SELinux in AOSP

1. Selinux in aosp

SELinux (Security-Enhanced Linux) in **AOSP (Android Open Source Project)** is a critical component of Android's security model. It enforces **Mandatory Access Control (MAC)** policies to confine privileged processes and apps to their least privilege.

Here's a breakdown of **SELinux in AOSP**:

1. SELinux Modes in Android

- **Permissive:** SELinux logs policy violations but doesn't block them.
- **Enforcing:** SELinux blocks and logs policy violations.

By default, Android runs in **enforcing mode** since Android 5.0 (Lollipop).

2. SELinux Policy in AOSP

Android uses its own set of policies located mainly in:

```bash
system/sepolicy/
```

With Android 8+, it's split into:

- **Platform policy:** AOSP-defined in `system/sepolicy/`
- **Vendor policy:** Defined by device manufacturers in `device/<vendor>/sepolicy/` or `vendor/<...>`

3. Key Directories

- `system/sepolicy/public/`: Policy headers that can be included by vendor policies.
- `system/sepolicy/private/`: Platform-specific policies not available to vendors.
- `system/sepolicy/prebuilts/`: Prebuilt policies, sometimes used during build.

4. Adding New SELinux Rules (Example)

Suppose you add a new HAL or daemon:

1. **Define a new type:**

   ```c
   type my_daemon, domain;
   ```

2. **Set domain transition:**

   ```c
   init_daemon_domain(my_daemon)
   ```

3. **Create a `my_daemon.te` file** in the appropriate policy directory.
4. **Add it to the `file_contexts`:**

   ```bash
   /vendor/bin/my_daemon  u:object_r:my_daemon_exec:s0
   ```

5. Compile SELinux Policy

When you build AOSP with:

```go
make -jX
```

The SELinux policies are compiled into **`sepolicy`** binary blobs and included in the system/vendor images.

---

6. Debugging SELinux Issues

- Use **`dmesg | grep avc`** or **`logcat`** to check denials.
- Use **`audit2allow`** tool to generate suggested policies.
- **`setenforce 0`** to switch to permissive temporarily for debugging.

Would you like a practical walkthrough on how to add a new SELinux rule for a custom service in AOSP?

2. Yes

Great! Here's a **practical walkthrough** for adding SELinux rules for a **custom service/daemon** in AOSP. Let's call the daemon **`mydaemon`** and assume it runs from **`/vendor/bin/mydaemon`**.

Step-by-Step: Add SELinux Policy for Custom Daemon in AOSP

---

1. Add the Binary

Place your binary at:

```php-template
vendor/<your_vendor>/<your_device>/mydaemon/
```

Update **`Android.mk`** or **`Android.bp`** to install it at:

```bash
/vendor/bin/mydaemon
```

---

2. Add File Contexts

In your device-specific policy directory:

File: `vendor/<your_vendor>/<your_device>/sepolicy/file_contexts`

```bash
/vendor/bin/mydaemon     u:object_r:mydaemon_exec:s0
```

---

3. Create `mydaemon.te` Type Enforcement File

File: `vendor/<your_vendor>/<your_device>/sepolicy/mydaemon.te`

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

init_daemon_domain(mydaemon)
```

---

4. Define Domain Transition

File: `vendor/<your_vendor>/<your_device>/sepolicy/mydaemon.te` (continued)

```c
# Allow mydaemon to read system files or sockets if needed
allow mydaemon self:process { fork execmem };
allow mydaemon mydaemon_exec:file { read execute open getattr };

# Example: allow access to /data/mydaemon/
allow mydaemon mydaemon_data_file:dir { search open read };
allow mydaemon mydaemon_data_file:file { read write open create };
```

---

5. Add `mydaemon` to `init.rc`

File: `vendor/<your_vendor>/<your_device>/init/init.your_device.rc`

```rc
service mydaemon /vendor/bin/mydaemon
    class main
    user system
    group system
    seclabel u:r:mydaemon:s0
    oneshot
```

---

6. Compile and Flash

Rebuild your AOSP:

```bash
source build/envsetup.sh
lunch aosp_<device>-userdebug
make -jX
```

Flash your images or boot into the system image.

---

7. Debugging Denials

Boot the device and use:

```bash
adb logcat | grep avc
```

You may see something like:

```bash
avc: denied { read } for pid=1234 comm="mydaemon" name="config" dev="dm-0" ...
```

Generate temporary policies (for debugging only):

```bash
adb pull /sys/fs/selinux/policy
audit2allow -p policy < log.txt > mydaemon_fix.te
```

Apply fixes gradually and test. Never allow overly permissive rules like **`allow mydaemon *`**.

Let me know if you want a ready-made policy snippet or a minimal working example you can drop into your AOSP tree.

3. Why it is required

Good question — here's **why SELinux policies are required** in AOSP, especially for custom daemons:

Why SELinux Policy is Required for Custom Services in AOSP

1. Android Enforces Mandatory Access Control (MAC)

Since Android 5.0, **SELinux runs in enforcing mode by default**, meaning:

- All access between processes, files, and system components must be explicitly allowed by policy.
- Anything not allowed is **blocked and logged** (via `avc: denied` messages).

If your custom daemon isn't declared in SELinux, it will get blocked when it tries to:

- Start via `init`
- Access any files, sockets, services, etc.

---

2. Every Process Needs a Domain

Android SELinux separates all apps and system services into **domains** like:

- `system_server`
- `hal_camera_default`
- `netd`
- `untrusted_app`
- etc.

Your custom service needs its own domain like `mydaemon`, otherwise it might:

- Run in `init` domain (bad security)
- Be denied access to files or sockets
- Be killed silently by the kernel

---

3. Secure by Design

SELinux limits the **attack surface**:

- If your daemon gets compromised, it **can't affect other services**.
- You define **exactly what it's allowed to do** (principle of least privilege).

---

4. GMS Certification Requirement

If you're shipping a device with **Google Mobile Services (GMS)**:

- Google enforces strict SELinux compliance.
- All services must have proper policy.
- `setenforce 0` (permissive mode) is not allowed in production.

---

5. Init Won't Start Unlabeled Services

If you add a service to `init.rc` without a valid `seclabel`, or the label isn't mapped in policy:

- **Init will refuse to start it.**

- You'll see logs like: `init: couldn't set SELinux label ...`

Summary

SELinux in Android is not optional. It's:

- **Mandatory** for system integrity
- **Required** for service startup
- **Essential** for security and certification

Let me know if you want a diagram or visual explanation of how SELinux enforces boundaries in Android!

4. How it works with android and it's flow

Here's a **complete flow of how SELinux works in Android** and how it integrates into the system — from **boot to runtime access control**:

SELinux in Android – Architecture & Flow

1. Boot Time Initialization

- The kernel loads and initializes **SELinux** early in the boot process.
- Kernel loads the **SELinux policy binary** from `/vendor/etc/selinux` and `/system/etc/selinux`.

2. `init` Process with SELinux

- `init` is the **first userspace process**, and it's SELinux-aware.
- It reads service definitions from `.rc` files (e.g. `init.rc`, `init.device.rc`, etc.).
- Each service is **assigned a** `seclabel` (i.e., an SELinux domain).

Example:

```rc
service mydaemon /vendor/bin/mydaemon
    seclabel u:r:mydaemon:s0
```

- `init` applies this SELinux label using `setexeccon()` before forking the process.

---

3. Process Launch and Labeling

- When `mydaemon` starts, it runs in the `mydaemon` **domain**.
- Its binary (`/vendor/bin/mydaemon`) must have a matching **file context** label:

  ```bash
  /vendor/bin/mydaemon  u:object_r:mydaemon_exec:s0
  ```

- If labels mismatch or are missing:
  - `init` **refuses to start** the process.
  - Or it runs in an **incorrect domain** and gets denied access to resources.

---

4. Access Control at Runtime

Once running, SELinux enforces access at runtime:

Example Flow:

`mydaemon` wants to read `/data/mydaemon/config.json`

1. **Subject:** `mydaemon` (process domain)
2. **Object:** `/data/mydaemon/config.json` (file type like `mydaemon_data_file`)
3. SELinux checks:
   - Is there a rule allowing `mydaemon` domain to `read` `mydaemon_data_file`?
   - If yes: Access granted
   - If no: Access **denied and logged**

Logged as:

```bash
avc: denied { read } for pid=1234 comm="mydaemon" name="config.json"
```

5. Logging & Debugging

SELinux logs go to **`logcat`** and **`dmesg`**:

```bash
adb logcat | grep avc
dmesg | grep avc
```

Use **`audit2allow`** to suggest rules (for debugging only):

```bash
audit2allow -p /sys/fs/selinux/policy < log.txt
```

Diagram of SELinux Flow in Android

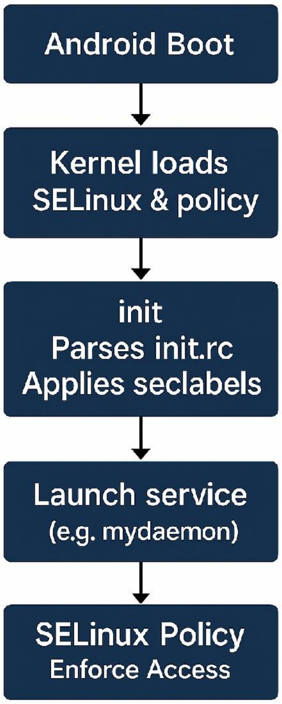
```text
+-------------------+
|    Android Boot   |
+--------+----------+
         |
         v
+--------+----------+
|    Kernel loads    |
| SELinux & policy   |
+--------+----------+
         |
         v
+--------+----------+
|       init        |
| Parses init.rc    |
| Applies seclabels |
+--------+----------+
         |
         v
+--------+----------+
| Launch service    |
| (e.g. mydaemon)   |
+--------+----------+
         |
         v
+--------+----------+
|  SELinux Policy   |
|  Enforce Access   |
|  (read/write etc) |
+-------------------+
```

Key SELinux Components in Android

Component	Role
`*.te` files	Define types and access rules
`file_contexts`	Maps file paths to SELinux labels
`sepolicy`	Compiled binary policy loaded at boot
`init.rc`	Starts services and assigns **`seclabel`**
`auditd`	Logs denials (built into kernel logging)

Let me know if you want this as a visual image or want to walk through adding rules for accessing specific Android subsystems (e.g., camera, network, sensors).

5. Yes



Android Boot
Kernel loads
SELinux & policy
init
Parses init.rc
Applies seclabels
Launch service
(e.g. mydaemon)
SELinux Policy
Enforce Access

6. You are making everything complex can you make it simple

Absolutely — let’s simplify it **a lot**.

What is SELinux in Android?

SELinux is like a **security guard** in Android.

Why is it needed?

It makes sure:

- **Apps and services** don’t do anything they’re **not allowed to do**.
- Even if a program gets hacked, it **can’t access** important system stuff.

How it works (Simple Flow)

1. **Android boots**
2. **SELinux loads security rules**
3. **Every app or service is given a label**
    - Like a badge: "This is mydaemon"
4. When the app tries to do something:
    - SELinux checks: **Is this allowed by the rules?**
    - If **yes** → Allow
    - If **no** → Block and log it

Example:

Your custom service **`mydaemon`** wants to read a file.

- If the SELinux rules **say it’s allowed**, it works.
- If not → You’ll see **“avc: denied”** in logs and it fails.

Why you need to write rules:

When you add a new service in AOSP:

- Android doesn’t know it
- So you must **tell SELinux what it's allowed to do**
    - Where it can read/write
    - What it can access
    - Who it can talk to

Want a super minimal real example of just the policy files you need for a custom service?

7. Yes

Perfect — here's the **most minimal working SELinux policy setup** for a custom service in AOSP, called **`mydaemon`**.

---

1. File Contexts

Tells SELinux how to label the binary.

`file_contexts`

```bash
/vendor/bin/mydaemon  u:object_r:mydaemon_exec:s0
```

---

2. Type Enforcement (TE) File

Defines the domain and basic permissions.

`mydaemon.te`

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

init_daemon_domain(mydaemon)

allow mydaemon self:process { fork };
allow mydaemon mydaemon_exec:file { read execute open };
```

> Add more rules only when needed.

---

3. Init.rc Entry

Starts your service and applies the SELinux label.

`init.yourdevice.rc`

```rc
service mydaemon /vendor/bin/mydaemon
    seclabel u:r:mydaemon:s0
    class main
    user system
    oneshot
```

---

4. Build System

Make sure your binary gets installed in **`/vendor/bin`** and your policy files are added to the device-specific **`sepolicy`** folder.

That's it!

When you boot the device:

- **`init`** starts **`mydaemon`**
- SELinux allows it to run with the right permissions
- If it tries something outside the rules, it's denied

Let me know if you want this in a zip, GitHub repo, or plugged into a real AOSP example.

8. File Contexts

Alright! Let's focus **just on `file_contexts`** — simple and to the point.

🔹 What is `file_contexts`?

It's a text file that tells SELinux:

> "When you see a file at this path, give it this security label."

This is how Android knows what domain or policy should apply to each file.

🔹 Syntax

```bash
<file path pattern>    <SELinux label>
```

Example:

```bash
/vendor/bin/mydaemon    u:object_r:mydaemon_exec:s0
```

🔍 Breaking it down:

- **`/vendor/bin/mydaemon`**: path to your binary
- **`u`**: SELinux user (always **`u`** on Android)
- **`object_r`**: Role (always **`object_r`** for files)
- **`mydaemon_exec`**: The **type** assigned to the file (defined in **`mydaemon.te`**)
- **`s0`**: Security level (always **`s0`** for most Android processes)

🔹 Where do I put `file_contexts`?

📁 Location (for device-specific services):

```php-template
device/<your_vendor>/<your_device>/sepolicy/file_contexts
```

Or if you're working with **Treble-enabled devices**, you might see:

```php-template
device/<your_vendor>/<your_device>/sepolicy/vendor/file_contexts
```

✅ What happens if it's missing?

If you don't add the right line to **`file_contexts`**, Android will:

- Not give the binary the correct label
- Deny the service from running or accessing files
- Show logs like:

```lua
avc: denied { execute } for path="/vendor/bin/mydaemon" ...
```

Let me know if you want a few real examples from AOSP's own **`file_contexts`**.

9. in more simpler

Of course! Let's make **`file_contexts`** super simple:

💡 What is `file_contexts`?

It's a file that tells Android:

> **"What label (permission group) should this file have?"**

📄 What does a line in `file_contexts` look like?

```bash
/vendor/bin/mydaemon    u:object_r:mydaemon_exec:s0
```

This means:

- File path: **`/vendor/bin/mydaemon`**
- Label it as: **`mydaemon_exec`**
  (so SELinux knows what rules to apply)

🛠️ Why is this needed?

When Android runs your service, it checks:

- "What label does this file have?"
- "What label should this service run with?"

If labels are missing or wrong — **it won't run**, or it will be **blocked**.

✅ What YOU need to do:

Just **add one line** like this in the file:

```bash
/vendor/bin/mydaemon    u:object_r:mydaemon_exec:s0
```

Put it in:

```php-template
device/<your_vendor>/<your_device>/sepolicy/file_contexts
```

That's it! Now Android knows how to treat your binary.

Want me to explain **`u:object_r:mydaemon_exec:s0`** even simpler too?

10. yes

Awesome — let's break this down as simply as possible:

🔹 `u:object_r:mydaemon_exec:s0` — What does it mean?

This is the **SELinux label**. Think of it like a full name with 3 parts:

```css
[user]   [role]       [type]           [level]
  u    : object_r : mydaemon_exec : s0
```

Now let's explain each one 👇

✅ 1. `u` → User

- Always `u` on Android.
- You can just think of it as: "required, ignore it."

✅ 2. `object_r` → Role

- Always `object_r` for files (like binaries, folders, etc.)
- Again, just use it — don't worry too much.

✅ 3. `mydaemon_exec` → Type

- **This is the most important part!**
- It tells SELinux what **rules** to use.
- You define this in `mydaemon.te`

💬 Think of it like:

> "If a file has `mydaemon_exec`, only services in the `mydaemon` domain can run it."

✅ 4. `s0` → Security level

- Always `s0` for Android.
- It's related to multilevel security (MLS), but Android doesn't really use it — so just keep it as `s0`.

✅ So... All together:

```bash
/vendor/bin/mydaemon   u:object_r:mydaemon_exec:s0
```

🔊 Means:

> "This file is a program that should follow `mydaemon_exec` rules. Only allowed processes can run it."

Let me know if you want a visual version or you want to test this on your AOSP build!

11. Type Enforcement (TE) File

Perfect — let's now simplify the **Type Enforcement (TE) file**, step-by-step, with only what you need to get started.

💡 What is a `.te` file?

It tells SELinux:

> "Here is a **new service/domain** and what it's allowed to do."

Basically:

- You name your service
- You give it permission to run
- You allow access to things (like files, sockets, etc.)

✅ Minimal Example: `mydaemon.te`

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

init_daemon_domain(mydaemon)
```

🧠 Explanation:

Line	What it does
`type mydaemon, domain;`	Defines a new SELinux domain (your service runs in this "box")
`type mydaemon_exec, exec_type, file_type;`	Says your binary (**`/vendor/bin/mydaemon`**) is executable
`init_daemon_domain(mydaemon)`	Lets **`init`** start your service safely and assigns it the correct label

📦 Where to put it?

Place this file in:

```php-template
device/<your_vendor>/<your_device>/sepolicy/mydaemon.te
```

Or in:

```bash
sepolicy/vendor/mydaemon.te
```

(Depending on where your device puts SELinux policy files.)

💬 That's enough to boot and run your service.

Once it's running, if it tries to:

- Read/write files
- Access sockets
- Talk to other services

And SELinux blocks it, you'll see errors like:

```bash
avc: denied { read } for pid=1234 comm="mydaemon" ...
```

Then you **add more `allow` rules** like:

```c
allow mydaemon some_file_type:file { read write };
```

But **only add what you need** when it's blocked. Start simple.

Let me know if you want an example of a `.te` file with `allow` rules too!

12. Yes please

Great! Here's a **simple working `.te` file** for your service **`mydaemon`**, with just enough **`allow`** rules to make it functional.

✅ Full Example: `mydaemon.te`

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

# Tell init it's a daemon and can be launched
init_daemon_domain(mydaemon)

# Allow running the binary
allow mydaemon mydaemon_exec:file { read execute open };

# Allow reading config files (example)
type mydaemon_config_file, file_type;
allow mydaemon mydaemon_config_file:file { read open getattr };

# Allow logging (log to logcat)
allow mydaemon logd:unix_dgram_socket sendto;

# Allow basic system operations
allow mydaemon self:process { fork execmem };
allow mydaemon self:capability { setuid setgid chown fowner };

# Allow access to /dev/null, /dev/urandom etc.
allow mydaemon device:chr_file { read write open };
```

🗂️ What else you'll need:

In `file_contexts`:

```bash
/vendor/bin/mydaemon         u:object_r:mydaemon_exec:s0
/data/mydaemon/config.json    u:object_r:mydaemon_config_file:s0
```

✅ Summary:

This **`.te`** file:

- Defines your service
- Lets it start
- Allows reading config
- Enables basic logging and process stuff

Need extras?

Let me know if your service:

- Talks to a HAL
- Uses network
- Accesses sensors or camera
  I'll add specific rules only for those.

Want me to bundle this into a working AOSP example or template?

13. How it talk to other process if this process in system and other in vendor

Great question! Let's break this down simply:

How can a `system` process talk to a `vendor` process with SELinux?

By default, **SELinux does *not* allow cross-boundary access** (system ↔ vendor), unless you **explicitly add allow rules** and label things properly.

✅ First: Understand domains

- Your **`mydaemon`** might run in **`u:r:mydaemon:s0`** (vendor domain)
- Another process, like a system app or service, might run in **`system_server`**, **`platform_app`**, etc.

✅ Second: How do they talk?

They usually communicate using:

- **Binder** (for Android services)
- **Sockets / pipes**
- **Shared files**

✅ Third: What SELinux checks

When **`system_server`** talks to **`mydaemon`**:

SELinux checks:

```text
allow system_server mydaemon: binder call;
```

or

```text
allow system_server mydaemon_socket:sock_file write;
```

✅ How to allow it

Example: allow `system_server` to talk to `mydaemon` via Binder

In **`mydaemon.te`** or a separate **`.te`** file:

```c
binder_use(mydaemon)  # Needed if using Binder
```

In **`system_server.te`** (or your interface policy):

```c
allow system_server mydaemon:binder call;
```

Example: Socket-based communication

Let's say **`mydaemon`** creates a socket **`/dev/socket/mydaemon`**.

You must:

1. Label the socket in **`file_contexts`**:

```bash
/dev/socket/mydaemon  u:object_r:mydaemon_socket:s0
```

2. Define the socket type:

```c
type mydaemon_socket, file_type, socket_type;
```

3. Allow access:

```c
allow system_server mydaemon_socket:sock_file write;
allow system_server mydaemon:unix_stream_socket connectto;
```

🧠 Bonus: Use `neverallow` as a guide

AOSP uses strict **`neverallow`** rules to stop:

- Vendor apps from talking to system services
- System processes from talking to unknown vendor daemons

So **you may need to define an `interface`** in **`system_private.te`** or **`vendor_sys.te`**.

✅ Summary

Comm Type	What to Allow
Binder	**`allow A B:binder call;`**
Socket	Label socket + allow **`sock_file`** & **`connectto`**
File	Allow file read/write

Let me know how **`mydaemon`** communicates — I'll give you exact rules.

14. What domain explain in simple ways

Absolutely — let's keep it super simple.

What is a SELinux "Domain"?

Think of a **domain** like a **box** around a running process.

> It controls **what that process is allowed to do**.

Real-life example:

You have 2 services:

- `mydaemon`
- `system_server`

They each run in **different SELinux domains**:

- `mydaemon` → `mydaemon` domain
- `system_server` → `system_server` domain

Why domains matter

Each domain:

- Has its **own set of permissions**
- Can **only do** what SELinux rules allow
- **Can't break out** or access other domains unless allowed

Example:

Process	Domain	What it's allowed to do
`mydaemon`	`mydaemon`	Read config, write logs
`system_server`	`system_server`	Talk to Android services, manage apps

If `mydaemon` tries to:

- Access something outside its domain (like camera or other process)
  → SELinux blocks it unless **explicit rules** are added.

Simple analogy:

Think of **domains like rooms** in a building:

- Each room has **its own key (label)**
- Only allowed people (apps/processes) can go in or out
- You must set up **permission rules** if one room needs to talk to another

---

Let me know if you want to **see the domain of a process** using `adb shell`. I can show you how!

15. Yes

Nice! Here's how you can **check the SELinux domain** of any running process on your Android device using **`adb`**.

✅ Step-by-step

1. Connect your device

```bash
adb devices
```

---

2. Get the PID of the process

Example: Let's say your process is **`mydaemon`**

```bash
adb shell ps -A | grep mydaemon
```

Output might look like:

```bash
system      1234  ...  /vendor/bin/mydaemon
```

Here, **`1234`** is the **PID**.

---

3. Check SELinux domain using `cat /proc`

```bash
adb shell cat /proc/1234/attr/current
```

You'll see something like:

```makefile
u:r:mydaemon:s0
```

This is the **SELinux context**:

- **`u`**: user (always **`u`**)
- **`r:mydaemon`**: domain → this is what you're looking for!
- **`s0`**: level (you can usually ignore)

Example for `system_server`:

```bash
adb shell ps -A | grep system_server
adb shell cat /proc/<PID>/attr/current
```

Might return:

```makefile
u:r:system_server:s0
```

Let me know if you want a quick shell script to automate this lookup!

16. Explain more

Sure! Let's go **step by step** and explain everything in more detail — still keeping it easy to understand.

What is a SELinux domain (again)?

A **domain** is a security "zone" where a **process runs**.

Each domain has a name, like:

- **`system_server`**
- **`vold`**
- **`netd`**
- **`mydaemon`** (if you made your own)

Each domain has a list of **what it can and can't do** — like:

- What files it can read
- What sockets it can use
- Which services it can talk to

How do you check what domain a process is in?

Let's do a real example with commands.

Step 1: List all running processes

```bash
adb shell ps -A
```

This shows a list like:

```bash
u0_a76     10234  ... com.example.app
root       1234   ... /vendor/bin/mydaemon
system     892    ... system_server
```

Step 2: Pick the process you want

Let's say you want to check **`mydaemon`**, and its **PID is 1234**.

Step 3: Use this command to see its SELinux context

```bash
adb shell cat /proc/1234/attr/current
```

This will print:

```makefile
u:r:mydaemon:s0
```

🔍 What does this mean?

Part	Meaning
`u`	SELinux user (always **`u`** on Android)
`r:mydaemon`	This is the **domain** → your process is running in the **`mydaemon`** security zone
`s0`	Security level (Android usually just uses **`s0`**)

Why this matters:

If your process is running in the **wrong domain**, SELinux might:

- Block it from reading files
- Block it from talking to other services
- Block it from even starting

So **you must make sure**:

- The binary is labeled correctly (via **`file_contexts`**)
- The **`.rc`** file sets the correct **`seclabel`**
- The **`.te`** file defines and allows what it needs

Example with `system_server`

Let's try another:

```bash
adb shell ps -A | grep system_server
```

Say the PID is 892.

Then:

```bash
adb shell cat /proc/892/attr/current
```

Returns:

```makefile
u:r:system_server:s0
```

Means: **`system_server`** runs in the **`system_server`** domain.

Want me to show how to **make sure your daemon runs in your domain** and not **`init`** or **`u:r:shell`** by mistake?

17. How two process in different domains talk with selinux please explain in simple

Great — let’s explain **how two processes in different SELinux domains talk to each other**, in the **simplest way possible**.

✅ First: What’s the problem?

Two processes — let’s say:

- **`system_server`** → in domain **`system_server`**
- **`mydaemon`** → in domain **`mydaemon`**

You want them to **communicate**, for example using:

- **Binder**
- **Socket**
- **File sharing**

But **SELinux will block it** unless you **tell it**:

> "These two are allowed to talk."

✅ Second: How does SELinux allow communication?

SELinux checks:

- **Who is sending** (source domain)
- **Who is receiving** (target domain or resource)
- **What method** is used (binder, socket, file, etc.)

Then it checks if a rule like this exists:

```c
allow system_server mydaemon: binder call;
```

If yes → allow
If no → deny

✅ Third: Common communication types

1. Binder

For Android service calls

```c
allow system_server mydaemon: binder call;
```

Means: system_server can call into mydaemon via binder

---

2. Socket

If mydaemon creates a socket (e.g., **`/dev/socket/mydaemon.sock`**):

You must:

1. Label the socket in **`file_contexts`**:

```bash
/dev/socket/mydaemon.sock  u:object_r:mydaemon_socket:s0
```

2. Add SELinux rules:

```c
allow system_server mydaemon_socket:sock_file write;
allow system_server mydaemon:unix_stream_socket connectto;
```

---

3. Shared File

If both processes read/write the same file:

```c
allow system_server mydaemon_shared_file:file { read write open };
```

✅ Summary (Super Simple)

Method	What you allow
Binder	**`allow A B: binder call;`**
Socket	**`allow A B_socket:sock_file write;`**
File	**`allow A file_type:file read;`**

If you don't write the **`allow`** rule, SELinux **denies access**.

Want help writing rules for your **specific case** (e.g., binder, socket)? Just tell me how the two processes are talking.

18. File contexts

Great! Let's now explain **`file_contexts`** in a very simple way.

✅ What is `file_contexts`?

It's a text file that tells SELinux:

> "This file or folder should have this security label."

That label controls:

- What domain can access it
- What permissions it gets

✅ Example line in `file_contexts`

```bash
/vendor/bin/mydaemon     u:object_r:mydaemon_exec:s0
```

This means:

- The binary at **`/vendor/bin/mydaemon`**
- Will get the SELinux label: **`u:object_r:mydaemon_exec:s0`**

✅ Why is this important?

If your binary or file **doesn't have the correct label**, SELinux will block access!

Even if your **`.te`** file allows it.

✅ File types you usually label

Path	Why label it
`/vendor/bin/...`	For executables (daemons, services)
`/dev/socket/...`	For communication via sockets
`/data/...`	For configs, logs, databases
`/mnt/vendor/...`	For vendor-specific files

✅ More examples

1. Binary

```bash
/vendor/bin/mydaemon    u:object_r:mydaemon_exec:s0
```

2. Socket

```bash
/dev/socket/mydaemon.sock    u:object_r:mydaemon_socket:s0
```

3. Config file

```bash
/data/mydaemon/config.json    u:object_r:mydaemon_config_file:s0
```

✅ Where to put this file?

Usually in your AOSP device tree:

```php-template
device/<vendor>/<device>/sepolicy/vendor/file_contexts
```

Or:

```bash
sepolicy/file_contexts
```

✅ After editing `file_contexts`

Run:

```bash
make sepolicy
```

And make sure you **relabel** if needed:

```bash
adb shell restorecon -v /vendor/bin/mydaemon
```

Want help writing the `file_contexts` line for your file? Just tell me the path!

19. How write into

Perfect — let's now explain **how to write into `file_contexts`** step-by-step, simply.

✅ Goal:

Tell SELinux:

> "This file (or folder) should have this label."

✅ Format of a `file_contexts` line:

```bash
<file path>    <SELinux label>
```

Example:

```bash
/vendor/bin/mydaemon    u:object_r:mydaemon_exec:s0
```

That line says:

- SELinux should label **`/vendor/bin/mydaemon`**
- As **`mydaemon_exec`**

✅ Step-by-step: How to add your own

1. Find the right `file_contexts` file

Location in AOSP (device-specific):

```bash
device/<your_vendor>/<your_device>/sepolicy/vendor/file_contexts
```

Or common location:

```bash
sepolicy/vendor/file_contexts
```

---

2. Edit the file and add your line

Example: If you have these files:

File	Purpose	SELinux type
`/vendor/bin/mydaemon`	Your binary	**`mydaemon_exec`**
`/data/mydaemon/config.json`	Your config	**`mydaemon_config_file`**
`/dev/socket/mydaemon.sock`	Socket	**`mydaemon_socket`**

Then add:

```bash
/vendor/bin/mydaemon        u:object_r:mydaemon_exec:s0
/data/mydaemon/config.json  u:object_r:mydaemon_config_file:s0
/dev/socket/mydaemon.sock   u:object_r:mydaemon_socket:s0
```

3. Build your image

Rebuild **`sepolicy`**:

```bash
make sepolicy
```

Or rebuild the system/vendor image:

```bash
make systemimage  # or vendorimage, if needed
```

4. (Optional) Relabel manually during testing

Use **`restorecon`**:

```bash
adb shell restorecon -v /vendor/bin/mydaemon
adb shell restorecon -v /data/mydaemon/config.json
```

5. Verify labels

Check if the label is applied:

```bash
adb shell ls -Z /vendor/bin/mydaemon
```

Expected output:

```bash
u:object_r:mydaemon_exec:s0  /vendor/bin/mydaemon
```

Want me to generate **`file_contexts`** lines for your actual files? Just tell me their paths and what they do.

20. How to find it

Great question! Let's go over **how to find the SELinux label (file context)** of any file on your Android device or emulator.

✅ How to find the SELinux label of a file

You use the `ls -Z` command.

Example:

```bash
adb shell ls -Z /vendor/bin/mydaemon
```

You might see:

```bash
u:object_r:mydaemon_exec:s0 /vendor/bin/mydaemon
```

This line tells you the **SELinux label** of that file.

✅ Step-by-step

1. Connect your device

```bash
adb devices
```

2. Check the file label

```bash
adb shell ls -Z <file_path>
```

Examples:

```bash
adb shell ls -Z /vendor/bin/mydaemon
adb shell ls -Z /data/mydaemon/config.json
adb shell ls -Z /dev/socket/mydaemon.sock
```

✅ Bonus: Check a directory of files

Want to check all files in a folder?

```bash
adb shell ls -Z /vendor/bin/
```

It will list SELinux labels for everything there.

✅ What if the label is wrong?

You can **reset it using `restorecon`**:

```bash
adb shell restorecon -v /vendor/bin/mydaemon
```

That applies the correct label from your **`file_contexts`**.

Let me know if you want help fixing a label that's showing wrong or missing!

21. But inwhich file we should write

Ah, now I get your question — you're asking:

> **"Where exactly should I write the `file_contexts` entries in AOSP?"**

Let's answer clearly and simply:

✅ Where to write `file_contexts` entries in AOSP?

You should add them in the **`file_contexts`** file for your device or vendor.

Common locations:

1. For device-specific files (like custom daemons)

```php-template
device/<vendor>/<device>/sepolicy/vendor/file_contexts
```

For example:

```bash
device/qcom/msm8998/sepolicy/vendor/file_contexts
```

---

2. For common vendor-wide changes (shared among devices)

```ruby
device/<vendor>/common/sepolicy/vendor/file_contexts
```

---

3. For system partition (if not using `vendor/`)

```php-template
device/<vendor>/<device>/sepolicy/private/file_contexts
```

Or:

```arduino
sepolicy/private/file_contexts
```

✅ Rule of thumb:

If your file is in...	Then edit this file...
`/vendor/...`	`vendor/file_contexts`
`/system/...`	`private/file_contexts`
`/data/...`	Same (depends if daemon is vendor or system)
`/dev/socket/...`	Depends on who creates it (usually vendor)

✅ Example

You created:

- **`/vendor/bin/mydaemon`**
- **`/dev/socket/mydaemon.sock`**

Then edit:

```php-template
device/<vendor>/<device>/sepolicy/vendor/file_contexts
```

Add:

```bash
/vendor/bin/mydaemon      u:object_r:mydaemon_exec:s0
/dev/socket/mydaemon.sock u:object_r:mydaemon_socket:s0
```

Let me know your file paths, and I'll tell you **exactly** where to put your **`file_contexts`** lines.

22. Files also have context

Yes — **absolutely!** Files in Android (and in any SELinux-enabled system) **also have their own SELinux context** — just like processes do.

✅ What is a file context?

A **file context** is a security label assigned to a file or directory.

