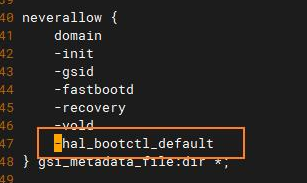
**根据log日志生成：**

**audit2allow -i a1.txt -o a1.te -p ../out/target/product/msmnile\_gvmq/recovery/root/sepolicy –e**

**file\_context末尾添加后，要加一个回车，否则编译报错.原因可能是因为编译需要将几个文件合起来。如果不加回车，当前文件末尾一行和下一个文件开头合到一行去了。**

**问题1**





**因为规则冲突，将hal\_bootctl\_default添加到gsid.te，排除这个type。**

**因gsid.te在private里，hal\_bootctl\_default定义在vendor里，故编译报错unknown type。**

**因为又不能直接把type hal\_bootctl\_default移到private或public。添加一个属性：**

**vim public/attributes：**

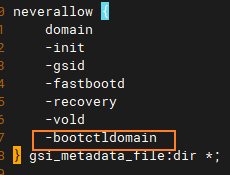
**attribute bootctldomain;**

**vim ./system/sepolicy/vendor/hal\_bootctl\_default.te：**

**type hal\_bootctl\_default, domain；改为-》**

**type hal\_bootctl\_default, domain, bootctldomain;**

**然后gsid.te里排除这个属性即可：**



**特殊权限Ioctl配置**

08-07 10:49:20.149 4795 4795 I iwpriv : type=1400 audit(0.0:3457): avc: denied { ioctl } for path="socket:[1607798]" dev="sockfs" ino=1607798 ioctlcmd=8bfc scontext=u:r:system\_app:s0 tcontext=u:r:system\_app:s0 tclass=udp\_socket permissive=1

从上面的日志来看，按照我们通用的添加权限的方式规则，应该如下：

allow system\_app self:udp\_socket ioctl;

但是实际效果如何？？？

操作依然被denied，这是由于ioctl的控制在底层划分的更细，需要允许对应ioctlcmd操作

具体方法为：

1、查找对应的ioctlcmd在ioctl\_defines中的定义，如上文中的8bfc，对应的是SIOCIWFIRSTPRIV\_1C（在对应的系统中查看定义）

2、在对应的文件加入如下的配置：

allowxperm system\_app self:udp\_socket ioctl SIOCIWFIRSTPRIV\_1C;

这样，在ioctl操作时，对应的ioctlcmd就会被允许了。

二、

allow mediacodec default\_prop:property\_service set;

 Android 8.1 及以上版本系统添加了权限限制，不允许普通进程设置系统属性，编译提示neverallow，解决方法如下：

sepolicy/property\_contexts文件：

use\_xxx u:object\_r:use\_xxx\_prop:s0

sepolicy/property.te文件：

type use\_xxx\_prop, property\_type, mlstrustedsubject;

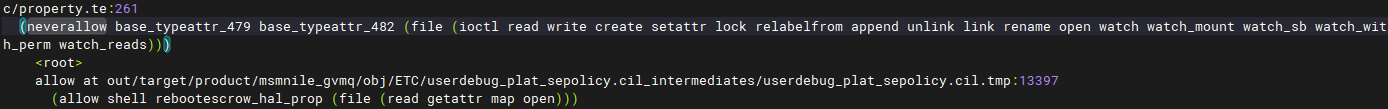
mediacodec.te文件：

set\_prop(mediacodec,use\_mpp\_mode\_prop);

解决方案 2

非系统域的属性设置则没有如上限制，可以将 use\_xxx 属性修改为 vendor.use\_xxx 改为 vender 域的属性

SELinux: The following public types were found added to the policy without an entry into the compatibility mapping file(s) found in private/compat/V.v/V.v[.ignore].cil, where V.v is the latest API level.



### 1. dac\_override

egbin: type=1400 audit(0.0:879): avc: denied { dac\_override } for capability=1 s:egbin:s0 tclass=capability permissive=1

* 1

​ 需要给egbin dac\_override权限，但是该权限是Android P的neverallow规则中的，不能被添加。dac\_override权限意思是容许进程旁路的所有DAC权限：uid，gid，ACL 等等，即有这个权限可以无视linux的权限中的用户、用户组。谷歌这样做的原因可能是这个dac\_override权限能力范围太大，不能给普通程序这个权限，只有少数程序有这个权限。

ueventd.te(7):allow ueventd self:capability { chown mknod net\_admin setgid fsetid sys\_rawio dac\_override fowner };

zygote.te(7):allow zygote self:capability { dac\_override setgid setuid fowner chown };

netd.te(44):allow netd self:capability { dac\_override chown fowner };

runas.te(14):dontaudit runas self:capability dac\_override;

vold.te(20):allow vold self:capability { net\_admin dac\_override mknod sys\_admin chown fowner fsetid };

installd.te(6):allow installd self:capability { chown dac\_override fowner fsetid setgid setuid };

tee.te(9):allow tee self:capability { dac\_override };

* 1
* 2
* 3
* 4
* 5
* 6
* 7

dac\_override权限问题一种解法  
出现这种问题可能原因是进程的组与需要访问的文件的组不同，进程没有权限访问改组的文件，需要申请dac\_override权限来旁路的所有DAC权限：uid，gid使进程可以访问该文件。以egbin为例，下面为开机时egbin的rc文件，及data的user和group。

drwxrwx--x 43 system system u:object\_r:system\_data\_file:s0 4096 1970-01-29 01:46 data

service egbin /system/bin/egbin start

class core

user root

group root

writepid /dev/stune/top-app/tasks

seclabel u:r:egbin:s0

* 1
* 2
* 3
* 4
* 5
* 6
* 7
* 8

​ 可以看到 egbin 执行时候user是root，而data是属于system，因此导致需要申请dac\_override才能访问data分区。如果不加dac\_override的权限，可以将egbin的组切换为system组即可访问，rc修改如下:

service egbin /system/bin/egbin start

class core

user root

group root system

writepid /dev/stune/top-app/tasks

seclabel u:r:egbin:s0

* 1
* 2
* 3
* 4
* 5
* 6

参考文献：https://blog.csdn.net/pen\_cil/article/details/89434349

https://android-review.googlesource.com/c/platform/external/sepolicy/+/174530/5/update\_engine.te#11

https://danwalsh.livejournal.com/79643.html

查看命令

man capabilities

### ****DAC\_READ\_SEARCH****

当我们在执行sh命令（如ls）时，可能会遇到selinux报有关 DAC\_READ\_SEARCH 的权限为问题。

07-05 18:38:00.268 I/ls ( 4276): type=1400 audit(0.0:598): avc: denied { dac\_read\_search } for capability=2 scontext=u:r:hello\_t:s0 tcontext=u:r:hello\_t:s0 tclass=capability permissive=1

* 1

首先我们来了解一下 capability 中的DAC\_READ\_SEARCH是什么？

在Linux中，root被分为64中不同的capabilities。比如加载内核模块的能力、绑定到1024以下端口的能力，我们这里的DAC\_READ\_SEARCH就是其中的一个。

DAC代表自由访问控制，大多数人将其理解为 standard Linux permissions，每个进程都有所有者/组。 为所有文件系统对象分配了所有者，组和权限标志。 DAC\_READ\_SEARCH允许特权进程忽略DAC的某些部分以进行读取和搜索。

在linux系统上，输入man capabilities可以看到有关DAC\_READ\_SEARCH的说明：

CAP\_DAC\_READ\_SEARCH

* Bypass file read permission checks and directory read and execute permission checks;\*

There is another CAPABILITY called DAC\_OVERRIDE

CAP\_DAC\_OVERRIDE

* Bypass file read, write, and execute permission checks.

man capabilities

从上面的说明我们可以看到，DAC\_OVERRIDE比 DAC\_READ\_SEARCH的权限更大，因为它可以忽略DAC规则来编写和执行内容，而不仅仅是读取内容。

据资料显示，DAC\_READ\_SEARCH的出现源于内核的改变，这种改变使内核变得更加安全了。

为了理解这句话，我们结合一个例子来进行讲解：

假设有一个叫 hello（hello\_t）的小程序，登录到系统后最终会执行。 该程序需要读取/etc/shadow。 在Linux系统上（比如Fedora / RHEL / CentOS以及其他 ），/etc /shadow 具有0000模式。 这意味着，即使系统没有以根用户身份运行（UID = 0），也不允许它们读取/写入/etc/shadow，除非它们具有DAC功能。

随着策略的发展，由于hello需要读取 /etc/shadow，它生成了DAC\_OVERIDE AVC，加入我们在编写sepolicy时，将它的sepolicy文件命名为hello\_t， 多年以来，一切工作正常，直到内核发生了变化……

如果某个进程尝试读取/etc/shadow，则允许该进程具有DAC\_OVERRIDE或DAC\_READ\_SEARCH。

在老版本的kernel里面有类似这样一段代码：

if DAC\_OVERRIDE or DAC\_READ\_SEARCH:

Read a file with 0000 mode.

* 1
* 2

但是，在新的kernel中，已经改成下面的形式：

if DAC\_READ\_SEARCH or DAC\_OVERRIDE

Read a file with 0000 mode.

* 1
* 2

假如在老的版本，你已经为 hello\_t 添加了 DAC\_OVERRIDE的权限，但是它从来不检查DAC\_READ\_SEARCH。

但是在新版本中，它首先检查DAC\_READ\_SEARCH，因此即使允许最终访问，我们也看到正在生成AVC。但是我们可以看到，虽然会生成avc打印，但是事实上，仍然允许访问。

在前面讲到，由于DAC\_OVERRIDE权限比较大，在不需要的情况下，如果需要添加DAC\_READ\_SEARCH的时候，建议可以去掉DAC\_OVERRIDE。因为其实hello\_t并不需要更改/etc/shadow，而只是读取而已，这样可以更加安全。

最后，消除DAC\_READ\_SEARCH最简单的方法，可以在我们的te文件里加上下面这一句：

dontaudit hello\_t self:capability { dac\_read\_search };

* 1

https://danwalsh.livejournal.com/77140.html

### 4 SYS\_PTRACE

当我们运行ps 、top等命令时，经常会碰到selinux报 SYS\_PTRACE 的权限问题。

通常，当我们的应用运行 ps 或者读取 /proc的内容时，会提示 SYS\_PTRACE 的问题。

https://bugzilla.redhat.com/show\_bug.cgi?id=1202043

比如selinux的打印可能像下面这样：

type=AVC msg=audit(1426354432.990:29008): avc: denied { sys\_ptrace } for pid=14391 comm="ps" capability=19 scontext=unconfined\_u:unconfined\_r:mozilla\_plugin\_t:s0-s0:c0.c1023 tcontext=unconfined\_u:unconfined\_r:mozilla\_plugin\_t:s0-s0:c0.c1023 tclass=capability permissive=0

* 1

sys\_ptrace通常指示一个进程正在尝试查看具有不同UID的另一个进程的内存。

通过man 命令 man capabilities 查看sys\_ptrace的说明：

CAP\_SYS\_PTRACE

* Trace arbitrary processes using ptrace(2);
* apply get\_robust\_list(2) to arbitrary processes;
* transfer data to or from the memory of arbitrary processes  
  using process\_vm\_readv(2) and process\_vm\_writev(2).
* inspect processes using kcmp(2).

