SELinux for Android 8.0

Changes & Customizations

Table of Contents

Overview

This document describes SELinux changes and customizations designed to support modularity and updatability of SELinux policy in Android 8.0. The goal of these changes is to enable System on Chip (SoC) vendors and Original Device Manufacturer (ODM) partners to customize SELinux settings in an isolated manner without cross-partition modifications.

Design goals

The SELinux policy build flow for Android 4.4 through Android 7.0 merged all sepolicy fragments (platform and non-platform) then generated monolithic files in the root directory. However, this flow contradicts the primary goal of Android 8.0 architecture, which is to allow partners to update their parts of the policy, build their images (`vendor.img`, `boot.img`, etc), then update those images independent of the platform or vice versa (i.e., perform a platform update without updating partner images).

Android 8.0 design goals are:

- **Policy Modularization**. In Android 4.4 through Android 7.0, most SELinux files resided in `rootfs`, thus SoC vendors and the ODM partners modified `boot.img` (for non-A/B devices) or `system.img` (for A/B devices) every time policy was modified. The Android 8.0 model provides a method for vendors and partners to change only their partitions when they need to modify their portions of the SELinux policy.
- **Policy Compatibility**. On devices running Android 8.0, it is possible to upgrade the platform image ahead of vendor/partner images; this can occur during an over-the-air (OTA) update, such as a framework OTA.

While it is possible to have higher/newer platform (framework) version running on the device, the opposite case is not supported; i.e., the non-platform images (`vendor.img/odm.img`) cannot have a new version than the platform (`system.img`). So, a newer platform version might introduce SELinux compatibility issues because the platform SELinux policy is at a newer version than vendor/partner SELinux parts of the policy. The Android 8.0 model provides a method to retain compatibility to prevent unnecessary simultaneous OTAs.

About Android 8.0 architecture

An Android device includes the following partitions:

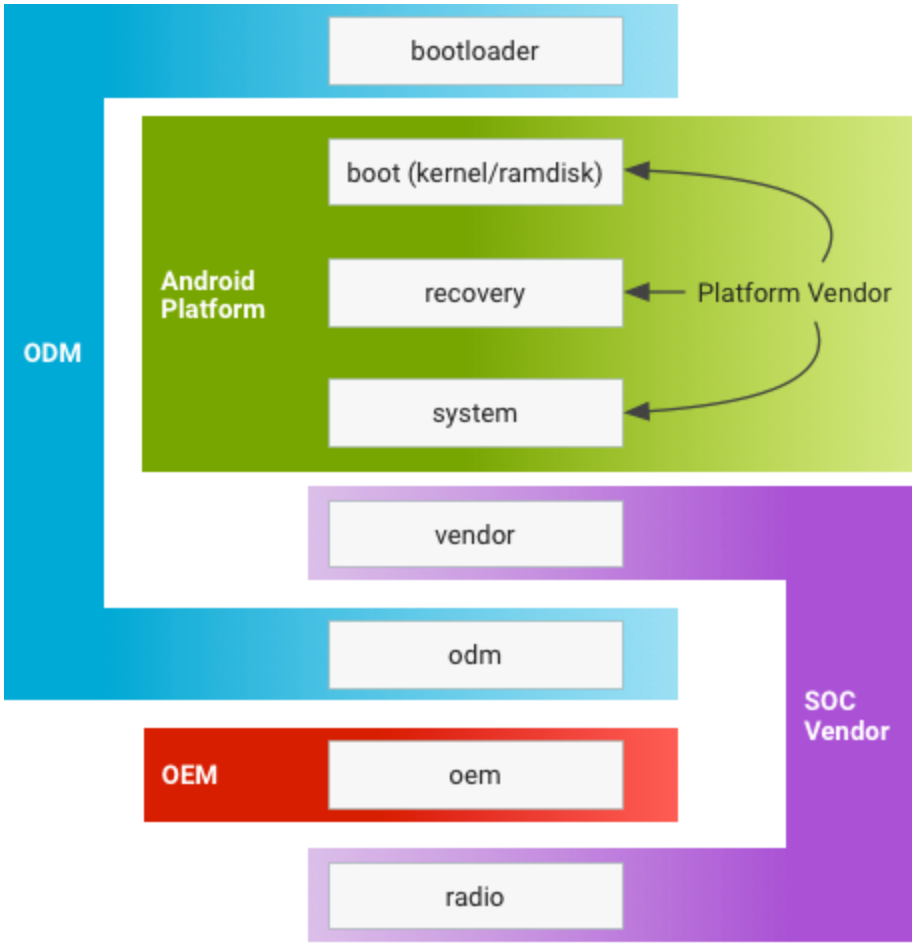

Figure 1. Android partitions.

- `system.img`. Contains mainly Android framework.
- `boot.img`. (kernel/ramdisk) Contains Linux kernel + Android patches.
- `vendor.img`. Contains SoC-specific code and configurations.
- `odm.img`. Contains device-specific code and configurations.

- `oem.img`. Contains OEM/carrier-related configurations and customizations.
- `bootloader`. Brings up the kernel (vendor-proprietary).
- `radio`. Modem (proprietary).

Prior to Android 8.0, the `vendor`, `odm`, and `oem` images were **optional**; files belonging to these images were placed in `boot.img` or `system.img` with symlinks (such as `/vendor` > `/system/vendor`) when absent. Android 8.0 makes the vendor partition **mandatory**.

The goal is to modularize Android partitions and make them interchangeable by defining a core, standard interface between the Android Platform (on `system.img`) and vendor-provided code. This standard interface enables the Android Platform to be updated without affecting the SoC and ODM partitions. For example, it should be possible to upgrade a device `system.img` from Android 8.0 to Android P while other images (such as `vendor.img`, `odm.img`, etc.) remain at Android 8.0. This modularity enables timely Android platform upgrades (such as monthly security updates) without requiring SoC/ODM partners to update SoC- and device-specific code.

About SELinux

SELinux is a labeling system that controls the permissions (read/write, etc.) a subject context has over a target object such as *`directory/device/file/process/socket/`*. (For an analogy, refer to Your visual how-to guide for SELinux policy enforcement.)

Each process and object has an associated label, which is also called a **context.** Contexts are comprised of a user, a role, a type and an multi-level-security (MLS) portion:

- The **type** of a process is often referred to as a **domain** and is defined in SELinux policy.
- The label of an **object** is usually decided by the corresponding security-contexts files.

