Selinux学习文档

1.背景知识

2.概念（示例：切换密码）

1.上下文

2. TEAC

3.domain transitions

4.roles

5.selinux的Multilevel安全模型

3.架构

1.内核层架构

a.LSM框架

b.LSM模块

2.用户层对象管理器

a.用户层对象管理器的内核支撑

b.用户层部分

4.类和权限

1.类的目的

2.定义一个类

3.常用类和常用权限

5.type enforcement

1.te的概念

2.type attribute 和aliases

3.AV规则

4.type 规则

6.角色和用户

1.SELinux的RBAC

a.概述

b.用角色管理用户权限

c.上下文里面的用户和角色

2.角色和角色声明

a.角色声明语句

b.角色allow语句

c.角色转换语句

d.角色控制语句

3.用户和用户声明

a.用户声明和关联角色

7.约束条件

1.概述

2.constraints语句

3.validatetrans语句

8.多等级安全

1.概述

2.多等级安全的上下文

a.定义安全等级

b.上下文的mls扩展

c.

3.多等级安全constrains

a.mlsconstrain语句

b.mlsvalidatetrans语句

4.其他影响

9.条件策略

1.概述

2.布尔变量

a.定义

b.管理布尔变量

3.条件语句

10．标签化

1.概述

2.文件标签化

3.网络和套接字标签化

a.网络接口标签化

b.网络节点标签化

c.网络端口标签化

d.套接字标签化

4.IPC标签化

5.其他标签形式

6.初始安全标识符

第二部分：概念

上下文:

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。

user:role:type

在手机中可以通过ps –Z命令查看。下图为JL618的ps –Z结果图

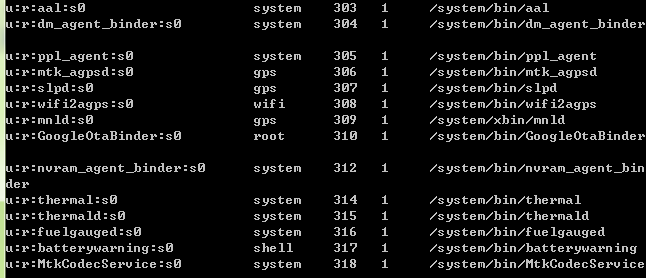


图1中最左边的那一列是进程的SContext，以第一个进程/system/bin/aal的SContext为例，其值为u:r:aal:s0，其中：

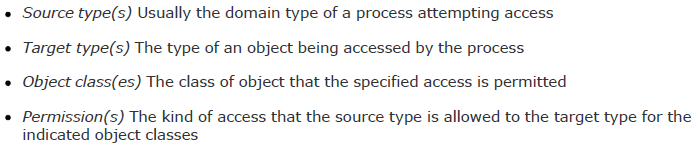
* u为user的意思。SEAndroid中定义了一个SELinux用户，值为u。
* r为role的意思。role是角色之意，它是SELinux中一种比较高层次，更方便的权限管理思路，即Role Based Access Control（基于角色的访问控制，简称为RBAC）。简单点说，一个u可以属于多个role，不同的role具有不同的权限。RBAC我们到最后再讨论。
* aal，代表该进程所属的Domain为aal。MAC的基础管理思路其实不是针对上面的RBAC，而是所谓的Type Enforcement Accesc Control（简称TEAC，一般用TE表示）。
* S0和SELinux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

Ps：对于进程来说，我们所说的”domain”,”domain type”,”subject type”,”process type”所指的都是同一样东西，就是”type”

TEAC：

在selinux中，所有的权限都是需要被准确定义的。无论是什么账户，selinux默认是没有任何权限的。也就是说，selinux是没有root用户权限的。定义权限的语法如下：

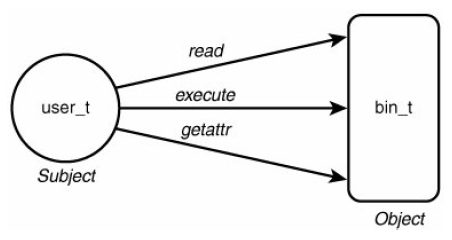
allow sourceType targetType:objectClass permissions



例子：

allow user\_t bin\_t : file {read execute getattr}

这个语句的意思就是允许user\_t的进程向bin\_t的进程进行read execute getattr的操作。如下图：



其中这些语句一般定义在.te文件中。常用object class 会定义在external/sepolicy/security\_classes

#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

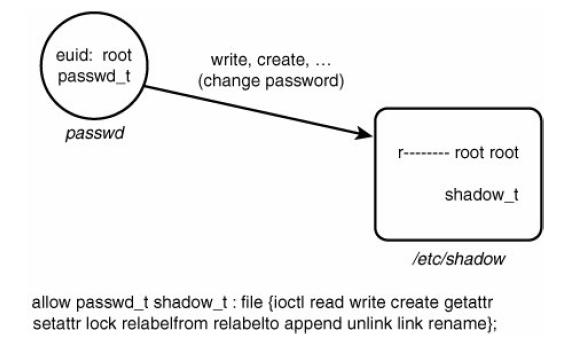
class udp\_socket

......

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

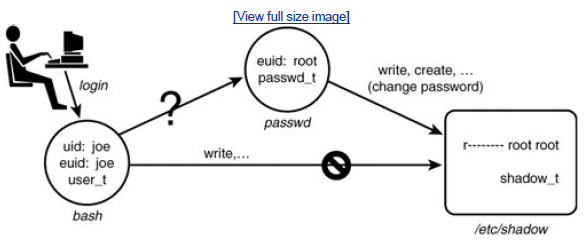
**class zygote**   #Android平台特有的zygote

另一个例子：



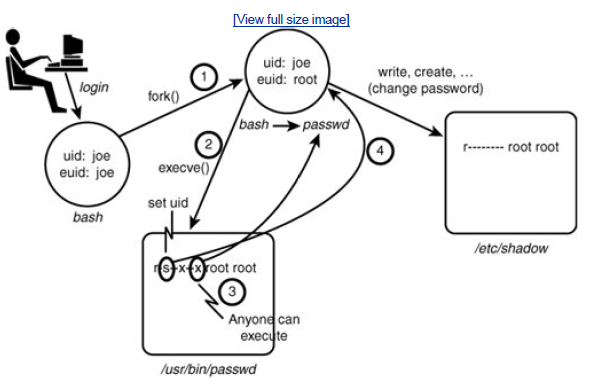
允许passwd\_t可以去更换shadow\_t的密码;

