Configuring the SELinux Policy

Stephen Smalley
NSA

sds@epoch.ncsc.mil

This work supported by NSA contract MDA904-01-C-0926 (SELinux)
Initial: February 2002, Last revised: Feb 2005
NAI Labs Report #02-007

Table of Contents

1. Introduction 2

2. Architectural Concepts and Definitions 2

2.1. Flask Concepts 2

2.2. Flask Definitions 4

3. Security Model 4

3.1. TE Model 4

3.2. RBAC Model 5

3.3. User Identity Model 6

4. Policy Language and the Example Policy Configuration 6

4.1. TE and RBAC Configuration Overview 7

4.2. TE Statements 8

4.3. RBAC Statements 16

4.4. User Declarations 19

4.5. Constraint Definitions 20

4.6. Security Context Specifications 21

4.7. File Contexts Configuration 24

5. Building and Applying the Policy 25

5.1. Compiling and Loading the Policy 25

5.2. Applying the File Contexts Configuration 26

6. Configuration Files for Security-Aware Applications 26

6.1. Default_Contexts 26

6.2. Default_Type 27

6.3. Initrc_Context 27

7. Customizing the Policy 27

7.1. Adding Users 27

7.2. Adding Permissions 28

7.3. Adding Programs to an Existing Domain 29

7.4. Creating a New Domain 29

7.5. Creating a New Type 31

7.6. Creating a New Role 32

References **32**

1. Introduction

NSA Security-Enhanced Linux (SELinux) is an implementation of a flexible and fine-grained mandatory access control (MAC) architecture called Flask in the Linux kernel[LoscoccoFreenix2001]. SELinux can enforce an administratively-defined security policy over all processes and objects in the system, basing decisions on labels containing a variety of security-relevant information. The architecture provides flexibility by cleanly separating the policy decision-making logic from the policy enforcement logic. The policy decision-making logic is encapsulated within a single component known as the security server with a general security interface. A wide range of security models can be implemented as security servers without requiring any changes to any other component of the system.

To demonstrate the architecture, SELinux provides an example security server that implements a combination of Type Enforcement (TE)[BoebertNCSC1985], Role-Based Access Control (RBAC)[FerraioloNCSC1992], and optionally Multi-Level Security (MLS). These security models provide significant flexibility through a set of policy configuration files. An example security policy configuration was developed to demonstrate how SELinux can be used to meet certain security goals and to provide a starting point for users [SmalleyNAITR2001][LoscoccoOLS2001].

This technical report describes how to configure the SELinux security policy for the example security server. Section 2 explains concepts defined by the Flask architecture that are important to configuring the policy. Section 3 describes the security model implemented by the example security server. The policy language and the example policy configuration are described in Section 4. Section 5 explains how the policy is built and applied to the system. Configuration files for security-aware applications are discussed in Section 6. Section 7 describes how to customize the policy for various purposes.

2. Architectural Concepts and Definitions

The Flask operating system security architecture provides flexible support for mandatory access control (MAC) policies[SpencerUsenixSec1999]. The SELinux implementation of the Flask architecture is described in [LoscoccoFreenix2001]. This section discusses concepts defined by the Flask architecture that are important to configuring the SELinux policy. It then discusses definitions specified by the Flask architecture that are used by both the policy enforcing code and by the policy configuration.

2.1. Flask Concepts

Every subject (process) and object (e.g. file, socket, IPC object, etc) in the system is assigned a collection of security attributes, known as a security context. A security context contains all of the security attributes associated with a particular subject or object that are relevant to the security policy. The content and format of a security context depends on the particular security model, so a security context is only interpreted by the security server. In order to better encapsulate security contexts and to provide greater efficiency, the policy enforcement code of SELinux typically handles security identifiers (SIDs) rather than security contexts. A SID is an integer that is mapped by the security server to a security context at runtime. SIDs are nonpersistent and local identifiers, and must be translated to security

contexts for labeling persistent objects such as files or for labeled networking. A small set of SID values are reserved for system initialization or predefined objects; these SID values are referred to as initial SIDs. Kernel SIDs are not exported to userspace; the kernel only returns security contexts to userspace. However, userspace policy enforcers may have their own SID mappings maintained by the userspace AVC that is included in libselinux.

When a security decision is required, the policy enforcement code passes a pair of SIDs (typically the SID of a subject and the SID of an object, but sometimes a pair of subject SIDs or a pair of object SIDs), and an object security class to the security server. The object security class indicates the kind of object, e.g. a process, a regular file, a directory, a TCP socket, etc. The security server looks up the security contexts for the SIDs. It then bases its decision on the attributes within the security contexts and the object security class. The security server provides two major kinds of security decisions to the policy enforcement code: labeling decisions and access decisions.

Labeling decisions, also referred to as transition decisions, specify the default security attributes to use for a new subject or a new object. A process transition decision is requested when a program is executed based on the current SID of the process and the SID of the program. An object transition decision is requested when a new object is created based on the SID of the creating process and the SID of a related object, e.g. the parent directory for file creations. These default labeling decisions can be overridden by security-aware applications using the SELinux API. In either case, the use of the new label must be approved by an access decision or the operation will fail.

The policy enforcement code is responsible for binding the labels to subjects and objects in the system. For transient objects such as processes and sockets, a SID can be associated with the corresponding kernel object. However, persistent files and directories require additional support to provide persistent labeling. Security contexts are stored persistently for files and directories using extended attributes, specifically the security.selinux attribute.

When converting an existing system to using SELinux, the extended attribute values for all files are typically initialized by a utility called **setfiles** from a file contexts configuration that specifies security contexts for files based on pathname regular expressions. If the Linux distribution already includes SELinux support, then the extended attributes will typically be set at install time by the corresponding package manager for the distribution, which may also be enhanced to use the file contexts configuration to determine the appropriate security context. Subsequently, the mapping is maintained dynamically by the policy enforcing code to reflect create, delete, and relabel operations.

The file contexts configuration is logically separate from the policy configuration. The file contexts configuration is not part of the kernel policy and is only used by userspace programs when initializing or resetting the extended attributes to their install-time values. The extended attribute values are then used at runtime by the policy enforcing code. The policy configuration is used by the security server and is needed to obtain security decisions. The policy configuration specifies security decisions based entirely on security attributes, whereas the file contexts configuration specifies security contexts for files based on pathnames. Using the pathname is only suitable for initial installation of the file; subsequently, the security attributes of the file should be tracked during runtime based on policy.

Access decisions specify whether or not a permission is granted for a given pair of SIDs and class. Each object class has a set of associated permissions defined to control operations on objects with that class. These permission sets are represented by a bitmap called an access vector. When an access decision is requested, the security server returns an allowed access vector containing the decisions for all of the permissions defined for the object class. These access vectors are cached by a component of the Flask architecture called the Access Vector Cache (AVC) for subsequent use by the policy enforcement code.

The AVC component provides an interface to the security server to support cache management for policy changes.

In addition to providing an allowed access vector, the security server provides two access vectors related to auditing. An auditallow decision indicates whether a permission check should be audited when it is granted. For example, if a permission is associated with a highly sensitive operation, it may be desirable to audit every use of that operation. An auditdeny decision indicates whether a permission check should be audited when it is denied.

2.2. Flask Definitions

A small set of configuration files are shared between the SELinux kernel module and the example policy configuration. These files define the Flask security classes, initial SIDs, and access vector permissions. This information is not specific to any particular security model, and should rarely change. Changes to these files require recompilation of the module and policy, and will typically require updates to the module and policy to use the new values. These files do not need to be modified to configure the security model implemented by the example security server.

The meaning of the security classes and access vector permissions in the original SELinux implementation was described in [LoscoccoNSATR2001] and [LoscoccoFreenix2001]. Changes made for the LSM-based SELinux implementation are described in [SmalleyModuleTR2001].

The source for these files is located in the `policy/flask` subdirectory of the policy sources. These files are installed into `/etc/selinux/(targeted|strict)/src/policy/flask` and are used when the policy configuration is compiled. The Flask configuration files are listed in Table 1.

Table 1. Architecture Configuration Files

Filename	Description
security_classes	Declares the security classes.
initial_sids	Declares initial SIDs.
access_vectors	Defines the access vector permissions for each class.

3. Security Model

The example security server implements a security model that is a combination of a Type Enforcement (TE) model, a Role-Based Access Control (RBAC) model, and optionally a Multi-Level Security (MLS) model. The TE model provides fine-grained control over processes and objects in the system, and the RBAC model provides a higher level of abstraction to simplify user management. The optional MLS model is not (yet) documented in this report. This section describes the TE and RBAC models, and then discusses the concept of user identity in SELinux.

3.1. TE Model

A traditional TE model binds a security attribute called a domain to each process, and it binds a security attribute called a type to each object. The traditional TE model treats all processes in the same domain identically and it treats all objects that have the same type identically. Hence, domains and types can be viewed as security equivalence classes. A pair of access matrices specify how domains can access types and how domains can interact with other domains. Each user is authorized to operate in certain domains.

