1 背景

Selinux扮演者Linux/Android系统资源隔离，访问权限控制等非常重要的角色，Linux系统中一切皆文件，因此Selinux也是基于此来控制每个进程对每个文件，属性等操作。

如果开启了强制模式，如果没有配置，各个进程可能无法启动，启动了部分功能可能无法使用。

由于每一个文件，进程，属性，app，服务，端口都有对应的类型值，只有对自身功能和代码比较熟悉的情况下，才不会疏漏。所有需要每一位FO根据自己功能和代码进行配置，下面说明一下存在哪些类型，主要看我们经常需要用到的context：

```
# android/system/sepolicy/
./vendor/vndservice_contexts
./vendor/file_contexts        # 文件类型的contexts，例如可执行文件，各个资源文件路径
./private/genfs_contexts
./private/port_contexts
./private/file_contexts
./private/property_contexts    # 属性值的contexts
./private/seapp_contexts       # apk的contexts
./private/hwservice_contexts   # 供应商进程的contexts，可暂时不关注
./private/initial_sid_contexts
./private/service_contexts     # 服务的contexts,默认出现default_android_service，就需要配置
./prebuilts/api/26.0/private/genfs_contexts
./prebuilts/api/26.0/private/port_contexts
./prebuilts/api/26.0/private/file_contexts
./prebuilts/api/26.0/private/property_contexts
./prebuilts/api/26.0/private/seapp_contexts
./prebuilts/api/26.0/private/hwservice_contexts
./prebuilts/api/26.0/private/initial_sid_contexts
./prebuilts/api/26.0/private/service_contexts
./prebuilts/api/27.0/private/genfs_contexts
./prebuilts/api/27.0/private/port_contexts
./prebuilts/api/27.0/private/file_contexts
./prebuilts/api/27.0/private/property_contexts
./prebuilts/api/27.0/private/seapp_contexts
./prebuilts/api/27.0/private/hwservice_contexts
./prebuilts/api/27.0/private/initial_sid_contexts
./prebuilts/api/27.0/private/service_contexts
./prebuilts/api/28.0/private/genfs_contexts
./prebuilts/api/28.0/private/port_contexts
./prebuilts/api/28.0/private/file_contexts
./prebuilts/api/28.0/private/property_contexts
./prebuilts/api/28.0/private/seapp_contexts
./prebuilts/api/28.0/private/hwservice_contexts
./prebuilts/api/28.0/private/initial_sid_contexts
./prebuilts/api/28.0/private/service_contexts
./prebuilts/api/28.0/public/property_contexts
./public/property_contexts
```

以上26.0/27.0/28.0是针对不同SDK的，如果是`修改的public/private/vendor`下的对应`xxx_context`，**需要在对应的SDK的contexts文件中添加进去，否则编译不通过。**

2 Selinux简要说明

我理解的Selinux，不管什么进程也好，什么属性也好，只分平台和非平台，这个平台和非平台是针对于Google发布的Android各个版本系统的，**如果是Google原生的，我们都认为是平台规则；如果是qcom，星河，德赛，博世等供应商的规则，我们认为是非平台的规则。**平台和非平台的规则配置是不一样的，但是差别不大。

平台的规则通常在`android/system/sepolicy`目录下配置规则，非平台的规则，通常在`android/device/<供应商>`目录下去配置。目前各个进程启动都没问题了，简单一点原则就是**先在`android/device`搜有没有规则文件，有的话就在自己模块的规则文件中添加规则；如果没有再在`android/system/sepolicy`目录下搜规则文件。**

3 配置说明

3.1 拒绝日志收集

3.1.1 使用logcat收集

使用如下指令，将日志导出到avc.log中

```
logcat -v time |grep -ie "avc:" > /sdcard/avc.log
```

然后全功能点检自己的功能，点检结束之后，停止日志收集，导出日志。

3.1.2 使用dlt日志收集

测试功能前删掉前面的日志，然后一直全功能测试自己的应用，功能测试完毕之后，导出dlt日志。

使用dlt viewer工具搜索`avc:`，将所有avc日志复制到另外一个文件中。

3.2 规则自动化转换

在`/android/external/selinux/prebuilts/bin`目录下有一个`audit2allow`工具。使用这个工具可以自动化转换avc拒绝日志。转化步骤如下：

3.2.1 编译工程

```
source build/envsetup.sh
lunch 51
make -j32 2>&1 | tee android_build_log.txt
```

