SELinux 学习笔记

Harry Ciao
harrytaurus2002@yahoo.com.cn

一个人对家庭、公司、社会的价值在于奉献
感谢这几年有机会能够比较系统地接触 SELinux
希望此文能够对 SELinux 在中国的推广和应用起到微薄的促进作用

感恩，感谢，祝愿每个人的心中都充满阳光和快乐

最后更新：2012-2-14

目录

注：
第 1 至 5 章包含自己对《SELinux By Example》一书的学习心得；
第 6 至 9 章包含自己实际经验的总结；
第 10 和 11 章为 SELinux 内核和用户态源代码分析总结；
第 12 章为 refpolicy 的编译过程源代码情景分析；
第 13 章打算收录一些和 SELinux 相关的应用的分析，比如 Labeled Networking，XACE 等。

1，Refpolicy 有 300+个 pp 的实现，了解核心 pp 的实现，必须首先了解应用程序的行为以及安全目标；
2，阅读 SELinux-aware 应用程序源代码，看如何使用 libselinux 库函数，学习 SELinux 用户态编程；
3，了解 SELinux 语法，标识符的二进制表示形式，内核态和用户态的数据结构；
4，学习 policy module 的 compile/expansion/link 过程；
5，按照类别，逐步掌握 LSM 回调函数的调用时机，所在内核子系统的原理，以及 SELinux 对 LSM 回调函数的实现（参考《Implementing SELinux as a LSM module》一文！）
6，掌握 SELinux 对各个内核子系统核心数据结构的扩展；

7，学习 XACE 规范，体会如何开发 Userspace Object Manager
8，积极阅读 Joshua，Dan 等人的 blog，向前辈们学习。
9，积极关注 SELinux 邮件列表上的问答和讨论，丰富自己的视野

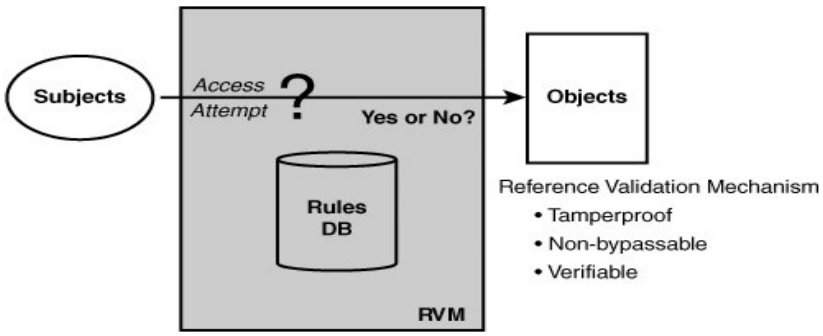
希望能通过自己坚持不懈的努力，证得圆满的 SELinux 知识和智慧。

引子

1，软件的缺陷不可避免（无论过去，现在，或将来）；
2，没有底层操作系统的支持就无法真正实现上层软件的安全性；
　　- D. Baker，《Fortresses Built Upon Sand》

1．操作系统中访问控制模型的演化

1.1 访问控制模型的概念（Reference Monitor）



由上图可见，访问控制模型由如下四部分组成：
1，Subjects：访问的发起者，比如系统中的进程；
2，Objects：被访问的对象，比如操作系统所管理的任何资源、数据（包括进程，任意类型文件，TCP 端口，甚至单个网络报文。总而言之，任何内核数据结构都可能成为被访问的对象）；
3，Rules DB：规则库，在用户态实现的**访问策略**，定义 Object 的属性并规定哪些 Subject 可以通过什么样的方式对它进行访问；
4，RVM(Reference Validation Mechanism)：在操作系统内实现的**机制**，是访问控制策略的**执行者**，在访问操作过程中根据规则库判断当前操作是否合法；

SELinux 首先需要**标识**访问者和被访问的对象，由下文可知相关信息保存在进程的 task_struct 以及其他内核数据结构中 security 指针所指向的数据结构中（注意这些数据结构都是运行时动态创建的，安全属性信息来源于文件在辅存上的扩展属性，以及规则库中的 Initial SID 定义等）。由用户态定义的访问规则库指定哪些访问者能够以何种方式访问哪些对象，而内核中的 SELinux 机制则根据当前操作的访问者和被访问对象，查询规则库得到 Yes/No 结论。

1.2 DAC（Discretionary Access Control）的致命伤

“discretionary”一词的含义为“not controlled by strict rules, but decided on **by someone** in a position of authority”，所以 DAC 的本质是由文件的属主定义其它用户对该文件的许可访问方式，其“owner-group-world”模型如下：

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 37084 2009-04-04 13:49 /usr/bin/passwd
```

系统中所有文件属主给各自文件所定义的“owner-group-world”模型的总和，即为 DAC 下访问规则库的实现。

DAC 的开发环境（相对封闭的开发社区，或大学实验室）和对软件使用环境的假设（软件没有缺陷且运行环境友好）注定了 DAC 存在着天生缺陷：**进程和文件的安全属性都基于（且总是基于）uid/euid 和 gid/egid，无论进程执行什么应用程序，其 uid/euid 都不改变**（暂不考虑 setuid 类程序），**无法将进程所执行程序的行为和可靠性（安全性）标识到进程的安全属性中，导致操作系统无法有针对性地对进程施加访问控制**。

比如，用户登录 shell 进程会创建子进程以执行 shell 的外部命令，子进程继承父进程的 uid/gid，所以无法通过 uid/gid 来区分父子进程，即无法区分用户人为的操作和通过程序执行的操作。比如，无法限制 passwd 程序只能被用户在命令行执行，而不能被属于该用户的其它进程执行。显然，“用户（登录 shell 进程）是可信的”绝对不等于“用户执行的程序（后继 fork 的子进程）也是可信的”。登录 shell 进程的行为由用户有意识地支配，而属于该用户的进程的行为则由其执行的实际程序决定。但是程序可能存在安全漏洞，一旦被攻击植入恶意代码，黑客将具有该用户在系统上的所有权力（比如恶意删除用户的文件，盗取 uid/gid 所能读取的文件的内容，执行 passwd 程序修改用户的密码）。

另外，用户进程可以执行各种应用程序，而这些程序自身的行为、对系统安全性的影响都不尽相同，比如网络类应用程序更容易受到外界病毒的攻击，相应地操作系统应该对此类应用施加更为严格的访问控制。显然无法通过进程的 uid/gid 来描述当前进程所执行的程序的安全性，因此操作系统也无法有针对性地实施访问控制。

综上所述，仅将 uid/gid 来作为进程和文件的安全属性是远远不够的。

最后，DAC 的另一个缺点是对权力的管理和划分不够细致（只有两种情况：root 或 non-root），无法进一步细分、限制 root 用户的能力。一旦 euid/egid 为 0 的进程被攻破，将危害整个系统的安全。

1.3 MAC（Mandatory Access Control）的起源

针对 DAC 的缺点，在 MAC 中不再由访问对象的属主定义不同用户对其的许可访问方式，而是由固定的规则库决定。

MAC 最初的研究由美国军方的 MLS（Multi-Level Security）应用所推动，它将访问主体和被访问对象分成不同的安全级别，严格控制信息只能从低安全级别向高安全级别流动：低安全级别的主体只能向高安全性的数据对象中追加新的数据，并且禁止读取；高安全级别的主体能够读取低安全级别的数据，并且禁止任何形式的写入（即“no read up，no write down”）。

MLS 只定位于数据保密性而并不关心数据完整性和最小权能原则（Least Privilege），以及对进程的能力进行分类。后来的 FLASK 安全系统模型着眼于解决这些不足，而 SELinux 则是 FLASK 在 Linux 内核中的实现。

1.4 SELinux 的 TE（Type Enforcement）模型

SELinux 作为 MAC 的一种实现，通过中央规则库（policy.X，二进制数据文件）给所有进程、所有文件、内核数据结构定义**各自的**安全标识（标签，label/type），明确定义被访问对象所支持的访问方式，并规定进程标签对被访问对象的合法的访问方式。在配置 SELinux 时给整个文件系统上的所有文件设置标签，在系统启动过程中 init 进程经由 selinuxfs 接口装载 policy.X 到内核空间，由内核中的 Security Server 在处理用户态系统调用时实时查询（注，这里所说的“Security Server”，即为 SELinux 内核驱动中定义的各种数据结构，比如 sidtab，policydb，AVC cache，以及在 services.c 文件中定义的各种以“security_”开头的函数）。

policy.X 的“X”为其版本号，由编译时所使用的 SELinux toolchain 及相应设置决定（toolchain 决定能够支持到的最大版本号，而 build.conf 和 semanage.conf 配置文件中的选项可分别指定以 Monolithic 方式编译和以 Modular 方式编译时的实际生成版本号。可以有意识地指定编译为较低版本格式，从而丢弃较高版本的特性，即发生 policy downgrade）。policy.X 可以由几百个子模块（称为 policy package 或 policy module，简称为 pp）链接而成，系统运行时只有特定的管理员角色（secadm_r）才能够安装、升级、删除子模块。

policy.X 给系统上所有文件定义“安全上下文”（Security Context，为包含若干域的字符串，参见下文），在安装 SELinux 时每个文件都会被“打标签” — 将文件的安全上下文保存到文件的扩展属性（Extended Attribute，xattr）中。SELinux 所使用的扩展属性的 key 为“security.selinux”（注意，只有在支持扩展属性的文件系统上才能部署 SELinux，NFS 就不支持）。

进程的标签通常又被称为进程所在的“domain”。一个进程在执行具有不同标签的程序文件时，其所处的 domain 通常（注意并不总是）会发生变化。policy.X 的开发者应该依据 Least Privilege 原则给每一个 domain 赋予其真正需要的权限，既支持应用程序合理合法的行为，同时也明确界定了应用的行为，从而能够根据应用程序自身的安全性、可信性有针对性地施加访问控制。即使运行相应程序的进程被黑，进程的行为也无法超越其所在 domain 的限制，从而将危害限制在一定的范围内。

可以把进程的 domain 想象为一个“房间”，一个房间和一个程序相对应，进程在执行程序时进入相应的房间。进入房间后进程能够执行一定的行为（为了完成该程序的既定功能），同时这个房间也是一个“牢笼”，进程只能在房间范围内活动而无法穿越墙壁，除非遵循许可的出口进入其他房间（即 Domain Transition），否则内核中的 Security Server 会严格记录进程的“撞墙”行为。policy.X 定义了不同房间的大小和形状（不同 domain 所具有的能力），在房间之间迁移的方式，什么用户能够进入什么房间等内容。

给进程定义了 domain/type、给文件定义了 label/type，policy.X 就可以明确地指定处在哪些 domain 的进程能够以何种方式访问哪些 label 的文件，比如对标记为 passwd_exec_t 的 passwd 可执行文件而言定义如下规则：

```
allow user_t passwd_exec_t : file { read getattr execute open } ;
```

这意味着：
1，处在 user_t 这个 domain 的进程能够对标签为 passwd_exec_t 的文件具有读和执行的能力。这将使得普通用户具有对 passwd 程序的执行能力；
2，没有被明确许可的其他访问方式，统统被禁止。比如 user_t 这个 domain 没有对此类文件的写权限；

注意，所有许可的访问操作，都必须用这样的 allow 规则**明确地**定义（SELinux 规则库为“白名单”，相比之下杀毒软件的病毒库为“黑名单”）。

总之，在 SELinux 中进程和文件的安全属性都不再是 uid/gid 信息，而是其 Security Context；同一个进程在执行不同的程序时，其 Security Context 通常会发生相应地改变（从而和被执行程序的安全属性、所需要的能力相对应）；进程的能力也不再由被访问的文件的属主决定，而是由固定的规则库明确地定义。

2. SELinux的概念

2.1 SELinux的“物质基础”－安全上下文（Security Context）

如上所述，SELinux访问控制规则库的“物质基础”即为进程和被访问对象的安全上下文（Security Context，简称SC），它由三部分组成：

```
user : role : type
```

注意上面的“user”为SELinux User，而非传统意义上的UNIX User。用户登录系统后，login程序根据当前规则库的定义，将UNIX User **映射**到相应SELinux User。不同SELinux User所能**扮演**的SELinux Role不同，而不同SELinux Role具有不同的能力（由规则库定义一个Role所能够**关联**的type，即生成有效的SC）。从而方便地给不同的UNIX User定义不同的能力。

对于人类用户而言，其进程SC中user和role的作用如上所述，type为该进程当前所处的domain。而对非系统用户而言，其进程SC中的user通常为system_u（代表系统资源），role通常为object_r。文件或其它内核数据结构的SC中，user通常为属主或父目录的user，role一般为object_r，type可由policy.X（明确指定的标签，继承父目录的标签，显式地由type_transition规则指定）或应用程序定义（通过写入proc/*/attr/fscreate接口）。

可以使用“id -Z”命令观察当前进程的SC，用“ls -Z”命令观察文件的SC，用“ps -Z”命令观察活跃进程的SC：

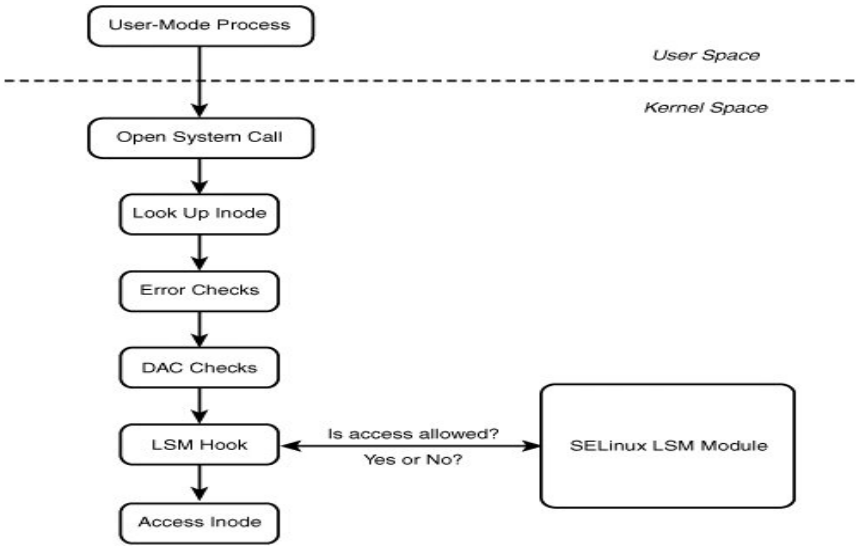
```
[root/sysadm_r/s0@~]# id -Z
root:sysadm_r:sysadm_t:s0-sl5:c0.c255
[root/sysadm_r/s0@~]# ls -Z /
-rwxr-xr-x  root root system_u:object_r:default_t:s0   README
drwxr-xr-x  root root system_u:object_r:bin_t:s0       bin
drwxr-xr-x  root root system_u:object_r:boot_t:s0      boot
drwxr-xr-x  root root system_u:object_r:device_t:s0    dev
drwxr-xr-x  root root system_u:object_r:etc_t:s0       etc
drwxr-xr-x  root root system_u:object_r:home_root_t:s0-sl5:c0.c255 home
drwxr-xr-x  root root system_u:object_r:lib_t:s0       lib
drwxr-xr-x  root root system_u:object_r:default_t:s0   man
drwxr-xr-x  root root system_u:object_r:mnt_t:s0       media
drwxr-xr-x  root root system_u:object_r:usr_t:s0       opt
dr-xr-xr-x  root root system_u:object_r:proc_t:s0      proc
drwxr-x---  root root root:object_r:user_home_dir_t:s0-sl5:c0.c255 root
drwxr-xr-x  root root system_u:object_r:bin_t:s0       sbin
drwxr-xr-x  root root system_u:object_r:security_t:s0  selinux
drwxr-xr-x  root root system_u:object_r:var_t:s0       srv
drwxr-xr-x  root root system_u:object_r:sysfs_t:s0     sys
drwxrwxrwt  root root system_u:object_r:tmp_t:s0-sl5:c0.c255 tmp
d---------  root root system_u:object_r:tmp_t:s0       tmp-inst
drwxr-xr-x  root root system_u:object_r:usr_t:s0       usr
drwxr-xr-x  root root system_u:object_r:var_t:s0       var
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ls -lZ /sbin/init /usr/sbin/sshd
-rwxr-xr-x+ l root root system_u:object_r:init_exec_t  89604 2008-04-11 21:50 /sbin/init
-rwxr-xr-x+ l root root system_u:object_r:sshd_exec_t 371588 2008-04-06 19:50 /usr/sbin/sshd
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps awZ
LABEL                                   PID  TTY STAT  TIME COMMAND
system_u:system_r:getty_t:s0-sl5:c0.c1023 1358 tty2 Ss+   0:00 /sbin/mingetty tty2
system_u:system_r:getty_t:s0-sl5:c0.c1023 1359 tty3 Ss+   0:00 /sbin/mingetty tty3
```

```
system_u:system_r:getty_t:s0-s15:c0.c1023 1360 tty4 Ss+   0:00 /sbin/mingetty tty4
system_u:system_r:getty_t:s0-s15:c0.c1023 1361 tty5 Ss+   0:00 /sbin/mingetty tty5
system_u:system_r:getty_t:s0-s15:c0.c1023 1362 tty6 Ss+   0:00 /sbin/mingetty tty6
root:sysadm_r:sysadm_t:s0-s15:c0.c1023    1373 ttyS0 Ss    0:04 -bash
root:staff_r:staff_t:s0-s15:c0.c1023      1785 pts/0 Ss     0:00 -bash
root:staff_r:newrole_t:s0-s15:c0.c1023    1917 pts/0 S    0:00 newrole -r sysadm_r -p
root:sysadm_r:sysadm_t:s0-s15:c0.c1023    1921 pts/0 S+    0:02 -/bin/bash
root:sysadm_r:sysadm_t:s0-s15:c0.c1023    2376 ttyS0 R+    0:00 ps awZ
[root/sysadm_r/s0@~]#
```

顺便提及，文件的 SC 保存在其辅存索引节点的“security.linux”扩展属性中，可以通过 C 库的 getxattr/setxattr 访问并修改；进程的 SC 保存在其 task_struct 中，可直接通过/proc/<pid>/attr/current 设置（参见 11.1 和 5.5 小节）。而其他内核数据结构的 SC 也保存在其内 security 指针所指向的数据结构中。

2.2 LSM (Linux Security Module)

LSM 的核心即在管理不同类型数据结构的子系统（比如进程管理，文件系统，inode，file，dentry，网络，IPC 等等）中加入的一组回调函数指针（LSM hooks）。它们用于确定新创建数据结构的 SC，并在访问数据对象时，就访问者 SC 和被访问对象的 SC 及访问方式，查询 Security Server 当前访问是否被 policy.X 所许可。如下图所示：



当 DAC 检查完成并许可当前访问之后、在内核执行实际的访问之前，调用 LSM 的 security_xxx 方法，询问 Security Server 当前访问是否被许可。比如 fs/open.c 的__dentry_open > security_dentry_open，它将直接调用具体安全机制所提供的相应方法：

```
int security_dentry_open(struct file *file)
{
        return security_ops->dentry_open(file);
}
```

其中 security_ops 指针在具体安全机制向 LSM 注册时指向其提供的 security_operations 方法表，对于 SELinux 而言即为 security/selinux/hooks.c 中实现的 selinux_ops 方法表（定义在 linux/security.h 中）。

LSM hooks 的使用把不同的内核子系统变成了相应的对象管理器（Object Manager）。在系统启动时初始化 Security Server，它在加载 policy.X 文件时建立相应的内核数据结构来描述 policy.X 的内容。为了提高效率，Security Server 使用 cache 缓存查询 policy.X 的结果。

LSM hooks 是内核为支持不同安全机制的具体实现所设计的上层框架，各种安全机制（比如 SELinux，root_plug，LIDS 等）在内核启动时注册自己所提供的安全检查函数表（security_operations 数据结构），原理类似于通过 VFS 支持各种不同文件系统。

LSM hooks 的设计简单高效，对内核的影响很小，且与具体安全机制的实现无关（truly generic，conceptually simple，minimally invasive，efficient），更多内容请见参考资料[7]。

2.3 Type Enforcement 的概念

SELinux 的规则库以“白名单”方式实现，即要求所有许可的操作都必须**显示地**由 allow 规则来定义，否则则被禁止。allow 规则由 4 部分组成：

```
allow subj_t obj_t : obj_class permissions;
```

其中 subj_t 为访问主体的 domain/type，obj_t 为被访问对象的 type，obj_class 为被访问对象所属的类型，permissions 为当前规则所许可的访问方式集合。比如，

```
allow user_t bin_t : file { read getattr execute open };
```

其含义为“赋予处于 user_t 这个 domain 中的进程对 file 类对象中标签为 bin_t 的对象的读、获得属性、执行、打开的权力”。

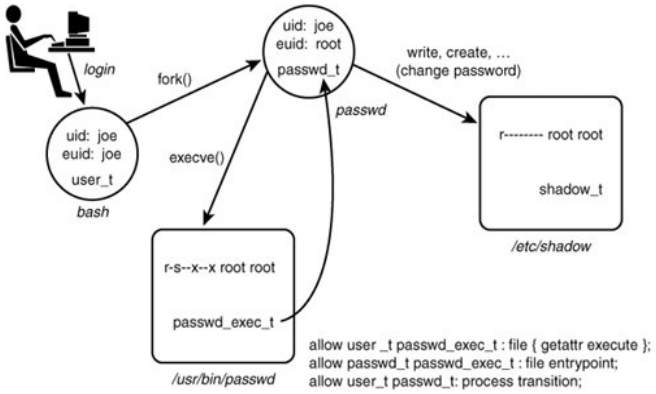
通过 obj_t 所属的 class（上例中为 file 类），即可得知该 obj_t 所支持的所有访问方式。allow 规则中指定的访问方式必须是相应 class 所支持的访问方式的子集。另外，规则中任何一个部分上都可以指定多个成份，在 subj_t/obj_t 上还可以使用属性（attribute），而由 SELinux toolchain 负责将属性展开。

policy.X 中正是通过访问者的 domain/type 和被访问对象的标签（以及被访问对象的 class 和访问方式）来决定某一访问是否被支持，即访问控制基于 type 来实现，所以称为“Type Enforcement”。

2.4 Domain Transition 的概念

进程在执行程序时通常进入和该程序相关的、新的 domain。针对该程序的可信性、安全性，SELinux 规则库给不同的 domain 定义各自的规则，清晰地定义所有合法的操作。当进程执行不同的程序时**通常**发生 domain 的切换（不同 domain 所具有的能力是不同的，由应用程序的正常行为决定）。

Domain transition 的概念类似于 setuid/setgid 机制，以用户登录并执行 passwd 程序为例：



1，登录后用户 shell 进程 SC 为 joe:user_r:user_t，处在 user_t domain 中；
2，fork 子 shell 进程，执行 passwd 程序；
3，通过 execve 系统调用执行 passwd 程序时，根据相应的 type_transition 规则，将 user_t 经由 passwd_exec_t 转变为 passwd_t。那么子 shell 进程处在 passwd_t domain 中执行。此过程类似子进程的 euid 由普通用户 joe 变成特权用户 root；
4，处在 passwd_t domain 中的子进程，有能力访问/etc/shadow 文件（标签为 shadow_t）设置新密码。相应地，只有 euid 为 root 的进程才能够修改/etc/shadow 文件；
5，子 shell 进程退出，给父进程发送 SIGCHLD 信号。父进程“reap”之以避免其成为 Zombie 状态。

由此可见，发生 domain transition 所需要的条件是：
1，声明可执行文件为相应 domain 的“接入点”： allow passwd_t passwd_exec_t:file entrypoint ；
2，许可用户进程能够执行“接入点”文件： allow user_t passwd_exec_t:file { execute ... } ；
3，许可两个 domain 之间的切换： allow user_t passwd_t:process transition ；
4，许可进程的角色和新 domain 之间的关联：role user_r types passwd_t ；（参见 2.5 小节）
5，使能 domain 切换：type_transition user_t passwd_exec_t:process passwd_t;
6，子进程能够使用从父进程继承的打开文件描述符，并向父进程发送 SIGCHLD 信号；（参见 2.6 小节）

另外，并不是所有的应用程序都要求在执行期间一定要进入新的 domain。比如/usr/bin 和/bin/下的许多可执行文件的标签都是 bin_t，包括所有的 shell 脚本，所有用户 domain（user_t/staff_t/sysadm_t 等）都对 bin_t **同时**具有{ execute execute_no_trans }权限，后者使得调用者 domain 在执行相应的程序时**不**发生 domain 切换，此时当前用户是否能够执行某个程序应该由（也只应该由）该用户的当前 domain 来决定。

否则（用户 domain 不具备成功地执行一个程序所需的全部能力，但是又希望用户能够执行该命令），就必须设计新的 domain 并赋予所必要的能力（从而支持程序的正常行为），并且使得该用户的 domain 都能够切换到这个新的 domain，从而顺利执行程序（而不是向用户原有 domain 中补充额外的新能力）。参见第 7 章。

2.5 Role 的作用

SELinux 通过 SC 中的 role 实现了“基于角色的访问控制”（RBAC — Role Based Access Control）。进程 SC 中的 role 限制了该进程能够转换到的新 type/domain 的集合。

在上一小节的例子中，为了支持从 user_t 到 passwd_t 的转换，必须显式地定义如下 role 规则：

```
role user_r types passwd_t;
```

可以将其中的“type”理解为动词，表示“和 xx 相关联”。该规则允许将 user_r 和 passwd_t 相关联，从而使得 joe:user_r:passwd_t 为一个合法的 SC。否则子 shell 通过 execve 系统调用执行 passwd 程序会失败，尽管相应的 allow 规则存在。

通过这个例子我们可以看出：通过 role 规则可以限制一个角色所能关联的合法的 type，从而达到限制该角色能力的目的。

2.6 Domain Transition 和 role 规则举例

举例：只有用户本人才能执行 passwd 程序修改密码，而用户所执行的程序无法执行 passwd 程序。

用户的登录 shell 进程的 domain 为：user_t/staff_t/sysadm_t/secadm_t/auditadm_t，在 policy.X 中只给这些 domain 调用了 usermanage_run_passwd 接口，从而使得这些 domain 获得了切换到 passwd_t 的能力。而其它 domain 的进程，比如 bin_t，由于不存在相应 allow 规则的支持，则无法转换到 passwd_t。refpolicy 源代码中相应接口的调用关系如下：

```
userdom_admin_user_template(sysadm),  或者
userdom_unpriv_user_template(secadm/auditadm/staff/user) > userdom_restricted_user_template
        >       userdom_login_user_template                             仅被上两者调用
                >       userdom_change_password_template                仅被上者调用
                        >       usermanage_run_passwd(user_t, user_r)   仅被上者调用，以 user 为例

interface(`usermanage_run_passwd',`
        gen_require(`
                type passwd_t;
        ')

        usermanage_domtrans_passwd($1)
        role $2 types passwd_t;                                         使得 user_u:user_r:passwd_t 合法
')

interface(`usermanage_domtrans_passwd',`                                许可参数 type 到 passwd_t 的转换
        gen_require(`
                type passwd_t, passwd_exec_t;
        ')

        files_search_usr($1)
        corecmd_search_bin($1)                                          使得 user_t 进程能够搜索/usr/bin/目录
        domtrans_pattern($1, passwd_exec_t, passwd_t)
')

define(`domtrans_pattern',`
        allow $1 $2:file { getattr open read execute };                 使得 user_t 能够执行 passwd_exec_t
        allow $1 $3:process transition;                                 许可从 user_t 到 passwd_t 的转换
        dontaudit $1 $3:process { noatsecure siginh rlimitinh };
        type_transition $1 $2:process $3;                               使能默认的 domain transition

        allow $3 $1:fd use;                                             passwd_t 能够使用 user_t 的打开文件
        allow $3 $1:fifo_file rw_fifo_file_perms;                       管道，并向其发送信号
        allow $3 $1:process sigchld;                                    passwd_t 向父进程 user_t 发送 sigchld 信号
')
```

相应接口调用和规则的作用，参见相应的注释。其他说明如下：
1，在 admin/usermanage.te 中通过 application_domain(passwd_t, passwd_exec_t)接口声明 passwd_t 以 passwd_exec_t 为自己的 entrypoint 文件，则上一小节中关于支持 domain transition 的 5 个条件全

部满足；
2，关于 process 类的 noatsecure 权限，以及 Auxiliary Table Secure 机制，参见后文 10.11.1 小节；
3，子进程继承父进程的打开文件描述符（如果没有设置 close_on_exec 标志的话），比如当前 terminal 的打开文件描述符被 duplicate 到子进程的 fd_set[0~2]。被打开文件的内核 file 数据结构的标签为当前进程 domain，即为 user_t。所以子进程 domain transition 到 passwd_t 后，它需要上述关于 fd 类对象的 use 能力以继续使用当前 terminal 文件；（另见 7.4.4 小节中关于 privfd 属性的认识）
4，子进程在结束时内核会向其父进程发送 SIGCHLD 信号，所以需要对 process 类对象的 sigchld 能力。

2.7 MLS 对 SC 的扩展

在需要保护政府或军方机密数据的场合，在 SELinux 的 TE 基础上应用 MLS 特性，它对 SC 的扩展如下：

```
user : role : type : sensitivity[:category list][- sensitivity[:category list]]
```

比如，

```
auditadm_u : auditadm_r : auditadm_t : s0 - s15:c0.c1023
```

含义：audit 管理员（auditadm_u）当前扮演的角色为 auditadm_r，所处的 domain 为 auditadm_t。其当前安全级别（Security Level，或简称为 SL）为 s0，最大能够达到的 SL 为 s15:c0.c1023。

SC 包含至少一个、至多两个 SL。一个 SL 由至少一个 sensitivity 和若干 category 组成。如果访问主体的 SC 有两个 SL，则减号前的为当前安全级别（current/low SL），减号后的为最高可达安全级别（clearance/high SL）。各级 sensitivity 之间为单调增关系，并要求 high SL 和 low SL 之间存在“dom”（dominate）关系。

SL 之间的四种关系为：dom，domby，eq，incomp。dom 和 incomp 的定义如下：

```
dom: (sensitivity of SL1 <= that of SL2) && (categories of SL1 subset of that of SL2)
incomp: categories of SL1 incomparable with that of SL2
```

用户可以通过“newrole -l”命令切换到 high SL 范围内的其它任何 SL。

举例：在 MLS 中 subject 必须**同时**满足如下条件才能成功访问 audit.log：

```
1, euid == 0;   (root user)             DAC rule
2, role == auditadm_r or sysadm_r;      TE rule, auditadm_t or sysadm_t could access auditd_log_t
3, SL dom s15:c0.c1023;                 MLS constraint, current SL must dom that of audit.log

[root/sysadm_r/s0@~]# date +%T
07:55:57
[root/sysadm_r/s0@~]# ls -Z /var/log/audit
ls: cannot access /var/log/audit: Permission denied
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# audhigh "ausearch -ts 07:55:57 -m avc -sv no"
Password:
----
time->Mon Feb 13 07:56:05 2012
type=SYSCALL msg=audit(1329119765.794:17): arch=40000003 sysca11=195 success=no exit=-13 a0=bfa8ccb4
a1=85f4500 a2=b7866ff4 a3=85f44f8 items=0 ppid=1407 pid=1502 auid=4294967295 uid=0 gid=0 euid=0 suid=0
fsuid=0 egid=0 sgid=0 fsgid=0 tty=ttyS0 ses=4294967295 comm="ls" exe="/bin/ls"
subj=root:sysadm_r:sysadm_t:s0-s15:c0.c1023 key=(nu11)
type=AVC msg=audit(1329119765.794:17): avc:  denied  { getattr } for  pid=1502 comm="ls"
```

```
path="/var/log/audit" dev=sda ino=49166 scontext=root:sysadm_r:sysadm_t:s0-sl5:c0.c1023
tcontext=system_u:object_r:auditd_log_t:sl5:c0.c1023 tclass=dir
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s sysadm_t -t auditd_log_t -c dir -p getattr"
Password:
Found 1 semantic av rules:
   allow sysadm_t auditd_log_t : dir { ioctl read write create getattr setattr lock relabelfrom relabelto
unlink link rename add_name remove_name reparent search rmdir open } ;
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# matchpathcon  /var/log/audit
/var/log/audit  system_u:object_r:auditd_log_t:sl5:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# newrole -l sl5 -- -c "ls -Z /var/log/audit"
Password:
ls: cannot access /var/log/audit: Permission denied
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# newrole -l sl5:c0.c1023 -- -c "ls -Z /var/log/audit"
Password:
-rw-------  root root system_u:object_r:auditd_log_t:sl5:c0.c1023 audit.log
[root/sysadm_r/s0@~]#
```

由此可见，sysadm_r 在 SL == s0 时无法访问/var/log/audit/目录，尽管 sysadm_t 具备对 auditd_log_t@dir 的 getattr 能力（注意，事先还需确定 subject/object 的标签是正确的。由于/var/log/audit 目录的标签是静态设置的，所以可以使用 matchpathcon 命令来验证。而如果一个文件是动态创建的，则必须考虑其他情况。参见 8.3 小节 AVC 错误消息的分析）。所以只能是非 TE 之外的其他约束限制了上述访问。确切地说，在 policy/mls 文件中实现的 MLS constraint 如下：

```
# the file "read" ops (note the check is dominance of the low level)
mlsconstrain { dir file lnk_file chr_file blk_file sock_file fifo_file } { read getattr execute }
        (( l1 dom l2 ) or
         (( t1 == mlsfilereadtoclr ) and ( h1 dom l2 )) or
         ( t1 == mlsfileread ) or
         ( t2 == mlstrustedobject ));
```

由于 sysadm_t 不属于 mlsfilereadtoclr 或者 mlsfileread 属性，而且 auditd_log_t 也不属于 mlstrustedobject 属性，所以在使能 MLS 时，针对 dir 类的 getattr 操作必须满足(l1 dom l2)条件。

另外，secadm_t 属于 mlsfileread 属性，所以 secadm_r 能够在 SL == s0 时访问/var/log/audit/目录：

```
[root/sysadm_r/s0@~]# seclow "seinfo --t=secadm_t -x"
Password:
   secadm_t
      privfd
      can_setenforce
      process_user_target
      can_relabelto_shadow_passwords
      can_change_object_identity
      can_relabelto_binary_policy
      mlsprocread
      mlsfileread
      userdomain
      mlsfiledowngrade
      can_setsecparam
      can_write_binary_policy
      ubac_constrained_type
      mlsfilewrite
      unpriv_userdomain
```

```
      m1sfi1eupgrade
      domain
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# sec1ow "1s -Z /var/1og/audit"
Password:
-rw-------  root root system_u:object_r:auditd_1og_t:s15:c0.c1023 audit.1og
[root/sysadm_r/s0@~]#
```

3. SELinux 的语法

3.1 Object Class and Permissions

系统中文件/资源的数量以百万或千万计，但是类型固定。为了便于定义 Object 的访问权限，把系统中所有资源划分为若干类（class），可以给同一类资源的所有实例定义一组相同的访问权限。

refpolicy 源代码 policy/flask/目录下的 security_classes 文件定义所有的 class，access_vectors 文件定义各个 class 的访问权限。比如和文件相关的 class 划分如下：

```
# file-related classes
class filesystem
class file
class dir
class fd
class lnk_file
class chr_file
class blk_file
class sock_file
class fifo_file
```

可以直接定义某个 class 的访问权限，比如：

```
class filesystem
{
     mount
     remount
     unmount
     getattr
     relabelfrom
     relabelto
     transition
     associate
     quotamod
     quotaget
}
```

也可以首先使用 common 语法定义一组访问权限，比如：

```
common file
{
     ioctl
     read
     write
     create
     getattr
     setattr
     lock
     relabelfrom
     relabelto
     append
     unlink
     link
     rename
     execute
     swapon
```

```
        quotaon                         # Allow file to be used as a quota database(quota.pp).
        mounton
    }
```

然后在定义某一类资源的访问权限时，继承某个公共的权限集合并追加私有的定义：

```
    class lnk_file
    inherits file

    class file
    inherits file
    {
        execute_no_trans
        entrypoint
        execmod
        open
    }

    class dir
    inherits file
    {
       add_name
       remove_name
       reparent
       search
       rmdir
    }
```

说明：
1，通常情况下不需要修改 security_classes 文件，除非在 Linux 内核中定义了新的数据结构类型，此时可以考虑在相应子系统中部署 LSM hooks，从而把它变成一个新的 Object Manager，并且在 SELinux 内核驱动的 classmap.h 文件以及 security_classes 文件中增加新的 class 及其权限的定义；
2，security_classes 文件中还定义了一些只在用户态 Object Manager（比如 XACE，DBUS）中所使用的类（比如 class db_database，class x_server 等），相应类的数据对象对内核不可见，所以只需在用户态定义；
3，在装载 policy.X 时执行“class mapping”操作，即将 policy.X 中定义的 class 的二进制索引值（由 toolchain 决定），和对应 class 在 SELinux 内核驱动中分配的索引值，建立映射关系。参见 10.8 小节。

3.2 Type，alias and Attribute

定义 type 的语法如下：

```
type type_name [alias alias_set] [, attribute_set] ;
```

在定义新的 type 时可以声明它的别名（alias）或同时将它加入某个属性（attribute）。

alias 用于声明一个 type 的别名，通常用于兼容老版本中的 type。比如在 system/userdomain.te 中定义 user_tty_device_t 如下：

```
type user_tty_device_t alias { staff_tty_device_t sysadm_tty_device_t secadm_tty_device_t
auditadm_tty_device_t unconfined_tty_device_t };
```

在老版本的 refpolicy 中 ssh/login 等程序在用户登录时把相应 tty 设备的标签 relabel 为 <user>_tty_device_t，但是在当前版本中它们被统一为 user_tty_device_t（参见下文）。

属性（attribute）概念有两层含义，既可以被理解为“具有一组共性的 type 的集合”，或者“这组 type 所具有的共性”。比如定义一个名为“file_type”的属性：

```
attribute file_type;
```

可以在定义某个 type 时就建立它与某个 attribute 的关联，比如：

```
type shadow_t, file_type;
```

或者稍后用 typeattribute 语法进行声明：

```
typeattribute shadow_t, file_type;
```

使用 attribute 可以在开发时有效地减少类似规则的数目（而由 toolchain 负责将其展开），并且可以“一劳永逸”地针对某种 type 指定它对某类 attribute 的能力。比如 sysadm_t 能够管理除了 shadow_t 之外的所有文件（无论文件类型或 type），这通过指定 sysadm_t 对 file_type 属性的能力实现：

```
userdom_admin_user_template(sysadm) >
        auth_manage_all_files_except_shadow(sysadm_t) >
                files_manage_all_files($1,$2 -shadow_t) >
                        manage_dirs_pattern($1, { file_type $2 }, { file_type $2 })
                        manage_files_pattern($1, { file_type $2 }, { file_type $2 })
                        manage_lnk_files_pattern($1, { file_type $2 }, { file_type $2 })
                        manage_fifo_files_pattern($1, { file_type $2 }, { file_type $2 })
                        manage_sock_files_pattern($1, { file_type $2 }, { file_type $2 })
                        ...
```

在相应的接口中通过调用上述 manage_xxx_pattern 宏使得 sysadm_t 具有除了 shadow_t 之外所有类型文件的管理能力。

使用属性的另一个好处是，此后新定义的任何类型的文件只要使用 typeattribute 规则加入 file_type 属性，就能够被 sysadm_t 所管理，而无须针对它就 sysadm_t 补充新的 allow 规则。这就象我们使用邮件列表来发信一样，由新用户自己负责订阅相应的邮件列表，而发送者无须关心当前哪些人订阅了该邮件列表，只要发送给该邮件列表那么所有订阅者都能够收到。

3.3 Access Vector Rules

此类规则是 SELinux TE 机制的基础。除了 allow 规则之外，还有 dontaudit，auditallow，neverallow 等规则。

一条 allow 规则允许含有多种 type，并允许混用 type 和 attribute，比如：

```
allow {user_t domain} {bin_t sbin_t file_type} : file execute ;
allow user_t bin_t : {file dir} {read getattr} ;
```

也可以从某个属性中使用减号以排除其中的一种 type，比如：

```
allow domain {exec_type -sbin_t} : file execute;
```

SELinux 产生的 AVC Denied Message 中含有关于一次事件的许多与安全相关的信息（发生时间，被访问的文件的路径，subj_t，obj_t，访问是否成功等等）。默认情况下只对禁止的操作生成相应的 AVC

Denied Message（显然这是处于效率的考虑－通常允许的操作个数远远大于禁止的操作个数），但是可以用 dontaudit 和 auditallow 规则修改上述默认行为。使用 dontaudit 规则可以屏蔽那些已知的、可预期的失败操作的消息，因为有些程序在已经能够实现其既定功能的情况下，可能会有额外的、并不一定必需的操作，比如：

```
dontaudit http_t etc_t : dir search;
```

那么当 http_t domain 不具有对/etc/目录的搜索权限时，上述命令使得 SELinux 不就失败操作生成记录信息，从而使得日志看起来更简洁。

使用 allowaudit 规则可以就某些成功的关键操作生成相应的消息，比如监控对/etc/shadow 文件的修改：

```
allowaudit domain shadow_t : file { write append };
```

或者监控 secadm_r 通过 semanage 工具修改当前 policy 的特性：

```
allowaudit semanage_t semanage_store_t : file manage_file_perms;
```

最后，使用 neverallow 规则可以明确地禁止某些操作，比如明确禁止用户直接修改/etc/shadow 文件：

```
neverallow user_t shadow_t : file write;
```

或者禁止进程转换到非进程 domain：（进程只能转换到进程的 domain，而不是文件的 type）

```
neverallow domain ~domain : process transition;
```

如果在 policy 中某处不小心添加了与已经存在的 neverallow 规则相违背的 allow 规则，则会导致编译错误。

3.4 Type Transition Rule

3.4.1 type_transition 规则介绍

type_transition 规则用于自动化 Domain Transition 或者确定新创建对象的标签，以重载其默认的、从父目录（containing directory）所继承的标签。无论哪种情况，规则中 obj_t 域都描述被访问对象的标签，比如被执行的文件的标签，或者新对象的默认标签（即父目录的标签）；class 域在描述 domain transition 时，为进程类 process；而在描述新创建对象的标签时，为新对象的类（注意不是父目录的类型 dir）。

通常情况下新创建文件的标签和其父目录的标签一致，但是可以使用 type_transition 规则为其指定特定的标签。在 kernel/files.if 中定义了许多 files_xxxx_filetrans 接口，比如 files_root_filetrans 定义如下：

```
interface(`files_root_filetrans',`
        gen_require(`
                type root_t;
        ')

        filetrans_pattern($1, root_t, $2, $3)
')
```

filetrans_pattern 接口及其调用的 rw_dir_perms 宏定义如下：

```
define(`filetrans_pattern',`
        allow $1 $2:dir rw_dir_perms;
        type_transition $1 $2:$4 $3;
')

define(`rw_dir_perms', `{ open read getattr lock search ioctl add_name remove_name write }')
```

比如，在 admin/quota.te 中对 quota_t 调用了该接口：

```
files_root_filetrans(quota_t, quota_db_t, file)
```

那么，quota_t 将通过该接口获得如下能力：

```
allow quota_t root_t:dir rw_dir_perms;
type_transition quota_t root_t:file quota_db_t;
```

上述 allow 规则使得 quota_t 具有对 root_t 目录的读写权限，type_transition 规则使得 quota_t 在 root_t:dir 下创建的 file 类文件的标签由默认的 root_t 转换为 quota_db_t。

sysadm_r 在运行 quotacheck 程序时进入 quota_t，如果在挂载根文件系统时使能 usrquota 选项，则 quotacheck 将在根目录下创建 aquota.user 文件。根目录“/”的标签为 root_t，所以新创建的文件的标签默认为 root_t，但是由于上述 type_transition 规则的存在，quotacheck 创建的 aquota.user 文件的标签实际为 quota_db_t。

这样做是必须的，否则 quota 程序在运行时创建 aquota.user 文件后将没有权限访问它：因为它的标签默认为 root_t，但是 quota_t 只对 quota_db_t 标签的文件有管理权限，而对 root_t 类型则没有。这是因为在 quota.te 中只定义了如下相关规则：

```
allow quota_t quota_db_t:file { manage_file_perms quotaon };
```

另外，在 quota.fc 中为根目录下的 aquota.user 文件也定义了默认的标签：

```
/a?quota\.(user|group)  --      gen_context(system_u:object_r:quota_db_t,s0)
```

这样做是为了给“已安装的”（而非运行时动态创建的）配置文件打上正确的标签。

总结：
1，.fc 文件负责定义一个应用程序在**安装时静态**创建的所有文件的 SC，而 .te 中的 files_xxxx_filetrans 接口定义相应 domain 在**运行时动态**创建的文件的 SC。

所以在开发新的 pp 时，如果相应程序在运行时会创建新文件，那么通常需要实现相应的 files_xxxx_filetrans 接口，并在 .fc 中定义已有文件的标签。比如由于版本或使用上的差异，导致使能 quota 的文件系统的挂载点在/mnt 或者/media，而不是当前 quota.fc 中所默认的那些目录（/，/home，/boot，/var，/var/spool，/etc），由于/mnt 和/media 目录的标签为 mnt_t，所以需要在 quota.te 中调用新接口 files_mnt_filetrans(quota_t)使得 quota_t 具备在 mnt_t 目录下创建 quota_db_t 文件的能力，同时在 quota.fc 中补充这些目录下 quota file 的默认标签：

```
/mnt(/.*)?/a?quota\.(user|group) -- gen_context(system_u:object_r:quota_db_t,s0)
/media(/.*)?/a?quota\.(user|group) -- gen_context(system_u:object_r:quota_db_t,s0)
```

2，根据 type_transition 规则还能够反推出某个文件的创建者 domain。

一个文件的 type 如果和其所在的父目录不同，则可能（注意，并不一定。文件标签的决定方法参见 7.1 小节）存在相应的 type_transition 规则指定新创建文件的标签，或者相应程序为 SELinux-aware 并调用了 setfscreatecon 函数（参见 10.6 小节）。比如在分析/var/spool/cron/root 文件的创建者时，发现它并没有继承父目录的标签，进而查找是否存在相应的 type_transition 规则：

```
[root/secadm_r/s0@QtCao contexts]# matchpathcon /var/spool/cron/root
/var/spool/cron/root    system_u:object_r:user_cron_spool_t:s0
[root/secadm_r/s0@QtCao contexts]# matchpathcon /var/spool/cron/
/var/spool/cron system_u:object_r:cron_spool_t:s0
[root/secadm_r/s0@QtCao contexts]#
[root/secadm_r/s0@QtCao contexts]# sesearch -SCT -t cron_spool_t -c file | grep  user_cron_spool_t
   type_transition admin_crontab_t cron_spool_t : file user_cron_spool_t;
   type_transition crontab_t cron_spool_t : file user_cron_spool_t;
[root/secadm_r/s0@QtCao contexts]#
```

由此可见，是 admin_crontab_t 或者 crontab_t 创建了该文件。（sysadm_r 在调用 crontab 时进入前者，而其他非特权用户在调用 crontab 时进入后者。）

3.4.2 type_transition 规则的新特性

type_transition 规则可以确定当处在某个特定 domain 的进程在相应 type 的目录下创建新对象时，新对象的标签。如果这个 domain 的进程在同一个目录下需要创建相同类型的**不同文件**（具有不同的文件名），而且要求它们具有不同的标签，显然现有的 type_transition 规则就不再适用了，因为它无法根据文件名区分不同的对象。

针对上述缺点，最新的 SELinux toolchain 和内核驱动已经给 type_transition 规则添加了第 5 个参数：“文件名”，即在原有 type_transition 规则基础上新创建对象的文件名必须和第 5 个域相匹配，该规则才生效。比如，sysadm_t 在/root/目录下创建.ssh/目录时，希望它的标签为 ssh_home_t；而创建 public_html/目录时，希望它的标签为 http_content_t，则可以使用如下接口实现：

```
filetrans_root_pattern(sysadm_t, dir, ssh_home_t, ".ssh")
filetrans_root_pattern(sysadm_t, dir, http_content_t, "public_html")
```

展开后相应的规则为：

```
type_transition sysadm_t root_t : dir ssh_home_t .ssh;
type_transition sysadm_t root_t : dir http_content_t public_html;
```

如果没有“区分文件名”的支持，则上述任何 type_transition 规则都不允许存在（否则所有新创建对象都被打成同一个标签），所以要求 sysadm 在创建相应的目录后，必须运行 chcon 命令显式地修改相应的标签或者运行 restorecon 修复标签（前提：这些目录的完整路径和标签必须已经在 file_contexts 文件中定义）。

而有了“区分文件名”的支持，就可以使用 type_transition 规则自动化给新对象设置标签的工作，从而避免了手工设置/修复标签的步骤，进一步减少使用中的错误。

3.5 神奇的 type_change 规则

3.5.1 type_change 规则的作用

当用户在某种 terminal 上登录时，login/remote/sshd 程序通过 pam_selinux.so 将当前 terminal 的标签 relabel 为和用户的角色相对应的新标签；用户登录后，如果使用 newrole 切换到新的角色，则 newrole 程序将再次 relabel 当前 terminal 设备，以和新角色相适应。

各种 terminal 设备的标签变化如下：

设备名	默认标签	用户登录后的新标签
/dev/console	console_device_t	user_tty_device_t
/dev/tty*	tty_device_t	user_tty_device_t
/dev/pts/*	devpts_t	user_devpts_t

type_change 规则即用于指定 terminal 的新标签。比如在 userdom_base_user_template 模板以如下方式调用 term_user_tty 接口：

```
term_user_tty($1_t, user_tty_device_t) >
        term_tty($2)                                    # 将 user_tty_device_t 加入 ttynode/serial_device 属性
        type_change $1 tty_device_t:chr_file $2;
```

其中$1 为各种用户 domain 的前缀（比如 user/sysadm/root 等），该 type_change 规则的意思是：当用户在 chr_file 类的 tty_device_t 设备上登录时，将该设备的标签 relabel 为 user_tty_device_t。

由于所有角色类型的用户都要通过 userdom_base_user_template 模板创建，所以这里为所有角色类型用户都定义了相应的 type_change 规则，而且无论登录用户的角色如何，tty 设备总是被 relabel 为 user_tty_device_t。值得一提的时，在使用较旧版本 refpolicy 的机器上使用不同用户从/dev/ttyN 登录后（Ctrl+Alt+FN），各个 tty 的标签如下：

```
# ls -Z /dev/tty[0-4]
system_u:object_r:tty_device_t                  /dev/tty0       # no user logs in on this device
unconfined_u:object_r:unconfined_tty_device_t   /dev/tty1       # cao in unconfined_r
staff_u:object_r:staff_tty_device_t             /dev/tty2       # harry in staff_r
staff_u:object_r:sysadm_tty_device_t            /dev/tty3       # harry in sysadm_r
system_u:object_r:tty_device_t                  /dev/tty4       # no user logs in on this device
```

没有用户登录之前，/dev/tty*设备节点的标签为 tty_device_t。而用户登录后，则相应 tty*设备节点根据登录用户的类型被 relabel 为相应的新标签。

注意在较新版本的 refpolicy 上（2.20091117 之后）取消了相关的“per-role template”：无论何种角色用户登录 tty 设备的标签都将被修改为 user_tty_device_t。后者为之前各种<role>_tty_device_t 的别名，以实现兼容性：

```
type user_tty_device_t alias { staff_tty_device_t sysadm_tty_device_t secadm_tty_device_t
auditadm_tty_device_t unconfined_tty_device_t };
```

3.5.2 Relabel terminal 所需的权限

为了重新给 terminal 打标签，除了上述 type_change 规则之外，还需要如下支持：
1，登录程序的 domain，比如 local_login_t，需要具备对原始标签 tty_device_t 的 relabelfrom 能力，即有能力改变原始标签。这是通过调用类似 term_relabel_unallocated_ttys 接口实现的，比如：

```
term_relabel_unallocated_ttys(local_login_t):
        allow $1 tty_device_t:chr_file { relabelfrom relabelto };
```

2，登录程序的 domain 还需要具备对新标签 user_tty_device_t 的 relabelto 能力，即有能力使用新标签。这是通过调用类似 term_relabel_all_ttys 接口实现的，比如：

```
term_relabel_all_ttys(local_login_t):
        allow $1 ttynode:chr_file { relabelfrom relabelto };
```

注意：
1）上一小节中通过 term_user_tty > term_tty 中已经将 user_tty_device_t 加入了 ttynode 属性。
2）赋予 local_login_t 对 ttynode 属性的 relabelfrom 能力和对 tty_device_t 的 relabelto 能力是为了支持在用户退出登录时恢复相应 tty 设备的默认标签。

3，登录程序的 domain 需要具备对 security_t:security 对象的 compute_relabel 能力，从而访问/selinux/relabel 文件。这是通过调用 selinux_compute_relabel_context 接口实现的。

login/remote/sshd 登录程序在其 PAM 配置文件中指定在 session 阶段的后期使用“pam_selinux.so open”， pam_selinux.c 中以如下方式调用 libselinux 库提供的 security_compute_relabel 函数：

```
pam_sm_open_session > security_label_tty > security_compute_relabel
```

该函数将用户 domain 和当前 terminal 的标签字符串写入/selinux/relabel 文件，然后以阻塞方式读取该文件返回 terminal 的新标签。由 selinuxfs 相应驱动查询 policy.X 中相应的 type_change 规则（参见 10.4.2 小节）。

由此可见，由 policy.X 来提供 terminal 设备的新标签（策略），而用户态应用程序(login > pam_selinux.so > libselinux API)只负责实现查询 policy.X 并应用其返回结果的机制，从而避免把 type_change 策略硬编码到用户态应用程序中。**由此可见，policy.X 提供策略，而内核 SELinux 驱动和用户态 SELinux-aware 应用程序都只实现查询或执行策略的机制。**

3.5.3 type_transition 和 type_change 规则的比较

1，type_transition 规则用于实现 domain transition 和 new object labeling，在应用程序的 pp 中定义，在 SELinux 内核驱动中实施（查询 policy.X，然后设置相应数据结构的 sid）；

2，type_change 规则用于修改当前 terminal 的标签，也在应用程序的 pp 中定义，但是在相应 SELinux-aware 应用程序中实施：在用户态源代码中通过调用 security_compute_relabel 函数查询 policy.X，并最终通过类似 fsetxattr 的函数设置 terminal 设备的新标签。

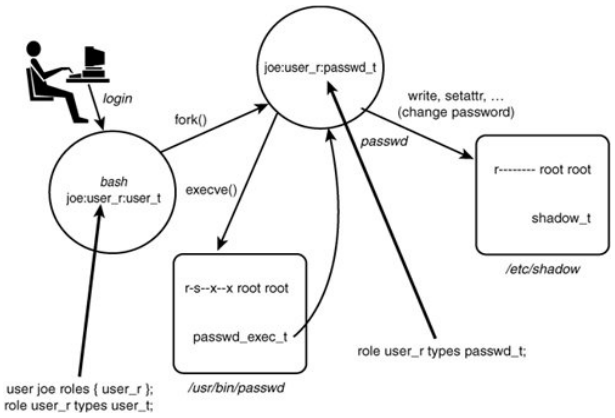
3.6 RBAC

3.6.1 基于角色的访问控制

SELinux 并不直接建立 SELinux User 和 domain 之间的联系，而是通过角色作为桥梁。此举好处如下：
1，降低 policy 复杂度：可能有上百个用户和上千种 domain/type，但是不同用户所扮演的不同角色只有有限几个；role 作为 user 和 type 之间的“中间层”，便于限制 user 的能力；
2，给不同用户赋予不同的优先级：用户通过扮演某种角色才能获得相应角色的能力（允许和某些 type 相关联），特权 type 只能和特权 role 向关联，而特权 role 只能由特权用户来扮演；

任何用户登录系统后都至少有一个默认扮演的角色。如果存在多个角色则可以在登录时选择，或者登录后用“newrole -r”切换。

RBAC 有两个规则：user declaration statement（user 规则)和 role declaration statement（role 规则)。还是以用户在命令行执行 passwd 程序为例，从 RBAC 角度看所必须的规则及其含义如下：




```
1. user joe roles user_r;                    # 用户 joe 能够扮演角色 user_r;
2. role user_r types user_t;                 # 角色 user_r 能够和 user_t 相关联;
3. role user_r types passwd_t;               # 角色 user_r 能够和 passwd_t 相关联;
```

另外 contexts/default_type 文件定义了某种角色所对应的默认 domain：

```
[root/sysadm_r/s0@~]# cat /etc/selinux/refpolicy-mls/contexts/default_type
auditadm_r:auditadm_t
secadm_r:secadm_t
sysadm_r:sysadm_t
staff_r:staff_t
unconfined_r:unconfined_t
user_r:user_t
[root/sysadm_r/s0@~]#
```

当 joe 用户（UNIX User）登录时，他被映射到 SELinux User “joe”，只能扮演一种角色 user_r。由于 user_r 角色能够和 user_t 相关联，由 default_type 文件可见 user_r 默认对应 user_t，所以 joe 用户登录后的 SC 为“joe:user_r:user_t”。正是由于 user_r 角色能够和 passwd_t 相关联，这才才是一个合法的 SC，从 user_t 到 passwd_t 的切换才能如期完成。

而对被访问的数据对象而言，其 SC 中的 user 和 role 的作用不是很大，role 通常是 object_r，而 user 为其创建者或父目录的 user（通常为 system_u，代表系统资源或后台进程）。比如：

```
[root/sysadm_r/s0@~]# ls -Z /usr/bin/passwd /etc/shadow
-r--------  root root system_u:object_r:shadow_t:s0     /etc/shadow
-rwsr-xr-x  root root system_u:object_r:passwd_exec_t:s0 /usr/bin/passwd
[root/sysadm_r/s0@~]#
```

policy.X 目前只对 process 和各种 socket 类实现了有意义的 role，而绝大部分数据对象的 role 仍是 object_r，尚未实现完整的 RBAC 机制。当前 SELinux 内核驱动和 policy.X 的实现给 object_r 开了许多“后门”（即没有实施 MAC 控制）：如果一个安全上下文的 role 为 object_r，则在 SELinux 内核函数 policydb_context_isvalid 以及 mls_context_isvalid 中，不检查 role-types，user-role 关系，以及 user-range 关系。

3.6.2 RBAC 机制的其他规则（Revisited）

除了上一小节所述的 user 规则和 role 规则之外，RBAC 相关其他语法如下：

1，allow 规则，许可角色之间的转换（单向）：

```
allow staff_r {sysadm_r secadm_r auditadm_r};
```

含义：允许从 staff_r 角色切换到 sysadm_r/secadm_r/auditadm_r 中的任一角色。注意，如需从其它角色切换到 staff_r，则还需要反向的 allow 规则，比如：

```
allow {sysadm_r secadm_r auditadm_r} staff_r;
```

如果没有角色的 allow 规则，则不允许使用“newrole -r”在多个角色之间切换。

2，dominance 规则，定义角色之间的继承关系：

```
dominance role rulea11_r {
        role sysadm_r;
        role secadm_r;
        role auditadm_r;
} ;
```

含义：rulea11_r 角色继承所有 dominated 角色的能力（rulea11_r 角色所能关联的 type 的集合，为三个子角色相应集合的超集），并且可以切换到任一 dominated 角色。

注意，role-dominance 语法已经过时，禁止被使用。它不但语义不清晰，而且在使用上也有严重的局限性：在 role-domiance 语法之后，如果再给 dominated role 建立和新 type 的关联，则这种关联关系无法再向上传到给 dominating role 了（除非在 role-types 规则中检查 role dominance 关系并向上传导）。

3，role 属性的使用

目前笔者开发的 role-attribute 规则（即 role attribute）已经被社区采纳（20110726 之后的 toolchain+refpolicy），能够在模块 link 阶段才处理 role-attribute 关系，在 expand 阶段将 role 属性的能力散播给所有从属的普通 role，完整地解决的 role-dominance 的所有问题。

可以使用 attribute_role 规则定义一个 role 属性。比如在 system/selinuxutil.te 文件中定义了一个 role 属性：

```
attribute_role newrole_roles;
```

首先它能够和 newrole_t 相关联：

```
role newrole_roles types newrole_t;
```

设计 newrole_roles 的目的是使得它和能够从 newrole_t 迁移到的所有其他 domain（比如 chkpwd_t，updpwd_t）相关联，所以在就 newrole_t 调用 auth_run_xxx_passwd 接口时，使用 newrole_roles：

```
auth_run_chk_passwd(newrole_t, newrole_roles)
auth_run_upd_passwd(newrole_t, newrole_roles)
```

在相应接口中，使得 newrole_t 能够迁移到新的 domain，并且使用 role-types 规则使得 newrole_roles

属性能够和新的 domain 相关联：

```
interface(`auth_run_chk_passwd',`
        gen_require(`
                type chkpwd_t;
        ')

        auth_domtrans_chk_passwd($1)
        role $2 types chkpwd_t;
')
```

最后，所有类型的用户都通过 seutil_run_newrole 接口切换到 newrole_t：

```
userdom_admin_user_template
userdom_unpriv_user_template
        > userdom_common_user_template
                > seutil_run_newrole($1_t, $1_r)

interface(`seutil_run_newrole',`
        gen_require(`
                attribute_role newrole_roles;
        ')

        seutil_domtrans_newrole($1)
        roleattribute $2 newrole_roles;
')
```

注意在该接口中使用 roleattribute 规则，将用户的角色（user_r/sysadm_r 等）加入了 newrole_roles 属性，从而使得各种用户在执行 newrole 期间，用户的角色能够顺利地和从 newrole_t 到达的其它 domain 相关联。

笔者给 SELinux toolchain 开发的 role-attribute 特性是为了解决“chain of run interface”的问题，role 属性的典型用例详见 9.4 小节。

3.6.3 RBAC 语法举例

1，refpolicy 源代码中 SELinux user 的定义方法。

在 policy/support/misc_macros.spt 中定义 gen_user 宏如下：

```
########################################
#
# gen_user(username, prefix, role_set, mls_default1evel, mls_range, [mcs_categories])
#
define(`gen_user',`dn1
ifdef(`users_extra',`dn1
ife1se(`$2',,,`user $1 prefix $2;')
',`dn1
user $1 roles { $3 }`'ifdef(`enable_mls', ` level $4 range $5')`'ifdef(`enable_mcs',` level s0 range
s0`'ife1se(`$6',,,` - s0:$6)');
')dn1
')
```

在 policy/users 文件中调用该宏定义默认的 SELinux User，比如

```
gen_user(user_u, user, user_r, s0, s0)
```

展开为：

```
user_u:user_r:user_t:s0-s0
```

而下述调用：

```
gen_user(staff_u, staff, staff_r sysadm_r ifdef(`enable_mls',`secadm_r auditadm_r'), s0, s0 -
mls_systemhigh, mcs_allcats)
```

在使能 MLS 特性时展开为：

```
staff_u:{staff_r sysadm_r secadm_r auditadm_r}:s0-s15:c0.c255
```

其中 enable_mls/enable_mcs 变量在 Makefile 中根据 TYPE 变量定义：

```
# enable MLS if requested.
ifeq "$(TYPE)" "mls"
        M4PARAM += -D enable_mls
        CHECKPOLICY += -M
        CHECKMODULE += -M
        gennetfilter += -m
endif
```

2，semanage 和 RBAC 相关的操作。

sysadm_r 可以使用“semanage login/user -l”命令观察当前 UNIX 用户到 SELinux 用户的映射关系，以及 SELinux 用户和角色的关联，比如：

```
[root/sysadm_r/s0@~]# semanage login -l

Login Name                SELinux User              MLS/MCS Range

__default__               user_u                    s0
root                      root                      s0-s15:c0.c1023
system_u                  system_u                  s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# semanage user -l

                Labeling   MLS/       MLS/
SELinux User    Prefix     MCS Level  MCS Range                      SELinux Roles

root            sysadm     s0         s0-s15:c0.c1023                auditadm_r staff_r secadm_r sysadm_r
staff_u         staff      s0         s0-s15:c0.c1023                auditadm_r staff_r secadm_r sysadm_r
sysadm_u        sysadm     s0         s0-s15:c0.c1023                sysadm_r
system_u        user       s0         s0-s15:c0.c1023                system_r
unconfined_u    unconfined s0         s0-s15:c0.c1023                unconfined_r
user_u          user       s0         s0                             user_r
[root/sysadm_r/s0@~]#
```

而 secadm_r 可以使用“semanage login/user -a/-m”添加或修改当前 UNIX 用户到 SELinux 用户的映射关系，以及 SELinux 用户和角色的关联，参见附录。

3.7 Constraints and MLS constraints

3.7.1 SELinux 的 constrain 语法

SELinux 内核驱动在判定当前操作是否合法时经过如下步骤:
1，根据该 obj_t 所属 class 所支持的访问方式，以及所有相关的 allow 规则，创建合法的相关 subj_t 对它的访问方式的位图;
2，从该位图中去掉那些被 constraints 所禁止的访问方式;
3，根据最终的位图决定当前 subj_t 是否能够访问;

由此可见，设计 constraints 的目的就是凌驾于任何可能的 allow 规则之上，限制整个规则库（相关源代码参见 10.3 小节中 security_compute_av > context_struct_compute_av 函数分析）。

constrain 语法由三部分组成:

```
constrain class_set perm_set expression;
```

其中 class_set 说明当前 constraint 所作用的 class，perm_set 说明该 class 的哪些访问权限受到限制，expression 为布尔表达式，即限制的具体方式。Expression 中的运算符和操作符如下:

```
u1, r1, t1 — source user, role and tyep;
u2, r2, t2 — target user, role and type;
==, !=  — set member or equivalent, or the contrary;
eq, dom, domby, incomp
```

可以针对 SC 中三个元素的任意组合设置限制，比如:

```
constrain process transition (u1 == u2 and r1 == r2);
```

即要求进程在发生 domain transition 前后，SC 中用户和角色必须保持一致。但有例外，在 login/ssh 程序在用户登录时需要改变当前 SC 的 user 和 role，比如 pam_selinux.so 将 login 进程所创建的 shell 进程的 SC，即从父进程 login 的 SC:

```
system_u:system_r:local_login_t
```

变成和当前登录用户相应的 SC:

```
root:sysadm_r:sysadm_t
```

或者，newrole 程序可以改变当前 SC 中的角色，比如由 staff_r 角色:

```
staff_u:staff_r:staff_t
```

变成 sysadm_r 角色:

```
staff_u:sysadm_r:sysadm_t
```

为了支持上述 SC 中 user 和 role 的变化，需要修改相应 constraint 限制条件，增加例外条件:

```
constrain process transition (u1 == u2 or t1 == privuser);
constrain process transition (r1 == r2 or t1 == privrole);
```

另外，登录服务 login/sshd 的 domain（local_login_t 和 ssh_t）都同时属于 privuser 和 privrole 这两个属性，所以能够在用户登录时将 login/sshd 所创建的 shell 子进程的 SC，从父进程的 SC 改变为和当前用户相应的 SC。

而 newrole_t 在 privrole 属性中，当父进程 fork 子进程执行 newrole 程序时，父进程的 SC 仍然为 staff_u:staff_r:staff_t，而子进程在执行 newrole 程序期间 SC 为 staff_u:staff_r:newrole_t，由于 newrole_t 属于 privrole 属性，所以子进程 SC 中的角色能够由 staff_r 切换为 sysadm_r。（注意，refpolicy 源代码中相应 constrain 规则的实现并不是这样的，但原理相同。参见 policy/constraints 文件。）

3.7.2 MLS 的 mlsconstrain 语法

mlsconstrain 规则的语法结构和 constrain 的相同：

```
mlsconstrain class_set perm_set expression;
```

只不过在 expression 中增加了 low/high security level 操作数：

```
ul, rl, tl, ll, hl — source user, role and type, low security level, high security level;
u2, r2, t2, l2, h2 — target user, role and type, low security level, high security level;
```

policy.X 中所有 MLS 限制条件都集中在 policy/mls 文件中，比如“no read up”概念可实现为：

```
mlsconstrain { dir file lnk_file chr_file blk_file sock_file fifo_file } { read getattr execute }
        ((l1 eq l2) or ( l1 dom l2 ) or (t1 == mlsfileread)) ;
```

即只允许“read equal/down”，或者当前 subj_t 属于 mlsfileread 属性。

MLS 的“no write down”概念可实现为：

```
mlsconstrain { file lnk_file fifo_file dir chr_file blk_file sock_file } { write create setattr relabelfrom
append unlink link rename mounton }
        (( l1 eq l2 ) or (l1 domby l2) or (t1 == mlsfilewrite)) ;
```

即只允许“write equal/up”，或者当前 subj_t 属于 mlsfilewrite 属性。

所有 MLS 属性都定义在 policy/modules/kernel/mls.te 中，并且在 mls.if 中定义了许多接口，使得调用者 domain 能够绕过 MLS 约束的限制，比如：

```
interface(`mls_file_read_all_levels',`
        gen_require(`
                attribute mlsfileread;
        ')
       typeattribute $1 mlsfileread;
')
```

该接口使得调用者 domain 加入 mlsfileread 属性，从而不受“no read up”的约束。很多与系统管理相关的 domain，比如 semanage_t，setfiles_t，newrole_t，udev_t，secadm_t 都调用了类似的接口。

值得一提的是，refpolicy-2.20091117 的实现中 MLS 的“低安全级别的 subject 只能向高安全级别的 object **追加**数据”的原则并没有实现。在各种系统管理员角色中只有 secadm_t 属于 mlsfilewrite 和 mlsfileread 属性，因此可以旁路 MLS 限制；而 sysadm_r 无法向高安全级别的文件中追加数据。

3.8 Boolean, Tunable and Optional Policy (Revisited)

应用程序实际的使用情况可能和相应 pp 开发者所假设的不一致，相应 pp 可能需要支持应用程序某些额外的行为（比如访问 NFS 文件系统），此时可以通过设置预定义的 boolean 来改变 policy.X 在**运行时**的部分规则，而无须重新修改、编译、装载 policy。

可以通过布尔变量以及 if-ifelse 语法改变 policy 的内容，如同 C 程序中的 if 语句。可以用 bool 语法或者 gen_bool 宏定义一个 boolean，比如：

```
bool secure_mode false;                          # 或 gen_bool(secure_mode,false)
```

这里定义了一个名为“secure_mode”的 boolean，并初始化为 false。

sysadm_r（或 secadm_r 在 MLS 下）可以通过改变布尔变量的值来“微调”当前 policy.X 的规则：由 boolean 变量的值决定其所控制的规则块是否生效。比如在 selinuxutil.te 中存在如下代码片段：

```
# if secure mode is enabled, then newrole
# can only transition to unprivileged users
if(secure_mode) {
        userdom_spec_domtrans_unpriv_users(newrole_t)
} else {
        userdom_spec_domtrans_all_users(newrole_t)
}
```

那么可以在运行时通过 setsebool 或者“semanage boolean”命令来改变 secure_mode 的值，从而决定实际生效的规则。

类似地，可以使用 tunable 语法或者 gen_tunable 宏来定义一个 tunable：

```
tunable allow_execheap false;          # 或 gen_tunable(allow_execheap,false)
```

从后文可知，SELinux toolchain 中使用相同的数据结构（struct cond_bool_datum_t）来描述 boolean 和 tunable 变量，也使用相同的数据结构（struct cond_node_t）来描述它们所控制的规则块。boolean 和 tunable 惟一的区别是：在编译时即根据 tunable 的值决定哪个规则块生效，而只将生效的部分最终写入 policy.X，而无效的部分直接被丢弃；对于 boolean，其控制的两个规则块都被写入 policy.X，从而能在运行时“切换”实际生效的规则。

所以，只有真正需要在运行时调整的规则块，才使用 boolean 进行控制。系统管理员可以根据 policy.X 所部署的实际环境的需要，在编译时即决定 tunable 的值，比如如果不需要支持从 console 登录（只需要支持从 tty 设备登录），则直接将 console_login 设置为 false。

通过 tunable 来关闭对不需要的特性的支持，能够显著降低 policy.X 的体积，参见 9.5 小节。

（顺便提及，截止目前 cjp 仍然没得及在 refpolicy 的 gen_tunable 宏中使用 tunable 语法来定义 tunable，而仍然使用老的 bool 语法，导致 tunable 和 boolean 实际上并没有分开，尽管 toolchain 所有功能已经就位。所以下文仍然保留 bool 定义方式。）

举例：user_ping 变量可以控制除 sysadm_r 之外的角色是否有能力执行 ping/traceroute 程序。

在 admin/netutils.te 中使用 gen_tunable()宏定义 user_ping 布尔变量：

```
gen_tunable(user_ping, false)
```

该宏定义在 support/loadable_module.spt 中：

```
define(`gen_tunable',`
        bool $1 dflt_or_overr(`$1'_conf,$2);
')

define(`tunable_policy',`
        gen_require(`
                declare_required_tunables(`$1')
        ')
        if (`$1') {
                $2
        ifelse(`$3',``,`',`} else {
                $3
        ')}
')
```

同文件中定义的 tunable_policy 宏，展开后即为使用相应 tunable 变量的 if-ifelse 结构。

赋予运行 ping 程序能力的两个接口在 admin/netutils.if 中定义：

```
interface(`netutils_run_ping_cond',`
        gen_require(`
                type ping_t;
        ')

        role $2 types ping_t;

        tunable_policy(`user_ping',`
                netutils_domtrans_ping($1)
        ')
')

interface(`netutils_run_ping',`
        gen_require(`
                type ping_t;
        ')

        netutils_domtrans_ping($1)
        role $2 types ping_t;
')
```

它们的惟一区别就是前者使用 user_ping 变量控制调用者 type 是否能够切换到 ping_t。

在 system/userdomain.if 中定义的模板 userdom_unpriv_user_template 用于定义非 sysadm_r 类角色（user_r/staff_r/secadm_r/auditadm_r），它会调用上述 netutils_run_ping_cond 接口：

```
userdom_unpriv_user_template:
        optional_policy(`
                netutils_run_ping_cond($1_t,$1_r)
                netutils_run_traceroute_cond($1_t,$1_r)
        ')
        ...
```

其中 optional_policy 使得 userdomain.pp 和 netutils.pp 没有必须的依赖关系－如果 netutils.pp 尚未安装，则 userdomain.pp 安装时也不会失败。

对于 sysadm_r，在 roles/sysadm.te 中直接调用 netutils_run_ping 接口，使得 sysadm_r 能够和 ping_t 关联，并且 sysadm_t 能够转换到 ping_t：

```
optional_policy(`
        netutils_run(sysadm_t, sysadm_r)
        netutils_run_ping(sysadm_t, sysadm_r)
        netutils_run_traceroute(sysadm_t, sysadm_r)
')
```

由于 user_ping 被默认初始化为 false，所以除了 sysadm_r 之外的其它角色默认不具备运行 ping 程序的能力。secadm_r 可以在控制台使用 getsebool/setsebool/togglesebool 命令来观察、改变、反转布尔变量的值，从而使得非 sysadm_r 具有执行 ping 或 traceroute 程序的能力：

```
[root/sysadm_r/s0@~]# getenforce
Enforcing
[root/sysadm_r/s0@~]# ping 128.224.162.248 -c 1
PING 128.224.162.248 (128.224.162.248) 56(84) bytes of data.
64 bytes from 128.224.162.248: icmp_seq=1 ttl=55 time=278 ms

--- 128.224.162.248 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 278ms
rtt min/avg/max/mdev = 278.751/278.751/278.751/0.000 ms
[root/sysadm_r/s0@~]# newrole -r secadm_r
Password:
[root/secadm_r/s0@~]# ping 128.224.162.248 -c 1
ping: icmp open socket: Permission denied
[root/secadm_r/s0@~]# getsebool user_ping
user_ping --> off
[root/secadm_r/s0@~]# setsebool user_ping on
[root/secadm_r/s0@~]# getsebool user_ping
user_ping --> on
[root/secadm_r/s0@~]# ping 128.224.162.248 -c 1
PING 128.224.162.248 (128.224.162.248) 56(84) bytes of data.
64 bytes from 128.224.162.248: icmp_seq=1 ttl=55 time=256 ms

--- 128.224.162.248 ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 256ms
rtt min/avg/max/mdev = 256.247/256.247/256.247/0.000 ms
[root/sysadm_r/s0@~]# getsebool -a | wc -l
95
[root/sysadm_r/s0@~]#
```

由此可见当把 user_ping 设置为 on 时，secadm_r 角色也能执行 ping 程序了。另外可以用“getsebool -a”命令获得所有布尔变量的当前值，它们的作用可参见源代码目录下 policy/boolean.conf 文件中的注释。

注意，如果想持久地改变一个布尔变量的值，必须使用“setsebool -P”或者“semanage boolean -m”命令，比如：

```
[root/sysadm_r/s0@~]#  semanage boolean -m -1 allow_mount_anyfile
type=1405 audit(1288604872.623:17): bool=allow_mount_anyfile val=1 old_val=0 auid=4294967295 ses=4294967295
type=1403 audit(1288604874.903:18): policy loaded auid=4294967295 ses=4294967295
```

```
[root/sysadm_r/s0@~]#
```

上述命令会在当前 policy store 下创建 booleans.local 文件，记录用户重载的 boolean 的值，参见 5.3 小节（SELinux 内核驱动根据 boolean 的当前值，决定相应规则块的 avtrue_list 或者 avfalse_list 分支上的规则加入内核 policydb_t 的 te_cond_avtab 哈希表，从而生效）。

3.9 Range Transition

如果需要某些程序在运行时创建的文件的 SL 高于默认的 s0，则相应程序必须在该 SL 中运行（因为文件对象的 SL 默认继承创建者的 low SL），即要求用户必须通过“newrole -l”命令来执行该程序；对于系统后台进程，还可以在相应的.te 中调用 init_ranged_daemon_domain 接口显式地指定运行时的 SL，比如：

```
init_ranged_daemon_domain(syslogd_t, syslogd_exec_t, mls_systemhigh)
```

系统启动过程中 init 程序（确切地说时 init 的子进程，在 initrc_t 中）执行 syslogd 的启动脚本，后者在执行 syslogd 程序时不但发生 domain transition 进入 syslogd_t，而且 SL 也同时切换为 mls_systemhigh。该接口的定义如下：

```
interface(`init_ranged_daemon_domain',`
        gen_require(`
                type initrc_t;
        ')

        init_daemon_domain($1,$2)                               # 使得 initrc_t 能够执行$2，然后切换到$1

        ifdef(`enable_mcs',`
                range_transition initrc_t $2:process $3;
        ')

        ifdef(`enable_mls',`
                range_transition initrc_t $2:process $3;        # 切换到指定的安全级别$3
                mls_rangetrans_target($1)                       # 将$1 加入 mlsrangetrans 属性
        ')
')
```

执行/etc/init.d/syslogd 启动脚本的 init 子进程的 domain 为 initrc_t，init_daemon_domain 接口使得 initrc_t 能够执行具体应用的可执行程序，并切换到具体应用程序的 domain 中。如果使能 MLS 特性，则上述 range_transition 规则使得 initrc_t 在执行可执行程序时，同时切换到指定的安全级别。

和 domain Transition 相关的 MLS 约束如下：

```
mlsconstrain process transition
     ( ( h1 dom h2 ) and
       (( l1 eq l2 ) or ( t1 == mlsprocsets1 ) or (( t1 == privrangetrans ) and ( t2 == mlsrangetrans )))
     );
```

即有两个条件：
1，h1 dom h2，即 clearance level 只能降低，无法升高；而且
2，当前安全级别保持不变，或者切换前的 domain 属于 mlsprocsets1 属性，或者切换前后的 domain 分别属于 privrangetrans 和 mlsrangetrans 属性。

上面在 init_ranged_daemon_domain > mls_rangetrans_target($1)接口中，将应用程序的 domain 加入了 mlsrangetrans 属性。另外，initrc_t/kernel_t/run_init_t 属于 privrangetrans 属性。二者相结合使得上述和 domain 切换相关的 MLS 约束能够被满足。

另外，所有能够直接设置子进程 SL 的 domain，比如 inetd_t/newrole_t/local_login_t/ssh_t/xdm_t 等，都在 mlsprocsets1 属性中。

3.10 Role Transition

type_transition 规则用于自动化 type 的转换（比如进程类，即发生 domain transition，或者文件类和 socket 类，即确定新创建对象的标签），与此类似，role_transition 规则用于自动化 role 的转换。比如在 logging_admin_audit 接口中存在如下规则：

```
role_transition $2 auditd_initrc_exec_t system_r;
allow $2 system_r;
```

那么当角色$2 在执行 auditd_initrc_exec_t 时切换到 system_r 角色中。注意，role_transition 只是自动化该切换过程，所以还必须显示地定义相应的 role-allow 规则以许可角色的切换。

role_transition 规则被广泛地用于 sysadm_r 通过 run_init 执行系统启动脚本的场合（即由系统管理员在系统启动后手工执行某些后台进程），以确保后台进程无论被 init 启动，或者被 sysadm 启动，都在 system_r 角色中运行。注意无论 init_t 或者 initrc_t，相应角色都是 system_r。

3.11 使用 setools 工具包分析 policy.X

setools 包提供了一组用于分析 SELinux 规则库的工具，它们是调试 policy.X 的得力助手。各种命令参见其主页（http://oss.tresys.com/projects/setools/）以及相应的 man 手册。

注意，无论 seinfo 还是 sesearch 命令，都可以在 Host 上或者 Target 上使用。在 Host 上使用时需要指定待读取的 policy.X 的路径名；在 Target 上使用时如果使能 MLS 特性则需要扮演 mls_systemhigh 安全级别或 secadm_r 角色。

3.11.1 seinfo 的使用

使用 seinfo 可以得到当前规则库中各种 SELinux 语法使用情况的摘要信息。比如在 Target 上执行 seinfo 结果如下：

```
[root/sysadm_r/s0@~]# seinfo
No default policy found.
[root/sysadm_r/s0@~]# seclow "seinfo"
Password:

Statistics for policy file: /etc/selinux/refpolicy-mls/policy/policy.24
Policy Version & Type: v.24 (binary, mls)

   Classes:            81    Permissions:       237
   Sensitivities:      16    Categories:       1024
   Types:            3400    Attributes:        227
   Users:               6    Roles:              14
   Booleans:            3    Cond. Expr.:         3
   Allow:          200417    Neverallow:          0
   Auditallow:          1    Dontaudit:       14763
   Type_trans:       8595    Type_change:        61
   Type_member:        16    Role allow:         29
```

```
   Role_trans:          5    Range_trans:        32
   Constraints:       308    Validatetrans:      17
   Initial SIDs:       27    Fs_use:             22
   Genfscon:           84    Portcon:           348
   Netifcon:            1    Nodecon:             0
   Permissives:         0    Polcap:              2

[root/sysadm_r/s0@~]#
```

注意，在 Target 上由于 seinfo 要访问 policy.X，而它的 SL 为 mls_systemhigh，所以应该使用 secadm_r 角色，或者 sysadm_r 扮演 mls_systemhigh 级别。

或者直接在 Host 使用 seinfo 命令，通过最后一个参数传递给它 Target 的 policy.X 文件的路径，比如：

```
cao@cao-laptop:~$ seinfo  /etc/selinux/refpolicy-mls/policy/policy.24

Statistics for policy file: /etc/selinux/refpolicy-mls/policy/policy.24
Policy Version & Type: v.24 (binary, mls)

   Classes:            81    Permissions:       237
   Sensitivities:      16    Categories:       1024
   Types:            3400    Attributes:        227
   Users:               6    Roles:              14
   Booleans:            3    Cond. Expr.:         3
   Allow:          200417    Neverallow:          0
   Auditallow:          1    Dontaudit:       14763
   Type_trans:       8595    Type_change:        61
   Type_member:        16    Role allow:         29
   Role_trans:          5    Range_trans:        32
   Constraints:       308    Validatetrans:      17
   Initial SIDs:       27    Fs_use:             22
   Genfscon:           84    Portcon:           348
   Netifcon:            1    Nodecon:             0
   Permissives:         0    Polcap:              2

cao@cao-laptop:~$
```

注意，如果这样使用，则 policy.X 的版本号不能大于 Host 上安装的 libsepol 所支持的最大版本号（否则可以通过指定“policy-version”变量降低 policy.X 的版本号）。

seinfo 可以输出某个语法成分的具体信息。比如显示当前 policy 中定义的所有 SELinux User：

```
[root/sysadm_r/s0@~]# newrole -l sl5:c0.c1023 -- -c "seinfo -u"
Password:

Users: 6
   sysadm_u
   system_u
   root
   staff_u
   user_u
   unconfined_u
[root/sysadm_r/s0@~]#
```

结合使用“-x”选项，可以关于当前 SELinux User 所能够扮演的 role 信息：

```
[root/sysadm_r/s0@~]# newrole -l sl5:c0.c1023 -- -c "seinfo -uroot -x"
```

```
Password:

   root
      default level: s0
      range: s0 - s15:c0.c1023
      roles:
         object_r
         auditadm_r
         staff_r
         secadm_r
         sysadm_r
[root/sysadm_r/s0@~]#
```

类似地，可以使用“-r”选项得到所有 role 的列表：

```
[root/sysadm_r/s0@~]# newrole -l s15:c0.c1023 -- -c "seinfo -r"
Password:

Roles: 14
   auditadm_r
   dbadm_r
   guest_r
   staff_r
   user_r
   logadm_r
   object_r
   secadm_r
   sysadm_r
   system_r
   webadm_r
   xguest_r
   nx_server_r
   unconfined_r
[root/sysadm_r/s0@~]#
```

进一步使用“-x”选项，得到该 role 能够相关联的 type 列表：

```
[root/sysadm_r/s0@~]# newrole -l s15:c0.c1023 -- -c "seinfo -rsysadm_r -x | head"
Password:

   system_r
      Dominated Roles:
         system_r
      Types:
         djbdns_tinydns_t
         sosreport_t
         portslave_t
         bootloader_t
         netutils_t
         qmail_tcp_env_t
[root/sysadm_r/s0@~]#
```

值得一提的是，目前并没有把任何 role 属性写入 policy.X（因为 role 属性的使命在 link 和 expand 时就已经完成：其能够关联的 type 集合，被散播给所有属于该 role 属性的普通 role），所以无法直接看到一个 role 属性能够关联的 type 集合。因此只能进一步观察从属于它的普通 role 的能力是否满足要求。比如，就 sysadm_r 调用了 seutils_run_newrole 接口加入 newrole_roles 属性，但是在实际使用中发现 sysadm_r 无法正常使用 newrole 命令，结果发现 sysadm_r 无法和 newrole_t 相关联，于是问题只能出在

newrole_roles 属性没有和 newrole_t 相关联。

结合使用“-t”和“-x”选项，可以得到一个 type 所属于的所有 domain 的信息：

```
cao@cao-laptop:~$ seinfo -tsysadm_t -x /etc/selinux/refpolicy-mls/policy/policy.24
   sysadm_t
      sepgsql_unconfined_type
      process_user_target
      xserver_unconfined_type
      can_change_object_identity
      can_relabelto_binary_policy
      xdrawable_type
      ubacproc
      mlsprocread
      admindomain
      userdomain
      xcolormap_type
      ubacfd
      can_write_binary_policy
      ubac_constrained_type
      ubacfile
      x_domain
      domain
cao@cao-laptop:~$
```

最后，使用 seinfo 还能够得到静态使用的 Initial SID，fs_use，genfscon 等命令的列表，详见手册页。

3.11.2 sesearch 的使用

可以用 sesearch 工具精细地查找一条具体的 SELinux 规则，用户在命令行指定待查找的语法类型、Subject/Object 的 type，Object 所属的 class，以及访问权限等信息。比如查找关于 crond_t domain 对 dir 类 bin_t 对象读操作相关的 allow 规则：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s crond_t -t bin_t -c dir -p read"
Password:
Found 1 semantic av rules:
   allow crond_t bin_t : dir { ioctl read getattr lock search open } ;

[root/sysadm_r/s0@~]#
```

同样，由于 sesearch 也需要读取 policy.X 文件，所以必须以 secadm_r 角色，或者 sysadm_r 采用 mls_systemhigh 安全级别来运行它。

又比如查找和 sysadm_t 新创建的 file 类型对象的 type-transition 规则：

```
cao@cao-laptop:~$ sesearch -SCT -s sysadm_t -c file /etc/selinux/refpolicy-mls/policy/policy.24
Found 3 semantic te rules:
   type_transition sysadm_t user_home_dir_t : file user_home_t;
   type_transition sysadm_t tmp_t : file user_tmp_t;
   type_transition sysadm_t tmpfs_t : file user_tmpfs_t;
cao@cao-laptop:~$
```

总之，sesearch 是分析 AVC Denied Message 的得力工具，详见其手册页，一定要熟练使用。

3.11.3 在 Host 上使用 apol 具分析 policy.X

相对于命令行 seinfo 和 sesearch 工具，setools 包还提供了支持图形界面的 apol 工具，其功能更为强大。在 Ubuntu Host 上也可以使用“sudo apt-get install setools”命令来安装它（但是这样得到的 apol 程序往往静态链接了较低版本的 libsepol，因此无法正常打开最新 toolchain 编译的 policy.X，所以通常需要指定 policy.X 的版本号为.24。或者使用最新 toolchain 在 Host 上重新编译 setools 包）。

1，在 Policy Components -> Types 中可以在“Search using regular expression”栏中输入待查的 type 或者 attribute，点击 OK，就可以得到当前 type/attribute 的属性。比如就 samhaind_t 的而言查询结果下：

```
TYPES (1):

samhaind_t (5 attributes)
    can_read_shadow_passwords
    daemon
    domain
    mlsfilewrite
    mlsrangetrans

ATTRIBUTES (0):
```

从而可以直观地观察 samhaind_t 所属的属性，比如检查它是否属于 mlsfilewrite 和 mlsrangetrans 属性。

2，在 Policy Rules -> TE Rules 中可以指定 source/target type 以及所有相关的规则类型：allow/auditallow/dontallow/neverallow/type_transition/type_member/type_change。比如输入 source = secadm_t，target = samhain_exec_t，然后点击“New Search”，可以得到如下结果：

```
17 rules match the search criteria.
Number of enabled conditional rules: 0
Number of disabled conditional rules: 0

dontaudit secadm_t non_security_file_type : file getattr ;
dontaudit secadm_t non_security_file_type : dir { ioctl read getattr lock search open } ;
dontaudit secadm_t non_security_file_type : lnk_file getattr ;
dontaudit secadm_t non_security_file_type : sock_file getattr ;
dontaudit secadm_t non_security_file_type : fifo_file getattr ;
allow secadm_t samhain_exec_t : file { read getattr relabelfrom relabelto execute open } ;
allow secadm_t samhain_exec_t : dir { ioctl read getattr lock relabelfrom relabelto search open } ;
allow secadm_t samhain_exec_t : lnk_file { getattr relabelfrom relabelto } ;
allow secadm_t samhain_exec_t : chr_file { getattr relabelfrom } ;
allow secadm_t samhain_exec_t : blk_file { getattr relabelfrom } ;
allow secadm_t samhain_exec_t : sock_file { getattr relabelfrom relabelto } ;
allow secadm_t samhain_exec_t : fifo_file { getattr relabelfrom relabelto } ;
allow secadm_t application_exec_type : file { ioctl read getattr lock execute execute_no_trans open } ;
allow secadm_t file_type : filesystem getattr ;
dontaudit secadm_t exec_type : file { execute execute_no_trans } ;
dontaudit secadm_t entry_type : file { ioctl read getattr execute execute_no_trans open } ;
type_transition secadm_t samhain_exec_t : process samhain_t;
```

说明，由于 samhain_exec_t 属于 file_type，exec_type，entry_type，non_security_file_type 和 application_exec_type 属性，因此上面的搜索结果也默认地包含 secadm_t 对这些属性的相关规则。

3，在 Analysis 中选择“Analysis Type”为“Domain Transition”，则可以分析从指定的 source domain 开始所有能够达到的 target domain（此时 Direction 选择 Forward）。也可以分析能够进入指定 target domain 的 source domain（此时 Direction 选择 Reverse）。比如输入 samhain_t 并选择反向分析，则得到树状图：

```
        samhain_t
            |- secadm_t
```

并且所有和它们之间 Domain Transition 相关的规则如下：

```
Domain transition from secadm_t to samhain_t

Process Transition Rules: 1
      allow secadm_t samhain_t : process transition ;

Entry Point File Types: 1
      samhain_exec_t
            File Entrypoint Rules: 1
            allow samhain_t samhain_exec_t : file { ioctl read getattr lock execute entrypoint open } ;

            File Execute Rules: 2
            allow secadm_t application_exec_type : file { ioctl read getattr lock execute execute_no_trans
open } ;
            allow secadm_t samhain_exec_t : file { read getattr relabelfrom relabelto execute open } ;

            Type_transition Rules: 1
            type_transition secadm_t samhain_exec_t : process samhain_t;
```

3.12 在 Host 上使用 Checkpolicy 访问用户态 Security Server

内核中的 Security Server 为 SELinux 底层机制，而用户态 policy.X 提供上层策略。在装载 policy.X 到内核空间时 SELinux 内核驱动创建相应的内核描述符，而 Security Server 借助它们做出决策。

为了便于开发和调试，在用户态 libsepol 中也实现了 Security Server 机制，并且可以直接使用“checkpolicy -Mdb”命令来执行 Security Server 以查询 policy.X。可以首先使用“context_to_sid”命令注册 context 字符串得到相应的 sid，然后调用相应的命令，计算 newsid，最后使用“sid_to_context”命令得到 newcontext 字符串。比如：

```
cao@cao-laptop:/work/selinux/refpolicy$ checkpolicy -Mdb /etc/selinux/refpolicy-mls/policy/policy.24
checkpolicy:  loading policy configuration from /etc/selinux/refpolicy-mls/policy/policy.24
libsepol.policydb_index_others: security:  6 users, 36 roles, 3636 types, 3 bools
libsepol.policydb_index_others: security: 16 sens, 1024 cats
libsepol.policydb_index_others: security:  81 classes, 223749 rules, 104 cond rules
checkpolicy:  policy configuration loaded

Select an option:
0)  Call compute_access_vector
1)  Call sid_to_context                    # 查询 sidtab 得到 SID 所对应的 context 字符串
2)  Call context_to_sid                    # 注册 context 字符串得到 SID
3)  Call transition_sid                    # 查询 type_transition 规则
4)  Call member_sid                        # 查询 type_member 规则
5)  Call change_sid                        # 查询 type_change 规则
6)  Call list_sids                         # 打印当前所有的 SID（包括 Initial SID）
7)  Call load_policy
8)  Call fs_sid
9)  Call port_sid
a)  Call netif_sid
b)  Call node_sid
c)  Call fs_use
d)  Call genfs_sid
```

```
e)  Call get_user_sids
f)  display conditional bools
g)  display conditional expressions
h)  change a boolean value
m)  Show menu again
q)  Exit

Choose:  2
scontext?  root:sysadm_r:sysadm_t:s0-sl5:c0.cl023

sid 28

Choose:  2
scontext?  system_u:object_r:console_device_t:s0

sid 29

Choose:  5
source sid?  28
target sid?  29
object class?  chr_file

sid 30

Choose:  1
sid?  30

scontext root:object_r:user_tty_device_t:s0

Choose:  6
sid 1 -> scontext system_u:system_r:kernel_t:sl5:c0.cl023
sid 2 -> scontext system_u:object_r:security_t:sl5:c0.cl023
sid 3 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 4 -> scontext system_u:object_r:fs_t:s0
sid 5 -> scontext system_u:object_r:file_t:s0
sid 6 -> scontext system_u:object_r:unlabeled_t:s0
sid 7 -> scontext system_u:object_r:unlabeled_t:s0
sid 8 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 9 -> scontext system_u:object_r:port_t:s0
sid 10 -> scontext system_u:object_r:netif_t:s0-sl5:c0.cl023
sid 11 -> scontext system_u:object_r:netlabel_peer_t:sl5:c0.cl023
sid 12 -> scontext system_u:object_r:node_t:s0-sl5:c0.cl023
sid 13 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 14 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 15 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 16 -> scontext system_u:object_r:unlabeled_t:s0
sid 17 -> scontext system_u:object_r:sysctl_t:s0
sid 18 -> scontext system_u:object_r:unlabeled_t:s0
sid 19 -> scontext system_u:object_r:unlabeled_t:s0
sid 20 -> scontext system_u:object_r:unlabeled_t:s0
sid 21 -> scontext system_u:object_r:unlabeled_t:s0
sid 22 -> scontext system_u:object_r:unlabeled_t:s0
sid 23 -> scontext system_u:object_r:unlabeled_t:s0
sid 24 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 25 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 26 -> scontext system_u:object_r:unlabeled_t:sl5:c0.cl023
sid 27 -> scontext system_u:object_r:null_device_t:s0
sid 28 -> scontext root:sysadm_r:sysadm_t:s0-sl5:c0.cl023
sid 29 -> scontext system_u:object_r:console_device_t:s0
```

```
sid 30 -> scontext root:object_r:user_tty_device_t:s0

Choose:  q
cao@cao-laptop:/work/selinux/refpolicy$
```

在这个例子中，SID 1～27 为 Initial SID，因此用户注册的 context 所对应的 SID 从 28 开始分配。相应 type_change 规则的结果对应 SID 30，再用“sid_to_context”命令即可得到新的安全上下文的字符串，该结果和在 Target 上使用 compute_relabel 命令得到的相同。

注意，由于用户态 Security Server 的代码疏于维护因而落后于内核 Security Server，因此应该以后者为准。

4. Reference Policy

NSA 最初实现的 SELinux policy 被称为“Example Policy”，最大的缺点是耦合性太高：所有 policy 标识符（user，type，attribute，role 等）的定义都是**全局的**，难于理解，修改和维护。

后来采用了现代软件设计思想重新设计的 policy 被称为“Reference Policy”（不同 OS 发行版的开发者可以在其基础上定制各自的 policy，故称其为“reference”），特点为：
1，一套代码树同时支持 strict/targeted 两种类型的 policy；支持 MLS/MCS；支持将 policy 编译成两种格式：Monolithic 或 Modular；
2，采用“高内聚”，“低耦合”的设计思想，pp 定义清晰的外部接口，所有 type 的定义都在模块内使用，取消了全局标识符（type/attribute 等）；
3，对文档的支持，在.if 中定义的 interface 都有 XML 格式的语法，可以用 make html 生成所有接口的文档；
4，简化并标准化了 policy 的编译参数，在社区发行版中都集中在 build.conf 中（见下）；

4.1 Reference Policy 代码树的主要结构

社区发行版 refpolicy 包的根目录结构如下：

```
refpolicy$ ls
build.conf  config/   doc/    Makefile  policy/  Rules.modular     support/
Changelog   COPYING  INSTALL  man/      README   Rules.monolithic  VERSION
```

1，build.conf 集中了所有的配置参数：

```
refpolicy$ grep = build.conf
#OUTPUT_POLICY = 18
TYPE = mls              # or: standard / mcs
NAME = refpolicy-mls
DISTRO = redhat         # or: debian / ubuntu / gentoo
UNK_PERMS = allow       # or: deny / reject
DIRECT_INITRC = n
MONOLITHIC = n          # MONOLITHIC = n时 make X.pp才生效（Makefile才会包含Rules.modular）
UBAC = y
CUSTOM_BUILDOPT =
MLS_SENS = 16
MLS_CATS = 1024
MCS_CATS = 1024
QUIET = y
```

特别注意，一定要针对不同的 Linux 发行版选择合适的 DISTRO 的值！这是因为 refpolicy 源代码的许多地方将针对不同发行版调用不同的接口，通过 ifdef(`distro_xxx',`xxx()')来控制。

UNK_PERMS 配置项可以被者设置为 allow/deny/reject。如果 SELinux 内核中引入了新的数据对象类型，但是缺乏 refpolicy 中的相应定义，即 policy.X 中没有该 class 及相关权限的用户态定义，相应规则也不存在。此时 SELinux 内核驱动根据该选项的值决定是否许可相关的操作。参见后文 security_compute_av 函数。

而用户可以在 CUSTOM_BUILDOPT 中自定义自己的编译选项（以空格间隔）：

```
ifneq "$(CUSTOM_BUILDOPT)" ""
        M4PARAM += $(foreach opt,$(CUSTOM_BUILDOPT),-D $(opt))
```

2，config/appconfig-<TYPE>/目录的内容为 SELinux 运行时的部分配置文件，将被安装

到/etc/selinux/<NAME>/contexts/目录下：

```
refpolicy$ ls config
appconfig-mcs/  appconfig-mls/  appconfig-standard/  local.users
$ ls config/appconfig-mls/
dbus_contexts             securetty_types
default_contexts          seusers
default_type              staff_u_default_contexts
failsafe_context          unconfined_u_default_contexts
guest_u_default_contexts  userhelper_context
initrc_context            user_u_default_contexts
media                     x_contexts
removable_context         xguest_u_default_contexts
root_default_contexts
```

3，Makefile 可选择性地包含 Rules.modular 或 Rules.monolithic 文件，由 build.conf 中定义的 MONOLITHIC 变量决定编译方式。只有当“MONOLITHIC = n”时才以模块方式编译，此时可以指定编译一个特定的模块，比如 make sysadm.pp。

4，policy/为 refpolicy 代码树所在目录：

```
refpolicy$ ls policy
booleans.conf  global_booleans  mls           policy_capabilities
constraints    global_tunables  modules/      support/
flask/         mcs              modules.conf  users

policy/flask$ find
.
./security_classes
./initial_sids
./access_vectors
./flask.py
./Makefile
./userspace
./userspace/common_perm_to_string.h
./userspace/av_inherit.h
./userspace/av_permissions.h
./userspace/class_to_string.h
./userspace/initial_sid_to_string.h
./userspace/flask.h
./userspace/av_perm_to_string.h
```

flask/目录下的 security_classes 和 access_vectors 文件包含所有 class 及其访问权限的定义，initial_sid 文件包含所有 Initial SID 的定义（由 SELinux 内核驱动初始化之后、init 在系统初始化时装载 policy.X 之前使用）。注意 userspace/子目录下的头文件，都是通过 python 脚本 flask.py 生成。

在 global_booleans 和 global_tunables 文件中定义**全局**使用的 boolean 或 tunable 变量及其默认值，而在单个 pp 内**局部**使用的 boolean 或 tunable 则在相应.te 中定义，比如 allow_ptrace 只在 sysadm.te 中定义并使用。

在执行“make conf”命令时根据 refpolicy/support/sedoctool.py 脚本创建 booleans.conf 文件，它从 global_booleans 和 global_tunables 文件以及各个.te 中抽取所有 boolean 和 tunable 的说明信息和默认值。booleans.conf 文件将被安装为/etc/selinux/$SELINUXTYPE/booleans。

constraints，mls，mcs 为和 standard/MLS/MCS 相关的约束的定义；

users 中包含默认 SELinux User 的定义，比如 system_u，user_u，staff_u，root，sysadm_u 等；

modules.conf 为所有 policy module 的列表以及编译方式：base/module/off。该文件可以由“make conf”命令生成。如果不需要编译某些模块，则可直接指定为 off。

support/*.spt 为各种支持文件，包含各种 m4 宏的定义，比如 interface，gen_tunable，gen_user 等。

policy_capabilities 文件列举了当前 refpolicy 实现所支持的能力，以及影响的 class 及其 perms 定义。

5，policy/modules/目录为所有 policy package（简称为 pp）的源代码目录：

```
refpolicy$ ls policy/modules
admin/  apps/  contrib/ kernel/ roles/  services/  system/
```

根据 pp 的层次和用途将其划分到不同的“layer”中：
kernel：描述 Linux kernel 提供的各种资源，比如 devices，terminal，files，filesystem
system：各种系统工具，比如 logging，mount，modutils，selinuxutil，userdomain
admin：系统管理工具，比如 dmesg，sudo，su，netutils
services：系统应用工具，比如 ssh，xserver，postgresql
roles：各种缺省 SELinux Role 的实现，比如各种 admin，staff 和 unprivuser
contrib：refpolicy 社区贡献的修改，或者新的 pp，往往都放在这里。

注意，理论上每种 layer 中的 pp 只依赖于同 layer 中的其他 pp，或者 system/kernel layer 中的 pp。system 依赖于 kernel，而除 kernel/system 之外的其他 layer 之间没有相互依赖的上下级关系。

由于 refpolicy 的维护者 cjp 个人精力有限，而社区又很活跃（主要来自于 RH），所以他决定把 system 和 kernel 目录外其它目录下的大部分内容，都放到了 contrib 目录下。这样 cjp 可以集中精力保证 refpolicy 最核心规则的质量，而对 contrib 目录中的内容审查得较不苛刻。

4.2 Policy Package 的源代码文件

在 reference policy 中，任何一个 pp 的实现都由三个文件组成：
.te：定义 type，role，attribute 等，根据 Least Privilege 原则赋予相应 domain 在能够支持应用程序正常行为条件下最小的权限；
.fc：定义和应用程序相关的所有安装文件的 SC（参见下文说明）；
.if：定义该 pp 向外导出的 interface/template，以便其他 domain 调用合适的接口获得对当前 pp 所定义的 domain/type 的某些能力；

针对.fc 文件的使用有三点说明：
1，首先，定义应用程序的所有安装文件（比如 rpm -qpl）的**默认标签**。
在使能 SELinux 之前需要给整个文件系统打标签，各个 pp 的.fc 被汇总为 file_contexts 文件，restorecon 参照之给相应文件被打上合适的标签，从而使得相应 domain 能够顺利使用这些文件。

2，其次，定义这些文件的**安装路径**。
.fc 的语法类似与 BRE 类似，只有在指定路径上的文件才能被打上所指定的标签。如果由于版本或使用上的差异，应用程序所使用的文件并没有安装在.fc 所期望的路径上，那么这些文件的标签就不会被正确的设置，而继承相关父目录的标签，这将导致应用程序 domain 没有足够的能力使用这些文件。此时起码应该把相关文件的实际路径加入.fc 文件中，并根据需要给.te 文件在相应目录下创建所需文件的能力（参见 3.4 小节中关于 quota 的例子）。

3，另外，.fc 文件只定义**已存在**文件的标签，在 restorecon 时使用。而相应 domain 在**运行时新建**文件

的标签，如果不希望继承相关父目录的标签，则必须在.te 中调用 files_xxx_filetrans 接口通过 type_transition 规则明确地指定（参见 3.4 小节 type_transition 规则的使用）。

值得一提的是，在系统运行后再使用 restorecon 修复文件的标签时必须慎重，因为它会把文件的标签恢复到.fc 中指定的初始状态，从而抹杀运行时的修改（比如通过 type_transition 和 type_change 规则，或 SELinux-aware 程序的显式设置）。

.te，.if 和.fc 文件的开发过程详见第 7 章开发 vlock.pp 的相关内容。

5. SELinux 的用户态设施

SELinux 在用户态的配置文件，policy.X，以及 policy store 都在/etc/selinux/目录下。在 libselinux 中实现了许多读取 SELinux 用户态配置文件的函数，在其 file_path_suffixes.h 中定义了各个配置文件的索引。SELinux-aware 应用程序借助 libselinux API 来访问这些设置。

可以使用 strace 命令来观察 SELinux-aware 应用程序的行为，观察其如何访问 SELinux 用户态设施的（比如仔细研究采用 8.8.1 小节所述方法得到的 mingetty/login 后台进程的 strace 结果）。

5.1 /etc/selinux/config

系统上可能同时安装若干套 policy.X，该文件定义系统启动后使用哪一个 policy.X 及其运行方式：

```
[root/sysadm_r/s0@~]# cd /etc/selinux/
[root/sysadm_r/s0@selinux]# ls -Z
-rw-r--r--  root root system_u:object_r:selinux_config_t:s0 config
drwxr-xr-x  root root system_u:object_r:selinux_config_t:s0 refpolicy-mls
-rw-------  root root system_u:object_r:selinux_config_t:s0 restorecond.conf
drwxr-xr-x  root root system_u:object_r:selinux_config_t:s0 seedit
-rw-r--r--  root root system_u:object_r:selinux_config_t:s0 semanage.conf
[root/sysadm_r/s0@selinux]#
[root/sysadm_r/s0@selinux]# cat config
SELINUXTYPE=refpolicy-mls
SELINUX=enforcing
[root/sysadm_r/s0@selinux]#
```

当前使用的 policy 所在的目录名由 SELINUXTYPE 变量指定，init 进程会在系统启动时装载/etc/selinux/$*SELINUXTYPE*/policy/policy.X。

SELINUX 变量可能的值为：enforcing/permissive/disable，其中 permissive 模式使得 SELinux 的判定结果（允许或禁止）不影响操作的实际执行，用于调试 policy.X。

另外，semanage.conf 文件为运行时 libsemanage 库行为的配置文件。比如可以通过“policy-version”选项指定以 Modular 方式编译时 policy.X 的版本号。所有配置项说明参见其 man 手册页。

5.2 /etc/selinux/$SELINUXTYPE/

进入当前被使用的 SELinux policy 所在的子目录后，使用 secadm_r 观察该目录结构如下：

```
[root/sysadm_r/s0@selinux]# cd refpolicy-mls/
[root/sysadm_r/s0@refpolicy-mls]# seclow "ls -Z"
Password:
drwxr-xr-x  root root system_u:object_r:default_context_t:s0 contexts
drwx------  root root system_u:object_r:selinux_config_t:s0 modules
drwxr-xr-x  root root system_u:object_r:policy_config_t:sl5:c0.cl023 policy
-rw-r--r--  root root system_u:object_r:selinux_config_t:sl5:c0.cl023 seusers
[root/sysadm_r/s0@refpolicy-mls]#
```

不同 SELinux policy 的安装目录树结构很类似，基本都包含上述文件和目录。

5.2.1 /etc/selinux/$SELINUXTYPE/seusers

seusers 文件在 secadm_r 执行“semanage login -m/-a”命令修改或添加 UNIX User 到 SELinux User 的

映射关系时由用户态 libsemanage 库自动生成，它的内容和“semanage login -l”命令的结果一致：

```
[root/sysadm_r/s0@refpolicy-mls]# seclow "cat seusers"

Password:
system_u:system_u:s0-s15:c0.c1023
root:root:s0-s15:c0.c1023
__default__:user_u:s0
[root/sysadm_r/s0@refpolicy-mls]# semanage login -l

Login Name                SELinux User              MLS/MCS Range

__default__               user_u                    s0
root                      root                      s0-s15:c0.c1023
system_u                  system_u                  s0-s15:c0.c1023
[root/sysadm_r/s0@refpolicy-mls]#
```

libselinux 源码中 seusers.c 文件的 getseuserbyname 函数打开的正是该文件，返回相应 UNIX User 所对应的 SELinux User 及其安全级别。该函数的型构参见其 man 手册页。

5.2.2 /etc/selinux/$SELINUXTYPE/contexts/

该目录下的配置文件决定各种场合下进程或者文件的 SC。

```
[root/sysadm_r/s0@refpolicy-mls]# cd contexts/
[root/sysadm_r/s0@contexts]# ls
customizable_types  failsafe_context    removable_context   users /
dbus_contexts       files/              securetty_types     x_contexts
default_contexts    initrc_context      sepgsql_contexts
default_type        netfilter_contexts  userhelper_context
[root/sysadm_r/s0@contexts]#

[root/sysadm_r/s0@contexts]# ls files/
file_contexts  file_contexts.homedirs  file_contexts.subs_dist  media
[root/sysadm_r/s0@contexts]#
[root/sysadm_r/s0@contexts]# head files/file_contexts
/.*     system_u:object_r:default_t:s0
/a?quota\.(user|group)  --      system_u:object_r:quota_db_t:s0
/sys(/.*)?      system_u:object_r:sysfs_t:s0
/xen(/.*)?      system_u:object_r:xen_image_t:s0
/mnt(/[^/]*)    -l      system_u:object_r:mnt_t:s0
/mnt(/[^/]*)?   -d      system_u:object_r:mnt_t:s0
/opt/.* system_u:object_r:usr_t:s0
/var/.* system_u:object_r:var_t:s0
/usr/.* system_u:object_r:usr_t:s0
/srv/.* system_u:object_r:var_t:s0
[root/sysadm_r/s0@contexts]#

[root/sysadm_r/s0@contexts]# cat files/media
cdrom system_u:object_r:removable_device_t:s0
floppy system_u:object_r:removable_device_t:s0
disk system_u:object_r:fixed_disk_device_t:s0
[root/sysadm_r/s0@contexts]#
```

files/目录下的文件被那些需要给文件设置 SC 的应用程序使用，比如 rpm，udev，restorecon 等。files/file_contexts 描述整个根文件系统的 SC，用 restorecon 命令给整个文件系统设置 SC 时读取的正是该文件。

files/file_contexts.homedirs 描述各个用户 HOME 目录的 SC。创建新用户后一般需要执行 genhomedircon 命令更新该文件，然后再用 restorecon 命令设置用户 HOME 目录的 SC。

files/media 文件为各种移动设备的默认标签。

users/[user]文件决定了 SELinux User 在使用某个登录设施时（比如 login/ssh/cron/su 等），如果该用户能够扮演多种角色，登录设施应该为相应的 shell 所设置的默认 role/type。

```
[root/sysadm_r/s0@contexts]# ls users
guest_u  root  staff_u  unconfined_u  user_u  xguest_u
[root/sysadm_r/s0@contexts]# cat users/root
......
system_r:local_login_t:s0       unconfined_r:unconfined_t:s0 sysadm_r:sysadm_t:s0 staff_r:staff_t:s0
user_r:user_t:s0
#
# Uncomment if you want to automatically login as sysadm_r
#
#system_r:sshd_t:s0             unconfined_r:unconfined_t:s0 sysadm_r:sysadm_t:s0 staff_r:staff_t:s0
user_r:user_t:s0
[root/sysadm_r/s0@contexts]# cat default_contexts
......
system_r:local_login_t:s0       user_r:user_t:s0 staff_r:staff_t:s0 sysadm_r:sysadm_t:s0
unconfined_r:unconfined_t:s0
system_r:sshd_t:s0              user_r:user_t:s0 staff_r:staff_t:s0 sysadm_r:sysadm_t:s0
unconfined_r:unconfined_t:s0
[root/sysadm_r/s0@contexts]#
```

users/[user]文件和 default_contexts 文件的格式相同，每一行第一对{role type}为相应登录设施的上下文，余下组合中第一个被当前 policy.X 所许可的该 SELinux User 能够扮演的 role 及相应 type 即为在该登录设施上该 SELinux User 的缺省上下文。users/[user]文件先于 default_contexts 文件被处理。

比如，root 用户能够扮演 sysadm/staff 等角色，因此在 local_login_t 上登录时，相应登录 shell 的安全上下文中扮演 sysadm_r 角色。由于 users/root 文件中关于“system_r:sshd_t:s0”的行被注销掉，因此当 root 用户使用 ssh 登录时，由 default_contexts 文件指明其登录 shell 的默认上下文中扮演 staff_r 角色。

值得一提的是，这些登录设施要么在自己的源代码中（比如 cron），要么通过相应的 PAM 模块（比如 login/ssh 都可在 session 阶段使用 pam_namespace.so）调用 getseuserbyname 函数获得 UNIX User 用户对应的 SELinux User 及其安全级别，然后调用 get_default_context_with_level 函数访问上述文件获得该用户的默认安全上下文，参见其 man 手册页。

用户在使用 newrole 命令切换到新角色时，无须显式指定相应的 type。和某种 role 对应的默认 type 由 default_type 文件指定：

```
[root/sysadm_r/s0@contexts]# cat default_type
auditadm_r:auditadm_t
secadm_r:secadm_t
sysadm_r:sysadm_t
staff_r:staff_t
unconfined_r:unconfined_t
user_r:user_t
[root/sysadm_r/s0@contexts]#
```

securetty_types 文件列举那些支持切换安全级别的 terminal 设备的标签，比如：

```
[root/sysadm_r/s0@contexts]# cat securetty_types
user_tty_device_t
[root/sysadm_r/s0@contexts]#
```

其中 user_tty_device_t 是所有 tty 以及 console 设备在用户登录后被 pam_selinux.so 根据相应 type_change 规则重新 relabel 后的标签，所以用户在 tty 或者 console 上登录后，能够使用“newrole -l”命令切换 SL。

如果在该文件中没有指定 user_devpts_t，则用户通过 ssh 在/dev/pty/*设备上登录后，将无法切换 SL：

```
[root/staff_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_devpts_t:s0   /dev/pts/0
[root/staff_r/s0@~]#
[root/staff_r/s0@~]# newrole -l s0:c1
Error: you are not allowed to change levels on a non secure terminal
[root/staff_r/s0@~]#
[root/staff_r/s0@~]# newrole -r sysadm_r -- -c "echo user_devpts_t >> /etc/selinux/refpolicy-
mls/contexts/securetty_types"
Password:
[root/staff_r/s0@~]#
[root/staff_r/s0@~]# newrole -l s0:c1
Password:
[root/staff_r/s0:c1@~]#
```

5.2.3 /etc/selinux/$SELINUXTYPE/contexts/initrc_context 文件，run_init 程序和系统启动脚本

和系统启动脚本，run_init 程序相关的总结如下：

1，系统启动脚本的标签要么为通用的 initrc_exec_t，要么为自定义的 xxx_initrc_exec_t，它们都属于 init_script_file_type 属性。init 进程在 initrc_t 中执行各启动脚本，initrc_t 对 init_script_file_type 属性同时具有 execute 和 entrypoint 权限：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s initrc_t -t init_script_file_type -c file -p
execute_no_trans"
Password:
Found 1 semantic av rules:
   allow initrc_t init_script_file_type : file { ioctl read getattr lock execute execute_no_trans open } ;

[root/sysadm_r/s0@~]#
```

这意味这 initrc_t 进程能够运行所有的 initrc_exec_t 和 xxx_initrc_exec_t 程序，并且仍然保持在 initrc_t domain 中。它们分别靠调用下述接口实现：

```
domain_entry_file(initrc_t, initrc_exec_t)
init_script_file(xxx_initrc_exec_t) > domain_entry_file(initrc_t, $1)
```

注意，对所有 xxx_initrc_exec_t 都必须调用 init_script_file 接口，使得 initrc_t 以它为 entrypoint，并且当 initrc_t 执行它时仍然处在 initrc_t domain 中（initrc_t 属于 init_run_all_scripts_domain 属性）：

```
interface(`init_script_file',`
        gen_require(`
                type initrc_t;
                attribute init_script_file_type, init_run_all_scripts_domain;
        ')

        typeattribute $1 init_script_file_type;                 # 将 xxx_initrc_exec_t 加入该属性
        domain_entry_file(initrc_t, $1)                         # 将 xxx_initrc_exec_t 作为 initrc_t 的接入点
        domtrans_pattern(init_run_all_scripts_domain, $1, initrc_t)
')
```

2，所以，在系统启动过程中 init 进程运行各个脚本时，domain 仍然在 initrc_t 中。

3，对所有安装启动脚本的应用程序 domain，如果为需要长期运行的系统后台进程，则必须调用 init_ranged_daemon_domain 或者 init_daemon_domain 接口；如果为短期运行的后台进程，则可以调用 init_ranged_system_domain 或者 init_system_domain 接口。比如：

```
init_daemon_domain(auditd_t, auditd_exec_t)
init_ranged_daemon_domain(auditd_t, auditd_exec_t, mls_systemhigh)      # when MLS is enabled

interface(`init_daemon_domain',`
        ......

        typeattribute $1 daemon;

        domain_type($1)
        domain_entry_file($1,$2)
        role system_r types $1;                         # 系统后台进程 SC 的角色为 system_r

        domtrans_pattern(initrc_t,$2,$1)
        ......
')
```

启动脚本中通常会调用相应的可执行程序，上述 domtrans_pattern(initrc_t，$2，$1)使得 initrc_t 在执行启动脚本时切换到相应程序的 domain 中。

4，initrc_context 文件**指定**了/usr/sbin/run_init 程序在执行时的 domain：

```
[root/sysadm_r/s0@~]# cat /etc/selinux/refpolicy-mls/contexts/initrc_context
system_u:system_r:initrc_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
```

从而实现 run_init+initrc_context 机制的目的：以和 init 进程运行系统启动脚本相同的方式，让 sysadm 能够运行启动脚本。当 sysadm_r 通过 run_init 运行启动脚本时，发生的 domain 切换如下：
1）sysadm_t → run_init_t，在运行 run_init 程序期间。
2）run_init_t → initrc_t，run_init 为 SELinux-aware 程序，它读取 initrc_context 文件的内容并调用 setexeccon 函数写入自己的/proc/self/attr/exec 文件，在通过 execve 系统调用执行相应启动脚本时进入 initrc_t。
3）initrc_t → xxx_t，在相应启动脚本运行可执行程序时。此阶段和由 init 执行启动脚本时行为一致。

使能 allow_ptrace 后（在编译时将其默认值修改为 true，或者在运行时通过 setsebool 或 semanage 命令），sysadm_r 可以通过 strace 观察 run_init 程序的行为，比如：

```
[root/sysadm_r/s0@contexts]# strace -e trace=open,read,write run_init /etc/init.d/samhain status 2>
```

```
/root/run_init.strace
Authenticating root.
Service samhain: Stopped
[root/sysadm_r/s0@contexts]#

[root/sysadm_r/s0@contexts]# grep -E "initrc|exec" /root/run_init.strace
open("/etc/selinux/refpolicy-mls/contexts/initrc_context", O_RDONLY) = 3
read(3, "system_u:system_r:initrc_t:s0-sl"..., 4096) = 43
open("/proc/self/task/2408/attr/exec", O_RDWR|O_LARGEFILE) = 3
write(3, "system_u:system_r:initrc_t:s0-sl"..., 43) = 43
[root/sysadm_r/s0@contexts]#
```

由此可见，当前进程（shell 的子进程，以切换到 run_init_t）从 initrc_context 文件中读取 SC 之后，随即将其写入自己的 attr/exec 文件，从而决定之后执行 execve 系统调用时所切换到的新 domain，即在 initrc_t 中执行启动脚本。

5，对于调用 init_ranged_system_domain 或者 init_system_domain 接口的 domain，还必须调用 init_use_script_pty 接口，因为由 run_init 启动的后台 domain 都使用 initrc_devpts_t 类型的 pty 设备。

在程序行为上，run_init 程序设置当前进程的/proc/self/attr/exec 为“initrc_t”后 execve open_init_pty 程序，使得在 initrc_t 中运行 open_init_pty 程序。后者调用 forkpty 函数分配新 pty 的设备并创建子进程，并使子进程使用新的 pty 设备。

在 policy.X 中调用 term_create_pty(initrc_t, initrc_devpts_t)接口：

```
interface(`term_create_pty',`
        gen_require(`
                type bsdpty_device_t, devpts_t, ptmx_t;
        ')

        dev_list_all_dev_nodes($1)
        allow $1 ptmx_t:chr_file rw_file_perms;

        allow $1 devpts_t:dir list_dir_perms;
        allow $1 devpts_t:filesystem getattr;
        dontaudit $1 bsdpty_device_t:chr_file { getattr read write };
        type_transition $1 devpts_t:chr_file $2;
')
```

其中 type_transition 规则使得当 open_init_pty 程序在/dev/pts/下创建新的 pty 设备时，相应设备的标签由默认的 devpts_t 转换为 initrc_devpts_t。

这样做的好处是，由 run_init 启动的后台进程 domain 都使用 initrc_devpts_t 标签的 pty 设备，而不是当前控制台的 user_devpts_t 设备。

5.3 Policy Store

在 host 上编译 refpolicy 包时，所有 pp 被链接、扩展为 policy.X 文件。而在 target 上并不知道当前 policy.X 是由哪些.pp 文件所生成，创建 policy store 的目的即为在 target 上重新执行由 pp 生成 policy.X 的过程，进而可以通过 semodule 命令来安装、卸载、更新任何一个 pp，或者使用 semanage 命令来动态地修改其属性。

secadm_r 使用如下命令创建 policy store：

```
cd /usr/share/selinux/refpolicy-mls/
ls *.pp | grep -v "base.pp" | xargs semodule -s refpolicy-mls -b base.pp -i
```

然后就会在/etc/selinux/refpolicy-mls/目录下创建 modules/active/目录：

```
[root/sysadm_r/s0@refpolicy-mls]# ls -lt modules/active/
total 3820
-rw-r--r-- 1 root root   31505 2012-01-05 03:31 base.pp
-rw------- 1 root root      32 2012-01-05 03:31 commit_num
-rw------- 1 root root  238227 2012-01-05 03:31 file_contexts          # == contexts/files/file_contexts
-rw-r--r-- 1 root root    8568 2012-01-05 03:31 file_contexts.homedirs
-rw------- 1 root root  242767 2012-01-05 03:31 file_contexts.template
-rw------- 1 root root    4540 2012-01-05 03:31 homedir_template
drwx------ 2 root root    8192 2012-01-05 03:31 modules               # /usr/share/selinux/refpolicy/*.pp
-rw------- 1 root root       0 2012-01-05 03:31 netfilter_contexts
-rw-r--r-- 1 root root 3291309 2012-01-05 03:31 policy.kern           # == policy/policy.X
-rw------- 1 root root      82 2012-01-05 03:31 seusers.final
-rw------- 1 root root     143 2012-01-05 03:31 users_extra
[root/sysadm_r/s0@refpolicy-mls]#
```

如果使用 semanage 修改 policy.X 的属性，比如修改 UNIX User 到 SELinux User 的映射，设置网络端口的 SC，修改 booleans 的值，修改 file_contexts 文件等，都会在 policy store 下创建相应的.local 文件，从而使得这种修改是持久的（permanent），而无须重新修改 refpolicy 源代码并重新编译安装（注意使用 chcon 修改的标签不是持久的，必须使用 semanage fcontext）。比如：

```
[root/sysadm_r/s0@active]# seclow "semanage port -a -t system_u:object_r:ssh_port_t:s0 -p tcp 222"
Password:
[root/sysadm_r/s0@active]#
[root/sysadm_r/s0@active]# cat ports.local | sed -e 's/#.*$//' | awk 'NF > 0'
portcon tcp 222 system_u:object_r:ssh_port_t:s0

[root/sysadm_r/s0@active]#
```

另外在创建 policy store 后还可以通过“semodule -l/i/r/u”命令来打印、安装、删除、更新一个 pp，参见附录 3 示例。

5.4 selinuxfs 目录树

（注，selinuxfs 的挂载点已经由/selinux/变成了/sys/fs/selinux/，但是 selinuxfs 目录树的结构以及各个文件的作用并没有发生太大变化，本小节暂时保留关于 selinuxfs 挂载点的描述。）

在传递“selinux=1”启动参数时，内核初始化函数（__initcall(init_sel_fs)）会注册 selinuxfs 驱动并调用 kern_mount 函数挂载它。SELinux 内核驱动通过它和用户态 SELinux-aware 应用程序通信。用户态应用程序可以借助 libselinux 库提供的 API 来访问该文件系统（而不是直接操作），selinuxfs 向用户态导出的文件如下：

```
[root/sysadm_r/s0@~]# cd /selinux/
[root/sysadm_r/s0@selinux]# ls -l
total 0
-rw-rw-rw-  1 root root    0 2012-01-05 05:42 access
dr-xr-xr-x  2 root root    0 2012-01-05 05:42 avc
dr-xr-xr-x  2 root root    0 2012-01-05 05:42 booleans               # 读写 boolean 的当前值和未来值
```

```
-rw-r--r--  1 root root    0 2012-01-05 05:42 checkreqprot
dr-xr-xr-x 83 root root    0 2012-01-05 05:42 class
--w-------  1 root root    0 2012-01-05 05:42 commit_pending_bools     # 使得所有 boolean 的未来值生效
-rw-rw-rw-  1 root root    0 2012-01-05 05:42 context
-rw-rw-rw-  1 root root    0 2012-01-05 05:42 create                   # 对应于 compute_create
-r--r--r--  1 root root    0 2012-01-05 05:42 deny_unknown
--w-------  1 root root    0 2012-01-05 05:42 disable
-rw-r--r--  1 root root    0 2012-01-05 05:42 enforce                  # 对应于 getenforce/setenforce
dr-xr-xr-x  2 root root    0 2012-01-05 05:42 initial_contexts         # Initial SID 字符串及其 SC
-rw-------  1 root root    0 2012-01-05 05:42 load
-rw-rw-rw-  1 root root    0 2012-01-05 05:42 member                   # 对应于 compute_member
-r--r--r--  1 root root    0 2012-01-05 05:42 mls
crw-rw-rw-  1 root root 1, 3 2012-01-05 05:42 null
-r--------  1 root root    0 2012-01-05 05:42 policy                   # 用于导出 policy.X
dr-xr-xr-x  2 root root    0 2012-01-05 05:42 policy_capabilities
-r--r--r--  1 root root    0 2012-01-05 05:42 policyvers
-r--r--r--  1 root root    0 2012-01-05 05:42 reject_unknown
-rw-rw-rw-  1 root root    0 2012-01-05 05:42 relabel                  # 对应于 compute_relabel
-r--r--r--  1 root root    0 2012-01-05 05:42 status
-rw-rw-rw-  1 root root    0 2012-01-05 05:42 user                     # 对应于 compute_user
[root/sysadm_r/s0@selinux]#
```

其中所有只读文件都用于向用户态导出 policy.X 及其运行状态的某些信息。而所有可读写的文件都是用户态机制查询 policy.X 的接口，通常用户态 SELinux-aware 程序向这些文件写入查询条件（比如访问主体和被访问对象的 SC），然后以阻塞方式读取 SELinuxfs 内核驱动查询 policy.X 的结果。参见 10.4 小节。

mls 和 policevers 为只读文件，如果当前使能 MLS 特性则 mls 返回值为 1，否则为 0。读取 policyvers 可以得到当前 SELinux 内核驱动所支持的 policy 最大版本号（不同的版本号对应不同的格式，用户态 SELinux 编译工具支持的最大版本号最好和 SELinux 内核驱动所支持的保持一致，如果比内核驱动所支持的最大版本号还大，则用户态 libsepol 库会在装载 policy.X 时将其 downgrade 为较低版本）：

```
[root/sysadm_r/s0@selinux]# cat mls
1[root/sysadm_r/s0@selinux]#
[root/sysadm_r/s0@selinux]# cat policyvers
26[root/sysadm_r/s0@selinux]#
```

通过 enforce 文件可以得到或设置当前 SELinux 的工作状态，最好使用 getenforce/setenforce 命令而不是直接访问该文件（注意只有 secadm_r 才能向该文件写入或者调用 setenforce，因为 sysadm_t 没有对 security_t 的 setenforce 权限）。另外使用 strace 可以看到 sestatus 命令也访问/selinux/下的文件：

```
[root/sysadm_r/s0@selinux]# cat enforce
1[root/sysadm_r/s0@selinux]#
[root/sysadm_r/s0@selinux]# strace -e trace=open,read sestatus  2>/root/sestatus.strace
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 26
Policy from config file:        refpolicy-mls
[root/sysadm_r/s0@selinux]#
[root/sysadm_r/s0@selinux]# cat ~/sestatus.strace | grep -E "/selinux/"
open("/selinux/enforce", O_RDONLY|O_LARGEFILE) = 3
open("/etc/selinux/config", O_RDONLY|O_LARGEFILE) = 3
open("/selinux/policyvers", O_RDONLY|O_LARGEFILE) = 3
open("/etc/selinux/config", O_RDONLY|O_LARGEFILE) = 3
[root/sysadm_r/s0@selinux]#
```

booleans/目录下为以各个 boolean 名字命名的文件，读取之可以得到相应 boolean 的当前值和未来值（pending value），secadm_r 可以修改若干 boolean 的将来值，然后向 commit_pending_bools 中写入 1，从而以原子方式同时改变多个布尔变量的当前值：

