

Within the application, audit events are generated and logged to the audit trail in records. Audit records contain tokens that provide the audit event and other relevant information such as the process ID of the process that generated the event, the machine on which the event occurred, and the date and time. The audit trail is the place where audit records generated by the kernel, system applications, and third-party applications are stored in files. The following figure presents these elements and their relationships.

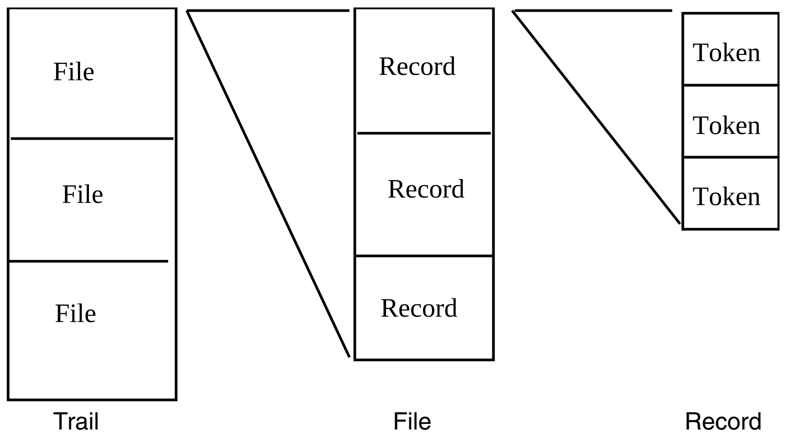

FIGURE 8–1 Audit Trail, Files, Records, and Tokens

It is up to you to decide exactly what information is logged to the audit record by deciding which tokens are passed to the `auditwrite`(3TSOL) routine. Audit records should be generated in third-party applications in the highest possible interface layer where the most precise information is available, and there is more opportunity to limit the generation of less useful audit records.

Privileged Operations

The process needs the `proc_audit_appl` privilege in its effective set to call the `auditwrite`(3TSOL) routine and log third-party audit records to the audit trail. It is also required for any operations that get auditing information such as the audit state.

Header Files and Libraries

To use the programming interface described in this chapter, you need the following header file:

```
#include <bsm/auditwrite.h>
```

The examples in this chapter compile with the following libraries:

```
-DTSOL -lbsm -lsocket -lnsl -lintl -ltsol
```

Declaration and Argument Types

The `auditwrite`(3TSOL) routine generates and logs third-party audit events.

```
int auditwrite(..., AW_END);
```

This library routine takes a variable number of arguments of the following three kinds. Refer to the `auditwrite`(3TSOL) man page for a complete listing of argument commands and their meaning. The code examples in this chapter use many of the possible argument commands.

- Control commands – control the behavior of the `auditwrite`(3TSOL) routine by, for example, directing the `auditwrite()` routine to add information to a partially built audit record (`AW_APPEND`) or send a complete audit record to the audit trail (`AW_WRITE`). The parameter list must have exactly one control command.
- Token commands – are typically specified when the control command is either `AW_WRITE` or `AW_APPEND`. Token commands describe the attributes that make up an audit record such as the event that occurred (`AW_EVENT`), a text message (`AW_TEXT`), or the path name leading to a file system object where the event occurred (`AW_PATH`). An attribute command is always followed by one or more

value parameters that supply values of the type indicated by the attribute parameter. The control command and attribute commands can appear in any order in the parameter list.

- Terminator command – `AW_END` is always positioned at the end of the parameter list to tell the `auditwrite`(3TSOL) routine to stop parsing.

Preliminary Setup for Code Examples

A certain amount of administrative setup needs to occur to create third-party events and classes, and view audit records logged to the audit trail. The following is a summary of the administrative setup required for the code examples in this chapter to work. Trusted Solaris Audit Administration explains these and other administrative procedures in detail.

First, check that auditing is enabled and turned on. It is enabled by default, but you can check with the `auditconfig`(1M) command and the `getcond` option. Run this command from the profile shell with the `sys_audit` or `proc_audit_appl` privilege. The `setcond` option turns auditing on and off.

```
phoenix% auditconfig -getcond
```

Audit File Setup

This section shows you how to set up the `audit_class`, `audit_event`, and `audit_control` files. The best way to edit these files is as follows:

1. **Assume the Security administrator role.**
2. **Launch the Application Manager.**
3. **Double click the System_Admin icon.**
4. **Double click the Audit Classes, Audit Events, or Audit Control action.**
5. **Edit each file as described in the following sections.**

Audit Classes and Audit Events

Create the third-party audit class `ec` and two audit events, `AUE_second_signature` and `AUE_second_signature_verify`. See the `audit_class`(4) and `audit_event`(4) man pages for more information on these files.

- Third-party audit classes are added to the `/etc/security/audit_class` file in the form `mask:name:description` as follows:

  ```
  0x00008000:ec:example class
  ```
- Third-party audit events are added to the `/etc/security/audit_event` file and assigned one of the numbers reserved for third-party events from 32768 to 65535. This file also contains the audit event to audit class mapping. The following lines add two events and map them to the example (`ec`) class:

  ```
  32768:AUE_second_signature:second signature requested:ec
  ```
  ```
  32769:AUE_second_signature_verify:second signature added:ec
  ```

Audit Control (Process Preselection Mask)

The process preselection mask specifies the audit classes to be audited by the process. To set up the preselection mask to audit for third-party events, edit the `/etc/security/audit_control` flag parameter as follows to audit events in the example (`ec`) class for success and failure.

```
flags:ec
```

Settings in `audit_control`(4) are global to all users in the system. To make a setting specific to a user, edit the `/etc/security/audit_user` file (the Audit Users action) as follows:

```
zelda:ec
```

See the `audit_control`(4) and `audit_user`(4) man pages for more information on these files and settings. Log out and log back in for the newly defined process preselection mask to take effect. You could also use `auditconfig`(1M) with the `-setpmask` option to set the process preselection mask on any existing processes, but it is probably easier to set one of these files and log out and log back in once.

Viewing the Audit Trail Setup

All audit records including audit records generated by the `auditwrite`(3TSOL) routine are logged to the audit trail in a series of binary files at `ADMIN_HIGH`. The location of the audit files is set in the `/etc/security/audit_control` file, and by default is `/var/audit`. The `praudit`(1M) command reads the audit trail files and interprets the binary data as human-readable audit records.

Assume a role with the `tail`(1) command and the `praudit`(1M) command with the `proc_audit_appl` and `proc_audit_tcb` privileges. Open a terminal at `ADMIN_HIGH`, change directory to where the audit records are stored, and execute the `tail` and `praudit` commands as shown to view the current audit file.

Note – This syntax works when there is only one `*not_terminated*` file. If there are others, delete the older ones before executing this command.

```
phoenix%  cd /var/audit
phoenix% tail -0f *not_terminated* | praudit
```

The audit daemon logs audit records to the audit partition until they reach their maximum capacity and then starts a new file. The file currently written to is the `not_terminated` audit file. View the `/etc/security/audit_data` file to determine which file is current.

Executable Code Setup

Put the `proc_audit_appl` privilege in the forced and allowed privilege sets of the executable file containing the example source code by executing `setfpriv`(1) from the profile shell with the `file_setpriv` privilege. "Assigning File Privileges using a Script" on page 254 explains how to do this with a script.

```
phoenix% setfpriv -s -f proc_audit_appl -a proc_audit_appl executable.file
```

Creating an Audit Record

An audit record is created by passing one control command and one or more token commands to the `auditwrite`(3TSOL) routine in one call (`AW_WRITE`) or several calls (`AW_APPEND` for each call with `AW_WRITE` in the last call). An audit record must have an `AW_EVENT token` and should have an `AW_RETURN` token to indicate which event occurred and whether the event succeeded or failed. See "Return Token" on page 155 for more information.

Making Invalid and Valid Calls

These examples show the different audit records logged to the audit trail when a call to the `auditwrite`(3TSOL) routine is invalid and valid. The structure of audit records and tokens is described in "Token Structure" on page 155.

Invalid Call

If you use more than one control command, or omit the control command, or do not include the `AW_END` terminator command, your code compiles and runs and a record is logged to the audit trail to record the invalid call to the `auditwrite`(3TSOL) routine. Note that the event is logged to the trail only if the process preselection mask audits the `AUE_auditwrite` event for failure.

This example shows an invalid `auditwrite`(3TSOL) routine call that omits the `AW_END` terminator command and the resulting audit record. The header files for the examples in the rest of this chapter are shown in this first program.

```
#include <bsm/auditwrite.h>
#include <tsol/label.h>
#include <sys/param.h>
#include <bsm/libbsm.h>
#include <tsol/priv.h>

main()
{
/* Invalid call missing AW_END. Do not do it this way. */
    auditwrite(AW_EVENT, "AUE_second_signature", AW_WRITE);
}
```

An invalid call is logged to `syslog`, and if the invalid record has enough information, it is also logged to the audit trail. In the example, the invalid call is logged to `syslog` only with the following information:

```
header, 194,2,auditwrite routine fail,,Fri Sep 06 10:11:33 1996,
+ 179 msec text,
auditwrite routine aborted: aw_errno = 6 = Command invalid, errno = 0
= no such device or address
subject,zelda,zelda,staff,zelda,staff,1774,348,0 0 phoenix
slabel,C
return,failure,-1
```

Valid Call

This call to the `auditwrite`(3TSOL) routine includes the `AW_END` command and logs the `AUE_second_signature` event to the audit trail.

```
/* Valid call that includes AW_END */
auditwrite(AW_EVENT, "AUE_second_signature", AW_WRITE, AW_END);
```

The viewing terminal shows this record:

```
header, 4022,2,second signature requested,,Fri Sep 06
  10:16:49 1996 + 969 msec
subject,zelda,zelda,staff,zelda,staff,1774,348,0 0 phoenix
slabel,C
return,success,0
```

Creating a Minimum Audit Record

An audit record consists of a sequence of tokens. Each token of the record starts with a token type followed by the token values. You can put whatever tokens and values you want into an audit record by passing the appropriate token commands to the `auditwrite`(3TSOL) routine.

At a minimum, every audit record has the header, subject, slabel, and return tokens. The `auditwrite`(3TSOL) routine call from the previous example generates the minimum audit record by specifying the `AW_EVENT` token command only.

Note – Remember the `proc_audit_appl` privilege is needed in the effective set whenever you call `auditwrite`(3TSOL). The code comments indicate where privilege bracketing as described in Chapter 3 should take place. The remaining examples will not show the comments, because it is assumed you understand to do this.

```
/* Turn proc_audit_appl on in the effective set */
auditwrite(AW_EVENT, "AUE_second_signature", AW_WRITE, AW_END);
/* Turn the proc_audit_appl privilege off */
```

The output lines below have one token each. The first word on each line is the token ID followed by the token components. The description text defined in `/etc/security/audit_event` (second signature requested) is added to the header token.

By default the subject, slabel (sensitivity label), and return tokens are placed in the audit record even though the `AW_SUBJECT`, `AW_SLABEL`, and `AW_RETURN` token commands were not passed to this `auditwrite`(3TSOL) routine call.

- By default, the subject and slabel token values contain the security attribute information and sensitivity label of the process.
- By default, the return token has a return value of 0 (success).

If you pass `AW_SUBJECT`, `AW_SLABEL`, or `AW_RETURN` to the `auditwrite`(3TSOL) routine, you must explicitly define the token values. Auditing preselection and post-selection rely on the return token value to select audit records by success or failure. Always include the return token and the appropriate success or failure value in an audit record as described in "Return Token" on page 155.

```
header, 4022,2,second signature requested,,Fri Sep 06
 10:16:49 1996 + 969 msec
subject,zelda,zelda,staff,zelda,staff,1774,348,0 0 phoenix
slabel,C
return,success,0
```

Token Structure

Trusted Solaris Audit Administration presents the structure for every token and byte sizes for each component. To help you get an idea of how to read the records and determine record size if space is a concern, the subject token structure is presented here.

Token	Audit	User	Group	Real user	Real group	Process	Session	Device	Machine
ID	ID	ID	ID	ID	ID	ID	ID	ID	ID
subject	zelda	root	other	root	other	1774	348	0 0	phoenix
1 byte	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes	4 bytes

Return Token

The Return token is `AW_RETURN` and takes a number (success or fail) and a return value. By default the return token indicates success and shows a return value of 0. You will want to set the return token value just before passing the token to the `auditwrite`(3TSOL) routine.

The return value affects whether or not the audit record is logged to the audit trail. If the process preselection mask audits the class to which the event belongs for failures only, a successful event is not logged. If the process preselection mask audits the class to which the event belongs for success only, a failed event is not logged. If the process preselection mask audits the class to which the event belongs for success and failure, successful and failed events are both logged. Also, the `auditreduce`(1M) post-selection program selects audit records by the success or failure value in the record's return token.

This example creates part of an audit record showing that a second signature was requested by the system. The `signature_request`() function attempts to obtain the signature and returns information on the success or failure of the attempt and sets the *signature_request* and *retval* parameters. The *succ_or_fail* parameter is set according to the value in *signature_request* and it and *retval* are passed as values for the `AW_RETURN` token.

```
char succ_or_fail;
u_int retval;

auditwrite(AW_TEXT, "Second signature needed,"
    AW_APPEND, AW_END);

if (signature_request() == -1) {
    succ_or_fail = -1;
    retval = -2;
} else {
    succ_or_fail = 0;
    retval = 1;
}

auditwrite(AW_EVENT, "AUE_second_signature",
    AW_RETURN succ_or_fail, retval,
    AW_WRITE, AW_END);
```

The signature was not obtained and the viewing terminal shows this record:

```
header,128,2,second signature requested,,Wed
  Sep 11 10:17:37 1996, + 239 msec
text, Second signature needed
return,failure,-2
subject,zelda,zelda,staff,zelda,staff,1905,536,0 0 phoenix
slabel,C
```

Queueing Audit Records

To minimize system call overhead, audit records can be queued (`AW_QUEUE`) and written in one call to the `auditwrite`(3TSOL) routine. In this example, audit queueing is turned on in the first call to the `auditwrite()` routine and set to flush entire records when the queue contains 200 bytes of audit records. You can force the queue to flush with the `AW_FLUSH` token, and the queue automatically flushes whenever it is turned off with the `AW_NOQUEUE` token.

In this example, the queue flushes manually after the second record is added in spite of the fact that the queue does not yet have 200 data bytes. The queue flushes again at the end when queueing is turned off.

The byte limit does not cause partial records to be written to the audit trail. If the queue holds 200 bytes, all records from all calls to the `auditwrite`(3TSOL) routine are flushed in their entirety when the limit is reached including any data bytes over 200 that make a complete audit record.

```
/* Set up queue to flush every 140 bytes */
    auditwrite(AW_QUEUE, 200, AW_END);

/* Write records to the queue */
    auditwrite(AW_EVENT, "AUE_second_signature",
        AW_TEXT, "First record in queue",
        AW_WRITE, AW_END);

    auditwrite(AW_EVENT, "AUE_second_signature_verify",
        AW_TEXT, "Second record in queue",
        AW_WRITE, AW_END);
```

```
/* Flush the queue manually */
    auditwrite(AW_FLUSH, AW_END);

/* Add another record */
    auditwrite(AW_EVENT, "AUE_second_signature",
        AW_TEXT, "Third record in queue",
        AW_WRITE, AW_END);

/* End queueing and flush */
    auditwrite(AW_NOQUEUE, AW_END);
```

The viewing terminal shows the following audit records:

```
header,204,3,second signature requested,,Mon Sep 09 10:26:28 1996,
 + 150 msec
text,First record in queue
subject,zelda,zelda,staff,zelda,staff,6098,5879,0 0 phoenix
slabel,C
return,success,0

header,204,4,second signature added,,Mon Sep 09 10:26:28 1996,
 + 152 msec
text,Second record in queue
subject,zelda,zelda,staff,zelda,staff,6098,5879,0 0 phoenix
slabel,C
return,success,0

header,204,5,second signature requested,,Mon Sep 09 10:26:28 1996,
 + 155 msec
text,Third record in queue
subject,zelda,zelda,staff,zelda,staff,6098,5879,0 0 phoenix
slabel,C
return,success,0
```

Specifying a Preselection Mask

Audit records are selected according to a process preselection mask set for the execution environment as explained in "Audit Control (Process Preselection Mask)" on page 151. In addition, the `auditwrite`(3TSOL) routine has an `AW_PRESELECT` token that takes an audit mask structure for its value. This token causes subsequent calls to `auditwrite()` to audit classes based on the settings in the audit mask value until the `AW_NOPRESELECT` token is passed to `auditwrite()` telling it to use the environment preselection mask.

This example creates a process preselection mask to audit the example class (`ec`) for failures and passes it to the `auditwrite`(3TSOL) routine with `AW_PRESELECT` token to put it into effect. Now, only failed events belonging to the example class are logged to the audit trail. When preselection is turned off, the environment process preselection mask is restored, which for the purposes of these examples, audits events in the example class for success and failure.

```
char succ_or_fail;
u_int retval;
au_mask_t mask;

/* Create mask to audit failed events only in the ec class */
    getauditflagsbin("-ec", &mask);

/* Use new audit preselection mask */
    auditwrite(AW_PRESELECT, &mask, AW_END);

/* Code to generate audit records */
    auditwrite(AW_TEXT, "Second signature needed," AW_APPEND, AW_END);
    if (signature_request() == -1) {
        succ_or_fail = -1;
        retval = -2;
    } else {
        succ_or_fail = 0;
        retval = 1;
    }
    auditwrite(AW_EVENT, "AUE_second_signature",
        AW_RETURN succ_or_fail, retval, AW_WRITE, AW_END);

/* Restore environment preselection mask */
/* Events in the ec class are again audited for success and failure */
    auditwrite(AW_NOPRESELECT, AW_END);
```

Creating Audit Records in Parallel

Audit records are created with the default record descriptor unless record descriptors (AW_GETRD) are used (similar to file descriptors). This example uses record descriptors *ad1* and *ad2* to create two records in parallel, writes *ad2* to the audit trail, and discards (AW_DISCARDRD) *ad1*. AW_DEFAULTRD (shown in the next example) switches record creation to the default record descriptor.

```
int ad1, ad2;
/* Get and use a record descriptor */
    auditwrite(AW_GETRD, &ad1, AW_END);
    auditwrite(AW_USERD, ad1, AW_END);

/* Append record information to the memory location at ad1 */
    auditwrite(AW_EVENT, "AUE_second_signature", AW_TEXT, "ad1 one",
    AW_APPEND, AW_END);

/* Get second record descriptor */
```

```
    auditwrite(AW_GETRD, &ad2, AW_END);

/* Append record information to ad1 */
    auditwrite(AW_PATH, "/export/home/zelda/document_4_sig_req",
    AW_APPEND, AW_END);

/* Use ad2 */
    auditwrite(AW_USERD, ad2, AW_END);

/* Append and write record at ad2 */
    auditwrite(AW_PATH, "/export/home/zelda/document_4_sig_ver",
    AW_APPEND, AW_END);
    auditwrite(AW_EVENT, “AUE_second_signature_verify”,
    AW_WRITE, AW_END);

/* Discard ad1 */
    auditwrite(AW_DISCARDRD, ad1, AW_END);
```

The viewing terminal shows the following record:

```
header,141,2,second signature requested,,Wed Sep 11 11:16:29 1996,
 + 150 msec
path,/export/home/zelda/document_4_sig_ver
subject,zelda,zelda,staff,zelda,staff,1983,536,0 0 phoenix
slabel,C
return,success,0
```

Using the Save Area

You can turn on a save area (AW_SAVERD) and store tokens there to be prepended to records before they are written to the audit trail. Getting and using a save area is similar to getting and using a record descriptor. The save areas is turned off with AW_NOSAVE.

```
    int ad1, ad2, ad3;
/* Turn on and use save area ad1 */
    auditwrite(AW_SAVERD, &ad1, AW_END);
    auditwrite(AW_USERD, ad1, AW_END);

/* Put text at ad1 to be prepended to other records */
    auditwrite(AW_TEXT, "Prepended Text", AW_APPEND, AW_END);

/* Use the default record descriptor and write an event there */
    auditwrite(AW_DEFAULTRD, AW_END);
    auditwrite(AW_EVENT, “AUE_second_signature”,
        AW_TEXT, "Default record",
        AW_WRITE, AW_END);
```

```
/* Get and use record descriptor ad2 */
    auditwrite(AW_GETRD, &ad2, AW_END);
    auditwrite(AW_USERD, ad2, AW_END);

/* Write an event to ad2 */
    auditwrite(AW_EVENT, "AUE_second_signature",
        AW_TEXT, "ad2 record", AW_WRITE, AW_END);

/* Discard the save area */
    auditwrite(AW_NOSAVE, AW_END);

/* Get and use record descriptor ad3 */
    auditwrite(AW_GETRD, &ad3, AW_END);
    auditwrite(AW_USERD, ad3, AW_END);

/* Write an event to ad3 */
    auditwrite(AW_EVENT, "AUE_second_signature_verify",
   AW_TEXT, "ad3 with no prepend", AW_WRITE, AW_END);
```

The viewing terminal shows these records:

```
header,132,2,second signature requested,,Wed Sep 11 11:16:29 1996,
 + 133 msec
text,Prepended Text
text,Default record
subject,zelda,zelda,staff,zelda,staff,1983,536,0 0 phoenix
slabel,C
return,success,0

header,128,2,second signature requested,,Wed Sep 11 11:16:29 1996,
 + 140 msec
text,Prepended Text
text,ad2 record
subject,zelda,zelda,staff,zelda,staff,1983,536,0 0 phoenix
slabel,C

header,125,2,second signature added,,Wed Sep 11 11:16:29 1996,
 + 143 msec
text,ad3 with no prepend
subject,zelda,zelda,staff,zelda,staff,1983,536,0 0 phoenix
slabel,C
```

Using the Server Area and Adding a Sensitivity Label

The `AW_SERVER` token turns on the trusted server option, which indicates the calling process is a server. When the trusted server is enabled, the `auditwrite`(3TSOL) routine automatically generates header and return tokens, but not the subject and slabel tokens automatically generated when the trusted server is not enabled (see "Creating a Minimum Audit Record" on page 154). When the trusted server is enabled, you must explicitly pass the `AW_SUBJECT` and `AW_SLABEL` tokens to include this information in the record.

