SELinux guide for QCLINUX

1. 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

2. Operations Guide

2.1.1. Enable SELinux

Currently SELinux is enabled in LE.QCLINUX.1.0 internally for external release it is disabled.

2.1.2. 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'

2.1.3. 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
    - adb umount -l /etc
    - 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 in **meta-qti-internal/conf/distro/include/qcom-internal-selinux.inc** to one of the three supported values: 'enforcing', 'permissive', or 'disabled'.

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.

3. **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"

3.1.1. Generate policies using audit2allow

Note: Policy version 33 is supported from ubuntu 22. So use ubuntu 22.04 version

- **Install selinux utilities**

Dockerfile

```
RUN sudo apt update

RUN sudo apt install policycoreutils selinux-utils selinux-basics
```

- **Verify installation:**

Docker Build commands

```
audit2allow -h
```

Note: You can use policy.33 file from the AU build folder for running audit2allow.

Below are the commands to run audit2allow:

```
audit2allow -i policies.txt -p policy.33                                                    # run
audit2allow tool

            where
                    'policies.txt' - is the file with list of denials.
                    'policy.33'    - the file from the build artifacts.
policy.33 is part of build tree (which gets build ) and located @ build-qcom-wayland/tmp-glibc/sysroots-
components/<machine>/refpolicy-targeted/etc/selinux/targeted/policy/
```

3.1.2. Restrictions / Guidelines for sepolicy writers

Permission which are not to be used

execmod

execmem

execute_no_trans

relabel

mount

setuid

setgid

dac_search /override

write to procfs /sysfs by userspace application until unless its justified and agreed with security team .

3.1.3. 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.

- **Checking for dontaudits:**

```
sesearch --dontaudit
```

4. **Writing a Policy Module**

Where to Create Files

- Policies are maintained centrally in "**layers/meta-qti-bsp/dynamic-layers/selinux/recipes-security/sepolicy/"** residing in LE.QCLINUX.1.0 for downstream services.
- If a service was upstreamed, corresponding policies should be uploaded to "**layers/meta-qti-distro/dynamic-layers/selinux/recipes-security /sepolicy/**" as a patch file, which will be upstreamed to refpolicy GitHub - SELinuxProject/refpolicy: SELinux Reference Policy v2 further.
- Upstream categorized sepolicies into five categories : apps, kernel, roles, system, services. Will be following similar source tree and some additional folders described below.
- **Apps**
    - This folder contains the policies of the applications. In our case, example: test apps, clients.

  Kernel
    - This folder contains the policies specific to filesystem(eg: procfs, sysfs, debugfs, functionfs, configfs, binderfs etc..), device, storage, network and some of the core kernel subsystems. Common nodes present across Qualcomm targets should be added in this folder.
        - device.*
            - dev node, sysfs nodes labelling.
            - dev nodes must be labelled to read or write.
            - read access will be given globally to read sysfs nodes.
            - write access will be given to sysfs nodes only for which specific label is given.
    -
        - filesystem.*
            - filesystem specific policies and labelling.
    -
        - files.*
            - types and interface for the basic filesystem layout (/, /etc, /tmp, /usr, etc.)
    -
        - domain.*
            - Core policies for domains.

  System
    - This folder contains the policies of the system initializers, such as
        - The Init systems like:
            - systemd or sysvinit.
            - Utilities that can do modifications to the system like:
            - modprobe, mount, ipsec, iptables ..etc.,

Target

- This folder contains the policies of the QC specific dev, sys and proc nodes that differ across QC targets.
- We will be maintaining target specific policy files for each target with MACHINE as file name.
- And respective target specific policy files will be picked while compilation.
- Example:

- For Kodiak, we will be adding policy files : qcm6490.te, qcm6490.if, qcm6490.fc

Services

- This folder contains the policies of the user space services like: ssh, chronyd etc.,

Admin

- This folder contains the policies of the services that run at admin privilege like: shutdown, dmesg, VPN..etc.

