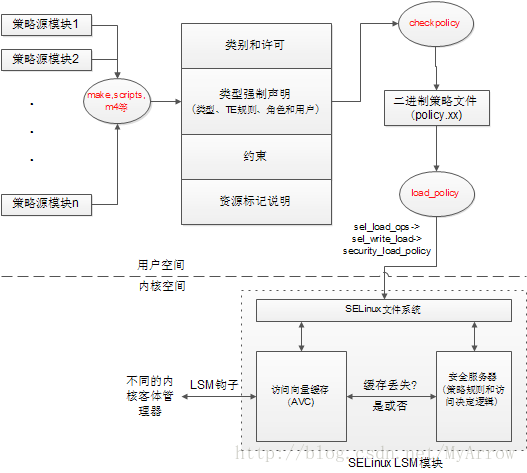
**1. 简介**

     SELinux策略语言主要描述policy.conf的相关语法，其相关部分如下图所示：



**2. 类型强制概念**

     SELinux策略大部分内容都是由多条类型强制规则构成的，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。

     SELinux策略大部分都是一套声明和规则一起定义的类型强制（TE：Type Enforcement）策略，一个定义良好、严格的TE策略可能包括上千个TE规则，TE规则数量的巨大并不令人惊奇，因为它们表达了所有由内核暴露出的允许对资源的访问权，这就意味着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，如果我们仔细思考一下现代Linux操作系统中进程和资源的数量，就明白为什么在策略中有那么多的TE规则了。当我们添加由TE规则控制的审核配置和标志时，对于具有严格限制的SELinux策略，常常会见到它包含有上千条规则，在“创建和编写SELinux安全策略”中，我们将会讨论如何创建和管理这些大量的规则，本文旨在理解TE规则是如何工作的。

    TE规则的绝对数量对理解SELinux策略是一个大的挑战，但是规则本身并不复杂，它们的分类相对较少，所有的规则基本上都属于两类范畴：

     • 访问向量（AV）规则

     • 类型规则

     我们使用AV规则允许或审核两个类型之间的访问权，我们在某些情况下使用类型规则控制默认的标记决定。

     SELinux的一个重要概念是TE规则是将权限与程序的访问结合在一起，而不是结合用户。所有SELinux策略语言特性都是处理主体（正常的运行中的进程）对客体（文件、目录和套接字等）的访问权的，主要集中于程序访问控制决策，这也是SELinux的主要益处，它允许SELinux策略编写者基于程序的功能和安全属性，加上用户要完成任务需要的所有访问权做出访问决策，可以将程序限制到功能合适，权限最小化的程度，因此，即使它出了故障或被攻击破坏，但整个系统的安全并不会受到威胁。

    SELinux是不会管用户的，可以给同一个程序指定多个域类型（因此有不同的特权集），这样就允许引入角色的概念，尽管如此，访问控制的标准仍然是基于程序的域类型而不是用户的特权。焦点是程序的访问权，而不是用户的访问权。

**3. 类型、属性和别名**

    正如你从术语类型强制猜测的那样，类型是构成TE规则的最小单位，SELinux主要就是使用类型来确定什么访问是被允许的；属性和别名是为减轻管理和使用类型的策略特性，我们使用属性利用单个标识符来引用一组类型。通常，策略语言允许我们在TE规则中类型的适当位置使用属性，而别名允许我们为类型定义另一个名字，别名标识符和类型标识符做同等地位对待。

**3.1 类型声明**

    在使用类型前，必须使用type语句明确地声明一个类型标识符，SELinux没有预定义类型，我们必须自行声明，例如：假设我们想声明一个类型（httpd\_t），并打算将其作为Web服务器的域类型，而另一个类型（http\_user\_content\_t）准备应用于用户数据文件，即Web服务器显示内容的文件，我们使用type语句进行声明，如下：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type httpd\_t;
2. type http\_user\_content\_t;

    声明了类型后就可以在安全上下文、TE规则和其它策略语句中使用它们了。

***类型声明的语法：***

    •  type 类型名称 [alias 别名集] [,属性集];

   1) 别名集：如果指定的不止一个别名标识符，要在一对大括号中用空格将各个别名区别开来，如：alias {aliasa\_t aliasb\_t}。  
   2) 属性集：是一个或多个预先声明的属性标识符，如果同时指定多个属性标识符，属性之间使用逗号进行分隔，如：type bin\_t, file\_type, exec\_type;

