The SELinux Notebook

April 30, 2024 (rev 2bf1132b22d7)

Copyright Information

Acknowledgements

The Notebook was originally created by *Richard Haines* who graciously donated the source material to the SELinux project.

The SELinux logo was designed by *Máirín Duffy*.

Introduction

This Notebook should help with explaining:

1. SELinux and its purpose in life.
2. The LSM / SELinux architecture, its supporting services and how they are implemented within GNU / Linux.
3. SELinux Networking, Virtual Machine, X-Windows, PostgreSQL and Apache/SELinux-Plus SELinux-aware capabilities.
4. The core SELinux kernel policy language and how basic policy modules can be constructed for instructional purposes.
5. An introduction to the new Common Intermediate Language (CIL) implementation.
6. The core SELinux policy management tools with examples of usage.
7. The Reference Policy architecture, its supporting services and how it is implemented.
8. The integration of SELinux within Android.

Notebook Overview

This volume has the following major sections:

SELinux Overview - Gives a description of SELinux and its major components to provide Mandatory Access Control services for GNU / Linux. Hopefully it will show how all the SELinux components link together and how SELinux-aware applications / object manager have been implemented (such as Networking, X-Windows, PostgreSQL and virtual machines).

SELinux Configuration Files - Describes all known SELinux configuration files with samples. Also lists any specific SELinux commands or libselinux APIs used by them.

SELinux Policy Language - Gives a brief description of each policy language statement, with supporting examples taken from the Reference Policy source. Also an introduction to the new CIL language (Common Intermediate Language).

The Reference Policy - Describes the Reference Policy and its supporting macros.

Android - An overview of the SELinux services used to support Android.

Object Classes and Permissions - Describes the SELinux object classes and permissions.

Notebook Examples

The Notebook examples are not embedded into any of the document formats described in **https://github.com/SELinuxProject/selinux-notebook/blob/main/BUILD.md**, however they will have links to them in their build directories.

Updated Editions

The SELinux Notebook is being maintained as part of the SELinux project, more recent editions may be available.

See: **https://github.com/SELinuxProject/selinux-notebook**

Table of Contents

Abbreviations and Terminology

Abbreviations

AV

Access Vector

AVC

Access Vector Cache

BLP

Bell-La Padula

CC

Common Criteria

CIL

Common Intermediate Language

CMW

Compartmented Mode Workstation

DAC

Discretionary Access Control

FLASK

Flux Advanced Security Kernel

Fluke

Flux kernel Environment

Flux

The Flux Research Group

ID

Identification

LSM

Linux Security Module

LAPP

Linux, Apache, PostgreSQL, PHP / Perl / Python

LSPP

Labeled Security Protection Profile

MAC

Mandatory Access Control

MCS

Multi-Category Security

MLS

Multi-Level Security

MMAC

Middleware Mandatory Access Control

NSA

National Security Agency

OM

Object Manager

OTA

over the air

PAM

Pluggable Authentication Module

RBAC

Role-based Access Control

RBACSEP

Role-based Access Control Separation

rpm

Red Hat Package Manager

SELinux

Security Enhanced Linux

SID

Security Identifier

SMACK

Simplified Mandatory Access Control Kernel

SUID

Super-user Identifier

TE

Type Enforcement

UID

User Identifier

XACE

X (windows) Access Control Extension

Terminology

These give a brief introduction to the major components that form the core SELinux infrastructure.

Access Vector (AV)

A bit map representing a set of permissions (such as open, read, write).

Access Vector Cache (AVC)

A component that stores access decisions made by the SELinux **Security Server** for subsequent use by **Object Managers**. This allows previous decisions to be retrieved without the overhead of re-computation. Within the core SELinux services there are two **Access Vector Caches**:

1. A kernel AVC that caches decisions by the **Security Server** on behalf of kernel based object managers.
2. A userspace AVC built into libselinux that caches decisions when SELinux-aware applications use ***avc_open****(3)* with ***avc_has_perm****(3)* or ***avc_has_perm_noaudit****(3)* function calls. This will save calls to the kernel after the first decision has been made. Note that the preferred option is to use the ***selinux_check_access****(3)* function as this will utilise the userspace AVC as explained in the **Computing Access Decisions** section.

Domain

For SELinux this consists of one or more processes associated to the type component of a **Security Context**. **Type Enforcement** rules declared in policy describe how the domain will interact with objects (see **Object Class**).

Linux Security Module (LSM)

A framework that provides hooks into kernel components (such as disk and network services) that can be utilised by security modules (e.g. **SELinux** and SMACK) to perform access control checks. Work is in progress to stack multiple modules

Mandatory Access Control

An access control mechanism enforced by the system. This can be achieved by 'hard-wiring' the operating system and applications or via a policy that conforms to a **Policy**. Examples of policy based MAC are **SELinux** and SMACK.

Multi-Level Security (MLS)

Based on the Bell-La & Padula model (BLP) for confidentiality in that (for example) a process running at a 'Confidential' level can read / write at their current level but only read down levels or write up levels. While still used in this way, it is more commonly used for application separation utilising the Multi-Category Security (MCS) variant.

Object Class

Describes a resource such as files, sockets or services. Each 'class' has relevant permissions associated to it such as read, write or export. This allows access to be enforced on the instantiated object by their **Object Manager**.

Object Manager

Userspace and kernel components that are responsible for the labeling, management (e.g. creation, access, destruction) and enforcement of the objects under their control. **Object Managers** call the **Security Server** for an access decision based on a source and target **Security Context** (or **SID**), an **Object Class** and a set of permissions (or **AV**s). The **Security Server** will base its decision on whether the currently loaded **Policy** will allow or deny access. An **Object Manager** may also call the **Security Server** to compute a new **Security Context** or **SID** for an object.

Policy

A set of rules determining access rights. In SELinux these rules are generally written in a kernel policy language using either ***m4(1)*** macro support (e.g. Reference Policy) or the CIL language. The **Policy** is then compiled into a binary format for loading into the **Security Server**.

Role Based Access Control

SELinux users are associated to one or more roles, each role may then be associated to one or more **Domain** types.

Role Based Access Control-Separation

Role-based separation of user home directories. An optional policy tunable is required: *tunableif enable_rbacsep*

Security Server

A sub-system in the Linux kernel that makes access decisions and computes security contexts based on **Policy** on behalf of SELinux-aware applications and Object Managers. The **Security Server** does not enforce a decision, it merely states whether the operation is allowed or not according to the **Policy**. It is the SELinux-aware application or **Object Manager** responsibility to enforce the decision.

Security Context

An SELinux **Security Context** is a variable length string that consists of the following mandatory components *user:role:type* and an optional *[:range]* component. Generally abbreviated to ‘context’, and sometimes called a ‘label’.

Security Identifier (SID)

SIDs are unique opaque integer values mapped by the kernel **Security Server** and userspace AVC that represent a **Security Context**. The SIDs generated by the kernel **Security Server** are `u32` values that are passed via the **Linux Security Module** hooks to/from the kernel **Object Managers**.

Type Enforcement

SELinux makes use of a specific style of type enforcement (TE) to enforce **Mandatory Access Control**. This is where all subjects and objects have a type identifier associated to them that can then be used to enforce rules laid down by **Policy**.

SELinux Overview

SELinux is the primary Mandatory Access Control (MAC) mechanism built into a number of GNU / Linux distributions. SELinux originally started as the Flux Advanced Security Kernel (FLASK) development by the Utah university Flux team and the US Department of Defence. The development was enhanced by the NSA and released as open source software (see: https://www.nsa.gov/what-we-do/research/selinux/).

Each of the sections that follow will describe a component of SELinux, and hopefully they are in some form of logical order.

Note: When SELinux is installed, there are three well defined directory locations referenced. Two of these will change with the old and new locations as follows:

Description	Old Location	New Location
The SELinux filesystem that interfaces with the kernel based security server. The new location has been available since Fedora 17.	*/selinux*	*/sys/fs/selinux*
The SELinux configuration directory that holds the sub-system configuration files and policies.	*/etc/selinux*	No change
The SELinux policy store that holds policy modules and configuration details. The new location has been available since Fedora 23.	*/etc/selinux/<SELINUXTYPE>/module*	*/var/lib/selinux/<SELINUXTYPE>*

Is SELinux useful

There are many views on the usefulness of SELinux on Linux based systems, this section gives a brief view of what SELinux is good at and what it is not (because it's not designed to do it).

SELinux is not just for military or high security systems where Multi-Level Security (MLS) is required (for functionality such as 'no read up' and 'no write down'), as using the 'type enforcement' (TE) functionality applications can be confined (or contained) within domains and limited to the minimum privileges required to do their job, so in a 'nutshell':

1. **The First Security Principle:** ***Trust nothing, trust nobody, all they want is your money and/or your information*** (*unless of course you can prove otherwise beyond all reasonable doubt*).
2. If SELinux is enabled, the policy defines what access to resources and operations on them (e.g. read, write) are allowed (i.e. SELinux stops all access unless allowed by policy). This is why SELinux is called a 'mandatory access control' (MAC) system.
3. The policy design, implementation and testing against a defined security policy or requirements is important, otherwise there could be 'a false sense of security'.
4. SELinux can confine an application within its own 'domain' and allow it to have the minimum privileges required to do its job. Should the application require access to networks or other applications (or their data), then (as part of the security policy design), this access would need to be granted (so at least it is known what interactions are allowed and what are not - a good security goal).

5. Should an application 'do something' it is not allowed by policy (intentional or otherwise), then SELinux would stop these actions.
6. Should an application 'do something' it is allowed by policy, then SELinux may contain any damage that maybe done intentional or otherwise. For example if an application is allowed to delete all of its data files or database entries and the bug, virus or malicious user gains these privileges then it would be able to do the same. However the good news is that if the policy 'confined' the application and data, all your other data should still be there.
7. User login sessions can be confined to their own domains. This allows clients they run to be given only the privileges they need (e.g. admin users, sales staff users, HR staff users etc.). This again will confine/limit any damage or leakage of data.
8. Some applications (X-Windows for example) are difficult to confine as they are generally designed to have total access to all resources. SELinux can generally overcome these issues by providing sandboxing services.
9. SELinux will not stop memory leaks or buffer over-runs (because it's not designed to do this), however it may contain the damage that may be caused by these flaws.
10. SELinux will not stop all viruses/malware getting into the system, as there are many ways they could be introduced (including legitimate users), however it should limit the damage or leaks they cause.
11. SELinux will not stop kernel vulnerabilities, however it may limit their effects.
12. If a user has the relevant permissions it is easy to add new rules to a SELinux policy using tools such as ***audit2allow****(1)*. Nevertheless be aware that this may start opening holes, so do double check the necessity of a given rule.
13. Finally, SELinux cannot stop anything allowed by the security policy, so good design is important.

The following maybe useful in providing a practical view of SELinux:

1. Your visual how-to guide for SELinux policy enforcement, available from: https://opensource.com/business/13/11/selinux-policy-guide
2. A discussion regarding Apache servers and SELinux that may look negative at first but highlights the containment points above. This is the initial study: http://blog.ptsecurity.com/2012/08/selinux-in-practice-dvwa-test.html, and this is a response to the study: http://danwalsh.livejournal.com/56760.html.
3. SELinux services have been added to Android. The presentation "Security Enhancements (SE) for Android" gives use-cases and types of Android exploits that SELinux could have overcome. The presentation and others are available at: https://events.static.linuxfound.org/sites/events/files/slides/abs2014_seforandroid_smalley.pdf
4. Older NSA documentation at: https://www.nsa.gov/what-we-do/research/selinux/documentation/ that is informative.

Core SELinux Components

Figure 1 shows a high level diagram of the SELinux core components that manage enforcement of the policy and comprise of the following:

1. A **Subject** that must be present to cause an action to be taken by an **Object** (such as read a file as information only flows when a subject is involved).
2. An Object Manager that knows the actions required of the particular resource (such as a file) and can enforce those actions (i.e. allow it to write to a file if permitted by the policy).
3. A Security Server that makes decisions regarding the subjects rights to perform the requested action on the object, based on the security policy rules.
4. A Security Policy that describes the rules using the SELinux **Kernel Policy Language**).
5. An Access Vector Cache (AVC) that improves system performance by caching security server decisions.

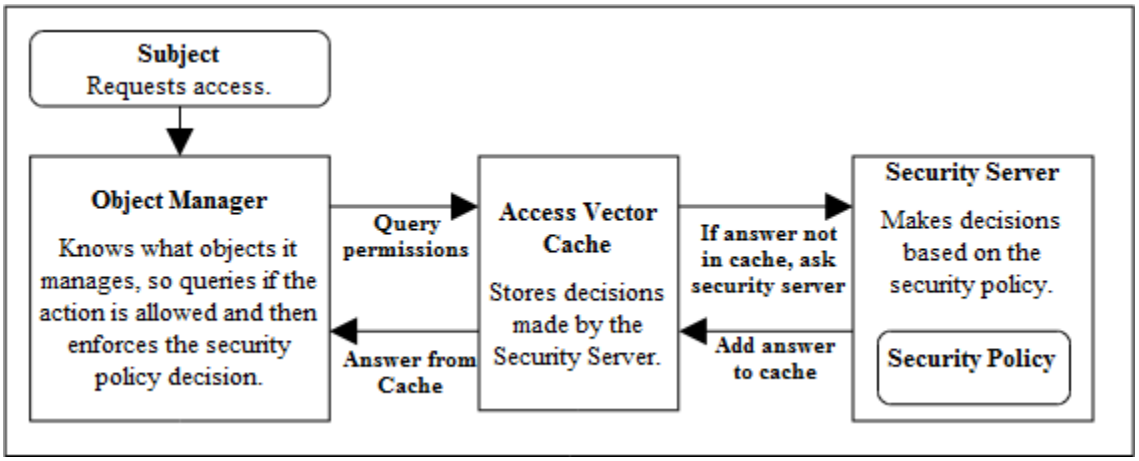


Figure 1: High Level Core SELinux Components - *Decisions by the Security Server are cached in the AVC to enhance performance of future requests. Note that it is the kernel and userspace Object Managers that enforce the policy.*

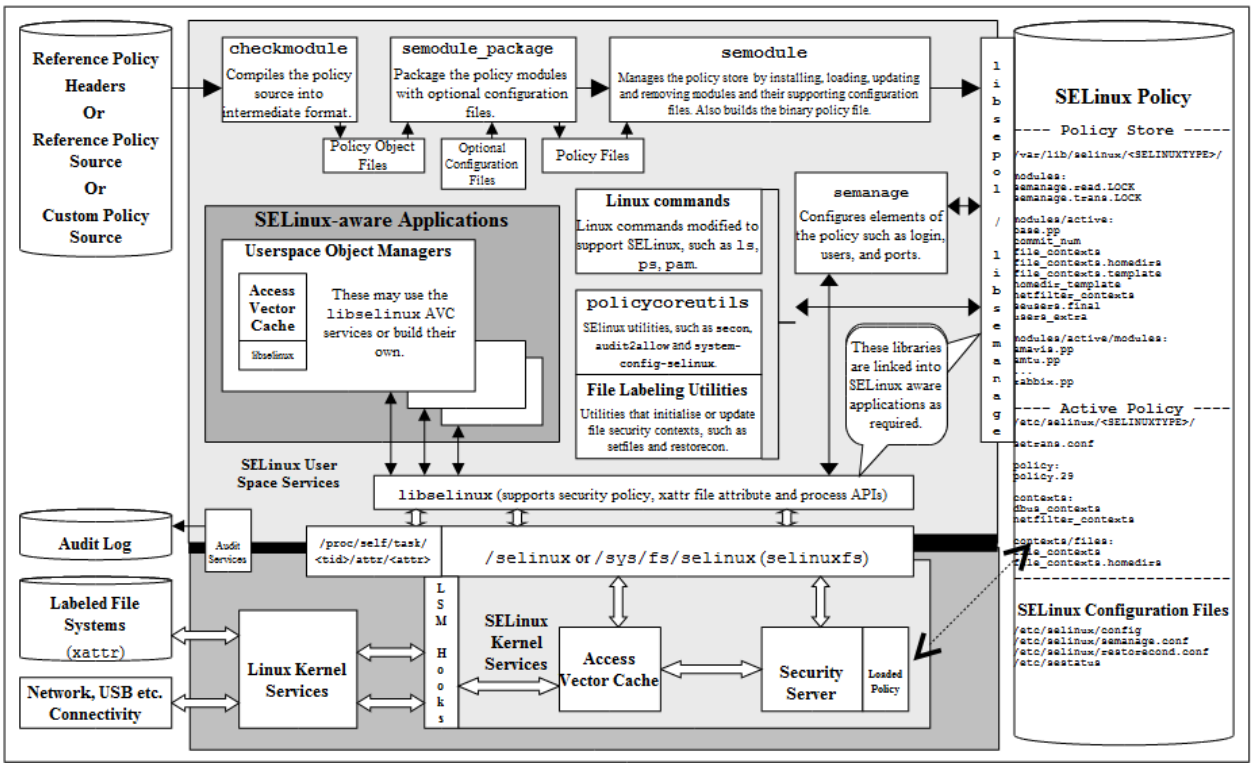


Figure 2: High Level SELinux Architecture - *Showing the major supporting services*

Figure 2 shows a more complex diagram of kernel and userspace with a number of supporting services that are used to manage the SELinux environment. This diagram will be referenced a number of times to explain areas of SELinux, therefore starting from the bottom:

1. In the current implementation of SELinux the security server is embedded in the kernel with the policy being loaded from userspace via a series of functions contained in the **libselinux** library (see **SELinux Userspace Libraries** for details). The object managers (OM) and access vector cache (AVC) can reside in:
    - **kernel space** - These object manages are for the kernel services such as files, directory, socket, IPC etc. and are provided by hooks into the SELinux sub-system via the Linux Security Module (LSM) framework (shown as LSM Hooks in) that is discussed in the **Linux Security Module and SELinux** section. The SELinux kernel AVC service is used to cache the security servers response to the kernel based object managers thus speeding up access decisions should the same request be asked in future.
    - **userspace** - These object managers are provided with the application or service that requires support for MAC and are known as 'SELinux-aware' applications or services. Examples of these are: X-Windows, D-bus messaging (used by the Gnome desktop), PostgreSQL database, Name Service Cache Daemon (*nscd*), and the GNU / Linux passwd command. Generally, these OMs use the AVC services built into the SELinux library (libselinux), however they could, if required supply their own AVC or not use an AVC at all (see **Implementing SELinux-aware Applications** for details).
2. The SELinux security policy (right hand side of **Figure 2**) and its supporting configuration files are contained in the /etc/selinux directory. This directory contains the main SELinux configuration file *config* that has the name of the policy to be loaded (via the *SELINUXTYPE* entry) and the initial enforcement mode[1] of the policy at load time (via the *SELINUX* entry). The */etc/selinux/<SELINUXTYPE>* directories contain policies that can be activated along with their configuration files (e.g. '*SELINUXTYPE=targeted*' will have its policy and associated configuration files located at */etc/selinux/targeted*). All known configuration files are shown in the **SELinux Configuration Files** sections.
3. SELinux supports a 'modular policy', this means that a policy does not have to be one large source policy but can be built from modules. A modular policy consists of a base policy that contains the mandatory information (such as object classes, permissions etc.), and zero or more policy modules where generally each supports a particular application or service. These modules are compiled, linked, and held in a 'policy

store' where they can be built into a binary format that is then loaded into the security server (in the diagram the binary policy is located at */etc/selinux/targeted/policy/policy.30*). The types of policy and their construction are covered in the **Types of SELinux Policy** section.

4. To be able to build the policy in the first place, policy source is required (top left hand side of **Figure 2**). This can be supplied in three basic ways:
    1. as source code written using the **Kernel Policy Language**, however it is not recommended for large policy developments.
    2. using the **Reference Policy** that has high level macros to define policy rules. This is the standard way policies are now built for SELinux distributions such as Red Hat and Debian and is discussed in the **The Reference Policy** section. Note that SE for Android also uses high level macros to define policy rules.
    3. using CIL (Common Intermediate Language). An overview can be found in the **CIL Policy Language** section. The https://github.com/DefenSec/dssp is a good example.
5. To be able to compile and link the policy source then load it into the security server requires a number of tools (top of **Figure 2**).
6. To enable system administrators to manage policy, the SELinux environment and label file systems, tools and modified GNU / Linux commands are used. These are mentioned throughout the Notebook as needed and summarised in **SELinux Commands**. Note that there are many other applications to manage policy, however this Notebook only concentrates on the core services.
7. To ensure security events are logged, GNU / Linux has an audit service that captures policy violations. The **Auditing Events** section describes the format of these security events.
8. SELinux supports network services that are described in the **SELinux Networking Support** section.

The **Linux Security Module and SELinux** section goes into greater detail of the LSM / SELinux modules with a walk through of a ***fork****(2)* and ***exec****(2)* process.

Mandatory Access Control

Mandatory Access Control (MAC) is a type of access control in which the operating system is used to constrain a user or process (the subject) from accessing or performing an operation on an object (such as a file, disk, memory etc.).

Each of the subjects and objects have a set of security attributes that can be interrogated by the operating system to check if the requested operation can be performed or not. For SELinux the:

- **subjects** are processes.
- **objects** are system resources such as files, sockets, etc.
- security attributes are the **security context**.
- Security Server within the Linux kernel authorizes access (or not) using the security policy (or policy) that describes rules that must be enforced.

Note that the subject (and therefore the user) cannot decide to bypass the policy rules being enforced by the MAC policy with SELinux enabled. Contrast this to standard Linux Discretionary Access Control (DAC), which also governs the ability of subjects to access objects, however it allows users to make policy decisions. The steps in the decision making chain for DAC and MAC are shown in **Figure 3**.

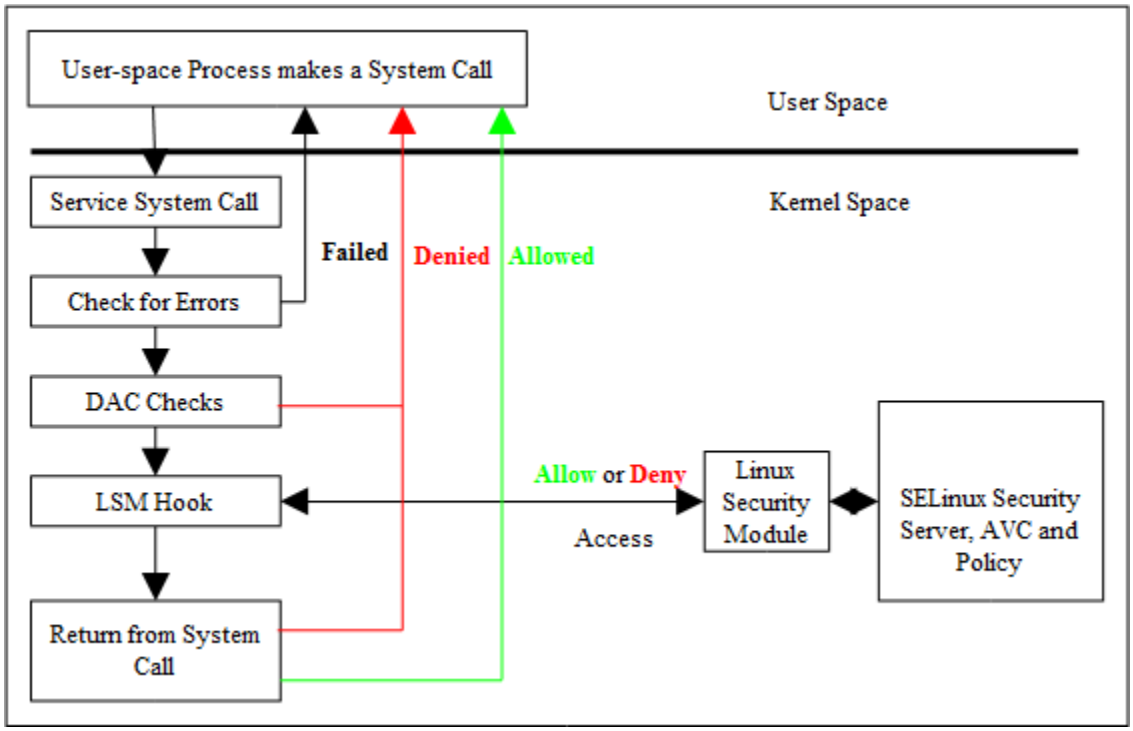


Figure 3: Processing a System Call - *The DAC checks are carried out first, if they pass then the Security Server is consulted for a decision.*

SELinux supports two forms of MAC:

Type Enforcement - Where processes run in domains and the actions on objects are controlled by policy. This is the implementation used for general purpose MAC within SELinux along with Role Based Access Control. The **Type Enforcement (TE)** and **Role Based Access Control** sections covers these in more detail.

Multi-Level Security - This is an implementation based on the Bell-La Padula (BLP) model, and used by organizations where different levels of access are required so that restricted information is separated from classified information to maintain confidentiality. This allows enforcement rules such as 'no write down' and 'no

read up' to be implemented in a policy by extending the security context to include security levels. The **MLS** section covers this in more detail along with a variant called Multi-Category Security (MCS).

The MLS / MCS services are now more generally used to maintain application separation, for example SELinux enabled:

- virtual machines use MCS categories to allow each VM to run within its own domain to isolate VMs from each other (see the **SELinux Virtual Machine Support** section).
- Android devices use dynamically generated MCS categories so that an app running on behalf of one user cannot read or write files created by the same app running on behalf of another user (see the **Security Enhancements for Android - Computing a Context** section).

SELinux Users

Users in Linux are generally associated to human users (such as Alice and Bob) or operator/system functions (such as admin), while this can be implemented in SELinux, SELinux user names are generally groups or classes of user. For example all the standard system users could be assigned an SELinux user name of *user_u* and administration staff under *staff_u*.

There is one special SELinux user defined that must never be associated to a Linux user as it a special identity for system processes and objects, this user is *system_u*.

The SELinux user name is the first component of a **Security Context** and by convention SELinux user names end in *_u*, however this is not enforced by any SELinux service (i.e. it is only to identify the user component), although CIL with namespaces does make identification of an SELinux user easier for example a 'user' could be declared as *unconfined.user*.

It is possible to add constraints and bounds on SELinux users as discussed in the **Type Enforcement (TE)** section.

Some policies, for example Android, only make use of one user called *u*.

Role-Based Access Control

To further control access to TE domains SELinux makes use of role-based access control (RBAC). This feature allows SELinux users to be associated to one or more roles, where each role is then associated to one or more domain types as shown in **Figure 4: Role Based Access Control**.

The SELinux role name is the second component of a 'security context' and by convention SELinux roles end in *_r*, however this is not enforced by any SELinux service (i.e. it is only used to identify the role component), although CIL with namespaces does make identification of a role easier for example a 'role' could be declared as *unconfined.role*.

It is possible to add constraints and bounds on roles as discussed in the **Type Enforcement** section.

Some policies, for example Android, only make use of one role called *r*.

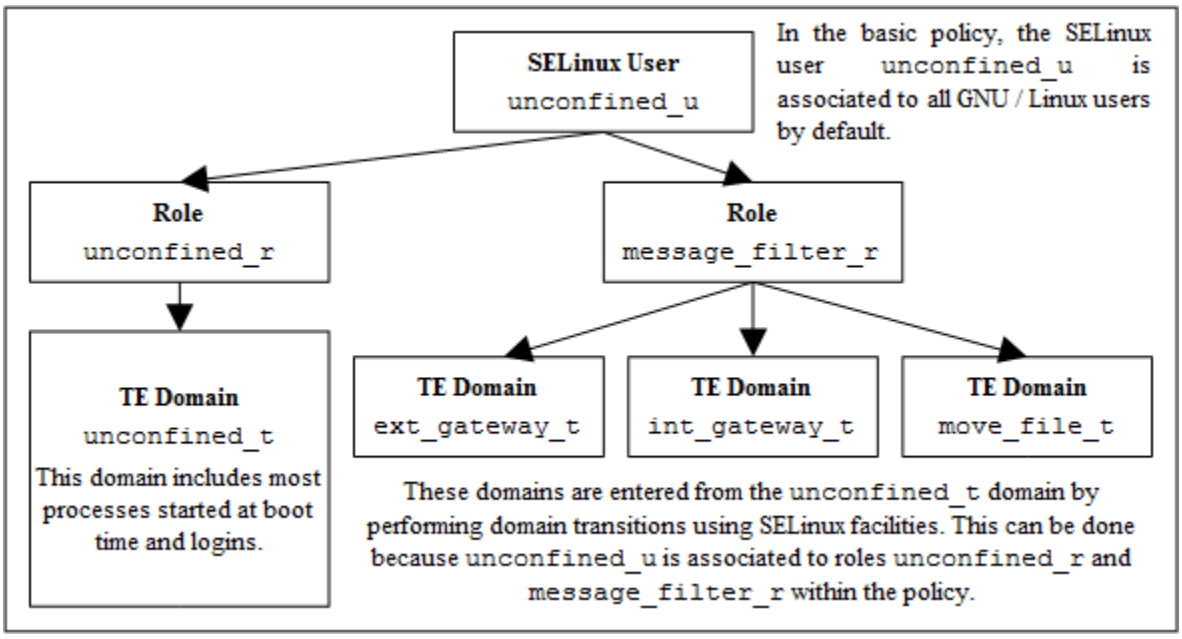


Figure 4: Role Based Access Control - *Showing how SELinux controls access via user, role and domain type association.*

Type Enforcement

SELinux makes use of a specific style of type enforcement (TE) to enforce mandatory access control. For SELinux it means that all **subjects** have a *type* identifier associated to them that can then be used to enforce rules laid down by policy.

The SELinux *type* identifier is a simple variable-length string that is defined in the policy and then associated to a **security context**. It is also used in the majority of **SELinux language statements and rules** used to build a policy that will, when loaded into the security server, enforce policy via the object managers.

Because the *type* identifier (or just 'type') is associated to all subjects and objects, it can sometimes be difficult to distinguish what the type is actually associated with (it's not helped by the fact that by convention, type identifiers end in *_t*). In the end it comes down to understanding how they are allocated in the policy itself and how they are used by SELinux services (although CIL policies with namespaces do help in that a domain process 'type' could be declared as *msg_filter.ext_gateway.process* with object types being any others (such as *msg_filter.ext_gateway.exec*).

Basically if the type identifier is used to reference a subject it is referring to a Linux process or collection of processes (a domain or domain type). If the type identifier is used to reference an object then it is specifying its object type (i.e. file type).

While SELinux refers to a subject as being an active process that is associated to a domain type, the scope of an SELinux type enforcement domain can vary widely. For example in the simple **Kernel policy** in the *notebook-examples*, all the processes on the system run in the *unconfined_t* domain, therefore every process is 'of type *unconfined_t*' (that means it can do whatever it likes within the limits of the standard Linux DAC policy as all access is allowed by SELinux).

It is only when additional policy statements are added to the simple policy that areas start to be confined. For example, an external gateway is run in its own isolated domain (*ext_gateway_t*) that cannot be 'interfered' with by any of the *unconfined_t* processes (except to run or transition the gateway process into its own domain). This scenario is similar to the 'targeted' policy delivered as standard in Red Hat Fedora where the majority of user space processes run under the *unconfined_t* domain.

The SELinux type is the third component of a 'security context' and by convention SELinux types end in *_t*, however this is not enforced by any SELinux service (i.e. it is only used to identify the type component), although as explained above CIL with namespaces does make identification of types easier.

Constraints

It is possible to add constraints on users, roles, types and MLS ranges, for example within a TE environment, the way that subjects are allowed to access an object is via a TE ***allow***, for example:

```
allow unconfined_t ext_gateway_t : process transition;
```

This states that a process running in the *unconfined_t* domain has permission to transition a process to the *ext_gateway_t* domain. However it could be that the policy writer wants to constrain this further and state that this can only happen if the role of the source domain is the same as the role of the target domain. To achieve this a constraint can be imposed using a ***constrain*** statement:

```
constrain process transition ( r1 == r2 );
```

This states that a process transition can only occur if the source role is the same as the target role, therefore a constraint is a condition that must be satisfied in order for one or more permissions to be granted (i.e. a constraint imposes additional restrictions on TE rules). Note that the constraint is based on an object class (*process* in this case) and one or more of its permissions.

The kernel policy language constraints are defined in the **Constraint Statements** section.

Bounds

It is possible to add bounds to users, roles and types, however currently only types are enforced by the kernel using the *typebounds* rule as described in the **Apache-Plus Support - Bounds Overview** section.

User and role bounds may be declared using CIL, however they are validated at compile time by the CIL compiler and NOT enforced by the SELinux kernel services. The **Bounds Rules** section defines the *typebounds* rule and also gives a summary of the *userbounds* and *rolebounds* rules.

Security Context

SELinux requires a security context to be associated with every process (or subject) and object that are used by the security server to decide whether access is allowed or not as defined by the policy.

The security context is also known as a 'security label' or just label that can cause confusion as there are many types of label depending on the context.

Within SELinux, a security context is represented as variable-length strings that define the SELinux user (this is not the Linux user id. The Linux user id is mapped to the SELinux user id by configuration files), their role, a type identifier and an optional MCS / MLS security range or level as follows:

```
user:role:type[:range]
```

Where:

user

- The SELinux user identity. This can be associated to one or more roles that the SELinux user is allowed to use.

role

- The SELinux role. This can be associated to one or more types the SELinux user is allowed to access.

type

- When a type is associated with a process, it defines what processes (or domains) the SELinux user (the subject) can access. When a type is associated with an object, it defines what access permissions the SELinux user has to that object.

range

- This field can also be know as a *level* and is only present if the policy supports MCS or MLS. The entry can consist of:
  - A single security level that contains a sensitivity level and zero or more categories (e.g. *s0*, *s1:c0*, *s7:c10.c15*).
  - A range that consists of two security levels (a low and high) separated by a hyphen (e.g. *s0 - s15:c0.c1023*).
- These components are discussed in the **Security Levels** section.

However note that:

1. Access decisions regarding a subject make use of all the components of the **security context**.
2. Access decisions regarding an object make use of the components as follows:
   1. the user is either set to a special user called *system_u*[2] or it is set to the SELinux user id of the creating process. It is possible to add constraints on users within policy based on their object class (an example of this is the Reference Policy UBAC (User Based Access Control) option.
   2. the role is generally set to a special SELinux internal role of *object_r*, although policy version 26 with kernel 2.6.39 and above do support role transitions on any object class. It is then possible to add constraints on the role within policy based on their object class.

The **Computing Security Contexts** section describes how SELinux computes the security context components based on a *source context*, *target context* and object *class*.

The examples below show security contexts for processes, directories and files (note that the policy did not support MCS or MLS, therefore no *level* field):

Example Process Security Context:

```
# These are process security contexts taken from a ps -Z command
# (edited for clarity) that show four processes:

LABEL                                       PID  TTY   CMD
unconfined_u:unconfined_r:unconfined_t      2539 pts/0 bash
unconfined_u:message_filter_r:ext_gateway_t 3134 pts/0 secure_server
unconfined_u:message_filter_r:int_gateway_t 3138 pts/0 secure_server
unconfined_u:unconfined_r:unconfined_t      3146 pts/0 ps

# Note the bash and ps processes are running under the
# unconfined_t domain, however the secure_server has two instances
# running under two different domains (ext_gateway_t and
# int_gateway_t). Also note that they are using the
# message_filter_r role whereas bash and ps use unconfined_r.
#
# These results were obtained by running the system in permissive mode.
```

Example Object Security Context:

```
# These are the message queue directory object security contexts
# taken from an ls -Zd command (edited for clarity):
system_u:object_r:in_queue_t /usr/message_queue/in_queue
system_u:object_r:out_queue_t /usr/message_queue/out_queue

# Note that they are instantiated with system_u and object_r
```

```
# These are the message queue file object security contexts
# taken from an ls -Z command (edited for clarity):
/usr/message_queue/in_queue:
unconfined_u:object_r:in_file_t Message-1
unconfined_u:object_r:in_file_t Message-2
/usr/message_queue/out_queue:
unconfined_u:object_r:out_file_t Message-10
unconfined_u:object_r:out_file_t Message-11

# Note that they are instantiated with unconfined_u as that was
# the SELinux user id of the process that created the files
# (see the process example above). The role remained as object_r.
```

Subjects

A subject is an active entity generally in the form of a person, process, or device that causes information to flow among objects or changes the system state.

Within SELinux a subject is an active process and has a **security context** associated with it, however a process can also be referred to as an object depending on the context in which it is being taken, for example:

1. A running process (i.e. an active entity) is a subject because it causes information to flow among objects or can change the system state.
2. The process can also be referred to as an object because each process has an associated object class[3] called ***process***. This process 'object', defines what permissions the policy is allowed to grant or deny on the active process.

An example is given of the above scenarios in the **Allowing a Process Access to Resources** section.

In SELinux subjects can be:

Trusted - Generally these are commands, applications etc. that have been written or modified to support specific SELinux functionality to enforce the security policy (e.g. the kernel, init, pam, xinetd and login). However, it can also cover any application that the organisation is willing to trust as a part of the overall system. Although (depending on your paranoia level), the best policy is to trust nothing until it has been verified that it conforms to the security policy. Generally these trusted applications would run in either their own domain (e.g. the audit daemon could run under auditd_t) or grouped together (e.g. the ***semanage****(8)* and ***semodule****(8)* commands could be grouped under *semanage_t*).

Untrusted - Everything else.

Objects

Within SELinux an object is a resource such as files, sockets, pipes or network interfaces that are accessed via processes (also known as subjects). These objects are classified according to the resource they provide with access permissions relevant to their purpose (e.g. read, receive and write), and assigned a **Security Context** as described in the following sections.

Object Classes and Permissions

Each object consists of a class identifier that defines its purpose (e.g. file, socket) along with a set of permissions (also known as Access Vectors (AV)) that describe what services the object can handle (read, write, send etc.). When an object is instantiated it will be allocated a name (e.g. a file could be called config or a socket my_connection) and a security context (e.g. *system_u:object_r:selinux_config_t*) as shown in **Figure 5: Object Class = 'file' and permissions**.

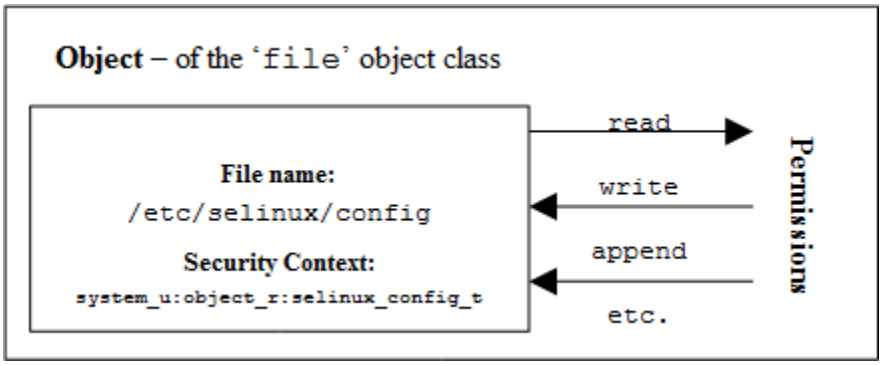


Figure 5: Object Class = 'file' and permissions - *the policy rules would define those permissions allowed for each process that needs access to the /etc/selinux/config file.*

The objective of the policy is to enable the user of the object (the subject) access to the minimum permissions needed to complete the task (i.e. do not allow write permission if only reading information).

These object classes and their associated permissions are built into the GNU / Linux kernel and user space object managers by developers and are therefore not generally updated by policy writers.

The object classes consist of kernel object classes (for handling files, sockets etc.) plus userspace object classes for userspace object managers (for services such as X-Windows or dbus). The number of object classes and their permissions can vary depending on the features configured in the GNU / Linux release. All the known object classes and permissions are described in **Appendix A - Object Classes and Permissions**.

Allowing a Process Access to Resources

This is a simple example that attempts to explain two points:

1. How a process is given permission to use an objects resource.
2. By using the *process* object class, show that a process can be described as a process or object.

An SELinux policy contains many rules and statements, the majority of which are *allow* rules that (simply) allows processes to be given access permissions to an objects resources.

The following allow rule and **Figure 6: The *allow* rule** illustrates 'a process can also be an object' as it allows processes running in the *unconfined_t* domain, permission to *transition* the external gateway application to the *ext_gateway_t* domain once it has been executed:

```
allow Rule | source_domain |  target_type  :  class  | permission
-----------▼---------------▼-------------------------▼------------
allow        unconfined_t    ext_gateway_t :  process  transition;
```

Where:

allow

- The SELinux language *allow* rule.

unconfined_t

- The source domain (or subject) identifier - in this case the shell that wants to *exec* the gateway application.

ext_gateway_t

- The target object identifier - the object instance of the gateway application process.

process

- The target object class - the *process* object class.

transition

- The permission granted to the source domain on the targets object - in this case the *unconfined_t* domain has transition permission on the *ext_gateway_t process* object.

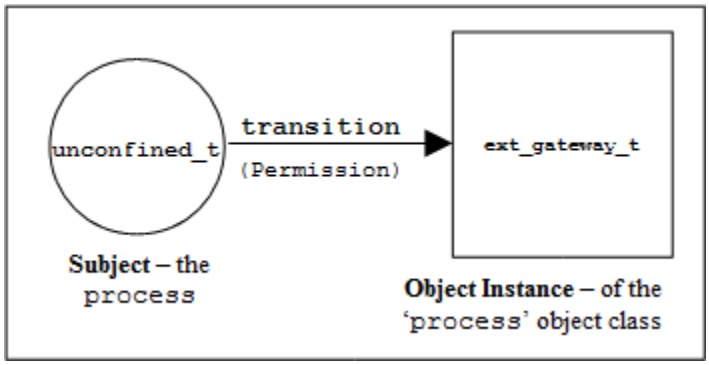


Figure 6: The *allow* rule - *Showing that the subject (the processes running in the unconfined_t domain) has been given the transition permission on the ext_gateway_t process object.*

It should be noted that there is more to a domain transition than described above, for a more detailed explanation, see the **Domain Transition** section.

Labeling Objects

Within a running SELinux enabled GNU / Linux system the labeling of objects is managed by the system and generally unseen by the users (until labeling goes wrong !!). As processes and objects are created and destroyed, they either:

1. Inherit their labels from the parent process or object. The policy default user, type, role and range statements can be used to change the behavior as discussed in the **Default Rules** section.
2. The policy type, role and range transition statements allow a different label to be assigned as discussed in the **Domain and Object Transitions** section.
3. SELinux-aware applications can assign a new label (with the policy's approval of course) using the **libselinux** API functions. The *process { setfscreate }* permission can be used to allow subjects to create files with a new label programmatically using the ***setfscreatecon**(3)* function, overriding default rules and transition statements.
4. An object manager (OM) can enforce a default label that can either be built into the OM or obtained via a configuration file (such as those used by **SELinux X-Windows Support**.
5. Use an '**initial security identifier**' (or initial SID). These are defined in all policies and are used to either set an initial context during the boot process, or if an object requires a default (i.e. the object does not already have a valid context).

The **Computing Security Contexts** section gives detail on how some of the kernel based objects are computed.

The SELinux policy language supports object labeling statements for file and network services that are defined in the **File System Labeling Statements** and **Network Labeling Statements** sections.

An overview of the process required for labeling filesystems that use extended attributes (such as ext3 and ext4) is discussed in the **Labeling Extended Attribute Filesystems** section.

Labeling Extended Attribute Filesystems

The labeling of file systems that implement extended attributes[4] is supported by SELinux using:

1. The *fs_use_xattr* statement within the policy to identify what filesystems use extended attributes. This statement is used to inform the security server how to label the filesystem.
2. A 'file contexts' file that defines what the initial contexts should be for each file and directory within the filesystem. The format of this file and how it is built from source policy is described in the **Policy Store Configuration Files - Building the File Labeling Support Files**[5] section.
3. A method to initialise the filesystem with these extended attributes. This is achieved by SELinux utilities such as ***fixfiles**(8)* and ***setfiles**(8)*. There are also commands such as ***chcon**(1)*, ***restorecon**(8)* and ***restorecond**(8)* that can be used to relabel files.

Extended attributes containing the SELinux context of a file can be viewed by the *ls -Z* or ***getfattr**(1)* commands as follows:

```
ls -Z myfile
-rw-r--r-- rch rch unconfined_u:object_r:user_home:s0 myfile
```

```
getfattr -n security.selinux myfile
# file_name: myfile
security.selinux="unconfined_u:object_r:user_home:s0

# Where -n security.selinux is the name of the extended
# attribute and 'myfile' is a file name. The security context
# (or label) held for the file is displayed.
```

Copying and Moving Files

Assuming that the correct permissions have been granted by the policy, the effects on the security context of a file when copied or moved differ as follows:

- copy a file - takes on label of new directory.
- move a file - retains the label of the file.

However, if the ***restorecond****(8)* daemon is running and the **restorecond.conf** file is correctly configured, then other security contexts can be associated to the file as it is moved or copied (provided it is a valid context and specified in the **file_contexts** file). Note that there is also the ***install****(1)* command that supports a *-Z* option to specify the target context.

The examples below show the effects of copying and moving files:

```
# These are the test files in the /root directory and their current security
# context:
#
-rw-r--r-- root root unconfined_u:object_r:unconfined_t copied-file
-rw-r--r-- root root unconfined_u:object_r:unconfined_t moved-file

# These are the commands used to copy / move the files:
# Standard copy file:
cp copied-file /usr/message_queue/in_queue

# Standard move file:
mv moved-file /usr/message_queue/in_queue

# The target directory (/usr/message_queue/in_queue) is labeled "in_queue_t".
# The results of "ls -Z" on the target directory are:
#
-rw-r--r-- root root unconfined_u:object_r:in_queue_t copied-file
-rw-r--r-- root root unconfined_u:object_r:unconfined_t moved-file
```

However, if the *restorecond* daemon is running:

```
# If the restorecond daemon is running with a restorecond.conf file entry of:
#
/usr/message_queue/in_queue/*

# AND the file_context file has an entry of:
#
/usr/message_queue/in_queue(/.*)? -- system_u:object_r:in_file_t

# Then all the entries would be set as follows when the daemon detects
# the files creation:
#
-rw-r--r-- root root unconfined_u:object_r:in_file_t copied-file
-rw-r--r-- root root unconfined_u:object_r:in_file_t moved-file

# This is because the restorecond process will set the contexts defined
in the file_contexts file to the context specified as it is created in
# the new directory.
```

This is because the *restorecond* process will set the contexts defined in the file_contexts file to the context specified as it is created in the new directory.

Labeling Subjects

On a running GNU / Linux system, processes inherit the security context of the parent process. If the new process being spawned has permission to change its context, then a 'type transition' is allowed that is discussed in the **Domain Transition** section. The policy language supports a number of statements to assign components to security contexts such as:

- *user*, *role* and *type* statements.

and to manage their scope:

- *role_allow* and *constrain*

and to manage their transition:

- *type_transition*, *role_transition* and *range_transition*

SELinux-aware applications can assign a new label (with the policy's approval of course) using the **libselinux** API functions. The *process { setexec setkeycreate setsockcreate }* permissions can be used to allow subjects to label processes, kernel keyrings, and sockets programmatically using the ***setexec**(3)*, ***setkeycreatecon**(3)* and ***setsockcreatecon**(3)* functions respectively, overriding transition statements.

The *kernel* **initial security identifier** is used to associate a specified label with kernel objects, including kernel threads (both those that are created during initialization but also kernel threads created later), kernel-private sockets and synthetic objects representing kernel resources (e.g. the "system" class).

It is true that processes created prior to initial policy load will also be in the kernel SID until/unless there is a policy loaded and either a policy-defined transition or an explicit setcon or setexeccon+execve, but that's just the typical default inheritance from creating task behavior for processes.

The context associated with the *unlabeled* **initial security identifier** is used as the fallback context for both subjects and objects when their label is invalidated by a policy reload (their SID is unchanged but the SID is transparently remapped to the unlabeled context). It is also assigned as the initial state for various objects e.g. inodes, superblocks, etc until they reach a point where a more specific label can be determined e.g. from an xattr or from policy.

Object Reuse

As GNU / Linux runs it creates instances of objects and manages the information they contain (read, write, modify etc.) under the control of processes, and at some stage these objects may be deleted or released allowing the resource (such as memory blocks and disk space) to be available for reuse.

GNU / Linux handles object reuse by ensuring that when a resource is re-allocated it is cleared. This means that when a process releases an object instance (e.g. release allocated memory back to the pool, delete a directory entry or file), there may be information left behind that could prove useful if harvested. If this should be an issue, then the process itself should clear or shred the information before releasing the object (which can be difficult in some cases unless the source code is available).

Computing Security Contexts

SELinux uses a number of policy language statements and *libselinux* functions to compute a security context via the kernel security server.

When security contexts are computed, the different kernel, userspace tools and policy versions can influence the outcome. This is because patches have been applied over the years that give greater flexibility in computing contexts. For example a 2.6.39 kernel with SELinux userspace services supporting policy version 26 can influence the computed role.

The security context is computed for an object using the following components: a source context, a target context and an object class.

The *libselinux* userspace functions used to compute a security context are:

- ***avc_compute_create****(3)* and ***security_compute_create****(3)*
- ***avc_compute_member****(3)* and ***security_compute_member****(3)*
- ***security_compute_relabel****(3)*

Note that these *libselinux* functions actually call the kernel equivalent functions in the security server (see kernel source *security/selinux/ss/services.c*: *security_compute_sid*, *security_member_sid* and *security_change_sid*) that actually compute the security context.

The kernel policy language statements that influence a computed security context are:

type_transition, *role_transition*, *range_transition*, *type_member* and *type_change*, *default_user*, *default_role*, *default_type* and *default_range* statements (their corresponding CIL statements).

The sections that follow give an overview of how security contexts are computed for some kernel classes and also when using the userspace *libselinux* functions.

Security Context Computation for Kernel Objects

Using a combination of the email thread: http://www.spinics.net/lists/selinux/msg10746.html and kernel 3.14 source, this is how contexts are computed by the security server for various kernel objects (also see the **Linux Security Module and SELinux** section.

Process

The initial task starts with the kernel security context, but the “init” process will typically transition into its own unique context (e.g. *init_t*) when the init binary is executed after the policy has been loaded. Some init programs re-exec themselves after loading policy, while in other cases the initial policy load is performed by the *initrd*/*initramfs* script prior to mounting the real *root* and executing the real *init* program.

Processes inherit their security context as follows:

1. On fork a process inherits the security context of its creator/parent.
2. On *exec*, a process may transition to another security context based on policy statements: *type_transition*, *range_transition*, *role_transition* (policy version 26), *default_user*, *default_role*, *default_range* (policy versions 27) and *default_type* (policy version 28) or if a security-aware process, by calling ***setexeccon**(3)* if permitted by policy prior to invoking exec.
3. If the loaded SELinux policy has the nnp_nosuid_transition policy capability enabled there are potentially two additional permissions that are required to permit a domain transition: nosuid_transition for nosuid mounted filesystems, and nnp_transition for threads with the no_new_privs flag. If the nnp_nosuid_transition policy capability is disabled, such domain transitions are denied but bounded domain transitions are still allowed. In bounded transitions, the target domain is only allowed a subset of the permissions of the source domain. See also **Linux Security Module and SELinux**.
4. At any time, a security-aware process may invoke ***setcon**(3)* to switch its security context (if permitted by policy) although this practice is generally discouraged - exec-based transitions are preferred.

Files

The default behavior for labeling files (actually inodes that consist of the following classes: files, symbolic links, directories, socket files, fifo’s and block/character) upon creation for any filesystem type that supports labeling is as follows:

1. The user component is inherited from the creating process (policy version 27 allows a *default_user* of source or target to be defined for each object class).
2. The role component generally defaults to the *object_r* role (policy version 26 allows a *role_transition* and version 27 allows a *default_role* of source or target to be defined for each object class).
3. The type component defaults to the type of the parent directory if no matching *type_transition* rule was specified in the policy (policy version 25 allows a filename *type_transition* rule and version 28 allows a *default_type* of source or target to be defined for each object class).
4. The *range*/*level* component defaults to the low/current level of the creating process if no matching *range_transition* rule was specified in the policy (policy version 27 allows a *default_range* of source or target with the selected range being low, high or low-high to be defined for each object class).

Security-aware applications can override this default behavior by calling ***setfscreatecon**(3)* prior to creating the file, if permitted by policy.

For existing files the label is determined from the *xattr* value associated with the file. If there is no *xattr* value set on the file, then the file is treated as being labeled with the default file security context for the filesystem. By default, this is the “*file*” initial SID, which is mapped to a context by the policy. This default may be overridden via the *defcontext=* mount option on a per-mount basis as described in ***mount**(8)*.

File Descriptors

Inherits the label of its creator/parent.

Filesystems

Filesystems are labeled using the appropriate *fs_use* kernel policy language statement as they are mounted, they are based on the *filesystem* type name (e.g. *ext4*) and their behaviour (e.g. *xattr*). For example if the policy specifies the following:

```
fs_use_task pipefs system_u:object_r:fs_t:s0
```

then as the *pipefs* filesystem is being mounted, the SELinux LSM security hook *selinux_set_mnt_opts* will call *security_fs_use* that will:

- Look for the filesystem name within the policy (*pipefs*)
- If present, obtain its behaviour (*fs_use_task*)
- Then obtain the allocated security context (*system_u:object_r:fs_t:s0*)

Should the behaviour be defined as *fs_use_task*, then the filesystem will be labeled as follows:

1. The user component is inherited from the creating process (policy version 27 allows a *default_user* of source or target to be defined).
2. The role component generally defaults to the *object_r* role (policy version 26 allows a *role_transition* and version 27 allows a *default_role* of source or target to be defined).
3. The type component defaults to the type of the target type if no matching *type_transition* rule was specified in the policy (policy version 28 allows a *default_type* of source or target to be defined).
4. The *range*/*level* component defaults to the low/current level of the creating process if no matching *range_transition* rule was specified in the policy (policy version 27 allows a *default_range* of source or target with the selected range being *low*, *high* or *low-high* to be defined).

Notes:

1. Filesystems that support *xattr* extended attributes can be identified via the mount command as there will be a '*seclabel*' keyword present.
2. There are mount options for allocating various context types: *context=*, *fscontext=*, *defcontext=* and *rootcontext=*. They are fully described in the ***mount**(8)* man page.

Network File System (nfsv4.2)

If labeled NFS is implemented with *xattr* support, then the creation of inodes are treated as described in the Files section.

INET Sockets

If a socket is created by the ***socket**(3)* call they are labeled as follows:

1. The user component is inherited from the creating process (policy version 27 allows a *default_user* of source or target to be defined for each socket object class).
2. The role component is inherited from the creating process (policy version 26 allows a *role_transition* and version 27 allows a *default_role* of source or target to be defined for each socket object class).
3. The type component is inherited from the creating process if no matching *type_transition* rule was specified in the policy and version 28 allows a *default_type* of source or target to be defined for each socket object class).
4. The *range*/*level* component is inherited from the creating process if no matching *range_transition* rule was specified in the policy (policy version 27 allows a *default_range* of source or target with the selected range being *low*, *high* or *low-high* to be defined for each socket object class).

Security-aware applications may use ***setsockcreatecon**(3)* to explicitly label sockets they create if permitted by policy.

If created by a connection they are labeled with the context of the listening process.

Some sockets may be labeled with the kernel SID to reflect the fact that they are kernel-internal sockets that are not directly exposed to applications.

IPC

Inherits the label of its creator/parent.

Message Queues

Inherits the label of its sending process. However if sending a message that is unlabeled, compute a new label based on the current process and the message queue it will be stored in as follows:

1. The user component is inherited from the sending process (policy version 27 allows a *default_user* of source or target to be defined for the message object class).
2. The role component is inherited from the sending process (policy version 26 allows a *role_transition* and version 27 allows a *default_role* of source or target to be defined for the message object class).
3. The type component is inherited from the sending process if no matching *type_transition* rule was specified in the policy and version 28 allows a *default_type* of source or target to be defined for the message object class).
4. The *range*/*level* component is inherited from the sending process if no matching *range_transition* rule was specified in the policy (policy version 27 allows a *default_range* of source or target with the selected range being *low*, *high* or *low-high* to be defined for the message object class).

Semaphores

Inherits the label of its creator/parent.

Shared Memory

Inherits the label of its creator/parent.

Keys

Inherits the label of its creator/parent.

Security-aware applications may use ***setkeycreatecon****(3)* to explicitly label keys they create if permitted by policy.

Using libselinux Functions

avc_compute_create and *security_compute_create*

The table below shows how the components from the source context *scon*, target context *tcon* and class *tclass* are used to compute the new context *newcon* (referenced by SIDs for ***avc_compute_create****(3)*). The following notes also apply:

1. Any valid policy ***role_transition***, ***type_transition*** and ***range_transition*** enforcement rules will influence the final outcome as shown.
2. For kernels less than 2.6.39 the context generated will depend on whether the class is *process* or any other class.
3. For kernels 2.6.39 and above the following also applies:
    - Those classes suffixed by *socket* will also be included in the *process* class outcome.
    - If a valid *role_transition* rule for *tclass*, then use that instead of the default *object_r*. Also requires policy version 26 or greater - see ***security_policyvers****(3)*.
    - If the *type_transition* rule is classed as the 'file name transition rule' (i.e. it has an *object_name* parameter), then provided the object name in the rule matches the last component of the objects name (in this case a file or directory name), then use the rules *default_type*. Also requires policy version 25 or greater.
4. For kernels 3.5 and above with policy version 27 or greater, the *default_user*, *default_role*, *default_range* statements will influence the *user*, *role* and *range* of the computed context for the specified class *tclass*.

With policy version 28 or greater the *default_type* statement can also influence the *type* in the computed context.

Computing avc_compute_create(3) and security_compute_create(3) contexts:

- ***user***
  - IF kernel >= 3.5 with a *default_user tclass target* rule then use *tcon user*
  - ELSE
  - Use *scon user*
- ***role***
  - IF kernel >=2.6.39, and there is a valid *role_transition* rule then use the rules ***new_role***
  - OR
  - IF kernel >= 3.5 with *default_role tclass source* rule then use *scon role*
  - OR
  - IF kernel >= 3.5 with *default_role tclass target* rule then use *tcon role*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket*, then use *scon role*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon role*
  - ELSE
  - Use *object_r*
- ***type***
  - IF there is a valid *type_transition* rule then use the rules ***default_type***
  - OR
  - IF kernel >= 3.5 with *default_type tclass source* rule then use *scon type*
  - OR
  - IF kernel >= 3.5 with *default_type tclass target* rule then use *tcon type*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon type*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon type*
  - **ELSE**
  - Use *tcon type*
- ***range***
  - IF there is a valid *range_transition* rule then use the rules ***new_range***
  - OR
  - IF kernel >= 3.5 with *default_range tclass source low* rule then use *scon low*
  - OR
  - IF kernel >= 3.5 with *default_range tclass source high* rule then use *scon high*
  - OR
  - IF kernel >= 3.5 with *default_range tclass source low_high* rule then use *scon range*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target low* rule then use *tcon low*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target high* rule then use *tcon high*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target low_high* rule then use *tcon range*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon range*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon range*
  - **ELSE**
  - Use *scon low*

avc_compute_member and *security_compute_member*

The table below shows how the components from the source context, *scon* target context, *tcon* and class, *tclass* are used to compute the new context *newcon* (referenced by SIDs for ***avc_compute_member**(3)*). The following notes also apply:

1. Any valid policy ***type_member*** enforcement rules will influence the final outcome as shown.
2. For kernels less than 2.6.39 the context generated will depend on whether the class is *process* or any other class.
3. For kernels 2.6.39 and above, those classes suffixed by *socket* are also included in the *process* class outcome.
4. For kernels 3.5 and above with policy version 28 or greater, the *default_role*, *default_range* statements will influence the *role* and *range* of the computed context for the specified class *tclass*. With policy version 28 or greater the *default_type* statement can also influence the *type* in the computed context.

Computing avc_compute_member(3) and security_compute_member(3) contexts:

- ***user***
  - Always uses *tcon user*
- ***role***
  - IF kernel >= 3.5 with *default_role tclass source* rule then use *scon role*
  - OR
  - IF kernel >= 3.5 with *default_role tclass target* rule then use *tcon role*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon role*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon role*
  - ELSE
  - Use *object_r*
- ***type***
  - IF there is a valid *type_member* rule then use the rules ***member_type***
  - OR
  - IF kernel >= 3.5 with *default_type tclass source* rule then use *scon type*
  - OR
  - IF kernel >= 3.5 with *default_type tclass target* rule then use *tcon type*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon type*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon type*
  - ELSE
  - Use *tcon type*
- ***range***
  - IF kernel >= 3.5 with *default_range tclass source low* rule then use *scon low*
  - OR
  - IF kernel >= 3.5 with *default_range tclass source high* rule then use *scon high*
  - OR
  - IF kernel >= 3.5 with *default_range tclass source low_high* rule then use *scon range*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target low* rule then use *tcon low*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target high* rule then use *tcon high*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target low_high* rule then use *tcon range*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon range*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon range*

- ELSE
- Use *scon low*

security_compute_relabel

The table below shows how the components from the source context, *scon* target context, *tcon* and class, *tclass* are used to compute the new context *newcon* for ***security_compute_relabel****(3)*. The following notes also apply:

1. Any valid policy ***type_change*** enforcement rules will influence the final outcome shown in the table.
2. For kernels less than 2.6.39 the context generated will depend on whether the class is *process* or any other class.
3. For kernels 2.6.39 and above, those classes suffixed by *socket* are also included in the *process* class outcome.
4. For kernels 3.5 and above with policy version 28 or greater, the *default_user*, *default_role*, *default_range* statements will influence the *user*, *role* and *range* of the computed context for the specified class *tclass*. With policy version 28 or greater the *default_type* statement can also influence the *type* in the computed context.

Computing security_compute_relabel(3) contexts:

- ***user***
  - If kernel >= 3.5 with a *default_user tclass target* rule then use *tcon user*
  - ELSE
  - Use *scon user*
- ***role***
  - IF kernel >= 3.5 with *default_role tclass source* rule then use *scon role*
  - OR
  - IF kernel >= 3.5 with *default_role tclass target* rule then use *tcon role*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon role*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon role*
  - ELSE
  - Use *object_r*
- ***type***
  - IF there is a valid *type_change* rule then use the rules ***change_type***
  - OR
  - IF kernel >= 3.5 with *default_type tclass source* rule then use *scon type*
  - OR
  - IF kernel >= 3.5 with *default_type tclass target* rule then use *tcon type*
  - OR
  - IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon type*
  - OR
  - IF kernel <= 2.6.38 and *tclass* is *process* then use *scon type*
  - ELSE
  - Use *tcon type*
- ***range***
  - IF kernel >= 3.5 with *default_range tclass source low* rule then use *scon low*
  - OR
  - IF kernel >= 3.5 with *default_range tclass source high* rule then use *scon high*
  - OR
  - IF kernel >= 3.5 with *default_range tclass source low_high* rule then use *scon range*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target low* rule then use *tcon low*
  - OR
  - IF kernel >= 3.5 with *default_range tclass target high* rule then use *tcon high*
  - OR

- IF kernel >= 3.5 with *default_range tclass target low_high* rule then use *tcon range*
- OR
- IF kernel >= 2.6.39 and *tclass* is *process* or **socket* then use *scon range*
- OR
- IF kernel <= 2.6.38 and *tclass* is *process* then use *scon range*
- ELSE
- Use *scon low*

Computing Access Decisions

There are a number of ways to compute access decisions within userspace SELinux-aware applications or object managers:

1. Use of the ***selinux_check_access**(3)* function is the recommended option. This utilises the AVC services discussed in bullet 3 in a single call that:
    - Dynamically resolves class and permissions strings to their class/permission values using ***string_to_security_class**(3)* and ***string_to_av_perm**(3)* with ***security_deny_unknown**(3)* to handle unknown classes/permissions.
    - Uses ***avc_has_perm**(3)* to check whether the decision is cached before calling ***security_compute_av_flags**(3)* (and caching the result), checks enforcing mode (both global and per-domain (permissive)), and logs any denials (there is also an option to add supplemental auditing information that is handled as described in ***avc_audit**(3)*.
2. Use functions that do not cache access decisions (i.e. they do not use the *libselinux* AVC services). These require a call to the kernel for every decision using ***security_compute_av**(3)* or ***security_compute_av_flags**(3)*. The ***avc_netlink_ *(3)*** functions can be used to detect policy change events. Auditing would need to be implemented if required.
3. Use functions that utilise the *libselinux* userspace AVC services that are initialised with ***avc_open**(3)*. These can be built in various configurations such as:
    - Using the default single threaded mode where ***avc_has_perm**(3)* will automatically cache entries, audit the decision and manage the handling of policy change events.
    - Implementing threads or a similar service that will handle policy change events and auditing in real time with ***avc_has_perm**(3)* or ***avc_has_perm_noaudit**(3)* handling decisions and caching. This has the advantage of better performance, which can be further increased by caching the entry reference.
4. Implement custom caching services with ***security_compute_av**(3)* or ***security_compute_av_flags**(3)* for computing access decisions. The ***avc_netlink_ *(3)*** functions can then be used to detect policy change events. Auditing would need to be implemented if required.

Where performance is important when making policy decisions, then the ***selinux_status_open**(3)*, ***selinux_status_updated**(3)*, ***selinux_status_getenforce**(3)*, ***selinux_status_policyload**(3)* and ***selinux_status_close**(3)* functions could be used to detect policy updates etc. as these do not require kernel system call over-heads once set up. Note that these functions are only available from *libselinux* 2.0.99, with Linux kernel 2.6.37 and above.

Domain and Object Transitions

This section discusses the *type_transition* statement that is used to:

- Transition a process from one domain to another (a domain transition).
- Transition an object from one type to another (an object transition).

These transitions can also be achieved using the **libselinux** API functions for SELinux-aware applications.

Domain Transition

A domain transition is where a process in one domain starts a new process in another domain under a different security context. There are two ways a process can define a domain transition:

- Using a *type_transition* statement, where the ***exec**(2)* system call will automatically perform a domain transition for programs that are not themselves SELinux-aware. This is the most common method and would be in the form of the following statement:

```
type_transition unconfined_t secure_services_exec_t : process ext_gateway_t;
```

- SELinux-aware applications can specify the domain of the new process using the **libselinux** API call ***setexeccon**(3)*. To achieve this the SELinux-aware application must also have the setexec permission, for example:

```
allow crond_t self:process setexec;
```

However, before any domain transition can take place the policy must specify that:

1. The source *domain* has permission to *transition* into the target domain.
2. The application binary file needs to be *executable* in the source domain.
3. The application binary file needs an *entry point* into the target domain.

The following is a *type_transition* statement taken an example loadable module message filter *ext_gateway.conf* that will be used to explain the transition process:

```
type_transition | source_domain |      target_type     |  class | target_domain;
----------------▼---------------▼----------------------▼--------▼----------------
type_transition   unconfined_t   secure_services_exec_t:process   ext_gateway_t;
```

This *type_transition* statement states that when a *process* running in the *unconfined_t* domain (the source domain) executes a file labeled *secure_services_exec_t*, the *process* should be changed to *ext_gateway_t* (the target domain) if allowed by the policy (i.e. transition from the *unconfined_t* domain to the *ext_gateway_t* domain).

However as stated above, to be able to *transition* to the *ext_gateway_t* domain, the following minimum permissions must be granted in the policy using *allow* rules, where (the bullet numbers correspond to those in **Figure 7: Domain Transition**):

(1) The *domain* needs permission to *transition* into the *ext_gateway_t* (target) domain:

```
allow unconfined_t ext_gateway_t : process transition;
```

(2) The executable file needs to be *executable* in the *unconfined_t* (source) domain, and therefore also requires that the file is readable:

```
allow unconfined_t secure_services_exec_t : file { execute read getattr };
```

(3) The executable file needs an *entry point* into the *ext_gateway_t* (target) domain:

```
allow ext_gateway_t secure_services_exec_t : file entrypoint;
```

These are shown in **Figure 7: Domain Transition** where *unconfined_t* forks a child process, that then exec's the new program into a new domain called *ext_gateway_t*. Note that because the *type_transition* statement is being used, the transition is automatically carried out by the SELinux enabled kernel.

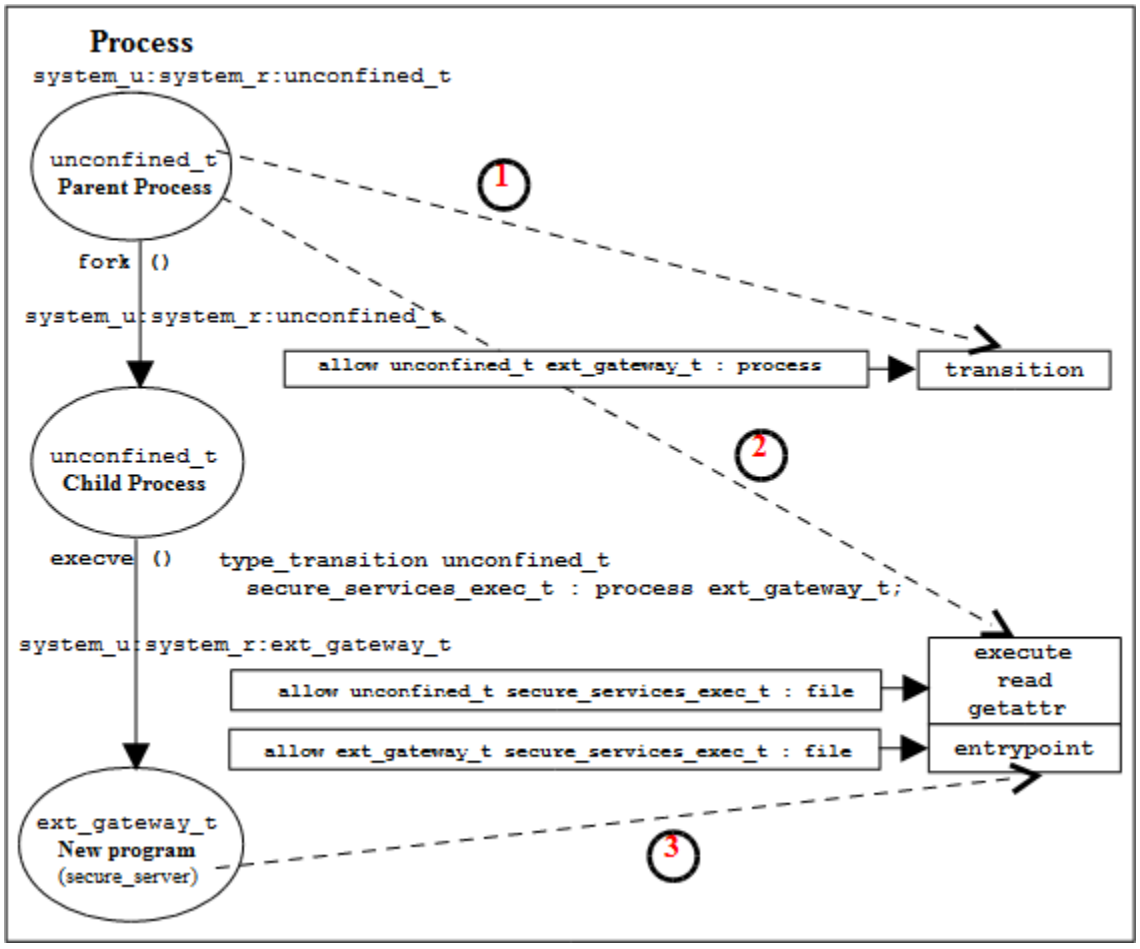


Figure 7: Domain Transition - *Where the secure_server is executed within the unconfined_t domain and then transitioned to the ext_gateway_t domain.*

Type Enforcement Rules

When building the *ext_gateway.conf* and *int_gateway.conf* modules the intention was to have both of these transition to their respective domains via *type_transition* statements. The *ext_gateway_t* statement would be:

```
type_transition unconfined_t secure_services_exec_t : process ext_gateway_t;
```

and the *int_gateway_t* statement would be:

```
type_transition unconfined_t secure_services_exec_t : process int_gateway_t;
```

However, when linking these two loadable modules into the policy, the following error was given:

```
semodule -v -s modular-test -i int_gateway.pp -i ext_gateway.pp
Attempting to install module 'int_gateway.pp':
Ok: return value of 0.
Attempting to install module 'ext_gateway.pp':
Ok: return value of 0.
Committing changes:
libsepol.expand_terule_helper: conflicting TE rule for (unconfined_t,
secure_services_exec_t:process): old was ext_gateway_t, new is int_gateway_t
libsepol.expand_module: Error during expand
libsemanage.semanage_expand_sandbox: Expand module failed
semodule: Failed!
```

This happened because the type enforcement rules will only allow a single 'default' type for a given source and target (see the **Type Statements** section). In the above case there were two *type_transition* statements with the same source and target, but different default domains. The *ext_gateway.conf* module had the following statements:

```
# Allow the client/server to transition for the gateways:

allow unconfined_t ext_gateway_t:process { transition };
allow unconfined_t secure_services_exec_t:file { read execute getattr };
allow ext_gateway_t secure_services_exec_t:file { entrypoint };
type_transition unconfined_t secure_services_exec_t:process ext_gateway_t;
```

And the *int_gateway.conf* module had the following statements:

```
# Allow the client/server to transition for the gateways:

allow unconfined_t int_gateway_t:process { transition };
allow unconfined_t secure_services_exec_t:file { read execute getattr };
allow int_gateway_t secure_services_exec_t:file { entrypoint };
type_transition unconfined_t secure_services_exec_t:process int_gateway_t;
```

While the allow rules are valid to enable the transitions to proceed, the two *type_transition* statements had different 'default' types (or target domains), that breaks the type enforcement rule.

It was decided to resolve this by:

- Keeping the *type_transition* rule for the 'default' type of *ext_gateway_t* and allow the secure server process to be exec'd from *unconfined_t* as shown in **Figure 7: Domain Transition**, by simply running the command from the prompt as follows:

```
# Run the external gateway 'secure server' application on port 9999 and
# let the policy transition the process to the ext_gateway_t domain:

secure_server 99999
```

- Use the SELinux ***runcon**(1)* command to ensure that the internal gateway runs in the correct domain by running *runcon* from the prompt as follows:

```
# Run the internal gateway 'secure server' application on port 1111 and
# use runcon to transition the process to the int_gateway_t domain:

runcon -t int_gateway_t -r message_filter_r secure_server 1111

# Note: The role is required as a role transition is defined in the policy.
```

The ***runcon**(1)* command makes use of a number of **libselinux** API functions to check the current context and set up the new context (for example ***getfilecon**(3)* is used to get the executable files context and ***setexeccon**(3)* is used to set the new process context). If all contexts are correct, then the ***execvp**(2)* system call is executed that exec's the secure_server application with the argument of '1111' into the *int_gateway_t* domain with the *message_filter_r* role. The *runcon* source can be found in the coreutils package.

Other ways to resolve this issue are:

1. Use the *runcon* command for both gateways to transition to their respective domains. The *type_transition* statements are therefore not required.
2. Use different names for the secure server executable files and ensure they have a different type (i.e. instead of *secure_service_exec_t* label the external gateway *ext_gateway_exec_t* and the internal gateway *int_gateway_exec_t*. This would involve making a copy of the application binary (which has already been done as part of the **module testing** by calling the server 'server' and labeling it *unconfined_t* and then making a copy called secure_server and labeling it *secure_services_exec_t*).
3. Implement the policy using the Reference Policy utilising the template interface principles discussed in the **template** section.

It was decided to use *runcon* as it demonstrates the command usage better than reading the man pages.

Object Transition

An object transition is where a new object requires a different label to that of its parent. For example a file is being created that requires a different label to that of its parent directory. This can be achieved automatically using a *type_transition* statement as follows:

```
type_transition ext_gateway_t in_queue_t:file in_file_t;
```

The following details an object transition used in an example *ext_gateway.conf* loadable module where by default, files would be labeled *in_queue_t* when created by the gateway application as this is the label attached to the parent directory as shown:

```
ls -Za /usr/message_queue/in_queue
drwxr-xr-x root root unconfined_u:object_r:in_queue_t .
drwxr-xr-x root root system_u:object_r:unconfined_t ..
```

However the requirement is that files in the *in_queue* directory must be labeled *in_file_t*. To achieve this the files created must be relabeled to *in_file_t* by using a *type_transition* rule as follows:

```
type_transition | source_domain |  target_type :    object
----------------▼---------------▼--------------▼-----------------
type_transition   ext_gateway_t    in_queue_t  : file in_file_t;
```

This *type_transition* statement states that when a *process* running in the *ext_gateway_t* domain (the source domain) wants to create a *file* object in the directory that is labeled *in_queue_t*, the file should be relabeled *in_file_t* if allowed by the policy (i.e. label the file *in_file_t*).

However as stated above to be able to create the file, the following minimum permissions need to be granted in the policy using *allow* rules, where:

- The source domain needs permission to *add file entries into the directory*:

```
allow ext_gateway_t in_queue_t : dir { write search add_name };
```

- The source domain needs permission to *create file entries*:

```
allow ext_gateway_t in_file_t : file { write create getattr };
```

- The policy can then ensure (via the SELinux kernel services) that files created in the *in_queue* are relabeled:

```
type_transition ext_gateway_t in_queue_t : file in_file_t;
```

An example output from a directory listing shows the resulting file labels:

```
ls -Za /usr/message_queue/in_queue
drwxr-xr-x root root unconfined_u:object_r:in_queue_t .
drwxr-xr-x root root system_u:object_r:unconfined_t ..
-rw-r--r-- root root unconfined_u:object_r:in_file_t Message-1
-rw-r--r-- root root unconfined_u:object_r:in_file_t Message-2
```

Multi-Level and Multi-Category Security

As stated in the **Mandatory Access Control (MAC)** section as well as supporting Type Enforcement (TE), SELinux also supports MLS and MCS by adding an optional *level* or *range* entry to the security context. This section gives a brief introduction to MLS and MCS.

Figure 8: Security Levels and Data Flows shows a simple diagram where security levels represent the classification of files within a file server. The security levels are strictly hierarchical and conform to the *Bell-La & Padula model* (BLP) in that (in the case of SELinux) a process (running at the 'Confidential' level) can read / write at their current level but only read down levels or write up levels (the assumption here is that the process is authorised).

This ensures confidentiality as the process can copy a file up to the secret level, but can never re-read that content unless the process 'steps up to that level', also the process cannot write files to the lower levels as confidential information would then drift downwards.

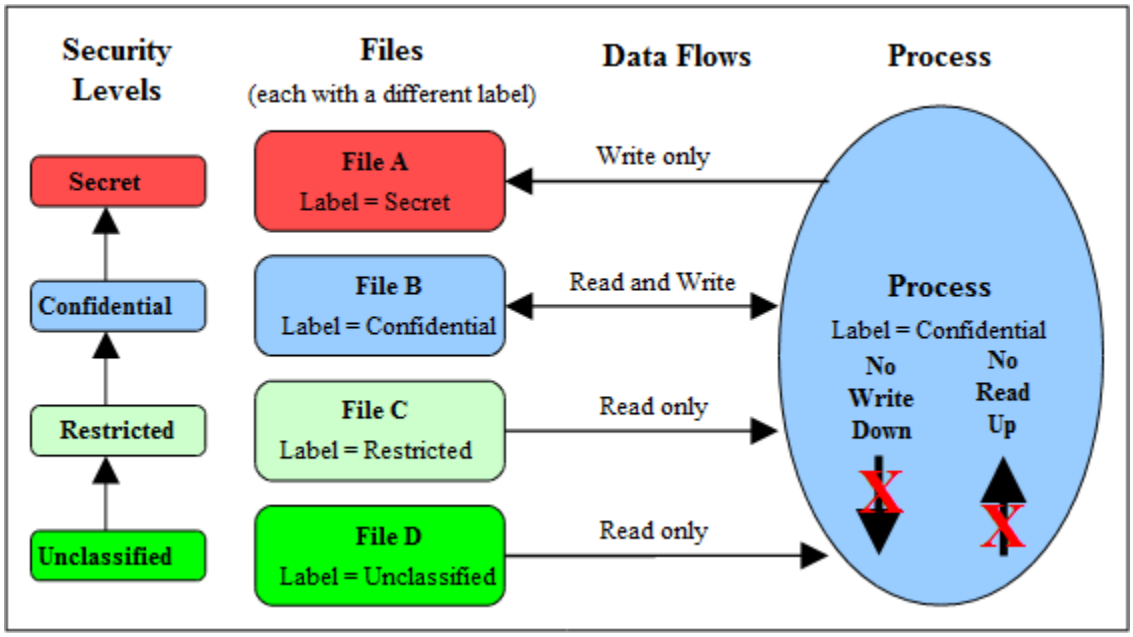


Figure 8: Security Levels and Data Flows - *This shows how the process can only 'Read Down' and 'Write Up' within an MLS enabled system.*

To achieve this level of control, the MLS extensions to SELinux make use of constraints similar to those described in the type enforcement **Type Enforcement - Constraints** section, except that the statement is called *mlsconstrain*.

However, as always life is not so simple as:

1. Processes and objects can be given a range that represents the low and high security levels.
2. The security level can be more complex, in that it is a hierarchical sensitivity and zero or more non-hierarchical categories.

3. Allowing a process access to an object is managed by 'dominance' rules applied to the security levels.
4. Trusted processes can be given privileges that will allow them to bypass the BLP rules and basically do anything (that the security policy allowed of course).
5. Some objects do not support separate read / write functions as they need to read / respond in cases such as networks.

The sections that follow discuss the format of a security level and range, and how these are managed by the constraints mechanism within SELinux using dominance rules.

MLS or MCS Policy

SELinux only knows of MLS, i.e. it has a MLS engine in the security server[6] and a MLS portion of the policy configuration that drives that engine. The MLS engine has been leveraged by two different types of policy:

- The MLS configuration modeled after Bell-LaPadula.
- The MCS configuration that allows a process or object to be labeled with categories. This has proved useful as a transparent isolation mechanism for sandbox, container, and virtualization runtimes (see the **Virtual Machine Support** section).

As a security level is a combination of a hierarchical sensitivity and a non-hierarchical (potentially empty) category set, MCS doesn't employ sensitivities since there is no hierarchical relationship to be enforced. Note however, that the SELinux kernel policy language defines *sensitivity* as a single value such as *s0*, *s1* etc.. Therefore the BLP defined sensitivity is formed in an SELinux MLS policy by a combination of the following (excluding the category statements):

```
sensitivity s0;
sensitivity s1;
dominance { s0 s1 } # s1 dominates s0
```

While an MCS policy does not use the BLP sensitivity definition, an SELinux policy still requires the following entries (basically defining a single level):

```
sensitivity s0;
dominance { s0 }
```

The number of sensitivities, number of categories, and the set of MLS constraints used to determine whether a permission is allowed are entirely up to the policy author. See the **The Reference Policy** section for its configuration parameters.

Security Levels

The optional MLS policy extension adds an additional security context component that consists of the following highlighted entries:

user:role:type ***:sensitivity[:category,...] - sensitivity[:category,...]***

Note that:

- Security Levels on objects are called Classifications.
- Security Levels on subjects are called Clearances.

The list below describes the components that make up a security level and how two security levels form a range for an MLS **Security Context**:

- **Low**
  - *sensitivity[:category, ...]*
    - For a process or subject this is the current level or sensitivity. For an object this is the current level or sensitivity.
- **SystemLow**
  - This is the lowest level or classification for the system (for SELinux this is generally *s0*, note that there are no categories).
- **High**
  - *sensitivity[:category, ...]*
    - For a process or subject this is the Clearance. For an object this is the maximum range.
- **SystemHigh**
  - This is the highest level or classification for the system (for an MLS Reference Policy the default is *s15:c0,c255*, although this is a configurable build option).

The format used in the policy language statements is fully described in the MLS Statements section, however a brief overview follows.

MLS / MCS Range Format

The components are used to define the MLS security levels and MCS categories within the security context are:

```
user:role:type:sensitivity[:category,...]  -  sensitivity [:category,...]
--------------▼------------------------▼-----▼-------------------------▼
               |        level           |  -  |          level          |
               |                         range                         |
```

Where:

sensitivity

- Sensitivity levels are hierarchical with (traditionally) *s0* being the lowest. These values are defined using the *sensitivity* statement. To define their hierarchy, the *dominance* statement is used.
- For MLS systems the highest sensitivity is the last one defined in the *dominance* statement (low to high). Traditionally the maximum for MLS systems is *s15* (although the maximum value for the *Reference Policy* is a build time option).
- For MCS systems there is only one *sensitivity* statement defined, and that is *s0*.

category

- Categories are optional (i.e. there can be zero or more categories) and they form unordered and unrelated lists of 'compartments'. These values are defined using the *category* statement, where for example *c0.c3* represents the range that consists of *c0 c1 c2 c3* and *c0, c3, c7* that represents an unordered list. Traditionally the values are between*c0* and *c255* (although the maximum value for the Reference Policy is a build time option).

level

- The level is a combination of the *sensitivity* and *category* values that form the actual security level. These values are defined using the *level* statement.