This example turns on the trusted server option, writes a record, writes another record including the sensitivity label, then turns off the trusted server option and writes a final record so you can see the difference. The `sys_trans_label` privilege is needed to translate the Secret sensitivity label because the process running at Confidential does not dominate Secret.

```
bslabel_t senslabel;

/* Create a sensitivity label of Secret */‘
    stobsl(“Secret”, &senslabel, NEW_LABEL, &error);

/* Turn on the trusted server option */
    auditwrite(AW_SERVER, AW_END);

/* Write a record to the audit trail */
    auditwrite(AW_EVENT, “AUE_second_signature”,
        AW_TEXT, "Some text",
        AW_WRITE, AW_END);

/* Write a record to the audit trail with the sensitivity label */
    auditwrite(AW_EVENT, “AUE_second_signature”,
        AW_TEXT, "Sensitivity label added",
        AW_SLABEL, &senslabel,
        AW_WRITE, AW_END);

/* Turn off the trusted server option */
    auditwrite(AW_NOSERVER, AW_END);

/* Write a final record to the audit trail */
    auditwrite(AW_EVENT, “AUE_second_signature”,
        AW_TEXT, "Some more text",
        AW_WRITE, AW_END);
```

The viewing terminal shows these records:

```
header,38,2,second signature requested,,Wed Sep 11 12:46:41 1996
 + 710 msec
text,Some text
return,success,0

header,38,2,second signature requested,,Wed Sep 11 12:46:41 1996
 + 780 msec
text,Sensitivity label added
slabel,S
return,success,0

header,112,2,second signature requested,,Wed Sep 11 12:46:41 1996
```

```
 + 799 msec
text,Some more text
return,success,0

subject,zelda,zelda,staff,zelda,staff,420,286,0 0 phoenix
slabel,C
return,success,0
```

Argument Information

The AW_ARG token lets you write argument information to an audit record. This example writes the return value for the signature_request() function, which is really the first and only parameter to the return() call inside the function. The argument number follows the AW_ARG token, which is followed by descriptive text and the argument value.

```
retval = signature_request();
auditwrite(AW_EVENT,
    "AUE_second_signature",
    AW_ARG, 1,
    "Signature request return value",
    retval);
```

The viewing terminal shows this record where the return value is written as 0xffffffff:

```
header,137,3,second signature requested,,Fri Mar 21 08:51:19 1997,+ 329 msec
argument,1,0xffffffff,Signature request return value
subject,zelda,zelda,staff,zelda,staff,420,286,0 0 phoenix
slabel,C
return,success,0
```

Command Line Arguments

The AW_EXEC_ARGS token lets you place the command line arguments stored in *argv* in the audit record.

```
main(argc, argv)
int argc;
char **argv;
{
/* Application code */
/* ... */
```

```
    auditwrite(AW_EVENT,
        “AUE_second_signature”,
        AW_EXEC_ARGS, argv
        AW_WRITE, AW_END);
}
```

The viewing terminal shows this record when the program is executed as follows: `program Hello World!`:

```
header,120,3,second signature requested,,Fri Mar 21 09:31:01 1997,
 +989 msec
exec_args,3,
program,Hello World!
subject,zelda,zelda,staff,zelda,staff,420,286,0 0 phoenix
slabel,C
return,success,0
```

Privilege Sets

The `AW_PRIVILEGE` token places a privilege set into the audit record. This example logs the allowed privilege set for the specified executable file to the audit record.

```
priv_set_t allowed_set;

PRIV_EMPTY(&allowed_set);

retval = getfpriv(“/export/home/zelda/program”,
    PRIV_ALLOWED,
    allowed_set);

auditwrite(AW_EVENT,
    “AUE_second_signature”,
    AW_PRIVILEGE, AU_PRIV_ALLOWED, &allowed_set,
    AW_WRITE, AW_END);
```

The viewing terminal shows this record:

```
header,116,3,second signature requested,,Fri Mar 21
 10:12:21 1997, + 809 msec
privilege,allowed,proc_audit_appl
subject,zelda,zelda,staff,zelda,staff,420,286,0 0 phoenix
slabel,C
return,success,0
```

Interprocess Communications Identifier

The AW_IPC token places the specified interprocess communications (IPC) identifier into the audit record. This example creates a semaphore set and puts the semaphore identifier into the audit record.

```
int semid;

semid = semget(IPC_PRIVATE, 3, IPC_CREAT);

auditwrite(AW_EVENT,
    “AUE_second_signature”,
    AW_IPC, AT_IPC_SEM, semid,
    AW_WRITE, AW_END);
```

The viewing terminal shows this record where 4 is the semaphore ID:

```
header,104,3,second signature requested,,Fri Mar 21
 12:45:21 1997, + 339 msec
IPC,sem,65539
subject,zelda,zelda,staff,zelda,staff,420,286,0 0 phoenix
slabel,C
return,success,0
```

CHAPTER 9

Accessing User and Rights Profile Data

This chapter describes the programming interfaces that read entries in the user and profile databases. Entries are stored in these databases when the system administrator sets up users and profiles using the Solaris Management Console (SMC) graphical user interfaces.

The User Databases

In the Solaris and the Trusted Solaris environment, user information is held in four databases:

- `user_attr`(4) – The `/etc/user_attr` file contains extended user attributes, using a keyword=value format.
- `auth_attr`(4) – The `/etc/security/auth_attr` file contains the definitions of authorizations, which can be included in rights profiles.
- `prof_attr`(4) – The `/etc/security/prof_attr` file contains the name, description, authorizations, subordinate rights profiles, and help files for rights profiles.
- `exec_attr`(4) – The `/etc/security/exec_attr` file contains commands and actions with security attributes assigned to rights profiles.

The following figure shows how the user databases work together and with `policy.conf`(4) and `label_encodings`(4) to provide user attributes.

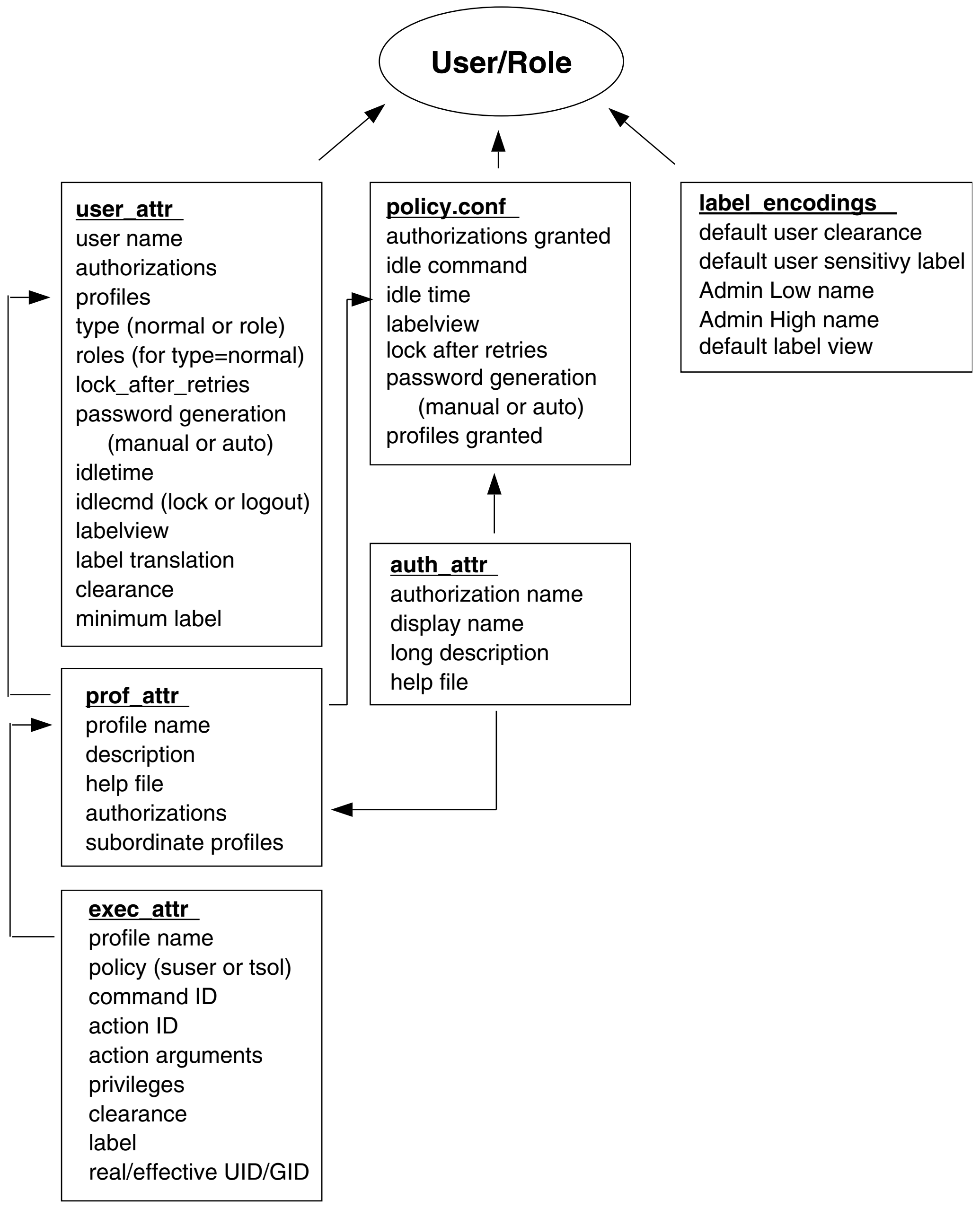

FIGURE 9–1 Trusted Solaris User Databases

The `user_attr` database contains the attributes shown, including a comma-separated list of profile names. The contents of the profiles are split between the prof_attr database, which contains profile identification information, authorizations assigned to the profile, and subordinate profiles, and the exec_attr database, which contains commands and actions with their associated security attributes. The auth_attr file supplies available authorizations to the prof_attr database and the policy.conf database. (Note that although it is possible to assign authorizations directly to users through user_attr, this practice is discouraged.) The policy.conf file supplies default attributes to be applied to all users on the machine. The label_encodings file supplies label defaults if they are not otherwise specified.

Note – The exec_attr entries within a profile are searched only in the scope in which that profile is found. The scope (files, NIS, or NIS+), is specified in the nsswitch.conf file.

Accessing the User Databases

The programming interfaces for manipulating user data require the following header files:

```
#include <user_attr.h>
#include <prof_attr.h>
#include <exec_attr.h>
#include <auth_attr.h>
```

The examples in this chapter compile with the following libraries:

```
-lsecdb -lnsl -lcmd -DTSOL
```

Working with User Data

The main interface for accessing user information is the getuserattr(3SECDB) family of interfaces. The getuserattr function enumerates the user_attr entries. The getusernam function searches for a user_attr entry with a given name. In similar fashion, the getuseruid function searches for a user_attr entry with a given UID. Successive calls to these functions return successive user_attr entries or NULL.

Working with Rights Header Data

The rights profile data is spread between two databases: `prof_attr`(4) and `exec_attr`(4). There are two corresponding interface families for accessing rights profiles data: `getprofattr`(3SECDB) and `getexecattr`(3SECDB).

The `getprofattr` function enumerates the `prof_attr` entries. The `getprofnam` function searches for a `prof_attr` entry with a given name. The `getproflist` function searches for supplementary profiles.

An example program using the `getprofattr` function follows.

```
#include <stdio.h>
#include <prof_attr.h>

main(int argc, char *argv[])
{
        profattr_t      *profp = NULL;
        int             i;
        char            *kv_str = NULL;
        char            *attr[] = {     PROFATTR_AUTHS_KW,
                                        PROFATTR_PROFS_KW,
                                        "help",
                                        NULL };

        if (argc != 2) {
                printf("\tUsage: %s \"profile name\"\n", argv[0]);
                printf("\t\tPut multi-word profile names in quotes\n");
                exit(1);
        }

        if ((profp = getprofnam(argv[1])) == NULL) {
                printf("\tNo prof_attr entry found for %s\n", argv[1]);
                exit(0);
        }
        if (profp->name)
                printf("\t%s: %s\n", PROFATTR_COL0_KW, profp->name);
        if (profp->res1)
                printf("\t%s: %s\n", PROFATTR_COL1_KW, profp->res1);
        if (profp->res2)
                printf("\t%s: %s\n", PROFATTR_COL2_KW, profp->res2);
        if (profp->desc)
                printf("\t%s: %s\n", PROFATTR_COL3_KW, profp->desc);
        if (profp->attr) {
                for (i = 0; attr[i] != NULL; i++) {
                        if (kv_str = kva_match(profp->attr, attr[i]))
                                printf("\t%s: %s\n", attr[i], kv_str);
                }
        }

        free_profattr(profp);
}
```

This program gets the six fields in the argument's `prof_attr` record and dumps them to a display as follows:

```
% getprof ''Media Backup''
    name: Media Backup
    res1:
    res2:
    desc: Backup files and file systems
    auths: solaris.device.allocate
    help: RtMediaBkup.html
```

Working with Rights Profile Execution Data

The rights profile data is spread between two databases: `prof_attr`(4) and `exec_attr`(4). The `getexecattr`(3SECDB).

This example program uses the `getexecattr()` routine to find the first exec_attr entry of type cmd in profile supplied.

```
#include <stdio.h>
#include <exec_attr.h>


main(int argc, char *argv[])
{
    execattr_t    *execp = NULL;
    int        i;
    int        search_flag = GET_ONE;
    char        *type = KV_COMMAND;
    char        *id = NULL;
    char        *kv_str = NULL;
    char        *attr[] = {    EXECATTR_EUID_KW,
                    EXECATTR_EGID_KW,
                    EXECATTR_UID_KW,
                    EXECATTR_GID_KW,
                    EXECATTR_PRIV_KW,
                    EXECATTR_LABEL_KW,
                    EXECATTR_CLEAR_KW,
                    NULL };

    if (argc != 2) {
        printf("\tUsage: %s \"profile name\"\n",  argv[0]);
        printf("\t\tPut multi-word profile name in quotes.\n");
        exit(1);
    }

    if ((execp = getexecprof(argv[1], type, id, search_flag)) == NULL) {
        printf("\tNo exec_attr entry found for id %s of type %s"
            " in profile %s\n",
            ((id == NULL) ? "NULL" : id), type, argv[1]);
        exit(0);
    }
    if (execp->name)
        printf("\t%s: %s\n", EXECATTR_COL0_KW, execp->name);
    if (execp->policy)
        printf("\t%s: %s\n", EXECATTR_COL1_KW, execp->policy);
    if (execp->type)
        printf("\t%s: %s\n", EXECATTR_COL2_KW, execp->type);
    if (execp->res1)
        printf("\t%s: %s\n", EXECATTR_COL3_KW, execp->res1);
    if (execp->res2)
        printf("\t%s: %s\n", EXECATTR_COL4_KW, execp->res2);
    if (execp->id)
        printf("\t%s: %s\n", EXECATTR_COL5_KW, execp->id);
    if (execp->attr) {
        for (i = 0; attr[i] != NULL; i++) {
            if (kv_str = kva_match(execp->attr, attr[i]))
                printf("\t%s: %s\n", attr[i], kv_str);
        }
    }

    free_execattr(execp);
}
```

Here is a typical result.

```
% getexecprof ‘‘Media Backup’’
        name: Media Backup
        policy: tsol
        type: cmd
        res1:
        res2:
        id: /usr/lib/fs/ufs/ufsdump
        egid: 3
        privs: 1,4,5,8,10,11,12,19,71
```

The next example program uses the `getexecattr()` routine to find the first exec_attr entry of type cmd in the first profile for the supplied user.

```
#include <stdio.h>
#include <exec_attr.h>

main(int argc, char *argv[])
{
    execattr_t    *execp = NULL;
    int        i;
    int        search_flag = GET_ONE;
    char        *type = KV_COMMAND;
    char        *id = NULL;
    char        *kv_str = NULL;
    char        *attr[] = {    EXECATTR_EUID_KW,
                    EXECATTR_EGID_KW,
                    EXECATTR_UID_KW,
                    EXECATTR_GID_KW,
                    EXECATTR_PRIV_KW,
                    EXECATTR_LABEL_KW,
                    EXECATTR_CLEAR_KW,
                    NULL };

    if (argc != 2) {
        printf("\tUsage: %s \"login name\"\n", argv[0]);
        exit(1);
    }

    if ((execp = getexecuser(argv[1], type, id, search_flag)) == NULL) {
        printf("\tNo exec_attr entry found for id %s of type %s"
            " for user %s\n",
            ((id == NULL) ? "NULL" : id), type, argv[1]);
        exit(0);
    }
    if (execp->name)
        printf("\t%s: %s\n", EXECATTR_COL0_KW, execp->name);
    if (execp->policy)
        printf("\t%s: %s\n", EXECATTR_COL1_KW, execp->policy);
    if (execp->type)
        printf("\t%s: %s\n", EXECATTR_COL2_KW, execp->type);
    if (execp->res1)
        printf("\t%s: %s\n", EXECATTR_COL3_KW, execp->res1);
    if (execp->res2)
        printf("\t%s: %s\n", EXECATTR_COL4_KW, execp->res2);
    if (execp->id)
        printf("\t%s: %s\n", EXECATTR_COL5_KW, execp->id);
    if (execp->attr) {
        for (i = 0; attr[i] != NULL; i++) {
            if (kv_str = kva_match(execp->attr, attr[i]))
                printf("\t%s: %s\n", attr[i], kv_str);
        }
    }

    free_execattr(execp);
}
```

Here is a typical result.

```
% getexecuser janez
        name: Media Backup
        policy: tsol
        type: cmd
        res1:
        res2:
        id: /usr/lib/fs/ufs/ufsdump
        egid: 3
        privs: 1,4,5,8,10,11,12,19,71
```

CHAPTER 10

Interprocess Communications

The Trusted Solaris environment enforces mandatory access controls and discretionary access controls between communicating processes on the same host and across the network. This chapter summarizes the interprocess communication (IPC) mechanisms available in the Trusted Solaris environment and how access controls and privileges apply.

Privileges and Communications

Interprocess communications might involve several types of privileges depending on the type of interprocess communication in use. The following guidelines can help you know which type of privilege to use. This chapter and the chapters that follow describe specific privileges in detail. Refer to the `priv_desc`(4) man page for a complete list of privileges with descriptions.

- Access and ownership controls between processes are overridden by process privileges such as `proc_mac_read` and `proc_owner`.
- Access controls between a process and a file are overridden by file privileges such as `file_mac_read` and `file_dac_write`.

- Access and ownership controls between a process and a System V IPC object are overridden by IPC privileges such as `ipc_mac_read` and `ipc_owner`).
- Access and ownership controls between two communication endpoints are overridden by network privileges such as `net_mac_read` and `net_upgrade_sl`.

Unnamed Pipes

Unnamed pipes form a one-way flow of data between two or more related processes. Because all processes communicating over a pipe share a common ancestor, they all have the same user ID, group ID, and sensitivity label inherited from the ancestor process unless privileges have been used to change those attributes. No mandatory or discretionary access checks are done when a pipe is opened, and no access checks are done for read and writes to a pipe.

If a process with an open pipe uses privilege to change its user ID, group ID, or sensitivity label, subsequent communication over the pipe effectively bypasses discretionary and mandatory access controls and the privileged process must apply its own controls to the communication.

The sensitivity label of the process writing the data is associated with each byte of data in the pipe. See the appropriate man page for specific information on security policy and applicable privileges.

Named Pipes (FIFOs)

Named pipes (FIFOs) are similar to unnamed pipes except they are associated with a file system entry that allows unrelated processes to find and open a named pipe for communication. Discretionary and mandatory access controls are enforced when the named pipe is opened and FIFO special file created based on the named pipe's permission bits and sensitivity label.

The sensitivity label of the process writing the data is associated with each byte of data send down the pipe. The mandatory access policy for writing to and reading from a named pipe is read-equal and write-equal. See the appropriate man page for specific information on security policy and applicable privileges.

Pseudo-Terminal Devices (PTYs)

Pseudo-terminal devices (PTYs) are automatically allocated special device files that operate in controller/slave pairs. A process opening one member of a pair communicates with a process opening the other member of the pair. The PTY pair emulates a terminal interface. PTYs are used for `cmdtool` windows and to support remote login services. Discretionary and mandatory access controls are enforced when the PTY is opened.

- If neither the slave nor the controller device is already open, the device special files for both devices are modified to set their user ID and sensitivity label to the opening process's effective user ID and sensitivity label with permission bits initialized to 600.
- If either the slave or the controller device is already open, discretionary and mandatory access controls use the user ID, permission bits, and sensitivity label already set on the device special file.

Data written to the controller device is read from the slave device after undergoing terminal input processing such as erase/kill. Data written to the slave device is read from the controller device after undergoing terminal output processing such as NL to CR-LF translation. The mandatory access policy to read from and write to a PTY is read-down and write-up. See the appropriate man page for specific information on security policy and applicable privileges.

Signals

Signals inform processes of asynchronous events. Discretionary access policy requires the sender's real or effective user ID to equal the receiver's real or effective user ID. The mandatory access policy is read-down and write-up. See the appropriate man page for specific information on security policy and applicable privileges.

Process Tracing

Process tracing is a debugging tool where one process manipulates the contents of another process by doing such things as reading from and writing to its address space and registers, altering its flow of control, and setting breakpoints. The discretionary access policy requires the effective user IDs of the processes be equal. The mandatory access policy for manipulating the contents of another process is read-equal and write-equal. See the appropriate man page for specific information on security policy and applicable privileges.

Mapped Memory

Mapped memory allows a process to map part or all of a file's contents into its address space. Once the file has been mapped, direct addressing of the file's contents is done through machine instruction accesses to the mapped memory region. A process can map multiple files, and the same file can be mapped into multiple processes.

Discretionary and mandatory access checks are performed when the file is opened. If a file is opened for read only, it may be mapped for reading only even when the file's attributes permit write access. See the appropriate man page for specific information on security policy and applicable privileges.

System V IPC

The Trusted Solaris environment supports System V IPC and provides additional interfaces for managing the CMW label, sensitivity label, and Access Control List (ACL) on System V IPC objects.

The sensitivity label of the process creating the System V IPC object is associated with each byte of data written to the object. The mandatory access policy is read-equal and write-equal. Privileged processes can access System V IPC objects at sensitivity labels other than the process sensitivity label. Chapter 11 describes the interfaces, security policy, and privileges for System V IPC objects.

Communication Endpoints

The Trusted Solaris environment supports interprocess communication over communication endpoints using the following socket-based mechanisms:

- Multilevel Ports
- Berkeley sockets
- Transport Layer Interface (TLI)
- Trusted Information Exchange (TSIX) library
- Remote Procedure Calls (RPC)

This section summarizes the socket communication mechanisms and related security policy. See the appropriate man page for specific information on security policy and applicable privileges.

Multilevel Ports

The Trusted Solaris environment supports single-level and multilevel ports. A multilevel port can receive data at any sensitivity label, and a single-level port can receive data at a designated sensitivity label only.

- Single-level port – A communication channel is established between two unprivileged applications. The sensitivity label of the communication endpoints must be equal.
- Multilevel port – A communication channel is established between an application with `net_mac_read` in its effective set and any number of unprivileged applications running at different sensitivity labels. The application with `net_mac_read` in the effective set of its process can receive all data from the applications regardless of the receiving application's sensitivity label or process clearance. A multilevel communication channel cannot be established where there is already a single-level connection.

See "Client-Server Application" on page 202 in Chapter 12 for a short example application that establishes a multilevel port connection using Berkeley sockets and the TSIX library.

Note – If a connection is multilevel, be sure the application does not make a connection at one sensitivity label and send or receive data at another sensitivity label causing data to reach an unauthorized destination.

Sockets and TLI

The Trusted Solaris environment supports network communication using Berkeley sockets and Transport Layer Interface (TLI) over single-level and multilevel ports. The UNIX address family of system calls establishes process-to-process connections on the same host using a special file specified with a fully resolved pathname. The internet address family of system calls establishes process-to-process connections across the network using IP addresses and port numbers.

