Trusted Extensions Developer's Guide

Part No: E54842
October 2017

Trusted Extensions Developer's Guide

Part No: E54842

Access to Oracle Support

Oracle customers that have purchased support have access to electronic support through My Oracle Support. For information, visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` or visit `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` if you are hearing impaired.

Référence: E54842

Accès aux services de support Oracle

Les clients Oracle qui ont souscrit un contrat de support ont accès au support électronique via My Oracle Support. Pour plus d'informations, visitez le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=info` ou le site `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=trs` si vous êtes malentendant.

Contents

Using This Documentation

- **Overview** – Describes how to use the programming interfaces to write new trusted applications for systems that run the Trusted Extensions feature of Oracle Solaris 11.3.
- **Audience** – Developers of secure, labeled programs. Note that the example programs in this book focus on the APIs being shown and do not perform error checking. Your applications should perform the appropriate error checking.
- **Required knowledge** – UNIX programming skills and understanding of site security requirements.

Product Documentation Library

Documentation and resources for this product and related products are available at `http://www.oracle.com/pls/topic/lookup?ctx=E53394-01`.

Feedback

Provide feedback about this documentation at `http://www.oracle.com/goto/docfeedback`.

♦ ♦ ♦ CHAPTER 1

Trusted Extensions APIs and Security Policy

The Trusted Extensions feature of the Oracle Solaris OS (Trusted Extensions) provides application programming interfaces (APIs) that enable you to write applications that access and handle labels. This chapter summarizes the API functionality and introduces you to the Trusted Extensions security policy.

For Trusted Extensions term definitions, see the glossary in the *Trusted Extensions User's Guide*.

This chapter covers the following topics:

- "Understanding Labels" on page 11
- "Trusted Extensions APIs" on page 15
- "Trusted Extensions Security Policy" on page 19

Understanding Labels

The Trusted Extensions software provides a set of policies and services to extend the security features of the Oracle Solaris operating system. These *extensions* provide access control that is based on label relationships.

Labels control access to data and maintain the classification of data. The labels are attributes that are interpreted by the system security policy. The *system security policy* is the set of rules that is enforced by system software to protect information that is being processed on the system. The term *security policy* can refer to the policy itself or to the implementation of the policy. For more information, see "Trusted Extensions Security Policy" on page 19.

This section includes overview information about label types, ranges, components, and relationships.

Label Types

The Trusted Extensions software defines two types of labels: sensitivity labels and clearance labels. A *sensitivity label* indicates the security level of an entity and is usually referred to as a *label*. A *clearance label* defines the upper boundary of a label range and is usually referred to as a *clearance*.

Sensitivity Labels

The Trusted Extensions software uses zones to contain classified information at various levels. Each level is associated with its own zone that has a sensitivity label. The sensitivity label specifies the sensitivity of the information in that zone and is applied to all of the subjects and objects in that zone. A label might be something like `CONFIDENTIAL`, `SECRET`, or `TOP SECRET`. A *subject* is an active entity, such as a process, that causes information to flow among objects or changes a system's state. An *object* is a passive entity that contains or receives data, such as a file or device. All processes that run in a zone, all files that are contained in a zone, and so on, have the same sensitivity label as their zone. All processes and objects have a sensitivity label that is used in mandatory access control (MAC) decisions. By default, sensitivity labels are visible in the windowing system.

Clearance Labels

The security administrator assigns a clearance to each user. A clearance is a label that defines the upper boundary of a label range. For example, if you have a clearance of `SECRET`, you can access information that is classified at this level or lower, but not information that is classified at a higher level. A *user clearance* is assigned by the security administrator. It is the highest label at which a user can access files and initiate processes during a session. In other words, a user clearance is the upper boundary of a user's account label range. At login, a user selects his session clearance. The *session clearance* determines which labels a user can access. The session clearance sets the *least upper bound* at which the user can access files and initiate processes during that login session. The session clearance is dominated by the user clearance.

Label Ranges

The security administrator defines label ranges and label sets to enforce *mandatory access control* (MAC) policy. A *label range* is a set of labels that is bounded at the upper end by a clearance or a limit and at the lower end by a minimum label. A *label limit* is the upper bound of a label range. A *label set* contains one or more discrete labels that might be disjoint from one another. Labels in a label set do not dominate one another.

Label Components

A label contains a hierarchical classification and a set of zero or more nonhierarchical compartments. A classification is also referred to as a *level* or a security level. A *classification* represents a single level within a hierarchy of labels, for example, TOP SECRET or UNCLASSIFIED. A *compartment* is associated with a classification and represents a distinct, nonhierarchical area of information in a system, such as private information for a human resources (HR) group or a sales group. A compartment limits access only to users who need to know the information in a particular area. For example, a user with a SECRET classification only has access to the secret information that is specified by the associated list of compartments, not to any other secret information. The classification and compartments together represent the label of the zone and the resources within that zone.

The textual format of a classification is specified in the label_encodings file and appears similar to this:

```
CLASSIFICATIONS:
name= CONFIDENTIAL; sname= C; value= 4; initial compartments= 4-5 190-239;
name= REGISTERED; sname= REG; value= 6; initial compartments= 4-5 190-239;
```

The textual format of a compartment is specified in the label_encodings file and appears similar to this:

```
WORDS:
name= HR; minclass= C; compartments= 0;
```

For more information about label definitions and label formats, see *Trusted Extensions Label Administration* and *Compartmented Mode Workstation Labeling: Encodings Format*. For information about the label APIs, see Chapter 2, “Labels and Clearances”.

Label Relationships

Comparing labels means that the label of a process is compared to the label of a target, which might be a sensitivity label or a clearance label. Based on the result of the comparison, the process is either granted access or denied access to the object. Access is granted only when the label of the process dominates the label of the target. Label relationships and dominance are described later in this section. For examples, see “Determining the Relationship Between Two Labels” on page 42.

A *security level* is a numerical classification. A label indicates the security level of an entity and might include zero or more compartments. An entity is something that can be labeled, such as a process, zone, file, or device.

Labels are of the following types and relate to each other in these ways:

- **Equal –** When one label is equal to another label, both of these statements are true:
  - The label's classification is numerically *equal to* the other label's classification.
  - The label has exactly the same compartments as the other label.
- **Dominant –** When one label dominates another label, both of these statements are true:
  - The label's classification is numerically *greater than or equal to* the other label's classification.
  - The label has exactly the same compartments as the other label.
- **Strictly dominant –** When one label strictly dominates another label, both of these statements are true:
  - The label's classification is numerically *greater than or equal to* the other label's classification.
  - The label has all the compartments that the other label has and at least one other compartment.
- **Disjoint –** When one label is disjoint with another label, both of these statements are true:
  - The labels are not equal.
  - Neither label dominates the other label.

The `label_encodings` file is used to specify the classifications and compartments for labels. See the `label_encodings(4)` man page.

When any type of label has a security level that is equal to or greater than the security level of a second label, the first label is said to *dominate* the second label. This comparison of security levels is based on classifications and compartments in the labels. The classification of the dominant label must be equal to or greater than the classification of the second label. Additionally, the dominant label must include all the compartments in the second label. Two equal labels are said to dominate each other.

In the following sample excerpt of the `label_encodings` file, the `REGISTERED` (`REG`) label dominates the `CONFIDENTIAL` (`C`) label. The comparison is based on the value of each label's `value` keyword. The value of the `REG` label's `value` keyword is numerically greater than or equal to the value of the `C` label's `value` keyword. Both labels dominate the `PUBLIC` (`P`) label.

The value of the `initial compartments` keyword shows the list of compartments that are initially associated with the classification. Each number in the `initial compartments` keyword is a *compartment bit*, each of which represents a particular compartment.

```
CLASSIFICATIONS:
```

```
name= PUBLIC; sname= P; value= 1;
name= CONFIDENTIAL; sname= C; value= 4; initial compartments= 4-5 190-239;
name= REGISTERED; sname= REG; value= 6; initial compartments= 4-5 190-239;
```

The following `label_encodings` excerpt shows that the `REG HR` label (Human Resources) dominates the `REG` label. The `REG HR` label has the `REGISTERED` classification and the `HR` compartment. The `compartments` keyword for the `HR` compartment sets the `0` compartment bit, so the `REG HR` classification has compartments `0`, `4`–`5`, and `190`–`239` set, which is more than the compartments set by the `REG` classification.

```
CLASSIFICATIONS:
name= REGISTERED; sname= REG; value= 6; initial compartments= 4-5 190-239;
...
WORDS:
name= HR; minclass= C; compartments= 0;
```

Sometimes, strict dominance is required to access an object. In the previous examples, the `REG` label strictly dominates the `P` label, and the `REG HR` label strictly dominates the `REG` label. When comparing labels, a `REG` label dominates another `REG` label.

Labels that do not dominate each other are said to be disjoint. A *disjoint* label might be used to separate departments in a company. In the following example, the `REG HR` label (Human Resources) is defined as being disjoint from the `REG Sales` label. These labels are disjoint because each compartment sets a different compartment bit.

```
CLASSIFICATIONS:
name= REGISTERED; sname= REG; value= 6; initial compartments= 4-5 190-239;
...
WORDS:
name= HR; minclass= C; compartments= 0;
name= Sales; minclass= C; compartments= 1;
```

For information about label APIs, see “Sensitivity Label APIs” on page 17.

Trusted Extensions APIs

This section introduces the following Trusted Extensions APIs that are described in this book:

- Label APIs
- Trusted X Window System APIs

In addition to these Trusted Extensions APIs, you can use the security APIs that are available with the Oracle Solaris OS. An application that runs on Trusted Extensions might require the manipulation of other security attributes. For example, the user and profile databases contain information about users, roles, authorizations, and profiles. These databases can restrict who can run a program. Privileges are coded into various Oracle Solaris programs and can also be coded into third-party applications.

For more information about these Oracle Solaris OS security APIs, see “Developing Privileged Applications,” in *Solaris Security for Developers Guide*.

The Oracle Solaris OS provides *discretionary access control* (DAC), in which the owner of the data determines who is permitted access to the data. The Trusted Extensions software provides additional access control, which is called mandatory access control (MAC). In MAC, ordinary users cannot specify or override the *security policy*. The security administrator sets the security policy.

Applications use Trusted Extensions APIs to obtain labels for hosts, zones, users, and roles. Where the security policy permits, the APIs enable you to set labels on user processes or on role processes. Setting a label on a zone or on a host is an administrative procedure, not a programmatic procedure.

The label APIs operate on opaque labels. In an *opaque label*, the internal structure of the label is not exposed. Using an opaque label enables existing programs that are created with the APIs to function even if the internal structure of the label changes. For example, you cannot use the label APIs to locate particular bits in a label. The label APIs enable you to obtain labels and to set labels. You can only set labels if you are permitted to do so by the security policy.

Label APIs

Labels, label ranges, and a label limit determine who can access information on a system that is configured with Trusted Extensions.

The label APIs are used to access, convert, and perform comparisons for labels, label ranges and limits, and the relationship between labels. A label can dominate another label, or a label can be disjoint from another label.

The `label_encodings` file defines the sensitivity labels, clearance labels, label ranges, and label relationships that pertain to your Trusted Extensions environment. This file also controls the appearance of labels. The security administrator is responsible for creating and maintaining the `label_encodings` file. See the `label_encodings(4)` man page.

The label of a process is determined by the zone in which the process executes.

All objects are associated with a label or sometimes with a label range. An object can be accessed at a particular label within the defined label range. The objects that are associated with a label range include the following:

- All users and all roles

- All hosts with which communications are permitted
- Zone interfaces and network interfaces
- Allocatable devices, such as tape drives, CD-ROM devices, and audio devices
- Other devices that are not allocatable, such as printers and workstations

  Workstation access is controlled by the label range that is set for the frame buffer or video display device. The security administrator sets this range by using the Device Manager GUI. By default, devices have a range from `ADMIN_LOW` to `ADMIN_HIGH`.

For more information about labels, see "Label Types" on page 12.

How Labels Are Used in Access Control Decisions

MAC compares the label of the process that is running an application with the label or the label range of any object that the process tries to access. MAC permits a process to read down to a lower label and permits a process to write to an equal label.

```
Label[Process] >= Label[Object]
```

A process bound to a multilevel port (MLP) can listen for requests at multiple labels and send replies to the originator of the request. In Trusted Extensions, such replies are write-equal.

```
Label[Process] = Label[Object]
```

Types of Label APIs

Sensitivity Label APIs

Sensitivity label APIs can be used to do the following:

- Obtain a process label
- Initialize labels
- Find the greatest lower bound or the least upper bound between two labels
- Compare labels for dominance and equality
- Check and set label types
- Convert labels to a readable format
- Obtain information from the `label_encodings` file
- Check that a sensitivity label is valid and within the system range

For a description of these APIs, see Chapter 2, "Labels and Clearances".

Clearance Label APIs

Users, devices, and network interfaces have label ranges. The upper bound of the range is effectively the clearance. If the upper bound of the range and the lower bound of the range are equal, the range is a single label.

Clearance label APIs can be used to do the following:

- Find the greatest lower bound or the least upper bound between two labels
- Compare labels for dominance and equality
- Convert clearances between the internal format and the hexadecimal format

For a description of these APIs, see Chapter 2, "Labels and Clearances".

Label Range APIs

A label range is used to set limits on the following:

- The labels at which hosts can send and receive information
- The labels at which processes acting on behalf of users and roles can work on the system
- The labels at which users can allocate devices

  This use of a label range restricts the labels at which files can be written to storage media on these devices.

Label ranges are assigned administratively. Label ranges can apply to users, roles, hosts, zones, network interfaces, printers, and other objects.

You can use the following methods to obtain information about label ranges:

- `getuserrange()` obtains the user's label range.
- `getdevicerange()` obtains the label range of a device.
- `tncfg -t` *template-name* `info` shows the label range of a template that is associated with a network interface.

For a description of these APIs, see Chapter 2, "Labels and Clearances".

Trusted X Window System APIs

The Trusted X Window System, Version 11, server starts at login. The server handles the workstation windowing system by using a trusted interprocess communication (IPC) path.

Windows, properties, selections, and ToolTalk™ sessions are created at multiple sensitivity labels as separate and distinct objects. The creation of distinct objects at multiple sensitivity labels is called *polyinstantiation*. Applications that are created with Motif widgets, Xt Intrinsics, Xlib, and desktop interfaces run within the constraints of the security policy. These constraints are enforced by extensions to the X11 protocols.

Chapter 5, “Trusted X Window System” describes the programming interfaces that can access the security attribute information described in “Trusted Extensions Security Policy” on page 19. These programming interfaces can also be used to translate the labels and clearances to text. The text can be constrained by a specified width and font list for display in the Trusted X Window System.

The Trusted X Window System stores the following security attributes:

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

The Trusted Path flag identifies a window as a Trusted Path window. The Trusted Path window protects the system from being accessed by untrusted programs. This window is always the topmost window, such as the screen stripe or login window.

Appendix B, “Trusted Extensions API Reference” lists the extensions that you can use to create an X11 trusted IPC path.

Trusted Extensions Security Policy

Sensitivity labels control access to data and maintain the classification of data. All processes and objects have a sensitivity label that is used in MAC decisions. The labels are attributes that are interpreted by the system security policy. The *system security policy* is the set of rules that is enforced by system software to protect information being processed on the system.

The following sections describe how the Trusted Extensions security policy affects multilevel operations, zones, and labels.

Multilevel Operations

When you create an operation that runs at multiple security levels, you must consider the following issues:

- Write-down policy in the global zone
- Default security attributes
- Default network policy
- Multilevel ports
- MAC-exempt sockets

Operations that run at multiple security levels are controlled by the global zone because only processes in the global zone can initiate processes at specified labels.

Write-Down Policy in the Global Zone

The ability of a subject, such as a process, to write an object whose label it dominates is referred to as *writing down*. The write-down policy in the global zone is specified administratively. Because global zone processes run at the `ADMIN_HIGH` label, certain file systems that are associated with other labels can be mounted read-write in the global zone. However, these special file system mounts must be administratively specified in automount maps, and they must be mounted by the global zone automounter. These mounts must have mount points within the zone path of the zone that has the same label as the exported file system. However, these mount points must not be visible from within the labeled zone.

For example, if the `PUBLIC` zone has a zone path of `/zone/public`, a writable mount point of `/zone/public/home/mydir` is permitted. However, a writable mount point of `/zone/public/root/home/mydir` is not permitted because it can be accessed by the labeled zone and *not* by the global zone. No cross-zone NFS mounts are permitted, which means that the NFS-mounted files can only be accessed by processes that run in the zone that mounted the file system. Global zone processes can write down to such files, subject to the standard discretionary access control (DAC) policy.

Local file systems associated with zones are protected from access by global zone processes by DAC, which uses file *permissions* and access control lists (ACLs). The parent directory of each zone's root (`/`) directory is only accessible by `root` processes or by processes that assert the `file_dac_search` privilege.

In general, the ability to write down from the global zone is restricted. Typically, writing down is used only when a file is reclassified by using the `setflabel()` interface or when privileged users drag and drop files between File Browser applications in different zones.

Default Security Attributes

Default security attributes are assigned to messages that arrive on Trusted Extensions hosts from other *host types*. The attributes are assigned according to settings in the network database files. For information about host types, their supported security attributes, and network database file defaults, see *Trusted Extensions Configuration and Administration*.

Default Network Policy

For network operations that send or receive data, the default policy is that the local process and the remote peer must have the same label. This policy applies to all zones, including the global zone, whose network label is `ADMIN_LOW`. However, the default network policy is more flexible than the policy for mounting file systems. Trusted Extensions provides administrative interfaces and programmatic interfaces for overriding the default network policy. For example, a system administrator can create an MLP in the global zone or in a labeled zone to enable listening at different labels.