A TE model supports strong controls over program execution and domain transitions. A program, like any other object, is assigned a type, and the TE access matrix specifies what types can be executed by each domain. Furthermore, the TE access matrix specifies what types can be executed to initially enter a domain. Hence, a domain can be associated with a particular entrypoint program and optionally with particular helper programs and/or shared libraries. This characteristic of TE is useful in associating permissions with a particular set of code based on its function and trustworthiness and in protecting against the execution of malicious code.

The SELinux TE model differs from the traditional TE model in that it uses a single type attribute in the security context for both processes and objects. A domain is simply a type that can be associated with a process. A single type can be used both as the domain of a process and as the type of a related object, e.g. the /proc/PID entries for a process. A single access matrix specifies how types can access or interact with other types in terms of the permissions defined by the Flask architecture. Although the example TE configuration often uses the term domain when referring to the type of a process, the SELinux TE model does not internally distinguish domains from types.

The SELinux TE model also differs from the traditional TE model in that it uses the security class information provided by the Flask architecture. A SELinux TE transition or access decision is based on a type pair and on the security class. Hence, the policy can treat objects that have the same type but different security classes differently. For example, the policy can distinguish a TCP socket created by a domain from a raw IP socket created by the same domain.

A third difference between the SELinux TE model and the traditional TE model is that the SELinux TE model does not directly associate users with domains. Instead, SELinux uses the RBAC model to provide an additional layer of abstraction between users and domains. This approach is discussed further in the next subsection.

A TE transition rule for a process specifies the new domain based on the current domain of the process and the type of the program. A TE transition rule for an object specifies the new type based on the domain of the creating process, the type of the related object, and the object security class. If no matching rule is found in the TE configuration, then the SELinux TE model provides a default behavior appropriate for the class of object. For a process, the domain of the process is left unchanged across the program execution. For an object, the type of the related object (e.g. the parent directory for files) is used for the new object.

A TE access vector rule specifies an access vector based on the type pair and object security class. Rules can be specified for each kind of access vector, including the allowed, auditallow, and auditdeny vectors. These access vector rules define the TE access matrix. If no matching rule is found in the TE configuration, then the SELinux TE model defines a default behavior for each kind of access vector. Permissions are denied unless there is an explicit allow rule. No permissions are audited when granted unless there is an explicit auditallow rule. Permissions are always audited when denied unless there is an explicit dontaudit rule.

3.2. RBAC Model

A traditional RBAC model authorizes users to act in certain roles, and assigns a set of permissions to each role. The SELinux RBAC model authorizes each user for a set of roles, and authorizes each role for a set of TE domains. A role dominance relationship can optionally be specified in the RBAC configuration to define a hierarchy among roles. The assignment of permissions is primarily deferred to the TE configuration. This approach combines the ease of management provided by the RBAC model with the fine-grained protections provided by the TE model.

The SELinux RBAC model maintains a role attribute in the security context of each process. For objects, the role attribute is typically set to a generic `object_r` role and is unused. Role transitions for processes are controlled through a combination of the RBAC and TE models. The RBAC configuration specifies authorized transitions between roles based on the pair of roles. However, it is also desirable to limit role transitions to certain programs to ensure that malicious code cannot cause such transitions. Hence, role transitions are typically limited to certain TE domains in the policy configuration.

3.3. User Identity Model

The Linux user identity attributes are unsuitable for use by SELinux. Linux uids are often changed simply to express a change in permissions or privileges as opposed to a change in the actual user, posing problems for user accountability. Linux uids can be changed at any time via the set*uid calls, providing no control over the inheritance of state or the initialization of the process in the new identity. Linux uids can be arbitrarily changed by superuser processes.

Rather than imposing new restrictions on the Linux user identity attributes, SELinux maintains a user identity attribute in the security context that is independent of the Linux user identity attributes. By using a separate user identity attribute, the SELinux mandatory access controls remain completely orthogonal to the existing Linux access controls. SELinux can enforce rigorous controls over changes to its user identity attribute without affecting compatibility with Linux uid semantics.

The policy configuration limits the ability to change the SELinux user identity attribute to certain TE domains. These domains are associated with certain programs, such as login, crond and sshd, that have been modified to call new library functions to set the SELinux user identity appropriately. Hence, user login sessions and cron jobs are initially associated with the appropriate SELinux user identity, but subsequent changes in the Linux uid may not be reflected in the SELinux user identity. In some cases, this is desirable in order to provide user accountability or to prevent security violations. In some distributions that have integrated SELinux support, the su program also changes the SELinux user identity to the new user identity, while other distributions leave the SELinux user identity unchanged by su.

Since the SELinux user identity is independent of the Linux uid, it is possible to maintain separate user identity spaces for SELinux and Linux, with an appropriate mapping performed by the programs that set the SELinux user identity. For example, rather than maintaining a separate entry for each Linux user in the SELinux policy, it may be desirable to map most Linux users to a single SELinux user that is unprivileged. This approach is suitable when it is not necessary to separate these users via the SELinux policy.

4. Policy Language and the Example Policy Configuration

The policy configuration is specified using a simple declarative language, originally documented informally in [LoscoccoNSATR2001]. An example security policy configuration was written in this language to demonstrate how SELinux can be used to meet certain security goals and to provide a starting point for users. The example policy configuration was originally documented in [SmalleyNAITR2001] and was also discussed in [LoscoccoOLS2001]. This section describes the policy language, using the example configuration when possible to illustrate the language constructs. This section also describes the logically separate file contexts configuration. This section does not (yet) describe the optional MLS configuration, nor does it yet describe the conditional policy statement support.

The complete grammar for the policy language is specified in the `policy_parse.y` yacc file in the source package for the **checkpolicy** program. This section includes excerpts from the grammar, edited in some cases for readability or to omit obsolete elements. The policy configuration source files are located in the `/etc/selinux/(targeted|strict)/src/policy` directory. Since **m4** macros are extensively defined and used within the example policy configuration to ease specification, this section also describes some of the commonly used macros. The macro definitions can be found in the `policy/macros/global_macros.te` or `policy/macros/core_macros.te` files unless otherwise noted.

A policy configuration consists of the following top-level components: Flask definitions, TE and RBAC declarations and rules, user declarations, constraint definitions, and security context specifications. The Flask definitions were discussed in Section 2.2. The TE and RBAC declarations and rules specify the policy logic for the TE and RBAC models. The user declarations define the users for the user identity model and authorize each user for particular roles. The constraint definitions specify additional restrictions on permissions that can be based on a combination of information from the user identity, TE, and RBAC models. The security context specifications provide security contexts for certain entities, such as initial SIDs, unlabeled filesystems, and network objects. The top-level production for a policy is:

```
policy -> flask te_rbac users opt_constraints contexts
```

4.1. TE and RBAC Configuration Overview

The TE and RBAC configuration declares the roles, domains, and types and defines the labeling and access vector rules for the TE and RBAC models. The policy language permits the TE and RBAC configuration to be intermingled freely. The syntax of the TE and RBAC declarations and rules is shown below:

```
te_rbac -> te_rbac_statement | te_rbac te_rbac_statement
te_rbac_statement -> te_statement | rbac_statement
te_statement -> attrib_decl |
                type_decl |
                type_transition_rule |
                type_change_rule |
                te_av_rule |
                te_assertion
rbac_statement -> role_decl |
```

```
role_dominance |
role_allow_rule
```

A TE statement can be an attribute declaration, a type declaration, a type transition rule, a type change rule, an access vector rule, or an assertion. A RBAC statement can be a role declaration, a role dominance definition, or a role allow rule. The TE statements are described in Section 4.2, and the RBAC statements are described in Section 4.3.

4.2. TE Statements

4.2.1. Attribute Declarations

A type attribute is a name that can be used to identify a set of types with a similar property. Each type can have any number of attributes, and each attribute can be associated with any number of types. Attributes are associated with types in type declarations, described in the next section. Prior to the first use of an attribute in a type declaration, the attribute must be explicitly declared. The syntax for an attribute declaration is as follows:

```
attrib_decl -> ATTRIBUTE identifier ';'
```

Several examples of attribute declarations are shown below:

```
attribute domain;
attribute privuser;
attribute privrole;
```

An attribute name can be used throughout the policy configuration to express the set of the types that are associated with that attribute. Attribute names exist within the same name space as types. However, an attribute name cannot be used in the type field of a security context. Attributes have no implicit meaning to SELinux. The meaning of all other attributes are completely defined through their usage within the configuration, but should be documented as comments preceding the attribute declaration.

4.2.2. Type Declarations

The TE configuration language requires that every type be declared. However, forward references to types are accepted, since the policy compiler performs two passes. Each type declaration specifies a primary name for the type, an optional set of aliases for the type, and an optional set of attributes for the type. The syntax for a type declaration is as follows:

```
type_decl -> TYPE identifier opt_alias_def opt_attr_list ';'
opt_alias_def -> ALIAS aliases | empty
aliases -> identifier | '{' identifier_list '}'
identifier_list -> identifier | identifier_list identifier
opt_attr_list -> ',' attr_list | empty
attr_list -> identifier | attr_list ',' identifier
```

The primary name and any of the alias names can be used interchangeably within the TE configuration. During runtime, the example security server always uses the primary name to identify the type when it returns the security context for a SID. An application can use the primary name or any of the alias names to identify the type when requesting a SID for a context. Aliases can be used to provide shorthand forms or synonyms for a type, but have no significance from a security perspective. Primary names and alias names exist in a single type name space and must all be unique.