The PAF_DISKLESS_BOOT process attribute flag supports diskless boot servers. When this flag is on, the security attribute information in network packet headers is not sent. Getting and setting process attribute flags is covered in Chapter 2.

UNIX Address Family

In the UNIX address family of interfaces, only one server bind can be established to a single file. The server process needs the net_mac_read privilege in its effective set if a multilevel port connection is desired. If a single-level port connection is made instead, the server process needs mandatory read-equal access to the socket, and the client process needs mandatory write-equal access. Both processes need mandatory and discretionary access to the file. If access to the file is denied, the process denied access needs the appropriate file privilege in its effective set to gain access.

A server process can establish multiple single-level binds with files of the same name residing in different SLDs within the same MLD. This approach differs from a multilevel port connection in that it sets up parallel single-level port connections (polyinstantiated ports) and does not require privilege unless mandatory or discretionary access is denied to the specified single-level directory. See Chapter 7.

Internet Address Family

In the internet address family, the process can establish a single-label or multilabel connection to privileged or unprivileged port numbers. To connect to privileged port numbers, the net_priv_addr privilege is required in addition to the net_mac_read privilege if a multilevel port connection is desired.

TSIX

The Trusted Security Information Exchange (TSIX) library provides interfaces for receiving security attributes on incoming messages, and changing security attributes on outgoing messages. A message initially has the security attribute information of its sending process. The TSIX library lets you change security attributes directly on the message, on the communication endpoint over which the message is sent, or both. See Chapter 12 for the programming interfaces and related privileges.

RPC

The Trusted Solaris environment remote procedure call (RPC) mechanism is built on Berkeley internet sockets and the Trusted Security Information Exchange (TSIX) library, and supports Transport Layer Interface (TLI). RPC allows a server process to invoke a procedure on behalf of a client process and handle security attribute information on the message. See Chapter 13 for a description of the RPC programming interfaces and related privileges.

CHAPTER 11

System V Interprocess Communication

The Trusted Solaris environment supports the System V interprocess communication (IPC) mechanism and provides security features for labeled communications between System V IPC objects and both privileged and unprivileged processes. The chapter covers the following topics:

Privileged Operations

System V IPC objects are subject to discretionary and mandatory access controls, and discretionary ownership controls.

A System V IPC object is created from a key and accessed by an object descriptor returned when the IPC object is created. The object descriptor, like a file descriptor, is used for future operations on the object. The sensitivity label of the System V IPC object is the same as the sensitivity label of its creating process unless the creating process has the privilege to create the System V IPC object at a different label. A process can access a System V IPC object at its same sensitivity label unless the process has the privilege to access a System V IPC object at another label. Because keys are qualified by the sensitivity label at which they are created, there can be many objects that use the same key, but no more than one instance of a key (object ID) at a given sensitivity label.

- Message queues allow processes to place messages into a queue where any process can retrieve the message.

- Semaphore sets synchronize processes and are often used to control concurrent access to shared memory regions.
- Shared memory regions allow multiple processes to attach to the same region of memory to access changes to the memory.

Discretionary Access and Ownership Controls

Discretionary access to a System V IPC object is granted or denied according to the read and write modes associated with the object for owner, group, and other in much the same way as file access. System V IPC objects also have the creator user and creator group sets that control attribute change requests. The process that creates a System V IPC object is the owner and can set the discretionary permission bits to any value. To override discretionary access and ownership restrictions, the process needs the `ipc_dac_read`, `ipc_dac_write`, or `ipc_owner` privilege in its effective set, depending on the interface used or operation requested.

Mandatory Access Controls

Unprivileged processes can only refer to System V IPC objects and return an IPC descriptor at the process's correct sensitivity label. This makes the mandatory access controls read-equal and write-equal and eliminates naming and access conflicts when an unmodified base Solaris application using System V IPC runs at multiple sensitivity labels. To override mandatory access restrictions, the process needs the `ipc_mac_read` or `ipc_mac_write` privilege in its effective set, depending on the interface used.

Note – You cannot change the sensitivity label once it has been created.

Data Types, Header Files, and Libraries

To use the programming interfaces described in this chapter, you need the following header file:

```
#include <sys/ipcl.h>
```

The examples in this chapter compile with the following library:

```
-ltsol
```

Labels

Data structures for labels (`bclabel_t` and `bslabel_t`) are described in Chapter 4.

Programming Interface Declarations

These programming interfaces let you manage labels on System V IPC objects. The original unlabeled interfaces are still valid and available. These Trusted Solaris extensions provide access to the label information.

Message Queues

The `getmsgqcmwlabel`(2) routine gets the message queue CMW label.

The `msggetl`(2) routine creates a message descriptor at the specified sensitivity label.

See the `msgget`(2) and `msgctl`(2) man pages.

```
int  getmsgqcmwlabel(int msqid, bclabel_t *cmwlabel);
int  msggetl(key_t key, int msgflg, bslabel_t *senslabel);
```

Semaphore Sets

The `getsemcmwlabel`(2) routine gets the semaphore set CMW label.

The `semgetl`(2) routine creates a semaphore set at the specified sensitivity label.

```
int  getsemcmwlabel(int semid, bclabel_t *cmwlabel);

int  semgetl(key_t key,
   int nsems,
   int semflg,
   bslabel_t *senslabel);
```

Shared Memory Regions

The `getshmcmwlabel`(2) routine gets the shared memory region CMW label.

The `shmgetl`(2) routine creates a shared memory region at the specified sensitivity label.

```
int  getshmcmwlabel(int shmid, bclabel_t *cmwlabel);
int  shmgetl(key_t key, size_t size, int shmflg, bslabel_t *senslabel);
```

Using Shared Memory Labels

This example creates an identifier for a shared memory region at Confidential and gets the CMW label on the same shared memory region. The program is running at Top Secret.

```
#include <sys/ipc.h>
#include <sys/types.h>
#include <tsol/label.h>
#include <sys/shm.h>

main()
{
    int          id, retval, error, pid;
    bclabel_t    cmwlabel;
    bslabel_t    senslabel;
    char        *string = (char *)0;

    retval = stobsl("CONFIDENTIAL", &senslabel, NEW_LABEL, &error);

/* Create shared memory region at Confidential */
/* Turn ipc_mac_write on in the effective set */
    id = shmgetl(IPC_PRIVATE, 256, IPC_CREAT|0666, &senslabel);
/* Turn off ipc_mac_write */

/* Get CMW label of shared memory region */
/* Turn ipc_mac_read on in the effective set */
    retval = getshmcmwlabel(id, &cmwlabel);
/* Turn off ipc_mac_read */

/* Print CMW label */
    bcltos(&cmwlabel, &string, 0, LONG_WORDS);
    printf("CMW label = %s\n", string);
}
```

The `printf`(1) statement prints the following:

```
CMW label = UNCLASSIFIED[C]
```

CHAPTER 12

Trusted Security Information Exchange Library

The Trusted Security Information Exchange (TSIX) library provides interfaces for managing the security attribute information on a network message from within client and server applications. The TSIX library is based on Berkeley sockets and supports the transport layer interface (TLI).

The security attributes are stored in the data packet header separate from the message so they can be read separately. For example, an application can use the TSIX library to retrieve the security attributes and then test the sensitivity label attribute to determine whether or not the process needs privilege to read the data in the packet.

Security Attributes

By default, messages originating on a Trusted Solaris system acquire the following security attributes from the sending process:

- Audit ID
- Audit information (process preselection mask, audit terminal ID, and audit session ID)

- Effective group ID
- Effective privilege set
- Effective user ID
- Network session ID
- Process attribute flags
- Process clearance
- Process ID
- Sensitivity label
- Supplementary group ID

The TSIX library lets you change the user ID, group ID, sensitivity label, process clearance, or privilege attributes before the message is sent.

The TSIX library also lets you retrieve the security attributes on an incoming message. Because a distributed network can have any combination of host types running different Trusted networking protocols, not all protocols support all security attributes. Messages coming from or going to a host type other than a Trusted Solaris host will have very few of the above security attributes.

For example, the audit ID, audit information, and supplementary group ID attributes can only be sent from and received by a host running the TSIX protocol, and when a packet originates on a Solaris host, none of the Solaris security attributes are present when the packet arrives on a Trusted Solaris host.

Note – The TSIX library can be used in any application written for the Trusted Solaris environment. The TSIX protocol is not required to use the TSIX library.

Default security attributes are assigned to messages arriving on Trusted Solaris hosts from other host types according to settings in the network database files. Security attributes retrieved by TSIX library calls from incoming messages come out of the network database files if they did not arrive with the message. See the *Trusted Solaris Administrator's document set* for information on host types, their supported security attributes, and network database file defaults.

The sensitivity label of data sent over the network must be within the origination, destination, and next hop destination workstation accreditation ranges. There is no privilege to override this restriction.

Privileged Operations

No privileges are required to read security attributes retrieved from an incoming message. The following sections describe privileges used on outgoing messages.

Replying with Same Sensitivity Label

A server process can receive a message over a multilevel port at any sensitivity label dominated by the server process's clearance. However, the server reply is normally at the sensitivity label of the server process unless the server process has the `net_reply_equal` privilege in its effective set in which case the reply is sent at the sensitivity label of the last message received. See Chapter 10 for a discussion on single-level and multilevel ports.

Note – Make sure the `net_reply_equal` privilege is turned off if the receiving process needs to reply at a sensitivity label different from that of the requesting process. See "TCP/IP Server" on page 202 for an example situation where `net_reply_equal` must be turned off.

Changing Sensitivity Label

To respond to a single-level client, the server process needs the `proc_set_sl` privilege in its effective set to change the sensitivity label of its child to be the same as the sensitivity label of the requesting client.

Changing Security Attribute Information

To change the user ID, group ID, sensitivity label, process clearance, or privilege security attribute on an outgoing message or on the communication endpoint for outgoing messages, a process needs the appropriate network privilege in its effective set.

Sensitivity Labels

The sending process can set the sensitivity label for a message or communication endpoint to a new sensitivity label that does not dominate the object's existing sensitivity label if it has the `net_downgrade_sl` privilege in its effective set. The sending process can set the sensitivity label for a message or communication endpoint to a new sensitivity label that dominates the existing object's sensitivity label it has the `net_upgrade_sl` privilege in its effective set.

Process Clearance

The sending process needs the `net_setclr` privilege in its effective set to change the clearance sent with the message.

The system ensures that the clearance always dominates the sensitivity label. There is no privilege to override this restriction.

User and Group IDs

The sending process needs the `net_setid` privilege in its effective set to change the user or group ID.

Privileges

The sending process needs the `net_setpriv` privilege in its effective set to specify privileges to be sent with the message. The specified privileges must be in the permitted set of the sending process.

Data Types, Header Files, and Libraries

To use the programming interfaces described in this chapter, you need the following header file.

```
#include <tsix/t6attrs.h>
```

The examples in this chapter compile with the following libraries:

```
-lsocket -lt6 -ltsol
```

Attribute Structure

The `t6attr_t` data structure can hold the full set security attributes.

Attribute Enumerations

The `t6attr_id_t` structure contains enumerated constants that represent the full set of security attribute values. Variables of type `t6attr_t` are initialized with these constants. Most of the constants have a fixed size in bytes as shown below; however, `T6_GROUPS,` has a variable size that reflects the actual size of its value.

- The `t6set_attr`(3NSL) routine takes a parameter of any type that must be cast to the appropriate type shown below.
- The `t6get_attr`(3NSL) routine returns a variable of any type that must be cast to the appropriate type shown below.

Enumerated Constant	Description	Data Type	Size in Bytes
`T6_SL`	Sensitivity label	bslabel_t	36
`T6_SESSION_ID`	Network session ID	sid_t	4
`T6_CLEARANCE`	Clearance	bclear_t	36
`T6_PRIVILEGES`	Effective privileges	priv_set_t	16
`T6_AUDIT_ID`	Audit ID	au_id_t	4
`T6_PID`	Process ID	pid_t	4
`T6_AUDIT_INFO`	Additional audit info	auditinfo_t	24
`T6_UID`	Effective User ID	uid_t	4
`T6_GID`	Effective Group ID	gid_t	4
`T6_GROUPS`	Supplementary Group IDs	gid_t	Variable
`T6_PROC_ATTR`	Process Attribute Flags	pattr_t	4

Attribute Mask

The `t6mask_t` data structure represents the set of security attributes of current interest. A variable of type `t6mask_t` is initialized by assigning the following enumerated values.

`T6M_SL`	Sensitivity label
`T6M_SESSION_ID`	Network session ID
`T6M_CLEARANCE`	Clearance
`T6M_PRIVILEGES`	Effective privilege set

`T6M_AUDIT_ID`	Audit ID
`T6M_PID`	Process ID
`T6M_AUDIT_INFO`	Terminal ID and preselection masks
`T6M_UID`	Effective User ID
`T6M_GID`	Effective Group ID
`T6M_GROUPS`	Supplementary Group IDs
`T6M_NO_ATTRS`	No attributes
`T6M_ALL_ATTRS`	All attributes

Programming Interface Declarations

These network library routines handle security attributes on messages sent to and received from a Trusted Solaris host.

Get Attribute Masks

These routines create an attribute mask of system supported security attributes, attributes of the space allocated in the attribute structure, and attributes present in an attribute structure. You can use these routines instead of assigning `t6mask_t` enumerated values to a mask variable.

```
t6mask_t t6supported_attrs(void);
t6mask_t t6allocated_attrs(t6attr_t t6ctl);
t6mask_t t6present_attrs(t6attr_t t6ctl);
```

Allocate and Free Space

The `t6alloc_blk`(3NSL) routine creates a security attribute structure with enough space allocated for the security attributes specified in *new_attrs*. The `t6free_blk`(3NSL) routine frees the space allocated for the security attribute structure *t6ctl*.

```
t6attr_t t6alloc_blk(t6mask_t mask);
void t6free_blk(t6attr_t t6ctl);
```

Send and Receive Data

The t6sendto(3NSL) routine sends security attributes with a message. The t6recvfrom(3NSL) routine receives a message and its security attributes. When t6new_attr(3NSL) is *on,* t6recvfrom(3NSL) receives security attributes only when the attributes in *new_attrs* have changed.

Note – These routines are specific to sockets. For Transport Layer Interface (TLI), use t6last_attr(3NSL) in place of t6recvfrom(3NSL) and t6new_attr(3NSL); and t6set_endpt_default(3NSL) in place of t6sendto(3NSL).

```
ssize_t t6sendto(int sock,
    const char *msg,
    size_t len,
    int flags,
    const struct sockaddr *name,
    socklen_t namelen,
    const t6attr_t handle);

ssize_t t6recvfrom(int sock,
    void *buffer,
    size_t len,
    int flags,
    struct sockaddr *name,
    Psocklen_t namelenp,
    t6attr_t handle,
    t6mask_t *new_mask);

int t6new_attr(int fd, t6cmd_t cmd);
```

Get and Set Security Attributes

The t6get_attr(3NSL) routine gets the attribute in *attr_type* from the security attribute structure *t6ctl*. The return value should be cast to the correct type as described in “Attribute Enumerations” on page 189.

The t6set_attr(3NSL) routine sets the attribute in *attr_type* with the value specified in *attr* in the security attribute structure *t6ctl*.

```
void *t6get_attr(t6attr_id_t attr_type,
    const t6attr_t t6ctl);

int t6set_attr(t6attr_id_t attr_type,
    const void *attr,
    t6attr_t t6ctl);
```

Examine Security Attributes

The `t6peek_attr`(3NSL) routine examines the security attributes in *attr_ptr* on the next byte of data to be received, and the `t6last_attr`(3NSL) routine examines the security attributes on the last byte of data received.

```
int t6peek_attr(int fd, t6attr_t attr_ptr, t6mask_t *new_attrs);

int t6last_attr(int fd, t6attr_t attr_ptr, t6mask_t *new_attrs);
```

Get the Size of One Security Attribute

The `t6size_attr`(3NSL) routine gets the size in bytes of the value for the security attribute specified in *attr_type* in the security attribute structure *t6ctl*.

```
size_t t6size_attr(t6attr_id_t attr_type, const t6attr_t t6ctl);
```

Copy and Duplicate Security Attributes

These routines make a copy of *attr_src*. Refer to the `t6copy_blk`(3NSL) and `t6dup_blk`(3NSL) man pages.

```
int t6copy_blk(const t6attr_t attr_src, t6attr_t attr_dest);
t6attr_t t6dup_blk(const t6attr_t attr_src);
```

Compare Security Attributes

This routine compares one security attribute structure to another. Refer to the `t6cmp_blk`(3NSL) man page.

```
int t6cmp_blk(t6attr_t t6ctl1, t6attr_t t6ctl2);
```

Clear Security Attributes

This routine clears the attributes specified in *mask* from *t6ctl*. Refer to the `t6clear_blk`(3NSL) man page.

```
void t6clear_blk(t6mask_t mask, t6attr_t t6ctl);
```

Get and Set Endpoint Attributes

The `t6set_endpt_default`(3NSL) routine sets the security attribute values in *attr* indicated by *mask* on the communication endpoint. The `t6get_endpt_mask`(3NSL) routine sets the endpoint *mask* only.

The `t6get_endpt_default`(3NSL) routine gets the security attribute values in *attr* indicated by *mask* from the communication endpoint. The `t6get_endpt_mask`(3NSL) routine gets the endpoint *mask* only.

```
int t6get_endpt_default(int fd,
    t6mask_t *mask,
    t6attr_t attr);

int t6set_endpt_mask(int fd,
    t6mask_t mask);

int t6set_endpt_default(int fd,
    t6mask_t mask,
    const t6attr_t attr_ptr);

int t6get_endpt_mask(int fd,
    t6mask_t *mask);
```

Turn Extended Security Operations On and Off

This routine turns the extended security operations on and off for compatibility with other vendors. The operations are on by default. When off, messages can be sent and received as long as the communications are with the mandatory and discretionary access controls of the system. Refer to the `t6ext_attr`(3NSL) man page.

```
int t6ext_attr(int fd, t6cmd_t cmd);
```

Getting and Setting Security Attributes

These examples show how to set up a security attribute structure and masks to specify security attributes on outgoing data. The first example sets new security attributes on the message, and the second example sets new security attributes on the communication endpoint.

Security Attributes on Messages

This example sets up new sensitivity label and clearance attribute values to send with *msg*. This is done by doing the following:

- Defining a mask, *sendmask*, with only the sensitivity label and clearance defined.
- Allocating the security attribute structure *sendattrs* with *sendmask* so the attribute structure has room only for these two attributes.
- Setting the attribute values of Top Secret for the sensitivity label and clearance in *sendattrs*.
- Setting up communications over a communication endpoint.
- Sending *msg* with the security attributes over the communication endpoint.

Because the process sending *msg* is at Confidential, it needs the `net_setclr` and `net_upgrade_sl` privileges in its effective set to change the clearance and sensitivity label. The new sensitivity label and clearance override the sensitivity label and clearance *msg* received from its sending process. The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

```
#include <tsix/t6attrs.h>
#include <label.h>
main()
{
    int retval, sock, error;
    t6attr_t sendattrs;
    t6mask_t sendmask;
    char *msg = "Hello World!";
    bslabel_t senslabel;
    bclear_t clearance;
    struct sockaddr_in sin;

/* Initialize a mask with the sensitivity label and */
/* process clearance security attribute fields */
    sendmask = T6M_SL | T6M_CLEARANCE;
/* Allocate space for two security attribute structures */
/* using the masks so only the space needed is allocated */
    sendattrs = t6alloc_blk(sendmask);
/* Initialize senslabel and clearance to Top Secret */
    stobsl("TOP SECRET", &senslabel;, NEW_LABEL, &error;);
    stobclear("TOP SECRET", &clearance;, NEW_LABEL, &error;);
/* Set attribute values for the security attribute fields */
/* to be sent with the message */
    retval = t6set_attr(T6_SL, &senslabel;, sendattrs);
    printf("Retval1 = %d\n", retval);
    retval = t6set_attr(T6_CLEARANCE, &clearance;, sendattrs);
    printf("Retval2 = %d\n", retval);
/* Set up socket communications */
/* ... */
/* Send changed security attributes with the message */
/* Turn net_setclr and net_upgrade_sl on in the effective set */
    retval = t6sendto(sock, msg, sizeof(msg), 0, (struct sockaddr *) &sin;,
        sizeof(sin), &sendattrs;);
/* Turn off the net_setclr and net_upgrade_sl privileges */
    printf("Retval3 = %d\n bytes", retval);
}
```

The `printf` statements print the following:

```
Retval1 = 0
Retval2 = 0
Retval3 = 4 bytes
```

Security Attributes on Communication Endpoints

The first part of this example sets only the sensitivity label security attribute specified in *sendattrs* on the communication endpoint by using a different mask (*endptmask*) with *sendattrs*. This way, when privileged process sends a message over the communication endpoint using a form of transmission other than the `t6sendto`(3NSL) routine, or using the `t6sendto`(3NSL) routine with an attribute set that does not specify the sensitivity label, the sensitivity label is picked up from the communication endpoint. Because the process setting security attributes on the communication endpoint is running at Secret, it needs the `net_upgrade_sl` privilege in its effective set. The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

The next statements change the mask on the communication endpoint to *sendmask*, retrieve the endpoint mask and put it in *getmask*, allocate *getattrs* to hold a clearance, and get the binary clearance from the communication endpoint defaults and store it in *getattrs*.

