

# 深入理解 SELinux SEAndroid（第一部分）

原创 2014 年 02 月 16 日 22:11:12

- 87059
- 编辑 删除 按哥的习惯，应该是全部洗剪吹完后再发，不过今年是马年，什么都强调 马上。所以 现在就先奉献 马上有第一部分 祝各位同仁，朋友 马年快乐。

## 深入理解 SELinux SEAndroid

SEAndroid 是 Google 在 Android 4.4 上正式推出的一套以 SELinux 为基础于核心的系统安全机制。而 SELinux 则是由美国 NSA（国安局）和一些公司（RedHat、Tresys）设计的一个针对 Linux 的安全加强系统。

NSA 最初设计的安全模型叫 FLASK，全称为 Flux Advanced Security Kernel（由 Uta 大学和美国国防部开发，后来由 NSA 将其开源），当时这套模型针对 DTOS 系统。后来，NSA 觉得 Linux 更具发展和普及前景，所以就在 Linux 系统上重新实现了 FLASK，称之为 SELinux。

Linux Kernel 中，SELinux 通过 Linux Security Modules 实现。在 2.6 之前，SELinux 通过 Patch 方式发布。从 2.6 开始，SELinux 正式入驻内核，成为标配。

思考：

## 1 同样是政府部门，差别咋这么大？

## 2 同样涉及操作系统和安全相关，NSA 为何敢用 Linux，为什么想方设法要开源？

由于 Linux 有多种发行版本，所以各家的 SELinux 表现形式也略有区别。具体到 Android 平台，Google 对其进行了一定得修改，从而得到 SEAndroid。

本文将先对 SELinux 相关知识进行介绍，然后看看 Android 是如何实现 SELinux 的（咱们只看用户空间）。

要求：最好能下到 Android 4.4 源码，可  
<http://blog.csdn.net/innost/article/details/14002899>

目标：学完本文，读者应该可以轻松修改相关安全策略文件，以进一步在安全方面定制自己的 Android 系统。

### 一 SELinux 背景知识

#### 1. DAC 和 MAC

SELinux 出现之前，Linux 上的安全模型叫 DAC，全称是 Discretionary Access Control，翻译为自主访问控制。DAC 的核心思想很简单，就是：

- 进程理论上所拥有的权限与执行它的用户的权限相同。比如，以 root 用户启动 Browser，那么 Browser 就有 root 用户的权限，在 Linux 系统上能干任何事情。

显然，DAC 太过宽松了，所以各路高手想方设法都要在 Android 系统上搞到 root 权限。那么 SELinux 如何解决这个问题呢？原来，它在 DAC 之外，设计了一个新的安全模型，叫 MAC (Mandatory Access Control)，翻译为强制访问控制。MAC 的处世哲学非常简单：即任何进程想在 SELinux 系统中干任何事情，都必须先在**安全策略配置文件中**赋予权限。凡是没有出现在安全策略配置文件中的权限，进程就没有该权限。来看一个 SEAndroid 中设置权限的例子：

[例子 1]

```
/*  
  
    from external/sepolicy/netd.te  
  
    下面这条 SELinux 语句表示 允许 (allow ) netd 域 (domain) 中的进程  ”写 (write)  
    “  
  
    类型为 proc 的文件  
  
    注意，SELinux 中安全策略文件有自己的一套语法格式，下文我们将详细介绍它  
  
*/  
  
allow netd proc:file write
```

如果没有在 `netd.te` 中使用上例中的权限配置 `allow` 语句，则 `netd` 就无法往 `/proc` 目录下得任何文件中写数据，即使 `netd` 具有 `root` 权限。

显然，MAC 比 DAC 在权限管理这一块要复杂，要严格，要细致得多。

那么，关于 DAC 和 MAC，此处笔者总结了几个知识点：

- Linux 系统先做 DAC 检查。如果没有通过 DAC 权限检查，则操作直接失败。通过 DAC 检查之后，再做 MAC 权限检查。
- SELinux 中也有用户的概念，但它和 Linux 中原有的 `user` 概念不是同一个东西。什么意思呢？比如，Linux 中的超级用户 `root` 在 SELinux 中可能就是一个没权限，没地位，打打酱油的“路人甲”。

当然，这一切都由 SELinux 安全策略的制定者来决定。

通过上述内容，读者应该能感觉到，在 SELinux 中，安全策略文件是最重要的。确实如此。事实上，对本文的读者而言，学习 SELinux 的终极目标应该是：

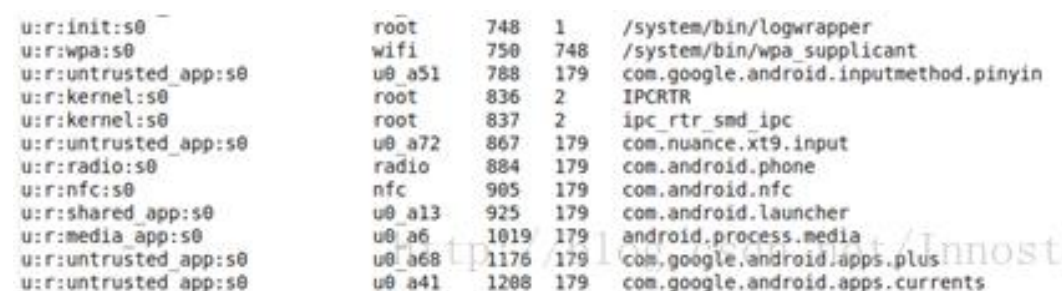
- 看懂现有的安全策略文件。
- 编写符合特定需求的安全策略文件。

前面也曾提到，SELinux 有自己的一套规则来编写安全策略文件，这套规则被称之为 SELinux Policy 语言。它是掌握 SELinux 的重点。

## 2. SELinux Policy 语言介绍

Linux 中有两种东西，一种死的（Inactive），一种活的（Active）。死的东西就是文件（Linux 哲学，万物皆文件。注意，万不可狭义解释为 File），而活的东西就是进程。此处的“死”和“活”是一种比喻，映射到软件层面的意思是：进程能发起动作，例如它能打开文件并操作它。而文件只能被进程操作。

SELinux 中，每种东西都会被赋予一个安全属性，官方说法叫 Security Context。Security Context（以后用 SContext 表示）是一个字符串，主要由三部分组成。例如 SEAndroid 中，进程的 SContext 可通过 **ps -Z** 命令查看，如图 1 所示：



SContext	PID	PPID	Command
u:r:init:s0	748	1	/system/bin/logwrapper
u:r:wpa:s0	750	748	/system/bin/wpa_supplicant
u:r:untrusted_app:s0	788	179	com.google.android.inputmethod.pinyin
u:r:kernel:s0	836	2	IPCRTR
u:r:kernel:s0	837	2	ipc_rtr_smd_ipc
u:r:untrusted_app:s0	867	179	com.nuance.xt9.input
u:r:radio:s0	884	179	com.android.phone
u:r:nfc:s0	905	179	com.android.nfc
u:r:shared_app:s0	925	179	com.android.launcher
u:r:media_app:s0	1019	179	android.process.media
u:r:untrusted_app:s0	1176	179	com.google.android.apps.plus
u:r:untrusted_app:s0	1208	179	com.google.android.apps.currents

图 1 Nexus 7 ps -Z 结果图

图 1 中最左边的那一列是进程的 SContext，以第一个进程 /system/bin/logwrapper 的 SContext 为例，其值为 u:r:init:s0，其中：

- u 为 user 的意思。SEAndroid 中定义了一个 SELinux 用户，值为 u。
- r 为 role 的意思。role 是角色之意，它是 SELinux 中一种比较高层次，更方便的权限管理思路，即 Role Based Access Control（基于角

色的访问控制，简称为 RBAC）。简单点说，一个 u 可以属于多个 role，不同的 role 具有不同的权限。RBAC 我们到最后再讨论。

- init，代表该进程所属的 Domain 为 init。MAC 的基础管理思路其实不是针对上面的 RBAC，而是所谓的 Type Enforcement Access Control（简称 TEAC，一般用 TE 表示）。对进程来说，Type 就是 Domain。比如 init 这个 Domain 有什么权限，都需要通过[例子 1] 中 allow 语句来说明。
- S0 和 SELinux 为了满足军用和教育行业而设计的 Multi-Level Security（MLS）机制有关。简单点说，MLS 将系统的进程和文件进行了分级，不同级别的资源需要对应级别的进程才能访问。后文还将详细介绍 MLS。

再来看文件的 SContext，读者可通过 `ls -Z` 来查看，如图 2 所示：

```
drwx----- root      root      u:object_r:rootfs:s0 root
drwxr-x--- root      root      u:object_r:rootfs:s0/sbin
lrwxrwxrwx root      root      u:object_r:rootfs:s0/sdcard -> /storage/emulated/legacy
-rw-r--r-- root      root      u:object_r:rootfs:s0/seapp_contexts
-rw-r--r-- root      root      u:object_r:rootfs:s0/sepolicy
drwxr-x--x root      sdcard_r u:object_r:rootfs:s0/storage
dr-xr-xr-x root      root      u:object_r:systemfs:s0/sys
drwxr-xr-x root      root      u:object_r:system_file:s0/system
drwxr-xr-x root      root      u:object_r:rootfs:s0/tmp-mksh
lrwxrwxrwx root      root      u:object_r:rootfs:s0/tombstones -> /data/tombstones
```

图 2 Nexus 7 `ls -Z` 结果图

图 2 中，倒数第二列所示为 Nexus 7 根目录下几个文件和目录的 SContext 信息，以第一行 root 目录为例，其信息为 `u:object_r:rootfs:s0`：

- u：同样是 user 之意，它代表创建这个文件的 SELinux user。

- `object_r`: 文件是死的东西，它没法扮演角色，所以在 SELinux 中，死的东西都用 `object_r` 来表示它的 role。
- `rootfs`: 死的东西的 Type，和进程的 Domain 其实是一个意思。它表示 root 目录对应的 Type 是 `rootfs`。
- `s0`: MLS 的级别。

根据 SELinux 规范，完整的 SContext 字符串为：

`user:role:type[:range]`

注意，方括号中的内容表示可选项。`s0` 属于 `range` 中的一部分。下文再详细介绍 `range` 所代表的 Security Level 相关的知识。

看，SContext 的核心其实是前三个部分：`user:role:type`。

刚才说了，MAC 基本管理单位是 TEAC（Type Enforcement Access Control），然后是高一级别的 Role Based Access Control。RBAC 是基于 TE 的，而 TE 也是 SELinux 中最主要的部分。

下面来看看 TE。

## 2.1 TE 介绍

在例子 1 中，大家已经见过 TE 的 `allow` 语句了，再来细致研究下它：

[例子 2]

```
allow netd proc:file write
```

这条语句的语法为：

- **allow**：TE 的 allow 语句，表示授权。除了 allow 之外，还有 allowaudit、donaudit、neverallow 等。
- **netd**：source type。也叫 subject，domain。
- **proc**：target type。它代表其后的 file 所对应的 Type。
- **file**：代表 Object Class。它代表能够给 subject 操作的一类东西。  
例如 File、Dir、socket 等。在 Android 系统中，有一个其他 Linux 系统没有的 Object Class，那就是 Binder。
- **write**：在该类 Object Class 中所定义的操作。

根据 SELinux 规范，完整的 allow 相关的语句格式为：

```
rule_name source_type target_type : class perm_set
```

我们直接来看几个实例：

[例子 3]

```
//SEAndroid 中的安全策略文件 policy.conf
```

```
#允许 zygote 域中的进程向 init type 的进程（Object Class 为 process）发送 sigchld  
信号
```

```
allow zygote init:process sigchld;
```

```
#允许 zygote 域中的进程 search 或 getattr 类型为 appdomain 的目录。注意，多个  
perm_set
```



#可用{}括起来

```
allow zygot appdomain:dir { getattr search };
```

#来个复杂点的:

#source\_type 为 unconfineddomain target\_type 为一组 type, 由

#{ fs\_type dev\_type file\_type }构成。object\_class 也包含两个, 为{ chr\_file file }

#perm\_set 语法比较奇特, 前面有一个~号。它表示除了{entrypoint relabelto}之外,

{chr\_file #file}这两个 object\_class 所拥有的其他操作

```
allow unconfineddomain {fs_type dev_type file_type}:{ chr_file file } \
```

```
~{entrypoint relabelto};
```

#特殊符号除了~外, 还有-号和\*号, 其中:

# 1): -号表示去除某项内容。

# 2): \*号表示所有内容。

#下面这条语句中, source\_type 为属于 appdomain, 但不属于 unconfineddomain 的进程。

#而 \*表示所有和 capability2 相关的权限

#neverallow: 表示绝不允许。

```
neverallow { appdomain -unconfineddomain } self:capability2 *;
```

特别注意，前面曾提到说权限必须显示声明，没有声明的话默认就没有权限。那 `neverallow` 语句就没必要存在了。因为“无权限”是不需要声明的。确实如此，`neverallow` 语句的作用只是在生成安全策略文件时进行检查，判断是否有违反 `neverallow` 语句的 `allow` 语句。例如，笔者修改 `shell.te` 中一个语句后，生成安全策略文件时就检测到了冲突，如图 3 所示：

```
make: Entering directory '/disk/android4.4'
out/host/linux-x86/bin/checkpolicy: loading policy configuration from out/target/product/generic/obj/ETC/sepol
libsepol.check assertion helper: neverallow on line 4002 violated by allow shell runas:process { transition };
Error while expanding policy
make: *** [out/target/product/generic/obj/ETC/sepolicy intermediates/sepolicy] Error 1
```

图 3 `neverallow` 的作用

如图 3 所示，笔者修改 `shell.te` 后，意外导致了一条 `allow` 语句与 `neverallow` 语句冲突，从而生成安全策略文件失败。

下面我们来看上述 `allow` 语句中所涉及到的 `object class` 和 `perm set`。

### (1) *Object class* 和 *Perm Set*

`Object class` 很难用语言说清楚它到底是怎么定义的，所以笔者也不废话，直接告诉大家常见的 `Object class` 有哪些。见下面的 `SEPolicy` 示例：

[external/sepolicy/security\_classes 示例]

.....

#此文件定义了 *Android* 平台中支持的 *Object class*

#根据 SELinux 规范, *Object Class* 类型由 *class* 关键字申明

# file-related classes

*class filesystem*

*class file* #代表普通文件

*class dir* #代表目录

*class fd* #代表文件描述符

*class lnk\_file* #代表链接文件

*class chr\_file* #代表字符设备文件

.....

# network-related classes

*class socket* #socket

*class tcp\_socket*

*class udp\_socket*

.....