```
[root/sysadm_r/s0@selinux]# cd booleans/
[root/sysadm_r/s0@booleans]# ls
secure_mode  secure_mode_insmod  secure_mode_policyload
[root/sysadm_r/s0@booleans]#
[root/sysadm_r/s0@booleans]# cat secure_mode
0 0
[root/sysadm_r/s0@booleans]# cat secure_mode_insmod
0 0
[root/sysadm_r/s0@booleans]# cat secure_mode_policyload
0 0
[root/sysadm_r/s0@booleans]#
```

注意，由于在 refpolicy 中应用了 tunable 关键字，所以只有真正的 boolean 才会被写入 policy.X；而各个 tunable 所控制的实际有效的规则分支已经被彻底地写入 policy.X（被“一劳永逸”地使能，而无法、也不需要在运行时切换生效的规则分支）

系统启动时 init（可以通过 initramfs）调用 load_policy 程序，它通过/selinux/load 接口将 policy.X 装载入内核，本质操作等同于如下 dd 操作：

```
dd if=/etc/selinux/refpolicy-mls/policy/policy.X of=/selinux/load bs=xxxx
```

5.5 /proc/pid/attr/目录树

进程的/proc/pid/attr/目录结构用于导出内核中该进程所使用的安全属性数据结构，比如：

```
[root/sysadm_r/s0@~]# ps -eZ | grep login
system_u:system_r:local_login_t:s0-s15:c0.c1023 1360 ? 00:00:00 login
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# cd /proc/1360/attr
[root/sysadm_r/s0@attr]# ls -l
total 0
-rw-rw-rw- 1 root root 0 Jan  5 09:55 current                  # 该进程当前的 SC
-rw-rw-rw- 1 root root 0 Jan  5 09:58 exec                     # 该进程在下一次执行 execve 时切换到的 SC
-rw-rw-rw- 1 root root 0 Jan  5 09:58 fscreate                 # 新创建文件的 SC
-rw-rw-rw- 1 root root 0 Jan  5 09:58 keycreate                        # 新创建 key 的 SC
-r--r--r-- 1 root root 0 Jan  5 09:58 prev                     # 最近一次 domain transition 之前的 SC
-rw-rw-rw- 1 root root 0 Jan  5 09:58 sockcreate               # 新创建 socket 的 SC
[root/sysadm_r/s0@attr]#
[root/sysadm_r/s0@attr]# cat current
system_u:system_r:local_login_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]# cat prev
system_u:system_r:getty_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]# cat exec
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]#
[root/sysadm_r/s0@attr]# ps axj | grep 1360
    1  1360  1360  1360 ?           -1 Ss       0   0:00 login -- root
 1360  1376  1376  1376 ttyS0     1539 Ss       0   0:01 -bash
 1376  1540  1539  1376 ttyS0     1539 S+       0   0:00 grep 1360
[root/sysadm_r/s0@attr]#
[root/sysadm_r/s0@attr]# ps axjZ | grep ttyS0
```

```
root:sysadm_r:sysadm_t:s0-s15:c0.c1023 1360 1376 1376 1376 ttyS0  1548 Ss       0   0:01 -bash
root:sysadm_r:sysadm_t:s0-s15:c0.c1023 1376 1548 1548 1376 ttyS0  1548 R+       0   0:00 ps axjZ
root:sysadm_r:sysadm_t:s0-s15:c0.c1023 1376 1549 1548 1376 ttyS0  1548 S+       0   0:00 grep ttyS0
[root/sysadm_r/s0@attr]#
[root/sysadm_r/s0@attr]# cat /proc/1376/attr/current
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]# cat /proc/1376/attr/prev
system_u:system_r:local_login_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]#
```

由此可见：
1，1360 为 login 后台进程，其当前处于 local_login_t，之前处在 getty_t，下一次执行 execve 系统调用时切换到 sysadm_t。
由 init 创建的某个 mingetty 子进程（1360）在打开/dev/ttyS0 设备后执行/bin/login 程序，domain 也随之从 getty_t 切换到 local_login_t；
2，login 进程（1360）在用户登录后创建子进程（1376）并执行用户的登录 shell。由于/proc/1360/attr/exec 指定 sysadm_t，所以登录 shell 进程迁移到 sysadm_t。

可以进一步观察 mingetty 和 init 后台进程的 domain transition 情况：

```
[root/sysadm_r/s0@attr]# ps axjZ | grep getty_t | head -n 1
system_u:system_r:getty_t:s0-s15:c0.c1023 1 1361 1361 1361 tty2   1361 Ss+      0   0:00 /sbin/mingetty
tty2
[root/sysadm_r/s0@attr]#
[root/sysadm_r/s0@attr]# cat /proc/1361/attr/current
system_u:system_r:getty_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]# cat /proc/1361/attr/prev
system_u:system_r:init_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]# cat /proc/1361/attr/exec
[root/sysadm_r/s0@attr]#
[root/sysadm_r/s0@attr]# cat /proc/1/attr/current
system_u:system_r:init_t:s0-s15:c0.c1023
[root/sysadm_r/s0@attr]# cat /proc/1/attr/prev
system_u:system_r:kernel_t:s15:c0.c1023
[root/sysadm_r/s0@attr]# cat /proc/1/attr/exec
[root/sysadm_r/s0@attr]#
```

由此可见：
1，init 进程在装载 policy.X 之前处在 kernel_t 中；
2，init 进程在装载 policy.X 之后会重新执行自己（参见 6.2 小节），从而切换到 init_t 中；
3，init 进程创建的子进程（1361）在执行/sbin/mingetty 程序时由 init_t 切换到 getty_t；

最后，修改/proc/*/attr/下的文件需要对 process 类的相应能力。可以使用如下 auditallow 规则来监控对相应文件的修改：

```
auditallow domain self:process setcurrent;
auditallow domain self:process setexec;
auditallow domain self:process setfscreate;
auditallow domain self:process setsockcreate;
```

audit-test 包引入的 trustedprograms/test_setcon.bash 测试用例就需要使能上述规则，从而监控对 attr/下文件的写操作。

6. SELinux 的安装

6.1 Ubuntu 上 SELinux 的安装

在 ubuntu 上，可以使用如下命令方便快捷地安装 SELinux：

```
sudo apt-get install selinux selinux-utils -y
```

安装的最后会提示用户重启系统，以便给整个文件系统设置 SC。之后系统启动时 init 会执行 initramfs 存档中的/etc/initramfs-tools/scripts/init-bottom/_load_policy 脚本来挂载 selinuxfs，装载 policy.X。

6.1.1 initramfs

initramfs 为保存在辅存上的 gzipped cpio 存档。内核中有 initramfs 文件系统驱动，系统启动时将辅存上 initramfs 映像拷贝到一块 RAM Disk 区域，然后挂载 initramfs 文件系统，运行其中的脚本启动 udev，设置 console，装载 policy.X 等。

PC 机上 initramfs 映像即为/boot/initrd.img-2.6.24-19-generic，由 update-initramfs 工具根据 /etc/initramfs-tools/scripts/和/usr/share/initramfs-tools/scripts/下的脚本创建。如果修改了上述目录下的某个脚本，则需要运行 update-initramfs 更新当前 initramfs 映像，参见手册。

6.1.2 Targeted Policy

Ubuntu 发行版上安装的 SELinux policy 为“Targeted”类型，用户运行的应用程序都在 unconfined_t 中运行，而所有和网络应用相关，或者属主为 root 的 setuid/setgid 的程序都在各自定义的 domain 中运行，因为这些程序容易受到外部攻击，或者关系到整个系统的安全性。由于 SELinux 只赋予 confined domain 能够运行的最小权限，因此处在相应 domain 中的进程对系统可能的危害被控制到最低。另外内核线程和系统后台服务例程分别运行在 kernel_t 和 initrc_t 中，它们也是 unconfined 的。

```
cao@cao-desktop:/etc$ id -Z
unconfined_u:unconfined_r:unconfined_t                  # unconfined user
cao@cao-desktop:/etc$ ps axjZ | grep "unconfined_t" | wc -l
105
cao@cao-desktop:/etc$ ps axjZ | grep  -v "unconfined_t" | awk '{print $1}' | sort | uniq -c | sort -nr
     43 system_u:system_r:kernel_t                      # 内核线程，unconfined
     28 system_u:system_r:initrc_t                      # 系统后台进程，unconfined
      4 system_u:system_r:xdm_t
      3 system_u:system_r:local_login_t
      3 system_u:system_r:getty_t
      1 system_u:system_r:xdm_xserver_t
      1 system_u:system_r:syslogd_t
      1 system_u:system_r:sshd_t
      1 system_u:system_r:restorecond_t
      1 system_u:system_r:mount_t
      1 system_u:system_r:klogd_t
      1 system_u:system_r:init_t
      1 system_u:system_r:dhcpc_t
      1 system_u:system_r:cupsd_t
      1 staff_u:sysadm_r:sysadm_t
      1 staff_u:staff_r:staff_t
      1 LABEL
```

Targeted policy 并不对 unconfined_t 类型做更多的约束，实际上只有 UNIX DAC 机制起作用，能够满足大部分普通桌面用户的安全需求，也能大大降低 policy 的复杂度。比如当前 Ubuntu 10.04.3 上只有 50 个 pp 被安装：

```
cao@cao-laptop:~$ cat /etc/issue
Ubuntu 10.04.3 LTS \n \l

cao@cao-laptop:~$ sudo semodule -l | wc -l
[sudo] password for cao:
50
cao@cao-laptop:~$
```

与之相比最新 refpolicy 中实现了 338 个 pp，而且数量还一直在增加。

6.2 某发行版上 SELinux 的启动过程（Revisited）

6.2.1 selinuxfs 的挂载

如果在内核启动命令行指定“selinux=1”，则在内核启动函数（__initcall(init_sel_fs)）中注册并挂载 selinuxfs。

6.2.2 判断内核是否使能了 SELinux

libselinux 定义了如下构造函数：

```
static void init_lib(void) __attribute__ ((constructor));
static void init_lib(void)
{
        selinux_page_size = sysconf(_SC_PAGE_SIZE);
        init_selinuxmnt();
}
```

则该构造函数将在所有链接了 libselinux 的应用程序的 main 函数执行前被调用。它首先通过 sysconf 函数读取当前系统上页面的大小并保存到 selinux_page_size 全局变量中，作为 libselinux 中许多访问 selinuxfs 的函数所分配的临时缓冲区大小。

然后调用 init_selinuxmnt 函数，通过如下过程检查并设置 selinux_mnt 指针变量，它指向 selinuxfs 挂载点路径字符串：
1，如果 selinux_mnt 不为空，则表示 selinuxfs 已经被挂载，可立即退出；
2，验证“/sys/fs/selinux”和“/selinux”路径上是否已经挂载了 selinuxfs（通过 statfs 函数读取所挂载文件系统的 magic）。如果是，则复制挂载点路径字符串并设置 selinux_mnt，然后退出；
3，否则，首先通过读取/proc/filesystems 函数检查内核是否支持 selinuxfs，然后读取/proc/mounts 检查当前 selinuxfs 是否已经被挂载上。如果是，则复制挂载点路径字符串并设置 selinux_mnt，然后退出；

至此，如果 SELinux 被使能，则 selinuxfs 一定被挂载在由 selinux_mnt 所指向的那个路径上。反之，如果 selinux_mnt 为 NULL，则说明当前内核没有使能 SELinux。于是，用户态 SELinux-aware 应用程序可以通过调用 is_selinux_enabled 函数检查该变量，从而判定 SELinux 当前是否被使能。比如 init 程序。

```
int is_selinux_enabled(void)
{
        int enabled = 0;
```

```
        security_context_t con;

        /* init_selinuxmnt() gets called before this function. We
         * will assume that if a selinux file system is mounted, then
         * selinux is enabled. */
        if (selinux_mnt) {
                /* Since a file system is mounted, we consider selinux
                 * enabled. If getcon_raw fails, selinux is still enabled.
                 * We only consider it disabled if no policy is loaded. */
                enabled = 1;
                if (getcon_raw(&con) == 0) {
                        if (!strcmp(con, "kernel"))
                                enabled = 0;
                        freecon(con);
                }
        }

        return enabled;
}
hidden_def(is_selinux_enabled)
```

如上文所述，如果 selinux_mnt 指向 NULL 则返回 0，表示 SELinux 尚未被使能。否则默认 SELinux 被使能，返回 1。除非 getcomn_raw 函数返回“kernel”字符串，即表示 policy.X 尚未被装载，此时返回 0。

6.2.3 init 程序

对于使用 sysvinit 包（比如 sysvinit-2.87）的系统，如果在内核命令行指定“selinux=1”则在系统启动时由/sbin/init 程序负责装载 policy.X。相关代码如下所示：

```
#ifdef WITH_SELINUX
        if (getenv("SELINUX_INIT") == NULL && is_selinux_enabled()) {
                putenv("SELINUX_INIT=YES");
                if ((&enforce) == 0) {
                        execv(myname, argv);                    # init 在装载 policy.X 后重新执行自己
                } else {
                        if (enforce > 0) {
                              /* SELinux in enforcing mode but load_policy failed */
                              /* At this point, we probably can't open /dev/console, so log() won't work */
                                fprintf(stderr,"Unable to load SELinux Policy. Machine is in enforcing
mode. Halting now.\n");
                                exit(1);
                        }
                }
        }
#endif

        /* Start booting. */
        argv0 = argv[0];
        argv[1] = NULL;
        setproctitle("init boot");
        init_main(df1_level);

        /*NOTREACHED*/
        return 0;
```

首先，在 sysvinit.spec 中的%build phase 就必须指定 WITH_SELINUX=yes 选项，从而使得 init/sulogin 程序和 libselinux/libsepol 相编译、链接。然后，init 程序在启动过程中检查内核是否已经使能

SELinux（is_selinux_enabled 函数返回 1），并且当前环境变量中是否设置了“SELINUX_INIT”。如果使能了 SELinux 但是“SELINUX_INIT”未定义，则调用 libselinux 库函数 selinux_init_load_policy，它调用 mount 函数再次挂载 selinuxfs 以确定其挂载点（通常 selinux 已经由内核在启动时挂载，则这里 mount 会返回 EBUSY），然后向/selinux/load 文件写入 policy.X。最后如果一切顺利，则 init 程序会通过 execv 系统调用再次执行自己从而使其运行在正确的 SC 中。反之，如果失败且用户指定 enforcing=1，则显然无法继续，只能使用 exit 退出。

注意，/sbin/init 文件的标签为 init_exec_t，并且 init 进程初次启动时处于 kernel_t 中（由 Initial SID 决定）。然后在再次运行时，由于相应 type_transition 规则生效而进入 init_t。参见 7.1.5 小节。

6.2.4 系统启动脚本

在/etc/inittab 中指定系统初始化时调用 rc.sysinit 脚本：

```
# System initialization.
si::sysinit:/etc/rc.d/rc.sysinit
```

而它又会调用/sbin/start_udev 启动 udev。如上文所述，由于 selinuxfs 已经在内核启动时被挂载，所以在 rc.sysinit 和 start_udev 脚本中才可以通过读取/proc/mounts 来判断 SELinux 是否被使能：

```
# Check SELinux status
selinuxfs="$(fstab_decode_str `LC_ALL=C awk '/ selinuxfs / { print $2 }' /proc/mounts`)"
SELINUX_STATE=
if [ -n "$selinuxfs" ] && [ "`cat /proc/self/attr/current`" != "kernel" ]; then
        if [ -r "$selinuxfs/enforce" ] ; then
                SELINUX_STATE=`cat "$selinuxfs/enforce"`
        else
                # assume enforcing if you can't read it
                SELINUX_STATE=1
        fi
fi
```

在上面的脚本中 selinuxfs 变量的值即为 selinuxfs 挂载点字符串。如果它非空，且当前进程所处的 domain 不是“kernel” Initial SID，则根据 selinuxfs 中的 enforce 接口文件的返回值来确定当前处于 Enforcing 或者 Permissive 模式；如果 enforce 文件无法读取，则默认为 Enforcing 模式。

否则，如果当前进程仍然处在 kernel Initial SID 中，则说明 policy.X 尚未被装载。由于策略都不存在，所以“施加策略的模式（Enforcing 或 Permissive）”的说法也就没有任何意义。此时 SELINUX_STATE 变量为空。

7．为应用程序开发新的 pp

7.1 Object 的标签由谁决定？（Revisited）

文件 SC 中 user 部分由其创建者 SC 的 user 部分决定，role 恒定为 object_r，而它的 type 和所在的文件系统特性相关。

7.1.1 使用 fs_use_xxx 语法定义的文件系统

refpolicy/policy/modules/kernel/*.te 使用如下语法定义相应文件系统的初始安全上下文（即相应文件系统超级块 sbsec->sid）：

```
fs_use_xattr                    # 所有支持 xattr 的文件系统，比如 ext2/jffs2
fs_use_trans                    # 比如 devpts/tmpfs
fs_use_task                     # 比如 pipefs/sockfs，使用创建者的 SID
```

在词法分析函数 define_fs_use 中分配相应的 ocontext_t 数据结构，其中 u.name 即为文件系统的名称；v.behavior 即为文件系统标签的确定方式（比如 SECURITY_FS_USE_XATTR 等），context[0]即为相应规则中指定的 SC 字符串。

相应 context_t 数据结构最终加入 policydb->ocontexts[OCON_FSUSE]队列并写入 policy.X。在装载 policy.X 时创建其内核描述符，并在挂载某种文件系统时，在内核 policydb->ocontexts[OCON_FSUSE]队列中查询和 u.name 匹配的元素，向 sidtab 注册其 context[0]，返回 sid[0]作为相应文件系统超集块的 SID，根据 v.behavior 确定 sbsec->behavior（参见下文 security_fs_use 函数分析）。

使用 fs_use_xxx 规则（以及 sysfs，尽管它使用 genfscon 规则）定义 SC 的文件系统，其 sbsec->flags 中 SE_SBLABELSUPP 标志有效，它对应于/proc/mounts 导出结果中的“seclabel”标志，比如：

```
[root/sysadm_r/s0@~]# grep "seclabel" /proc/mounts
/dev/root / ext2 rw,seclabel,relatime,errors=continue 0 0                                  # fs_use_xattr
/sys /sys sysfs rw,seclabel,relatime 0 0                                                   # sysfs
devpts /dev/pts devpts rw,seclabel,relatime,gid=5,mode=620 0 0                             # fs_use_trans
tmpfs /dev/shm tmpfs rw,rootcontext=system_u:object_r:tmpfs_t:s0,seclabel,relatime 0 0 # fs_use_trans
[root/sysadm_r/s0@~]#
```

7.1.1.1 使用 xattr 保存 SC 的文件系统

使用 fs_use_xattr 规则声明那些使用扩展属性来保存文件 SC 的文件系统：

```
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$ grep fs_use_xattr kernel/*
kernel/filesystem.te:fs_use_xattr btrfs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr encfs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr ext2 gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr ext3 gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr ext4 gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr ext4dev gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr gfs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr gfs2 gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr gpfs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr jffs2 gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr jfs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr lustre gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_xattr xfs gen_context(system_u:object_r:fs_t,s0);
```

```
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$
```

初始化此类文件系统中新文件的 inode 时，在 inode_doinit_with_dentry 函数中检查 sbsec->behavior，如果为 SECURITY_FS_USE_XATTR，则从“security.selinux”扩展属性中读取 SC 字符串，并向 sditab 注册，返回的 sid 保存在 isec->sid 中。

这些文件系统中文件的 SC 可由如下方式决定：

1，如果文件已经存在，且当前/etc/selinux/refpolicy-mls/contexts/files/file_contexts 中存在其路径名的精确匹配，则在执行 restorecon 时它被打上 file_contexts 中指定的标签。比如：

```
[root/sysadm_r/s0@~]# which checkpolicy
/usr/bin/checkpolicy
[root/sysadm_r/s0@~]# grep checkpolicy /etc/selinux/refpolicy-mls/contexts/files/file_contexts
/usr/bin/checkpolicy    --      system_u:object_r:checkpolicy_exec_t:s0
[root/sysadm_r/s0@~]# restorecon -v /usr/bin/checkpolicy
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ls -Z /usr/bin/checkpolicy
-rwxr-xr-x  root root system_u:object_r:checkpolicy_exec_t:s0 /usr/bin/checkpolicy
[root/sysadm_r/s0@~]#
```

注意，restorecon 命令的“-v”选项仅检查文件的当前标签是否正确，如果没有输出，则表示和 file_contexts 中指定的一致。

file_contexts 文件的内容是由相应 Makefile 的$(fc)相应规则生成的，为当前所有 pp 的.fc 文件内容的集合。如果不存在某个文件路径名的精确匹配，则采用缺省匹配。

另外，在 file_contexts 文件中可以使用“<<none>>”来显式地指定某些文件不存在静态标签，而应该在运行时确定，比如：

```
[root/sysadm_r/s0@~]# grep -E "^/selinux" /etc/selinux/refpolicy-mls/contexts/files/file_contexts
/selinux/.*                     <<none>>
/selinux                -d      <<none>>
[root/sysadm_r/s0@~]# grep -E ".*-s.*<<none>>" /etc/selinux/refpolicy-mls/contexts/files/file_contexts
/tmp/\.ICE-unix/.*      -s      <<none>>
/tmp/\.X11-unix/.*      -s      <<none>>
[root/sysadm_r/s0@~]#
```

由下文可知，运行时 selinuxfs 目录树的标签由 genfscon 规则指定。而/tmp/.X11-unix/下的所有 AF_UNIX 套接字文件的标签，默认由创建者决定（也可由相应的 type_transition 规则决定，或者由应用程序的/proc/self/attr/sockcreate 设置决定）。

2，如果文件已经存在，但是在上述 file_contexts 文件中没有文件路径名的精确定义，则在 restorecon 时它将被打上其所在父目录的标签。比如/bin:/sbin:/usr/bin:/usr/sbin 下的许多可执行程序的标签都是 bin_t。

3，对于在运行时动态创建的文件，如果当前 policy.X 中存在相应的 type_transition 规则，则新文件的标签由该规则明确定义。比如当前 policy.X 中存在如下规则：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s sysadm_t -t tmp_t -c file"
Password:
Found 1 semantic te rules:
   type_transition sysadm_t tmp_t : file user_tmp_t;
[root/sysadm_r/s0@~]#
```

它的含义是，sysadm_t 在创建新的正规文件时，如果它的默认标签为 tmp_t，则自动转换为 user_tmp_t。/tmp 目录的标签为 tmp_t，所以其下文件的默认标签也为 tmp_t。但是由于上述 type_transition 规则的存在当 sysadm_t 在/tmp 下创建新文件时，其实际标签为 user_tmp_t：

```
[root/sysadm_r/s0@~]# ls -Zd /tmp
drwxrwxrwt  root root system_u:object_r:tmp_t:s0-s15:c0.c1023 /tmp
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# touch /tmp/sysadm_file
[root/sysadm_r/s0@~]# ls -Z /tmp
-rw-r--r--  root root root:object_r:user_tmp_t:s0       sysadm_file
[root/sysadm_r/s0@~]#
```

另外，可以使用 compute_create 命令查询上述 type_transition 规则：

```
[root/sysadm_r/s0@~]# compute_create `id -Z` system_u:object_r:tmp_t:s0-s15:c0.c1023 file
root:object_r:user_tmp_t:s0
[root/sysadm_r/s0@~]#
```

4，如果在应用程序源代码中使用 setfscreatecon 函数，也可以指定新创建文件的标签。这样就无须在相应 pp 中使用 type_transition 规则了。但是必须修改源码（使得相应应用程序成为 SELinux-aware 的），而且可能存在许多不同的版本，所以还是在相应 pp 中使用 type_transition 规则能够“一劳永逸”地解决该问题。

另外，通常也只有 SELinux Utility 才需要调用 setfscreatecon 函数，比如 policycoreutils/sandbox 等。

7.1.1.2 基于 Transition SID 的文件系统

一些虚拟文件系统基于 type_transition 规则来确定其中文件的标签，在 refpolicy 源代码中使用 fs_use_trans 规则来声明这些虚拟文件系统的**默认标签**，比如：

```
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$ grep fs_use_trans kernel/*
kernel/devices.te:      fs_use_trans devtmpfs   gen_context(system_u:object_r:device_t,s0);
kernel/filesystem.te:   fs_use_trans hugetlbfs  gen_context(system_u:object_r:hugetlbfs_t,s0);
kernel/filesystem.te:   fs_use_trans mqueue     gen_context(system_u:object_r:tmpfs_t,s0);
kernel/filesystem.te:   fs_use_trans shm        gen_context(system_u:object_r:tmpfs_t,s0);
kernel/filesystem.te:   fs_use_trans tmpfs      gen_context(system_u:object_r:tmpfs_t,s0);
kernel/terminal.te:     fs_use_trans devpts     gen_context(system_u:object_r:devpts_t,s0);
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$
```

（或者可以使用“seinfo --fs_use | grep fs_use_trans”命令得到所有此类文件系统列表）

初始化此类文件系统中新文件的 inode 时，在 inode_doinit_with_dentry 函数中检查 sbsec->behavior，如果为 SECURITY_FS_USE_TRANS，则从当前文件的创建者 sid（isec->task_sid）和文件系统 sid（sbsec->sid）出发，检查是否有匹配的 type_transition 规则。如果有，则按照该规则的定义来设置新文件的 isec->sid。否则它默认等于文件系统的 sid（sbsec->sid）。

由此可见，devpts 文件系统的默认标签为 devpts_t，可以用“ls -Zd /dev/pts”来验证：

```
[root/sysadm_r/s0@~]# grep devpts /proc/mounts
devpts /dev/pts devpts rw,seclabel,relatime,gid=5,mode=620 0 0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ls -Zd /dev/pts
```

```
drwxr-xr-x  root root system_u:object_r:devpts_t:s0-s15:c0.c1023 /dev/pts
[root/sysadm_r/s0@~]#
```

由于存在如下 type_transition 规则（userdom_base_user_template > term_create_pty）：

```
type_transition staff_t devpts_t : chr_file user_devpts_t;
```

则当 staff_t 通过 ssh 登录时，其 controlling terminal 为/dev/pts/0，可以看到该设备的标签为 usr_devpts_t：

```
[root/staff_r/s0@~]# seclow "sesearch -SCT -s staff_t -t devpts_t -c chr_file"
Password:
Found 1 semantic te rules:
   type_transition staff_t devpts_t : chr_file user_devpts_t;
[root/staff_r/s0@~]#
[root/staff_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_devpts_t:s0   /dev/pts/0
[root/staff_r/s0@~]#
```

7.1.1.3 基于创建者 SID 的文件系统

使用 fs_use_task 规则来声明那些使用创建者 SID 作为整个文件系统 SC 的文件系统：

```
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$ grep fs_use_task kernel/*
kernel/filesystem.te:fs_use_task eventpollfs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_task pipefs gen_context(system_u:object_r:fs_t,s0);
kernel/filesystem.te:fs_use_task sockfs gen_context(system_u:object_r:fs_t,s0);
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$
```

初始化此类文件系统中新文件的 inode 时，在 inode_doinit_with_dentry 函数中检查 sbsec->behavior，如果为 SECURITY_FS_USE_TASK，则将文件创建者的 sid（isec->task_sid）直接作为新文件的 sid（isec->sid）

7.1.2 Generalized Security Context Labeling

另外一些虚拟文件系统，整个文件系统结构的 SC 都通过一条 genfscon 规则来确定，比如：

```
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$ grep genfscon kernel/selinux.te
genfscon selinuxfs / gen_context(system_u:object_r:security_t,s0)
genfscon securityfs / gen_context(system_u:object_r:security_t,s0)
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$
```

那么 selinuxfs 所在的整个/selinux/目录结构的 SC 都为“system_u:object_r:security_t:s0”.

相应词法分析函数 define_genfs_context 将创建对应的 ocontext_t 和 genfs_t 数据结构，将文件系统名称保存在 genfs_t 的 fstype 中，根目录名称保存到 ocontext_t 的 u.name 中，安全上下问字符串保存到 context[0]中。最终组织到 policydb->genfs 队列中。

也可以使用“seinfo --genfscon=xxxx”来得到这些文件系统的 SC：

```
[root/sysadm_r/s0@~]# newrole -r secadm_r -p
Password:
[root/secadm_r/s0@~]# seinfo --genfscon=selinuxfs
   genfscon selinuxfs /      system_u:object_r:security_t:s0
[root/secadm_r/s0@~]# seinfo --genfscon=sysfs
```

```
   genfscon sysfs /      system_u:object_r:sysfs_t:s0
[root/secadm_r/s0@~]# seinfo --genfscon=proc
   genfscon proc /sys/kernel/modprobe      system_u:object_r:sysctl_modprobe_t:s0
   genfscon proc /sys/kernel/hotplug      system_u:object_r:sysctl_hotplug_t:s0
   genfscon proc /sys/net/unix      system_u:object_r:sysctl_net_unix_t:s0
   genfscon proc /fs/openafs      system_u:object_r:proc_afs_t:s0
   genfscon proc /sys/crypto      system_u:object_r:sysctl_crypto_t:s0
   genfscon proc /sys/kernel      system_u:object_r:sysctl_kernel_t:s0
   genfscon proc /kallsyms      system_u:object_r:system_map_t:s0
   genfscon proc /sysvipc      system_u:object_r:proc_t:s0
   genfscon proc /net/rpc      system_u:object_r:sysctl_rpc_t:s0
   genfscon proc /sys/net      system_u:object_r:sysctl_net_t:s0
   genfscon proc /sys/dev      system_u:object_r:sysctl_dev_t:s0
   genfscon proc /mdstat      system_u:object_r:proc_mdstat_t:s0
   genfscon proc /sys/fs      system_u:object_r:sysctl_fs_t:s0
   genfscon proc /sys/vm      system_u:object_r:sysctl_vm_t:s0
   genfscon proc /kcore      system_u:object_r:proc_kcore_t:s15:c0.c1023
   genfscon proc /mtrr      system_u:object_r:mtrr_device_t:s0
   genfscon proc /kmsg      system_u:object_r:proc_kmsg_t:s15:c0.c1023
   genfscon proc /net      system_u:object_r:proc_net_t:s0
   genfscon proc /xen      system_u:object_r:proc_xen_t:s0
   genfscon proc /sys      system_u:object_r:sysctl_t:s0
   genfscon proc /irq      system_u:object_r:sysctl_irq_t:s0
   genfscon proc /      system_u:object_r:proc_t:s0
[root/secadm_r/s0@~]#
```

注意 genfscon 规则中路径部分为相对于挂载点的相对路径，比如 proc 文件系统挂载到/proc，则/proc/irq 文件的 label 为 sysctl_irq_t。

注意，除了 sysfs 文件系统之外、使用 genfscon 规则确定整个文件系统 SC 的文件系统，由于相应文件系统驱动中没有实现访问扩展属性的方法，其 sbsec->flags 中 SE_SBLABELSUPP 标志被清除。相应地从/proc/mounts 导出结果中没有“seclabel”标志。比如：

```
[root/sysadm_r/s0@~]# grep -v "seclabel" /proc/mounts
rootfs / rootfs rw 0 0
none /selinux selinuxfs rw,relatime 0 0
/proc /proc proc rw,relatime 0 0
/proc/bus/usb /proc/bus/usb usbfs rw,relatime 0 0
none /proc/sys/fs/binfmt_misc binfmt_misc rw,relatime 0 0
none /var/lib/nfs/rpc_pipefs rpc_pipefs rw,relatime 0 0
[root/sysadm_r/s0@~]#
```

7.1.4 Mount-Point Labeling

在挂载一个文件系统时可以通过“context=”选项指定挂载点以及整个文件系统的标签，无论对于已经存在的文件还是新创建的文件（实现原理参见 selinux_inode_init_security 函数和 inode_doinit_with_dentry 函数，直接根据 sbsec->mntpoint_sid 来设置 isec->sid）。而且 Mount-Point Labeling 能够重载当前 policy.X 中相关文件标签的设置，无论在 file_contexts 中还是 type_transition 规则。

比如，首先创建新目录/dev/shm，观察它的标签为 device_t：

```
[root/sysadm_r/s0@~]# umount /dev/shm
[root/sysadm_r/s0@~]# rm -fr /dev/shm
[root/sysadm_r/s0@~]# mkdir /dev/shm
[root/sysadm_r/s0@~]# ls -Zd /dev/shm
```

```
drwxr-xr-x  root root root:object_r:device_t:s0        /dev/shm
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# restorecon -v /dev/shm
restorecon reset /dev/shm context root:object_r:device_t:s0->system_u:object_r:tmpfs_t:s0
[root/sysadm_r/s0@~]#
```

（restorecon -v 结果说明 file_contexts 中它的标签为 tmpfs_t）

然后挂载 tmpfs 到该目录，同时用 context 选项指定该文件系统目录树的缺省安全上下文：

```
[root/sysadm_r/s0@~]# mount -t tmpfs tmpfs /dev/shm -o rw,context="system_u:object_r:tmpfs_t:s0"
[root/sysadm_r/s0@~]# grep tmpfs /proc/mounts
tmpfs /dev/shm tmpfs rw,context=system_u:object_r:tmpfs_t:s0,relatime 0 0
[root/sysadm_r/s0@~]# restorecon -v /dev/shm
[root/sysadm_r/s0@~]#
```

注意，使用“Mount Point Labeling”方式确定整个文件系统 SC 时，即使文件系统能够提供安全属性，但是由于被 sbsec->mntpoint_sid 所重载，所以相应文件系统 sbsec->flags 中清除 SE_SBLABELSUPP 标志。于是在/proc/mounts 导出的结果中没有“seclabel”标志。

可以验证，尽管存在相应的 type_transition 规则，由于在挂载时使用了 context 选项，sysadm_t 在其中新创建文件的标签将继承挂载点的标签，而不是由 type_transition 规则决定：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s sysadm_t -t tmpfs_t"
Password:
Found 5 semantic te rules:
   type_transition sysadm_t tmpfs_t : file user_tmpfs_t;
   type_transition sysadm_t tmpfs_t : dir user_tmpfs_t;
   type_transition sysadm_t tmpfs_t : lnk_file user_tmpfs_t;
   type_transition sysadm_t tmpfs_t : sock_file user_tmpfs_t;
   type_transition sysadm_t tmpfs_t : fifo_file user_tmpfs_t;
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# id -Z
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]# touch /dev/shm/sysadm
[root/sysadm_r/s0@~]# mkdir /dev/shm/sysadm_dir
[root/sysadm_r/s0@~]# ls -Z /dev/shm
-rw-r--r--  root root system_u:object_r:tmpfs_t:s0     sysadm
drwxr-xr-x  root root system_u:object_r:tmpfs_t:s0     sysadm_dir
[root/sysadm_r/s0@~]#
```

反之，如果在 mount 命令行没有指定“context”选项，则 SELinux 规则库中相应的 type_transition 规则生效：

```
[root/sysadm_r/s0@~]# umount /dev/shm
[root/sysadm_r/s0@~]# mount -t tmpfs tmpfs /dev/shm
[root/sysadm_r/s0@~]# grep shm /proc/mounts
tmpfs /dev/shm tmpfs rw,rootcontext=system_u:object_r:tmpfs_t:s0,seclabel,relatime 0 0
[root/sysadm_r/s0@~]# ls -Z /dev/shm
[root/sysadm_r/s0@~]# touch /dev/shm/sysadm
[root/sysadm_r/s0@~]# mkdir /dev/shm/sysadm_dir
[root/sysadm_r/s0@~]# ls -Z /dev/shm
-rw-r--r--  root root root:object_r:user_tmpfs_t:s0    sysadm
drwxr-xr-x  root root root:object_r:user_tmpfs_t:s0    sysadm_dir
[root/sysadm_r/s0@~]#
```

注意，tmpfs 使用 fs_use_trans 规则确定 SC，如果没有指定“context=”选项则 seclabel 标志生效。

另外，在挂载 tmpfs 时/sbin/mount.tmpfs 脚本会在没有显示指定'(fs|def|root)?context='选项时默认使用“rootcontext”选项保证被挂载文件系统根目录的标签在 mount 前后不发生变化（参见/sbin/mount.tmpfs 中的 bug 信息）。

7.1.5 Initial SID

安全上下文在用户态通过字符串来描述，而在 SELinux 内核中使用 context 数据结构来表示它。给每个 context 数据结构都分配一个 u32 sid，该 sid 保存在相应内核数据结构的安全属性扩展中（以描述该内核数据结构的安全上下文）。SELinux 内核驱动使用 sidtab 哈希表来描述所有已经分配了 sid 的 context 数据结构，通过查询该哈希表即可得到一个安全上下文所对应的 sid。

几乎所有 sid 的分配和注册都是在运行时完成的，但是在 refpolicy 中定义了 27 个“Initial SID”，用于在 SELinux 内核驱动初始化时、由 init 进程通过 selinuxfs 接口加载 policy.X 之前，描述相应内核设施的初始安全属性。

在部署 SELinux 环境时（之前 init 已经完成 policy.X 的装载），通过 restorecon 命令给整个文件系统上所有正规文件和目录打标签。而系统上其它内核数据结构安全属性的初始状态，比如所有进程的起点 init 进程的安全属性，就需要由相应的 Initial SID “kernel”来决定，它是整个系统上 domain transition 的起点。到用户登录系统后，相关进程可以发生如下切换：

```
kernel_t -> init_t -> getty_t -> local_login_t -> sysadm_t
```

这里需要强调的是，在 init 进程装载 policy.X 之前，它处于 kernel_t 这个出发点上（由 selinux_init 函数决定，参见下文）；装载后 init 再次执行自己时，由于 policy.X 已经被装载（SELinux 内核 Security Server 完成初始化），相应 type_transition 规则生效，所以 init 进程切换到 init_t。这一点可以从/proc/1/attr/prev 验证：

```
[root/sysadm_r/s0@~]# cat /proc/1/attr/prev
system_u:system_r:kernel_t:s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# cat /proc/1/attr/current
system_u:system_r:init_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s kernel_t -c process"
Password:
Found 5 semantic te rules:
   type_transition kernel_t udev_exec_t : process udev_t;
   type_transition kernel_t anaconda_exec_t : process anaconda_t;
   type_transition kernel_t init_exec_t : process init_t;
   type_transition kernel_t insmod_exec_t : process insmod_t;
   type_transition kernel_t hotplug_exec_t : process hotplug_t;
[root/sysadm_r/s0@~]#
```

7.1.5.1 Initial SID 和 Initial SC 的定义

Initial SID 首先在 flask/initial_sids 文件中声明，比如：

```
cao@cao-laptop:/work/selinux/refpolicy/policy$ grep -v "#" flask/initial_sids | awk "NF>0" | wc -l
27
cao@cao-laptop:/work/selinux/refpolicy/policy$ grep -v "#" flask/initial_sids | awk "NF>0"
sid kernel
sid security
```

```
sid unlabeled
sid fs
sid file
sid file_labels
sid init
sid any_socket
sid port
sid netif
sid netmsg
sid node
sid igmp_packet
sid icmp_socket
sid tcp_socket
sid sysctl_modprobe
sid sysctl
sid sysctl_fs
sid sysctl_kernel
sid sysctl_net
sid sysctl_net_unix
sid sysctl_vm
sid sysctl_dev
sid kmod
sid policy
sid scmp_packet
sid devnull
cao@cao-laptop:/work/selinux/refpolicy/policy$
```

在 SELinux toolchain 的 define_initial_sid 函数中处理上述规则，其主要步骤如下：
1，分配一个 ocontext_t 数据结构（参见下文），由 newc 指针指向；
2，读取名称字符串，比如“kernel”，保存到 newc->u.name 中；
3，基于当前 policydb->ocontexts[OCON_ISID]队列队首元素的 sid[0]的数值，加 1，作为当前 sid 的新的索引；
4，最后，将 newc 所指向 ocontext_t 数据结构加入 policydb->ocontexts[OCON_ISID]队列队首。

由此可见，Initial SID 的索引从 1 开始。

然后在具体的 pp 中定义 Initial SID 所对应的安全上下文（我把它们称为“Initial SC”）。比如在 kernel.te 中定义了如下 19 个 Initial SC：

```
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$ grep sid kernel/kernel.te | grep -v "#" | wc -l
19
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$ grep sid kernel/kernel.te | grep -v "#" | head
sid kernel              gen_context(system_u:system_r:kernel_t,mls_systemhigh)
sid sysctl              gen_context(system_u:object_r:sysctl_t,s0)
sid unlabeled           gen_context(system_u:object_r:unlabeled_t,mls_systemhigh)
sid any_socket          gen_context(system_u:object_r:unlabeled_t,mls_systemhigh)
sid file_labels         gen_context(system_u:object_r:unlabeled_t,s0)
sid icmp_socket         gen_context(system_u:object_r:unlabeled_t,mls_systemhigh)
sid igmp_packet         gen_context(system_u:object_r:unlabeled_t,mls_systemhigh)
sid init                gen_context(system_u:object_r:unlabeled_t,s0)
sid kmod                gen_context(system_u:object_r:unlabeled_t,mls_systemhigh)
sid policy              gen_context(system_u:object_r:unlabeled_t,mls_systemhigh)
cao@cao-laptop:/work/selinux/refpolicy/policy/modules$
```

比如，Initial SID “kernel”对应的 SC 为“system_u:system_r:kernel_t:mls_systemhigh”，和从/proc/1/attr/prev 中读取的结果一致。

在 SELinux toolchain 的 define_initial_sid_context 函数中处理上述规则，它首先检查相应 sid 是否已经定义在 policydb->ocontexts[OCON_ISID]队列中。然后调用 parse_security_context 函数执行以下操作：
1，继续从 token queue 中弹出 SC 中 user 部分的字符串，在当前 policydb->p_users.table 哈希表中查找相应的 userdatum_t 数据结构；
2，将 userdatum_t 数据结构的 s.value，保存到相应 context->user 域中。即为相应 user 标识符的 policy value；
3，采用类似的方法确定 SC 中的其他各个域的 policy value，并保存到 context 数据结构的相应域中。

7.1.5.2 Initial SID 和 Initial SC 的写出和解析

和 Initial SID 相关的 ocontext 数据结构的定义如下：

```
typedef struct ocontext {
        union {
                char *name;     /* name of initial SID, fs, netif, fstype, path */
                ......
        } u;
        ......
        context_struct_t context[2];    /* security context(s) */
        sepol_security_id_t sid[2];     /* SID(s) */
        struct ocontext *next;
} ocontext_t;
```

如上文所述，Initial SID 的编号保存在 sid[0]域中，名称字符串由 u.name 指向，相应 Initial SC 由 context[0]来描述。所有 Initial SID 相关的 ocontext_t 数据结构组织在 policydb->ocontexts[OCON_ISID]队列中。

在创建 policy.X 时，下述函数调用链将 sid[0]和 context[0]域写出 policy.X：

```
ocontext_write > ocontext_write_selinux > context_write > mls_write_range_helper
```

（TODO：好像没有看到写出 u.name 的代码？但是从 seinfo 的结果来看，policy.X 中应该包含所有的相关信息）

可以使用“checkpolicy -dbM”来直接解析 policy.X，得到 sid[0]及相对应的 context[0]，比如：

```
Choose:  6
sid 1 -> scontext system_u:system_r:kernel_t:s15:c0.c1023
sid 2 -> scontext system_u:object_r:security_t:s15:c0.c1023
sid 3 -> scontext system_u:object_r:unlabeled_t:s15:c0.c1023
sid 4 -> scontext system_u:object_r:fs_t:s0
sid 5 -> scontext system_u:object_r:file_t:s0
sid 6 -> scontext system_u:object_r:unlabeled_t:s0
sid 7 -> scontext system_u:object_r:unlabeled_t:s0
sid 8 -> scontext system_u:object_r:unlabeled_t:s15:c0.c1023
sid 9 -> scontext system_u:object_r:port_t:s0
sid 10 -> scontext system_u:object_r:netif_t:s0-s15:c0.c1023
......
```

注意 sid[0]数值从 1 开始分配，各个 Initial SID 的定义顺序由其在 flask/initial_sids 文件中的出现顺序决定。

在装载、解析 policy.X 时，下述函数调用链根据 policy.X 中 ocontext 数据结构的二进制表示，创建相应的内核描述符：

```
security_load_policy > policydb_read > ocontext_read > context_read_and_validate
                        > policydb_load_isid > sidtab_insert
```

其中在 ocontext_read 函数中分配 ocontext 数据结构并加入内核 policydb->ocontexts[OCON_ISID]队列的首部。在 context_read_and_validate 中读取、验证 SC 的各个域并保存到 context[0]数据结构中。

在 policydb_load_isid 函数中初始化 sidtab 哈希表，然后将 policydb->ocontexts[OCON_ISID]队列中所有元素注册到 sidtab 中，完成 sidtab 的初始化。

7.1.5.3 Initial SID 和 Initial SC 的使用

1，如上文所述，init 进程在装载 policy.X 时将所有 Initial SID 及其 Initial SC 注册到 sidtab 中。在此之前，在 SELinux 内核驱动初始化时，selinux_init > cred_init_security 函数给 init 进程分配安全扩展属性，并将其中 tsec->osid 和 tsec->sid 都设置为 1，即为 SECINITSID_KERNEL，即为“kernel”Initial SID 的编号：

```
/*
 * initialise the security for the init task
 */
static void cred_init_security(void)
{
        struct cred *cred = (struct cred *) current->real_cred;
        struct task_security_struct *tsec;

        tsec = kzalloc(sizeof(struct task_security_struct), GFP_KERNEL);
        if (!tsec)
                panic("SELinux:  Failed to initialize initial task.\n");

        tsec->osid = tsec->sid = SECINITSID_KERNEL;
        cred->security = tsec;
}
```

所以在 init 进程再装载完 policy.X 并重新执行自己而发生 domain transition 时，就可以查询 sidtab 返回其 tsec->sid == 1 所对应的 Initial SC 了，即为“system_u:system_r:kernel_t:s15:c0.c1023”。

2，在 policy.X 完成解析、Initial SID 被注册到 sidtab 之后，全局变量 ss_initialized 才会被置 1。在此之前如果需要完成（sid，context）之间的转换，就必须经由 initial_sid_to_string[]数组（数组元素为 Initial SID 的名称字符串，索引即为相应 sid 数值）。估计其所在文件 initial_sid_to_string.h 由用户态 refpolicy 源码树下的同名文件拷贝而来，后者经由 flask/flask.py 生成。

3，Initial_SID 的另外一个作用就是作为相关内核数据结构安全属性的默认初始 sid，比如：

```
static int superblock_alloc_security(struct super_block *sb)
{
        struct superblock_security_struct *sbsec;
```

```
        sbsec = kzalloc(sizeof(struct superblock_security_struct), GFP_KERNEL);
        if (!sbsec)
                return -ENOMEM;

        mutex_init(&sbsec->lock);
        INIT_LIST_HEAD(&sbsec->isec_head);
        spin_lock_init(&sbsec->isec_lock);
        sbsec->sb = sb;
        sbsec->sid = SECINITSID_UNLABELED;
        sbsec->def_sid = SECINITSID_FILE;
        sbsec->mntpoint_sid = SECINITSID_UNLABELED;
        sb->s_security = sbsec;

        return 0;
}
```

7.1.6 进程创建的内核数据结构的标签

进程创建的内核数据结构，比如打开文件描述符 fd（对应 file 数据结构），或者 socket（注意不是其所 bind 到的 sock_file），它们的 type 和 SL 都继承于创建者。比如 syslogd_t 创建的 AF_UNIX socket(属于 unix_dgram_socket 类型）的 type 就是 syslogd_s_t（若无相应 type_transition 规则则为 syslogd_t），SL 继承了 mls_systemhigh；而它所 bind 到的 sock_file，即/dev/log，其标签为 devlog_t，SL 也继承了 mls_systemhigh：

```
[root/sysadm_r/s0@~]# ps -eZ | grep syslog
system_u:system_r:syslogd_t:s15:c0.c1023 1162 ? 00:02:10 syslogd
[root/sysadm_r/s0@~]
[root/sysadm_r/s0@~]# compute_create system_u:system_r:syslogd_t:s15:c0.c1023
system_u:system_r:syslogd_t:s15:c0.c1023 unix_dgram_socket
system_u:system_r:syslogd_s_t:s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s syslogd_t -t syslogd_t -c unix_dgram_socket"
Password:
Found 1 semantic te rules:
   type_transition syslogd_t syslogd_t : unix_dgram_socket syslogd_s_t;
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# netstat -a | grep log
unix  13     [ ]         DGRAM                    2351   /dev/log
[root/sysadm_r/s0@~]# ls -Z /dev/log
srw-rw-rw-  root root system_u:object_r:devlog_t:s15:c0.c1023 /dev/log
[root/sysadm_r/s0@~]#
```

注意：被打开的文件自身（由 inode 数据结构表示）和相应的打开文件描述符（由 file 数据结构表示），是不同的 object；AF_UNIX socket 及其 bind 到的 sock_file，也是不同的 object，在 SELinux 中给它们赋予不同的 type。理论上只要对象由不同的内核数据结构来描述，就可能具备独立的标签。

7.2 何时需要为应用程序开发 pp?

有言在先：并不是对所有新安装的应用程序都必须开发相应的 pp。

应用程序通常被安装在 PATH 环境变量所指路径下，如果在 file_contexts 文件中没有为其明确指定标签，则它们都继承所在目录的标签（通常为 bin_t）。以/usr/bin/为例，只有个别可执行文件的标签不是

bin_t:

```
[root/sysadm_r/s0@bin]# pwd
/usr/bin
[root/sysadm_r/s0@bin]# ls -Z | awk -F: '{print $3}' | wc -l
681
[root/sysadm_r/s0@bin]# ls -Z | awk -F: '{print $3}' | grep -v bin_t | wc -l
19
[root/sysadm_r/s0@bin]# ls -Z | awk -F: '{print $3}' | grep -v bin_t | sort -u
checkpolicy_exec_t
chfn_exec_t
crontab_exec_t
groupadd_exec_t
locate_exec_t
mysqld_exec_t
mysqld_safe_exec_t
newrole_exec_t
passwd_exec_t
rpm_exec_t
rsync_exec_t
ssh_agent_exec_t
ssh_exec_t
ssh_keygen_exec_t
sudo_exec_t
[root/sysadm_r/s0@bin]#
```

当前 policy.X 中所有用户类型以及许多应用程序 domain 都具有对 bin_t 的{ execute execute_no_trans }权限（确切地说，所有调用了 corecmd_exec_bin 接口的 domain），比如：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -t bin_t -c file -p execute_no_trans -s user_t"
Password:
Found 1 semantic av rules:
   allow user_t bin_t : file { ioctl read getattr lock execute execute_no_trans entrypoint open } ;
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -t bin_t -c file -p execute_no_trans -s crontab_t"
Password:
Found 1 semantic av rules:
   allow crontab_t bin_t : file { ioctl read getattr lock execute execute_no_trans open } ;
[root/sysadm_r/s0@~]#
```

execute 权限使得相应 domain 具有对 bin_t 的可执行能力，而 execute_no_trans 权限使得相应 domain 在执行 bin_t 期间，不发生 domain transition 而保持在原有的 domain 中。所以，是否能够成功执行 bin_t 类型应用程序将由调用者所在的 domain 决定：**只要合法的调用者 domain 已经具有了足够的能力来支持可执行程序的行为，就不再需要为它定义新的 domain 并要求在执行期间发生切换。**

反之，必须开发相应的 pp 以实现新的 domain 以支持可执行程序的行为。由于原有调用者不具备新 domain 的权限，所以要求在执行可执行程序期间发生 domain transition（从而在不给调用者 domain 增加权限的前提下，保证应用程序能够顺利执行）。

以 mtree 为例，/usr/sbin/mtree 的默认标签为 bin_t，而系统管理员 sysadm_r 已经能够顺利使用 mtree：

```
[root/sysadm_r/s0@~]# ls -Z /usr/sbin/mtree
-rwxr-xr-x  root root system_u:object_r:bin_t:s0       /usr/sbin/mtree
[root/sysadm_r/s0@~]# /usr/sbin/mtree -c -K cksum,md5digest,shaldigest,rmdl60digest -s 333332213445 -p /bin
> /dev/null
```

```
mtree: /bin checksum: 1901152446
[root/sysadm_r/s0@~]# echo $?
0
[root/sysadm_r/s0@~]#
```

这是因为 sysadm_r 在执行 mtree 期间仍旧保持在 sysadm_t，而目前 policy.X 所赋予 sysadm_t 的权限足够支持 mtree 程序的行为，因此就不需要为其设计相应的 pp 了。

与之相比 vlock 就不行了。如果修改/usr/sbin/vlock-main 的标签为 bin_t，则 user_r 在命令行调用 vlock 则出错：

```
root@qemu-host:/root> id -Z
root:secadm_r:secadm_t:s0-sl5:c0.cl023
root@qemu-host:/root> chcon -t bin_t /usr/sbin/vlock-main
root@qemu-host:/root> ls -Z /usr/sbin/vlock-main
-rws--x--x  root root system_u:object_r:bin_t:s0       /usr/sbin/vlock-main
root@qemu-host:/root>

-bash-3.2$ vlock
This TTY is now locked.

Please press [ENTER] to unlock.
cao's Password:
vlock: System error
```

在 sysadm_r 登录的控制台上可以看到相应的错误信息：

```
root@qemu-host:/root> id -Z
root:sysadm_r:sysadm_t:s0-sl5:c0.cl023
root@qemu-host:/root> ps -eZ | grep vlock
user_u:user_r:user_t:s0          1004 pts/1    00:00:00 vlock-main
root@qemu-host:/root>
root@qemu-host:/root> dmesg | grep vlock
type=1400 audit(1291280687.705:92): avc:  denied  { setuid } for  pid=1011 comm="vlock-main" capability=7
scontext=user_u:user_r:user_t:s0 tcontext=user_u:user_r:user_t:s0 tclass=capability
type=1400 audit(1291280688.861:93): avc:  denied  { setuid } for  pid=1012 comm="vlock-main" capability=7
scontext=user_u:user_r:user_t:s0 tcontext=user_u:user_r:user_t:s0 tclass=capability
type=1400 audit(1291280688.909:94): avc:  denied  { create } for  pid=1004 comm="vlock-main"
scontext=user_u:user_r:user_t:s0 tcontext=user_u:user_r:user_t:s0 tclass=netlink_audit_socket
type=1400 audit(1291280688.917:95): avc:  denied  { write } for  pid=1004 comm="vlock-main" name="log"
dev=tmpfs ino=9285 scontext=user_u:user_r:user_t:s0 tcontext=system_u:object_r:devlog_t:sl5:c0.cl023
tclass=sock_file
root@qemu-host:/root>
```

由此可见，在执行 vlock-main 程序期间用户进程的 domain 为 user_t，它不具备如下能力：
1，对自己的 setuid 能力；
2，对 netlink_audit_socket 类型对象的 create 能力；
3，写入套接字文件/dev/log（devlog_t）的能力。

以第 2 点为例，我们需要创建 vlock.pp，把/usr/sbin/vlock-main 的标签定义为 vlock_exec_t，使得在调用 vlock-main 时切换到 vlock_t，那么我们就可以通过定义如下规则：

```
allow vlock_t self:netlink_audit_socket { create_netlink_socket_perms nlmsg_relay };
```

使得 vlock_t 具有对 netlink_audit_socket 类型对象的 create 能力，从而避免上述错误消息。

注意，根据“Least Privilege”原则，显然不能把上述能力添加到 user_t 中。

7.3 设计 pp 的一般过程

1，从程序的配置文件、源代码中了解程序的行为和所需要使用的系统资源，比如：
- 网络 port，外设；
- /etc/下的配置文件；
- /var/lib/下的中间文件，库文件；
- /var/run/下的 PID 文件，socket 文件；
- /var/log/xxx/下的日志文件；

可以使用 rpm -qpl 命令得到相应 rpm 包**静态安装**的文件列表。

另外，可以借助 GRSecurity 的 Learning Mode，在没有使能 SELinux 时得到程序**运行时**所访问的系统资源列表。或者使用 strace 了解程序在**运行时**的资源使用情况，比如使用“strace -ff -e trace=file -o output_file”命令得到应用程序所有和文件相关的操作（参见下文），或者使用“strace -p <pid>”功能监控一个后台进程的行为（参见 8.8.1 小节）。

2，明确安全目标，比如：
- 在保证程序完成既定功能的情况下，尽量减少赋予它的权限，从而最大限度地保护整个系统；
- 只有 sysadm_r 角色能够创建、删除、修改该应用在用户态的所有文件（比如配置文件，log/pid 文件等）；
- 只有 secadm_r 角色才能运行该可执行程序（比如定义 xxx_run 接口，并只对 secadm_t 调用该接口）；
- 应用程序运行在 s0 或者 mls_systemhigh 的 Security Level 中；

3，创建.te 文件，赋予新 domain 足够的能力，比如：
- 给相关文件定义 type，attribute，entrypoint，并使得新 domain 对这些文件具有足够的能力；
- 允许使用系统资源、控制台、外设等等；
- 允许调用其他应用程序；
- 定义**运行时**创建的文件的默认标签（使用 type_transition 规则）；

可以在 file_contexts 中查找相关系统资源文件对应的标签，然后找到定义它们的 pp，从而确定支持的接口。最后在.te 中调用合适的接口从而对相应的标签具有所需的能力。

注意，如果所需访问的外设和板子的 BSP 实现相关（比如某些设备上只有触摸屏，而其他的只有键盘和鼠标），则可以使用若干 tunable 来控制对相关接口的调用，而最终用户可以针对具体使用的硬件环境来设置对应的 tunable。

4，创建.if 文件，向其它 pp 开放相应的权限，比如：
- 定义 xxx_domtrans 和 xxx_run 接口，使得某些用户 domain 能够调用 xxx_run 接口（比如对 secadm_r 调用）；
- 定义 xxx_manage_xxx_files 接口，赋予管理相应用户态文件的能力（比如对 sysadm_r 调用）；

注意，即使在当前的实现中并不一定会调用所有的 xxx_manage_xxx_files 接口，但是也应该在.if 中实现它们，这样将来用户就可以在定制其 policy 时方便地使用了。

5，创建.fc 文件，给所有**静态安装**的文件、中间文件、临时文件设置 SC；

6，编译，安装，在 Permissive 模式下执行程序，观察相应的 AVC Denied Message；

7，调整.te 规则，比如：
- 补充必须的 TE 规则
- dontaudit 那些已知的、非必须的行为；

8，对于发行版的维护者而言，可以将第一次实现的 domain 用“permissive”关键字来定义，从而使得 SELinux 内核 Security Server 以 permissive 模式对待该 domain（不以规则库的查询结果阻碍正常执行），但是整个系统仍然处在 Enforcing 模式下。这样维护者可以根据用户反馈的错误消息修正相应的规则，直到下一个版本时再正式以“type”关键字定义。

详见《SELinux By Example》第 14 章关于为 IRC 创建 pp 的过程。

7.4 为 vlock 程序编写 vlock.pp

7.4.1 第一阶段：定义基本的.te，.fc 和.if 文件

1，声明基本类型，定义基本类型的属性及相互关系

```
type vlock_t;
type vlock_exec_t;
application_domain(vlock_t,vlock_exec_t)
```

注意 application_domain 接口足以将新定义的 domain/type 加入所必须的各种属性，比如 vlock_t 属于 domain 属性，vlock_exec_t 属于 exec_type 属性，且二者具有 entrypoint 关系。

2，设计相关文件的标签，可以使用 rpm -qpl 命令得到安装文件列表。

```
/etc/pam.d/vlock                        etc_t
/usr/bin/vlock                          vlock_exec_t
/usr/share/doc/vlock-1.3                usr_t
/usr/share/doc/vlock-1.3/COPYING        usr_t
/usr/share/doc/vlock-1.3/README         usr_t
/usr/share/man/man1/vlock.1.gz          man_t
```

除了/usr/bin/vlock 之外，其他文件都采用默认的标签即可，所以只需在 vlock.fc 中定义如下内容：

```
/usr/bin/vlock  --      gen_context(system_u:object_r:vlock_exec_t,s0)
```

3，在.if 文件中定义相应接口，使得各种用户类型都可以调用 vlock_exec_t 程序，并经由之 domain transition 切换到 vlock_t 中：

```
interface(`vlock_domtrans',`
        gen_require(`
                type vlock_t, vlock_exec_t;
        ')

        corecmd_search_bin($1)          # Enable caller's domain to search /usr/bin/, since we are there
        domtrans_pattern($1, vlock_exec_t, vlock_t)
')

interface(`vlock_run',`
        gen_require(`
                type vlock_t;
        ')
```

```
        vlock_domtrans($1)
        role $2 types vlock_t;          # Enable caller's role to associate with our new type/domain
')
```

然后，可以就 sysadm.te，staff.te 和 unprivuser.te 调用 vlock_run(domain，role)接口，使得相应的用户角色 domain 能够切换到 vlock_t 并组成合法的 SC。注意，社区不鼓励在 userdom_common_user_template 模板中直接调用该接口（用于创建 admin 和各种类型的 unpriv 用户，从而使得所有用户都能够进入 vlock_t），而是在针对 sysadm.te/staff.te/unprivuser.te 分别调用，从而精细地控制哪种用户能够使用 vlock。

4，定义 vlock_t 应该具有的能力

4.1 首先使用 GRsecurity 得到 vlock 运行时的资源使用列表:

```
        /                               h
        /dev                            h
        /dev/console
        /dev/tty                        rw

        /etc                            r
        /etc/security                   h
        /etc/security/pam_env.conf      r
        /etc/pam.d
        /etc/pam.d/other                r
        /etc/pam.d/system-auth          r       etc_t
        /etc/pam.d/vlock                r
        /etc/grsec                      h
        /etc/ssh                        h

        /etc/shadow-                    h       shadow_t
        /etc/gshadow                    h       shadow_t
        /etc/gshadow-                   h       shadow_t

        /usr                            h
        /usr/bin/vlock                  x

        /var                            h
        /var/run

        /lib64                          rx

        /proc
        /proc/filesystems               r       # kernel_read_system_state
        /proc/kcore                     h
        /proc/sys                       h
        /proc/bus                       h

        /selinux                                # selinux_dontaudit_getattr_fs

        -CAP_ALL
        +CAP_AUDIT_WRITE                        # allow vlock_t self: capability audit_write
        +CAP_AUDIT_CONTROL                      # redundant
```

4.2 然后使用 strace 得到 vlock 所打开的文件列表，进一步验证 vlock_t 需要访问的系统资源。注意需要使能 allow_ptrace 布尔变量，过滤和 ENOENT，SIGCHLD 相关条目后结果如下:

```
root@qemu-host:/root> getenforce
Enforcing
root@qemu-host:/root> getseboo1 a11ow_ptrace
a11ow_ptrace --> off
root@qemu-host:/root> strace -o l v1ock
/usr/bin/v1ock: 1ine 224: /usr/sbin/v1ock-main: Operation not permitted
root@qemu-host:/root> setseboo1 a11ow_ptrace 1
type=1405 audit(1288448597.947:42): boo1=a11ow_ptrace va1=1 o1d_va1=0 auid=4294967295 ses=4294967295
root@qemu-host:/root> getseboo1 a11ow_ptrace
a11ow_ptrace --> on
root@qemu-host:/root>
root@qemu-host:/root> strace -e trace=open -o l v1ock
This TTY is now 1ocked.

P1ease press [ENTER] to un1ock.
root's Password:

root@qemu-host:/root> grep -v ENOENT 1 | grep -v SIGCHLD
open("/usr/1ib/1ibread1ine.so.5", O_RDONLY) = 3
open("/usr/1ib/1ibhistory.so.5", O_RDONLY) = 3
open("/1ib/1ibncurses.so.5", O_RDONLY)  = 3
open("/1ib/1ibd1.so.2", O_RDONLY)       = 3
open("/1ib/1ibc.so.6", O_RDONLY)        = 3
open("/1ib/1ibtinfo.so.5", O_RDONLY)    = 3
open("/dev/tty", O_RDWR|O_NONBLOCK|O_LARGEFILE) = 3
open("/proc/meminfo", O_RDONLY)         = 3
open("/usr/bin/v1ock", O_RDONLY|O_LARGEFILE) = 3
open("/1ib/1ibd1.so.2", O_RDONLY)       = 3
open("/1ib/1ibpam.so.0", O_RDONLY)      = 3
open("/1ib/1ibc.so.6", O_RDONLY)        = 3
open("/1ib/1ibaudit.so.0", O_RDONLY)    = 3
open("/etc/pam.d/v1ock", O_RDONLY|O_LARGEFILE) = 3
open("/etc/pam.d/system-auth", O_RDONLY|O_LARGEFILE) = 4
open("/1ib/security/pam_env.so", O_RDONLY) = 5
open("/1ib/security/pam_unix.so", O_RDONLY) = 5
open("/1ib/1ibns1.so.1", O_RDONLY)      = 5
open("/1ib/1ibcrypt.so.1", O_RDONLY)    = 5
open("/1ib/1ibse1inux.so.1", O_RDONLY)  = 5
open("/1ib/security/pam_deny.so", O_RDONLY) = 5
open("/1ib/security/pam_permit.so", O_RDONLY) = 4
open("/1ib/security/pam_1imits.so", O_RDONLY) = 4
open("/etc/pam.d/other", O_RDONLY|O_LARGEFILE) = 3
open("/1ib/security/pam_warn.so", O_RDONLY) = 4
open("/etc/nsswitch.conf", O_RDONLY)    = 3
open("/1ib/1ibnss_compat.so.2", O_RDONLY) = 3
open("/1ib/1ibnss_nis.so.2", O_RDONLY)  = 3
open("/1ib/1ibnss_f1es.so.2", O_RDONLY) = 3
open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
open("/proc/se1f/task/1092/attr/current", O_RDONLY|O_LARGEFILE) = 3
open("/etc/security/pam_env.conf", O_RDONLY|O_LARGEFILE) = 3
open("/etc/passwd", O_RDONLY|O_CLOEXEC) = 3
root@qemu-host:/root>
```

注意，这里只使用 strace 跟踪了 open 系统调用，而实际开发中还需要指定其他的 trace 项，比如使用“-e trace=network”跟踪对网络资源的使用。通常情况下可使用“-e trace=f1le -e trace=network -e trace=process”选项。

4.3 理论上我们需要赋予 vlock_t 相应的权限从而能够正确访问这些文件。实际中由于上述列表可能包含某些 vlock 并不真正需要访问的文件，或者和某些冗余操作相关（无法访问相应的文件并不影响 vlock 的正确执行），则应该使用 dontaudit 规则屏蔽相应的错误信息（参见下文）。

还需要注意的是，GRSecurity 得到的有关进程所需权能的描述，可能和 SELinux 上实际需要的并不一致。比如上面说 vlock 需要 CAP_AUDIT_WRITE 和 CAP_AUDIT_CONTROL 能力，而实际上 vlock_t 只需要 audit_write 能力即可：

```
cao@cao-laptop:~$ sesearch -SCA -s vlock_t -c capability /etc/selinux/refpolicy-mls/policy/policy.24
Found 1 semantic av rules:
   allow vlock_t vlock_t : capability audit_write ;
cao@cao-laptop:~$
```

audit_write capability 许可用户态进程发送 audit 消息，而 audit_control 许可进程修改 audit 规则，显然对 vlock_t 不应该许可该能力（只能对 sysadm_t/auditadm_t 许可该能力）。

另外，由于 vlock_t 属于 domain 属性，利用 apol 工具可以看到该属性已经对某些类型的目录和文件具有 getattr/search/open 甚至 ioctl/read/lock 权限（使用 apol 时可指定精确匹配“Only direct matches”，则可以防止对 domain 属性展开）：

```
9 rules match the search criteria.
Number of enabled conditional rules: 0
Number of disabled conditional rules: 0

allow domain setrans_var_run_t : dir { getattr search open } ;
allow domain lib_t : dir { ioctl read getattr lock search open } ;
allow domain root_t : dir { ioctl read getattr lock search open } ;
allow domain usr_t : dir { getattr search open } ;
allow domain var_t : dir { getattr search open } ;
allow domain device_t : dir { ioctl read getattr lock search open } ;
allow domain etc_t : dir { ioctl read getattr lock search open } ;
allow domain proc_t : dir { getattr search open } ;
allow domain var_run_t : dir { ioctl read getattr lock search open } ;

---

Number of enabled conditional rules: 0
Number of disabled conditional rules: 0

allow domain ld_so_cache_t : file { ioctl read getattr lock open } ;
allow domain lib_t : file { ioctl read getattr execute open } ;
allow domain ld_so_t : file { ioctl read getattr execute open } ;
allow domain textrel_shlib_t : file { ioctl read getattr execute execmod open } ;
```

比如 lib_t 为/lib/*或/lib64/*的标签，ld_so_t 为加载器/lib/ld-2.11.1.so 的标签，应用程序应该能够正常使用加载器和动态链接库；/var/，/var/run/的标签分别为 var_t，var_run_t，应用程序通常都在/var/run/下创建自己的 pid 文件（使用自定义标签 xxx_var_run_t）。所以，如果 domain 属性对 GRSecurity 列表中的文件和目录已经有足够的访问权限，就不需要再调用相应的接口了。

参照 domain 属性已经具有的能力，需要给 vlock_t 增加的权限如下（部分）：

```
files_read_etc_files(vlock_t)
auth_domtrans_chk_passwd(vlock_t)
corecmd_list_bin(vlock_t)
corecmd_read_bin_symlinks(vlock_t)
```

```
kernel_read_system_state(vlock_t)
```

4.4 由于用户可能从 console，tty 或 pty 登录，因此需要对 vlock_t 调用如下接口确保对各种设备都有读写能力：

```
userdom_use_user_terminals(vlock_t)
```

虽然 console/tty/pty 设备的默认标签为 console_device_t/tty_device_t/devpts_t，但是 login/ssh 程序在用户登录时会根据相应的 type_change 规则将 console/tty/pty 设备的标签 relabel 为 user_tty_device_t/user_tty_device_t/user_devpts_t。

注意，这里只需要就 vlock_t 调用 userdom_use_user_terminal 接口，而对于 login/ssh/mingetty 等程序的 domain 才需要调用 term_use_all_terms 接口。由于 user_devpts_t 和 user_tty_device_t 也属于 ptynode 和 ttynode 属性，因此该接口是 userdom_use_user_terminals 接口的超集。

4.5 由于用户可能采用任意安全级别，而/var/log/tallylog 的安全级别为 s0，因此如果用户采用了高于 s0 的任何安全级别，由于 MLS 的“no write down”原则 vlock_t 将无法写入 tallylog 文件。相应 MLS constraint 如下：

```
# the "single level" file "write" ops
mlsconstrain { file lnk_file fifo_file dir chr_file blk_file sock_file } { write create setattr relabelfrom
append unlink link rename mounton }
        (( l1 eq l2 ) or
         (( t1 == mlsfilewritetocir ) and ( h1 dom l2 ) and ( l1 domby l2 )) or
         (( t2 == mlsfilewriteinrange ) and ( l1 dom l2 ) and ( h1 domby h2 )) or
         ( t1 == mlsfilewrite ) or
         ( t2 == mlstrustedobject ));
```

由此可见必须满足如下约束之一，关于文件的写操作才能顺利进行：
1）subject 和 object 的当前安全级别相同；
2）subject 属于 mlsfilewritetocir 属性，则即使 object 的当前安全级别高于 subject，但是只要不高于 subject 的 clearance level，则 subject 仍能正常写入 object。相应安全级别关系示图如下：
l1 ------- h1
      l2
3）object 属于 mlsfilewriteinrange 属性，则即使 subject 的当前级别高于 object，但只要 subject 的安全级别范围为 object 的子集，则许可“write down”行为。相应安全级别关系示图如下：
     l1 ------- h1
l2 ------------------ h2
4）subject 属于 mlsfilewrite 属性，则可以写入任何安全级别的 object；
5）object 属于 mlstrustedobject 属性，则可以被处于任何安全级别的 subject 写入；

就 vlock 而言，用户的安全级别可能为 mls_systemhigh（s15:c0.c1023），而 tallylog 的安全级别只是 s0，因此上述前 3 种条件都不满足。目前对 vlock_t 调用了 mls_file_write_all_levels 接口，或许更加合适的做法是对 faillog_t 调用 mls_trusted_object 接口（因为可能有若干 domain 都需要写入 tallylog 文件）。

7.4.2 第二阶段：根据 AVC Denied Msg 补充相应的规则

安装 vlock.pp，并执行 restorecon /usr/bin/vlock，在 permissive 模式下运行 vlock，得到的 AVC Denied Msg 及需要补充调用的接口如下：

```
1. allow vlock_t self:netlink_audit_socket { create_netlink_socket_perms };
```

type=AVC msg=audit(1268114828.208:431): avc: denied { create } for pid=2444 comm="vlock" scontext=staff_u:secadm_r:secadm_t:s0-s15:c0.c255 tcontext=staff_u:secadm_r:secadm_t:s0-s15:c0.c255 tclass=netlink_audit_socket

2. domain_use_interactive_fds(vlock_t)　　详见后文

type=USER_AUTH msg=audit(1268187673.008:196): user pid=2371 uid=0 auid=501 ses=2 subj=staff_u:staff_r:vlock_t:s0-s15:c0.c255 msg='op=PAM:authentication acct="root" exe="/usr/bin/vlock" (hostname=?, addr=?, terminal=? res=failed)'

3. allow vlock_t self:fifo_file rw_fifo_file_perms;

type=AVC msg=audit(1268187673.004:190): avc: denied { write } for pid=2371 comm="vlock" path="pipe:[11365]" dev=pipefs ino=11365 scontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tcontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tclass=fifo_file

4. miscfiles_read_localization(vlock_t)

type=AVC msg=audit(1268187673.004:193): avc: denied { read } for pid=2371 comm="vlock" name="localtime" dev=sda1 ino=229589 scontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tcontext=system_u:object_r:locale_t:s0 tclass=file

vlock 需要访问/etc/localtime 配置文件。

5. allow vlock_t self:unix_dgram_socket { create connect };

time->Wed Mar 10 02:21:13 2010
type=SYSCALL msg=audit(1268187673.008:195): arch=c000003e syscall=41 success=no exit=-13 a0=1 a1=2 a2=0 a3=0 items=0 ppid=2341 pid=2371 auid=501 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0 ses=2 comm="vlock" exe="/usr/bin/vlock" subj=staff_u:staff_r:vlock_t:s0-s15:c0.c255 key=(null)
type=AVC msg=audit(1268187673.008:195): avc: denied { create } for pid=2371 comm="vlock" scontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tcontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tclass=unix_dgram_socket

type=AVC msg=audit(1268187782.492:243): avc: denied { connect } for pid=2408 comm="vlock" scontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tcontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tclass=unix_dgram_socket

unix_dgram_socket 描述本地机器上 AF_UNIX socket，具有 connect 权限才能使用 connect 函数发起到另外一个 socket 的连接。可以使用“strace -e trace=network”来观察应用程序发出的和网络相关的系统调用。

6. logging_send_syslog_msg(vlock_t)

type=AVC msg=audit(1268187782.492:243): avc: denied { write } for pid=2408 comm="vlock" name="log" dev=tmpfs ino=10513 scontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255 tcontext=system_u:object_r:devlog_t:s15:c0.c255 tclass=sock_file

/dev/log 为 syslogd 后台进程创建的 sock_file（其标签为 devlog_t），其上绑定了 type=syslogd_t 的 socket。许多用户态程序都向/dev/log 文件写入发给 syslogd 的消息。注意/dev/log 文件由 syslogd_t:mls_systemhigh 创建，因此它的安全级别继承了 mls_systemhigh，而其标签由相应的 type_transition 规则决定。

7.4.3 第三阶段：使用 dontaudit 规则屏蔽与冗余操作相关的错误信息

1，在 vlock 能够正常运行的情况下，仍发现有有关于 vlock_t 缺少对 vlock_t:capability 的 setuid/setgid 能力的 AVC 消息。通过阅读其 readme 发现，当前鼓励在配置时使用--enable-pam 选项，从而使得 vlock 程序不必直接访问/etc/shadow 文件，即不要求 vlock 程序为 setuid 的，因此不必赋予 vlock_t setuid 能力。所以可以增加如下 dontaudit 规则屏蔽此类消息：

```
# dont audit the failed attempt of vlock_t to setuid/setgid, because
# 1. we used --enable-pam for vlock to use PAM to authenticate passwd
# 2. no guarantee that making vlock setuid is safe
dontaudit vlock_t self:capability { setuid setgid };
```

--enable-pam 选项意味着必须调用 auth_domtrans_chk_passwd 接口，因为/usr/sbin/vlock-main 使用 pam_unix 模块进行身份认证，而后者使用 unix_chkpwd，由它来访问/etc/shadow 文件（注意 unix_chkpwd 程序已经是 setuid 的了，并且 chkpwd_t 被相应赋予了 setuid 能力）。

另外，对于 login/ssh 程序，它们需要根据登录用户的身份调用 setuid 函数正确地设置子 shell 进程的 uid，因此必须赋予 local_login_t setuid 能力。

2，通常情况下不调用 libselinux 的应用程序不需要访问/selinux，调用 selinux_dontaudit_getattr_fs 接口屏蔽下面的错误信息：

```
time->Wed Mar 10 22:24:02 2010
type=PATH msg=audit(1268259842.388:12828): item=0 name="/selinux" inode=1 dev=00:0c mode=040755 ouid=0
ogid=0 rdev=00:00 obj=system_u:object_r:security_t:s0
type=CWD msg=audit(1268259842.388:12828):  cwd="/root"
type=SYSCALL msg=audit(1268259842.388:12828): arch=c000003e syscall=137 success=no exit=-13 a0=3160417ba7
a1=73f9fac65ef0 a2=1000 a3=606ca0 items=1 ppid=2628 pid=3525 auid=501 uid=0 gid=0 euid=0 suid=0 fsuid=0
egid=0 sgid=0 fsgid=0 tty=pts1 ses=3 comm="vlock" exe="/usr/bin/vlock" subj=staff_u:staff_r:vlock_t:s0-
s15:c0.c255 key=(null)
type=AVC msg=audit(1268259842.388:12828): avc:  denied  { getattr } for  pid=3525 comm="vlock" name="/"
dev=selinuxfs ino=1 scontext=staff_u:staff_r:vlock_t:s0-s15:c0.c255
tcontext=system_u:object_r:security_t:s0 tclass=filesystem
```

3，/var/run/utmp 用户记录当前系统上各种用户的登录信息，详见其手册页。通常情况下至少不需要授予对 utmp 的写权限，因此至少应该使用 init_dontaudit_write_utmp 接口。就 vlock 而言，它对 utmp 的读操作也是不需要的。因此使用 init_dontaudit_rw_utmp 接口屏蔽相关错误信息：

```
time->Mon Mar 15 04:37:07 2010
type=PATH msg=audit(1268627827.856:1028): item=0 name="/var/run/utmp" inode=261300 dev=08:01 mode=0100664
ouid=0 ogid=22 rdev=00:00 obj=system_u:object_r:initrc_var_run_t:s0
type=CWD msg=audit(1268627827.856:1028):  cwd="/root"
type=SYSCALL msg=audit(1268627827.856:1028): arch=c000003e syscall=2 success=no exit=-13 a0=315edlb99f
a1=80002 a2=2 a3=72885a657cb0 items=1 ppid=2447 pid=3034 auid=1000 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0
sgid=0 fsgid=0 tty=pts0 ses=2 comm="vlock" exe="/usr/bin/vlock" subj=staff_u:secadm_r:vlock_t:s0-
s15:c0.c255 key=(null)
type=AVC msg=audit(1268627827.856:1028): avc:  denied  { read write } for  pid=3034 comm="vlock"
name="utmp" dev=sda1 ino=261300 scontext=staff_u:secadm_r:vlock_t:s0-s15:c0.c255
tcontext=system_u:object_r:initrc_var_run_t:s0 tclass=file
```

4，由实验得知，vlock_t 也并不一定需要针对/home 和/home/$USER/的 search 能力，因此需要调用 files_dontaudit_search_home 和 userdom_dontaudit_search_user_home_dirs 接口屏蔽相关错误信息：

```
-bash-3.2$
This TTY is now locked.

Please press [ENTER] to unlock.
cao's Password:
type=1400 audit(1288071131.540:22): avc:  denied  { search } for pid=779 comm="vlock-main" name="home"
dev=sda ino=16385 scontext=user_u:user_r:vlock_t tcontext=system_u:object_r:home_root_t tclass=dir
-bash-3.2$
```

```
root@qemu-host:/root> vlock
This TTY is now locked.

Please press [ENTER] to unlock.
root's Password:
type=1400 audit(1288070940.844:18): avc:  denied  { search } for pid=689 comm="vlock-main" name="root"
dev=sda ino=81921 scontext=root:sysadm_r:vlock_t tcontext=root:object_r:user_home_dir_t tclass=dir
root@qemu-host:/root>
```

7.4.4 其他注意事项

1，在根据 AVC Denied Msg 补充规则时，注意分析，而不能一味地按照 AVC Denied Msg 的字面意思进行翻译！比如，如果/usr/bin/vlock 没有被正确地设置为 vlock_exec_t，而仍为默认的 bin_t，则可以得到如下信息：

```
time->Thu Mar 11 03:54:32 2010
type=SYSCALL msg=audit(1268279672.556:14360): arch=c000003e syscall=41 success=no exit=-13 a0=10 a1=3 a2=9
a3=0 items=0 ppid=2472 pid=3921 auid=501 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=pts0
ses=2 comm="vlock" exe="/usr/bin/vlock" subj=staff_u:secadm_r:secadm_t:s0-s15:c0.c255 key=(null)
type=AVC msg=audit(1268279672.556:14360): avc:  denied  { create } for  pid=3921 comm="vlock"
scontext=staff_u:secadm_r:secadm_t:s0-s15:c0.c255 tcontext=staff_u:secadm_r:secadm_t:s0-s15:c0.c255
tclass=netlink_audit_socket
```

在执行 vlock 期间仍然为 secadm_t。这是由于 vlock 的标签为 bin_t，而 secadm_t 具有对它的 execute_no_trans 权限，所以并不会发生 domain transition。显然，这里应该及时发现所期望的 domain transition 没有发生，而不是给 secadm_t 增加任何额外的能力！

2，在大部分规则已经开发完毕的情况下，如果希望根据 AVC 消息进一步加入新的规则时，可以采用如下方法**迅速地使用**新的规则（在 target 上执行）：

```
# copy AVC Denied Msgs into denied.txt
cat denied.txt | audit2allow -m local > local.te
checkmodule -M -m -o local.mod local.te
semodule_package -o local.pp -m local.mod
semodule -i local.pp
```

local.pp 包含新引入的规则，使用 semodule 工具将其立即加入当前 policy，还可以使用 sesearch 确认新规则的确生效了。从而避免“修改 vlock.te 文件 → 再编译 vlock.pp → 再升级 pp → 再判断”的方法，迅速地判断相应的规则是否具有期望的效果。

3，积极采用“拿来主义”。
一开始 vlock 总是无法正常使用：甚至没有弹出验证用户密码的机会，而且 AVC 消息说 PAM session 失败。最终采用照搬 newrole_t/run_init_t 所调用的接口，首先使得 vlock 能正常工作，再逐一排除的方法，最终确定需要调用 domain_use_interactive_fds(vlock_t)接口才能使得 vlock 正确地使用 PAM。

在开发.pp 时可以多多借鉴具有类似行为的命令的.pp 的实现方法！

4，对 privfd 属性的认识。
privfd 属性定义于 kernel/domain.te：

```
# widely-inheritable file descriptors
attribute privfd;
```

在 kernel/domain.if 中定义了与此相关的两个接口：

```
##      Make the file descriptors of the specified
##      domain for interactive use (widely inheritable)
```

```
domain_interactive_fd($1_t)     →      typeattribute $1 privfd

##      privfd is for passing the terminal file handle to the user process
##      Inherit and use file descriptors from
##      domains with interactive programs.
domain_use_interactive_fds($1_t)        →      allow $1 privfd:fd use;
```

注意这里有两个对象：被打开的文件和文件描述符，它们有各自的标签。这是因为它们分别对应不同的内核数据结构：inode 和 file。通常情况下，文件描述符的 type 总是继承于相应进程的 domain。根据上述注释信息，对交互式应用程序的 domain 应该调用 domain_interactive_fd 接口，使得其文件描述符加入 privfd 属性。另一方面，所有需要使用交互式应用程序已经打开文件的进程，都应该调用 domain_use_interactive_fd 接口，从而对 fd 类的 privfd 属性具有 use 能力。

一言以蔽之，父进程打开文件，子进程从父进程继承打开文件描述符，从而共享同一个 file 数据结构。由于 file 数据结构的标签等于父进程的 domain，所以需要相应的 allow 规则使得子进程 domain 能够使用它们。

```
[root/secadm_r/s0@~]# ls -lZ /proc/self/fd
lr-x------  root root root:secadm_r:secadm_t:s0-s15:c0.c1023 0 -> /dev/console
lrwx------  root root root:secadm_r:secadm_t:s0-s15:c0.c1023 1 -> /dev/console
lrwx------  root root root:secadm_r:secadm_t:s0-s15:c0.c1023 2 -> /dev/console
lr-x------  root root root:secadm_r:secadm_t:s0-s15:c0.c1023 3 -> /proc/1516/fd
[root/secadm_r/s0@~]#
[root/secadm_r/s0@~]# sesearch -SCA -s vlock_t -c fd -p use
Found 9 semantic av rules:
   allow vlock_t auditadm_t : fd use ;
   allow vlock_t privfd : fd use ;
   allow vlock_t staff_t : fd use ;
   allow vlock_t user_t : fd use ;
   allow vlock_t secadm_t : fd use ;
   allow vlock_t sysadm_t : fd use ;
   allow vlock_t unconfined_t : fd use ;
   allow vlock_t rpm_t : fd use ;
   allow vlock_t vlock_t : fd use ;
[root/secadm_r/s0@qemu-host ~]#
```

用户登录 shell 为交互式进程，由上可见，/proc/self/fd/[0|1|2]打开文件描述符的 type 和当前用户登录 shell 进程的 domain 相同。上述 allow 规则使得无论什么用户调用 vlock 时，vlock_t domain 都具有对相应 type 的文件描述符的使用能力。

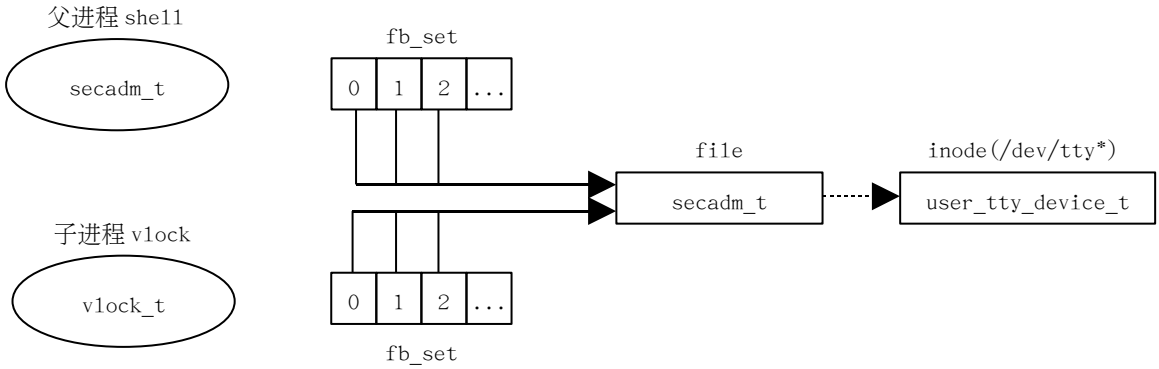
实际上，各种用户的 domain 类型都已经加入 privfd 属性了（通过调用链 userdom_unpriv_user_template > userdom_restricted_user_template > domain_interactive_fd），因此只需要使用 allow vlock_t privfd : fd use ;一条规则即可。

5，注意，上述讨论仅限于父子进程对同一个 file 数据结构的共享问题。另一方面，应用程序（子进程）还必须对相应被打开的文件自身（比如上面例子中的/dev/console 设备文件）具有读写权限。

这是显而易见的，当前进程 domain 必须具备访问和被打开文件相关的所有内核数据结构的权限！

输入输出设备可能为/dev/console，/dev/tty*，或者/dev/pts/*，它们有各自的标签：user_tty_device_t 或 user_devpts_t，所以可以就 vlock_t 调用 userdom_use_user_terminals 接口。

交互式应用程序的 domain 和外设，以及相应打开文件描述符的关系可总结如下图：




```
1. typeattribute secadm_t privfd;
2. allow vlock_t privfd:fd use;
3. allow vlock_t user_tty_device_t:chr_file rw_term_perms;
```

7.5 为 samhain 程序编写 samhain.pp

vlock 程序可以被各种角色的用户运行，只在前台运行而没有后台进程，也没有配置文件、启动脚本以及 log 和 pid 文件，其 pp 的实现相对简单。而为 samhain（文件系统一致性检查工具）编写 pp 时，就不得不考虑许多新的问题，比如上述文件的标签，samhain 后台进程的安全级别，由谁来运行或管理 samhain 后台进程。

7.5.1 第一阶段：定义基本的.te，.fc 和.if 文件

1，设计相关文件的标签，可以使用 rpm -qpl 命令得到安装文件列表。

```
/etc/rc\.d/init\.d/samhain      --      gen_context(system_u:object_r:samhain_initrc_exec_t,s0)
/etc/samhainrc                  --      gen_context(system_u:object_r:samhain_etc_t,mls_systemhigh)
/usr/sbin/samhain               --      gen_context(system_u:object_r:samhain_exec_t,s0)
/usr/sbin/samhain_setpwd        --      gen_context(system_u:object_r:samhain_exec_t,s0)
/var/lib/samhain(/.*)?                  gen_context(system_u:object_r:samhain_db_t,mls_systemhigh)
/var/log/samhain_log            --      gen_context(system_u:object_r:samhain_log_t,mls_systemhigh)
/var/log/samhain_log\.lock      --      gen_context(system_u:object_r:samhain_log_t,mls_systemhigh)
/var/run/samhain\.pid           --      gen_context(system_u:object_r:samhain_var_run_t,mls_systemhigh)
```

samhain 应用包括上述启动脚本，配置文件，可执行程序，文件系统的 signature database 文件，log 文件，log.lock 文件以及 pid 文件。分别为这些文件定义 samhain_xxx_t 标签而不是继承于父目录的缺省标签，从而减少它们能够被普通用户访问的可能性。可以定义相应的规则确保只有特定的管理员角色才能访问它们。

为了进一步提高安全性，samhain 的配置文件，signature 数据库，log 文件的安全级别都应该是 mls_systemhigh，使得相应管理员角色必须具有 clearance level 或者属于 mlsfilewrite 属性才能管理这些文件。注意，不包含关键信息的其他文件，比如启动脚本，可执行程序等，都应该属于 s0 级别。

注意，samhain 的日志文件和数据库文件含有关键信息，所以它们的安全级别为 mls_systemhigh。而它们都是由 samhain 后台进程在运行时创建、更新的，因此要求 samhain 后台进程运行在 mls_systemhigh 级别中。这一点可以通过 range_transition 规则来强制保证，参见下文。

2，声明基本类型，定义基本类型的属性及相互关系。

```
type samhain_etc_t;
files_config_file(samhain_etc_t)
```

samhain 的配置文件/etc/samhainrc，samhain_etc_t 将被加入 file_type 和 configfile 属性。

```
type samhain_exec_t;
corecmd_executable_file(samhain_exec_t)
```

/usr/sbin/samhain 可执行程序，samhain_exec_t 将被加入 file_type 和 exec_type 属性。（加入 exec_type 属性使得该类型文件能够被调用了 corecmd_xxx_all_executables 接口的 domain 以相应的方式访问）

```
type samhain_log_t;
logging_log_file(samhain_log_t)
```

samhain 的 log 和 log.lock 文件/var/log/samhain_log 和/var/log/samhain_log.lock。log.lock 文件由 samhain 后台进程在启动时创建，在结束时销毁。logging_log_file 接口将相应 type 加入 file_type 和 logfile 属性（被广泛地用于 logging_xxx_all_logs 接口中），并且和标签为 tmp_t 以及 tmpfs_t 的文件系统相关联（associate，即许可在 tmp_t 目录下和 tmpfs 中保存相应 type 的文件）。

```
type samhain_db_t;
files_type(samhain_db_t)
```

samhain 的文件系统指纹数据库文件/var/lib/samhain/samhain_file，在执行“samhain -t init”时由 samhain 后台进程创建。

```
type samhain_initrc_exec_t;
init_script_file(samhain_initrc_exec_t)
```

samhain 的启动脚本/etc/rc.d/init.d/samhain，可以通过 start/stop 控制 samhain 后台进程的启动和终止，并用 status 命令查询当前状态。注意“samhain -t init”命令运行期间 status 命令是无法正确反映其状态的（此时创建了 log.lock 文件，但是没有 pid 文件）。

```
type samhain_var_run_t;
files_pid_file(samhain_var_run_t)
```

samhain 的 pid 文件，在执行“/etc/init.d/samhain start”，或者“samhain -t check/update”命令时由 samhain 后台进程创建，同时创建的还有 log.lock 文件，它们的内容都是 samhain 后台进程的 pid。注意，由于默认/etc/samhainrc 中指定“Daemon = yes”，因此执行 check 或 update 命令结束后 samhain 后台进程仍将存在。

```
# Domain for command line access
samhain_service_template(samhain)
application_domain(samhain_t, samhain_exec_t)

# Domain for samhain service started by samhain init script
samhain_service_template(samhaind)

ifdef(`enable_mcs',`
        # This is system instead of daemon to work around
        # a type transition conflict
        init_ranged_system_domain(samhaind_t, samhain_exec_t, mcs_systemhigh)
')

ifdef(`enable_mls',`
```

```
        # This is system instead of daemon to work around
        # a type transition conflict
        init_ranged_system_domain(samhaind_t, samhain_exec_t, mls_systemhigh)
')
```

（注意，根据 cjp 的注释，把原先调用的 init_ranged_daemon_domain 修改为 init_ranged_system_domain，是作为使能 DIRECT_INITRC 时 type transition conflict 的一个 workaround，具体情况我还需要进一步验证。）

samhain 的可执行程序/usr/sbin/samhain 和/usr/sbin/samhain_setpwd 的标签都是 samhain_exec_t。当用户使用“samhain -t init”命令启动 samhain 进程时，它能够**创建、写入**数据库文件；而当使用 samhain 的启动脚本运行 samhain 后台进程时，它就只需要**读取**数据库文件了。因此在这里设计了两个 domain，分别对应从命令行启动的 samhain 后台进程（samhain_t）和以启动脚本启动的 samhain 后台进程（samhaind_t）。两个 domain 共享进行文件系统一致性检查所需要的所有能力，不同之处包括：
1）samhain_t 能够管理数据库文件，而 samhaind_t 只能够读取数据库文件；
2）samhaind_t 具有向 samhain_t 发送信号的能力（从而使得 samhain 启动脚本的 stop 命令可用）；
3）samhaind_t 需要对 samhain_exec_t 的 execute_no_trans 能力（从实践中发现）；
4）samhain_t 在用户命令行启动（执行 init 期间在前台运行，check/update 可在后台运行），需要访问用户的 tty 或者 pty 设备（user_tty_device_t 和 user_devpts_t），因此需要调用 userdom_use_user_terminals 接口。如果需要和用户交互则还必须调用 domain_use_interactive_fds 接口；
5）samhaind_t 由 run_init 启动在后台运行，需要访问的 pty 设备为 initrc_devpts_t，因此需要调用 init_use_script_ptys 接口。

分别使用 init_ranged_daemon_domain 和 application_domain 接口来声明它们和 samhain 可执行程序之间的关系。进行文件系统一致性检查所需要的所有能力都在 samhain_service_template 模板中描述，而在.te 中可以描述各自的个性。

注意，可以使用 attribute 或者 template 来描述不同 type/domain 之间的共性，在这里使用 template 显然是一个很好的选择。

为了保证 samhain_t 和 samhaind_t domain 的安全级别为 mls_systemhigh，需要如下两点支持：
1）在.te 中调用 init_ranged_daemon_domain(samhaind_t, samhain_exec_t, mls_systemhigh)接口，使得 init 进程或者 run_init（initrc_t）在执行“/etc/init.d/samhain start”时 samhaind_t 在指定的安全级别中运行；
2）在.if 的 samhain_run 接口中使用如下规则，从而确保 samhain_t 在 mls_systemhigh 安全级别中运行：

```
        ifdef(`enable_mls', `
                range_transition $1 samhain_exec_t:process mls_systemhigh;
        ')
```

那么无论任何管理员角色都应该以 mls_systemhigh 级别来运行 samhain 命令，比如：

```
        newrole -l s15:c0.c1023 -- -c “samhain -t init”
```

否则和 domain transition 相关的 MLS 约束将失败。

另外，由于 samhain 后台进程在运行时创建、写入其 log 文件，log 文件属于 mls_systemhigh 但是其所在目录/var/log/的级别为 s0，因此 samhain(d)_t 已经加入了 mlsfilewrite 属性，所以要求 samhain 后台进程运行在 mls_systemhigh 中不会有额外的副作用（比如能够在其他安全级别的终端上运行，比如 user_devpts_t:s0）。

3，在.if 文件中定义如下模板和接口：

```
template(`samhain_service_template',`                    # 描述 samhaind_t 和 samhain_t 之间的共性
interface(`samhain_domtrans',`                           # 能够运行 samhain 程序并执行 domain transition
interface(`samhain_run',`                                # 相应角色能够和 samhain_t 相关联
interface(`samhain_manage_config_files',`                # 管理 samhain 配置文件
interface(`samhain_manage_db_files',`                    # 管理 samhain 指纹数据库文件
interface(`samhain_manage_init_script_files',`           # 管理 samhain 启动脚本
interface(`samhain_manage_log_files',`                   # 管理 samhain 日志文件
interface(`samhain_manage_pid_files',`                   # 管理 samhain PID 文件
interface(`samhain_admin',`                              # 管理 samhain 进程和所有相关文件，包括/proc/pid/*
```

samhain_run 在 userdom_security_admin_template 中调用，samhain_admin 在 system.te 中调用。，尽管 samhain_manage_xxx_files 接口目前没有被使用，但是仍建议在这里定义它们，这样用户在定制自己的安全策略时就可以直接调用了。

4，在 samhain_service_template 模板中定义进行文件系统检查所需的能力。

4.0，声明 samhain_t 和 samhaind_t domain，并建立它们和 samhain 可执行文件的联系：

```
type $1_t;
domain_type($1_t)
domain_entry_file($1_t, samhain_exec_t)
```

4.1，能够读取配置文件，管理 log/log.lock 文件，管理 PID 文件

```
# 读取配置文件
read_files_pattern($1_t, etc_t, samhain_etc_t)

# 管理 PID 文件
manage_files_pattern($1_t, samhain_var_run_t, samhain_var_run_t)
files_pid_filetrans($1_t, samhain_var_run_t, file)

# 管理 log 和 log.lock 文件
manage_files_pattern($1_t, samhain_log_t, samhain_log_t)
logging_log_filetrans($1_t, samhain_log_t, file)

# PID 文件，log/log.lock 文件所在的目录都是 s0 的
mls_file_write_all_levels($1_t)
```

注意，由于 samhain 后台进程在运行时会创建 pid，log，log.lock 文件，因此必须调用相应的 xxx_xxx_filetrans 接口通过 type_transition 规则指定这些动态创建的目录和文件的标签（否则继承父目录的标签）。

另外，由于 samhain 相应文件的安全级别都是 mls_systemhigh，而所在目录/var/log/，/var/run/，/var/lib/都是 s0，因此还必须调用 mls_file_write_all_levels 接口把 samhain_t 加入 mlsfilewrite 属性，从而满足相应的 MLS 约束。

4.2，能够监控所有类型的文件。

文件的属性信息（属主，设备号，访问权限，大小，访问时间戳）等信息都保存在文件的 inode 中。为了访问各种文件的 inode，需要给 samhain(d)_t 给予 getattr 能力。对于某些 type 的文件甚至还需要给予 open/read 能力：

```
# Get the attribute of all kinds of files in the rootfs
dev_getattr_all_blk_files($1_t)        #  device_t:dir search; device_node:blk_file getattr
dev_getattr_all_chr_files($1_t)        #  device_t:dir search; device_node:blk_file getattr
dev_getattr_generic_blk_files($1_t)    #  device_t:dir search; device_t:blk_file getattr
dev_getattr_generic_chr_files($1_t)    #  device_t:dir search; device_t:blk_file getattr

files_getattr_all_dirs($1_t)           #  file_type:dir search; file_type:dir getattr
files_getattr_all_files($1_t)          #  file_type:dir search; file_type:file getattr
files_getattr_all_symlinks($1_t)
files_getattr_all_pipes($1_t)
files_getattr_all_sockets($1_t)
files_getattr_all_mountpoints($1_t)

# Read from the file_type attribute and the lnk_file class
files_read_all_files($1_t)             #  file_type:dir search; file_type:file { open read }
files_read_all_symlinks($1_t)

# For the mountpoint of /selinux, /proc, /tmp, /sys
fs_getattr_all_dirs($1_t)              # filesystem_type:dir getattr
```

注意，samhain 的行为（比如检查哪些文件，或者检查它们的哪些属性）都属于上层策略范畴，在/etc/samhainrc 中定义。而在 samhain.pp 的实现中不应该对用户态 samhain 的使用策略做任何假设，应该支持所有可能、合理的操作。

对于所有虚拟文件系统，比如/selinux/等，至多检查其挂载点目录的属性（比如设备号是否改变），而无须访问其下的内容。假设在/etc/samhainrc 中能够保证该行为，则在.te 中只需调用 fs_getattr_all_dirs 接口。

另外，由于 samhain 后台进程运行在 mls_systemhigh 安全级别中，因此可以 read down 所有安全级别的文件，故没有必要再调用 mls_file_read_all_levels 接口了。

5，定义 samhain_t 和 samhaind_t 的个性。

```
# Samhain local policy
manage_files_pattern(samhain_t, samhain_db_t, samhain_db_t)
files_var_lib_filetrans(samhain_t, samhain_db_t, { file dir })

domain_use_interactive_fds(samhain_t)
userdom_use_user_terminals(samhain_t)
```

命令行启动的 samhain 进程能够管理数据库文件，而以启动脚本启动的 samhain 进程就只能读取它：

```
# Samhaind local policy
read_files_pattern(samhaind_t, samhain_db_t, samhain_db_t)
```

6，使得 secadm 或者 sysadm 能够运行 samhain。

在 userdom_security_admin_template 中调用 samhain_run 接口：

```
optional_policy(`
        samhain_run($1, $2)
')
```

这样，当 MLS 使能时可以使用 secadm 来运行 samhain 可执行程序；当 MLS 没有使能时，使用 sysadm 来运行 samhain。由于 samhain 提供启动脚本，sysadm 总是可以通过 run_init 工具来启动、结束 samhain 服务。

另外在 sysadm.te 中调用 samhain_admin 接口：

```
optional_policy(`
        samhain_admin(sysadm_t)
')
```

由于 samhain 的用户态配置文件，log 文件，数据库文件的安全级别都是 mls_systemhigh，但是它们所处的目录都是 s0，所以在 samhain_admin 接口中调用了 mls_file_write_all_levels 接口（见下说明），使得调用者在 mls_systemhigh 级别时能够从/var/log/或/var/lib/中删除相关的文件，这样 sysadm 就能够删除 samhain 的日志和数据库文件了。

说明：根据 cjp 的意见，mls_file_write_all_levels 接口所赋予的权限太大了，因此不应在 samhain_admin 中调用（所以后来删除了该接口调用），而应该期望调用者自己已经具有了 mlsfilewrite 属性，正如假设调用者已经具有了从/var/log/和/var/lib/中删除文件的能力一样。

目前 sysadm 在 mls_systemhigh 级别中能够删除/var/lib/samhain/samhain_file 文件（因为其父目录也是 mls_systemhigh 级别的），但是无法删除/var/lib/samhain/目录以及/var/log/samhain_log 文件（由于它们的父目录都为 s0 级别的），所以只能在 permissive 模式下进行删除。

7.5.2 第二阶段：根据 AVC Denied Msg 补充相应的规则

安装 samhain.pp，更新 sysadm.pp，并给所有相关文件打上正确的标签，在 permissive 模式下使用 samhain，得到的 AVC Denied Msg 及需要补充调用的接口如下（忽略某些错误消息中的 SL 信息，因为它们在 SL 没有正确设置时获得）：

```
1. dev_read_urand($1_t)                                 # needs to read from /dev/urandom only
   dev_dontaudit_read_rand($1_t)                        # doesn't have to read from /dev/random

type=1400 audit(1289019002.120:126): avc:  denied  { open } for  pid=923 comm="samhain" name="urandom"
dev=tmpfs ino=3349 scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tcontext=system_u:object_r:urandom_device_t:s0 tclass=chr_file
type=1400 audit(1289016257.451:70): avc:  denied  { read } for  pid=917 comm="samhain" name="urandom"
dev=tmpfs ino=3349 scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tcontext=system_u:object_r:urandom_device_t:s0 tclass=chr_file

2. allow $1_t self:capability { dac_override dac_read_search fowner ipc_lock };

type=1400 audit(1289041587.271:659): avc:  denied  { dac_override } for  pid=1053 comm="samhain"
capability=1  scontext=root:sysadm_r:samhain_t:s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s15:c0.c1023
tclass=capability
type=1400 audit(1289041587.276:660): avc:  denied  { dac_read_search } for  pid=1053 comm="samhain"
capability=2  scontext=root:sysadm_r:samhain_t:s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s15:c0.c1023
tclass=capability
type=1400 audit(1289019311.787:171): avc:  denied  { fowner } for  pid=923 comm="samhain" capability=3
scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tclass=capability
type=1400 audit(1289016257.463:72): avc:  denied  { ipc_lock } for  pid=917 comm="samhain" capability=14
scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tclass=capability

3. allow samhain_t self:process { setsched setrlimit };

type=1400 audit(1289046082.563:4148): avc:  denied  { setsched } for  pid=1212 comm="samhain"
scontext=root:sysadm_r:samhain_t:s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s15:c0.c1023 tclass=process
type=1400 audit(1289019002.395:129): avc:  denied  { setrlimit } for  pid=923 comm="samhain"
scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
```

tclass=process
type=1400 audit(1289049958.091:4484): avc: denied { signull } for pid=1458 comm="samhain"
scontext=system_u:system_r:samhain_t:s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s15:c0.c1023
tclass=process
type=1400 audit(1289191452.575:161): avc: denied { signal } for pid=1162 comm="samhain"
scontext=system_u:system_r:samhain_t:s15:c0.c1023 tcontext=system_u:system_r:samhain_t:s15:c0.c1023
tclass=process

4. logging_send_syslog_msg($1_t)

type=1400 audit(1289019002.407:130): avc: denied { create } for pid=923 comm="samhain"
scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tclass=unix_dgram_socket
type=1400 audit(1289019002.407:131): avc: denied { connect } for pid=923 comm="samhain"
scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tclass=unix_dgram_socket
type=1400 audit(1289019002.416:132): avc: denied { write } for pid=923 comm="samhain" name="log"
dev=tmpfs ino=9271 scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023
tcontext=system_u:object_r:devlog_t:s15:c0.c1023 tclass=sock_file

samhain_t 和 samhaind_t 都需要操作/dev/log sock_file 文件。

5. can_exec(samhaind_t, samhain_exec_t)

type=1400 audit(1289051109.395:4629): avc: denied { execute_no_trans } for pid=1529 comm="samhain"
path="/usr/sbin/samhain" dev=sda ino=8425 scontext=system_u:system_r:samhaind_t:s15:c0.c1023
tcontext=system_u:object_r:samhain_exec_t:s15:c0.c1023 tclass=file

注意，只有当/etc/init.d/samhain start 时才会弹出上述错误信息，而 samhain -t check 可以正常执行。

6. kernel_getattr_core_if(samhain_t)

CRIT   : [2010-11-06T15:01:33+0000] interface=<1stat>, msg=<Permission denied>, userid=<0>,
path=</proc/kcore>
type=1400 audit(1289179787.105:71): avc: denied { getattr } for pid=719 comm="samhain"
path="/proc/kcore" dev=proc ino=4026531972 scontext=root:sysadm_r:samhain_t:s15:c0.c1023
tcontext=system_u:object_r:proc_kcore_t:s15:c0.c1023 tclass=file

7, allow $1_t self:process signull;

type=1400 audit(1290350545.980:226): avc: denied { signull } for pid=1228 comm="samhain"
scontext=system_u:system_r:samhaind_t:s15:c0.c1023 tcontext=system_u:system_r:samhaind_t:s15:c0.c1023
tclass=process

type=1400 audit(1290405579.564:75): avc: denied { signull } for pid=772 comm="samhain"
scontext=root:secadm_r:samhain_t:s15:c0.c1023 tcontext=root:secadm_r:samhain_t:s15:c0.c1023 tclass=process

8, allow samhaind_t { samhain_t self }:process signal_perms;

type=1400 audit(1290350862.928:630): avc: denied { sigkill } for pid=1310 comm="samhain"
scontext=system_u:system_r:samhaind_t:s15:c0.c1023 tcontext=system_u:system_r:samhaind_t:s15:c0.c1023
tclass=process

注意，如果没有使能 samhain_t:process signal 权限则/etc/init.d/stop 时会失败（无法结束后台进程，log.lock 也没有被正确删除）！

9, seutil_sigchld_newrole(samhain_t)

type=1400 audit(1289187227.147:69): avc: denied { sigchld } for pid=912 comm="newrole"

scontext=root:sysadm_r:samhain_t:s15:c0.c1023 tcontext=root:sysadm_r:newrole_t:s0-s15:c0.c1023 tclass=process

这是由于 secadm 需要在控制台使用 newrole 命令切换到 mls_systemhigh 安全级别来执行 samhain 程序，因此 samhain_t 需要给其父进程 newrole_t 发送 sigchld 的能力。

7.5.3 第三阶段：使用 dontaudit 规则屏蔽与冗余操作相关的错误信息

1. dontaudit $1_t self:capability sys_resource ;

type=1400 audit(1289019002.391:128): avc: denied { sys_resource } for pid=923 comm="samhain" capability=24 scontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tcontext=root:sysadm_r:samhain_t:s0-s15:c0.c1023 tclass=capability

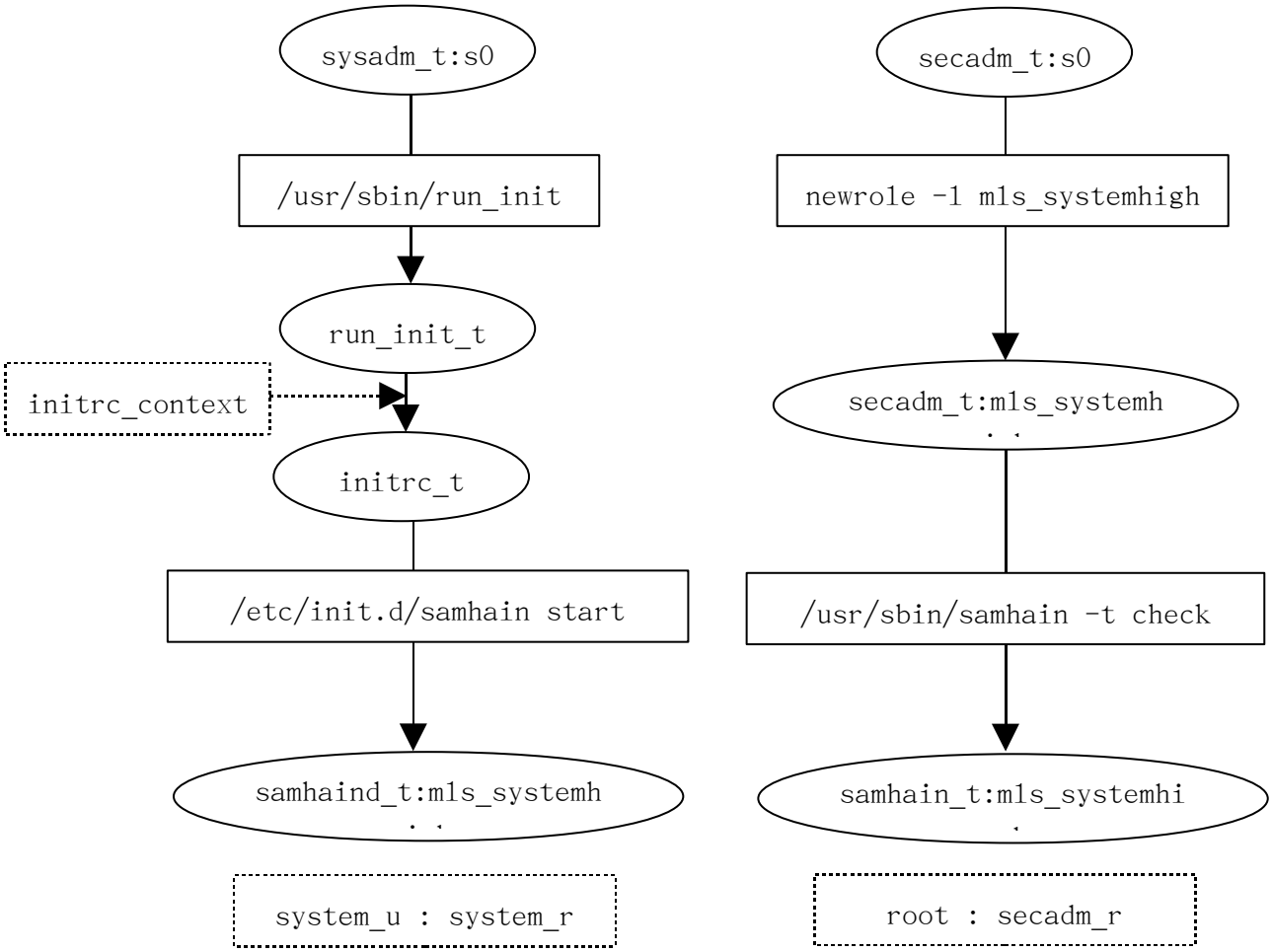
这是因为 capability 类的 sys_resource 权限将许可相应的 domain 改变系统资源的能力，比如 quota 分额，ext2 文件系统的预留空间，IPC 消息队列的长度等。samhain 作为属性检查器，应该不需要这个能力吧?

2, dontaudit $1_t self:capability sys_ptrace;

type=1400 audit(1290407525.237:286): avc: denied { sys_ptrace } for pid=925 comm="samhain" capability=19 scontext=system_u:system_r:samhaind_t:s15:c0.c1023 tcontext=system_u:system_r:samhaind_t:s15:c0.c1023 tclass=capability

sys_ptrace 能力使得进程 domain 可以 ptrace 其他进程，cjp 的意见是通常可以 dontaudit 此类错误信息。

7.5.4 图解：使用 samhain 时的 Domain Transition 过程



说明：
1，由于只在 userdom_security_admin_template 中调用 samhain_run 接口，因此当 MLS 使能时只有 secadm 才能运行/usr/sbin/samhain 程序；
2，由于/usr/sbin/samhain 的安全级别为 mls_systemhigh，所以 secadm 必须进入相应的安全级别来运行 samhain，从而确保 samhain_t 在 mls_systemhigh 中运行；
3，secadm 运行 samhain 期间，samhain 后台进程的 SC 为：root:secadm_r:samhain_t:mls_systemhigh。如果在 samhain_run 中使用 role_transition 规则，则可以改变上述 SC 中的角色为 system_r。但是目前不需要这样做。
4，由于 samhain 同时提供启动脚本，所以 sysadm 仍然可以通过 run_init 命令来启动、停止 samhain 后台进程。此时其 SC 为 system_u:system_r:samhaind_t:mls_systemhigh。其中角色 system_r 和安全级别 mls_systemhigh 都是在 samhain.te 中调用的 init_ranged_daemon_domain 接口决定的。

遗留问题：
1，为什么通过 run_init 运行时 SC 中 user 变成了 system_u 而不是 sysadm 所对应的 root?