SELinux policy also contains the rules that state how each domain may access each object. In Android 4.4 → Android 7.0, SELinux policy files (`sepolicy`, `file_contexts.bin`, `property_contexts etc`) are included the rootfs image as follows:

```
/
.
.
├── file_contexts.bin (file_contexts pre-N)
├── property_contexts
├── seapp_contexts
├── sepolicy
├── service_contexts
.
```

These files contain SELinux policy rules and labels from **all** development participants, including ODM, SoC, and AOSP. In Android 8.0, these files are modular.

SELinux for Android 7.x.

The following sections describe the SELinux policy and contexts build flow for Android 7.0.

SELinux source files

SELinux customization involves the following files:

- external/selinux: External SELinux project, used to build HOST command line utilities to compile SELinux policy and labels.
    - external/selinux/libselinux: Android uses only a subset of the external libselinux project along with some Android specific customizations. For details, refer to external/selinux/README.android.
    - external/selinux/libsepol:
        - chkcon: Determine if a security context is valid for a given binary policy (host executable).
        - libsepol: SELinux library for manipulating binary security policies (host static/shared library, target static library).
    - external/selinux/checkpolicy: SELinux policy compiler (host executables: checkpolicy, checkmodule, and dispol). Depends on libsepol.
- system/sepolicy: Core Android SELinux policy configurations, including contexts and policy files. Major sepolicy build logic is also here (`system/sepolicy/Android.mk`).

SELinux build logic

SELinux policy is created by combining the core AOSP policy with device-specific customizations. The combined policy is then passed to the policy compiler and various checkers. Device-specific customization is done through the **BOARD_SEPOLICY_DIRS** variable defined in device-specific Boardconfig.mk file. This global build variable contains a list of directories that specify the order in which to search for additional policy files.

For example, a SoC vendor and an ODM might each add a directory, one for the SoC-specific settings and another for device-specific settings, to generate the final SELinux configurations for a given device:

- `BOARD_SEPOLICY_DIRS += device/$SoC/common/sepolicy`
- `BOARD_SEPOLICY_DIRS += device/$SoC/$DEVICE/sepolicy`

The content of `file_contexts` files in `system/sepolicy` and `BOARD_SEPOLICY_DIRS` are concatenated to generate the `file_contexts.bin` on the device:

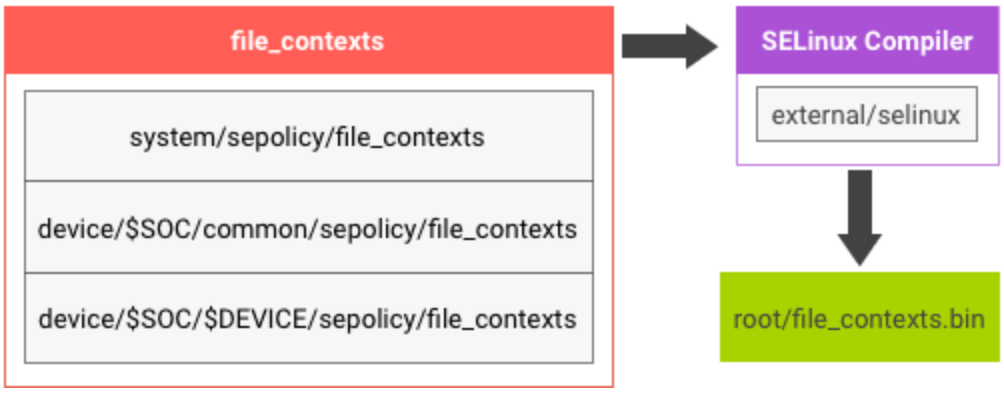

Figure 2. SELinux build logic.

The **sepolicy** file consists of multiple source files:

- The plain text `policy.conf` is generated by concatenating `security_classes`, `initial_sids`, ... *.`te`, `genfs_contexts`, and `port_contexts` in that order.
- For each file (e.g., `security_classes`), its content is the concatenation of the files with the same name under `system/sepolicy/` and `BOARDS_SEPOLICY_DIRS`.
- The `policy.conf` is sent to SELinux compiler for syntax checking and compiled into binary format as sepolicy on the device.

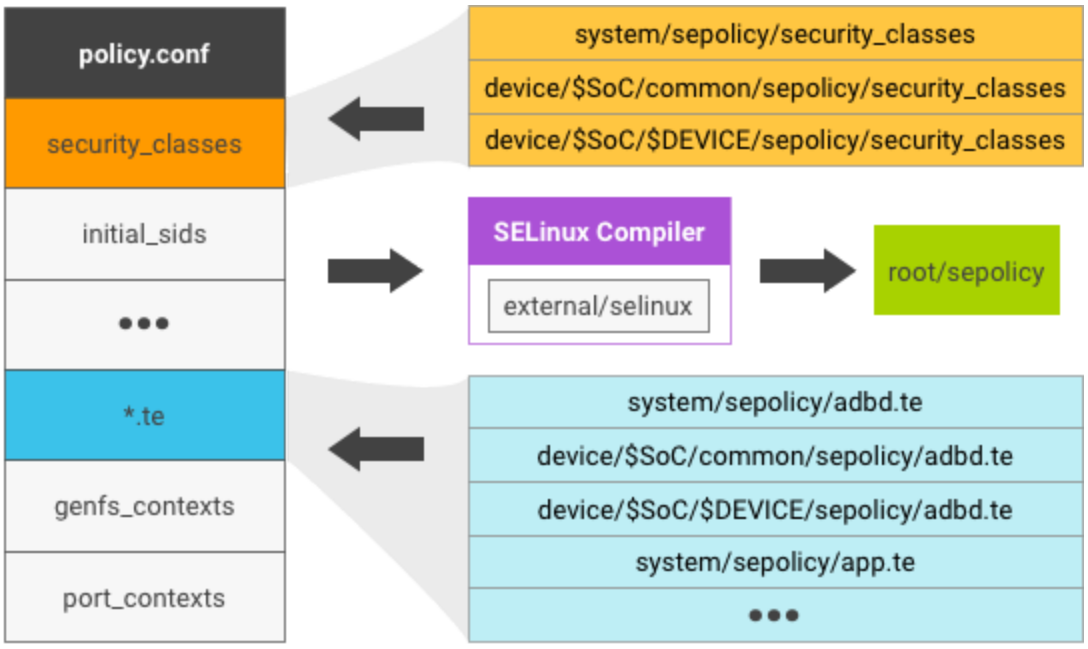

Figure 3. SELinux policy file.