首先使用如上指令全编整个系统，保证工具链，所有仓库都能编译通过，否则待会儿配置selinux的时候，编译不通过不好定位。

3.2.2 转化拒绝日志

```
# 进入android/external/selinux/prebuilts/bin
audit2allow -i denied.txt > selinux_rule.txt
```

- denied.txt是上面搜集的avc日志文件
- selinux_rule.txt是转化后的规则文件

3.2.3 转化后的规则说明

```
#============= adbd ==============
allow adbd ctl_start_prop:property_service set;
allow adbd np_systemlog_data_file:dir { getattr open read };
allow adbd np_systemlog_data_file:file { open read };

#============= hal_audio_default ==============
allow hal_audio_default netd:unix_stream_socket connectto;
allow hal_audio_default self:tcp_socket { connect create };
allow hal_audio_default system_prop:file { open read };

#============= system_app ==============
allow system_app broadcastradio_device:chr_file ioctl;
allow system_app default_prop:property_service set;
allow system_app firewalldaemon:unix_stream_socket connectto;
allow system_app np_wifi_prop:property_service set;
allow system_app proc:file { getattr open read };
allow system_app shell_data_file:dir search;
allow system_app socket_device:sock_file write;
allow system_app system_app_data_file:file execute_no_trans;
allow system_app system_data_file:dir { open read };
allow system_app system_data_file:file { lock open };
allow system_app vendor_usb_prop:file { open read };
```

1. `#============= adbd ==============`

这一行说明规则应该配置在`adbd.te`中，所以在`/android/device/<供应商>`和`/android/system/sepolicy`目录下搜索`adbd.te`文件，找到之后将对应的规则加进去，然后保证能编译通过，保证`out/target/product/msmnile_gvmq/obj/ETC/`目录下搜索刚添加的规则已经配置即可。

2. `allow adbd np_systemlog_data_file:dir { getattr open read };`

这条规则中，`adbd`代表一个type，`np_systemlog_data_file`代表一个type。

配置原则应该在`np_systemlog_data_file`所在module的目录下，创建`adbd.te`文件，然后加规则，这样方便平台化移植。

3.出现默认规则配置方法

！！！！！！！！！！！此条非常重要！！！！！！！！！！！！！因为当前项目中很多应用的属性和文件都是默认的，还没有配，导致部分功能异常。

```
allow system_app default_prop:property_service set;
```

如果出现了`default_prop`、`vendor_file`、`default_android_service`等默认的type的时候，通常配置到对应的te规则文件中都会编译不通过，或者通过了也不会生效。

此时需要看原始日志，看是哪个属性或者文件没有定义type。原始日志如下：

```
Line 178: 23834 2020/01/02 00:40:21.964711 71.4783 71 ECU1 KERN KERN 328 log
error verbose 1 selinux: avc: denied { set } for property=persist.bt.autoconnect
pid=2900 uid=1000 gid=1000 scontext=u:r:system_app:s0
tcontext=u:object_r:default_prop:s0 tclass=property_service permissive=1\x0a
```

从上面日志可以看出`persist.bt.autoconnect`属性被认为是默认属性了，因此需要按照属性的配置规则，给该属性取一个type名字，然后再增加具体的allow。属性详细配置方法，后面专门讲。

3.3 各个Contexts配置说明

为了平台化移植，所有规则和contexts定义都尽量放在`android/device/gxatek/common/legacy/sepolicy`目录下。

> 命名说明：
>
> **非平台type的命名规则：**np_xxxxxx_type。例如时钟同步服务：np_timesync_service

> 目录说明：
>
> 在`/module/np_types`中定义contexts和type，contexts定义在对应的xxx_contexts文件中，type定义在`module.te`中。
>
> 在`/module/np_rules`中定义**平台**的规则。
>
> 在`/module/mix_rules`中定义**项目差异**规则。

> 理解selinux tips: 在type定义的时候，`,`逗号可以简单的理解成java中的继承，如果继承了某一种type, 就具备了该type的能力。

> 定义comtext模板：u:object_r:xxxxxx:s0　`xxxxxx`替换成自己给context取的名字。

3.3.1 service配置

- 定义Context

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/servic
e_contexts
timesync                                  u:object_r:np_timesync_service:s0
```

- 定义Type

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/service.te
type np_timesync_service, system_api_service, system_server_service,
service_manager_type;

# 上面的继承关系，可以看出定义np_timesync_service为系统服务
```