Example policy entries:

```
# MLS Policy statements:
sensitivity s0;
sensitivity s1;
dominance { s0 s1 }
category c0;
category c1;
level s0:c0.c1;
level s1:c0.c1;
```

```
# MCS Policy statements:
sensitivity s0;
dominance { s0 }
category c0;
category c1;
level s0:c0.c1;
```

Translating Levels

When writing policy for MLS / MCS security level components it is usual to use an abbreviated form such as *s0*, *s1* etc. to represent sensitivities and *c0*, *c1* etc. to represent categories. This is done simply to conserve space as they are held on files as extended attributes and also in memory. So that these labels can be represented in human readable form, a translation service is provided via the **setrans.conf** configuration file that is used by the ***mcstransd**(8)* daemon. For example *s0* = Unclassified, *s15* = Top Secret and *c0* = Finance, *c100* = Spy Stories. The ***semanage**(8)* command can be used to set up this translation and is shown in the **setrans.conf** configuration file section.

Managing Security Levels via Dominance Rules

As stated earlier, allowing a process access to an object is managed by ***dominance*** rules applied to the security levels. These rules are as follows:

Security Level 1 dominates Security Level 2

- If the sensitivity of *Security Level 1* is equal to or higher than the sensitivity of *Security Level 2* and the categories of *Security Level 1* are the same or a superset of the categories of *Security Level 2*.

Security Level 1 is dominated by Security Level 2

- If the sensitivity of *Security Level 1* is equal to or lower than the sensitivity of *Security Level 2* and the categories of *Security Level 1* are a subset of the categories of *Security Level 2*.

Security Level 1 equals Security Level 2

- If the sensitivity of *Security Level 1* is equal to *Security Level 2* and the categories of *Security Level 1* and *Security Level 2* are the same set (sometimes expressed as: both Security Levels dominate each other).

Security Level 1 is incomparable to Security Level 2

- If the categories of *Security Level 1* and *Security Level 2* cannot be compared (i.e. neither Security Level dominates the other).

To illustrate the usage of these rules, **Table 1: MLS Security Levels** lists the security level attributes in a table to show example files (or documents) that have been allocated labels such as *s3:c0*. The process that accesses these files (e.g. an editor) is running with a range of *s0 - s3:c1.c5* and has access to the files highlighted within the grey box area.

As the MLS *dominance* statement is used to enforce the sensitivity hierarchy, the security levels now follow that sequence (lowest = *s0* to highest = *s3*) with the categories being unordered lists of 'compartments'. To allow the process access to files within its scope and within the dominance rules, the process will be constrained by using the *mlsconstrain* statement as illustrated in **Figure 9: *mlsconstrain* Statements controlling Read Down & Write Up**.

Security Level	**Sensitivity**								
	Category ->	*c0*	*c1*	*c2*	*c3*	*c4*	*c5*	*c6*	*c7*
Secret	*s3*	*s3:c0*					***s3:c5***	*s3:c6*	
Confidential	*s2*		***s2:c1***	***s2:c2***	***s2:c3***	***s2:c4***			s2:c7
Restricted	*s1*	*s1:c0*	***s1:c1***						s1:c7
Unclassified	*s0*	*s0:c0*			***s0:c3***				s0:c7

Table 1: MLS Security Levels - *Showing that a process running at s0 - s3:c1.c5 has access to the highlighted* ***sensitivity:category*** *files.*

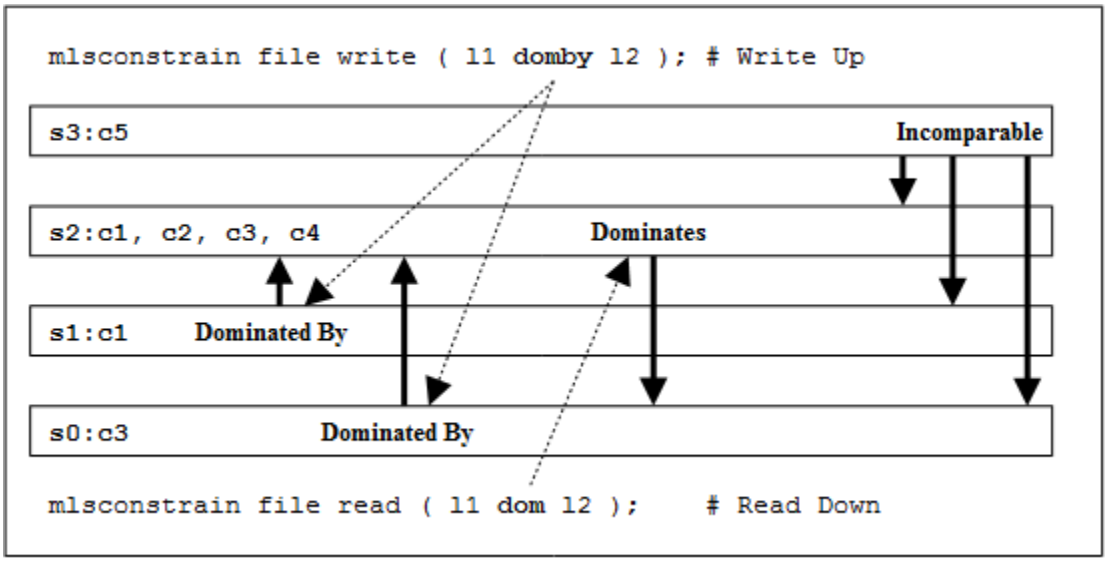


Figure 9: Showing mlsconstrain Statements controlling Read Down & Write Up - *This ties in with* **Table 1: MLS Security Levels** *that shows a process running with a security range of s0 - s3:c1.c5.*

Using **Figure 9: *mlsconstrain* Statements controlling Read Down & Write Up**:

- To allow write-up, the source level (*l1*) must be **dominated by** the target level (*l2*):
  - Source level = *s0:c3* or *s1:c1*
  - Target level = *s2:c1.c4*

As can be seen, either of the source levels are **dominated by** the target level.

- To allow read-down, the source level (*l1*) must **dominate** the target level (*l2*):
  - Source level = *s2:c1.c4*
  - Target level = *s0:c3*

As can be seen, the source level does **dominate** the target level.

However in the real world the SELinux MLS Reference Policy does not allow the write-up unless the process has a special privilege (by having the domain type added to an attribute), although it does allow the read-down. The default is to use *l1 eq l2* (i.e. the levels are equal). The reference policy MLS source file (policy/mls) shows these *mlsconstrain* statements.

MLS Labeled Network and Database Support

Networking for MLS is supported via the NetLabel CIPSO (commercial IP security option) and CALIPSO (Common Architecture Label IPv6 Security Option) services as discussed in the **SELinux Networking Support** section.

PostgreSQL supports labeling for MLS database services as discussed in the **SE-PostgreSQL Support** section.

Common Criteria Certification

While the *Common Criteria* certification process is beyond the scope of this Notebook, it is worth highlighting that specific Red Hat GNU / Linux versions of software, running on specific hardware platforms with SELinux / MLS policy enabled, have passed the Common Criteria evaluation process. Note, for the evaluation (and deployment) the software and hardware are tied together, therefore whenever an update is carried out, an updated certificate should be obtained.

The Red Hat evaluation process cover the:

- Labeled Security Protection Profile (*LSPP*) - This describes how systems that implement security labels (i.e. MLS) should function.
- Controlled Access Protection Profile (*CAPP*) - This describes how systems that implement DAC should function.

An interesting point:

- Both Red Hat Linux 5.1 and Microsoft Server 2003 (with XP) have both been certified to EAL4+ , however while the evaluation levels may be the same the Protection Profiles that they were evaluated under were: Microsoft CAPP only, Red Hat CAPP and LSPP. Therefore always look at the protection profiles as they define what was actually evaluated.

Types of SELinux Policy

This section describes the different type of policy descriptions and versions that can be found within SELinux.

The type of SELinux policy can described in a number of ways:

1. Source code - These can be described as: **Reference Policy** or **Custom**. They are generally written using **Kernel Policy Language**, **Reference Policy Support Macros**, or using **CIL**
2. They can also be classified as: **Monolithic**, **Base Module or Loadable Module**.
3. Policies can also be described by the **type of policy functionality** they provide such as: targeted, mls, mcs, standard, strict or minimum.
4. Classified using language statements - These can be described as **Modular, Optional** or **Conditional**.
5. Binary or Kernel policy. These are the compiled policy used by the kernel.
6. Classification can also be on the '**policy version**' used (examples are version 22, 23 and 24).
7. Policy can also be generated depending on the target platform of either 'selinux' (the default) or 'xen' (see the SELinux policy generation tools ***checkpolicy****(8)*, ***secilc****(8)* and ***semanage****(8) target_platform* options).

As can be seen the description of a policy can vary depending on the context.

Reference Policy

Note that this section only gives an introduction to the Reference Policy, the installation, configuration and building of a policy using this is contained in **The Reference Policy** section.

The Reference Policy is now the standard policy source used to build Linux based SELinux policies, and its main aim is to provide a single source tree with supporting documentation that can be used to build policies for different purposes such as confining important daemons, supporting MLS / MCS and locking down systems so that all processes are under SELinux control.

The Reference Policy is now used by all major distributions of Linux, however each distribution makes its own specific changes to support their 'version of the Reference Policy'. For example, the Fedora distribution is based on a specific build of the standard Reference Policy that is then modified and distributed by Red Hat and distributed as a number of RPMs.

The Reference Policy can be built as a Monolithic policy or as a Modular policy that has a 'base module' with zero or more optional 'loadable modules'.

Policy Functionality Based on Name or Type

Generally a policy is installed with a given name such as *targeted*, *mls*, *refpolicy* or *minimum* that attempts to describes its functionality. This name then becomes the entry in:

1. The directory pointing to the policy location (e.g. if the name is *targeted*, then the policy will be installed in */etc/selinux/targeted*).

2. The *SELINUXTYPE* entry in the */etc/selinux/config* file when it is the active policy (e.g. if the name is *targeted*, then a *SELINUXTYPE=targeted* entry would be in the */etc/selinux/config* file).

This is how the reference policies distributed with Fedora are named, where:

- minimum - supports a minimal set of confined daemons within their own domains. The remainder run in the unconfined_t space. Red Hat pre-configure MCS support within this policy.
- targeted - supports a greater number of confined daemons and can also confine other areas and users. Red Hat pre-configure MCS support within this policy.
- mls - supports server based MLS systems.

The Reference Policy also has a *TYPE* description that describes the type of policy being built by the build process, these are:

- standard - supports confined daemons and can also confine other areas and users.
- mcs - As standard but supports MCS labels.
- mls - supports server based MLS systems.

The *NAME* and *TYPE* entries are defined in the reference policy *build.conf* file that is described in the Reference Policy **Source Configuration Files** section.

Custom Policy

This generally refers to a policy source that is either:

1. A customised version of the Reference Policy (i.e. not the standard distribution version e.g. Red Hat policies).
2. A policy that has been built using policy language statements (CIL or Kernel) to build a specific policy such as the basic policy built in the Notebook *notebook-examples/selinux-policy* there are following policies:
    - **Kernel Policy Language**
    - **CIL Policy Language**

These examples were built using the Notebook 'build-sepolicy' command that is described in ***build-sepolicy***, this uses the Reference Policy *policy/flask* files (*security_classes*, *access_vectors* and *initial_sids*) to build the object classes/permissions. The kernel source has a similar policy build command in *scripts/selinux/mdp*, however it uses the selinux kernel source files to build the object classes/permissions (see kernel *Documentation/admin-guide/LSM/SELinux.rst* for build instructions, also the **Notebook Sample Policy - README**).

Monolithic Policy

A Monolithic policy is an SELinux policy that is compiled from one source file called (by convention) *policy.conf* (i.e. it does not use the Loadable Module Policy infrastructure which therefore makes it suitable for embedded systems as there is no policy store overhead.

Monolithic policies are generally compiled using the ***checkpolicy****(8)* SELinux command.

The Reference Policy supports building of monolithic policies.

In some cases the kernel policy binary file is also called a monolithic policy.

Loadable Module Policy

The loadable module infrastructure allows policy to be managed on a modular basis, in that there is a base policy module that contains all the core components of the policy (i.e. the policy that should always be present), and zero or more modules that can be loaded/unloaded as required (for example if there is a module to enforce policy for ftp, but ftp is not used, then that module could be unloaded).

There are number of components that form the infrastructure:

1. Policy source code that is constructed for a modular policy with a base module and optional loadable modules.
2. Utilities to compile and link modules and place them into a 'policy store'.
3. Utilities to manage the modules and associated configuration files within the 'policy store'.

Figure 2: High Level SELinux Architecture shows these components along the top of the diagram. The files contained in the policy store are detailed in the **Policy Store Configuration Files** section.

The policy language was extended to handle loadable modules as detailed in the **Modular Policy Support Statements** section. For a detailed overview whiteon how the modular policy is built into the final **binary policy** for loading into the kernel, see "**SELinux Policy Module Primer**".

Optional Policy

The loadable module policy infrastructure supports an ***optional*** policy statement that allows policy rules to be defined but only enabled in the binary policy once the conditions have been satisfied.

Conditional Policy

Conditional policies can be implemented in monolithic or loadable module policies and allow parts of the policy to be enabled or not depending on the state of a boolean flag at run time. This is often used to enable or disable features within the policy (i.e. change the policy enforcement rules).

The boolean flag status is held in kernel and can be changed using the ***setsebool****(8)* command either persistently across system re-boots or temporarily (i.e. only valid until a re-boot). The following example shows a persistent conditional policy change:

```
setsebool -P ext_gateway_audit false
```

The conditional policy language statements are the *bool* Statement that defines the boolean flag identifier and its initial status, and the *if* Statement that allows certain rules to be executed depending on the state of the boolean value or values. See the **Conditional Policy Statements** section.

Binary Policy

This is also know as the kernel policy and is the policy file that is loaded into the kernel and is located at /etc/selinux/<SELINUXTYPE>/policy/policy.<version>. Where <*SELINUXTYPE*> is the policy name specified in the SELinux configuration file /etc/selinux/config and <version> is the SELinux **policy version**.

The binary policy can be built from source files supplied by the Reference Policy or custom built source files.

An example */etc/selinux/config* file is shown below where the *SELINUXTYPE=targeted* entry identifies the policy name that will be used to locate and load the active policy:

```
SELINUX=permissive
SELINUXTYPE=targeted
```

From the above example, the actual binary policy file would be located at */etc/selinux/targeted/policy* and be called *policy.32* (as version 32 is supported by Fedora):

/etc/selinux/targeted/policy/policy.32

Policy Versions

SELinux has a policy database (defined in the libsepol library) that describes the format of data held within a **binary policy**, however, if any new features are added to SELinux (generally language extensions) this can result in a change to the policy database. Whenever the policy database is updated, the policy version is incremented.

The ***sestatus****(8)* command will show the maximum policy version supported by the kernel in its output as follows:

```
SELinux status:                 enabled
SELinuxfs mount:                /sys/fs/selinux
SELinux root directory:         /etc/selinux
Loaded policy name:             targeted
Current mode:                   enforcing
Mode from config file:          enforcing
Policy MLS status:              enabled
Policy deny_unknown status:     allowed
Memory protection checking:     actual (secure)
Max kernel policy version:      32
```

The following table describes the features added for each policy version and its corresponding modular policy version. When these features are implemented there may also be functionality added to the kernel, libselinux and/or libsepol. If known, these version requirements are also listed.

Policy: 15 Module: 4

The base version when SELinux was merged into the kernel.

Policy: 16

Added **Conditional Policy** support (the bool feature).

Policy: 17

Added support for IPv6.

Policy: 18

Added Netlink support.

Policy: 19 Module: 5

Added MLS support, plus the *validatetrans* Statement.

Policy: 20

Reduced the size of the access vector table.

Policy: 21 Module: 6

Added support for the MLS *range_transition* Statement.

Policy: 22 Module: 7

Added *policycap* Statement that allows various kernel options to be enabled as described in the **Policy Configuration Statements** section.

Policy: 23 Module: 8

Added support for the *permissive* statement. This allows a domain to run in permissive mode while the others are still confined (instead of the all or nothing set by the *SELINUX* entry in the */etc/selinux/config* file).

Policy: 24 Module: 9 / 10

Add support for the *typebounds* statement. This was added to support a hierarchical relationship between two domains in multi-threaded web servers as described in **A secure web application platform powered by SELinux**.

Policy: 25 Module: 11

Add support for file name transition in the *type_transition* rule. Requires kernel 2.6.39 minimum.

Policy: 26 Module: 12 / 13

Add support for a class parameter in the *role_transition* rule and support for the *attribute_role* and *roleattribute* statements. These require kernel 2.6.39 minimum.

Module: 14

Separate tunables.

Policy: 27 Module: 15

Support setting object defaults for the user, role and range components when computing a new context. Requires kernel 3.5 minimum.

Policy: 28 Module: 16

Support setting object defaults for the type component when computing a new context. Requires kernel 3.5 minimum.

Policy: 29 Module: 17

Support attribute names within constraints. This allows attributes as well as the types to be retrieved from a kernel policy to assist ***audit2allow****(8)* etc. to determine what attribute needs to be updated. Note that the attribute does not determine the constraint outcome, it is still the list of types associated to the constraint. Requires kernel 3.14 minimum.

Policy: 30 Module: 18

For the *selinux* target platform adds new *xperm* rules as explained in the **Extended Access Vector Rules** section. This is to support ioctl allowlists as explained in the ***ioctl* Operation Rules** section. Requires kernel 4.3 minimum. For modular policy support requires libsepol 2.7 minimum.

Policy: 30

For the '*xen*' target platform support the *devicetreecon* statement and also expand the existing I/O memory range to 64 bits as explained in the **Xen Statements** section.

Policy: 31 Module: 19

Add InfiniBand (IB) partition key (Pkey) and IB end port object labeling as explained in the **InfiniBand Labeling Statements** section. Requires kernel 4.13 minimum.

Policy: 32 Module: 20

Specify *glblub* as a *default_range* default and the computed transition will be the intersection of the MLS range of the two contexts. See the **default_range** for details. Requires kernel 5.5 minimum.

Policy: 33

Implement a more space-efficient form of storing filename transition rules in the binary policy (also decreases policy load time). Requires kernel 5.8 minimum with libsepol version 3.2 minimum.

SELinux Permissive and Enforcing Modes

SELinux has three major modes of operation:

Enforcing - SELinux is enforcing the loaded policy.

Permissive - SELinux has loaded the policy, however it is not enforcing the policy rules. This is generally used for testing as the audit log will contain the AVC denied messages as defined in the **Auditing SELinux Events** section. The SELinux utilities such as ***audit2allow**(1)* and ***audit2why**(8)* can then be used to determine the cause and possible resolution by generating the appropriate allow rules.

Disabled - The SELinux infrastructure is not enabled, therefore no policy can be loaded.

These flags are set in the */etc/selinux/config* file as described in the **Global Configuration Files** section.

There is another method for running specific domains in permissive mode using the kernel policy *permissive* statement. This can be used directly in a user written module or ***semanage**(8)* will generate the appropriate module and load it using the following example command:

```
# This example will add a new module in:
#   /var/lib/selinux/<SELINUXTYPE>/active/modules/400/permissive_unconfined_t
# and then reload the policy:

semanage permissive -a unconfined_t
```

It is also possible to set permissive mode on a userspace object manager using the *libselinux* function ***avc_open**(3)*, for example the **X-Windows Object Manager** uses *avc_open()* to set whether it will always run permissive, enforcing or follow the current SELinux enforcement mode.

The ***sestatus**(8)* command will show the current SELinux enforcement mode in its output, however it does not display individual domain or object manager enforcement modes.

Auditing SELinux Events

For SELinux there are two main types of audit event:

1. **AVC Audit Events** - These are generated by the AVC subsystem as a result of access denials, or where specific events have requested an audit message (i.e. where an *auditallow* rule has been used in the policy).
2. **SELinux-aware Application Events** - These are generated by the SELinux kernel services and SELinux-aware applications for events such as system errors, initialisation, policy load, changing boolean states, setting of enforcing / permissive mode, relabeling etc.

The audit and event messages are generally stored in one of the following logs (in F-27 anyway):

1. The SELinux kernel boot events are logged in the */var/log/dmesg* log.
2. The system log */var/log/messages* contains messages generated by SELinux before the audit daemon has been loaded.
3. The audit log */var/log/audit/audit.log* contains events that take place after the audit daemon has been loaded. The AVC audit messages of interest are described in the AVC Audit Events section with others described in the General SELinux Audit Events section. Fedora uses the audit framework ***auditd**(8)* as standard.

Notes:

1. It is not mandatory for SELinux-aware applications to audit events or even log them in the audit log. The decision is made by the application designer.
2. The format of audit messages do not need to conform to any format, however where possible applications should use the ***audit_log_user_avc_message**(3)* function with a suitably formatted message if using ***auditd**(8)*. The type of audit events possible are defined in the *include/libaudit.h* and *include/linux/audit.h* files.
3. Those libselinux library functions that output messages do so to *stderr* by default, however this can be changed by calling ***selinux_set_callback**(3)* and specifying an alternative log handler.

AVC Audit Events

The **AVC Audit Message Keyword Descriptions** table describes the general format of AVC audit messages in the *audit.log* when access has been denied or an audit event has been specifically requested. Other types of events are shown in the section that follows.

AVC Audit Message Keyword Descriptions:

type

- For SELinux AVC events this can be:
  - *type=AVC* - for kernel events.
  - *type=USER_AVC* - for user-space object manager events.
- Note that once the AVC event has been logged, another event with *type=SYSCALL* may follow that contains further information regarding the event.

- The AVC event can always be tied to the relevant *SYSCALL* event as they have the same *serial_number* in the *msg=audit(time:serial_number)* field as shown in the following example:
  - ***type=AVC*** *msg=audit(1243332701.744:**101**): avc: denied { getattr } for pid=2714 comm="ls" path="/usr/lib/locale/locale-archive" dev=dm-0 ino=353593 scontext=system_u:object_r:unlabeled_t:s0 tcontext=system_u:object_r:locale_t:s0 tclass=file*
  - ***type=SYSCALL*** *msg=audit(1243332701.744:**101**): arch=40000003 syscall=197 success=yes exit=0 a0=3 a1=553ac0 a2=552ff4 a3=bfc5eab0 items=0 ppid=2671 pid=2714 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts1 ses=1 comm="ls" exe="/bin/ls" subj=system_u:object_r:unlabeled_t:s0 key=(null)*

msg

- This will contain the audit keyword with a reference number (e.g. *msg=audit(1243332701.744:101)*)

avc

- This will be either denied when access has been denied or granted when an *auditallow* rule has been defined by the policy.
- The entries that follow the *avc=* field depend on what type of event is being audited. Those shown below are generated by the kernel AVC audit function, however the user space AVC audit function will return fields relevant to the application being managed by their Object Manager.

pid and *comm*

- If a task, then log the process id (*pid*) and the name of the executable file (*comm*).

capability

- If a capability event then log the identifier.

path, *name*, *dev* and *ino*

- If a File System event then log the relevant information. Note that the *name* field may not always be present.

laddr, *lport*, *faddr* and *fport*

- If a Socket event then log the Source / Destination addresses and ports for IPv4 or IPv6 sockets (*AF_INET*).

path

- If a File Socket event then log the path (*AF_UNIX*).

saddr, *src*, *daddr*, *dest* and *netif*

- If a Network event then log the Source / Destination addresses and ports with the network interface for IPv4 or IPv6 networks (*AF_INET*).

sauid, *hostname* and *addr*

- IPSec security association identifiers.

resid and *restype*

- X-Windows resource ID and type.

scontext

- The security context of the source or subject.

tcontext

- The security context of the target or object.

tclass

- The object class of the target or object.

permissive

- Keyword introduced in Linux 4.17 to indicate whether the event was denied or granted due to global or per-domain permissive mode.

Example Audit Events

This is an example ***denied*** message - note that there are two ***type=AVC*** calls, but only one corresponding ***type=SYSCALL*** entry.

```
type=AVC msg=audit(1242575005.122:101): avc: denied { rename } for
pid=2508 comm="canberra-gtk-pl"
name="c73a516004b572d8c845c74c49b2511d:runtime.tmp" dev=dm-0 ino=188999
scontext=test_u:staff_r:oddjob_mkhomedir_t:s0
tcontext=test_u:object_r:gnome_home_t:s0 tclass=lnk_file

type=AVC msg=audit(1242575005.122:101): avc: denied { unlink } for
pid=2508 comm="canberra-gtk-pl"
name="c73a516004b572d8c845c74c49b2511d:runtime" dev=dm-0 ino=188578
scontext=test_u:staff_r:oddjob_mkhomedir_t:s0
tcontext=system_u:object_r:gnome_home_t:s0 tclass=lnk_file

type=SYSCALL msg=audit(1242575005.122:101): arch=40000003 syscall=38
success=yes exit=0 a0=82d2760 a1=82d2850 a2=da6660 a3=82cb550 items=0
ppid=2179 pid=2508 auid=500 uid=500 gid=500 euid=500 suid=500 fsuid=500
egid=500 sgid=500 fsgid=500 tty=(none) ses=1 comm="canberra-gtk-pl"
exe="/usr/bin/canberra-gtk-play"
subj=test_u:staff_r:oddjob_mkhomedir_t:s0 key=(null)
```

These are example X-Windows object manager audit messages:

```
type=USER_AVC msg=audit(1267534171.023:18): user pid=1169 uid=0
auid=4294967295 ses=4294967295
subj=system_u:unconfined_r:unconfined_t msg='avc: denied { getfocus }
for request=X11:GetInputFocus comm=X-setest xdevice="Virtual core
keyboard" scontext=unconfined_u:unconfined_r:x_select_paste_t
tcontext=system_u:unconfined_r:unconfined_t tclass=x_keyboard :
exe="/usr/bin/Xorg" sauid=0 hostname=? addr=? terminal=?'

type=USER_AVC msg=audit(1267534395.930:19): user pid=1169 uid=0
auid=4294967295 ses=4294967295
subj=system_u:unconfined_r:unconfined_t msg='avc: denied { read } for
request=SELinux:SELinuxGetClientContext comm=X-setest resid=3c00001
restype=<unknown>
scontext=unconfined_u:unconfined_r:x_select_paste_t
tcontext=unconfined_u:unconfined_r:unconfined_t tclass=x_resource :
exe="/usr/bin/Xorg" sauid=0 hostname=? addr=? terminal=?'
```

This is an example **granted** audit message:

```
type=AVC msg=audit(1239116352.727:311): avc: granted { transition } for
pid=7687 comm="bash" path="/usr/move_file/move_file_c" dev=dm-0
ino=402139 scontext=unconfined_u:unconfined_r:unconfined_t
tcontext=unconfined_u:unconfined_r:move_file_t tclass=process

type=SYSCALL msg=audit(1239116352.727:311): arch=40000003 syscall=11
success=yes exit=0 a0=8a6ea98 a1=8a56fa8 a2=8a578e8 a3=0 items=0
ppid=2660 pid=7687 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
sgid=0 fsgid=0 tty=(none) ses=1 comm="move_file_c"
exe="/usr/move_file/move_file_c"
subj=unconfined_u:unconfined_r:move_file_t key=(null)
```

General SELinux Audit Events

This section shows a selection of non-AVC SELinux-aware services audit events taken from the audit.log. For a list of valid *type*= entries, the following include files should be consulted: *include/libaudit.h* and *include/linux/audit.h*.

Note that there can be what appears to be multiple events being generated for the same event. For example the kernel security server will generate a *MAC_POLICY_LOAD* event to indicate that the policy has been reloaded, but then each userspace object manager could then generate a *USER_MAC_POLICY_LOAD* event to indicate that it had also processed the event.

Policy reload - *MAC_POLICY_LOAD*, *USER_MAC_POLICY_LOAD* - These events were generated when the policy was reloaded.

```
type=MAC_POLICY_LOAD msg=audit(1336662937.117:394): policy loaded
auid=0 ses=2

type=SYSCALL msg=audit(1336662937.117:394): arch=c000003e syscall=1
success=yes exit=4345108 a0=4 a1=7f0a0c547000 a2=424d14 a3=7fffe3450f20
items=0 ppid=3845 pid=3848 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0
egid=0 sgid=0 fsgid=0 tty=pts2 ses=2 comm="load_policy"
exe="/sbin/load_policy"
subj=unconfined_u:unconfined_r:load_policy_t:s0-s0:c0.c1023
key=(null)

type=USER_MAC_POLICY_LOAD msg=audit(1336662938.535:395): pid=0 uid=0
auid=4294967295 ses=4294967295
subj=system_u:system_r:xserver_t:s0-s0:c0.c1023 msg='avc: received
policyload notice (seqno=2) : exe="/usr/bin/Xorg" sauid=0 hostname=?
addr=? terminal=?'
```

Change enforcement mode - *MAC_STATUS* - This was generated when the SELinux enforcement mode was changed:

```
type=MAC_STATUS msg=audit(1336836093.835:406): enforcing=1
old_enforcing=0 auid=0 ses=2

type=SYSCALL msg=audit(1336836093.835:406): arch=c000003e syscall=1
success=yes exit=1 a0=3 a1=7fffe743f9e0 a2=1 a3=0 items=0 ppid=2047
pid=5591 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0
```

```
tty=pts0 ses=2 comm="setenforce" exe="/usr/sbin/setenforce"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key=(null)
```

Change boolean value - *MAC_CONFIG_CHANGE* - This event was generated when ***setsebool****(8)* was run to change a boolean. Note that the boolean name plus new and old values are shown in the *MAC_CONFIG_CHANGE* type event with the *SYSCALL* event showing what process executed the change.

```
type=MAC_CONFIG_CHANGE msg=audit(1336665376.629:423):
bool=domain_paste_after_confirm_allowed val=0 old_val=1 auid=0
ses=2

type=SYSCALL msg=audit(1336665376.629:423): arch=c000003e syscall=1
success=yes exit=2 a0=3 a1=7fff42803200 a2=2 a3=7fff42803f80 items=0
ppid=2015 pid=4664 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
sgid=0 fsgid=0 tty=pts0 ses=2 comm="setsebool" exe="/usr/sbin/setsebool"
subj=unconfined_u:unconfined_r:setsebool_t:s0-s0:c0.c1023 key=(null)
```

NetLabel - *MAC_UNLBL_STCADD* - Generated when adding a static non-mapped label. There are many other NetLabel events possible, such as: *MAC_MAP_DEL*, *MAC_CIPSOV4_DEL*, ...

```
type=MAC_UNLBL_STCADD msg=audit(1336664587.640:413): netlabel: auid=0
ses=2 subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
netif=lo src=127.0.0.1
sec_obj=system_u:object_r:unconfined_t:s0-s0:c0,c100 res=1

type=SYSCALL msg=audit(1336664587.640:413): arch=c000003e syscall=46
success=yes exit=96 a0=3 a1=7fffde77f160 a2=0 a3=666e6f636e753a72
items=0 ppid=2015 pid=4316 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0
egid=0 sgid=0 fsgid=0 tty=pts0 ses=2 comm="netlabelctl"
exe="/sbin/netlabelctl"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key=(null)
```

Labeled IPSec - *MAC_IPSEC_EVENT* - Generated when running ***setkey****(8)* to load IPSec configuration:

```
type=MAC_IPSEC_EVENT msg=audit(1336664781.473:414): op=SAD-add auid=0
ses=2 subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
sec_alg=1 sec_doi=1
sec_obj=system_u:system_r:postgresql_t:s0-s0:c0,c200 src=127.0.0.1
dst=127.0.0.1 spi=592(0x250) res=1

type=SYSCALL msg=audit(1336664781.473:414): arch=c000003e syscall=44
success=yes exit=176 a0=4 a1=7fff079d5100 a2=b0 a3=0 items=0 ppid=2015
pid=4356 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0
tty=pts0 ses=2 comm="setkey" exe="/sbin/setkey"
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023 key=(null)
```

SELinux kernel errors - *SELINUX_ERR* - These example events were generated by the kernel security server. These were generated by the kernel security server because *anon_webapp_t* has been give privileges that are greater than that given to the process that started the new thread (this is not allowed).

```
type=SELINUX_ERR msg=audit(1311948547.151:138):
op=security_compute_av reason=bounds
```

```
scontext=system_u:system_r:anon_webapp_t:s0-s0:c0,c100,c200
tcontext=system_u:object_r:security_t:s0 tclass=dir
perms=ioctl,read,lock

type=SELINUX_ERR msg=audit(1311948547.151:138):
op=security_compute_av reason=bounds
scontext=system_u:system_r:anon_webapp_t:s0-s0:c0,c100,c200
tcontext=system_u:object_r:security_t:s0 tclass=file
perms=ioctl,read,write,getattr,lock,append,open
```

These were generated by the kernel security server when an SELinux-aware application was trying to use ***setcon****(3)* to create a new thread. To fix this a *typebounds* statement is required in the policy.

```
type=SELINUX_ERR msg=audit(1311947138.440:126):
op=security_bounded_transition result=denied
oldcontext=system_u:system_r:httpd_t:s0-s0:c0.c300
newcontext=system_u:system_r:anon_webapp_t:s0-s0:c0,c100,c200

type=SYSCALL msg=audit(1311947138.440:126): arch=c000003e syscall=1
success=no exit=-1 a0=b a1=7f1954000a10 a2=33 a3=6e65727275632f72
items=0 ppid=3295 pid=3473 auid=4294967295 uid=48 gid=48 euid=48 suid=48
fsuid=48 egid=48 sgid=48 fsgid=48 tty=(none) ses=4294967295 comm="httpd"
exe="/usr/sbin/httpd" subj=system_u:system_r:httpd_t:s0-s0:c0.c300
key=(null)
```

The *security_compute_sid* event is generated by the security server when an invalid context is encountered. In this scenario a specified domain transition results in an invalid context *wheel.id:wheel.role:mount.subj:s0* because the *wheel.role* role is not authorised to associate with the *mount.subj* type. To fix this a *RBAC* role types rule is required in the policy.

```
SELINUX_ERR op=security_compute_sid
invalid_context="wheel.id:wheel.role:mount.subj:s0"
scontext=wheel.id:wheel.role:user.subj:s0
tcontext=sys.id:sys.role:mount.exec:s0 tclass=process

SYSCALL arch=c000003e syscall=59 success=no exit=-13 a0=55cbda501fe0
a1=55cb da502220 a2=55cbda49b210 a3=8 items=0 ppid=1303 pid=99865
auid=1000 uid=1000 gid=1000 euid=1000 suid=1000 fsuid=1000 egid=1000
sgid=1000 fsgid=1000 tty=pts1 ses=1 comm="bash" exe="/usr/bin/bash"
subj=wheel.id:wheel.role:user.subj:s0 key=(null)
```

Role changes - *USER_ROLE_CHANGE* - Used ***newrole****(1)* to set a new role that was not valid.

```
type=USER_ROLE_CHANGE msg=audit(1336837198.928:429): pid=0 uid=0
auid=0 ses=2
subj=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
msg='newrole:
old-context=unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023
new-context=?: exe="/usr/bin/newrole" hostname=? addr=?
terminal=/dev/pts/0 res=failed'
```

Capability Audit Exemptions

In the general case a rejected capability check will result in an audit event. There are however some instances in the kernel where denied capability checks are not audited, which could lead to differences in behavior between enforcing and permissive mode.

List of exemptions (no guarantee for completeness)(locations are based on kernel v6.5 unless otherwise specified):

- *fs/proc/base.c#L1110*, *fs/proc/base.c#L1129*

  If not granted *CAP_SYS_RESOURCE* the OOM kill score adjustment min value is not set.

- *fs/overlayfs/inode.c#L429*, *fs/xattr.c#L1298*

  If not granted *CAP_SYS_ADMIN* in its namespace extended attributes in the *trusted* namespace are not listed.

- *fs/xfs/xfs_fsmap.c#L894*

  If not granted *CAP_SYS_ADMIN* the XFS data device's *bnobt* is queried instead of *rmapbt*.

- *fs/xfs/xfs_ioctl.c#L1199*, *fs/xfs/xfs_iops.c#L709*

  If not granted *CAP_FOWNER* XFS quota checks on transactions are performed.

- *io_uring/io_uring.c#L3887*

  If not granted *CAP_IPC_LOCK* io_uring operations are accounted against the user's RLIMIT_MEMLOCK limit.

- *kernel/capability.c#L519*

  If not granted *CAP_SYS_PTRACE* tracing an unsafe (e.g. *no_new_privs* set or shared, see *fs/exec.c:check_unsafe_exec()*) task or a coredump of a non-user process is not permitted.

- *kernel/ksyms_common.c#L37*

  If not granted *CAP_SYSLOG* kallsyms information are not shown, except if kernel profiling is enabled and is explicitly not set to paranoid.

- *kernel/ptrace.c#L282*

  If not granted *CAP_SYS_PTRACE* in its namespace several fields in the *PID* directory entry *stat* files are not populated (*startcode*, *endcode*, *startstack*, *kstkesp*, *kstkeip*, *wchan*, *start_data*, *end_data*, *start_brk*, *arg_start*, *arg_end*, *env_start*, *env_end* and *exit_code*).

- *kernel/seccomp.c#L662*

  If not granted *CAP_SYS_ADMIN* in its namespace preparing a seccomp filter running without *no_new_privs* is not permitted.

- *lib/vsprintf.c#L881*

  If not granted *CAP_SYSLOG* restricted pointers are not included in strings formatted via *%pK*.

- *net/vmw_vsock/af_vsock.c#L779*

  If not granted *CAP_NET_ADMIN* in its namespace new *VSOCK* sockets are not marked as trusted.

- *net/sysctl_net.c#L48*

  If not granted *CAP_NET_ADMIN* in its namespace the inodes of */proc/sys/net* have more restricted *DAC* permissions.

- *security/commoncap.c#L1405*

  If not granted *CAP_SYS_ADMIN* allocation of a new virtual mapping are restricted in size to reserve memory for sysadmin.

- *security/integrity/ima/ima_policy.c#L607*

  If not granted *CAP_SETUID* rules regarding foreign *UID*s are not matched.

- *security/integrity/ima/ima_policy.c#L618*

  If not granted *CAP_SETGID* rules regarding foreign *GID*s are not matched.

- *security/landlock/syscalls.c#L413*

  If not granted *CAP_SYS_ADMIN* in its namespace enforcing a Landlock ruleset running without *no_new_privs* is not permitted.

Polyinstantiation Support

GNU / Linux supports the polyinstantiation of directories that can be utilised by SELinux via the Pluggable Authentication Module (PAM) as explained in the next section. The **Polyinstantiation of directories in an SELinux system** also gives a more detailed overview of the subject.

Polyinstantiation of objects is also supported for X-windows selections and properties that are discussed in the X-windows section. Note that sockets are not yet supported.

To clarify polyinstantiation support:

1. SELinux has *libselinux* functions and a policy rule to support polyinstantiation.
2. The polyinstantiation of directories is a function of GNU / Linux not SELinux (as more correctly, the GNU / Linux services such as PAM have been modified to support polyinstantiation of directories and have also been made SELinux-aware. Therefore their services can be controlled via policy).
3. The polyinstantiation of X-windows selections and properties is a function of the XSELinux Object Manager and the supporting XACE service.

Polyinstantiated Objects

Determining a polyinstantiated context for an object is supported by SELinux using the policy language *type_member* statement and the ***avc_compute_member**(3)* and ***security_compute_member**(3)* libselinux API functions. These are not limited to specific object classes, however only *dir*, *x_selection* and *x_property* objects are currently supported.

Polyinstantiation support in PAM

PAM supports polyinstantiation (namespaces) of directories at login time using the Shared Subtree / Namespace services available within GNU / Linux (the ***namespace.conf**(5)* man page is a good reference). Note that PAM and Namespace services are SELinux-aware.

The default installation of Fedora does not enable polyinstantiated directories, therefore this section will show the configuration required to enable the feature and some **examples**.

To implement polyinstantiated directories PAM requires the following files to be configured:

- A **pam_namespace** module entry added to the appropriate */etc/pam.d/* login configuration file (e.g. login, sshd, gdm etc.). Fedora already has these entries configured, with an example */etc/pam.d/gdm-password* file being:

```
auth     [success=done ignore=ignore default=bad] pam_selinux_permit.so
auth        substack      password-auth
auth        optional      pam_gnome_keyring.so
auth        include       postlogin

account     required      pam_nologin.so
account     include       password-auth
```

```
password    substack      password-auth
-password   optional      pam_gnome_keyring.so use_authtok

session     required      pam_selinux.so close
session     required      pam_loginuid.so
session     optional      pam_console.so
session     required      pam_selinux.so open
session     optional      pam_keyinit.so force revoke
session     required      pam_namespace.so
session     include       password-auth
session     optional      pam_gnome_keyring.so auto_start
session     include       postlogin
```

- Entries added to the */etc/security/namespace.conf* file that defines the directories to be polyinstantiated by PAM (and other services that may need to use the namespace service). The entries are explained in the *namespace.conf* section, with the default entries in Fedora being (note that the entries are commented out in the distribution):

```
# polydir  instance-prefix     method  list_of_uids
/tmp       /tmp-inst/          level   root,adm
/var/tmp   /var/tmp/tmp-inst/  level   root,adm
$HOME      $HOME/$USER.inst/   level
```

Once these files have been configured and a user logs in (although not root or adm in the above example), the PAM **pam_namespace** module would unshare the current namespace from the parent and mount namespaces according to the rules defined in the namespace.conf file. The Fedora configuration also includes an */etc/security/namespace.init* script that is used to initialise the namespace every time a new directory instance is set up. This script receives four parameters: the polyinstantiated directory path, the instance directory path, a flag to indicate if a new instance, and the user name. If a new instance is being set up, the directory permissions are set and the ***restorecon****(8)* command is run to set the correct file contexts.

***namespace.conf* Configuration File**

Each line in the namespace.conf file is formatted as follows:

```
polydir instance_prefix method list_of_uids
```

Where:

polydir

- The absolute path name of the directory to polyinstantiate. The optional strings *$USER* and *$HOME* will be replaced by the user name and home directory respectively.

instance_prefix

- A string prefix used to build the pathname for the polyinstantiated directory. The optional strings *$USER* and *$HOME* will be replaced by the user name and home directory respectively.

method

- This is used to determine the method of polyinstantiation with valid entries being:
  - *user* - Polyinstantiation is based on user name.
  - *level* - Polyinstantiation is based on user name and MLS level.

    - *context* - Polyinstantiation is based on user name and security context.
- Note that *level* and *context* are only valid for SELinux enabled systems.

list_of_uids

- A comma separated list of user names that will not have polyinstantiated directories. If blank, then all users are polyinstantiated. If the list is preceded with an '~' character, then only the users in the list will have polyinstantiated directories. There are a number of optional flags available that are described in the ***namespace.conf****(5)* man page.

Example Configurations

This section shows two sample *namespace.conf* configurations, the first uses the *method=user* and the second *method=context*. It should be notedPAM that while polyinstantiation is enabled, the full path names will not be visible, it is only when polyinstantiation is disabled that the directories become visible.

Example 1 - *method=user*:

Set the */etc/security/namespace.conf* entries as follows:

```
# polydir  instance-prefix     method  list_of_uids
/tmp       /tmp-inst/          user    root,adm
/var/tmp   /var/tmp/tmp-inst/  user    root,adm
$HOME      $HOME/$USER.inst/   user
```

Login as a normal user and the PAM / Namespace process will build the following polyinstantiated directories:

```
# The directories will contain the user name as a part of
# the polyinstantiated directory name as follows:

# /tmp
/tmp/tmp-inst/rch

# /var/tmp:
/var/tmp/tmp-inst/rch

# $HOME
/home/rch/rch.inst/rch
```

Example 2 - *method=context*:

Set the */etc/security/namespace.conf* entries as follows:

```
# polydir  instance-prefix     method   list_of_uids
/tmp       /tmp-inst/          context  root,adm
/var/tmp   /var/tmp/tmp-inst/  context  root,adm
$HOME      $HOME/$USER.inst/   context
```

Login as a normal user and the PAM / Namespace process will build the following polyinstantiated directories:

```
# The directories will contain the security context and
# user name as a part of the polyinstantiated directory
# name as follows:
```

```
# /tmp
/tmp/tmp-inst/unconfined_u:unconfined_r:unconfined_t_rch

# /var/tmp:
/var/tmp/tmp-inst/unconfined_u:unconfined_r:unconfined_t_rch

# $HOME
/home/rch/rch.inst/unconfined_u:unconfined_r:unconfined_t_rch
```

Polyinstantiation support in X-Windows

The X-Windows SELinux object manager and XACE (X Access Control Extension) supports *x_selection* and *x_property* polyinstantiated objects as discussed in the **SELinux X-Windows Support** section.

Polyinstantiation support in the Reference Policy

The reference policy *files.te* and *files.if* modules (in the kernel layer) support polyinstantiated directories. There is also a global tunable (a boolean called *polyinstantiation_enabled*) that can be used to set this functionality on or off during login. By default this boolean is set *false* (off).

The polyinstantiation of X-Windows objects (*x_selection* and *x_property*) are not currently supported by the reference policy.

PAM Login Process

Applications used to provide login services (such as ***ssh**(1)*) in Fedora use the PAM (Pluggable Authentication Modules) infrastructure to provide the following services:

- **Account Management** - This manages services such as password expiry, service entitlement (i.e. what services the login process is allowed to access).
- **Authentication Management** - Authenticate the user or subject and set up the credentials. PAM can handle a variety of devices including smart-cards and biometric devices.
- **Password Management** - Manages password updates as needed by the specific authentication mechanism being used and the password policy.
- **Session Management** - Manages any services that must be invoked before the login process completes and / or when the login process terminates. For SELinux this is where hooks are used to manage the domains the subject may enter.

The ***pam**(8)* and ***pam.conf**(5) man* pages describe the services and configuration in detail and only a summary is provided here covering the SELinux services.

The PAM configuration for Fedora is managed by a number of files located in the */etc/pam.d* directory which has configuration files for login services such as: *gdm*, *gdm-autologin*, *login*, *remote* and *sshd*.

There are also a number of PAM related configuration files in */etc/security*, although only one is directly related to SELinux that is described in the */etc/security/sepermit.conf* section and also the ***sepermit.conf**(5)*.

The main login service related PAM configuration files (e.g. *gdm*) consist of multiple lines of information that are formatted as follows:

```
service type control module-path arguments
```

Where:

service

- The service name such as *gdm* and *login* reflecting the login application. If there is a */etc/pam.d* directory, then this is the name of a configuration file name under this directory. Alternatively, a configuration file called */etc/pam.conf* can be used. Fedora uses the */etc/pam.d* configuration.

type

- These are the management groups used by PAM with valid entries being: *account*, *auth*, *password* and *session* that correspond to the descriptions given above. Where there are multiple entries of the same '*type*', the order they appear could be significant.

control

- This entry states how the module should behave when the requested task fails. There can be two formats: a single keyword such as *required*, *optional*, and *include*; or multiple space separated entries enclosed in square brackets consisting of (see the ***pam.conf**(5)* man pages):

```
[value1=action1 value2=action2 ..]
```

module-path

- Either the full path name of the module or its location relative to */lib/security* (but does depend on the system architecture).

arguments

- A space separated list of the arguments that are defined for the module.

The */etc/pam.d/sshd* PAM configuration file for the OpenSSH service is as follows:

```
#%PAM-1.0

auth       substack     password-auth
auth       include      postlogin
account    required     pam_sepermit.so
account    required     pam_nologin.so
account    include      password-auth
password   include      password-auth
session    required     pam_selinux.so close
session    required     pam_loginuid.so
session    required     pam_selinux.so open
session    required     pam_namespace.so
session    optional     pam_keyinit.so force revoke
session    optional     pam_motd.so
session    include      password-auth
session    include      postlogin
```

The core services are provided by PAM, however other library modules can be written to manage specific services such as support for SELinux. The ***pam_sepermit**(8)* and ***pam_selinux**(8)* SELinux PAM modules use the *libselinux* API to obtain its configuration information and the three SELinux PAM entries highlighted in the above configuration file perform the following functions:

- ***pam_sepermit.so*** - Allows pre-defined users the ability to logon provided that SELinux is in enforcing mode (see the */etc/security/sepermit.conf* section).
- ***pam_selinux.so open*** - Allows a security context to be set up for the user at initial logon (as all programs exec'ed from here will use this context). How the context is retrieved is described in the ***Policy Configuration Files*** *- seusers* section.
- ***pam_selinux.so close*** - This will reset the login programs context to the context defined in the policy.

Linux Security Module and SELinux

This section gives a high level overview of the LSM and SELinux internal kernel structure and workings as enabled in a kernel. A more detailed view can be found in the **Implementing SELinux as a Linux Security Module** that was used extensively to develop this section (and also using the SELinux kernel source code). The major areas covered are:

1. How the LSM and SELinux modules work together.
2. The major SELinux internal services.
3. The ***fork****(2)* and ***exec****(2)* system calls are followed through as an example to tie in with the transition process covered in the **Domain Transition** section.
4. The SELinux filesystem */sys/fs/selinux*.
5. The */proc* filesystem area most applicable to SELinux.

The LSM Module

The LSM is the Linux security framework that allows 3rd party access control mechanisms to be linked into the GNU / Linux kernel. There are a number of services that utilise the LSM:

1. SELinux - the subject of this Notebook.
2. AppArmor is a MAC service based on pathnames and does not require labeling or relabeling of filesystems. See http://wiki.apparmor.net/ for details.
3. Simplified Mandatory Access Control Kernel (SMACK). See http://www.schaufler-ca.com/ for details.
4. Tomoyo that is a name based MAC and details can be found at http://sourceforge.jp/projects/tomoyo/docs.
5. Yama extends the DAC support for ***ptrace****(2)*. See *Documentation/security/Yama.txt* for further details.
6. Lockdown feature. If set to integrity, kernel features that allow userspace to modify the running kernel are disabled. If set to confidentiality, kernel features that allow userspace to extract confidential information from the kernel are disabled.

The basic idea behind LSM is to:

- Insert security function hooks and security data structures in the various kernel services to allow access control to be applied over and above that already implemented via DAC. The type of service that have hooks inserted are shown in with an example task and program execution shown in the Fork System Call Walk-thorough and Process Transition Walk-thorough sections.
- Allow registration and initialisation services for the 3rd party security modules.
- Allow process security attributes to be available to userspace services by extending the */proc* filesystem with a security namespace as shown in */proc/self/attr/* Filesystem Files. These are located at:

```
/proc/<self|pid>/attr/<attr>
```

```
/proc/<self|pid>/task/<tid>/attr/<attr>
```

Where <*pid*> is the process id, <*tid*> is the thread id, and <*attr*> is the entry described in the */proc/self/attr/* Filesystem Files section.

- Support filesystems that use extended attributes (SELinux uses *security.selinux* as explained in the **Labeling Extended Attribute Filesystems** section).
- Later kernels allow 'module stacking' where the LSM modules can be called in a predefined order, for example:

```
lockdown,yama,loadpin,safesetid,integrity,selinux,smack,tomoyo,apparmor,bpf
```

It should be noted that the LSM does not provide any security services itself, only the hooks and structures for supporting 3rd party modules. If no 3rd party module is loaded, the capabilities module becomes the default module thus allowing standard DAC access control.

These are the type of kernel services that LSM has inserted security hooks and structures to allow access control to be managed by 3rd party modules:

Program execution	Filesystem operations	Inode operations
File operations	Task operations	Netlink messaging
Unix domain networking	Socket operations	XFRM operations
Key Management operations	IPC operations	Memory Segments
Semaphores	Capability	Sysctl
Syslog	Audit	

/proc/self/attr/ Filesystem Files

The following */proc/self/attr/* filesystem files are used by the kernel services and via libselinux (for userspace) to manage setting and reading of security contexts within the LSM defined data structures:

current

- Contains the current process security context.

exec

- Used to set the security context for the next exec call.

fscreate

- Used to set the security context of a newly created file.

keycreate

- Used to set the security context for keys that are cached in the kernel.

prev

- Contains the previous process security context.

sockcreate

- Used to set the security context of a newly created socket.

Core LSM Source Files

The core kernel LSM source files that form the LSM are as follows:

include/linux/lsm_hooks.h

- Contains comments on each of the LSM hooks use and requirements, also defines the main security structures for module stacking.

include/linux/lsm_hook_defs.h

- Defines each *LSM_HOOK* that will be expanded by macros in *lsm_hooks.h*.

include/linux/security.h

- *if/else* table of LSM hooks expanding those that are defined in the kernel build or no-op those that are not.

security/commoncap.c, *security/device_cgroup.c*

- Some capability functions were in various kernel modules have been consolidated into these source files.

security/inode.c

- This allows the 3rd party security module to initialise a security filesystem. In the case of SELinux this would be */sys/fs/selinux* that is defined in the *security/selinux/selinuxfs.c* source file.

security/security.c

- Contains the LSM framework initialisation services that will set up the hooks described in *include/linux/lsm_hooks.h* and those in the capability source files. It also provides functions to initialise 3rd party modules.

security/lsm_audit.c

- Contains common LSM audit functions.

security/min_addr.c

- Minimum VM address protection from userspace for DAC and LSM.

LSM Program Loading Hooks

The LSM hooks for validating program loading and execution (*security_bprm_** functions) are listed with their *security/selinux/hooks.c* links (although see *include/linux/lsm_hooks.h* for greater detail). These hooks are used in the **Process Transition Walk-thorough** section.

security_bprm_set_creds->selinux_bprm_set_creds

- Set up security information in the *bprm->security* field based on the file to be exec'ed contained in *bprm->file*. SELinux uses this hook to check for domain transitions and the whether the appropriate permissions have been granted, and obtaining a new SID if required.

security_bprm_committing_creds->selinux_bprm_committing_creds

- Prepare to install the new security attributes of the process being transformed by an *execve* operation. SELinux uses this hook to close any unauthorised files, clear parent signal and reset resource limits if required.

security_bprm_committed_creds->selinux_bprm_committed_creds

- Tidy up after the installation of the new security attributes of a process being transformed by an *execve* operation. SELinux uses this hook to check whether signal states can be inherited if new SID allocated.

security_bprm_check->selinux_bprm_check_security

- This hook is not used by SELinux.

The SELinux Module

This section does not go into detail of all the SELinux module functionality as the **Implementing SELinux as a Linux Security Module** does this (although a bit dated).

However this section does attempt to highlight the way some areas work by using the example described in the **Domain and Object Transitions** section. To achieve this, these will describe the:

- **Fork System Call Walk-thorough**
- **Process Transition Walk-thorough**

Core SELinux Source Files

The diagrams shown in **Figure 2: High Level SELinux Architecture** and **Figure 12: The Main LSM / SELinux Modules** can be used to see how some of these SELinux modules fit together with the *security/selinux* following files:

avc.c

- Access Vector Cache functions and structures. The function calls are for the kernel services, however they have been ported to form the *libselinux* userspace library.

exports.c

- Exported SELinux services for SECMARK (as there is SELinux specific code in the *netfilter* source tree).

hooks.c

- Contains all the SELinux functions that are called by the kernel resources via the *security_ops* structure or the *security_hook_list* in kernels supporting multiple LSMs (they form the kernel resource object managers). There are also support functions for managing process exec's, managing SID allocation and removal, interfacing into the AVC and Security Server.

netif.c

- These manage the mapping between labels and SIDs for the *net** language statements when they are declared in the active policy.

netnode.c, *netport.c*, *netlabel.c*

- The interface between ***NetLabel****(8)* services and SELinux.

netlink.c, *nlmsgtab.c*

- Manages the notification of policy updates to resources including userspace applications via *libselinux*.

selinuxfs.c

- The *selinuxfs* pseudo filesystem (*/sys/fs/selinux*) that imports/exports security policy information to/from userspace services. The services exported are shown in the **SELinux Filesystem** section.

xfrm.c

- Contains the IPSec XFRM (transform) hooks for SELinux.

include/classmap.h, *include/initial_sid_to_string.h*

- Contains all the kernel security classes and permissions. *initial_sid_to_string.h* contains the initial SID contexts. These are used to build the *flask.h* and *av_permissions.h* kernel configuration files when the kernel is being built (using the *genheaders* script defined in the *selinux/Makefile*). These files are now built support the dynamic security class mapping structure removing the need for fixed class to SID mapping.

ss/avtab.c

- AVC table functions for inserting / deleting entries.

ss/conditional.c

- Support boolean statement functions and implements a conditional AV table to hold entries.

ss/ebitmap.c

- Bitmaps to represent sets of values, such as types, roles, categories, and classes.

ss/hashtab.c

- Hash table.

ss/mls.c

- Functions to support MLS.

ss/policydb.c

- Defines the structure of the policy database. See the **SELinux Policy Module Primer** article for details on the structure.

ss/services.c

- This contains the supporting services for kernel hooks defined in *hooks.c*, the AVC and the Security Server. For example the *security_transition_sid()* that computes the SID for a new subject / object shown in **Figure 12: The Main LSM / SELinux Modules**.

ss/sidtab.c

- The SID table contains the security context indexed by its SID value.

ss/status.c

- Interface for *selinuxfs/status*. Used by the ***selinux_status_*(3)*** functions.

ss/symtab.c

- Maintains associations between symbol strings and their values.

Fork System Call Walk-thorough

This section walks through the ***fork**(2)* system call shown in **Figure 7: Domain Transition** starting at the kernel hooks that link to the SELinux services. The way the SELinux hooks are initialised into the LSM *security_ops* function table (or the *security_hook_list* in kernels supporting multiple LSMs) are also described.

Using **Figure 10: Hooks for the *fork(2)* system call**, the major steps to check whether the *unconfined_t* process has permission to use the *fork* permission are:

1. The *kernel/fork.c* has a hook that links it to the LSM function *security_task_create()* that is called to check access permissions.
2. Because the SELinux module has been initialised as the security module, the *security_ops*/*security_hook_list* table has been set to point to the SELinux *selinux_task_create()* function in *hooks.c*.
3. The *selinux_task_create()* function check whether the task has permission via the *current_has_perm(current, PROCESS__FORK)* function.
4. This will result in a call to the AVC via the *avc_has_perm()* function in *avc.c* that checks whether the permission has been granted or not. First (via *avc_has_perm_noaudit()*) the cache is checked for an entry. Assuming that there is no entry in the AVC, then the *security_compute_av()* function in *services.c* is called.
5. The *security_compute_av()* function will search the SID table for source and target entries, and if found will then call the *context_struct_compute_av()* function. The *context_struct_compute_av()* function carries out many checks to validate whether access is allowed. The steps are (assuming the access is valid):
    - Initialise the AV structure so that it is clear.
    - Check the object class and permissions are correct. It also checks the status of the *allow_unknown* flag (see the **SELinux Filesystem**, ***/etc/selinux/semanage.conf*** and **Reference Policy Build Options** *build.conf* and *UNK_PERMS* sections.
    - Checks if there are any type enforcement rules (*AVTAB_ALLOWED*, *AVTAB_AUDITALLOW*, *AVTAB_AUDITDENY*, *AVTAB_XPERMS*).
    - Check whether any conditional statements are involved via the *cond_compute_av()* function in *conditional.c*.
    - Remove permissions that are defined in any *constraint* rule via the *constraint_expr_eval()* function call (in *services.c*). This function will also check any MLS constraints.
    - *context_struct_compute_av()* checks if a process transition is being requested (it is not). If it were, then the *transition* and *dyntransition* permissions are checked and whether the role is changing.
    - Finally check whether there are any constraints applied via the *typebounds* rule.
6. Once the result has been computed it is returned to the *kernel/fork.c* system call via the initial *selinux_task_create()* function. In this case the ***fork**(2)* call is allowed.
7. **The End.**

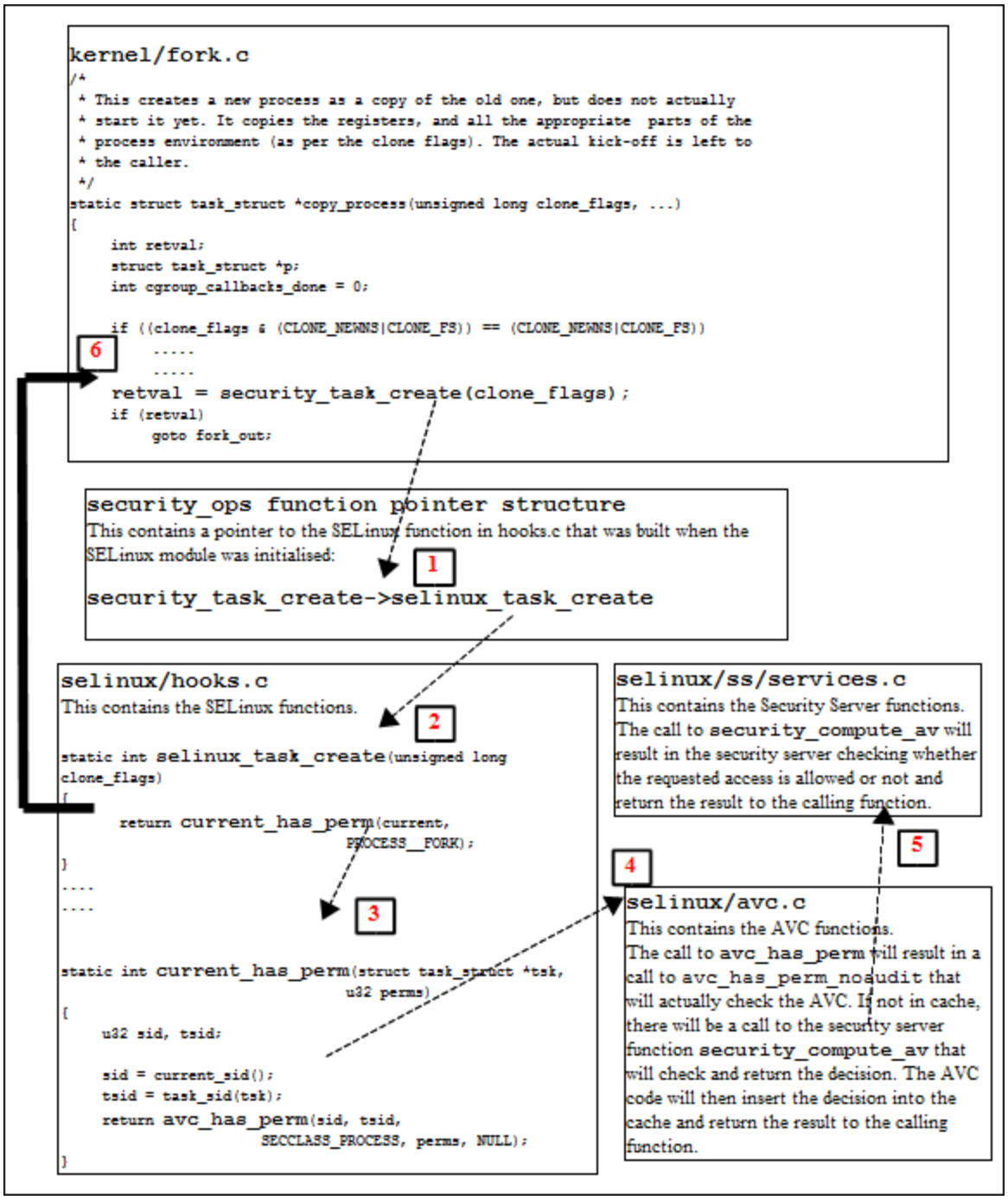


Figure 10: Hooks for the *fork(2)* system call - *This describes the steps required to check access permissions for Object Class process and permission fork.*

Process Transition Walk-thorough

This section walks through the ***execve(2)*** and checking whether a process transition to the *ext_gateway_t* domain is allowed, and if so obtain a new SID for the context *unconfined_u:message_filter_r:ext_gateway_t* as shown in **Figure 7: Domain Transition**.

The process starts with the Linux operating system issuing a *do_execve* (that passes over the file name to be run and its environment + arguments) call from the CPU specific architecture code to execute a new program (for example, from *arch/ia64/kernel/process.c*). The *do_execve()* function is located in the *fs/exec.c* source code module and does the loading and final exec as described below.

do_execve() has a number of calls to *security_bprm_** functions that are a part of the LSM (see *include/linux/lsm_hooks.h*), and are hooked by SELinux during the initialisation process (in *security/selinux/hooks.c*). **The LSM / SELinux Program Loading Hooks** describes these *security_bprm* functions that are hooks for validating program loading and execution (although see *lsm_hooks.h* for greater detail).

Therefore starting at the *do_execve()* function and using **Figure 11: Process Transition**, the following major steps will be carried out to check whether the *unconfined_t* process has permission to transition the *secure_server* executable to the *ext_gateway_t* domain:

1. The executable file is opened, a call issued to the *sched_exec()* function and the *bprm* structure is initialised with the file parameters (name, environment and arguments).
2. Via the *prepare_binprm()* function call the UID and GIDs are checked and a call issued to *security_bprm_set_creds()* that will carry out the following:
3. Call *cap_bprm_set_creds* function in *commoncap.c*, that will set up credentials based on any configured capabilities. If ***setexeccon****(3)* has been called prior to the *exec*, then that context will be used otherwise call *security_transition_sid()* function in *services.c*. This function will then call *security_compute_sid()* to check whether a new SID needs to be computed. This function will (assuming there are no errors):
    - Search the SID table for the source and target SIDs.
    - Sets the SELinux user identity.
    - Set the source role and type.
    - Checks that a *type_transition* rule exists in the *AV table* and / or the *conditional AV table* (see **Figure 12: The Main LSM / SELinux Modules**).
    - If a *type_transition*, then also check for a *role_transition* (there is a role change in the *ext_gateway.conf* policy module), set the role.
    - Check if any MLS attributes by calling *mls_compute_sid()* in *mls.c*. It also checks whether MLS is enabled or not, if so sets up MLS contexts.
    - Check whether the contexts are valid by calling *compute_sid_handle_invalid_context()* that will also log an audit message if the context is invalid.
    - Finally obtains a SID for the new context by calling *sidtab_context_to_sid()* in *sidtab.c* that will search the SID table (see **Figure 12: The Main LSM / SELinux Modules**) and insert a new entry if okay or log a kernel event if invalid.
4. The *selinux_bprm_set_creds()* function will continue by checking via the *avc_has_perm()* functions (in *avc.c*) whether the *file* class *file_execute_no_trans* is set (in this case it is not), therefore the *process* class *transition* and *file* class *entrypoint* permissions are checked (in this case they are allowed), therefore the new SID is set, and after checking various other permissions, control is passed back to the *do_execve* function.
5. The *exec_binprm* function will ultimately commit the credentials calling the SELinux *selinux_bprm_committing_creds* and *selinux_bprm_committed_creds*.
6. Various strings are copied (args etc.) and a check is made to see if the exec succeeded or not (in this case it did), therefore the *security_bprm_free()* function is ultimately called to free the *bprm* security structure.
7. **The End.**

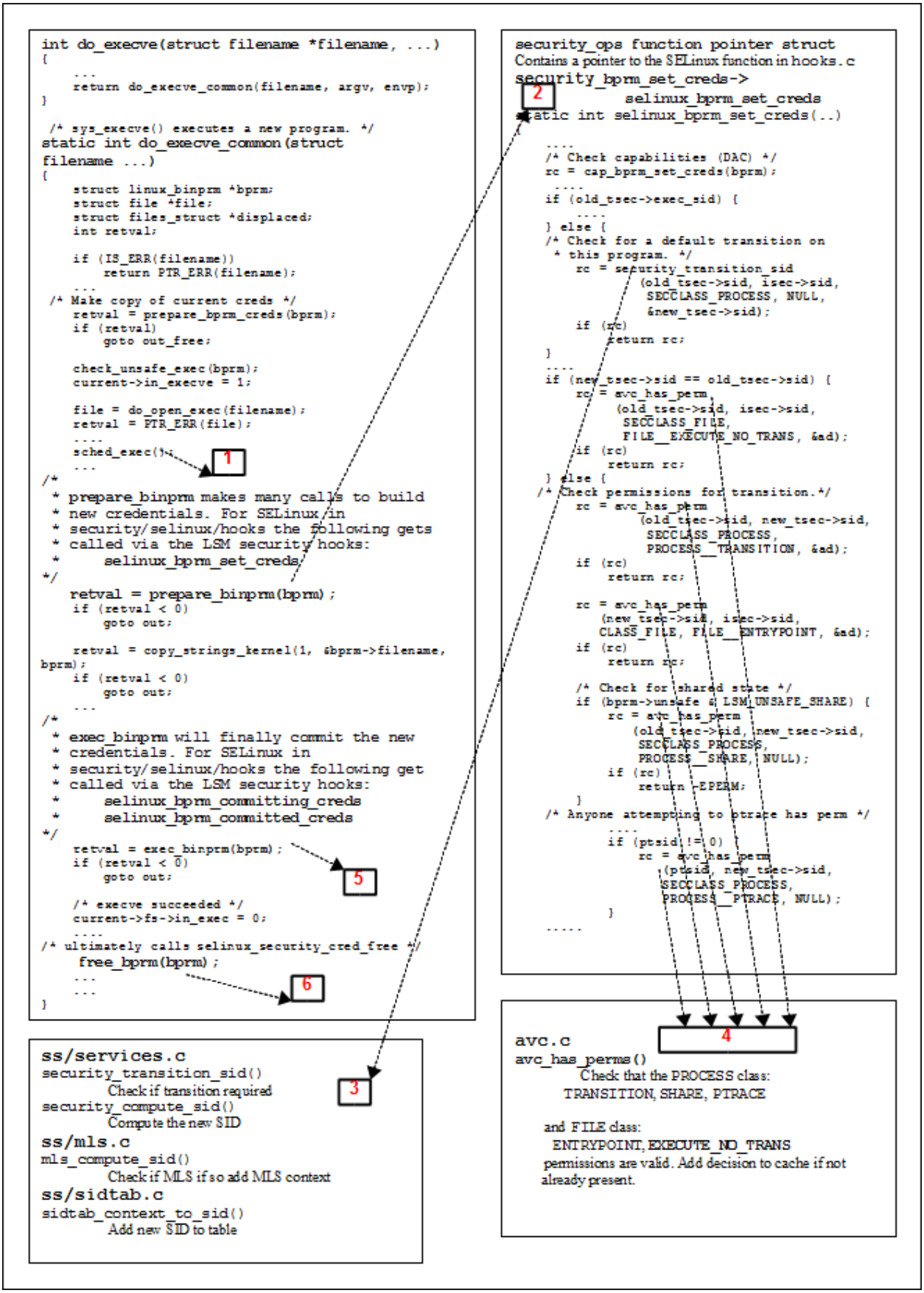

int do_execve(struct filename *filename, ...)
{
...
return do_execve_common(filename, argv, envp);
}
/* sys_execve() executes a new program. */
static int do_execve_common(struct filename ...)
{
struct linux_binprm *bprm;
struct file *file;
struct files_struct *displaced;
int retval;
if (IS_ERR(filename))
return PTR_ERR(filename);
...
/* Make copy of current creds */
retval = prepare_bprm_creds(bprm);
if (retval)
goto out_free;
check_unsafe_exec(bprm);
current->in_execve = 1;
file = do_open_exec(filename);
retval = PTR_ERR(file);
....
sched_exec();
...
/*
* prepare_binprm makes many calls to build
* new credentials. For SELinux/in
* security/selinux/hooks the following gets
* called via the LSM security hooks:
* selinux_bprm_set_creds
*/
retval = prepare_binprm(bprm);
if (retval < 0)
goto out;
retval = copy_strings_kernel(1, &bprm->filename,
bprm);
if (retval < 0)
goto out;
...
/*
* exec_binprm will finally commit the new
* credentials. For SELinux in
* security/selinux/hooks the following get
* called via the LSM security hooks:
* selinux_bprm_committing_creds
* selinux_bprm_committed_creds
*/
retval = exec_binprm(bprm);
if (retval < 0)
goto out;
/* execve succeeded */
current->fs->in_exec = 0;
....
/* ultimately calls selinux_security_cred_free */
free_bprm(bprm);
...
...
}
1
5
6
security_ops function pointer struct
Contains a pointer to the SELinux function in hooks.c
security_bprm_set_creds->
selinux_bprm_set_creds
2
static int selinux_bprm_set_creds(..)
....
/* Check capabilities (DAC) */
rc = cap_bprm_set_creds(bprm);
....
if (old_tsec->exec_sid) {
....
} else {
/* Check for a default transition on
* this program. */
rc = security_transition_sid
(old_tsec->sid, isec->sid,
SECCLASS_PROCESS, NULL,
&new_tsec->sid);
if (rc)
return rc;
}
....
if (new_tsec->sid == old_tsec->sid) {
rc = avc_has_perm
(old_tsec->sid, isec->sid,
SECCLASS_FILE,
FILE__EXECUTE_NO_TRANS, &ad);
if (rc)
return rc;
} else {
/* Check permissions for transition.*/
rc = avc_has_perm
(old_tsec->sid, new_tsec->sid,
SECCLASS_PROCESS,
PROCESS__TRANSITION, &ad);
if (rc)
return rc;
rc = avc_has_perm
(new_tsec->sid, isec->sid,
CLASS_FILE, FILE__ENTRYPOINT, &ad);
if (rc)
return rc;
/* Check for shared state */
if (bprm->unsafe & LSM_UNSAFE_SHARE) {
rc = avc_has_perm
(old_tsec->sid, new_tsec->sid,
SECCLASS_PROCESS,
PROCESS__SHARE, NULL);
if (rc)
return -EPERM;
}
/* Anyone attempting to ptrace has perm */
....
if (ptsid != 0) {
rc = avc_has_perm
(ptsid, new_tsec->sid,
SECCLASS_PROCESS,
PROCESS__PTRACE, NULL);
}
.....
ss/services.c
security_transition_sid()
Check if transition required
security_compute_sid()
Compute the new SID
ss/mls.c
mls_compute_sid()
Check if MLS if so add MLS context
ss/sidtab.c
sidtab_context_to_sid()
Add new SID to table
3
avc.c
avc_has_perms()
Check that the PROCESS class:
TRANSITION, SHARE, PTRACE
and FILE class:
ENTRYPOINT, EXECUTE_NO_TRANS
permissions are valid. Add decision to cache if not
already present.
4

Figure 11: Process Transition - *This shows the major steps required to check if a transition is allowed from the unconfined_t domain to the ext_gateway_t domain.*

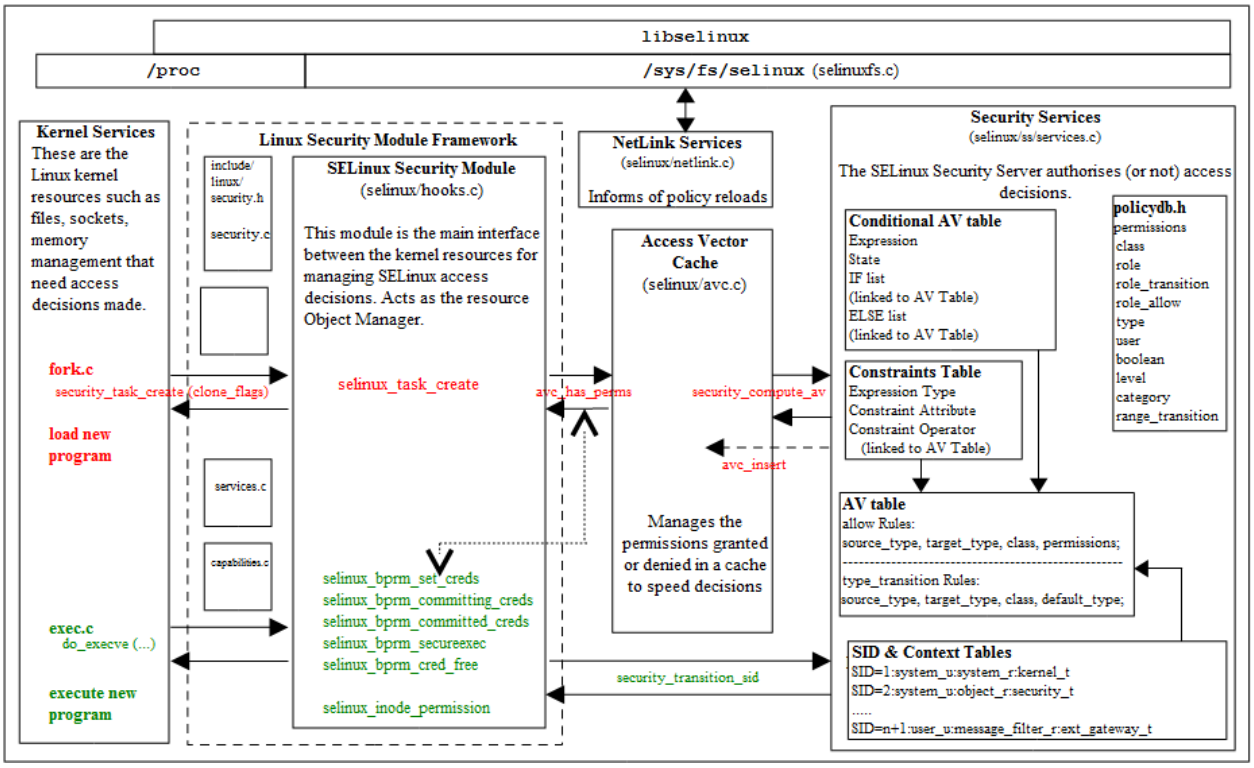


Figure 12: The Main LSM / SELinux Modules - *The fork and exec functions link to* ***Figure 7: Domain Transition*** *where the transition process is described.*

SELinux Filesystem

The table below shows the information contained in the SELinux filesystem (*selinuxfs*) */sys/fs/selinux* where the SELinux kernel service exports information regarding its configuration and active policy. *selinuxfs* is a read/write interface used by SELinux library functions for userspace SELinux-aware applications and object managers. Note: while it is possible for userspace applications to read/write to this interface, it is not recommended - use the ***libselinux*** or *libsepol* library.

/sys/fs/selinux - Directory

- This is the root directory where the SELinux kernel exports relevant information regarding its configuration and active policy for use by the libselinux library.

access

- Compute access decision interface that is used by the ***security_compute_av****(3)*,***security_compute_av_flags****(3)*, ***avc_has_perm****(3)* and ***avc_has_perm_noaudit****(3)* functions. The kernel security server (see *security/services.c*) converts the contexts to SIDs and then calls the *security_compute_av_user* function to compute the new SID that is then converted to a context string. Requires *security { compute_av }* permission.

checkreqprot

- *0* = Check protection applied by kernel (default since kernel v4.4). *1* = Check protection requested by application. These apply to the *mmap* and *mprotect* kernel calls. Default value can be changed at boot time via the *checkreqprot*= parameter. Requires *security { setcheckreqprot }* permission. Note *checkreqprot* will be

deprecated at some stage, with the default set to 0. See https://github.com/SELinuxProject/selinux-kernel/wiki/DEPRECATE-checkreqprot

commit_pending_bools

- Commit new boolean values to the kernel policy. Requires *security { setbool }* permission.

context

- Validate context interface used by the ***security_check_context**(3)* function. Requires *security { check_context }* permission.

create

- Compute create labeling decision interface that is used by the ***security_compute_create**(3)* and ***avc_compute_create**(3)* functions. The kernel security server (see *security/selinux/ss/services.c*) converts the contexts to SIDs and then calls the *security_transition_sid_user* function to compute the new SID that is then converted to a context string. Requires *security { compute_create }* permission.

deny_unknown, *reject_unknown*

- These two files export *deny_unknown*, (read by the ***security_deny_unknown**(3)* function) and *reject_unknown* status to user space. These are taken from the *handle-unknown* parameter set in the ***/etc/selinux/semanage.conf*** when policy is being built and are set as follows:
  - *deny:reject*
  - 0:0 = Allow unknown object class / permissions. This will set the returned AV with all 1's.
  - 1:0 = Deny unknown object class / permissions (the default). This will set the returned AV with all 0's.
  - 1:1 = Reject loading the policy if it does not contain all the object classes / permissions.

disable

- Disable SELinux until next reboot.

enforce

- Get or set enforcing status. Requires *security { setenforce }* permission.

load

- Load policy interface. Requires *security { load_policy }* permission.

member

- Compute polyinstantiation membership decision interface that is used by the ***security_compute_member**(3)* and ***avc_compute_member**(3)* functions. The kernel security server (see *security/selinux/ss/services.c*) converts the contexts to SIDs and then calls the *security_member_sid()* function to compute the new SID that is then converted to a context string. Requires *security { compute_member }* permission.

mls

- Returns 1 if MLS policy is enabled or 0 if not.

null

- The SELinux equivalent of */dev/null* for file descriptors that have been redirected by SELinux.

policy

- Interface to upload the current running policy in kernel binary format. This is useful to check the running policy using ***apol**(1)*, *dispol*/*sedispol* etc. (e.g. *apol /sys/fs/selinux/policy*).

policyvers

- Returns supported policy version for kernel. Read by ***security_policyvers****(3)* function.

relabel

- Compute relabeling decision interface that is used by the ***security_compute_relabel****(3)* function. The kernel security server (see *security/selinux/ss/services.c*) converts the contexts to SIDs and then calls the *security_change_sid()* function to compute the new SID that is then converted to a context string. Requires *security { compute_relabel }* permission.

status

- This can be used to obtain enforcing mode and policy load changes with much less over-head than using the *libselinux* netlink / call backs. This was added for Object Managers that have high volumes of AVC requests so they can quickly check whether to invalidate their cache or not. The status structure indicates the following:
  - *version* - Version number of the status structure. This will increase as other entries are added.
  - *sequence* - This is incremented for each event with an even number meaning that the events are stable. An odd number indicates that one of the events is changing and therefore the userspace application should wait before reading the status of any event.
  - *enforcing - 0* = Permissive mode, *1* = Enforcing mode.
  - *policyload* - This contains the policy load sequence number and should be read and stored, then compared to detect a policy reload.
  - *deny_unknown - 0* = Allow and *1* = Deny unknown object classes / permissions. This is the same as the *deny_unknown* entry above.

user

- Compute reachable user contexts interface that is used by the deprecated ***security_compute_user****(3)* function. The kernel security server (see *security/selinux/ss/services.c*) converts the contexts to SIDs and then calls the *security_get_user_sids()* function to compute the user SIDs that are then converted to context strings. Requires *security { compute_user }* permission.

validatetrans

- Compute *validatetrans* decision interface. Requires *security { validate_trans }* permission. Used by the ***security_validatetrans****(3)* function.

/sys/fs/selinux/avc - Directory

- This directory contains information regarding the kernel AVC that can be displayed by the ***avcstat****(8)* command.

avc/cache_stats

- Shows the kernel AVC lookups, hits, misses etc.

avc/cache_threshold

- The default value is 512, however caching can be turned off (but performance suffers) by: *echo 0 > /selinux/avc/cache_threshold* Requires *security { setsecparam }* permission.

avc/hash_stats

- Shows the number of kernel AVC entries, longest chain etc.

/sys/fs/selinux/booleans - Directory

- Contains a file named after each boolean defined by policy.

booleans/<boolean_name>

- Each file contains the current and pending status of the boolean (0 = false or 1 = true). The ***getsebool****(8)*, ***setsebool****(8)* and ***sestatus****(8)* commands use this interface via the *libselinux* library functions.

/sys/fs/selinux/initial_contexts - Directory

- Contains a file named after each initial SID defined by policy.

initial_contexts/<initial_sid>

- Each file contains the initial SID context as defined by policy (e.g. *any_socket* context: *system_u:object_r:unlabeled_t:s0*).

/sys/fs/selinux/policy_capabilities - Directory

- Contains a file named after each policy capability defined by policy using the *policycap* statement. Their default values are false.

policy_capabilities/always_check_network

- If true SECMARK and NetLabel peer labeling are always enabled even if there are no SECMARK, NetLabel or Labeled IPsec rules configured. This forces checking of the *packet* class to protect the system should any rules fail to load or they get maliciously flushed. Requires kernel 3.13 minimum.

policy_capabilities/cgroup_seclabel

- Allows userspace to set labels on cgroup/cgroup2 files, enabling fine-grained labeling of cgroup files by userspace. Requires kernel 4.11 minimum.

policy_capabilities/extended_socket_class

- Enables the use of separate socket security classes for all network address families rather than the generic socket class.

policy_capabilities/genfs_seclabel_symlinks

- Enables fine-grained labeling of symlinks in pseudo filesystems based on *genfscon* rules.

policy_capabilities/ioctl_skip_cloexec

- If true always allow FIOCLEX and FIONCLEXE ioctl permissions (from kernel 5.18).

policy_capabilities/network_peer_controls

- If true the following *network_peer_controls* are enabled:
  - *node { sendto recvfrom }*
  - *netif { ingress egress }*
  - *peer { recv }*

policy_capabilities/nnp_nosuid_transition

- Enables SELinux domain transitions to occur under *no_new_privs* (NNP) or on *nosuid* mounts if the corresponding permission (*nnp_transition* for NNP, *nosuid_transition* for *nosuid*, defined in the *process2* security class) is allowed between the old and new contexts.

policy_capabilities/open_perms

- If true the *open* permission is enabled by default on the following object classes: *dir*, *file*, *fifo_file*, *chr_file*, *blk_file*.