   3) 类型声明在整个策略中，以及基础载入模块和非基础载入模块中都是有效的。但在有条件的语句中无效。

**3.2 类型和属性**

      可能你已经想到，一个大型的，复杂的策略可能包括上万个代表系统上不同资源的类型，例如：Fedora Core 4（FC 4）的targeted策略相对较小，但也声明了超过800个类型。由于默认的规则是拒绝所有的访问，所以任何一个访问都需要明确地被允许，这就导致了类型注定会很冗长，这时策略语言的属性特性就派上用场了。属性可以理解为：

      1) 类型的性质或属性，或二者兼得

      2) 一组类型

      在任何一种情况下，原理都是相同的。

      假设我们想让一个备份程序可以访问所有的文件，首先我们创建一个备份应用程序的域类型（backup\_t），并允许它访问与任何文件关联的类型：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type backup\_t;
2. allow backup\_t httpd\_user\_content\_t : file read;
3. allow backup\_t shadow\_t : file read;

     为了完成这个例子，我们编写了一个用于其他所有文件类型的规则，依赖声明的类型数量，我们需要大量的allow规则授予备份程序足够的访问权（每个类型都要一个）。另外，每次向策略中增加一个文件类型时，同时要为backup\_t增加一条allow规则，这是一个单调且错误频出的过程，属性使得这种"组访问"更容易指定，通过将所有关联的文件类型定义一个属性，然后授予该属性的访问权（而不是每个类型了），于是，我们可以使用一条规则授予backup\_t必需的访问权。  
     使用attribute语句进行属性声明，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. attribute file\_type;

     这个语句声明一个叫做file\_type的属性，类型和属性共享相同的命名空间，因此，类型和属性的名字不能雷同，假设我们将所有适当的类型都与属性file\_type进行关联，然后就可以使用一条规则来指定backup\_t读取这些文件，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow backup\_t file\_type :file read;

     在我们使用了一条规则替代了上千条allow规则，而授予的访问权却是一样的，当这个策略编译好后，这条规则会自动扩展成上千条规则，分别控制不同文件类型的访问，更重要的是，当我们给文件定义了一个新类型时，我们需要做的仅仅是将这个新类型与file\_type属性进行关联，域类型backup\_t将会自动获得读取访问权。

***属性声明语法：***  
    • attribute 属性名称;

    1) 属性和类型，别名都在同一个命名空间，因此不能与其他类型或别名重名。

    2) 属性声明在整个策略，基础载入模块和非基础载入模块中都有效，但在有条件的语句中无效。

**3.3 关联类型和属性**

**3.3.1 在type中指定属性**

     迄今为止，我们已经讨论了如何定义类型和属性，下面将要讲述的是如何将它们关联起来，最常见的关联方式是在用type语句声明类型时就指定其属性，例如：我们可以将声明http\_user\_content\_t类型的语句修改为：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type http\_user\_content\_t,file\_type;

     这个语句描述了声明类型http\_user\_content\_t时，同时关联了file\_type属性，它会自动向具有file\_type属性的类型组中添加http\_user\_content\_t类型，但从概念上将，它实质上已经改变了http\_user\_content\_t类型的性质，因为它现在已经具有基于属性的访问许可了，而不只局限于类型本身了。

     一个类型可以有多个属性，例如：我们可以再为所有Web服务器要用的文件创建一个属性httpdcontent，拥有httpdcontent属性的类型可能是拥有file\_type属性的类型的一个子集，下面的代码扩展了我们前面的例子：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type httpd\_user\_content\_t, file\_type, httpdcontent;
2. type shadow\_t, file\_type;
4. allow backup\_t file\_type : file read;
5. allow httpd\_t httpdcontent : file read;

     类型具有的属性数量没有限制，就和类型一样，我们可以合理定义相应的属性。

**3.3.2 使用typeattribute指定属性**

    除了使用type语句关联类型和属性外，还可以使用typeattribute语句，这个语句允许我们在声明类型时，单独关联属性，在策略中可能也就是一个单独的文件了，例如：将前面举的示例语句：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type httpd\_user\_content\_t, file\_type, httpdcontent;

     分成两条语句进行表述：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type httpd\_user\_content\_t;
2. typeattribute httpd\_user\_content\_t file\_type, httpdcontent;

      typeattribute允许我们在一个地方定义类型，在另一个地方关联属性，增强了语言的灵活性，在设计策略源文件时，可以考虑进行模块化设计了。

***typeattribute语句语法：***  
    • typeattribute 类型名 属性名;  
      1) 一个或多个事先声明的属性标识符，如果指出多个属性标识符，属性标识符之间使用逗号分隔，如typeattribute bin\_t file\_type, exec\_type;  
      2) typeattribute语句在单个策略，基础载入模块和非基础载入模块中都是有效的，只有在条件语句中无效。

