SELinux for LE Targets

SELinuxLE.pdf

Introduction

The SELinux Policy is the set of rules that guide the SELinux security engine. It defines *types* for file objects and *domains* for processes. In general, all system call accesses go through SELinux which decides if the access should be allowed or not

- SELinux (Security-Enhanced Linux) is an implementation of mandatory access control (MAC) in the Linux kernel using the Linux Security Modules (LSM) framework.
- Restricts privileges of user programs and system servers using security labels and preinstalled policies
- SELinux enforces MAC policies that confine user programs and system servers to the minimum amount of privilege they require to do their jobs
- Security Policies are implemented using:

    1. Type Enforcement (TE)
    2. Role-based access control (RBAC)
    3. Multi-level Security (MLS)

SELinux LE FAQ Confluence Page

~~https://qwiki.qualcomm.com/quic/SELinux_LE_FAQ~~

LE-SElinux-FAQ

Operations Guide

Enable SELinux

Use a distro with 'selinux' selected in 'DISTRO_FEATURES'

> A clean build is required after changing this value on Thud and below. Incremental builds after making this change will have a compile error related to libselinux version changes

Query status of SELinux

Get the status of the system with getenforce on target. This can return one of three values: 'enforcing', 'permissive', or 'disabled'

Change SELinux Mode

- Select a mode at runtime by running setenforce with a number

Command	Result
setenforce 0	Move to permissive mode
setenforce 1	Move to enforcing mode

- Select a mode from bootup without recompiling

    - adb pull /etc/selinux/config
    - Edit SELINUX= to one of the three supported values: 'enforcing', 'permissive', or 'disabled'
    - adb remount -o remount,rw /
    - adb push config /etc/selinux/config
- Change DEFAULT_ENFORCING build flag to one of the three supported values: 'enforcing', 'permissive', or 'disabled'
    - Location depends on the machine you are building for
    - Recompiling qti-*-image and reflashing system image is required

SELinux 'disabled' mode still leaves behind many code paths that go through the SELinux framework. This is not useful for KPI testing or checking for bugs in SELinux framework. It also does not allow any more access than permissive mode. Therefore we recommend removing selinux from DISTRO_FEATURES if the feature needs to be completely disabled for testing.

Reading Denial Messages

SELinux denials appear in console logs and in dmesg output. They will look similar to the below message. The recommended approach is to grep for the 'avc' string

State

audit: type=1400 audit(1595843096.039:3): avc: **denied** { getattr } for pid=305 comm="dnsmasq" path="/run/systemd/resolve/resolv.conf" dev="tmpfs" ino=11079 scontext=system_u:system_r:dnsmasq_t:s0-s15:c0.c1023 tcontext=system_u:object_r:init_var_run_t:s0 tclass=file permissive=1

Denied:

- This rule blocks access in enforcing mode. In permissive mode (permissive=1), this is not actually blocked, this message is just a warning

Granted:

- Rarely seen. The access was allowed, but the policy writer wants to know about this access. Typically indicates the policy writer intends to deprecate the access

Permission

audit: type=1400 audit(1595843096.039:3): avc: denied **{ getattr }** for pid=305 comm="dnsmasq" path="/run/systemd/resolve/resolv.conf" dev="tmpfs" ino=11079 scontext=system_u:system_r:dnsmasq_t:s0-s15:c0.c1023 tcontext=system_u:object_r:init_var_run_t:s0 tclass=file permissive=1

Which permission was requested.

Common values are: create open getattr setattr read write search add_name remove_name rmdir lock ioctl

Many times, policy writers know they need to open files, but forget about searching and calling getattr on parent directories to find the file to open. Macros are typically used to handle edge cases.

Source Context

audit: type=1400 audit(1595843096.039:3): avc: denied { getattr } for **pid=305 comm="dnsmasq"** path="/run/systemd/resolve/resolv.conf" dev="tmpfs" ino=11079 **scontext=system_u:system_r:dnsmasq_t:s0-s15:c0.c1023** tcontext=system_u:object_r:init_var_run_t:s0 tclass=file permissive=1

The source context of the **process** that attempted the access. In this case, a process labeled as dnsmasq_t. The name of the process will also appear in the comm= field (dnsmasq with pid 305)

Target Context

audit: type=1400 audit(1595843096.039:3): avc: denied { getattr } for pid=305 comm="dnsmasq" **path="/run/systemd/resolve/resolv.conf" dev="tmpfs" ino=11079** scontext=system_u:system_r:dnsmasq_t:s0-s15:c0.c1023 **tcontext=system_u:object_r:init_var_run_t:s0 tclass=file** permissive=1

The target context of the file or process a process is attempting to access. In this case, the access is to a regular file labeled as init_var_run_t.