Roles

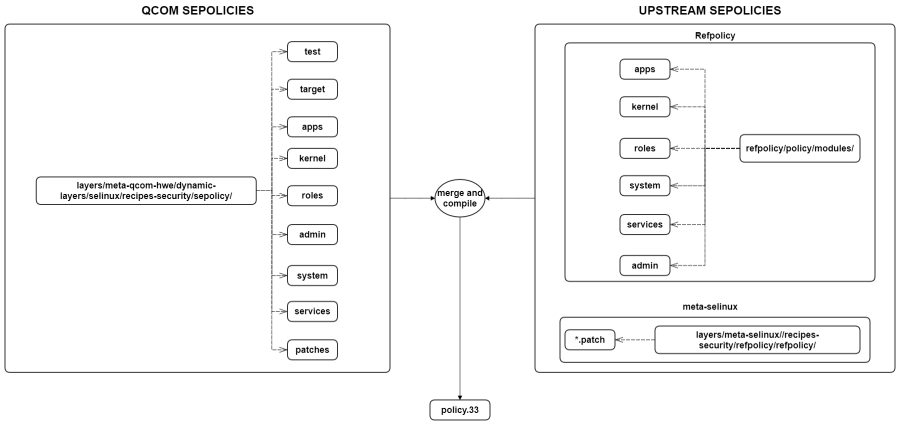
- To set permissions for user roles as sysadm, staff, unconfined, etc

Patches

- This folder contains the policies of the Qualcomm up-streamed services in the form of patches.

Test

- This folder contains test policies for the test binaries specific to targets and common to all targets.



4.1.1. 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(qcom_<module>, 1.0)
type qcom_<module>_t;
type qcom_<module>_exec_t;
init_daemon_domain(qcom_<module>_t, qcom_<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_daemon_domain is a macro for a set of rules that do the below:
  - When systemd (or any init system) executes a file labeled qcom_<module>_exec_t, the kernel will set the context of the new process to qcom_<module>_t.
  - A separate rule is required for each unique context. We want to name the module with respective service name match, Eg: qcom_<module> , Where 'module' should be service name.

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:qcom_<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 qcom_<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 '\.'

4.1.1.1. Example

```
/usr/bin/test_diag -- gen_context(system_u:object_r:qcom_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. If you don't have any changes in this file still you should this create if corresponding . te, .fc exists and also the below commented xml code snippet in the example should be added which will be used to make decision whether the module should be compiled as base module or as a separate module.

4.1.1.2. Example-1: To build policies of a service as a base module.

xml code snippet for the policies of a service to build as base module

```
## <summary>
## Explain what module is for, this content has to be added at the start of the file (just after copyright)
## </summary>
## <required val="true">
##      This required tag makes the module as base.
## </required>
```

4.1.1.3. Example-1: To build policies of a service as a separate module.

xml code snippet to policies of a service as a separate module

```
## <summary>
## Explain what module is for, this content has to be added at the start of the file (just after copyright). Do
not add required tag
## </summary>
```

4.1.2. 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.

qcom_<module>.te

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

allow qcom_<module>_t qcom_<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!

qcom_<module>.fc

```
/dev/<module> -c gen_context(system_u:object_r:qcom_<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.

/sys

For now, any service can have read access on sysfs.

dev_read_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)
')
```

qcom_<module>.te

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

If any QC service needs both read and write access on a QC owned node, then

- The node must be labelled in kernel/filesystem.fc file.
- A custom interface must be defined with read and write access in kernel/filesystem.if file.
- And the newly defined custom interface must be called from service specific .te file

filesystem.te

```
type qcom_<node_name>_t;
genfscon sysfs /kernel/rtimd/<node_name>    gen_context(system_u:object_r:qcom_<node_name>_t,s0)
dev_associate_sysfs(qcom_<node_name>_t)
```

filesystem.if

```
########################################
## <summary>
##      Allow caller to modify <hardware> state information.
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary>
## </param>
#
interface(`qcom_dev_rw_sysfs_<node_name>',`
        gen_require(`
                type qcom_<node_name>_t;
        ')

        rw_files_pattern($1, qcom_<node_name>_t, qcom_<node_name>_t)
        read_lnk_files_pattern($1, qcom_<node_name>_t, qcom_<node_name>_t)

        list_dirs_pattern($1, qcom_<node_name>_t, qcom_<node_name>_t)
')
```

qcom_<module>.te

```
qcom_dev_rw_sysfs_<node_name>(<module>_t)  # read-write
```

/etc

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

4.1.2.1. 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)
')
```

qcom_<module>.te

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

4.1.2.2. 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.

qcom_<module>.te

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

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

qcom_<module>.fc

```
/etc/<module>/<module>.conf -- gen_context(system_u:object_r:qcom_<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.

4.1.2.3. 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.

4.1.2.4. 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)
')
```

qcom_<module>.te

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

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

allow qcom_<module>_t qcom_<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.

4.1.2.5. 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.