**3.4 别名(为确保兼容性而存在)**

     别名是引用类型时的一个备选的名字，能够使用类型名的地方就可以使用别名，包括TE规则，安全上下文和标记语句，别名通常用于策略改变时保证一致性，例如：一个旧策略可能引用了类型netscape\_t，更新后的策略可能将类型名改为mozilla\_t了，但同时提供了一个别名netscape\_t以保证与旧模块能够正确兼容。

3.4.1 在type中声明别名

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type mozilla\_t alias netscape\_t, domain;

     注意别名声明是放在属性的前面的。

3.4.2 使用typealias申明别名

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. # 这两条语句等同于
2. type mozilla\_t, domain;
3. typealias mozilla\_t alias netscape\_t;
5. #下面这一条语句
6. type mozilla\_t alias netscape\_t, domain;

   ***typealias语句语法***  
      • typealias 类型名称 alias 别名名称

     1) 类型名称：要添加别名的类型的名称，类型必须使用type语句单独声明，而且这里只能指定一个类型名称。

     2) 别名名称：如果同时指定多个别名，别名之间用空格分开，并使用大括号将所有别名括起来，如{aliasa\_t aliasb\_t}。  
     3) typealias语句在单个策略，基础载入模块和非基础载入模块中都有效，只有在条件语句中无效。

**4. 访问向量规则**

    AV规则就是按照对客体类别的访问许可指定其具体含义的规则，SELinux策略语言目前支持四类AV规则：

   在代码中，客体类别的访问许可集是由一些叫做访问向量的掩码表现的，因此就有了术语访问向量。

   • **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

   本小节剩余部分将详细讨论这些规则的语法和语义，以及一些示例。

**4.1 通用AV规则语法**

   虽然这些规则的用途不一样，但它们的基本语法是一样的，每个规则都要包含下面五个元素：

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

     • **源类型**：授予访问的类型，通常是进程的域类型

     • **目标类型**：客体的类型，它被授权可以访问的类型

     • **客体类别**：客体的类别

     • **许可**：表示主体对客体访问时允许的操作类型（也叫做访问向量）。

    一个简单的AV规则有一个源类型，目标类型，客体类别和许可，在我们前面的allow规则中可以看到许多AV规则，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : file execute;

    这个allow规则的源类型为user\_t，目标类型为bin\_t，客体类别file，许可execute，这个规则可以解读为"允许user\_t执行类型为bin\_t的文件"

**4.1.1 AV规则的密钥(哈希Key)**

     在内核中，所有的AV规则都是通过一组【源类型+目标类型+类别】进行唯一性标识，这个三重组叫做一个密钥，当做哈希表使用，缓存在策略[**数据结构**](http://lib.csdn.net/base/31)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。

     那么，如果不止一个规则使用同一个密钥（即相同的源类型，目标类型和许可）时会发生什么状况呢？如下面的规则：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : file execute;
2. allow user\_t bin\_t : file read;

     类型为user\_t的进程对类型为bin\_t的文件是可读还是可执行？答案是两者皆可。所有有相同密钥的规则通过checkpolicy进行组合，编译后的策略将只有一条规则，但它同时具有read和execute许可，它们都会被安全服务器接受。所有的AV都按照这种方式进行累加。

**4.1.2 使用AV规则中的属性**

     虽然到目前为止我们看到的AV规则都很简单，但语法支持多种方法列出类型、客体类别和许可，使我们可以灵活地利用，并使规则语句更简单。

     在前面的简单样式的规则示例中，直接引用了源类型（user\_t）和目标类型（bin\_t），这样在源类型或目标类型中要引用多个类型也是很方便的，其中一个方法就是使用属性，在AV规则中能使用类型的地方都可以使用属性(类型组)。

     例如，假设我们定义了一个属性（exec\_type），我们打算将其与所有的普通用户程序（通过域类型user\_t标记）都可以执行的文件类型关联，那么我们可以将上面的例子改为引用属性exec\_type，而不用再明确地指定类型bin\_t了，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t exec\_type : file execute;

    与属性关联的每个类型都有一个独立的密钥。

    我们也可以在AV的源类型位置处使用属性，或者干脆在源类型和目标类型处都使用属性，例如：假设我们创建了一个属性（domain），并将所有的域类型（包括user\_t）都与其关联，我们想要所有的域类型都可以执行属性为file\_type的文件类型，使用一条规则就实现这个目标：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow domain exec\_type : file execute;