It looks like:

```bash
u:object_r:<type>:s0
```

Example:

```bash
u:object_r:mydaemon_exec:s0
```

Here's what each part means:

Part	Meaning
`u`	SELinux user (`u` is common for all in Android)
`object_r`	Role (usually `object_r` for files)
`mydaemon_exec`	**Type** (important part for SELinux rules)
`s0`	Security level (usually just `s0` in Android)

✅ Why does this matter?

Because **processes can only access files** if SELinux **allows their domain to access the file's context**.

✅ Real-world example:

You have:

- A process in `mydaemon` domain
- A binary at `/vendor/bin/mydaemon`

If the file has the context:

```makefile
u:object_r:mydaemon_exec:s0
```

Then you can allow it like:

```c
allow mydaemon mydaemon_exec:file { read execute open };
```

But if the file had the wrong context like:

```makefile
u:object_r:system_file:s0
```

SELinux would **block it**, even if everything else is fine.

✅ How do you check file contexts?

```bash
adb shell ls -Z /path/to/your/file
```

Example:

```bash
adb shell ls -Z /vendor/bin/mydaemon
```

Output:

```bash
u:object_r:mydaemon_exec:s0 /vendor/bin/mydaemon
```

✅ How to fix if the context is wrong?

```bash
adb shell restorecon -v /vendor/bin/mydaemon
```

This resets the file label using your **`file_contexts`** rules.

Would you like help writing **`file_contexts`** and **`.te`** rules together for your setup? Just drop the paths or domains you're working with!

23. Yes

Awesome! Let's go through a **full example** of how to properly set up **`file_contexts`** and **`.te`** rules for your custom daemon or service — step-by-step and easy.

✅ Example setup: You have a custom daemon

Let's say you added a daemon called **`mydaemon`**, located at:

- **Binary**: **`/vendor/bin/mydaemon`**
- **Socket**: **`/dev/socket/mydaemon.sock`**
- **Config file**: **`/data/mydaemon/config.json`**

You want it to:

- Run in its own domain: **`mydaemon`**
- Only access its config file and socket
- Be allowed to start by **`init`**

✅ Step 1: Define file labels in `file_contexts`

Open this file:

```php-template
device/<vendor>/<device>/sepolicy/vendor/file_contexts
```

Add these lines:

```bash
/vendor/bin/mydaemon              u:object_r:mydaemon_exec:s0
/dev/socket/mydaemon.sock        u:object_r:mydaemon_socket:s0
/data/mydaemon/config.json       u:object_r:mydaemon_config_file:s0
```

✅ Step 2: Create the `mydaemon.te` file

Location:

```php-template
device/<vendor>/<device>/sepolicy/vendor/mydaemon.te
```

Example content:

```c
# Declare the domain
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;
type mydaemon_socket, sock_file, file_type;
type mydaemon_config_file, file_type;

# Allow daemon to start from init
init_daemon_domain(mydaemon)

# Allow execution of the binary
allow mydaemon mydaemon_exec:file { read execute open };

# Allow creation and usage of the socket
allow mydaemon mydaemon_socket:sock_file { create write getattr setattr };
allow mydaemon self:unix_stream_socket create_stream_socket_perms;
```

```
# Allow access to its config
allow mydaemon mydaemon_config_file:file { read open getattr };
```

✅ Step 3: Add to `init.rc`

In **`device/<vendor>/<device>/init/`**, your **`mydaemon.rc`** might look like:

```bash
service mydaemon /vendor/bin/mydaemon
    class main
    user root
    group system
    seclabel u:r:mydaemon:s0
    socket mydaemon stream 660 system system
    oneshot
```

✅ Step 4: Rebuild and relabel

- Build:

  ```bash
  make sepolicy
  make vendorimage
  ```

- On device, make sure files are relabeled:

  ```bash
  adb shell restorecon -Rv /vendor/bin/mydaemon
  adb shell restorecon -Rv /dev/socket/mydaemon.sock
  adb shell restorecon -Rv /data/mydaemon
  ```

- Verify:

  ```bash
  adb shell ls -Z /vendor/bin/mydaemon
  adb shell ps -A -Z | grep mydaemon
  ```

✅ Done!

Now:

- Your binary is labeled
- SELinux knows the rules
- Your daemon runs safely and is allowed only what you define

---

Want me to help you write this for **your actual file and domain names**? Just tell me:

- The path to your binary
- What files/sockets it uses
- What other process should talk to it (if any)

24. In concise

Sure — here's the **concise version**:

✅ Goal: Add SELinux rules for `mydaemon`

Files used:

- `/vendor/bin/mydaemon` → binary
- `/dev/socket/mydaemon.sock` → socket
- `/data/mydaemon/config.json` → config

✅ 1. `file_contexts`

Edit:

```php-template
device/<vendor>/<device>/sepolicy/vendor/file_contexts
```

Add:

```bash
/vendor/bin/mydaemon              u:object_r:mydaemon_exec:s0
/dev/socket/mydaemon.sock        u:object_r:mydaemon_socket:s0
/data/mydaemon/config.json       u:object_r:mydaemon_config_file:s0
```

✅ 2. `mydaemon.te`

Create/edit:

```php-template
device/<vendor>/<device>/sepolicy/vendor/mydaemon.te
```

Add:

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;
type mydaemon_socket, sock_file, file_type;
type mydaemon_config_file, file_type;

init_daemon_domain(mydaemon)

allow mydaemon mydaemon_exec:file { read execute open };
allow mydaemon mydaemon_socket:sock_file { create write };
allow mydaemon self:unix_stream_socket create_stream_socket_perms;
allow mydaemon mydaemon_config_file:file { read open };
```

✅ 3. `mydaemon.rc`

```bash
service mydaemon /vendor/bin/mydaemon
    class main
    user root
    group system
    seclabel u:r:mydaemon:s0
```

```
    socket mydaemon stream 660 system system
    oneshot
```

✅ 4. Build & Relabel

```bash
make sepolicy
make vendorimage
adb shell restorecon -Rv /vendor/bin/mydaemon /data/mydaemon /dev/socket/
```

Done!

Let me know if you want this template filled for your real file names.

25. How to add this rc file to init rc

Great question! Here's how to **add your custom `.rc` file** (like **`mydaemon.rc`**) so it gets included in Android's **`init`** process.

✅ Step-by-step: Add your `mydaemon.rc` to `init`

1. Place your `.rc` file

Put your **`mydaemon.rc`** here:

```php-template
device/<vendor>/<device>/mydaemon.rc
```

---

2. Register it in the device makefile

Edit this file:

```php-template
device/<vendor>/<device>/<device>.mk
```

And **add:**

```makefile
PRODUCT_COPY_FILES += \
    device/<vendor>/<device>/mydaemon.rc:$(TARGET_COPY_OUT_VENDOR)/etc/init/mydaemon.rc
```

This copies your **`.rc`** into **`/vendor/etc/init/`**, where Android's **`init`** system will find it at boot.

---

3. Rebuild

Rebuild your image:

```bash
make vendorimage
```

---

4. Verify on device

Check that it was installed:

```bash
adb shell ls /vendor/etc/init/
```

You should see **`mydaemon.rc`**.

That's it — your service will now be started by Android's **`init`** using your custom **`.rc`** file!

Want help writing the **`.rc`** file content too?

26. PRODUCT_COPY_FILES += \
device/<vendor>/<device>/mydaemon.rc:$(TARGET_COPY_OUT_VENDOR)/etc/init/mydaemon.rc

Yes — that's exactly the correct line to put in your device makefile (`<device>.mk`) to copy your `.rc` file into the vendor `init/` directory at build time.

✅ Breakdown of this line:

```makefile
PRODUCT_COPY_FILES += \
    device/<vendor>/<device>/mydaemon.rc:$(TARGET_COPY_OUT_VENDOR)/etc/init/mydaemon.rc
```

Part	Meaning
`PRODUCT_COPY_FILES += \`	Appending to the list of files to copy into the system image
`device/<vendor>/<device>/mydaemon.rc`	Path of your `.rc` file in the AOSP tree
`$(TARGET_COPY_OUT_VENDOR)/etc/init/mydaemon.rc`	Destination path in the built vendor image (`/vendor/etc/init/`)

✅ Example for a real device:

If your device is `qcom/sdm660`:

```makefile
PRODUCT_COPY_FILES += \
    device/qcom/sdm660/mydaemon.rc:$(TARGET_COPY_OUT_VENDOR)/etc/init/mydaemon.rc
```

✅ After adding this line:

1. Run `make vendorimage`
2. Flash the new vendor image or full image set
3. Check on device:

```bash
adb shell ls /vendor/etc/init/mydaemon.rc
```

Let me know if you want to automate service startup in this `.rc` too!

27. So it means that only domains matter not partitions if selinux policy is not valid

Exactly! You're getting it.

✅ Short Answer:

Yes — in SELinux, *domains* (i.e., process contexts) are what matter most. Not partitions.

If a SELinux policy **does not allow** one domain to interact with another domain or file type — **it doesn't matter** whether they're in the same or different partitions (`/system`, `/vendor`, `/data`, etc.).

✅ Example:

Imagine:

- Process A is in domain `mydaemon`
- Process B is in domain `hal_camera_default`
- A file is in `/vendor/bin/`, but labeled `system_file`

Even if they're on the same partition (`/vendor`), **SELinux will block** communication **if the policy doesn't allow `mydaemon` to interact with `hal_camera_default` or `system_file`.**

✅ Why partitions are still "kinda" important:

Partition location affects:

1. **Which file_contexts file** you edit (`vendor/file_contexts`, `system/private/file_contexts`, etc.)
2. **Which policy scope** the labels fall under (`system`, `vendor`, `product`, etc.)

But at runtime, SELinux just checks:

> *"Can domain X access type Y with this operation?"*

Not: *"Which partition is this file on?"*

✅ Summary:

SELinux checks	Important?
Process domain	YES
File/socket label (type)	YES
Allowed **`allow`** rules in **`.te`**	YES
Partition (**`/vendor`**, **`/system`**, etc.)	**NO** (not directly checked by SELinux)

Let me know if you want a diagram or real example showing two processes in different domains communicating or getting blocked.

28. What is default policy for domain to domain interaction

Great question — and very important to understand!

✅ Default SELinux policy for domain-to-domain interaction in Android:

By default, domain-to-domain interaction is denied.

SELinux follows a **"deny by default"** approach. This means:

> **Unless you explicitly `allow` domain A to interact with domain B, it will be blocked.**

✅ Example:

Let's say:

- Process A runs in domain: `mydaemon`
- Process B runs in domain: `hal_camera_default`

If `mydaemon` tries to send a binder or socket message to `hal_camera_default`, it will **fail** unless your policy includes:

```c
allow mydaemon hal_camera_default:binder { call transfer };
allow mydaemon hal_camera_default:unix_stream_socket connectto;
```

✅ So what happens by default?

Without allow rules:

Action	SELinux Result
Binder call from one domain to another	**Denied**
Socket connect	**Denied**
File access	**Denied**
Sending signals	**Denied**
Any cross-domain communication	**Denied**

✅ How do you allow it?

You use rules in your `.te` file like:

```c
allow mydaemon hal_service:service_manager find;
allow mydaemon hal_camera_default:binder call;
allow mydaemon hal_camera_default:unix_stream_socket connectto;
```

✅ How to find what to allow?

Use:

```bash
adb shell dmesg | grep avc
```

Example AVC denial:

```bash
avc: denied { call } for pid=1234 comm="mydaemon" scontext=u:r:mydaemon:s0 tcontext=u:r:hal_camera_default:s0 tclass=binder
```

This tells you:

- Source: **`mydaemon`**
- Target: **`hal_camera_default`**
- Class: **`binder`**
- Permission: **`call`**

And now you know exactly what to **`allow`**.

Let me know if you'd like help writing an allow rule for a specific denial you're seeing.

29. Yes

Perfect! Just give me the **AVC denial message** you're seeing — like this example:

```bash
avc: denied { call } for pid=1234 comm="mydaemon" scontext=u:r:mydaemon:s0 tcontext=u:r:hal_camera_default:s0 tclass=binder
```

From that, I'll help you write the exact **`allow`** rule you need for your **`.te`** file.

You can get AVC logs by running:

```bash
adb shell dmesg | grep avc
```

or

```bash
adb logcat | grep avc
```

Once you paste it here, I'll turn it into a proper policy rule for you.

30. different types of allow rules

Great! Let's go over the **different types of `allow` rules** in SELinux — especially those you'll commonly use in Android.

✅ What an `allow` rule looks like:

```c
allow <source_domain> <target_type_or_domain>:<class> { <permissions> };
```

This tells SELinux:

> Allow a subject (process in **`<source_domain>`**) to perform **`<permissions>`** on an object (file, socket, or another domain) of **`<class>`** type.

✅ 1. File access

For reading, writing, or executing files.

```c
allow mydaemon mydaemon_config_file:file { read open getattr };
allow mydaemon mydaemon_exec:file { execute open };
```

- **`file`** → file or binary
- Permissions: **`read`**, **`write`**, **`open`**, **`execute`**, **`getattr`**, etc.

✅ 2. Directory access

To access folders (e.g. list files).

```c
allow mydaemon my_config_dir:dir { search read open };
```

- Permissions: **`search`**, **`add_name`**, **`remove_name`**, etc.

✅ 3. Binder communication

Android processes often talk over Binder.

```c
allow myapp system_server:binder call;
```

- **`binder`** → interprocess communication
- Permissions: **`call`**, **`transfer`**

✅ 4. Socket communication

For UNIX domain or network sockets.

```c
allow mydaemon hal_camera_default:unix_stream_socket connectto;
allow mydaemon self:unix_stream_socket create_stream_socket_perms;
```

- Permissions: **`connectto`**, **`create`**, **`write`**, etc.

✅ 5. Service manager

Allowing a domain to find or add services.

```c
allow mydaemon hal_camera_service:service_manager find;
allow system_server myservice:service_manager add;
```

✅ 6. Signal sending

For sending **`kill`**, **`sigchld`**, etc.

```c
allow app_a app_b:process sigchld;
allow system_server mydaemon:process sigkill;
```

✅ 7. Property access

Android system properties (**`setprop`**, **`getprop`**):

```c
allow mydaemon my_property:property_service set;
allow myapp system_prop:property_service getattr;
```

✅ 8. Shared memory / IPC

For pipes, sockets, semaphores, etc.

```c
allow appdomain system_server:fifo_file { write read };
allow myservice myservice:shm { read write create };
```

✅ 9. Tmpfs/anon inodes

If your service uses **`/dev/ashmem`**, **`/dev/ion`**, etc.

```c
allow mydaemon ashmem_device:chr_file { read write };
```

✅ Summary Cheat Sheet:

Purpose	Class	Example Permission
File access	**`file`**	**`read`**, **`write`**, **`execute`**
Directory access	**`dir`**	**`search`**, **`open`**
Binder	**`binder`**	**`call`**, **`transfer`**
Socket	**`unix_stream_socket`**	**`connectto`**, **`create`**
Service manager	**`service_manager`**	**`find`**, **`add`**
Signals	**`process`**	**`sigchld`**, **`sigkill`**
Property	**`property_service`**	**`set`**, **`getattr`**
IPC/shared mem	**`fifo_file`**, **`shm`**	**`read`**, **`write`**

Got a real denial or use case? I can help write the exact allow rules you need.

31. what is class here explain in simple

Awesome question — and let's keep it **super simple**!

✅ What is `class` in SELinux?

`class` means:
👉 *"What kind of object are we talking about?"*

It's the **type of resource** your process is trying to access.

✅ Real-world analogy:

Imagine you're at a mall:

- 🧍 You = the process (in a domain)
- 🎯 The place you want to go = object (file, socket, process, etc.)
- 📦 The *class* = what kind of thing it is (shop, ATM, lift, etc.)
- 🎟️ The *permissions* = what you're allowed to do there (buy, withdraw, press button)

✅ Example in SELinux:

```c
allow mydaemon myconfig_file:file { read open };
```

Part	Meaning
`mydaemon`	The process (domain)
`myconfig_file`	The object (file label/type)
`file`	✅ **This is the `class`** – it's a **file**
`{ read open }`	What the process is allowed to do

✅ Common `class` values (in human terms):

SELinux Class	Real-world meaning
`file`	A regular file
`dir`	A folder/directory
`binder`	Android's binder IPC (talking to other processes)
`unix_stream_socket`	A socket used for talking (like a pipe)
`service_manager`	The Android service registry (e.g., HALs)
`property_service`	System properties (`getprop`, `setprop`)
`process`	Another process (to signal or control)
`fifo_file`	Named pipe (used for IPC)
`chr_file`	Character device (e.g., `/dev/null`)

✅ Why does SELinux use classes?

Because **permissions differ based on the type of object.**

- You can **read** a file but not a socket.
- You can **connectto** a socket, but not a directory.

✅ TL;DR:

Class = the type of object (file, dir, socket, etc.)
It tells SELinux *what kind of thing you're trying to access*, so it knows what rules to apply.

Want me to explain a real **`class`** you saw in an AVC log? Just paste it here.

32. table of all classes with possible permission

Certainly! Here's a concise table of common SELinux object classes and their associated permissions:

📁 File and Directory Classes

Class	Permissions
`file`	`read`, `write`, `execute`, `append`, `getattr`, `setattr`, `create`, `unlink`, `rename`, `link`, `open`, `ioctl`, `lock`, `relabelfrom`, `relabelto`, `entrypoint`, `execute_no_trans`, `execmod`
`dir`	`search`, `add_name`, `remove_name`, `reparent`, `rmdir`, `getattr`, `setattr`, `read`, `write`, `create`, `unlink`, `rename`, `link`, `open`, `ioctl`, `lock`, `relabelfrom`, `relabelto`

🔌 Socket Classes

Class	Permissions
`unix_stream_socket`	`connectto`, `accept`, `create`, `write`, `read`, `getattr`, `setattr`, `ioctl`, `listen`, `bind`, `connect`, `getopt`, `setopt`, `shutdown`, `recvfrom`, `sendto`, `name_bind`
`tcp_socket`	`connectto`, `accept`, `create`, `write`, `read`, `getattr`, `setattr`, `ioctl`, `listen`, `bind`, `connect`, `getopt`, `setopt`, `shutdown`, `recvfrom`, `sendto`, `name_bind`
`udp_socket`	`create`, `write`, `read`, `getattr`, `setattr`, `ioctl`, `bind`, `connect`, `getopt`, `setopt`, `shutdown`, `recvfrom`, `sendto`, `name_bind`

🔁 IPC (Inter-Process Communication) Classes

Class	Permissions
`process`	`fork`, `transition`, `sigchld`, `sigkill`, `sigstop`, `signull`, `signal`, `ptrace`, `getsched`, `setsched`, `getsession`, `getpgid`, `setpgid`, `getcap`, `setcap`, `getattr`, `setexec`, `setfscreate`, `noatsecure`, `siginh`, `dyntransition`, `getrlimit`, `setrlimit`, `rlimitinh`, `setcurrent`, `execmem`, `execstack`, `execheap`, `setkeycreate`, `setsockcreate`
`shm`	`create`, `destroy`, `read`, `write`, `associate`, `getattr`, `setattr`, `lock`, `relabelfrom`, `relabelto`
`sem`	`create`, `destroy`, `read`, `write`, `associate`, `getattr`, `setattr`, `lock`, `relabelfrom`, `relabelto`
`msgq`	`create`, `destroy`, `read`, `write`, `associate`, `getattr`, `setattr`, `lock`, `relabelfrom`, `relabelto`

🧩 Other Notable Classes

Class	Permissions
`binder`	`call`, `transfer`, `impersonate`, `set_context_mgr`, `get_context_mgr`
`service_manager`	`add`, `find`, `list`
`property_service`	`set`, `get`
`capability`	`chown`, `dac_override`, `dac_read_search`, `fowner`, `fsetid`, `kill`, `setgid`, `setuid`, `setpcap`, `linux_immutable`, `net_bind_service`, `net_broadcast`, `net_admin`, `net_raw`, `ipc_lock`, `ipc_owner`, `sys_module`, `sys_rawio`, `sys_chroot`, `sys_ptrace`, `sys_pacct`, `sys_admin`, `sys_boot`, `sys_nice`, `sys_resource`, `sys_time`, `sys_tty_config`, `mknod`, `lease`, `audit_write`, `audit_control`, `setfcap`

For a comprehensive list of SELinux object classes and their permissions, you can refer to the SELinux Notebook on GitHub or the Red Hat SELinux documentation.

Let me know if you need further details or examples on any specific class or permission!

33. can two processes in same partition talk each other without selinux policy?

🔒 **No — two processes in the same partition cannot talk to each other without SELinux policy allowing it.**

✅ Simple Explanation:

SELinux does not care about:

- What partition the processes are on (`/system`, `/vendor`, `/data`, etc.)

SELinux only cares about:

- The **domain** of the source process (e.g. `mydaemon`)
- The **domain** of the target process (e.g. `hal_camera_default`)
- Whether there's an **`allow` rule** in the SELinux policy that says:

> "Yes, this domain can talk to that one over this type of communication."

📦 Same Partition ≠ Communication Allowed

Even if both processes:

- Live in `/system/bin/`
- Are started by the same `init.rc`
- Are built from the same source

They **still cannot communicate** unless:

- An `allow` rule is written in their `.te` files
- AND the file/socket/binder they use is also allowed by SELinux

---

✅ Example:

Both processes in `/system/bin/`:

```bash
/system/bin/mydaemon    → domain: mydaemon
/system/bin/otherproc   → domain: system_server
```

If `mydaemon` tries to call `otherproc` via binder or socket, it will get blocked unless you have:

```c
allow mydaemon system_server:binder call;
```

Even though they're on the **same partition**.

🧠 TL;DR:

✅ Can communicate?	Condition
Yes	If SELinux policy explicitly allows it
❌ No	If SELinux does **not** allow it
❌ No	Even if on same partition

Let me know if you want to test this with a real example or denial log — I'll help you write the rule.

34. what if they belong to same domain

✅ **If two processes belong to the same domain, they *can* communicate — as long as the type of communication is allowed within that domain.**

Let's break it down simply 👇

🔸 Scenario: Same Domain

Example:

- Process A: `/system/bin/daemon1`
- Process B: `/system/bin/daemon2`
- Both run in domain: `mydaemon`

Then:

- **SELinux treats them equally**
- Communication is controlled by whether the domain (`mydaemon`) is allowed to interact with itself.

✅ SELinux Rule Example

If `mydaemon` wants to send a binder call to another process **also in** `mydaemon`, you'd need:

```c
allow mydaemon mydaemon:binder call;
```

Or if they use a **socket**, you'd need:

```c
allow mydaemon mydaemon:unix_stream_socket connectto;
```

So yes, even for **same-domain communication**, **you still need a policy rule**, but it's a **self-to-self rule**.

🔐 Summary:

Situation	Can they communicate?	Need a rule?
Same domain	✅ Yes, if rule exists	✅ Yes
Same domain, no rule	❌ No	✅ Add rule
Different domain, no rule	❌ No	✅ Add rule
Different domain, rule exists	✅ Yes	✅ Rule needed

Want me to help write a self-domain allow rule for your daemon? Just tell me how they communicate (e.g., binder, socket).

35. in this still we need allow rules

✅ Yes — **even if both processes are in the same SELinux domain**, you still need **explicit `allow` rules** to let them communicate.
SELinux is strict like that. 👮

🤔 Why?

Because **SELinux controls both:**

1. **Who** is accessing (the source domain)
2. **What** is being accessed (the target type/domain/class)

So when both processes are in the **same domain**, the rule just happens to look like:

```c
allow mydaemon mydaemon:<class> { <permissions> };
```

You're still telling SELinux:

> "Hey, it's okay if processes under **`mydaemon`** talk to each other using **`<class>`** with **`<permissions>`**."

🧠 Real Example

If two **`mydaemon`** processes want to talk via **binder**:

```c
allow mydaemon mydaemon:binder call;
```

If they use a **Unix socket**:

```c
allow mydaemon mydaemon:unix_stream_socket connectto;
```

Or for **shared files**:

```c
allow mydaemon myconfig_file:file { read write open };
```

🔐 Summary:

Both in same domain?	Need allow rule?	Example
✅ Yes	✅ Yes	**`allow mydaemon mydaemon:binder call;`**
❌ No	✅ Yes	**`allow A B:binder call;`**

Even for **self-to-self communication**, SELinux needs to be told what's okay.

Let me know the actual communication type, and I'll give you the exact rule!

36. so inwhich case we don't need sepolicy?

Great question! 🤔
Let's break this down **simply and clearly**:

✅ When You Don't Need to Modify SEPolicy

There are a few rare cases where **you don't need to write or modify SELinux policy**:

✅ 1. Using only already-permitted behavior

If your process:

- Runs in an **existing trusted domain** (like `init`, `system_server`, `hal_*`)
- Uses **files, sockets, and services** that are already labeled and allowed
- Doesn't try to access new things outside its default permissions

✅ **Then SELinux will already allow it.**

📦 *Example:*
A small shell script running under `init` just touches logs or reads `/proc` — no new policy needed.

✅ 2. Processes talking through `binder` and already registered services

- If process A (in domain `system_app`) talks to process B (in domain `system_server`)
- And that is **already defined in existing AOSP SEPolicy**

✅ Then you're fine — **as long as AOSP allows that interaction**.

✅ 3. Your app runs in `untrusted_app` or `platform_app`

- Most Android apps run in `untrusted_app`
- They follow strict rules, and as long as they:
  - Don't try to access `/vendor`, `/system/bin`, `/dev/` etc.
  - Use normal APIs (file storage, camera, etc.)

✅ Then **you don't need to change SEPolicy.**

✅ 4. Using standard HALs and system services

- If you're working with a standard Android HAL (like camera, audio, GPS)
- And follow the **HIDL / AIDL pattern**
- And register services properly

✅ Then all policies are already defined — **no SEPolicy modification needed**.

❌ You WILL Need SEPolicy When:

You do this...	You need SEPolicy?
Add a **new daemon**	✅ Yes
Create a **new binder service**	✅ Yes
Access **non-default files**	✅ Yes
Use a **new socket or device**	✅ Yes

You do this...	You need SEPolicy?
Access **another domain's process**	✅ Yes
Change behavior of **existing domains**	✅ Yes

🧠 TL;DR:

Scenario	Need SEPolicy?
Using existing domain + features	❌ No
New component or new interaction	✅ Yes

Want to tell me your exact use case? I'll tell you **if** and **what** SEPolicy you need — in 2 lines.

37. please explain all available domain in android

In Android's SELinux implementation, **domains** define the security context under which processes operate. Each domain specifies the permissions and access controls for processes, ensuring a secure and compartmentalized system. Here's an overview of commonly used SELinux domains in Android:

📱 Application Domains

- **`untrusted_app`**: Default domain for third-party apps installed from the Play Store or sideloaded.
- **`platform_app`**: For apps signed with the platform key, typically system apps.
- **`system_app`**: Apps running under the system UID, often pre-installed system applications.Stack Overflow
- **`isolated_app`**: Used for apps that run in isolated processes, providing additional security.
- **`priv_app`**: Privileged apps with elevated permissions, usually located in **`/system/priv-app`**.

⚙️ System Service Domains

- **`init`**: The initial process that starts during boot and launches other services.
- **`system_server`**: Hosts core Android services like activity manager and package manager.
- **`vold`**: Volume daemon responsible for mounting and managing storage volumes.
- **`netd`**: Network daemon handling network configuration and management.
- **`zygote`**: Process that launches Android applications by forking.

🔌 Hardware Abstraction Layer (HAL) Domains

- **`hal_camera_default`**, **`hal_audio_default`**, etc.: Domains for specific hardware components, facilitating communication between the Android framework and hardware drivers.

🛠️ Vendor and Custom Domains

- **`vendor_init`**: Handles vendor-specific initialization tasks.
- **`vendor_service`**: Custom services provided by device manufacturers.
- **Custom domains**: Manufacturers can define additional domains for proprietary services or applications.

🔒 Security and Policy Domains

- **`keystore`**: Manages cryptographic keys and operations.
- **`gatekeeperd`**: Handles device authentication mechanisms like PIN and password verification.
- **`keymaster`**: Provides hardware-backed cryptographic services.

🧩 Other Notable Domains

- **`bluetooth`**: Manages Bluetooth services and profiles.
- **`media_server`**: Handles media playback and recording functionalities.
- **`logd`**: Collects and manages system logs.
- **`surfaceflinger`**: Composites window surfaces for display.

These domains are defined and managed through SELinux policy files, typically located in the AOSP source under **`external/sepolicy`** or within device-specific directories. Each domain's policy specifies the allowed interactions with other domains and system resources, enforcing the principle of least privilege. Android Goosource+2

If you need details on a specific domain or guidance on defining custom domains, feel free to ask!

38. Mac and Dac explain in simpler

Absolutely! Let's make **MAC** and **DAC** super simple to understand — like you're explaining it to a friend. 😄

🔐 DAC – *Discretionary Access Control*

🧠 **Think: Normal Linux file permissions.**

- The **owner** of the file decides who can read/write/execute it.
- You've seen this with **`chmod`**, **`chown`**, **`ls -l`**, etc.

📦 **Example:**