7.6 使用 SLIDE 来开发 pp，分析 SELinux 源代码

使用 tresys 开发的 SELinux IDE(SLIDE)，可以非常方便地开发新 pp 或者分析现有 SELinux 的源代码。

比如可以象 Source Insight 那样显示选中接口或者宏的定义，非常方便。另外在开发一个新的 pp 时，可以指定相关 domain 的一些通用特性，比如应用程序的类型 Application/Daemon，是否有配置文件及其路径等，SLIDE 就可以创建基本的.te 文件和.if 文件，包含基本的 type definition 以及文件的 SC。

在 Ubuntu 上首先需要用“sudo apt-get install eclipse”安装 Eclipse，然后按照
http://oss.tresys.com/projects/slide/wiki/download
上的说明安装 SLIDE 插件即可。

SLIDE 的详细使用说明参见：
http://oss.tresys.com/projects/slide/chrome/site/help/com.tresys.slide.doc.user/webdocs.htm

7.7 编写 pp 时的注意事项

1，编写.te/.fc/.if 文件时，需要遵守 refpolicy coding style：
http://oss.tresys.com/projects/refpolicy/wiki/StyleGuide

按照指定的顺序来书写规则，调用不同 layer 中提供的接口。

2，特别注意：在.if 文件中编写注释时，一定不能包含单引号(')
否则 m4 会认为当前 block（比如当前 interface，template）结束，导致含有单引号的注释行之后的剩余行都不能被正确地解析！所以在.te/.if/.fc 中不能使用带单引号的注释，比如不能使用如下注释行：

```
# don't do this
```

经历：
在编写 samhain_admin 接口时原先调用了 mls_file_write_all_levels 接口，对 sysadm_t 调用 samhain_admin 接口，但是实际使用时发现 sysadm_t 仍然无法从/var/log/:s0 中删除 samhain_log_t:s15:c0.c1023！检查 tmp/sysadm.tmp 发现没有如期调用 mls_file_write_all_levels 接口。

百思不得其解。后来发现把 samhain_admin 中的注释如果都删掉，则 sysadm.tmp 中就可以正常调用 mls_file_write_all_levels 接口了。

现在终于明白，原先在关于 mls_file_write_all_levels 接口的注释中使用了单引号（比如 Samhain's pid, log, log.lock files are all in /var/log/ of s0, while they are all of mls_systemhigh），正是“Samhain's pid”中的这个单引号导致 m4 认为 samhain_admin 接口结束了，因此就没有正确地展开剩余的接口调用，于是在 sysadm.tmp 中就看不到对 mls_file_write_all_levels 的调用了。

3，使用 optional_policy 要注意！
如果一个 interface/template 以 optional_policy 方式被调用，如果在运行时其 gen_require 所声明的 type/attribute 不能被全部解析，则它实际上不会生效！

比如在 samhain.if 中定义 samhain_admin 接口，如果在其 gen_require 中定义了一个不存在的 type，比如：

```
interface(`samhain_admin',`
        gen_require(`
                type samhain_t, samhain_db_t, samhain_etc_t;
```

```
                type samhain_initrc_exec_t, samhain_log_t, samhain_var_run_t;
+               type doesnot_exist_type;
        ')

        allow $1 samhain_t:process { ptrace signal_perms };
        ps_process_pattern($1, samhain_t)
        ...
```

由于在 sysadm.te 中以 optional_policy 方式来调用该接口：

```
optional_policy(`
        samhain_admin(sysadm_t)
')
```

所以在编译 sysadm.pp 时编译器不会报错。甚至，在 tmp/sysadm.tmp 中发现该接口仍然能被展开：

```
##### begin samhain_admin(sysadm_t) depth:
#line 292
#line 292
#line 292
#line 292
                require {
#line 292
#line 292
                type samhain_t, samhain_db_t, samhain_etc_t;
#line 292
                type samhain_initrc_exec_t, samhain_log_t, samhain_var_run_t;
#line 292
                type doesnot_exist_type;
#line 292
#line 292
                } # end require
```

但是在实际使用中发现，升级了的 sysadm.pp 中将无法正确调用 samhain_admin 接口：

```
root@qemu-host:/root> semodule -l | grep sysadm
sysadm  2.1.4
root@qemu-host:/root> sesearch -SCA -s sysadm_t -t samhain_t -c process
Found 1 semantic av rules:
   allow sysadm_t samhain_t : process { sigchld sigkill sigstop signull signal ptrace getattr } ;

root@qemu-host:/root> semodule -u sysadm.pp                    # Update a policy module
root@qemu-host:/root> semodule -l | grep sysadm
sysadm  2.1.5
root@qemu-host:/root> sesearch -SCA -s sysadm_t -t samhain_t -c process

root@qemu-host:/root>
```

升级前，samhain_admin > ps_process_pattern($1, samhain_t) ，因此 sysadm_t 能够对 samhain_t 具有 ptrace 等能力。升级后则该能力消失了，就是因为其中调用的 doesnot_exist_type 无法被解析。

所以当以 optional_policy 方式调用的接口无法如期生效时，一定要仔细检查它所依赖的 type/attribute 在运行时是否能够被完全满足！

8. SELinux 问题分析步骤总结

当 SELinux 处于 Enforcing 模式下时某些程序的执行会失败，通常可以使用 ausearch 工具从 audit 系统中观察到相应的错误信息，但是有些时候无法直接找到对应的错误消息。在这里总结此类问题的分析方法。

8.1 排除 DAC 权限的问题

使用“ls -l”检查相关文件的属主和权限。如果 DAC 的权限许可，则就是 policy.X 中的规则显式地拒绝了当前操作的执行。另外如果是 DAC 的问题则不会有相应的 AVC Denied Msg 存在（注意逆命题不成立，因为 SELinux 可以使用 dontaudit 规则 suppress 某些错误信息）。

如果把 SELinux 切换到 Permissive 模式后问题仍然存在，那么就一定是 SELinux 阻止了程序的正常运行，真正原因往往比单纯的“相关 pp 缺少规则”要复杂的多，切记万万不可一上来就修改相关 pp 的实现。

8.2 检查用户当前所扮演的角色

某些操作只有特定的角色才有足够的权限执行，所以要注意使用正确的角色执行相应的命令。

SELinux 的 RBAC 机制将传统 UNIX root 帐户的能力划分到不同的角色中，当 MLS 特性使能时，sysadm_r，secadm_r，auditadm_r 三种角色各司其职。比如默认情况下只有 sysadm_r 能够修改/etc/、执行 dmesg；只有 secadm_r 能够调用 semodule/semanage 改变运行中的 policy 及其属性；只有 auditadm_r 才能管理和 audit 以及系统日志相关的设施。

所以当使用 sysadm_r 角色执行命令失败时，不妨尝试其他管理员角色（或者扮演 mls_systemhigh 安全级别）。

8.3 分析 AVC Denied Message 的步骤（Revisited 重要！）

从分析失败操作相关的 AVC Denied Message 入手**区分**问题的根源：

- Subject 所处的 domain 不正确，或
- Object 的标签不正确（要么没有被正确地创建，要么没有被正确地设置），或
- 缺少 TE 规则（subj_t 的确需要更多的权限），或
- 缺少 RBAC 规则（比如当前 role 无法和新 domain/type 相关联），或
- 相关约束不满足（比如 UBAC 约束，或者 MLS 约束）

使用 ausearch 工具可以得到和失败操作相关的系统调用以及 AVC 错误消息，比如：

```
[root/sysadm_r/s0@~]# date +%T
07:55:57
[root/sysadm_r/s0@~]# ls -Z /var/log/audit
ls: cannot access /var/log/audit: Permission denied
[root/sysadm_r/s0@~]# audhigh "ausearch -ts 07:55:57 -m avc -sv no"
Password:
----
time->Mon Feb 13 07:56:05 2012
type=SYSCALL msg=audit(1329119765.794:17): arch=40000003 sysca11=195 success=no exit=-13 a0=bfa8ccb4
al=85f4500 a2=b7866ff4 a3=85f44f8 items=0 ppid=1407 pid=1502 auid=4294967295 uid=0 gid=0 euid=0 suid=0
fsuid=0 egid=0 sgid=0 fsgid=0 tty=ttyS0 ses=4294967295 comm="ls" exe="/bin/ls"
subj=root:sysadm_r:sysadm_t:s0-s15:c0.c1023 key=(nu11)
```

```
type=AVC msg=audit(1329119765.794:17): avc:  denied  { getattr } for  pid=1502 comm="ls"
path="/var/log/audit" dev=sda ino=49166 scontext=root:sysadm_r:sysadm_t:s0-s15:c0.c1023
tcontext=system_u:object_r:auditd_log_t:s15:c0.c1023 tclass=dir
[root/sysadm_r/s0@~]#
```

ausearch 命令的常用参数如下：
1，“-ts”：只搜索指定时间戳之后的 log；
2，“-m avc”：只搜索类型为 AVC 的消息；
3，“-sv no”：只搜索失败消息（由于 auditallow 规则，某些成功操作也会触发 audit 记录）；
另外，还可以通过“-se”选项搜索和指定 SC 相关的错误消息，参见 ausearch 的手册页。

AVC 错误消息中包含和失败操作相关的许多信息，比如：
1，“syscall=195”，在 x86_32 体系结构上即为__NR_stat64 系统调用（参见内核源代码中 arch/x86/include/asm/unistd_32.h 文件）；
2，“success=no exit=-13”，操作失败，错误码为 13，即为 EACCESS（参见 /usr/include/asm-generic/errno-base.h）；
3，“pid=1502 uid=0 euid=0 comm=ls path=/var/log/audit”，当前进程的属主为 root，进程 ID 为 1502，正在执行的操作为 ls 命令，访问/var/log/audit 文件；
4，“scontext=root:sysadm_r:sysadm_t:s0-s15:c0.c1023”，这个是 Subject 的安全上下文；
5，“tcontext=system_u:object_r:auditd_log_t:s15:c0.c1023 tclass=dir”，这个是 Object 的安全上下文；

我们以上述错误消息为例说明分析步骤：
1，首先判断 Subject 的 domain 是否正确。
从“comm”字段可以得到所执行的程序，比如为“/bin/ls”。如果可执行程序的标签为 xxx_exec_t，则在执行该可程序期间相应进程进入 xxx_t domain；如果标签为 bin_t，则仍然在调用者 domain 中。

```
[root/sysadm_r/s0@~]# ls -Z /bin/ls
-rwxr-xr-x  root root system_u:object_r:bin_t:s0       /bin/ls
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# id -Z
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
```

所以在执行 ls 命令期间，仍然处在 sysadm_r 默认的安全上下文中。

通常情况下，只要整个文件系统都被正确地打标签（应用程序文件的标签正确），且 policy.X 能够在系统启动过程中被 init 进程顺利地装载到内核（init 进程的标签正确），则后继创建进程的 domain 就不会有错误。

2，然后判断 Object 的标签是否正确。
从“path”字段可以得到被访问对象的路径，比如为“/var/log/audit”。由于它的标签是**静态**创建的，所以可以使用 matchpathcon 命令来验证它的标签是否正确：

```
[root/sysadm_r/s0@~]# matchpathcon /var/log/audit
/var/log/audit  system_u:object_r:auditd_log_t:s15:c0.c1023
[root/sysadm_r/s0@~]#
```

所以在上述错误消息中，object 的标签时正确的。

特别需要注意的是，matchpathcon 所参照的是 file_contexts 文件中定义的静态标签，而不知道对所创建对象标签的动态修改，比如 type_transition 或 type_change 规则，或者 SELinux-aware 程序通

过/proc/self/attr/xxxcreate 接口直接指定。所以如果错误消息中反应的 obj_t 和 matchpathcon 的结果不同，还需要进一步检查是否存在相应的 type_transition 或 type_change 规则，甚至分析相应程序的源代码，考察它对 proc 接口的使用情况。

所以，一旦使用 restorecon 命令则文件的标签将被恢复为系统运行前的默认设置，这将抹杀掉系统运行期间对文件标签的动态修改（比如/dev/console 的标签在用户登录时，被 pam_selinux.so 由默认的 console_device_t 重新打标为 user_tty_device_t）。所以一定要慎用 restorecon 命令，理论上应该只在 single 模式下初次配置 SELinux 时、以及从错误中恢复时使用！

如果 Object 的标签不正确，则应该修复它的标签！（而不是盲目地给 Subject 补充规则）。比如，曾经在系统启动后看到如下错误信息：

```
type=1400 audit(1264605445.716:5): avc:  denied  { read write } for  pid=1151 comm="modprobe"
path="/dev/null" dev=tmpfs ino=1788 scontext=system_u:system_r:insmod_t:s0-s15:c0.c255
tcontext=system_u:object_r:device_t:s0 tclass=file

type=1400 audit(1264605455.536:178): avc:  denied  { write } for  pid=2251 comm="sendmail" name="null"
dev=tmpfs ino=2477 scontext=system_u:system_r:sendmail_t:s0-s15:c0.c255
tcontext=system_u:object_r:device_t:s0 tclass=chr_file
```

和相应脚本相比对，确定是在 rc.sysinit 中执行“modprobe xxxx > /dev/null 2>&1”操作以及 sendmail 脚本执行“make all -C /etc/mail -s > /dev/null”操作，重定向到/dev/null 时产生的。这两个消息显示操作发生时，/dev/null 的标签为 device_t，class 为 chr_file。

我们把它和当前 SC 中的定义相比较，发现/dev/null 的标签被定义为：

```
/dev/null       -c      system_u:object_r:null_device_t:s0
```

由此可见，/dev/null 的 type 应该是 null_device_t，而不是 device_t！显然，应该在系统启动过程中及时地修复/dev/null 的标签，而不是给相应 domain 添加关于 device_t@chr_file 的{ read write }规则（参见 8.4）小节。

3，验证是否缺少相应的 TE 规则。
从错误消息可见，sysadm_t 在使用 ls 命令试图对/var/log/audit 目录执行 getattr 操作时失败。扮演 secadm_r 角色可以使用 sesearch 命令来检查是否存在相应的 allow 规则：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s sysadm_t -t auditd_log_t -c dir -p getattr"
Password:
Found 1 semantic av rules:
   allow sysadm_t auditd_log_t : dir { ioctl read write create getattr setattr lock relabelfrom relabelto
unlink link rename add_name remove_name reparent search rmdir open } ;
[root/sysadm_r/s0@~]#
```

由此可见，sysadm_t 是具备对 auditd_log_t@dir 的 getattr 能力的。

如果确定 Subj_t 的确缺乏对 Obj_t 的 TE 规则，则在就 Subj_t 调用相应的接口以补充相应规则之前，可以考虑是否能够将 Obj_t 转换为当前 Subj_t 已经有能力操作的其他 type。比如应用程序配置文件的安装路径不正确（参见 8.5 小节），或者使用 type_transition 或 type_change 规则明确地改变新创建文件的标签，或者登录设备的标签（参见 9.1 小节）。

4，（当需要拼接新的安全上下文时）验证是否缺少相应的 RBAC 规则。

在执行新的程序或者创建新文件或内核数据对象时，都可以使用 type_transition 规则来指定当前进程的新 domain，或者重载新文件或新数据对象的标签（不采用默认的标签）。无论是否发生 role transition，在新的安全上下文中 role 都必须能够和 type 相关联，从而组成一个合法的安全上下文。所以必要时可以使用“seinfo -r -x”命令检查相关的 role 能否和新 type 向关联。必要时可以使用“seinfo -u -x”命令检查相应 SELinux User 能否扮演指定的 role。

由于在我们的例子中不需要拼接新的安全上下文，所以这种可能性无须考虑。

5，至此，检查是否存在相应的约束导致操作失败。
和 MLS 特性相关的约束实现在 policy/mls 文件中，所有和 dir 类 getattr 权限相关的约束如下：

```
# the file "read" ops (note the check is dominance of the low level)
mlsconstrain { dir file lnk_file chr_file blk_file sock_file fifo_file } { read getattr execute }
        (( l1 dom l2 ) or
         (( t1 == mlsfilereadtoclr ) and ( h1 dom l2 )) or
         ( t1 == mlsfileread ) or
         ( t2 == mlstrustedobject ));
```

由于后三条“旁路”约束的条件都不满足（可以使用 seinfo 命令检查 subj_t 和 obj_t 所在的属性），所以该约束要求在使能 MLS 时，Subject 的 low SL 必须能够 Dominate Object 的 low SL。正是由于 sysadm_r 在执行 ls 命令时处在 s0 中，所以才无法访问 SL 为 s15:c0.c1023 的/var/log/audit 目录！

另外，和 UBAC 特性相关的约束实现在 policy/constraints 文件中，必要的时候也需要检查相关约束是否被满足。

8.4 在系统启动过程中适时地修复错误的文件标签

静态文件标签不正确的原因之一：没有被正确地设置。那么可以在系统启动过程中**适时地**执行 restorecon 修复错误的标签。

在配置 SELinux 时通常已经整个文件系统设置了 SC，但是某些文件位于系统启动过程中动态挂载的文件系统内，比如/dev 上的 tmpfs：它由 rc.sysinit > start_udev 脚本挂载，后者负责创建/dev/*下的节点。此时应该在挂载了 tmpfs 文件系统之后或者创建了设备文件之后，再执行 restorecon 来正确地设置标签。

在配置 SELinux 时如果以“single selinux=1 enforcing=0”启动，则在控制台使用 restorecon 给整个文件系统打标签之前/dev/上就已经挂载了 tmpfs，此时/dev/下的原始设备节点比如/dev/console 和/dev/null，它们在挂载 tmpfs 之前的标签就还没有被正确地设置，此时就必须在 rc.sysinit 的前期调用 restorecon 正确地设置它们的标签，以保证它们能够被 mount 等程序在 rc.sysinit 调用 start_udev 之前正确地访问。另外，如果以“init=/bin/bash”启动而给整个文件系统设置标签，就不存在这个问题了。

后记：从 udev-161 版本开始 start_udev 脚本中不再挂载 tmpfs/devtmpfs 到/dev 上，这个工作转交内核完成：需要使能 CONFIG_DEVTMPFS=y 和 CONFIG_DEVTMPFS_MOUNT=y 选项。

8.5 应用程序的实际行为要和其 pp 的假设相一致

静态文件标签不正确的原因之二：由于 SELinux 发行版和实际安装的应用程序的行为或使用方式之间的差异造成的。

在为应用程序开发 pp 前，必须研究应用程序的行为并制定相应的安全目标，遵守“Least Privilege”

原则给相应 domain 赋予其能够正常执行的最小能力。**pp 的开发者所假设的应用程序的行为，和实际使用中应用程序的行为，很可能出现不一致的情况**。此时要么修改相应 pp，添加相应的规则适应应用程序的实际行为，要么修改应用程序的配置，使它的行为被 pp 所许可。对于特定发行版的开发者而言，可以使用后者，从而避免修改 pp；但是对于 refpolicy 的维护者而言，前者往往更为可取，这是因为用户态的实现方法太多了，无法面面俱到，此时修改规则则可以一劳永逸地解决问题。另外，通常可以根据不同发行版上应用程序行为的细小差异来微调相应的规则（比如使用 DISTRO 编译选项，或者使用 tunable）。

syslog-ng 是一个很好的例子。在 refpolicy-2.20091117 的实现中，logging.te 许可 syslogd_t 对标签是 syslogd_var_lib_t 的目录有{ write add_name }的能力，而对 var_t 或 var_lib_t 的目录就只有{ open search }的能力，并且在 logging.fc 中把/var/lib/syslog-ng(/.*)?的标签设置为 syslogd_var_lib_t。由此可见，logging.te 的作者假定并许可 syslogd_t 在/var/lib/syslog-ng/目录下创建自己的状态文件 syslog-ng.persist。

另一方面，在 syslog-ng 的 spec 文件中可以在 configure 中通过“--localstatedir”配置项来指定状态文件的安装路径。该选项的默认值是/var/，因此默认情况下 syslog-ng 程序会尝试在/var/下创建 syslog-ng.persist 文件。该操作一定会失败，不但因为 syslogd_t 没有在 var_t 下创建文件的能力，而且即使创建成功，logging.fc 中也没有定义/var/syslog-ng.persist 的标签，因此它的标签会被设置为 var_t，而不是 syslogd_var_lib_t，这将导致 syslogd_t 没有足够的权限支持 syslog-ng 程序对该文件的操作。

因此，必须显示地指定“--localstatedir=/var/lib/syslog-ng/”，使得 syslog-ng 程序的行为和 logging.pp 所许可的能力相一致。

另外一个好例子是 quota。quota_t 的当前实现认为 aquota.user 文件可能存在的路径为/，/boot，/home，/etc，/var/spool 等，在 quota.te 中为这些目录分别调用了 files_xxx_filetrans 接口，使得 quota_t 具有在这些目录下创建 quota_db_t 文件的能力。

但是，如果使能 quota 的文件系统在系统启动后才被手工挂载到/mnt 或/media 目录下，则不但 quota_t 没有在 mnt_t 目录下创建 quota_db_t 文件的能力，quota_t 也没有访问 mnt_t 文件的使用能力。此时应该新定义 files_mnt_filetrans 接口并对 quota_t 调用之，使得 quota_t 在 mnt_t 目录下创建 aquota.user 文件时能够自动转换其标签从 mnt_t 到 quota_db_t。另外还必须在 quota.fc 中为/mnt 和/media 下的 aquota.user 文件定义默认的标签，确保该文件如果已经存在也能够被正确地设置标签。

体会：从.fc 文件和.te 文件（files_xxx_filetrans 规则）可以确定 pp 开发者对相应程序的行为的假设，一旦和程序的实际行为不符就需要调整。

8.6 其他注意事项

8.6.1 在 Permissive 模式下调试

如果升级 refpolicy 包后发生比较严重的问题，比如无法 login，则在内核启动命令行设置 enforcing=0，即以 Permissive 模式启动 SELinux，这样可以使得 SELinux 能够正常启动，然后再用 dmesg 观察相应的 AVC 错误消息，并添加相应的规则或修复标签，直到问题解决。

如果在 SELinux 启动后执行某些操作失败，则可以使用 secadm_r 执行“setenforce 0”，并用 sysadm_r 执行“dmesg -n 8”，再执行相应操作，从而在控制台得到 AVC Denied Msg。

可以说，“进入 Permissive 模式”-> “观察 AVC Denied Msg”-> “调整应用程序的行为”／“补充规则”／“修正标签”是解决大部分 SELinux 问题的不二法门。

另外，在开发新的 domain 时，也可以把它声明为 permissive 的，从而在开发和测试周期内收集相关的 AVC 错误消息（而不影响整个系统仍然处于 enforcing 模式下）。

8.6.2 取消所有的 dontaudit 规则

AVC 错误消息是解决 SELinux 问题最有价值的线索。只要解决了所有的 AVC 错误消息，SELinux 就没有理由拒绝任何操作。但是有一个例外：SELinux 会对某些已知的、期望的失败操作使用 dontaudit 规则，从而在拒绝操作发生时不产生相应的 AVC 错误消息。因此在极端情况下，似乎没有任何可用的 AVC 错误消息，但是相应操作仍然失败。

此时可以在 target 上在 Permissive 模式下执行“semodule -DB”命令，即要求 rebuild + reload policy.X 并且去掉所有的 dontaudit 规则，这样重启后就可以看到所有被 SELinux 隐藏的失败操作了。

如果还没有找到所期望的 AVC 错误消息，则直接分析相应的.te 文件，寻找任何可能执行 dontaudit 的接口，然后去掉对该接口的调用，重新安装该模块，再重启（或执行 semodule -B），就一定能够找到相应的 AVC 错误消息。

另外一条经验就是可以尝试从不同的 terminal 上登录，比如串口控制台或 ssh，或许能够发现程序行为的区别（由相应 domain 对不同 terminal 具有不同的能力造成）。只要能以其他方式成功登录系统，就方便调试了。

8.6.3 当心 MLS Constraints

当 MLS 特性使能时，还需要判断失败操作是否是 MLS 约束造成的。通常用“no read up，no write down”原则即可判断。可以参照如下步骤来处理 MLS 约束相关的问题：
1，使用 sesearch 工具确认相应 domain 具有访问 type 的能力，从而排除缺少相应 TE 规则问题；
2，在 policy/mls 文件中查找和相应操作相关的 mlsconstrain 规则，明确满足该规则所需要的条件；
3，让 Subject 在和 Object 的安全级别相同的安全级别中运行；
4，将 Subject 或者 Object 加入相应的属性，使得上述规则为真。

比如，曾经在系统启动后发现大量错误信息：

```
type=1400 audit(1288773294.060:386): avc:  denied  { sendto } for  pid=459 comm="klogd" path="/dev/log"
scontext=system_u:system_r:klogd_t:s0-sl5:c0.cl023 tcontext=system_u:system_r:syslogd_t:sl5:c0.cl023
tclass=unix_dgram_socket

type=1400 audit(1288835627.019:398): avc:  denied  { sendto } for  pid=844 comm="logger" path="/dev/log"
scontext=system_u:system_r:initrc_t:s0-sl5:c0.cl023 tcontext=system_u:system_r:syslogd_t:sl5:c0.cl023
tclass=unix_dgram_socket

type=1400 audit(1288680823.705:390): avc:  denied  { sendto } for  pid=720 comm="unix_chkpwd"
path="/dev/log" scontext=root:sysadm_r:chkpwd_t:s0-sl5:c0.cl023
tcontext=system_u:system_r:syslogd_t:sl5:c0.cl023 tclass=unix_dgram_socket
```

首先通过 sesearch 工具确认 klogd_t 对于/dev/log 文件（sock_file），以及绑定于其上的 AF_UNIX socket 都有足够的访问权限：

```
allow klogd_t syslogd_t : unix_dgram_socket sendto ;
allow klogd_t devlog_t : sock_file { write getattr append open } ;
```

如下 MLS 约束产生了上述错误信息：

```
mlsconstrain unix_dgram_socket sendto
        (( l1 eq l2 ) or
         (( t1 == mlsnetwriteranged ) and ( l1 dom l2 ) and ( l1 domby h2 )) or
         (( t1 == mlsnetwritetoclr ) and ( h1 dom l2 ) and ( l1 domby l2 )) or
         ( t1 == mlsnetwrite ) or
         ( t2 == mlstrustedobject ));
```

klogd_t 的安全级别为 s0，而绑定在/dev/log 上的 AF_UNIX socket（syslogd_t）的安全级别为 s15:c0.c1023。为了满足上述 MLS 约束，可以把 klogd_t 加入 mlsnetwrite 属性，或者把 syslogd_t 加入 mlstrustedobject 属性。

注意，由于许多应用程序（logger/unix_chkpwd/klogd）都需要通过写入/dev/log 向 syslogd 发送消息，因此不应该把它们的 domain 都加入 mlsnetwrite 属性。相反，只需要把 syslogd_t 类型加入 mlstrustedobject 属性即可，这样任何安全级别的 domain 都能“满足”（跳过）上述 MLS 规则了。

补充说明：
1，由于在 system/logging.te 中当 MLS 被使能时就 syslogd_t 调用了如下接口：

```
init_ranged_daemon_domain(syslogd_t, syslogd_exec_t, mls_systemhigh)
```

故在系统启动时 syslogd 将运行在 mls_systemhigh 安全级别中，它创建的 AF_UNIX socket 将继承同样的 type 和安全级别。（注意，syslogd 运行时创建/dev/log 文件，并和 AF_UNIX socket 相绑定。/dev/log 的类型是 sock_file）

2，SELinux 区分 sock_file 和绑定于其上的 UNIX 套接字。比如/dev/log sock_file 的标签为 devlog_t，而相应 AF_UNIX socket 的类型为 syslogd_t，和其创建者 domain 保持一致。

3，调用 mls_trusted_object(syslogd_t)的问题是，syslogd_t 是一个进程 domain，其/proc/pid/*的标签也为 syslogd_t，调用该接口使得这些文件也属于 mlstrustedobject 属性，显然冗余了，因为我们只需要 syslogd_t 所创建的 AF_UNIX socket 的 type 属于该属性。

可能的解法：要么为 AF_UNIX socket 设计一个新的 type（比如 syslogd_s_t）且属于 mlstrustedobject 属性；要么在 syslogd 代码中调用 setsockcreatecon 函数。

4）可以在源代码目录中检查 tmp/xxx.tmp 文件来确认相应的 type 是否如期加入了某一属性。比如就 sysadm 调用了 samhain_admin 接口，后者又调用了 mls_file_write_all_levels 接口，则可以用如下方法验证 sysadm_t 已经被加入了 mlsfilewrite 属性：

```
$ grep typeattribute tmp/sysadm.tmp | grep sysadm_t | grep mls
        typeattribute sysadm_t mlsprocread;
        typeattribute sysadm_t mlsfilewrite;
```

另外，如果编译时 build.conf 中 MONOLITHIC=y，则可以检查源代码根目录下的 policy.conf 文件。

或者以 MONOLITHIC=y 编译，然后使用 apol 工具就可以直观地看到一个 type/attribute 的所有属性；或者使用 seinfo 命令。

8.6.4 检查 SELinux-aware 应用程序的配置和编译选项

必要时检查用户态 SELinux-aware 应用程序是否如期地和 libselinux 一起编译。比如 udev，应该在编译时指定“USE_SELINUX=true”，那么运行时 udevd 能够检查 SELinux 是否被使能，如果是，则在创建设备

节点时调用 libselinux 相应函数正确地设置其标签。

运行时所创建的新文件的标签由两种确定方式：
1，在应用程序相应的.te 中使用 files_xxxx_filetrans 规则；
2，在应用程序中调用 libselinux 的 API，写入当前进程的/proc/<pid>/attr/fscreate 文件。

如果应用程序创建的文件的标签不对，则也需要检查在编译时是否使能了对 SELinux 的支持！

8.6.5 积极地和社区交互

如果自己修改了 refpolicy 的当前实现，则应该积极地把 patch 发送到 refpolicy 邮件列表上讨论，这样的好处多多：
1，及时得到高人的指点，确保做出正确的判断，以正确的方式解决问题；
2，和牛人交流，能够迅速地提高自己；
3，一旦自己的 fix 被社区所接纳，则能够降低未来的维护开销；
4，扩大自己在社区内的知名度（影响力）

另外，由于 selinux/refpolicy 社区很活跃，维护者可能无法短期内就自己的 patch 做出回应，此时不要奇怪（更不能写信催促）:-P

8.6.6 使用 strace 直接定位失败的系统调用（重要！）

通常情况下错误消息都是解决缺失规则问题的好线索，但是某些情况下这种方法行不通。因为：
1，相应的错误消息并不对应程序失败的真正原因，因此根据它给相应 domain 补充更多的能力并不能解决任何问题。这样的错误消息原本可以使用 dontaudit 规则屏蔽掉；
2，由于已经存在的 dontaudit 规则，真正失败的操作可能没有错误消息产生。

可以采用更加主动、有见地的方法直接分析程序非正常退出前失败的系统调用：使用 strace 分析程序的具体行为，从而确定此时进程所处的 domain，被访问对象的 type，以及进程 domain 所缺失的能力（每个系统调用通常有 process 类的同名 permission 相对应），然后可以用“sesearch --dontaudit”命令验证相应 dontaudit 规则的存在。

详见下文 8.8.1 小节。

8.7 SELinux 问题分析过程和方法举例（Revisited）

写在前面的话：
最关键的、以及最有价值的是问题的分析过程和方法，而不在于所得到的结论。给某个 domain 调用新的接口以赋予更多的能力，这件事本身很简单（而且做多了也有些无趣），而关键在于理解为什么需要这样做。

8.7.1 实例一：用户无法在 console 上正常登录 - 使用 strace 定位失败操作

问题：在 Enforcing 模式下，如果从串口控制台登录（即内核启动命令行中指定“console=ttyS0”），则用户无法正常登录，得到如下错误信息：

```
Kernel 2.6.34.10-WR4.3.0.0_standard on  i686 (console)
QtCao login: root

Kernel 2.6.34.10-WR4.3.0.0_standard on  i686 (console)
```

```
QtCao login:
```

即输入登录用户名“root”并按下回车键后，并没有弹出“Password:”提示符从而输入密码。登录过程总是被阻塞在要求输入用户名的阶段。

这是一个非常直接和严重的问题。

【第一次尝试（被误导）】
首先以 Permissive 模式启动系统。由于此时可以正常登录，所以问题一定和 SELinux 有关。

使用 ausearch 命令得到最近的 AVC 错误消息（也可以先手工清空/var/log/audit/audit.log 文件并重启系统，从而避免无关错误消息的干扰）。既然和登录过程相关，则关注和 mingetty 或 login 程序相关的错误消息，结果如下：

```
type=AVC msg=audit(1323930310.071:5): avc:  denied  { setattr } for  pid=1360 comm="login" name="console"
dev=sda ino=57347 scontext=system_u:system_r:local_login_t:s0-s15:c0.c1023
tcontext=system_u:object_r:console_device_t:s0 tclass=chr_file
```

上述错误消息说明：login 程序所在的 domain（local_login_t）缺乏对 console 设备（console_device_t）的 setattr 能力。进一步使用 sesearch 命令验证相关权限不存在：

```
[root/sysadm_r/s0@~]# sesearch -SCA -s local_login_t -t console_device_t -c chr_file
Found 1 semantic av rules:
   allow local_login_t console_device_t : chr_file { ioctl write getattr lock relabelfrom relabelto append
open } ;
[root/sysadm_r/s0@~]#
```

问题：真的是由于缺乏这个能力导致 login 程序执行不正常么？不妨尝试添加相应的能力。

值得一提的是，在以 permissive 模式登录后发现 console 已经被正确地由 console_device_t 重新打标（relabel）为 user_tty_device_t（参见后文 9.1 小节）：

```
[root/sysadm_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_tty_device_t:s0 /dev/console
[root/sysadm_r/s0@~]#
```

从而可以断定上述失败的 setattr 操作是在 login 进程通过 pam_selinux.so 模块 relabel console 之前发生的。因为 local_login_t 对 user_tty_device_t 具有 setattr 能力（通过调用 term_setattr_all_ttys 接口。后继分析证明这里的推断是正确的，参见下文）。

由 kernel/terminal.if 可见，term_setattr_console 接口用于给调用者 domain 赋予对 console 的 setattr 能力。就 local_login_t 调用之：

```
tunable_policy(`console_login',`
+       term_setattr_console(local_login_t)
        # Able to relabel /dev/console to user tty types.
        term_relabel_console(local_login_t)
')
```

重新编译并安装 policy.X 后，相应 AVC 错误消息不再存在。但是仍然无法在 enforcing 模式下从 console 登录！所以问题另有原因。既然问题一定和 SELinux 相关，而又不存在有价值的错误消息，那么通常只有一种可能：相应 dontaudit 规则的存在使得错误消息被屏蔽了！

如前文所述，此时可以使用“semodule -DB”命令取消所有的 dontaudit 规则，然后进一步得到和 getty/login 进程相关的错误消息。但是这样做之前，还有更有针对性的分析方法。

【第二次尝试】
尽管无法从 console 登录，但是可以通过 ssh 来登录系统（此时需要把 user_devpts_t 加入 securetty_types 文件，从而在/dev/pts/*设备上能够用 newrole 来切换安全级别，进而正常使用 ausearch 命令访问 audit.log）。

新的发现如下：
1，用 ausearch 仍然无法找到有价值的线索；

2，使用“ps axj”命令，发现/sbin/mingetty 后台进程没能正常执行/bin/login 程序。即，在 console 上看到的如下信息：

```
Kernel 2.6.39+ on  i686 (console)
QtCao login:
```

它应该对应 mingetty 后台进程：

```
   1  1636  1636  1636 ttyS0     1636 Ss+      0   0:00 /sbin/mingetty console
```

在输入用户名时，mingetty 后台进程 execve 执行 login 程序；而在身份认证成功后，执行用户的默认的 shell。即在该问题被解决后，或者在 permissive 模式下，应该能看到如下结果：

```
   1  1636  1636  1636 ?           -1 Ss       0   0:00 login -- root
1636  1652  1652  1652 ttyS0     1652 Ss+      0   0:00 -bash
```

可是，在 enforcing 模式下，从 ps 结果中看不到 login 后台进程。

3，在/var/log/syslog 或者/var/log/user.log 中发现有如下错误信息：

```
Jan 10 06:52:57 QtCao login: FATAL: bad tty
```

从而进一步确认 login 后台进程的失败是和操作当前 tty 设备（即/dev/console）相关。

如果能够进一步确认失败的系统调用，则基本上就可以确定相应 domain 所需要的权限了。对于在命令行执行的程序而言，在使能 allow_ptrace 时 sysadm_r 可以使用 strace 观察应用程序所发出的系统调用及其返回值。对于 mingetty 或 login 程序而言，同样可以使用 strace 的“-p”选项“attach”到指定 pid 的进程上并开始跟踪。步骤如下：

1，以 Enforcing 模式启动，系统启动后在 console 上看到如下信息：

```
Kernel 2.6.34.10-WR4.3.0.0_standard on  i686 (console)
QtCao login:
```

2，root 以 ssh 方式登录系统，其默认角色为 staff_r，使用 newrole 切换到 sysadm_r，并验证 sysadm_t 对所有 domain 具有 ptrace 能力（即 allow_ptrace 被定义为 true）：

```
[root/staff_r/s0@~]# tty
/dev/pts/0
[root/staff_r/s0@~]# newrole -r sysadm_r -p
Password:
```

```
[root/sysadm_r/s0@~]# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 26
Policy from config file:        refpolicy-mls
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s sysadm_t -c process -p ptrace | grep domain"
Password:
   allow sysadm_t domain : process { sigchld sigkill sigstop signull signal ptrace setsched getattr } ;
[root/sysadm_r/s0@~]#
```

3，确定相关 mingetty 进程的 pid，并使用 strace attach 它：

```
[root/sysadm_r/s0@~]# ps axj | grep mingetty
    1  1391  1391  1391 tty2      1391 Ss+      0   0:00 /sbin/mingetty tty2
    1  1392  1392  1392 tty3      1392 Ss+      0   0:00 /sbin/mingetty tty3
    1  1393  1393  1393 tty4      1393 Ss+      0   0:00 /sbin/mingetty tty4
    1  1394  1394  1394 tty5      1394 Ss+      0   0:00 /sbin/mingetty tty5
    1  1395  1395  1395 tty6      1395 Ss+      0   0:00 /sbin/mingetty tty6
    1  1750  1750  1750 ttyS0     1750 Ss+      0   0:00 /sbin/mingetty console
 1725  1752  1751  1416 pts/0     1751 S+       0   0:00 grep mingetty
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# strace -p 1750 2> /root/mingetty_strace.txt
```

由此可见，打开/dev/console 的 mingetty 进程的 PID 为 1750，使用 strace 跟踪其行为。注意 strace 进程通过 stderr 输出其结果，可以把它重定向到一个文件以便后继分析。

4，此时在 console 设备上，输入登录用户名“root”后按下回车键，发现无法正常登录系统；

5，此时在第 3 步运行的 strace 进程因被调试的 1750 号进程的退出而与其剥离并结束。使用“grep -v”命令过滤 strace 结果中有关 ENOENT（打开库文件失败时）和 EBADF（关闭不存在的打开文件描述符时）的内容：

```
[root/sysadm_r/s0@~]# grep -v ENOENT /root/mingetty_strace.txt | grep -v EBADF >
/root/mingetty_strace_concised.txt
[root/sysadm_r/s0@~]# cat -n /root/mingetty_strace_concised.txt
...
[root/sysadm_r/s0@~]#
```

至此，就可以逐段分析 mingetty 程序的行为了：

6，片段一：

```
 1 Process 1750 attached - interrupt to quit
 2 read(0, "r", 1)                       = 1
 3 read(0, "o", 1)                       = 1
 4 read(0, "o", 1)                       = 1
 5 read(0, "t", 1)                       = 1
 6 read(0, "\n", 1)                      = 1
 7 execve("/bin/login", ["/bin/login", "--", "root"], [/* 8 vars */]) = 0
 8 brk(0)                                = 0x9755000
 9 fcntl64(0, F_GETFD)                   = 0
10 fcntl64(1, F_GETFD)                   = 0
11 fcntl64(2, F_GETFD)                   = 0
```

从以上片段得知，mingetty 后台进程在读取完登录用户名之后，就通过 execve 系统调用执行/bin/login 程序，即变成 login 后台进程。相应 type_transition 规则使得该进程的 domain 由 getty_t 切换到 local_login_t（这一点可以由 login 进程的/proc/pid/attr/prev 的内容确认）。

7，片段二：

```
 73 ioctl(0, SNDCTL_TMR_TIMEBASE or TCGETS, 0xbff5e334) = -1 ENOTTY (Inappropriate ioctl for device)
 74 time(NULL)                             = 1326178377
...
 82 socket(PF_FILE, SOCK_DGRAM|SOCK_CLOEXEC, 0) = 3
 83 connect(3, {sa_family=AF_FILE, path="/dev/log"}, 110) = 0
 84 send(3, "<11>Jan 10 06:52:57 login: FATAL"..., 41, MSG_NOSIGNAL) = 41
...
 89 exit_group(1)                          = ?
 90 Process 1379 detached
```

由此可见，在 login 进程因出错而退出前，它通过 AF_UNIX socket 向/dev/log 写入有关 login 进程的消息，比如“<11>Jan 10 06:52:57 login: FATAL”，而之前最近发生的错误是在对 fd[0]执行 ioctl 系统调用时发生的，而错误码正是 ENOTTY。

注意，mingetty 进程执行 login 程序后，其 fd[0]并不在 exec 时关闭。而 mingetty 进程的 fd[0]即为其父进程（init 进程）打开的/dev/console 文件！所以，这里正说明是 login 进程在就/dev/console 进行 ioctl 时发生了错误，导致 login 进程退出（然后 init 会 spawn 新的子进程执行 mingetty，重新打开 console 设备）。

8，至此，基本上可以确定 login 程序的失败是由于 local_login_t 缺乏对 console_device_t 的 ioctl 能力。使用 sesearch 确认如下：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s local_login_t -c chr_file -t console_device_t"
Password:
Found 1 semantic av rules:
   allow local_login_t console_device_t : chr_file { ioctl write getattr lock relabelfrom relabelto append
open } ;
[root/sysadm_r/s0@~]#
```

有些意外地发现，ioctl 权限已经具备。再仔细检查，居然缺少 read 权限。估计在相应 SELinux 回调函数的实现中，read 权限为 ioctl 操作的必要条件。

另外，由于错误发生时一直没有有价值的线索，因此可以推断出一定存在相应的 dontaudit 规则屏蔽了错误消息，可以使用 sesearch 加以确认：

```
[root/sysadm_r/s0@~]# seclow "sesearch -SC --dontaudit -s local_login_t -c chr_file -t console_device_t"
Password:
Found 1 semantic av rules:
   dontaudit local_login_t console_device_t : chr_file { ioctl read getattr lock open } ;
[root/sysadm_r/s0@~]#
```

所以在当前 policy.X 的实现中，不但没有赋予 local_login_t 关于 console_device_t 的 read 能力，而且进一步使用了 dontaudit 规则屏蔽了错误消息的产生。由此可见 refpolicy 的开发者是不需要要从 console 上登录的。

9，找到问题的真正原因后，需要给 local_login_t 补充对 console_device_t 的 read 能力。注意，上述

dontaudit 规则中相关的权限组合{ ioctl read getattr lock open } 对应于 read_chr_file_perms 宏:

```
define(`read_chr_file_perms',`{ getattr open read lock ioctl }')
```

所以我们可以就 local_login_t 调用 term_read_console 接口以补充 read_chr_file_perms 宏所代表的能力:

```
interface(`term_read_console',`
        gen_require(`
                type console_device_t;
        ')

        dev_list_all_dev_nodes($1)
        allow $1 console_device_t:chr_file read_chr_file_perms;
')

 tunable_policy(`console_login',`
+       # Able to ioctl /dev/console before relabeled by pam_selinux.so
+       term_read_console(local_login_t)
        # Able to relabel /dev/console to user tty types.
        term_relabel_console(local_login_t)
 ')
```

10，补充说明：相关 dontaudit 规则是在哪里引入的?

在编译 policy.X 时在 tmp/下创建的各个模块的 xxx.tmp 文件，它们可以用于确认接口调用关系及各个接口展开后的规则。

从 tmp/localllogin.tmp 文件可见，相关 dontaudit 规则通过如下接口调用引入:

```
logging_send_syslog_msg(local_login_t) > term_dontaudit_read_console(local_login_t)
```

由 logging_send_syslog_msg 接口中的注释可见，如果 syslog 后台进程不存在，则 C 库的 syslog 函数将直接写入 console 设备，所以在该接口中只赋予了调用者 domain 关于 console 的写能力，并且屏蔽了和 read 能力相关的错误信息。

至此，该问题被圆满分析并解决:-)

11，最后，通过分析 mingetty/login 进程的 strace 结果，我们可以得到如下信息:
. 如何操作 console 设备
. 何时使用 PAM 模块
. 何时、访问了什么 selinuxfs 接口文件
. 何时、访问了什么/etc/selinux/refpolicy-mls/下的设施
. 何时、如何 relabel console 设备

比如:

```
[root/sysadm_r/s0@~]# grep selinux mingetty_strace_concised_num.txt
   134  open("/lib/libselinux.so.1", O_RDONLY)  = 4
   142  statfs64("/selinux", 84, {f_type=0xf97cff8c, f_bsize=4096, f_blocks=0, f_bfree=0, f_bavail=0,
f_files=0, f_ffree=0, f_fsid={0, 0}, f_namelen=255, f_frsize=4096}) = 0
   149  open("/lib/security/pam_selinux.so", O_RDONLY) = 4
   470  open("/etc/selinux/config", O_RDONLY|O_LARGEFILE) = 3
   477  open("/selinux/mls", O_RDONLY|O_LARGEFILE) = 3
```

```
   487  open("/etc/selinux/refpolicy-mls/seusers", O_RDONLY|O_LARGEFILE) = 3
   497  open("/selinux/mls", O_RDONLY|O_LARGEFILE) = 3
   504  open("/selinux/user", O_RDWR|O_LARGEFILE) = 3
   516  open("/etc/selinux/refpolicy-mls/contexts/users/root", O_RDONLY|O_LARGEFILE) = 3
   522  open("/etc/selinux/refpolicy-mls/contexts/default_contexts", O_RDONLY|O_LARGEFILE) = 3
   528  open("/selinux/mls", O_RDONLY|O_LARGEFILE) = 3
   547  getxattr("/dev/console", "security.selinux", "system_u:object_r:console_device_t:s0", 255) = 38
   554  open("/selinux/relabel", O_RDWR|O_LARGEFILE) = 3
   562  setxattr("/dev/console", "security.selinux", "root:object_r:user_tty_device_t:s0", 35, 0) = 0
[root/sysadm_r/s0@~]#
```

比如在 554 行通过/selinux/relabel 文件，向 policy.X 查询当前用户在/dev/console 上登录时应该如何 relabel console。打开 strace 结果文件，可以看到 login 进程先向/selinux/relabel 文件写入当前用户 SC 字符串，设备 SC 字符串和设备类型字符串，然后再以阻塞方式（注意，open 时没有指定 O_NONBLOCK 标志）读取其返回结果：

```
   554  open("/selinux/relabel", O_RDWR|O_LARGEFILE) = 3
   555  write(3, "root:sysadm_r:sysadm_t:s0-s15:c0"..., 79) = 79
   556  read(3, "root:object_r:user_tty_device_t:"..., 4095) = 35
```

由此可见，当 sysadm_t 在 console 上登录时，它将被 relabel 为 user_tty_device_t。另外该过程还可以用 compute_relabel 命令复现如下：

```
[root/sysadm_r/s0@~]# compute_relabel
usage:  compute_relabel scontext tcontext tclass
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# id -Z
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# matchpathcon /dev/console
/dev/console    system_u:object_r:console_device_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_relabel `id -Z` system_u:object_r:console_device_t:s0 chr_file
root:object_r:user_tty_device_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_tty_device_t:s0 /dev/console
[root/sysadm_r/s0@~]#
```

8.7.2 实例二：sysadm_r 无法正确使用 useradd 命令

（注，这个例子已经很老了，差不多是在 2008 年左右碰到的，而且后来更新后的 refpolicy 中也已解决了这个问题。但是这并不妨碍它作为一个很好的例子来展示问题的分析过程）

问题：在 Enforcing 模式下，root 用户扮演 sysadm_r 角色时调用 useradd/userdel 失败，比如：

```
root@cp3020:/root> id -Z
root:sysadm_r:sysadm_t:s0-s15:c0.c255
root@cp3020:/root> getenforce
Enforcing
root@cp3020:/root> useradd -m harry
useradd: PAM authentication failed
root@cp3020:/root>
```

首先排除 DAC 问题－确保当前用户具有命令的执行权限：

```
root@cp3020:/root> ls -lt /usr/sbin/useradd
-rwxr-x--- 1 root root 89992 Oct 13 09:55 /usr/sbin/useradd
```

从而确定是 SELinux 阻止了该命令的运行。sysadm_r 应该能够在 Enforcing 下自由地添加、删除用户，所以排除角色的问题。另外也不是 boolean 的问题。

第一步，获得命令执行时的时间戳，扮演 auditadm_r 角色，用 ausearch 工具在 audit 日志中查找相关的 SELinux AVC 记录：

```
root@cp3020:/root> date=`date +%H:%M:%S`
root@cp3020:/root> echo $date
19:57:36
root@cp3020:/root> useradd -m harry
useradd: PAM authentication failed
root@cp3020:/root> newrole -r auditadm_r -l sl5:c0.c255
Password:
root@cp3020:/root> ausearch -c useradd -ts 19:57:36 -ue 0 -sv no
----
time->Wed Sep 16 19:57:46 2009
type=PATH msg=audit(1253131066.534:3941): item=0 name="/sbin/unix_chkpwd" inode=3596402 dev=08:01
mode=0104755 ouid=0 ogid=0 rdev=00:00 obj=system_u:object_r:chkpwd_exec_t:s0
type=CWD msg=audit(1253131066.534:3941):  cwd="/root"
type=SYSCALL msg=audit(1253131066.534:3941): arch=c000003e sysca11=59 success=no exit=-13
a0=7f20d0242f98 al=78e407085c60 a2=7f20d0444e30 a3=304554da70 items=1 ppid=5578 pid=5579 auid=0 uid=0 gid=0
euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=ttyS0 ses=226 comm="useradd" exe="/usr/sbin/useradd"
subj=root:sysadm_r:useradd_t:s0-sl5:c0.c255 key=(nu11)
type=AVC msg=audit(1253131066.534:3941): avc:  denied  { execute } for  pid=5579 comm="useradd"
name="unix_chkpwd" dev=sdal ino=3596402 scontext=root:sysadm_r:useradd_t:s0-sl5:c0.c255
tcontext=system_u:object_r:chkpwd_exec_t:s0 tclass=file
root@cp3020:/root>
```

注：
1，ausearch 参数的用法参见其手册；
2，如果非 MLS，则 secadm_r & auditadm_r 的能力都集中于 sysadm_r(更像 UNIX 中的 root 用户了)；

由上述 AVC denied message 可见，在执行 useradd 命令时进程的 domain 已经切换到 useradd_t，它没有对 chkpwd_exec_t 的“execute”权限。后者正是/sbin/unix_chkpwd 程序的 type：

```
root@cp3020:/root> ls -lt /sbin/unix_chkpwd -Z
-rwsr-xr-x  root root system_u:object_r:chkpwd_exec_t:s0 /sbin/unix_chkpwd
```

所以，useradd_t 没有 chkpwd_exec_t 的执行权限，导致当前进程无法切换到 chkpwd_t。调用 unix_chkpwd 失败将导致其调用者 pam_unix 模块失败返回，于是看到“useradd: PAM authentication failed ”错误信息。

第二步，在 policy/modules/目录树下，查找定义了 chkpwd_exec_t 的.te 文件：

```
refpolicy/policy/modules$ find -name *.te -exec grep chkpwd_exec_t {} + | grep type
./system/authlogin.te:type chkpwd_exec_t;
```

第三步，在相应的.if 文件中，查找相应的接口，比如

```
########################################
## <summary>
##      Run unix_chkpwd to check a password.
```

```
## </summary>
## <param name="domain">
##      <summary>
##      Domain allowed access.
##      </summary> refpolicy
## </param>
#
interface(`auth_domtrans_chk_passwd',`
        gen_require(`
                type chkpwd_t, chkpwd_exec_t, shadow_t;
        ')

        corecmd_search_bin($1)
        domtrans_pattern($1, chkpwd_exec_t, chkpwd_t)
        ...
')
```

第四步，查找定义 useradd_t 的.te 文件，在其中调用 auth_domtrans_chk_passwd 接口使得 useradd_t 具备该接口所赋予的权限：

```
refpolicy/policy/modules$ find -name *.te -exec grep useradd_t {} + | grep type
./admin/usermanage.te:type useradd_t;
./admin/usermanage.te:role system_r types useradd_t;

--- a/policy/modules/admin/usermanage.te
+++ b/policy/modules/admin/usermanage.te
@@ -468,6 +468,7 @@ auth_etc_filetrans_shadow(useradd_t)
 auth_rw_lastlog(useradd_t)
 auth_rw_faillog(useradd_t)
 auth_use_nsswitch(useradd_t)
+auth_domtrans_chk_passwd(useradd_t)

 init_use_fds(useradd_t)
 init_rw_utmp(useradd_t)
```

然后将编译好的 usermanage.pp 拷贝到 target 上，使用 secadm_r 角色在 Permissive 下使用命令“semodule -u usermanager.pp”来更新该 pp（单独升级一个 pp 需要递增.te 中的版本号），并可以用“semodule -l | grep usermanage”来观察升级前后该 pp 版本号的变化。

之后遇到的新问题如下，采用同样的方法进行分析：

```
   root@cp3020:/root> id -Z
   root:sysadm_r:sysadm_t:s0-s15:c0.c255
   root@cp3020:/root> getenforce
   Enforcing
   root@cp3020:/root> date=`date +%H:%M:%S`
   root@cp3020:/root> echo $date
   00:16:27
   root@cp3020:/root> useradd -m tester
   useradd: cannot create directory /home/tester
   root@cp3020:/root> newrole -r auditadm_r -l s15:c0.c255
   Password:
   root@cp3020:/root> ausearch -c useradd -ts 00:16:27 -ue 0 -sv no
   ----
   time->Thu Sep 17 00:16:50 2009
   type=PATH msg=audit(1253146610.660:198): item=0 name="/home/tester" inode=2932737 dev=08:01 mode=040755
ouid=0 ogid=0 rdev=00:00 obj=system_u:object_r:home_root_t:s0-s15:c0.c255
```

```
   type=CWD msg=audit(1253146610.660:198):  cwd="/root"
   type=SYSCALL msg=audit(1253146610.660:198): arch=c000003e sysca11=83 success=no exit=-13 a0=814ea0 al=0
a2=0 a3=6165726373662f72 items=1 ppid=2371 pid=2378 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0
fsgid=0 tty=ttyS0 ses=1 comm="useradd" exe="/usr/sbin/useradd" subj=root:sysadm_r:useradd_t:s0-s15:c0.c255
key=(nu11)
   type=AVC msg=audit(1253146610.660:198): avc:  denied  { create } for  pid=2378 comm="useradd"
name="tester" scontext=root:sysadm_r:useradd_t:s0-s15:c0.c255 tcontext=user_u:object_r:user_home_dir_t:s0-
s15:c0.c255 tclass=dir
   root@cp3020:/root>
```

由此可见，现在的问题是 useradd_t 没有对 user_home_dir_t 的“create”权限。继续修改 usermanage.te，对 useradd_t 调用 system/userdomain.if 导出的 userdom_create_user_home_dirs 接口，使得 useradd_t 具备相应的权限：

```
--- a/policy/modules/admin/usermanage.te
+++ b/policy/modules/admin/usermanage.te
@@ -485,10 +486,13 @@ seutil_domtrans_setfiles(useradd_t)

 userdom_use_unpriv_users_fds(useradd_t)
 # Add/remove user home directories
+userdom_create_user_home_dirs(useradd_t)
 userdom_manage_user_home_content_dirs(useradd_t)
 userdom_manage_user_home_content_files(useradd_t)
 userdom_home_filetrans_user_home_dir(useradd_t)
 userdom_user_home_dir_filetrans_user_home_content(useradd_t, notdevfile_class_set)
+# Set attributes in the user home directories
+userdom_setattr_user_home_contents(useradd_t)
```

更新了 usermanage.pp 后，最后遇到的新问题如下，采用相同方法分析：

```
   root@cp3020:/root> id -Z
   root:sysadm_r:sysadm_t:s0-s15:c0.c255
   root@cp3020:/root> getenforce
   Enforcing
   root@cp3020:/root> useradd -m tester
   root@cp3020:/root> ls -lt /home/ | grep tester
   d--------- 2 root       root       4096 Sep 17 05:35 tester
   root@cp3020:/root> ssh tester@localhost
   Password:

   Could not chdir to home directory /home/tester: Permission denied
   -bash: /home/tester/.bash_profile: Permission denied
   tester@cp3020:/$ id -Z
   user_u:user_r:user_t:s0
   teste2@cp3020:/$ exit
   logout
   -bash: /home/tester/.bash_logout: Permission denied
   Connection to local closed.
   root@cp3020:/root> newrole -r auditadm_r -l s15:c0.c255
   Password:
   root@cp3020:/root> ausearch -c useradd -ue 0 -sv no
   ...
   time->Thu Sep 17 00:35:04 2009
   type=PATH msg=audit(1253147704.032:257): item=0 name="/home/tester" inode=2933879 dev=08:01 mode=040000
ouid=0 ogid=0 rdev=00:00 obj=user_u:object_r:user_home_dir_t:s0-s15:c0.c255
   type=CWD msg=audit(1253147704.032:257):  cwd="/root"
   type=SYSCALL msg=audit(1253147704.032:257): arch=c000003e sysca11=90 success=no exit=-13 a0=814ea0
a1=1c0 a2=0 a3=0 items=1 ppid=2415 pid=2420 auid=0 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0
```

```
tty=ttyS0 ses=1 comm="useradd" exe="/usr/sbin/useradd" subj=root:sysadm_r:useradd_t:s0-s15:c0.c255
key=(null)
   type=AVC msg=audit(1253147704.032:257): avc:  denied  { setattr } for  pid=2420 comm="useradd"
name="tester" dev=sda1 ino=2933879 scontext=root:sysadm_r:useradd_t:s0-s15:c0.c255
tcontext=user_u:object_r:user_home_dir_t:s0-s15:c0.c255 tclass=dir
```

现在的问题是 useradd_t 没有对 user_home_dir_t 的 setattr 权限，才导致新用户 HOME 目录的 uid/gid 和 DAC 权限都不正确。我们需要对 useradd_t 赋予如下能力：

```
allow useradd_t { user_home_t user_home_dir_t }:file setattr;
```

由于 pp 封装性的要求，我们不能把上述 allow 语句直接写在 usermanager.te 中，可以在 userdomain.if 中增加一个 interface，使得调用者对 HOME 目录以及其下文件都有 setattr 权限：

```
--- a/policy/modules/system/userdomain.if
+++ b/policy/modules/system/userdomain.if
@@ -1611,6 +1611,25 @@ interface(`userdom_dontaudit_setattr_use

+########################################
+ ## <summary>
+##     Set attributes in the user home directories
+## </summary>
+## <param name="domain">
+##     <summary>
+##     Domain allowed access.
+##     </summary>
+## </param>
+#
+interface(`userdom_setattr_user_home_contents',`
+       gen_require(`
+               type user_home_t, user_home_dir_t;
+       ')
+
+       allow $1 user_home_t:file setattr;
+       allow $1 user_home_dir_t:dir setattr;
+')
+
```

然后在 usermanage.te 中对 useradd_t 调用该接口即可：

```
--- a/policy/modules/admin/usermanage.te
+++ b/policy/modules/admin/usermanage.te
@@ -485,10 +486,13 @@ seutil_domtrans_setfiles(useradd_t)

 userdom_use_unpriv_users_fds(useradd_t)
 # Add/remove user home directories
+userdom_create_user_home_dirs(useradd_t)
 userdom_manage_user_home_content_dirs(useradd_t)
 userdom_manage_user_home_content_files(useradd_t)
 userdom_home_filetrans_user_home_dir(useradd_t)
 userdom_user_home_dir_filetrans_user_home_content(useradd_t, notdevfile_class_set)
+# Set attributes in the user home directories
+userdom_setattr_user_home_contents(useradd_t)
```

至此，sysadm_r 在 Enforcing 模式下就可以成功调用 useradd/userdel 了：

```
   root@cp3020:/root> id -Z
   root:sysadm_r:sysadm_t:s0-s15:c0.c255
```

```
root@cp3020:/root> getenforce
Enforcing
root@cp3020:/root> useradd -m tester04
root@cp3020:/root> ls -lt /home/ | grep tester04
total 88
drwx------ 2 tester04   users      4096 Sep 17 06:00 tester04
root@cp3020:/root> passwd tester04
New UNIX password:
BAD PASSWORD: is too simple
Retype new UNIX password:
passwd: password updated successfully
root@cp3020:/root> ssh tester04@localhost
Password:

tester04@cp3020:~$ id -Z
user_u:user_r:user_t:s0
tester04@cp3020:~$ pwd
/home/tester04
tester04@cp3020:~$ touch 1
tester04@cp3020:~$ ls -Za
drwx------  tester04 users user_u:object_r:user_home_dir_t:s0 .
drwxr-xr-x  root     root  system_u:object_r:home_root_t:s0-s15:c0.c255 ..
-rw-r--r--  tester04 users user_u:object_r:user_home_t:s0   1
tester04@cp3020:~$ exit
logout
Connection to localhost closed.
root@cp3020:/root>
root@cp3020:/root> getenforce
Enforcing
root@cp3020:/root> userdel tester04
root@cp3020:/root> id tester04
id: tester04: No such user
root@cp3020:/root>
```

9. SELinux 开发笔记

本章节内容为作者为 SELinux 或 refpolicy 开发的若干新特性，目前都已经被社区所采纳。在这里详细记录每个开发活动的原始需求（待解决的问题）、设计过程、实现细节、测试结果和其它经验总结。建议先阅读完 10～12 章的相应内容后，再看本章的相关内容。

9.1 使能对/dev/console 的支持

系统启动后，可以通过 console 或者 tty 设备登录。比如在/etc/inittab 中有如下设置：

```
# Run gettys in standard runlevels
1:2345:respawn:/sbin/mingetty console
2:2345:respawn:/sbin/mingetty tty2
3:2345:respawn:/sbin/mingetty tty3
4:2345:respawn:/sbin/mingetty tty4
5:2345:respawn:/sbin/mingetty tty5
6:2345:respawn:/sbin/mingetty tty6
```

则 mingetty 程序打开并初始化 console，然后执行 login 程序，或者通过 PAM 进行身份认证，创建子进程执行用户的登录 shell，借助 pam_selinux.so 设置用户登录 shell 的安全上下文，并 relabel 当前的 terminal 设备（强烈建议使用 8.8.1 小节中的方法，用“strace -p”命令观察 mingetty/login 后台进程的具体行为）。

假设使用串口控制台（即使用“console=ttyS0，115200n8”为内核启动参数），则从 console 登录后可以看到相应后台进程之间的关系如下：

```
[root/sysadm_r/s0@~]# ps axj
 PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
    0     1     1     1 ?           -1 Ss       0   0:01 init [3]
    1   869   869   869 ?           -1 S<s      0   0:00 /sbin/udevd -d
    1  1162  1162  1162 ?           -1 Ss       0   0:00 syslogd -m 0
    1  1166  1166  1166 ?           -1 Ss       0   0:00 klogd -x
    1  1182  1182  1182 ?           -1 Ss      32   0:00 rpcbind
    1  1200  1200  1200 ?           -1 Ss       0   0:00 rpc.statd
    1  1210  1210  1210 ?           -1 Ss       0   0:00 watchquagga -d zebra bg
    1  1235  1235  1235 ?           -1 Ss       0   0:00 rpc.idmapd
    1  1242  1242  1242 ?           -1 Ss1      0   0:00 /usr/bin/tcf-agent -d -
    1  1253  1253  1253 ?           -1 Ss1      0   0:00 dbus-daemon --system
    1  1277  1277  1277 ?           -1 Ss       0   0:00 /usr/sbin/acpid
    1  1286  1286  1286 ?           -1 Ss       0   0:00 /usr/sbin/sshd
    1  1296  1296  1296 ?           -1 Ss       0   0:00 xinetd -stayalive -pidf
    1  1314  1314  1314 ?           -1 Ss       0   0:00 sendmail: accepting con
    1  1324  1324  1324 ?           -1 Ss      51   0:00 sendmail: Queue runner@
    1  1332  1329  1130 ?           -1 S       99   0:00 boa
    1  1339  1339  1339 ?           -1 Ss       0   0:00 crond
    1  1349  1349  1349 ?           -1 Ss       0   0:00 /usr/sbin/atd
    1  1357  1357  1357 ?           -1 Ss       0   0:00 login -- root
    1  1358  1358  1358 tty2      1358 Ss+      0   0:00 /sbin/mingetty tty2
    1  1359  1359  1359 tty3      1359 Ss+      0   0:00 /sbin/mingetty tty3
    1  1360  1360  1360 tty4      1360 Ss+      0   0:00 /sbin/mingetty tty4
    1  1361  1361  1361 tty5      1361 Ss+      0   0:00 /sbin/mingetty tty5
    1  1362  1362  1362 tty6      1362 Ss+      0   0:00 /sbin/mingetty tty6
  869  1369   869   869 ?           -1 S<       0   0:00 /sbin/udevd -d
  869  1370   869   869 ?           -1 S<       0   0:00 /sbin/udevd -d
```

```
 1357  1373  1373  1373 ttyS0      1433 Ss       0   0:00 -bash
 1373  1433  1433  1373 ttyS0      1433 R+       0   0:00 ps axj
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# cat /proc/1357/attr/prev
system_u:system_r:getty_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# cat /proc/1357/attr/current
system_u:system_r:local_login_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
```

由此可见：
1，init 进程创建子进程（1357～1362）执行 mingetty 程序打开并初始化 console 或者 tty 设备；
2，mingetty 后台进程（1357）直接执行 login 程序，从而变成 login 后台进程。其 domain 随即也由 getty_t 切换为 local_login_t（可以由 ssh 登录系统并使用“strace -p 1357”观察输入登录用户名之前 mingetty 后台进程的行为，参见 8.8.1 小节）；
3，login 程序创建子进程（1373）执行 root 用户的登录 shell，即为 bash；
4，ps 输出结果中“TTY”表示当前的 terminal 设备，有可能时 tty，pty 或 console。对应内核线程或者系统后台进程而言不需要使用 terminal，因此它们的 TTY 一栏均为“?”。由于使用串口控制台，root 用户的登录 shell 所使用的 terminal 为 ttyS0。

9.1.1 提出问题：20101213 及之前的 refpolicy 缺乏对 console 的支持

1，在 Enforcing 模式下无法在 console 上正常登录。控制台上显示如下信息：

```
INIT: Id "1" respawning too fast: disabled for 5 minutes
INIT: no more processes left in this runlevel
```

其中 Id“1”即代表/etc/inittab 中“mingetty console”那一行。由此可见 mingetty 程序无法正常打开并初始化 console 设备。另外即使在 Permissive 模式下登录后，也找不到相应的 AVC Denied Message，由此可见一定存在相关的 dontaudit 规则屏蔽了错误消息。

2，在 Permissive 模式下从 console 登录后，可以看到关于各种用户类型在使用/dev/console 时的 AVC Denied Message。

3，若用户在执行应用程序时切换到新的 domain，则新 domain 在使用 console 时产生 AVC Denied Message。

4，用户在使用 newrole 切换到其他角色时将产生相应的 AVC Denied Message。

9.1.2 分析问题

针对上面关于 console 的使用问题，逐项分析如下：
1，mingetty 程序打开并初始化/dev/console 失败，由于 mingetty 进程的 domain（getty_t）和/dev/console 的标签正确（console_device_t），原因只能是 getty_t 缺少对 console_device_t 的相应权限。

果然，在 getty.te 中就 getty_t 调用了 term_donaudit_use_console 接口！则不但没有赋予 getty_t 就 console_device_t 的 read/write 能力，甚至还进一步使用了 dontaudit 规则屏蔽的相应的错误信息。这正是在 Permissive 模式下看不到相应 AVC Denied Message 的原因。

另外，该 dontaudit 规则的使用恰恰说明 refpolicy 的开发者们**有意识地**没有添加对 console 的使用支持！应该是在他们的开发环境中只能从 tty 设备上登录而不需要从 console 上登录，另外的一个证据是在

config/appconfig-x/securetty_types 文件中只包含了 user_tty_device_t，而没有包含 console_device_t。即用户在本地 tty 上登录后能够使用 newrole 改变安全级别，如果从 console 或者远程 ssh 登录（使用 pty 设备），则不能改变安全级别。显然，console 和 tty 一样都在本地，则应该允许在 console 上改变用户 domain 的安全级别（注意 tty/console 的安全级别随之改变）。

2，用户从 tty 或者 pty 设备上登录后，相应 terminal 设备的标签被 login/remote/sshd 程序通过 pam_selinux.so 模块 relabel 为 user_tty_device_t 和 user_devpts_t。所有用户类型均通过 userdom_base_user_template 模板创建，它就各种用户 domain 赋予了如下和 terminal 相关的权限：

```
term_user_pty($1_t, user_devpts_t)              # type_change devpts_t -> user_devpts_t
term_user_tty($1_t, user_tty_device_t)          # type_change tty_device_t -> user_tty_device_t
term_create_pty($1_t, user_devpts_t)            # type_transition devpts_t -> user_devpts_t
allow $1_t user_devpts_t:chr_file { setattr rw_chr_file_perms };
allow $1_t user_tty_device_t:chr_file { setattr rw_chr_file_perms };
```

但是没有赋予用户 domain 就 console_device_t 的 type_change 以及 rw_chr_file_perms 权限。因此用户的登录 shell 或者在用户 domain 中执行的应用程序在读写当前 terminal=console 时将产生相应的错误信息。

3，由于用户登录后 tty 和 pty 设备被 relabel 为 user_tty_device_t 和 user_devpts_t，许多应用程序的 domain 都具备对它们的读写能力，这通过调用 userdom_use_user_terminals 接口实现。但是这些 domain 都没有调用 term_use_console 接口，因此无法正常使用 terminal=console。

4，能够 relabel tty 设备的应用程序 domain 都调用了 term_relabel_all_ttys 接口，但是它们没有调用 term_relabel_console 接口，因此不具备对 console 的 relabelfrom/to 能力。

进一步，在 userdom_base_user_template 模板中也没有指明应该如何给 console 重新打标签（即，不存在相应的 type_change 规则以指明 console 的新标签）。

9.1.3 解决问题

就上述各种 domain 缺少对 console_device_t 的 read/write/relabel 权限的问题，解决方法如下：
1，毫无疑问，必须就 getty_t 调用 term_use_console 接口，使得能够从 console 上正常登录；

（不合理）2，可以就各种用户 domain 调用 term_use_console 接口，使得用户登录 shell 能够读写 console。对该接口的调用可以放到 userdom_base_user_template 模板中；

（不合理）3，可以就需要访问 terminal 的各种应用程序 domain 补充调用 term_use_console 接口。由于此类 domain 为数众多而且大多都调用了 logging_send_syslog_msg 接口，因此可以在该接口内增加调用 term_use_console 接口；

4，可以就当前能够 relabel tty 设备的 domain，继续调用 term_relabel_console 接口，可以把它放到 term_relabel_all_ttys 接口中。

上面的方案存在两个问题。第 3 点的毛病最多，因为“使用 syslog 的 domain”并不等于“使用 terminal 的 domain”。需要使用 syslog 的 domain 才调用了 logging_send_syslog_msg 接口，在其中调用了 term_write_console 接口的目的是在写入 syslog 时能够同时将日志写入 console。而这些应用程序可能根本不需要使用 terminal，或者所使用的 terminal 未必是 console，因此在这里补充调用 term_read_console 接口是不合适的。另外没有和第 4 点相配合的 type_change 规则。

如果一个 domain 缺乏对某种 type（比如 console_device_t）的访问能力，但是具备对其他 type（比如 user_tty_device_t）的相应能力，除了采用上面的思路给这个 domain 增加对第一种 type 的访问的能力之外，另一种思路是将第一种 type **转化**为第二种 type，这样无须给 domain 新增任何其他的能力！

由于无法 read/write/relabel console_device_t 的 domain 已经具备对 user_tty_device_t 的相应能力，因此只需要想办法在用户登录时把/dev/console 的标签 relabel 为 user_tty_device_t 即可！方法如下：

2，在 userdom_base_user_template 模板中定义关于 console 的 type_change 规则，所有用户类型从 console 登录后其标签被 relabel 为 user_tty_device_t。该规则可以放到 term_user_tty 接口中：

```
type_change $1 console_device_t:chr_file $2;
```

3，使得 local_login_t 能够 relabel console_device_t，即在 locallogin.te 中调用：

```
term_relabel_console(local_login_t)
```

4，可以把上述行为用一个 tunable（console_login）来控制，用户可以根据自己系统的实际使用情况决定是否支持上述行为。由于 console_login 将在多个 pp 中使用，所以需要把它定义在 global_tunables 文件中（而只在单个 pp 中使用的 tunable 或 boolean，则在相应.te 中定义即可，比如 allow_ptrace 只在 sysadm.te 中定义和使用）。

这样，我们就无须担心识别、补充给相应 domain 关于 console_device_t 的相应能力了，该标签在用户登录后将不复存在（除非以 single 方式启动系统，绕过 login > pam_selinux.so 过程）。

9.1.4 测试结果

登录后可以观察/dev/console 设备当前的安全上下文，policy 中定义的默认安全上下文，以及被 relabel 的新的安全上下文如下：

```
[root/sysadm_r/s0@~]# tty
/dev/console
[root/sysadm_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_tty_device_t:s0 /dev/console
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# matchpathcon `tty`
/dev/console    system_u:object_r:console_device_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_relabel root:sysadm_r:sysadm_t:s0-s15:c0.c1023
system_u:object_r:console_device_t:s0 chr_file
root:object_r:user_tty_device_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s sysadm_t -t console_device_t"
Password:
Found 1 semantic te rules:
ET type_change sysadm_t console_device_t : chr_file user_tty_device_t; [ console_login ]
[root/sysadm_r/s0@~]#
```

由此可见，在 root 用户登录后/dev/console 设备的 SC 的 user 和 type 都被重新打标了。

9.1.5 使用 strace 观察 console 被重新打标签的细节（new）

采用 8.8.1 小节中的方法，我们可以用 strace 来观察 login 后台进程给/dev/console 重新设置标签的完

整细节，比如相应 strace 结果片段如下：

```
547 getxattr("/dev/console", "security.selinux", "system_u:object_r:console_device_t:s0", 255) = 38
...
554 open("/selinux/relabel", O_RDWR|O_LARGEFILE) = 3
555 write(3, "root:sysadm_r:sysadm_t:s0-sl5:c0"..., 79) = 79
556 read(3, "root:object_r:user_tty_device_t:"..., 4095) = 35
...
562 setxattr("/dev/console", "security.selinux", "root:object_r:user_tty_device_t:s0", 35, 0) = 0
```

由此可见，
1，首先通过 getxattr 函数得到/dev/console 文件的“security.selinux”扩展属性，其中保存该文件的 SC；
2，将登录用户的 SC，和登录设备的 SC，写入/selinux/relabel 文件；
3，相应 selinuxfs 驱动函数将查询内核 policy.X，返回相应 type_change 规则的结果，即将 console 重新打标为 user_tty_device_t；
4，最后通过 setxattr 函数，重置 console 的 SC 即可。

9.2 Socket Labeling 开发

在 SELinux 内核驱动中和创建 socket 相关的两个函数 selinux_socket_create 和 selinux_socket_post_create 中，都调用如下函数确定 socket 的 sid：

```
static u32 socket_sockcreate_sid(const struct task_security_struct *tsec)
{
        return tsec->sockcreate_sid ? : tsec->sid;
}
```

由此可见，如果此前用户态应用程序没有通过/proc/pid/attr/sockcreate 文件写入指定的 socket 上下文（则当前进程 tsec->sockcreate_sid == 0），则 socket 将默认地继承当前进程的 sid，所以 socket 继承其创建者的安全上下文。

9.2.1 提出问题：socket 默认继承其创建者的 SID 的副作用

对于 syslogd 后台进程而言，它的安全级别为 mls_systemhigh：

```
[root/sysadm_r/s0@~]# newrole -l sl5:c0.c1023 -- -c "ps Z -C syslogd"
Password:
LABEL                              PID TTY      STAT   TIME COMMAND
system_u:system_r:syslogd_t:sl5:c0.c1023 382 ? Ss     0:00 syslogd -m 0
[root/sysadm_r/s0@~]#
```

因此 syslogd 创建的 unix_dgram_socket 对象（用于 bond 到/dev/log，生成后者 socket file）的安全级别也是 mls_systemhigh。而其他运行在 mls_systemlow 级别中的进程，比如 klogd_t，则无法正常向该 unix_dgram_socket 发送数据，相关错误信息如下：

```
type=1400 audit(1298535101.654:868): avc:  denied  { sendto } for  pid=385 comm="klogd" path="/dev/log"
scontext=system_u:object_r:klogd_t:s0 tcontext=system_u:object_r:syslogd_t:sl5:c0.c1023
tclass=unix_dgram_socket
```

这是由于相应的 MLS 约束禁止了数据流从 mls_systemlow 到 mls_systemhigh 的流动：

```
mlsconstrain unix_dgram_socket sendto
        (( l1 eq l2 ) or
         (( t1 == mlsnetwriteranged ) and ( l1 dom l2 ) and ( l1 domby h2 )) or
         (( t1 == mlsnetwritetoclr ) and ( h1 dom l2 ) and ( l1 domby l2 )) or
         ( t1 == mlsnetwrite ) or
         ( t2 == mlstrustedobject ));
```

由于 klogd_t:s0 和 syslogd_t:s15:c0.c1023 无法满足 mlsnetwriteranged 或者 mlsnetwritetoclr 属性的要求，所以要么将发送者 klogd_t 加入 mlsnetwrite 属性，要么将目的 socket syslogd_t 加入 mlstrustedobject 属性。由于许多用户态应用程序都需要写入/dev/log，因此采用后者方案。

但是，这样做的一个副作用是，syslogd_t 同时还是 syslogd 的相应 procfs 目录子树的标签，显然为了 socket 的需要将 syslogd_t 加入任何 MLS 属性对于该 procfs 子树而言都不是必须的。

另外，一个进程可能创建不同 class 的 socket，按照当前的实现它们的 type 都将和创建者进程 domain 保持一致。如果能够实现 socket labeling，则同一进程创建的不同 class 的 socket 之间就可以有各自不同的 type。

9.2.2 分析问题

显然，避免上述副作用的惟一途径是支持给 socket 赋予和创建者不同的 type。设计方案如下：
1，在.te 中（策略），定义 socket 的 type，赋予创建者以及 socket 使用者对该 type 的相应能力，**指定创建者在创建 socket 时使用特定的 type**；
2，在 SELinux 内核驱动中（机制），当创建 socket 时**从策略中查询**当前进程新创建的 socket 的 type；

关键点为在策略中指定 socket 的 type，相应地在机制中向策略查询它。

在策略中可以使用 type_transition 规则指定任何对象的 type；在 SELinux 内核驱动中可以使用 security_transition_sid 函数（向策略查询该 type_transition 规则的结果）计算该对象的 sid。

因此，可以在 logging.te 中使用 type_transition 规则指定 syslogd_t 创建的 unix_dgram_socket 对象的标签，而不是默认继承 syslogd_t；并且在 socket_sockcreate_sid 函数中调用 security_transition_sid 函数重新计算 socket 的 sid，而非默认返回创建者的 sid。

其他注意事项：
1，并不是所有的 domain 都一定要求其 socket 具有和自己不一样的 type，因此当前实现的行为必须保留；
2，创建者的 role 必须能够和 socket 的 type 组成合法的安全上下文；
3，socket 的 type 可以和其创建者不同，但是角色以及 MLS 属性必须仍然和创建者保持一致。否则会给 labeled networking 以及 network access control 造成麻烦！

9.2.3 解决问题

1，在策略中给 syslogd 的 unix_dgram_socket 对象定义单独的 type，以及所需的能力：

```
type syslogd_s_t;
role system_r types syslogd_s_t;
mls_trusted_object(syslogd_s_t)
```

注意得益于 syslogd_s_t，我们可以给它调用 mls_trusted_object 接口，而不是针对 syslogd_t！

2，在策略中赋予 syslogd_t 以及其他使用者对于 syslogd_s_t 的相应能力：

```
-allow syslogd_t self:unix_dgram_socket create_socket_perms;
+allow syslogd_t syslogd_s_t:unix_dgram_socket create_socket_perms;

-allow syslogd_t self:unix_dgram_socket sendto;
+allow syslogd_t syslogd_s_t:unix_dgram_socket sendto;

-       allow $1 syslogd_t:unix_dgram_socket sendto;
+       allow $1 syslogd_s_t:unix_dgram_socket sendto;
```

3，在策略中使用 type_transition 规则明确定义 unix_dgram_socket 对象的标签：

```
+type_transition syslogd_t syslogd_t:unix_dgram_socket syslogd_s_t;
```

4，在 SELinux 内核驱动中修改 socket_sockcreate_sid 函数如下：

```
static int socket_sockcreate_sid(const struct task_security_struct *tsec, u16 secclass, u32 *socksid)
{
        if (tsec->sockcreate_sid > SECSID_NULL) {
                *socksid = tsec->sockcreate_sid;
                return 0;
        }

        return security_transition_sid(tsec->sid, tsec->sid, secclass, socksid);
}
```

注意，必须向调用者传递 security_transition_sid 函数的返回值！这样如果该函数返回错误码，则调用者可以以该错误码退出，从而及时发现在计算新对象的 SID 时发生的错误。比如曾经看到如下出错信息：

```
type=1401 audit(1298776242.225:10): security_compute_sid:  invalid context
system_u:system_r:syslogd_s_t:s15:c0.c1023 for scontext=system_u:system_r:syslogd_t:s15:c0.c1023
tcontext=system_u:system_r:syslogd_t:s15:c0.c1023 tclass=unix_dgram_socket
```

由此可见，当 SELinux 内核机制在执行相应的 type_transition 规则时，发现“system_u:system_r:syslogd_s_t:s15:c0.c1023”并不是一个有效的上下文，因此在策略中必须指定：

```
role system_r types syslogd_s_t;
```

5，在 SELinux 内核驱动中修改 security_compute_sid 函数，判断当前 class 是否为 socket class 之一。如果是则在拼接 newcontext 时使其默认的 role/type 和创建者保持一致，从而避免 socket class 对象的 role 为“object_r”，并在调用 mls_compute_sid 函数时传递布尔变量 sock：

```
        bool sock;

        if (kern) {
                tclass = unmap_class(orig_tclass);
                sock = security_is_socket_class(orig_tclass);
        } else {
                tclass = orig_tclass;
                sock = security_is_socket_class(map_class(tclass));
        }
```

```
 /* Set the role and type to default values. */
if ((tclass == policydb.process_class) || (sock == true)) {
        /* Use the current role and type of process. */
        newcontext.role = scontext->role;
        newcontext.type = scontext->type;
} else {
        /* Use the well-defined object role. */
        newcontext.role = OBJECT_R_VAL;
        /* Use the type of the related object. */
        newcontext.type = tcontext->type;
}
...
rc = mls_compute_sid(scontext, tcontext, tclass, specified, &newcontext, sock);
```

6，在 SELinux 内核驱动中修改 mls_compute_sid 函数：

```
case AVTAB_TRANSITION:
         ......
        /* Fallthrough */
case AVTAB_CHANGE:
        if ((tclass == policydb.process_class) || (sock == true))
                /* Use the process MLS attributes. */
                return mls_context_cpy(newcontext, scontext);
        else
                /* Use the process effective MLS attributes. */
                return mls_context_cpy_low(newcontext, scontext);
```

对于 type_transition/change 规则，如果对象为 process 类或者任何 socket 类，则为其保留创建者当前的 MLS attribute。而对于其他类的对象，比如 files 或者 chr_file，则为其只保留创建者的 low level。

7，在 SELinux 内核驱动中修改 genheaders.c 文件，自动创建 security_is_socket_class 函数，从而消除将来的维护开销 - 如果在 classmap.h 中新定义了一个 class，如果它是一个 socket 类，则能够自动加入 security_is_socket_class 函数中：

```
const char *needle = "SOCKET";
char *substr;

fprintf(fout, "\nstatic inline bool security_is_socket_class(u16 kern_tclass)\n");
fprintf(fout, "{\n");
fprintf(fout, "\tbool sock = false;\n\n");
fprintf(fout, "\tswitch (kern_tclass) {\n");
for (i = 0; secclass_map[i].name; i++) {
        struct security_class_mapping *map = &secclass_map[i];
        substr = strstr(map->name, needle);
        if (substr && strcmp(substr, needle) == 0)
                fprintf(fout, "\tcase SECCLASS_%s:\n", map->name);
}
fprintf(fout, "\t\tsock = true;\n");
fprintf(fout, "\t\tbreak;\n");
fprintf(fout, "\tdefault:\n");
fprintf(fout, "\t\tbreak;\n");
fprintf(fout, "\t}\n\n");
fprintf(fout, "\treturn sock;\n");
fprintf(fout, "}\n");
```

该函数遍历在 classmap.h 中定义的 secclass_map[]数组，如果一个 class 的名字字符串有效，则检查其

是否只以“SOCKET”字符串结尾并且只包含一个这样的字符串（strstr 函数返回子串的第一个出现位置。如果父串不以子串为结尾，则 strcmp 将返回多余部分的第一个字符的 ASCII 码）。如果是，则添加一个 case 分支。如果参数 kern_tclass 为任意表示 socket 类的 SECCLASS_XXX 宏之一，则返回 sock = true，否则默认返回 sock = false。

8，在 SELinux 内核驱动中增加 u16 map_class(u16)函数

```
/*
 * Get kernel value for class from its policy value
 */
static u16 map_class(u16 pol_value)
{
        u16 i;

        for (i = 1; i < current_mapping_size; i++) {
                if (current_mapping[i].value == pol_value)
                        return i;
        }

        return SECCLASS_NULL;                   # class not defined in kernel
}
```

注意 security_is_socket_class 函数将 class 的 kernel value 和 SECCLASS_XXX 宏相比较，所以如果 security_compute_sid 被从用户态调用（比如 compute_create 命令），则用户态传递的将是 class 的 policy value，即 checkpolicy 为该 class 分配的 index value，也就是相应 class_datum.value，所以在调用 security_is_socket_class 函数之前必须转换为相应的 kernel value。

这个工作在 map_class 函数中完成，若某个 selinux_mapping 元素的 value 和 class policy value 相同，则返回其数组索引即可（有关 class mapping 及其建立的分析参见下文）。注意，如果没有找到对应的 kernel value，则应该返回 SECCLASS_NULL（0），这种情况说明用户态使用了一个内核态尚未定义的 class。

附，之前这段代码有一个 bug：如果查找失败则返回 pol_value。则对于 x_drawable 类（只被用户态的 X 所使用。X 为用户态的 Object Manager，定义并管理和图形相关的许多类，比如窗口或者图标等，它们对于内核而言都是不可见的），其新建对象的 role 被设置为创建者的 role，而应该是“object_r”：

```
[root/sysadm_r/s0@QtCao ~]# compute_create `id -Z` `id -Z` x_drawable
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@QtCao ~]#
[root/sysadm_r/s0@QtCao ~]# cat /selinux/class/x_drawable/index
31
[root/sysadm_r/s0@QtCao ~]#

cao@cao-laptop:/work/linux-2.6/security/selinux$ grep -i x_drawable flask.h
cao@cao-laptop:/work/linux-2.6/security/selinux$
cao@cao-laptop:/work/linux-2.6/security/selinux$ grep 31 flask.h
#define SECCLASS_NETLINK_FIREWALL_SOCKET                31
cao@cao-laptop:/work/linux-2.6/security/selinux$
```

如上所示，x_drawable 这个类只定义在 refpolicy 中，而在 SELinux 内核驱动的 secclass_map[]中没有定义。如果 map_class 当查找失败时返回 pol_value，则返回其 policy value，即 31，恰好和 SECCLASS_NETLINK_FIREWALL_SOCKET 类的 kernel value 相等，导致 security_is_socket_class 函数错误地认为 x_drawable 是一个 socket 类型。

修正上述错误，在查找失败时返回 SECCLASS_NULL，则能够正确设置 x_drawable 类对象的 role：

```
[root/sysadm_r/s0@QtCao ~]# compute_create `id -Z` `id -Z` x_drawable
root:object_r:sysadm_t:s0
[root/sysadm_r/s0@QtCao ~]#
```

9.2.4 测试结果

在 socket_sockcreate_sid 函数中调用 security_transition_sid 函数，如果成功返回且 socksid 不等于 tsec->sid，则打印它们的数值并通过 dump_stack 函数打印函数调用链：

```
Pid: 395, comm: syslogd Not tainted 2.6.34.8-WR4.2.0.0_standard #13
Call Trace:
 [<c1247042>] socket_sockcreate_sid+0x52/0xc0
 [<c1247143>] selinux_socket_post_create+0x93/0x180
 [<c1240010>] security_socket_post_create+0x20/0x30
 [<c143bdb2>] __sock_create+0x252/0x3d0
 [<c1072234>] ? audit_syscall_exit+0x2d4/0x300
 [<c143bf97>] sock_create+0x37/0x50
 [<c143c1d6>] sys_socket+0x56/0xe0
 [<c143d38f>] sys_socketcall+0x8f/0x2f0
 [<c15254a1>] system_call_done+0x0/0x4
tsec->sid: ab, context: system_u:system_r:syslogd_t:s15:c0.c1023
socksid: af, context: system_u:system_r:syslogd_s_t:s15:c0.c1023
```

由此可见，syslogd 进程创建的某个 socket 和它具有不同的 sid，在安全上下文中 type 不同，但是 user，role，MLS attribute 都相同。

socket_sockcreate_sid > security_transition_sid 调用必须能够处理当前策略中没有和 socket 对象相关的 type_transition 规则的情况，此时必须保留原来的行为 - socket 继承创建者的 sid。另外，从用户态也能够正确地返回 socket 的安全上下文。

这些都可以通过 compute_create 命令来验证：

```
[root/sysadm_r/s0@~]# compute_create `id -Z` `id -Z` unix_stream_socket
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_create system_u:system_r:syslogd_t:s15:c0.c1023
system_u:system_r:syslogd_t:s15:c0.c1023 unix_stream_socket
system_u:system_r:syslogd_t:s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_create system_u:system_r:syslogd_t:s15:c0.c1023
system_u:system_r:syslogd_t:s15:c0.c1023 unix_dgram_socket
system_u:system_r:syslogd_s_t:s15:c0.c1023
[root/sysadm_r/s0@~]#
```

由此可见，如果不存在相应的 type_transition 规则，则 socket 的安全上下文和其创建者的相同，否则只有 type 不一样。

9.3 给 role_transition 规则添加 class 的支持

9.3.1 提出问题 - 当前 role_transition 规则只对 process 类有效

目前 libsepol/checkpolicy 以及 SELinux 内核机制中都只支持对 process 类使用 role_transition 规则，因此在 role_transition 规则的语法中甚至都不需要指定 class（而在 SELinux 内核机制中被默认地设置为 process 类）。将来为了实现对 RBAC 的完全支持，需要在 refpolicy 中指定不同类对象 newcontext 的默认设置方法，并允许使用 type/role/range_transition 规则重载默认实现，这就需要对 role_transition 规则添加 class 的支持（而 type/range_transition 规则都已实现）。

checkpolicy 中定义的 role_transition 规则尚未支持指定 class，其语法定义如下：

```
role_trans_def          : ROLE_TRANSITION names names identifier ';'
                        {if (define_role_trans()) return -1; }
                        ;
```

相应的在 security_compute_sid 函数中，只就 process 类应用(enforce) role_transition 规则：

```
        /* Check for class-specific changes. */
        if  (tclass == policydb.process_class) {              # only for the process class
                if (specified & AVTAB_TRANSITION) {           # type/role/range_transition 规则
                        /* Look for a role transition rule. */
                        for (roletr = policydb.role_tr; roletr; roletr = roletr->next) {
                                if (roletr->role == scontext->role && roletr->type == tcontext->type) {
                                        /* Use the role transition rule. */
                                        newcontext.role = roletr->new_role;
                                        break;
                                }
                        }
                }
        }
```

policydb_t 数据结构的 process_class 域记录 process 类的 kernel value；当前所有 role_transition 规则的展开描述组织在 policydb_t 的 role_tr 队列中。由上述代码可见 role_transition 规则只针对 process 类：如果当前进程的角色(scontext->role)以及可执行文件的 type(tcontext->type)和某个规则匹配，则当前进程在 exec 期间（domain transition 到 newcontext 中）其 role 将改变为该规则指定的新角色。

9.3.2 分析问题

为了给 role_transition 规则添加 class 的支持，直观上不难确定需要如下工作：
1，给 libsepol 的 role_transition 规则描述符增加关于 class 的域；
2，修改 checkpolicy 关于 role_transition 规则的词法分析函数，能够识别并保存 class；
3，修改 libsepol 创建 policy.X 和 pp 文件的函数，在二进制文件中增加 class 的二进制表示；
4，相应地，在 role_transition 规则的 SELinux 内核描述符中增加 class 域，并且在 policy.X 的解析函数中能够读取并保存 class 的二进制表示；
5，在 security_compute_sid 内核函数中计算所有类的新对象的 newcontext 时，检查是否有匹配的 role_transition 规则，如果有则重载 newcontext.role 的默认实现。

进一步思考，发现还需要考虑如下方面：
1，role_transition 规则在用户态需要两种描述数据结构：在针对 policy.X 的“展开”描述中使用 uint32_t 来描述规则中的 source_type，target_type 等域；而在针对模块的“字面”描述中主要借助 ebitmap 位图来描述 attribute-type 关系和规则中相应域的扩展。

相应地，在规则的“展开”描述和“字面”描述的数据结构中都应该添加描述 class 的域：前者使用 uint32_t，而后者使用 ebitmap 位图。

2，给 role_transition 规则增加了 class 的支持后，需要增加用户态所支持的 policy.X（kernel policy)以及模块（base 模块，非 base 模块）的最大版本号，并在描述兼容性的数组中注册。

3，尽管增加了 policy.X 和模块的最大版本号，但仍能够编译为较低版本的格式，即发生 policy downgrade，那么在 policy.X 中应该将 role_transition 规则的二进制表示恢复到没有添加对 class 支持前的状态，并且“过滤掉”refpolicy 规则库中所有使用了新特性的 role_transition 规则，而只保留那些使用原有语法的规则。

4，在用户态和内核态中都有一些工具或设施用于调试目的，需要修改相应的调试工具/设施，能够访问（读取或写出）role_transition 规则的 class 域。

9.3.3 解决问题

这里我们根据模块的编译，link，expand 顺序来介绍在各个环节上需要增加的对 class 的支持。

1，给 role_transition 规则的“展开”描述和“字面”描述中增加描述 class 的数据结构：

```
 typedef struct role_trans {
        uint32_t role;          /* current role */
        uint32_t type;          /* program executable type, or default new object type (aka, parent folder
type) */
+       uint32_t tclass;        /* process class, or new object class */
        uint32_t new_role;      /* new role */
        struct role_trans *next;
 } role_trans_t;

 typedef struct role_trans_rule {
        role_set_t roles;       /* current role */
        type_set_t types;       /* program executable type, or default new object type */
+       ebitmap_t classes;      /* process class, or new object class */
        uint32_t new_role;      /* new role */
        struct role_trans_rule *next;
 } role_trans_rule_t;
```

role_trans_rule_t 数据结构为 role_transition 规则的“字面”描述，使用 ebitmap 位图来描述规则中各个域上可能存在的扩展，因此也需要使用 ebitmap 来描述规则中可能指定的一组 class 的集合。编译 role_transition 规则时创建 role_trans_rule_t，并加入当前模块当前 block/decl（由 stack_top->decl 指向）的 avrule_decl_t.role_tr_rules 队列。

注意，由于规则中 class 位置上不允许使用特殊字符，因此不需要使用 type_set_t 或者 role_set_t 数据结构中的 flags 标志，而直接使用 ebitmap 位图描述 class 即可（type_set_t 和 role_set_t 数据结构就是在 ebitmap 的基础上通过 flags 来记录特殊字符，比如通配符，取反，或者减号操作）。

而在模块的 link/expand 过程中，根据 role_trans_rule_t 中各种位图非 0 位置的所有有效组合，逐一创建 role_trans_t 数据结构，其中每个域都描述相应标识符写入 policy.X 的 policy value，因此只需用 uint32_t 来描述一个 class 即可。

2，更新 policy.X 和 base 模块、非 base 模块所支持的最大版本号，并注册到描述兼容性的 policydb_compat[]数组中：

```
+#define POLICYDB_VERSION_ROLETRANS             26
+#define POLICYDB_VERSION_MAX   POLICYDB_VERSION_ROLETRANS

+#define MOD_POLICYDB_VERSION_ROLETRANS         12
+#define MOD_POLICYDB_VERSION_MAX MOD_POLICYDB_VERSION_ROLETRANS

        {
+        .type = POLICY_KERN,                                    # kernel policy, policy.X
+        .version = POLICYDB_VERSION_ROLETRANS,
+        .sym_num = SYM_NUM,
+        .ocon_num = OCON_NODE6 + 1,
+        .target_platform = SEPOL_TARGET_SELINUX,
+       },

         {
+        .type = POLICY_BASE,                                    # base.pp
+        .version = MOD_POLICYDB_VERSION_ROLETRANS,
+        .sym_num = SYM_NUM,
+        .ocon_num = OCON_NODE6 + 1,
+        .target_platform = SEPOL_TARGET_SELINUX,
+       },

+       {
+        .type = POLICY_MOD,                                     # non-base modules
+        .version = MOD_POLICYDB_VERSION_ROLETRANS,
+        .sym_num = SYM_NUM,
+        .ocon_num = 0,
+        .target_platform = SEPOL_TARGET_SELINUX,
+       },
```

3，修改 role_transition 规则语法和词法分析函数 define_role_trans：
1）允许在 role_transition 规则的语法中指明 class，并使用参数 class_specified 表示是否指定了 class；
2）如果 class_specified == 1，则在词法分析时在循环中读取可能指定的 class：就每一个描述 class 的 token，检查其 scope 是否在当前模块中定义（这里一定满足，因为 policy_module 宏声明 all_kernel_call_perms 为外部依赖），并查询当前模块的 p_classes 符号表得到其 class_datum_t 数据结构，然后以其(policy value - 1)为索引，设置到临时位图 e_classes 中（注意，role_transition 规则中使用的任何标识符都必须在当前模块内定义或者声明过，declare_symbol 和 require_symbol 函数都会为定义或声明的标识符创建相应的 xxx_datum_t 和 scope_datum_t 数据结构并注册到相应的符号表）；
3）否则，在 p_classes 符号表中查找“process”类的 class_datum_t，以其(policy value - 1)为索引设置到 e_classes 中；
4）将 e_classes 位图复制到 role_trans_rule_t.classes 中；
至此，就可以把该 role_trans_rule_t 添加到当前 block/decl 的 role_tr_rules 队列中了。

另外，在第 4 步之前还设计了一个额外的操作：将当前 role_transition 规则的“字面”描述展开，并和当前模块 policydb_t.role_tr 队列中所有的“展开”描述相比较，以检查在当前模块源代码中是否出现冲突或者重复定义。如果有则报错退出，否则将“展开”描述加入 role_tr 队列（以便进行后继比较）。

注意，对于模块而言真正需要的是 block/decl 的 role_tr_rules 队列中的“字面”描述，而上述 policydb_t.role_tr 队列中的“展开”描述仅仅是为了检查当前模块源代码中是否存在冲突或重复的定义。

因此在展开时需要再增加一层循环得到 e_classes 中所有非 0 位索引，并且比较 class。具体实现参见下文 define_role_trans 函数源代码分析。

4，至此 role_transition 规则中的 class 就可以被正确编译了，下面需要修改 role_trans_rule_write 函数在创建模块的 pp 文件时正确地创建 class 的二进制表示。

直观上，只需要通过 ebitmap_write 函数将 role_trans_rule_t.classes 位图写入 pp 文件即可。但是为了支持 policy downgrade 还必须考虑更多的情况。role_transition 规则在是否使用 class 域上可能存在如下三种可能：
1）没有指定 class；
2）指定了惟一的“process” class；
3）指定了 class，不是或者不全是“process” class；

注意，由于在 role_trans_rule_t 数据结构和 define_role_trans 函数中无条件地增加了对 class 的支持，因此编译后 role_trans_rule_t.classes 位图总是存在。对于上述第 1 和 2 种情况而言，classes 位图中的非 0 位只有 1 位且以“process”类的(policy value - 1)为索引；而第 3 种情况中位图中可能存在多个非 0 位置，且不一定以“process”类的(policy value - 1)为索引。

如果在 semanage.conf 文件中没有设定 policy-version 参数，在 checkmnodule.c 代码中默认情况下 pp 的版本号为当前 libsepol 所支持的最大模块版本号 MOD_POLICYDB_VERSION_MAX，它已经被设置为 MOD_POLICYDB_VERSION_ROLETRANS。此时可以无条件地将所有 role_transition 规则都写入 pp 文件，无论 class 域为上述何种情况。

但是，如果指定 policy-version < MOD_POLICYDB_VERSION_MAX，则应该只保留满足上述 1 和 2 两种情况的 role_transition 规则，而忽略第 3 种情况的规则，且不改变原有 role_transition 规则二进制表示的创建方法（即不写入 class 域）。

所以在 role_trans_rule_write 函数中，用 new_roletr 变量来表示是否发生 policy downgrade。如果是，则在计算当前 block/decl 的 role_tr_rules 队列总数时跳过上述第 3 种情况：

```
        int new_roletr = p->policyvers >= MOD_POLICYDB_VERSION_ROLETRANS;

        for (tr = t; tr; tr = tr->next)
                if (new_roletr || only_process(&tr->classes))
                        nel++;
```

其中 only_process 函数用于检查它中是否包含了除“process”之外的其他类：

```
static int only_process(ebitmap_t *in)
{
        unsigned int i;
        ebitmap_node_t *node;

        ebitmap_for_each_bit(in, node, i) {
                if (ebitmap_node_get_bit(node, i) && i != SECCLASS_PROCESS - 1)
                        return 0;
        }

        return 1;
}
```

如上所述如果发生 policy downgrade，则 new_roletr 标志为 0，此时不计算使用了非 process 类的 role_transition 规则，因为它们不被写入 pp 和 policy.X，而且 role_transition 的二进制表示中不包含 class 的二进制表示：

```
        for (tr = t; tr; tr = tr->next) {
+               if (!new_roletr && !only_process(&tr->classes)) {
+                       if (!warned)
+                               WARN(fp->handle, "Discarding role_transition "
+                                       "rules for security classes other than \"process\"");
+                       warned = 1;
+                       continue;
+               }
                if (role_set_write(&tr->roles, fp))
                        return POLICYDB_ERROR;
                if (type_set_write(&tr->types, fp))
                        return POLICYDB_ERROR;
+               if (new_role)
+                       if (ebitmap_write(&tr->classes, fp))
+                               return POLICYDB_ERROR;
                buf[0] = cpu_to_le32(tr->new_role);
                items = put_entry(buf, sizeof(uint32_t), 1, fp);
                if (items != 1)
                        return POLICYDB_ERROR;
        }
```

5，至此 role_transition 规则能够被正常编译并写入 pp 文件了。在模块的 link/expand 过程开始前，需要读取 pp 文件中规则的二进制表示并保存到规则的“字面”描述中，对 role_transition 规则而言即为 role_trans_rule_t 数据结构。因此在 role_trans_rule_read 函数里需要调用 ebitmap_read 函数从 pp 文件中读取 role_trans_rule_t.classes 域。

和 role_trans_rule_write 函数相对应，如果发生 policy downgrade 则 pp 中将不含有 classes 的二进制表示，此时直接以(SECCLASS_PROCESS - 1)为索引，设置 classes 域即可。

```
                if (p->policyvers >= MOD_POLICYDB_VERSION_ROLETRANS) {
                        if (ebitmap_read(&tr->classes, fp))
                                return -1;
                } else {
                        if (ebitmap_set_bit(&tr->classes, SECCLASS_PROCESS - 1, 1))
                                return -1;
                }
```

6，在 link 过程中，需要把当前模块的所有 block/decl 都拷贝到 base 模块，当前 block/decl 中的所有规则描述数据结构都会被逐一拷贝。在 copy_role_trans_list 函数中拷贝当前 block/decl 的 avrule_decl_t.role_tr_rules 队列中的每一个数据结构时，复制其 classes 位图到新创建的 role_trans_rule_t 数据结构（由 new_rule 指向）的 classes 位图中：

```
+               ebitmap_for_each_bit(&cur->classes, cnode, i) {
+                       if (ebitmap_node_get_bit(cnode, i)) {
+                               assert(module->map[SYM_CLASSES][i]);
+                               if (ebitmap_set_bit(&new_rule->classes,
+                                                   module->map[SYM_CLASSES][i] - 1, 1)) {
+                                       goto cleanup;
+                               }
+                       }
+               }
```

注意，link 过程中拷贝规则前已经调用 xxx_copy_callback 函数将当前模块的所有符号表合并到了 base 模块的相应符号表，并且给每个标识符都重新基于 base.p_xxx 符号表的 nprim 重新分配了 policy

value，新旧 policy value 的转换关系保存在当前模块的 policy_module_t.map[SYM_XXX][]数组中。因此需要查询 map[SYM_CLASSES][]数组得到 classes 位图中所有非 0 位的新 policy value，再设置到新位图中。

注意，ebitmap 和 map[][]数组都是使用标识符的(policy value - 1)作为索引，因此以 ebitmap 中非 0 位的位置直接索引 map[][]即可得到该标识符新的 policy value，以其值减 1 再设置到新位图中。

7，在 expand 过程中，需要展开 base 模块所有 block/decl 中所有和规则相关的数据结构，将规则的“字面”描述扩展为“展开”描述，并组织到 out 模块相应数据结构中去。在 copy_role_trans 函数中展开 role_trans_rule_t 数据结构时，需要再增加一层循环以处理 classes 位图，并在和 out 模块 policydb_t.role_tr 队列中的元素逐一比较时考察 role_trans_t.tclass 域。

详见后文 copy_role_trans 函数分析。

8，至此，已经处理了 role_trans_rule_t.classes 的 read，write，link，expand 环节还需要修改 role_trans_rule_init 和 role_trans_role_write 函数，以正确地初始化和销毁 classes 位图。

9，如果指定 MONOLITHIC = y，即以 kernel policy 方式编译直接创建 policy.X，那么和第 4 条对应，在 role_trans_write 函数中将根据是否发生 policy downgrade 决定是否创建 role_trans_t.tclass 域的二进制表示：

```
+       int new_roletr = (p->policy_type == POLICY_KERN && p->policyvers >= POLICYDB_VERSION_ROLETRANS);
+       int warning_issued = 0;

        nel = 0;
        for (tr = r; tr; tr = tr->next)
+               if(new_roletr || tr->tclass == SECCLASS_PROCESS)
+                       nel++;

        buf[0] = cpu_to_le32(nel);
        items = put_entry(buf, sizeof(uint32_t), 1, fp);
        if (items != 1)
                return POLICYDB_ERROR;
        for (tr = r; tr; tr = tr->next) {
+               if (!new_roletr && tr->tclass != SECCLASS_PROCESS) {
+                       if (!warning_issued)
+                               WARN(fp->handle, "Discarding role_transition "
+                                    "rules for security classes other than \"process\"");
+                       warning_issued = 1;
+                       continue;
+               }
                buf[0] = cpu_to_le32(tr->role);
                buf[1] = cpu_to_le32(tr->type);
                buf[2] = cpu_to_le32(tr->new_role);
                items = put_entry(buf, sizeof(uint32_t), 3, fp);
                if (items != 3)
                        return POLICYDB_ERROR;
+               if (new_roletr) {
+                       buf[0] = cpu_to_le32(tr->tclass);
+                       items = put_entry(buf, sizeof(uint32_t), 1, fp);
+                       if (items != 1)
+                               return POLICYDB_ERROR;
+               }
        }
```

使用 new_roletr 表示是否发生 policy downgrade。如果没有指定 OUTPUT_POLICY，或者它的值不小于 POLICYDB_VERSION_ROLETRANS，则不发生 policy downgrade，则此时所有 role_transition 规则都会被写入 policy.X 文件，且带有 tclass 域的二进制表示。

否则，只把 tclass == SECCLASS_PROCESS 的那些 role_trans_t 数据结构写入 policy.X 文件，且不包含 tclass 域的二进制表示。

10，如果指定 MONOLITHIC = y，即以 kernel policy 方式编译直接创建 policy.X，那么和第 5 条对应，在 role_trans_read 函数中根据是否发生 policy downgrade 得知 policy.X 中是否带有 role_trans_t.tclass 域的二进制表示

```
+       int new_roletr = (p->policy_type == POLICY_KERN && p->policyvers >= POLICYDB_VERSION_ROLETRANS);

+               if (new_roletr) {
+                       rc = next_entry(buf, fp, sizeof(uint32_t));
+                       if (rc < 0)
+                               return -1;
+                       tr->tclass = le32_to_cpu(buf[0]);
+               } else
+                       tr->tclass = SECCLASS_PROCESS;
```

如果当前 policy.X 的版本号不小于 POLICYDB_VERSION_ROLETRANS，则说明带有 tclass 域的二进制表示，读取之，否则设置为默认的 SECCLASS_PROCESS（即在旧版本中 role_transition 规则的语义上不支持指定 class，policy.X 中也没有二进制表示。与此相应，SELinux 内核中只针对“process”类应用 role_transition 规则）。

11，最后，checkpolicy/test/dismod.c 文件用于调试 pp，它可以打印模块中所有规则。那么在“display role transitions”功能中应该打印 role_trans_rule_t.classes 位图：

```
 void display_role_trans(role_trans_rule_t * tr, policydb_t * p, FILE * fp)
 {
        for (; tr; tr = tr->next) {
                fprintf(fp, "role transition ");
                display_mod_role_set(&tr->roles, p, fp);
                display_type_set(&tr->types, 0, p, fp);
+               fprintf(fp, " :");
+               display_class_set(&tr->classes, p, fp);
+               display_id(p, fp, SYM_ROLES, tr->new_role - 1, "");
                fprintf(fp, "\n");
        }
 }

+void display_class_set(ebitmap_t *classes, policydb_t *p, FILE *fp)
+{
+       int i, num = 0;
+
+       for (i = ebitmap_startbit(classes); i < ebitmap_length(classes); i++) {
+               if (!ebitmap_get_bit(classes, i))
+                       continue;
+               num++;
+               if (num > 1) {
+                       fprintf(fp, "{");
+                       break;
+               }
+       }
+
```

```
+       for (i = ebitmap_startbit(classes); i < ebitmap_length(classes); i++) {
+               if (ebitmap_get_bit(classes, i))
+                       display_id(p, fp, SYM_CLASSES, i, "");
+       }
+
+       if (num > 1)
+               fprintf(fp, " }");
+}
+
```

该函数很简单，首先计算 classes 位图中非 0 位的个数。如果多余 1 个，则打印扩展符号“{ }”，然后逐个打印非 0 位的标识符名称，display_id 即访问当前模块的 sym_val_to_name[SYM_CLASSES][i]，直接得到当前 class 标识符的名称字符串。

9.3.4 测试结果

1，在 sysadm.te 末尾增加如下规则：

```
role_transition sysadm_r user_home_t:{ file dir } sysadm_r;
role sysadm_r types user_home_t;
```

即当 sysadm_r 在 user_home_t 目录下创建 file 或 dir 类对象时，新对象 SC 中 role 变成 sysadm_r（和创建者保持一致，而不再是默认的 object_r）。注意“role xxx types xxx”规则是为了保证新对象的 SC 是合法的。

```
gen_require(`
        type vlock_exec_t, vlock_t;
')
role_transition sysadm_r vlock_exec_t system_r;
```

即当 sysadm_r 在执行 vlock_exec_t 程序时，当前进程 SC 的 role 为 system_r（而不是默认地保持进程的原有 role）。注意此规则中没有指定 class，此时词法分析程序应该设置为默认值（即 process 类）。

2，以 MONOLITHIC = n 方式编译，得到 sysadm.pp 后使用 checkpolicy/test/dismod 命令打印 pp 中的 role_transition 规则：

```
/work/selinux/selinux/checkpolicy$ test/dismod /work/selinux/refpolicy/sysadm.pp
Reading policy...
......
Command ('m' for menu):  7
role transitions:
--- begin avrule block ---
decl 1:
role transition  sysadm_r [vlock_exec_t] : [process] system_r
role transition  sysadm_r [user_home_t] :{ [file] [dir] } sysadm_r
--- begin avrule block ---
decl 2:
......
--- begin avrule block ---
decl 342:

Command ('m' for menu):  q
/work/selinux/selinux/checkpolicy$
```

由此可见，词法分析程序能够正确地处理 role_transition 规则中的 class 域，无论是否指定，是否存在扩展。

3，进一步，检查最终生成的 policy.X 中 role_transition 规则的 class 的二进制表示是否正确，可以使用 xxd 命令打印 policy.X 的二进制表示：

```
/work/selinux/refpolicy$ ls -l /etc/selinux/refpolicy-mls/policy/
total 5728
-rw-r--r--. 1 root root 5849662 2011-03-25 13:12 policy.26
/work/selinux/refpolicy$ xxd /etc/selinux/refpolicy-mls/policy/policy.26 > policy_26_xxd
/work/selinux/refpolicy$ vim policy_26_xxd
......
055c510:                                     0800  1.....S.........
055c520: 0000 0300 0000 a006 0000 0b00 0000 0200  ................
055c530: 0000 0300 0000 a103 0000 0b00 0000 0200  ................
055c540: 0000 0800 0000 b707 0000 0b00 0000 0200  ................
055c550: 0000 0800 0000 a70a 0000 0b00 0000 0200  ................
055c560: 0000 0a00 0000 db00 0000 0b00 0000 0200  ................
055c570: 0000 0a00 0000 8e05 0000 0a00 0000 0600  ................
055c580: 0000 0a00 0000 8e05 0000 0a00 0000 0700  ................
055c590: 0000 0c00 0000 9209 0000 0b00 0000 0200  ................
055c5a0: 0000
......
/work/selinux/refpolicy$
```

说明：
1）在 class 的二进制表示在 new_role 的后面（这样做是为了简化 role_trans_write 函数的代码，参见上文）；
2）各种标识符的 policy value 如下（分析 policy.X 中相应标识符的二进制存储格式得到）：

sysadm_r == 0a，user_home_t == 58e，file == 06，dir = 07，vlock_exec_t == db

由此可见，policy.X 中新增 role_transition 规则的 class 的二进制表示正确。

4，运行时测试一：验证涉及非“process”类的 role_transition 规则能够正确地执行：

```
[root/sysadm_r/s0@~]# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 26
Policy from config file:        refpolicy-mls
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# id -Z
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]# ls -Zd
dr-xr-x---  root root root:object_r:user_home_dir_t:s0-s15:c0.c1023 .
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# mkdir parent
[root/sysadm_r/s0@~]# ls -Z
drwxr-xr-x  root root root:object_r:user_home_t:s0     parent
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_create root:sysadm_r:sysadm_t:s0-s15:c0.c1023 root:object_r:user_home_t:s0
dir
root:sysadm_r:user_home_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_create root:sysadm_r:sysadm_t:s0-s15:c0.c1023 root:object_r:user_home_t:s0
file
root:sysadm_r:user_home_t:s0
```

```
   [root/sysadm_r/s0@~]#
   [root/sysadm_r/s0@~]# compute_create root:sysadm_r:sysadm_t:s0-s15:c0.c1023 root:object_r:user_home_t:s0
lnk_file
   root:object_r:user_home_t:s0
   [root/sysadm_r/s0@~]#
   [root/sysadm_r/s0@~]# cd parent
   [root/sysadm_r/s0@parent]# mkdir dir
   [root/sysadm_r/s0@parent]# touch file
   [root/sysadm_r/s0@parent]# ln -s file file_lk
   [root/sysadm_r/s0@parent]# mkfifo pipe
   [root/sysadm_r/s0@parent]#
   [root/sysadm_r/s0@parent]# ls -Z
   drwxr-xr-x  root root root:sysadm_r:user_home_t:s0     dir
   -rw-r--r--  root root root:sysadm_r:user_home_t:s0     file
   lrwxrwxrwx  root root root:object_r:user_home_t:s0     file_lk -> file
   prw-r--r--  root root root:object_r:user_home_t:s0     pipe
   [root/sysadm_r/s0@parent]#
```

说明：
1）首先运行 sestatus 命令检查当前内核所装载的 policy.X 的版本号，以确认其增加了对 class 的支持；
2）由于 role_transition 规则中 target_type 为 user_home_t，而用户 HOME 目录为 user_home_dir_t，因此在 HOME 下创建一个子目录以满足规则的要求；
3）进而使用 compute_create 命令（通过/selinux/context 接口）访问内核中的 SELinux Security Server，查询当前进程在子目录下创建的新对象的上下文。由此可见 file/dir 类对象的 role 不再默认继承 object_r，而由匹配的 role_transition 规则决定；
4）最后在子目录下创建若干类型的文件，进一步验证 file/dir 类新对象的 role 为 sysadm_r。

5，运行时测试二：验证涉及“process”类的 role_transition 规则运行正确：

```
   [root/sysadm_r/s0@~]# ls -Z /usr/sbin/vlock-main
   -rws--x--x  root root system_u:object_r:vlock_exec_t:s0 /usr/sbin/vlock-main
   [root/sysadm_r/s0@~]#
   [root/sysadm_r/s0@~]# seclow "setenforce 0"
   Password:
   [root/sysadm_r/s0@~]# compute_create root:sysadm_r:sysadm_t:s0-s15:c0.c1023
system_u:object_r:vlock_exec_t:s0 process
   root:system_r:vlock_t:s0-s15:c0.c1023
   [root/sysadm_r/s0@~]#

   [root/staff_r/s0@~]# vlock &
   [1] 743
   [root/staff_r/s0@~]# ps Z -C vlock-main
   LABEL                              PID TTY      STAT   TIME COMMAND
   root:staff_r:vlock_t:s0-s15:c0.c1023 743 pts/0 T      0:00 /usr/sbin/vlock-main
   [1]+  Stopped                 vlock
   [root/staff_r/s0@~]#

   [root/sysadm_r/s0@~]# vlock &
   [1] 759
   [root/sysadm_r/s0@~]# ps Z -C vlock-main
   LABEL                              PID TTY      STAT   TIME COMMAND
   root:staff_r:vlock_t:s0-s15:c0.c1023 743 pts/0 T      0:00 /usr/sbin/vlock-main
   root:system_r:vlock_t:s0-s15:c0.c1023 759 ttyS0 T     0:00 /usr/sbin/vlock-main
   [1]+  Stopped                 vlock
   [root/sysadm_r/s0@~]#
```

说明：

1）首先检查 vlock-main 程序的标签，并用 compute_create 命令返回当 sysadm_r 运行 vlock 程序（为 vlock-main 程序的封装脚本）期间的暗渠阿上下文。由此可见在运行 vlock 程序期间当前进程的 role 由 sysadm_r 变成 system_r；
2）staff_r 和 sysadm_r 角色分别以后台方式运行 vlock；通过“ps -Z C”命令检查运行 vlock-main 程序的进程的安全上下文。由此可见 sysadm_r 角色运行它时当前进程的 role 的确变成 system_r，而 staff_r 角色在运行它时当前进程的 role 仍然保持为 staff_r；
3）在 sysadm_r 角色运行 vlock 命令之前首先将 SELinux 切换到 Permissive 模式，使得非法的 root-system_r，system_r-vlock_t 组合并不影响程序的运行（我们的目的是为了演示对 process 类对象应用 role_transition 规则。由此可见，用户态 refpolicy 中在使用 role_transition 规则时必须考虑和新角色相关的 user-role，role-type 组合都是合法的）。

6，验证 SELinux 内核驱动在根据已装载的 policy.X 的描述数据结构 policydb_t，重新创建 policy.X 时能够正确地创建 role_transition 规则中 class 域的二进制表示：

```
[root/sysadm_r/s0@~]# setenforce 1
[root/sysadm_r/s0@~]# sestatus
SELinux status:                 enabled
SELinuxfs mount:                /selinux
Current mode:                   enforcing
Mode from config file:          enforcing
Policy version:                 26
Policy from config file:        refpolicy-mls
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# cat /selinux/policy > policy_read
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ls -l policy_read
-rw-r--r-- 1 root root 5849662 Mar 25 05:26 policy_read
[root/sysadm_r/s0@~]# ls -l /etc/selinux/refpolicy-mls/policy/
total 5732
-rw-r--r-- 1 root root 5849662 Mar 25 05:15 policy.26
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# xxd policy_read > policy_read_xxd
[root/sysadm_r/s0@~]# vim policy_read_xxd
......
055c510:                                      0800  1.....S.........
055c520: 0000 0300 0000 a006 0000 0b00 0000 0200  ................
055c530: 0000 0300 0000 a103 0000 0b00 0000 0200  ................
055c540: 0000 0800 0000 b707 0000 0b00 0000 0200  ................
055c550: 0000 0800 0000 a70a 0000 0b00 0000 0200  ................
055c560: 0000 0a00 0000 db00 0000 0b00 0000 0200  ................
055c570: 0000 0a00 0000 8e05 0000 0a00 0000 0600  ................
055c580: 0000 0a00 0000 8e05 0000 0a00 0000 0700  ................
055c590: 0000 0c00 0000 9209 0000 0b00 0000 0200  ................
055c5a0: 0000
......
[root/sysadm_r/s0@~]#
```

说明：
1）通过/selinux/policy 文件，可以读出当前已装载到内核的 policy.X 的二进制表示，触发 policydb_write > role_trans_write 函数调用链；
2）首先检查/selinux/policy 输出的 policy.X 和被装载的 policy.X 的大小相同；
3）然后使用 xxd 打印 role_transition 规则的二进制表示。和第 3 步的结果相比较，完全相同。

9.3.5 其他说明

restorecon/setfiles 命令可以根据 file_contexts 文件或 policy store 中保存的“semanage fcontext -a”结果，将整个文件系统的标签恢复到初始状态，但是它们无法知道或使用任何运行时的 transition 规则，无论 type/role/range transition，因此在上述 role_transition 规则生效后，使用 restorecon 命令将恢复文件的 role 为“object_r”：

```
[root/sysadm_r/s0@~]# cd parent/
[root/sysadm_r/s0@parent]#
[root/sysadm_r/s0@parent]# ls -Z
drwxr-xr-x  root root root:sysadm_r:user_home_t:s0     dir
-rw-r--r--  root root root:sysadm_r:user_home_t:s0     file
lrwxrwxrwx  root root root:object_r:user_home_t:s0     file_lk -> file
prw-r--r--  root root root:object_r:user_home_t:s0     pipe
[root/sysadm_r/s0@parent]#
[root/sysadm_r/s0@parent]# restorecon . -R
[root/sysadm_r/s0@parent]#
[root/sysadm_r/s0@parent]# ls -Z
drwxr-xr-x  root root root:object_r:user_home_t:s0     dir
-rw-r--r--  root root root:object_r:user_home_t:s0     file
lrwxrwxrwx  root root root:object_r:user_home_t:s0     file_lk -> file
prw-r--r--  root root root:object_r:user_home_t:s0     pipe
[root/sysadm_r/s0@parent]#
```

由此可见，在系统运行了一段时间后不宜使用 restorecon 大面积地修改文件系统的标签，可以用它修复已知的错误标签，从而避免之前应用过的 user/role/type transition 规则失效。

9.3.6 经验总结

对“policy downgrade”有了正确的认识，它有两方面的影响：
1，影响对 refpolicy 规则库中规则的取舍；
2，影响相应规则的二进制表示格式，进而影响 policy.X 或者 pp 的读、写方法；

比如如果选择较低版本，则在创建 pp 时将忽略所有使用了高版本特性的规则，且就保留下来的符合较低版本特性的规则，其二进制表示中也不含有高版本特性。

事后诸葛亮：
当自己最初的 5 个 commit 被社区接受后，被发现存在如下主要的错误：
1，就 policy.X 没有考虑 policy downgrade，在 role_trans_write 函数中存在一个 bug：当计算 role_tr 队列的元素个数时，对于 downgraded policy，应该跳过所有 tr->tclass != SECCLASS_PROCESS 的元素，即应该舍弃所有使用了新特性的规则；

2，没有把应用在 policy.X 相关函数上的处理逻辑，应用到 modules 的相关函数上。
由于默认情况下 refpolicy 以模块方式编译（MONOLITHIC = n），所以在 pp 文件的创建，读取，link，expand 过程中都应该提供对 role_transition 规则的 class 支持，且考虑 policy downgrade。

Checklist：改变用户态或内核态所支持的最大版本号时所必须的测试
1．不指定 build.conf 文件中的 OUTPUT_POLICY 变量，MONOLITHIC = y，编译默认版本的 policy.X；
2．不指定 semanage.conf 文件中的 policy-version 变量，MONOLITHIC = n，以模块方式编译 policy.X；
3．检查上述两种方法得到的 policy.X，相应语法成分的二进制表示；
4．OUTPUT_POLICY = MAX-1，MONOLITHIC = y，检查 policy downgrade；
5．policy-version = MAX-1，MONOLITHIC = n，检查 policy downgrade；
6．装载默认版本的 policy.X；

7. 装载 downgraded 版本的 policy.X;

另外，如果希望得到 policy module downgraded，则可以直接修改 checkmodule.c 文件，在 write_binary_policy 函数中直接设置 policyvers 全局变量，比如：

```
        policyvers = MOD_POLICYDB_VERSION_MAX - 1;
```

从而使得 pp 的版本号并不是最大版本号，从而检查相应工具处理不同版本 pp 的能力。

注意：
1，当用户态 libsepol 所支持的 policy.X 的最大版本号增加后，SELinux 内核中所支持的最大版本号也必须增加。否则在 load policy.X 时将发生 policy downgrade;
2，由于 pp 只被用户态所需要，所以如果只需要改变 pp 的最大版本号而 policy.X 的格式不需要改变，则无须修改内核驱动。比如添加 role attribute 的支持就是这种情况。

另外，用户态也维护了 Security Server（可以通过 checkpolicy -d 来访问），所以理论上在改变内核 SS 时也应该同时修改用户态 SS。而实际上由于它和内核中的 SS 被分开维护，用户态 SS 的代码比较陈旧。况且目前用户态的 Object Manager（比如 Xorg，dbus）都使用内核态的 Security Server，所以并不硬性地要求必须修改用户态 SS，这也进一步导致用户态 SS 更加过时。

9.4 增加 role attribute 的支持（new）

9.4.1 提出问题之一：role-dominance 规则的局限性

RBAC 机制的核心思想就是通过限制可以和一个 role 相匹配的 type 的集合，来定义该 role 的“能力”。即使相应的 TE 规则存在，如果没有许可当前 role 和新 domain 相组合，Domain Transition 也会失败（因为新的 domain 无法和原有 role 组成合法的 SC）。

能够和一个 role 相结合的 type 的集合，由 role_datum_t.types type_set_t 数据结构描述。由第 12 章可见，在 role-types 规则的词法分析函数中设置 role_datum_t.types 集合，而在模块的 link 阶段将一个 role 能够 type（动词）的所有 type 相合并。

目前 libsepol 支持如下 role dominance 语法：

```
gen_require('
        role sysadm_r, secadm_r, auditadm_r;
')

dominance { role rootuser_r {
        role sysadm_r;
        role secadm_r;
        role auditadm_r;
} }
```

即 rootuser_r 为 dominating role，而其他三种 role 均为 dominated role。dominating role 具有所有 dominated role 的“能力”，因此 rootuser_r 能够和所有能够和 sysadm_r/secadm_r/auditadm_r 相结合的 type 相结合，即 rootuser_r 的 role_datum_t.types 为后三者的超集。所以，role-dominance 规则的核心操作就是将 dominated role 的 role_datum_t.types 集合**合并**到 dominating role 的 role_datum_t.types.types 位图。

下面我们首先分析 role-dominance 规则的词法分析函数：

```
role_datum_t *define_role_dom(role_datum_t *r)
{
        role_datum_t *role;
        char *role_id;
        ebitmap_node_t *node;
        unsigned int i;
        int ret;
```

参数 r 指向 dominated role 的 role_datum_t 数据结构。

```
        if (pass == 1) {
                role_id = queue_remove(id_queue);
                free(role_id);
                return (role_datum_t *) 1;      /* any non-NULL value */
        }

        yywarn("Role dominance has been deprecated");

        role_id = queue_remove(id_queue);
        if (!is_id_in_scope(SYM_ROLES, role_id)) {
                yyerror2("role %s is not within scope", role_id);
                free(role_id);
                return NULL;
        }
```

首先从 id_queue 中获得 dominating role 的名称字符串。role-dominance 规则除了指定 dominating/dominated role 之间的关系外，还可以用于定义一个 dominating role。因此首先在当前模块的 p_roles 符号表中查找，如果查找失败则插入：

```
        role = (role_datum_t *)hashtab_search(policydbp->p_roles.table, role_id);
        if (!role) {
                role = (role_datum_t *)malloc(sizeof(role_datum_t));
                if (!role) {
                        yyerror("out of memory");
                        free(role_id);
                        return NULL;
                }
                memset(role, 0, sizeof(role_datum_t));
                ret = declare_symbol(SYM_ROLES, (hashtab_key_t)role_id,
                                     (hashtab_datum_t)role, &role->s.value, &role->s.value);
                switch (ret) {
                case -3:{
                                yyerror("Out of memory!");
                                goto cleanup;
                        }
                case -2:{
                                yyerror2("duplicate declaration of role %s",
                                         role_id);
                                goto cleanup;
                        }
                case -1:{
                                yyerror("could not declare role here");
                                goto cleanup;
                        }
                case 0:
                case 1:{
                                break;
```

```
                }
        default:{
                        assert(0);      /* should never get here */
                }
        }
        if (ebitmap_set_bit(&role->dominates, role->s.value - 1, TRUE)) {
                yyerror("Out of memory!");
                goto cleanup;
        }
```

任何一个 role，其 role_datum_t.dominates 位图总是“自包含”的。

```
        }
```

接下来处理 dominating/dominated role 之间的关系：

```
        if (r) {
                ebitmap_t types;
                ebitmap_init(&types);
                ebitmap_for_each_bit(&r->dominates, node, i) {
                        if (ebitmap_node_get_bit(node, i))
                                if (ebitmap_set_bit(&role->dominates, i, TRUE))
                                        goto oom;
                }
```

首先将 dominated role 的 dominates 位图，合并（escalate）到 dominating role 中。这样可以支持若干层次的 role-dominance 关系。

```
                if (type_set_expand(&r->types, &types, policydbp, 1)) {         # alwaysexpand == 1
                        ebitmap_destroy(&types);
                        return NULL;
                }

                ebitmap_for_each_bit(&types, node, i) {
                        if (ebitmap_node_get_bit(node, i))
                                if (ebitmap_set_bit (&role->types.types, i, TRUE))
                                        goto oom;
                }
                ebitmap_destroy(&types);
```

然后将 dominated role 的 role_datum_t.types type_set_t 集合，扩展为 types 临时位图（处理特殊字符以及 type 属性），然后将 types 临时位图合并到 dominating role 的 role_datum_t.types.types 位图中。

```
                if (!r->s.value) {
                        /* free intermediate result */
                        type_set_destroy(&r->types);
                        ebitmap_destroy(&r->dominates);
                        free(r);
                }
                /*
                 * Now go through all the roles and escalate this role's
                 * dominates and types if a role dominates this role.
                 */
                hashtab_map(policydbp->p_roles.table, dominate_role_recheck, role);
```

最后遍历当前所有定义的 role，通过 dominate_role_recheck 函数将当前 dominating role 的 dominates

和 role_datum_t.types.types 位图，进一步合并（escalate）到 dominating 它的那些 role 中（即传递到 role-dominance 层次关系中的更高层次）。这样可以解决 refpolicy 中若干相关 role-dominance 规则的出现顺序问题（比如先处理了 A dominates B，后来又处理了 B dominates C，则需要将 B 的 dominates 和 types.types 位图（包含了 C 的）更新到 A 中）。

```
        }
        return role;
    cleanup:
        free(role_id);
        role_datum_destroy(role);
        free(role);
        return NULL;
    oom:
        yyerror("Out of memory");
        goto cleanup;
}
```

在 define_role_dom 函数中处理了一对 role 的 dominating/dominated 关系后，需要遍历当前模块 p_roles 符号表的所有 role，将当前 dominating role 的 dominates 和 role_datum_t.types.types 位图，进一步传递（escalate）到 dominating 它的那些 role 中，从而解决 refpolicy 中若干相关 role-dominance 规则的出现顺序问题。

```
/* This function eliminates the ordering dependency of role dominance rule */
static int dominate_role_recheck(hashtab_key_t key, hashtab_datum_t datum, void *arg)
{
        role_datum_t *rdp = (role_datum_t *)arg;
        role_datum_t *rdatum = (role_datum_t *)datum;
        ebitmap_node_t *node;
        int i;
```

rdp 指向 dominating role，而 rdatum 指向 p_roles 符号表中的当前 role 的 role_datum_t 数据结构。

```
        /* Don't bother to process against self role */
        if (rdatum->s.value == rdp->s.value)
                return 0;
```

显然如果遍历到 dominating role 自己，则直接退出。否则，检查当前 role 是否 dominate rdp，即 (rdp->s.value - 1)是否在 rdatum->dominates 位图中：

```
        /* If a dominating role found */
        if (ebitmap_get_bit(&(rdatum->dominates), rdp->s.value - 1)) {
                ebitmap_t types;
                ebitmap_init(&types);
                if (type_set_expand(&rdp->types, &types, policydbp, 1)) {
                        ebitmap_destroy(&types);
                        return -1;
                }
                /* raise types and dominates from dominated role */
                ebitmap_for_each_bit(&rdp->dominates, node, i) {
                        if (ebitmap_node_get_bit(node, i))
                                if (ebitmap_set_bit (&rdatum->dominates, i, TRUE))
                                        goto oom;
                }

                ebitmap_for_each_bit(&types, node, i) {
                        if (ebitmap_node_get_bit(node, i))
                                if (ebitmap_set_bit (&rdatum->types.types, i, TRUE))
```

```
                                goto oom;
            }
            ebitmap_destroy(&types);
        }
```

如果是，则将 rdp->dominates 位图合并到 rdatum->dominates 位图，并将 rdp->types 集合展开为临时 types 位图，最终合并到 rdatum->types.types 位图中。

```
        /* go through all the roles */
        return 0;
      oom:
        yyerror("Out of memory");
        return -1;
}
```

比如，假如 role_a 为 role_b 的超集，则后来定义 role_b 为 role_c 的超集时，也将合并了 role_c 的 types.types 和 dominates 位图的 role_b 的相应位图，进一步传递（escalate）到 role_a 的相应集合中，从而实现在 role_a -> role_b -> role_c 层级关系中，role_a 为所有子集的超集。

我们可以看出，尽管 dominate_role_recheck 函数可以很好地处理相关的若干 role-dominance 规则之间的编译顺序问题，role-dominance 规则存在一个致命的局限性：假设 role_a -> role_b，在相应 role-dominance 规则之后又通过 role-types 规则许可 role_b 和新的 type 相结合，则更新了的 role_b 的 types 集合却无法再传递（escalate）到 role_a 了！

造成这种局限性的原因，恰恰在于 role-types 规则在更新了一个 role 的 types.types 位图时，无法及时传递给（escalate）它的 dominating role。退一步，即使 role-types 规则考虑了对 role-dominance 的影响，由于可能存在 role-dominance 的层次关系，在编译时一直传递到最上层的 role 的代价会非常大。

一个可能的解决办法就是在模块的 link 阶段的最后，遍历 base.p_roles 符号表两遍：第一遍合并一个 role 的最终的 types.types 位图；第二遍，就当前 role 遍历 base.p_roles 符号表：如果能找到它的 dominating role，将其 domiantes/types.types 位图传递（escalate）到 dominating role 的相应位图。必须注意的是，该 dominating role 有可能被更高层次的其他 role 所 dominated，因此必须一直传递到更高层次的 role 为止（可以使用递归函数实现，退出条件为无法找到更高层次的 dominating role）。

9.4.2 提出问题之二：期望的 role attribute 使用模型

在邮件讨论中 cjp 希望使用 role attribute 来方便地处理“chain of run interface”的问题，可以有两点改进。比如假设有一个管理员角色(admin_t，admin_r)，希望 admin_t 能够 transition 到 foo_t，而 foo_t 能够进一步 transition 到 bar_t（注意并不要求 admin_t 能够直接转换到 bar_t），当前实现方法如下：

```
foo.te:
bar_domtrans(foo_t)                     # 使能 foo_t -> bar_t 的转换

foo.if:
interface foo_run(
        foo_domtrans($1)                # 使能$1 -> foo_t 的转换
        role $2 types foo_t;
        bar_run(foo_t,$2)               # 使能 foo_t -> bar_t 的转换，冗余，真正需要的 role $2 types bar_t;
)

bar.if:
```

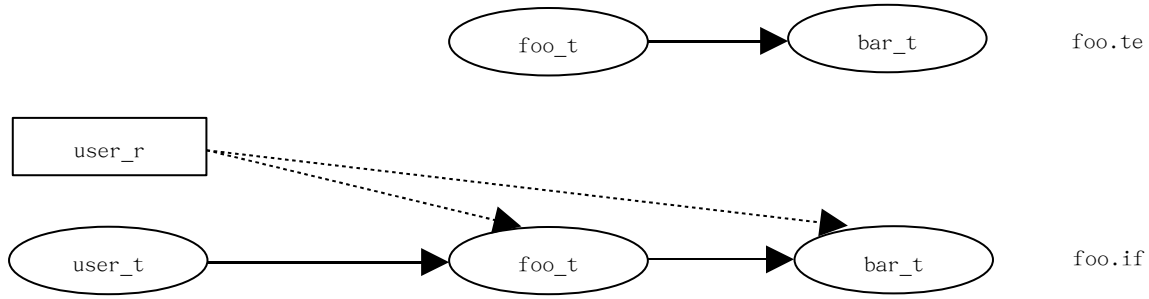
```
interface bar_run(
        bar_domtrans($1)                # 使能$1 -> bar_t的转换
        role $2 types bar_t;
)
```

假设在foo程序执行期间需要fork子进程以执行bar程序（比如在调用newrole程序时需要调用unix_chkpwd程序检查密码），那么通常在foo.te中即调用bar_domtrans接口以许可从foo_t到bar_t的转换。

在foo_run接口中调用bar_run接口的主要目的是为了让用户角色$2能够和bar_t相关联从而组成合法的SC。但是直接调用bar_run接口也重复指定了foo_t到bar_t的转换，这个不是必须的，真正需要的是“role $2 types bar_t;”规则，但是出于模块封装性的考虑，不能在foo_run接口中直接使用该规则，而应该调用bar_run接口。

另外，如果忘记了在foo_run中调用bar_run，则$1 -> foo_t -> bar_t的转换就无法完成，因为$2无法和bar_t相组合。

这种实现方法可以图示如下：



其中实线箭头表示domain transition，而虚线箭头表示role和type的关联。注意user_r能够和foo_t以及bar_t相关联，是靠在foo_run接口中调用bar_run接口实现的。

如果使用role attribute，则上述情景可以按照如下方法实现：

```
foo.te:
attribute_role foo_roles;               # 定义foo_roles属性
role foo_roles types foo_t;             # foo_roles属性能够和foo_t相组合
bar_run(foo_t, foo_roles)               # 许可foo_t -> bar_t，且foo_roles属性能够和bar_t相组合

foo.if:
interface foo_run(
        foo_domtrans($1)
        roleattribute $2 foo_roles;     # 将$2加入foo_roles属性
)
```

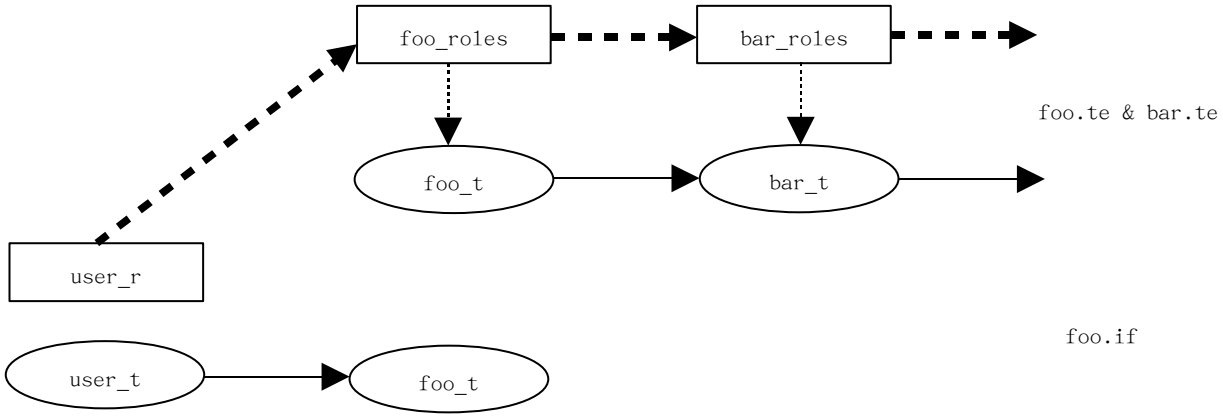
即在foo.te中而不是在foo.if的foo_run接口中调用bar_run接口，而且许可foo_roles属性能够和foo_t以及bar_t相组合。这样在foo_run接口中除了使能$1 -> foo_t的转换之外，只需使用roleattribute规则将$2加入foo_roles属性即可。

注意在bar.pp中也用相同的思路实现：

```
bar.te:
attribute_role bar_roles;
```

```
role bar_roles types bar_t;
xxx_run(bar_t, bar_roles)

bar.if:
interface bar_run(
        bar_domtrans($1)
        roleattribute $2 bar_roles;
)
```

上述实现可以图示如下：



图中加粗的虚线表示普通 role 或 role 属性之间的包含关系。

由于在 bar.te 中也可能会定义 bar_roles 属性，所以在 foo.te 中调用 bar_run(foo_t, foo_roles)时将会使用 roleattribute 规则将 foo_roles 属性加入 bar_roles 属性：

```
roleattribute foo_roles bar_roles;
```

那么，所有能够和 bar_roles 属性相结合的 type，也都被许可和 foo_roles 属性相结合。以此类推，最终使得**foo_roles 属性能够和所有从 foo_t 开始可以到达的 domain 相组合。**

注意，bar_roles 属性为 foo_roles 属性的超集意味着 bar_roles 的所有能力都被 foo_roles 继承，但是反之不然。所以实际上是 foo_roles 属性的能力为其所从属的 bar_roles 属性的超集。

9.4.3 分析问题

【编译阶段】
注：由于这部分和数据结构相关，而对数据结构的修改是无条件的，所以无须考虑 policy downgrade。（而根据数据结构生成 pp 或者 policy.X 时，可能需要放弃某些域，此即 policy downgrade）

1．数据结构的定义
1）在 role_datum_t 数据结构中增加标识普通 role 异或 role 属性的 flavor 域，以及 role 属性的 roles 位图。
2）在 attribute_role 规则中定义 role 属性时设置 flavor == ROLE_ATTRIB，而在 role-attr 规则中定义普通 role 时设置 flavor == ROLE_ROLE；在 role-attr 规则和 roleattribute 规则中设置 role 属性的 roles 位图。
3）在所有 role_datum_t 数据结构的初始化/释放函数中处理新增的 roles 位图。

2．词法分析
1）修改 declare_role 函数的实现：增加参数 isattr，根据它来设置 role_datum_t.flavor;
2）增加 attribute_role 规则用于定义一个 role 属性，调用 declare_role 函数时参数 isattr = TRUE;
3）增加 role-attr 规则用于定义一个普通 role，调用 declare_role 函数时参数 isattr = FALSE。而且支持同时定义它所从属的 role 属性;
4）增加 roleattribute 规则将一个已经定义过的普通 role 或者 role 属性，加入另一个 role 属性;
5）修改 role-types 规则的处理函数：不再调用 declare_role 来定义一个 role，而要求所涉及的 role 或者 role 属性此前必须被定义或者声明过，从而避免只定义一个普通 role 时的二义性（这种情况应该由 role-attr 规则来处理）
6）增加对 attribute_role 规则的 require 支持，从而能够将一个 role 属性声明为外部引用：

```
require_decl_def        : ROLE        { $$ = require_role; }
...
+                       | ATTRIBUTE_ROLE   { $$ = require_attribute_role; }
...
                        ;
```

将原来的 require_role 函数重命名为 require_role_or_attribute 并增加一个参数 isattr 。
require_role 和 require_attribute_role 分别以如下方式调用该新增函数：

```
require_role(pass) > require_role_or_attribute(pass, 0)
require_attribute_role(pass) > require_role_or_attribute(pass, 1)
```

3．思考所有使用 role 的规则的编译过程应该如何处理 role 属性
1）user-roles 规则
这里不需要任何额外的工作，如果当前 id 对应 role 属性，则也是将其 s.value - 1 设置到 user_datum_t.roles.roles 中。
2）role-allow 规则
这里不需要任何额外的工作，如果当前 id 对应 role 属性，则也是将其 s.value - 1 设置到 role_allow_rules 的 roles 或者 new_roles 域（均为 role_set_t 数据结构）的 roles 位图中。
3）role-types 规则
这里不需要任何额外的工作，无论为普通 role 或者 role 属性，define_role_type > set_types 都将 type 的 s.value - 1 设置到 role_datum_t.types 中。
4）role-transition 规则
对于 subject_role 域无须对 role 属性做额外的处理，也是将其 s.value - 1 设置到 role_trans_rule_t 的 roles.roles 位图中;
但是需要增加对 new_role 的检查，不能为 role 属性！