SELinux files

Android devices typically contain the following SELinux-related files:

- `selinux_version`
- `sepolicy`: binary output after combining policy files (`security_classes`, `initial_sids`, *.`te`, etc.)
- `file_contexts`
- `property_contexts`
- `seapp_contexts`
- `service_contexts`
- `system/etc/mac_permissions.xml`

For more details, refer to Implementing SELinux on source.android.com.

SELinux initialization

When the system boots up, SELinux is in *permissive* mode (and not in *enforcing* mode). The init process performs the following tasks:

1. Loads sepolicy files from ramdisk into the kernel through `/sys/fs/selinux/load`.
2. Switches SELinux to enforcing mode.
3. Re-`exec()`s itself to apply the SELinux domain rule to itself.

To shorten the boot time, perform the re-`exec()` on the init process as soon as possible.

Android 8.0 SELinux design

The following sections describe the SELinux design considerations and changes to support the design goals (policy modularity and partial updatability) of Android 8.0.

First stage mount

Before Android 8.0, SELinux files were built by merging device-specific (i.e. non-platform) policy files with the Android (i.e. platform) policy files monolithically. Android 8.0 provides a method to keep these **non-platform** changes separate from the **platform** SELinux policy so partners can build and update settings independently.

As modularized SELinux policy files are stored on partner partitions (e.g. `/vendor`), the init process must mount the `system` and `vendor` partitions earlier so it can read SELinux files

from those partitions and merge them with core SELinux files in the `system` directory (before loading them into the kernel). For details, refer to the change that mounts these partitions earlier.

SELinux contexts labeling

File contexts

Android 8.0 introduces the following changes for `file_contexts`:

- To avoid additional compilation overhead on device during boot, `file_contexts` cease to exist in the binary form. Instead, they are readable, regular expression text file such as `{property, service}_contexts` (as they were pre-7.0).
- The `file_contexts` are split between two files:
  - `Plat_file_contexts`
    - Android platform `file_context` that has no device-specific labels, except for labeling parts of `/vendor` partition that must be labeled precisely to ensure proper functioning of the platform (e.g. the sepolicy files).
    - Must reside in `system` partition at `/system/etc/selinux/plat_file_contexts` on device and be loaded by `init` at the start along with the non-platform `file_context`.
  - `Nonplat_file_contexts`
    - Device-specific `file_context` built by combining `file_contexts` found in the directories pointed to by `BOARD_SEPOLICY_DIRS` in the device's `Boardconfig.mk` files.
    - Must be installed at `/vendor/etc/selinux/nonplat_file_contexts` in `vendor` partition and be loaded by `init` at the start along with the platform `file_context`.

Property contexts

In Android 8.0, the `property_contexts` is split between two files:

- `plat_property_contexts`
  - Android platform `property_context` that has no device-specific labels.
  - Must reside in `system` partition at `/system/etc/selinux/plat_property_contexts` and be loaded by `init` at the start along with the non-platform `property_contexts`.

- nonplat_property_contexts
  - Device-specific property_context built by combining property_contexts found in the directories pointed to by BOARD_SEPOLICY_DIRS in device's Boardconfig.mk files.
  - Must reside in vendor partition at /vendor/etc/selinux/nonplat_property_contexts and be loaded by init at the start along with the platform property_context

Service contexts

In Android 8.0, the service_contexts is split between the following files:

- plat_service_contexts
  - Android platform-specific service_context for the servicemanager. The service_context has no device-specific labels.
  - Must reside in system partition at /system/etc/selinux/plat_service_contexts and be loaded by servicemanager at the start along with the non-platform service_contexts.
- nonplat_service_contexts
  - Device-specific service_context built by combining service_contexts found in the directories pointed to by BOARD_SEPOLICY_DIRS in the device's Boardconfig.mk files.
  - Must reside in vendor partition at /vendor/etc/selinux/nonplat_service_contexts and be loaded by servicemanager at the start along with the platform service_contexts
  - Although servicemanager looks for this file at boot time, for a fully compliant TREBLE device, the nonplat_service_contexts MUST not exist. This is because all interaction between vendor and system processes MUST go through hwservicemanager/hwbinder.
- plat_hwservice_contexts
  - Android platform hwservice_context for hwservicemanager that has no device specific labels.
  - Must reside in system partition at /system/etc/selinux/plat_hwservice_contexts and be loaded by hwservicemanager at the start along with the nonplat_hwservice_contexts.
- nonplat_hwservice_contexts

    - Device-specific `hwservice_context` built by combining `hwservice_contexts` found in the directories pointed to by `BOARD_SEPOLICY_DIRS` in the device's `Boardconfig.mk` files.
    - Must reside in `vendor` partition at `/vendor/etc/selinux/nonplat_hwservice_contexts` and be loaded by `hwservicemanager` at the start along with the `plat_service_contexts`.

- `vndservice_contexts`
    - Device-specific `service_context` for the `vndservicemanager` built by combining `vndservice_contexts` found in the directories pointed to by `BOARD_SERPOLICY_DIRS` in the device's `Boardconfig.mk`.
    - This file must reside in `vendor` partition at `/vendor/etc/selinux/vndservice_contexts` and be loaded by `vndservicemanager` at the start.

Seapp contexts

In Android 8.0, the `seapp_contexts` is split between two files:

- `plat_seapp_contexts`
    - Android platform `seapp_context` that has no device-specific changes.
    - Must reside in `system` partition at `/system/etc/selinux/plat_seapp_contexts.`
- `nonplat_seapp_contexts`
    - Device-specific extension to platform `seapp_context` built by combining `seapp_contexts` found in the directories pointed to by `BOARD_SEPOLICY_DIRS` in the device's `Boardconfig.mk` files.
    - Must reside in `vendor` partition at `/vendor/etc/selinux/nonplat_seapp_contexts`.

MAC permissions

In Android 8.0, the `mac_permissions.xml` is split between two files:

- Platform `mac_permissions.xml`
    - Android platform `mac_permissions.xml` that has no device-specific changes.
    - Must reside in `system` partition at `/system/etc/selinux/.`

- Non-Platform `mac_permissions.xml`
  - Device-specific extension to platform `mac_permissions.xml` built from `mac_permissions.xml` found in the directories pointed to by `BOARD_SEPOLICY_DIRS` in the device's `Boardconfig.mk` files.
  - Must reside in `vendor` partition at `/vendor/etc/selinux/`.