Security attributes on the communication endpoint override the attributes acquired from the sending process. The security attributes on the message override the attributes from the communication endpoint.

```
#include <tsix/t6attrs.h>
include <tsol/label.h>
#include <tsol/priv.h>
main()
{ t6mask_t sendmask, endptmask, getmask;
    int fd, sock, retval;
    t6attr_t sendattrs, getattrs;
    sendmask = T6M_SL | T6M_CLEARANCE;
    sendattrs = t6alloc_blk(sendmask);


    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
/* Initialize a mask with the sensitivity label field */
```

```
    endptmask = T6M_SL;
/* Set the attribute in sendattrs indicated by the mask */
/* Turn net_upgrade_sl on in the effective set */
    set_effective_priv(PRIV_ON, 1, PRIV_NET_UPGRADE_SL);

    retval = t6set_endpt_default(sock, endptmask, &sendattrs;);

    set_effective_priv(PRIV_OFF, 1, PRIV_NET_UPGRADE_SL);
    printf("t6set_endpt_default return val: %d\n", retval);
/* Turn off the net_upgrade_sl privilege */
/* Change the endpoint mask to a different mask */

    retval = t6set_endpt_mask(sock, sendmask);

    printf("t6set_endpt_mask return val: %d\n", retval);
/* Get the current endpoint mask */

    retval = t6get_endpt_mask(sock, &getmask;);

    printf("t6get_endpt_mask return val: %d\n", retval);
/* Get the default clearance on the endpoint */

    getmask = T6M_CLEARANCE;
    getattrs = t6alloc_blk(getmask);
    retval = t6get_endpt_default(sock, &getmask;, getattrs);

    printf("t6get_endpt_default return val: %d\n", retval);
}
```

Receiving and Retrieving Security Attributes

This example receives a message with security attributes and retrieves the security attribute information.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>
main()
{
    char buf[512];
    int retval, len = sizeof(buf), sock;
    t6mask_t recvmask;
    t6attr_t recvattrs;
    bslabel_t *senslabel;
    bclear_t *clearance;
    struct sockaddr_in sin;
    t6mask_t rcv_mask;
/* Initialize a mask with all security attribute fields */
    recvmask = T6M_ALL_ATTRS;
    recvattrs = t6alloc_blk(recvmask);
/* Code to set up socket communications */
/* ... */
/* Receive security attributes on the message */
    retval = t6recvfrom(sock, buf, len, 0,(struct sockaddr *) &sin;,
        sizeof (sin), recvattrs, &rcv_mask;);
/* Retrieve security attribute Values */
    senslabel = (bslabel_t *)t6get_attr(T6_SL, recvattrs);
    clearance = (bclear_t *)t6get_attr(T6_CLEARANCE, recvattrs);
}
```

The next example creates *newmask* with no attributes specified, calls the `t6new_attr`(3NSL) routine with a value of `T6_ON`, and calls the `t6recvfrom`(3NSL) routine with *newmask*. This combination tells the `t6recvfrom()` routine to get the security attribute information with the message only when one or more security attributes are different from the set of security attributes on the last message received. The `t6recvfrom()` call returns the full set of security attributes requested; not just the changed security attributes. When security attributes change, the *newmask* value becomes non-zero so you check this value to find out when to look for new security attributes.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    char buf[512];
    int retval, len = sizeof(buf), sock;
    t6mask_t newmask;
    t6attr_t recvattrs;

/* Code to set up socket communications */
/* ... */

/* Create mask to look for change in the sensitivity label */
    newmask = T6M_NO_ATTRS;

/* Turn on new attributes and test for sensitivity label */
    retval = t6new_attr(sock, T6_ON) > 0;
    retval = t6recvfrom(sock, buf, len, 0, 0, 0, recvattrs, &newmask);

    if(newmask > 0)
        {/* Process security attribute information */}
}
```

Examining Attributes

You can retrieve the security attributes for either the next byte of data to be read or the last byte of data read. This example uses the sensitivity label mask to peek at the sensitivity label of the next byte of data and look up the sensitivity label on the last byte of data.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    char buf[512]
    int retval, sock;
    int len = sizeof(buf);
    t6mask_t recvmask;
    t6attr_t recvattrs;

    recvmask = T6M_SL;
    recvattrs = t6alloc_blk(recvmask);

/* Code to set up socket communications */
/* ... */

/* Peek at sensitivity label on next byte of data */
     retval = t6peek_attr(sock, recvattrs, &recvmask);

/* Look up sensitivity label on last byte of data */
     retval = t6last_attr(sock, recvattrs, &recvmask);
}
```

Getting Attribute Size

The `t6size_attr`(3NSL) return value contains the size in bytes of the specified attribute if the call was successful and -1 otherwise. This example gets the size of the clearance attribute in *sendattrs*.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    size_t size;
    t6attr_t sendattrs;
    size = t6size_attr(T6_CLEARANCE, sendattrs);
    printf("Clearance size = %d\n", size);
}
```

The `printf`(1)statement prints the following fixed size for clearances:

```
Clearance size = 36
```

Copying and Duplicating Attribute Structures

The TSIX library provides routines for copying and duplicating an attribute structure. They both do the same thing using different parameter lists. Use the one that meets your application requirements. This example shows the two ways to copy the security attributes in *sendattrs* to *recvattrs*.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    size_t size;
    t6attr_t sendattrs, recvattrs;
    t6mask_t sendmask, recvmask;

    recvmask = T6M_SL;
    recvattrs = t6alloc_blk(recvmask);
    sendmask = T6M_CLEARANCE;
    sendattrs = t6alloc_blk(sendmask);

/* Copy the attributes in sendattrs to recvattrs */
    t6copy_blk(sendattrs, recvattrs);

/* Duplicate the attributes in sendattrs to recvattrs */
    recvattrs = t6dup_blk(sendattrs);
}
```

Compare Attribute Structures

This example compares the *sendattrs* with *recvattrs* for equality.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    int retval;
    t6attr_t sendattrs, recvattrs;
    t6mask_t sendmask, recvmask;

    recvmask = T6M_SL;
    recvattrs = t6alloc_blk(recvmask);
    sendmask = T6M_SESSION_ID;
    sendattrs = t6alloc_blk(sendmask);

    retval = t6cmp_blk(sendattrs, recvattrs);
    printf("Does sendattrs = recvattrs? %d\n", retval);
}
```

The `printf` statement prints the following where 0 means the structures are equal and any non-zero value means they are not.

```
Does sendattrs = recvattrs? 5
```

Clear Attribute Structure

This example clears the session ID attribute value from *recvattrs*. Space is still allocated in the attribute structure, but the attribute values are `NULL`.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    t6attr_t recvattrs;
    t6mask_t recvmask, clrmask;

    recvmask = T6M_ALL_ATTRS; recvattrs = t6alloc_blk(recvmask);
    clrmask = T6M_SESSION_ID;
    t6clear_blk(clrmask, recvattrs);
}
```

Creating Attribute Masks

This example shows three ways to create an attribute mask in addition to instantiating a mask structure and or'ing the desired enumerated fields.

```
#include <tsix/t6attrs.h>
#include <tsol/label.h>

main()
{
    t6mask_t suppmask, allocmask, presentmask;
    t6mask_t getmask, recvmask;
    t6attr_t attrs, getattrs, recvattrs;

    recvmask = T6M_ALL_ATTRS; recvattrs = t6alloc_blk(recvmask);
    getmask = T6M_CLEARANCE; getattrs = t6alloc_blk(getmask);

/* Get mask of system-supported attributes */
    suppmask = t6supported_attrs();

/* Get mask of attributes for which space is */
/* allocated in rcvattrs (T6M_ALL_ATTRS)*/
    allocmask = t6allocated_attrs(rcvattrs);

/* Get mask of attributes present in getattrs */
    presentmask = t6present_attrs(getattrs);
}
```

Free Space

At the end of a program, free all space allocated for variables of type `t6attr_t`.

```
t6free_blk(sendattrs);
t6free_blk(recvattrs);
t6free_blk(getattrs)
t6free_blk(attrs);
```

Client-Server Application

This section presents a short client-server application using Berkeley sockets and the TSIX library to transfer data and security attribute information across the network. The communication path is connection-oriented using the internet domain (TCP/IP). The server is a concurrent process that supplies information about upcoming meetings at different sensitivity levels. To get the service, the client connects to the server and requests the information for a specified sensitivity level.

TCP/IP Server

The server process uses the `net_mac_read` privilege to bind to a multilevel port to serve single-level clients at different sensitivity levels. Chapter 10 describes multilevel and single-level ports.

The *msg_array* structure contains meeting information at Confidential, Secret, Top Secret, and `NULL`. To respond to a single-level client, the server process needs the `proc_set_sl` privilege in its effective set to change the sensitivity label of its child to be the same as the client.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

#include <tsol/label.h>
#include <tsix/t6attrs.h>

struct msg {
    char *sl;
    bslabel_t *bsl;
    char *msg;
} msg_array[] = {
    {"CONFIDENTIAL", NULL, "Staff Meeting at 1:00 pm, Rm 200"},
    {"SECRET", NULL, "Manager Meeting at 10:00 am, Rm 303"},
    {"TOP SECRET", NULL, "Exective Meeting at 3:00 pm, Rm 902"},
    {NULL, NULL, NULL};
};
```

This first part of the main program sets the process clearance to `ADMIN_HIGH` so the child process can set its sensitivity label to the sensitivity label of the requesting client. The `proc_setclr` privilege is needed for this task.

The code comments indicate where privilege bracketing as described in Chapter 3 should take place. With privilege bracketing, the `net_reply_equal` privilege should be off so the server can reply to the client at the sensitivity label specified by the *msg_array* data and not the sensitivity label of the requesting client. The code comments show at what point the `net_repy_equal` privilege must be off for the example to work.

```
main(int argc, char **argv)
{
    int fd, newfd, chpid, index, error;
    struct sockaddr_in serv_addr;
    bclear_t clearance;

    if (argc != 2) {
        printf("Usage: %s host\n", argv[0]);
        exit(1);
    }
    printf("PID = %ld\n", getpid());

    /* Set the process clearance to ADMIN_HIGH
    /* Turn the proc_setclr privilege on in the effective set */

    bclearhigh(&clearance);
    if (setclearance(&clearance) != 0) {
        perror("setclearance");
        exit(1);
    }
    /* Turn the proc_setclr privilege off */
```

This next main program segment creates binary sensitivity label from the data in *msg_array*. The binary labels are used later with the TSIX library routines.

```
    /* Obtain binary labels for run time efficiency */

    index = 0;
    while (msg_array[index].sl != NULL) {
        if ((msg_array[index].bsl =
            (bslabel_t *) malloc(sizeof (bslabel_t))) == NULL) {
            printf("No memory");
            exit (1);
        }
        if (stobsl(msg_array[index].sl, msg_array[index].bsl,
            NEW_LABEL, &error) != 1) {
            printf("converting SL %s failed\n",
            msg_array[index].sl);
            exit(1);
        }
        index++;
    }
```

This next main program segment sets up endpoint communications by creating a socket, binding it to a name, and listening on the socket for client requests. The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

```
    if ((fd = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
    memset(&serv_addr, 0, sizeof (serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
    serv_addr.sin_port = htons(10000);

/* Turn net_mac_read on in the effective set */
    if (bind(fd, (struct sockaddr *) &serv_addr,
        sizeof (serv_addr)) < 0) {
        perror("bind");
        exit(1);
    }
/* Turn the net_mac_read privilege off */

    listen(fd, 5);
```

The while loop accepts client connections on the socket and forks a process to handle each client request. The forked process prepares structures to receive the incoming message and its sensitivity label, and to set the sensitivity label portion of the process CMW label to the incoming sensitivity label. It also allocates *handle_in* with enough space to receive the sensitivity label on the incoming message, allocates *handle_out* with enough space to send a sensitivity label with the outgoing message, and receives the message and security attribute information with the `t6recvfrom`(3NSL) routine.

```
    while (1) {
        if ((newfd = accept(fd, NULL, 0)) < 0) {
            perror("accept");
            exit(1);
        }
        printf("Request Received\n");
        if ((chpid = fork()) < 0) {
            perror("fork");
            exit(1);
        } else if (chpid == 0) {        /* child process */
            t6attr_t handle_in;
            t6attr_t handle_out;
            t6mask_t mask_in = T6M_SL;
            int client_index = 0;
            bslabel_t *bsl;
            bclabel_t bcmwlabel;
            char buf[256];
            int index, buflen = 256;
            t6mask_t new_mask = T6M_NO_ATTRS;
            char *string = (char *) 0;
            char any;
            close(fd);
            printf("child PID = %ld\n", getpid());

/* Process client request */
            if ((handle_in = t6alloc_blk(mask_in)) == NULL) {
                printf("t6attr_alloc: no memory");
                exit(1);
            }
            if (t6recvfrom(newfd, buf, buflen, 0, 0, 0,
                handle_in, &new_mask) < 0) {
                perror("t6recvfrom");
                exit(1);
            }
```

The last main program segment extracts the sensitivity label received, sets the sensitivity label of the process to that of the client, and sends the reply to the client. The code comments indicate where privilege bracketing as described in Chapter 3 should take place.

```
    /* Get sensitivity label */
        if ((bsl = (bslabel_t *) t6get_attr(T6_SL,
                handle_in)) == NULL) {
                printf("t6get_attr: no SL available");
                exit(1);
        }
        if (bsltos(bsl, &string, 0, LONG_WORDS) < 0) {
                perror("bsltos");
                exit(1);
        }
        printf("Requestor’s SL = %s\n", string);

    /* Set the sensitivity label of the child process to */
    /* that of the client */
        if (getcmwplabel(&bcmwlabel) != 0) {
                perror("getcmwplabel");
                exit(1);
        }
        setcsl(&bcmwlabel, bsl);

    /* Turn proc_set_sl on in the effective set */
        if (setcmwplabel(&bcmwlabel, SETCL_SL) < 0) {
                perror("setcmwplabel");
                exit (1);
        }
    /* Turn the proc_set_sl privilege off */

    /*
     * Retreive the msg_array entry that matches the sensitivity
     * label of the client.
     */
        while (msg_array[client_index].sl != NULL) {
            if (blequal(bsl, msg_array[client_index].bsl)
            break;
            client_index++;
        }
        if (msg_array[client_index].sl == NULL) {
            perror("No message for the client's SL");
            exit (1);
        }
        send(new_fd, msg_array[client_index].msg,
            strlen(msg_array[client_index].msg));

        exit (0);
```

TCP/IP Client

To request the service, the client program connects to the server, sends a request, and waits for the meeting message. If the connection is closed before a message is received, the client exits because there is no meeting at its sensitivity label. If a message is received, the client uses `t6recvfrom`(3NSL) to obtain the message. Code to process the information is not shown in the example.

This first part of the program sets up data structures for the client request and server response.

```
#include <stdio.h>
#include <sys/types.h>
#include <fcntl.h>
#include <sys/socket.h>
#include <string.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <tsol/label.h>
#include <tsix/t6attrs.h>

char *clnt_req = "Request Meeting Info";

main(int argc, char **argv)
{
    int sock, retval;
    char buf[256];
    int buflen = 256;
    int num;
    struct sockaddr_in serv_addr;
    struct hostent *hostent;
    bslabel_t *bsl;
    t6mask_t new_mask, sl_mask = T6M_SL;
    t6attr_t handle;
    char    *string = (char *)0;
```

This next main program segment processes the command-line *argc* and *argv* inputs to get the host name and port number of the server and establishes a connection.

```
    if (argc != 2) {
        printf("Usage: %s host\n", argv[0]);
        exit (1);
    }
    if ((hostent = gethostbyname(argv[1])) == NULL) {
        perror("gethostbyname");
        exit(1);
    }

    memset((void *) &serv_addr, 0, sizeof (serv_addr));
    serv_addr.sin_family = AF_INET;
    serv_addr.sin_port = htons(10000);
    memcpy((void *) &serv_addr.sin_addr,
        (void *) hostent->h_addr_list[0], hostent->h_length);

    if ((sock = socket(AF_INET, SOCK_STREAM, 0)) < 0) {
        perror("socket");
        exit(1);
    }
    if (connect(sock, (struct sockaddr *)&serv_addr,
        sizeof (serv_addr)) < 0) {
        perror("connect");
        exit(1);
    }
    printf("Connected\n");
    if ((handle = t6alloc_blk(sl_mask)) == NULL) {
        printf("t6attr_alloc: no memory");
        exit(1);
    }
```

This next main program segment sends the request to the server. The request is sent at the sensitivity label at which the client process is executing. When the server processes the request, it sends back meeting information for the sensitivity label at which the request is made only. The `t6recvfrom`(3NSL) routine receives the meeting information.

```
/* Send a request to server */
    write(sock, clnt_req, strlen(clnt_req));

    if ((num = t6recvfrom(sock, buf, buflen, 0, 0, 0, handle,
        &new_mask)) < 0) {
        perror("t6recvfrom");
        exit (1);
    } else if (num == 0) {
        printf("Connection closed, nothing matches.\n");
        exit(0);
    } else
        printf("Received Reply\n");

    retval = bsltos(bsl, &string, 0, LONG_WORDS);
    printf("Retval = %d, Sensitivity label = %s\n", retval, string);
    printf("Message = %s\n", buf);
}
```

Running the Programs

The server process starts and waits for a client request.

```
phoenix% serverProgram phoenix
PID = 655
```

When the client process is started at Confidential, the `printf` statements in the client print the following. The sensitivity label of the server does not matter because it is a multilevel connection.

```
phoenix% clientProgram phoenix
Received Reply
Message = Staff Meeting at 1:00 pm, Rm 200
```

The server process prints the following after fulfilling the client request:

```
Request Received
child PID = 657
Requestor's SL = C
Attributes List (alloc_mask = 0x00000040, attr_mask = 0x00000040):
child: exiting
```

In the following example, the client process is instead started at Secret.

```
phoenix% clientProgram phoenix
Received Reply
Message = Manager Meeting at 10:00 am, Rm 303
```

The server process prints the following after fulfilling the client request:

```
Request Received
child PID = 661
Requestor's SL = S
Attributes List (alloc_mask = 0x00000040, attr_mask = 0x00000040):
child: exiting
```

CHAPTER 13

Remote Procedure Calls

The Trusted Solaris remote procedure call (RPC) mechanism is built on Berkeley internet sockets and the Trusted Security Information Exchange (TSIX) library, and supports Transport Layer Interface (TLI). Trusted Solaris modifications to RPC enable a server process to receive security attribute information on incoming client requests, and change security attribute information on an outgoing response to a client. Chapter 12 describes the privileges required to change security attribute information on messages.

In addition, mappings have been extended to include sensitivity labels to separate and protect mappings according to their sensitivity label.

Mapping

Mapping is a relationship maintained by the RPC binder service between an ordered triple (program number, version number, and network ID) and a service address on a machine serviced by the RPC binder. The current set of mappings represents the available registered RPC services on a host. The Trusted Solaris environment supports single-level mapping and multilevel mapping.

- Program number – A number assigned to identify the remote procedure.
- Version number – The version number of the remote procedure.

- Network ID – The network transport to which the network address refers.

Single-Level Mapping

A single-level mapping is a mapping the RPC binder service advertises only to clients that have the same sensitivity label as the server that created the mapping.

Multilevel Mapping

A multilevel mapping (MLM) is a mapping the RPC binder service advertises to all clients regardless of their sensitivity label. A multilevel mapping is created when a server has the `net_mac_read` privilege in its effective set when it makes the RPC library call to register the service with the RPC binder service.

Multilevel Ports

A multilevel port is created when a server has the `net_mac_read` privilege in its effective set when it makes the RPC library call to create the port. See "Multilevel Ports" on page 210 in Chapter 10 for a discussion of multilevel ports.

Security Attributes

The server handle for RPC library calls is a pointer to an `SVCXPRT` data structure, and the client handle for RPC library calls is a pointer to a `CLIENT` data structure. In the Trusted Solaris environment, both structures have additional fields that point to security attribute information.

The security attributes pointed to by the server and client handles are based on the TSIX library. See Chapter 12 for information on the library routines and privileges required to change security attributes.

Note – The caller must free all memory blocks allocated for security attribute pointers.

Servers

The following security attribute fields of the SVCXPRT structure can be accessed directly by the server process:

```
t6attr_t xp_tsol_incoming_attrsp
t6attr_t xp_tsol_outgoing_attrsp
t6mask_t xp_tsol_incoming_new_attrs
```

A server can receive one or more security attributes of incoming client requests by using t6alloc_blk(3NSL) to allocate an opaque structure with space for the security attributes and setting xp_tsol_incoming_attrsp field in the SVCXPRT structure to point to the security attribute structure.

A privileged server can set security attributes on a request to the server by using t6alloc_blk(3NSL) to allocate an opaque structure with space for the security attributes and setting xp_tsol_outgoing_attrsp field in the SVCXPRT to point to the security attribute structure. The RPC library routines pick up the attributes and send them as the attributes for the response.

A server can examine the security attributes on the next and last bytes of data by using the xp_tsol_incoming_new_attrs field in the SVCXPRT structure to point to specific incoming attributes to be examined.

Clients

The following security attribute fields of the CLIENT structure can be accessed directly by the client process:

```
t6attr_t cl_tsol_incoming_attrsp
t6attr_t cl_tsol_outgoing_attrsp
```

A client can receive one or more security attributes of incoming server responses by using t6alloc_blk(3NSL) to allocate an opaque structure with space for the attributes and setting cl_tsol_incoming_attrsp field in the CLIENT structure to point to the security attribute structure.

A privileged client can set security attributes on a request to the server by using t6alloc_blk(3NSL) to allocate an opaque structure with space for the security attributes and setting xp_tsol_outgoing_attrsp field in the CLIENT structure to point to the security attribute structure. The RPC library routines pick up the attributes and send them as the attributes for the response.

Header Files and Libraries

The following header file is necessary to use the RPC programming interfaces.

```
#include <rpc/rpc.h>
```

The examples in this chapter compile with the following libraries:

```
-DTSOL -lt6 -lnsl -lsocket -ltsol
```

Programming Interfaces

The Trusted Solaris environment introduces the `rpcb_getallmaps()` routine in the `rpcbind()` interface. This section lists the RPC man pages that have information specific to the Trusted Solaris environment added.

- `rpc`(3NSL)
- `rpc_clnt_calls`(3NSL)
- `rpc_svc_calls`(3NSL)
- `rpc_clnt_create`(3NSL)
- `rpc_svc_create`(3NSL)
- `rpc_svc_reg`(3NSL)
- `rpcbind`(3NSL)
- `rpcbind`(1M)
- `rpcinfo`(1M)

Client-Server Application

This is a simple client-server application to show how security attributes are sent and received with RPC library routines. Command line arguments supply the server name and a user ID, and the server process retrieves the user ID sent by the client, multiplies the input by 2, and sends the result to the client. To run the programs, compile them with the libraries listed in "Header Files and Libraries" on page 212.

Header File

The following header file `rpc_test.h` is required for the example application to compile.

```
#include <rpc/rpc.h>
#include <rpc/types.h>
#define RPC_TEST_PROG ((u_long)1234567890)
#define RPC_TEST_VERS ((u_long)1)
#define RPC_TEST_DOUBLE1 ((u_long)1)
#define RPC_TEST_EXIT1 ((u_long)2)
```

Client Program

This part of the client program accepts command line inputs and creates a client handle.

```
#include <stdio.h>
#include <stdlib.h>
#include <rpc/rpc.h>
#include <netdb.h>
#include <tsix/t6attrs.h>
#include "rpc_test.h"

extern int
main(int argc, char *argv[])
{
    struct timeval time_out;
    CLIENT *handlep;
    enum clnt_stat stat;
    int input, output;
    uid_t uid;
    if (argc < 2 || argc > 3) {
        fprintf(stderr,
            "Usage: simple_rpc_clnt_test HOSTNAME [UID]\n");
        exit(1);
    }

    handlep = clnt_create(argv[1], RPC_TEST_PROG,
        RPC_TEST_VERS, "udp");
        if (handlep == (CLIENT *) NULL) {
            fprintf(stderr, "Couldn't create client%s.\n",
                clnt_spcreateerror(""));
        exit(1);
    }
```

This part of the client program sets the client handle to point to the space allocated for the user ID to be input from the command line, sets the user ID value, sends the value to the server process, and waits for the server response. The client prints out the server response before it exits.