【pp 的创建和读取】
由于 policy.X 中的 role_datum_t 的二进制表示不需要含有 roles 位图信息（在 expand 阶段已经处理过了），所以不需要定义新的 policy.X 的版本，那么 SELinux 内核驱动和用户态 libsepol 都不需要增加 policy.X 的最大版本号。

但是 pp 中的 role_datum_t 的二进制表示包含 flavor 和 roles 位图信息，所以应该增加 pp 的最大版本号！

另外，由于在 expand 过程中 role 属性的能力已经被“散播”给从属于它的所有普通 role，所以在 policy.X 中不需要创建描述 role 属性的 role_datum_t 数据结构。

如果发生 pp downgrade，则 role_datum_t 的二进制描述将丢弃 flavor 标志和 roles 位图（从而和较低

版本的 pp 格式表示一致）。也正是因为这个原因，导致在 pp downgrade 时 role 属性完全没有作用，所以此时也应该忽略所有的 role 属性。

1）pp 的创建
一方面在 role_write 函数中，如果当前 role_datum_t.flavor == ROLE_ATTRIB，且写出 policy.X 或者发生 pp downgrade，则立即返回成功；

否则，在 pp 版本号不小于 MOD_POLICYDB_VERSION_ROLEATTR 时才写出 role_datum_t 的 roles 位图和 flavor 域。

另一方面在 policydb_write 函数中，在调用 role_write 函数之前，应该在相应的条件下从 p_roles.table 的元素个数 nel 中，扣除所有 role 属性的个数。然后再转换为小头并写入 policy.X。

2）pp 的读取
在解析 role_datum_t 的二进制表示时读取 roles 位图和 flavor 域，如果 pp 版本号不小于 MOD_POLICYDB_VERSION_ROLEATTR;

注意，无论读写，只需针对 module 才需要考虑 flavor 和 roles 位图，因为 policy.X 中不需要这两个域！而且只在 pp 版本号不小于 MOD_POLICYDB_VERSION_ROLEATTR 时进行，从而使得在升级 libsepol/checkpolicy 后，仍然能够正确处理老版本工具所创建的老版本的 pp。（“升级 libsepol 工具并不应该强制要求重新编译所有的 pp”，这是 Steve Lawrance 的建议）

【模块的 link】
注，由于这部分和数据结构相关，所以无须考虑 policy downgrade。

1）在 role_copy_callback 函数中检查是否出现冲突的定义（普通 role VS role 属性），并且复制 flavor（注意此时并没有复制 base 中 role 属性的 roles 位图，而由 role_fix_callback 函数完成）。
2）在 role_fix_callback 函数中修正 role 属性的 roles 位图：remap 后和 base 模块相应 role 属性的 roles 位图相合并。
3）user-roles 规则，role_allow 规则和 role_transition 规则在 link 阶段无须关心 role_set_t.roles 位图中的非 0 位是否为属性，直接 remap 并合并即可（只有等到 expand 的时候才需要真正关心）。
4）在 link_modules 的后部，调用 expand_role_attributes 函数遍历 base 模块中的所有 role 属性：如果其 roles 位图中的非 0 位代表一个 role 属性，则将该位**清 0** 并将子 role 属性的 roles 位图和父 role 属性的 roles 位图合并；重复检查父 role 属性的 roles 位图直到其中的非 0 位都为普通 role 为止，从而确保高级 role 属性的 roles 位图，含有**所有**低级 role 属性的 roles 位图（注意实际上越高级的 role 属性代表更为一般/普通化的能力，而较低级的 role 属性可以在此基础上有自己的“个性”）。

注意，expand_role_attribute 函数可以确保在 link 过程结束后，base.p_roles 符号表中所有 role 属性的 roles 位图不再包含其他 role 属性，从而为 expand 过程的 role_fix_callback 函数中将 role 属性的 types 散播（populate）给所有从属的普通 role 的 types，以及 role_set_t.roles 的处理创造所需的必要条件。参见下文。

另外，在 expand 过程的 role_fix_callback 函数中使用 assert 宏，验证 role 属性在 out 模块中的 roles 位图，不再含有任何其他 role 属性。expand_role_attribute 函数即为该 assert 提供必须的支撑。

无论 role 属性在哪里定义（无论 global block 或者一个 optional block），只要当 roleattribute 或者 role-attr 规则出现在一个 optional block 中，get_local_role 函数都将获得其在当前 block/decl 中的副本，并将普通 role 和 role 属性的关系记录到这个**副本的 roles 位图**中（也就是记录到当前 block/decl 中，那么如果该 decl 无效则不会污染 base.p_roles 符号表）。所以，必须在调用上述

expand_role_attribute 函数之前必须先调用 populate_roleattributes 函数遍历 base 模块 global 队列中从第 2 个 block 开始的所有 block，处理当前 block 的使能了的 decl 的 symtab[SYM_ROLES]符号表，将其中所有 role 属性的 roles 位图，**合并**到它在 base.p_roles 符号表中相应元素中（由 link 过程保证在 block/decl 中定义的标识符，其在 base.p_xxx 符号表中的副本一定存在）。

如果少了这一步，那么如果用 optional_block 宏来修饰 roleattribute 或者 role-attr 规则（即将它们放到一个 optional block，而不是当前模块的 global block 中时），相应的从属关系将会被遗漏！

至此，get_local_role 函数的影响/行为，将被 populate_roleattributes 函数的影响/行为**互补**。

【模块的 expand】
注，由于这部分和数据结构相关，所以无须考虑 policy downgrade。

1）在 role_copy_callback 函数中拷贝 role_datum_t 时，同时拷贝 flavor（注意普通 role 所从属的 role 属性的能力（types.types 位图），尚未散播到普通 role。该工作由新增 role_fix_callback 函数完成，因为首先需要处理完毕普通 role 和 role 属性的 types type_set_t 数据结构，并确定 expand_state_t.rolemap 数组）。
2）在 role_fix_callback 函数中处理 role 属性：拷贝 role_datum_t.roles 位图；取出其 role_datum_t.roles 位图中的所有非 0 位（都为普通 role，由 link_modules > expand_role_attributes 来保证），得到相应普通 role 在 out 模块中的 role_datum_t，然后将当前 role 属性在 out 模块中的 types.types 位图，散播（populate）到普通 role 在 out 中的 role_datum_t 的 types.types 位图中！
3）在 role_set_expand 函数中扩展 role_set_t.roles 位图中的非 0 位，如果它为 role 属性则与其 roles 位图相合并。

注意：
1）在 link_modules > expand_role_attributes 已经保证 role 属性的 roles 位图只含有普通 role，因此只需展开一次即可。
2）所有和 role 标识符相关的规则在展开时也要考虑相应的标识符是否为 role 属性，而这些工作都通过 role_set_expand 来完成！
3）在 expand 过程中也只需要考虑 base.p_roles 中的 role 属性，而无须再关心任何 block/decl 的私有 p_roles 符号表中的 role 属性（因为它们已经被 populate_roleattributes 函数“提升”到 base.p_roles 中了）。

9.4.4 解决问题

【编译阶段】
1，首先在 role_datum_t 数据结构中增加 flavor 域，用于描述为普通 role 或者 role 属性，并随即定义相应的宏。而 role 属性的核心数据结构即为 roles 位图，用于记录所有属于该 role 属性的普通 role 的 (policy value - 1)：

```
typedef struct role_datum {
        symtab_datum_t s;
        ebitmap_t dominates;    /* set of roles dominated by this role */
        type_set_t types;       /* set of authorized types for role */
        ebitmap_t cache;        /* This is an expanded set used for context validation during parsing */
        uint32_t bounds;        /* bounds role, if exist */
+#define ROLE_ROLE 0             /* regular role in kernel policies */
+#define ROLE_ATTRIB 1           /* attribute */
+        uint32_t flavor;
+        ebitmap_t roles;        /* roles with this attribute */
```

```
} role_datum_t;
```

注意：
1）对于普通 role，roles 位图不会被使用到；
2）把普通 role 对应的 ROLE_ROLE 定义为 0，那么在分配 role_datum_t 数据结构时默认为普通 role。这样的好处是在读取较低版本的 pp 时，无法（无须）读取 flavor 标志和 roles 位图，此时无须显示设置 flavor 标志；

2，然后定义 role-attr，roleattribute，attribute_role 规则的语法，并修改 role-types 规则的功能：

```
-rbac_dec1              : role_type_def
+rbac_dec1              : attribute_role_def                          # 新增 attribute_role 规则
+                       | role_type_def
                        | role_dominance
                        | role_trans_def
                        | role_allow_def
+                       | roleattribute_def                           # 新增 roleattribute 规则
+                       | role_attr_def                               # 新增 role-attr 规则
                        ;

+attribute_role_def     : ATTRIBUTE_ROLE identifier ';'
+                       {if (define_attrib_role()) return -1; }
+                       ;
 role_type_def          : ROLE identifier TYPES names ';'
                        {if (define_role_types()) return -1;}
-                       | ROLE identifier';'                        # 原有 role-types 规则不再用于定义 role
-                       {if (define_role_types()) return -1;}
+                       ;
+role_attr_def          : ROLE identifier opt_attr_list ';'
+                       {if (define_role_attr()) return -1;}
                        ;
```

attribute_role 规则用于定义一个 role 属性，而 role-attr 规则既可以定义一个普通 role，也可以同时声明它所从属的 role 属性（由 opt_attr_list 列表描述）。注意 role 属性只能用 attribute_role 规则来定义，而普通 role 只能用 role-attr 规则来定义，它们在调用 declare_role 函数时将分别传递 TRUE/FALSE 参数。另外，原来的 role-types 规则不再用于定义一个普通 role，而仅声明普通 role 或者 role 属性能够结合的 types 集合。

```
@@ -440,6 +448,9 @@ role_def            : ROLE identifier_push ';'
                        | ROLE identifier_push '{' roles '}'
                         {$$ = define_role_dom((role_datum_t*)$4); if ($$ == 0) return -1;}
                        ;
+roleattribute_def      : ROLEATTRIBUTE identifier id_comma_list ';'
+                       {if (define_roleattribute()) return -1;}
+                       ;
```

roleattribute 规则用于将一个普通 role，或者一个 role 属性，加入 id_comma_list 列表（以逗号分割的 token 序列）中的每一个 role 属性。

```
 require_dec1_def       : ROLE        { $$ = require_role; }
                        | TYPE        { $$ = require_type; }
                        | ATTRIBUTE   { $$ = require_attribute; }
+                       | ATTRIBUTE_ROLE   { $$ = require_attribute_role; }
                        | USER        { $$ = require_user; }
                        | BOOL        { $$ = require_bool; }
                        | SENSITIVITY { $$ = require_sens; }
```

如果 attribute_role 规则在 gen_require 宏内被使用，则调用 require_attribute_role 函数处理，即声明一个 role 属性为外部依赖。

3，接下来提供相应规则的语法分析函数：

```
-role_datum_t *declare_role(void)
+role_datum_t *declare_role(unsigned char isattr)
{
        char *id = queue_remove(id_queue), *dest_id = NULL;
        role_datum_t *role = NULL, *dest_role = NULL;
        int retval;
        uint32_t value;

        if (id == NULL) {
                yyerror("no role name");
                return NULL;
        }
        if ((role = (role_datum_t *)malloc(sizeof(*role))) == NULL) {
                yyerror("Out of memory!");
                free(id);
                return NULL;
        }
        role_datum_init(role);
+       role->flavor = isattr ? ROLE_ATTRIB : ROLE_ROLE;
        retval = declare_symbol(SYM_ROLES, id, (hashtab_datum_t *) role, &value, &value);
```

修改 declare_role 函数的型构增加参数 isattr，根据该参数的值设置 role_datum_t.flavor 域。在 attribute_role 规则中传递 TRUE，而在 role-attr 规则中传递 FALSE。

```
+role_datum_t *get_local_role(char *id, uint32_t value, unsigned char isattr)
+{
+       role_datum_t *dest_roledatum;
+       hashtab_t roles_tab;
+
+       assert(stack_top->type == 1);
+
+       if (stack_top->parent == NULL) {
+               /* in global, so use global symbol table */
+               roles_tab = policydbp->p_roles.table;
+       } else {
+               roles_tab = stack_top->decl->p_roles.table;
+       }
+
+       dest_roledatum = hashtab_search(roles_tab, id);
+       if (!dest_roledatum) {
+               dest_roledatum = (role_datum_t *)malloc(sizeof(role_datum_t));
+               if (dest_roledatum == NULL) {
+                       free(id);
+                       return NULL;
+               }
+
+               role_datum_init(dest_roledatum);
+               dest_roledatum->s.value = value;
+               dest_roledatum->flavor = isattr ? ROLE_ATTRIB : ROLE_ROLE;
+
+               if (hashtab_insert(roles_tab, id, dest_roledatum)) {
+                       free(id);
+                       role_datum_destroy(dest_roledatum);
```

```
+                       free(dest_roledatum);
+                       return NULL;
+               }
+       } else {
+               free(id);
+               if (dest_roledatum->flavor != isattr ? ROLE_ATTRIB : ROLE_ROLE)
+                       return NULL;
+       }
+
+       return dest_roledatum;
+}
```

增加 get_local_role 函数，获得当前 role-attr 规则或者 roleattribute 规则**所在 block 的** symtab[SYM_ROLES]符号表，然后在该符号表中查找相应 role 属性是否被注册过。如果没有则注册，如果注册过则检查 flavor 是否一致。

这样做的原因是，相应 role 属性的定义或声明 block，以及当前 role-attr 规则或者 roleattribute 规则所在的 block，可能不是同一个 block！因为 role 属性定义或者声明最终被记录在当前模块的 unconditional block 中（由 symtab_insert 函数的实现可见，它被 decare_symbol 和 require_symbol 函数调用），而相应 role-attr/roleattribute 规则可能位于一个 optional block 中。那么，如果 optional block 的外部依赖无法满足，则其中所有规则定义都应该失效。所以应该把普通 role 和 role 属性之间的联系，记录到 role-attr/roleattribute 规则所在 block 的 symtab[SYM_ROLES]符号表中，否则可能“污染”当前模块的全局符号表。

在 role-attr 规则和 roleattribute 规则的词法分析函数中都需要调用 get_local_role 函数。

```
+int require_role(int pass)
+{
+       return require_role_or_attribute(pass, 0);
+}
+
+int require_attribute_role(int pass)
+{
+       return require_role_or_attribute(pass, 1);
+}
+
```

新增 require_role_or_attribute 函数，并使得 require_role 函数和 require_attribute_role 函数成为它的封装函数，它们在调用前者时分别传递 0 和 1 为第 2 个参数，指名声明为外部依赖的符号为普通 role 或者 role 属性。

```
-int require_role(int pass)
+static int require_role_or_attribute(int pass, unsigned char isattr)
 {
        char *id = queue_remove(id_queue);
        role_datum_t *role = NULL;
@@ -831,6 +881,7 @@ int require_role(int pass)
                return -1;
        }
        role_datum_init(role);
+       role->flavor = isattr ? ROLE_ATTRIB : ROLE_ROLE;
        retval = require_symbol(SYM_ROLES, id, (hashtab_datum_t *)role, &role->s.value, &role->s.value);
```

同时 require_role_or_attribute 函数即为原来的 require_role 函数增加了 isattr 参数，根据它来设置声明为外部依赖的 role 标识符的 role_datum_t.flavor 域。（声明一个普通 role 或者 role 属性为外部依赖时也会把它们的 role_datum_t 数据结构注册到当前模块的全局符号表中，正如定义它们时那样，因

此也需要设置 flavor 域）

```
+/* The role-types rule is no longer used to declare regular role or
+ * role attribute, but solely aimed for declaring role-types associations.
+ */
 int define_role_types(void)
 {
        role_datum_t *role;
@@ -1786,9 +1789,25 @@ int define_role_types(void)
                return 0;
        }

-       if ((role = declare_role()) == NULL) {
+       id = (char *)queue_remove(id_queue);
+       if (!id) {
+               yyerror("no role name for role-types rule?");
+               return -1;
+       }
+
+       if (!is_id_in_scope(SYM_ROLES, id)) {
+               yyerror2("role %s is not within scope", id);
+               free(id);
+               return -1;
+       }
+
+       role = hashtab_search(policydbp->p_roles.table, id);
+       if (!role) {
+               yyerror2("unknown role %s", id);
+               free(id);
+               return -1;
        }
```

修改 role-types 函数的语法实现：不再直接调用 declare_role 函数（用于定义一个 role），而仅是定义和一个 role 标识符（无论普通 role 亦或 role 属性）相关联的 types 集合。所以直接要求该 role 标识符在当前模块的全局符号表中被注册过（注意，无论在模块的哪个 block 中定义或者声明一个标识符，相关 xxx_datum_t 数据结构都会被注册到该模块的全局符号表中。参见 symbol_insert 函数）。

```
+int define_attrib_role(void)
+{
+       if (pass == 2) {
+               free(queue_remove(id_queue));
+               return 0;
+       }
+
+       /* Declare a role attribute */
+       if (declare_role(TRUE) == NULL)
+               return -1;
+
+       return 0;
+}
```

attribute_role 规则的语法函数很简单，直接调用 declare_role(TRUE)函数定义一个 role 属性即可。

```
+int define_role_attr(void)
+{
+       char *id;
+       role_datum_t *r, *attr;
+
```

```
+       if (pass == 2) {
+               while ((id = queue_remove(id_queue)))
+                       free(id);
+               return 0;
+       }
+
+       /* Declare a regular role */
+       if ((r = declare_role(FALSE)) == NULL)
+               return -1;
+
+       while ((id = queue_remove(id_queue))) {
+               if (!is_id_in_scope(SYM_ROLES, id)) {
+                       yyerror2("attribute %s is not within scope", id);
+                       free(id);
+                       return -1;
+               }
+               attr = hashtab_search(policydbp->p_roles.table, id);
+               if (!attr) {
+                       /* treat it as a fatal error */
+                       yyerror2("role attribute %s is not declared", id);
+                       free(id);
+                       return -1;
+               }
+
+               if (attr->flavor != ROLE_ATTRIB) {
+                       yyerror2("%s is a regular role, not an attribute", id);
+                       free(id);
+                       return -1;
+               }
+
+               if ((attr = get_local_role(id, attr->s.value, 1)) == NULL) {
+                       yyerror("Out of memory!");
+                       return -1;
+               }
+
+               if (ebitmap_set_bit(&attr->roles, (r->s.value - 1), TRUE)) {
+                       yyerror("out of memory");
+                       return -1;
+               }
+       return 0;
+}
```

而 role-attr 规则通过调用 declare_role(FALSE)来定义一个普通 role，并且支持同时指定该普通 role 所从属的若干 role 属性。注意在 while 循环中处理每个可能的 role 属性，首先也是在当前模块的全局 p_roles 符号表中检查其是否被注册过，然后再通过 get_local_role 函数获得该 role 属性在当前 role-attr 规则所在 block 的 symtab[SYM_ROLES]符号表中的 role_datum_t，然后将从属关系记录到其 roles 位图中（而不一定就是 p_roles 符号表中的 role_datum_t.roles 位图）。

```
+int define_roleattribute(void)
+{
+       char *id;
+       role_datum_t *r, *attr;
+
+       if (pass == 2) {
+               while ((id = queue_remove(id_queue)))
+                       free(id);
+               return 0;
+       }
```

```
+
+       id = (char *)queue_remove(id_queue);
+       if (!id) {
+               yyerror("no role name for roleattribute definition?");
+               return -1;
+       }
+
+       if (!is_id_in_scope(SYM_ROLES, id)) {
+               yyerror2("role %s is not within scope", id);
+               free(id);
+               return -1;
+       }
+       r = hashtab_search(policydbp->p_roles.table, id);
+       if (!r || r->flavor != ROLE_ROLE) {
+               yyerror2("unknown role %s, or not a regular role", id);
+               free(id);
+               return -1;
+       }
+
+       while ((id = queue_remove(id_queue))) {
+               if (!is_id_in_scope(SYM_ROLES, id)) {
+                       yyerror2("attribute %s is not within scope", id);
+                       free(id);
+                       return -1;
+               }
+               attr = hashtab_search(policydbp->p_roles.table, id);
+               if (!attr) {
+                       /* treat it as a fatal error */
+                       yyerror2("role attribute %s is not declared", id);
+                       free(id);
+                       return -1;
+               }
+
+               if (attr->flavor != ROLE_ATTRIB) {
+                       yyerror2("%s is a regular role, not an attribute", id);
+                       free(id);
+                       return -1;
+               }
+
+               if ((attr = get_local_role(id, attr->s.value, 1)) == NULL) {
+                       yyerror("Out of memory!");
+                       return -1;
+               }
+               if (ebitmap_set_bit(&attr->roles, (r->s.value - 1), TRUE)) {
+                       yyerror("out of memory");
+                       return -1;
+               }
+       }
+
+       return 0;
+}
```

roleattribute 规则用于将一个普通 role 加入若干 role 属性，和上面 role-attr 规则不一样的地方在于要求该普通 role 标识符已经被定义或这声明过，因此直接在当前模块的 p_roles 符号表中查找而不是调用 declare_role 函数。而后面的实现和上一个函数相同。

【读写 pp】

无论 type 属性还是 role 属性，其内核描述符中都没有 flavor 域及相应的位图数据结构，因此 policy.X 中 type 属性和 role 属性的二进制描述也都没有 flavor 和 ebitmap 的二进制描述（这是由于在模块的 expand 过程中已经将规则中的 type 属性展开了，而且将 role 属性的能力广播到了其所有从属的普通 role 上）。所以，在创建 role_datum_t 数据结构的二进制描述时，只有在写入 pp 文件且**没有**发生 downgrade 时才需要写出 flavor 标志和 roles 位图。

如果写出 pp 但是发生 downgrade，则由于 flavor 标志和 roles 位图都被丢弃，造成 role 属性没有任何作用，所以也应该跳过为 role 属性创建二进制描述符的操作。

当 role 属性被跳过时，还需要从 p_roles.table->nel 中扣除 role 属性的个数。

升级后的 libsepol/checkpolicy 工具仍然应该能够正确读取较低版本的 pp，或者生成较低版本的 pp，所以尽管无须增加 policy.X 的最大版本号，也应该增加 pp 的最大版本号。

```
#define MOD_POLICYDB_VERSION_ROLEATTRIB         13
#define MOD_POLICYDB_VERSION_MAX MOD_POLICYDB_VERSION_ROLEATTRIB

@@ -972,6 +972,19 @@ static int role_write(hashtab_key_t key, hashtab_datum_t datum, void *ptr)

        role = (role_datum_t *) datum;

+       /*
+        * Role attributes are redundant for policy.X, skip them
+        * when writing the roles symbol table. They are also skipped
+        * when pp is downgraded.
+        *
+        * Their numbers would be deducted in policydb_write().
+        */
+       if ((role->flavor == ROLE_ATTRIB) &&
+           ((p->policy_type == POLICY_KERN) ||
+            (p->policy_type != POLICY_KERN &&
+             p->policyvers < MOD_POLICYDB_VERSION_ROLEATTRIB)))
+               return POLICYDB_SUCCESS;
+
        len = strlen(key);
        items = 0;
        buf[items++] = cpu_to_le32(len);

......

+       if (p->policy_type != POLICY_KERN &&
+           p->policyvers >= MOD_POLICYDB_VERSION_ROLEATTRIB) {
+               buf[0] = cpu_to_le32(role->flavor);
+               items = put_entry(buf, sizeof(uint32_t), 1, fp);
+               if (items != 1)
+                       return POLICYDB_ERROR;
+
+               if (ebitmap_write(&role->roles, fp))
+                       return POLICYDB_ERROR;
+       }
+
        return POLICYDB_SUCCESS;
 }
```

用户态的 policydb_t 数据结构可用于描述 pp 和 policy.X，policy_type 域指名所描述对象的类型。如果为 POLICY_KERN（即为 policy.X），则跳过 role 属性；如果创建 pp 且发生 downgrade，则也跳过 role 属

性，且所有普通 role 的二进制描述中不包含 flavor 标志和 roles 位图；只有在 pp 版本号不小于 MOD_POLICYDB_VERSION_ROLEATTRIB 时才包含所有普通 role 和 role 属性，并写出 flavor 和 roles 位图的二进制描述。

写出 policy.X 时需要相应地从 p_roles.table->nel 中扣除 role 属性的个数:

```
+static int role_attr_uncount(hashtab_key_t key __attribute__ ((unused)),
+                              hashtab_datum_t datum, void *args)
+{
+       role_datum_t *role = datum;
+       uint32_t *p_nel = args;
+
+       if (role->flavor == ROLE_ATTRIB) {
+               /* uncount attribute from total number of roles */
+               (*p_nel)--;
+       }
+       return 0;
+}
+
 /*
  * Write the configuration data in a policy database
  * structure to a policy database binary representation
@@ -1926,7 +1947,7 @@ int policydb_write(policydb_t * p, struct policy_file *fp)
        num_syms = info->sym_num;
        for (i = 0; i < num_syms; i++) {
                buf[0] = cpu_to_le32(p->symtab[i].nprim);
-               buf[1] = cpu_to_le32(p->symtab[i].table->nel);
+               buf[1] = p->symtab[i].table->nel;

                /*
                 * A special case when writing type/attribute symbol table.
@@ -1939,6 +1965,20 @@ int policydb_write(policydb_t * p, struct policy_file *fp)
                    p->policy_type == POLICY_KERN) {
                        hashtab_map(p->symtab[i].table, type_attr_uncount, &buf[1]);
                }
+
+               /*
+                * Another special case when writing role/attribute symbol
+                * table, role attributes are redundant for policy.X, or
+                * when the pp's version is not big enough. So deduct
+                * their numbers from p_roles.table->nel.
+                */
+               if ((i == SYM_ROLES) &&
+                   ((p->policy_type == POLICY_KERN) ||
+                    (p->policy_type != POLICY_KERN &&
+                     p->policyvers < MOD_POLICYDB_VERSION_ROLEATTRIB)))
+                       hashtab_map(p->symtab[i].table, role_attr_uncount, &buf[1]);
+
+               buf[1] = cpu_to_le32(buf[1]);
                items = put_entry(buf, sizeof(uint32_t), 2, fp);
                if (items != 2)
                        return POLICYDB_ERROR;
                if (hashtab_map(p->symtab[i].table, write_f[i], &pd))
                        return POLICYDB_ERROR;
```

即在循环中通过 write_f[i]函数写入当前符号表 p->symtab[i].table 时，如果时 SYM_ROLES 符号表，则通过 hashtab_map 函数遍历它，通过 role_attr_uncount 函数从 p_roles.table->nel 中减去所有 role 属

性的个数。注意，必须在调整完毕后才能进行大小头的转换。

相应地，只有在读取 pp 文件时才需要读入 flavor 和 roles 位图。同样，如果从较低版本的 pp 中读取，则无须（无法）读取它们：

```
@@ -2071,6 +2071,17 @@ static int role_read(policydb_t * p
                if (type_set_read(&role->types, fp))
                        goto bad;
        }
+
+       if (p->policy_type != POLICY_KERN) {
+           p->policyvers >= MOD_POLICYDB_VERSION_ROLEATTRIB) {
+               rc = next_entry(buf, fp, sizeof(uint32_t));
+               if (rc < 0)
+                       goto bad;
+
+               role->flavor = le32_to_cpu(buf[0]);
+
+               if (ebitmap_read(&role->roles, fp))
+                       goto bad;
+       }
```

【模块的 link 过程】
在 link 过程中将各个模块的 role 属性向 base 模块合并时，需要检查是否出现 flavor 定义的冲突并合并 roles 位图。

在 role_copy_callback 函数中如果发现当前模块和 base 模块都定义了同一名字的 role 属性，则检查是否出现 flavor 定义冲突：

```
static int role_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        int ret;
        char *id = key, *new_id = NULL;
        role_datum_t *role, *base_role, *new_role = NULL;
        link_state_t *state = (link_state_t *)data;

        role = (role_datum_t *)datum;

        base_role = hashtab_search(state->base->p_roles.table, id);

+       if (base_role != NULL) {
+               /* role already exists.  check that it is what this
+                * module expected.  duplicate declarations (e.g., two
+                * modules both declare role foo_r) is checked during
+                * scope_copy_callback(). */
+               if (role->flavor == ROLE_ATTRIB && base_role->flavor != ROLE_ATTRIB) {
+                       ERR(state->handle,
+                           "%s: Expected %s to be a role attribute, but it was already declared as a
regular role.",
+                           state->cur_mod_name, id);
+                       return -1;
+               } else if (role->flavor != ROLE_ATTRIB && base_role->flavor == ROLE_ATTRIB) {
+                       ERR(state->handle,
+                           "%s: Expected %s to be a regular role, but it was already declared as a role
attribute.",
+                           state->cur_mod_name, id);
```

```
+                       return -1;
+               }
+       } else {
                if (state->verbose)
                        INFO(state->handle, "copying role %s", id);

                if ((new_id = strdup(id)) == NULL) {
                        goto cleanup;
                }

                if ((new_role =
                     (role_datum_t *) malloc(sizeof(*new_role))) == NULL) {
                        goto cleanup;
                }
                role_datum_init(new_role);

                /* new_role's dominates, types and roles field will be copied
                 * during role_fix_callback() */
+               new_role->flavor = role->flavor;
                new_role->s.value = state->base->p_roles.nprim + 1;
                ret = hashtab_insert(state->base->p_roles.table,
                                     (hashtab_key_t)new_id, (hashtab_datum_t)new_role);
                if (ret) {
                        goto cleanup;
                }
                state->base->p_roles.nprim++;
                base_role = new_role;
        }

        if (state->dest_decl) {
                new_id = NULL;
                if ((new_role = malloc(sizeof(*new_role))) == NULL) {
                        goto cleanup;
                }
                role_datum_init(new_role);
+               new_role->flavor = base_role->flavor;
                new_role->s.value = base_role->s.value;
                if ((new_id = strdup(id)) == NULL) {
                        goto cleanup;
                }
                if (hashtab_insert (state->dest_decl->p_roles.table, new_id, new_role)) {
                        goto cleanup;
                }
                state->dest_decl->p_roles.nprim++;
        }

        state->cur->map[SYM_ROLES][role->s.value - 1] = base_role->s.value;
        return 0;

      cleanup:
        ERR(state->handle, "Out of memory!");
        role_datum_destroy(new_role);
        free(new_id);
        free(new_role);
        return -1;
}
```

在将当前 role 属性向 base.p_roles 或者目的 block 的 symtab[SYM_ROLES]符号表中拷贝时，需要拷贝 flavor 域。

待调用 role_copy_callback 函数处理完当前模块 p_roles 符号表中的所有 role 标识符后，当前模块 policy_module_t.map[SYM_ROLES]数组建立完毕，此时就可以调用 role_fix_callback 函数处理 role 标识符的 dominates/types/roles 域了。就 role 属性的 roles 位图，借助上述 remap 数组得到临时位图 e_tmp，然后再和 base 模块中的 role_datum_t.roles 位图合并：

```
@@ -1046,6 +1066,24 @@ static int role_fix_callback(hashtab_key_t key, hashtab_datum_t datum,
                goto cleanup;
        }
        ebitmap_destroy(&e_tmp);
+
+       if (role->flavor == ROLE_ATTRIB) {
+               ebitmap_init(&e_tmp);
+               ebitmap_for_each_bit(&role->roles, rnode, i) {
+                       if (ebitmap_node_get_bit(rnode, i)) {
+                               assert(mod->map[SYM_ROLES][i]);
+                               if (ebitmap_set_bit(&e_tmp, mod->map[SYM_ROLES][i] - 1, 1)) {
+                                       goto cleanup;
+                               }
+                       }
+               }
+               if (ebitmap_union(&dest_role->roles, &e_tmp)) {
+                       goto cleanup;
+               }
+               ebitmap_destroy(&e_tmp);
+       }
+
```

注意，roles 位图中非 0 位为相应普通 role 的(policy value - 1)，而 remap 数组也以它为索引，因此可以直接使用。另外 remap 数组元素的值为 policy value，所以需要减 1 后记录到 e_tmp 临时位图中。

【模块的 expand 过程】
1，在 expand 过程中需要将 base 模块中 role 属性的 roles 位图复制到 out 模块中，因此 remap 位图的操作必须待 expand_state_t.rolemap[]数组建立好后进行。所以新增 role_fix_callback 函数，在 role_copy_callback 后调用完成上述操作。

另外 expand 过程还需要将 role 属性的“能力”（即 role_datum_t.types 集合）广播（populate）到所有成员普通 role 中，从而“兑现”role 属性的能力。该操作也放到 role_fix_callback 函数中完成。

注意先前在 role_copy_callback 函数中已经处理了所有 role 标识符在 out 模块中的 role_datum_t.types.types 位图了。

```
+/* For the role attribute in the base module, escalate its counterpart's
+ * types.types ebitmap in the out module to the counterparts of all the
+ * regular role that belongs to the current role attribute. Note, must be
+ * invoked after role_copy_callback so that state->rolemap is available.
+ */
+static int role_fix_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
+{
+       char *id, *base_reg_role_id;
+       role_datum_t *role, *new_role, *regular_role;
+       expand_state_t *state;
+       ebitmap_node_t *rnode;
+       unsigned int i;
+       ebitmap_t mapped_roles;
+
```

```
+	id = key;
+	role = (role_datum_t *)datum;
+	state = (expand_state_t *)data;
+
+	if (strcmp(id, OBJECT_R) == 0) {
+		/* object_r is never a role attribute by far */
+		return 0;
+	}
+
+	if (role->flavor != ROLE_ATTRIB)
+		return 0;
+
+	if (state->verbose)
+		INFO(state->handle, "fixing role attribute %s", id);
+
+	new_role = (role_datum_t *)hashtab_search(state->out->p_roles.table, id);
+
+	assert(new_role != NULL && new_role->flavor == ROLE_ATTRIB);
+
+	ebitmap_init(&mapped_roles);
+	if (map_ebitmap(&role->roles, &mapped_roles, state->rolemap))
+		return -1;
+	if (ebitmap_union(&new_role->roles, &mapped_roles)) {
+		ERR(state->handle, "Out of memory!");
+		ebitmap_destroy(&mapped_roles);
+		return -1;
+	}
+	ebitmap_destroy(&mapped_roles);
+
+	ebitmap_for_each_bit(&role->roles, rnode, i) {
+		if (ebitmap_node_get_bit(rnode, i)) {
+			/* take advantage of sym_val_to_name[] of the base module */
+			base_reg_role_id = state->base->p_role_val_to_name[i];
+			regular_role = (role_datum_t *)hashtab_search( state->out->p_roles.table,
+								base_reg_role_id);
+
+			assert(regular_role != NULL && regular_role->flavor == ROLE_ROLE);
+
+			if (ebitmap_union(&regular_role->types.types, &new_role->types.types)) {
+				ERR(state->handle, "Out of memory!");
+				return -1;
+			}
+		}
+	}
+
+	return 0;
+}
```

核心操作就是获得 role 属性 roles 位图中的每一个非 0 位，借助 base 模块的 p_role_val_to_name 数组得到相应普通 role 的名称字符串，然后在 out 模块的 p_roles 符号表中查找得到该普通 role 在 out 模块中的 role_datum_t，最后将 role 属性在 out 模块中的 new_role->types.types 位图合并到普通 role 在 out 模块中的相应位图，从而实现从属于一个 role 属性的所有普通 role，都可以和那些能够和 role 属性相结合的 types 相结合。

```
	/* copy roles */
	if (hashtab_map(state.base->p_roles.table, role_copy_callback, &state))
		goto cleanup;
	if (hashtab_map(state.base->p_roles.table, role_bounds_copy_callback, &state))
```

```
                goto cleanup;
+       /* escalate the type_set_t in a role attribute to all regular roles that belongs to it. */
+       if (hashtab_map(state.base->p_roles.table, role_fix_callback, &state))
+               goto cleanup;
```

注意，由于 link 过程的 populate_roleattributes 函数的行为已经**互补**了 get_local_role 函数的行为，即把 block/dec1 私有 p_roles 符号表中 role 属性的 roles 位图“提升”到了 base.p_roles 符号表中，在 expand 过程中就**只应该针对 base.p_roles 符号表调用 role_fix_callback 函数**而无须（也不应该）针对 block/dec1 的私有 p_roles 符号表！

否则，role_fix_callback 函数中如下 assert 宏：

```
        assert(regular_role != NULL && regular_role->flavor == ROLE_ROLE);
```

在 role 属性嵌套时就会失败！比如如下语句出现在一个 optional block 中时：

```
        roleattribute <sub role attr> <parent role attr>;
```

这是因为 link 过程中的 expand_role_attributes 函数只会处理 base.p_roles 符号表，而不是 block/dec1 的私有 p_roles 符号表，导致该私有 p_roles 符号表中父 role 属性的 roles 位图仍然记录有子 role 属性，进而导致上述 assert 宏的第 2 个条件失败。

2，expand 过程还需要展开 role_set_t.roles 中可能存在的 role 属性。

之前没有 role 属性时 role_set_t.roles 位图中只记录普通 role，所以如果 rolemap 转换数组有效，则将 role_set_t.roles 位图转换为 mapped_roles 临时位图；如果 rolemap 转换数组无效，则 mapped_roles 临时位图直接为其拷贝。最后再将 mapped_roles 位图中的每一位逐位拷贝到参数 r 所指向的输出位图中。

支持 role 属性后 role-allow 和 role-transition 规则中可能使用 role 属性，因此 role_set_t.roles 位图中的非 0 位可能为 role 属性。此时需要将普通 role 的非 0 位直接复制到另一个临时位图 roles 中，而将 role 属性的 roles 位图合并到临时位图 roles 中。

最后在 rolemap 有效的情况下将临时位图 roles 转换为 mapped_roles 位图。

```
-int role_set_expand(role_set_t * x, ebitmap_t * r, policydb_t * p, uint32_t * rolemap)
+/* Expand a role set into an ebitmap containing the roles.
+ * This handles the attribute and flags.
+ * Attribute expansion depends on if the rolemap is available.
+ * During module compile the rolemap is not available, the
+ * possible duplicates of a regular role and the role attribute
+ * the regular role belongs to could be properly handled by
+ * copy_role_trans and copy_role_allow.
+ */
+int role_set_expand(role_set_t * x, ebitmap_t * r, policydb_t * out, policydb_t * base, uint32_t *
rolemap)
        ...
+       ebitmap_t mapped_roles, roles;
+       policydb_t *p = out;
+       role_datum_t *role;

+       ebitmap_init(&mapped_roles);
```

```
+       ebitmap_init(&roles);
+
        if (rolemap) {
-               if (map_ebitmap(&x->roles, &mapped_roles, rolemap))
-                       return -1;
+               assert(base != NULL);
+               ebitmap_for_each_bit(&x->roles, rnode, i) {
+                       if (ebitmap_node_get_bit(rnode, i)) {
+                               /* take advantage of p_role_val_to_struct[] of the base module */
+                               role = base->role_val_to_struct[i];
+                               assert(role != NULL);
+                               if (role->flavor == ROLE_ATTRIB) {
+                                       if (ebitmap_union(&roles, &role->roles))
+                                               goto bad;
+                               } else {
+                                       if (ebitmap_set_bit(&roles, i, 1))
+                                               goto bad;
+                               }
+                       }
+               }
+               if (map_ebitmap(&roles, &mapped_roles, rolemap))
+                       goto bad;
        } else {
                if (ebitmap_cpy(&mapped_roles, &x->roles))
                        goto bad;
        }

        ebitmap_for_each_bit(&mapped_roles, rnode, i) {
                if (ebitmap_node_get_bit(rnode, i)) {
                        if (ebitmap_set_bit(r, i, 1))
                                goto bad;
                }
        }
```

注意：
1，函数原来的参数 p 恒为指向 out 模块的指针。现在把它重命名为 out，并且增加新指针 base，以便于获得 role_set_t.roles 位图中 role 属性的 role_datum_t - 借用 base 模块的 role_val_to_struct 数组，它在 expand 过程中一定可用！
2，在编译阶段 role-allow 规则和 role-trans 规则的词法分析函数中也会调用 role_set_expand 函数。注意此时为编译过程而非 expand 过程，所以 rolemap 不可用！在编译时展开的目的是为了避免添加完全重复的规则，此时我们不展开（也无法展开）role 属性，所以下面的两条规则被认为是不同的（假设 role_a 从属于 role_attr）：

```
role_transition role_a some_exec_type : file system_r;
role_transition role_attr some_exec_type : file system_r;
```

尽管如此，在 copy_role_allow 函数和 copy_role_trans 函数中最终会发现并避免重复的规则的“字面”描述。

【支持将一个 role 属性加入另外一个 role 属性】
由 role 属性的期望使用模型可知，应该支持将一个 role 属性加入另外一个 role 属性的 roles 位图。那么 role_datum_t.roles 位图中的非 0 位可能为一个 role 属性。

但是由上文可知，在 expand 过程中需要把一个 role 属性在 out 模块中的 types.types 位图，广播到其所有从属普通 role 属性在 out 模块中的相应位图。在 role_fix_callback 函数中认为 base 模块中一个 role 属性的 roles 位图中的所有非 0 位均为普通 role：

```
assert(regular_role != NULL && regular_role->flavor == ROLE_ROLE);
```

所以必须在 expand 过程调用 role_fix_callback 函数之前，将一个 role 属性的 roles 位图中所包含的其他 role 属性展开（即将子 role 属性的 roles 位图合并到父 role 属性的 roles 位图中），从而确保父 role 属性的 roles 位图中只包含普通 role。

由于子 role 属性中可能包含下一级别 role 属性，因此必须在展开后重新检查父 role 属性的 roles 位图，一直到其中没有任何 role 属性为止。

显然，该操作应该在 expand 过程之前的 link 过程完成。在 link 过程的最后所有 role 属性的 roles 位图最终被确定，此时就可以调用 expand_role_attribute 函数来将子 role 属性的 roles 位图扩展到父 role 属性中了。

```
@@ -1902,8 +1902,9 @@ int define_roleattribute(void)
                return -1;
        }
        r = hashtab_search(policydbp->p_roles.table, id);
-       if (!r || r->flavor != ROLE_ROLE) {
-               yyerror2("unknown role %s, or not a regular role", id);
+       /* We support adding one role attribute into another */
+       if (!r) {
+               yyerror2("unknown role %s", id);
                free(id);
                return -1;
        }
```

显然在 roleattribute 规则中不应该再限制第一个 role 标识符只能为普通 role。

```
+static int expand_role_attributes(hashtab_key_t key, hashtab_datum_t datum, void * data)
+{
+       char *id;
+       role_datum_t *role, *sub_attr;
+       link_state_t *state;
+       unsigned int i;
+       ebitmap_node_t *rnode;
+
+       id = key;
+       role = (role_datum_t *)datum;
+       state = (link_state_t *)data;
+
+       if (strcmp(id, OBJECT_R) == 0){
+               /* object_r is never a role attribute by far */
+               return 0;
+       }
+
+       if (role->flavor != ROLE_ATTRIB)
+               return 0;
+
+       if (state->verbose)
+               INFO(state->handle, "expanding role attribute %s", id);
+
+restart:
+       ebitmap_for_each_bit(&role->roles, rnode, i) {
+               if (ebitmap_node_get_bit(rnode, i)) {
+                       sub_attr = state->base->role_val_to_struct[i];
+                       if (sub_attr->flavor != ROLE_ATTRIB)
```

```
+                               continue;
+
+                       /* remove the sub role attribute from the parent
+                        * role attribute's roles ebitmap */
+                       if (ebitmap_set_bit(&role->roles, i, 0))
+                               return -1;
+
+                       /* loop dependency of role attributes */
+                       if (sub_attr->s.value == role->s.value)
+                               continue;
+
+                       /* now go on to expand a sub role attribute
+                        * by escalating its roles ebitmap */
+                       if (ebitmap_union(&role->roles, &sub_attr->roles)) {
+                               ERR(state->handle, "Out of memory!");
+                               return -1;
+                       }
+
+                       /* sub_attr->roles may contain other role attributes,
+                        * re-scan the parent role attribute's roles ebitmap */
+                       goto restart;
+               }
+       }
+
+       return 0;
+}
+
```

就当前 role 属性 roles 位图中的某一个非 0 位，由 base 模块的 role_val_to_struct 数组得到其 role_datum_t 数据结构。如果为普通 role，则直接跳过。否则为 role 属性，则将其在父 role 属性 roles 位图中的当前非 0 位清除，并合并子 role 属性到 roles 位图。注意如果出现“自包含”现象，则直接将代表子 role 属性的非 0 位清除即可，而无须合并 roles 位图。

最后，由于子 role 属性的 roles 位图中的非 0 位可能比当前位置 i 的数值要小，所以必须重新开始再次扫描父 role 属性的 roles 位图。

另外必须说明的是，get_local_role 函数将 optional_policy 宏修饰的 block/decl 中定义的 role-attribute 关系，保存到了当前 block/decl 的私有符号表而不是模块的全局符号表中！因此在 link 过程的最后处理 role 属性之间的包含关系之前，必须遍历 base 模块所有 block 中当前使能的 decl 的私有符号表，将其中可能存在的 role-attribute 关系复制到 base.p_roles 符号表中！只有这样才能够和 get_local_role 函数的行为相互呼应，否则必定遗漏！

这个光荣的任务由 populate_roleattributes > populate_decl_roleattributes 函数完成：

```
+/* For any role attribute in a declaration's local symtab[SYM_ROLES] table,
+ * copy its roles ebitmap into its duplicate's in the base->p_roles.table.
+ */
+static int populate_decl_roleattributes(hashtab_key_t key, hashtab_datum_t datum, void *data)
+{
+       char *id = key;
+       role_datum_t *decl_role, *base_role;
+       link_state_t *state = (link_state_t *)data;
+
+       decl_role = (role_datum_t *)datum;
+
```

```
+	if (strcmp(id, OBJECT_R) == 0) {
+		/* object_r is never a role attribute by far */
+		return 0;
+	}
+
+	if (decl_role->flavor != ROLE_ATTRIB)
+		return 0;
+
+	base_role = (role_datum_t *)hashtab_search(state->base->p_roles.table, id);
+	assert(base_role != NULL && base_role->flavor == ROLE_ATTRIB);
+
+	if (ebitmap_union(&base_role->roles, &decl_role->roles)) {
+		ERR(state->handle, "Out of memory!");
+		return -1;
+	}
+
+	return 0;
+}
+
+static int populate_roleattributes(link_state_t *state, policydb_t *pol)
+{
+	avrule_block_t *block;
+	avrule_decl_t *decl;
+
+	if (state->verbose)
+		INFO(state->handle, "Populating role-attribute relationship "
+				"from enabled declarations' local symtab.");
+
+	/* Iterate through all of the blocks skipping the first(which is the
+	 * global block, is required to be present and can't have an else).
+	 * If the block is disabled or not having an enabled decl, skip it.
+	 */
+	for (block = pol->global->next; block != NULL; block = block->next)
+	{
+		decl = block->enabled;
+		if (decl == NULL || decl->enabled == 0)
+			continue;
+
+		if (hashtab_map(decl->symtab[SYM_ROLES].table, populate_decl_roleattributes, state))
+			return -1;
+	}
+
+	return 0;
+}
+
```

由于 global block 一定有效，而且没有 else 分支，所以从其后下一个 block 开始。就当前 block 如果存在有效的 decl，则调用 populate_decl_roleattributes 函数处理该 decl 的私有 symtab[SYM_ROLES]符号表，将其中 role 属性的 roles 位图合并到 base.p_roles 符号表中对应元素的位图中。

```
@@ -2455,6 +2571,22 @@ int link_modules(sepol_handle_t * handle,
 		goto cleanup;
 	}
+
+	/* Now that all role attribute's roles ebitmap have been settled,
+	 * escalate sub role attribute's roles ebitmap into that of parent.
+	 *
+	 * First, since some role-attribute relationships could be recorded
```

```
+        * in some decl's local symtab(see get_local_role()), we need to
+        * populate them up to the base.p_roles table. */
+       if (populate_roleattributes(&state, state.base)) {
+               retval = SEPOL_EREQ;
+               goto cleanup;
+       }
+
+       /* Now do the escalation. */
+       if (hashtab_map(state.base->p_roles.table, expand_role_attributes, &state))
+               goto cleanup;
+
```

正如上文所言，在 link 过程的最后就 base.p_roles 符号表调用 expand_role_attributes 函数，但是实现必须处理所有有效的 block/decl 中可能记录的 role-attributes 关系！

9.4.5 测试结果

1．根据 Chris 提出的 role 属性的使用模型，他期望通过 role 属性解决“chain of run interface”的问题。比如 sysadm_t -> rpm_t，而 rpm_script_t -> semanage_t，而后者又能够进入 load_policy_t 和 setfiles_t。也就是说，没有必要许可 sysadm_t 直接进入 load_policy_t 或 setfiles_t。

```
 interface(`rpm_run',`
        gen_require(`
-               type rpm_t, rpm_script_t;
+               attribute_role rpm_roles;
        ')

        rpm_domtrans($1)
-       role $2 types { rpm_t rpm_script_t };
-       seutil_run_loadpolicy(rpm_script_t, $2)
-       seutil_run_semanage(rpm_script_t, $2)
-       seutil_run_setfiles(rpm_script_t, $2)
+       roleattribute $2 rpm_roles;
 ')
```

所以可以在 rpm_run 接口中去掉对 seutil_run_xxx 接口的调用，而仅仅许可$1 到 rpm_t 的转换，并利用 roleattribute 语法将$2 加入 rpm_roles 属性。

另外，在 rpm.te 中定义 rpm_roles 属性，通过 role-types 规则使得它能够和 rpm_t 以及 rpm_script_t 相结合，并且直接在 rpm.te（而不是 rpm.if）中调用 seutil_run_semanage 接口。注意还需要删除就 rpm_script_t 调用的 seutil_domtrans_xxx 接口（从而使得 rpm_script_t 只能进入 semanage_t，而后者才能够进入 setfiles_t 和 load_policy_t）：

```
-seutil_domtrans_loadpolicy(rpm_script_t)
-seutil_domtrans_setfiles(rpm_script_t)
-seutil_domtrans_semanage(rpm_script_t)

+# Test: add one role attribute into another
+attribute_role rpm_roles;
+role rpm_roles types { rpm_t rpm_script_t };
+seutil_run_semanage(rpm_script_t, rpm_roles)
```

按照同样的方法修改 selinuxutil.if 和.te 文件：

```
 interface(`seutil_run_semanage',`
        gen_require(`
```

```
-               type semanage_t;
+               attribute_role semanage_roles;
        ')

        seutil_domtrans_semanage($1)
-       seutil_run_setfiles(semanage_t, $2)
-       seutil_run_loadpolicy(semanage_t, $2)
-       role $2 types semanage_t;
+       roleattribute $2 semanage_roles;
 ')

+role semanage_roles types { semanage_t setfiles_t load_policy_t };
+seutil_run_setfiles(semanage_t, semanage_roles)
+seutil_run_loadpolicy(semanage_t, semanage_roles)
```

在 selinuxutil.te 中定义 semanage_roles 属性，许可它和 semanage_t，setfiles_t 和 load_policy_t 相结合，并且直接调用 seutil_run_xxx 接口使能从 semanage_t 到 setfiles_t 和 load_policy_t 的转换。而在 selinux_run_semanage 中就不再需要调用 seutil_run_xxx 接口了，而是通过 roleattribute 将$2 加入 semanage_roles 属性。

```
+role test_r;
+userdom_unpriv_user_template(test)
+optional_policy(`
+       rpm_run(test_t, test_r)
+')
+
```

最后在 unprivuser.te 中新定义一个 test_r，就它调用 rpm_run 接口。这样就可以验证 test_t -> rpm_t -> rpm_script_t -> semange_t -> setfiles_t，而 test_t 无法直接进入 semanage_t/setfiles_t，并且 test_r 能够和上述所有 type 相结合。

2，修改 refpolicy 后编译 policy.X，在 semanage.conf 中指定 policy-version = 24（从而和 Target 上的 libsepol/setools 所支持的最大版本号一致），观察 test_t/test_r 的特性：

2.1 验证 test_t 能够进入 rpm_t，但是无法直接进入 rpm_script_t，semanage_t，load_policy_t/setfiles_t：

```
sh-3.2# sesearch -SCA -s test_t -t rpm_t -c process -p transition
Found 1 semantic av rules:
   allow test_t rpm_t : process transition ;

sh-3.2# sesearch -SCA -s test_t -t rpm_script_t -c process -p transition

sh-3.2# sesearch -SCA -s test_t -t semanage_t -c process -p transition

sh-3.2# sesearch -SCA -s test_t -t load_policy_t -c process -p transition

sh-3.2# sesearch -SCA -s test_t -t setfiles_t -c process -p transition
```

2.2 rpm_t 能够进入 rpm_script_t，但是无法直接进入 semanage_t，load_policy_t/setfiles_t：

```
sh-3.2# sesearch -SCA -s rpm_t -t rpm_script_t -c process -p transition
Found 1 semantic av rules:
   allow rpm_t rpm_script_t : process transition ;

sh-3.2# sesearch -SCA -s rpm_t -t semanage_t -c process -p transition
```

```
sh-3.2# sesearch -SCA -s rpm_t -t load_policy_t -c process -p transition

sh-3.2# sesearch -SCA -s rpm_t -t setfiles_t -c process -p transition
```

2.3 rpm_script_t能够进入semanage_t，但是无法直接进入load_policy_t/setfiles_t:

```
sh-3.2# sesearch -SCA -s rpm_script_t -t semanage_t -c process -p transition
Found 1 semantic av rules:
   allow rpm_script_t semanage_t : process transition ;

sh-3.2# sesearch -SCA -s rpm_script_t -t load_policy_t -c process -p transition

sh-3.2# sesearch -SCA -s rpm_script_t -t setfiles_t -c process -p transition
```

2.4 semanage_t能够直接进入load_policy_t & setfiles_t:

```
sh-3.2# sesearch -SCA -s semanage_t -t load_policy_t -c process -p transition
Found 1 semantic av rules:
   allow semanage_t load_policy_t : process transition ;

sh-3.2# sesearch -SCA -s semanage_t -t setfiles_t -c process -p transition
Found 1 semantic av rules:
   allow semanage_t setfiles_t : process transition ;
```

2.5. test_r能够和rpm_t，rpm_script_t，semanage_t，setfiles_t ，load_policy_t相结合:

```
sh-3.2# compute_create root:test_r:test_t:s0 system_u:object_r:rpm_exec_t:s0 process
root:test_r:rpm_t:s0
sh-3.2#

sh-3.2# compute_create root:test_r:rpm_script_t:s0 system_u:object_r:semanage_exec_t:s0 process
root:test_r:semanage_t:s0
sh-3.2#

sh-3.2# compute_create root:test_r:semanage_t:s0 system_u:object_r:setfiles_exec_t:s0 process
root:test_r:setfiles_t:s0
sh-3.2#

sh-3.2# compute_create root:test_r:semanage_t:s0 system_u:object_r:load_policy_exec_t:s0 process
root:test_r:load_policy_t:s0
sh-3.2#
```

3，使用Host上的apol工具，观察test_r的属性（注意也必须指定policy-version = 24，因为当前apol只能打开policy.24，而无法打开policy.26）：

```
test_r (28 types)
    chfn_t
    chkpwd_t
    consoletype_t
    ddclient_t
    dhcpc_t
    hostname_t
    ifconfig_t
    insmod_t
    iptables_t
    load_policy_t
    loadkeys_t
```

```
    netutils_t
    newrole_t
    pam_t
    passwd_t
    ping_t
    pppd_t
    pptp_t
    rpm_script_t
    rpm_t
    semanage_t
    setfiles_t
    test_t
    traceroute_t
    updpwd_t
    user_home_t
    usernetctl_t
    utempter_t

rpm_roles (2 types)
    rpm_script_t
    rpm_t

semanage_roles (3 types)
    load_policy_t
    semanage_t
    setfiles_t
```

4，观察 test_r 的二进制表示

4.1 首先得到相关标识符的 policy value：

```
0047a40: 7406 0000 0024 0a00 0001 0000 0000 0000  t....$..........
0047a50: 0074 6573 745f 7407 0000 0025 0a00 0001  .test_t....%....

test_t: policy value = 0xa24

0036560: 0000 004a 0300 0001 0000 0000 0000 0072  ...J...........r
0036570: 706d 5f74 0700 0000 4b03 0000 0100 0000  pm_t....K.......

rpm_t: policy value = 0x34a

0041050: 0000 6d6f 6e6f 7064 5f65 7463 5f74 0c00  ..monopd_etc_t..
0041060: 0000 8907 0000 0100 0000 0000 0000 7270  ..............rp
0041070: 6d5f 7363 7269 7074 5f74 0f00 0000 8a07  m_script_t......

rpm_script_t: policy value = 0x789

004d800: 7365 6375 7269 7479 5f74 0a00 0000 490c  security_t....I.
004d810: 0000 0100 0000 0000 0000 7365 6d61 6e61  ..........semana
004d820: 6765 5f74 0900 0000 4a0c 0000 0300 0000  ge_t....J.......

semanage_t: policy value = 0xc49

00492c0: 7075 745f 7865 7665 6e74 5f74 0d00 0000  put_xevent_t....
00492d0: ae0a 0000 0100 0000 0000 0000 6c6f 6164  ............load
00492e0: 5f70 6f6c 6963 795f 740c 0000 00af 0a00  _policy_t.......

load_policy_t: policy value = 0xaae
```

```
004d660: 740a 0000 003f 0c00 0001 0000 0000 0000  t....?..........
004d670: 0073 6574 6669 6c65 735f 7414 0000 0010  .setfiles_t.....

setfiles_t: policy value = 0xc3f
```

4.2 再观察 test_r 的 types.types 位图中是否包含了上述标识符:

```
002d050: 0600 0000 0000 0000 7465 7374 5f72 4000  ........test_r@.
002d060: 0000 4000 0000 0100 0000 0000 0000 2000  ..@........... .
002d070: 0000 0000 0000 4000 0000 800c 0000 1400  ......@.........
002d080: 0000 8000 0000 0000 0000 0400 0000 4001  ..............@.
002d090: 0000 0000 0000 0001 0000 0002 0000 0000  ................
002d0a0: 0000 0000 0001 4002 0000 0000 0000 0010  ......@.........
002d0b0: 0000 8002 0000 0000 0040 0000 0000 0003  .........@......
002d0c0: 0000 0000 0004 0000 0030, 4003 0000 0002  .........0@.....
002d0d0: 0000 0000 0000, c003 0000 0000 0000 0000  ................
002d0e0: 0080 0004 0000 0000 0000 0000 0008 8005  ................
002d0f0: 0000 0000 0008 0000 0000 4006 0000 1000  ..........@.....
002d100: 0000 0000 0000 8006 0000 0200 0000 0000  ................
002d110: 0000, 8007 0000 0001 0000 0000 0000 8009  ................
002d120: 0000 0000 0210 0000 0410 c009 0000 0000  ................
002d130: 0100 0000 0000, 000a 0000 3000 0000 0800  ..........0.....
002d140: 0000, 800a 0000 0000 0000 0020 0000, 000b  ........... ....
002d150: 0000 0008 0000 0000 0000, 000c 0000 0000  ................
002d160: 0000 0000 0040, 400c 0000 0001 0000 0020  .....@@.........
002d170: 0000

   test_r: policy value = 0x06
     dominates:
            mz = 0x40, highbit = 0x40, node = 1
            startbit = 0, map: 2000 0000 0000 0000
                    policy value: 0x06(test_r)
     types.types:
            mz = 0x40, highbit = 0xc80, node = 0x14
            ......
            startbit = 0x340, map: 0002 0000 0000 0000
                    policy value: 0x34a(rpm_t)
            ......
            startbit = 0x780, map: 0001 0000 0000 0000
                    policy value: 0x789(rpm_script_t)
            ......
            startbit = 0xa00, map: 3000 0000 0800 0000
                    policy value: 0xa01, 0xa02, 0xa24(test_t)
            startbit = 0xa80, map: 0000 0000 0020 0000
                    policy value: 0xaae(load_policy_t)
            startbit = 0xc00, map: 0000 0000 0000 0040
                    policy value: 0xc3f(setfiles_t)
            startbit = 0xc40, map: 0001 0000 0020 0000
                    policy value: 0xc49(semanage_t), 0xc6e
```

5, 额外的 role 属性间“循环依赖”的测试。

6.1 当 semanage_roles 属性和 rpm_roles 属性之间没有循环依赖时, 从属于 semanage_roles 属性的普通 role, 比如 secadm_r, 是无法和 rpm_t 或 rpm_script_t 相结合的（rpm_roles 属性能够和它们相结合）:

```
002cb60: 0000 0a00 0000 0000 0000 7365 6361 646d  ..........secadm
```

```
002cb70: 5f72 4000 0000 4000 0000 0100 0000 0000  _r@...@.........
002cb80: 0000 0002 0000 0000 0000 4000 0000 400d  ..........@...@.
002cb90: 0000 1900 0000 8000 0000 0000 0000 0200  ................
...
002cbe0: 0000 0400 0200 0000 1800, 4003 0000 0002  ..........@.....
002cbf0: 0000 0000 0000 c003 0000 0000 0000 0000  ................
...
002cc30: 0800 4006 0000 0000 0000 0000 0100, 4007  ..@...........@.
002cc40: 0000 0000 0000 0000 2000 4009 0000 0000  ........ .@.....
002cc50: 0000 0000 0420 8009 0000 0000 0820 0000  ..... ....... ..

secadm_r:
     types.types:
             mz = 0x40, highbit = 0xd40, node = 0x19
             startbit = 0x340, map: 0002 0000 0000 0000
                     policy value: 34a, ...

             startbit = 0x740, map: 0000 0000 0000 2000
                     policy value: 776
```

6.2 一旦使用下面的规则：

```
roleattribute semanage_roles rpm_roles;
```

在 semanage_roles 属性和 rpm_roles 属性之间建立其循环依赖，则 rpm_roles 属性的能力，也将被广播到从属于 semanage_roles 属性的所有普通 role 中。此时 secadm_r 就可以和 rpm_t 以及 rpm_script_t 相结合了：

```
002cb60: 0000 0a00 0000 0000 0000 7365 6361 646d  ..........secadm
002cb70: 5f72 4000 0000 4000 0000 0100 0000 0000  _r@...@.........
002cb80: 0000 0002 0000 0000 0000 4000 0000 400d  ..........@...@.
002cb90: 0000 1900 0000 8000 0000 0000 0000 0200  ................
...
002cbe0: 0000 0400 0200 0000 1800, 4003 0000 0102  ..........@.....
002cbf0: 0000 0000 0000 c003 0000 0000 0000 0000  ................
...
002cc30: 0800 4006 0000 0000 0000 0000 0100, 4007  ..@...........@.
002cc40: 0000 0000 0000 0000 3000 4009 0000 0000  ........0.@.....
002cc50: 0000 0000 0420 8009 0000 0000 0820 0000  ..... ....... ..

secadm_r:
     types.types:
             mz = 0x40, highbit = 0xd40, node = 0x19
             startbit = 0x340, map: 0102 0000 0000 0000
                     policy value: 341(rpm_t), ...

             startbit = 0x740, map: 0000 0000 0000 3000
                     policy value: 775(rpm_script_t), 776
```

9.4.6 一个有意思的编译问题

在上面小节提供的 refpolicy 的 debug patch 中，在 selinuxutil.te 中第一次定义 semanage_roles 属性，而在其他模块比如 likewise.te 中调用 seutil_run_semanage 接口。

以模块方式编译没有任何问题，但是如果以 Monolithic 方式编译则遇到如下错误：

```
/usr/bin/checkpolicy -M -U allow policy.conf -o policy.26
/usr/bin/checkpolicy:  loading policy configuration from policy.conf
policy/modules/services/likewise.te":140:ERROR 'role attribute semanage_roles is not declared' at token ';'
on line 1494287:
#line 140
        roleattribute system_r semanage_roles;
checkpolicy:  error(s) encountered while parsing configuration
```

上述错误信息“role attribute semanage_roles is not declared”是由 define_roleattribute 函数在 hashtab_search(policydbp->p_roles.table, id)返回 NULL 时打印的，由此可见此时 semanage_roles 标识符尚未注册到 p_roles 符号表中。

奇怪！在 seutil_run_semanage 接口的实现中分明使用 gen_require 宏来申明对 semanage_roles 的外部依赖了呀！

打开 policy.conf，在 1494287 行果然是关于 semanage_roles 的 roleattribute 规则，发现在 likewise.te 中调用的 seutil_run_semanage 接口展开后没有包含声明 semanage_roles 为外部依赖的 require 关键字：

```
##### begin seutil_run_semanage(lsassd_t,system_r) depth: 1
#line 140

#line 140

#line 140

#line 140

#line 140

#line 140

#line 140

#line 140

#line 140
##### begin seutil_domtrans_semanage(lsassd_t) depth: 2
```

如果以模块方式编译 likewise.pp，则检查 tmp/likewise.tmp 中的相关内容如下：

```
##### begin seutil_run_semanage(lsassd_t,system_r) depth: 1
#line 140

#line 140

#line 140

#line 140
                require {
#line 140

#line 140
                attribute_role semanage_roles;
#line 140
```

```
#line 140
                } # end require
#line 140

#line 140

#line 140

#line 140

#line 140
##### begin seutil_domtrans_semanage(lsassd_t) depth: 2
```

所以问题的根源就是，seutil_run_semanage 接口中使用的 gen_require 宏在以 Monolithic 方式编译时没有被正确地展开！

为什么会这样?

gen_require 宏的定义如下：

```
define(`gen_require',`
        ifdef(`self_contained_policy',`                         # 以 monolithic 方式编译时
                ifdef(`__in_optional_policy',`                  # 在 optional_policy 宏中使用时
                        require {
                                $1
                        } # end require
                ')
        ',`
                require {
                        $1
                } # end require
        ')
')
```

其中 self_contained_policy 变量在以 Monolithic 方式编译时被传递给 m4，而在以模块方式编译时没有定义。所以，如果以 Monolithic 方式编译，gen_require 宏能否展开为 require 关键字由 __in_optional_policy 变量来决定，而它的定义如下：

```
define(`optional_policy',`
        ifelse(regexp(`$1',`\W'),`-1',`
                refpolicywarn(`deprecated use of module name ($1) as first parameter of optional_policy()
block.')
                optional_policy(shift($*))
        ',`
                optional {``pushdef(`__in_optional_policy')
                        $1
                ifelse(`$2',`',`',`} else {
                        $2
                ')}``popdef(`__in_optional_policy')``ifndef(`__in_optional_policy',` # end optional')
        ')
')
```

由此可见，在 optional_policy 宏修饰的 block 内__in_optional_policy 变量才被定义。所以只有在 optional block 中使用的 gen_require 宏才会被 m4 翻译为 require 关键字。**如果在 global block 内使用，则翻译为空！**

注意上面的结论是针对以 Monolithic 方式编译而言，如果以模块方式编译，则 gen_require 宏总是被翻译为 require 关键字。

这样就可以理解以 Monolithic 方式编译时为什么 policy.conf 文件中 seutil_run_semanage 接口展开后没有包含声明 semanage_roles 为外部依赖的语句了：正是因为该接口在 likewise.te 的 global block 中被调用的，所以翻译为空。（如果加入 optional_block 宏修饰对该接口的调用，则该问题就没有了）

其实这样做是有道理的：所有的规则都集中在 policy.conf 文件中，那么 semanage_roles 宏的定义一定存在。因此造成上述问题的真正原因就只能有一个：在当前 patch 的实现中对 semanage_roles 的引用先于对它的定义！

所以，我们可以把 semanage_roles 和 rpm_roles 宏的定义，由 selinuxutil.te 和 rpm.te 移到 kernel.te 中，从而使得在 policy.conf 中它们的定义早于对它们的引用。同时，在 selinuxutil.te 和 rpm.te 中使用 gen_require 宏来声明对它们的外部依赖，这样就能够使两种编译方式都满足：如果以 Monolithic 方式编译，则 gen_require 宏翻译为空，从而避免在 policy.conf 中重复定义这两个属性；如果以模块方式编译，则总是展开为外部依赖声明！

9.4.7 有关 role-types 规则语法的讨论

在设计 role-attr 规则时我参照了 type-attr 规则的语法：用于声明一个 type，并且可以同时声明它所从属的 type 属性。所以 role-attr 规则用于声明一个普通 role，并且可以同时声明它所从属的 role 属性。

而原来的 role-types 规则不但可以声明一个普通 role，而且还能建立它和若干 type 的联系。为了避免在定义 role 时的二义性，我取消了 role-types 规则声明普通 role 的能力，仅仅让它专注于建立 role 和 types 的联系（否则，对于“role xxx_r;”语法，应该调用 role-types 还是 role-attr 规则呢？）。

其实 role-types 规则存在一个很大的问题，就是引入了**声明和使用的二义性**（“ambiguity between declaration and use”），如果一个普通 role 此前没有被定义过，则该规则会随即定义它（称为“implicit declaration”），而不是要求该 role 之前被明确定义过（称为“explicit declaration”）。这样会导致在人们犯错的时候 checkpolicy 编译器无法及时发现，比如如下错误：

```
role http_t types http_t;
mozilla_run_plugin(mozilla_t, $2)                    #导致 role staff_t types mozilla_t;
seutil_run_semanage(lsassd_t, lsassd_t)
```

避免这种错误的方法，就是在语法中避免“声明和使用的二义性”，即明确要求“先声明，后使用”。比如使用 role 规则定义普通 role，使用 attribute_role 规则定义 role 属性；而在 role-types 规则和 roleattribute 规则中都要求所有涉及的普通 role 或者 role 属性都已经被定义。这样除非人们故意定义了：

```
role http_t;
```

否则

```
role http_t types http_t;
```

这种错误就一定会在编译时被及时发现。

目前社区已经采纳了去除 role-types 规则中隐含定义 role 的做法（而放到 role-attr 规则中），接下来就应该进一步去掉 role-attr 和 type-attr 规则中定义和使用的二义性，而将它们简化为 role 和 type 规则，而只能通过 roleattribute 和 typeattribute 规则，建立普通 role/type 和相应属性之间的联系。

9.5 区分 tunable 和 boolean（new）

9.5.1 提出问题 - 无用的 tunable 分支被写入 policy.X

目前 refpolicy 中使用 boolean 来实现 tunable。比如 gen_tunable 宏仍然使用“bool”关键字来定义一个 tunable：

```
define(`gen_tunable',`
	bool $1 dflt_or_overr(`$1'_conf,$2);
')
```

即 tunable 和 boolean 没有分别。相应地，在 tunable_policy 宏定义中，使用 declare_required_symbols 宏来进一步声明对相关 tunable/boolean 的依赖：

```
define(`tunable_policy',`
	gen_require(`
		declare_required_symbols(`$1')
	')
	if (`$1') {
		$2
	ifelse(`$3',``,``,`} else {
		$3
	')}
')

define(`declare_required_symbols',`
ifelse(regexp($1, `\w'), -1, ``, `dn1
bool regexp($1, `\(\w+\)', `\1');
declare_required_symbols(regexp($1, `\w+\(.*\)', `\1'))dn1
') dn1
')
```

由 tunable_policy 宏的展开可见，可以使用一个 tunable/boolean 来控制一个 if-ifelse 结构。如果相应条件表达式的逻辑值为真，则 if 分支的规则有效；如果条件表达式的逻辑值为假，则 ifelse 分支有效。即可以在**运行时**通过改变相应 tunable/boolean 的数值，来改变条件表达式的逻辑值，进而决定哪个分支的规则生效。

为了实现这一点，将 if-ifelse 结构的两个分支上的所有规则（分别组织在 cond_node_t 的 avtrue_list 和 avfalse_list 队列中）展开后写入 te_cond_avtab 哈希表，并且创建指向相应 avtab_key_t 数据结构的索引结构 cond_av_list_t，分别组织在 cond_node_t 的 true_list 和 false_list 队列中。而将 cond_node_t 的条件表达式描述队列，true_list/false_list 队列一并写入 policy.X，并且在内核 policydb_t 中创建相应的描述数据结构。

运行时如果某个 boolean/tunable 的状态值发生改变，一方面可以根据 true_list/false_list 队列中的索引数据结构迅速地定位 te_cond_avtab 哈希表中的 avtab_key_t 数据结构，清除或者设置 specified 域中的 AVTAB_ENABLED 标志位（参见内核函数 security_set_bools > evaluate_cond_node）；另一方面在内核函数 security_compute_sid 中，对于从 te_cond_avtab 哈希表中匹配的规则，只有当其 AVTAB_ENABLED 标志被设置时才使用它。

实际上，真正需要在运行时切换的规则很少，它们应该由使用 boolean 的 if-ifelse 结构来描述。目前在 refpolicy/policy/global_booleans 文件中只定义了 3 个 boolean：

```
gen_bool(secure_mode,false)
gen_bool(secure_mode_insmod,false)
```

gen_bool(secure_mode_policyload,false)

而在 refpolicy/policy/global_tunables 文件中，以及各种.te 中定义的 tunable 却有 150+个。**所有这些 tunable 的逻辑值应该根据所部署的软硬件环境来“一劳永逸”地在编译时确定，而无须在运行时改变，所以相应 if-ifelse 结构中始终只有一个分支的规则有效且总是有效。**

比如 console_login 应该只对需要支持从 console 登录的系统才设置为真，而对于其他的软硬件环境设置为假。所以对于后者而言由 console_login 控制的 if-ifelse 结构就只需将 ifelse 分支中的规则永久地生效（即扩展后加入 te_avtab 哈希表，而不是 te_cond_avtab 哈希表），并直接丢弃 if 分支中的规则。而不是象现在这样也把整个 tunable 的 if-ifelse 结构都加入 te_cond_avtab 并写入 policy.X。

实际上许多 tunable 的 if-ifelse 结构中都没有 ifelse 结构，那么对于默认为假（从而条件表达式为假）的结构而言，应当被完全排除在 policy.X 之外（因为它们不适用于 refpolicy 所部署的软硬件条件），这样可以大大缩小 policy.X 的大小，并显著节约 link/expand 的时间。

9.5.2 分析问题

上面问题的核心内容就是只保留 tunable 的 if-ifelse 结构中实际有效的分支并永久地写入 policy.X，而无效的分支则彻底不写入（即被丢弃）。

1，首先，需要区分 tunable 和 boolean 标识符，从而对各自所控制的 if-ifelse 结构采用不同的策略。

由于 tunable 和 boolean 标识符的本质即为一个布尔变量，所以 tunable 仍然可以使用当前 cond_bool_datum_t 数据结构进行描述，但是需要新增加一个域（标志位）以区分 tunable 和 boolean。

相应地，必须设计一个新的关键字“tunable”用来定义 tunable，而“bool”关键字用于定义 boolean。这两个关键字的词法分析函数（无论 define 还是 require）的处理都大致相同，根据关键字的不同相应地设置/清除上述标志位。

在 gen_tunable 宏中使用这个新的 tunable 关键字，而原有 bool 关键字则仍在 gen_bool 宏中使用。这样，在 global_tunables 文件和各种.te 中使用 gen_tunable 宏定义的标识符，就真正地变成一个 tunable 而不再是 boolean 了。

2，进一步，需要区分 tunable/boolean 的 if-ifelse 结构的定义方式。

当前 tunable_policy 宏将条件表达式中的标识符当作 boolean 声明为外部依赖，现在有了 tunable 标识符的定义，就可以将它们按照 tunable 声明为外部依赖了。另外还需要定义一个新的 boolean_policy 宏，将条件表达式中的标识符当作 boolean 声明为外部依赖。这样一来就可以在相应的 require 函数中确定标识符的属性，设置/清除 cond_bool_datum_t 数据结构中的标志位了。

3，最终，在编译时对 tunable/boolean 的 if-ifelse 结构采用不同的策略。

无论关于 tunable 的或者关于 boolean 的 if-ifelse 结构，都用 cond_node_t 数据结构进行描述，所以可以给该数据结构设计一个域（标志位），以区分 if-ifelse 结构的属性。

对于关于 boolean 的 if-ifelse 结构，当前 toolchain 的处理行为无须改变；而对于关于 tunable 的 if-ifelse 结构，则应该只将其实际有效的分支一劳永逸地使能并写入 policy.X；而对于实际无效的分支，无须展开并写入 policy.X（即被丢弃）。

可以通过两点来实现这一点：

1）首先在 link 过程的最后遍历所有 block 的惟一有效 dec1，检查 cond_list 队列中的每一个 cond_node_t 元素的属性，**如果它为 tunable 的，则根据当前状态值将 avtrue_list 或 avfalse_list 队列中的所有元素，直接加入当前 dec1->avrules 队列的末尾**（并相应地将原来的队列指针 avtrue_list 或 avfalse_list 设置为 NULL，以避免内存泄漏），这样这些规则将在 expand 过程中被展开并注册到 te_avtab 哈希表中；
2）然后在 expand 过程中处理 dec1->cond_list 队列中的每一个 cond_node_t 元素时，如果为 tunable 则直接跳过。

即只有 boolean 的 if-ifelse 结构的两个分支上的规则才按照当前方法处理（展开并注册到 te_cond_avtab 哈希表并最终写入 policy.X），而 tunable 控制的有效分支上的规则，则被当作当前 dec1 非条件规则处理。

cond_node_t 的属性，应该根据其条件表达式中 COND_BOOL 类型分量的属性为 tunable 或 boolean 来决定：如果都是 boolean，则该 cond_node_t 被认为是 boolean 的，上述标志位被清除；如果都是 tunable，则该 cond_node_t 被认为是 tunable 的，上述标志位被设置。我们不允许在同一个条件表达式中混用 tunable 和 boolean。

注意，由于一个 tunable 或 boolean 标识符可能在不同的模块中定义和使用，因此在编译单个模块时无法确定当前标识符的真实属性！比如在模块 A 中将一个标识符 b 使用在了 tunable_policy 宏中，则 b 被当作 tunable 加入模块 A 的 p_bools 符号表，即在编译时被当作 tunable。进一步，在 link 的初期它被拷贝到 base 模块；另外，b 标识符在模块 B 中被 gen_bool 宏定义为 boolean，则在编译 B 模块时无法知道该标识符在其他模块内是如何被使用的。进一步，只有在 link 初期拷贝 B 模块的 p_bools 符号表到 base 模块时，才能最终发现关于 b 标识符属性的冲突定义：当前模块中的属性定义和 base 模块中已有的定义（来自于其他模块）相反。

所以，只有在 link 过程中当合并各个模块的 p_bools 符号表到 base 模块时，才能发现相互冲突的定义和使用；只有在 link 过程中所有模块的 p_bools 符号表都合并完成后，才能最终确定一个 cond_node_t 的属性（而在编译 if-ifelse 结构时，无法确定它的真正属性）。

4，最后，tunable 标识符无须写入 policy.X

由于 policy.X 和内核 policydb_t 数据结构中不再需要关于 tunable 的 cond_node_t 数据结构，所有 tunable 标识符也无须写入 policy.X。可以在 expand 过程中拷贝 base.p_bools 符号表到 out 模块时检查 cond_bool_datum_t 的属性，跳过 tunable 标识符即可。

9.5.3 解决问题

【数据结构】
1，扩展 cond_bool_datum_t 数据结构增加 flags 域，目前只定义一个标志位 COND_BOOL_FLAGS_TUANBLE：

```
typedef struct cond_bool_datum {
        symtab_datum_t s;
        int state;
+#define COND_BOOL_FLAGS_TUNABLE        0x01    /* is this a tunable? */
+       uint32_t flags;
 } cond_bool_datum_t;
```

2，扩展 cond_node_t 数据结构增加 flags 域，目前只定义一个标志位 COND_NODE_FLAGS_TUNABLE：

```
typedef struct cond_node {
```

```
        int cur_state;                                          # 条件表达式当前状态值
        cond_expr_t *expr;                                      # 条件表达式队列
        /* these true/false lists point into te_avtab when that is used */
        cond_av_list_t *true_list;                              # 在 expand 时创建，只写入 policy.X
        cond_av_list_t *false_list;                             # 在 expand 时创建，只写入 policy.X
        /* and these are used during parsing and for modules */
        avrule_t *avtrue_list;                                  # 在编译时创建，只写入 pp
        avrule_t *avfalse_list;                                 # 在编译时创建，只写入 pp
        /* these fields are not written to binary policy */
        unsigned int nbools;
        uint32_t bool_ids[COND_MAX_BOOLS];
        uint32_t expr_pre_comp;
        struct cond_node *next;
+       /* a tunable conditional, calculated and used at expansion */
+#define        COND_NODE_FLAGS_TUNABLE 0x01
+       uint32_t flags;                                          # 只在 expand 过程中计算并使用
} cond_node_t;
```

注意：
1，如上所述，在编译时并不能确定一个 tunable/boolean 标识符真正的属性，声明和使用的冲突只有在 link 时才能发现，因此只有在 link 过程中才能确定一个 cond_node_t 真实的属性。

真正需要记录属性的是 cond_bool_datum_t 数据结构，cond_node_t 的属性根据条件表达式中各个 COND_BOOL 分量在 link 时计算，而无须为条件表达式分量描述符 cond_expr_t 设计 flags 域。

2，无论时用户态还是内核态的 policydb_t 数据结构，都无须改变。写入 policy.X 的 cond_node_t 只需包含关于 boolean 的 true_list/false_list 队列，所以内核态 cond_node_t 数据结构无须改变。

由于内核态 policydb_t 及驱动无须改变，所以只需要增加用户态模块的版本号（用户态 policy.X 的版本也无须改变）：

```
+#define MOD_POLICYDB_VERSION_TUNABLE_SEP      14
+#define MOD_POLICYDB_VERSION_MAX              MOD_POLICYDB_VERSION_TUNABLE_SEP
```

【编译】
首先需要定义 tunable 关键字的词法分析函数以及相应的 require 函数。

由于 tunable/boolean 标识符都使用 cond_bool_datum_t 数据结构，所以它们的编译方法基本相同。惟一不同之处就是对 tunable 类型的标识符设置其 cond_bool_datum_t.flags 中的 TUNABLE 标志位。这一点可以通过给当前的处理函数增加参数 is_tunable 来实现。

将当前 define_bool 函数改名为 define_bool_tunable 并增加参数 is_tunable，该参数由词法分析决定：如果遇到 tunable 关键字则在调用 define_bool_tunable 函数时传递 1；否则如果遇到 boolean 关键字，则传递 0：

```
TUNABLE |
tunable                         { return(TUNABLE); }

te_decl                 : attribute_def
                        | type_def
                        | typealias_def
                        | typeattribute_def
```

```
                        | typebounds_def
                        | bool_def
+                       | tunable_def
                        | transition_def
                        | range_trans_def
                        | te_avtab_def
                        | permissive_def
                        ;

bool_def                : BOOL identifier bool_val ';'
                        { if (define_bool_tunable(0)) return -1; }
                        ;
tunable_def             : TUNABLE identifier bool_val ';'
                        { if (define_bool_tunable(1)) return -1; }
                        ;
```

这样，在 define_bool_tunable 函数中就可以根据参数值决定是否设置当前 cond_bool_datum_t.flags 中的标志位：

```
int define_bool_tunable(int is_tunable)
{
        char *id, *bool_value;
        cond_bool_datum_t *datum;
        int ret;
        uint32_t value;

        if (pass == 2) {
                while ((id = queue_remove(id_queue)))
                        free(id);
                return 0;
        }

        id = (char *)queue_remove(id_queue);
        if (!id) {
                yyerror("no identifier for bool definition?");
                return -1;
        }
        if (id_has_dot(id)) {
                free(id);
                yyerror("boolean identifiers may not contain periods");
                return -1;
        }
        datum = (cond_bool_datum_t *)malloc(sizeof(cond_bool_datum_t));
        if (!datum) {
                yyerror("out of memory");
                free(id);
                return -1;
        }
        memset(datum, 0, sizeof(cond_bool_datum_t));
        if (is_tunable)
                datum->flags |= COND_BOOL_FLAGS_TUNABLE;
        ret = declare_symbol(SYM_BOOLS, id, datum, &value, &value);
        switch (ret) {
        case -3:{
                        yyerror("Out of memory!");
                        goto cleanup;
                }
        case -2:{
```

```
                        yyerror2("duplicate declaration of boolean %s", id);
                        goto cleanup;
                }
        case -1:{
                        yyerror("could not declare boolean here");
                        goto cleanup;
                }
        case 0:
        case 1:{
                        break;
                }
        default:{
                        assert(0);      /* should never get here */
                }
        }
        datum->s.value = value;
        bool_value = (char *)queue_remove(id_queue);
        if (!bool_value) {
                yyerror("no default value for bool definition?");
                free(id);
                return -1;
        }

        datum->state = (int)(bool_value[0] == 'T') ? 1 : 0;
        return 0;
      cleanup:
        cond_destroy_bool(id, datum, NULL);
        return -1;
}
```

注意，由于整个 cond_bool_datum_t 数据结构使用 memset 函数初始化，所以默认情况下并不会设置 TUNABLE 标志，只对 tunable 标识符设置该标志。

另外，由上述代码可见，当前 tunable/boolean 的状态值是由 token queue 后继字符是否为字符“T”来决定。词法分析函数会在缺省定义为 true 时插入字符“T”，在缺省定义为 false 时插入字符“F”。

类似地，将当前 require_bool 函数改名为 require_bool_tunable 并增加参数 is_tunable，在 require 结构中遇到 tunable 关键字时调用新增 require_tunable 函数，它直接调用 require_bool_tunable 函数并传递 1；遇到 boolean 关键字时调用 require_bool 函数，它直接调用 require_bool_tunable 函数并传递 0：

```
require_decl_def        : ROLE          { $$ = require_role; }
                        | TYPE          { $$ = require_type; }
                        | ATTRIBUTE     { $$ = require_attribute; }
                        | ATTRIBUTE_ROLE    { $$ = require_attribute_role; }
                        | USER          { $$ = require_user; }
                        | BOOL          { $$ = require_bool; }
+                       | TUNABLE       { $$ = require_tunable; }
                        | SENSITIVITY   { $$ = require_sens; }
                        | CATEGORY      { $$ = require_cat; }
                        ;

-int require_bool(int pass)
+static int require_bool_tunable(int pass, int is_tunable)
 {
        char *id = queue_remove(id_queue);
        cond_bool_datum_t *booldatum = NULL;
```

```
@@ -1063,6 +1063,8 @@ int require_bool(int pass)
                yyerror("Out of memory!");
                return -1;
        }
+       if (is_tunable)
+               booldatum->flags |= COND_BOOL_FLAGS_TUNABLE;
        retval =
            require_symbol(SYM_BOOLS, id, (hashtab_datum_t *) booldatum,
                           &booldatum->s.value, &booldatum->s.value);
@@ -1094,6 +1096,16 @@ int require_bool(int pass)
        }
 }

+int require_bool(int pass)
+{
+       return require_bool_tunable(pass, 0);
+}
+
+int require_tunable(int pass)
+{
+       return require_bool_tunable(pass, 1);
+}
+
```

和 define_bool_tunable 函数中相同，在 require_bool_tunable 函数中也是根据参数 is_tunable 来决定是否设置 TUNABLE 标志。

值得一提的是，只要 tunable/boolean 标识符被错误地使用在 boolean_policy 或者 tunable_policy 宏中，则在编译当前模块时，它们就会被相应地 require 为 boolean 和 tunable，所以只有等到 link 时拷贝各个模块的 p_bools 符号表时才能发现是否有声明和使用之间的冲突。

【pp 的读写】
对于版本号不小于 MOD_POLICYDB_VERSION_TUNABLE_SEP 的模块，在读写 p_bools 符号表以及 cond_node_t 数据结构时都需要读写新增的 flags 标志位。

对 cond_read_bool 和 cond_write_bool 函数的修改如下：

```
-int cond_read_bool(policydb_t * p
-                   __attribute__ ((unused)), hashtab_t h,
+int cond_read_bool(policydb_t * p,
+                   hashtab_t h,
                    struct policy_file *fp)
 {
        char *key = 0;
@@ -597,6 +597,15 @@ int cond_read_bool(policydb_t * p
        if (rc < 0)
                goto err;
        key[len] = 0;
+
+       if (p->policy_type != POLICY_KERN && p->policyvers >= MOD_POLICYDB_VERSION_TUNABLE_SEP) {
+               rc = next_entry(buf, fp, sizeof(uint32_t));
+               if (rc < 0)
+                       goto err;
+               booldatum->flags = le32_to_cpu(buf[0]);
+       }
+
```

```
        if (hashtab_insert(h, key, booldatum))
                goto err;

@@ -621,6 +622,15 @@ static int cond_write_bool(hashtab_key_t key, hashtab_datum_t datum, void *ptr)
        items = put_entry(key, 1, len, fp);
        if (items != len)
                return POLICYDB_ERROR;
+
+       if (p->policy_type != POLICY_KERN && p->policyvers >= MOD_POLICYDB_VERSION_TUNABLE_SEP) {
+               buf[0] = cpu_to_le32(booldatum->flags);
+               items = put_entry(buf, sizeof(uint32_t), 1, fp);
+               if (items != 1)
+                       return POLICYDB_ERROR;
+       }
+
        return POLICYDB_SUCCESS;
 }
```

类似地，对 cond_read_node 和 cond_write_node 函数的修改如下：

```
@@ -810,6 +819,14 @@ static int cond_read_node(policydb_t * p, cond_node_t * node, void *fp)
                if (avrule_read_list(p, &node->avfalse_list, fp))
                        goto err;
        }
+
+       if (p->policy_type != POLICY_KERN && p->policyvers >= MOD_POLICYDB_VERSION_TUNABLE_SEP) {
+               rc = next_entry(buf, fp, sizeof(uint32_t));
+               if (rc < 0)
+                       goto err;
+               node->flags = le32_to_cpu(buf[0]);
+       }

        return 0;
       err:

@@ -727,6 +737,14 @@ static int cond_write_node(policydb_t * p,
                        return POLICYDB_ERROR;
        }

+       if (p->policy_type != POLICY_KERN && p->policyvers >= MOD_POLICYDB_VERSION_TUNABLE_SEP) {
+               buf[0] = cpu_to_le32(node->flags);
+               items = put_entry(buf, sizeof(uint32_t), 1, fp);
+               if (items != 1)
+                       return POLICYDB_ERROR;
+       }
+
        return POLICYDB_SUCCESS;
 }
```

注意，在创建 policy.X 时不需要写入 tunable 标识符，所以可以在创建 policy.X 时跳过 out->p_bools 符号表中的所有 tunable 标识符。实际上有一个更好的解决办法，即在 expand 过程中根本不从 base->p_bools 符号表中拷贝 tunable 标识符到 out->p_bools 符号表，这样在写出 policy.X 时就无须考虑 tunable 了，当前代码也无须改动。

说明，由后文可知，cond_node_t.flags 中的 TUNABLE 标志位是在 link 之后、expand 之前才最终确定的。所以在读写模块 pp 文件时并不真正需要读写 flags 域，目前仍然保留的原因是为了考虑到将来的方便：将来可能会有其他标志位被定义。

【link 过程】
如上文所述，在 link 过程中逐一合并各个模块的 p_bools 符号表到 base.p_bools 符号表，此时需要拷贝 tunable/boolean 标识符的 flags 域，并检查模块之间是否就一个标识符存在声明/使用上的冲突：

```
static int bool_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        int ret;
        char *id = key, *new_id = NULL;
        cond_bool_datum_t *booldatum, *base_bool, *new_bool = NULL;
        link_state_t *state = (link_state_t *)data;
        scope_datum_t *scope;

        booldatum = (cond_bool_datum_t *)datum;

        base_bool = hashtab_search(state->base->p_bools.table, id);
        if (base_bool == NULL) {                                        # 第一次向 base.p_bools 符号表拷贝
                if (state->verbose)
                        INFO(state->handle, "copying boolean %s", id);

                if ((new_id = strdup(id)) == NULL) {
                        goto cleanup;
                }

                if ((new_bool = (cond_bool_datum_t *)malloc(sizeof(*new_bool))) == NULL) {
                        goto cleanup;
                }
                new_bool->s.value = state->base->p_bools.nprim + 1;

                ret = hashtab_insert(state->base->p_bools.table,
                                     (hashtab_key_t) new_id,
                                     (hashtab_datum_t) new_bool);
                if (ret) {
                        goto cleanup;
                }
                state->base->p_bools.nprim++;
                base_bool = new_bool;
+               base_bool->flags = booldatum->flags;                            # 同时拷贝 flags 标志
+       } else if ((booldatum->flags & COND_BOOL_FLAGS_TUNABLE) !=
+                  (base_bool->flags & COND_BOOL_FLAGS_TUNABLE)) {
+                       /* A mismatch between boolean/tunable declaration
+                        * and usage(for example a boolean used in the
+                        * tunable_policy() or vice versa).
+                        *
+                        * This is not allowed and bail out with errors */
+                       ERR(state->handle,
+                           "%s: Mismatch between boolean/tunable definition "
+                           "and usage for %s", state->cur_mod_name, id);
+                       return -1;
        }
```

如果当前模块中该标识符的 TUNABLE 标志的设置情况和 base.p_bools 符号表中已有的不一样，则说明在先前已经完成拷贝的其他模块中按照相反的方式使用了该标识符，比如将 tunable/boolean 标识符在 boolean_policy/tunable_policy 中使用。此时直接报错退出。

```
        /* Get the scope info for this boolean to see if this is the declaration,
```

```
         * if so set the state */
        scope = hashtab_search(state->cur->policy->p_bools_scope.table, id);
        if (!scope)
                return SEPOL_ERR;
        if (scope->scope == SCOPE_DECL) {
                base_bool->state = booldatum->state;
+               /* Only the declaration rather than requirement
+                * decides if it is a boolean or tunable. */
+               base_bool->flags = booldatum->flags;
        }
```

如果当前模块为该标识符真正的定义者（注意除了 role/user 其他类标识符只允许被定义一次），则该标识符的属性由该定义者模块决定。

```
        state->cur->map[SYM_BOOLS][booldatum->s.value - 1] = base_bool->s.value;
        return 0;

      cleanup:
        ERR(state->handle, "Out of memory!");
        cond_destroy_bool(new_id, new_bool, NULL);
        return -1;
}
```

【expand 过程】
如上所述，在 link 过程中拷贝完成所有模块的符号表以及 block/decl 到 base 模块后，就可以遍历 base 模块所有 block 链表，就每一个 block 惟一的 decl，逐一处理其 decl->cond_list 队列中的每一个 cond_node_t 元素，如果当前为 tunable，则将有效的分支的规则直接加入当前 decl->avrules 队列的末尾，并将原有指针设置为 NULL。

上述操作由 discard_tunables 函数完成。注意它需要修改 decl->avrules 队列以及 cond_node_t 数据结构中的 avtrue_list 或 avfalse_list 指针。由后文可知，为了维护 link 过程结束后得到的 link.pp 的完整性，特此将该函数的调用处由 link 过程的后部移到了 expand 过程的最开始。

```
@@ -2678,6 +2766,16 @@ int expand_module(sepol_handle_t * handle,
        expand_state_t state;
        avrule_block_t *curblock;

+       /* Append tunable's avtrue_list or avfalse_list to the avrules list
+        * of its home decl depending on its state value, so that the effect
+        * rules of a tunable would be added to te_avtab permanently. Whereas
+        * the disabled unused branch would be discarded.
+        *
+        * Originally this function is called at the very end of link phase,
+        * however, we need to keep the linked policy intact for analysis
+        * purpose. */
+       discard_tunables(base);
+
        expand_state_init(&state);

        state.verbose = verbose;
```

discard_tunables 函数的实现如下：

```
+static void discard_tunables(policydb_t *pol)
+{
```

```
+       avrule_block_t *block;
+       avrule_decl_t *decl;
+       cond_node_t *cur_node;
+       cond_expr_t *cur_expr;
+       int cur_state;
+       avrule_t *tail, *to_be_appended;
+
+       /* Iterate through all cond_node of all enabled decls, if a cond_node
+        * is about tunable, calculate its state value and concatenate one of
+        * its avrule list to the current decl->avrules list.
+        *
+        * Note, such tunable cond_node would be skipped over in expansion,
+        * so we won't have to worry about removing it from decl->cond_list
+        * here :-)
+        *
+        * If tunables and booleans co-exist in the expression of a cond_node,
+        * then tunables would be "transformed" as booleans.
+        */
+       for (block = pol->global; block != NULL; block = block->next) {
+               decl = block->enabled;
+               if (decl == NULL || decl->enabled == 0)
+                       continue;
+
+               tail = decl->avrules;
+               while (tail && tail->next)
+                       tail = tail->next;
+
+               for (cur_node = decl->cond_list; cur_node != NULL; cur_node = cur_node->next) {
```

该函数的主体结构包含两个循环：外层循环逐一遍历 base->gloabl 链表上每个 block 的惟一有效 decl；内层循环注意遍历当前 decl->cond_list 队列中的每个 cond_node_t 元素。

注意进入内层循环之前，tail 都指向当前 decl->avrules 队列的末尾。

```
+                       int booleans, tunables;
+                       cond_bool_datum_t *booldatum;
+
+                       booleans = tunables = 0;
+
+                       for (cur_expr = cur_node->expr; cur_expr != NULL; cur_expr = cur_expr->next) {
+                               if (cur_expr->expr_type != COND_BOOL)
+                                       continue;
+                               booldatum = pol->bool_val_to_struct[cur_expr->bool - 1];
+                               if (booldatum->flags & COND_BOOL_FLAGS_TUNABLE)
+                                       tunables++;
+                               else
+                                       booleans++;
+                       }
+
```

就当前 cond_node_t 元素，首先根据其条件表达式描述队列中 COND_BOOL 元素的属性，确定该 cond_node_t 的属性。注意此时直接可以从 base 模块的 bool_val_to_struct[]数组中获得 tunable/boolean 标识符的 cond_bool_datum_t 数据结构的指针。

这里就 tunable/boolean 分别计数，理论上在 link 过程的 bool_copy_callback 函数中已经能够发现有关 tunable/boolean 在声明/使用上的不一致性，所以两个计数器中应该有且只有一个大于 0，可以使用 assert 宏以确认这一点。

注意，每次内层循环开始都需要将 tunable/boolean 计数器归零。

```
+                       /* bool_copy_callback() at link phase has ensured
+                        * that no mixture of tunables and booleans in one
+                        * expression. */
+                       assert(!(booleans && tunables));
+
```

对于 boolean 控制的 if-else 结构，清除 TUNABLE 标志；而对于 tunable 控制的 if-else 结构，则设置该标志：

```
+                       if (booleans) {
+                               cur_node->flags &= ~COND_NODE_FLAGS_TUNABLE;
+                       } else {
+                               cur_node->flags |= COND_NODE_FLAGS_TUNABLE;
+                               cur_state = cond_evaluate_expr(pol, cur_node->expr);
+                               if (cur_state == -1) {
+                                       printf("Expression result was undefined, skipping all rules\n");
+                                       continue;
+                               }
+
+                               to_be_appended = (cur_state == 1) ?
+                                       cur_node->avtrue_list : cur_node->avfalse_list;
+
```

对于 tunable 的 if-else 结构，计算其条件表达式的状态值，如果为 1，则将 avtrue_list 队列中的元素加入 decl->avrules 队列的末尾；如果为 0，则是 avfalse_list 队列。

```
+                               if (tail)
+                                       tail->next = to_be_appended;
+                               else
+                                       tail = decl->avrules = to_be_appended;
+
+                               /* Now that the effective branch has been
+                                * appended, neutralize its original pointer */
+                               if (cur_state == 1)
+                                       cur_node->avtrue_list = NULL;
+                               else
+                                       cur_node->avfalse_list = NULL;
+
+                               /* Update the tail of decl->avrules for
+                                * further concatenation */
+                               while (tail && tail->next)
+                                       tail = tail->next;
```

如上文所述，在进入内层循环处理当前 decl->cond_list 队列之前，tail 就已经指向当前 decl->avrules 队列的末尾了。直接将当前 cond_node_t 的 avtrue_list 或者 avfalse_list 队列中的元素加入 tail->next 即可。注意处理 tail 此时为 NULL 的情况，并且相应地将 avtrue_list 或者 avfalse_list 队列的指针清空（这样逻辑上才完整，正确）。最后还必须更新 tail 指针指向新的末尾元素。

```
+                       }
+               }
+       }
+}
+
```

另一方面，在 expand 过程中展开每个 dec1->cond_list 队列中的每个 cond_node_t 中的规则时，由于 discard_tunables 函数已经将 tunable 实际有效分支中的规则加入到了 dec1->avrules 队列（从而使得它们被当作该 dec1 的非条件规则处理），所以在此时在 cond_node_copy 函数中应该直接跳过所有 tunable 的 cond_node_t 元素。

并且不需要从 base 模块向 out 模块拷贝 tunable 标识符。

```
@@ -1014,6 +1014,11 @@ static int bool_copy_callback(hashtab_key_t key, hashtab_datum_t datum,
                return 0;
        }

+       if (bool->flags & COND_BOOL_FLAGS_TUNABLE) {
+               /* Skip tunables */
+               return 0;
+       }
+
        if (state->verbose)
                INFO(state->handle, "copying boolean %s", id);
@@ -1046,6 +1051,7 @@ static int bool_copy_callback(hashtab_key_t key, hashtab_datum_t datum,
        state->boolmap[bool->s.value - 1] = new_bool->s.value;

        new_bool->state = bool->state;
+       new_bool->flags = bool->flags;                  # 仍然保留拷贝 booleans 的 flags 域（将来或许有用）

        return 0;
 }
@@ -1940,6 +1946,13 @@ static int cond_node_copy(expand_state_t * state, cond_node_t * cn)
        if (cond_node_copy(state, cn->next)) {
                return -1;
        }
+
+       /* If current cond_node_t is of tunable, its effective branch
+        * has been appended to its home dec1->avrules list during link
+        * and now we should just skip it. */
+       if (cn->flags & COND_NODE_FLAGS_TUNABLE)
+               return 0;
+
        if (cond_normalize_expr(state->base, cn)) {
                ERR(state->handle, "Error while normalizing conditional");
                return -1;
```

【保留所有的 tunable】
至此，我们已经实现只把实际有效的 tunable 分支中的规则，永久地写入 policy.X 并生效，而且 policy.X 中不再包含任何 tunable 标识符，以及和 tunable 相关的 cond_node_t 数据结构。

根据 Dan Walsh 的反馈，在调试 SELinux 规则库时通常会使用 audit2allow/audit2why 工具，它们会提示所缺少的规则和哪一个 tunable/boolean 相关，从而提示系统管理员应该使能相应的 tunable/boolean。所以，如果象现在这样永久地从 policy.X 中去掉所有的 tunable，则显然对 SELinux 的调试不利。

为了支持对 SELinux 的调试，即在 policy.X 中保留当前所有的 tunable 以及它们所控制的 if-else 结构，特意给 semodule 程序设计了一个新的选项“-P/--preserve_tunables”，如果在创建 module store 时该选项被设置，则 semodule 程序将调用 libsemanage/libsepol 中的相应函数，设置 sepol_handle_t 数据结构中的 preserve_tunables 标志位。该标志位默认为 0（表示不需要在 policy.X 中保留 tunable），只有在对 semodule 程序显式地指定“-P”选项时才被设置为 1。

sepol_handle_t 数据结构在使用 semodule 命令创建 module store 时被传递给 link 和 expand 过程（分别通过 semanage_link_sandbox 和 semanage_expand_sandbox 函数，而它们都被 semodule > semanage_commit > semanage_direct_commit 函数调用），所以在 discard_tunables 函数中就可以检查 preserve_tunables 标志位，根据其是否被设置来决定丢弃或保留所有的 tunable。修改后的代码如下：

```
static void discard_tunables(sepol_handle_t *sh, policydb_t *pol)
{
......
        int preserve_tunables = 0;

        if (sh && sh->preserve_tunables)
                preserve_tunables = 1;
```

首先给 discard_tunables 函数增加参数 sh 指向调用者创建的 sepol_handle_t 数据结构。根据其中 preserve_tunables 域的值来相应地设置本地变量 preserve_tunables。注意，link/expand 过程可能在 checkmodule/checkpolicy 命令中被使用，而此时并不需要 semanage_handle_t/sepol_handle_t 数据结构，所以必须判断参数 sh 是否为 NULL。

```
        for (block = pol->global; block != NULL; block = block->next) {
                decl = block->enabled;
                if (decl == NULL || decl->enabled == 0)
                        continue;

                tail = decl->avrules;
                while (tail && tail->next)
                        tail = tail->next;

                for (cur_node = decl->cond_list; cur_node != NULL; cur_node = cur_node->next) {
                        int booleans, tunables, i;
                        cond_bool_datum_t *booldatum;
                        cond_bool_datum_t *tmp[COND_EXPR_MAXDEPTH];

                        booleans = tunables = 0;
                        memset(tmp, 0, sizeof(cond_bool_datum_t *) * COND_EXPR_MAXDEPTH);

                        for (cur_expr = cur_node->expr; cur_expr != NULL; cur_expr = cur_expr->next) {
                                if (cur_expr->expr_type != COND_BOOL)
                                        continue;
                                booldatum = pol->bool_val_to_struct[cur_expr->bool - 1];
                                if (booldatum->flags & COND_BOOL_FLAGS_TUNABLE)
                                        tmp[tunables++] = booldatum;
                                else
                                        booleans++;
                        }
```

在通过 COND_BOOL 类型分量来判断一个条件表达式的属性（即一个 cond_node_t 的属性）时，除了 tunables 计数器之外，还使用了一个最多容纳 COND_EXPR_MAXDEPTH 个元素的指针数组，用于缓存所有 tunable 标识符的 cond_bool_datum_t 数据结构的指针。注意可以直接从 base 模块的 bool_val_to_struct[]数组获取 tunable 标识符的 cond_bool_datum_t 数据结构。

```
                        /* bool_copy_callback() at link phase has ensured
                         * that no mixture of tunables and booleans in one
                         * expression. However, this would be broken by the
                         * request to preserve tunables */
                        if (!preserve_tunables)
                                assert(!(booleans && tunables));
```

原本在 link 过程中（确切地说在 bool_copy_callback 函数中）已经禁止 tunable/boolean 相互混用，所以才可以使用 assert 宏来再次确认这一点。由于下面将在需要保留所有的 tunable 标识符时清除其 TUNABLE 标志，这将违反 assert 宏所检查的条件，所以此时必须跳过 assert 检查（比如，当需要保留 tunable 时当前 dec1 中的所有 tunable 的 TUNABLE 标志位被清除，而它和后继 dec1 中的其他 tunable 一起使用，则在后继 dec1 处理过程中该 tunable 就会被当作 boolean，于是被认为发生了 tunable/boolean 混用的情况）。

```
                if (booleans || preserve_tunables) {
                        cur_node->flags &= ~COND_NODE_FLAGS_TUNABLE;
                        if (tunables) {
                                for (i = 0; i < tunables; i++)
                                        tmp[i]->flags &= ~COND_BOOL_FLAGS_TUNABLE;
                        }
                } else {
......
```

保留 tunable 标识符及其 cond_node_t 数据结构的核心操作，就是清除它们的 TUNABLE 标志位。从而使得在 expand 过程中能够将 tunable 标识符从 base 模块拷贝到 out 模块，以及在 cond_node_copy 函数中将 cond_node_t 加入 out->cond_list 队列，并将其 avtrue_list/avfalse_list 队列中的规则展开后注册到 out->te_cond_avtab 哈希表，并创建 avtab_key_t 数据结构的索引结构 cond_av_list_t 并加入 out->cond_list 中新元素的 true_list/false_list 队列，从而最终被写入 policy.X。

【对 refpolicy 的修改】
有了 toolchain 对“tunable”关键字的支持后，首先就是修改 gen_tunable 宏使用 tunable 关键字而不是 bool 关键字：

```
define(`gen_tunable',`
        tunable $1 dflt_or_overr(`$1'_conf,$2);
')
```

然后，在 tunable_policy 宏中调用 declare_required_tunables 宏，将在条件表达式中所使用的标识符都当作 tunable 声明为外部依赖：

```
define(`tunable_policy',`
        gen_require(`
                declare_required_tunables(`$1')
        ')
        if (`$1') {
                $2
        ifelse(`$3',``,`',`} else {
                $3
        ')}
')

define(`declare_required_tunables',`
ifelse(regexp($1, `\w'), -1, ``, `dn1
tunable regexp($1, `\(\w+\)', `\1');
declare_required_tunables(regexp($1, `\w+\(.*\)', `\1'))dn1
') dn1
')
```

另外，增加 boolean_policy 宏，用于声明和 boolean 相关的 if-ifelse 结构。注意它调用 declare_required_booleans 宏，将条件表达式中的标识符按照 boolean 声明为外部依赖：

```
define(`boolean_policy',`
        gen_require(`
                declare_required_booleans(`$1')
        ')
        if (`$1') {
                $2
        ifelse(`$3',``,`',`} else {
                $3
        ')}
')

define(`declare_required_booleans',`
ifelse(regexp($1, `\w'), -1, ``, `dnl
bool regexp($1, `\(\w+\)', `\1');
declare_required_booleans(regexp($1, `\w+\(.*\)', `\1'))dnl
') dnl
')
```

最后，还需要修改 refpolicy/support/sedoctool.py 脚本中的 gen_booleans_conf 函数，去掉其中将 tunable 加入 policy/booleans.conf 的代码片段，从而保证该文件只包含 booleans 相关信息（注意，这样做显然是不够的，因为在使用 semodule 的“-P”选项保留所有的 tunable 时，它们又应该出现在 booleans.conf 文件中。显然不应该直接删除相应代码，而应该将“-P”选项是否被使用的信息传递给 sedoctool.py 脚本，从而决定是否写出 tunable 相关信息到 booleans.conf 文件中。这个问题仍有待解决）。

后记：到目前为止（2012 年 1 月初），本小节关于 refpolicy 的修改仍然没有被社区所合并。cjp 的解释是他年前非常忙，尚未抽出时间仔细考虑应该如何修改 refpolicy。

9.5.4 测试结果

1，默认情况下不保留 tunable，编译后发现 policy.X 明显变小，由 466k 直降到 316k 字节。

这是由于绝大部分 tunable（总数超过 150 个）的默认值都为 0，而且只有一个 if 分支而没有 ifelse 分支，所以它们通通被忽略。

2，此时真正的 boolean 只有 4 个，可以使用“checkpolicy -dbM”查看：

```
Choose:  f
secure_mode : 0
pppd_can_insmod : 0
secure_mode_insmod : 0
secure_mode_policyload : 0
```

3，另外，目前“console_login”这个 tunable 的缺省值为真，所以在 policy.X 中包含相关的 type_change 规则，可以使用 apol 工具查看：

```
ll rules match the search criteria.
Number of enabled conditional rules: 0
Number of disabled conditional rules: 0

type_change auditadm_t console_device_t : chr_file user_tty_device_t;
type_change dbadm_t console_device_t : chr_file user_tty_device_t;
type_change guest_t console_device_t : chr_file user_tty_device_t;
type_change logadm_t console_device_t : chr_file user_tty_device_t;
type_change secadm_t console_device_t : chr_file user_tty_device_t;
```

```
type_change staff_t console_device_t : chr_file user_tty_device_t;
type_change sysadm_t console_device_t : chr_file user_tty_device_t;
type_change unconfined_t console_device_t : chr_file user_tty_device_t;
type_change user_t console_device_t : chr_file user_tty_device_t;
type_change webadm_t console_device_t : chr_file user_tty_device_t;
type_change xguest_t console_device_t : chr_file user_tty_device_t;
```

4，将“console_login”tunable 的默认值设置为假，重编 policy.X，则使用 apol 工具发现上述 type_change 规则不复存在：

```
0 rules match the search criteria.
Number of enabled conditional rules: 0
Number of disabled conditional rules: 0
```

相应地，policy.X 的体积也从 3163312 字节进一步降低到了 3163180 字节。

5，给 semodule 命令使用“-P”选项，保留所有的 tunable。编译后发现 policy.X 又恢复到了之前的 466k 字节。进一步使用“checkpolicy -dbM”查看，可以看到包含所有的 tunable：

```
ls /usr/share/selinux/refpolicy-mls/*.pp | grep -v base.pp | sudo /usr/sbin/semodule -P -s refpolicy-mls
-b /usr/share/selinux/refpolicy-mls/base.pp

cao@cao-laptop:/etc/selinux/refpolicy-mls$ ls -lt policy/
total 12240
-rw-r--r--. 1 root root 4666684 2011-08-29 14:49 policy.24
cao@cao-laptop:/etc/selinux/refpolicy-mls$

Choose:  f
allow_ftpd_full_access : 0
allow_zebra_write_config : 0
cdrecord_read_content : 0
fcron_crond : 0
mmap_low_allowed : 0
samba_share_fusefs : 0
sepgsql_enable_users_ddl : 1
allow_ftpd_use_cifs : 0
allow_java_execstack : 0
cron_can_relabel : 0
openvpn_enable_homedirs : 0
......
```

9.5.5 其他经验总结

1，libsepol.so 和 libsemanage.so 动态链接库的编译过程都使用了“--version-script=”选项，由具体的文件显示地指定所创建的动态链接库需要导出哪些符号。比如：

```
cc -Werror -Wall -W -Wundef -Wshadow -Wmissing-noreturn -Wmissing-format-attribute -I. -I../include
-D_GNU_SOURCE  -shared -o libsepol.so.1 assertion.lo avrule_block.lo avtab.lo boolean_record.lo booleans.lo
conditional.lo constraint.lo context.lo context_record.lo debug.lo ebitmap.lo expand.lo genbools.lo
genusers.lo handle.lo hashtab.lo hierarchy.lo iface_record.lo interfaces.lo link.lo mls.lo module.lo
node_record.lo nodes.lo polcaps.lo policydb.lo policydb_convert.lo policydb_public.lo port_record.lo
ports.lo roles.lo services.lo sidtab.lo symtab.lo user_record.lo users.lo util.lo write.lo -Wl,-
soname,libsepol.so.1,--version-script=libsepol.map,-z,defs
```

而 libsepol.map 文件的内容如下：

```
cao@cao-laptop:/work/selinux/selinux/libsepol/src$ cat libsepol.map
{
  global:
        sepol_module_package_*; sepol_link_modules; sepol_expand_module; sepol_link_packages;
        sepol_bool_*; sepol_genbools*;
        sepol_context_*; sepol_mls_*; sepol_check_context;
        sepol_iface_*;
        sepol_port_*;
        sepol_node_*;
        sepol_user_*; sepol_genusers; sepol_set_delusers;
        sepol_msg_*; sepol_debug;
        sepol_handle_*;
        sepol_policydb_*; sepol_set_policydb_from_file;
        sepol_policy_kern_*;
        sepol_policy_file_*;
        sepol_get_disable_dontaudit;
        sepol_set_disable_dontaudit;
        sepol_set_expand_consume_base;
        sepol_get_preserve_tunables; sepol_set_preserve_tunables;
  local: *;
};
```

.map 文件的 global 字段用于显式地声明当前.so 所导出的符号（能够被其他 ELF 所使用）；而 local 字段默认地匹配所有其他字符串，即不在 global 字段声明的所有其它函数，都不被导出（只能在库内部使用），和在源代码中使用“static”关键字的效果一样。

对于 libsepol 的两个新增函数 sepol_get/set_preserve_tunables，必须导出它们以被 libsemanage.so 所使用。在上述文件 global 字段的最后加入这两个函数名，则使用 nm 命令可以看到 libsepol.so 的确导出了它们（注意为大写的“T”）：

```
cao@cao-laptop:/work/selinux/selinux/libsepol/src$ nm libsepol.so | grep preserve_tunables
000000000001a826 T sepol_get_preserve_tunables
000000000001a861 T sepol_set_preserve_tunables
cao@cao-laptop:/work/selinux/selinux/libsepol/src$
```

在开发过程中犯过的一个错误就是在修改 libsepol.map 文件时这两个函数名都忘记了末尾的“s”，导致他么呢没有被任何 global 字段中的模式所匹配，实际上就被匹配成 local，于是它们就没有被 libsepol.so 所导出（注意为小写的“t”）：

```
cao@cao-laptop:/work/selinux/selinux/libsepol/src$ nm libsepol.so | grep preserve_tunables
000000000001a7b6 t sepol_get_preserve_tunables
000000000001a7f1 t sepol_set_preserve_tunables
cao@cao-laptop:/work/selinux/selinux/libsepol/src$
```

注意，nm 的结果中“T”为导出符号，而“t”为非导出的、内部使用的符号！

最终导致链接 libsemanage.so 的过程报错：这两个函数没有被定义（undefined reference）：

```
cao@cao-laptop:/work/selinux/selinux/libsemanage/src$ cc -Werror -Wall -W -Wundef -Wshadow -Wmissing-
noreturn -Wmissing-format-attribute -I../include -I/usr/include -D_GNU_SOURCE   -shared -o libsemanage.so.1
boolean_record.lo booleans_active.lo booleans_activedb.lo booleans_file.lo booleans_local.lo
booleans_policy.lo booleans_policydb.lo context_record.lo database_activedb.lo database.lo database_file.lo
database_join.lo database_llist.lo database_policydb.lo debug.lo direct_api.lo fcontext_record.lo
fcontexts_file.lo fcontexts_local.lo fcontexts_policy.lo genhomedircon.lo handle.lo iface_record.lo
interfaces_file.lo interfaces_local.lo interfaces_policy.lo interfaces_policydb.lo modules.lo
node_record.lo nodes_file.lo nodes_local.lo nodes_policy.lo nodes_policydb.lo parse_utils.lo
policy_components.lo port_record.lo ports_file.lo ports_local.lo ports_policy.lo ports_policydb.lo
```

```
semanage_store.lo seuser_record.lo seusers_file.lo seusers_local.lo seusers_policy.lo user_base_record.lo
user_extra_record.lo user_record.lo users_base_file.lo users_base_policydb.lo users_extra_file.lo
users_join.lo users_local.lo users_policy.lo utilities.lo conf-scan.lo conf-parse.lo -lsepol -lselinux
-lbz2 -lustr -L/usr/lib -Wl,-soname,libsemanage.so.1,--version-script=libsemanage.map,-z,defs
direct_api.lo: In function `semanage_direct_connect':
direct_api.c:(.text+0x872): undefined reference to `sepol_set_preserve_tunables'
direct_api.c:(.text+0x88c): undefined reference to `sepol_set_preserve_tunables'
direct_api.lo: In function `semanage_direct_commit':
direct_api.c:(.text+0x2361): undefined reference to `sepol_get_preserve_tunables'
direct_api.c:(.text+0x2382): undefined reference to `sepol_get_preserve_tunables'
direct_api.c:(.text+0x23a1): undefined reference to `sepol_get_preserve_tunables'
handle.lo: In function `semanage_get_preserve_tunables':
handle.c:(.text+0x6a9): undefined reference to `sepol_get_preserve_tunables'
handle.lo: In function `semanage_set_preserve_tunables':
handle.c:(.text+0x6f5): undefined reference to `sepol_set_preserve_tunables'
collect2: ld returned 1 exit status
```

修改了在 libsepol.map 中的错误、使得 libsepol.so 正确地导出了这两个函数后，上述错误就自然被解决了。

2，在调试上述错误期间，发现可以在命令行给 gcc 传递参数“-Wl,--verbose”（注意“W”后是字母“l”而不是数字“1”，以表示在 link 时使用），使 gcc 打印使用“-l”选项所链接的库的绝对路径，比如：

```
attempt to open /usr/lib/libsepol.so succeeded
-lsepol (/usr/lib/libsepol.so)
```

从而确定链接过程所使用的库是正确的。

另外，可以使用“gcc --print-search-dirs”打印 gcc 对库的搜索路径。

3，可以使用 linux 内核源代码中的 scripts/checkpatch.pl 和 scripts/cleanpatch 脚本来发现并修正和 whitespace 相关的错误，比如：

```
cao@cao-laptop:/work/linux-2.6$ scripts/checkpatch.pl /work/selinux/selinux/tmp2/0003-Write-and-read-
TUNABLE-flags-in-related-data-structu.patch
ERROR: "foo * bar" should be "foo *bar"
#30: FILE: libsepol/src/conditional.c:567:
+int cond_read_bool(policydb_t * p,

ERROR: trailing whitespace
#55: FILE: libsepol/src/conditional.c:822:
+^I$

ERROR: trailing whitespace
#99: FILE: libsepol/src/write.c:741:
+^I    p->policyvers >= MOD_POLICYDB_VERSION_TUNABLE_SEP) {^I$

total: 3 errors, 0 warnings, 75 lines checked

NOTE: whitespace errors detected, you may wish to use scripts/cleanpatch or
      scripts/cleanfile

/work/selinux/selinux/tmp2/0003-Write-and-read-TUNABLE-flags-in-related-data-structu.patch has style
problems, please review.  If any of these errors
are false positives report them to the maintainer, see
CHECKPATCH in MAINTAINERS.
```

```
cao@cao-laptop:/work/linux-2.6$ scripts/cleanpatch /work/selinux/selinux/tmp2/0003-Write-and-read-TUNABLE-flags-in-related-data-structu.patch
cleanpatch: /work/selinux/selinux/tmp2/0003-Write-and-read-TUNABLE-flags-in-related-data-structu.patch
cao@cao-laptop:/work/linux-2.6$ scripts/checkpatch.pl /work/selinux/selinux/tmp2/0003-Write-and-read-TUNABLE-flags-in-related-data-structu.patch
ERROR: "foo * bar" should be "foo *bar"
#30: FILE: libsepol/src/conditional.c:567:
+int cond_read_bool(policydb_t * p,

total: 1 errors, 0 warnings, 75 lines checked

/work/selinux/selinux/tmp2/0003-Write-and-read-TUNABLE-flags-in-related-data-structu.patch has style problems, please review.  If any of these errors
are false positives report them to the maintainer, see
CHECKPATCH in MAINTAINERS.
cao@cao-laptop:/work/linux-2.6$
```

虽然在开发用户态代码时并没有明确强制要求运行 checkpatch.pl 脚本来检查 patch，但是最好也这样做并改正其中的 whitespace 错误，否则容易给别人造成麻烦。

4，关于 discard_tunable 函数的调用位置。

原来我是把这个函数放到 link_modules 函数的末尾位置调用，因为在 link 过程的后部就可以确定一个 boolean/tunable 标识符以及一个 cond_node_t 的属性了。这正是得益于如下事实：link 后得到的 base.pp 即为当前 refpolicy 的“圆满”描述：所有 block/dec1 中定义的标识符和规则以及各种结构的描述，悉数被拷贝到 base.pp。

所以才可以基于 base.pp 展开对 refpolicy 的分析工作。（TODO：Tresys 公司利用什么工具，如何分析 base.pp？）

因此，我们需要维持 base.pp 的完整性。由于 discard_tunables 会修改当前 dec1->avrules 队列（追加该 dec1 中所有 tunable 有效的分支的规则），因此 Joshua 强烈建议将该函数由 link 过程的末尾转移到 expand 过程的最开始（现在正是放到了 expand_module 函数的最开始），这样不但不影响该函数的任何操作，也能够维护 base.pp 的完整性。

5，使用 gdb 调试 segmentation fault。

在 toolchain 开发期间难免会引入 segmentation fault。传统的做法是在关键函数的入、出口上增加打印语句，或者在关键函数执行流上增加打印语句，从而逐步定位 segmentation fault 的爆发点。

这样做往往需要采用“二分法”，耗费许多精力逐步添加许多打印语句，最终才能完成定位工作。

更为准确和快速的方法就是使用 gdb 调试 semodule 创建 module store 的过程：
1）首先指定“DEBUG=1”并重新编译 toolchain：

```
sudo make DEBUG=1 install
```

2）然后在 gdb 的“--args”选项后面，悉数补充 semodule 命令行的所有参数，比如：

```
gdb --args /usr/sbin/semodule -s refpolicy -b /usr/share/selinux/refpolicy/base.pp -i /usr/share/selinux/refpolicy/abrt.pp ...
```

3）接着就可以使用 gdb 的“run”命令直接运行，并在发生 segfault 时使用“bt”命令打印当前的函数调用链，比如：

```
Program received signal SIGSEGV, Segmentation fault.
```

```
0x00007fb03c2fd052 in ?? () from /lib/libc.so.6
(gdb) bt
#0  0x00007fb03c2fd052 in ?? () from /lib/libc.so.6
#1  0x00007fb03bc4e691 in ustr_replace_cstr () from /usr/lib/libustr-1.0.so.1
#2  0x00007fb03c615846 in replace_all (
    str=0xac3ab10 "HOME_DIR/.+\tsystem_u:object_r:user_home_t",
    rep1=0x7fff5d711cd0) at genhomedircon.c:442
#3  0x00007fb03c615ad8 in write_home_dir_context (s=0x7fff5d711df0,
    out=0x10301f50, tp1=0xe2dc170, user=0xe25cec0 "user_u",
    seuser=0xe25cec0 "user_u", home=0xe2913d3 "/home/[^/]*",
    role_prefix=0xe261d30 "user", level=0x0) at genhomedircon.c:509
#4  0x00007fb03c616b50 in write_context_file (s=0x7fff5d711df0, out=0x10301f50)
    at genhomedircon.c:942
#5  0x00007fb03c616df0 in semanage_genhomedircon (sh=0x228b790,
    policydb=0xaab4370, usepasswd=1) at genhomedircon.c:1017
#6  0x00007fb03c611baa in semanage_direct_commit (sh=0x228b790)
    at direct_api.c:1008
#7  0x00007fb03c617c44 in semanage_commit (sh=0x228b790) at handle.c:435
#8  0x0000000000402471 in main ()
(gdb)
```

和采用“二分法”手工添加 printf 语句相比，gdb 显然便捷许多，我们可以直观地看出 segfault 爆发点在 genhomedircon.c:442 行 replace_all 函数中，gdb 同时还能显示该函数的调用参数:-)

6，如果修改了 boolean 或 tunable 的默认值，则建议手工删除 policy/booleans.conf 文件（或者直接执行 make bare 做充分彻底的清除），然后执行 make conf 和 make load 重新编译 policy.X。

我们可以确认（通过 apol 或者 sesearch 工具），对于默认值由 false 修改为 true 的 tunable，尽管其没有被写入 policy.X，但是其控制 avtrue_list 分支上的规则能够正确地生效。

9.N 在策略中指定 newcontext 的缺省设置方法（todo）

9.N.1 提出问题 - newcontext 的设置策略被硬编码到机制中

目前 refpolicy 和 SELinux 内核驱动都没有实现对 RBAC 的**完全**支持。比如从 security_compute_sid 函数的代码片段来看：

```
/* Set the role and type to default values. */
if ((tclass == policydb.process_class) || (sock == true)) {
        /* Use the current role and type of process. */
        newcontext.role = scontext->role;
        newcontext.type = scontext->type;
} else {
        /* Use the well-defined object role. */
        newcontext.role = OBJECT_R_VAL;
        /* Use the type of the related object. */
        newcontext.type = tcontext->type;
}
```

只有 process 类或者各种 socket 类的对象的 role/type 默认继承创建者的 role/type，而其他所有类（包括 UNIX 所有文件类型等）对象的 role 都被设置为“object_r”。相应的，在 policydb_context_isvalid 和 mls_context_isvalid 函数中只有当一个 SC 中的 role 不是 object_r 时才检查 user-role 和 role-type 之间，以及 user-range 之间是否能够合法地关联。

其实，**这段代码最大的问题并不是只特殊处理了 process 和 socket 类，而是把不同类新创建对象的 newcontext 的缺省设置方法硬编码到了 SELinux 内核驱动中！**而在这里本应该只实现机制，而不同类对象 newcontext 的缺省设置方法完全属于策略范畴，所以应该在 refpolicy 中实现。

将来希望在 refpolicy 中定义如下新语法，来指定不同类对象的 SC 中各个部分的缺省构建方法，比如：

```
user_default { file dir process socket ... } fromsource;

role_default { file dir ... } fromtarget;
role_default { process socket ... } fromsource;

type_default { file dir ... } fromtarget;
type_default { process socket ... } fromsource;

range_default { file dir ... } fromtarget;
range_default { process socket ... } fromsource;
```

同时 refpolicy 中还提供 role/type/range transition 规则，以重载缺省的设置方法。那么在 security_compute_sid 函数中就可以根据上述缺省规则来设置 newcontext 的个域，然后再应用各种 transition 规则重载缺省的设置。

10. SELinux 内核驱动分析小记

10.1 LSM 核心数据结构及相应回调函数

在 include/linux/security.h 中定义的 security_operations 数据结构包含了许多函数指针，它们正是 LSM 上层框架和 LSM 具体实现之间的接口。在 security/security.c 中定义的 security_ops 为指向上述数据结构的指针，当某个 LSM 的具体实现向内核注册时（调用 register_security 函数），security_ops 被指向具体实现所提供的 security_operations 数据结构，比如 SELinux 的 selinux_ops 结构。

在 include/linux/security.h 中还定义了所有和安全检查相关的函数，都以 security_开头，内核源代码中对这些回调函数的调用即为 LSM 上层框架。 这些回调函数通常在访问内核数据结构前被调用，用于检查当前执行流是否具有相应的权限。如果 CONFIG_SECURITY 无效，则这些函数都是空函数，否则几乎都通过调用 security_ops 所指向的 LSM 具体实现的相关方法来完成，比如：

```
int security_dentry_open(struct file *file)
{
        return security_ops->dentry_open(file);
}
```

每种 LSM 的具体实现都注册自己的 security_operations 方法表（参见 register_security 函数的调用者），其中的函数可以分成如下几类：

```
linux/include/linux$ grep "Security hook" security.h
 * Security hooks for program execution operations.                    # domain transition
 * Security hooks for filesystem operations.
 * Security hooks for inode operations.
 * Security hooks for file operations
 * Security hooks for dentry
 * Security hooks for task operations.
 * Security hooks for Netlink messaging.
 * Security hooks for Unix domain networking.
 * Security hooks for socket operations.
 * Security hooks for XFRM operations.                                 # Labeled networking
 * Security hooks affecting all Key Management operations
 * Security hooks affecting all System V IPC operations.
 * Security hooks for individual messages held in System V IPC message queues
 * Security hooks for System V IPC Message Queues
 * Security hooks for System V Shared Memory Segments
 * Security hooks for System V Semaphores
 * Security hooks for Audit
```

《Implementing SELinux as a Linux Security Module》一文描述了上述各类中每个回调函数在 SELinux 上的实现方法。

10.2 SELinux 核心数据结构

10.2.1 SELinux 对内核数据结构的扩展

当 CONFIG_SECURITY 有效时，相关内核数据结构中会增加一个 void *指针，比如 task_struct 中的 security 指针，inode/file 中的 i_security/i_security 指针，LSM 具体实现对相应内核数据结构的扩展就由这些指针所指向。

SELinux 对内核数据结构的扩展都定义在 security/selinux/include/objsec.h 中，相关数据结构的命名方式均采用在原有内核数据结构名称中插入“_security_”字样，比如 task_struct 的安全属性为 task_security_struct。

10.2.1.1 进程的安全属性

```
struct task_security_struct {
        u32 osid;               /* SID prior to last execve /          # /proc/pid/attr/prev
        u32 sid;                /* current SID /                       # /proc/pid/attr/current
        u32 exec_sid;           /* exec SID */                          # /proc/pid/attr/exec
        u32 create_sid;         /* fscreate SID */                      # /proc/pid/attr/fscreate
        u32 keycreate_sid;      /* keycreate SID */                     # /proc/pid/attr/keycreate
        u32 sockcreate_sid;     /* fscreate SID */                      # /proc/pid/attr/sockcreate
};
```