It is also possible to separately declare additional aliases and/or attributes for a type using the **typealias** and **typeattribute** statements. These statements will add aliases and attributes to already declared types. The syntax for these statements is as follows:

```
typealias_decl -> TYPEALIAS identifier ALIAS aliases ';'
typeattribute_decl -> TYPEATTRIBUTE identifier attr_list ';'
```

Several type declarations from the example policy configuration related to the secure shell daemon are shown below:

```
type sshd_t, domain, privuser, privrole, privlog, privowner;
type sshd_exec_t, file_type, exec_type, sysadmfile;
type sshd_tmp_t, file_type, sysadmfile, tmpfile;
type sshd_var_run_t, file_type, sysadmfile, pidfile;
```

The `sshd_t` type is the domain of the daemon process. The `sshd_exec_t` type is the type of the `sshd` executable. The `sshd_tmp_t` and `sshd_var_run_t` types are the types for temporary files and PID files, respectively, that are created by the daemon process. Each of these types has a set of associated attributes that are used in rules within the TE configuration.

4.2.3. TE Transition Rules

As described in Section 3.1, TE transition rules specify the new domain for a process or the new type for an object. In either case, the new type is based on a pair of types and a class. For a process, the first type, referred to as the source type, is the current domain and the second type, referred to as the target type, is the type of the executable. For an object, the source type is the domain of the creating process and the target type is the type of a related object, e.g. the parent directory for files. A TE transition rule for a process or an object uses the following syntax:

```
type_transition_rule -> TYPE_TRANSITION source_types target_types ':' classes new_type ';'
source_types -> set
target_types -> set
classes -> set
new_type -> identifier
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{' identifier_lis
```

The syntax permits concise specification of multiple TE transition rules through the optional use of sets for the source type, target type, or security class fields. Type attribute names can also be used to specify the source or target type fields to represent all types with that attribute. The tilde (~) character can be

used to indicate the complement of a set. The asterisk (*) character can be used to represent all types or classes. If multiple rules are specified for a given type pair and class, then warnings are issued by the policy compiler and the last such rule is used.

Several TE transition rules from the example policy configuration related to the secure shell daemon are shown below:

```
type_transition initrc_t sshd_exec_t:process sshd_t;
type_transition sshd_t tmp_t:{ dir file lnk_file
                               sock_file fifo_file } sshd_tmp_t;
type_transition sshd_t shell_exec_t:process user_t;
```

The `initrc_t` type is the domain entered when the `init` process runs the `/etc/rc.d` scripts. The first rule specifies that this domain should transition to the `sshd_t` domain when it executes a program with the `sshd_exec_t` type. The `tmp_t` type is the type of the `/tmp` directory. The second rule specifies that when the `sshd_t` domain creates a file in a directory with this type, the new subdirectory or file should be labeled with the `sshd_tmp_t` type. The `shell_exec_t` type is the type of shell programs. The last rule specifies that the `sshd_t` domain should transition to the `user_t` domain when it executes a program with the type `shell_exec_t`.

It is often desirable to specify a single TE transition rule that specifies the new type for a set of related classes, e.g. all file classes. Hence, a set of macros are defined for related classes, as shown in Table 2. Any one of these macros can be used in the class field of a TE transition rule, as shown in the following:

```
type_transition sshd_t tmp_t:notdevfile_class_set sshd_tmp_t;
type_transition cardmgr_t tmp_t:devfile_class_set cardmgr_dev_t;
```

Table 2. Class Macros

Macro Name	Description
dir_file_class_set	All directory and file classes.
file_class_set	All file classes (excludes dir).
notdevfile_class_set	Non-device file classes.
devfile_class_set	Device file classes.
socket_class_set	All socket classes.
dgram_socket_class_set	Datagram socket classes.
stream_socket_class_set	Stream socket classes.
unpriv_socket_class_set	Unprivileged socket classes (excludes rawip, netlink, packet, key).

Since each TE transition rule requires a set of corresponding TE access vector rules to authorize the operation, macros are provided that expand to common combinations of TE labeling rules and TE access vector rules. These macros are typically used instead of directly specifying TE transition rules. The `domain_auto_trans` macro is defined for domain transitions, and the `file_type_auto_trans` macro is defined for file type transitions. The `domain_auto_trans` macro takes the current domain, the program type, and the new domain as its parameters. The `file_type_auto_trans` macro takes the creating domain, the parent directory type, the new file type, and optionally the set of file classes

(defaulting to non-device file classes) as its parameters. The initial set of example transition rules shown earlier in this section are specified indirectly in the example policy configuration by using these macros, as shown below:

```
domain_auto_trans(initrc_t, sshd_exec_t, sshd_t)
file_type_auto_trans(sshd_t, tmp_t, sshd_tmp_t)
domain_auto_trans(sshd_t, shell_exec_t, user_t)
```

4.2.4. TE Change Rules

In addition to supporting TE transition rules, the TE configuration language also permits specification of TE change rules. These rules are not used by the kernel, but can be obtained and used by security-aware applications using the `security_compute_relabel` function. A TE change rule specifies the new type to use for a relabeling operation based on the domain of a user process, the current type of the object, and the class of the object. In the example policy configuration, these rules are used to specify the types to use when system daemons relabel terminal devices for user sessions. The syntax is the same as a TE transition rule except for the use of the `type_change` keyword.

Several examples of TE change rules are shown below:

```
type_change user_t tty_device_t:chr_file user_tty_device_t;
type_change sysadm_t tty_device_t:chr_file sysadm_tty_device_t;
type_change user_t sshd_devpts_t:chr_file user_devpts_t;
type_change sysadm_t sshd_devpts_t:chr_file sysadm_devpts_t;
```

The first pair of rules specify the types for user and administrator terminals for ordinary terminal devices. The `login` process obtains and uses the decisions specified by these rules when creating local user sessions. The second pair of rules specify the types for user and administrator terminals for pseudo-terminal devices that were initially allocated by the `sshd` daemon.

4.2.5. TE Access Vector Rules

A TE access vector rule specifies a set of permissions based on a type pair and an object security class. These rules define the TE access matrix, as discussed in Section 3.1. Rules can be specified for each kind of access vector, including the allowed, auditallow, and auditdeny vectors. The syntax of an access vector rule is:

```
te_av_rule -> av_kind source_types target_types ':' classes permissions ';'
av_kind -> ALLOW | AUDITALLOW | DONTAUDIT
source_types -> set
target_types -> set
classes -> set
permissions -> set
set -> '*' | identifier | nested_id_set | '~' identifier | '~' nested_id_set | identifier '
nested_id_set -> '{' nested_id_list '}'
nested_id_list -> nested_id_element | nested_id_list nested_id_element
nested_id_element -> identifier | '-' identifier | nested_id_set
```

As with TE transition rules, the syntax permits concise specification of multiple TE access vector rules through the optional use of sets for the source type, target type, class, or permission fields. Type attribute names can be used in any of the type fields. The tilde (~) character and the asterisk (*) character can be used in any of the fields. The minus (-) character may be used prior to a type or type attribute to exclude the type or set of types with the attribute from the type set, e.g. file_type - shadow_t would correspond to all types with the file_type attribute except for the shadow_t type. The identifier `self` can be used in the target type field to indicate that the rule should be applied between each source type and itself. If multiple classes are specified in the class field, then each permission in the permission field must be defined for that class. If multiple allow, auditallow, or dontaudit access vector rules are specified for a given type pair and class, then the union of the permission sets is used.

Several TE access vector rules from the example policy configuration related to the secure shell daemon are shown below:

```
allow sshd_t sshd_exec_t:file { read execute entrypoint };
allow sshd_t sshd_tmp_t:file { create read write getattr setattr link unlink rename };
allow sshd_t user_t:process transition;
```

The first rule specifies that the `sshd_t` domain can read, execute, and be entered via a file with the `sshd_exec_t` type. The second rule specifies that the domain can create and access files with the `sshd_tmp_t` type. The third rule specifies that the domain can transition to the `user_t` domain.

As with TE transition rules, a class macro can be used in the class field of a TE access vector rule. Care should be taken in using these macros to avoid granting unintended accesses, e.g. using the `file_class_set` macro instead of the `notdevfile_class_set` macro in a rule will grant permissions to all file classes, including device files. Since SELinux defines a large number of fine-grained permissions for each class, macros are also defined for common groupings of permissions. As with the class macros, care should be taken when using these macros to avoid granting unintended accesses. Some of these macros are shown in Table 3, Table 4, and Table 5. Any one of these macros can be used in the permissions field of a TE transition rule, as shown in the following:

```
allow sshd_t sshd_tmp_t:notdevfile_class_set create_file_perms;
allow sshd_t sshd_tmp_t:dir create_dir_perms;
```

Table 3. File Permission Macros

Macro Name	Description
stat_file_perms	Permissions to call stat or access on a file.
x_file_perms	Permissions to execute a file.
r_file_perms	Permissions to read a file.
rx_file_perms	Permissions to read and execute a file.
rw_file_perms	Permissions to read and write a file.
ra_file_perms	Permissions to read and append to a file.
link_file_perms	Permissions to link, unlink, or rename a file.
create_file_perms	Permissions to create, access, and delete a file.
r_dir_perms	Permissions to read and search a directory.