```bash
-rw-r--r-- 1 alice alice  file.txt
```

- Alice owns the file.
- She can give or deny access to others.

✅ It's *discretionary* — **the owner has control**.

🔐 MAC – *Mandatory Access Control*

🧠 **Think: SELinux policies.**

- The **system** decides what can happen — *not the user*.
- Even if **`chmod`** says "yes," **SELinux can still say NO**.

📦 **Example:**

- App A has **`read`** permission to a file (DAC says yes)
- But if SELinux says "App A is not allowed to read this type of file," ❌ access denied!

✅ It's *mandatory* — **the system enforces it**, no exceptions.

🎯 Quick Comparison

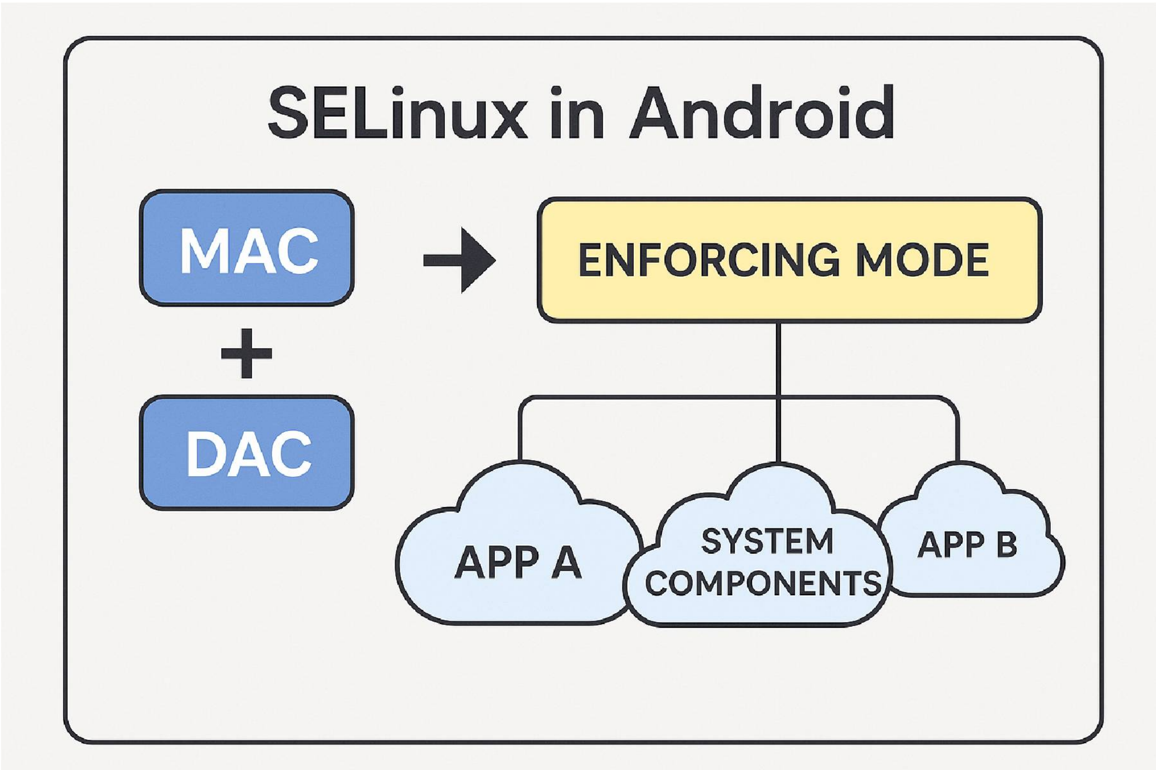
Feature	DAC (Discretionary)	MAC (Mandatory)
Controlled by	File owner	Security policy (e.g., SELinux)
Can override?	Yes (owner decides)	No (only policy changes matter)
Example tool	**`chmod`**, **`chown`**	**`selinux`**, **`.te`** rules
Android use	Unix file permissions	SELinux

📱 Android uses both:

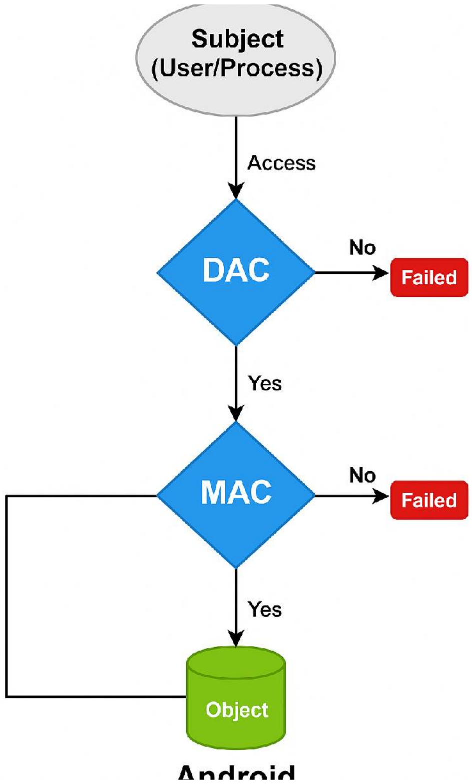
- **DAC** is still there (classic file permissions)
- **MAC** (SELinux) is what **actually protects system components and apps**

Let me know if you want a real-world Android example comparing DAC and MAC!

39. generate diagram for android



40. Uploaded image
generate like this



41. Uploaded image
explain each block in simpler

Sure! Let's explain each block in the image you posted in **simple terms** — perfect for beginners learning **SELinux and access control** on Android/Linux.

🧍 Subject (User/Process)

- This is the **user or app** (process) trying to do something — like **read a file**, **open a socket**, etc.
- It is the **"doer"** — the one trying to access a resource.

🔷 DAC (Discretionary Access Control)

- Linux-style file permissions (`chmod`, `chown`).
- Checks: “Is this user allowed by ownership and permissions?”
- If **No**, ❌ access **denied** right here.
- If **Yes**, it goes to the next step → **MAC**.

🔷 MAC (Mandatory Access Control)

- SELinux comes into play here.
- Even if DAC said “yes,” SELinux can still say “no.”
- Checks based on **security policies** and **contexts**.
- If **MAC allows**, access is granted to the object.
- If **MAC denies**, ❌ access blocked.

🟩 Object

- This is the **target** — the file, socket, service, etc. being accessed.

📦 Access Vector Cache (AVC)

- A **fast lookup table**.
- It stores recent allow/deny decisions so the kernel can check quickly without re-processing everything.

🧠 Security Server (SS)

- If AVC doesn't know the answer (cache miss), it asks the **Security Server**.
- The server checks the **SELinux policy** and returns **YES/NO**.

📃 Policy

- The set of rules that defines **who can access what**.
- Example: `allow system_server logd:unix_dgram_socket sendto;`

📏 MLS (Multi-Level Security) — Optional

- For **advanced security levels** (like in military systems).
- Not used often in regular Android but shown here for completeness.

🔁 Flow Summary (Left to Right):

1. Subject requests access.
2. First, DAC checks basic file permissions.
3. If allowed, MAC checks the request using SELinux.
4. If AVC has the answer, it returns it.
5. If not, the Security Server checks policy and updates AVC.
6. Based on that decision, access is **granted** or **denied**.

🧠 TL;DR:

- DAC = File permissions
- MAC = SELinux policies
- AVC = Fast cache for decisions
- Security Server = Brain that enforces SELinux rules

Let me know if you want this flow with a **real Android example**, like `system_server` trying to read `/dev/something`!

42. .

It looks like you entered just a dot — do you want to:

- Ask a follow-up about the diagram or SELinux?
- Get a real Android example of this DAC → MAC access flow?
- See how this applies when building custom services or HALs?

Let me know — happy to help!

43. Security-Enhanced Linux in Android

bookmark_border
As part of the Android security model, Android uses Security-Enhanced Linux (SELinux) to enforce mandatory access control (MAC) over all processes, even processes running with root/superuser privileges (Linux capabilities). Many companies and organizations have contributed to Android's SELinux implementation. With SELinux, Android can better protect and confine system services, control access to application data and system logs, reduce the effects of malicious software, and protect users from potential flaws in code on mobile devices.

SELinux operates on the principle of default denial: Anything not explicitly allowed is denied. SELinux can operate in two global modes:

Permissive mode, in which permission denials are logged but not enforced.
Enforcing mode, in which permissions denials are both logged and enforced.
Android includes SELinux in enforcing mode and a corresponding security policy that works by default across AOSP. In enforcing mode, disallowed actions are prevented and all attempted violations are logged by the kernel to dmesg and logcat. When developing, you should use these errors to refine your software and SELinux policies before enforcing them. For more details, see Implementing SELinux.

SELinux also supports a per-domain permissive mode in which specific domains (processes) can be made permissive while placing the rest of the system in global enforcing mode. A domain is simply a label identifying a process or set of processes in the security policy, where all processes labeled with the same domain are treated identically by the security policy. Per-domain permissive mode enables incremental application of SELinux to an ever-increasing portion of the system and policy development for new services (while keeping the rest of the system enforcing).

Background
The Android security model is based in part on the concept of application sandboxes. Each application runs in its own sandbox. Prior to Android 4.3, these sandboxes were defined by the creation of a unique Linux UID for each application at time of installation. Android 4.3 and later uses SELinux to further define the boundaries of the Android application sandbox.

In Android 5.0 and later, SELinux is fully enforced, building on the permissive release of Android 4.3 and the partial enforcement of Android 4.4. With this change, Android shifted from enforcement on a limited set of crucial domains (installd, netd, vold and zygote) to everything (more than 60 domains). Specifically:

Everything is in enforcing mode in Android 5.x and higher.
No processes other than init should run in the init domain.
Any generic denial (for a block_device, socket_device, default_service) indicates that device needs a special domain.
Android 6.0 hardened the system by reducing the permissiveness of our policy to include better isolation between users, IOCTL filtering, reduced threat of exposed services, further tightening of SELinux domains, and extremely limited /proc access.

Android 7.0 updated SELinux configuration to further lock down the application sandbox and reduce attack surface. This release also broke up the monolithic mediaserver stack into smaller processes to reduce the scope of their permissions. For more details, see Protecting Android with more Linux kernel defenses and Hardening the media stack.

Android 8.0 updated SELinux to work with Treble, which separates the lower-level vendor code from the Android system framework. This release updated SELinux policy to allow device manufacturers and SOC vendors 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.

While it is possible to have higher/newer platform (framework) version running on the device, the opposite case is not supported; the vendor images (vendor.img/odm.img) cannot have a newer 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 SELinux parts of the policy. The Android 8.0 model provides a method to retain compatibility to prevent unnecessary simultaneous OTAs.

Additional resources
For help constructing useful SELinux policies, refer to the following resources.

Note: Some SELinux concepts are not used by Android. For further details, see Specificity.
The SELinux Notebook, up-to-date reference for SELinux. This document contains further details on the policy language, the meaning of each of the keywords and how security contexts are computed.
Your visual how-to guide for SELinux policy enforcement
Security Enhancements for Linux
Security Enhanced (SE) Android: Bringing Flexible MAC to Android
Implementing SELinux as a Linux Security Module
Configuring the SELinux Policy

Thanks! You've shared a detailed overview of **SELinux in Android**, and it's a great reference. Let me now **simplify and summarize** the key points for easy understanding:

🔐 What is SELinux in Android?

SELinux (Security-Enhanced Linux) is a **Mandatory Access Control (MAC)** system that Android uses to enforce strict security rules on apps and system processes.

It controls:

- Which process can access which file, socket, or service
- Even root (UID 0) cannot bypass SELinux

🧠 Why Android uses SELinux

SELinux helps Android:

- 🔒 Lock down system services (e.g., media, netd)
- 🔐 Isolate apps and their data
- 🛡️ Limit the damage from bugs or malware
- 🔍 Enforce strict access even if DAC (file permissions) say "yes"

🚦 Modes of SELinux

Mode	What it does
Permissive	Logs violations but does **not block** actions
Enforcing	Logs **and blocks** unauthorized actions

Android uses **enforcing mode by default** since Android 5.0.

🎯 Domains

- A **domain** is a SELinux label assigned to a process (e.g., `system_server`, `netd`, `untrusted_app`)
- All processes in the same domain follow the same policy
- SELinux policy defines which domains can interact

🔸 **Per-domain permissive mode**:
You can keep one process in permissive (for debugging) while the rest of Android stays enforcing.

📜 Android SELinux Timeline

Android Version	SELinux Status
4.3	Introduced SELinux (permissive)
4.4	Partial enforcing (zygote, netd, vold)
5.0	Full enforcement (60+ domains)
6.0	Hardened policies (IOCTL filtering, proc isolation)
7.0	Sandboxing improvements (media stack split)
8.0	Treble support — separated **platform** and **vendor** policy parts

📦 Treble and SELinux (Android 8+)

- Platform policy lives in `system.img`
- Vendor policy lives in `vendor.img` / `odm.img`
- **Vendor can't override platform policy**

- Platform must support older vendor policy — to allow updates without breaking compatibility

🔍 Key Concepts Summary

Term	Meaning
MAC	Mandatory Access Control (SELinux enforces rules beyond file permissions)
DAC	Discretionary Access Control (normal Linux permissions: `chmod`, `chown`)
Domain	SELinux label for a process
Context	Full label for files or processes (e.g. `u:object_r:system_file:s0`)
AVC	Access Vector Cache – stores allow/deny decisions for speed
Policy	Rules written in `.te` files that define who can access what and how

✅ What You Should Remember as a Developer

- SELinux is always on in production (enforcing mode)
- You must write proper `.te` rules for your HALs/daemons
- Denials appear in `dmesg` or `logcat | grep avc`
- Use **permissive domains during development**, not whole-system permissive
- Vendor policy must be compatible with platform policy

If you'd like:

- A flowchart of SELinux flow
- A `.te` rule writing guide
- Examples of fixing real AVC denials

Just ask — I'll generate or explain them for you.

44. SELinux concepts

bookmark_border
Review this page to become familiar with SELinux concepts.

Mandatory access control
Security Enhanced Linux (SELinux), is a mandatory access control (MAC) system for the Linux operating system. As a MAC system, it differs from Linux's familiar discretionary access control (DAC) system. In a DAC system, a concept of ownership exists, whereby an owner of a particular resource controls access permissions associated with it. This is generally coarse-grained and subject to unintended privilege escalation. A MAC system, however, consults a central authority for a decision on all access attempts.

SELinux has been implemented as part of the Linux Security Module (LSM) framework, which recognizes various kernel objects, and sensitive actions performed on them. At the point at which each of these actions would be performed, an LSM hook function is called to determine whether or not the action should be allowed based on the information for it stored in an opaque security object. SELinux provides an implementation for these hooks and management of these security objects, which combine with its own policy, to determine the access decisions.

Along with other Android security measures, Android's access control policy greatly limits the potential damage of compromised machines and accounts. Using tools like Android's discretionary and mandatory access controls gives you a structure to ensure your software runs only at the minimum privilege level. This mitigates the effects of attacks and reduces the likelihood of errant processes overwriting or even transmitting data.

In Android 4.3 and higher, SELinux provides a mandatory access control (MAC) umbrella over traditional discretionary access control (DAC) environments. For instance, software must typically run as the root user account to write to raw block devices. In a traditional DAC-based Linux environment, if the root user becomes compromised that user can write to every raw block device. However, SELinux can be used to label these devices so the process assigned the root privilege can write to only those specified in the associated policy. In this way, the process cannot overwrite data and system settings outside of the specific raw block device.

See Use Cases for more examples of threats and ways to address them with SELinux.

Enforcement levels
SELinux can be implemented in varying modes:

Permissive - SELinux security policy isn't enforced, only logged.
Enforcing - Security policy is enforced and logged. Failures appear as EPERM errors.
This choice is binary and determines whether your policy takes action or merely allows you to gather potential failures. Permissive is especially useful during implementation.

Types, attributes, and rules
Android relies on the Type Enforcement (TE) component of SELinux for its policy. It means that all objects (such as, file, process or socket) have a type associated with them. For instance, by default, an app has the type untrusted_app. For a process, its type is also known as its domain. It's possible to annotate a type with one or many attributes. Attributes are useful to refer to multiple types at the same time.

Objects are mapped to classes (for example, a file, a directory, a symbolic link, a socket) and the different kinds of access for each class are represented by permissions. For instance, the permission open exists for the class file. While types and attributes are regularly updated as part of the Android SELinux policy, permissions and classes are statically defined and rarely updated as part of a new Linux release.

A policy rule comes in the form: allow source target:class permissions; where:

Source - The type (or attribute) of the subject of the rule. Who is requesting the access?
Target - The type (or attribute) of the object. To what is the access requested?
Class - The kind of object (for example, file, socket) being accessed.
Permissions - The operation (or set of operations) (efor example, read, write) being performed.
An example of a rule is:

allow untrusted_app app_data_file:file { read write };
This says that apps are allowed to read and write files labeled app_data_file. There exist other types for apps. For instances, isolated_app is used for app services with isolatedProcess=true in their manifest. Instead of repeating the rule for both types, Android uses an attribute named appdomain for all the types that covers apps:

Associate the attribute appdomain with the type untrusted_app.
typeattribute untrusted_app appdomain;

Associate the attribute appdomain with the type isolated_app.
typeattribute isolated_app appdomain;

allow appdomain app_data_file:file { read write };
When a rule is written that specifies an attribute name, that name is automatically expanded to the list of domains or types associated with the attribute. Some notable attributes are:

domain - attribute associated with all process types,
file_type - attribute associated with all file types.
Macros
For file access in particular, there are many kinds of permission to consider. For instance, the read permission isn't enough to open the file or call stat on it. To simplify the rule definition, Android provides a set of macros to handle the most common cases. For example, in order to include the missing permissions such as open, the rule above could be rewritten as:

allow appdomain app_data_file:file rw_file_perms;
See the global_macros and te_macros files for more example of useful macros. Macros should be used whenever possible to help reduce the likelihood of failures due to denials on related permissions.

Once a type is defined, it needs to be associated with the file or process it represents. See Implementing SELinux for more details on how this association is done. For further information on rules, see the SELinux Notebook.

Security context and categories
When debugging SELinux policies or labelling files (using file_contexts or when ing ls -Z), you might come across a security context (also known as a label). For example: u:r:untrusted_app:s0:c15,c256,c513,c768. A security context has the format: user:role:type:sensitivity[:categories]. You can usually ignore the user, role and sensitivity fields of a context (see Specificity). The type field is explained in the previous section. categories are part of the Multi-Level Security (MLS) support in SELinux. In Android 12 and higher, categories are used to:

Isolate the app data from access by another app,
Isolate the app data from one physical user to another.
Specificity
Android doesn't use all the features provided by SELinux. When reading external documentation, keep these points in mind:

The majority of the policies in AOSP are defined using the Kernel Policy Language. There are some exceptions for using Common Intermediate Language (CIL).
SELinux users aren't used. The only user defined is u. When necessary, physical users are represented using the categories field of a security context.
SELinux roles and Role-Based Access Control (RBAC) aren't used. Two default roles are defined and used: r for subjects and object_r for objects.
SELinux sensitivities aren't used. The default s0 sensitivity is always set.
SELinux booleans aren't used. When the policy is built for a device, it doesn't depend on the state of the device. This simplifies the auditing and debugging of policies.

Great! You've shared an extensive overview of **SELinux concepts in Android**. Let me now explain this in **simpler, well-structured language** — perfect for revision, interviews, or hands-on engineering.

🔐 What is SELinux in Android?

Security-Enhanced Linux (SELinux) is a **Mandatory Access Control (MAC)** system used by Android to tightly control **what processes can access what resources**, even if they're root.

🧠 Why?

To prevent:

- App data leaks
- Root or system process compromise
- Malware from misusing permissions

🆚 MAC vs DAC

Concept	DAC (Discretionary Access Control)	MAC (SELinux)
Who decides?	File owner (**`chmod`**, **`chown`**)	Centralized policy (SELinux)
Can be bypassed by root?	✅ Yes	❌ No
Example	rw-r--r-- permissions	allow **`untrusted_app`** to access **`file`**

🚦 SELinux Modes

Mode	Description
Permissive	SELinux logs violations but doesn't block actions
Enforcing	SELinux logs **and** blocks actions not allowed by the policy

🛠️ Use **permissive** for development & debugging.
🔐 Android ships in **enforcing** mode.

🔑 Core SELinux Concepts

1. Types and Domains

- **Type**: Label assigned to a resource (file, socket, process)
- **Domain**: Special name for the type assigned to a process

🔸 Example:

```c
type untrusted_app;
```

2. Attributes

- Group multiple types under a single name to write fewer rules

🔸 Example:

```c
typeattribute untrusted_app appdomain;
typeattribute isolated_app appdomain;
```

Now **`allow appdomain something...`** applies to both.

📜 Rules: The `allow` Statement

Format:

```c
allow <source> <target>:<class> <permissions>;
```

Meaning:

- **source**: Who is accessing? (e.g., **`untrusted_app`**)
- **target**: What is being accessed? (e.g., **`app_data_file`**)
- **class**: What kind of object is it? (e.g., **`file`**, **`socket`**)
- **permissions**: What actions? (e.g., **`read`**, **`write`**, **`open`**)

🔸 Example:

```c
allow untrusted_app app_data_file:file { read write open };
```

🔁 Macros: Shortcuts for Common Rules

Instead of writing many permissions, you can use macros:

```c
allow appdomain app_data_file:file rw_file_perms;
```

✅ Macros expand to many safe defaults — use them to reduce errors.

🔖 Security Contexts (Labels)

Format:

```ruby
u:r:<type>:s0[:categories]
```

Where:

- **`u`** → SELinux user (Android always uses **`u`**)
- **`r`** → Role (Android uses **`r`** for processes)
- **`type`** → Domain/type (this is the most important part!)
- **`s0`** → Sensitivity (Android always uses **`s0`**)
- **`categories`** → Used for **app/user isolation**

🔸 Example:

```makefile
u:r:untrusted_app:s0:c512,c768
```

📦 What Android Uses (and Ignores)

SELinux Feature	Android Status	Notes
Users	✅ **u** only	No multiple SELinux users
Roles (RBAC)	❌ Not used	Always **`r`** and **`object_r`**
Sensitivities (MLS)	❌ Not used	Always **`s0`**
Categories	✅ Used for isolation	E.g., separate app/user storage
Booleans	❌ Not used	No runtime switching of rules

📅 SELinux Timeline in Android

Version	SELinux Status
4.3	Introduced, permissive
4.4	Partial enforcement (e.g., zygote, vold, netd)
5.0	Fully enforced (all domains)
6.0	Tightened policies, IOCTL filtering, /proc access
7.0	Media stack sandboxing
8.0 (Treble)	Platform-vendor policy separation

🛠️ Policy Implementation in Android

- Policies written in **`.te`**, **`.fc`**, **`Android.mk`** / **`Android.bp`**
- Labels assigned via **`file_contexts`**

- Compiled into **`sepolicy`** binary at build time
- Stored in **`/sepolicy`**, loaded by kernel at boot

🔍 Tools for Debugging

Tool	Use
`logcat	grep avc`
`audit2allow`	Suggest allow rules from denials
`ls -Z`	Show SELinux context of files
`ps -Z`	Show SELinux domain of processes

Would you like a **flowchart of how SELinux checks access**, or a **real denial example and how to fix it**?

Let me know and I'll generate it for you.

45. domain vs type vs context vs label
explain in simple words with example

Great question! These terms — **domain**, **type**, **context**, and **label** — are core to understanding SELinux, and they can be confusing at first. Let's break them down in **simple words** with a clear **example**. ✅

🎯 What is a Label?

A **label** is a full SELinux security tag assigned to a **file** or **process**. It's also called a **security context**.

🧱 Format of a Label / Context:

```pgsql
user:role:type:level
```

🔸 Example (Android):

```makefile
u:r:untrusted_app:s0:c512,c768
```

This whole thing is a **label** (aka **security context**).

🧠 What is a Context?

Context = **Label**

They're the **same**. It just depends on whether you say:

- "the file's **context** is..."
- or "the file is **labeled** with..."

💡 **Just remember:**
➡️ A context is the **full SELinux label** with 4 parts.

🏷️ What is a Type?

The **type** is the **3rd part** of the context (label):

```bash
u:r:📌type:s0
```

This is the **most important part** to SELinux! It determines:

- What a process **can do**
- What a file **can be used for**

🔸 Example:

- Process label: `u:r:untrusted_app:s0` → type = `untrusted_app`
- File label: `u:object_r:app_data_file:s0` → type = `app_data_file`

🚀 What is a Domain?

A **domain** is just a **type** assigned to a **process**.

You can say:

- "Process runs in domain **`system_server`**"
- Meaning: its **type** is **`system_server`**

🔁 So:
Process Type = Domain
File Type = File Type

📦 Example Comparison

Concept	For File	For Process (App)
Label/Context	**`u:object_r:app_data_file:s0`**	**`u:r:untrusted_app:s0`**
Type	**`app_data_file`**	**`untrusted_app`**
Domain	❌ (not used)	✅ **`untrusted_app`** is domain

🔄 How It Works Together

🧪 Rule Example:

```c
allow untrusted_app app_data_file:file { read write };
```

- **`untrusted_app`** → **domain** (process type)
- **`app_data_file`** → **file type**
- **`file`** → class
- **`read write`** → permissions

➡️ This means:
Processes labeled as `untrusted_app` can read/write files labeled as `app_data_file`.

🧠 TL;DR

Term	Meaning	Example
Label or **Context**	Full SELinux tag (for file or process)	**`u:r:system_server:s0`**
Type	The key field in a label	**`system_server`**, **`app_data_file`**
Domain	The type assigned to a process	**`untrusted_app`**, **`vold`**, **`init`**

Let me know if you want a **diagram showing this visually** — I can generate one for you!

46. 2. Attributes
Group multiple types under a single name to write fewer rules

🔸 Example:

c
Copy
Edit
typeattribute untrusted_app appdomain;
typeattribute isolated_app appdomain;
Now allow appdomain something... applies to both.

explain wiht example

Absolutely! Let's break down **attributes** in SELinux using a **realistic Android example**, so you clearly understand why and how they're used.

🎯 What is an SELinux Attribute?

An **attribute** is like a **tag** or **group name** that can be assigned to **multiple types**.

This helps you write **less repetitive policy**.

🧠 Why Use Attributes?

Imagine you have:

- 5 different **app types** (`untrusted_app`, `isolated_app`, `platform_app`, etc.)
- All of them need to access the same file type.

❌ Without attributes:

```c
allow untrusted_app app_data_file:file { read write };
allow isolated_app app_data_file:file { read write };
allow platform_app app_data_file:file { read write };
```

✅ With attributes:

```c
typeattribute untrusted_app appdomain;
typeattribute isolated_app appdomain;
typeattribute platform_app appdomain;

allow appdomain app_data_file:file { read write };
```

Now, **one rule** covers them all!

📦 Realistic Android Example

Let's say:

- You have 3 app types: `untrusted_app`, `isolated_app`, `platform_app`
- All should access `app_data_file`

1. ✅ Assign Attribute

```c
typeattribute untrusted_app appdomain;
typeattribute isolated_app appdomain;
```

```
typeattribute platform_app appdomain;
```

> You're saying: "These 3 types belong to the **appdomain** group."

---

2. ✅ Write One Allow Rule Using the Attribute

```c
allow appdomain app_data_file:file { read write };
```

> This applies to **all types** that have the **`appdomain`** attribute.

🔁 Behind the Scenes

When SELinux compiles this, it expands:

```c
allow appdomain app_data_file:file { read write };
```

into:

```c
allow untrusted_app app_data_file:file { read write };
allow isolated_app app_data_file:file { read write };
allow platform_app app_data_file:file { read write };
```

✅ Same result, cleaner code!

🧠 TL;DR

Term	Meaning
`typeattribute`	Assigns an attribute to a type
`appdomain`	Attribute that groups multiple app types
`allow appdomain ...`	Rule applies to **all** grouped types

Let me know if you want a **diagram or table** to help visualize how attributes expand — happy to generate it!

47. steps for this

Here are the **clear, step-by-step instructions** to define and use **SELinux attributes** in Android SELinux policy:

✅ Goal:

Use an **attribute** (e.g., `appdomain`) to group types like `untrusted_app`, `isolated_app`, and then write one allow rule for all.

🔧 Steps:

🥇 Step 1: Define the Attribute (Optional if already defined)

In most Android SELinux setups, common attributes like `appdomain`, `domain`, etc., are **already defined** in AOSP policy.

But if you're making a **custom attribute**, declare it in your `.te` or a shared `.te` file:

```c
attribute my_custom_group;
```

> ✅ This tells SELinux, "I'm making a new attribute name."

🥈 Step 2: Assign Types to the Attribute

In your `.te` file(s), associate **existing types** with that attribute.

Example: Group app domains under `appdomain`

```c
typeattribute untrusted_app appdomain;
typeattribute isolated_app appdomain;
typeattribute platform_app appdomain;
```

> ✅ These types now **belong to** the `appdomain` attribute.

🥉 Step 3: Write Rules Using the Attribute

Instead of repeating the same rule for each type, write it **once** using the attribute:

```c
allow appdomain app_data_file:file rw_file_perms;
```

> ✅ This expands into separate `allow` rules for each type in `appdomain`.

🧪 Step 4: Build the Policy

After adding these to the correct `.te` files:

- Make sure your `Android.mk` or `Android.bp` includes the policy directory
- Run the full AOSP build (e.g., `make bootimage`, `make sepolicy`)

🧾 File Summary:

File	What you do in it
`untrusted_app.te`	`typeattribute untrusted_app appdomain;`
`isolated_app.te`	`typeattribute isolated_app appdomain;`

File	What you do in it
`platform_app.te`	`typeattribute platform_app appdomain;`
`some_policy.te`	`allow appdomain app_data_file:file rw_file_perms;`
`global_macros`	Contains macro definitions like `rw_file_perms`

💡 Helpful Tip:

Use **`audit2allow`** when debugging AVC denials, and you'll often see multiple rules for similar types — that's a great case to introduce attributes.

Let me know if you want to:

- See this in a real AOSP device policy structure
- Add your own custom attribute in a device-specific policy

Happy to help!

48. Implement SELinux

bookmark_border
SELinux is set up to default-deny, which means that every single access for which it has a hook in the kernel must be explicitly allowed by policy. This means a policy file is comprised of a large amount of information regarding rules, types, classes, permissions, and more. A full consideration of SELinux is out of the scope of this document, but an understanding of how to write policy rules is now essential when bringing up new Android devices. There is a great deal of information available regarding SELinux already. See Supporting documentation for suggested resources.

Key files
To enable SELinux, integrate the latest Android kernel and then incorporate the files found in the system/sepolicy directory. When compiled, those files comprise the SELinux kernel security policy and cover the upstream Android operating system.

In general, you shouldn't modify the system/sepolicy files directly. Instead, add or edit your own device-specific policy files in the /device/manufacturer/device-name/sepolicy directory. In Android 8.0 and higher, the changes you make to these files should only affect policy in your vendor directory. For more details on separation of public sepolicy in Android 8.0 and higher, see Customizing SEPolicy in Android 8.0+. Regardless of Android version, you're still modifying these files:

Policy files
Files that end with *.te are SELinux policy source files, which define domains and their labels. You may need to create new policy files in /device/manufacturer/device-name/sepolicy, but you should try to update existing files where possible.

Context files
Context files are where you specify labels for your objects.

file_contexts assigns labels to files and is used by various userspace components. As you create new policies, create or update this file to assign new labels to files. To apply new file_contexts, rebuild the filesystem image or run restorecon on the file to be relabeled. On upgrades, changes to file_contexts are automatically applied to the system and userdata partitions as part of the upgrade. Changes can also be automatically applied on upgrade to other partitions by adding restorecon_recursive calls to your init.board.rc file after the partition has been mounted read-write.
genfs_contexts assigns labels to filesystems, such as proc or vfat that don't support extended attributes. This configuration is loaded as part of the kernel policy but changes might not take effect for in-core inodes, requiring a reboot or unmounting and re-mounting the filesystem to fully apply the change. Specific labels may also be assigned to specific mounts, such as vfat using the context=mount option.
property_contexts assigns labels to Android system properties to control what processes can set them. This configuration is read by the init process during startup.
service_contexts assigns labels to Android binder services to control what processes can add (register) and find (lookup) a binder reference for the service. This configuration is read by the servicemanager process during startup.
seapp_contexts assigns labels to app processes and /data/data directories. This configuration is read by the zygote process on each app launch and by installd during startup.
mac_permissions.xml assigns a seinfo tag to apps based on their signature and optionally their package name. The seinfo tag can then be used as a key in the seapp_contexts file to assign a specific label to all apps with that seinfo tag. This configuration is read by system_server during startup.
keystore2_key_contexts assigns labels to Keystore 2 namespaces. These namespace are enforced by the keystore2 daemon. Keystore has always provided UID/AID based namespaces. Keystore 2 additionally enforces sepolicy defined namespaces. A detailed description of the format and conventions of this file can be found here.
BoardConfig.mk makefile
After editing or adding policy and context files, update your /device/manufacturer/device-name/BoardConfig.mk makefile to reference the sepolicy subdirectory and each new policy file. For more information about the BOARD_SEPOLICY variables, see system/sepolicy/README file.

```
BOARD_SEPOLICY_DIRS += \
    <root>/device/manufacturer/device-name/sepolicy

BOARD_SEPOLICY_UNION += \
    genfs_contexts \
    file_contexts \
    sepolicy.te
```

After rebuilding, your device is enabled with SELinux. You can now either customize your SELinux policies to accommodate your own additions to the Android operating system as described in Customization or verify your existing setup as covered in Validation.

When the new policy files and BoardConfig.mk updates are in place, the new policy settings are automatically built into the final kernel policy file. For more information about how sepolicy is built on the device, see Building sepolicy.

Implementation
Note: audit2allow isn't provided as part of AOSP anymore. Use the version from your host (package 'policycoreutils-python-utils' on Debian and

Ubuntu).
To get started with SELinux:

Enable SELinux in the kernel: CONFIG_SECURITY_SELINUX=y
Change the kernel_cmdline or bootconfig parameter so that:

BOARD_KERNEL_CMDLINE := androidboot.selinux=permissive
or

BOARD_BOOTCONFIG := androidboot.selinux=permissive
This is only for initial development of policy for the device. After you have an initial bootstrap policy, remove this parameter so your device is enforcing or it fails CTS.
Boot up the system in permissive and see what denials are encountered on boot:
On Ubuntu 14.04 or newer:

adb shell su -c dmesg | grep denied | audit2allow -p out/target/product/BOARD/root/sepolicy
On Ubuntu 12.04:

adb pull /sys/fs/selinux/policy
adb logcat -b all | audit2allow -p policy
Evaluate the output for warnings that resemble init: Warning! Service name needs a SELinux domain defined; please fix! See Validation for instructions and tools.
Identify devices, and other new files that need labeling.
Use existing or new labels for your objects. Look at the *_contexts files to see how things were previously labeled and use knowledge of the label meanings to assign a new one. Ideally, this is an existing label that fits into policy, but sometimes a new label is needed, and rules for access to that label are needed. Add your labels to the appropriate context files.
Identify domains/processes that should have their own security domains. You likely need to write a completely new policy for each. All services spawned from init, for instance, should have their own. The following commands help reveal those that remain running (but ALL services need such a treatment):

adb shell su -c ps -Z | grep init

adb shell su -c dmesg | grep 'avc: '
Review init.device.rc to identify any domains that don't have a domain type. Give them a domain early in your development process to avoid adding rules to init or otherwise confusing init accesses with ones that are in their own policy.
Set up BOARD_CONFIG.mk to use BOARD_SEPOLICY_* variables. See the README in system/sepolicy for details on setting this up.
Examine the init.device.rc and fstab.device file and make sure every use of mount corresponds to a properly labeled filesystem or that a context= mount option is specified.
Go through each denial and create SELinux policy to properly handle each. See the examples in Customization.
You should start with the policies in the AOSP and then build upon them for your own customizations. For more information about policy strategy and a closer look at some of these steps, see Writing SELinux Policy.

Use cases
Here are specific examples of exploits to consider when crafting your own software and associated SELinux policies:

Symlinks: Because symlinks appear as files, they are often read as files, which can lead to exploits. For instance, some privileged components, such as init, change the permissions of certain files, sometimes to be excessively open.

Attackers might then replace those files with symlinks to code they control, allowing the attacker to overwrite arbitrary files. But if you know your app never traverses a symlink, you can prohibit it from doing so with SELinux.

System files: Consider the class of system files that should be modified only by the system server. Still, since netd, init, and vold run as root, they can access those system files. So if netd became compromised, it could compromise those files and potentially the system server itself.

With SELinux, you can identify those files as system server data files. Therefore, the only domain that has read/write access to them is system server. Even if netd became compromised, it couldn't switch domains to the system server domain and access those system files although it runs as root.

App data: Another example is the class of functions that must run as root but shouldn't get to access app data. This is incredibly useful as wide-ranging assertions can be made, such as certain domains unrelated to app data being prohibited from accessing the internet.

setattr: For commands such as chmod and chown, you could identify the set of files where the associated domain can conduct setattr. Anything outside of that could be prohibited from these changes, even by root. So an app might run chmod and chown against those labeled app_data_files but not shell_data_files or system_data_files.

Here is a **simple step-by-step guide** to help you understand how to **implement SELinux in AOSP** for a new Android device or custom service.

🧱 SELinux in Android – Implementation Steps

🔧 1. Enable SELinux in Kernel

In your kernel `.config` file (or Kconfig):

```c
CONFIG_SECURITY_SELINUX=y
CONFIG_SECURITY_SELINUX_BOOTPARAM=y
```

🧾 2. Set Boot Parameter

During initial bring-up, make SELinux permissive to observe denials.

In `BoardConfig.mk`:

```make
BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
```

or (for bootconfig devices):

```make
BOARD_BOOTCONFIG += androidboot.selinux=permissive
```

🔁 **Later remove this** to pass CTS (must be enforcing in production).

📁 3. Create Device-Specific SELinux Directory

```bash
/device/<vendor>/<device>/sepolicy/
```

🧑‍💻 4. Write Policy Files

You'll create/edit these:

File	Purpose
`*.te`	Define domains (process types) and access rules
`file_contexts`	Assign labels to files and dirs
`genfs_contexts`	Assign labels to filesystems without xattrs (e.g., vfat)
`property_contexts`	Label system properties (`setprop`, `getprop`)
`service_contexts`	Label binder services
`seapp_contexts`	Label apps and `/data/data/<package>` dirs
`mac_permissions.xml`	Link app signature/package to `seinfo` tags
`keystore2_key_contexts`	Label keystore2 namespaces

🧩 5. Add Policy to `BoardConfig.mk`

In your device's `BoardConfig.mk`:

```make
BOARD_SEPOLICY_DIRS += device/<vendor>/<device>/sepolicy
```

Optionally add specific files:

```make
BOARD_SEPOLICY_UNION += \
    file_contexts \
    genfs_contexts \
    my_custom.te
```

🔄 6. Boot in Permissive & Capture AVC Denials

Option A: Modern Ubuntu

```bash
adb shell su -c dmesg | grep denied | audit2allow -p out/target/product/<device>/root/sepolicy
```

Option B: Manually

```bash
adb pull /sys/fs/selinux/policy
adb logcat -b all | audit2allow -p policy
```

📌 **`audit2allow`** is no longer in AOSP. Use the one on host system:

```bash
sudo apt install policycoreutils-python-utils
```

🔍 7. Analyze Denials and Add Rules

For each **`avc: denied`**:

1. Identify source and target domains/types
2. Decide if the access is valid
3. Add **`allow`** rule in the relevant **`.te`** file

Example:

```c
allow my_daemon my_log_file:file rw_file_perms;
```

Use macros like **`rw_file_perms`**, **`get_prop`**, etc.

🧪 8. Label Files Using `file_contexts`

Example:

```c
/vendor/bin/mydaemon   u:object_r:mydaemon_exec:s0
```

Rebuild system/vendor image or run:

```bash
adb shell restorecon -v /vendor/bin/mydaemon
```

🧠 9. Assign New Domain to Service

In **`init.<device>.rc`** or **`mydaemon.rc`**:

```rc
service mydaemon /vendor/bin/mydaemon
    seclabel u:r:mydaemon:s0
```

If no **`seclabel`**, it defaults to **`init`** — not good!

🗃️ 10. Customize `seapp_contexts` and `mac_permissions.xml` (For apps only)

- Assign seinfo tags to signed apps
- Use these to set app labels

📦 11. Build & Flash

Rebuild and flash the system:

```bash
source build/envsetup.sh
lunch <target>
make bootimage systemimage vendorimage
```

🧪 12. Switch to Enforcing

Once policies are ready and clean:

```make
# REMOVE
# BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
```

Boot into enforcing mode.

✅ Result

Your device now runs with full SELinux **Mandatory Access Control**, and:

- Services are sandboxed by domains
- Files have labeled access
- Root is not enough to bypass access restrictions

📌 Example Use Cases

Case	How SELinux Helps
init writes wrong file	Block it with domain rules and file contexts
compromised netd tries to access system logs	Denied by **`netd`** domain policy
root chmod app data	Only allowed if labeled **`app_data_file`** and **`setattr`** is permitted

---

Would you like a **flowchart** for these steps, or a working **demo example of a new service with SELinux setup**?

Let me know and I'll generate it!

49. ↪
a working **demo example of a new service with SELinux setup**?
yes

Great! Here's a **working demo example** of adding a **custom service** in Android with **SELinux setup**, step-by-step. This covers:

- Adding a new binary (`mydaemon`)
- Defining an `init` service
- Adding SELinux domain
- Labeling files and service
- Building and testing

✅ Demo Goal

We'll create a new native service:

```bash
/vendor/bin/mydaemon
```

It will:

- Run as a background service
- Use its own SELinux domain `mydaemon`
- Access a log file: `/data/vendor/mydaemon/log.txt`

🧱 1. Create Native Binary

`mydaemon.cpp` (in `device/<vendor>/<device>/mydaemon/`)

```cpp
#include <unistd.h>
#include <fstream>

int main() {
    std::ofstream logfile("/data/vendor/mydaemon/log.txt", std::ios::app);
    logfile << "mydaemon started with PID " << getpid() << "\n";
    logfile.close();
    while (true) sleep(60);
    return 0;
}
```

🛠️ 2. Add Android.mk

Create `Android.mk` in same directory:

```make
LOCAL_PATH := $(call my-dir)

include $(CLEAR_VARS)
LOCAL_MODULE := mydaemon
LOCAL_SRC_FILES := mydaemon.cpp
LOCAL_MODULE_TAGS := optional
LOCAL_MODULE_PATH := $(TARGET_OUT_VENDOR)/bin
LOCAL_MODULE_CLASS := EXECUTABLES
include $(BUILD_EXECUTABLE)
```

📋 3. Add Init `.rc` File

File: `device/<vendor>/<device>/mydaemon.rc`

```rc
service mydaemon /vendor/bin/mydaemon
    class late_start
    user system
    group system
    seclabel u:r:mydaemon:s0
    oneshot
```

📂 4. Copy `.rc` File

In `device.mk`:

```make
PRODUCT_COPY_FILES += \
    device/<vendor>/<device>/mydaemon.rc:$(TARGET_COPY_OUT_VENDOR)/etc/init/mydaemon.rc
```

🔐 5. Add SELinux Policy

A. Domain Definition – `mydaemon.te`

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

init_daemon_domain(mydaemon)

allow mydaemon self:process { fork sigchld sigkill sigstop signull signal };
allow mydaemon mydaemon_exec:file { read execute open };

# Log file access
type mydaemon_log, file_type, data_file_type;
allow mydaemon mydaemon_log:file { open read write append create };
```

B. File Contexts – `file_contexts`

```c
/vendor/bin/mydaemon           u:object_r:mydaemon_exec:s0
/data/vendor/mydaemon(/.*)?    u:object_r:mydaemon_log:s0
```

C. Service Context – `service_contexts`

```c
mydaemon                    u:object_r:mydaemon_service:s0
```

🧩 6. Hook into BoardConfig.mk

```make
BOARD_SEPOLICY_DIRS += device/<vendor>/<device>/sepolicy
```

🧪 7. Build and Test

Rebuild:

```bash
source build/envsetup.sh
lunch <device>
make bootimage vendorimage systemimage
```

Flash and reboot the device.

🔍 8. Check It Works

```bash
adb shell
ps -Z | grep mydaemon
ls -Z /vendor/bin/mydaemon
ls -Z /data/vendor/mydaemon/log.txt
logcat | grep mydaemon
```

If running in permissive mode:

```bash
adb shell dmesg | grep avc
```

Use `audit2allow` to resolve denials.

📈 Summary

File	Purpose
`mydaemon.cpp`	Simple binary writing to log
`mydaemon.rc`	Starts daemon via init
`mydaemon.te`	Declares SELinux domain and rules
`file_contexts`	Maps file paths to labels
`service_contexts`	Labels the binder service (if any)

📥 Optional: Want This as a GitHub repo or image diagram?

Let me know if you'd like:

- A **GitHub-ready project zip**
- A **diagram** showing this flow visually
- Or full `Android.bp` alternative instead of `Android.mk`

Happy to generate those too!

50. ↪
🔧 1. **Enable SELinux in Kernel** In your kernel `.config` file (or Kconfig): c CopyEdit `CONFIG_SECURITY_SELINUX=y CONFIG_SECURITY_SELINUX_BOOTPARAM=y`
explain this and why we mentioned here

Great question!

Let's break this down clearly and **explain what those kernel configs do** and **why they're required** when implementing SELinux in Android.

🔧 `CONFIG_SECURITY_SELINUX=y`

✅ What it means:

This tells the **Linux kernel to include SELinux support**.

Without this line, the kernel:

- Will **not compile in** the SELinux module
- Cannot load any SELinux policy
- Cannot enforce access control rules

🧠 Why it's needed:

SELinux works via **hooks** inside the Linux kernel (through the **LSM — Linux Security Modules** framework). If this option is not set, those hooks are **disabled**, and you lose the ability to enforce or log SELinux rules.

> 💡 Think of it as:
> *"Enable the SELinux engine inside the kernel."*

🔧 `CONFIG_SECURITY_SELINUX_BOOTPARAM=y`

✅ What it means:

This allows **boot-time control** over SELinux behavior via a kernel boot parameter:

```bash
androidboot.selinux=permissive
```

🧠 Why it's needed:

- When you're **developing** SELinux policy, you start in **permissive mode** to observe violations.
- This flag allows you to **switch between permissive and enforcing** using boot arguments — without rebuilding the kernel.

🔄 Example use:

```make
# In BoardConfig.mk
BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
```

This will boot the system in **permissive mode**, where:

- Access violations are **logged**, but **not blocked**.
- You can use tools like **`audit2allow`** to generate rules based on logs.

Later, for production:

```make
# Remove or change to enforcing
BOARD_KERNEL_CMDLINE += androidboot.selinux=enforcing
```

✅ Summary Table

Config	What It Does	Required?	Why?
`CONFIG_SECURITY_SELINUX=y`	Enables SELinux engine in the kernel	✅ Yes	Without this, SELinux is dead
`CONFIG_SECURITY_SELINUX_BOOTPARAM=y`	Lets you switch SELinux mode at boot time	✅ Yes	Makes development easier

🚦 In Short:

Without:

```c
CONFIG_SECURITY_SELINUX=y
```

You cannot use SELinux at all.

Without:

```c
CONFIG_SECURITY_SELINUX_BOOTPARAM=y
```

You're stuck with **whatever SELinux mode** was compiled into the kernel — no flexibility during development.

Let me know if you'd like a diagram showing this in boot flow — happy to generate one!

51. ↪
BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
what this statement

Great — let's explain:

📌 Statement:

```make
BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
```

✅ What it does:

This line **adds a boot argument** to the kernel command line for your Android device:

```ini
androidboot.selinux=permissive
```

This tells the **Android kernel and init process** to boot the system with **SELinux in permissive mode**.

🧠 Why we use it:

- **Permissive mode**: SELinux **logs** all policy violations but **does not block** anything.
- This is useful during:
  - First-time bring-up of a device
  - Writing new SELinux policies
  - Debugging AVC (access vector cache) denials

Once your policies are stable, you remove this line to enable **enforcing mode**.

🔄 Example Flow:

Phase	`BOARD_KERNEL_CMDLINE` Setting	Result
Development	`androidboot.selinux=permissive`	SELinux logs denials only
Production	(omit or use `androidboot.selinux=enforcing`)	SELinux blocks all denied actions

---

📦 Where it's used:

This line goes in:

```php-template
device/<vendor>/<device>/BoardConfig.mk
```

So that the bootloader passes this argument during kernel boot.

✅ Summary:

Field	Meaning
`BOARD_KERNEL_CMDLINE`	Boot-time kernel arguments

Field	Meaning
`androidboot.selinux`	Sets SELinux mode
`permissive`	Don't enforce policy, just log denials

Let me know if you want a diagram of this included in Android boot flow (e.g., bootloader → kernel → init → SELinux init).

52. ↪
🧑‍💻 4. **Write Policy Files** You'll create/edit these: | File | Purpose | | --- | --- | | `*.te` | Define domains (process types) and access rules | | `file_contexts` | Assign labels to files and dirs | | `genfs_contexts` | Assign labels to filesystems without xattrs (e.g., vfat) | | `property_contexts` | Label system properties (`setprop`, `getprop`) | | `service_contexts` | Label binder services | | `seapp_contexts` | Label apps and `/data/data/<package>` dirs | | `mac_permissions.xml` | Link app signature/package to `seinfo` tags | | `keystore2_key_contexts` | Label keystore2 namespaces |
explain in details

Perfect. Let's go step-by-step and **explain each of these SELinux policy files in detail** — what they are, where they live, and **why you need them** when working with Android SELinux.

🧑‍💻 1. `*.te` – Type Enforcement Files

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/<name>.te
```

📖 Purpose:

These files define **SELinux types** and **access rules** — especially:

- **Domains** for processes (e.g., `type mydaemon, domain`)
- **What resources a process can access** (files, sockets, properties)
- **Allow rules** (e.g., `allow mydaemon myfile:file { read write };`)

🧠 Think of it as:

> "Who can talk to what, and how?"

📁 2. `file_contexts` – File and Directory Labeling

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/file_contexts
```

📖 Purpose:

Assigns **SELinux labels** (aka contexts) to **file paths**, like:

```c
/vendor/bin/mydaemon      u:object_r:mydaemon_exec:s0
/data/vendor/mydaemon(/.*)?   u:object_r:mydaemon_data_file:s0
```

This tells SELinux:

> "When you see this path, assign this context."

🛠️ Used by:

- `restorecon` to relabel files
- `init` during startup
- File system creation

💽 3. `genfs_contexts` – Non-Ext4 File System Labeling

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/genfs_contexts
```

📖 Purpose:

Assigns labels to **file systems that don't support extended attributes** — e.g.,:

- `vfat`
- `proc`
- `sysfs`

Example:

```c
genfscon vfat /mnt/media_rw u:object_r:media_rw_data_file:s0
```

📢 Required when:

- Mounting USB, SD cards, or FAT partitions.

🧾 4. `property_contexts` – System Properties Labeling

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/property_contexts
```

📖 Purpose:

Controls which domains can `setprop`/`getprop`.

Example:

```c
persist.my.feature.enabled      u:object_r:mydaemon_prop:s0
```

Used to enforce:

> "Only `mydaemon` can set this property."

🔌 5. `service_contexts` – Binder Service Labeling

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/service_contexts
```

📖 Purpose:

Maps Android binder service names to SELinux labels.

Example:

```c
mydaemon          u:object_r:mydaemon_service:s0
```

🛠️ Used by:

- **`servicemanager`** to enforce which process can:
  - **Register** this service
  - **Call** this service

🚨 Without this:

- Your binder service may not be found or started properly.

📱 6. `seapp_contexts` – App Process & Data Labeling

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/seapp_contexts
```

📖 Purpose:

Defines how **apps and their `/data/data/<package>` folders** are labeled.

Uses info like:

- **`seinfo`** (from **`mac_permissions.xml`**)
- UID ranges
- UserId (multi-user)

Example:

```c
user=system seinfo=platform name=com.android.phone domain=platform_app type=platform_app_data_file
```

🧠 This enforces:

> "This app runs in this domain and owns these data files."

📦 7. `mac_permissions.xml` – App Signature Mapping

📌 Location:

```swift
system/sepolicy/private/mac_permissions.xml`
```

Or overlay a copy in:

```php-template
device/<vendor>/<device>/sepolicy/
```

📖 Purpose:

Maps apps (by **signing key** or package name) to a **`seinfo` tag**, like:

```xml
<default>
    <seinfo value="default"/>
</default>

<signature signature="@platform" >
    <package name="com.android.phone" seinfo="platform"/>
</signature>
```

Then **`seinfo`** is used in **`seapp_contexts`**.

🔐 Enforces:

> "All apps signed with this cert belong to this SELinux group."

🔐 8. `keystore2_key_contexts` – Keystore 2 Namespaces

📌 Location:

```php-template
device/<vendor>/<device>/sepolicy/keystore2_key_contexts
```

📖 Purpose:

Assign SELinux types to **Keystore 2 key namespaces**.

Example:

```c
key_namespace="mydaemon_ns"  u:object_r:mydaemon_keystore_key:s0
```

This is required for:

- **Labeling and isolating encryption keys**
- **Access control on who can use keys**

🎯 Summary Table

File	Purpose
`*.te`	Define domains, types, allow rules
`file_contexts`	Label files/directories with SELinux context
`genfs_contexts`	Label filesystems (like FAT, procfs) without extended attributes
`property_contexts`	Restrict who can **`setprop`**/**`getprop`**
`service_contexts`	Restrict access to binder services
`seapp_contexts`	Label apps and their data
`mac_permissions.xml`	Map app signature/package to seinfo tags for use in **`seapp_contexts`**
`keystore2_key_contexts`	Label Keystore 2 key namespaces

Would you like:

- A **visual diagram** of this file system and process relationship?
- A **template folder structure** to copy for your own sepolicy?

Let me know!

53. Type Enforcement File

Let's go step-by-step and **explain the Type Enforcement (TE) file** in Android SELinux — in the **simplest way possible**.

📄 What is a TE File?

A **Type Enforcement (TE)** file (ending in `.te`) is where you **define security rules** for a **domain** (process) in SELinux.

🔧 Location:

```php-template
device/<vendor>/<device>/sepolicy/mydaemon.te
```

🔍 What's Inside a TE File?

The `.te` file does **three things**:

Purpose	Example	Meaning
1. Declare domain (type)	`type mydaemon, domain;`	Define a new SELinux domain
2. Allow access rules	`allow mydaemon myfile:file { read write };`	Allow process to access files
3. Use macros for common rules	`init_daemon_domain(mydaemon)`	Apply common settings for system daemons

✅ Example TE File

```c
# 1. Declare this process domain
type mydaemon, domain;

# 2. Declare the executable file type
type mydaemon_exec, exec_type, file_type;

# 3. Tell init: "This is a daemon"
init_daemon_domain(mydaemon)

# 4. Allow basic process actions
allow mydaemon self:process { fork signal sigchld };

# 5. Allow access to its own binary
allow mydaemon mydaemon_exec:file { read open execute };

# 6. Allow writing to a log file
type mydaemon_log, file_type;
allow mydaemon mydaemon_log:file { open write append };
```

🧠 What Each Line Means

🔹 **`type mydaemon, domain;`**

Create a **domain** called **`mydaemon`** — a label for the process.

🔹 **`type mydaemon_exec, exec_type, file_type;`**

Label for the **binary file** (**`/vendor/bin/mydaemon`**).

🔹 **`init_daemon_domain(mydaemon)`**

Shortcut macro that:

- Declares rules to start the daemon

- Allows it to transition from **`init`** to **`mydaemon`** domain

🔹 **`allow ...`**

Defines **who can do what to whom**, for example:

```c
allow A B:C { perms };
```

Means: **Domain `A` is allowed to access object `B` of class `C` with permissions `perms`**

🔐 Why It's Needed

SELinux is **default-deny**:

- Without **`.te`** file, your process **won't be allowed to run or access anything**.
- You must explicitly say: "**`mydaemon`** is allowed to run, read its binary, and write to its log."

🧱 Real World Analogy

SELinux Concept	Real World Analogy
`mydaemon` domain	Job title (e.g., "Bank Teller")
`allow` rules	List of what the job can/can't do
`type`	Label/tag on a person or object

📌 Summary

Field	What It Does
`type A, domain`	Define a process domain
`type B, file_type`	Define a file label
`allow A B:C { X }`	Allow domain A to do action X on object B of class C
`init_daemon_domain(A)`	Macro to simplify creating system daemon rules

Let me know if you'd like:

- A **template** TE file
- Or a **diagram** showing how this interacts with **`init`**, file contexts, and processes.

> 54. ↪
> Allows it to transition from `init` to `mydaemon` domain
> what is this

Great follow-up!

Let's explain **what "transition from `init` to `mydaemon` domain" means — simply and clearly**.

🔁 What is a Domain Transition?

In SELinux, **a domain is a label for a process**. When a process starts, SELinux assigns it a domain **based on which binary was executed** and what policy says.

A **domain transition** is when:

> **One process starts another**, and SELinux moves the new process into a **different domain**.

🧠 Why it matters in Android:

- In Android, **`init`** is the first process to run after the kernel.
- **`init`** is responsible for **starting your service** (e.g., **`mydaemon`**) using an **`.rc`** file.

Example **`mydaemon.rc`**:

```rc
service mydaemon /vendor/bin/mydaemon
    class main
    user system
    group system
    seclabel u:r:mydaemon:s0
```

When **`init`** runs **`/vendor/bin/mydaemon`**, we want the **new process** to run in the **`mydaemon` domain**, not **`init`**.

🔐 Why SELinux cares

Without a proper **transition rule**, the process may:

- **Fail to start**
- Inherit wrong permissions (e.g., **`init`** domain)

So we need SELinux to:

> "Allow **`init`** to start **`/vendor/bin/mydaemon`**, and when it does, assign the **mydaemon domain** to that new process."

✅ This is what this line does:

```c
init_daemon_domain(mydaemon)
```

It's a **macro** (shortcut) that:

1. **Creates a domain transition rule** from **`init`** → **`mydaemon`**
2. Tells SELinux:

   > "When **`init`** executes **`mydaemon_exec`**, assign the new process the **`mydaemon`** domain."

🧱 Real-world analogy:

Actor	Action	SELinux Role
Parent (**`init`**)	Runs a child program	Domain transition
Child (**`mydaemon`**)	Gets own job title / role	Assigned **`mydaemon`** domain

---

🧩 Behind the scenes:

The macro expands to rules like:

```
c

domain_auto_trans(init, mydaemon_exec, mydaemon)
```

Which means:

> If process in **`init`** domain runs **`mydaemon_exec`**, put the new process in **`mydaemon`** domain.

✅ Summary

Concept	Meaning
Domain	SELinux label for a process
Domain Transition	When one process (e.g., **`init`**) starts another in a **new domain**
`init_daemon_domain(X)`	Macro that enables safe domain transition for **`init`**-started daemons
Why It's Needed	Without it, your daemon may not start or get wrong permissions

---

Let me know if you'd like this shown in a diagram — "init → exec → new domain".

55. type mydaemon, domain;

Let’s break down this simple but **important** SELinux line:

🔹 Line:

```c
type mydaemon, domain;
```

🧠 Meaning in Simple Words:

- **`type mydaemon`** → You're **creating a label** (called **`mydaemon`**).
- **`domain`** → You're saying that this **label is for a process** (not a file, not a socket).

So together:

> ✅ You're telling SELinux:
> “Create a **process domain** called **`mydaemon`**.”

📌 Why it's needed:

Every process in Android runs under a **SELinux domain**.
This line defines the domain your process (e.g., a system service, daemon, app) will use.

🔧 When to use it:

If you have a **custom service or daemon**, like:

```bash
/vendor/bin/mydaemon
```

...and you want it to run under its **own SELinux label**, not a default one like **`init`** or **`system_server`**,
→ then you **must define** a new domain like this:

```c
type mydaemon, domain;
```

🧱 Analogy:

SELinux Term	Real World Example
`mydaemon`	Job title = “Security Officer”
`domain`	Says: “This is a job title for people (processes), not for files or buildings.”

✅ What happens next:

This declaration lets you:

- Write **allow rules** for **`mydaemon`**
- Use **domain transitions** from **`init`**
- Label your binary with **`mydaemon_exec`**

- Control what this process can do

🧩 Example of usage:

```c
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

# Transition from init when exec mydaemon binary
init_daemon_domain(mydaemon)

# Allow process to run its own binary
allow mydaemon mydaemon_exec:file { read open execute };
```

Let me know if you'd like this shown in a small **visual flow**, or if you'd like help writing your `.te` file end-to-end.

56. type mydaemon_exec, exec_type, file_type;

Let's break down this line:

🔹 Line:

```c
type mydaemon_exec, exec_type, file_type;
```

✅ Simple Explanation:

This tells SELinux:

> "Define a **label** called **`mydaemon_exec`**, and mark it as a type of **executable file**."

📌 Purpose of Each Word:

Part	Meaning
`type mydaemon_exec`	Declares a **new label** for your daemon binary (like **`/vendor/bin/mydaemon`**)
`exec_type`	Tells SELinux: "This type is an **executable file**"
`file_type`	Says: "This type applies to **files**" (not sockets, processes, etc.)

🔧 Why it's needed:

SELinux needs to:

1. **Label** your binary with a type (**`mydaemon_exec`**)
2. Know that it's an **executable file**
3. Allow **domain transitions** from **`init`** to **`mydaemon`**

Without this, SELinux doesn't know:

- What kind of object **`mydaemon_exec`** is
- Whether it's safe to run it
- Whether it can transition into the right domain

🔁 How it fits in the flow:

1. You label the file in **`file_contexts`**:

   ```c
   /vendor/bin/mydaemon u:object_r:mydaemon_exec:s0
   ```

2. Then in **`mydaemon.te`**, you define:

   ```c
   type mydaemon_exec, exec_type, file_type;
   type mydaemon, domain;
   init_daemon_domain(mydaemon)
   ```

🧠 What this enables:

✅ Domain transition:

> **`init`** executes **`mydaemon_exec`** → SELinux transitions to **`mydaemon`** domain.

✅ Access control:

> You can now allow **`mydaemon`** to access **`mydaemon_exec`**:

```c
allow mydaemon mydaemon_exec:file { read open execute };
```

🧱 Real-world analogy:

SELinux Concept	Real-world Analogy
`mydaemon_exec`	Label on the **binary file**
`exec_type`	Means: "This is an **executable program**"
`file_type`	Means: "This label applies to **a file**"

📌 Summary

Field	Role
`type`	Declares a new SELinux type
`exec_type`	Means it's executable (can cause domain transitions)
`file_type`	Means it labels a file
`mydaemon_exec`	The label to apply to **`/vendor/bin/mydaemon`**

Let me know if you want a **full working example** including **`.rc`**, **`.te`**, and **`file_contexts`**!

57. yes

Perfect! Here's a **complete working example** of integrating a custom daemon with SELinux in Android, including:

1. `init .rc` file
2. `*.te` (Type Enforcement) file
3. `file_contexts` file
4. `BoardConfig.mk` integration
5. Final flow diagram

✅ 1. `init/mydaemon.rc`

Place this in:

```php-template
device/<vendor>/<device>/mydaemon.rc
```

```rc
service mydaemon /vendor/bin/mydaemon
    class main
    user system
    group system
    seclabel u:r:mydaemon:s0
    oneshot
```

✅ 2. `sepolicy/mydaemon.te`

Place this in:

```php-template
device/<vendor>/<device>/sepolicy/mydaemon.te
```

```c
# Define the process domain
type mydaemon, domain;
type mydaemon_exec, exec_type, file_type;

# Allow init to start the daemon and transition to mydaemon domain
init_daemon_domain(mydaemon)

# Allow daemon to execute its own binary
allow mydaemon mydaemon_exec:file { read open execute };

# Allow basic actions
allow mydaemon self:process { fork sigchld sigkill };

# (Optional) Allow log writing
type mydaemon_log, file_type;
allow mydaemon mydaemon_log:file { write append open };
```

✅ 3. `sepolicy/file_contexts`

Place this in:

```php-template
device/<vendor>/<device>/sepolicy/file_contexts
```

```c
/vendor/bin/mydaemon          u:object_r:mydaemon_exec:s0
/vendor/log/mydaemon.log      u:object_r:mydaemon_log:s0
```

✅ 4. `BoardConfig.mk`

Edit your device-specific **`BoardConfig.mk`**:

```php-template
device/<vendor>/<device>/BoardConfig.mk
```

```make
# Add the sepolicy directory
BOARD_SEPOLICY_DIRS += \
    device/<vendor>/<device>/sepolicy
```

If using Android 11+ and **`BOARD_SEPOLICY_VERS`** is defined, make sure your file is included.

✅ 5. `Android.mk` or `.bp` (for binary)

Make sure your daemon binary is built and copied to the right path (**`/vendor/bin/mydaemon`**). Use **`Android.mk`** or **`.bp`** like:

```make
LOCAL_MODULE := mydaemon
LOCAL_SRC_FILES := mydaemon.c
LOCAL_MODULE_CLASS := EXECUTABLES
LOCAL_MODULE_PATH := $(TARGET_OUT_VENDOR)/bin
include $(BUILD_EXECUTABLE)
```

🔄 Final Flow Diagram:

```pgsql
      +--------------------+
      |     init.rc        |
      |--------------------|
      | service mydaemon   |
      | exec: /vendor/bin/mydaemon
      +--------------------+
                |
                v
+------------------------------+
| SELinux file_contexts        |
| - mydaemon_exec              |
| → Label the binary           |
+------------------------------+
                |
                v
+------------------------------+
| SELinux policy (mydaemon.te)|
| - type mydaemon              |
| - type mydaemon_exec         |
| - init_daemon_domain         |
| - allow rules                |
+------------------------------+
                |
                v
+------------------------------+
| Running process              |
| Label: u:r:mydaemon:s0       |
| Accesses only what policy    |
| allows                       |
+------------------------------+
```

🧪 Test it

Once compiled and flashed:

```bash
# Check daemon is running
adb shell ps -Z | grep mydaemon

# Check domain assigned
u:r:mydaemon:s0    system     ... /vendor/bin/mydaemon
```

If not working, check:

```bash
adb logcat | grep avc:
```

Let me know if you want:

- A working sample `.c` file for **`mydaemon`**
- An **audit2allow** suggestion to fix denials
- Or a **debugging checklist** for SELinux startup issues

58. ↪
file_contexts
why this

Great! You're asking **"Why do we need `file_contexts`?"** — let's explain it in **very simple terms**.

✅ What is `file_contexts`?

`file_contexts` is the file where you **assign SELinux labels (contexts) to files or paths** on the filesystem.

❓ Why is it needed?

SELinux controls access using **labels**, not just file paths or Linux permissions.

So even if your binary **`/vendor/bin/mydaemon`** exists:

> If it doesn't have the **right SELinux label**, your service will **not get the correct permissions or domain**.

🔍 Example

Suppose we have this line in **`file_contexts`**:

```c
/vendor/bin/mydaemon  u:object_r:mydaemon_exec:s0
```

This means:

- The binary file **`/vendor/bin/mydaemon`** is given the **SELinux label `mydaemon_exec`**
- That label is required to allow SELinux to:
  - Match it with your **`.te`** file
  - Do a domain transition (**`init`** → **`mydaemon`** domain)

🧩 How it fits in the full flow

File	Role
`mydaemon.te`	Defines **`mydaemon`** domain and rules
`file_contexts`	Says "this file = mydaemon_exec label"
`init.rc`	Runs the binary and sets seclabel
SELinux at runtime	Matches label to domain for transition

Without **`file_contexts`**:

- SELinux doesn't know your binary is **`mydaemon_exec`**
- The **domain transition will fail**
- You may see errors like:

  ```log
  avc: denied { transition } for ...
  ```

🧪 When it runs?

The label from **`file_contexts`** is applied:

- **During build**, when the filesystem is created
- Or at runtime using `restorecon` or during OTA upgrade

✅ Summary (in 1 line):

> `file_contexts` maps **file paths to SELinux labels**, so that the kernel knows what type each file is and enforces the correct access and domain behavior.

Let me know if you'd like:

- Help generating `file_contexts` entries automatically
- How to re-label a file using `restorecon` or `chcon`

59. confusing

No worries — let's make it **super simple** with a real-world analogy 👇

🎯 What is `file_contexts` in simple words?

It's a file where you **tell SELinux which label to give to which file** on your device.

📦 Analogy: File labels = Tags on boxes

Imagine you have boxes in a warehouse:

- One box = your file (e.g. `/vendor/bin/mydaemon`)
- A tag = SELinux label (e.g. `mydaemon_exec`)

If you don't tag the box right, no one knows:

- What's inside
- Who's allowed to open it

That's what `file_contexts` does:

> 🏷️ It tells the system:
> "Hey, this file at `/vendor/bin/mydaemon` should be tagged as `mydaemon_exec`."

🔧 Why is this important?

Because in your `.te` file, you said:

```c
type mydaemon_exec, exec_type, file_type;
```

This only works **if the binary file** (`/vendor/bin/mydaemon`) is labeled `mydaemon_exec`.

So you must connect:

- The **file** → `/vendor/bin/mydaemon`
- The **label** → `mydaemon_exec`

You do that in:

```c
file_contexts
```

✅ Example

Your `file_contexts` file contains:

```c
/vendor/bin/mydaemon  u:object_r:mydaemon_exec:s0
```

💡 This means:

> The file `/vendor/bin/mydaemon` should be labeled with `mydaemon_exec`.

That's it!

🔁 When does this happen?

1. When Android **builds the image**, it labels files using **`file_contexts`**
2. You can also fix a label with:

```bash
restorecon /vendor/bin/mydaemon
```

🧠 TL;DR

Item	Meaning
`file_contexts`	Maps file paths to SELinux labels
Why needed?	SELinux rules only work if files are labeled
Label used where?	In **`.te`** rules (e.g., **`type mydaemon_exec`**)

Would you like a **visual flow** to see how this works end-to-end?

60. ↪
mydaemon_exec
why this name

Great question! Let's break down **why we name it `mydaemon_exec`** in a simple and clear way:

🔹 Why the name `mydaemon_exec`?

The name follows a **common SELinux naming pattern**:

```php-template
<process_name>_exec
```

- **`mydaemon`** → the name of your **service/process**
- **`_exec`** → this label is used for the **executable file** (binary) of that service

---

✅ What it tells SELinux:

```c
type mydaemon_exec, exec_type, file_type;
```

This says:

> "This label **`mydaemon_exec`** is for a **file** that is **executable**, and is the binary that starts the **`mydaemon`** process."

🧠 Why use `_exec`?

Because SELinux needs to:

1. Know that the **file is executable**
2. Allow **`init`** or other domains to **run** this file
3. Trigger a **domain transition** — e.g., from **`init`** to **`mydaemon`**

The **`_exec`** suffix helps us remember:
👉 **This is not a running process. It's the file that gets executed.**

🔄 Relationship:

SELinux Type	Purpose	Example File or Process
`mydaemon_exec`	The binary file that gets executed	**`/vendor/bin/mydaemon`**
`mydaemon`	The domain of the running process	The **`mydaemon`** service in **`ps -Z`**

📦 Analogy (Box + Label)

Thing	Label Used	Why
Binary file	**`mydaemon_exec`**	It's what gets executed
Running process	**`mydaemon`**	It's the domain it runs in

✏️ Can I use a different name?

Technically yes, but:

- You must use **consistent names** in `.te`, `file_contexts`, and `init.rc`
- Using `_exec` makes your code **readable** and **standard**

✅ Summary

Name	Role in SELinux
`mydaemon_exec`	Label for your daemon's executable file
`mydaemon`	Label/domain for the running process

This naming makes it very clear what each part is for.

Let me know if you'd like a diagram to show how `init` runs the binary and transitions domains using `mydaemon_exec`.

61. ↪
exec_type, file_type
what are these two

Great — let's now explain these two **SELinux type attributes** in **very simple words**:

✅ Line:

```c
type mydaemon_exec, exec_type, file_type;
```

This means:

> "Create a new label called **`mydaemon_exec`**, and mark it as a type of **file** that is also **executable**."

Let's break down the two keywords you're asking about:

🔹 `file_type`

What it means:
👉 "This label applies to **regular files**."

Why it's needed:
So SELinux knows that **`mydaemon_exec`** will be used to label a **file** (not a process, socket, etc).

🧠 Think of it like saying:

> "This is a **label for a file** on the filesystem."

🔹 `exec_type`

What it means:
👉 "This file can be **executed** by a process and may trigger a **domain transition**."