Domain Transitions:

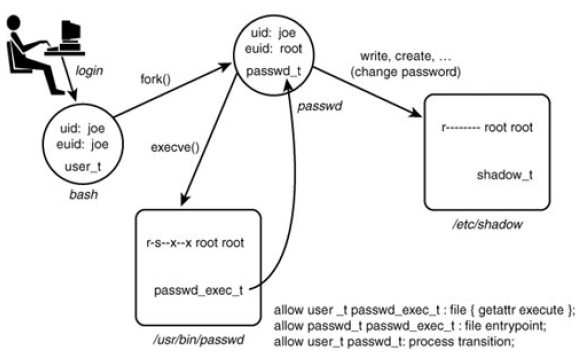


如上图所示：在joe想要去更换shadow的密码，selinux是不允许user\_t来直接更换密码的，因为user\_t的上下文与可以更换密码的passwd\_t的上下文不同。

为了解决这个问题，标准linux中是通过setuid来解决的



而selinux中是用domain transition来解决该问题的。



为了完成domain transition，我们需要做如下三步来搞定这个动作：

1. allow user\_t passwd\_exec\_t :file { getattr execute }

首先，你得让user\_t域中的进程能够执行type为passwd\_exec\_t的文件

1. allow passwd\_t passwd\_exec\_t : file entrypoint

然后，你还得告诉SELinux，切换入口（对应为entrypoint权限）为执行passwd\_exec\_t类型

1. allow user\_t user\_t: process transition

最后，你还得告诉SELiux，允许user\_t做DT切换以进入user\_t域

如果要确保domain transition的正确性,就必须保证三个要素：

1. 确保新的domain type有对可执行文件有入口点的权限
2. 确保当前domain type对入口文件有执行权限
3. 确保当前domain type对新的domain type有transition权限

为了保证domain transition被正确执行,selinux中定义了宏来帮助我们，在external/sepolicy/te\_macros中

#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**;

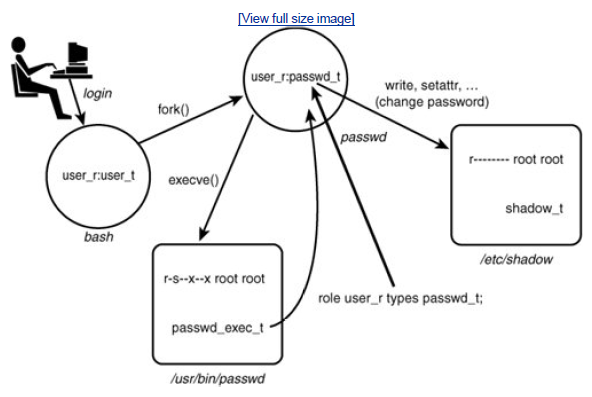
**dontaudit $1 $3:process noatsecure**;

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

')

Roles:

Selinux中还提供了一种RBAC(role-based access control)的形式。这种形式是基于type enforcement的；在selinux中权限控制是通过type enforcement来完成的。而Roles限制了能够进行切换的domain的上下文。以上面更换密码的例子来说：



我们需要为我们的role(user\_r)中添加一个权限:

Role user\_r types passwd\_t

这个语句声明了了一个user\_r 的role 并且将type为passwd\_t的进程绑在了user\_r.这种联合意味着这个passwd\_t 可以安全的被user\_r执行

Selinux的Multilevel安全模式:

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。

(注意：你可以控制selinux启用mls还是不启用mls)，完整的SContext为：

* MLS未启用前：user\_u:role\_r:type\_t
* MLS启用后：ser:role:type:sensitivity[:category,...]- sensitivity [:category,...]

四种操作符:

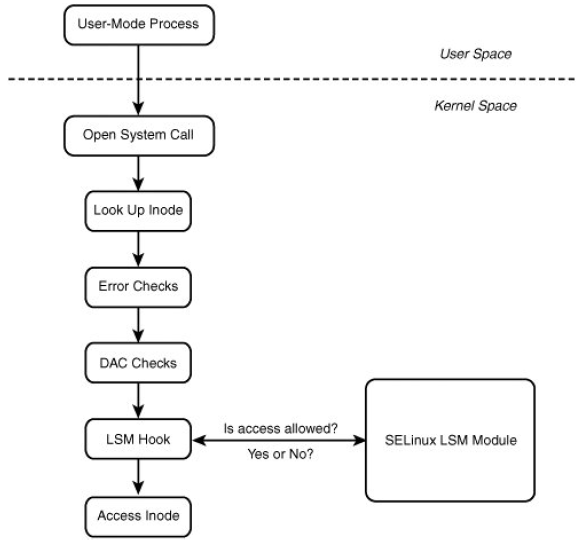
dom,domby,incomp,eq：仅针对role，表示统治，被统治，没关系和相同（和==一样）

第三部分：架构

1.内核层架构

a)LSM框架

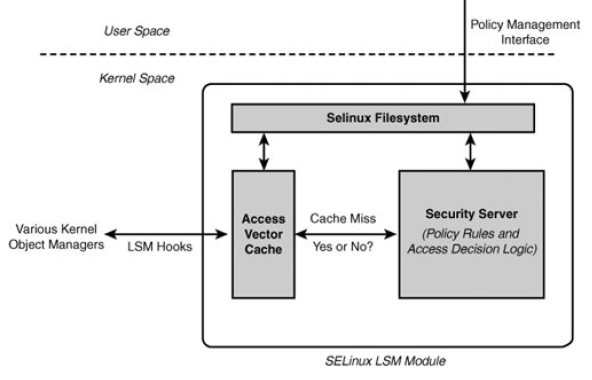
LSM框架思想是用来允许安全模块通过接入到内核的方式来进一步限制默认的linux的安全模型(DAC)。LSM在内核系统的调用逻辑中提供了一套hooks。这套东西会紧跟在标准linux全县检查后面。调用逻辑如下图:



使用LSM框架的一个结果就是SElinux会在标准linux权限检查成功后才起作用。在实际情况中，因为SElinux的权限控制比标准linux的DAC安全模式严格且不会覆盖DAC的结果，SElinux没有任何不利因素。

b)SElinux LSM模块

SElinux内核架构是一个为了微型内核环境开发的沙箱结构。这个结构有三个主要组件：security server，object managers,和the access vector cache.如下图所示:



Security server:负责的是策略判断。Security server位于内核层，而使用的策略却是通过位于用户层的策略管理接口来获得的规格来呈现的。通过这些规则的多样性，才使得SELinux能够高度适配多种安全规则。沙箱结构也是为了能够将security server能够通过代码来替换从而来实现新的权限控制策略而不需要更换整个架构。在实际使用中，不需要替换security server，因为type enforcement提供了几乎所有的权限控制安全策略。

Object managers:指的是security server通过策略判断的控制的资源。在内核空间中，可以看作是建立和管理内核等级的对象的模块。就像文件系统,进程管理系统,进程交互系统。在LSM架构中，object managers表现为LSM hooks。这些hooks散布在内核的模块中，能够调用SELinux的LSM模块来进行权限判断，从而能够强制确定是否允许或者拒绝调用内核资源。

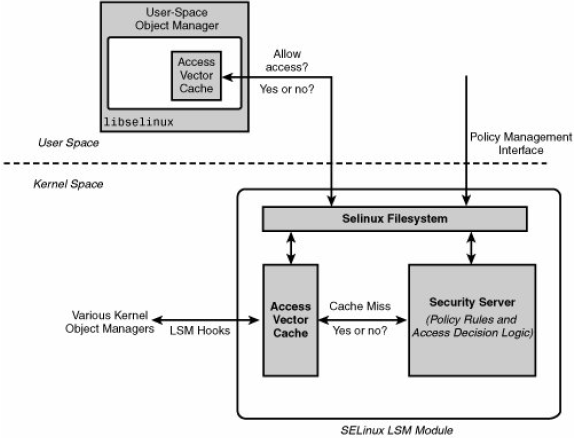
The access vector cache(AVC):AVC被security server调用，使得在权限检查中性能提升，还开放了SELinux的接口来被LSM hooks和内核调用

2.用户层对象管理器

SELinux结构中最强大的功能之一就是可以将用户层的资源应用到内核层。

1. 用户层对象管理器的内核支撑

SELinux可以直接通过内核的security server来支撑用户对象。如下图所示:



使用这种方法时，用户层对象管理器就像是内核层对象管理器，必须查询内核来获得权限控制。最大的不同就是用户层对象管理器不能使用内核层AVC。每个服务器都必须有自己独立的AVC来保存一些内核需要的策略。用户层的AVC存放在libselinux的库中。

另一个不同就是用户对象管理器不存在LSM hooks。但是通过包含在库libselinux中自身的AVC的内部接口，AVC可以处理一些缓存和查询来支撑这个管理器

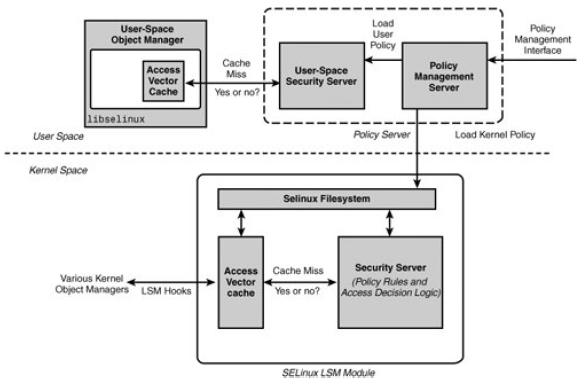
这种方法还存在了一些缺点。第一个弱点，在运用type enforcement时，该管理器必须要定义一些类来表示资源；第二个弱点，使用这种方法，内核security server管理了一些不存在内核的类。这会增加内核的内存开销和内核策略校验的花费

1. 策略服务器架构:

为了规避使用内核security server来为用户对象管理器支撑和增强SELinux的安全能力，需要在用户层中建立一个能够支撑用户对象管理器的工程，这个工程有两个主要目的：

1. 通过建立一个用户层security server来对用户对象管理器更好的支撑
2. 通过建立一个策略管理服务器用一部分空间表示用户对象管理器的对象而提供细化的权限控制。

如下图所示:



在这个架构中，所有的操作和管理都是由policy managerment server(PMS)控制的。PMS是一个可以创建表示资源的类和利用这些资源加强和细化权限控制。这个特点是SELinux的重要的安全强化。

PMS的另一个特点就是将内核层和用户层分离。用户层对象管理器会通过用户层的安全服务器（USSS）来控制。这样可以解耦内核层和用户对象管理器，用户对象管理器会查询USSS，AVCs也会通过USSS注册，更新。

第四部分：类和权限

1.类的目的:

类和权限是SELinux权限控制的基础。类代表着一系列资源（文件，套接字等），权限代表着对这些资源的操作（读，写，连接等）。类和权限不仅仅是在SELinux中存在，在标准linux中也是存在的。

有的时候，权限不只是一个，有可能是是一组权限，也被称为（access vector），代表着资源可以执行的所有操作。这些类和权限也有版本，是跟随着SELinux和内核Linux的版本走的。

2.定义类:

例子:

external/sepolicy/security\_classes

#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。

定义权限:

Permissions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Permissions就包括read，write，open，create,execute等。

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

[external/sepolicy/access\_vectors]

#SELinux规范中，定义Permissions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Permissions命令class所继承

#以下是Android平台中，file对应的权限（Permissions）。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Permissions，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 }

3.常用类和常用权限:

文件相关类:

|  |  |
| --- | --- |
| 客体类别 | 描述 |
| filesystem | 文件系统（如一个真实的分区） |
| file | 普通文件 |
| dir | 目录 |
| fd | 文件描述符 |
| lnk\_file | 符号链接 |
| chr\_file | 字符文件 |
| blk\_file | 块文件 |
| sock\_file | UNIX域套接字 |
| fifo\_file | 命名管道 |