Multilevel Ports

Caution - Use extreme caution when using a multilevel port to violate MAC policy. When you must use this mechanism, ensure that your server application enforces MAC policy.

Multilevel ports (MLPs) are listed in the `tnzonecfg` administrative database. Processes within the zone can bind to MLPs if these processes assert the `net_bindmlp` privilege. If the port number is less than 1024, the `net_privaddr` privilege must also be asserted. Such bindings allow a process to accept connections at all labels that are associated with the IP addresses to which the process is bound. The labels that are associated with a network interface are specified in the `tnrhdb` database and the `tnrhtp` database. The labels can be specified by a range, by a set of explicit enumerated labels, or by a combination of both.

When a privileged process that is bound to an MLP receives a TCP request, the reply is automatically sent with the label of the requester. For UDP datagrams, the reply is sent with the label that is specified by the `SO_RECVUCRED` option.

The privileged process can implement a more restrictive MAC policy by comparing the label of the request to other parameters. For example, a web server could compare the label of the requesting process with the label of the file specified in the URL. The remote label can be determined by using the `getpeerucred()` function, which returns the credentials of the remote peer. If the peer is running in a zone on the same host, the `ucred_get()` library routine returns a full set of credentials. Regardless of whether the peer is local or remote, the label of the peer is accessible from the `ucred` data structure by using the `ucred_getlabel()` function. This label can be compared with other labels by using functions such as `bldominates()`.

A zone can have single-level ports and multilevel ports. See "Multilevel Port Information" on page 45.

MAC-Exempt Sockets

The Trusted Extensions software provides an explicit socket option, `SO_MAC_EXEMPT`, to specify that the socket can be used to communicate with an endpoint at a lower label.

Caution - The `SO_MAC_EXEMPT` socket option must *never* be used unintentionally. Use extreme caution when using this socket option to disable MAC policy. When you must use this mechanism, ensure that your client application enforces MAC policy.

The Trusted Extensions software restricts the use of the `SO_MAC_EXEMPT` option in these ways:

- To explicitly set the socket option, a process must assert the `net_mac_aware` privilege.
- To further restrict the use of this socket option, the `net_mac_aware` privilege can be removed from the limit set for ordinary users.

See the `user_attr`(4) man page for details.

Sometimes, explicitly setting the socket option is not practical, such as when the socket is managed by a library. In such circumstances, the socket option can be set implicitly. The `setpflags()` system call enables you to set the `NET_MAC_AWARE` process flag. Setting this process flag also requires the `net_mac_aware` privilege. All sockets that are opened while the process flag is enabled automatically have the `SO_MAC_EXEMPT` socket option set. See the `setpflags`(2) and `getpflags`(2) man pages.

For applications that cannot be modified or recompiled, use the `ppriv -M` command to pass the `net_mac_aware` process flag to the application. In this case, all sockets that are opened by the application have the `SO_MAC_EXEMPT` option set. However, child processes of the application do not have this process flag or the related privilege.

Whenever you can, scrutinize and modify the source code of an application when you need to use the SO_MAC_EXEMPT socket option. If you *cannot* make such modifications to the code or if a safer method is not available to you, you may use the ppriv -M command.

The SO_MAC_EXEMPT socket option has been used sparingly by the Oracle Solaris OS. This option has been used by the NFS client. An NFS client might need to communicate with an NFS server that runs at a different label on an untrusted operating system. The NFS client enforces MAC policy to ensure that inappropriate requests are not granted.

In the Oracle Solaris OS, both the NFS server and client code include and enforce MAC policy so that communications between the Oracle Solaris client or server and an untrusted client or server has MAC policy enabled. To enable an untrusted host to communicate with a system that runs Trusted Extensions, the untrusted host must have an entry in the tnrhdb database. For more information, see "Labeling Hosts and Networks" in *Trusted Extensions Configuration and Administration*.

Zones and Labels

All objects on a system configured with Trusted Extensions are associated with a zone. Such zones are called *labeled zones*. A labeled zone is a non-global zone and is accessible to ordinary users. A user who is cleared at more than one label is permitted access to a zone at each of those labels.

The *global zone* is a special zone that contains files and processes that control the security policy of the system. Files in the global zone can only be accessed by roles and by privileged processes.

Labels in the Global Zone

The global zone is assigned a range of labels. The range is from ADMIN_LOW to ADMIN_HIGH. ADMIN_HIGH and ADMIN_LOW are *administrative labels*.

Objects in the global zone that are shared with other zones are assigned the ADMIN_LOW label. For example, files in the /usr, /sbin, and /lib directories are assigned the ADMIN_LOW label. These directories and their contents are shared by all zones. These files and directories are typically installed from packages and are generally not modified, except during packaging or patching procedures. To modify ADMIN_LOW files, a process must typically be run by superuser or by someone who has all privileges.

Information that is private to the global zone is assigned the label `ADMIN_HIGH`. For example, all processes in the global zone and all administrative files in the `/etc` directory are assigned the `ADMIN_HIGH` label. Home directories that are associated with roles are assigned the `ADMIN_HIGH` label. Multilevel information that is associated with users is also assigned the `ADMIN_HIGH` label. See "Multilevel Operations" on page 20. Access to the global zone is restricted. Only system services and administrative roles can execute processes in the global zone.

Labeled Zones

Non-global zones are called *labeled zones*. Each labeled zone has a unique label. All objects within a labeled zone have the same label. For example, all processes that run in a labeled zone have the same label. All files that are writable in a labeled zone have the same label. A user who is cleared for more than one label has access to a labeled zone at each label.

Trusted Extensions defines a set of label APIs for zones. These APIs obtain the labels that are associated with labeled zones and the path names within those zones:

- `getpathbylabel()`
- `getzoneidbylabel()`
- `getzonelabelbyid()`
- `getzonelabelbyname()`
- `getzonerootbyid()`
- `getzonerootbylabel()`
- `getzonerootbyname()`

For more information about these APIs, see "Accessing Labels in Zones" on page 32.

The label of a file is based on the label of the zone or of the host that owns the file. Therefore, when you relabel a file, the file must be moved to the appropriate labeled zone or to the appropriate labeled host. This process of relabeling a file is also referred to as *reclassifying* a file. The `setflabel()` library routine can relabel a file by moving the file. To relabel a file, a process must assert the `file_upgrade_sl` privilege or the `file_downgrade_sl` privilege. See the `getlabel`(2) and `setflabel`(3TSOL) man pages.

For more information about setting privileges, see "Developing Privileged Applications," in *Solaris Security for Developers Guide*.

♦ ♦ ♦ CHAPTER 2

Labels and Clearances

This chapter describes the Trusted Extensions APIs for performing basic label operations such as initializing labels, and comparing labels and clearances. This chapter also describes the APIs for accessing the label of a process.

This chapter covers the following topics:

Chapter 3, "Label Code Examples" provides code examples for the programming interfaces that are described in this chapter.

Privileged Operations and Labels

When an operation can bypass or override the security policy, the operation requires special privileges in its effective set.

Privileges are added to the effective set programmatically or administratively in these ways:

- If the executable file is owned by `root` and has the set user ID permission bit set, it starts with all privileges in its effective set. For example, the File Browser starts with all privileges in its effective set. Then, File Browser programmatically relinquishes most of its privileges to retain only the ones it needs to perform drag-and-drop operations across labels.
- The administrator can specify privileges in manifest files for SMF services or in the RBAC database `exec_attr` file for general commands. For more information about this file, see the `exec_attr`(4) man page.

The operation needs special privileges when translating binary labels and when upgrading or downgrading sensitivity labels.

Users and roles can run operations with special privileges. These privileges can be specified by using *rights profiles*. Applications can be written to run certain functions with certain privileges, as well. When you write an application that must assume special privileges, make sure that you enable the privilege only while running the function that needs it and that you remove the privilege when the function completes. This practice is referred to as *privilege bracketing*. For more information, see *Developer’s Guide to Oracle Solaris 11.3 Security*.

- **Translating binary labels –** You can translate a label between its internal representation and a string. If the label being translated is not dominated by the label of the process, the calling process requires the `sys_trans_label` privilege to perform the translation.
- **Upgrading or downgrading sensitivity labels –** You can *downgrade* or *upgrade* the sensitivity label on a file. If the file is not owned by the calling process, the calling process requires the `file_owner` privilege in its effective set. For more information, see the `setflabel`(3TSOL) man page.

  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. The `file_downgrade_sl` privilege also allows a file to be relabeled to a disjoint label.

  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.

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

- The application is launched at one sensitivity label and accesses data in objects at that same sensitivity label.
- The application is launched at one sensitivity label and accesses data in objects at other sensitivity labels, but the mandatory access operations are permitted by the system security policy. For example, read-down is allowed by MAC.

If an application tries to access data at sensitivity labels other than the sensitivity label of its process and access is denied, the process needs privileges to gain access. *Privileges* enable an application to bypass MAC or DAC. For example, the `file_dac_read`, `file_dac_write`, and `file_dac_search` privileges bypass DAC. The `file_upgrade_sl` and `file_downgrade_sl` privileges bypass MAC. No matter how access is obtained, the application design must not compromise the classification of the data that is accessed.

When your application changes its own sensitivity label or the sensitivity label of another object, be sure to close all file descriptors. An open file descriptor might leak sensitive data to other processes.

Label APIs

This section describes the APIs that are available for basic label operations. To use these APIs, you must include the following header file:

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

The label APIs compile with the `-ltsol` library option.

The Trusted Extensions APIs include data types for the following:

- **Sensitivity label –** The `m_label_t` type definition represents a sensitivity label. The `m_label_t` structure is opaque.

  Interfaces accept a variable of type `m_label_t` as a parameter. Interfaces can return sensitivity labels in a variable of type `m_label_t`. The `m_label_t` type definition is compatible with the `blevel_t` structure.
- **Sensitivity label range –** 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`.

The APIs for the following operations are described in this section:

- Enabling and disabling a Trusted Extensions system
- Setting a specific label encodings file
- Detecting a Trusted Extensions system
- Accessing the process sensitivity label
- Allocating and freeing memory for labels
- Obtaining and setting the label of a file
- Obtaining label ranges
- Accessing labels in zones
- Obtaining the remote host type
- Translating between labels and strings
- Comparing labels

Enabling and Disabling a Trusted Extensions System

The `labeling_enable()` routine is used to enable a Trusted Extensions system. The `labeling_disable()` routine is used to disable a Trusted Extensions system. These functions are asynchronous; they return before the enabling or disabling of the system completes. The following routine description includes the prototype declaration for each routine:

```
int labeling_enable(uint_t flags);
```

The labeling_enable() routine returns 0 upon successful completion. Otherwise, it returns -1 and errno indicates the error.

```
int labeling_disable(uint_t flags);
```

The labeling_disable() routine returns 0 upon successful completion. Otherwise, it returns -1 and errno indicates the error.

The possible flags are LABELING_DELAY, LABELING_FORCE, and LABELING_SYSLOG. See the labeling_enable(3TSOL) man page.

Setting the Label Encodings File

The labeling_set_encodings() routine is used to activate a specific label encodings file on a Trusted Extensions system. The following routine description includes the prototype declaration for the routine:

```
int labeling_set_encodings(const char *path);
```

The labeling_set_encodings() routine returns 0 upon successful completion. Otherwise, it returns -1 and errno indicates the error.

See the labeling_set_encodings(3TSOL) man page.

You can use the is_system_labeled() routine to detect whether the system is labeled.

Detecting a Trusted Extensions System

The is_system_labeled() routine is used to determine whether you are running on a Trusted Extensions system. The following routine description includes the prototype declaration for each routine:

```
int is_system_labeled(void);
```

The is_system_labeled() routine returns TRUE (1) if the Trusted Extensions software is installed and active. Otherwise, it returns FALSE (0).

See the is_system_labeled(3C) man page.

You can also use these other interfaces to determine whether the system is labeled:

- **X client.** If you are writing an X client that depends on multilevel functionality, use the XQueryExtension() routine to query the X server for the SUN_TSOL extension.
- **Shell script.** If you are writing a shell script that will determine whether the system is labeled, use the plabel command. See the plabel(1) man page.

  The following example shows the smf_is_system_labeled() function used by the /lib/svc/share/smf_include.sh script:

```
#
#  Returns zero (success) if system is labeled (aka Trusted Extensions).
#  1 otherwise.
#
smf_is_system_labeled() {
        [ ! -x /bin/plabel ] && return 1
        /bin/plabel > /dev/null 2>&1
        return $?
}
```

Accessing the Process Sensitivity Label

The getplabel() and ucred_getlabel() routines are used to access the sensitivity label of a process. The following routine descriptions include the prototype declaration for each routine:

```
int getplabel(m_label_t *label_p);
```

The getplabel() routine obtains the process label of the calling process.

See the getplabel(3TSOL) man page.

```
m_label_t *ucred_getlabel(const ucred_t *uc);
```

The ucred_getlabel() routine obtains the label in the credential of the remote process.

See the ucred_getlabel(3C) man page.

Allocating and Freeing Memory for Labels

The m_label_alloc(), m_label_dup(), and m_label_free() routines are used to allocate and free memory for labels. The following routine descriptions include the prototype declaration for each routine:

```
m_label_t *m_label_alloc(const m_label_type_t label_type);
```

The `m_label_alloc()` routine allocates a label in an `m_label_t` data structure on the heap. Labels must be allocated before calling routines such as `getlabel()` and `fgetlabel()`. Some routines, such as `str_to_label()`, automatically allocate an `m_label_t` structure.

When you create a label by using the `m_label_alloc()` routine, you can set the label type to be a sensitivity label or a clearance label.

```
int m_label_dup(m_label_t **dst, const m_label_t *src);
```

The `m_label_dup()` routine duplicates a label.

```
void m_label_free(m_label_t *label);
```

The `m_label_free()` routine frees the memory that was allocated for a label.

When you allocate an `m_label_t` structure or when you call another routine that automatically allocates an `m_label_t` structure, you are responsible for freeing the allocated memory. The `m_label_free()` routine frees the allocated memory.

See the `m_label`(3TSOL) man page.

Obtaining and Setting the Label of a File

The `setflabel()` routine, the `getlabel()` system call, and the `fgetlabel()` system call are used to obtain and set the label of a file. The following descriptions include the prototype declarations for the routine and the system calls:

```
int setflabel(const char *path, const m_label_t *label_p);
```

The `setflabel()` routine changes the sensitivity label of a file. When the sensitivity label of a file changes, the file is moved to a zone that corresponds to the new label. The file is moved to a new path name that is relative to the root of the other zone.

See the `setflabel`(3TSOL) man page.

For example, if you use the `setflabel()` routine to change the label of the file `/zone/internal/documents/designdoc.odt` from `INTERNAL` to `RESTRICTED`, the new path of the file will be `/zone/restricted/documents/designdoc.odt`. Note that if the destination directory does not exist, the file is not moved.

When you change the sensitivity label of a file, the original file is deleted. The only exception occurs when the source and destination file systems are loopback-mounted from the same underlying file system. In this case, the file is renamed.

When a process creates an object, the object inherits the sensitivity label of its calling process. The setflabel() routine programmatically sets the sensitivity label of a file system object.

The File Browser application and the setlabel command permit an authorized user to move an existing file to a different sensitivity label. See the setlabel(1) man page.

```
int getlabel(const char *path, m_label_t *label_p);
```

The getlabel() system call obtains the label of a file that is specified by *path*. The label is stored in an m_label_t structure that you allocate.

See the getlabel(2) man page.

```
int fgetlabel(int fd, m_label_t *label_p);
```

The fgetlabel() system call obtains the label of an open file by specifying a file descriptor.

When you allocate an m_label_t structure, you are responsible for freeing the allocated memory by using the m_label_free() routine. See the m_label(3TSOL) man page.

Obtaining Label Ranges

The getuserrange() and getdevicerange() routines are used to obtain the label range of a user and a device, respectively. The following routine descriptions include the prototype declaration for each routine:

```
m_range_t *getuserrange(const char *username);
```

The getuserrange() routine obtains the label range of the specified user. The lower bound in the range is used as the initial workspace label when a user logs in to a multilevel desktop. The upper bound, or clearance, is used as an upper limit to the available labels that a user can assign to labeled workspaces.

The default value for a user's label range is specified in the label_encodings file. The value can be overridden by the user_attr file.

See the setflabel(3TSOL), label_encodings(4), and user_attr(4) man pages.

```
bl_range_t *getdevicerange(const char *device);
```

The getdevicerange() routine obtains the label range of a user-allocatable device. If no label range is specified for the device, the default range has an upper bound of ADMIN_HIGH and a lower bound of ADMIN_LOW.

You can use the `list_devices` command to show the label range for a device.

See the `list_devices`(1) man page.

Accessing Labels in Zones

These functions obtain label information from objects in zones. The following routine descriptions include the prototype declaration for each routine:

```
char *getpathbylabel(const char *path, char *resolved_path, size_t bufsize, const
m_label_t *sl);
```

The `getpathbylabel()` routine expands all symbolic links and resolves references to `/./`, `/../`, removes extra slash (`/`) characters, and stores the zone path name in the buffer named by *resolved_path*. The *bufsize* variable specifies the size in bytes of this buffer. The resulting path does not have any symbolic link components or any `/./`, `/../`. This function can only be called from the global zone.

The zone path name is relative to the sensitivity label, *sl*. To specify a sensitivity label for a zone name that does not exist, the process must assert either the `priv_file_upgrade_sl` or the `priv_file_downgrade_sl` privilege, depending on whether the specified sensitivity label dominates or does not dominate the process sensitivity label.