Object ownership and labeling

The Android 8.0 goal of independent updates for platform and vendor components means that ownership must be clearly defined for each object. For example, if the vendor labels `/dev/foo` and the platform then labels `/dev/foo` in a subsequent OTA, there will be undefined behavior. For SELinux, this manifests as a labeling collision. The device node can have only a single label which resolves to whichever label is applied last. As a result:

- Processes that *need access* to the unsuccessfully applied label will lose access to the resource.
- Processes that *gain access* to the file may break because the wrong device node was created.

System properties also have potential for naming collisions that could result in undefined behavior on the system (as well as for SELinux labeling). Collisions between platform and vendor labels can occur for any object that has an SELinux label, including properties, services, processes, files, and sockets. To avoid these issues, clearly define ownership of these objects.

In addition to label collisions, SELinux type/attribute names may also collide. A type/attribute name collision will always result in a policy compiler error.

Type/attribute namespacing

SELinux does not allow multiple declarations of the same type/attribute. Policy with duplicate declarations will fail to compilation. To avoid type and attribute name collisions, all non-platform declarations should be namespace starting with `np_`.

```
type foo, domain; → type np_foo, domain;
```

System Property and process labeling ownership

Avoiding labeling collisions is best solved using property namespaces. Vendors should prefix their property names with `vendor`. Examples:

```
foo.xxx → vendor.foo.xxx
```

```
ro.foo.xxx  → ro.vendor.foo.xxx
persist.foo.xxx  → persist.vendor.foo.xxx
```

This naming convention is **recommended** in Android 8.0 and will be enforced in Android P.

File ownership

Preventing collisions for files is challenging because platform and vendor policy both commonly provide labels for all filesystems. Unlike type naming, namespacing of files is not practical since many of them are created by the kernel. Instead, we need to enumerate over the different filesystems and provide rules as to where the platform and vendor policy may provide labels. For Android 8.0, these are recommendations without technical enforcement. In the future, these recommendations will be enforced by the Vendor Test Suite (VTS).

System (/system)

Only the system image must provide labels for `/system` components through `file_contexts`, `service_contexts` etc. If labels for `/system` components are added in `/vendor` policy, a framework-only OTA update may not be possible.

Vendor (/vendor)

The AOSP SELinux policy already labels parts of `vendor` partition the platform interacts with, which enables writing SELinux rules for platform processes to be able to talk and/or access parts of `vendor` partition. Examples:

/vendor path	Platform-Provided Label	Platform Processes depending on the label
`/vendor(/.*)?`	`vendor_file`	All HAL clients in framework, ueventd, etc.
`/vendor/framework(/.*)?`	`vendor_framework_file`	dex2oat, appdomain, etc.
`/vendor/app(/.*)?`	`vendor_app_file`	dex2oat, installd, idmap, etc.
`/vendor/overlay(/.*)`	`vendor_overlay_file`	system_server, zygote, idmap, etc.

***** *Find more examples in `system/sepolicy/private/file_contexts`*

As a result, specific rules must be followed (enforced through `neverallows`) when labelling additional files in `vendor` partition:

- `vendor_file` must be the default label in for all files in `vendor` partition. The platform policy requires this to access passthrough HAL implementations.
- All new `exec_types` added in `vendor` partition through vendor SEPolicy must have

`vendor_file_type` attribute. This is enforced through neverallows.

- To avoid conflicts with future platform/framework updates, avoid labelling files other than `exec_types` in `vendor` partition.
- All library dependencies for AOSP-identified same process HALs must be labelled as `same_process_hal_file`.

Procfs (/proc)

Files in `/proc` may be labeled using only the `genfscon` label. In Android 7.0, both the platform and vendor policy used `genfscon` to label files in `procfs`.

Recommendation: Only platform policy labels `/proc`. If `vendor` processes need access to files in `/proc` that are currently labeled with the default label (`proc`), vendor policy MUST not explicitly label them and should instead use the generic `proc` type to add rules for vendor domains. This allows the platform updates to accommodate future kernel interfaces exposed through `procfs` and label them explicitly as needed.

Debugfs (/sys/kernel/debug)

`Debugfs` can be labeled in both `file_contexts` and `genfscon`. In Android 7.0, both platform and vendor label `debugfs`.

Recommendation: In the short term, only vendor policy may label `debugfs`. In the long term, remove `debugfs`.

Tracefs (/sys/kernel/debug/tracing)

`Tracefs` can be labeled in both `file_contexts` and `genfscon`. In Android 7.0, only the platform labels `tracefs`.

Recommendation: Only platform may label tracefs.

Sysfs (/sys)

Files in `/sys` may be labeled using both `file_contexts` and genfscon. In Android 7.0, both platform and vendor use `file_contexts` and `genfscon` to label files in `sysfs`.

Recommendation: The platform may label only the select files listed below:

```
/sys/class/leds/
/sys/devices/system/cpu(/.*)?
u:object_r:sysfs_devices_system_cpu:s0
/sys/devices/virtual/block/zram\d+(/.*)?
u:object_r:sysfs_zram:s0
/sys/devices/virtual/block/zram\d+/uevent
u:object_r:sysfs_zram_uevent:s0
```

```
/sys/devices/virtual/misc/hw_random(/.*)?
u:object_r:sysfs_hwrandom:s0
/sys/power/wake_lock -- u:object_r:sysfs_wake_lock:s0
/sys/power/wake_unlock -- u:object_r:sysfs_wake_lock:s0
/sys/kernel/uevent_helper --    u:object_r:usermodehelper:s0
/sys/module/lowmemorykiller(/.*)? --
u:object_r:sysfs_lowmemorykiller:s0
/sys/devices/virtual/timed_output/vibrator/enable
u:object_r:sysfs_vibrator:s0
```

Otherwise, only vendor may label files.

tmpfs (/dev)

Files in `/dev` may be labeled in `file_contexts`. In Android 7.0, both platform and vendor label files here.

Recommendation: Vendor may label only files in `/dev/vendor` (e.g., `/dev/vendor/foo`, `/dev/vendor/socket/bar`).

Rootfs (/)

Files in `/` may be labeled in `file_contexts`. In Android 7.0, both platform and vendor label files here.

Recommendation: Only system may label files in `/`.

Data (/data)

Data is labeled through a combination of `file_contexts` and `seapp_contexts`.

Recommendation: Disallow vendor labeling outside `/data/vendor`. Only platform may label other parts of `/data`.