*class binder* #Android 平台特有的 *binder*

```
class zygote #Android 平台特有的 zygote
```

#Android 平台特有的属性服务。注意其后的 *userspace* 这个词

```
class property_service # userspace 和用户空间中的 SELinux 权限检查有关，下文再解释
```

上述示例展示了 SEAndroid 中 Object Class 的定义，其中：

- Object Class 需要通过 class 语句申明。这些申明一般放在一个叫 security\_class 的文件中。
- 另外，这些 class 和 kernel 中相关模块紧密结合。

据说：在 kernel 编译时会根据 security\_class 文件生成对应的头文件。从这里可以看出，SELinux 需要根据发行平台来做相应修改。同时可以看出，该文件一般也不需要我们去修改。

再来看 Perm set。Perm set 指得是某种 Object class 所拥有的操作。以 file 这种 Object class 而言，其拥有的 Perm set 就包括 read, write, open, create, execute 等。

和 Object class 一样，SELinux 或 SEAndroid 所支持的 Perm set 也需要声明，来看下面的例子：

```
[external/sepolicy/access_vectors]
```

```
#SELinux 规范中，定义 perm set 有两种方式，一种是使用下面的 common 命令
```

#其格式为: *common common\_name { permission\_name ... }* *common* 定义的 *perm set* 能

#被另外一种 *perm set* 命令 *class* 所继承

#以下是 *Android* 平台中, *file* 对应的权限 (*perm set*)。其大部分权限读者能猜出是干什么的。

#有一些权限需要结合文后的参考文献来学习

*common file {*

*ioctl read write create getattr setattr lock relabelfrom relabelto*

*append unlink link rename execute swapon quotaon mounton }*

#除了 *common* 外, 还有一种 *class* 命令也可定义 *perm set*, 如下面的例子:

#*class* 命令的完整格式是:

#*class class\_name [ inherits common\_name ] { permission\_name ... }*

#*inherits* 表示继承了某个 *common* 定义的权限 注意, *class* 命令定义的权限其实针对得就是

#某个 *object class*。它不能被其他 *class* 继承

*class dir inherits file {*

```
add_name remove_name reparent search rmdir open audit_access execmod
}
```

#来看 SEAndroid 中的 *binder* 和 *property\_service* 这两个 *Object class* 定义了哪些操作权限

```
class binder {

    impersonate call set_context_mgr transfer }

class property_service { set }
```

提示：Object class 和 Perm set 的具体内容（SELinux 中其实叫 Access Vector）都和 Linux 系统/Android 系统密切相关。所以，从知识链的角度来看，Linux 编程基础很重要。

## （2） *type*, *attribute* 和 *allow* 等

现在再来看 *type* 的定义，和 *type* 相关的命令主要有三个，如下面的例子所示：

[external/sepolicy 相关文件]

```
#type 命令的完整格式为：type type_id [alias alias_id,] [attribute_id]
```

#其中，方括号中的内容为可选。*alias* 指定了 *type* 的别名，可以指定多个别名。

#下面这个例子定义了一个名为 *shell* 的 *type*，它和一个名为 *domain* 的属性（*attribute*）关联

```
type shell, domain; #本例来自 shell.te, 注意, 可以关联多个 attribute
```

#属性由 *attribute* 关键字定义, 如 *attributes* 文件中定义的 *SEAndroid* 使用的属性有:

```
attribute domain
```

```
attribute file_type
```

#可以在定义 *type* 的时候, 直接将其和某个 *attribute* 关联, 也可以单独通过

#*typeattribute* 将某个 *type* 和某个或多个 *attribute* 关联起来, 如下面这个例子

#将前面定义的 *system* 类型和 *mltrustedsubject* 属性关联了起来

```
typeattribute system mltrustedsubject
```

**特别注意:** 对初学者而言, *attribute* 和 *type* 的关系最难理解, 因为“*attribute*”这个关键词实在是没取好名字, 很容易产生误解:

- 实际上, *type* 和 *attribute* 位于同一个命名空间, 即不能用 *type* 命令和 *attribute* 命令定义相同名字的东西。
- 其实, *attribute* 真正的意思应该是类似 *type* (或 *domain*) group 这样的概念。比如, 将 *type A* 和 *attribute B* 关联起来, 就是说 *type A* 属于 *group B* 中的一员。

使用 `attribute` 有什么好处呢？一般而言，系统会定义数十或数百个 `Type`，每个 `Type` 都需要通过 `allow` 语句来设置相应的权限，这样我们的安全策略文件编起来就会非常麻烦。有了 `attribute` 之后呢，我们可以将这些 `Type` 与某个 `attribute` 关联起来，然后用一个 `allow` 语句，直接将 `source_type` 设置为这个 `attribute` 就可以了：

- 这也正是 `type` 和 `attribute` 位于同一命名空间的原因。
- 这种做法实际上只是减轻了 TE 文件编写者的烦恼，安全策略文件在编译时会将 `attribute` 拓展为其包含的 `type`。如例子 4 所示：

[例子 4]

```
#定义两个 type，分别是 A_t 和 B_t，它们都管理到 attribute_test
```

```
type A_t attribute_test;
```

```
type B_t attribute_test;
```

```
#写一个 allow 语句，直接针对 attribute_test
```

```
allow attribute_test C_t:file {read write};
```

```
#上面这个 allow 语句在编译后的安全策略文件中会被如下两条语句替代：
```

```
allow A_t C_t:file {read write};
```

```
allow B_t C_t:file {read write};
```



前面讲过，TE 的完整格式为：

```
rule_name source_type target_type : class perm_set
```

所以，attribute 可以出现在 source\_type 中，也可以出现在 target\_type 中。

提示：一般而言，定义 type 的时候，都会在名字后添加一个\_t 后缀，例如 type system\_t。而定义 attribute 的时候不会添加任何后缀。但是 Android 平台没使用这个约定俗成的做法。不过没关系，SEAndroid 中定义的 attribute 都在 external/sepolicy/attribute 这个文件中，如果分不清是 type 还是 attribute，则可以查看这个文件中定义了哪些 attribute。

最后我们来看看 TE 中的 rule\_name，一共有四种：

- allow：赋予某项权限。

**allowaudit**：audit 含义就是记录某项操作。默认情况下是 SELinux 只记录那些权限检查失败的操作。allowaudit 则使得权限检查成功的操作也被记录。注意，allowaudit 只是允许记录，它和赋予权限没关系。赋予权限必须且只能使用 allow 语句。**dontaudit**：对那些权限检查失败的操作不做记录。**neverallow**：前面讲过，用来检查安全策略文件中是否有违反该项规则的 allow 语句。

如例子 5 所示：

[例子 5]

```
#来自 external/sepolicy/netd.te 文件
```

```
#永远不允许 netd 域中的进程 读写 dev_type 类型的 块设备文件（Object class 为 blk_file）
```

```
neverallow netd dev_type:blk_file { read write }
```

### (3) RBAC 和 constrain

绝大多数情况下，SELinux 的安全配置策略需要我们编写各种各样的 `xx.te` 文件。由前文可知，`.te` 文件内部应该包含包含了各种 `allow`，`type` 等语句了。这些都是 TEAC，属于 SELinux MAC 中的核心组成部分。

在 TEAC 之上，SELinux 还有一种基于 Role 的安全策略，也就是 RBAC。RBAC 到底是如何实施相关的权限控制呢？我们先来看 SEAndroid 中 Role 和 User 的定义。

[external/sepolicy/roles]

*#Android 中只定义了一个 role，名字就是 r*

```
role r;
```

*#将上面定义的 r 和 attribute domain 关联起来*

```
role r types domain;
```

再来看 user 的定义。

[external/sepolicy/users]

*#支持 MLS 的 user 定义格式为：*

```
#user seuser_id roles role_id level mls_level range mls_range;
```

*#不支持 MLS user 定义格式为：*

```
#user seuser_id roles role_id;
```

#SEAndroid 使用了支持 *MLS* 的格式。下面定义的这个 *user u*，将和 *role r* 关联。

#注意，一个 *user* 可以和多个 *role* 关联。

#*level* 之后的是该 *user* 具有的安全级别。*s0* 为最低级，也就是默认的级别，

*mls\_systemHigh*

#为 *u* 所能获得的最高安全级别（*security level*）。此处暂且不表 *MLS*

```
user u roles { r } level s0 range s0 - mls_systemhigh;
```

那么，Roles 和 User 中有什么样的权限控制呢？

1) 首先，我们应该允许从一个 *role* 切换（SELinux 用 Transition 表达切换之意）到另外一个 *role*，例如：

#注意，关键字也是 *allow*，但它和前面 *TE* 中的 *allow* 实际上不是一种东西

#下面这个 *allow* 允许 *from\_role\_id* 切换到 *to\_role\_id*

```
allow from_role_id to_role_id;
```

2) 角色之间的关系。SELinux 中，Role 和 Role 之间的关系和公司中的管理人员的层级关系类似，例如：

#*dominance* 语句属于 *deprecated* 语句，*MLS* 中有新的定义层级相关的语句。不过此处要介绍的是

*#selinux* 中的层级关系

*#*下面这句话表示 *super\_r* *dominate* (统治, 关键词 *dom*) *sysadm\_r* 和 *secadm\_r* 这两个角色

*#*反过来说, *sysadm\_r* 和 *secadm\_r* *dominate by* (被统治, 关键词 *domby*) *super\_r*

*#*从 *type* 的角度来看, *super\_r* 将自动继承 *sysadm\_r* 和 *secadm\_r* 所关联的 *type* (或 *attribute*)

```
dominance { role super_r {role sysadm_r; role secadm_r; }
```

3) 其他内容, 由于 SEAndroid 没有使用, 此处不表。读者可阅读后面的参考文献。

话说回来, 怎么实现基于 Role 或 User 的权限控制呢? SELinux 提供了一个新的关键词, 叫 *constrain*, 来看下面这个例子:

[例子 6]

*#constrain* 标准格式为:

```
# constrain object_class_set perm_set expression ;
```

*#*下面这句话表示只有 *source* 和 *target* 的 *user* 相同, 并且 *role* 也相同, 才允许

*#write object\_class* 为 *file* 的东东

```
constrain file write (u1 == u2 and r1 == r2) ;
```

前面已经介绍过 `object_class` 和 `perm_set` 了，此处就不再赘述。`constrain` 中最关键的是 `experssion`，它包含如下关键词：

- `u1,r1,t1`：代表 source 的 user, role 和 type。
- `u2,r2,t2`：代表 target 的 user,role 和 type。
- `==`和`!=`：`==`表示相等或属于，`!=`表示不等或不属于。对于 `u,r` 来说，`==`和`!=`表示相等或不等，而当诸如 `t1`“`==`或`!=`”某个 attribute 时，表示源 type 属于或不属于这个 attribute。
- `dom,domby,incomp,eq`：仅针对 role，表示统治，被统治，没关系和相同（和`==`一样）

关于 `constrain`，再补充几个知识点：

- SEAndroid 中没有使用 `constrain`，而是用了 MLS 中的 `mlsconstrain`。所以下文将详细介绍它。
- `constrain` 是对 TEAC 的加强。因为 TEAC 仅针对 Type 或 Domain，没有针对 user 和 role 的，所以 `constrain` 在 TEAC 的基础上，进一步加强了权限控制。在实际使用过程中，SELinux 进行权限检查时，先检查 TE 是否满足条件，然后再检查 `constrain` 是否也满足条件。二者都通过了，权限才能满足。

关于 RBAC 和 `constrain`，我们就介绍到此。

提示：笔者花了很长时间来理解 RBAC 和 `constrain` 到底是想要干什么。其实这玩意很简单。因为 TE 是 Type Enforcement，没 user 和 role 毛事，

而 RBAC 则可通过 `constrain` 语句来在 `user` 和 `role` 上再加一些限制。当然，`constrain` 也可以对 `type` 进行限制。如此而已！

## 2.2 Labeling 介绍

前面陆陆续续讲了些 SELinux 中最常见的东西。不过细心的人可能会问这样一个问题：这些 `SContext` 最开始是怎么赋给这些死的和活的东西的？Good Question！

提示：SELinux 中，设置或分配 `SContext` 给进程或文件的工作叫 Security Labeling，土语叫打标签。

### (1) `sid` 和 `sid_context`

这个问题的回答嘛，其实也蛮简单。Android 系统启动后（其他 Linux 发行版类似），`init` 进程会将一个编译完的安全策略文件传递给 `kernel` 以初始化 `kernel` 中的 SELinux 相关模块（姑且用 `Linux Security Module:LSM` 来表示它把），然后 `LSM` 可根据其中的信息给相关 `Object` 打标签。

提示：上述说法略有不准，先且表述如此。

`LSM` 初始化时所需要的信息以及 `SContext` 信息保存在两个特殊的文件中，以 Android 为例，它们分别是：

- `initial_sids`：定义了 `LSM` 初始化时相关的信息。`SID` 是 SELinux 中一个概念，全称是 `Security Identifier`。`SID` 其实类似 `SContext` 的

key 值。因为在实际运行时，如果老是去比较字符串（还记得吗，SContext 是字符串）会严重影响效率。所以 SELinux 会用 SID 来匹配某个 SContext。

- `initial_sid_context`: 为这些 SID 设置最初的 SContext 信息。

来看这两个文件的内容：

[external/sepolicy/initial\_sids 和 initial\_sid\_context]

#先看 *initial\_sids*

*sid kernel* #sid 是关键词，用于定义一个 *sid*

*sid security*

*sid unlabeled*

*sid fs*

*sid file*

*sid file\_labels*

*sid init*

.....