SELinux 通过/proc/pid/attr/下的各种文件导出该数据结构各个域的值，参见注释。用户态 SELinux-aware 应用程序通过 libselinux 提供的 API 来访问这些文件，参见 11.1 小节。

10.2.1.2 文件和打开文件描述符的安全属性

```
struct inode_security_struct {
        struct inode *inode;    /* back pointer to inode object */
        struct list_head list;  /* list of inode_security_struct */
        u32 task_sid;           /* SID of creating task */              # 创建者的SID
        u32 sid;                /* SID of this object */                # 当前文件的SID
        u16 sclass;             /* security class of this object */     # 当前文件的class kernel value
        unsigned char initialized;      /* initialization flag */       # 在被创建时初始化SID后设置
        struct mutex lock;
};

struct file_security_struct {
        u32 sid;                /* SID of open file description */              # 打开文件描述符对象的SID
        u32 fown_sid;           /* SID of file owner (for SIGIO) */
        u32 isid;               /* SID of inode at the time of file open */     # 文件自身的SID
        u32 pseqno;             /* Policy seqno at the time of file open */     # 用于Revalidation
};
```

inode_security_struct 和 file_security_struct 分别和已打开文件的 inode 和 file 数据结构相对应。isec->sid 即为该文件的安全属性。对于支持扩展属性的文件系统（使用 fs_use_xattr 规则），即为相应文件磁盘索引节点的“security.selinux”扩展属性中保存的 SC 标签注册到 sidtab 后返回的值。对于以其他方式确定 SC 的文件系统，isec->sid 的确定方式由 sbsec->behavior 决定（参见 inode_doinit_with_dentry 函数）。

isec->task_sid 为文件创建者进程的 sid。对于使用 fs_use_task 规则来确定 SC 的文件系统，其中所有文件的 isec->sid 均等于 isec->task_sid（参见 inode_doinit_with_dentry 函数）。

fsec->sid 为相应打开文件描述符的 sid，注意它和被打开文件属于不同的内核数据对象。在 SELinux 中区分二者，fsec->sid 总是等于打开文件的进程的 sid。所以打开文件描述符 fd 继承相应进程的 type，这也就能够解释为什么在 domtrans_pattern 宏中存在如下规则：

```
allow $3 $1:fd use;
```

它许可子进程的新 domain（$3）能够继续使用从父进程（$1）那里继承的打开文件描述符。

在 UNIX 上“万物皆文件”，因此只在 isec->sclass 域来记录相应 object 所属的 class 信息。注意 SELinux class 信息无须保存在 SC 或 SID 中，而由相应 object 自身的属性（inode->i_mode）决定。

注意 fsec->pseqno 保存在打开文件时 avc_cache 数据结构的“编号”。如果在打开文件后重新装载了 policy.X，或者修改了任意 boolean 的数值，则 avc_cache.latest_notif 都将跟随全局静态变量 latest_granting 而递增 1（同时当前 avc cache 也会被 flush，即所有现存 avc_node 都会被删除），那么在 selinux_file_permission 函数中（在实际文件操作执行前被调用）就会发现 fsec->pseqno 和 avc_cache.latest_notif 数值不一致，此时就需要重新计算当前进程 domain 对相应对象的权限了（称为 revalidation）。

10.2.1.3 socket 的安全属性

```
struct sk_security_struct {
#ifdef CONFIG_NETLABEL
        enum {                          /* NetLabel state */
                NLBL_UNSET = 0,
                NLBL_REQUIRE,
                NLBL_LABELED,
                NLBL_REQSKB,
                NLBL_CONNLABELED,
        } nlbl_state;
        struct netlbl_lsm_secattr *nlbl_secattr; /* NetLabel sec attributes */
#endif
        u32 sid;                        /* SID of this object */
        u32 peer_sid;                   /* SID of peer */
        u16 sclass;                     /* sock security class */
};
```

sk_security_struct 数据结构即描述一个 socket 对象的安全属性，由 socket->sock->sk_security 指针所指向。sclass 为该 socket 所在的 SELinux class，sid 即为该 socket 的 sid（通常继承于其创建者进程，但其 type 也可能会被 type_transition 规则或者显式地通过/proc/pid/attr/sockcreate 而改变），而 peer_sid 即为和该 socket 已经建立了面向链接的（比如 unix_stream_socket 或者 tcp_socket 类型）对端 socket 的 sid。

在 selinux_socket_getpeersec_stream 函数中即获得 sksec->peer_sid，并通过 security_sid_to_context 函数返回对应的 SC 字符串，最终通过 copy_to_user 返回给用户态的 getpeercon 函数。

10.2.1.4 文件系统超级块的安全属性

```
struct superblock_security_struct {
        struct super_block *sb;         /* back pointer to sb object */
        u32 sid;                        /* SID of file system superblock */           # fscontext=
        u32 def_sid;                    /* default SID for labeling */                # defcontext=
        u32 mntpoint_sid;               /* SECURITY_FS_USE_MNTPOINT context for files */ # context=
        unsigned int behavior;          /* labeling behavior */                  # sbsec->sid 的确认方式
        unsigned char flags;            /* which mount options were specified */ # mount data 中选项的类型
        struct mutex lock;
        struct list_head isec_head;     # 需要被重新初始化的 isec 队列（相应文件在装载 policy.X 之前被打开）
        spinlock_t isec_lock;
};
```

superblock_security_struct 数据结构用于描述一个文件系统超级块的安全属性，其中

sid/def_sid/mntpoint_sid 域分别由 mount 选项 fscontext/defcontext/context 确定，分别描述文件系统自身（即超级块数据结构本身）的 SC、文件缺省 SC 以及“Mount Point Labeling”时整个文件系统的 SC。

在 superblock_alloc_security 函数中，将 sid 和 mntpoint_sid 初始化为 SECINITSID_UNLABELED，将 def_sid 初始化为 SECINITSID_FILE。

behavior 域指名该文件系统自身安全属性（即超级块数据结构 sbsec->sid 域）的确定方式，由 refpolicy 中相应 fs_use_xxx/genfscon 规则决定。如果没有任何匹配的规则，则设置为 SECURITY_FS_USE_NONE。相关定义如下：

```
#define SECURITY_FS_USE_XATTR          1 /* use xattr */                              # fs_use_xattr
#define SECURITY_FS_USE_TRANS          2 /* use transition SIDs, e.g. devpts/tmpfs */ # fs_use_trans
#define SECURITY_FS_USE_TASK           3 /* use task SIDs, e.g. pipefs/sockfs */      # fs_use_task
#define SECURITY_FS_USE_GENFS          4 /* use the genfs support */                  # genfscon
#define SECURITY_FS_USE_NONE           5 /* no labeling support */
#define SECURITY_FS_USE_MNTPOINT       6 /* use mountpoint labeling */                # context=
```

flags 域用于描述挂载文件系统时所指定 mount data 中选项的类型，以及 sbsec 自身的其他属性（比如是否完成初始化等），目前定义的标志位如下：

```
/* Mask for just the mount related flags */
#define SE_MNTMASK      0x0f
/* Super block security struct flags for mount options */
#define CONTEXT_MNT     0x01                                    # context=
#define FSCONTEXT_MNT   0x02                                    # fscontext=
#define ROOTCONTEXT_MNT 0x04                                    # rootcontext=
#define DEFCONTEXT_MNT  0x08                                    # defcontext=

/* Non-mount related flags */
#define SE_SBINITIALIZED        0x10                            # sbsec 已完成初始化
#define SE_SBPROC               0x20                            # 相应文件系统为 procfs
#define SE_SBLABELSUPP          0x40                            # 没使用 genfscon 规则（除 sysfs 之外）
```

在为打开文件创建内核 inode 数据结构时需要初始化其 inode_security_struct 数据结构。如果此时 policy.X 尚未装载、Security Server 尚未初始化（ss_initialized 等于 0），则在 inode_doinit_with_dentry 函数中将其加入 sbsec->isec_head 队列；等到 Security Server 完成初始化后，需要遍历所有已经挂载的文件系统系统的 super_block 数据结构，按照指定的规则初始化相应的 sbsec 及其 sbsec->isec_head 队列中的所有 isec 数据结构。

sbsec 数据结构中各个域的设置和使用详见 10.6.1 小节。

（TODO：补充对其他内核数据结构的扩展 xxx_security_struct 的分析）

10.2.2 AVC 数据结构

通过 AVC（Access Vector Cache）来提高 SELinux 内核驱动查询 Security Server 的效率，把就某一特定（ssid，tsid，tclass）三元组查询的结果（tsid@tclass 所支持的访问方式，以及对该 ssid 所许可的访问方式）组织到一张哈希表中，只有当 cache miss 时才查询 Security Server。

AVC 的核心为一张哈希表，由如下 avc_cache 数据结构定义：

```
struct avc_cache {
```

```
        struct list_head        slots[AVC_CACHE_SLOTS];
        spinlock_t              slots_lock[AVC_CACHE_SLOTS];   /* lock for writes */
        atomic_t                lru_hint;                      /* LRU hint for reclaim scan */
        atomic_t                active_nodes;
        u32                     latest_notif;                  /* latest revocation notification */
};
```

哈希表长 512，即含有 512 个冲突项链表，每个链表中的元素即为 avc_node 数据结构。就某一个（ssid，tsid，tclass）三元组根据如下哈希算法选择判定结果所在的队列：

```
static inline int avc_hash(u32 ssid, u32 tsid, u16 tclass)
{
        return (ssid ^ (tsid<<2) ^ (tclass<<4)) & (AVC_CACHE_SLOTS - 1);
}
```

avc_node 数据结构是 avc_entry 数据结构的封装，提供 list_head 连接件把 avc_entry 组成到 avc_cache 的某个队列中：

```
struct avc_node {
        struct avc_entry        ae;
        struct list_head        list;
        struct rcu_head         rhead;
};
```

而 avc_entry 数据结构描述某特定的（ssid，tsid，tclass）三元组及向 Security Server 的查询结果（由 av_decision 数据结构描述）：

```
struct avc_entry {
        u32                     ssid;
        u32                     tsid;
        u16                     tclass;
        struct av_decision      avd;
        atomic_t                used;   /* used recently */
};
```

其中 av_decision 数据结构（简称 avd）描述当前 tsid@tclass 对 ssid 所支持的访问方式：

```
struct av_decision {
        u32 allowed;
        u32 auditallow;
        u32 auditdeny;
        u32 seqno;
        u32 flags;
};

/* definitions of av_decision.flags */
#define AVD_FLAGS_PERMISSIVE    0x0001
```

refpolicy 实现的 TE 规则分为两类：AVTAB_AV 和 AVTAB_TYPE，前者包含 allow/auditallow/auditdeny 规则，后者包含 type_transition/member/change 规则。就一组特定的（ssid，tsid，tclass）可能定义有不同类型的 AVTAB_AV 规则，所以在 av_decision 数据结构中分别设计的相应的字段以保存相应规则的输出。

目前任何 class 都只支持最多 32 个 permission，所以可以用一个 u32 字段描述当前 tsid@tclass 对相应 ssid 所支持的 permission 位图。

注意，seqno 为创建相应 avc_node 数据结构时 avc_cache 数据结构的“编号”avc_cache.latest_notif，而它等于全局静态变量 latest_granting 的数值：初始为 0，每次 reload policy.X 后或修改任意 boolean 时递增 1。

avc_has_perm_noaudit 函数用于检查当前操作是否被许可。它首先调用 security_compute_av 函数根据（ssid，tsid，tclass）计算相应的 av_decision。如果 ssid 相关的 domain 为 permissive 的（即在 refpolicy 中用“permissive”规则而非普通“type”规则定义，则其被加入 policy.X 的 permissive_map 位图），此时 av_decision.flags 中的 AVD_FLAGS_PERMISSIVE 标志被设置，从而在 avc_has_perm_noaudit 函数中即使发现 ssid 对 tsid/class 所请求的操作不允许，也不返回失败。参见下文。

10.2.3 内核 policydb 中保存 TE 规则的数据结构

refpolicy 实现的规则分为两类：TE 规则和 RBAC 规则。TE 规则又进一步分为两类：AVTAB_AV 类规则和 AVTAB_TYPE 类规则，分别包含 allow/auditallow/auditdeny 规则以及 type_transition/member/change 规则。

注意，无论何种类型的规则，policy.X 和 SELinux 内核驱动中只会涉及规则的“展开描述”，而规则的“字面描述”（比如含有属性，或使用特殊符号“-”）只会出现在 pp 中。在 expand 过程中展开规则的字面描述，比如将属性的能力散播给所有其成员等。

TE 规则的格式如下：

```
规则类型（即 specified） source_type target_type : target_class perms/new type;
```

前三部分由 avtab_key 数据结构描述，规则的结果（tsid@tclass 对 ssid 所许可的能力位图，或者 new type）则由 avtab_datum 数据结构描述。

```
struct avtab_key {
        u16 source_type;        /* source type */
        u16 target_type;        /* target type */
        u16 target_class;       /* target object class */
#define AVTAB_ALLOWED           0x0001
#define AVTAB_AUDITALLOW        0x0002
#define AVTAB_AUDITDENY         0x0004
#define AVTAB_AV                (AVTAB_ALLOWED | AVTAB_AUDITALLOW | AVTAB_AUDITDENY)
#define AVTAB_TRANSITION        0x0010
#define AVTAB_MEMBER            0x0020
#define AVTAB_CHANGE            0x0040
#define AVTAB_TYPE              (AVTAB_TRANSITION | AVTAB_MEMBER | AVTAB_CHANGE)
#define AVTAB_ENABLED_OLD   0x80000000 /* reserved for used in cond_avtab */
#define AVTAB_ENABLED           0x8000 /* reserved for used in cond_avtab */
        u16 specified;  /* what field is specified */
};
```

注意，specified 域不但指明规则的类型(AVTAB_AV 或者 AVTAB_TYPE)，而且标识当前规则是否生效（有效时 AVTAB_ENABLED 标志被设置）。该特性用于实现在运行时通过 boolean 的状态值来控制条件规则块（conditional policy）中实际生效的规则（条件规则块中所有规则的展开描述都加入 te_cond_avtab 哈希表，根据 boolean 的当前状态值确定相应条件表达式的逻辑值，进而设置或清除相应分支上所有规则的 AVTAB_ENABLED 标志）。

```
struct avtab_datum {
```

```
        u32 data; /* access vector or type value */
};
```

注意，由于当前每个 class 最多只支持 32 个权限，所以可以用一个 u32 类型描述所有支持的权限的位图。

描述 TE 规则的 avtab_node 数据结构又组成 te_avtab 和 te_cond_avtab 规则哈希表（分别对应永久有效的规则和受 boolean 控制的条件规则）：

```
struct avtab_node {
        struct avtab_key key;
        struct avtab_datum datum;
        struct avtab_node *next;
};

struct avtab {
        struct avtab_node **htable;
        u32 nel;        /* number of elements */
        u32 nslot;      /* number of hash slots */
        u16 mask;       /* mask to compute hash func */

};

struct policydb {
        ......
        /* type enforcement access vectors and transitions */
        struct avtab te_avtab;
        ......
        /* type enforcement conditional access vectors and transitions */
        struct avtab te_cond_avtab;
        ......
};
```

10.2.4 内核 policydb 中保存 RBAC 规则的数据结构

RBAC 规则包含 role-types/allow/transition/attribute/dominance 规则，但是需要保留到运行时的只有 role-allow 和 role-transition 规则，而 role-types/attribute/dominance 规则在编译时被完全处理（规则的作用被写入相应 role 标识符的 role_datum_t 数据结构）。

由于 role-allow 和 role-transition 规则具有不同的语法结构，因此需要使用不同的数据结构描述。它们定义在 policydb.h 中：

```
struct role_allow {
        u32 role;               /* current role */
        u32 new_role;           /* new role */
        struct role_allow *next;
};

struct role_trans {
        u32 role;               /* current role */
        u32 type;               /* program executable type, or new object type */
        u32 tclass;             /* process class, or new object class */
        u32 new_role;           /* new role */
        struct role_trans *next;
};
```

数据结构中的域和相应规则中的各个域一一对应，不再赘述。

10.2.5 SELinux 规则在内核中的检查点总结（new）

在此提前归纳一下各种规则在内核中的检查点，我们将在后文中逐步见到它们。

<table>
<tr><td colspan="2">RBAC rules</td><td>role-allow</td><td rowspan="3">security_compute_av > context_struct_compute_av</td></tr>
<tr><td rowspan="6">TE rules</td><td rowspan="3">AVTAB_AV</td><td>allow</td></tr>
<tr><td>audita11ow</td></tr>
<tr><td>auditdeny</td></tr>
<tr><td rowspan="3">AVTAB_TYPE</td><td>type_transition</td><td rowspan="6">security_compute_sid > policydb_context_isvalid</td></tr>
<tr><td>type_change</td></tr>
<tr><td>type_member</td></tr>
<tr><td colspan="2" rowspan="3">RBAC rules</td><td>role-types</td></tr>
<tr><td>role-attribute/dominance</td></tr>
<tr><td>role-transition</td></tr>
<tr><td colspan="2">User rules</td><td>user-roles</td></tr>
</table>

10.3 情景分析：打开文件时的相关函数调用链

在 linux/include/security.h 中定义的 security_xxx 函数为 LSM 的上层框架，它们在访问内核数据结构的内核控制路径上被调用，检查当前执行流是否有足够的权限执行相应的操作。

在打开一个文件时创建并设置相应的 file 数据结构，__dentry_open 函数设置已创建的 file 数据结构，比如 f->f_mapping = inode->i_mapping; f->f_pos = 0; f->f_fop = fops_get(inode->i_fop)，然后调用 security_dentry_open 函数。如前所述，该函数调用 LSM 具体实现所提供的 dentry_open 函数，在 SELinux 中即为 selinux_dentry_open 函数。如果它的返回值大于 0（表示失败），则__dentry_open 中会通过 cleanup_all 处代码执行清理操作，比如释放 file 数据结构，减少 dentry/inode 引用计数等。

```
static int selinux_dentry_open(struct file *file)
{
        struct file_security_struct *fsec;
        struct inode *inode;
        struct inode_security_struct *isec;
        inode = file->f_path.dentry->d_inode;
        fsec = file->f_security;
        isec = inode->i_security;

        /*
         * Save inode label and policy sequence number
         * at open-time so that selinux_file_permission
         * can determine whether revalidation is necessary.
         * Task label is already saved in the file security
         * struct as its SID.
         */
        fsec->isid = isec->sid;                         # inode label, 即被打开文件自身的 SID
        fsec->pseqno = avc_policy_seqno();
```

首先获得待打开文件的 file/inode 中和安全相关的数据结构的地址（用 fsec/isec 指向），设置 fsec->isid = isec->sid，即为文件自身的 SID。注意在创建 file 数据结构时相应的 fsec->sid 已经被设置为

当前进程的 sid。

正如注释所述，在打开文件时将此时 avc cache 的“编号”即 avc_cache.latest_notif 保存在 fsec->pseqno 中，从而在 selinux_file_permission 函数中（在实际文件操作发生前被调用）检查是否需要重新计算 avc_node（比如重新装载 policy.X 或者修改任意 boolean 之后，当前所有 avc_cache 中的 avc_node 都会被删除）。

```
        /*
         * Since the inode label or policy seqno may have changed
         * between the selinux_inode_permission check and the saving
         * of state above, recheck that access is still permitted.
         * Otherwise, access might never be revalidated against the
         * new inode label or new policy.
         * This check is not redundant - do not remove.
         */
        return inode_has_perm(current, inode, file_to_av(file), NULL);
}
```

最后通过 inode_has_perm 函数检查是否许可当前进程以所请求的方式访问该文件（通过 file_to_av 函数返回当前进程打开该文件的方式位图）。

另外，正如这里的注释所述，在当前进程打开文件之后，该文件可能被其他进程重新 relabel，或者其他进程触发了 AVC cache 需要被 flush 的操作。所以在实际文件操作执行前必须检查是否需要执行 revalidation。

```
static int inode_has_perm(struct task_struct *tsk, struct inode *inode,
                          u32 perms, struct avc_audit_data *adp)
{
        struct task_security_struct *tsec;
        struct inode_security_struct *isec;
        struct avc_audit_data ad;

        if (unlikely(IS_PRIVATE(inode)))
                return 0;

        tsec = tsk->security;
        isec = inode->i_security;

        if (!adp) {
                adp = &ad;
                AVC_AUDIT_DATA_INIT(&ad, FS);
                ad.u.fs.inode = inode;
        }

        return avc_has_perm(tsec->sid, isec->sid, isec->sclass, perms, adp);
}
```

如上可见，inode_has_perm 为 avc_has_perm 的封装函数，传递当前进程的 sid 和目标文件的 tsid@tclass 信息，以及当前进程所请求的访问方式。由后者完成判定并产生相应的 AVC 消息：

```
int avc_has_perm(u32 ssid, u32 tsid, u16 tclass, u32 requested, struct avc_audit_data *auditdata)
{
        struct av_decision avd;
        int rc;

        rc = avc_has_perm_noaudit(ssid, tsid, tclass, requested, 0, &avd);
```

```
        avc_audit(ssid, tsid, tclass, requested, &avd, rc, auditdata);
        return rc;
}
```

真正的判定由 avc_has_perm_noaudit 函数中完成，然后根据当前 policy.X 的设置进一步调用 avc_audit 函数打印 AVC 消息（通常情况下请求被禁止时打印消息，除非 dontallow 规则存在；或者在请求被许可时存在 auditallow 规则）。参数 avd 为返回参数，用于设置和当前（ssid，tsid，tclass）相关的 av_decision 数据结构：

```
int avc_has_perm_noaudit(u32 ssid, u32 tsid, u16 tclass, u32 requested,
                         unsigned flags, struct av_decision *avd)
{
        struct avc_node *node;
        int rc = 0;                                     # 默认值为许可当前操作
        u32 denied;

        BUG_ON(!requested);
```

参数 requested 指明当前 ssid 对 tsid@tclass 所请求的访问方式，所以一定不能为 NULL。

```
        rcu_read_lock();

        node = avc_lookup(ssid, tsid, tclass);
        if (unlikely(!node)) {
                rcu_read_unlock();
                security_compute_av(ssid, tsid, tclass, avd);
                rcu_read_lock();
                node = avc_insert(ssid, tsid, tclass, avd);
        } else {
                memcpy(avd, &node->ae.avd, sizeof(*avd));
                avd = &node->ae.avd;
        }
```

首先调用 avc_lookup 函数，它通过 avc_search_node 函数在 AVC cache 中查询是否存在描述当前请求的 avc_node 数据结构，如果没有找到则需要调用 security_compute_av 函数查询 security server，得到相应的 av_decision 结果，再通过 avc_insert 函数创建一个描述当前（ssid，tsid@tclass）的判定结果的 avc_node 数据结构并加入 AVC cache。

否则，将已经存在的 avc_node.ae.avd 数据结构的地址复制到参数 avd 中。

```
        denied = requested & ~(avd->allowed);
```

av_decision 数据结构中的 allowed 字段描述当前 policy.X 所许可的 ssid 对 tsid@tclass 的访问方式。如果当前请求的访问方式**不全**在 avd.allowed 位图内，则 denied 变量不为 0，即为拒绝。

```
        if (denied) {
                if (flags & AVC_STRICT)
                        rc = -EACCES;
                else if (!selinux_enforcing || (avd->flags & AVD_FLAGS_PERMISSIVE))
                        avc_update_node(AVC_CALLBACK_GRANT, requested, ssid, tsid, tclass, avd->seqno);
                else
                        rc = -EACCES;
        }

        rcu_read_unlock();
        return rc;
```

}

如果被拒绝，则应返回-EACCESS，除非当前 SELinux 工作在 Permissive 模式（全局量 selinux_enforcing 为 0），或者当前 domain 为 Permissive Domain（av_decision.flags 中 AVD_FLAGS_PERMISSIVE 标志有效。注意，它刚刚在上面 security_compute_av 函数中根据当前 ssid 的 type 是否为 permissive 的而设置。参见下文）。

注意，通常在开发一个新的 domain 时，在开发和测试阶段将该 domain 设置为 Permissive Domain，这样即可以收集相关的 AVC 错误信息，也不影响整个系统仍然处于 Enforcing 模式。比如 Fedora 的开发就是这样：在一个版本中新增加的 domain 默认被设置为 permissive 的，而再下一个版本中再变成普通的 domain。

security_compute_av 函数计算和一组（ssid，tsid，tclass）相对应的 av_decision 结果，由输入-输出参数 avd 写回调用者分配的数据结构。注意，参数 orig_tclass 为 SELinux 内核驱动给当前 class 定义的索引，比如从 isec->sclass 而来，由下文可见通过 unmap_class 函数转换为该 class 在用户态 policydb_t 中定义的编号，以便查询 security server（用户态 checkpolicy 和 SELinux 内核驱动分别定义 class 的索引，二者的转换关系由 current_mapping 数组描述）。

```
/**
 * security_compute_av - Compute access vector decisions.
 * @ssid: source security identifier
 * @tsid: target security identifier
 * @tclass: target security class
 * @avd: access vector decisions
 *
 * Compute a set of access vector decisions based on the
 * SID pair (@ssid, @tsid) for the permissions in @tclass.
 */
void security_compute_av(u32 ssid,
                         u32 tsid,
                         u16 orig_tclass,              # class kernel value
                         struct av_decision *avd)
{
        u16 tclass;
        struct context *scontext = NULL, *tcontext = NULL;

        read_lock(&policy_rwlock);
        avd_init(avd);
```

首先调用 avc_init 函数初始化 av_decision 数据结构：allowed/auditallow 清 0，auditdeny 设置为全 1，flags 清 0，将 seqno 设置为当前 latest_granting 全局变量的值。

```
        if (!ss_initialized)
                goto allow;
```

如果 security server 尚未完成初始化，则许可所有的操作（将 allowed 置为全 1）。

```
        scontext = sidtab_search(&sidtab, ssid);
        if (!scontext) {
                printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n", __func__, ssid);
                goto out;
        }

        /* permissive domain? */
        if (ebitmap_get_bit(&policydb.permissive_map, scontext->type))
                avd->flags |= AVD_FLAGS_PERMISSIVE;
```

```
        tcontext = sidtab_search(&sidtab, tsid);
        if (!tcontext) {
                printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n", __func__, tsid);
                goto out;
        }
```

然后在 sidtab 中以 ssid 和 tsid 为索引，查找相应的 scontext 和 tcontext 数据结构。如果当前 domain（scontext->type）还被记录到 permissive_map 位图中，则设置 av_decision.flags 中的 AVD_FLAGS_PERMISSIVE 标志。

```
        tclass = unmap_class(orig_tclass);              # map class kernel value to policy value
```

如上文所述，通过 unmap_class 函数将 class kernel value 转换为 class policy value。

```
        if (unlikely(orig_tclass && !tclass)) {         # policy.X 中缺乏相应相应 class 的定义和规则的支持
                if (policydb.allow_unknown)
                        goto allow;
                goto out;                               # 即为 deny
        }
```

注意，由下文 selinux_set_mapping 函数可知，如果一个 class 只在内核中定义但是在 policy.X 中没有定义（即内核中引入了新的数据对象类型，比如加入了新的 Object Manager，但是在 refpolicy 中没有定义相关的 class 和规则支持），则相应 selinux_mapping.value 的数值为 0。在这种情况下，如果 policydb 中的 allow_unknown 标志有效（由 refpolicy build.conf 中的 UNK_PERMS 选项的配置决定，通过 checkmodule/checkpolicy 的“-U”选项写入 policy.X），则许可所有操作；否则直接退出，即不许可任何操作（avd_init 函数已将 avd->allowed 清 0）。

```
        context_struct_compute_av(scontext, tcontext, tclass, avd);
```

调用 context_struct_compute_av 函数计算 tcontext@tclass 对 scontext 所许可的能力，用 av_decision 数据结构描述（包含三个位图，分别对应三种 AVTAB_AV 类规则），参见下文。

```
        map_decision(orig_tclass, avd, policydb.allow_unknown);
```

注意，上面调用 context_struct_compute_av 函数得到的 avd 位图（描述 tcontext@tclass 对 scontext 所许可的权限），其中的非 0 位均为相应 perm 在 class 内的 policy value（因为它由查询 te_avtab 或 te_cond_avtab 得到，而它们描述 policy.X 提供的规则）！所以需要进一步调用 map_decision 函数将 avd 中位图的非 0 位转换为相应的 kernel value 并重新生成位图。参见 10.8 小节。

SELinux 内核驱动和 refpolicy 都各自定义 class 以及 perm 的索引，所以**必须把从 policy.X 查询后的 avd 结果转换为用 kernel value 表示的位图，才能继续在 SELinux 内核驱动中使用！**（与此对应，在向 p_classes 查询前将 class 的 kernel value 转换为 policy value）

```
out:
        read_unlock(&policy_rwlock);
        return;
allow:
        avd->allowed = 0xffffffff;
        goto out;
}

/*
 * Compute access vectors based on a context structure pair for the permissions in a particular class.
 */
```

```
static void context_struct_compute_av(struct context *scontext,
                                      struct context *tcontext,
                                      u16 tclass,                       # class policy value
                                      struct av_decision *avd)
{
        struct constraint_node *constraint;
        struct role_allow *ra;
        struct avtab_key avkey;
        struct avtab_node *node;
        struct class_datum *tclass_datum;
        struct ebitmap *sattr, *tattr;
        struct ebitmap_node *snode, *tnode;
        unsigned int i, j;

        avd->allowed = 0;
        avd->auditallow = 0;
        avd->auditdeny = 0xffffffff;
```

初始化 av_decision 数据结构，此处和上文 avd_init 函数的行为相同。

```
        if (unlikely(!tclass || tclass > policydb.p_classes.nprim)) {
                if (printk_ratelimit())
                        printk(KERN_WARNING "SELinux:  Invalid class %hu\n", tclass);
                return;
        }
```

首先检查用户态 class 索引是否有效。如果为 0 或者超过 p_classes.nprim（即为用户态定义的最大 class 索引），则说明当前 class 只在内核中定义，而在 refpolicy 中没有定义（即 policy.X 中不存在相应 class/perm 定义和规则定义），则立即返回（注意 avd->allowed 位图被初始化为全 0）。

```
        tclass_datum = policydb.class_val_to_struct[tclass - 1];
```

用（tclass - 1）索引 policydb_t 的 class_val_to_struct 数组，即得到相应 class_datum 数据结构的地址，下面就可以获得该 class 的 constraint 了。

```
        /*
         * If a specific type enforcement rule was defined for this permission check, then use it.
         */
        avkey.target_class = tclass;
        avkey.specified = AVTAB_AV;
```

refpolicy 实现的 TE 规则分为两类：AVTAB_AV 和 AVTAB_TYPE，前者包含 allow/auditallow/auditdeny 三种规则，后者包含 type_transition/member/change 三种规则。所有规则都组织在 policydb_t.te_avtab 和 te_cond_avtab 哈希表中。哈希表元素为 avtab_node，其中 avtab_key 即描述某条规则的（类型，source_type，target_type，target_class），而 avtab_datum 即为该规则的最后一个域：tsid@tclass 对 ssid 所许可的权限的位图，或者新的 type。

security_compute_av 函数就是组装 avtab_key 数据结构，在 te_avtab 和 te_cond_avtab 哈希表中查找和（ssid，tsid@tclass）匹配 AVTAB_AV 类规则，由输出参数 avd 返回 tsid@tclass 对 ssid 所许可的各种访问方式。

另外，security_compute_sid 函数则关心 AVTAB_TYPE 类规则，此时 avtab_datum 即为新的 type。

```
        sattr = flex_array_get(policydb.type_attr_map_array, scontext->type - 1);
        BUG_ON(!sattr);
        tattr = flex_array_get(policydb.type_attr_map_array, tcontext->type - 1);
```

```
BUG_ON(!tattr);
ebitmap_for_each_positive_bit(sattr, snode, i) {
        ebitmap_for_each_positive_bit(tattr, tnode, j) {
                avkey.source_type = i + 1;
                avkey.target_type = j + 1;
                for (node = avtab_search_node(&policydb.te_avtab, &avkey); node;
                     node = avtab_search_node_next(node, avkey.specified)) {
                        if (node->key.specified == AVTAB_ALLOWED)
                                avd->allowed |= node->datum.data;
                        else if (node->key.specified == AVTAB_AUDITALLOW)
                                avd->auditallow |= node->datum.data;
                        else if (node->key.specified == AVTAB_AUDITDENY)
                                avd->auditdeny &= node->datum.data;
                }

                /* Check conditional av table for additional permissions */
                cond_compute_av(&policydb.te_cond_avtab, &avkey, avd);
```

就特定的一组（source_type, target_type, target_class）可以分别定义了 allow/auditallow/auditdeny 规则，所以按照相匹配的规则的类型将其结果记录到 avd 的相应字段中去。

最后还要考虑 te_cond_avtab 中的规则。

（TODO: type_attr_map 的设计用意是什么？为什么要就一个 scontext->type 找到它所属的 attribute 然后再把该 attribute 中所有 type 都牵扯进来？）

```
        }
}

/*
 * Remove any permissions prohibited by a constraint (this includes the MLS policy).
 */
constraint = tclass_datum->constraints;
while (constraint) {
        if ((constraint->permissions & (avd->allowed)) &&
            !constraint_expr_eval(scontext, tcontext, NULL, constraint->expr)) {
                avd->allowed &= ~(constraint->permissions);
        }
        constraint = constraint->next;
}
```

遍历 class_datum->constraints 队列中的每一个约束，如果当前 scontext，tcontext 和约束的条件相符，则从 avd->allowed 位图中删除哪些被该约束所禁止的能力。

```
/*
 * If checking process transition permission and the
 * role is changing, then check the (current_role, new_role) pair.
 */
if (tclass == policydb.process_class &&                         # 进程类
    (avd->allowed & policydb.process_trans_perms) &&            # 进程类的 transition 权限
    scontext->role != tcontext->role) {                         # Domain transition 前后 role 改变
        for (ra = policydb.role_allow; ra; ra = ra->next) {
                if (scontext->role == ra->role && tcontext->role == ra->new_role)
                        break;
        }
        if (!ra)
                avd->allowed &= ~policydb.process_trans_perms;
```

```
        }
```

如果需要发生 domain transition 而且 role 同时也发生改变，则检查是否有许可从原来的 role 变成新 role 的 role-allow 规则。如果没有，则不允许发生 domain transition。

policydb_t.process_trans_perms 设计的惟一目的就是记录 process 类“transition”权限在 process 类内的索引，如果 avd->allowed 位图包含 process_trans_perms 位则说明许可 domain transition，此时进一步遍历 policydb_t.role_allow 队列寻找是否存在一个匹配的 role-allow 规则。如果没有找到，则从 avd->allowed 位图中清除 process_trans_perms 位图。

注意，role-allow 规则即在此处被使用，而 role-transition 规则在 security_compute_sid 中被使用。另外 role-types/attribute/dominance 规则都是在编译时被处理，其结果在运行时 security_compute_sid > policydb_context_isvalid 中检查。

```
        /*
         * If the given source and target types have boundary
         * constraint, lazy checks have to mask any violated
         * permission and notice it to userspace via audit.
         */
        type_attribute_bounds_av(scontext, tcontext, tclass, avd);
}
```

（TODO：暂且跳过有关 type->bounds 相关的内容）

10.4 通过 SELinuxfs 访问内核 Security Server

在 Host 上使用“checkpolicy -Mdb <policy.xx>”命令可以访问用户态的 Security Server（在 libsepol 中实现），而在 Target 上可以借助 libselinux 提供的若干命令，通过 selinuxfs 提供的文件来访问内核态的 Security Server。

SELinuxfs 用户态接口的使用方法可以通过分析 libselinux/src/下的相应文件得到：

```
booleans.c:                     snprintf(fname, len, "%s%s%s", selinux_mnt, SELINUX_BOOL_DIR, name);
booleans.c:                     snprintf(path, sizeof path, "%s/commit_pending_bools", selinux_mnt);
check_context.c:                snprintf(path, sizeof path, "%s/context", selinux_mnt);
compute_av.c:                   snprintf(path, sizeof path, "%s/access", selinux_mnt);
compute_create.c:               snprintf(path, sizeof path, "%s/create", selinux_mnt);
compute_member.c:               snprintf(path, sizeof path, "%s/member", selinux_mnt);
compute_relabel.c:              snprintf(path, sizeof path, "%s/relabel", selinux_mnt);
compute_user.c:                 snprintf(path, sizeof path, "%s/user", selinux_mnt);
deny_unknown.c:                 snprintf(path, sizeof(path), "%s/deny_unknown", selinux_mnt);
disable.c:                      snprintf(path, sizeof path, "%s/disable", selinux_mnt);
enabled.c:                      snprintf(path, sizeof path, "%s/mls", selinux_mnt);
getenforce.c:                   snprintf(path, sizeof path, "%s/enforce", selinux_mnt);
load_policy.c:                  snprintf(path, sizeof path, "%s/load", selinux_mnt);
policyvers.c:                   snprintf(path, sizeof path, "%s/policyvers", selinux_mnt);
setenforce.c:                   snprintf(path, sizeof path, "%s/enforce", selinux_mnt);
```

而 selinuxfs 相应文件的访问方法表及访问权限在 sel_fill_super 函数中指定：

```
        static struct tree_descr selinux_files[] = {
                [SEL_LOAD] = {"load", &sel_load_ops, S_IRUSR|S_IWUSR},
                [SEL_ENFORCE] = {"enforce", &sel_enforce_ops, S_IRUGO|S_IWUSR},
```

```
        [SEL_CONTEXT] = {"context", &transaction_ops, S_IRUGO|S_IWUGO},
        [SEL_ACCESS] = {"access", &transaction_ops, S_IRUGO|S_IWUGO},
        [SEL_CREATE] = {"create", &transaction_ops, S_IRUGO|S_IWUGO},
        [SEL_RELABEL] = {"relabel", &transaction_ops, S_IRUGO|S_IWUGO},
        [SEL_USER] = {"user", &transaction_ops, S_IRUGO|S_IWUGO},
        [SEL_POLICYVERS] = {"policyvers", &sel_policyvers_ops, S_IRUGO},
        [SEL_COMMIT_BOOLS] = {"commit_pending_bools", &sel_commit_bools_ops, S_IWUSR},
        [SEL_MLS] = {"mls", &sel_mls_ops, S_IRUGO},
        [SEL_DISABLE] = {"disable", &sel_disable_ops, S_IWUSR},
        [SEL_MEMBER] = {"member", &transaction_ops, S_IRUGO|S_IWUGO},
        [SEL_CHECKREQPROT] = {"checkreqprot", &sel_checkreqprot_ops, S_IRUGO|S_IWUSR},
        [SEL_REJECT_UNKNOWN] = {"reject_unknown", &sel_handle_unknown_ops, S_IRUGO},
        [SEL_DENY_UNKNOWN] = {"deny_unknown", &sel_handle_unknown_ops, S_IRUGO},
        [SEL_STATUS] = {"status", &sel_handle_status_ops, S_IRUGO},
        [SEL_POLICY] = {"policy", &sel_policy_ops, S_IRUSR},
        /* last one */ {""}
    };
```

10.4.1 /selinux/load 和 load_policy 命令 - 装载冰解析 policy.X 二进制文件

相关函数调用链：sel_write_load > security_load_policy
所要求的权限：security { load_policy }
使用者 domain 所需要调用的接口：selinux_load_policy

注意，目前只对 kernel_t 和 load_policy_t 调用了 selinux_load_policy 接口，它们分别对应系统启动时 init 进程通过 libselinux 函数装载 policy.X，或者在命令行执行/sbin/load_policy 命令装载 policy.X 时的进程 domain。

selinuxfs 为/selinux/load 文件实现的方法表为：

```
static const struct file_operations sel_load_ops = {
        .write          = sel_write_load,
        .llseek         = generic_file_llseek,
};
```

该函数将 buf 中 count 字节的 policy.X 文件由用户态拷贝到内核中，解析 policy.X 的内容创建 policydb 数据结构并建立所有策略的内核描述符。比如在下面的函数调用链中：

```
sel_write_load > security_load_policy > policy_read > type_read
```

在 policy_read 中通过如下表格驱动从 policy.X 中读出以一定格式存储的策略实现，然后在相应的 xxx_read 方法中解析、创建相应的内核描述符 xxx_datum 数据结构，最后注册到 policydb 的相应的 symtab 中：

```
        for (i = 0; i < info->sym_num; i++) {                           # 逐一创建各个 symtab
                rc = next_entry(buf, fp, sizeof(u32)*2);
                if (rc < 0)
                        goto bad;
                nprim = le32_to_cpu(buf[0]);
                nel = le32_to_cpu(buf[1]);                              # 相应语法元素的总数
                for (j = 0; j < nel; j++) {                             # 逐一读出各个语法元素
                        rc = read_f[i](p, p->symtab[i].table, fp);
                        if (rc)
                                goto bad;
                }
```

```
                p->symtab[i].nprim = nprim;
        }
```

注意这里有两层循环：外层循环按照先后顺序逐一处理 8 个符号表（symtab）；内层循环读取每一种符号表中各个符号的二进制描述。对于 type 规则而言，解析函数为 type_read：

```
static int type_read(struct policydb *p, struct hashtab *h, void *fp)
{
        char *key = NULL;
        struct type_datum *typdatum;
        int rc, to_read = 3;
        __le32 buf[4];
        u32 len;

        typdatum = kzalloc(sizeof(*typdatum), GFP_KERNEL);
        if (!typdatum) {
                rc = -ENOMEM;
                return rc;
        }
```

首先分配一个 type_datum 数据结构。不同版本的 policy.X 的组织结构略有差异。如果版本号>=24，则 policy.X 中定义的每个 type 将由 4 个长字描述，因此调整读取的长字数量：

```
        if (p->policyvers >= POLICYDB_VERSION_BOUNDARY)
                to_read = 4;

        rc = next_entry(buf, fp, sizeof(buf[0]) * to_read);
        if (rc < 0)
                goto bad;

        len = le32_to_cpu(buf[0]);
        typdatum->value = le32_to_cpu(buf[1]);
        if (p->policyvers >= POLICYDB_VERSION_BOUNDARY) {
                u32 prop = le32_to_cpu(buf[2]);

                if (prop & TYPEDATUM_PROPERTY_PRIMARY)
                        typdatum->primary = 1;
                if (prop & TYPEDATUM_PROPERTY_ATTRIBUTE)
                        typdatum->attribute = 1;

                typdatum->bounds = le32_to_cpu(buf[3]);
        } else {
                typdatum->primary = le32_to_cpu(buf[2]);
        }
```

首先从 policy.X 中读取 4 个长字，分别描述 type 字符串的长度，编译器给该 type 分配的数值编号，属性，bounds 数值。从对 le32_to_cpu 函数的调用可见 policy.X 是以小头（Little Endian）方式保存的。

```
        key = kmalloc(len + 1, GFP_KERNEL);
        if (!key) {
                rc = -ENOMEM;
                goto bad;
        }
        rc = next_entry(key, fp, len);
        if (rc < 0)
                goto bad;
        key[len] = '\0';
```

紧随其后的就是 type 字符串本身了，分配所需空间并从 policy.X 中读取该字符串。最后，向 p_types 符号表中注册（如果存在则出错返回，否则分配一个 hashtab_node 数据结构并插入相应的冲突队列）：

```
        rc = hashtab_insert(h, key, typdatum);
        if (rc)
                goto bad;
out:
        return rc;
bad:
        type_destroy(key, typdatum, NULL);
        goto out;
}
```

10.4.2 /selinux/relabel 及 compute_relabel 命令 - 查询 type_change 规则

相关函数调用链：sel_write_relabel > security_change_sid > security_compute_sid
所要求的权限：security { compute_relabel }
使用者 domain 所需要调用的接口：selinux_compute_relabel_context

pam_selinux.so 模块被系统登录服务进程所调用，用于在用户登录时 relabel 其 controlling terminal 的标签和用户的身份相适应。pam_selinux.so 通过/selinux/relabel 文件获得登录设备的新的安全上下文。该过程可以通过 compute_relabel 命令来复现：

```
[root/sysadm_r/s0@~]# id -Z
root:sysadm_r:sysadm_t:s0-sl5:c0.cl023
[root/sysadm_r/s0@~]# tty
/dev/console
[root/sysadm_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_tty_device_t:s0 /dev/console
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# matchpathcon `tty`
/dev/console    system_u:object_r:console_device_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_relabel `id -Z`  system_u:object_r:console_device_t:s0 chr_file
root:object_r:user_tty_device_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s sysadm_t -t console_device_t -c chr_file"
Password:
Found 1 semantic te rules:
ET type_change sysadm_t console_device_t : chr_file user_tty_device_t; [ console_login ]
[root/sysadm_r/s0@~]#
```

如上所示，/dev/console 的默认标签为 console_device_t，在 sysadm 用户登录后它被 relabel 为 user_tty_device_t。该结果和用 sesearch 得到的相同。

10.4.3 /selinux/create 及 compute_create 命令 - 查询 type_transition 规则

相关函数调用链：sel_write_create > security_transition_sid_user > security_compute_sid
所要求的权限：security { compute_create }
使用者 domain 所需要调用的接口：selinux_compute_create_context

通过/selinux/create 文件和 compute_create 命令可以向 policy.X 查询和参数 sid pair 相关的 type_transition 规则，其返回结果和用 sesearch 得到的相同。比如：

```
[root/sysadm_r/s0@~]# compute_create system_u:system_r:syslogd_t:s15:c0.c1023
system_u:system_r:syslogd_t:s15:c0.c1023 unix_dgram_socket
system_u:system_r:syslogd_s_t:s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SC --type -s syslogd_t -c unix_dgram_socket"
Password:
Found 1 semantic te rules:
   type_transition syslogd_t syslogd_t : unix_dgram_socket syslogd_s_t;
[root/sysadm_r/s0@~]#
```

由此可见当 syslogd_t:mls_systemhigh 在创建 unix_dgram_socket 类对象时，其默认的标签为 syslogd_s_t，而不是继承创建者的标签，正是因为存在对应的 type_transition 规则。

否则，socket 将默认继承创建者的安全上下文：

```
[root/sysadm_r/s0@~]# compute_create root:sysadm_r:sysadm_t:s0-s15:c0.c1023 root:sysadm_r:sysadm_t:s0-
s15:c0.c1023 unix_dgram_socket
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SC --type -s sysadm_t -t sysadm_t -c unix_dgram_socket"
Password:
[root/sysadm_r/s0@~]# seclow "sesearch -SC --allow -s sysadm_t -t sysadm_t -c unix_dgram_socket -p create"
Password:
Found 2 semantic av rules:
   allow sysadm_t sysadm_t : unix_dgram_socket { ioctl read write create getattr setattr append bind
connect getopt setopt shutdown sendto } ;
DT allow sysadm_t sysadm_t : unix_dgram_socket { ioctl read write create getattr setattr append bind
connect getopt setopt shutdown } ; [ init_upstart ]
[root/sysadm_r/s0@~]#
```

由此可见，sysadm_t 具备对 self:unix_dgram_socket 对象的 create 能力，在没有相应 type_transition 规则的情况下该对象的 type 继承创建者 domain。

注意，任何情况下 socket 对象的 role 和 MLS 属性都必须和创建者保持一致，参见上文“Separate socket type 开发”的相应内容。

10.4.4 /selinux/member 及 compute_member 命令 - 查询 type_member 规则

相关函数调用链：sel_write_member > security_member_sid > security_compute_sid
所要求的权限：security { compute_member }
使用者 domain 所需要调用的接口：selinux_compute_member

如果使能多态，则系统登录设施比如 login/sshd/crond 在用户登录时将在 parentdir 下为 polydir 创建相应的 memberdir，并 relabel 为新的标签。这些操作由 pam_namespace.so 完成，可以用 compute_member 命令复现如下：

```
[root/sysadm_r/s0@~]# getenforce
Permissive
[root/sysadm_r/s0@~]# ls -Zd /tmp
drwxrwxrwt  root root system_u:object_r:tmp_t:s0-s15:c0.c1023 /tmp
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_member `id -Z` system_u:object_r:tmp_t:s0-s15:c0.c1023 dir
system_u:object_r:user_tmp_t:s0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCT -s sysadm_t -t tmp_t -c dir"
```

```
Password:
Found 2 semantic te rules:
   type_member sysadm_t tmp_t : dir user_tmp_t;
   type_transition sysadm_t tmp_t : dir user_tmp_t;
[root/sysadm_r/s0@~]#
```

注意，只有登录设施所在的 domain（比如 sshd_t 或 local_login_t）在 allow_polyinstantiation 为 true 时具备 compute_member 权限，因此在命令行直接调用 compute_member 需要在 Permissive 模式下进行。

10.4.5 /selinux/access 文件和 compute_av 命令 - 查询 allow 规则

相关函数调用链：sel_write_access > security_compute_av_user > context_struct_compute_av
所要求的权限：security { compute_av }
使用者 domain 所需要调用的接口：selinux_compute_access_vector

通过/selinux/access 文件和 compute_av 命令均可以向 policy.X 查询 scontext 对 tcontext 的相关规则，和使用 sesearch 命令查询到的结果相同：

```
[root/sysadm_r/s0@~]# compute_av root:sysadm_r:cronjob_t:s0-s15:c0.c1023 root:object_r:user_cron_spool_t:s0
file
allowed= { entrypoint }
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s cronjob_t -t user_cron_spool_t -c file"
Password:
Found 1 semantic av rules:
   allow cronjob_t user_cron_spool_t : file entrypoint ;
[root/sysadm_r/s0@~]#
```

10.4.6 /selinux/user 文件和 compute_user 命令 - 查询用户登录后可能的 SC

相关函数调用链：sel_write_user > security_get_user_sids
所要求的权限：security { compute_user }
使用者 domain 所需要调用的接口：selinux_compute_user_contexts

通过/selinux/user 文件和 compute_user 命令都可以向 policy.X 查询从一个“source context”开始，指定的 user 能够到达的“user context”集合。比如：

```
[root/sysadm_r/s0@~]# compute_user
usage:  compute_user context user
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps -eZ | grep login
system_u:system_r:local_login_t:s0-s15:c0.c1023 1359 ? 00:00:00 login
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_user system_u:system_r:local_login_t:s0-s15:c0.c1023 root
root:auditadm_r:auditadm_t:s0-s15:c0.c1023
root:staff_r:staff_t:s0-s15:c0.c1023
root:secadm_r:secadm_t:s0-s15:c0.c1023
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps -eZ | grep ssh
system_u:system_r:sshd_t:s0-s15:c0.c1023 1288 ? 00:00:00 sshd
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_user system_u:system_r:sshd_t:s0-s15:c0.c1023 root
root:auditadm_r:auditadm_t:s0-s15:c0.c1023
```

```
root:staff_r:rssh_t:s0-s15:c0.c1023
root:staff_r:staff_t:s0-s15:c0.c1023
root:secadm_r:secadm_t:s0-s15:c0.c1023
root:sysadm_r:rssh_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps -eZ | grep cron
system_u:system_r:crond_t:s0-s15:c0.c1023 1341 ? 00:00:01 crond
system_u:system_r:crond_t:s0-s15:c0.c1023 1351 ? 00:00:00 atd
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# compute_user system_u:system_r:crond_t:s0-s15:c0.c1023 root
root:staff_r:cronjob_t:s0-s15:c0.c1023
root:sysadm_r:cronjob_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
```

类似地，可以使用 getseuser 命令获得一个 Linux User 在某种设备上登录后可能扮演的 SC。如果有多个选择，则可进一步使用 getdefaultcon 命令得到默认的 SC。比如：

```
[root/sysadm_r/s0@~]# getseuser
usage:  getseuser linuxuser fromcon
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# getseuser root system_u:system_r:local_login_t:s0-s15:c0.c1023
seuser:  root, level (null)
Context 0       root:sysadm_r:sysadm_t:s0-s15:c0.c1023
Context 1       root:staff_r:staff_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# getdefaultcon root system_u:system_r:local_login_t:s0-s15:c0.c1023
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#

[root/sysadm_r/s0@~]# getseuser root system_u:system_r:sshd_t:s0-s15:c0.c1023
seuser:  root, level (null)
Context 0       root:staff_r:staff_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# getdefaultcon root system_u:system_r:sshd_t:s0-s15:c0.c1023
root:staff_r:staff_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#

[root/sysadm_r/s0@~]# getseuser root system_u:system_r:crond_t:s0-s15:c0.c1023
seuser:  root, level (null)
Context 0       root:sysadm_r:cronjob_t:s0-s15:c0.c1023
Context 1       root:staff_r:cronjob_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# getdefaultcon root system_u:system_r:crond_t:s0-s15:c0.c1023
root:sysadm_r:cronjob_t:s0-s15:c0.c1023
[root/sysadm_r/s0@~]#
```

10.4.7 /selinux/initial_contexts/ - 查询 Initial SID 对应的安全上下文

/selinux/initial_contexts/目录数下导出了 27 个文件，文件名即为各个 Initial SID 的名称字符串（和 refpolicy/policy/flask/initial_sids 文件中的 SID 声明相一致），文件内容即为该 SID 所对应的安全上下文字符串（和相关 pp 中的 SID 定义相一致）：

```
[root/sysadm_r/s0@~]# cd /selinux/initial_contexts/
[root/sysadm_r/s0@initial_contexts]# ls
any_socket   icmp_socket  netif   scmp_packet  sysctl_kernel    tcp_socket
devnull      igmp_packet  netmsg  security     sysctl_modprobe  unlabeled
file         init         node    sysctl       sysctl_net
```

```
file_labels  kernel       policy  sysctl_dev   sysctl_net_unix
fs           kmod         port    sysctl_fs    sysctl_vm
[root/sysadm_r/s0@initial_contexts]#
[root/sysadm_r/s0@initial_contexts]# cat kernel
system_u:system_r:kernel_t:sl5:c0.cl023
[root/sysadm_r/s0@initial_contexts]#
```

注意这些文件用于导出内核 policydb 数据结构中和 Initial SID 相关的 ocontexts 数据结构的内容，而这些数据结构正是在装载、解析 policy.X 时被创建的。也可以用 seinfo 命令直接读取 policy.X 中的相关内容，比如：

```
[root/sysadm_r/s0@initial_contexts]# seclow "seinfo --initialsid=kernel -x"
Password:
             kernel:  system_u:system_r:kernel_t:sl5:c0.cl023
[root/sysadm_r/s0@initial_contexts]#
```

显然它们的结果是一样的。

libselinux 定义的 security_get_initial_context 函数读取该目录数下相应文件的内容。

10.4.8 /selinux/class/ - 查询内核 class_datum 数据结构 (todo)

```
/selinux/class/xxx/index          class policy value
/selinux/class/xxx/perms/xxx      permission policy value
```

而内核代码中使用的 class number，permission number，在 flask.h/av_permissions.h 中定义，由 genheaders 程序根据 classmap.h 生成，为 kernel value。

10.5 情景分析：Domain transition 的实现

一个用户可以使用“newrole -r”命令切换到新的角色，同时切换到新角色的默认 domain 中。比如：

```
newrole -r secadm_r -p
```

当前 shell 进程的 domain 为 sysadm_t，在运行 newrole 程序期间进入 newrole_t。newrole 程序从 contexts/default_type 文件获得 secadm_r 的默认 domain 为 secadm_t，然后 fork 子进程，后者调用 libselinux 的 setexeccon 函数将新的 domain（secadm_t）写入子进程的/proc/pid/attr/exec 文件，然后通过 exec 函数运行当前用户的登录 shell。可以通过 ssh 再次登录系统，观察在 ttyS0 上登录并执行 newrole 命令的相关进程的状态：

```
[root/sysadm_r/s0@~]# tty
/dev/pts/0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps axj | grep login
    1  1357  1357  1357 ?           -1 Ss       0   0:00 login -- root
 1448  1507  1506  1400 pts/0     1506 S+       0   0:00 grep login
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps axj | grep ttyS0
 1357  1373  1373  1373 ttyS0     1424 Ss       0   0:00 -bash
 1373  1420  1420  1373 ttyS0     1424 S        0   0:00 newrole -r secadm_r -p
 1420  1424  1424  1373 ttyS0     1424 S+       0   0:00 -/bin/bash
 1448  1513  1512  1400 pts/0     1512 S+       0   0:00 grep ttyS0
```

```
[root/sysadm_r/s0@~]#
```

login 后台进程（1357）创建子进程（1273）执行用户的登录 shell（bash），由于 newrole 命令为 shell 的外部命令，所以它继续创建子进程（1420）来执行 newrole 命令，newrole 进程将再次创建子进程，以再次执行当前用户的登录 shell（还是 bash），只不过运行在和 secadm_r 相对应的默认 domain secadm_t 中。相关进程的 SC 如下：

```
[root/sysadm_r/s0@~]# ps -eZ | grep login
system_u:system_r:local_login_t:s0-s15:c0.c1023 1357 ? 00:00:00 login
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ps -eZ | grep ttyS0
root:sysadm_r:sysadm_t:s0-s15:c0.c1023          1373 ttyS0 00:00:00 bash
root:sysadm_r:newrole_t:s0-s15:c0.c1023         1420 ttyS0 00:00:00 newrole
root:secadm_r:secadm_t:s0-s15:c0.c1023          1424 ttyS0 00:00:00 bash
[root/sysadm_r/s0@~]#
```

问题：newrole 进程所创建的子进程（1424）在 exec 执行 shell 期间，内核如何设置其 domain 为 secadm_t 的?

10.5.1 selinux_setprocattr 函数 - /proc/pid/attr/*文件驱动

通常情况下进程执行 exec 时发生的 domain transition 由相应的 type_transition 规则决定，而应用程序能够通过写入/proc/<pid>/attr/exec 文件重载它。用户态应用程序调用 libselinux 的 setexeccon 函数向/proc/<pid>/attr/exec 文件写入下一次执行 exec 函数时当前进程切换到的新 domain。在 fs/base.c 中定义/proc/<pid>/attr/*文件的读写方法：

```
static const struct file_operations proc_pid_attr_operations = {
        .read           = proc_pid_attr_read,
        .write          = proc_pid_attr_write,
};
```

其中 proc_pid_attr_write 函数的主要操作就是参数检查，然后分配一个内核页框并从用户地址空间中拷贝传递给 setexeccon 函数的参数，最终调用 security_setprocattr 函数设置当前进程的 task_security_struct 数据结构中的 exec_sid 域：

```
        length = security_setprocattr(task, (char*)file->f_path.dentry->d_name.name, (void*)page, count);
```

LSM 框架的 security_setprocattr 函数转而调用 LSM 具体实现的 setprocattr 函数：

```
int security_setprocattr(struct task_struct *p, char *name, void *value, size_t size)
{
        return security_ops->setprocattr(p, name, value, size);
}
```

就 SELinux 而言即为 selinux_setprocattr 函数，它的参数分别为当前进程 task_struct 的指针，/proc/pid/attr/*文件的名字字符串，新 SC 字符串指针及其长度：

```
static int selinux_setprocattr(struct task_struct *p, char *name, void *value, size_t size)
{
        struct task_security_struct *tsec;
        struct task_struct *tracer;
        u32 sid = 0;
        int error;
        char *str = value;
```

```
if (current != p) {
        /* SELinux only allows a process to change its own security attributes. */
        return -EACCES;
}
```

SELinux 只允许一个进程改变自己的 domain，如果和/proc/pid/attr/exec 文件相关的进程不是当前进程，则出错退出。接着检查当前进程是否能够设置 p 所指的目的进程的/proc/pid/attr/*文件：

```
/*
 * Basic control over ability to set these attributes at all.
 * current == p, but we'll pass them separately in case the
 * above restriction is ever removed.
 */
if (!strcmp(name, "exec"))
        error = task_has_perm(current, p, PROCESS__SETEXEC);
else if (!strcmp(name, "fscreate"))
        error = task_has_perm(current, p, PROCESS__SETFSCREATE);
else if (!strcmp(name, "keycreate"))
        error = task_has_perm(current, p, PROCESS__SETKEYCREATE);
else if (!strcmp(name, "sockcreate"))
        error = task_has_perm(current, p, PROCESS__SETSOCKCREATE);
else if (!strcmp(name, "current"))
        error = task_has_perm(current, p, PROCESS__SETCURRENT);
else
        error = -EINVAL;
if (error)
        return error;
```

SELinux 为每个/proc/pid/attr/*文件都设计了相应的权限，只有当前进程具备相应的权限时才能执行相应的操作。比如调用 setexeccon 函数来设置/proc/pid/attr/exec 文件的进程就必须具备 PROCESS_SETEXEC 能力，即为 process 类的 setexec 权限。注意，任意进程的/proc/pid/attr/目录树的 SC 即为进程自身的 SC。所以必须为相应 domain 定义如下规则：

```
allow xxx self:process setexec;
```

比如在 system/selinuxutil.te 中为 newrole_t 明确定义如下规则：

```
allow newrole_t self:process setexec;
```

task_has_perm 函数为 avc_has_perm 的封装函数：

```
return avc_has_perm(tsec1->sid, tsec2->sid, SECCLASS_PROCESS, perms, NULL);
```

它查询 tsec1->sid 对 tsec2->sid 是否具有 SECCLASS_PROCESS 类的相应权限。注意上面的检查已经确保 tsec1 和 tsec2 是相同的。

```
/* Obtain a SID for the context, if one was specified. */
if (size && str[1] && str[1] != '\n') {         # 如果 value 字符串有意义
        if (str[size-1] == '\n') {              # 如果以换行符结束，则替换以 NULL 结束
                str[size-1] = 0;
                size--;
        }
        error = security_context_to_sid(value, size, &sid);
        if (error == -EINVAL && !strcmp(name, "fscreate")) {
                if (!capable(CAP_MAC_ADMIN))
```

```
                return error;
        error = security_context_to_sid_force(value, size, &sid);
    }
    if (error)
        return error;
}
```

上面这段代码调用 security_context_to_sid 函数解析 SC 字符串的各个成员，得到相应 SELinux 语法成分在内核中的 xxx_datum 描述符，设置并向 sidtab 注册一个 context 数据结构，最终返回相应的 sid（详见下文）。下面就可以把它记录到 tsec 的相应域中：

```
    /* Permission checking based on the specified context is
       performed during the actual operation (execve,
       open/mkdir/...), when we know the full context of the
       operation.  See selinux_bprm_set_security for the execve
       checks and may_create for the file creation checks. The
       operation will then fail if the context is not permitted. */
    tsec = p->security;
    if (!strcmp(name, "exec"))
        tsec->exec_sid = sid;
    else if (!strcmp(name, "fscreate"))
        tsec->create_sid = sid;
    else if (!strcmp(name, "keycreate")) {
        error = may_create_key(sid, p);
        if (error)
            return error;
        tsec->keycreate_sid = sid;
    } else if (!strcmp(name, "sockcreate"))
        tsec->sockcreate_sid = sid;
    else if (!strcmp(name, "current")) {
        struct av_decision avd;

        if (sid == 0)
            return -EINVAL;
        /*
         * SELinux allows to change context in the following case only.
         *  - Single threaded processes.
         *  - Multi threaded processes intend to change its context into
         *    more restricted domain (defined by TYPEBOUNDS statement).
         */
        ...
        ...
    } else
        return -EINVAL;

    return size;
}
```

至此，SELinux 内核驱动对 setexeccon 函数的处理完成，当前进程下一次执行 exec 函数时所切换到的新 domain 的信息被写入当前进程的 tsec->exec_sid 域中，而如何利用它来进一步设置 tsec->sid，就是 do_execve 及相关 SELinux 内核驱动的任务了。

10.5.2 do_execve 的行为和相关 SELinux 内核驱动

1，do_execve 函数处理 sys_execve 系统调用，它分配一个 linux_binprm 数据结构，用于描述当前进程即将装载的可执行文件的信息，比如可执行文件的名称 filename，该文件首部 128 字节的信息（即为

elf32_hdr/elf64_hdr 数据结构），指向文件的 file 数据结构的指针，当前进程在执行该文件时的 e_uid 和 e_gid 信息，包含命令行参数和环境变量字符串的页面等。

2，do_execve > security_brpm_alloc = selinux_bprm_alloc_security 函数给 linux_binprm 数据结构分配相应的 bprm_security_struct 数据结构，其中 set 标志表示该数据结构是否已经被设置过，sid 表示当前进程装载该可执行文件后的 domain，unsafe 用于在 bprm_apply_creds 和 bprm_post_apply_creds 函数之间传递错误信息。

3，do_execve > prepare_binprm 函数设置 bprm->e_uid/e_gid 信息。如果可执行文件所在的文件系统在挂载时没有设置 MNT_NOSUID 标志，即表示允许使用该文件系统中的 setuid/setgid 程序，则如果该文件的 S_ISUID 标志位有效，则设置 brpm->e_uid = inode->i_uid，即在执行该文件期间当前进程的 e_uid 变成该文件属主的 i_uid。

prepare_binprm 函数会调用 security_bprm_set = selinux_bprm_set_security 函数，其终极目标是确定 bsec->sid = newsid，要么等于 tsec->exec_sid，要么为默认 domain transition 相关的新的 domain。同时做 SELinux 权限检查：如果 tsec->sid 和 newsid 相等，即不发生 domain transition，则 tsec->sid 对可执行文件（isec->sid）必须具有 FILE_EXECUTE_NO_TRANS 能力。否则 tsec->sid 对 newsid 必须具有 PROCESS_TRANSITION 能力，同时 newsid 对可执行文件（isec->sid）具有 FILE_ENTRYPOINT 能力。参见下文。

4，do_execve > search_binary_handler 遍历 format 链表，调用每种格式的 load_binary 函数。如果某种格式的驱动能够“认领”该可执行文件，则成功返回。

elf_format.load_elf_binary 函数销毁当前进程的全部地址空间：释放 mm_struct 以及所有的 vma_area_struct 数据结构，并清除页表。然后通过 do_mmap 函数建立对可执行文件代码段、数据段，以及 loader 代码段和数据段的私有文件映射，建立栈相关的线性区并拷贝命令行参数和环境变量参数，设置堆等。

5，load_elf_binary > compute_creds > security_bprm_apply_creds = selinux_bprm_apply_creds 函数设置 tsec->osid = tsec->sid。如果 tsec->sid 和 bsec->sid 不同，则设置 tsec->sid = bsec->sid。至此使得当前进程在 exec 可执行文件时改变其 domain。另外，如果当前进程被其他进程跟踪，则还需要检查 tracer 进程的 domain 对新的 domain（bsec->sid）是否具有 PROCESS_PTRACE 能力，如果没有则设置 bsec->unsafe=1。

6，load_elf_binary > compute_creds > security_bprm_post_apply_creds = selinux_bprm_post_apply_creds 函数检查 bsec->unsafe 是否被设置，如果是，则给当前进程发送 sigkill 信号并退出（使得当前进程无法顺利执行 exec 系统调用）。如果不需要发生 domain 切换（tsec->osid == tsec->sid）则直接退出。否则就新的 domain，关闭它无法访问的从父进程继承的打开文件描述符（调用 inode_has_perm 和 file_has_perm 函数）。检查当前进程新的 domain 是否对其父进程具有 siginh 能力（继承信号状态），如果否则 flush 当前未决信号并 unblock 所有信号。最后判断新 domain 对原有 domain 是否具有 rlimitinh 能力（继承资源 limits.conf 限制），如果没有则把当前进程所有 limits 的 soft 值设置为当前进程 hard limits 和 init 进程 soft limits 的较小值。

selinux_bprm_set_security 函数的目的就是设置 linux_binprm 中的 brpm_security_struct 数据结构，主要是确定 bsec->sid，并进行相应的权限检查。

```
static int selinux_bprm_set_security(struct linux_binprm *bprm)
{
        struct task_security_struct *tsec;
        struct inode *inode = bprm->file->f_path.dentry->d_inode;      # 指向可执行程序文件的 inode
        struct inode_security_struct *isec;
```

```
struct bprm_security_struct *bsec;
u32 newsid;
struct avc_audit_data ad;
int rc;

rc = secondary_ops->bprm_set_security(bprm);                    # 此步骤暂时跳过
if (rc)
        return rc;

bsec = bprm->security;
if (bsec->set)
        return 0;
```

至此，bsec 指向和当前 linux_binprm 数据结构相关的 bprm_security_struct 数据结构。如果其中 set 位已被设置，则直接退出。

```
tsec = current->security;                                       # 指向当前进程的 task_security_task
isec = inode->i_security;                                       # 指向可执行程序文件的 inode_security_task

/* Default to the current task SID. */
bsec->sid = tsec->sid;
```

使用 bsec->sid 描述当前进程 exec 可执行文件后新的 domain，默认认为不发生 domain 切换。注意子进程的 tsec->sid 继承于父进程的 tsec->sid（可参考 fork 的相关代码）。

```
/* Reset fs, key, and sock SIDs on execve. */
tsec->create_sid = 0;
tsec->keycreate_sid = 0;
tsec->sockcreate_sid = 0;
```

子进程在执行 exec 时，清除所有的 create_sid 位。

```
if (tsec->exec_sid) {
        newsid = tsec->exec_sid;
        /* Reset exec SID on execve. */
        tsec->exec_sid = 0;
} else {
        /* Check for a default transition on this program. */
        rc = security_transition_sid(tsec->sid, isec->sid, SECCLASS_PROCESS, &newsid);
        if (rc)
                return rc;
}
```

如果此时 tsec->exec_sid 不为 0，则说明当前进程之前已经指定了新的 domain，把它保存在 newsid 中并清除 exec_sid。否则，应该向 policy.X 查询当前进程在执行相应的程序时相关的 type_transition 规则，返回匹配规则所指定的新 domain。

```
AVC_AUDIT_DATA_INIT(&ad, FS);
ad.u.fs.path = bprm->file->f_path;              # 跳过该部分

if (bprm->file->f_path.mnt->mnt_flags & MNT_NOSUID)
        newsid = tsec->sid;
```

如果可执行程序所在的文件系统在挂载时指定了 MNT_NOSUID 标志，则禁止 domain transition?

```
if (tsec->sid == newsid) {
```

```
                rc = avc_has_perm(tsec->sid, isec->sid, SECCLASS_FILE, FILE__EXECUTE_NO_TRANS, &ad);
                if (rc)
                        return rc;
        }
```

如果不需要发生 domain transition，则检查当前进程 domain 对相应的可执行程序 type 是否具有 execute_no_trans 权限。（avc_has_perm 函数参见上文）

否则就需要发生 domain transition 了。首先检查是否许可 domain transition，然后检查新 domain 是否以当前可执行程序为其 entrypoint。

```
        else {
                /* Check permissions for the transition. */
                rc = avc_has_perm(tsec->sid, newsid, SECCLASS_PROCESS, PROCESS__TRANSITION, &ad);
                if (rc)
                        return rc;

                rc = avc_has_perm(newsid, isec->sid, SECCLASS_FILE, FILE__ENTRYPOINT, &ad);
                if (rc)
                        return rc;

                /* Clear any possibly unsafe personality bits on exec: */
                current->personality &= ~PER_CLEAR_ON_SETID;

                /* Set the security field to the new SID. */
                bsec->sid = newsid;
        }
```

如果检查都通过，则把新 domain 的 SID 记录在 bsec->sid 中，即为当前进程执行该文件期间的 tsec->sid。最后设置 bsec->set 为 1。

```
        bsec->set = 1;
        return 0;
}
```

10.6 情景分析：文件系统的挂载和新文件的创建

可以通过如下方法确定新创建文件的标签：
1，在应用程序源代码中调用 setfscreatecon 函数来直接指定；
2，通过 type_transition 规则明确指定动态创建文件的标签（从而不继承父目录的标签）；
3，在挂载文件系统时指定整个文件系统的标签；

问题：上述机制是如何实现的?

必须从新创建文件所在文件系统的挂载过程说起，分析超级块安全属性中各个 sid 域的确定方法。superblock_security_struct 数据结构的定义参见 10.2.1.4 小节。

10.6.1 文件系统的挂载过程（new）

mount 系统调用的型构定义如下：

```
SYSCALL_DEFINE5(mount, char __user *, dev_name,              # 被挂载文件系统所在设备的路径
                       char __user *, dir_name,              # 挂载点的路径
```

```
                char __user *, type,                    # 文件系统的类型
                unsigned long, flags,                   # 使用“-o”选项传递的标志
                void __user *, data)                    # 用户指定的其他参数，称为“mount data”
```

各个参数的含义如注释所示。将用户态的字符串拷贝到内核中后，随即调用 do_mount 函数：

```
        ret = do_mount(kernel_dev, kernel_dir, kernel_type, flags, (void *)data_page);

long do_mount(char *dev_name, char *dir_name, char *type_page, unsigned long flags, void *data_page)
{
        struct path path;
        int retval = 0;
        int mnt_flags = 0;

        /* Discard magic */
        if ((flags & MS_MGC_MSK) == MS_MGC_VAL)
                flags &= ~MS_MGC_MSK;

        /* Basic sanity checks */
        if (!dir_name || !*dir_name || !memchr(dir_name, 0, PAGE_SIZE))
                return -EINVAL;
```

检查挂载点路径是一个有内容的字符串。

```
        if (data_page)
                ((char *)data_page)[PAGE_SIZE - 1] = 0;
```

如果用户态指定了 mount data 字符串，则确保包含它的页面以 0 结尾。

```
        /* ... and get the mountpoint */
        retval = kern_path(dir_name, LOOKUP_FOLLOW, &path);
        if (retval)
                return retval;
```

调用 kern_path 函数执行路径名的解析，得到挂载点目录的 path 数据结构：

```
struct path {
        struct vfsmount *mnt;
        struct dentry *dentry;
};
```

其中 mnt 指向挂载点目录所在文件系统的 vfsmount 结构；dentry 指向挂载点目录的 dentry 数据结构。

```
        retval = security_sb_mount(dev_name, &path, type_page, flags, data_page);
        if (retval)
                goto dput_out;
```

调用 security_sb_mount 函数，检查当前进程是否具有对挂载点目录的 file 类的 mounton 能力。参见下文。

```
        /* Default to relatime unless overriden */
        if (!(flags & MS_NOATIME))
                mnt_flags |= MNT_RELATIME;

        /* Separate the per-mountpoint flags */
        if (flags & MS_NOSUID)
                mnt_flags |= MNT_NOSUID;
```

```
        if (flags & MS_NODEV)
                mnt_flags |= MNT_NODEV;
        if (flags & MS_NOEXEC)
                mnt_flags |= MNT_NOEXEC;
        if (flags & MS_NOATIME)
                mnt_flags |= MNT_NOATIME;
        if (flags & MS_NODIRATIME)
                mnt_flags |= MNT_NODIRATIME;
        if (flags & MS_STRICTATIME)
                mnt_flags &= ~(MNT_RELATIME | MNT_NOATIME);
        if (flags & MS_RDONLY)
                mnt_flags |= MNT_READONLY;

        flags &= ~(MS_NOSUID | MS_NOEXEC | MS_NODEV | MS_ACTIVE | MS_BORN |
                   MS_NOATIME | MS_NODIRATIME | MS_RELATIME| MS_KERNMOUNT |
                   MS_STRICTATIME);
```

将 mount 参数的某些内容，清空并保存到 mnt_flags 域中（TODO：为什么要做如此的区分？）。

```
        if (flags & MS_REMOUNT)         # -o remount，更新挂载标志
                retval = do_remount(&path, flags & ~MS_REMOUNT, mnt_flags, data_page);
        else if (flags & MS_BIND)       # -o bind，使用 loopback 设备挂载普通文件，或使一个目录有多个访问点
                retval = do_loopback(&path, dev_name, flags & MS_REC);
        else if (flags & (MS_SHARED | MS_PRIVATE | MS_SLAVE | MS_UNBINDABLE))
                retval = do_change_type(&path, flags);          # sharedsubtree
        else if (flags & MS_MOVE)       # --move，更换挂载点
                retval = do_move_mount(&path, dev_name);
        else
                retval = do_new_mount(&path, type_page, flags, mnt_flags, dev_name, data_page);
```

根据 mount 参数调用不同的函数处理，各种参数的含义参见 mount 手册页。对于第一次挂载的文件系统而言，调用 do_new_mount 函数。

```
dput_out:
        path_put(&path);
        return retval;
}
```

selinux_mount 函数检查当前执行流是否对挂载点目录具有 file 类的 mounton 能力；如果执行 remount 操作，则还需要检查是否对相应文件系统具有 filesystem 类的 remount 能力。

```
static int selinux_mount(char *dev_name, struct path *path, char *type, unsigned long flags, void *data)
{
        const struct cred *cred = current_cred();

        if (flags & MS_REMOUNT)
                return superblock_has_perm(cred, path->mnt->mnt_sb, FILESYSTEM__REMOUNT, NULL);
        else
                return path_has_perm(cred, path, FILE__MOUNTON);
}
```

do_new_mount 函数挂载一个新的文件系统，并将其 vfsmount 数据结构加入当前进程的 namespace。参数 path 指向挂载点的 path 数据结构，type 为待挂载文件系统的类型，flags/mnt_flags 为传递给 mount 的标志，name 指向设备路径字符串，data 指向的内核页面包含“mount data”，比如“context=”等。

```
/*
 * create a new mount for userspace and request it to be added into the namespace's tree
 */
```

```
static int do_new_mount(struct path *path, char *type, int flags, int mnt_flags, char *name, void *data)
{
        struct vfsmount *mnt;
        int err;

        if (!type)
                return -EINVAL;

        /* we need capabilities... */
        if (!capable(CAP_SYS_ADMIN))                    # DAC 关于 capability 的检查
                return -EPERM;
```

执行 mount 操作的进程必须具备 CAP_SYS_ADMIN 能力。另外，mount 进程（mount_t）具备 capability 类的 sys_admin 能力。

```
        mnt = do_kern_mount(type, flags, name, data);
        if (IS_ERR(mnt))
                return PTR_ERR(mnt);

        err = do_add_mount(mnt, path, mnt_flags);       # 将新的 vfsmount 加入当前进程的 namespace
        if (err)
                mntput(mnt);
        return err;
}
```

由 do_kern_mount 函数执行真正的 mount 操作，创建并设置和该文件系统相关的 vfsmount 数据结构，最后由 do_add_mount 函数将其加入当前进程的 namespace（从而使得当前进程能够“看到”这个新挂载的文件系统，否则仍将看到挂载点目录之前的内容，即在路径名解析时无法“进入”新挂载文件系统的根目录）。

```
struct vfsmount * do_kern_mount(const char *fstype, int flags, const char *name, void *data)
{
        struct file_system_type *type = get_fs_type(fstype);
        struct vfsmount *mnt;
        if (!type)
                return ERR_PTR(-ENODEV);

        mnt = vfs_kern_mount(type, flags, name, data);          # 创建 sb, root_denty, vfsmount
        if (!IS_ERR(mnt) && (type->fs_flags & FS_HAS_SUBTYPE) && !mnt->mnt_sb->s_subtype)
                mnt = fs_set_subtype(mnt, fstype);              # 暂时跳过

        put_filesystem(type);                                   # 递减 file_system_type 的引用计数
        return mnt;
}
EXPORT_SYMBOL_GPL(do_kern_mount);
```

各种文件系统驱动在向内核注册时都提供各自的 file_system_type 数据结构，其中的 mount 函数指针指向该文件系统驱动所实现的挂载方法（注意，文件系统是在一种设备上的特定数据组织格式，因此必须由具体文件系统的驱动向内核注册自己的挂载方法）。调用 get_fs_type 函数根据被挂载文件系统的类型查找它的 file_system_type 数据结构，然后调用 vfs_kern_mount 函数执行挂载操作（创建 sb，root_dentry 和 vfsmount 数据结构）：

```
struct vfsmount * vfs_kern_mount(struct file_system_type *type, int flags, const char *name, void *data)
{
        struct vfsmount *mnt;
        struct dentry *root;
```

```
        if (!type)
                return ERR_PTR(-ENODEV);

        mnt = alloc_vfsmnt(name);
        if (!mnt)
                return ERR_PTR(-ENOMEM);
```

通过 alloc_vfsmnt 函数从内核 mnt_cache 中分配一个 vfsmount 数据结构，复制相应设备的路径字符串并由 vfsmount 的 mnt_devname 域指向。

```
        if (flags & MS_KERNMOUNT)
                mnt->mnt_flags = MNT_INTERNAL;

        root = mount_fs(type, flags, name, data);
        if (IS_ERR(root)) {
                free_vfsmnt(mnt);
                return ERR_CAST(root);
        }
```

mount_fs 函数执行挂载操作，创建相应文件系统的超级块并返回被挂载文件系统的根目录的 dentry 数据结构。

```
        mnt->mnt_root = root;                                   # 被挂载文件系统的根目录的 dentry
        mnt->mnt_sb = root->d_sb;                               # 被挂载文件系统的 SB
        mnt->mnt_mountpoint = mnt->mnt_root;
        mnt->mnt_parent = mnt;
        return mnt;
}
EXPORT_SYMBOL_GPL(vfs_kern_mount);
```

最后设置被挂载文件系统的 vfsmount 数据结构，注意其 mnt_mountpoint 暂时指向被挂载文件系统的根目录的 dentry 结构，最终将指向挂载点目录的 dentry 结构；mnt_parent 暂时指向被挂载文件系统的 vfsmount 结构，最终将指向挂载点目录所在文件系统的 vfsmount 结构。

```
struct dentry * mount_fs(struct file_system_type *type, int flags, const char *name, void *data)
{
        struct dentry *root;
        struct super_block *sb;
        char *secdata = NULL;
        int error = -ENOMEM;

        if (data && !(type->fs_flags & FS_BINARY_MOUNTDATA)) {
                secdata = alloc_secdata();              # return (char *)get_zeroed_page(GFP_KERNEL)
                if (!secdata)
                        goto out;
                error = security_sb_copy_data(data, secdata);
                if (error)
                        goto out_free_secdata;
        }
```

如果 mount data 为二进制的（FS_BINARY_MOUNTDATA 标志有效），则不需要、也无法解析。否则调用 alloc_secdata 函数分配一个额外的内核页面，并调用 security_sb_copy_data 函数解析 mount data 的格式并复制它到新的页面中。注意 mount data 只有如下 4 种合法的格式:

```
#define CONTEXT_STR     "context="
#define FSCONTEXT_STR   "fscontext="
```

```
#define ROOTCONTEXT_STR "rootcontext="
#define DEFCONTEXT_STR  "defcontext="

        root = type->mount(type, flags, name, data);
        if (IS_ERR(root)) {
                error = PTR_ERR(root);
                goto out_free_secdata;
        }
```

通过 file_system_type 数据结构的 mount 函数指针调用具体文件系统驱动提供的 mount 方法，创建被挂载文件系统的超级块的 super_block 数据结构，创建并返回根目录的 dentry 数据结构（回调函数指针即为 VFS 驱动和具体文件系统驱动之间的接口）。

具体文件系统的 mount 方法都通过 VFS 驱动提供的 mount_bdev 函数实现，但是传递各自定义的 xxx_fill_super 函数（比如 ext2_fill_super），以按照具体设备上的特定数据组织格式读取设备上文件系统超级块的内容。

```
        sb = root->d_sb;
        BUG_ON(!sb);
```

局部变量 sb 指向已挂载文件系统的超级块数据结构。

```
        WARN_ON(!sb->s_bdi);
        WARN_ON(sb->s_bdi == &default_backing_dev_info);
        sb->s_flags |= MS_BORN;

        error = security_sb_kern_mount(sb, flags, secdata);
        if (error)
                goto out_sb;
```

文件系统挂载完毕后，通过 security_sb_kern_mount 函数来进一步设置文件系统超级块的安全属性，并检查当前进程是否有足够的能力挂载当前文件系统（确切地说是 sbsec->sid）。参见下文。

```
        /*
         * filesystems should never set s_maxbytes larger than MAX_LFS_FILESIZE
         * but s_maxbytes was an unsigned long long for many releases. Throw
         * this warning for a little while to try and catch filesystems that
         * violate this rule.
         */
        WARN((sb->s_maxbytes < 0), "%s set sb->s_maxbytes to "
                "negative value (%lld)\n", type->name, sb->s_maxbytes);

        up_write(&sb->s_umount);
        free_secdata(secdata);
        return root;
out_sb:
        dput(root);
        deactivate_locked_super(sb);
out_free_secdata:
        free_secdata(secdata);
out:
        return ERR_PTR(error);
}
```

调用具体文件系统驱动的 mount 方法从设备上读取文件系统超级块后，通过 security_sb_kern_mount 函数函数设置超级块的安全属性，并检查当前进程（mount_t）是否具有相应的能力执行指定的操作。在 SELinux 上而言即为 selinux_sb_kern_mount 函数：

```
static int selinux_sb_kern_mount(struct super_block *sb, int flags, void *data)
{
        const struct cred *cred = current_cred();
        struct common_audit_data ad;
        int rc;

        rc = superblock_doinit(sb, data);
        if (rc)
                return rc;
```

首先调用 superblock_doinit 函数解析 mount data 中和文件系统 SC 相关的字符串，并初始化超级块安全属性 sbsec 中的相应域。

```
        /* Allow all mounts performed by the kernel */
        if (flags & MS_KERNMOUNT)
                return 0;

        COMMON_AUDIT_DATA_INIT(&ad, DENTRY);
        ad.u.dentry = sb->s_root;
        return superblock_has_perm(cred, sb, FILESYSTEM__MOUNT, &ad);
}
```

最后调用 superblock_has_perm 函数检查当前进程是否能够挂载指定的文件系统：

```
/* Check whether a task can perform a filesystem operation. */
static int superblock_has_perm(const struct cred *cred,
                               struct super_block *sb,
                               u32 perms,
                               struct common_audit_data *ad)
{
        struct superblock_security_struct *sbsec;
        u32 sid = cred_sid(cred);

        sbsec = sb->s_security;
        return avc_has_perm(sid, sbsec->sid, SECCLASS_FILESYSTEM, perms, ad);
}
```

该函数为 avc_has_perm 函数的封装。注意实际检查的是当前进程针对超级块数据结构自身（sbsec->sid）关于 filesystem 类的 mount 能力。

superblock_doinit 函数解析 mount data 中和文件系统 SC 相关的字符串，并初始化超级块安全属性 sbsec 中的相应域。注意参数 data 指向 mount data，可能为 NULL。

值得一提的是，有下文可见，superblock_doinit 函数还在装载 policy.X 之后被调用，用于初始化当前所有已经挂载的文件系统的 sbsec 数据结构，以及文件系统中相关文件的 inode 的 isec 数据结构（组织在 sbsec->isec_head 链表中）。

```
/*
 * string mount options parsing and call set the sbsec
 */
static int superblock_doinit(struct super_block *sb, void *data)
{
        int rc = 0;
        char *options = data;
        struct security_mnt_opts opts;
```

使用 security_mnt_opts 数据结构来描述 mount data 的解析结果:

```
struct security_mnt_opts {
        char **mnt_opts;
        int *mnt_opts_flags;
        int num_mnt_opts;
};
```

mnt_opts 为指针数组，mnt_opts_flags 为整型数组，它们的长度总是 5（等于 NUM_SEL_MNT_OPTS 宏），而实际长度都由 num_mnt_opts 来指定，取决于 mount data 中实际出现的选项的个数。mnt_opts 数组元素指向“(def|fs|root)?context=”或“seclabel”字符串。由于这些字符串在 mount data 中的出现顺序不一定，因此需要用 mnt_opts_flags 数组中的相应元素来标识对应字符串的类型，比如:

```
#define CONTEXT_MNT             0x01
#define FSCONTEXT_MNT           0x02
#define ROOTCONTEXT_MNT         0x04
#define DEFCONTEXT_MNT          0x08
```

注意，除使用 genfscon 规则描述的文件系统（除 sysfs 之外），或使用“Mount Point Labeling”方式的文件系统，或不支持 SC 的文件系统之外，在 selinux_set_mnt_opts > sb_finish_set_opts 函数中默认地设置 sbsec->flags 中的 SE_SBLABELSUPP 标志。参见下文。

```
        security_init_mnt_opts(&opts);
```

调用 security_init_mnt_opts 函数初始化一个 security_mnt_opts 数据结构（指针数组都为 NULL，长度为 0）。

```
        if (!data)
                goto out;
```

如果用户态没有指定任何 mount data 或者为 binary mount data，则无须解析，直接按照刚初始化的 opts 数据结构设置 sbsec 中和 sid 相关的各个域。

否则，调用 selinux_parse_opts_str 函数解析 mount data，将解析的结果用 opts 结构来描述:

```
        BUG_ON(sb->s_type->fs_flags & FS_BINARY_MOUNTDATA);
        rc = selinux_parse_opts_str(options, &opts);
        if (rc)
                goto out_err;
```

注意在调用 selinux_parse_opts_str 函数前，再次用 BUG_ON 宏检查 mount data 不是 binary 的。该函数解析 mount data 中的各个和 SC 相关的选项，拷贝并由 mnt_opts[i]指针所指向，相应选项的类型由 mnt_opts_flags[i]描述。

最后根据 security_mnt_opts 数据结构来设置 sbsec 的各个域:

```
out:
        rc = selinux_set_mnt_opts(sb, &opts);

out_err:
        security_free_mnt_opts(&opts);
        return rc;
}
```

selinux_set_mnt_opts 函数根据 mount data 解析的结果，设置超级块安全属性中的相应域:

```
static int selinux_set_mnt_opts(struct super_block *sb, struct security_mnt_opts *opts)
{
        const struct cred *cred = current_cred();
        int rc = 0, i;
        struct superblock_security_struct *sbsec = sb->s_security;     # 被挂载文件系统的 sbsec
        const char *name = sb->s_type->name;                           # 被挂载文件系统的名称
        struct inode *inode = sbsec->sb->s_root->d_inode;              # 被卦杂文件系统根目录的 inode
        struct inode_security_struct *root_isec = inode->i_security;   # 被挂载文件系统根目录的 isec
        u32 fscontext_sid = 0, context_sid = 0, rootcontext_sid = 0;
        u32 defcontext_sid = 0;                                        # 用于保存 SC 选项对应的 sid
        char **mount_options = opts->mnt_opts;                 # mount data 中“*context=”字符串的值
        int *flags = opts->mnt_opts_flags;                     # 相应“*context=”字符串的值的类型
        int num_opts = opts->num_mnt_opts;
```

如果没有指定 mount data，则 security_mnt_opts 数据结构中的数组指针都为 NULL，长度为 0。

```
        mutex_lock(&sbsec->lock);
```

设置 sbsec 数据结构之前，获得相应的 mutex。在该函数退出前再释放。

```
        if (!ss_initialized) {
                if (!num_opts) {
                        /* Defer initialization until selinux_complete_init,
                           after the initial policy is loaded and the security
                           server is ready to handle calls. */
                        goto out;
                }
                rc = -EINVAL;
                printk(KERN_WARNING "SELinux: Unable to set superblock options "
                        "before the security server is initialized\n");
                goto out;
        }
```

由于需要将 SC 选项字符串向 sidtab 注册并返回相应的 sid，显然该操作在 policy.X 尚未装载、Security Server 尚未完成初始化时无法进行。所以如果 Security Server 尚未初始化且在 mount data 中指定了 SC 选项，则出错退出。

注意，正如注释所述，如果此时 mount data 中并没有指定任何 SC 选项，则 sbsec 中相应 sid 域的初始化工作延迟到 selinux_complete_init 函数中完成。相关函数调用链如下：

```
security_load_policy > selinux_complete_init > iterate_supers(delayed_superblock_init, NULL)
        > superblock_doinit(sb, NULL);
```

注意：
1，在 security_load_policy 函数的后部，在设置了 ss_initialized 标志后，调用 selinux_complete_init 函数；
2，它调用 iterate_supers 函数，就内核 super_blocks 队列中的所有 SB，调用 delayed_superblock_init 函数；
3，它即为 superblock_doinit 函数的封装函数，注意此时指向 mount data 字符串的参数为 NULL，即没有 mount data；
4，所以，等 selinux_set_mnt_opts 函数被再次调用时，ss_initialized 标志就有效了。
（即，在装载 policy.X 之前就被挂载且不指定 mount data 的文件系统，要经过该函数两次：第一次直接退出；第二次才根据 policy.X 设置 sbsec，以及 sbsec->isec_head 队列中的所有 isec）

```
        /*
```

```
 * Binary mount data FS will come through this function twice.  Once
 * from an explicit call and once from the generic calls from the vfs.
 * Since the generic VFS calls will not contain any security mount data
 * we need to skip the double mount verification.
 *
 * This does open a hole in which we will not notice if the first
 * mount using this sb set explict options and a second mount using
 * this sb does not set any security options.  (The first options
 * will be used for both mounts)
 */
if ((sbsec->flags & SE_SBINITIALIZED) && (sb->s_type->fs_flags & FS_BINARY_MOUNTDATA)
    && (num_opts == 0))
        goto out;
```

（暂时跳过这里关于使用 binary mount data 时两次调用该函数的情况。）

在循环中逐一处理各个 SC 字符串选项，num_opts 为选项的个数：

```
/*
 * parse the mount options, check if they are valid sids.
 * also check if someone is trying to mount the same sb more
 * than once with different security options.
 */
for (i = 0; i < num_opts; i++) {
        u32 sid;

        if (flags[i] == SE_SBLABELSUPP)
                continue;
```

如果 mount data 中即使出现了“seclabel”选项，也不会用 mnt_opts[]/mnt_opts_flags[]数组的相应元素来描述它。但是在向用户态返回 sbsec 时（比如读取/proc/mounts 文件时），会使用 mnt_opts[]/mnt_opts_flags[]数组元素来描述它。对于 mount 系统调用无须考虑 mnt_opts_flags[]中的 SE_SBLABELSUPP 元素。

```
        rc = security_context_to_sid(mount_options[i], strlen(mount_options[i]), &sid);
        if (rc) {
                printk(KERN_WARNING "SELinux: security_context_to_sid"
                       "(%s) failed for (dev %s, type %s) errno=%d\n",
                       mount_options[i], sb->s_id, name, rc);
                goto out;
        }
```

就当前 SC 选项的值，调用 security_context_to_sid 函数相 sidtab 注册，并返回对应的 sid。然后即可以根据当前 SC 选项的类型，设置 sbsec->flags 中的标志位并将 sid 保存到相应的局部变量中：

```
        switch (flags[i]) {
        case FSCONTEXT_MNT:
                fscontext_sid = sid;

                if (bad_option(sbsec, FSCONTEXT_MNT, sbsec->sid, fscontext_sid))
                        goto out_double_mount;

                sbsec->flags |= FSCONTEXT_MNT;
                break;
```

通过 bad_option 函数检查新指定的“fscontext=”选项是否和 sbsec->sid 冲突。如果没有，则将相应 sid 保存在 fscontext_sid 变量中并设置 sbsec->flags 中的 FSCONTEXT_MNT 标志。

```
                case CONTEXT_MNT:
                        context_sid = sid;

                        if (bad_option(sbsec, CONTEXT_MNT, sbsec->mntpoint_sid, context_sid))
                                goto out_double_mount;

                        sbsec->flags |= CONTEXT_MNT;
                        break;
```

通过 bad_option 函数检查新指定的“context=”选项是否和 sbsec->mntpoint_sid 冲突。如果没有，则将相应 sid 保存在 context_sid 变量中并设置 sbsec->flags 中的 CONTEXT_MNT 标志。

```
                case ROOTCONTEXT_MNT:
                        rootcontext_sid = sid;

                        if (bad_option(sbsec, ROOTCONTEXT_MNT, root_isec->sid, rootcontext_sid))
                                goto out_double_mount;

                        sbsec->flags |= ROOTCONTEXT_MNT;
                        break;
```

通过 bad_option 函数检查新指定的“rootcontext=”选项是否和被挂载文件系统根目录的 sid（root_isec->sid）冲突。如果没有，则将相应 sid 保存在 rootcontext_sid 变量中并设置 sbsec->flags 中的 ROOTCONTEXT_MNT 标志。

```
                case DEFCONTEXT_MNT:
                        defcontext_sid = sid;

                        if (bad_option(sbsec, DEFCONTEXT_MNT, sbsec->def_sid, defcontext_sid))
                                goto out_double_mount;

                        sbsec->flags |= DEFCONTEXT_MNT;
                        break;
```

通过 bad_option 函数检查新指定的“defcontext=”选项是否和 sbsec->def_sid 冲突。如果没有，则将相应 sid 保存在 defcontext_sid 变量中并设置 sbsec->flags 中的 DEFCONTEXT_MNT 标志。

```
                default:
                        rc = -EINVAL;
                        goto out;
                }
        }

        if (sbsec->flags & SE_SBINITIALIZED) {
                /* previously mounted with options, but not on this attempt? */
                if ((sbsec->flags & SE_MNTMASK) && !num_opts)
                        goto out_double_mount;
                rc = 0;
                goto out;
        }
```

如果当前被挂载文件系统之前已经被挂载过一次且指定了 mount data，但是当前这次没有指定，则保留之前挂载时所指定的 mount data。

```
        if (strcmp(sb->s_type->name, "proc") == 0)
                sbsec->flags |= SE_SBPROC;
```

如果是 proc 文件系统，则设置 sbsec->flags 中的 SE_SBPROC 标志（TODO：该标志的作用？）。

```
/* Determine the labeling behavior to use for this filesystem type. */
rc = security_fs_use((sbsec->flags & SE_SBPROC) ? "proc" : sb->s_type->name,
                     &sbsec->behavior, &sbsec->sid);
if (rc) {
        printk(KERN_WARNING "%s: security_fs_use(%s) returned %d\n",
               __func__, sb->s_type->name, rc);
        goto out;
}
```

调用 security_fs_use 函数返回 refpolicy 中给文件系统定义的 SC。如果存在相应的定义，则要么使用 fs_use_xattr/fs_use_task/fs_use_trans 规则，要么使用 genfscon 规则。

对于所有使用 fs_use_xxx 规则定义 SC 的文件系统，相应规则的描述符及其指定的 SC 的解析结果被组织在 policydb.ocontexts[OCON_FSUSE]队列中。security_fs_use 函数比较当前被挂载文件系统的名称和各个 ocontext 数据结构的 u.name 域。如果有匹配项，则返回其 v.behavior 的值到 sbsec->behavior 中（描述使用了哪一个规则 SECURITY_FS_USE_XATTR/TASK/TRANS），并将其 context[0]注册到 sidtab 中，由 sbsec->sid 返回相应的 sid。

如果没有匹配项（即没有使用 fs_use_xxx 规则），则可能使用了 genfscon 规则（比如 sysfs，proc，selinuxfs 等各种虚拟文件系统）。因此 security_fs_use 函数进一步调用 security_genfs_sid 函数，在 policydb.genfs 队列中查找和当前 fstype 相匹配的 genfs 数据结构。如果存在匹配项，且其 ocontext 数据结构的 u.name 等于“/”且 v.sclass 等于 dir class 的 policy value，则将其 context[0]注册到 sidtab 中并由 sbsec->sid 返回。此时 sbsec->behavior 返回 SECURITY_FS_USE_GENFS。

如果没有匹配项，则 sbsec->sid 被设置为 SECINITSID_UNLABELED，sbsec->behavior 被设置为 SECURITY_FS_USE_NONE。

总之，sbsec->sid 的初始值要么由相应的 fs_use_xxx/genfscon 规则指定，要么为 SECINITSID_UNLABELED。由下文可见，设置 sbsec 中任何和 sid 相关的域，当前进程都必须具有对 sbsec->sid 的 filesystem 类的 relabelfrom 能力。

```
/* sets the context of the superblock for the fs being mounted. */
if (fscontext_sid) {
        rc = may_context_mount_sb_relabel(fscontext_sid, sbsec, cred);
        if (rc)
                goto out;

        sbsec->sid = fscontext_sid;
}
```

如果 mount data 中指定了有效的“fscontext=”选项，则将指定的 SC 对应的 sid 作为文件系统自身（即超级块）的 sid。

由于要更新 sbsec->sid，所以调用 may_context_mount_sb_relabel 函数检查当前进程是否对原来的 sbsec->sid 具有 filesystem 类的 relabelfrom 能力，以及对新的 fscontext_sid 值具有 filesystem 类的 relabelto 能力。

如果具备相关能力，则根据 fscontext_sid 来设置 sbsec->sid。

```
/*
```

```
         * Switch to using mount point labeling behavior.
         * sets the label used on all file below the mountpoint, and will set
         * the superblock context if not already set.
         */
        if (context_sid) {
                if (!fscontext_sid) {
                        rc = may_context_mount_sb_relabel(context_sid, sbsec, cred);
                        if (rc)
                                goto out;
                        sbsec->sid = context_sid;
```

如果指定了“context=”选项，则采用“Mount Point Labeling”方式确定整个文件系统的标签，适用于不支持安全属性的文件系统或者连接不可信的设备的情况，此时以 sbsec->mntpoint_sid 作为整个文件系统上所有数据对象的安全属性。所以如果没有同时指定“fscontext=”选项，则将超级块自身的 sid 也设置为 context_sid。为此需要通过 may_context_mount_sb_relabel 函数检查当前进程对原有 sbsec->sid 具有 filesystem 类的 relabelfrom 能力，对 context_sid 具有 filesystem 类的 relabelto 能力，然后才能更新 sbsec->sid 为 context_sid。

```
                } else {
                        rc = may_context_mount_inode_relabel(context_sid, sbsec, cred);
                        if (rc)
                                goto out;
                }
```

如果同时指定了“context=”和“fscontext=”选项（注意，根据 mount 的 man 手册页，fscontext/defcontext 和 context 是互斥的），则分别设置 sbsec 中的 mntpoint_sid 和 sid 域（注意之前已经将 sbsec->sid 设置为 fscontext_sid），则调用 may_context_mount_inode_relabel 函数检查当前进程对 sbsec->sid（即 fscontext_sid）具有 filesystem 类的 relabelfrom 能力，且 context_sid 对它具有 filesystem 类的 associate 能力。

```
                if (!rootcontext_sid)
                        rootcontext_sid = context_sid;
```

在采用“Mount Point Labeling”时，如果没有同时指定“rootcontext=”选项，则用 context_sid 设置它。

```
                sbsec->mntpoint_sid = context_sid;
                sbsec->behavior = SECURITY_FS_USE_MNTPOINT;
        }
```

显然，根据 context_sid 来设置 sbsec->mntpoint_sid，并设置 sbsec->behavior 为 SECURITY_FS_USE_MNTPOINT。

```
        if (rootcontext_sid) {
                rc = may_context_mount_inode_relabel(rootcontext_sid, sbsec, cred);
                if (rc)
                        goto out;

                root_isec->sid = rootcontext_sid;
                root_isec->initialized = 1;
        }
```

至此，rootcontext_sid 变量一定不为空：要么由“rootcontext=”选项指定，要么由“context=”选项决定。将其作为文件系统根目录的安全属性，并设置其 isec->initialized 标志。事先需要检查 rootcontext_sid 对 sbsec->sid 是否具有 filesystem 类的 associate 能力。

```
        if (defcontext_sid) {
                if (sbsec->behavior != SECURITY_FS_USE_XATTR) {
                        rc = -EINVAL;
                        printk(KERN_WARNING "SELinux: defcontext option is "
                               "invalid for this filesystem type\n");
                        goto out;
                }
```

只有支持扩展属性的文件系统（使用 fs_use_xattr 规则）才能支持“defcontext=”选项（作为未打标签文件默认的 SC）。

```
                if (defcontext_sid != sbsec->def_sid) {
                        rc = may_context_mount_inode_relabel(defcontext_sid, sbsec, cred);
                        if (rc)
                                goto out;
                }

                sbsec->def_sid = defcontext_sid;
        }
```

如果需要更新 sbsec->def_sid，则需要检查 defcontext_sid 对 sbsec->sid 是否具有 filesystem 类的 associate 能力。

至此，sbsec 中各个 sid 域确定完毕。还需要继续调用 sb_finish_set_opts 函数完成剩余“善后”工作，比如设置 sbsec->flags 中的 SE_SBINITIALIZED 标志、初始化所有已经创建的 inode 的安全属性等操作。

```
        rc = sb_finish_set_opts(sb);
out:
        mutex_unlock(&sbsec->lock);
        return rc;
out_double_mount:
        rc = -EINVAL;
        printk(KERN_WARNING "SELinux: mount invalid.  Same superblock, different "
               "security settings for (dev %s, type %s)\n", sb->s_id, name);
        goto out;
}

static int sb_finish_set_opts(struct super_block *sb)
{
        struct superblock_security_struct *sbsec = sb->s_security;
        struct dentry *root = sb->s_root;
        struct inode *root_inode = root->d_inode;
        int rc = 0;

        if (sbsec->behavior == SECURITY_FS_USE_XATTR) {
                /* Make sure that the xattr handler exists and that no
                   error other than -ENODATA is returned by getxattr on
                   the root directory.  -ENODATA is ok, as this may be
                   the first boot of the SELinux kernel before we have
                   assigned xattr values to the filesystem. */
                if (!root_inode->i_op->getxattr) {
                        printk(KERN_WARNING "SELinux: (dev %s, type %s) has no "
                               "xattr support\n", sb->s_id, sb->s_type->name);
                        rc = -EOPNOTSUPP;
                        goto out;
                }

                rc = root_inode->i_op->getxattr(root, XATTR_NAME_SELINUX, NULL, 0);
```

```
                if (rc < 0 && rc != -ENODATA) {
                        if (rc == -EOPNOTSUPP)
                                printk(KERN_WARNING "SELinux: (dev %s, type "
                                       "%s) has no security xattr handler\n", sb->s_id, sb->s_type->name);
                        else
                                printk(KERN_WARNING "SELinux: (dev %s, type "
                                       "%s) getxattr errno %d\n", sb->s_id, sb->s_type->name, -rc);
                        goto out;
                }
        }
```

正如注释所述，如果文件系统的 SC 由 fs_use_xttr 规则所描述，则相应文件系统驱动必须提供 getxattr 和 setxattr 方法，以从具体设备上读取文件的扩展属性。这里检查相应文件系统驱动提供的 inode_operation 方法表中是否实现了 getxattr 方法。如果是，则尝试读出被挂载文件系统根目录的“security.selinux”扩展属性。如果返回非 ENODATA 之外的其他错误码，则表示无法正常读取扩展属性（被挂载文件系统上可能尚未部署标签，因此返回 ENODATA 不能当作错误）。

```
        sbsec->flags |= (SE_SBINITIALIZED | SE_SBLABELSUPP);
```

由于在调用者中已经完成了 sbsec 中各个 sid 域的设置，因此设置 sbsec->flags 中的 SE_SBINITIALIZED 标志。注意，同时默认地设置 SE_SBLABELSUPP 标志。

```
        if (sbsec->behavior > ARRAY_SIZE(labeling_behaviors))
                printk(KERN_ERR "SELinux: initialized (dev %s, type %s), unknown behavior\n",
                       sb->s_id, sb->s_type->name);
        else
                printk(KERN_DEBUG "SELinux: initialized (dev %s, type %s), %s\n",
                       sb->s_id, sb->s_type->name, labeling_behaviors[sbsec->behavior-1]);
```

如上文所述，sbsec->behavior 描述相应文件系统 SC 的定义方式，由 refpolicy 中使用的规则决定。根据它索引 labeling_behaviors 字符串指针数组，并打印该文件系统已经被挂载的信息，比如：

```
[root/sysadm_r/s0@~]# dmesg | grep "SELinux: initialized" | grep -e ext2 -e selinuxfs
[    4.421857] SELinux: initialized (dev selinuxfs, type selinuxfs), uses genfs_contexts
[    4.431115] SELinux: initialized (dev sda, type ext2), uses xattr
[root/sysadm_r/s0@~]#
```

```
        if (sbsec->behavior == SECURITY_FS_USE_GENFS ||
            sbsec->behavior == SECURITY_FS_USE_MNTPOINT ||
            sbsec->behavior == SECURITY_FS_USE_NONE ||
            sbsec->behavior > ARRAY_SIZE(labeling_behaviors))
                sbsec->flags &= ~SE_SBLABELSUPP;

        /* Special handling for sysfs. Is genfs but also has setxattr handler*/
        if (strncmp(sb->s_type->name, "sysfs", sizeof("sysfs")) == 0)
                sbsec->flags |= SE_SBLABELSUPP;
```

如果文件系统驱动实现了访问文件扩展属性的方法，则能够提供文件的 SC，则设置 sbsec->flags 中的 SE_SBLABELSUPP 标志。否则清除该标志。注意，使用 genfscon 规则以及“Mount Point Labeling”都可以一劳永逸地指定整个文件系统的标签，因此相应文件系统驱动不需要实现访问文件扩展属性的方法，所以清除该标志。

由于 sysfs 文件系统驱动实现了 setxattr 方法，所以对其保留了 SE_SBLABELSUPP 标志。

```
        /* Initialize the root inode. */
        rc = inode_doinit_with_dentry(root_inode, root);
```

inode_doinit_with_dentry 函数根据所在文件系统 SC 的确定方式（根据 sbsec->behavior 域）经由 sbsec 中的相应 sid 域，确定新创建文件的 sid（isec->sid）。在挂载文件系统时调用该函数以确定其根目录的 sid，比如，如果指定“Mount Point Labeling”（SECURITY_FS_USE_MNTPOINT 标志有效），则直接设置为 sbsec->mntpoint_sid；如果使用 fs_use_task 规则（SECURITY_FS_USE_TASK 标志有效），则直接设置为文件创建者进程的 sid（isec->task_sid）。

```
        /* Initialize any other inodes associated with the superblock, e.g.
           inodes created prior to initial policy load or inodes created
           during get_sb by a pseudo filesystem that directly populates itself. */
        spin_lock(&sbsec->isec_lock);

next_inode:
        if (!list_empty(&sbsec->isec_head)) {
                struct inode_security_struct *isec =
                                list_entry(sbsec->isec_head.next, struct inode_security_struct, list);
                struct inode *inode = isec->inode;
                spin_unlock(&sbsec->isec_lock);
                inode = igrab(inode);
                if (inode) {
                        if (!IS_PRIVATE(inode))
                                inode_doinit(inode);
                        iput(inode);
                }
                spin_lock(&sbsec->isec_lock);
                list_del_init(&isec->list);
                goto next_inode;
        }
        spin_unlock(&sbsec->isec_lock);
out:
        return rc;
}
```

在访问文件之前需要正确设置其 inode 的安全属性 inode_security_struct 数据结构，该操作还是在 inode_doinit_with_dentry 函数中完成。如果此时 Security Server 尚未完成初始化（ss_initialized 等于 0），则将相应 isec 数据结构加入其所在文件系统超级块的 sbsec->isec_head 队列。

由上文所述，在装载、解析 policy.X 之后需要为之前已经挂载的文件系统，设置其超级块的安全属性。相关函数调用链如下：

```
security_load_policy > selinux_complete_init > iterate_supers(delayed_superblock_init, NULL)
        > superblock_doinit(sb, NULL) > selinux_set_mnt_opts > sb_finish_set_opts
                > inode_doinit(inode, NULL) > inode_doinit_with_dentry(inode, NULL)
```

inode_doinit 函数直接调用 inode_doinit_with_dentry 函数完成该工作。注意此时 ss_initialized 等于 1，根据 sbsec->behavior 以相应的方式确定 isec->sid（注意，这仅仅是 isec->sid 的初始值，在 security_compute_sid 函数中还可能被 tsec->fscreate_sid 或者匹配的 type_transition 规则所重载）。

10.6.2 确定新创建文件的标签

1，mknodat 系统调用将调用 vfs_create 函数：

```
SYSCALL_DEFINE4(mknodat, int, dfd, const char __user *, filename, int, mode, unsigned, dev) > vfs_create

int vfs_create(struct inode *dir, struct dentry *dentry, int mode, struct nameidata *nd)
{
```

```
        int error = may_create(dir, dentry);

        if (error)
                return error;

        if (!dir->i_op->create)
                return -EACCES; /* shouldn't it be ENOSYS? */
        mode &= S_IALLUGO;
        mode |= S_IFREG;
        error = security_inode_create(dir, dentry, mode);              # 所有相关 MAC 检查
        if (error)
                return error;

        error = dir->i_op->create(dir, dentry, mode, nd);              # 创建 inode，设置新文件的标签
        if (!error)
                fsnotify_create(dir, dentry);
        return error;
}
```

首先在 security_inode_create 函数中进行 LSM 安全性检查，就 SELinux 而言调用 selinux/hooks.c 中的 may_create 函数，然后再调用具体文件系统提供的 inode_operations 方法表中的 create 函数创建 inode，并确定、设置新文件的标签。

```
int security_inode_create(struct inode *dir, struct dentry *dentry, int mode)
{
        if (unlikely(IS_PRIVATE(dir)))                                 # ((inode)->i_flags & S_PRIVATE)
                return 0;
        return security_ops->inode_create(dir, dentry, mode);
}

static int selinux_inode_create(struct inode *dir, struct dentry *dentry, int mask)
{
        return may_create(dir, dentry, SECCLASS_FILE);
}
```

may_create 函数检查当前进程是否能够在指定的目录下创建新文件，并检查当前进程能否创建指定 sid 的文件，以及新文件的 sid 能否和其所在文件系统相关联（associate）。

注意该函数并不设置新文件的标签，而仅检查所有相关的操作是否被许可。

```
/* Check whether a task can create a file. */
static int may_create(struct inode *dir, struct dentry *dentry, u16 tclass)
{
        const struct cred *cred = current_cred();
        const struct task_security_struct *tsec = cred->security;
        struct inode_security_struct *dsec;
        struct superblock_security_struct *sbsec;
        u32 sid, newsid;
        struct common_audit_data ad;
        int rc;

        dsec = dir->i_security;                         # 新文件所在父目录的 inode_security_struct
        sbsec = dir->i_sb->s_security;                  # 新文件所在文件系统的 superblock_security_struct

        sid = tsec->sid;                                # 当前进程（创建者进程）的 sid
        newsid = tsec->create_sid;                      # 读取/proc/pid/attr/fscreate 的设置结果
```

```
        COMMON_AUDIT_DATA_INIT(&ad, FS);
        ad.u.fs.path.dentry = dentry;

        rc = avc_has_perm(sid, dsec->sid, SECCLASS_DIR, DIR__ADD_NAME | DIR__SEARCH, &ad);
        if (rc)
                return rc;
```

首先检查当前进程对父目录是否具有{ add_name search }能力。

```
        if (!newsid || !(sbsec->flags & SE_SBLABELSUPP)) {
                rc = security_transition_sid(sid, dsec->sid, tclass, &newsid);
                if (rc)
                        return rc;
        }
```

如果当前进程没有通过其/proc/pid/attr/fscreate 文件显式地指定新文件的标签，或者其所在文件系统 sbsec->flags 中的 SE_SBLABELSUPP 标志也没有被设置，则通过 security_transition_sid > security_compute_sid 计算新创建文件的 sid，由 newsid 参数返回（role == object_r，type 默认继承父目录的标签，但可被 type_transition 规则重载）。

```
        rc = avc_has_perm(sid, newsid, tclass, FILE__CREATE, &ad);
        if (rc)
                return rc;
```

然后检查当前进程是否对 SECCLASS_FILE 类的 sid 具有 create 能力。

```
        return avc_has_perm(newsid, sbsec->sid, SECCLASS_FILESYSTEM, FILESYSTEM__ASSOCIATE, &ad);
}
```

最后检查新文件的 sid 能否和相应文件系统 superblock 的 sid 相关联。

2，在 vfs_create 函数中，实际创建新文件 inode 的操作由具体文件系统的相应 create 方法实现，它将确定新文件的安全上下文并使其生效（设置 isec->sid 并写入磁盘索引节点的 xattr）。

在 linux/fs/目录下为各种不同文件系统的驱动，它们都提供 namei.c 文件，在其中定义该文件系统的索引节点操作方法表 inode_operations。比如：

```
ext2/namei.c:const struct inode_operations ext2_dir_inode_operations = {
        .create         = ext2_create,
        .lookup         = ext2_lookup,
        .link           = ext2_link,
        .unlink         = ext2_unlink,
        .symlink        = ext2_symlink,
        .mkdir          = ext2_mkdir,
        .rmdir          = ext2_rmdir,
        .mknod          = ext2_mknod,
        .rename         = ext2_rename,
#ifdef CONFIG_EXT2_FS_XATTR
        .setxattr       = generic_setxattr,
        .getxattr       = generic_getxattr,
        .listxattr      = ext2_listxattr,
        .removexattr    = generic_removexattr,
#endif
        .setattr        = ext2_setattr,
        .get_acl        = ext2_get_acl,
};
```

其中 ext2_dir_inode_operations 方法表的 create 函数即为 ext2_create，它又调用 ext2_new_inode > ext2_init_security 函数设置新文件的标签：

```
vfs_create > ext2_dir_inode_operations.create = ext2_create > ext2_new_inode > ext2_init_security

int ext2_init_security(struct inode *inode, struct inode *dir)
{
        int err;
        size_t len;
        void *value;                                            # SELinux security context
        char *name;                                             # “selinux”

        err = security_inode_init_security(inode, dir, &name, &value, &len);  # 返回新文件的 SC 字符串
        if (err) {
                if (err == -EOPNOTSUPP)
                        return 0;
                return err;
        }
        err = ext2_xattr_set(inode, EXT2_XATTR_INDEX_SECURITY, name, value, len, 0); # 写入磁盘索引节点
        kfree(name);
        kfree(value);
        return err;
}
```

security_inode_init_security 函数确定新创建文件的安全属性（初始化 inode_security_struct 数据结构），返回新文件的安全上下文的 key 字符串（“selinux”），value 字符串及其长度（即 SC 字符串），以便为接下来写入新文件 inode 的 xattr 做准备。注意 name 和 value 字符串都是在 security_inode_init_security 内部分配，由其调用者 ext2_init_security 函数在使用后负责释放。

```
int security_inode_init_security(struct inode *inode, struct inode *dir,
                                 char **name, void **value, size_t *len)
{
        if (unlikely(IS_PRIVATE(inode)))
                return -EOPNOTSUPP;
        return security_ops->inode_init_security(inode, dir, name, value, len);
}

static int selinux_inode_init_security(struct inode *inode, struct inode *dir,
                                       char **name, void **value, size_t *len)
{
        const struct cred *cred = current_cred();
        const struct task_security_struct *tsec = cred->security;
        struct inode_security_struct *dsec;
        struct superblock_security_struct *sbsec;
        u32 sid, newsid, clen;
        int rc;
        char *namep = NULL, *context;

        dsec = dir->i_security;                         # 父目录的 inode_security_struct
        sbsec = dir->i_sb->s_security;                  # 相应文件系统的 superblock_security_struct

        sid = tsec->sid;                                # 当前进程的 sid
        newsid = tsec->create_sid;                      # 应用程序源代码中通过 setfscreatecon 函数设置的结果

        if ((sbsec->flags & SE_SBINITIALIZED) && (sbsec->behavior == SECURITY_FS_USE_MNTPOINT))
                newsid = sbsec->mntpoint_sid;
```

首先，如果文件系统 sbsec 已经完成初始化（SE_SBINITIALIZED 标志有效）且使用了“Mount-Point Labeling”（SECURITY_FS_USE_MNTPOINT 标志位被设置），则新文件（确切地说是整个文件系统上的所有文件）的 sid 都等于 sbsec->mntpoint_sid（即“context=”选项所指 SC 字符串对应的 sid）。

由此可见，“Mount Point Labeling”可以重载 attr/fscreate 文件的设置，以及 policy.X 中的 type_transition 规则，以及 Security Server 给新文件设置的默认 sid。

```
else if (!newsid || !(sbsec->flags & SE_SBLABELSUPP)) {
        rc = security_transition_sid(sid, dsec->sid,
                                     inode_mode_to_security_class(inode->i_mode), &newsid);
        if (rc) {
                printk(KERN_WARNING "%s:  "
                       "security_transition_sid failed, rc=%d (dev=%s " "ino=%ld)\n",
                       __func__, -rc, inode->i_sb->s_id, inode->i_ino);
                return rc;
        }
}
```

如果没有调用 setfscreatecon 函数显式地设置新文件的 SC，或者相应文件系统不支持或不需要提供 SC（SE_SBLABELSUPP 标志无效），则查询 policy.X 得到当前进程在相应目录下创建新文件的 sid（要么继承父目录的标签，要么由相应 type_transition 规则显式地指定，security_tranition_sid 函数总是能够返回新文件的 newsid）。

注意，在该 else-if 分支中并没有判断 sbsec->flags 中的 SE_SBINITIALIZED 标志，而在 security_transition_sid > security_compute_sid 函数中，如果该标志无效，则直接返回 newsid 为 dsec->sid。

至此，新文件的 newsid 一定不为 0。

```
/* Possibly defer initialization to selinux_complete_init. */
if (sbsec->flags & SE_SBINITIALIZED) {
        struct inode_security_struct *isec = inode->i_security;
        isec->sclass = inode_mode_to_security_class(inode->i_mode);
        isec->sid = newsid;
        isec->initialized = 1;
}
```

然后将新文件的 newsid 保存在 isec->sid 中并设置 isec->sclass 为新文件所属 class 的 kernel value，并设置 isec->initialized 标志。

注意，正如注释所述，如果此时 sbsec->flags 中 SE_SBINITIALIZED 标志没有被设置，即 policy.X 尚未被装载、Security Server 尚未初始化，显然无法设置 isec。会在相关函数调用链 selinux_d_instantiate > inode_doinit_with_dentry 函数中将当前 isec 加入 sbsec->isec_head 队列，而等到 policy.X 被装载完毕后，再初始化它们。

```
if (!ss_initialized || !(sbsec->flags & SE_SBLABELSUPP))
        return -EOPNOTSUPP;
```

如果相应文件系统不支持存储 SC，则直接返回错误码-EOPNOTSUPP。

```
if (name) {
        namep = kstrdup(XATTR_SELINUX_SUFFIX, GFP_NOFS);       # “selinux”
        if (!namep)
                return -ENOMEM;
        *name = namep;
```

```
        }
```

将 name 所指指针指向“selinux”字符串。最后调用 security_sid_to_context_force 函数向 sidtab 哈希表查询 newsid 所对应的 context 数据结构，并通过 policydb 中的“字符串-数值”指针数组拼接安全上下文字符串（参见下文）：

```
        if (value && len) {                                                    # 若输出参数有效
                rc = security_sid_to_context_force(newsid, &context, &clen);
                if (rc) {
                        kfree(namep);
                        return rc;
                }
                *value = context;
                *len = clen;
        }

        return 0;
}
```

返回 ext2_init_security 函数，通过 security_inode_init_security 函数初始化了文件的 isec，并通过参数 name/value/len 返回新文件的 SC 字符串及其长度后，就可以调用 ext2_xattr_set 函数将 SC 字符串写入磁盘索引节点的“security.selinux”扩展属性了。

10.7 Context 数据结构和 u32 sid 之间的映射

用户态所使用的安全上下文都以“user:role:type[:mls_range]”字符串表示，使用“ls -Z”或者“ps -Z”命令观察到的文件或进程的安全上下文都是一个字符串，另外写入当前进程/proc/<pid>/attr/*文件的也是字符串（参见上文 selinux_setprocattr 函数）。

在将 policy.X 经由/selinuxfs/load 文件装入内核时，SELinux 内核驱动 policy_read 函数就 policy.X 中的各类语法成分的组织顺序，调用相应的 xxx_read 函数（比如 type_read），从 policy.X 中解析单个语法成分（即标识符，identifier）的名称字符串及标识符的属性，用相应的 xxx_datum 数据结构（比如 type_datum）描述标识符的属性，并通过 hashtab_node 建立名称字符串及其属性的关联，然后注册到 policydb.p_xxx.tables 符号表中。

出于效率的考虑，显然不能把 SC 字符串直接保存在各个内核数据结构的安全属性中，而且尚未解析的 SC 字符串也无法直接使用，况且各种标识符之间可能组成各种不同的 SC。于是在 SELinux 内核空间用 context 数据结构来描述 SC，其中各个域都以 u32 来表示，该数值等于相应 SELinux 语法成分 xxx_datum.value，即为 checkpolicy 给该标识符分配的数值（称为“policy value”）。

SELinux 内核空间给每个有效的 context 分配一个 u32 sid，所有内核数据结构的安全扩展中都通过 sid 来描述当前对象的安全上下文。sidtab_node 用于建立 context 数据结构和其 sid 之间的关联，组织为 sidtab 哈希表。这样当同一个 context 数据结构被不同的安全属性所使用时，它们只需保存同样的 sid 数值即可，而无需保存整个 context 数据结构。

10.7.1 sidtab_node 的定义和 sidtab 的组织结构

sidtab 哈希表及其元素定义在 selinux/ss/sidtab.h 中：

```
struct sidtab_node {
```

```
        u32 sid;                                /* security identifier */
        struct context context;                 /* security context structure */
        struct sidtab_node *next;
};
```

sidtab_node 数据结构将一个 sid 和一个 context 相关联。next 用于组织哈希表冲突项的单向链表。

```
struct sidtab {
        struct sidtab_node **htable;
        unsigned int nel;                       /* number of elements */
        unsigned int next_sid;                  /* next SID to allocate */
        unsigned char shutdown;
        spinlock_t lock;                        /* 保护整个数据结构的自旋锁 */
};
```

htable 指向一个有 128 个指针的指针数组。根据一个 sidtab_node->sid 的末 7 位（注意，2^7=128）将其加入相应的冲突队列，在队列中各元素以 sid 从小到大的方式排列。

nel 为当前哈希表中元素的个数；next_sid 为下一个可用的 sid 值（分配后递增 1）；shutdown 为“关闭”标志，在撤销当前 sidtab 时设置。

10.7.2 sidtab_insert 函数 - sidtab_node 的插入

sidtab_insert 函数创建一个新的 sitab_node 元素并加入 sidtab 哈希表，建立一个已分配的 sid 和一个 context 数据结构之间的关联。

```
int sidtab_insert(struct sidtab *s, u32 sid, struct context *context)
{
        int hvalue, rc = 0;
        struct sidtab_node *prev, *cur, *newnode;

        if (!s) {
                rc = -ENOMEM;
                goto out;
        }

        hvalue = SIDTAB_HASH(sid);
```

首先根据 sid 数值计算散列值，即简单地保留 sid 的末 7 位：

```
#define SIDTAB_HASH_BITS              7
#define SIDTAB_HASH_BUCKETS           (1 << SIDTAB_HASH_BITS)
#define SIDTAB_HASH_MASK              (SIDTAB_HASH_BUCKETS-1)
#define SIDTAB_HASH(sid)              (sid & SIDTAB_HASH_MASK)

        prev = NULL;
        cur = s->htable[hvalue];                        # 为什么不是 (s->htable)[hvalue] ?
        while (cur && sid > cur->sid) {
                prev = cur;
                cur = cur->next;
        }
```

然后从 htable[hvalue]所指向的队列首部开始遍历：如果待加入的 sidtab_node->sid 大于当前元素的 sid，则后移 cur 指针，直至队列末尾，或者 cur 所指元素的 sid 不小于待加入的节点的 sid 为止（注意此时 prev 指向插入点元素）。

```
        if (cur && sid == cur->sid) {
                rc = -EEXIST;
                goto out;
        }
```

如果当前 sid 已经被使用过，则出错返回（sid 的分配是单调递增的，而且只能到达 UINT_MAX）。

```
        newnode = kmalloc(sizeof(*newnode), GFP_ATOMIC);
        if (newnode == NULL) {
                rc = -ENOMEM;
                goto out;
        }
        newnode->sid = sid;
        if (context_cpy(&newnode->context, context)) {
                kfree(newnode);
                rc = -ENOMEM;
                goto out;
        }
```

否则分配一个 sidtab_node 数据结构保存当前 sid 并复制参数所指 context 数据结构。

```
       if (prev) {
                newnode->next = prev->next;
                wmb();
                prev->next = newnode;
        } else {
                newnode->next = s->htable[hvalue];
                wmb();
                s->htable[hvalue] = newnode;
        }
```

然后把 newnode 插入 prev 所指节点之后。如果 prev 为 NULL，则说明当前冲突队列原先为空。注意这里调用 wmb 函数构筑了写内存屏障以确保按照正确的顺序执行链表插入语句（假设前后语句的执行顺序颠倒，则将在链表中形成“回路”，造成“链表腐败”）。

```
        s->nel++;
        if (sid >= s->next_sid)
                s->next_sid = sid + 1;
out:
        return rc;
}
```

最后递增元素计数器。如果新插入的节点的 sid 大于或等于 sidtab->next_sid，则更新 next_sid 为其后继数值，从而避免下一次的分配冲突。

10.7.3 sidtab_context_to_sid 函数 - 返回或分配 sid

sidtab_context_to_sid 函数查找 sidtab 返回指定 context 数据结构的 sid。如果该 context 数据结构尚未注册到 sidtab 中，则调用上述 sidtab_insert 函数注册。最后通过 out_sid 返回新分配的 sid。

```
int sidtab_context_to_sid(struct sidtab *s, struct context *context, u32 *out_sid)
{
        u32 sid;
        int ret = 0;
        unsigned long flags;

        *out_sid = SECSID_NULL;                                 # 0x00000000, unspecified sid
```

```
        sid = sidtab_search_context(s, context);
```

首先调用 sidtab_search_context 函数，逐一遍历 sidtab 中所有 127 个冲突队列，在队列中逐一寻找匹配的 sidtab_node 节点，则返回其 sid；否则返回 0。

```
        if (!sid) {
                spin_lock_irqsave(&s->lock, flags);
                /* Rescan now that we hold the lock. */
                sid = sidtab_search_context(s, context);
                if (sid)
                        goto unlock_out;
```

如果返回 0，则说明该 context 数据结构尚未“记录”到 sidtab 中，此时分配一个新的 sidtab_node 数据结构，复制 context 的内容并插入 sidtab。

在修改 sidtab 之前首先要获得相应的 spinlock。由于在等待获得 spinlock 期间其它执行流可能已经完成了该 context 的注册工作，因此在结束等待后要再次检查。（体会此处内核代码的精妙！）

```
                /* No SID exists for the context.  Allocate a new one. */
                if (s->next_sid == UINT_MAX || s->shutdown) {
                        ret = -ENOMEM;
                        goto unlock_out;
                }
                sid = s->next_sid++;
```

如果所有可用的 next_sid 已经用完，或者当前 sidtab 处在删除过程中，则失败返回。否则占用当前 sidtab->next_sid，后者递增 1。

```
                if (context->len)
                        printk(KERN_INFO
                               "SELinux:  Context %s is not valid (left unmapped).\n", context->str);
                ret = sidtab_insert(s, sid, context);
```

然后就调用 sidtab_insert 函数使用该 sid 将当前 context 加入 sidtab。最后通过返回参数 out_sid 返回相应的 sid 值：

```
                if (ret)
                        s->next_sid--;
unlock_out:
                spin_unlock_irqrestore(&s->lock, flags);
        }

        if (ret)
                return ret;

        *out_sid = sid;
        return 0;
}
```