/sys/fs/selinux/class - Directory

- Contains a directory named after each class defined by policy.

class/<object_class> - Directory

- Each class directory contains an *index* file and a *perms* directory.

class/<object_class>/index

- This file contains an allocated class number that is derived from the policy source order (e.g. alg_socket is the 124th class entry in the policy taken from the Fedora *policy/flask/security_classes* file (or 121st for the Reference Policy)).

class/<object_class>/perms - Directory

- Contains one file named after each permission defined by policy.

class/<object_class>/perms/<perm_name>

- Each file is named by the permission assigned in policy and contains an allocated permission number that is derived from the policy source order (e.g. *accept* is the 15th permission from the Fedora or Reference Policy *policy/flask/access_vector* file for the *alg_socket*).

Notes:

1. Kernel SIDs are not passed to userspace only the context strings.
2. The */proc* filesystem exports the process security context string to userspace via */proc/<self|pid>/attr* and */proc/<self|pid>/task/<tid>/attr/<attr>* interfaces.

SELinux Userspace Libraries

The versions of kernel and SELinux tools and libraries influence the features available, therefore it is important to establish what level of functionality is required for the application. The **Policy Versions** section shows the policy versions and the additional features they support.

Details of the libraries, core SELinux utilities and commands with source code are available at:

https://github.com/SELinuxProject/selinux/wiki

libselinux Library

libselinux contains all the SELinux functions necessary to build userspace SELinux-aware applications and object managers using 'C', Python, Ruby and PHP languages.

The library hides the low level functionality of (but not limited to):

- The SELinux filesystem that interfaces to the SELinux kernel security server.
- The proc filesystem that maintains process state information and security contexts - see ***proc****(5)*.
- Extended attribute services that manage the extended attributes associated to files, sockets etc. - see ***attr****(5)*.
- The SELinux policy and its associated configuration files.

The general category of functions available in *libselinux* are shown below, with **Appendix B - *libselinux* API Summary** giving a complete list of functions.

Access Vector Cache Services

Allow access decisions to be cached and audited.

Boolean Services

Manage booleans.

Class and Permission Management

Class / permission string conversion and mapping.

Compute Access Decisions

Determine if access is allowed or denied.

Compute Labeling

Compute labels to be applied to new instances of on object.

Default File Labeling

Obtain default contexts for file operations.

File Creation Labeling

Get and set file creation contexts.

File Labeling

Get and set file and file descriptor extended attributes.

General Context Management

Check contexts are valid, get and set context components.

Key Creation Labeling

Get and set kernel key creation contexts.

Label Translation Management

Translate to/from, raw/readable contexts.

Netlink Services

Used to detect policy reloads and enforcement changes.

Process Labeling

Get and set process contexts.

SELinux Management Services

Load policy, set enforcement mode, obtain SELinux configuration information.

SELinux-aware Application Labeling

Retrieve default contexts for applications such as database and X-Windows.

Socket Creation Labeling

Get and set socket creation contexts.

User Session Management

Retrieve default contexts for user sessions.

The *libselinux* functions make use of a number of files within the SELinux sub-system:

1. The SELinux configuration file *config* that is described in the *[/etc/selinux/config](#)* section.
2. The SELinux filesystem interface between userspace and kernel that is generally mounted as */selinux* or */sys/fs/selinux* and described in the **SELinux Filesystem** section.
3. The *proc* filesystem that maintains process state information and security contexts - see ***proc**(5)*.
4. The extended attribute services that manage the extended attributes associated to files, sockets etc. - see ***attr**(5)*.
5. The SELinux kernel binary policy that describes the enforcement policy.
6. A number of *libselinux* functions have their own configuration files that in conjunction with the policy, allow additional levels of configuration. These are described in the **Policy Configuration Files** section.

There is a static version of the library that is not installed by default:

```
dnf install libselinux-static
```

libsepol Library

libsepol - To build and manipulate the contents of SELinux kernel binary policy files.

There is a static version of the library that is not installed by default:

```
dnf install libsepol-static
```

This is used by commands such as ***audit2allow****(8)* and ***checkpolicy****(8)* as they require access to functions that are not available in the dynamic library, such as *sepol_compute_av()*, *sepol_compute_av_reason()* and *sepol_context_to_sid()*.

libsemanage Library

libsemanage - To manage the policy infrastructure.

SELinux Networking Support

SELinux controls network access in the kernel at two locations: at the socket interface, and when packets are processed by the protocol stacks. Controls at the socket interface are invoked when a task makes network related system calls and thus the access permission checks mimic the sockets programming interface (e.g. ***bind****(2)* vs. *node_bind*). Packet controls are more distant from applications and they are invoked whenever any packets are received, forwarded or sent.

Packet level controls include:

- Packet labeling with SECMARK: class *packet*
- Peer labeling with Labeled IPSec or NetLabel: class *peer*
- Interface control: class *netif*
- Network node control: class *node*

Controls at socket interface include:

- TCP/UDP/SCTP/DCCP ports: class *port*

SECMARK is a security extension in Linux network packet processing, which allows packets or sessions to be marked with security labels.

NetLabel is a framework for explicit network labeling that abstracts away the underlying labeling protocol from the LSMs. Labeled IPsec and the NetLabel framework are the current access controls for class *peer*, with NetLabel supporting CIPSO for IPv4, CALIPSO for IPv6, and a fallback/static labeling for unlabeled IPv4 and IPv6 networks.

Packet controls can be organized further according to the source of the label for the packets, which can be internal or external:

Internal labeling - This is where network objects are labeled and managed internally within a single machine (i.e. their labels are not transmitted as part of the session with remote systems). There are two types supported: SECMARK and NetLabel with the static/fallback “protocol”. As an example, SECMARK access controls could restrict *firefox_t* to talking only to network services on TCP port 80 while NetLabel fallback/static rules could restrict *firefox_t* to only receive data from specific IP addresses on a specific network interface.

Labeled Networking - This is where labels are passed to/from remote systems where they can be interpreted and a MAC policy enforced on each system. There are three types supported: Labeled IPSec, NetLabel/CIPSO (Commercial IP Security Option) and NetLabel/CALIPSO (Common Architecture Label IPv6 Security Option). With labeled networking, it’s possible to compare the security attributes (SELinux label) of the sending peer with the security context of the receiving peer. A simple example with Labeled IPSec is restricting *firefox_t* to talking only to *httpd_t*, while NetLabel/CIPSO & CALIPSO could restrict domains with MLS security level *s32* not to talk with domains with level *s31*.

SELinux network access controls are not enabled by default. They can be enabled by configuring SECMARK, NetLabel or Labeled IPsec rules, or forced on with the policy capability ***always_check_network***.

There are three policy capability options that can be set within policy using the *policycap* statement that affect networking configuration:

network_peer_controls - This is always enabled in the latest Reference Policy source. **Figure 14: Fallback Labeling** shows the difference between the policy capability being set to 0 and 1.

always_check_network - This capability would normally be set to false. If true SECMARK and NetLabel peer labeling are always enabled even if there are no SECMARK, NetLabel or Labeled IPsec rules configured. This forces checking of the *packet* class to protect the system should any rules fail to load or they get maliciously flushed. Requires kernel 3.13 minimum.

extended_socket_class - Enable separate security classes for all network address families previously mapped to the *socket* class and for ICMP and SCTP sockets previously mapped to the *rawip_socket* class. Requires kernel 4.11+.

The policy capability settings are available in userspace via the SELinux filesystem as shown in the **SELinux Filesystem** section.

The NetLabel tools need to be installed to support peer labeling with CIPSO and CALIPSO or fallback labeling:

```
dnf install netlabel_tools
```

To support Labeled IPSec the IPSec tools need to be installed:

```
dnf install ipsec-tools
```

It is also possible to use an alternative Labeled IPSec service that was OpenSwan but is now distributed as LibreSwan:

```
dnf install libreswan
```

It is important to note that the kernel must be configured to support these services (*CONFIG_NETLABEL*, *CONFIG_NETWORK_SECMARK*, *CONFIG_NF_CONNTRACK_SECMARK*, *CONFIG_NETFILTER_XT_TARGET_CONNSECMARK*, *CONFIG_NETFILTER_XT_TARGET_SECMARK*, *CONFIG_IP_NF_SECURITY*, *CONFIG_IP6_NF_SECURITY*). At least Fedora and Debian kernels are configured to handle all the above services.

The Linux networking package *iproute* has an SELinux aware socket statistics command ***ss**(8)* that will show the SELinux context of network processes (*-Z* or *–context* option) and network sockets (*-z* or *–contexts* option). Note that the socket contexts are taken from the inode associated to the socket and not from the actual kernel socket structure (as currently there is no standard kernel/userspace interface to achieve this).

Packet controls: SECMARK

SECMARK makes use of the standard kernel NetFilter framework that underpins the GNU / Linux IP networking sub-system. NetFilter services automatically inspects all incoming and outgoing packets and can place controls on interfaces, IP addresses (nodes) and ports with the added advantage of connection tracking. The SECMARK security extensions allow security contexts to be added to packets (SECMARK) or sessions (CONNSECMARK), belonging to object class of *packet*.

The NetFilter framework inspects and tag packets with labels as defined within ***iptables**(8)* (also 'nftables' ***nft**(8)* from version 9.3 with kernel 4.20) and then uses the security framework (e.g. SELinux) to enforce the policy rules. Therefore SECMARK services are not SELinux specific as other security modules using the LSM infrastructure could also implement the same services (e.g. SMACK).

While the implementation of iptables / NetFilter is beyond the scope of this Notebook, there are tutorials available. The *selinux-testsuite inet_socket* and *sctp* tests have examples of *iptables* and *nftables* using SECMARK, some of which are shown below.

Figure 13: SECMARK Processing shows the basic structure with the process working as follows:

- A table called the *security table* is used to define the parameters that identify and 'mark' packets that can then be tracked as the packet travels through the networking sub-system. These 'marks' are called SECMARK and CONNSECMARK.
- A SECMARK is placed against a packet if it matches an entry in the security table applying a label that can then be used to enforce policy on the packet.
- The CONNSECMARK 'marks' all packets within a session[7] with the appropriate label that can then be used to enforce policy.

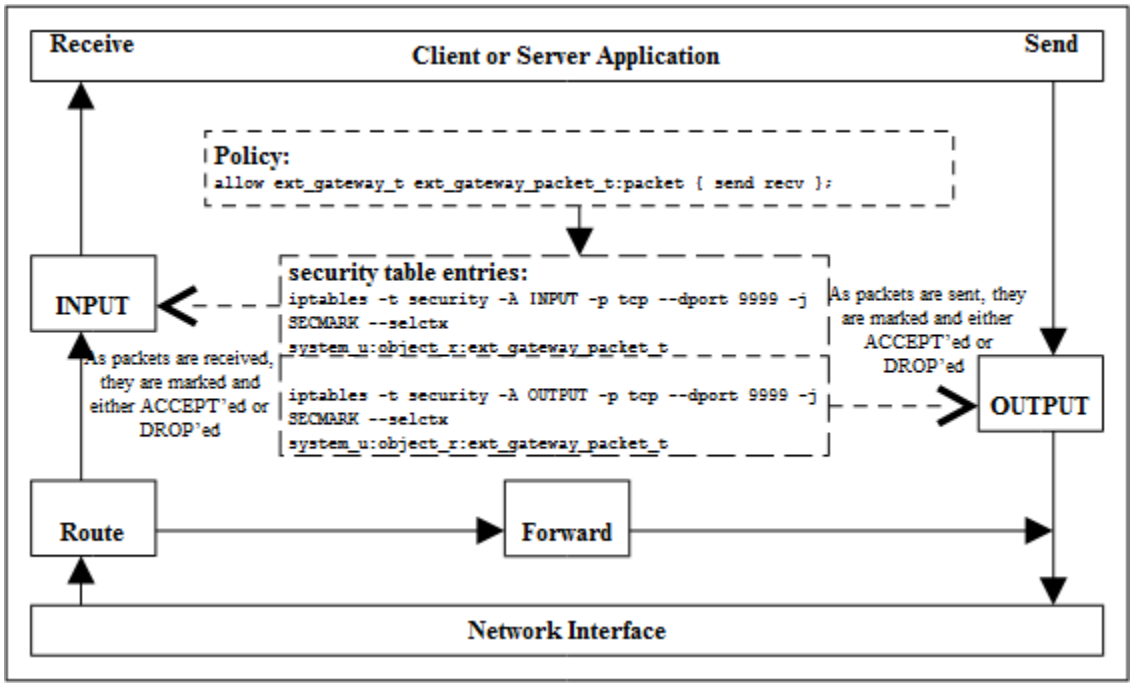


Figure 13: SECMARK Processing - *Received packets are processed by the INPUT chain where labels are added to the appropriate packets that will either be accepted or dropped by the SECMARK process. Packets being sent are treated the same way.*

Example iptables and nftables entries are shown below. Note that the tables will not load correctly if the policy does not allow the ip/nftables domain to relabel the security table entries unless permissive mode is enabled (i.e. the tables must have the relabel permission for each entry in the table).

An ipv4 *security table iptables* entry is as follows:

```
# Flush the security table.
iptables -t security -F

# Create a chain for new connection marking.
iptables -t security -N NEWCONN

# Accept incoming connections, label SYN packets, and copy labels to connections.
iptables -t security -A INPUT -i lo -p tcp --dport 65535 -m state --state NEW -j
NEWCONN
iptables -t security -A NEWCONN -j SECMARK --selctx
system_u:object_r:test_server_packet_t:s0
iptables -t security -A NEWCONN -j CONNSECMARK --save
iptables -t security -A NEWCONN -j ACCEPT
```

```
# Common rules which copy connection labels to established and related packets.
iptables -t security -A INPUT -m state --state ESTABLISHED,RELATED -j CONNSECMARK --
restore
iptables -t security -A OUTPUT -m state --state ESTABLISHED,RELATED -j CONNSECMARK --
restore

# Label UDP packets similarly.
iptables -t security -A INPUT -i lo -p udp --dport 65535 -j SECMARK --selctx
system_u:object_r:test_server_packet_t:s0
iptables -t security -A OUTPUT -o lo -p udp --sport 65535 -j SECMARK --selctx
system_u:object_r:test_server_packet_t:s0
```

The equivalent *nftable* entry for an ipv6 *security table* is as follows:

```
add table ip6 security

table ip6 security {
    secmark inet_server {
        "system_u:object_r:test_server_packet_t:s0"
    }
    map secmapping_in_out {
        type inet_service : secmark
        elements = { 65535 : "inet_server" }
    }
    chain input {
        type filter hook input priority 0;

        ct state new meta secmark set tcp dport map @secmapping_in_out
        ct state new meta secmark set udp dport map @secmapping_in_out
        ct state new ct secmark set meta secmark

        ct state established,related meta secmark set ct secmark
    }
    chain output {
        type filter hook output priority 0;

        ct state new meta secmark set tcp dport map @secmapping_in_out
        ct state established meta secmark set udp dport map @secmapping_in_out
        ct state new ct secmark set meta secmark

        ct state established,related meta secmark set ct secmark
    }
}
```

Before the SECMARK rules can be loaded, TE rules must be added to define the types, and also allow domains to send and/or receive objects of *packet* class:

```
type test_server_packet_t, packet_type;

allow my_server_t test_server_packet_t:packet { send recv };
```

The following articles explain the SECMARK service:

- *Transitioning to Secmark*
- New secmark-based network controls for SELinux

Packet controls: NetLabel - Fallback Peer Labeling

Fallback labeling can optionally be implemented on a system if the Labeled IPSec or CIPSO/CALIPSO is not being used (hence 'fallback labeling'). If either Labeled IPSec or CIPSO/CALIPSO are being used, then these take priority. There is an article "Fallback Label Configuration Example" that explains their usage, the ***netlabelctl****(8)* man page is also a useful reference.

The network peer controls have been extended to support an additional object class of *peer* that is enabled by default in the Fedora policy as the *network_peer_controls* in */sys/fs/selinux/policy_capabilities* is set to '*1*'. **Figure 14: Fallback Labeling** shows the differences between the policy capability *network_peer_controls* being set to 0 and 1.

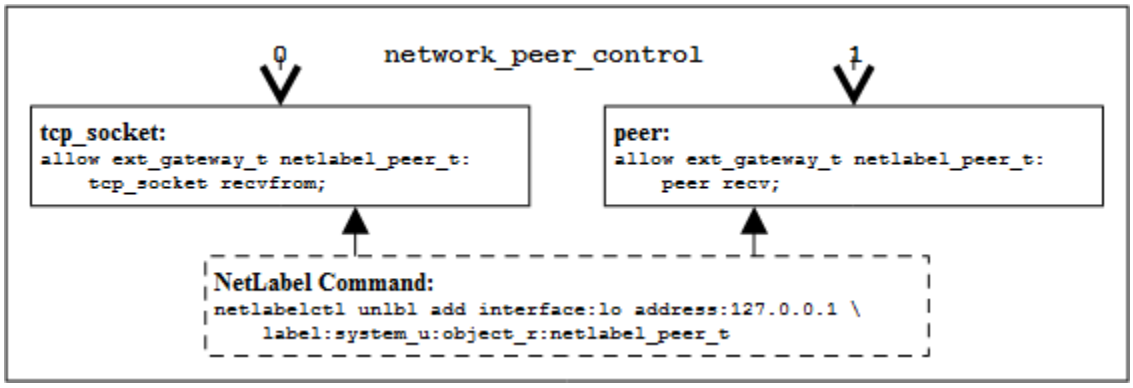


Figure 14: Fallback Labeling - *Showing the differences between the policy capability* **network_peer_controls*** set to 0 and 1.*

The *selinux-testsuite inet_socket* and *sctp* tests have examples of fallback labeling, and the following are a set of ***netlabelctl****(8)* commands from the *sctp* test:

```
netlabelctl map del default
netlabelctl map add default address:0.0.0.0/0 protocol:unlbl
netlabelctl map add default address:::/0 protocol:unlbl
netlabelctl unlbl add interface:lo address:127.0.0.0/8
label:system_u:object_r:netlabel_sctp_peer_t:s0
netlabelctl unlbl add interface:lo address:::1/128
label:system_u:object_r:deny_assoc_sctp_peer_t:s0
# Display Netlabel configuration:
netlabelctl -p map list
```

Note that the security contexts must be valid in the policy otherwise the commands will fail.

Before the NetLabel rules can be loaded, TE rules must be added to define the types. Then the rules can allow domains to receive data from objects of *peer* class:

```
type netlabel_sctp_peer_t;

allow my_server_t netlabel_sctp_peer_t:peer recv;
```

Note that sending can't be controlled with *peer* class.

Packet controls: NetLabel – CIPSO/CALIPSO

To allow MLS or MCS **security levels** to be passed over a network between MLS or MCS systems[8], the Commercial IP Security Option (CIPSO) protocol is used. This is defined in the **CIPSO Internet Draft** document (this is an obsolete document, however the protocol is still in use). The protocol defines how security levels are encoded in the IP packet header for IPv4.

The Common Architecture Label IPv6 Security Option (CALIPSO) protocol described in **rfc 5570** is supported in kernels from 4.8.

CIPSO/CALIPSO will only pass the MLS or MCS component of the security context over the network, however in loopback mode CIPSO allows the full security context to be passed as explained in the "**Full SELinux Labels Over Loopback with NetLabel and CIPSO**" available at http://paulmoore.livejournal.com/7234.html.

The protocol is configured by the NetLabel service ***netlabelctl**(8)* and can be used by other security modules that use the LSM infrastructure. The implementation supports:

1. A non-translation option (Tag Type 1 '**bit mapped**') where labels are passed to / from systems unchanged (for host to host communications as show in **Figure 15**).
    - Note that CALIPSO only supports this option, and an example ***netlabelctl**(8)* command setting a DOI of 16 is:

```
netlabelctl calipso add pass doi:16
```

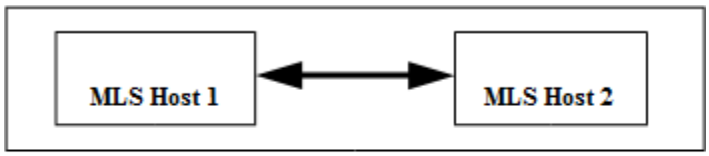


Figure 15: - *MLS Systems on the same network*

1. Allow a maximum of 256 sensitivity levels and 240 categories to be mapped (Tag Type 2 '**enumerated**').
2. A translation option (Tag Type 5 '**range**') where both the sensitivity and category components can be mapped for systems that have either different definitions for labels or information can be exchanged over different networks (for example using an SELinux enabled gateway as a guard as shown in **Figure 16**. An example ***netlabelctl**(8)* command setting a DOI of 8 is: *netlabelctl cipsov4 add pass doi:8 tags:5*

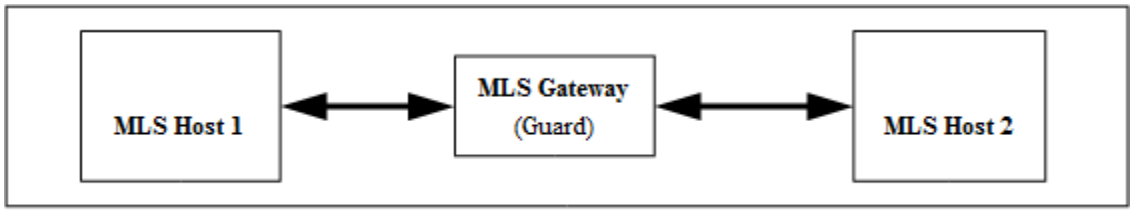


Figure 16: - *MLS Systems on different networks communicating via a gateway*

There are CIPSO/CALIPSO examples in the ***notebook-examples/network/netlabel*** section. The CALIPSO example ***netlabelctl**(8)* commands for loopback are:

```
netlabelctl calipso add pass doi:12345678
netlabelctl map del default
netlabelctl map add default address:0.0.0.0/0 protocol:unlbl
netlabelctl map add default address:::/0 protocol:unlbl
netlabelctl map add default address:::1 protocol:calipso,12345678
# Display Netlabel configuration:
netlabelctl -p map list
```

The examples use the *nb_client*/*nb_server* from the Notebook examples section, plus the standard Fedora 'targeted' policy for the tests.

Packet controls: Labeled IPSec

Labeled IPSec has been built into the standard GNU / Linux IPSec services as described in the "**Leveraging IPSec for Distributed Authorization**".

Figure 17: IPSec communications shows the basic components that form the service based on IPSec tools where it is generally used to set up either an encrypted tunnel between two machines (a virtual private network (VPN)) or an encrypted transport session. The extensions defined in the **Leveraging IPSec for Distributed Authorization** document describe how the security context is configured and negotiated between the two systems (called security associations (SAs) in IPSec terminology).

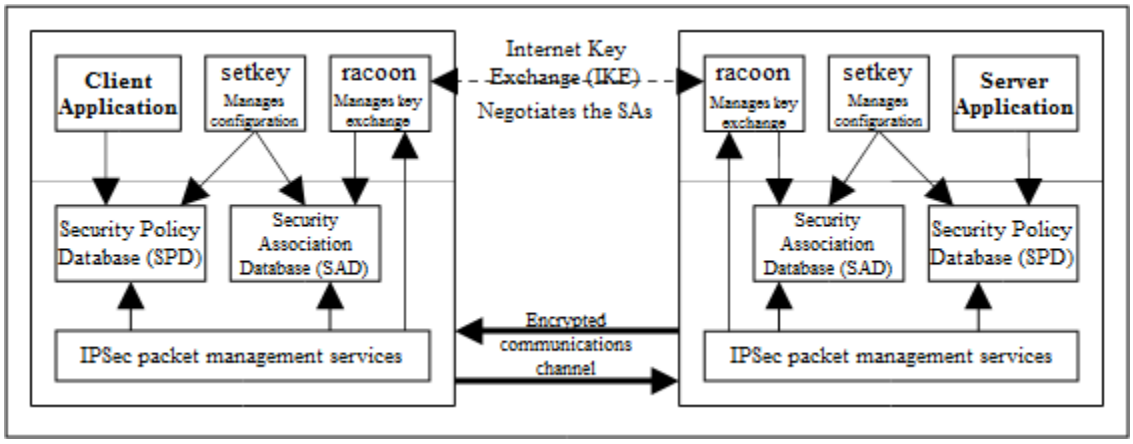


Figure 17: IPSec communications - *The SPD contains information regarding the security contexts to be used. These are exchanged between the two systems as part of the channel set-up.*

Basically what happens is as follows:

1. The security policy database (SPD) defines the security communications characteristics to be used between the two systems. This is populated using the **setkey**(8) or ***ip-xfrm****(8)* commands.
2. The SAs have their configuration parameters such as protocols used for securing packets, encryption algorithms and how long the keys are valid held in the Security Association database (SAD). For Labeled IPSec the security context (or labels) is also defined within the SAD. SAs can be negotiated between the two systems using either racoon or pluto[9] that will automatically populate the SAD or manually by the setkey utility (see the example below).
3. Once the SAs have been negotiated and agreed, the link should be active.

A point to note is that SAs are one way only, therefore when two systems are communicating (using the above example), one system will have an SA, SAout for processing outbound packets and another SA, SAin, for processing the inbound packets. The other system will also create two SAs for processing its packets.

Each SA will share the same cryptographic parameters such as keys and protocol (e.g. ESP (encapsulated security payload) and AH (authentication header)).

The object class used for the association of an SA is association and the permissions available are as follows:

polmatch

- Match the SPD context (-ctx) entry to an SELinux domain (that is contained in the SAD -ctx entry)

recvfrom

- Receive from an IPSec association.

sendto

- Send to an IPSec association.

setcontext

- Set the context of an IPSec association on creation (e.g. when running ***setkey**(8)* the process will require this permission to set the context in the SAD and SPD, also *racoon* / *pluto* will need this permission to build the SAD).

When running Labeled IPSec it is recommended that the systems use the same type/version of policy to avoid any problems with them having different meanings.

There are two possible labeled IPSec solutions available on Fedora:

1. IPSec Tools - This uses the ***setkey**(8)* tools and ***racoon**(8)* Internet Key Exchange (IKE) daemon.
2. LibreSwan - This uses ***ipsec**(8)* tools and ***pluto**(8)* Internet Key Exchange (IKE) daemon. Note that the Fedora build uses the kernel *netkey* services as Libreswan can be built to support other types.

Both work in much the same way but use different configuration files.

If interoperating between pluto and racoon, note that the DOI Security Context type identifier has never been defined in any standard. Pluto is configurable and defaults to '*32001*', this is the IPSEC Security Association Attribute identifier reserved for private use. Racoon is hard coded to a value of '*10*', therefore the pluto ***ipsec.conf**(5)* configuration file *secctx-attr-type* entry must be set as shown in the following example:

```
config setup
    protostack=netkey
    plutodebug=all
    logfile=/var/log/pluto/pluto.log
    logappend=no
    # A "secctx-attr-type" MUST be present:
    secctx-attr-type=10
    # Labeled IPSEC only supports the following values:
    #   10 = ECN_TUNNEL - Used by racoon(8)
    #   32001 = Default - Reserved for private use (see RFC 2407)
    # These are the "IPSEC Security Association Attributes"

conn selinux_labeled_ipsec_test
    # ikev2 MUST be "no" as labeled ipsec is not yet supported by IKEV2
    # There is a draft IKEV2 labeled ipsec document (July '20) at:
    #   https://tools.ietf.org/html/draft-ietf-ipsecme-labeled-ipsec-03
    ikev2=no
    auto=start
    rekey=no
    authby=secret   # set in '/etc/ipsec.secrets'. See NOTE
```

```
    type=transport
    left=192.168.1.198
    right=192.168.1.148
    ike=aes256-sha2     # See NOTE
    phase2=esp
    phase2alg=aes256    # See NOTE
    # The 'policy-label' entry is used to determine whether SELinux will
    # allow or deny the request using the labels from:
    #   connection policy label from the applicable SAD entry
    #   connection flow label from the applicable SPD entry (this is taken
    #   from the 'conn <name> policy-label' entry).
    # selinux_check_access(SAD, SPD, "association", "polmatch", NULL);
    policy-label=system_u:object_r:ipsec_spd_t:s0
    leftprotoport=tcp
    rightprotoport=tcp

# NOTE:
#   The authentication methods and encryption algorithms should be chosen
#   with care and within the constraints of those available for
#   interoperability.
#   Racoon is no longer actively supported and has a limited choice of
#   algorithms compared to LibreSwan.
```

The Fedora version of racoon has added functionality to support loopback, however pluto does not. The default for Fedora is that ipsec is disabled for loopback, however the following commands will enable racoon to work in loopback until a re-boot:

```
echo 0 > /proc/sys/net/ipv4/conf/lo/disable_xfrm
echo 0 > /proc/sys/net/ipv4/conf/lo/disable_policy
```

By default Fedora does not enable IPSEC via its default firewall configuration, therefore the server side requires the following command:

```
firewall-cmd --add-service ipsec
```

There are two simple examples in the ***notebook-examples/network/ipsec*** section. These use ***setkey****(8)* and commands directly and therefore do not require the IKE daemons.

The ***ip-xfrm****(8)* example:

```
echo 0 > /proc/sys/net/ipv4/conf/lo/disable_xfrm
echo 0 > /proc/sys/net/ipv4/conf/lo/disable_policy
ip xfrm policy flush
ip xfrm state flush

ip xfrm state add src 127.0.0.1 dst 127.0.0.1 proto ah spi 0x200 ctx
"unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c1023" auth sha1 0123456789012345
ip xfrm policy add src 127.0.0.1 dst 127.0.0.1 proto tcp dir out ctx
"system_u:object_r:ipsec_spd_t:s0" tmpl proto ah mode transport level required

ip xfrm policy show
ip xfrm state show
```

The examples use the *nb_client*/*nb_server* from the Notebook examples section, plus the standard Fedora 'targeted' policy for the tests.

There is a further example in the **Secure Networking with SELinux** http://securityblog.org/brindle/2007/05/28/secure-networking-with-selinux article and a good reference covering **Basic Labeled IPsec Configuration** available at: http://www.redhat.com/archives/redhat-lspp/2006-November/msg00051.html

Packet controls: Access Control for Network Interfaces

SELinux domains can be restricted to use only specific network interfaces. TE rules must define the interface types and then allow a domain to *egress* in class *netif* for the defined interface types:

```
require {
        attribute netif_type;
        class netif { egress ingress };
}

type external_if_t, netif_type;
type loopback_if_t, netif_type;

allow my_server_t external_if_t:netif egress;
allow my_server_t loopback_if_t:netif egress;
```

The interfaces must also be labeled with ***semanage****(8)* (or by using *netifcon* statements in the policy):

```
semanage interface -a -t loopback_if_t -r s0 lo
semanage interface -a -t external_if_t -r s0 eth0
```

The checks for *ingress* in class *netif* however use the peer label of the remote peer (not the receiving task on the local system) as subject:

```
type internet_peer_t;

allow internet_peer_t external_if_t:netif ingress;
```

The peers must also be labeled with various methods provided by ***netlabelctl****(8)*. A simple example with fallback/static labeling is:

```
netlabelctl unlbl add default address:2000::/3
label:system_u:object_r:internet_peer_t:s0
```

Packet controls: Access Control for Network Nodes

Domains can be restricted by SELinux to access and bind sockets to only dedicated network nodes (in practice, IP addresses).

The node types must be defined and then the node types can be used for TE rules as target context. TE rules to allow a domain to *sendto* for class *node* and to *node_bind* (for incoming connections) for class *tcp_socket*:

```
require {
        attribute node_type;
```

```
        class node { sendto recvfrom };
        class tcp_socket node_bind;
}

type loopback_node_t, node_type;
type internet_node_t, node_type;
type link_local_node_t, node_type;
type multicast_node_t, node_type;

allow my_server_t loopback_node_t:node sendto;
allow my_server_t loopback_node_t:tcp_socket node_bind;
allow my_server_t internet_node_t:node sendto;
```

After the types have been defined, corresponding node rules can be added with *semanage* (or *nodecon* statements):

```
semanage node -a -M /128 -p ipv6 -t loopback_node_t -r s0 ::1
semanage node -a -M /3 -p ipv6 -t internet_node_t -r s0 2000::
semanage node -a -M /8 -p ipv6 -t link_local_node_t -r s0 fe00::
semanage node -a -M /8 -p ipv6 -t multicast_node_t -r s0 ff00::
```

The checks for *recvfrom* in class *node* however use the peer label as subject:

```
allow internet_peer_t internet_node_t:node { recvfrom sendto };
```

Socket controls: Access Control for Network Ports

SELinux policy can also control access to ports used by various networking protocols such as TCP, UDP, SCTP and DCCP. In contrast to packet level controls, port controls are close to how networked applications use the socket system calls. Thus the controls typically involve checking if a task can perform an operation on a network socket, e.g. ***bind****(2)* would cause an access check for *node_bind*. These are usually easy to understand and don't require any special network configuration.

TE rules must define the port types and then allow a domain to *name_connect* (outgoing) or *name_bind* (incoming) in class *tcp_socket* (or *udp_socket* etc) for the defined port types:

```
require {
        attribute port_type;
        class tcp_socket { name_bind name_connect };
}

type my_server_port_t, port_type;

allow my_server_t my_server_port_t:tcp_socket name_connect;
allow my_server_t my_server_port_t:tcp_socket name_bind;
```

The ports must also be labeled with *semanage* (or *portcon* statements):

```
semanage port -a -t my_server_port_t -p tcp -r s0 12345
```

Only ports that fall outside the local, or ephemeral, port range are subject to the additional *name_bind* access check. You can see the current ephemeral port range on your system by checking the *net.ipv4.ip_local_port_range* sysctl:

```
sysctl net.ipv4.ip_local_port_range
```

Labeled Network FileSystem (NFS)

Version 4.2 of NFS supports labeling between client/server and requires the ***exports**(5)* | ***exportfs**(8)* '*security_label*' option to be set:

```
exportfs -o rw,no_root_squash,security_label localhost:$MOUNT
```

Labeled NFS requires kernel 3.14 and the following package installed:

```
dnf install nfs-utils
```

Labeled NFS clients must use a consistent security policy.

The *selinux-testsuite tools/nfs.sh* tests labeled NFS using various labels.

SELinux Virtual Machine Support

SELinux support is available in the KVM/QEMU and Xen virtual machine (VM) technologies[10] that are discussed in the sections that follow, however the package documentation should be read for how these products actually work and how they are configured.

Currently the main SELinux support for virtualisation is via *libvirt* that is an open-source virtualisation API used to dynamically load guest VMs. Security extensions were added as a part of the **Svirt** project and the SELinux implementation for the KVM/QEMU package (*qemu-kvm* and *libvirt* rpms) is discussed using some examples. The Xen product has Flask/TE services that can be built as an optional service, although it can also use the security enhanced *libvirt* services as well.

The sections that follow give an introduction to KVM/QEMU, then *libvirt* support with some examples using the Virtual Machine Manager to configure VMs, then an overview of the Xen implementation follows.

To ensure all dependencies are installed run:

```
dnf install libvirt qemu virt-manager
```

KVM / QEMU Support

KVM is a kernel loadable module that uses the Linux kernel as a hypervisor and makes use of a modified QEMU emulator to support the hardware I/O emulation. The "**Kernel-based Virtual Machine**" gives a good overview of how KVM and QEMU are implemented. It also provides an introduction to virtualisation in general. Note that KVM requires virtulisation support in the CPU (Intel-VT or AMD-V extensions).

The SELinux support for VMs is implemented by the *libvirt* sub-system that is used to manage the VM images using a Virtual Machine Manager, and as KVM is based on Linux it has SELinux support by default. There are also Reference Policy modules to support the overall infrastructure (KVM support is in various kernel and system modules with a *virt* module supporting the *libvirt* services). **Figure 18: KVM Environment** shows a high level overview with two VMs running in their own domains. The ***libvirt*** Support section shows how to configure these and their VM image files.

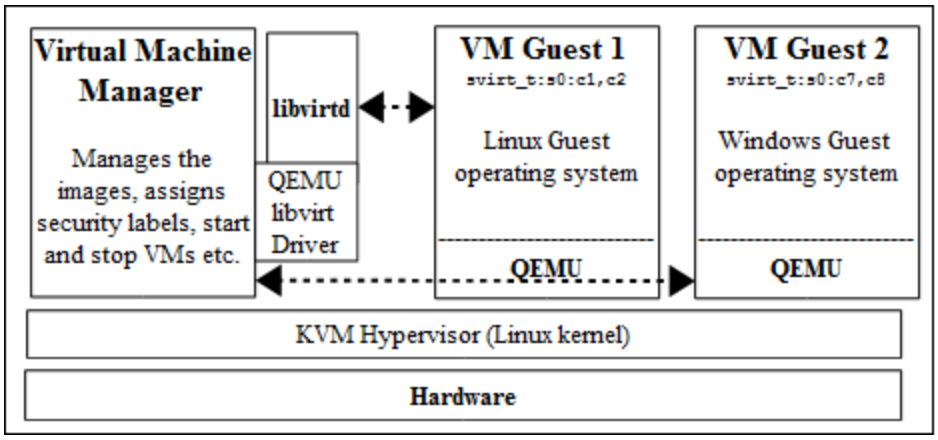


Figure 18: KVM Environment - *KVM provides the hypervisor while QEMU provides the hardware emulation services for the guest operating systems. Note that KVM requires CPU virtualisation support.*

libvirt Support

The Svirt project added security hooks into the *libvirt* library that is used by the *libvirtd* daemon. This daemon is used by a number of VM products (such as KVM, QEMU and Xen) to start their VMs running as guest operating systems.

The VM supplier can implement any security mechanism they require using a product specific libvirt **driver** that will load and manage the images. The SELinux implementation supports four methods of labeling VM images, processes and their resources with support from the Reference Policy *modules/services/virt* loadable module. To support this labeling, *libvirt* requires an MCS or MLS enabled policy as the ***level*** entry of the security context is used (*user:role:type:level*).

The link http://libvirt.org/drvqemu.html#securityselinux has details regarding the QEMU driver and the SELinux confinement modes it supports.

VM Image Labeling

This sections assumes VM images have been generated using the simple Linux kernel available at: http://wiki.qemu.org/Testing (the *linux-0.2.img.bz2* disk image), this image was renamed to reflect each test, for example '*Dynamic_VM1.img*'.

These images can be generated using the VMM by selecting the 'Create a new virtual machine' menu, 'importing existing disk image' then in step 2 Browse... selecting 'Choose Volume: *Dynamic_VM1.img*' with OS type: *Linux*, Version: *Generic 2.6.x kernel* and change step 4 'Name' to *Dynamic_VM1*.

Dynamic Labeling

The default mode is where each VM is run under its own dynamically configured domain and image file therefore isolating the VMs from each other (i.e. every time the VM is run a different and unique MCS label will be generated to confine each VM to its own domain). This mode is implemented as follows:

1. An initial context for the process is obtained from the */etc/selinux/<SELINUXTYPE>/contexts/virtual_domain_context* file (the default is *system_u:system_r:svirt_tcg_t:s0*).
2. An initial context for the image file label is obtained from the */etc/selinux/<SELINUXTYPE>/contexts/virtual_image_context* file. The default is *system_u:system_r:svirt_image_t:s0* that allows read/write of image files.
3. When the image is used to start the VM, a random MCS *level* is generated and added to the process context and the image file context. The process and image files are then transitioned to the context by the *libselinux*

API calls *setfilecon* and *setexeccon* respectively (see *security_selinux.c* in the *libvirt* source). The following example shows two running VM sessions each having different labels:

VM Image	Object	Dynamically assigned security context
Dynamic_VM1	*process*	*system_u:system_r:svirt_tcg_t:s0:c585,c813*
	file	*system_u:system_r:svirt_image_t:s0:c585,c813*
Dynamic_VM2	*process*	*system_u:system_r:svirt_tcg_t:s0:c535,c601*
	file	*system_u:system_r:svirt_image_t:s0:c535,c601*

The running image *ls -Z* and *ps -eZ* are as follows, and for completeness an *ls -Z* is shown when both VMs have been stopped:

```
# Both VMs running:

ls -Z /var/lib/libvirt/images
system_u:object_r:svirt_image_t:s0:c585,c813 Dynamic_VM1.img
system_u:object_r:svirt_image_t:s0:c535,c601 Dynamic_VM2.img

ps -eZ | grep qemu
system_u:system_r:svirt_tcg_t:s0:c585,c813 8707 ? 00:00:44 qemu-system-x86

system_u:system_r:svirt_tcg_t:s0:cc535,c601 8796 ? 00:00:37 qemu-system-x86
```

```
# Both VMs stopped (note that the categories are now missing AND
# the type has changed from svirt_image_t to virt_image_t):

ls -Z /var/lib/libvirt/images
system_u:object_r:virt_image_t:s0 Dynamic_VM1.img
system_u:object_r:virt_image_t:s0 Dynamic_VM2.img
```

Shared Image

If the disk image has been set to shared, then a dynamically allocated *level* will be generated for each VM process instance, however there will be a single instance of the disk image.

The Virtual Machine Manager can be used to set the image as shareable by checking the *Shareable* box as shown in **Figure 19**.

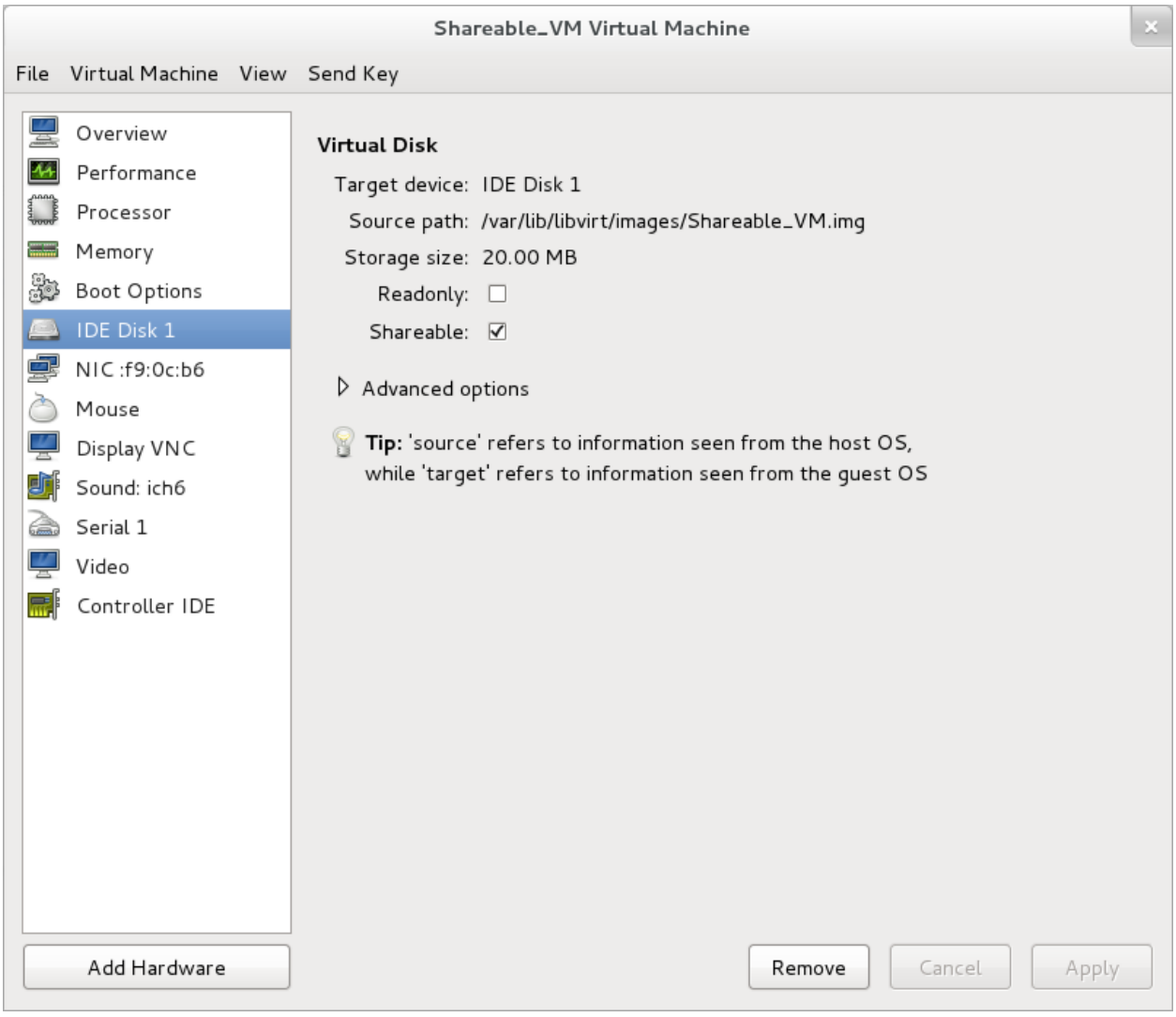


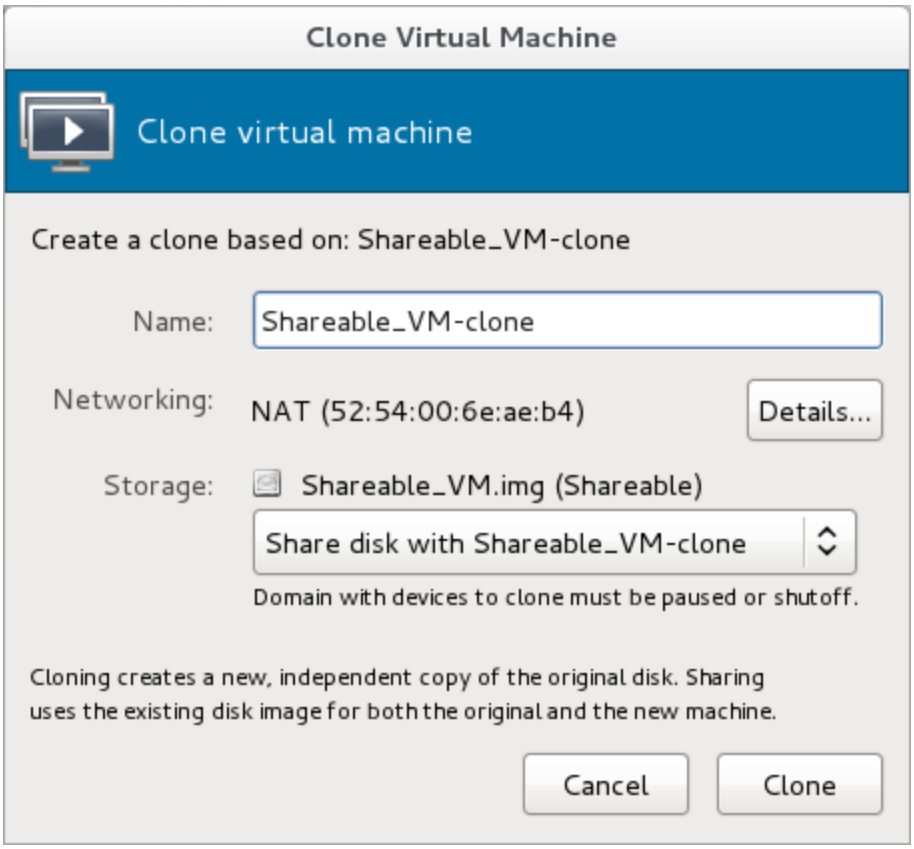
Figure 19: Setting the Virtual Disk as Shareable

This will set the image (*Shareable_VM.xml*) resource XML configuration file located in the */etc/libvirt/qemu* directory <*disk*> contents as follows:

```
# /etc/libvirt/qemu/Shareable_VM.xml:

<disk type='file' device='disk'>
    <driver name='qemu' type='raw'/>
    <source file='/var/lib/libvirt/images/Shareable_VM.img'/>
    <target dev='hda' bus='ide'/>
    <shareable/>
    <address type='drive' controller='0' bus='0' unit='0'/>
</disk>
```

As the two VMs will share the same image, the *Shareable_VM* service needs to be cloned and the VM resource name selected was *Shareable_VM-clone* as shown in the following screen shot:



The resource XML file <*disk*> contents generated are shown - note that it has the same *source file* name as the *Shareable_VM.xml* file shown above.

```
# /etc/libvirt/qemu/Shareable_VM-clone.xml:

<disk type='file' device='disk'>
    <driver name='qemu' type='raw'/>
    <source file='/var/lib/libvirt/images/Shareable_VM.img'/>
    <target dev='hda' bus='ide'/>
    <shareable/>
    <address type='drive' controller='0' bus='0' unit='0'/>
</disk>
```

With the targeted policy on Fedora the shareable option gave a error when the VMs were run as follows:

- **Could not allocate dynamic translator buffer**

The audit log contained the following AVC message:

```
type=AVC msg=audit(1326028680.405:367): avc: denied { execmem } for
pid=5404 comm="qemu-system-x86"
scontext=system_u:system_r:svirt_t:s0:c121,c746
tcontext=system_u:system_r:svirt_t:s0:c121,c746 tclass=process
```

To overcome this error, the following boolean needs to be enabled with ***setsebool****(8)* to allow access to shared memory (the *-P* option will set the boolean across reboots):

```
setsebool -P virt_use_execmem on
```

Now that the image has been configured as shareable, the following initialisation process will take place:

1. An initial context for the process is obtained from the */etc/selinux/<SELINUXTYPE>/contexts/virtual_domain_context* file (the default is *system_u:system_r:svirt_tcg_t:s0*).
2. An initial context for the image file label is obtained from the */etc/selinux/<SELINUXTYPE>/contexts/virtual_image_context* file. The default is *system_u:system_r:svirt_image_t:s0* that allows read/write of image files.
3. When the image is used to start the VM a random MCS level is generated and added to the process context (but not the image file). The process is then transitioned to the appropriate context by the *libselinux* API calls *setfilecon* and *setexeccon* respectively. The following example shows each VM having the same file label but different process labels:

VM Image	Object	Security context
Shareable_VM	*process*	*system_u:system_r:svirt_tcg_t:s0:c231,c245*
Shareable_VM-clone	*process*	*system_u:system_r:svirt_tcg_t:s0:c695,c894*
	file	*system_u:system_r:svirt_image_t:s0*

The running image *ls -Z* and *ps -eZ* are as follows and for completeness an *ls -Z* is shown when both VMs have been stopped:

```
# Both VMs running and sharing same image:

ls -Z /var/lib/libvirt/images
system_u:object_r:svirt_image_t:s0 Shareable_VM.img

# but with separate processes:
ps -eZ | grep qemu
system_u:system_r:svirt_t:s0:c231,c254 6748 ? 00:01:17 qemu-system-x86
system_u:system_r:svirt_t:s0:c695,c894 7664 ? 00:00:03 qemu-system-x86
```

```
# Both VMs stopped (note that the type has remained as svirt_image_t)

ls -Z /var/lib/libvirt/images
system_u:object_r:svirt_image_t:s0 Shareable_VM.img
```

Static Labeling

It is possible to set static labels on each image file, however a consequence of this is that the image cannot be cloned using the VMM, therefore an image for each VM will be required. This is the method used to configure VMs on MLS systems as there is a known label that would define the security level. With this method it is also possible to configure two or more VMs with the same security context so that they can share resources. A useful reference is at: http://libvirt.org/formatdomain.html#seclabel.

If using the Virtual Machine Manager GUI, then by default it will start each VM running as they are built, therefore they need to be stopped and restarted once configured for static labels, the image file will also need to be

relabeled. An example VM configuration follows where the VM has been created as *Static_VM1* using the Fedora *targeted* policy in enforcing mode (just so all errors are flagged during the build):

1. To set the required security context requires editing the *Static_VM1* configuration file using ***virsh**(1)* as follows:

```
virsh edit Static_VM1
```

Then add the following at the end of the file:

```
....

</devices>

<!-- The <seclabel> tag needs to be placed btween the existing
    </devices> and </domain> tags -->

    <seclabel type='static' model='selinux' relabel='no'>
        <label>system_u:system_r:svirt_t:s0:c1022,c1023</label>
    </seclabel>
</domain>
```

For this example *svirt_t* has been chosen as it is a valid context (however it will not run as explained in the text). This context will be written to the *Static_VM1.xml* configuration file in */etc/libvirt/qemu*.

1. If the VM is now started an error will be shown as follows:

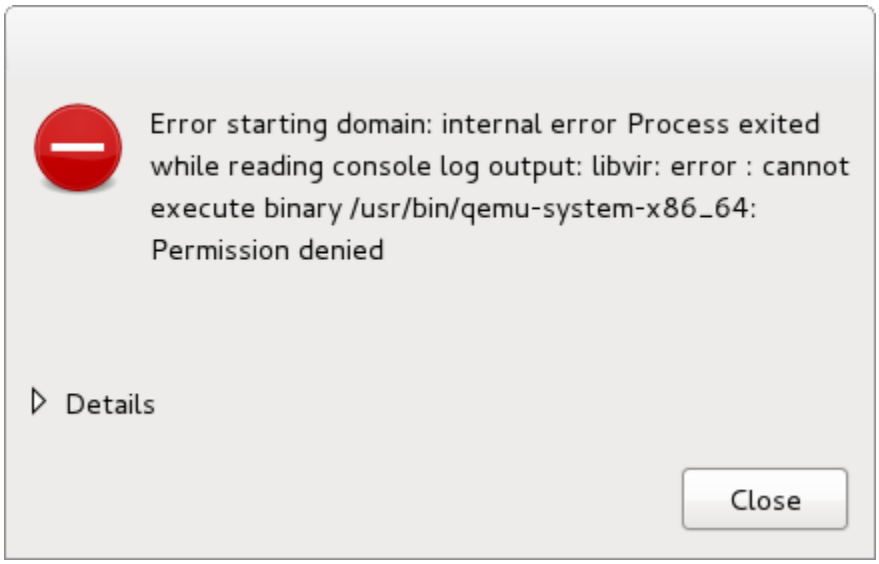


Figure 21: Image Start Error

This is because the image file label is incorrect as by default it is labeled *virt_image_t* when the VM image is built (and *svirt_t* does not have read/write permission for this label):

```
# The default label of the image at build time:

system_u:object_r:virt_image_t:s0 Static_VM1.img
```

There are a number of ways to fix this, such as adding an allow rule or changing the image file label. In this example the image file label will be changed using ***chcon****(1)* as follows:

```
# This command is executed from /var/lib/libvirt/images
#
# This sets the correct type:

chcon -t svirt_image_t Static_VM1.img
```

Optionally, the image can also be relabeled so that the *[level]* is the same as the process using *chcon* as follows:

```
# This command is executed from /var/lib/libvirt/images
#
# Set the MCS label to match the process (optional step):

chcon -l s0:c1022,c1023 Static_VM1.img
```

1. Now that the image has been relabeled, the VM can now be started.

The following example shows two static VMs (one is configured for *unconfined_t* that is allowed to run under the targeted policy - this was possible because the '*setsebool -P virt_transition_userdomain on*' boolean was set that allows *virtd_t* domain to transition to a user domain (e.g. *unconfined_t*).

VM Image	Object	Static security context
Static_VM1	*process*	*system_u:system_r:svirt_t:s0:c1022,c1023*
	file	*system_u:system_r:svirt_image_t:s0:c1022,c1023*
Static_VM2	*process*	*system_u:system_r:unconfined_t:s0:c11,c22*
	file	*system_u:system_r:virt_image_t:s0*

The running image *ls -Z* and *ps -eZ* are as follows, and for completeness an *ls -Z* is shown when both VMs have been stopped:

```
# Both VMs running (Note that Static_VM2 did not have file level reset):

ls -Z /var/lib/libvirt/images
system_u:object_r:svirt_image_t:s0:c1022,c1023 Static_VM1.img
system_u:object_r:virt_image_t:s0 Static_VM2.img

ps -eZ | grep qemu
system_u:system_r:svirt_t:s0:c585,c813 6707 ? 00:00:45 qemu-system-x86
system_u:system_r:unconfined_t:s0:c11,c22 6796 ? 00:00:26 qemu-system-x86
```

```
# Both VMs stopped (note that Static_VM1.img was relabeled svirt_image_t
# to enable it to run, however Static_VM2.img is still labeled
# virt_image_t and runs okay. This is because the process is run as
# unconfined_t that is allowed to use virt_image_t):
```

```
system_u:object_r:svirt_image_t:s0:c1022,c1023 Static_VM1.img
system_u:object_r:virt_image_t:s0 Static_VM2.img
```

Xen Support

This is not supported by SELinux in the usual way as it is built into the actual Xen software as a 'Flask/TE' extension for the XSM (Xen Security Module). Also the Xen implementation has its own built-in policy (*xen.te*) and supporting definitions for access vectors, security classes and initial SIDs for the policy. These Flask/TE components run in Domain 0 as part of the domain management and control supporting the Virtual Machine Monitor (VMM) as shown in **Figure 22: Xen Hypervisor**.

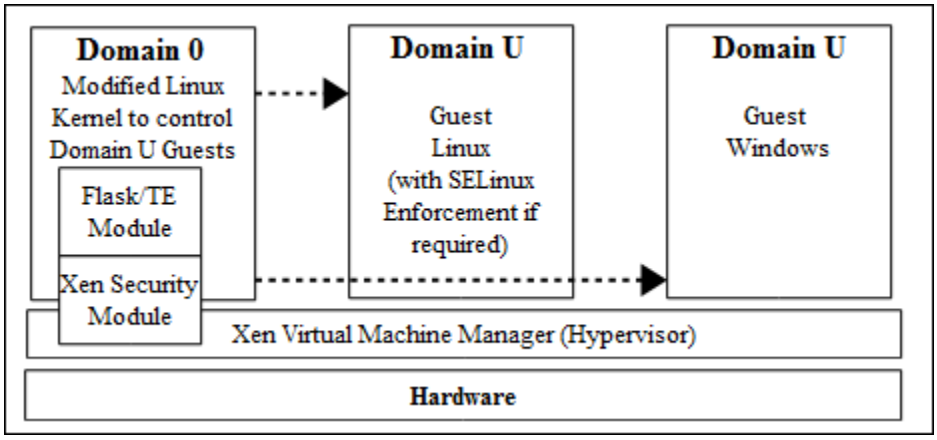


Figure 22: Xen Hypervisor - *Using XSM and Flask/TE to enforce policy on the physical I/O resources*

The "**How Does Xen Work**" document describes the basic operation of Xen, the "**Xen Security Modules**" describes the XSM/Flask implementation, and the *xsm-flask.txt* file in the Xen source package describes how SELinux and its supporting policy is implemented.

However (just to confuse the issue), there is another Xen policy module (also called *xen.te*) in the Reference Policy to support the management of images etc. via the Xen console.

For reference, the Xen policy supports additional policy language statements that defined in the **Xen Statements** section.

X-Windows SELinux Support

The SELinux X-Windows (XSELinux) implementation provides fine grained access control over the majority of the X-server objects (known as resources) using an X-Windows extension acting as the object manager (OM). The extension name is **SELinux**.

The X-Windows object classes and permissions are listed in the **X Windows Object Classes** section and the Reference Policy modules have been updated to enforce policy using the XSELinux object manager.

On Fedora XSELinux is disabled in the targeted policy via the *xserver_object_manager* boolean.

Infrastructure Overview

It is important to note that the X-Windows OM operates on the low level window objects of the X-server. A windows manager (such as Gnome or twm) would then sit above this, however they (the windows manager or even the lower level Xlib) would not be aware of the policy being enforced by SELinux. Therefore there can be situations where X-Windows applications get bitter & twisted at the denial of a service. This can result in either opening the policy more than desired, or just letting the application keep aborting, or modifying the application.

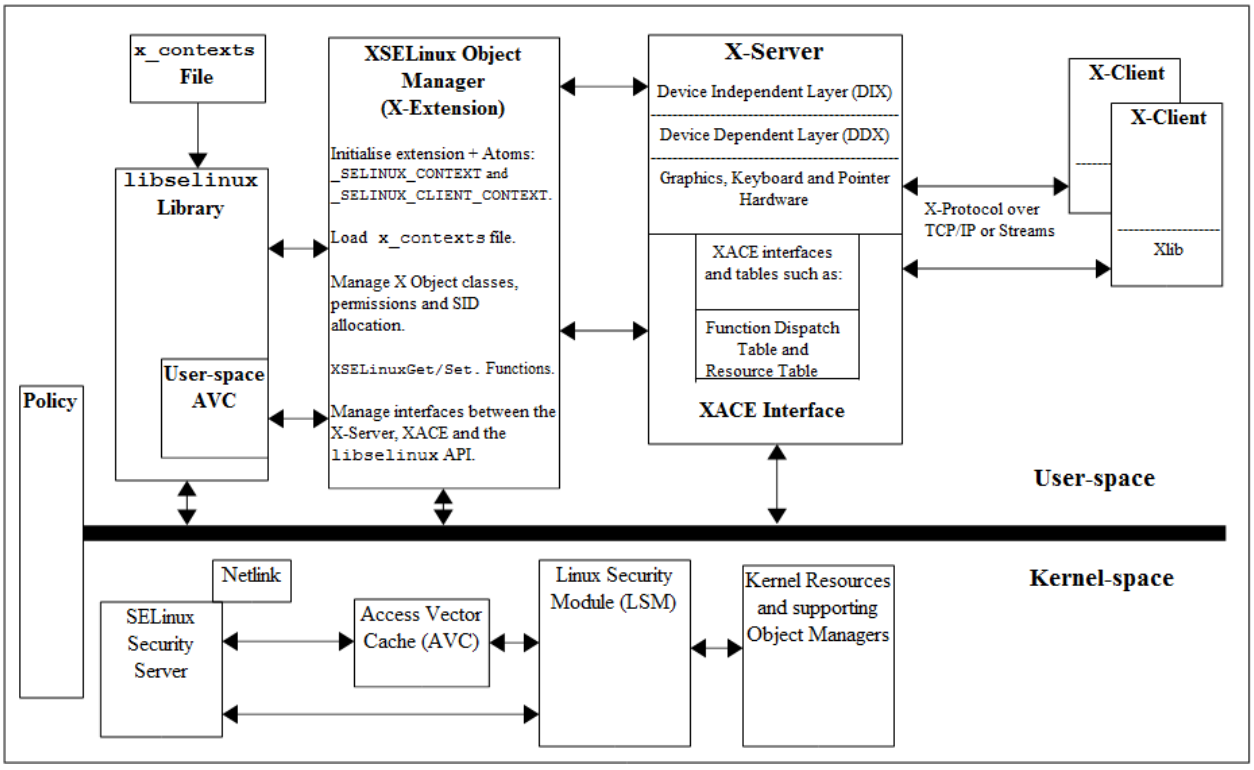


Figure 23: X-Server and XSELinux Object Manager - *Showing the supporting services. The kernel space services are discussed in the **Linux Security Module and SELinux** section.*

Using **Figure 23: X-Server and XSELinux Object Manager**, the major components that form the overall XSELinux OM are (top left to right):

The Policy - The Reference Policy has been updated, however in Fedora the OM is enabled for mls and disabled for targeted policies via the *xserver-object-manager* boolean. Enabling this boolean also initialises the XSELinux OM extension. Important note - The boolean must be present in any policy and be set to *true*, otherwise the object manager will be disabled as the code specifically checks for the boolean.

libselinux - This library provides the necessary interfaces between the OM, the SELinux userspace services (e.g. reading configuration information and providing the AVC), and kernel services (e.g. security server for access decisions and policy update notification).

x_contexts File - This contains default context configuration information that is required by the OM for labeling certain objects. The OM reads its contents using the ***selabel_lookup****(3)* function.

XSELinux Object Manager - This is an X-extension for the X-server process that mediates all access decisions between the X-server (via the XACE interface) and the SELinux security server (via *libselinux*). The OM is initialised before any X-clients connect to the X-server.

The OM has also added XSELinux functions that are described in to allow contexts to be retrieved and set by userspace SELinux-aware applications.

XACE Interface - This is an 'X Access Control Extension' (XACE) that can be used by other access control security extensions, not only SELinux. Note that if other security extensions are linked at the same time, then the X-function will only succeed if allowed by all the security extensions in the chain.

This interface is defined in the "**X Access Control Extension Specification**". The specification also defines the hooks available to OMs and how they should be used. The provision of polyinstantiation services for properties and selections is also discussed. The XACE interface is a similar service to the LSM that supports the kernel OMs.

X-server - This is the core X-Windows server process that handles all request and responses to/from X-clients using the X-protocol. The XSELinux OM is intercepting these request/responses via XACE and enforcing policy decisions.

X-clients - These connect to the X-server are are typically windows managers such as Gnome, twm or KDE.

Kernel-Space Services - These are discussed in the **Linux Security Module and SELinux** section.

Polyinstantiation

The OM / XACE services support polyinstantiation of properties and selections allowing these to be grouped into different membership areas so that one group does not know of the exsistance of the others. To implement polyinstantiation the *poly_* keyword is used in the ***x_contexts*** for the required selections and properties, there would then be a corresponding ***type_member*** in the policy to enforce the separation by computing a new context with either ***security_compute_member****(3)* or ***avc_compute_member****(3)*.

Note that the current Reference Policy does not implement polyinstantiation, instead the MLS policy uses ***mlsconstrain*** to limit the scope of properties and selections.

Configuration Information

This section covers:

- How to enable/disable the OM X-extension.
- How to determine the OM X-extension opcode.
- How to configure the OM in a specific SELinux enforcement mode.
- The *x-contexts* configuration file.

Enable/Disable the OM from Policy Decisions

The Reference Policy has an *xserver_object_manager* boolean that enables/disables the X-server policy module and also stops the object manager extension from initialising when X-Windows is started. The following command will enable the boolean, however it will be necessary to reload X-Windows to initialise the extension (i.e. run the **init 3** and then **init 5** commands):

```
setsebool -P xserver_object_manager true
```

If the boolean is set to *false*, the x-server log will indicate that "SELinux: Disabled by boolean". Important note - If the boolean is not present in a policy then the object manager will always be enabled (therefore if not required then either do not include the object manager in the X-server build or add the boolean to the policy and set it to false or add a disabled entry to the **xorg.conf** file as described in the next section.

Configure OM Enforcement Mode

If the X-server object manager needs to be run in a specific SELinux enforcement mode, then the option may be added to the *xorg.conf* file (normally in */etc/X11/xorg.conf.d*). The option entries are as follows:

- SELinux mode disabled
- SELinux mode permissive
- SELinux mode enforcing

Note that the entry must be exact otherwise it will be ignored. An example entry is:

```
Section "Module"
    SubSection "extmod"
        Option "SELinux mode enforcing"
    EndSubSection
EndSection
```

If there is no entry, the object manager will follow the current SELinux enforcement mode.

Determine OM X-extension Opcode

The object manager is treated as an X-server extension and its major opcode can be queried using Xlib *XQueryExtension* function as follows:

```
/* Get the SELinux Extension opcode */

if (!XQueryExtension (dpy, "SELinux", &opcode, &event, &error)) {
    perror ("XSELinux extension not available");
    exit (1);
    }
else
    printf("XQueryExtension for XSELinux Extension - Opcode: %d Events: %d Error:
%d\n",
           opcode, event, error);

/* Have XSELinux Object Manager */
```

The *x_contexts* File

The *x_contexts* file contains default context information that is required by the OM to initialise the service and then label objects as they are created. The policy will also need to be aware of the context information being used as it will use this to enforce policy or transition new objects. A typical entry is as follows:

```
# object_type  object_name  context
selection      PRIMARY      system_u:object_r:clipboard_xselection_t:s0
```

or for polyinstantiation support:

```
# object_type    object_name  context
poly_selection   PRIMARY      system_u:object_r:clipboard_xselection_t:s0
```

The *object_name* can contain '*' for 'any' or '?' for 'substitute'.

The OM uses the ***selabel**(3)* functions (such as ***selabel_lookup**(3)*) that are a part of *libselinux* to fetch the relevant information from the *x_contexts* file.

The valid *object_type* entries are *client*, *property*, *poly_property*, *extension*, *selection*, *poly_selection* and *events*.

The *object_name* entries can be any valid X-server resource name that is defined in the X-server source code and can typically be found in the *protocol.txt* and *BuiltInAtoms* source files (in the *dix* directory of the **xorg-server** source package), or user generated via the Xlib libraries (e.g. *XInternAtom*).

Notes:

1. The way the XSELinux extension code works (see *xselinux_label.c* - SELinuxAtomToSIDLookup()) is that non-poly entries are searched for first, if an entry is not found then it searches for a matching poly entry. The reason for this behavior is that when operating in a secure environment all objects would be polyinstantiated unless there are specific exemptions made for individual objects to make them non-polyinstantiated. There would then be a 'poly_selection' or 'poly_property' at the end of the section.
2. For systems using the Reference Policy all X-clients connecting remotely will be allocated a security context from the *x_contexts* file of:

```
# object_type object_name context
client * system_u:object_r:remote_t:s0
```

A full description of the *x_contexts* file format is given in the ***x_contexts*** section.

SELinux Extension Functions

The XSELinux Extension Functions listed below are supported by the object manager as X-protocol extensions.

Note that **XSELinuxGet*** functions return a default context, however those with Minor Parameter: 2, 6, 9, 11, 16 and 18 will not return a value unless one has been set by the appropriate **XSELinuxSet*** function (Minor Parameter: 1, 5, 8, 10, 15 and 17).

Function Name	Minor Parameter	Opcode
XSELinuxQueryVersion	0	None

Returns the XSELinux version. Fedora returns 1.1.

Function Name	Minor Parameter	Opcode
XSELinuxSetDeviceCreateContext	1	Context + Len

Sets the context for creating a device object (*x_device*).

Function Name	Minor Parameter	Opcode
XSELinuxGetDeviceCreateContext	2	None

Retrieves the context set by *XSELinuxSetDeviceCreateContext*.

Function Name	Minor Parameter	Opcode
XSELinuxSetDeviceContext	3	DeviceID + Context + Len

Sets the context for creating the specified DeviceID object.

Function Name	Minor Parameter	Opcode
XSELinuxGetDeviceContext	4	DeviceID

Retrieves the context set by *XSELinuxSetDeviceContext*.

Function Name	Minor Parameter	Opcode
XSELinuxSetWindowCreateContext	5	Context + Len

Set the context for creating a window object (*x_window*).

Function Name	Minor Parameter	Opcode
XSELinuxGetWindowCreateContext	6	None

Retrieves the context set by *XSELinuxSetWindowCreateContext*.

Function Name	Minor Parameter	Opcode
XSELinuxGetWindowContext	7	WindowID

Retrieves the specified WindowID context.

Function Name	Minor Parameter	Opcode
XSELinuxSetPropertyCreateContext	8	Context

Sets the context for creating a property object (*x_property*).

Function Name	Minor Parameter	Opcode
XSELinuxGetPropertyCreateContext	9	None

Retrieves the context set by *XSELinuxSetPropertyCreateContext*.

Function Name	Minor Parameter	Opcode
XSELinuxSetPropertyUseContext	10	Context + Len

Sets the context of the property object to be retrieved when polyinstantiation is being used.

Function Name	Minor Parameter	Opcode
XSELinuxGetPropertyUseContext	11	None

Retrieves the property object context set by *SELinuxSetPropertyUseContext*.

Function Name	Minor Parameter	Opcode
XSELinuxGetPropertyContext	12	WindowID + AtomID

Retrieves the context of the property atom object.

Function Name	Minor Parameter	Opcode
XSELinuxGetPropertyDataContext	13	WindowID + AtomID

Retrieves the context of the property atom data.

Function Name	Minor Parameter	Opcode
XSELinuxListProperties	14	WindowID

Lists the object and data contexts of properties associated with the selected WindowID.

Function Name	Minor Parameter	Opcode
XSELinuxSetSelectionCreateContext	15	Context + Len

Sets the context to be used for creating a selection object.

Function Name	Minor Parameter	Opcode
XSELinuxGetSelectionCreateContext	16	None

Retrieves the context set by *SELinuxSetSelectionCreateContext*.

Function Name	Minor Parameter	Opcode
XSELinuxSetSelectionUseContext	17	Context + Len

Sets the context of the selection object to be retrieved when polyinstantiation is being used. See the *XSELinuxListSelections* function for an example.

Function Name	Minor Parameter	Opcode
XSELinuxGetSelectionUseContext	18	None

Retrieves the selection object context set by *SELinuxSetSelectionUseContext*.

Function Name	Minor Parameter	Opcode
XSELinuxGetSelectionContext	19	AtomID

Retrieves the context of the specified selection atom object.

Function Name	Minor Parameter	Opcode
XSELinuxGetSelectionDataContext	20	AtomID

Retrieves the context of the selection data from the current selection owner (*x_application_data* object).

Function Name	Minor Parameter	Opcode
XSELinuxListSelections	21	None

Lists the selection atom object and data contexts associated with this display. The main difference in the listings is that when (for example) the *PRIMARY* selection atom is polyinstantiated, multiple entries can returned. One has the context of the atom itself, and one entry for each process (or x-client) that has an active polyinstantiated entry, for example:

Atom: PRIMARY - label defined in the *x_contexts* file (this is also for non-poly listing):

- Object Context: *system_u:object_r:primary_xselection_t*
- Data Context: *system_u:object_r:primary_xselection_t*

Atom: PRIMARY - Labels for client 1:

- Object Context: *system_u:object_r:x_select_paste1_t*
- Data Context: *system_u:object_r:x_select_paste1_t*

Atom: PRIMARY - Labels for client 2:

- Object Context: *system_u:object_r:x_select_paste2_t*
- Data Context: *system_u:object_r:x_select_paste2_t*

Function Name	Minor Parameter	Opcode
XSELinuxGetClientContext	22	ResourceID

Retrieves the client context of the specified ResourceID.

PostgreSQL SELinux Support

This section gives an overview of PostgreSQL version 11.x with the *sepgsql* extension to support SELinux. It assumes some basic knowledge of PostgreSQL that can be found at: http://wiki.postgresql.org/wiki/Main_Page

It is important to note that PostgreSQL from version 11.x has the necessary infrastructure to support labeling of database objects via external 'providers'. The *sepgsql* extension has therefore been added that provides SELinux labeling. The extension is not installed by default but as an option as outlined in the sections that follow. Because of these changes the original version 9.0 patches are no longer supported (i.e. the SE-PostgreSQL database engine is replaced by the PostgreSQL database engine plus the *sepgsql* extension). A consequence of this change is that row level labeling is no longer supported by SELinux, however a Row Level Security (RLS) option has now been added within core PostgreSQL that does not utilise labeling.

The features of sepgsql 11.x and its setup are covered in the following document:

https://www.postgresql.org/docs/11/sepgsql.html

sepgsql Overview

The *sepgsql* extension adds SELinux mandatory access controls (MAC) to database objects such as tables, columns, views, functions, schemas and sequences. **Table 1: Database Security Context Information** shows a simple database with one table and two columns, each with their object class and associated security context (the **Internal Tables** section shows these entries from the *testdb* database in the **Notebook sepgsql Example**. The database object classes and permissions are described in **Appendix A - Object Classes and Permissions**.

database (*db_database*) - context = 'unconfined_u:object_r:postgresql_db_t:s0' This context is inherited from the database directory label - ls -Z /var/lib/pgsql/data	
schema (*db_schema*) - security_label = 'unconfined_u:object_r:sepgsql_schema_t:s0:c10'	
table (*db_table*) - security_label = 'unconfined_u:object_r:sepgsql_table_t:s0:c20'	
column 1 (*db_column*) - security_label = 'unconfined_u:object_r:sepgsql_table_t:s0:c30'	**column 2** - (*db_column*) security_label = 'unconfined_u:object_r:sepgsql_table_t:s0:c40'

Table 1: Database Security Context Information - *Showing the security contexts that can be associated to a schema, table and columns.*

To use SE-PostgreSQL each Linux user must have a valid PostgreSQL database role (not to be confused with an SELinux role). The default installation automatically adds a user called *pgsql* with a suitable database role.

If a client is connecting remotely and labeled networking is required, then it is possible to use IPSec or NetLabel as discussed in the **SELinux Networking Support** section (the "**Security-Enhanced PostgreSQL Security Wiki**" also covers these methods of connectivity with examples).

Using **Figure 24a: SE-PostgreSQL Services**, the database client application (that could be provided by an API for Perl/PHP or some other programming language) connects to a database and executes SQL commands. As the SQL

commands are processed by PostgreSQL, each operation performed on an object is checked by the object manager (OM) to see if the operation is allowed by the security policy or not.

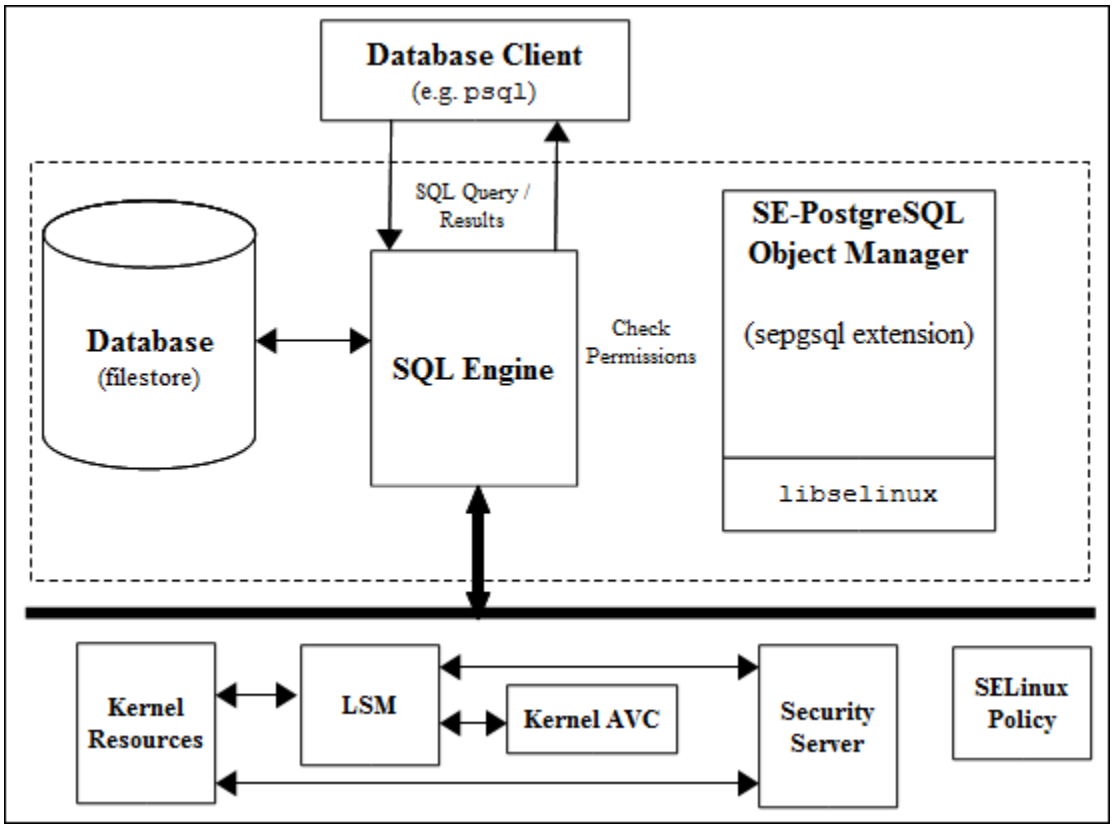


Figure 24a: SE-PostgreSQL Services - *The Object Manager checks access permissions for all objects under its control.*

SE-PostgreSQL supports SELinux services via the *libselinux* library with AVC audits being logged via the standard PostgreSQL logfile as described in the **Logging Security Events** section.

Installing SE-PostgreSQL

The **https://www.postgresql.org/docs/11/sepgsql.html** page contains all the information required to install the *sepgsql* extension.

There are also instructions in the **Notebook sepgsql Example - README** that describes building the example database used in the sections below.

SECURITY LABEL SQL Command

The '*SECURITY LABEL*' SQL command has been added to PostgreSQL to allow security providers to label or change a label on database objects. The full syntax is defined in the ***security_label****(7)* man page, with some examples:

```
-- These set the security label on objects (default provider being SELinux):

SECURITY LABEL ON SCHEMA test_ns IS
'unconfined_u:object_r:sepgsql_schema_t:s0:c10';

SECURITY LABEL ON TABLE test_ns.info IS
'unconfined_u:object_r:sepgsql_table_t:s0:c20';
```

```
SECURITY LABEL ON COLUMN test_ns.info.user_name IS
'unconfined_u:object_r:sepgsql_table_t:s0:c30';

SECURITY LABEL ON COLUMN test_ns.info.email_addr IS
'unconfined_u:object_r:sepgsql_table_t:s0:c40';
```

Additional SQL Functions

The following functions have been added:

sepgsql_getcon()

Returns the client security context.

sepgsql_mcstrans_in(text con)

Translates the readable *range* of the context into raw format provided the ***mcstransd****(8)* daemon is running.

sepgsql_mcstrans_out(text con)

Translates the raw *range* of the context into readable format provided the ***mcstransd****(8)* daemon is running.

sepgsql_restorecon(text specfile)

Sets security contexts on all database objects (must be superuser) according to the *specfile*. This is normally used for initialisation of the database by the *sepgsql.sql* script. If the parameter is NULL, then the default *sepgsql_contexts* file is used. See ***selabel_db****(5)* details.

postgresql.conf Entries

The *postgresql.conf* file supports the following additional entries to enable and manage SE-PostgreSQL:

- This entry is mandatory to enable the *sepgsql* extension to be loaded:

```
shared_preload_libraries = 'sepgsql'
```

- These entries are optional and default to '*off*'.

```
# This enables sepgsql to always run in permissive mode:
sepgsql.permissive = on

# This enables printing of audit messages regardless of the policy setting:
sepgsql.debug_audit = on
```

To view these settings the *SHOW* SQL statement can be used (*psql* output shown):

```
SHOW sepgsql.permissive;
sepgsql.permissive
---------------
on
(1 row)
```

```
SHOW sepgsql.debug_audit;
sepgsql.debug_audit
---------------
on
(1 row)
```

Logging Security Events

SE-PostgreSQL manages its own AVC audit entries in the standard PostgreSQL log normally located within the */var/lib/pgsql/data/pg_log* directory and by default only errors are logged (Note that there are no SE-PostgreSQL AVC entries added to the standard *audit.log*). The '*sepgsql.debug_audit* = *on*' can be set to log all audit events.

Internal Tables

To support the overall database operation PostgreSQL has internal tables in the system catalog that hold information relating to databases, tables etc. This section will only highlight the *pg_seclabel* table that holds the security label and other references. The *pg_seclabel* is shown in the table below and has been taken from http://www.postgresql.org/docs/11/static/catalog-pg-seclabel.html.

Name	Type	References	Comments
objoid	oid	any OID column	The OID of the object this security label pertains to.
classoid	oid	pg_class.oid	The OID of the system catalog this object appears in.
objsubid	int4		For a security label on a table column, this is the column number (the *objoid* and *classoid* refer to the table itself). For all other objects this column is zero.
provider	text		The label provider associated with this label. Currently only SELinux is supported.
label	text		The security label applied to this object.

These are entries taken from a '*SELECT* FROM pg_seclabel;' *command that refers to the example* testdb* database built using the **Notebook - testdb-example.sql**:

```
objoid  | classoid | objsubid | provider |          label
--------+----------+----------+----------
+-----------------------------------------------
16390   | 2615     |    0     | selinux  |
unconfined_u:object_r:sepgsql_schema_t:s0:c10
16391   | 1259     |    0     | selinux  |
unconfined_u:object_r:sepgsql_table_t:s0:c20
16391   | 1259     |    1     | selinux  |
unconfined_u:object_r:sepgsql_table_t:s0:c30
```

```
16391   | 1259     |    2     | selinux  |
unconfined_u:object_r:sepgsql_table_t:s0:c40
```

The first entry is the schema, the second entry is the table itself, and the third and fourth entries are columns 1 and 2.

There is also a built-in 'view' to show additional detail regarding security labels called '*pg_seclabels*'. Using '*SELECT* FROM pg_seclabels;*' command, the entries shown above become:

```
objoid|classoid|objsubid|objtype|objnamespace|  objname     | provider| label
------+--------+--------+-------+------------+--------------+---------
+------------------------
16390 | 2615   |    0   | schema| 16390      | test_ns      | selinux |
unconfined_u:object_r:sepgsql_schema_t:s0:c10
16391 | 1259   |    0   | table | 16390      | test_ns.info | selinux |
unconfined_u:object_r:sepgsql_table_t:s0:c20
16391 | 1259   |    1   | column| 16390      | test_ns.info.| selinux |
unconfined_u:object_r:sepgsql_table_t:s0:c30
      |        |        |       |            | user_name    |         |
16391 | 1259   |    2   | column| 16390      | test_ns.info.| selinux |
unconfined_u:object_r:sepgsql_table_t:s0:c40
      |        |        |       |            | email_addr   |         |
```

Apache SELinux Support

Apache web servers are supported by SELinux using the Apache policy modules from the Reference Policy (*httpd* modules), however there is no specific Apache object manager. There is though an SELinux-aware shared library and policy that will allow finer grained access control when using Apache with threads. The additional Apache module is called *mod_selinux.so* and has a supporting policy module called *mod_selinux.pp*.

```
dnf install mod_selinux
```

The *mod_selinux* policy module makes use of the *typebounds* statement that was introduced into version 24 of the policy (requires a minimum kernel of 2.6.28). *mod_selinux* allows threads in a multi-threaded application (such as Apache) to be bound within a defined set of permissions in that the child domain cannot have greater permissions than the parent domain.