这些访问类型可能应该处理为： dontaudited

运行ps命令是特权进程，可能会导致sys\_ptrace发生。

/proc下有一些特殊数据，特权进程可以通过运行ps命令来访问这些数据，运行ps的进程几乎从未真正需要此数据，调试工具使用了该数据，来看看在哪里设置了进程的一些随机内存。

因此最容易的修改方法是处理为：dontaudit

**dontaudit**：表示不记录违反规则的决策信息，且违反规则不影响运行(允许操作且不记录)

dontaudit mozilla\_plugin\_t self:capability { sys\_ptrace };

* 1

https://danwalsh.livejournal.com/71981.html

### 5 execute和execute\_no\_trans

当我们通过sh脚本或者bin文件运行系统命令（比如ls、ps等 ）时，可能会遇到下面的selinux avc提示：

avc: denied { execute } for comm="hello.sh" name="toolbox" dev="dm-1" ino=223 scontext=u:r:hello:s0 tcontext=u:object\_r:vendor\_toolbox\_exec:s0 tclass=file permissive=0

avc: denied { execute\_no\_trans } for comm="hello.sh" path="/vendor/bin/toolbox" dev="dm-1" ino=223 scontext=u:r:hello:s0 tcontext=u:object\_r:vendor\_toolbox\_exec:s0 tclass=file permissive=0

* 1
* 2
* 3

通常我们会按照sepolicy的规则，添加规则下面的规则到te文件：

allow hello vendor\_toolbox\_exec:file { execute execute\_no\_trans };

* 1

这时，我们编译工程时，可能会遇到编译失败的问题，提示neverallow：

libsepol.report\_failure: neverallow on line 952 of system/sepolicy/public/domain.te (or line 12401 of policy.conf) violated by allow hello vendor\_toolbox\_exec:file { execute execute\_no\_trans };

* 1

我遇到的情况有一个/system/bin/hello的可执行文件，执行ntpdate命令时，出现这个avc。而/system分区下没有ntpdate命令，因此会自动去执行/vendor/bin/ntpdate命令。

当我们添加sepolicy规则时，之所以出现neverallow，就在于我们是通过/system分区去执行/vendor分区命令。由于Google启动的Treble计划，为实现分区可独立升级，为防止分区升级造成分区调用出现问题。因此，是禁止分区之间调用的。

经过一段时间的尝试之后，通过两种方法解决了这个问题：

* 将/vendor/bin/ntpdate移植到/system/bin/下，当然因为在/system/bin/下的这个bin文件只有我们自己使用，为安全起见，可以给文件名改名处理，比如/system/bin/ntpdate\_my。
* 通过hidl机制，做一个服务去执行/vendor/bin/ntpdate，然后将结果返回给/system分区，不过这种做法比较适合只取一次结果的情况，不适合像top、ping之类的命令

type=1400 audit(0.0:214): avc: denied { entrypoint } for path="/system/bin/csity\_dhcplog.sh" dev="dm-2" ino=471 scontext=u:r:start-ssh:s0 tcontext=u:object\_r:csity\_dhcplog\_system\_exec:s0 tclass=file permissive=0



FAILED: out/target/product/msmnile\_gvmq/obj/FAKE/sepolicy\_freeze\_test\_intermediates/sepolicy\_freeze\_test

/bin/bash -c "(diff -rq -x bug\_map system/sepolicy/prebuilts/api/30.0/public system/sepolicy/public ) && (diff -rq -x bug\_map system/sepolicy/prebuilts/api/30.0/private system/sepolicy/private ) && (touch out/target/product/msmnile\_gvmq/obj/FAKE/sepolicy\_freeze\_test\_intermediates/sepolicy\_freeze\_test )"

Public和private里的要拷贝到prebuilt/aip/xx里面

Sysfs proc debugfs等不能在file\_contexts里配置，要在genfs\_contexts里用genfscon，编译后所在cil里。

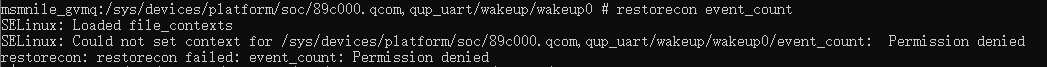
比如红旗C801在/sys路径下有个

：/sys/devices/platform/soc/89c000.qcom,qup\_uart/wakeup/wakeup0/event\_count

如果写在file\_contexts里：

/sys/devices/platform/soc/89c000.qcom,qup\_uart/wakeup/wakeup0/event\_count u:object\_r: sysfs\_wakeup:s0

，执行命令：restorecon event\_count，报错：



正确配置如下：

vim ./device/qcom/sepolicy\_vndr/generic/vendor/common/genfs\_contexts

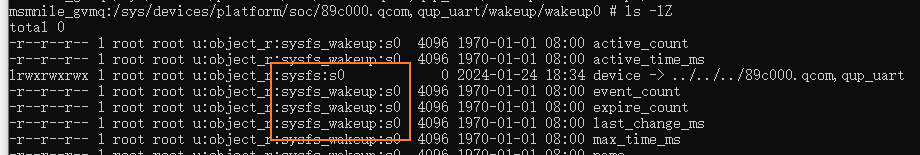
添加：

genfscon sysfs /devices/platform/soc/89c000.qcom,qup\_uart/wakeup/wakeup0 u:object\_r:sysfs\_wakeup:s0

注意路径，前面不要带/sys。因为genfscon sysfs已经指明了是sys。一开始不知道写的是路径全称，配置了无效。

编译make selinux\_policy -j32，然后push out/target/product/msmnile\_gvmq/vendor/etc/selinux，

生效：



avc: denied { open } for path="/sys/devices/platform/soc/a800000.ssusb/a800000.dwc3/xhci-hcd.0.auto/usb2/wakeup/wakeup64/event\_count" dev="sysfs" ino=47979 scontext=u:r:system\_suspend:s0 tcontext=u:object\_r:sysfs:s0 tclass=file permissive=1这个报错配置：allow system\_suspend sysfs:file { open read };是没用的。正确配置如上文，用genfscon。