See the `getpathbylabel`(3TSOL) man page.

```
m_label_t *getzoneidbylabel(const m_label_t *label);
```

The `getzoneidbylabel()` routine returns the zone ID of the zone whose label is *label*. This routine requires that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone.

See the `getzoneidbylabel`(3TSOL) man page.

```
m_label_t *getzonelabelbyid(zoneid_t zoneid);
```

The `getzonelabelbyid()` routine returns the MAC label of *zoneid*. This routine requires that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone.

See the `getzonelabelbyid`(3TSOL) man page.

```
m_label_t *getzonelabelbyname(const char *zonename);
```

The `getzonelabelbyname()` routine returns the MAC label of the zone whose name is *zonename*. This routine requires that the specified zone's state is at least `ZONE_IS_READY`.

The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone.

See the `getzonelabelbyname`(3TSOL) man page.

`m_label_t *getzonerootbyid(zoneid_t zoneid);`

The `getzonerootbyid()` routine returns the root path name of *zoneid*. This routine requires that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone. The returned path name is relative to the root path of the caller's zone.

See the `getzonerootbyid`(3TSOL) man page.

`m_label_t *getzonerootbylabel(const m_label_t *label);`

The `getzonerootbylabel()` routine returns the root path name of the zone whose label is *label*. This routine requires that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone. The returned path name is relative to the root path of the caller's zone.

See the `getzonerootbylabel`(3TSOL) man page.

`m_label_t *getzonerootbyname(const char *zonename);`

The `getzonerootbyname()` routine returns the root path name of *zonename*. This routine requires that the specified zone's state is at least `ZONE_IS_READY`. The zone of the calling process must dominate the specified zone's label, or the calling process must be in the global zone. The returned path name is relative to the root path of the caller's zone.

See the `getzonerootbyname`(3TSOL) man page.

Obtaining the Remote Host Type

This routine determines the remote host type. The following routine description includes the prototype declaration:

`tsol_host_type_t tsol_getrhtype(char *hostname);`

The `tsol_getrhtype()` routine queries the kernel-level network information to determine the host type that is associated with the specified host name. *hostname* can be a regular host name, an IP address, or a network wildcard address. The returned value is one of the enumerated types that is defined in the `tsol_host_type_t` structure. Currently, these types are `UNLABELED` and `SUN_CIPSO`.

See the `tsol_getrhtype`(3TSOL) man page.

Translating Between Labels and Strings

The label_to_str() and str_to_label() routines are used to translate between labels and strings. The following routine descriptions include the prototype declaration for each routine:

```
int label_to_str(const m_label_t *label, char **string, const m_label_str_t
conversion_type, uint_t flags);
```

The label_to_str() routine translates a label, m_label_t, to a string. You can use this routine to translate a label into a string that hides the classification name. This format is suitable for storing in public objects. The calling process must dominate the label to be translated, or the process must have the sys_trans_label privilege.

See the label_to_str(3TSOL) man page.

The label_to_str() routine allocates memory for the translated string. The caller must free this memory by calling the free() routine.

See the free(3C) man page.

```
int str_to_label(const char *string, m_label_t **label, const m_label_type_t label_type,
uint_t flags, int *error);
```

The str_to_label() routine translates a label string to a label, m_label_t. When you allocate an m_label_t structure, you must free the allocated memory by using the m_label_free() routine.

When you create a label by using the str_to_label() routine, you can set the label type to be a sensitivity label or a clearance label.

See the str_to_label(3TSOL) and m_label(3TSOL) man pages.

Readable Versions of Labels

The label_to_str() routine provides readable versions of labels. The M_LABEL conversion type returns a string that is classified at that label. The M_INTERNAL conversion type returns a string that is unclassified. The classified string version is typically used for displays, as in windows. The classified string might not be suitable for storage. Several conversion types are offered for printing purposes. All printing types show a readable string that is classified at the label that the string shows.

The conversion_type parameter controls the type of label conversion. The following are valid values for conversion_type, although not all types of conversion are valid for both level types:

- M_LABEL is a string of the label that is based on the type of label: sensitivity or clearance. This label string is classified at the level of the label and is therefore not safe for storing in a public object. For example, an M_LABEL string such as CONFIDENTIAL is not safe for storing in a public directory because the words in the label are often classified.

- M_INTERNAL is a string of an unclassified representation of the label. This string is safe for storing in a public object. For example, an M_INTERNAL string such as 0x0002-04-48 is safe for storing in an LDAP database.
- M_COLOR is a string that represents the color that the security administrator has associated with the label. The association between the label and the color is stored in the LOCAL DEFINITIONS section of the label_encodings file.

Label Encodings File

The label_to_str() routine uses the label definitions in the label_encodings file. The encodings file is a text file that is maintained by the security administrator. The file contains site-specific label definitions and constraints. This file is kept in /etc/security/tsol/label_encodings. For information about the label_encodings file, see *Trusted Extensions Label Administration*, *Compartmented Mode Workstation Labeling: Encodings Format*, and the label_encodings(4) man page.

Comparing Labels

The blequal(), bldominates(), and blstrictdom() routines are used to compare labels. The blinrange() routine is used to determine whether a label is within a specified label range. In these routines, a *level* refers to a classification and a set of compartments in a sensitivity label or in a clearance label.

```
int blequal(const blevel_t *level1, const blevel_t *level2);
```

The blequal() routine compares two labels to determine whether *level1* equals *level2*.

```
int bldominates(const m_label_t *level1, const m_label_t *level2);
```

The bldominates() routine compares two labels to determine whether *level1* dominates *level2*.

```
int blstrictdom(const m_label_t *level1, const m_label_t *level2);
```

The blstrictdom() routine compares two labels to determine whether *level1* strictly dominates *level2*.

```
int blinrange(const m_label_t *level, const brange_t *range);
```

The blinrange() routine determines whether the label, *level*, is within the specified range, *range*.

These routines return a nonzero value when the comparison is true and a value of `0` when the comparison is false. For more information about these routines, see the `blcompare`(3TSOL) man page.

For more information about label relationships, see "Label Relationships" on page 13.

The `blmaximum()` and `blminimum()` routines are used to determine the upper and lower bounds of the specified label range.

```
void blmaximum(m_label_t *maximum_label, const m_label_t *bounding_label);
```

The `blmaximum()` routine compares two labels to find the least upper bound of the range. The *least upper bound* is the lower of two clearances, which is used to determine whether you have access to a system of a particular clearance.

For instance, use this routine to determine the label to use when creating a new labeled object that combines information from two other labeled objects. The label of the new object will dominate both of the original labeled objects.

See the `blminmax`(3TSOL) man page.

```
void blminimum(m_label_t *minimum_label, const m_label_t *bounding_label);
```

The `blminimum()` routine compares two labels to find the label that represents the greatest lower bound of the range that is bounded by the two levels. The *greatest lower bound* is the higher of two labels, which is also used to determine whether you have access to a system of a particular clearance.

See the `blminmax`(3TSOL) man page.

Acquiring a Sensitivity Label

Sensitivity labels are acquired from labeled zones and from other processes. A user can start a process only at the current sensitivity label of the current zone.

When a process creates an object, the object inherits the sensitivity label of its calling process. You can use the `setlabel` command or the `setflabel()` routine to set the sensitivity label of a file system object. See the `setlabel`(1) and `setflabel`(3TSOL) man pages.

The following script, `runwlabel`, runs a program that you specify in the labeled zone that you specify. You must run this script from the global zone.

EXAMPLE 1 runwlabel Script

The runwlabel script must first acquire the sensitivity label of the labeled zone in which you want to run the specified program. This script uses the getzonepath command to obtain the zone path from the label that you specify on the command line. See the getzonepath(1) man page.

Next, the runwlabel script uses the zoneadm command to find the zone name associated with the zone path, which was acquired by the getzonepath command. See the zoneadm(1M) man page.

Finally, the runwlabel script uses the zlogin command to run the program that you specify in the zone associated with the label you specified. See the zlogin(1) man page.

To run the zonename command in the zone associated with the Confidential: Internal Use Only label, run the runwlabel script from the global zone. For example:

```
system1% runwlabel "Confidential : Internal Use Only" zonename
```

The following shows the source of the runwlabel script:

```
#!/sbin/sh
#
# Usage:
# runwlabel "my-label" my-program
#
[ ! -x /usr/sbin/zoneadm ] && exit 0    # zones not installed

PATH=/usr/sbin:/usr/bin; export PATH

# Get the zone path associated with the "my-label" zone
# Remove the trailing "/root"
zonepath=`getzonepath "$1" | sed -e 's/\/root$//'`
progname="$2"

# Find the zone name that is associated with this zone path
for zone in `zoneadm list -pi | nawk -F: -v zonepath=${zonepath} '{
        if ($4 == zonepath) {
            print $2
        }
    }'`; do

        # Run the specified command in the matching zone
        zlogin ${zone} ${progname}
    done
exit
```

The following script, `runinzone`, runs a program in a zone that you specify even if the zone is not booted. You must run this script from the global zone.

EXAMPLE 2 `runinzone` Script

The script first boots the zone you specified, and then it uses the `zlogin` command to run the `waitforzone` script in the specified zone.

The `waitforzone` script waits for the local zone automounter to come up, and then it runs the program you specified as the user you specified.

To run the `/usr/bin/xclock` command in the `public` zone, run the following from the global zone:

```
system1% runinzone public terry /usr/bin/xclock
```

The following shows the source of the `runinzone` script:

```
#!/sbin/ksh
zonename=$1
user=$2
program=$3

# Boot the specified zone
zoneadm -z ${zonename} boot

# Run the command in the specified zone
zlogin ${zonename} /bin/demo/waitforzone ${user} ${program} ${DISPLAY}
```

The `runinzone` script calls the following script, `waitforzone`:

```
#!/bin/ksh
user=$1
program=$2
display=$3

# Wait for the local zone automounter to come up
# by checking for the auto_home trigger being loaded

while [ ! -d /home/${user} ]; do
sleep 1
done

# Now, run the command you specified as the specified user

su - ${user} -c "${program} -display ${display}"
```

♦ ♦ ♦ CHAPTER 3

Label Code Examples

This chapter contains several code examples that show how to use the label APIs that are described in Chapter 2, “Labels and Clearances”.

This chapter covers the following topics:

Obtaining a Process Label

This code example shows how to obtain and print the sensitivity label of the zone in which this program is run.

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

main()
{
    m_label_t* pl;
    char *plabel = NULL;
    int retval;

    /* allocate an m_label_t for the process sensitivity label */
    pl = m_label_alloc(MAC_LABEL);
    /* get the process sensitivity label */
    if ((retval = getplabel(pl)) != 0) {
        perror("getplabel(pl) failed");
        exit(1);
    }

    /* Translate the process sensitivity label to text and print */
    if ((retval = label_to_str(pl, &plabel, M_LABEL, LONG_NAMES)) != 0) {
        perror("label_to_str(M_LABEL, LONG_NAMES) failed");
        exit(1);
    }
    printf("Process label = %s\n", plabel);

    /* free allocated memory */
    m_label_free(pl);
    free(plabel);
}
```

The `printf()` statement prints the sensitivity label. The sensitivity label is inherited from the zone in which the program is run. The following shows the text output of this example program:

```
Process label = ADMIN_LOW
```

The text output depends on the specifications in the `label_encodings` file.

Obtaining a File Label

You can obtain a file's sensitivity label and perform operations on that label.

This code example uses the `getlabel()` routine to obtain the file's label. The `fgetlabel()` routine can be used in the same way, but it operates on a file descriptor.

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

main()
{
    m_label_t* docLabel;
    const char* path = "/zone/restricted/documents/designdoc.odt";
    int retval;
    char* label_string;

    /* allocate label and get the file label specified by path */
    docLabel = m_label_alloc(MAC_LABEL);
    retval = getlabel(path, docLabel);

    /* translate the file's label to a string and print the string */
    retval = label_to_str(docLabel, &label_string, M_LABEL, LONG_NAMES);
    printf("The file's label = %s\n", label_string);

    /* free allocated memory */
    m_label_free(docLabel);
    free(label_string);
}
```

When you run this program, the output might look similar to this:

```
The file's label = CONFIDENTIAL : INTERNAL USE ONLY
```

Setting a File Sensitivity Label

When you change the sensitivity label of a file, the file is moved to a new zone that matches the file's new label.

In this code example, the process is running at the `CONFIDENTIAL` label. The user who is running the process has a `TOP SECRET` clearance. The `TOP SECRET` label dominates the `CONFIDENTIAL` label. The process upgrades the sensitivity label to `TOP SECRET`. The user needs the Upgrade File Label RBAC authorization to successfully perform the upgrade.

The following program is called `upgrade-afile`.

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

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

    /* Create new sensitivity label value */
    if ((retval = str_to_label(string, &fsenslabel, MAC_LABEL, L_DEFAULT, &err)) != 0) {
        perror("str_to_label(MAC_LABEL, L_DEFAULT) failed");
        exit(1);
    }

    /* Set file label to new value */
    if ((retval = setflabel("/export/home/zelda/afile", &fsenslabel)) != 0) {
        perror("setflabel("/export/home/zelda/afile") failed");
        exit(1);
    }

    m_label_free(fsenslabel);
}
```

The result of running this program depends on the process's label, relative to the label of the file that was passed to the process.

Before and after you run this program, you use the getlabel command to verify the file's label. As the following shows, before the program runs, the label for afile is CONFIDENTIAL. After the program runs, the label for afile is TOP SECRET.

```
% pwd
/export/home/zelda
% getlabel afile
afile: CONFIDENTIAL
% update-afile
% getlabel afile
afile: TOP SECRET
```

If you run the getlabel command from a window labeled CONFIDENTIAL after you reclassified the file, it is no longer visible. If you run the getlabel command in a window labeled TOP SECRET, you can see the reclassified file.

Determining the Relationship Between Two Labels

If your application accesses data at different sensitivity labels, perform checks in your code to ensure that the process label has the correct relationship to the data label before you permit an access operation to occur. You check the sensitivity label of the object that is being accessed to determine whether access is permitted by the system.

The following code example shows how to test two sensitivity labels for equality, dominance, and strict dominance. The program checks whether a file's label is dominated by or is equal to the process's label.

```
#include <stdio.h>
#include <stdlib.h>

#include <tsol/label.h>

main(int argc, char *argv[])
{
    m_label_t *plabel;
    m_label_t *flabel;

    plabel = m_label_alloc(MAC_LABEL);
    flabel = m_label_alloc(MAC_LABEL);

    if (getplabel(plabel) == -1) {
        perror("getplabel");
        exit(1);
    }
    if (getlabel(argv[1], flabel) == -1) {
        perror("getlabel");
        exit(1);
    }

    if (blequal(plabel, flabel)) {
        printf("Labels are equal\n");
    }
    if (bldominates(plabel, flabel)) {
        printf("Process label dominates file label\n");
    }
    if (blstrictdom(plabel, flabel)) {
        printf("Process label strictly dominates file label\n");
    }

    m_label_free(plabel);
    m_label_free(flabel);

    return (0);
}
```

The text output of this program depends on the process's label, relative to the label of the file that was passed to the process, as follows:

- Because "dominates" includes "equal" when the labels are equal, the output is the following:

  ```
  Labels are equal
  Process label dominates file label
  ```
- If the process's label strictly dominates the file's label, the output is the following:

  ```
  Process label strictly dominates file label
  ```

Obtaining the Color Names of Labels

This code example uses the `label_to_str()` function to obtain the color name of a label. The mappings between color names and labels are defined in the `label_encodings` file.