- For targets that are files:

If you are unsure what file has the context, the message may sometimes contain the full path or can sometimes give the inode (ino=) and filesystem type (dev=).

tclass= will say what kind of file this is. Regular files show up as 'file', but character files (chr_file), sockets, links, and directories (dir) will have different values.

- For targets that are processes:

If the scontext and tcontext are the same, you can use 'self' in place of the tcontext when writing rules.

Permissive

audit: type=1400 audit(1595843096.039:3): avc: denied { getattr } for pid=305 comm="dnsmasq" path="/run/systemd/resolve/resolv.conf" dev="tmpfs" ino=11079 scontext=system_u:system_r:dnsmasq_t:s0-s15:c0.c1023 tcontext=system_u:object_r:init_var_run_t:s0 tclass=file **permissive=1**

If the denial was thrown while the system was in permissive mode, then the denial **does not mean the access was blocked.** This is merely a warning that needs to be fixed before moving the system back to enforcing mode. If your service still doesn't work even when the system is in permissive mode, check for other problems first.

In enforcing mode, this value will appear as "permissive=0"

audit2allow

There is a tool that can be run to parse denial messages into usable policy rules that go into .te files (more info below).

On Ubuntu hosts, run the below command to install policycoreutils

```
sudo apt-get install policycoreutils
```

After installing, pull the policy file from the device to your host machine and copy your console logs to the host machine and run the below on the host machine to parse the logs

Below are the commands

```
adb pull /sys/fs/selinux
/policy                                                                                    # to pull the
policy file from device
adb wait-for-device shell dmesg | grep avc > policies.txt                # to get the denials and save it in
policies.txt file
audit2allow -i policies.txt -p policy                                                        #
run audit2allow tool
```

⚠ Auto-generated policies from this tool will likely be too permissive for merging. Linux Security team will likely suggest alternatives to these policies, especially for more complex modules. Please allow time for review and reimplementation

⚠ There is a bug in audit2allow tool provided by Ubuntu 16.04 (and maybe others). The first line will not be processed. To workaround this, duplicate the first line and save the updated log file before running audit2allow

⛔ Allow rules with a source context of 'initrc_t' will not be approved. This is an indication of an unlabeled service. You must write a policy module and transition the process to a new domain (see "Writing a Policy Module" below")

Note about dontaudit

There is a rule that can suppress denial messages while still blocking access. Typically rules like these are added if there is a denial message for an action that a policy owner did not consider essential for normal operations. If you suspect SELinux is not showing a denial, it may be because of this rule.

An example of when this is useful is when using the pidof tool. pidof will attempt to get the pid number associated with a process name. It does this by searching through /proc. If a process 'foo' has a pid of '123', then when pidof reads /proc/123/comm and sees the result of 'foo', it returns '123' to the caller. SELinux denials are generated for every attempt to read the process name of other processes as well (/proc/120/comm, /proc/121/comm, etc...) which we do not allow. So dontaudit can be used in this case to suppress log messages relating to searching for processes that would always fail this check anyways.