SELinux policy building and customization

In Android 8.0, SELinux policy is split into **platform** and **non-platform** components to allow independent platform/non-platform policy updates while maintaining compatibility.

The platform sepolicy is further split into **platform private** and **platform public** parts to export specific types and attributes to non-platform policy writers. The platform public types/attributes are guaranteed to be maintained as stable APIs for a given platform version. Compatibility with previous platform public types/attributes can be guaranteed for several versions using platform mapping files.

Platform public sepolicy

The platform public sepolicy includes everything defined under system/sepolicy/public. The platform can assume the types and attributes defined under public policy are stable APIs for a given platform version. This forms the part of the sepolicy that is exported by platform on which non-platform (i.e. device) policy developers may write additional device-specific policy.

Types are versioned according to the version of the policy that non-platform files are written against, defined by the PLATFORM_SEPOLICY_VERSION build variable. The versioned public policy is then included with the non-platform policy and (in its original form) in the platform policy. Thus, the final policy includes the private platform policy, the current platform's public sepolicy, the device-specific policy, and the versioned public policy corresponding to the platform version against which the device policy was written.

Platform private sepolicy

The platform private sepolicy includes everything defined under system/sepolicy/private. This part of the policy forms platform-only types, permissions, and attributes required for platform functionality. These are **not exported** to the vendor/device policy writers. Non-platform policy writers must not write their policy extensions based on types/attributes/rules defined in platform private sepolicy. Moreover, these rules are allowed to be modified or may disappear as part of a framework-only update.

Platform private mapping

The platform private mapping includes policy statements that map the attributes exposed in platform public policy of the previous platform versions to the concrete types used in current platform public sepolicy. This ensures non-platform policy that was written based on platform public attributes from the previous platform public sepolicy version(s) continues to work. The versioning is based on the PLATFORM_SEPOLICY_VERSION build variable set in AOSP for a given platform version. A separate mapping file exists for each previous platform version from which this platform is expected to accept vendor policy.

Building SELinux policy

SELinux policy in Android 8.0 is made by combining pieces from `/system` and `/vendor`. Logic for setting this up appropriate is in `/platform/system/sepolicy/Android.mk`.

Policy exists in the following locations:

Location	Contains
`system/sepolicy/public`	Contains the platform's sepolicy API.
`system/sepolicy/private`	Contains platform implementation details (vendors can ignore)

`system/sepolicy/vendor`	Contains freebies for vendors to use (vendors can ignore if desired)
`BOARD_SEPOLICY_DIRS`	Contains vendor sepolicy.

The build system takes this policy and produces platform and non-platform policy components on the system partition and vendor partition, respectively. Steps include:

1. Converting policies to the SELinux CIL format, specifically:
    - public platform policy
    - combined private + public policy
    - public + vendor and BOARD_SEPOLICY_DIRS policy
2. Versioning the policy provided by public as part of the vendor policy. Done by using the produced public CIL policy to inform the combined public + vendor + BOARD_SEPOLICY_DIRS policy as to which parts must be turned into attributes that will be linked to the platform policy.
3. Creating a mapping file linking the platform and non-platform parts. Initially, this just links the types from the public policy with the corresponding attributes in the vendor policy; later it will also provide the basis for the file maintained in future platform versions, enabling compatibility with vendor policy targeting this platform version.
4. Combining policy files (describe both on-device and precompiled solutions).
    - Combine mapping, platform and non-platform policy.
    - Compile output binary policy file.

Policy compatibility

This section describes the design for handling the policy compatibility issues with platform OTAs, where new platform SELinux settings may differ from old vendor SELinux settings.

The design considers a binary distinction between *platform* and *non-platform*; the scheme becomes more complicated if non-platform partitions generate dependencies, such as `platform` < `vendor` < `oem`. While the Android 8.0 model handles separation between the SoC (`vendor`) and the ODM (`odm`) partitions, the working model for SE policy is that the SoC and ODM policy is a single piece.

Goals	Assumptions
• Align with and enable Android 8.0 architecture goals.	• Vendor update may never go up a version before the platform, i.e. the platform

• Ensure vendor code continues to work after platform update. • Ability to deprecate/change policy. • No required knowledge of specific version changes for policy development. • Enable and encourage vendor policy customization and specification.	version is the same or newer than that on vendor. • A set window of versions must be supported. • Vendor policy version is made available to determine the platform policy to be delivered; this is similar to determining Vendor Native Development Kit (VNDK) compatibility and needs.

SELinux global policy is divided into private and public components based on the Android 8.0 model. Public components consist of the policy and associated infrastructure, which are guaranteed to be available for a platform version. This policy will be exposed to vendor policy writers to enable vendors to build a vendor policy file, which when combined with the platform-provided policy, results in a fully-functional policy for a device.

- For versioning, the exported platform-public policy will be written as *attributes*.
- For ease of policy writing, exported types will be transformed into *versioned attributes* as part of the policy build process. Public types may also be used directly in labeling decisions provided by vendor contexts files.

A mapping must be maintained between exported concrete types in platform policy and the corresponding versioned attributes for each platform version. This ensures that when objects are labeled with a type, it does not break behavior guaranteed by the platform-public policy in a previous version. This mapping is maintained by keeping a mapping file up-to-date for each platform version, which keeps attribute membership information for each type exported in public policy.

Compatibility attributes

SELinux policy is an interaction between source and target types for specific object classes and permissions. Every object (processes, files, etc.) affected by SELinux policy may have only one type, but that type may have multiple attributes.

Policy is written mostly in terms of existing types:

```
allow source_type target_type:target_class permission(s);
```

This works because the policy was written with knowledge of all types. However, if the vendor policy and platform policy use specific types, and the label of a specific object changes in only one of those policies, the other may contain policy that gained or lost

access previously relied upon. For example:

```
File_contexts:
/sys/A   u:object_r:sysfs:s0
Platform: allow p_domain sysfs:class perm;
Vendor: allow v_domain sysfs:class perm;
```

Could be changed to:

```
File_contexts:
/sys/A   u:object_r:sysfs_A:s0
```

Although the vendor policy would remain the same, the `v_domain` would lose access due to the lack of policy for the new `sysfs_A` type.

By defining a policy in terms of attributes, we can give the underlying object a type that has an attribute corresponding to policy for both the platform and vendor code. This can be done for all types to effectively create an *attribute-policy* wherein concrete types are never used. In practice, this is required only for the portions of policy that overlap between platform and vendor, which are defined and provided as *platform public policy* that gets built as part of the vendor policy.