Why it's needed:
So SELinux knows:

- This file is an **executable binary**
- If a process like **`init`** runs it, SELinux can **switch to another domain** (like **`mydaemon`**)

🧠 Think of it like saying:

> "This is a **program** that can be launched and changes the process domain."

🧩 How these work together

You're telling SELinux:

> "**`mydaemon_exec`** is a **file** that can be **executed**, and may trigger a domain change to **`mydaemon`**."

So that:

- The **file** **`/vendor/bin/mydaemon`** is tagged with **`mydaemon_exec`**
- When **`init`** runs it, the **process** runs in the **`mydaemon`** domain

---

🧠 Real-life analogy

Term	Think of it like...
`file_type`	Says “This thing is a regular file.”
`exec_type`	Says “This file can be executed like a program.”

✅ Summary

Attribute	Meaning	Needed for
`file_type`	It's a file	So SELinux can label a file
`exec_type`	It's executable, may change domain	To allow process start and domain transition

Let me know if you want to see a full flow of how **`init`** → **`exec_type`** → domain transition works!

62. ↪
u:r:mydaemon:s0
what is the meaning of this

Great! Let's break down this SELinux **context string**:

```makefile
u:r:mydaemon:s0
```

It's called a **security context** or **label**.

✅ Structure of a SELinux context

A typical SELinux context looks like:

```pgsql
user : role : type : level
```

So in your case:

Part	Value	Meaning
user	`u`	SELinux user (Android only uses `u`)
role	`r`	SELinux role (Android only uses `r` for processes)
type	`mydaemon`	The **SELinux domain** the process runs in (very important!)
level	`s0`	Sensitivity level (Android always uses `s0`)

🔍 So what does `u:r:mydaemon:s0` mean?

> This is a **process** (or file) that:
>
> - belongs to SELinux user `u`
> - is using the default role `r`
> - is running in the domain `mydaemon`
> - is at security level `s0` (single-level security)

💡 **Key point:**
The most important part for you is:

```nginx
mydaemon
```

This is the **domain** you defined in your `.te` file:

```c
type mydaemon, domain;
```

🧠 Why is this important?

Because SELinux decides:

- Who can talk to whom
- What files a process can access

Based on this context.
If **`mydaemon`** domain has no allow rules, the kernel will block it.

✅ Summary

Part	Value	Android usage
user	**`u`**	Always **`u`** in Android
role	**`r`**	Always **`r`** (process role)
type	**`mydaemon`**	Custom domain you define in **`.te`**
level	**`s0`**	Android uses single level (MLS)

Let me know if you want:

- A diagram of domain transition using this context
- Or a cheat sheet of common SELinux contexts in Android

63. ↪
/vendor/bin/mydaemon u:object_r:mydaemon_exec:s0 /vendor/log/mydaemon.log u:object_r:mydaemon_log:s0
how to interpret this

You're asking how to **interpret these SELinux context mappings**:

```bash
/vendor/bin/mydaemon      u:object_r:mydaemon_exec:s0
/vendor/log/mydaemon.log  u:object_r:mydaemon_log:s0
```

Let's break it down **line by line** in very simple words 👇

🔹 1. `/vendor/bin/mydaemon u:object_r:mydaemon_exec:s0`

This means:

> The file **`/vendor/bin/mydaemon`** should be labeled with:

```nginx
SELinux context → u:object_r:mydaemon_exec:s0
```

🔍 Interpretation:

Part	Meaning
`/vendor/bin/mydaemon`	Path to the binary file
`u`	SELinux user → Always **u** in Android
`object_r`	Role for objects (files)
`mydaemon_exec`	Type → Marks this as the **executable label** used for launching the service
`s0`	Security level (Android always uses **`s0`**)

✅ So, this tells SELinux:
→ "Hey, when you see this file, treat it as a **launchable binary** for the **`mydaemon`** service."

🔹 2. `/vendor/log/mydaemon.log u:object_r:mydaemon_log:s0`

This means:

> The file **`/vendor/log/mydaemon.log`** should be labeled with:

```nginx
SELinux context → u:object_r:mydaemon_log:s0
```

🔍 Interpretation:

Part	Meaning
`/vendor/log/mydaemon.log`	Path to a log file for your daemon
`mydaemon_log`	Type label used to control **read/write** access to this file
`object_r`	File role
`u`, **`s0`**	Standard Android user and level

✅ This says:
→ "This file is a log file for **`mydaemon`**, and only processes allowed by SELinux to access **`mydaemon_log`** can read/write it."

🧩 Why does this matter?

If you don't set these labels:

- The binary may **not execute**
- The daemon process may **not get the correct domain**
- The daemon may **not be allowed** to write to `/vendor/log/mydaemon.log`

SELinux enforces based on **type (label)** — not just Linux permissions.

✅ Summary Table

File	SELinux Context	Purpose
`/vendor/bin/mydaemon`	`mydaemon_exec`	Marks it as an executable binary
`/vendor/log/mydaemon.log`	`mydaemon_log`	Marks it as a daemon-specific log file

Let me know if you want help:

- Writing the rules to allow `mydaemon` to write to this log
- Or checking current file labels on your device with `ls -Z`

64. who is selinux user here?

Great question!

In the SELinux context:

```bash
/vendor/bin/mydaemon      u:object_r:mydaemon_exec:s0
```

The **SELinux user** is the first field:

```nginx
u
 ↑
```

🔹 Who is the SELinux user here?

Answer: `u`

In Android, the only SELinux user used is:

```sql
u → The default SELinux user
```

🔍 What is a SELinux user?

In traditional SELinux (like on Fedora or RHEL):

- SELinux users can be things like `staff_u`, `user_u`, `system_u`
- They're used for **role-based access control** (RBAC)

BUT in Android:

- Android **does not use** SELinux roles or user-based access
- So **all processes and objects** are labeled with the same SELinux user: `u`

✅ Example Context

```pgsql
u:object_r:mydaemon_exec:s0
|   |         |            └── MLS level (always s0 in Android)
|   |         └──────────────── SELinux type (used for access control)
|   └────────────────────────── SELinux role (object_r for files, r for processes)
└────────────────────────────── SELinux user (always u in Android)
```

🧠 Summary

Field	Value in Android	Purpose
SELinux user	`u`	Default, same for everything
SELinux role	`object_r`	For objects (files, dirs, sockets)
SELinux type	`mydaemon_exec`	Controls access (this is the key part)
Level	`s0`	Used for isolation, always `s0`

So in short:

> The SELinux user in Android is **always `u`**, and it doesn't affect access control. The **type field** (like `mydaemon_exec`) is what really matters.

Let me know if you want to dive into the **role** or **type** fields next!

65. what are subjet and object

Great question!

Let's break down **subject vs object in SELinux** in the simplest way possible 👇

🧠 SELinux Rule Structure:

SELinux access rules are written like:

```php-template
allow <subject> <object> : <class> <permissions>;
```

🔹 What is a Subject?

> The **subject** is the **thing that is *doing*** the action — usually a **process**.

Examples of **subjects**:

- An app process (`untrusted_app`)
- A system service (`system_server`)
- Your custom daemon (`mydaemon`)
- The `init` process

🔹 What is an Object?

> The **object** is the **thing being acted on** — usually a **file**, socket, property, etc.

Examples of **objects**:

- A file (`/vendor/bin/mydaemon`)
- A log file (`/vendor/log/mydaemon.log`)
- A system property (`persist.sys.foo`)
- A binder service
- A socket

✅ Real Example

Let’s say your daemon process (`mydaemon`) wants to write to a log file labeled `mydaemon_log`.

The rule looks like:

```c
allow mydaemon mydaemon_log:file { write append };
```

Role	SELinux Type	Meaning
Subject	`mydaemon`	The process doing the write
Object	`mydaemon_log`	The file being written to
Class	`file`	It's a file-type object
Permission	`write append`	Allowed actions

🧩 Analogy (Person + Door)

Concept	Analogy	Example in SELinux
Subject	You	A process (like **`mydaemon`**)
Object	A locked door	A file (like **`mydaemon_log`**)
Action	Opening the door	Writing to the file

TL;DR

Term	Meaning	In SELinux Context
Subject	The **actor**, usually a process	**`mydaemon`**, **`system_server`**
Object	The **target**, like files	**`mydaemon_log`**, **`app_data`**

The SELinux policy controls **what subjects can do to which objects**.

Let me know if you want a diagram to visualize this!

66. Customize SELinux

bookmark_border
After you've integrated the base level of SELinux functionality and thoroughly analyzed the results, you may add your own policy settings to cover your customizations to the Android operating system. These policies must still meet the Android Compatibility program requirements and must not remove the default SELinux settings.

Manufacturers shouldn't remove existing SELinux policy. Otherwise, they risk breaking the Android SELinux implementation and the apps it governs. This includes third-party apps that will likely need to be improved to be compliant and operational. Apps must require no modification to continue functioning on SELinux-enabled devices.

When embarking upon customizing SELinux, remember to:

Write SELinux policy for all new daemons
Use predefined domains whenever appropriate
Assign a domain to any process spawned as an init service
Become familiar with the macros before writing policy
Submit changes to core policy to AOSP
And remember not to:

Create incompatible policy
Allow end user policy customization
Allow MDM policy customizations
Scare users with policy violations
Add backdoors
See the Kernel Security Features section of the Android Compatibility Definition document for specific requirements.

SELinux uses a whitelist approach, meaning all access must be explicitly allowed in policy in order to be granted. Since Android's default SELinux policy already supports the Android Open Source Project, you aren't required to modify SELinux settings in any way. If you do customize SELinux settings, take great care not to break existing apps. To get started:

Use the latest Android kernel.
Adopt the principle of least privilege.
Address only your own additions to Android. The default policy works with the Android Open Source Project codebase automatically.
Compartmentalize software components into modules that conduct singular tasks.
Create SELinux policies that isolate those tasks from unrelated functions.
Put those policies in *.te files (the extension for SELinux policy source files) within the /device/manufacturer/device-name/sepolicy directory and use BOARD_SEPOLICY variables to include them in your build.
Make new domains permissive initially. This is done by using a permissive declaration in the domain's .te file.
Analyze results and refine your domain definitions.
Remove the permissive declaration when no further denials appear in userdebug builds.
After you've integrated your SELinux policy change, add a step to your development workflow to ensure SELinux compatibility going forward. In an ideal software development process, SELinux policy changes only when the software model changes and not the actual implementation.

As you start customizing SELinux, first audit your additions to Android. If you've added a component that conducts a new function, ensure the component meets Android's security policy, as well as any associated policy crafted by the OEM, before turning on enforcing mode.

To prevent unnecessary issues, it is better to be overbroad and over-compatible than too restrictive and incompatible, which results in broken device functions. Conversely, if your changes will benefit others, you should submit the modifications to the default SELinux policy as a patch. If the patch is applied to the default security policy, you won't need to make this change with each new Android release.

Example policy statements
SELinux is based upon the M4 computer language and therefore supports a variety of macros to save time.

In the following example, all domains are granted access to read from or write to /dev/null and read from /dev/zero.

```
# Allow read / write access to /dev/null
allow domain null_device:chr_file { getattr open read ioctl lock append write};

# Allow read-only access to /dev/zero
allow domain zero_device:chr_file { getattr open read ioctl lock };
```
This same statement can be written with SELinux *_file_perms macros (shorthand):

Allow read / write access to /dev/null
allow domain null_device:chr_file rw_file_perms;

Allow read-only access to /dev/zero
allow domain zero_device:chr_file r_file_perms;
Example policy
Here is a complete example policy for DHCP, which we examine below:

type dhcp, domain;
permissive dhcp;
type dhcp_exec, exec_type, file_type;
type dhcp_data_file, file_type, data_file_type;

init_daemon_domain(dhcp)
net_domain(dhcp)

allow dhcp self:capability { setgid setuid net_admin net_raw net_bind_service
};
allow dhcp self:packet_socket create_socket_perms;
allow dhcp self:netlink_route_socket { create_socket_perms nlmsg_write };
allow dhcp shell_exec:file rx_file_perms;
allow dhcp system_file:file rx_file_perms;
For /proc/sys/net/ipv4/conf/*/promote_secondaries
allow dhcp proc_net:file write;
allow dhcp system_prop:property_service set ;
unix_socket_connect(dhcp, property, init)

type_transition dhcp system_data_file:{ dir file } dhcp_data_file;
allow dhcp dhcp_data_file:dir create_dir_perms;
allow dhcp dhcp_data_file:file create_file_perms;

allow dhcp netd:fd use;
allow dhcp netd:fifo_file rw_file_perms;
allow dhcp netd:{ dgram_socket_class_set unix_stream_socket } { read write };
allow dhcp netd:{ netlink_kobject_uevent_socket netlink_route_socket
netlink_nflog_socket } { read write };
Let’s dissect the example:

In the first line, the type declaration, the DHCP daemon inherits from the base security policy (domain). From the previous statement examples, DHCP can read from and write to /dev/null.

In the second line, DHCP is identified as a permissive domain.

In the init_daemon_domain(dhcp) line, the policy states DHCP is spawned from init and is allowed to communicate with it.

In the net_domain(dhcp) line, the policy allows DHCP to use common network functionality from the net domain such as reading and writing TCP packets, communicating over sockets, and conducting DNS requests.

In the line allow dhcp proc_net:file write;, the policy states DHCP can write to specific files in /proc. This line demonstrates SELinux's fine-grained file labeling. It uses the proc_net label to limit write access to only the files under /proc/sys/net.

The final block of the example starting with allow dhcp netd:fd use; depicts how apps may be allowed to interact with one another. The policy says DHCP and netd may communicate with one another via file descriptors, FIFO files, datagram sockets, and UNIX stream sockets. DHCP may only read to and write from the datagram sockets and UNIX stream sockets and not create or open them.

Available controls
Class Permission
file

ioctl read write create getattr setattr lock relabelfrom relabelto append
unlink link rename execute swapon quotaon mounton
directory

add_name remove_name reparent search rmdir open audit_access execmod
socket

ioctl read write create getattr setattr lock relabelfrom relabelto append bind
connect listen accept getopt setopt shutdown recvfrom sendto recv_msg send_msg
name_bind
filesystem

mount remount unmount getattr relabelfrom relabelto transition associate
quotamod quotaget
process

fork transition sigchld sigkill sigstop signull signal ptrace getsched setsched
getsession getpgid setpgid getcap setcap share getattr setexec setfscreate
noatsecure siginh setrlimit rlimitinh dyntransition setcurrent execmem
execstack execheap setkeycreate setsockcreate
security

compute_av compute_create compute_member check_context load_policy
compute_relabel compute_user setenforce setbool setsecparam setcheckreqprot
read_policy
capability

chown dac_override dac_read_search fowner fsetid kill setgid setuid setpcap
linux_immutable net_bind_service net_broadcast net_admin net_raw ipc_lock
ipc_owner sys_module sys_rawio sys_chroot sys_ptrace sys_pacct sys_admin
sys_boot sys_nice sys_resource sys_time sys_tty_config mknod lease audit_write
audit_control setfcap
MORE

AND MORE

neverallow rules
SELinux neverallow rules prohibit behavior that should never occur. With compatibility testing, SELinux neverallow rules are now enforced across devices.

The following guidelines are intended to help manufacturers avoid errors related to neverallow rules during customization. The rule numbers used here correspond to Android 5.1 and are subject to change by release.

Rule 48: neverallow { domain -debuggerd -vold -dumpstate -system_server } self:capability sys_ptrace;
See the man page for ptrace. The sys_ptrace capability grants the ability to ptrace any process, which allows a great deal of control over other processes and should belong only to designated system components, outlined in the rule. The need for this capability often indicates the presence of something that isn't meant for user-facing builds or functionality that isn't needed. Remove the unnecessary component.

Rule 76: neverallow { domain -appdomain -dumpstate -shell -system_server -zygote } { file_type -system_file -exec_type }:file execute;
This rule is intended to prevent the execution of arbitrary code on the system. Specifically, it asserts that only code on /system gets executed, which allows security guarantees thanks to mechanisms such as Verified Boot. Often, the best solution when encountering a problem with this neverallow rule is to move the offending code to the /system partition.

Customize SEPolicy in Android 8.0 and higher
This section provides guidelines for vendor SELinux policy in Android 8.0 and higher, including details on Android Open Source Project (AOSP) SEPolicy and SEPolicy extensions. For more information about how SELinux policy is kept compatible across partitions and Android versions, see Compatibility.

Policy placement
In Android 7.0 and earlier, device manufacturers could add policy to BOARD_SEPOLICY_DIRS, including policy meant to augment AOSP policy across different device types. In Android 8.0 and higher, adding a policy to BOARD_SEPOLICY_DIRS places the policy only in the vendor image.

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

system/sepolicy/public. Includes policy exported for use in vendor-specific policy. Everything goes into the Android 8.0 compatibility infrastructure. Public policy is meant to persist across releases so you can include anything /public in your customized policy. Because of this, the type of policy that can be placed in /public is more restricted. Consider this the platform's exported policy API: Anything that deals with the interface between /system and /vendor belongs 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/manufacturer/device-name/sepolicy. Includes device-specific policy. Also includes device customizations to policy, which in Android 8.0 and higher corresponds to policy for components on the vendor image.
In Android 11 and higher, system_ext and product partitions can also include partition-specific policies. system_ext and product policies are also split into public and private, and vendors can use system_ext's and product's public policies, like the system policy.

SYSTEM_EXT_PUBLIC_SEPOLICY_DIRS. Includes policy exported for use in vendor-specific policy. Installed to system_ext partition.
SYSTEM_EXT_PRIVATE_SEPOLICY_DIRS. Includes policy necessary for the functioning of the system_ext image, but of which vendor image policy should have no knowledge. Installed to system_ext partition.
PRODUCT_PUBLIC_SEPOLICY_DIRS. Includes policy exported for use in vendor-specific policy. Installed to product partition.
PRODUCT_PRIVATE_SEPOLICY_DIRS. Includes policy necessary for the functioning of the product image, but of which vendor image policy should have no knowledge. Installed to product partition.
Note: when GSI is used, OEM's system_ext and product partitions won't be mounted. The rules in the vendor sepolicy that uses OEM's system_ext and product public policy become NOP because the OEM-specific type definitions are missing.
Note: Be extra careful when using system_ext and product public policies. Public policies act as exported API between system_ext/product and vendor. Partners are supposed to manage compatibility issues themselves.
Supported policy scenarios
On devices launching with Android 8.0 and higher, 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/manufacturer/device-name/sepolicy. New policy governing how vendor components interact with (only) other vendor components should involve types present only in device/manufacturer/device-name/sepolicy. Policy written here allows code on vendor to work, won't be updated as part of a framework-only OTA, and is 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/manufacturer/device-name/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 won't be updated as part of a framework-only OTA and is 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/manufacturer/device-name/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.

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 (that is, 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 won't 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/manufacturer/device-name/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, that is, don't remove AOSP public policy. The new public types can then be used for policy in system/sepolicy/private and in device/manufacturer/device-name/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 aren't 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 and higher don't 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 the next Android release 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 (for example, hal_foo attribute) but as attribute partner extensions aren't 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 aren't supported, so all policy dictating how the system components interact with new vendor components (and which isn't handled by attributes already present in AOSP system/sepolicy/public) must be in device/manufacturer/device-name/sepolicy. This means that system types cannot change the access allowed to vendor types as part of a framework-only OTA.

Was this helpful?

Yes, this is a **very helpful and detailed guide** on **customizing SELinux (SEPolicy)** in Android, especially for device manufacturers (OEMs) and AOSP developers. Let me summarize and explain key points from this in a structured and simple manner:

🔐 Purpose of SEPolicy Customization

Android's default SELinux policy already protects the AOSP codebase. But if you're:

- Adding custom daemons or services,
- Implementing new HALs,
- Creating your own apps or frameworks,

...then **you must add or customize SELinux policies** to protect and control access for your additions — without breaking the compatibility guarantees enforced by Android.

✅ What You MUST Do

Action	Why
Write new SELinux `.te` policy files	Define rules for your daemons, files, sockets, etc.
Use existing system-defined types and macros wherever possible	To stay compatible with AOSP and reduce duplication
Assign a domain to every init service	Helps SELinux enforce access control per-process

Action	Why
Use **permissive** mode for new domains at first	Helps in debugging policy violations before enforcing
Place your policies in **device/manufacturer/device-name/sepolicy**	This keeps them device/vendor-specific and CTS-compliant
Use **BOARD_SEPOLICY_*** variables in **BoardConfig.mk**	Ensures your policies are compiled into the build

❌ What You MUST NOT Do

Don't do this	Why it's bad
Delete or weaken existing AOSP policy	Breaks compatibility and security
Let end users or MDMs change policy	Violates the MAC (mandatory access control) model
Add debug backdoors or bypass rules	Fails CTS, security audits, and real-world protection
Make new policy too restrictive too soon	It may break system functionality — start permissive first

📂 Where to Put Policies (Android 8+)

Path	Purpose
system/sepolicy/public/	Stable API — types/attributes vendor policy can use
system/sepolicy/private/	Internal system policy — vendor should not touch
device/<vendor>/<device>/sepolicy/	Your custom policy for vendor daemons, HALs, etc.
system_ext, **product** (Android 11+)	Same separation model for those partitions

⚠️ Neverallow Rules

These are **hard "deny rules"** that prevent insecure behavior.

Example:

```c
neverallow { domain -debuggerd -vold -system_server } self:capability sys_ptrace;
```

> This means: only **debuggerd**, **vold**, and **system_server** may use **ptrace**. No other domain is ever allowed.

If you hit a **neverallow** violation, your policy or daemon behavior is unsafe or disallowed.

🧪 Your Workflow

1. Add your daemon or HAL in AOSP.
2. Boot in **permissive** mode and collect AVC denials (**dmesg**, **logcat**, **audit2allow**).
3. Add SELinux rules in **.te** files (e.g., **mydaemon.te**) and label files in **file_contexts**.
4. Gradually refine your policy using real denial logs.
5. When stable, switch to **enforcing** mode.
6. Validate with CTS and VTS.

📦 Example DHCP Policy Summary

```c
type dhcp, domain;
permissive dhcp;  // temporarily permissive
type dhcp_exec, exec_type, file_type;

init_daemon_domain(dhcp)
net_domain(dhcp)

allow dhcp self:capability { setuid net_raw };
allow dhcp proc_net:file write;
```

Line	Meaning
`type dhcp, domain;`	Declares **`dhcp`** as a process domain
`permissive dhcp;`	Lets it log violations without enforcing
`init_daemon_domain(dhcp)`	Allows **`init`** to start **`dhcp`**
`net_domain(dhcp)`	Grants common networking permissions
`allow dhcp proc_net:file write;`	Let **`dhcp`** write to **`/proc/sys/net/...`**

🧠 Key Takeaways

- SELinux is **mandatory** and **whitelist-based** — nothing is allowed unless explicitly permitted.
- You **must isolate** each component and give it **minimal required access** (principle of least privilege).
- **Proper policy modularization** and **partition separation** are critical for updatability and compatibility.
- Don't fight the system: **use macros**, **reuse types**, and **respect neverallow rules**.

Would you like a checklist or template for creating your own **`.te`** + **`file_contexts`** + **`BoardConfig.mk`** changes for your daemon or service?

67. ↪
⚠️ Neverallow Rules ------------------ These are **hard “deny rules”** that prevent insecure behavior. **Example:** c CopyEdit `neverallow { domain -debuggerd -vold -system_server } self:capability sys_ptrace;` > This means: only `debuggerd`, `vold`, and `system_server` may use `ptrace`. No other domain is ever allowed. If you hit a `neverallow` violation, your policy or daemon behavior is unsafe or disallowed.
explain in simple words with example

Sure! Let’s break this down **simply** with an example.

🔐 What is a `neverallow` Rule in SELinux?

A **`neverallow` rule** is like a **"hard no"** in SELinux.

It says:

> “**No matter what**, these types/domains are **never** allowed to do this specific thing.”

It is **enforced at compile time**. If your policy tries to allow it, **the build fails**.

🧠 Think of it Like This

Imagine a building with **many rooms (domains)** and **many doors (access types)**.

You put up a **sign** on one door:
🛑 **Never allow anyone except security and maintenance staff to enter the control room.**

If someone else (e.g. cook, janitor) tries to get access—even if someone says “they're allowed”—the system says **NO!**, and **throws an error**.

🧪 Example Rule:

```c
neverallow { domain -debuggerd -vold -system_server } self:capability sys_ptrace;
```

🔍 Let's Interpret It:

- **`domain`** = all processes
- **`-debuggerd -vold -system_server`** = **except** these three
- **`self:capability sys_ptrace;`** = accessing the **`ptrace`** capability (used to inspect/control other processes)

🚫 What It Means:

> “**All processes** are **forbidden** from using **`ptrace`**, **except** for **`debuggerd`**, **`vold`**, and **`system_server`**.”

Why? Because **`ptrace`** can be dangerous — it lets you spy on or control other processes. Only safe system components should use it.

🧩 What Happens If You Break This?

Suppose you write this in your **`.te`** file:

```c
allow mydaemon self:capability sys_ptrace;
```

If you try to build AOSP, you'll get an error like:

```vbnet
Error: neverallow rule at system/sepolicy/... violated by allow rule in mydaemon.te
```

✅ This protects system security by **preventing developers from accidentally giving too much power** to untrusted processes.

✅ Summary

Concept	Meaning
`neverallow`	A rule that **must never** be violated
Enforced when?	At **build time**, not at runtime
If violated	**Build fails** with error
Purpose	To protect the system from bad policies
Example	Disallowing **`ptrace`** except for key daemons

Would you like to see how to safely test and debug such cases without hitting **`neverallow`** errors?

68. Build SELinux policy

bookmark_border
This page covers how SELinux policy is built. SELinux policy is built from the combination of core AOSP policy (platform) and device-specific policy (vendor). The SELinux policy build flow for Android 4.4 through Android 7.0 merged all sepolicy fragments then generated monolithic files in the root directory. This meant that SoC vendors and ODM manufacturers modified boot.img (for non-A/B devices) or system.img (for A/B devices) every time policy was modified.

In Android 8.0 and higher, platform and vendor policy is built separately. SOCs and OEMs can update their parts of the policy, build their images (such as, vendor.img and boot.img), then update those images independent of platform updates.

However, as modularized SELinux policy files are stored on /vendor partitions, 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).

Source files
The logic for building SELinux is in these 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, see 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).
For more details on the files in system/sepolicy Implementing SELinux.

Android 7.x and lower
This section covers how SELinux policy is built in Android 7.x and lower.

Build process for Android 7.x and lower
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:

This image shows the SELinux build logic for Android 7.x.
Figure 1. 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 files, genfs_contexts, and port_contexts in that order.
For each file (such as 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.
This image shows the files that generate the SELinux policy file
for Android 7.x.
Figure 2. SELinux policy file.

SELinux files
After compiling, Android devices running 7.x and earlier typically contain the following SELinux-related files:

selinux_version
sepolicy: binary output after combining policy files (such as, security_classes, initial_sids, and *.te)
file_contexts

property_contexts
seapp_contexts
service_contexts
system/etc/mac_permissions.xml
For more details, see Implementing SELinux.

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

Loads sepolicy files from ramdisk into the kernel through /sys/fs/selinux/load.
Switches SELinux to enforcing mode.
Runs re-exec() 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 and higher
In Android 8.0, SELinux policy is split into platform and vendor components to allow independent platform/vendor 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 vendor 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.

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

Policy exists in the following locations:

Location	Contains
system/sepolicy/public	The platform's sepolicy API
system/sepolicy/private	Platform implementation details (vendors can ignore)
system/sepolicy/vendor	Policy and context files that vendors can use (vendors can ignore if desired)
BOARD_SEPOLICY_DIRS	Vendor sepolicy
BOARD_ODM_SEPOLICY_DIRS (Android 9 and higher)	Odm sepolicy
SYSTEM_EXT_PUBLIC_SEPOLICY_DIRS (Android 11 and higher)	System_ext's sepolicy API
SYSTEM_EXT_PRIVATE_SEPOLICY_DIRS (Android 11 and higher)	System_ext implementation details (vendors can ignore)
PRODUCT_PUBLIC_SEPOLICY_DIRS (Android 11 and higher)	Product's sepolicy API
PRODUCT_PRIVATE_SEPOLICY_DIRS (Android 11 and higher)	Product implementation details (vendors can ignore)

The build system takes this policy and produces system, system_ext, product, vendor, and odm policy components on the corresponding partition. Steps include:

Converting policies to the SELinux Common Intermediate Language (CIL) format, specifically:
public platform policy (system + system_ext + product)
combined private + public policy
public + vendor and BOARD_SEPOLICY_DIRS policy
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.
Creating a mapping file linking the platform and vendor 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.
Combining policy files (describe both on-device and precompiled solutions).
Combine mapping, platform and vendor policy.
Compile output binary policy file.
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 vendor (that is, device) policy developers may write additional device-specific policy.

Types are versioned according to the version of the policy that vendor files are written against, defined by the PLATFORM_SEPOLICY_VERSION build variable. The versioned public policy is then included with the vendor 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 aren't 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 vendor 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. For more details, see Compatibility.

Android 11 and higher
system_ext and product sepolicy
In Android 11, system_ext policy and product policy are added. Like the platform sepolicy, system_ext policy and product policy are split into public policy and private policy.

Public policy is exported to vendor. Types and attributes become stable API, and vendor policy can refer to types and attributes in the public policy. Types are versioned according to PLATFORM_SEPOLICY_VERSION, and the versioned policy is included to the vendor policy. The original policy is included to each of system_ext and product partition.

Private policy contains system_ext-only and product-only types, permissions, and attributes required for system_ext and product partitions' functionality. Private policy is invisible to vendor, implying that these rules are internal and allowed to be modified.

system_ext and product mapping
system_ext and product are allowed to export their designated public types to vendor. However, the responsibility to maintain compatibility is on each partner's own. For compatibility, partners can provide their own mapping files which map the versioned attributes of previous versions to concrete types used in current public sepolicy.

To install a mapping file for system_ext, place a cil file containing the desired mapping information to {SYSTEM_EXT_PRIVATE_SEPOLICY_DIRS}/compat/{ver}/{ver}.cil, and then add system_ext_{ver}.cil to PRODUCT_PACKAGES.
To install a mapping file for product, place a cil file containing the desired mapping information to {PRODUCT_PRIVATE_SEPOLICY_DIRS}/compat/{ver}/{ver}.cil, and then add product_{ver}.cil to PRODUCT_PACKAGES.
Refer to an example which adds a mapping file of redbull device's product partition.

Precompiled SELinux policy
Before init turns on SELinux, init gathers all CIL files from partitions (system, system_ext, product, vendor and odm) and compiles them into binary policy, the format which can be loaded to kernel. As the compilation takes time (usually 1-2 seconds), the CIL files are pre-compiled at build time and placed at either /vendor/etc/selinux/precompiled_sepolicy or /odm/etc/selinux/precompiled_sepolicy, along with the sha256 hashes of the input CIL files. At runtime, init checks if any of the policy file has been updated by comparing the hashes. If nothing has changed, init loads the precompiled policy. If not, init compiles on the fly and uses it instead of the precompiled one.

More specifically, precompiled policy is used if all of the following conditions are met. Here, {partition} represents the partition where the precompiled policy exists: either vendor or odm.

Both /system/etc/selinux/plat_sepolicy_and_mapping.sha256 and /{partition}/etc/selinux/precompiled_sepolicy.plat_sepolicy_and_mapping.sha256 exist and are identical.
Both /system_ext/etc/selinux/system_ext_sepolicy_and_mapping.sha256 and /{partition}/etc/selinux/precompiled_sepolicy.system_ext_sepolicy_and_mapping.sha256 don't exist. Or both exist and are identical.
Both /product/etc/selinux/product_sepolicy_and_mapping.sha256 and /{partition}/etc/selinux/precompiled_sepolicy.product_sepolicy_and_mapping.sha256 don't exist. Or both exist and are identical.
If any of them differ, init falls back to the on-device compilation path. See system/core/init/selinux.cpp for more details.

Here's a **simple explanation** of how **SELinux policy is built** in Android, with examples and a focus on Android 8.0+ (the current model used in modern Android):

✅ What is SELinux Policy in Android?

SELinux (Security-Enhanced Linux) policy controls **which process (subject)** can **access which resource (object)** and **how (permission)**.

> Example:
>
> - **`dhcp`** process can **read/write** **`/dev/null`**
> - **`netd`** process **cannot** use **`ptrace`**

🏗️ How SELinux Policy is Built (Android 8.0 and Higher)

🧩 Key Change in Android 8.0:

Before Android 8.0:

- One big `sepolicy` file combined platform + vendor rules.
- Any small vendor change required rebuilding `boot.img` or `system.img`.

Now (Android 8.0+):

- **Platform policy** and **vendor policy** are built **separately**.
- They are merged at **runtime** during boot.
- This supports **modular updates** (like OTA updates or Project Treble).

📁 Where is Policy Stored?

Policy Type	Location Example	Who writes it	Used by
Platform Public	`system/sepolicy/public`	Google (AOSP)	OEMs can use
Platform Private	`system/sepolicy/private`	Google	Internal only
Vendor Policy	`device/vendor/device_name/sepolicy/`	OEMs (you!)	For device
system_ext Policy	`system_ext/sepolicy/public	private`	OEMs (shared libs)
product Policy	`product/sepolicy/public	private`	OEMs (apps/tools)

> Example:
> Your camera HAL is added to `/vendor`, so you write its policy in:
> `device/myvendor/mydevice/sepolicy/camera.te`

🔄 How They're Merged at Boot

At boot time:

1. **Init** reads all `.cil` files (SELinux Intermediate Language format) from:
   - `/system`
   - `/vendor`
   - `/product`
   - `/system_ext`
   - `/odm`
2. It checks if **precompiled binary policy** is available.
3. If yes and valid → loads fast ✅
   If not → compiles on device (slower) ❌

📦 Precompiled Policy

To **save boot time**, Android pre-compiles SELinux policy into a binary file at build time.