The client program needs the `net_setid` privilege in its effective set to send a changed outgoing user ID. The code comments indicate where privilege bracketing should occur.

```
    if (argc == 3) {
        handlep->cl_tsol_outgoing_attrsp = t6alloc_blk(T6M_UID);
        if (handlep->cl_tsol_outgoing_attrsp == NULL) {
            fprintf(stderr, "Can't create attr buffer\n");
            exit(1);
        }

        printf ("Sending UID %s\n", argv[2]);
        uid = atoi(argv[2]);
        if (t6set_attr(T6_UID, &uid,
            handlep->cl_tsol_outgoing_attrsp) != 0) {
            fprintf(stderr, "Error returned by t6set_attr.\n");
            exit(1);
        }
    }
    time_out.tv_sec = 30;
    time_out.tv_usec = 0;
    input = 3;

/* Turn net_uid on in the effective set */
    stat = clnt_call(handlep, RPC_TEST_DOUBLE1, xdr_int,
        (caddr_t) &input, xdr_int, (caddr_t) &output, time_out);
    if (stat != RPC_SUCCESS) {
        fprintf(stderr, "Call failed. %s.\n",
            clnt_sperror(handlep, ""));
            exit(1);
    }
/* Turn off the net_uid privilege */

    printf("Response received: %d\n", output);
    (void) clnt_destroy(handlep);

    return (0);
}
```

Server Program

The server program sets the server handle to point to the space allocated space for all security attributes.

```
#include <stdio.h>
#include <stdlib.h>
#include <rpc/rpc.h>
#include <tsix/t6attrs.h>
#include "rpc_test.h"
static void proc_1(struct svc_req *rqstp, SVCXPRT *transp);
extern int
```

```
main(int argc, char *argv[])
{
    SVCXPRT *handlep;
    struct netconfig *netconfigp;
    netconfigp = getnetconfigent("udp");
    if (netconfigp == NULL) {
        fprintf(stderr, "Cannot find netconfig entry for udp.\n");
        exit(1);
    }

    handlep = svc_tp_create(proc_1, RPC_TEST_PROG,
        RPC_TEST_VERS, netconfigp);

    if (handlep == NULL) {
        fprintf(stderr, "Cannot create service.\n");
        exit(1);
    }
    freenetconfigent(netconfigp);
    handlep->xp_tsol_incoming_attrsp = t6alloc_blk(T6M_ALL_ATTRS);
    if (handlep->xp_tsol_incoming_attrsp == NULL) {
        fprintf(stderr, "Can't create attr buffer\n");
        exit(1);
    }
    svc_run();
    return (0);
}
```

Remote Procedure

The remote procedure receives the user ID from command line arguments, and multiplies the input by 2, sends the result to the client and prints the response before exiting.

```
static void
proc_1(struct svc_req *rqstp, SVCXPRT *handlep)
{
    int input;
    int result;
    uid_t *uidp;

    switch(rqstp->rq_proc) {
    case NULLPROC:
            svc_sendreply(handlep, xdr_void, NULL);
            break;
    case RPC_TEST_DOUBLE1:
            if (!svc_getargs(handlep, xdr_int, (caddr_t) &input)) {
                fprintf(stderr, "Error from svc_getargs\n");
                svcerr_systemerr(handlep);
            }
            uidp = (uid_t *) t6get_attr(T6_UID,
                handlep->xp_tsol_incoming_attrsp);
            if (uidp == NULL)
                fprintf(stderr, "Error from t6get_attr.\n");
            else printf("Client's UID is %d\n", *uidp);
            result = 2 * input;
            if (!svc_sendreply(handlep, xdr_int, (caddr_t) &result)) {
                fprintf(stderr, "Error from sendreply\n");
                svcerr_systemerr(handlep);
            }
            svc_freeargs(handlep, xdr_int, (caddr_t) &input);
            break;

    default:
            fprintf(stderr, "Call to unexpected procedure number %d\n",
                rqstp->rq_proc);
        svcerr_noproc(handlep);
        break;
    }
}
```

Running the Simple Application

The client process takes the server host name and a user ID as input parameters and prints that it is sending the specified user ID:

```
owl%  phoenix
phoenix% owl 2570
Sending UID 2570
```

The server retrieves the user ID and prints it out as follows:

```
Client's UID is 2570
```

The client process prints the server response and then exits:

```
Response received: 6
phoenix%
```

CHAPTER 14

Trusted X Window System

This chapter uses a short Motif application to describe Trusted X Window System security policy and the Trusted Solaris interfaces.

X Windows Environment

The Trusted Solaris environment uses the Trusted Common Desktop Environment (CDE) which is an enhanced version of CDE 1.0.2. Trusted CDE uses the X Window System, Version 11, with the Trusted Solaris X Window System server. The Trusted X Window System server has protocol extensions to support mandatory access controls, discretionary access controls, and the use of privileges. Clients connect to the Trusted X Window System server over UNIX domain and TCP/IP domain network connections.

Data transfer sessions are instantiated at different sensitivity labels and user IDs (polyinstantiated). This is so data in an unprivileged client at one sensitivity label or user ID is not transferred to another client at another sensitivity label or user ID in violation of the Trusted X Window System discretionary access controls and mandatory access policies of write-equal and read-down.

Trusted Solaris X Window System programming interfaces let you get and set security-related attribute information and translate binary labels to text using a font list and width to apply a style such as Helvetica 14 point bold to the text string output. These interfaces are usually called by administrative applications written with Motif widgets, Xt Intrinsics, Xlib, and CDE interfaces.

- Getting security-related information – These interfaces operate at the Xlib level, which make X protocol requests. You use Xlib interfaces to obtain data for the input parameter values.
- Translating labels from binary to text – These interfaces operate at the Motif level. The input parameters are the binary label, a font list to specify the appearance of the output string, and the desired width. A compound string using of the specified style and width is returned.

Security Attributes

The Trusted X Window System interfaces manage security-related attribute information for various X Window objects. If your application GUI is created with Motif only, you need to use XToolkit routines within the Motif application to retrieve the Xlib object IDs underlying the Motif widgets to handle security attribute information for an Xlib object.

The X Window objects for which security attribute information can be retrieved by the Trusted X Window System interfaces are window, property, X Window Server, and the connection between the client and the X Window Server. Xlib provides calls to retrieve window, property, display, and client connection IDs.

- Windows – Present output to the end user and accept input from clients.
- Properties – A property is an arbitrary collection of data accessed by the property name. Property names and property types can be referenced by an atom, which is a 32-bit unique identifier and a character name string.

The security attributes for windows, properties, and client connections consist of ownership IDs and CMW label information. See “Data Types, Header Files, and Libraries” on page 222 for information on the structures for capturing some of these attributes, and “Programming Interface Declarations” on page 224 for information on the interfaces that get and set security attribute information.

Security Policy

Window, property, and pixmap objects have a user ID, client ID, and a CMW label. Graphic contexts, fonts, and cursors have a client ID only. The connection between the client and the X Window Server has a user ID, X Window Server ID, and a CMW label.

The user ID is the ID of the client that created the object. The client ID is related to the connection number to which the client that creates the object is connected.

The discretionary access policy requires a client to own an object to perform any operations on the object. A client owns an object when the client's user ID equals the object's ID. For a connection request, the user ID of the client must be in the Access Control List (ACL) of the owner of the X Window Server workstation or the client must assert the Trusted Path attribute as described in "Get and Set Process Security Attribute Flags" on page 50.

The mandatory access policy is write-equal, read-equal for naming windows, and read-down for properties. The sensitivity label portion of the CMW label is set to the sensitivity label of the creating client. The information label portion of the CMW label is always `ADMIN_LOW`.

- Modify, create, or delete – The sensitivity label of the client must equal the object's sensitivity label.
- Name, read, or retrieve – The client's sensitivity label must dominate the object's sensitivity label.
- Connection request – The sensitivity label of the client must be dominated by the session clearance of the owner of the X Window Server workstation or the client must assert the Trusted Path attribute as described in "Get and Set Process Security Attribute Flags" on page 50

Windows can have properties that contain information to be shared among clients. Window properties are created at the sensitivity label at which the application is running so access to the property data is segregated by its sensitivity label. clients can create properties, store data in a property on a window, and retrieve the data from a property subject to mandatory and discretionary access restrictions. See `/usr/openwin/server/tsol/property.atoms` to specify properties that are not polyinstantiated.

Root Window

The root window is at the top of the window hierarchy. The root window is a public object that does not belong to any client, but has data that must be protected. The root window attributes are protected at `ADMIN_LOW`.

Client Windows

A client usually has at least one top-level client window that descends from the root window, and additional windows nested within the top-level window. All windows that descend from the client's top-level window have the same sensitivity label.

Override-Redirect Windows

Override-redirect windows such as menus and certain dialog boxes cannot take the input focus away from another client to prevent the input focus from accepting input into a file at the wrong sensitivity label. Override-redirect windows are owned by the creating client and cannot be used by other clients to access data at another sensitivity label.

Keyboard, Pointer, and Server Control

A client needs mandatory and discretionary access to gain keyboard, pointer, or server control. To reset the focus, a client must own the focus or have the `win_devices` privilege.

To warp a pointer, the client needs pointer control and mandatory and discretionary access to the destination window. X and Y coordinate information can be obtained for events that involve explicit user action.

Selection Manager

The Selection Manager arbitrates user-level inter-window data moves such as cut-and-paste or drag-and-drop where information is transferred between untrusted windows. When a transfer is attempted, Selection Manager captures the transfer, verifies the controlling user's authorization, and requests confirmation and labeling information from the user. The Selection Manager displays whenever the end user attempts a data move without your writing application code.

The administrator can set autoconfirm for some transfer types in which case the Selection Manager does not appear. If the transfer meets mandatory and discretionary access policies, the data transfer completes. The File Manager and Window Manager also act as selection agents for their private drop sites. See `/usr/openwin/server/tsol/selection.atoms` to specify selection targets that are polyinstantiated. See `/usr/dt/config/sel_config` to determine which selection targets are automatically confirmed.

Default Resources

Resources not created by clients are default resources labeled `ADMIN_LOW. Only` clients running at `ADMIN_LOW` or with the appropriate privileges can modify default resources.

- Root window attributes – All clients have read and create access, but only privileged clients have write or modify access. See “Privileged Operations” on page 221.
- Default cursor – Clients are free to reference the default cursor in protocol requests.
- Predefined atoms – The `/usr/openwin/server/tsol/public.atoms` file contains a read-only list of predefined atoms.

Moving Data Between Windows

A client needs the `win_selection` privilege to move data between one window and another without going through the “Selection Manager” on page 220.

Getting and setting process attribute flags is covered in Chapter 2.

Privileged Operations

Library routines that access a window, property or atom name without user involvement require mandatory and discretionary access. Library routines that access framebuffer graphic contexts, fonts, and cursors require discretionary access and may also require additional privilege for special tasks as described below.

The client may need one or more of the following privileges in its effective set if access to the object is denied: `win_dac_read`, `win_dac_write`, `win_mac_read`, or `win_mac_write`. See `/usr/openwin/server/tsol/config.privs` to enable or disable these policies..

Configuring and Destroying Resources

A client needs the `win_config` privilege in its effective set to configure or destroy windows or properties permanently retained by the X Window Server. The screen saver timeout is an example of such a resource.

Input Devices

A client needs the `win_devices` privilege in its effective set to get and set keyboard and pointer controls or modify pointer button and key mappings.

Direct Graphics Access

A client needs the `win_dga` privilege in its effective set to use the direct graphics access (DGA) X protocol extension.

Downgrading labels

A client needs the `win_downgrade_sl` privilege in its effective set to change the sensitivity label on a window, pixmap, or property to a new label that does not dominate the existing label.

Upgrading Labels

A client process needs the `win_upgrade_sl` privilege in its effective set to change the sensitivity label on a window, pixmap, or property to a new label that dominates the existing label.

Setting a Font Path

A client needs the `win_fontpath` privilege in its effective set to modify the font path.

Data Types, Header Files, and Libraries

To use the Trusted X11 programming interfaces described in this chapter, you need the following header files:

```
#include <tsol/Xtsol.h>
```

The Trusted X11 examples compile with the following library:

```
-lXtsol -ltsol
```

To use the X11 Windows label clipping programming interfaces described in this chapter, you need the following header file:

```
#include <tsol/label_clipping.h>
```

The label clipping examples compile with the following library:

```
-lDtTsol -ltsol
```

Object Type

The `ResourceType` type definition indicates the type of resource to be handled. The value can *IsWindow*.

Object Attributes

The `XTsolResAttributes` structure contains the resource attributes.

`CARD32`	`ouid`	User ID of workstation server owner
`CARD32`	`uid`	User ID of window
`bslabel_t`	`sl`	Sensitivity label

Property Attributes

The XTsolPropAttributes structure contains the property attributes.

`CARD32`	`uid`	User ID of property
`bslabel_t`	`sl`	Sensitivity label

Client Attributes

The `XTsolClientAttributes` structure contains the client attributes.

`uid_t`	`uid`	ID of user that started the client.
`gid_t`	`gid`	Group ID

`pid_t`	`pid`	Process ID
`u_long`	`sessionid`	Session ID
au_id_t	auditid	Audit ID
`u_long`	`iaddr`	Internet address of workstation where the client is running.

Setting Flag

The `setting_flag` type definition defines CMW label flag values as follows:

`SETCL_SL` – Set the sensitivity label portion of the CMW label. `SETCL_ALL` – Set the entire CMW label.

CMW Label

A data structure to represent a binary CMW label. Interfaces accept and return a binary CMW label in a structure of type `bclabel_t`.

Clearance

A type definition to represent a clearance. Interfaces accept as parameters and return binary clearances in a structure of type `bclear_t`.

Programming Interface Declarations

This section provides declarations for the Trusted X11 interfaces and the X11 Windows label clipping interfaces.

Window Attributes

This routine returns the resource attributes for a window ID in **resattrp*. Refer to the `XTSOLgetResAttributes`(3) man page.

```
Status XTSOLgetResAttributes(Display *display,
    XID object,
    ResourceType resourceFlag,
    XTsolResAttributes *resattrp);
```

Property Attributes

This routine returns the property attributes for a property hanging on a window ID in **propattrp*. Refer to the `XTSOLgetPropAttributes`(3) man page.

```
Status XTSOLgetPropAttributes(Display *display,
    Window win,
    Atom property,
    XTsolPropAttributes *propattrp);
```

Client Connection Attributes

This routine returns the client attributes in **clientattrp*. Refer to the `XTSOLgetClientAttributes`(3) man page.

```
Status XTSOLgetClientAttributes(Display *display,
    XID win,
    XTsolClientAttributes *clientattrp);
```

Window CMW Label

These routines get and set the CMW label of a window. Refer to the `XTSOLgetResLabel`(3) and `XTSOLsetResLabel`(3) man pages.

```
Status XTSOLgetResLabel(Display *display,
    XID object,
    ResourceType resourceFlag,
    bclabel_t *cmwlabel);

void XTSOLsetResLabel(Display *display,
    XID object,
    ResourceType resourceFlag,
    bclabel_t *cmwLabel,
    enum setting_flag labelFlag);
```

Window User ID

These interfaces get and set the user ID of a window. Refer to the XTSOLgetResUID(3) and XTSOLsetResUID(3) man pages.

```
Status XTSOLgetResUID(Display *display,
    XID object,
    ResourceType resourceFlag,
    uid_t *uidp);

void XTSOLsetResUID(Display *display,
    XID object,
    ResourceType resourceFlag,
    uid_t *uidp);
```

Property CMW Label

These routines get and set the CMW label of a property hanging on a window. Refer to the XTSOLgetPropLabel(3) and XTSOLsetPropLabel(3) man page.

```
Status XTSOLgetPropLabel(Display *display,
    Window win,
    Atom property,
    bclabel_t *cmwlabel);

void XTSOLsetPropLabel(Display *display,
    Window win,
    Atom property,
    bclabel_t *cmwLabel,
    enum setting_flag labelFlag);
```

Property User ID

These interfaces get and set the user ID of a property hanging on a window. Refer to the XTSOLgetPropUID(3) and XTSOLsetPropUID(3) man pages.

```
Status XTSOLgetPropUID(Display *display,
    Window winID,
    Atom property,
    uid_t *uidp);

void XTSOLsetPropUID(Display *display,
    Window win,
    Atom property,
    uid_t *uidp);
```

Workstation Owner ID

These routines get and set the user ID for the owner of the workstation server. Refer to the `XTSOLgetWorkstationOwner`(3) and `XTSOLsetWorkstationOwner`(3) man pages.

Note – `XTSOLsetWorkstationOwner`(3) is reserved for the Window Manager.

```
Status XTSOLgetWorkstationOwner(Display *display, uid_t *uidp);
void XTSOLsetWorkstationOwner(Display *display, uid_t *uidp);
```

X Window Server Clearance and Minimum Label

These routines set the session high clearance and the session low minimum label for the X Window Server. Refer to the `XTSOLsetSessionHI`(3) and `XTSOLsetSessionLO`(3) man pages.

- The session high clearance is set from the workstation owner's clearance at login, and must be dominated by the owner's clearance and the upper bound of the machine monitor's label range. Once changed, connection requests from clients running at a sensitivity label higher than the window server clearance are rejected unless they have privilege.
- The session low minimum label is set from the workstation owner's minimum label at login and must be greater than the user's administratively set minimum label and the lower bound of the machine monitor's label range. Once changed, connection requests from clients running at a sensitivity label lower than the window server sensitivity label are rejected unless they have privilege.

Note – These interfaces are reserved for the Window Manager.

```
void XTSOLsetSessionHI(Display *display, bclear_t *clearance);
void XTSOLsetSessionLO(Display *display, bslabel_t *sl);
```

Trusted Path Window

These routines makes the specified window the trusted path window and test whether the specified window is the trusted path window. Refer to the `XTSOLMakeTPWindow`(3) man page.

```
void XTSOLMakeTPWindow(Display *dpy, Window win);
Bool XTSOLIsWindowTrusted(Display *display, Window win);
```

Screen Stripe Height

These interfaces get and set the screen stripe height – an additive and subtractive operation. Be careful you do not end up with no screen stripe or a very large screen stripe. Refer to the `XTSOLsetSSHeight`(3) and `XTSOLgetSSHeight`(3) man pages.

Note – These interfaces are reserved for the Window Manager.

```
Status XTSOLgetSSHeight(Display *display,
    int screen_num,
    int *newHeight);

void XTSOLsetSSHeight(Display *display,
    int screen_num,
    int newHeight);
```

Polyinstantiation Information

This routine lets a client get property information from a property at a different sensitivity label from the client. In the first call, specify the desired sensitivity label and user ID, and set *enabled* to True. Then call `XTSOLgetPropAttributes`(3), `XTSOLgetPropLabel`(3), or `XTSOLgetPropUID`(3), and finish up by calling this routine again with *enabled* set to False. Refer to the `XTSOLsetPolyInstInfo`(3) man page.

```
void XTSOLsetPolyInstInfo(Display *dpy,
    bslabel_t *senslabel,
    uid_t *userID, int enabled);
```

X11 Windows Label Clipping Interfaces

These routines translate a binary CMW label, sensitivity label, or clearance to a compound string using a font list. The returned string is clipped to the specified pixel width, or if *width* equals the display width (*display*), the label is word wrapped using a width of half the display width. See “Binary and Text Label Translation” on page 113 for a description of the *flags* parameter. Refer to the `labelclipping`(3TSOL) man page.

```
/* CMW label */
    XmString Xbcltos(Display *display,
        const bclabel_t *cmwlabel,
        const Dimension width,
        const XmFontList fontlist,
        const int flags);

/* Sensitivity label */
```

```
    XmString Xbsltos(Display *display,
        const bslabel_t *senslabel,
        const Dimension width,
        const XmFontList fontlist,
        const int flags);

/* Clearance */
    XmString Xbcleartos(Display *display,
        const bclear_t *clearance,
        const Dimension width,
        const XmFontList fontlist,
        const int flags);
```

Example Motif Application

The example Motif application in the following figure launches `xclock` or `xterm` applications. It is simple because its purpose is to show how Trusted Solaris X Windows programming interfaces are called from within a Motif application. The application's process sensitivity label is Confidential and the information label is ADMIN_LOW.

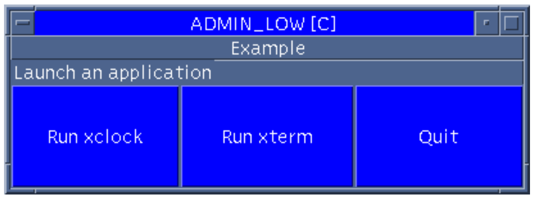

FIGURE 14–1 Simple Motif Application

The next headings provide example code segments that use the Trusted Solaris interface calls to handle security attributes and translate a binary label to text with a font list. The code segments focus on handling window security attributes because those are the most common operations in application programs. Often a client will retrieve security attributes (using the appropriate privileges) for an object created by another application and check the attributes to determine if an operation on the object is permitted by the system's discretionary ownership policies and the mandatory write-equal and read-down policies. If access is denied, the application raises an error or uses privilege as appropriate. See "Privileged Operations" on page 221 for information on when privileges are needed.

The source code for the simple Motif application including the code segments below is provided in "Code" on page 234. Xlib calls to retrieve object IDs to pass to the Trusted

Solaris programming interfaces should be made after the appropriate object has been created so there is an ID to retrieve. In this source code, the Xlib calls are after `XtRealizeWidget()` is called.

Getting Window Attributes

The `XTSOLgetPropAttributes`(3) routine returns security-related attributes for a window. You supply the display and window IDs, a flag to indicate the object you want security attributes on is a window, and an `XtsolResAttributes` structure to receive the returned attributes. The client is getting the security attributes for a window it created so no privileges are required.

```
/* Retrieve underlying window and display IDs with Xlib calls */
    window = XtWindow(topLevel);
    display = XtDisplay(topLevel);

/* Retrieve window security attributes */
    retval = XTSOLgetResAttributes(display, window, IsWindow, &winattrs);

/* Translate labels to strings */
    retval = bsltos(&winattrs.sl, &string1, 0, LONG_WORDS);

/* Print security attribute information */
    printf("Workstation owner ID = %d, User ID = %d, Label = %s\n",
   winattrs.ouid, winattrs.uid,string1, string2,    string3);
```

The `printf`(1) statement prints the following:

```
Workstation owner ID = 29378
User ID = 29378
Label = CONFIDENTIAL
```

Translate Label with Font List

This example gets the process sensitivity label and translates it to text using a font list and pixel width. A label widget is created with the string for its label. The process sensitivity label equals the window sensitivity label so no privileges are required.

When the final string is longer than the width, it is clipped and the clipped indicator is used. The clipped indicator for a clipped sensitivity label is described in "Sensitivity and Information Labels" on page 117 and on the `sbsltos`(3TSOL) man page. Note that the X Window System label translation interfaces clip to the number of pixels specified, and the label clipping interfaces clip to the number of characters.

If your site uses a `label_encodings` file in a language other than English, the translation might not work on accent characters in the ISO standard above 128, and will not work on the Asian character set.

```
    retval = getcmwplabel(&cmwlabel);
    getcsl(&senslabel, &cmwlabel);

/* Create the font list and translate the label using it */
    italic = XLoadQueryFont(XtDisplay(topLevel),
        "-adobe-times-medium-i-*-*-14-*-*-*-*-*-iso8859-1");
    fontlist = XmFontListCreate(italic, "italic");
    xmstr = Xbsltos(XtDisplay(topLevel), &senslabel, width, fontlist,
        LONG_WORDS);

/* Create a label widget using the font list and label text*/
    i=0;
    XtSetArg(args[i], XmNfontList, fontlist); i++;
    XtSetArg(args[i], XmNlabelString, xmstr); i++;
    label = XtCreateManagedWidget("label", xmLabelWidgetClass,
        form, args, i);
```

The source code for the italicized sensitivity label string and the non-italicized "Launch and application" label is in "Code" on page 234. Launch the application with any command line argument to see the italicized sensitivity label string in the label widget as shown in the following figure.

