<http://blog.csdn.net/mike8825/article/details/49428417>

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

目录

[1. 简要语法 1](#_Toc509478860)

[1.1. 安全上下文 1](#_Toc509478861)

[1.2. SELinux Policy语言介绍 2](#_Toc509478862)

[1.3. 安全上下文与TE对应关系 2](#_Toc509478863)

[2. 详细语法 3](#_Toc509478864)

[2.1. DAC和MAC 3](#_Toc509478865)

[2.2. SELinux Policy 语言 3](#_Toc509478866)

[2.2.1. Security Context 3](#_Toc509478867)

[2.2.2. TE介绍 5](#_Toc509478868)

[2.2.3. 关联：labeling 11](#_Toc509478869)

# 简要语法

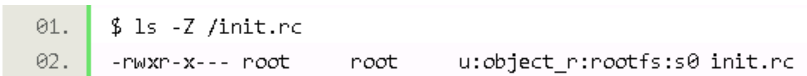
## 安全上下文

安全上下文（Security Context）实际上就是一个附加在对象上的标签（Tag）， 也是用户空间RD修改最多的。这个标签实际上就是一个字符串，它由四部分内容组成，分别是SELinux用户、 SELinux角色、类型、安全级别，每一个部分都通过一个冒号来分隔，格式为：  
“user:role:type:sensitivity”

例子是：

“u:object\_r:default\_android\_service:s0

可用ls –Z查看文件的安全上下文，例



上面的安全上下文表明文件/init.rc的SELinux用户、 SELinux角色、类型和安全级别分别为u、 object\_r、 rootfs和

再看一个进程的安全上下文(ps -Z)



在安全上下文中， **只有类型（Type）才是最重要的**。 SELinux用户、SELinux角色和安全级别都几乎可以忽略不计的，因为截至Android6.0只定义了一个用户u，文件的角色是object\_r，进程的角色是r，安全级别也只有一个s0。

## SELinux Policy语言介绍

TE语言的完整格式为：

rule\_name source\_type target\_type : object\_class perm\_set

⮚ rule\_name： allow、 allowaudit、 dontaudit、 neverallow等。  
⮚ source\_type： **主体进程**类型，也叫subject， domain。  
⮚ target\_type： **客体资源**类型。例如proc代表其后的file所对应的Type。  
⮚ object\_class：它代表能够给subject操作的一类东西，例如File、 Dir、 socket、  
process、 property、 Binder等。  
⮚ perm\_set：在类object\_class中所定义的操作的一个子集

RD最常用语句是allow语句，例子：allow netd proc:file write

⮚ 理解方式一：允许netd进程读（write） proc类型的file；

⮚ 理解方式二：在domain为netd的进程中，允许（allow）在类型为proc的file对象上面执行write操作，也就是file.write；

## 安全上下文与TE对应关系

下面我们看一下log中常见的avc拒绝信息：

[ 27.962493] c3 type=1400 audit(1325820283.567:4): avc: denied { rename }  
for pid=193 comm="installd" name="media" dev="tmpfs" ino=6449  
scontext=u:r:installd:s0tcontext=u:object\_r:tmpfs:s0tclass=dir

• 上面的**scontext和tcontext都是安全上下文**，我们分别称它们为**主体和客体**，**主体一般都是进程，客体则是主体访问的资源**。 它们都有四个字段,这四个字段只需要关注类型即可.如：u:r:installd:s0 分别为： SELinux用户、 SELinux角色、 类型和安全级别。  
• **scontext** 对应前面的 **source\_type**

• **tcontext** 对应前面的 **target\_type**

• **tclass** 对应前面的 **object\_class**

• **{ \* }** 对应前面的 **perm\_set**

# 详细语法

## 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。

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。

## SELinux Policy 语言

### Security Context

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：

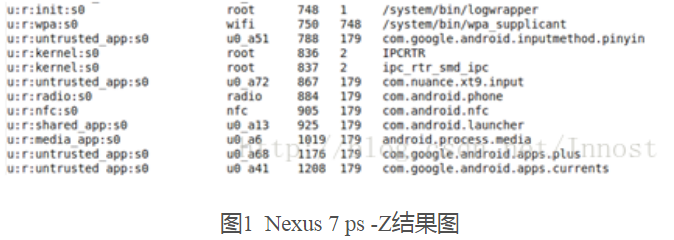


图1中最左边的那一列是进程的SContext，以第一个进程/system/bin/logwrapper的SContext为例，其值为u:r:init:s0，其中：

* 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 Accesc 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