- Precompiled policy lives in:

```swift
/vendor/etc/selinux/precompiled_sepolicy
/odm/etc/selinux/precompiled_sepolicy
```

- At runtime, `init` verifies SHA256 hash of each `.cil` file.
  - If all match → load precompiled 🔥
  - If mismatch → recompile on-device 🐢

🗂️ Build Process Flow

Step-by-Step (Simplified):

1. Platform builds:
   - **`platform_public.cil`** (API for vendor use)
   - **`platform_private.cil`** (internal)
2. Vendor builds:
   - Vendor rules using public types → **`vendor.cil`**
3. Mapping files:
   - Links platform types (vN) → vendor attributes (vM) for **compatibility**
4. Final binary **`sepolicy`** is compiled from:

```pgsql
platform.cil + system_ext.cil + product.cil + vendor.cil + odm.cil + mapping files
```

👀 Example Policy File Hierarchy

Assume you're building a device for **`mycompany/mydevice`**.

You might place files like:

```bash
device/mycompany/mydevice/sepolicy/
├── file_contexts
├── mydaemon.te            # Type enforcement
├── mydaemon_file.te       # File rules
├── mydaemon_service.te    # Binder/service access
├── mydaemon_init.rc       # Starts daemon
```

🔄 BOARD_SEPOLICY_DIRS

In your **`BoardConfig.mk`**:

```make
BOARD_SEPOLICY_DIRS += device/mycompany/common/sepolicy
BOARD_SEPOLICY_DIRS += device/mycompany/mydevice/sepolicy
```

This tells the build system to **include your `.te` and `.cil` files** when compiling the final policy.

🧠 Summary Table

Concept	Explanation
Platform policy	Core SELinux rules from AOSP (Google)
Vendor policy	Your rules (device-specific)
Public policy	Shared interface types/attributes
Private policy	Internal-only, not for vendor use
Mapping file	Links old platform public APIs to new ones for compatibility
Precompiled policy	Faster boot (stored binary)
BOARD_SEPOLICY_DIRS	Points to your `.te` and `.cil` source files

💡 Tip

To **debug SELinux denials**, use:

```bash
adb shell dmesg | grep avc
```

To **test permissively before enforcing**, use:

```te
permissive mydaemon;
```

Then switch to **enforcing** once clean.

Would you like a diagram of the SELinux policy build flow from system/vendor?

69. ↪
.cil
what is this

`.cil` stands for **Common Intermediate Language** in the context of **SELinux**.

🔍 What is `.cil`?

`.cil` files are **intermediate** representations of SELinux policy rules — a **standard format** that is easier to parse and manage than the older `.te` (Type Enforcement) or `.conf` formats.

- Introduced in **Android 8.0+** as part of **modular SELinux policy** support.
- SELinux tools (like `checkpolicy`) now **compile policies from `.cil`** instead of from raw `.te` files.
- Used to **merge platform + vendor policies** reliably across devices.

🧠 Why `.cil` Matters in Android?

Before Android 8.0:

- All SELinux policy was built into one giant `policy.conf` → compiled to binary `sepolicy`.

After Android 8.0:

- Android modularized the policy.
- All policy fragments (system, vendor, product, etc.) are **converted to `.cil` files first**, then merged.

🏗️ What does a `.cil` file look like?

Here's a basic `.cil` example (human-readable):

```lisp
(type mydaemon)
(type mydaemon_exec)
(typeattribute exec_type (mydaemon_exec))
(allow mydaemon self (process (transition)))
```

It's like a **LISP-style syntax**:

- `type`, `allow`, `typeattribute` = keywords
- `mydaemon`, `self`, `process` = domains/subjects
- `(transition)` = permissions

🧩 Where do you see `.cil` files?

They are typically found **during the build process** in:

```bash
out/target/product/<device>/sepolicy_intermediates/
```

Or precompiled versions on the device at:

```bash
/vendor/etc/selinux/
```

Examples:

- `vendor_sepolicy.cil`
- `plat_sepolicy.cil`
- `product_sepolicy.cil`

⚙️ How are `.cil` files used?

They are:

1. **Generated** from `.te`, `.fc`, `.booleans`, etc.
2. **Merged** across partitions (system + vendor + product)
3. **Compiled** into final `sepolicy` binary using tools like `secilc`.

✅ Summary

Term	Meaning
`.cil`	Common Intermediate Language (for SELinux)
Used In	Android 8.0+ SELinux policy builds
Format	LISP-style, machine-readable and mergeable
Output	Final compiled `sepolicy` binary for kernel

Let me know if you want a `.te` → `.cil` example mapping or the full SELinux compilation pipeline!

70. who create this cli

The `.cil` **(Common Intermediate Language)** files used in **SELinux** are generated by the **SELinux policy compiler toolchain**, specifically:

🛠️ Tools that Create `.cil` Files

1. **`checkpolicy`**
   - Originally compiled `.te` (Type Enforcement) policies into binary.
   - In modern Android and SELinux systems, this supports generating `.cil` format.
2. **`secilc` (SELinux CIL Compiler)** ✅
   👉 **This is the main tool used to generate and compile `.cil` files.**
   - It reads `.cil` files.
   - It compiles them into the final `sepolicy` binary that is loaded by the Linux kernel.
   - Also used to validate and link `.cil` fragments from different partitions.

📍 Where is `secilc` from?

- It's part of the **SELinux userspace project** maintained by the **NSA (National Security Agency)**.
- In Android, it's included in AOSP under:

  ```bash
  external/selinux/checkpolicy/secilc
  ```

📦 In AOSP Build System

In Android (AOSP):

- `.te` → converted to `.cil` using `m4` macros and SELinux build scripts.
- These are then fed to `secilc` to build the final binary policy.

The relevant Android build files involved:

```bash
system/sepolicy/Android.mk
```

👷 Who Triggers the Creation?

The `.cil` files are created:

- **Automatically** during the Android **build process**.
- Triggered by the build system (`make`, `soong`, `ninja`) when compiling SELinux policy.

If you're a vendor or OEM:

- You define `.te`, `file_contexts`, etc. in your device-specific directories.
- The build system converts and merges these into `.cil` automatically.

🔁 Summary

Component	Role
`secilc`	Compiles `.cil` → `sepolicy` binary
`checkpolicy`	Older tool; now also supports `.cil` output

Component	Role
AOSP Build	Automates `.te` → `.cil` conversion
NSA / SELinux	Origin of the SELinux compiler toolchain

Would you like a step-by-step `.te` → `.cil` → `sepolicy` example?

71. prebuilt selinux policy and not build?

Great question. Let's clarify:

🔒 What Is a Prebuilt SELinux Policy?

A **prebuilt SELinux policy** is a **precompiled `.cil` → `sepolicy` binary**, generated **at build time**, and **shipped inside the Android image** so that `init` **doesn't need to compile policy at boot time**.

Instead of compiling `.cil` → `sepolicy` during **every boot**, which takes 1–2 seconds, Android (from 10+) supports **loading a precompiled binary policy** directly—**if and only if** the hashes of the inputs match.

🔄 Why Use Prebuilt Instead of Build-Time Compilation on Device?

	Build at Runtime (during boot)	Prebuilt Policy (`precompiled_sepolicy`)
🔄 Compiled by	`init` using `secilc`	Already compiled during AOSP build
⏱️ Time	Slower boot (adds 1–2 seconds)	Faster boot time
🔍 Hash check	Not needed	Must validate `.sha256` hashes of `.cil` inputs
📦 Stored in	Memory (compiled in RAM)	`/vendor/etc/selinux/precompiled_sepolicy`
🛠️ Update safe	Useful for dynamic builds/debugging	Useful for production-ready devices

📁 Where Are Prebuilt Policies Stored?

For example:

```swift
/vendor/etc/selinux/precompiled_sepolicy
/vendor/etc/selinux/precompiled_sepolicy.plat_sepolicy_and_mapping.sha256
/vendor/etc/selinux/precompiled_sepolicy.system_ext_sepolicy_and_mapping.sha256
/vendor/etc/selinux/precompiled_sepolicy.product_sepolicy_and_mapping.sha256
```

✅ When Is Prebuilt Policy Used?

During early boot, `init` checks if the **SHA-256 hashes** of all partition `.cil` input files match the ones used during pre-compilation:

```c
if (all_hashes_match) {
    load_precompiled_policy();
} else {
    compile_and_load_on_boot();
}
```

This logic is in:

```swift
system/core/init/selinux.cpp
```

📦 Prebuilt Policy Is Created During the Android Build

As part of `make` / `soong` / `ninja`:

- `.te`, `.fc`, `.seapp_contexts` → `.cil`
- `.cil` files are compiled using `secilc` into:

  - `sepolicy`
  - `.sha256` hash files
- These are copied to `precompiled_sepolicy` under `vendor/` or `odm/`

🧠 Why Prefer Prebuilt in Production?

- ✅ **Faster boot**
- ✅ **Avoids on-device compile**
- ✅ **Ensures consistent, validated policy**
- ✅ **No dependency on `secilc` in initramfs**

🚫 When Is Runtime Compilation Forced?

Prebuilt policy will **NOT** be used if:

1. Any `.sha256` hash mismatch occurs.
2. A partition (e.g., system_ext, product) is missing a hash or `.cil`.
3. Vendor has a mismatched PLATFORM_SEPOLICY_VERSION.

Then `init` falls back to:

```bash
/system/bin/secilc -o /sys/fs/selinux/policy ... all .cil files ...
```

Summary

Term	Means
Prebuilt SELinux	Policy compiled ahead of time during Android build
Stored at	`/vendor/etc/selinux/precompiled_sepolicy`
Used when	All `.sha256` hashes of `.cil` inputs match at boot
Benefit	Fast boot, no need to compile policy at runtime
Fallback	`init` compiles `.cil` → binary if any hash mismatch or file missing

Would you like me to show how to generate your own precompiled policy during an AOSP build?

72. Uploaded image
Uploaded image

These two diagrams show how **SELinux policy** is built on Android 7.x and below. Let's break both down **simply and clearly**.

✅ Diagram 1: SELinux `file_contexts` Build Process

This shows how the `file_contexts` file is created.

🔸 What is `file_contexts`?

It's a file that **maps file paths on the device** (like `/system/bin/app_process`) to **SELinux labels** (like `u:object_r:zygote_exec:s0`).

🔸 Where are inputs taken from?

- `system/sepolicy/file_contexts` – AOSP base context definitions
- `device/$SOC/common/sepolicy/file_contexts` – SoC vendor additions
- `device/$SOC/$DEVICE/sepolicy/file_contexts` – Device-specific overrides

These are **combined** and compiled using the **SELinux Compiler** from `external/selinux`.

🔸 Output:

- A binary file: `file_contexts.bin` in `root/` directory.
- This binary is loaded during Android boot and used to label files.

✅ Diagram 2: SELinux `policy.conf` Build Process

This shows how the complete SELinux policy is built.

🔸 What is `policy.conf`?

It's a **combined source file** made by merging multiple types of policy files.

🔸 Components:

1. `security_classes`: Defines types of objects (e.g., `file`, `process`, `socket`)
2. `initial_sids`: Predefined security identifiers used at boot
3. `*.te` **files**: Type enforcement rules — who can do what to whom
4. `genfs_contexts`: Contexts for filesystems not managed by `file_contexts` (e.g., procfs, sysfs)
5. `port_contexts`: Contexts for network ports

🔸 Where do these come from?

From three layers:

- `system/sepolicy`
- `device/$SoC/common/sepolicy`
- `device/$SoC/$DEVICE/sepolicy`

Each layer contributes its own files.

🔸 Build process:

- All these are **merged** into `policy.conf`
- Passed to the **SELinux compiler** (`checkpolicy` tool from `external/selinux`)

- Result: binary `sepolicy` file in `/root/sepolicy`

🤔 Why is this build system important?

- It allows **custom policy** to be added per-device and per-SOC
- Keeps AOSP policy intact (for compatibility)
- Enables separation of **vendor** and **platform** policy

🔍 Summary (in simple words):

Part	What it does	Built from	Output
`file_contexts.bin`	Labels files by path	Multiple `file_contexts`	Used to label file system
`sepolicy`	Main policy rules (who can do what)	Merged `*.te`, `security_classes`, etc.	Loaded into kernel

Let me know if you'd like to go through **Android 8.0+ (split policy)** version of this too.

73. Policy compatibility

bookmark_border
This article describes how Android handles the policy compatibility issues with platform OTAs, where new platform SELinux settings may differ from old vendor SELinux settings.

Treble-based SELinux policy design considers a binary distinction between platform and vendor policy; the scheme becomes more complicated if vendor partitions generate dependencies, such as platform < vendor < oem.

In Android 8.0 and higher, SELinux global policy is divided into private and public components. 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.
Android maintains a mapping 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 doesn't 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.

Object ownership and labeling
When customizing policy in Android 8.0 and higher, ownership must be clearly defined for each object to keep platform and vendor policy separate. 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 doesn't allow multiple declarations of the same type/attribute. Policy with duplicate declarations will fail to compilation. To avoid type and attribute name collisions, all vendor declarations should be namespaced starting with vendor_.

type foo, domain; → type vendor_foo, domain;
System property and process labeling ownership
Avoiding labeling collisions is best solved using property namespaces. To easily identify platform properties and avoid name conflicts when renaming or adding exported-platform properties, ensure all vendor properties have their own prefixes:

Property type	Acceptable prefixes
control properties	ctl.vendor.
ctl.start$vendor.
ctl.stop$vendor.
init.svc.vendor.
read-writable	vendor.
read-only	ro.vendor.
ro.boot.
ro.hardware.
persistent	persist.vendor.
Vendors can continue to use ro.boot.* (which comes from the kernel cmdline) and ro.hardware.* (an obvious hardware-related property).

All the vendor services in init rc files should have vendor. for services in init rc files of non-system partitions. Similar rules are applied to the SELinux labels for the vendor properties (vendor_ for the vendor properties).

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 isn't practical since many of them are created by the kernel. To prevent these collisions, follow the naming guidance for filesystems in this section. 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 might 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.
* For more examples, see 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 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 shouldn't 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 to Android 10, both platform and vendor label debugfs.

In Android 11, debugfs can't be accessed or mounted on production devices. Device manufacturers should 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 genfscon to label files in sysfs.

Recommendation: The platform may label sysfs nodes that aren't device-specific. 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 (for example, /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.

Genfs labels version
Starting with vendor API level 202504, newer SELinux labels assigned with genfscon in system/sepolicy/compat/plat_sepolicy_genfs_{ver}.cil are

optional for older vendor partitions. This allows older vendor partitions to keep their existing SEPolicy implementation. This is controlled by the Makefile variable BOARD_GENFS_LABELS_VERSION which is stored in /vendor/etc/selinux/genfs_labels_version.txt.

Example:

In vendor API level 202404, the /sys/class/udc node is labeled sysfs by default.
Starting from vendor API level 202504, /sys/class/udc is labeled sysfs_udc.
However, /sys/class/udc might be in use by vendor partitions using API level 202404, either with the default sysfs label or a vendor-specific label. Unconditionally labeling /sys/class/udc as sysfs_udc could break compatibility with these vendor partitions. By checking BOARD_GENFS_LABELS_VERSION, the platform keeps using the previous labels and permissions for the older vendor partitions.

BOARD_GENFS_LABELS_VERSION can be greater than or equal to vendor API level. For instance, vendor partitions using API level 202404 can set BOARD_GENFS_LABELS_VERSION to 202504 to adopt new labels introduced in 202504. See the list of 202504-specific genfs labels.

When labeling genfscon nodes, the platform must consider older vendor partitions and implement fallback mechanisms for compatibility when needed. The platform can use platform-only libraries to query the genfs labels version.

On native, use libgenfslabelsversion. See genfslabelsversion.h for the header file of libgenfslabelsversion.
On Java, use android.os.SELinux.getGenfsLabelsVersion().
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. Type foo is mapped 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 aren't 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 includes the corresponding version) it is transformed into the policy that will go to the vendor portion of the device (shown in the transformed

Common Intermediate Language (CIL)):

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

As vendor policy is never ahead of the platform, it shouldn't 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. Android maintains 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 doesn't 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 policy (CIL):

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

In the v2 vendor 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 didn't previously interact with it directly so no corresponding policy exists. The new attribute corresponding to the type doesn't have an attribute in the previous version, and so wouldn't 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 relies on a rule enabling access to sysfs, and needs to include that rule 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 policy (CIL):

(typeattribute sysfs_v1)
(allow … sysfs_v1 …)
In the v2 vendor 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 possess, 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 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 policy (CIL):

(typeattribute sysfs_A_v1)
(allow … sysfs_A_v1 …)
(typeattribute sysfs_v1)
(allow … sysfs_v1 …)
In the v2 vendor 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 policy (CIL):

(typeattribute foo_v1)
(allow foo …)
(typeattribute sysfs_v1)
(allow sysfs_v1 …)
In the v2 vendor policy (CIL):

(typeattribute sysfs_v2)
(allow sysfs_v2 …)
New class/permissions
This scenario occurs when a platform upgrade introduces new policy components that don't exist in previous versions. For example, when Android added the servicemanager object manager that created the add, find, and list permissions, vendor daemons wanting to register with the servicemanager needed permissions that weren't 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 shouldn't cause issues. The object manager class and permissions could remain defined in policy until the vendor version no longer uses it. This is 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 doesn't 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).

Attribute changes for Android 9
Devices upgrading to Android 9 can use the following attributes, but devices launching with Android 9 must not.

Violator attributes
Android 9 includes these domain-related attributes:

data_between_core_and_vendor_violators. Attribute for all domains that violate the requirement of not sharing files by path between vendor and coredomains. Platform and vendor processes shouldn't use on-disk files to communicate (unstable ABI). Recommendation:
Vendor code should use /data/vendor.
System shouldn't use /data/vendor.
system_executes_vendor_violators. Attribute for all system domains (except init and shell domains) that violate the requirement of not executing vendor binaries. Execution of vendor binaries has unstable API. Platform shouldn't execute vendor binaries directly. Recommendation:
Such platform dependencies on vendor binaries must be behind HIDL HALs.
OR

coredomains that need access to vendor binaries should be moved to the vendor partition and thus, stop being coredomain.
Untrusted attributes
Untrusted apps that host arbitrary code shouldn't have access to HwBinder services, except those considered sufficiently safe for access from such apps (see safe services below). The two main reasons for this are:

HwBinder servers don't perform client authentication because HIDL currently doesn't expose caller UID information. Even if HIDL did expose such data, many HwBinder services either operate at a level below that of apps (such as, HALs) or must not rely on app identity for authorization. Thus, to be safe, the default assumption is that every HwBinder service treats all its clients as equally authorized to perform operations offered by the service.
HAL servers (a subset of HwBinder services) contain code with higher incidence rate of security issues than system/core components and have access to the lower layers of the stack (all the way down to hardware) thus increasing opportunities for bypassing the Android security model.
Safe services
Safe services include:

same_process_hwservice. These services (by definition) run in the process of the client and thus have the same access as the client domain in which the process runs.
coredomain_hwservice. These services don't pose risks associated with reason #2.
hal_configstore_ISurfaceFlingerConfigs. This service is specifically designed for use by any domain.
hal_graphics_allocator_hwservice. These operations are also offered by surfaceflinger Binder service, which apps are permitted to access.
hal_omx_hwservice. This is a HwBinder version of the mediacodec Binder service, which apps are permitted to access.
hal_codec2_hwservice. This is a newer version of hal_omx_hwservice.
Useable attributes
All hwservices not considered safe have the attribute untrusted_app_visible_hwservice. The corresponding HAL servers have the attribute untrusted_app_visible_halserver. Devices launching with Android 9 MUST NOT use either untrusted attribute.

Recommendation:

Untrusted apps should instead talk to a system service that talks to the vendor HIDL HAL. For example, apps can talk to binderservicedomain, then mediaserver (which is a binderservicedomain) in turn talks to the hal_graphics_allocator.
OR

Apps that need direct access to vendor HALs should have their own vendor-defined sepolicy domain.
File attribute tests
Android 9 includes build time tests that ensure all files in specific locations have the appropriate attributes (such as, all files in sysfs have the required sysfs_type attribute).

Platform-public policy
The platform-public policy is the core of conforming to the Android 8.0 architecture model without simply maintaining the union of platform policies from v1 and v2. Vendors are 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 (that is, vendor_sepolicy.cil).

Types and rules are automatically translated in the vendor-generated policy into attribute_vN such that all platform-provided types are versioned attributes (however attributes aren't 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 satisfies 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 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 maintainers 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, the Android platform releases an 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, 26.0 for Oreo, 27.0 for Oreo-MR1, and 28.0 for Android 9. Uprevs aren't always whole numbers. For example, if an MR bump to a versions necessitates an incompatible change in system/sepolicy/public but not an API bump, then that sepolicy version could be: vN.1. The version present in a development branch is a never-to-be-used-in-shipping-devices 10000.0.

Android may deprecate oldest version when upreving. For input on when to deprecate a version, Android may collect the number of devices with vendor policies running that Android version and still receiving major platform updates. If the number is less than a certain threshold, that version is deprecated.

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 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.

system_ext public and product public policy
Starting in Android 11, the system_ext and product partitions are allowed to export their designated public types to the vendor partition. Like platform public policy, the vendor uses types and rules automatically translated into the versioned attributes, for example, from type into type_N, where N is the version of the platform which the vendor partition is built against.

When the system_ext and product partitions are based on the same platform version N, the build system generates base mapping files to system_ext/etc/selinux/mapping/N.cil and product/etc/selinux/mapping/N.cil, which contain identity mappings from type to type_N. The vendor can access type with the versioned attribute type_N.

In case that only the system_ext and product partitions are updated, say N to N+1 (or later), while the vendor stays at N, the vendor may lose access to the types of the system_ext and product partitions. To prevent breakage, the system_ext and product partitions should provide mapping files from concrete types into type_N attributes. Each partner is responsible for maintaining the mapping files, if they are going to support N vendor with N+1 (or later) system_ext and product partitions.

To do that, partners are expected to:

Copy the generated base mapping files from N system_ext and product partitions to their source tree.
Amend the mapping files as needed.
Install the mapping files to N+1 (or later) system_ext and product partitions.
For example, suppose that N system_ext has one public type named foo_type. Then system_ext/etc/selinux/mapping/N.cil in the N system_ext partition will look like:

```
(typeattributeset foo_type_N (foo_type))
(expandtypeattribute foo_type_N true)
(typeattribute foo_type_N)
```

If bar_type is added to N+1 system_ext, and if bar_type should be mapped to foo_type for N vendor, N.cil can be updated from

```
(typeattributeset foo_type_N (foo_type))
```

to

```
(typeattributeset foo_type_N (foo_type bar_type))
```

and then installed to N+1 system_ext's partition. N vendor can continue accessing to N+1 system_ext's foo_type and bar_type.

SELinux contexts labeling
To support the distinction between platform and vendor sepolicy, the system builds SELinux context files differently to keep them separate.

File contexts
Android 8.0 introduced 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 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 vendor file_context.
vendor_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/vendor_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 vendor property_contexts.
vendor_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/vendor_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 vendor service_contexts.
vendor_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/vendor_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 vendor_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 vendor_hwservice_contexts.
vendor_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/vendor_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_SEPOLICY_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.
vendor_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/vendor_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/.

This extensive content you've posted is a **complete explanation of Android's SELinux policy compatibility model**, especially post-Treble (Android 8.0+). Here's a **clean summary and explanation** so you can understand and remember it easily:

✅ Goal of This Policy Compatibility Framework

- Allow **Android platform** to be updated independently of the **vendor partition**.
- Ensure that **old vendor SELinux policies** continue to work with **new platform policies** (during OTA).

- Maintain **policy stability** via versioned attributes and context file separation.

🧱 Main Concepts

1. Split Policy Architecture (Android 8.0+)

- **Platform SELinux policy** lives in `/system`.
- **Vendor SELinux policy** lives in `/vendor`.

This allows:

- Independent updates.
- Compatibility across versions.
- Better modularity (system/vendor separation).

📂 Policy Components and Files

Context Type	Platform File	Vendor File
`file_contexts`	`/system/etc/selinux/plat_file_contexts`	`/vendor/etc/selinux/vendor_file_contexts`
`property_contexts`	`/system/etc/selinux/plat_property_contexts`	`/vendor/etc/selinux/vendor_property_contexts`
`service_contexts`	`/system/etc/selinux/plat_service_contexts`	`/vendor/etc/selinux/vendor_service_contexts`
`seapp_contexts`	`/system/etc/selinux/plat_seapp_contexts`	`/vendor/etc/selinux/vendor_seapp_contexts`
`mac_permissions.xml`	`/system/etc/selinux/mac_permissions.xml`	`/vendor/etc/selinux/mac_permissions.xml`
`hwservice_contexts`	`/system/etc/selinux/plat_hwservice_contexts`	`/vendor/etc/selinux/vendor_hwservice_contexts`
`vndservice_contexts`	*n/a*	`/vendor/etc/selinux/vndservice_contexts`

🧠 Versioned Attributes

Android exports **public types** from platform policy using **versioned attributes** like:

```java
(typeattributeset foo_v28 (foo))
```

This enables:

- Older vendor policies (e.g., written for v27) to still work with platform v28.
- Concrete platform types like `sysfs`, `binder_device`, etc., to remain accessible even if relabeled in newer versions.

🔁 Compatibility Examples

Scenario: Platform relabels `/sys/A` from `sysfs` → `sysfs_A`

Compatibility fix:

- Vendor policy allowed access to `sysfs`.
- Platform now adds:

  ```java
  (typeattributeset sysfs_v27 (sysfs sysfs_A))
  ```

- Vendor still uses:

  ```bash
  (allow v_domain sysfs_v27:dir read;)
  ```

✅ **No breakage**

🛡️ Collision Prevention

Type/Attribute Collisions:

- Vendor types must be prefixed with `vendor_`

```bash
type foo → type vendor_foo
```

Property Collisions:

- Use property namespaces:
  - `vendor.*`, `persist.vendor.*`, `ro.vendor.*`, etc.

File Label Collisions:

- Platform owns `/system`, `/`, `/data` (except `/data/vendor`)
- Vendor owns `/vendor`, `/dev/vendor`, etc.
- Use `vendor_file` as default for vendor partition.

⚙️ Genfs Label Compatibility

- Genfs labels (e.g., `/sys/*`, `/proc/*`) are also version-controlled.
- Controlled by: `BOARD_GENFS_LABELS_VERSION`
- Prevents mismatches like `/sys/class/udc` → `sysfs_udc`

💬 Platform-Public Policy

- AOSP exposes **platform-public policy** to vendors (types/attributes/rules).
- These are automatically converted to versioned attributes during build.
- Vendors write policy using familiar syntax, it's translated to safe versioned CIL:

```arduino
allow foo bar:file read; → (allow foo_vN bar_vN (file (read)))
```

📚 Examples of Versioning

🔸 New Type:

- Platform adds `sysfs_A`
- Maps to: `sysfs_v1 → (sysfs sysfs_A)`

🔸 Removed Type:

- `foo` removed
- `foo_v1 → (foo)` retained for compatibility, then dropped later

🔸 New Class/Perm:

- Platform adds `new_class`
- May allow vendor domains with:

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

🛑 Neverallows & Violators (Android 9+)

`neverallow` rules enforce separation:

- Platform must **not execute vendor binaries** directly.
- No sharing files between platform and vendor via `/data/` (use `/data/vendor` instead).
- Vendor must avoid exposing unsafe HALs to untrusted apps (use system services as proxies).

🧩 system_ext & product Public Policy (Android 11+)

- These partitions can now **export types to vendor** just like `/system`.
- Mappings like:

  ```java
  (typeattributeset foo_type_30 (foo_type bar_type))
  ```

  are installed in `/system_ext/etc/selinux/mapping/30.cil`

🔍 Developer Summary Checklist

Task	Rule
Add a new type?	Prefix with `vendor_` if in vendor policy
Want compatibility across OTA?	Use attributes, not concrete types
Label new `/dev/*` node?	Use `/dev/vendor/*` path
Add new property?	Prefix with `vendor.` or `ro.vendor.` etc.
Access `/sys` or `/proc`?	Let platform label; vendor should access generically
Building vendor policy?	Use BOARD_SEPOLICY_DIRS and versioned attribute sets

If you want, I can give you:

- Visual diagrams of this split structure
- `BoardConfig.mk` setup for vendor SEPolicy
- Hands-on diff example of platform + vendor sepolicy for a real OTA scenario

Just let me know!

74. explain in more simpler

Sure! Here's a **very simple and clear explanation** of everything you just posted — without losing the important meaning.

🧱 What is This About?

Android uses **SELinux** to enforce security.

But when Android gets updated (platform update/OTA), the **vendor code (from device makers)** might be old.

So Android needs a smart way to **keep both platform and vendor SELinux policies working together** — even if they're built at different times.

🧠 Main Idea

Android **splits SELinux policy** into two parts:

Part	Who owns it	Lives in
Platform	Android OS (Google)	`/system`
Vendor	OEM/SoC vendor (e.g., Qualcomm)	`/vendor`

✅ Goal

Make sure:

- **Vendor policy still works** after Android OTA (platform update).
- **Android platform doesn't break** even if vendor code is old.

💡 How Is This Achieved?

1. Use Attributes Instead of Raw Types

Instead of hardcoding labels like this:

```bash
allow myapp sysfs:dir read;
```

Android uses **versioned attributes**:

```bash
allow myapp sysfs_v28:dir read;
```

That means:

- When `sysfs` changes → Android will still allow `myapp` to access it using `sysfs_v28`.
- The mapping is maintained in a file like:

  ```java
  (typeattributeset sysfs_v28 (sysfs sysfs_A))
  ```

2. File Ownership Rules

To prevent **label conflicts**, Android says:

- Only **system** can label files in `/system`

- Only **vendor** can label files in `/vendor`
- Files in `/data` should only be labeled by platform **unless** they're inside `/data/vendor`

This avoids problems where two sides try to label the **same file differently**.

---

3. Property & Service Labeling

Avoid name clashes by using **prefixes**:

Thing	Use This Prefix
System properties	`ro.vendor.*`, `persist.vendor.*`, `vendor.*`
Services in init	Name them like `vendor.servicename`
Types/Attributes	`vendor_foo`, not `foo`

---

4. Context Files Are Split

Each SELinux context file is **split into platform and vendor parts**.

For example:

Context Type	Platform File	Vendor File
`file_contexts`	`/system/etc/selinux/plat_file_contexts`	`/vendor/etc/selinux/vendor_file_contexts`
`property_contexts`	`/system/etc/selinux/plat_property_contexts`	`/vendor/etc/selinux/vendor_property_contexts`
And so on...	...	...

This keeps policies clean and avoids overlap.

---

5. Genfs Labels Compatibility

Some files (like `/proc`, `/sys`) are auto-labeled by the kernel.

Android adds versioning here too:

- File `/sys/class/udc` might be labeled `sysfs` in version 202404
- Later labeled `sysfs_udc` in version 202504

To avoid breaking old vendor code, Android checks a file:

```swift
/vendor/etc/selinux/genfs_labels_version.txt
```

If it's older, platform keeps using old label (`sysfs`).

---

6. Policy Changes Between Versions

Change Type	What Happens
Platform adds new type	New attribute is created, like `sysfs_A_v28`
Platform removes type	Vendor keeps using older attribute like `foo_v27`
Object label changed	Both old and new labels mapped in attribute
Platform adds new class or permission	Platform gives allow rule to all vendor domains that may need it

---

7. Untrusted App Rules (Android 9+)

- Apps **should not** talk to HALs (they might not do permission checks).
- If apps really need access → use system service as proxy.

---

8. Extra Notes for `system_ext` and `product`

From Android 11, `system_ext` and `product` can also share SELinux types with vendor.

But they must:

- Maintain versioned mappings like **`foo_type_30`**
- Place them in **`/system_ext/etc/selinux/mapping/30.cil`**

📌 Summary: What You Should Remember

What	Why It Matters
Platform & vendor SELinux are split	To support independent OTA updates
Use versioned attributes (like **`foo_v28`**)	To avoid breakage when platform labels change
Label ownership rules	Avoid collision (e.g., **`/vendor`** only labeled by vendor)
Prefix everything	Prevent type/property/service name clashes
Context files are split	Keeps platform and vendor policies separate
Old vendor + new platform = OK	Because of attribute mapping and compatibility checks
Only platform can add new permissions/classes	Vendor just consumes what's allowed

---

Would you like a **diagram** of this compatibility system next? Or maybe a **sample vendor sepolicy** layout showing how it uses these rules?

75. please explain with example

Absolutely! Let's walk through a **real-world-style example** to explain how Android SELinux **policy compatibility** works, using a device that **updates from Android 11 to Android 12**, but keeps its old vendor partition.

✅ Setup

Imagine this:

- You're using an Android device with:
  - **Platform (system)** → Upgraded to Android 12 (API 31)
  - **Vendor (vendor)** → Still built for Android 11 (API 30)

You just got an OTA update, but the vendor hasn't shipped an update yet.

Android must **ensure the system doesn't break** due to mismatched SELinux policies.

👇 Problem Scenario

🧩 BEFORE (Android 11)

- `/sys/class/udc` → was labeled:

  ```bash
  type: sysfs
  ```

- The **vendor policy** had this rule:

  ```cil
  allow my_vendor_daemon sysfs:dir read;
  ```

So vendor code can read from `/sys/class/udc`.

🧩 AFTER (Android 12)

- Android platform improved security and **changed the label**:

  ```bash
  /sys/class/udc → type: sysfs_udc
  ```

Now vendor policy breaks, because:

```cil
allow my_vendor_daemon sysfs:dir read;  # Not valid anymore!
```

💡 Solution: Versioned Attributes & Mappings

Android fixes this **automatically** using a **versioned attribute mapping**.

Step 1: Platform defines versioned attributes

Platform has:

```cil
(type sysfs)
(type sysfs_udc)

(typeattributeset sysfs_v30 (sysfs sysfs_udc))  # Android 11
(typeattributeset sysfs_v31 (sysfs))            # Android 12
(typeattributeset sysfs_udc_v31 (sysfs_udc))    # Android 12
```

This tells the system:

- In Android 11 (v30), both **`sysfs`** and **`sysfs_udc`** are treated as **`sysfs_v30`**
- In Android 12 (v31), they're split.