SELinux : avc: denied { add } for

# pid=1686 uid=1000 name=com.hsae.auto.IBINDER\_CANBUS

# scontext=u:r:system\_app:s0 tcontext=u:object\_r:default\_android\_service:s0

# tclass=service\_manager permissive=1 "

类似的还有default\_android\_vndservice。表示你的binder没有定义selinux上下文，所以是默认的：

cat vendor/vndservice\_contexts

manager u:object\_r:service\_manager\_vndservice:s0

\* u:object\_r:default\_android\_vndservice:s0

这种类型的报错需要配置：

// at: private/service\_contexts

com.hsae.auto.IBINDER\_CANBUS u:object\_r:hsae\_auto\_canbus\_service:s0

// public/service.te

type hsae\_auto\_canbus\_service, app\_api\_service, system\_server\_service, service\_manager\_type;

以及allow语句：

allow system\_app hsae\_auto\_canbus\_service:service\_manager { add find };

或add\_service(system\_app, hsae\_auto\_canbus\_service)

编译报错：

neverallow on line 753 of system/sepolicy/private/system\_server.te (or line 52074 of policy.conf) violated by allow system\_app hsae\_auto\_canbus\_service:service\_manager { add };

libsepol.report\_failure: neverallow on line 18 of system/sepolicy/public/system\_app.te (or line 29638 of policy.conf) violated by allow system\_server hsae\_auto\_canbus\_service:service\_manager { add };

分析一下，system\_server.te定义了：

add\_service(system\_server, system\_server\_service);

###########################################

# add\_service(domain, service)

# Ability for domain to add a service to service\_manager

# and find it. It also creates a neverallow preventing

# others from adding it.

define(`add\_service', `

allow $1 $2:service\_manager { add find };

neverallow { domain -$1 } $2:service\_manager add;

')

即不允许除了system\_server之外的类型往service\_manager里面添加服务。即整个系统只能有一个add\_service, 已经给了system\_server，就不能再给sysem\_app。解决方法是把无关attribute移除：

type hsae\_auto\_canbus\_service, service\_manager\_type;

命令：

找到audioserver这个主体是在哪定义的：

find . -name "\*.te" -not -path "./out/\*" -type f | xargs grep "type audioserver"

**将 selinux 切换为 Enforcing 强制模式（如果已经是Enforcing模式可省略）**

setentforce 1

**#验证功能并打印log**

logcat | grep avc

永久打开SELinux（以高通8155平台为例）：

#### 更改kernel\_cmdline参数

https://192.168.64.47:8443/c/Android/QCOM\_SA8155\_6155/platform/vendor/qcom/msmnileau\_gvmq/+/38769/1/BoardConfig.mk

BOARD\_KERNEL\_CMDLINE := console=hvc0,115200 debug user\_debug=31 loglevel=9 print-fatal-signals=1 androidboot.console=ttyAMA0 androidboot.hardware=qcom androidboot.selinux=enforcing androidboot.memcg=1 init=/init swiotlb=4096 androidboot.usbcontroller=a600000.dwc3 androidboot.recover\_usb=1 firmware\_class.path=/vendor/firmware\_mnt/image kpti=0 msm\_cfg.cfg\_sel=1 pcie\_ports=compat androidboot.dtbo\_idx=1

宽容模式enforcing改成permissive

#### 内核启用SElinux

配置CONFIG\_SECURITY\_SELINUX=y（例如kernel/msm-5.4/kernel/configs/android-base.config）

**4.新增APP类型**

当前系统中app的domain和文件类型默认的分配原则为：

1、platform签名 + share systemuid -> system\_app + system\_app\_data\_file；

2、platform签名 + 普通用户-> platform\_app + app\_data\_file；

3、platform签名 + 特定用户 ： 如bluetooth，对应的domain为bluetooth + bluetooth\_data\_file

4、三方应用 -> untrusted\_app + app\_data\_file

根据特性需求，需要新增app类型：

1、system/sepolicy/mac\_permissions.xml

2、seapp\_contexts

user=\_app seinfo=theme name=com.nearme.themespace\* domain=theme\_app type=theme\_app\_data\_file levelFrom=user

3、file.te

type theme\_app\_data\_file, file\_type, data\_file\_type;

4、theme\_app.te

type theme\_app, domain, mlstrustedsubject;

这样，签名是@THEME(定义在system/sepolicy/keys.conf文件)和包名是comthemespace的应用，domain是theme\_app，对应的文件类型是theme\_app\_data\_file

**5.新增修改文件类型**

如某些需求，需要新增文件类型，只允许特定的domain访问，如存放字体文件的目录，只允许主题应用写入和删除，其他domain只有读的权限。

1、file\_contexts

/data/system/font(/.\*)? u:object\_r:font\_data\_file:s0

2、file.te

type font\_data\_file, file\_type, data\_file\_type;

3、system/core/rootdir/init.rc

在restorecon\_recursive /data之前加入：

mkdir /data/system/font 0777 system system

chmod 0777 /data/system/font

这样，adb shell ls -l -Z /data/system/font 查看到的就是上面配置的文件类型：

-rwxrwxrwx 1 u0\_a18 u0\_a18 u:object\_r:font\_data\_file:s0:c512,c768 3553188 2017-08-19 16:40 Regular.ttf

**6.新增可执行程序和service**

在init.rc中新增service，但没有对应的bin，一般通过sh脚本启动一些功能。这类service，需要在配置rc时，加上seclabel操作，之后，init启动service时，就会根据这个关键字解析出对应的context。以screen\_record service为例

1、system/core/rootdir/init.rc

service screen\_record /system/bin/sh /autochmod.sh "screen\_record”

…

seclabel u:r:screen\_record:s0

2、screen\_record.te

type screen\_record, domain;

type screen\_record\_exec, exec\_type, file\_type;

init\_daemon\_domain(screen\_record)

binder\_use(screen\_record)

allow screen\_record shell\_exec:file {read open execute\_no\_trans execute getattr entrypoint};

allow screen\_record rootfs:lnk\_file {getattr};

allow screen\_record rootfs:file {read open};

allow screen\_record system\_file:file { read open execute\_no\_trans };

allow screen\_record toolbox\_exec:file { read open execute\_no\_trans execute getattr};

allow screen\_record self:capability dac\_override;

allow screen\_record self:process execmem;