Macro Name	Description
rw_dir_perms	Permissions to read and modify a directory.
ra_dir_perms	Permissions to read and add entries to a directory.
create_dir_perms	Permissions to create, access, and delete a directory.
mount_fs_perms	Permissions to mount and unmount a filesystem.

Table 4. Socket Permission Macros

Macro Name	Description
rw_socket_perms	Permissions to use a socket.
create_socket_perms	Permissions to create and use a socket.
rw_stream_socket_perms	Permissions to use a stream socket.
create_stream_socket_perms	Permissions to create and use a stream socket.

Table 5. IPC Permission Macros

Macro Name	Description
r_sem_perms	Permissions to read a semaphore.
rw_sem_perms	Permissions to create and use a semaphore.
r_msgq_perms	Permissions to read a message queue.
rw_msgq_perms	Permissions to create and use a message queue.
r_shm_perms	Permissions to read shared memory.
rw_shm_perms	Permissions to create and use shared memory.

As discussed in Section 4.2.3, macros are defined for common groupings of TE transition rules and TE access vector rules, e.g. `domain_auto_trans` and `file_type_auto_trans`. In some cases, it is desirable to grant the necessary permissions for a domain transition or file type transition without making it the default behavior. For example, by default, `sshd_t` transitions to `user_t` when executing the shell, but it is also permitted to explicitly transition to `sysadm_t`. Hence, macros are also provided that expand to the necessary TE access vector rules without any TE transition rules: `domain_trans` and `file_type_trans`. The following example shows how the two domain transition macros are used for the secure shell daemon:

```
domain_auto_trans(sshd_t, shell_exec_t, user_t)
domain_trans(sshd_t, shell_exec_t, sysadm_t)
```

Other macros are also defined for common groupings of TE access vector rules. Some of the macros are listed in Table 6. As with other macros, care should be taken when using these macros to avoid granting unintended permissions.

Table 6. TE Access Vector Rule Macros

Macro Name	Parameters	Description
general_domain_access	The current domain.	Authorizes a domain to access processes, /proc/PID files, file descriptors, pipes, Unix sockets, and System V IPC objects within the domain. Also grants a few other common permissions for domains.
general_proc_read_access	The current domain.	Authorizes a domain to read most of /proc, excluding the /proc/PID files and certain sensitive files.
base_file_read_access	The current domain.	Authorizes a domain to read and search certain system file types.
uses_shlib	The current domain.	Authorizes a domain to execute the types for the dynamic linker and system shared libraries.
can_network	The current domain.	Authorizes a domain to create UDP and TCP sockets and to access the network.
domain_trans and domain_auto_trans	The current domain. The executable type. The new domain.	Authorizes a domain transition, and makes it the default in the latter macro.
file_type_trans and file_type_auto_trans	The creating domain. The parent directory type. The new file type. The file classes (optional)	Authorizes a file type transition, and makes it the default in the latter macro.
can_exec	The current domain. The file type.	Authorizes a domain to execute a type without changing domains.
can_exec_any	The current domain.	Authorizes a domain to execute the types for a variety of system files.
can_unix_connect	The client domain. The server domain.	Authorizes one domain to connect to another via a Unix stream socket. A separate allow rule is needed to grant access to the type for the socket file.
can_unix_send	The sending domain. The receiving domain.	Authorizes one domain to send to another via a Unix datagram socket. A separate allow rule is needed to grant access to the type for the socket file.

Macro Name	Parameters	Description
can_sysctl	The current domain.	Authorizes a domain to modify any sysctl parameters.
can_create_pty	The current domain's prefix (without the _t).	Defines a type transition for /dev/pts files and authorizes the domain for the pty type.
can_create_other_pty	The creating domain's prefix. The other domain's prefix.	Authorizes a domain to create ptys for another domain.

4.2.6. TE Access Vector Assertions

The TE configuration language allows the policy writer to define a set of access vector assertions that are checked by the policy compiler. An access vector assertion specifies permissions that should not be in an access vector for a given type pair and class. If any of the specified permissions are in an access vector for that type pair and class, then the policy compiler will reject the TE configuration. Assertions can be used to detect errors in the TE access vector rules that may not be evident from a manual inspection of the rules, due to the use of sets, macros, aliases, and attributes.

At present, assertions can only be defined for the allowed access vectors, but support for assertions on the other kinds of access vectors could be easily added. Assertions on the allowed access vector are specified using the `neverallow` keyword and otherwise use the same syntax as an access vector rule. Some examples of assertions are shown below:

```
neverallow domain ~domain:process transition;
neverallow ~{ kmod_t insmod_t rmmod_t ifconfig_t } self:capability sys_module;
neverallow local_login_t ~login_exec_t:file entrypoint;
```

The `domain` attribute is associated with all types that are used as domains in the example policy configuration, and expands to that set of types when used in this rule. The first assertion verifies that a domain can never transition to a non-domain. This ensures that every type that is used for a process is properly tagged with the `domain` attribute. The second assertion verifies that only certain domains can use the `sys_module` capability. The third assertion verifies that the `local_login_t` domain can only be entered via a program with the `login_exec_t` type.

4.2.7. Unused TE Rules

The TE configuration language defines another kind of TE rule: type member rules. Type member rules specify the type for a member of a polyinstantiated object based on the domain of the process and the type of the polyinstantiated object. Since SELinux does not yet implement polyinstantiation, these rules are not currently used and this report does not discuss them in detail. The syntax of a type member rule is identical to a TE transition rule except for the use of a `type_member` keyword. Polyinstantiation is discussed in [SpencerUsenixSec1999].

4.2.8. Example TE Configuration

Since the TE configuration specifies the fine-grained protections for processes and objects, it is the largest component of the example policy configuration. The TE configuration is organized into a collection of files to ease management. However, this internal structure is not imposed by the policy language and can be changed if desired. The only ordering restriction is that the **m4** macros must be defined prior to use.

The TE configuration files are listed in Table 7. The `macros` directory contains m4 macros that are used by the TE configuration. The `attrib.te` file declares the type attributes. The `types` directory contains declarations for general types and rules specifying relationships between these types. The `domains` directory contains the declarations and rules for each domain, and is further subdivided into several logical groupings. Types that are associated with a particular domain are declared in the appropriate domain definition file within this directory rather than in the `types` directory. The `assert.te` file specifies TE assertions.

Table 7. TE Configuration Files

Filename	Description
tunables/*.tun	Defines policy tunables for customization.
attrib.te	Defines type attributes.
macros/program/*.te	Defines macros for specific program domains.
macros/*.te	Defines commonly used macros.
types/*.te	Defines general types.
domains/user.te	Defines unprivileged user domains.
domains/admin.te	Defines administrator domains.
domains/misc/*.te	Defines miscellaneous domains not associated with a particular program.
domains/program/*.te	Defines domains for specific programs.
domains/program/unused/*.te	Optional domains for further programs.
assert.te	Defines assertions on the TE configuration.

The example TE configuration is focused on protecting the integrity of the base system, confining and protecting system processes, and protecting administrator processes. It also includes an example of how to confine a user's browser. Other kinds of security goals, such as trusted pipelines or data confidentiality, can be achieved through the TE configuration, but are not currently demonstrated in the example TE configuration.

The example TE configuration defines types to protect the integrity of the kernel, system software, system configuration information, and system logs. It defines domains for a large number of system processes and several privileged user programs, along with corresponding types for objects accessed by these processes. Three domains are defined for user processes: the `user_t` domain for ordinary users and the `staff_t` and `sysadm_t` domains for system administrators. These domains are associated with the user's initial login shell. These domains automatically transition to other domains to gain or shed permissions when user programs with certain types are run.