```
#include <stdlib.h>
#include <stdio.h>

#include <tsol/label.h>

int
main()
{
```

```
    m_label_t *plabel;
    char *label = NULL;
    char *color = NULL;

    plabel = m_label_alloc(MAC_LABEL);

    if (getplabel(plabel) == -1) {
        perror("getplabel");
        exit(1);
    }

    if (label_to_str(plabel, &color, M_COLOR, 0) != 0) {
        perror("label_to_string(M_COLOR)");
        exit(1);
    }
    if (label_to_str(plabel, &label, M_LABEL, DEF_NAMES) != 0) {
        perror("label_to_str(M_LABEL)");
        exit(1);
    }

    printf("The color for the \"%s\" label is \"%s\".\n, label, color);

    m_label_free(plabel);

    return (0);
}
```

If the `label_encodings` file maps the color blue to the label `CONFIDENTIAL`, the program prints the following:

```
The color for the "CONFIDENTIAL" label is "BLUE".
```

♦ ♦ ♦ CHAPTER 4

Interprocess Communications

A system that is configured with Trusted Extensions enforces mandatory access control (MAC) and discretionary access control (DAC). Access control is enforced between communicating processes on the same host and across the network. This chapter summarizes the interprocess communication (IPC) mechanisms that are available in a system configured with Trusted Extensions. This chapter also discusses how access controls apply.

This chapter covers the following topics:

- "Multilevel Port Information" on page 45
- "Communication Endpoints" on page 46

Multilevel Port Information

A system that is configured with Trusted Extensions supports single-level and multilevel ports. These ports are used to create connections between applications. A multilevel port can receive data within the range of sensitivity labels that is defined for that port. 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 the `net_bindmlp` privilege in its effective set and any number of unprivileged applications that run at different sensitivity labels. The application with the `net_bindmlp` privilege in the effective set of its process can receive all data from the applications, regardless of the receiving application's sensitivity label.

  A multilevel port is a server-side mechanism to establish a connection between two Trusted Extensions applications that are running at different labels. If you want a Trusted Extensions client application to communicate with a service that runs on an untrusted operating system at a different label, you might be able to use the `SO_MAC_EXEMPT` socket option. For more information, see "MAC-Exempt Sockets" on page 22.

Caution - If a connection is multilevel, ensure that the application does not make a connection at one sensitivity label, and then send or receive data at another sensitivity label. Such a configuration would cause data to reach an unauthorized destination.

The Trusted Network library provides an interface to retrieve the label from a packet. The programmatic manipulation of network packets is not needed. Specifically, you cannot change the security attributes of a message before it is sent. Also, you cannot change the security attributes on the communication endpoint over which the message is sent. You can read the label of a packet, just as you read other security information of a packet. The `ucred_getlabel()` function is used to retrieve label information.

If your application requires the use of a multilevel port, that port cannot be created programmatically. Rather, you must tell the system administrator to create a multilevel port for the application.

For more information about multilevel ports, see the following:

- "Zones and Multilevel Ports" in *Trusted Extensions Configuration and Administration*
- "How to Create a Multilevel Port for a Zone" in *Trusted Extensions Configuration and Administration*

Communication Endpoints

The Trusted Extensions software supports IPC over communication endpoints by using the following socket-based mechanisms:

- Berkeley sockets
- Transport Layer Interface (TLI)
- Remote procedure calls (RPC)

This section summarizes the socket communication mechanisms and the related security policy. See the appropriate man page for specific information about the security policy and applicable privileges.

In addition to these mechanisms, Trusted Extensions also supports multilevel ports. See "Multilevel Port Information" on page 45.

Berkeley Sockets and TLI

The Trusted Extensions software supports network communication by using Berkeley sockets and the TLI over single-level ports and multilevel ports. The `AF_UNIX` family of system calls establishes interprocess connections in the same labeled zone by means of a special file that is specified by using a fully resolved path name. The AF_INET family of system calls establishes interprocess connections across the network by using IP addresses and port numbers.

AF_UNIX Family

In the AF_UNIX family of interfaces, only one server bind can be established to a single special file, which is a UNIX® domain socket. The AF_UNIX family does not support multilevel ports.

Like UNIX domain sockets, doors and named pipes use special files for rendezvous purposes.

The default policy for all Trusted Extensions IPC mechanisms is that they are all constrained to work within a single labeled zone. The following are exceptions to this policy:

- The global zone administrator can make a named pipe (FIFO) available to a zone whose label dominates the owning zone. The administrator does this by loopback-mounting the directory that contains the FIFO.

  A process that runs in the higher-level zone is permitted to open the FIFO in read-only mode. A process is not permitted to use the FIFO to write down.
- A labeled zone can access global zone door servers if the global zone rendezvous file is loopback-mounted into the labeled zone.

  The Trusted Extensions software depends on the door policy to support the labeld and nscd doors-based services. The default zonecfg template specifies that the /var/tsol/doors directory in the global zone is loopback-mounted into each labeled zone.

AF_INET Family

In the AF_INET family, the process can establish a single-label connection or a multilabel connection to privileged or unprivileged port numbers. To connect to privileged port numbers, the net_priv_addr privilege is required. If a multilevel port connection is sought, the net_bindmlp privilege is also required.

The server process needs the net_bindmlp privilege in its effective set for a multilevel port connection. 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, any process that is denied access needs the appropriate file privilege in its effective set to gain access.

The following code example shows how a multilevel server can obtain the labels of its connected clients. The standard C library function getpeerucred() obtains a connected socket or a STREAM peer's credentials. In the context of Trusted Extensions, when the listening socket of a multilevel port server accepts a connection request, the first argument is typically a client socket file descriptor. The Trusted Extensions application uses the `getpeerucred()` function in exactly the same way a normal application program does. The Trusted Extensions addition is `ucred_getlabel()`, which returns a label. For more information, see the `ucred_get`(3C) man page.

```
/*
 * This example shows how a multilevel server can
 * get the label of its connected clients.
 */
void
remote_client_label(int svr_fd)
{
    ucred_t *uc = NULL;
    m_label_t *sl;
    struct sockaddr_in6 remote_addr;

    bzero((void *)&remote_addr, sizeof (struct sockaddr_in6));

    while (1) {
        int clnt_fd;
        clnt_fd = accept(svr_fd, (struct sockaddr *)&remote_addr,
            &sizeof (struct sockaddr_in6));

        /*
         * Get client attributes from the socket
         */
        if (getpeerucred(clnt_fd, &uc) == -1) {
            return;
        }

        /*
         * Extract individual fields from the ucred structure
         */

        sl = ucred_getlabel(uc);

        /*
         * Security label usage here
         * .....
         */

        ucred_free(uc);
        close(clnt_fd);
    }
}
```

RPC Mechanism

The Trusted Extensions software provides multilevel port support for remote procedure calls (RPCs). A client application can send inquiries to a server's PORTMAPPER service (port 111) whether or not a particular service is available. If the requested service is registered with the PORTMAPPER on the server, the server will dynamically allocate an anonymous port and return this port to the client.

On a Trusted Extensions system, an administrator can configure the PORTMAPPER port as a multilevel port so that multiple single-level applications can use this service. If the PORTMAPPER port is made a multilevel port, all anonymous ports allocated by the PORTMAPPER service are also multilevel ports. There are no other programmable interfaces or administrative interfaces to control anonymous multilevel ports.

Using Multilevel Ports With UDP

The PORTMAPPER service described in the previous section is implemented by using UDP. Unlike TCP, UDP sockets are not connection oriented, so some ambiguity might arise about which credentials to use when replying to a client on a multilevel port. Therefore, the client's request socket must be explicitly associated with the server's reply packet. To make this association, use the SO_RECVUCRED socket option.

When SO_RECVUCRED is set on a UDP socket, the kernel UDP module can pass a label in a ucred structure as ancillary data to an application. The level and type values of the ucred are SOL_SOCKET and SCM_UCRED, respectively.

An application can handle this ucred structure in one of these ways:

- Copy this ucred structure from the receiving buffer to the send buffer
- Reuse the receiving buffer as the send buffer and leave the ucred structure in the receiving buffer

The following code excerpt shows the reuse case.

```
/*
 * Find the SCM_UCRED in src and place a pointer to that
 * option alone in dest. Note that these two 'netbuf'
 * structures might be the same one, so the code has to
 * be careful about referring to src after changing dest.
 */
static void
extract_cred(const struct netbuf *src, struct netbuf *dest)
```

```
{
    char *cp = src->buf;
    unsigned int len = src->len;
    const struct T_opthdr *opt;
    unsigned int olen;

    while (len >= sizeof (*opt)) {
        /* LINTED: pointer alignment */
        opt = (const struct T_opthdr *)cp;
        olen = opt->len;
        if (olen > len || olen < sizeof (*opt) ||
            !IS_P2ALIGNED(olen, sizeof (t_uscalar_t)))
            break;
        if (opt->level == SOL_SOCKET &&
            opt->name == SCM_UCRED) {
            dest->buf = cp;
            dest->len = olen;
            return;
        }
        cp += olen;
        len -= olen;
    }
    dest->len = 0;
}
```

The following code excerpt shows how to access the user credential from the receiving buffer:

```
void
examine_udp_label()
{
    struct msghdr   recv_msg;
    struct cmsghdr  *cmsgp;
    char message[MAX_MSGLEN+1];
    char inmsg[MAX_MSGLEN+1];
    int on = 1;

    setsockopt(sockfd, SOL_SOCKET, SO_RECVUCRED, (void *)&on,
        sizeof (int));

    [...]

    while (1) {
        if (recvmsg(sockfd, &recv_msg, 0) < 0) {
            (void) fprintf(stderr, "recvmsg_errno:   %d\n", errno);
            exit(1);
            }

            /*
             * Check ucred in ancillary data
             */
            ucred = NULL;

            for (cmsgp = CMSG_FIRSTHDR(&recv_msg); cmsgp;
                cmsgp = CMSG_NXTHDR(&recv_msg, cmsgp)) {
                if (cmsgp->cmsg_level == SOL_SOCKET &&
                    cmsgp->cmsg_type == SCM_UCRED) {
                    ucred = (ucred_t *)CMSG_DATA(cmsgp);
                        break;
                        }

                if (ucred == NULL) {
                    (void) sprintf(&message[0],
                        "No ucred info in ancillary data with UDP");
                } else {
                    /*
                     * You might want to extract the label from the
                     * ucred by using ucred_getlabel(3C) here.
                     */
                }

            }

            [...]

            if (message != NULL)
                (void) strlcpy(&inmsg[0], message, MAX_MSGLEN);
            /*
             * Use the received message so that it will contain
             * the correct label
             */
            iov.iov_len = strlen(inmsg);
            ret = sendmsg(sockfd, &recv_msg, 0);
        }
}
```

♦ ♦ ♦ CHAPTER 5

Trusted X Window System

This chapter describes the Trusted Extensions X Window System APIs. This chapter also includes a short Motif application that is used to describe the Trusted X Window System security policy and the Trusted Extensions interfaces.

This chapter covers the following topics:

Trusted X Window System Environment

A system that is configured with Trusted Extensions uses the Trusted Extensions X Window System. The Trusted Extensions X Window System includes protocol extensions to support mandatory access control (MAC), discretionary access control (DAC), and the use of privileges.

Data transfer sessions are *polyinstantiated*, meaning that they are instantiated at different sensitivity labels and user IDs. Polyinstantiation ensures that 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. Such a transfer might violate the Trusted X Window System DAC policies and the MAC policies of write-equal and read-down.

The Trusted Extensions X Window System APIs enable you to obtain and set security-related attribute information. These APIs also enable you to translate labels to strings by using a font list and width to apply a style to the text string output. For example, the font might be 14-point, bold Helvetica. These interfaces are usually called by administrative applications.

- **Obtaining security-related information –** These interfaces operate at the Xlib level where X protocol requests are made. Use Xlib interfaces to obtain data for the input parameter values.
- **Translating labels to strings –** These interfaces operate at the Motif level. The input parameters are the label, a font list that specifies the appearance of the text string output, and the desired width. A compound string of the specified style and width is returned.

For declarations of these routines, see "Trusted Extensions X Window System APIs" on page 58.

Trusted X Window System Security Attributes

The Trusted X Window System interfaces manage security-related attribute information for various X Window System objects. Use the GNOME Toolkit or Motif to create a GUI application. The Motif application must use XToolkit routines to retrieve the Xlib object IDs underlying the Motif widgets to handle security attribute information for an Xlib object.

The X Window System 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 the window, property, display, and client connection IDs.

A window displays output to the user and accepts input from clients.

A property is an arbitrary collection of data that is accessed by the property name. Property names and property types can be referenced by an *atom*, which is a unique, 32-bit identifier and a character name string.

The security attributes for windows, properties, and client connections consist of ownership IDs and sensitivity label information. For information about the structures for capturing some of these attributes, see "Data Types for X11" on page 59. For information about the interfaces that obtain and set security attribute information, see "Trusted Extensions X Window System APIs" on page 58.

Trusted X Window System Security Policy

Window, property, and pixmap objects have a user ID, a client ID, and a sensitivity 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, an X Window Server ID, and a sensitivity 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 DAC policy requires a client to own an object to perform any operations on that 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.

The MAC policy is write-equal for windows and pixmaps, and read-equal for naming windows. The MAC policy is read-down for properties. The sensitivity label is set to the sensitivity label of the creating client. The following shows the MAC policy for these actions:

- **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.

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 MAC and DAC restrictions. To specify properties that are not polyinstantiated, update the `/usr/lib/xorg/TrustedExtensionsPolicy` file.

These sections describe the security policy for the following:

- Root window
- Client windows
- Override-redirect windows
- Keyboard, pointer, and server control
- Selection Manager
- Default window resources
- Moving data between windows

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 it 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. This prevents 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 MAC and DAC to gain control of the keyboard, pointer, and server. To reset the focus, a client must own the focus or have the `win_devices` privilege in its effective set.

To warp a pointer, the client needs pointer control and MAC and DAC to the destination window. X and Y coordinate information can be obtained for events that involve explicit user action.

Selection Manager

The Selection Manager application arbitrates user-level interwindow data moves, such as cut and paste or drag and drop, where information is transferred between untrusted windows. When a transfer is attempted, the Selection Manager captures the transfer, verifies the controlling user's authorization, and requests confirmation and labeling information from the user. Any time the user attempts a data move, the Selection Manager automatically appears. You do not need to update your application code to get the Selection Manager to appear.

The administrator can set automatic confirmation for some transfer types, in which case the Selection Manager does not appear. If the transfer meets the MAC and DAC policies, the data transfer completes. The File Browser and the window manager also act as selection agents for their private drop sites. See the /usr/lib/xorg/TrustedExtensionsPolicy file to specify selection targets that are polyinstantiated. See the /usr/share/gnome/sel_config file to determine which selection targets are automatically confirmed.

Default Window Resources

Resources that are not created by clients are default resources that are protected at ADMIN_LOW. Only clients that run at ADMIN_LOW or with the appropriate privileges can modify default resources.

The following are window resources:

- **Root window attributes –** All clients have read and create access, but only privileged clients have write or modify access. See "Privileged Operations and the Trusted X Window System" on page 57.
- **Default cursor –** Clients are free to reference the default cursor in protocol requests.
- **Predefined atoms –** The TrustedExtensionsPolicy file contains a read-only list of predefined atoms.

Moving Data Between Windows

A client needs the win_selection privilege in its effective set to move data between one window and another window without going through the Selection Manager. See "Selection Manager" on page 56.

Privileged Operations and the Trusted X Window System

Library routines that access a window, property, or atom name without user involvement require MAC and DAC. Library routines that access frame buffer graphic contexts, fonts, and cursors require discretionary access and might also require additional privileges for special tasks.

The client might 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 the TrustedExtensionsPolicy file to enable or disable these privileges.

This list shows the privileges needed to perform the following tasks:

- **Configuring and destroying window resources –** A client process needs the `win_config` privilege in its effective set to configure or destroy windows or properties that are permanently retained by the X Window Server. The screen saver timeout is an example of such a resource.
- **Using window input devices –** A client process needs the `win_devices` privilege in its effective set to obtain and set keyboard and pointer controls, or to modify pointer button mappings and key mappings.
- **Using direct graphics access –** A client process needs the `win_dga` privilege in its effective set to use the direct graphics access (DGA) X protocol extension.
- **Downgrading window labels –** A client process needs the `win_downgrade_sl` privilege in its effective set to change the sensitivity label of a window, pixmap, or property to a new label that does not dominate the existing label.
- **Upgrading window labels –** A client process needs the `win_upgrade_sl` privilege in its effective set to change the sensitivity label of a window, pixmap, or property to a new label that dominates the existing label.
- **Setting a font path on a window –** A client process needs the `win_fontpath` privilege in its effective set to modify the font path.

Trusted Extensions X Window System APIs

To use the Trusted X11 APIs, you need the following header file:

```
#include <X11/extensions/Xtsol.h>
```

The Trusted X11 examples compile with the `-lXtsol` and `-ltsol` library options.

To use the X11 label-clipping APIs, you need the following header file:

```
#include <Dt/label_clipping.h>
```

The label-clipping examples compile with the `-lDtTsol` and `-ltsol` library options.

The following sections provide data types and declarations for the Trusted X11 interfaces and the X11 label-clipping interfaces:

- Data types for X11
- Accessing attributes
- Accessing and setting a window label

- Accessing and setting a window user ID
- Accessing and setting a window property label
- Accessing and setting a window property user ID
- Accessing and setting a workstation owner ID
- Setting the X Window Server clearance and minimum label
- Working with the Trusted Path window
- Accessing and setting the screen stripe height
- Setting window polyinstantiation information
- Working with the X11 label-clipping interface

Data Types for X11

The following data types are defined in `X11/extensions/Xtsol.h` and are used for the Trusted Extensions X Window System APIs:

- **Object type for X11 –** The `ResourceType` definition indicates the type of resource to be handled. The value can be `IsWindow`, `IsPixmap`, or `IsColormap`.

  `ResourceType` is a type definition to represent a clearance. Interfaces accept a structure of type `m_label_t` as parameters and return clearances in a structure of the same type.
- **Object attributes for X11 –**The `XTsolResAttributes` structure contains these resource attributes:

  ```
  typedef struct _XTsolResAttributes {
      CARD32     ouid;    /* owner uid */
      CARD32     uid;     /* uid of the window */
      m_label_t  *sl;     /* sensitivity label */
  } XTsolResAttributes;
  ```
- **Property attributes for X11 –** The `XTsolPropAttributes` structure contains these property attributes:

  ```
  typedef struct _XTsolPropAttributes {
      CARD32     uid;     /* uid of the property */
      m_label_t  *sl;     /* sensitivity label */
  } XTsolPropAttributes;
  ```
- **Client attributes for X11 –** The `XTsolClientAttributes` structure contains these client attributes:

  ```
  typedef struct _XTsolClientAttributes {
      int      trustflag; /* true if client masked as trusted */
      uid_t    uid;       /* owner uid who 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 addr of host where client is running */
    } XTsolClientAttributes;
```

Accessing Attributes

The following routines are used to access resource, property, and client attributes:

`Status XTSOLgetResAttributes(Display *display, XID object, ResourceType type, XTSOLResAttributes *winattrp);`

> This routine returns the resource attributes for a window ID in *winattrp*. See the `XTSOLgetResAttributes`(3XTSOL) man page.

`Status XTSOLgetPropAttributes(Display *display, Window window, Atom property, XTSOLPropAttributes *propattrp);`

> This routine returns the property attributes for a property hanging on a window ID in *propattrp*. See the `XTSOLgetPropAttributes`(3XTSOL) man page.

`Status XTSOLgetClientAttributes(Display *display, XID windowid, XTsolClientAttributes *clientattrp);`

> This routine returns the client attributes in *clientattrp*. See the `XTSOLgetClientAttributes`(3XTSOL) man page.

Accessing and Setting a Window Label

The `XTSOLgetResLabel()` and `XTSOLsetResLabel()` routines are used to obtain and set the sensitivity label of a window.

`Status XTSOLgetResLabel(Display *display, XID object, ResourceType type, m_label_t *sl);`

> This routine obtains the sensitivity label of a window. See the `XTSOLgetResLabel`(3XTSOL) man page.

`Status XTSOLsetResLabel(Display *display, XID object, ResourceType type, m_label_t *sl);`

> This routine sets the sensitivity label of a window. See the `XTSOLsetResLabel`(3XTSOL) man page.

Accessing and Setting a Window User ID

The XTSOLgetResUID() and XTSOLsetResUID() routines are used to obtain and set the user ID of a window.