如上，新增service需要的默认权限在步骤2中，需要的时候拷贝、修改screen\_record为对应的service即可。

#### ****编译及验证****

**(1) 编译**

在根目录下编译 /system/sepolicy/ 中的文件，编译命令如下：

mmm /system/sepolicy/

1. make selinux\_policy -j8 *// vendor 和system 都修改了*
2. make selinux\_policy\_nonsystem -j8 *// 只修改 vendor相关的sepolicy*
3. 编译成功后，只需要替换如下文件 对应分区的\*.cil文件和\*\_contexts，即可快速验证，

示例命令行：

adb push out/product/target/xxx/system/etc/selinux/\* system/etc/selinux

adb push out/product/target/xxx/vendor/etc/selinux/\* vendor/etc/selinux

**其它分区如product亦然。**

#### ****编译错误怎么看：****

例如

libsepol.report\_failure: neverallow on line 99 of system/sepolicy/private/coredomain.te (or line 36611 of policy.conf) violated by allow system\_server sysfs:file { read write open };

libsepol.check\_assertions: 1 neverallow failures occurred  
Error while expanding policy

**即coredomain.te的第99行的策略与我们加的冲突。**

Android并不会使用SELinux提供的所有功能。注意以下几点：

* AOSP中的大部分政策都是使用内核政策语言定义的。在使用通用中间语言(CIL)时，会存在一些例外情况
* 不使用SELinux用户。唯一定义的用户是u。必要时，系统会使用安全上下文的类别字段表示实际用户
* 不使用SELinux角色和基于角色的访问权限控制 (RBAC)。定义并使用了两个默认角色：r（适用于主题）和object\_r（适用于对象）
* 不使用SELinux敏感度。已始终设置好默认的s0敏感度
* 不使用SELinux布尔值。一旦设备政策构建完成，该政策不再取决于设备状态。这简化了政策的审核和调试过程

#### SElinux相关模块

**SELinux 的构建逻辑位于以下文件中：**

* external/selinux：外部SELinux项目，用于构建HOST命令行实用工具以编译SELinux政策和标签
  + external/selinux/libselinux：Android仅使用外部libselinux项目的一个子集，以及一些Android专用自定义内容（参阅external/selinux/README.android了解详情）
  + external/selinux/libsepol：
    - chkcon：确定安全环境对指定的二进制政策（主机可执行文件）是否有效
    - libsepol：用于操控二进制安全政策（主机静态/共享库、目标静态库）的 SELinux 库
  + external/selinux/checkpolicy：SELinux 政策编译器（主机可执行文件：checkpolicy、checkmodule和dispol）。依赖于libsepol
* system/sepolicy：核心Android SELinux政策配置，包括上下文和政策文件。主要sepolicy构建逻辑也位于此处 (system/sepolicy/Android.mk)

### SElinux关键文件

#### 2.2.1. 上下文的描述文件（\*\_contexts）

可以在上下文的描述文件中为需要的对象指定标签

* file\_contexts：用于为文件分配标签，并且可供多种用户空间组件使用。在创建新政策时，需创建或更新该文件，以便为文件分配新标签。如需应用新的file\_contexts，需重新构建文件系统映像，或对要重新添加标签的文件运行restorecon（比如重新加载权限restorecon -R /（对应权限目录））。在升级时，对file\_contexts所做的更改会在升级过程中自动应用于系统和用户数据分区。此外，还可以通过以下方式使这些更改在升级过程中自动应用于其他分区：在以允许读写的方式装载相应分区后，将restorecon\_recursive（重载权限）调用添加到init.board.rc文件中
* property\_contexts：用于为Android系统属性分配标签，以便控制哪些进程可以设置这些属性。在启动期间，init进程会读取此配置
* service\_contexts：用于为Android Binder服务分配标签，以便控制哪些进程可以为相应服务添加（注册）和查找（查询）Binder引用。在启动期间，servicemanager进程会读取此配置
* genfs\_contexts：用于为不支持扩展属性的文件系统（例如，proc或vfat）分配标签。此配置会作为内核政策的一部分进行加载，但更改可能对内核inode无效。要全面应用更改，需要重新启动设备，或卸载并重新装载文件系统。此外，通过使用context=mount选项，还可以为装载的特定系统文件（例如vfat）分配特定标签
* seapp\_contexts：用于为**应用进程和/data/data目录**分配标签。在每次应用启动时，zygote进程都会读取此配置；在启动期间，installd会读取此配置
* mac\_permissions.xml：用于根据应用签名和应用软件包名称（后者可选）为应用分配seinfo标记。随后，分配的seinfo标记可在seapp\_contexts文件中用作密钥，以便为带有该seinfo标记的所有应用分配特定标签。在启动期间，system\_server会读取此配置

**简单来说：**

* file\_contexts //系统中所有file\_contexts安全上下文
* seapp\_contexts //app安全上下文
* property\_contexts //属性的安全上下文
* service\_contexts //service文件安全上下文
* genfs\_contexts //虚拟文件系统安全上下文