#再来看 *initial\_sid\_context*

*sid kernel u:r:kernel:s0* #将 *initial\_sids* 中定义的 *sid* 和初始的 SContext 关联起来

```
sid security u:object_r:kernel:s0
```

```
sid unlabeled u:object_r:unlabeled:s0
```

```
sid fs u:object_r:labeledfs:s0
```

```
sid file u:object_r:unlabeled:s0
```

```
sid file_labels u:object_r:unlabeled:s0
```

```
sid init u:object_r:unlabeled:s0
```

**提示：**sid 的细节需要查看 LSM 的实现。此处不拟深究它。另外，这两个文件也是和 Kernel 紧密相关的，所以一般不用修改它们。

## (2) Domain/Type Transition 和宏

SEAndroid 中，init 进程的 SContext 为 u:r:init:s0，而 init 创建的子进程显然不会也不可能拥有和 init 进程一样的 SContext（否则根据 TE，这些子进程也就在 MAC 层面上有了和 init 一样的权限）。那么这些子进程的 SContext 是怎么被打上和其父进程不一样的 SContext 呢？

SELinux 中，上述问题被称为 Domain Transition，即某个进程的 Domain 切换到一个更合适的 Domain 中去。Domain Transition 也是需要我们在安全策略文件中来配置的，而且有相关的关键词，来看例子 7。

[例子 7-1]

```
#先要使用 type_transition 语句告诉 SELinux
```



*#type\_transition* 的完整格式为：

```
# type_transition source_type target_type : class default_type;
```

#对 *Domain Transition* 而言有如下例子：

```
type_transition init_t apache_exec_t : process apache_t;
```

上面这个例子的解释如下，请读者务必仔细：

- 当 **init\_t** Domain 中的进程执行 **type** 为 **apache\_exec\_t** 类型的可执行文件（fork 并 **execv**）时，其 **class**（此处是 **process**）所属 Domain（对 **process** 而言，肯定是指 Domain）需要切换到 **apache\_t** 域。

明白了吗？要做 DT，肯定需要先 fork 一个子进程，然后通过 **execv** 打开一个新的可执行文件，从而进入变成那个可执行文件对应的活物！所以，在 *type\_transition* 语句中，**target\_type** 往往是那个可执行文件（死物）的 **type**。**default\_type** 则表示 **execv** 执行后，这个活物默认的 Domain。

另外，对 DT 来说，**class** 一定会是 **process**。

请注意，DT 属于 Labeling 一部分，但这个事情还没完。因为打标签也需要相关权限。所以，上述 *type\_transition* 不过是开了一个头而已，要真正实施成功这个 DT，还需要下面至少三个 **allow** 语句配合：

[例子 7-2]

#首先，你得让 **init\_t** 域中的进程能够执行 **type** 为 **apache\_exec\_t** 的文件

```
allow init_t apache_exec_t : file execute;
```

#然后, 你还得告诉 SELinux, 允许 *init\_t* 做 DT 切换以进入 *apache\_t* 域

```
allow init_t apache_t : process transition;
```

#最后, 你还得告诉 SELinux, 切换入口(对应为 *entrypoint* 权限)为执行 *apache\_exec\_t* 类型

#的文件

```
allow apache_t apache_exec_t : file entrypoint;
```

为什么会需要上述多达三个权限呢? 这是因为在 Kernel 中, 从 fork 到 execv 一共设置了三处 Security 检查点, 所以需要三个权限。

提示: 读者不必纠结这个了, 按照规范做就完了。不过..., 这导致我们写 TE 文件时候会比较麻烦啊!

确实比较麻烦, 不过 SELinux 支持宏, 这样我们可以定义一个宏语句把上述 4 个步骤全部包含进来。在 SEAndroid 中, 系统定义的宏全在 *te\_macros* 文件中, 其中和 DT 相关的宏定义如下:

[external/sepolicy/te\_macros]

#定义 *domain\_trans* 宏。\$1,\$2 等等代表宏的第一个, 第二个....参数

```
define(`domain_trans',`
```

# SEAndroid 在上述三个最小权限上, 还添加了自己的一些权限

```
allow $1 $2:file { getattr open read execute };
```

```

allow $1 $3:process transition;

allow $3 $2:file { entrypoint read execute };

allow $3 $1:process sigchld;

donaudit $1 $3:process noatsecure;

allow $1 $3:process { siginh rlimitinh };

')

```

#定义 `domain_auto_trans` 宏，这个宏才是我们在 `te` 中直接使用的

#以例子 7 而言，该宏的用法是：

```

#define domain_auto_trans(init_t, apache_exec_t, apache_t)

define(`domain_auto_trans', `

# 先 allow 相关权限

domain_trans($1,$2,$3)

# 然后设置 type_transition

type_transition $1 $2:process $3;

')

```

在 `external/sepolicy/init_shell.te` 中就有上述宏的用法：

```
./init_shell.te:4:domain_auto_trans(init, shell_exec, init_shell)
```

除了 DT 外，还有针对 Type 的 Transition。举个例子，假设目录 A 的 SContext 为 u:r:dir\_a，那么默认情况下在该目录下创建的文件都具有 u:r:dir\_a 这个 SContext。所以我们也要针对死得东西进行打标签。

和 DT 类似，TT 的语句也是 type\_transition，而且要顺利完成 Transition，也需要申请相关权限。废话不再多说，我们直接看 te\_macros 是怎么定义 TT 所需要的宏的：

```
[external/sepolicy/te_macros]
```

```
# 定义 file_type_trans(domain, dir_type, file_type)宏

#

define(`file_type_trans', `

# ra_dir_perms 是一个宏，由 global_macros 文件定义，其值为：

#define(`ra_dir_perms', `{ r_dir_perms add_name write }')

allow $1 $2:dir ra_dir_perms;

# create_file_perms 也是一个宏，定义在 global_macros 文件中，其值为：

# define(`create_file_perms', `{ create setattr rw_file_perms

#                               link_file_perms }')
```

```
#而 r_dir_perms=define(`r_dir_perms', `{ open getattr read search ioctl }
```

```
allow $1 $3:notdevfile_class_set create_file_perms;
```

```
allow $1 $3:dir create_dir_perms;
```

```
)
```

```
# 定义 file_type_auto_trans(domain, dir_type, file_type)宏
```

#该宏的含义是：当 *domain* 域中的进程在某个 *Type* 为 *dir\_type* 的目录中创建文件时，  
该文件的

#*SContext* 应该是 *file\_type*

```
define(`file_type_auto_trans',`
```

```
file_type_trans($1, $2, $3)
```

```
type_transition $1 $2:dir $3;
```

#*notdevfile\_class\_set* 也是一个宏，由 *global\_macros* 文件定义，其值为

```
# define(`notdevfile_class_set', `{ file lnk_file sock_file fifo_file }')
```

```
type_transition $1 $2:notdevfile_class_set $3;
```

```
)
```

WoW，SEAndroid 太这两个宏定义太复杂了，来看看官方文档中的最小声明是什么：

[例子 8]

```
type_transition acct_t var_log_t:file wttmp_t;

allow acct_t var_log_t:dir { read getattr lock search ioctl

                           add_name remove_name write };

allow acct_t wttmp_t:file { create open getattr setattr read

                           write append rename link unlink ioctl lock };
```

在 SEAndroid 的 app.te 中，有如下 TT 设置：

```
./app.te:86:file_type_auto_trans(appdomain, download_file, download_file)
```

DT 和 TT 就介绍到这，翻来覆去就这么点东西，多看几遍就“柜”（用柜字，打一成语，参考 2014 年中国首次猜谜大会）了

=====未完，待续=====

# 深入理解 SELinux SEAndroid 之二

转载 2016 年 06 月 27 日 16:46:33

- 标签:
- *selinux /*
- 分析
- 950
- 编辑 删除 接 第 一 部 分 的 内 容

(<http://blog.csdn.net/innost/article/details/19299937>)。

今天公司年会，哥高兴，所以发布第二部。SELinux/SEAndroid 一共分三部分。第一和第二部分是 SELinux 的基础知识，第三部分是 SEAndroid 的工作源码分析。

## 深入理解 SELinux SEAndroid 第二部分

### 3) File/File System 打 label

前面一节中，读者见识到了 DT 和 TT。不过这些描述的都是 Transition，即从某种 Type 或 Domain 进入另外一种 Type 或 Domain，而上述内容并没有介绍最初的 Type 怎么来。在 SELinux 中，对与 File 相关的死货（比“死东西”少些一个字）还有一些特殊的语句。

直接看 SEAndroid 中的文件吧。

[external/sepolicy/file\_contexts]

#从 file\_contexts 这个文件名也可看出，该文件描述了死货的 SContext

#果然：SEAndroid 多各种预先存在的文件，目录等都设置了初始的 SContext

#注意下面这些\*,?号，代表通配符

/dev(/.\*)? u:object\_r:device:s0

```

/dev/akm8973.*    u:object_r:akm_device:s0
/dev/accelerometer u:object_r:accelerometer_device:s0
/dev/alarm        u:object_r:alarm_device:s0
/dev/android_adb.* u:object_r:adb_device:s0
/dev/ashmem        u:object_r:ashmem_device:s0
/dev/audio.*       u:object_r:audio_device:s0
/dev/binder        u:object_r:binder_device:s0
/dev/block(/.*)?   u:object_r:block_device:s0
.....

```

#注意下面的--号，SELinux 中类似的符号还有：

**#'-b' - Block Device '-c' - Character Device**

**#'-d' - Directory '-p' - Named Pipe**

**#'-l' - Symbolic Link '-s' - Socket**

**#'--' - Ordinary file**

```

/system(/.*)?    u:object_r:system_file:s0
/system/bin/ash   u:object_r:shell_exec:s0
/system/bin/mksh  u:object_r:shell_exec:s0
/system/bin/sh    -- u:object_r:shell_exec:s0
/system/bin/run-as -- u:object_r:runas_exec:s0
/system/bin/app_process u:object_r:zygote_exec:s0
/system/bin/servicemanager u:object_r:servicemanager_exec:s0
/system/bin/surfaceflinger u:object_r:surfaceflinger_exec:s0
/system/bin/drmserver u:object_r:drmserver_exec:s0

```

上面的内容很简单，下面来个面生的：

[external/sepolicy/fs\_use]

#fs\_use 中的 fs 代表 file system.fs\_use 文件描述了 SELinux 的 labeling 信息

#在不同文件系统时的处理方式

#对于常规的文件系统，SContext 信息存储在文件节点（inode）的属性中，系统可通过 getattr

#函数读取 inode 中的 SContext 信息。对于这种 labeling 方式，SELinux 定义了

#fs\_use\_xattr 关键词。这种 SContext 是永远性得保存在文件系统中

**fs\_use\_xattr** yaffs2 u:object\_r:labeledfs:s0;

**fs\_use\_xattr** jffs2 u:object\_r:labeledfs:s0;

**fs\_use\_xattr** ext2 u:object\_r:labeledfs:s0;

**fs\_use\_xattr** ext3 u:object\_r:labeledfs:s0;



```
fs_use_xattr ext4 u:object_r:labeledfs:s0;
fs_use_xattr xfs u:object_r:labeledfs:s0;
fs_use_xattr btrfs u:object_r:labeledfs:s0;
```

#对于虚拟文件系统,即 Linux 系统运行过程中创建的 VFS,则使用 **fs\_use\_task** 关键字描述

#目前也仅有 **pipefs** 和 **sockfs** 两种 VFS 格式

```
fs_use_task pipefs u:object_r:pipefs:s0;
fs_use_task sockfs u:object_r:sockfs:s0;
```

#还没完,还有一个 **fs\_use\_trans**,它也是用于 Virtual File System,但根据 SELinux 官方

#描述,好像这些 VFS 是针对 pseudo terminal 和临时对象。在具体 labeling 的时候,会根据

#**fs\_use\_trans** 以及 TT 的规则来决定最终的 SContext

#我们以下面这个例子为例:

```
fs_use_trans devpts u:object_r:devpts:s0;
```

#假设还有一条 TT 语句

```
#type_transition sysadm_t devpts : chr_file sysadm_devpts_t:s0;
```

#表示当 sysadm\_t 的进程在 Type 为 **devpts** 下创建一个 chr\_file 时,其 SContext 将是

#sysadm\_devpts\_t:s0。如果没有这一条 TT,则将使用 **fs\_use\_trans** 设置的 SContext:

#**u:object\_r:devpts:s0** 注意,和前面的 TT 比起来,这里并不是以目录为参考对象,而是

#以 **FileSystem** 为参考对象

```
fs_use_trans tmpfs u:object_r:tmpfs:s0;
fs_use_trans devtmpfs u:object_r:device:s0;
fs_use_trans shm u:object_r:shm:s0;
fs_use_trans mqueue u:object_r:mqueue:s0;
```

到此,我们介绍了 **fs\_use\_xattr**, **fs\_use\_task** 和 **fs\_use\_trans**,那么这三

种打标签的方法是否涵盖了所有情况呢?答案肯定是否,因为我们还有一个兄弟没出场呢。

[external/sepolicy/genfs\_context]

#genfs 中的 gen 为 generalized 之意，即上述三种情况之外的死货，就需要使用 **genfscon**

#关键词来打 labeling 了。一般就是/目录，proc 目录，sysfs 等

**genfscon rootfs / u:object\_r:rootfs:s0**

**genfscon proc / u:object\_r:proc:s0**

**genfscon proc /net/xt\_qtaguid/ctrl u:object\_r:qtaguid\_proc:s0**

.....

到此，绝大部分能想到的死货怎么打标签就介绍完了。

#### (4) 给网络数据包/端口打标签

不过，从知识完整性角度看，还有对网络数据包打标签的工作，这也是 SELinux 新增的功能。不过，它涉及到与 iptables 相关的工作，所以笔者也不想过多讨论。在 SEAndroid 中，selinux-network.sh 脚本就是来干这个事情的，其内容如图 4 所示：

```
#!/system/bin/sh
IPTABLES="/system/bin/iptables"

#IPTABLES -t security -A INPUT -i wlan0 -j SECMARK --selctx u:object_r:packet:s0
#IPTABLES -t security -A INPUT -i lo -j SECMARK --selctx u:object_r:lo_packet:s0
#IPTABLES -t security -A INPUT -i ppp0 -j SECMARK --selctx u:object_r:ppp0_packet:s0
#IPTABLES -t security -A INPUT -i ppp1 -j SECMARK --selctx u:object_r:ppp1_packet:s0
#IPTABLES -t security -A INPUT -i ppp2 -j SECMARK --selctx u:object_r:ppp2_packet:s0
#IPTABLES -t security -A INPUT -i ppp3 -j SECMARK --selctx u:object_r:ppp3_packet:s0

#IPTABLES -t security -A OUTPUT -o wlan0 -j SECMARK --selctx u:object_r:packet:s0
#IPTABLES -t security -A OUTPUT -o lo -j SECMARK --selctx u:object_r:lo_packet:s0
#IPTABLES -t security -A OUTPUT -o ppp0 -j SECMARK --selctx u:object_r:ppp0_packet:s0
#IPTABLES -t security -A OUTPUT -o ppp1 -j SECMARK --selctx u:object_r:ppp1_packet:s0
#IPTABLES -t security -A OUTPUT -o ppp2 -j SECMARK --selctx u:object_r:ppp2_packet:s0
#IPTABLES -t security -A OUTPUT -o ppp3 -j SECMARK --selctx u:object_r:ppp3_packet:s0
```

图 4 网络数据包打标签

由图 4 可以看出，SEAndroid 暂时也没放开网络数据包打标签的功能。

"-j SECMARK --selctx SContext"是 iptables（需要支持 SELinux 功能）新增选项，用来给各种数据包也打上标签。

除了数据包外，还可以给端口打标签，这是由 `portcon` 关键词来完成的。

此处不再详述，读者有个概念即可。

## 2.3 Security Level 和 MLS

### (1) Security Level

上文介绍的 TE, RBAC 基本满足了“平等社会”条件下的权限管理，但它无法反映现实社会中等级的概念。为此，SELinux 又添加了一种新的权限管理方法，即 Multi-Lever Security，多等级安全。多等级安全信息也被添加到 SContext 中。所以，在 MLS 启用的情况下（注意，你可以控制 SELinux 启用用 MLS 还是不启用 MLS），完整的 SContext 由

- MLS 未启用前： `user_u:role_r:type_t`。
- MLS 启用后， `user:role:type:sensitivity[:category,...]- sensitivity[:category,...]`。

看，MLS 启用后，SContext type 后面的字段变得非常复杂，看着有些头晕（至少笔者初学它时是这样的）。下面马上来解释它。

[Security-level 解析]

```
|-->low security level<--| - |-->high security level<--|  
sensitivity[:category,...] - sensitivity [:category,...]
```

上述字符串由三部分组成：

- `low security level`：表明当前 SContext 所对应的东西（活的或死的）的当前（也就是最小）安全级别。
- 连字符“-”，表示 range

- **high security level**: 表明当前 SContext 所对应的东西（活的或死的）的最高可能获得的安全级别（英文叫 **clearance**，不知道笔者的中文解释是否正确）。

security level 由两部分组成, 先来看第一部分由 **sensitivity** 关键字定义的 **sensitivity**, 其用法见如下例子:

[例子 9]

#用 **sensitivity** 定义一个 **sens\_id**, **alias** 指定别名。

**sensitivity sens\_id alias alias\_id [ alias\_id ];**

#比如:

**sensitivity s0 alias unclassified**

**sensitivity s1 alias secret**

**sensitivity s2 alias top-secret**

.....