4.3. RBAC Statements

4.3.1. Role Declarations and Dominance

Roles are declared and authorized for particular domains (types) through role declarations and optionally through the role dominance relationship. A role declaration specifies the name of the role and the set of domains for which the role is authorized. The syntax for a role declaration is as follows:

```
role_decl -> ROLE identifier TYPES types ';'
types -> set
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{' identifier_lis
```

The RBAC configuration language uses the `types` keyword in the role declaration because the SELinux TE model uses a single type abstraction. However, specifying any type that is not a domain serves no purpose (but also does no harm). Multiple role declarations can be specified for a single role, in which case the union of the types will be authorized for the role. This feature in combination with the ability to intermingle RBAC and TE statements permits a role declaration for each domain to be located with the definition of the domain in the TE configuration if desired. This approach is used in the example policy configuration. Some examples of role declarations are shown below:

```
role system_r types { kernel_t initrc_t getty_t klogd_t };
role user_r types { user_t user_netscape_t };
role sysadm_r types { sysadm_t run_init_t };
```

Role dominance definitions can optionally be used to specify a hierarchy among roles. A role automatically inherits any domains that are authorized for any role that it dominates in the hierarchy in addition to any domains specified in role declarations. A role can be defined solely through a role dominance definition if desired, in which case the role will only be authorized for the domains of roles that it dominates. Role dominance definitions are not used in the example policy configuration. The syntax for a role dominance statement is as follows:

```
role_dominance -> DOMINANCE '{' roles '}'
roles -> role_def | roles role_def
role_def -> ROLE identifier ';' | ROLE identifier '{' roles '}'
```

4.3.2. Role Allow Rules

A role allow rule specifies authorized transitions between roles based on the pair of roles. Unlike domain transitions, the RBAC policy does not control role transitions based on the type of the entrypoint program. However, role transitions can be restricted based on type attributes using the constraints configuration, as discussed in Section 4.5. The syntax of a role allow rule is:

```
role_allow_rule -> ALLOW current_roles new_roles ';'
current_roles -> set
new_roles -> set
```

```
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{' identifier_lis
```

In the example policy configuration, role allow rules serve little purpose, since most pairings of roles need to be authorized to support normal transitions from system daemons to user or administrator shells, from user shells to administrator shells (via `newrole`), and from administrator shells to system daemons (via `run_init`). The example policy configuration uses the constraints configuration to limit role transitions to certain TE domains that are associated with processes such as `login` and `newrole`. Examples of role allow rules are shown below:

```
allow system_r { user_r sysadm_r };
allow user_r sysadm_r;
allow sysadm_r system_r;
```

4.3.3. Role Transition Rules

The RBAC configuration language defines one other kind of RBAC rule: role transition rules. A role transition rule specifies the new role of a process based on the current role of the process and the TE type of the executable. If no matching rule is specified, then the process remains in the same role by default. Although role transitions can be defined in the policy, the preferred technique for automatically changing permissions upon program execution is to define a domain transition while remaining in the same role. Role changes should typically only occur explicitly at the request of the user, although automatic role transitions can be useful for operations such as restarting daemons. The syntax of a role transition rule is:

```
role_transition_rule -> ROLE_TRANSITION current_roles types new_role ';'
current_roles -> set
types -> set
new_role -> identifier
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{' identifier_lis
```

4.3.4. Example RBAC Configuration

The RBAC configuration was originally centralized in the `rbac` file, but has been decomposed into individual role declarations, role allow rules, and role transition rules throughout the TE configuration to support easy removal or adding of domains without modifying a centralized file each time. This also allowed the macros to properly instantiate role declarations and rules for domains. The `rbac` file still exists and can contain other role rules as desired. The example RBAC configuration does not specify any role dominance declarations.

The example RBAC configuration defines four roles for processes. All system processes run in the `system_r` role. This role is authorized for each of the domains defined for system processes. The `user_r`, `staff_r` and `sysadm_r` roles are defined for ordinary users and administrators, with `staff_r` as the default login role for administrators and `sysadm_r` as the role they must enter in order to perform admin tasks. Each of these roles is authorized for the corresponding `user_t`, `staff_t`, or `sysadm_t` domain, as well as for domains for appropriate user programs. A **newrole** program was added to support

role changes within a user session and a **run_init** program was added to support running `rc` scripts in the proper role and user identity.

4.4. User Declarations

The user declarations define each user recognized by the policy and specifies the set of authorized roles for each of these users. Only the user identities specified in this configuration can be used in a security context. The user configuration has the following syntax:

```
users -> user_decl | users user_decl
user_decl -> USER identifier ROLES set ';'
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{' identifier_lis
```

The user identity attribute in the security context remains unchanged by default when a program is executed. Security-aware applications, such as the modified `login` or `sshd` programs, can explicitly specify a different user identity using the setexeccon function prior to an execve. The ability to transition to a different user identity can be controlled based on the TE domain through the constraints configuration, as discussed in Section 4.5. There are no user transition or user allow rules in the policy language. Examples of user declarations are shown below:

```
user system_u roles system_r;
user root roles { staff_r sysadm_r };
user jdoe roles user_r;
```

The example user configuration is located in the `users` file. It defines a system user identity, a generic user identity, a root user identity, and a couple of example users. The example users should be removed during installation, as mentioned in the installation `README`. Each of the other users are discussed further below.

The `system_u` user identity is defined for system processes and objects. There should be no corresponding Linux user identity in `/etc/passwd` for `system_u`, and a user process should never be assigned this identity. This user identity is authorized for the `system_r` role.

The `user_u` user identity is a generic user identity for unprivileged users that do not need to be separated by the policy. This concept was introduced in Section 3.3. There should be no corresponding Linux user identity in `/etc/passwd` for this user. libselinux will map Linux users who do not have a particular entry defined in the `users` file to this generic user identity for the SELinux security context. This user identity is authorized for the `user_r` role for ordinary users. This identity can be removed if the administrator does not wish to grant any access to users who lack specific entries in the `users` file.

The remaining users listed in the configuration correspond to Linux user identities in `/etc/passwd`. These user identities are assigned to user processes when the login shell or cron job is created. Entries are not required for pseudo users who do not perform logins or run cron jobs. Furthermore, if the `user_u` user identity is retained in this file, then entries are only required for users who should have administrator access (or who otherwise need to be separated by the policy from other users, e.g. if additional roles and domains are defined for users).

The `root` user identity is authorized for the `staff_r` and the `sysadm_r` roles. It is important to note that processes that have the Linux root uid do not necessarily have the SELinux `root` user identity, since these identities are independent. The SELinux user identity can only be set by certain TE domains such as the domain for `login`. The SELinux `root` user identity is not assigned to setuid root programs or to system processes. In some distributions that have integrated SELinux support, the SELinux user identity is set by the **su** program, while other distributions do not have **su** change the SELinux user identity.

4.5. Constraint Definitions

The constraint definitions specify additional constraints on permissions in the form of boolean expressions that must be satisfied in order for the specified permissions to be granted. The boolean expressions can be based on the user identity, role, or type attributes in the pair of security contexts. The syntax of the constraints configuration is as follows:

```
opt_constraints -> constraints | empty
constraints -> constraint_def | constraints constraint_def
constraint_def -> CONSTRAIN classes permissions cexpr ';'
classes -> set
permissions -> set
cexpr -> '(' cexpr ')' | not cexpr | expr and expr | expr or expr |
          U1 op U2 | U1 op user_set | U2 op user_set |
          R1 role_op R2 | R1 op role_set | R2 op role_set
          T1 op T2 | T1 op type_set | T2 op type_set
not -> '!' | NOT
and -> '&&' | AND
or -> '||' | OR
op -> '==' | '!='
role_op -> op | DOM | DOMBY | INCOMP
user_set -> set
role_set -> set
type_set -> set
set -> '*' | identifier | '{' identifier_list '}' | '~' identifier | '~' '{' identifier_lis
```

The same constraint can be imposed on multiple classes and permissions by optionally specifying sets for the class or permission fields. If multiple classes are specified, then each permission must be defined for each of the specified classes. The boolean expression must then evaluated to true in order for the specified permissions to be granted. Several primitives are supported in the boolean expression. The user identity attributes of the pair of security contexts, represented by `u1` and `u2`, can be compared with each other or with a particular set of user identities. Likewise, the role attributes, represented by `r1` and `r2`, can be compared with each other or with a particular set of roles. In addition to simple equality comparisons, the roles can be compared based on the dominance relationship. The type attributes, represented by `t1` and `t2`, can be compared with each other or with a particular set of types.

The example constraints configuration is located in the `constraints` file. The example policy configuration uses constraints to restrict the ability to transition to different roles or user identities to certain TE domains. To ease specification of these constraints, the example policy configuration defines type attributes for domains that are privileged with respect to setting the user identity on processes (the `privuser` attribute), domains that are privileged with respect to setting the role on processes (the

`privrole` attribute), and domains that are privileged with respect to setting the user identity on files (the `privowner` attribute). These attributes are then associated with the proper set of domains and used in the constraints configuration. Example constraints are shown below:

```
constrain process transition ( u1 == u2 or t1 == privuser );
constrain process transition ( r1 == r2 or t1 == privrole );
constrain dir_file_class_set { create relabelto relabelfrom }
          ( u1 == u2 or t1 == privowner );
```

The first constraint requires that the user identity remain the same on a process transition unless the current domain of the process is in the set of types with the `privuser` attribute. The second constraint likewise prevents role changes unless unless the current domain of the process is in the set of types with the `privrole` attribute. The last constraint prevents a process from creating or relabeling a file with a different user identity unless it has the `privowner` attribute.