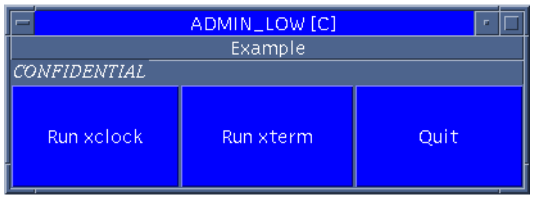

FIGURE 14–2 Italicized Label Text

Getting a Window CMW Label

This example gets the CMW label on a window. The process sensitivity label equals the window sensitivity label so no privileges are required.

```
/* Retrieve window CMW label */
    retval = XTSOLgetResLabel(display, window, IsWindow, &cmwlabel);
```

```
/* Translate labels to string and print */
    retval = bcltos(&cmwlabel, &string, 0, LONG_WORDS);
    printf("CWM label = %s\n", string);
```

The `printf`(1)statement prints the following:

```
CMW label = ADMIN_LOW[C]
```

Setting a Window CMW Label

This example sets the CMW label on a window. The new sensitivity label dominates the window's and process's sensitivity label. The client needs the `sys_trans_label` privilege in its effective set to translate a label it does not dominate, and the `win_upgrade_sl` privilege to change the window sensitivity label.

```
/* Translate text string to binary sensitivity label and */
/* Turn sys_trans_label on in the effective set */
    retval = stobsl(&string4, &senslabel, NEW_LABEL, &error);
/* Turn sys_trans_label off */

/* Set the sensitivity label in the cmwlabel structure */
    setcsl(&cmwlabel, &senslabel);

/* Set sensitivity label portion of CMW label with new value */
/* and turn win_upgrade_sl on in the effective set */
    retval = XTSOLsetResLabel(display, window, IsWindow, &cmwlabel,
        SETCL_SL);
/* Turn the win_upgrade_sl privilege off */
```

Getting the Window User ID

This example gets the window user ID. The process owns the window resource and is running at the same sensitivity label so no privileges are required.

```
/* Get the user ID of the window */
    retval = XTSOLgetResUID(display, window, IsWindow, &uid);
```

Getting the X Window Server Workstation Owner ID

This example gets the ID of the user logged in to X Window Server. The process sensitivity label equals the window sensitivity label so no privileges are required.

```
/* Get the user ID of the window */
    retval = XTSOLgetWorkstationOwner(display, &uid);
```

Source Code

This is the source code for the simple Motif applications shown in Figure 14–1 and Figure 14–2. Launch it with any command line argument to see the text label string in italic font in the label widget.

Resource File

Here is the Resource file for the simple Motif application. One way to use it is to create the file and set the *XENVIRONMENT* variable with the pathname.

```
phoenix% setenv XENVIRONMENT /export/home/zelda/resfile
```

```
Example.*geometry: 400X100
Example.*orientation: XmHORIZONTAL
Example.*label.labelString: Launch an application
Example.*xclock.labelString: Run xclock
Example.*xterm.labelString: Run xterm
Example.*xmag.labelString: Run xmag
Example.*goodbye.labelString: Quit
Example.*XmPushButton*background: blue
Example.*XmLabel*foreground: white
Example.*XmLabel*foreground: white
```

Compile Command

```
phoenix% cc -I/usr/openwin/include -I/usr/dt/include ex.c -o Example \
-L/usr/openwin/lib -L/usr/dt/lib -lXm -lXt -lX11 -lXtsol -ltsol -lDtTsol
```

Code

```
#include <stdio.h>
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xm/Xm.h>
#include <Xm/Label.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>
#include <tsol/Xtsol.h>
#include <Dt/label_clipping.h>

XTsolResAttributes winattrs;
int retval, error;
uid_t uid;
Window window;
Display *display;
char *string = (char *)0, *string1 = (char *)0, *string2 = (char *)0,
    *string3 = (char *)0, *string4 = "SECRET";
XmFontList fontlist;
XmString xmstr;
XFontStruct *italic;
Arg args[9];
Dimension width = 144;
Widget stringLabel;
bslabel_t senslabel;
bclabel_t cmwlabel;

/* Callbacks */

void Xclock(Widget w, caddr_t client_data, caddr_t call_data)
{ system("xclock &"); }

void Xterm(Widget w, caddr_t client_data, caddr_t call_data)
{ system("xterm &"); }

void Quit(Widget w, caddr_t client_data, caddr_t call_data)
{
    fprintf(stderr, "exiting  . . .\n");
    exit(0);
}
main(int argc, char **argv)
{
    Widget rowcolumn, label, xclock, xterm, quit, form, topLevel;
    int i = 0;
    Arg args[9];

/* Create Widgets */
    topLevel = XtInitialize(argv[0], "XMCmds1", NULL, 0, &argc, argv);
    form = XtCreateManagedWidget("form", xmFormWidgetClass,
           topLevel, NULL, 0);

/* Launch application with any command argument to use the */
/* Text label string and font list for the label widget */
```

```
    if (argc == 2) {
/* Create the font list and translate the label using it */
        retval = getcmwplabel(&cmwlabel);
        getcsl(&senslabel, &cmwlabel);
        italic = XLoadQueryFont(XtDisplay(topLevel),
            "-adobe-times-medium-i-*-*-14-*-*-*-*-*-iso8859-1");
        fontlist = XmFontListCreate(italic, "italic");
        xmstr = (XmString)Xbsltos(XtDisplay(topLevel), &senslabel,
            width, fontlist, LONG_WORDS);

/* Create a label widget using the font list and label text*/
        i=0;
        XtSetArg(args[i], XmNfontList, fontlist); i++;
        XtSetArg(args[i], XmNlabelString, xmstr); i++;
        label = XtCreateManagedWidget("label", xmLabelWidgetClass,
            form, args, i);
    }

/* Launch application with no command arguments to use the text */
/* in the resource file for the label widget */

    else {
        label = XtCreateManagedWidget("label", xmLabelWidgetClass,
                form, NULL, 0); }

/* Continue widget creation */
    i=0;
    XtSetArg(args[i], XmNtopAttachment, XmATTACH_WIDGET); i++;
    XtSetArg(args[i], XmNtopWidget, label); i++;
    XtSetArg(args[i], XmNleftAttachment, XmATTACH_FORM); i++;
    XtSetArg(args[i], XmNrightAttachment, XmATTACH_POSITION); i++;
    XtSetArg(args[i], XmNrightPosition, 33); i++;
    XtSetArg(args[i], XmNbottomAttachment, XmATTACH_FORM); i++;
    xclock = XtCreateManagedWidget("xclock", xmPushButtonWidgetClass,
       form, args, i);

    i=0;
    XtSetArg(args[i], XmNtopAttachment, XmATTACH_WIDGET); i++;
    XtSetArg(args[i], XmNtopWidget, label); i++;
    XtSetArg(args[i], XmNleftAttachment, XmATTACH_POSITION); i++;
    XtSetArg(args[i], XmNleftPosition, 33); i++;
    XtSetArg(args[i], XmNrightAttachment, XmATTACH_POSITION);i++;Ha>
```

Changing Window Configuration

Window behavior in the Trusted Solaris operating environment can be modified by changing the settings in these files:

- `/usr/dt/bin/Xsession`

  This script starts the session and window managers. It sets the font path and other session-wide default values.
- `/usr/dt/bin/Xtsolusersession`

  This script establishes the context for starting applications in a workspace. For example, the script contains lines that source the $HOME/`.dtprofile` for a user with any login shell except the `pfsh`. (Therefore, by default, the `.dtprofile` is not sourced for an account whose login shell is the `pfsh`.) A site's security administrator can change this behavior by modifying the script.
- `/usr/openwin/server/tsol/config.privs`

  The file can be used to remove the security checks for specified privileges. Security checks are not enforced for those privileges contained in this file. For example, including `win_fontpath` in the file relaxes the restriction on loading fonts.
- `/usr/openwin/server/tsol/property.atoms`

  The property atoms specified in this file are not polyinstantiated.
- `/usr/openwin/server/tsol/public.atoms`

  The atoms specified in this file can be accessed by the XGetAtomName routine.
- `/usr/openwin/server/tsol/selection.atoms`

  The selection atoms in this file are polyinstantiated.

CHAPTER 15

Label Builder

The Trusted Solaris environment provides a set of Motif-based programming interfaces for creating an interactive user interface that builds valid sensitivity labels, CMW labels, or clearances from user input. This set of interfaces is called Label builder, and will be most often called from within administrative applications.

Label builder graphical user interfaces are used in the Trusted Solaris environment. The *Trusted Solaris User's Guide* describes these interfaces from the end user's point of view. This discussion describes the functionality provided by the Label builder library routines.

- "Header Files and Libraries" on page 237
- "Programming Interfaces" on page 238
- "Creating an Interactive User Interface" on page 238
- "Online Help" on page 247

Header Files and Libraries

To use the programming interfaces described in this section, you need the following header file.

```
#include <Dt/ModLabel.h>
```

The examples in this chapter compile with the following libraries:

```
-lDtTsol -ltsol
```

Programming Interfaces

The following programming interfaces are available for building label GUIs. The data types and parameter lists are covered in "Creating an Interactive User Interface" on page 238.

```
ModLabelData *tsol_lbuild_create(Widget widget,
    void (*event_handler)() ok_callback,
        lbuild_attributes extended_operation
        ...,
         NULL);

    void tsol_lbuild_destroy(ModLabelData *lbdata);

    void *tsol_lbuild_get(ModLabelData *lbdata,
        lbuild_attributes extended_operation);

    void tsol_lbuild_set(ModLabelData *lbdata,
        lbuild_attributes extended_operation,
        ...,
        NULL);
```

Creating an Interactive User Interface

The following figure shows the graphical user interface (GUI) created from the code after the figure. The `main` program creates a parent form (*form*) with one pushbutton (*display*). The pushbutton callback displays the Label builder dialog box created in the call to `tsol_lbuild_create`(3TSOL).

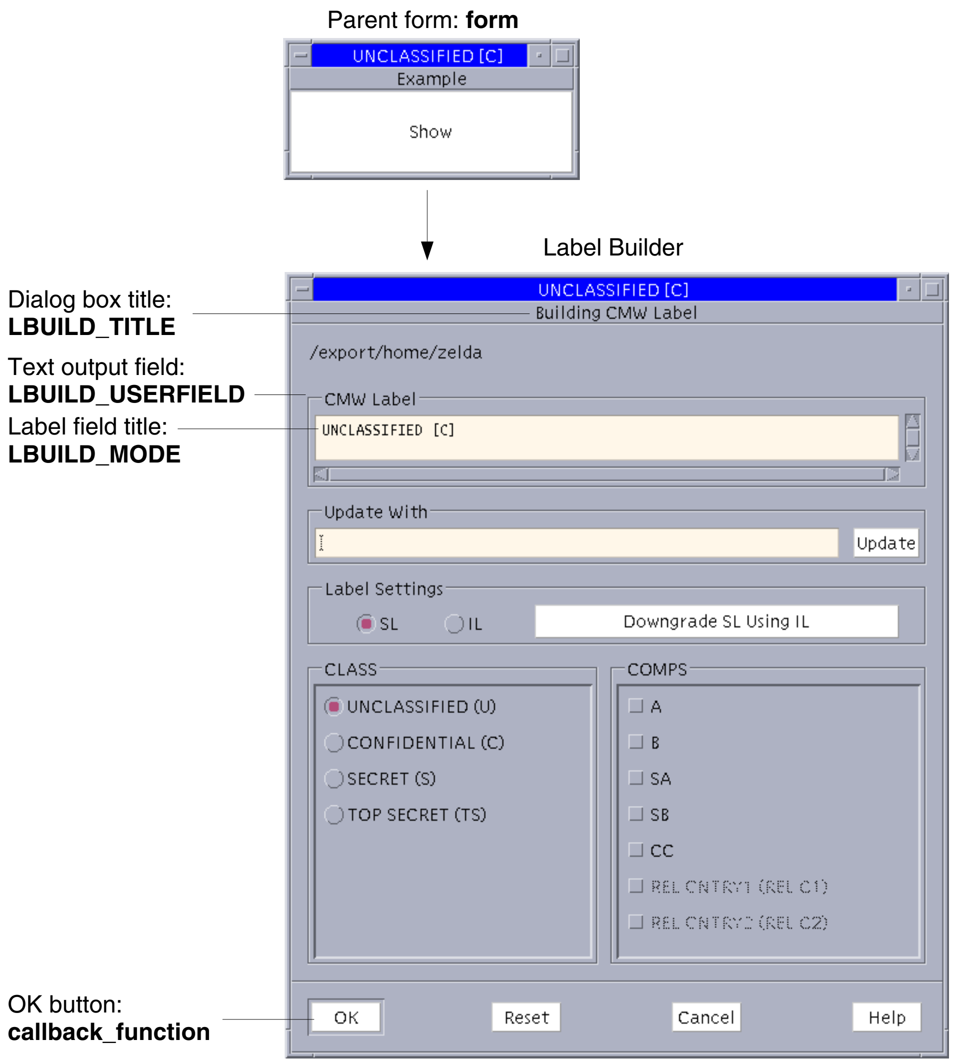

FIGURE 15–1 CMW Label Building Interface

The Label builder dialog box on the right appears when the Show pushbutton on the left is selected. The callouts point out where the parameters passed to `tsol_lbuild_create`(3TSOL) appear on the Label builder dialog box.

```
#include <X11/Intrinsic.h>
#include <X11/StringDefs.h>
#include <Xm/Xm.h>
#include <Xm/PushB.h>
#include <Xm/Form.h>
#include <Dt/ModLabel.h>
```

```
ModLabelData *data;

/* Callback passed to tsol_lbuild_create() */
void callback_function()
{
    char *title, *userval;
    char *string = (char *)0;
    char *string1 = (char *)0;
    int mode, view;
    Boolean show;
    bslabel_t sl_label, work_sl_label;
    Position x, y;

/* Your application-specific implementation goes here */
    printf("OK pushbutton called\n");

/* Query settings */
    mode = (int)tsol_lbuild_get(data, LBUILD_MODE);
    title = (String)tsol_lbuild_get(data, LBUILD_TITLE);
    sl_label = *(bslabel_t*) tsol_lbuild_get(data, LBUILD_VALUE_SL);
    work_sl_label = *(bslabel_t*) tsol_lbuild_get(data, LBUILD_WORK_SL);
    view = (int )tsol_lbuild_get(data, LBUILD_VIEW);
    x = (Position ) tsol_lbuild_get(data, LBUILD_X);
    y = (Position ) tsol_lbuild_get(data, LBUILD_Y);
    userval = (char *)tsol_lbuild_get(data, LBUILD_USERFIELD);
    show = (Boolean )tsol_lbuild_get(data, LBUILD_SHOW);

    bsltos(&sl_label, &string, 0, LONG_WORDS);
    bsltos(&work_sl_label, &string1, 0, LONG_WORDS);
    printf("Mode = %d, Title = %s, SL = %s, WorkSL = %s, View = %d, ",
        mode, title, string, string1, view);
    printf("X = %d, Y = %d, Userval = %s, Show = %d\n",
        x, y, userval, show);

}

/* Callback to display dialog box upon pushbutton press */
void Show(Widget display, caddr_t client_data, caddr_t call_data)
{
    tsol_lbuild_set(data, LBUILD_SHOW, TRUE, NULL);
}

main(int argc, char **argv)
{
    Widget     form, topLevel, display;
    Arg args[9];
    int i = 0, error, retval;
    char *sl_string = "[C]";
    bslabel_t sl_label;

    topLevel = XtInitialize(argv[0], "XMcmds1", NULL, 0, &argc, argv);
    form = XtCreateManagedWidget("form",
        xmFormWidgetClass, topLevel, NULL, 0);
    retval = stobsl(sl_string, &sl_label, NEW_LABEL, &error);
    printf("Retval = %d\n", retval);

    data = tsol_lbuild_create( form, callback_function,
        LBUILD_MODE, LBUILD_MODE_SL,
        LBUILD_TITLE, "Building Sensitivity Label",
        LBUILD_VALUE_SL, sl_label,
        LBUILD_VIEW, LBUILD_VIEW_EXTERNAL,
        LBUILD_X, 200,
        LBUILD_Y, 200,
        LBUILD_USERFIELD, "/export/home/zelda",
        LBUILD_SHOW, FALSE,
        NULL);

    i = 0;
    XtSetArg(args[i], XmNtopAttachment, XmATTACH_FORM); i++;
    XtSetArg(args[i], XmNleftAttachment, XmATTACH_FORM); i++;
    XtSetArg(args[i], XmNrightAttachment, XmATTACH_FORM); i++;
    XtSetArg(args[i], XmNbottomAttachment, XmATTACH_FORM); i++;
    display = XtCreateManagedWidget("Show",
        xmPushButtonWidgetClass, form, args, i);
    XtAddCallback(display, XmNactivateCallback, Show,0);
    XtRealizeWidget(topLevel);

    XtMainLoop();

    tsol_lbuild_destroy(data);

}
```

The `printf`(1) statements print the following:

```
OK pushbutton called
Mode = 12, Title = Building Sensitivity label,
Label = [C], WorkSL = [S],
View = 1, X = 200, Y = 200,
Userval = /export/home/zelda,
Show = 1
```

Label Builder Behavior

The Label builder dialog box prompts the end user for information and generates a valid CMW label from the input. The input can be entered from the keyboard or by choosing options. Either way, Label builder ensures that a valid label or clearance as defined in the `label_encodings`(4) file for the system is built.

Label builder provides default behavior for the OK, Reset, Cancel, and Update pushbuttons. The callback passed to `tsol_lbuild_create`(3TSOL) is mapped to the OK pushbutton to provide application-specific behavior.

Keyboard Entry

The Update pushbutton takes the text the end user types into the `Update With` field and checks that the string is a valid label or clearance as defined in the `label_encodings`(4) file.

- If the input is not valid, Label builder raises an error to the end user.
- If the input is valid, Label builder updates the text in the Label field above and stores the value in the appropriate working label field of the `ModLabelData` variable returned by `tsol_lbuild_create`(3TSOL). See "ModLabelData Structure" on page 246.

When the end user selects the OK pushbutton, the user-built value is handled according to the OK pushbutton callback implementation.

Selecting Options

The Label Settings radio button options let you build a sensitivity label or clearance from classifications and compartments, or an information label from classifications, compartments, and markings. Depending on the mode, one of these buttons might be grayed out. This approach is independent of the keyboard entry and Update pushbutton method described above.

The classifications, compartments, and markings information are from the `label_encodings`(4) file for the system. The combinations and constraints specified in the `label_encodings` file are enforced by graying out invalid combinations. The Label field updates the Label field above and stores the value in the appropriate working label field of the `ModLabelData` variable returned by `tsol_lbuild_create`(3TSOL) (see "ModLabelData Structure" on page 246) when the end user chooses options. The end user can build a sensitivity label, clearance, or sensitivity label portion of a CMW label from the classifications (`CLASS`) and compartments (`COMPS`) radio buttons listed.

When the end user selects the OK pushbutton, the user-built value is handled according to the OK pushbutton callback implementation.

Reset Pushbutton

The Rest pushbutton sets the text in the Label field to what its value was when the application started.

Cancel Pushbutton

The Cancel pushbutton exits the application.

Application-Specific Functionality

It is up to you to add application-specific callbacks, error handling, and other functionality to go with the valid label or clearance generated by the Label builder user interface.

Privileged Operations

Label builder shows the user only those classifications (and related compartments and markings) dominated by the workspace sensitivity label unless the executable has the `sys_trans_label` privilege in its effective set.

Your application-specific implementation for the OK pushbutton callback might require privileges.

If the end user does not have the authorization to upgrade or downgrade labels, or if the user-built label is out of the user's accreditation range, the OK and Reset buttons are grayed to prevent the end user from completing the task. There are no privileges to override these restrictions.

Create Routine

The `tsol_lbuild_create`(3TSOL) routine accepts any widget, Boolean value, callback, and a `NULL` terminated series of operation and value pairs. A variable of type *ModLabelData* is returned.

- Widget – Label builder can build the dialog box from any widget.
- Callback function – The callback function activates when the OK pushbutton is pressed. This callback provides application-specific behavior.
- Operation and value pairs – The operation (left) side of the pair specifies an extended operation (see "Extended Operations" on page 244) and the value (right) side specifies the value. In some cases, the value is an enumerated constant, and in other cases, you provide a value. The pairs can be specified in any order, but every operation you specify requires a valid value.
- The return value is a data structure that contains information on the dialog box just created. The information comes from the `tsol_lbuild_create`(3TSOL) input parameters and user activities during execution. Label builder provides default values for some fields where no values have been specified.

  Use the `tsol_lbuild_get`(3TSOL) and `tsol_lbuild_set`(3TSOL) routines to programmatically access and change the information in this variable. The data structure is described in "ModLabelData Structure" on page 246.

```
data= tsol_lbuild_create( form, callback_function,
    LBUILD_MODE, LBUILD_MODE_CMW,
    LBUILD_TITLE, "Building CMW Label",
    LBUILD_VALUE_CMW, cmwlabel,
    LBUILD_VIEW, LBUILD_VIEW_EXTERNAL
    LBUILD_X, 200,
    LBUILD_Y, 200,
    LBUILD_USERFIELD "/export/home/zelda"
    LBUILD_SHOW, FALSE,
    NULL);
```

Extended Operations

This section describes the extended operations and valid values you can pass to `tsol_lbuild_create`(3TSOL), `tsol_lbuild_get`(3TSOL), and `tsol_lbuild_set`(3TSOL). The values passed to `tsol_lbuild_create()` are stored in its return value of type `ModLabelData` where they can be accessed by calls to `tsol_lbuild_get()` and `tsol_lbuild_set()`. The `ModLabelData` structure is described in "ModLabelData Structure" on page 246.

All extended operations are valid to pass to `tsol_lbuild_get`(3TSOL). However the `*WORK*` operations are not valid to pass to `tsol_lbuild_set`(3TSOL) or `tsol_lbuild_create`(3TSOL) because these values are set by Label builder according to end user input. These exceptions are noted in the descriptions.

`LBUILD_MODE` – You can tell `tsol_lbuild_create()` to create a user interface to build information labels, sensitivity labels, CMW labels, or clearances. Value is `LBUILD_MODE_CMW` by default.

- `LBUILD_MODE_SL` – Build a sensitivity label.
- `LBUILD_MODE_CMW` – Build a CMW label.
- `LBUILD_MODE_CLR` – Build a clearance.

Note – Knowing how labels are configured for the system on which your application will run can help you know which mode to use. For example, you would not have a user build an information label on a system that does not use information labels. "Query System Security Configuration" on page 45 in Chapter 2explains how to check the system security configuration.

`LBUILD_VALUE_SL` – The starting sensitivity label displayed in the Label field above the Update With field when the mode is `LBUILD_MODE_SL`. This value is `ADMIN_LOW` by default.

`LBUILD_VALUE_CMW` – The starting CMW label displayed in the Label field above the Update With field when the mode is `LBUILD_MODE_CMW`. This value is `ADMIN_LOW[ADMIN_LOW]` by default.