## 安全上下文文件

在Android源码中会存在几个重要的上下文文件， 用来描述系统和属性，服务等的上下文信息， 路径在system/sepolicy

|  |  |
| --- | --- |
| file\_contexts | 根系统中所有文件的安全上下文， 如/system/bin， /system/etc等文件，源码路径如： system/sepolicy/private/file\_contexts |
| property\_contexts | 属性的安全上下文，源码路径如： system/sepolicy/private/property\_contexts |
| genfs\_contexts | Generalized filesystem虚拟文件系统安全上下文，如proc, sysfs,debugfs文件系统，源码路径如： system/sepolicy/private/genfs\_contexts |
| service\_contexts | service文件安全上下文，用于为 Android Binder 服务分配标签，以便控制哪些进程可以为相应服务添加（注册）和查找（查询）Binder 引用。在启动期间，servicemanager 进程会读取此配置，HIDL机制出现后，就有两个，源码路径：  system/sepolicy/private/hwservice\_contexts  system/sepolicy/private/service\_contexts |
| mac\_permission.xml | 用于根据应用签名和应用软件包名称（后者可选）为应用分配 seinfo 标记。随后，分配的 seinfo 标记可在 seapp\_contexts 文件中用作密钥，以便为带有该 seinfo 标记的所有应用分配特定标签。在启动期间，system\_server 会读取此配置， 源码路径为： system/sepolicy/private/mac\_permissions.xml |
| seapp\_contexts | app安全上下文，用于描述apk安装之后的目录文件和 /data/data 目录分配标签。在每次应用启动时，zygote 进程都会读取此配置；在启动期间，installd 会读取此配置。源码路径如： system/sepolicy/private/seapp\_contexts |

### 常用可用权限列表



**在Android 8.0及更高版本中，政策位于AOSP中的以下位置：**

* system/sepolicy/public:其中包括所导出的用于供应商特定政策的政策。所有内容都会纳入Android 8.0兼容性基础架构。公共政策会保留在不同版本上，因此可以在自定义政策的/public中添加任何内容。正因如此，可存放在/public中的政策类型的限制性更强。将此目录视为相应平台的已导出政策API：处理/system与/vendor之间的接口的所有内容都位于这里
* system/sepolicy/private:包括系统映像正常运行所必需（但供应商映像政策应该不知道）的政策
* system/sepolicy/vendor:包括位于/vendor但存在于核心平台树（非设备特定目录）中的组件的相关政策。这是构建系统区分设备和全局组件的软件工件；从概念上讲，这是下述设备专用政策的一部分
* device/manufacturer/device-name/sepolicy:包含设备专用政策，以及对政策进行的设备自定义（在Android 8.0及更高版本中，该政策对应于供应商映像组件的相关政策）

在Android 11及更高版本中，system\_ext和product分区还可以包含特定于分区的政策。system\_ext和product政策也分为公共政策和私有政策，且供应商可以使用system\_ext和product的公共政策（例如系统政策）

* SYSTEM\_EXT\_PUBLIC\_SEPOLICY\_DIRS:包括所导出的用于供应商特定政策的政策。已安装到system\_ext分区（**编译结果会存在system\_ext/etc/selinux**）
* SYSTEM\_EXT\_PRIVATE\_SEPOLICY\_DIRS:包括system\_ext映像正常运行所必需（但供应商映像政策应该不知道）的政策。已安装到 system\_ext分区（**编译结果会存在system\_ext/etc/selinux**）
* PRODUCT\_PUBLIC\_SEPOLICY\_DIRS:包括所导出的用于供应商特定政策的政策。已安装到product分区（**主要针对packages/services的权限，编译结果在system/product/etc/selinux**）
* PRODUCT\_PRIVATE\_SEPOLICY\_DIRS:包括product映像正常运行所必需（但供应商映像政策应该不知道）的政策。已安装到product分区

**注意：**在使用system\_ext和product公共政策时要格外小心。公共政策充当system\_ext/product和vendor之间的导出API。合作伙伴应自行管理兼容性问题

selinux相关上下文和策略文件也被分成平台和非平台，system/sepolicy/Android.mk中有说明:

system/sepolicy/private：平台私有规则，不会向vendor部分暴露。里面包含了各种策略控制te文件，以及上面提到的上下文文件。

system/sepolicy/public: 平台共有策略的全部定义，里面包含了各种策略控制te文件，

system/sepolicy/vendor 厂商规则，可引用public的规则，不能引用private的规则

device/manufacturer/device-name/sepolicy 厂商自定义的规则，包括如上的vendor部分

plat\_sepolicy.cil： 所有平台策略转成cil，会在init启动的时候进行动态编译， 由很多的te文件组合而来

vendor\_sepolicy.cil： 所有厂商策略转成cil，会在init启动的时候进行动态编译， 由很多的te文件组合而来

## 5. 特殊分区概念和权限

在Android 11及更高版本中，system\_ext和product分区还可以包含特定于分区的政策。宏也会变更，替换原来的BOARD\_PLAT\_PUBLIC\_SEPOLICY\_DIR和BOARD\_PLAT\_PRIVATE\_SEPOLICY\_DIR

查看Android 12源码/system/sepolicy/README的说明：

SYSTEM\_EXT\_PUBLIC\_SEPOLICY\_DIRS += device/acme/roadrunner-sepolicy/systemext/public

SYSTEM\_EXT\_PRIVATE\_SEPOLICY\_DIRS += device/acme/roadrunner-sepolicy/systemext/private

PRODUCT\_PUBLIC\_SEPOLICY\_DIRS += device/acme/roadrunner-sepolicy/product/public

PRODUCT\_PRIVATE\_SEPOLICY\_DIRS += device/acme/roadrunner-sepolicy/product/private

The old BOARD\_PLAT\_PUBLIC\_SEPOLICY\_DIR and BOARD\_PLAT\_PRIVATE\_SEPOLICY\_DIR

variables have been deprecated in favour of SYSTEM\_EXT\_\*.