4.6. Security Context Specifications

The security contexts specifications provide security contexts for various entities such as initial SIDs, pseudo filesystem entries, and network objects. It also specifies the labeling behavior to use for each filesystem type. Each of these entities and the corresponding configuration is discussed in the following subsections. The top-level production for the security contexts configuration is:

```
contexts -> initial_sid_contexts fs_uses opt_genfs_contexts net_contexts
```

4.6.1. Initial SID Contexts

As discussed in Section 2, initial SIDs are SID values that are reserved for system initialization or predefined objects. The initial SID contexts configuration specifies a security context for each initial SID. A security context consists of a user identity, a role, and a type. The syntax of the initial SID contexts configuration is shown below:

```
initial_sid_contexts -> initial_sid_context_def |
                        initial_sid_contexts initial_sid_context_def
initial_sid_context_def -> SID identifier security_context
security_context -> user ':' role ':' type
user -> identifier
role -> identifier
type -> identifier
```

The example initial SID contexts configuration is located in the `initial_sid_contexts` file. A separate domain or type is defined for each initial SID so that the TE configuration can distinguish among the initial SIDs. All of the initial SID contexts use the `system_u` user identity, since they represent system processes and objects. Initial SID contexts for processes use the `system_r` role, while those for objects use the `object_r` predefined role. Several examples of initial SID contexts entries are shown below:

```
sid kernel system_u:system_r:kernel_t
sid init   system_u:system_r:init_t
sid proc   system_u:object_r:proc_t
```

4.6.2. Filesystem Labeling Behaviors

When a filesystem is mounted by the SELinux kernel, the security server is consulted to determine the proper labeling behavior for inodes in the filesystem based on the filesystem type. The labeling behavior for a filesystem type can be specified using the `fs_use` configuration or using the `genfs_contexts` configuration. If no labeling behavior is specified for a filesystem type, then all inodes in that filesystem will be labeled with the security context associated with the `unlabeled` initial SID.

For conventional disk-based filesystem types that support extended attributes and the security xattr namespace, SELinux can use the extended attributes to determine the security context of inodes within the filesystem. This behavior is specified using a `fs_use_xattr` statement with the filesystem type name. Several examples are shown below:

```
fs_use_xattr ext2 system_u:object_r:fs_t;
fs_use_xattr ext3 system_u:object_r:fs_t;
fs_use_xattr xfs system_u:object_r:fs_t;
```

For pseudo filesystem types representing pipe and socket objects, SELinux typically assigns the context of the creating process to the inode that represents the object. This behavior is specified using the `fs_use_task` statement with the filesystem type name and a security context to use for the filesystem itself. Two examples are shown below:

```
fs_use_task pipefs system_u:object_r:fs_t;
fs_use_task sockfs system_u:object_r:fs_t;
```

For pseudo filesystems representing pseudo terminals and shared memory or temporary objects, SELinux typically assigns a context derived from both the context of the creating process and a context associated with the filesystem type. These derived contexts are determined based on type transition rules within the configuration. This behavior is specified using the `fs_use_trans` statement with the filesystem type name and a security context to use for the filesystem itself. Two examples are shown below:

```
fs_use_trans devpts system_u:object_r:devpts_t;
fs_use_trans tmpfs system_u:object_r:tmpfs_t;
```

The syntax of the `fs_use` configuration is:

```
fs_uses -> fs_use_def | fs_uses fs_use_def
fs_use_def -> FS_USE_XATTR fstype security_context ';' |
              FS_USE_TASK fstype security_context ';'
              FS_USE_TRANS fstype security_context ';'
```

4.6.3. Genfs Contexts

For filesystem types that do not support security xattrs and can not use one of the fixed labeling schemes specified in `fs_use`, the `genfs_contexts` configuration is consulted to determine a security context based on the filesystem type, the file pathname, and optionally the file type. The filesystem is labeled with the same security context as the root directory when this configuration is used. Except for proc and /selinux/booleans, all other filesystems are limited to a single entry (/) that covers all inodes in the filesystem with a default file context. Extending support to other filesystem types requires enhancements to the SELinux module and requires an assessment of whether a pathname can be reliably generated for an inode in the filesystem type, as with proc (via the proc_dir_entry tree) and /selinux/booleans.

Pathnames are specified relative to the root of the filesystem. The specification with the longest matching pathname prefix and (if specified) a matching file type is used. The file type is specified using the character shown in the mode field by **ls**. The syntax of the genfs contexts configuration is shown below:

```
opt_genfs_contexts -> genfs_contexts | empty
genfs_contexts -> genfs_context_def | genfs_contexts genfs_context_def
genfs_context_def -> GENFSCON fstype pathprefix '-' file_type security_context |
                     GENFSCON fstype pathprefix security_context
file_type -> 'b' | 'c' | 'd' | 'p' | 'l' | 's' | '-'
```

The example genfs contexts configuration is located in the `genfs_contexts` file. It provides example definitions for several pseudo filesystem types, including `proc`, `sysfs`, and `selinuxfs` among others. The example genfs contexts configuration assigns a single type to most of `/proc`, with distinct types assigned to the `kmsg` and `kcore` files as examples of finer-grained access. The `/proc/PID` directories do not use this configuration, since their contexts are implicitly derived from the context of the associated process. Example entries for `/proc` are shown below:

```
genfscon proc /    system_u:object_r:proc_t
genfscon proc /kmsg   system_u:object_r:proc_kmsg_t
genfscon proc /kcore   system_u:object_r:proc_kcore_t
```

4.6.4. Network Object Contexts

The network object contexts configuration permits the specification of security contexts for ports, network interfaces, and nodes (hosts). The security context associated with a port is used in permission checks to control the ability to bind to a given port and to send to or receive from a port. A network interface has two associated security contexts: the context of the interface and the default context to assign to unlabeled packets received on the interface (the latter is not presently used; it is an artifact of the earlier labeled networking implementation). A node has a single security context. The security contexts of network interface and nodes are used in the networking permission checks and can be used to control network traffic. For each of these objects, an appropriate initial SID is defined to use as a default

context if no matching entry is found in the configuration. The syntax of the network object security contexts configuration is shown below:

```
net_contexts -> opt_port_contexts opt_netif_contexts opt_node_contexts
opt_port_contexts -> port_contexts | empty
port_contexts -> port_context_def | port_contexts port_context_def
port_context_def -> PORTCON protocol port security_context |
                    PORTCON protocol portrange security_context
protocol -> 'tcp' | 'udp'
port -> integer
portrange -> port '-' port
opt_netif_contexts -> netif_contexts | empty
netif_contexts -> netif_context_def | netif_contexts netif_context_def
netif_context_def -> NETIFCON interface device_context packet_context
device_context -> security_context
packet_context -> security_context
opt_node_contexts -> node_contexts | empty
node_contexts -> node_context_def | node_contexts node_context_def
node_context_def -> NODECON ipv4_address ipv4_mask security_context
```

The example network contexts configuration is located in the `net_contexts` file. Security contexts are defined for many port numbers, including a general entry covering all otherwise unspecified reserved ports. Other unspecified port numbers default to the `port` initial SID. Likewise, examples are provided for security contexts for network interfaces and nodes. Several examples of network contexts entries are shown below:

```
# Ports
portcon tcp 80  system_u:object_r:http_port_t
portcon tcp 8080  system_u:object_r:http_port_t
# Network interfaces
netifcon eth0 system_u:object_r:netif_eth0_t system_u:object_r:netmsg_eth0_t
netifcon eth1 system_u:object_r:netif_eth1_t system_u:object_r:netmsg_eth1_t
# Nodes
nodecon 10.33.10.66 255.255.255.255 system_u:object_r:node_zeus_t
nodecon 10.33.10.0 255.255.255.0 system_u:object_r:node_any_t
```

4.7. File Contexts Configuration

As explained in Section 2.1, the security contexts of persistent files are maintained using extended attributes in each filesystem. The extended attributes are initialized during installation using the **setfiles** program or the package manager software for the distro. This program reads the file contexts configuration that specifies security contexts for files based on pathname regular expressions. It then creates or updates the extended attributes.

The file contexts configuration is located under the `policy/file_contexts` subdirectory. It is generated from one base configuration file (`types.fc`) and a collection of configuration files specific to

each program domain (`program/*.fc`). Each specification within these configuration files has the syntax:

```
file_context_spec -> pathname_regexp opt_security_context |
                     pathname_regexp '-' file_type opt_security_context
file_type -> 'b' | 'c' | 'd' | 'p' | 'l' | 's' | '-'
opt_security_context -> <<none>> | user ':' role ':' type
user -> identifier
role -> identifier
type -> identifier
```

By default, each pathname regular expression is an anchored match on both ends, i.e. a caret (^) is prepended and a dollar sign ($) is appended automatically. This default can be overridden by using .* at the beginning and/or end of the expression. The optional file type field specifies the file type as shown in the mode field by **ls**. If specified, then the specification must match both the pathname regular expression and the file type. The value of '<<none>>' indicates that matching files should not be relabeled. The last matching specification is used. If there are multiple hard links to a file that match different specifications, then a warning is displayed by the **setfiles** utility but the file is still labeled based on the last matching specification other than '<<none>>'

Several examples of file contexts specifications are shown below:

```
/bin(|/.*)    system_u:object_r:bin_t
/bin/login    system_u:object_r:login_exec_t
/bin/bash    system_u:object_r:shell_exec_t
/dev/[^/]*tty[^/]*  system_u:object_r:tty_device_t
.*/lost\+found(|/.*)  system_u:object_r:lost_found_t
```

5. Building and Applying the Policy

The policy configuration is compiled into a binary representation that can be loaded into the kernel. In addition to compiling and loading the policy, filesystems must be labeled appropriately in order for the policy to be applied to a system. This section describes how the policy is compiled and loaded, and how the file contexts configuration is applied to the filesystem.