```
Status XTSOLgetResUID(Display *display, XID object, ResourceType type, uid_t *uidp);
```

This routine obtains the user ID of a window. See the XTSOLgetResUID(3XTSOL) man page.

```
Status XTSOLsetResUID(Display *display, XID object, ResourceType type, uid_t *uidp);
```

This routine sets the user ID of a window. See the XTSOLsetResUID(3XTSOL) man page.

Accessing and Setting a Window Property Label

The XTSOLgetPropLabel() and XTSOLsetPropLabel() routines are used to obtain and set the sensitivity label of a property hanging on a window ID.

```
Status XTSOLgetPropLabel(Display *display, Window window, Atom property, m_label_t *sl);
```

This routine obtains the sensitivity label of a property hanging on a window ID. See the XTSOLgetPropLabel(3XTSOL) man page.

```
Status XTSOLsetPropLabel(Display *display, Window window, Atom property, m_label_t *sl);
```

This routine sets the sensitivity label of a property hanging on a window ID. See the XTSOLsetPropLabel(3XTSOL) man page.

Accessing and Setting a Window Property User ID

The XTSOLgetPropUID() and XTSOLsetPropUID() routines are used to obtain and set the user ID of a property hanging on a window ID.

```
Status XTSOLgetPropUID(Display *display, Window window, Atom property, uid_t *uidp);
```

This routine obtains the user ID of a property hanging on a window ID. See the XTSOLgetPropUID(3XTSOL) man page.

```
Status XTSOLsetPropUID(Display *display, Window window, Atom property, uid_t *uidp);
```

This routine sets the user ID of a property hanging on a window ID. See the XTSOLsetPropUID(3XTSOL) man page.

Accessing and Setting a Workstation Owner ID

The XTSOLgetWorkstationOwner() and XTSOLsetWorkstationOwner() routines are used to obtain and set the user ID of the owner of the workstation server.

Note - The XTSOLsetWorkstationOwner() routine should only be used by the window manager.

```
Status XTSOLgetWorkstationOwner(Display *display, uid_t *uidp);
```

This routine obtains the user ID of the owner of the workstation server. See the XTSOLgetWorkstationOwner(3XTSOL) man page

```
Status XTSOLsetWorkstationOwner(Display *display, uid_t *uidp);
```

This routine sets the user ID of the owner of the workstation server. See the XTSOLsetWorkstationOwner(3XTSOL) man page.

Setting the X Window Server Clearance and Minimum Label

The XTSOLsetSessionHI() and XTSOLsetSessionLO() routines are used to set the session high clearance and the session low minimum label for the X Window Server. Session high must be within the user's range. Session low is the same as the user's minimum label for the multilevel session.

Note - These interfaces should only be used by the window manager.

```
Status XTSOLsetSessionHI(Display *display, m_label_t *sl);
```

The session high clearance is set from the workstation owner's clearance at login. The session high clearance must be dominated by the owner's clearance and by the upper bound of the system's monitor's label range. Once changed, connection requests from clients that run at a sensitivity label higher than the window server clearance are rejected unless they have privileges. See the XTSOLsetSessionHI(3XTSOL) man page.

```
Status XTSOLsetSessionLO(Display *display, m_label_t *sl);
```

The session low minimum label is set from the workstation owner's minimum label at login. The session low minimum label must be greater than the user's administratively set minimum label and the lower bound of the system's monitor's label range. When this setting is changed, connection requests from clients that run at a sensitivity label lower than the window server sensitivity label are rejected unless they have privileges. See the XTSOLsetSessionLO(3XTSOL) man page.

Working With the Trusted Path Window

The XTSOLMakeTPWindow() and XTSOLIsWindowTrusted() routines are used to make the specified window the Trusted Path window and to test whether the specified window is the Trusted Path window.

```
Status XTSOLMakeTPWindow(Display *display, Window *w);
```

This routine makes the specified window the Trusted Path window. See the XTSOLMakeTPWindow(3XTSOL) man page.

```
Bool XTSOLIsWindowTrusted(Display *display, Window *window);
```

This routine tests whether the specified window is the Trusted Path window. See the XTSOLIsWindowTrusted(3XTSOL) man page.

Accessing and Setting the Screen Stripe Height

The XTSOLgetSSHeight() and XTSOLsetSSHeight() routines are used to obtain and set the screen stripe height.

Note - These interfaces should only be used by the window manager.

```
Status XTSOLgetSSHeight(Display *display, int screen_num, int *newHeight);
```

This routine obtains the screen stripe height. See the XTSOLgetSSHeight(3XTSOL) man page.

```
Status XTSOLsetSSHeight(Display *display, int screen_num, int newHeight);
```

This routine sets the screen stripe height. Be careful that you do not end up without a screen stripe or with a very large screen stripe. See the XTSOLsetSSHeight(3XTSOL) man page.

Setting Window Polyinstantiation Information

```
Status XTSOLsetPolyInstInfo(Display *display, m_label_t sl, uid_t *uidp, int enabled);
```

The `XTSOLsetPolyInstInfo()` routine enables a client to obtain property information from a property at a different sensitivity label than the client. In the first call, you specify the desired sensitivity label and the user ID, and set the `enabled` property to True. Then, you call `XTSOLgetPropAttributes()`, `XTSOLgetPropLabel()`, or `XTSOLgetPropUID()`. To finish, you call the `XTSOLsetPolyInstInfo()` routine again with the `enabled` property set to False. See the `XTSOLsetPolyInstInfo`(3XTSOL) man page.

Working With the X11 Label-Clipping Interface

```
int label_to_str(const m_label_t *label, char **string, const m_label_str_t
conversion_type, uint_t flags);
```

The `label_to_str()` routine translates a sensitivity label or clearance to a string. See the `label_to_str`(3TSOL) man page.

Using Trusted X Window System Interfaces

The following sections provide example code excerpts that use Trusted Extensions interface calls. These calls handle security attributes and translate a label to a string. The excerpts focus on handling window security attributes, the most commonly managed attributes in application programs. Often, a client retrieves security attributes by using the appropriate privileges for an object that was created by another application. The client then checks the attributes to determine whether an operation on the object is permitted by the system's security policy. The security policy covers DAC policies and the MAC write-equal and read-down policies. If access is denied, the application generates an error or uses privileges, as appropriate. See "Privileged Operations and the Trusted X Window System" on page 57 for a discussion about when privileges are needed.

You must create an object before you can retrieve its ID to pass to the Trusted Extensions APIs.

Obtaining Window Attributes

The XTSOLgetResAttributes() routine returns security-related attributes for a window. You supply the following:

- Display ID
- Window ID
- Flag to indicate that the object for which you want security attributes is a window
- XtsolResAttributes structure to receive the returned attributes

Because the client is obtaining the security attributes for a window that the client created, no privileges are required.

Note that the example programs in this book focus on the APIs being shown and do not perform error checking. Your applications should perform the appropriate error checking.

```
/* 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 = label_to_str(&winattrs.sl, &plabel, M_LABEL, LONG_NAMES);

/* Print security attribute information */
printf("Workstation Owner ID = %d\nUser ID = %d\nLabel = %s\n",
winattrs.ouid, winattrs.uid, string1);
```

The printf statement prints the following:

```
Workstation Owner ID = 29378
User ID = 29378
Label = CONFIDENTIAL
```

Translating the Window Label With the Font List

This example shows how to obtain the process sensitivity label and translate it to a string by using a font list and the pixel width. A label widget is created with the string for its label. The process sensitivity label equals the window sensitivity label. Therefore, no privileges are required.

When the final string is longer than the width, the string is clipped and the clipped indicator is used. Note that the X Window System label-translation interfaces clip to the specified number of pixels, while the label-clipping interfaces clip to the number of characters.

Note - 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. The following example does not work for the Asian character set.

```
 retval = getplabel(&senslabel);

/* 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);
```

Obtaining a Window Label

This example shows how to obtain the sensitivity label for a window. The process sensitivity label equals the window sensitivity label. Therefore, no privileges are required.

```
/* Retrieve window label */
retval = XTSOLgetResLabel(display, window, IsWindow, &senslabel);

/* Translate labels to string and print */
retval = label_to_str(label, &string, M_LABEL, LONG_NAMES);
printf("Label = %s\n", string);
```

The `printf` statement, for example, prints the following:

```
Label = PUBLIC
```

Setting a Window Label

This example shows how to set the sensitivity label on a window. The new sensitivity label dominates the sensitivity label of the window and the process. The client needs the `sys_trans_label` privilege in its effective set to translate a label that the client does not dominate. The client also needs the `win_upgrade_sl` privilege to change the window's sensitivity label.

For more information about using privileges, see *Developer's Guide to Oracle Solaris 11.3 Security*.

```
/* Translate text string to sensitivity label */
retval = label_to_str(string4, &label, M_LABEL, L_NO_CORRECTION, &error);

/* Set sensitivity label with new value */
retval = XTSOLsetResLabel(display, window, IsWindow, label);
```

Obtaining the Window User ID

This example shows how to obtain the window user ID. The process owns the window resource and is running at the same sensitivity label. Therefore, no privileges are required.

```
/* Get the user ID of the window */
retval = XTSOLgetResUID(display, window, IsWindow, &uid);
```

Obtaining the X Window Server Workstation Owner ID

This example shows how to obtain the ID of the user who is logged in to the X Window Server. The process sensitivity label equals the window sensitivity label. Therefore, no privileges are required.

```
/* Get the user ID of the window */
retval = XTSOLgetWorkstationOwner(display, &uid);
```

♦ ♦ ♦ CHAPTER 6

Label Builder GUI

Trusted Extensions provides the tgnome-selectlabel utility, which creates an interactive label builder GUI. You can use this utility in an administrative application to produce a GUI that enables users to build valid sensitivity labels or clearances.

This chapter covers the following topics:

- “tgnome-selectlabel Utility” on page 69
- “Adding the Label Builder GUI to an Administration Application” on page 71

tgnome-selectlabel Utility

The tgnome-selectlabel utility enables you to create a label builder GUI with which you can specify label information. You *must* specify both the minimum label and the maximum label when you use the utility to create a label builder GUI.

```
tgnome-selectlabel -n min-label -x max-label [-a yes|no ] [-d label-string] \
[-f internal|human] [-h] [-i "window-title"] [-m clearance|sensitivity] \
[-t "operation-description"]
```

The clearance and sensitivity label information that is used by the label builder GUI is specified in the label_encodings file.

The tgnome-selectlabel utility supports the following options:

-a yes|no

Enables accreditation checking, which determines whether the label is well formed and if it is within the accreditation range for the labeled zone, user, or resource. The default value is no.

You generally use the accreditation check to assign a label to a user, and not to assign a label to a zone or a template. Labels such as admin_low and admin_high, and any labels explicitly stated as not valid in the `label_encodings` file, are not valid when accreditation checking is enabled.

`-d` *label-string*

Specifies the label to select by default in the label builder GUI. This label must be within the label range bounded by the specified minimum and maximum labels. The default value is `TRUSTED PATH`.

`-f internal|human`

Specifies whether to use the internal or human-readable label format. The internal label format is a hexadecimal representation of the label, which is safe for storing in a public directory. The human-readable label format is formed according to the rules in the `label_encodings` file and is suitable for user interfaces. The default value is `internal`.

`-i` *window-title*

Specifies an optional window title for the label builder that you create. The default value is `Label Selector`.

`-h`

Shows the `tgnome-selectlabel` usage statement.

`-m clearance|sensitivity`

Specifies whether to use clearance mode or sensitivity mode. Clearances correspond to the `CLEARANCES` section in the `label_encodings` file. The words used in the clearances might have different spellings or prefixes. Some labels might be valid as clearances but not as sensitivity labels. When you assign labels to users, the minimum label must be specified as a sensitivity label, and the user's clearance must be specified as a clearance. You must specify a sensitivity label for zones. The default value is `sensitivity`.

`-n` *min-label*

Specifies the minimum label. You must specify the minimum label.

`-t` *operation-description*

Specifies a text description of the operation to be performed. The default value is `Select a Label`.

`-x` *max-label*

Specifies the maximum label. You must specify the maximum label.

The following table shows both the short and long option formats. When you use the long option format, you must use an equal sign between the option name and the specified value. For example, `--format=internal` uses the internal label format.

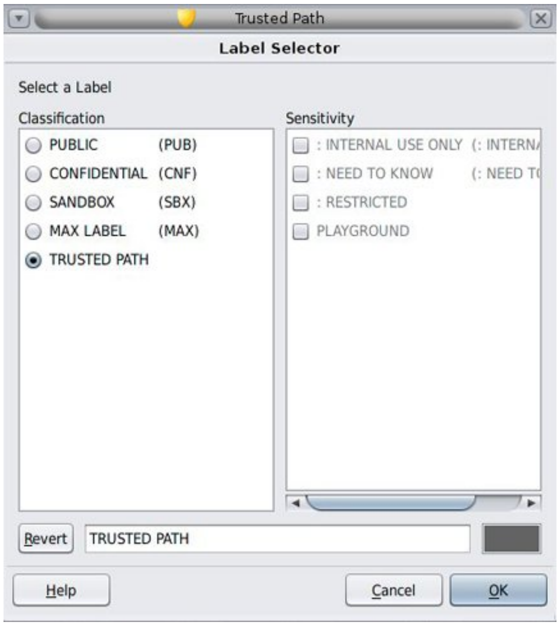
Short Option Form	Long Option Form	Short Option Form	Long Option Form
`-a`	`--accredcheck`	`-m`	`--mode`
`-d`	`--default`	`-n`	`--min`
`-f`	`--format`	`-t`	`--text`
`-h`	`--help`	`-x`	`--max`
`-i`	`--title`		

Adding the Label Builder GUI to an Administration Application

Figure 1, "Label Builder GUI," on page 71 shows the label builder GUI that is created by the following `tgnome-selectlabel` command:

```
# tgnome-selectlabel -n admin_low -x admin_high
```

FIGURE 1 Label Builder GUI



Select a Label

Classification

○ PUBLIC (PUB)
○ CONFIDENTIAL (CNF)
○ SANDBOX (SBX)
○ MAX LABEL (MAX)
● TRUSTED PATH

Sensitivity

□ : INTERNAL USE ONLY (: INTERN
□ : NEED TO KNOW (: NEED T
□ : RESTRICTED
□ PLAYGROUND

Revert TRUSTED PATH

The label builder GUI in the figure includes the Classification and Sensitivity lists, the default label, and the Revert, Help, Cancel, and OK buttons.

You can build a sensitivity label or a clearance by selecting radio buttons, check boxes, or both from the classifications and compartments that are shown in the Classification and Sensitivity lists.

The information about the classifications, compartments, and markings is specified in the `label_encodings` file for the system. The combinations and constraints that are specified in the `label_encodings` file are enforced by graying out invalid combinations.

The Revert button resets the text in the adjacent field to the value that was set when the application started.

The Label Selector dialog box shows only those classifications and related compartments that are dominated by the workspace sensitivity label. If the executable has the `sys_trans_label` privilege in its effective set, more classifications and compartments might be shown.

If a user does not have the authorization to upgrade or downgrade labels, the OK and Revert buttons are grayed out. The same is true if the user-built label is outside of the user's label range. The grayed-out buttons prevent the user from completing the task. No privileges can override these restrictions.

Using the Label Builder GUI in the `txzonemgr` Command

The Trusted Extensions `txzonemgr` command uses the `tgnome-selectlabel` utility to create label builder GUIs that help the user specify labels for the following:

- Labeled zones
- Multilevel ports

The following examples show how the `tgnome-selectlabel` utility is used in the `txzonemgr` command.

EXAMPLE 3 `selectLabel()` Function

The `selectLabel()` function uses the `tgnome-selectlabel` utility to create a label builder GUI that helps the user select the label for a new labeled zone.

```
title="Labeled Zone Manager 2.2"
msg_getlabel=${gettext "Select a label for the")
```

```
...

selectLabel() {
        hexlabel=$(tgnome-selectlabel \
                --title="$title" \
                --text="$msg_getlabel $zonename zone:" \
                --min="${DEFAULTLABEL}"  \
                --default="${DEFAULTLABEL}"  \
                --max=$(chk_encodings -X) \
                --accredcheck=yes \
                --mode=sensitivity \
                --format=internal \
                2>/dev/null)
        if [ $? = 0 ] ; then
                setTNdata
                if [ $? -ne 0 ] ; then
                        return 1
                fi
        fi
    }
```

EXAMPLE 4 `getLabelRange()` Function

The `getLabelRange()` function uses the `tgnome-selectlabel` utility to select the minimum and maximum labels for a multilevel port.