Defining public policy as versioned attributes satisfies two policy compatibility goals:

- **Ensure Vendor code continues to work after platform update**. Achieved by adding attributes to concrete types for objects corresponding to those on which vendor code relied, preserving access.
- **Ability to deprecate policy**. Achieved by clearly delineating policy sets into attributes that can be removed as soon as the version to which they correspond no longer is supported. Development can continue in the platform, knowing the old policy is still present in the vendor policy and will be automatically removed when/if it upgrades.

Policy writability

To meet the goal of not requiring knowledge of specific version changes for policy development, Android 8.0 includes a mapping between platform-public policy types and their attributes. We do this by mapping type `foo` to attribute `foo_vN`, where `N` is the version targeted. `vN` corresponds to the PLATFORM_SEPOLICY_VERSION build variable and is of the form `MM.nn`, where `MM` corresponds to the platform SDK number and `NN` is a platform sepolicy specific version.

Attributes in public policy are not versioned, but rather exist as an API on which platform and vendor policy can build to keep the interface between the two partitions stable. Both

platform and vendor policy writers can continue to write policy as it is written today.

Platform-public policy exported as `allow source_foo target_bar:class perm;` is included as part of the vendor policy. During compilation (which would include the corresponding version) it is transformed into the policy that will go to the vendor portion of the device (shown in the transformed CIL):

```
(allow source_foo_vN target_bar_vN (class (perm)))
```

As vendor policy is never ahead of the platform, it should not be concerned with prior versions. However, platform policy will need to know how far back vendor policy is, include attributes to its types, and set policy corresponding to versioned attributes.

Policy diffs

Automatically creating attributes by adding `_vN` to the end of each type does nothing without mapping of attributes to types across version diffs. We need to maintain a mapping between versions for attributes and a mapping of types to those attributes. This is done in the aforementioned mapping files with statements, such as (CIL):

```
(typeattributeset foo_vN (foo))
```

Platform upgrades

The following section details scenarios for platform upgrades.

Same types

This scenario occurs when an object does not change labels in policy versions. This is the same for source and target types and can be seen with `/dev/binder`, which is labeled `binder_device` across all releases. It is represented in transformed policy as:

```
binder_device_v1 … binder_device_vN
```

When upgrading from `v1` → `v2`, the platform policy must contain:

```
type binder_device; -> (type binder_device) (in CIL)
```

In the v1 mapping file (CIL):

```
(typeattributeset binder_device_v1 (binder_device))
```

In the v2 mapping file (CIL):

```
(typeattributeset binder_device_v2 (binder_device))
```

In the v1 vendor (non_plat) policy (CIL):

```
(typeattribute binder_device_v1)
(allow binder_device_v1 …)
```

In the v2 vendor (non_plat) policy (CIL):

```
(typeattribute binder_device_v2)
(allow binder_device_v2 …)
```

New types

This scenario occurs when the platform has added a new type, which can happen when adding new features or during policy hardening.

- **New feature**. When the type is labeling an object that was previously non-existent (such as a new service process), the vendor code did not previously interact with it directly so no corresponding policy exists. The new attribute corresponding to the type does not have an attribute in the previous version, and so would not need an entry in the mapping file targeting that version.
- **Policy hardening**. When the type represents policy hardening, the new type attribute must link back to a chain of attributes corresponding to the previous one (similar to the previous example changing `/sys/A` from `sysfs` to `sysfs_A`). Vendor code would rely on a rule enabling access to `sysfs`, and would need to include it as an attribute of the new type

  When upgrading from `v1` → `v2`, the platform policy must contain:

  ```
  type sysfs_A; -> (type sysfs_A) (in CIL)
  Type sysfs; (type sysfs) (in CIL)
  ```

  In the v1 mapping file (CIL):

  ```
  (typeattributeset sysfs_v1 (sysfs sysfs_A))
  ```

  In the v2 mapping file (CIL):

```
(typeattributeset sysfs_v2 (sysfs))
(typeattributeset sysfs_A_v2 (sysfs_A))
```

In the v1 vendor (non_plat) policy (CIL):

```
(typeattribute sysfs_v1)
(allow … sysfs_v1 …)
```

In the v2 vendor (non_plat) policy (CIL):

```
(typeattribute sysfs_A_v2)
(allow … sysfs_A_v2 …)
(typeattribute sysfs_v2)
(allow … sysfs_v2 …)
```

Removed types

This (rare) scenario occurs when a type is removed, which can happen when the underlying object:

- Remains but gets a different label.
- Is removed by the platform.

During policy loosening, a type is removed and the object labeled with that type is given a different, already-existing label. This represents a merging of attribute mappings: The vendor code must still be able to access the underlying object by the attribute it used to posses, but the rest of the system must now be able to access it with its new attribute.

If the attribute to which it has been switched is new, then relabeling is the same as in the new type case, except that when an existing label is used, the addition of the old attribute new type would cause other objects also labeled with this type to be newly accessible. This is essentially what is done by the platform, though, and is deemed to be an acceptable tradeoff to maintain compatibility.

```
(typeattribute sysfs_v1)
(allow … sysfs_v1 …)
```

Example Version 1: Collapsing types (removing sysfs_A)

When upgrading from `v1` → `v2`, the platform policy must contain:

```
type sysfs; (type sysfs) (in CIL)
```

In the v1 mapping file (CIL):

```
(typeattributeset sysfs_v1 (sysfs))
(type sysfs_A) # in case vendors used the sysfs_A label on objects
(typeattributeset sysfs_A_v1 (sysfs sysfs_A))
```

In the v2 mapping file (CIL):

```
(typeattributeset sysfs_v2 (sysfs))
```

In the v1 vendor (non_plat) policy (CIL):

```
(typeattribute sysfs_A_v1)
(allow … sysfs_A_v1 …)
(typeattribute sysfs_v1)
(allow … sysfs_v1 …)
```

In the v2 vendor (non_plat) policy (CIL):

```
(typeattribute sysfs_v2)
(allow … sysfs_v2 …)
```

Example Version 2: Removing completely (foo type)

When upgrading from `v1` → `v2`, the platform policy must contain:

```
# nothing - we got rid of the type
```

In the v1 mapping file (CIL):

```
(type foo) #needed in case vendors used the foo label on objects
(typeattributeset foo_v1 (foo))
```

In the v2 mapping file (CIL):

```
# nothing - get rid of it
```

In the v1 vendor (non_plat) policy (CIL):

```
(typeattribute foo_v1)
(allow foo …)
(typeattribute sysfs_v1)
(allow sysfs_v1 …)
```

In the v2 vendor (non_plat) policy (CIL):

```
(typeattribute sysfs_v2)
(allow sysfs_v2 …)
```

New class/permissions

This scenario occurs when a platform upgrade introduces new policy components that do not exist in previous versions. For example, when we added the `servicemanager` object manager that created the add, find, and list permissions, vendor daemons wanting to register with the `servicemanager` would have needed permissions that were not available. In Android 8.0, only the platform policy may add new classes and permissions.