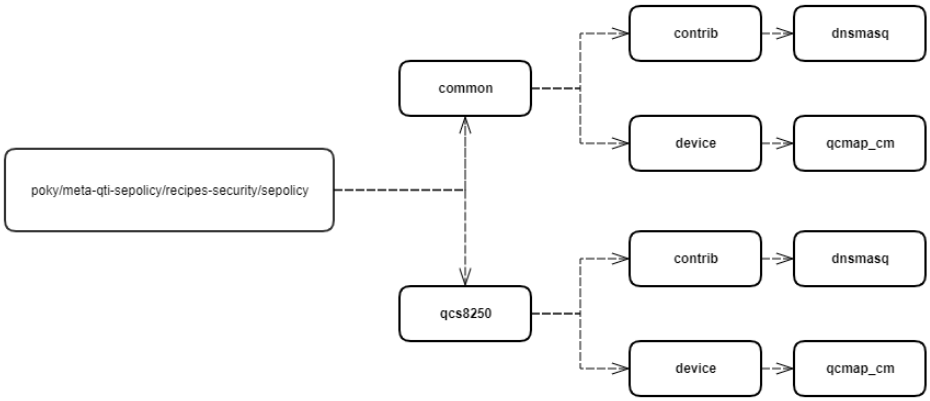
Writing a Policy Module

Where to Create Files

Policies are maintained centrally in poky/meta-qti-sepolicy/recipes-security/sepolicy.

- Policies common to all targets (MDM/IOT/Auto) go in the 'common' directory. Target-specific policies may go in the target specific directories
    - Fully QC-owned policies currently go into the 'device' subfolder. Overrides to upstream policies are provided in other directories.

For example:



Defining a Module

On LE targets, policy modules **must** contain 3 individual files ending with .fc, .te, and .if. The .fc and .if files can be blank (except for copyright markings), but **all 3 file types must be present for compilation to work**!

Type Enforcement (.te)

This file provides the actual rules that allow domains to make system calls

A typical .te file requires the below 4 lines at minimum (<module> has to be replaced name of the .te file without the .te suffix. Compilation of policy_module will fail if there is a mismatch)

```
policy_module(<module>, 1.0)
type <module>_t;
type <module>_exec_t;
init_vendor_domain(<module>_t, <module>_exec_t)
```

- policy_module declares a new policy module and a version string. You can update the version string if you want, but Linux Security team does not mandate this
  - This is only needed once per file
- init_vendor_domain is a macro for a set of rules that do the below:
  - When systemd (or any init system) executes a file labeled <module>_exec_t, the kernel will set the context of the new process to <module>_t
  - A separate rule is required for each unique context. While we would prefer having the filename and daemon name match, <module> can be whatever you want here.

There are a number of different keywords that are suitable for use in these files. Macros defined by upstream sources or by other QC teams typically expand into a collection of the below keywords.

Keyword	Description
type	Define a new type for use by the rest of the SELinux framework
allow	Grant access from one process to a file or another process
type_transition	Tell the kernel to create files with a specific SELinux context when a process with a particular SELinux context creates it. Extremely useful when creating new files at runtime
dontaudit	Block access and don't report a denial. Useful to hide false positives
neverallow	Do not allow compilation to succeed if a conflicting allow rule is present. This is a compile-time check, not a runtime check
attribute	Declare an attribute that can be used as a target for multiple different types when combined with typeattribute
typeattribute	Associates previously declared types to one or more previously declared attributes
auditallow	Throw an avc: granted message into the kernel logs whenever the access is attempted even if the access would otherwise be granted. **This does not allow the access in enforcing mode by itself**

File Contexts (.fc)

The file provides the labels that should be given to files on the filesystem. The syntax of the file is a whitespace separated table with 3 columns

File Path	File Type	Context
<path/to/your/binary>	--	gen_context(system_u:object_r:<module>_exec_t,s0)

- File path is the full path to the file as it appears on the target
    - If multiple rules could refer to the same path, the longest file path takes effect
    - Exactly duplicated rules cause a compilation failure
    - Regex is supported
- File type is a special code that indicates what type of file the rule should apply to. '--' matches with regular files. '-c' matches with characters files (there are other options as well). This field can also be left blank to match with any type of file.
- Context is a macro that expands to the label given to the file. system_u, object_r, and s0 should usually be left alone. The only part that changes is the <module>_exec_t part.

File paths are parsed as regex. The '.' character is actually a wildcard. If you actually meant to match on '.', make sure to escape by placing a backslash in front '\.'

Example

```
/usr/bin/test_diag -- gen_context(system_u:object_r:diag_exec_t,s0)
```

Interface (.if)

This file provides macros that other policy makers can use to communicate with your module.

Generally, the contents of this file are similar to those of the .te files. You can also leave this file blank if no other policy module is expected to communicate with either your process or the files your process manages. $1, $2, etc... can be used to accept parameters from callers to supply their own types (starting from 1, not 0).