复制

从Android 11开始，system\_ext和product分区可以将其指定的公共类型导出到vendor分区。

当system\_ext和product分区基于同一平台版本N时，构建系统会生成到system\_ext/etc/selinux/mapping/N.cil和product/etc/selinux/mapping/N.cil的基本映射文件，其中包含从type到type\_N的身份映射。vendor可以通过版本化属性type\_N访问type

### 5.1. product分区概念

* Android 11上，编译结果目录根目录有个product，但是里面除了etc没有其他文件，不会生成镜像；在system也有product，里面存在实际结果文件（含build.prop）
* Android 12上，编译结果目录根目录没有product；在system也有product，里面存在实际结果文件（不包含build.prop）

**使用以下编译标记向 product 分区中安装模块：**

* Android.bp 中的 product\_specific: true
* Android.mk 中的 LOCAL\_PRODUCT\_MODULE := true

### 5.2. system\_ext分区

* Android 11上（未开启动态分区），编译结果目录根目录有个system\_ext，但是里面除了etc没有其他文件，不会生成镜像；在system也有system\_ext，里面存在实际结果文件（含build.prop）
* Android 12上（开启动态分区），编译结果目录根目录有system\_ext，并且会生成system\_ext.img镜像，但是分区大小配置是同system一起；在system也有system\_ext，但是是软链接链接到根目录的system\_ext

**使用以下编译标记向system\_ext分区中安装模块：**（宏的源码参阅build\soong\androidmk\androidmk\android.go）

* Android.bp中配置 system\_ext\_specific: true，就将编译到 system\system\_ext\ 中
* Android.mk 中配置 LOCAL\_SYSTEM\_EXT\_MODULE := true，就将编译到 system\system\_ext\ 中；LOCAL\_PRIVILEGED\_MODULE := true 决定在 priv-app 文件中，不加 LOCAL\_PRIVILEGED\_MODULE 则编译到 system\system\_ext\app\

### 6.2. 转储用户和内核堆栈

在某些情况下，事件日志中包含的信息不足以查明拒绝事件的来源。通常，获取调用链（包括内核和用户空间）有助于更好地了解发生拒绝事件的原因。 最新的内核定义了一个名为 avc:selinux\_audited 的跟踪点。使用Android simpleperf可启用此跟踪点并获取调用链。

**支持的配置：**

* 支持5.10及更高版本的Linux内核（尤其是Android通用内核分支mainline和android12-5.10），也支持android12-5.4分支。可以使用simpleperf来确定您的设备上是否定义了跟踪点：adb root && adb shell simpleperflist|grep avc:selinux\_audited
* 应该可以重现正在调试的事件。使用Simpleperf时不支持启动时间事件；不过，您仍然可以重启服务以触发事件

#### 6.2.1. simpleperf抓取调用堆栈链

调用链是一个统一的内核和用户空间调用链，可发起跟踪从用户空间直到内核中发生拒绝事件的位置，更好地查看代码流

**获取调用链：**

第一步是使用simpleperf record录制事件：adb shell -t "cd /data/local/tmp && su root simpleperf record -a -g -e avc:selinux\_audited"

第二步应触发导致拒绝事件的事件。之后，应停止录制。在此例中应使用Ctrl-c获取样本：^Csimpleperf I cmd\_record.cpp:751] Samples recorded: 1. Samples lost: 0.

最后，可使用simpleperf report检查获取的堆栈轨迹。 例如：

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

### 在机台中的sepolicy相关文件

**•Vendor/etc/selinux**

1）.两个cil 文件

  1.1 plat\_pub\_versioned.cil -->public platform policy

  1.2 vendor\_sepolicy.cil-->vendor and BOARD\_SEPOLICY\_DIRS policy

2）. 一些contexts 文件

vendor\_file\_contexts vendor\_hwservice\_contexts vendor\_property\_contexts vendor\_seapp\_contexts vndservice\_contexts -->主要是BOARD\_SEPOLICY\_DIRS下面的contexts几个文件拼起来的

3.）其他几个文件

plat\_sepolicy\_vers.txt

precompiled\_sepolicy

precompiled\_sepolicy.plat\_and\_mapping.sha256

vendor\_mac\_permissions.xml

**•System/etc/selinux**

├── mapping

 │   ├── 26.0.cil-->system/sepolicy/private/compat/26.0/26.0.cil

 │   ├── 27.0.cil-->system/sepolicy/private/compat/27.0/27.0.cil

 │    └── 28.0.cil-->system/sepolicy/public中的type 映射的cil. 格式为：

                (typeattributeset proc\_pid\_max\_28\_0 (proc\_pid\_max))

                (expandtypeattribute (proc\_pid\_max\_28\_0) true)

                (typeattribute proc\_pid\_max\_28\_0)

├── plat\_and\_mapping\_sepolicy.cil.sha256 -->校验值

├── plat\_file\_contexts -->system/sepolicy/private/+vendor/app-prebuilt/data/sepolicy/

                                                 +device/qcom/sepolicy/private/

├── plat\_hwservice\_contexts-->system/sepolicy/private/hwservice\_contexts

├── plat\_mac\_permissions.xml--> system/sepolicy/private/mac\_permissions.xml

├── plat\_property\_contexts-->system/sepolicy/private/+device/qcom/sepolicy/private/

├── plat\_seapp\_contexts-->system/sepolicy/private/seapp\_contexts

├── plat\_service\_contexts-->system/sepolicy/private/+vendor/app-prebuilt/data/sepolicy/

                                                      +device/qcom/sepolicy/private/

├── selinux\_denial\_metadata-->sytsem/sepolicy/private/bug\_map

└── plat\_sepolicy.cil-->system/sepolicy/private/ +system/sepolicy/public中的policy

### 4.兼容性

在Android p中，有一些为了之后的兼容性而有一些做法。

**4.1. public中的type映射为\_vN的attribute(其中vN是版本。）**

     (typeattributeset foo\_vN (foo)) -->在vN.cil中

**4.2. Platform-public policy**导出为allow source\_foo target\_bar:class perm; 会被包入到vendor policy中。在编译的时候这会转化为vendor image 中的policy（在转化后的CIL中）：

 (allow source\_foo\_vN target\_bar\_vN (class (perm))) -->在plat\_pub\_versioned.cil和plat\_sepolicy.cil中

**4.3 对于platform 升级，会有以下几种情况。**

**4.3.1.Same type**

例如/dev/binder，/dev/binder在所有的版本中的label都是binder\_device。这体现在转换之后的policy中如下：

binder\_device\_v1 … binder\_device\_vN

从v1->v2升级时，platform policy必须包含：

type binder\_device; -> (type binder\_device) (in CIL)