10.7.4 security_transition_sid 函数 - 计算新 subject/object 的 sid

该函数根据 ssid，tsid 以及 tclass 查询 policy.X 的 te_avtab 或 te_cond_avtab 哈希表中相应的 AVTAB_TRANSITION 类规则（type_transition/change/member）。type_transition 规则用于 Domain Transition 或者 New Object Labeling，相应的参数 tsid 分别为可执行文件或者相应目录文件的 sid，查询结果返回新 domain 的 sid 或者新创建文件的 sid；type_change 规则用于在用户登录时改变

tty/console/pty 设备的标签，查询结果返回设备节点被 relabel 后的新 sid；type_member 规则用于指定 polyinstantiated object 的标签，目前仍很少使用。

```
int security_transition_sid(u32 ssid, u32 tsid, u16 tclass, u32 *out_sid)
{
        return security_compute_sid(ssid, tsid, tclass, AVTAB_TRANSITION, out_sid, true);
}
```

无论 type_transition/change 规则都要改变现有 object 的 type，因此在调用 security_compute_sid 函数时指定待查询规则的类型为 AVTAB_TRANSITION，由该函数查询描述在 policydb.te_avtab 或者 te_cond_avtab 哈希表中的相应规则，确定相应 object 的新的 context，并向 sidtab 查询/注册，返回相应的 sid。

```
static int security_compute_sid(u32 ssid, u32 tsid, u16 orig_tclass,
                                u32 specified, u32 *out_sid, bool kern)
{
        struct context *scontext = NULL, *tcontext = NULL, newcontext;
        struct role_trans *roletr = NULL;
        struct avtab_key avkey;
        struct avtab_datum *avdatum;
        struct avtab_node *node;
        u16 tclass;
        int rc = 0;
        bool sock;

        if (!ss_initialized) {
                switch (orig_tclass) {
                case SECCLASS_PROCESS: /* kernel value */
                        *out_sid = ssid;
                        break;
                default:
                        *out_sid = tsid;
                        break;
                }
                goto out;
        }
```

如果 security server 尚未完成初始化，且 orig_tclass 为进程类，则输出 sid 为 subject sid，即不发生安全上下文的改变（注意不仅仅是不发生 domain transition，user，role，MLS level 都不变）。注意此时 policydb 中 policy_class 尚未被初始化（记录 process class 的 policy value），所以直接和 SECCLASS_PROCESS 宏相比较。对于其他情况比如创建新文件，则继承其所在目录的安全上下文（注意，在 policy.X 装载前创建的新文件的 sid 默认等于其所在父目录的 sid。而在 policy.X 装载后的回调函数将调用 inode_doinit_with_dentry 函数，再按照 policy.X 确定相应的 isec->sid）。

```
        context_init(&newcontext);                              # context 数据结构初始化

        read_lock(&policy_rwlock);                              # 在访问 policydb 或者查询 sidtab 期间持有读锁

        if (kern) {
                tclass = unmap_class(orig_tclass);
                sock = security_is_socket_class(orig_tclass);
        } else {
                tclass = orig_tclass;
                sock = security_is_socket_class(map_class(tclass));
        }
```

如果 kern == true，则表示被内核态使用，此时参数 orig_tclass 为 class 的 kernel value，因此在下

面调用 avtab_search 函数查询 TE 规则前得转化为相应的 policy value。如果 kern == false，则表示被用户态使用（比如 compute_create/av/member/relabel 命令），则参数 orig_tclass 已经是 class 的 policy value 了，无须转换（用户态通过/selinux/class/xxx/index 文件得到某个 class 的 policy value）。如果相应的 class 属于某种 socket class，则布尔变量 sock 为真。

```
scontext = sidtab_search(&sidtab, ssid);
if (!scontext) {
        printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n", __func__, ssid);
        rc = -EINVAL;
        goto out_unlock;
}
tcontext = sidtab_search(&sidtab, tsid);
if (!tcontext) {
        printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n", __func__, tsid);
        rc = -EINVAL;
        goto out_unlock;
}
```

然后在 sidtab 中查找 subject/target 的 context 数据结构。由于 sid 是在相应 context 数据结构向 sidtab 注册时获得的（二者存在 1:1 映射关系），因此没有理由找不到，否则代表了一个内核错误。

接下来就是要确定 newcontext 的各个域了。

```
/* Set the user identity. */
switch (specified) {
case AVTAB_TRANSITION:
case AVTAB_CHANGE:
        /* Use the process user identity. */
        newcontext.user = scontext->user;
        break;
case AVTAB_MEMBER:
        /* Use the related object owner. */
        newcontext.user = tcontext->user;
        break;
}
```

首先确定 user，对于 type_transition/change 规则，它和 subject context 的 user 保持一致，从而使得子进程继承父进程的 user，被 relabel 的登录设备的 user 和用户的相同。type_member 用于指定多态下 memberdir 的 type，此时 user 和所在父目录的保持一致。

```
/* Set the role and type to default values. */
if (tclass == policydb.process_class || sock == true) {
        /* Use the current role and type of process. */
        newcontext.role = scontext->role;
        newcontext.type = scontext->type;
} else {
        /* Use the well-defined object role. */
        newcontext.role = OBJECT_R_VAL;                         # define OBJECT_R_VAL 1
        /* Use the type of the related object. */
        newcontext.type = tcontext->type;
}
```

然后确定 role/type。由上面可知，tclass 为当前 class 的 policy value。如果属于 process 或者 socket 类，则默认保持创建者的 role/MLS 属性；其他类，比如新创建的文件或者目录，role 总是为“object_r”，type 继承于其所在父目录文件的 type。

注意，这里正是 SELinux 缺省规则的实现之处：子进程继承父进程的 sid，文件继承其父目录的 sid！

在确定了缺省的上下文之后，下面就要向 Security Server 查询相应的规则了，以重载缺省实现：

```
/* Look for a type transition/member/change rule. */
avkey.source_type = scontext->type;                    # subject type
avkey.target_type = tcontext->type;                    # object type
avkey.target_class = tclass;                           # policy value for object class
avkey.specified = specified;                           # type_transition/change/member
avdatum = avtab_search(&policydb.te_avtab, &avkey);
```

policy.X 中的所有永久 TE 规则用 policydb.te_avtab 哈希表描述。规则的输入和输出分别用其中的 avtab_key 和 avtab_datum 成员来表示。这里根据规则的输入查找 policydb.te_avtab 哈希表，得到相应规则的输出。

```
/* If no permanent rule, also check for enabled conditional rules */
if (!avdatum) {
        node = avtab_search_node(&policydb.te_cond_avtab, &avkey);
        for (; node; node = avtab_search_node_next(node, specified)) {
                if (node->key.specified & AVTAB_ENABLED) {
                        avdatum = &node->datum;
                        break;
                }
        }
}
```

如果没有在 policydb.te_avtab 中找到相应的规则描述，则在 policydb.te_cond_avtab 中继续寻找。注意，对于在 te_cond_avtab 中匹配的规则，必须检查其 avtab_key_t.specified 中 AVTAB_ENABLED 标志位是否被设置。

受 boolean 控制的 conditional block 的两个分支的所有规则都被加入 te_cond_avtab 哈希表，当前生效的规则由 conditional block 条件表达式的状态值决定。如果 boolean 的状态值反转，相应条件表达式的值反转，则相关规则的 AVTAB_ENABLED 标志被改变。此即为通过 boolean 来调整运行时规则库行为的实现方法！

```
if (avdatum) {
        /* Use the type from the type transition/member/change rule. */
        newcontext.type = avdatum->data;
}
```

如果存在相应的永久规则，或者相应的条件规则有效，则一定能够查找到相应规则的输出并设置 newcontext.type；否则它保持之前默认规则的设置。

至此，newcontext 的 user/role/type 都基本确定，下面还要考虑可能的 Role Transition。

```
/* Check for class-specific changes. */
if  (tclass == policydb.process_class) {
        if (specified & AVTAB_TRANSITION) {
                /* Look for a role transition rule. */
                for (roletr = policydb.role_tr; roletr; roletr = roletr->next) {
                        if (roletr->role == scontext->role && roletr->type == tcontext->type) {
                                /* Use the role transition rule. */
                                newcontext.role = roletr->new_role;
                                break;
                        }
                }
```

```
        }
    }
```

类似于 Domain Transition，一个进程 exec 可执行文件时也可能发生 Role Transition，新的 role 由 role_transition 规则指定，比如：

```
role_transition $2 syslogd_initrc_exec_t system_r;
```

即当进程执行 syslogd_initrc_exec_t 文件时，其角色切换到 system_r。所有 role_transition 规则由 policydb.role_tr 队列来描述，如果找到和 subject role、object type 对应的 role_trans 对象，则修改 newcontext.role 为指定的新角色。体会：role_transition 规则的语义正是在这里实现的。

另外，后来在用户态 libsepol 和 SELinux 内核态都添加了 role_transition 规则的 class 支持，使得 role_transition 规则能够被应用于各种 class，参见第 9 章中相应章节。

至此，newcontext 的 user/role/type 完全确定，下面继续确定其 mls_range。

```
    /* Set the MLS attributes. This is done last because it may allocate memory. */
    rc = mls_compute_sid(scontext, tcontext, tclass, specified, &newcontext, sock);
    if (rc)
        goto out_unlock;
```

注意在 mls_compute_sid 的当前实现中，如果是 process 或和 socket 类对象，则完全拷贝创建者的 MLS attribute，否则只拷贝创建者的 current/low level。

在发生 Domain Transition 的同时可能发生 Range Transition，即进程在执行可执行程序期间其 MLS security level 会发生改变，由 range_transition 规则指定，比如：

```
range_transition $1 samhain_exec_t:process mls_systemhigh;
```

即当进程执行 samhain_exec_t 文件时，切换到 mls_systemhigh 安全级别中。所有 range_transition 规则由 policydb.range_tr 队列来描述，根据 newcontext 寻找相应的 range_transition 规则以确定 newcontext.range。注意对于 type_change 规则如果 object class 不是 process（比如登录设备文件），则 newcontext.range.level[0]/[1]都设置为 subject context.range.level[0]，这正是当用户在当前 terminal 设备上切换到更高安全级别时相应 terminal 设备的 current/low 安全级别随之改变的原因！

至此 newcontext 完全拼接完毕，接下来向 policydb 确认其有效性：

```
    /* Check the validity of the context. */
    if (!policydb_context_isvalid(&policydb, &newcontext)) {
        rc = compute_sid_handle_invalid_context(scontext, tcontext, tclass, &newcontext);
        if (rc)
            goto out_unlock;
    }
```

在 policydb_context_isvalid 函数中首先将 user/role/type 和 policydb.p_xxx.nprim 向比较，后者记录了相应 p_xxx.tables 中所有元素 value 的最大值。然后通过 policydb.xxx_val_to_struct 数组得到相应语法成分的 xxx_datum，确认 user-role，role-type，user-range 是否合法。所有这些检查都是通过查看相应成分的 policy value 是否被包含在 xxx_datum 数据结构的 ebitmap 位图中完成的。

注意在 policydb_context_isvalid > mls_context_isvalid 的当前实现中，就“object_r”绕过了上述所有检查。所以如果新创建对象的角色不等于“object_r”，则在相应 pp 的实现中必须定义相应规则以满足检查。

如果无效，则进一步调用 compute_sid_handle_invalid_context 函数处理：它将 scontext/tcontext/newcontext 都转化为字符串，调用 audit_log 函数打印日志。

```
        /* Obtain the sid for the context. */
        rc = sidtab_context_to_sid(&sidtab, &newcontext, out_sid);
out_unlock:
        read_unlock(&policy_rwlock);
        context_destroy(&newcontext);
out:
        return rc;
}
```

最后就可以把有效的 newcontext 注册到 sidtab 中了：如果存在则返回对应 sid，否则注册返回相应 sid。最后释放 policy_rwlock 读锁，以及 newcontext 数据结构（在向 sidtab 注册时会创建其拷贝）。

10.7.5 创建 context 并注册到 sidtab 以获得 sid 的时机

任何内核数据结构的安全属性中都以 u32 sid 描述相应数据结构的安全上下文。sid 有效说明对应的 context 数据结构已经存在并且在 sidtab 中注册过了。每当得到一个新的 context 时（比如进入到新的 domain 或者创建新的文件或内核对象时），都需要调用 sidtab_context_to_sid 函数向 sidtab 查询它所对应的 sid，如果不存在则注册之并返回相应的 sid。

由 sidtab_context_to_sid 函数的调用者创建 context 数据结构，我们在这里整理所有可能的调用者。

1，Initial SID string 向 sidtab 注册时

```
  security_load_policy
> policydb_load_isid
> sidtab_insert
```

2，Domain Transition 时
当进程 exec 可执行文件时可能会发生 Domain Transition，根据当前进程 tsec->sid 以及可执行文件 isec->sid 查询 policy 得到新的 domain 和新的 context。

```
  selinux_bprm_set_security
> security_transition_sid
> security_compute_sid                  # 根据 subject/target context，计算 newcontext
> sidtab_context_to_sid
```

3，创建新文件时

```
  vfs_create
> may_create
> security_transition_sid
> security_compute_sid
> sidtab_context_to_sid

  vfs_create
> ...
> security_inode_init_security == selinux_inode_init_security
> security_compute_sid
> sidtab_context_to_sid
```

4，当用户写入/proc/<pid>/attr/*文件时
用户通过 attr/下的 current/exec 文件来显式指定当前进程的 SC 字符串，或者下一次执行 exec 时切换到的 SC 字符串。

```
  selinux_setprocattr
> security_context_to_sid              # 将 SC 字符串转化为一个 context 数据结构
> sidtab_context_to_sid
```

10.7.6 security_context_to_sid 函数 - 返回 SC 字符串对应的 sid

由上文得知，当用户将新的安全上下文字符串写入/proc/<pid>/attr/*文件时，selinux_setprocattr 函数通过 security_context_to_sid 函数创建、注册相应的 context 数据结构并最终返回相应的 sid，然后保存到当前进程 tsec 的相应域中。

```
int security_context_to_sid(const char *scontext, u32 scontext_len, u32 *sid)
{
        return security_context_to_sid_core(scontext, scontext_len, sid, SECSID_NULL, GFP_KERNEL, 0);
}
```

该函数解析 scontext 所指 SC 字符串的各个成员，查询 policydb 中的相应符号表得到各个成员所对应的 xxx_datum 数据结构，然后确定 SC 字符串相对应的 context 数据结构，最后将其注册到 sidtab 中并返回相应的 sid。

```
static int security_context_to_sid_core(const char *scontext, u32 scontext_len,
                                        u32 *sid, u32 def_sid, gfp_t gfp_flags, int force)
{
        char *scontext2, *str = NULL;
        struct context context;
        int rc = 0;

        if (!ss_initialized) {
                int i;

                for (i = 1; i < SECINITSID_NUM; i++) {
                        if (!strcmp(initial_sid_to_string[i], scontext)) {
                                *sid = i;
                                return 0;
                        }
                }
                *sid = SECINITSID_KERNEL;
                return 0;
        }
```

如果 Security Server 尚未初始化，则在 initial_sid_to_string 数组中查询和指向当前安全上下文字符串的指针的索引。注意，如果找不到则一律返回“kernel_t”的 Initial SID，这也正是系统初始化期间大多数内核设施和数据结构的标签。

```
        *sid = SECSID_NULL;                     /* 0x0, unspecified SID */

        /* Copy the string so that we can modify the copy as we parse it. */
        scontext2 = kmalloc(scontext_len + 1, gfp_flags);
        if (!scontext2)
                return -ENOMEM;
        memcpy(scontext2, scontext, scontext_len);
        scontext2[scontext_len] = 0;
```

为了下面将安全上下文字符串解析为 context 数据结构，这里先复制一份拷贝，以便下面调用 string_to_context_struct 函数时将字符串中的冒号替换为 NULL 字符。

```
        if (force) {
                /* Save another copy for storing in uninterpreted form */
                str = kstrdup(scontext2, gfp_flags);
                if (!str) {
                        kfree(scontext2);
                        return -ENOMEM;
                }
        }
        read_lock(&policy_rwlock);
        rc = string_to_context_struct(&policydb, &sidtab, scontext2, scontext_len, &context, def_sid);
        if (rc == -EINVAL && force) {
                context.str = str;
                context.len = scontext_len;
                str = NULL;
        } else if (rc)
                goto out;
```

现在就可以通过 string_to_context_struct 函数，解析 scontext2 所指安全上下文的各个成分查询 policydb 中相应的符号表得到对应的 xxx_datum 数据结构，将 xxx_datum.value 赋予 context 数据结构的相应域。注意，不能单纯地通过冒号的分割完成转换，同时还要查询 policy.X 以验证安全上下文的正确性。另外，如果为上下文无效且 force 标志被设置，则将其复制到 context->str 中。

```
        rc = sidtab_context_to_sid(&sidtab, &context, sid);
        context_destroy(&context);
out:
        read_unlock(&policy_rwlock);
        kfree(scontext2);
        kfree(str);
        return rc;
}
```

最后就可以调用 sidtab_context_to_sid 函数将 context 数据结构注册到 sidtab 中并返回一个 sid 了。注意在 sidtab_insert 函数中会拷贝 context 数据结构到 sidtab_node.context。尽管 context 为局部变量，还需要调用 context_destroy 函数释放其内指针所指空间。

10.7.7 sidtab_search_core 函数 - sidtab_node 的查找

查询 sidtab 哈希表的工作由 sidtab_search_core 函数完成:

```
struct context *sidtab_search(struct sidtab *s, u32 sid)
{
        return sidtab_search_core(s, sid, 0);
}

struct context *sidtab_search_force(struct sidtab *s, u32 sid)
{
        return sidtab_search_core(s, sid, 1);
}

static struct context *sidtab_search_core(struct sidtab *s, u32 sid, int force)
{
        int hvalue;
        struct sidtab_node *cur;
```

```
        if (!s)
                return NULL;

        hvalue = SIDTAB_HASH(sid);
        cur = s->htable[hvalue];
        while (cur && sid > cur->sid)
                cur = cur->next;
```

首先以当前 sid 的末 7 位作为散列值，遍历相应的冲突项单向链表。链表元素以 sid 从小到大排列，找到第一个其 sid 不小于参数的 sidtab_node 元素。

```
        if (force && cur && sid == cur->sid && cur->context.len)
                return &cur->context;
```

如果 sid 相等，则返回其 context 数据结构的地址。（为什么要检查 force?）

```
        if (cur == NULL || sid != cur->sid || cur->context.len) {
                /* Remap invalid SIDs to the unlabeled SID. */
                sid = SECINITSID_UNLABELED;
                hvalue = SIDTAB_HASH(sid);
                cur = s->htable[hvalue];
                while (cur && sid > cur->sid)
                        cur = cur->next;
                if (!cur || sid != cur->sid)
                        return NULL;
        }

        return &cur->context;
}
```

注意，除了 Initial SID 是静态分配的之外，所有其他 sid 都是在运行时动态分配的-在向 sidtab 注册一个 context 数据结构时。如果找不到和当前 sid 对应的 sidtab_node 数据结构，则说明它是无效的，只能把它映射到“unlabeled”Initial SID，查找并返回相应的 context 数据结构。

10.7.8 security_sid_to_context_core 函数 - 返回 sid 所对应的 SC 字符串

当新创建一个文件时，它的安全上下文 isec->sid 可能由如下方式决定：Mount-Point Labeling，即采用其所在文件系统挂载点的标签；继承其所在父目录的标签；由当前进程写入其/proc/<pid>/attr/fscreate 文件指定；由相应的 type_transition 规则指定。无论何种方式内核中以 u32 数据类型来描述 sid，但是在创建新文件时还需要把其安全上下文写入其磁盘索引节点的“security.linux”扩展属性中（参见上文），这就需要查询 sidtab 哈希表返回其 sid 所对应的安全上下文字符串。

```
int security_sid_to_context(u32 sid, char **scontext, u32 *scontext_len)
{
        return security_sid_to_context_core(sid, scontext, scontext_len, 0);
}

int security_sid_to_context_force(u32 sid, char **scontext, u32 *scontext_len)
{
        return security_sid_to_context_core(sid, scontext, scontext_len, 1);
}
```

security_sid_to_context 函数返回和参数 sid 对应的安全上下文字符串，注意该函数的调用者只传递指向该字符串的指针地址，而由函数自身分配字符串本身的空间。上层调用者（比如 ext2_init_security

函数）在使用后负责释放。

```
static int security_sid_to_context_core(u32 sid, char **scontext, u32 *scontext_len, int force)
{
        struct context *context;
        int rc = 0;

        if (scontext)
                *scontext = NULL;                       # 将指针初始化为 NULL
        *scontext_len  = 0;

        if (!ss_initialized) {
                if (sid <= SECINITSID_NUM) {
                        char *scontextp;

                        *scontext_len = strlen(initial_sid_to_string[sid]) + 1;
                        if (!scontext)
                                goto out;
                        scontextp = kmalloc(*scontext_len, GFP_ATOMIC);
                        if (!scontextp) {
                                rc = -ENOMEM;
                                goto out;
                        }
                        strcpy(scontextp, initial_sid_to_string[sid]);
                        *scontext = scontextp;
                        goto out;
                }
```

如果 Security Server 尚未初始化，则有效的 sid 只能是固化在 SELinux 内核驱动中的 Initial SID。如果参数 sid 属于 Initial SID，则直接从 initial_sid_to_string 数组中拷贝相应的字符串；否则打印出错信息并返回：

```
                printk(KERN_ERR "SELinux: %s:  called before initial "
                       "load_policy on unknown SID %d\n", __func__, sid);
                rc = -EINVAL;
                goto out;
        }

        read_lock(&policy_rwlock);
        if (force)
                context = sidtab_search_force(&sidtab, sid);
        else
                context = sidtab_search(&sidtab, sid);
        if (!context) {
                printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n",
                        __func__, sid);
                rc = -EINVAL;
                goto out_unlock;
        }
```

然后在持有 policy_rwlock 读锁的情况下查询 sidtab 哈希表，返回该 sid 所对应的 sidtab_node 数据结构中的 context 数据结构的地址。

```
        rc = context_struct_to_string(context, scontext, scontext_len);
out_unlock:
        read_unlock(&policy_rwlock);
out:
        return rc;
```

}

最后再将该 context 数据结构的内容转化为字符串。由于 context 数据结构中 user/role/type 都是 u32 类型，因此需要查询 policydb 数据结构中的 p_user_val_to_name，p_role_val_to_name 和 p_type_val_to_name 指针数组，以 u32 数值索引相应的指针数组得到字符串表示，分配合适的空间并逐一拷贝字符串。

10.8 Class Mapping

10.8.1 Class Mapping 的作用

在 refpolicy 中所有 class 及 permission 在 flask/下 security_classes 和 access_vectors 文件中定义。在编译时 checkpolicy 为它们分配相应的索引并保存在 policy.X 中。在装载 policy.X 时，/selinux/load 文件的驱动解析其二进制表示并创建相应的内核描述符。

而 SELinux 内核驱动所定义（支持）的 class 及其权限定义在 classmap.h 文件中，每一个 class 及其权限都由字符串表示，由一个 security_class_mapping 数据结构描述。在编译 SELinux 内核驱动时 genheaders 程序遍历 classmap.h 文件，为每个 class 及其权限创建相应的宏定义，保存在 flask.h 和 av_permissions.h 文件中。SELinux 内核驱动正是使用这两个文件定义的宏来描述某个 class 及其权限（所以它们是“先使用，后定义”）。

对一个 class 而言，用户态的 checkpolicy 及内核工具 genheaders 都为它**独立地**分配索引值，我们称前者为 policy value，而称后者为 kernel value。SELinux 内核驱动使用 class kernel value，但是在 checkpolicy 生成的规则的二进制表示中却使用 class policy value，因此在 security_compute_sid 和 security_compute_av 函数查询 policy.X 前必须首先将 kernel value 转换为 policy value。

10.8.2 Class Mapping 的创建

class 的 policy value 和 kernel value 之间的转换借助 current_mapping[]数组实现，数组元素的数据结构定义如下：

```
static struct selinux_mapping *current_mapping;
static u16 current_mapping_size;

struct selinux_mapping {
        u16 value;                              # class policy value
        unsigned num_perms;                     # class 含有的 perm 个数
        u32 perms[sizeof(u32) * 8];             # class 内 perm 的 policy value，一个 class 最多支持 32 个 perm
};
```

以 class 的 kernel value 索引 current_mapping 数组，相应的 selinux_mapping.value 即为该 class 的 policy value（unmap_class 函数）。以权限的 kernel value 索引相应 class 的 selinux_mapping.perms[]数组，即得到该权限的 policy value。

kernel value 和 policy value 之间的这种“1 对 1 映射”是在 security_load_policy > selinux_set_mapping 时建立的，后者遍历 classmap.h 文件中所有 security_class_mapping 数据结构，根据 class 名字字符串查询 p_classes 符号表，得到相应 class_datum_t 数据结构的地址，从而得到其 policy value。进而查询其 class_datum_t.permissions 符号表即得到相应权限的 policy value。把它们都保存在 selinux_mapping 数据结构中。

```
static int selinux_set_mapping(struct policydb *pol,
                               struct security_class_mapping *map,     # class/perm kernel definition
                               struct selinux_mapping **out_map_p,
                               u16 *out_map_size)
{
        struct selinux_mapping *out_map = NULL;
        size_t size = sizeof(struct selinux_mapping);
        u16 i, j;
        unsigned k;
        bool print_unknown_handle = false;

        /* Find number of classes in the input mapping */
        if (!map)
                return -EINVAL;
        i = 0;
        while (map[i].name)
                i++;
```

首先计算 secclass_map 数据结构中含有的元素个数，即 SELinux 内核驱动所定义的 class 字符串及其 perm 字符串的集合的个数。

```
        /* Allocate space for the class records, plus one for class zero */
        out_map = kcalloc(++i, size, GFP_ATOMIC);
        if (!out_map)
                return -ENOMEM;
```

然后分配 current_mapping 数据结构，注意为“Class Zero”多分配了一个元素，而 secclass_map 中是不包含它的。

```
        /* Store the raw class and permission values */
        j = 0;
        while (map[j].name) {
                struct security_class_mapping *p_in = map + (j++);              # secclass_map
                struct selinux_mapping *p_out = out_map + j;                    # current_mapping
```

在循环中遍历 secclass_map 集合中的各个元素。注意，current_mapping[0]被预留给了“Class Zero”，secclass_map[0]对应 current_mapping[1]，以此类推。相应地，genheaders 程序也使得 secclass_map[0]对应的 SECCLASS_XXX 从 1 开始分配。

```
                /* An empty class string skips ahead */
                if (!strcmp(p_in->name, "")) {
                        p_out->num_perms = 0;
                        continue;
                }
```

对于名称字符串为 NULL 的 secclass_map 元素，相应 current_mapping[]中 value 和 num_perms 都为 0（上面使用了 kcalloc 函数）。

```
                p_out->value = string_to_security_class(pol, p_in->name);
                if (!p_out->value) {
                        printk(KERN_INFO "SELinux:  Class %s not defined in policy.\n", p_in->name);
                        if (pol->reject_unknown)
                                goto err;
                        p_out->num_perms = 0;
                        print_unknown_handle = true;
                        continue;
                }
```

然后就可以用 class 的名称字符串查询 p_classes 符号表了，并返回其 class_datum.value 的值。注意，如果查询失败则直接返回 0。所以对于那些只在 SELinux 内核驱动（即 classmap.h 中）定义但是 refpolicy 中没有定义的 class，如果 policydb_t.rej_unknown 标志有效（由 refpolicy 的 build.conf 文件中的 UNK_PERMS 变量的值决定），则装载 policy.X 的过程失败退出。否则，设置该 class 的 policy value 和 num_perms 都为 0。

相应 current_mapping[].value 和 num_perms 都为 0。

然后在内层循环中处理当前 class 的 perm 字符串指针数组:

```
                k = 0;
                while (p_in->perms && p_in->perms[k]) {
                        /* An empty permission string skips ahead */
                        if (!*p_in->perms[k]) {
                                k++;
                                continue;
                        }
                        p_out->perms[k] = string_to_av_perm(pol, p_out->value, p_in->perms[k]);
                        if (!p_out->perms[k]) {
                                printk(KERN_INFO
                                       "SELinux:  Permission %s in class %s not defined in policy.\n",
                                       p_in->perms[k], p_in->name);
                                if (pol->reject_unknown)
                                        goto err;
                                print_unknown_handle = true;
                        }

                        k++;
                }
                p_out->num_perms = k;

        }
```

还是同样的处理方法，查询 class_datum 相关的 common 哈希表，或者该 class 自定义的 permssion 哈希表，得到相应名称 perms 的 policy value，记录到 current_mapping[].perms[]元素中。同样，如果一个 perm 只在内核中定义但是在 refpolicy 中没有定义，则根据当前 policydb_t.rej_unknown 标志位来进行相应的处理。

```
        if (print_unknown_handle)
                printk(KERN_INFO "SELinux: the above unknown classes and permissions will be %s\n",
                       pol->allow_unknown ? "allowed" : "denied");
```

如果发现了任何 class/perm 只在内核中定义但是在 refpolicy 中没有定义的情况，则打印提示信息，以及 policydb_t.allow_unknown 标志的值。

```
        *out_map_p = out_map;
        *out_map_size = i;
        return 0;
err:
        kfree(out_map);
        return -EINVAL;
}
```

10.8.3 Class Mapping 的使用 - class/perm 内核态和用户态索引的转换

security_compute_sid 函数的 orig_tclass 参数为调用者所使用的 class value，可能为 kernel value，也可能为 policy value。如果参数 kern == true 即表示被内核使用，则需要调用 unmap_class 函数将 kernel value 转换为 policy value 以便调用 avtab_search 函数查询 TE 规则：

```
        if (kern) {
                tclass = unmap_class(orig_tclass);
                sock = security_is_socket_class(orig_tclass);
        } else {
                tclass = orig_tclass;
                sock = security_is_socket_class(map_class(tclass));
        }
```

与此相对，如果 kern == false 则表示被用户态所调用，则在需要使用 kernel value 的场合（比如需要传递给 security_is_socket_class 函数时）首先需要调用 map_class 函数将 policy value 转换为 kernel value（参见上文 9.2 小节“Separate Socket SID 开发”）。

unmap_class 和 map_class 函数的实现如下：

```
/*
 * Get real, policy values from mapped values
 */
static u16 unmap_class(u16 tclass)
{
        if (tclass < current_mapping_size)
                return current_mapping[tclass].value;

        return tclass;
}

/*
 * Get kernel value for class from its policy value
 */
static u16 map_class(u16 pol_value)
{
        u16 i;

        for (i = 1; i < current_mapping_size; i++) {
                if (current_mapping[i].value == pol_value)
                        return i;
        }

        return SECCLASS_NULL;
}
```

注意在 map_class 函数中，由于 current_mapping[0]对应“Class Zero”，所以从 current_mapping[1]开始查找。如果在 current_mapping 中没有找到任何匹配的元素，则必须返回 SECCLASS_NULL，以表示这个 class 在 SELinux 内核驱动中没有定义（比如用户态 Object Manager 例如 X/dbus 就会使用只由它们定义和管理的 class，这些 class 对 SELinux 内核驱动是透明的）。

除了将 class 的内核索引和用户态索引相互转换之外，class mapping 还用于将用户态 perm 索引转换为内核态的索引。

在 security_compute_av > context_struct_compute_av 函数得到就当前(scontext, tcontext,

tclass)的avd结构后，继续调用map_decision函数将avd中的位图：将用户态索引转换为相应的内核态索引：

```
static void map_decision(u16 tclass, struct av_decision *avd, int allow_unknown)
{
        if (tclass < current_mapping_size) {
```

显然，如果一个class只在用户态定义而没有在内核态定义，则无法做转换。

```
                unsigned i, n = current_mapping[tclass].num_perms;
                u32 result;

                for (i = 0, result = 0; i < n; i++) {
                        if (avd->allowed & current_mapping[tclass].perms[i])
                                result |= 1<<i;
                        if (allow_unknown && !current_mapping[tclass].perms[i])
                                result |= 1<<i;
                }
                avd->allowed = result;
```

current_mapping[].perms[]数组以perm的kernel value为索引，相应元素的值为该perm的policy value。如果一个perm的policy value出现在avd的相应位图中，则记录该perm的kernel value。

如果一个perm的policy value为0（即说明它只在用户态定义，而在内核态没有定义），且policydb_t.allow_unknown表示有效，则仍将其算为一个被许可的perm。

最终将avd->allowed由用policy value表示的位图，转换为用kernel value表示的位图！

```
                for (i = 0, result = 0; i < n; i++)
                        if (avd->auditallow & current_mapping[tclass].perms[i])
                                result |= 1<<i;
                avd->auditallow = result;

                for (i = 0, result = 0; i < n; i++) {
                        if (avd->auditdeny & current_mapping[tclass].perms[i])
                                result |= 1<<i;
                        if (!allow_unknown && !current_mapping[tclass].perms[i])
                                result |= 1<<i;
                }
                /*
                 * In case the kernel has a bug and requests a permission
                 * between num_perms and the maximum permission number, we
                 * should audit that denial
                 */
                for (; i < (sizeof(u32)*8); i++)
                        result |= 1<<i;
                avd->auditdeny = result;
        }
}
```

然后用同样的方法处理avd->auditallow和auditdeny位图。

10.8.4 增加class或者权限的方法

就每个class的每个权限，都需要某个SELinux内核回调函数以检查当前执行流是否具有所必须的权限。因此如果要添加一个新的class或者权限，必须：

1，修改 refpolicy
　　在 flask.h/security_classes 和 access_vectors 文件的最后定义；
　　定义赋予该权限的接口；
　　就相应 domain 调用该接口；
2，修改内核驱动
　　在 classmap.h 中定义；
　　在合适的 SELinux 回调函数中通过 avc_has_perm 函数检查；

注意，如果内核 classmap.h 中定义的 class/perm 在 policy.X 中没有定义，则根据 refpolicy 的 build.conf 文件中定义的 UNK_PERMS 来决定如何处理（reject/deny/allow）。UNK_PERMS 的值决定 policy.X 中 reject_unknown 和 allow_unknown 位的值。

参见 http://selinuxproject.org/page/Adding_New_Permissions

10.9 和文件操作相关的回调函数

在 SELinux 内核驱动向 LSM 注册的回调函数 selinux_ops 中，和文件操作相关的回调函数如下：

```
.file_permission =              selinux_file_permission,
.file_alloc_security =          selinux_file_alloc_security,
.file_free_security =           selinux_file_free_security,
.file_ioctl =                   selinux_file_ioctl,
.file_mmap =                    selinux_file_mmap,
.file_mprotect =                selinux_file_mprotect,
.file_lock =                    selinux_file_lock,
.file_fcntl =                   selinux_file_fcntl,
.file_set_fowner =              selinux_file_set_fowner,
.file_send_sigiotask =          selinux_file_send_sigiotask,
.file_receive =                 selinux_file_receive,
```

本章节介绍上述函数的实现方法，并顺带介绍 LSM 回调函数所在内核子系统（Object Manager）相关的运行机制，以及相应 LSM 回调函数的调用时机。

10.9.1 selinux_file_mprotect 回调函数

应用程序可以使用 mprotect 命令来指定进程地址空间中[addr，addr+len-1]区域内页框的访问权限：

```
#include <sys/mman.h>
int mprotect(const void *addr, size_t len, int prot)
```

注意 addr 必须页地址对齐，参数 len 也会被 mprotect 系统调用处理函数对齐到页大小（对齐到下一个页框边界处），参数 prot 为相应页框的新的保护权限（访问权限）。

将栈、堆、或者匿名映射所分配的内存空间设置为可执行的（PROT_EXEC）等于开启了严重的安全漏洞：黑客一般都是利用软件自身的缺陷（比如 Stack Overflow）在进程地址空间中植入可执行代码，并且想法跳转并执行该段代码（比如覆盖 Stack 内保存的返回地址），从而获得当前进程相同的权限。所以应该尽量限制可执行的地址空间的大小。

LSM 在 mprotect 系统调用处理函数中安装了 security_file_mprotect 回调函数。mprotect 系统调用处理函数将在一个循环内处理[addr，addr+len-1]所涉及的所有 VMA，就每个 VMA 都通过 security_file_mprotect 函数检查是否许可当前进程按照指定的方式设置 VMA 中页框的访问权限。

SELinux 实现的回调函数如下，参数 reqprot 为用户态应用程序调用 mprotect 函数使用的实际参数，而 prot 为内核修正后的参数（如果 reqprot 中 PROT_READ 有效，而当前体系结构上 PROT_READ 包含 PROT_EXEC，则补充后者）：

```
static int selinux_file_mprotect(struct vm_area_struct *vma, unsigned long reqprot, unsigned long prot)
{
        const struct cred *cred = current_cred();

        if (selinux_checkreqprot)
                prot = reqprot;
```

全局标志 selinux_checkreqprot 可以修正 mprotect/mmap 系统调用中相应回调函数的行为：检查应用程序的原始参数、或者检查内核修正后的参数，其数值由 CONFIG_SECURITY_SELINUX_CHECKREQPROT_VALUE 决定，默认为 1，即检查用户态的原始参数。由于函数实现中检查 prot，所以将 prot 重新设置为 reqport。

全局变量 default_noexec 在 selinux_init 函数中按照如下方式初始化：

```
default_noexec = !(VM_DATA_DEFAULT_FLAGS & VM_EXEC);
```

其中 VM_DATA_DEFAULT_FLAGS 的定义如下，即为当前体系结构上 VMA 所涉及页框的默认保护模式：

```
#define VM_DATA_DEFAULT_FLAGS \
        (((current->personality & READ_IMPLIES_EXEC) ? VM_EXEC : 0 ) | \
                VM_READ | VM_WRITE | VM_MAYREAD | VM_MAYWRITE | VM_MAYEXEC)
```

页框的默认保护模式支持读写，如果当前体系结构上读操作“包含”执行操作，即 READ_IMPLIES_EXEC 标志位有效，则默认地也支持执行操作。

所以，default_noexec 用于描述 VMA 所对应的页框是否默认地设置了 PROT_EXEC 标志。

```
        if (default_noexec && (prot & PROT_EXEC) && !(vma->vm_flags & VM_EXEC)) {
```

default_noexec 等于 1（即页框默认没有被设置 PROT_EXEC 标志）但是当前用户态要求设置该标志，且进一步确认 VMA 自身访问模式中没有设置 VM_EXEC 标志，所以需要修改 VMA 和页框的访问模式加入可执行标志。

如上文所述，使得一个 VMA（或者对应的页框）即可写又可执行（无论栈、堆、或匿名映射以 COW 方式获得的页框）是一个严重的安全漏洞，所以 SELinux 需要检查当前进程 domain 是否有相应的能力。

```
                int rc = 0;
                if (vma->vm_start >= vma->vm_mm->start_brk && vma->vm_end <= vma->vm_mm->brk) {
                        rc = cred_has_perm(cred, cred, PROCESS__EXECHEAP);
```

如果该 VMA 属于进程的堆，则进程要求将自己的堆变成可写可执行，则当前 domain 必须具备 PROCESS_EXECHEAP 能力。

```
                } else if (!vma->vm_file &&
                           vma->vm_start <= vma->vm_mm->start_stack &&
                           vma->vm_end >= vma->vm_mm->start_stack) {
                        rc = current_has_perm(current, PROCESS__EXECSTACK);
```

如果该 VMA 属于进程的栈，则进程要求将自己的栈变成可写可执行，则当前 domain 必须具备 PROCESS_EXESTACK 能力。

另外，绝大多数情况下都不需要将栈或堆设置为可执行的，这往往是一个严重的安全漏洞。对于真的需要在运行时生成可执行代码的情况，也应该两次调用 mmap 建立起对同一个临时匿名文件的两个映射：一个为可写，另一个为可执行，参见 Ulrich Drepper 的文章《SELinux Memory Protection Tests》。

```
                } else if (vma->vm_file && vma->anon_vma) {
                        /*
                         * We are making executable a file mapping that has
                         * had some COW done. Since pages might have been
                         * written, check ability to execute the possibly
                         * modified content.  This typically should only
                         * occur for text relocations.
                         */
                        rc = file_has_perm(cred, vma->vm_file, FILE__EXECMOD);
                }
```

vma->vm_file 指针不为 NULL，说明当前 VMA 映射了一个正规文件；vma->anon_vma 指针不为 NULL，说明和该 VMA 中的某些页面建立了私有映射（所以才在写操作时以 COW 方式获得新页框，而该新页框不对应原始被映射的文件，不再在 page cache 中，所以为匿名映射）。综合二者，则说明该 VMA 私有映射了一个文件而且某些页面已经发生了写入操作，和写操作相关的页表项也指向了以 COW 方式获得的新页框（不再指向 page cache 中和文件相关的页框）。

由于现在要将该 VMA 设置为可执行，所以要求当前 domain 对该文件具有 FILE_EXECMOD 能力（即，对修改了的部分有执行能力）。

正如注释所言，要求对私有映射文件可执行，只有在相应可执行文件为代码相关（需要在装载时发生代码段的重定位）时才会发生（推测：Loader 如果发现可执行文件的.text 为代码相关的，则建立私有文件映射）。**而位置相关代码往往是一个编译时的 bug**：没有指示编译器生成位置无关代码，或者编译器没有做到这一点。之前发现某个体系结构的交叉编译器没有正常支持-pie 选项，导致 mount 等应用程序为位置相关的，则在系统启动时得到错误信息如下：

```
type=1400 audit(946684818.974:3): avc:  denied  { execmod } for  pid=953 comm="mount" path="/bin/mount"
dev=mmcblk0p1 ino=72470 scontext=system_u:system_r:mount_t:s0-s15:c0.c255
tcontext=system_u:object_r:mount_exec_t:s0 tclass=file
mount: error while loading shared libraries: cannot restore segment prot after reloc: Permission denied
type=1400 audit(946684819.005:4): avc:  denied  { execmod } for  pid=954 comm="mount" path="/bin/mount"
dev=mmcblk0p1 ino=72470 scontext=system_u:system_r:mount_t:s0-s15:c0.c255
tcontext=system_u:object_r:mount_exec_t:s0 tclass=file
awk: cmd. line:1: fatal: cannot open file `/proc/mounts' for reading (No such file or directory)
```

最后调用 file_map_prot_check 函数进一步检查当前进程 domain 能否将 VMA 设置为指定的保护模式：

```
                if (rc)
                        return rc;
        }

        return file_map_prot_check(vma->vm_file, prot, vma->vm_flags&VM_SHARED);
}
```

file_map_prot_check 函数用于检查当前进程能否建立相应的文件映射，或者改变已有映射的保护模式：

```
static int file_map_prot_check(struct file *file, unsigned long prot, int shared)
{
        const struct cred *cred = current_cred();
        int rc = 0;
```

```
        if (default_noexec && (prot & PROT_EXEC) && (!file || (!shared && (prot & PROT_WRITE)))) {
                /*
                 * We are making executable an anonymous mapping or a
                 * private file mapping that will also be writable.
                 * This has an additional check.
                 */
                rc = cred_has_perm(cred, cred, PROCESS__EXECMEM);
                if (rc)
                        goto error;
        }
```

如果 VMA 默认没有设置可执行标志且现在打算将它设置为可执行的，如果当前 VMA 没有参与文件映射（则只能为匿名映射），或者参与私有可写文件映射（则写入时会以 COW 方式获得匿名映射的页框），则这两种方式都将使进程获得可写且可执行的页框，则要求当前进程 domain 具有 PROCESS_EXECMEM 能力。

```
        if (file) {
                /* read access is always possible with a mapping */
                u32 av = FILE__READ;

                /* write access only matters if the mapping is shared */
                if (shared && (prot & PROT_WRITE))                    # 因为私有文件映射将写入新的页框
                        av |= FILE__WRITE;

                if (prot & PROT_EXEC)
                        av |= FILE__EXECUTE;

                return file_has_perm(cred, file, av);
        }
```

如果为文件映射，则自然还要检查当前进程 domain 对文件是否具有对应的能力。注意，只有共享可写文件映射才需要真正写入文件（而私有可写文件映射将以 COW 获得新的页框，新写入的内容和文件没有任何关系，因此也就不需要对文件的写操作了）。 最后调用 file_has_perm 函数，检查当前进程对打开文件描述符是否具有 use 能力，以及文件本身是否具有相应的能力。

```
error:
        return rc;
}
```

注意，如果相应的文件被重新打标签，或者相应的规则发生改变，那么之前许可的文件操作（比如 read/write/mmap）都需要被重新审查。这要求能够根据改变的标签或者规则，先定位出相关的文件，然后再找到所有相关的、已经许可的文件操作，然后才能支持“完备的撤销”（Full revocation）。而目前尚未实现这一点。（注意，即使定位出相关的进程，但撤销发生时进程可能正在操作文件，如何使得进程优雅地结束文件操作？）

文件的 file 对象（打开文件描述符）和 inode 对象（代表文件自身），在内核中为不同的对象，SELinux 对它们都进行了扩展，要分别检查当前进程 domain 是否具有对它们的访问能力。如果文件描述符不是由当前进程创建（比如从父进程继承），则当前进程必须具备对它的 FD__USE 能力。

如注释所言，如果参数 av 为 0，则表示只需要访问打开文件描述符而不需要访问文件自身，比如执行 lseek 操作时（只需要改变 file->f_ops 指针）

```
/* Check whether a task can use an open file descriptor to
   access an inode in a given way.  Check access to the
   descriptor itself, and then use dentry_has_perm to
   check a particular permission to the file.
   Access to the descriptor is implicitly granted if it
```

```
   has the same SID as the process.  If av is zero, then
   access to the file is not checked, e.g. for cases
   where only the descriptor is affected like seek. */
static int file_has_perm(const struct cred *cred, struct file *file, u32 av)
{
        struct file_security_struct *fsec = file->f_security;
        struct inode *inode = file->f_path.dentry->d_inode;
        struct common_audit_data ad;
        u32 sid = cred_sid(cred);
        int rc;

        COMMON_AUDIT_DATA_INIT(&ad, PATH);
        ad.u.path = file->f_path;

        if (sid != fsec->sid) {
                rc = avc_has_perm(sid, fsec->sid, SECCLASS_FD, FD__USE, &ad);
                if (rc)
                        goto out;
        }
```

如果 fsec->sid 和当前进程的 sid 相同，则相应文件由当前进程打开，相应文件描述符因此而创建。否则说明相应文件不是由当前进程打开（比如 fd 从父进程继承而来），需要检查当前进程是否具有对该打开文件描述符的 FD__USE 能力（即 class fd 的 use 权限）。

```
        /* av is zero if only checking access to the descriptor. */
        rc = 0;
        if (av)
                rc = inode_has_perm(cred, inode, av, &ad, 0);

out:
        return rc;
}
```

最后调用 inode_has_perm 函数检查当前进程 domain 是否能够访问该文件。

10.10 和 AF_UNIX socket 相关的回调函数（todo）

```
        .unix_stream_connect =  selinux_socket_unix_stream_connect,
        .unix_may_send =        selinux_socket_unix_may_send,


static int selinux_socket_unix_stream_connect(struct sock *sock,              # client 端的套接字
                                              struct sock *other,             # server 端的监听套接字
                                              struct sock *newsk)             # server 端的已连接套接字
{
        struct sk_security_struct *sksec_sock = sock->sk_security;
        struct sk_security_struct *sksec_other = other->sk_security;
        struct sk_security_struct *sksec_new = newsk->sk_security;
        struct common_audit_data ad;
        int err;

        COMMON_AUDIT_DATA_INIT(&ad, NET);
        ad.u.net.sk = other;

        err = avc_has_perm(sksec_sock->sid, sksec_other->sid,
```

```
                        sksec_other->sclass, UNIX_STREAM_SOCKET__CONNECTTO, &ad);
        if (err)
                return err;
```

首先调用 avc_has_perm 函数检查 client 端的套接字，对 server 端的监听套接字是否有 unix_stream_socket 类的 connectto 能力。

```
        /* server child socket */
        sksec_new->peer_sid = sksec_sock->sid;
        err = security_sid_mls_copy(sksec_other->sid, sksec_sock->sid, &sksec_new->sid);
        if (err)
                return err;

        /* connecting socket */
        sksec_sock->peer_sid = sksec_new->sid;

        return 0;
}
```

注意 client 端的套接字，以及 server 端的已连接套接字的 sksec->peer_sid 相互指向对方。另外 server 端已连接套接字自身的 sid 的 user/role/type 部分，和 server 端的监听套接字的保持一致；而 mls_range 部分，则和 client 端的套接字的保持一致（从而实现“Labeled Networking”）。

10.11 和程序执行相关的操作（todo）

```
        .bprm_set_creds =               selinux_bprm_set_creds,
        .bprm_committing_creds =        selinux_bprm_committing_creds,
        .bprm_committed_creds =         selinux_bprm_committed_creds,
        .bprm_secureexec =              selinux_bprm_secureexec,
```

selinux_bprm_set_creds 函数在之前的版本中叫做 selinux_bprm_set_security 函数，在上文已经见过。

10.11.1 selinux_bprm_secureexec 函数 - 扩展 AT_SECURE 机制

10.11.1.1 C 库 AT_SECURE 机制介绍

LSM 的 bprm_secureexec 函数用于在内核态由 loader 确定是否需要使能 secure 模式，相应地设置用户栈中 Auxiliary Table 中的 AT_SECURE 项的值。从 execve 系统调用返回后（首先执行 linker 的代码，mmap 动态链接库文件并解析符号），linker 检查 AT_SECURE 项的值，如果为 1（即需要启动 secure 模式），则清除当前进程属于 UNSECURE_ENVVARS 集合（unsecure environment variables）的环境变量。

在 execve 系统调用执行期间，内核会在当前进程的用户栈的栈底保存如下数据结构：NULL，环境变量字符串（以 name=value 的形式），命令行参数字符串，Auxiliary Table，envp[]指针数组，argv[]指针数组，argc，main 函数后的返回地址。

Auxiliary Table（简称为“AT”）的表项称为 Auxiliary Vector，以 id=value 的形式表示。在进程的 mm_struct 数据结构中设计有表示 AT 的数组：

```
        unsigned long saved_auxv[AT_VECTOR_SIZE]; /* for /proc/PID/auxv */
```

在 load_elf_binary > create_elf_binary 函数中，设置 mm->saved_auxv 数组（代码片段摘录如下）：

```
        /* Create the ELF interpreter info */
        elf_info = (elf_addr_t *)current->mm->saved_auxv;
        /* update AT_VECTOR_SIZE_BASE if the number of NEW_AUX_ENT() changes */
#define NEW_AUX_ENT(id, val) \
        do { \
                elf_info[ei_index++] = id; \
                elf_info[ei_index++] = val; \
        } while (0)
```

首先将 elf_info 指针指向 mm->saved_auxv[]数组。由 NEW_AUX_ENT 宏的定义可见，该数组中相邻元素分别表示一个 Auxiliary Vector 的 id 和 value。

```
#ifdef ARCH_DLINFO
        /*
         * ARCH_DLINFO must come first so PPC can do its special alignment of
         * AUXV.
         * update AT_VECTOR_SIZE_ARCH if the number of NEW_AUX_ENT() in
         * ARCH_DLINFO changes
         */
        ARCH_DLINFO;
#endif
```

设置 AT 表格，包含当前平台的信息（比如 AT_PAGESZ），可执行程序自身的信息（比如 AT_PHDR 等）：

```
        NEW_AUX_ENT(AT_HWCAP, ELF_HWCAP);
        NEW_AUX_ENT(AT_PAGESZ, ELF_EXEC_PAGESIZE);
        NEW_AUX_ENT(AT_CLKTCK, CLOCKS_PER_SEC);
        NEW_AUX_ENT(AT_PHDR, load_addr + exec->e_phoff);
        NEW_AUX_ENT(AT_PHENT, sizeof(struct elf_phdr));
        NEW_AUX_ENT(AT_PHNUM, exec->e_phnum);
        NEW_AUX_ENT(AT_BASE, interp_load_addr);
        NEW_AUX_ENT(AT_FLAGS, 0);
        NEW_AUX_ENT(AT_ENTRY, exec->e_entry);
        NEW_AUX_ENT(AT_UID, cred->uid);
        NEW_AUX_ENT(AT_EUID, cred->euid);
        NEW_AUX_ENT(AT_GID, cred->gid);
        NEW_AUX_ENT(AT_EGID, cred->egid);
        NEW_AUX_ENT(AT_SECURE, security_bprm_secureexec(bprm));
        NEW_AUX_ENT(AT_RANDOM, (elf_addr_t)(unsigned long)u_rand_bytes);
        NEW_AUX_ENT(AT_EXECFN, bprm->exec);
        if (k_platform) {
                NEW_AUX_ENT(AT_PLATFORM, (elf_addr_t)(unsigned long)u_platform);
        }
        if (k_base_platform) {
                NEW_AUX_ENT(AT_BASE_PLATFORM, (elf_addr_t)(unsigned long)u_base_platform);
        }
        if (bprm->interp_flags & BINPRM_FLAGS_EXECFD) {
                NEW_AUX_ENT(AT_EXECFD, bprm->interp_data);
        }
#undef NEW_AUX_ENT
```

注意 AT_SECURE 分量的 ID 定义为 23，其值由 security_bprm_secureexec 函数确定。当没有使能 SELinux 时，直接调用 cap_bprm_secureexec 函数：

```
static inline int security_bprm_secureexec(struct linux_binprm *bprm)
{
        return cap_bprm_secureexec(bprm);
}
```

```
int cap_bprm_secureexec(struct linux_binprm *bprm)
{
        const struct cred *cred = current_cred();

        if (cred->uid != 0) {                                   # 暂时跳过这部分代码
                if (bprm->cap_effective)
                        return 1;
                if (!cap_isclear(cred->cap_permitted))
                        return 1;
        }

        return (cred->euid != cred->uid || cred->egid != cred->gid);
}
```

由此可见，如果当前进程执行的是一个 setuid 或者 setgid 程序，则返回 1（表示启动 secure 模式）。这个很好理解：进程在执行 setuid/setgid 程序期间其 euid/egid 将编程程序所有者的 uid/gid（比如 root），如果进程有意在 fork+exec 之前设置自己的环境变量（优先指向含有恶意代码的库文件，而不是系统库文件），则能够以 root 身份执行这些恶意代码。

此时启动 secure 模式，由 linker 在执行程序之前，清除从父进程继承的所有环境变量（sanitize environment），就可以消除这个安全隐患。具体行为在 C 库如下函数中实现：

首先，在 elf/dl-sysdep.c 文件的_dl_sysdep_start 函数中设置读取 AT_SECURE 向量的数值，保存到__libc_enable_secure 变量中：

```
        case AT_SECURE:
#ifndef HAVE_AUX_SECURE
        seen = -1;
#endif
        INTUSE(__libc_enable_secure) = av->a_un.a_val;
        break;
```

然后，在 elf/rtld.c 文件的 process_envvars 函数中根据__libc_enable_secure 标志位处理环境变量：

```
  /* Extra security for SUID binaries.  Remove all dangerous environment variables.  */
  if (__builtin_expect (INTUSE(__libc_enable_secure), 0))
    {
      static const char unsecure_envvars[] =
#ifdef EXTRA_UNSECURE_ENVVARS
        EXTRA_UNSECURE_ENVVARS
#endif
        UNSECURE_ENVVARS;
```

UNSECURE_ENVVARS 集合包含在启动 secure 模式下需要清除的环境变量，定义在 sysdeps/generic/unsecvars.h 文件中，由于每一个字串都是一个 AT 向量的名称字符串，所以每个字串都独立地以“\0”结尾：

```
/* Environment variable to be removed for SUID programs.  The names are
   all stuffed in a single string which means they have to be terminated
   with a '\0' explicitly.  */
#define UNSECURE_ENVVARS \
  "GCONV_PATH\0"                                                \
  "GETCONF_DIR\0"                                               \
  "HOSTALIASES\0"                                               \
  "LD_AUDIT\0"                                                  \
```

```
  "LD_DEBUG\0"                                                    \
  "LD_DEBUG_OUTPUT\0"                                             \
  "LD_DYNAMIC_WEAK\0"                                             \
  "LD_LIBRARY_PATH\0"                                             \
  "LD_ORIGIN_PATH\0"                                              \
  "LD_PRELOAD\0"                                                  \
  "LD_PROFILE\0"                                                  \
  "LD_SHOW_AUXV\0"                                                \
  "LD_USE_LOAD_BIAS\0"                                            \
  "LOCALDOMAIN\0"                                                 \
  "LOCPATH\0"                                                     \
  "MALLOC_TRACE\0"                                                \
  "NIS_PATH\0"                                                    \
  "NLSPATH\0"                                                     \
  "RESOLV_HOST_CONF\0"                                            \
  "RES_OPTIONS\0"                                                 \
  "TMPDIR\0"                                                      \
  "TZDIR\0"
```

显然，LD_PRELOAD 和 LD_LIBRARY_PATH 都需要被清除。

```
    const char *nextp;

    nextp = unsecure_envvars;
    do
      {
        unsetenv(nextp);
        /* We could use rawmemchr but this need not be fast.  */
        nextp = (char *)(strchr)(nextp, '\0') + 1;
      }
    while (*nextp != '\0');
```

然后在一个 while 循环中，就 UNSECURE_ENVVARS 集合中的每一个字符串，调用 unsetenv 函数删除相应的环境变量。注意 strchr 函数返回当前字符串结尾“\0”字符的位置，加 1 后即指向后继字符串。

```
    if (__access ("/etc/suid-debug", F_OK) != 0)
      {
        unsetenv ("MALLOC_CHECK_");
        GLRO(dl_debug_mask) = 0;
      }

    if (mode != normal)
      _exit (5);
  }
```

10.11.1.2 C 库 AT_SECURE 机制演示

示例程序 read-auxv.c（可以从 http://www.wienand.org/junkcode/linux/read-auxv.c 下载）访问某一个进程的/proc/pid/auxv 文件（负责导出该进程 mm->saved_auxv 数组的内容到用户态）。

read-auxv 程序在循环中读取/proc/pid/auxv 文件，每次读取一个 AT 向量到 Elf32_auxv_t 数据结构 aux 中，然后打印名称和数值：

```
    while (read(fd, &aux, sizeof(Elf32_auxv_t))) {
            printf("%s : %lx\n", aux_to_str(aux.a_type), aux.a_un.a_val);
    }
```

在文件中增加一行语句：打印 getenv（“LD_LIBRARY_PATH”）的结果，从而验证 secure 模式是否被真正执行了：

```
printf("LD_LIBRARY_PATH: %s\n", getenv("LD_LIBRARY_PATH"));
```

1，root 将该可执行程序通过 chmod 命令设置为 setuid/setgid 的，则普通用户 usera 执行它时，结果如下：

```
[root/sysadm_r/s0@~]# chmod 6755 read-auxv
[root/sysadm_r/s0@~]# mv read-auxv /home/usera/
[root/sysadm_r/s0@~]# su - usera
[usera@QtCao ~]$ pwd
/home/usera
[usera@QtCao ~]$ ls -l read-auxv
-rwsr-sr-x 1 root root 7119 Aug  5 08:24 read-auxv
[usera@QtCao ~]$ printenv LD_LIBRARY_PATH
[usera@QtCao ~]$ export LD_LIBRARY_PATH=/home/usera
[usera@QtCao ~]$ printenv LD_LIBRARY_PATH
/home/usera
[usera@QtCao ~]$ ./read-auxv se1f
AT_SYSINFO : ffffe414
AT_SYSINFO_EHDR : ffffe000
AT_HWCAP : 780abfd
AT_PAGESZ : 1000
AT_CLKTCK : 64
AT_PHDR : 8048034
AT_PHENT : 20
AT_PHNUM : 7
AT_BASE : b78ac000
AT_FLAGS : 0
AT_ENTRY : 80483d0
AT_UID : 1f4
AT_EUID : 0
AT_GID : 1f4
AT_EGID : 0
AT_SECURE : 1
UNKNOWN : bfb2ffab
UNKNOWN : bfb31ff0
AT_PLATFORM : bfb2ffbb
AT_NULL : 0
LD_LIBRARY_PATH: (nu11)
[usera@QtCao ~]$
```

由此可见，由于 euid/egid 不再等于 uid/gid，所以 cap_bprm_secureexec 函数将返回 1，即启动 secure 模式。这一点从 AT_SECURE 向量的数值可以验证，而且从结果来看 LD_LIBRARY_PATH 环境变量的确被清除了（在父进程中不为 NULL）。

2，如果取消 setuid/setgid 标志位，普通用户 usera 再次执行，结果如下：

```
[usera@QtCao ~]$ exit
logout
[root/sysadm_r/s0@~]# chmod 755 /home/usera/read-auxv
[root/sysadm_r/s0@~]# su - usera
[usera@QtCao ~]$ ls -l read-auxv
-rwxr-xr-x 1 root root 7119 Aug  5 08:24 read-auxv
[usera@QtCao ~]$
[usera@QtCao ~]$ printenv LD_LIBRARY_PATH
```

```
[usera@QtCao ~]$ export LD_LIBRARY_PATH=/home/usera
[usera@QtCao ~]$ printenv LD_LIBRARY_PATH
/home/usera
[usera@QtCao ~]$ ./read-auxv se1f
AT_SYSINFO : ffffe414
AT_SYSINFO_EHDR : ffffe000
AT_HWCAP : 780abfd
AT_PAGESZ : 1000
AT_CLKTCK : 64
AT_PHDR : 8048034
AT_PHENT : 20
AT_PHNUM : 7
AT_BASE : b77b0000
AT_FLAGS : 0
AT_ENTRY : 80483d0
AT_UID : 1f4
AT_EUID : 1f4
AT_GID : 1f4
AT_EGID : 1f4
AT_SECURE : 0
UNKNOWN : bf950f0b
UNKNOWN : bf951ff0
AT_PLATFORM : bf950f1b
AT_NULL : 0
LD_LIBRARY_PATH: /home/usera
[usera@QtCao ~]$
```

由此可见，此时无须启动 secure 模式，AT_SECURE 向量的值为 0，在子进程执行 read-auxv 程序期间，保留了父进程（she11）的 LD_LIBRARY_PATH 环境变量。

10.11.1.3 SELinux 对 AT_SECURE 机制的扩展（Revisited）

当使能 SELinux 时，securiy_bprm_secureexec 函数通过 SELinux 提供的回调函数 selinux_bprm_secureexec 实现：

```
int security_bprm_secureexec(struct linux_binprm *bprm)
{
        return security_ops->bprm_secureexec(bprm);
}

static int selinux_bprm_secureexec(struct linux_binprm *bprm)
{
        const struct task_security_struct *tsec = current_security();
        u32 sid, osid;
        int atsecure = 0;

        sid = tsec->sid;
        osid = tsec->osid;

        if (osid != sid) {
                /* Enable secure mode for SIDs transitions unless
                   the noatsecure permission is granted between
                   the two SIDs, i.e. ahp returns 0. */
                atsecure = avc_has_perm(osid, sid, SECCLASS_PROCESS, PROCESS__NOATSECURE, NULL);
        }

        return (atsecure || cap_bprm_secureexec(bprm));
}
```

由此可见，该函数就是在上述 cap_bprm_secureexec 函数的基础上，如果在 exec 期间发生 sid 的改变时（注意，不局限于 domain transition，还可能繁盛 role_change 或 range_change），则查询 Security Server 检查新 sid 是否对原来的 sid 具有 process 类的 noatsecure 权限（表示关闭 secure 模式）。如果没有，则返回值 atsecure 为 1，否则为 0。

也就是说，AT_SECURE 向量的值不仅仅由传统方法决定，也由 sid 改变后新 sid 对旧 sid 是否具有 noatsecure 权限决定。如果 cap_bprm_secureexec 函数返回 0，但是 avc_has_perm 返回 1（表示失败），则 AT_SECURE 向量仍然被设置为 1（即需要启动 secure 模式）。

可见，SELinux 进一步强化了需要执行 secure 模式的时机：如果 sid 发生改变，且新旧 sid 之间没有 noatsecure 关系，则一定会启动 secure 模式清除子进程的部分环境变量。

反之，如果希望 SELinux 不影响在 sid 改变后子进程所继承的父进程环境变量，则应该对子进程的 domain 添加对父进程 domain 的 noatsecure 能力，比如：

```
allow xdm_t xserver_t:process { noatsecure siginh rlimitinh signal sigkill };
```

从而使得 X 窗口管理器能够保留从 X Server 继承的环境变量，包括 UNSECURE_ENVVARS 那部分。

另外，由于上面 avc_has_perm 的结果仅仅用于设置 AT_SECURE 变量的值，而不决定当前执行流是否能顺利执行，而且这种检查在 execve 系统调用时总是会发生，因此默认情况下使用 dontaudit 规则屏蔽相关的错误信息。比如在 domain_transition_pattern 宏中指定：

```
define(`domain_transition_pattern',`
        allow $1 $2:file { getattr open read execute };
        allow $1 $3:process transition;
        dontaudit $1 $3:process { noatsecure siginh rlimitinh };
')
```

否则 audit log 中就会充斥大量和 noatsecure 相关的无用错误消息了。

在之前的例子中，由于 read-auxv 程序的标签为 user_home_t，且当前进程（sysadm_t）对它具有 execute_no_trans 能力，所以在执行该程序期间不发生 sid 的切换，AT_SECURE 标志是否被设置只由 cap_bprm_secureexec 函数决定：如果不是 setuid/setgid 类程序的，则该标志被清除，即保留 UNSECURE_ENVVARS 环境变量：

```
[root/sysadm_r/s0@~]# echo $LD_LIBRARY_PATH
sss
[root/sysadm_r/s0@~]# ls -Z read-auxv
-rwxr-xr-x  root root root:object_r:user_home_t:s0      read-auxv                # 非 setuid/setgid 程序
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# seclow "sesearch -SCA -s sysadm_t -t user_home_t -c file -p execute_no_trans"
Password:
Found 1 semantic av rules:
   allow sysadm_t user_home_t : file { ioctl read write create getattr setattr lock relabelfrom relabelto
append unlink link rename execute execute_no_trans entrypoint open } ;

[root/sysadm_r/s0@~]# ./read-auxv self | grep -e AT_SECURE -e LD_LIBRARY _PATH
AT_SECURE : 0
LD_LIBRARY_PATH: sss
[root/sysadm_r/s0@~]#
```

但是，一旦在执行 read-auxv 程序期间引发 sid 的改变（比如通过触发 domain transition），则可以看

到 AT_SECURE 标志被设置，LD_LIBRARY_PATH 环境变量被清除

```
[root/sysadm_r/s0@~]# chcon -t vlock_exec_t read-auxv
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ls -Z read-auxv
-rwxr-xr-x  root root root:object_r:vlock_exec_t:s0    read-auxv
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# ./read-auxv se1f | grep -e AT_SECURE -e LD_LIBRARY _PATH
AT_SECURE : 1
LD_LIBRARY_PATH: (nu11)
[root/sysadm_r/s0@~]#
```

比如把 read-auxv 可执行文件的标签改为 vlock_exec_t，则在执行该程序期间将切换到 vlock_t，而 vlock_t 对 sysadm_t 没有 process 类的 noatsecure 能力，所以 AT_SECURE 标志被设置，包括 LD_LIBRARY_PATH 在内的 UNSECURE_ENVVARS 环境变量集合被清除。

11．用户态应用程序对 SELinux 的支持

许多用户态程序都可以在 spec 或 Makefile 中选择是否支持 SELinux。如果是，则和 libselinux 或 libsepol 包一起编译。比如 udev 包，如果在编译时选择支持 SELinux，则在运行时就能调用 libselinux 相关 API 判断 SELinux 是否被使能，如果是则在创建设备节点时正确设置其 SC。用户态所有 SELinux-aware 应用程序都通过 libselinux 提供的 API 来访问 selinuxfs，从而和内核 Security Server 交互。

11.1 libselinux 相关文件分析

libselinux 库封装了对/selinux/*和/etc/selinux/$SELINUXTYPE/contexts/*文件的访问方法。要想了解 selinuxfs 驱动导出的/selinux/*接口该如何使用，可以阅读 libselinux 源代码。

11.1.1 selinux_config.c 文件

在 src/selinux_config.c 文件中定义 libselinux 所需要访问的 24 个/etc/selinux/$SELINUXTYPE/*文件的编号，在 file_path_suffixes.h 中建立文件编号和**相对路径**的联系，所有这些文件的相对路径名字符串被保存在 file_path_suffixes_data.str 中，每个文件的路径名的起始偏移则被保存在 file_path_suffixes_idx[24]数组中。

在 init_selinux_config 函数中读取/etc/selinux/config 配置文件，得到 SELINUX 和 SELINUXTYPE 变量的值，并设置 file_paths[24]字符串指针数组指向各文件的**绝对路径**字符串，从而可以使用 get_path(idx)函数直接获得相应文件的绝对路径－直接返回 file_paths[idx]即可。

11.1.2 getfilecon.c 文件

SELinux 利用文件的扩展属性保存安全上下文信息。POSIX 定义了四种文件扩展属性的相关操作（读取 xattr，设置 xattr，得到所有 xattr 列表，删除 xattr），就每种操作又定义了三种型构的 API：指定路径，追随符号连接；指定路径，不追随符号连接；指定文件描述符。libselinux 使用前两种操作，分别提供了这两个操作相应的三种 API，它们通过调用如下 POSIX 函数来实现：

```
ssize_t getxattr(const char *path, const char *key, void *value, size_t size);
ssize_t lgetxattr(const char *path, const char *key, void *value, size_t size);
ssize_t fgetxattr(int fd, const char *key, void *value, size_t size);

ssize_t setxattr(const char *path, const char *key, void *value, size_t size, int flags);
ssize_t lsetxattr(const char *path, const char *key, void *value, size_t size, int flags);
ssize_t fsetxattr(int fd, const char *key, void *value, size_t size, int flags);
```

其中扩展属性的键值 key 总是“security.selinux”，size 为键值字符串长度＋1。对于 setxattr 类型的操作 flags 总是 0，表示如果没有则创建，如果存在则修改。

11.1.3 procattr.c 文件

该文件定义的一系列函数利用/proc/pid/attr/*接口文件读取、设置当前进程（或指定 pid 的进程）相关的安全属性。/proc/pid/attr/*各个文件和 task_security_struct 数据结构中的各个 sid 域相对应（参见 10.2.1.1 小节）。核心函数 getprocattrcon_raw 和 setprocattrcon_raw 的型构如下：

```
static int getprocattrcon_raw(security_context_t * context, pid_t pid, const char *attr)
```

如果参数 pid>0，则访问该进程的相应 attr 文件：

```
rc = asprintf(&path, "/proc/%d/attr/%s", pid, attr);
```

否则用__NR_gettid 系统调用得到当前进程的 tid，访问当前进程的属性文件：

```
rc = asprintf(&path, "/proc/self/task/%d/attr/%s", tid, attr);
```

然后直接通过 read/write 函数读取、设置相应进程的安全上下文。

11.1.4 compute_relabel.c 文件（访问/selinux/relabel）

所有在用户登录、改变角色时需要 relabel 当前终端设备的程序，比如 newrole 或 pam_selinux.so，都要调用 security_compute_relabel 函数，根据当前用户 domain，终端设备的 type 和 class，查询/selinux/relabel 接口，得到终端设备的新 type，并最终通过某种形式的 setxattr 函数修改终端设备的安全上下文。

这样做的好处是只需要在 policy.X 中以 type_change 规则的形式保存不同用户在某类设备上登录时该设备的新 type（属于策略范畴），而不需要把新 type 硬编码到相应的用户态应用程序中（属于机制范畴）。后者通过 selinuxfs 向 policy.X 查询即可。即 security_compute_relabel 函数实现了向 policy.X 查询相应策略的机制。它的型构如下：

```
int security_compute_relabel(security_context_t scon, security_context_t tcon,
                             security_class_t tclass, security_context_t * newcon)
```

其中各个参数分别和 type_change 规则中的各个域相对应，比如：

```
type_change user_t tty_device_t:chr_file user_tty_device_t;
```

即，当用户 user_t 在字符类型 tty_device_t 设备（比如/dev/tty*）上登录时，把相应设备的标签 relabel 为 user_tty_device_t。

该函数首先通过 selinux_trans_to_raw 函数向 setransd 后台进程发出请求，翻译 scon/tcon 为裸字符串，然后通过 security_compute_relabel_raw 函数，打开/selinux/relabel 文件并写入“%s%s%hu”格式的 scon，tcon，tclass 字符串，最后读取该文件，得到相应 type_change 规则的最后一个域的值。

和各种用户 domain 相关的 type_change 规则以如下方式定义：

```
userdom_admin_user_template >                                          (sysadm)
userdom_unpriv_user_template > userdom_restricted_user_template >      (user/staff/secadm/auditadm)
        userdom_login_user_template > userdom_base_user_template >
                term_user_tty($1_t, user_tty_device_t) >
                        type_change $1 tty_device_t:chr_file $2;
```

另外，所有使用 security_compute_relabel 函数的用户态应用程序的相应 domain（比如 local_login_t 或 ssh_t）都需要调用 selinux_compute_relabel_context 接口以便能够访问/selinux/relabel 文件。

最后，在/selinux/relabel 文件的写方法 sel_write_relabel 函数中检查当前进程 domain 是否具有对 security_t@security 的 compute_relabel 能力。

11.2 newrole 源代码分析

11.2.0 newrole 命令的使用模型

使用 newrole 命令可以“改变”当前 shell 进程的安全上下文中除了 SELinux user 之外的其他部分，从而使得当前 SELinux user 可以扮演其所许可的其他角色或者安全级别。

其实 newrole 命令并没有真正改变原有 shell 进程的安全上下文，而是创建了一个子 shell 进程，它的安全上下文由父进程的安全上下文和命令行参数共同决定。

注意，运行 newrole 命令的父进程（为 shell 的子进程，因为 newrole 显然是 shell 的外部命令）和 newrole 命令所创建的子进程，都使用相同的 terminal（ttyn = ttyname(STDIN_FILENO)）（可能是 tty，或者 pty，或者 console）。在创建子进程之前以及等待子进程运行结束期间，父进程就把当前 terminal 的标签 relabel 为和子进程 domain 相对应的新标签，而 terminal 的安全级别则和子进程的保持一致。当子进程结束后父进程又会恢复 terminal 为原来的标签和安全级别。

当 root 用户通过 ssh 登录系统时默认扮演 staff_r 角色，可以看到它所使用的 pty 设备的标签被 relabel 为 user_devpts_t。并且当 staff 用户切换到其他的安全级别时，相应 pty 设备的安全级别也随之改变。比如：

```
[root/staff_r/s0@~]# id -Z
root:staff_r:staff_t:s0-s15:c0.c1023
[root/staff_r/s0@~]#
[root/staff_r/s0@~]# tty
/dev/pts/0
[root/staff_r/s0@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_devpts_t:s0   /dev/pts/0
[root/staff_r/s0@~]#
[root/staff_r/s0@~]# newrole -l s15
Password:
[root/staff_r/s15@~]#
[root/staff_r/s15@~]# ls -Z `tty`
crw--w----  root tty root:object_r:user_devpts_t:s15  /dev/pts/0
[root/staff_r/s15@~]#

[root/sysadm_r/s0@~]# newrole -l s15:c0.c1023 -- -c "ps axjZ"
LABEL                              PPID   PID  PGID   SID TTY      TPGID STAT   UID   TIME COMMAND
......
system_u:system_r:sshd_t:s0-s15:c0.c1023 1 1288 1288 1288 ?         -1 Ss       0   0:00 /usr/sbin/sshd
system_u:system_r:sshd_t:s0-s15:c0.c1023 1288 2259 2259 2259 ?      -1 Ss       0   0:01 sshd: root@pts/0
root:staff_r:staff_t:s0-s15:c0.c1023 2259 2263 2263 2263 pts/0    2311 Ss       0   0:00 -bash
root:staff_r:newrole_t:s0-s15:c0.c1023 2263 2307 2307 2263 pts/0  2311 S        0   0:00 newrole -l s15
root:staff_r:staff_t:s15-s15:c0.c1023 2307 2311 2311 2263 pts/0   2311 S+       0   0:00 -/bin/bash
......
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# newrole -l s15:c0.c1023 -- -c "ls -Z1 /proc/2307/fd"
Password:
total 0
lrwx------ 1 root:staff_r:newrole_t:s0-s15:c0.c1023 root root 64 Jan 30 07:20 0 -> /dev/pts/0
lrwx------ 1 root:staff_r:newrole_t:s0-s15:c0.c1023 root root 64 Jan 30 07:44 1 -> /dev/pts/0
lrwx------ 1 root:staff_r:newrole_t:s0-s15:c0.c1023 root root 64 Jan 30 07:22 2 -> /dev/pts/0
lrwx------ 1 root:staff_r:newrole_t:s0-s15:c0.c1023 root root 64 Jan 30 07:44 3 -> /dev/pts/0
[root/sysadm_r/s0@~]#
[root/sysadm_r/s0@~]# newrole -l s15:c0.c1023 -- -c "ls -Z1 /proc/2311/fd"
Password:
total 0
```

```
lr-x------ 1 root:staff_r:staff_t:s15-s15:c0.c1023 root root 64 Jan 30 07:43 0 -> /dev/pts/0
lrwx------ 1 root:staff_r:staff_t:s15-s15:c0.c1023 root root 64 Jan 30 07:43 1 -> /dev/pts/0
lrwx------ 1 root:staff_r:staff_t:s15-s15:c0.c1023 root root 64 Jan 30 07:30 2 -> /dev/pts/0
lrwx------ 1 root:staff_r:staff_t:s15-s15:c0.c1023 root root 64 Jan 30 07:43 255 -> /dev/pts/0
[root/sysadm_r/s0@~]#
```

（注意，由于通过 ssh 登录的 root 用户使用 newrole 切换到了 s15 安全级别，所以通过 console 登录的 root 用户必须切换到 s15:c0.c1023 安全级别，才能观察到处于各种安全级别的所有进程。）

由此可见：
1，用户通过 ssh 登录时，系统创建 sshd 后台进程（1288）的子进程（2259），它使用 pts/0 设备；
2，它 fork 子进程，执行该用户的登录 shell（2263）。注意 root 用于在 sshd_t 登录设备上的默认角色为 staff_r；
3，用户执行 newrole 命令，由于是 shell 的外部命令，所以在 shell 进程的子进程（2307）中执行；
4，shell 的子进程执行 newrole 命令，它又创建新的子进程（2311）在指定的安全上下文和安全级别中再次执行用户的登录 shell。注意 2311 进程的 current level 为 s15，而其父进程 2307 仍然在 s0 中。

注意：
1，始终只有一个“/dev/pts/0”设备，它在 2311 执行期间的安全级别为 s15；
2，幸好此时执行 newrole 命令的父进程（2307）处于挂起状态，否则它继续访问/dev/pts/0 很有可能被 MLS constraints 所拒绝；
3，打开文件描述符（对应 file_security_struct 数据结构）的 sid 由打开进程决定。执行 newrole 命令的父进程（2307）对“/dev/pts/0”设备的打开文件描述符的 type 为 newrole_t，安全级别为 s0；而其子进程对“/dev/pts/0”设备的打开文件描述符的 type 为 staff_t，安全级别为 s15。

11.2.1 main 函数

main 函数所执行的步骤如下：
1，设置基本环境，比如检查 SELinux 是否被使能，设置信号处理函数等。

2，通过 getprevcon(&old_context)得到当前 shell 的安全上下文，由 ttyn=ttyname(STDIN_FILENO)得到当前 terminal 设备（和命令行直接运行 tty 命令结果一致），并调用 parse_command_line_arguments 函数结合命令行参数得到子 shell 的安全上下文。

3，通过 authenticate_via_pam 或者 authenticate_via_shadow_passwd 函数，借助 PAM 或者/etc/shadow 文件认证用户的身份。

4，调用 relabel_tty 函数，首先 re-open 当前 terminal（以便重新以特定的方式打开），然后 relabel 它为和子进程 domain 相对应的 type（从 policy.X 中查询相应的 type_change 规则）。

注意，如果使用 newrole 改变了安全级别，则当前 terminal 设备的安全级别也相应地改变。而运行 newrole 程序的父进程仍处于原来的安全级别中。

父子进程共享同一个 terminal，它的标签和安全级别由子进程决定。父进程仍处在原先的安全级别上，但是它处于挂起状态，因此不会使用更高安全级别的 terminal（将被拒绝）。

5，fork 子进程。父进程在 do-while 循环中 wait 子进程结束，然后使用 relabel_tty_device 函数恢复当前 terminal 设备原来的 type 以及安全级别。

如果使用 PAM，则调用 pam_end_session/pam_end 函数。

子进程关闭继承下来的0/1/2打开文件描述符，然后重新open当前terminal设备三次，从而可以重新设置打开方式，并创建新的file数据结构及相应的file_security_struct数据结构。

6，子进程（注意此时还在父进程的newrole地址空间中）从/etc/shadow文件中得到用户的登录shell，并通过setexeccon(new_context)函数将新的domain写入/proc/pid/attr/exec文件（在执行exec时生效），然后调用pam_open_session函数（作用？），通过restore_environment函数设置子进程的环境变量，最后通过execv(pw.pw_shell, argv + optind - 1)系统调用装载相应shell的地址空间并传递newrole命令行的余下参数（用户在使用newrole时可直接指定子进程shell要执行的命令）。

11.2.2 parse_command_line_arguments函数

```
static int parse_command_line_arguments(int argc, char **argv, char *ttyn,
                                        security_context_t old_context,
                                        security_context_t * new_context,
                                        int *preserve_environment)
```

该函数检查是否允许在当前tty设备上执行newrole命令，分析newrole的命令行参数，组装新的安全上下文并检查其有效性。

当前shell的安全上下文由参数old_context传递，子shell的安全上下文则由输出参数new_context返回。如果用户在命令行指定“-p”选项，则输出参数preserve_environment为1，否则为0。为1时子shell将保留父shell当前的环境变量，否则子shell的环境变量根据用户/etc/shadow的来设置，并且PATH被设置为缺省的最小值“/usr/bin:/bin”。

首先用getopt函数获得命令行所有选项及其参数。如果指定“-r/-t/-l”选项，则相应optarg的值分别保存在局部变量role_s，type_s，level_s中。对于level_s还要调用fgetfilecon函数获得当前STDIN_FILENO的安全上下文，并通过selinux_check_securetty_context函数检查它是否被记录到securetty_types文件中（检查是否为能够改变安全级别的设备）。

如果role_s非空而type_s为空，则通过get_default_type函数得到和此role相应的默认的type。如果失败，则调用send_audit_message函数写入audit日志。

父shell当前安全上下文由参数old_context传递进来（通过getprevcon函数得到），通过context_new函数分配一个context_private_t数据结构，就当前old_context中冒号分隔的各个成员，复制，并用context_private_t中的字符串指针数组来索引。

创建context_private_t数据结构的目的就是拆分安全上下文为各个域，以便于修改。利用context_role_set，context_type_set，context_range_set函数，用局部变量role_s，type_s，level_s的值修改指针数组索引的相应字符串。最后分配并拼接新的安全上下文字符串（其空间即由context_private_t的current_str指针所指向）

然后通过security_check_context函数，通过向/selinux/context写入新sc字符串的方式来检查其合法性。并通过strdup函数复制到输出参数*new_context中。（调用者传递指针的地址，这里将该指针指向新sc字符串的地址，该字符串的地址也由context_private_t.current_str指向，由context_new分配）

11.2.3 relabel_tty函数

```
static int relabel_tty(const char *ttyn, security_context_t new_context,
                       security_context_t * tty_context,
                       security_context_t * new_tty_context)
```

在用户登录或者通过 newrole 命令切换 SC 时要重新给当前 terminal 设备打标签。ttyn 为当前 terminal 设备的文件名（和使用 tty 命令得到的结果相同），new_context 为子进程的 type（即用户通过 newrole 命令所期望的新 type），该函数通过 libselinux 函数从 policy.X 中得到相应 type_change 规则中当前 tty 设备的新 type，设置它并通过 new_tty_context 参数返回。

用户执行 newrole 命令时，当前父 shell 进程的 stdin/stdout/stderr 打开文件描述符都指向当前 terminal 设备。这里首先 re-open 该 tty 设备得到新的 fd，从而便于重新设置。执行操作如下：

fcntl(fd, F_SETFL, fcntl(fd, F_GETFL, 0) & ~ O_NONBLOCK) （为什么要这样？）

然后通过 fgetfilecon(fd, &tty_con)函数得到该 terminal 设备现在的 type，再调用如下函数：

security_compute_relabel(new_context, tty_con, SECCALSS_CHR_FILE, &new_tty_con)

根据子 shell 进程的 type、该 terminal 设备当前的 type，得到其新的 type。

最后通过 fsetfilecon(fd, new_tty_con)函数（最终调用 fsetxattr 函数）设置 terminal 设备为新的标签，并通过输出参数返回 tty_con，tty_new_con 的值。

比较：用户登陆时 terminal 的 relabel 是由相应登录程序调用 pam_selinux.so 实现的。而在用户扮演不同的角色时 newrole 命令靠自己“再次”relabel 当前的 terminal 设备。

11.3 PAM 模块分析

11.3.1 pam_selinux.so 作用分析（TODO）

login/remote/sshd 登录程序通常使用 pam_selinux.so 模块设置用户登录 shell 的安全上下文，并 relabel 相应的 terminal 设备为和用户的角色相对应的类型。

比如在/etc/pam.d/login 中可以以如下方式使用 pam_selinux.so：

```
[root/sysadm_r/s0@~]# cd /etc/pam.d/
[root/sysadm_r/s0@pam.d]# grep pam_selinux * -1
login
remote
sshd
[root/sysadm_r/s0@pam.d]#  vim login
...
# pam_selinux.so close should be the first session rule （从而使得其后 pam 模块在 login 进程的 SC 中运行）
session    required     pam_selinux.so close
session    include      system-auth
session    required     pam_loginuid.so
session    optional     pam_console.so
# pam_selinux.so open should only be followed by sessions to be executed in the user context （从而使得其后的 pam 模块在用户自己的 SC 中运行）
session    required     pam_selinux.so open select_context
session    required     pam_namespace.so
```

当用户在本地登录时，login 程序创建新的会话，创建子进程执行该用户在/etc/shadow 文件中所指定的登录 shell。pam_selinux.so 通过如下函数调用链来 relabel 当前的 terminal 设备：

pam_selinux.c:
pam_sm_open_session > security_label_tty > security_compute_relabel

另外，在 login 程序的 PAM 配置文件的 session 部分中，“pam_selinux.so close”后的 PAM 模块在 login 进程的 SC 中运行；而“pam_selinux.so open”后的 PAM 模块则在用户的 SC 中运行。

TODO：分析 pam_selinux.so 的源代码，看如何做到上面这一点的。

11.3.2 pam_loginuid.so 作用分析

一方面，除了 sudo/su 之外的其他“Entrypoint Application”比如 sshd/login/crond/atd/xdm/vsftpd 的 PAM 配置文件都必须使用 pam_loginuid.so 来设置用户登录进程的 loginuid 信息，用于审计（audit）。

另一方面，这些应用程序相应的 pp 必须调用 logging_set_loginuid 接口赋予相应 domain capability 类的 audit_control 能力

```
# Set login uid
interface(`logging_set_loginuid',`
        allow $1 self:capability audit_control;
        allow $1 self:netlink_audit_socket { r_netlink_socket_perms nlmsg_relay };
')
```

曾经忘记对 crond_t 调用该接口，得到相应 audit 错误消息如下：

```
time->Fri Jan 28 05:30:02 2011
type=USER_START msg=audit(1296192602.112:2919): user pid=2652 uid=0 auid=4294967295 ses=4294967295
subj=system_u:system_r:crond_t:s0-s15:c0.c255 msg='op=PAM:session_open acct="root" exe="/usr/sbin/crond" (hostname=?,
addr=?, terminal=cron res=failed)'
----
time->Fri Jan 28 05:30:02 2011
type=USER_END msg=audit(1296192602.124:2920): user pid=2652 uid=0 auid=4294967295 ses=4294967295
subj=system_u:system_r:crond_t:s0-s15:c0.c255 msg='op=PAM:session_close acct="root" exe="/usr/sbin/crond" (hostname=?,
addr=?, terminal=cron res=failed)'
----
time->Fri Jan 28 05:30:02 2011
type=SYSCALL msg=audit(1296192602.092:2918): arch=40000003 syscall=4 success=no exit=-1 a0=3 a1=bffdf910 a2=1 a3=0
items=0 ppid=2208 pid=2652 auid=4294967295 uid=0 gid=0 euid=0 suid=0 fsuid=0 egid=0 sgid=0 fsgid=0 tty=(none)
ses=4294967295 comm="crond" exe="/usr/sbin/crond" subj=system_u:system_r:crond_t:s0-s15:c0.c255 key=(null)
type=AVC msg=audit(1296192602.092:2918): avc:  denied  { audit_control } for  pid=2652 comm="crond" capability=30
scontext=system_u:system_r:crond_t:s0-s15:c0.c255 tcontext=system_u:system_r:crond_t:s0-s15:c0.c255 tclass=capability
```

crond 使用 pam_loginuid.so，后者在调用 session_open/close 时失败。从**错误信息的时间戳**上可以看到 AVC Denied Message 和失败操作相关！

“audit_control”能力的作用有两个：“Allows the process to change auditing rules. Set login UID.”由此可见的确需要给 crond_t 赋予它。

11.3.3 pam_namespace.so 作用分析

11.3.3.1 多态（polyinstantiation）的作用

pam_namespace.so 模块用于实现多态，可以被需要建立用户登录会话（login session）的应用程序（比如 sshd/login）使用，在建立用户登录会话时创建独立的域名空间（namespace）。注意，只有在同一个 namespace 中，看到的文件系统视图才是相同的。多态能够使得不同用户之间，以及同一个用户扮演不同的 SC 时，使用不同 namespace，从而访问不同的 tmp 和 HOME 目录。

/etc/security/namespace.conf 文件指名需要被多态的目录的绝对路径，instance parent 目录的绝对路径，多态的方法，以及哪些用户不参与多态。比如：

```
[root/sysadm_r/s0@QtCao ~]# cat /etc/security/namespace.conf
/tmp     /tmp-inst/            level      root,adm
/var/tmp /var/tmp/tmp-inst/    level      root,adm
$HOME    $HOME/$USER.inst/      level
[root/sysadm_r/s0@QtCao ~]#
```

以/tmp 为例，相关目录的术语以及所属的 SELinux 属性如下所示：

```
/tmp                                                  poly dir                    polydir
/tmp-inst                                             poly instance parent dir    polyparent
/tmp-inst/system_u:object_r:tmp_t:s0-s15:c0.c255_staff  poly instance member dir    polymember
```

pam_namespace.so 首先将用户登录会话的 namespace 从其 parent namespace 中分离出来（disassociate），在 instance parent dir 下创建 instance member dir，其命名方式由多态方式决定，然后将它 bind mount 到 poly dir，最后调用 namespace.init 脚本初始化。

所以，当不同用户登录时，或者同一个用户以不同 SC 登录时（比如 role 或 level 不同），则处在不同的 namespace 中，从而看到不同的 bind mount 的结果。

注意，如果和用户当前 level 相对应的 member dir 已经存在，则 pam_namespace.so 传递给 namespace.init 的$3 参数为 0，后者直接退出。

```
$ cat etc/security/namespace.init
#!/bin/sh -p
# It receives polydir path as $1, the instance path as $2,
# a flag whether the instance dir was newly created (0 - no, 1 - yes) in $3,
# and user name in $4.
#
# The following section will copy the contents of /etc/skel if this is a
# newly created home directory.
if [ "$3" = 1 ]; then
        # This line will fix the labeling on all newly created directories
        [ -x /sbin/restorecon ] && /sbin/restorecon "$1"
        user="$4"
        passwd=$(getent passwd "$user")
        homedir=$(echo "$passwd" | cut -f6 -d":")
        if [ "$1" = "$homedir" ]; then
                gid=$(echo "$passwd" | cut -f4 -d":")
                cp -rT /etc/skel "$homedir"
                chown -R "$user":"$gid" "$homedir"
                mask=$(awk '/^UMASK/{gsub("#.*$", "", $2); print $2; exit}' /etc/login.defs)
                mode=$(printf "%o" $((0777 & ~$mask)))
                chmod ${mode:-700} "$homedir"
```

```
                [ -x /sbin/restorecon ] && /sbin/restorecon -R "$homedir"
        fi
fi

exit 0
```

（TODO：详细分析 namespace.init 脚本的作用）

“level”方式使用用户的 MLS Level 及其用户名作为 member dir 名称的后缀。比如 root 用户以默认的安全级别 s0-s15:c0.c1023 从 console 登录系统后，login 为其在/root/root.inst/下创建的 member dir 为：

```
（root 用户在 console 上 login，默认安全级别为 s0-s15:c0.c1023）
[root/sysadm_r/s0@QtCao ~]# id -Z
root:sysadm_r:sysadm_t:s0-s15:c0.c1023
[root/sysadm_r/s0@QtCao ~]# ls
[root/sysadm_r/s0@QtCao ~]# touch s0-s15:c0.c255
[root/sysadm_r/s0@QtCao ~]# ls
s0-s15:c0.c255
[root/sysadm_r/s0@QtCao ~]#

（root 用户通过 ssh 登录，注意 sshd 并不使用 pam_namespace.so）
[root/secadm_r/s0@QtCao ~]# ls /root/root.inst/ -Zd
d---------  root root system_u:object_r:user_home_dir_t:s0 /root/root.inst/
[root/secadm_r/s0@QtCao ~]# ls /root/root.inst/ -Z
drwx------  root 0 root:object_r:user_home_dir_t:s0-s15:c0.c1023
root:object_r:user_home_dir_t:s0-s15:c0.c1023_root
[root/secadm_r/s0@QtCao ~]#
```

root 用户从 console 上退出后，重新以 s15:c0.c255 重新登录，则看不到之前创建的名为“s0-s15:c0.c255”的文件，从 ssh 登录的 root 可以看到/root/root.inst/下针对不同的 level 分别创建了不同的 member dir：

```
[root/sysadm_r/s15:c0.c255@QtCao ~]# id -Z
root:sysadm_r:sysadm_t:s15:c0.c255
[root/sysadm_r/s15:c0.c255@QtCao ~]# ls
[root/sysadm_r/s15:c0.c255@QtCao ~]#

[root/secadm_r/s0@QtCao ~]# ls /root/root.inst/ -Z
drwx------  root 0 root:object_r:user_home_dir_t:s0-s15:c0.c1023
root:object_r:user_home_dir_t:s0-s15:c0.c1023_root                    (member dir 1)
drwx------  root 0 root:object_r:user_home_dir_t:s0-s15:c0.c1023
root:object_r:user_home_dir_t:s15:c0.c255_root                        (member dir 2)
[root/secadm_r/s0@QtCao ~]#

[root/secadm_r/s0@QtCao ~]# ls /root/root.inst/root:object_r:user_home_dir_t:s0-s15:c0.c1023_root
s0-s15:c0.c255
[root/secadm_r/s0@QtCao ~]# ls /root/root.inst/root:object_r:user_home_dir_t:s15:c0.c255_root
[root/secadm_r/s0@QtCao ~]#
```

注意这两个 member dir 的标签都是“root:object_r:user_home_dir_t:s0-s15:c0.c1023”，即和 poly dir 的一致；member dir 名称的前半部分“user:role:type”也和 poly dir 的保持一致，而后半部分为登录用户的 level 及用户名称。

其他支持的多态方式为“context”和“tmpfs”，分别表示使用用户名及其完整的 SC 作为 memberdir 的名称，将 tmpfs 文件系统挂载到 polydir 上。详见 man namespace.conf。

11.3.3.2 LSPP 对多态的配置

在配置 LSPP 时，多态的配置过程如下：

```
*** Update polyinstantiation (pam_namespace) configuration? (y/n) [y]: y
Creating base dirs: /tmp-inst/
/var/tmp-inst/
/home/home.inst/
+mkdir -p -m 0 /tmp-inst/ /var/tmp-inst/ /home/home.inst/
+semanage fcontext -a -t tmp_t /tmp-inst
+semanage fcontext -a -t tmp_t /var/tmp-inst
+semanage fcontext -a -t tmp_t /home/home\.inst
+restorecon /tmp-inst/ /var/tmp-inst/ /home/home.inst/
+restorecon /etc/security/namespace.init
Archiving /etc/security/namespace.conf
+mv /etc/security/namespace.conf /etc/security/namespace.conf-20110303-0030
+cp -p /usr/share/capp-lspp/namespace.conf /etc/security/namespace.conf
+chmod 644 /etc/security/namespace.conf
+chown root.root /etc/security/namespace.conf
+restorecon /etc/security/namespace.conf
```

根据 namespace.conf 中的说明，instance parent dir 必须存在且权限为 000，因此首先创建这些目录并同时指定其权限。为了便于后台进程 domain 的访问，设置 instance parent dir 的标签为 tmp_t，因此使用 semanage fcontext -a 添加这些路径的标签并设置（但是从/root/root.inst/的使用来看它好像是由 pam_namespace.so 创建的，而且标签也不是 tmp_t，不知道这样是否有什么副作用）。

11.3.3.3 SELinux 对 polyinstantiation 的支持

由上可知，pam_namespace.so 的主要行为及其所在的 domain 所必须具备的能力如下：
1，在 parent dir 下创建 member dir，因此要具有对前者的{ add_name search }能力，和对后者的{ create setattr }能力；
2，把当前进程的 namespace 从其 parent namespace 中 unshare，把 member dir bind-mount 到 poly dir，因此需要具有 sys_admin 能力，和对 poly dir 的{ open mounton }能力；
3，调用 namespace.init 脚本初始化 member dir，因此需要对后者的{ search write }能力；
4，为了执行 namespace.init，需要具有执行 bin_t，以及切换到 setfiles_t 的能力；
5，可能需要对 parent dir 的 relabelfrom 能力，以及对 member dir 的 relabelto 能力；

另外，由 pam_namespace.so 的源代码可知它具有以下行为：
1，如果“use_default_context”选项有效，则会调用 libselinux 提供的 getseuserbyname 函数访问 seusers 文件。而该文件为 mls_systemhigh，因此相应的 domain 必须属于 mlsfileread 属性；
2，为了支持“tmpfs”多态方法，需要对 tmpfs_t:filesystem 的{ mount，unmount }能力；
3，由于需要读写/selinux/member 文件，因此需要 security_t:security 的 compute_member 能力；

为此，在 policy.X 中设计了 polydir，polyparent，polymember 属性，对相应目录的 type 调用对应的接口以加入相应的属性。另外把上述能力都放在 files_polyinstantiate_all 接口中，并对所有能够建立登录会话的应用程序的.te 调用，比如 sshd/login 等。最后，使用 allow_polyinstantiation tunable 来控制该接口是否生效。

为了支持对 pam_namespace.so 的使用，相应 domain 比如 crond_t 必须调用 files_polyinstantiate_all 接口，并且把 allow_polyinstantiate 设置为 true。

11.3.3.4 解决在使能多态后 crond 的使用问题

由于当前已经在 login 的 PAM 配置文件中使用了 pam_namespace.so，因此 root 从串口登录后将使用/root/root.inst/*下的 memberdir 作为自己的 HOME（它被 bind mount 到/root/）。此时尽管已经要求 cronjob 写入/root/cron/results 文件，通过本地串口 login 设施登录的 root 无法看到该输出文件：

```
[root/sysadm_r/s0@QtCao ~]# mkdir cron
[root/sysadm_r/s0@QtCao ~]# crontab -l
* * * * * echo `date; id -Z` >> /root/cron/results
[root/sysadm_r/s0@QtCao ~]#
[root/sysadm_r/s0@QtCao ~]# date
Wed Feb 16 07:43:05 GMT 2011
[root/sysadm_r/s0@QtCao ~]# find cron/
cron/
[root/sysadm_r/s0@QtCao ~]#
```

另一方面，如果 root 从 ssh 登录（sshd 不使用 pam_namespace.so），则使用挂载点/root 目录，因此可以看到/root/cron/results 文件：

```
[root/secadm_r/s0@QtCao ~]# ls
cron  root.inst
[root/secadm_r/s0@QtCao ~]# cat cron/results
Wed Feb 16 07:46:01 GMT 2011 root:sysadm_r:cronjob_t:s0-s15:c0.c1023
[root/secadm_r/s0@QtCao ~]# tail -n 1 /var/log/cron.log
Feb 16 07:46:01 QtCao crond[4126]: (root) CMD (echo `date; id -Z` >> /root/cron/results)
[root/secadm_r/s0@QtCao ~]#
[root/secadm_r/s0@QtCao ~]# find . -name cron -exec ls -l {} +
./cron:
total 8
-rw-r--r-- 1 root root 138 2011-02-16 07:47 results

./root.inst/root:object_r:user_home_dir_t:s0-s15:c0.c1023_root/cron:
total 0
[root/secadm_r/s0@QtCao ~]#
```

从上面 find 的结果来看，cronjob 进程写入的不是被多态了的/root/目录，而是挂载点/root/目录。问题的核心是 cronjob 进程和用户的登录 shell 不再同一个 namespace 中，导致看到的文件系统视图不一样，因此“看不到”member dir 已经被 bind mount 到 poly dir 上。

因此必须对 crond 也使用 pam_namespace.so 模块，使得 crond 能够拼凑被多态了的用户的 HOME 目录，使得 cronjob process 能够和用户的登录 shell 共享同一个 namespace。

参照上文对 crond_t 调用 files_polyinstantiate_all 接口，使能 allow_polyinstantiation 变量，重启机器，在 crond 的 PAM 配置文件中使用 pam_namespace.so，就可以看到 cronjob 进程能够顺利写入被多态了的/root/目录了：

```
[root/sysadm_r/s0@QtCao ~]# grep namespace /etc/pam.d/crond
session    required   pam_namespace.so no_unmount_on_close
[root/sysadm_r/s0@QtCao ~]#

[root/secadm_r/s0@QtCao ~]# find . -name cron -exec ls -l {} +
./cron:
total 8
-rw-r--r-- 1 root root 345 2011-02-16 07:50 results

./root.inst/root:object_r:user_home_dir_t:s0-s15:c0.c1023_root/cron:
```

```
total 8
-rw-r--r-- 1 root root 69 2011-02-16 07:51 results
[root/secadm_r/s0@QtCao ~]# tail cron/results
Wed Feb 16 07:49:01 GMT 2011 root:sysadm_r:cronjob_t:s0-s15:c0.c1023
Wed Feb 16 07:50:02 GMT 2011 root:sysadm_r:cronjob_t:s0-s15:c0.c1023
[root/secadm_r/s0@QtCao ~]#
[root/secadm_r/s0@QtCao ~]# tail root.inst/root\:object_r\:user_home_dir_t\:s0-
s15\:c0.c1023_root/cron/results
Wed Feb 16 07:51:02 GMT 2011 root:sysadm_r:cronjob_t:s0-s15:c0.c1023
Wed Feb 16 07:52:02 GMT 2011 root:sysadm_r:cronjob_t:s0-s15:c0.c1023
[root/secadm_r/s0@QtCao ~]#
```

11.3.3.5 pam_namespace.so 源代码分析（TODO）

通过使用“debug”参数，可以打印 pam_namespace.so 的调试信息如下，可辅助分析其源代码。

```
[root/sysadm_r/s0@QtCao ~]#  cat /var/log/auth.log
......
Feb 16 07:38:01 QtCao crond[3866]: PAM unable to resolve symbol: pam_sm_acct_mgmt
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): open_session - start
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Parsing config file /etc/security/namespace.conf
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded polydir: '/tmp'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded ruser polydir: '/tmp'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded instance prefix: '/tmp-inst/'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded polydir: '/var/tmp'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded ruser polydir: '/var/tmp'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded instance prefix: '/var/tmp/tmp-inst/'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded polydir: '/root'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded ruser polydir: '/root'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Expanded instance prefix: '/root/root.inst/'
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Configured poly dirs:
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): dir='/tmp' iprefix='/tmp-inst/' meth=3
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): override user 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): override user 3
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): dir='/var/tmp' iprefix='/var/tmp/tmp-inst/' meth=3
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): override user 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): override user 3
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): dir='/root' iprefix='/root/root.inst/' meth=3
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Set up namespace for pid 3866
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /tmp for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Overriding poly for user 0 for dir /tmp
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /var/tmp for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Overriding poly for user 0 for dir /var/tmp
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /root for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Need poly ns for user 0 for dir /root
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /tmp for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /tmp for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /var/tmp for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /var/tmp for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /root for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Checking for ns override in dir /root for uid 0
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Setting poly ns for user 0 for dir /root
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Set namespace for directory /root
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): poly_name root:object_r:user_home_dir_t:s0-
s15:c0.c1023_root
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): Inst ctxt root:object_r:user_home_dir_t:s0-
s15:c0.c1023 Orig ctxt root:object_r:user_home_dir_t:s0-s15:c0.c1023
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): instance_dir
/root/root.inst/root:object_r:user_home_dir_t:s0-s15:c0.c1023_root
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): namespace setup ok for pid 3866
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): close_session - start
Feb 16 07:38:02 QtCao crond[3866]: pam_namespace(crond:session): close_session - successful
[root/secadm_r/s0@QtCao ~]#
```

11.3.3.6 有关 pam_namespace.so 的剩余问题

1，member dir 的标签应该和 poly dir 保持一致，否则将会给许多应用程序造成麻烦。比如 polydir 的标签为 user_home_dir_t，如果 member dir 的为 user_home_t，则许多应用程序无法写入后者（但是能够写入前者）

```
[root/secadm_r/s0@QtCao ~]# ls -Zd /root/
dr-xr-x---  root root root:object_r:user_home_dir_t:s0-s15:c0.c1023 /root/
[root/secadm_r/s0@QtCao ~]#
[root/secadm_r/s0@QtCao ~]# ls -Z
drwxr-xr-x  root root root:object_r:user_home_t:s0     cron
d---------  root root system_u:object_r:user_home_dir_t:s0 root.inst
[root/secadm_r/s0@QtCao ~]#
[root/secadm_r/s0@QtCao ~]# mkdir root.inst/newdir
[root/secadm_r/s0@QtCao ~]# ls -Z root.inst/
drwxr-xr-x  root root root:object_r:user_home_t:s0     newdir
drwx------  root 0    root:object_r:user_home_dir_t:s0-s15:c0.c1023 root:object_r:user_home_dir_t:s0-
s15:c0.c1023_root
[root/secadm_r/s0@QtCao ~]#
```

看 pam_namespace.so 的源代码，如何获得 poly dir 的标签，并以此 relabel parent dir 和 member dir

当前 member dir 的 FC 被明确指定了么（是否是在配置 LSPP 时）？是 namespace.init 中 restorecon 命令完成的么？

2，多态能够同时 unmount 用户挂载的文件系统，如何实现的。

> Any mounts/unmounts performed in the parent
> namespace, such as mounting of devices, are not reflected in the
> session namespace. To propagate selected mount/unmount events from the
> parent namespace into the disassociated session namespace, an
> administrator may use the special shared-subtree feature.

3，如果 root 采用非默认的 MLS range 登录，则 cronjob 进程还是只能写入如下目录：
/root/root.inst/root:object_r:user_home_dir_t:s0-s15:c0.c1023_root

即使 root 扮演更高级别的 level，但是其多态 member dir 仍然是 user_home_t:s0，因此 cronjob_t:s0 还是能够写入。关键问题是此时 crond 无从知道 root 所扮演的 level 到底是什么，因此无法得知相应的 member dir 的名称。

但是，既然使用 pam_namespace.so 将 cronjob 进程加入了 login shell 的 namespace，那么无论后者如何实现多态，前者也能够看到 bind mount 的结果呀，这样就不应该出现这个问题了？！

可能需要进一步参考附录 9 中的方法，通过 cronjob files 文件名称，或者在文件中告诉 crond 任务提交者当下所扮演的角色和安全级别，并且作用到 pam_namespace.so 拼接 member dir 目录名称的过程中。

4，思考，为什么对 crond 使用了 pam_namespace.so，就可以使得 cronjob 进程和 login 创建的 shell 进程，共享同一个 namespace？pam_namespace.so 的实现中发出了什么系统调用？如何实现这一点的？

看 ULK，关于 namespace 的内核数据结构，和 vfsmount 数据结构之间的关系。

12. refpolicy的编译，链接，扩展

12.1 描述标识符的数据结构

在refpolicy中可以使用class/common/type/role/alias/attribute/user关键字定义不同类型的标识符（identifier），词法分析程序创建描述该标识符SELinux属性的xxx_datum_t数据结构（比如policy value，和其他标识符之间的关系等）。标识符的名称字符串及其xxx_datum_t数据结构被组织在policydb_t数据结构的相应符号表中，在创建policy.X或者pp时创建其二进制表示。

12.1.1 type_datum_t

```
/* Type attributes */
typedef struct type_datum {
        symtab_datum_t s;
        uint32_t primary;       /* primary name? can be set to primary value if below is TYPE_ */
#define TYPE_TYPE 0             /* regular type or alias in kernel policies */
#define TYPE_ATTRIB 1           /* attribute */
#define TYPE_ALIAS 2            /* alias in modular policy */
        uint32_t flavor;
        ebitmap_t types;        /* types with this attribute */
#define TYPE_FLAGS_PERMISSIVE   0x01
        uint32_t flags;
        uint32_t bounds;        /* bounds type, if exist */
} type_datum_t;
```

1，type_datum_t数据结构用于描述type/alias/attribute，具体的类型由flavor域描述；
2，对于alias有两种表示方法：flavor == TYPE_TYPE && primary == 0或者flavor == TYPE_ALIAS。对于前者，alias的policy value保存在s.value中；对于后者，alias的policy value保存在primary中；
3，s.value为当前type的policy value；
4，types位图用于描述属于一个attribute的所有type（以其(policy value - 1)为索引将位图相应位置位）；
5，flags目前只有一个标志位TYPE_FLAGS_PERMISSIVE。
如果一个type没有使用“type”规则而是使用“permissive”规则定义，则在编译器中将该为置1。进而写出到policy.X或pp的p_types符号表中；当读取policy.X或者pp的p_types符号表时，如果一个type标识符的该标志位有效，则把它的policy value直接记录到policydb_t.permissive_map位图中。（关于Permissive Domain：无论当前SELinux的运行模式为Enforcing或者Permissive模式，Permissive Domain总在Permissive模式下运行，参见后文）
6，bounds域用于保存当前type所附属的type的policy value，参见define_typebounds_helper函数。附属type的能力只能是其从属type的能力的子集。

12.1.2 common_datum_t

```
/* Permission attributes */
typedef struct perm_datum {
        symtab_datum_t s;
} perm_datum_t;

/* Attributes of a common prefix for access vectors */
typedef struct common_datum {
        symtab_datum_t s;
        symtab_t permissions;   /* common permissions */
} common_datum_t;
```

common 用于描述可被 class 共享的若干 perm 集合。
1，common 集合自身具有 policy value，根据当前模块 p_commons 符号表的 nprim 分配；
2，集合关系由 permissions 符号表描述，每个 perm 除自身名称字符串外，还具有 policy value；
3，perm 的 policy value 在其所属的 common 的内部定义，基于 common_datum_t.permissions 符号表的 nprim 分配；

12.1.3 class_datum_t

```
/* Class attributes */
typedef struct class_datum {
        symtab_datum_t s;
        char *comkey;                           /* common name */
        common_datum_t *comdatum;               /* common datum */
        symtab_t permissions;                   /* class-specific permission symbol table */
        constraint_node_t *constraints;         /* constraints on class permissions */
        constraint_node_t *validatetrans;       /* special transition rules */
} class_datum_t;

typedef struct constraint_node {
        sepol_access_vector_t permissions;      /* constrained permissions */
        constraint_expr_t *expr;                /* constraint on permissions */
        struct constraint_node *next;           /* next constraint */
} constraint_node_t;
```

class 用于描述不同对象/数据结构的类别，相同“类别”的对象/数据结构支持一组相同的访问权限。一个 class 所支持的权限可以共享某一个 common，并且可以有自己的“专属”权限。
1，一个 class 有自己的 policy value，根据所在模块的 p_classes 符号表的 nprim 分配；
2，comkey/comdatum 分别指向相关 common 的名称字符串及其 common_datum_t 数据结构，注意它们都已经注册到当前模块的 p_commons 符号表中；
3，permissions 符号表用于描述当前 class 的“专属”权限；
4，constraints 队列描述定义在该 class 上的约束，每一个 constraint_node_t 数据结构描述作用于当前 class **某个** perm 上的一组约束。每一个 constraint_expr_t 数据结构描述作用于当前 perm 上的**一种**约束表达式。
5，注意，根据 expand.c 中定义的 class_copy_callback 函数，class_datum_t.permissions.nprim 为当前 class 私有 perm 和共享的 common_datum_t.permissions.nprim **之和**。

```
typedef struct constraint_expr {
#define CEXPR_NOT               1       /* not expr */
#define CEXPR_AND               2       /* expr and expr */
#define CEXPR_OR                3       /* expr or expr */
#define CEXPR_ATTR              4       /* attr op attr */
#define CEXPR_NAMES             5       /* attr op names */
        uint32_t expr_type;             /* expression type */

#define CEXPR_USER 1            /* user */
#define CEXPR_ROLE 2            /* role */
#define CEXPR_TYPE 4            /* type */
#define CEXPR_TARGET 8          /* target if set, source otherwise */
#define CEXPR_XTARGET 16        /* special 3rd target for validatetrans rule */
#define CEXPR_L1L2 32           /* low level 1 vs. low level 2 */
#define CEXPR_L1H2 64           /* low level 1 vs. high level 2 */
#define CEXPR_H1L2 128          /* high level 1 vs. low level 2 */
#define CEXPR_H1H2 256          /* high level 1 vs. high level 2 */
#define CEXPR_L1H1 512          /* low level 1 vs. high level 1 */
```

```
#define CEXPR_L2H2 1024         /* low level 2 vs. high level 2 */
        uint32_t attr;          /* attribute */

#define CEXPR_EQ     1          /* == or eq */
#define CEXPR_NEQ    2          /* != */
#define CEXPR_DOM    3          /* dom */
#define CEXPR_DOMBY  4          /* domby  */
#define CEXPR_INCOMP 5          /* incomp */
        uint32_t op;            /* operator */

        ebitmap_t names;        /* names */
        struct type_set *type_names;

        struct constraint_expr *next;   /* next expression */
} constraint_expr_t;
```

（TODO：还得进一步理解 policy_parser.y 和 policy_define.c 中定义的 define_cexpr 和 define_constraints 函数，从而理解 constraint_expr_t 中的各个域）

12.1.4 role_datum_t

```
/* Role attributes */
typedef struct role_datum {
        symtab_datum_t s;
        ebitmap_t dominates;    /* set of roles dominated by this role */
        type_set_t types;       /* set of authorized types for role */
        ebitmap_t cache;        /* This is an expanded set used for context validation during parsing */
        uint32_t bounds;        /* bounds role, if exist */
} role_datum_t;
```

1，s.value 即为当前 role 标识符的 policy value；
2，dominates 位图用于描述被当前 role 所 dominate 的其它 role 的集合，以 sub-role 的(policy value - 1)为索引将位图中相应位置位；
3，types 用于描述当前 role 能够关联的所有 type 的集合，在 expand_module > policydb_index_others > policydb_role_cache 函数中，将 types 中的 types/negset 结合 type_set_t.flags 展开为 cache 位图。cache 位图中所有非 0 位即为能够和当前 role 相关联的 type 的(policy value - 1)，可直接索引 type_value_to_struct[]得到相应 type 的 type_datum_t 的地址；
4，bounds 应该描述当前 role 所从属的 super-role 的 policy value（？）。

注意，在 RBAC 中 role 的作用/目的就是控制能够和那些 type 相关联而组成合法的 SC（但是还需要额外的 TE 规则以支持 domain transition），因此 role_datum_t 数据结构中最重要的域莫过于 types 了。在 SELinux 内核的 role_datum_t 中也设计了 types 位图，在 security_compute_sid > policydb_context_isvalid 中检查当前拼凑的 newcontext 中 type 是否在 role 的 types 位图中。

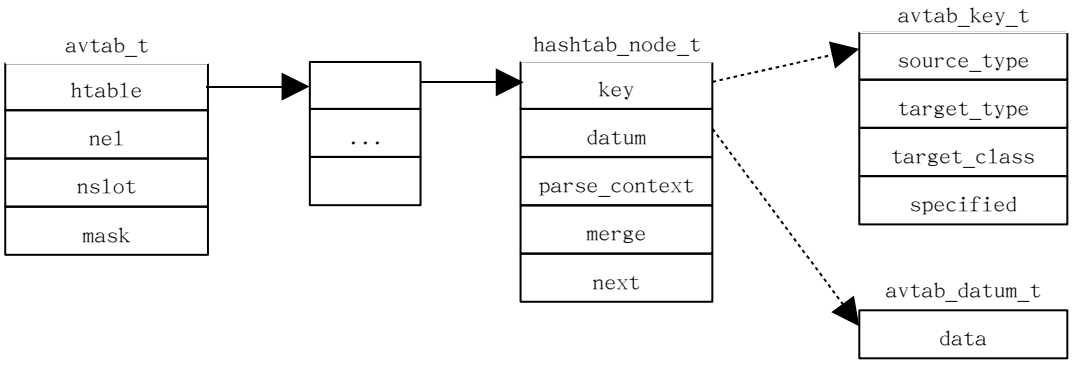
dominates 域描述当前 role（dominating role）能够 dominate 哪些其他的 role（dominated role），而 dominating role 能够关联所有 dominated role 相关联的所有 types，所以相应的词法分析函数将 dominated role 的 dominates 和 types 位图，合并（escalate）到 dominating role 的 dominates 和 types 位图中。

注意，正是由于 role dominance 规则仅仅在**编译时**处理，因此在 role dominance 规则之后如果 dominated role 能够和新的 type 相关联，则 dominating role 无法获得这种新的关联。与之相比对于 type attribute，在 link 过程中通过 type_fix_callback 函数将当前 type 属性的 types 位图（将非 0 位置的坐标转换为新的坐标后）合并到 base 模块的相应 type 属性中，所以任何一个 type-attribute 关系声明都会最终生效。

12.2 描述规则的数据结构

由下文可知，policydb_t 使用不同的数据结构表示同一规则的“字面”和“展开”描述：在模块中使用“字面”表示，而在 policy.X 中使用“展开”描述。这些数据结构在 policydb_t 中的组织方式不同，在词法分析时分别加入各自的队列。相应地，同一规则在模块和 policy.X 的二进制表示也不同。

12.2.1 AVTAB_AV 和 AVTAB_TYPE 类规则



任何一条 TE 规则（比如 allow 或 type_transition 规则）都满足如下格式：

规则类型 source_type target_type : target_class permission/new_type;

在 policy.X 中使用 avtab_key_t 和 avtab_datum_t 数据结构作为规则的“展开”描述，它们都组织在 avtab_node_t 数据结构中。

```
typedef struct avtab_key {
        uint16_t source_type;
        uint16_t target_type;
        uint16_t target_class;
#define AVTAB_ALLOWED     1
#define AVTAB_AUDITALLOW  2
#define AVTAB_AUDITDENY   4
#define AVTAB_NEVERALLOW 128
#define AVTAB_AV         (AVTAB_ALLOWED | AVTAB_AUDITALLOW | AVTAB_AUDITDENY)
#define AVTAB_TRANSITION 16
#define AVTAB_MEMBER     32
#define AVTAB_CHANGE     64
#define AVTAB_TYPE       (AVTAB_TRANSITION | AVTAB_MEMBER | AVTAB_CHANGE)
#define AVTAB_ENABLED_OLD 0x80000000
#define AVTAB_ENABLED    0x8000 /* reserved for used in cond_avtab */
        uint16_t specified;     /* what fields are specified */
} avtab_key_t;

typedef struct avtab_datum {
        uint32_t data;          /* access vector or type */
} avtab_datum_t;
```

其中 avtab_key_t.specified 域用于描述“规则类型”，根据规则的 key 计算当前规则的哈希值并组织在不同的冲突队列中，所有规则用 policydb_t.te_avtab 哈希表描述，参见上图。而条件规则则用 policydb_t.te_cond_list 哈希表描述。对于 AVTAB_AV 类规则，avtab_datum_t.data 即为所许可的 permission 的位图，对于 AVTAB_TYPE 类规则，它即为 new_type。

```
struct avtab_node {
```

```
        avtab_key_t key;
        avtab_datum_t datum;
        avtab_ptr_t next;
        void *parse_context;    /* generic context pointer used by parser;
                                 * not saved in binary policy */
        unsigned merged;        /* flag for avtab_write only;
                                   not saved in binary policy */
};
```

注意 avtab_node_t.parse_context 指针，在当前 avtab_node_t 对应着一个条件规则的展开描述时有效，指向该条件规则在 out->cond_list 队列中相应元素的 true_list 或 false_list 队列的首地址。参见下文 find_avtab_node 函数分析。

这是因为，条件规则可能和非条件规则，展开后具有相同的 avtab_key_t，因此在搜索冲突项链表时，要根据 parse_context 指针寻找和当前条件规则对应的 avtab_node_t 节点。

在**模块**中使用 avrule_t 数据结构作为规则的“字面”描述：

```
typedef struct avrule {
/* these typedefs are almost exactly the same as those in avtab.h - they are
 * here because of the need to include neverallow and dontaudit messages */
#define AVRULE_ALLOWED     1
#define AVRULE_AUDITALLOW  2
#define AVRULE_AUDITDENY   4
#define AVRULE_DONTAUDIT   8
#define AVRULE_NEVERALLOW 128
#define AVRULE_AV          (AVRULE_ALLOWED | AVRULE_AUDITALLOW | AVRULE_AUDITDENY | AVRULE_DONTAUDIT |
AVRULE_NEVERALLOW)
#define AVRULE_TRANSITION 16
#define AVRULE_MEMBER     32
#define AVRULE_CHANGE     64
#define AVRULE_TYPE       (AVRULE_TRANSITION | AVRULE_MEMBER | AVRULE_CHANGE)
        uint32_t specified;                     # 规则类型
#define RULE_SELF 1
        uint32_t flags;                         # 表明在 target_type 位置上是否使用了“self”
        type_set_t stypes;
        type_set_t ttypes;
        class_perm_node_t *perms;
        unsigned long line;     /* line number from policy.conf where this rule originated  */
        struct avrule *next;
} avrule_t;

typedef struct class_perm_node {
        uint32_t class;                                         # 所涉及的 class
        uint32_t data;          /* permissions or new type */   # 所涉及的 permission 位图
        struct class_perm_node *next;
} class_perm_node_t;
```

和在 policy.X 中使用的 avtab_key_t 描述方式相比，在模块中使用的 avrule_t 描述方式的主要区别如下：
1，前者中使用 uint16_t 描述一组确定的 source_type/target_type/target_class，而在后者中使用 type_set_t 数据结构描述 source type 和 target type 的“字面”语义，可能包含扩展以及“*”，“~”或“-”等操作符；
2，前者组织在 avtab 哈希表中（便于内核快速查找），而后者组织在当前模块当前 block/decl（由 stack_top->decl 所指向）的 avrule_decl_t.avrules 链表中；
3，前者经由后者 link/expand 而创建。

class_perm_node_t 数据结构用于描述和规则中所涉及的 target_class 的那些权限位。由于规则中可能在 class 域上使用扩展，所以需要为**每个** class 都分配一个 class_perm_node_t 数据结构，并组织在 avrule_t.perms 队列中。由后文可见 class_perm_node_t.data 即为规则中所涉及的当前 class 的 permission 的位图（相应位被置位，目前每个 class 最多只支持 32 个 permission）。

12.2.2 role_transition 规则

```
typedef struct role_trans_rule {
        role_set_t roles;       /* current role */
        type_set_t types;       /* program executable type, or new object type */
        ebitmap_t classes;      /* process class, or new object class */
        uint32_t new_role;      /* new role */
        struct role_trans_rule *next;
} role_trans_rule_t;
```

role_trans_rule_t 数据结构为 role_transition 规则的“字面”描述，roles/types 中都含有 ebitmap 和 flags 数据结构，支持规则中各个成分的语法扩展，以及特殊字符“*”、“-”和“~”的使用（比如从一个属性中去除指定的 type）。在编译模块时，词法分析程序根据当前 role_transition 规则创建其“字面”描述，并加入当前 block/decl（由 stack_top->decl 指向）的 avrule_decl_t.role_tr_rules 队列。

```
typedef struct role_trans {
        uint32_t role;          /* current role */
        uint32_t type;          /* program executable type, or new object type */
        uint32_t tclass;        /* process class, or new object class */
        uint32_t new_role;      /* new role */
        struct role_trans *next;
} role_trans_t;
```

而 role_trans_t 数据结构为 role_transition 规则的“展开”描述。经过模块的 compile/link/expand 过程，role_transition 规则的“字面”描述被最终展开为 role_trans_t 数据结构，组织在 policydb_t.role_tr 队列中，最终根据该队列中的元素写入 policy.X 文件。

12.3 用户态 policydb_t 数据结构分析

为了能够在没有 refpolicy 源代码时、在运行时动态地修改当前 SELinux policy 的属性并增加新的规则（比如增加 Linux user 到 seuser 的映射，指定 seuser 所能够扮演的 role，或者使用 semodule 安装新的模块），由用户态的 libsepol 库提供 policy 的组织结构和访问方法，而用户态的应用程序（比如 init，load_policy，semanage，semodule）就可以使用相应的库函数修改 policy.X 的内容了。比如 load_policy 在从磁盘上读取 policy.X 后会根据单独的配置文件修改其中 boolean 的设置或者增加 seuser（但是这种修改并不会回写到磁盘上）。

12.3.1 policydb_t 数据结构综述

用户态的 policydb_t 数据结构用于描述定义于 policy.X（kernel policy）、base 模块或非 base 模块中所有 SELinux 标识符和规则，根据它创建 policy.X 映像或者模块的 pp 文件。SELinux 内核驱动在读取 policy.X 映像时，翻译其中标识符和规则的二进制描述并创建相应的内核态数据结构，再组织到内核态 policydb_t 数据结构中。

注意，用户态 policydb_t 包含了一些模块所特有的数据结构（比如规则的“字面”描述，类似于 { file_type - shadow_t }，或者表示标识符 scope 的数据结构），这些部分在内核态的 policydb_t 中

并不需要。由于从任何一个模块的角度看都无法知道一个标识符在整个 policy 中的完整属性（比如无法知道 file_type 属性所包含的所有 type 集合），因此无法在编译模块时展开规则，而延迟到将所有模块 link，expand 而创建最终的 policy.X 时再展开。这样内核态 policydb_t 中可以使用 avtab 哈希表根据一组具体的 source_type/target_type/target_class 来快速索引 TE 规则。

与此类似，policydb_t 中和 policy.X 相关的设施对模块而言也是不需要的，比如提供快速查找 TE 规则的哈希表（而 role_transition/role_allow/range_transition 规则由于个数不多目前仍组织为单向链表）。

尽管描述 policy.X 和模块中描述规则的数据结构不尽相同，还是把它们都放到一个 policydb_t 中，这样做可以尽可能地简化代码避免重复（另，base 模块和非 base 模块的区别仅是只有在前者中支持指定 filesystem 或者 networking 的 contexts）。

用户态 policydb_t 数据结构的分段介绍如下：

```
/* The policy database */
typedef struct policydb {
#define POLICY_KERN SEPOL_POLICY_KERN
#define POLICY_BASE SEPOL_POLICY_BASE
#define POLICY_MOD SEPOL_POLICY_MOD
        uint32_t policy_type;
        char *name;
        char *version;
        int  target_platform;

        /* Set when the policydb is modified such that writing is unsupported */
        int unsupported_format;

        /* Whether this policydb is mls, should always be set */
        int mls;
```

1，policy_type 描述该 policydb_t 数据结构所描述的主体的类型（比如读取的是 policy.X，或者 base.pp，或者其他 pp）；
2，version 和 name 只对模块有效，为 x.pp 的名称和版本号，在.te 中用 policy_module 宏指定，在 define_policy 函数中设置。
如果在 MONOLITHIC=n 时编译，可以指定/etc/selinux/semanage.conf 中的 policy-version 变量为较小的版本号，即为最终 link/expand 后得到的 policy.X 的版本号。

注意对于 policy.X 而言，后缀“.X”即为版本号，默认情况下为 libsepol 所支持的最大版本号，但是可以用 build.conf 文件中定义的 OUTPUT_POLICY 变量重载，可以把它指定为较小的版本号，即在编译时发生 policy downgrade，即可以把同一套 refpolicy 实现编译为不同的版本，而不同版本 policy.X 的规则数量及二进制表示的格式会存在差异（在编译为较低版本时，所有使用新特性的规则将被排除，并且保留下来的规则仍沿用旧版本定义的二进制格式），详见后文。

另外，如果 policy.X 版本号大于 SELinux 内核所支持的版本号，则在 load 的过程中会发生 policy downgrade。所以，如果递增了用户态 Security Server 的版本号，需要相应地修改内核态 Security Server 的版本号以支持/使用用户态定义的新特性。
3，mls 表示是否支持 MLS 特性；
（TODO：进一步了解 target_platform 和 unsupported_format 的作用）

上面这些域无论 policy.X 或者模块都需要。

```
        /* symbol tables */
```

```
        symtab_t symtab[SYM_NUM];
#define p_commons symtab[SYM_COMMONS]
#define p_classes symtab[SYM_CLASSES]
#define p_roles symtab[SYM_ROLES]
#define p_types symtab[SYM_TYPES]
#define p_users symtab[SYM_USERS]
#define p_bools symtab[SYM_BOOLS]
#define p_levels symtab[SYM_LEVELS]
#define p_cats symtab[SYM_CATS]

        /* symbol names indexed by (value - 1) */
        char **sym_val_to_name[SYM_NUM];
#define p_common_val_to_name sym_val_to_name[SYM_COMMONS]
#define p_class_val_to_name sym_val_to_name[SYM_CLASSES]
#define p_role_val_to_name sym_val_to_name[SYM_ROLES]
#define p_type_val_to_name sym_val_to_name[SYM_TYPES]
#define p_user_val_to_name sym_val_to_name[SYM_USERS]
#define p_bool_val_to_name sym_val_to_name[SYM_BOOLS]
#define p_sens_val_to_name sym_val_to_name[SYM_LEVELS]
#define p_cat_val_to_name sym_val_to_name[SYM_CATS]

        /* class, role, and user attributes indexed by (value - 1) */
        class_datum_t **class_val_to_struct;
        role_datum_t **role_val_to_struct;
        user_datum_t **user_val_to_struct;
        type_datum_t **type_val_to_struct;
```

在 refpolicy 中所有定义的标识符都分门别类地组织在各自的符号表（哈希表）中，相应的 p_xxx_val_to_name[]指针数组用标识符的(policy value - 1)索引其 key（即标识符的名称字符串）；进一步对于 class/user/role/type 类标识符 xxx_val_to_struct[]指针数组用其(policy value - 1)索引其 xxx_datum_t 数据结构。

由于 policy value 从 1 开始分配，所以以(policy value - 1)索引上面的指针数组或者 ebitmap 位图。

上面这些域无论 policy.X 或者模块都需要。

```
        /* module stuff section -- used in parsing and for modules */

        /* keep track of the scope for every identifier.  these are
         * hash tables, where the key is the identifier name and value
         * a scope_datum_t.  as a convenience, one may use the
         * p_*_macros (cf. struct scope_index_t declaration). */
        symtab_t scope[SYM_NUM];
```