Example

```
########################################
## <summary>
##       Read user data files
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
## <param name="type">
##      <summary>
##      File type allowed to read
##      </summary>
## </param>
#
interface(`read_user_data_files',`
    gen_require(`
        attribute user_data;
    ')

    search_user_data_dir($1)
    read_files_pattern($1, user_data, $2)
    read_lnk_files_pattern($1, user_data, $2)
')
```

Defining a File Context

The exact allowable rules here depend on where the file is created. A summary of commonly seen patterns is compiled below, but this is non-exhaustive. If you need access to another directory not seen here (e.g. /cache, /persist), then Linux Security team can help identify the right policies to use

/dev

/dev is a temporary kernel filesystem providing character files interacting with various kernel drivers

- Generic access to QC managed **device_t** character files will not be allowed. A unique label **must** be declared and used.
- Creation of files from userspace will not be allowed except in /dev/socket
    - The kernel is expected to create all of the necessary files in this filesystem. Userspace attempting to create any file here should be treated as a bug and fixed outside of SELinux.

In .te files, declare your custom type as a dev_node.

<module>.te

```
type <module>_dev_t;
dev_node(<module>_dev_t)


allow <module>_t <module>_dev_t:chr_file rw_chr_file_perms;
```

A common mistake is using **file** instead of **chr_file** here. For /dev in particular, making note of tclass= in the denial message is important!

<module>.fc

```
/dev/<module> -c gen_context(system_u:object_r:<module>_dev_t,s0)
```

In .fc files, files in /dev are typically character device nodes. Use the '-c' file type option here to help optimize the regex parsing.

For /dev/socket, an interface has been provided to simplify management of socket files. It allows your process to create the socket with your custom dev_node type and communicate with it.

device_manage_socket_file

```
interface(`device_manage_socket_file',`
    gen_require(`
        type socket_device_t;
    ')

    type_transition $1 socket_device_t:sock_file $2 $3;
    manage_sock_files_pattern($1, socket_device_t, $2)
')
```

<module>.te

```
type <module>_sock_t;
dev_node(<module>_sock_t)


# Use one of the below variants
device_manage_socket_file(<module>_t, <module>_sock_t)  # Only labels files
device_manage_socket_file(<module>_t, <module>_sock_t, { file dir })  # Label directories as well as files
```

/sys

For performance reasons, most files in sysfs are currently unlabeled, both from QC and from upstream. A future update may require unique labels for QC sysfs nodes

For now, use one of the below interfaces

dev_read_sysfs / dev_rw_sysfs

```
interface(`dev_read_sysfs',`
        gen_require(`
                type sysfs_t;
        ')
        read_files_pattern($1, sysfs_t, sysfs_t)
        read_lnk_files_pattern($1, sysfs_t, sysfs_t)
        list_dirs_pattern($1, sysfs_t, sysfs_t)
')

interface(`dev_rw_sysfs',`
        gen_require(`
                type sysfs_t;
        ')
        rw_files_pattern($1, sysfs_t, sysfs_t)
        read_lnk_files_pattern($1, sysfs_t, sysfs_t)
        list_dirs_pattern($1, sysfs_t, sysfs_t)
')
```

<module>.te

```
dev_read_sysfs(<module>_t)  # read-only
dev_rw_sysfs(<module>_t)  # read-write
```

/data

Files inside /data are currently treated differently from other files on the filesystem. This directory could contain sensitive user information. The usual generic access to files is blocked and instead, various macros have been provided to help track access to this directory.

Creating Files and Directories

Declare the data type in the modules .te and .fc files

<module>.te

```
type <module>_data_t;
user_data_files_type(<module>_data_t)
```

<module>.fc

```
/data/<module>/<module>.dat -- gen_context(system_u:object_r:<module>_data_t,s0)
```

Recommended

Due to performance constraints on NAND storage devices in particular, the recommended option for creating files and directories is to preinstall at least the parent directories during compilation and flash the directories as part of userdata image.