#Question: 从 **alias** 看, 似乎 **s0** 的级别<**s1** 的级别<**s2** 的级别。但是

#**alias** 并不是 **sensitivity** 的必要选项, 而且名字可以任取。

#在 SELinux 中, 真正设置 **sensitivity** 级别的是由下面这个关键词表示

**dominance {s0 s1 s2.....sn}**

#在上述 **dominance** 语句中, 括号内最左边的 **s0** 级别最低, 依次递增, 直到最右边的 **sn** 级别最高

再来看 security level 第二部分, 即 **category** 关键字及用法, 如例 10 所示:

[例子 10]

#**category cat\_id alias alias\_id;**

#比如:

**category c0**

**category c1 #等**

#**category** 和 **sensitivity** 不同, 它定义的是类别, 类别之间是没有层级关系的。

比如,

#小说可以是一中 **category**, 政府公文是另外一种 **category**,

SEAndroid 中:

- sensitivity 只定义了 s0
- category 定义了从 c0 到 c1023, 共 1024 个 category。

sensitivity 和 category 一起组成了一个 security level (以后简称 SLevel), SLevel 由关键字 level 声明, 如下例所示:

[例子 11]

```
#level sens_id [ :category_id ];
```

#注意, SLevel 可以没有 category\_id。看一个例子:

#sensitivity 为 s0, category 从 c0, c1,c2 一直到 c255, 注意其中的.号

```
level s0:c0.c255;
```

#没有 category\_id, 如:

```
level s0
```

和 Role 类似, SL1 和 SL2 之间的关系有:

- dom: 如果 SL1 dom SL2 的话, 则 SL1 的 sensitivity  $\geq$  SL2 的 sensitivity, SL1 的 category 包含 SL2 的 category(即 Category of SL1 是 Category of SL2 的超集)。

例如:

```
SL1="s2:c0.c5" dom SL2="s0:c2,c3"
```

- domby: 和 dom 相反。
- eq: sensitivity 相等, category 相同。
- incomp: 不可比。sensitivity 不可比, category 也不可比。

现在回过头来看 SContext, 其完整格式为:

```
user:role:type:sensitivity[:category,...]- sensitivity [:category,...]
```

#前面例子中, 我们看到 **Android** 中, SContext 有:

**u:r:init:s0** #在这种 case 中, Low SLevel 等于 High SLevel, 而且 SLevel 没有包含 Category

好了，知道了 SLevel 后，下面来看看它如何在 MAC 中发挥自己的力量。

和 constrain 类似，MLS 在其基础上添加了一个功能更强大的 mlsconstrain 关键字。

## (2) mlsconstrain 和 no read down/write up

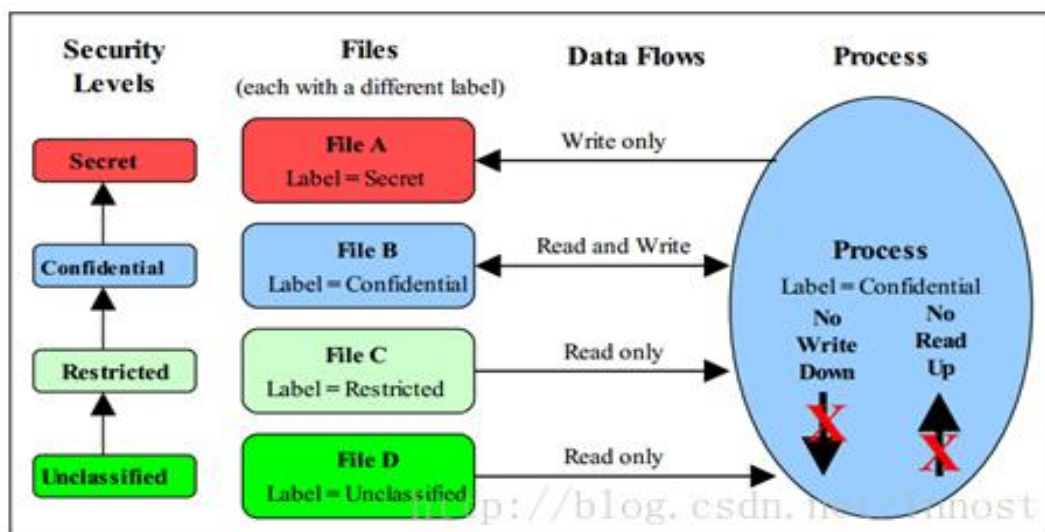
mlsconstrain 语法和 constrain 一样一样的：

**mlsconstrain class perm\_set expression;**

和 constrain 不一样的是，expression 除了 u1,u2,r1,r2,t1,t2 外还新增了：

- l1,l2：小写的 L。l1 表示源的 low sensitivity level。l2 表示 target 的 low sensitivity。
- h1,h2：小写的 H。h1 表示源的 high sensitivity level。h2 表示 target 的 high sensitivity。
- l 和 h 的关系，包括 dom,domby,eq 和 incomp。

mlsconstrain 只是一个 Policy 语法，那么我们应该如何充分利用它来体现多层级安全管理呢？来看图 5。



## 图 5 MLS 的作用

MLS 在安全策略上有一个形象的描述叫 no write down 和 no read up:

- 高级别的东西不能往低级别的东西里边写数据：这样可能导致高级别的数据泄露到低级别中。如图 4 中，Process 的级别是 Confidential，它可以往同级别的 File B 中读写数据，但是只能往高级别的 File A(级别是 Secret)里边写东西。
- 高级别的东西只能从低级别的东西里边读数据：比如 Process 可以从 File C 和 File D 中读数据，但是不能往 File C 和 File D 上写数据。

反过来说：

1 低级别的东西只能往高级别的东西里边写数据

-----我和小伙伴们解释这一条的时候，小伙伴惊呆了，我也惊呆了。他们的想法是“低级别往高级别里写，岂不是把数据破坏了？”。晕！这里讨论的是泄不泄密的问题，不是讨论数据被破坏的事情。破坏就破坏了，只要没泄密就完了。

2 低级别的东西不能从高级别的东西那边读数据

### (3) MLS in SEAndroid

再来看看 SEAndroid 中的 MLS:

- 首先，系统中只有一个 sensitivity level，即 s0。
- 系统中有 1024 个 category，从 c0 到 c1023。

读者通过 `mmm external/sepolicy --just-print` 可以打印出 sepolicy 的 makefile 执行情况，其中有这样的内容：

#m4 用来处理 Policy 文件中的宏

m4 -D mls\_num\_sens=1 -D mls\_num\_cats=1024

在 external/sepolicy/mls 文件中有：

[external/sepolicy/mls]

#SEAndroid 定义的两个和 MLS 相关的宏，位于 mls\_macro 文件中

**gen\_sens(mls\_num\_sens) #mls\_num\_sens=1**

**gen\_cats(mls\_num\_cats) #mls\_num\_cats=1024**

#下面这个宏生成 SLevel

**gen\_levels(mls\_num\_sens,mls\_num\_cats)**

没必要解释上面的宏了，最终的 policy.conf 中（2.4 节将介绍它是怎么来的），我们可以看到：

[out/target/product/generic/obj/ETC/sepolicy\_intermediates/policy.conf]

**sensitivity s0;**

**dominance { s0 }**

**category c0;**

.....#目前能告诉大家的是，policy.conf 文件中，宏，attribute 等都会被一一处理喔！

**category c1023;**

**level s0:c0.c1023; #定义 SLevel**

#SEAndroid 中，mls\_systemlow 宏取值为 s0

#mls\_systemhigh 宏取值为 s0:c0.c1023

**user u roles { r } level s0 range s0 - s0:c0.c1023; #定义 u**

最后，来看一下 mlsconstain 的例子：

[例子 12]

mlsconstrain dir search

(( l1 dom l2 ) or

(( t1 == **mlsfilereadtocl**r ) and ( h1 dom l2 )) or

( t1 == **mlsfileread** ) or

( t2 == **mlstrustedobject** ));

#上述标粗体的都是 attribute

不解释！

## 2.4 编译安全策略文件



到此，SELinux Policy 语言中的基本要素都讲解完毕，相信读者对着真实的策略文件再仔细研究下就能彻底搞明白。

不过，我们前面反复提到的安全策略文件到底是什么？我们前面看到的例子似乎都是文本文件，难道就它们是安全策略文件吗？

拿个例子说事，来看图 6 中 Android 的策略文件：

```
root@innost:/thunderst/disk2/android4.4/external/sepolicy# ls
Android.mk      global_macros    netd.te         servicemanager.te
NOTICE          gpsd.te          nfc.te          shared_app.te
README          hci_attach.te   ping.te         shell.te
access_vectors  healthd.te       platform_app.te su.te
adbd.te         hostapd.te       policy.conf     su_user.te
app.te          init.te          policy_capabilities
attributes      init_shell.te    property.te     surfaceflinger.te
bluetooth.te    initial_sid_contexts
bluetoothd.te   initial_sids      property_contexts
clatd.te        installd.te       qemuclatd.te   system.te
dbusd.te        isolated_app.te   racoon.te      te_macros
debuggerd.te    kernel.te         radio.te        tee.te
device.te       keys.conf         release_app.te  tools
dhcp.te         keystore.te       rild.te        ueventd.te
dnsmasq.te      mac_permissions.xml
domain.te       media_app.te      roles           unconfined.te
drmserver.te    mediaserver.te   runas.te       untrusted_app.te
file.te         mls               seapp_contexts users
file_contexts   mls_macros       security_classes vold.te
fs_use          mtp.te           selinux-network.sh
genfs_contexts  net.te           watchdogd.te   wpa_supplicant.te
                                 zygotd.te     zygote.te
```

图 6 Android 策略文件

Android 中，SELinux 的安全策略文件如图 6 所示。这么多文件，如何处理呢？来看图 7：

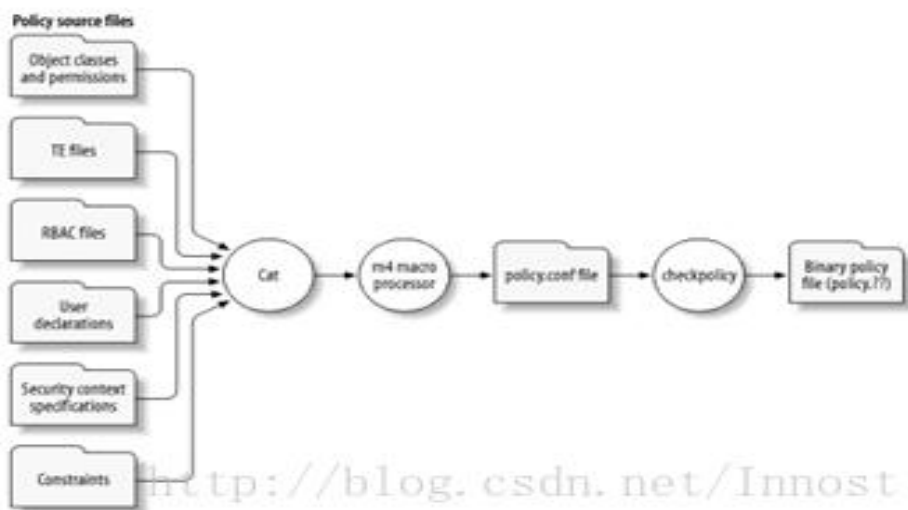


图 7 SELinux 安全配置文件生成

由图 7 可知：

- 左边一列代表安全配置的源文件。也即是大家在图 6 中看到的各种 te 文件，还有一些特殊的文件，例如前文提到的 `initial_sid`，`initial_sid_contexts`，`access_vectors`、`fs_use`、`genfs_contexts` 等。在这些文件中，我们要改的一般也是针对 TE 文件，其他文件由于和 kernel 内部的 LSM 等模块相关，所以除了厂家定制外，我们很难有机会去修改。
- 这些文件都是文本文件，它们会被组合到一起（图 7 中是用 `cat` 命令，不同平台处理方法不相同，但大致意思就是要把这些源文件的内容搞到一起）。
- 搞到一起后的文件中有使用宏的地方，这时要利用 `m4` 命令对这些宏进行拓展。`m4` 命令处理完后得到的文件叫 `policy.conf`。前面我们也见过这个文件了，它是所有安全策略源文件的集合，宏也被替换。所以，读者可以通过 `policy.conf` 文件查看整个系统的安全配置情况，而不用到图 6 中那一堆文件中去找来找去的。
- `policy.conf` 文件最终要被 `checkpolicy` 命令处理。该命令要检查 `neverallow` 是否被违背，语法是否正确等。最后，`checkpolicy` 会将 `policy.conf` 打包生成一个二进制文件。在 SEAndroid 中，该文件叫 `sepolicy`，而在 Linux 发行版本上，一般叫 `policy.26` 等名字。26 表示 SELinux 的版本号。