```
title="Labeled Zone Manager 2.2"
msg_getmin=$(gettext "Select the minimum network label for the")
msg_getmax=$(gettext "Select the maximum network label for the")

...

getLabelRange() {
        deflabel=$(tncfg -t $cipso_template info min_label|cut -d= -f2)
        minlabel=$(tgnome-selectlabel \
                --title="$title" \
                --text="$msg_getmin $zonename zone:" \
                --min="admin_low" \
                --max="$hexlabel" \
                --default="$deflabel" \
                --accredcheck=no \
                --mode=sensitivity \
                --format=internal \
                2>/dev/null)
        [ $? != 0 ] && return
        deflabel=$(tncfg -t $cipso_template info max_label|cut -d= -f2)
        maxlabel=$(tgnome-selectlabel \
                --title="$title" \
                --text="$msg_getmax $zonename zone:" \
                --min="${minlabel}"  \
                --max="admin_high" \
                --default="$deflabel" \
                --accredcheck=no \
                --mode=sensitivity \
                --format=internal \
                2>/dev/null)
        [ $? != 0 ] && return

        hostType=cipso
        updateTemplate
}
```

♦ ♦ ♦ CHAPTER 7

Trusted Web Guard Prototype

This chapter describes the configuration of a safe web browsing prototype called Web Guard. Web Guard is configured to isolate a web server and its web content to prevent attacks from the Internet.

The Web Guard prototype described in this chapter is not a complete solution. Rather, the prototype is intended to demonstrate how multilevel ports can be used to proxy URL requests across label boundaries. A more complete solution would include authentication, data filtering, auditing, and so on.

The primary implementation of the prototype is administrative. The prototype uses multilevel ports, trusted networking, and Apache web server configuration to set up Web Guard. In addition to the administrative example, you can use some programmatic methods to set up the safe web browsing prototype.

This chapter covers the following topics:

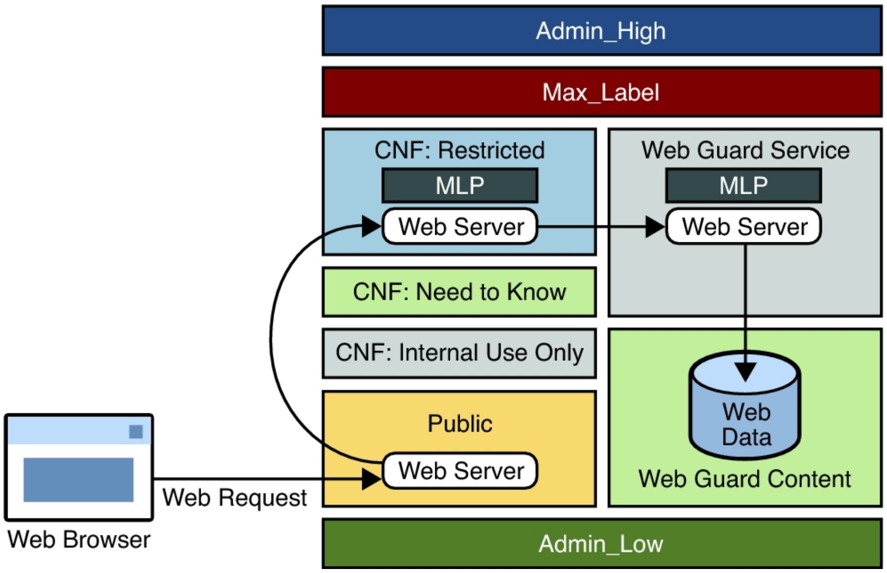
- "Administrative Web Guard Prototype" on page 75
- "Accessing Lower-Level Untrusted Servers" on page 82

Administrative Web Guard Prototype

This section provides an example of a safe web browsing prototype that isolates a web server and its web content to prevent attacks from the Internet. This Web Guard prototype takes advantage of administrative trusted networking features to configure a two-stage filter that restricts access to a protected web server and web content. This prototype was implemented solely by administrative means. No programming was required.

The following figure shows the configuration of the Web Guard prototype in a multilevel environment. The label relationships are shown by how the labels are positioned in the figure. Vertical relationships represent label dominance, while horizontal relationships represent disjoint labels.

FIGURE 2 Web Guard Configuration



Web requests come in to the web server that is configured in the `public` zone and are passed to the web server that is configured in the `restricted` zone.

The `restricted` zone uses a multilevel port (MLP) to listen for requests at port `8080` of the `public` zone. This web server passes the requests to the `webservice` labeled zone.

The `webservice` zone also uses an MLP to listen for requests at port `80` of the `restricted` zone and reads content from the `webcontent` labeled zone.

The `webcontent` zone is in the ready state and has its web content stored in the `/export/home` file system, which is automatically mounted in all other labeled zones. When a zone is in the ready state, no processes run in that zone. Thus, the zone is essentially a disk drive attached directly to the `webservice` zone.

You configure the Web Guard prototype by performing these high-level tasks:

1. Modifying the `label_encodings` file to configure the labels in your safe web browsing environment

   The default `label_encodings` file is updated to configure two new labels: `WEB GUARD SERVICE` and `WEB GUARD CONTENT`. See "Modifying the `label_encodings` File" on page 77.

2. Configuring trusted networking

   The private IP addresses and MLPs are configured on the restricted and webservice labeled zones. See "Configuring Trusted Networking" on page 80.
3. Configuring the Apache web servers

   The public, restricted, and webservice zones all have web servers configured. In this example, the web server used is Apache. See "Configuring the Apache Web Servers" on page 81.

Modifying the label_encodings File

The default label_encodings file is updated to configure two new labels: WEB GUARD SERVICE and WEB GUARD CONTENT. The SANDBOX label, which is part of the default file, is modified to serve as the WEB GUARD CONTENT label. The WEB GUARD SERVICE label is added.

You must install the label_encodings file in the /etc/security/tsol directory. You can install this file on top of an existing Trusted Extensions installation.

After you install the updated file in the /etc/security/tsol directory, activate the new label_encodings file:

```
# svcadm restart svc:/system/labeld
```

The following shows the label_encodings file used in this Web Guard prototype.

```
* ident "@(#)label_encodings.simple 5.15 05/08/09 SMI"
*
* Copyright 2005 Sun Microsystems, Inc.  All rights reserved.
* Use is subject to license terms.
*
* This example shows how to specify labels that meet an actual
* site's legal information  protection requirements for
* labeling email and printer output. These labels may also
* be used to enforce mandatory access control checks based on user
* clearance labels and sensitivity labels on files and directories.

VERSION= Sun Microsystems, Inc. Example Version - 6.0. 2/15/05

CLASSIFICATIONS:

name= PUBLIC; sname= PUB; value= 2; initial compartments= 4;
name= CONFIDENTIAL; sname= CNF; value= 4; initial compartments= 4;
name= WEB GUARD; sname= WEB; value= 5; initial compartments= 0;
name= MAX LABEL; sname= MAX; value= 10; initial compartments= 0 4 5;
```

```
INFORMATION LABELS:

WORDS:

name= :; prefix;

name= INTERNAL USE ONLY; sname= INTERNAL; compartments= 1 ~2; minclass= CNF;
name= NEED TO KNOW; sname= NEED TO KNOW; compartments= 1-2 ~3; minclass= CNF;
name= RESTRICTED; compartments= 1-3; minclass= CNF;
name= CONTENT; compartments= 0 ~1 ~2 ~3; minclass= WEB;
name= SERVICE; compartments= 5; minclass= WEB;

REQUIRED COMBINATIONS:

COMBINATION CONSTRAINTS:

SENSITIVITY LABELS:

WORDS:

name= :; prefix;

name= INTERNAL USE ONLY; sname= INTERNAL; compartments= 1 ~2; minclass= CNF;
prefix= :

name= NEED TO KNOW; sname= NEED TO KNOW; compartments= 1-2 ~3; minclass= CNF;
prefix= :

name= RESTRICTED; compartments= 1-3; minclass= CNF; prefix= :

name= CONTENT; compartments= 0 ~1 ~2 ~3; minclass= WEB;

name= SERVICE; compartments= 5; minclass= WEB;

REQUIRED COMBINATIONS:

COMBINATION CONSTRAINTS:

CLEARANCES:

WORDS:

name= INTERNAL USE ONLY; sname= INTERNAL; compartments= 1 ~2; minclass= CNF;
name= NEED TO KNOW; sname= NEED TO KNOW; compartments= 1-2 ~3; minclass= CNF;
name= RESTRICTED; sname= RESTRICTED; compartments= 1-3; minclass= CNF;
name= CONTENT; compartments= 0 ~1 ~2 ~3; minclass= WEB;
name= SERVICE; compartments= 5; minclass= WEB;
```

```
REQUIRED COMBINATIONS:

COMBINATION CONSTRAINTS:

CHANNELS:

WORDS:

PRINTER BANNERS:

WORDS:

ACCREDITATION RANGE:

classification= PUB; all compartment combinations valid;
classification= WEB; all compartment combinations valid;
classification= CNF; all compartment combinations valid except: CNF

minimum clearance= PUB;
minimum sensitivity label= PUB;
minimum protect as classification= PUB;

* Local site definitions and locally configurable options.

LOCAL DEFINITIONS:

default flags= 0x0;
forced flags= 0x0;

Default Label View is Internal;

Classification Name= Classification;
Compartments Name= Sensitivity;

Default User Sensitivity Label= PUB;
Default User Clearance= CNF NEED TO KNOW;

COLOR NAMES:

    label= Admin_Low;           color= #bdbdbd;

    label= PUB;                 color= blue violet;
    label= WEB SERVICE;         color= yellow;
    label= CNF;                 color= navy blue;
    label= CNF : INTERNAL USE ONLY;  color= blue;
    label= CNF : NEED TO KNOW;  color= #00bfff;
    label= CNF : RESTRICTED;    color= #87ceff;
    label= Admin_High;          color= #636363;

* End of local site definitions
```

For more information about the `label_encodings` file, see *Trusted Extensions Label Administration*.

Configuring Trusted Networking

The `restricted` and `webservice` zones are assigned a private IP address in addition to the IP address that they already share. Each private IP address has a multilevel port configured and is associated with a restricted label set.

The following table shows the network configuration for each of the labeled zones.

Zone Name	Zone Label	Local IP Address	Host Name	Multilevel Port	Security Label Set
restricted	CONFIDENTIAL : RESTRICTED	192.0.2.6	proxy	8080/tcp	PUBLIC
webservice	WEB GUARD SERVICE	192.0.2.3	webservice	80/tcp	CONFIDENTIAL : RESTRICTED
webcontent	WEB GUARD CONTENT	None			

First, you must create the new zones. You can clone an existing zone, such as the `public` zone. After these zones are created, use the `zonecfg` command to add a network (with the address specified in the table) and your local interface name.

For example, the following command associates the `192.0.2.6` IP address and the `net0` interface with the `restricted` zone:

```
# zonecfg -z restricted
add net
set address=192.0.2.6
set physical=net0
end
exit
```

After you specify the IP address and network interface for each labeled zone, use the `txzonemgr` script to configure the remaining values in the table. When you finish the configuration process, start or restart the affected zones. In the global zone, add routes for the new addresses, where *shared-IP-addr* is the shared IP address.

```
# route add proxy shared-IP-addr
# route add webservice shared-IP-addr
```

Configuring the Apache Web Servers

An instance of the Apache web server runs in the `public` zone, the `restricted` zone, and the `webservice` zone. The `/etc/apache2/httpd.conf` file is updated in each zone as follows:

- `public` **zone –** Specify the IP address or host name of the server for the `ServerName` keyword, and update the proxy configuration as follows:

  ```
  ServerName myserver

  ProxyRequests Off
  ProxyPass /demo http://proxy:8080/demo
  ProxyPassReverse /demo http://proxy:8080/demo
  ```
- `restricted` **zone –** Specify the listen proxy port and the port. Then, specify the IP address or host name of this zone for the `ServerName` keyword, and update the proxy configuration as follows:

  ```
  Listen proxy:8080
  Port 8080

  ServerName proxy

  ProxyRequests Off
  ProxyPass /demo http://webservice
  ProxyPassReverse /demo http://webservice
  ```

  You might also want to set up some filtering of the web requests, such as dirty word filters, or other filters to restrict the types of requests for web content.
- `webservice` **zone –** Specify the IP address or host name of this zone for the `ServerName` keyword, and point to the location of the web content directory in the `DocumentRoot` keyword and the `<Directory>` element as follows:

  ```
  ServerName webservice

  DocumentRoot "/zone/webcontent/export/home/www/htdocs"
  <Directory "/zone/webcontent/export/home/www/htdocs">
  ```

After you have updated the Apache web server configuration files for each labeled zone, store your web content in the `/export/home/www/htdocs` directory of the `webcontent` zone.

Create the demo directory in the /export/home/www/htdocs directory, and then create an index.html file in the demo directory to use for testing.

The /export/home directory is automatically mounted by using lofs into the webservice zone when it is booted. The webcontent zone only needs to brought up to the ready state.

```
# zoneadm -z webcontent ready
```

When a zone is in the ready state, no processes are running in that zone. The zone's file system can be mounted read-only by the webservice zone. Accessing the web content in this way ensures that the content cannot be changed.

Running the Trusted Web Guard Demonstration

From your browser in the public zone or from a remote browser running at the PUBLIC label, type the following URL:

http://*server-name*/demo

The browser should show the default index.html file from the webcontent zone.

Note that the Web Guard flow cannot by bypassed. The web server in the webservice zone cannot receive packets from the public zone or from any remote host. The web content cannot be changed because the webcontent zone is in the ready state.

Accessing Lower-Level Untrusted Servers

Sometimes a client needs to be able to access a server on an unlabeled system. An *unlabeled system* is a system that does not run the Trusted Extensions software. In such a case, you cannot use multilevel ports because they are restricted to privileged servers that run in the global zone or in labeled zones.

For example, suppose your browser is running in the INTERNAL zone. You want to access a web server that runs on a single-level network that has been assigned the PUBLIC sensitivity label by means of the tnrhdb database. Such access is not permitted by default. However, you could write a privileged proxy server to forward the HTTP request to the PUBLIC web server. The proxy should use a special Trusted Extensions socket option called SO_MAC_EXEMPT. This socket option permits a request to be sent to an untrusted lower-level service, and permits the reply from that service to be returned to the requester.

Note - The use of the SO_MAC_EXEMPT option represents an unprotected downgrade channel and should be used *very carefully*. The SO_MAC_EXEMPT option cannot be set unless the calling process has the PRIV_NET_MAC_AWARE privilege in its effective set. Such a process must enforce its own data filtering policy to prevent leaking higher-level data to the lower-level service. For example, the proxy should sanitize URLs to restrict words from being used as values.

The following code excerpt demonstrates the use of SO_MAC_EXEMPT in a modified version of the wget command's connect_to_ip() routine in connect.c. The call to setsockopt() has been added to show how to set the SO_MAC_EXEMPT option.