<module>.bb

```
do_install() {
        install -m 750 -o <user> -g <group> -d ${D}/<module>
}
```

Backup Option

If preinstalling the directory is not feasible, an interface has been provided to allow services to create files in /data with the appropriate SELinux labels. Your service will still need an alternate approach to assign UIDs and GIDs.

data_filetrans

```
interface(`data_filetrans',`
    gen_require(`
        type data_t;
    ')
    filetrans_pattern($1, data_t, $2, $3, $4)
')
```

<module>.te

```
data_filetrans(<module>_t, <module>_data_t)
```

You should still make the changes to <module>.fc as recommended above to avoid issues when restorecon is called during initial boot.

Managing Files

As the typical allow rules do not work on user_data_files_type files, another interface has been provided to manage the specific files your service manages

manage_user_data_files

```
interface(`manage_user_data_files',`
    gen_require(`
        type data_t;
        attribute user_data;
        attribute domain_can_write_userdata;
    ')

    read_user_data_files($1, $2)

    typeattribute $1 domain_can_write_userdata;
    allow $1 data_t:dir create;
    manage_dirs_pattern($1, user_data, $2)
    manage_files_pattern($1, user_data, $2)
    manage_lnk_files_pattern($1, user_data, $2)
    manage_sock_files_pattern($1, user_data, $2)
')
```

<module>.te

```
manage_user_data_files(<module>_t, <module>_data_t)
```

/etc

/etc is typically used for configuration files. By default, files in this directory are read-only.

Read Only

While a unique context is preferred, the default etc_t context can be used if there are complications with making a unique context

files_read_etc_files

```
interface(`files_read_etc_files',`
        gen_require(`
                type etc_t;
        ')
        allow $1 etc_t:dir list_dir_perms;
        read_files_pattern($1, etc_t, etc_t)
        read_lnk_files_pattern($1, etc_t, etc_t)
')
```

<module>.te

```
files_read_etc_files(<module>_t)
```

Writable

A mechanism (VOLATILE_BINDS) has been provided to make specific files writable

```
VOLATILE_BINDS = "\
/systemrw/adb_devid  /etc/adb_devid\n\
/systemrw/build.prop /etc/build.prop\n\
/systemrw/data /etc/data/\n\
/systemrw/data/adpl /etc/data/adpl/\n\
/systemrw/data/usb /etc/data/usb/\n\
/systemrw/data/miniupnpd /etc/data/miniupnpd/\n\
/systemrw/data/ipa /etc/data/ipa/\n\
/systemrw/rt_tables /etc/data/iproute2/rt_tables\n\
/systemrw/boot_hsusb_comp /etc/usb/boot_hsusb_comp\n\
/systemrw/boot_hsic_comp /etc/usb/boot_hsic_comp\n\
/systemrw/misc/wifi /etc/misc/wifi/\n\
/systemrw/bluetooth /etc/bluetooth/\n\
/systemrw/allplay /etc/allplay/\n\
"
```

Any file made writable in this way **must** have a unique SELinux context. All access controls in /systemrw are copied over from the /etc location by the framework.

<module>.te

```
type <module>_etc_t;
files_type(<module>_etc_t)

allow <module>_t <module>_etc_t:file rw_file_perms;
```

<module>.fc

```
/etc/<module>/<module>.conf -- gen_context(system_u:object_r:<module>_etc_t,s0)
```

/var/volatile/log

(symlinked from /var/log)

Files in this directory are writable but not persistent across reboots. Preinstalling access control rules is not possible here.

Recommended Approach

The recommended way to use systemd-tmpfiles to create a parent directory with the right uid, gid, and SELinux contexts. Child files created in these directories inherit access rules from the parent directory.

Backup Option

If this is not possible, use logging_log_filetrans to label the file as your process creates it. This will only address the SELinux context. You will need to find some alternate approach to setting the UID and GID of the file.