- 最后，我们再把这个 sepolicy 文件传递到 kernel LSM 中，整个安全策略配置就算完成。

提示：请读者务必将上述步骤搞清楚。

图 8 所示为 SEAndroid 中 sepolicy makefile 的执行情况：

```
make: Entering directory `./disk/android4.4'
mkdir -p out/target/product/generic/obj/ETC/sepolicy_intermediates/
m4 -D mls_num_sens=1 -D mls_num_cats=1024 -s external/sepolicy/security_classes external/sepolicy/initial
s external/sepolicy/access_vectors external/sepolicy/global_macros external/sepolicy/mls_macros external/
licy/mls external/sepolicy/policy_capabilities external/sepolicy/te_macros external/sepolicy/attributes e
nal/sepolicy/adbd.te external/sepolicy/app.te external/sepolicy/bluetooth.te external/sepolicy/bluetoothd
external/sepolicy/clatd.te external/sepolicy/dbusd.te external/sepolicy/debuggerd.te external/sepolicy/de
.te external/sepolicy/dhcp.te external/sepolicy/dnsmasq.te external/sepolicy/domain.te external/sepolicy/
erver.te external/sepolicy/file.te external/sepolicy/gpsd.te external/sepolicy/hci_attach.te external/sep
y/healthd.te external/sepolicy/hostapd.te external/sepolicy/init.te external/sepolicy/init_shell.te exter
sepolicy/installd.te external/sepolicy/isolated_app.te external/sepolicy/kernel.te external/sepolicy/keys
.te external/sepolicy/media_app.te external/sepolicy/mediaserver.te external/sepolicy/mtp.te external/sep
y/net.te external/sepolicy/netd.te external/sepolicy/nfc.te external/sepolicy/ping.te external/sepolicy/p
orm_app.te external/sepolicy/ppp.te external/sepolicy/property.te external/sepolicy/qemud.te external/sep
y/racoon.te external/sepolicy/radio.te external/sepolicy/release_app.te external/sepolicy/rild.te externa
policy/runas.te external/sepolicy/sdcardd.te external/sepolicy/servicemanager.te external/sepolicy/shared
.te external/sepolicy/shell.te external/sepolicy/su.te external/sepolicy/surfaceflinger.te external/sepol
system.te external/sepolicy/tee.te external/sepolicy/ueventd.te external/sepolicy/unconfined.te external/
licy/untrusted_app.te external/sepolicy/vold.te external/sepolicy/watchdogd.te external/sepolicy/wpa_supp
nt.te external/sepolicy/zygote.te external/sepolicy/roles external/sepolicy/users external/sepolicy/initi
id contexts external/sepolicy/fs_use external/sepolicy/genfs_contexts external/sepolicy/port_contexts > o
target/product/generic/obj/ETC/sepolicy_intermediates/policy.conf
sed '/dontaudit/d' out/target/product/generic/obj/ETC/sepolicy_intermediates/policy.conf > out/target/pro
/generic/obj/ETC/sepolicy_intermediates/policy.conf.dontaudit
mkdir -p out/target/product/generic/obj/ETC/sepolicy_intermediates/
out/host/linux-x86/bin/checkpolicy -M -c 26 -o out/target/product/generic/obj/ETC/sepolicy_intermediates/
licy out/target/product/generic/obj/ETC/sepolicy_intermediates/policy.conf
out/host/linux-x86/bin/checkpolicy -M -c 26 -o out/target/product/generic/obj/ETC/sepolicy_intermediates/
olicy.dontaudit out/target/product/generic/obj/ETC/sepolicy_intermediates/policy.conf.dontaudit
echo "Install: out/target/product/generic/root/sepolicy"
mkdir -p out/target/product/generic/root/
out/host/linux-x86/bin/acp -fp out/target/product/generic/obj/ETC/sepolicy_intermediates/sepolicy out/tar
```

图 8 sepolicy makefile 执行情况

看明白了吗？

提示：

想知道如何打印 make 命令的执行情况？请使用“**--just-print**”选项

进阶阅读：

1) 上述做法是将所有源文件打包生成一个单一的安全策略文件，这种方式叫 Monolithic

policy。显然，在什么都模块化的今天，这种方式虽然用得最多，但还是比较土。

SELinux 还支持另外一种所谓的模块化 Policy。这种 Policy 分 Base Policy 和 Module

Policy 两个。BasePolicy 为基础，先加载，然后可以根据情况动态加载 Module Policy

目前 SEAndroid 还没有该功能，不过以后可能会支持。相信有了它，开发定制企业级

安全管理系统就更方便些。

2) 安全策略源文件非常多。基本上，我们都会在一个参考源文件上进行相应修改，

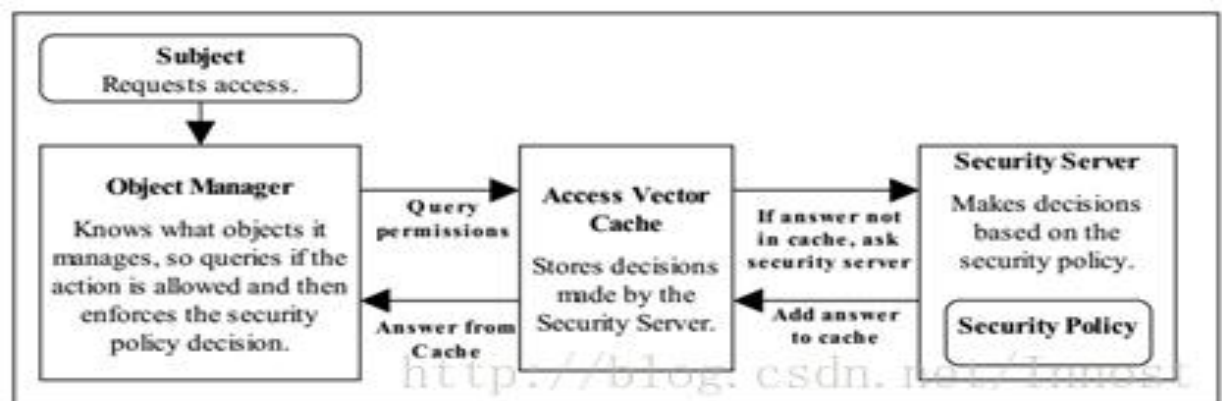
而不会完全从头到尾都自己写。所以，在发行版上有一个 Reference Policy，里边

涵盖了普适的，常用的策略。很明显，AOSP 4.4 中的 sepolicy 也提供了针对 Android

平台的 Reference Policy

## 2.5 拓展讨论

最后，作为拓展讨论，我们来看看 SELinux 作为一套复杂的系统安全模块增强，其实现架构如图 9 所示：



## 图 9 SELinux Component 组成

其中：

- **Subject:** 代表发起操作的对象，一般是 Process。SELinux 需要检查 Subject 是否满足权限要求
- **Object Manager:** 管理着 Object 及相应的 SContext。OM 将向 Access Vector Cache 查询所要求的操作是否有权限。
- **AVC** 主要起一个加速的作用，它将缓存一些权限检查的结果。当相同的权限检查请求过来时，直接从 AVC 中返回所缓存的结果。
- 如果 AVC 没有这条权限检查的结果，那么它将向 Security Server 去查询。SS 内部保存有 SePolicy，它可以根据 SEPolicy 计算出权限检查的结果。

图 9 中所示的 SELinux Component 可以：

- 上述这些模块全部运行在 Kernel 中，它们也是 LSM SELinux 的核心模块。
- OM 和 AVC 可以存在于 UserSpace 中，这种 case 叫 SELinux aware 的 application。说白了，就是一个使用 SELinux 的安全监管系统。在 Android 中，Kernel 和 Userspace 的 SELinux 都使用了。对于 userspace 的 SELinux 相关 app 来说，需要使用开源动态库 libselinux。在 Android 平台中，该库位于 external/libselinux。Userspace 的 SELinux APP 也会和 Kernel 中的 LSM SELinux 交互，所以不能在只有 Kernel SELinux 的系统中单独使用 SELinux app。

图 10 展示了一个完整的 SELinux 系统结构：

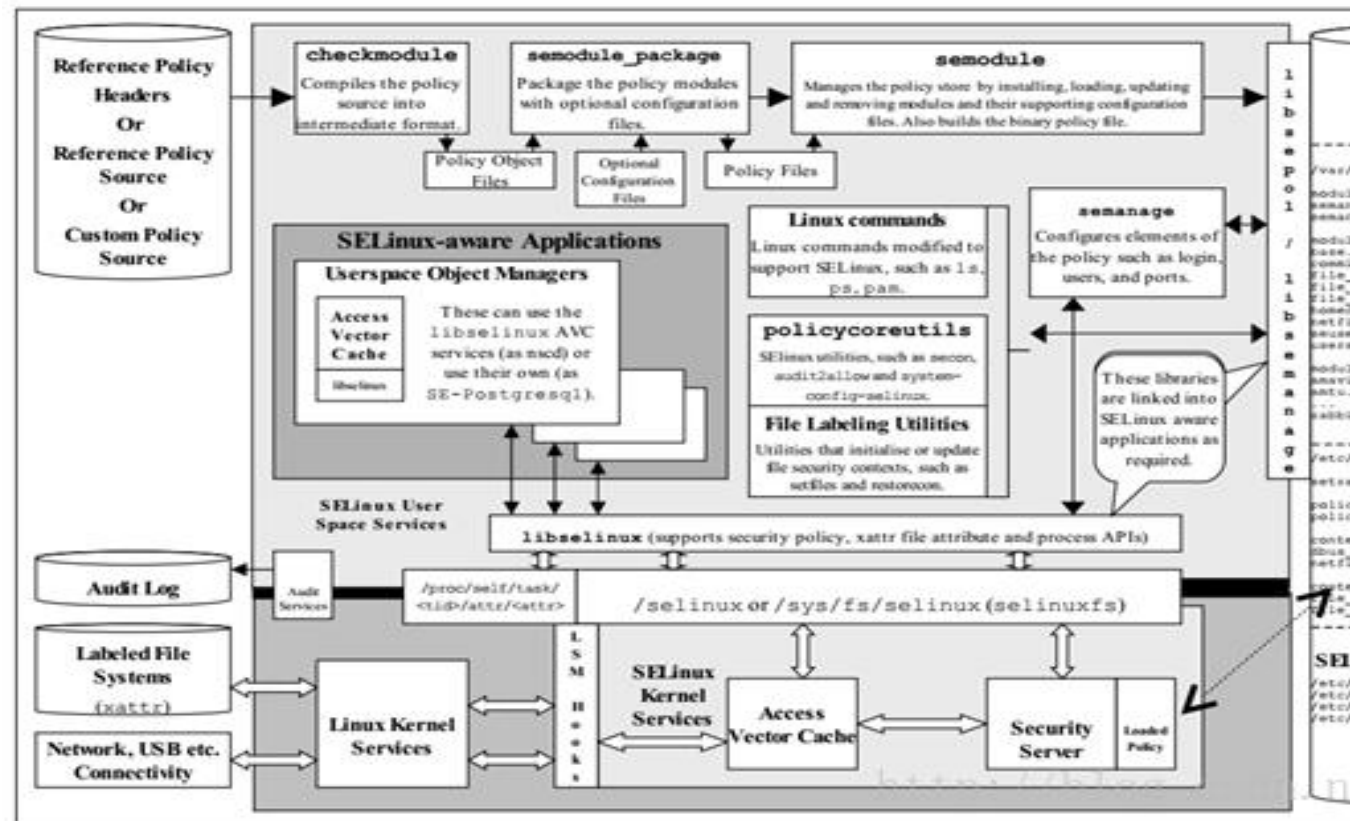


图 10 SELinux 系统结构

图 10 比较复杂，很大的原因是它包含了其他 Linux 发行版本上的一些和 SELinux 相关的工具，我们从上往下看：

- 最 顶 上 ， Reference Policy, checkmodule, semodule\_package, semodule 等讲得都是 Policy 编译相关的工具和参考文件。这些东西编译完后，会生成最右边那个圆柱体 SELinux Policy
- 中间的 SELinux-aware APP, Linux Commands, policycoreutils, file Labeling utils, semanage 等，都是 Linux 发行版中常用的 SELinux 管理工具。



- SELinux-aware APP 借助 libselinux 库,将最右边的 SELinux Policy 配置文件传递到 kernel 中。这其实是通过往系统一些特殊的文件中写数据来完成的。例如/selinux 或/sys/fs/selinux 等。
- 然后我们进入最下边的 Kernel 中的 SELinux,它包含 AVC,和 LSM 挂钩的 LSM Hook, Security Server 等等。

## 2.6 参考文献介绍

SELinux 比较复杂,对于初学者,建议看如下几本书:

### 1 SELinux NSA's Open Source Security Enhanced Linux:

下载地址: <http://download.csdn.net/detail/innost/6947063>

评价:讲得 SELinux 版本比较老,不包括 MLS 相关内容。但是它是极好的入门资料。如果你完全没看懂本文,则建议读本文。

### 2 SELinux by Example Using Security Enhanced Linux:

下载地址: <http://download.csdn.net/detail/innost/6947093>

评价:这本书比第 1 本书讲得 SELinux 版本新,包括 MLS 等很多内容,几乎涵盖了目前 SELinux 相关的所有知识。读者可跳过 1 直接看这本书。

### 3 The\_SELinux\_Notebook\_The\_Foundations\_3rd\_Edition:

下载地址: <http://download.csdn.net/detail/innost/6947077>

评价:这是官方网站上下的文档,但它却是最不适合初学者读的。该书更像一个汇总,解释,手册文档。所以,请务必看完 1 或 2 的基础上再来看它。

# 深入理解 SELinux SEAndroid（最后部分）

原创 2014 年 02 月 23 日 20:46:51

- 38081

- 编辑 删除 接 第 二 部 分 的 内 容

(<http://blog.csdn.net/innost/article/details/19641487>)

SEAndroid 最后一部分

全 文 PDF 下 载 地 址 为 :

<http://vdisk.weibo.com/s/z68f8l0xZUS9w>

## 深入理解 SELinux SEAndroid（结局）

## 二 SEAndroid 源码分析

有了上文的 SELinux 的基础知识，本节再来看看 Google 是如何在 Android 平台定制 SELinux 的。如前文所示，Android 平台中的 SELinux 叫 SEAndroid。

先来看 SEAndroid 安全策略文件的编译。



## 1. 编译 *sepolicy*

Android 平台中：

- **external/sepolicy**：提供了 Android 平台中的安全策略源文件。同时，该目录下的 **tools** 还提供了诸如 **m4,checkpolicy** 等编译安全策略文件的工具。注意，这些工具运行于主机（即不是提供给 Android 系统使用的）
- **external/libselinux**：提供了 Android 平台中的 **libselinux**，供 Android 系统使用。
- **external/libsepol**：提供了供安全策略文件编译时使用的一个工具 **checkcon**。

对我们而言，最重要的还是 **external/sepolicy**。所以先来看它。

读者还记得上文提到的如何查看 **make** 命令的执行情况吗？通过：

```
mmm external/sepolicy --just-print
```

，我们可以看到 **sepolicy** 编译时都干了些什么。

```
#以后用 SEPOLICY_TEMP 代替
```

```
# out/target/product/generic/obj/ETC/sepolicy_intermediates 字符串
```

#创建临时目录

```
mkdir -p out/target/product/generic/obj/ETC/sepolicy_intermediates/
```

#----->处理一堆输入源文件，最终输出为 *policy.conf*

#执行 *m4* 命令，用来生成 *plicy.conf* 文件。*m4* 命令将扩展 *SEAndroid* 定义的一些宏

```
m4 -D mls_num_sens=1 -D mls_num_cats=1024 -s
```

#*m4* 的输入文件。下面标黑体的是 *SEAndroid* 一些系统相关的文件，一般不会修改它

```
security_classes initial_sids access_vectors
```

```
global_macros mls_macros mls
```

```
policy_capabilities te_macros attributes
```