To allow all domains that could have been created or extended by vendor policy to use the new class without obstruction, the platform policy needs to include a rule similar to:

```
allow {domain -coredomain} *:new_class perm;
```

This may even require policy allowing access for all interface (public policy) types, to be sure vendor image gains access. If this results in unacceptable security policy (as it may have with the servicemanager changes), a vendor upgrade could potentially be forced.

Removed class/permissions

This scenario occurs when an object manager is removed (such as the `ZygoteConnection` object manager) and should not cause issues. The object manager class and permissions could remain defined in policy until the vendor version no longer uses it. This would be done by adding the definitions to the corresponding mapping file.

Vendor customization for new/relabeled types

New vendor types are at the core of vendor policy development as they are needed to describe new processes, binaries, devices, subsystems, and stored data. As such, it is imperative to allow the creation of vendor-defined types.

As vendor policy is always the oldest on the device, there is no need to automatically convert all vendor types to attributes in policy. The platform does not rely on anything labeled in vendor policy because the platform has no knowledge of it; however, the platform will provide the attributes and public types it uses to interact with objects labeled with these types (such as domain, `sysfs_type`, etc.). For the platform to continue to interact correctly with these objects, the attributes and types must be appropriately applied and specific rules may need to be added to the customizable domains (such as `init`).

Platform-public policy

The platform-public policy is the heart of conforming to the Android 8.0 architecture model without simply maintaining the union of platform policies from v1 and v2. Vendors will be exposed to a subset of platform policy that contains useable types and attributes and rules on those types and attributes which then becomes part of vendor policy (i.e. `nonplat_sepolicy.cil`).

Types and rules will be automatically translated in the vendor-generated policy into `attribute_vN` such that all platform-provided types are versioned attributes (however attributes are not versioned). The platform is responsible for mapping the concrete types it provides into the appropriate attributes to ensure that vendor policy continues to function and that the rules provided for a particular version are included. The combination of platform-public policy and vendor policy should satisfy the Android 8.0 architecture model goal of allowing independent platform and vendor builds.

Mapping to attribute chains

When using attributes to map to policy versions, a type maps to an attribute or multiple attributes, ensuring objects labeled with the type are accessible via attributes corresponding to their previous types.

Maintaining a goal to hide version information from the policy writer means automatically generating the versioned attributes and assigning them to the appropriate types. In the common case of static types, this is straightforward: `type_foo_v2` maps to `type_foo_v1`.

For an object label change such as `sysfs` → `sysfs_A` or `mediaserver` → `audioserver`, creating this mapping is non-trivial (and is described in the examples above). Platform policy maintainer must determine how to create the mapping at transition points for objects, which requires understanding the relationship between objects and their assigned labels and determining when this occurs. For backwards compatibility, this complexity needs to be managed on the platform side, which is the only partition that may uprev.

Version uprevs

For simplicity, we uprev the platform sepolicy version when a new release branch is cut. As described above, the version number is contained in `PLATFORM_SEPOLICY_VERSION` and

is of the form `MM.nn`, where `MM` corresponds to the SDK value and `nn` is a private value maintained in `/platform/system/sepolicy`. For example, `19.0` for Kitkat, `21.0` for Lollipop, `22.0` for Lollipop-MR1 `23.0` for Marshmallow, `24.0` for Nougat, `25.0` for Nougat-MR1, and `26.0` for O. An MR bump to O necessitating an incompatible change in `system/sepolicy/public` but not an API bump could then result in releasing a new version: `26.1`. The version present in a development branch is a never-to-be-used-in-shipping-devices `10000.0`.

We may deprecate oldest version when upreving. For example, in Marshmallow, we may support LMP-MR1 (`v23`), LMP (`v22`), and KK (`19.0`). When Nougat (`25.0`) is released, we may drop KK (`19.0`). For input on when to deprecate a version X, we may collect the number of devices with vendor policies running on X which can still receive major platform update without vendor update. If the number is less than a certain threshold, then we deprecate X.

Performance impact of multiple attributes

As described in: https://github.com/SELinuxProject/cil/issues/9 a large number of attributes assigned to a type result in performance issues in the event of a policy cache (avc) miss.

This was confirmed to be an issue in Android, so changes were made to Android 8.0 to remove attributes added to the policy by the policy compiler, as well as to remove unused attributes. These changes resolved performance regressions.

Customizing SEPolicy

The split of SELinux policy and automatic versioning of public types allows policy writers to write policy for their customizations to Android in the framework as well as the vendor implementation. However, to not break the Design goals, the design has limitations and imposes rules to be followed when customizing policy for an Android device.

This section provides guidelines for partner SELinux policy in Android 8.0, including details on Android Open Source Project (AOSP) SEPolicy and SEPolicy extensions.

Policy placement

In Android 7.0, partners could add policy to BOARD_SEPOLICY_DIRS, including policy meant to augment AOSP policy across different device types. In Android 8.0, adding a policy to BOARD_SEPOLICY_DIRS places the policy only in the vendor image.

In Android 8.0 AOSP, policy exists in the following locations:

- **system/sepolicy/public**. Includes policy exported for use in vendor-specific policy. Everything goes into the Android 8.0 compatibility infrastructure, and vendors include it as part of their policy so it can be relied upon (leading to restrictions on the type of policy that can be placed here). Consider this the platform's exported policy API:

Anything that deals with the interface between `/system` and `/vendor` should be here.

- **system/sepolicy/private**. Includes policy necessary for the functioning of the system image, but of which vendor image policy should have no knowledge.
- **system/sepolicy/vendor**. Includes policy for components that go in /vendor but exist in the core platform tree (not device-specific directories). This is an artifact of build system's distinction between devices and global components; conceptually this is a part of the device-specific policy described below.
- **device/XXX/YYY/sepolicy**. Includes device-specific policy. Also includes device customizations to policy, which in Android 8.0 now corresponds to policy for components on the vendor image.