网络相关类:

|  |  |
| --- | --- |
| 客体类别 | 描述 |
| socket | 其它的套接字 |
| tcp\_socket | TCP套接字 |
| udp\_socket | UDP套接字 |
| rawip\_socket | 发送原始IP数据包的套接字 |
| node | 代表一个IP地址或一段IP地址的主机 |
| netif | 网络接口（如eth0） |
| netlink\_socket | 所有其它的Netlink套接字 |
| packet\_socket | 发送其它类型的原始数据包的套接字 |
| key\_socket | PF\_KEY协议家族的套接字，用于管理IPsec中的密钥 |
| unix\_stream\_socket | 本地机器上（unix域）的IPC流套接字 |
| unix\_dgram\_socket | 本地机器上（unix域）的IPC数据报套接字 |
| netlink\_route\_socket | 用于控制和管理网络资源如路由表和IP地址的Netlink套接字 |
| netlink\_firewall\_socket | 用于创建用户空间防火墙过滤器的Netlink套接字 |
| netlink\_tcpdiag\_socket | 用于监视TCP连接的Netlink套接字 |
| netlink\_nflog\_socket | 用于接收Netfilter日志消息的Netlink套接字 |
| netlink\_xfrm\_socket | 用于获取、管理和设置IPsec<IP Security>参数的Netlink套接字 |
| netlink\_selinux\_socket | 用于接收策略载入通知，强制模式切换和清空AVC缓存的Netlink套接字 |
| netlink\_audit\_socket | 用于控制审核的Netlink套接字 |
| netlink\_ip6fw\_socket | 用于创建用户空间IPV6防火墙过滤器的Netlink套接字 |
| netlink\_dnrt\_socket | 用于控制DECnet路由的Netlink套接字 |
| association | IPsec安全联盟 |
| netlink\_kobject\_uevent\_socket | 用于在用户空间接收内核事件通知的Netlink套接字 |
| appletalk\_socket |  |
| packet |  |
| dccp\_socket |  |
| tun\_socket |  |

IPC相关类：

|  |  |
| --- | --- |
| 客体类别 | 描述 |
| sem | 信号量 |
| msg | 消息队列中的消息 |
| msgq | 消息队列 |
| shm | 共享内存段 |
| ipc | 已经没有使用了 |

其他杂类：

|  |  |
| --- | --- |
| 客体类别 | 描述 |
| security | 内核中的SELinux安全服务器 |
| process | SELinux中的进程 |
| system | 整个系统 |
| capability | Linux中表示权利的特权 |
| key |  |
| memprotect |  |
| peer |  |
| capability2 |  |
| kernel\_service |  |

文件权限:

权限主要分为三类：直接映像到标准Linux访问控制权限的权限，标准Linux权限的扩展和SELinux特定的权限。

|  |  |  |
| --- | --- | --- |
| file权限 | 描述 | 分类 |
| ioctl | ioctl（2）系统调用请求 |  |
| read | 读取文件内容，对应标准Linux下的r访问权 | 标准Linux权限 |
| write | 写入文件内容，对应标准Linux下的w访问权 | 标准Linux权限 |
| create | 创建一个新文件 |  |
| getattr | 获取文件的属性，如访问模式（例如：stat，部分ioctl） |  |
| setattr | 改变文件的属性，如访问模式（例如：chmod，部分ioctl） |  |
| lock | 设置和清除文件锁 |  |
| relabelfrom | 从现有类型改变安全上下文 | SELinux特定权限 |
| relabelto | 改变新类型的安全上下文 | SELinux特定权限 |
| append | 附加到文件内容（即用o\_append标记打开） | 文件扩展权限 |
| unlink | 移除硬链接（删除） |  |
| link | 创建一个硬链接 |  |
| rename | 重命名一个硬链接 |  |
| execute | 执行，与标准Linux下的x访问权一致 | 标准Linux权限 |
| swapon | 不赞成使用。它用于将文件当做换页/交换空间 | 文件扩展权限 |
| quotaon | 允许文件用作一个限额数据库 | 文件扩展权限 |
| mounton | 用作挂载点 | 文件扩展权限 |
| audit\_access |  |  |
| open | 打开文件 |  |
| execmod | 使被修改过的文件可执行（含有写时复制的意思） | SELinux特定权限 |
| execute\_no\_trans | 在访问者域转换的执行文件（即没有域转换） | SELinux特定权限 |
| entrypoint | 通过域转换，可以用作新域的入口点的文件 | SELinux特定权限 |

1.标准Linux权限

权限read，write和execute基本上与标准Linux学科读，写和执行（即r，w，x）类似，但与标准权限检查有些不同，在标准Linux中，访问权通常是在文件打开时进行检查，在SELinux中，访问权是每次使用时都会检查，read权限包括了读取整个文件的能力，它包括以一种随机方式访问文件的权限，write权限包括写入文件的权限，包括附加。与read权限类似，write权限包括了随机访问写，当一个文件被映像到内存中时，也会检查read和write权限，例如：mmap(2)系统调用，或使用mprotect(2)系统调用保护现有映像被改变。

execute权限使用execve(2)系统调用控制执行文件的能力，不管有没有域转换（参考下面的exec\_no\_trans），它都是必需要有的，execute权限在成功使用一个文件作为共享库时也是必需的。

2.标准Linux访问控制的扩展

SELinux的一个好处就是它提供了额外的权限，可以更细粒度地进行控制。

在标准Linux中，创建文件的能力受到写入容纳该文件的目录的权限限制，在SELinux中，create权限直接控制创建每个特定SELinux类型文件的能力，使用这个权限，我们可以允许一个域类型创建etc\_t类型的文件，而不是shadow\_t类型，与SELinux中的大多数权限类似，文件create权限是必需的，但并不充分，例如：创建域类型也必须要有权在dir客体中创建客体，并且要有创建file客体的权限，我们可能需要write权限。