#*Android* 系统中的 *te* 文件。

```
adbd.te app.te bluetoothd.te bluetooth.te clatd.te dbusd.te debuggerd.te
```

```
device.te dhcp.te dnsmasq.te domain.te drmserver.te file.te gpsd.te hci_attach.te
```

```
healthd.te hostapd.te init_shell.te init.te installd.te isolated_app.te kernel.te
```

```
keystore.te media_app.te mediaserver.te mtp.te netd.te net.te nfc.te ping.te
```

```
platform_app.te ppp.te property.te qemud.te racoon.te radio.te release_app.te
```

```
rild.te runas.te sdcardd.te servicemanager.te shared_app.te shell.te
```

```
surfaceflinger.te  su.te  system.te  tee.te  ueventd.te  unconfined.te
```

```
untrusted_app.te vold.te watchdogd.te wpa_supplicant.te zygote.te
```

#其他文件

```
roles  users initial_sid_contexts fs_use genfs_contexts port_contexts
```

#m4: 将上述源文件处理完后, 生成 `policy.conf`

```
> SEPOLICY_TEMP/policy.conf
```

#下面这个命令将根据 `policy.conf` 中的内容, 再生成一个 `policy.conf.dontaudit` 文件

```
sed '/dontaudit/d'
```

```
SEPOLICY_TEMP/policy.conf >
```

```
SEPOLICY_TEMP/policy.conf.dontaudit
```

```
mkdir -p SEPOLICY_TEMP/
```

#----->根据 `policy.conf` 文件, 生成二进制文件。SEAndroid 中, 它叫 `sepolicy`

#执行 `checkpolicy`, 输入是 `policy.conf`, 输出是 `sepolicy`

#-M 选项表示支持 MLS

```
checkpolicy -M -c 26 -o SEPOLICY_TEMP/sepolicy
```

```
SEPOLICY_TEMP/policy.conf
```

#执行 checkpolicy, 输入是 policy.conf.dontaudit, 输出是 sepolicy.dontaudit

```
checkpolicy -M -c 26 -o
```

```
SEPOLICY_TEMP/sepolicy.dontaudit
```

```
SEPOLICY_TEMP/policy.conf.dontaudit
```

#--->将 sepolicy 拷贝到对应目标平台的 root 目录下

```
echo "Install: out/target/product/generic/root/sepolicy"
```

```
acp -fp SEPOLICY_TEMP/sepolicy
```

```
out/target/product/generic/root/sepolicy
```

#---->生成 file\_context 文件

#用 FILE\_CONTEXT\_TEMP 代替

# out/target/product/generic/obj/ETC/file\_contexts\_intermediates 字符串

```
mkdir -p FILE_CONTEXT_TEMP/
```

```
m4 -s external/sepolicy/file_contexts > FILE_CONTEXT_TEMP/file_contexts
```

```
checkfc SEPOLICY_TEMP/sepolicy
```

```
FILE_CONTEXT_TEMP/file_contexts
```

```
echo "Install: out/target/product/generic/root/file_contexts"
```

```
acp -fp FILE_CONTEXT_TEMP/file_contexts
```

```
out/target/product/generic/root/file_contexts
```

#--->生成 **seapp\_context** 文件，这个是 **Android** 平台特有的，其作用我们下文再介绍

#用 **SEAPP\_CONTEXT\_TEMP** 代替

```
# out/target/product/generic/obj/ETC/seapp_contexts_intermediates
```

```
mkdir -p SEAPP_CONTEXT_TEMP/
```

```
checkseapp -p SEPOLICY_TEMP /sepolicy
```

```
-o SEAPP_CONTEXT_TEMP/seapp_contexts
```

```
SEAPP_CONTEXT_TEMP/seapp_contexts.tmp
```

```
echo "Install: out/target/product/generic/root/seapp_contexts"
```

```
acp -fp SEAPP_CONTEXT_TEMP/seapp_contexts
```

```
out/target/product/generic/root/seapp_contexts
```

#---->和 *Android* 平台中的属性相关。*SEAndroid* 中，设置属性也需要相关权限

#用 *PROPERTY\_CONTEXT\_TMP* 代替：

```
# out/target/product/generic/obj/ETC/property_contexts_intermediat  
es
```

```
mkdir -p PROPERTY_CONTEXT_TMP/
```

```
m4 -s external/sepolicy/property_contexts >
```

```
PROPERTY_CONTEXT_TMP/property_contexts
```

```
checkfc -p TARGET_SEPOLICY_TEMP/sepolicy
```

```
PROPERTY_CONTEXT_TMP/property_contexts
```

```
echo "Install: out/target/product/generic/root/property_contexts"
```

```
acp -fp PROPERTY_CONTEXT_TMP/property_contexts
```

```
out/target/product/generic/root/property_contexts
```

上面展示了 `sepolicy` 编译的执行情况，读者最好自己尝试一下。注意，`checkfc`，`checkseapp` 等都是 SEAndroid 编译时使用的工具，它们用来做策略检查，看看是否有规则不符合的地方。

总结：

- `sepolicy` 的重头工作是编译 `sepolicy` 安全策略文件。这个文件来源于众多的 `te` 文件，初始化相关的文件（`initial_sid`，`initial_sid_context`，`users`，`roles`，`fs_context` 等）。
- `file_context`：该文件记载了不同目录的初始化 `SContext`，所以它和死货打标签有关。
- `seapp_context`：和 Android 中的应用程序打标签有关。
- `property_contexts`：和 Android 系统中的属性服务（`property_service`）有关，它为各种不同的属性打标签。

下面我们来看看和 SEAndroid 相关的代码，故事从 `init` 开始。

## 2. `init` 的 SEAndroid 定制

Android 平台中，SEAndroid 的初始化由进程的祖先 `init` 的 `main` 函数完成，相关代码如下所示：

[-->`init.c:main`]

```
process_kernel_cmdline();
```

```
//向 SELinux 设置两个回调函数，主要是打印 log
```

```
union selinux_callback cb;
```

```
cb.func_log = klog_write;
```

```
selinux_set_callback(SELINUX_CB_LOG, cb);
```

```
cb.func_audit = audit_callback;
```

```
//selinux_set_callback 由 libselinux 提供。读者可 google libselinux 各个 API
```

```
//的作用
```

```
selinux_set_callback(SELINUX_CB_AUDIT, cb);
```

```
//①初始化 SEAndroid
```

```
selinux_initialize();
```

```
//②给下面几个目录打标签！
```

```
restorecon("/dev");
```

```
restorecon("/dev/socket");
```

```
restorecon("/dev/__properties__");
```

```
restorecon_recursive("/sys");
```

上述代码中的两个重要函数：



- `selinux_initialize`: 初始化 SEAndroid:
- 一堆的 `restorercon`，全称应该是 `restore context`: 就是根据 `file_contexts` 中的内容给一些目录打标签。

先来看 `selinux_initialize`:

## 2.1 `selinux_initialize` 分析

[-->init.c:: `selinux_initialize`]

```
static void selinux_initialize(void)
```

```
{
```

```
/*判断 selinux 功能是否启用。方法是:
```

```
1) /sys/fs/selinux 是否存在。或者
```

```
2) ro.boot.selinux 属性不为 disabled
```

```
*/
```

```
if (selinux_is_disabled()) return;
```

```
//加载 sepolicy 文件
```

```
if (selinux_android_load_policy() < 0) {...}
```

```
selinux_init_all_handles();
```

```
/*selinux 有两种工作模式，
```

“*permissive*”：所有操作都被允许（即没有 *MAC*），但是如果有违反权限的话，会记录日志

“*enforcing*”：所有操作都会进行权限检查

```
*/
```

```
bool is_enforcing = selinux_is_enforcing();
```

```
//设置 SELinux 的模式
```

```
security_setenforce(is_enforcing);
```

```
}
```

来看上述代码中的两个函数：

- **selinux\_android\_load\_policy**：加载 sepolicy 文件。
- **selinux\_init\_all\_handles**：初始化 file\_context，seapp\_context 及 property\_context 相关内容。

### (1) *selinux\_android\_load\_policy*

来看 selinux\_android\_load\_policy，其代码如下所示：

```
[-->external/libselinux/src/android.c:: selinux_android_load_policy]
```

```

int selinux_android_load_policy(void)

{

    char *mnt = SELINUXMNT; // 值为/sys/fs/selinux

    int rc; // 挂载/sys/fs/selinux, SELINUXFS 值为"selinuxfs"

    rc = mount(SELINUXFS, mnt, SELINUXFS, 0, NULL);

    .....

    // /sys/fs/selinux 为 userpace 和 kernel 中的 SELinux 模块交互的通道

    set_selinuxmnt(mnt); // 此函数定义在 selinux.h 中, 属于 libselinux API.

    return selinux_android_reload_policy(); // 加载 SEAndroid 中的 policy 文件

}

```

图 11 展示了 Nexus 7 上 /sys/fs/selinux 的内容:

```

root@flo:/sys/fs/selinux # ls -l
-rw-rw-rw- root    root    0 1970-01-01 07:00 access
dr-xr-xr-x root    root    1970-01-01 07:00 avc
dr-xr-xr-x root    root    1970-01-01 07:00 booleans
-rw-r--r-- root    root    0 1970-01-01 07:00 checkreqprot
dr-xr-xr-x root    root    1970-01-01 07:00 class
--w----- root    root    0 1970-01-01 07:00 commit_pending bools
-rw-rw-rw- root    root    0 1970-01-01 07:00 context
-rw-rw-rw- root    root    0 1970-01-01 07:00 create
-r--r--r-- root    root    0 1970-01-01 07:00 deny_unknown
--w----- root    root    0 1970-01-01 07:00 disable
-rw-r--r-- system system 0 1970-01-01 07:00 enforce
dr-xr-xr-x root    root    1970-01-01 07:00 initial_contexts
-rw----- system system 0 1970-01-01 07:00 load
-rw-rw-rw- root    root    0 1970-01-01 07:00 member
-r--r--r-- root    root    0 1970-01-01 07:00 mls
crw-rw-rw- root    root    1, 3 1970-01-01 07:00 null
-r----- root    root    0 1970-01-01 07:00 policy
dr-xr-xr-x root    root    1970-01-01 07:00 policy_capabilities
-r--r--r-- root    root    0 1970-01-01 07:00 policyvers
-r--r--r-- root    root    0 1970-01-01 07:00 reject_unknown
-rw-rw-rw- root    root    0 1970-01-01 07:00 relabel
-r--r--r-- root    root    0 1970-01-01 07:00 status
-rw-rw-rw- root    root    0 1970-01-01 07:00 user
root@flo:/sys/fs/selinux # cat booleans
/system/bin/sh: cat: booleans: Is a directory
1|root@flo:/sys/fs/selinux # ls booleans/
in_qemu
root@flo:/sys/fs/selinux # cat booleans/in_qemu
0 0root@flo:/sys/fs/selinux # echo "1" >booleans/in_qemu
root@flo:/sys/fs/selinux # cat booleans/in_qemu
0 1root@flo:/sys/fs/selinux # echo "1" > commit_pending_bools
root@flo:/sys/fs/selinux # cat booleans/in_qemu
1 1root@flo:/sys/fs/selinux #

```

图 11 /sys/fs/selinux 的内容

用户空间进程可同读写/sys/fs/selinux 的各个文件或其中的子目录来通知 Kernel 中的 SELinux 完成相关的操作。

我们此处此处举一个例子，如图 11 下方红框中的 booleans 文件夹：

- 我们可以 SELinux 的安全配置文件中写一些类似 if/else 的语句。  
if 中的是布尔判断条件。比如 booleans 文件夹下有一个 in\_qemu 文件，这个就是 sepolicy 安全配置文件中的一个布尔变量。**in\_qemu** 定义在 **domain.te** 文件中，关键词是 **bool**。

- **cat booleans/in\_qemu:** 打印 in\_qemu 布尔变量的取值。读者会发现它的值为“0 0”。为什么有两个零呢，这第一个 0 是它的当前值，第二个零代表 pending 取值。即还没有赋值给当前值的一个中间变量。如果我们通过 `echo "1" > booleans/in_qemu` 的话，第二个零将变成 1。
- 为什么需要有中间变量呢？读者注意图 11 的右上方有一个 `commit_pending_bools` 文件。原来，通过在布尔变量中设置一个 pending 变量，我们可以实现批处理操作。即先修改 1 个或多个布尔变量的 pending 变量，然后往 `commit_pending_bools` 写 1，这样这些 1 个或多个的布尔变量将使用 pending 变量取代当前值。

接下来看看 `selinux_android_reload_policy` 函数：

```
[-->external/libselinux/src/android.c:: selinux_android_reload_policy]
```

```
int selinux_android_reload_policy(void)
```

```
{
```

```
    int fd = -1, rc; struct stat sb; void *map = NULL;
```

```
    int i = 0;
```

```
    // sepolicy_file 指明 sepolicy 文件的路径。Android 中有两处，第一个是
```

```
    // /data/security/current/sepolicy。第二个是 root 目录下的 sepolicy 文件。
```

//下面这段逻辑可知，SEAndroid 只使用其中的一个，如果/data/目录下有 *sepolicy* 文件，则

//优先使用它

```
while (fd < 0 && sepolicy_file[i]) {
```

```
    fd = open(sepolicy_file[i], O_RDONLY | O_NOFOLLOW);
```

```
    i++;
```

```
}
```

```
.....
```

```
map = mmap(NULL, sb.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
```

```
.....
```

//假设使用根目录下的 *sepolicy* 文件。下面这个函数由 *selinux.h* 定义，它将此文件加载到

//内核中

```
rc = security_load_policy(map, sb.st_size);
```

```
.....
```

```
munmap(map, sb.st_size);
```

```
close(fd);
```

```
return 0;
```

```
}
```

init 通过 mmap 的方式，将 sepolicy 文件传递给了 kernel。init 使用了 libselinux 提供的 API 函数来完成相关操作。而 libselinux 则是通过操作 /sys/fs/selinux 下的文件来完成和 Kernel 中 SELinux 模块的交互。libselinux 库的 API 不是我们研究的重点，感兴趣的兄弟请自己研究源码。

总之，selinux\_android\_load\_policy 干得最重要的一件事情就是将 sepolicy 文件传递给 Kernel，这样 Kernel 就有了安全策略配置文件，后续的 MAC 才能开展起来。

在此，请读者注意 sepolicy 文件的位置：

- 首先查看 /data/security/current/sepolicy：笔者自定义的策略文件一般放在这里，init 优先使用它。
- /sepolicy：root 根目录下的 sepolicy，如果 data 目录下没有 sepolicy，则使用它。系统默认的 sepolicy 在此。

## (2) selinux\_init\_all\_handles

前面讲过，init 要给一些死货和 property 打标签，为了完成这个工作，根据 libselinux 的 API，init 需要先创建两个 handler，代码在 selinux\_init\_all\_handles 中：

[-->init.c:: selinux\_init\_all\_handles]

```
void selinux_init_all_handles(void)
```

```
{
```

```
    sehandle = selinux_android_file_context_handle();
```

```
    sehandle_prop = selinux_android_prop_context_handle();
```

```
}
```