logging_log_filetrans

```
########################################
## <summary>
##        Create an object in the log directory, with a private type.
## </summary>
## <desc>
##        <p>
##        Allow the specified domain to create an object
##        in the general system log directories (e.g., /var/log)
##        with a private type.  Typically this is used for creating
##        private log files in /var/log with the private type instead
##        of the general system log type. To accomplish this goal,
##        either the program must be SELinux-aware, or use this interface.
##        </p>
##        <p>
##        Related interfaces:
##        </p>
##        <ul>
##                <li>logging_log_file()</li>
##        </ul>
##        <p>
##        Example usage with a domain that can create
##        and append to a private log file stored in the
##        general directories (e.g., /var/log):
##        </p>
##        <p>
##        type mylogfile_t;
##        logging_log_file(mylogfile_t)
##        allow mydomain_t mylogfile_t:file { create_file_perms append_file_perms };
##        logging_log_filetrans(mydomain_t, mylogfile_t, file)
##        </p>
## </desc>
## <param name="domain">
##        <summary>
##        Domain allowed access.
##        </summary>
## </param>
## <param name="private type">
##        <summary>
##        The type of the object to be created.
##        </summary>
## </param>
## <param name="object">
##        <summary>
##        The object class of the object being created.
##        </summary>
## </param>
## <param name="name" optional="true">
##        <summary>
##        The name of the object being created.
##        </summary>
## </param>
## <infoflow type="write" weight="10"/>
#
interface(`logging_log_filetrans',`
        gen_require(`
                type var_log_t;
        ')
        files_search_var($1)
        filetrans_pattern($1, var_log_t, $2, $3, $4)
')
```

<module>.te

```
type <module>_log_t;
logging_log_file(<module>_log_t)

# Use one of the below variants
logging_log_filetrans(<module>_t, <module>_log_t, { file dir })  # All 'file' or 'dir' objects created by
<module>_t in /var/volatile/log are labeled as <module>_log_t
logging_log_filetrans(<module>_t, <module>_log_t, file, "module.log")  # All files named "module.log" created
by <module>_t in /log are labeled as <module>_log_t

allow <module>_t <module>_log_t:file manage_file_perms;
```

/tmp

Files in this directory are writable but not persistent across reboots. Preinstalling access control rules is not possible here.

Recommended Approach

The recommended way to use systemd-tmpfiles to create a parent directory with the right uid, gid, and SELinux contexts. Child files created in these directories inherit access rules from the parent directory.

Backup Option

If this is not possible, use files_tmp_filetrans to label the file as your process creates it. This will only address the SELinux context. You will need to find some alternate approach to setting the UID and GID of the file.

files_tmp_filetrans

```
########################################
## <summary>
##        Create an object in the tmp directories, with a private
##        type using a type transition.
## </summary>
## <param name="domain">
##        <summary>
##        Domain allowed access.
##        </summary>
## </param>
## <param name="private type">
##        <summary>
##        The type of the object to be created.
##        </summary>
## </param>
## <param name="object">
##        <summary>
##        The object class of the object being created.
##        </summary>
## </param>
## <param name="name" optional="true">
##        <summary>
##        The name of the object being created.
##        </summary>
## </param>
#
interface(`files_tmp_filetrans',`
        gen_require(`
                type tmp_t;
        ')
        filetrans_pattern($1, tmp_t, $2, $3, $4)
')
```

<module>.te

```
type <module>_tmp_t;
files_tmp_file(<module>_tmp_t)

# Use one of the below variants
files_tmp_filetrans(<module>_t, <module>_tmp_t, { file dir })  # All 'file' or 'dir' objects created by
<module>_t in /tmp are labeled as <module>_tmp_t
files_tmp_filetrans(<module>_t, <module>_tmp_t, file, "module.tmp")  # All files named "module.tmp" created by
<module>_t in /tmp are labeled as <module>_tmp_t

allow <module>_t <module>_tmp_t:file manage_file_perms;
```

/run

(symlinked from /var/run)

Files in this directory are writable but not persistent across reboots. Preinstalling access control rules is not possible here.

Recommended Approach

The recommended way to use systemd-tmpfiles to create a parent directory with the right uid, gid, and SELinux contexts. Child files created in these directories inherit access rules from the parent directory.