3.3.2 文件配置

此处的**文件**包含**进程可执行文件**或者进程中需要使用到的文件。

1.进程可执行文件配置

- 定义Context

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/file_contexts
/(vendor|system/vendor)/bin/hw/vendor\.ts\.timesync@1\.0-service
u:object_r:np_timesync_exec:s0
```

- 定义Type

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/timesync.te
type np_timesync, domain;
type np_timesync_exec, exec_type, vendor_file_type, file_type;
```

- 定义规则

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/timesync.te
init_daemon_domain(np_timesync)

# 声明规则的意思是：设置init转换到daemon域，可执行文件是xxxx_exec，此例可执行文件是
np_timesync_exec，此时init进程就可以通过解析rc配置文件拉起np_timesync这个进程。
```

2.普通文件配置

- 定义Context

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/SystemLog/np_types/file_
contexts
/data/vendor/log(/.*)?
u:object_r:np_systemlog_data_file:s0
/dev/socket/dlt(/.*)?
u:object_r:np_systemlog_dlt_socket:s0
/log(/.*)?
u:object_r:np_systemlog_data_file:s0
```

- 定义Type

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/SystemLog/np_types/file.
te
type np_systemlog_data_file, file_type, data_file_type;
type np_systemlog_dlt_socket, file_type, coredomain_socket,
mlstrustedobject;
```

- 定义规则

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/SystemLog/np_rules/syste
mlog.te
allow np_systemlog_deamon np_systemlog_data_file:dir create_dir_perms;
allow np_systemlog_deamon np_systemlog_data_file:file create_file_perms;

# 以上规则表示：允许 np_systemlog_deamon类型 在 np_systemlog_data_file 类型中创建
目录
# 以上规则表示：允许 np_systemlog_deamon类型 在 np_systemlog_data_file 类型中创建
文件
```

3.3.3 hal服务配置

- 定义Context

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/hwserv
ice_contexts
vendor.ts.timesync::ITimeSync    u:object_r:np_timesync_hwservice:s0
```

- 定义Type

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/hwserv
ice.te
type np_timesync_hwservice, hwservice_manager_type;
```

- 定义规则

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_rules/timesync.te
add_hwservice(np_timesync, np_timesync_hwservice)
allow np_timesync np_timesync_hwservice:hwservice_manager find;
# 以上两条规则的意思是:
#            允许 np_timesync类型 向 np_timesync_hwservice类型 添加服务
#            允许 np_timesync类型 查询 np_timesync_hwservice类型服务

hwbinder_use(np_timesync)
vndbinder_use(np_timesync)
# Android P及以后创建hal服务需要加上以上两句规则，代表：允许从init域转换到np_timesync域
```

3.3.4 应用签名相关配置

我理解与底层服务有交互的app都需要配置seapp，如果不用和底层交互的app，我觉得可以不配，例如时间同步应用APK有和hal服务交互的场景，所以需要配seapp。如果是第三方app，或者没有直接和HAL层交互的app可以不配。从部分博客介绍该配置和签名有关系，

- 定义Context

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/service_contexts
timesync                                    u:object_r:np_timesync_service:s0
```

- 定义Type

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/service.te
type np_timesync_service, system_api_service, system_server_service,
service_manager_type;
```

- 定义seapp

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/np_types/seapp_contexts
user=system seinfo=platform name=com.ts.car.time domain=np_timesync
type=system_app_data_file
```

- 定义规则

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/TimeSync/mix_rules/system_app.te
allow system_app np_timesync_hwservice:hwservice_manager find;
```

3.3.5 属性配置

- 定义Contexts

```
#
android/device/gxatek/common/legacy/sepolicy/vendor/SystemLog/np_types/prope
rty_contexts
vendor.qnx.tag                                  u:object_r:system_prop:s0   #
系统属性是google平台定义的
vendor.logexporter
u:object_r:np_systemlog_deamon_prop:s0
vendor.logexporter.path
u:object_r:np_systemlog_deamon_prop:s0
```

- 定义Type

```
#
android/device/gxatek/common/legacy/sepolicy/public/SystemLog/np_types/prope
rty.te
type np_systemlog_deamon_prop, property_type;
```

- 定义规则

```
#
android/device/gxatek/common/legacy/sepolicy/private/SystemLog/mix_rules/log
exporter.te
allow np_logexporter_deamon np_systemlog_deamon_prop:file getattr;
# 以上规则的意思是：允许 np_logexporter_deamon类型  获取np_systemlog_deamon_prop域
的属性
```

3.4 检查编译引用

以上规则我们配置好之后，需要确认好策略mk脚本引用链，否则可能导致配置了，无法编入到系统。

3.4.1 非平台策略引用链