创建两个 handler，主要为后续做 labeling 控制。我们来看看 prop 的 context:

[-->init.c::selinux\_android\_prop\_context\_handle]

```
struct selabel_handle* selinux_android_prop_context_handle(void)
```

```
{
```

```
    int i = 0;
```

```
    struct selabel_handle* sehandle = NULL;
```

```
    //setopts_prop 也有两个值:
```

```
    //第一个是/data/security/property_contexts。第二个是/property_contexts
```

```
    while ((sehandle == NULL) && seopts_prop[i].value) {
```



```

        sehandle = selabel_open(SELABEL_CTX_ANDROID_PROP,
&seopts_prop[i], 1);

        i++;

    }

    //假设采用的是根目录下的 property_context 文件

    .....

    return sehandle;

}

```

handler 其实就是为了打开 xxx\_context 文件。由于它们和 restorecon 有关，下面直接来看 restorecon 函数，看看这些 handler 是怎么个用法。

## 2.2 restorecon 分析

[-->init.c::restorecon]

```

int restorecon(const char *pathname)

{

    char *secontext = NULL;

    struct stat sb;

    int i;

```

```
if (is_selinux_enabled() <= 0 || !sehandle)
```

```
return 0;
```

```
if (lstat(pathname, &sb) < 0) return -errno;
```

```
//查找 file_context 文件中是否包含有 pathname 路径的控制选项
```

```
if (selabel_lookup(sehandle, &secontext, pathname, sb.st_mode) < 0)
```

```
return -errno;
```

```
//设置 pathname 目录的 security_context, lsetfilecon 的实现非常简单，就是调用
```

```
//
```

```
if (lsetfilecon(pathname, secontext) < 0) {
```

```
    freecon(secontext);
```

```
    return -errno;
```

```
}
```

```
freecon(secontext);
```

```
return 0;
```

```
}
```

想知道 selinux 是如何 labeling 一个文件或目录的吗？答案在 lsetfilecon 中：

```
[-->external/libselinux/src/lsetfilecon.c:: lsetfilecon]
```

```
int lsetfilecon(const char *path, const security_context_t context)
```

```
{
```

```
    //设置文件系统的属性
```

```
    return lsetxattr(path, XATTR_NAME_SELINUX, context, strlen(context) +
```

```
    1,0);
```

```
}
```

## 2.3 property 权限检查

一般而言，SELinux 权限检查都是由 kernel 来完成的，不过对于 Android 平台中的 Property 而言，这却完全是一个用户空间的内容。所以，我们看看 init 是如何使用 libselinux 来完成用户空间的权限检查的。

每当其他进程通过 setprop 函数设置属性时，property\_service 中有一个叫 check\_

```
[system/core/init/property_service.c:: check_mac_perms]
```

```
static int check_mac_perms(const char *name, char *sctx)
```

```
{
```

```
if (is_selinux_enabled() <= 0) return 1;
```

```
char *tctx = NULL;
```

```
const char *class = "property_service";
```

```
const char *perm = "set";
```

```
int result = 0;
```

```
.....
```

```
//检查 property_context 中是否定义了目标 SContext，即 tctx。
```

```
if (selabel_lookup(sehandle_prop, &tctx, name, 1) != 0) goto err;
```

//将源 SContext 和目标 SContext 进行比较，判断是否有相关权限。*name* 是属性的名字

// 源 SContext 是调用 setprop 进程的 SContext。目标 SContext 是 property\_context

```
//文件中定义的 SContext。
```

```
if (selinux_check_access(sctx, tctx, class, perm, name) == 0)
```

```
result = 1;
```

```
    freecon(tctx);

err:

    return result;

}
```

怎么样？理解起来并不困难吧？用户空间的权限检查主要就是通过 `selinux_check_access` 完成，其输入参数包括：

- 源的 SContext：它就是调用 `setprop` 的进程的 SContext
- 目标的 SContext：不同的属性有不同的 SContext，这是在 `property_context` 中定义的。
- 要检查的 Object class（系统所支持的类在 `external/sepolicy/security_classes` 文件中定义）。
- 操作名称（`perm`，由 access vector 定义。对 Property 这种 Object class 而言，其唯一需要做权限检查的操作就是 `set`。读者可参考 `external/sepolicy/access_vectors` 这个文件）。

具体的哪一个属性（`name` 参数指定，就是具体指明哪一文件）。

提示：关于这些 API 的说明，读者请参考 [http://selinuxproject.org/page/User\\_Resources](http://selinuxproject.org/page/User_Resources) 中的 Manual pages 文档。

下面我们来看 Android 中应用程序是如何使用 SELinux 的。

### 3. 应用程序中的 SELinux

对应用程序而言，最重要的工作就是管理它们的 DT 和 TT：

- 所有 Android 的 Application 对应的进程都是从 `zygote` 进程中 `fork` 出来的。从前文介绍 DT 的知识可知，在做 DT 时，可以根据所执行的不同 Type 的文件来转换到不同的 DT。但这个对 Android 而言不可行。因为 `zygote` 在 `fork` 子进程后，并没有执行 `execv`。
- `apk` 在安装后，都会在 `/data/data/` 目录下建立自己对于的文件夹，这个工作是由 `installd` 来完成的。同样，`installd` 应该给这些不同的文件夹打上对应的 `label`。

我们先来看应用程序的 DT。

#### 3.1 Java 应用程序的 DT

Android 中应用进程（就是 APK 所在的进程）的 DT 转换其实很简单，它及其具有 Android 特色：

- 普通的 DT 是根据所 `execv` 文件的 Type 来设置 DT 转换条件。
- 而 Android 中则根据该 APK 签名信息来讲最终的进程转换到几种预设值的 Domain 中。

##### (1) `mac_permissions.xml` 的用途

我们先来看 `PackageManagerService`：

```
[-->PackageManagerService.java::PackageManageService]
```

.....

/\*下面这个函数将尝试解析

1)/data/security/mac\_permissions.xml 或

2)/system/etc/security/mac\_permissions.xml 中的内容。

\*/

*mFoundPolicyFile = SELinuxMMAC.readInstallPolicy();*

注意，mac\_permissions.xml 位于 external/sepolicy 中，图 12 所示为该文件的原始：

```
<?xml version="1.0" encoding="utf-8"?>
<policy>

  <!-- Platform dev key in AOSP -->
  <signer signature="@PLATFORM" >
    <seinfo value="platform" />
  </signer>

  <!-- Media dev key in AOSP -->
  <signer signature="@MEDIA" >
    <seinfo value="media" />
  </signer>

  <!-- shared dev key in AOSP -->
  <signer signature="@SHARED" >
    <seinfo value="shared" />
  </signer>

  <!-- release dev key in AOSP -->
  <signer signature="@RELEASE" >
    <seinfo value="release" />
  </signer>

  <!-- All other keys -->
  <default>
    <seinfo value="default" />
  </default>

</policy>
```

图 12 mac\_permissions.xml 的内容

在系统过程中，图 12 中的@RELEASE，@PLATFORM 等内容会被 RELEASE、PLATFORM 签名信息替换。图 13 所示为 Nexus 7 中该文件的内容。

```
130|root@flo:/ # cat /system/etc/security/mac_permissions.xml
<?xml version="1.0" encoding="iso-8859-1"?><!-- AUTOGENERATED FILE DO NOT MODIFY --><policy><signer signature="3082035d308202470206013f34416fa1300b06092a864886f70d0101053074310b3009060355040613025553311330110603550408130a43616c69666f726e6961311630140603550407130d4d6f756e7461696e205669657731143012060355040a130b476f6f676c6520496e632e3110300e060355040b1307416e64726f69643110300e06035504031307416e64726f6964301e170d3133303631313137323131315a170d3330303131393033313430375a3074310b3009060355040613025553311330110603550408130a43616c69666f726e6961311630140603550407130d4d6f756e7461696e205669657731143012060355040a130b476f6f676c6520496e632e3110300e060355040b1307416e64726f69643110300e06035504031307416e64726f696430820122300d06092a864886f70d010105000382010f003082010a0282010100cb2495d82b70c510d593e37f3e9e3abb7f3f1f188c64c4b068cc4be6591cfc69f73e6167d05c62320d85a4ea719783383712f039aa5f33d2054ac790227b578ba73110377e233f899ea7af70219c85fa232b125e6fd8cfab5e0f8c814665aldf297b4191af4aa78b47ea44e8283d9d559d99544d9b372f18ef0e1536830ac5811ec47a97214416ec71a7cec0c9934cfefa57662f85100f0ad07d610cdd45f196d174a3ef477f58e1eeb72ab2a76b4383d334a5851120daf5d76c1215f0d9eb8d054dddec77aaf64855270ff28c8e8e29a6c84367a858d400fa83ff71459eb8e3f354c72a476b213ad170b0eff666ad13f47a0e05ca04804833504f1802162b990203010001300b06092a864886f70d01010503820101007a401e677fa4aa35bed5cb5491577c9b9f14742b994d5b9afc87e62d59d9785c2779e0308b130f600b2349c4b18d415cb9822d78c32dac343f25fd28e45e414b50138b0713e0e6997a8ed560d5f7bb22144321aa8489cb9c9577f286d518d4f3cbf953586cc7de2d220f80a6a9a1dcd620571a7359ac19b6f7b1c153912093192636db4bb21368fbde0c626d3c7d23b7f02edd3b5e6c272fc368b5cc88302b8e2ce2f701c0b7e1647d574b7a9fc0b334a0600507cefcfa03fa4a4b2bca3f2a5a81f460df5ad66986d082dc3d2f5b36fe4a8c799fb8e3b1fc3df28924fe4a5c8aa538b07a908f50cff5eebca01001bc3c9ba85e33c5c51d03a640409aa81c0d"><seinfo value="platform"/></signer><signer signature="3082035d308202470206013f3441d234300b06092a864886f70d0101053074310b3009060355040613025553311330110603550408130a43616c69666f726e6961311630140603550407130d4d6f756e7461696e205669657731143012060355040a130b476f6f676c6520496e632e3110300e060355040b1307416e64726f69643110300e06035504031307416e64726f6964301e170d3133303631313137323133365a170d3338303131393033313430375a3074310b3009060355040613025553311330110603550408130a43616c69666f726e6961311630140603550407130d4d6f756e7461696e205669657731143012060355040a130b476f6f676c6520496e632e3110300e060355040b1307416e64726f69643110300e06035504031307416e64726f696430820122300d06092a864886f70d010105000382010f003082010a0282010100b10a921b9836515f06369feddb94eb7e5d0451a8a5dae20555c6aa37a10e043b2aee6a203201f53eea2a4dfc95fac32931e5ee026cf17d7fc3e1e1a85791b2709c009422441b518380c6aeadd29ecfeb7098e80f04857ca1bad28d3alc7ee866c42cb9acaa4bce3bbe651363d7ea32a3961e2705ffa4c24ea9058906910f0f42543a7d80146075366cadf99e29df4b0765131b6a75f84d06e663d0eb379bf3cf13d55366d7a5b27173326faa6ba42b45b70b30ea54ac21116dfc22967e6305c1064ef5dae580a1600000938d69ac84736a8d3ed298c3e06df42aa2824aefad84358cd02ee04a45d274e87c785b83c35d2945266fd99a9cdf7e916558d43eeffc290203010001300b06092a864886f70d01010503820101005851e96c6257b518ddc48c7f2c204d1e3cc53aeaac2ad66876cc475797556135be33f37133f73cb65eb43bc814e3c13ec2808fe980dfa90701cf5d8bebc036617ba4c9765d82e6933ce1ab91ca015c19b634617c986a88971bda0db537383a4108ffe603d598355d1399e87ca9da3a5ff03ac08caa0968701c2f19294c10fc100ab9ab39b227db184327d26096a38484a275abad6db23a2e8955063e0a9fcb37ce1048ca8c6f59033bde230186b47ea59d8b3031f98e19942cf64cd11ea5683ef92ed518faafbe7bd887bab1d2fad14fe949ddd4e2cb284a768c07a04ddc6c53ce3286fec5350a0cd408205e8f397b5142d95d8a41ee0226e66674fa461311f"><seinfo value="media"/></signer><signer signature="308203
```

图 13 Nexus7 中 mac\_permissions.xml 内容

mac\_permissions.xml 保存了不同签名所对应的 seinfo：如 seinfo 为 platform 时的签名是什么，seinfo 为 media 的时候签名又是什么。那么，这些信息有啥用呢？来看下文。

## （2）扫描 APK

当 APK 安装时，也就是 APK 被 PKMGS 扫描的时候，有如下的代码：



[-->PackageManagerService.java::ScanPackageLI]

```
if (mFoundPolicyFile) {
```

```
    //下面这个函数将根据签名信息赋值 seinfo 值给对应的 apk
```

```
    SELinuxMMAC.assignSeinfoValue(pkg);
```

```
}
```

[-->SELinuxMMAC.java::assignSeinfoValue]

```
public static void assignSeinfoValue(PackageParser.Package pkg) {
```

```
    //对于系统 app(预装的, 位于 system 目录下的)
```

```
    if (((pkg.applicationInfo.flags & ApplicationInfo.FLAG_SYSTEM) != 0) ||
```

```
        ((pkg.applicationInfo.flags &
```

```
            ApplicationInfo.FLAG_UPDATED_SYSTEM_APP) != 0)) {
```

```
        for (Signature s : pkg.mSignatures) {
```

```
            if (s == null) continue;
```

```
            //sSigSeinfo 存储了 mac_permissions.xml 中 seinfo 标签的内容
```

```
            if (sSigSeinfo.containsKey(s)) {
```

```
String seinfo = pkg.applicationInfo.seinfo = sSigSeinfo.get(s);
```

```
return;
```

```
}
```

```
}
```

```
//sPackageSeinfo 存储了 xml 中 package 标签下 seinfo 子标签的内容
```

```
if (sPackageSeinfo.containsKey(pkg.packageName)) {
```

```
String seinfo = pkg.applicationInfo.seinfo =
```

```
sPackageSeinfo.get(pkg.packageName);
```

```
return;
```

```
}
```

```
}
```

```
//default 标签中 seinfo 的值
```

```
String seinfo = pkg.applicationInfo.seinfo = sSigSeinfo.get(null);
```

```
.....
```

```
}
```

assignSeinfoValue 的功能如上代码所示，它根据 apk 的签名信息来赋值不同的 seinfo，也就是诸如"platform"，"media"之类的值。

提示：大家能想出为什么要设置 seinfo 吗？恩，它就是 Android 为 App 定义的 SContext 中的 Domain 的值。

### (3) App 的 DT 转换

ActivityManagerService 负责启动目标应用进程，相关代码如下所示：

[-->ActivityManagerService.java:: startProcessLocked]

```
Process.ProcessStartResult startResult =
```

```
    Process.start("android.app.ActivityThread",
```

```
        app.processName, uid, uid, gids, debugFlags, mountExternal,
```

```
        app.info.targetSdkVersion, app.info.seinfo, null);
```