    为了更好地解释规则扩展的原理，假设我们的策略关联了类型为user\_t和staff\_t的属性domain，以及文件类型为bin\_t，local\_bin\_t和sbin\_t的属性exec\_type，那么上面那一条规则的效果就等同于下面这些规则：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : file execute;
2. allow user\_t local\_bin\_t : file execute;
3. allow user\_t sbin\_t : file execute;
5. allow staff\_t bin\_t : file execute;
6. allow staff\_t local\_bin\_t : file execute;
7. allow staff\_t sbin\_t : file execute;

**4.1.3 AV规则中的多类型和属性**

   在AV规则中的源和目标区域，我们都没有限制类型和属性的数量，相反，可以在源和目标字段处列出多个类型和属性，如果有多个类型或属性时，它们之间使用空格进行分隔，并使用大括号将它们括起来，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t { bin\_t sbin\_t } : file execute;

     在这个规则中，目标是bin\_t和sbin\_t，在源和目标区域有多个类型或属性时，展开方法同单个属性一样，在前面的例子中，内核包括两个密钥，每个目标类型都有一个。

     我们还可以在源或目标区域混合类型和属性，也可以在这两个位置都使用混合的形式，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow {user\_t domain} {bin\_t file\_type sbin\_t} : file execute ;

    如果我们明确地列出了类型以及类型具有的属性，这是可以的，即我们实际上列出了两次类型，内核会自动处理这个冗余，只会处理一个实例规则。

**4.1.4 特殊类型self**

    策略语言保留了一个关键字self，它用于AV规则中的目标区域，可以当做一个类型使用，如下面这两条规则是相等的：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. # 这两条规则是相等的
2. allow user\_t user\_t : process signal;
3. allow user\_t self : process signal;

    关键字self说明目标类型使用的源类型自身，即目标类型等于源类型，前面的例子中，第二条规则只是用关键字创建了一条规则，表明源类型和目标类型都是user\_t。

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. # 这两条规则
2. allow user\_t user\_t : process signal;
3. allow staff\_t staff\_t : process signal;
5. #等于下面这一条规则
6. allow {user\_t staff\_t} self : process signal;

    注意：你可能只会在AV规则的目标区域使用特殊类型self，特别要注意的是不能在AV规则的源区域使用self类型，另外，也不能声明一个类型或属性标识符叫做self。

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow domain domain : process signal; # 每个进程都能向它自己和其它进程发送signal
2. allow domain self : process signal;   # 每个进程都能向它自己发送signal

**4.1.5 "非"特殊操作符**

    AV规则中最后一个类型语法是类型否定，它可以从一个类型列表中将某个类型移除，也可以用于用一个属性中移除某个类型，通过在要移除的类型名称前面放一个非操作符（-）实现，例如：我们想让所有的域类型都可以访问所有属性为exec\_type的文件，除了sbin\_t类型外，那么编写规则时就可以这样：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow domain { exec\_type -sbin\_t } : file execute;
2. #等同于
3. allow domain { -sbin\_t exec\_type } : file execute;

    这个规则在展开时就好像属性exec\_type没有包括类型sbin\_t一样。

**4.1.6 在AV规则中指定客体类别和许可**

    AV规则也可以包括客体类别和许可列表，语法和类型一致，使用空格进行分隔，并用大括号括起来，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : { file dir } { read getattr };
3. #这条规则将会产生两个密钥，每个客体类别一个，这条规则等同于下面这两条规则：
4. allow user\_t bin\_t : file { read getattr };
5. allow user\_t bin\_t : dir { read getattr };

   注意客体类别被展开了，但每条规则都有相同的许可列表，这意味着列表中的所有许可对所有客体类别都是有效的。

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. # 无效的规则，因为search对于客体类别file是无效的
2. allow user\_t bin\_t : { file dir } { read getattr search };
4. #当许可对两个客体类别不是都有效时，需要两条规则
5. allow user\_t bin\_t : file { read getattr };
6. allow user\_t bin\_t : dir { read getattr search } ;

**4.1.7 AV规则中的特殊许可操作符**

    对于列在AV规则中的许可，我们可以使用两个特殊的操作符，第一个是通配符（\*），通配符包括了客体类别的所有许可：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : { file dir } \*;

    这条规则扩展后将包括file和dir的所有许可。

    通配符语法与列出所有的许可有点不同，使用通配符时，许可包括每个客体类别的许可，此时不会考虑其中一个许可是否对另一个客体类别是否有效，这样就可以在规则中使用多个客体类别，即使这些客体类别有不同的许可，因此，上面这条规则就安全地处理了许可，不会像前面那条规则那样，这里只对dir客体类别有效的规则不会影响到file客体类别。