5.1. Compiling and Loading the Policy

The policy configuration sources must be compiled into a binary representation before it can be read by the kernel. The compilation is performed by running **make policy** in the `/etc/selinux/(strict|targeted)/src/policy` directory. The compilation involves three steps. First, the example policy configuration files are concatenated togther. Second, the **m4** macro processor is applied to the resulting concatenation to expand macros, yielding the `policy.conf` file. The **checkpolicy** policy compiler is then run on this file to generate the binary representation in the `policy.VERSION` file, where VERSION represents the version number.

The `policy.VERSION` file can be installed into the `/etc/selinux/(strict|targeted)/policy` directory by running **make install**. The policy will then be loaded into the kernel when the kernel is next rebooted. If a runtime policy change is desired (and authorized by the policy configuration), then the **make load** command can be run to load the policy into a running kernel.

5.2. Applying the File Contexts Configuration

The file contexts configuration must be applied to the filesystem, creating or updating the extended attributes, before the extended attributes can be used by the kernel. The extended attributes can be created or updated by running **make relabel** in the policy source directory or alternatively by running **fixfiles relabel**. The **fixfiles** script is a contributed front-end wrapper for the **setfiles** program that applies setfiles to the installed file contexts configuration and all mounted filesystems that support the security extended attributes. After SELinux has been installed, the extended attributes in each filesystem are maintained dynamically by the SELinux kernel to reflect create, delete, and relabel operations. However, **fixfiles relabel** can be run at any time to update the extended attributes with a new file contexts configuration or to reset the mappings to the original configuration.

The contributed **restorecon** utility can be run to restore a specific list of files to their original settings from the file contexts configuration. restorecon can also be applied recursively via the -R option, but this is not the default behavior, unlike setfiles. Certain options to the **fixfiles** script use this utility internally rather than **setfiles** in order to apply selective relabeling.

The **chcon** utility program can also be used to set the security context of a file to a specified value. The usage of this utility is similar to the **chown** or **chmod** utilities. Unlike restorecon or setfiles, chcon allows the specific security context to be specified rather than consulting the file contexts configuration, thereby allowing for customization of specific files to more specific security contexts than the defaults.

6. Configuration Files for Security-Aware Applications

SELinux includes a set of new and modified applications that have some degree of awareness of the mandatory access controls. Some of these applications require their own configuration files that are related to the policy. This section describes these application configuration files.

The application configuration file sources are located in the `/etc/selinux/(strict|targeted)/src/policy/appconfig` directory. The configuration files used at runtime are installed under the `/etc/selinux/(strict|targeted)/contexts` directory. The most commonly used configuration files are discussed below, but there are a number of other configuration files included in the example policy and used by additional applications that are not discussed here.

6.1. Default_Contexts

Applications that need to set security contexts for user processes use the `get_default_context` or `get_ordered_context_list` libselinux functions. Internally, these functions consult the kernel policy to determine the set of legal security contexts for the user that are reachable by the application and then refine and order this set based on the `default_contexts` configuration file. Any context in a

default_contexts configuration that is not within the set of legal contexts for the user that can be reached from the application will be ignored.

Each line of the default_contexts file specifies an entry consisting of a partial context for the application followed by a list of one or more partial contexts for users in the desired prioritization order. A partial context is a context without a user identity value. Partial contexts are used in the list of user contexts since the user identity can be inferred (it is the user who was authenticated or whose crontab file was read). Partial contexts are used for the application as the application may run under different user identities at different times. In the simplest form, an entry identifies the application context and then provides a single user context to use as the default.

In the example default_contexts file, login and ssh sessions default to user_r:user_t or staff_r:staff_t. Users can then use newrole to change to a different role if authorized for another role. System cron jobs default to system_r:system_crond_t, while user cron jobs default to user_r:user_crond_t. A derived domain (user_crond_t) is used so that the policy can grant different permissions to user cron jobs than to user sessions.

An administrator may also create a per-user default_contexts file in the `/etc/selinux/(strict|targeted)/contexts/users` directory with a filename identical to the username. If such a file exists for the user, then any entries in it are given higher priority than the entries in the system-wide default_contexts file. For example, the root user typically has such a per-user default_contexts file so that he will default to sysadm_r:sysadm_t for local logins.

6.2. Default_Type

The `default_type` file defines the default type (domain) for each role. Each line specifies a role:type pair, and the appropriate type is selected by matching the role field. This file is used by programs like newrole to automatically provide a default domain when the user selects a role. If no entry is specified, then the user must explicitly specify a domain.

6.3. Initrc_Context

The `initrc_context` file defines the security context for running `/etc/rc.d` scripts via the **run_init** program. It consists of a single line specifying the proper security context. The **run_init** program transitions to this security context and then runs the specified script. This ensures that the scripts are executed from the same context as when they are run by `init`.

7. Customizing the Policy

This section describes how to customize the policy. It discusses how to perform various common changes to the policy, from adding users and permissions to defining entirely new domains, types, and roles.

7.1. Adding Users

When a user is added to the system, the policy may need to be updated to recognize the user. As discussed in Section 3.3 and Section 4.4, it may be appropriate to simply map the new user to the generic `user_u` user identity if the new user only requires unprivileged access and does not need to be separated from other such users by the policy. In that case, no updates to the policy are required.

If the user must be recognized by the policy, then the administrator must add the user to the `policy/users` file, specifying the set of authorized roles for the user, and reload the policy via **make load** in the `policy` directory.

As an example, suppose that the administrator has added a user `steve` to the system who should be authorized for both the `staff_r` and `sysadm_r` roles. To update the policy, the administrator would add an entry to the `policy/users` file as shown below, and run **make load** to reload the policy:

```
user steve roles { staff_r sysadm_r };
```

7.2. Adding Permissions

After installing SELinux, the administrator may discover that additional permissions must be allowed in order for the system to function properly. It is advisable to run SELinux in permissive mode initially and to exercise the standard operations of the system in order to generate audit messages for all operations that would have been denied by the example policy. These messages can typically be found in the **dmesg** output or `/var/log/messages` with the prefix `avc: denied`. A couple of example audit messages that might be generated during the execution of system cron jobs are shown below:

```
avc:  denied  { rename } for  pid=26878 exe=/usr/sbin/logrotate
      path=/var/log/messages.4 dev=03:02 ino=1345261
      scontext=system_u:system_r:system_crond_t
      tcontext=system_u:object_r:var_log_t tclass=file
avc:  denied  { create } for  pid=26878 exe=/usr/sbin/logrotate
      path=/var/log/messages dev=03:02 ino=1345261
      scontext=system_u:system_r:system_crond_t
      tcontext=system_u:object_r:var_log_t tclass=file
```

The critical fields of each avc denied message are the list of permissions, the source security context (scontext), the target security context (tcontext), and the target security class (tclass). These example audit messages show that the `system_crond_t` domain is being denied permissions to rename and create files with the `var_log_t` type. The other fields in each audit message provide any information about the specific processes and objects that can be determined when the audit message is generated. These messages show that the process was running the **logrotate** program and was attempting to access files in the `/var/log` directory.

The audit messages should be carefully reviewed to determine whether the denied permission should be allowed via a TE allow rule (described in Section 4.2.5). The **audit2allow** script provides an example of how to automatically convert the audit messages to TE allow rules that grant the denied permissions, but these rules should be reviewed to ensure that they do not violate the desired security goals. Other options

include placing the process into a different domain or placing the object into a different type, possibly requiring the definition of new domains and/or types. It is also sometimes desirable to continue denying the permission, but to disable auditing of the permission via a TE dontaudit rule.

In the case of the example audit messages, the denied permissions could be allowed by adding the following rule to the `policy/domains/program/crond.te` file:

```
allow system_crond_t var_log_t:file { rename create setattr unlink };
```

However, granting these permissions to the `system_crond_t` domain allows all system cron jobs to access these files. A better approach would be to define a separate domain for the `logrotate` program that has these permissions. This approach is discussed further in Section 7.4.

Not all permission denials can be solved simply through modifying the TE configuration. It may be necessary to modify the RBAC configuration (described in Section 4.3) or the constraints configuration (described in Section 4.5) as well. In the example policy, these configurations are relevant for the process transition permission when the role or user identity changes and for the file create or relabel permissions when the user identity of the file differs from the process.

After updating the policy configuration to allow the denied permissions, the administrator must then build and load the new policy by running **make load** in the `policy` directory. The permissions should then be granted on subsequent operations. If the same denials persist, then it is likely that the permission is being denied by the RBAC or constraints configuration and that these configurations were not updated by the administrator.