These components are known as 'Apache / SELinux Plus' and are described in the sections that follow, however a full description including configuration details is available from:

https://code.google.com/archive/p/sepgsql/wikis/Apache_SELinux_plus.wiki

The objective of these Apache add-on services is to achieve a fully SELinux-aware web stack (although not there yet). For example, currently the LAPP[11] (Linux, Apache, PostgreSQL, PHP / Perl / Python) stack has the following support:

L - Linux has SELinux support.

A - Apache has partial SELinux support using the 'Apache SELinux Plus' module.

P - PostgreSQL has SELinux support using the PostgreSQL *sepgsql* extension.

P - PHP / Perl / Python are not currently SELinux-aware, however PHP and Python do have support for libselinux functions in packages: PHP - with the *php-pecl-selinux* package, Python - with the *libselinux-python* package.

The "A secure web application platform powered by SELinux" document gives a good overview of the LAPP architecture.

mod_selinux Overview

What the *mod_selinux* module achieves is to allow a web application (or a 'request handler') to be launched by Apache with a security context based on policy rather than that of the web server process itself, for example:

1. A user sends an HTTP request to Apache that requires the services of a web application (Apache may or may not apply HTTP authentication).
2. Apache receives the request and launches the web application instance to perform the task:
    - Without *mod_selinux* enabled the web applications security context is identical to the Apache web server process, it is therefore not possible to restrict it privileges.
    - With *mod_selinux* enabled, the web application is launched with the security context defined in the *mod_selinux.conf* file (*selinuxDomainVal <security_context>* entry). It is also possible to restrict its privileges as described in the Bounds Overview section.
3. The web application exits, handing control back to the web server that replies with the HTTP response.

Bounds Overview

Because multiple threads share the same memory segment, SELinux was unable to check the information flows between these different threads when using ***setcon****(3)* in pre 2.6.28 kernels. This meant that if a thread (the parent) should launch another thread (a child) with a different security context, SELinux could not enforce the different permissions.

To resolve this issue the *typebounds* statement was introduced with kernel support in 2.6.28 that stops a child thread (the 'bounded domain') having greater privileges than the parent thread (the 'bounding domain') i.e. the child thread must have equal or less permissions than the parent.

For example the following *typebounds* statement and *allow* rules:

```
#          parent  | child
#          domain  | domain
typebounds httpd_t   httpd_child_t;

allow httpd_t etc_t:file { getattr read };
allow httpd_child_t etc_t:file { read write };
```

State that the parent domain (*httpd_t*) has *file : { getattr read }* permissions. However the child domain (*httpd_child_t*) has been given *file : { read write }*. At run-time, this would not be allowed by the kernel because the parent does not have *write* permission, thus ensuring the child domain will always have equal or less privileges than the parent.

When ***setcon****(3)* is used to set a different context on a new thread without an associated *typebounds* policy statement, then the call will return 'Operation not permitted' and an *SELINUX_ERR* entry will be added to the audit log stating *op=security_bounded_transition result=denied* with the old and new context strings.

Should there be a valid *typebounds* policy statement and the child domain exercises a privilege greater that that of the parent domain, the operation will be denied and an *SELINUX_ERR* entry will be added to the audit log stating *op=security_compute_av reason=bounds* with the context strings and the denied class and permissions.

SELinux Configuration Files

This section explains each SELinux configuration file with its format, example content and where applicable, any supporting SELinux commands or **libselinux** library API functions.

Where configuration files have specific man pages, these are noted by adding the man page section (e.g. ***semanage.config**(5)*).

This Notebook classifies the types of configuration file used in SELinux as follows:

1. **Global Configuration files** that affect the active policy and their supporting SELinux-aware applications, utilities or commands. This Notebook will only refer to the commonly used configuration files.
2. **Policy Store Configuration Files** that are managed by the **semanage**(8) and **semodule**(8) commands. These are used to build the majority of the **Policy Configuration Files** and should NOT be edited as together they describe the overall 'policy' configuration.
3. **Policy Configuration Files** used by an active (run time) policy/system. Note that there can be multiple policy configurations on a system (e.g. */etc/selinux/targeted* and */etc/selinux/mls*), however only one can be the active policy.
4. The **SELinux Filesystem** located under the */sys/fs/selinux* directory and reflects the current configuration of SELinux for the active policy. This area is used extensively by the libselinux library for userspace object managers and other SELinux-aware applications. These files and directories should not be updated by users (the majority are read only anyway), however they can be read to check various configuration parameters and viewing the currently loaded policy using tools such as ***apol**(1)* (e.g. *apol /sys/fs/selinux/policy*).

The Policy Store

Version 2.7 of *libsemanage*, *libsepol*, and *policycoreutils* had the policy module store has moved from */etc/selinux/<SELINUXTYPE>/modules* to */var/lib/selinux/<SELINUXTYPE>*.

This new infrastructure now makes it possible to build policies containing a mixture of Reference Policy modules, kernel policy language modules and modules written in the CIL language as shown in the following examples:

```
# Compile and install a base and two modules written in kernel language:

checkmodule -o base.mod base.conf
semodule_package -o base.pp -m base.mod -f base.fc
checkmodule -m ext_gateway.conf -o ext_gateway.mod
semodule_package -o ext_gateway.pp -m ext_gateway.mod -f gateway.fc
checkmodule -m int_gateway.conf -o int_gateway.mod
semodule_package -o int_gateway.pp -m int_gateway.mod
semodule -s modular-test --priority 100 -i base.pp ext_gateway.pp int_gateway.pp
```

```
# Compile and install an updated module written in CIL:

semodule -s modular-test --priority 400 -i custom/int_gateway.cil
```

```
# Show a full listing of modules:

semodule -s modular-test --list-modules=full
400 int_gateway cil
100 base pp
100 ext_gateway pp
100 int_gateway pp
```

```
# Show a standard listing of modules:

semodule -s modular-test --list-modules=standard
base
ext_gateway
int_gateway
```

The ***semodule****(8)* command now has a number of new options, with the most significant being:

1. Setting module priorities (*-X | --priority*), this is discussed in The priority Option section.
2. Listing modules (*--list-modules=full | standard*). The '*full*' option shows all the available modules with their priority and policy format. The '*standard*' option will only show the highest priority, enabled modules.

The priority Option

Priorities allow multiple modules with the same name to exist in the policy store, with the higher priority module included in the final kernel binary, and all lower priority modules of the same name ignored. For example:

```
semodule --priority 100 --install distribution/apache.pp

semodule --priority 400 --install custom/apache.pp
```

Both apache modules are installed in the policy store as 'apache', but only the custom apache module is included in the final kernel binary. The distribution apache module is ignored. The *--list-modules* options can be used to show these:

```
# Show a full listing of modules:

semodule --list-modules=full
400 apache pp
100 base pp
100 apache pp
```

```
# Show a standard listing of modules:

semodule --list-modules=standard
base
apache
```

The main use case for this is the ability to override a distribution provided policy, while keeping the distribution policy in the store.

This makes it easier for distributions, 3rd parties, configuration management tools (e.g. puppet), local administrators, etc. to update policies without erasing each others changes. This also means that if a distribution, 3rd party etc. updates a module, providing the local customisation is installed at a higher priority, it will override the new distribution policy.

This does require that policy managers adopt some kind of scheme for who uses what priority. No strict guidelines currently exist, however the value used by the *semanage_migrate_store* script is *--priority 100* as this is assumed to be migrating a distribution. If a value is not provided, *semodule* will use a default of *--priority 400* as it is assumed to be a locally customised policy.

When *semodule* builds a lower priority module when a higher priority is already available, the following message will be given: "*A higher priority <name> module exists at priority <999> and will override the module currently being installed at priority <111>*".

Converting policy packages to CIL

A component of the update is to add a facility that converts compiled policy modules (known as policy packages or the **.pp* files) to CIL format. This is achieved via a *pp* to CIL high level language conversion utility located at */usr/libexec/selinux/hll/pp*. This utility can be used manually as follows:

```
cat module_name.pp | /usr/libexec/selinux/hll/pp > module_name.cil
```

There is no man page for '*pp*', however the help text is as follows:

```
Usage: pp [OPTIONS] [IN_FILE [OUT_FILE]]

Read an SELinux policy package (.pp) and output the equivalent CIL.
If IN_FILE is not provided or is -, read SELinux policy package from
standard input. If OUT_FILE is not provided or is -, output CIL to
standard output.

Options:
-h, --help print this message and exit
```

Global Configuration Files

- */etc/selinux/config*
- */etc/selinux/semanage.conf*
- */etc/selinux/restorecond.conf*
- *restorecond-user.conf*
- */etc/selinux/newrole_pam.conf*
- */etc/sestatus.conf*
- */etc/security/sepermit.conf*

Listed in the sections that follow are the common configuration files used by SELinux and are therefore not policy specific. The two most important files are:

- */etc/selinux/config* - This defines the policy to be activated and its enforcing mode.
- */etc/selinux/semanage.conf* - This is used by the SELinux policy configuration subsystem for modular or CIL policies.

/etc/selinux/config

If this file is missing or corrupt no SELinux policy will be loaded (i.e. SELinux is disabled). The man page is ***selinux_config****(5)*, this is because 'config' has already been taken. The config file controls the state of SELinux using the following parameters:

```
SELINUX=enforcing|permissive|disabled
SELINUXTYPE=policy_name
SETLOCALDEFS=0|1
REQUIRESEUSERS=0|1
AUTORELABEL=0|1
```

Where:

SELINUX

This entry can contain one of three values:

- *enforcing* - SELinux security policy is enforced.
- *permissive* - SELinux logs warnings (see the **Auditing SELinux Events** section) instead of enforcing the policy (i.e. the action is allowed to proceed).
- *disabled* - No SELinux policy is loaded. Note that this configures the global SELinux enforcement mode. It is still possible to have domains running in permissive mode and/or object managers running as disabled, permissive or enforcing, when the global mode is enforcing or permissive. Note setting *SELINUX=disabled* will be deprecated at some stage, in favor of the existing kernel command line switch *selinux=0*, which allows users to disable SELinux at system boot. See https://github.com/SELinuxProject/selinux-kernel/wiki/DEPRECATE-runtime-disable that explains how to achieve this on various Linux distributions.

SELINUXTYPE

The *policy_name* is used as the directory name where the active policy and its configuration files will be located. The system will then use this information to locate and load the policy contained within this directory structure. The policy directory must be located at: */etc/selinux*

SETLOCALDEFS

Deprecated - This optional field should be set to 0 (or the entry removed) as the policy store management infrastructure (***semanage****(8)* / ***semodule****(8)*) is now used. If set to 1, then ***init****(8)* and ***load_policy****(8)* will read the local customisation for booleans and users.

REQUIRESEUSERS

Deprecated - This optional field can be used to fail a login if there is no matching or default entry in the *seusers* file or if the file is missing. It is checked by the *libselinux* function ***getseuserbyname****(3)* that is used by SELinux-aware login applications such as ***PAM****(8)*. If it is set to 0 or the entry missing:

- ***getseuserbyname****(3)* will return the GNU / Linux user name as the SELinux user.

If it is set to 1:

- ***getseuserbyname****(3)* will fail.

AUTORELABEL

This is an optional field. If set to '*0*' and there is a file called *.autorelabel* in the root directory, then on a reboot, the loader will drop to a shell where a root logon is required. An administrator can then manually relabel the file system. If set to '1' or the parameter name is not used (the default) there is no login for manual relabeling, however should the */.autorelabel* file exist, then the file system will be automatically relabeled using *fixfiles -F restore*. In both cases the */.autorelabe*l file will be removed so relabeling is not done again.

Example */etc/selinux/config* file contents are:

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
# enforcing - SELinux security policy is enforced.
# permissive - SELinux prints warnings instead of enforcing.
# disabled - No SELinux policy is loaded.
SELINUX=permissive
#
# SELINUXTYPE= can take one of these two values:
# targeted - Targeted processes are protected,
# mls - Multi Level Security protection.
SELINUXTYPE=targeted
```

/etc/selinux/semanage.conf

The ***semanage.config****(5)* file controls the configuration and actions of the ***semanage****(8)* and ***semodule****(8)* set of commands using the following parameters:

```
module-store = method
policy-version = policy_version
expand-check = 0|1
file-mode = mode
save-previous = true|false
save-linked = true|false
disable-genhomedircon = true|false
handle-unknown = allow|deny|reject
bzip-blocksize = 0|1..9
bzip-small true|false
usepasswd = true|false
ignoredirs dir [;dir] …
```

```
store-root = <path>
compiler-directory = <path>
remove-hll = true|false
ignore-module-cache = true|false
target-platform = selinux | xen

[verify kernel]
path = <application_to_run>
args = <arguments>
[end]

[verify module]
path = <application_to_run>
args = <arguments>
[end]

[verify linked]
path = <application_to_run>
args = <arguments>
[end]

[setfiles]
path = <application_to_run>
args = <arguments>
[end]

[sefcontext_compile]
path = <application_to_run>
args = <arguments>
[end]

[load_policy]
path = <application_to_run>
args = <arguments>
[end]
```

Where:

module-store

The method can be one of four options:

1. *directlibsemanage* will write directly to a module store. This is the default value.
2. *sourcelibsemanage* manipulates a source SELinux policy.
3. */foo/bar* - Write via a policy management server, whose named socket is at /foo/bar. The path must begin with a '/'.
4. *foo.com:4242* - Establish a TCP connection to a remote policy management server at *foo.com*. If there is a colon then the remainder is interpreted as a port number; otherwise default to port 4242.

policy-version

This optional entry can contain a policy version number, however it is normally commented out as it then defaults to that supported by the system.

expand-check

This optional entry controls whether hierarchy checking on module expansion is enabled (1) or disabled (0). The default is 0. It is also required to detect the presence of policy rules that are to be excluded with *neverallow* rules.

file-mode

This optional entry allows the file permissions to be set on runtime policy files. The format is the same as the mode parameter of the ***chmod**(1)* command and defaults to 0644 if not present.

save-previous

This optional entry controls whether the previous module directory is saved (TRUE) after a successful commit to the policy store. The default is to delete the previous version (FALSE).

save-linked

This optional entry controls whether the previously linked module is saved (TRUE) after a successful commit to the policy store. Note that this option will create a *base.linked* file in the module policy store. The default is to delete the previous module (FALSE).

disable-genhomedircon

This optional entry controls whether the embedded *genhomedircon* function is run when using the ***semanage**(8)* command. The default is FALSE.

handle-unknown

This optional entry controls the kernel behaviour for handling permissions defined in the kernel but missing from the policy. The options are: *allow* the permission, *reject* by not loading the policy or *deny* the permission. The default is *deny*. Note: to activate any change, the base policy needs to be rebuilt with the *semodule -B* command.

bzip-blocksize

This optional entry determines whether the modules are compressed or not with *bzip*. If the entry is *0*, then no compression will be used (this is required with tools such as ***apol**(1)*. This can also be set to a value between *1* and *9* that will set the block size used for compression (*bzip* will multiply this by 100,000, so '*9*' is faster but uses more memory).

bzip-small

When this optional entry is set to *TRUE* the memory usage is reduced for compression and decompression (the *bzip -s* or *--small* option). If *FALSE* or no entry present, then does not try to reduce memory requirements.

usepasswd

When this optional entry is set to *TRUE semanage* will scan all password records for home directories and set up their labels correctly. If set to *FALSE* (the default if no entry present), then only the */home* directory will be automatically re-labeled.

ignoredirs

With a list of directories to ignore (separated by ';') when setting up users home directories. This is used by some distributions to stop labeling */root* as a home directory.

store-root

Specify an alternative policy store path. The default is */var/lib/selinux*.

compiler-directory

Specify an alternate directory that will hold the High Level Language (HLL) to CIL compilers. The default is */usr/libexec/selinux/hll*.

remove-hll

When set *TRUE*, HLL files will be removed after compilation into CIL (Read ***semanage.conf****(5)* for the consequences of removing these files). Default is *FALSE*.

ignore-module-cache

Whether or not to ignore the cache of CIL modules compiled from HLL. The default is *false*.

target-platform

Target platform for generated policy. Default is *selinux*, the alternate is *xen*.

[verify kernel] .. [end]

Start an additional set of entries that can be used to validate the kernel policy with an external application during the build process. There may be multiple *[verify kernel]* entries. The validation process takes place before the policy is allowed to be inserted into the store with a worked example shown in **Policy Validation Example**

[verify module] .. [end]

Start an additional set of entries that can be used to validate each module by an external application during the build process. There may be multiple *[verify module]* entries.

[verify linked] .. [end]

Start an additional set of entries that can be used to validate module linking by an external application during the build process. There may be multiple *[verify linked]* entries.

[load_policy] .. [end]

Replace the default load policy application with this new policy loader. Defaults are either: */sbin/load_policy* or */usr/sbin/load_policy*.

[setfiles] .. [end]

Replace the default ***setfiles****(8)* application with this new *setfiles*. Defaults are either: */sbin/setfiles* or */usr/sbin/setfiles*.

[sefcontexts_compile] .. [end]

Replace the default file context build application with this new builder. Defaults are either: */sbin/sefcontexts_compile* or */usr/sbin/sefcontexts_compile*

Example *semanage.config* file contents are:

```
# /etc/selinux/semanage.conf
module-store = direct
expand-check = 0

[verify kernel]
path = /usr/local/bin/validate
args = $@
[end]
```

/etc/selinux/restorecond.conf

restorecond-user.conf

The *restorecond.conf* file contains a list of files that may be created by applications with an incorrect security context. The ***restorecond****(8)* daemon will then watch for their creation and automatically correct their security context to that specified by the active policy file context configuration files (located in the */etc/selinux/<SELINUXTYPE>/contexts/files* directory).

Each line of the file contains the full path of a file or directory. Entries that start with a tilde '~' will be expanded to watch for files in users home directories (e.g. *~/public_html* would cause the daemon to listen for changes to *public_html* in all logged on users home directories).

1. It is possible to run *restorecond* in a user session using the *-u* option (see ***restorecond****(8)*). This requires a *restorecond-user.conf* file to be installed as shown in the examples below.
2. The files names and location can be changed if *restorecond* is run with the *-f* option.

Example *restorecond.conf* file contents are:

```
/etc/services
/etc/resolv.conf
/etc/samba/secrets.tdb
/etc/mtab
/var/run/utmp
/var/log/wtmp
```

Example *restorecond-user.conf* file contents are:

```
# This entry expands to listen for all files created for all
# logged in users within their home directories:
~/*
~/public_html/*
```

/etc/selinux/newrole_pam.conf

The optional *newrole_pam.conf* file is used by ***newrole****(1)* and maps commands to ***PAM****(8)* service names.

/etc/sestatus.conf

The ***sestatus.conf****(5)* file is used by the ***sestatus****(8)* command to list files and processes whose security context should be displayed when the -v flag is used (sestatus -v).

The file has the following parameters:

```
[files]
List of files to display context

[process]
List of processes to display context
```

Example *sestatus.conf* file contents are:

```
[files]
/etc/passwd
/etc/shadow
/bin/bash
/bin/login
/bin/sh
/sbin/agetty
/sbin/init
/sbin/mingetty
/usr/sbin/sshd
/lib/libc.so.6
/lib/ld-linux.so.2
/lib/ld.so.1

[process]
/sbin/mingetty
/sbin/agetty
/usr/sbin/sshd
```

/etc/security/sepermit.conf

The ***sepermit.conf****(5)* file is used by the *pam_sepermit.so* module to allow or deny a user login depending on whether SELinux is enforcing the policy or not. An example use of this facility is the Red Hat kiosk policy where a terminal can be set up with a guest user that does not require a password, but can only log in if SELinux is in enforcing mode.

The entry is added to the appropriate */etc/pam.d* configuration file, with the example shown being the */etc/pam.d/gdm-password* file (the **PAM Login Process** section describes PAM in more detail):

```
auth     [success=done ignore=ignore default=bad] pam_selinux_permit.so
auth        substack      password-auth
....
session     optional      pam_gnome_keyring.so auto_start
session     include       postlogin
```

The usage is described in ***pam_sepermit****(5)*, with the following example that describes the configuration:

```
# /etc/security/sepermit.conf
#
# Each line contains either:
#        - a user name
#        - a group name, with @group syntax
#        - a SELinux user name, with %seuser syntax
# Each line can contain optional arguments separated by :
# The possible arguments are:
#        - exclusive - only single login session will
#          be allowed for the user and the user's processes
#          will be killed on logout
#
#        - ignore - The module will never return PAM_SUCCESS status
#          for the user.
#
```

```
# An example entry for 'kiosk mode':
xguest:exclusive
```

Policy Store Configuration Files

NOTE: Files in this area are private and subject to change, they should only be modified using the ***semodule****(8)* and ***semanage****(8)* commands.

Depending on the SELinux userspace library release being used the default policy stores will be located at:

- */etc/selinux/<SELINUXTYPE>/modules* - This is the default for systems that support versions < 2.4 of the SELinux userspace library. The migration process to >= 2.4 is described at https://github.com/SELinuxProject/selinux/wiki/Policy-Store-Migration.
- */var/lib/selinux* - This is the default base for systems that support versions >= 2.4 of the SELinux userspace library. This base may be overridden by the ***store-root*** parameter defined in the ***semanage.conf****(5)* file. Multiple policy stores are supported by appending the <*SELINUXTYPE*>, for example:
  - */var/lib/selinux/mls*
  - */var/lib/selinux/targeted*

The Policy Store files are either installed, updated or built by the ***semodule****(8)* and ***semanage****(8)* commands as a part of the build process with the resulting files being copied over to the **The Policy Store** area.

The policy configuration files are described relative to the default ***store-root*** at */var/lib/selinux* with <*SELINUXTYPE*> being *targeted*.

The ***semanage****(8)* command must be used to configure policy (the actual policy and supporting configuration files) within a specific **Policy Store**. The command types are:

- ***semanage boolean*** Manage booleans to selectively enable functionality
- ***semanage dontaudit*** Disable/Enable *dontaudit* rules in policy
- ***semanage export*** Output local customizations
- ***semanage fcontext*** Manage file context mapping definitions
- ***semanage ibendport*** Manage infiniband end port type definitions
- ***semanage ibpkey*** Manage infiniband pkey type definitions

- ***semanage import*** Import local customizations
- ***semanage interface*** Manage network interface type definitions
- ***semanage login*** Manage login mappings between linux users and SELinux confined users
- ***semanage module*** Manage SELinux policy modules
- ***semanage node*** Manage network node type definitions
- ***semanage permissive*** Manage process type enforcement mode.
- ***semanage port*** Manage network port type definitions
- ***semanage user*** Manage SELinux confined users (Roles and levels for an SELinux user)

active/modules Directory Contents

Under this directory are the respective priority directories containing the compiled modules, plus a '*disabled*' directory listing any disabled modules. For an overview of these 'priority options, see the **SELinux Configuration Files - The priority Option** section.

The modules within the policy store may be compressed or not depending on the value of the *bzip-blocksize* parameter in the ***semanage.conf****(5)* file. The modules and their status can be listed using the ***semanage module -l*** command as shown below.

```
semanage module -l

Module Name               Priority  Language
abrt                      100       pp
accountsd                 100       pp
...
telnet                    100       pp
test_mlsconstrain         400       cil   Disabled
test_overlay_defaultrange 400       cil
test_policy               400       pp
...
```

tmp Policy Store (build failure)

When adding/updating a policy module and it fails to build for some reason, the *tmp* directory (*/var/lib/selinux<SELINUXTYPE>/tmp*) will contain a copy of the failed policy for inspection. An example ***semodule*** failure message indicating the failing line number is:

```
Failed to resolve mlsconstrain statement at /var/lib/selinux/targeted/tmp/modules/400/
test_mlsconstrain/cil:1
```

active/commit_num

This is a binary file used by ***semanage*** for managing updates to the store. The format is not relevant to policy construction.

active/policy.kern

This is the binary policy file built by either the ***semanage****(8)* or ***semodule****(8)* commands (depending on the configuration action), that is then becomes the */etc/selinux/<SELINUXTYPE>/policy/policy.<ver>* binary policy that will be loaded into the kernel.

active/policy.linked

active/seusers.linked

active/seusers_extra.linked

These are saved policy files prior to merging local changes to improve performance.

active/booleans.local

This file is created and updated by the ***semanage boolean*** command and holds boolean values. The booleans must already be present in the policy, as this command effectively turns them on and off, reloading policy (activating the new value) if requested.

Example *semanage boolean* command to modify a boolean value:

```
semanage boolean -m --on daemons_enable_cluster_mode
```

The resulting *booleans.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

daemons_enable_cluster_mode=1
```

disable_dontaudit

This file is only present when the ***semodule**(8)* '-D' flag is used to to build the policy or ***semanage dontaudit***. It indicates that a policy has been built without the *dontaudit* rules. This allows utilities such as ***audit2allow**(8)* to list all denials to assist debugging policy.

active/file_contexts

This file becomes the policy */etc/selinux/<SELINUXTYPE>/contexts/files/file_contexts* file. It is built as described in the **Building the File Labeling Support Files** section.

Example *file_contexts* content entries:

```
# active/file_contexts - These sample entries have
# been taken from the targeted policy.
# The keywords HOME_DIR, HOME_ROOT, USER and ROLE have been
# removed and placed in the homedir_template file.

/.* system_u:object_r:default_t:s0
/[^/]+ -- system_u:object_r:etc_runtime_t:s0
/a?quota.(user|group) -- system_u:object_r:quota_db_t:s0
/nsr(/.*)? system_u:object_r:var_t:s0
/sys(/.*)? system_u:object_r:sysfs_t:s0
/xen(/.*)? system_u:object_r:xen_image_t:s0
/mnt(/[^/]*) -l system_u:object_r:mnt_t:s0
/mnt(/[^/]*)? -d system_u:object_r:mnt_t:s0
```

```
/bin/.* system_u:object_r:bin_t:s0
/dev/.* system_u:object_r:device_t:s0
/usr/.* system_u:object_r:usr_t:s0
/var/.* system_u:object_r:var_t:s0
/run/.* system_u:object_r:var_run_t:s0
/srv/.* system_u:object_r:var_t:s0
/tmp/.* <<none>>
```

```
# ./contexts/files/file_contexts - Sample entries from the
# MLS reference policy.
# Notes:
# 1) The fixed_disk_device_t is labeled SystemHigh (s15:c0.c255)
# as it needs to be trusted. Also some logs and configuration
# files are labeled SystemHigh as they contain sensitive
# information used by trusted applications.
#
# 2) Some directories (e.g. /tmp) are labeled
# SystemLow-SystemHigh (s0-s15:c0.c255) as they will
# support polyinstantiated directories.

/.*system_u:object_r:default_t:s0
/a?quota.(user|group) --system_u:object_r:quota_db_t:s0
/mnt(/[^/]*) -lsystem_u:object_r:mnt_t:s0
/mnt/[^/]*/.*<<none>>
/dev/.*mouse.* -csystem_u:object_r:mouse_device_t:s0
/dev/.*tty[^/]* -csystem_u:object_r:tty_device_t:s0
/dev/[shmx]d[^/]*-bsystem_u:object_r:fixed_disk_device_t:s15:c0.c255
/var/[xgk]dm(/.*)?system_u:object_r:xserver_log_t:s0
/dev/(raw/)?rawctl-csystem_u:object_r:fixed_disk_device_t:s15:c0.c255
/tmp -dsystem_u:object_r:tmp_t:s0-s15:c0.c255
/dev/pts -dsystem_u:object_r:devpts_t:s0-s15:c0.c255
/var/log -dsystem_u:object_r:var_log_t:s0-s15:c0.c255
/var/tmp -dsystem_u:object_r:tmp_t:s0-s15:c0.c255
/var/run -dsystem_u:object_r:var_run_t:s0-s15:c0.c255
/usr/tmp -dsystem_u:object_r:tmp_t:s0-s15:c0.c255
```

Building the File Labeling Support Files

The process used to build the *file_contexts* and the *file_contexts.template* files is shown in **Figure 25: File Context Configuration Files**.

1. They are built by ***semanage**(8)* using entries from the 'Policy File Labeling' statements extracted from the *active/modules* entries (the <*module_name*>*.fc* files, this will include any CIL *(filecon ....)* statements that are within CIL policy files).
2. As a part of the ***semanage**(8)* build process, these two files will also have *file_contexts.bin* and *file_contexts.homedirs.bin* files present in the Policy Configuration Files *./contexts/files* directory. This is because ***semanage**(8)* requires these in the Perl compatible regular expression (PCRE) internal format. They are generated by the ***sefcontext_compile**(8)* utility.

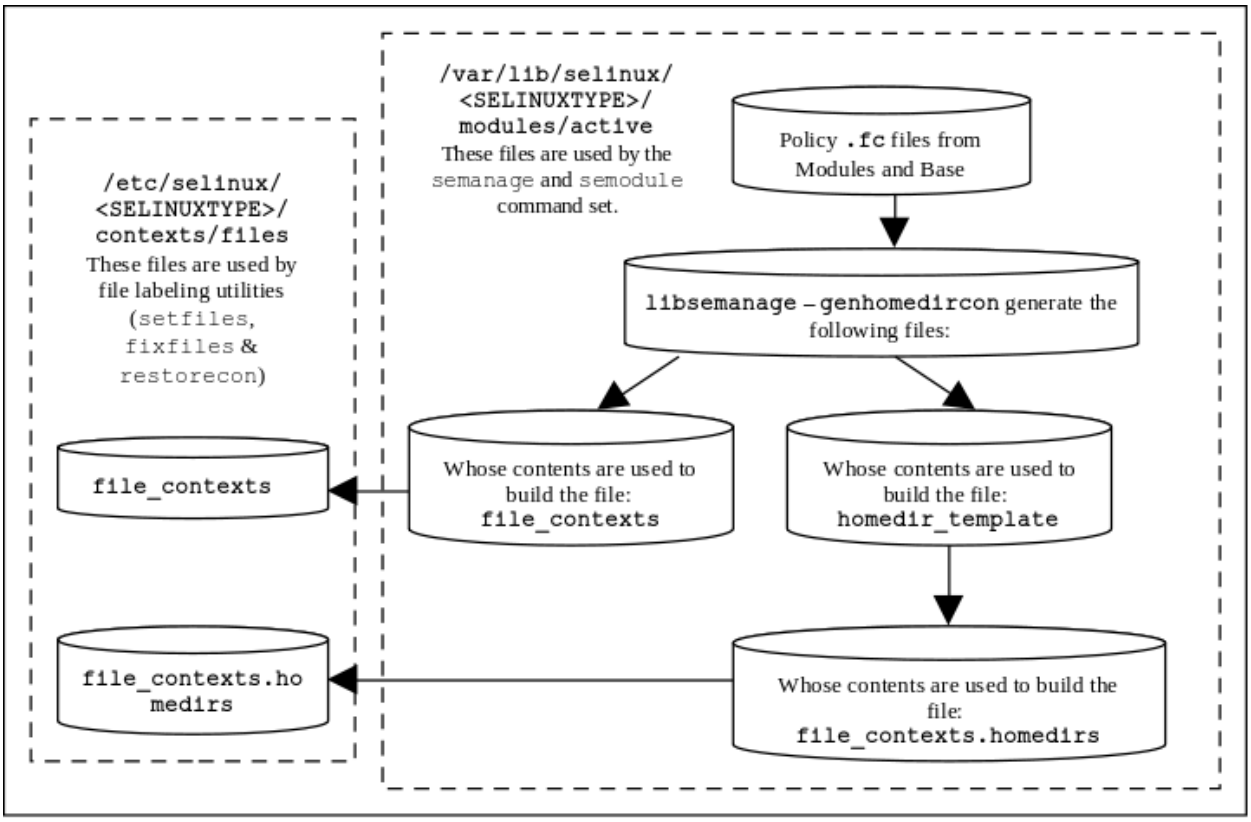


Figure 25: File Context Configuration Files - *The two files copied to the policy area will be used by the file labeling utilities to relabel files.*

The format of these files is:

```
pathname_regexp [file_type] security_context | <<none>>
```

Where:

pathname_regexp

- An entry that defines the pathname that may be in the form of a regular expression. The metacharacters '^' (match beginning of line) and '$' (match end of line) are automatically added to the expression by the routines that process this file, however they can be over-ridden by using '.*' at either the beginning or end of the expression (see the example *file_contexts* files below).
- The source policy **.fc* and *file_contexts.template* files may also contain the keywords *HOME_ROOT*, *HOME_DIR*, *ROLE* and *USER* that will be replaced as explained in the next table.

file_type

- One of the following optional file_type entries (note if blank means "all file types"):
  - *-b* - Block Device
  - *-c* - Character Device
  - *-d* - Directory
  - *-p* - Named Pipe (FIFO)
  - *-l* - Symbolic Link
  - *-s* - Socket File
  - *--* - Ordinary file
- By convention this entry is known as *file type*, however it really represents the 'file object class'.

security_context

- This entry can be either:
  - The security context, including the MLS / MCS level or range if applicable that will be assigned to the file.
  - A value of <<none>> can be used to indicate that matching files should not be re-labeled.

Keywords that can be in policy source *.fc files and then form the *file_contexts.template* file entries are:

HOME_ROOT

- This keyword is replaced by the Linux users root home directory, normally */home* is the default.

HOME_DIR

- This keyword is replaced by the Linux users home directory, normally */home/* is the default.

USER

- This keyword will be replaced by the users Linux user id.

ROLE

- This keyword is replaced by the *prefix* entry from the *users_extra* configuration file that corresponds to the SELinux users user id. Example *users_extra* configuration file entries are:
  - *user user_u prefix user;*
  - *user staff_u prefix staff;*
- It is used for files and directories within the users home directory area. The prefix can be added by the semanage *login* command as follows (although note that the *-P* option is suppressed when help is displayed as it is generally it is not used (defaults to *user*)):

```
# Add a Linux user:

adduser rch
# Modify staff_u SELinux user and prefix:
semanage user -m -R staff_r -P staff staff_u

# Associate the SELinux user to the Linux user:
semanage login -a -s staff_u rch
```

Example policy source file from Reference Policy *policy/modules/system/userdomain.fc*:

```
HOME_DIR    -d  gen_context(system_u:object_r:user_home_dir_t,s0-mls_systemhigh)
HOME_DIR    -l  gen_context(system_u:object_r:user_home_dir_t,s0-mls_systemhigh)
HOME_DIR/.+     gen_context(system_u:object_r:user_home_t,s0)
/tmp/gconfd-USER -d gen_context(system_u:object_r:user_tmp_t,s0)
/root(/.*)?     gen_context(system_u:object_r:admin_home_t,s0)
/root/\.cert(/.*)?  gen_context(system_u:object_r:home_cert_t,s0)
/root/\.pki(/.*)?   gen_context(system_u:object_r:home_cert_t,s0)
/root/\.debug(/.*)? <<none>>
```

Example policy source file from Reference Policy *policy/modules/kernel/files.fc*:

```
#
# HOME_ROOT
# expanded by genhomedircon
```

```
#
HOME_ROOT       -d  gen_context(system_u:object_r:home_root_t,s0-mls_systemhigh)
HOME_ROOT/home-inst     -d  gen_context(system_u:object_r:home_root_t,s0-
mls_systemhigh)
HOME_ROOT/\-inst        -d  gen_context(system_u:object_r:home_root_t,s0-
mls_systemhigh)
HOME_ROOT       -l  gen_context(system_u:object_r:home_root_t,s0)
HOME_ROOT/\.journal     <<none>>
HOME_ROOT/lost\+found   -d  gen_context(system_u:object_r:lost_found_t,mls_systemhigh)
HOME_ROOT/lost\+found/.*    <<none>>

# HOME_ROOT is where user HOME dirs reside as per /etc/default/useradd, but
# there may be a compat symlink from /home to HOME_ROOT. We want to make sure
# that symlink itself is always labeled home_root_t so that e.g. systemd can
# getattr as it follows it.
/home       -l  gen_context(system_u:object_r:home_root_t,s0)
```

active/file_contexts.local

This file is created and updated by the ***semanage fcontext*** command. It is used to hold file context information on files and directories that were not delivered by the core policy (i.e. they are not defined in any of the **.fc* files delivered in the base and loadable modules).

The ***semanage*** command will add the information to the *file_contexts.local* file and then copy this file to the ./contexts/files/file_contexts.local file, where it will be used when the file context utilities are run.

The format of the *file_contexts.local* file is the same as the *modules/active/file_contexts* file.

Example ***semanage fcontext*** command to add a new entry:

```
semanage fcontext -a -t unlabeled_t /usr/move_file
```

The resulting *file_contexts.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

/usr/move_file    system_u:object_r:unlabeled_t:s0
```

active/homedir_template

This file is built as described in the **Building the File Labeling Support Files** section and is used to build the *active/file_contexts.homedirs* file.

Example file contents:

```
# ./active/homedir_template - These sample entries have
# been taken from the targeted policy and show the
# HOME_DIR, HOME_ROOT and USER keywords that are used to manage
# users home directories:

/tmp/gconfd-USER/.* --  system_u:object_r:gconf_tmp_t:s0
```

```
/tmp/gconfd-USER    -d  system_u:object_r:user_tmp_t:s0
...
HOME_ROOT/home-inst -d  system_u:object_r:home_root_t:s0
...
HOME_ROOT/\.journal <<none>>
...
HOME_DIR/.+ system_u:object_r:user_home_t:s0
```

active/file_contexts.homedirs

This file becomes the policy */etc/selinux/<SELINUXTYPE>/contexts/files/file_contexts.homedirs* file. It is built as described in the **Building the File Labeling Support Files** section. It is then used by the file labeling utilities to ensure that users home directory areas are labeled according to the policy.

The file can be built by the ***genhomedircon****(8)* command (that just calls */usr/sbin/semodule -Bn*) or if using ***semanage*** with user or login options to manage users, where it is called automatically as it is now a libsepol library function.

Example *file_contexts.homedirs* **contents:**

```
# ./active/file_contexts.homedirs - These sample entries have
# been taken from the targeted policy and show how the
# HOME_DIR, HOME_ROOT and USER keywords have been transformed as they
# follow the entries from the ./active/homedir_template sample above:

/tmp/gconfd-[^/]+/.*    --  unconfined_u:object_r:gconf_tmp_t:s0
/tmp/gconfd-[^/]+   -d  unconfined_u:object_r:user_tmp_t:s0
...
/home/home-inst -d  system_u:object_r:home_root_t:s0
...
/home/\.journal <<none>>
...
#
# Home Context for user user_u
#
/home/[^/]+/.+  unconfined_u:object_r:user_home_t:s0
```

active/seusers

active/seusers.local

The *active/seusers* file becomes the policy */etc/selinux/<SELINUXTYPE>/seusers* file, the *active/seusers.local* file holds entries added when adding users via ***semanage****(8)*.

The *seusers* file is built or modified when:

1. Building a policy where an optional *seusers* file has been included in the base package via the ***semodule_package****(8)* command (signified by the *-s* flag) as follows:
    - *semodule_package -o base.pp -m base.mod -s seusers ...*
    - The *seusers* file would be extracted by the subsequent ***semodule*** command when building the policy to produce the *seusers.final* file.
2. The ***semanage login*** command is used to map Linux users to SELinux users as follows:
    - *semanage login -a -s staff_u rch*

    - This action will update the *seusers* file that would then be used to produce the seusers.final file with both policy and locally defined user mapping.
3. It is also possible to associate a Linux group of users to an SELinux user as follows:
    - *semanage login -a -s staff_u %staff_group*

The format of the *seusers* and *seusers.local* **files are as follows:**

```
[%]user_id:seuser_id[:range]
```

Where:

user_id

- Where *user_id* is the Linux user identity. If this is a Linux *group_id* then it will be preceded with the '*%*' sign as shown in the example below.

seuser_id

- The SELinux user identity.

range

- The optional *level* or *range*.

Example *seusers* **file contents:**

```
# active/modules/seusers
root:unconfined_u:s0-s0:c0.c1023
__default__:unconfined_u:s0-s0:c0.c1023
```

now use ***semanage login*** command to add a Linux user:

```
semanage login -a -s user_u rch
```

the resulting *seusers.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

rch:user_u:s0
```

and the resulting *seusers* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

root:unconfined_u:s0-s0:c0.c1023
__default__:unconfined_u:s0-s0:c0.c1023
rch:user_u:s0
```

active/users_extra

active/users_extra.local

active/users.local

These three files work together to describe SELinux user information as follows:

1. The *users_extra* and *users_extra.local* files are used to map a *prefix* to a users home directories as discussed in the **Building the File Labeling Support Files** section, where it is used to replace the *ROLE* keyword. The *prefix* is linked to an SELinux user id and should reflect the users role.
    - The *users_extra* file contains all the policy *prefix* entries, and the *users_extra.local* file contains those generated by the ***semanage user*** command.
    - The *users_extra* file can optionally be included in the base package via the ***semodule_package****(8)* command (signified by the *-u* flag) as follows:
    - *semodule_package -o base.pp -m base.mod -u users_extra ...*
    - The *users_extra* file would then be extracted by a subsequent semodule command when building the policy.
2. The *users.local* file is used to add new SELinux users to the policy without editing the policy source itself (with each line in the file following a policy language **user Statement**). This is useful when only the Reference Policy headers are installed and additional users need to be added. The ***semanage user*** command will allow a new SELinux user to be added that would generate the *user.local* file and if a *-P* flag has been specified, then a *users_extra.local* file is also updated (note: if this is a new SELinux user and a prefix is not specified a default *prefix* of user is generated).

The format of the *users_extra* and *users_extra.local* **files are:**

```
user seuser_id prefix prefix_id;
```

Where:

user

- The user keyword.

seuser_id

- The SELinux user identity.

prefix

- The prefix keyword.

prefix_id

- An identifier that will be used to replace the *ROLE* keyword within the *active/homedir_template* file when building the *active/file_contexts.homedirs* file for the relabeling utilities to set the security context on users home directories.

Example *users_extra* **file contents:**

```
user user_u prefix user;
user staff_u prefix user;
user sysadm_u prefix user;
user root prefix user;
```

Example *semanage user* command to add a new SELinux user:

```
semanage user -a -R staff_r -P staff test_u
```

the resulting *users_extra.local* file is as follows:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

user test_u prefix staff;
```

and the resulting *users_extra* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

user user_u prefix user;
user staff_u prefix user;
user sysadm_u prefix user;
user root prefix user;
user test_u prefix staff;
```

and the resulting *users.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

user test_u roles { staff_r } level s0 range s0;
```

active/interfaces.local

This file is created and updated by the ***semanage interface*** command to hold network interface information that was not delivered by the core policy. The new interface information is then built into the policy.

Each line of the file contains a *netifcon* statement that is defined in the ***netifcon*** section.

Example *semanage interface* command:

```
semanage interface -a -t netif_t -r s0:c20.c250 enp7s0
```

The resulting *interfaces.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

netifcon enp7s0 system_u:object_r:netif_t:s0:c20.c250
system_u:object_r:netif_t:s0:c20.c250
```

active/nodes.local

This file is created and updated by the ***semanage node*** command to hold network address information that was not delivered by the core policy. The new node information is then built into policy.

Each line of the file contains a *nodecon* statement that is defined along with examples in the policy language ***nodecon*** section.

Example *semanage node* command:

```
semanage node -a -M 255.255.255.255 -t node_t -r s0:c20.c250 -p ipv4 127.0.0.2
```

The resulting *nodes.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

nodecon ipv4 127.0.0.2 255.255.255.255 system_u:object_r:node_t:s0:c20.c250
```

active/ports.local

This file is created and updated by the ***semanage port*** command to hold network port information that was not delivered by the core policy. The new port information is then built into the policy.

Each line of the file contains a *portcon* statement that is defined along with examples in the policy language ***portcon*** section.

Example *semanage port* command:

```
semanage port -a -t port_t -p tcp -r s0:c20.c350 8888
```

The resulting *ports.local* file will be:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

portcon tcp 8888 system_u:object_r:port_t:s0:c20.c350
```

Set domain permissive mode

The ***semanage permissive*** command will either add or remove a policy module that sets the requested domain in permissive mode.

Example *semanage permissive* command to set permissive mode:

```
semanage permissive -a tabrmd_t
```

This will by default add a CIL policy module to *active/modules/400/permissive_tabrmd_t*, that if expanded will contain:

```
(typepermissive tabrmd_t)
```

Note that the CIL *typepermissive* statement is used, the equivalent kernel policy statement would be ***permissive***.

Policy Configuration Files

- setrans.conf
- *secolor.conf*
- *policy/policy.<ver>*
- *contexts/customizable_types*
- *contexts/default_contexts*
- *contexts/dbus_contexts*
- *contexts/default_type*
- *contexts/failsafe_context*
- *contexts/initrc_context*
- *contexts/lxc_contexts*
- *contexts/netfilter_contexts* - Obsolete
- *contexts/openrc_contexts*
- *contexts/openssh_contexts*
- *contexts/removable_context*
- *contexts/sepgsql_contexts*
- *contexts/snapperd_contexts*
- *contexts/securetty_types*
- *contexts/systemd_contexts*
- *contexts/userhelper_context*
- *contexts/virtual_domain_context*
- *contexts/virtual_image_context*
- *contexts/x_contexts*
- *contexts/files/file_contexts*
- *contexts/files/file_contexts.local*
- *contexts/files/file_contexts.homedirs*
- *contexts/files/file_contexts.subs*
- *contexts/files/file_contexts.subs_dist*
- *contexts/files/media*
- *contexts/users/[seuser_id]*
- *logins/<linuxuser_id>*
- *users/local.users*

Each file discussed in this section is relative to the policy name as follows:

```
/etc/selinux/<SELINUXTYPE>
```

All files under this area form the 'running policy' once the */etc/selinux/config* files **SELINUXTYPE=policy_name** entry is set to load the policy.

The majority of files located here are installed by the ***semodule****(8)* and/or ***semanage****(8)* commands. However it is possible to build a custom monolithic policy that only use the files installed in this area (i.e. do not use ***semanage****(8)* or ***semodule****(8)*). For example the simple **policy** described in the Notebook examples could run at init 3 (i.e. no X-Windows) and only require the following configuration files:

- *seusers* - For login programs.
- *policy/policy.<ver>* - The binary policy loaded into the kernel.
- *context/files/file_contexts* - To allow the filesystem to be relabeled.

If the simple policy is to run at init 5, (i.e. with X-Windows) then an additional two files are required:

- *context/dbus_contexts* - To allow the dbus messaging service to run under SELinux.
- *context/x_contexts* - To allow the X-Windows service to run under SELinux.

seusers

The ***seusers**(5)* file is used by login programs (normally via the *libselinux* library) and maps GNU / Linux users (as defined in the *user* / *passwd* files) to SELinux users (defined in the policy). A typical login sequence would be:

- Using the GNU / Linux *user_id*, lookup the *seuser_id* from this file. If an entry cannot be found, then use the *__default__* entry.
- To determine the remaining context to be used as the security context, read the *contexts/users/[seuser_id]* file. If this file is not present, then:
- Check for a default context in the *contexts/default_contexts* file. If no default context is found, then:
- Read the *contexts/failsafe_context* file to allow a fail safe context to be set.

Note: The *system_u* user is defined in this file, however there must be **no** *system_u* Linux user configured on the system.

The format of the *seusers* file is the same as the files described in the *active/seusers* section, where an example ***semanage**(8)* user command is also shown.

Example *seusers* file contents:

```
# seusers file for an MLS system. Note that the system_u user
# has access to all security levels and therefore should not be
# configured as a valid Linux user.

root:root:s0-s15:c0.c1023
__default__:user_u:s0-s0
```

Supporting libselinux API functions are:

- ***getseuser**(3)*
- ***getseuserbyname**(3)*

booleans

booleans.local

NOTE: These were removed in libselinux 3.0

Generally these ***booleans**(5)* files are not present unless using older Reference policies. ***semanage**(8)* is now used to manage booleans via the Policy Store as described in the *active/booleans.local* file section.

For systems that do use these files:

- ***security_set_boolean_list**(3)* - Writes a *boolean.local* file if flag *permanent* = '*1*'.
- ***security_load_booleans**(3)* - Will look for a *booleans* or *booleans.local* file here unless a specific path is specified.

Both files have the same format and contain one or more boolean names.

The format is:

```
boolean_name value
```

Where:

boolean_name

- The name of the boolean.

value

- The default setting for the boolean that can be one of the following:
  - *true* | *false* | *1* | *0*

Note that if *SETLOCALDEFS* is set in the SELinux */etc/selinux/config* file, then ***selinux_mkload_policy**(3)* will check for a *booleans.local* file in the ***selinux_booleans_path**(3)*, and also a *local.users* file in the ***selinux_users_path**(3)*.

booleans.subs_dist

The *booleans.subs_dist* file (if present) will allow new boolean names to be allocated to those in the active policy. This file was added because many older booleans began with '*allow*' that made it difficult to determine what they did. For example the boolean *allow_console_login* becomes more descriptive as *login_console_enabled*. If the *booleans.subs_dist* file is present, then either name may be used. ***selinux_booleans_subs_path**(3)* will return the active policy path to this file and ***selinux_boolean_sub**(3)* will will return the translated name.

Each line within the substitution file *booleans.subs_dist* is:

```
policy_bool_name new_name
```

Where:

policy_bool_name

- The policy boolean name.

new_name

- The new boolean name.

Example:

```
# booleans.subs_dist
allow_auditadm_exec_content auditadm_exec_content
allow_console_login         login_console_enabled
allow_cvs_read_shadow       cvs_read_shadow
allow_daemons_dump_core     daemons_dump_core
```

When ***security_get_boolean_names**(3)* or ***security_set_boolean**(3)* is called with a boolean name and the *booleans.subs_dist* file is present, the name will be looked up and if using the *new_name*, then the *policy_bool_name* will be used (as that is what is defined in the active policy).

Supporting libselinux API functions are:

- ***selinux_booleans_subs_path**(3)*
- ***selinux_booleans_sub**(3)*
- ***security_get_boolean_names**(3)*
- ***security_set_boolean**(3)*

setrans.conf

The ***setrans.conf****(8)* file is used by the ***mcstransd****(8)* daemon (available in the mcstrans rpm). The daemon enables SELinux-aware applications to translate the MCS / MLS internal policy levels into user friendly labels.

There are a number of sample configuration files within the *mcstrans* package that describe the configuration options in detail that are located at */usr/share/mcstrans/examples*.

The daemon will not load unless a valid MCS or MLS policy is active.

The translations can be disabled by adding the following line to the file:

```
disable = 1
```

This file will also support the display of information in colour. The configuration file that controls this is called *secolor.conf* and is described in the *secolor.conf* section.

The file format is fully described in ***setrans.conf****(8)*.

Example file contents:

```
# UNCLASSIFIED
Domain=NATOEXAMPLE

s0=SystemLow
s15:c0.c1023=SystemHigh
s0-s15:c0.c1023=SystemLow-SystemHigh

Base=Sensitivity Levels
s1=UNCLASSIFIED
s3:c0,c2,c11,c200.c511=RESTRICTED
s4:c0,c2,c11,c200.c511=CONFIDENTIAL
s5:c0,c2,c11,c200.c511=SECRET

s1:c1=NATO UNCLASSIFIED
s3:c1,c200.c511=NATO RESTRICTED
s4:c1,c200.c511=NATO CONFIDENTIAL
s5:c1,c200.c511=NATO SECRET

Include=/etc/selinux/mls/setrans.d/rel.conf
Include=/etc/selinux/mls/setrans.d/eyes-only.conf
Include=/etc/selinux/mls/setrans.d/constraints.conf

# UNCLASSIFIED
```

Supporting libselinux API functions are:

- ***selinux_translations_path****(3)*
- ***selinux_raw_to_trans_context****(3)*
- ***selinux_trans_to_raw_context****(3)*

secolor.conf

The **secolor.conf**(5) file controls the colour to be associated to the components of a context when information is displayed by an SELinux colour-aware application (currently none, although there are two examples in the Notebook source tarball under the *libselinux/examples* directory).

The file format is as follows:

```
color color_name = #color_mask

context_component string fg_color_name bg_color_name
```

Where:

color

- The color keyword.

color_name

- A descriptive name for the colour (e.g. *red*).

color_mask

- A colour mask starting with a hash '*#*' that describes the RGB colours with black being *#000000* and white being *#ffffff*.

context_component

- The colour translation supports different colours on the context string components (*user*, *role*, *type* and *range*). Each component is on a separate line.

string

- This is the *context_component* string that will be matched with the *raw* context component passed by ***selinux_raw_context_to_color****(3)*. A wildcard '*' may be used to match any undefined *string* for the *user*, *role* and *type context_component* entries only.

fg_color_name

- The *color_name* string that will be used as the foreground colour. A *color_mask* may also be used.

bg_color_name

- The *color_name* string that will be used as the background colour. A *color_mask* may also be used.

Example file contents:

```
color black = #000000
color green = #008000
color yellow = #ffff00
color blue = #0000ff
color white = #ffffff
color red = #ff0000
color orange = #ffa500
color tan = #D2B48C
user * = black white
role * = white black
```

```
type * = tan orange
range s0-s0:c0.c1023 = black green
range s1-s1:c0.c1023 = white green
range s3-s3:c0.c1023 = black tan
range s5-s5:c0.c1023 = white blue
range s7-s7:c0.c1023 = black red
range s9-s9:c0.c1023 = black orange
range s15:c0.c1023 = black yellow
```

Supporting libselinux API functions are:

- ***selinux_colors_path****(3)*
- ***selinux_raw_context_to_color****(3)* - this call returns the foreground and background colours of the context string as the specified RGB 'colour' hex digits as follows:

```
user : role : type : range
#000000 #ffffff #ffffff #000000 #d2b48c #ffa500 #000000 #008000
black white white black tan orange black green
```

policy/policy.<ver>

This is the binary policy file that is loaded into the kernel to enforce policy and is built by either ***checkpolicy****(8)* or ***semodule****(8)*. Life is too short to describe the format but the libsepol source could be used as a reference or for an overview the "**SELinux Policy Module Primer**" notes.

By convention the file name extension is the policy database version used to build the policy, however is is not mandatory as the true version is built into the policy file. The different policy versions are discussed in the **Types of SELinux Policy - Policy Versions** section.

contexts/customizable_types

The ***customizable_types****(5)* file contains a list of types that will not be relabeled by the ***setfiles****(8)* or **restorecon**(8) commands. The commands check this file before relabeling and excludes those in the list unless the *-F* flag is used (see the man pages).

The file format is as follows:

```
type
```

Where:

type

- The type defined in the policy that needs to excluded from relabeling. An example is when a file has been purposely relabeled with a different type to allow an application to work.

Example file contents:

```
mount_loopback_t
public_content_rw_t
public_content_t
swapfile_t
```

```
sysadm_untrusted_content_t
sysadm_untrusted_content_tmp_t
```

Supporting libselinux API functions are:

- ***is_context_customizable****(3)*
- ***selinux_customizable_types_path****(3)*
- ***selinux_context_path****(3)*

contexts/default_contexts

The ***default_contexts****(5)* file is used by SELinux-aware applications that need to set a security context for user processes (generally the login applications) where:

1. The GNU / Linux user identity should be known by the application.
2. If a login application, then the SELinux user (seuser), would have been determined as described in the *seusers* file section.
3. The login applications will check the *contexts/users/[seuser_id]* file first and if no valid entry, will then look in the *[seuser_id]* file for a default context to use.

The file format is as follows:

```
role:type[:range] role:type[:range] ...
```

Where:

role:type[:range]

- The file contains one or more lines that consist of *role:type[:range]* pairs (including the MLS / MCS *level* or *range* if applicable).
- The entry at the start of a new line corresponds to the partial *role:type[:range]* context of (generally) the login application.
- The other *role:type[:range]* entries on that line represent an ordered list of valid contexts that may be used to set the users context.

Example file contents:

```
system_r:crond_t:s0 system_r:system_crond_t:s0
system_r:local_login_t:s0 user_r:user_t:s0
system_r:remote_login_t:s0 user_r:user_t:s0
system_r:sshd_t:s0 user_r:user_t:s0
system_r:sulogin_t:s0 sysadm_r:sysadm_t:s0
system_r:xdm_t:s0 user_r:user_t:s0
```

Supporting libselinux API functions are:

Note that the *contexts/users/[seuser_id]* file is also read by some of these functions.

- ***selinux_contexts_path****(3)*
- ***selinux_default_context_path****(3)*
- ***get_default_context****(3)*
- ***get_ordered_context_list****(3)*
- ***get_ordered_context_list_with_level****(3)*
- ***get_default_context_with_level****(3)*
- ***get_default_context_with_role****(3)*

- ***get_default_context_with_rolelevel**(3)*
- ***query_user_context**(3)*
- ***manual_user_enter_context**(3)*

An example use in this Notebook (to get over a small feature) is that when the initial **basic policy** was built, no default_contexts file entries were required as only one *role:type* of *unconfined_r:unconfined_t* had been defined, therefore the login process did not need to decide anything (as the only user context was *unconfined_u:unconfined_r:unconfined_t*).

However when adding the **loadable module** that used another type (*ext_gateway_t*) but with the same role and user (e.g. *unconfined_u:unconfined_r:ext_gateway_t*), then it was found that the login process would always set the logged in user context to *unconfined_u:unconfined_r:ext_gateway_t* (i.e. the login application now had a choice and chose the wrong one, probably because the types are sorted and 'e' comes before 'u').

The end result was that as soon as enforcing mode was set, the system got bitter and twisted. To resolve this the *default_contexts* file entries were set to:

```
unconfined_r:unconfined_t unconfined_r:unconfined_t
```

The login process could now set the context correctly to *unconfined_r:unconfined_t*. Note that adding the same entry to the *contexts/users/unconfined_u* configuration file instead could also have achieved this.

contexts/dbus_contexts

This file is for the dbus messaging service daemon (a form of IPC) that is used by a number of GNU / Linux applications such as GNOME and KDE desktops. If SELinux is enabled, then this file needs to exist in order for these applications to work. The ***dbus-daemon**(1)* man page details the contents and the Free Desktop web site has detailed information at:

http://dbus.freedesktop.org

Example file contents:

```
<!DOCTYPE busconfig PUBLIC "-//freedesktop//DTD D-BUS Bus Configuration 1.0//EN"
 "http://www.freedesktop.org/standards/dbus/1.0/busconfig.dtd">
<busconfig>
  <selinux>
  </selinux>
</busconfig>
```

Supporting libselinux API function is:

- ***selinux_context_path**(3)*

contexts/default_type

The **default_type**(5) file allows SELinux-aware applications such as **newrole**(1) to select a default type for a role if one is not supplied.

The file format is as follows:

```
role:type
```

Where:

role:type

- The file contains one or more lines that consist of *role:type* entries. There should be one line for each role defined within the policy.

Example file contents:

```
auditadm_r:auditadm_t
secadm_r:secadm_t
sysadm_r:sysadm_t
staff_r:staff_t
unconfined_r:unconfined_t
user_r:user_t
```

Supporting libselinux API functions are:

- ***selinux_default_type_path**(3)*
- ***get_default_type**(3)*

contexts/failsafe_context

The **failsafe_context**(5) is used when a login process cannot determine a default context to use. The file contents will then be used to allow an administrator access to the system.

The file format is as follows:

```
role:type[:range]
```

Where:

role:type[:range]

- A single line that has a valid context to allow an administrator access to the system, including the MLS / MCS *level* or *range* if applicable.

Example file contents:

```
# Taken from the MLS policy.

sysadm_r:sysadm_t:s0
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*
- ***selinux_failsafe_context_path**(3)*
- ***get_default_context**(3)*
- ***get_default_context_with_role**(3)*
- ***get_default_context_with_level**(3)*
- ***get_default_context_with_rolelevel**(3)*
- ***get_ordered_context_list**(3)*
- ***get_ordered_context_list_with_level**(3)*

contexts/initrc_context

This is used by the ***run_init****(8)* command to allow system services to be started in the same security context as init. This file could also be used by other SELinux-aware applications for the same purpose.

The file format is as follows:

```
user:role:type[:range]
```

Where:

user:role:type[:range]

- The file contains one line that consists of a security context, including the MLS / MCS *level* or *range* if applicable.

Example file contents:

```
# Taken from the MLS policy
# Note that the init process has full access via the range s0-s15:c0.c255.

system_u:system_r:initrc_t:s0-s15:c0.c255
```

Supporting libselinux API functions are:

- ***selinux_context_path****(3)*

contexts/lxc_contexts

This file supports labeling lxc containers within the *libvirt* library (see libvirt source *src/security/security_selinux.c*).

The file format is as follows:

```
process = "security_context"
file = "security_context"
content = "security_context"
```

Where:

process

- A single *process* entry that contains the lxc domain security context, including the MLS / MCS *level* or *range* if applicable.

file

- A single *file* entry that contains the lxc file security context, including the MLS / MCS *level* or *range* if applicable.

content

- A single *content* entry that contains the lxc content security context, including the MLS / MCS *level* or *range* if applicable.

sandbox_kvm_process

sandbox_lxc_process

- These entries may be present and contain the security context.

Example file contents:

```
process = "system_u:system_r:container_t:s0"
content = "system_u:object_r:virt_var_lib_t:s0"
file = "system_u:object_r:container_file_t:s0"
ro_file="system_u:object_r:container_ro_file_t:s0"
sandbox_kvm_process = "system_u:system_r:svirt_qemu_net_t:s0"
sandbox_kvm_process = "system_u:system_r:svirt_qemu_net_t:s0"
sandbox_lxc_process = "system_u:system_r:container_t:s0"
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*
- ***selinux_lxc_context_path**(3)*

contexts/netfilter_contexts - Obsolete

This file was to support the Secmark labeling for Netfilter / iptable rule matching of network packets - Never been used.

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*
- ***selinux_netfilter_context_path**(3)*

contexts/openrc_contexts

OpenRC is a dependency-based init system that works with the system-provided *init* program, normally */sbin/init*. This config file will only be present if *openrc* is installed, see **https://github.com/OpenRC/openrc**

The file format is as follows:

```
run_init=[domain]
```

Where:

run_init

- The keyword *run_init*. Note that there must not be any spaces around the '=' sign.

domain

- The domain type for the process.

Example file contents:

```
run_init=run_init_t
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*

- ***selinux_openrc_contexts_path**(3)*

contexts/openssh_contexts

Used by *openssh* (***ssh**(1)*) for privilege separated processes in the preauthentication phase. This is a Red Hat specific policy configuration file.

The file format is as follows:

```
privsep_preauth=[domain]
```

Where:

privsep_preauth

- The keyword *privsep_preauth*

domain

- The domain type for the privilege separated processes in the preauthentication phase.

Example file contents:

```
privsep_preauth=sshd_net_t
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*
- ***selinux_openssh_contexts_path**(3)*

contexts/removable_context

The **removable_context**(5) file contains a single default label that should be used for removable devices that are not defined in the *contexts/files/media* file.

The file format is as follows:

```
user:role:type[:range]
```

Where:

user:role:type[:range]

- The file contains one line that consists of a security context, including the MLS / MCS *level* or *range* if applicable.

Example file contents:

```
system_u:object_r:removable_t:s0
```

Supporting libselinux API functions are:

- ***selinux_removable_context_path**(3)*

contexts/sepgsql_contexts

This file contains the default security contexts for SE-PostgreSQL database objects and is described in ***selabel_db**(5)*.

The file format is as follows:

```
object_type object_name context
```

Where:

object_type

- This is the string representation of the object type.

object_name

- These are the object names of the specific database objects. The entry can contain '*' for wildcard matching or '*?*' for substitution. Note that if the '*' is used, then be aware that the order of entries in the file is important. The '*' on its own is used to ensure a default fallback context is assigned and should be the last entry in the *object_type* block.

context

- The security *context* that will be applied to the object.

Example file contents:

```
# object_type  object_name  context
db_database    my_database  system_u:object_r:my_sepgsql_db_t:s0
db_database        *        system_u:object_r:sepgsql_db_t:s0
db_schema         *.*       system_u:object_r:sepgsql_schema_t:s0
```

contexts/snapperd_contexts

Used by ***snapper**(8)* for filesystem snapshot management to set an SELinux context on ***btrfs**(8)* subvolumes. This is a Red Hat specific policy configuration file.

The file format is as follows:

```
snapperd_data = user:role:type[:range]
```

Where:

snapperd_data

- The keyword *snapperd_data*

user:role:type[:range]

- The security context including the MLS / MCS *level* or *range* if applicable.

Example file contents:

```
snapperd_data = system_u:object_r:snapperd_data_t:s0
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*
- ***selinux_snapperd_contexts_path**(3)*

contexts/securetty_types

The ***securetty_types**(5)* file is used by the ***newrole**(1)* command to find the type to use with tty devices when changing roles or levels.

The file format is as follows:

```
type
```

Where:

type

- Zero or more type entries that are defined in the policy for tty devices.

Example file contents:

```
sysadm_tty_device_t
user_tty_device_t
staff_tty_device_t
```

Supporting libselinux API functions are:

- ***selinux_securetty_types_path**(3)*

contexts/systemd_contexts

This file contains security contexts to be used by tasks run via ***systemd**(8)*.

The file format is as follows:

```
service_class = security_context
```

Where:

service_class

- One or more entries that relate to the ***systemd**(1)* service (e.g. runtime, transient).

security_context

- The security context, including the MLS / MCS *level* or *range* if applicable of the service to be run.

Example file contents:

```
runtime=system_u:object_r:systemd_runtime_unit_file_t:s0
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*
- ***selinux_systemd_contexts_path**(3)*

contexts/userhelper_context

This file contains the default security context used by the system-config-* applications when running from root.

The file format is as follows:

```
security_context
```

Where:

security_context

- The file contains one line that consists of a full security context, including the MLS / MCS *level* or *range* if applicable.

Example file contents:

```
system_u:sysadm_r:sysadm_t:s0
```

Supporting libselinux API functions are:

- ***selinux_context_path**(3)*

contexts/virtual_domain_context

The ***virtual_domain_context**(5)* file is used by the virtulization API (*libvirt*) and provides the qemu domain contexts available in the policy (see libvirt source *src/security/security_selinux.c*). There may be two entries in this file, with the second entry being an alternative domain context.

Example file contents:

```
system_u:system_r:svirt_t:s0
system_u:system_r:svirt_tcg_t:s0
```

Supporting libselinux API functions are:

- ***selinux_virtual_domain_context_path**(3)*

contexts/virtual_image_context

The ***virtual_image_context**(5)* file is used by the virtulization API (*libvirt*) and provides the image contexts that are available in the policy (see libvirt source *src/security/security_selinux.c*). The first entry is the image file context and the second entry is the image content context.

Example file contents:

```
system_u:object_r:svirt_image_t:s0
system_u:object_r:virt_content_t:s0
```

Supporting libselinux API functions are:

- ***selinux_virtual_image_context_path**(3)*

contexts/x_contexts

The ***x_contexts**(5)* file provides the default security contexts for the X-Windows SELinux security extension. The usage is discussed in the SELinux X-Windows Support section. The MCS / MLS version of the file has the appropriate level or range information added.

A typical entry is as follows:

```
# object_type object_name  context
selection      PRIMARY     system_u:object_r:clipboard_xselection_t:s0
```

Where:

object_type

- These are types of object supported and valid entries are: *client*, *property*, *poly_property*, *extension*, *selection*, *poly_selection* and *events*.

object_name

- These are the object names of the specific X-server resource such as *PRIMARY*, *CUT_BUFFER0* etc. They are generally defined in the X-server source code (*protocol.txt* and *BuiltInAtoms* in the *dix* directory of the *xorg-server* source package). This can contain '*' for 'any' or '*?*' for 'substitute' (see the *CUT_BUFFER?* entry where the '*?*' would be substituted for a number between 0 and 7 that represents the number of these buffers).

context

- This is the security context that will be applied to the object. For MLS/MCS systems there would be the additional MLS label.

Supporting libselinux API functions are:

- ***selinux_x_context_path**(3)*
- ***selabel_open**(3)*
- ***selabel_close**(3)*
- ***selabel_lookup**(3)*
- ***selabel_stats**(3)*

contexts/files/file_contexts

The ***file_contexts**(5)* file is managed by the ***semodule**(8)* and ***semanage**(8)* commands[12] as the policy is updated (adding or removing modules or updating the base), and therefore should not be edited.

The file is used by a number of SELinux-aware commands (***setfiles**(8)*, ***fixfiles**(8)*, ***restorecon**(8)* to relabel either part or all of the file system.

Note that users home directory file contexts are not present in this file as they are managed by the *file_contexts.homedirs* file as explained below.

The format of the *file_contexts* file is the same as the files described in the *active/file_contexts* file section.

There may also be a *file_contexts.bin* present that is built and used by ***semanage**(8)*. The format of this file conforms to the Perl compatible regular expression (PCRE) internal format.

Supporting libselinux API functions are:

- ***selinux_file_context_path**(3)*
- ***selabel_open**(3)*
- ***selabel_close**(3)*
- ***selabel_lookup**(3)*
- ***selabel_stats**(3)*

contexts/files/file_contexts.local

This file is added by the ***semanage fcontext*** command as described in the *active/file_contexts.local* file section to allow locally defined files to be labeled correctly. The ***file_contexts**(5)* man page also describes this file.

Supporting libselinux API functions are:

- ***selinux_file_context_local_path**(3)*

contexts/files/file_contexts.homedirs

This file is managed by the ***semodule**(8)* and ***semanage**(8)* commands as the policy is updated (adding or removing users and modules or updating the base), and therefore should not be edited.

It is generated by the ***genhomedircon**(8)* command (in fact by ***semodule -Bn*** that rebuilds the policy) and used to set the correct contexts on the users home directory and files.

It is fully described in the *active/file_contexts.homedirs* file section. The ***file_contexts**(5)* man page also describes this file.

There may also be a *file_contexts.homedirs.bin* present that is built and used by ***semanage**(8)*. The format of this file conforms to the Perl compatible regular expression (PCRE) internal format.

Supporting libselinux API functions are:

- ***selinux_file_context_homedir_path**(3)*
- ***selinux_homedir_context_path**(3)*

contexts/files/file_contexts.subs

contexts/files/file_contexts.subs_dist

These files allow substitution of file names (*.subs* for local use and *.subs_dist* for GNU / Linux distributions use) for the *libselinux* function ***selabel_lookup**(3)*. The ***file_contexts**(5)* man page also describes this file.

The subs files contain a list of space separated path names such as:

```
/myweb /var/www
/myspool /var/spool/mail
```

Then (for example), when ***selabel_lookup**(3)* is passed a path */myweb/index.html* the functions will substitute the */myweb* component with */var/www*, with the final result being:

```
/var/www/index.html
```

Supporting libselinux API functions are:

- ***selinux_file_context_subs_path**(3)*
- ***selinux_file_context_subs_dist_path**(3)*
- ***selabel_lookup**(3)*
- ***matchpathcon**(3)* (deprecated)
- ***matchpathcon_index**(3)* (deprecated)

contexts/files/media

The **media**(5)* file is used to map media types to a file context. If the media_id cannot be found in this file, then the default context in the *contexts/removable_context* is used instead.

The file format is as follows:

```
media_id file_context
```

Where:

media_id

- The media identifier (those known are: cdrom, floppy, disk and usb).

file_context

- The context to be used for the device. Note that it does not have the MLS / MCS level).

Example file contents:

```
cdrom system_u:object_r:removable_device_t:s0
floppy system_u:object_r:removable_device_t:s0
disk system_u:object_r:fixed_disk_device_t:s0
```

Supporting libselinux API functions are:

- ***selinux_media_context_path**(3)*

contexts/users/[seuser_id]

These optional files are named after the SELinux user they represent. Each file has the same format as the *contexts/default_contexts* file and is used to assign the correct context to the SELinux user (generally during login). The ***user_contexts**(5)* man page also describes these entries.

Example file contents - From the 'targeted` *unconfined_u* file:

```
system_r:crond_t:s0     unconfined_r:unconfined_t:s0
unconfined_r:unconfined_cronjob_t:s0
system_r:initrc_t:s0        unconfined_r:unconfined_t:s0
system_r:local_login_t:s0   unconfined_r:unconfined_t:s0
system_r:remote_login_t:s0  unconfined_r:unconfined_t:s0
system_r:rshd_t:s0      unconfined_r:unconfined_t:s0
```

```
system_r:sshd_t:s0      unconfined_r:unconfined_t:s0
system_r:cockpit_session_t:s0   unconfined_r:unconfined_t:s0
system_r:sysadm_su_t:s0     unconfined_r:unconfined_t:s0
system_r:unconfined_t:s0    unconfined_r:unconfined_t:s0
system_r:xdm_t:s0       unconfined_r:unconfined_t:s0
system_r:init_t:s0      unconfined_r:unconfined_t:s0
```

Supporting libselinux API functions are:

- ***selinux_user_contexts_path**(3)*
- ***selinux_users_path**(3)*
- ***selinux_usersconf_path**(3)*
- ***get_default_context**(3)*
- ***get_default_context_with_role**(3)*
- ***get_default_context_with_level**(3)*
- ***get_default_context_with_rolelevel**(3)*
- ***get_ordered_context_list**(3)*
- ***get_ordered_context_list_with_level**(3)*

logins/<linuxuser_id>

These optional files are used by SELinux-aware login applications such as PAM (using the *pam_selinux* module) to obtain an SELinux user name and level based on the GNU / Linux login id and service name. It has been implemented for SELinux-aware applications such as FreeIPA (Identity, Policy Audit - see http://freeipa.org/page/Main_Page for details). The ***service_seusers**(5)* man page also describes these entries.

The file name is based on the GNU/Linux user that is used at log in time (e.g. *ipa*).

If ***getseuser**(3)* fails to find an entry, then the *seusers* file is used to retrieve default information.

The file format is as follows:

```
service_name:seuser_id:level
```

Where:

service_name

- The name of the service.

seuser_id

- The SELinux user name.

level

- The run level

Example file contents:

```
# logins/ipa example entries
ipa_service:user_u:s0
another_service:unconfined_u:s0
```

Supporting libselinux API functions are:

- ***getseuser**(3)*

users/local.users

NOTE: These were removed in libselinux 3.0

Generally the ***local.users**(5)* file is not present if ***semanage**(8)* is being used to manage users, however if ***semanage*** is not being used then this file may be present (it could also be present in older Reference or Example policies).

The file would contain local user definitions in the form of *user* statements as defined in the *active/users.local* section.

Note that if *SETLOCALDEFS* is set in the SELinux */etc/selinux/config* file, then ***selinux_mkload_policy**(3)* will check for a *booleans.local* file in the ***selinux_booleans_path**(3)*, and also a *local.users* file in the ***selinux_users_path**(3)*.

The SELinux Policy Languages

There are two methods of writing 'raw' policy statements and rules:

1. The **Kernel Policy Language** section is intended as a reference of the kernel policy language statements and rules with supporting examples taken from the Reference Policy sources. Also all of the language updates to Policy DB version 32 should have been captured. For a more detailed explanation of the policy language the **SELinux by Example** book is recommended.
2. The Common Intermediate Language (CIL) project defines a new policy definition language that has an overview of its motivation and design at: https://github.com/SELinuxProject/cil/wiki, however some of the language statement definitions are out of date. The **CIL Policy Language** section gives an overview.

However more likely, policy is written using the **The Reference Policy** as the base with kernel policy statements and rules added as required, for example:

```
################################
#
# Policy for testing stat operations
#

attribute test_stat_domain;                  ### Kernel policy statement ###

# Types for test file.
type test_stat_file_t;                       ### Kernel policy statement ###
files_type(test_stat_file_t)                 ### Reference Policy macro  ###

# Domain for process that can get attributes on the test file.
type test_stat_t;
domain_type(test_stat_t)
unconfined_runs_test(test_stat_t)
typeattribute test_stat_t test_stat_domain;
typeattribute test_stat_t testdomain;
allow test_stat_t test_stat_file_t:file getattr;   ### Kernel policy rule ###

# Domain for process that cannot set attributes on the test file.
type test_nostat_t;
domain_type(test_nostat_t)
unconfined_runs_test(test_nostat_t)
typeattribute test_nostat_t test_stat_domain;
typeattribute test_nostat_t testdomain;

# TODO: what is a replacement for this in refpolicy???
# Allow all of these domains to be entered from sysadm domain
require {                                        ### Modular Policy rule ###
    type ls_exec_t;
}
domain_transition_pattern(sysadm_t, ls_exec_t, test_stat_domain)
domain_entry_file(test_stat_domain, ls_exec_t)
```

CIL Overview

The CIL language statements are not included in the SELinux Notebook as they have been documented within the CIL compiler source, available at:

https://github.com/SELinuxProject/selinux/tree/master/secilc/docs

A PDF version is included in this documentation: **CIL Reference Guide**

The CIL compiler source can be found at: https://github.com/SELinuxProject/selinux.git within the *secilc* and *libsepol* sections and can be cloned via: - *git clone https://github.com/SELinuxProject/selinux.git*

While the CIL design web pages give the main objectives of CIL, from a language perspective it will:

- Apply name and usage consistency to the current kernel language statements. For example the kernel language uses *attribute* and *attribute_role* to declare identifiers, whereas CIL uses *typeattribute* and *roleattribute*. Also statements to associate types or roles have been made consistent and enhanced to allow expressions to be defined. Some examples are:

Kernel	CIL
attribute	*typeattribute*
typeattribute	*typeattributeset*
attribute_role	*roleattribute*
roleattribute	*roleattributeset*
allow	*allow*
allow (role)	*roleallow*
dominance	*sensitivityorder*

- Additional CIL statements have been defined to enhance functionality:
  - *classpermission* - Declare a *classpermissionset* identifier.
  - *classpermissionset* - Associate class / permissions also supporting expressions.
  - *classmap* / *classmapping* - Statements to support declaration and association of multiple *classpermissionset*'s. Useful when defining an *allow* rule with multiple class/permissions.
  - *context* - Statement to declare security context.
- Allow named and anonymous definitions to be supported.
- Support namespace features allowing policy modules to be defined within blocks with inheritance and template features.
- Remove the order dependency in that policy statements can be anywhere within the source (i.e. remove dependency of class, sid etc. being within a base module).
- Able to define macros and calls that will remove any dependency on M4 macro support.
- Directly generate the binary policy file and other configuration files - currently the *file_contexts* file.
- Support transformation services such as delete, transform and inherit with exceptions.

An simple CIL policy is as follows:

```
; These CIL statements declare a user, role, type and range of:
;    unconfined.user:unconfined.role:unconfined.process:s0-s0
;
; A CIL policy requires at least one 'allow' rule and sid to be declared
```

```
; before a policy will build.
;
(handleunknown allow)
(mls true)
(policycap open_perms)
(category c0)
(categoryorder (c0))
(sensitivity s0)
(sensitivityorder (s0))
(sensitivitycategory s0 (c0))
(level systemLow (s0))
(levelrange low_low (systemLow systemLow))
(sid kernel)
(sidorder (kernel))
(sidcontext kernel unconfined.sid_context)
(classorder (file))
(class file (read write open getattr))

; Define object_r role. This must be assigned in CIL.
(role object_r)

; The unconfined namespace:
(block unconfined
    (user user)
    (userrange user (systemLow systemLow))
    (userlevel user systemLow)
    (userrole user role)
    (role role)
    (type process)
    (roletype object_r process)
    (roletype role process)

    ; Define a SID context:
    (context sid_context (user role process low_low))

    (type object)
    (roletype object_r object)

    ; An allow rule:
    (allow process object (file (read)))
)
```

For more complex CIL policy try these: https://github.com/DefenSec

There is a CIL policy in the Notebook examples with a utility that will produce a base policy in either the kernel policy language or CIL (*notebook-tools/build-sepolicy*). The only requirement is that the *initial_sids*, *security_classes* and *access_vectors* files from the Reference policy are required, although the Fedora 31 versions are supplied in the *notebook-examples/selinux-policy/flask-files* directory.

An example output: **CIL policy**

```
Usage: build-sepolicy [-k] [-M] [-c|-p|-s] -d flask_directory -o output_file
-k    Output kernel classes only.
-M    Output an MLS policy.
-c    Output a policy in CIL language (else generate kernel policy language policy).
```

```
-p    Output a file containing class and classpermissionsets + their order for use by
CIL policies.
-s    Output a file containing initial SIDs + their order for use by CIL policies.
-d    Directory containing the initial_sids, security_classes and access_vectors Flask
files.
-o    The output file that will contain the policy source or header file.
```

There is another CIL policy in the notebook examples called “cil-policy” that takes a slightly different approach where the goal is to keep the policy as simple as possible. It requires *semodule*, Linux 5.7, SELinux 3.1 and can be installed by executing *make install*. It leverages some modern SELinux features, most notably where the requirement for ordered security classes is lifted. With this you are no longer expected to be aware of all the access vectors managed by Linux in order to align your security class declarations with the order in which they are declared in the kernel. A module store is created by *semodule* to give easy access to the source and that allows for full control over the policy.

Kernel Policy Language

This section covers the policy source file types and what kernel policy statements and rule are allowed in each. The **Policy Language Index** then has links to each section within this document.

Policy Source Files

There are three basic types of policy source file[13] that can contain language statements and rules. The three types of policy source file[14] are:

Monolithic Policy - This is a single policy source file that contains all statements. By convention this file is called policy.conf and is compiled using the ***checkpolicy****(8)* command that produces the binary policy file.

Base Policy - This is the mandatory base policy source file that supports the loadable module infrastructure. The whole system policy could be fully contained within this file, however it is more usual for the base policy to hold the mandatory components of a policy, with the optional components contained in loadable module source files. By convention this file is called base.conf and is compiled using the ***checkpolicy****(8)* or ***checkmodule****(8)* command.

Module (or Non-base) Policy - These are optional policy source files that when compiled, can be dynamically loaded or unloaded within the policy store. By convention these files are named after the module or application they represent, with the compiled binary having a '.pp' extension. These files are compiled using the ***checkmodule****(8)* command.

Table 1 shows the order in which the statements should appear in source files with the mandatory statements that must be present.

Base Entries	M/O
Security Classes (class)	m
Initial SIDs	m
Access Vectors (permissions)	m
require Statement	o
MLS sensitivity, category and level Statements	o
MLS Constraints	m
Policy Capability Statements	o
Attributes	o
Booleans	o
Default user, role, type, range rules	o
Type / Type Alias	m

Base Entries	M/O
Roles	m
Policy Rules (allow, dontaudit etc.)	m
Users	m
Constraints	o
Default SID labeling	m
fs_use_xattr Statements	o
fs_use_task and fs_use_trans Statements	o
genfscon Statements	o
portcon, netifcon and nodecon Statements	o
Module Entries	**M/O**
module Statement	m
require Statement	o
Attributes	o
Booleans	o
Type / Type Alias	o
Roles	o
Policy Rules	o
Users	o

Table 1: Base and Module Policy Statements - *There must be at least one of each of the mandatory statements, plus at least one allow rule in a policy to successfully build.*

The language grammar defines what statements and rules can be used within the different types of source file. To highlight these rules, the following table is included in each statement and rule section to show what circumstances each one is valid within a policy source file.

Policy Type:

Monolithic Policy	Base Policy	Module Policy
Yes/No	Yes/No	Yes/No

Where:

Monolithic Policy

- Whether the statement is allowed within a monolithic policy source file or not.

Base Policy

- Whether the statement is allowed within a base (for loadable module support) policy source file or not.

Module Policy

- Whether the statement is allowed within the optional loadable module policy source file or not.

Conditional, Optional and Require Statement Rules

The language grammar specifies what statements and rules can be included within:

1. **Conditional Policy** rules that are part of the kernel policy language.
2. *optional* and *require* rules that are NOT part of the kernel policy language, but **Reference Policy *m4(1)*** macros used to control policy builds (see the **Modular Policy Support Statements** section.

To highlight these rules the following table is included in each statement and rule section to show what circumstances each one is valid within a policy source file:

Conditional Policy Statements:

***if* Statement**	***optional* Statement**	***require* Statement**
Yes/No	Yes/No	Yes/No

Where:

if Statement

- Whether the statement is allowed within a conditional statement (*if/else* construct). Conditional statements can be in all types of policy source file.

optional Statement

- Whether the statement is allowed within the *optional { rule_list }* construct.

require Statement

- Whether the statement is allowed within the *require { rule_list }* construct.

MLS Statements and Optional MLS Components

The **MLS Statements** section defines statements specifically for MLS support. However when MLS is enabled, there are other statements that require the MLS component of a security context as an argument, (for example the **Network Labeling Statements**), therefore these statements show an example taken from the MLS **Reference Policy** build.

General Statement Information

1. Identifiers can generally be any length but should be restricted to the following characters: a-z, A-Z, 0-9 and _ (underscore).
2. A '#' indicates the start of a comment in policy source files.
3. All statements available to policy version 29 have been included.
4. When multiple source and target entries are shown in a single statement or rule, the compiler (***checkpolicy****(8)* or ***checkmodule****(8)*) will expand these to individual statements or rules as shown in the following example:

```
# This allow rule has two target entries console_device_t and tty_device_t:
allow apm_t { console_device_t tty_device_t }:chr_file { getattr read write append
ioctl lock };
```

```
# The compiler will expand this to become:
allow apm_t console_device_t:chr_file { getattr read write append ioctl lock };
# and:
allow apm_t tty_device_t:chr_file { getattr read write append ioctl lock };
```

Therefore when comparing the actual source code with a compiled binary using (for example) ***apol****(8)*, **sedispol** or **sedismod**, the results will differ (however the resulting policy rules will be the same).

1. Some statements can be added to a policy via the policy store using the **semanage**(8) command. Examples of these are shown where applicable, however the **semanage** man page should be consulted for all the possible command line options.
2. **Table 2** lists words reserved for the SELinux policy language.

alias	allow	allowxperm	and
attribute	attribute_role	auditallow	auditallowxperm
auditdeny	bool	category	cfalse
class	clone	common	constrain
ctrue	default_range	default_role	default_type
default_user	dom	domby	dominance
dontaudit	else	equals	expandattribute
false	filename	filesystem	fscon
fs_use_task	fs_use_trans	fs_use_xattr	genfscon
h1	h2	high	ibendportcon
ibpkeycon	identifier	if	incomp
inherits	iomemcon	ioportcon	ipv4_addr
ipv6_addr	l1	l2	level
low	low_high	mlsconstrain	mlsvalidatetrans
module	netifcon	neverallow	neverallowxperm
neverallowxperm	nodecon	not	notequal
number	object_r	optional	or
path	pcidevicecon	permissive	pirqcon
policycap	portcon	r1	r2
r3	range	range_transition	require
role	roleattribute	roles	role_transition
sameuser	sensitivity	sid	source
t1	t2	t3	target

true	type	typealias	typeattribute
typebounds	type_change	type_member	types
type_transition	u1	u2	u3
user	validatetrans	version_identifier	xor

Table 2: Policy language reserved words

Table 3 shows what policy language statements and rules are allowed within each type of policy source file, and whether the statement is valid within an *if/else* construct, *optional {rule_list}*, or *require {rule_list}* statement.

Statement / Rule	Monolithic Policy	Base Policy	Module Policy	Conditional Statements	optional Statement	require Statement
allow	Yes	Yes	Yes	Yes	Yes	No
allow - Role	Yes	Yes	Yes	No	Yes	No
allowxperm	Yes	Yes	Yes	No	No	No
attribute	Yes	Yes	Yes	No	Yes	Yes
attribute_role	Yes	Yes	Yes	No	Yes	Yes
auditallow	Yes	Yes	Yes	Yes	Yes	No
auditallowxperm	Yes	Yes	Yes	No	No	No
auditdeny (Deprecated)	Yes	Yes	Yes	Yes	Yes	No
bool	Yes	Yes	Yes	No	Yes	Yes
category	Yes	Yes	No	No	No	Yes
class	Yes	Yes	No	No	No	Yes
common	Yes	Yes	No	No	No	No
constrain	Yes	Yes	No	No	No	No
default_user	Yes	Yes	No	No	No	No
default_role	Yes	Yes	No	No	No	No
default_type	Yes	Yes	No	No	No	No
default_range	Yes	Yes	No	No	No	No
dominance - MLS	Yes	Yes	No	No	No	No
dominance - Role (Deprecated)	Yes	Yes	Yes	No	Yes	No

Statement / Rule	Monolithic Policy	Base Policy	Module Policy	Conditional Statements	optional Statement	require Statement
dontaudit	Yes	Yes	Yes	Yes	Yes	No
dontauditxperm	Yes	Yes	Yes	No	No	No
expandattribute	Yes	Yes	Yes	No	Yes	No
fs_use_task	Yes	Yes	No	No	No	No
fs_use_trans	Yes	Yes	No	No	No	No
fs_use_xattr	Yes	Yes	No	No	No	No
genfscon	Yes	Yes	No	No	No	No
ibpkeycon	Yes	Yes	Yes	No	No	No
ibendportcon	Yes	Yes	Yes	No	No	No
if	Yes	Yes	Yes	No	Yes	No
level	Yes	Yes	No	No	No	No
mlsconstrain	Yes	Yes	No	No	No	No
mlsvalidatetrans	Yes	Yes	No	No	No	No
module	No	No	Yes	No	No	No
netifcon	Yes	Yes	No	No	No	No
neverallow	Yes	Yes	Yes [15]	No	Yes	No
neverallowxperm	Yes	Yes	Yes	No	No	No
nodecon	Yes	Yes	No	No	No	No
optional	No	Yes	Yes	No	Yes	No
permissive	Yes	Yes	Yes	Yes	Yes	No
policycap	Yes	Yes	No	No	No	No
portcon	Yes	Yes	No	No	No	No
range_transition	Yes	Yes	Yes	No	Yes	No
require	No	Yes [16]	Yes	Yes [17]	Yes	No
role	Yes	Yes	Yes	No	Yes	Yes
roleattribute	Yes	Yes	Yes	No	Yes	No

Statement / Rule	Monolithic Policy	Base Policy	Module Policy	Conditional Statements	optional Statement	require Statement
role_transition	Yes	Yes	Yes	No	Yes	No
sensitivity	Yes	Yes	No	No	No	Yes
sid	Yes	Yes	No	No	No	No
type	Yes	Yes	Yes	No	No	Yes
type_change	Yes	Yes	Yes	Yes	Yes	No
type_member	Yes	Yes	Yes	Yes	Yes	No
type_transition	Yes	Yes	Yes	Yes [18]	Yes	No
typealias	Yes	Yes	Yes	No	Yes	No
typeattribute	Yes	Yes	Yes	No	Yes	No
typebounds	Yes	Yes	Yes	No	Yes	No
user	Yes	Yes	Yes	No	Yes	Yes
validatetrans	Yes	Yes	No	No	No	No

Table 3: The policy language statements and rules that are allowed within each type of policy source file - *The left hand side of the table shows what Policy Language Statements and Rules are allowed within each type of policy source file. The right hand side of the table shows whether the statement is valid within the if/else construct, optional {rule_list}, or require {rule_list} statement.*

Policy Language Index

The policy language statement and rule sections are as follows:

Note these are not kernel policy statements, but used by the Reference Policy to assist policy build:

- Modular Policy Support Statements

Policy Configuration Statements

policycap

Policy version 22 introduced the *policycap* statement to allow new capabilities to be enabled or disabled in the kernel via policy in a backward compatible way. For example policies that are aware of a new capability can enable the functionality, while older policies would continue to use the original functionality.

The statement definition is:

```
policycap capability;
```

Where:

policycap

The *policycap* keyword.

capability

A single *capability* identifier that will be enabled for this policy.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# This statement enables the network_peer_controls to be enabled
# for use by the policy.
#
policycap network_peer_controls;
```

Adding A New Policy Capability

The kernel, userspace libsepol library and policy must be updated to enable the new capability as described below. For readability, the new capability should follow a consistent naming convention, where:

- policy capability identifier is a lower-case string.
- enum definition is `POLICYDB_CAP_` with the identifier appended in upper-case.
- kernel function call is `selinux_policycap_` with the identifier appended in lower-case.