```
int
connect_to_ip (const ip_address *ip, int port, const char *print)
{
  struct sockaddr_storage ss;
  struct sockaddr *sa = (struct sockaddr *)&ss;
  int sock;
  int on = 1;

  /* If PRINT is non-NULL, print the "Connecting to..." line, with
     PRINT being the host name we're connecting to.  */
  if (print)
    {
      const char *txt_addr = pretty_print_address (ip);
      if (print && 0 != strcmp (print, txt_addr))
    logprintf (LOG_VERBOSE, _("Connecting to %s|%s|:%d... "),
          escnonprint (print), txt_addr, port);
      else
    logprintf (LOG_VERBOSE, _("Connecting to %s:%d... "), txt_addr, port);
    }

  /* Store the sockaddr info to SA.  */
    sockaddr_set_data (sa, ip, port);

  /* Create the socket of the family appropriate for the address.  */
    sock = socket (sa->sa_family, SOCK_STREAM, 0);
    if (sock < 0)
      goto err;

    if (setsockopt (sock, SOL_SOCKET, SO_MAC_EXEMPT, &on, sizeof (on)) == -1) {
      perror("setsockopt SO_MAC_EXEMPT");
    }

  #if defined(ENABLE_IPV6) && defined(IPV6_V6ONLY)
    if (opt.ipv6_only) {
      /* In case of error, we will go on anyway... */
      int err = setsockopt (sock, IPPROTO_IPV6, IPV6_V6ONLY, &on, sizeof (on));
    }
```

```
#endif
```

♦ ♦ ♦ CHAPTER 8

Experimental Java Bindings for the Trusted Extensions Label APIs

This chapter describes an experimental set of Java™ classes and methods that mirror the label application programming interfaces (APIs) that are provided with the Trusted Extensions software. The Java implementation of the Trusted Extensions label APIs is intended to be used to create label-aware applications. As a result, all of the label APIs provided by Trusted Extensions are not part of the Java implementation.

The presentation of these experimental Java APIs (Java bindings) demonstrate how the Trusted Extensions features can be expanded into the Java development environment.

Caution - These experimental Java bindings are *not* a supported part of the Trusted Extensions software.

This chapter covers the following topics:

- "Java Bindings Overview" on page 85
- "Structure of the Experimental Java Label Interfaces" on page 86
- "Java Bindings" on page 88

Java Bindings Overview

The Java language is an untapped resource for creating label-aware applications that run in secure, multilevel arenas. These experimental Java bindings provide a foundation on which to develop more applications, such as system audit log generation and system resource controls.

Adding platform services to the Java environment will enable Java applications to handle sensitive multilevel data.

Trusted Extensions provides label services through the label daemon, `labeld`. This daemon is available to processes that run in the global zone and in labeled zones.

The Java bindings described in this chapter are Java Native Interface (JNI™) implementations of some of the Trusted Extensions label APIs. The experimental JNI code calls the Trusted Extensions label library functions to extend some of the label functionality to the Java language. Constructors and methods in these Java classes call private JNI interfaces, written in C, that in turn call the Trusted Extensions APIs. For example, the `SolarisLabel.dominates` method calls a private JNI interface written in C that calls the `bldominates()` routine. These experimental Java bindings have been developed using Java 2 Platform, Standard Edition 5.0. For more information about JNI, see Java Native Interface Documentation (`http://download.oracle.com/javase/1.5.0/docs/guide/jni/`).

Structure of the Experimental Java Label Interfaces

The JNI implementation of the Trusted Extensions label APIs introduces several label-related classes that relate to each other in this way:

- `SolarisLabel` abstract class
  - `ClearanceLabel` subclass
  - `SensitivityLabel` subclass
- `Range` class

`SolarisLabel` Abstract Class

The `SolarisLabel` abstract class provides the foundation for common and native methods related to Trusted Extensions labels. The `SensitivityLabel` and `ClearanceLabel` subclasses inherit members from this abstract class. Static factories for creating sensitivity labels and clearance labels are also provided by the abstract class.

Static factories and methods throw exceptions when errors are encountered to ensure that no mandatory access control-related errors occur silently.

This abstract class defines the following general-purpose methods that are used to compare labels and to translate labels to strings:

- `dominates`
- `equals`
- `setFileLabel`
- `strictlyDominates`

- toColor
- toInternal
- toRootPath
- toString
- toText
- toTextLong
- toTextShort

The equals, dominates, and strictlyDominates methods are analogous to the blequal(), bldominates(), and blstrictdom() label APIs currently available with Trusted Extensions. The setFileLabel method is analogous to the setflabel() routine currently available with Trusted Extensions.

The rest of the methods (such as toText, toInternal, and toColor) are related in function to the label_to_str() routine that is currently available with Trusted Extensions. These methods enable you to translate a label to a particular type of string. Depending on the label relationship of the process and the object, you might need privileges in your effective set to translate a label to a human-readable form. For instance, the Java Virtual Machine (JVM™) process must be running with the sys_trans_label privilege to translate labels that it does not dominate.

The SolarisLabel abstract class also includes the following static factories:

- getClearanceLabel
- getFileLabel
- getSensitivityLabel
- getSocketPeer

The string that you pass as a label to getSensitivityLabel or getClearanceLabel can be in one of the following forms:

- Human-readable form of the label, such as PUBLIC
- Internal form of the label, such as 0x0002-08-08

Only the internal form of the label is suitable for storage and for transmission over a network connection, as the internal form does not reveal the actual label. For more information, see “Readable Versions of Labels” on page 34.

The ClearanceLabel and SensitivityLabel subclasses extend the SolarisLabel abstract class. These subclasses each inherit the common methods provided by the SolarisLabel abstract class.

`ClearanceLabel` Subclass

The `ClearanceLabel` subclass extends the `SolarisLabel` abstract class and defines the `getMaximum` and `getMinimum` methods, which return the `ClearanceLabel` object that represents the least upper bound and the greatest lower bound, respectively.

`SensitivityLabel` Subclass

The `SensitivityLabel` subclass extends the `SolarisLabel` abstract class and defines the `getMaximum` and `getMinimum` methods, which return the `SensitivityLabel` object that represents the least upper bound and the greatest lower bound, respectively.

`Range` Class

The `Range` class represents a Java version of a Trusted Extensions label range.

This class defines the following general-purpose methods that are used to obtain the upper and lower labels in a label range and to determine whether a label is within a specified label range:

- `getLower`
- `getUpper`
- `inRange`

The `Range` class also includes the following static factories that create range objects:

- `getDeviceRange`
- `getLabelRange`
- `getUserRange`

The `getDeviceRange` and `getUserRange` static factories create range objects based on the range for the specified device and the specified user, respectively. The `getLabelRange` static factory enables you to create a label range where you specify the upper and lower bounds for the range.

Java Bindings

The Java implementation of the Trusted Extensions label APIs is intended to be used to create label-aware applications. As a result, not all of the label APIs provided by Trusted Extensions are part of the Java implementation.

The Java classes and methods that are presented in this chapter mimic the following general label functionality shown in "Label APIs" on page 27:

- Detecting a Trusted Extensions system
- Accessing the process sensitivity label
- Allocating and freeing memory for label objects
- Obtaining and setting the label of a file
- Obtaining label range objects
- Accessing labels in zones
- Obtaining the remote host type
- Translating between labels and strings
- Comparing label objects

Detecting a Trusted Extensions System

These Java bindings do not include methods to determine whether a system is labeled. Rather, the library will fail to load if Trusted Extensions is not enabled.

Accessing the Process Sensitivity Label

These Java bindings do not include methods to obtain the label of a process. In Trusted Extensions, a process that runs in a labeled zone has the same label as the zone.

Allocating and Freeing Memory for Label Objects

These Java bindings take advantage of the Java "garbage-collection" functionality. As a result, you do not need to explicitly free the memory used by label objects as you do for the label APIs described in "Obtaining and Setting the Label of a File" on page 30.

Obtaining and Setting the Label of a File

These Java bindings use the Java `File` object to obtain and set file labels. Use the `getFileLabel` static factory to obtain the label from the file's `File` object. To set a file label to another specified label, use the `setFileLabel` method on the file's `File` object.

In addition to obtaining the sensitivity label of a file, the getSocketPeer static factory enables you to obtain the sensitivity label for the peer endpoint of a socket.

The getFileLabel static factory and the setFileLabel method correspond to the getlabel() system call and the setflabel() routine, respectively. For more information, see "Obtaining and Setting the Label of a File" on page 30 and the getlabel(2) and setflabel(3TSOL) man pages.

The following descriptions include the prototype declarations for the static factories and the method:

```
public static SensitivityLabel getFileLabel(java.io.File file)
```

The getFileLabel static factory obtains the label of a Java File object that is specified by *file*.

```
public static SensitivityLabel getSocketPeer(java.net.Socket socket)
```

The getSocketPeer static factory obtains a sensitivity label object from the specified socket, *socket*.

The following code fragment obtains the sensitivity label object of the socket, *s*:

```
SensitivityLabel sl = SolarisLabel.getSocketPeer(s);
```

The following example code shows how to create a server socket on port 9090 and then obtain the sensitivity label of the peer end of the accepted connection. This code example also outputs the internal and human-readable forms, the color, and the root path of the obtained socket peer label.

```
import java.io.*;
import java.net.*;
import solarismac.*;

public class ServerSocketTest
{

    public static void main (String args[]) {

        System.out.println("ServerSocketTest Start");

        CreateListener();

        System.out.println("ServerSocketTest End");

    }

    /*
    * Listen for connections on port then print the peer connection label.
```

```
        * You can use telnet host 9090 to create a client connection.
        */
        private static void CreateListener() {
            int port = 9090;

            ServerSocket acceptSocket;
            Socket s;
            try {
                System.out.println("Creating ServerSocket on port " + port);

                acceptSocket = new ServerSocket(port);

                System.out.println("ServerSocket created, waiting for connection");

                s = acceptSocket.accept();

                /*
                 * Get the Sensitivity Label for the peer end of the socket.
                 */
                SensitivityLabel socksl = SolarisLabel.getSocketPeer(s);

                System.out.println("Client connected...");
                System.out.println(" toInternal: " + socksl.toInternal());
                System.out.println(" toText: " + socksl.toText());
                System.out.println(" toString: " + socksl.toString());
                System.out.println(" toColor: " + socksl.toColor());
                System.out.println(" toRootPath: " + socksl.toRootPath());
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }
```

`public static void setFileLabel(java.io.File file, SensitivityLabel label)`

The `setFileLabel` method changes the sensitivity label of the specified file to the specified label. When the sensitivity label of a file changes, the file is moved to the zone that corresponds to the new label. The file is moved to a new path name that is relative to the root of the other zone.

For example, if you use the `setFileLabel` method to change the label of the file `/zone/internal/documents/designdoc.odt` from `INTERNAL` to `RESTRICTED`, the new path of the file will be `/zone/restricted/documents/designdoc.odt`. Note that if the destination directory does not exist, the file is not moved.

The following code fragment shows how you might change the label of the file:

```
SolarisLabel.setFileLabel(new File("/zone/internal/documents/designdoc.odt"),
SolarisLabel.getSensitivityLabel("RESTRICTED"));
```

When you change the sensitivity label of a file, the original file is deleted. The only exception occurs when the source and destination file systems are loopback-mounted from the same underlying file system. In this case, the file is renamed.

The Java virtual machine must be running with the appropriate privilege (`file_upgrade_sl` or `file_downgrade_sl`) to relabel a file.

For more information about setting privileges, see "Developing Privileged Applications," in *Solaris Security for Developers Guide*. See also the `setflabel`(3TSOL) man page.

Obtaining Label Range Objects

The `getLabelRange` static factory creates a label range object. The `getUserRange` and `getDeviceRange` static factories obtain label range objects for a user and a device, respectively. The `getUpper` and `getLower` methods are used to obtain the upper and lower labels of the range, respectively. In addition, the `inRange` method determines whether the specified label is in a range. For more information about the `inRange` method, see "Comparing Label Objects" on page 96.

The `getUserRange` and `getDeviceRange` static factories correspond to the `getuserrange()` and `getdevicerange()` routines. For more information, see "Obtaining Label Ranges" on page 31.

The following constructor and method descriptions include the prototype declaration for each constructor:

```
public static Range getDeviceRange(java.lang.String device)
```

The `getDeviceRange` static factory obtains the label range of a user-allocatable device. If no label range is specified for the device, the default range has an upper bound of `ADMIN_HIGH` and a lower bound of `ADMIN_LOW`.

You can use the `list_devices` command to show the label range for a device. See the `list_devices`(1) man page.

```
public static <L extends SolarisLabel,U extends SolarisLabel> Range getLabelRange(L
lower, U upper)
```

The `getLabelRange` static factory creates a label range. The static factory takes the lower bound value in the range and the upper bound, or clearance, as parameters. An exception is thrown if *upper* does not dominate *lower*.

```
public L getLower()
```

The `getLower` method returns the lower portion of the range.

```
public U getUpper()
```

The `getUpper` method returns the upper portion of the range.

```
public static Range getUserRange(java.lang.String user)
```

The `getUserRange` static factory obtains the label range of the specified user. The lower bound in the range is used as the initial workspace label when a user logs in to a multilevel desktop. The upper bound, or clearance, is used as an upper limit to the available labels that a user can assign to labeled workspaces.

The default value for a user's label range is specified in the `label_encodings` file. The value can be overridden by the `user_attr` file.

For more information, see the `getuserrange`(3TSOL) man page.

Accessing Labels in Zones

The following description includes the prototype declaration for the method:

```
public final java.lang.String toRootPath()
```

This method returns the root path name of the zone for the specified sensitivity label.

The following code excerpt shows how to obtain the root path for the `PUBLIC` sensitivity label:

```
SensitivityLabel sl = SolarisLabel.getSensitivityLabel("PUBLIC");
System.out.println("toRootPath: " + sl.toRootPath();
```

This method throws a `java.io.IOException` if an invalid label is specified or if no zone is configured for the specified label.

This method mimics the `getzonerootbylabel()` routine. See the `getzonerootbylabel`(3TSOL) man page. See also "Accessing Labels in Zones" on page 32.

Obtaining the Remote Host Type

The Java implementation of the Trusted Extensions label APIs does not include interfaces for obtaining the remote host type.

Translating Between Labels and Strings

The `SolarisLabel` abstract class includes methods for translating between labels and strings, which are inherited by its subclasses.

These methods translate the internal representation of a label (`m_label_t`) to `String` objects.

You can use the `toInternal` method to translate a label into a string that hides the classification name. This format is suitable for storing labels in public objects.

The running Java virtual machine must dominate the label to be translated, or it must have the `sys_trans_label` privilege. See the `label_to_str`(3TSOL) man page.

Some of the label values are based on data in the `label_encodings` file.

The following methods mimic the `label_to_str()` routine. See the `label_to_str`(3TSOL) man page.

`public final java.lang.String toColor()`

> This method returns the color of the `SolarisLabel` object. The value is suitable for use by HTML.

`public final java.lang.String toInternal()`

> This method returns the internal representation of the label that is safe for storing in a public object. An internal conversion can later be parsed to its same value. This is the same as using the `toString` method.
>
> These two methods differ in the way that they handle errors. If the `toInternal` method encounters an error, it returns a `java.io.IOException`. However, if the `toString` method encounters an error, it returns a null.

`public java.lang.String toString()`

> This method returns the internal hexadecimal version of the label in string form, which is the same as using the `toInternal` method.
>
> These two methods differ in the way that they handle errors. If the `toString` method encounters an error, it returns a null. However, if the `toInternal` method encounters an error, it returns a `java.io.IOException`.

`public java.lang.String toText()`

> This method returns a human-readable text string of the `SolarisLabel` object.

`public java.lang.String toTextLong()`

> This method returns the long human-readable text string of the `SolarisLabel` object.

`public java.lang.String toTextShort()`

This method returns the short human-readable text string of the `SolarisLabel` object.

The following methods perform label translations that are suitable for output. These methods mimic some of the functionality of the `label_to_str()` routine. See the `label_to_str`(3TSOL) and `m_label`(3TSOL) man pages.

`public java.lang.String toCaveats()`

This method returns a human-readable text string that is suitable for output.

This method is only available for `SensitivityLabel` objects, not for `ClearanceLabel` objects.

`public java.lang.String toChannels()`

This method returns a human-readable text string that is suitable for output.

This method is only available for `SensitivityLabel` objects, not for `ClearanceLabel` objects.

`public java.lang.String toFooter()`

This method returns a human-readable text string that is appropriate for use as the sensitivity label. This sensitivity label appears at the bottom of banner and trailer pages.

This method is only available for `SensitivityLabel` objects, not for `ClearanceLabel` objects.

`public java.lang.String toHeader()`

This method returns a human-readable text string that is appropriate for use as the sensitivity label. This sensitivity label appears at the top of banner and trailer pages.

This method is only available for `SensitivityLabel` objects, not for `ClearanceLabel` objects.

`public java.lang.String toProtectAs()`

This method returns a human-readable text string that is suitable for the downgrade section of the banner page.

This method is only available for `SensitivityLabel` objects, not for `ClearanceLabel` objects.

Methods such as `toText`, `toInternal`, and `toColor` do not translate from a string to a label. To translate a string to a sensitivity label or to a clearance label, you must call the `getSensitivityLabel` or `getClearanceLabel` static factories, respectively. The following static factories mimic the `str_to_label()` routine. See the `str_to_label`(3TSOL) and `m_label`(3TSOL) man pages.

`public static ClearanceLabel getClearanceLabel(java.lang.String label)`

This static factory creates a clearance label from the specified string. The following examples create new clearance labels based on a label name and the internal hexadecimal name of a label:

```
ClearanceLabel cl = SolarisLabel.getClearanceLabel("PUBLIC");
ClearanceLabel cl = SolarisLabel.getClearanceLabel("0x0002-08-08");
```

`public static SensitivityLabel getSensitivityLabel(java.lang.String label)`

This static factory creates a sensitivity label from the specified string. The following examples create new sensitivity labels based on a label name and the internal hexadecimal name of a label:

```
SensitivityLabel sl = SolarisLabel.getSensitivityLabel("PUBLIC");
SensitivityLabel sl = SolarisLabel.getSensitivityLabel("0x0002-08-08");
```

Comparing Label Objects

The following `equals`, `dominates`, and `strictlyDominates` methods are used to compare labels, and correspond to the `blequal()`, `bldominate()`, and `blstrictdom()` routines. The `inRange` method is used to determine whether a label is within a specified label range, and corresponds to the `blinrange()` routine. In these methods, a *label* refers to a classification and a set of compartments in a sensitivity label or in a clearance label. For more information, see "Comparing Labels" on page 35 and the `blcompare`(3TSOL) man page.

`public boolean dominates(SolarisLabel label)`

The `dominates` method compares two labels to determine whether one label dominates the other.

The following example code shows how you can make the comparison. The `CNF : INTERNAL` label is being compared to check its dominance over the `PUBLIC` label.

```
SensitivityLabel sl = SolarisLabel.getSensitivityLabel("CNF : INTERNAL");
boolean isDominant = sl.dominates(SolarisLabel.getSensitivityLabel("PUBLIC"));
```

`public boolean equals(java.lang.Object obj)`

The `equals` method compares two labels to determine whether they are equal.

The following example code shows how you can make the comparison:

```
SensitivityLabel sl = SolarisLabel.getSensitivityLabel("CNF : INTERNAL");
boolean isSame = sl.equals(SolarisLabel.getSensitivityLabel("PUBLIC"));
```

`public boolean Range inRange(SensitivityLabel label)`

The `inRange` method determines whether the specified label is within the range. This method belongs to the `Range` class.

The following code fragment shows how you can verify that a file is within a user's clearance range:

```
import solarismac.*;
import java.io.*;

public class Example1
{
    public static void main (String args[]) {

    try {
    Range range;

        range = Range.getUserRange("jweeks");
        SensitivityLabel fsl =
            SolarisLabel.getFileLabel(new File("/etc/passwd"));
        boolean isInRange;

        isInRange = Range.inRange(fsl);

        if (isInRange)
            System.out.println("File is within user's range");
        else
            System.out.println("File is not within user's range");

        }
        catch (Exception e) {
            e.PrintStackTrace();
        }
    }
}
```

`public boolean strictlyDominates(SolarisLabel label)`

The `strictlyDominates` method compares two labels to determine whether one label strictly dominates the other. When a label strictly dominates another, it dominates the other label, but is not equal to the other label.

The following example code shows how you can make the comparison. The `CNF : INTERNAL` label is being compared to check its strict dominance over the `PUBLIC` label.