LBUILD_VALUE_CLR – The starting clearance displayed in the Label field above the Update With field when the mode is LBUILD_MODE_CL. This value is ADMIN_LOW by default.

LBUILD_USERFIELD – A character string prompt that displays at the top of the Label builder dialog box. Value is NULL by default.

LBUILD_SHOW – Show or hide the Label builder dialog box. Value is FALSE by default.

- TRUE – Show the Label builder dialog box.
- FALSE – Hide the Label builder dialog box.

LBUILD_TITLE – A character string title that appears at the top of the Label builder dialog box. Value is NULL by default.

LBUILD_WORK_SL – The sensitivity label the end user is building. Value is updated to the end user's input when the end user selects the Update pushbutton or interactively chooses an option. Value is ADMIN_LOW by default. Not a valid extended operation for tsol_lbuild_set(3TSOL) or tsol_lbuild_create(3TSOL).

LBUILD_WORK_CMW – The CMW label the end user is building. Value is updated to the end user's input value when the end user selects the Update pushbutton or interactively chooses an option. Value is ADMIN_LOW[ADMIN_LOW] by default. Not a valid extended operation for tsol_lbuild_set(3TSOL)or tsol_lbuild_create(3TSOL)

LBUILD_WORK_CLR – The clearance the end user is building. Value is updated to the end user's input value when the end user selects the Update pushbutton or interactively chooses an option. Value is ADMIN_LOW by default. Not a valid extended operation for tsol_lbuild_set(3TSOL) or tsol_lbuild_create(3TSOL)

LBUILD_X – The X offset in pixels from the top-left corner of the Label builder dialog box in relation to the top-left corner of the screen. By default the Label builder dialog box is positioned in the middle of the screen.

LBUILD_Y – The Y offset in pixels from the top-left corner of the Label builder dialog box in relation to the top-left corner of the screen. By default the Label builder dialog box is positioned in the middle of the screen.

LBUILD_UPPER_BOUND – The highest classification (and related compartments and markings) available to the user as radio buttons for interactively building a label or clearance. A value you supply should be within the user's accreditation range. If no value is supplied, this value is the user's workspace sensitivity label, or if the executable has the sys_trans_label privilege, this value is the user's clearance.

LBUILD_LOWER_BOUND – The lowest classification (and related compartments and markings) available to the user as radio buttons for interactively building a label or clearance. This value is the user's minimum label.

LBUILD_CHECK_AR – Check that the user-built label is within the user's accreditation range. A value of 1 means check and a value of 0 means do not check. If the label is out of range, an error message is raised to the end user. The default is check.

LBUILD_VIEW – Use the internal or external label representation. Value is LBUILD_VIEW_EXTERNAL by default.

- LBUILD_VIEW_INTERNAL – Use the internal names for the highest and lowest labels in the system: ADMIN_HIGH and ADMIN_LOW.
- LBUILD_VIEW_EXTERNAL – Promote an ADMIN_LOW label to the next lowest label and demote an ADMIN_HIGH label to the next highest label.

ModLabelData Structure

The ModLabelData structure contains information on the state of the Label builder interface created in the call to tsol_lbuild_create(3TSOL). The following table describes the ModLabelData fields. All fields except the widgets and callback are accessible by specifying the listed extended operation and a valid value in a call to tsol_lbuild_set(3TSOL) and/or tsol_lbuild_get(3TSOL). See "Extended Operations" on page 244 for descriptions of the extended operations.

TABLE 15–1 ModLabelData Structure

Data Type	Field	Extended Operation/Description	Comments
int	mode	LBUILD_MODE	
int	check_ar	LBUILD_CHECK_AR	
int	view	LBUILD_VIEW	
Bool	show	LBUILD_SHOW	
char	*userfield	LBUILD_USERFIELD	
char	*lbuild_title	LBUILD_TITLE	
Position	x	LBUILD_X	
Position	y	LBUILD_Y	
bslabel_t	sl	LBUILD_VALUE_SL	
bclabel_t	cmw	LBUILD_VALUE_CMW	
bclear_t	clr	LBUILD_VALUE_CLR	

TABLE 15–1 ModLabelData Structure *(Continued)*

Data Type	Field	Extended Operation/Description	Comments
`bslabel_t`	`sl_work`	LBUILD_WORK_SL	Not valid for `tsol_lbuild_set()` or `tsol_lbuild_create()`.
`bclabel_t`	`cmw_work`	LBUILD_WORK_CMW	
`bclear_t`	`clr_work`	LBUILD_WORK_CLR	
`brange_t`	`range`	LBUILD_UPPER_BOUND, LBUILD_LOWER_BOUND	
`Widget`	`lbuild_dialog`	Label builder dialog box	
`Widget`	`ok`	OK pushbutton	
`Widget`	`cancel`	Cancel pushbutton	
`Widget`	`reset`	Reset pushbutton	
`Widget`	`help`	Help pushbutton	
`void`	`(*event_handler)()`	Callback passed to `tsol_lbuild_create()`	

Online Help

The Help pushbutton and other widgets used in the user interface can be accessed directly from your application code through the `lbl_shell` field in the ModLabelData structure. To add online help to your application, follow the procedures and guidelines in the document listed below.

Common Desktop Environment: Help System Author's and Programmer's Guide, Part No. 802-1578-10, from Sun Microsystems, Inc.

APPENDIX **A**

Programmer's Reference

This appendix provides reference materials for the following topics.

Man Pages

The Trusted Solaris release installs Solaris man pages and man pages specific to the Trusted Solaris environment. Where appropriate, the Solaris man pages are modified to contain security-related information relevant to the Trusted Solaris environment.

Reading Man Pages

The `intro` man pages provide global security policy information for Trusted Solaris man pages in a section. Security information specific to the interfaces on a particular man page is in the Description section, in the Errors section under `EPERM`, `EACCES`, or `ESRCH`, and in the Summary of Trusted Solaris Changes section at the end.

If a man page for a library routine has no security information on it and the routine has an underlying system call, check the man page for the underlying system call. The underlying system call enforces security policy for all library routines built on top of it.

- When the library routine and system call have similar names such as `fopen`(3UCB) and `open`(2), the policy information is on the system call man page only.
- Whether the names are similar or not, the library routine man pages for routines based on system calls have a cross-reference in their `See Also` section to the system call man page with the policy information.

If no Trusted Solaris man page exists for a Solaris interface, the interface has not been modified for the Trusted Solaris environment, or has been modified, but there is no security-related behavior to be aware of when using the interface in the Trusted Solaris environment.

Making Shared Libraries Trusted

To be used by any application that requires privilege(s), shared libraries must be trusted. In the Trusted Solaris environment, the dynamic linking of privileged applications to shared libraries is restricted—to ensure that privileged applications can never use untrusted libraries. A privileged application that tries to link to an untrusted library fails with an error such as: "error: cannot load **** libraries."

For information on how an administrator makes a library trusted, see "Testing New Software for Security" in *Trusted Solaris Administrator's Procedures*, and see also examples in the `crle`(1) man page.

Header File Locations

Most Trusted Solaris header files are located in `/usr/include/tsol` and include headers in `/usr/include/sys/tsol`. However, a few header files are modified from the Solaris operating environment, and are therefore located in other directories as follows:

Header File Name	Category of Interfaces
/usr/dt/include/label_clipping.h	X11 Window label translation
/usr/dt/include/Dt/ModLabel.h	Label Builder
/usr/openwin/include/tsol/Xtsol.h	X Window System
/usr/dt/include/Dt/label_clipping.h	Label clipping with font list

Header File Name	Category of Interfaces
/usr/include/rpc/rpc.h	Remote procedure calls (RPC)
/usr/include/sys/ipc/ipc/h	Interprocess communications (IPC)
/usr/include/sys/msg.h	System V message queues
/usr/include/sys/sem.h	System V semaphore sets
/usr/include/sys/shm.h	System V shared memory regions
/usr/include/sys/tsol/stream.h	Trusted streams
`/usr/include/bsm/auditwrite.h`	Auditing

Abbreviations in Names

Many of the Trusted Solaris interfaces and data structure names use the short abbreviations shown below in their names. Knowing the abbreviations will help you recognize the purpose of an interface or structure from its name.

Abbreviation	Name
attr	attribute
auth	authorization
`b`	binary
`c or cl`	CMW Label
`clear`	clearance
`cmw`	`CMW label`
`ent`	`entry`
`f`	`file`
fs	file system
`h`	hexadecimal
`i` or `il`	`information` Label
`l`	level, label, or symbolic link
lbuild	label builder

mld	multilevel directory
p	process
priv	privilege
prof	profile
prop	properties
r	reentrant
`res`	`resource`
`s`	string
`sec`	`security`
`sl`	sensitivity Label
sld	single-level directory
t6 or TSIX	Trusted Security Information Exchange
tp	Trusted Path
tsol	Trusted Solaris
xtsol	Trusted X11 Server

Developing, Testing, and Debugging

Development, testing, and debugging should take place on an isolated development system to prevent software bugs and incomplete code from compromising security policy on the main system.

- Remove extra debugging code especially code that provides undocumented features and back doors that bypass security checks.
- Make application data manipulation easy to follow so it can be inspected for security problems by the system administrator before installation.
- Test return codes for all programming interfaces. An unsuccessful call can have unpredictable results. When an unexpected error condition occurs, the application should always terminate.
- Test all functionality by running the application at all sensitivity labels and from all roles at which you expect it to run.
  - If the program is run by a normal user (not by a role), launch it from the command line as a normal user at the labels in the user accreditation range at

which it is intended to run.

   - If the program is run by a role, launch it from the command line from the administrative role at one the administrative label at which it is intended to run (`ADMIN_HIGH` or `ADMIN_LOW`), or from the user role at the labels in the user accreditation range at which it is intended to run.

- Test all functionality under privilege debugging mode so you know if the application has all the privileges it needs, or if it is attempting to perform privileged tasks that it should not be attempting.
- Know and follow privilege bracketing.
- Know the security implications of using privileges, and make sure the application does not compromise system security by its use of privilege.
- If you are applying the SUNWSpro debugger/dbx/dbxtool to test a privileged application, you must start the debugger first and then attach it to a running process. You cannot start the debugger with the command name as an argument.

Privilege Debugging

Privilege debugging mode is described in *Trusted Solaris Administrator's Procedures*. This is a summary of the steps for enabling privilege debugging and using `runpd`(1M) under privilege debugging mode to test an application.

1. Privilege debugging mode allows an application to succeed when it does not have the privileges it needs and tells you which privileges are missing.
2. In the `/etc/system` file, set the *tsol_privs_debug* variable to 1. This file is `ADMIN_LOW` and the owner is root.
3. In the `/etc/syslog.conf` file, uncomment the kern.debug; local0.debug line. This file is `ADMIN_LOW` and the owner is sys.
4. Touch the `/var/log/privdebug.log` file. This file is `ADMIN_HIGH` and the owner is root.
5. Reboot your system.
6. Assume an administrative role with `runpd`(1M) in the profile.
7. Use the `runpd()` command to invoke the executable and find out which privileges, if any, are missing. The following command line invokes the executable file in Zelda's confidential home directory. Information on missing privileges displays at the command line and is logged to the `/var/log/privdebug.log` file.

```
phoenix# runpd /export/home/.MLD.Zelda/.SLD.2/executable

runpd terminated with a status of 1

process runpd pid 822 lacking privilege file_mac_search to
perform special method upon resource VNODE (Jan 29 12:45)
```

```
process runpd pid 822 lacking privilege file_mac_read to
perform read method upon resource VNODE (Jan 29 12:45)
```

1. Interpret privilege numbers in the `/var/log/privdebug.log` file. The privilege number appears after the word privilege. Process 822 lacks privilege numbers 11 and 10 which correspond to `file_mac_search` and `file_mac_read`.

```
Jan 29 12:45:39 phoenix unix DEBUG: runpd pid 822 lacking
privilege 11 to 5 79

Jan 29 12:45:39 phoenix unix DEBUG: runpd pid 822 lacking
privilege 10 to 2 79
```

Assigning File Privileges using a Script

How to write privileged scripts to be deployed and used by others in your organization is described in *Trusted Solaris Administrator's Procedures*. This section briefly explains how to create a script that uses `setfpriv`(1) to assign forced and allowed privileges to an executable file for testing and debugging an application during application development.

First of all, the user or role you are working in needs a profile with the `setfpriv`(1) command and `file_setpriv` privilege assigned to it. The Object Privilege Management profile in the default system has these. To run the script from any shell and have the commands invoked by the script run under the profile shell and inherit your profile privileges, invoke `pfsh`(1M) at the top of the script as shown in the example below.

The example assigns forced and allowed privileges to `executable`. The `-s -f` options set forced privileges on `executable`, and the `-a` option sets allowed privileges on `executable`. This script will quit with the error: `executable: not owner` unless the `file_setpriv` privilege is inherited by the commands.

```
#/bin/pfsh
setfpriv -s -f
ipc_mac_write,proc_setsl,sys_trans_label -a
ipc_mac_write,proc_setsl,sys_trans_label executable
```

When you use a script to put forced and allowed privileges on an executable file, keep the following points in mind:

- If you remove a privilege from the allowed set specified in the script, you must also remove it from the forced set. If you remove it from the allowed set only, you will see the error: `executable: Invalid argument` when you run the script.
- If your program inherits privileges, launch it from the command line in the profile shell with the privileges to be inherited.
    - The allowed set of the executable file must have the privileges to be inherited.

- If your program is going to only inherit privileges, the forced set of the executable file should be empty.
- If your program takes a different action when a privilege is forced from when it is inherited, launch the program with the privilege in the forced and allowed set, and launch the program again with the privilege in the allowed set from a profile shell that has the privilege to be inherited.

Note – Always test the program at all labels at which it is intended to run.

Releasing an Application

You submit a fully tested and debugged application to the system administrator for application integration. The application can be submitted as a CDE action or software package. If the application uses privilege, the system administrator evaluates (or has someone else evaluate) the application source code and security information you supply with the CDE action or software package to verify the use of privilege does not compromise system security.

Notify the system administrator of new auditing events, audit classes, or X Window System properties your application uses because he or she will need to put them into the correct files. See Chapter 8 and Chapter 14 for more information.

Creating a CDE Action

A CDE action is launched from the work space by a user or role and inherits the privileges assigned to it in that user's or role's profile. A CDE action is a set of instructions that work like application macros or programming interfaces to automate desktop tasks such as running applications and opening data files. In the Trusted Solaris environment, applications are started from the work space as CDE actions. How to create a CDE action is fully described in the *Common Desktop Environment: Advanced User's and System Administrator's Guide*, Part Number: 802-1575-10. SunSoft, a Sun Microsystems, Inc. business, produces the guide.

Note – When you create a CDE action, always create an `f.action` rather than an `f.exec`. An `f.exec` executes the program as root with all privileges.

The system administrator puts the CDE action into the appropriate profiles and assigns inheritable privileges (if any) to the CDE action. The executable files associated with the CDE action need allowed privileges if the program inherits privileges and might or might not need forced privileges. You should list the inheritable, forced, and allowed, privileges the program uses (if any), indicate the labels at which the application is intended to run, and supply any effective user or group IDs required. The system administrator assigns forced and allowed privileges to the executable file, and assigns inheritable privileges, label ranges, and effective user and group IDs to the CDE action in the profile.

Creating a Software Package

The System V Release 4 application binary interface (ABI) specifies a software distribution model called software packaging that you use to package software for integration by the system administrator. All software distributed using the ABI model is guaranteed to install on all ABI-compliant systems.

When you create the software package, you supply security attribute information in the optional `tsolinfo`(4) file (described below), which is used in the package installation procedure. This file is optional because default security attributes are assigned during package installation in the event no security attribute information is provided with the package.

Package Files

A package consists of package objects (the files to be installed) and control files (files that control how, when, where, and if the package is installed). Information about packages already installed on the system is stored in the software installation database in `/var/sadm/install/contents.`

The Solaris operating environment provides the following commands for creating and installing ABI-compliant software packages.

`pkginfo`(1)	Display software package information.
`pkgparam`(1)	Display package parameter values.
`pkgask`(1M)	Create a request script.
`installf`(1M)	Add an entry to the software installation database.
`removef`(1M)	Remove an entry from the software installation database.

To create a package, you set up the following files:

- Required information files.
  - `pkginfo`(4)

  - `prototype`(4)
- Optional information files as needed
- Optional mandatory access control (MAC) security attributes file, `tsolinfo`(4).
- Optional packaging scripts as needed.

`pkgmk`(1) uses `pkginfo`(4) and `prototype`(4) to construct a software package. The optional scripts customize the installation and remove packages.

MAC Security Attributes

The `tsolinfo`(4) file contains entries associated with package objects that require special security attributes. If a package object does not have any `tsolinfo` entries associated with it, it is assigned a default set of security attributes derived from the file system where the package is finally installed. This file can contain one or more entries per package object in the following format, where all fields in the format are required for each entry.

```
attribute_name object_name attribute_value
```

Here is a list of possible attribute names, what they mean, and how to specify them.

Attribute Name	Description	Attribute Value
forced_privs	Package object forced privileges	Comma-separated list of privileges.
allowed_privs	Package object allowed privileges	Comma-separated list of privileges.
public	Package object is public.	No attribute value.
mld	Package object is a multilevel directory	No attribute value

The following example `tsolinfo`(4) file entries specify security attributes for the `sendmail`(1M) package objects.

Attribute Name	Package Object Name	Attribute value
`mld`	`var/spool.mail`	
`mld`	`var/mail`	
`mld`	`var/tmp`	
`allowed_privs`	`usr/lib/sendmail`	`all`
`forced_privs`	`usr/lib/sendmail`	`file_mac_write`

Attribute Name	Package Object Name	Attribute value
label	etc/security/tsol	[admin_high]

Description

- The var/spool/mail, var/mail, and var/tmp package objects are multilevel directories. The MLD attribute has no attribute values.
- The /usr/lib/sendmail object has All system privileges in its allowed privilege set.
- The /usr/lib/sendmail object has a comma-separated list of privileges in its forced set.
- The etc/security/tsol file has a CMW label in brackets, ADMIN_HIGH.

Edit Existing Package

To find and edit an existing package, search the software installation database with the grep(1) command. The information returned includes the package name.

```
% cat /var/sadm/install/contents | grep /usr/lib/object
```

Once you have the package name, you can find the package definition for that package and edit the tsolinfo(4) file. If no tsolinfo file exists, create one. If you create a tsolinfo file, add it to the prototype file so the pkgmk(1) command can find the tsolinfo file.

Add New Package

To add a new package, refer to the *Application Packaging Developer's Guide* for the Solaris operating environment for detailed information on creating packages. This section summarizes the concepts and steps.

The following Solaris commands are for creating new software packages.

pkginfo(1)	Display software package information.
pkgparam(1)	Display package parameter values.
pkgmk(1)	Create a software package.
pkgproto(1)	Generate a prototype file for input to pkgmk(1).
pkgtrans(1)	Transfer and/or translate a package.

The following figure shows the files that you are responsible for creating, the role of the pkgmk(1) command, and the resulting directory structure or package.

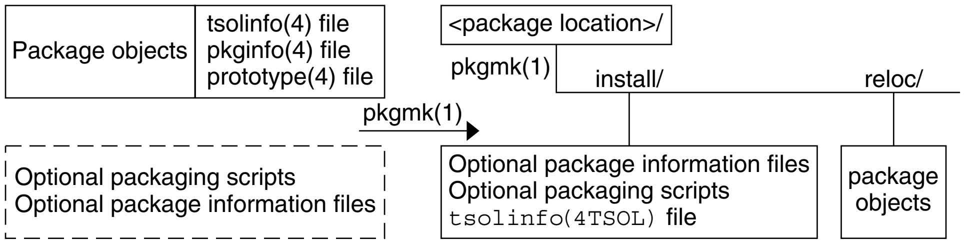

FIGURE A–1 Add New Package

▼ Create Required files

1. **Create a `pkginfo`(4) file using the man page.**
2. **Create a `prototype`(4) file using the man page.**

   Use the `pkgproto`(1) command to generate a `prototype`(4) file template.

▼ Create Optional Files and Scripts

1. **Create the `tsolinfo`(4) file using the man page.**

   Make sure `tsolinfo` is listed in the `prototype` file so that the `pkgmk`(1) command can find the `tsolinfo` file.
2. **Create optional package information files as needed.**
3. **Create optional packaging scripts as needed.**

▼ Create the Package

1. **Run the `pkgmk`(1) command.**
2. **Save the package to storage media**
3. **Give the storage media to the system administrator for installation.**

Prototype File

You can create a prototype file with any editor. There must be one entry for every package component. The following is a sample prototype file that contains an entry for the `tsolinfo` file. The `tsolinfo` file is preceded by the letter *i* to indicate an information file. The letter *f* indicates a standard executable or data file. Refer to the `prototype`(4) man page for more information.

```
# Package "prototype" file for the bbp device driver.
# Bidirectional Parallel Port Driver for SBus Printer Card.
#
i pkginfo
i request
i copyright
i postinstall
i tsolinfo
f none bbp.kmod 0444 root sys
f none bbp_make_node 0555 root sys
f none bbp_remove_node 0555 root sys
```

APPENDIX **B**

Trusted Solaris Interfaces Reference

This appendix has programming interface listings and chapter cross-references. Declaration listings are grouped by security topic. Name and section number listings are grouped by system calls, kernel functions, and library routines.

System Security Configuration

See Chapter 2.

```
long secconf(int name);
```

File System Security Attributes and Flags

See Chapter 2.

```
int fgetfsattr(int fd, u_long type, void *buf_P);
int fgetfattrflag(const char *path, secflgs_t *flags);
int fsetfattrflag(int fildes, secflgs_t *flags);

int getfattrflag(int fildes, secflgs_t *flags);
int getfsattr(char *path, u_long type, void *buf_P, int len);
int setfattrflag(const char *path, secflgs_t which, secflgs_t flags);

int mldgetfattrflag(const char *path, secflgs_t *flags)
int mldsetfattrflag(const char * path, secflgs_t which, secflgs_t flags))
```

Process Security Attribute Flags

See Chapter 2.

```
int getpattr(pattr_type_t type, pattr_flag_t *value);
int setpattr(pattr_type_t type, pattr_flag_t value);
```

Privileges

See Chapter 3.

```
int fgetfpriv(int fd, priv_ftype_t type, priv_set_t *priv_set);
int fsetfpriv(int fd, priv_op_t op, priv_ftype_t type,
              priv_set_t *priv_set);
int getfpriv(char *path, priv_ftype_t type, priv_set_t *priv_set);
int getppriv(priv_ptype_t type, priv_set_t *priv_set);

int setfpriv(char *path, priv_op_t op, priv_ftype_t type,
             priv_set_t *priv_set);
int setppriv(priv_op_t op, priv_ptype_t type, priv_set_t *priv_set);
int setppriv(priv_op_t op, priv_ptype_t type, priv_set_t *priv_set);

char *get_priv_text(const priv_t priv_id);
char *priv_to_str(const priv_t priv_id);
char *priv_set_to_str(priv_set_t *priv_set, const char sep,
                      char *buf, int *blen);
priv_t str_to_priv(const char *priv_name);
char *str_to_priv_set(const char *priv_names, priv_set_t *priv_set,
     const char *sep);
```

Privilege Macros

See Chapter 3.

```
PRIV_ASSERT(priv_set, priv_id)
PRIV_CLEAR(priv_set, priv_id)
PRIV_EMPTY(priv_set)
PRIV_EQUAL(priv_set_a, Priv_set_b)
PRIV_FILL(priv_set)
PRIV_INTERSECT(priv_set_a, priv_set_b)
```

```
PRIV_INVERSE(priv_set)
PRIV_ISASSERT(priv_set, priv_id)
PRIV_ISEMPTY(priv_set)
PRIV_ISFULL(priv_set)
PRIV_ISSUBSET(priv_set_a, priv_set_b)
PRIV_TEST(priv_id, errno)
PRIV_UNION(priv_set_a, priv_set_b)
PRIV_XOR(priv_set_a, priv_set_b,)
```

Labels

See Chapter 4.