Kernel Updates

In kernel source update the following three files with the new capability:

security/selinux/include/policycap_names.h

Add new entry at end of this list:

```
/* Policy capability names */
const char *selinux_policycap_names[__POLICYDB_CAP_MAX] = {
    ...
    "genfs_seclabel_symlinks",
    "ioctl_skip_cloexec",
    "new_name"
};
```

security/selinux/include/policycap.h

Add new entry at end of this list:

```
/* Policy capabilities */
enum {
    ...
    POLICYDB_CAP_GENFS_SECLABEL_SYMLINKS,
    POLICYDB_CAP_IOCTL_SKIP_CLOEXEC,
    POLICYDB_CAP_NEW_NAME,
    __POLICYDB_CAP_MAX
};
```

security/selinux/include/security.h

Add a new call to retrieve the loaded policy capability state:

```
static inline bool selinux_policycap_new_name(void)
{
    struct selinux_state *state = &selinux_state;

    return READ_ONCE(state->policycap[POLICYDB_CAP_NEW_NAME]);
}
```

Finally in the updated code that utilises the new policy capability do something like:

```
if (selinux_policycap_new_name())
    do this;
```

```
else
    do that;
```

libsepol Library Updates

In selinux userspace source update the following two files with the new capability:

selinux/libsepol/src/polcaps.c

Add new entry at end of this list:

```
static const char * const polcap_names[] = {
    ...
    "genfs_seclabel_symlinks",  /* POLICYDB_CAP_GENFS_SECLABEL_SYMLINKS */
    "ioctl_skip_cloexec",       /* POLICYDB_CAP_IOCTL_SKIP_CLOEXEC */
    "new_name",         /* POLICYDB_CAP_NEW_NAME */
    NULL
};
```

selinux/libsepol/include/sepol/policydb/polcaps.h

Add new entry at end of this list:

```
/* Policy capabilities */
enum {
    ...
    POLICYDB_CAP_GENFS_SECLABEL_SYMLINKS,
    POLICYDB_CAP_IOCTL_SKIP_CLOEXEC,
    POLICYDB_CAP_NEW_NAME,
    __POLICYDB_CAP_MAX
};
```

Reference Policy Updates

To enable the new capability in Reference Policy, add a new entry to this file:

policy/policy_capabilities

New *policycap* statement added to end of file:

```
# A description of the capability
policycap new_name;
```

To disable the capability, comment out the entry:

```
# A description of the capability
#policycap new_name;
```

CIL Policy Updates

To enable the capability in policy, add the following entry to a CIL source file:

```
; A description of the capability
(policycap new_name)
```

Default Object Rules

These rules allow a default user, role, type and/or range to be used when computing a context for a new object. These require policy version 27 or 28 with kernels 3.5 or greater, for *glblub* support version 32 with kernel 5.5 is required.

default_user

Allows the default user to be taken from the source or target context when computing a new context for an object of the defined class. Requires policy version 27.

The statement definition is:

```
default_user class default;
```

Where:

default_user

The *default_user* rule keyword.

class

One or more *class* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'.

default

A single keyword consisting of either *source* or *target* that will state whether the default user should be obtained from the source or target context.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# When computing the context for a new file object, the user
# will be obtained from the target context.
```

```
default_user file target;
```

```
# When computing the context for a new x_selection or x_property
# object, the user will be obtained from the source context.

default_user { x_selection x_property } source;
```

default_role

Allows the default role to be taken from the source or target context when computing a new context for an object of the defined class. Requires policy version 27.

The statement definition is:

```
default_role class default;
```

Where:

default_role

The *default_role* rule keyword.

class

One or more *class* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'.

default

A single keyword consisting of either *source* or *target* that will state whether the default role should be obtained from the source or target context.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# When computing the context for a new file object, the role
# will be obtained from the target context.

default_role file target;
```

```
# When computing the context for a new x_selection or x_property
# object, the role will be obtained from the source context.

default_role { x_selection x_property } source;
```

default_type

Allows the default type to be taken from the source or target context when computing a new context for an object of the defined class. Requires policy version 28.

The statement definition is:

```
default_type class default;
```

Where:

default_type

The *default_type* rule keyword.

class

One or more *class* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'.

default

A single keyword consisting of either *source* or *target* that will state whether the default type should be obtained from the source or target context.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# When computing the context for a new file object, the type
# will be obtained from the target context.

default_type file target;
```

```
# When computing the context for a new x_selection or x_property
# object, the type will be obtained from the source context.
```

```
default_type { x_selection x_property } source;
```

default_range

Allows the default range or level to be taken from the source or target context when computing a new context for an object of the defined class. Requires policy version 27.

Policy version 32 with kernel 5.5 allows the use of *glblub* as a *default_range* default and the computed transition will be the intersection of the MLS range of the two contexts. The *glb* (greatest lower bound) *lub* (lowest upper bound) of a range is calculated as the greater of the low sensitivities and the lower of the high sensitivities.

The statement definition is:

```
default_range class [default range] | [glblub];
```

Where:

default_range

The *default_range* rule keyword.

class

One or more *class* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'.

default

A single keyword consisting of either *source* or *target* that will state whether the default level or range should be obtained from the source or target context.

range

A single keyword consisting of either: *low*, *high* or *low_high* that will state what part of the range should be used.

glblub

The *glblub* keyword used instead of *[default range]*.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# When computing the context for a new file object, the lower
# level will be taken from the target context range.

default_range file target low;
```

```
# When computing the context for a new x_selection or x_property
# object, the range will be obtained from the source context.

default_type { x_selection x_property } source low_high;
```

```
# When computing the context of a new object with a level of:
#     s0-s1:c0.c12
#  and a level of:
#     s0-s1:c0.c1023
# the resulting computed level will be:
#     s0-s1:c0.c12

default_range db_table glblub;
```

User Statements

user

The user statement declares an SELinux user identifier within the policy and associates it to one or more roles. The statement also allows an optional MLS level and range to control a users security level. It is also possible to add SELinux user id's outside the policy using the 'semanage user' command that will associate the user with roles previously declared within the policy.

The statement definition is:

```
user seuser_id roles role_id;
```

Or for MCS/MLS Policy:

```
user seuser_id roles role_id level mls_level range mls_range;
```

Where:

user

The *user* keyword.

seuser_id

The SELinux user identifier.

roles

The *roles* keyword.

role_id

One or more previously declared *role* or *attribute_role* identifiers. Multiple *role* identifiers consist of a space separated list enclosed in braces '{}'.

level

If MLS is configured, the MLS *level* keyword.

mls_level

The users default MLS security level that has been previously declared with a *level* statement. Note that the compiler only accepts the *sensitivity* component of the *level* (e.g. s0).

range

If MLS is configured, the MLS *range* keyword.

mls_range

The range of security levels that the user can run. The format is described in the "MLS *range* Definition" section.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	Yes	Yes

Examples:

```
# Using the user statement to define an SELinux user user_u that
# has been assigned the role of user_r. The SELinux user_u is a
# generic user identity for Linux users who have no specific
# SELinux user identity defined.
#

user user_u roles { user_r };
```

MLS Examples:

```
# Using the user statement to define an MLS SELinux user user_u
# that has been assigned the role of user_r and has a default
# login security level of s0 assigned, and is only allowed
# access to the s0 range of security levels:

user user_u roles { user_r } level s0 range s0;
```

```
# Using the user statement to define an MLS SELinux user
# sysadm_u that has been assigned the role of sysadm_r and has
# a default login security level of s0 assigned, and is
# allowed access to the range of security levels (low - high)
# between s0 and s15:c0.c255:

user sysadm_u roles { sysadm_r } level s0 range s0-s15:c0.c255;
```

***semanage**(8)* **Command example:**

```
# Add user mque_u to SELinux and associate to the unconfined_r role:

semanage user -a -R unconfined_r mque_u
```

This command will produce the following files in the default <SELINUXTYPE> policy store and then activate the policy:

/var/lib/selinux/<SELINUXTYPE>/active/users.local:

```
# This file is auto-generated by libsemanage
# Do not edit directly.
```

```
user mque_u roles { unconfined_r } ;
```

/var/lib/selinux/<SELINUXTYPE>/active/users_extra:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

user mque_u prefix user;
```

/var/lib/selinux/<SELINUXTYPE>/active/users_extra.local:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

user mque_u prefix user;
```

Role Statements

Policy version 26 introduced two new role statements aimed at replacing the deprecated role *dominance* rule by making role relationships easier to understand. These new statements: *attribute_role* and *roleattribute* are defined in this section with examples.

role

The *role* statement either declares a role identifier or associates a role identifier to one or more types (i.e. authorise the role to access the domain or domains). Where there are multiple role statements declaring the same role, the compiler will associate the additional types with the role.

The statement definition to declare a role is:

```
role role_id;
```

The statement definition to associate a role to one or more types is:

```
role role_id types type_id;
```

Where:

role

The *role* keyword.

role_id

The identifier of the role being declared. The same *role* identifier can be declared more than once in a policy, in which case the *type_id* entries will be amalgamated by the compiler.

types

The optional *types* keyword.

type_id

When used with the *types* keyword, one or more type, *typealias* or *attribute* identifiers associated with the *role_id*. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'. For *role* statements, only *type*, *typealias* or *attribute* identifiers associated to domains have any meaning within SELinux.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	Yes	Yes

Examples:

```
# Declare the roles:
role system_r;
role sysadm_r;
role staff_r;
role user_r;
role secadm_r;
role auditadm_r;

# Within the policy the roles are then associated to the
# required types with this example showing the user_r role
# being associated to two domains:

role user_r types user_t;
role user_r types chfn_t;
```

attribute_role

The *attribute_role* statement declares a role attribute identifier that can then be used to refer to a group of roles.

The statement definition is:

```
attribute_role attribute_id;
```

Where:

attribute_role

The *attribute_role* keyword.

attribute_id

The *attribute* identifier.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	Yes	Yes

Examples:

```
# Using the attribute_role statement to declare attributes that
# can then refers to a list of roles. Note that there are no
# roles associated with them yet.

attribute_role role_list_1;
attribute_role srole_list_2;
```

roleattribute

The *roleattribute* statement allows the association of previously declared roles to one or more previously declared *attribute_roles*.

The statement definition is:

```
roleattribute role_id attribute_id;
```

Where:

roleattribute

The *roleattribute* keyword.

role_id

The identifier of a previously declared *role*.

attribute_id

One or more previously declared *attribute_role* identifiers. Multiple entries consist of a comma ‘,’ separated list.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	Yes	No

Examples:

```
# Using the roleattribute statement to associate a previously
# declared role of service_r to a previously declared
```

```
# role_list_1 attribute_role.

attribute_role role_list_1;
role service_r;

# The association using the roleattribute statement:
roleattribute service_r role_list_1;
```

allow

The 'role *allow*' rule checks whether a request to change roles is allowed, if it is, then there may be a further request for a *role_transition* so that the process runs with the new role or role set.

Note that the 'role *allow*' rule has the same keyword as the *allow* AV rule.

The statement definition is:

```
allow from_role_id to_role_id;
```

Where:

allow

The role *allow* rule keyword.

from_role_id

One or more *role* or *attribute_role* identifiers that identify the current role. Multiple entries consist of a space separated list enclosed in braces '{}'.

to_role_id

One or more *role* or *attribute_role* identifiers that identify the target role. Multiple entries consist of a space separated list enclosed in braces '{}'.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Example:

```
# Using the role allow rule to define authorised role
# transitions in the Reference Policy. The current role
# sysadm_r is granted permission to transition to the secadm_r
# role in the MLS policy.
```

```
allow sysadm_r secadm_r;
```

role_transition

The *role_transition* rule specifies that a role transition is required, and if allowed, the process will run under the new role. From policy version 25, the *class* can now be defined.

The statement definition is:

```
role_transition current_role_id type_id new_role_id;
```

Or from Policy version 25:

```
role_transition current_role_id type_id : class new_role_id;
```

Where:

role_transition

The *role_transition* keyword.

current_role_id

One or more *role* or *attribute_role* identifiers that identify the current role. Multiple entries consist of a space separated list enclosed in braces '{}'.

type_id

One or more *type*, *typealias* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'.

class

For policy versions >= 25 an object *class* that applies to the role transition. If omitted defaults to the *process* object class.

new_role_id

A single *role* identifier that will become the new role.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Example:

```
role_transition system_r unconfined_exec_t:process unconfined_r;
```

dominance - Deprecated

This rule has been deprecated and therefore should not be used. The role dominance rule allows the *dom_role_id* to dominate the *role_id* (consisting of one or more roles). The dominant role will automatically inherit all the type associations of the other roles.

Notes:

1. There is another dominance rule for MLS (see the **MLS *dominance*** statement.
2. The role dominance rule is not used by the **Reference Policy** as the policy manages role dominance using the ***constrain*** statement.
3. Note the usage of braces '{}' and the ';' in the statement.

The statement definition is:

```
dominance { role dom_role_id { role role_id; } }
```

Where:

dominance

The *dominance* keyword.

role

The *role* keyword.

dom_role_id

The dominant role identifier.

role_id

For the simple case each *{ role role_id; }* pair defines the *role_id* that will be dominated by the *dom_role_id*.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Example:

```
# This shows the dominance role rule, note however that it
# has been deprecated and should not be used.
```

```
dominance { role message_filter_r { role unconfined_r };}
```

Type Statements

These statements share the same namespace, therefore the general convention is to use *_t* as the final two characters of a *type* identifier to differentiate it from an attribute identifier as shown in the following examples:

```
# Statement Identifier  Comment
#-------------------------------------------------------------------------
type bin_t;              # A type identifier generally ends with _t

attribute file_type;     # An attribute identifier generally ends with _type
```

type

The *type* statement declares the type identifier and any optional associated *alias* or *attribute* identifiers. Type identifiers are a component of the **Security Context**.

The statement definition is:

```
type type_id [alias alias_id] [, attribute_id];
```

Where:

type

The *type* keyword.

type_id

The *type* identifier.

alias

Optional *alias* keyword that signifies alternate identifiers for the *type_id* that are declared in the *alias_id* list. An alternative way to specify aliases is to use the *typealias* statement.

alias_id

One or more *alias* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'.

attribute_id

One or more optional *attribute* identifiers that have been previously declared by the *attribute* statement. Multiple entries consist of a comma ',' separated list, also note the lead comma.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	Yes	Yes

Examples:

```
# Using the type statement to declare a type of shell_exec_t,
# where exec_t is used to identify a file as an executable type.

type shell_exec_t;
```

```
# Using the type statement to declare a type of bin_t, where
# bin_t is used to identify a file as an ordinary program type.

type bin_t;
```

```
# Using the type statement to declare a type of bin_t with two
# alias names. The sbin_t is used to identify the file as a
# system admin program type.

type bin_t alias { ls_exec_t sbin_t };
```

```
# Using the type statement to declare a type of boolean_t that also
# associates it to a previously declared attribute booleans_type statement

attribute booleans_type; # declare the attribute

type boolean_t, booleans_type; # and associate with the type
```

```
# Using the type statement to declare a type of setfiles_t that
# also has an alias of restorecon_t and one previously declared
# attribute of can_relabelto_binary_policy associated with it.

attribute can_relabelto_binary_policy;

type setfiles_t alias restorecon_t, can_relabelto_binary_policy;
```

```
# Using the type statement to declare a type of
# ssh_server_packet_t that also associates it to two previously
```

```
# declared attributes packet_type and server_packet_type.

attribute packet_type; # declare attribute 1
attribute server_packet_type; # declare attribute 2

# Associate the type identifier with the two attributes:
type ssh_server_packet_t, packet_type, server_packet_type;
```

attribute

An *attribute* statement declares an identifier that can then be used to refer to a group of *type* identifiers.

The statement definition is:

```
attribute attribute_id;
```

Where:

attribute

The *attribute* keyword.

attribute_id

The *attribute* identifier.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	Yes

Examples:

```
# Using the attribute statement to declare attributes domain,
# daemon, file_type and non_security_file_type:

attribute domain;
attribute daemon;
attribute file_type;
attribute non_security_file_type;
```

expandattribute

Overrides the compiler defaults for the expansion of one or more previously declared *attribute* identifiers.

This rule gives more control over type attribute expansion and removal. When the value is true, all rules involving the type attribute will be expanded and the type attribute will be removed from the policy. When the value is false, the type attribute will not be removed from the policy, even if the default expand rules or “-X” option cause the rules involving the type attribute to be expanded.

The statement definition is:

```
expandattribute attribute_id expand_value;
```

Where:

expandattribute

The *expandattribute* keyword.

attribute_id

One or more *attribute* identifiers that have been previously declared by the *attribute* statement. Multiple entries consist of a space separated list enclosed in braces ‘{}’.

expand_value

Either true or false.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
Yes	Yes	No

Examples:

```
# Using the expandattribute statement to forcibly expand a
# previously declared domain attribute.

# The previously declared attribute:
attribute domain;

# The attribute stripping using the expandattribute statement:
expandattribute domain true;
```

typeattribute

The *typeattribute* statement allows the association of previously declared types to one or more previously declared attributes.

The statement definition is:

```
typeattribute type_id attribute_id;
```

Where:

typeattribute

The *typeattribute* keyword.

type_id

The identifier of a previously declared *type*.

attribute_id

One or more previously declared *attribute* identifiers. Multiple entries consist of a comma ',' separated list.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Examples:

```
# Using the typeattribute statement to associate a previously
# declared type of setroubleshootd_t to a previously declared
# domain attribute.

# The previously declared attribute:
attribute domain;

# The previously declared type:
type setroubleshootd_t;

# The association using the typeattribute statement:
typeattribute setroubleshootd_t domain;
```

```
# Using the typeattribute statement to associate a type of
# setroubleshootd_exec_t to two attributes file_type and
# non_security_file_type.
# These are the previously declared attributes:

attribute file_type;
attribute non_security_file_type;

# The previously declared type:
```

```
type setroubleshootd_exec_t;

# These are the associations using the typeattribute statement:
typeattribute setroubleshootd_exec_t file_type, non_security_file_type;
```

typealias

The *typealias* statement allows the association of a previously declared *type* to one or more *alias* identifiers (an alternative way is to use the *type* statement).

The statement definition is:

```
typealias type_id alias alias_id;
```

Where:

typealias

The *typealias* keyword.

type_id

The identifier of a previously declared *type*.

alias

The *alias* keyword.

alias_id

One or more *alias* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Examples:

```
# Using the typealias statement to associate the previously
# declared type mount_t with an alias of mount_ntfs_t.
# Declare the type:
type mount_t;

# Then alias the identifier:
typealias mount_t alias mount_ntfs_t;
```

```
# Using the typealias statement to associate the previously
# declared type netif_t with two alias, lo_netif_t and netif_lo_t.

# Declare the type:
type netif_t;

# Then assign two alias identifiers lo_netif_t and netif_lo_t:
typealias netif_t alias { lo_netif_t netif_lo_t };
```

permissive

Policy version 23 introduced the *permissive* statement to allow the named domain to run in permissive mode instead of running all SELinux domains in permissive mode (that was the only option prior to version 23). Note that the *permissive* statement only tests the source context for any policy denial.

The statement definition is:

```
permissive type_id;
```

Where:

permissive

The *permissive* keyword.

type_id

The *type* identifier of the domain that will be run in permissive mode.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Examples:

Set a permissive domain using the ***semanage****(8)* command :

```
# semanage supports enabling and disabling of permissive
# mode using the following command:

semanage permissive -a|d type

# This example will add a new module in:
#     /var/lib/selinux/<SELINUXTYPE>/active/modules/400/
```

```
# called permissive_unconfined_t and then reload the policy:

semanage permissive -a unconfined_t
```

Build a loadable policy module so that permissive mode can be easily enabled or disabled by adding or removing the module:

```
# This is an example loadable module that would allow the
# domain to be set to permissive mode.
#
module permissive_unconfined_t 1.0.0;

require {
    type unconfined_t;
}

permissive unconfined_t;
```

type_transition

The type_transition rule specifies the default type to be used for domain transition or object creation. Kernels from 2.6.39 with Policy versions from 25 also support the 'name transition rule' extension. See the **Computing Security Contexts** section for more details. Note than an *allow* rule must be used to authorise the transition.

The statement definitions are:

```
type_transition source_type target_type : class default_type;
```

Policy versions 25 and above also support a 'name transition' rule however, this is only appropriate for the file classes:

```
type_transition source_type target_type : class default_type object_name;
```

Kernel 5.12 introduced the 'name transition' rule for anonymous inodes that is described with an example in the **anon_inode Object Class** section.

Where:

type_transition

The *type_transition* rule keyword.

source_type target_type

One or more source / target *type*, *typealias* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'. Since libsepol 3.4 the *self* keyword is supported as a target.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

default_type

A single *type* or *typealias* identifier that will become the default process *type* for a domain transition or the *type* for object transitions.

object_name

For the 'name transition' rule this is matched against the objects name (i.e. the last component of a path). If *object_name* exactly matches the object name, then use *default_type* for the *type*.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
Yes (no name transitions)	Yes	No

Example - Domain Transition:

```
# Using the type_transition statement to show a domain
# transition (as the statement has the process object class).

# The rule states that when a process of type initrc_t executes
# a file of type acct_exec_t, the process type should be changed
# to acct_t if allowed by the policy (i.e. Transition from the
# initrc_t domain to the acc_t domain).

type_transition initrc_t acct_exec_t:process acct_t;

# Note that to be able to transition to the acc_t domain the
# following minimum permissions need to be granted in the policy
# using allow rules (as shown in the allow rule section).
# File needs to be executable in the initrc_t domain:
allow initrc_t acct_exec_t:file execute;

# The executable file needs an entry point into the acct_t domain:
allow acct_t acct_exec_t:file entrypoint;

# Process needs permission to transition into the acct_t domain:
allow initrc_t acct_t:process transition;
```

Example - Object Transition:

```
# Using the type_transition statement to show an object
# transition (as it has other than process in the class).
# The rule states that when a process of type acct_t creates a
# file in the directory of type var_log_t, by default it should
# have the type wtmp_t if allowed by the policy.

type_transition acct_t var_log_t:file wtmp_t;
```

```
# Note that to be able to create the new file object with the
# wtmp_t type, the following minimum permissions need to be
# granted in the policy using allow rules (as shown in the
# allow rule section).

# A minimum of: add_name, write and search on the var_log_t
# directory. The actual policy has:
#
allow acct_t var_log_t:dir { read getattr lock search ioctl add_name remove_name
write };

# A minimum of: create and write on the wtmp_t file. The actual policy has:
#
allow acct_t wtmp_t:file { create open getattr setattr read write append rename link
unlink ioctl lock };
```

Example - Name Transition:

```
# type_transition to allow using the last path component as
# part of the information in making labeling decisions for
# new objects. An example rule:
#

type_transition unconfined_t etc_t : file system_conf_t eric;

# This rule says if unconfined_t creates a file in a directory
# labeled etc_t and the last path component is "eric" (must be
# an exact strcmp) it should be labeled system_conf_t.
```

type_change

The *type_change* rule specifies a default *type* when relabeling an existing object. For example userspace SELinux-aware applications would use ***security_compute_relabel****(3)* and *type_change* rules in policy to determine the new context to be applied. Note that an *allow* rule must be used to authorise access. See the **Computing Security Contexts** section for more details.

The statement definition is:

```
type_change source_type target_type : class change_type;
```

Where:

type_change

The *type_change* rule keyword.

source_type target_type

One or more source / target *type*, *typealias* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'. Since libsepol 3.4 the *self* keyword is supported as a target.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

change_type

A single *type* or *typealias* identifier that will become the new *type*.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
Yes	Yes	No

Examples:

```
# Using the type_change statement to show that when relabeling a
# character file with type sysadm_devpts_t on behalf of
# auditadm_t, the type auditadm_devpts_t should be used:

type_change auditadm_t sysadm_devpts_t:chr_file auditadm_devpts_t;
```

```
# Using the type_change statement to show that when relabeling a
# character file with any type associated to the attribute
# server_ptynode on behalf of staff_t, the type staff_devpts_t
# should be used:

type_change staff_t server_ptynode:chr_file staff_devpts_t;
```

type_member

The *type_member* rule specifies a default type when creating a polyinstantiated object. For example a userspace SELinux-aware application would use ***avc_compute_member**(3)* or ***security_compute_member**(3)* with *type_member* rules in policy to determine the context to be applied. Note that an *allow* rule must be used to authorise access. See the **Computing Security Contexts** section for more details.

The statement definition is:

```
type_member source_type target_type : class member_type;
```

Where:

type_member

The *type_member* rule keyword.

source_type target_type

One or more source / target *type*, *typealias* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'. Since libsepol 3.4 the *self* keyword is supported as a target.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

member_type

A single *type* or *typealias* identifier that will become the polyinstantiated *type*.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
Yes	Yes	No

Example:

```
# Using the type_member statement to show that if the source
# type is sysadm_t, and the target type is user_home_dir_t,
# then use user_home_dir_t as the type on the newly created
# directory object.

type_member sysadm_t user_home_dir_t:dir user_home_dir_t;
```

Bounds Rules

- *typebounds*

Bounds handling was added in version 24 of the policy and consisted of adding *userbounds*, *rolebounds* and *typebounds* information to the policy. However only the *typebounds* rule is currently implemented by ***checkpolicy****(8)* and ***checkmodule****(8)* with kernel support from 2.6.28.

The CIL language does support *userbounds* and *rolebounds* but these are resolved at policy compile time, not via the kernel at run-time (i.e. they are NOT enforced by the SELinux kernel services). The **CIL Reference Guide** gives details.

typebounds

The *typebounds* rule was added in version 24 of the policy. This defines a hierarchical relationship between domains where the bounded domain cannot have more permissions than its bounding domain (the parent). It requires kernel 2.6.28 and above to control the security context associated to threads in multi-threaded applications.

The statement definition is:

```
typebounds bounding_domain bounded_domain;
```

Where:

typebound

The *typebounds* keyword.

bounding_domain

The *type* or *typealias* identifier of the parent domain.

bounded_domain

One or more *type* or *typealias* identifiers of the child domains. Multiple entries consist of a comma ‘,’ separated list.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Example:

```
# This example states that:
#   The httpd_child_t cannot have file:{write} due to lack of permissions
#   on httpd_t which is the parent. It means the child domains will always
#   have equal or less privileges than the parent.

# The typebounds statement:
typebounds httpd_t httpd_child_t;

# The parent is allowed file 'getattr' and 'read':
allow httpd_t etc_t : file { getattr read };

# However the child process has been given 'write' access that
# will not be allowed by the kernel SELinux security server.
allow httpd_child_t etc_t : file { read write };
```

Access Vector Rules

The AV rules define what access control privileges are allowed for processes and objects. There are four types of AV rule: *allow*, *dontaudit*, *auditallow*, and *neverallow* as explained in the sections that follow with a number of examples to cover all the scenarios.

The general format of an AV rule is that the *source_type* is the identifier of a process that is attempting to access an object identifier *target_type*, that has an object class of *class*, and *perm_set* defines the access permissions *source_type* is allowed.

From Policy version 30 with the target platform '*selinux*', the AVC rules have been extended to expand the permission sets from a fixed 32 bits to permission sets in 256 bit increments. The format of the new *allowxperm*, *dontauditxperm*, *auditallowxperm* and *neverallowxperm* rules are discussed in the **Extended Access Vector Rules** section.

The common format for Access Vector Rules are:

```
rule_name source_type target_type : class perm_set;
```

Where:

rule_name

The applicable *allow*, *dontaudit*, *auditallow*, and *neverallow* rule keyword.

source_type, *target_type*

One or more source / target *type*, *typealias* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'. The *target_type* can have the self keyword instead of *type*, *typealias* or *attribute* identifiers. This means that the *target_type* is the same as the *source_type*. The *neverallow* rule also supports the wildcard operator '*' to specify that all types are to be included and the complement operator '~' to specify all types are to be included except those explicitly listed.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

perm_set

The access permissions the source is allowed to access for the target object (also known as the Access Vector). Multiple entries consist of a space separated list enclosed in braces '{}'. The optional wildcard operator '*' specifies that all permissions for the object *class* can be used. The complement operator '~' is used to specify all permissions except those explicitly listed (although the compiler issues a warning if the *dontaudit* rule has '~'.

The statements are valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
Yes: *allow*, *dontaudit*, *auditallow* No: *neverallow*	Yes	No

allow

The allow rule checks whether the operations between the *source_type* and *target_type* are allowed for the class and permissions defined. It is the most common statement that many of the **Reference Policy** helper macros and interface definitions expand into multiple allow rules.

Examples:

```
# Using the allow rule to show that initrc_t is allowed access
# to files of type acct_exec_t that have the getattr, read and
# execute file permissions:

allow initrc_t acct_exec_t:file { getattr read execute };
```

```
# This rule includes an attribute filesystem_type and states
# that kernel_t is allowed mount permissions on the filesystem
# object for all types associated to the filesystem_type attribute:

allow kernel_t filesystem_type:filesystem mount;
```

```
# This rule includes the self keyword in the target_type that
# states that staff_t is allowed setgid, chown and fowner
# permissions on the capability object:

allow staff_t self:capability { setgid chown fowner };

# This would be the same as the above:
allow staff_t staff_t:capability { setgid chown fowner };
```

```
# This rule includes the wildcard operator (*) on the perm_set
# and states that bootloader_t is allowed to use all permissions
# available on the dbus object that are type system_dbusd_t:
allow bootloader_t system_dbusd_t:dbus *;

# This would be the same as the above:
allow bootloader_t system_dbusd_t:dbus { acquire_svc send_msg };
```

```
# This rule includes the complement operator (~) on the perm_set
# and two class entries file and chr_file.
#
# The allow rule states that all types associated with the
# attribute files_unconfined_type are allowed to use all
# permissions available on the file and chr_file objects except
# the execmod permission when they are associated to the types
# listed within the attribute file_type:

allow files_unconfined_type file_type:{ file chr_file } ~execmod;
```

dontaudit

The *dontaudit* rule stops the auditing of denial messages as it is known that this event always happens and does not cause any real issues. This also helps to manage the audit log by excluding known events.

Example:

```
# Using the dontaudit rule to stop auditing events that are
# known to happen. The rule states that when the traceroute_t
# process is denied access to the name_bind permission on a
# tcp_socket for all types associated to the port_type
# attribute (except port_t), then do not audit the event:

dontaudit traceroute_t { port_type -port_t }:tcp_socket name_bind;
```

auditallow

Audit the event as a record as it is useful for auditing purposes. Note that this rule only audits the event, it still requires the *allow* rule to grant permission.

Example:

```
# Using the auditallow rule to force an audit event to be
# logged. The rule states that when the ada_t process has
# permission to execstack, then that event must be audited:

auditallow ada_t self:process execstack;
```

neverallow

This rule specifies that an *allow* rule must not be generated for the operation, even if it has been previously allowed. The *neverallow* statement is a compiler enforced action, where the ***checkpolicy****(8)*, ***checkmodule****(8)*[19] or ***secilc****(8)*[20] compiler checks if any allow rules have been generated in the policy source, if so it will issue a warning and stop.

Examples:

```
# Using the neverallow rule to state that no allow rule may ever
# grant any file read access to type shadow_t except those
```

```
# associated with the can_read_shadow_passwords attribute:

neverallow ~can_read_shadow_passwords shadow_t:file read;
```

```
# Using the neverallow rule to state that no allow rule may ever
# grant mmap_zero permissions any type associated to the domain
# attribute except those associated to the mmap_low_domain_type
# attribute (as these have been excluded by the negative operator '-'):

neverallow { domain -mmap_low_domain_type } self:memprotect mmap_zero;
```

Extended Access Vector Rules

There are four extended AV rules implemented from Policy version 30 with the target platform ‘selinux’ that expand the permission sets from a fixed 32 bits to permission sets in 256 bit increments: *allowxperm*, *dontauditxperm*, *auditallowxperm* and *neverallowxperm*.

The rules for extended permissions are subject to the ‘operation’ they perform with Policy version 30 and kernels from 4.3 supporting ioctl allowlists (if required to be declared in modular policy, then libsepol 2.7 minimum is required).

The common format for Extended Access Vector Rules are:

```
rule_name source_type target_type : class operation xperm_set;
```

Where:

rule_name

The applicable *allowxperm*, *dontauditxperm*, *auditallowxperm* or *neverallowxperm* rule keyword.

source_type

One or more source / target *type*, *typealias* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces '{}'. Entries can be excluded from the list by using the negative operator '-'.

target_type

The target_type can have the *self* keyword instead of *type*, *typealias* or *attribute* identifiers. This means that the *target_type* is the same as the *source_type*.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

operation

A key word defining the operation to be implemented by the rule. Currently only the *ioctl* operation is supported by the kernel policy language and kernel as described in the *ioctl* Operation Rules section.

xperm_set

One or more extended permissions represented by numeric values (i.e. *0x8900* or *35072*). The usage is dependent on the specified *operation*. Multiple entries consist of a space separated list enclosed in braces '{}'. The complement operator '~' is used to specify all permissions except those explicitly listed. The range operator '-' is used to specify all permissions within the *low* – *high* range. An example is shown in the *ioctl* Operation Rules section.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	No

ioctl Operation Rules

Use cases and implementation details for ioctl command allowlists are described in detail at http://marc.info/?l=selinux&m=143336061925628&w=2, with the final policy format changes shown in the example below with a brief overview (also see http://marc.info/?l=selinux&m=143412575302369&w=2) that is the final upstream kernel patch).

Ioctl calls are generally used to get or set device options. Policy versions > 30 only controls whether an *ioctl* permission is allowed or not, for example this rule allows the object class *tcp_socket* the *ioctl* permission:

```
allow src_t tgt_t : tcp_socket ioctl;
```

From Policy version 30 it is possible to control ***ioctl****(2) 'request'* parameters provided the *ioctl* permission is also allowed, for example:

```
allow src_t tgt_t : tcp_socket ioctl;

allowxperm src_t tgt_t : tcp_socket ioctl ~0x8927;
```

The *allowxperm* rule states that all ioctl request parameters are allowed for the source/target/class with the exception of the value *0x8927* that (using *include/linux/sockios.h*) is **SIOCGIFHWADDR**, or 'get hardware address'.

An example audit log entry denying an ioctl request to add a routing table entry (**SIOCADDRT** - *ioctlcmd=890b*) for *goldfish_setup* on a *udp_socket* is:

```
type=1400 audit(1437408413.860:6): avc: denied { ioctl } for pid=81
comm="route" path="socket:[1954]" dev="sockfs" ino=1954 ioctlcmd=890b
scontext=u:r:goldfish_setup:s0 tcontext=u:r:goldfish_setup:s0
tclass=udp_socket permissive=0
```

Notes:

1. Important: The ioctl operation is not 'deny all', it is targeted at the specific source/target/class set of ioctl commands. As no other *allowxperm* rules have been defined in the example, all other ioctl calls may continue to use any valid request parameters (provided there are *allow* rules for the *ioctl* permission).
2. As the ***ioctl****(2)* function requires a file descriptor, its context must match the process context otherwise the *fd { use }* class/permission is required.
3. To deny all ioctl requests for a specific source/target/class the *xperm_set* should be set to *0* or *0x0*.
4. From the 32-bit ioctl request parameter value only the least significant 16 bits are used. Thus *0x8927*, *0x00008927* and *0xabcd8927* are the same extended permission.
5. To decode a numeric ioctl request parameter into the corresponding textual identifier see https://www.kernel.org/doc/html/latest/userspace-api/ioctl/ioctl-decoding.html

Object Class and Permission Statements

For those who write or manager SELinux policy, there is no need to define new objects and their associated permissions as these would be done by those who actually design and/or write object managers.

A list of object classes used by the **Reference Policy** can be found in the *./policy/flask/security_classes* file.

There are two variants of the *class* statement for writing policy:

1. There is the *class* statement that declares the actual class identifier or name.
2. There is a further refinement of the *class* statement that associates permissions to the class as discussed in the **Associating Permissions to a Class** section.

class (1)

Object classes are declared within a policy with the following statement definition:

```
class class_id
```

Where:

class

The *class* keyword.

class_id

The *class* identifier.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	Yes

Example:

```
# Define the PostgreSQL db_tuple object class
#
class db_tuple
```

Associating Permissions to a Class

Permissions can be defined within policy in two ways:

1. Define a set of common permissions that can then be inherited by one or more object classes using further *class* statements.
2. Define *class* specific permissions. This is where permissions are declared for a specific object class only (i.e. the permission is not inherited by any other object class).

A list of classes and their permissions used by the **Reference Policy** can be found in the *./policy/flask/access_vectors* file.

common

Declare a *common* identifier and associate one or more *common* permissions.

The statement definition is:

```
common common_id { perm_set }
```

Where:

common

The *common* keyword.

common_id

The *common* identifier.

perm_set

One or more permission identifiers in a space separated list enclosed within braces '{}'.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# Define the common PostgreSQL permissions

common database { create drop getattr setattr relabelfrom relabelto }
```

class (2)

Inherit and / or associate permissions to a previously declared *class* identifier.

The statement definition is:

```
class class_id [ inherits common_set ] [ { perm_set } ]
```

Where:

class

The *class* keyword.

class_id

The previously declared *class* identifier.

inherits

The optional *inherits* keyword that allows a set of common permissions to be inherited.

common_set

A previously declared *common* identifier.

perm_set

One or more optional permission identifiers in a space separated list enclosed within braces '{}'.

Note: There must be at least one *common_set* or one *perm_set* defined within the statement.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	Yes

Examples:

```
# The following example shows the db_tuple object class being
# allocated two permissions:

class db_tuple { relabelfrom relabelto }
```

```
# The following example shows the db_blob object class inheriting
# permissions from the database set of common permissions (as described
# in the Associating Permissions to a Class section):
```

```
class db_blob inherits database
```

```
# The following example (from the access_vector file) shows the
# db_blob object class inheriting permissions from the database
# set of common permissions and adding a further four permissions:

class db_blob inherits database { read write import export }
```

Conditional Policy Statements

- *if*
- *bool*

Conditional policies consist of a bool statement that defines a condition as *true* or *false*, with a supporting *if* / *else* construct that specifies what rules are valid under the condition as shown in the example below:

```
bool allow_daemons_use_tty true;

if (allow_daemons_use_tty) {
    # Rules if condition is true;
} else {
    # Rules if condition is false;
}
```

Table 3 in the 'Kernel Policy Language' section shows what policy statements or rules are valid within the *if* / *else* construct under the "Conditional Statements" column.

The *bool* statement default value can be changed when a policy is active by using the ***setsebool**(3)* command as follows:

```
# This command will set the allow_daemons_use_tty bool to false,
# however it will only remain false until the next system
# re-boot where it will then revert back to its default state
# (in the above case, this would be true).

setsebool allow_daemons_use_tty false
```

```
# This command will set the allow_daemons_use_tty bool to false,
# and because the -P option is used (for persistent), the value
# will remain across system re-boots. Note however that all
# other pending bool values will become persistent across
# re-boots as well (see setsebool(8) man page).

setsebool -P allow_daemons_use_tty false
```

The ***getsebool**(3)* command can be used to query the current *bool* statement value as follows:

```
# This command will list all bool values in the active policy:
getsebool -a
```

```
# This command will show the current allow_daemons_use_tty bool
# value in the active policy:

getsebool allow_daemons_use_tty
```

bool

The *bool* statement is used to specify a boolean identifier and its initial state (*true* or *false*) that can then be used with the *if* statement to form a 'conditional policy' as described in the Types of SELinux Policy section.

The statement definition is:

```
bool bool_id default_value;
```

Where:

bool

The *bool* keyword.

bool_id

The boolean identifier.

default_value

Either true or false.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	Yes

Examples:

```
# Using the bool statement to allow unconfined executables to
# make their memory heap executable or not. As the value is
# false, then by default they cannot make their heap executable.

bool allow_execheap false;
```

```
# Using the bool statement to allow unconfined executables to
# make their stack executable or not. As the value is true,
# then by default their stacks are executable.

bool allow_execstack true;
```

if

The if statement is used to form a 'conditional block' of statements and rules that are enforced depending on whether one or more boolean identifiers evaluate to *TRUE* or *FALSE*. An *if* / *else* construct is also supported.

The only statements and rules allowed within the *if* / *else* construct are:

allow, *auditallow*, *auditdeny*, *dontaudit*, *type_member*, *type_transition* (except *file_name_transition*), *type_change* and *require*.

The statement definition is:

```
if (conditional_expression) { true_list } [ else { false_list } ]
```

Where:

if

The *if* keyword.

conditional_expression

One or more *bool_name* identifiers that have been previously defined by the *bool* Statement. Multiple identifiers must be separated by the following logical operators: &&, ¦ ¦, ^, !, ==, !=. The *conditional_expression* is enclosed in brackets '()'.

true_list

A list of rules enclosed within braces '{}' that will be executed when the *conditional_expression* is 'true'. Valid statements and rules are highlighted within each language definition statement.

else

Optional *else* keyword.

false_list

A list of rules enclosed within braces '{}' that will be executed when the optional *else* keyword is present and the *conditional_expression* is *false*. Valid statements and rules are highlighted within each language definition statement.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Examples:

```
# An example showing a boolean and supporting if statement.
bool allow_execmem false;
```

```
# The bool allow_execmem is FALSE therefore the allow statement is not executed:

if (allow_execmem) {
    allow sysadm_t self:process execmem;

}
```

```
# An example showing two booleans and a supporting if statement.

bool allow_execmem false;
bool allow_execstack true;

# The bool allow_execmem is FALSE and allow_execstack is TRUE
# therefore the allow statement is not executed:

if (allow_execmem && allow_execstack) {
    allow sysadm_t self:process execstack;

}
```

```
# An example of an IF - ELSE statement where the bool statement
# is FALSE, therefore the ELSE statements will be executed.
#

bool read_untrusted_content false;

if (read_untrusted_content) {
    allow sysadm_t { sysadm_untrusted_content_t sysadm_untrusted_content_tmp_t }:dir
{ getattr search read lock ioctl };
.....

} else {
    dontaudit sysadm_t { sysadm_untrusted_content_t
    sysadm_untrusted_content_tmp_t }:dir { getattr search read lock ioctl };
    ...
}
```

Constraint Statements

constrain

The *constrain* statement allows further restriction on permissions for the specified object classes by using boolean expressions covering: source and target types, roles and users as described in the examples.

The statement definition is:

```
constrain class perm_set expression | expr ...;
```

Where:

constrain

The *constrain* keyword.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

perm_set

One or more permissions. Multiple entries consist of a space separated list enclosed in braces '{}'. Since ***checkpolicy****(8)* 3.4 the wildcard operator, '*', can be used to specify that all permissions are to be included, and the complement operator, '~', can be used to specify all permissions are to be included except those explicitly listed.

expression

There must be one constraint *expression* or one or more *expr*'s. An *expression* consists of '*operand operator operand*' as follows:

- *(u1 op u2)*
- *(r1 role_op r2)*
- *(t1 op t2)*
- *(u1 op names)*
- *(u2 op names)*
- *(r1 op names)*
- *(r2 op names)*
- *(t1 op names)*
- *(t2 op names)*
- Where:
  - *u1*, *r1*, *t1* = Source *user*, *role*, *type*
  - *u2*, *r2*, *t2* = Target *user*, *role*, *type*
- And:
  - *op :* == | *!=*
  - *role_op :* == | *!=* | *eq* | *dom* | *domby* | *incomp*
  - *names : name* | *{ name_list }*
  - *name_list : name* | *name_list name*

expr

Zero or more *expr*'s, the valid operators and syntax are:

- *(not expression)*
- *(expression and expression)*
- *(expression or expression)*

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

These examples have been taken from the **Reference Policy** source *./policy/constraints* file.

```
# This constrain statement is the "SELinux process identity
# change constraint" taken from the Reference Policy source and
# contains multiple expressions.
#
# The overall constraint is on the process object class with the
# transition permission, and is stating that a domain transition
# is being constrained by the rules listed (u1 == u2 etc.),
# however only the first two expressions are explained.
#
# The first expression u1 == u2 states that the source (u1) and
# target (u2) user identifiers must be equal for a process
# transition to be allowed.
#
# However note that there are a number of or operators that can
# override this first constraint.
#
# The second expression:
#    ( t1 == can_change_process_identity and t2 == process_user_target )
# states that if the source type (t1) is equal to any type
# associated to the can_change_process_identity attribute, and
# the target type (t2) is equal to any type associated to the
# process_user_target attribute, then a process transition is allowed.
# What this expression means in the 'standard' build Reference
# Policy is that if the source domain is either cron_t,
# firstboot_t, local_login_t, su_login_t, sshd_t or xdm_t (as
# the can_change_process_identity attribute has these types
# associated to it) and the target domain is sysadm_t (as that
# is the only type associated to the can_change_process_identity
# attribute), then a domain transition is allowed.
#
```

```
# SELinux process identity change constraint:
constrain process transition (
    u1 == u2
    or
    ( t1 == can_change_process_identity and t2 == process_user_target )
    or
    ( t1 == cron_source_domain and ( t2 == cron_job_domain or u2 == system_u ))
    or
    ( t1 == can_system_change and u2 == system_u )
    or
    ( t1 == process_uncond_exempt ) );
```

```
# This constrain statement is the "SELinux file related object
# identity change constraint" taken from the Reference Policy
# source and contains two expressions.
#
# The overall constraint is on the listed file related object
# classes (dir, file etc.), covering the create, relabelto, and
# relabelfrom permissions. It is stating that when any of the
# object class listed are being created or relabeled, then they
# are subject to the constraint rules listed (u1 == u2 etc.).
#
# The first expression u1 == u2 states that the source (u1) and
# target (u2) user identifiers (within the security context)
# must be equal when creating or relabeling any of the file
# related objects listed.
#
# The second expression:
#    or t1 == can_change_object_identity
#
# states or if the source type (t1) is equal to any type
# associated to the can_change_object_identity attribute, then
# any of the object class listed can be created or relabeled.
#
# What this expression means in the 'standard' build
# Reference Policy is that if the source domain (t1) matches a
# type entry in the can_change_object_identity attribute, then
# any of the object class listed can be created or relabeled.
#
# SELinux file related object identity change constraint:

constrain { dir file lnk_file sock_file fifo_file chr_file blk_file } { create
relabelto relabelfrom }
    (u1 == u2 or t1 == can_change_object_identity);
```

validatetrans

The *validatetrans* statement is used to control the ability to change the objects security context.

The first context *u1:r1:t1* is the context before the transition, the second context *u2:r2:t2* is the context after the transition, and the third *u3:r3:t3* is the context of the process performing the transition.

Note there are no *validatetrans* statements specified within the **Reference Policy** source.

The statement definition is:

```
validatetrans class  expression | expr ...;
```

Where:

validatetrans

The *validatetrans* keyword.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

expression

There must be one constraint *expression* or one or more *expr*'s. An *expression* consists of '*operand operator operand*' as follows:

- *(u1 op u2)*
- *(r1 role_op r2)*
- *(t1 op t2)*
- *(u1 op names)*
- *(u2 op names)*
- *(u3 op names)*
- *(r1 op names)*
- *(r2 op names)*
- *(r3 op names)*
- *(t1 op names)*
- *(t2 op names)*
- *(t3 op names)*
- Where:
  - *u1*, *r1*, *t1* = Source *user*, *role*, *type*
  - *u2*, *r2*, *t2* = Target *user*, *role*, *type*
  - *u3*, *r3*, *t3* = Process *user*, *role*, *type*
- And:
  - *op :* == | *!=*
  - *role_op :* == | *!=* | *eq* | *dom* | *domby* | *incomp*
  - *names : name* | *{ name_list }*
  - *name_list : name* | *name_list name*

expr

Zero or more *expr*'s, the valid operators and syntax are:

- *(not expression)*
- *(expression and expression)*
- *(expression or expression)*

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
validatetrans { file } ( t1 == unconfined_t );
```

mlsconstrain

The *mlsconstrain* statement allows further restriction on permissions for the specified object classes by using boolean expressions covering: source and target types, roles, users and security levels as described in the examples.

The statement definition is:

```
mlsconstrain class perm_set expression | expr ...;
```

Where:

mlsconstrain

The *mlsconstrain* keyword.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

perm_set

One or more permissions. Multiple entries consist of a space separated list enclosed in braces '{}'. Since ***checkpolicy****(8)* 3.4 the wildcard operator, '*', can be used to specify that all permissions are to be included, and the complement operator, '~', can be used to specify all permissions are to be included except those explicitly listed.

expression

There must be one constraint *expression* or one or more *expr*'s. An *expression* consists of '*operand operator operand*' as follows:

- *(u1 op u2)*
- *(r1 role_mls_op r2)*
- *(t1 op t2)*
- *(l1 role_mls_op l2)*
- *(l1 role_mls_op h2)*
- *(h1 role_mls_op l2)*
- *(h1 role_mls_op h2)*
- *(l1 role_mls_op h1)*
- *(l2 role_mls_op h2)*
- *(u1 op names)*
- *(u2 op names)*
- *(r1 op names)*
- *(r2 op names)*
- *(t1 op names)*

- *(t2 op names)*
- Where:
  - *u1*, *r1*, *t1*, *l1*, *h1* = Source *user*, *role*, *type*, *low*, *high*
  - *u2*, *r2*, *t2*, *l2*, *h2* = Target *user*, *role*, *type*, *low*, *high*
- And:
  - *op : == | !=*
  - *role_mls_op : == | != | eq | dom | domby | incomp*
  - *names : name | { name_list }*
  - *name_list : name | name_list name*

expr

Zero or more *expr*'s, the valid operators and syntax are:

- *(not expression)*
- *(expression and expression)*
- *(expression or expression)*

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

This example has been taken from the **Reference Policy** source **./policy/mls* constraints file. These are built into the policy at build time and add constraints to many of the object classes.

```
# The MLS Reference Policy mlsconstrain statement for searching
# directories that comprises of multiple expressions. Only the
# first two expressions are explained.
#
# Expression 1 ( l1 dom l2 ) reads as follows:
#   The dir object class search permission is allowed if the source low
#   security level is dominated by the targets low security level.
# OR
# Expression 2 (( t1 == mlsfilereadtoclr ) and ( h1 dom l2 ))
# reads as follows:
#   If the source type is equal to a type associated to the
#   mlsfilereadtoclr attribute and the source high security
#   level is dominated by the targets low security level,
#   then search permission is allowed on the dir object class.

mlsconstrain dir search
    (( l1 dom l2 ) or
    (( t1 == mlsfilereadtoclr ) and ( h1 dom l2 )) or
```

```
    ( t1 == mlsfileread ) or
    ( t2 == mlstrustedobject ));
```

mlsvalidatetrans

The *mlsvalidatetrans* is the MLS equivalent of the *validatetrans* statement where it is used to control the ability to change the objects security context.

The first context *u1:r1:t1:l1-h1* is the context before the transition, the second context *u2:r2:t2:l2-h2* is the context after the transition, and the third *u3:r3:t3:*[*range*] is the context of the process performing the transition.

The statement definition is:

```
mlsvalidatetrans class expression | expr ...;
```

Where:

mlsvalidatetrans

The *mlsvalidatetrans* keyword.

class

One or more object classes. Multiple entries consist of a space separated list enclosed in braces '{}'.

expression

There must be one constraint *expression* or one or more *expr*'s. An *expression* consists of '*operand operator operand*' as follows:

- *(u1 op u2)*
- *(r1 role_mls_op r2)*
- *(t1 op t2)*
- *(l1 role_mls_op l2)*
- *(l1 role_mls_op h2)*
- *(h1 role_mls_op l2)*
- *(h1 role_mls_op h2)*
- *(l1 role_mls_op h1)*
- *(l2 role_mls_op h2)*
- *(u1 op names)*
- *(u2 op names)*
- *(u3 op names)*
- *(r1 op names)*
- *(r2 op names)*
- *(r3 op names)*
- *(t1 op names)*
- *(t2 op names)*
- *(t3 op names)*
- Where:
  - *u1*, *r1*, *t1*, *l1*, *h1* = Source *user*, *role*, *type*, *low*, *high*
  - *u2*, *r2*, *t2*, *l2*, *h2* = Target *user*, *role*, *type*, *low*, *high*
  - *u3*, *r3*, *t3*, [*range*] = Process *user*, *role*, *type*, [*range*]
- And:
  - *op :* == | *!=*
  - *role_mls_op :* == | *!=* | *eq* | *dom* | *domby* | *incomp*
  - *names : name* | *{ name_list }*

- *name_list : name | name_list name*

expr

Zero or more *expr*'s, the valid operators and syntax are:

- *(not expression)*
- *(expression and expression)*
- *(expression or expression)*

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

This example has been taken from the **Reference Policy** source *./policy/mls* file.

```
# The MLS Reference Policy mlsvalidatetrans statement for
# managing the file upgrade/downgrade rules that comprises of
# multiple expressions. Only the first two expressions are explained.
#
# Expression 1: ( l1 eq l2 ) reads as follows:
#   For a file related object to change security context, its
#   current (old) low security level must be equal to the new
#   objects low security level.
#
# The second part of the expression:
#   or (( t3 == mlsfileupgrade ) and ( l1 domby l2 )) reads as follows:
#     or the process type must equal a type associated to the
#     mlsfileupgrade attribute and its current (old) low security
#     level must be dominated by the new objects low security level.
#
mlsvalidatetrans { dir file lnk_file chr_file blk_file sock_file fifo_file }
    ((( l1 eq l2 ) or
    (( t3 == mlsfileupgrade ) and ( l1 domby l2 )) or
    (( t3 == mlsfiledowngrade ) and ( l1 dom l2 )) or
    (( t3 == mlsfiledowngrade ) and ( l1 incomp l2 ))) and (( h1 eq h2 ) or
    (( t3 == mlsfileupgrade ) and ( h1 domby h2 )) or
    (( t3 == mlsfiledowngrade ) and ( h1 dom h2 )) or
    (( t3 == mlsfiledowngrade ) and ( h1 incomp h2 ))));
```

MLS Statements

The optional MLS policy extension adds an additional security context component that consists of the following highlighted entries:

user:role:type: ***sensitivity[:category,...] - sensitivity [:category,...]***

These consist of a mandatory hierarchical **sensitivity** and optional non-hierarchical **category**'s. The combination of the two comprise a **level** or security level. Depending on the circumstances, there can be one level or a **range**.

To make the security levels more meaningful, it is possible to use the ***mcstransd****(8)* daemon to translate these to human readable formats. The ***semanage****(8)* command will allow this mapping to be defined as discussed in the **setrans.conf** section.

MLS range Definition

The MLS range is appended to a number of statements and defines the lowest and highest security levels. The range can also consist of a single level as discussed at the start of the **MLS section**.

The definition is:

```
low_level [ - high_level ]
```

Where:

low_level

The processes lowest level identifier that has been previously declared by a *level* statement. If a *high_level* is not defined, then it is taken as the same as the *low_level*.

-

The optional hyphen '-' separator if a *high_level* is also being defined.

high_level

The processes highest level identifier that has been previously declared by a *level* statement.

sensitivity

The sensitivity statement defines the MLS policy sensitivity identifies and optional alias identifiers.

The statement definition is:

```
sensitivity sens_id [alias sensitivityalias_id ...];
```

Where:

sensitivity

The *sensitivity* keyword.

sens_id

The *sensitivity* identifier.

alias

The optional *alias* keyword.

sensitivityalias_id

One or more sensitivity alias identifiers in a space separated list.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	Yes

Examples:

```
# The MLS Reference Policy default is to assign 16 sensitivity
# identifiers (s0 to s15):

sensitivity s0;
....
sensitivity s15;
```

```
# The policy does not specify any alias entries, however a valid
# example would be:

sensitivity s0 alias secret wellmaybe ornot;
```

dominance

When more than one *sensitivity* statemement is defined within a policy, then a *dominance* statement is required to define the actual hierarchy between all sensitivities.

The statement definition is:

```
dominance { sensitivity_id ... }
```

Where:

dominance

The *dominance* keyword.

sensitivity_id

A space separated list of previously declared *sensitivity* or *sensitivityalias* identifiers in the order lowest to highest. They are enclosed in braces '{}', and note that there is no terminating semi-colon ';'.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# The MLS Reference Policy dominance statement defines s0 as the
# lowest and s15 as the highest sensitivity level:

dominance { s0 s1 s2 s3 s4 s5 s6 s7 s8 s9 s10 s11 s12 s13 s14 s15 }
```

category

The *category* statement defines the MLS policy category identifiers and optional alias identifiers.

The statement definition is:

```
category category_id [alias categoryalias_id ...];
```

Where:

category

The *category* keyword.

category_id

The *category* identifier.

alias

The optional *alias* keyword.

categoryalias_id

One or more *alias* identifiers in a space separated list.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	Yes

Examples:

```
# The MLS Reference Policy default is to assign 256 category
# identifiers (c0 to c255):

category c0;
...
category c255;
```

```
# The policy does not specify any alias entries, however a valid
# example would be:

category c0 alias planning development benefits;
```

level

The *level* statement enables the previously declared sensitivity and category identifiers to be combined into a Security Level.

Note there must only be one *level* statement for each *sensitivity* statemement.

The statement definition is:

```
level sensitivity_id [ :category_id ];
```

Where:

level

The *level* keyword.

sensitivity_id

A previously declared *sensitivity* or *sensitivityalias* identifier.

category_id

An optional set of zero or more previously declared *category* or *categoryalias* identifiers that are preceded by a colon ':', that can be written as follows:

- The period '.' separating two *category* identifiers means an inclusive set (e.g. *c0.c16*).
- The comma ',' separating two *category* identifiers means a non-contiguous list (e.g. *c21,c36,c45*).

Both separators may be used (e.g. *c0.c16,c21,c36,c45*).

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# The MLS Reference Policy default is to assign each Security Level with
# the complete set of categories (i.e. the inclusive set from c0 to c255):

level s0:c0.c255;
...
level s15:c0.c255;
```

range_transition

The *range_transition* statement is primarily used by the init process or administration commands to ensure processes run with their correct MLS range (for example *init* would run at **SystemHigh** and needs to initialise / run other processes at their correct MLS range). The statement was enhanced in Policy version 21 to accept other object classes.

The statement definition is (for pre-policy version 21):

```
range_transition source_type target_type new_range;
```

or (for policy version 21 and greater):

```
range_transition source_type target_type : class new_range;
```

Where:

range_transition

The *range_transition* keyword.

source_type, *target_type*

One or more source / target *type* or *attribute* identifiers. Multiple entries consist of a space separated list enclosed in braces'{}'. Entries can be excluded from the list by using the negative operator'-'.

class

The optional object *class* keyword (this allows policy versions 21 and greater to specify a class other than the default of *process*).

new_range

The new MLS range for the object class. The format of this field is described in the MLS range Definition section.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	Yes	No

Examples:

```
# A range_transition statement from the MLS Reference Policy
# showing that a process anaconda_t can transition between
# systemLow and systemHigh depending on calling applications level.

range_transition anaconda_t init_script_file_type:process s0-s15:c0.c255;
```

```
# Two range_transition statements from the MLS Reference Policy
# showing that init will transition the audit and cups daemon
# to systemHigh (that is the lowest level they can run at).

range_transition initrc_t auditd_exec_t:process s15:c0.c255;
range_transition initrc_t cupsd_exec_t:process s15:c0.c255;
```

mlsconstrain

This is described in the **Constraint Statements - *mlsconstrain*** section.

mlsvalidatetrans

This is described in the **Constraint Statements - *mlsvalidatetrans*** section.

Security ID (SID) Statement

There are two *sid* statements, the first one declares the actual *sid* identifier and is defined at the start of a policy source file. The second statement is used to associate an initial security context to the *sid*, this is used when SELinux initialises but the policy has not yet been activated or as a default context should an object have an invalid label.

sid

The *sid* statement declares the actual SID identifier and is defined at the start of a policy source file.

The statement definition is:

```
sid sid_id
```

Where:

sid

The *sid* keyword.

sid_id

The *sid* identifier.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# This example has been taken from the Reference Policy source
# policy/flask/initial_sids file and declares some of the initial SIDs:
#

sid kernel
sid security
sid unlabeled
sid fs
```

sid context

The *sid context* statement is used to associate an initial security context to the SID.

The statement definition is:

```
sid sid_id context
```

Where:

sid

The *sid* keyword.

sid_id

The previously declared *sid* identifier.

context

The initial security context.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# This is from a targeted policy:

sid unlabeled
...
sid unlabeled system_u:object_r:unlabeled_t
```

```
# This is from an MLS policy. Note that the security level
# is set to SystemHigh as it may need to label any object in
# the system.

sid unlabeled
...
sid unlabeled system_u:object_r:unlabeled_t:s15:c0.c255
```

File System Labeling Statements

There are four types of file labeling statements: *fs_use_xattr*, *fs_use_task*, *fs_use_trans* and *genfscon* that are explained below.

The filesystem identifiers *fs_name* used by these statements are defined by the SELinux teams who are responsible for their development, the policy writer then uses those needed to be supported by the policy.

A security context is defined by these filesystem labeling statements, therefore if the policy supports MCS / MLS, then an *mls_range* is required as described in the **MLS range Definition** section.

fs_use_xattr

The *fs_use_xattr* statement is used to allocate a security context to filesystems that support the extended attribute *security.selinux*. The labeling is persistent for filesystems that support these extended attributes, and the security context is added to these files (and directories) by the SELinux commands such as ***setfiles****(8)* as explained in the **Labeling Extended Attribute Filesystems** section.

The statement definition is:

```
fs_use_xattr fs_name fs_context;
```

Where:

fs_use_xattr

The *fs_use_xattr* keyword.

fs_name

The filesystem name that supports extended attributes. Example names are: *encfs*, *ext2*, *ext3*, *ext4*, *ext4dev*, *gfs*, *gfs2*, *jffs2*, *jfs*, *lustre* and *xfs*.

fs_context

The security context allocated to the filesystem.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# These statements define file systems that support extended
# attributes (security.selinux).

fs_use_xattr encfs system_u:object_r:fs_t:s0;
fs_use_xattr ext2 system_u:object_r:fs_t:s0;
fs_use_xattr ext3 system_u:object_r:fs_t:s0;
```

fs_use_task

The *fs_use_task* statement is used to allocate a security context to pseudo filesystems that support task related services such as pipes and sockets.

The statement definition is:

```
fs_use_task fs_name fs_context;
```

Where:

fs_use_task

The *fs_use_task* keyword.

fs_name

Filesystem name that supports task related services. Example valid names are: *eventpollfs*, *pipefs* and *sockfs*.

fs_context

The security context allocated to the task based filesystem.

The statement is valid in:

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	No

Example:

```
# These statements define the file systems that support pseudo
# filesystems that represent objects like pipes and sockets, so
# that these objects are labeled with the same type as the
# creating task.

fs_use_task eventpollfs system_u:object_r:fs_t:s0;
fs_use_task pipefs system_u:object_r:fs_t:s0;
fs_use_task sockfs system_u:object_r:fs_t:s0;
```

fs_use_trans

The *fs_use_trans* statement is used to allocate a security context to pseudo filesystems such as pseudo terminals and temporary objects. The assigned context is derived from the creating process and that of the filesystem type based on transition rules.

The statement definition is:

```
fs_use_trans fs_name fs_context;
```

Where:

fs_use_trans

The *fs_use_trans* keyword.

fs_name

Filesystem name that supports transition rules. Example names are: *mqueue*, *shm*, *tmpfs* and *devpts*.

fs_context

The security context allocated to the transition based on that of the filesystem.

The statement is valid in:

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# These statements define pseudo filesystems such as devpts
# and tmpfs where objects are labeled with a derived context.

fs_use_trans mqueue system_u:object_r:tmpfs_t:s0;
fs_use_trans shm system_u:object_r:tmpfs_t:s0;
fs_use_trans tmpfs system_u:object_r:tmpfs_t:s0;
fs_use_trans devpts system_u:object_r:devpts_t:s0;
```

genfscon

The *genfscon* statement is used to allocate a security context to filesystems that cannot support any of the other file labeling statements (*fs_use_xattr*, *fs_use_task* or *fs_use_trans*). Generally a filesystem would have a single default security context assigned by *genfscon* from the root (*/*) that would then be inherited by all files and directories on that filesystem. File entries can be, if supported by the underlying filesystem, labeled with a specific security context (as shown in the examples), which is useful for pseudo filesystems exporting kernel state (e.g. *proc*, *sysfs*, *cgroup2*, *securityfs*, *selinuxfs*). Note that there is no terminating semi-colon on this statement.

The statement definition is:

```
genfscon fs_name partial_path [filetype_specifier] fs_context
```

Where:

genfscon

The *genfscon* keyword.

fs_name

The filesystem name.

partial_path

If *fs_name* is a virtual kernel filesystem, then the partial path (see the examples). For all other types, this must be */*.

filetype_specifier

Optional filetype specifier to apply the context only to a specific file type. Valid specifiers are:

- *-b* block device
- *-c* character device
- *-d* directory
- *-p* named pipe
- *-l* symbolic link
- *-s* socket
- – regular file

If omitted the context applies to all file types.

fs_context

The security context allocated to the filesystem

The statement is valid in:

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

MLS Examples:

```
# The following examples show those filesystems that only
# support a single security context across the filesystem
# with the MLS levels added.

genfscon msdos / system_u:object_r:dosfs_t:s0
genfscon iso9660 / system_u:object_r:iso9660_t:s0
```

```
genfscon usbfs / system_u:object_r:usbfs_t:s0
genfscon selinuxfs / system_u:object_r:security_t:s0
```

```
# The following examples show pseudo kernel filesystem entries. Note that
# the /kmsg has the highest sensitivity level assigned (s15) because
# it is a file containing possibly sensitive information.

genfscon cgroup2 "/user.slice" -d system_u:object_r:cgroup_user_slice_t:s0

genfscon proc / system_u:object_r:proc_t:s0
genfscon proc /sysvipc system_u:object_r:proc_t:s0
genfscon proc /fs/openafs system_u:object_r:proc_afs_t:s0
genfscon proc /kmsg system_u:object_r:proc_kmsg_t:s15:c0.c255

genfscon selinuxfs /booleans/secure_mode_policyload --
system_u:object_r:secure_mode_policyload_boolean_t:s0

genfscon sysfs /devices/system/cpu/online -- system_u:object_r:cpu_online_sysfs_t:s0
```

Network Labeling Statements

The network labeling statements are used to label the following objects:

Network interfaces - This covers those interfaces managed by the ***ifconfig****(8)* command.

Network nodes - These are generally used to specify host systems using either IPv4 or IPv6 addresses.

Network ports - These can be either udp, tcp, dccp or sctp port numbers.

A security context is defined by these network labeling statements, therefore if the policy supports MCS / MLS, then an *mls_range* is required as described in the **MLS Statements - MLS range Definition** section. Note that there are no terminating semi-colons ';' on these statements.

If any of the network objects do not have a specific security context assigned by the policy, then the value given in the policies initial SID is used (*netif*, *node* or *port* respectively), as shown below:

```
# Network Initial SIDs from the MLS Reference Policy:

sid netif system_u:object_r:netif_t:s0 - s15:c0.c255
sid node system_u:object_r:node_t:s0 - s15:c0.c255
sid port system_u:object_r:port_t:s0
```

Network Address Formats

IPv4 Address Format

IPv4 addresses are represented in dotted-decimal notation (four numbers, each ranging from 0 to 255, separated by dots as shown:

```
192.77.188.166
```

IPv6 Address Formats

IPv6 addresses are written as eight groups of four hexadecimal digits, where each group is separated by a colon ':' as follows:

```
2001:0db8:85a3:0000:0000:8a2e:0370:7334
```

To shorten the writing and presentation of addresses, the following rules apply:

Any leading zeros in a group may be replaced with a single '0' as shown:

```
2001:db8:85a3:0:0:8a2e:370:7334
```

Any leading zeros in a group may be omitted and be replaced with two colons '::', however this is only allowed once in an address as follows:

```
2001:db8:85a3::8a2e:370:7334
```

The *localhost* (loopback) address can be written as:

```
0000:0000:0000:0000:0000:0000:0000:0001

Or

::1
```

An undetermined IPv6 address i.e. all bits are zero is written as:

```
::
```

netifcon

The *netifcon* statement is used to label network interface objects (e.g. eth0) for peer labeling (see the ***netif* object class**).

It is also possible to use the ***semanage**(8) interface* command to associate the interface to a security context.

The statement definition is:

```
netifcon netif_id netif_context packet_context
```

Where:

netifcon

The *netifcon* keyword.

netif_id

The network interface name (e.g. eth0).

netif_context

The security context allocated to the network interface.

packet_context

The security context allocated packets. Note that these are defined but unused. The ***iptables**(8)* / ***nft**(8)* **SECMARK services** should be used to label packets.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	No

Example:

```
# The following netifcon statement has been taken from the
# MLS policy that shows an interface name of lo with the same
# security context assigned to both the interface and packets.

netifcon lo system_u:object_r:lo_netif_t:s0 - s15:c0.c255
system_u:object_r:unlabeled_t:s0 - s15:c0.c255
```

***semanage**(8)* **Command example:**

```
semanage interface -a -t netif_t eth2
```

This command will produce the following file in the default <SELINUXTYPE> policy store and then activate the policy:

/var/lib/selinux/<SELINUXTYPE>/active/interfaces.local:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

netifcon eth2 system_u:object_r:netif_t:s0
system_u:object_r:netif_t:s0
```

nodecon

The *nodecon* statement is used to label network address objects for peer labeling (see the ***node* object class** that represent IPv4 or IPv6 IP addresses and network masks.

It is also possible to add SELinux these outside the policy using the ***semanage**(8) node* command that will associate the node to a security context.

The statement definition is:

```
nodecon subnet netmask node_context
```

Where:

nodecon

The *nodecon* keyword.

subnet

The subnet or specific IP address in IPv4 or IPv6 format. Note that the subnet and netmask values are used to ensure that the *node_context* is assigned to all IP addresses within the subnet range.

netmask

The subnet mask in IPv4 or IPv6 format.

node_context

The security context for the node.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# The MLS policy nodecon statement using an IPv4 address:

nodecon 127.0.0.1 255.255.255.255 system_u:object_r:lo_node_t:s0 - s15:c0.c255
```

```
# The MLS policy nodecon statement for the multicast address
# using an IPv6 address:

nodecon ff00:: ff00:: system_u:object_r:multicast_node_t:s0 - s15:c0.c255
```

***semanage**(8)* **Command example:**

```
semanage node -a -t node_t -p ipv4 -M 255.255.255.255 127.0.0.2
```

This command will produce the following file in the default <SELINUXTYPE> policy store and then activate the policy:

/var/lib/selinux/<SELINUXTYPE>/active/nodes.local:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

nodecon ipv4 127.0.0.2 255.255.255.255 system_u:object_r:node_t:s0
```

portcon

The *portcon* statement is used to label udp, tcp, dccp or sctp ports.

It is also possible to add a security context to ports outside the policy using the ***semanage****(8) port* command that will associate the port (or range of ports) to a security context.

The statement definition is:

```
portcon protocol port_number port_context
```

Where:

portcon

The *portcon* keyword.

protocol

The protocol type. Valid entries are udp, tcp or dccp.

port_number

The port number or range of ports. The ranges are separated by a hyphen '-'.

port_context

The security context for the port or range of ports.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
# The MLS policy portcon statements:
portcon tcp 20 system_u:object_r:ftp_data_port_t:s0
portcon tcp 21 system_u:object_r:ftp_port_t:s0
portcon tcp 600-1023 system_u:object_r:hi_reserved_port_t:s0
portcon udp 600-1023 system_u:object_r:hi_reserved_port_t:s0
portcon tcp 1-599 system_u:object_r:reserved_port_t:s0
portcon udp 1-599 system_u:object_r:reserved_port_t:s0
```

semanage*(8)* **Command example:**

```
semanage port -a -t reserved_port_t -p udp 1234
```

This command will produce the following file in the default <SELINUXTYPE> policy store and then activate the policy:

/var/lib/selinux/<SELINUXTYPE>/active/ports.local:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

portcon udp 1234 system_u:object_r:reserved_port_t:s0
```

InfiniBand Labeling Statements

To support access control for InfiniBand (IB) partitions and subnet management, security contexts are provided for: Partition Keys (Pkey) that are 16 bit numbers assigned to subnets and their IB end ports. An overview of the SELinux IB implementation can be found at: http://marc.info/?l=selinux&m=149519833917911&w=2.

Note that there are no terminating semi-colons ';' on these statements.

ibpkeycon

The *ibpkeycon* statement is used to label IB partition keys.

It is also possible to add a security context to partition keys outside the policy using the *semanage ibpkey* command that will associate the *pkey* (or range of pkeys) to a security context.

The statement definition is:

```
ibpkeycon subnet pkey pkey_context
```

Where:

ibpkeycon

The *ibpkeycon* keyword.

subnet

IP address in IPv6 format.

pkey

Partition key number or range. The range is separated by a hyphen '-'.

pkey_context

The security context for the pkey(s).

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
ibpkeycon fe80:: 0xFFFF system_u:object_r:default_ibpkey_t:s0
ibpkeycon fe80:: 0-0x10 system_u:object_r:public_ibpkey_t:s0
```

***semanage**(8)* **Command example:**

```
semanage ibpkey -a -t default_ibpkey_t -x fe80:: 0xFFFF
```

The above command will produce the following file: */var/lib/selinux/<SELINUXTYPE>/active/ibpkeys.local* in the default <*SELINUXTYPE*> policy store and then activate the policy:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

ibpkeycon fe80:: 0xFFFF system_u:object_r:default_ibpkey_t:s0
```

ibendportcon

The *ibendportcon* statement is used to label IB end ports.

It is also possible to add a security context to ports outside the policy using the *semanage ibendport* command that will associate the end port to a security context.

The statement definition is:

```
ibendportcon device_id port_number port_context
```

Where:

ibendportcon

The *ibendportcon* keyword.

device_id

Device name

port_number

Single port number.

port_context

The security context for the port.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	No

Examples:

```
ibendportcon mlx4_0 2 system_u:object_r:opensm_ibendport_t:s0
ibendportcon mlx5_0 1 system_u:object_r:opensm_ibendport_t:s0
```

***semanage**(8)* **Command example:**

```
semanage ibendport -a -t opensm_ibendport_t -z mlx4_0 2
```

This command will produce the following file */var/lib/selinux/<SELINUXTYPE>/active/ibendports.local* in the default <*SELINUXTYPE*> policy store and then activate the policy:

```
# This file is auto-generated by libsemanage
# Do not edit directly.

ibendportcon mlx4_0 2 system_u:object_r:opensm_ibendport_t:s0
```

Xen Statements

Xen policy supports additional policy language statements: *iomemcon*, *ioportcon*, *pcidevicecon*, *pirqcon* and *devicetreecon* that are discussed in the sections that follow, also the **XSM/FLASK Configuration** document contains further information.

Policy version 30 introduced the *devicetreecon* statement and also expanded the existing I/O memory range to 64 bits in order to support hardware with more than 44 bits of physical address space (32-bit count of 4K pages).

To compile these additional statements using ***semodule****(8)*, ensure that the ***semanage.conf****(5)* file has the *policy-target=xen* entry.

iomemcon

Label i/o memory. This may be a single memory location or a range.

The statement definition is:

```
iomemcon addr context
```

Where:

iomemcon

The *iomemcon* keyword.

addr

The memory address to apply the context. This may also be a range that consists of a start and end address separated by a hyphen '-'.

context

The security context to be applied.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
iomemcon 0xfebd9 system_u:object_r:nicP_t
iomemcon 0xfebe0-0xfebff system_u:object_r:nicP_t
```

ioportcon

Label i/o ports. This may be a single port or a range.

The statement definition is:

```
ioportcon port context
```

Where:

ioportcon

The *ioportcon* keyword.

port

The *port* to apply the context. This may also be a range that consists of a start and end port number separated by a hyphen '-'.

context

The security context to be applied.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Examples:

```
ioportcon 0xeac0 system_u:object_r:nicP_t
ioportcon 0xecc0-0xecdf system_u:object_r:nicP_t
```

pcidevicecon

Label a PCI device.