```
SensitivityLabel sl = SolarisLabel.getSensitivityLabel("CNF : INTERNAL");
boolean isStrictlyDominant =
    sl.strictlyDominates(SolarisLabel.getSensitivityLabel("PUBLIC"));
```

For more information about label relationships, see "Label Relationships" on page 13.

The getMaximum and getMinimum methods are used to determine the least upper bound and the greatest lower bound of the specified label range, respectively. These methods mirror the functionality of the blmaximum() and blminimum() routines. For more information, see "Comparing Labels" on page 35 and the blminmax(3TSOL) man page.

For instance, use the getMaximum method to determine the label to use when creating a new labeled object that combines information from two other labeled objects. The label of the new object will dominate both of the original labeled objects. Each method is defined by the ClearanceLabel and SensitivityLabel subclasses.

```
public ClearanceLabel getMaximum(ClearanceLabel bounding)
```

The getMaximum method creates a new clearance label object that is the lowest label that can dominate two label objects you specify. The resulting object is the least upper bound of the range. getMaximum returns an object in the internal form of the clearance label.

```
public ClearanceLabel getMinimum(ClearanceLabel bounding)
```

The getMinimum method creates a new clearance label object that is the highest label that is dominated by two labels you specify. The resulting object is the greatest lower bound of the range. getMinimum returns an object in the internal form of the clearance label.

```
public SensitivityLabel getMaximum(SensitivityLabel bounding)
```

The getMaximum method creates a new sensitivity label object that is the lowest label that can dominate two label objects you specify. The resulting object is the least upper bound of the range. getMaximum returns an object in the internal form of the sensitivity label.

```
public SensitivityLabel getMinimum(SensitivityLabel bounding)
```

The getMinimum method creates a new sensitivity label object that is the highest label that is dominated by two labels you specify. The resulting object is the greatest lower bound of the range. getMinimum returns an object in the internal form of the sensitivity label.

The following table shows label input and output from the getMaximum and getMinimum methods.

TABLE 1 Using the getMinimum and getMaximum Methods

Input Labels	getMinimum Output	getMaximum Output
SECRET A B TOP SECRET A B SA SB CC	SECRET A B	TOP SECRET A B SA SB CC
SECRET A B	SECRET A	TOP SECRET A B SA CC

Input Labels	getMinimum Output	getMaximum Output
TOP SECRET A SA CC		
SECRET A B TOP SECRET	SECRET	TOP SECRET A B
SECRET A TOP SECRET B	SECRET	TOP SECRET A B

♦ ♦ ♦ APPENDIX A

Programmer's Reference

This appendix explains where to find information about developing, testing, and releasing label-aware applications to an environment that uses the Trusted Extensions software.

This appendix covers the following topics:

- "Header File Locations" on page 101
- "Abbreviations Used in Interface Names and Data Structure Names" on page 101
- "Developing, Testing, and Debugging an Application" on page 102

Header File Locations

Most Trusted Extensions header files are located in the `/usr/include/tsol` directory and in the `/usr/include/sys/tsol` directory. The locations of other header files are shown in the following table.

Header File and Its Location	Category of Interface
`/usr/include/X11/extensions/Xtsol.h`	X Window System
`/usr/include/libtsnet.h`	Trusted network library
`/usr/include/bsm/libbsm.h`	Audit library

Abbreviations Used in Interface Names and Data Structure Names

Many of the Trusted Extensions interface names and data structure names use the following short abbreviations. Knowing the abbreviations of these names will help you recognize the purpose of an interface or structure.

TABLE 2 Name Abbreviations Used by Trusted Extensions APIs

Abbreviation	Name
`attr`	Attribute
`b`	Binary
`clear`	Clearance
`ent`	Entry
`f`	File
`fs`	File system
`h`	Hexadecimal
`l`	Level, label, or symbolic link
`prop`	Properties
`r`	Re-entrant
`res`	Resource
`s`	String
`sec`	Security
`sl`	Sensitivity label
`tp`	Trusted Path
`tsol`	Trusted Extensions
`xtsol`	Trusted X11 Server

Developing, Testing, and Debugging an Application

You must develop, test, and debug an application on an isolated development system to prevent software bugs and incomplete code from compromising the security policy on the main system.

Follow these guidelines:

- Remove extra debugging code, especially code that provides undocumented features and code that bypasses security checks.
- Make application data manipulation easy to follow so that the manipulation can be inspected for security problems by an 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 must always terminate.
- Test all functionality by running the application at all sensitivity labels and from all roles that you expect will run the application.
  - If the program is run by an ordinary user and not by a role, start the program from the command line at the labels where the program is intended to run.

  - If the program is run by a role, start the program from the command line in the global zone or from the user role at the labels where the program is intended to run.
- Test all functionality under privilege debugging mode so that you know whether the application has all the privileges it needs. This type of testing also determines whether the application is attempting to perform privileged tasks that it should not be performing.
- Know the security implications of using privileges. Ensure that the application does not compromise system security by its use of privileges.
- Know and follow good privilege bracketing practices.

  See *Developer's Guide to Oracle Solaris 11.3 Security*.
- If you use the `SUNWspro debugger` or the `dbx` command to test a privileged application, start the debugger *before* you attach it to a running process. You cannot start the debugger with the command name as an argument.

♦ ♦ ♦ APPENDIX B

Trusted Extensions API Reference

This appendix provides application programming interface (API) listings and cross-references to their use. Declarations are grouped by security topic.

This appendix covers the following topics:

- “Process Security Attribute Flags APIs” on page 105
- “Label APIs” on page 105
- “Label-Clipping APIs” on page 107
- “RPC APIs” on page 107
- “Trusted X Window System APIs” on page 107
- “Oracle Solaris Library Routines and System Calls That Use Trusted Extensions Parameters” on page 108
- “System Calls and Library Routines in Trusted Extensions” on page 109

Process Security Attribute Flags APIs

The following Oracle Solaris APIs accept Trusted Extensions parameters:

- `uint_t getpflags(uint_t flag);`
- `int setpflags(uint_t flag, uint_t value);`

Label APIs

The label APIs are introduced in Chapter 2, “Labels and Clearances”. Sample code is provided in Chapter 3, “Label Code Examples”.

The following lists the types of label-related APIs and shows the prototype declarations of the routines and system calls for each type:

- **Enabling and disabling a Trusted Extensions system**
  - `int labeling_disable(uint_t flags);`
  - `int labeling_enable(uint_t flags);`
- **Specifying a** `label_encodings` **file**
  - `int labeling_set_encodings(const char *path);`
- **Accessing the** `label_encodings` **file**
  - `m_label_t *m_label_alloc(const m_label_type_t label_type);`
  - `int m_label_dup(m_label_t **dst, const m_label_t *src);`
  - `void m_label_free(m_label_t *label);`
  - `int label_to_str(const m_label_t *label, char **string, const m_label_str_t conversion_type, uint_t flags);`
- **Comparing level relationships**
  - `int blequal(const m_label_t *level1, const m_label_t *level2);`
  - `int bldominates(const m_label_t *level1, const m_label_t *level2);`
  - `int blstrictdom(const m_label_t *level1, const m_label_t *level2);`
  - `int blinrange(const m_label_t *level, const brange_t *range);`
  - `void blmaximum(m_label_t *maximum_label, const m_label_t *bounding_label);`
  - `void blminimum(m_label_t *minimum_label, const m_label_t *bounding_label);`
- **Accessing label ranges**
  - `m_range_t *getuserrange(const char *username);`
  - `blrange_t *getdevicerange(const char *device);`
- **Accessing labels in zones**
  - `char *getpathbylabel(const char *path, char *resolved_path, size_t bufsize, const m_label_t *sl);`
  - `m_label_t *getzonelabelbyid(zoneid_t zoneid);`
  - `m_label_t *getzonelabelbyname(const char *zonename);`
  - `zoneid_t *getzoneidbylabel(const m_label_t *label);`
  - `char *getzonerootbyid(zoneid_t zoneid);`
  - `char *getzonerootbylabel(const m_label_t *label);`
  - `char *getzonerootbyname(const char *zonename);`
- **Obtaining the remote host type**

  - `tsol_host_type_t tsol_getrhtype(char *hostname);`
- **Accessing and modifying sensitivity labels**
  - `int fgetlabel(int fd, m_label_t *label_p);`
  - `int getlabel(const char *path, m_label_t *label_p);`
  - `int setflabel(const char *path, const m_label_t *label_p);`
  - `int getplabel(m_label_t *label_p);`
  - `int label_to_str(const m_label_t *label, char **string, const m_label_str_t conversion_type, uint_t flags);`
  - `int str_to_label(const char *string, m_label_t **label, const m_label_type_t label_type, uint_t flags, int *error);`

Label-Clipping APIs

For information about this label-clipping API, see Chapter 5, “Trusted X Window System”.

```
int label_to_str(const m_label_t *label, char **string,
const m_label_str_t conversion_type, uint_t flags);
```

RPC APIs

Trusted Extensions does not provide interfaces for remote procedure calls (RPC). RPC interfaces have been modified to work with Trusted Extensions. For conceptual information, see Chapter 4, “Interprocess Communications”.

Trusted X Window System APIs

For information about the Trusted X Window System APIs, see Chapter 5, “Trusted X Window System”.

- `Status XTSOLgetResAttributes(Display *display, XID object, ResourceType type, XTSOLResAttributes *winattrp);`
- `Status XTSOLgetPropAttributes(Display *display, Window window, Atom property, XTSOLPropAttributes *propattrp);`

- `Status XTSOLgetClientAttributes(Display *display, XID windowid, XTsolClientAttributes *clientattrp);`
- `Status XTSOLgetResLabel(Display *display, XID object, ResourceType type, m_label_t *sl);`
- `Status XTSOLsetResLabel(Display *display, XID object, ResourceType type, m_label_t *sl);`
- `Status XTSOLgetResUID(Display *display, XID object, ResourceType type, uid_t *uidp);`
- `Status XTSOLsetResUID(Display *display, XID object, ResourceType type, uid_t *uidp);`
- `Status XTSOLgetPropLabel(Display *display, Window window, Atom property, m_label_t *sl);`
- `Status XTSOLsetPropLabel(Display *display, Window window, Atom property, m_label_t *sl);`
- `Status XTSOLgetPropUID(Display *display, Window window, Atom property, uid_t *uidp);`
- `Status XTSOLsetPropUID(Display *display, Window window, Atom property, uid_t *uidp);`
- `Status XTSOLgetWorkstationOwner(Display *display, uid_t *uidp);`
- `Status XTSOLsetWorkstationOwner(Display *display, uid_t *uidp);`
- `Status XTSOLsetSessionHI(Display *display, m_label_t *sl);`
- `Status XTSOLsetSessionLO(Display *display, m_label_t *sl);`
- `Status XTSOLMakeTPWindow(Display *display, Window *w);`
- `Bool XTSOLIsWindowTrusted(Display *display, Window *window);`
- `Status XTSOLgetSSHeight(Display *display, int screen_num, int *newheight);`
- `Status XTSOLsetSSHeight(Display *display, int screen_num, int newheight);`
- `Status XTSOLsetPolyInstInfo(Display *display, m_label_t sl, uid_t *uidp, int enabled);`

Oracle Solaris Library Routines and System Calls That Use Trusted Extensions Parameters

The following Oracle Solaris interfaces either include Trusted Extensions parameters or are used in this guide with Trusted Extensions interfaces:

- `int auditon(int cmd, caddr_t data, int length);`

- `void free(void *ptr);`
- `int getpeerucred(int fd, ucred_t **ucred);`
- `uint_t getpflags(uint_t flag);`
- `int is_system_labeled(void);`
- `int setpflags(uint_t flag, uint_t value);`
- `int getsockopt(int s, int level, int optname, void *optval, int *optlen);`
- `int setsockopt(int s, int level, int optname, const void *optval, int optlen);`
- `int socket(int domain, int type, int protocol);`
- `ucred_t *ucred_get(pid_t pid);`
- `m_label_t *ucred_getlabel(const ucred_t *uc);`

System Calls and Library Routines in Trusted Extensions

The following table lists the Trusted Extensions system calls and routines. The table also provides references to descriptions and declarations of the interface and to examples of the interface that appear in this guide. The man page section is included as part of the name of each system call and routine.

TABLE 3 System Calls and Library Routines That Are Used in Trusted Extensions

System Call or Library Routine	Cross-Reference to Description	Cross-Reference to Example
`bldominates`(3TSOL)	“Label Relationships” on page 13 “Comparing Labels” on page 35	“Determining the Relationship Between Two Labels” on page 42
`blequal`(3TSOL)	“Comparing Labels” on page 35	“Determining the Relationship Between Two Labels” on page 42
`blmaximum`(3TSOL)	“Comparing Labels” on page 35	
`blminimum`(3TSOL)	“Comparing Labels” on page 35	
`blstrictdom`(3TSOL)	“Comparing Labels” on page 35	
`fgetlabel`(2)	“Labeled Zones” on page 24 “Obtaining and Setting the Label of a File” on page 30	
`free`(3C)	“Translating Between Labels and Strings” on page 34	
`getlabel`(2)	“Labeled Zones” on page 24 “Obtaining and Setting the Label of a File” on page 30	“Obtaining a File Label” on page 40

System Call or Library Routine	Cross-Reference to Description	Cross-Reference to Example
`getpathbylabel`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getpflags`(2)	"MAC-Exempt Sockets" on page 22	
`getplabel`(3TSOL)	"Accessing the Process Sensitivity Label" on page 29	"Translating the Window Label With the Font List" on page 65
`getuserrange`(3TSOL)	"Obtaining Label Ranges" on page 31	
`getzoneidbylabel`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getzonelabelbyid`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getzonelabelbyname`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getzonerootbyid`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getzonerootbylabel`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getzonerootbyname`(3TSOL)	"Accessing Labels in Zones" on page 32	
`label_to_str`(3TSOL)	"Translating Between Labels and Strings" on page 34	"Obtaining a Process Label" on page 39
`m_label_alloc`(3TSOL)	"Allocating and Freeing Memory for Labels" on page 29	"Obtaining a Process Label" on page 39 "Obtaining a File Label" on page 40
`m_label_dup`(3TSOL)	"Allocating and Freeing Memory for Labels" on page 29	
`m_label_free`(3TSOL)	"Allocating and Freeing Memory for Labels" on page 29	"Obtaining a Process Label" on page 39
`setflabel`(3TSOL)	"Obtaining and Setting the Label of a File" on page 30 "Obtaining and Setting the Label of a File" on page 30	
`setpflags`(2)	"MAC-Exempt Sockets" on page 22	
`str_to_label`(3TSOL)	"Translating Between Labels and Strings" on page 34	"Obtaining a File Label" on page 40
`tsol_getrhtype`(3TSOL)	"Obtaining the Remote Host Type" on page 33	
`ucred_get`(3C)	"Multilevel Ports" on page 21	
`ucred_getlabel`(3C)	"Multilevel Ports" on page 21	
`XTSOLgetClientAttributes`(3XTSOL)	"Accessing Attributes" on page 60	
`XTSOLgetPropAttributes`(3XTSOL)	"Accessing Attributes" on page 60	
`XTSOLgetPropLabel`(3XTSOL)	"Accessing and Setting a Window Property Label" on page 61	
`XTSOLgetPropUID`(3XTSOL)	"Accessing and Setting a Window Property Label" on page 61	
`XTSOLgetResAttributes`(3XTSOL)	"Obtaining Window Attributes" on page 65	
`XTSOLgetResLabel`(3XTSOL)	"Obtaining a Window Label" on page 66	
`XTSOLgetResUID`(3XTSOL)	"Obtaining the Window User ID" on page 67	

System Call or Library Routine	Cross-Reference to Description	Cross-Reference to Example
	"Accessing and Setting a Window User ID" on page 61	
XTSOLgetSSHeight(3XTSOL)	"Accessing and Setting the Screen Stripe Height" on page 63	
XTSOLgetWorkstationOwner(3XTSOL)	"Accessing and Setting a Workstation Owner ID" on page 62	
XTSOLIsWindowTrusted(3XTSOL)	"Working With the Trusted Path Window" on page 63	
XTSOLMakeTPWindow(3XTSOL)	"Working With the Trusted Path Window" on page 63	
XTSOLsetPolyInstInfo(3XTSOL)	Chapter 5, "Trusted X Window System"	
XTSOLsetPropLabel(3XTSOL)	"Accessing and Setting a Window Property Label" on page 61	
XTSOLsetPropUID(3XTSOL)	"Accessing and Setting a Window Property Label" on page 61	
XTSOLsetResLabel(3XTSOL)	"Setting a Window Label" on page 67	
XTSOLsetResUID(3XTSOL)	"Accessing and Setting a Window User ID" on page 61	
XTSOLsetSessionHI(3XTSOL)	"Setting the X Window Server Clearance and Minimum Label" on page 62	
XTSOLsetSessionLO(3XTSOL)	"Setting the X Window Server Clearance and Minimum Label" on page 62	
XTSOLsetSSHeight(3XTSOL)	"Accessing and Setting the Screen Stripe Height" on page 63	
XTSOLsetWorkstationOwner(3XTSOL)	"Accessing and Setting a Workstation Owner ID" on page 62	

Index