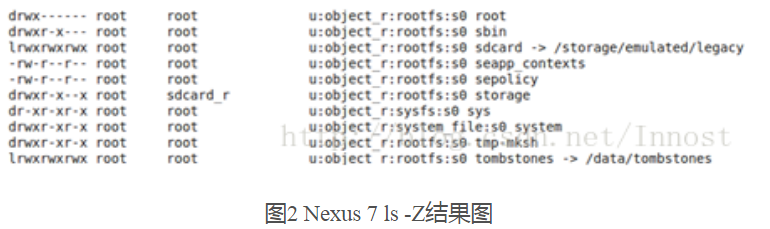


图2中，倒数第二列所示为Nexus 7根目录下几个文件和目录的SContext信息，以第一行root目录为例，其信息为u:object\_r:rootfs:s0：

* 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]

### TE介绍

#### 公式：rule\_name source\_type target\_type : class perm\_set

TE语句最常见的语句就是allow，type等等，下面依次讲解。

[例子]

**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、dontaudit、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。

##### rule\_name

除了allow之外，还有allowaudit、dontaudit、neverallow等，其中：

* 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: 用来检查安全策略文件中是否有违反该项规则的allow语句。

**注意**：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。

##### object class

对应公式中的class。Object class很难用语言说清楚它到底是怎么定义的, 直接上例子：

|  |
| --- |
| [external/sepolicy/security\_classes示例]  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  class process #进程 |

关于class：

* 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
* 这些class和kernel中相关模块紧密结合

##### 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。其中：

* “{}”: 多个perm\_set可用{}括起来
* “~”: 如果前面有一个~号。它表示除了括号{entrypoint relabelto}之外的操作。比如：
  + allow unconfineddomain {fs\_type dev\_type file\_type}:{ chr\_file file } ~{entrypoint relabelto};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
* “-“: 表示去除某项内容
* “\*”: 表示所有内容
  + neverallow { appdomain -unconfineddomain } self:capability2 \*;表示source\_type为属于appdomain，但不属于unconfinedomain的进程。而 \*表示所有和capability2相关的权限

和Object class一样，SELinux或SEAndroid所支持的Perm set也需要声明，声明perm set有两种方法：

1. common命令：common common\_name { permission\_name ... }

例子：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

|  |
| --- |
| common file {  ioctl read write create getattr setattr lock relabelfrom relabelto  append unlink link rename execute swapon quotaon mounton } |

common定义的perm set能被另外一种perm set命令class所继承。

1. class命令：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（即后面的permission name）其实针对得就是某个object class。它不能被其他class继承。

例子：

|  |
| --- |
| class dir inherits file {  add\_name remove\_name reparent search rmdir open audit\_access execmod  } |

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。

##### type

type包括source type和target type。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
* 属性由attribute关键字定义
* 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
* type可以是进程、文件系统等。

【attribute】关键字

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 |

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

【typeattribue关键字】

用于将type与attribute关联

|  |
| --- |
| typeattribute system mlstrustedsubject |

**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:

|  |
| --- |
| #定义两个type，分别是A\_t和B\_t，它们都关联到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}; |

#### 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(Type Enforcement Accesc Control)，属于SELinux MAC(Role Based Accesc Control)中的核心组成部分。

在TEAC之上，SELiunx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？

##### role and user定义

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;** |

##### role之间的转换

我们允许从一个role切换（SELinux用Transition表达切换之意）到另外一个role 。语法为：

allow from\_role\_id to\_role\_id; #这个allow允许from\_role\_id切换到to\_role\_id

**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

##### role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。有员工，也有主管。

dominance语句用来指定这种层级关系：

**dominance { role role\_a {role role\_b; role role\_c; }**

|  |
| --- |
| #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; } |

##### constrain:基于user和role的权限控制

语法格式为：

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) ; |

其中，expression最为关键：

* 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，再说明几点：

* **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。

提示：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！

### 关联：labeling

这些SContext最开始是怎么赋给这些死的（文件）和活的（进程）东西的？

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

#### 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。

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。**sid其实就是上面的type和attribute。**
* initial\_sid\_context：为这些SID设置最初的SContext信息。

|  |
| --- |
| [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 |

#### Domain/type transition

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 Transtition，即某个进程的Domain(某种程度，domain 等同于type或者attribute。进程叫domain，文件叫type)切换到一个更合适的Domain中去。

##### domain transition语法

**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

|  |
| --- |
| #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;  解释为：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(domain transition)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

|  |
| --- |
| #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SELiux，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**。

##### domain\_auto\_trans宏

上述方法导致我们写TE文件时候会比较麻烦。SELinux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;  dontaudit $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', `**  # 先allow相关权限  domain\_trans($1,$2,$3)  # 然后设置type\_transition  type\_transition $1 $2:process $3;  ') |

对于上面的4个步骤的例子，可以简单写为：

domain\_auto\_trans(init\_t, apache\_exec\_t, apache\_t)

##### type transition

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（因为domain和type某种程度是一个意思，一个是活的进程，一个是死的文件），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]  type\_transition acct\_t var\_log\_t:file wtmp\_t;  allow acct\_t var\_log\_t:dir { read getattr lock search ioctl  add\_name remove\_name write };  allow acct\_t wtmp\_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) |