只有模块才需要使用 scope 符号表。哈希表元素的 key 为标识符名称字符串，而 datum 为 scope_datum_t 数据结构，用于描述标识符的“定义域”和“使用域”，即当前标识符在哪个 avrule_decl_t 中定义或者使用。和 symtab 类似，按照标识符的不同类型将它们组织在不同的 scope 符号表中。

```
        /* module rule storage */
        avrule_block_t *global;
        /* avrule_decl index used for link/expand */
        avrule_decl_t **decl_val_to_struct;
```

avrule_block_t 和 avrule_decl_t 数据结构用于描述一个模块中所有的规则。任何一个模块都包含一个 unconditional block（即模块的主体部分，包含标识符的定义，规则的定义等，又称为 global block），若干 optional block（即使用 optional_block 宏限制的接口调用），若干 conditional

block（即使用 tunable_block 宏限制的接口调用）。

在 policydb_init 函数中分配并初始化 policydb_t 数据结构时，就创建了当前模块的第一个 avrule_block_t 数据结构，由 policydb_t.global 指针指向。当遇到一个 optional_block 宏时，才创建另外一个 avrule_block_t 数据结构，组织到 policydb_t.global 队列中。

在任何一个 block 中所定义的标识符和规则，实际上使用 avrule_decl_t 数据结构来描述。任何一个 block 都至少包含一个 decl，组织在 avrule_block_t.branch_list 队列中。但是任何时候只有一个被使能/有效（比如 optional block 中只有一个分支是有效的），由 avrule_block_t.enabled 指向。另外为了 link/expand 过程的使用方便，在 policydb_t 中为 avrule_decl_t 数据结构设计了 decl_val_to_struct[]指针数组（在模块内每个 avrule_decl_t 都有惟一的索引，即其 decl_id）。

```
/* compiled storage of rules - use for the kernel policy */
```

只有 policy.X 或者内核 policydb_t 才需要以下的数据结构：

```
/* type enforcement access vectors and transitions */
avtab_t te_avtab;
```

te_avtab 哈希表用于描述所有非条件规则，哈希表元素的 key 为相应 TE 规则的类型，source_type，target_type， target_class，而 datum 为相应 TE 规则的最后一个域（permissions 或者 new_type），参见上文。

```
/* bools indexed by (value - 1) */
cond_bool_datum_t **bool_val_to_struct;
/* type enforcement conditional access vectors and transitions */
avtab_t te_cond_avtab;
/* linked list indexing te_cond_avtab by conditional */
cond_list_t *cond_list;
```

te_cond_avtab 哈希表用于描述所有的条件规则；cond_list 队列用于组织所有条件规则的 cond_node_t 数据结构，用于描述相应 if-else 结构的条件表达式的状态值，以及指向条件规则在 te_cond_avtab 哈希表中 avtab_node_t 元素的索引数据结构 cond_av_list_t 结构。

```
/* role transitions */
role_trans_t *role_tr;

/* role allows */
role_allow_t *role_allow;

/* range transitions */
range_trans_t *range_tr;
```

role_tr，role_allow，range_tr 分别指向 policy.X 中 role_transition，role_allow 和 range_transition 规则“展开”描述的单向链别。

```
/* security contexts of initial SIDs, unlabeled file systems,
   TCP or UDP port numbers, network interfaces and nodes */
ocontext_t *ocontexts[OCON_NUM];

/* security contexts for files in filesystems that cannot support
   a persistent label mapping or use another
   fixed labeling behavior. */
genfs_t *genfs;
```

ocontexts 和 genfs 只在 policy.X 和 base 模块中有效，为相应 contexts 的定义（非 base 模块不能定义这些 contexts）。

```
        ebitmap_t *type_attr_map;

        ebitmap_t *attr_type_map;       /* not saved in the binary policy */
```

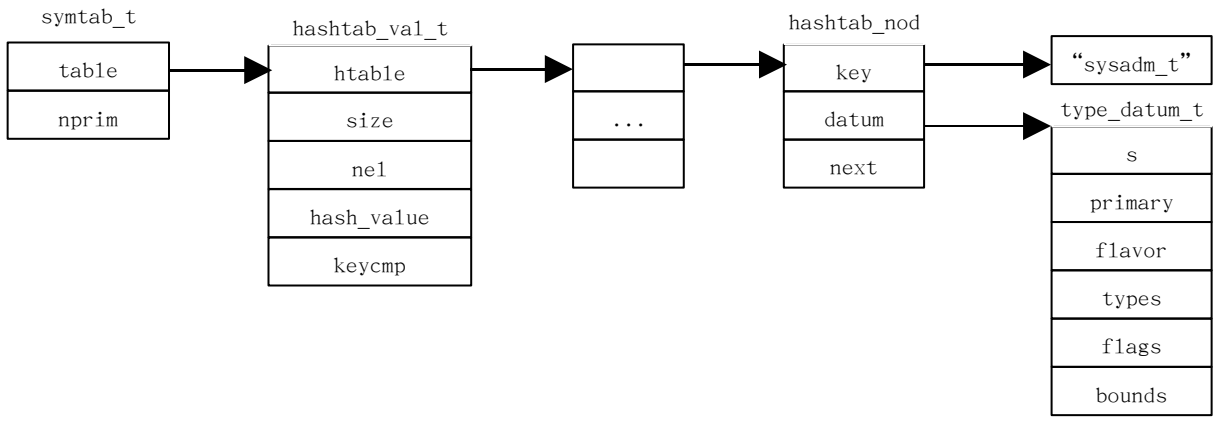
type_attr_map[]位图数组以普通 type 的(policy value - 1)为索引，得到的 ebitmap 用于描述当前普通 type 所从属的所有属性 type；attr_type_map[]位图数组以属性 type 的(policy value - 1)为索引，得到的 ebitmap 即为当前属性 type 的 type_datum_t.types 位图的拷贝。在模块 expand 过程的最后，由 expand_module 创建这两个数组并调用 type_attr_map 函数初始化，它们的长度均为 out 模块 p_types.nprim，参见下文。

```
        ebitmap_t policycaps;

        /* this bitmap is referenced by type NOT the typical type-1 used in other
           bitmaps.  Someday the 0 bit may be used for global permissive */
        ebitmap_t permissive_map;

        unsigned policyvers;

        unsigned handle_unknown;
} policydb_t;
```

（TODO：剩余这几个数据结构的作用尚未涉及）

12.3.2 symtab 符号表

policydb_t 或者 avrule_decl_t 数据结构中 symtab 符号表的组织结构关系如下图所示：



1，所有标识符的 key（即标识符名称字符串）及其 xxx_datum_t 数据结构都由一个 hashtab_node_t 数据结构描述；
2，htable 指向指针数组，元素指向冲突项链表。size 为指针数组的长度，而 nel 为该哈希表中所有 hashtab_node_t 的个数。hash_value 和 keycmp 分别指向哈希值计算和比较函数（比如 strcmp）；
3，注意 nprim 为当前符号表中所有标识符 policy value 的最大值。比如当所有非 base 模块都向 base 模块 link 时，要依据 base.p_types.nprim 为模块的 type 重新分配 policy value。

12.3.3 avrule_block_t，avrule_decl_t 和 scope_stack_t

任何模块都至少包含一个 global block（模块首部 policy_module 宏之后的部分），还可能包含若干

optional block（使用 optional_policy 宏），它们被统称为 avrule block。对于 optional block 还可能存在两个分支，任何时候只有一个分支有效（无效的分支称为 else branch）。

模块中还可能包含若干 conditional block（使用 tunable_policy 宏）。

所有这些 block 都用 avrule_block_t 数据结构描述：

```
typedef struct avrule_block {
        avrule_decl_t *branch_list;
        avrule_decl_t *enabled; /* pointer to which branch is enabled.  this is
                                   used in linking and never written to disk */
#define AVRULE_OPTIONAL 1
        uint32_t flags;         /* any flags for this block, currently just optional */
        struct avrule_block *next;
} avrule_block_t;
```

由后文 enable_avrule 函数可见，任何一个 block 至多有两个 decl 组织在 branch_list 队列中。任何情况下当前 block 中只有一个 decl 有效，由 enabled 指针指向。对于有效的 decl，其 avrule_decl_t.enabled 标志位被设置。

如果当前 block 由 policy_module 宏创建，则称当前 block 为 unconditional block；如果由 optional_policy 宏创建，则称为 optional block，此时 flags 中 AVRULE_OPTIONAL 标志位有效。

注意，tunable_policy 宏不创建 block/decl，而被翻译为 if-else conditional，此时用 cond_node_t 数据结构描述 if/else 分支中的规则，加入当前 avrule_decl_t 的 cond_list 队列。即，一个 decl 中可能包含若干 tunable_policy 宏。（注意，目前尚未支持 tunable_policy 宏的嵌套。需要么？）

由 enable_avrule 函数可见，link 过程的后期默认使能 optional block 的 non-else decl，如果它的外部依赖不满足，则使能 else decl（branch_list 队列的第 2 个元素）。

一个 block 中所有语法成分：标识符定义或引用、规则的定义、接口的调用，都用 avrule_decl_t 数据结构描述，它们组织在 block 的 branch_list 队列中，其分段介绍如下：

```
typedef struct avrule_decl {
        uint32_t decl_id;
        uint32_t enabled;       /* whether this block is enabled */
```

一个模块中所有 avrule_decl_t 数据结构都有惟一的编号，decl_id 即为当前结构的编号。

对于一个 optional block 而言，只有其所有外部引用都被满足时其 non-else 或者 else 分支的 decl 才会被使能，此时其 enabled 标志位被置位。参见 link_modules > enable_avrules 函数。

```
        cond_list_t *cond_list;                 # if-else conditional
        avrule_t *avrules;                      # AVTAB_AV 或 AVTAB_TYPE 类规则
        role_trans_rule_t *role_tr_rules;       # role_transition 规则
        role_allow_rule_t *role_allow_rules;    # role_allow 规则
        range_trans_rule_t *range_tr_rules;     # range_transition 规则
```

上面这些数据结构即用于描述一个 block 中的所有规则。注意 cond_list 队列中的每一个 cond_node_t 元素分别描述一个 tunable_policy 宏的 if-else 结构。

```
        scope_index_t required;         /* symbols needed to activate this block */
        scope_index_t declared;         /* symbols declared within this block */
```

scope_index_t 数据结构包含若干位图，用于描述当前 block 所引用或定义的 7 种标识符，可用于计算 block 之间的依赖关系图。

```
        /* type transition rules with a 'name' component */
        filename_trans_rule_t *filename_trans_rules;

        /* for additive statements (type attribute, roles, and users) */
        symtab_t symtab[SYM_NUM];

        /* In a linked module this will contain the name of the module
         * from which this avrule_decl originated. */
        char *module_name;

        struct avrule_decl *next;
} avrule_decl_t;
```

在编译模块时，每当发现一个新的 block 就为其创建相应的 avrule_block_t 和 avrule_decl_t 数据结构，并且调用 push_stack 函数创建相应的 scope_stack_t 数据结构。当前模块的所有 block 的 scope_stack_t 数据结构组织成一个栈，栈顶由 stack_top 全局变量指向。参见后文 12.4.1 小节。

```
union stack_item_u {
        avrule_block_t *avrule;
        cond_list_t *cond_list;
};

typedef struct scope_stack {
        union stack_item_u u;
        int type;               /* for above union: 1 = avrule block, 2 = conditional */
        avrule_decl_t *decl;    /* if in an avrule block, which
                                 * declaration is current */
        avrule_t *last_avrule;
        int in_else;            /* if in an avrule block, within ELSE branch */
        int require_given;      /* 1 if this block had at least one require */
        struct scope_stack *parent, *child;
} scope_stack_t;
```

在模块源代码中可以使用 policy_module 宏来定义一个模块（的 unconditional/global block）；用 optional_policy 宏来声明一个 optional block（根据外部依赖是否被满足而决定是否被使能）；用 tunable_policy 宏来声明一个 if-else conditional，注意它并不对应任何 block/decl，而是隶属于当前 block/decl。

global block 和 optional block 统称为 avrule block，此时 type == 1;

对 optional block 而言，如果不在“else”分支上，则 in_else 为 0，否则为 1。

注意在“else”分支上的 optional block，或者在 if-else conditional 中不能定义标识符，也不能声明外部依赖。

当前 block/decl 中如果使用 require 声明了一个外部依赖，则 require_given 标志位有效，参见 require_symbol 函数。

注意，当前 stack_item_u 数据结构的设计有冗余，因为一个 if-else conditional 并不对应一个 block/decl。而且从代码来看，push_stack 函数的第一个参数总是 1 而不会是 2。

12.3.4 scope_datum_t - 描述标识符的定义者和使用者

任何标识符都可以在一个模块的某一个 block 内定义，而被该模块的其他 block、或者被其他模块的 block 所使用。因此需要用 scope_datum_t 数据结构描述一个标识符的定义者和使用者，从而确定模块之间的依赖关系图，并确认一个 block 的外部依赖是否被满足。

```
/* Every identifier has its own scope datum.  The datum describes if
 * the item is to be included into the final policy during
 * expansion. */
typedef struct scope_datum {
/* Required for this decl */
#define SCOPE_REQ  1
/* Declared in this decl */
#define SCOPE_DECL 2
        uint32_t scope;
        uint32_t *decl_ids;
        uint32_t decl_ids_len;
        /* decl_ids is a list of avrule_decl's that declare/require
         * this symbol.  If scope==SCOPE_DECL then this is a list of
         * declarations.  If the symbol may only be declared once
         * (types, bools) then decl_ids_len will be exactly 1.  For
         * implicitly declared things (roles, users) then decl_ids_len
         * will be at least 1. */
} scope_datum_t;
```

模块的 policydb_t.scope[]符号表用于描述当前模块内各标识符的定义或声明所在的 block/decl，该符号表元素的 key 即为标识符名称字符串，datum 为相应的 scope_datum_t 数据结构，其中 scope 标志位表明当前数据结构的类型（定义者或使用者），decl_ids[]数组描述所有定义或使用该标识符的 avrule_decl_t 数据结构的编号，而 decl_ids_len 为该数组的长度。

一个标识符在一个模块内的 scope_datum_t 数据结构的 scope 域要么为 SCOPE_DECL，要么为 SCOPE_REQ，只有这两种可能，这是因为：
1，如果首先在一个 block/decl 中声明为 SCOPE_REQ，但是后来的 block/decl 中又声明为 SCOPE_DECL，则作为 SCOPE_DECL 处理（即，该模块既声明了该标识符，又定义了它，则算作该标识符的定义者。注意，此时 decl_ids[]数组长度>1，而且前面若干元素记录的 block/decl 可能为使用者而不是定义者）；
2，如果一个标识符已经被认为由当前模块内定义，但是如果当前 block/decl 又声明它为外部依赖，则为非法情况；
3，如果一个模块内若干 block/decl 都声明了同一个标识符为外部依赖，则为合法情况（此时该标识符的 scope_datum_t.decl_ids[]数组中记录所有 block/decl 的编号）；
4，如果一个模块内若干 block/decl 都定义了同一个标识符，除非为 role/user 类标识符，否则为非法情况（就 role/user 类标识符，decl_ids[]数组长度>1；对于其他类标识符，该数组长度==1）。

详见下文 symtab_insert 函数。

另外，对于完成 link 过程的 base 模块而言，每个标识符都只保留有一个 SCOPE_DECL 类型的 scope_datum_t 数据结构，参见下文 scope_copy_callback 函数分析。而在 expand 过程中拷贝 base 模块的标识符到 out 模块时，检查当前标识符在 base 模块相应 scope 符号表中的 scope_datum_t 的类型是否为 SCOPE_DECL，及其 decl_ids[]数组中索引的 avrule_decl_t.enabled 标志位是否有效。参见 is_id_enabled 函数。

12.3.5 scope_index_t - 描述一个 block/decl 内定义或引用的标识符

```
/* scope_index_t holds all of the symbols that are in scope in a
 * particular situation.  The bitmaps are indices (and thus must
```

```
 * subtract one) into the global policydb->scope array. */
typedef struct scope_index {
        ebitmap_t scope[SYM_NUM];
#define p_classes_scope scope[SYM_CLASSES]
#define p_roles_scope scope[SYM_ROLES]
#define p_types_scope scope[SYM_TYPES]
#define p_users_scope scope[SYM_USERS]
#define p_bools_scope scope[SYM_BOOLS]
#define p_sens_scope scope[SYM_LEVELS]
#define p_cat_scope scope[SYM_CATS]

        /* this array maps from class->value to the permissions within
         * scope.  if bit (perm->value - 1) is set in map
         * class_perms_map[class->value - 1] then that permission is
         * enabled for this class within this decl.  */
        ebitmap_t *class_perms_map;
        /* total number of classes in class_perms_map array */
        uint32_t class_perms_len;
} scope_index_t;
```

如上文所述，avrule_decl_t.required/declared 即为 scope_index_t 数据结构，用于描述当前 block 所引用或者定义的 7 种标识符（在相应位图中以标识符的(policy value - 1)为索引，相应位被置位）。

在 link 过程中完成标识符的拷贝后，各个标识符的定义/引用情况被确定（各个标识符的 scope_datm_t 数据结构完成合并），此时就可以检查 avrule_decl_t.required 数据结构中所引用的标识符是否被定义。如果否，则在 link 过程的后部在 enable_avrules 函数中不使能当前 avrule_decl_t。

12.3.6 cond_node_t - 描述一个 if-else conditional

```
/*
 * A cond node represents a conditional block in a policy. It
 * contains a conditional expression, the current state of the expression,
 * two lists of rules to enable/disable depending on the value of the
 * expression (the true list corresponds to if and the false list corresponds
 * to else)..
 */
typedef struct cond_node {
        int cur_state;
        cond_expr_t *expr;
        /* these true/false lists point into te_avtab when that is used */
        cond_av_list_t *true_list;                              # 等到 expand 时加入 te_cond_avtab 时才创建
        cond_av_list_t *false_list;                             # 等到 expand 时加入 te_cond_avtab 时才创建
        /* and these are using during parsing and for modules */
        avrule_t *avtrue_list;                                  # 只在 compile/link 时使用，不写入 policy.X
        avrule_t *avfalse_list;                                 # 只在 compile/link 时使用，不写入 policy.X
        /* these fields are not written to binary policy */     # 只在 compile/link 时使用，不写入 policy.X
        unsigned int nbools;
        uint32_t bool_ids[COND_MAX_BOOLS];
        uint32_t expr_pre_comp;
        struct cond_node *next;
} cond_node_t;
```

tunable_policy 宏被 m4 展开为 if-else 结构，根据表达式中若干 tunable/boolean 的值，决定 if 分支或者 else 分支中的规则被生效（对于 tunable，在 link/expand 时生效的规则会被永久写入 policy.X；而对于 boolean，true/false 规则都会被写入 policy.X，根据运行时 boolean 的数值决定采纳哪一个）
1，cur_state 域即为当前条件表达式的数值；

由于条件表达式所涉及的 tunable/boolean，可能并不完全在当前模块内定义，所以必须要等到 link 过程中拷贝完所有的符号表之后才能确定当前条件表达式的结果。
2，expr 队列即用于描述条件表达式，参见下文；
3，avtrue_list 和 avfalse_list 队列分别组织 if 分支和 else 分支的规则。注意 avrule_t 数据结构为规则的“字面”描述；
注意，具体哪一个分支的规则最终生效，还得有 cur_state 的值决定，而和字面单词的含义无关；
4，true_list 和 false_list 队列中的元素都为指针，指向 if/else 分支中规则的“展开”描述在 te_cond_avtab 哈希表中的相应表项；
5，nbools 和 bool_ids[]数组描述条件表达式中 tunable/boolean 标识符的个数，以及标识符的(policy value - 1)信息
6，expr_pre_comp 为条件表达式的计算值，当包含的 boolean/tunable 个数不超过 5 时。

注意，avtrue_list 和 avfalse_list 队列只在编译和 link 时被使用。而 if/else 分支的所有规则的展开描述在 expand 过程中加入 te_cond_avtab 哈希表，在该哈希表中的节点分别由 true_list/false_list 中的元素指向，这两个队列只在 expand 过程中才被创建，并最终被写入 policy.X。

这样的好处是在运行时改变 boolean 的数值时，可以从 te_cond_avtab 哈希表中迅速地定位所有相关的 avtab_node_t 节点，反转 avtab_key_t.specified 中的 AVTAB_ENABLED 标志（标志当前“展开”规则是否有效）

```
/*
 * A conditional expression is a list of operators and operands
 * in reverse polish notation.
 */
typedef struct cond_expr {
#define COND_BOOL       1       /* plain bool */
#define COND_NOT        2       /* !bool */
#define COND_OR         3       /* bool || bool */
#define COND_AND        4       /* bool && bool */
#define COND_XOR        5       /* bool ^ bool */
#define COND_EQ         6       /* bool == bool */
#define COND_NEQ        7       /* bool != bool */
#define COND_LAST       COND_NEQ
        uint32_t expr_type;
        uint32_t bool;
        struct cond_expr *next;
} cond_expr_t;
```

一个条件表达式可能包含若干 tunable/boolean 标识符以及运算符号。对于每一个标识符和运算符号，都使用一个 cond_expr_t 数据结构进行描述。
1，expr_type 域描述当前元素的属性，比如 COND_BOOL，相应地 bool 域保存该 tunable/boolean 的 policy value；
2，如果当前元素为元素符号，则 bool 域不被使用；

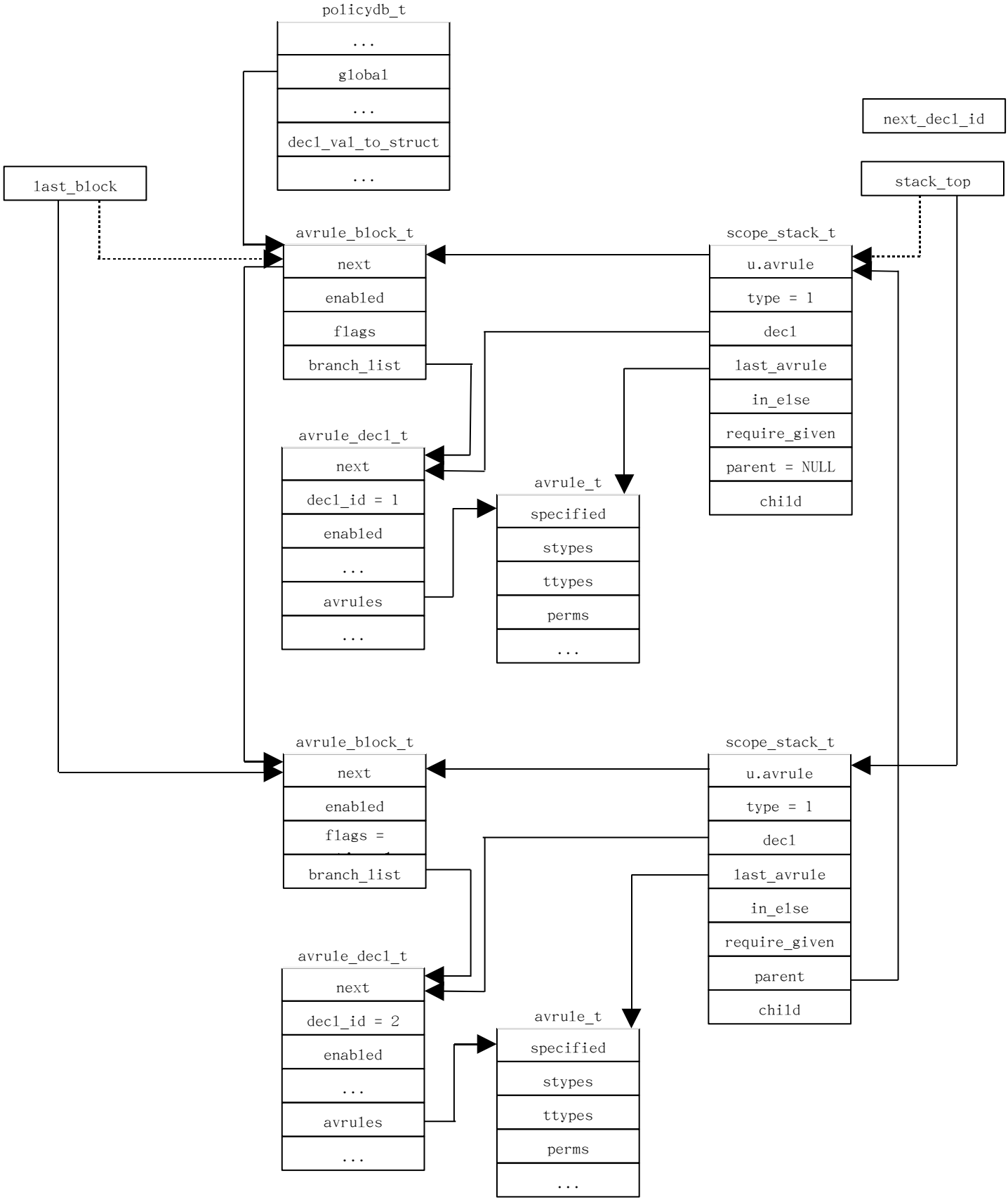
```
typedef struct avtab_node *avtab_ptr_t;

/*
 * Each cond_node_t contains a list of rules to be enabled/disabled
 * depending on the current value of the conditional expression. This
 * struct is for that list.
 */
typedef struct cond_av_list {
        avtab_ptr_t node;
```

```
        struct cond_av_list *next;
} cond_av_list_t;
```

如上文所述，true_list/false_list 队列中的每个元素都为指向 te_cond_avtab 哈希表中相应 avtab_node_t 节点的指针。在 find_avtab_node 函数中创建，参见下文。

12.4 module的编译 - checkmodule

12.4.1 编译过程核心数据结构关系图



12.4.2 define_policy - policy_module 词法分析

任何模块的.te 文件的第一个 statement 总是使用 policy_module 宏定义当前模块的名字和版本号，它定义在 policy/support/loadable_module.spt 文件中：

```
##############################
#
# For adding the module statement
#
define(`policy_module',`
        ifndef(`self_contained_policy',`
                module $1 $2;

                require {
                        role system_r;
                        all_kernel_class_perms

                        ifdef(`enable_mcs',`
                                dcl_sens(0,0)
                                dcl_cats(0,decr(mcs_num_cats))
                        ')

                        ifdef(`enable_mls',`
                                dcl_sens(0,decr(mls_num_sens))
                                dcl_cats(0,decr(mls_num_cats))
                        ')
                }
        ')
')
```

其中“self_contained_policy”变量在 Rules.modular 中定义，目前好像只针对 base 模块有效（base 模块必须是自包含的，没有外部依赖）。而对于非 base 模块，则该变量没有被定义，所以使用“module $1 $2”宏定义当前模块，并使用“require”申明该模块的外部依赖。

在 checkpolicy/policy_scan.l 文件中，就 module 关键字返回 MODULE token：

```
module|MODULE                   { return(MODULE); }
```

在 checkpolicy/policy_parser.y 文件中，MODULE token 的处理方法如下：

```
module_def              : MODULE identifier version_identifier ';'
                        { if (define_policy(pass, 1) == -1) return -1; }
```

所以，使用 define_policy 函数来处理 policy_module 声明：

```
int define_policy(int pass, int module_header_given)
{
        char *id;

        if (module_header_given) {
                if (policydbp->policy_type != POLICY_MOD) {
                        yyerror ("Module specification found while not building a policy module.\n");
                        return -1;
                }

                if (pass == 2) {
                        while ((id = queue_remove(id_queue)) != NULL)
```

```
                free(id);
        } else {
                id = (char *)queue_remove(id_queue);
                if (!id) {
                        yyerror("no module name");
                        return -1;
                }
                policydbp->name = id;
                if ((policydbp->version = queue_remove(id_queue)) == NULL) {
                        yyerror ("Expected a module version but none was found.");
                        return -1;
                }
        }
```

当编译 module 时，module_header_given 参数为 1，于是从 id_queue 中读取模块的名字和 version 信息，保存到 policydb_t 数据结构中。

（TODO：词法分析函数都有对 pass 数值的判断，如果为特定数值 1 或者 2，则会弹出 id_queue 队列中所有的 token。为什么这样做？谁？何时？确定传递特定的 pass 数值？）

```
        } else {
                if (policydbp->policy_type == POLICY_MOD) {
                        yyerror ("Building a policy module, but no module specification found.\n");
                        return -1;
                }
        }

        /* the first declaration within the global avrule block will always have an id of 1 */
        next_decl_id = 2;
```

一个模块内所有 avrule_decl_t 数据结构都有惟一的编号（从 1 开始）。由于在 policydb_init 函数中创建当前模块的第一个 avrule_block_t 和第一个 avrule_decl_t 时，已经设置第一个 avrule_decl_t.decl_id = 1，于是将 next_decl_id 设置为 2（在 link 过程中拷贝模块的 avrule_decl_t 数据结构到 base 模块时需要重新分配 decl_id，新旧数值的关系保存在该模块的 policy_module_t.avdecl_map[]数组中，参见下文）。

```
        /* reset the scoping stack */
        while (stack_top != NULL) {
                pop_stack();
        }
        if (push_stack(1, policydbp->global, policydbp->global->branch_list) == -1) {
                return -1;
        }
```

一个模块中所有 block 都有相应的 scope_stack_t 数据结构描述，它们组成一个栈，由 stack_top 全局变量指向栈顶。在开始编译一个新的模块时，如果栈非空则调用 pop_stack 函数释放栈内所有的 scope_stack_t 元素，然后将该模块的第一个 avrule_block_t 数据结构入栈。

最后，用 last_block 全局变量指向模块的第一个 avrule_block_t 数据结构。由下文可知，该指针用于组织 policydb_t.global 链表。

```
        last_block = policydbp->global;
        return 0;
}
```

模块的 avrule_block_t 以及相应的 scope_stack_t 数据结构之间的关系参见上图。

12.4.3 begin_optional - optional_policy 词法分析

由后文可见，optional_policy 宏展开后的关键字为“optional”，而在 checkpolicy/policy_parser.y 中指定使用 begin_optional 函数处理该关键字：

```
optional|OPTIONAL       { return(OPTIONAL); }

optional_dec1           : OPTIONAL
                        { if (begin_optional(pass) == -1) return -1; }
                        ;

int begin_optional(int pass)
{
        avrule_block_t *block = NULL;
        avrule_dec1_t *dec1;
        if (pass == 1) {
                /* allocate a new avrule block for this optional block */
                if ((block = avrule_block_create()) == NULL ||
                    (dec1 = avrule_dec1_create(next_dec1_id)) == NULL) {
                        goto cleanup;
                }
                block->flags |= AVRULE_OPTIONAL;
                block->branch_list = dec1;
                last_block->next = block;
```

首先为 optional block 定义新的 avrule_block_t 和 avrule_dec1_t 数据结构，以容纳描述该 optional block 内引用和规则的数据结构。注意将 block->flags 设置为 AVRULE_OPTIONAL，并根据 next_dec1_id 全局量确定 avrule_dec1_t.dec1_id 的数值，并且借助 last_block 全局指针构建当前模块的 policydb_t.global 链表。

```
        } else {
                /* select the next block from the chain built during pass 1 */
                block = last_block->next;
                assert(block != NULL && block->branch_list != NULL &&
                       block->branch_list->dec1_id == next_dec1_id);
                dec1 = block->branch_list;
        }
```

（TODO：暂时跳过该部分。为何 pass == 2 时为处理 optional block 中的 else block？）

```
        if (push_stack(1, block, dec1) == -1) {
                goto cleanup;
        }
        stack_top->last_avrule = NULL;
        last_block = block;
        next_dec1_id++;
        return 0;
```

最后通过 push_stack 函数，创建新的 scope_stack_t 元素将 optional block 的 block/dec1 数据结构加入当前模块的 scope stack。并更新 last_block 指针和 next_dec1_id 计数器。

```
      cleanup:
        yyerror("Out of memory!");
        avrule_block_destroy(block);
        return -1;
}
```

12.4.4 declare_type - type 标识符的定义

在 checkpolicy/policy_parser.y 中定义 type 标识符的定义方法如下：

```
type_def                : TYPE identifier alias_def opt_attr_list ';'
                        {if (define_type(1)) return -1;}
                        | TYPE identifier opt_attr_list ';'
                        {if (define_type(0)) return -1;}
                        ;

opt_attr_list           : ',' id_comma_list
                        |
                        ;

id_comma_list           : identifier
                        | id_comma_list ',' identifier
                        ;
```

其中 alias_def 为当前 type 的别名（alias），而 opt_attr_list 为当前 type 所属的若干属性。如果定义了 alias 则传递给 define_type 的参数为 1；否则为 0。而在 type 标识符定义中总是可以声明其所在的属性列表。注意 id_comma_list 采用了递归定义：要么为单个标识符，或者为以逗号间隔的若干标识符；而 opt_attr_list 以逗号开始，后接 id_comma_list。所以如果在定义 type 时同时指定其所属的 attribute，则在 type 定义后应该有一个逗号，否则为语法错误。比如：

```
        type selinux_config_t , semanage_store_t;
```

```
int define_type(int alias)
{
        char *id;
        type_datum_t *datum, *attr;

        if (pass == 2) {
```

（TODO：暂且跳过这里，为什么处理 type bounds 时 pass == 2?）

```
                /*
                 * If type name contains ".", we have to define boundary
                 * relationship implicitly to keep compatibility with
                 * old name based hierarchy.
                 */
                if ((id = queue_remove(id_queue))) {
                        char *bounds, *delim;

                        if ((delim = strrchr(id, '.')) && (bounds = strdup(id))) {
                                bounds[(size_t)(delim - id)] = '\0';

                                if (define_typebounds_helper(bounds, id))
                                        return -1;
                                free(bounds);
                        }
                        free(id);
                }
```

如果 type 标识符中含有“.”，比如 some_type.suffix，则当前 type（即 some_type.suffix）从属于 some_type，其能力是 some_type 的一个子集。所以用 bounds 字符串表示父 type，而 id 表示子 type，然后调用 define_typebounds_helper 函数，设置 type.bounds = bounds.s.value。

```
        if (alias) {
                while ((id = queue_remove(id_queue)))
                        free(id);
        }

        while ((id = queue_remove(id_queue)))
                free(id);
        return 0;
}

if ((datum = declare_type(TRUE, FALSE)) == NULL) {     # primary == TRUE, isattr == FALSE
        return -1;
}
```

然后调用 declare_type 函数完成新定义的 type 标识符的所有处理工作，参见下文。注意这里为 type 的第一次定义，所以才调用 declare_type 函数。与之相比在 define_typeattribute 函数中，由于要求 type 已经被定义或者声明过，因此直接在当前模块的 scope[SYM_TYPES]和 p_types 符号表中查找，如果找不到则报错。

```
if (alias) {
        if (add_aliases_to_type(datum) == -1) {
                return -1;
        }
}
```

如果定义 type 标识符时同时指定了其 alias，则调用 add_aliases_to_type 函数处理。为 alias 标识符创建 type_datum_t 和 scope_datum_t 数据结构并插入相应的符号表，惟一的不同是不为 alias 分配单独的 policy value，而是复用相应 type 的 policy value（体会，只有这样才满足 alias 的定义！）。

处理完 type 规则中的 type/alias 定义后，最后处理属性列表。

```
while ((id = queue_remove(id_queue))) {
        if (!is_id_in_scope(SYM_TYPES, id)) {
                yyerror2("attribute %s is not within scope", id);
                free(id);
                return -1;
        }
```

通常情况下在 type 规则中指定的属性都是在同一个模块中定义的，如果不是事先也应该用 require 声明为外部依赖。所以首先检查属性标识符是否已经在当前模块相应的 scope 符号表中注册过。通常都应该注册过，属性标识符的 scope_datum_t.decl_ids[]数组描述定义或引用该属性的 block/decl 的 decl_id，进而在 stack_top 栈中进一步查找是否存在这样的 block/decl。如果没有找到则返回 0。

```
        attr = hashtab_search(policydbp->p_types.table, id);
        if (!attr) {
                /* treat it as a fatal error */
                yyerror2("attribute %s is not declared", id);
                return -1;
        }
```

这段代码假设 type 属性的定义或声明，应该放到当前模块的 global block 中。如果在当前模块的 p_types 符号表中查找失败，则认为是一个致命的错误。

```
        if (attr->flavor != TYPE_ATTRIB) {
                yyerror2("%s is a type, not an attribute", id);
```

```
                return -1;
        }
```

进而从当前模块的 p_types 符号表中查询该属性的 type_datum_t 数据结构，并进一步核实其 flavor 为 TYPE_ATTRIB。

```
        if ((attr = get_local_type(id, attr->s.value, 1)) == NULL) {
                yyerror("Out of memory!");
                return -1;
        }
```

由下文可见，get_local_type 函数获得当前 type/typeattribute 规则所在 block（未必是 global block，还有可能是一个 optional block）的 symtab[SYM_TYPES]符号表中相应 type 属性的 type_datum_t，这样下面就可以把 type-attribute 关系记录到这个 type_datum_t 了。

即：type 属性的定义或者声明在 global block 中，但是相应的 type/typeattribute 规则却可能在一个 optional block 中，所以应该把 type-attribute 关系设置到**当前规则所在 block** 的符号表中！这样假如该 optional block 无效，导致当前 type 规则中定义的 type 无效，相关 type-attribute 关系也不会遗留到 global block 的符号表中。

另外，在 expand 过程的最后，会针对 base 模块所有 block 的当前有效的 dec1 的私有符号表调用 attr_convert_callback 函数，将可能记录在 block/dec1 的私有符号表中的 type-attribute 关系拷贝到 out.p_types 符号表中。只有这样才能和 get_local_type 函数的行为相互呼应，否则使用 optional_policy 宏的 block/dec1 中定义的 type-attribute 关系必定被遗漏！！

```
        if (ebitmap_set_bit(&attr->types, datum->s.value - 1, TRUE)) {
                yyerror("Out of memory");
                return -1;
        }
    }
```

最后以 type 的(policy value - 1)为索引，设置属性的 type_datum_t.types 位图的相应位（该位图描述了属于该属性的所有 type）。

```
    return 0;
}
```

get_local_type 函数被 define_type 和 define_typeattribute 函数调用。由于相应的 type/typeattribute 规则可能在 global block 中，也可能在一个 optional block 中（非 else 分支上）。而 optional block 是否被使能取决于其外部依赖能否被满足，如果无法满足，则相应规则所声明的 type-attribute 关系也应该无效（即规则本身不被使能）。所以才需要调用 get_local_type 函数获得 type 属性在当前 block（即规则所在 block）的符号表中的副本，然后设置其中的 types 位图。这样可以避免相应 type/typeattribute 规则无效时还把该 type-attribute 关系记录到当前模块 p_types 符号表中（最终导致在 expand 属性时会失败！）

```
/* Return a type_datum_t for the local avrule_decl with the given ID.
 * If it does not exist, create one with the same value as 'value'.
 * This function assumes that the ID is within scope.  c.f.,
 * is_id_in_scope().
 *
 * NOTE: this function usurps ownership of id afterwards.  The caller
 * shall not reference it nor free() it afterwards.
 */
```

```
type_datum_t *get_local_type(char *id, uint32_t value, unsigned char isattr)
{
        type_datum_t *dest_typdatum;
        hashtab_t types_tab;

        assert(stack_top->type == 1);
```

只有 avrule block 才能定义标识符、声明外部依赖，而在 conditional block 中禁止（因为它是否生效受到相应 boolean 的控制），所以首先使用 assert 宏来检查当前 type 或 typeattribute 规则是否不再 conditional block 中。

```
        if (stack_top->parent == NULL) {
                /* in global, so use global symbol table */
                types_tab = policydbp->p_types.table;
        } else {
                types_tab = stack_top->decl->p_types.table;
        }
```

从和当前 block 对应的 scope_stack_t 数据结构（由 stack_top 指向）中即可判断当前 block 是否为 global block，亦或一个 optional block。显然对于 global block，下面需要在当前模块的 p_types 符号表中查找 type 属性的 type_datum_t；而对于 optional block，则应该在该 block 内部的 avrule_decl_t.symtab[SYM_TYPES]符号表中查找。

```
        dest_typdatum = hashtab_search(types_tab, id);
        if (!dest_typdatum) {
                dest_typdatum = (type_datum_t *) malloc(sizeof(type_datum_t));
                if (dest_typdatum == NULL) {
                        free(id);
                        return NULL;
                }
                type_datum_init(dest_typdatum);
                dest_typdatum->s.value = value;
                dest_typdatum->flavor = isattr ? TYPE_ATTRIB : TYPE_TYPE;
                dest_typdatum->primary = 1;
                if (hashtab_insert(types_tab, id, dest_typdatum)) {
                        free(id);
                        type_datum_destroy(dest_typdatum);
                        free(dest_typdatum);
                        return NULL;
                }
```

如果当前 block 的 symtab[SYM_TYPES]符号表中没有相应属性的 type_datum_t，则复制其 policy value 和 flavor 等信息并插入。

```
        } else {
                free(id);
                if (dest_typdatum->flavor != isattr ? TYPE_ATTRIB : TYPE_TYPE) {
                        return NULL;
                }
        }
```

否则，只需要检查是否存在 flavor 的冲突即可。

```
        return dest_typdatum;
}
```

declare_type 其实为 declare_symbol 的封装函数，在为 type 标识符分配了 type_datum_t 数据结构之后，调用后者完成所有后继处理操作：

```
[define_type > declare_type]

type_datum_t *declare_type(unsigned char primary, unsigned char isattr)
{
        char *id;
        type_datum_t *typdatum;
        int retval;
        uint32_t value = 0;

        id = (char *)queue_remove(id_queue);
        if (!id) {
                yyerror("no type/attribute name?");
                return NULL;
        }

        if (strcmp(id, "self") == 0) {
                yyerror("'self' is a reserved type name and may not be declared.");
                free(id);
                return NULL;
        }
```

self 为保留关键字，因此不能作为某个 type 的名称字符串。

```
        typdatum = (type_datum_t *)malloc(sizeof(type_datum_t));
        if (!typdatum) {
                yyerror("Out of memory!");
                free(id);
                return NULL;
        }
        type_datum_init(typdatum);
        typdatum->primary = primary;
        typdatum->flavor = isattr ? TYPE_ATTRIB : TYPE_TYPE;
```

给当前 type 标识符创建 type_datum_t 数据结构，并初始化。在 define_type > declare_type 时，参数 primary == TRUE，isattr == FALSE。所以对于非属性的 type 而言，flavor == TYPE_TYPE；

接着就可以调用 declare_symbol 函数完成如下操作：
1，将当前 type 标识符的名称字符串及其 type_datum_t 数据结构插入当前模块的 p_types 符号表；
2，为当前 type 基于 p_types.nprim 分配 policy value；
3，为当前 type 查询或创建 scope_datum_t 数据结构并插入当前模块的相应 scope 符号表；
4，以当前 type 的(policy value - 1)为索引，设置当前 block/decl 的 avrule_decl_t.declared 中的相应位图。

```
        retval = declare_symbol(SYM_TYPES, id, typdatum, &value, &value);
        if (retval == 0 || retval == 1) {
                if (typdatum->primary) {
                        typdatum->s.value = value;
                }
```

declare_symbol 函数返回 0 表示向 p_types 符号表插入成功；返回 1 表示相应哈希表元素已经存在。而返回参数 value 即为给当前 type 标识符分配的 policy value，在 type_datum_t.primary 不为 0 时将 policy value 保存在 type_datum_t.s.value 中（对于 alias，如果 flavor == TYPE_TYPE，则其 primary == 0；如果 flavor == TYPE_ALIAS，则其 primary 中保存相关 type 的 policy value）。

（返回值为 1 时应该释放调用者中分配的、typdatum 所指向的 type_datum_t 数据结构？）

最后处理 declare_symbol > symtab_insert 返回的错误码：

```
        } else {
                /* error occurred (can't have duplicate type declarations) */
                free(id);
                type_datum_destroy(typdatum);
                free(typdatum);
        }
        switch (retval) {
        case -3:{
                        yyerror("Out of memory!");
                        return NULL;
                }
        case -2:{
                        yyerror2("duplicate declaration of type/attribute");
                        return NULL;
                }
        case -1:{
                        yyerror("could not declare type/attribute here");
                        return NULL;
                }
        case 0:
        case 1:{
                        return typdatum;
```

在没有错误时返回已经处理过的 type_datum_t 数据结构指针。

```
                }
        default:{
                        assert(0);      /* should never get here */
                }
        }
}
```

declare_symbol 函数将标识符的(key，datum)二元组插入相应的符号表，查询或创建标识符的 scope_datum_t 数据结构并插入当前模块的相应 scope 符号表，最后设置 avrule_decl_t.declared 数据结构的相应 scope 位图。参数 dest_value 和 datum_value 返回给相应标识符分配的 policy value：

```
[define_type > declare_type > declare_symbol]

/* Attempt to declare a symbol within the current declaration.  If
 * currently within a non-conditional and in a non-else branch then
 * insert the symbol, return 0 on success if symbol was undeclared.
 * For roles and users, it is legal to have multiple declarations; as
 * such return 1 to indicate that caller must free() the datum because
 * it was not added.  If symbols may not be declared here return -1.
 * For duplicate declarations return -2.  For all else, including out
 * of memory, return -3.  Note that dest_value and datum_value might
 * not be restricted pointers. */
int declare_symbol(uint32_t symbol_type, hashtab_key_t key, hashtab_datum_t datum,
                   uint32_t * dest_value, uint32_t * datum_value)
{
        avrule_decl_t *decl = stack_top->decl;
        int retval;
```

新定义的标识符总是针对当前 block/decl，即由 stack_top 指向的 scope_stack_t 数据结构的 decl 域指向。

```
        /* first check that symbols may be declared here */
        if (!is_declaration_allowed()) {
                return -1;
        }
```

正如注释中所述，在 conditional block（tunable_policy）或者 optional block 的 else 分支中不允许定义新的标识符。调用 is_declaration_allowed 函数检查当前 block 相应的 scope_stack_t（即由 stack_top 指向）中的 type 和 in_else 标志位。

然后调用 symtab_insert 函数完成如下操作：
1，将当前标识符的（key，datum）插入当前模块的相应 p_xxx 符号表；
2，如果插入成功，则为当前标识符基于相应 p_xxx.nprim 分配 policy value，如果已存在则直接返回 1；
3，在当前模块的相应 scope 符号表中查询是否存在该标识符的 scope_datum_t 数据结构，如果没有则插入，并将当前 block/decl 的 decl_id 记录到 scope_datum_t.decl_ids[]数组中；如果有则检查当前的定义或引用是否和先前的定义或引用冲突；

```
        retval = symtab_insert(policydbp, symbol_type, key, datum, SCOPE_DECL, decl->decl_id, dest_value);
        if (retval == 1 && dest_value) {
                symtab_datum_t *s =
                    (symtab_datum_t *)hashtab_search(policydbp->symtab[symbol_type].table, key);
                assert(s != NULL);

                if (symbol_type == SYM_LEVELS) {
                        *dest_value = ((level_datum_t *)s)->level->sens;
                } else {
                        *dest_value = s->value;
                }
```

symtab_insert 函数返回 1 表示该标识符的(key，datum)已经插入相应的符号表，则返回已经存在的 xxx_datum_t 数据结构，得到已经分配的 policy value，由参数 dest_value 返回。

注意 declare_symbol > symtab_insert 时会继续传递参数 dest_value。当 declare_type > declare_symbol 时该参数不为 NULL，表示需要为当前标识符分配 policy value；否则则不需要，比如当 add_aliases_to_type > declare_symbol 时该参数为 NULL。

```
        } else if (retval == -2) {
                return -2;
        } else if (retval < 0) {
                return -3;
        } else {                /* fall through possible if retval is 0 */
        }

        if (datum_value != NULL) {
                if (ebitmap_set_bit(decl->declared.scope + symbol_type, *datum_value - 1, 1)) {
                        return -3;
                }
        }
        return retval;
}
```

最后，由于当前定义了一个新的标识符，因此在当前 avrule_decl_t 数据结构的 declared 域（而不是

required）的相应位图中（(declared.scope)[symbo_type]），以该标识符的(policy value - 1)为索引将相应位置位。由下文可见当前新定义标识符的 policy value 正是根据当前模块 policydb_t.p_xxx.nprim 来分配的。

symtab_insert 函数将某个标识符的(key, datum)二元组插入模块的相应符号表，为其分配 policy value，并为该标识符创建 scope_datum_t 数据结构，将定义或引用该标识符的 block/decl 的编号设置到 scope_datum_t.decl_ids[]数组内，最后插入模块的相应 scope 符号表。

[define_type > declare_type > declare_symbol > symtab_insert]

```
/* Declare a symbol for a certain avrule_block context.  Insert it
 * into a symbol table for a policy.  This function will handle
 * inserting the appropriate scope information in addition to
 * inserting the symbol into the hash table.
 *
 * arguments:
 *   policydb_t *pol       module policy to modify
 *   uint32_t sym          the symbole table for insertion (SYM_*)
 *   hashtab_key_t key     the key for the symbol - not cloned
 *   hashtab_datum_t data  the data for the symbol - not cloned
 *   scope                 scope of this symbol, either SCOPE_REQ or SCOPE_DECL
 *   avrule_decl_id        identifier for this symbol's encapsulating declaration
 *   value (out)           assigned value to the symbol (if value is not NULL)
 *
 * returns:
 *   0                     success
 *   1                     success, but symbol already existed as a requirement
 *                         (datum was not inserted and needs to be free()d)
 *   -1                    general error
 *   -2                    scope conflicted
 *   -ENOMEM               memory error
 *   error codes from hashtab_insert
 */
int symtab_insert(policydb_t * pol, uint32_t sym, hashtab_key_t key, hashtab_datum_t datum,
                  uint32_t scope, uint32_t avrule_decl_id, uint32_t * value)
{
        int rc, retval = 0;
        unsigned int i;
        scope_datum_t *scope_datum;

        /* check if the symbol is already there.  multiple
         * declarations of non-roles/non-users are illegal, but
         * multiple requires are allowed. */

        /* FIX ME - the failures after the hashtab_insert will leave
         * the policy in a inconsistent state. */
        rc = hashtab_insert(pol->symtab[sym].table, key, datum);
        if (rc == SEPOL_OK) {
                /* if no value is passed in the symbol is not primary (i.e. aliases) */
                if (value)
                        *value = ++pol->symtab[sym].nprim;
```

直接调用 hashtab_insert 函数将(key, datum)所指名称字符串及相应的 xxx_datum_t 数据结构加入**当前模块的全局符号表**。如果插入成功且需要为当前标识符分配 policy value，则基于相应符号表的 nprim 顺序分配。如果(key, datum)已经被注册过，则 hashtab_insert 函数返回 SEPOL_EEXIST。即，无论当前模块内有几个 block/decl 定义或者声明了该标识符，它在当前模块的全局符号表内只被注册一次。

注意，无论当前标识符是在当前模块的哪个 block 中被定义的（无论 unconditional block，或者 optional block 的非 else 分支），都义无反顾地注册到当前模块 unconditional block 的符号表（即该模块的全局符号表）中！同时，该标识符的 scope_datum_t.decl_ids[]数组记录模块内定义或者声明该标识符的 block/decl 的编号。

另外，对于 type-attribute 关系或者 role-attribute 关系，则记录到相应规则所在 block 的符号表中。注意这个符号表并不一定总是 unconditional block 的符号表，因为相应规则可能定义在一个 optional block 中。这样可以**确保当规则所在 block/decl 没有被使能时，模块的全局符号表内不包含无效的 type-attribute 或 role-attribute 关系**（参见 get_local_type 或 get_local_role 函数）。

由上文可见，add_aliases_to_type > declare_symbol > symtab_insert 时最终参数 value 为 NULL，表示不需要为 alias 再分配额外的 policy value（而复用相关 type 的 policy value）。

```
} else if (rc == SEPOL_EEXIST) {
        retval = 1;     /* symbol not added -- need to free() later */
} else {
        return rc;      # other errors
}
```

然后为该标识符分配 scope_datum_t 数据结构并加入当前模块的相应 scope 符号表:

```
/* get existing scope information; if there is not one then create it */
scope_datum = (scope_datum_t *)hashtab_search(pol->scope[sym].table, key);
if (scope_datum == NULL) {
        hashtab_key_t key2 = strdup((char *)key);
        if (!key2)
                return -ENOMEM;
        if ((scope_datum = malloc(sizeof(*scope_datum))) == NULL) {
                free(key2);
                return -ENOMEM;
        }
        scope_datum->scope = scope;
        scope_datum->decl_ids = NULL;
        scope_datum->decl_ids_len = 0;
        if ((rc = hashtab_insert(pol->scope[sym].table, key2, scope_datum)) != 0) {
                free(key2);
                free(scope_datum);
                return rc;
        }
```

首先在模块的相应 scope 符号表中查找是否已经存在相应的 scope_datum_t，因为当前模块可能已经定义了或者声明了对该标识符的外部依赖。如果没有则创建并插入。注意将当前 block/decl 的 decl_id 加入 scope_datum_t.decl_ids[]数组的操作在最后进行。即，一个模块内可能存在若干 block/decl 定义或者证明同一个标识符，但是它只有惟一的一个 scope_datum_t 数据结构，只要被许可，相应 block/decl 都记录到 scope_datum_t.decl_ids[]数组中。

如果相应 scope_datum_t 已经插入相应 scope 符号表，则表示当前模块已经出现了引用或定义该标识符的规则。所以现在可能的情况包括：重复定义，重复地声明外部依赖，先定义后声明外部依赖，先声明外部依赖后定义，下面分别处理各种情况。

注意只要没有出错退出或出现重复，最后还是会调用 add_i_to_a 函数将当前 block/decl 的 decl_id 加入该标识符的 scope_datum_t.decl_ids[]数组中。

```
} else if (scope_datum->scope == SCOPE_DECL && scope == SCOPE_DECL) {
        /* disallow multiple declarations for non-roles/users */
```

```
                if (sym != SYM_ROLES && sym != SYM_USERS) {
                        return -2;
                }
```

如果在一个模块中出现一个标识符的重复定义，则检查是否被许可。只有 role/user 才许可重复定义（即重复使用 role 或 user 规则指定一个 role 能够和哪些 type 组成合法的 SC，或者指定一个 user 能够扮演哪些 role），否则出错退出。

```
        } else if (scope_datum->scope == SCOPE_REQ && scope == SCOPE_DECL) {
                scope_datum->scope = SCOPE_DECL;
```

如果在模块中已经用 require 声明了对一个标识符的外部依赖，后来又自己定义了该标识符，则将该模块认为是该标识符的定义者，因此修改已有 scope_datum_t 的类型为 SCOPE_DECL。对于这种情况下面在检查重复时会直接返回。

```
        } else if (scope_datum->scope != scope) {
                /* This only happens in DECL then REQUIRE case, which is handled by caller */
                return -2;
        }
```

这种情况表示在模块中事先已经定义了一个标识符，后来又用 require 声明了对它的一个外部依赖。显然这种情况是非法的，应该报错退出。下文可见由调用者 require_symbol 处理这种情况。

注意，上面的分类处理放过了重复声明为外部依赖的情况。这种情况是合法的，下面会在重复检查时直接返回。

最后就可以调用 add_i_to_a 函数将模块内当前 block/decl 的 decl_id 加入该标识符的 scope_datum_t.decl_ids[]数组了。在实际加入前还必须避免出现重复。

```
        /* search through the pre-existing list to avoid adding duplicates */
        for (i = 0; i < scope_datum->decl_ids_len; i++) {
                if (scope_datum->decl_ids[i] == avrule_decl_id) {
                        /* already there, so don't modify its scope */
                        return retval;
                }
        }

        if (add_i_to_a(avrule_decl_id, &scope_datum->decl_ids_len, &scope_datum->decl_ids) == -1) {
                return -ENOMEM;
        }

        return retval;
}
```

add_i_to_a 函数调用 realloc 函数扩充 decl_ids[]数组，以容纳新元素（即为定义或引用该标识符的当前 block/decl 的 decl_id）。

一个标识符的 scope_datum_t.decl_ids[]数组可能存在如下可能:
1，包含多个定义者 ID;
2，包含多个使用者 ID;
3，前面多个元素为使用者 ID，后面多个元素为定义者 ID;
4，前面多个元素为使用者 ID，最后面一个元素为定义者 ID;
5，只包含一个定义者 ID;
6，只包含一个使用者 ID;

那么，对于 role/user 类标识符，decl_ids[]数组中可能的情况为：1，2，3，4，5，6；
而对于其他类标识符，decl_ids[]数组可能的情况为：2，4，5，6；

（TODO：link/expand 过程能否正确处理所有这些情况？）

12.4.5 require_type - 声明对 type 标识符的外部依赖

在模块的.te/.if 文件中可以使用 gen_require 声明当前 block/decl 所依赖的外部标识符及类型，比如：

```
gen_require(`
        type vlock_exec_t, vlock_t;
')
```

对于 type 标识符而言，最终使用 require_type 函数进行处理：

```
int require_type(int pass)
{
        return require_type_or_attribute(pass, 0);
}
```

require_type_or_attribute 函数在为声明为外部依赖的 type 标识符创建了 type_datum_t 后，调用 require_symbol > symtab_insert 函数，将其 type_datum_t 加入当前模块的 p_types 符号表并分配 policy value，并创建 scope_datum_t 并加入当前模块的 scope[SYM_TYPES]符号表。当前 block/decl 的 decl_id 记录在 scope_datum_t.decl_ids[]数组中。

```
static int require_type_or_attribute(int pass, unsigned char isattr)
{
        char *id = queue_remove(id_queue);
        type_datum_t *type = NULL;
        int retval;
        if (pass == 2) {
                free(id);
                return 0;
        }
        if (id == NULL) {
                yyerror("no type name");
                return -1;
        }
        if ((type = malloc(sizeof(*type))) == NULL) {
                free(id);
                yyerror("Out of memory!");
                return -1;
        }
        type_datum_init(type);
        type->primary = 1;
        type->flavor = isattr ? TYPE_ATTRIB : TYPE_TYPE;
        retval = require_symbol(SYM_TYPES, id, (hashtab_datum_t *)type, &type->s.value, &type->s.value);
        if (retval != 0) {
                free(id);
                free(type);
        }
        switch (retval) {
        case -3:{
                        yyerror("Out of memory!");
                        return -1;
                }
        case -2:{
```

```
                        yyerror("duplicate declaration of type/attribute");
                        return -1;
                }
        case -1:{
                        yyerror("could not require type/attribute here");
                        return -1;
                }
        case 0:{
                        return 0;
                }
        case 1:{
                        return 0;       /* type already required */
                }
        default:{
                        assert(0);      /* should never get here */
                }
        }
}

[require_type > require_type_or_attribute > require_symbol]

/* Attempt to require a symbol within the current scope.  If currently
 * within an optional (and not its else branch), add the symbol to the
 * required list.  Return 0 on success, 1 if caller needs to free()
 * datum.  If symbols may not be declared here return -1.  For duplicate
 * declarations return -2.  For all else, including out of memory,
 * return -3..  Note that dest_value and datum_value might not be
 * restricted pointers.
 */
int require_symbol(uint32_t symbol_type, hashtab_key_t key, hashtab_datum_t datum,
                   uint32_t * dest_value, uint32_t * datum_value)
{
        avrule_decl_t *decl = stack_top->decl;
        int retval;
```

和 declare_symbol 中一样，当前 block/decl 由 stack_top->decl 所指向。

```
        /* first check that symbols may be required here */
        if (!is_require_allowed()) {
                return -1;
        }
```

只有在 global/unconditional block，或者 optional block 的非 else 分支上才许可声明外部依赖。调用 is_require_allowed 函数进行检查。

上文介绍 declare_symbol > symtab_insert 时已经介绍过 symtab_insert 函数的功能，摘抄如下：
1，将当前标识符的（key，datum）插入当前模块的相应 p_xxx 符号表；
2，如果插入成功，则为当前标识符基于相应 p_xxx.nprim 分配 policy value，如果已经存在则直接返回 1；
3，在当前模块的相应 scope 符号表中查询是否存在该标识符的 scope_datum_t 数据结构，如果没有则插入，将当前 block/decl 的 decl_id 记录到 scope_datum_t.decl_ids[]数组中；如果有则检查当前的定义或外部依赖声明是否和先前的（定义或外部依赖声明）冲突；

注意，对于外部依赖的标识符，在当前模块中也为其分配并注册 xxx_datum_t 和 scope_datum_t 数据结构，并分配 policy value（如果没有 policy value 则无法编译当前模块内所有使用该外部标识符的规则）。

```
        retval = symtab_insert(policydbp, symbol_type, key, datum, SCOPE_REQ, decl->decl_id, dest_value);
```

```
if (retval == 1) {
        symtab_datum_t *s =
            (symtab_datum_t *)hashtab_search(policydbp->symtab[symbol_type].table, key);
        assert(s != NULL);

        if (symbol_type == SYM_LEVELS) {
                *dest_value = ((level_datum_t *)s)->level->sens;
        } else {
                *dest_value = s->value;
        }
```

symtab_insert 返回 1 则表示相应标识符已经注册过了，再次查找模块内相应的符号表以验证。

symtab_insert 返回 2 表示“scope conflicting”，即此前已经在该模块中定义过该标识符，而现在又用 require 声明为外部依赖的情况（参见上文）：

```
} else if (retval == -2) {
        /* ignore require statements if that symbol was
         * previously declared and is in current scope */
        int prev_declaration_ok = 0;
        if (is_id_in_scope(symbol_type, key)) {
                if (symbol_type == SYM_TYPES) {
                        /* check that previous symbol has same type/attribute-ness */
                        unsigned char new_isattr = ((type_datum_t *)datum)->flavor;
                        type_datum_t *old_datum =
                           (type_datum_t *)hashtab_search(policydbp->symtab[SYM_TYPES].table, key);
                        assert(old_datum != NULL);
                        unsigned char old_isattr = old_datum->flavor;
                        prev_declaration_ok = (old_isattr == new_isattr ? 1 : 0);
                } else {
                        prev_declaration_ok = 1;
                }
        }
```

再次验证该符号的 scope 是否在“当前模块范围内”，即当前模块是否已经定义或声明过该标识符（则当前模块的某个 block/decl 的 decl_id 已经被记录到标识符的 scope_datum_t.decl_ids[]数组中）。对于非 type/attribute 类标识符，直接设置 prev_declaration_ok == 1；对于 type/attribute 类标识符，还的检查先后两次定义/声明的 flavor 是否相同，只有相同时才能设置 prev_declaration_ok == 1（即认为当前的外部依赖声明没有使得事先的定义无效）。

```
        if (prev_declaration_ok) {
                /* ignore this require statement because it
                 * was already declared within my scope */
                stack_top->require_given = 1;
                return 1;
        } else {
                /* previous declaration was not in scope or
                 * had a mismatched type/attribute, so generate an error */
                return -2;
        }
```

如果 prev_declaration_ok == 1，则可以忽略当前的 require 声明并直接返回 1（认为当前模块是该标识符的定义者而不是引用者），否则返回-2。
（问题：为什么直接返回 1 前还需要设置当前 block/decl 的 scope_stack_t.require_given 标志位？）

```
} else if (retval < 0) {
        return -3;
```

```
        } else {                /* fall through possible if retval is 0 or 1 */
        }

        if (datum_value != NULL) {
                if (ebitmap_set_bit(decl->required.scope + symbol_type, *datum_value - 1, 1)) {
                        return -3;
                }
        }
        stack_top->require_given = 1;
        return retval;
}
```

最后，设置当前 block/decl 的 scope_stack_t.required 数据结构中的相应位图，并设置 require_given 标志位。

12.4.6 define_te_avtab - TE 规则的词法分析

在 policy_parser.y 中定义 TE 规则包含如下 5 个种类：

```
te_avtab_def            : allow_def
                        | auditallow_def
                        | auditdeny_def
                        | dontaudit_def
                        | neverallow_def
                        ;
```

而每种 TE 规则的语法结构及处理方法定义如下：

```
allow_def               : ALLOW names names ':' names names  ';'
                        {if (define_te_avtab(AVRULE_ALLOWED)) return -1; }
                        ;
auditallow_def          : AUDITALLOW names names ':' names names ';'
                        {if (define_te_avtab(AVRULE_AUDITALLOW)) return -1; }
                        ;
auditdeny_def           : AUDITDENY names names ':' names names ';'
                        {if (define_te_avtab(AVRULE_AUDITDENY)) return -1; }
                        ;
dontaudit_def           : DONTAUDIT names names ':' names names ';'
                        {if (define_te_avtab(AVRULE_DONTAUDIT)) return -1; }
                        ;
neverallow_def          : NEVERALLOW names names ':' names names  ';'
                        {if (define_te_avtab(AVRULE_NEVERALLOW)) return -1; }
```

这 5 种规则的语法格式均是：

规则类型 source_type target_type : target_class permissions ;

因此都可以使用 define_te_avtab 函数进行处理，其参数 which 即为 TE 规则的类型，而整条规则的 token 保存在 id_queue 中。

```
int define_te_avtab(int which)
{
        char *id;
        avrule_t *avrule;
        int i;

        if (pass == 1) {
                for (i = 0; i < 4; i++) {
                        while ((id = queue_remove(id_queue)))
```

```
                        free(id);
                }
                return 0;
        }
```

当 pass == 1 时，从 id_queue 中弹出当前规则的除规则类型 token 之外的其他 4 个 token。（TODO: pass 的作用？何时为 1 何时为 2 ？）

```
        if (define_te_avtab_helper(which, &avrule))
                return -1;
```

TE 规则的词法分析是由 define_te_avtab_helper 函数完成的，它创建一个 avrule 数据结构描述当前的 TE 规则（由参数 avrule 指针指向）。

```
        /* append this avrule to the end of the current rules list */
        append_avrule(avrule);
        return 0;
}
```

最后调用 append_avrule 函数将其注册到模块的当前 block/decl 的 avrule_decl_t.avrules 队列里，参见下文。

define_te_avtab_helper 函数创建一个 avrule_t 数据结构描述当前的 TE 规则，参数 which 为规则的类型，而当前 TE 规则的 token 保存在 id_queue 中。

```
int define_te_avtab_helper(int which, avrule_t ** rule)
{
        char *id;
        class_datum_t *cladatum;
        perm_datum_t *perdatum = NULL;
        class_perm_node_t *perms, *tail = NULL, *cur_perms = NULL;
        ebitmap_t tclasses;
        ebitmap_node_t *node;
        avrule_t *avrule;
        unsigned int i;
        int add = 1, ret = 0;
        int suppress = 0;

        avrule = (avrule_t *)malloc(sizeof(avrule_t));
        if (!avrule) {
                yyerror("memory error");
                ret = -1;
                goto out;
        }
        avrule_init(avrule);
        avrule->specified = which;
        avrule->line = policydb_lineno;
```

首先分配一个 avrule_t 数据结构并初始化。由此可见 specified 即为规则类型，而 line 为当前 policydb_lineno 的值。

```
        while ((id = queue_remove(id_queue))) {
                if (set_types(&avrule->stypes, id, &add, which == AVRULE_NEVERALLOW ? 1 : 0)) {
                        ret = -1;
                        goto out;
                }
        }
```

首先从 id_queue 中读取 source_type 字符串，将 avrule.stypes 位图中的相应位置位。注意，由于在该位置上可能使用扩展“{ }”，因此需要在循环中注意处理。并且使用 add 参数处理从一个属性中**去除**指定的 type 的情况，比如“file_type - shadow_t”。add 初始化为 1，则将“file_type”设置到 type_set_t.types 位图中；如果遇到“-”特殊字符，则 set_types 函数中将返回参数 add 设置为 0，则在处理下一个 token“shadow_t”时将它加入 type_set_t.negset 位图中。

set_types 函数检查读出的 source_type 是否在合法的 scope 中，然后读取当前模块的 policydb_t.p_types 符号表，查找以 source_type 为 key 的哈希表元素的 type_datum_t 数据结构（注意 TE 规则所使用的标识符必须已经定义过），以当前标识符的(policy value - 1)为索引，根据 add 的值设置 stypes.types 或者 negset 位图。

```
add = 1;
while ((id = queue_remove(id_queue))) {
        if (strcmp(id, "self") == 0) {
                free(id);
                avrule->flags |= RULE_SELF;
                continue;
        }
        if (set_types(&avrule->ttypes, id, &add, which == AVRULE_NEVERALLOW ? 1 : 0)) {
                ret = -1;
                goto out;
        }
}
```

然后使用类似方法处理 target_type 字符串，将 avrule_t.ttypes 位图中的相应位置置位。如果为“self”，则设置 avrule_t.flags 中的相应标志位。

注意，set_types 函数的参数 staralllowed 由当前规则的类型决定。由此可见，只有 neverallow 规则才许可在 source_type 和 target_type 域上使用特殊字符“*”或“~”。

下面读取规则中的 class 域。由于也可能使用扩展而指定多个 class，使用 tclasses 临时位图保存所有 class 的(policy value - 1)。

```
ebitmap_init(&tclasses);
while ((id = queue_remove(id_queue))) {
        if (!is_id_in_scope(SYM_CLASSES, id)) {
                yyerror2("class %s is not within scope", id);
                ret = -1;
                goto out;
        }
        cladatum = hashtab_search(policydbp->p_classes.table, id);
        if (!cladatum) {
                yyerror2("unknown class %s used in rule", id);
                ret = -1;
                goto out;
        }
        if (ebitmap_set_bit(&tclasses, cladatum->s.value - 1, TRUE)) {
                yyerror("Out of memory");
                ret = -1;
                goto out;
        }
        free(id);
}
```

首先读取 class 字符串，并查找当前 policydb_t.p_classes 符号表得到其 class_datum_t 数据结构。然

后以该 class 的(policy value - 1)为索引，将 tclasses 位图的相应位置位。

由于可以在语法成分中使用扩展（使用花括弧包含若干 type 或者 class），因此 stypes/ttypes 中的位图，以及 tclasses 位图中可能有若干位被置位。需要为每个 class 创建 class_perm_node_t 数据结构，用于描述当前规则所涉及的该 class 的那些权限位，所有这些数据结构将组织在 perms 指向的队列中：

```
perms = NULL;
ebitmap_for_each_bit(&tclasses, node, i) {
        if (!ebitmap_node_get_bit(node, i))
                continue;
```

ebitmap_for_each_bit 宏可以遍历位图中的所有位（无论置位与否），ebitmap_node_get_bit 用于取出位图中的某一位。如果没有被置位，则继续下一轮循环。

```
        cur_perms = (class_perm_node_t *)malloc(sizeof(class_perm_node_t));
        if (!cur_perms) {
                yyerror("out of memory");
                ret = -1;
                goto out;
        }
        class_perm_node_init(cur_perms);
        cur_perms->class = i + 1;                       # class's policy value
        if (!perms)
                perms = cur_perms;
        if (tail)
                tail->next = cur_perms;
        tail = cur_perms;
}
```

如果相应位被置位，则创建一个 class_perm_node_t 数据结构，并将其 class 设置为当前位置+1，即为某个 class 的 policy value。

注意，一个 class token 在规则中的先后位置，和其在位图中非 0 位的位置、以及相应 class_perm_node_t 元素在队列中的先后位置没有任何关系（因为先出现的 class token，其 policy value 不一定较小）。由于根据 tclasses 位图的非 0 位顺序地创建 class_perm_node_t 元素，因此一个 class 在 tclasses 位图中的非 0 位及其在 class_perm_node_t 队列中的相应元素一一对应，先后关系相同。

为规则中所有指定的 class 创建了 class_perm_node_t 数据结构之后，就可以继续处理规则的最后一个域（permission 或者 new_type）并且设置 class_perm_node_t.data。

注意，一个 permission 的 policy value 仅在其所在的 class 内有效，根据 class_datum_t.permissions 符号表或者相应 common_datum_t.permissions 符号表的 nprim 来定义。因此下面就需要就每一个 permission 字符串，得到其所在 class 的 class_datum_t，在相应 permissions 符号表中查找 perm_datum_t，从而获得当前 class 内该 permission 的 policy value。

```
while ((id = queue_remove(id_queue))) {                 # 就当前 permission token
        cur_perms = perms;                              # 从头遍历 perms 队列，并且同时
        ebitmap_for_each_bit(&tclasses, node, i) {      # 从头遍历 tclasses 位图
                if (!ebitmap_node_get_bit(node, i))
                        continue;
                cladatum = policydbp->class_val_to_struct[i];
```

最后处理 permission 扩展或 new_type。注意如果规则中 class 和 permission 部分都存在扩展，那么所有指定的 permission 应该对每一个指定的 class 都有效。每一个 while 循环（外层循环）处理一个

permission 标识符：就当前 permission 标识符遍历整个 class_perm_data_t 队列，以便为所有的 class 都设置 class_perm_node_t.data 位图中的相应位。注意 cur_perms 指针用于遍历该队列。

内层循环处理每一个指定的 class。之前 tclasses 位图已经记录了当前规则中所有指定的 class，用位图中被置位位置索引当前模块的 class_val_to_struct 数组，即得相应 class 的 class_datum_t 数据结构。注意 tclasses 位图和 class_val_to_struct 数组均是以 class 的(policy value - 1)为索引的。

由上文可见，根据 tclasses 位图中的非 0 位创建 class_perm_node_t 元素并顺序加入 perms 链表。所以 tclasses 位图中的非 0 位和 perms 链表中的元素一一对应，因此在内层循环中**同步**遍历这两个数据结构。

```
if (strcmp(id, "*") == 0) {
        /* set all permissions in the class */
        cur_perms->data = ~0U;
        goto next;
}
```

如果当前 permission 标识符为通配符“*”，则表示当前 class 的所有定义的权限都被包括。因此将 class_perm_node_t.data 的所有位置上都置位。

```
if (strcmp(id, "~") == 0) {
        /* complement the set */
        if (which == AVRULE_DONTAUDIT)
                yywarn("dontaudit rule with a ~?");
        cur_perms->data = ~cur_perms->data;
        goto next;
}
```

如果当前 permission 标识符为“~”，则表示**非**此前出现过的所有权限，所以将 class_perm_node_t.data 位图当前状态取反。

```
perdatum = hashtab_search(cladatum->permissions.table, id);
if (!perdatum) {
        if (cladatum->comdatum) {
              perdatum = hashtab_search(cladatum->comdatum->permissions.table,id);
        }
}
```

处理了“*”和“~”之后，就需要在当前 class 所定义/支持的 permissions 中查找当前 permission 了。一个 class 所定义（支持）的 permission 有两种定义方式：要么为 class 所专门定义，要么继承于某个 common。因此分别在 class_datum_t.permissions 和 class_datum_t.comdatum->permissions 符号表中查找。

```
if (!perdatum) {
        if (!suppress)
                yyerror2("permission %s is not defined for class %s", id,
                     policydbp->p_class_val_to_name[i]);
        continue;                       # 内层循环 continue，继续处理下一个 class
} else if (!is_perm_in_scope(id, policydbp->p_class_val_to_name[i])) {
        if (!suppress) {
                yyerror2("permission %s of class %s is not within scope", id,
                     policydbp->p_class_val_to_name[i]);
        }
        continue;                       # 内层循环 continue，继续处理下一个 class
} else {
        cur_perms->data |= 1U << (perdatum->s.value - 1);
}
```

如果无法找到相应的 permission_datum_t 或者 scope 检查失败，则报错并继续下一个循环处理后继 class。否则，将该 permission 的(policy value - 1)记录到当前 class 的 class_perm_data_t.data 位图的相应位上。

（TODO：我觉得如果出错，则不应该为 continue 而应该为 goto next；如果当前 permission 对当前 class 无效，则应该继续处理下一个 class，所以也应该步进 perms 队列的下一个元素）

由此可见，在当前的实现中每个 class 最多只支持 32 个 permission。

```
                next:
                  cur_perms = cur_perms->next;
        }  # 内层循环，处理每一个 class
```

每个 class 的 class_perm_node_t 元素在 perms 队列中出现的顺序和 class 出现的顺序相同，因此在开始新的内层循环处理后继 class 之前，步进 cur_perms 指针指向 perms 队列的下一个元素。

```
                free(id);

        } # 外层循环，处理每一个 permission
```

当 ebitmap_for_each_bit 宏遍历完 tclasses 的所有位后，内层循环结束。此时如果从 id_queue 中仍能读取字符串，即扩展部分中的下一个 permission，则开始新一轮外层循环。注意 cur_perms 指针重新指向 class_perm_node_t 队列首部并且在内层循环中重新遍历 tclasses 位图，以便为当前 permission 设置所有相关 class 的 class_perm_node_t.data 位图的相应位。

```
        ebitmap_destroy(&tclasses);

        avrule->perms = perms;
        *rule = avrule;
```

所有 class_perm_node_t.data 位图都设置完毕后，就可以释放 tclasses 临时变量中的位图数据结构了。最后将参数 rule 所指向的指针，指向新创建的 avrule_t 数据结构，注意其 perms 指针指向 class_perm_node_t 元素队列。

至此，当前 TE 规则就可以用 avrule_t 数据结构完整地描述了。

```
      out:
        return ret;
}
```

接下来就需要把 avrule_t 数据结构加入当前模块的当前 block 的当前 decl 数据结构的 avrules 队列中：

```
void append_avrule(avrule_t * avrule)
{
        avrule_decl_t *decl = stack_top->decl;

        /* currently avrules follow a completely different code path
         * for handling avrules and compute types
         * (define_cond_avrule_te_avtab, define_cond_compute_type);
         * therefore there ought never be a conditional on top of the
         * scope stack */
        assert(stack_top->type == 1);

        if (stack_top->last_avrule == NULL) {
                decl->avrules = avrule;
```

```
        } else {
                stack_top->last_avrule->next = avrule;
        }
        stack_top->last_avrule = avrule;
}
```

将 avrule_t 数据结构加入 avrule_decl_t.avrules 队列的末尾，并更新和当前 avrule_block_t 数据结构相应的 scope_stack_t 数据结构的 last_avrule 指针（它用于组织 avrule_decl_t.avrules 队列）。

12.4.7 define_role_trans - role_transition 规则的词法分析

在 checkpolicy 的 policy_parse.y 中指定 role_transition 规则的词法格式如下：

```
role_trans_def          : ROLE_TRANSITION names names identifier ';'
                        {if (define_role_trans(0)) return -1; }
                        | ROLE_TRANSITION names names ':' names identifier ';'
                        {if (define_role_trans(1)) return -1;}
                        ;
```

由此可见，如果 role_transition 规则中指定了 class，则传递给 define_role_trans 函数的参数为 1，否则为 0。注意规则中最后一个域 new_role 要求为单个 token，而其他域为“names”即可以存在扩展。

```
int define_role_trans(int class_specified)
{
        char *id;
        role_datum_t *role;
        role_set_t roles;
        type_set_t types;
        class_datum_t *cladatum;
        ebitmap_t e_types, e_roles, e_classes;
        ebitmap_node_t *tnode, *rnode, *cnode;
        struct role_trans *tr = NULL;
        struct role_trans_rule *rule = NULL;
        unsigned int i, j, k;
        int add = 1;
```

注意 role_set_t 数据结构就是在 ebitmap 的基础上，通过 flags 来描述特殊字符“*”和“~”。考虑了特殊字符后得到的位图由 e_roles 来描述。而 type_set_t 数据结构中包含两个位图 types/negset，以及描述特殊字符的 flags，所有因素考虑后得到的位图由 e_types 描述。参见后文 role/type_set_expand 函数。

```
        if (pass == 1) {
                while ((id = queue_remove(id_queue)))
                        free(id);
                while ((id = queue_remove(id_queue)))
                        free(id);
                if (class_specified)
                        while ((id = queue_remove(id_queue)))
                                free(id);
                id = queue_remove(id_queue);
                free(id);
                return 0;
        }
```

如果 pass == 1，则需要弹出 id_queue 中关于当前 role_transition 规则的所有 token。class_specified 参数标识规则中是否指定了 class 域。对于可能使用“{}”扩展的成分，需要在 while

循环中弹出所有可能的 token。注意 role_transition 规则的最有一个域为 new_role 部分，由于不允许存在扩展（只支持在规则中指定一个 token），因此在处理时无须使用 while 循环。

```
role_set_init(&roles);
ebitmap_init(&e_roles);
type_set_init(&types);
ebitmap_init(&e_types);
ebitmap_init(&e_classes);

while ((id = queue_remove(id_queue))) {
        if (set_roles(&roles, id))
                return -1;
}
```

首先调用 set_roles 函数处理规则中的 role 部分，该函数完成如下主要操作:
1，检查 id 字符串是否为特殊字符“*”或者“~”，如果是则报错返回（在 role_transition 规则中不允许使用这些特殊字符）；
2，调用 is_id_in_scope 函数，检查和 id 字符串对应的 scope_datum_t 是否在当前模块的 scope[SYM_ROLES]符号表中，如果没有则返回 1；如果有，则进一步检查当前模块中是否存在定义或者声明该标识符的 block/dec1，如果存在则返回成功，否则返回 0;
3，在当前模块的 p_roles 符号表中以 id 查找对应的 role_datum_t 数据结构，得到其 policy value;
4，以该 role 标识符的(policy value - 1)为索引，设置 role_set_t.roles 位图中的相应位;

```
add = 1;
while ((id = queue_remove(id_queue))) {
        if (set_types(&types, id, &add, 0))
                return -1;
}
```

接着调用 set_types 函数处理规则中的 type 部分，注意参数 add == 1，表示设置 type_set_t.types 位图（而不是 type_set_t.negset 位图）；参数 staralLowed == 0 表示在 role_transition 规则的 target_type 域部分不允许出现特殊字符“*”或“~”。该函数主要操作如下:
1，如果 id 为特殊字符，则根据参数 staralLowed 来处理。如果被许可，则设置 type_set_t.flags 中相应标志位；否则报错返回;
2，调用 is_id_in_scope 函数，在当前模块的 scope[SYM_TYPES]符号表中查找和 id 对应的 scope_datum_t 数据结构。检查方法同在 set_roles 中的相应描述;
3，查找当前模块的 p_types 符号表，得到和 id 对应的 type_datum_t 数据结构，得到其 policy value;
4，根据 add 参数为 1 或者 0，以该 type 标识符的(policy value - 1)为索引，设置 type_set_t.types 或者 negset 位图中的相应位;

注意，set_types 函数根据是否有“-”特殊字符来设置 add 返回参数，以处理类似{ file_type -shadow_t }的情况：在调用 set_types 函数之前 add == 1，则将 file_type 记录到 type_set_t.types 位图中；遇到“-”字符时改变 add == 0，那么在下一个循环中处理“shadow_t”时将其加入 type_set_t.negset 位图。

下面处理规则中可能指定的 class。由于规则中 class 位置上不允许使用特殊字符，因此直接使用 ebitmap 位图即可（而无须使用 type_set_t 或者 role_set_t 数据结构）。

```
if (class_specified) {
        while ((id = queue_remove(id_queue))) {
                if (!is_id_in_scope(SYM_CLASSES, id)) {
                        yyerror2("class %s is not within scope", id);
                        free(id);
                        return -1;
```

```
                }
                cladatum = hashtab_search(policydbp->p_classes.table, id);
                if (!cladatum) {
                        yyerror2("unknow class %s", id);
                        return -1;
                }
                ebitmap_set_bit(&e_classes, cladatum->s.value - 1, TRUE);
                free(id);
        }
```

如果在 role_transition 规则中使用了 class，则类似于 set_roles/set_types 函数中的处理，需要检查每个 class 标识符的 scope，并在当前模块的 p_classes 符号表中查找对应的 class_datum_t，最后以其(policy value - 1)为索引设置 e_classes 位图中的相应位。

```
} else {
        cladatum = hashtab_search(policydbp->p_classes.table, "process");
        if (!cladatum) {
                yyerror2("could not find process class for legacy role_transition statement");
                return -1;
        }

        ebitmap_set_bit(&e_classes, cladatum->s.value - 1, TRUE);
}
```

如果在规则中没有使用 class，则直接在 p_classes 符号表中查找“process”进程，以其(policy value - 1)为索引，设置 e_classes 位图。

体会：role_transition 规则中如果不指定 class 则默认为“process”class，正是在这里实现的！注意，由于规则描述数据结构 role_trans_rule_t 中 classes 的增加是一定的（不受 policy downgrade）的影响，因此无论实际规则中是否指定了 class 域，在词法分析程序中都应该正确地处理，正确地设置 role_trans_rule_t.classes 域。

最后处理规则的最后一个域 new_role：

```
id = (char *)queue_remove(id_queue);
if (!id) {
        yyerror("no new role in transition definition?");
        goto bad;
}
if (!is_id_in_scope(SYM_ROLES, id)) {
        yyerror2("role %s is not within scope", id);
        free(id);
        goto bad;
}
role = hashtab_search(policydbp->p_roles.table, id);
if (!role) {
        yyerror2("unknown role %s used in transition definition", id);
        goto bad;
}
```

由于该域不存在扩展，因此无须在循环中处理 token。还是首先检查 scope，并且在当前模块的 p_roles 符号表中查找对应的 role_datum_t 数据结构，从而将其 policy value 保存到 role_trans_rule_t.new_role 域中。

以模块方式编译时就当前 role_transition 规则创建其“字面”描述，即 role_trans_rule_t 数据结构，并加入当前模块当前 block/decl（由 stack_top->decl 指向）的 avrule_decl_t.role_tr_trans 队列。

但是正如注释所示，在这样做之前还必须将“字面”描述展开以便检查是否存在冲突的定义。

注意，此处展开的惟一目的是为了检查是否存在冲突的定义，因为按模块编译需要得到规则的“字面”描述而非“展开”描述。

```
/* This ebitmap business is just to ensure that there are not conflicting role_trans rules */
if (role_set_expand(&roles, &e_roles, policydbp, NULL))
        goto bad;
```

调用 role_set_expand 函数，将 role_set_t 数据结构展开为一个 ebitmap 位图：
1，如果 role_set_t.flags 中 ROLE_STAR 有效，表示涉及相关模块 p_roles 符号表中定以的所有 role 标识符，则将小于或等于 p_roles.nprim 的所有位置都设置到 e_roles 位图中，就可以直接退出了；
2，经由临时位图 mapped_roles，复制 roles.roles 位图到 e_roles 中，如果参数 map 不为 NULL，则经过该数组得到新的 policy value；
3，最后，如果 roles.flags 中 ROLE_COMP 有效，则将 e_roles 位图的内容“取反”。

```
if (type_set_expand(&types, &e_types, policydbp, 1))
        goto bad;
```

调用 type_set_expand 函数，将 type_set_t 扩展为一个 ebitmap 位图：
1，首先处理 type_set_t.types 位图。如果参数 alwaysexpand == 1，或者 type_set_t 中 negset 有效或者 flags 有效（表示包含特殊字符），则如果 type_set_t.types 中非 0 位表述的如果是一个 attribute，则需要把它展开（经由当前模块的 p_type_val_to_struct[]数组得到该属性的 type_datum_t，拷贝其中 type_datum_t.types 位图到临时位图 types 中）；如果 type_set_t.types 中非 0 位不是属性，则照搬其非 0 位置到临时位图 types 中。
如果上述三个条件都不满足，总是将 types 位图的非 0 位置复制到临时位图 types 中（而不关心它是否为属性）；
2，然后按照同样的逻辑处理 type_set_t.negset 位图，得到临时位图 neg_types；
3，处理 type_set_t.flags 中的 TYPE_STAR 标志位。如果该位有效，则说明当前模块 p_types 符号表中的所有 type 标识符，除非在 neg_types 位图中或者为属性，都应该设置到输出位图 e_types 中（对于通配符“*”显然不需要再考虑属性，只需要去除 neg_types 位图中的哪些位即可）并直接退出；
4，否则就 types 中的每一位，如果没有同时出现在 neg_types 位图中，则设置输出位图 e_types 中的相应位；
5，最后处理 type_set_t.flags 中的 TYPE_COMP 标志位。如果该位有效，则说明当前 p_types 符号表中所有 type 标识符，如果在输出位图 e_types 中相应位置为 0 **或者无法找到**（因为 p_types.nprim 可能远大于 e_types 位图当前长度），则设置 1；如果在 e_types 位图中已经为 1，则清 0。

```
ebitmap_for_each_bit(&e_roles, rnode, i) {
        if (!ebitmap_node_get_bit(rnode, i))
                continue;
        ebitmap_for_each_bit(&e_types, tnode, j) {
                if (!ebitmap_node_get_bit(tnode, j))
                        continue;
                ebitmap_for_each_bit(&e_classes, cnode, k) {
                        if (!ebitmap_node_get_bit(cnode, k))
                                continue;
```

此处三层循环为展开规则的“字面”描述的核心逻辑，即获得 role/type/class 位置上所有有效的组合(i，j，k)，和当前模块 policydb_t.role_tr 队列中 role_transition 规则的“展开”描述相比较：

```
                        for (tr = policydbp->role_tr; tr; tr = tr->next) {
                                if (tr->role == (i + 1) && tr->type == (j + 1) &&
                                    tr->tclass == (k + 1)) {
                                        yyerror2("duplicate role  transition for (%s,%s,%s)",
```

```
                                        role_val_to_name(i+1),
                                        policydbp->p_type_val_to_name[j],
                                        policydbp->p_class_val_to_name[k]);
                                goto bad;
                        }
                }
```

如果能找到和当前(i，j，k)相匹配的role_trans_t数据结构，则说明出现了重复或者冲突定义，报错返回（由此可见，就role_transition规则甚至不允许出现完全相同的重复定义）。

```
                tr = malloc(sizeof(struct role_trans));
                if (!tr) {
                        yyerror("out of memory");
                        return -1;
                }
                memset(tr, 0, sizeof(struct role_trans));
                tr->role = i + 1;
                tr->type = j + 1;
                tr->tclass = k + 1;
                tr->new_role = role->s.value;

                tr->next = policydbp->role_tr;
                policydbp->role_tr = tr;
```

否则说明没有冲突或重复的定义，则分配一个role_trans_t数据结构记录当前的(i，j，k)组合，并加入policydb_t.role_tr队列（保存当前的“展开”描述纯粹是为了和后继的相比较！）

```
                        }
                }
        }
```

在检查了没有冲突之后，就可以创建真正关心的role_trans_rule_t数据结构以保存role_transition规则的“字面”描述了：

```
        /* Now add the real rule */
        rule = malloc(sizeof(struct role_trans_rule));
        if (!rule) {
                yyerror("out of memory");
                return -1;
        }
        memset(rule, 0, sizeof(struct role_trans_rule));
        rule->roles = roles;
        rule->types = types;
        rule->classes = e_classes;
        rule->new_role = role->s.value;
```

注意这里是整个role_set_t/type_set_t/ebitmap/uint32_t数据结构的拷贝。

```
        append_role_trans(rule);
```

最后通过append_role_trans函数将该role_trans_rule_t数据结构加入当前模块当前block/decl（由stack_top->decl指向）的avrule_decl_t.role_tr_trans队列。

```
        ebitmap_destroy(&e_roles);
        ebitmap_destroy(&e_types);

        return 0;
```

```
      bad:
        return -1;
}
```

12.4.8 define_conditional - if-else conditional的词法分析

tunable_policy宏被展开为if-else conditional，其词法定义如下：

```
cond_stmt_def           : IF cond_expr '{' cond_pol_list '}' cond_else
                        { if (pass == 2) { if (define_conditional((cond_expr_t*)$2, (avrule_t*)$4,
(avrule_t*)$6) < 0) return -1;  }}
                        ;
cond_else               : ELSE '{' cond_pol_list '}'
                        { $$ = $3; }
                        | /* empty */
                        { $$ = NULL; }
```

由此可见，if关键词后为cond_expr，即条件表达式。而“{}”之间的规则用cond_pol_list来描述（无论if分支或else分支）。如果和上述模式匹配，则调用define_conditional函数，参数中$2即为模式中第2个成分cond_expr，$4即为if分支的cond_pol_list部分，而$6即为cond_else部分的返回值，由cond_else模式的定义可见，它即为else分支的cond_pol_list部分。

注意，有cond_stmt_def模式可见，if分支的规则（$4）将被记录在avtrue_list队列中，而else分支的规则（$6）将被记录在avfalse_list队列中。

条件表达式cond_expr的模式定义如下：

```
cond_expr               : '(' cond_expr ')'
                        { $$ = $2;}                                          # 去除小括号
                        | NOT cond_expr
                        { $$ = define_cond_expr(COND_NOT, $2, 0);   if ($$ == 0) return -1; }
                        | cond_expr AND cond_expr
                        { $$ = define_cond_expr(COND_AND, $1, $3);  if ($$ == 0) return -1; }
                        | cond_expr OR cond_expr
                        { $$ = define_cond_expr(COND_OR, $1, $3);   if ($$ == 0) return -1; }
                        | cond_expr XOR cond_expr
                        { $$ = define_cond_expr(COND_XOR, $1, $3);  if ($$ == 0) return -1; }
                        | cond_expr EQUALS cond_expr
                        { $$ = define_cond_expr(COND_EQ, $1, $3);   if ($$ == 0) return -1; }
                        | cond_expr NOTEQUAL cond_expr
                        { $$ = define_cond_expr(COND_NEQ, $1, $3);  if ($$ == 0) return -1; }
                        | cond_expr_prim
                        { $$ = $1; }
                        ;
cond_expr_prim          : identifier
                        { $$ = define_cond_expr(COND_BOOL,0, 0);    if ($$ == COND_ERR) return -1; }
                        ;
```

条件表达式中的每一个boolean/tunable标识符和每一个运算符，都使用一个cond_expr_t数据结构进行描述。根据是否为标识符，以及运算符关键字，给define_cond_expr函数传递相应的expr_type。

cond_pol_list模式用于描述if分支或else分支中的规则：

```
cond_pol_list           : cond_pol_list cond_rule_def
                        { $$ = define_cond_pol_list((avrule_t *)$1, (avrule_t *)$2); }
```

```
                        | /* empty */
                        { $$ = NULL; }
                        ;
```

define_cond_pol_list 函数即将当前新解析的 avrule_t 数据结构，加入当前 avtrue_list 或者 avfalse_list 的首部，然后返回新队列首部，做为 define_conditional 函数的$4 参数。

```
cond_rule_def           : cond_transition_def
                        { $$ = $1; }
                        | cond_te_avtab_def
                        { $$ = $1; }
                        | require_block
                        { $$ = NULL; }
                        ;
cond_te_avtab_def       : cond_allow_def
                          { $$ = $1; }
                        | cond_auditallow_def
                          { $$ = $1; }
                        | cond_auditdeny_def
                          { $$ = $1; }
                        | cond_dontaudit_def
                          { $$ = $1; }
                        ;
...
cond_allow_def          : ALLOW names names ':' names names  ';'
                        { $$ = define_cond_te_avtab(AVRULE_ALLOWED) ;
                          if ($$ == COND_ERR) return -1; }
                        ;
cond_auditallow_def     : AUDITALLOW names names ':' names names ';'
                        { $$ = define_cond_te_avtab(AVRULE_AUDITALLOW) ;
                          if ($$ == COND_ERR) return -1; }
                        ;
cond_auditdeny_def      : AUDITDENY names names ':' names names ';'
                        { $$ = define_cond_te_avtab(AVRULE_AUDITDENY) ;
                          if ($$ == COND_ERR) return -1; }
                        ;
cond_dontaudit_def      : DONTAUDIT names names ':' names names ';'
                        { $$ = define_cond_te_avtab(AVRULE_DONTAUDIT);
                          if ($$ == COND_ERR) return -1; }
                        ;
...
cond_transition_def     : TYPE_TRANSITION names names ':' names identifier filename ';'
                        { $$ = define_cond_filename_trans() ;
                          if ($$ == COND_ERR) return -1;}
                        | TYPE_TRANSITION names names ':' names identifier ';'
                        { $$ = define_cond_compute_type(AVRULE_TRANSITION) ;
                          if ($$ == COND_ERR) return -1;}
                        | TYPE_MEMBER names names ':' names identifier ';'
                        { $$ = define_cond_compute_type(AVRULE_MEMBER) ;
                          if ($$ ==  COND_ERR) return -1;}
                        | TYPE_CHANGE names names ':' names identifier ';'
                        { $$ = define_cond_compute_type(AVRULE_CHANGE) ;
                          if ($$ == COND_ERR) return -1;}
                        ;
```

由 cond_rule_def 模式的定义可见，在 if-else conditional（即 tunable_policy 宏中）所许可的规则只有 AVRULE_AV 和 AVRULE_TYPE 两类以及 require 关键字，而不包含任何 RBAC 规则或者 typeattribute 等规则。

cond_te_avtab_def 模式定义了所许可的 AVRULE_AV 规则的具体类型，它们都通过 define_cond_te_avtab 函数来解析整条规则，其参数即为规则的具体类型，由规则的关键字决定。

cond_transition_def 模式定义了所许可的 AVRULE_TYPE 规则的具体类型，它们都通过 define_cond_compute_type 函数来解析整条规则，其参数即为规则的具体类型，由规则的关键字决定。

处理 AVRULE_AV 类规则的 define_cond_te_avtab 函数和 define_te_avtab 函数都通过调用 define_te_avtab_helper 函数实现，二者只是返回值不同。处理 AVRULE_TYPE 类规则的 define_cond_compute_type 函数和 define_compute_type 函数也类似。

如前所述，define_conditional 函数用于创建一个描述整个 if-else 结构的 cond_node_t 数据结构。其参数分别为条件表达式描述队列，if 分支和 else 分支的规则队列。

该新创建的数据结构将被加入当前 dec1->cond_list 队列。如果能在该队列中找到 expr 完全匹配的已有元素，则直接将 if 分支和 else 分支的规则合并到已有元素相应队列的末尾。

```
int define_conditional(cond_expr_t * expr, avrule_t * t, avrule_t * f)
{
        cond_expr_t *e;
        int depth;
        cond_node_t cn, *cn_old;

        /* expression cannot be NULL */
        if (!expr) {
                yyerror("illegal conditional expression");
                return -1;
        }

        if (!t) {
                if (!f) {
                        /* empty is fine, destroy expression and return */
                        cond_expr_destroy(expr);
                        return 0;
                }
                /* Invert */
                t = f;
                f = 0;
                expr = define_cond_expr(COND_NOT, expr, 0);
                if (!expr) {
                        yyerror("unable to invert");
                        return -1;
                }
        }
```

如果 avtrue_list 队列为空，如果 avfalse_list 队列也为空，则释放条件表达式描述队列。如果 avfalse_list 队列此时非空，则交换这两个队列，并且在当前条件表达式描述队列的**末尾**，增加一个描述取反（COND_NOT）的 cond_expr_t 数据结构，从而将整个条件表达式取反。

（由此可见，条件表达式描述队列中 cond_expr_t 元素的顺序，和实际表达式的顺序**相反**。加入条件表达式为“a && b”，则描述队列中元素的先后顺序为“b”“&&”“a”）

接下来检查条件表达式中含有的 boolean/tunable 标识符的个数能否满足运算符的要求：

```
/* verify expression */
depth = -1;
for (e = expr; e; e = e->next) {
        switch (e->expr_type) {
        case COND_NOT:
                if (depth < 0) {
                        yyerror ("illegal conditional expression; Bad NOT");
                        return -1;
                }
                break;
        case COND_AND:
        case COND_OR:
        case COND_XOR:
        case COND_EQ:
        case COND_NEQ:
                if (depth < 1) {
                        yyerror ("illegal conditional expression; Bad binary op");
                        return -1;
                }
                depth--;
                break;
        case COND_BOOL:
                if (depth == (COND_EXPR_MAXDEPTH - 1)) {
                        yyerror ("conditional expression is like totally too deep");
                        return -1;
                }
                depth++;
                break;
        default:
                yyerror("illegal conditional expression");
                return -1;
        }
}
if (depth != 0) {
        yyerror("illegal conditional expression");
        return -1;
}
```

depth 初始化为-1，表示期望条件表达式中包含至少一个 boolean/tunable 标识符；在遇到一个 COND_BOOL cond_expr_t 数据结构（即出现了一个 boolean/tunable 标识符）后，depth 递增 1；在遇到任何一个双目运算符后，depth 递减 1（表示还需要一个 boolean/tunable 标识符）；在遇到单目运算符（COND_NOT）时，depth 数值不变。

如果最终 depth 结果非 0，则条件表达式非法。

```
/*  use tmp conditional node to partially build new node */
memset(&cn, 0, sizeof(cn));
cn.expr = expr;
cn.avtrue_list = t;
cn.avfalse_list = f;
```

经过检查后，将条件表达式描述队列和 if-else 规则队列，记录到一个 cond_node_t 数据结构中。

```
/* normalize/precompute expression */
if (cond_normalize_expr(policydbp, &cn) < 0) {
        yyerror("problem normalizing conditional expression");
        return -1;
}
```

然后，调用 cond_normalize_expr 函数设置 cond_node_t 数据结构中的 nbool 和 bool_ids[]数组，并初步计算当 boolean 个数不超过 5 时条件表达式的值，记录到 expr_pre_comp 位图中。

```
        /* get the existing conditional node, or create a new one */
        cn_old = get_current_cond_list(&cn);
        if (!cn_old) {
                return -1;
        }
```

get_current_cond_list 函数在当前 decl->cond_list 规列中寻找和 cn 相匹配的 cond_node_t 元素，返回其地址。如果不存在则创建新的 cond_node_t，并复制参数 cn 的表达式和 nbool，bool_ids，expr_pre_comp 内容，最后加入队列的首部并返回其地址。

这是因为一个 decl 中可能存在多个 if-else 结果，对于条件表达式相同的结果，则 if/else 分支可以合并。

```
        append_cond_list(&cn);
```

然后调用 append_cond_list 函数，进一步将 cn 的 avtrue_list/avfalse_list 队列，加入当前 decl->cond_list 相应元素的相应队列的末尾。

至此，cn 元素的使命就完成了，最后释放该数据结构。

```
        /* note that there is no check here for duplicate rules, nor
         * check that rule already exists in base -- that will be
         * handled during conditional expansion, in expand.c */

        cn.avtrue_list = NULL;
        cn.avfalse_list = NULL;
        cond_node_destroy(&cn);

        return 0;
}
```

cond_normalize_expr 函数尝试简化条件表达式，而且若含有的 boolean/tunable 变量的个数不超过 5 个，则计算其预估值。

```
/* precompute and simplify an expression if possible.  If left with !expression, change
 * to expression and switch t and f. precompute expression for expressions with limited
 * number of bools.
 */
int cond_normalize_expr(policydb_t * p, cond_node_t * cn)
{
        cond_expr_t *ne, *e;
        cond_av_list_t *tmp;
        unsigned int i, j, orig_value[COND_MAX_BOOLS];
        int k;
        uint32_t test = 0x0;
        avrule_t *tmp2;

        cn->nbools = 0;
        memset(cn->bool_ids, 0, sizeof(cn->bool_ids));
        cn->expr_pre_comp = 0x0;
```

初始化 cond_node_t 数据结构中上述域。

```
/* take care of !expr case */
ne = NULL;
e = cn->expr;
/* becuase it's RPN look at last element */
while (e->next != NULL) {
        ne = e;
        e = e->next;
}
```

首先简化取反逻辑。如上文所述，expr 队列中元素的顺序为条件表达式各元素的倒序。循环后，e 指向 expr 队列的末尾元素，ne 指向次末尾元素。

```
if (e->expr_type == COND_NOT) {
        if (ne) {
                ne->next = NULL;
        } else {        /* ne should never be NULL */
                printf ("Found expr with no bools and only a ! - this should never happen.\n");
                return -1;
        }
        /* swap the true and false lists */
        tmp = cn->true_list;
        cn->true_list = cn->false_list;
        cn->false_list = tmp;
        tmp2 = cn->avtrue_list;
        cn->avtrue_list = cn->avfalse_list;
        cn->avfalse_list = tmp2;

        /* free the "not" node in the list */
        free(e);
}
```

如果 expr 队列的末尾元素为 COND_NOT，则反转 avtrue_list/avfalse_list 和 true_list/false_list，并将末尾元素出列并删除。

```
/* find all the bools in the expression */
for (e = cn->expr; e != NULL; e = e->next) {
        switch (e->expr_type) {
        case COND_BOOL:
                i = 0;
                /* see if we've already seen this bool */
                if (!bool_present(e->bool, cn->bool_ids, cn->nbools)) {
                        /* count em all but only record up to COND_MAX_BOOLS */
                        if (cn->nbools < COND_MAX_BOOLS)
                                cn->bool_ids[cn->nbools++] = e->bool;
                        else
                                cn->nbools++;
                }
                break;
        default:
                break;
        }
}
```

遍历 expr 队列中的 boolean/tunable 元素，统计它们的个数到 nbool 域中，至多记录前 5 个 boolean/tunable 的 policy value 到 bool_ids[]数组中。bool_present 函数检查当前 boolean 是否已经记录到 bool_ids[]数组中，如果此时 nbools 已经大于 5，则直接返回 0（因为 bool_ids[]中至多只能记录 5 个 boolean）。

最后，如果 expr 队列中的 boolean 个数不超过 5，则计算 expr 的预估值。

```
        /* only precompute for exprs with <= COND_MAX_BOOLS */
        if (cn->nbools <= COND_MAX_BOOLS) {
                /* save the default values for the bools so we can play with them */
                for (i = 0; i < cn->nbools; i++) {
                        orig_value[i] = p->bool_val_to_struct[cn->bool_ids[i] - 1]->state;
                }
```

bool_ids[]数组中已经记录了 boolean 的 policy value，经过 policydb_t.bool_val_to_struct 数组得到其 cond_bool_datum_t 数据结构，state 域即为 boolean 的状态值。将 boolean 的状态值保存在 orig_value[]数组中，以便下面修改后恢复。

```
                /* loop through all possible combinations of values for bools in expression */
                for (test = 0x0; test < (0x1U << cn->nbools); test++) {
                        /* temporarily set the value for all the bools in the
                         * expression using the corr.  bit in test */
                        for (j = 0; j < cn->nbools; j++) {
                                p->bool_val_to_struct[cn->bool_ids[j] - 1]->state =
                                                                (test & (0x1 << j)) ? 1 : 0;
                        }
                        k = cond_evaluate_expr(p, cn->expr);
                        if (k == -1) {
                                printf ("While testing expression, expression result "
                                     "was undefined - this should never happen.\n");
                                return -1;
                        }
                        /* set the bit if expression evaluates true */
                        if (k)
                                cn->expr_pre_comp |= 0x1 << test;
                }
```

假设有 5 个 boolean，则一共有 32 种组合。外层 for 循环处理每一种组合。每种组合都需要给记录在 bool_ids[]中的 5 个 boolean 设置相应的状态值，然后调用 cond_evaluate_expr 函数计算当前组合下整个 expr 的结果（1 或 0），然后记录在 expr_pre_comp 位图的相应位中。

正是由于 expr_pre_comp 位图只有 32 为，所以 bool_ids[]数组中至多只能记录 5 个 boolean。

```
                /* restore bool default values */
                for (i = 0; i < cn->nbools; i++)
                        p->bool_val_to_struct[cn->bool_ids[i] - 1]->state = orig_value[i];
        }
```

计算完 expr_pre_comp 位图后，恢复所有 boolean 的状态值。

```
        return 0;
}
```

get_current_cond_list 函数在当前 decl 的 cond_list 规列中寻找和参数 cond_node_t 相匹配的 cond_node_t 元素，返回其地址。如果不存在则创建新的 cond_node_t，并加入队列的首部并返回其地址。

这是因为一个 decl 中可能存在多个 if-else 结果，对于条件表达式相同的结果，则 if/else 分支可以合并。

```
cond_list_t *get_current_cond_list(cond_list_t * cond)
{
        /* FIX ME: do something different here if in a nested conditional? */
        avrule_decl_t *decl = stack_top->decl;
        return get_decl_cond_list(policydbp, decl, cond);
}

/* Get a conditional node from a avrule_decl with the same expression.
 * If that expression does not exist then create one. */
cond_list_t *get_decl_cond_list(policydb_t * p, avrule_decl_t * decl, cond_list_t * cond)
{
        cond_list_t *result;
        int was_created;

        result = cond_node_find(p, cond, decl->cond_list, &was_created);
        if (result != NULL && was_created) {
                result->next = decl->cond_list;
                decl->cond_list = result;
        }
        return result;
}
```

核心函数为 cond_node_find，在 decl->cond_list 队列中寻找和参数 cond 具有相同 expr 的 cond_node_t 元素。如果找不到则创建，此时 was_created 返回参数被设置，将新创建的元素插入 decl->cond_list 队首。

```
/* Find a conditional (the needle) within a list of existing ones (the
 * haystack) that has a matching expression.  If found, return a
 * pointer to the existing node, setting 'was_created' to 0.
 * Otherwise create a new one and return it, setting 'was_created' to
 * 1. */
cond_node_t *cond_node_find(policydb_t * p, cond_node_t * needle, cond_node_t * haystack, int *was_created)
{
        while (haystack) {
                if (cond_expr_equal(needle, haystack)) {
                        *was_created = 0;
                        return haystack;
                }
                haystack = haystack->next;
        }
        *was_created = 1;

        return cond_node_create(p, needle);
}
```

cond_expr_equal 函数检查两个 expr 队列 a 和 b 是否完全相同，相同返回 1，否则为 0。

```
/*
 * Determine if two conditional expressions are equal.
 */
int cond_expr_equal(cond_node_t * a, cond_node_t * b)
{
        cond_expr_t *cur_a, *cur_b;

        if (a == NULL || b == NULL)
                return 0;
```

如果任意一个队列为空（或者两个都为空），则无法判断是否相同，所以返回假。

```
        if (a->nbools != b->nbools)
                return 0;
```

如果两个队列中 boolean 个数不同，则返回假。

```
        /* if exprs have <= COND_MAX_BOOLS we can check the precompute values for the expressions. */
        if (a->nbools <= COND_MAX_BOOLS && b->nbools <= COND_MAX_BOOLS) {
                if (!same_bools(a, b))
                        return 0;
                return (a->expr_pre_comp == b->expr_pre_comp);
        }
```

此时两个队列的 boolean 个数相同，如果都不超过 5，则首先判断含有的 boolean 是否相同。如果不相同，则返回假。如果相同，则可以比较 expr_pre_comp 位图是否相同而返回真或假。

至此，两个队列的 boolean 个数相同，但是个数超过 5，此时需要遍历两个 expr 队列，逐一检查各个元素是否相同：

```
        /* for long expressions we check for exactly the same expression */
        cur_a = a->expr;
        cur_b = b->expr;

        while (1) {
                if (cur_a == NULL && cur_b == NULL)
                        return 1;
```

如果同时到达两个队列的末尾，则说明此前的判断都为真（各个元素都相同），则说明整个队列都相同，所以返回真。

```
                else if (cur_a == NULL || cur_b == NULL)
                        return 0;
```

如果只有一个队列到达末尾，而另一个还有其他的操作符，则返回假。

```
                if (cur_a->expr_type != cur_b->expr_type)
                        return 0;
```

就当前的队列元素，如果类型不同，则返回假。

```
                if (cur_a->expr_type == COND_BOOL) {
                        if (cur_a->bool != cur_b->bool)
                                return 0;
                }
```

如果类型相同，对于 COND_BOOL 类型，则进一步检查是否为相同的 boolean。如果不是则返回假。

```
                cur_a = cur_a->next;
                cur_b = cur_b->next;
        }
        return 1;
}
```

cond_node_create 函数分配一个 cond_node_t 数据结构，并复制参数 cond_node_t 中的表达式和 bool_ids 等内容。

```
/* Create a new conditional node, optionally copying
```

```
 * the conditional expression from an existing node.
 * If node is NULL then a new node will be created
 * with no conditional expression.
 */
cond_node_t *cond_node_create(policydb_t * p, cond_node_t * node)
{
        cond_node_t *new_node;
        unsigned int i;

        new_node = (cond_node_t *)malloc(sizeof(cond_node_t));
        if (!new_node) {
                return NULL;
        }
        memset(new_node, 0, sizeof(cond_node_t));

        if (node) {
                new_node->expr = cond_copy_expr(node->expr);
                if (!new_node->expr) {
                        free(new_node);
                        return NULL;
                }
                new_node->cur_state = cond_evaluate_expr(p, new_node->expr);
                new_node->nbools = node->nbools;
                for (i = 0; i < min(node->nbools, COND_MAX_BOOLS); i++)
                        new_node->bool_ids[i] = node->bool_ids[i];
                new_node->expr_pre_comp = node->expr_pre_comp;
        }

        return new_node;
}
```

append_cond_list 函数将参数 cond 中的 avture_list/avfalse_list 队列，和当前 dec1->cond_list 队列中相应元素的相应队列相合并。

```
/* Append the new conditional node to the existing ones.  During
 * expansion the list will be reversed -- i.e., the last AV rule will
 * be the first one listed in the policy.  This matches the behavior
 * of the upstream compiler. */
void append_cond_list(cond_list_t * cond)
{
        cond_list_t *old_cond = get_current_cond_list(cond);
        avrule_t *tmp;

        assert(old_cond != NULL);       /* probably out of memory */
```

如上文所述，get_current_cond_list 函数要么从当前 dec1->cond_list 中返回和 cond 表达式相同的元素的地址，要么创建新的元素（并复制除 avtrue_list/true_list 等非队列元素），所以返回值 old_cond 一定不为 NULL。

然后，将参数 cond 的 avtrue_list 和 avfalse_list 队列，分别加入 old_cond 元素相应队列的**末尾**：

```
        if (old_cond->avtrue_list == NULL) {
                old_cond->avtrue_list = cond->avtrue_list;
        } else {
                for (tmp = old_cond->avtrue_list; tmp->next != NULL; tmp = tmp->next) ;
                tmp->next = cond->avtrue_list;
        }
```

```
        if (old_cond->avfalse_list == NULL) {
                old_cond->avfalse_list = cond->avfalse_list;
        } else {
                for (tmp = old_cond->avfalse_list; tmp->next != NULL; tmp = tmp->next) ;
                tmp->next = cond->avfalse_list;
        }
}
```

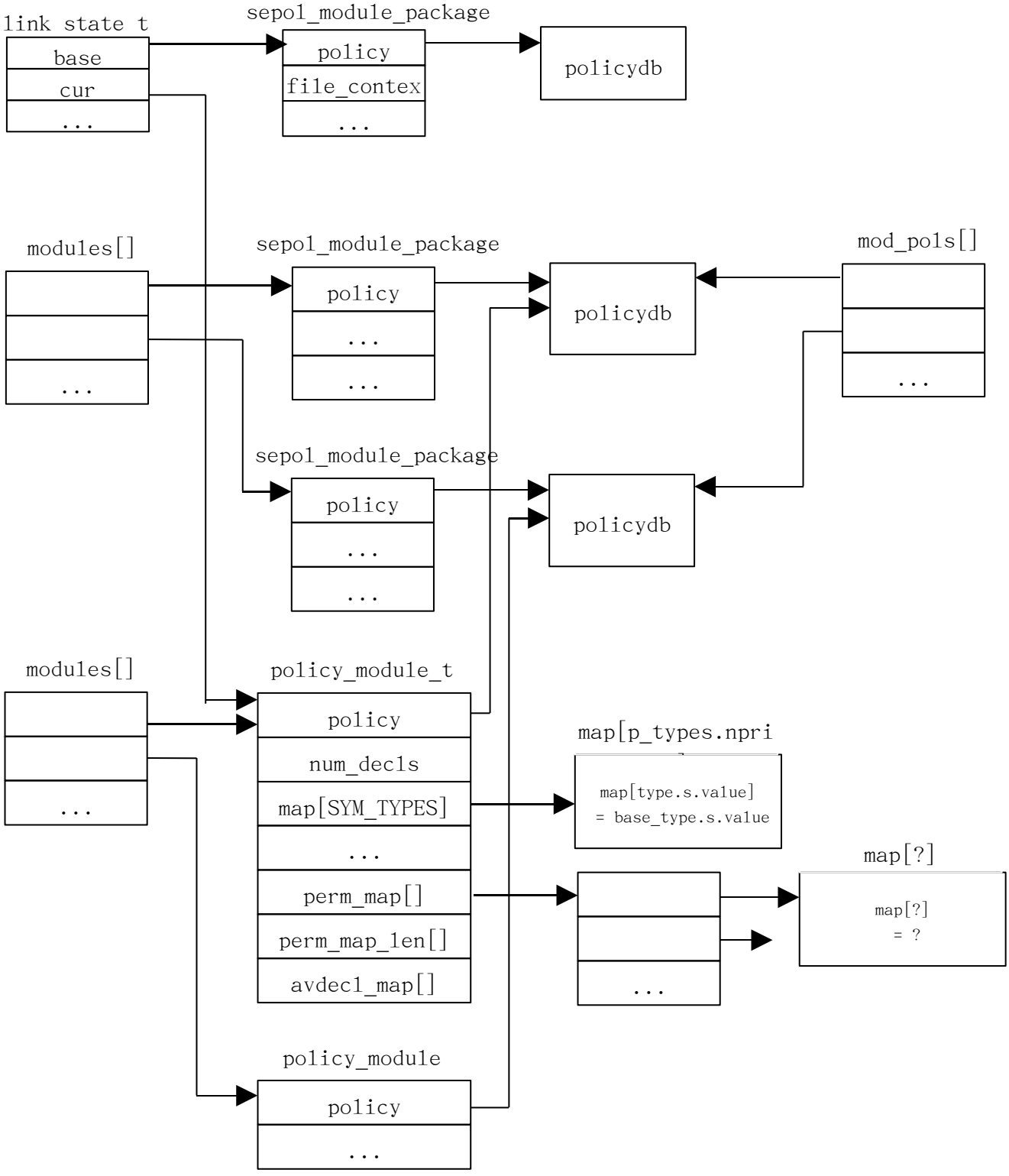
12.5 module的链接 - semodule_link

semodule_link命令可以手工将若干模块modpkg链接到basemodpkg上，其用法如下：

```
semodule_link [-Vv] [-o outfile] basemodpkg modpkg1 [modpkg2]...
```

其中参数均为通过checkmodule和semodule_package创建的二进制pp文件。semodule_link调用load_module函数读取并解析pp文件，用sepol_module_package_t数据结构描述每一个模块，并调用sepol_link_package函数完成link过程，即逐一将各个模块定义的符号表都合并到base模块的相应符号表中，并拷贝各个模块的所有block/decl数据结构（各个模块中定义的规则），并创建链接后的base.pp文件。

12.5.1 链接过程核心数据结构关系图


link state t
base
cur
...
sepol_module_package
policy
file_contex
...
policydb
modules[]
...
sepol_module_package
policy
...
...
policydb
mod_pols[]
...
sepol_module_package
policy
...
...
policydb
modules[]
...
policy_module_t
policy
num_decls
map[SYM_TYPES]
...
perm_map[]
perm_map_len[]
avdecl_map[]
map[p_types.npri
map[type.s.value]
= base_type.s.value
map[?]
map[?]
= ?
...
policy_module
policy
...