File Systems

```
int getcmwfsrange(char *path, brange_t *range);
int fgetcmwfsrange(int fd, brange_t *range);
```

Label Encodings File

```
char bltocolor(const blevel_t *label);
char bltocolor_t(const blevel_t *label, const int size, char *color_name);
int labelinfo(struct label_info *info);
int labelvers(char **version, const int length);
```

Reentrant Routines

```
char halloc(const unsigned char id);
void hfree(char *hex);
char *bcltoh_r(const bclabel_t *label, char *hex);
char *bsltoh_r(const bslabel_t *label, char *hex);
```

Levels

```
int blequal(const blevel_t *level1, const blevel_t *level2);
int bldominates(const blevel_t *level1, const blevel_t *level2);
int blstrictdom(const blevel_t *level1, const blevel_t *level2);
int blinrange(const blevel_t *level, const brange_t *range);
void blmaximum(blevel_t *maximum_label, const blevel_t *bounding_label);
void blminimum(blevel_t *minimum_label, const blevel_t *bounding_label);
```

Label Types

```
int bltype(const void *label, const unsigned char type);
void setbltype(void *label, const unsigned char type);
```

Sensitivity Labels

```
void bslhigh(bslabel_t *label);
void bsllow(bslabel_t *label);
void bslundef(bslabel_t *label);
int bslvalid(const bslabel_t *senslabel);
int blinset(const blevel_t *senslabel, const set_id *id);
int bsltos(const bslabel_t *label, char **string, const int length,
    const int flags);
int stobsl(const char *string, bslabel_t *label, const int flags,
    int *error);
char *sbsltos(const bslabel_t *label, const int length);
char *bsltoh(const bslabel_t *label);
int htobcl(const char *hex, bclabel_t *label);
```

CMW Labels

```
int getcmwlabel(const char *path, const bclabel_t *label);
int setcmwlabel(const char *path, const bclabel_t *label,
    const setting_flag_t flag);
int fgetcmwlabel(const int fd, bclabel_t *label);
int fsetcmwlabel(const int fd, const bclabel_t *label,
    const setting_flag_t flag);
int lgetcmwlabel(const int fd, bclabel_t *label);
int lsetcmwlabel(const int fd, const bclabel_t *label,
    const setting_flag_t flag);
int getcmwplabel(const bclabel_t *label);
int setcmwplabel(const bclabel_t *label, const setting_flag_t flag);
void bclhigh(bclabel_t *label);
void bcllow(bclabel_t *label);
void bclundef(bclabel_t *label);
void getcsl(bslabel_t *destination_label, const bclabel_t *source_label);
void setcsl(bclabel_t *destination_label, const bslabel_t *source_label);
int bcltos(const bclabel_t *label, char **string, const int length,
    const int flags);
int stobcl(const char *string, bclabel_t *label, const int flags,
   int *error);
```

```
char *sbcltos(const bclabel_t *label, const int length);
char *bcltobanner(const bclabel_t *label, struct banner_fields *fields,
    const int flags);
bslabel_t *bcltosl(bclabel_t *label);
char *bcltoh(const bclabel_t *label);
int htobcl(const char *hex, bclabel_t *label);
```

Label Clipping Interfaces

See Chapter 14.

```
    XmString Xbcltos(Display *display,
        const bclabel_t *cmwlabel,
        const Dimension width,
        const XmFontList fontlist,
        const int flags);
    XmString Xbsltos(Display *display,
        const bslabel_t *senslabel,
        const Dimension width,
        const XmFontList fontlist,
        const int flags);
    XmString Xbcleartos(Display *display,
        const bclear_t *clearance,
        const Dimension width,
        const XmFontList fontlist,
        const int flags);
```

Clearances

See Chapter 6.

```
int getclearance(bclear_t *clearance);
int setclearance(bclear_t *clearance);
void bclearhigh(bclear_t *clearance);
void bclearlow(bclear_t *clearance);
void bclearundef(bclear_t *clearance);
int blequal(const blevel_t *level1, const blevel_t *level2);
int bldominates(const blevel_t *level1, const blevel_t *level2);
int blstrictdom(const blevel_t *level1, const blevel_t *level2);
int blinrange(const blevel_t *level, const brange_t *range);
void blmaximum(blevel_t *maximum_label, const blevel_t *bounding_label);
void blminimum(blevel_t *minimum_label, const blevel_t *bounding_label);
int bltype(const void *clearance, const unsigned char type);
```

```
void setbltype(void *clearance, const unsigned char type);
int bclearvalid(const bclear_t *clearance);
int bcleartos(const bclear_t *clearance, char **string, const int len,
    const int flags);
int stobclear(const char *string, bclear_t *clearance, const int flags,
    int *error);
char *sbcleartos(const bclear_t *clearance, const int len);
char *bcleartoh(const bclear_t *clearance);
int htobclear(const char *s, bclear_t *clearance);
char *h_alloc(const unsigned char id);
void h_free(char *hex);
char *bcleartoh_r(const bclear_t *clearance, char *hex);
```

Application Auditing

See Chapter 8.

```
int auditwrite(..., AW_END);
```

Multilevel Directories

See Chapter 7 and Chapter 2.

```
int getsldname(const char *path_name, const bslabel_t *slabel,
    char *name_buf, const int len);
int fgetsldname(const int fd, const bslabel_t *slabel_p,
    char *name_buf, const int len);
int getmldadorn(const char *path_name, char *adorn_buf);
int fgetmldadorn(const int fd, char adorn_buf);
int mldstat(const char *path_name,struct stat *stat_buf);
int mldlstat(const char *path_name, struct stat *stat_buf);
char *mldgetcwd(char *buf, size_t size);
int adornfc(const char *path_namechar *adorned_name);
char *mldrealpath(const char *path_name, char *resolved_path);
char *mldrealpathl(const char *path_name, char *resolved_path,
    const bslabel_t *senslabel);

/* These system calls are described in Chapter 2.
int mldgetfattrflag(const char *path, secflgs_t *flags)
int mldsetfattrflag(const char *path, secflgs_t which, secflgs_t flags))
```

Database Access

See Chapter 9.

```
userattr_t *getuserattr(void);
userattr_t *getusernam(const char *name);
userattr_t *getuseruid(uid_t uid);
void *free_userattr(userattr_t *userattr);
void setuserattr(void);
void enduserattr(void);
profattr_t *getprofattr(void);
profattr_t *getprofnam(const char *name);
void free_profattr(profattr_t *pd);
void setprofattr(void);
void endprofattr(void);
void getproflist(const char *profname, char **proflist, int*profcnt);
void free_proflist(char **proflist, int profcnt);
execattr_t *getexecattr(void);
void free_execattr(execattr_t *ep);
void setexecattr(void);
void endexecattr(void);
execattr_t *getexecuser(const char *username, const char *type,
 const char *id, int search_flag);
execattr_t *getexecprof(const char *profname, const char *type,
 const char *id, int search_flag);
execattr_t *match_execattr(execattr_t *ep, char *profname, char *type,
 char *id);
```

System V IPC

See Chapter 11.

Message Queues

```
int getmsgqcmwlabel(int msqid, bclabel_t *cmwlabel);
int msggetl(key_t key, int msgflg, bslabel_t *senslabel);
```

Semaphore Sets

```
int getsemcmwlabel(int semid, bclabel_t *cmwlabel);
int semgetl(key_t key, int nsems, int semflg, bslabel_t *senslabel);
```

Shared Memory Regions

```
int getshmcmwlabel(int shmid, bclabel_t *cmwlabel);
int shmgetl(key_t key, size_t size, int shmflg,bslabel_t *senslabel);
```

TSIX

See Chapter 12

```
t6mask_t t6supported_attrs(void);
t6mask_t t6allocated_attrs(t6attr_t t6ctl);
t6mask_t t6present_attrs(t6attr_t t6ctl);
t6attr_t t6alloc_blk(t6mask_t mask);
void t6free_blk(t6attr_t t6ctl);
int t6sendto(int sock, const char *msg, size_t len, int flags,
     const struct sockaddr *name, socklen_t namelen,
 const t6attr_t handle);
int t6recvfrom(int sock, void *buffer, size_t len, int flags,
  struct sockaddr *name, Psocklen_t namelenp, t6attr_t handle,
  t6mask_t *new_mask);
int t6new_attr(int fd, t6cmd_t cmd);
void *t6get_attr(t6attr_id_t attr_type, const t6attr_t t6ctl);
int t6set_attr(t6attr_id_t attr_type, const void *attr,
  t6attr_t t6ctl);
int t6peek_attr(int fd, t6attr_t attr_ptr, t6mask_t *new_attrs);
int t6last_attr(int fd, t6attr_t attr_ptr, t6mask_t *new_attrs);
size_t t6size_attr(t6attr_id_t attr_type, const t6attr_t t6ctl);
void t6copy_blk(const t6attr_t attr_src, t6attr_t attr_dest);
t6attr_t t6dup_blk(const t6attr_t attr_src);
int t6cmp_blk(t6attr_t t6ctl1, t6attr_t t6ctl2);
void t6clear_blk(t6mask_t mask, t6attr_t t6ctl);
int t6get_endpt_default(int fd, t6mask_t *mask, t6attr_t attr);
int t6set_endpt_mask(int fd, t6mask_t mask);
int t6set_endpt_default(int fd, t6mask_t mask,const t6attr_t attr_ptr);
int t6get_endpt_mask(int fd, t6mask_t *mask);
int t6ext_attr(int fd, t6cmd_t cmd);
```

RPC

There are no Trusted Solaris interfaces for remote procedure calls (RPC). RPC interfaces are modified to work in the Trusted Solaris operating environment. See Chapter 13 for conceptual information and a simple example application.

Label Builder

See Chapter 15.

```
ModLabelData *tsol_lbuild_create(Widget widget,
    void (*event_handler)() OK_callback,
    ..., NULL);

void tsol_lbuild_destroy(ModLabelData *lbdata);

XtPointer tsol_lbuild_get(ModLabelData *lbdata,
..., NULL);

void tsol_lbuild_set(ModLabelData *lbdata, extended_operation, NULL);
```

X Window System

See Chapter 14.

```
Status XTSOLgetResAttributes(Display *display, XID object,
      ResourceType resourceFlag, XTsolResAttributes *resattrp);
Status XTSOLgetPropAttributes(Display *display, Window win, Atom property,
      XTsolPropAttributes *propattrp);
Status XTSOLgetClientAttributes(Display *display, XID win,
      XTsolClientAttributes *clientattrp);
Status XTSOLgetResLabel(Display *display, XID object,
      ResourceType resourceFlag, bclabel_t *cmwlabel);
void XTSOLsetResLabel(Display *display, XID object,
   ResourceType resourceFlag, bclabel_t *cmwLabel,
   enum setting_flag labelFlag);
Status XTSOLgetResUID(Display *display, XID object,
   ResourceType resourceFlag, uid_t *uidp);
void XTSOLsetResUID(Display *display, XID object,
      ResourceType resourceFlag, uid_t *uidp);
Status XTSOLgetPropLabel(Display *display, Window win,
      Atom property, bclabel_t *cmwlabel);
void XTSOLsetPropLabel(Display *display, Window win, Atom property,
      bclabel_t *cmwLabel, enum setting_flag labelFlag);
Status XTSOLgetPropUID(Display *display, Window winID, Atom property,
  uid_t *uidp);
void XTSOLsetPropUID(Display *display, Window win,
      Atom property, uid_t *uidp);
Status XTSOLgetWorkstationOwner(Display *display, uid_t *uidp);
void XTSOLsetWorkstationOwner(Display *display, uid_t *uidp);
void XTSOLsetSessionHI(Display *display, bclear_t *clearance);
void XTSOLsetSessionLO(Display *display, bslabel_t *sl)
void XTSOLMakeTPWindow(Display *dpy, Window win);
Bool XTSOLIsWindowTrusted(Display *display, Window win);
Status XTSOLgetSSHeight(Display *display, int screen_num, int *newHeight);
void XTSOLsetSSHeight(Display *display, int screen_num, int newHeight);
void XTSOLsetPolyInstInfo(Display *dpy, bslabel_t *senslabel, uid_t *userID,
      int enabled);
```

Trusted Streams

These interfaces are kernel interfaces for creating trusted streams. See the man pages for information on them. They may be documented in this guide at a later date.

```
tsol_strattr_t *tsol_get_strattr(mblk_t *mp);
void tsol_set_strattr(mblk_t *mp, tsol_strattr_t *strattr);
```

System Calls

The system calls listing is organized alphabetically. It provides the chapter number where the interface is covered in this guide. You can also use the information to find the interface declaration in one of the previous topical lists.

TABLE B–1 System Calls

Programming Interface	Where Covered
`fgetcmwfsrange`(2)	Chapter 4
`fgetcmwlabel`(2)	Chapter 4

TABLE B–1 System Calls *(Continued)*

Programming Interface	Where Covered
`fgetfattrflag`(2)	Chapter 2
`fgetfpriv`(2)	Chapter 3
`fgetfsattr`(2)	Chapter 2
`fgetmldadorn`(2)	Chapter 7
`fgetsldname`(2)	Chapter 7
`fsetcmwlabel`(2)	Chapter 4
`fsetfattrflag`(2)	Chapter 2
`fsetfpriv`(2)	Chapter 3
`getclearance`(2)	Chapter 6
`getcmwfsrange`(2)	Chapter 4
`getcmwlabel`(2)	Chapter 4
`getcmwplabel`(2)	Chapter 4
`getfsattr`(2)	Chapter 2
`getmldadorn`(2)	Chapter 7
`getmsgqcmwlabel`(2)	Chapter 11
`getpattr`(2)	Chapter 2
`getppriv`(2)	Chapter 3
`getsemcmwlabel`(2)	Chapter 11
`getshmcmwlabel`(2)	Chapter 11
`getsldname`(2)	Chapter 7
`getcmwlabel`(2)	Chapter 4
`lsetcmwlabel`(2)	Chapter 4
`mldgetfattrflag`(2)	Chapter 2
`mldsetfattrflag`(2)	Chapter 2
`mldstat`(3TSOL)	Chapter 7
`mldlstat`(3TSOL)	Chapter 7
`msggetl`(2)	Chapter 11
`msgrcv`(2)	Chapter 11

TABLE B–1 System Calls *(Continued)*

Programming Interface	Where Covered
`msgsnd`(2)	Chapter 11
`secconf`(2)	Chapter 2
`semgetl`(2)	Chapter 11
`semop`(2)	Chapter 11
`setclearance`(2)	Chapter 6
`setcmwlabel`(2)	Chapter 4
`setcmwplabel`(2)	Chapter 4
`setfattrflag`(2)	Chapter 2
`setfpriv`(2)	Chapter 3
`setpattr`(2)	Chapter 2
`setppriv`(2)	Chapter 3
`shmgetl`(2)	Chapter 11

Trusted Kernel Functions for Drivers

The trusted kernel functions listing is organized alphabetically. See the man pages or "Trusted Streams" on page 270 for information on them. They may be documented in this guide at a later date.

- `tsol_get_strattr`(9F)
- `tsol_set_strattr`(9F)

Library Routines

The library routines listing is organized alphabetically. It provides the chapter number where the interface is covered in this guide. You can also use the information to find the interface declaration in one of the previous topical lists.

TABLE B–2 Library Routines

Library Routine	Where Covered
`adornfc`(3TSOL)	Chapter 7
`auditwrite`(3TSOL)	Chapter 8
`bclearhigh`(3TSOL)	Chapter 6
`bclearlow`(3TSOL)	Chapter 6
`bcleartoh`(3TSOL)	Chapter 6
`bcleartoh_r`(3TSOL)	Chapter 6
`bcleartos`(3TSOL)	Chapter 6
`bclearundef`(3TSOL)	Chapter 6
`bclearvalid`(3TSOL)	Chapter 6
`bclhigh`(3TSOL)	Chapter 4
`bcllow`(3TSOL)	Chapter 4
`bcltobanner`(3TSOL)	Chapter 4
`bcltoh`(3TSOL)	Chapter 4
`bcltoh_r`(3TSOL)	Chapter 4
`bcltos`(3TSOL)	Chapter 4
`bcltosl`(3TSOL)	Chapter 4
`bclundef`(3TSOL)	Chapter 4
`bldominates`(3TSOL)	Chapter 4, Chapter 6
`blequal`(3TSOL)	Chapter 4, Chapter 6
`blinrange`(3TSOL)	Chapter 4, Chapter 6
`blinset`(3TSOL)	Chapter 4
`blmaximum`(3TSOL)	Chapter 4, Chapter 6
`blminimum`(3TSOL)	Chapter 4, Chapter 6
`blmanifest`(3TSOL)	Chapter 4
`blportion`(3TSOL)	Chapter 4
`blstrictdom`(3TSOL)	Chapter 4
`bltocolor`(3TSOL)	Chapter 4
`bltype`(3TSOL)	Chapter 4, Chapter 6

TABLE B–2 Library Routines *(Continued)*

Library Routine	Where Covered
`bslhigh`(3TSOL)	Chapter 4
`bsllow`(3TSOL)	Chapter 4
`bsltoh`(3TSOL)	Chapter 4
`bsltoh_r`(3TSOL)	Chapter 4
`bsltos`(3TSOL)	Chapter 4
`bslundef`(3TSOL)	Chapter 4
`bslvalid`(3TSOL)	Chapter 4
`endprofent`(3TSOL)	Chapter 9
`endprofstr`(3TSOL)	Chapter 9
`enduserent`(3TSOL)	Chapter 9
`free_profent`(3TSOL)	Chapter 9
`free_profstr`(3TSOL)	Chapter 9
`free_userent`(3TSOL)	Chapter 9
`get_priv_text`(3TSOL)	Chapter 3
`getcsl`(3TSOL)	Chapter 4
`getexecattr`(3SECDB)	Chapter 9
`getprofent`(3TSOL)	Chapter 9
`getprofentbyname`(3TSOL)	Chapter 9
`getprofstr`(3TSOL)	Chapter 9
`getprofstrbyname`(3TSOL)	Chapter 9
`getuserent`(3TSOL)	Chapter 9
`getuserentbyname`(3TSOL)	Chapter 9
`getuserentbyuid`(3TSOL)	Chapter 9
`h_alloc`(3TSOL)	Chapter 4, Chapter 6
`h_free`(3TSOL)	Chapter 4, Chapter 6
`htobcl`(3TSOL)	Chapter 4
`htobclear`(3TSOL)	Chapter 6
`htobsl`(3TSOL)	Chapter 4

TABLE B–2 Library Routines *(Continued)*

Library Routine	Where Covered
`labelinfo`(3TSOL)	Chapter 4
`labelvers`(3TSOL)	Chapter 4
`mldgetcwd`(3TSOL)	Chapter 7
`mldrealpath`(3TSOL)	Chapter 7
`priv_set_to_str`(3TSOL)	Chapter 3
`priv_to_str`(3TSOL)	Chapter 3
`sbcleartos`(3TSOL)	Chapter 4
`sbcltos`(3TSOL)	Chapter 4
`sbsltos`(3TSOL)	Chapter 4
`set_effective_priv`(3TSOL)	Chapter 3
`set_inheritable_priv`(3TSOL)	Chapter 3
`set_permitted_priv`(3TSOL)	Chapter 3
`setbltype`(3TSOL)	Chapter 4, Chapter 6
`setcsl`(3TSOL)	Chapter 4
`setprofent`(3TSOL)	Chapter 9
`setprofstr`(3TSOL)	Chapter 9
`setuserent`(3TSOL)	Chapter 9
`stobcl`(3TSOL)	Chapter 4
`stobclear`(3TSOL)	Chapter 6
`stobsl`(3TSOL)	Chapter 4
`str_to_priv`(3TSOL)	Chapter 3
`str_to_priv_set`(3TSOL)	Chapter 3
`t6alloc_blk`(3NSL)	Chapter 12
`t6clear_blk`(3NSL)	Chapter 12
`t6cmp_blk`(3NSL)	Chapter 12
`t6copy_blk`(3NSL)	Chapter 12
`t6dup_blk`(3NSL)	Chapter 12
`t6ext_attr`(3NSL)	Chapter 12

TABLE B–2 Library Routines *(Continued)*

Library Routine	Where Covered
`t6free_blk`(3NSL)	Chapter 12
`t6get_attr`(3NSL)	Chapter 12
`t6get_endpt_default`(3NSL)	Chapter 12
`t6get_endpt_mask`(3NSL)	Chapter 12
`t6last_attr`(3NSL)	Chapter 12
`t6new_attr`(3NSL)	Chapter 12
`t6peek_attr`(3NSL)	Chapter 12
`t6recvfrom`(3NSL)	Chapter 12
`t6sendto`(3NSL)	Chapter 12
`t6set_endpt_default`(3NSL)	Chapter 12
`t6set_endpt_mask`(3NSL)	Chapter 12
`t6size_attr`(3NSL)	Chapter 12
`Xbcleartos`(3TSOL)	Chapter 14.
`Xbcltos`(3TSOL)	Chapter 14.
`Xbsltos`(3TSOL)	Chapter 14.
`XTSOLIsWindowTrusted`(3)	Chapter 14
`XTSOLMakeTPWindow`(3)	Chapter 14
`XTSOLgetClientAttributes`(3)	Chapter 14
`XTSOLgetPropAttributes`(3)	Chapter 14
`XTSOLgetPropLabel`(3)	Chapter 14
`XTSOLgetPropUID`(3)	Chapter 14
`XTSOLgetResAttributes`(3)	Chapter 14
`XTSOLgetResLabel`(3)	Chapter 14
`XTSOLgetResUID`(3)	Chapter 14
`XTSOLgetSSHeight`(3)	Chapter 14
`XTSOLgetWorkstationOwner`(3)	Chapter 14
`XTSOLsetPolyInstInfo`(3)	Chapter 14
`XTSOLsetPropLabel`(3)	Chapter 14

TABLE B–2 Library Routines *(Continued)*

Library Routine	Where Covered
`XTSOLsetPropUID`(3)	Chapter 14
`XTSOLsetResLabel`(3)	Chapter 14
`XTSOLsetResUID`(3)	Chapter 14
`XTSOLsetSSHeight`(3)	Chapter 14
`XTSOLsetSessionHI`(3)	Chapter 14
`XTSOLsetSessionLO`(3)	Chapter 14
`XTSOLsetWorkstationOwner`(3)	Chapter 14

Index