The statement definition is:

```
pcidevicecon pci_id context
```

Where:

pcidevicecon

The *pcidevicecon* keyword.

pci_id

The PCI identifier.

context

The security context to be applied.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
Yes	Yes	No

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
pcidevicecon 0xc800 system_u:object_r:nicP_t
```

pirqcon

Label an interrupt level.

The statement definition is:

```
pirqcon irq context
```

Where:

pirqcon

The *pirqcon* keyword.

irq

The interrupt request number.

context

The security context to be applied.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	No

Example:

```
pirqcon 33 system_u:object_r:nicP_t
```

devicetreecon

Label device tree nodes.

The statement definition is:

```
devicetreecon path context
```

Where:

devicetreecon

The *devicetreecon* keyword.

path

The device tree path. If this contains spaces enclose within “” as shown in the example.

context

The security context to be applied.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
Yes	Yes	No

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	No	No

Example:

```
devicetreecon "/this is/a/path" system_u:object_r:arm_path
```

Modular Policy Support Statements

This section contains statements used to support policy modules. They are not part of the kernel policy language.

module

This statement is mandatory for loadable modules (non-base) and must be the first line of any module policy source file. The identifier should not conflict with other module names within the overall policy, otherwise it will over-write an existing module when loaded via the semodule command. The *semodule -l* command can be used to list all active modules within the policy.

The statement definition is:

```
module module_name version_number;
```

Where:

module

The *module* keyword.

module_name

The *module* name.

version_number

The module version number in M.m.m format (where M = major version number and m = minor version numbers). Since Reference Policy release 2.20220106 the *version_number* argument is optional. If missing '1' is set as a default to satisfy the policy syntax.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
No	No	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
No	No	No

Example:

```
# Using the module statement to define a loadable module called
# bind with a version 1.0.0:
```

```
module bind 1.0.0;
```

require

The require statement is used for two reasons:

1. Within loadable module policy source files to indicate what policy components are required from an external source file (i.e. they are not explicitly defined in this module but elsewhere). The examples below show the usage.
2. Within a base policy source file, but only if preceded by the ***optional*** to indicate what policy components are required from an external source file (i.e. they are not explicitly defined in the base policy but elsewhere). The examples below show the usage.

The statement definition is:

```
require { rule_list }
```

Where:

require

The *require* keyword.

require_list

One or more specific statement keywords with their required identifiers in a semi-colon ';' separated list enclosed within braces '{}'. The examples below show these in detail. The valid statement keywords are:

- *role*, *type*, *attribute*, *user*, *bool*, *sensitivity* and *category* - The keyword is followed by one or more identifiers in a comma ',' separated list, with the last entry being terminated with a semi-colon ';'.
- *class* - The class keyword is followed by a single object class identifier and one or more permissions. Multiple permissions consist of a space separated list enclosed within braces '{}'. The list is then terminated with a semi-colon ';'.

The statement is valid in:

Policy Type

Monolithic Policy	**Base Policy**	**Module Policy**
No	Yes (only if proceeded by the *optional* Statement)	Yes

Conditional Policy Statements

***if* Statement**	***optional* Statement**	***require* Statement**
Yes (only if proceeded by the *optional* Statement)	Yes	No

Examples:

```
# A series of require statements showing various entries:
require {
```

```
    role system_r;
    class security { compute_av compute_create compute_member
    check_context load_policy compute_relabel compute_user
    setenforce setbool setsecparam setcheckreqprot };
    class capability2 { mac_override mac_admin };
}

#
require {
    attribute direct_run_init, direct_init, direct_init_entry;
    type initrc_t;
    role system_r;
    attribute daemon;
}

#
require {
    type nscd_t, nscd_var_run_t;
    class nscd { getserv getpwd getgrp gethost shmempwd shmemgrp
    shmemhost shmemserv };
}
```

optional

The optional statement is used to indicate what policy statements may or may not be present in the final compiled policy. The statements will be included in the policy only if all statements within the optional `{ rule list }` can be expanded successfully, this is generally achieved by using a ***require*** statement at the start of the list.

The statement definition is:

```
optional { rule_list } [ else { rule_list } ]
```

Where:

optional

The *optional* keyword.

rule_list

One or more statements enclosed within braces '{}'. The list of valid statements is given in **Table 3:** of the Kernel Policy Language section.

else

An optional *else* keyword.

rule_list

As the *rule_list* above.

The statement is valid in:

Policy Type

Monolithic Policy	Base Policy	Module Policy
No	Yes	Yes

Conditional Policy Statements

if Statement	*optional* Statement	*require* Statement
No	Yes	No

Examples:

```
# Use of optional block in a base policy source file.
optional {
    require {
        type unconfined_t;
    } # end require

    allow acct_t unconfined_t:fd use;
} # end optional

# Use of optional / else blocks in a base policy source file.
optional {
    require {
        type ping_t, ping_exec_t;
    } # end require

    allow dhcpc_t ping_exec_t:file { getattr read execute };
    .....

    require {
        type netutils_t, netutils_exec_t;
    } # end require

    allow dhcpc_t netutils_exec_t:file { getattr read execute };
    .....

    type_transition dhcpc_t netutils_exec_t:process netutils_t;
    ...
} else {
    allow dhcpc_t self:capability setuid;
    .....

} # end optional
```

The Reference Policy

The **Reference Policy** is now the standard policy source used to build Linux SELinux policies. This provides a single source tree with supporting documentation that can be used to build policies for different purposes such as: confining important daemons, supporting MLS / MCS type policies and locking down systems so that all processes are under SELinux control.

This section details how the Reference Policy is:

1. Constructed and types of policy builds supported.
2. Adding new modules to the build.
3. Installation as a full Reference Policy source or as Header files.
4. Impact of the migration process being used to convert compiled module files (*.*pp*) to CIL.
5. Modifying the configuration files to build new policies.
6. Explain the support macros.

Note that the Reference Policy uses **NAME** to define the policy name. This then becomes part of the policy location (i.e. */etc/selinux/*<*NAME*>). In most documentation the policy name is defined using the <*SELINUXTYPE*> convention, as that is from the */etc/selinux/config* file entry **SELINUXTYPE**=. This part of the Notebook uses both forms.

Reference Policy Overview

Strictly speaking the 'Reference Policy' should refer to the policy taken from the master repository or the latest released version. This is because most Linux distributors take a released version and then tailor it to their specific requirements.

All examples in this section are based on the master Reference Policy repository that can be checked out using the following:

```
# Check out the core policy:
git clone https://github.com/SELinuxProject/refpolicy.git
```

A list of releases can be found at https://github.com/SELinuxProject/refpolicy/releases

The Fedora distribution is built from a specific standard Reference Policy build, modified and distributed by Red Hat as a source RPM. These RPMs can be obtained from http://koji.fedoraproject.org. The master Fedora policy source can be found at: https://github.com/fedora-selinux/selinux-policy

Figure 26: The Reference Policy Source Tree shows the layout of the reference policy source tree, that once installed would be located at

```
/etc/selinux/<SELINUXTYPE>/src/policy
```

Where the <**SELINUXTYPE**> entry is taken from the *build.conf* file as discussed in the **Reference Policy Build Options** - *build.conf* section. The **Installing and Building the Reference Policy Source** section explains a simple build from source.

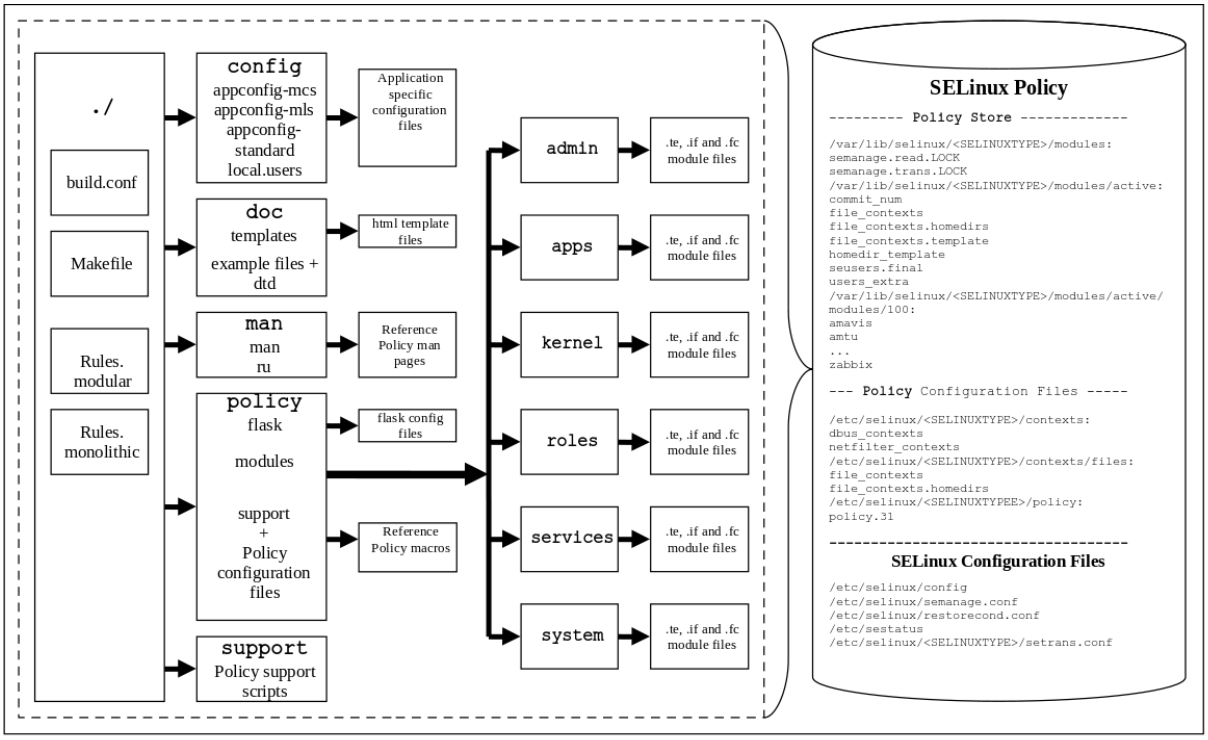


Figure 26: The Reference Policy Source Tree - *When building a modular policy, files are added to the policy store. For monolithic builds the policy store is not used.*

The Reference Policy can be used to build two policy types:

1. **Loadable Module Policy** - A policy that has a base module for core services and has the ability to load / unload modules to support applications as required. This is now the standard used by Linux distributions.
2. **Monolithic Policy** - A policy that has all the required policy information in a single base policy and does not require the services of the module infrastructure (***semanage**(8)* or ***semodule**(8)*). These are more suitable for embedded or minimal systems.

Each of the policy types are built using module files that define the specific rules required by the policy as detailed in the **Reference Policy Module Files** section. Note that the monolithic policy is built using the same module files by forming a single 'base' source file.

The Reference Policy relies heavily on the ***m4**(1)* macro processor as the majority of supporting services are m4 macros.

Distributing Policies

It is possible to distribute the Reference Policy in two forms:

1. As source code that is then used to build policies. This is not the general way policies are distributed as it contains the complete source that most administrators do not need. The **Reference Policy Source** section describes the source and the **Installing and Building the Reference Policy Source** section describes how to install the source and build a policy.
2. As 'Policy Headers'. This is the most common way to distribute the Reference Policy. Basically, the modules that make up 'the distribution' are pre-built and then linked to form a base and optional modules. The 'headers' that make-up the policy are then distributed along with makefiles and documentation. A policy writer can then build policy using the core modules supported by the distribution, and using development tools they can add their own policy modules. The **Reference Policy Headers** section describes how these are installed and used to build modules.

The policy header files and documentation for Fedora are distributed in:

- **selinux-policy** - Contains the SELinux */etc/selinux/config* file and rpm macros
- **selinux-policy-devel** - Contains the 'Policy Header' development environment that is located at */usr/share/selinux/devel*
- **selinux-policy-doc** - Contains man pages and the html policy documentation that is located at */usr/share/doc/selinux-policy/html*

These rpms contain a specific policy type containing configuration files and packaged policy modules (**.pp*). The policy will be installed in the */etc/selinux/<SELINUXTYPE>* directory, along with its configuration files.

- **selinux-policy-targeted** - This is the default Fedora policy.
- selinux-policy-minimum
- selinux-policy-mls

The selinux-policy-sandbox rpm contains the sandbox module for use by the *policycoreutils-sandbox* package. This will be installed as a module for one of the three main policies described above.

Policy Functionality

As can be seen from the policies distributed with Fedora above, they can be classified by the name of the functionality they support (taken from the *SELINUXTYPE* entry of the *build.conf* as shown in the **Reference Policy Build Options - build.conf** section, for example the Fedora policies support:

- minimum - MCS policy that supports a minimal set of confined daemons within their own domains. The remainder run in the *unconfined_t* space.
- targeted - MCS policy that supports a greater number of confined daemons and can also confine other areas and users.
- mls - MLS policy for server based systems.

Reference Policy Module Files

The reference policy modules are constructed using a mixture of **support macros**, **interface calls** and **Kernel Policy Language Statements**, using three principle types of source file:

- A private policy file that contains statements required to enforce policy on the specific GNU / Linux service being defined within the module. These files are named *<module_name>.te*. For example the *ada.te* file shown below has two statements:
  - one to state that the *ada_t* process has permission to write to the stack and memory allocated to a file.
  - one that states that if the *unconfined_domain* module is loaded, then allow the *ada_t* domain unconfined access. Note that if the flow of this statement is followed it will be seen that many more interfaces and macros are called to build the final raw SELinux language statements. An expanded module source isshown in the **Module Expansion Process** section.
- An external interface file that defines the services available to other modules. These files are named *<module_name>.if*. For example the *ada.if* file shown below has two interfaces defined for other modules to call:
  - *ada_domtrans* - that allows another module (running in domain *$1*) to run the ada application in the *ada_t* domain.
  - *ada_run* - that allows another module to run the ada application in the *ada_t* domain (via the *ada_domtrans* interface), then associate the *ada_t* domain to the caller defined role (*$2*) and terminal (*$3*). Provided of course that the caller domain has permission. Note that there are two types of interface specification:
  - **Access Interfaces** - These are the most common and define interfaces that *.te* modules can call as described in the ada examples. They are generated by the *interface* macro as detailed in the *interface* section.

    - **Template Interfaces** - These are required whenever a module is required in different domains and allows the type(s) to be redefined by adding a prefix supplied by the calling module. The basic idea is to set up an application in a domain that is suitable for the defined SELinux user and role to access but not others. These are generated by the *template* macro as detailed in the *template* section.
- A file labeling file that defines the labels to be added to files for the specified module. These files are named <*module_name*>*.fc*. The build process will amalgamate all the **.fc* files and finally form the ***file_contexts*** file that will be used to label the filesystem. For example the *ada.fc* file shown below requires that the specified files are labeled *system_u:object_r:ada_exec_t:s0*. The <*module_name*> must be unique within the reference policy source tree and should reflect the specific Linux service being enforced by the policy.

The following examples from the ada module files show how they are made from the Policy Macros, Interface calls and kernel policy statements and rules:

ada.te file contents:

```
policy_module(ada, 1.5.0)

########################################
#
# Declarations
#

attribute_role ada_roles;           # Kernel policy statement
roleattribute system_r ada_roles;

type ada_t;
type ada_exec_t;
application_domain(ada_t, ada_exec_t)       # call into 'application.if'
role ada_roles types ada_t;

########################################
#
# Local policy
#

allow ada_t self:process { execstack execmem };

userdom_use_inherited_user_terminals(ada_t) # call into 'userdomain.if'

optional_policy(`               # Macro in loadable_module.spt
    unconfined_domain(ada_t)
')
```

ada.if 'interface calls' file contents:

```
## <summary>GNAT Ada95 compiler.</summary>

########################################
## <summary>
##  Execute the ada program in the ada domain.
## </summary>
## <param name="domain">
##  <summary>
##  Domain allowed to transition.
```

```
##  </summary>
## </param>
#
interface(*ada_domtrans',*          # Defining an interface
    gen_require(`               # Macro in loadable_module.spt
        type ada_t, ada_exec_t;
    ')

    corecmd_search_bin($1)
    domtrans_pattern($1, ada_exec_t, ada_t) # Macro in misc_patterns.spt
')

########################################
## <summary>
##  Execute ada in the ada domain, and
##  allow the specified role the ada domain.
## </summary>
## <param name="domain">
##  <summary>
##  Domain allowed to transition.
##  </summary>
## </param>
## <param name="role">
##  <summary>
##  Role allowed access.
##  </summary>
## </param>
#
interface(*ada_run',*
    gen_require(`
        attribute_role ada_roles;
    ')

    ada_domtrans($1)
    roleattribute $2 ada_roles;     # Kernel policy statement
')
```

ada.fc file contents:

```
                # gen_context is a macro in misc_macros.spt
/usr/bin/gnatbind   --  gen_context(system_u:object_r:ada_exec_t,s0)
/usr/bin/gnatls --  gen_context(system_u:object_r:ada_exec_t,s0)
/usr/bin/gnatmake   --  gen_context(system_u:object_r:ada_exec_t,s0)

/usr/libexec/gcc(/.*)?/gnat1    --  gen_context(system_u:object_r:ada_exec_t,s0)
```

Reference Policy Documentation

One of the advantages of the reference policy is that it is possible to automatically generate documentation as a part of the build process. This documentation is defined in XML and generated as HTML files suitable for viewing via a browser.

The documentation for Fedora can be viewed in a browser using */usr/share/doc/selinux-policy/html/index.html* once the *selinux-policy-doc* rpm has been installed. The documentation for the Reference Policy source will be available

at <*location*>*/src/policy/doc/html* once *make html* has been executed (the <*location*> is the location of the installed source after *make install-src* has been executed as described in the **Installing and Building the Reference Policy Source** section). The Reference Policy documentation may also be available at a default location of */usr/share/doc/refpolicy-*<*VERSION*>*/html* if *make install-doc* has been executed (where <*VERSION*/> is the entry from the source *VERSION* file.

Figure 27 shows an example screen shot of the documentation produced for the ada module interfaces.

Figure 27: Example Documentation Screen Shot

Reference Policy Source

This section explains the source layout and configuration files, with the actual installation and building covered in the **Installing and Building the Reference Policy Source** section.

The source has a README file containing information on the configuration and installation processes that has been used within this section (and updated with the authors comments as necessary). There is also a VERSION file that contains the Reference Policy release date, this can then be used to obtain a change list https://github.com/SELinuxProject/refpolicy/releases.

Source Layout

Figure 26: The Reference Policy Source Tree shows the layout of the reference policy source tree, that once installed would be located at:

/etc/selinux/<*SELINUXTYPE*>*/src/policy*

The following sections detail the source contents:

- **Reference Policy Files and Directories** - Describes the files and their location.
- **Source Configuration Files** - Details the contents of the *build.conf* and *modules.conf* configuration files.
- **Source Installation and Build Make Options** - Describes the *make* targets.

- **Modular Policy Build Structure** - Describes how the various source files are linked together to form a base policy module *base.conf* during the build process.

The **Installing and Building the Reference Policy Source** section then describes how the initial source is installed and configured to allow a policy to be built.

Reference Policy Files and Directories

The **Reference Policy Files and Directories** list shows the major files and their directories with a description of each taken from the README file (with comments added). All directories are relative to the root of the Reference Policy source directory *./policy*.

The *build.conf* and *modules.conf* configuration files are further detailed in the **Source Configuration Files** section as they define how the policy will be built.

During the build process, a file is generated in the *./policy* directory called either *policy.conf* or *base.conf* depending whether a monolithic or modular policy is being built. This file is explained in the **Modular Policy Build Structure** section.

Reference Policy Files and Directories:

Makefile

- General rules for building the policy.

Rules.modular

- Makefile rules specific to building loadable module policies.

Rules.monolithic

- Makefile rules specific to building monolithic policies.

build.conf

- Options which influence the building of the policy, such as the policy type and distribution. This file is described in the **Reference Policy Build Options - build.conf** section.

config/*appconfig-<type>*

- Application configuration files for all configurations of the Reference Policy where <*type*> is taken from the *build.conf* **TYPE** entry that are currently: standard, MLS and MCS). These files are used by SELinux-aware programs and described in the **SELinux Configuration Files** section.

config/file_contexts.subs_dist

- Used to configure file context aliases (see the **contexts/files/file_contexts.subs and file_contexts.subs_dist File** section).

config/local.users

- The file read by load policy for adding SELinux users to the policy on the fly. Note that this file is not used in the modular policy build.

doc/html/*

- When *make html* has been executed, contains the in-policy XML documentation, presented in web page form.

doc/policy.dtd

- The *doc/policy.xml* file is validated against this DTD.

doc/policy.xml

- This file is generated/updated by the conf and html make targets. It contains the complete XML documentation included in the policy.

doc/templates/*

- Templates used for documentation web pages.

man/*

- Various man pages for modules (ftp, http etc.)

support/*

- Tools used in the build process.

policy/flask/initial_sids

- This file has declarations for each initial SID. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/flask/security_classes

- This file has declarations for each security class. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/flask/access_vectors

- This file defines the common permissions and class specific permissions and is described in the **Modular Policy Build Structure** section.

policy/modules/*

- Each directory represents a layer in Reference Policy. All of the modules are contained in one of these layers. The *contrib* modules are supplied externally to the Reference Policy, then linked into the build. The files present in each directory are: *metadata.xml* that describes the layer and *<module_name>.te*, *.if* and *.fc* that contain policy source as described in the **Reference Policy Module Files** section. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/support/*

- Reference Policy support macros are described in the **Reference Policy support Macros** section.

policy/booleans.conf

- This file is generated/updated by *make conf*. It contains the booleans in the policy and their default values. If tunables are implemented as booleans, tunables will also be included. This file will be installed as the */etc/selinux/<NAME>/booleans file* (note that this is not true for any system that implements the modular policy - see the **Booleans, Global Booleans and Tunable Booleans** section).

policy/constraints

- This file defines constraints on permissions in the form of boolean expressions that must be satisfied in order for specified permissions to be granted. These constraints are used to further refine the type enforcement rules and the role allow rules. Typically, these constraints are used to restrict changes in user identity or role to certain domains. (Note that this file does not contain the MLS / MCS constraints as they

are in the *mls* and *mcs* files described below). The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/context_defaults

- This would contain any specific *default_user*, *default_role*, *default_type* and/or *default_range* rules required by the policy.

policy/global_booleans

- This file defines all booleans that have a global scope, their default value, and documentation. See the **Booleans, Global Booleans and Tunable Booleans** section.

policy/global_tunables

- This file defines all tunables that have a global scope, their default value, and documentation. See the **Booleans, Global Booleans and Tunable Booleans** section.

policy/mcs

- This contains information used to generate the *sensitivity*, *category*, *level* and *mlsconstraint* statements used to define the MCS configuration. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/mls

- This contains information used to generate the *sensitivity*, *category*, *level* and *mlsconstraint* statements used to define the MLS configuration. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/modules.conf

- This file contains a listing of available modules, and how they will be used when building Reference Policy. To prevent a module from being used, set the module to “off”. For monolithic policies, modules set to “base” and “module” will be included in the policy. For modular policies, modules set to “base” will be included in the base module; those set to “module” will be compiled as individual loadable modules. This file is described in the **Reference Policy Build Options - policy/modules.conf** section.

policy/policy_capabilities

- This file defines the policy capabilities that can be enabled in the policy. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

policy/users

- This file defines the users included in the policy. The file usage in policy generation is described in the **Modular Policy Build Structure** section.

securetty_types and *setrans.conf*

- These files are not part of the standard Reference Policy distribution but are added by Fedora source updates.

Source Configuration Files

There are two major configuration files (*build.conf* and *modules.conf*) that define the policy to be built and are detailed in this section.

Reference Policy Build Options - build.conf

This file defines the policy type to be built that will influence its name and where the source will be located once it is finally installed. An example file content is shown in the **Installing and Building the Reference Policy Source** section where it is used to install and then build the policy.

The *build.conf* **Entries** below explains the fields that can be defined within this file. The supplied fields will then be used by the *make* process to set *m4* macro parameters. These macro definitions are also used within the module source files to control how the policy is built with examples shown in the ***ifdef*** section.

***build.conf* Entries:**

TYPE

- String - Available options are *standard*, *mls*, and *mcs*. For a type enforcement only system, set standard. This optionally enables multi-level security (MLS) or multi-category security (MCS) features. This option controls *enable_mls*, and *enable_mcs* policy blocks.

NAME

- String (optional) - Sets the name of the policy; the <*NAME*> is used when installing files to e.g., */etc/selinux/*<*NAME*> and */usr/share/selinux/*<*NAME*>. If not set, the policy type (*TYPE*) is used.

DISTRO

- String (optional) - Enable distribution-specific policy. Available options are redhat, gentoo, and debian. This option controls *distro_redhat*, *distro_gentoo*, and *distro_debian* build option policy blocks.

MONOLITHIC

- Boolean - If set, a monolithic policy is built, otherwise a modular policy is built.

DIRECT_INITRC

- Boolean - If set, sysadm will be allowed to directly run init scripts, instead of requiring the *run_init* tool. This is a build option instead of a tunable since role transitions do not work in conditional policy. This option controls *direct_sysadm_daemon* policy blocks.

OUTPUT_POLICY

- Integer - Set the version of the policy created when building a monolithic policy. This option has no effect on modular policy.

UNK_PERMS

- String - Set the kernel behavior for handling of permissions defined in the kernel but missing from the policy. The permissions can either be allowed (*allow*), denied (*deny*), or the policy loading can be rejected (*reject*).

UBAC

- Boolean - If set, the SELinux user will be used additionally for approximate role separation.

SYSTEMD

- Boolean - If set, ***systemd****(1)* will be assumed to be the init process provider.

MLS_SENS

- Integer - Set the number of sensitivities in the MLS policy. Ignored on *TYPE* entries of *standard* and *mcs*.

MLS_CATS

- Integer - Set the number of categories in the MLS policy. Ignored on *TYPE* entries of *standard* and *mcs*.

MCS_CATS

- Integer - Set the number of categories in the MCS policy. Ignored on *TYPE* entries of *standard* and *mls*.

QUIET

- Boolean - If set, the build system will only display status messages and error messages. This option has no effect on policy.

WERROR

- Boolean - If set, the build system will treat warnings as errors. If any warnings are encountered, the build will fail.

Reference Policy Build Options - policy/modules.conf

This file will not be present until *make conf* is run and controls what modules are built within the policy, see the **Building the modules.conf File** section.

Example entries:

```
# Layer: kernel
# Module: kernel
# Required in base
#
# Policy for kernel threads, proc filesystem,and unlabeled processes and
# objects.
#
kernel = base

# Layer: admin
# Module: amanda
#
# Advanced Maryland Automatic Network Disk Archiver.
#
amanda = module

# Layer: admin
# Module: ddcprobe
#
# ddcprobe retrieves monitor and graphics card information
#

ddcprobe = off
```

The only active lines (those without comments) contain:

```
<module_name> = base | module | off
```

However note that the comments are important as they form part of the documentation when it is generated by the *make html* target.

Where:

module_name

- The name of the module to be included within the build.

base

- The module will be in the base module for a modular policy build (*build.conf* entry *MONOLITHIC* = n).

module

- The module will be built as a loadable module for a modular policy build. If a monolithic policy is being built (*build.conf* entry *MONOLITHIC* = y), then this module will be built into the base module.

off

- The module will not be included in any build.

Generally it is up to the policy distributor to decide which modules are in the base and those that are loadable, however there are some modules that MUST be in the base module. To highlight this there is a special entry at the start of the modules interface file (*.if*) that has the entry <*required val=“true”*> as shown below (taken from the *kernel.if* file):

```
## <summary>
##  Policy for kernel threads, proc filesystem,
##  and unlabeled processes and objects.
## </summary>
## <required val="true">
##  This module has initial SIDs.
## </required>
```

The *modules.conf* file will also reflect that a module is required in the base by adding a comment ‘Required in base’ when the make conf target is executed (as all the *.if* files are checked during this process and the *modules.conf* file updated).

```
# Layer: kernel
# Module: kernel
# Required in base
#
# Policy for kernel threads, proc filesystem,
# and unlabeled processes and objects.
#
kernel = base
```

Those marked as *Required in base* are shown in the **Mandatory modules.conf Entries** (note that Fedora and the standard Reference Policy may be different)

Mandatory *modules.conf* Entries: ‘Layer - Module Name - Comments’

kernel

- *corecommands*
  - Core policy for shells and generic programs in: */bin*, */sbin*, */usr/bin*, and */usr/sbin*. The *.fc* file sets up the labels for these items.
  - All the interface calls start with ‘*corecmd_*’.

- *corenetwork*
  - Policy controlling access to network objects and also contains the initial SIDs for these. The *.if* file is large and automatically generated.
  - All the interface calls start with '*corenet_*'.
- *devices*
  - This module creates the device node concept and provides the policy for many of the device files. Notable exceptions are the mass storage and terminal devices that are covered by other modules (that is a char or block device file, usually in */dev*). All types that are used to label device nodes should use the dev_node macro. Additionally this module controls access to:
    1. the device directories containing device nodes.
    2. device nodes as a group
    3. individual access to specific device nodes covered by this module.
  - All the interface calls start with '*dev_*'.
- *domain*
  - Contains the core policy for forming and managing domains.
  - All the interface calls start with '*domain_*'.
- *files*
  - This module contains basic filesystem types and interfaces and includes:
    1. The concept of different file types including basic files, mount points, tmp files, etc.
    2. Access to groups of files and all files.
    3. Types and interfaces for the basic filesystem layout (*/*, */etc*, */tmp* ...).
    4. Contains the file initial SID.
  - All the interface calls start with '*files_*'.
- *filesystem*
  - Contains the policy for filesystems and the initial SID.
  - All the interface calls start with '*fs_*'.
- *kernel*
  - Contains the policy for kernel threads, proc filesystem, and unlabeled processes and objects. This module has initial SIDs.
  - All the interface calls start with '*kernel_*'.
- *mcs*
  - Policy for Multicategory security. The *.te* file only contains attributes used in MCS policy.
  - All the interface calls start with '*mcs_*'.
- *mls*
  - Policy for Multilevel security. The *.te* file only contains attributes used in MLS policy. All the interface calls start with '*mls_*'.
- *selinux*
  - Contains the policy for the kernel SELinux security interface (*selinuxfs*).
  - All the interface calls start with '*selinux_*'.
- *terminal*
  - Contains the policy for terminals.
  - All the interface calls start with '*term_*'.
- *ubac*
  - Disabled by Fedora but enabled on standard Ref Policy. Support user-based access control.

system

- *application*
  - Enabled by Fedora but not standard Ref Policy. Defines attributes and interfaces for all user apps.
- *setrans*
  - Enabled by Fedora but not standard Ref Policy. Support for ***mcstransd****(8)*.

Building the modules.conf File

The file can be created by an editor, however it is generally built initially by *make conf* that will add any additional modules to the file. The file can then be edited to configure the required modules as base, module or off.

As will be seen in the **Installing and Building the Reference Policy Source** section, the Fedora reference policy source comes with a number of pre-configured files that are used to produce the required policy including multiple versions of the *modules.conf* file.

Source Installation and Build Make Options

This section explains the various make options available that have been taken from the *README* file (with some additional minor comments).

General Make targets:

install-src

- Install the policy sources into */etc/selinux/<NAME>/src/policy*, where <NAME> is defined in the *build.conf* file. If it is not defined, then *TYPE* is used instead. If a *build.conf* does not have the information, then the Makefile will default to the current entry in the */etc/selinux/config* file or default to *refpolicy*. A pre-existing source policy will be moved to */etc/selinux/<NAME>/src/policy.bak*.

conf

- Regenerate *policy.xml*, and update/create *modules.conf* and *booleans.conf*. This should be done after adding or removing modules, or after running the bare target. If the configuration files exist, their settings will be preserved. This must be run on policy sources that are checked out from the CVS repository before they can be used. Note that if *make bare* has been executed before this make target, or it is a first build, then the *modules/kernel/corenetwork.??.in* files will be used to generate the *corenetwork.te* and *corenetwork.if* module files. These **.in* files may be edited to configure network ports etc. (see the *# network_node* example entries in *corenetwork.te*).

clean

- Delete all temporary files, compiled policy, and file_contexts. Configuration files are left intact.

bare

- Do the clean make target and also delete configuration files, web page documentation, and policy.xml.

html

- Regenerate *policy.xml* and create web page documentation in the *doc/html* directory.

install-appconfig

- Installs the appropriate SELinux-aware configuration files.

Make targets specific to modular (loadable modules) policies:

base

- Compile and package the base module. This is the default target for modular policies.

modules

- Compile and package all Reference Policy modules configured to be built as loadable modules.

MODULENAME.pp

- Compile and package the *MODULENAME* Reference Policy module.

all

- Compile and package the base module and all Reference Policy modules configured to be built as loadable modules.

install

- Compile, package, and install the base module and Reference Policy modules configured to be built as loadable modules.

load

- Compile, package, and install the base module and Reference Policy modules configured to be built as loadable modules, then insert them into the module store.

validate

- Validate if the configured modules can successfully link and expand.

install-headers

- Install the policy headers into */usr/share/selinux/<NAME>*. The headers are sufficient for building a policy module locally, without requiring the complete Reference Policy sources. The *build.conf* settings for this policy configuration should be set before using this target.

install-docs

- Build and install the documentation and example module source with Makefile. The default location is */usr/share/doc/refpolicy-<VERSION>*, where the version is the value in the *VERSION* file.

Make targets specific to monolithic policies:

policy

- Compile a policy locally for development and testing. This is the default target for monolithic policies.

install

- Compile and install the policy and file contexts.

load

- Compile and install the policy and file contexts, then load the policy.

enableaudit

- Remove all dontaudit rules from policy.conf.

relabel

- Relabel the filesystem.

checklabels

- Check the labels on the filesystem, and report when a file would be relabeled, but do not change its label.

restorelabels

- Relabel the filesystem and report each file that is relabeled.

Booleans, Global Booleans and Tunable Booleans

The three files *booleans.conf*, *global_booleans* and *global_tunables* are built and used as follows:

booleans.conf

- This file is generated / updated by *make conf*, and contains all the booleans in the policy with their default values. If tunable and global booleans are implemented then these are also included. This file can also be delivered as a part of the Fedora reference policy source as shown in the **Installing and Building the Reference Policy Source** section. This is generally because other default values are used for booleans and not those defined within the modules themselves (i.e. distribution specific booleans). When the *make install* is executed, this file will be used to set the default values. Note that if booleans are updated locally the policy store will contain a ***booleans.local*** file.

global_booleans

- These are booleans that have been defined in the *global_tunables* file using the ***gen_bool*** macro. They are normally booleans for managing the overall policy and currently consist of the following (where the default values are *false*): *secure_mode*

global_tunables

- These are booleans that have been defined in module files using the ***gen_tunable*** macro and added to the *global_tunables* file by *make conf*. The ***tunable_policy*** macros are defined in each module where policy statements or interface calls are required. They are booleans for managing specific areas of policy that are global in scope. An example is *allow_execstack* that will allow all processes running in *unconfined_t* to make their stacks executable.

Modular Policy Build Structure

This section explains the way a modular policy is constructed, this does not really need to be known but is used to show the files used that can then be investigated if required.

When *make all* or *make load* or *make install* are executed the *build.conf* and *modules.conf* files are used to define the policy name and what modules will be built in the base and those as individual loadable modules.

Basically the source modules (*.te*, *.if* and *.fc*) and core flask files are rebuilt in the *tmp* directory where the reference policy macros in the source modules will be expanded to form actual policy language statements as described in the **Kernel Policy Language** section. The **Base Module Build** section shows these temporary files that are used to form the *base.conf* (for modular policies) or the *policy.conf* (for a monolithic policy) file during policy generation.

The *base.conf* file will consist of language statements taken from the module defined as *base* in the *modules.conf* file along with the constraints, users etc. that are required to build a complete policy.

The individual loadable modules are built in much the same way as shown in the **Module Build** section.

Base Module Build

The following shows the temporary build files used to build the base module *base.conf* as a part of the *make* process.

tmp/pre_te_files.conf

- *policy/flask/security_classes*
  - The object classes supported by the kernel
- *policy/flask/initial_sids*
  - The initial SIDs supported by the kernel.

- *policy/flask/access_vectors*
  - The object class permissions supported by the kernel.
- *policy/mls* or *policy/mcs*
  - This is either the expanded mls or mcs file depending on the type of policy being built.
- *policy/policy_capabilities*
  - These are the policy capabilities that can be configured / enabled to support the policy.

tmp/all_attrs_types.conf

- *policy/modules/*/*.te* and *policy/modules/*/*.if*
  - This area contains all the *attribute*, *bool*, *type* and *typealias* statements extracted from the *.te* and *.if* files that form the base module.

tmp/global_bools.conf

- *policy/global_tunables.conf* and *policy/global_bools.conf*
  - Contains the global and tunable bools extracted from the conf files.

tmp/only_te_rules.conf

- *policy/modules*
  - Contains the rules extracted from each of the modules *.te* and *.if* files defined in the *modules.conf* file as ***base***.

tmp/all_post.conf

- *policy/users*
  - Contains the expanded users from the users file.
- *policy/constraints*
  - Contains the expanded constraints from the constraints file.
- *policy/modules/*/*.te*
  - Contains the default SID labeling extracted from the *.te* files.
- *policy/modules/*/*.te* and *policy/modules/*/*.if*
  - Contains the *fs_use_xattr*, *fs_use_task*, *fs_use_trans* and *genfscon* statements extracted from each of the modules *.te* and *.if* files defined in the *modules.conf* file as ***base***.
  - Contains the *netifcon*, *nodecon* and *portcon* statements extracted from each of the modules *.te* and *.if* files defined in the *modules.conf* file as ***base***.

tmp/base.fc.tmp

- *policy/modules/*/*.fc*
  - Contains the expanded file context file entries extracted from the *.fc* files defined in the *modules.conf* file as ***base***.

tmp/seusers

- *policy/seusers*
  - Expanded seusers file.

These are the commands used to compile, link and load the base policy module, where *UNK_PERMS* and *NAME* are those defined in the *build.conf* file:

```
semodule_package -o $@ -m $(base_mod) -f $(base_fc) -u $(users_extra) -s $(tmpdir)/
seusers
...
checkmodule -U $(UNK_PERMS) base.conf -o tmp/base.mod*
...
```

```
semodule -s $(NAME) -i $(modpkgdir)/$(notdir $(base_pkg)) $(foreach mod,$(mod_pkgs),-i
$(modpkgdir)/$(mod))
```

Module Build

The following shows the temporary module files used to build each module as a part of the *make* process (i.e. those modules marked as *module* in *modules.conf*).

tmp/<module_name>.tmp

- *policy/modules/*/<module_name>.te* and *policy/modules/*/<module_name>.if*
  - For each module defined as *module* in the *modules.conf* configuration file, a source module is produced that has been extracted from the *.te* and *.if* files for that module.

tmp<module_name>.mod

- *tmp/<module_name>.tmp*
  - For each module defined as *module* in the *modules.conf* configuration file, an object module is produced from executing the ***checkmodule****(8)* command shown below.

tmp/<module_name>.fc.tmp

- *policy/modules/*/<module_name>.fc*
  - For each module defined as *module* in the *modules.conf* configuration file, an expanded file context file is built.

These are the commands (from *Rules.modular*) is used to build each module and insert it into the module store, where *NAME* is that defined in the *build.conf* file:

```
checkmodule -m $(@:.mod=.tmp) -o $@
...
semodule_package -o $@ -m $< -f $<.fc
...
semodule -s $(NAME) -i $(modpkgdir)/$(notdir $(base_pkg)) $(foreach mod,$(mod_pkgs),-i
$(modpkgdir)/$(mod))
```

Creating Additional Layers

One objective of the reference policy is to separate the modules into different layers reflecting their ‘service’ (e.g. kernel, system, app etc.). While it can sometimes be difficult to determine where a particular module should reside, it does help separation, however because the way the build process works, each module must have a unique name.

If a new layer is required, then the following will need to be completed:

1. Create a new layer directory *./policy/modules/LAYERNAME* that reflects the layer’s purpose.
2. In the *./policy/modules/LAYERNAME* directory create a metadata.xml file. This is an XML file with a summary tag and optional desc (long description) tag that should describe the purpose of the layer and will be used as a part of the documentation. An example is as follows:

```
<summary>ABC modules for the XYZ components.</summary>
```

Installing and Building the Reference Policy Source

This section will give a brief overview of how to build the Reference Policy for an MCS modular build that is similar (but not the same) as the Fedora targeted policy. The Fedora version of the targeted policy build is discussed but building without using the rpm spec file is more complex.

Building Standard Reference Policy

This will run through a simple configuration process and build of a reference policy similar to the Fedora targeted policy. By convention the source is installed in a central location and then for each type of policy a copy of the source is installed at */etc/selinux/<SELINUXTYPE>/src/policy*.

The basic steps are:

- Install master Reference Policy Source:

```
# Check out the core policy:
git clone https://github.com/SELinuxProject/refpolicy.git
```

- Edit the *build.conf* file to reflect the policy to be built, the minimum required is setting the *NAME* = entry. An example file with *NAME* = *refpolicy-test* is as follows:

```
########################################
#
# Policy build options
#

# Policy version
# By default, checkpolicy will create the highest
# version policy it supports.  Setting this will
# override the version.  This only has an
# effect for monolithic policies.
#OUTPUT_POLICY = 18

# Policy Type
# standard, mls, mcs
# Note Red Hat always build the MCS Policy Type for their 'targeted' version.
TYPE = mcs

# Policy Name
# If set, this will be used as the policy
# name.  Otherwise the policy type will be
# used for the name.
# This entry is also used by the 'make install-src' process to copy the
# source to the /etc/selinux/<NAME>/src/policy directory.
NAME = refpolicy-test

# Distribution
# Some distributions have portions of policy
# for programs or configurations specific to the
# distribution.  Setting this will enable options
# for the distribution.
# redhat, gentoo, debian, suse, and rhel4 are current options.
# Fedora users should enable redhat.
```

```
DISTRO = redhat

# Unknown Permissions Handling
# The behavior for handling permissions defined in the
# kernel but missing from the policy.  The permissions
# can either be allowed, denied, or the policy loading
# can be rejected.
# allow, deny, and reject are current options.
# Fedora use allow for all policies except MLS that uses 'deny'.
UNK_PERMS = deny

# Direct admin init
# Setting this will allow sysadm to directly
# run init scripts, instead of requiring run_init.
# This is a build option, as role transitions do
# not work in conditional policy.
DIRECT_INITRC = n

# Systemd
# Setting this will configure systemd as the init system.
SYSTEMD = y

# Build monolithic policy.  Putting y here
# will build a monolithic policy.
MONOLITHIC = n

# User-based access control (UBAC)
# Enable UBAC for role separations.
# Note Fedora disables UBAC.
UBAC = n

# Custom build options.  This field enables custom
# build options.  Putting foo here will enable
# build option blocks named foo.  Options should be
# separated by spaces.
CUSTOM_BUILDOPT =

# Number of MLS Sensitivities
# The sensitivities will be s0 to s(MLS_SENS-1).
# Dominance will be in increasing numerical order
# with s0 being lowest.
MLS_SENS = 16

# Number of MLS Categories
# The categories will be c0 to c(MLS_CATS-1).
MLS_CATS = 1024

# Number of MCS Categories
# The categories will be c0 to c(MLS_CATS-1).
MCS_CATS = 1024

# Set this to y to only display status messages
# during build.
QUIET = n
```

```
# Set this to treat warnings as errors.
WERROR = n
```

- Run *make install-src* to install source at policy build location.
- Change to the */etc/selinux/<SELINUXTYPE>/src/policy* directory where an unconfigured basic policy has been installed.
- Run *make conf* to build an initial *policy/booleans.conf* and *policy/modules.conf* files. For this simple configuration these files will not be edited.
    - This process will also build the *policy/modules/kernel/corenetwork.te* and *corenetwork.if* files if not already present. These would be based on the contents of *corenetwork.te.in* and *corenetwork.if.in* configuration files (for this simple configuration these files will not be edited).
- Run *make load* to build the policy, add the modules to the store and install the binary kernel policy plus its supporting configuration files.
- As the Reference Policy has NOT been tailored specifically for Fedora, it MUST be run in permissive mode.
- The policy should now be built and can be checked using tools such as ***apol****(8)* or loaded by editing the */etc/selinux/config* file, running '*touch /.autorelabel*' and rebooting the system.

Building the Fedora Policy

Note, the Fedora **selinux-policy** started life as **Reference Policy Version: 2.20130424** and, is now basically a large patch on top of the original Reference Policy.

Building Fedora policies by hand is complex as they use the *rpmbuild/SPECS/selinux-policy.spec* file, therefore this section will give an overview of how this can be achieved, the reader can then experiment (the spec file gives an insight). The build process assumes that an equivalent '*targeted*' policy will be built named '*targeted-FEDORA*'.

Note: The following steps were tested on Fedora 31 with no problems.

Install the source as follows:

```
rpm -Uvh selinux-policy-<version>.src.rpm
```

The *rpmbuild/SOURCES* directory contents that will be used to build a copy of the **targeted** policy are as follows (there are other files, however they are not required for this exercise):

selinux-policy-<commit_num>.tar.gz

- The Reference Policy version 2.20130424 with Fedora specific updates that should be unpacked into: *rpmbuild/SOURCES/selinux-policy-<commit_num>*

selinux-policy-contrib-<commit_num>.tar.gz

- The Reference Policy contribution modules. Unpack the files and install them into: *rpmbuild/SOURCES/selinux-policy-<commit_num>/policy/modules/contrib*

container-selinux.tgz

- Fedora Containers module. Unpack and then install into: *rpmbuild/SOURCES/selinux-policy-<commit_num>/policy/modules/contrib*

permissivedomains.cil

- Badly named file – just adds *system_r* to *roleattributeset*. Copy this to *rpmbuild/SOURCES/selinux-policy-<commit_num> as it will be installed by* semodule* once the policy has been built.

modules-targeted-base.conf and *modules-targeted-contrib.conf*

- Concatenate both files and copy this to become: *rpmbuild/SOURCES/selinux-policy-<commit_num>/policy/modules.conf*

booleans-targeted.conf

- Replace the *rpmbuild/SOURCES/selinux-policy-<commit_num>/policy/booleans.conf* file with this version.

securetty_types-targeted

- Replace the *rpmbuild/SOURCES/selinux-policy-<commit_num>/config/appconfig-mcs/securetty_types* file with this version.

users-targeted

- Replace the *rpmbuild/SOURCES/selinux-policy-<commit_num>/policy/users* file with this version.

The basic steps to build are:

- Edit the *rpmbuild/SOURCES/selinux-policy-<commit_num>build.conf* file to reflect the policy to be built (as this is an earlier version of the Reference Policy it has less options than the current Reference Policy):

```
########################################
# Policy build options
#
# Policy version
# By default, checkpolicy will create the highest version policy it supports.
# Setting this will override the version. This only has an effect for
# monolithic policies.
#OUTPUT_POLICY = 18

# Policy Type
# standard, mls, mcs. Note Red Hat builds the MCS Policy Type
# for their 'targeted' version.
TYPE = mcs

# Policy Name
# If set, this will be used as the policy name. Otherwise the policy type
# will be used for the name. This entry is also used by the 'make install-src'
# process to copy the source to the:
#    /etc/selinux/<SELINUXTYPE>/src/policy directory.
NAME = targeted-FEDORA

# Distribution
# Some distributions have portions of policy for programs or configurations
# specific to the distribution. Setting this will enable options for the
# distribution. redhat, gentoo, debian, suse, and rhel4 are current options.
# Fedora users should enable redhat.
DISTRO = redhat

# Unknown Permissions Handling
# The behaviour for handling permissions defined in the kernel but missing
# from the policy. The permissions can either be allowed, denied, or
# the policy loading can be rejected.
# allow, deny, and reject are current options. Fedora use allow for all
# policies except MLS that uses 'deny'.
```

```
UNK_PERMS = allow

# Direct admin init
# Setting this will allow sysadm to directly run init scripts, instead of
# requiring run_init. This is a build option, as role transitions do not
# work in conditional policy.
DIRECT_INITRC = n

# Build monolithic policy. Putting y here will build a monolithic
policy.
MONOLITHIC = n

# User-based access control (UBAC)
# Enable UBAC for role separations. Note Fedora disables UBAC.
UBAC = n

# Custom build options. This field enables custom build options. Putting
# foo here will enable build option blocks foo. Options should be
separated by spaces.
CUSTOM_BUILDOPT =

# Number of MLS Sensitivities
# The sensitivities will be s0 to s(MLS_SENS-1). Dominance will be in
# increasing numerical order with s0 being lowest.
MLS_SENS = 16

# Number of MLS Categories.
# The categories will be c0 to c(MLS_CATS-1).
MLS_CATS = 1024

# Number of MCS Categories
# The categories will be c0 to c(MLC_CATS-1).
MCS_CATS = 1024

# Set this to y to only display status messages during build.
QUIET = n
```

- From *rpmbuild/SOURCES/selinux-policy-<commit_num>* run *make conf* to initialise the build (it creates two important files in *./selinux-policy-<commit_num>/policy/modules/kernel* called *corenetwork.te* and *corenetwork.if*).
- From *rpmbuild/SOURCES/selinux-policy-<commit_num>* run *make install-src* to install source at policy build location.
- Change to the */etc/selinux/targeted-FEDORA/src/policy* directory where the policy has been installed.
- Run *make load* to build the policy, add the modules to the store and install the binary kernel policy plus its supporting configuration files.
    - Note that the policy store will default to */var/lib/selinux/targeted-FEDORA*, with the modules in *active/modules/400/<module_name>*, there will also be CIL versions of the modules.
- Install the *permissivedomains.cil* module as follows:
    - *semodule -s targeted-FEDORA -i permissivedomains.cil*
- The policy should now be built and can be checked using tools such as ***apol****(8)* or loaded by editing the */etc/selinux/config* file (setting to ***permissive*** mode for safety), running '*touch /.autorelabel*' and rebooting the system. It should have the same number of rules, types, classes etc. as the original release.

Reference Policy Headers

This method of building policy and adding new modules is used for distributions that do not require access to the source code.

Note that the Reference Policy header and the Fedora policy header installations are slightly different as described below.

Building and Installing the Header Files

To be able to fully build the policy headers from the reference policy source two steps are required:

1. Ensure the source is installed and configured as described in the **Installing and Building the Reference Policy Source** section. This is because the *make load* (or *make install*) command will package all the modules as defined in the *modules.conf* file, producing a *base.pp* and the relevant *.pp* packages. The build process will then install these in the */usr/share/selinux/<SELINUXTYPE>* directory.
2. Execute the *make install-headers* that will:
    - Produce a *build.conf* file that represents the contents of the master *build.conf* file and place it in the */usr/share/selinux/<SELINUXTYPE>/include* directory.
    - Produce the XML documentation set that reflects the source and place it in the */usr/share/selinux/<SELINUXTYPE>/include* directory.
    - Copy a development *Makefile* for building from policy headers to the */usr/share/selinux/<SELINUXTYPE>/include* directory.
    - Copy the support macros *.spt* files to the */usr/share/selinux/<SELINUXTYPE>/include/support* directory. This will also include an *all_perms.spt* file that will contain macros to allow all classes and permissions to be resolved.
    - Copy the module interface files (*.if*) to the relevant module directories at: */usr/share/selinux/<SELINUXTYPE>/include/modules*.

Using the Reference Policy Headers

Note that this section describes the standard Reference Policy headers, the Fedora installation is slightly different and described in the **Using Fedora Supplied Headers** section.

Once the headers are installed as defined above, new modules can be built in any local directory. An example set of module files are located in the reference policy source at */etc/selinux/<SELINUXTYPE>/src/policy/doc* and are called *example.te*, *example.if*, and *example.fc*.

During the header build process a *Makefile* was included in the headers directory. This *Makefile* can be used to build the example modules by using makes *-f* option as follows (assuming that the example module files are in the local directory):

```
make -f /usr/share/selinux/<NAME>/include/Makefile
```

However there is another *Makefile* (*./policy/doc Makefile.example*) that can be installed in the users home directory (*$HOME*) that will call the master *Makefile*:

```
AWK ?= gawk

NAME ?= $(shell $(AWK) -F= '/^SELINUXTYPE/{ print $$2 }' /etc/selinux/config)
SHAREDIR ?= /usr/share/selinux
HEADERDIR := $(SHAREDIR)/$(NAME)/include

include $(HEADERDIR)/Makefile
```

The **Header Policy Build Make Targets** shows the make targets for modules built from headers.

Header Policy Build Make Targets:

MODULENAME.pp

- Compile and package the *MODULENAME* local module.

all

- Compile and package the modules in the current directory.

load

- Compile and package the modules in the current directory, then insert them into the module store.

refresh

- Attempts to reinsert all modules that are currently in the module store from the local and system module packages.

xml

- Build a *policy.xml* from the XML included with the base policy headers and any XML in the modules in the current directory.

Using Fedora Supplied Headers

The Fedora distribution installs the headers in a slightly different manner as Fedora installs (via the selinux-policy-devel rpm):

- A *modules-base.lst* and *modules-contrib.lst* containing a list of installed modules under */usr/share/selinux/<NAME>*.
- The development header files are installed in the */usr/share/selinux/devel* directory. The example modules are also in this directory and the *Makefile* is also slightly different to that used by the Reference Policy source.
- The documentation is installed in the */usr/share/doc/selinux-policy/html* directory.

An example usage to build a policy module called *ipsec_test_policy.te* and insert this into the policy store is:

```
make -f /usr/share/selinux/devel/Makefile ipsec_test_policy.pp
semodule -i ipsec_test_policy.pp
```

Reference Policy Support Macros

This section explains some of the support macros used to build reference policy source modules. These macros are located at:

- *./policy/support* for the reference policy source.
- */usr/share/selinux/<NAME>/include/support* for Reference Policy installed header files.
- */usr/share/selinux/devel/support* for Fedora installed header files.

The following support macro file contents are explained:

- *loadable_module.spt* - Loadable module support.
- *misc_macros.spt* - Generate users, bools and security contexts.
- *mls_mcs_macros.spt* - MLS / MCS support.
- *file_patterns.spt* - Sets up allow rules via parameters for files and directories.

- *ipc_patterns.spt* - Sets up allow rules via parameters for Unix domain sockets.
- *misc_patterns.spt* - Domain and process transitions.
- *obj_perm_sets.spt* - Object classes and permissions.

When the header files are installed the *all_perms.spt* support macro file is also installed that describes all classes and permissions configured in the original source policy.

Support Macros described in this section:

loadable_module.spt

- *policy_module* Macro
  - For adding the *module* statement and mandatory *require* block entries.
- *gen_require* Macro
  - For use in interfaces to optionally insert a *require* block.
- *optional_policy* Macro
  - Optional policy handling.
- *gen_tunable* Macro
  - Tunable declaration.
- *tunable_policy* Macro
  - Tunable policy handling.
- *interface* Macro
  - Generate the access interface block.
- *template* Macro
  - Generate *template* interface block.

misc_macros.spt

- *gen_context* Macro
  - Generate a security context.
- *gen_user* Macro
  - Generate an SELinux user.
- *gen_bool* Macro
  - Generate a boolean.

mls_mcs_macros.spt

- *gen_cats* Macro
  - Declares categories c0 to c(N-1).
- *gen_sens* Macro
  - Declares sensitivities s0 to s(N-1) with dominance in increasing numeric order with s0 lowest, s(N-1) highest.
- *gen_levels* Macro
  - Generate levels from s0 to (N-1) with categories c0 to (M-1).

Notes:

1. The macro calls can be in any configuration file read by the build process and can be found in (for example) the *users*, *mls*, *mcs* and *constraints* files.
2. There are four main m4 *ifdef* parameters used within modules:
   - *enable_mcs* - this is used to test if the MCS policy is being built.
   - *enable_mls* - this is used to test if the MLS policy is being built.
   - *enable_ubac* - this enables the user based access control within the *constraints* file.
   - *hide_broken_symptoms* - this is used to hide errors in modules with *dontaudit* rules.

These are also mentioned as they are set by the initial build process with examples shown in the *ifdef* section.

1. The macro examples in this section have been taken from the reference policy module files and shown in each relevant "**Example Macro**" section. The macros are then expanded by the build process to form modules containing the policy language statements and rules in the *tmp* directory. These files have been extracted and modified for readability, then shown in each relevant "**Expanded Macro**" section.
2. An example policy that has had macros expanded is shown in the **Module Expansion Process** section.
3. Be aware that spaces between macro names and their parameters are not allowed:

Correct:

```
policy_module(ftp, 1.7.0)
```

Incorrect:

```
policy_module (ftp, 1.7.0)
```

Loadable Policy Macros

The loadable policy module support macros are located in the *loadable_module.spt* file.

policy_module **Macro**

This macro will add the ***module*** to a loadable module, and automatically add a ***require*** with pre-defined information for all loadable modules such as the *system_r* role, kernel classes and permissions, and optionally MCS / MLS information (*sensitivity* and *category* statements).

The macro definition is:

```
policy_module(module_name,version)
```

Where:

policy_module

- The *policy_module* macro keyword.

module_name

- The module identifier that must be unique in the module layers.

version_number

- The module version number in M.m.m format (where M = major version number and m = minor version numbers). Since release 2.20220106 the *version_number* argument is optional. If missing '1' is set as a default to satisfy the policy syntax.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	No	No

Example Macro:

```
# This example is from the modules/services/ftp.te module:
#

policy_module(ftp, 1.23.0)
```

Expanded Macro:

```
# This is the expanded macro from the tmp/ftp.tmp file:
#
module ftp 1.23.0;

require {
    role system_r;
    class security {compute_av compute_create .... };
    ....
    class capability2 (mac_override mac_admin };
    # If MLS or MCS configured then the:
    sensitivity s0;
    ....
    category c0;
    ....
}
```

***gen_require* Macro**

For use within module files to insert a *require* block.

The macro definition is:

```
gen_require(`require_statements')
```

Where:

gen_require

- The *gen_require* macro keyword.

require_statements

- These statements consist of those allowed in the policy language ***require*** Statement.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	Yes	No

Example Macro:

```
# This example is from the modules/services/cron.te module:
#
```

```
gen_require(`class passwd rootok;')
```

Expanded Macro:

```
# This is the expanded macro from the tmp/cron.tmp file:
#

require {
    class passwd rootok;
}
```

optional_policy Macro

For use within module files to insert an *optional* block that will be expanded by the build process only if the modules containing the access or template interface calls that follow are present. If one module is present and the other is not, then the optional statements are not included.

The macro definition is:

```
optional_policy(`optional_statements')
```

Where:

optional_policy

- The *optional_policy* macro keyword.

optional_statements

- These statements consist of those allowed in the policy language ***optional*** Statement.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	Yes	No

Example Macro:

```
# This example is from the modules/services/ftp.te module and
# shows the optional_policy macro with two levels.
#

optional_policy(`
    corecmd_exec_shell(ftpd_t)

    files_read_usr_files(ftpd_t)

    cron_system_entry(ftpd_t, ftpd_exec_t)

    optional_policy(`
        logrotate_exec(ftpd_t)
```

```
    ')
')
```

Expanded Macro:

```
# This is the expanded macro from the tmp/ftp.tmp file showing
# the policy language statements with both optional levels expanded.
#

optional {
    ##### begin corecmd_exec_shell(ftpd_t) depth: 1
    require {
        type shell_exec_t;
        } # end require

        ##### begin corecmd_list_bin(ftpd_t) depth: 2
        require {
            type bin_t;
            } # end require

            ##### begin corecmd_search_bin(ftpd_t) depth: 3
            require {
                type bin_t;
            } # end require
            allow ftpd_t bin_t:dir { getattr search open };
            allow ftpd_t bin_t:lnk_file { getattr read };

                ##### begin files_search_usr(ftpd_t) depth: 4
                require {
                    type usr_t;
                } # end require
                allow ftpd_t usr_t:dir { getattr search open };
            ##### end files_search_usr(ftpd_t) depth: 3
        ##### end corecmd_search_bin(ftpd_t) depth: 2
        allow ftpd_t bin_t:dir { getattr search open };
        allow ftpd_t bin_t:dir { getattr search open read lock ioctl };
    ##### end corecmd_list_bin(ftpd_t) depth: 1
    allow ftpd_t shell_exec_t:file { { getattr open map read execute ioctl } ioctl
lock execute_no_trans };
    ##### end corecmd_exec_shell(ftpd_t) depth: 0

    ##### begin files_read_usr_files(ftpd_t) depth: 1
    require {
        type usr_t;
    } # end require
    allow ftpd_t usr_t:dir { getattr search open read lock ioctl };
    allow ftpd_t usr_t:dir { getattr search open };
    allow ftpd_t usr_t:file { getattr open read lock ioctl };
    allow ftpd_t usr_t:dir { getattr search open };
    allow ftpd_t usr_t:lnk_file { getattr read };
    ##### end files_read_usr_files(ftpd_t) depth: 0

    ##### begin cron_system_entry(ftpd_t,ftpd_exec_t) depth: 1
    require {
```

```
        type crond_t, system_cronjob_t;
        type user_cron_spool_log_t;
    } # end require
    allow ftpd_t user_cron_spool_log_t:dir { getattr search open };
    allow ftpd_t user_cron_spool_log_t:file { open { getattr read write append ioctl
lock } };
    allow system_cronjob_t ftpd_exec_t:file { getattr open map read execute ioctl };
    allow system_cronjob_t ftpd_t:process transition;
    dontaudit system_cronjob_t ftpd_t:process { noatsecure siginh rlimitinh };
    type_transition system_cronjob_t ftpd_exec_t:process ftpd_t;
    allow ftpd_t system_cronjob_t:fd use;
    allow ftpd_t system_cronjob_t:fifo_file { getattr read write append ioctl lock };
    allow ftpd_t system_cronjob_t:process sigchld;
    allow crond_t ftpd_exec_t:file { getattr open map read execute ioctl };
    allow crond_t ftpd_t:process transition;
    dontaudit crond_t ftpd_t:process { noatsecure siginh rlimitinh };
    type_transition crond_t ftpd_exec_t:process ftpd_t;
    allow ftpd_t crond_t:fd use;
    allow ftpd_t crond_t:fifo_file { getattr read write append ioctl lock };
    allow ftpd_t crond_t:process sigchld;
    role system_r types ftpd_t;
    ##### end cron_system_entry(ftpd_t,ftpd_exec_t) depth: 0
    optional {
        ##### begin logrotate_exec(ftpd_t) depth: 1
        require {
            type logrotate_exec_t;
        } # end require

            ##### begin corecmd_search_bin(ftpd_t) depth: 2
            require {
                type bin_t;
            } # end require
            allow ftpd_t bin_t:dir { getattr search open };
            allow ftpd_t bin_t:lnk_file { getattr read };

                ##### begin files_search_usr(ftpd_t) depth: 3
                require {
                    type usr_t;
                } # end require
                allow ftpd_t usr_t:dir { getattr search open };
            ##### end files_search_usr(ftpd_t) depth: 2
        ##### end corecmd_search_bin(ftpd_t) depth: 1

        allow ftpd_t logrotate_exec_t:file { { getattr open map read execute ioctl }
ioctl lock execute_no_trans };

    ##### end logrotate_exec(ftpd_t) depth: 0
    }
} # end optional
```

***gen_tunable* Macro**

This macro defines booleans that are global in scope. The corresponding ***tunable_policy*** macro contains the supporting statements allowed or not depending on the value of the boolean. These entries are extracted as a part

of the build process (by the *make conf* target) and added to the *global_tunables* file where they can then be used to alter the default values for the *make load* or *make install* targets.

Note that the comments shown in the example MUST be present as they are used to describe the function and are extracted for the **documentation**.

The macro definition is:

```
gen_tunable(`boolean_name',boolean_value)
```

Where:

gen_tunable

- The *gen_tunable* macro keyword.

boolean_name

- The *boolean* identifier.

boolean_value

The *boolean* value that can be either *true* or *false*.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	Yes	No

Example Macro:

```
# This example is from the modules/services/ftp.te module:

## <desc>
##  <p>
##  Determine whether ftpd can use NFS
##  used for public file transfer services.
##  </p>
## </desc>
gen_tunable(allow_ftpd_use_nfs, false)
```

Expanded Macro:

```
# This is the expanded macro from the tmp/ftp.tmp file:
#

bool allow_ftpd_use_nfs false;
```

***tunable_policy* Macro**

This macro contains the statements allowed or not depending on the value of the boolean defined by the ***gen_tunable*** macro.

The macro definition is:

```
tunable_policy('gen_tunable_id',tunable_policy_rules`)
```

Where:

tunable_policy

- The *tunable_policy* macro keyword.

gen_tunable_id

- This is the boolean identifier defined by the *gen_tunable* macro. It is possible to have multiple entries separated by *&&* or || as shown in the example.

tunable_policy_rules

- These are the policy rules and statements as defined in the ***if*** statement policy language section.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	Yes	No

Example Macro:

```
# This example is from the modules/services/ftp.te module
# showing the use of the boolean with the && operator.
#

tunable_policy(`allow_ftpd_use_nfs && allow_ftpd_anon_write',`
    fs_manage_nfs_files(ftpd_t)
')
```

Expanded Macro:

```
# This is the expanded macro from the tmp/ftp.tmp file.
#

if (allow_ftpd_use_nfs && allow_ftpd_anon_write) {
    ##### begin fs_manage_nfs_files(ftpd_t)
    require {
        type nfs_t;
    } # end require
    allow ftpd_t nfs_t:dir { read getattr lock search ioctl add_name remove_name
write };
    allow ftpd_t nfs_t:file { create open getattr setattr read write append rename
link unlink ioctl lock };
} # end allow_ftpd_use_nfs && allow_ftpd_anon_write
```

***interface* Macro**

Access *interface* macros are defined in the interface module file (*.if*) and form the interface through which other modules can call on the modules services (as shown in and described in the **Module Expansion Process** section.

The macro definition is:

```
interface(`name',`interface_rules')
```

Where:

interface

- The *interface* macro keyword.

name

- The *interface* identifier that should be named to reflect the module identifier and its purpose.

interface_rules

- This can consist of the support macros, policy language statements or other *interface* calls as required to provide the service.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
No	Yes	No

Example Interface Definition:

Note that the comments shown in the example MUST be present as they are used to describe the function and are extracted for the **documentation**.

```
# This example is from the modules/services/ftp.if module
# showing the 'ftp_read_config' interface.
#

########################################
## <summary>
##  Read ftpd configuration files.
## </summary>
## <param name="domain">
##  <summary>
##  Domain allowed access.
##  </summary>
## </param>
#
interface(`ftp_read_config',`
    gen_require(`
        type ftpd_etc_t;
    ')

    files_search_etc($1)
    allow $1 ftpd_etc_t:file read_file_perms;
')
```

Expanded Macro: (taken from the *base.conf* file):

```
# Access Interfaces are only expanded at policy compile time
# if they are called by a module that requires their services.
#

# In this example the ftp_read_config interface is called from
# the init.te module via the optional_policy macro as shown
# below with the expanded code shown afterwards.

#
######## From ./policy/policy/modules/system/init.te ########
#

# optional_policy(`
#   ftp_read_config(initrc_t)
# ')

#
############ Expanded policy statements taken from init.tmp ##############
#
optional {
    ##### begin ftp_read_config(initrc_t) depth: 1
    require {
        type ftpd_etc_t;
    } # end require

    ##### begin files_search_etc(initrc_t) depth: 2
    require {
        type etc_t;
    } # end require

    allow initrc_t etc_t:dir { getattr search open };
    ##### end files_search_etc(initrc_t) depth: 1
    allow initrc_t ftpd_etc_t:file { getattr open read lock ioctl };
    ##### end ftp_read_config(initrc_t) depth: 0
} # end optional
```

***template* Macro**

A template interface is used to help create a domain and set up the appropriate rules and statements to run an application / process. The basic idea is to set up an application in a domain that is suitable for the defined SELinux user and role to access but not others. Should a different user / role need to access the same application, another domain would be allocated (these are known as ‘derived domains’ as the domain name is derived from caller information).

The main differences between an application interface and a template interface are:

- An access interface is called by other modules to perform a service.
- A template interface allows an application to be run in a domain based on user / role information to isolate different instances.

Note that the comments shown in the example MUST be present as they are used to describe the function and are extracted for the **documentation**.

The macro definition is:

```
template(`name',`template_rules')
```

Where:

template

- The *template* macro keyword.

name

- The *template* identifier that should be named to reflect the module identifier and its purpose. By convention the last component is *_template* (e.g. *ftp_per_role_template*).

template_rules

- This can consist of the support macros, policy language statements or *interface* calls as required to provide the service.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
No	Yes	No

Example Macro:

```
# This example is from the modules/services/djbdns.if module
# showing the 'djbdns_daemontools_domain_template' template interface.
#

######################################
## <summary>
##  The template to define a djbdns domain.
## </summary>
## <param name="domain_prefix">
##  <summary>
##  Domain prefix to be used.
##  </summary>
## </param>
#
template(`djbdns_daemontools_domain_template',`
    gen_require(`
        attribute djbdns_domain;
    ')

    ######################################
    #
    # Declarations
    #

    type djbdns_$1_t, djbdns_domain;
    type djbdns_$1_exec_t;
    domain_type(djbdns_$1_t)
    domain_entry_file(djbdns_$1_t, djbdns_$1_exec_t)
    role system_r types djbdns_$1_t;
```

```
    type djbdns_$1_conf_t;
    files_config_file(djbdns_$1_conf_t)

    ########################################
    #
    # Local policy
    #

    daemontools_service_domain(djbdns_$1_t, djbdns_$1_exec_t)
    daemontools_read_svc(djbdns_$1_t)

    allow djbdns_$1_t djbdns_$1_conf_t:dir list_dir_perms;
    allow djbdns_$1_t djbdns_$1_conf_t:file read_file_perms;
')
```

Expanded Macro:

```
# Template Interfaces are only expanded at policy compile time
# if they are called by a module that requires their services.
# This has been called from services/djbdns.te via:
#    djbdns_daemontools_domain_template(dnscache)
# and expanded in tmp/djbdns.tmp.
#

# Note these are very long, so only start / end shown. To trace these:
# 1) Note the line number where djbdns_daemontools_domain_template(dnscache)
#    is called in djbdns.te. In this case it is line 13.
# 2) In the appropriate tmp file where the macro has been expanded, in this
#    case tmp/djbdns.tmp, search for '#line 13'. These entries are all the
#    expanded components to build djbdns_daemontools_domain_template(dnscache).

##### begin djbdns_daemontools_domain_template(dnscache) depth: 1
    require {
        attribute djbdns_domain;
    } # end require

    ########################################
    #
    # Declarations
    #
    type djbdns_dnscache_t, djbdns_domain;
    type djbdns_dnscache_exec_t;
##### begin domain_type(djbdns_dnscache_t) depth: 2
    # start with basic domain
......
......
......
##### end files_search_var(djbdns_dnscache_t) depth: 2
    allow djbdns_dnscache_t svc_svc_t:dir { getattr search open read lock ioctl };
    allow djbdns_dnscache_t svc_svc_t:file { getattr open read lock ioctl };
##### end daemontools_read_svc(djbdns_dnscache_t) depth: 1
    allow djbdns_dnscache_t djbdns_dnscache_conf_t:dir { getattr search open read lock
ioctl };
```

```
    allow djbdns_dnscache_t djbdns_dnscache_conf_t:file { getattr open read lock
ioctl };
##### end djbdns_daemontools_domain_template(dnscache) depth: 0
```

Miscellaneous Macros

These macros are in the *misc_macros.spt* file.

gen_context Macro

This macro is used to generate a valid security context and can be used in any of the module files. Its most general use is in the *.fc* file where it is used to set the files security context.

The macro definition is:

```
gen_context(context[,mls,[mcs_categories]])
```

Where:

gen_context

- The *gen_context* macro keyword.

context

- The security context to be generated. This can include macros that are relevant to a context as shown in the example below.

mls

- MLS labels if MLS enabled in the policy.

mcs_categories

- MCS categories if MCS is enabled. MLS argument is required, but ignored.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	Yes	Yes

Example Macro (MCS):

```
# This example shows gen_context being used for a custom app

/var/lib/custom_app(/.*)?
gen_context(system_u:object_r:custom_app_t,mls_systemhigh,c9)
```

Expanded Macro (MCS):

```
# This is the expanded entry built into contexts/files/file_contexts:

/var/lib/custom_app(/.*)?  system_u:object_r:custom_app_t:s0:c9
```

Example Macro (MLS):

```
# This example shows gen_context being used to generate a
# security context in the kernel/storage.fc module:

/dev/\.tmp-block-.* -c
gen_context(system_u:object_r:fixed_disk_device_t,mls_systemhigh)
```

Expanded Macro (MLS):

```
# This is the expanded entry built into contexts/files/file_contexts:

/dev/\.tmp-block-.*  -c  system_u:object_r:fixed_disk_device_t:s15:c0.c1023
```

***gen_user* Macro**

This macro is used to generate a valid ***user*** Statement and add an entry in the ***users_extra*** configuration file if it exists.

The macro definition is:

```
gen_user(username, prefix, role_set, mls_defaultlevel, mls_range, [mcs_categories])
```

Where:

gen_user

- The *gen_user* macro keyword.

username

- The SELinux user id.

prefix

- SELinux users without the prefix will not be in the *users_extra* file. This is added to user directories by *genhomedircon* as discussed in the **Building the File Labeling Support Files** section.

role_set

- The user roles.

mls_defaultlevel

- The default level if MLS / MCS policy.

mls_range

- The range if MLS / MCS policy.

mcs_categories

The categories if MLS / MCS policy.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	No	No

Example Macro:

```
# This example has been taken from the policy/policy/users file:
#

ifdef(`direct_sysadm_daemon',*
    gen_user(root, sysadm, sysadm_r staff_r ifdef(`enable_mls',`secadm_r auditadm_r')
system_r, s0, s0 - mls_systemhigh, mcs_allcats)
',`
    gen_user(root, sysadm, sysadm_r staff_r ifdef(`enable_mls',`secadm_r auditadm_r'),
s0, s0 - mls_systemhigh, mcs_allcats)
')
```

Expanded Macro:

```
# The expanded gen_user macro from the base.conf for an MLS
# build. Note that the prefix is not present. This is added to
# the users_extra file as shown below.
#

user root roles { sysadm_r staff_r secadm_r auditadm_r } level s0 range s0 -
s15:c0.c1023;
```

```
# policy/tmp/users_extra file entry:
#

user root prefix sysadm;
```

gen_bool Macro

This macro defines a boolean and requires the following steps:

1. Declare the ***boolean*** in the ***global_booleans*** file.
2. Use the boolean in a module fileswith an ***if/else*** Statement as shown in the example.

Note that the comments shown in the example MUST be present as they are used to describe the function and are extracted for the **documentation**.

The macro definition is:

```
gen_bool(name,default_value)
```

Where:

gen_bool

- The *gen_bool* macro keyword.

name

- The *boolean* identifier.

default_value

- The value *true* or *false*.

The macro is only valid in the *global_booleans* file but the *boolean* declared can be used in the following module types:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
Yes	Yes	No

Example Macro:

```
# This example is from kernel/kernel.te  where the bool is declared.
# The comments must be present as it is used to generate the documentation.
#

## <desc>
## <p>
## Disable kernel module loading.
## </p>
## </desc>
gen_bool(secure_mode_insmod, false)
```

```
# Example usage from the kernel/kernel.te module:
#

if( ! secure_mode_insmod ) {
    allow can_load_kernmodule self:capability sys_module;
    allow can_load_kernmodule self:system module_load;

    files_load_kernel_modules(can_load_kernmodule)

    # load_module() calls stop_machine() which
    # calls sched_setscheduler()
    # gt: there seems to be no trace of the above, at
    # least in kernel versions greater than 2.6.37...
    allow can_load_kernmodule self:capability sys_nice;
    kernel_setsched(can_load_kernmodule)
}
```

```
# This example is from policy/booleans.conf where the bool set for policy.
#
# Disable kernel module loading.
#
secure_mode_insmod = false
```

Expanded Macro:

```
# This has been taken from the base.conf source file after
# expansion by the build process of the kernel.te module.
#

######################################
#
# Kernel module loading policy
#

if( ! secure_mode_insmod ) {
    allow can_load_kernmodule self:capability sys_module;
    allow can_load_kernmodule self:system module_load;
##### begin files_load_kernel_modules(can_load_kernmodule) depth: 1
##### begin files_read_kernel_modules(can_load_kernmodule) depth: 2
    allow can_load_kernmodule modules_object_t:dir { getattr search open read lock
ioctl };
    allow can_load_kernmodule modules_object_t:dir { getattr search open };
    allow can_load_kernmodule modules_object_t:file { getattr open read lock ioctl };
    allow can_load_kernmodule modules_object_t:dir { getattr search open };
    allow can_load_kernmodule modules_object_t:lnk_file { getattr read };
##### end files_read_kernel_modules(can_load_kernmodule) depth: 1
    allow can_load_kernmodule modules_object_t:system module_load;
##### end files_load_kernel_modules(can_load_kernmodule) depth: 0
    # load_module() calls stop_machine() which
    # calls sched_setscheduler()
    # gt: there seems to be no trace of the above, at
    # least in kernel versions greater than 2.6.37...
    allow can_load_kernmodule self:capability sys_nice;
    allow can_load_kernmodule kernel_t:process setsched;
##### end kernel_setsched(can_load_kernmodule) depth: 0
}
```

MLS and MCS Macros

These macros are in the *mls_mcs_macros.spt* file.

gen_cats Macro

This macro will generate a ***category*** statement for each category defined. These are then used in the *base.conf* / *policy.conf* source file and also inserted into each module by the ***policy_module***. The *policy/policy/mcs* and *mls* configuration files are the only files that contain this macro in the current reference policy.

The macro definition is:

```
gen_cats(mcs_num_cats | mls_num_cats)
```

Where:

gen_cats

- The *gen_cats* macro keyword.

mcs_num_cats

mls_num_cats

- These are the maximum number of categories that have been extracted from the *build.conf* file *MCS_CATS* or *MLS_CATS* entries and set as m4 parameters.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
na	na	na

Example Macro:

```
# This example is from the policy/policy/mls configuration file.
#

gen_cats(mls_num_cats)
```

Expanded Macro:

```
# This example has been extracted from the base.conf source file.

category c0;
category c1;
...
category c1023;
```

***gen_sens* Macro**

This macro will generate a ***sensitivity*** for each sensitivity defined. These are then used in the *base.conf* / *policy.conf* source file and also inserted into each module by the ***policy_module***. The *policy/policy/mcs* and *mls* configuration files are the only files that contain this macro in the current reference policy (note that the *mcs* file has *gen_sens(1)* as only one sensitivity is required).

The macro definition is:

```
gen_sens(mls_num_sens)
```

Where:

gen_sens

- The *gen_sens* macro keyword.

mls_num_sens

- These are the maximum number of sensitivities that have been extracted from the *build.conf* file *MLS_SENS* entries and set as an m4 parameter.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
na	na	na

Example Macro:

```
# This example is from the policy/policy/mls configuration file.
#

gen_cats(mls_num_sens)
```

Expanded Macro:

```
# This example has been extracted from the base.conf source file.

sensitivity s0;
sensitivity s1;
...
sensitivity s15;
```

gen_levels Macro

This macro will generate a *level* for each level defined. These are then used in the *base.conf* / *policy.conf* source file. The *policy/policy/mcs* and *mls* configuration files are the only files that contain this macro in the current reference policy.

The macro definition is:

```
gen_levels(mls_num_sens,mls_num_cats)
```

Where:

gen_levels

- The *gen_levels* macro keyword.

mls_num_sens

- This is the parameter that defines the number of sensitivities to generate. The MCS policy is set to '*1*'.

mls_num_cats

mcs_num_cats

- This is the parameter that defines the number of categories to generate.

The macro is valid in:

Private Policy File (*.te*)	External Interface File (*.if*)	File Labeling Policy File (*.fc*)
na	na	na

Example Macro:

```
# This example is from the policy/policy/mls configuration file.
#
```

```
gen_levels(mls_num_sens,mls_num_cats)
```

Expanded Macro:

```
# This example has been extracted from the base.conf source file.
# Note that the all categories are allocated to each sensitivity.

level s0:c0.c1023;
level s1:c0.c1023;
...
level s15:c0.c1023;
```

System High/Low Parameters

These macros define system high etc. as shown.