    第二个特殊操作符是求补算操作符（~），即除了列出的许可外，其它的许可都包括，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : file ~{ write setattr ioctl };

   编译时，这条规则允许所有的许可，除了write，setattr和ioctl外，与通配符类似，求补算操作符也扩大了客体类别的许可列表。

**4.1.8 通用访问向量规则语法**

***完整的AV规则通用语法如下***：

     • 规则名称 类型集 类型集：类别集 许可集;  
     1) **规则名称**：访问向量规则的名称，有效的规则名称是allow，auditallow，auditdeny，dontaudit和neverallow。

     2) **类型集**： 一个或多个类型和（或）属性，规则中源和目标类型有其独立的类型集，多个类型集或属性使用空格进行分隔，并使用大括号将它们括起来，如{bin\_t sbin\_t}。可以使用（-）来排除类型，如{exec\_type -sbin\_t}。在目标类型区域可以使用关键字self，但在源类型区域不能使用。neverallow规则也支持通配符来代表所有的类型，求补算操作符（~）也表示所有的类型，除了明确列出的之外。

     3) **类别集**： 一个或多个客体类别，多个客体类别必须使用大括号括起来，如{file lnk\_file}。  
     4) **许可集**： 一个或多个许可，所有许可对类别集列出的所有客体类别都要有效，多个许可必须用大括号括起来，如{read create}。通配符（\*）指出所有客体类别的所有许可，求补算操作符（~）用于指出所有的许可，除了明确列出的之外。  
     所有AV规则在单个策略，基础载入模块和非基础载入模块中都有效，所有AV规则除了auditdeny，neverallow规则外，其它的在条件语句中也有效。

**4.2 允许(allow)规则**

     到目前为止，你已经看到了许多的allow规则，allow规则是策略中最常见的规则，它实现了SELinux策略的主要目的（即允许访问）。  
    正如前面讨论的，我们使用allow规则指出了所有运行时授予的许可，它们是SELinux策略中允许许可的唯一方法，记住，**默认情况下，不允许任何访问**，我们指定了两个类型列表（源和目标类型），根据列出的客体类别的许可指定访问权，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. allow user\_t bin\_t : file { read execute };

    这个规则允许任何安全上下文中类型具有user\_t的进程对任何安全上下文中具有类型为bin\_t的普通文件所有read和execute访问权。allow规则共享了通用AV规则的的所有语法，并且也没有增加任何额外的语法了。

**4.3 审核(audit)规则**

   SELinux有大量的工具记录日志信息，或审核、访问尝试被策略允许或拒绝的信息。审核消息通常叫做"AVC消息"，它提供了详细了关于访问尝试的信息，包括是允许还是拒绝，源和目标的安全上下文，以及其它一些访问尝试涉及到资源信息。AVC消息与其它内核消息类似，都是存储在**/var/log**目录下的日志文件中，它是策略开发、系统管理和系统监视不可缺少的工具。在此，我们检查是哪一个访问尝试产生了审核消息。

**默认情况下，SELinux不会记录任何允许的访问检查，只会记录被拒绝的访问检查**。这并没什么奇怪的，在大多数系统上，每秒会允许成千上万的访问，只有很少的一部分会被拒绝，允许的访问通常是在预料之中的，通常不需要审核，被拒绝的访问通常是（但不总是）非预期的访问，对它们进行审核便于管理员发现策略的bug和可能的入侵尝试。策略语言允许我们取消这些默认的预料之中的拒绝审核消息，改为记录允许的访问尝试审核消息。

    SELinux提供两个AV规则允许我们控制审核哪一种访问尝试：dontaudit和auditallow。使用这两条规则我们就可以改变默认的审核类型了，最常用的是dontaudit规则，它指出哪一个访问尝试被拒绝时不审核，这样就覆盖了SELinux默认的审核所有拒绝的访问尝试的行为。

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. dontaudit httpd\_t etc\_t : dir search;

    记住，审核（audit）规则让我们覆盖了默认的审核设置，allow规则指出了什么访问是允许的，auditallow规则不允许访问，它只审核允许的许可。

**注意：**在许可模式和强制模式下审核是不一样的。运行在强制模式下时，每次允许或拒绝时都会进行审核，应该在策略中对审核频率进行限制（可以使用auditctl实现）。在许可模式下时，只会记录第一次访问尝试，直到下一次策略载入，或固定为强制模式，在开发时通常使用的就是许可模式，这种模式可以减少日志文件的大小。