```
# android/device/qcom/msmnile_gvmq/AndroidProducts.mk
$(LOCAL_DIR)/msmnile_gvmq.mk
||
# android/device/qcom/msmnile_gvmq/msmnile_gvmq.mk
include device/qcom/msmnile_gvmq/gxatek/x5r.mk
||
# android/device/qcom/msmnile_gvmq/gxatek/x5r.mk
include device/gxatek/common/firewall.mk    # 到模块了
||
# android/device/gxatek/common/firewall.mk
BOARD_SEPOLICY_DIRS +=
device/gxatek/common/legacy/sepolicy/vendor/Firewall/np_rules \

device/gxatek/common/legacy/sepolicy/vendor/Firewall/mix_rules \

device/gxatek/common/legacy/sepolicy/vendor/Firewall/np_types
```

从引用链可以看出，我们的所有Context，Type，规则都配置在`device/gxatek/common/legacy/sepolicy/vendor/Firewall`里面。

3.4.2 平台策略引用链

所有文件都在`android/system/sepolicy`中，但是需要注意的是这里配置之后，需要注意同步修改`android/system/sepolicy/prebuilts/api/<SDK版本>/`目录下的对应文件，保持一直。

3.5 编译策略

```
source build/envsetup.sh
lunch 51
make sepolicy
```

使用以上指令可以对Selinux策略模块单独编译。

3.6 验证编译策略

经过以前经验，经常会有策略没有编译到最终生成的文件中去的情况，因此每次编译策略结束之后，都最后grep检测一下策略是否配置成功了。免得到时候烧机验证失败重新编。完整验证一次太消耗时间了！！！

3.6.1 总策略

策略编译成功之后，所有的规则都在如下目录中，在这个目录下grep就可以找到规则是否配置成功。

```
out/target/product/msmnile_gvmq/obj/ETC/
```

3.6.2 平台策略

最终平台策略会保存在如下目录，如果配置的平台策略，可以在如下目录中grep。

```
# android/out/target/product/msmnile_gvmq/system/etc/selinux
mapping    plat_hwservice_contexts    plat_seapp_contexts
selinux_denial_metadata
plat_and_mapping_sepolicy.cil.sha256  plat_mac_permissions.xml
plat_sepolicy.cil
plat_file_contexts    plat_property_contexts     plat_service_contexts
```

3.6.3 非平台策略

最终非平台策略会保存在如下目录，如果配置的非平台策略，可以在如下目录中grep。

```
# android/out/target/product/msmnile_gvmq/vendor/etc/selinux
plat_pub_versioned.cil  precompiled_sepolicy.plat_and_mapping.sha256
vendor_mac_permissions.xml  vendor_sepolicy.cil  plat_sepolicy_vers.txt
vendor_file_contexts  vendor_property_contexts    vndservice_contexts
precompiled_sepolicy    vendor_hwservice_contexts  vendor_seapp_contexts
```

3.6.4 小结

一般检查策略是否配置成功，就直接在`总策略`目录中找就好了。只要能编译成功，总策略里面有，编译系统会自动将策略分开放到最终系统的`平台/非平台`目录中。

3.7 策略上机验证

3.7.1 刷整包验证

通过全编整个系统之后，导出`output`目录的所有内容，使用`qfile`工具烧写整包验证是否还会报拒绝策略。

3.7.2 刷Android系统测试

通过全编Android系统之后，导出`system.img`和`vendor.img`，然后使用fastboot烧写这两个分区，重启验证是否还会报之前的拒绝策略。

3.7.3 直接更改策略方式验证

如果修改的是非平台规则，可以导出3.6.3小结目录下的所有策略，直接强制替换到车机`/vendor/etc/selinux`目录，然后重启车机验证是否还有拒绝规则报出。

如果是修改的平台规则，就至少要重烧Android系统，策略才生效。

3.7.4 小结

规则配了之后会出现：`一个规则配了，另一个规则又报问题`的现象，因此如果是非平台策略，可以先配了编译完成替换`/vendor/etc/selinux`目录的所有内容先逐步验证，当所负责的app或者进程点检全功能不报拒绝策略之后，再用刷Android系统或者刷整包的方式验证，确定自己模块不报拒绝策略，就完成Selinux配置了。

参考文档

1.linux kernel - Add vendor service to ServiceManager with android treble architecture - Stack Overflow