根据《深入理解 Android 卷 I》第 4 章对 zygote 的介绍，zygote 进程将 fork 一个子进程，相关函数在：

[-->ZygoteConnection.java::runOnce]

```
pid    =    Zygote.forkAndSpecialize(parsedArgs.uid,    parsedArgs.gid,
    parsedArgs.gids,
```

```
    parsedArgs.debugFlags, rlimits, parsedArgs.mountExternal,
```

```
    parsedArgs.seInfo,parsedArgs.niceName);
```

该函数由 JNI 实现，代码在 `dalvik/vm/native/ dalvik_system_Zygote.cpp` 中，其中最重要的是内部所调用的 `forkAndSpecializeCommon`:

[-->`dalvik_system_Zygote.cpp:: forkAndSpecializeCommon`]

```
pid = fork();
```

```
if (pid == 0) {
```

```
.....
```

```
err = setSELinuxContext(uid, isSystemServer, selInfo, niceName);
```

```
.....}
```

[-->`external/libselinux/android.c::selinux_android_setcontext`]

```
int selinux_android_setcontext(uid_t uid,int isSystemServer,
```

```
const char *seinfo,const char *pkgname)
```

```
{
```

```
char *orig_ctx_str = NULL, *ctx_str; context_t ctx = NULL;
```

```
int rc = -1;
```

```
if (is_selinux_enabled() <= 0) return 0;
```

```
// 重要函数： seapp_context_init ， 内部将调用  
selinux_android_seapp_context_reload
```

```
//以加载 seapp_contexts 文件。
```

```
// 1) /data/security/current/seapp_contexts 或者
```

```
// 2) /seapp_contexts 本例而言，就是根目录下的这个 seapp_context 文件
```

```
__selinux_once(once, seapp_context_init);
```

```
rc = getcon(&ctx_str);
```

```
ctx = context_new(ctx_str);
```

```
orig_ctx_str = ctx_str;
```

```
//从 zygote 进程 fork 出来后，最初的 SContext 取值为 u:r:zygote:s0
```

```
//下面这个函数将根据 uid, pkgname 等设置最终的 SC。例如 u:r:system_app:s0 等
```

```
rc = seapp_context_lookup(SEAPP_DOMAIN, uid, isSystemServer, seinfo,  
pkgname,
```

```
ctx);
```

```
ctx_str = context_str(ctx);
```

```
rc = security_check_context(ctx_str);
```

```

if (strcmp(ctx_str, orig_ctx_str)) {

    rc = setcon(ctx_str);

}

rc = 0;

.....

return rc;

}

```

图 14 所示为 seapp\_context 的内容，非常简单：

```

isSystemServer=true.domain=system
user=system.domain=system_app.type=system_data_file
user=bluetooth.domain=bluetooth.type=bluetooth_data_file
user=nfc.domain=nfc.type=nfc_data_file
user=radio.domain=radio.type=radio_data_file
user=_app.domain=untrusted_app.type=app_data_file.levelFrom=none
user=_app.seinfo=platform.domain=platform_app.type=platform_app_data_file
user=_app.seinfo=shared.domain=shared_app.type=platform_app_data_file
user=_app.seinfo=media.domain=media_app.type=platform_app_data_file
user=_app.seinfo=release.domain=release_app.type=platform_app_data_file
user=_isolated.domain=isolated_app
user=shell.domain=shell.type=shell_data_file

```

图 14 seapp\_context 内容

上面代码中的 seapp\_context\_lookup 将根据图 14 的内容，通过不同的 apk 所对应的 seinfo，找到他们的目标 domain，然后再设置为它们新的 SContext。例如图 15 的 Nexus 7 中 ps -Z 的结果图。

u:r:shared_app:s0	u0_a13	925	179	com.android.launcher
u:r:media_app:s0	u0_a6	1019	179	android.process.media
u:r:untrusted_app:s0	u0_a41	1208	179	com.google.android.apps.currents
u:r:kernel:s0	root	1651	2	msm_sat0
u:r:init:s0	root	1825	227	daemonsu:10040
u:r:untrusted_app:s0	u0_a8	3012	179	com.google.process.gapps
u:r:untrusted_app:s0	u0_a8	3211	179	com.google.android.gms
u:r:untrusted_app:s0	u0_a8	3233	179	com.google.process.location
u:r:untrusted_app:s0	u0_a60	10460	179	com.google.android.apps.magazines
u:r:untrusted_app:s0	u0_a21	12115	179	com.google.android.googlequicksearchbox:se
u:r:untrusted_app:s0	u0_a51	15290	179	com.google.android.inputmethod.pinyin
u:r:untrusted_app:s0	u0_a68	16212	179	com.google.android.apps.plus
u:r:untrusted_app:s0	u0_a43	16407	179	com.google.android.apps.docs
u:r:untrusted_app:s0	u0_a42	21930	179	com.google.android.deskclock

图 15 ps -Z 查看 apk 进程的 SContext

**seapp\_context\_lookup** 是完成从 seapp\_context 文件内容映射到具体对应为哪个 Domain 的关键函数，该函数第一次看起来吓死人，其实蛮简单。这里就不再多说。

anyway，SEAndroid 中，不同应用程序将根据它们的签名信息得到对应的 SContext（主要是 Domain，MLS 其实没用上，但以后可以用上，这是通过图 14 中的 levelFrom 语句来控制的，具体可参考 seapp\_context\_lookup 的实现）。

DT 完成后，我们看系统如何为它们的对应文件夹打标签

### 3.2 App data 目录的 TT

还是在 PackageManagerService 的 scanPackageLI 函数中，

```
[-->PackageManagerService.java:: scanPackageLI]
```

```
int ret = createDataDirsLI(pkgName, pkg.applicationInfo.uid,
```

```
pkg.applicationInfo.seinfo);
```

createDataDirsLI 最终会调用 installd 实现的函数：

```
[-->installd/commands.c::install]
```

```
//内部调用 selinux_android_setfilecon2, 它和上文的 selinux_android_setcontext
```

```
//几乎一样。最终它将设置 pkgdir 的 SContext。注意，它主要根据 seapp_context 文件中的
```

```
//type 字段来确定最终的 Type 值。
```

```
if (selinux_android_setfilecon2(pkgdir, pkgname, seinfo, uid) < 0) {
```

```
.....
```

```
}
```

图 16 展示了 `ls -Z /data/data` 目录下的结果。

```
drwxr-x--x u0_a63 u0_a63 u:object_r:platform_app_data_file:s0 com.android.noisefield
drwxr-x--x u0_a64 u0_a64 u:object_r:platform_app_data_file:s0 com.android.packageinstallle
r
drwxr-x--x u0_a35 u0_a35 u:object_r:platform_app_data_file:s0 com.android.pacprocessor
drwxr-x--x u0_a66 u0_a66 u:object_r:platform_app_data_file:s0 com.android.phasebeam
drwxr-x--x radio radio u:object_r:radio_data_file:s0 com.android.phone
drwxr-x--x u0_a38 u0_a38 u:object_r:platform_app_data_file:s0 com.android.printspooler
drwxr-x--x u0_a1 u0_a1 u:object_r:platform_app_data_file:s0 com.android.providers.calen
dar
drwxr-x--x u0_a3 u0_a3 u:object_r:platform_app_data_file:s0 com.android.providers.conta
cts
```

图 16 /data/data 目录下 `ls -Z` 的结果

是不是和图 14 中 `seapp_context` 文件的 `type` 字段描述一样一样的？

## 4 小试牛刀



下面，笔者将通过修改 shell 的权限，使其无法设置属性。

先来看 shell 的 te，如下所示：

```
[external/sepolicy/shell.te]
```

```
# Domain for shell processes spawned by ADB
```

```
type shell, domain;
```

```
type shell_exec, file_type;
```

```
#shell 属于 unconfined_domain, unconfined 即是不受限制的意思
```

```
unconfined_domain(shell)
```

```
# Run app_process.
```

```
# XXX Split into its own domain?
```

```
app_domain(shell)
```

unconfined\_domain 是一个宏，它将 shell 和如下两个 attribute 相关联：

```
[external/sepolicy/te_macros]
```

```
#####
```

```
# unconfined_domain(domain)
```

```
# Allow the specified domain to do anything.
```

```
#
```

```
define(`unconfined_domain',`
```

```
typeattribute $1 mltrustedsubject; #这个和 MLS 有关
```

```
typeattribute $1 unconfineddomain;
```

```
')
```

unconfineddomain 权限很多，它的 allow 语句定义在 unconfined.te 中：

```
[external/sepolicy/unconfined.te]
```

```
.....
```

```
allow unconfineddomain property_type:property_service set;
```

从上面可以看出，shell 所关联的 unconfineddomain 有权限设置属性。所以，我们把它改成：

```
allow {unconfineddomain -shell} property_type:property_service set;
```

通过一个“-”号，将 shell 的权限排除。

然后：

- 我们 mmm external/sepolicy，得到 sepolicy 文件。
- 将其 push 到/data/security/current/sepolicy 目录下

- 接着调用 `setprop selinux.reload_policy 1`，使得 init 重新加载 `sepolicy`，由于 `/data` 目录下有了 `sepolicy`，所以它将使用这个新的。

图 17 所示为整个测试的例子：

```
root@flo:/ # #before we reload policy
root@flo:/ # getprop wlan.driver.status
fail
root@flo:/ # setprop wlan.driver.status test_ok
root@flo:/ # getprop wlan.driver.status
test_ok
root@flo:/ # setprop selinux.reload_policy 1
root@flo:/ # #after we loads the new policy, shell has no perm to setprop
root@flo:/ # setprop wlan.driver.status test_fail
root@flo:/ # getprop wlan.driver.status
test_ok
root@flo:/ # setprop wlan.driver.status hello_nihao
root@flo:/ # getprop wlan.driver.status
test_ok
root@flo:/ #
```

<http://blog.csdn.net/Innost>

图 17 测试结果

根据图 17：

- 重新加载 `sepolicy` 之前，笔者可通过 "`setprop wlan.driver.status test_ok`" 设置该属性的值为 `test_ok`。注意，这个值笔者瞎设的。
- 当通过 `setprop selinux.reload_policy 1` 的命令后，init 重新加载了 `sepolicy`。
- 从此，笔者再 `setprop wlan.driver.status` 都不能修改该属性的值。

图 18 所示为 `dmesg` 输出，可以看出，当 `selinux` 使用了 `data` 目录下这个新的 `sepolicy` 后，shell 的 `setprop` 权限就被否了！

```
<4>[40253.720457] SELinux: Loaded policy from /data/security/current/sepolicy
<4>[40286.494688] avc: received policyload notice (seqno=5)
<4>[40286.495512] avc: denied { set } for property=wlan.driver.status scontext=u:r:shell:s0 tco
ncontext=u:object_r:system_prop:s0 tclass=property_service
<3>[40286.495970] init: sys prop: permission denied uid:0 /name:wlan.driver.status t/Innost
```

图 18 dmesg 输出

提示：前面曾提到过 audit，日志一类的事情。恩，这个日志由 kernel 输出，可借助诸如 audit2allow 等 host 上的工具查看哪些地方有违反权限的地方。系统会将源，目标 SContext 等信息都打印出来。

### 三 全文总结

本文对 SELinux 的核心知识进行了介绍。从入门角度来说，有了这些内容，SELinux 大概 80% 左右的知识都已经介绍，剩下的工作就是不断去修改和尝试不同的安全配置文件。

然后我们对 SEAndroid 进行了相关介绍，这部分基本上反映了 Android 是如何利用这些安全配置文件来构造自己的安全环境的。

从目前 AOSP SEAndroid 安全配置源文件来看，很多 te 文件中都使用了如下这样的语句：

```
./policy.conf:3641:permissive adbd;  
./policy.conf:4016:permissive bluetooth;  
./policy.conf:4046:permissive bluetoothd;  
./policy.conf:4110:permissive clatd;  
./policy.conf:4179:permissive dbusd;  
./policy.conf:4243:permissive debuggerd;  
./policy.conf:4376:permissive dhcp;  
./policy.conf:4448:permissive dnsmasq;  
./policy.conf:4686:permissive drmserver;  
./policy.conf:4865:permissive gpsd;  
./policy.conf:4937:permissive hci_attach;  
./policy.conf:5002:permissive healthd;  
./policy.conf:5076:permissive hostapd;  
./policy.conf:5145:permissive init;  
./policy.conf:5178:permissive init_shell;  
./policy.conf:5342:permissive isolated_app;  
./policy.conf:5380:permissive kernel;  
./policy.conf:5398:permissive keystore;  
./policy.conf:5466:permissive media_app;  
./policy.conf:5515:permissive mediaserver;  
./policy.conf:5584:permissive mtp;  
./policy.conf:5935:permissive nfc;  
./policy.conf:5964:permissive ping;  
./policy.conf:6014:permissive platform_app;  
./policy.conf:6063:permissive ppp;  
./policy.conf:6126:permissive qemuud;  
./policy.conf:6188:permissive racoon;  
./policy.conf:6201:permissive radio;  
./policy.conf:6243:permissive release_app;  
./policy.conf:6292:permissive rild;  
./policy.conf:6361:permissive runas;  
./policy.conf:6408:permissive sdcardd;  
./policy.conf:6472:permissive servicemanager;  
./policy.conf:6539:permissive shared_app;  
./policy.conf:6620:permissive su;  
./policy.conf:6669:permissive surfaceflinger;  
./policy.conf:6741:permissive system_app;  
./policy.conf:6770:permissive system;  
./policy.conf:6802:permissive tee;  
./policy.conf:6865:permissive ueventd;  
./policy.conf:6922:permissive untrusted_app;  
./policy.conf:7139:permissive watchdogd;  
./policy.conf:7151:permissive wpa;
```

图 19 permissive 定义

其中，`permissive` 关键词表示不用对上述这些 `type/domain` 进行 MAC 监管。`permissive` 一般用于测试某个策略，看是否对整个系统有影响。一旦测验通过，就可以把 `permissive` 语句移掉，以真正提升安全。

基于 SEAndroid，广大搞机人可以：

- 针对行业，开发更加安全的安全策略。
- 定制 MLS，针对企业级或军用级的多层权限管理。
- 不知道 google 有没有意向实现 Modular Policy，这样的话，SELinux 灵活性更高。

另外，要提醒读者的是，安全配置需要考虑的东西非常多，稍有不甚，就会影响系统其他模块的运行。比如笔者在研究 SELinux 时，不小心把 Ubuntu 的图像界面系统启动不了，后来只能移除 SELinux 后才解决。这也是为什么 SELinux 出来这么多年，但是大家好像碰到它的机会很少的原因，因为它的配置实在是太麻烦，很容易出错！

最后，反复提醒读者，一旦修改了策略文件，务必进行全方位，多层面测试。

关于 SEAndroid 的更多官方说明，请参考

<http://source.android.com/devices/tech/security/se-linux.html>