Backup Option

If this is not possible, use files_pid_filetrans to label the file as your process creates it. This will only address the SELinux context. You will need to find some alternate approach to setting the UID and GID of the file.

files_pid_filetrans

```
########################################
## <summary>
##        Create an object in the process ID directory, with a private type.
## </summary>
## <desc>
##        <p>
##        Create an object in the process ID directory (e.g., /var/run)
##        with a private type.  Typically this is used for creating
##        private PID files in /var/run with the private type instead
##        of the general PID file type. To accomplish this goal,
##        either the program must be SELinux-aware, or use this interface.
##        </p>
##        <p>
##        Related interfaces:
##        </p>
##        <ul>
##                <li>files_pid_file()</li>
##        </ul>
##        <p>
##        Example usage with a domain that can create and
##        write its PID file with a private PID file type in the
##        /var/run directory:
##        </p>
##        <p>
##        type mypidfile_t;
##        files_pid_file(mypidfile_t)
##        allow mydomain_t mypidfile_t:file { create_file_perms write_file_perms };
##        files_pid_filetrans(mydomain_t, mypidfile_t, file)
##        </p>
## </desc>
## <param name="domain">
##        <summary>
##        Domain allowed access.
##        </summary>
## </param>
## <param name="private type">
##        <summary>
##        The type of the object to be created.
##        </summary>
## </param>
## <param name="object">
##        <summary>
##        The object class of the object being created.
##        </summary>
## </param>
## <param name="name" optional="true">
##        <summary>
##        The name of the object being created.
##        </summary>
## </param>
## <infoflow type="write" weight="10"/>
#
interface(`files_pid_filetrans',`
        gen_require(`
                type var_t, var_run_t;
        ')
        allow $1 var_t:dir search_dir_perms;
        allow $1 var_run_t:lnk_file read_lnk_file_perms;
        filetrans_pattern($1, var_run_t, $2, $3, $4)
')
```

Upstream has renamed the files_pid_* interfaces to files_runtime_* in the latest branches. We will be making this change in QC policies as well after the upstream policies have been pulled in for an OS update

<module>.te

```
type <module>_run_t;
files_pid_file(<module>_run_t)

# Use one of the below variants
files_pid_filetrans(<module>_t, <module>_run_t, { file dir })  # All 'file' or 'dir' objects created by
<module>_t in /run are labeled as <module>_run_t
files_pid_filetrans(<module>_t, <module>_run_t, file, "module.pid")  # All files named "module.pid" created by
<module>_t in /run are labeled as <module>_run_t

allow <module>_t <module>_tmp_t:file manage_file_perms;
```

/proc

/proc/<pid>

The various /proc/<pid> folders contain information specific to each process. This information is generally kept private and is therefore sandboxed from other processes.

> Attempts to allow **generic** access to all /proc/<pid> folders will not be approved. You must provide the destination context(s) of the process your service is trying to access and restrict access to only contexts that are necessary.

These folders are labeled with **process** contexts instead of file contexts like in almost all other directories. Most processes need to access one or more files inside their own /proc/<pid> folder. If this is the case, then you may use the special keyword 'self' instead of the full name of the target context when writing allow rules.

```
allow <module>_t self:fifo_socket rw_fifo_file_perms;
```

If your service is trying to find another service and scans through /proc/<pid> folders to find the processes' information, you may use dontaudit to suppress denial messages shown for processes your service does not otherwise interact with.

```
allow <module>_t <other_module>_t:file read_file_perms;
dontaudit <module>_t domain:file read_file_perms;
```

In this example, if we assume a process running in <module>_t context has a pid of '123' and a process running in <other_module>_t context has a pid of '255', then <module>_t is allowed to read /proc/255/comm but is blocked from reading /proc/1/comm which always has a context of init_t. No denials will be reported in logs as this would be expected behavior.

/proc (all other non pid folders)

Access to general information about the system is usually safe to access and an interface has been provided upstream to allow this access. We typically allow read access to this, although writing may need discussion

kernel_read_system_state

```
interface(`kernel_read_system_state',`
        gen_require(`
                type proc_t;
        ')
        read_files_pattern($1, proc_t, proc_t)
        read_lnk_files_pattern($1, proc_t, proc_t)
        list_dirs_pattern($1, proc_t, proc_t)
')
```

<module>.te

```
kernel_read_system_state(<module>_t)
```

References

- qwiki documentation (older): http://qwiki.qualcomm.com/quic/SELinux_LE
- Presentation of LE-Sepolicy (basics) – Pdf
- FAQs : LE-SElinux-FAQ
- UserSpace utils @ https://github.com/SELinuxProject/selinux/releases