**4.4 neverallow规则**

    最后一个AV规则是neverallow规则，我们使用这个规则来指定永远不会被allow规则执行的访问，你可能会疑惑，为什么会有这个规则？因为默认情况下所有的访问都是被拒绝的，设计这个规则的主要目的是为了帮助编写策略时，可以明确地指出不想要的访问许可，因此可以预防意外发生，回想一下，在一个SELinux策略中可能包含成千上万条规则，可能不小心加入了我们本不想授予的访问权，此时，neverallow规则就可以帮助预防这种情况发生，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. neverallow user\_t shadow\_t : file write;

     这条neverallow规则可以有效地阻止我们在策略中添加一条允许user\_t对类型为shadow\_t的文件进行写操作的规则，如果添加了这样的规则在编译时就会报错，这条规则不会移除访问权，它只是会产生编译错误。我们在编写策略时，neverallow规则往往放在allow规则前面，首先声明哪些访问是明确地被拒绝的，然后再声明哪些访问是可以接受的，这样就可以预防我们人为出错了。

     neverallow规则支持一些特殊的其它AV规则不支持的语法，在neverallow规则中的源和目标类型列表中可以使用通配符（\*）和求补算操作符（~），如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. neverallow \* domain : dir ~{ read getattr };

    这条规则指出没有哪条allow可以授予任何类型对具有domain属性的类型的目录有任何访问权，除了read和getattr访问权外（即读访问权），这条规则的中通配符意味着所有的类型，在真实的策略中，类似这样的规则很常见，它们用来阻止对/proc/目录适当的访问。

    我们从前面这个例子中看出，在源类型列表中需要使用通配符，因为我们想要指出任何类型或所有类型，包括那些还没有创建的类型，使用通配符可以预防我们未来犯错。

    另一个常见的neverallow规则是：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. neverallow domain ~domain : process transition;

    这条neverallow规则增强了domain属性，它指出了进程不能转换到无domain属性的类型，这就使得要为一个类型无doamin属性的进程创建一个有效的策略是不可能的。

**5. 类型规则**

    类型规则在创建客体或在运行过程中重新标记时指定其默认类型，它仅提供一个新的**默认类型标记**。在策略语言中定义了两个类型规则：

     • type\_transition：在域转换过程中标记行为发生时以及创建客体时，指定其默认的类型。

     • type\_change：使用SELinux的应用程序执行标记时指定其默认类型。

     我们叫这些规则为"类型规则"，因为它们与AV规则类似，除了规则的末尾是一个类型名而不是许可集外。

**5.1 通用类型规则语法**

      与AV规则一样，每条类型规则有不同的用途和语义，但它们的语法都是通用的，每条类型规则都具有下列五项元素：  
      • 规则名称：type\_transition或type\_change

      • 源类型：创建或拥有进程的类型

      • 目标类型：包含新的或重新标记的客体的客体类型

      • 客体类别：新创建的或重新标记的客体的类别

      • 默认类型：新创建的或重新标记的客体的单个默认类型

***类型规则语法***  
      • 规则名称 类型集 类型集:类别集  单个默认类型;

      1) 规则名称：类型规则的名称，有效的规则名称有type\_transition，type\_change和type\_member。  
      2) 类型集：一个或多个类型或属性。在规则中源和目标类型有其独立的类型集，多个类型和属性使用空格进行分隔，并用大括号将它们括起来，如{bin\_t sbin\_t}，可以在类型名前放一个（-）符合将其排除，如{exec\_type –sbin\_t}。

      3) 类别集：一个或多个客体类别，多个客体类别必须使用大括号括起来，并用空格分开，如{file lnk\_file}。  
      4) 默认类型：为新创建的或重新标记的客体类别指定的单个默认类型，这里不能使用属性和多个类型。  
      5) 所有类型规则在单个策略，基础载入模块，非基础载入模块和条件语句中都有效。

     类型规则语法大部分都和AV规则类似，但也有一些不同的地方，首先就是类型规则中没有许可，不像AV规则那样，类型规则不指定访问权或审核，因此就需要许可了；第二个不同点是客体类别没有关联目标类型，相反，客体类别指的是将要被默认类型标记的客体。

     最简单的类型规则包括一个源默认类型，一个目标默认类型和一个客体类别，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type\_transition user\_t passwd\_exec\_t : process passwd\_t;

     它指出了当一个类型为user\_t的进程执行一个类型为passwd\_exec\_t的文件时，进程类型将会尝试转换，除非有其它请求，默认是换到passwd\_t，当声明的客体类别是进程（process）时，隐含着目标类型要与file客体类别关联，声明的客体类别（process）与源和默认类型关联，这个隐藏着的关联很容易被忽略，即使你成为一个策略编写专家也容易犯这个错。