最后的文件扩展权限是mounton，quotaon和swapon。mounton权限控制使用文件作为一个挂载点的能力（mount(2)系统调用），更常见的是使用目录作为一个挂载点，然而，在执行绑定挂载时（MS\_BIND），可以使用文件作为挂载点。quotaon权限控制存储限额信息，当使用quotactl(2)系统调用（q\_quotaon）开启限额时，存储限额信息的文件的路径就确定了，调用进程域类型必须要有那个文件的quotaon权限，才能成功完成系统调用。

3.SELinux特定权限

对于文件而言有五种SELinux特定权限：relabelfrom，relabelto，execute\_no\_trans，enTRypoint和execmod。

relabelfrom和relabelto权限控制域类型将文件从一个类型改为另一个类型的能力，为了使重新标记文件成功，域类型必须要有该文件客体当前类型的relabelfrom权限，并且还要有新类型的relabelto权限，注意这些权限不允许控制确切的权限对，域可以将它有relabelfrom权限的任何类型改为它有relabelto权限的任何类型，在重新标记时可以增加约束。

execute\_no\_trans权限允许域执行一个无域转换的文件，这个权限还不够执行一个文件，还需要execute权限，没有execute\_no\_trans权限，进程可能只能在一个域内执行，如果我们想要确保一个执行过程总是会引发一个域转换（或失败）时，此时就会想要排除execute\_no\_trans权限，例如：当登陆进程为一个用户登陆执行一个shell时，我们总是想要shell进程从有特权的登陆域类型转移出来。

entrypoint权限控制使用可执行文件允许域转换的能力，execute，execute\_no\_trans和entrypoint权限允许精确控制什么代码可以执行什么域类型，SELinux控制各个程序域类型的能力是它能够提供强壮灵活的安全的主要原因。

execmod权限控制执行在进程内存中已经被修改了的内存映像文件的能力，这在防止共享库被另一个进程修改时非常有用，没有这个权限时，如果一个内存映像文件在内存中已经被修改了，进程就不能再执行这个文件了。

第五部分:type enforcement

1.te的概念

大多数SELinux的策略内容是由多条type endorcement(te)语句构成的，这些规则控制被允许的使用权，大多数默认转换标志，审核，以及固定部分的检查。

一个完善的te策略往往包含了数万条的te规则，绝大多数的语句是用来允许调用linux内核资源，也就是说每个进程对每个资源的访问尝试都必须至少要有一条允许的te访问规则。

全部的te规则数量庞大，会对我们理解SELinux策略是一个挑战。但是单条语句来说，并不复杂。它们的分类较少，大致上分为两类:

* Access vector(AV)
* Type rules

我们运用AV规则来允许或者检查两个类型，我们通过上下文借由类型来联系各资源。

SELinux中的一个重要概念就是te规则将权限和程序的访问结合在一起，而不是同用户。所有的SELinux的语言规定了程序访问类(文件 套接字等)的权限。这种侧重在程序访问权限的特点是利用SELinux的好处之一。它允许编写者基于功能和程序安全性，再加上用户完成任务所需要的所有权限编写策略。这样就可以严格控制程序，让它在获得最小权限的情况下仍能功能完善的运转。

2. type attribute 和aliases

Types,(类型)是te规则的基本组成部分。SELinux是基于类型来决定哪些权限是被允许的。Attribute(属性)和aliases(别名)是为了减轻管理和使用类型而创建的。我们使用属性来指代一组类型。在大多数情况下，属性可以替代类型在te规则的位置。我们给类型定义一个新的名字来方便我们使用，也就是别名。

**type type\_name [ alias alias\_set ] [ ,attribute\_set];**

上面是定义类型的完整语法。

在SELinux中external/sepolicy这个文件中

#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

#typeattribue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mlstrustedsubject属性关联了起来

**typeattribute system mlstrustedsubject**

Type和attribute的用法

* 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。如下例子所示：

[例子]

#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。

3.AV规则：

AV规则主要有四种规则:

• **allow**： 表示允许主体对客体执行允许的操作

• **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)

• **auditallow**：表示允许操作并记录访问决策信息(允许操作且记录)

• **neverallow**： 表示不允许主体对客体执行指定的操作

语法:

• 规则名称:allow，dontaudit,auditallow,neverallow

• 源类型:授予访问的权限的类型，通常是进程的域类型

• 目标类型:被授权访问的类型

• 对象类:被访问的类

• 权限:被允许的权限(AV)

一个简单语句为:

allow user\_t bin\_t : file execute;

AV规则的键

在内核中，所有的规则都是唯一通过一组源类型，目标类型，对象类作为密钥来唯一确定的。这组密钥被称为AV规则的键，存放在一张hash表中，缓存在策略数据结构中，规则是靠这个密钥存储和检索的，当一个进程产生了一个访问请求时，SELinux LSM模块被要求允许基于这个密钥进行访问。

3.type规则:

类型规则为运行状态的时候创建或重新打戳的对象指定了默认类型。类型规则有两类:

* Type\_transition 在域转换过程中标记行为发生时以及创建对象时，指定其默认的类型。
* Type\_change 在被SELinux的应用重新标记时，指定其默认的类型

语法:

* 规则名称： type\_transition和type\_change
* 源类型：创建或进程所属的类型
* 目标类型:包含的新的或重新标记的类型
* 对象类:即将被创建或重新标记的对象
* 默认类型:新创建或重新标记的类型

完整语法:

**rule\_name source\_types target\_types ：object\_class default\_types**

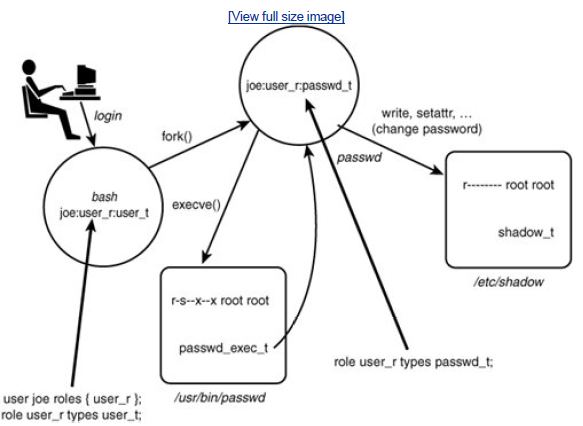
第六部分:角色和用户

1. SELinux的RBAC

a.概述

在SELinux刚开发的时候，RBAC是在te规则的基础和支撑下建立的。我们通过将domain类型和一个多多个角色联结来控制一个角色的权限。RBAC的语句并不确保权限。后来，RBAC演化成通过包含domain类型，角色，用户的上下文来限制te策略。域转换是受用户的角色约束的，最终约束了用户的总体权限。