4.1.2.6. 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)
')
```

qcom_<module>.te

```
type qcom_<module>_t;
type qcom_<module>_tmp_t;

fs_associate_tmpfs(qcom_<module>_tmp_t)
files_tmp_filetrans(qcom_<module>_t, qcom_<module>_tmp_t, file)

manage_files_pattern(qcom_<module>_t, qcom_<module>_tmp_t, qcom_<module>_tmp_t)
```

/run

(symlinked from /var/run)

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

4.1.2.7. 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.

4.1.2.8. 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 qcom_<module>_run_t;
files_runtime_file(qcom_<module>_run_t)

files_runtime_filetrans(qcom_<module>_t, qcom_<module>_run_t, {file sock_file dir})

allow qcom_<module>_t qcom_<module>_run_t:file manage_file_perms;
```

/proc

4.1.2.9. /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 qcom_<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 qcom_<module>_t qcom_<other_module>_t:file read_file_perms;
dontaudit qcom_<module>_t domain:file read_file_perms;
```

In this example, if we assume a process running in qcom_<module>_t context has a pid of '123' and a process running in qcom_<other_module>_t context has a pid of '255', then qcom_<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.

4.1.2.10. /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)
')
```

qcom_<module>.te

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

5. General Guidelines

- Every tech team is expected to create their own folder under /data, /persist, /etc, /bin etc and can be given full control.

- Any resource that (socket, dev node, files or folder) that you think are going to be access by other tech team, we expect you to create an Interface in corresponding .if file for doing so.,
- **Interface should have the allow/dontaudit rule required, where scontext should be passed while calling the interface:**

  allow <scontext> tcontext:tclass

  Unknown macro: {permissions}

  ;

6. Build policies

Currently there are two policy designs provided by upstream refpolicy.

1. Monolithic - Single policy binary file (policy.33) will be generated with all the policies.
2. Modular - A seperate policy binary file(qcom_<module>.pp) will be generated for each policy module.

For now we will be supporting Monolithic approach. We may support Modular approach in future. Keeping this mind whenever we write policies, we must make sure the those policies are getting compiled with both Monolithic and Modular approach.

6.1. Build policies with Monolithic approach

For now, we configured policy design to Monolithic by configuring the variable "POLICY_MONOLITHIC" (Line:1 path: meta-qti-bsp/dynamic-layers /selinux/recipes/security/sepolicy/refpolicy-targeted_git.bbappend) to "y". So, after writing policies you can simply build the policies issuing the command "**bi tbake refpolicy-targeted".**

6.2. Build policies with Modular approach

You need to configure the variable "POLICY_MONOLITHIC " (Line:1 path: meta-qti-bsp/dynamic-layers/selinux/recipes/security/sepolicy/refpolicy-targeted_git.bbappend) to "n" and build the policies issuing **"bitbake refpolicy-targeted".**

Following is the cmd that can be used for building sepolicy (both Moolithic and Modular) after modifying POLICY_MONOLITHIC flag. Every tech team/change contributor is expected to get this tested before approvals.

export SHELL=/bin/bash && MACHINE=<Machine_id> DISTRO=qcom-wayland source setup-environment

bitbake refpolicy-targeted -c cleansstate;bitbake refpolicy-targeted

Supported Machines:

- qcm6490-idp
- qcs6490-rb3gen2-vision-kit
- qcs6490-rb3gen2-industrial-kit
- qcs6490-rb3gen2-core-kit
- qcs9100-ride-sx
- qcs8300-ride-sx
- qcs9075-ride-sx
- qcs9075-iq-9075-evk
- qcs9075-iq-9075-evk-ifp
- qcs8275-iq-8275-evk
- qcs8275-iq-8275-evk-ifp
- qcs8275-iq-8275-evk-pro-sku

Modular Approach: IPK loading on K2C - Linux Security - Qualcomm Confluence

7. Upstreaming policies

Note:

- All QC Upstream/downstream services need to work with enforcing.
- Downstream services maintain policies in meta-qti-bsp.
- Policies for Upstreamed QC services must be up-streamed to refpolicy.
- And upstreamed policies will be maintained as patches in meta-qti-bsp till the latest refpolicy release tag gets updated in the meta-selinux.

7.1. Which policies needs to be upstreamed?

If both the services of scontext and tcontext are upstreamed then the corresponding policies needs to be upstreamed.

Example:

Denial:

[11.428827] audit: type=1400 audit(1651167746.455:3): avc: denied { getattr } for pid=440 comm="systemd-sysv-ge" path="/etc/init.d/chronyd" dev="sda5" ino=200 **scontext=system_u:system_r:systemd_generator_t:s0-s15:c0.c1023** tcontext=system_u:object_r:**chronyd_initrc_exec_t**:s0 tclass=file permissive=1[11.457051] audit: type=1400

Policy:

allow systemd_generator_t chronyd_initrc_exec_t:file getattr;

7.2. **Patch generation instructions**

- We need to generate two patches.
  - One patch for internal usage.
    - Should be raised on top of refpolicy tag updated in scarthgap meta-selinux.
  - Another patch to upstream the policies.
    - Should be raised on tip of refpolicy.
- **Steps to sync refpolicy and generate patches for internal usage**
  - Clone refpolicy and apply patches.