Supported policy scenarios

On devices launching with Android 8.0, the vendor image must work with the OEM system image and the reference AOSP system image provided by Google (and pass CTS on this reference image). These requirements ensure a clean separation between the framework and the vendor code. Such devices support the following scenarios.

vendor-image-only extensions

Example: Adding a new service to `vndservicemanager` from the vendor image that supports processes from the vendor image.

As with devices launching with previous Android versions, add device-specific customization in `device/XXX/YYY/sepolicy`. New policy governing how vendor components interact with (only) other vendor components **should involve types present only in `device/XXX/YYY/sepolicy`**. Policy written here allows code on vendor to work, will not be updated as part of a framework-only OTA, and will be present in the combined policy on a device with the reference AOSP system image.

vendor-image support to work with AOSP

Example: Adding a new process (registered with `hwservicemanager` from the vendor image) that implements an AOSP-defined HAL.

As with devices launching with previous Android versions, perform device-specific customization in `device/XXX/YYY/sepolicy`. The policy exported as part of `system/sepolicy/public/` is available for use, and is shipped as part of the vendor policy. Types and attributes from the public policy may be used in new rules dictating interactions with the new vendor-specific bits, subject to the provided `neverallow` restrictions. As with the vendor-only case, new policy here will not be updated as part of a framework-only OTA and will be present in the combined policy on a device with the reference AOSP system image.

system-image-only extensions

Example: Adding a new service (registered with servicemanager) that is accessed only by other processes from the system image.

Add this policy to `system/sepolicy/private`. You can add extra processes or objects to enable functionality in a partner system image, provided those new bits don't need to interact with new components on the vendor image (specifically, such processes or objects must fully function without policy from the vendor image). The policy exported by `system/sepolicy/public` is available here just as it is for vendor-image-only extensions. This policy is part of the system image and could be updated in a framework-only OTA, but will not be present when using the reference AOSP system image.

vendor-image extensions that serve extended AOSP components

Example: A new, non-AOSP HAL for use by extended clients that also exist in the AOSP system image (such as an extended system_server).

Policy for interaction between system and vendor must be included in the `device/XXX/YYY/sepolicy` directory shipped on the vendor partition. This is similar to the above scenario of adding vendor-image support to work with the reference AOSP image, except the modified AOSP components may also require additional policy to properly operate with the rest of the system partition (which is fine as long as they still have the public AOSP type labels).

Policy for interaction of public AOSP components with system-image-only extensions should be in `system/sepolicy/private`. This has the following consequences for the following system - vendor image combinations:

System	Vendor	Notes
O Partner	O	Combined policy has knowledge of both parts.
O AOSP	O	Combined policy includes new types and policy from vendor image, but as AOSP does not have its components extended to use the new HAL, it isn't needed. The lack of extension also means the partner policy in `system/sepolicy/private` is unnecessary.
P Partner	O	You may need to make changes to: • *System components that used AOSP types*. For example, if the extended `system_server` were split into multiple processes, the new processes could have new types provided they are properly linked to the old types using the compatibility infrastructure (provided by AOSP). • *System-only components*. Includes adding policy for how such components interact with components labeled with AOSP public types (done in `system/sepolicy/private`).

P AOSP	O	Similar to O AOSP system image - O vendor image.

system-image extensions that access only AOSP interfaces

Example: A new, non-AOSP system process must access a HAL on which AOSP relies.

This is similar to the system-image-only extension example, except new system components may interact across the `system/vendor` interface. Policy for the new system component must go in `system/sepolicy/private`, which is acceptable provided it is through an interface already established by AOSP in `system/sepolicy/public` (i.e. the types and attributes required for functionality are there). While policy could be included in the device-specific policy, it would be unable to use other `system/sepolicy/private` types or change (in any policy-affecting way) as a result of a framework-only update. The policy may be changed in a framework-only OTA, but will not be present when using an AOSP system image (which won't have the new system component either).

vendor-image extensions that serve new system components

Example: Adding a new, non-AOSP HAL for use by a client process without an AOSP analogue (and thus requires its own domain).

Similar to the AOSP-extensions example, policy for interactions between system and vendor must go in the `device/XXX/YYY/sepolicy` directory shipped on the vendor partition (to ensure the system policy has no knowledge of vendor-specific details). You can add new public types that extend the policy in `system/sepolicy/public`; this should be done only in addition to the existing AOSP policy, i.e. do not remove AOSP public policy. The new public types can then be used for policy in `system/sepolicy/private` and in `device/XXX/YYY/sepolicy`.

Keep in mind that every addition to `system/sepolicy/public` adds complexity by exposing a new compatibility guarantee that must be tracked in a mapping file and which is subject to other restrictions. Only new types and corresponding allow rules may be added in `system/sepolicy/public`; attributes and other policy statements are not supported. In addition, new public types cannot be used to directly label objects in the `/vendor` policy.

Unsupported policy scenarios

Devices launching with Android 8.0 do not support the following policy scenario and examples.

Additional extensions to system-image that need permission to new vendor-image components after a framework-only OTA

Example: A new non-AOSP system process, requiring its own domain, is added in Android

P and needs access to a new, non-AOSP HAL.

Similar to new (non-AOSP) system and vendor components interaction, except the new system type is introduced in a framework-only OTA. Although the new type could be added to the policy in `system/sepolicy/public`, the existing vendor policy has no knowledge of the new type as it is tracking only the Android 8.0 system public policy. AOSP handles this by exposing vendor-provided resources via an attribute (e.g. `hal_foo` attribute) but as attribute partner extensions are not supported in `system/sepolicy/public`, this method is unavailable to vendor policy. Access must be provided by a previously-existing public type.

Example: A change to a system process (AOSP or non-AOSP) must change how it interacts with new, non-AOSP vendor component.

The policy on the system image must be written without knowledge of specific vendor customizations. Policy concerning specific interfaces in AOSP is thus exposed via attributes in system/sepolicy/public so that vendor policy can opt-in to future system policy which uses these attributes. However, **attribute extensions in `system/sepolicy/public` are not supported**, so all policy dictating how the system components interact with new vendor components (and which is not handled by attributes already present in AOSP `system/sepolicy/public`) must be in `device/XXX/YYY/sepolicy`. This means that system types cannot change the access allowed to vendor types as part of a framework-only OTA.