Trusted Extensions Developer's Guide

Part No: E61032
November 2020

Accessibilité de la documentation

Pour plus d'informations sur l'engagement d'Oracle pour l'accessibilité de la documentation, visitez le site Web Oracle Accessibility Program, à l'adresse : `http://www.oracle.com/pls/topic/lookup?ctx=acc&id=docacc`.

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 servers that run the Trusted Extensions feature of Oracle Solaris 11.4.
- **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=E37838-01`.

Feedback

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

♦ ♦ ♦ C H A P T E R 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 Configuration and Administration*.

This chapter covers the following topics:

- “Understanding Labels” on page 11
- “Trusted Extensions APIs” on page 15
- “Trusted Extensions Security Policy” on page 18

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 18.

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.

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(5)` 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 Label 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(5)` 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 during PAM authentication by the `pam_tsol_account.so.1` module
- All multilevel ports on hosts that permit communications
- All multilevel interfaces for zones and networks
- Allocatable devices, such as CD-ROM devices and audio devices, when allocated from the global zone
- Other devices that are not allocatable, such as printers and workstations, when allocated from the global zone

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 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.

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`(5) 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 19. 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.
- 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(5)` 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 11.4 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 the `tninfo` command in a shell script. 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 Trusted Extensions.
#  1 otherwise.
#
smf_is_system_labeled() {
    [ ! -x /bin/tninfo ] && return 1
    /bin/tninfo > /dev/null 2>&1
    return $?
}
```

On a system that is not running Trusted Extensions, the exit status is `1` and the output is `Trusted Extensions must be enabled`. On a Trusted Extensions system, the script returns an exit status of `0` with no error message. For more information, see the `tninfo`(8) man page.

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 `setlabel` command permits 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 label of the zone where the user is being authenticated must be within the user's label range. For the global zone, the user's label range must be `ADMIN_LOW` to `ADMIN_HIGH`. The minimum and maximum labels are applicable when an authorized user uses the `setlabel` command to change the label of a file.

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`(5), and `user_attr`(5) 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`(5) 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`(8) 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".
```

♦ ♦ ♦ C H A P T E R 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

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 21.

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);
        }
}
```

♦ ♦ ♦ 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 53
- "Abbreviations Used in Interface Names and Data Structure Names" on page 53
- "Developing, Testing, and Debugging an Application" on page 54

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/libtsnet.h`	Trusted network 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 1 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

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 11.4 Security*.
- If you use a debugger such as 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 57
- "Label APIs" on page 57
- "RPC APIs" on page 59
- "Oracle Solaris Library Routines and System Calls That Use Trusted Extensions Parameters" on page 59
- "System Calls and Library Routines in Trusted Extensions" on page 60

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);`

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".

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:

- `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 2 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 23 "Obtaining and Setting the Label of a File" on page 30	
`free`(3C)	"Obtaining a Process Label" on page 39	
`getlabel`(2)	"Labeled Zones" on page 23 "Obtaining and Setting the Label of a File" on page 30	"Obtaining a File Label" on page 40
`getpathbylabel`(3TSOL)	"Accessing Labels in Zones" on page 32	
`getpflags`(2)	"MAC-Exempt Sockets" on page 21	
`getplabel`(3TSOL)	"Accessing the Process Sensitivity Label" on page 29	
`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)	"Obtaining a Process Label" on page 39	"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

System Call or Library Routine	Cross-Reference to Description	Cross-Reference to Example
		“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 21	
`str_to_label`(3TSOL)	“Obtaining a Process Label” on page 39	“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 20	
`ucred_getlabel`(3C)	“Multilevel Ports” on page 20	

Index