7.3. Adding Programs to an Existing Domain

An administrator may wish to add a program to an existing domain that is already being used for related programs that require similar permissions. First, the administrator should locate an appropriate domain by examining the existing program domains under `policy/domains/program` and by examining how existing programs are associated with the executable types for those domains in `policy/file_contexts/program`. After selecting an appropriate domain, the administrator should verify that a domain transition is defined from the desired starting domain to the new domain. If not, then an appropriate `domain_auto_trans` rule should be added to the domain's `.te` file and the policy should be reloaded via **make load**.

The administrator must then relabel the program with the executable type for the domain. This relabeling can be performed either using **chcon** or by updating the file contexts configuration and using **restorecon**, as discussed in Section 5.2. If a process is already running the program, the administrator must then restart the process in order to place it into the domain, typically using **run_init** for system processes.

As an example, suppose that an administrator wants to add a new filesystem administration utility to the system that requires similar permissions to the `fsck` program. Looking at the file contexts configuration, the administrator would see that `fsck` is labeled with the `fsadm_exec_t` type. Looking under the `policy/domains/program` directory, the administrator would find the `fsadm.te` file with the definitions for the corresponding `fsadm_t` domain. After verifying that this domain is appropriate for the new utility, the administrator can add an entry to `policy/file_contexts/program/fsadm.fc` for the new utility and run **restorecon** on the program or use **chcon** to manually set the context on the program.

7.4. Creating a New Domain

After installing SELinux or after installing a new software package, the administrator may discover that some system processes are left in the `initrc_t` domain in the output of **ps -eZ**. These system processes should either be disabled or placed into an appropriate domain. This may simply involve adding the program to an existing domain, as discussed in Section 7.3, or it may require creating a new domain. The administrator may also discover that new domains are needed to address denied permissions, as discussed in Section 7.2, for system processes or user programs. New domains are also needed when new roles are defined.

To create a new domain, the administrator should first create a new `.te` file under the `policy/domains` directory and populate it with appropriate TE declarations and rules. As an example, the creation of the `policy/domains/program/logrotate.te` file for the `logrotate` program will be discussed. The need for a separate domain for the `logrotate` program was introduced in Section 7.2. The domain definition begins by declaring the domain and its executable type using type declaration rules (described in Section 4.2.2), as shown below:

```
type logrotate_t, domain, privowner;
type logrotate_exec_t, file_type, sysadmfile, exec_type;
```

To grant the new domain a basic set of starting permissions, the `general_domain_access` macro (described in Table 6 in Section 4.2.5) can be used as shown below:

```
general_domain_access(logrotate_t)
```

For least privilege purposes, it may be desirable to instead individually define specific rules tailored for the new domain.

If the program is known to create files in shared directories, e.g. `/tmp` files, then the administrator can declare types for these files and file type transition rules (described in Section 4.2.3). An example type declaration and file type transition rule for temporary files created by `logrotate` is shown below:

```
type logrotate_tmp_t, file_type, sysadmfile, tmpfile;
file_type_auto_trans(logrotate_t, tmp_t, logrotate_tmp_t)
```

More concisely, this same set of rules could be expressed using the `tmp_domain` macro. Likewise, if the program is known to require certain permissions, then these permissions can be allowed by the administrator. Since the administrator knows that the program requires permissions to the `/var/log` files, the following rules might be initially specified:

```
allow logrotate_t var_log_t:dir rw_dir_perms;
allow logrotate_t var_log_t:file create_file_perms;
```

To cause the domain to be entered automatically from system cron jobs and from administrator shells when `logrotate` is executed, domain transition rules (described in Section 4.2.3) should be added for the appropriate domains. These rules can either be placed in the new domain's `.te` file or in the files for the source domains. Typically, if the source domain transitions to many different domains (e.g. every daemon or many programs), it is preferable to place the rule in the target domain to ease adding new domains and provide better encapsulation. Examples of these rules are shown below:

```
domain_auto_trans(system_crond_t, logrotate_exec_t, logrotate_t)
domain_auto_trans(sysadm_t, logrotate_exec_t, logrotate_t)
```

The first rule along with some related allow rules that are typically needed for system cron jobs can be more concisely written using the `system_crond_entry` macro.

After providing a minimal definition of the domain and transitions into the domain, the administrator should authorize roles for the domain. Role declarations (described in Section 4.3.1) can be placed either in the domain's `.te` file or in the `policy/rbac` file. The former approach is preferable in order to encapsulate the domain's definition. Role declarations for the `logrotate_t` domain are shown below:

```
role system_r types logrotate_t;
role sysadm_r types logrotate_t;
```

The updated policy configuration can then be compiled and loaded by running **make load** in the `policy` directory. The administrator should then add the program to the file contexts configuration and run **restorecon** on the program or run **chcon** to manually set the context. A `policy/file_contexts/program/logrotate.fc` configuration file for `logrotate` is shown below:

```
/usr/sbin/logrotate  system_u:object_r:logrotate_exec_t
```

The administrator can then try running the program in its new domain to discover whether additional permissions are required. If the program is to be run as a system process, the administrator should use **run_init** to start it. If additional permissions are required, then the steps in Section 7.2 can be followed to complete the domain.

7.5. Creating a New Type

New types can be created to provide distinct protection for specific objects. An administrator may also discover that new types are needed to address denied permissions, as discussed in Section 7.2. To create a new type, the administrator should first add a type declaration (described in Section 4.2.2) to the TE configuration. If the type is associated with a particular domain, then the declaration should be placed in the domain's `.te` file. If the type is a general type, then the declaration can be placed in one of the files under `policy/types`.

If automatic transitions to this type are desired, then the administrator should define type transition (described in Section 4.2.3) rules for the appropriate domains. The administrator should add appropriate TE allow rules to the TE configuration to permit authorized domains to access the type. The administrator can then build and reload the policy via **make load**. After updating the policy, the administrator can then apply the type to a file by updating the file contexts configuration and running **restorecon** on the file or by using **chcon**.

As an example, consider the `/dev/initctl` named pipe, which is used to interact with the `init` process. The `initctl_t` type was defined for this file in the `policy/domains/program/init.te` file, as shown below:

```
type initctl_t, file_type, sysadmfile;
```

Since this file is created at runtime, a file type transition rule must be specified to ensure that it is always created with this type. The file type transition rule for this type is:

```
file_type_auto_trans(init_t, device_t, initctl_t)
```

Two other domains need to access this object: the domain for the `/etc/rc.d` scripts and the domain for the system administrator. Hence, the following TE allow rules are added to the `policy/domains/program/initrc.te` and `policy/domains/admin.te` files:

```
allow initrc_t initctl_t:fifo_file rw_file_perms;
allow sysadm_t initctl_t:fifo_file rw_file_perms;
```

The policy can then be reloaded via **make load**. The administrator would then add the following entry to `policy/file_contexts/program/init.fc` and apply restorecon to the file:

```
/dev/initctl   system_u:object_r:initctl_t
```

7.6. Creating a New Role

New roles can be created to provide separation among users beyond the simple division between ordinary users and administrators. To add a new role, the administrator should use the `full_user_role` macro to instantiate a role and a set of domain definitions for the initial login domain for the role and for programs run from the role. To allow transitions between the new role and other roles, the `role_tty_type_change` macro in `domains/user.te` may be used. Appropriate users should be authorized for the new role in `policy/users`. The policy can then be reloaded via **make load**.

After updating the policy, the administrator should add an entry for the role to the `/etc/selinux/(strict|targeted)/contexts/default_type` application configuration file.

References

[BoebertNCSC1985] W. Boebert and R. Kain, “A Practical Alternative to Hierarchical Integrity Policies”, *Proceedings of the Eighth National Computer Security Conference*, 1985.

[FerraioloNCSC1992] David Ferraiolo and Richard Kuhn, “Role-Based Access Controls”, *Proceedings of the 15th National Computer Security Conference*, October 1992.

[LoscoccoFreenix2001] Peter Loscocco and Stephen Smalley, “Integrating Flexible Support for Security Policies into the Linux Operating System”, *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, The USENIX Association, June 2001.

[LoscoccoOLS2001] Peter Loscocco and Stephen Smalley, “Meeting Critical Security Objectives with Security-Enhanced Linux”, *Proceedings of the 2001 Ottawa Linux Symposium*, July 2001.

[LoscoccoNSATR2001] Peter Loscocco and Stephen Smalley, “Integrating Flexible Support for Security Policies into the Linux Operating System”, *NSA Technical Report*, February 2001.

[SmalleyNAITR2001] Stephen Smalley and Timothy Fraser, “A Security Policy Configuration for the Security-Enhanced Linux”, *NAI Labs Technical Report*, February 2001.

[SmalleyModuleTR2001] Stephen Smalley, Chris Vance, and Wayne Salamon, “Implementing SELinux as a Linux Security Module”, *NAI Labs Report #01-043*, December 2001.

[SpencerUsenixSec1999] Ray Spencer, Stephen Smalley, Peter Loscocco, Mike Hibler, David Andersen, and Jay Lepreau, “The Flask Security Architecture: System Support for Diverse Security Policies”, *Proceedings of the Eighth USENIX Security Symposium*, The USENIX Association, August 1999.