Step 2: Vendor policy uses `sysfs_v30`

The old vendor policy written against Android 11 gets **auto-translated** as:

```cil
(allow my_vendor_daemon sysfs_v30:dir read;)
```

Now, even though **`/sys/class/udc`** is labeled **`sysfs_udc`**, it still works because:

```cil
sysfs_udc ∈ sysfs_v30
```

✔️ **Access is preserved!**

📂 Where Does This Mapping Live?

Android uses these mapping files:

File	Example
`/system/etc/selinux/mapping/30.0.cil`	Maps Android 11 types to attributes
`/system/etc/selinux/mapping/31.0.cil`	Maps Android 12 types to attributes

These are included in platform builds and used to build the final policy image.

🏷️ Example of Type Namespacing

Suppose vendor wants to create a new service and policy:

```bash
service vendor.foo /vendor/bin/foo
```

In **`vendor.te`**:

```cil
type vendor_foo, domain;
init_daemon_domain(vendor_foo)
```

Without namespacing:

```cil
type foo, domain;  # ❌ may clash with platform's foo
```

With namespacing:

```cil
type vendor_foo, domain;  # ✅ safe
```

🏗️ Example: File Label Ownership

Suppose both platform and vendor want to label **`/dev/foo`**.

- Platform's **`plat_file_contexts`**:

```bash
/dev/foo    u:object_r:foo_device:s0
```

- Vendor's **`vendor_file_contexts`**:

```bash
/dev/foo    u:object_r:vendor_foo_device:s0
```

⚠️ Whichever loads **last wins**, causing undefined behavior!

To avoid this:

- Only vendor should label **`/dev/vendor/foo`**
- Platform should avoid touching vendor nodes

🎯 Final Summary with Example References

Concept	Example
Label changes across Android versions	**`/sys/class/udc`** changes from **`sysfs`** → **`sysfs_udc`**
Backward compatibility via attributes	**`sysfs_udc`** is added to **`sysfs_v30`**
Vendor policy doesn't break	**`allow my_vendor_daemon sysfs_v30:dir read;`**
Type namespacing	Use **`vendor_foo`** instead of **`foo`**
Avoid property collisions	Use **`persist.vendor.foo`** not **`persist.foo`**
Avoid file_context collision	Only vendor labels **`/dev/vendor/*`**
Auto-generated mapping	File: **`/system/etc/selinux/mapping/30.0.cil`**

Would you like me to:

- **Show you a real policy diff between two versions**, or
- **Draw a diagram** showing how versioned attribute mappings preserve compatibility?

76. Selinux flow in android explain steps by steps

Here's a **step-by-step explanation** of the **SELinux flow in Android**, from **boot time policy loading** to **runtime enforcement** — explained **clearly and practically**:

✅ OVERVIEW

SELinux (Security-Enhanced Linux) is a Mandatory Access Control (MAC) system used in Android to restrict what processes can do.

🔄 Step-by-Step SELinux Flow in Android

Step 1: Build Time – Compile SEPolicy

1. **Policy sources are defined** in:
    - `system/sepolicy/` → platform policy
    - `device/*/sepolicy/` → device/vendor policy
    - `BOARD_SEPOLICY_DIRS += device/...` in `BoardConfig.mk`
2. **Android.bp/Android.mk/CMake** builds the following:
    - `.cil` (Common Intermediate Language) policy files
    - `plat_sepolicy.cil`, `vendor_sepolicy.cil`, and their versioned mapping files
3. **Mapping is done to maintain backward compatibility**
    - Type `foo` → attribute `foo_vN`
    - Mapping files: `/system/etc/selinux/mapping/NN.cil`
4. Output:
    - `sepolicy` binary file → final merged policy
    - `plat_file_contexts`, `vendor_file_contexts`
    - Other context files (e.g., `service_contexts`, `property_contexts`)

Step 2: Early Boot – Init Loads SELinux

1. **Kernel loads SELinux** as **early as possible**
    - From `init.rc` and kernel command line:

      ```ini
      androidboot.selinux=permissive/enforcing
      ```
2. **Init process (`/init`) sets SELinux to enforcing**
    - Loads:
        - `/system/etc/selinux/plat_file_contexts`
        - `/vendor/etc/selinux/vendor_file_contexts`
        - `/system/etc/selinux/sepolicy` (compiled policy)
3. **Applies `restorecon` on all mounted filesystems** (like `/system`, `/vendor`, `/data`, etc.)
   Labels all files according to `*_file_contexts`.

Step 3: Daemon/Process Startup

1. **Init starts processes** defined in `.rc` files with `seclabel` from:

- `/system/etc/selinux/plat_service_contexts`
- `/vendor/etc/selinux/vendor_service_contexts`

2. Init applies correct **security context** (e.g., `u:r:system_server:s0`) using `sepolicy`.

Step 4: Label Assignment & Enforcement

1. **Every object (file, socket, property, process)** gets a **label**.
   Example:

```bash
/dev/socket/foo → u:object_r:foo_socket:s0
```

2. **Every process runs in a domain (type)**:
   Example:

```yaml
mediaserver → domain: mediaserver
```

3. When `mediaserver` tries to access `/dev/socket/foo`, **SELinux checks:**

```arduino
allow mediaserver foo_socket:sock_file { read write connect };
```

   If not allowed → DENIAL logged in `dmesg` or `logcat`:

```arduino
avc: denied { connect } for pid=1234 ...
```

Step 5: Runtime Auditing and Logging

- **All access checks** are logged via `auditd`:

```perl
adb shell dmesg | grep avc
adb logcat | grep avc
```

- Developer uses `audit2allow` to generate missing allow rules:

```bash
audit2allow -i avc_log.txt -o my_vendor.te
```

Step 6: Policy Update (OTA / Platform Upgrade)

- If platform gets updated, but vendor partition stays old:
  - Use **versioned attributes**
  - Mapping files like `/system/etc/selinux/mapping/30.0.cil` maintain compatibility

Step 7: Enforcing or Permissive Mode

- Enforcing: SELinux **blocks** disallowed access
- Permissive: SELinux **logs** but doesn't block

Check mode:

```bash
getenforce
```

Set temporarily (root only):

```bash
setenforce 0  # permissive
setenforce 1  # enforcing
```

Step 8: Treble Separation (Android 8+)

- Platform vs Vendor policies are split:

Type	Policy	Loaded From
`plat_sepolicy.cil`	Platform	/system/etc/selinux
`vendor_sepolicy.cil`	Vendor	/vendor/etc/selinux
`mapping/*.cil`	Compatibility	/system/etc/selinux/mapping

- Ensures vendor and system can be updated independently.

Step 9: Access Mediation

Whenever any app or process:

- Opens a file
- Sends a binder call
- Creates a socket
- Reads a property

SELinux intercepts the operation → checks against its policy → allows or denies.

📌 Example Flow: App Reading a System File

1. App process runs in `untrusted_app` domain
   Label: `u:r:untrusted_app:s0`
2. File is `/sys/class/backlight/brightness`
   Label: `u:object_r:sysfs_backlight:s0`
3. SELinux checks policy:

   ```cil
   allow untrusted_app sysfs_backlight:file { read open };
   ```

4. If policy doesn't exist:
   - **Access denied**
   - Log entry in dmesg
   - App sees `Permission denied`

🧠 Summary Flow Chart

```text
+----------------------------+
|  Boot Kernel with SELinux  |
+-------------+--------------+
              ↓
+-------------v--------------+
|  /init loads policy files  |
+-------------+--------------+
              ↓
```

```
 +-------------v-------------+
 |  Restorecon: Label files   |
 +-------------+-------------+
               ↓
 +-------------v-------------+
 | Start services via init.rc |
 +-------------+-------------+
               ↓
 +-------------v-------------+
 | Processes run in domains   |
 | (mediaserver, netd, etc.)  |
 +-------------+-------------+
               ↓
 +-------------v-------------+
 | Access checks by SELinux  |
 | via allow rules            |
 +-------------+-------------+
               ↓
 |  Allowed?  |→ Yes → Access granted
 |            |→ No  → Denied + AVC log
```

Would you like:

- A **visual diagram**?
- A **real debug example** from `dmesg`?
- A **test policy rule** walkthrough?

Let me know how you'd like to continue.

77. restorecon explain in details with example

Great! Let's break down `restorecon` in Android SELinux in a clear, detailed way — with examples.

✅ What is `restorecon`?

`restorecon` is a command-line tool that **restores the SELinux context (label)** of a file or directory **based on the SELinux policy and file_contexts rules**.

🧠 Why it's needed?

Sometimes file contexts (labels) might get:

- Removed (due to formatting, updates, wrong creation)
- Set incorrectly (by apps, scripts, bugs)

This causes **access denials** or incorrect behavior.

> `restorecon` fixes these by re-applying the correct label from `file_contexts`.

📦 Example Scenario

Imagine you copy a file into `/data/`:

```bash
cp /system/etc/hosts /data/myhosts
```

Now check the SELinux context:

```bash
ls -Z /data/myhosts
```

❌ You might see:

```makefile
u:object_r:system_file:s0
```

But Android expects `/data/*` files to be `u:object_r:data_file:s0`.

✅ Fix it using `restorecon`

```bash
restorecon /data/myhosts
```

Now:

```bash
ls -Z /data/myhosts
```

Shows:

```makefile
u:object_r:app_data_file:s0
```

✔️ Correct context restored.

🔧 Syntax and Options

```bash
restorecon [options] <file|dir>
```

Common Options:

Option	Description
-R	Recursive for directories
-v	Verbose (shows what it's doing)
-n	Don't change anything (dry-run)
-F	Force relabel, even if already correct

🧪 Example 1: Relabel a file

```bash
restorecon -v /data/myfile
```

Output:

```css
Relabeled /data/myfile from system_file to app_data_file
```

🧪 Example 2: Recursive relabel a directory

```bash
restorecon -Rv /data/app/
```

Output:

```bash
Relabeled /data/app/myapp to u:object_r:app_data_file:s0
Relabeled /data/app/another to u:object_r:app_data_file:s0
```

🔄 When is `restorecon` Used in Android?

📌 During Boot:

In **`init.rc`**, like:

```rc
on post-fs-data
    restorecon_recursive /data
```

This ensures **`/data`**, **`/data/app`**, **`/data/vendor`**, etc. have correct SELinux labels.

📌 After Factory Reset:

If user wipes data, `/data` is reformatted. `restorecon` is used to relabel everything.

📌 In Recovery or Debug Shell:

You may manually run:

```bash
adb shell restorecon -Rv /data
```

🔍 How It Knows the Correct Labels?

It uses the **file_contexts** file:

- `/system/etc/selinux/plat_file_contexts`
- `/vendor/etc/selinux/vendor_file_contexts`

Contains rules like:

```text
/data(/.*)?  u:object_r:app_data_file:s0
/dev/socket(/.*)? u:object_r:socket_device:s0
```

These rules are used to **match the path** and determine correct label.

🚫 What Happens Without `restorecon`?

If a file has the wrong label:

- App can't access it.
- Denial in `logcat`:

  ```text
  avc: denied { read } for pid=1234 ...
  ```

- Example:
  A camera app can't access `/data/vendor/camera` because it's labeled `system_file` instead of `vendor_data_file`.

🧠 Advanced: Check vs Restore

Command	What it does
`ls -Z`	Shows current SELinux context
`matchpathcon /path`	Shows what *should* be the context
`restorecon /path`	Actually fixes it

Example:

```bash
ls -Z /data/myfile
# u:object_r:unlabeled:s0

matchpathcon /data/myfile
# /data/myfile u:object_r:app_data_file:s0

restorecon /data/myfile
# Now correctly labeled.
```

🧠 Summary

Feature	Description
`restorecon`	Fixes SELinux labels for files
Based on	**`file_contexts`** rules
Used during	Boot, OTA, recovery, debugging
Required for	Proper app and daemon access
Alternative	**`chcon`** (manual but non-permanent)

Would you like:

- a full boot-time policy + restorecon diagram?
- help debugging a real **`avc: denied`** issue?

Let me know!

78. How selinux labels/contexts work in policy matching

Great question. Understanding **how SELinux labels/contexts work in policy matching** is **key to debugging Android security denials** and writing policy. Let's break it down **step-by-step** with examples and visuals (described).

🔐 What is a SELinux label (context)?

A **SELinux context** is a **security tag** that helps determine access control. It is made up of:

```pgsql
user:role:type:level
```

Example:

```makefile
u:object_r:vendor_file:s0
```

Parts:

Part	Meaning
`user`	Usually **u** (ignored in Android)
`role`	Usually **`object_r`** for files
`type`	🔥 Main part used in policy matching (like **`vendor_file`**, **`system_server`**)
`level`	MLS/MCS level, usually **`s0`** in Android

⚙️ How Policy Matching Works

SELinux policy defines **access control rules** between a **subject** (process) and an **object** (file/socket/etc).

The rule syntax:

```php-template
allow <subject_type> <object_type>:<class> <permissions>;
```

For example:

```cil
allow system_server vendor_file:file { read write open };
```

This says:
`system_server` (process) can **`read`**, **`write`**, **`open`** a file labeled with **`vendor_file`**.

🧱 The Match Happens on TYPEs

The actual enforcement works by comparing:

1. The **type of the process**
2. The **type of the object (file/socket/etc)**

The SELinux **kernel engine**:

- Checks the subject's **`security_context`** → gets its **`type`**
- Checks the target object's **`context`** → gets its **`type`**

- Checks if there is an **`allow` rule** for that **`<subject_type> <object_type>`** combination

✅ If yes → access allowed
❌ If no → access denied (**`avc: denied`**)

🧪 Example Walkthrough

📁 File:

```bash
ls -Z /vendor/lib/libfoo.so
# u:object_r:vendor_file:s0
```

🧠 `vendor_file` is the object type

🧑‍💻 Process:

```bash
ps -Z | grep system_server
# u:r:system_server:s0
```

`system_server` is the **subject type**

✅ Allowed?

Policy must contain:

```cil
allow system_server vendor_file:file { read open };
```

Otherwise, you'll get:

```logcat
avc: denied { read } for pid=1234 comm="system_server" name="libfoo.so" dev="sda1" ino=12345 scontext=u:r:system_server:s0
tcontext=u:object_r:vendor_file:s0
```

🔄 Where Labels Come From

- For files: via **`file_contexts`**
- For processes: via **`seapp_contexts`**, **`init.rc`**, service declarations
- For sockets: via **`socket_contexts`**
- For properties: via **`property_contexts`**

🔍 Example Matching Rule

Let's say a vendor camera app needs access to a device file:

📂 File:

```bash
/dev/vendor/camera0 → u:object_r:vendor_camera_device:s0
```

🤖 Process:

```bash
ps -Z | grep myvendorcamera
# u:r:vendor_camera_app:s0
```

Policy needed:

```cil
allow vendor_camera_app vendor_camera_device:chr_file { read write open ioctl };
```

If this rule is missing → access denied.

📘 Types vs Attributes

❓ What's the difference?

- **Type** = actual label (`vendor_camera_device`)
- **Attribute** = group of types (like `vendor_data_file_type`, `domain`)

Policy with attributes:

```cil
attribute vendor_data_file_type;
type vendor_camera_device, vendor_data_file_type;

allow vendor_camera_app vendor_data_file_type:file { read write };
```

This allows `vendor_camera_app` to access **any type** with the `vendor_data_file_type` attribute.

✅ Scalable
✅ Easier policy writing
✅ Used heavily in Android

🧠 Summary Flow

```text
[Step 1] File labeled via file_contexts → gets a "type"

[Step 2] Process started via init → gets a "type"

[Step 3] Policy defines: allow <process_type> <file_type>:<class> <perms>;

[Step 4] Access is checked by SELinux kernel engine

[Result] If allow exists → ✅ access
          Else → ❌ avc: denied
```

🧰 Tools to Debug

Tool	What it does
`ls -Z`	Shows current context
`ps -Z`	Shows process context
`audit2allow`	Suggests rules from `dmesg` AVC denials
`sepolicy-analyze`	Analyzes compiled policy

🧪 Bonus Tip: Check Matching with `audit2allow`

```bash
dmesg | grep avc > avc.txt
```

```
audit2allow -i avc.txt
```

Shows suggested rule:

```cil
allow system_server vendor_file:file { read open };
```

Would you like a **visual flow diagram** of this entire process? I can generate one too.

79. How context files used with .te files in selinux android

Great question! Let's break down **how SELinux context files (like `file_contexts`, `property_contexts`, etc.) are used with `.te` policy files** in Android.

🔍 Quick Overview

File Type	Purpose
`.te` files	Define **types**, **domains**, and **rules** (e.g. `allow`, `type`, `attribute`)
Context files (e.g., `file_contexts`)	Map **objects on the system** (files, properties, sockets, services) to **SELinux types**

Together, they ensure that **objects on the system get the right label**, and that the **policy knows how processes interact** with those objects.

🔄 Relationship: .te ↔ context files

> You define **rules and access control** in `.te` files
> You **assign labels** (types) to actual objects in `*_contexts` files

🧱 Step-by-step Example

Let's take a vendor camera daemon as an example.

✅ Step 1: Define a type in `.te`

File: `vendor_camera_app.te`

```cil
type vendor_camera_app, domain;
type vendor_camera_device, dev_type;

allow vendor_camera_app vendor_camera_device:chr_file { read write open };
```

This does 3 things:

- Declares two types: one for the process, one for the device node.
- Tags `vendor_camera_app` as a **domain** (i.e. process)
- Grants access to the device node

✅ Step 2: Assign labels in context files

File: `file_contexts`

```bash
/dev/vendor/camera0    u:object_r:vendor_camera_device:s0
```

This tells the system:

> Label the `/dev/vendor/camera0` file as `vendor_camera_device`

✅ Step 3: Service label in `init.rc` or service_contexts

If you're declaring a service in `init.rc`:

```rc
service vendor.camera /vendor/bin/camdaemon
    class late_start
```

```
    user camera
    seclabel u:r:vendor_camera_app:s0
```

Or in **`service_contexts`** (used by **`servicemanager`**):

```bash
vendor.camera     u:object_r:vendor_camera_service:s0
```

✅ Step 4: At runtime...

- The **camera daemon process** will run with label **`u:r:vendor_camera_app:s0`**
- The **device node** **`/dev/vendor/camera0`** will be labeled **`u:object_r:vendor_camera_device:s0`**
- SELinux will match the access rule from the **`.te`** file:

```cil
allow vendor_camera_app vendor_camera_device:chr_file { read write open };
```

If this rule doesn't exist → you'll see an **`avc: denied`** in the logs.

🔄 Context File Types and their Roles

Context File	What it Labels	Example
`file_contexts`	Files and directories	**`/vendor/bin/foo`** → **`vendor_exec`**
`service_contexts`	**`init.rc`** service names (Android < Q)	**`my_service`** → **`my_service_context`**
`hwservice_contexts`	HIDL hwservice names	**`android.hardware.camera@1.0::ICamera`** → **`hal_camera_hwservice`**
`seapp_contexts`	Apps based on UID/package	UID 10035 → **`untrusted_app`**
`property_contexts`	System properties	**`persist.vendor.foo.*`** → **`vendor_prop`**
`socket_contexts`	UNIX domain sockets	**`/dev/socket/foo`** → **`vendor_socket`**

💡 Summary

Component	Role
`.te` files	Define policy: **`types`**, **`rules`**, **`domains`**
`*_contexts`	Map system objects to **`types`** (for labels)
Runtime Matching	Type of subject (process) ↔ type of object
Enforced by	Kernel SELinux engine

✅ Example Policy+Context Flow

```text
1. You define `type my_daemon, domain;` in `.te`
2. You label the binary path `/vendor/bin/my_daemon` in `file_contexts`
   → `u:object_r:my_daemon_exec:s0`
3. You declare service in `init.rc` with `seclabel u:r:my_daemon:s0`
4. Policy allows:
   `allow my_daemon my_data:file { read write };`
5. Files labeled with `my_data` type using `file_contexts`
6. At boot:
   - Binary is labeled
   - Process runs with `my_daemon`
   - File has `my_data`
   - Kernel checks the rule → access granted
```

Would you like a real Android source tree example (like AOSP **`system_server.te`** + **`file_contexts`**) to go deeper?

80. How selinux policies from system used in vendor?

Excellent question! Let's clearly explain **how SELinux policies from the system (AOSP/platform) are used by the vendor partition**, step by step — with examples.

🔧 Background: Treble and Policy Separation

Since **Android 8.0 (Oreo)** and Project Treble, the **SELinux policy is split** into two independently built parts:

Part	Who builds it	Location	Purpose
Platform (system)	AOSP / Google	`/system`	Core Android services & framework
Vendor	SoC/OEM Vendor	`/vendor`	HALs, device daemons, custom services

But these two must **work together** to form a **complete SELinux policy**.

✅ How Platform Policy Is Used in Vendor

🔹 1. Public Platform Policy is Exported

- The platform exposes a subset of its policy as **"platform-public"**.
- Located in:

```swift
system/sepolicy/public/
system/sepolicy/private/  ← (for internal platform use)
```

- These define types, attributes, and rules **safe for the vendor to use**.

  For example:

```c
# system/sepolicy/public/domain.te
type domain;
typeattribute domain coredomain;
```

These are exposed so vendors can write:

```c
allow my_hal_process domain:process { fork };
```

🔹 2. Attributes Used for Compatibility

- Android transforms **types into versioned attributes** for cross-version compatibility.
- For example:

```c
type logd;
# becomes
typeattributeset logd_v28 (logd)
```

So the vendor can safely refer to `logd_v28` even if `logd` changes in a future Android.

🔹 3. Vendor Policy Includes Platform Public Types

- During build, the vendor includes:

- Platform public types and attributes (`*_vN`)
- Rules using those attributes

Vendor build uses:

```make
BOARD_SEPOLICY_DIRS += device/myvendor/mydevice/sepolicy
```

- These `.te` files use **platform public types** exposed by AOSP.

🔹 4. Policy is Compiled Together (Split Policy)

At build time:

- Platform builds its policy (`plat_sepolicy.cil`)
- Vendor builds its policy (`vendor_sepolicy.cil`)
- Mapping file ensures compatibility (`mapping/28.0.cil`)

At runtime:

```text
Final policy = plat_sepolicy.cil + mapping.cil + vendor_sepolicy.cil
```

🧪 Example: vendor daemon accessing logd

👣 What's in system SEPolicy

File: `system/sepolicy/public/logd.te`

```c
type logd;
typeattribute logd logd_service;
```

Mapping file: `mapping/28.0.cil`

```cil
(typeattributeset logd_v28 (logd))
```

🏭 What vendor writes:

File: `device/vendor/sepolicy/mydaemon.te`

```c
type mydaemon, domain;

allow mydaemon logd_v28:unix_stream_socket connectto;
```

📦 Result:

- Platform exposes `logd`
- Mapping ensures vendor refers to `logd_v28`
- The rule is valid across Android upgrades
- The vendor binary can connect to `logd`

🗂️ Context Files Separation

- Platform provides:
  - `/system/etc/selinux/plat_file_contexts`
  - `/system/etc/selinux/plat_sepolicy.cil`
- Vendor provides:
  - `/vendor/etc/selinux/vendor_file_contexts`
  - `/vendor/etc/selinux/vendor_sepolicy.cil`

They are loaded **together at boot**.

🎯 Summary

Step	Description
1️⃣	Platform exposes safe types/attributes in `system/sepolicy/public/`
2️⃣	Mapping files create versioned attribute wrappers (e.g. `logd_v28`)
3️⃣	Vendor uses only public platform types/attributes (indirectly)
4️⃣	At build time, platform and vendor policies are compiled separately
5️⃣	At runtime, policies are loaded together (`plat + mapping + vendor`)
6️⃣	Compatibility is ensured across Android versions

Would you like to see how this looks in a real AOSP + vendor tree (e.g., from `device/google` or `device/qcom`) with full paths and build steps?

81. Still confused

No worries — SELinux policy interaction between **system (platform)** and **vendor** is one of the trickiest Android topics. Let's break it down with a **real-world analogy** and a **simple, practical example**.

🎯 Big Picture Analogy: "Government (System) vs Local Offices (Vendor)"

Imagine Android as a **country**:

- The **platform** (AOSP system) is the **central government**. It defines national rules (SELinux types like `logd`, `system_server`, etc.).
- The **vendor** is a **local state government** (OEM or SoC vendor like Qualcomm or Samsung). It can write **local laws**, but must follow public national laws.

So:

- The **platform exposes only certain rules publicly** (`public/` policy).
- The **vendor must use only those public rules** to stay compatible with future "national updates" (platform OTAs).
- Both "governments" write their own policy, but it must **compile together** into one working system.

🧪 Minimal Example — How System and Vendor SEPolicy Work Together

🧵 Say we have a platform component:

- System service: `logd`
- Type: `logd`

Platform defines:

```c
// system/sepolicy/public/logd.te
type logd;
typeattribute logd logd_service;
```

It says "there is a logd service, and you can refer to it using the `logd_service` attribute."

🧵 Now, vendor has a custom HAL daemon: `myvendord`

Vendor wants it to talk to logd (e.g., send logs).

Vendor writes:

```c
// device/vendor/sepolicy/myvendord.te
type myvendord, domain;

allow myvendord logd_service:unix_stream_socket connectto;
```

> ✅ Here, the vendor uses `logd_service`, which is **public** and safe to use.

🧠 But What Happens Under the Hood?

When building:

1. **Platform compiles its `.te` and `.cil` to produce:**
    - `plat_sepolicy.cil`
    - `mapping/29.0.cil` (says "`logd_service` means logd in SDK 29")

2. **Vendor compiles its `.te` using:**
   - **Only public types** like **`logd_service`**
   - Resulting in: **`vendor_sepolicy.cil`**
3. **Final boot image merges:**

```text
plat_sepolicy.cil
```

- vendor_sepolicy.cil
- mapping/29.0.cil
  = full policy loaded at boot

```
```

📌 Key Rules to Remember

Thing	Who Writes It	Notes
`logd.te`	Platform	Real type
`logd_service`	Platform (public)	Attribute exposed to vendor
`myvendord.te`	Vendor	Must only use public attributes or own types
`mapping/29.0.cil`	Platform	Ensures compatibility between versions
Final policy	Merged at build time	Loaded at boot

🚨 What If Vendor Used `logd` Instead of `logd_service`?

```c
allow myvendord logd:unix_stream_socket connectto; // ❌ ILLEGAL!
```

- ❌ Compile error: Vendor can't directly use internal platform types like **`logd`**.
- ✅ Must use **`logd_service`** (a public attribute) instead.

📦 File Overview

File	Purpose	Partition
`/system/etc/selinux/plat_sepolicy.cil`	Platform rules	system
`/vendor/etc/selinux/vendor_sepolicy.cil`	Vendor rules	vendor
`/system/etc/selinux/mapping/29.0.cil`	Version mapping	system
`/vendor/etc/selinux/vendor_file_contexts`	File label mappings	vendor

✅ Summary

Step	What Happens
1️⃣	Platform writes public **`.te`** rules (exports types as attributes)
2️⃣	Vendor writes **`.te`** using only public types or its own
3️⃣	Build system compiles platform & vendor SEPolicy separately
4️⃣	Mapping file maps platform types → versioned attributes
5️⃣	Final merged policy loaded by **`init`** at boot
6️⃣	SELinux enforces all access rules from combined policy

Would it help if I showed:

- Full working AOSP + vendor example (with actual file_contexts + .te)?
- Or a diagram of policy flow?

Let me know what angle makes it clearer for you!

82. What is mapping here

Great question!

🔁 What is SELinux Mapping in Android?

In **Android SELinux**, the **mapping** is a special mechanism that connects:

> **Public platform attributes** (like `logd_service_v29`)
> ⬌
> **Actual internal types** (like `logd`) in platform SEPolicy

It helps the **vendor policy** (which can't see platform internals) **stay compatible** with future Android versions — even if the platform changes the internal types!

📦 What Is the Mapping File?

In your AOSP build, it looks like:

```swift
/system/etc/selinux/mapping/29.0.cil
```

- **"29.0"** is the `PLATFORM_SEPOLICY_VERSION`
- It is **auto-generated** during platform SEPolicy build
- It is written in **CIL** (Common Intermediate Language)

🧠 Why Mapping Exists?

Because:

- The **platform can change** internal type names or split/merge types across versions
- But the **vendor SEPolicy is older** (frozen) and written using **public attributes**

Mapping makes sure that:

- `logd_service_v29` still points to `logd`
- Even if platform changes `logd → logd_core + logd_debug` in version 30

🔍 Real Example

Suppose `logd` is a type in platform:

```cil
(type logd)
(typeattributeset logd_service_v29 (logd))
```

And the vendor says:

```cil
(allow myvendor_daemon logd_service_v29 (unix_stream_socket (connectto)))
```

This **works** even though:

- The vendor never sees the actual `logd` type

- Mapping file tells the policy compiler: "`logd_service_v29` means `logd`"

🧱 How It's Structured (CIL Syntax)

In `/system/etc/selinux/mapping/29.0.cil`, you might see:

```cil
(typeattributeset logd_service_v29 (logd))
(expandtypeattribute logd_service_v29 true)
(typeattribute logd_service_v29)
```

This means:

- `logd_service_v29` is a versioned public attribute
- It includes the type `logd`
- It can be used in vendor policy

🧩 Where It's Used

File	Who uses it
`mapping/29.0.cil`	Used by vendor SEPolicy compiler
`plat_sepolicy.cil`	Compiled from platform `.te` files
`vendor_sepolicy.cil`	Compiled from vendor `.te` files
Merged at boot	Loaded by `init` SELinux engine

💡 Summary

Concept	Meaning
`logd`	Internal platform SELinux type (hidden from vendor)
`logd_service`	Public attribute exported for vendor use
`logd_service_v29`	Versioned attribute pointing to `logd` in Android 10/SDK 29
`mapping/29.0.cil`	Maps public versioned attributes → internal types
Why?	Allows vendor policy to be stable and forward-compatible

✅ Without Mapping...

- You'd have to **rewrite vendor policy** after every Android update
- Treble's separation of platform/vendor would break

Would you like to see:

- A full mini working example (type, attribute, mapping)?
- A diagram of the mapping workflow?

Let me know how you'd prefer to visualize this!

83. What is attribute here

Great follow-up!

🔶 What is an Attribute in SELinux?

In **SELinux**, an **`attribute`** is a **label group** — like a **tag** or **category** — that can be applied to multiple **`types`**.

It lets you **write policy rules once**, and apply them to **many types** grouped by that attribute.

🧠 Why Attributes Exist

Writing this:

```te
allow my_app foo:dir read;
allow my_app bar:dir read;
allow my_app baz:dir read;
```

Is long and error-prone.

Instead, we can **group types**:

```cil
(typeattributeset my_dir_types (foo bar baz))
(allow my_app my_dir_types:dir read;)
```

✅ Benefits of Attributes

Feature	What it helps with
✅ DRY	Don't Repeat Yourself
✅ Scalability	Easy to manage large rule sets
✅ Compatibility	Used for versioning in Android

🧱 Syntax

In CIL

```cil
(type foo)
(type bar)
(type baz)

(typeattribute my_dir_types)
(typeattributeset my_dir_types (foo bar baz))

(allow my_app my_dir_types:dir read;)
```

In `.te` (TE language)

```te
type my_app, domain;
type foo, file_type;
type bar, file_type;
type baz, file_type;

attribute my_dir_types;
typeattribute foo my_dir_types;
typeattribute bar my_dir_types;
typeattribute baz my_dir_types;
```

```
allow my_app my_dir_types:dir read;
```

📦 In Android SELinux

Android uses attributes to:

Purpose	Example
Group types by function	`hal_file_type`, `vendor_file_type`
Enable version compatibility	`logd_service_v29`, `sysfs_v31`
Control access by domains	`untrusted_app_domain`, `coredomain`
Vendor-policy flexibility	`vendor_exec_type`, `vendor_socket_type`

🔁 Example (Android-specific)

Platform:

```cil
(type logd)
(typeattributeset logd_service_v29 (logd))
(typeattribute logd_service_v29)
```

Vendor:

```cil
(allow my_vendor_app logd_service_v29:unix_stream_socket connectto)
```

Here, `logd_service_v29` is the **attribute** the vendor uses — they don't see `logd`, but thanks to **mapping**, it still works!

⚠️ Attributes ≠ Types

Term	What is it?
`type`	A single unique label
`attribute`	A tag grouping many types

🔍 Check Attributes in Policy

Use `sesearch`:

```bash
sesearch --attributes -A -s mydomain
```

Or dump everything from compiled policy:

```bash
seinfo -a  # lists attributes
seinfo -x  # shows type → attribute mapping
```

🔚 Summary

Concept	Meaning
`attribute`	A **group label** for **types**
Purpose	Write policy for multiple types at once
Android use	Versioning (`foo_vN`), compatibility, grouping

Concept	Meaning
Mapping	Connects **`attribute`** (public) → **`type`** (platform-internal)

Would you like a **step-by-step attribute demo** in AOSP or a visual diagram?

84. Validate SELinux

bookmark_border
Android strongly encourages OEMs to test their SELinux implementations thoroughly. As manufacturers implement SELinux, they should apply the new policy to a test pool of devices first.

After applying a new policy, make sure SELinux is running in the correct mode on the device by issuing the command getenforce.

This prints the global SELinux mode: either Enforcing or Permissive. To determine the SELinux mode for each domain, you must examine the corresponding files or run the latest version of sepolicy-analyze with the appropriate (-p) flag, present in /platform/system/sepolicy/tools/.

Read denials
Check for errors, which are routed as event logs to dmesg and logcat and are viewable locally on the device. Manufacturers should examine the SELinux output to dmesg on these devices and refine settings prior to public release in permissive mode and eventual switch to enforcing mode. SELinux log messages contain avc: and so may easily be found with grep. It is possible to capture the ongoing denial logs by running cat /proc/kmsg or to capture denial logs from the previous boot by running cat /sys/fs/pstore/console-ramoops.