    # git clone https://github.com/SELinuxProject/refpolicy.git

    # git reset --hard <commit_id> (Commit Id for scarthgap: 71f4bd1992e05bcd79dc5234f8a30deeb141aa3d)

  - Make your changes, commit and generate patch.

    # git add .

    # git commit -m "commit message"

    # git format-patch -1 HEAD

  - Update the patch number in the name of the patch file by looking at the patches in **LE.QCLINUX.1.0/layers/meta-selinux/recipes-security/refpolicy/refpolicy/**
    - For suppose your patch name "**0001-dummy.patch**" and the latest patch in **LE.QCLINUX.1.0/layers/meta-selinux/recipes-security/refpolicy/refpolicy/ & LE.QCLINUX.1.0/layers/meta-qti-bsp/dynamic-layers/selinux/recipes-security/sepolicy/patches** is "**0069-fc-fstools-apply-policy-to-findfs-alternative.patch**". The update the patch number as "**0070-dummy.patch**"
  - Copy the patch to **LE.QCLINUX.1.0/layers/meta-qti-bsp/dynamic-layers/selinux/recipes-security/sepolicy/patches.**
  - Build image and verify.
- **Steps to generate and submit patch for upstream refpolicy**
  - Clone refpolicy

    # git clone https://github.com/SELinuxProject/refpolicy.git

  - Add changes, generate patch and submit to upstream using quicinc mail.
  - **Contribution guidelines from upstream:** HowToContribute · SELinuxProject/refpolicy Wiki · GitHub

8. Review Guideline for sepolicy

Following talks about general guideline for +2 approvals. There might be exception and amendments based on usecases: K2L SEPolicies review guidelines

9. Using seinfo tool

- Status of selinux subsystem. "seinfo": this will give much info like below snapshot
  - *sh-5.1# seinfo*

    Statistics for policy file: /sys/fs/selinux/policy <<< policy file referred for info

    Policy Version: 33 (MLS enabled) <<< shows version

    Target Policy: selinux

    Handle unknown classes: allow <<< if unknow class is seen should allow or reject (better setting reject in production)

    Classes: 131 Permissions: 423 <<< count of classes and permission

    Sensitivities: 16 Categories: 1024

    Types: 4347 Attributes: 319

    Users: 6 Roles: 14

Booleans: 338 Cond. Expr.: 359

Allow: 75162 Neverallow: 0

Auditallow: 23 Dontaudit: 15592

Type_trans: 10278 Type_change: 85

Type_member: 16 Range_trans: 36

Role allow: 30 Role_trans: 428

Constraints: 64 Validatetrans: 0

MLS Constrain: 227 MLS Val. Tran: 17

Permissives: 13 Polcap: 5 <<< number of domain in permissive

Defaults: 0 Typebounds: 0

Allowxperm: 0 Neverallowxperm: 0

Auditallowxperm: 0 Dontauditxperm: 0

Ibendportcon: 0 Ibpkeycon: 0

Initial SIDs: 27 Fs_use: 29

Genfscon: 94 Portcon: 475

Netifcon: 1 Nodecon: 0

- Check list of service running in permissive "seinfo --permissive ". You can check help for other cmds.

10. Modular SEpolicy model :

SElinux can be compiled in 2 modes as monolithic sepolicy (single sepolicy image) and Modular sepolicy (each module is going to be separated as <module name>.pp).

Tech teams are expected to make sure we are always compliance with both the modes. For additional details you can visit: go/modular-sepolicy

11. OSTree and SELinux :

go/ostree-selinux

12. References

- qwiki documentation (older): http://qwiki.qualcomm.com/quic/SELinux_LE
- FAQs : LE-SElinux-FAQ
- UserSpace utils @ https://github.com/SELinuxProject/selinux/releases
- Issue in upstream and handling : https://github.com/SELinuxProject/refpolicy/blob/main/SECURITY.md
- General security guide :https://www.konsulko.com/wp-content/uploads/2020/07/DD5_Security_Hardening_NA20.pdf
- Link to the K2C selinux Sessions (08 Nov 2023 if you face access issue please try joining linux.security.training group)
- Other useful internal links :
  - Security FR – page tracking FR timeline
  - Apps Images page --published images
  - go/k2cdocs — early days guildeline on upstreaming email id policy .
  - go/K2LOpenAIs --Documentation AI tracker
  - LINT/Image page
  - SEPolicy review guidelines for K2L
  - Linux Kernel platform