例子如下:



这个例子说明了,一个用户声明语句(user)和两个角色声明语句(role)。这些语句在策略中联结了用户，角色，类型标识符。

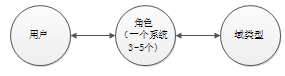
上述user声明语句声明了用户joe和角色user\_r.这句画告诉了SELinux joe和user\_r在上下文中是共存的。没有这句话的话，用户joe和角色user\_r执行上下文的时候是非法的且SELinux会造成拒绝创建对象和拒绝域切换的结果。

这两个role声明联结了角色user\_t和域user\_t和passwd\_t,同user声明一样，role声明也是为了让执行上下文合法。在特殊情况下，没有role声明联结域passwd\_t，即使te策略允许了，域切换也会失败。如果我们不想user\_r角色来执行更改密码的程序，我们只要简单的去掉这个role声明，即使te规则允许了该权限，上下文也永远不会被内核创建。

b. 用角色管理用户权限

就像上面的例子讲的，我们不能直接将域和用户联结到一起。取而代之的是，我们将角色和域联结到一起，然后再将角色和SELinux的用户关联到一起。这就间接联结有两个目的:

* 这会简化管理所有策略的复杂度，一个系统通常只有几个roles，但有可能有成千上万个用户和域。直接关联域和用户会使得管理难度增加。将域指向给一个拥有权限集的为数不多的角色，然后将这些角色指向给用户是更方便管理的方法。
* SELinux中的roles基于自身的权限和责任通过“活动”的role来允许我们来限制users的权限。在任何一个进程中，同一时间只有一个“活动”的role



C. 上下文里面的用户和角色

在更改密码的实例中，我们并没有包含完整的上下文。这说明了相对来说，user和role在objects上下文中重要性要低一点。即使这样，object也拥有完整的上下文，用户字段用于审查，角色字段没有用户。

2.角色和角色声明

除了object\_r外，SELinux中并没有内建的roles。Roles就像域一样需要被策略来声明和定义。Role有四种策略声明语句:

* 角色声明语句
* 角色allow语句
* 角色转换语句
* 角色控制语句

a.角色声明语句

如果没有被声明且联结到域，role声明语句就声明了一个role标识符，如下语句:

**Role user\_r types user\_t;**

**Role user\_r types passwd\_t;**

这些语句将域user\_t 和passwd\_t联结到了角色user\_r。如你所见，role语句可以对相同的role标记符进行重复定义。第一个role语句声明了一个role也可用于后续的声明中。对同一个role多重声明联系一个类型集是常见的用例

具体语法为:

**Role role\_name [types type\_set];**

b.角色allow语句

SELinux中提供了一种在程序执行中通过execve()系统函数调用更改role的方法。这个特点近乎于域切换。角色allow语句通过声明控制着role在程序执行的时候从源role切换到目标role。Role切换需要用户授权给新的role，并且通过allow语句声明权限。如下例子:

**Allow staff\_r sysadm\_r;**

这个例子中允许角色staff\_r在角色转换中转换成角色sysadm\_r，这个转换只存在从staff\_r转换到sysadm\_r，需要再写另一条从sysadm\_r转换到staff\_r的语句。

具体语法为：

**Allow role\_set role\_set;**

c.角色转换语句

因为role可以在程序执行过程中改变，我们需要一种用策略语言去实现自动转换role的方法。类比域，我们利用域切换去自动完成，对于role，我们也需要role转换语句(role\_transition)。这语句和域切换类似除了它具体指出了在执行文件是的默认role转换。如下例子:

**Role\_transition sysadm\_r http\_exec\_t system\_r;**

这个例子中，除非出现其他条件，当一个程序在角色sysadm\_r环境下在域http\_exec\_t中执行一个文件时，SELinux会尝试将角色切换成system\_r。

具体语法为:

**Role\_transitioin role\_set type\_set role;**

d.角色控制语句

角色控制语句声明了一个role和其他role的关系。我们可以用这些语句去创建一个roles的关系图。在这种情况下,处于父角色的role自然就会包含它和它子角色相关的域。如下例子:

**Dominance { role super\_r { role sysadm\_r; role secadm\_r } }**

这个例子中声明了一个父角色super\_r,当它被声明出来后，他就可以控制子角色sysadm\_r和secadm\_r。这个父角色super\_r中会包含所有的sysadm\_r和secadm\_r有关联的域，如果关联关系被其他高等级角色改变后，改变也会发生在super\_r中。注意任何添加到父角色的类型都没有通过dominance语句进行继承。因此如果在dominance语句后再有域绑定到secadm\_r中，super\_r是不会在关联新的域的。这个语句并没有广泛被使用。

具体语法为:

**Dominance { role role\_name { role\_set } }**

3.用户和用户声明

Linux和SELinux的用户标记符是不同的，不相关的。在SELinux中，进程的用户标记符在Linux中和在SELinux中是不同的。SELinux设计这种有别于传统Linux的用户标记符视为了建立一套不可改变的用户标记符系统。在传统Linux中，用户标记符的改变会改变权限(如:root用户)。在大多数情况下，真正的和起作用的用户表支付都会跟着改变。这样对跟踪用户记录和审查，授权及其他都是更困难的。传统Linux和SELinux的用户标识符分开使得在Linux用户标识符变更的时候不会影响SELinux

a.声明用户和关联角色

用户的声明语句在策略中声明了一个用户标记符，并且将它和一个或多个角色关联起来。这个用户声明是一个唯一的和SELinux的用户想关联的策略语句。如下例子:

**User joe roles { user\_r };**

这个例子声明了一个用户joe，且将角色user\_r关联到了joe上。不像角色声明语句夹杂在众多te语句中，用户声明语句会紧跟在所有域声明语句且会在约束条件前。

具体语法为:

**User user\_name roles role\_set;**

第七部分:约束

1.概述:

我们在SELinux中的LSM模型中，access vctor cache(AVC)是被一组包含了安全标识符，目标标识符和目标类别。Sids为上下文的内部引用。

其中约束的主要目的是对一些权限实现全局限制不管是否有allow规则。所有的约束条件每次都会被security\_compute\_av()所检查来返回给AVC。所以，约束可以在SELinux策略中进一步限制权限。

SELinux中有两种类型的约束。Constrain语句是最常用的约束条件，可以让你基于用户，角色，或源类型和目标类型的上下文来进一步限制权限。Validatetrans语句是一个最近新加入SELinux，可以让你基于旧的，新的，执行的上下文来进一步限制对上下文改变的权限。

2.constrain语句:

Constrain 语句有三个元素:一个被constraint应用的类集，一个将要被约束的权限集，和一个约束的布尔表达式。约束条件语句位于应用的类中。

具体语法为:

**Constrain class\_set perm\_set expression;**

其中expression中有一下关键字

* t1,r1,u1 ：源 域，角色，用户
* t2,r2,u2 ：目标 域，角色，用户

表达式有: == != eq dom domby incomp

==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）

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。

3. Validatetrans语句：

SELinux中还支持第二种约束语句，validatetrans。它可以让我们进一步掌握改变对支持的类的上下文的能力，目前来说，支持的类只有文件管理系统的类。

Validatetrans语句能够让你将新的和旧的类的上下文和对应的第三方的该津城城准备重打标签的类的上下文。因此，新的关键字出现了，t3,r3,u3，代表着执行的上下文的域，角色，用户。\*1关键字代表着旧的上下文，\*2代表着新的上下文。具体语法如下:

**Validatetrans class\_set expression;**

**第八部分:多等级安全**

1.概述

MLS是另一种引用在安全问题上的强权限管理形式，特别应用在政府级别的数据控制。(android也是用的该形式).多数的早期计算机安全探索就是为了实现搭载了MLS权限控制的系统。SELinux就提供了这种可行性支持。尽管te搭建了SELinux权限控制的基础，我们仍然可以通过MLS来提供MLS方式的强制权限管理。在SELinux中，MLS会是te的扩展，你不能脱离te而直接使用MLS。

我们通过新建一个声明自己是MLS策略的内核策略文件来开启SELinux的MLS。这个内核策略的核心方法就是会编译带有-M条件checkpolicy程序。这样，我们就可以通过这个内核策略从而使得内核增加MLS机制。

2.MLS的上下文

如我们较早前所提，当MLS被开启时，上下文会增加两个属性：低安全级别和高安全级别。

* 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。

安全级别本身也有两个属性:重要度(sensitivity)和类别集。重要度是一个反映了数据重要型的等级森严的模型。比如像政府文件分类的最高机密，机密，未指定一样。类别集是无序的反映了数据类型的不同。你必须具有足够高的重要度和正确类别才能访问数据。

a.定义安全等级

我们通过重要度语句来定义重要度,如下：

Sensitivity s0;

Sensitivity s1;

我们还可以为重要度指定别名，如下语句:

Sensitivity s1 alias unclassified;

因为重要度是层级分布的，所有我们必须在策略中通过dominance语句来指定重要度的等级。

Dominance { s0 s1 s2 s3 } # s0 is “low” and s3 “high”

这个语句从低到高列举了重要度，因此s0重要度最低。

类别定义的语句和重要度定义语句几乎是一样的。如重要度一样，类别也可以指定别名。不同的是，类别是没有层级的，不需要dominance语句来声明它们之间的层级。例子如下:

Category c0 alias blue;

Category c1 alias red;

Category c2 alias green;

Category c3 alias orange;

Category c4 alias white;

在android的运用中:

* 只定义了一个重要度 s0
* 定义了1024个类别，从c0 到c1023

重要度和类别组成了安全等级。我们用level关键字来定义安全等级。Level语句指明了你拥有的重要度和类别及。例子如下:

Level s0:c0.c4;

Level s1:c0.c4;

这些语句声明你可以用上面声明的重要度等级来使用上面所有声明的类别。

在android中:

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

b.上下文的MLS扩展:

在MLS的上下文中，上下文新增了两个安全等级:一个低等级/当前安全等级和一个高等级/可获得的安全等级。概括来说,低等级反映了一个对象的当前进程或数据重要度。高等级反映了该用户标识符可获得安全等级。语法如下:

User:role:type:sensitivity[:category,…][-sensitivity[:category,…]]

如果没有指明高安全等级，那么高安全等级就会和低安全等级一样。

3．多等级安全constrains

SELinux支持两种mls约束语句，mlsconstrain和mlsvalidatetrans，都可以允许我们声明mls权限控制规则。这两种声明语句除了可以让你基于包含着安全等级的上下文来控制权限外和其他没有mls的部分是相同的。我们可以在开启了mls的策略中运用这些语句，而在没开启mls策略中，就只能使用constraints。在android中我们使用的是mls语句。

a.mlsconstrain语句:

**mlsconstrain class\_set perm\_set expression;**

例子:

Mlsconstrain file { create relabelto } ( l2 eq h2 );

意味着：如果在file类型中要有create relabelto这两个权限的话，低安全等级必须和高安全等级相同。

b.mlsvalidatetrans语句:

我们还需要检查另外一种mls约束，mlsvalidatetrans。这个语法和validatetrans很想除了没有安全等级的关键字属性(l1-h1,l2-h2,l3-h3)。还有个不同点是mlsvalidatetrans比validatetrans更常用。具体语法为:

**Mlsvalidatetrans class\_set expression;**

例子:

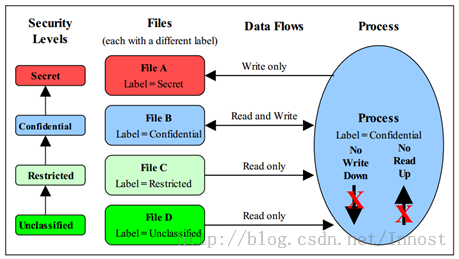
Mlsvalidatetrans file ((l1 eq l2) or ((t3 == mlsfileeupgrade) and (l1 domby l2))

or ((t3 == mlsfiledowngrade ) and (l1 dom l2 or l1 incomp l2 )))