```
mls_systemlow
```

```
# gives:

s0
```

```
mls_systemhigh
```

```
# gives:

s15:c0.c1023
```

```
mcs_systemlow
```

```
# gives:

s0
```

```
mcs_systemhigh
```

```
# gives:

s0:c0.c1023
```

```
mcs_allcats
```

```
# gives:

c0.c1023
```

ifdef / *ifndef* Parameters

This section contains examples of the common *ifdef* / *ifndef* parameters that can be used in module source files.

hide_broken_symptoms

This is used within modules as shown in the example. The parameter is set up by the *Makefile* at the start of the build process.

Example Macro:

```
# This example is from the modules/kernel/domain.te module.
#

ifdef(`hide_broken_symptoms',`
    # This check is in the general socket
    # listen code, before protocol-specific
    # listen function is called, so bad calls
    # to listen on UDP sockets should be silenced
    dontaudit domain self:udp_socket listen;
')
```

enable_mls and *enable_mcs*

These are used within modules as shown in the example. The parameters are set up by the *Makefile* with information taken from the *build.conf* file at the start of the build process.

Example Macros:

```
# This example is from the modules/kernel/kernel.te module.
#

ifdef(`enable_mls',`
    role secadm_r;
    role auditadm_r;
')
```

```
# This example is from the modules/services/ftp.te module.
#

ifdef(`enable_mcs',`
    init_ranged_daemon_domain(ftpd_t, ftpd_exec_t, s0 - mcs_systemhigh)
')
```

enable_ubac

This is used within the *./policy/constraints* configuration file to set up various attributes to support user based access control (UBAC). These attributes are then used within the various modules that want to support UBAC. This support was added in version 2 of the Reference Policy.

The parameter is set up by the *Makefile* with information taken from the *build.conf* file at the start of the build process (*ubac = y | ubac = n*).

Example Macro:

```
# This example is from the policy/constraints file.
# Note that the ubac_constrained_type attribute is defined in
# modules/kernel/ubac.te module.

define(`basic_ubac_conditions',`
    ifdef(`enable_ubac',`
        u1 == u2
        or u1 == system_u
        or u2 == system_u
        or t1 != ubac_constrained_type
        or t2 != ubac_constrained_type
    ')
')
```

direct_sysadm_daemon

This is used within modules as shown in the example. The parameter is set up by the *Makefile* with information taken from the *build.conf* file at the start of the build process (if *DIRECT_INITRC = y*).

Example Macros:

```
# This example is from the modules/system/selinuxutil.te module.
#

ifndef(`direct_sysadm_daemon',`
    ifdef(`distro_gentoo',`
        # Gentoo integrated run_init:
        init_script_file_entry_type(run_init_t)

        init_exec_rc(run_init_t)
    ')
')
```

Module Expansion Process

The objective of this section is to show how the modules are expanded by the reference policy build process to form files that can then be compiled and then loaded into the policy store by using the *make MODULENAME.pp* target.

The files shown are those produced by the build process using the ada policy modules from the Reference Policy source tree (*ada.te*, *ada.if* and *ada.fc*) that are shown in the **Reference Policy Module Files** section.

The initial build process will build the source text files in the *policy/tmp* directory as *ada.tmp* and *ada.mod.fc* (that are basically build equivalent *ada.conf* and *ada.fc* formatted files). The basic steps are shown in , and the resulting expanded code shown in and then described in the **Module Expansion Process** section.

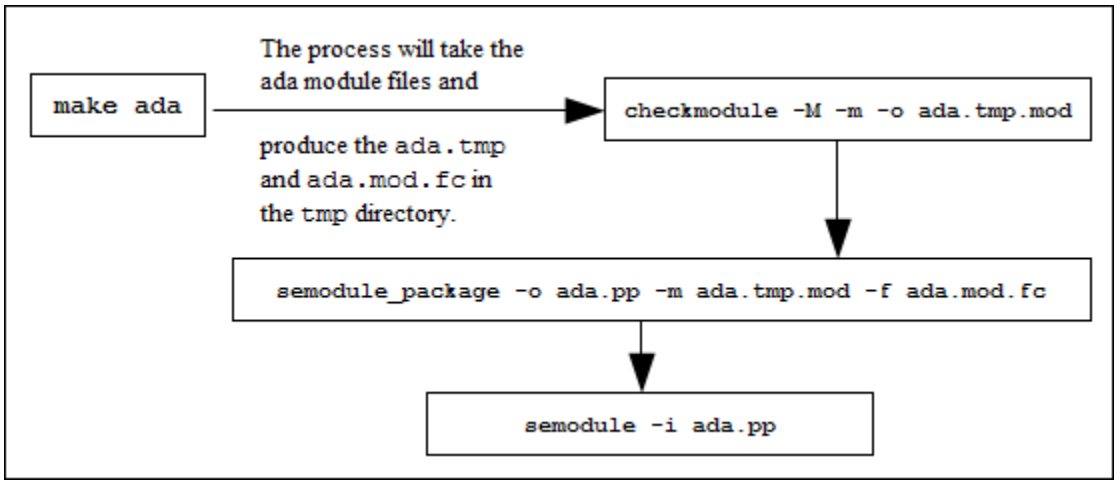


Figure 28: The *make ada* sequence of events

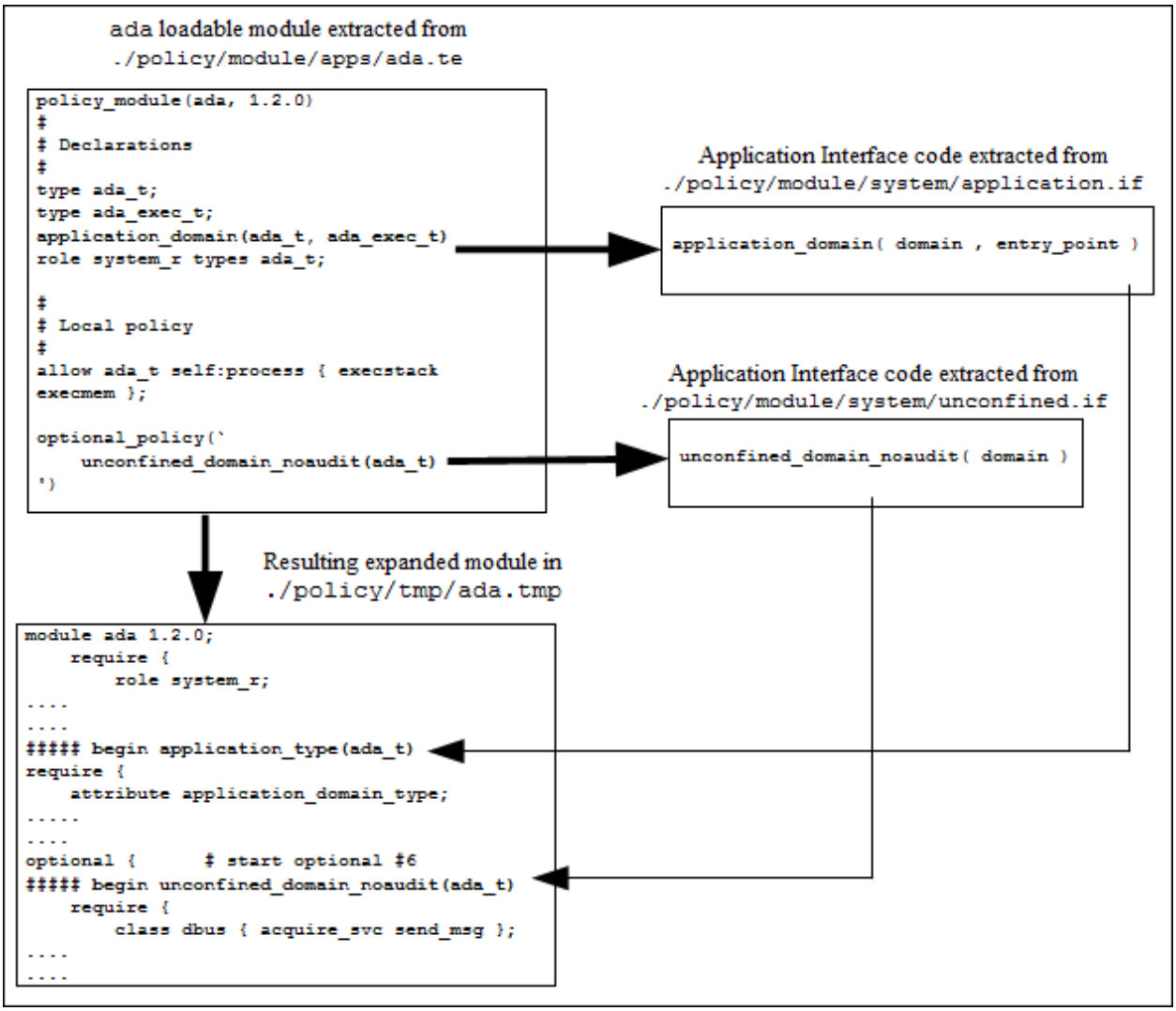


Figure 29: The expansion process

Hardening SELinux

The Reference Policy sets a good basis for learning how to operate a system with SELinux. The policy allows system administrators and users to continue using working methods that they learned before becoming familiar with SELinux, because of its “targeted” model and reasonable defaults.

However, it’s possible to tighten the policy considerably. The Reference Policy gives several options for hardening but for maximum control over the policy, it’s possible to write custom modules or even replace the Reference Policy entirely.

The hardening suggestions are listed in the rough order of difficulty.

Tuning Booleans

The Reference Policy uses booleans to control optional aspects of the policy. Changing the booleans is a very easy way to tune the SELinux policy. The booleans can be also changed temporarily, without committing them yet to the on-disk policy, so changes are automatically reverted on next boot.

For example, recent Firefox browsers can work with policy boolean `mozilla_execstack` set to `off`. This can reduce the surface to vulnerabilities which could use an executable stack. The boolean can be changed using ***setsebool***(*8*):

```
# Check current status:
getsebool mozilla_execstack
mozilla_execstack --> on

# Change temporarily:
setsebool mozilla_execstack=off

# Write to persistent policy on disk:
setsebool -P mozilla_execstack=off
```

Disabling Modules

By default, the Reference Policy enables most policy modules to support any system. But when the system is known well by the administrator, it’s possible to disable modules which aren’t used.

As an example, if Firefox isn’t installed, the module `mozilla` can be disabled in the policy. The hardening effect comes from reducing the allow rules, some of which may apply to paths which aren’t actively monitored because the corresponding application isn’t installed.

Care should be taken to never disable modules which actually are used, since this may weaken the policy instead, because the disabled module could have contained an application more strictly than what the policy allows without the module. Disabling critical modules can also result in system breakage. It’s also possible to remove modules, but on package upgrade they are typically reinstalled.

Examples:

```
# Disable Firefox module
semodule --disable=mozilla
# Remove the module entirely
semodule --remove=mozilla
```

Users and Roles

In the “targeted” model, both unprivileged users and the system administrator (`root`) are unconfined and the TE rules are very relaxed. However, it’s possible to change the SELinux user for these accounts to more confined variants.

For the unprivileged users, the confined user in the Reference Policy is `user_u` with corresponding role `user_r`. There’s also `staff_u` and `staff_r` to allow transitioning to system administrative roles by logging in as `staff_u:staff_r` and escalating to `staff_u:sysadm_r` or another role for administrative tasks with ***newrole**(1)*.

For the system administrator there are several options: `root` SELinux user, which is mostly unconfined and `sysadm_u`, which is more confined. The role for both is `sysadm_r`.

It’s also possible to divide the powers of the system administrator to several roles, `sysadm_r`, `auditadm_r`, `logadm_r`, `dbadm_r` and `secadm_r`. This can be useful when an organization wants to ensure that even the system administrators can be held accountable for their actions. The roles can be also useful for management of labor, for example having dedicated persons for managing databases or SELinux policies.

It should be noted that since this isn’t the default way of operating SELinux, the Reference Policy may need to be supplemented and the administrators, even users, may need to be more aware of SELinux in order to be able to operate the system.

Example:

```
# User `test` has been added earlier with `adduser` or `useradd`.
semanage login --add --seuser user_u --range 's0' test
```

Network Controls

With network controls of SELinux, it’s possible to enhance firewall rules with knowledge of SELinux types. Traditional firewall rules only affect the whole system by allowing certain ports and protocols but blocking others. With `nftables` and `iptables` it’s also possible to make this more fine grained: certain users can access the network but others may not. By also using SELinux controls it’s possible to fine tune this to the application level: `mozilla_t` can connect to the Internet but some other applications can’t. SELinux packet controls can be also used to combine Deep Packet Inspection with SELinux TE rules.

Chapter Networking Support presents the controls with examples.

Custom Policy Modules

Further hardening can be achieved by replacing policy modules from the Reference Policy with custom modules. Typically the modules in the Reference Policy are written to allow all possible modes of operation for an application or its users, since the writers of the policy don’t know the specifics of each installation. Thus the SELinux rules may be more relaxed than what could be optimal for a specific case. When the exact environment and usage patterns are known, it’s possible to write replacement policy modules to remove excess rules and hence reduce attack surface.

As a minimum, it should be ensured that all continuously running services and main user applications have a dedicated policy module or rules, instead of running in for example `init_t`, `initrc_t` or `unconfined_t` types which may offer low level of protection.

Fine Grained Network Controls

In an internal network of an organization, where all entities can agree on the same SELinux policy, using IPSec, CIPSO and CALIPSO may allow further policy controls. In addition to SELinux domain of the source application, even the SELinux domain (or at least MCS/MLS category/sensitivity) of the target server can be used in TE rules.

Fully Custom Policy

It's also possible to write custom SELinux policies for an entire system with non-trivial effort.

The rules can also be analyzed with various SELinux tools, such as `apol`, `sedta`, `seinfo`, `sepolicy`, `sesearch` and many more. With the tools it may be possible to find hardening opportunities or errors in the policy.

```
# Find out which domains can transition to custom domain
# `my_thunderbird_t` and what are the rules affecting the transition:
sedta --source my_thunderbird_t --reverse
Transition 1: user_t -> my_thunderbird_t

Domain transition rule(s):
allow user_t my_app_ta:process { sigkill signal signull transition };

Entrypoint my_thunderbird_exec_t:
        Domain entrypoint rule(s):
        allow my_thunderbird_t my_thunderbird_exec_t:file { entrypoint execute getattr
map read };

        File execute rule(s):
        allow user_t my_app_exec_ta:file { execute getattr open read };

        Type transition rule(s):
        type_transition user_t my_thunderbird_exec_t:process my_thunderbird_t;


1 domain transition(s) found.
```

```
# Check ports to which domain `mozilla_t` can connect:
sepolicy network -d mozilla_t

mozilla_t: tcp name_connect
        443, 80 (my_http_port_t)
        1080 (my_socks_port_t)
```

The downside of making the SELinux rules as tight as possible is that when the applications (or hardware components or network configuration) are updated, there's a possibility that the rules may also need updating because of the changes. Less generic rules are also less generally useful for different configurations, so the rules may need tuning for each installation.

Implementing SELinux-aware Applications

The following definitions attempt to explain the difference between the two types of userspace SELinux application (however the distinction can get 'blurred'):

SELinux-aware - Any application that provides support for SELinux. This generally means that the application makes use of SELinux libraries and/or other SELinux applications. Example SELinux-aware applications are the Pluggable Authentication Manager (***PAM**(8)*) and SELinux commands such as ***runcon**(1)*. It is of course possible to class an object manager as an SELinux-aware application.

Object Manager - Object Managers are a specialised form of SELinux-aware application that are responsible for the labeling, management and enforcement[21] of the objects under their control.

Generally the userspace Object Manager forms part of an application that can be configured out should the base Linux OS not support SELinux.

Example userspace Object Managers are:

- X-SELinux is an optional X-Windows extension responsible for labeling and enforcement of X-Windows objects.
- Dbus has an optional Object Manager built if SELinux is defined in the Linux build. This is responsible for the labeling and enforcement of Dbus objects.
- SE-PostgreSQL is an optional extension for PostgreSQL that is responsible for the labeling and enforcement of PostgreSQL database and supporting objects.

Therefore the basic distinction is that Object Managers manage their defined objects on behalf of an application, whereas general SELinux-aware applications do not (they rely on 'Object Managers' to do this e.g. the kernel based Object Managers such as those that manage filesystem, IPC and network labeling).

Implementing SELinux-aware Applications

This section puts forward various points that may be useful when developing SELinux-aware applications and object managers using *libselinux*.

1. Determine the security objectives and requirements.
2. Because these applications manage labeling and access control, they need to be trusted.
3. Where possible use the libselinux **_raw* functions as they avoid the overhead of translating the context to/from the readable format (unless of course there is a requirement for a readable context - see ***mcstransd**(8)*).
4. Use ***selinux_set_mapping**(3)* to limit the classes and permissions to only those required by the application.
5. The standard output for messages generated by *libselinux* functions is *stderr*. Use ***selinux_set_callback**(3)* with ***SELINUX_CB_LOG** type* to redirect these to a log handler.
6. Do not directly reference SELinux configuration files, always use the *libselinux* path functions to return the location. This will help portability as SELinux has some changes in the pipe-line for the location of the policy configuration files and the SELinux filesystem.
7. Use the ***selabel_*(3)*** functions to determine a files default context as the ***matchpathcon*(3)*** series of functions are deprecated - see ***selabel_file**(5)*.
8. Do not use class IDs directly, use ***string_to_security_class**(3)* that will take the class string defined in the policy and return the class ID/value. Always check the value is > 0. If *0*, then signifies that the class is

unknown and the **deny_unknown** flag setting in the policy will determine the outcome of any decision - see ***security_deny_unknown**(3)*.

9. Do not use permission bits directly, use ***string_to_av_perm**(3)* that will take the permission string defined in the policy and return the permission bit mask.
10. Where performance is important when making policy decisions (i.e. using ***security_compute_av**(3)*, ***security_compute_av_flags**(3)*, ***avc_has_perm**(3)* or ***avc_has_perm_noaudit**(3)*), then use the ***selinux_status_*(3)*** functions to detect policy updates etc. as these do not require system call over-heads once set up. Note that the *selinux_status_** functions are only available from *libselinux* 2.0.99, with Linux kernel 2.6.37 and above.
11. There are changes to the way contexts are computed for sockets in kernels 2.6.39 and above as described in the **Computing Security Contexts** section. The functions affected by this are: ***avc_compute_create**(3)*, ***avc_compute_member**(3)*, ***security_compute_create**(3)*, ***security_compute_member**(3)* and ***security_compute_relabel**(3)*.
12. It is possible to set an undefined context if the process has ***capability**(7)* ***CAP_MAC_ADMIN*** and class ***capability2*** with ***mac_admin*** permission in the policy. This is called 'deferred mapping of security contexts' and is explained at: http://git.kernel.org/?p=linux/kernel/git/torvalds/linux-2.6.git;a=commit;h=12b29f34558b9b45a2c6eabd4f3c6be939a3980f

Implementing Object Managers

To implement object managers for applications, an understanding of the application is essential, because as a minimum:

- What object types and their permissions are required.
- Where in the code object instances are created.
- Where access controls need to be applied.

While this section cannot help with those points, here are some notes to help during the design phase:

1. Determine what objects are required and the access controls (permissions) that need to be applied.
2. Does SELinux already have some of these object classes and permissions defined. For standard Linux OS objects such as files, then these would be available. If so, the object manager should remap them with ***selinux_set_mapping**(3)* so only those required are available.

However, do not try to reuse a current object that may be similar to the requirements, it will cause confusion at some stage. Always generate new classes/permissions.

1. If the application has APIs or functions that integrate with other applications or scripts, then as part of the object manager implementation these may need to support the use of security contexts (examples are X-Windows and SE-PostgreSQL that provide functions for other applications to use). Therefore if required, provide common functions that can be used to label the objects.
2. Determine how the initial objects will be labeled. For example will a configuration file be required for default labels, if so how will this be introduced into the SELinux userspace build. Examples of these are the X-Windows (***selabel_x**(5)*), SE-PostgreSQL (***selabel_db**(3)*), and file context series of files (***selabel_file**(5)*).
3. Will the labeling need to be persistent across policy and system reloads or not. X-Windows is an example of a non-persistent, and SE-PostgreSQL is an example of a persistent object manager.
4. Will support for the standard audit log or its own be required (the *libselinux* functions default to *stderr*). Use ***selinux_set_callback**(3)* to manage logging services.
5. Decide whether an AVC cache is required or not. If the object manager handles high volumes of requests then an AVC will be required.
6. Will the object manager need to do additional processing when policy or enforcement changes are detected. This could be clearing any caches or resetting variables etc.. If so, then ***selinux_set_callback**(3)* will be used to set up these functions. These events are detected via the ***netlink**(7)* services, see ***avc_open**(3)* and ***avc_netlink_open**(3)* for the various options available.
7. If possible implement a service like XACE for the application, and use it to interface with the applications SELinux object manager. The XACE interface acts like the LSM which supports SELinux as well as other

providers such as SMACK. The XACE interface is defined in the **X Access Control Extension Specification**, and for reference, the SE-PostgreSQL service also implements a similar interface.

Reference Policy Changes

When adding a new object manager to SELinux, it will require at least a new policy module to be added. This section assumes that the SELinux Reference Policy is in use and gives some pointers. The latest Reference Policy source can be obtained from: **https://github.com/SELinuxProject/refpolicy**

The main points to note when adding to the Reference Policy are:

1. Create sample Reference Policy policy modules (**.te*, **.if* and **.fc* module files) that provide rules for managing the new objects as described in the **The Reference Policy** section.
    - The SE-PostgreSQL modules provide an example, see the *./refpolicy/policy/modules/services/postgresql.* files in the Reference Policy source.
2. Create any new policy classes and permissions for the Reference Policy, these will need to be built into the base module as described in the Adding New Object Classes and Permissions section.

Note, that if no new object classes, permissions or constraints are being added to the policy, then the Reference Policy source code does not require modification, and supplying the module files (**.te*, **.if* and **.fc*) should suffice.

1. Create any constraints required as these need to be built into the base module of the Reference Policy. They are added to the *./refpolicy/policy/constraints*, *mcs* and *mls* files. Again the SE-PostgreSQL entries in these files give examples (find the *db_** class entries).
2. Create any SELinux configuration files (context, user etc.) that need to be added to the policy at build time.
3. Either produce an updated Reference Policy source or module patch, depending on whether new classes/constraints have been added. Note that by default a new module will be generated as a 'module', if it is required that the module is in the base (unusual), then add an entry ***<required val="true">*** to the start of the interface file as shown below:

```
# <summary>
##Comment regarding interface file

## </summary>
## <required val="true">
##Comment on reason why required in base
## </required>
```

Adding New Object Classes and Permissions

Because userspace object managers do not require their new classes and permissions to be built into the kernel, the configuration is limited to the actual policy (generally the Reference Policy) and the application object manager code. New classes are added to the Reference Policy *security_classes* file and permissions to the *access_vectors* file.

The class configuration file is at:

./refpolicy/policy/flask/security_classes

and each entry must be added to the end of the file in the following format:

```
class object_name   # userspace
```

Where ***class*** is the class keyword and *object_name* is the name of the object. The *# userspace* is used by build scripts to detect userspace objects.

The permissions configuration file is at:

./refpolicy/policy/flask/access_vectors

and each entry must be added to the end of the file in the following format:

```
class object_name
{
    perm_name
    ....
}
```

Where class is the *class* keyword, *object_name* is the name of the object and *perm_name* is the name given to each permission in the class (there is a limit of 32 permissions within a class). It is possible to have a common permission section within this file, see the file object entry in the *access_vectors* file for an example.

The same principle applies to adding new class/permissions to Android although the flask files are located in the *system/sepolicy/private* directory.

Note that CIL policies do not use flask files and class/permissions must be declared using the *class*, *classpermission*, and *classorder* statements.

For reference, http://selinuxproject.org/page/Adding_New_Permissions describes how new kernel object classes and permissions are added to the system and is summarised as follows for kernels >= 2.6.33 that use dynamic class/perm discovery:

1. Edit *security/selinux/include/classmap.h* in the kernel tree and add the required definition. This will define the class and/or permission for use in the kernel; the corresponding symbol definitions will be automatically generated during the kernel build. If not defined in the policy, then the class and/or permission will be handled in accordance with the policy's *handle_unknown* definition, which can be reject (refuse to load the policy), deny (deny the undefined class/permission), or allow (allow the undefined class/permission). *handle_unknown* is set to allow in Fedora policies.
2. Edit *refpolicy/policy/flask/security_classes* and/or *access_vectors* in the refpolicy tree and add your definition. This will define the class and permission for use in the policy. These are generally added to the class and/or permission at the end of the existing list of classes or permissions for that class for backward compatibility with older kernels. The class and/or permission definition in policy need not line up with the definition in the kernel's classmap, as the values will be dynamically mapped by the kernel. Then add allow rules as appropriate to the policy for the new permissions.

Embedded Systems

This section lists some of the general decisions to be taken when implementing SELinux on embedded systems, it is by no means complete.

Two embedded SELinux projects are used as examples (OpenWrt and Android) with the main emphasis on policy development as this is considered the most difficult area. The major difference between OpenWrt and Android is that SELinux is not tightly integrated in OpenWrt, therefore MAC is addressed in policy rather than also adding additional SELinux-awareness to services as in Android[22].

An alternative MAC service to consider is **Smack** (Simplified Mandatory Access Control Kernel) as used in the Samsung **Tizen** and **Automotive Grade Linux** projects. Smack can have a smaller, less complex footprint than SELinux.

References

These papers on embedded systems can be used as references, however they are old (2007 - 2015):

- **Security Enhanced (SE) Android: Bringing Flexible MAC to Android** from http://www.cs.columbia.edu/~lierranli/coms6998-7Spring2014/papers/SEAndroid-NDSS2013.pdf describes the initial Android changes.
- **Reducing Resource Consumption of SELinux for Embedded Systems with Contributions to Open-Source Ecosystems** from https://www.jstage.jst.go.jp/article/ipsjjip/23/5/23_664/_article describes a scenario where *libselinux* was modified and *libsepol* removed for their embedded system (however no links to their final modified code, although there are many threads on the https://lore.kernel.org/selinux/ list discussing these changes). It should be noted that these libraries have changed since the original article, therefore it should be used as a reference for ideas only. They also used a now obsolete policy editor ***seedit*** to modify Reference Policies.
- **Using SELinux security enforcement in Linux-based embedded devices** from https://eudl.eu/doi/10.4108/icst.mobilware2008.2927 describes enabling SELinux on a Nokia 770 Internet Tablet.
- **Filesystem considerations for embedded devices** from https://events.static.linuxfound.org/sites/events/files/slides/fs-for-embedded-full_0.pdf discusses various embedded filesystems performance and reliability.

General Requirements

Note 1 - This section discusses the Reference Policy 'Monolithic' and 'Modular' policy builds, however this can be confusing, so to clarify:

- The Reference Policy builds both 'Monolithic' and 'Modular' policy using policy modules defined in a *modules.conf* file.

- The 'Monolithic' build process builds the final policy using ***checkpolicy**(8)* and therefore does NOT make use of the ***semanage**(8)* services to modify policy during runtime.
- The 'Modular' build process builds the final policy using ***semodule**(8)* and therefore CAN make use of the ***semanage**(8)* services to modify policy during runtime. This requires additional resources as it makes use of the 'policy store[23]' as described in the **SELinux Configuration Files - The Policy Store** and **Policy Store Configuration Files** sections. To be clear, it is possible to build a 'Modular' policy on the host system, then install the resulting **Policy Configuration Files** onto the target system (i.e. no 'policy store' on the target system).
- Also note that the Reference Policy 'Monolithic' and 'Modular' builds do not build the exact same list of policy configuration files.

Note 2 - If the requirement is to build the policy in CIL, it is possible to emulate the above by:

- Building policy using ***secilc**(8)* will build a 'Monolithic' policy.
- Building policy using ***semodule**(8)* will build a 'Modular' policy. This can then make use of the ***semanage**(8)* services to modify policy during runtime as it makes use of the 'policy store[24]'.
- A useful feature of CIL is that statements can be defined to generate the ***file_contexts**(5)* entries in a consistent manner.

Note 3 - Is there a requirement to build/rebuild policy on the target, if so does it also need to be managed during runtime:

- If build/rebuild policy on the target with NO semanage support, then only ***checkpolicy**(8)* or ***secilc**(8)* will be required on target.
- If building on the target with runtime changes then ***semodule**(8)* and ***semanage**(8)* are required.
- If no requirement to build policy on the target, then these are not needed.

Note 4 - Do any of the target filesystems support extended attributes (***xattr**(7)*), if so then ***restorecon**(8)* or ***setfiles**(8)* may be required on the target to label files (see the **Labeling Files** section).

Project Repositories

The current SELinux userspace source can be obtained from https://github.com/SELinuxProject/selinux and the current stable releases from https://github.com/SELinuxProject/selinux/releases.

The current Reference Policy source can be obtained from https://github.com/SELinuxProject/refpolicy and the current stable releases from https://github.com/SELinuxProject/refpolicy/releases.

The current SETools (***apol**(1)* etc.) source can be obtained from https://github.com/SELinuxProject/setools and the current stable releases from https://github.com/SELinuxProject/setools/releases.

Project Requirements

The project requirements will determine the following:

- Kernel Version
  - The kernel version will determine the maximum policy version supported. The **Policy Versions** section details the policy versions, their supported features and SELinux library requirements.
- Support ***xattr**(7)* Filesystems
  - If extended attribute filesystems are used then depending on how the target is built/loaded it will require ***restorecon**(8)* or ***setfiles**(8)* to label these file systems. The policy will also require a *file_contexts* that is used to provide the labels.
- Multi-User
  - Generally only one user and user role are required, this is the case for OpenWrt and Android. If multi-user then PAM services may be required.

- Support Tools
  - These would generally be either **BusyBox** (OpenWrt) or **Toybox** (Android). Both of these can be built with SELinux enabled utilities.
- Embedded Filesystems
  - The https://elinux.org/File_Systems#Embedded_Filesystems and **Filesystem considerations for embedded devices** discuss suitable embedded filesystems. If extended attribute (***xattr****(7)*) filesystems are required, then a policy will require a supporting ***file_contexts****(5)* file and the ***restorecon****(8)* utility to label the filesystem.
- SELinux Policy Support:
  - Use the Reference Policy, bespoke CIL policy or bespoke policy using ***m4****(1)* macros as used by Android (if starting with a bespoke policy then CIL is recommended). Also need to consider:
    - If using the Reference Policy on the target device use either:
      - Monolithic Policy - Use this for minimum resource usage. Also the policy is not so easy to update such items as network port and interface definitions (may need to push a new version to the device).
      - Modular Policy - Only use this if there is a requirement to modify the device policy during runtime.
    - Is MCS/MLS Support is required. The **MLS or MCS Policy** section gives a brief introduction. The OpenWrt Project does not use MLS/MCS policy, however Android does use MCS for application sandboxing as shown in the **SE Android - Computing Process Context Examples** section.
    - Is Conditional Policy (***booleans****(8)*) support required. This allows different policy rules to be enabled/disabled at runtime (Android and OpenWrt do not support Booleans).
    - SELinux 'user' and user 'roles' (the subject). Generally there would only be one of each of these, for example Android and the OpenWrt CIL policy both use user: *u* role: *r*. Note that the *object_r* role is used to label objects.

SELinux Libraries and Utilities

The **Project Repositories** section lists the code that should be installed on the host build system, not all of these would be required on the target system.

A possible minimum list of SELinux items required on the target system are:

- *libselinux* - Provides functions to load policy, label processes and files etc. A list of functions is in **Appendix B - libselinux API Summary**
- *libsepol* - Provides services to build/load policy.
- ***restorecon****(8)* - Label files.
- The policy plus supporting configuration files.

Whether ***setenforce****(8)* is deployed on the target to set enforcing or permissive modes will depend on the overall system requirements.

If ***booleans****(8)* are supported on the target, then ***setsebool****(8)* will be required unless ***semanage****(8)* services are installed.

If the target policy is to be:

- Built on the device, then either ***checkpolicy****(8)* or ***secilc****(8)* will be required.
- Managed on the device during runtime, then ***semanage****(8)*, ***semodule****(8)* and their supporting services will be required.

Depending on the target memory available it would be possible to modify the SELinux libraries as there is legacy code that could be removed. Also (for example) if the userspace avc (***avc_*****(3)*) services in the *libselinux* library are not required these could be removed. It should be noted that currently there are no build options to do this.

Labeling Files

If there is a need to support ***xattr**(7)* filesystems on the target then these need to be labeled via the ***file_contexts**(5)* file that would be generated as part of the initial policy build.

For example RAM based filesystems will require labeling before use (as Android does). To achieve this either ***setfiles**(8)* or ***restorecon**(8)* will need to be run.

These are based on common source code (https://github.com/SELinuxProject/selinux/tree/master/policycoreutils/setfiles) with the majority of functionality built into *libselinux*, therefore it matters little which is used, although ***restorecon**(8)* is probably the best choice as it's smaller and does not support checking files against a different policy.

***setfiles**(8)* will label files recursively on directories and is generally used by the initial SELinux installation process, whereas ***restorecon**(8)* must have the *-r* flag set to label files recursively on directories and is generally used to correct/update files on the running system.

Loading Policy

When the standard *libselinux* and the ***load_policy**(8)* utility are used to load policy, it will always be loaded from the */etc/selinux/<SELINUXTYPE>/policy* directory, where <*SELINUXTYPE*> is the entry from the ***/etc/selinux/config*** file:

```
# This file controls the state of SELinux on the system.
# SELINUX= can take one of these three values:
#     enforcing - SELinux security policy is enforced.
#     permissive - SELinux prints warnings instead of enforcing.
#     disabled - No SELinux policy is loaded.
SELINUX=enforcing
# SELINUXTYPE= The <NAME> of the directory where the active policy and its
#              configuration files are located.
SELINUXTYPE=targeted
```

Note setting *SELINUX=disabled* will be deprecated at some stage, in favor of the existing kernel command line switch *selinux=0*, which allows users to disable SELinux at system boot. See https://github.com/SELinuxProject/selinux-kernel/wiki/DEPRECATE-runtime-disable that explains how to achieve this on various Linux distributions.

The standard Linux SELinux policy load sequence is as follows:

- Obtain policy version supported by the kernel.
- Obtain minimum policy version supported by *libsepol*.
- Determine policy load path via */etc/selinux/config* <*SELINUXTYPE*> entry.
- Search for a suitable policy to load by comparing the kernel and *libsepol* versions using the */etc/selinux/<SELINUXTYPE>/policy/policy.<ver>* file extension.
- Load and if necessary downgrade the policy. This happens if the policy being loaded has a greater version than the kernel supports. Note that if the policy was built with *–handle-unknown=deny* (*UNK_PERMS* in *build.conf*) and there are unknown classes or permissions, the policy will not be loaded.

The only known deviation from this is the Android project that has its own specific method as described in the **SE for Android** - *external/selinux/libselinux* section. To inspect the code, see the ***selinux_android_load_policy()*** function in https://android.googlesource.com/platform/external/selinux/+/refs/heads/master/libselinux/src/android/android_platform.c. Basically this maps the policy file to memory, then calls ***security_load_policy**(3)* to load the policy (as Android does not use the version extension or load policy from the */etc/selinux/<SELINUXTYPE>/policy* directory).

The OpenWrt Project

The OpenWrt project is a Linux based system targeting embedded devices that can be built with either the **Reference Policy** or a **customised CIL policy**

The policy to configure is selected from the *menuconfig* options once OpenWrt is installed:

```
make menuconfig

# Select:
    Global build settings  --->
# Enable SELinux:
      [*] Enable SELinux
            default SELinux type (targeted)  --->
# Select either Reference Policy or customised CIL policy (dssp):
            ( ) targeted
            (X) dssp
```

To build and inspect the CIL policy:

```
git clone https://git.defensec.nl/selinux-policy.git
cd selinux-policy
make policy
```

There should be a binary *policy.<ver>* file that can be viewed using tools such as ***apol**(1)*. The auto-generated ***file_contexts**(5)* file can be viewed with a text editor.

Note that no *policy.conf* is generated when building CIL policy as ***secilc**(8)* is used. To build a *policy.conf* file for inspection via a text editor run:

```
checkpolicy -bF -o policy.conf policy.31
```

This work in progress document https://github.com/doverride/openwrt-selinux-policy/blob/master/README.md contains instructions to assemble OpenWrt from modules applicable to a particular system and how to build on top of it. Also explained is how to fork the policy to form a new base for building a customised target policy.

The Android Project

This is fully discussed in the **SE for Android** section with a section below that explains **Building A Sample Android Policy** to examine its construction.

Building A Small Monolithic Reference Policy

This section describes how a smaller monolithic Reference Policy can be built as a starter policy. It supports the minimum of policy modules that can be defined in a *modules.conf* file, this file is described in the **Reference Policy Build Options - *policy/modules.conf*** section.

To start download the Reference Policy source:

```
git clone https://github.com/SELinuxProject/refpolicy.git
cd refpolicy
```

For the initial configuration, either replace the current *build.conf* file with the sample ***build.conf*** or edit the current *build.conf* file to the requirements (e.g. MONOLITHIC = y)

Install the source policy in the build directory:

```
make install-src
cd /etc/selinux/<NAME>/src/policy
```

The following mandatory *make conf* step will build the initial *policy/booleans.conf* and *policy/modules.conf* files.

This process will also build the *policy/modules/kernel/corenetwork.te* and *corenetwork.if* files. These would be based on the contents of *corenetwork.te.in* and *corenetwork.if.in* configuration files.

For this build only the *policy/modules.conf* will be replaced with the sample version.

```
make conf
```

Replace the current *policy/modules.conf* with the sample ***modules.conf*** and run:

```
make install
```

The binary policy will now be built in the */etc/selinux/<NAME>/policy* directory. The */etc/selinux/<NAME>/src/policy/policy.conf* file contains the policy language statements used to generate the binary policy.

The *policy.conf* file can be examined with a text editor and the binary *policy.<ver>* file can be viewed using tools such as ***apol****(1)*.

Adding Additional Modules

Note that if the *modules.conf* file is modified to add additional modules, *make clean* MUST be run before *make install* or *make load.*

The ease of adding additional modules to the *policy/modules.conf* file depends on the modules dependencies, for example to add the *ftp* module:

```
# Layer: services
# Module: ftp
#
# File transfer protocol service.
#
ftp = module
```

Now run:

```
make clean
make install
```

to build the policy, this will flag a dependency error:

```
policy/modules/services/ftp.te:488:ERROR 'type ssh_home_t is not within scope'
```

This shows that the *ftp* module relies on the *ssh* module, therefore need to also add:

```
# Layer: services
# Module: ssh
#
# Secure shell client and server policy.
#
ssh = module
```

Now run:

```
make clean
make install
```

This should build a valid policy. Although note that adding some modules will lead to a string of dependent modules.

If a suitable module cannot be found in the *policy/modules* directory, then one can be generated and added to the store. To generate policy modules using output from the audit log, see ***audit2allow**(1)* (however review any policy generated). The **Reference Policy** section explains the format of these module files.

The Clean-up

Once a policy is complete it could be cleaned up by removing components that are not required for example:

- The *file_contexts* generated will have entries that could be deleted.
- Unused boleans could be removed.
- Review Policy Capabilities.
- Remove unused classes and permissions (*policy/flask/security_classes* and *policy/flask/access_vectors*).
- There are a number of policy configuration files that can be removed within *etc/selinux/refpolicy* (e.g. *etc/selinux/refpolicy/contexts/x_contexts*).

These will probably amount to small fry, but every little helps!!

Building A Sample Android Policy

A purpose built embedded policy example is the Android policy that is discussed in the **SE for Android** section. This policy has become more complex over time, however they did start with a basic policy that can be explored as described below that does not require obtaining the full AOSP source and build environment.

Android - The SELinux Policy section describes how an Android policy is constructed using ***m4**(1)* macros, **.te* files etc., similar to the **Reference Policy**.

To build a sample policy for inspection:

- Obtain a copy of the Android policy built for 4.1, note that only the core policy is built here as Android adds device specific policy modules as per its build configuration (an example build with a device is shown later).

```
git clone https://android.googlesource.com/platform/external/sepolicy
cd sepolicy
git checkout android-4.1.1_r1
```

- Copy the text below into a *Makefile* installed in the *sepolicy* directory.

```
build_policy:
    m4 -D mls_num_sens=1 \
        -D mls_num_cats=1024 \
        -s security_classes \
        initial_sids \
        access_vectors \
        global_macros \
        mls_macros \
        mls \
        policy_capabilities \
        te_macros \
        attributes \
        *.te \
        roles \
        users \
        ocontexts > policy.conf
    checkpolicy -U deny -M -o sepolicy policy.conf
```

- Run *make* to build the policy. There should be a *policy.conf* file that can be examined with a text editor and a binary *sepolicy* policy file that can be viewed using tools such as ***apol**(1)*. Note the order in which the *policy.conf* file is built as it conforms to the layout described in the **Kernel Policy Language** section.

Over time the Android policy locked down more and more processes and then became more complex as policy version control was required when upgrading. The **Brillo** release was their first IoT release and can be built using the instructions in the *brillo/Makefile* To build a policy containing a device, follow the instructions in the *brillo-device/Makefile* as a device policy must be obtained from the Android repository.

Later Android split policy into private and public segments, they also used CIL for some policy components as described in the **Android - The SELinux Policy** section. The **Android 10** release policy is an example where this split policy is used. This can be built using the instructions in the *android-10/Makefile*.

Security Enhancements for Android

This section gives an overview of the enhancements made to Android to add SELinux services to Security Enhancements for Android™ (SE for Android).

The main objective is to provide a reference for the tools, commands, policy building tools and file formats for the SELinux components of Android based on AOSP master as of May '20). The AOSP git repository can be found at **https://android.googlesource.com**

Note: Check the AOSP tree for any changes as there has been many updates to how SELinux is configured/built over the years.

The following link describes how to validate SELinux in Android: https://source.android.com/security/selinux/ and "**Security Enhanced (SE) Android: Bringing Flexible MAC to Android**" available at http://www.cs.columbia.edu/~lierranli/coms6998-7Spring2014/papers/SEAndroid-NDSS2013.pdf being a recommended read.

The white paper "An Overview of Samsung KNOX" also gives an overview of how SELinux / Android are being integrated with other security services (such as secure boot and integrity measurement) to help provide a more secure mobile platform.

SE for Android Project Updates

This gives a high level view of the new and updated projects to support SE for Android services and covers AOSP with any additional SEAndroid functions noted. These are not a complete set of updates, but give some idea of the scope.

external/selinux/libselinux

Provides the SELinux userspace function library that is installed on the device. It has additional functions to support Android as summarised in *external/selinux/README.android*. It is built from a merged upstream version (https://github.com/SELinuxProject/selinux) with Android specific additions (https://android.googlesource.com/platform/external/selinux/) such as:

selinux_android_setcontext()

- Sets the correct domain context when launching applications using ***setcon**(3)*. Information contained in the *seapp_contexts* file is used to compute the correct context.
- It is called by *frameworks/base/core/jni/com_android_internal_os_Zygote.cpp* when forking a new process and the *system/core/run-as/run-as.cpp* utility for app debugging.

selinux_android_restorecon()

- Sets the correct context on application directory / files using ***setfilecon**(3)*. Information contained in the *seapp_contexts* file is used to compute the correct context.
- The function is used in many places, such as *system/core/init/selinux.cpp* and *system/core/init/ueventd.cpp* to label devices.
- *frameworks/native/cmds/installd/installdNativeService.cpp* when installing a new app.
- *frameworks/base/core/jni/android_os_SELinux.cpp* for the Java *native_restorecon* method.

selinux_android_restorecon_pkgdir()

- Used by *frameworks/native/cmds/installd/InstalldNativeService.cpp* for the package installer.

selinux_android_load_policy()

selinux_android_load_policy_from_fd()

- Used by *system/core/init/selinux.cpp* to initialise SELinux policy, the following note has been extracted:

```
// IMPLEMENTATION NOTE: Split policy consists of three CIL files:
// * platform -- policy needed due to logic contained in the system image,
// * non-platform -- policy needed due to logic contained in the vendor image,
// * mapping -- mapping policy which helps preserve forward-compatibility of
//   non-platform policy with newer versions of platform policy.
//
// secilc is invoked to compile the above three policy files into a single
// monolithic policy file. This file is then loaded into the kernel.
```

There is a labeling service for ***selabel_lookup**(3)* to query the Android *property_contexts* and *service_contexts* files. See *frameworks/native/cmds/servicemanager/Access.cpp* for an example.

Various Android services will also call (not a complete list):

- ***selinux_status_updated**(3)*, ***is_selinux_enabled**(3)*, to check whether anything changed within the SELinux environment (e.g. updated configuration files).
- ***selinux_check_access**(3)* to check if the source context has access permission for the class on the target context.
- ***selinux_label_open**(3)*, ***selabel_lookup**(3)*, ***selinux_android_file_context_handle***, ***lgetfilecon**(3)*, ***selinux_android_prop_context_handle***, ***setfilecon**(3)*, ***setfscreatecon**(3)* to manage file labeling.
- ***selabel_lookup_best_match*** called via *system/core/init/selabel.cpp* by *system/core/init/devices.cpp* when *ueventd* creates a device node as it may also create one or more symlinks (for block and PCI devices). Therefore a “best match” look-up for a device node is based on its real path, plus any links that may have been created.

external/selinux/libsepol

Provides the policy userspace library for building policy on the host and is not available on the device. There are no specific updates to support Android except an *Android.bp* file.

external/selinux/checkpolicy

Provides the policy build tool. Added support for MacOS X. Not available on the device as policy rebuilds are done in the development environment. There are no specific updates to support Android except an *Android.bp* file.

bootable/recovery

Changes to manage file labeling on recovery using functions such as **selinux_android_file_context_handle()**, ***selabel_lookup**(3)* and ***setfscreatecon**(3)*.

build

Changes to build SE for Android and manage file labeling on images and OTA (over the air) target files.

frameworks/base

JNI - Add SELinux support functions such as *isSELinuxEnabled* and *setFSCreateCon*.

SELinux Java class and method definitions.

Checking Zygote connection contexts.

Managing file permissions for the package manager and wallpaper services.

system/core

SELinux support services for toolbox/toybox (e.g. *load_policy*, *runcon*).

SELinux support for system initialisation (e.g. *init*, *init.rc*).

SELinux support for auditing avc’s (*auditd*).

system/sepolicy

This area contains information required to build the SELinux kernel policy and its supporting files. Android splits the policy into sections:

- ***private*** - This is policy specifically for the core components of Android that looks much like the reference policy, that has the policy modules (*.*te* files), class / permission files etc..
- ***vendor*** - This is vendor specific policy.
- ***public*** - This is public specific policy.

The policy is built and installed on the target device along with its supporting configuration files.

The policy files are discussed in the **SELinux Policy Files** section and support tools in the **Policy Build Tools** section.

The Android specific object classes are described in the **Android Classes & Permissions** section.

The **Embedded Systems - Building A Sample Android Policy** section explains how to build basic Android policies. These can be explored without requiring the full AOSP source and build environment.

kernel

All Android kernels support the Linux Security Module (LSM) and SELinux services, however they are based on various versions, therefore the latest SELinux enhancements may not always be present. The **Kernel LSM / SELinux Support** section describes the Android kernel changes and the **Linux Security Module and SELinux** section describes the core SELinux services in the kernel.

device

Build information for each device that includes device specific policy as discussed in the **The SELinux Policy** and **Managing Device Policy Files** sections.

Kernel LSM / SELinux Support

The paper "Security Enhanced (SE) Android: Bringing Flexible MAC to Android" available at http://www.internetsociety.org/sites/default/files/02_4.pdf gives a good review of what did and didn't change in the kernel to support Android. The only major change was to support the Binder IPC service that consists of the following:

1. LSM hooks in the binder code (*drivers/android/binder.c*) and (*include/linux/security.h*)
2. Hooks in the LSM security module (*security/security.c*).
3. SELinux support for the binder object class and permissions (*security/selinux/include/classmap.h*) that are shown in the **Android Classes & Permissions** section. Support for these permission checks are added to *security/selinux/hooks.c*.

Kernel 5.0+ supports Dynamically Allocated Binder Devices, therefore configuring specific devices (e.g. **CONFIG_ANDROID_BINDER_DEVICES="binder"**) is no longer required (use ***CONFIG_ANDROID_BINDERFS*=*y*** instead).

Android Classes & Permissions

Additional classes have been added to Android and are listed below with descriptions of their permissions. The policy files *system/sepolicy/private/security_classes* and *system/sepolicy/private/access_vectors* contain the complete list with descriptions available at: **Appendix A - Object Classes and Permissions**. However, note that while the *security_classes* file contains many entries, not all are required for Android.

binder

Manage the Binder IPC service (4 unique permissions).

Permissions:

call

- Perform a binder IPC to a given target process (can A call B?).

impersonate

- Perform a binder IPC on behalf of another process (can A impersonate B on an IPC).

set_context_mgr

- Register self as the Binder Context Manager aka *servicemanager* (global name service). Can A set the context manager to B, where normally A == B.

transfer

- Transfer a binder reference to another process (can A transfer a binder reference to B?).

property_service

This is a userspace object to manage the Android Property Service in *system/core/init/property_service.cpp* (1 unique permission).

Permission:

set

- Set a property.

service_manager

This is a userspace object to manage the loading of Android services in *frameworks/native/cmds/servicemanager/service_manager/Access.cpp* (3 unique permissions).

Permissions:

add

- Add a service.

find

- Find a service.

list

- List services.

hwservice_manager

This is a userspace object to manage the loading of Android services in *system/hwservicemanager/AccessControl.cpp* (3 unique permissions).

Permissions:

add

- Add a service.

find

- Find a service.

list

- List services.

keystore_key

This is a userspace object to manage the Android keystores. See *system/security/keystore/key_store_service.cpp* (19 unique permissions).

get_state

- check if keystore okay.

get

- Get key.

insert

- Insert / update key.

delete

- Delete key.

exist

- Check if key exists.

list

- Search for matching string.

reset

- Reset keystore for primary user.

password

- Generate new keystore password for primary user.

lock

- Lock keystore.

unlock

- Unlock keystore.

is_empty

- Check if keystore empty.

sign

- Sign data.

verify

- Verify data.

grant

- Add or remove access.

duplicate

- Duplicate the key.

clear_uid

- Clear keys for this *uid*.

add_auth

- Add hardware authentication token.

user_changed

- Add/Remove *uid*.

gen_unique_id

- Generate new keystore password for this *uid*.

drmservice

This is a userspace object to allow finer access control of the Digital Rights Management services. See *frameworks/av/drm/drmserver/DrmManagerService.cpp* (8 unique permissions).

Permissions:

consumeRights

- Consume rights for content.

setPlaybackStatus

- Set the playback state.

openDecryptSession

- Open the DRM session for the requested DRM plugin.

closeDecryptSession

- Close DRM session.

initializeDecryptUnit

- Initialise the decrypt resources.

decrypt

- Decrypt data stream.

finalizeDecryptUnit

- Release DRM resources.

pread

- Read the data stream.

SELinux Enabled Commands

A subset of the Linux SELinux commands have been implemented in Android and are listed below. Some are available as Toolbox or Toybox commands (see *system/core/shell_and_utilities/README.md*) and can be run via *adb shell*, for example:

```
adb shell pm list permissions -g
```

An example set of SELinux commands available are:

getenforce

- Returns the current enforcing mode.

setenforce

- Modify the SELinux enforcing mode: *setenforce [enforcing|permissive|1|0]*

load_policy

- Load new policy into kernel: *load_policy policy-file*

ls

- Supports *-Z* option to display security context.

ps

- Supports *-Z* option to display security context.

restorecon

- Restore file default security context as defined in the *file_contexts* or *seapp_contexts* files. The options are: *-D* - data files, *-F* - Force reset, *-n* - do not change, *-R*/*-r* - Recursive change, *-v* - Show changes.
    - *restorecon [-DFnrRv] pathname*

chcon

- Change security context of file. The options are: *-h*- Change symlinks, *-R* - Recurse into subdirectories, *-v* - Verbose output.
    - *chcon [-hRv] context file...*

runcon

- Run command in specified security context: *runcon context program args...*

id

- If SELinux is enabled then the security context is automatically displayed.

SELinux Public Methods

The public methods implemented are equivalent to *libselinux* functions and shown below. They have been taken from *frameworks/base/core/java/android/os/SELinux.java*.

The SELinux class and its methods are not available in the Android SDK, however if developing SELinux enabled apps within AOSP then Reflection proguard flags would be used (for example in the TV package *AboutFragment.java* calls **SELinux.isSELinuxEnabled()**.

String fileSelabelLookup(String path)

- Get context associated with path by file_contexts.
  - Return a string representing the security context or null on failure.

boolean isSELinuxEnabled()

- Determine whether SELinux is enabled or disabled.
  - Return *true* if SELinux is enabled.

boolean isSELinuxEnforced()

- Determine whether SELinux is permissive or enforcing.
  - Returns *true* if SELinux is enforcing.

boolean setFSCreateContext(String context)

- Sets the security context for newly created file objects. *context* is the security context to set.
  - Returns *true* if the operation succeeded.

boolean setFileContext(String path, String context)

- Change the security context of an existing file object. *path* represents the path of file object to relabel. *context* is the new security context to set.
  - Returns *true* if the operation succeeded.

String getFileContext(String path)

- Get the security context of a file object. *path* the pathname of the file object.
  - Returns the requested security context or null.

String getPeerContext(FileDescriptor fd)

- Get the security context of a peer socket. *FileDescriptor* is the file descriptor class of the peer socket.
  - Returns the peer socket security context or null.

String getContext()

- Gets the security context of the current process.
  - Returns the current process security context or null.

String getPidContext(int pid)

- Gets the security context of a given process id. *pid* and *int* representing the process id to check.
  - Returns the security context of the given pid or null.

boolean checkSELinuxAccess(String scon, String tcon, String tclass, String perm)

- Check permissions between two security contexts. *scon* is the source or subject security context. *tcon* is the target or object security context. *tclass* is the object security class name. *perm* is the permission name.
  - Returns true if permission was granted.

boolean restorecon(String pathname)

- Restores a file to its default SELinux security context. If the system is not compiled with SELinux, then true is automatically returned. If SELinux is compiled in, but disabled, then true is returned. *pathname* is the pathname of the file to be relabeled.
    - Returns true if the relabeling succeeded.
    - *exception NullPointerException* if the pathname is a null object.

boolean native_restorecon(String pathname, int flags)

- Restores a file to its default SELinux security context.
    - Returns *true* if the relabeling succeeded.
    - *exception NullPointerException* if the pathname is a null object.

boolean restorecon(File file)

- Restores a file to its default SELinux security context. If the system is not compiled with SELinux, then true is automatically returned. If SELinux is compiled in, but disabled, then true is returned. *file* is the file object representing the path to be relabeled.
    - Returns true if the relabeling succeeded.
    - *exception NullPointerException* if the file is a null object.

boolean restoreconRecursive(File file)

- Recursively restores all files under the given path to their default SELinux security context. If the system is not compiled with SELinux, then true is automatically returned. If SELinux is compiled in, but disabled, then true is returned. *pathname* is the pathname of the file to be relabeled.
    - Returns a boolean indicating whether the relabeling succeeded.

Android Init Language SELinux Extensions

The Android init process language has been expanded to support SELinux as shown below. The complete Android *init* language description is available in the *system/core/init/readme.txt* file.

seclabel <securitycontext>

- service option: Change to security context before exec'ing this service. Primarily for use by services run from the rootfs, e.g. *ueventd*, *adbd*. Services on the system partition can instead use policy defined transitions based on their file security context. If not specified and no transition is defined in policy, defaults to the *init* context.

restorecon <path>

- action command: Restore the file named by <*path*> to the security context specified in the *file_contexts* configuration. Not required for directories created by the *init.rc* as these are automatically labeled correctly by *init*.

restorecon_recursive <path> [<path>]

- action command: Recursively restore the directory tree named by <*path*> to the security context specified in the *file_contexts* configuration. Do NOT use this with paths leading to shell-writable or app-writable directories, e.g. */data/local/tmp*, */data/data* or any prefix thereof.

setcon <securitycontext>

- action command: Set the current process security context to the specified string. This is typically only used from *early-init* to set the *init* context before any other process is started (see *init.rc* example above).

Examples of their usage are shown in the following *init.rc* file segments:

system/core/rootdir/init.rc

```
## Daemon processes to be run by init.
##
service ueventd /system/bin/ueventd
    class core
    critical
    seclabel u:r:ueventd:s0
    shutdown critical

# Set SELinux security contexts on upgrade or policy update.
    restorecon --recursive --skip-ce /data
```

The SELinux Policy

This section covers the SELinux policy, its supporting configuration files and Android specific configuration files (e.g. seapps_context). These file formats are detailed in the Policy File Formats section.

These comments have been extracted from *system/sepolicy/Android.mk*, there are many useful comments in this file regarding its build.

```
# sepolicy is now divided into multiple portions:
# public - policy exported on which non-platform policy developers may write
#   additional policy.  types and attributes are versioned and included in
#   delivered non-platform policy, which is to be combined with platform policy.
# private - platform-only policy required for platform functionality but which
#  is not exported to vendor policy developers and as such may not be assumed
#  to exist.
# vendor - vendor-only policy required for vendor functionality. This policy can
#  reference the public policy but cannot reference the private policy. This
#  policy is for components which are produced from the core/non-vendor tree and
#  placed into a vendor partition.
# mapping - This contains policy statements which map the attributes
#  exposed in the public policy of previous versions to the concrete types used
#  in this policy to ensure that policy targeting attributes from public
#  policy from an older platform version continues to work.

# build process for device:
# 1) convert policies to CIL:
#    - private + public platform policy to CIL
#    - mapping file to CIL (should already be in CIL form)
#    - non-platform public policy to CIL
#    - non-platform public + private policy to CIL
# 2) attributize policy
#    - run script which takes non-platform public and non-platform combined
#      private + public policy and produces attributized and versioned
#      non-platform policy
# 3) combine policy files
#    - combine mapping, platform and non-platform policy.
#    - compile output binary policy file
```

SELinux Policy Files

The core policy files are contained in *system/sepolicy*, with device specific policy in *device/<vendor>/<device>/sepolicy* Once generated, the policy and its supporting configuration files will be installed on the device as part of the build process.

The following files (along with any vendor and device specific policy) are used to build the kernel binary policy file.

private/mls_macro, *public/global_macros*, *public/ioctl_macros*, *public/neverallow_macros*, *public/te_marcos*

- These contain the ***m4**(1)* macros that control and expand the policy files to build a policy in the kernel policy language to be compiled by ***checkpolicy**(8)*. The **Kernel Policy Language** section defines the kernel policy language.

private/access_vectors, *private/security_classes*

- These have been modified to support the new Android classes and permissions.

private/initial_sids, *private/initial_sids_contexts*

- Contains the system initialisation (before policy is loaded) and failsafe (for objects that would not otherwise have a valid label).

fs_use, *genfs_contexts*, *port_contexts*

- Contains policy context for various devices, see the **Kernel Policy Language** section for details.

private/users, *private/roles*

- These define the only user (*u*) and role (*r*) used by the policy.

private/mls

- Contains the constraints to be applied to the defined classes and permissions.

private/policy_capabilities

- Contains the policy capabilities enabled for the kernel policy (see ***policycap*** statement).

*.te

- The **.te* files are the policy module definition files. These are the same format as the standard reference policy and are expanded by the m4 macros. There is (generally) one *.te* file for each domain/service defined containing the policy rules. The device/vendor may also produce these.

file_contexts

- Contains default file contexts for setting the SELinux extended file attributes (***attr**(1)*). The format of this file is defined in the *file_contexts* section. The device/vendor may also produce these.

Android Policy Files

These Android specific files along with any device versions will be used to determinn whether access is allowed or not based on the security contexts contained within them.

seapp_contexts

- Contains information to allow domain or data file contexts to be computed based on parameters as discussed in the *seapp_contexts* section.

property_contexts

- Contains default contexts for Android property services as discussed in the *property_contexts* section.

service_contexts, *hwservice_contexts* and *vndservice_contexts*

- Contains default contexts for Android services as discussed in the *service_contexts* section. The hardware and vendor service files share the same format and use *selabel_open(SELABEL_CTX_ANDROID_SERVICE ..);* and ***selabel_lookup****(3)* for processing files.

mac_permissions.xml

- The Middleware Mandatory Access Control (MMAC) file assigns an *seinfo* tag to apps based on their signature and optionally their package name. The *seinfo* tag can then be used as a key in the *seapp_contexts* file to assign a specific label to all apps with that *seinfo* tag. The configuration file is read by *system_server* during start-up. The main code for the service is *frameworks/base/services/java/com/android/server/pm/SELinuxMMAC.java*, however it does hook into other Android services such as *PackageManagerService.java*. Its format is discussed in the *mac_permissions.xml* section.

Device Specific Policy

Some of this section has been extracted from the *system/sepolicy/README* that should be checked in case there have been updates. It describes how files in *system/sepolicy* can be manipulated during the build process to reflect requirements of different device vendors whose policy files would be located in the *device/<vendor>/<board>/sepolicy* directory.

Important Note: Android policy has a number of ***neverallow*** rules defined in the core policy to ensure that ***allow*** rules are never added to domains that would weaken security. However developers may need to customise their device policies, and as a consequence they may fail one or more of these rules.

Managing Device Policy Files

Additional, per device, policy files can be added into the policy build. These files should have each line including the final line terminated by a newline character (0x0A). This will allow files to be concatenated and processed whenever the ***m4****(1)* macro processor is called by the build process. Adding the newline will also make the intermediate text files easier to read when debugging build failures. The sets of file, service and property contexts files will automatically have a newline inserted between each file as these are common failure points.

These device policy files can be configured through the use of the *BOARD_VENDOR_SEPOLICY_DIRS* variable. This variable should be set in the *BoardConfig.mk* file in the device or vendor directories.

BOARD_VENDOR_SEPOLICY_DIRS contains a list of directories to search for additional policy files. Order matters in this list. For example, if you have 2 instances of *widget.te* files in the *BOARD_VENDOR_SEPOLICY_DIRS* search path, then the first one found (at the first search dir containing the file) will be concatenated first. Reviewing *out/target/product/<device>/obj/ETC/sepolicy_intermediates/policy.conf* will help sort out ordering issues.

Example *BoardConfig.mk* usage from the Tuna device *device/samsung/tuna/BoardConfig.mk*:

```
BOARD_VENDOR_SEPOLICY_DIRS += device/samsung/tuna/sepolicy
```

Additionally, OEMs can specify BOARD_SEPOLICY_M4DEFS to pass arbitrary m4 definitions during the build. A definition consists of a string in the form of *macro-name=value*. Spaces must NOT be present. This is useful for building modular policies, policy generation, conditional file paths, etc. It is supported in the following file types:

- All *.te and SE Linux policy files as passed to checkpolicy
- file_contexts
- service_contexts

- property_contexts
- keys.conf

Example *BoardConfig.mk* Usage:

```
BOARD_SEPOLICY_M4DEFS += btmodule=foomatic \
                         btdevice=/dev/gps
```

Policy Build Tools

The kernel policy is compiled using ***checkpolicy****(8)* via the *system/sepolicy/Android.mk* file. There are also a number of Android specific tools used to assist in policy build/configuration that are described as follows (some text taken from *system/sepolicy/tools/README*):

build_policies.sh - A tool to build SELinux policy for multiple targets in parallel. This is useful for quickly testing a new test or neverallow rule on multiple targets.

Usage:

```
    ./build_policies.sh ~/android/master ~/tmp/build_policies
    ./build_policies.sh ~/android/master ~/tmp/build_policies sailfish-eng walleye-eng
```

checkfc - A utility for checking the validity of a file_contexts or a property_contexts configuration file. Used as part of the policy build to validate both files. Requires the sepolicy file as an argument in order to check the validity of the security contexts in the file_contexts or property_contexts file.

Usage1:

```
    checkfc sepolicy file_contexts
    checkfc -p sepolicy property_contexts
```

- Also used to compare two file_contexts or file_contexts.bin files. Displays one of subset, equal, superset, or incomparable.

Usage2:

```
    checkfc -c file_contexts1 file_contexts2
```

Example:

```
   $ checkfc -c out/target/product/shamu/system/etc/general_file_contexts out/target/
product/shamu/root/file_contexts.bin
subset
```

checkseapp - A utility for merging together the main seapp_contexts configuration and the device-specific one, and simultaneously checking the validity of the configurations. Used as part of the policy build process to merge and validate the configuration.

Usage:

```
    checkseapp -p sepolicy input_seapp_contexts0 [input_seapp_contexts1...] -o
seapp_contexts
```

insertkeys.py - A helper script for mapping tags in the signature stanzas of *mac_permissions.xml* to public keys found in pem files (see the ***mac_permissions.xml*** file section). The resulting *mac_permissions.xml* file will also be stripped of comments and whitespace.

Also uses information contained in various *keys.conf* files that are described in the **keys.conf** file section.

post_process_mac_perms - A tool to help modify an existing mac_permissions.xml with additional app certs not already found in that policy. This becomes useful when a directory containing apps is searched and the certs from those apps are added to the policy not already explicitly listed.

Usage:

```
    post_process_mac_perms [-h] -s SEINFO -d DIR -f POLICY

      -s SEINFO, --seinfo SEINFO  seinfo tag for each generated stanza
      -d DIR, --dir DIR           Directory to search for apks
      -f POLICY, --file POLICY    mac_permissions.xml policy file
```

sepolicy-check - A tool for auditing a sepolicy file for any allow rule that grants a given permission.

Usage:

```
    sepolicy-check -s <domain> -t <type> -c <class> -p <permission> -P out/target/
product/<board>/root/sepolicy
```

sepolicy-analyze - A tool for performing various kinds of analysis on a sepolicy file. During policy build it is used to check for any *permissive* domains (not allowed) and *neverallow* assertions

version_policy - Takes the given public platform policy, a private policy and a version number to produced a combined “versioned” policy file.

Logging and Auditing

Android supports auditing of SELinux events via the AOSP logger service that can be viewed using *logcat*, for example:

```
adb logcat > logcat.log
```

Example SELinux audit events (avc denials) are:

```
/system/bin/init: type=1107 audit(0.0:7): uid=0 auid=4294967295 ses=4294967295
subj=u:r:init:s0 msg='avc: denied { set } for property=vendor.wlan.firmware.version
pid=357 uid=1010 gid=1010 scontext=u:r:hal_wifi_default:s0
tcontext=u:object_r:vendor_default_prop:s0 tclass=property_service permissive=0' b/
131598173

traced_probes: type=1400 audit(0.0:9): avc: denied { read } for name="format"
dev="tracefs" ino=5283 scontext=u:r:traced_probes:s0
tcontext=u:object_r:debugfs_tracing_debug:s0 tclass=file permissive=0
```

```
dmesg   : type=1400 audit(0.0:198): avc: denied { syslog_read } for
scontext=u:r:shell:s0 tcontext=u:r:kernel:s0 tclass=system permissive=0
```

Note that before the auditing daemon is loaded, messages will be logged in the kernel buffers that can be read using ***dmesg**(1)*:

```
adb shell dmesg
```

Policy File Formats

This section details the following Android policy files:

- *file_contexts*
- *seapp_contexts*
- *service_contexts*
- *property_contexts*
- *mac_permissions.xml*
- *keys.conf*

file_contexts

This file is used to associate default contexts to files for file systems that support extended file attributes. It is used by the file labeling commands such as *restorecon*.

The build process supports additional *file_contexts* files allowing devices to specify their entries as described in the **Managing Device Policy Files** section.

Each line within the file consists of the following:

```
pathname_regexp [file_type] security_context
```

Where:

pathname_regexp

- An entry that defines the pathname that may be in the form of a regular expression (see the example *file_contexts* files below). Note that */dev/block* devices may have additional keywords: *by-name* and *by-uuid* These are resolved by *system/core/init/devices.cpp* as a block device *name*/*uuid* may change based on the detection order etc..

file_type

- This entry equates to the file mode and also the file related object class (e.g. *S_IFSOCK* = *sock_file* class).One of the following optional *file_type entries* (note if blank means “match all file types”):
  - *-b* - Block Device
  - *-c* - Character Device
  - *-d* - Directory
  - *-p* - Named Pipe (FIFO)
  - *-l* - Symbolic Link
  - *-s* - Socket File
  - *–* - Ordinary file

security_context

- This entry can be either:
  - The security context that will be assigned to the file.
  - A value of <<*none*>> that states matching files should not be re-labeled.

Example entries:

```
########################################
# Root
/                   u:object_r:rootfs:s0

# Data files
/adb_keys           u:object_r:adb_keys_file:s0
/build\.prop        u:object_r:rootfs:s0
/default\.prop      u:object_r:rootfs:s0
/fstab\..*          u:object_r:rootfs:s0
/init\..*           u:object_r:rootfs:s0
/res(/.*)?          u:object_r:rootfs:s0
/selinux_version    u:object_r:rootfs:s0
/ueventd\..*        u:object_r:rootfs:s0
/verity_key         u:object_r:rootfs:s0
...
#########################
# Devices
#
/dev(/.*)?      u:object_r:device:s0
/dev/adf[0-9]*      u:object_r:graphics_device:s0
/dev/adf-interface[0-9]*\.[0-9]*    u:object_r:graphics_device:s0
/dev/adf-overlay-engine[0-9]*\.[0-9]*   u:object_r:graphics_device:s0
/dev/ashmem     u:object_r:ashmem_device:s0
/dev/ashmem(.*)?    u:object_r:ashmem_libcutils_device:s0
...
############################
# Vendor files from /(product|system/product)/vendor_overlay
#
# NOTE: For additional vendor file contexts for vendor overlay files,
# use device specific file_contexts.
#
/(product|system/product)/vendor_overlay/[0-9]+/.*   u:object_r:vendor_file:s0
```

These are example *device/google/coral-sepolicy/vendor/qcom/sm8150/file_contexts* entries that has a '***by-name***' entry.

```
# Same process file
/vendor/lib(64)?/hw/gralloc\.msmnile\.so
u:object_r:same_process_hal_file:s0
/vendor/lib(64)?/hw/vulkan\.msmnile\.so
u:object_r:same_process_hal_file:s0
/vendor/lib(64)?/hw/gralloc\.sm8150\.so
u:object_r:same_process_hal_file:s0

/(vendor|system/vendor)/bin/sscrpcd                     u:object_r:sensors_exec:s0
```

```
/dev/block/platform/soc/1d84000\.ufshc/by-name/abl_[ab]
u:object_r:custom_ab_block_device:s0
/dev/block/platform/soc/1d84000\.ufshc/by-name/aop_[ab]
u:object_r:custom_ab_block_device:s0
/dev/block/platform/soc/1d84000\.ufshc/by-name/apdp_[ab] u:object_r:dp_block_device:s0
```

The '***by-name***'entry can also exist in *fstab* files as shown in this example taken from *device/generic/goldfish/fstab.ranchu*:

```
# Android fstab file.
...
/dev/block/pci/pci0000:00/0000:00:06.0/by-name/metadata /metadata ext4 .....
```

seapp_contexts

The build process supports additional *seapp_contexts* files allowing devices to specify their entries as described in the Device Specific Policy section.

The following sections will show:

1. The default *system/sepolicy/private/seapp_contexts* file entries.
2. A description of the *seapp_contexts* entries and their usage.
3. A brief description of how a context is computed using either the *selinux_android_setcontext* or *selinux_android_ setfilecon* function using the *seapp_contexts* file entries.
4. Examples of computed domain and directory contexts for various apps.

Default Entries

The default Android *system/sepolicy/private/seapp_contexts* file contains the following types of entry:

```
# only the system server can be in system_server domain
neverallow isSystemServer=false domain=system_server
neverallow isSystemServer="" domain=system_server
...
# Ephemeral Apps must run in the ephemeral_app domain
neverallow isEphemeralApp=true domain=((?!ephemeral_app).)*

isSystemServer=true domain=system_server_startup

user=_app seinfo=platform name=com.android.traceur domain=traceur_app
type=app_data_file levelFrom=all
user=system seinfo=platform domain=system_app type=system_app_data_file
...
user=_app minTargetSdkVersion=28 fromRunAs=true domain=runas_app levelFrom=all
user=_app fromRunAs=true domain=runas_app levelFrom=user
```

Entry Definitions

The following has been extracted from the *system/sepolicy/private/seapp_contexts* file:

```
# The entries in this file define how security contexts for apps are determined.
# Each entry lists input selectors, used to match the app, and outputs which are
# used to determine the security contexts for matching apps.
```

```
#
# Input selectors:
#       isSystemServer (boolean)
#       isEphemeralApp (boolean)
#       isOwner (boolean)
#       user (string)
#       seinfo (string)
#       name (string)
#       path (string)
#       isPrivApp (boolean)
#       minTargetSdkVersion (unsigned integer)
#       fromRunAs (boolean)
#
# All specified input selectors in an entry must match (i.e. logical AND).
# An unspecified string or boolean selector with no default will match any
# value.
# A user, name, or path string selector that ends in * will perform a prefix
# match.
# String matching is case-insensitive.
# See external/selinux/libselinux/src/android/android_platform.c,
# seapp_context_lookup().
#
# isSystemServer=true only matches the system server.
# An unspecified isSystemServer defaults to false.
# isEphemeralApp=true will match apps marked by PackageManager as Ephemeral
# isOwner=true will only match for the owner/primary user.
# user=_app will match any regular app process.
# user=_isolated will match any isolated service process.
# Other values of user are matched against the name associated with the process
# UID.
# seinfo= matches against the seinfo tag for the app, determined from
# mac_permissions.xml files.
# The ':' character is reserved and may not be used in seinfo.
# name= matches against the package name of the app.
# path= matches against the directory path when labeling app directories.
# isPrivApp=true will only match for applications preinstalled in
#       /system/priv-app.
# minTargetSdkVersion will match applications with a targetSdkVersion
#       greater than or equal to the specified value. If unspecified,
#       it has a default value of 0.
# fromRunAs=true means the process being labeled is started by run-as. Default
# is false.
#
# Precedence: entries are compared using the following rules, in the order shown
# (see external/selinux/libselinux/src/android/android_platform.c,
# seapp_context_cmp()).
#       (1) isSystemServer=true before isSystemServer=false.
#       (2) Specified isEphemeralApp= before unspecified isEphemeralApp=
#             boolean.
#       (3) Specified isOwner= before unspecified isOwner= boolean.
#       (4) Specified user= string before unspecified user= string;
#             more specific user= string before less specific user= string.
#       (5) Specified seinfo= string before unspecified seinfo= string.
#       (6) Specified name= string before unspecified name= string;
#             more specific name= string before less specific name= string.
```

```
#       (7) Specified path= string before unspecified path= string.
#             more specific name= string before less specific name= string.
#       (8) Specified isPrivApp= before unspecified isPrivApp= boolean.
#       (9) Higher value of minTargetSdkVersion= before lower value of
#              minTargetSdkVersion= integer. Note that minTargetSdkVersion=
#              defaults to 0 if unspecified.
#       (10) fromRunAs=true before fromRunAs=false.
# (A fixed selector is more specific than a prefix, i.e. ending in *, and a
# longer prefix is more specific than a shorter prefix.)
# Apps are checked against entries in precedence order until the first match,
# regardless of their order in this file.
#
# Duplicate entries, i.e. with identical input selectors, are not allowed.
#
# Outputs:
#       domain (string)
#       type (string)
#       levelFrom (string; one of none, all, app, or user)
#       level (string)
#
# domain= determines the label to be used for the app process; entries
# without domain= are ignored for this purpose.
# type= specifies the label to be used for the app data directory; entries
# without type= are ignored for this purpose.
# levelFrom and level are used to determine the level (sensitivity + categories)
# for MLS/MCS.
# levelFrom=none omits the level.
# levelFrom=app determines the level from the process UID.
# levelFrom=user determines the level from the user ID.
# levelFrom=all determines the level from both UID and user ID.
#
# levelFrom=user is only supported for _app or _isolated UIDs.
# levelFrom=app or levelFrom=all is only supported for _app UIDs.
# level may be used to specify a fixed level for any UID.
#
# For backwards compatibility levelFromUid=true is equivalent to levelFrom=app
# and levelFromUid=false is equivalent to levelFrom=none.
#
#
# Neverallow Assertions
# Additional compile time assertion checks for the rules in this file can be
# added as well. The assertion
# rules are lines beginning with the keyword neverallow. Full support for PCRE
# regular expressions exists on all input and output selectors. Neverallow
# rules are never output to the built seapp_contexts file. Like all keywords,
# neverallows are case-insensitive. A neverallow is asserted when all key value
# inputs are matched on a key value rule line.
#
```

Computing Process Context Examples

The following is an example taken as the system server is loaded:

```
selinux_android_setcontext() parameters:
    uid 1000
    isSystemServer true
    seinfo null
    pkgname null

seapp_contexts lookup parameters:
    uid 1000
    isSystemServer true
    seinfo null
    pkgname null
    path null

Matching seapp_contexts entry:
    isSystemServer=true domain=system_server

Outputs:
    domain system_server
    level s0

Computed context = u:r:system_server:s0
username computed from uid = system

Result using ps -Z command:

LABEL                 USER  PID PPID NAME
u:r:system_server:s0 system 630 329  system_server
```

This is the ’radio’ application that is part of the platform:

```
selinux_android_setcontext() parameters:
    uid 1001
    isSystemServer false
    seinfo platform
    pkgname com.android.phone

seapp_contexts lookup parameters:
    uid 1001 (computes user=radio entry)
    isSystemServer false
    seinfo platform
    pkgname com.android.phone
    path null

Matching seapp_contexts entry:
    user=radio domain=radio type=radio_data_file

Outputs:
    domain radio
    level s0

Computed context = u:r:radio:s0
username computed from uid = radio

Result using ps -Z command:
```

```
LABEL        USER  PID PPID NAME
u:r:radio:s0 radio 959 329  com.android.phone
```

This is a third party app *com.example.myapplication*:

```
selinux_android_setcontext() parameters:
    uid 10149
    isSystemServer false
    seinfo default
    pkgname com.example.myapplication

seapp_contexts lookup parameters:
    uid 10149 (computes user=_app entry)
    isSystemServer false
    seinfo default
    pkgname com.example.myapplication
    path null

Matching seapp_contexts entry:
    user=_app domain=untrusted_app type=app_data_file levelFrom=user

Outputs:
    domain untrusted_app
    level s0:c149,c256,c512,c768

Computed context = u:r:untrusted_app:s0:c149,c256,c512,c768
username computed from uid = u0_a149

Result using ps -Z command:
LABEL                                     USER    PID  PPID NAME
u:r:untrusted_app:s0:c149,c256,c512,c768 u0_a149 1138 64   com.example.myapplication
```

property_contexts

This file holds property service keys and their contexts that are processes by *system/core/init/property_service.cpp*. This service will also resolve the special keywords: *prefix string*, *exact bool* etc.

The build process supports additional *property_contexts* files allowing vendors to specify their entries.

The file format is:

```
property_key security_context type value
```

type

- prefix or exact

value

- int, double, bool or string

Example entries:

```
##########################
# property service keys
#
#
net.rmnet               u:object_r:net_radio_prop:s0
net.gprs                u:object_r:net_radio_prop:s0
net.ppp                 u:object_r:net_radio_prop:s0
net.qmi                 u:object_r:net_radio_prop:s0
net.lte                 u:object_r:net_radio_prop:s0
...
cache_key.bluetooth.      u:object_r:binder_cache_bluetooth_server_prop:s0 prefix
string
cache_key.system_server.  u:object_r:binder_cache_system_server_prop:s0 prefix string
cache_key.telephony.      u:object_r:binder_cache_telephony_server_prop:s0 prefix
string
...
ro.com.android.dataroaming        u:object_r:telephony_config_prop:s0 exact bool
ro.com.android.prov_mobiledata    u:object_r:telephony_config_prop:s0 exact bool
ro.radio.noril                    u:object_r:telephony_config_prop:s0 exact string
ro.telephony.call_ring.multiple   u:object_r:telephony_config_prop:s0 exact bool
ro.telephony.default_cdma_sub     u:object_r:telephony_config_prop:s0 exact int
```

service_contexts

This file holds binder service keys and their contexts that are matched against binder object names using ***selabel_lookup****(3)*. The returned context will then be used as the target context as described in the example below to determine whether the binder service is allowed or denied (see *frameworks/native/cmds/servicemanager/Access.cpp*).

The build process supports additional *service_contexts* files allowing devices to specify their entries.

The file format is:

```
service_key security_context
```

Example *service_contexts* Entries:

```
android.hardware.identity.IIdentityCredentialStore/default
u:object_r:hal_identity_service:s0
android.hardware.light.ILights/default
u:object_r:hal_light_service:s0
android.hardware.power.IPower/default
u:object_r:hal_power_service:s0
android.hardware.rebootescrow.IRebootEscrow/default
u:object_r:hal_rebootescrow_service:s0
android.hardware.vibrator.IVibrator/default
u:object_r:hal_vibrator_service:s0

accessibility                           u:object_r:accessibility_service:s0
account                                 u:object_r:account_service:s0
activity                                u:object_r:activity_service:s0
```

```
activity_task                               u:object_r:activity_task_service:s0
adb                                         u:object_r:adb_service:s0
```

Example *hwservice_contexts* Entries:

```
android.frameworks.automotive.display::IAutomotiveDisplayProxyService
u:object_r:fwk_automotive_display_hwservice:s0
android.frameworks.bufferhub::IBufferHub
u:object_r:fwk_bufferhub_hwservice:s0
android.frameworks.cameraservice.service::ICameraService
u:object_r:fwk_camera_hwservice:s0
android.frameworks.displayservice::IDisplayService
u:object_r:fwk_display_hwservice:s0
android.frameworks.schedulerservice::ISchedulingPolicyService
u:object_r:fwk_scheduler_hwservice:s0
```

Example *vndservice_contexts* Entries:

```
manager                 u:object_r:service_manager_vndservice:s0
*                       u:object_r:default_android_vndservice:s0
```

mac_permissions.xml

The *mac_permissions.xml* file is used to configure Run/Install-time MMAC policy and provides x.509 certificate to *seinfo* string mapping so that Zygote spawns an app in the correct domain. See the **Computing Process Context Examples** section for how this is achieved using information also contained in the *seapp_contexts* file.

An example *mac_permissions.xml* file:

```
<?xml version="1.0" encoding="utf-8"?>
<policy>
    <signer signature="@VRCORE" >
      <package name="com.google.vr.vrcore" >
        <seinfo value="vrcore" />
      </package>
    </signer>
    <signer signature="@VRCORE_DEV" >
      <package name="com.google.vr.vrcore" >
        <seinfo value="vrcore" />
      </package>
    </signer>
</policy>
```

The **signer signature**= entry may have the public base16 signing key present in the string or it may have an entry starting with '@', where the keyword (e.g. VRCORE_DEV) extracts the key from a *pem* file as discussed in the **keys.conf** section. If a base16 key is required, it can be extracted from a package using the *post_process_mac_perms* utility.

The build process also supports additional *mac_permissions.xml* files allowing devices to specify their entries as described in the **Managing Device Policy Files** section.

Policy Rules

The following rules have been extracted from the AOSP *mac_permissions.xml* file:

1. A signature is a hex encoded X.509 certificate or a tag defined in *keys.conf* and is required for each signer tag. The signature can either appear as a set of attached cert child tags or as an attribute.
2. A signer tag must contain a *seinfo* tag XOR multiple package stanzas.
3. Each signer/package tag is allowed to contain one *seinfo* tag. This tag represents additional info that each app can use in setting a SELinux security context on the eventual process as well as the apps data directory.
4. *seinfo* assignments are made according to the following rules:

- Stanzas with package name refinements will be checked first.
- Stanzas w/o package name refinements will be checked second.
- The "default" *seinfo* label is automatically applied.

Valid stanzas can take one of the following forms:

```
// single cert protecting seinfo
<signer signature="@PLATFORM" >
  <seinfo value="platform" />
</signer>

// multiple certs protecting seinfo (all contained certs must match)
<signer>
  <cert signature="@PLATFORM1"/>
  <cert signature="@PLATFORM2"/>
  <seinfo value="platform" />
</signer>

// single cert protecting explicitly named app
<signer signature="@PLATFORM" >
  <package name="com.android.foo">
    <seinfo value="bar" />
  </package>
</signer>

// multiple certs protecting explicitly named app (all certs must match)
<signer>
  <cert signature="@PLATFORM1"/>
  <cert signature="@PLATFORM2"/>
  <package name="com.android.foo">
    <seinfo value="bar" />
  </package>
</signer>
```

keys.conf

The *keys.conf* file is used by **insertkeys.py** for mapping the "@..." tags in *mac_permissions.xml*, *mmac_types.xml* and *content_provider.xml* signature entries with public keys found in *pem* files.

An example *keys.conf* file from *system/sepolicy/private* is as follows:

```
#
# Maps an arbitrary tag [TAGNAME] with the string contents found in
# TARGET_BUILD_VARIANT. Common convention is to start TAGNAME with an @ and
```

```
# name it after the base file name of the pem file.
#
# Each tag (section) then allows one to specify any string found in
# TARGET_BUILD_VARIANT. Typically this is user, eng, and userdebug. Another
# option is to use ALL which will match ANY TARGET_BUILD_VARIANT string.
#