**5.2 类型转换规则type\_transition**

     我们使用type\_transition规则指定默认类型，目前有两种格式的type\_transiton规则：

     1) 支持默认域转换事件

     2) 支持客体转换，它允许我们指定默认的客体标记

    这两种形式的type\_transition规则帮助增强了SELinux透明转换到Linux用户的安全性，默认情况下，在SELinux中，新创建的客体继承包括它们的客体的类型（如目录），进程会继承父进程的类型，type\_transition规则允许我们覆盖这些默认类型，这非常有用，例如：为了确保密码程序在/tmp/目录下创建一个文件时要给一个不同与普通用户的类型。

     type\_transition规则没有allow访问权，它仅提供一个新的默认类型标记，要成功进行类型转换，也必须要一套相关联的allow规则，以允许进程类型可以创建客体和标记客体。此外，默认的标记指定在type\_transition规则中了，只有创建进程没有明确地覆盖默认标记行为它才有效。

**5.2.1 默认域转换（进程类型转换process）**

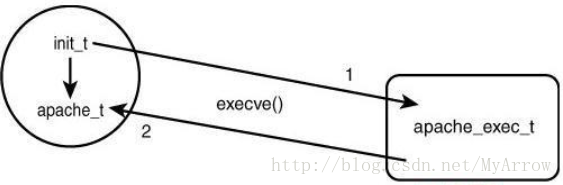
   让我们一起来详细地看一下这条规则中的域转换格式，执行一个文件时，域转换改变了进程的类型，如下面这条规则：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type\_transition init\_t apache\_exec\_t : process apache\_t;

    这条规则指出类型为init\_t的进程执行一个类型为apache\_exec\_t的文件时，进程类型将会转换到apache\_t。客体类别process只表示这是一个域转换规则的格式。

   下图显示了一个域转换，实际上，域转换只是改变了进程现有的类型，而不是新创建了一个进程，这是因为在Linux转换创建一个新的进程首先是要调用fork()系统调用复制一份现有的进程，如果进程类型在fork上被改变了，它就会允许域在新的域中执行任意的代码了，通过execve()系统调用执行一个新的程序时，发生域转换时就更安全些。



      正如前面谈到的，只有当策略允许了有关的访问权时才会发生类型转换，域转换要成功，策略必须允许下面三个访问权：  
      • execute：源类型（init\_t）对目标类型（apache\_exec\_t）文件有execute许可  
      • transition：源域（init\_t）对默认类型（apache\_t）必须要有transition许可  
      • entrypoint： 新的（默认）类型（apache\_t）对目标类型（apache\_exec\_t）文件必须要有entryponit许可

     同时，上面的域转换规则要想成功，还必须要有下面的allow规则：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. # 这条域转换规则.
2. type\_transition init\_t apache\_exec\_t : process apache\_t;
4. # 至少需要下面三条allow规则才能成功
5. allow init\_t apache\_exec\_t : file execute;
6. allow init\_t apache\_t : process transition;
7. allow apache\_t apache\_exec\_t : file entrypoint;

    在实际中，除了上面这几个最小allow规则外，我们可能还想增加一些额外的规则，例如：常见的有默认类型向源类型发送exit信号（即sigchld许可），继承文件描述符，使用管道进行通信。

**域转换**最关键的概念是清楚地定义了入口点，即类型为apache\_exec\_t的文件对新的默认类型apache\_t有entrypoint许可，入口点文件允许我们严格控制哪个程序可以在哪个域中执行（可以认为这就是类型强制的安全特性），我们知道只有程序的可执行文件的类型对域有entrypoint许可时，这个程序才可以进入一个给定的域，因此我们可以知道并控制哪个程序有哪个特权了。

**5.2.2 默认客体转换(file)**

     客体转换规则为新创建的客体指定一个默认的类型，实际上，我们通常是在与文件系统有关的客体（如file，dir，lnk\_file等）上使用这种type\_transition规则，和域转换一样，这些规则只会引发一个默认客体标记尝试，也只有策略允许了有关的访问权时，尝试才会成功。  
    客体转换规则由客体类别进行标记，如：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type\_transition passwd\_t tmp\_t : file passwd\_tmp\_t;