SELinux error messages are rate-limited after boot complete to avoid swamping the logs. To make sure you see all the relevant messages you can disable this by running adb shell auditctl -r 0.

With this output, manufacturers can readily identify when system users or components are in violation of SELinux policy. Manufacturers can then repair this bad behavior, either by changes to the software, SELinux policy, or both.

Specifically, these log messages indicate what processes would fail under enforcing mode and why. Here is an example:

```
avc: denied  { connectto } for  pid=2671 comm="ping" path="/dev/socket/dnsproxyd"
scontext=u:r:shell:s0 tcontext=u:r:netd:s0 tclass=unix_stream_socket
```

Interpret this output like so:

The { connectto } above represents the action being taken. Together with the tclass at the end (unix_stream_socket), it tells you roughly what was being done to what. In this case, something was trying to connect to a unix stream socket.
The scontext (u:r:shell:s0) tells you what context initiated the action. In this case this is something running as the shell.
The tcontext (u:r:netd:s0) tells you the context of the action's target. In this case, that's a unix_stream_socket owned by netd.
The comm="ping" at the top gives you an additional hint about what was being run at the time the denial was generated. In this case, it's a pretty good hint.
Another example:

```
adb shell su root dmesg | grep 'avc: '
```

Output:

```
<5> type=1400 audit: avc:  denied  { read write } for  pid=177
comm="rmt_storage" name="mem" dev="tmpfs" ino=6004 scontext=u:r:rmt:s0
tcontext=u:object_r:kmem_device:s0 tclass=chr_file
```

Here are the key elements from this denial:

Action - the attempted action is highlighted in brackets, read write or setenforce.
Actor - The scontext (source context) entry represents the actor, in this case the rmt_storage daemon.
Object - The tcontext (target context) entry represents the object being acted upon, in this case kmem.
Result - The tclass (target class) entry indicates the type of object being acted upon, in this case a chr_file (character device).
Dump user and kernel stacks
In some cases, the information contained in the event log isn't sufficient to pinpoint the origin of the denial. It is often useful to gather the call chain, including kernel and userspace, to better understand why the denial occurred.

Recent kernels define a tracepoint named avc:selinux_audited. Use Android simpleperf to enable this tracepoint and capture the callchain.

Supported configuration
Linux kernel >= 5.10, in particular Android Common Kernel branches mainline and android12-5.10 are supported. The android12-5.4 branch is also supported. You may use simpleperf to determine if the tracepoint is defined on your device: adb root && adb shell simpleperf list | grep avc:selinux_audited. For other kernel versions, you may cherry pick commits dd81662 and 30969bc.
It should be possible to reproduce the event you are debugging. Boot time events aren't supported using simpleperf; however you may still be

able to restart the service to trigger the event.
Capture the call chain
The first step is to record the event using simpleperf record:

```
adb shell -t "cd /data/local/tmp && su root simpleperf record -a -g -e avc:selinux_audited"
```

Then, the event that caused the denial should be triggered. After that, the recording should be stopped. In this example, by using Ctrl-c, the sample should have been captured:

```
^Csimpleperf I cmd_record.cpp:751] Samples recorded: 1. Samples lost: 0.
```

Finally, simpleperf report may be used to inspect the captured stacktrace. For instance:

```
adb shell -t "cd /data/local/tmp && su root simpleperf report -g --full-callgraph"
[...]
Children  Self     Command  Pid   Tid   Shared Object                                 Symbol
100.00%   0.00%    dmesg    3318  3318  /apex/com.android.runtime/lib64/bionic/libc.so  __libc_init
       |
       -- __libc_init
          |
           -- main
              toybox_main
              toy_exec_which
              dmesg_main
              klogctl
              entry_SYSCALL_64_after_hwframe
              do_syscall_64
              __x64_sys_syslog
              do_syslog
              selinux_syslog
              slow_avc_audit
              common_lsm_audit
              avc_audit_post_callback
              avc_audit_post_callback
```

The call chain above is a unified kernel and userspace call chain. It gives you with a better view of the code flow by starting the trace from userspace all the way down to the kernel where the denial happens. For more information on simpleperf, see the Simpleperf Executable commands reference

Switch to permissive
Caution: Permissive mode isn't supported on production devices. CTS tests confirm enforcing mode is enabled.
SELinux enforcement can be disabled with adb on userdebug or eng builds. To do so, first switch ADB to root by running adb root. Then, to disable SELinux enforcement, run:

```
adb shell setenforce 0
```

Or at the kernel command line (during early device bring-up):

```
androidboot.selinux=permissive
androidboot.selinux=enforcing
```

Or through bootconfig in Android 12:

```
androidboot.selinux=permissive
androidboot.selinux=enforcing
```

Use audit2allow
Note: audit2allow isn't provided as part of AOSP anymore. Use the package supplied by your Linux distribution (package policycoreutils-python-utils on Debian and Ubuntu).
The audit2allow tool takes dmesg denials and converts them into corresponding SELinux policy statements. As such, it can greatly speed SELinux development.

To use it, run:

```
adb pull /sys/fs/selinux/policy
```

adb logcat -b events -d | audit2allow -p policy
Note: Running these commands doesn't change bugreport.txt because all logs are already there, including the ones from before the last reboot. On devices running an OTA or development flash, old and new violations are mixed until another reboot. To address this, reboot the device again or filter console-ramoops and LAST_LOGCAT from your bugreport.
Nevertheless, care must be taken to examine each potential addition for overreaching permissions. For example, feeding audit2allow the rmt_storage denial shown earlier results in the following suggested SELinux policy statement:

#============= shell ==============
allow shell kernel:security setenforce;
#============= rmt ==============
allow rmt kmem_device:chr_file { read write };
This would grant rmt the ability to write kernel memory, a glaring security hole. Often the audit2allow statements are only a starting point. After employing these statements, you may need to change the source domain and the label of the target, as well as incorporate proper macros, to arrive at a good policy. Sometimes the denial being examined should not result in any policy changes at all; rather the offending app should be changed.

Here is a simplified **step-by-step explanation** of how to **validate SELinux on Android**, including **`getenforce`**, denial analysis, **`audit2allow`**, and debugging with **`simpleperf`**.

✅ Step-by-Step SELinux Validation in Android

🔹 Step 1: Check SELinux Mode

Check if SELinux is running in **Enforcing** or **Permissive** mode:

```bash
adb shell getenforce
```

- **`Enforcing`**: SELinux is blocking unauthorized access.
- **`Permissive`**: SELinux logs violations but does not block.

Use **Permissive mode** for debugging:

```bash
adb shell setenforce 0
```

✅ **Tip**: Only works on **`userdebug`** or **`eng`** builds.

🔹 Step 2: Trigger Denials (Run Test Cases)

After applying new policies or app changes, run your app or system actions that might interact with protected resources.

🔹 Step 3: Read SELinux Denials (AVC Logs)

Search denial logs in **`dmesg`** or **`logcat`**:

```bash
adb shell dmesg | grep 'avc:'
adb logcat | grep 'avc:'
```

Sample denial log:

```bash
avc: denied { connectto } for pid=2671 comm="ping" path="/dev/socket/dnsproxyd"
scontext=u:r:shell:s0 tcontext=u:r:netd:s0 tclass=unix_stream_socket
```

📖 **What it means:**

Part	Meaning
`scontext`	Source context (who tried)
`tcontext`	Target context (what was targeted)
`tclass`	Object type (e.g., socket, file)
`{ connectto }`	Access being denied

🔹 Step 4: Disable Audit Log Rate Limiting (optional)

To avoid missing AVC logs:

```bash
adb shell auditctl -r 0
```

🔹 Step 5: Use `audit2allow` to Suggest Rules (on host)

1. Pull the SELinux policy:

```bash
adb pull /sys/fs/selinux/policy
```

2. Use audit2allow on PC:

```bash
adb logcat -b events -d | audit2allow -p policy
```

Example output:

```te
allow rmt kmem_device:chr_file { read write };
```

⚠️ **Dangerous permissions like `kmem_device` access should NOT be blindly accepted.**

🔹 Step 6: Apply Clean Policy Fixes

Now update `.te` or `.cil` files accordingly with **reviewed** allow rules, or correct the `context` labels.

Also use **macros** instead of raw allow statements when possible:

```te
allow domain sysfs_type:dir r_dir_perms;
```

🔹 Step 7: Capture Call Stack (Optional Debug)

If the log doesn't help enough, capture the **callchain** using `simpleperf`:

```bash
adb shell "cd /data/local/tmp && su root simpleperf record -a -g -e avc:selinux_audited"
# Trigger the action
# Stop recording (Ctrl+C if using interactive shell)
adb shell "cd /data/local/tmp && su root simpleperf report -g --full-callgraph"
```

This shows **userspace + kernel** trace.

🔹 Step 8: Re-test in Enforcing Mode

Switch back:

```bash
adb shell setenforce 1
```

Test if everything still works. If yes, policy is valid.

🚦 Summary Table

Step	Command/Tool	Purpose
Check SELinux mode	`getenforce`	Check enforcing/permissive
View denials	`dmesg`, `logcat`, `grep 'avc:'`	See violations
Disable rate-limit	`auditctl -r 0`	Show all logs
Suggest rules	`audit2allow -p policy`	Generate allow statements
Record stacktrace	`simpleperf record/report`	Debug who/what caused denial
Apply fixes	Update `.te`, `.cil`, use macros	Create clean policy
Re-test	`setenforce 1`	Make sure it works in enforcing mode

Would you like a full example (denial → audit2allow → fix in `.te`)?

85. ↪
adb shell auditctl -r 0
explain this

Sure! Let's explain this command:

🔹 Command:

```bash
adb shell auditctl -r 0
```

🔹 What it does:

This command **disables the SELinux audit log rate limiting** on the Android device.

Normally, Android **limits the number of denial messages** (`avc: denied`) printed to logs **after boot completes**, to avoid spamming the logs with repeated or noisy messages.

By setting:

```bash
-r 0
```

You are telling the audit system:

> "Don't limit any denial logs — log all AVC denials no matter how frequent."

🔹 Why it's useful:

When you're debugging SELinux, you **need to see every denial** to properly fix policy. Rate-limiting would hide repeated violations, making it hard to debug.

🔹 Example:

❌ With rate limiting:

You might see only **one denial** even if the problem happens **100 times**.

✅ After:

```bash
adb shell auditctl -r 0
```

You will see **all 100 denials**, which helps you:

- See patterns
- Use `audit2allow` more accurately
- Know if your fix really worked

🔹 Notes:

- This only works on `userdebug` or `eng` builds.
- It's a **temporary runtime change**, doesn't persist after reboot.

Would you like to see how to **re-enable rate-limiting** or how to use `auditctl` for other audit settings?

86. ↪
🔹 Step 5: Use `audit2allow` to Suggest Rules (on host)
this one

Great! Let's now explain in **simple and clear terms** what Step 5 – using **`audit2allow`** – means, with an **example**:

🔹 What is `audit2allow`?

`audit2allow` is a **tool** that:

- Takes **SELinux denial logs** (like **`avc: denied ...`**)
- And **generates suggested SELinux policy rules** to allow the denied actions.

✅ Why use `audit2allow`?

When you see logs like:

```arduino
avc: denied { read write } for pid=1234 comm="my_app" ...
```

It means your app or service is **blocked** by SELinux.

Instead of **manually writing rules**, you can use **`audit2allow`** to automatically generate a **starting policy fix**.

🛠️ Steps to Use `audit2allow`

🧑‍💻 Example workflow:

1. **Pull current policy from device** (needed for context):

```bash
adb pull /sys/fs/selinux/policy
```

2. **Get denial logs** (from device logs):

```bash
adb logcat -b events -d | audit2allow -p policy
```

📝 What's happening:

- **`adb logcat -b events -d`** fetches SELinux denials (**`avc: denied`**) from logs.
- **`audit2allow -p policy`** reads those and suggests rules.

📌 Example

Say this is your log:

```bash
avc: denied { connectto } for pid=1234 comm="ping" path="/dev/socket/dnsproxyd"
scontext=u:r:shell:s0 tcontext=u:r:netd:s0 tclass=unix_stream_socket
```

Run:

```bash
adb pull /sys/fs/selinux/policy
adb logcat -b events -d | audit2allow -p policy
```

You may get:

```te
#============= shell ==============
allow shell netd:unix_stream_socket connectto;
```

👆 This is the suggested rule to allow **`shell`** domain to **`connectto`** the **`netd`** domain's socket.

⚠️ Important Warning

Just because **`audit2allow`** suggests a rule doesn't mean you should blindly allow it!

Example:

If it says:

```te
allow rmt kmem_device:chr_file { read write };
```

It means: "Allow rmt service to read/write kernel memory" ❌❌
That's dangerous. You might need to **fix the service** instead of allowing it.

✅ Good Practice

Use **`audit2allow`** as a:

> 🔍 **Guide or hint**, not a final answer.

Then:

- Review the suggestion
- Replace raw rules with proper **macros**
- Add it to the correct **`.te`** file

Would you like me to show how to take such a denial and **manually write a clean rule using macros**?

87. Write SELinux policy

bookmark_border
The Android Open Source Project (AOSP) provides a solid base policy for the apps and services that are common across all Android devices. Contributors to AOSP regularly refine this policy. The core policy is expected to make up about 90–95% of the final on-device policy with device-specific customizations making up the remaining 5–10%. This article focuses on these device-specific customizations, how to write device-specific policy, and some of the pitfalls to avoid along the way.

Device bringup
While writing device-specific policy, follow these steps.

Run in permissive mode
When a device is in permissive mode, denials are logged but not enforced. Permissive mode is important for two reasons:

Permissive mode ensures that policy bringup doesn't delay other early device bringup tasks.
An enforced denial may mask other denials. For example, file access typically entails a directory search, file open, then file read. In enforcing mode, only the directory search denial would occur. Permissive mode ensures all denials are seen.
The simplest way to put a device into permissive mode is using the kernel command line. This can be added to the device's BoardConfig.mk file: platform/device/<vendor>/<target>/BoardConfig.mk. After modifying the command line, perform make clean, then make bootimage, and flash the new boot image.

After that, confirm permissive mode with:

adb shell getenforce
Two weeks is a reasonable amount of time to be in global permissive mode. After addressing the majority of denials, move back into enforcing mode and address bugs as they come in. Domains still producing denials or services still under heavy development can be temporarily put into permissive mode, but move them back to enforcing mode as soon as possible.

Enforce early
In enforcing mode, denials are both logged and enforced. It is a best practice to get your device into enforcing mode as early as possible. Waiting to create and enforce device-specific policy often results in a buggy product and a bad user experience. Start early enough to participate in dogfooding and ensure full test coverage of functionality in real world usage. Starting early ensures security concerns inform design decisions. Conversely, granting permissions based solely on observed denials is an unsafe approach. Use this time to perform a security audit of the device and file bugs against behavior that shouldn't be allowed.

Remove or delete existing policy
There are a number of good reasons to create device-specific policy from scratch on a new device, which include:

Security auditing
Overly permissive policy
Policy size reduction
Dead policy
Address denials of core services
Denials generated by core services are typically addressed by file labeling. For example:

avc: denied { open } for pid=1003 comm=”mediaserver” path="/dev/kgsl-3d0”
dev="tmpfs" scontext=u:r:mediaserver:s0 tcontext=u:object_r:device:s0
tclass=chr_file permissive=1
avc: denied { read write } for pid=1003 name="kgsl-3d0" dev="tmpfs"
scontext=u:r:mediaserver:s0
tcontext=u:object_r:device:s0 tclass=chr_file permissive=1
is completely addressed by properly labeling /dev/kgsl-3d0. In this example, tcontext is device. This represents a default context where everything in /dev receives the “ device” label unless a more specific label is assigned. Simply accepting the output from audit2allow here would result in an incorrect and overly permissive rule.

To solve this kind of problem, give the file a more specific label, which in this case is gpu_device. No further permissions are needed as the mediaserver already has necessary permissions in core policy to access the gpu_device.

Other device-specific files that should be labeled with types predefined in core policy:

block devices

audio devices
video devices
sensors
nfc
gps_device
files in /sys
files in /proc
In general, granting permissions to default labels is wrong. Many of these permissions are disallowed by neverallow rules, but even when not explicitly disallowed, best practice is to provide a specific label.

Label new services and address denials
Init-launched services are required to run in their own SELinux domains. The following example puts service “foo” into its own SELinux domain and grants it permissions.

The service is launched in our device's init.device.rc file as:

```
service foo /system/bin/foo
    class core
```

Create a new domain "foo"
Create the file device/manufacturer/device-name/sepolicy/foo.te with the following contents:

```
# foo service
type foo, domain;
type foo_exec, exec_type, file_type;

init_daemon_domain(foo)
```

This is the initial template for the foo SELinux domain, to which you can add rules based on the specific operations performed by that executable.

Label /system/bin/foo
Add the following to device/manufacturer/device-name/sepolicy/file_contexts:

```
/system/bin/foo   u:object_r:foo_exec:s0
```

This makes sure the executable is properly labeled so SELinux runs the service in the proper domain.

Build and flash the boot and system images.
Refine the SELinux rules for the domain.
Use denials to determine the required permissions. The audit2allow tool provides good guidelines, but only use it to inform policy writing. Don't just copy the output.

Switch back to enforcing mode
It's fine to troubleshoot in permissive mode, but switch back to enforcing mode as early as possible and try to remain there.

Common mistakes
Here are a few solutions for common mistakes that happen when writing device-specific policies.

Overuse of negation
The following example rule is like locking the front door but leaving the windows open:

```
allow { domain -untrusted_app } scary_debug_device:chr_file rw_file_perms
```

The intent is clear: everyone but third-party apps may have access to the debug device.

The rule is flawed in a few ways. The exclusion of untrusted_app is trivial to work around because all apps may optionally run services in the isolated_app domain. Likewise, if new domains for third-party apps are added to AOSP, they also have access to scary_debug_device. The rule is overly permissive. Most domains won't benefit from having access to this debugging tool. The rule should have been written to allow only the domains that require access.

Debug features in production
Debug features shouldn't be present on production builds nor should their policy.

The simplest alternative is to only allow the debug feature when SELinux is disabled on eng/userdebug builds, such as adb root and adb shell setenforce 0.

Another safe alternative is to enclose debug permissions in a userdebug_or_eng statement.

Policy size explosion
Characterizing SEAndroid Policies in the Wild describes a concerning trend in the growth of device policy customizations. Device-specific policy should account for 5–10% of the overall policy running on a device. Customizations in the 20%+ range almost certainly contain over privileged domains and dead policy.

Unnecessarily large policy:

Takes a double hit on memory as the policy sits in the ramdisk and is also loaded into kernel memory.
Wastes disk space by necessitating a larger bootimage.
Affects runtime policy lookup times.
The following example shows two devices where the manufacturer-specific policy comprised 50% and 40% of the on-device policy. A rewrite of the policy yielded substantial security improvements with no loss in functionality, as shown below. (AOSP devices Shamu and Flounder are included for comparison.)

Figure 1: Comparison of device-specific policy size after security audit.

Figure 1. Comparison of device-specific policy size after security audit.

In both cases, the policy was dramatically reduced both in size and in number of permissions. The decrease in policy size is almost entirely due to removing unnecessary permissions, many of which were likely rules generated by audit2allow that were indiscriminately added to the policy. Dead domains were also an issue for both devices.

Grant the dac_override capability
A dac_override denial means that the offending process is attempting to access a file with the incorrect unix user/group/world permissions. The proper solution is almost never to grant the dac_override permission. Instead change the unix permissions on the file or process. A few domains such as init, vold, and installd genuinely need the ability to override unix file permissions to access other processes' files. See Dan Walsh's blog for a more in-depth explanation.

Here's a **simple step-by-step explanation** of how to **write and refine SELinux policy** for your **Android device** based on the document you provided:

🔐 Goal:

To create **device-specific SELinux policy** safely and correctly during Android device bring-up, ensuring **security** and **functionality**.

🔁 Step-by-Step Guide:

✅ Step 1: Start in Permissive Mode

- In **permissive**, SELinux logs denials (AVC messages) but **doesn't block** anything.
- Helps catch **all policy issues** early during bring-up.

How to do this:
Edit your device's **`BoardConfig.mk`**:

```makefile
BOARD_KERNEL_CMDLINE += androidboot.selinux=permissive
```

Then:

```bash
make clean
make bootimage
fastboot flash boot boot.img
adb shell getenforce  # should return "Permissive"
```

✅ Step 2: Fix Denials Using Logs (Not Blindly)

- Run the device and use it normally.

- Collect denials:

```bash
adb shell dmesg | grep avc:
```

- DO NOT just allow them using `audit2allow` blindly. Instead:
  - **Label the files properly**
  - **Refactor your policy** to follow least privilege.

✅ Step 3: Label Device Nodes Correctly

Example denial:

```lua
avc: denied { open } for comm="mediaserver" path="/dev/kgsl-3d0"
```

Fix:

- Label `/dev/kgsl-3d0` with a more specific type like `gpu_device`, **not** generic `device`.

Edit your `file_contexts`:

```plaintext
/dev/kgsl-3d0    u:object_r:gpu_device:s0
```

✅ Step 4: Create a Custom Domain for Your Service

Let's say your service is:

```plaintext
service foo /system/bin/foo
```

1. Create new domain:
Create file:

```php-template
device/<vendor>/<device>/sepolicy/foo.te
```

Content:

```te
type foo, domain;
type foo_exec, exec_type, file_type;

init_daemon_domain(foo)
```

2. Label the executable in `file_contexts`:

```plaintext
/system/bin/foo    u:object_r:foo_exec:s0
```

✅ Step 5: Build, Flash, and Observe

Rebuild and flash system and boot images.

✅ Step 6: Refine Policies Using AVC Denials

Use:

```bash
adb logcat | grep avc:
```

Or:

```bash
adb logcat -b events -d | audit2allow -p /sys/fs/selinux/policy
```

Carefully interpret the output and write correct rules in `.te` file.

✅ Step 7: Switch Back to Enforcing Mode

Once most issues are solved:

```bash
adb shell setenforce 1
```

Or update your **`BoardConfig.mk`**:

```makefile
BOARD_KERNEL_CMDLINE += androidboot.selinux=enforcing
```

🧠 Tips & Common Mistakes:

🚫 Mistake: Overuse of Negation

Bad:

```te
allow { domain -untrusted_app } scary_debug_device:chr_file rw_file_perms;
```

✅ Instead: allow **only specific domains** that need access.

🚫 Mistake: Allowing Debug in Production

✅ Use:

```te
userdebug_or_eng(`
    allow foo debug_device:chr_file rw_file_perms;
')
```

🚫 Mistake: Granting `dac_override`

Bad:

```te
allow foo self:capability dac_override;
```

✅ Instead: Fix Linux file permissions (owner, group, mode).

🚫 Mistake: Overly Large Policies

- Keep **device-specific policy** to **<10%** of the total.

- Remove unused types, domains, and deny rules.

✅ Summary

Step	What to Do	Why
1	Start in permissive mode	Log all denials without blocking
2	Observe AVC denials	Understand what's blocked
3	Label files properly	Avoid overly broad permissions
4	Create new domain	Isolate service securely
5	Flash and test	Run your service and log denials
6	Refine **`.te`** and **`file_contexts`**	Based on AVC logs
7	Move to enforcing mode	Secure production device

If you want, I can help you **write a real SELinux policy** for your own service step-by-step. Just tell me the name and what it does.

88. The init process has nearly unrestricted permissions and uses input scripts from both the system and vendor partitions to initialize the system during the boot process. This access causes a huge hole in the Treble system/vendor split, as vendor scripts may instruct init to access files, properties, etc. that don't form part of the stable system-vendor application binary interface (ABI).

Vendor init is designed to close this hole by using a separate security-enhanced Linux (SELinux) domain vendor_init to run commands found in /vendor with vendor-specific permissions.

Mechanism
Vendor init forks a subprocess of init early in the boot process with the SELinux context u:r:vendor_init:s0. This SELinux context has considerably fewer permissions than the default init context and its access is confined to files, properties, etc. that are either vendor-specific or part of the stable system-vendor ABI.

Init checks each script it loads to see if its path starts with /vendor and if so, tags it with an indication that its commands must be run in the vendor init context. Each init builtin is annotated with a boolean that specifies whether or not the command must be run in the vendor init subprocess:

Most commands that access the file system are annotated to run in the vendor init subprocess and are therefore subjected to the vendor init SEPolicy.
Most commands that impact internal init state (e.g., starting and stopping services) are run within the normal init process. These commands are made aware that a vendor script is calling them to do their own non-SELinux permissions handling.
The main processing loop of init contains a check that if a command is annotated to run in the vendor subprocess and originates from a vendor script, that command is sent through inter-process communication (IPC) to the vendor init subprocess, which runs the command and sends the result back to init.

Use vendor init
Vendor init is enabled by default and its restrictions apply to all init scripts present in the /vendor partition. Vendor init should be transparent to vendors whose scripts are already not accessing system only files, properties, etc.

However, if commands in a given vendor script violate the vendor init restrictions, the commands fail. Failing commands have a line in the kernel log (visible with dmesg) from init indicating failure. An SELinux audit accompanies any failing command that failed due to the SELinux policy.
Example of a failure including an SELinux audit:

type=1400 audit(1511821362.996:9): avc: denied { search } for pid=540 comm="init" name="nfc" dev="sda45" ino=1310721 scontext=u:r:vendor_init:s0 tcontext=u:object_r:nfc_data_file:s0 tclass=dir permissive=0
init: Command 'write /data/nfc/bad_file_access 1234' action=boot (/vendor/etc/init/hw/init.walleye.rc:422) took 2ms and failed: Unable to write to file '/data/nfc/bad_file_access': open() failed: Permission denied
If a command fails, there are two options:

If the command is failing due to an intended restriction (such as if the command is accessing a system file or property), the command must be re-implemented in a Treble-friendly way, going through only stable interfaces. Neverallow rules prevent adding permissions to access system files that aren't part of the stable system-vendor ABI.
If the SELinux label is new and isn't already granted permissions in the system vendor_init.te nor excluded permissions through the neverallow rules, the new label may be granted permissions in the device-specific vendor_init.te.
For devices launching before Android 9, the neverallows rules may be bypassed by adding the data_between_core_and_vendor_violators typeattribute to the device-specific vendor_init.te file.

Code locations
The bulk of the logic for the vendor init IPC is in system/core/init/subcontext.cpp.

The table of commands is in the BuiltinFunctionMap class in system/core/init/builtins.cpp and includes annotations that indicate if the command must run in the vendor init subprocess.

The SEPolicy for vendor init is split across the private (system/sepolicy/private/vendor_init.te) and public (system/sepolicy/public/vendor_init.te) directories in system/sepolicy.

Here's a **simplified explanation** of **Vendor Init** in Android with **step-by-step flow**, **why it was introduced**, and **how it works**, including **examples** to help you understand better.

🚨 Problem Before Vendor Init

Android's **`init`** process:

- Has **full root** + SELinux **`u:r:init:s0`** context (very powerful).
- Reads init scripts from **both `/system` and `/vendor`**.
- That means **vendor scripts could access or modify system internals** (breaking the **Treble** system/vendor separation).

  🖍️ This **breaks Treble** because vendors might depend on internal, unstable system behaviors instead of using **stable vendor interfaces**.

✅ Solution: Vendor Init

Android introduced **Vendor Init** to **limit what vendor scripts can do** by running them in a **restricted SELinux domain**.

🔄 Step-by-Step Flow of Vendor Init

1. During early boot, `init` process forks a subprocess

- This subprocess runs in **`vendor_init` domain**:

  ```yaml
  SELinux context: u:r:vendor_init:s0
  ```

- This domain has **limited permissions**, unlike **`init`** (u:r:init:s0).

---

2. Init loads all init.rc files

- If the path starts with **`/vendor`**, init **marks commands** in that script to run inside the **vendor subprocess**.

  Example:

  ```swift
  /vendor/etc/init/hw/init.vendor.rc
  ```

---

3. Each init command is annotated in AOSP code

- It tells init whether it should:
  - Run in main **`init`** process (e.g., **`start`**, **`stop`**)
  - Or in **vendor_init subprocess** (e.g., **`mkdir`**, **`write`**, **`chmod`**)

  ```cpp
  // builtins.cpp (AOSP)
  {"write", {1, 2, &do_write}, SubcontextType::Vendor},
  {"start", {1, 1, &do_start}, SubcontextType::Init},
  ```

---

4. IPC communication

- Init sends the command to the **vendor subprocess** if needed.
- Subprocess runs the command using **limited vendor SELinux policy**.
- It sends the **result (success/failure)** back to the main init.

🔐 Why It's Safe?

Vendor subprocess can **only access**:

- **Vendor-specific files**
- **Stable vendor-system interfaces** (defined by Treble)

- Can't touch **`/system`**, system props, or core system files

🧪 Example Failure (When vendor tries to break rules)

Suppose a vendor script tries to write to **`/data/nfc/bad_file_access`**:

```bash
write /data/nfc/bad_file_access 1234
```

This causes denial:

```log
avc: denied { search } for ...
scontext=u:r:vendor_init:s0 tcontext=u:object_r:nfc_data_file:s0 ...
```

Meaning:

- Command was executed from vendor script.
- SELinux denied access because **`vendor_init`** domain is **not allowed to write** to **`nfc_data_file`**.

🚧 How to Fix This?

Option 1: Change your design

- Use **stable interfaces** to communicate with system.
- E.g., don't write directly to system-owned files.

Option 2: Extend vendor policy (if allowed)

- Add permission in your **device's `vendor_init.te`**:

```te
allow vendor_init nfc_data_file:dir search;
allow vendor_init nfc_data_file:file write;
```

BUT:

- Fails if **`nfc_data_file`** is considered **system-owned** (because of **neverallow rules** in AOSP).
- Only safe if label is **vendor-owned** or not in conflict with Treble.

🗂️ Related AOSP Code Locations

Purpose	Path
Vendor Init IPC	**`system/core/init/subcontext.cpp`**
Builtin commands table	**`system/core/init/builtins.cpp`**
Vendor Init SELinux policy	**`system/sepolicy/public/vendor_init.te`** and **`private/vendor_init.te`**

🚩 Summary Table

Item	Traditional Init	Vendor Init
SELinux domain	**`u:r:init:s0`**	**`u:r:vendor_init:s0`**
Permissions	Full root	Restricted to vendor/ABI
Purpose	Boot system	Run vendor-specific scripts safely
Treble safe?	❌ No	✅ Yes
Runs scripts from	**`/system`**, **`/vendor`**	Only **`/vendor`** (in subprocess)

✅ Final Notes

- **Vendor Init** = Safe sandbox for vendor startup logic.
- Keeps **system and vendor partition separate** as required by Treble.
- If you see failures in `/vendor/etc/init/` scripts, always check:
  - SELinux logs (`dmesg`, `logcat`)
  - Whether the command is allowed in `vendor_init` domain
  - If not, either **refactor the script** or **carefully update** `vendor_init.te`.

Would you like an actual example `.rc` script and `.te` fix walkthrough?

89. tools used in selinux

Here's a list of essential **tools used in SELinux**, especially in the context of **Android** development and debugging, along with their **purpose and examples**:

🛠️ SELinux Tools (Android & Linux)

Tool	Purpose	Example Usage
`getenforce`	Shows current SELinux mode (Enforcing / Permissive / Disabled)	`adb shell getenforce`
`setenforce`	Sets SELinux mode at runtime (only on userdebug/eng)	`adb shell setenforce 0` (set to permissive)
`sestatus`	Shows detailed SELinux status (Linux only)	`sestatus`
`audit2allow`	Converts denial logs into allow rules	`audit2allow -p policy < log.txt`
`audit2why`	Explains why access was denied	`audit2why < denial.log`
`sepolicy-analyze`	Analyzes `.te` policies, shows unused types, etc.	`sepolicy-analyze policy allow`
`checkpolicy`	Compiles SELinux `.te` source to binary `.policy`	`checkpolicy -M -c 30 mypolicy.te`
`checkcon`	Checks current SELinux label of a file or process	`ls -Z /system/bin/app_process`
`restorecon`	Restores file label based on file_contexts	`restorecon -v /data/myfile`
`chcon`	Manually change a file's SELinux context	`chcon u:object_r:vendor_file:s0 myfile`
`semodule`	Installs or manages SELinux modules	`semodule -i mypolicy.pp`
`simpleperf`	Captures kernel + user call stacks (Android)	`adb shell simpleperf record -a -g -e avc:selinux_audited`
`dmesg`	Kernel log (shows AVC denials)	`dmesg
`logcat`	Android logs (also includes SELinux AVC logs)	`adb logcat
`cat /sys/fs/pstore/console-ramoops`	Shows crash/denial logs from last boot	`adb shell cat /sys/fs/pstore/console-ramoops`

🗂️ Important Files Used with Tools

File	Purpose
`/sys/fs/selinux/policy`	Current policy binary file (used with audit2allow)
`file_contexts`	Maps file paths to SELinux labels
`sepolicy`	Compiled policy file
`*.te`	Type enforcement policy source files
`*.fc`	File context mapping files
`*.pp`	Binary policy modules

🧪 Example: How to Use the Tools Together

1. ✅ **Check SELinux status**:

```bash
adb shell getenforce
```

2. 🧾 **See denials**:

```bash
adb shell dmesg | grep avc
```

3. 🧠 **Understand the denial**:

```bash
adb logcat | grep avc > denial.log
audit2why < denial.log
```

4. 📜 **Generate suggested allow rule**:

```bash
adb pull /sys/fs/selinux/policy
adb logcat -b events -d | audit2allow -p policy
```

5. 🔧 **Fix file context issue**:

```bash
restorecon -v /my/path/to/fix
```

6. 📈 **Advanced: Use simpleperf for callstack**:

```bash
adb shell simpleperf record -a -g -e avc:selinux_audited
```

🔐 Summary

These tools help in:

- Inspecting SELinux mode and status
- Diagnosing AVC denials
- Debugging context mismatches
- Writing and verifying policy rules
- Maintaining strict yet functional policies

Would you like me to show a mini workflow using these tools with a real denial and **`.te`** file fix?