[@PLATFORM]
ALL : $DEFAULT_SYSTEM_DEV_CERTIFICATE/platform.x509.pem

[@MEDIA]
ALL : $DEFAULT_SYSTEM_DEV_CERTIFICATE/media.x509.pem

[@NETWORK_STACK]
ALL : $MAINLINE_SEPOLICY_DEV_CERTIFICATES/networkstack.x509.pem

[@SHARED]
ALL : $DEFAULT_SYSTEM_DEV_CERTIFICATE/shared.x509.pem

# Example of ALL TARGET_BUILD_VARIANTS
[@RELEASE]
ENG       : $DEFAULT_SYSTEM_DEV_CERTIFICATE/testkey.x509.pem
USER      : $DEFAULT_SYSTEM_DEV_CERTIFICATE/testkey.x509.pem
USERDEBUG : $DEFAULT_SYSTEM_DEV_CERTIFICATE/testkey.x509.pem
```

Appendix A - Object Classes and Permissions

Introduction

This section contains a list of object classes and their associated permissions that have been taken from the Fedora policy sources. There are also additional entries for Xen. The Android specific classes and permissions are shown in the **Security Enhancements for Android** section.

The SElinux Testsuite has tests that exercise a number of these object classes/permissions and is a useful reference: **https://github.com/SELinuxProject/selinux-testsuite**

In most cases the permissions are self explanatory as they are those used in the standard Linux function calls (such as 'create a socket' or 'write to a file'). Some SELinux specific permissions are:

relabelfrom

- Used on most objects to allow the objects security context to be changed from the current type.

relabelto

- Used on most objects to allow the objects security context to be changed to the new type.

entrypoint

- Used for files to indicate that they can be used as an entry point into a domain via a domain transition.

execute_no_trans

- Used for files to indicate that they can be used as an entry point into the calling domain (i.e. does not require a domain transition).

execmod

- Generally used for files to indicate that they can execute the modified file in memory.

Where possible the specific object class permissions are explained, however for some permissions it is difficult to determine what they are used for, also some have been defined in documentation but never implemented in code.

Defining Object Classes and Permissions

The **Reference Policy** already contains the default object classes and permissions required to manage the system and supporting services.

For those who write or manager SELinux policy, there is no need to define new objects and their associated permissions as these would be done by those who actually design and/or write object managers.

Note: In theory a policy could be defined with no classes or permissions then set the *handle_unknown* flag when building the policy to *allow* (***checkpolicy****(8)* and ***secilc****(8) [-U handle-unknown (allow,deny,reject)]*). However: - CIL requires at least one class to be defined. - The *process* class with its *transition* and *dyntransition* permissions are still required for default labeling behaviors, role and range transitions in older policy versions.

The **Object Class and Permission Statements** section specifies how these are defined within the Kernel Policy Language, and the **CIL Reference Guide** specifies the CIL Policy Language.

Kernel Object Classes and Permissions

Common Permissions

Common File Permissions

The following table describes the common *file* permissions that are inherited by a number of object classes.

Permissions - 25 permissions:

append

- Append to file.

audit_access

- The rules for this permission work as follows:
  - If a process calls *access()* or *faccessat()* and SELinux denies their request there will be a check for a *dontaudit* rule on the *audit_access* permission.
  - If there is a *dontaudit* rule on *audit_access* an AVC event will not be written.
  - If there is no *dontaudit* rule an AVC event will be written for the permissions requested (*read*, *write*, or *exec*).
- Notes:
  1. There will never be a denial message with the *audit_access* permission as this permission does not control security decisions.
  2. *allow* and *auditallow* rules with this permission are therefore meaningless, however the kernel will accept a policy with such rules, but they will do nothing.

create

- Create new file.

execute

- Execute the file with domain transition.

execmod

- Make executable a file that has been modified by copy-on-write.

getattr

- Get file attributes.

ioctl

- I/O control system call requests.

link

- Create hard link.

lock

- Set and unset file locks.

map

- Allow a file to be memory mapped via ***mmap****(2)*.

mounton

- Use as mount point.

open

- Added in 2.6.26 Kernel to control the open permission.

quotaon

- Enable quotas.

read

- Read file contents.

relabelfrom

- Change the security context based on existing type.

relabelto

- Change the security context based on the new type.

rename

- Rename file.

setattr

- Change file attributes.

unlink

- Delete file (or remove hard link).

watch

- Watch for file changes.

watch_mount

- Watch for mount changes.

watch_sb

- Watch for superblock changes.

watch_with_perm

- Allow ***fanotify****(7)* masks.

watch_reads

- Required to receive notifications from read-exclusive events.

write

- Write or append file contents.

Common Socket Permissions

The following table describes the common socket permissions that are inherited by a number of object classes.

Permissions - 21 Permissions:

accept

- Accept a connection.

append

- Write or append socket content.

bind

- Bind to a name.

connect

- Initiate a connection.

create

- Create new socket.

getattr

- Get socket information.

getopt

- Get socket options.

ioctl

- Get and set attributes via ioctl call requests.

listen

- Listen for connections.

lock

- Lock and unlock socket file descriptor.

map

- Allow a file to be memory mapped via ***mmap****(2)*.

name_bind

- *AF_INET* - Controls relationship between a socket and the port number.
- *AF_UNIX* - Controls relationship between a socket and the file.

read

- Read data from socket.

recvfrom

- Receive datagrams from socket.

relabelfrom

- Change the security context based on existing type.

relabelto

- Change the security context based on the new type.

sendto

- Send datagrams to socket.

setattr

- Change attributes.

setopt

- Set socket options.

shutdown

- Terminate connection.

write

- Write data to socket.

Common IPC Permissions

The following table describes the common IPC permissions that are inherited by a number of object classes.

Permissions - 9 Permissions:

associate

- shm - Get shared memory ID.
- msgq - Get message ID.
- sem - Get semaphore ID.

create

- Create.

destroy

- Destroy.

getattr

- Get information from IPC object.

read

- shm - Attach shared memory to process.
- msgq - Read message from queue.
- sem - Get semaphore value.

setattr

- Set IPC object information.

unix_read

- Read.

unix_write

- Write or append.

write

- shm - Attach shared memory to process.
- msgq - Send message to message queue.
- sem - Change semaphore value.

Common Capability Permissions

Permission - 32 permissions - Text from */usr/include/linux/capability.h*

audit_control

- Change auditing rules. Set login UID.

audit_write

- Send audit messages from user space.

chown

- Allow changing file and group ownership.

dac_override

- Overrides all DAC including ACL execute access.

dac_read_search

- Overrides DAC for read and directory search.

fowner

- Grant all file operations otherwise restricted due to different ownership except where *FSETID* capability is applicable. DAC and MAC accesses are not overridden.

fsetid

- Overrides the restriction that the real or effective user ID of a process sending a signal must match the real or effective user ID of the process receiving the signal.

ipc_lock

- Grants the capability to lock non-shared and shared memory segments.

ipc_owner

- Grant the ability to ignore IPC ownership checks.

kill

- Allow signal raising for any process.

lease

- Grants ability to take leases on a file.

linux_immutable

- Grant privilege to modify *S_IMMUTABLE* and *S_APPEND* file attributes on supporting filesystems.

mknod

- Grants permission to creation of character and block device nodes.

net_admin

- Allow the following: interface configuration; administration of IP firewall; masquerading and accounting; setting debug option on sockets; modification of routing tables; setting arbitrary process / group ownership on sockets; binding to any address for transparent proxying; setting TOS (type of service); setting promiscuous mode; clearing driver statistics; multicasting; read/write of device-specific registers; activation of ATM control sockets.

net_bind_service

- Allow low port binding. Port < 1024 for TCP/UDP. VCI < 32 for ATM.

net_raw

- Allows opening of raw sockets and packet sockets.

net_broadcast

- Grant network broadcasting and listening to incoming multicasts.

setfcap

- Allow the assignment of file capabilities.

setgid

- Allow ***setgid****(2)* allow ***setgroups****(2)* allow fake *gids* on credentials passed over a socket.

setpcap

- Transfer capability maps from current process to any process.

setuid

- Allow all ***setsuid****(2)* type calls including fsuid. Allow passing of forged pids on credentials passed over a socket.

sys_admin

- Allow the following: configuration of the secure attention key; administration of the random device; examination and configuration of disk quotas; configuring the kernel's syslog; setting the domainname; setting the hostname; calling bdflush(); mount() and umount(), setting up new smb connection; some autofs root ioctls; nfsservctl; VM86_REQUEST_IRQ; to read/write pci config on alpha; irix_prctl on mips (setstacksize); flushing all cache on m68k (sys_cacheflush); removing semaphores; locking/unlocking of shared memory segment; turning swap on/off; forged pids on socket credentials passing; setting readahead and flushing buffers on block devices; setting geometry in floppy driver; turning DMA on/off in xd driver; administration of md devices; tuning the ide driver; access to the nvram device; administration of apm_bios, serial and bttv (TV) device; manufacturer commands in isdn CAPI support driver; reading non-standardized portions of pci configuration space; DDI debug ioctl on sbpcd driver; setting up serial ports; sending raw qic-117 commands; enabling/disabling tagged queuing on SCSI controllers and sending arbitrary SCSI commands; setting encryption key on loopback filesystem; setting zone reclaim policy.

sys_boot

- Grant ability to reboot the system.

sys_chroot

- Grant use of the ***chroot****(2)* call.

sys_module

- Allow unrestricted kernel modification including but not limited to loading and removing kernel modules. Allows modification of kernel's bounding capability mask. See sysctl.

sys_nice

- Grants privilege to change priority of any process. Grants change of scheduling algorithm used by any process.

sys_pacct

- Allow modification of accounting for any process.

sys_ptrace

- Allow ptrace of any process.

sys_rawio

- Grant permission to use ***ioperm****(2)* and ***iopl****(2)* as well as the ability to send messages to USB devices via */proc/bus/usb*.

sys_resource

- Override the following: resource limits; quota limits; reserved space on ext2 filesystem; size restrictions on IPC message queues; max number of consoles on console allocation; max number of keymaps.
- Set resource limits.
- Modify data journaling mode on ext3 filesystem.
- Allow more than 64hz interrupts from the real-time clock.

sys_time

- Grant permission to set system time and to set the real-time lock.

sys_tty_config

- Grant permission to configure tty devices.

Common Capability2 Permissions

Permissions - 8 permissions:

audit_read

- Allow reading audits logs.

bpf

- Create maps, do other *sys_bpf()* commands and load *SK_REUSEPORT* progs. Note that loading tracing programs also requires *CAP_PERFMON* and that loading networking programs also requires *CAP_NET_ADMIN*.

block_suspend

- Prevent system suspends (was *epollwakeup*)

mac_admin

- Allow MAC configuration state changes. For SELinux allow contexts not defined in the policy to be assigned. This is called 'deferred mapping of security contexts' and is explained at: http://marc.info/?l=selinux&m=121017988131629&w=2

mac_override

- Allow MAC policy to be overridden (not used).

perfmon

- Allow system performance monitoring and observability operations.

syslog

- Allow configuration of kernel *syslog* (*printk()* behaviour).

wake_alarm

- Trigger the system to wake up.

Common Database Permissions

The following table describes the common database permissions that are inherited by the database object classes. The **Security-Enhanced PostgreSQL Security Wiki** explains the objects, their permissions and how they should be used in detail.

Permissions - 6 Permissions:

create

- Create a database object such as a *TABLE*.

drop

- Delete (*DROP*) a database object.

getattr

- Get metadata - needed to reference an object (e.g. *SELECT ... FROM ...*).

relabelfrom

- Change the security context based on existing type.

relabelto

- Change the security context based on the new type.

setattr

- Set metadata - this permission is required to update information in the database (e.g. *ALTER ...*).

Common X_Device Permissions

The following table describes the common *x_device* permissions that are inherited by the X-Windows *x_keyboard* and *x_pointer* object classes.

Permissions - 19 permissions:

add

- Unused.

bell

- Unused.

create

- Unused.

destroy

- Unused.

force_cursor

- Get window focus.

freeze

- Unused.

get_property

- Required to create a device context (source code).

getattr

- Unused.

getfocus

- Unused.

grab

- Set window focus.

list_property

- Unused.

manage

- Unused.

read

- Unused.

remove

- Unused.

set_property

- Unused.

setattr

- TBC

setfocus

- Unused.

use

- Unused.

write

- Unused.

File Object Classes

filesystem

A mounted *filesystem*

Permissions - 10 unique permissions:

associate

- Use type as label for file.

getattr

- Get file attributes.

mount

- Mount filesystem.

quotaget

- Get quota information.

quotamod

- Modify quota information.

relabelfrom

- Change the security context based on existing type.

relabelto

- Change the security context based on the new type.

remount

- Remount existing mount.

unmount

- Unmount filesystem.

watch

- Watch for filesystem changes.

dir

A Directory

Permissions - Inherit 25 **Common File Permissions** + 5 unique:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

add_name

- Add entry to the directory.

remove_name

- Remove an entry from the directory.

reparent

- Change parent directory.

rmdir

- Remove directory.

search

- Search directory.

file

Ordinary file

Permissions - Inherit 25 **Common File Permissions** + 2 unique:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

entrypoint

- Entry point permission for a domain transition.

execute_no_trans

- Execute in the caller's domain (i.e. no domain transition).

lnk_file

Symbolic links

Permissions - Inherit 25 **Common File Permissions**:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

chr_file

Character files

Permissions - Inherit 25 **Common File Permissions**:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

blk_file

Block files

Permissions - Inherit 25 **Common File Permissions**:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

sock_file

UNIX domain sockets

Permissions - Inherit 25 **Common File Permissions**:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

fifo_file

Named pipes

Permissions - Inherit 25 **Common File Permissions**:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

anon_inode

Control anonymous-inode files via the kernel *anon_inode_getfd_secure()* function. Policy controls anonymous inodes by adding a name-based ***type_transition*** rule that assigns a *type* to anonymous-inode files created in a domain. The *name* used for the name-based transition is the name associated with the anonymous inode for file listings, for example:

```
type uffd_t;
type_transition sysadm_t sysadm_t : anon_inode uffd_t "[userfaultfd]";
allow sysadm_t uffd_t:anon_inode { create };
```

The current implementations that make use of this service are:

- ***userfaultfd****(2)* (from kernel 5.12) described in the patch series https://lore.kernel.org/selinux/20210108222223.952458-1-lokeshgidra@google.com/
- ***io_uring****(7)* (from kernel 5.16) described in the patch series https://lore.kernel.org/all/163172459001.88001.17463922586800990358.stgit@olly/
- ***memfd_secret****(2)* (from kernel 6.0) described in the patch https://lore.kernel.org/linux-mm/20220125143304.34628-1-cgzones@googlemail.com/

Permissions - Inherit 25 **Common File Permissions**:

- *append*, *audit_access*, *create*, *execute*, *execmod*, *getattr*, *ioctl*, *link*, *lock*, *map*, *mounton*, *open*, *quotaon*, *read*, *relabelfrom*, *relabelto*, *rename*, *setattr*, *unlink*, *watch*, *watch_mount*, *watch_sb*, *watch_with_perm*, *watch_reads*, *write*

fd

File descriptors

Permissions - 1 unique permission:

use

- Inherit *fd* when process is executed and domain has been changed.
- Receive *fd* from another process by Unix domain socket.
- Get and set attribute of *fd*.

Network Object Classes

node

IP address or range of IP addresses, used when peer labeling is configured.

Permissions - 2 unique permissions:

recvfrom

- Network interface and address check permission for use with the *ingress* permission.

sendto

- Network interface and address check permission for use with the *egress* permission.

netif

Network Interface (e.g. eth0) used when peer labeling is configured.

Permissions - 2 unique permissions:

egress

- Each packet leaving the system must pass an *egress* access control. Also requires the *node sendto* permission.

ingress

- Each packet entering the system must pass an *ingress* access control. Also requires the *node recvfrom* permission.

socket

Socket that is not part of any other specific SELinux socket object class.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

tcp_socket

Protocol: *PF_INET*, *PF_INET6* Family Type: *SOCK_STREAM*

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

name_connect

- Connect to a specific port type.

node_bind

- Bind to a node.

udp_socket

Protocol: *PF_INET*, *PF_INET6* Family Type: *SOCK_DGRAM*

Permissions - Inherit 21 **Common Socket Permissions** + 1 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

node_bind

- Bind to a node.

rawip_socket

Protocol: *PF_INET*, *PF_INET6* Family Type: *SOCK_RAW*

Permissions - Inherit 21 **Common Socket Permissions** + 1 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

node_bind

- Bind to a node.

packet_socket

Protocol: *PF_PACKET* Family Type: All.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

unix_stream_socket

Communicate with processes on same machine. Protocol: *PF_STREAM* Family Type: *SOCK_STREAM*

Permissions - Inherit 21 **Common Socket Permissions** + 1 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

connectto

- Connect to server socket.

unix_dgram_socket

Communicate with processes on same machine. Protocol: *PF_STREAM* Family Type: *SOCK_DGRAM*

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

tun_socket

TUN is Virtual Point-to-Point network device driver to support IP tunneling.

Permissions - Inherit 21 **Common Socket Permissions** + 1 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

attach_queue

- Attach to an interface queue.

IPSec Network Object Classes

association

IPSec security association

Permissions - 4 unique permissions:

polmatch

- Match IPSec Security Policy Database (SPD) context (-ctx) entries to an SELinux domain (contained in the Security Association Database (SAD)).

recvfrom

- Receive from an IPSec association.

sendto

- Send to an IPSec association.

setcontext

- Set the context of an IPSec association on creation.

key_socket

IPSec key management. Protocol: *PF_KEY* Family Type: All

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_xfrm_socket

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

nlmsg_read

- Get IPSec configuration information.

nlmsg_write

- Set IPSec configuration information.

Netlink Object Classes

Netlink sockets communicate between userspace and the kernel - also see ***netlink****(7)*.

netlink_socket

Netlink socket that is not part of any specific SELinux Netlink socket class. Protocol: *PF_NETLINK* Family Type: All other types that are not part of any other specific netlink object class.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_route_socket

Netlink socket to manage and control network resources.

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

nlmsg_read

- Read kernel routing table.

nlmsg_write

- Write to kernel routing table.

netlink_firewall_socket (Deprecated)

Netlink socket for firewall filters.

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

nlmsg_read

- Read netlink message.

nlmsg_write

- Write netlink message.

netlink_tcpdiag_socket

Netlink socket to monitor TCP connections.

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

nlmsg_read

- Request information about a protocol.

nlmsg_write

- Write netlink message.

netlink_nflog_socket

Netlink socket for Netfilter logging

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_selinux_socket

Netlink socket to receive SELinux events such as a policy or boolean change.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_audit_socket

Netlink socket for audit service.

Permissions - Inherit 21 **Common Socket Permissions** + 5 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

nlmsg_read

- Query status of audit service.

nlmsg_readpriv

- List auditing configuration rules.

nlmsg_relay

- Send userspace audit messages to theaudit service.

nlmsg_tty_audit

- Control TTY auditing.

nlmsg_write

- Update audit service configuration.

netlink_ip6fw_socket (Deprecated)

Netlink socket for IPv6 firewall filters.

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

nlmsg_read

- Read netlink message.

nlmsg_write

- Write netlink message.

netlink_dnrt_socket

Netlink socket for DECnet routing

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_kobject_uevent_socket

Netlink socket to send kernel events to userspace.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_iscsi_socket

Netlink socket to support RFC 3720 - Internet Small Computer Systems Interface (iSCSI).

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_fib_lookup_socket

Netlink socket to Forwarding Information Base services.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_connector_socket

Netlink socket to support kernel connector services.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_netfilter_socket

Netlink socket netfilter services.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_generic_socket

Simplified netlink socket.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_scsitransport_socket

SCSI transport netlink socket.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_rdma_socket

Remote Direct Memory Access netlink socket.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

netlink_crypto_socket

Kernel crypto API netlink socket.

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

Miscellaneous Network Object Classes

peer

NetLabel and Labeled IPsec have separate access controls, the network peer label consolidates these two access controls into a single one (see http://paulmoore.livejournal.com/1863.html for details).

Permissions - 1 unique permission:

recv

- Receive packets from a labeled networking peer.

packet

Supports *secmark* services where packets are labeled using iptables to select and label packets, SELinux then enforces policy using these packet labels.

Permissions - 5 unique permissions:

forward_in

- Allow inbound forwarded packets.

forward_out

- Allow outbound forwarded packets.

recv

- Receive inbound locally consumed packets.

relabelto

- Control how domains can apply specific labels to packets.

send

- Send outbound locally generated packets.

appletalk_socket

Appletalk socket

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

dccp_socket

Datagram Congestion Control Protocol (DCCP)

Permissions - Inherit 21 **Common Socket Permissions** + 2 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

name_connect

- Allow DCCP name ***connect****(2)*.

node_bind

- Allow DCCP ***bind****(2)*.

Sockets via *extended_socket_class*

These socket classes that were introduced by the *extended_socket_class* policy capability in kernel version 4.16.

sctp_socket

Stream Control Transmission Protocol (SCTP)

Permissions - Inherit 21 **Common Socket Permissions** + 3 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

association

- Allow an SCTP association.

name_connect

- Allow SCTP ***connect****(2)* and ***connectx****(3)*.

node_bind

- Allow SCTP ***bind****(2)* and ***bindx****(3)*.

icmp_socket

Internet Control Message Protocol (ICMP)

Permissions - Inherit 21 **Common Socket Permissions** + 1 unique:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

node_bind

- Allow ICMP ***bind****(2)*.

Miscellaneous Extended Socket Classes

Class:

- *ax25_socket* - Amateur X.25
- *ipx_socket* - Internetwork Packet Exchange
- *netrom_socket* - Part of Amateur X.25
- *atmpvc_socket* - Asynchronous Transfer Mode Permanent Virtual Circuit
- *x25_socket* - X.25
- *rose_socket* - Remote Operations Service Element
- *decnet_socket* - Everyone knows this
- *atmsvc_socket* - Asynchronous Transfer Mode Switched Virtual Circuit
- *rds_socket* - Remote Desktop Protocol
- *irda_socket* - Infrared Data Association (IrDA)
- *pppox_socket* - Point-to-Point Protocol over Ethernet/ATM ...
- *llc_socket* - Link Level Control
- *can_socket* - Controller Area Network
- *tipc_socket* - Transparent Inter Process Communication
- *bluetooth_socket*
- *iucv_socket* - Inter User Communication Vehicle
- *rxrpc_socket* - A reliable two phase transport
- *isdn_socket* - Integrated Services Digital Network
- *phonet_socket* - packet protocol used by Nokia cellular modems
- *ieee802154_socket* - For low-rate wireless personal area networks (LR-WPANs).
- *caif_socket* - Communication CPU to Application CPU Interface
- *alg_socket* - algorithm interface
- *nfc_socket* - Near Field Communications
- *vsock_socket* - Virtual Socket protocol
- *kcm_socket* - Kernel Connection Multiplexor
- *qipcrtr_socket* - communicating with services running on co-processors in Qualcomm platforms
- *smc_socket* - Shared Memory Communications
- *xdp_socket* - eXpress Data Path

Permissions - Inherit 21 **Common Socket Permissions**:

- *accept*, *append*, *bind*, *connect*, *create*, *getattr*, *getopt*, *ioctl*, *listen*, *lock*, *map*, *name_bind*, *read*, *recvfrom*, *relabelfrom*, *relabelto*, *sendto*, *setattr*, *setopt*, *shutdown*, *write*

BPF Object Class

bpf

Support for extended Berkeley Packet Filters ***bpf****(2)*

Permissions - 5 unique permissions:

map_create

- Create map

map_read

- Read map

map_write

- Write to map

prog_load

- Load program

prog_run

- Run program

Performance Event Object Class

perf_event

Control ***perf****(1)* events

Permissions - 6 unique permissions:

cpu

- Monitor cpu

kernel

- Monitor kernel

open

- Open a perf event

read

- Read perf event

tracepoint

- Set tracepoints

write

- Write a perf event

Lockdown Object Class

The *lockdown* class and associated SELinux LSM hook (added in kernel 5.6), have been removed from kernel 5.16 for the reasons discussed in https://lore.kernel.org/selinux/163292547664.17566.8479687865641275719.stgit@olly/.

lockdown (Deprecated)

Stop userspace extracting/modify kernel data.

Permissions - 2 unique permissions:

confidentiality

- Kernel features that allow userspace to extract confidential information from the kernel are disabled.

integrity

- Kernel features that allow userspace to modify the running kernel are disabled.

IPC Object Classes

ipc (Deprecated)

Interprocess communications

Permissions - Inherit 9 **Common IPC Permissions**:

- *associate*, *create*, *destroy*, *getattr*, *read*, *setattr*, *unix_read*, *unix_write*, *write*

sem

Semaphores

Permissions - Inherit 9 **Common IPC Permissions**:

- *associate*, *create*, *destroy*, *getattr*, *read*, *setattr*, *unix_read*, *unix_write*, *write*

msgq

IPC Message queues

Permissions - Inherit 9 **Common IPC Permissions** + 1 unique:

- *associate*, *create*, *destroy*, *getattr*, *read*, *setattr*, *unix_read*, *unix_write*, *write*

enqueue

- Send message to message queue.

msg

Message in a queue

Permissions - Inherit 9 **Common IPC Permissions** + 2 unique:

- *associate*, *create*, *destroy*, *getattr*, *read*, *setattr*, *unix_read*, *unix_write*, *write*

receive

- Read (and remove) message from queue.

send

- Add message to queue.

shm

Shared memory segment

Permissions - Inherit 9 **Common IPC Permissions** + 1 unique:

- *associate*, *create*, *destroy*, *getattr*, *read*, *setattr*, *unix_read*, *unix_write*, *write*

lock

- Lock or unlock shared memory.

Process Object Class

process

An object is instantiated for each process created by the system.

Permissions - 31 unique permissions:

dyntransition

- Dynamically transition to a new context using ***setcon****(3)*.

execheap

- Make the heap executable.

execmem

- Make executable an anonymous mapping or private file mapping that is writable.

execstack

- Make the main process stack executable.

fork

- Create new process using ***fork****(2)*.

getattr

- Get process security information.

getcap

- Get Linux capabilities of process.

getpgid

- Get group Process ID of another process.

getsched

- Get scheduling information of another process.

getsession

- Get session ID of another process.

getrlimit

- Get process rlimit information.

noatsecure

- Disable secure mode environment cleansing.

ptrace

- Trace program execution of parent (***ptrace****(2)*).

rlimitinh

- Inherit rlimit information from parent process.

setcap

- Set Linux capabilities of process.

setcurrent

- Set the current process context.

setexec

- Set security context of executed process by ***setexecon****(3)*.

setfscreate

- Set security context by ***setfscreatecon****(3)*.

setkeycreate

- Set security context by ***setkeycreatecon****(3)*.

setpgid

- Set group Process ID of another process.

setrlimit

- Change process rlimit information.

setsched

- Modify scheduling information of another process.

setsockcreate

- Set security context by ***setsockcreatecon****(3)*.

share

- Allow state sharing with cloned or forked process.

sigchld

- Send *SIGCHLD* signal.

siginh

- Inherit signal state from parent process.

sigkill

- Send *SIGKILL* signal.

signal

- Send a signal other than *SIGKILL*, *SIGSTOP*, or *SIGCHLD*.

signull

- Test for exisitence of another process without sending a signal.

sigstop

- Send *SIGSTOP* signal

transition

- Transition to a new context on *exec()*.

process2

Extension of *process* class.

Permissions - 2 unique permissions:

nnp_transition

- Enables SELinux domain transitions to occur under *no_new_privs* (*NNP*).

nosuid_transition

- Enables SELinux domain transitions to occur on *nosuid* ***mount****(8)*

Security Object Class

security

This is the security server object and there is only one instance of this object (for the SELinux security server).

Permissions - 13 unique permissions:

check_context

- Determine whether the context is valid by querying the security server.

compute_av

- Compute an access vector given a source, target and class.

compute_create

- Determine context to use when querying the security server about a transition rule (*type_transition*).

compute_member

- Determine context to use when querying the security server about a membership decision (*type_member* for a polyinstantiated object).

compute_relabel

- Determines the context to use when querying the security server about a relabeling decision (*type_change*).

compute_user

- Determines the context to use when querying the security server about a user decision (*user*).

load_policy

- Load the security policy into the kernel (the security server).

read_policy

- Read the kernel policy to userspace.

setbool

- Change a boolean value within the active policy.

setcheckreqprot (deprecated)

- Set if SELinux will check original protection mode or modified protection mode (read-implies-exec) for *mmap* / *mprotect*.

setenforce

- Change the enforcement state of SELinux (permissive or enforcing).

setsecparam

- Set kernel access vector cache tuning parameters.

validate_trans

- Compute a *validatetrans* rule.

System Operation Object Class

Note that while this is defined as a kernel object class, the userspace ***systemd****(1)* has hitched a ride.

system

This is the overall system object and there is only one instance of this object.

Permissions - 6 unique permissions:

ipc_info

- Get info about an IPC object.

module_load

- Required permission when reading a file that is a 'kernel module'. See http://marc.info/?l=selinux&m=145988689809307&w=2 for an example.

module_request

- Request the kernel to load a module.

syslog_console

- Control output of kernel messages to the console with ***syslog****(2)*.

syslog_mod

- Clear kernel message buffer with ***syslog****(2)*.

syslog_read

- Read kernel message with ***syslog****(2)*.

User-space Permissions - 8 unique added for use by ***systemd****(1)*:

disable

- Unused.

enable

- Enable default target/file.

halt

- Allow systemd to close down.

reboot

- Allow reboot by system manager.

reload

- Allow reload.

stop

- Unused.

start

- Unused.

status

- Obtain systemd status

Miscellaneous Kernel Object Classes

kernel_service

Used to add kernel services.

Permissions - 2 unique permissions:

use_as_override

- Grant a process the right to nominate an alternate process SID for the kernel to use as an override for the SELinux subjective security when accessing information on behalf of another process. For example, *CacheFiles* when accessing the cache on behalf of a process accessing an NFS file needs to use a subjective security ID appropriate to the cache rather than the one the calling process is using. The *cachefilesd* daemon will nominate the security ID to be used.

create_files_as

- Grant a process the right to nominate a file creation label for a kernel service to use.

key

Manage Keyrings.

Permission - 7 unique permissions:

create

- Create a keyring.

link

- Link a key into the keyring.

read

- Read a keyring.

search

- Search a keyring.

setattr

- Change permissions on a keyring.

view

- View a keyring.

write

- Add a key to the keyring.

memprotect

Protect lower memory blocks.

Permission - 1 unique permission:

mmap_zero

- Security check on mmap operations to see if the user is attempting to ***mmap****(2)* to low area of the address space. The amount of space protected is indicated by a proc tunable (*/proc/sys/vm/mmap_min_addr*). Setting this value to 0 will disable the checks. The http://eparis.livejournal.com/891.html describes additional checks that will be added to the kernel to protect against some kernel exploits (by requiring *CAP_SYS_RAWIO* (root) and the SELinux *memprotect* / *mmap_zero* permission instead of only one or the other).

binder

Manage the Binder IPC service.

Permission - 4 unique permissions:

call

- Perform a binder IPC to a given target process (can A call B?).

impersonate

- Perform a binder IPC on behalf of another process (can A impersonate B on an IPC).

set_context_mgr

- Register self as the Binder Context Manager aka *servicemanager* (global name service). Can A set the context manager to B, where normally A == B.

transfer

- Transfer a binder reference to another process (can A transfer a binder reference to B?).

user_namespace

Manage user namespaces.

Permission - 1 unique permission:

create

- Create a new user namespace.

io_uring

Manage security-sensitive usages of the *io_uring* subsystem.

Permission - 3 unique permissions:

override_creds

- Use another process's credentials via an *io_uring* personality (Can A use B's credentials when submitting a new *io_uring* operation). Personalities are explained in ***io_uring_register****(2)* and can be thought of as credential references, which allow a caller to store a copy of their credentials, including SELinux labels, in an *io_uring* for use by other processes.

sqpoll

- Use an *io_uring* asynchronous polling thread.

cmd

- Use *IORING_OP_URING_CMD* on a given file.

Capability Object Classes

capability

Used to manage the Linux capabilities granted to root processes.

Permissions - Inherit 32 **Common Capability Permissions**:

- *audit_write*, *chown*, *dac_override*, *dac_read_search*, *fowner*, *fsetid*, *ipc_lock*, *ipc_owner*, *kill*, *lease*, *linux_immutable*, *mknod*, *net_admin*, *net_bind_service*, *net_raw*, *netbroadcast*, *setfcap*, *setgid*, *setpcap*, *setuid*, *sys_admin*, *sys_boot*, *sys_chroot*, *sys_module*, *sys_nice*, *sys_pacct*, *sys_ptrace*, *sys_rawio*, *sys_resource*, *sys_time*, *sys_tty_config*

capability2

Extended *capability* class.

Permissions - Inherit 8 **Common Capability2 Permissions**:

- *audit_read*, *bpf*, *block_suspend*, *mac_admin*, *mac_override*, *perfmon*, *syslog*, *wake_alarm*

cap_userns

Used to manage the Linux capabilities granted to namespace processes.

Permissions - Inherit 32 **Common Capability Permissions**:

- *audit_write*, *chown*, *dac_override*, *dac_read_search*, *fowner*, *fsetid*, *ipc_lock*, *ipc_owner*, *kill*, *lease*, *linux_immutable*, *mknod*, *net_admin*, *net_bind_service*, *net_raw*, *netbroadcast*, *setfcap*, *setgid*, *setpcap*, *setuid*, *sys_admin*, *sys_boot*, *sys_chroot*, *sys_module*, *sys_nice*, *sys_pacct*, *sys_ptrace*, *sys_rawio*, *sys_resource*, *sys_time*, *sys_tty_config*

cap2_userns

Extended *cap_userns* class.

Permissions - Inherit 8 **Common Capability2 Permissions**:

- *audit_read*, *bpf*, *block_suspend*, *mac_admin*, *mac_override*, *perfmon*, *syslog*, *wake_alarm*

InfiniBand Object Classes

Support for Mellanox InfiniBand smart adapters, see: https://www.mellanox.com/products/infiniband-adapter-cards

infiniband_pkey

Manage partition keys.

Permissions - 1 unique permission:

access

- Access one or more partition keys based on their subnet.

infiniband_endport

Manage packets.

Permissions - 1 unique permission:

manage_subnet

- Allow send and receive of subnet management packets on the end port specified by the device name and port.

Userspace Object Classes

X Windows Object Classes

These are userspace objects managed by XSELinux.

x_drawable

The drawable parameter specifies the area into which the text will be drawn. It may be either a pixmap or a window. Some of the permission information has been extracted from an http://marc.info/?l=selinux&m=121485496531386&q=raw email describing them in terms of an MLS system.

Permissions - 19 unique permissions:

add_child

- Add new window. Normally SystemLow for MLS systems.

blend

- There are two cases:
  1. Allow a non-root window to have a transparent background.
  2. The application is redirecting the contents of the window and its sub-windows into a memory buffer when using the Composite extension. Only *SystemHigh* processes should have the blend permission on the root window.

create

- Create a drawable object. Not applicable to the root windows as it cannot be created.

destroy

- Destroy a drawable object. Not applicable to the root windows as it cannot be destroyed.

get_property

- Read property information. Normally *SystemLow* for MLS systems.

getattr

- Get attributes from a drawable object. Most applications will need this so *SystemLow*.

hide

- Hide a drawable object. Not applicable to the root windows as it cannot be hidden.

list_child

- Allows all child window IDs to be returned. From the root window it will show the client that owns the window and their stacking order. If hiding this information is required then processes should be *SystemHigh*.

list_property

- List property associated with a window. Normally *SystemLow* for MLS systems.

manage

- Required to create a context, move and resize windows. Not applicable to the root windows as it cannot be resized etc.

override

- Allow setting the *override-redirect* bit on the window. Not applicable to the root windows as it cannot be overridden.

read

- Read window contents. Note that this will also give read permission to all child windows, therefore (for MLS), only *SystemHigh* processes should have read permission on the root window.

receive

- Allow receiving of events. Normally *SystemLow* for MLS systems (but could leak information between clients running at different levels, therefore needs investigation).

remove_child

- Remove child window. Normally *SystemLow* for MLS systems.

send

- Allow sending of events. Normally *SystemLow* for MLS systems (but could leak information between clients running at different levels, therefore needs investigation).

set_property

- Set property. Normally SystemLow for MLS systems (but could leak information between clients running at different levels, therefore needs investigation. Polyinstantiation may be required).

setattr

- Allow window attributes to be set. This permission protects operations on the root window such as setting the background image or colour, setting the colormap and setting the mouse cursor to display when the cursor is in the window, therefore only *SystemHigh* processes should have the *setattr* permission.

show

- Show window. Not applicable to the root windows as it cannot be hidden.

write

- Draw within a window. Note that this will also give write permission to all child windows, therefore (for MLS), only *SystemHigh* processes should have *write* permission on the root window.

x_screen

The specific screen available to the display (X-server) (*hostname:display_number.screen*)

Permissions - 8 unique permissions:

getattr

- TBC

hide_cursor

- TBC

saver_getattr

- TBC

saver_hide

- TBC

saver_setattr

- TBC

saver_show

- TBC

setattr

- TBC

show_cursor

- TBC

x_gc

The graphics contexts allows the X-server to cache information about how graphics requests should be interpreted. It reduces the network traffic.

Permissions - 5 unique permissions:

create

- Create Graphic Contexts object.

destroy

- Free (dereference) a Graphics Contexts object.

getattr

- Get attributes from Graphic Contexts object.

setattr

- Set attributes for Graphic Contexts object.

use

- Allow GC contexts to be used.

x_font

An X-server resource for managing the different fonts.

Permissions - 6 unique permissions:

add_glyph

- Create glyph for cursor

create

- Load a font.

destroy

- Free a font.

getattr

- Obtain font names, path, etc.

remove_glyph

- Free glyph

use

- Use a font.

x_colormap

An X-server resource for managing colour mapping. A new colormap can be created using *XCreateColormap*.

Permissions - 10 unique permissions:

add_color

- Add a colour.

create

- Create a new Colormap.

destroy

- Free a Colormap.

getattr

- Get the color gamut of a screen.

install

- Copy a virtual colormap into the display hardware.

read

- Read color cells of colormap.

remove_color

- Remove a colour

uninstall

- Remove a virtual colormap from the display hardware.

use

- Use a colormap.

write

- Change color cells in colormap.

x_property

An InterClient Communications (ICC) service where each property has a name and ID (or Atom). Properties are attached to windows and can be uniquely identified by the *windowID* and *propertyID*. XSELinux supports polyinstantiation of properties.

Permissions - 7 unique permissions:

append

- Append a property.

create

- Create property object.

destroy

- Free (dereference) a property object.

getattr

- Get attributes of a property.

read

- Read a property.

setattr

- Set attributes of a property.

write

- Write a property.

x_selection

An InterClient Communications (ICC) service that allows two parties to communicate about passing information. The information uses properties to define the format (e.g. whether text or graphics). XSELinux supports polyinstantiation of selections.

Permissions - 4 unique permissions:

getattr

- Get selection owner (*XGetSelectionOwner*).

read

- Read the information from the selection owner

setattr

- Set the selection owner (*XSetSelectionOwner*).

write

- Send the information to the selection requestor.

x_cursor

The cursor on the screen

Permissions - 7 unique permissions:

create

- Create an arbitrary cursor object.

destroy

- Free (dereference) a cursor object.

getattr

- Get attributes of the cursor.

read

- Read the cursor.

setattr

- Set attributes of the cursor.

use

- Associate a cursor object with a window.

write

- Write a cursor.

x_client

The X-client connecting to the X-server.

Permissions - 4 unique permissions:

destroy

- Close down a client.

getattr

- Get attributes of X-client.

manage

- Required to create an X-client context.

setattr

- Set attributes of X-client.

x_device

These are any other devices used by the X-server as the keyboard and pointer devices have their own object classes.

Permissions - Inherit 19 **Common X Device Permissions**:

- *add*, *bell*, *create*, *destroy*, *force_cursor*, *freeze*, *get_property*, *getattr*, *getfocus*, *grab*, *list_property*, *manage*, *read*, *remove*, *set_property*, *setattr*, *setfocus*, *use*, *write*

x_server

The X-server that manages the display, keyboard and pointer.

Permissions - 6 unique permissions:

debug

- TBC

getattr

- TBC

grab

- TBC

manage

- Required to create a context. (source code)

record

- TBC

setattr

- TBC

x_extension

An X-Windows extension that can be added to the X-server (such as the XSELinux object manager itself).

Permissions - 2 unique permissions:

query

- Query for an extension.

use

- Use the extensions services.

x_resource

These consist of Windows, Pixmaps, Fonts, Colormaps etc. that are classed as resources.

Permissions - 2 unique permissions:

read

- Allow reading a resource.

write

- Allow writing to a resource.

x_event

Manage X-server events.

Permissions - 2 unique permissions:

receive

- Receive an event

send

- Send an event

x_synthetic_event

Manage some X-server events (e.g. *configNotify*). Note the *x_event* permissions will still be required.

Permissions - 2 unique permissions:

receive

- Receive an event

send

- Send an event

x_application_data

Not specifically used by XSELinux, however is used by userspace applications that need to manage copy and paste services (such as the *CUT_BUFFER*s).

Permission - 3 unique permissions:

copy

- Copy the data

paste

- Paste the data

paste_after_confirm

- Need to confirm that the paste is allowed.

x_pointer

The mouse or other pointing device managed by the X-server.

Permissions - Inherit 19 **Common X_Device Permissions**:

- *add*, *bell*, *create*, *destroy*, *force_cursor*, *freeze*, *get_property*, *getattr*, *getfocus*, *grab*, *list_property*, *manage*, *read*, *remove*, *set_property*, *setattr*, *setfocus*, *use*, *write*

x_keyboard

The keyboard managed by the X-server.

Permissions - Inherit 19 **Common X_Device Permissions**:

- *add*, *bell*, *create*, *destroy*, *force_cursor*, *freeze*, *get_property*, *getattr*, *getfocus*, *grab*, *list_property*, *manage*, *read*, *remove*, *set_property*, *setattr*, *setfocus*, *use*, *write*

Database Object Classes

These are userspace objects - The PostgreSQL database supports these with their SE-PostgreSQL database extension. The "**Security-Enhanced PostgreSQL Security Wiki**" explains the objects, their permissions and how they should be used in detail.

db_database

Manage a database.

Permissions - Inherit 6 **Common Database Permissions** + 3 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

access

- Required to connect to the database - this is the minimum permission required by an SE-PostgreSQL client.

install_module

- Required to install a dynamic link library.

load_module

- Required to load a dynamic link library.

db_table

Manage a database table.

Permissions - Inherit 6 **Common Database Permissions** + 5 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

delete

- Required to delete from a table with a *DELETE* statement, or when removing the table contents with a *TRUNCATE* statement.

insert

- Required to insert into a table with an *INSERT* statement, or when restoring it with a *COPY FROM* statement.

lock

- Required to get a table lock with a *LOCK* statement.

select

- Required to refer to a table with a *SELECT* statement or to dump the table contents with a *COPY TO* statement.

update

- Required to update a table with an *UPDATE* statement.

db_schema

Manage a database schema.

Permissions - Inherit 6 **Common Database Permissions** + 3 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

search

- Search for an object in the schema.

add_name

- Add an object to the schema.

remove_name

- Remove an object from the schema.

db_procedure

Manage a database procedure.

Permissions - Inherit 6 **Common Database Permissions** + 3 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

entrypoint

- Required for any functions defined as Trusted Procedures.

execute

- Required for functions executed with SQL queries.

install

- Install a procedure.

db_column

Manage a database column.

Permissions - Inherit 6 **Common Database Permissions** + 3 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

insert

- Required to insert a new entry using the *INSERT* statement.

select

- Required to reference columns.

update

- Required to update a table with an *UPDATE* statement.

db_tuple

Manage a database tuple.

Permissions - 7 unique:

delete

- Required to delete entries with a *DELETE* or *TRUNCATE* statement.

insert

- Required when inserting a entry with an *INSERT* statement, or restoring tables with a *COPY FROM* statement.

relabelfrom

- The security context of an entry can be changed with an *UPDATE* to the *security_context* column at which time *relabelfrom* and *relabelto* permission is evaluated. The client must have *relabelfrom* permission to the security context before the entry is changed, and *relabelto* permission to the security context after the entry is changed.

relabelto

- See *relabelfrom*.

select

- Required when: reading entries with a *SELECT* statement, returning entries that are subjects for updating queries with a *RETURNING* clause, or dumping tables with a *COPY TO* statement. Entries that the client does not have *select* permission on will be filtered from the result set.

update

- Required when updating an entry with an *UPDATE* statement. Entries that the client does not have update permission on will not be updated.

use

- Controls usage of system objects that require permission to “use” objects such as data types, tablespaces and operators.

db_blob

Manage a database blob.

Permissions - Inherit 6 **Common Database Permissions** + 4 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

export

- Export a binary large object by calling the *lo_export()* function.

import

- Import a file as a binary large object by calling the *lo_import()* function.

read

- Read a binary large object the *loread()* function.

write

- Write a binary large object with the *lowrite()* function.

db_view

Manage a database view.

Permissions - Inherit 6 **Common Database Permissions** + 1 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

expand

- Allows the expansion of a 'view'.

db_sequence

A sequential number generator.

Permissions - Inherit 6 **Common Database Permissions** + 3 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

get_value

- Get a value from the sequence generator object.

next_value

- Get and increment value.

set_value

- Set an arbitrary value.

db_language

Support for script languages such as Perl and Tcl for SQL Procedures

Permissions - Inherit 6 **Common Database Permissions** + 2 unique:

- *create*, *drop*, *getattr*, *relabelfrom*, *relabelto*, *setattr*

implement

- Whether the language can be implemented or not for the SQL procedure.

execute

- Allow the execution of a code block using a *DO* statement.

Miscellaneous Userspace Object Classes

passwd

Controlling changes to passwd information.

Permissions - 5 unique permissions:

chfn

- Change another users finger info.

chsh

- Change another users shell.

crontab

- crontab another user.

passwd

- Change another users passwd.

rootok

- *pam_rootok* check - skip authentication.

nscd

Manage the Name Service Cache Daemon.

Permissions - 12 unique permissions:

admin

- Allow the ***nscd****(8)* daemon to be shut down.

getgrp

- Get group information.

gethost

- Get host information.

getnetgrp

- TBC

getpwd

- Get password information.

getserv

- Get ?? information.

getstat

- Get the AVC stats from the nscd daemon.

shmemgrp

- Get shmem group file descriptor.

shmemhost

- Get shmem host descriptor. ??

shmemnetgrp

- TBC

shmempwd

- TBC

shmemserv

- TBC

dbus

Manage the D-BUS Messaging service that is required to run various services.

Permissions - 2 unique permissions:

acquire_svc

- Open a virtual circuit (communications channel).

send_msg

- Send a message.

context

These permissions are used for SELinux configuration file context entries and context translations for MCS/MLS policy.

Permissions - 2 unique permissions:

contains

- Check configuration file contains a valid context entry.

translate

- Translate a raw label to a meaningful text string.

service

Manage ***systemd****(1)* services.

Permissions - 8 unique permissions:

disable

- Disable services.

enable

- Enable services.

kill

- Kill services.

load

- Load services

reload

- Restart *systemd* services.

start

- Start *systemd* services.

status

- Read service status.

stop

- Stop *systemd* services.

proxy

Manages ***gssd****(8)* services.

Permissions - 1 unique permission:

read

- Read credentials.

Appendix B - *libselinux* API Summary

These functions have been taken from the following header files of *libselinux* version 3.0:

- */usr/include/selinux/avc.h*
- */usr/include/selinux/context.h*
- */usr/include/selinux/get_context_list.h*
- */usr/include/selinux/get_default_type.h*
- */usr/include/selinux/label.h*
- */usr/include/selinux/restorecon.h*
- */usr/include/selinux/selinux.h*

The appropriate ***man****(3)* pages should consulted for detailed usage.

Some useful notes:

1. Use ***free****(3)* instead of ***freecon****(3)* for context strings as they are now defined as *char**. Note this does not apply to ***context_free****(3)*.
2. There must be more ???

avc_add_callback - avc.h

Register a callback for security events.

avc_audit - avc.h

Audit the granting or denial of permissions in accordance with the policy. This function is typically called by ***avc_has_perm****(3)* after a permission check, but can also be called directly by callers who use ***avc_has_perm_noaudit****(3)* in order to separate the permission check from the auditing. For example, this separation is useful when the permission check must be performed under a lock, to allow the lock to be released before calling the auditing code.

avc_av_stats - avc.h

Log AV table statistics. Logs a message with information about the size and distribution of the access vector table. The audit callback is used to print the message.

avc_cache_stats - avc.h

Get cache access statistics. Fill the supplied structure with information about AVC activity since the last call to ***avc_init****(3)* or ***avc_reset****(3)*.

avc_cleanup - avc.h

Remove unused SIDs and AVC entries. Search the SID table for SID structures with zero reference counts, and remove them along with all AVC entries that reference them. This can be used to return memory to the system.

avc_compute_create - avc.h

Compute SID for labeling a new object. Call the security server to obtain a context for labeling a new object. Look up the context in the SID table, making a new entry if not found.

avc_compute_member - avc.h

Compute SID for polyinstantation. Call the security server to obtain a context for labeling an object instance. Look up the context in the SID table, making a new entry if not found.

avc_context_to_sid, avc_context_to_sid_raw - avc.h

Get SID for context. Look up security context *ctx* in SID table, making a new entry if *ctx* is not found. Store a pointer to the SID structure into the memory referenced by *sid*, returning 0 on success or -1 on error with *errno* set.

avc_destroy - avc.h

Free all AVC structures. Destroy all AVC structures and free all allocated memory. User-supplied locking, memory, and audit callbacks will be retained, but security-event callbacks will not. All SID's will be invalidated. User must call ***avc_init****(3)* if further use of AVC is desired.

avc_entry_ref_init - avc.h

Initialize an AVC entry reference. Use this macro to initialize an *avc* entry reference structure before first use. These structures are passed to ***avc_has_perm****(3)*, which stores cache entry references in them. They can increase performance on repeated queries.

avc_get_initial_sid - avc.h

Get SID for an initial kernel security identifier. Get the context for an initial kernel security identifier specified by name using ***security_get_initial_context****(3)* and then call ***avc_context_to_sid****(3)* to get the corresponding SID.

avc_has_perm - avc.h

Check permissions and perform any appropriate auditing. Check the AVC to determine whether the requested permissions are granted for the SID pair (*ssid*, *tsid*), interpreting the permissions based on tclass, and call the security server on a cache miss to obtain a new decision and add it to the cache. Update *aeref* to refer to an AVC entry with the resulting decisions. Audit the granting or denial of permissions in accordance with the policy. Return 0 if all requested permissions are granted, -1 with errno set to EACCES if any permissions are denied or to another value upon other errors.

avc_has_perm_noaudit - avc.h

Check permissions but perform no auditing. Check the AVC to determine whether the requested permissions are granted for the SID pair (*ssid*, *tsid*), interpreting the permissions based on tclass, and call the security server on a cache miss to obtain a new decision and add it to the cache. Update aeref to refer to an AVC entry with the resulting decisions, and return a copy of the decisions in *avd*. Return 0 if all requested permissions are granted, -1 with errno set to EACCES if any permissions are denied, or to another value upon other errors. This function is typically called by ***avc_has_perm****(3)*, but may also be called directly to separate permission checking from auditing, e.g. in cases where a lock must be held for the check but should be released for the auditing.

avc_init (deprecated)

Use *avc_open*. Initialize the AVC. Initialize the access vector cache. Return 0 on success or -1 with errno set on failure. If *msgprefix* is NULL, use *uavc*. If any callback structure references are NULL, use default methods for those callbacks (see the definition of the callback structures).

avc_netlink_acquire_fd - avc.h

Create a netlink socket and connect to the kernel.

avc_netlink_check_nb - avc.h

Wait for netlink messages from the kernel.

avc_netlink_close - avc.h

Close the netlink socket.

avc_netlink_loop - avc.h

Acquire netlink socket fd. Allows the application to manage messages from the netlink socket in its own main loop.

avc_netlink_open - avc.h

Release netlink socket fd. Returns ownership of the netlink socket to the library.

avc_netlink_release_fd - avc.h

Check netlink socket for new messages. Called by the application when using ***avc_netlink_acquire_fd****(3)* to process kernel netlink events.

avc_open - avc.h

Initialize the AVC. This function is identical to ***avc_init****(3)* except the message prefix is set to *avc* and any callbacks desired should be specified via ***selinux_set_callback****(3)*.

avc_reset - avc.h

Flush the cache and reset statistics. Remove all entries from the cache and reset all access statistics (as returned by ***avc_cache_stats****(3)*) to zero. The SID mapping is not affected. Return 0 on success, -1 with errno set on error.

avc_sid_stats - avc.h

Log SID table statistics. Log a message with information about the size and distribution of the SID table. The audit callback is used to print the message.

avc_sid_to_context, *avc_sid_to_context_raw - avc.h*

Get copy of context corresponding to SID. Return a copy of the security context corresponding to the input sid in the memory referenced by *ctx*. The caller is expected to free the context with ***freecon****(3)*. Return 0 on success, -1 on failure, with errno set to ENOMEM if insufficient memory was available to make the copy, or EINVAL if the input SID is invalid.

checkPasswdAccess (deprecated) - *selinux.h*

Use ***selinux_check_passwd_access****(3)* or preferably ***selinux_check_access****(3)*. Check a permission in the passwd class. Return 0 if granted or -1 otherwise.

context_free - context.h

Free the storage used by a context structure.

context_new - context.h

Return a new context initialized to a context string.

context_range_get - context.h

Get a pointer to the range.

context_range_set - context.h

Set the range component. Returns nonzero if unsuccessful.

context_role_get - context.h

Get a pointer to the role.

context_role_set - context.h

Set the role component. Returns nonzero if unsuccessful.

context_str - context.h

Return a pointer to the string value of *context_t*. Valid until the next call to ***context_str**(3)* or ***context_free**(3)* for the same *context_t*.

context_type_get - context.h

Get a pointer to the type.

context_type_set - context.h

Set the type component. Returns nonzero if unsuccessful.

context_user_get - context.h

Get a pointer to the user.

context_user_set - context.h

Set the user component. Returns nonzero if unsuccessful.

fgetfilecon, fgetfilecon_raw - selinux.h

Wrapper for the ***xattr**(7)* API - Get file context, and set **con* to refer to it. Caller must free via ***freecon**(3)*.

fini_selinuxmnt - selinux.h

Clear *selinuxmnt* variable and free allocated memory.

freecon - selinux.h

Free the memory allocated for a context by any of the *get** calls.

freeconary - selinux.h

Free the memory allocated for a context array by ***get_ordered_context_list**(3)*.

fsetfilecon, fsetfilecon_raw - selinux.h

Wrapper for the ***xattr**(7)* API - Set file context.

get_default_context - get_context_list.h

Get the default security context for a user session for *user* spawned by *fromcon* and set **newcon* to refer to it. The context will be one of those authorized by the policy, but the selection of a default is subject to user customizable preferences. If *fromcon* is NULL, defaults to current context. Returns 0 on success or -1 otherwise. Caller must free via ***freecon**(3)*.

get_default_context_with_level - get_context_list.h

Same as ***get_default_context**(3)*, but use the provided MLS level rather than the default level for the user.

get_default_context_with_role - get_context_list.h

Same as ***get_default_context**(3)*, but only return a context that has the specified role.

get_default_context_with_rolelevel - get_context_list.h

Same as ***get_default_context**(3)*, but only return a context that has the specified role and level.

get_default_type - get_default_type.h

Get the default type (domain) for *role* and set *type* to refer to it. Caller must free via ***free**(3)*. Return 0 on success or -1 otherwise.

get_ordered_context_list - get_context_list.h

Get an ordered list of authorized security contexts for a user session for *user* spawned by *fromcon* and set **conary* to refer to the NULL-terminated array of contexts. Every entry in the list will be authorized by the policy, but the ordering is subject to user customizable preferences. Returns number of entries in **conary*. If *fromcon* is NULL, defaults to current context. Caller must free via ***freeconary****(3)*.

get_ordered_context_list_with_level - get_context_list.h

Same as ***get_ordered_context_list****(3)*, but use the provided MLS level rather than the default level for the user.

getcon, getcon_raw - selinux.h

Get current context, and set **con* to refer to it. Caller must free via ***freecon****(3)*.

getexeccon, getexeccon_raw - selinux.h

Get exec context, and set **con* to refer to it. Sets **con* to NULL if no exec context has been set, i.e. using default. If non-NULL, caller must free via ***freecon****(3)*.

getfilecon, getfilecon_raw - selinux.h

Wrapper for the ***xattr****(7)* API - Get file context, and set **con* to refer to it. Caller must free via ***freecon****(3)*.

getfscreatecon, getfscreatecon_raw - selinux.h

Get fscreate context, and set **con* to refer to it. Sets **con* to NULL if no fs create context has been set, i.e. using default.If non-NULL, caller must free via ***freecon****(3)*.

getkeycreatecon getkeycreatecon_raw - selinux.h

Get keycreate context, and set **con* to refer to it. Sets **con* to NULL if no key create context has been set, i.e. using default. If non-NULL, caller must free via ***freecon****(3)*.

getpeercon, getpeercon_raw - selinux.h

Wrapper for the socket API - Get context of peer socket, and set **con* to refer to it. Caller must free via ***freecon****(3)*.

getpidcon, getpidcon_raw - selinux.h

Get context of process identified by pid, and set **con* to refer to it. Caller must free via ***freecon****(3)*.

getprevcon, getprevcon_raw - selinux.h

Get previous context (prior to last exec), and set **con* to refer to it. Caller must free via ***freecon****(3)*.

getseuser - selinux.h

Get the SELinux username and level to use for a given Linux username and service. These values may then be passed into the ***get_ordered_context_list****(3)* and ***get_default_context****(3)* functions to obtain a context for the user. Returns 0 on success or -1 otherwise. Caller must free the returned strings via ***free****(3)*.

getseuserbyname - selinux.h

Get the SELinux username and level to use for a given Linux username. These values may then be passed into the ***get_ordered_context_list****(3)* and ***get_default_context****(3)* functions to obtain a context for the user. Returns 0 on success or -1 otherwise. Caller must free the returned strings via ***free****(3)*.

getsockcreatecon, getsockcreatecon_raw - selinux.h

Get sockcreate context, and set **con* to refer to it. Sets **con* to NULL if no socket create context has been set, i.e. using default. If non-NULL, caller must free via ***freecon****(3)*.

init_selinuxmnt - selinux.h

There is a man page for this, however it is not a user accessible function (internal use only - although the ***fini_selinuxmnt****(3)* is reachable).

is_context_customizable - selinux.h

Returns whether a file context is customizable, and should not be relabeled.

is_selinux_enabled - selinux.h

Return 1 if running on a SELinux kernel, or 0 if not or -1 for error.

is_selinux_mls_enabled - selinux.h

Return 1 if we are running on a SELinux MLS kernel, or 0 otherwise.

lgetfilecon, lgetfilecon_raw - selinux.h

Wrapper for the ***xattr****(7)* API - Get file context, and set **con* to refer to it. Caller must free via ***freecon****(3)*.

lsetfilecon, lsetfilecon_raw - selinux.h

Wrapper for the xattr API- Set file context for symbolic link.

manual_user_enter_context - get_context_list.h

Allow the user to manually enter a context as a fallback if a list of authorized contexts could not be obtained. Caller must free via ***freecon****(3)*. Returns 0 on success or -1 otherwise.

matchmediacon - selinux.h

Match the specified media and against the media contexts configuration and set **con* to refer to the resulting context. Caller must free con via ***freecon****(3)*.

matchpathcon (deprecated) - *selinux.h*

Match the specified *pathname* and *mode* against the file context configuration and set **con* to refer to the resulting context. *mode* can be 0 to disable mode matching. Caller must free via freecon. If ***matchpathcon_init****(3)* has not already been called, then this function will call it upon its first invocation with a NULL path.

matchpathcon_checkmatches (deprecated) - *selinux.h*

Check to see whether any specifications had no matches and report them. The *str* is used as a prefix for any warning messages.

matchpathcon_filespec_add (deprecated) - *selinux.h*

Maintain an association between an inode and a specification index, and check whether a conflicting specification is already associated with the same inode (e.g. due to multiple hard links). If so, then use the latter of the two specifications based on their order in the file contexts configuration. Return the used specification index.

matchpathcon_filespec_destroy (deprecated) - *selinux.h*

Destroy any inode associations that have been added, e.g. to restart for a new filesystem.

matchpathcon_filespec_eval (deprecated) - *selinux.h*

Display statistics on the hash table usage for the associations.

matchpathcon_fini (deprecated) - *selinux.h*

Free the memory allocated by ***matchpathcon_init****(3)*.

matchpathcon_index (deprecated) - *selinux.h*

Same as ***matchpathcon****(3)*, but return a specification index for later use in a ***matchpathcon_filespec_add****(3)* call.

matchpathcon_init (deprecated) - *selinux.h*

Load the file contexts configuration specified by *path* into memory for use by subsequent *matchpathcon* calls. If *path* is NULL, then load the active file contexts configuration, i.e. the path returned by ***selinux_file_context_path****(3)*. Unless the *MATCHPATHCON_BASEONLY* flag has been set, this function also checks for a *path.homedirs* file and a *path.local* file and loads additional specifications from them if present.

matchpathcon_init_prefix (deprecated) - *selinux.h*

Same as ***matchpathcon_init****(3)*, but only load entries with regexes that have stems that are prefixes of *prefix*.

mode_to_security_class - selinux.h

Translate mode_t to a security class string name (e.g. *S_ISREG = file*).

print_access_vector - selinux.h

Display an access vector in a string representation.

query_user_context- get_context_list.h

Given a list of authorized security contexts for the user, query the user to select one and set **newcon* to refer to it. Caller must free via ***freecon****(3)*. Returns 0 on success or -1 otherwise.

realpath_not_final - selinux.h

Resolve all of the symlinks and relative portions of a *pathname*, but NOT the final component (same a ***realpath****(3)* unless the final component is a symlink. Resolved path must be a path of size *PATH_MAX + 1*.

rpm_execcon (deprecated) - *selinux.h*

Use ***setexecfilecon****(3)* and ***execve****(2)*. Execute a helper for rpm in an appropriate security context.

security_av_perm_to_string - selinux.h

Convert access vector permissions to string names.

security_av_string - selinux.h

Returns an access vector in a string representation. User must free the returned string via ***free****(3)*.

security_canonicalize_context, *security_canonicalize_context_raw - selinux.h*

Canonicalize a security context. Returns a pointer to the canonical (primary) form of a security context in *canoncon* that the kernel is using rather than what is provided by the userspace application in *con*.

security_check_context, *security_check_context_raw - selinux.h*

Check the validity of a security context.

security_class_to_string - selinux.h

Convert security class values to string names.

security_commit_booleans - selinux.h

Commit the pending values for the booleans.

security_compute_av, *security_compute_av_raw - selinux.h*

Compute an access decision. Queries whether the policy permits the source context *scon* to access the target context *tcon* via class *tclass* with the *requested* access vector. The decision is returned in *avd*.

security_compute_av_flags, *security_compute_av_flags_raw - selinux.h*

Compute an access decision and return the flags. Queries whether the policy permits the source context *scon* to access the target context *tcon* via class *tclass* with the *requested* access vector. The decision is returned in *avd*. that has an additional *flags* entry. Currently the only *flag* defined is *SELINUX_AVD_FLAGS_PERMISSIVE* that indicates the decision was computed on a permissive domain (i.e. the *permissive* policy language statement has been used in policy or ***semanage****(8)* has been used to set the domain in permissive mode). Note this does not indicate that SELinux is running in permissive mode, only the *scon* domain.

security_compute_create, *security_compute_create_raw - selinux.h*

Compute a labeling decision and set **newcon* to refer to it. Caller must free via ***freecon****(3)*.

security_compute_create_name, *security_compute_create_name_raw - selinux.h*

This is identical to ***security_compute_create****(3)* but also takes the name of the new object in creation as an argument. When a *type_transition* rule on the given class and the *scon* / *tcon* pair has an object name extension, **newcon* will be returned according to the policy. Note that this interface is only supported on the kernels 2.6.40 or later. For older kernels the object name is ignored.

security_compute_member, *security_compute_member_raw - selinux.h*

Compute a polyinstantiation member decision and set **newcon* to refer to it. Caller must free via ***freecon****(3)*.

security_compute_relabel, *security_compute_relabel_raw - selinux.h*

Compute a relabeling decision and set **newcon* to refer to it. Caller must free via ***freecon****(3)*.

security_compute_user, *security_compute_user_raw* (deprecated) - *selinux.h*

Compute the set of reachable user contexts and set **con* to refer to the NULL-terminated array of contexts. Caller must free via ***freeconary****(3)*.

security_deny_unknown - selinux.h

Get the behavior for undefined classes / permissions.

security_disable - selinux.h

Disable SELinux at runtime (must be done prior to initial policy load).

security_get_boolean_active - selinux.h

Get the active value for the boolean.

security_get_boolean_names - selinux.h

Get the boolean names

security_get_boolean_pending - selinux.h

Get the pending value for the boolean.

security_get_initial_context, *security_get_initial_context_raw - selinux.h*

Get the context of an initial kernel security identifier by name. Caller must free via ***freecon****(3)*.

security_getenforce - selinux.h

Get the enforce flag value.

security_load_booleans (deprecated) - *selinux.h*

Load policy boolean settings. Path may be NULL, in which case the booleans are loaded from the active policy boolean configuration file.

security_load_policy - selinux.h

Load a policy configuration.

security_policyvers - selinux.h

Get the policy version number.

security_set_boolean - selinux.h

Set the pending value for the boolean.

security_set_boolean_list - selinux.h

Save a list of booleans in a single transaction.

security_setenforce - selinux.h

Set the enforce flag value.

security_validatetrans, *security_validatetrans_raw - selinux.h*

Validate a transition. This determines whether a transition from *scon* to *newcon* using *tcon* as the target for object class *tclass* is valid in the loaded policy. This checks against the *mlsvalidatetrans* and *validatetrans* constraints in the loaded policy. Returns 0 if allowed and -1 if an error occurred with errno set.

selabel_close - label.h

Destroy the specified handle, closing files, freeing allocated memory, etc. The handle may not be further used after it has been closed.

selabel_cmp - label.h

Compare two label configurations. Returns:

- *SELABEL_SUBSET* - if *h1* is a subset of *h2*
- *SELABEL_EQUAL* - if *h1* is identical to *h2*
- *SELABEL_SUPERSET* - if *h1* is a superset of *h2*
- *SELABEL_INCOMPARABLE* - if *h1* and *h2* are incomparable

selabel_digest - label.h

Retrieve the SHA1 digest and the list of specfiles used to generate the digest. The *SELABEL_OPT_DIGEST* option must be set in ***selabel_open****(3)* to initiate the digest generation.

selabel_get_digests_all_partial_matches - label.h

Returns true if the digest of all partial matched contexts is the same as the one saved by ***setxattr****(2)*, otherwise returns false. The length of the SHA1 digest will always be returned. The caller must free any returned digests.

selabel_hash_all_partial_matches - label.h

Returns true if the digest of all partial matched contexts is the same as the one saved by ***setxattr****(2)*, otherwise returns false.

selabel_lookup, *selabel_lookup_raw* - *label.h*

Perform a labeling lookup operation. Return 0 on success, -1 with *errno* set on failure. The *key* and *type* arguments are the inputs to the lookup operation; appropriate values are dictated by the backend in use. The result is returned in the memory pointed to by *con* and must be freed by ***freecon****(3)*.

selabel_lookup_best_match, *selabel_lookup_best_match_raw* - *label.h*

Obtain a best match SELinux security context - Only supported on file backend. The order of precedence for best match is:

- An exact match for the real path (key) or
- An exact match for any of the links (aliases), or
- The longest fixed prefix match.

selabel_open - *label.h*

Create a labeling handle. Open a labeling backend for use. The available backend identifiers are:

- *SELABEL_CTX_FILE* - *file_contexts*.
- *SELABEL_CTX_MEDIA* - media contexts.
- *SELABEL_CTX_X* - *x_contexts*.
- *SELABEL_CTX_DB* - SE-PostgreSQL contexts.
- *SELABEL_CTX_ANDROID_PROP* – *property_contexts*.
- *SELABEL_CTX_ANDROID_SERVICE* – *service_contexts*.

Options may be provided via the *opts* parameter; available options are:

- *SELABEL_OPT_UNUSED* - no-op option, useful for unused slots in an array of options.
- *SELABEL_OPT_VALIDATE* - validate contexts before returning them (boolean value).
- *SELABEL_OPT_BASEONLY* - don't use local customizations to backend data (boolean value).
- *SELABEL_OPT_PATH* - specify an alternate path to use when loading backend data.
- *SELABEL_OPT_SUBSET* - select a subset of the search space as an optimization (file backend).
- *SELABEL_OPT_DIGEST* – request an SHA1 digest of the specfiles.

Not all options may be supported by every backend. Return value is the created handle on success or NULL with errno set on failure.

selabel_partial_match - *label.h*

Performs a partial match operation, returning TRUE or FALSE. The *key* parameter is a file *path* to check for a direct or partial match. Returns TRUE if a direct or partial match is found, FALSE if not.

selabel_stats - *label.h*

Log a message with information about the number of queries performed, number of unused matching entries, or other operational statistics. Message is backend-specific, some backends may not output a message.

selinux_binary_policy_path - *selinux.h*

Return path to the binary policy file under the policy root directory.

selinux_booleans_path (deprecated) - *selinux.h*

Return path to the booleans file under the policy root directory.

selinux_boolean_sub - *selinux.h*

Reads the */etc/selinux/TYPE/booleans.subs_dist* file looking for a record with *boolean_name*. If a record exists ***selinux_boolean_sub****(3)* returns the translated name otherwise it returns the original name. The returned value needs to be freed. On failure NULL will be returned.

selinux_booleans_subs_path - selinux.h

Returns the path to the *booleans.subs_dist* configuration file.

selinux_check_access - selinux.h

Used to check if the source context has the access permission for the specified class on the target context. Note that the permission and class are reference strings. The *aux* parameter may reference supplemental auditing information. Auditing is handled as described in ***avc_audit****(3)*. See ***security_deny_unknown****(3)* for how the *deny_unknown* flag can influence policy decisions.

selinux_check_passwd_access (deprecated) - *selinux.h*

Use ***selinux_check_access****(3)*. Check a permission in the passwd class. Return 0 if granted or -1 otherwise.

selinux_check_securetty_context - selinux.h

Check if the tty_context is defined as a securetty. Return 0 if secure, < 0 otherwise.

selinux_colors_path - selinux.h

Return path to file under the policy root directory.

selinux_contexts_path - selinux.h

Return path to contexts directory under the policy root directory.

selinux_current_policy_path - selinux.h

Return path to the current policy.

selinux_customizable_types_path - selinux.h

Return path to customizable_types file under the policy root directory.

selinux_default_context_path - selinux.h

Return path to default_context file under the policy root directory.

selinux_default_type_path - get_default_type.h

Return path to *default_type* file.

selinux_failsafe_context_path - selinux.h

Return path to failsafe_context file under the policy root directory.

selinux_file_context_cmp - selinux.h

Compare two file contexts, return 0 if equivalent.

selinux_file_context_homedir_path - selinux.h

Return path to *file_context.homedir* file under the policy root directory.

selinux_file_context_local_path - selinux.h

Return path to *file_context.local* file under the policy root directory.

selinux_file_context_path - selinux.h

Return path to *file_context* file under the policy root directory.

selinux_file_context_subs_path - selinux.h

Return path to *file_context.subs* file under the policy root directory.

selinux_file_context_subs_dist_path - selinux.h

Return path to *file_context.subs_dist* file under the policy root directory.

selinux_file_context_verify - selinux.h

Verify the context of the file *path* against policy. Return 0 if correct.

selinux_get_callback - selinux.h

Used to get a pointer to the callback function of the given *type*. Callback functions are set using ***selinux_set_callback****(3)*.

selinux_getenforcemode - selinux.h

Reads the /etc/selinux/config file and determines whether the machine should be started in enforcing (1), permissive (0) or disabled (-1) mode.

selinux_getpolicytype - selinux.h

Reads the */etc/selinux/config* file and determines what the default policy for the machine is. Calling application must free *policytype*.

selinux_homedir_context_path - selinux.h

Return path to file under the policy root directory. Note that this file will only appear in older versions of policy at this location. On systems that are managed using ***semanage****(8)* this is now in the policy store.

selinux_init_load_policy - selinux.h

Perform the initial policy load. This function determines the desired enforcing mode, sets the *enforce* argument accordingly for the caller to use, sets the SELinux kernel enforcing status to match it, and loads the policy. It also internally handles the initial *selinuxfs* mount required to perform these actions. The function returns 0 if everything including the policy load succeeds. In this case, init is expected to re-exec itself in order to transition to the proper security context. Otherwise, the function returns -1, and init must check *enforce* to determine how to proceed. If enforcing (*enforce* > 0), then init should halt the system. Otherwise, init may proceed normally without a re-exec.

selinux_lsetfilecon_default - selinux.h

This function sets the file context to the system defaults. Returns 0 on success.

selinux_lxc_contexts_path - selinux.h

Return the path to the *lxc_contexts* configuration file.

selinux_media_context_path - selinux.h

Return path to file under the policy root directory.

selinux_mkload_policy - selinux.h

Make a policy image and load it. This function provides a higher level interface for loading policy than ***security_load_policy****(3)*, internally determining the right policy version, locating and opening the policy file,

mapping it into memory, manipulating it as needed for current boolean settings and/or local definitions, and then calling ***security_load_policy****(3)* to load it. *preservebools* is a boolean flag indicating whether current policy boolean values should be preserved into the new policy (if 1) or reset to the saved policy settings (if 0). The former case is the default for policy reloads, while the latter case is an option for policy reloads but is primarily for the initial policy load.

selinux_netfilter_context_path - selinux.h

Returns path to the netfilter_context file under the policy root directory.

selinux_path - selinux.h

Returns path to the policy root directory.

selinux_policy_root - selinux.h

Reads the /etc/selinux/config file and returns the top level directory.

selinux_raw_context_to_color - selinux.h

Perform context translation between security contexts and display colors. Returns a space-separated list of ten ten hex RGB triples prefixed by hash marks, e.g. *#ff0000*. Caller must free the resulting string via ***free****(3)*. Returns -1 upon an error or 0 otherwise.

selinux_raw_to_trans_context - selinux.h

Perform context translation between the human-readable format (*translated*) and the internal system format (*raw*). Caller must free the resulting context via ***freecon****(3)*. Returns -1 upon an error or 0 otherwise. If passed NULL, sets the returned context to NULL and returns 0.

selinux_removable_context_path - selinux.h

Return path to *removable_context* file under the policy root directory.

selinux_restorecon - restorecon.h

Relabel files that automatically calls ***selinux_restorecon_default_handle****(3)* and ***selinux_restorecon_set_sehandle****(3)* first time through to set the ***selabel_open****(3)* parameters to use the currently loaded policy *file_contexts* and request their computed digest. Should other ***selabel_open****(3)* parameters be required see ***selinux_restorecon_set_sehandle****(3)*.

selinux_restorecon_xattr - restorecon.h

Read/remove *RESTORECON_LAST* xattr entries that automatically calls ***selinux_restorecon_default_handle****(3)* and ***selinux_restorecon_set_sehandle****(3)* first time through to set the ***selabel_open****(3)* parameters to use the currently loaded policy *file_contexts* and request their computed digest. Should other ***selabel_open****(3)* parameters be required see ***selinux_restorecon_set_sehandle****(3)*, however note that a *file_contexts* computed digest is required for ***selinux_restorecon_xattr****(3)*.

selinux_restorecon_default_handle - restorecon.h

Sets default ***selabel_open****(3)* parameters to use the currently loaded policy and *file_contexts*, also requests the digest.

selinux_restorecon_set_alt_rootpath - restorecon.h

Use alternate rootpath.

selinux_restorecon_set_exclude_list - restorecon.h

Add a list of directories that are to be excluded from relabeling.

selinux_restorecon_set_sehandle - restorecon.h

Set the global fc handle. Called by a process that has already called ***selabel_open****(3)* with its required parameters, or if ***selinux_restorecon_default_handle****(3)* has been called to set the default ***selabel_open****(3)* parameters.

selinux_securetty_types_path - selinux.h

Return path to the securetty_types file under the policy root directory.

selinux_sepgsql_context_path - selinux.h

Return path to *sepgsql_context* file under the policy root directory.

selinux_set_callback - selinux.h

Sets the callback according to the type: *SELINUX_CB_LOG*, *SELINUX_CB_AUDIT*, *SELINUX_CB_VALIDATE*, *SELINUX_CB_SETENFORCE*, *SELINUX_CB_POLICYLOAD*

selinux_set_mapping - selinux.h

Userspace class mapping support that establishes a mapping from a user-provided ordering of object classes and permissions to the numbers actually used by the loaded system policy.

selinux_set_policy_root - selinux.h

Sets an alternate policy root directory path under which the compiled policy file and context configuration files exist.

selinux_status_open - avc.h

Open and map SELinux kernel status page.

selinux_status_close - avc.h

Unmap and close kernel status page.

selinux_status_updated - avc.h

Inform whether the kernel status has been updated.

selinux_status_getenforce - avc.h

Get the enforce flag value.

selinux_status_policyload - avc.h

Get the number of policy loads.

selinux_status_deny_unknown - avc.h

Get behaviour for undefined classes/permissions.

selinux_systemd_contexts_path - selinux.h

Returns the path to the *systemd_contexts* configuration file.

selinux_reset_config - selinux.h

Force a reset of the loaded configuration. **WARNING**: This is not thread safe. Be very sure that no other threads are calling into *libselinux* when this is called.

selinux_trans_to_raw_context - selinux.h

Perform context translation between the human-readable format (*translated*) and the internal system format (*raw*). Caller must free the resulting context via ***freecon**(3)*. Returns -1 upon an error or 0 otherwise. If passed NULL, sets the returned context to NULL and returns 0.

selinux_translations_path - selinux.h

Return path to setrans.conf file under the policy root directory.

selinux_user_contexts_path - selinux.h

Return path to file under the policy root directory.

selinux_users_path (deprecated) - *selinux.h*

Return path to file under the policy root directory.

selinux_usersconf_path - selinux.h

Return path to file under the policy root directory.

selinux_virtual_domain_context_path - selinux.h

Return path to file under the policy root directory.

selinux_virtual_image_context_path - selinux.h

Return path to file under the policy root directory.

selinux_x_context_path - selinux.h

Return path to x_context file under the policy root directory.

selinuxfs_exists - selinux.h

Check if selinuxfs exists as a kernel filesystem.

set_matchpathcon_canoncon (deprecated) - *selinux.h*

Same as ***set_matchpathcon_invalidcon**(3)*, but also allows canonicalization of the context, by changing *context* to refer to the canonical form. If not set, and *invalidcon* is also not set, then this defaults to calling ***security_canonicalize_context**(3)*.

set_matchpathcon_flags (deprecated) - *selinux.h*

Set flags controlling operation of ***matchpathcon_init**(3)* or ***matchpathcon**(3)*:

- *MATCHPATHCON_BASEONLY* - Only process the base file_contexts file.
- *MATCHPATHCON_NOTRANS* - Do not perform any context translation.
- *MATCHPATHCON_VALIDATE* - Validate/canonicalize contexts at init time.

set_matchpathcon_invalidcon (deprecated) - *selinux.h*

Set the function used by ***matchpathcon_init**(3)* when checking the validity of a context in the *file_contexts* configuration. If not set, then this defaults to a test based on ***security_check_context**(3)*. The function is also responsible for reporting any such error, and may include the *path* and *lineno* in such error messages.

set_matchpathcon_printf (deprecated) - *selinux.h*

Set the function used by ***matchpathcon_init**(3)* when displaying errors about the *file_contexts* configuration. If not set, then this defaults to *fprintf(stderr, fmt, ...)*.

set_selinuxmnt - selinux.h

Set the path to the selinuxfs mount point explicitly. Normally, this is determined automatically during libselinux initialization, but this is not always possible, e.g. for /sbin/init which performs the initial mount of *selinuxfs*.

setcon, *setcon_raw* - *selinux.h*

Set the current security context to con. Note that use of this function requires that the entire application be trusted to maintain any desired separation between the old and new security contexts, unlike exec-based transitions performed via ***setexeccon****(3)*. When possible, decompose your application and use ***setexeccon****(3)* + ***execve****(3)* instead. Note that the application may lose access to its open descriptors as a result of a ***setcon****(3)* unless policy allows it to use descriptors opened by the old context.

setexeccon, *setexeccon_raw* - *selinux.h*

Set exec security context for the next ***execve****(3)*. Call with NULL if you want to reset to the default.

setexecfilecon - *selinux.h*

Set an appropriate security context based on the filename of a helper program, falling back to a new context with the specified type.

setfilecon, *setfilecon_raw* - *selinux.h*

Wrapper for the xattr API - Set file context.

setfscreatecon, *setfscreatecon_raw* - *selinux.h*

Set the fscreate security context for subsequent file creations. Call with NULL if you want to reset to the default.

setkeycreatecon, *setkeycreatecon_raw* - *selinux.h*

Set the keycreate security context for subsequent key creations. Call with NULL if you want to reset to the default.

setsockcreatecon, *setsockcreatecon_raw* - *selinux.h*

Set the sockcreate security context for subsequent socket creations. Call with NULL if you want to reset to the default.

sidget (deprecated) - *avc.h*

From 2.0.86 this is a no-op.

sidput (deprecated) - *avc.h*

From 2.0.86 this is a no-op.

string_to_av_perm - *selinux.h*

Convert string names to access vector permissions.

string_to_security_class - *selinux.h*

Convert string names to security class values.

Appendix C - SELinux Commands

This section gives a brief summary of the SELinux specific commands. Some of these have been used within this Notebook, however the appropriate man pages do give more detail and the SELinux project site has a page that details all the available tools and commands at:

https://github.com/SELinuxProject/selinux/wiki/Tools

***audit2allow**(1)*

Generates policy allow rules from an audit log file.

***audit2why**(8)*

Describes audit log messages and why access was denied.

***avcstat**(8)*

Displays the AVC statistics.

***chcat**(8)*

Change or remove a catergory from a file or user.

***chcon**(1)*

Changes the security context of a file.

***checkmodule**(8)*

Compiles base and loadable modules from source.

***checkpolicy**(8)*

Compiles a monolithic policy from source.

***fixfiles**(8)*

Update / correct the security context of for filesystems that use extended attributes.

***genhomedircon**(8)*

Generates file configuration entries for users home directories. This command has also been built into ***semanage**(8)*, therefore when using the policy store / loadable modules this does not need to be used.

***getenforce**(1)*

Shows the current enforcement state.

***getsebool**(8)*

Shows the state of the booleans.

***load_policy**(8)*

Loads a new policy into the kernel. Not required when using ***semanage**(8)* / ***semodule**(8)* commands.

***matchpathcon**(8)*

Show a files path and security context.

***newrole**(1)*

Allows users to change roles - runs a new shell with the new security context.

***restorecon**(8)*

Sets the security context on one or more files.

***run_init**(8)*

Runs an *init* script under the correct context.

***runcon**(1)*

Runs a command with the specified context.

***selinuxenabled**(1)*

Shows whether SELinux is enabled or not.

***semanage**(8)*

Used to configure various areas of a policy within a policy store.

***semodule**(8)*

Used to manage the installation, upgrading etc. of policy modules.

***semodule_expand**(8)*

Manually expand a base policy package into a kernel binary policy file.

***semodule_link**(8)*

Manually link a set of module packages.

***semodule_package**(8)*

Create a module package with various configuration files (file context etc.)

***sestatus**(8)*

Show the current status of SELinux and the loaded policy.

***setenforce**(1)*

Sets / unsets enforcement mode.

***setfiles**(8)*

Initialise the extended attributes of filesystems.

***setsebool**(8)*

Sets the state of a boolean to on or off persistently across reboots or for this session only.

Appendix D - Debugging Policy - Hints and Tips

I’m sure there is more to add here !!!

Appendix E - Policy Validation Example

This example has been taken from **http://selinuxproject.org/page/PolicyValidate** just in case the site is removed some day.

libsemanage(8) is the library responsible for building a kernel policy from policy modules. It has many features but one that is rarely mentioned is the policy validation hook. This example will show how to make a basic validator and tell ***libsemanage*** to run it before allowing any policy updates.

The sample validator uses ***sesearch****(1)* to search for a rule between *user_t* and *shadow_t*. The purpose of this validator is to never allow a policy update that allows *user_t* to access *shadow_t*.

To use the script below requires the **setools-console** package to be installed.

Make a file in */usr/local/bin/validate* that contains the following (run *chmod +x* or ***semodule****(8)* will fail):

```
#!/bin/bash

# Usage: validate <policy file>

# The following searches for a file rule with user_t as the source and
# shadow_t as the target.
# If the output of sesearch has "Found", meaning matching rules were found,
# then grep will return 0 otherwise it will return 1. This is actually the
# reverse of the logic we want, so we'll reverse it.

sesearch --allow -s user_t -t shadow_t -c file $1 | grep "Found" > /dev/null

if [ $? == 1 ]; then
        exit 0
fi

exit 1
```

Then add the validation script to */etc/selinux/semanage.conf*:

```
[verify kernel]
path = /usr/local/bin/validate
args = $@
[end]
```

Next try rebuilding the policy with no changes:

```
semodule -B
```

It should succeed, therefore build a module that would violate this rule:

```
module badmod 1.0;

require {
      type user_t, shadow_t;
```

```
      class file { read };
}

allow user_t shadow_t : file read;
```

Do the standard compilation steps:

```
checkmodule -o badmod.mod badmod.te -m -M

checkmodule:  loading policy configuration from badmod.te
checkmodule:  policy configuration loaded
checkmodule:  writing binary representation (version 17) to badmod.mod

semodule_package -m badmod.mod -o badmod.pp
```

And then attempt to insert it:

```
semodule -i badmod.pp
semodule: Failed!
```

Now run ***sesearch*** to ensure that there is no matching rule:

```
sesearch --allow -s user_t -t shadow_t -c file
```

Note that there are also a **[verify module]** and **[verify linked]** options as described in the **Global Configuration Files** - *semanage.conf* file section.

1. When SELinux is enabled, the policy can be running in 'permissive mode' (*SELINUX=permissive*), where all accesses are allowed. The policy can also be run in 'enforcing mode' (*SELINUX=enforcing*), where any access that is not defined in the policy is denied and an entry placed in the audit log. SELinux can also be disabled (at boot time only) by setting *SELINUX=disabled*. There is also support for the ***permissive*** statement that allows a domain to run in permissive mode while the others are still confined (instead of all or nothing set by *SELINUX=*). Note setting *SELINUX=disabled* will be deprecated at some stage, in favor of the existing kernel command line switch *selinux=0*, which allows users to disable SELinux at system boot. See https://github.com/SELinuxProject/selinux-kernel/wiki/DEPRECATE-runtime-disable that explains how to achieve this on various Linux distributions.↩

2. The user *system_u* name is not mandatory, it is used to signify a special user in the Reference Policy. It is also used in some SELinux utilities.↩

3. The object class and its associated permissions are explained in **Appendix A - Object Classes and Permissions - Process Object Class**↩

4. These file systems store the security context in an attribute associated with the file.↩

5. Note that this file contains the contexts of all files in all extended attribute filesystems for the policy. However within a modular policy (and/or CIL modules) each module describes its own file context information, that is then used to build this file.↩

6. See **Figure 2: High Level SELinux Architecture** in the 'Core SELinux Components' section.↩

7. For example, an ftp session where the server is listening on a specific port (the destination port) but the client will be assigned a random source port. The *CONNSECMARK* will ensure that all packets for the ftp session are marked with the same label.↩

8. Note only the security levels are passed over the network as the other components of the security context are not part of standard MLS systems (as it may be that the remote end is a Trusted Solaris system)↩

9. These are the Internet Key Exchange (IKE) daemons that exchange encryption keys securely and also supports Labeled IPSec parameter exchanges.↩

10. KVM (Kernel-based Virtual Machine) and Xen are classed as 'bare metal' hypervisors and they rely on other services to manage the overall VM environment. QEMU (Quick Emulator) is an emulator that emulates the BIOS and I/O device functionality and can be used standalone or with KVM and Xen.↩

11. This is similar to the LAMP (Linux, Apache, MySQL, PHP/Perl/Python) stack, however MySQL is not SELinux-aware.↩

12. As each module would have its own *file_contexts* component that is either added or removed from the policies overall */etc/selinux/<SELINUXTYPE>/contexts/files/file_contexts* file.↩

13. It is important to note that the Reference Policy builds policy using makefiles and m4 support macros within its own source file structure. However, the end result of the make process is that there can be three possible types of source file built (depending on the *MONOLITHIC=Y/N* build option). These files contain the policy language statements and rules that are finally complied into a binary policy.↩

14. This does not include the *file_contexts* file as it does not contain policy statements, only default security contexts (labels) that will be used to label files and directories.↩

15. *neverallow* statements are allowed in modules, however to detect these the *semanage.conf* file must have the *expand-check=1* entry present.↩

16. Only if preceded by the *optional* statement.↩

17. Only if preceded by the *optional* statement.↩

18. 'Name transition rules' are not allowed inside *conditional* statements.↩

19. *neverallow* statements are allowed in modules, however to detect these the *semanage.conf* file must have the 'expand-check=1' entry present.↩

20. The *--disable-neverallow* option can be used with ***secilc**(8)* to disable *neverallow* rule checking.↩

21. The SELinux security server does not enforce a decision, it merely states whether the operation is allowed or not according to the policy. It is the object manager that enforces the decision of the policy / security server, therefore an object manager must be trusted. This is also true of labeling, the object manager ensures that labels are applied to their objects as defined by policy.↩

22. An example of this integration is setting a new process context as shown in the Zygote code: https://android.googlesource.com/platform/frameworks/base/+/refs/heads/android10-dev/core/jni/com_android_internal_os_Zygote.cpp#1095. The **SE for Android** section explains SELinux integration within Android AOSP services.↩

23. The 'policy store' holds policy modules in 'policy package' format (*.*pp* files).↩

24. The 'policy store' holds policy modules as compressed CIL text files.↩