这有一些复杂，首先，基本的要求是当file类的上下文变更的时候，当前安全等级必须相同(l1 eq l2).其中还运用了两个属性(mlsfileupgrade, mlsfiledowngrade)。当进程有mlsfileupgrade这个属性时且l1是l2子集，那么可以upgrading，当进程有mlsfiledowngrade这个属性时且l1是l2的父集，或完全不同，那么可以downgrading。

4.其他作用:

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？



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。如上图，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。

第九部分:条件策略

1.概述:

支持条件策略是SELinux的一个重要特性。它允许我们可以通过一个开关来控制权限集。例子如下:

Bool docked true;

If (docked) ｛

# 允许一些权限

｝ else {

# 允许一些权限

}

我们声明了一个变量docked，然后通过docked的值来选择添加哪些权限进系统

2.布尔变量:

a.定义:

定义语句:

**bool bool\_name default\_value;**

b.管理布尔值:

管理这些布尔值可以很方便的让我们去确定我们所需要的策略。因此，SELinux需要一套系统来能够让我们在运行状态下更改哪些标记的值。不同于那些一旦编译就不会变更的策略，这些布尔值可以在运行状态被改变。

这套更改布尔值的系统成为pseudo文件系统。这套系统是user控件和内核的LSM中间的接口。这些变量存放在**/selinux/Booleans/**中。

例子:

#cat /selinux/Booleans/user\_ping

1 1

第一个1为当前布尔值，第二个1为未定布尔值

更改布尔变量的值需要两个步骤:

1. 更改未定布尔值
2. 提交本次更改

例子:

步骤一：

# cat /serlinux/Booleans/user\_ping

1 1

# echo 0 > /serlinux/Booleans/user\_ping

# cat /serlinux/Booleans/user\_ping

1 0

步骤二:

# echo 1 > /serlinux/Booleans/user\_ping

# cat /serlinux/Booleans/user\_ping

0 0

3.条件语句:

我们需要动态的更改权限集所以引进了条件语句。语法如下:

**If ( cond\_expression ){ true\_list }[ else { false\_list }];**

第十部分:标签

1.概述:

在SELinux中，任何一个对象都有一个上下文与之对应，上下文是SELinux能够进行强权限控制的保证。在SELinux的策略语言中，给进程和文件设置和分配上下文的过程成为标签(labeling)。具体有四种形式:

* 策略声明:通过SELinux的策略语言进行声明。
* 默认:一些对象是被对象管理器编码的时候就分配了上下文了。
* 程序请求标签:SELinux提供了一系列系统级接口来允许程序请求打标签。
* 初始化SIDS:SELinux中包含了一个安全标识码集，用来标签一些对象以防止有漏打和非法打的。

2.文件标签化

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，该文件描述了文件的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，Server 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

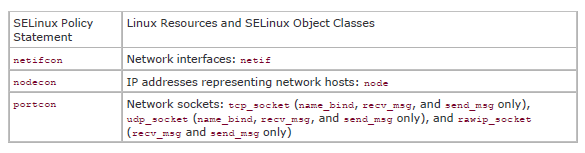
genfscon proc / u:object\_r:proc:s0

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

......

3.网络和套接字标签化

网络和套接字的标签化是用策略声明和初始化sids的方式，程序请求的方式没有运用在网络标签化上。关键字如下:



1. 网络接口标签化

网络接口是被netifcon或者包含netif的sids所标签的。例子如下:

Netifcon eth0 system\_u:object\_:netif\_t

System\_u:object\_r:packet\_t

这则声明提供给了网络设备eth0的上下文是 system\_u:object\_:netif\_t，还有默认的为packets标签的上下文System\_u:object\_r:packet\_t。

语法如下:

**Netifcon interface if\_context packet\_context**

b.网络节点标签化

网络节点是被nodecon或者包含node的sids所标签的。例子如下:

Nodecon 127.0.0.1 255.255.255.255 system\_u:object\_r:node\_lo\_t

这则声明将所有的IPV4的127.0.0.1的节点和255.255.255.255的子网掩码标签了system\_u:object\_r:node\_lo\_t 的上下文。

IPV4如上，IPV^也同样如此

Nodecon ::1 ffff:ffff:ffff:ffff:ffff:ffff:ffff:ffff system\_u:object\_r:node\_lo\_t

语法如下:

**Nodecon subnet netmask context**

c.网络端口标签化

网络端口是被portcon或者包含port的sids所标签的。例子如下:

Portcon tcp 80 system\_u:object\_r:http\_port\_t

这则声明将头部为tcp的80端口标签上了system\_u:object\_r:http\_port\_t的上下文。

语法如下:

**Portcon protocol port\_num context**

d.套接字标签化

套接字在创建后会被自动继承创建它的进程的上下文。套接字可以用来检查上述声明的端口权限。例子如下:

当一个上下文为system\_uLsystem\_r：httpd\_t的进程创建的套接字拥有相同的上下文。它就可以通过这个套接字进行收发讯息。

Allow httpd\_t self : tcp\_socket { read write send\_msg recv\_msg };

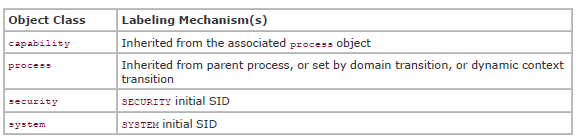
这则语句解释了一个tcp套接字是怎样被标签的。套接字的标签都是在用户空间的进程所打的，包括本地套接字:如netlink和unix domain sockets。被内核创建的套接字会被内核初始化sids来标签它的上下文。

4.IPC标签化

进城间通信的本质是消息的传递，消息会被创建该消息的进程所分配一个上下文。

5.其他标签形式

还有其他标签的关键字如下：



6.初始化安全标识码

初始化安全标识码用于两种环境:策略还没有被加载到系统的时候和有上下文缺失或非法上下文时。

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 unlabel ed

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。