在v1的映射文件中（CIL， v1.cil中）:

(typeattributeset binder\_device\_v1 (binder\_device))

在v2的映射文件中（CIL，v2.cil中）:

 (typeattributeset binder\_device\_v2 (binder\_device))

在v1的vendor policy中(CIL，plat\_plat\_versioned.cil):

(typeattribute binder\_device\_v1)  
(allow binder\_device\_v1 …)

在v2的vendor policy中(CIL，plat\_plat\_versioned.cil) ：

(typeattribute binder\_device\_v2)  
(allow binder\_device\_v2 …)

**4.3.2.New types**

例如新加的type为sysfs\_A ，v1->v2升级时，platform policy（both plat\_pub\_versioned.cil和plat\_sepolicy.cil）必须包含:

type sysfs\_A; -> (type sysfs\_A) (in CIL)  
type sysfs; (type sysfs) (in CIL)

在v1的映射文件中（CIL，v1.cil中）:

(typeattributeset sysfs\_v1 (sysfs sysfs\_A))

在v2的映射文件中（CIL，v2.cil中）:

(typeattributeset sysfs\_v2 (sysfs))  
(typeattributeset sysfs\_A\_v2 (sysfs\_A))

在v1的vendor policy中(CIL，plat\_pub\_versioned.cil中):

(typeattribute sysfs\_v1)  
(allow … sysfs\_v1 …)

在v2的vendor policy中(CIL，plat\_pub\_versioned.cil中):

(typeattribute sysfs\_A\_v2)  
(allow … sysfs\_A\_v2 …)  
(typeattribute sysfs\_v2)  
(allow … sysfs\_v2 …)

**4.3.3 Removed types**

 (typeattribute sysfs\_v1)  
(allow … sysfs\_v1 …)

**4.3.3.1.Example Version 1: Collapsing types (removing sysfs\_A)**

v1->v2升级时，platform policy必须包含:

type sysfs; (type sysfs) (in CIL)

在v1的映射文件中（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))

在v2的映射文件中（CIL）:

(typeattributeset sysfs\_v2 (sysfs))

在v1的vendor policy中(CIL):

(typeattribute sysfs\_A\_v1)  
(allow … sysfs\_A\_v1 …)  
(typeattribute sysfs\_v1)  
(allow … sysfs\_v1 …)

在v2的vendor policy中(CIL):

(typeattribute sysfs\_v2)  
(allow … sysfs\_v2 …)

**4.3.3.2.Example Version 2: Removing completely (foo type)**

v1->v2升级时，platform policy必须包含:

# nothing - we got rid of the type

在v1的映射文件中（CIL）:

(type foo) #needed in case vendors used the foo label on objects  
(typeattributeset foo\_v1 (foo))

在v2的映射文件中（CIL）:

# nothing - get rid of it

在v1的vendor policy中(CIL):

(typeattribute foo\_v1)  
(allow foo …)  
(typeattribute sysfs\_v1)  
(allow sysfs\_v1 …)

在v2的vendor policy中(CIL):

(typeattribute sysfs\_v2)  
(allow sysfs\_v2 …)

PS:其实主要是参考官网上的内容做的一点整理。后面后更多理解再来编辑。

**本文大部分内容参考：**

<https://cloud.tencent.com/developer/article/2170439>

<https://blog.csdn.net/cassie_huang/article/details/82775328>

<https://blog.51cto.com/u_15243273/5426739>

<https://lineageos.org/engineering/HowTo-SELinux/>

**https://www.cnblogs.com/blogs-of-lxl/p/7515023.html**

<https://dl.acm.org/doi/fullHtml/10.1145/3442381.3450007>**（看不懂）**

<https://blog.csdn.net/ch853199769/article/details/82501078/>

<https://blog.csdn.net/woai110120130/article/details/128060783>

**https://blog.csdn.net/Luoshengyang/article/details/37749383**

**问题：**

**研究system app配置，mac\_permission.xml等文件意思**

**进程里执行子进程、执行shell脚本，标签是什么？transition是什么意思，如果shell脚本在file\_contexts里配置了label，以及te文件，如果没有又是怎样？**

**Adb shell执行执行test.sh,test.sh的上下文是什么？**

**进程的标签**

**Exewrapper是一个可执行程序，他的adb root后，执行这个程序，标签是：**

**u:r:su:s0**

**现配置：**

**//** **file\_contexts**

**/system/bin/exewrapper u:object\_r:exewrapper\_system\_exec:s0**

**//te**

**type exewrapper\_system,domain;**

**type exewrapper\_system\_exec, file\_type, system\_file\_type;**

**type\_transition su exewrapper\_system\_exec:process exewrapper\_system;**

**这样执行程序后，进程的label就是u:r:exewrapper\_system:s0。另外exewrapper里面fork出的子进程，标签也是u:r:exewrapper\_system:s0。**

**说明如果不声明type\_transition，进程的label是继承的父进程。**

**考虑到adb shell可能是root或非root，应定义：**

**type\_transition {su shell} exewrapper\_system\_exec:process exewrapper\_system;**

**同样，在exewrapper里创建子进程，也一样配：**

**/system/bin/testsubprocess u:object\_r:testsubprocess\_system\_exec:s0**

**type testsubprocess\_system, domain;**

**type testsubprocess\_system\_exec, file\_type, system\_file\_type;**

**type\_transition exewrapper\_system testsubprocess\_system\_exec:process testsubprocess\_system;**

**或者也可以只声明type testsubprocess\_system, domain;，然后通过代码来设置子进程的label：**

  std::string cc = "u:r:testsubprocess\_system:s0";

  security\_context\_t sContext  = cc.data();

  status\_t st = ForkExecvp(cmd, &output, sContext);

**原理是在fork后，execvp的前面调用setexeccon。**