      这条type\_transition规则指出当一个类型为passwd\_t的进程在一个类型为tmp\_t的目录下创建一个普通文件（file客体类别）时，默认情况下，如果策略允许的话，新创建的文件类型应该为passwd\_tmp\_t，注意客体类别目标类型不是tmp\_t而是默认类型passwd\_tmp\_t，在这个例子中，tmp\_t隐含关联了dir客体类别，因为它是唯一能够容纳文件的客体类别，同样，和前面一样，策略必须允许对默认标记的访问，对于前面的例子，对类型为tmp\_t的目录的访问权需要包括add\_name，write和search，对类型为passwd\_tmp\_t的文件要有read和write访问权。

      这个例子很典型，它显示了一个解决在同一个目录下多个应用程序共享和继承的安全问题，如在一个临时目录下，客体转换规则对于那些在运行时创建的客体非常有用。

      某些情况下不能使用客体转换规则，当进程需要在同一个客体容器中创建有多个不同类型的客体时，一条type\_transition规则还不够，例如：假设一个进程在/tmp/目录下创建两个UNIX域套接字，这些套接字将用于和其它域通信，如果我们想给每个sock文件不同的类型，客体转换规则将不能满足了，这时需要两条规则，它们有相同的源类型，目标类型和客体类别，只是默认类型不同，但这样会在编译时产生错误，解决这个问题的办法是在安装时创建sock文件，并明确地标记它们，将sock文件分别放在不同目录类型的目录下，或让进程在创建时明确地请求类型。

**5.3 类型改变规则type\_change**

     我们使用type\_change规则为使用SELinux特性的应用程序执行重新标记指定默认类型，和type\_transition规则类似，type\_change规则指定默认标记，但不允许访问。

     与type\_transition规则不同点如下：

      • type\_change规则的影响不会在内核中生效，而是依赖于用户空间应用程序，如login或sshd。

      为了在策略基础上重新标记客体，如下面的规则：

**[cpp]** [view plain](http://blog.csdn.net/myarrow/article/details/10105961) [copy](http://blog.csdn.net/myarrow/article/details/10105961)

1. type\_change sysadm\_t tty\_device\_t : chr\_file sysadm\_tty\_device\_t;

     这条type\_change规则指出以sysadm\_t名义重新标记一个类型为tty\_device\_t的字符文件时，应该使用sysadm\_tty\_device\_t类型。

     这条规则是最常见的使用type\_change规则的示例，它在用户登陆时重新标记终端设备，login程序会通过一个内核接口查询SELinux模块中的策略，传递类型sysadm\_t和tty\_device\_t，接收sysadm\_tty\_device\_t类型作为重新标记的类型，这个机制允许在一个新的登陆会话过程中，登陆进程以用户的名义标记tty设备，将特殊的类型封装到策略中，而不用硬编码到应用程序中。

    我们可能很少使用type\_change规则，因为它们通常只由核心操作系统服务使用。

**6. 小结**

     • 类型是SELinux中访问控制的主要基础。它们起着所有客体（进程，文件，目录，套接字等）访问控制属性的作用，类型使用Types语句声明。

     • 属性是类型组。在大多数策略中，能够使用类型的地方就可以使用属性。在使用属性前，我们必须先声明，在Types声明语句中，我们可以将类型添加到属性中，或使用typeattribute语句也行。

     • 别名是类型的另一个名字，主要用于重新命名类型时保持向后的兼容性，可以在声明类型时就声明一个别名，或单独使用typealias语句声明。

     • 有四个AV规则，它们的语法都一样：allow，neverallow，auditallow和dontaudit。

     • 我们使用allow规则指出访问一个域类型时需要一个什么客体类型，我们根据客体类别和许可指定访问权。  
默认情况下，访问被允许时不产生审核消息，而是访问被拒绝时产生。我们使用dontaudit规则指出被拒绝的访问不产生审核消息，我们使用auditallow规则指出允许的访问要产生审核消息。

     • AV规则（如allow）是累加的，对于一个给定的源类型，目标类型和客体类别的密钥，在运行时，被允许和被审核的访问权是所有引用了该密钥的规则的并集。

     • 我们使用neverallow规则指出了永远都不会被allow规则允许的固定属性，如果某个allow规则违背了这个原则，checkpolicy编译器在编译时就会产生一个错误。

     • 有两个类型规则，它们的语法是一样的：type\_transition和type\_change。类型规则没有allow访问权，相反，它们指定了客体创建和重新标记事件想要的默认标记策略。

     • 我们使用type\_transition规则在创建新的客体时标记它（客体转换），或在执行一个新的应用程序时改变进程的类型（域转换）。

     • 我们使用type\_change规则为重新标记客体指定默认的类型，它们用于SELinux敏感的程序如login和sshd。

     • 策略分析工具apol在理解和分析复杂的SELinux策略时是一个非常有价值工具。