12.5.2 symtab 符号表的拷贝

12.5.2.1 p_types 符号表的拷贝

在模块内部 type 标识符的 policy value 根据 p_types.nprim 顺序分配，在将模块的 p_types 符号表合并到 base 模块的 p_types 符号表时，需要给模块的 type 重新分配 policy value（由下文可见从 base 模块的 p_types.nprim 的下一个数值开始分配），并且创建相应的 map[]数组建立 link 前后 policy value 的变化关系（以原有(policy value - 1)为索引，而数组元素即为合并到 base 模块后新分配的 policy value）。

注意，一个 type 只有一个定义者模块，但可以包含若干使用者模块。一个模块内使用的 type 标识符要么为自己定义的，要么声明为外部依赖。无论定义者或者使用者模块都会在自己的 p_types 符号表和 scope[SYM_TYPES]符号表中注册该 type 的 type_datum_t 和 scope_datum_t 数据结构。因此任何一个模块向 base 模块拷贝了该 type 之后，在拷贝其他模块的相同 type 时都会出现重复，此时只需要参照在 base 模块中的新 policy value，设置当前模块的 policy_module_t.map[SYM_TYPES][]数组中相应的转换关系即可，参见下文。

type_copy_callback 函数用于拷贝一个模块的 p_types 符号表，它在被 hashtab_map 函数调用时，传递的参数为每一个 type 标识符的 key（即标识符的名称字符串），type 的 type_datum_t 数据结构，以及上层 link_state_t 数据结构的指针（其 base 指针指向 base 模块的 policydb_t 数据结构）。

```
[semodule_link > semodule_link_package > link_modules > hashtab_map > type_copy_callback]

/* Copy types and attributes from a module into the base module. The
 * attributes are copied, but the types that make up this attribute
 * are delayed type_fix_callback(). */
static int type_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        int ret;
        char *id = key, *new_id = NULL;
        type_datum_t *type, *base_type, *new_type = NULL;
        link_state_t *state = (link_state_t *)data;

        type = (type_datum_t *)datum;
        if ((type->flavor == TYPE_TYPE && !type->primary) || type->flavor == TYPE_ALIAS) {
                /* aliases are handled later, in alias_copy_callback() */
                return 0;
        }
```

如果 type_datum_t.flavor == TYPE_ALIAS，或者尽管为 TYPE_TYPE 但是 primary == 0，则为别名。别名由 alias_copy_callback 函数处理，在此跳过。

```
        base_type = hashtab_search(state->base->p_types.table, id);
```

首先在 base 模块的 p_types 符号表中查找是否存在相同名称的 type 标识符。如果找到，则进一步检查是否存在冲突的属性定义（注意重复的定义在 scope_copy_callback 函数中处理，只许可 role/user 类型存在重复定义，此时其 scope 将被合并，type_datum_t.flags 将被合并）。

若 type_datum_t 已存在，则说明已经有该 type 标识符的其他使用者（定义或引用）模块被 link 到 base 模块中了。

```
        if (base_type != NULL) {
                /* type already exists.  check that it is what this
                 * module expected.  duplicate declarations (e.g., two
```

```
		 * modules both declare type foo_t) is checked during
		 * scope_copy_callback(). */
		if (type->flavor == TYPE_ATTRIB && base_type->flavor != TYPE_ATTRIB) {
			ERR(state->handle,
			    "%s: Expected %s to be an attribute, but it was already declared as a type.",
			    state->cur_mod_name, id);
			return -1;
		} else if (type->flavor != TYPE_ATTRIB && base_type->flavor == TYPE_ATTRIB) {
			ERR(state->handle,
			    "%s: Expected %s to be a type, but it was already declared as an attribute.",
			    state->cur_mod_name, id);
			return -1;
		}
		/* permissive should pass to the base type */
		base_type->flags |= (type->flags & TYPE_FLAGS_PERMISSIVE);
```

无论在当前 type 或者 base_type，只要二者之一为 attribute 且另外一个不是，则报错返回。如果都是普通 type，或者都是 attribute，则只需合并 type_datum_t.flags，下面就是在函数的末尾在 map[]数组中设置新旧 policy value 的关系了。

注意：
1，只有在 link 过程中才需要检查不同模块之间就同一个 type 是否存在冲突的 flavor 设置，而在 expand 过程中就不需要再考虑了。这也是 link.c 和 expand.c 中各自定义的 type_copy_callback 函数的主要区别之一；
2，正如注释所述，冲突定义的检查由 scope_copy_callback 函数完成；
3，还需要合并 type_datum_t.flags 中的 TYPE_FLAGS_PERMISSIVE 标志到 base 模块中（标记当前 domain 为 Permissive Domain）；

如果在 base 模块的 p_types 符号表中当前 type 标识符不存在，则需要将其插入 base 模块的 p_types 符号表。因此复制当前 type 的(key，datum)二元组，并拷贝相应的域：

```
	} else {
		if (state->verbose)
			INFO(state->handle, "copying type %s", id);

		if ((new_id = strdup(id)) == NULL) {
			goto cleanup;
		}

		if ((new_type = (type_datum_t *)calloc(1, sizeof(*new_type))) == NULL) {
			goto cleanup;
		}

		new_type->primary = type->primary;
		new_type->flags = type->flags;
		new_type->flavor = type->flavor;
		/* for attributes, the writing of new_type->types is done in type_fix_callback() */

		new_type->s.value = state->base->p_types.nprim + 1;
```

由此可见，new_type 的 policy value 也是基于 base 模块的 p_types.nprim 顺序分配（和模块的编译过程相同，参见上文 symtab_insert 函数），从而保证 base 模块中所有 type 都有惟一的 policy value。接下来就可以调用 hashtab_insert 函数将 new_type 插入 base.p_types 符号表了：

```
		ret = hashtab_insert(state->base->p_types.table,
				     (hashtab_key_t)new_id, (hashtab_datum_t)new_type);
```

```
        if (ret)
                goto cleanup;

        state->base->p_types.nprim++;
        base_type = new_type;
}
```

最后更新 base 模块的 p_types.nprim 数值。至此当前 type 已经被复制到 base 模块中，且 base_type 指针指向其在 base 模块中的副本。

```
if (state->dest_decl) {
        new_id = NULL;
        if ((new_type = calloc(1, sizeof(*new_type))) == NULL) {
                goto cleanup;
        }
        new_type->primary = type->primary;
        new_type->flavor = type->flavor;
        new_type->flags = type->flags;
        new_type->s.value = base_type->s.value;
        if ((new_id = strdup(id)) == NULL) {
                goto cleanup;
        }
        if (hashtab_insert(state->dest_decl->p_types.table, new_id, new_type)) {
                goto cleanup;
        }
        state->dest_decl->p_types.nprim++;
}
```

link_state_t.dest_decl 指针指向当前被复制的 block/decl 的 avrule_decl_t 数据结构。如果有效，则还需要进一步将当前 type 在 base 模块中的副本（因为使用 base_type->s.value，而其它域都相同），再次插入 dest_decl 指针所指向的 block/decl 的 symtab[SYM_TYPES]符号表，从而保证 base.p_types 符号表和被复制的 block/decl 的 symtab[SYM_TYPES]符号表中具有一致/相同的(id，type_datum_t)二元组。

注意：
1，copy_identifiers 函数中将 link_state_t.dest_decl 设置为参数 dest_decl，而只有在 link_modules > copy_module > copy_avrule_block > copy_avrule_decl > copy_identifier 调用链中 dest_decl 参数指向**新创建的**目的 avrule_decl_t 数据结构，**从而完整地复制原有 block/decl 的 type/user/role 类型符号表**！（注意，只有 type/user/role_copy_callback 函数中会检查 link_state_t.dest_decl 指针是否为空，且在不为空时复制当前标识符的 xxx_datum_t 数据结构并加入 dest_decl->p_xxx 符号表）

2，而在 link_modules > copy_identifiers 以及 copy_module > copy_identifiers 时均为 NULL，这两种情况都对应复制当前模块的 p_xxx 符号表到 base 的相应符号表，而不包括复制模块任意 block/decl 的符号表。

3，从当实现来看，对一个 block/decl 的符号表调用 xxx_copy_callback 函数时，**不仅会复制到新的 block/decl 的相应符号表中，而且也会复制到 base 模块的符号表中！**

所以一个重要的结论是：**当 link 过程结束后 base 模块的 p_xxx 符号表中就含有了所有模块所有 block 中定义的标识符！**

4，显然，在 expand.c 中定义的 type_copy_callback 函数不需要上述行为。

```
        state->cur->map[SYM_TYPES][type->s.value - 1] = base_type->s.value;
        return 0;

      cleanup:
        ERR(state->handle, "Out of memory!");
        free(new_id);
        free(new_type);
        return -1;
}
```

最后就是建立当前模块的 policy_module_t 数据结构的 map[SYM_TYPES][]数组了，以 type 在当前模块内的(policy value - 1)为索引，数组元素值为在 base 模块中新分配的 policy value。

注意：
1，link_state_t.cur 在 link_modules 函数中就某一模块通过 hashtab_map 函数调用 xxx_copy_callback 函数前已经指向该模块的 policy_module_t 数据结构。

2，无论拷贝当前模块的 p_xxx 符号表（即 global block 的符号表），还是其某个 block/decl 的符号表到 base 模块的 p_xxx 符号表时，都在当前模块的 policy_module_t.map[][]中记录新旧 policy value 的转换关系！

从下文可知，当前模块的 policy_module_t.map[SYM_TYPES][]数组用于**修正**模块的 role 被合并到 base 模块后 role_datum_t 中的 types 位图：能够和该 role 相关联的 type 的 policy value 已被重新分配，因此必须重新设置 type 在 role_datum_t.types 位图中的位置。

另外，被模块使用的任何一个标识符，要么被模块定义，要么被声明为外部引用。从 declare_symbol 和 require_symbol 来看，无论定义或引用，都会将该标识符注册到模块的 p_xxx 符号表以及相应的 scope 符号表中。模块的 policy_module_t.map[]数组总能描述该标识符在当前模块内和在 base 模块内 policy value 的变化关系。

假设有三个模块 A/B/C，A 模块定义一个 type，B/C 引用这个 type，假设该 type 在 A/B/C 模块中各自分配的 policy value 分别为 100/101/102。如果 B 模块首先向 base 模块 link，该 type 新的 policy value 为 1000，则 A/C 模块 link 时则会发现 type_datum_t 已经注册过。最终 A/B/C 模块中该 type 的新旧 policy value 转换关系如下：

```
A: policy_module_t.map[SYM_TYPES][100] = 1000 ;
B: policy_module_t.map[SYM_TYPES][101] = 1000 ;
C: policy_module_t.map[SYM_TYPES][102] = 1000 ;
```

12.5.2.2 所有其他标识符符号表的拷贝

在 libsepol/src/link.c 中定义所有类型标识符的拷贝函数如下：

```
static int (*copy_callback_f[SYM_NUM]) (hashtab_key_t key, hashtab_datum_t datum, void *datap) = {
        NULL, class_copy_callback, role_copy_callback, type_copy_callback,
        user_copy_callback, bool_copy_callback, sens_copy_callback, cat_copy_callback
};
```

注意，symtab[SYM_COMMONS]应该只能在 base 模块内定义，因此无须拷贝回调函数。

在 copy_identifiers 函数中以“表格驱动”方式逐一调用上述函数，以合并当前模块的某一符号表到 base 模块的相应符号表中，参见下文。该函数在 link_modules 函数中显式地调用完 type_copy_callback 函数后调用：

```
        copy_identifiers(&state, modules[i]->policy->symtab, NULL);
```

另外，由于 user/role/type_datum_t 数据结构中都直接或间接地包含 ebitmap 位图数据结构（比如描述扩展的 role_set_t 和 type_set_t 数据结构），用于描述 user-role，role-type，type-attribute 关系，位图所使用的 policy value 都是模块内确定的，因此还需要进一步按照相应标识符在 base 模块内新的 policy value 修正所有相关的 ebitmap。这个工作由相关 xxx_fix_callback 函数完成，在 copy_identifiers 函数的最后也以“表格驱动”方式被调用，参见下文。

```
static int (*fix_callback_f[SYM_NUM]) (hashtab_key_t key, hashtab_datum_t datum, void *datap) = {
        NULL, NULL, role_fix_callback, type_fix_callback, user_fix_callback, NULL, NULL, NULL
};
```

copy_identifiers 函数合并当前模块的所有类型符号表中的 xxx_datum_t 数据结构到 base 模块的相应类型符号表，并修正 xxx_datum_t 数据结构中使用的位图。

```
[semodule_link > semodule_link_package > link_modules > copy_identifiers]

static int copy_identifiers(link_state_t * state, symtab_t * src_symtab, avrule_decl_t * dest_decl)
{
        int i, ret;

        state->dest_decl = dest_decl;
```

如果调用 copy_identifiers 函数时参数 dest_decl 不为 NULL，则将它保存在 link_state_t.dest_decl 指针中。由上文 type_copy_callback 函数可见，在将模块的 type 复制到 base.p_types 符号表之后，如果 dest_decl 指针有效，则还要再次插入它所指向的 avrule_decl_t.symtab[SYM_TYPES]符号表。（只有在 copy_avrule_block > copy_avrule_decl > copy_identifier 时参数 dest_decl 不为 NULL，为在 copy_avrule_decl 函数中新创建的 avrule_decl_t 数据结构的地址）

```
        for (i = 0; i < SYM_NUM; i++) {
                if (copy_callback_f[i] != NULL) {
                        ret = hashtab_map(src_symtab[i].table, copy_callback_f[i], state);
                        if (ret)
                                return ret;
                }
        }
```

接下来就以“表格驱动”的方式，就当前模块 symtab[]中的每类符号表，调用 copy_callback_f[]函数指针数组中的相应方法。

```
        if (hashtab_map(src_symtab[SYM_TYPES].table, type_bounds_copy_callback, state))
                return -1;

        if (hashtab_map(src_symtab[SYM_TYPES].table, alias_copy_callback, state))
                return -1;

        if (hashtab_map(src_symtab[SYM_ROLES].table, role_bounds_copy_callback, state))
                return -1;

        if (hashtab_map(src_symtab[SYM_USERS].table, user_bounds_copy_callback, state))
                return -1;
```

所有的符号表都合并完成后，types/roles/users 符号表还需要根据标识符的新的 policy value 更新原有 xxx_datum_t 中所使用的各种 ebitmap，分别由相应的函数完成：

```
        /* then fix bitmaps associated with those newly copied identifiers */
```

```
        for (i = 0; i < SYM_NUM; i++) {
                if (fix_callback_f[i] != NULL &&
                    hashtab_map(src_symtab[i].table, fix_callback_f[i], state)) {
                        return -1;
                }
        }
        return 0;
}
```

体会：相应位图的修正工作不能在拷贝标识符的 xxx_datum_t 时进行，而**应该等当前模块内相关类型的所有标识符都拷贝完成后才能进行！**

比如 role_datum_t.dominates 位图描述了被当前 super-role 所 dominate 的所有 sub-role。无论 super-role 还是 sub-role，在当前模块中要么被定义，要么被声明为外部引用，所以在当前模块的 p_roles 符号表中都注册相关的 role_datum_t 数据结构。在处理 super-role 时，相关 sub-role 可能还没有被处理过，在 policy_module_t.map[SYM_ROLES][]数组中相关条目尚未建立，因此必须等到 p_roles 符号表中的所有元素都被合并到 base 之后才能修正 dominates 位图。

进一步，如果涉及其他类型的标识符，比如 user-role，role-type，则必须等待其他类型标识符都被合并才能修正相应的位图。

12.5.2.3 p_roles 符号表的修正

在使用 role_copy_callback 函数拷贝了非 base 模块的 p_roles 符号表后，还需要调用 role_fix_callback 函数再次遍历当前模块的 p_roles 符号表，因为在 role_copy_callback 函数中并没有处理当前 role 已经被合并到 base 模块中的情况，此时需要进一步将当前 role 的 dominates 和 types 位图，合并到 base 中已经存在的相同 key 的 role_datum_t 的 dominates 和 types 位图中。

role_fix_callback 函数处理非 base 模块的 p_roles 哈希表的一个元素，参数为当前 role 标识符的 key，以及相应的 role_datum_t 数据结构。

```
[semodule_link > semodule_link_package > link_modules > copy_identifiers > hashtab_map > role_fix_callback]

static int role_fix_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        unsigned int i;
        char *id = key;
        role_datum_t *role, *dest_role = NULL;
        link_state_t *state = (link_state_t *)data;
        ebitmap_t e_tmp;
        policy_module_t *mod = state->cur;              # 指向 link_modules 过程中当前模块的 policy_module_t
        ebitmap_node_t *rnode;
        hashtab_t role_tab;

        role = (role_datum_t *)datum;
        if (state->dest_decl == NULL)
                role_tab = state->base->p_roles.table;
        else
                role_tab = state->dest_decl->p_roles.table;
```

注意，link 过程中只有 user/role/type_copy_callback 函数会检查 link_state_t.dest_decl 指针是否为空，且在非空时还将当前标识符的 xxx_datum_t 数据结构注册到 base 模块中新的 block/decl 的相应符号表中，因此 user/role/type_fix_copy_callback 函数需要支持两种不同的 symtab：

1，link_modules > copy_identifiers > role_fix_callback 函数调用链负责修正**当前模块**的 p_xxx 符号表，此时 link_state_t.dest_decl 指针为 NULL，所以这里将 role_tab 哈希表初始化为 base 模块的 p_roles 符号表；

2，link_modules > copy_module > copy_avrule_block > copy_avrule_decl > copy_identifiers > role_fix_callback 调用链负责修正**当前 block/decl** 的符号表，参数 dest_decl 不为 NULL，所以这里将 role_tab 哈希表初始化为相应 avrule_decl_t 的 p_roles 符号表。

```
dest_role = hashtab_search(role_tab, id);
assert(dest_role != NULL);
```

合并后，在 base 模块 p_roles 符号表中一定能够找到具有相同 key 的 role_datum_t 数据结构，用 assert 宏确保这一点。

```
if (state->verbose) {
        INFO(state->handle, "fixing role %s", id);
}

ebitmap_init(&e_tmp);
ebitmap_for_each_bit(&role->dominates, rnode, i) {
        if (ebitmap_node_get_bit(rnode, i)) {
                assert(mod->map[SYM_ROLES][i]);
                if (ebitmap_set_bit(&e_tmp, mod->map[SYM_ROLES][i] - 1, 1)) {
                        goto cleanup;
                }
        }
}
```

首先合并 role_datum_t 数据结构中的 dominates 位图，它记录了当前 role 能够 dominate 哪些其他的 role（位图中在被 dominate 的 role 的(policy value - 1)位置上置 1）。

ebitmap_for_each_bit 宏能够遍历 ebitmap.node 所指队列中的每个 ebitmap_node_t 节点元素。
ebitmap_node_get_bit 宏能够进一步取出 rnode 所指当前 ebitmap_node_t 数据结构中每个位置上的值。

首先使用 assert 宏确认当前模块 policy_module_t.map[SYM_ROLES][i]数组元素非 0，然后以相应的元素值（即该 role 标识符在 base 中的新 policy value）减 1 为索引，设置到临时位图 e_tmp 中。

```
if (ebitmap_union(&dest_role->dominates, &e_tmp)) {
        goto cleanup;
}
```

然后调用 ebitmap_union > ebitmap_or 函数将两张位图合并（注意在 role_copy_callback 中并没有合并 dest_role.dominates 位图，而是由 role_fix_callback 函数在这里完成）。

```
if (type_set_or_convert(&role->types, &dest_role->types, mod, state)) {
        goto cleanup;
}
ebitmap_destroy(&e_tmp);
return 0;
```

接着处理 dest_role.types 数据结构，它包含 types/negset 两个位图。用位图中每个非 0 位置（即相应 type 标识符在模块内的 policy value 减 1）索引模块的 policy_module_t.map[SYM_TYPES][]数组得到合并后新的 policy value，并设置到 dest_role.types 位图中。参见下文。

```
cleanup:
  ERR(state->handle, "Out of memory!");
  ebitmap_destroy(&e_tmp);
  return -1;
}
```

体会：各个模块都可以使用“role xxx types xxx”规则来定义一个 role 可以和哪些 type 组成合法的 SC，使用“allow role_xx role_yy”来定义一个 role 能够切换到哪些其他的 role。正是在 role_fix_callback 函数中将当前模块 role 的相应 types 和 dominates 位图合并到 base 模块中！（注意，在当前模块的 role/type 都被合并到 base 后，得到新的 policy value 后才能进行）

```
[semodule_link > semodule_link_package > link_modules > copy_identifiers > hashtab_map > role_fix_callback
> type_set_or_convert]

/* OR 2 typemaps together and at the same time map the src types to
 * the correct values in the dst typeset.
 */
static int type_set_or_convert(type_set_t * types, type_set_t * dst,
                               policy_module_t * mod, link_state_t * state)
{
        type_set_t ts_tmp;

        type_set_init(&ts_tmp);
        if (type_set_convert(types, &ts_tmp, mod, state) == -1) {
                goto cleanup;
        }
        if (type_set_or_eq(dst, &ts_tmp)) {
                goto cleanup;
        }
        type_set_destroy(&ts_tmp);
        return 0;

      cleanup:
        ERR(state->handle, "Out of memory!");
        type_set_destroy(&ts_tmp);
        return -1;
}
```

type_set_convert 函数的参数 types 指向模块内 role_datum_t.types 数据结构，dst 指向 base 模块内 dest_role.types 数据结构。就第一个位图中的非 0 位置，以其为索引查找当前模块的 policy_module_t.map[SYM_TYPES][]数组，以数组元素的值减 1 将第二个位图的相应位置位：

```
static int type_set_convert(type_set_t * types, type_set_t * dst,
                            policy_module_t * mod, link_state_t * state __attribute__ ((unused)))
{
        unsigned int i;
        ebitmap_node_t *tnode;
        ebitmap_for_each_bit(&types->types, tnode, i) {
                if (ebitmap_node_get_bit(tnode, i)) {
                        assert(mod->map[SYM_TYPES][i]);
                        if (ebitmap_set_bit (&dst->types, mod->map[SYM_TYPES][i] - 1, 1)) {
                                goto cleanup;
                        }
                }
        }

        ebitmap_for_each_bit(&types->negset, tnode, i) {
```

```
                if (ebitmap_node_get_bit(tnode, i)) {
                        assert(mod->map[SYM_TYPES][i]);
                        if (ebitmap_set_bit (&dst->negset, mod->map[SYM_TYPES][i] - 1, 1)) {
                                goto cleanup;
                        }
                }
        }
        dst->flags = types->flags;
        return 0;

      cleanup:
        return -1;
}
```

按照上述方法设置 dst type_set_t 中的 types 和 negset 位图，并复制原来的 flags。注意在 flags 中描述特殊字符“*”、“~”和“-”，特殊字符在 expand 过程中才被展开。

12.5.3 scope 符号表的拷贝

copy_module 函数调用 scope_copy_callback 函数将当前模块的所有类型的 scope 符号表合并到 base 模块中相应类型的 scope 符号表：

```
        /* then copy the scoping tables */
        for (i = 0; i < SYM_NUM; i++) {
                state->symbol_num = i;          # 当前拷贝的 scope 符号表类型，由 scope_copy_callback 函数使用
                if (hashtab_map(module->policy->scope[i].table, scope_copy_callback, state)) {
                        return -1;
                }
        }
```

由此可见 link_state_t.symbol_num 为当前循环所处理的 scope 符号表的类型。scope_copy_callback 函数每次处理一个标识符的 key 和 scope_datum_t 数据结构：

```
static int scope_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        unsigned int i;
        int ret;
        char *id = key, *new_id = NULL;
        scope_datum_t *scope, *base_scope;
        link_state_t *state = (link_state_t *)data;
        uint32_t symbol_num = state->symbol_num;                # 当前处理的标识符类型
        uint32_t *avdecl_map = state->cur->avdecl_map;          # 当前模块的 block/decl 的 decl_id 转换数组

        scope = (scope_datum_t *)datum;

        /* check if the base already has a scope entry */
        base_scope = hashtab_search(state->base->scope[symbol_num].table, id);
```

首先在 base 模块相应类型的 scope 符号表中查找是否已经存在相同 key 的元素。如果没有，则说明当前模块定义或引用了一个此前尚未出现（尚未 link 到 base 模块中）的标识符，则应该复制并插入 base 模块的相应类型的 scope 符号表：

```
        if (base_scope == NULL) {
                scope_datum_t *new_scope;
                if ((new_id = strdup(id)) == NULL) {
                        goto cleanup;
                }
```

```
        if ((new_scope = (scope_datum_t *)calloc(1, sizeof(*new_scope))) == NULL) {
                free(new_id);
                goto cleanup;
        }
        ret = hashtab_insert(state->base->scope[symbol_num].table,
                             (hashtab_key_t)new_id, (hashtab_datum_t)new_scope);
        if (ret) {
                free(new_id);
                free(new_scope);
                goto cleanup;
        }
        new_scope->scope = SCOPE_REQ;   /* this is reset further down */
        base_scope = new_scope;
}
```

至此，base_scope 总是指向 base 模块中当前标识符的 scope_datum_t 数据结构。对于第一次插入的标识符，其 base_scope->scope 暂时设置为 SCOPE_REQ，会在下面 if-else-if 的第 1 和 3 种分支继续处理。

```
if (base_scope->scope == SCOPE_REQ && scope->scope == SCOPE_DECL) {
        /* this module declared symbol, so overwrite the old
         * list with the new decl ids */
        base_scope->scope = SCOPE_DECL;
        free(base_scope->decl_ids);
        base_scope->decl_ids = NULL;
        base_scope->decl_ids_len = 0;
        for (i = 0; i < scope->decl_ids_len; i++) {
                if (add_i_to_a(avdecl_map[scope->decl_ids[i]],
                               &base_scope->decl_ids_len, &base_scope->decl_ids) == -1) {
                        goto cleanup;
                }
        }
```

如果一个模块提供了一个标识符的定义，而此前该标识符并在 base 中不存在，或者合并到 base 中的其他模块都只是该标识符的引用者，则释放 base_scope->decl_ids[]数组，并拷贝定义者模块内为其设置的相应内容。

注意 scope_datum_t.decl_ids[]保存了模块内使用（定义或引用）该标识符的 block/decl 的 decl_ids，它们都已经被合并到 base 模块中并分配了新的 decl_id，因此要使用当前模块的 policy_module_t.avdecl_map[]数组转换为新的 decl_id。

```
} else if (base_scope->scope == SCOPE_DECL && scope->scope == SCOPE_REQ) {
        /* this module depended on a symbol that now exists, so don't do anything */
```

如果一个模块是当前标识符的使用者，而其定义者模块已经合并到 base 模块中，则这里什么都不做（从而和第 1 种情况的逻辑相吻合）。

体会：上面这两种逻辑将使得完成 link 过程的 base 模块的 scope 中，应该只含有 SCOPE_DECL 类型的元素！否则就说明某些标识符的外部依赖尚未满足，link 过程应该失败。由下文可知，在 expand 过程中相应 xxx_copy_callback > is_id_enabled 函数检查当前 key 对应的 scope_datum_t 是否存在；scope->scope 是否为 SCOPE_DECL，并且至少存在一个有效的、定义了该符号的 block/decl。

```
} else if (base_scope->scope == SCOPE_REQ && scope->scope == SCOPE_REQ) {
        /* symbol is still required, so add to the list */
        for (i = 0; i < scope->decl_ids_len; i++) {
                if (add_i_to_a(avdecl_map[scope->decl_ids[i]],
```

```
                                &base_scope->decl_ids_len, &base_scope->decl_ids) == -1) {
                        goto cleanup;
                }
        }
```

如果该标识符此前在 base 模块中并不存在，或者当前模块是它的另外一个引用者，则合并该模块中当前标识符的 scope 信息到 base_scope 中。注意也需要查询 avdecl_map[]数组转换该模块中相关 block/decl 在 base 模块中的 decl_id。

```
        } else {
                /* this module declared a symbol, and it was already
                 * declared.  only roles and users may be multiply
                 * declared; for all others this is an error. */
                if (symbol_num != SYM_ROLES && symbol_num != SYM_USERS) {
                        ERR(state->handle, "%s: Duplicate declaration in module: %s %s",
                            state->cur_mod_name, symtab_names[state->symbol_num], id);
                        return -1;
                }
                for (i = 0; i < scope->decl_ids_len; i++) {
                        if (add_i_to_a(avdecl_map[scope->decl_ids[i]],
                                       &base_scope->decl_ids_len, &base_scope->decl_ids) == -1) {
                                goto cleanup;
                        }
                }
        }
        return 0;

      cleanup:
        ERR(state->handle, "Out of memory!");
        return -1;
}
```

剩下的只能是当前标识符被重复定义的情况。目前只允许 role/user 类型标识符出现重复定义，即在一个模块内、或者多个模块内使用若干 role/user 规则定义若干 role-type，user-role 关系，那么只需要合并当前模块内对该标识符的 scope 信息到 base_scope 即可。除此之外均为非法情况。

12.5.4 链接过程的主要函数调用链

如前文所述，semodule_link 程序调用 load_module 函数读取二进制 pp 文件，并用 sepol_module_package_t 数据结构描述每一个模块，其中包含模块的 policydb_t 数据结构，以及指向模块内定义的文件 SC、SELinux 用户等字符串及相应长度。参数 modules 为非 base 模块的 sepol_module_package_t 数据结构的指针数组，num_modules 为指针数组的长度。

```
[main@semodule_link.c > semodule_link_packages]

/* Links the module packages into the base.  Returns 0 on success, -1
 * if a requirement was not met, or -2 for all other errors. */
int sepol_link_packages(sepol_handle_t * handle, sepol_module_package_t * base,
                        sepol_module_package_t ** modules, int num_modules, int verbose)
{
        policydb_t **mod_pols = NULL;
        int i, retval;

        if ((mod_pols = calloc(num_modules, sizeof(*mod_pols))) == NULL) {
                ERR(handle, "Out of memory!");
                return -2;
```

```
        }
        for (i = 0; i < num_modules; i++) {
                mod_pols[i] = &modules[i]->policy->p;
        }
```

首先创建另外一个指针数组 mod_pols[]，其元素指向所有非 base 模块的 policydb_t 数据结构。然后调用 link_modules 函数实现 link 过程:

```
        retval = link_modules(handle, &base->policy->p, mod_pols, num_modules, verbose);
        free(mod_pols);
        if (retval == -3) {
                return -1;
        } else if (retval < 0) {
                return -2;
        }

        if (link_file_contexts(base, modules, num_modules) == -1) {
                ERR(handle, "Out of memory!");
                return -2;
        }

        if (link_netfilter_contexts(base, modules, num_modules) == -1) {
                ERR(handle, "Out of memory!");
                return -2;
        }

        return 0;
}
```

最后通过 link_file_contexts 和 link_netfilter_contexts 函数，将各个非 base 模块内定义的 file_contexts 和 netfilter_contexts 字符串逐一追加到 base 模块相应字符串的尾部。这两个函数很简单，不再详述。

所谓模块的“link”其实就是将非 base 模块 policydb_t 数据结构的各个域，合并到 base 模块 policydb_t 数据结构的相应域上（包括标识符的符号表及各种规则的描述和组织数据结构）。link_modules 函数的参数包括 base 模块 policydb_t 数据结构的指针，其它模块 policydb_t 数据结构的指针数组及其长度。

```
[semodule_link > semodule_link_packages > link_modules]

/* Link a set of modules into a base module. This process is somewhat
 * similar to an actual compiler: it requires a set of order dependent
 * steps.  The base and every module must have been indexed prior to
 * calling this function.
 */
int link_modules(sepol_handle_t * handle, policydb_t * b, policydb_t ** mods, int len, int verbose)
{
        int i, ret, retval = -1;
        policy_module_t **modules = NULL;
        link_state_t state;
        uint32_t num_mod_decls = 0;

        memset(&state, 0, sizeof(state));
        state.base = b;
        state.verbose = verbose;
        state.handle = handle;
```

链接过程使用的核心数据结构为 link_state_t，它为 link_modules 函数的自动变量，这里首先初始化。

```
if (b->policy_type != POLICY_BASE) {
        ERR(state.handle, "Target of link was not a base policy.");
        return -1;
}

/* first allocate some space to hold the maps from module
 * symbol's value to the destination symbol value; then do
 * other preparation work */
if ((modules = (policy_module_t **)calloc(len, sizeof(*modules))) == NULL) {
        ERR(state.handle, "Out of memory!");
        return -1;
}
```

如前文所述，每一个模块的二进制 pp 文件都由 sepol_module_package_t 数据结构描述。在链接的过程中，模块内所有标识符（无论在模块内定义，或被声明为外部依赖）的 policy value 必须在 base 模块相应符号表的 nprim 数值基础上被重新分配。policy_module_t 数据结构用于描述 link 前后标识符 policy value，block/decl 的 decl_id 等的变化关系。这里首先创建该数据结构的指针数组，下面再逐一为所有非 base 模块创建 policy_module_t 数据结构。

```
for (i = 0; i < len; i++) {
        if (mods[i]->policy_type != POLICY_MOD) {
                ERR(state.handle, "Tried to link in a policy that was not a module.");
                goto cleanup;
        }
        if (mods[i]->mls != b->mls) {
                if (b->mls)
                        ERR(state.handle, "Tried to link in a non-MLS module with an MLS base.");
                else
                        ERR(state.handle, "Tried to link in an MLS module with a non-MLS base.");
                goto cleanup;
        }

        if ((modules[i] = (policy_module_t *)calloc(1, sizeof(policy_module_t))) == NULL) {
                ERR(state.handle, "Out of memory!");
                goto cleanup;
        }
        modules[i]->policy = mods[i];
        if (prepare_module(&state, modules[i]) == -1) {
                goto cleanup;
        }
        num_mod_decls += modules[i]->num_decls;
}
```

就每一个非 base 模块，首先检查其类型是否为 POLICY_MOD，是否和 base 模块的 MLS 属性相一致。为每个非 base 模块都创建 policy_module_t 数据结构，由 modules[]指针数组的相应元素指向，其 policy 指针也指向当前模块的 policydb_t 数据结构。

prepare_module 函数初始化 policy_module_t 数据结构：根据模块各类符号表的 nprim（为最大 policy value 值）分配相应长度的数组，由相应 map[]元素指向；统计模块的所有 avrule_decl_t 数据结构的个数 num_decl，并分配 perm_map_len 和 avrule_map 数组，以及 perm_maps 指针数组等。

```
if (prepare_base(&state, num_mod_decls) == -1) {
        goto cleanup;
}
```

每个模块的规则都组织在若干 block/decl 中，相应 avrule_block_t 数据结构组织在其 policydb_t.global 链表里，而每个 avrule_block_t 数据结构中又有若干 avrule_decl_t 数据结构组织在其 branch_list 链表里。每个 avrule_decl_t 数据结构都有惟一的 decl_id，而 policydb_t 中的 decl_val_to_struct 指针数组可以用它索引相应的 avrule_decl_t 数据结构。

为了合并所有其他模块的 avrule_decl_t 数据结构，需要首先释放 base 模块的 decl_val_to_struct 指针数组，并且“扩容”至能够容纳所有模块的所有 avrule_decl_t 数据结构的地址。该工作由 prepare_base 函数完成，另外还设置 link_state_t 数据结构的其他域，比如将 last_base_avrule_block 和 last_avrule_block 指针指向 base 的最后一个 avrule_block_t 数据结构，next_decl_id 为在 base 的 avrule_decl_t 基础上下一个可用的编号（以便给所有非 base 模块的 avrule_decl_t 数据结构重新编号，而新旧编号的映射关系将记录到 policy_module_t.avrule_map 数组中）。

```
/* copy all types, declared and required */
for (i = 0; i < len; i++) {
        state.cur = modules[i];
        state.cur_mod_name = modules[i]->policy->name;
        ret = hashtab_map(modules[i]->policy->p_types.table, type_copy_callback, &state);
        if (ret) {
                retval = ret;
                goto cleanup;
        }
}
```

给模块准备好了 policy_module_t 数据结构之后，就可以合并其符号表到 base 模块中了。link_state_t.cur 指向当前被处理模块的 policy_module_t 数据结构。调用 type_copy_callback 函数实现 type/attribute/alias 的合并，首先查询 base.p_types 符号表中是否存在相同 key 的元素，如果没有则复制 key 和 type_datum_t，并调用 hashtab_insert 插入。注意复制后的 base_type.s.value 从 base.p_types.nprim 开始重新分配，新旧 policy value 的关系则由 policy_module_t.map[SYM_TYPES][] 数组来描述。参见上文。

注意，这里首先显式地调用 type_copy_callback 函数，合并**所有**模块的 p_types 符号表。然后再逐一针对各个模块调用 copy_identifiers 函数拷贝当前模块的所有其它类型的符号表。注意，到目前为止合并的都是模块的 p_xxx 符号表（即模块的 global block 的符号表），**而不包含模块的任意 block/decl 的符号表**（因为传递的参数 dest_decl == NULL）：

```
/* then copy everything else, including aliases, and fixup attributes */
for (i = 0; i < len; i++) {
        state.cur = modules[i];
        state.cur_mod_name = modules[i]->policy->name;
        ret = copy_identifiers(&state, modules[i]->policy->symtab, NULL);
        if (ret) {
                retval = ret;
                goto cleanup;
        }
}
```

至此，所有非 base 模块的所有 p_xxx 符号表都被合并到 base 模块中，而且所有的 fixup 函数都已被调用，参见上文。

```
if (policydb_index_others(state.handle, state.base, 0)) {
        ERR(state.handle, "Error while indexing others");
        goto cleanup;
}
```

base 模块的所有符号表都完成合并后，就可以进一步设置其 policydb_t 中若干根据标识符的(policy value - 1)查询标识符的 key 和 xxx_datum_t 数据结构的数组了，比如 8 个 p_xxx_val_to_name[]数组，以及 4 个 class/role/user/type_val_to_struct[]数组。这些数组的初始化和设置工作都在 policydb_index_others > hashtab_map > xxx_index 函数中完成，比如 type_index 函数的核心操作为就 base 模块 p_types 符号表中当前元素的 key 和 typdatum，以相应标识符的(policy value - 1)为索引，设置指针数组元素：

```
p->p_type_val_to_name[typdatum->s.value - 1] = (char *)key;
p->type_val_to_struct[typdatum->s.value - 1] = typdatum;
```

```
/* copy and remap the module's data over to base */
for (i = 0; i < len; i++) {
        state.cur = modules[i];
        ret = copy_module(&state, modules[i]);
        if (ret) {
                retval = ret;
                goto cleanup;
        }
}
```

然后逐一拷贝各个模块的各种 scope 符号表，以及模块中的所有 block/decl 数据结构（包含规则描述符，以及当前 block/decl 的私有符号表），参见下文。

```
/* re-index base, for symbols were added to symbol tables  */
if (policydb_index_classes(state.base)) {
        ERR(state.handle, "Error while indexing classes");
        goto cleanup;
}
if (policydb_index_others(state.handle, state.base, 0)) {
        ERR(state.handle, "Error while indexing others");
        goto cleanup;
}
```

至此，所有模块的所有 block/decl 定义的符号表都已经合并到 base 模块的 p_xxx 符号表中，就可以再次调用上述函数建立完整的索引机制。

```
if (enable_avrules(&state, state.base)) {
        retval = SEPOL_EREQ;
        goto cleanup;
}
```

调用 enable_avrules 函数，遍历 global 队列中所有 block 的 decl 的外部依赖是否被满足，如果 unconditional block/decl 的外部依赖不满足，则报错。对于 optional block，根据情况设置 avrule_block_t.enable 指向 non-else 或者 else decl（并设置 avrule_decl_t.enabled 标志）。参见下文。

```
retval = 0;

cleanup:
  for (i = 0; modules != NULL && i < len; i++) {
        policy_module_destroy(modules[i]);
  }
  free(modules);
  free(state.decl_to_mod);
  return retval;
}
```

copy_module 函数合并当前模块的所有符号表，avrule_block/decl_t 数据结构和 scope 数据结构到 base 中。
（TODO：在 link_modules 调用 copy_module 之前，已经显式地调用过 copy_identifiers 函数完成了所有符号表的拷贝，岂不这里会发生重复？而且目前 copy_module 只在 link.c 中被 link_modules 所调用，因此能否删除 copy_module > copy_identifers）

[semodule_link > semodule_link_packages > link_modules > copy_module]

```
/* Copy a module over to a base, remapping all values within.  After
 * all identifiers and rules are done, copy the scoping information.
 * This is when it checks for duplicate declarations. */
static int copy_module(link_state_t * state, policy_module_t * module)
{
        int i, ret;
        avrule_block_t *cur;
        state->cur = module;
        state->cur_mod_name = module->policy->name;

        /* first copy all of the identifiers */
        ret = copy_identifiers(state, module->policy->symtab, NULL);
        if (ret) {
                return ret;
        }
```

首先调用 copy_identifiers 函数拷贝当前模块的所有符号表到 base 的相应符号表，并根据当前模块的标识符新旧 policy value 转换关系修正所有相关的 ebitmap 数据结构。

```
        /* next copy all of the avrule blocks */
        for (cur = module->policy->global; cur != NULL; cur = cur->next) {
                ret = copy_avrule_block(state, module, cur);
                if (ret) {
                        return ret;
                }
        }
```

然后在循环中通过 copy_avrule_block 函数逐一拷贝当前非 base 模块 global 链表中的所有 avrule_block_t 数据结构到 base 模块中，它会进一步调用 copy_avrule_decl 函数逐一拷贝 avrule_block_t.branch_list 链表上的所有 avrule_decl_t 数据结构（包括其中组织各种规则的队列），参见下文。

注意，只有在 link 过程中对所有的模块都调用过 copy_module > copy_identifiers 函数后，才能判断所有 block/decl 的外部依赖是否被满足。所以在 link 过程的后部才调用 enable_avrules 函数确定生效的 block/decl。而此时则义无反顾地拷贝当前模块的所有 block/decl 到 base 模块中。

而在 expand 过程中，只处理那些有效的 block/decl 中记录的所有规则。

```
        /* then copy the scoping tables */
        for (i = 0; i < SYM_NUM; i++) {
                state->symbol_num = i;
                if (hashtab_map(module->policy->scope[i].table, scope_copy_callback, state)) {
                        return -1;
                }
        }
```

```
        return 0;
}
```

最后通过 scope_copy_callback 函数，将所有非 base 模块的所有类型的 scope 符号表合并到 base 模块中，注意合并后 base 模块中应该只保留类型为 SCOPE_DECL 的 scope_datum_t 数据结构，参见上文。

所有模块的 scope 符号表都被合并到 base 模块后，在 enable_avrules 函数中就可以检查任意 decl 的外部依赖是否被满足了。

copy_avrule_block 函数复制当前模块的一个 avrule_block_t 数据结构到 base 模块的 global 链表中。

```
[semodule_link > semodule_link_packages > link_modules > copy_module > copy_avrule_block]

static int copy_avrule_block(link_state_t * state, policy_module_t * module, avrule_block_t * block)
{
        avrule_block_t *new_block = avrule_block_create();
        avrule_decl_t *decl, *last_decl = NULL;
        int ret;

        if (new_block == NULL) {
                ERR(state->handle, "Out of memory!");
                ret = -1;
                goto cleanup;
        }

        new_block->flags = block->flags;
```

首先分配一个新的 avrule_block_t 数据结构，并拷贝 flags。接下来遍历现有 block 的 avrule_block_t.branch_list 队列，复制其中所有 avrule_decl_t 数据结构，并加入新 block 的 branch_list 队列：

```
        for (decl = block->branch_list; decl != NULL; decl = decl->next) {
                avrule_decl_t *new_decl = avrule_decl_create(state->next_decl_id);
                if (new_decl == NULL) {
                        ERR(state->handle, "Out of memory!");
                        ret = -1;
                        goto cleanup;
                }
```

首先分配一个新的 avrule_decl_t 数据结构。由上文可知 link_state_t.next_decl_id 此前被设置下一个可用的 id，现在要用它为复制过来的各个模块的 avrule_decl_t 数据结构重新设置 decl_id。

```
                if (module->policy->name != NULL) {
                        new_decl->module_name = strdup(module->policy->name);
                        if (new_decl->module_name == NULL) {
                                ERR(state->handle, "Out of memory\n");
                                ret = -1;
                                goto cleanup;
                        }
                }
```

复制 avrule_decl_t.module_name 为当前模块的名字。

```
                if (last_decl == NULL) {
                        new_block->branch_list = new_decl;
                } else {
```

```
                last_dec1->next = new_dec1;
        }
        last_dec1 = new_dec1;
```

将该 avrule_dec1_t 数据结构加入新 block 的 avrule_block_t.branch_list 链表。局部变量 last_dec1 指向链表的末尾元素。

```
        state->base->dec1_val_to_struct[state->next_dec1_id - 1] = new_dec1;
        state->dec1_to_mod[state->next_dec1_id] = module->policy;
```

进一步，设置 base 模块的 dec1_val_to_struct[]和 dec1_to_mod[]索引数组。在 prepare_base 函数中已经为它们分配了足够的空间以容纳所有模块的 avrule_dec1_t 数据结构的指针。

```
        module->avdec1_map[dec1->dec1_id] = new_dec1->dec1_id;
```

用 policy_module_t.avdec1_map[]数组记录一个 avrule_dec1_t 数据结构在复制到 base 前后其 dec1_id 的变化关系。

```
        ret = copy_avrule_dec1(state, module, dec1, new_dec1);
        if (ret) {
                goto cleanup;
        }

        state->next_dec1_id++;
} # 复制原有 block 的一个 avrule_dec1_t
```

调用 copy_avrule_dec1 函数拷贝当前 avrule_dec1_t 数据结构中的所有规则到 new_dec1 中。注意要根据相关标识符在 base 模块中的新 policy value 来调整描述规则的数据结构中相关的域。

至此，原有 block 的 branch_list 队列中的所有 avrule_dec1_t 都被复制，且加入了 new_block.branch_list 队列。最后就需要把 new_block 加入 base 模块的 global 队列末尾，它由 link_state_t.last_varule_block 指针所指向：

```
        state->last_avrule_block->next = new_block;
        state->last_avrule_block = new_block;
        return 0;

      cleanup:
        avrule_block_list_destroy(new_block);
        return ret;
}
```

copy_avrule1_dec1 函数复制模块的一个 avrule_dec1_t 数据结构，其中描述当前 block/dec1 规则的所有数据结构都需要被复制。

```
[semodule_link > semodule_link_packages > link_modules > copy_module > copy_avrule_block >
copy_avrule_dec1]

static int copy_avrule_dec1(link_state_t * state, policy_module_t * module,
                            avrule_dec1_t * src_dec1, avrule_dec1_t * dest_dec1)
{
        int ret;

        /* copy all of the RBAC and TE rules */
        if (copy_avrule_list(src_dec1->avrules, &dest_dec1->avrules, module, state) == -1
```

```
            || copy_role_trans_list(src_decl->role_tr_rules,
                                    &dest_decl->role_tr_rules, module, state) == -1
            || copy_role_allow_list(src_decl->role_allow_rules,
                                    &dest_decl->role_allow_rules, module, state) == -1
            || copy_cond_list(src_decl->cond_list,
                                    &dest_decl->cond_list, module, state) == -1) {
                return -1;
        }

        if (copy_range_trans_list(src_decl->range_tr_rules, &dest_decl->range_tr_rules, module, state))
                return -1;

        /* finally copy any identifiers local to this declaration */
        ret = copy_identifiers(state, src_decl->symtab, dest_decl);
        if (ret < 0) {
                return ret;
        }

        /* then copy required and declared scope indices here */
        if (copy_scope_index(&src_decl->required, &dest_decl->required, module, state) == -1 ||
            copy_scope_index(&src_decl->declared, &dest_decl->declared, module, state) == -1) {
                return -1;
        }

        return 0;
}
```

role_trans_rule_t 数据结构用于描述 role_transition 规则的“字面”描述。copy_role_trans_list 函数用于复制 list 链表中的 role_trans_rule_t 数据结构并加入 dst 所指向的链表。注意需要根据相应标识符在 base 模块中的新 policy value 来重新设置 new_rule 中所有使用 ebitmap 的域和 new_role 域。

```
[semodule_link > semodule_link_packages > link_modules > copy_module > copy_avrule_block > copy_avrule_decl
> copy_role_trans_list]

static int copy_role_trans_list(role_trans_rule_t * list, role_trans_rule_t ** dst,
                                policy_module_t * module, link_state_t * state)
{
        role_trans_rule_t *cur, *new_rule = NULL, *tail;
        unsigned int i;
        ebitmap_node_t *cnode;

        cur = list;
        tail = *dst;
        while (tail && tail->next) {
                tail = tail->next;
        }
```

如果 dst 所指链表非空，则用 tail 指向其末尾元素，以便下面在其后插入新元素。

```
        while (cur) {
                if ((new_rule = (role_trans_rule_t *)malloc(sizeof(role_trans_rule_t))) == NULL) {
                        goto cleanup;
                }
                role_trans_rule_init(new_rule);
```

循环处理 list 链表中的所有元素。首先分配一个新的 role_trans_rule_t 数据结构并初始化。

```
                if (role_set_or_convert(&cur->roles, &new_rule->roles, module, state)
```

```
                || type_set_or_convert(&cur->types, &new_rule->types, module, state)) {
                    goto cleanup;
            }
```

调用 role/type_set_or_convert 函数复制 role/type_set_t 域，都需要查询当前模块 policy_module_t.map[SYM_ROLES/TYPES][]数组得到原有位图的非 0 位置（即标识符在模块内的 policy value 减 1）在 base 模块内的新位置（即该标识符在 base 模块内的 policy value），然后设置新位图的相应位。

```
            ebitmap_for_each_bit(&cur->classes, cnode, i) {
                    if (ebitmap_node_get_bit(cnode, i)) {
                            assert(module->map[SYM_CLASSES][i]);
                            if (ebitmap_set_bit(&new_rule->classes,
                                                module->map[SYM_CLASSES][i] - 1, 1)) {
                                    goto cleanup;
                            }
                    }
            }

            new_rule->new_role = module->map[SYM_ROLES][cur->new_role - 1];
```

然后使用相同的方法处理 role_transition 规则中的 classes 和 new_role 域。

```
            if (*dst == NULL) {
                    *dst = new_rule;
            } else {
                    tail->next = new_rule;
            }
            tail = new_rule;
            cur = cur->next;
    }
```

最后将 new_rule 插入 dst 所指链表，并更新 tail 和 cur 指针。

```
    return 0;
  cleanup:
    ERR(state->handle, "Out of memory!");
    role_trans_rule_list_destroy(new_rule);
    return -1;
}
```

copy_scope_index 函数用于复制一个 avrule_decl_t 中的 scope_index_t 数据结构中的所有 ebitmap 位图。核心操作是就位图中非 0 位置，查询当前模块的 policy_module_t.map[SYM_XXX][]数组，得到相应标识符在 base 模块中的新 policy value，设置到新位图中去。

```
[semodule_link > semodule_link_packages > link_modules > copy_module > copy_avrule_block > copy_avrule_decl
> copy_scope_index]

static int copy_scope_index(scope_index_t * src, scope_index_t * dest,
                            policy_module_t * module, link_state_t * state)
{
    unsigned int i, j;
    uint32_t largest_mapped_class_value = 0;
    ebitmap_node_t *node;
```

avrule_decl_t.required/declared 分别描述在当前 block/decl 中所引用的或定义的标识符（位图中以其(policy value - 1)为索引，相应位置位）。每次循环拷贝一种类型的位图。

```
/* copy the scoping information for this avrule decl block */
for (i = 0; i < SYM_NUM; i++) {
        ebitmap_t *srcmap = src->scope + i;
        ebitmap_t *destmap = dest->scope + i;
        if (copy_callback_f[i] == NULL) {
                continue;
        }
```

由于在模块中不能定义 common，所以跳过 i == SYM_COMMONS == 0 的情况。

```
        ebitmap_for_each_bit(srcmap, node, j) {
                if (ebitmap_node_get_bit(node, j)) {
                        assert(module->map[i][j] != 0);
                        if (ebitmap_set_bit(destmap, module->map[i][j] - 1, 1) != 0) {
                                goto cleanup;
                        }
```

就当前标识符类型的位图，逐一遍历 ebitmap_node_t 数据结构；就当前 ebitmap_node_t 数据结构，遍历其中的所有位。如果被置位，说明当前 block/decl 定义或引用了相应的标识符，那么当前模块 policy_module_t.map[i][j]中的相应元素的值一定不为 0（为在 xxx_copy_callback 函数中确定的在 base 模块中的新的 policy value）。

使用 assert 宏确认，新位图中以相应标识符的(新 policy value - 1)为索引，相应位置置位。

下面确定目的 scope_index_t.class_perms_map[]数组的长度:

```
                        if (i == SYM_CLASSES &&
                            largest_mapped_class_value < module->map[SYM_CLASSES][j]) {
                                largest_mapped_class_value = module->map[SYM_CLASSES][j];
                        }
```

largest_mapped_class_value 用于保存当前模块中所使用的 class 在 base 中的最大 policy value 值。注意 scope_index_t.class_perms_map 所指向的 ebitmap 位图数组用 class 的(policy value - 1)来索引，其长度也由当前 block/decl 中所使用的 class 的 policy value 的最大值（而不是 class 个数）决定（因为 class_perms_map 指向的是位图，而并不知道和该位图相关的 class 的 policy value），参见 add_perm_to_class 函数。

```
                }
        }
}

/* next copy the enabled permissions data  */
if ((dest->class_perms_map = malloc(largest_mapped_class_value *
                                    sizeof(*dest->class_perms_map))) == NULL) {
        goto cleanup;
}
for (i = 0; i < largest_mapped_class_value; i++) {
        ebitmap_init(dest->class_perms_map + i);
}
dest->class_perms_len = largest_mapped_class_value;
```

得到当前 block/decl 中所使用的 class 的在 base 模块中的 policy value 的最大值后，就可以分配 ebitmap 数组并初始化。

```
for (i = 0; i < src->class_perms_len; i++) {
```

```
                ebitmap_t *srcmap = src->class_perms_map + i;

                ebitmap_t *destmap = dest->class_perms_map + module->map[SYM_CLASSES][i] - 1;
                ebitmap_for_each_bit(srcmap, node, j) {
                        if (ebitmap_node_get_bit(node, j) &&
                            ebitmap_set_bit(destmap, module->perm_map[i][j] - 1, 1)) {
                                goto cleanup;
                        }
                }
        }
```

最后遍历 src class_perms_map 数组，就其中的每一个 ebitmap 找到在 dest class_perms_map 数组中对应的 ebitmap，然后将 src ebitmap 中的每一个非 0 位，经转换后得到新的“坐标”，设置 dest ebitmap 的相应位。

注意，非 base 模块通常不许可定义新的 class，只能引用 base 模块中定义的 class。非 base 模块内 class 的 policy value 根据模块的 p_classes.nprim 顺序地分配（数值较小），而当前模块的 policy_module_t.map[SYM_CLASSES][]数组的长度即由 p_classes.nprim 确定，所以对于当前模块而言，class_perms_map[]数组中不会包含尚未使用的元素。

然而，由于 base 模块中会定义众多 class，因此该 class 的新 policy value 的数值较大，所以复制后的 class_perms_map[]数组中反倒很有可能出现没有使用的元素。

```
        return 0;

      cleanup:
        ERR(state->handle, "Out of memory!");
        return -1;
}
```

copy_cond_list 函数用于复制当前 decl->cond_list 队列中的所有元素，并加入 new_decl->cond_list 队列。

```
[semodule_link > semodule_link_packages > link_modules > copy_module > copy_avrule_block > copy_avrule_decl
> copy_cond_list]

static int copy_cond_list(cond_node_t * list, cond_node_t ** dst,
                          policy_module_t * module, link_state_t * state)
{
        unsigned i;
        cond_node_t *cur, *new_node = NULL, *tail;
        cond_expr_t *cur_expr;

        tail = *dst;
        while (tail && tail->next)
                tail = tail->next;
```

首先到达 new_decl->cond_list 队列的末尾，以便向其追加新的元素。

然后在循环中逐一处理当前 decl->cond_list 队列中的每一个元素。

```
        cur = list;
        while (cur) {
                new_node = (cond_node_t *)malloc(sizeof(cond_node_t));
                if (!new_node) {
                        goto cleanup;
```

```
		}
		memset(new_node, 0, sizeof(cond_node_t));

		new_node->cur_state = cur->cur_state;
		new_node->expr = cond_copy_expr(cur->expr);
		if (!new_node->expr)
			goto cleanup;
```

分配新的 cond_node_t 数据结构，并拷贝 cur_state 域和 expr 队列。由于在 expr 队列中含有 boolean 标识符，所以必须将相应 cond_expr_t 数据结构中的 bool 域修正为该 boolean 在 base 模块中新的 policy value：

```
		/* go back through and remap the expression */
		for (cur_expr = new_node->expr; cur_expr != NULL; cur_expr = cur_expr->next) {
			/* expression nodes don't have a bool value of 0 - don't map them */
			if (cur_expr->expr_type != COND_BOOL)
				continue;
			assert(module->map[SYM_BOOLS][cur_expr->bool - 1] != 0);
			cur_expr->bool = module->map[SYM_BOOLS][cur_expr->bool - 1];
		}
```

显然，描述运算符的 cond_expr_t.bool 域不被使用，只需要转换 COND_BOOL 的 cond_expr_t.bool 域。

```
		new_node->nbools = cur->nbools;
		/* FIXME should COND_MAX_BOOLS be used here? */
		for (i = 0; i < min(cur->nbools, COND_MAX_BOOLS); i++) {
			uint32_t remapped_id = module->map[SYM_BOOLS][cur->bool_ids[i] - 1];
			assert(remapped_id != 0);
			new_node->bool_ids[i] = remapped_id;
		}
		new_node->expr_pre_comp = cur->expr_pre_comp;
```

类似地，还需要修正 bool_ids[]数组中记录的 bool 的 policy value。注意该数组最多容纳 COND_MAX_BOOL 个元素，而 nbools 为 boolean 总个数，所以 bool_ids[]数组中元素的实际数量为二者的较小值。

原样复制 nbools 和 expr_pre_comp 域。

```
		if (copy_avrule_list (cur->avtrue_list, &new_node->avtrue_list, module, state)
		    || copy_avrule_list(cur->avfalse_list, &new_node->avfalse_list, module, state)) {
			goto cleanup;
		}
```

通过 copy_avrule_list 函数，复制整个 avture_list/avfalse_list 队列中的各个元素。注意此期间发生标识符 policy value 的转换。

至此 new_node 已经完成复制，将其加入 new_decl->cond_list 队列的末尾。

```
		if (*dst == NULL) {
			*dst = new_node;
		} else {
			tail->next = new_node;
		}
		tail = new_node;
		cur = cur->next;
	}
```

```
        return 0;
      cleanup:
        ERR(state->handle, "Out of memory!");
        cond_node_destroy(new_node);
        free(new_node);
        return -1;

}
```

enable_avrule 函数判断 base 模块 global 队列中每个 block 的 decl 的外部依赖是否被满足，如果一个 block 的 branch_list 队列上只有一个 decl，如果它的外部依赖可全被满足，则设置 block->enable 指向它，且其 enabled 标志被置位；如果只有一个 decl 且外部依赖无法被满足，则报错退出。如果一个 block 的 branch_list 队列上有两个 decl，则根据第一个 decl（non-else）的外部依赖是否满足来决定标识 non-else 分支还是 else 分支的 decl 为有效。

```
[semodule_link > semodule_link_packages > link_modules > enable_avrules]

/* Enable all of the avrule_decl blocks for the policy. This simple
 * algorithm is the following:
 *
 * 1) Enable all of the non-else avrule_decls for all blocks.
 * 2) Iterate through the non-else decls looking for decls whose requirements
 *    are not met.
 *    2a) If the decl is non-optional, return immediately with an error.
 *    2b) If the decl is optional, disable the block and mark changed = 1
 * 3) If changed == 1 goto 2.
 * 4) Iterate through all blocks looking for those that have no enabled
 *    decl. If the block has an else decl, enable.
 *
 * This will correctly handle all dependencies, including mutual and
 * circular. The only downside is that it is slow.
 */
static int enable_avrules(link_state_t * state, policydb_t * pol)
{
        int changed = 1;
        avrule_block_t *block;
        avrule_decl_t *decl;
        missing_requirement_t req;
        int ret = 0, rc;

        if (state->verbose) {
                INFO(state->handle, "Determining which avrules to enable.");
        }

        /* 1) enable all of the non-else blocks */
        for (block = pol->global; block != NULL; block = block->next) {
                block->enabled = block->branch_list;            # 指向当前 block 中有效的惟一 decl
                block->enabled->enabled = 1;                    # 生效的 decl 的 enabled 标志被设置
                for (decl = block->branch_list->next; decl != NULL; decl = decl->next)
                        decl->enabled = 0;                      # branch_list 中其它元素为无效的 else 分支
        }
```

link_modules > copy_module 完成后所有模块的所有 block/decl 都被复制到 base.global 队列。遍历该队列，将每个 block 的 branch_list 队列上的第一个 avrule_decl_t 认为是有效的，将其 enabled 标志位置位，并且认为 branch_list 队列中后继 avrule_decl_t 都是无效的。

体会，由此可见一个 block 如果是 optional block，则 non-else 分支为 branch_list 队列的第一个元素。剩余的元素均为 else 分支。且默认地认为 non-else 分支是有效的，而众多 else 分支是无效的。

```
/* 2) Iterate */
while (changed) {
        changed = 0;
```

上面将所有 block 的 branch_list 队列中的第一个 decl 认为是有效的。如果不是这样，则 changed 会被重新设置为 1，表示需要重新检查整个 global 队列中的所有 block。这是因为一旦某个 non-else 的 optional block/decl 的外部依赖无法被满足，则它就不能够提供有效标识符的定义了（在标识符的 scope_datum_t.decl_ids[]中记录了该 block/decl 的 decl_id，但是其 enabled 标志为 0，进而影响 is_id_enabled 中的判断），那么所有引用了这些标识符的 block/decl 也就面临无效的可能，所以必须检查！

```
for (block = pol->global; block != NULL; block = block->next) {
        if (block->enabled == NULL) {
                continue;
        }
```

内层循环逐一遍历 global 队列上的所有 block，每个 block 内当前生效的 decl 由 enable 指向。如果当前 block 没有任何有效的 decl，则继续下一个循环。否则，获得当前 block 的 branch_list 队列上的第一个 decl（由上文可见，它就应该是由 enable 指向的那个惟一有效的 non-else 分支），继续检查其外部依赖能否被满足：

```
        decl = block->branch_list;
        if (state->verbose) {
                char *mod_name = decl->module_name ? decl->module_name : "BASE";
                INFO(state->handle, "check module %s decl %d\n", mod_name, decl->decl_id);
        }
        rc = is_decl_requires_met(state, decl, &req);
        if (rc < 0) {
                ret = SEPOL_ERR;
                goto out;
        } else if (rc == 0) {
                decl->enabled = 0;
                block->enabled = NULL;
                changed = 1;
                if (!(block->flags & AVRULE_OPTIONAL)) {
                        print_missing_requirements(state, block, &req);
                        ret = SEPOL_EREQ;
                        goto out;
                }
        }
```

调用 is_decl_requires_met 函数检查当前认为是有效的 decl 的外部依赖是否都满足。如果不满足，则清除 avrule_decl_t.enabled 标志，并设置 block->enable 为 NULL，changed = 1，从而使能下一轮外部循环。

注意，对于 optional block 如果其 non-else decl 的外部依赖不满足，则使能 else decl（branch_list 队列的第 2 个元素），参见下文。对于 unconditional block 的惟一 decl，如果其外部依赖不满足则报错退出。

```
        } // for
} // while

/* 4) else handling
```

```
         *
         * Iterate through all of the blocks skipping the first (which is the
         * global block, is required to be present, and cannot have an else).
         * If the block is disabled and has an else decl, enable that.
         *
         * This code assumes that the second block in the branch list is the else
         * block. This is currently supported by the compiler.
         */
        for (block = pol->global->next; block != NULL; block = block->next) {
                if (block->enabled == NULL) {
                        if (block->branch_list->next != NULL) {
                                block->enabled = block->branch_list->next;
                                block->branch_list->next->enabled = 1;
                        }
                }
        }
```

最后重新遍历 global 队列，从第二个 block 开始，如果 block->enable 为 NULL 且它有 else 分支，则使能该分支。

注意 global 队列中第一个 block 为当前模块的 global/unconditional block，没有 else 分支，branch_list 队列中只有一个元素。

体会，由此可见 avrule_block_t.branch_list 队列上至多有两个分支。如果有两个，则 avrule_block_t.flags 中 AVRULE_OPTIONAL 标志有效，且第一个为 non-else block/decl，另外一个为 else block/else。

```
      out:
        if (state->verbose)
                debug_requirements(state, pol);

        return ret;
}
```

avrule_decl_t.required 域为 scope_index_t 数据结构，它包含 7 个 ebitmap 分别描述在当前 decl 内引用的相应类型的标识符（除 SYM_COMMON 之外，只能在 base 中定义），另外 class_perms_map[]数组以当前 decl 所引用的 class 的(policy value - 1)为索引，相应的 ebitmap 描述当前 decl 所引用的该 class 的具体 permission。

is_decl_requires_met 函数检查当前 decl 的 required 中所有引用的标识符，是否在 base 模块中存在至少一个有效的定义，即当前 decl 的外部依赖是否能被满足。如果是则返回 1，否则返回 0 并填充 missing_requirement 数据结构标识未被定义的标识符的类型和 policy value。

具体的检查操作主要由 is_id_enabled 函数完成，对于 class 标识符，还的进一步检查当前 decl 所涉及的 permission 是否被 class 所定义。

```
[semodule_link > semodule_link_packages > link_modules > enable_avrules > is_decl_requires_met]

/* Check if the requirements are met for a single declaration.  If all
 * are met return 1.  For the first requirement found to be missing,
 * if 'missing_sym_num' and 'missing_value' are both not NULL then
 * write to them the symbol number and value for the missing
 * declaration.  Then return 0 to indicate a missing declaration.
 * Note that if a declaration had no requirement at all (e.g., an ELSE
 * block) this returns 1. */
```

```
static int is_decl_requires_met(link_state_t * state, avrule_decl_t * decl,
                                struct missing_requirement *req)
{
        /* (This algorithm is very unoptimized.  It performs many
         * redundant checks.  A very obvious improvement is to cache
         * which symbols have been verified, so that they do not need
         * to be re-checked.) */
        unsigned int i, j;
        ebitmap_t *bitmap;
        char *id, *perm_id;
        policydb_t *pol = state->base;
        ebitmap_node_t *node;

        /* check that all symbols have been satisfied */
        for (i = 0; i < SYM_NUM; i++) {
                if (i == SYM_CLASSES) {
                        /* classes will be checked during permissions checking phase below */
                        continue;
                }
```

7 种标识符，首先检查除了 SYM_CLASSES 外的 6 种。

```
                bitmap = &decl->required.scope[i];
                ebitmap_for_each_bit(bitmap, node, j) {
                        if (!ebitmap_node_get_bit(node, j)) {
                                continue;
                        }

                        /* check base's scope table */
                        id = pol->sym_val_to_name[i][j];
                        if (!is_id_enabled(id, state->base, i)) {
                                /* this symbol was not found */
                                if (req != NULL) {
                                        req->symbol_type = i;
                                        req->symbol_value = j + 1;
                                }
                                return 0;
                        }
```

获得当前类型标识符的 ebitmap 数据结构，其中每一个非 0 位都描述了被当前 decl 所引用的一个标识符的(policy value - 1)，调用 is_id_enabled 函数检查 base 模块的相应 scope[SYM_XXX]符号表获得该标识符的 scope_datum_t，如果类型不是 SCOPE_DECL，则说明为定义；否则继续检查是否至少存在一个定义了该标识符的有效的 decl，如果存在则返回 1 否则返回 0。（遍历 scope_datum_t.decl_ids[]数组，就每个元素索引当前模块的 decl_val_to_struct[]数组获得相应 decl 的 avrule_decl_t 数据结构的地址，再查询其 enabled 标志位）

如果当前引用在 base 模块中尚未被定义，则以 0 返回，同时返回当前引用（标识符）的类型和 policy value。

```
                }
        }
```

至此，当前 avrule_decl_t.required 中除了 SYM_CLASSES 类型之外所有外部引用都能被满足，下面结合 class_perms_map[]数组验证对 class 的外部依赖是否能够被满足：

```
        /* check that all classes and permissions have been satisfied */
        for (i = 0; i < decl->required.class_perms_len; i++) {
```

```
bitmap = dec1->required.class_perms_map + i;
```

外层循环处理 class_perms_map[]数组中的一个 ebitmap 元素，注意该数组以 class 的 policy value 为索引。

```
ebitmap_for_each_bit(bitmap, node, j) {
        struct find_perm_arg fparg;
        class_datum_t *cladatum;
        uint32_t perm_value = j + 1;
        scope_datum_t *scope;

        if (!ebitmap_node_get_bit(node, j)) {
                continue;
        }
```

内层循环处理当前 ebitmap 中所有非 0 位置，当前非 0 位置所对应的 permission 的 policy value 用 perm_value 表示。一个有效的(i，j)组合，即表示当前 dec1 所引用了 policy value == i 的 class 中 policy value == j + 1 的那个 permission。

```
id = pol->p_class_val_to_name[i];
cladatum = pol->class_val_to_struct[i];
```

获得 policy value == i 的 class 的名称字符串和 class_datum_t 数据结构。下面就需要检查这个 class 中 policy value == j + 1 的那个 permission 是否是有效的。

```
scope = hashtab_search(state->base->p_classes_scope.table, id);
if (scope == NULL) {
        ERR(state->handle, "Could not find scope information for class %s", id);
        return -1;
}

fparg.valuep = perm_value;              # permission's policy value
fparg.key = NULL;                       # permission 的名称字符串（返回参数）

hashtab_map(cladatum->permissions.table, find_perm, &fparg);
if (fparg.key == NULL && cladatum->comdatum != NULL)
        hashtab_map(cladatum->comdatum->permissions. table, find_perm, &fparg);
perm_id = fparg.key;
assert(perm_id != NULL);
```

通过 find_perms 函数遍历该 class 的 class_datum_t.permissions 符号表，或者共享的 common 的 permissions 符号表，通过 fparg.key 返回指定 policy value 的 permission 的名称字符串。

注意，使用 assert 宏检查当前 class 一定定义了该 permission（由 link 过程保证？）

```
if (!is_perm_enabled(id, perm_id, state->base)) {
        if (req != NULL) {
                req->symbol_type = SYM_CLASSES;
                req->symbol_value = i + 1;
                req->perm_value = perm_value;
        }
        return 0;
}
```

最后通过 is_perm_enabled 函数检查当前 class 是否定以以 perm_id 命名的 permission，并且借助 is_id_enabled 函数检查 class 是否在 base 模块中存在至少一个有效的定义者 dec1。如果失败则返回 0.

```
                } // 遍历完 class_perms_map 中一个元素的所有非 0 位
        } // 遍历完整个 class_perms_map 数组的所有元素

        /* all requirements have been met */
        return 1;
}
```

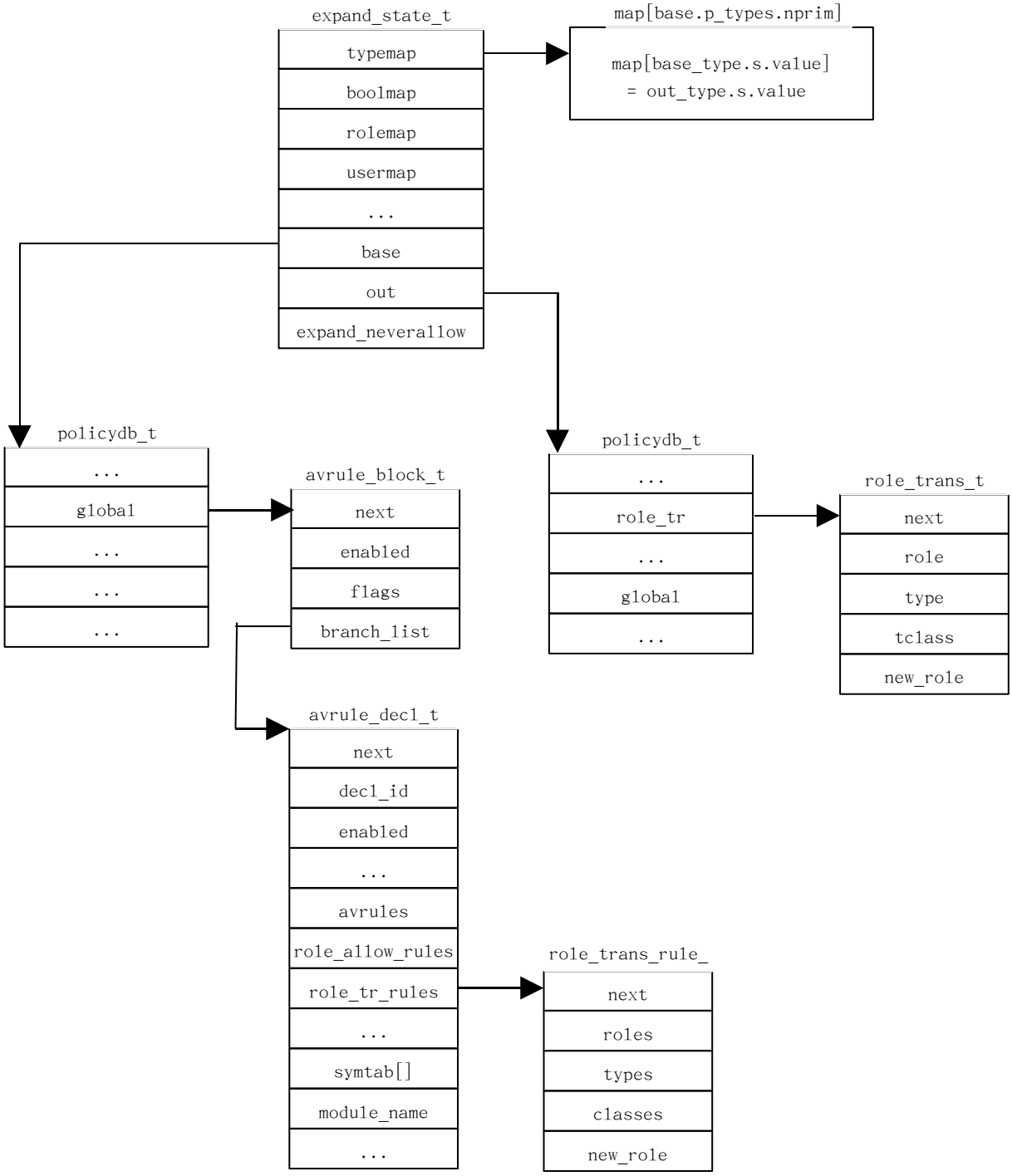
12.6 module的扩展 - semodule_expand

可以按照如下方法使用 semodule_expand 命令根据链接完毕的 base.pp 创建 policy.X 的二进制映像:

```
semodule_expand [-V -c [version]] basemodpkg outputfile
```

在模块的二进制表示中所有规则按照其字面语义进行描述，规则中某个域可能包含扩展（使用“{}”的部分）或者使用属性，而在创建 policy.X 时，必须“展开”规则中的扩展部分，并且把属性“展开”为所包含的所有 type，从而方便内核 Security server 查找相应的规则并将结果缓存在 AVC 中。

12.6.1 expand 过程核心数据结构关系图



12.6.2 type 的拷贝

在 sepol_expand_module 函数中，通过 hashtab_map 函数调用 type_copy_callback 函数来逐一处理 base 模块 p_types 符号表中的所有元素（link 结束后它包含所有模块定义的 type 标识符）。

注意 link.c 和 expand.c 文件中都定义了各自的静态 type_copy_callback 函数，它们都是把一个模块 p_types 符号表中的元素复制到另一个模块的 p_types 符号表，并且重新分配 type 的 policy value。

```
[semodule_expand > sepol_expand_module > hashtab_map > type_copy_callback]

static int type_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        int ret;
        char *id, *new_id;
        type_datum_t *type, *new_type;
        expand_state_t *state;

        id = (char *)key;
        type = (type_datum_t *)datum;
        state = (expand_state_t *)data;

        if ((type->flavor == TYPE_TYPE && !type->primary) || type->flavor == TYPE_ALIAS) {
                /* aliases are handled later */
                return 0;
        }
```

type_datum_t 中的 flavor 用于表示当前数据结构所描述的类型：普通 type，或者 alias，或者 attribute。由此可见，别名（alias）有两种表示方式。在 type_copy_callback 中跳过对 alias 的处理，因为 alias 必须等所有的 type/attribute 都被处理完，最终在 alias_copy_callback 函数中处理。

```
        if (!is_id_enabled(id, state->base, SYM_TYPES)) {
                /* identifier's scope is not enabled */
                return 0;
        }
```

在 scope_copy_callback 函数中将当前模块 scope 符号表中的一对(key，scope_datum_t)向 base 模块的相应 scope 符号表合并时，如果 base_scope->scope == SCOPE_REQ 而当前 scope->scope == SCOPE_DECL，则释放原有 base_scope->decl_ids[]数组并用 scope->decl_ids[]数组进行替换。这种逻辑将使得在 link 过程结束后，base 模块的 scope 符号表中应该只剩余 SCOPE_DECL 类型的元素，否则就说明某个被引用的符号从来没有被定义过。

is_id_enabled 函数检查 base 模块相应类型的 scope 符号表，如果下列任意条件不满足，则返回 0：
1，相应 key 对应的 scope_datum_t 数据结构存在；
2，相应 scope_datum_t.scope == SCOPE_DECL；
3，相应 scope_datum_t.decl_ids[]数组中所记录的 block/decl，任何一个的 avrule_decl_t.enabled 被置位（即，至少能找到一个被使能的、定义了该符号的 block/decl）；

```
        if (state->verbose)
                INFO(state->handle, "copying type or attribute %s", id);

        new_id = strdup(id);
        if (new_id == NULL) {
                ERR(state->handle, "Out of memory!");
                return -1;
        }
```

```
        new_type = (type_datum_t *)malloc(sizeof(type_datum_t));
        if (!new_type) {
                ERR(state->handle, "Out of memory!");
                free(new_id);
                return SEPOL_ENOMEM;
        }
        memset(new_type, 0, sizeof(type_datum_t));

        new_type->flavor = type->flavor;
        new_type->flags = type->flags;
        new_type->s.value = ++state->out->p_types.nprim;
        if (new_type->s.value > UINT16_MAX) {
                free(new_id);
                free(new_type);
                ERR(state->handle, "type space overflow");
                return -1;
        }
        new_type->primary = 1;                                  # 对于 type/attribute 而言, primary == 1
        state->typemap[type->s.value - 1] = new_type->s.value;
```

复制 base 模块当前 type 元素的 key 和 type_datum_t 数据结构。注意 new_type 的 policy value 根据 out 模块 p_types.nprim 重新开始分配（后者从 0 开始每次递增 1），并且将新旧 policy value 的转换关系记录到 expand_state.typemap[]数组中。

根据 Joshua 的 blog 这样做目的是确保 out 模块 p_types.nprim 最小化。注意 symtab_t.nprim 为当前符号表中标识符的最大 policy value（注意不是 hashtab 元素个数！后者由 symtab_t.table.nel 描述），可能存在没有被使用的数据区间。

注意，这里并没有处理 type_datum_t.types 域（描述属于一个 attribute 的所有 type），只有当 base 模块的 p_types 符号表被处理完一遍（expand_state_t.typemap[]设置好后），才能再次遍历 base 模块的 p_types 符号表，修正 attribute-type 关系。参见下文 attr_convert_callback 函数。

```
        ret = hashtab_insert(state->out->p_types.table,
                             (hashtab_key_t)new_id, (hashtab_datum_t)new_type);
        if (ret) {
                free(new_id);
                free(new_type);
                ERR(state->handle, "hashtab overflow");
                return -1;
        }
```

将复制好的 new_type 注册到 out 模块的 p_types 符号表。

```
        if (new_type->flags & TYPE_FLAGS_PERMISSIVE)
                if (ebitmap_set_bit(&state->out->permissive_map, new_type->s.value, 1)) {
                        ERR(state->handle, "Out of memory!\n");
                        return -1;
                }

        return 0;
}
```

new_type->flags 拷贝自 type->flags，如果其中 TYPE_FLAGS_PERMISSIVE 标志位有效，则以其 policy value 为索引，设置 out 模块的 permissive_map 位图中的相应位。注意，这里是把 policy value 直接记录到位图中，而不像通常那样记录(policy value - 1)。由 policydb_t 数据结构定义中的注释可见，permissive_map 的第 0 位将来可以用作 global permissive（估计起到全局 Permissive 模式的效果）。

当 base 模块的 p_types 符号表被处理完一遍后，expand_state_t.typemap[]数组设置完毕，此时就可以再次编译 base 模块的 p_types 符号表，根据 type 在 out 模块中的新 policy value 修正所有 attribute-types 关系了。

```
[semodule_expand > sepol_expand_module > hashtab_map > attr_convert_callback]

static int attr_convert_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        char *id;
        type_datum_t *type, *new_type;
        expand_state_t *state;
        ebitmap_t tmp_union;

        id = (char *)key;
        type = (type_datum_t *)datum;
        state = (expand_state_t *)data;

        if (type->flavor != TYPE_ATTRIB)
                return 0;
```

显然 attr_convert_callback 函数只处理属性，因此对于普通 type 或者 alias，直接以 0 退出。

```
        if (!is_id_enabled(id, state->base, SYM_TYPES)) {
                /* identifier's scope is not enabled */
                return 0;
        }
```

例行检查，检查当前属性是否在 base 模块中存在至少一个有效的定义者 block/decl。

```
        if (state->verbose)
                INFO(state->handle, "converting attribute %s", id);

        new_type = hashtab_search(state->out->p_types.table, id);
        if (!new_type) {
                ERR(state->handle, "attribute %s vanished!", id);
                return -1;
        }
```

在 expand_module 函数中调用 attr_convert_callback 函数之前，已经调用 type_copy_callback 函数将 base 模块 p_types 符号表的所有元素拷贝到 out 模块的 p_types 符号表，且 type 标识符的新旧 policy value 转换关系由 expand_state_t.typemap[]数组记录。

```
        if (map_ebitmap(&type->types, &tmp_union, state->typemap)) {
                ERR(state->handle, "out of memory");
                return -1;
        }
```

核心操作由 map_ebitmap 函数完成，就 base 中一个属性的 type_datum_t.types 位图的非 0 位，查询 expand_state_t.typemap[]数组得到其新的 policy value，以此为坐标设置临时位图 tmp_union，最后将 tmp_union 设置到 out 中的 new_type.types 位图中。

```
        /* then union tmp_union onto &new_type->types */
        if (ebitmap_union(&new_type->types, &tmp_union)) {
                ERR(state->handle, "Out of memory!");
                return -1;
```

```
        }
        ebitmap_destroy(&tmp_union);

        return 0;
}
```

12.6.3 common 的拷贝

common 用于定义可以被 class 共享的若干权限的集合，一个 common 其实就是一个字符串的集合，集合关系用 symtab_t 符号表描述，集合中每个权限字符串都有各自的 policy value。

由于在非 base 模块中不允许定义 common，因此在 link 过程中不需要拷贝 common。但是在 expand 过程中就需要把 base 模块中的 common 符号表拷贝到 out 模块中，该任务通过 common_copy_callback 函数完成，它每次拷贝一个 common 的(key，common_datum_t)到 out 模块中：

```
        /* copy commons */
        if (hashtab_map (state.base->p_commons.table, common_copy_callback, &state)) {
                goto cleanup;
        }

[sepol_expand_module > expand_module > hashtab_map > common_copy_callback]

static int common_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        int ret;
        char *id, *new_id;
        common_datum_t *common, *new_common;
        expand_state_t *state;

        id = (char *)key;
        common = (common_datum_t *)datum;
        state = (expand_state_t *)data;

        if (state->verbose)
                INFO(state->handle, "copying common %s", id);

        new_common = (common_datum_t *)alloc(sizeof(common_datum_t));
        if (!new_common) {
                ERR(state->handle, "Out of memory!");
                return -1;
        }
        memset(new_common, 0, sizeof(common_datum_t));
        if (symtab_init(&new_common->permissions, PERM_SYMTAB_SIZE)) {
                ERR(state->handle, "Out of memory!");
                free(new_common);
                return -1;
        }
```

首先创建一个新的 common_datum_t 数据结构并初始化，注意其中 symtab 符号表含有 32 个指针数组。

```
        new_id = strdup(id);
        if (!new_id) {
                ERR(state->handle, "Out of memory!");
                free(new_common);
                return -1;
        }
```

```
        new_common->s.value = common->s.value;
        state->out->p_commons.nprim++;
```

然后复制 common 的名称字符串，并确定 new_common 的 policy value - 注意它等于原来 common 的 policy value！（而不是基于 out 模块的 p_commons.nprim 开始分配，否则在 expand_state_t 中就必须设计 commonmap[]转换数组了！）

注意将 out 模块的 p_commons.nprim 递增 1。

```
        ret = hashtab_insert(state->out->p_commons.table, new_id, (hashtab_datum_t *)new_common);
        if (ret) {
                ERR(state->handle, "hashtab overflow");
                free(new_common);
                free(new_id);
                return -1;
        }
```

把(key, new_common)注册到 out 模块的 p_commons 符号表中。注意在 hashtab_insert 函数中只能增加 p_commons.table.nel（它描述哈希表内元素个数）。

```
        if (hashtab_map(common->permissions.table, perm_copy_callback, &new_common->permissions)) {
                ERR(state->handle, "Out of memory!");
                return -1;
        }

        return 0;
}
```

最后，处理原有 common 中 permissions 符号表，就其中的每个权限分配新的 policy value，并注册到 new_common 的 permissions 符号表中去。这个工作由 perm_copy_callback 函数完成：

```
[sepol_expand_module > expand_module > hashtab_map > common_copy_callback > perm_copy_callback]

static int perm_copy_callback(hashtab_key_t key, hashtab_datum_t datum, void *data)
{
        int ret;
        char *id, *new_id;
        symtab_t *s;
        perm_datum_t *perm, *new_perm;

        id = key;
        perm = (perm_datum_t *)datum;
        s = (symtab_t *)data;
```

就 common 中的一个 perm 而言，key 为其名称字符串，而 datum 为 perm_datum_t 数据结构，其中只有一个域用于描述该 perm 的 policy value。 注意 perm 的 policy value 只在当前 common 内有效，因为它基于 common 内 permissions 符号表的 nprim 分配（参见 define_common_perms 函数）。

perm_copy_callback 函数的 data 参数指向 new_common 中的 permissions 符号表。

```
        new_perm = (perm_datum_t *)malloc(sizeof(perm_datum_t));
        if (!new_perm) {
                return -1;
        }
        memset(new_perm, 0, sizeof(perm_datum_t));

        new_id = strdup(id);
```

```
        if (!new_id) {
                free(new_perm);
                return -1;
        }
```

分配一个新的 perm_datum_t 数据结构，并复制原有 perm 的名称字符串。

```
        new_perm->s.value = perm->s.value;
        s->nprim++;
```

注意 new_perm 保留原有的 policy value，这一点和对 common 的处理相同（否则在 expand_state_t 中将需要 permmap[]数组）。

```
        ret = hashtab_insert(s->table, new_id, (hashtab_datum_t *)new_perm);
        if (ret) {
                free(new_id);
                free(new_perm);
                return -1;
        }
        return 0;
}
```

最后将(new_id, new_perm)注册到 new_common.permissions 符号表。

12.6.10 expand 过程的核心函数调用链

```
[semodule_expand > sepol_expand_module]

int sepol_expand_module(sepol_handle_t * handle, sepol_policydb_t * base,
                        sepol_policydb_t * out, int verbose, int check)
{
        return expand_module(handle, &base->p, &out->p, verbose, check);
}

/* Linking should always be done before calling expand, even if
 * there is only a base since all optionals are dealt with at link time
 * the base passed in should be indexed and avrule blocks should be
 * enabled.
 */
int expand_module(sepol_handle_t * handle, policydb_t * base, policydb_t * out, int verbose, int check)
{
        int retval = -1;
        unsigned int i;
        expand_state_t state;
        avrule_block_t *curblock;

        expand_state_init(&state);

        state.verbose = verbose;
        state.typemap = NULL;
        state.base = base;
        state.out = out;
        state.handle = handle;
```

expand 过程需要两个 policydb_t 数据结构，一个是 linked 了的 base 模块，另外一个是崭新的 out 模块，它根据 base 模块重新构建。重新构建时需要给 user/role/type/bool 标识符再次重新分配 policy

value，新旧数值的变化关系由 expand_state_t.xxxmap 数组描述。

```
if (base->policy_type != POLICY_BASE) {
        ERR(handle, "Target of expand was not a base policy.");
        return -1;
}

state.out->policy_type = POLICY_KERN;
state.out->policyvers = POLICYDB_VERSION_MAX;
```

out 所指 policydb_t 用于描述并创建 kernel policy 的二进制映像，因此设置类型为 POLICY_KERN。而且默认版本号为当前 libsepol 所支持的最大版本号 POLICYDB_VERSION_MAX。

```
/* Copy mls state from base to out */
out->mls = base->mls;
out->handle_unknown = base->handle_unknown;

/* Copy target from base to out */
out->target_platform = base->target_platform;

/* Copy policy capabilities */
if (ebitmap_cpy(&out->policycaps, &base->policycaps)) {
        ERR(handle, "Out of memory!");
        goto cleanup;
}
```

out 复制 base 的上述相关属性。

```
if ((state.typemap = (uint32_t *)calloc(state.base->p_types.nprim, sizeof(uint32_t))) == NULL) {
        ERR(handle, "Out of memory!");
        goto cleanup;
}

state.boolmap = (uint32_t *)calloc(state.base->p_bools.nprim, sizeof(uint32_t));
if (!state.boolmap) {
        ERR(handle, "Out of memory!");
        goto cleanup;
}

state.rolemap = (uint32_t *)calloc(state.base->p_roles.nprim, sizeof(uint32_t));
if (!state.rolemap) {
        ERR(handle, "Out of memory!");

state.usermap = (uint32_t *)calloc(state.base->p_users.nprim, sizeof(uint32_t));
if (!state.usermap) {
        ERR(handle, "Out of memory!");
        goto cleanup;
}
```

这里为 expand_state_t 数据结构中的 4 个 map 数组分配空间。由于数组以 base 模块中标识符的 policy value 为索引，所以数组长度应该为相应符号表中标识符的最大 policy value。

```
/* order is important - types must be first */
```

这个注释非常重要！

在一个安全上下文“user:role:type[:level/category]”中：

1，一个 SELinux User 可以扮演不同的角色，因此在 user_datum_t 中使用了 role_set_t 数据结构；一个 SELinux User 被分配特定的 MLS range，因此在 user_datum_t 中使用了 mls_semantic_range_t 数据结构；
2，一个 role 可以和不同的 type 相关联，因此在 role_datum_t 中使用了 type_set_t 数据结构；

所以，必须首先扩展 type 及 type 属性，设置完毕 expand_state_t.typemap[]之后，才能扩展 role，从而正确地扩展 role_datum_t.types，得到 expand_state_t.rolemap[]；最后才能扩展 user，从而正确地扩展 user_datum_t.roles。

另外，只有等待所有的标识符都被扩展完毕后才能够正确地处理规则和约束，比如 copy_and_expand_avrule_block 和 constraint_copy_callback 函数在 expand_module 的后部才被调用，参见下文。

```
/* copy types */
if (hashtab_map(state.base->p_types.table, type_copy_callback, &state)) {
        goto cleanup;
}

/* convert attribute type sets */
if (hashtab_map(state.base->p_types.table, attr_convert_callback, &state)) {
        goto cleanup;
}
```

如上文所述，首先需要遍历一遍 base.p_types 符号表，为其所有 type 重新分配在 out 模块中的 policy value，新旧 policy value 的转换关系由 expand_state_t.typemap[]数组描述。在得到了所有 type 的新 policy value 之后，再次遍历 base.p_types 符号表中的所有 type 属性，修正其在 out.p_types 符号表中的“副本”（counterpart）的 types 位图。

注意，对 type 属性的处理必须等待所有普通 type 都被处理完之后，从而得到它们在 out.p_types 符号表中新的 policy value，才能更新 type 属性在 out 中的副本的 types 位图。

```
/* copy commons */
if (hashtab_map(state.base->p_commons.table, common_copy_callback, &state)) {
        goto cleanup;
}
```

调用 common_copy_callback 函数，拷贝 base.p_commons 符号表中的所有(key，common_datum_t)到 out 模块的 p_commons 符号表。注意所有 common 及其中所有 perm 的 policy value 都保持不变（否则在 expand_state_t 中需要 commonmap/permmap 数组描述新旧 policy value 的转换关系）。

```
/* copy classes, note, this does not copy constraints, constraints can't be
 * copied until after all the blocks have been processed and attributes are complete */
if (hashtab_map(state.base->p_classes.table, class_copy_callback, &state)) {
        goto cleanup;
}
```

调用 class_copy_callback 函数，拷贝 base.p_classes 符号表中所有(key，class_datum_t)到 out 模块的 p_classes 符号表。注意 class 自身，私有 perm 的 policy value 般保持不变。

```
/* copy type bounds */
if (hashtab_map(state.base->p_types.table, type_bounds_copy_callback, &state))
        goto cleanup;
```

调用 type_bounds_copy_callback 函数，再次遍历 base.p_types 符号表，就 type_datum_t.bounds != 0 的元素，查询 expand_state_t.typemap[]得到其 super-type 在 out 模块中的 policy value，以此更新当前 type 在 out 模块中的 type_datum_t.bounds 域。

```
/* copy aliases */
if (hashtab_map(state.base->p_types.table, alias_copy_callback, &state))
        goto cleanup;
```

alias 有两种情况：flavor == TYPE_ALIAS，此时相关联 type 的 policy value 保存在 alias 的 primary 中，注意 alias 自身有独立的 policy value，保存在 s.value 中；或者 flavor == TYPE_TYPE 且 primary == 0，此时相关联 type 的 policy value 保存在 alias 的 s.value 中，即 alias 自身没有独立的 policy value。

对 alias 的处理要放到所有 type/attribute 处理完成后。调用 alias_copy_callback 函数，再次遍历 base.p_types 符号表，就其中的所有 alias 复制其 type_datum_t 数据结构，使用 primary（TYPE_ALIAS 类型）或者 s.value（TYPE_TYPE 类型）查询 expand_state_t.typemap[]数组得到相关联 type 在 out 模块中的 policy value 并设置到 new_alias.s.value，最后将 new_alias 插入 out 模块的 p_types 符号表。

注意，这样使得 out 模块中 alias 的 s.value 总是等于相关联的 type 的 policy value，而在 base 模块中的却不一定！另外这也要求 out 模块中 alias 的 flavor 总是 TYPE_TYPE。

```
/* index here so that type indexes are available for role_copy_callback */
if (policydb_index_others(handle, out, verbose)) {
        ERR(handle, "Error while indexing out symbols");
        goto cleanup;
}
```

out 模块的所有的符号表都创建完毕后，就可以调用 policydb_index_others 函数创建其中的 p_xxx_val_to_name[]以及 xxx_val_to_struct[]数组了，比如：

```
        p->p_type_val_to_name[typdatum->s.value - 1] = (char *)key;
        p->type_val_to_struct[typdatum->s.value - 1] = typdatum;
```

即通过标识符的 policy value 查询其名称字符串和相应的 xxx_datum_t 数据结构。

```
/* copy roles */
if (hashtab_map(state.base->p_roles.table, role_copy_callback, &state))
        goto cleanup;
if (hashtab_map(state.base->p_roles.table, role_bounds_copy_callback, &state))
        goto cleanup;
```

调用 role_copy_callback 函数，遍历 base.p_roles 符号表，就每一个元素的 key 查找 out 模块 p_roles 符号表，如果 out 中和 key 对应的 role_datum_t 不存在则复制(key，role_datum_t)并注册到 out 模块 p_roles 符号表；如果已经存在，则将当前 role 中的 dominates/types 位图合并到其 out 中副本的相应位图中。

注意：
1，由于 expand_state_t.typemap[]事先已经创建，在 role_copy_callback > expand_convert_type_set 函数中就已经修正了 types 位图。而 dominates 位图的修正要延迟到所有 block/decl 中的 p_roles 符号表也被合并到 out 后才能进行。

2，如果当前 role 的 key 为“object_r”，则只设置 expand_state_t.rolemap[role->s.value - 1] = 1 就直接退出了。即对于“object_r”，在 out 模块中不存在 role_datum_t 数据结构，因为在 SELinux 内核驱动中检查 SC 有效性时将跳过 object_r 相关的所有 user-role，role-type，user-mls 检查。

```
/* copy MLS's sensitivity level and categories - this needs to be done
 * before expanding users (they need to be indexed too) */
if (hashtab_map(state.base->p_levels.table, sens_copy_callback, &state))
        goto cleanup;
```

```
        if (hashtab_map(state.base->p_cats.table, cats_copy_callback, &state))
                goto cleanup;
        if (policydb_index_others(handle, out, verbose)) {
                ERR(handle, "Error while indexing out symbols");
                goto cleanup;
        }
```

然后拷贝 sensitivity level 和 category 信息，并建立相应的索引数据结构。注意，每次重复调用 policydb_index_others 函数都会释放、并重新调用所有 index_f 函数重新建立索引数据结构，显然这里有大量的重复操作。

至此，role/type/level/category 符号表都已经拷贝到 out 模块中，接下来就可以拷贝 user 符号表了：

```
        /* copy users */
        if (hashtab_map(state.base->p_users.table, user_copy_callback, &state))
                goto cleanup;
        if (hashtab_map(state.base->p_users.table, user_bounds_copy_callback, &state))
                goto cleanup;

        /* copy bools */
        if (hashtab_map(state.base->p_bools.table, bool_copy_callback, &state))
                goto cleanup;

        if (policydb_index_classes(out)) {
                ERR(handle, "Error while indexing out classes");
                goto cleanup;
        }
```

(TODO: 补充关于拷贝上述类型标识符的内容)

```
        if (policydb_index_others(handle, out, verbose)) {
                ERR(handle, "Error while indexing out symbols");
                goto cleanup;
        }

        /* loop through all decls and union attributes, roles, users */
        for (curblock = state.base->global; curblock != NULL; curblock = curblock->next) {
                avrule_decl_t *decl = curblock->enabled;

                if (decl == NULL) {
                        /* nothing was enabled within this block */
                        continue;
                }

                /* convert attribute type sets */
                if (hashtab_map(decl->p_types.table, attr_convert_callback, &state)) {
                        goto cleanup;
                }

                /* copy roles */
                if (hashtab_map(decl->p_roles.table, role_copy_callback, &state))
                        goto cleanup;

                /* copy users */
                if (hashtab_map(decl->p_users.table, user_copy_callback, &state))
                        goto cleanup;
        }
```

然后遍历所有 block/dec1 链表，调用相应的方法拷贝其中的 user/role/types 标识符到 out 模块中。一个 avrule_block_t 可能有多个 avrule_decl_t，但是只有一个被使能，由 avrule_block_t.enabled 指向。

注意，在 link 过程的后部调用 enable_avrules 函数判断一个 block 中哪个 decl 的外部依赖能够被满足，从而设置 avrule_block_t.enabled 指针及相应 avrule_decl_t.enabled 标志位。参见上文。

link 过程中各个 block/decl 中定义的标识符既被拷贝到 base 模块中相应 block/decl，也会拷贝到 base 模块的全局符号表中。后者使得 expand 过程中拷贝 base 模块的全局符号表到 out 模块时，block/decl 的私有符号表中的标识符也随即被拷贝到 out 模块。但是，由**于编译过程中使用了 get_local_type 函数，它使得 block/decl 中定义的 type 标识符的某些属性仍然被保留在 block/decl 的私有符号表中！** 比如 type_datum_t.types 位图。即，在某个用 optional_policy 宏修饰的 block 中定义的 type-attribute 关系仍然在该 block 的私有符号表中，尽管相应 attribute 标识符已经被合并到 base.p_types 中。所以必须在这里遍历所有 block/decl 的 p_types 符号表，通过 attr_convert_callback 函数将 block/decl 中的 type-attribute 关系也合并到 out.p_types 符号表中，否则将被遗漏！

一个 type 属性和其他 type 之间的相互作用关系，需要在扩展规则时传递给属于该 type 属性的所有普通 type。所以上述从 block/decl 中拷贝 type-attribute 关系的操作必须在下面调用 copy_and_expand_avrule_block 函数之前完成。

```
/* remap role dominates bitmaps */
 if (hashtab_map(state.out->p_roles.table, role_remap_dominates, &state)) {
        goto cleanup;
```

在 base.p_roles 符号表和所有 block/decl 的 p_roles 符号表都合并到 out 模块后，expand_state_t.rolemap[]数组建立完毕，此时就可以遍历 out 模块 p_roles 符号表，调用 role_remap_dominates 函数修正 sub-role 的 policy value 了。另外，如上文所述在 role_copy_callback > expand_convert_type_set 函数中已经完成了 role_datum_t.types 符号表的修正工作。

```
if (copy_and_expand_avrule_block(&state) < 0) {
        ERR(handle, "Error during expand");
        goto cleanup;
}
```

调用 copy_and_expand_avrule_block 函数，展开所有 block/decl 中所有规则的“字面”描述，生成“展开”描述并组织到 out 模块的相应数据结构中：对于各个 decl->avrules 队列中的规则，展开后被加入 te_avtab 哈希表；对于各个 decl->cond_list 中的条件规则，展开后被加入 te_cond_avtab 哈希表，并且 cond_node_t 还被加入 out 模块 policydb_t.cond_list 队列；而各个 decl 中的各种 RBAC 类规则，展开后被加入 out 模块 policydb_t 中对应的队列。参见下文。

最终根据规则的“展开”描述创建相应的二进制表示，写入 policy.X 文件。

```
/* copy constraints */
if (hashtab_map(state.base->p_classes.table, constraint_copy_callback, &state)) {
        goto cleanup;
}

cond_optimize_lists(state.out->cond_list);
evaluate_conds(state.out);
```

如上文所述，在 copy_and_expand_avrule_block 函数中，会展开各个 decl->cond_list 中的条件规则，加入 te_con_avtab 哈希表，同时各个 cond_node_t 数据结构也将加入 out->cond_list 队列。现在重新计

算该队列中各个 cond_node_t 的状态值。

```
/* copy ocontexts */
if (ocontext_copy(&state, out->target_platform))
        goto cleanup;

/* copy genfs */
if (genfs_copy(&state))
        goto cleanup;
```

（TODO：增加上述相关拷贝的分析）

expand 过程的倒数第 2 步就是创建 type<->attribute 之间的双向联系了。policydb_t 中的 type_attr_map[]和 attr_type_map[]都为 ebitmap 数组，两个数组长度均为 out 模块 p_types.nprim，因为它们都需要通过 type/attribute 的 policy value 来索引。

type_attr_map[type.s.value - 1]位图描述当前普通 type 所属的各种属性（其中非 0 位为一个属性 type 的(policy value - 1)）；attr_type_map[type.s.value - 1]位图为当前属性 type 的 type_datum_t.types 位图的拷贝。

```
/* Build the type<->attribute maps and remove attributes. */
state.out->attr_type_map = malloc(state.out->p_types.nprim * sizeof(ebitmap_t));
state.out->type_attr_map = malloc(state.out->p_types.nprim * sizeof(ebitmap_t));
if (!state.out->attr_type_map || !state.out->type_attr_map) {
        ERR(handle, "Out of memory!");
        goto cleanup;
}

for (i = 0; i < state.out->p_types.nprim; i++) {
        ebitmap_init(&state.out->type_attr_map[i]);
        ebitmap_init(&state.out->attr_type_map[i]);
        /* add the type itself as the degenerate case */
        if (ebitmap_set_bit(&state.out->type_attr_map[i], i, 1)) {
                ERR(handle, "Out of memory!");
                goto cleanup;
        }
}
```

注意即使为普通 type，也在 type_attr_map[type.s.value - 1]位图中以自己的(policy value - 1)为索引将相应位置位，即为“degenerate case”特殊情况。

```
if (hashtab_map(state.out->p_types.table, type_attr_map, &state))
        goto cleanup;
```

最终通过 type_attr_map 函数完成 type_attr_map[]和 attr_type_map[]的初始化。

```
if (check) {
        if (hierarchy_check_constraints(handle, state.out))
                goto cleanup;

        if (check_assertions(handle, state.out, state.out->global->branch_list->avrules))
                 goto cleanup;
}
```

（TODO：补充有关 check_assertions 的说明）

```
retval = 0;
```

```
cleanup:
  free(state.typemap);
  free(state.boolmap);
  free(state.rolemap);
  free(state.usermap);
  return retval;
}
```

12.6.11 展开规则的“字面”描述 - copy_and_expand_avrule_block 函数

当编译模块的某个规则时，创建该规则的“字面”描述并加入当前 block/decl 数据结构中相应的组织结构（单向链表）。link 过程结束后，base 模块 global 链表中即包含所有模块实现的全部规则。在 expand 过程中需要展开所有规则的“字面”描述，创建“展开”描述并组织在 te_avtab/te_cond_avtab 哈希表及 role_tr/role_allow/range_tr 队列中（条件规则展开描述的指针，则组织在 out->cond_list 队列中），最后根据这些组织结构中的“展开”描述创建规则在 policy.X 中的二进制表示。

另外，装载如 SELinux 内核的 policy.X 及根据它创建的 policydb_t 中都只需要包含规则的“展开”描述，这样有利于根据规则的 avtab_key_t 数据结构（specified，source_type，target_type，target_class）来查找规则哈希表得到相应的 avtab_datum_t 结果。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block]

/*
 * Expands the avrule blocks for a policy. RBAC rules are copied. Neverallow
 * rules are copied or expanded as per the settings in the state object; all
 * other AV rules are expanded.  If neverallow rules are expanded, they are not
 * copied, otherwise they are copied for later use by the assertion checker.
 */
static int copy_and_expand_avrule_block(expand_state_t * state)
{
        avrule_block_t *curblock = state->base->global;
        avrule_block_t *prevblock;
        int retval = -1;

        if (avtab_alloc(&state->out->te_avtab, MAX_AVTAB_SIZE)) {
                ERR(state->handle, "Out of Memory!");
                return -1;
        }

        if (avtab_alloc(&state->out->te_cond_avtab, MAX_AVTAB_SIZE)) {
                ERR(state->handle, "Out of Memory!");
                return -1;
        }
```

分配 out 模块的 te_avtab 和 te_cond_avtab 规则哈希表，每个元素 avtab_node_t 包含一条规则的 avtab_key_t 和 avtab_datum_t 数据结构。参见上文。

在一个循环中遍历 base 模块 global 队列中的所有 block 中被惟一使能的 decl 数据结构，展开其中所有类型规则的“字面”描述为若干“展开”描述，加入 out 模块相应组织结构中。

```
        while (curblock) {
                avrule_decl_t *decl = curblock->enabled;
                avrule_t *cur_avrule;
```

每个 avrule_block_t 可能包含若干 avrule_decl_t，但是只有一个被使能，因此只需访问由

avrule_block_t.enabled 指向的 avrule_decl_t 数据结构。

```
if (decl == NULL) {
        /* nothing was enabled within this block */
        goto cont;
}
```

如果当前 block 没有被使能的 decl，则跳转到 cont 处准备开始下一个循环。

```
/* copy role allows and role trans */
if (copy_role_allows(state, decl->role_allow_rules) != 0 ||
    copy_role_trans(state, decl->role_tr_rules) != 0) {
        goto cleanup;
}
```

调用 copy_role_allows 和 copy_role_trans 函数扩展当前 block/decl 中的 role_allow 和 role_transition 规则。这两个函数的实现方法很类似，参见下文。

```
/* expand the range transition rules */
if (expand_range_trans(state, decl->range_tr_rules))
        goto cleanup;
```

调用 expand_range_trans 函数展开 range_transition 规则，也与 copy_role_trans 函数的实现类似，不再重复。

下面就可以展开当前 decl->avrules 队列中定义的所有 AVRULE_AV/TYPE 规则了，在循环中逐一处理。注意当前 block/decl 中的 avrule_t 数据结构的 stypes/ttypes 位图仍然使用的是相应标识符在 base 模块中的 policy value，需要根据 expand_state_t.typemap 数组转换成在 out 模块中的新 policy value。而 class/permission 的 policy value 保持不变。

```
/* copy rules */
cur_avrule = decl->avrules;
while (cur_avrule != NULL) {
        if (!(state->expand_neverallow) && (cur_avrule->specified & AVRULE_NEVERALLOW)) {
                /* copy this over directly so that assertions are checked later */
                if (copy_neverallow(state->out, state->typemap, cur_avrule))
                        ERR(state->handle, "Error while copying neverallow.");
```

如果当前规则的为 neverallow 规则，且 expand_state_t.expand_neverallow == 0（表示不扩展 neverallow 规则而是直接拷贝其“字面”描述），则修正该 neverallow 规则中的 stypes/ttypes 位图，并拷贝到 out 模块 global 链表中第一个 block/decl 元素的 avrules 队列中。

否则，无论是否为 neverallow 规则都需要调用 convert_and_expand_rule 函数扩展为“展开”描述并加入 out 模块的 te_avtab 哈希表：

```
        } else {
                if (cur_avrule->specified & AVRULE_NEVERALLOW) {
                        state->out->unsupported_format = 1;
                }
                if (convert_and_expand_rule(state->handle, state->out, state->typemap,
                                            cur_avrule, &state->out->te_avtab, NULL,
                                            NULL, 0, state->expand_neverallow)
                    != EXPAND_RULE_SUCCESS) {
                        goto cleanup;
                }
        }
```

处理完 avrule_decl_t.avrules 队列中的当前元素后，准备处理下一个元素：

```
                cur_avrule = cur_avrule->next;
        }
```

现在继续处理当前 decl->cond_list 队列中的所有条件规则。所有条件规则被展开后，将注册到 out 模块的 te_cond_avtab 哈希表，同时各个 cond_node_t 数据结构（只保留 cur_state，true_list/false_list 域）同时被拷贝到 out->cond_list 队列中。参见下文。

```
        /* copy conditional rules */
        if (cond_node_copy(state, decl->cond_list))
                goto cleanup;
```

至此，当前 decl 中所有规则的队列都被处理完，准备处理下一个 block/decl。

```
cont:
        prevblock = curblock;
        curblock = curblock->next;

        if (state->handle && state->handle->expand_consume_base) {
                /* set base top avrule block in case there
                 * is an error condition and the policy needs to be destroyed */
                state->base->global = curblock;
                avrule_block_destroy(prevblock);
        }
```

如果 expand_state_t.handle 指针非空，且相应 sepol_handle_t 中 expand_consume_base 标志有效，则扩展完当前 block/decl 的规则后，将其释放（目前在 semodule_expand 程序的 main 函数中设置该表示位）。

```
  } // while(curblock)

  retval = 0;

cleanup:
  return retval;
}
```

copy_role_trans 函数用于将当前 block/decl 的 role_tr_rules 队列中所有 role_transition 规则的“字面”描述扩展为“展开”描述，加入 out 模块的 role_tr 队列。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > copy_role_trans]

static int copy_role_trans(expand_state_t * state, role_trans_rule_t * rules)
{
        unsigned int i, j, k;
        role_trans_t *n, *l, *cur_trans;
        role_trans_rule_t *cur;
        ebitmap_t roles, types;
        ebitmap_node_t *rnode, *tnode, *cnode;

        /* start at the end of the list */
        for (l = state->out->role_tr; l && l->next; l = l->next) ;
```

role_transition 规则的“字面”描述为 role_trans_rule_t 数据结构，规则中的 source_role，target_type，target_class 都使用位图数据结构（能够描述若干标识符的 policy value）；而其“展

开”描述为 role_trans_t 数据结构，规则中相应语法成分均由 uint32_t 来描述（一个具体标识符的 policy value）。

这里首先到达 out 模块 role_tr 队列的末尾，以便**追加** role_transition 规则的“展开”描述。

```
cur = rules;
while (cur) {
        ebitmap_init(&roles);
        ebitmap_init(&types);

        if (role_set_expand(&cur->roles, &roles, state->out, state->rolemap)) {
                ERR(state->handle, "Out of memory!");
                return -1;
        }
        if (expand_convert_type_set(state->out, state->typemap, &cur->types, &types, 1)) {
                ERR(state->handle, "Out of memory!");
                return -1;
        }
```

在循环中遍历当前 avrule_decl_t.role_tr_rules 队列中的所有 role_trans_rule_t 数据结构。注意字面描述 roles/types 位图中仍然使用的是相应标识符在 base 模块中的 policy value，在展开时必须查找 expand_state_t.rolemap/typemap 数组，获得在 out 模块中的 policy value。

转换后的 source_role，target_type 保存在 roles/types 位图中，它们都是局部变量。注意在 expand 过程中 class/common 等标识符的 policy value 保持不变。

```
        ebitmap_for_each_bit(&roles, rnode, i) {
                if (!ebitmap_node_get_bit(rnode, i))
                        continue;
                ebitmap_for_each_bit(&types, tnode, j) {
                        if (!ebitmap_node_get_bit(tnode, j))
                                continue;
                        ebitmap_for_each_bit(&cur->classes, cnode, k) {
                                if (!ebitmap_node_get_bit(cnode, k))
                                        continue;
```

此段为展开的核心逻辑，在三层循环中确定 roles，types，classes 位图中非 0 位置的所有**组合**方式，最内层循环即为一组有效的 source_type，target_type，target_class 的 policy value 组合(i，j，k)。

```
                                        cur_trans = state->out->role_tr;
                                        while (cur_trans) {
                                                unsigned int out_role;
                                                out_role = state->rolemap[cur->new_role - 1];

                                                if ((cur_trans->role == i + 1) &&
                                                        (cur_trans->type == j + 1) &&
                                                        (cur_trans->tclass == k + 1)) {
                                                        if (cur_trans->new_role == out_role) {
                                                                break;
                                                        } else {
                                                                ERR(state->handle,
                                                                 "Conflicting role trans rule %s
%s : %s { %s vs %s}",

                                                                 state->out->p_role_val_to_name[i],
                                                                 state->out->p_type_val_to_name[j],
                                                                 state->out->p_class_val_to_name[k],
```

```
                                                        state->out-
>p_role_val_to_name[out_role - 1]);
                                                        state->out-
>p_role_val_to_name[cur_trans->new_role - 1]);
                                                       return -1;
                                                }
                                        }
                                        cur_trans = cur_trans->next;
                                }
                                if (cur_trans)
                                        continue;
```

就当前(i，j，k)组合，查询 out 模块 role_tr 队列中是否存在相同的元素。如果是，则 break 出 while 循环，并在这里使用 continue 结束最内层循环（因为相同的展开描述已存在）。同时还能够检查是否存在相互冲突的规则定义，如果存在则直接报错退出。

注意，在将当前 block/decl 中定义的 role_transition 规则的展开描述和 out 模块中 role_tr 队列中的展开描述比较时，必须都使用相应标识符在 out 模块中的 policy value！上面的 i/j 都是转换过的，但是在比较 new_role 域时，也必须首先将当前模块的 cur->new_role 转换为在 out 中的新 policy value!

```
                                n = (role_trans_t *)malloc(sizeof(role_trans_t));
                                if (!n) {
                                        ERR(state->handle, "Out of memory!");
                                        return -1;
                                }
                                memset(n, 0, sizeof(role_trans_t));
                                n->role = i + 1;
                                n->type = j + 1;
                                n->tclass = k + 1;
                                n->new_role = state->rolemap[cur->new_role - 1];
                                if (l)
                                        l->next = n;
                                else
                                        state->out->role_tr = n;

                                l = n;
```

否则，就说明不存在和当前(i，j，k)组合相同的“展开”描述，于是分配一个新的 role_trans_t 数据结构，经过指针 l 组织在 out 模块的 role_tr 队列里。

注意，i/j/k 都已经是相应标识符在 out 模块中的新 policy value 了，而 new_role 还需要查询 expand_state_t.rolemap[]来修正。

```
                                        }
                                }
                        }

                ebitmap_destroy(&roles);
                ebitmap_destroy(&types);

                cur = cur->next;
        }
        return 0;
}
```

当前 role_trans_rule_t 数据结构扩展完后，递进 cur 指针指向当前 avrule_decl_t.role_tr_rules 队列的下一个元素。

在 expand 过程中通过 convert_and_expand_rule 函数，将 base 模块中一个 block/dec1 的当前 avrule_t 数据结构扩展为“展开”描述，并加入 out 模块中 te_avtab 规则哈希表：

```
        if (convert_and_expand_rule(state->handle, state->out, state->typemap,
                                    cur_avrule, &state->out->te_avtab, NULL,
                                    NULL, 0, state->expand_neverallow) != EXPAND_RULE_SUCCESS) {
                goto cleanup;
        }
```

尽管参数 enabled == 0，即所有加入 out 模块 te_avtab 哈希表的 avtab_node_t 元素的 avtab_key_t.specified 中将会清除 AVTAB_ENABLED 标志位。但实际上只有对于从 te_cond_avtab 哈希表中匹配的规则才需要检查该标志位（对 te_avtab 中的规则，不使用该标志），参见 security_compute_sid 函数。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > convert_and_expand_rule]

/*
 * Expand a rule into a given avtab - checking for conflicting type
 * rules in the destination policy.  Return EXPAND_RULE_SUCCESS on
 * success, EXPAND_RULE_CONFLICT if the rule conflicts with something
 * (and hence was not added), or EXPAND_RULE_ERROR on error.
 */
static int convert_and_expand_rule(sepol_handle_t * handle, policydb_t * dest_pol, uint32_t * typemap,
                                   avrule_t * source_rule, avtab_t * dest_avtab,
                                   cond_av_list_t ** cond, cond_av_list_t ** other, int enabled,
                                   int do_neverallow)
{
        int retval;
        ebitmap_t stypes, ttypes;
        unsigned char alwaysexpand;

        if (!do_neverallow && (source_rule->specified & AVRULE_NEVERALLOW))
                return EXPAND_RULE_SUCCESS;
```

参数 do_neverallow 表示是否处理 neverallow 规则。如果当前规则为 neverallow 且 do_neverallow == 0，则不进一步处理而直接返回成功。

```
        ebitmap_init(&stypes);
        ebitmap_init(&ttypes);

        /* Force expansion for type rules and for self rules. */
        alwaysexpand = ((source_rule->specified & AVRULE_TYPE) || (source_rule->flags & RULE_SELF));
```

alwaysexpand 标志位表示当前规则是否为三种 type 规则（type_transition，type_change，type_member）的一种，或者在 target_type 上使用了“self”（无论是 AV 规则或者三种 type 规则）。如果 alwaysexpand == 1，则总是展开。

```
        if (expand_convert_type_set(dest_pol, typemap, &source_rule->stypes, &stypes, alwaysexpand))
                return EXPAND_RULE_ERROR;

        if (expand_convert_type_set (dest_pol, typemap, &source_rule->ttypes, &ttypes, alwaysexpand))
                return EXPAND_RULE_ERROR;
```

调用 expand_convert_type_set 函数，将当前规则中 stypes/ttypes 数据结构修正并展开为相应的位图：1，调用 map_ebitmap 函数，就 type_set_t.types 位图中的所有非 0 位，根据 typemap 转换得到新的非 0

位坐标，设置到 tmpset.types 位图中；
2，同理处理 type_set_t.negset 位图到 tmpset.negset 位图中；
3，调用 type_set_expand 函数，根据 alwaysexpand 标志位是否被置位，tmpset.flags 是否被设置以及 tmpset.negset 是否非空决定如何扩展 tmpset 到位图中。

```
        retval = expand_rule_helper(handle, dest_po1, typemap,
                                    source_rule, dest_avtab,
                                    cond, other, enabled, &stypes, &ttypes);
        ebitmap_destroy(&stypes);
        ebitmap_destroy(&ttypes);
        return retval;
}
```

在 convert_and_expand_rule 函数中已经把当前 AVRULE_AV 或 AVRULE_TYPE 规则 source_type，target_type 上的扩展，属性，特殊字符都处理完毕，得到相应语法位置上展开了的 stypes/ttypes 位图，现在就可以调用 expand_rule_helper 函数得到 stypes/ttypes 位图中的所有非 0 组合了。

显然需要使用两层循环，在外层循环得到 stypes 位图的一个非 0 值 i，在内层循环确定 ttypes 位图的一个非 0 值 j，(i，j)即可确定一组有效的(source_type，target_type)组合。注意，如果当前规则中在 target_type 上使用了“self”，则只需一层循环即可而无须考虑遍历 ttypes 位图的内层循环。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > convert_and_expand_rule >
expand_rule_helper]

static int expand_rule_helper(sepol_handle_t * handle, policydb_t * p, uint32_t * typemap,
                              avrule_t * source_rule, avtab_t * dest_avtab,
                              cond_av_list_t ** cond, cond_av_list_t ** other,
                              int enabled, ebitmap_t * stypes, ebitmap_t * ttypes)
{
        unsigned int i, j;
        int retval;
        ebitmap_node_t *snode, *tnode;

        ebitmap_for_each_bit(stypes, snode, i) {
                if (!ebitmap_node_get_bit(snode, i))
                        continue;
                if (source_rule->flags & RULE_SELF) {
                        if (source_rule->specified & AVRULE_AV) {
                                if ((retval = expand_avrule_helper(handle, source_rule->specified,
                                                                   cond, i, i, source_rule->perms,
                                                                   dest_avtab, enabled)) != EXPAND_RULE_SUCCESS) {
                                        return retval;
                                }
                        } else {
                                if ((retval = expand_terule_helper(handle, p, typemap,
                                                                   source_rule->specified, cond, other, i, i,
                                                                   source_rule->perms, dest_avtab,
                                                                   enabled)) != EXPAND_RULE_SUCCESS) {
                                        return retval;
                                }
                        }
                }
```

如果 avrule_t.flags 的 SELF_RULE 标志位有效，则无须进入内层循环处理 ttypes 位图，注意 if 的两个分支都会直接 return。根据规则的类型分别调用不同的函数处理，注意 stype = ttype = i；

```
                ebitmap_for_each_bit(ttypes, tnode, j) {
```

```
                        if (!ebitmap_node_get_bit(tnode, j))
                                continue;
                        if (source_rule->specified & AVRULE_AV) {
                                if ((retval = expand_avrule_helper(handle,
                                                                   source_rule->specified, cond, i, j,
                                                                   source_rule->perms, dest_avtab,
                                                                   enabled)) != EXPAND_RULE_SUCCESS) {
                                        return retval;
                                }
                        } else {

                                if ((retval = expand_terule_helper(handle, p, typemap,
                                                                   source_rule->specified, cond, other, i, j,
                                                                   source_rule->perms, dest_avtab,
                                                                   enabled)) != EXPAND_RULE_SUCCESS) {
                                        return retval;
                                }
                        }
```

思路同上，注意 stype = i; ttype = j;

```
                }
        }

        return EXPAND_RULE_SUCCESS;
}
```

expand_avrule_helper 函数根据一组确定的(source_type, target_type)，遍历当前 AV 规则的 class_perm_node_t 元素的队列，结合每一个 class_perm_node_t 元素组建一个 avtab_key_t 元素，在 avtab 哈希表中查找是否有包含该 avtab_key_t 的 avtab_node_t 数据结构。如果没有则创建并注册一个。

注意一个 class_perm_node_t 元素描述了当前 AV 规则指定的和一个 class 相关的所有 permission 位图，将其 data 合并到 avtab_node_t 中的 avtab_datum_t 中。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > convert_and_expand_rule >
expand_rule_helper > expand_avrule_helper]

static int expand_avrule_helper(sepol_handle_t * handle, uint32_t specified,
                                cond_av_list_t ** cond,
                                uint32_t stype, uint32_t ttype,
                                class_perm_node_t * perms, avtab_t * avtab, int enabled)
{
        avtab_key_t avkey;
        avtab_datum_t *avdatump;
        avtab_ptr_t node;
        class_perm_node_t *cur;
        uint32_t spec = 0;

        if (specified & AVRULE_ALLOWED) {
                spec = AVTAB_ALLOWED;
        } else if (specified & AVRULE_AUDITALLOW) {
                spec = AVTAB_AUDITALLOW;
        } else if (specified & AVRULE_AUDITDENY) {
                spec = AVTAB_AUDITDENY;
        } else if (specified & AVRULE_DONTAUDIT) {
                if (handle && handle->disable_dontaudit)
                        return EXPAND_RULE_SUCCESS;
                spec = AVTAB_AUDITDENY;
```

```
        } else if (specified & AVRULE_NEVERALLOW) {
                spec = AVTAB_NEVERALLOW;
        } else {
                assert(0);      /* unreachable */
        }
```

在规则的字面描述 avrule_t 中，规则的类型由 AVRULE_XXX 宏表示；而在规则的展开描述 avtab_node_t 中，规则的类型由 AVTAB_XXX 宏表示。这里就各种 AV 规则的类型，将 AVRULE_XXX 宏转换为对应的 AVTAB_XXX 宏。

当前(source_type，target_type)可能涉及若干 class 的一组相同的 permission。因此要将当前规则中 class 扩展部分展开，在循环中处理当前规则的 class_perm_node_t 队列的一个元素：

```
        cur = perms;
        while (cur) {
                avkey.source_type = stype + 1;
                avkey.target_type = ttype + 1;
                avkey.target_class = cur->class;                # 当前规则所涉及的一个 class 的 policy value
                avkey.specified = spec;                         # 当前规则的类型
```

注意，在规则字面描述中，用相应标识符的(policy value - 1)作为索引将位图中的相应为置位。所以获得位图非 0 位的索引后，加 1 后即为相应标识符的 policy value！

```
                node = find_avtab_node(handle, avtab, &avkey, cond);
                if (!node)
                        return EXPAND_RULE_ERROR;
```

调用 find_avtab_node 函数，在处理 decl->avrules 队列时，参数 avtab 指向 out->te_avtab 哈希表，而 cond 为 NULL；在处理 decl->cond_list 队列时，参数 avtab 指向 out->te_cond_avtab 哈希表，cond 指向当前 cond_node_t 数据结构。

find_avtab_node 函数在指定的哈希表中查找是否存在包含指定 avtab_key_t 的 avtab_node_t 元素，如果有则返回其地址；否则创建一个 avtab_node_t 并注册，最后返回其地址。

在处理条件规则时，还将创建指向 te_cond_avtab 哈希表中 avtab_node_t 的指针 cond_av_list_t 结构，相应的 cond_node_t 数据结构也被加入 out->cond_list 队列。参见下文。

```
                if (enabled) {
                        node->key.specified |= AVTAB_ENABLED;
                } else {
                        node->key.specified &= ~AVTAB_ENABLED;
                }
```

根据参数 enabled 的值设置该 avtab_node_t 元素中 avtab_key_t.specified 中的 AVTAB_ENABLED 标志位。注意，对于 out->te_avtab 中的规则，忽略该标志位；而对于 te_cond_avtab 中的规则，该标志位决定了当前规则是否有效。

注意，一个条件规则 cond_node_t 中的 avtrue_list 和 avfalse_list 中的所有规则展开后都将加入 te_cond_avtab 哈希表。运行时根据相应 boolean 状态的变化，经由 out->cond_list 中的指针，迅速定位到 out->te_cond_avtab 中的 avtab_key_t 数据结构并修改 AVTAB_ENABLED 标志位。

```
                avdatump = &node->datum;
                if (specified & AVRULE_ALLOWED) {
                        avdatump->data |= cur->data;
                } else if (specified & AVRULE_AUDITALLOW) {
```

```
                avdatump->data |= cur->data;
        } else if (specified & AVRULE_NEVERALLOW) {
                avdatump->data |= cur->data;
        } else if (specified & AVRULE_AUDITDENY) {              # 就审计的权限位置1
                /* Since a '0' in an auditdeny mask represents
                 * a permission we do NOT want to audit
                 * (dontaudit), we use the '&' operand to
                 * ensure that all '0's in the mask are
                 * retained (much unlike the allow and
                 * auditallow cases).
                 */
                avdatump->data &= cur->data;
        } else if (specified & AVRULE_DONTAUDIT) {              # 就不审计的权限位置0
                if (avdatump->data)
                        avdatump->data &= ~cur->data;
                else
                        avdatump->data = ~cur->data;
        } else {
                assert(0);      /* should never occur */
        }
```

最后将当前 class_perm_node_t.data 位图**合并**到 avtab_node_t 节点的 avtab_datum_t 中。注意具有相同 avtab_key_t 的 avtab_node_t 事先很可能已经注册过了，所以总是将当前 AV 规则涉及的 permission **合并**到已有 avtab_node_t 的 avtab_datum_t 中。

```
                cur = cur->next;
        }
        return EXPAND_RULE_SUCCESS;
}
```

expand_terule_helper 函数处理 type_transition，type_change，type_member 规则中的一种。和 AV 规则相比，type 类规则有如下不同：
1，AV 规则的最后一个域为 class 的 permission 集合，若干具有相同 avtab_key_t 的 AV 规则的最后一个域可以相互合并；而 type 规则的最后一个域为 new_type，无法合并，且不允许重复或冲突；
2，在 expand 过程中保留 class/permission 的 policy value，因此无须转换。而必须转换 type 规则中的 new_type，因此需要使用 expand_state_t.typemap 数组。

除此之外 expand_terule_helper 函数和 expand_avrule_helper 函数很类似，不再细述。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > convert_and_expand_rule >
expand_rule_helper > expand_terule_helper]

static int expand_terule_helper(sepol_handle_t * handle,
                                policydb_t * p, uint32_t * typemap, uint32_t specified,
                                cond_av_list_t ** cond, cond_av_list_t ** other,
                                uint32_t stype, uint32_t ttype,
                                class_perm_node_t * perms, avtab_t * avtab, int enabled)
{
        avtab_key_t avkey;
        avtab_datum_t *avdatump;
        avtab_ptr_t node;
        class_perm_node_t *cur;
        int conflict;
        uint32_t oldtype = 0, spec = 0;

        if (specified & AVRULE_TRANSITION) {
                spec = AVTAB_TRANSITION;
```

```
} else if (specified & AVRULE_MEMBER) {
        spec = AVTAB_MEMBER;
} else if (specified & AVRULE_CHANGE) {
        spec = AVTAB_CHANGE;
} else {
        assert(0);      /* unreachable */
}

cur = perms;
while (cur) {
        uint32_t remapped_data = typemap ? typemap[cur->data - 1] : cur->data;
        avkey.source_type = stype + 1;
        avkey.target_type = ttype + 1;
        avkey.target_class = cur->class;
        avkey.specified = spec;

        conflict = 0;
        /* check to see if the expanded TE already exists --
         * either in the global scope or in another
         * conditional AV tab */
        node = avtab_search_node(&p->te_avtab, &avkey);
        if (node) {
                conflict = 1;
        } else {
                node = avtab_search_node(&p->te_cond_avtab, &avkey);
                if (node && node->parse_context != other) {
                        conflict = 2;
                }
        }

        if (conflict) {
                avdatump = &node->datum;
                if (specified & AVRULE_TRANSITION) {
                        oldtype = avdatump->data;
                } else if (specified & AVRULE_MEMBER) {
                        oldtype = avdatump->data;
                } else if (specified & AVRULE_CHANGE) {
                        oldtype = avdatump->data;
                }

                if (oldtype == remapped_data) {
                        /* if the duplicate is inside the same scope (eg., unconditional
                         * or in same conditional then ignore it */
                        if ((conflict == 1 && cond == NULL) || node->parse_context == cond)
                                return EXPAND_RULE_SUCCESS;
                        ERR(handle, "duplicate TE rule for %s %s:%s %s",
                            p->p_type_val_to_name[avkey.source_type - 1],
                            p->p_type_val_to_name[avkey.target_type - 1],
                            p->p_class_val_to_name[avkey.target_class - 1],
                            p->p_type_val_to_name[oldtype - 1]);
                        return EXPAND_RULE_CONFLICT;
                }
                ERR(handle, "conflicting TE rule for (%s, %s:%s):  old was %s, new is %s",
                    p->p_type_val_to_name[avkey.source_type - 1],
                    p->p_type_val_to_name[avkey.target_type - 1],
                    p->p_class_val_to_name[avkey.target_class - 1],
                    p->p_type_val_to_name[oldtype - 1],
                    p->p_type_val_to_name[remapped_data - 1]);
                return EXPAND_RULE_CONFLICT;
```

```
                }

                node = find_avtab_node(handle, avtab, &avkey, cond);
                if (!node)
                        return -1;
                if (enabled) {
                        node->key.specified |= AVTAB_ENABLED;
                } else {
                        node->key.specified &= ~AVTAB_ENABLED;
                }

                avdatump = &node->datum;
                if (specified & AVRULE_TRANSITION) {
                        avdatump->data = remapped_data;
                } else if (specified & AVRULE_MEMBER) {
                        avdatump->data = remapped_data;
                } else if (specified & AVRULE_CHANGE) {
                        avdatump->data = remapped_data;
                } else {
                        assert(0);      /* should never occur */
                }

                cur = cur->next;
        }

        return EXPAND_RULE_SUCCESS;
}
```

如上文所述，expand 过程中在 copy_and_expand_avrule_block 函数中处理每一个 block 中惟一被使能的 dec1 的所有规则队列，创建规则的“展开”描述并加入 out 模块的相应队列。

cond_node_copy 函数用于处理当前 dec1->cond_list 队列中的所有条件规则，展开后加入 te_cond_avtab 哈希表，并就当前 cond_node_t 元素在 out->cond_list 中创建相应的元素，然后初始化后者中的 true_list/false_list 队列。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > cond_node_copy]

/* copy the nodes in *reverse* order -- the result is that the last
 * given conditional appears first in the policy, so as to match the
 * behavior of the upstream compiler */
static int cond_node_copy(expand_state_t * state, cond_node_t * cn)
{
        cond_node_t *new_cond, *tmp;

        if (cn == NULL) {
                return 0;
        }

        if (cond_node_copy(state, cn->next)) {
                return -1;
        }
```

根据注释，该函数倒序地处理 dec1->cond_list 中的所有 cond_node_t 元素（即后出现的规则被先处理），从而和当前编译器的行为保持一致。所以使用了递归的设计，优先处理当前 cond_node_t 元素的下一个元素，直到不存在下一个元素（即到达 dec1->cond_list 队列的末尾），此时直接以 cn == NULL 为真而返回真，然后返回到此处继续处理末尾元素。

末尾元素处理完后返回 0，然后还是退出到此处继续处理其前驱元素。如果任何一个元素处理出错，则返回错误。

```
if (cond_normalize_expr(state->base, cn)) {
        ERR(state->handle, "Error while normalizing conditional");
        return -1;
}
```

cond_normalize_expr 函数将简化当前 cond_node_t 的 expr 队列中的取反逻辑（去掉 COND_NOT 并取反 avtrue_list 和 avfalse_list），统计 expr 队列中 COND_BOOL 的个数到 nbools 中，并记录前 5 个 boolean 的 policy value 到 bool_ids[]数组中，然后计算预算值 expr_pre_comp。

```
/* create a new temporary conditional node with the booleans mapped */
tmp = cond_node_create(state->base, cn);
if (!tmp) {
        ERR(state->handle, "Out of memory");
        return -1;
}
```

调用 cond_node_create 函数创建一个临时的 cond_node_t 数据结构，复制当前 cond_node_t 的各个域。然后调用 cond_node_map_bools 函数根据 state->boolmap[]数组修正所记录的 boolean 的 policy value 为在 out 中的新值，并更新状态值。参见下文，

```
if (cond_node_map_bools(state, tmp)) {
        ERR(state->handle, "Error mapping booleans");
        return -1;
}
```

调用 cond_node_search 函数在 out->cond_list 队列中查找或者创建和 tmp 具有相同 expr 队列的 cond_node_t 数据结构（若新创建，则插入 out->cond_list 队首）。

```
new_cond = cond_node_search(state->out, state->out->cond_list, tmp);
if (!new_cond) {
        cond_node_destroy(tmp);
        free(tmp);
        ERR(state->handle, "Out of memory!");
        return -1;
}
```

创建 tmp 的目的就是为了转换 cond_node_t 中记录的 boolean 的 policy value，然后在 out->cond_list 中搜索。至此它的使命已经完成，就可以释放了。

```
cond_node_destroy(tmp);
free(tmp);
```

最后，通过 cond_avrule_list_copy 函数展开当前 cn 的 avtrue_list 和 avfalse_list 中的所有条件规则，注册到 out->te_cond_avtab 哈希表，比创建指向各个 avtab_node_t 数据结构的索引数据结构 cond_av_list_t，再组织到 new_cond 的 true_list 和 false_list 中去。

注意，由上文可知，new_cond 已经指向 out->cond_list 队列中的元素，所以这里设置的不是当前模块的 cn 的 true_list/false_list 队列，而是 out 模块中的。（cond_node_t 数据结构中的 true_list/false_list 只需要此时在 expand 过程中创建，用于写入 policy.X，在 SELinux 内核中 cond_node_t 数据结构不需要 avtrue_list 和 avfalse_list 队列，它们仅仅被模块所需要。）

注意，对于当前 cond_node_t 的 avtrue_list 队列，参数 enabled 即为当前状态值 cur_state；而对于 avfalse_list 队列，参数 enabled 则为它的**取反**。即实现根据 cond_node_t 的状态值来决定 if-else 分支中哪个分支生效。（另外，运行时根据 boolean 状态值的变化，经由 true_list/false_list 中的索引迅速定位到 te_cond_avtab 哈希表中的 avtab_key_t 数据结构，然后反转 specified 域中的 AVTAB_ENABLED 标志。）

```
        if (cond_avrule_list_copy(state->out, cn->avtrue_list, &state->out->te_cond_avtab,
                                  &new_cond->true_list, &new_cond->false_list, state->typemap,
                                  new_cond->cur_state, state))
                return -1;
        if (cond_avrule_list_copy(state->out, cn->avfalse_list, &state->out->te_cond_avtab,
                                  &new_cond->false_list, &new_cond->true_list, state->typemap,
                                  !new_cond->cur_state, state))
                return -1;

        return 0;
}
```

cond_node_create 函数创建一个新的 cond_node_t 数据结构，如果参数 node 不为 NULL，则复制 node 所指 cond_node_t 数据结构。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > cond_node_copy > cond_node_create]

cond_node_t *cond_node_create(policydb_t * p, cond_node_t * node)
{
        cond_node_t *new_node;
        unsigned int i;

        new_node = (cond_node_t *)malloc(sizeof(cond_node_t));
        if (!new_node) {
                return NULL;
        }
        memset(new_node, 0, sizeof(cond_node_t));
```

如果参数 node 为 NULL，则返回新创建的 cond_node_t。注意此时它不含有任何有意义的域（没有 expr 队列）。

否则，复制 node 所指 cond_node_t 中的各个域到新创建的 cond_node_t 中：

```
        if (node) {
                new_node->expr = cond_copy_expr(node->expr);
                if (!new_node->expr) {
                        free(new_node);
                        return NULL;
                }
                new_node->cur_state = cond_evaluate_expr(p, new_node->expr);
                new_node->nbools = node->nbools;
                for (i = 0; i < min(node->nbools, COND_MAX_BOOLS); i++)
                        new_node->bool_ids[i] = node->bool_ids[i];
                new_node->expr_pre_comp = node->expr_pre_comp;
        }

        return new_node;
}
```

注意在复制完 node->expr 队列后，就可以计算当前 cond_node_t 的状态值了。

cond_node_map_bools 函数，根据当前 state->boolmap[]数组重新映射一个 cond_node_t 中所有 boolean 的 policy value。注意它们出现于 COND_NODE 表达式以及 bool_ids[]数组中。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > cond_node_copy > cond_node_map_bools]

static int cond_node_map_bools(expand_state_t * state, cond_node_t * cn)
{
        cond_expr_t *cur;
        unsigned int i;

        cur = cn->expr;
        while (cur) {
                if (cur->bool)
                        cur->bool = state->boolmap[cur->bool - 1];
                cur = cur->next;
        }

        for (i = 0; i < min(cn->nbools, COND_MAX_BOOLS); i++)
                cn->bool_ids[i] = state->boolmap[cn->bool_ids[i] - 1];

        if (cond_normalize_expr(state->out, cn)) {
                ERR(state->handle, "Error while normalizing conditional");
                return -1;
        }

        return 0;
}
```

在根据当前 state->boolmap[]数组重新映射所有 boolean 的 policy value 后，还需要再次调用 cond_normalize_expr 函数修正当前 cond_node_t 的状态值。

在 expand 过程中 cond_node_copy 函数处理当前 dec1->cond_list 中的所有条件规则，调用 cond_avrule_list_copy 函数展开 avtrue_list 或 avfalse_list 中的规则，注册到 te_cond_avtab 哈希表，并创建相应 avtab_key_t 数据结构的索引，并加入 out->cond_list 中相应元素的 true_list 或 false_list 队列。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > cond_node_copy >
cond_avrule_list_copy]

static int cond_avrule_list_copy(policydb_t * dest_pol, avrule_t * source_rules,
                                 avtab_t * dest_avtab, cond_av_list_t ** list,
                                 cond_av_list_t ** other, uint32_t * typemap,
                                 int enabled, expand_state_t * state)
{
        avrule_t *cur;

        cur = source_rules;                     # avtrue_list 或者 avfalse_list
        while (cur) {
                if (convert_and_expand_rule(state->handle, dest_pol, typemap, cur, dest_avtab,
                                            list, other, enabled, 0) != EXPAND_RULE_SUCCESS) {
                        return -1;
                }
                cur = cur->next;
        }
```

```
        return 0;
}
```

在循环中处理 avtrue_list 或 avfalse_list 中的每一个条件规则，具体工作由 convert_and_expand_rule 函数实现，在上文已经见过。惟一需要补充的是 find_avtab_node 函数，它将已经得到的规则的展开描述，注册到相应的哈希表（te_avtab 或者 te_cond_avtab）。如果是条件规则，则创建指向其 avtabv_node_t 数据结构的索引 cond_av_list_t 数据结构，并加入 out->cond_list 中相应元素的 true_list 或 false_list 队列。

```
[sepol_expand_module > expand_module > copy_and_expand_avrule_block > convert_and_expand_rule >
expand_rule_helper > expand_avrule_helper > find_avtab_node]

/* Search for an AV tab node within a hash table with the given key.
 * If the node does not exist, create it and return it; otherwise
 * return the pre-existing one.
*/
static avtab_ptr_t find_avtab_node(sepol_handle_t * handle,
                                   avtab_t * avtab, avtab_key_t * key,
                                   cond_av_list_t ** cond)
{
        avtab_ptr_t node;
        avtab_datum_t avdatum;
        cond_av_list_t *nl;

        node = avtab_search_node(avtab, key);
```

首先在相应哈希表中查找是否有匹配 avtab_key_t 的 avtab_node_t 元素。如果不存在，则下面会创建新的元素并注册。

```
        /* If this is for conditional policies, keep searching in case
           the node is part of my conditional avtab. */
        if (cond) {
                while (node) {
                        if (node->parse_context == cond)
                                break;
                        node = avtab_search_node_next(node, key->specified);
                }
        }
```

如果参数 cond 不为 NULL，则对应条件规则（cond 指向 out->cond_list 中某个 cond_node_t 的 true_list 或 false_list 队列的地址）。**由于条件规则和非条件规则可能具有相同的 avtab_key_t，因此要继续在哈希表的冲突队列中查找属于当前 cond_node_t 的那一个 avtab_node_t 结构。**如果找到，则 break；否则 node 为 NULL。

注意，条件规则展开后注册到 te_cond_avtab 哈希表中的每一个 avtab_node_t 元素，其 parse_context 指针指向相应条件规则的 cond_node_t 的 true_list 或 false_list 队首。

如果 node 不存在，则需要创建一个新的 avtab_node_t 数据结构，并注册到相应的哈希表：

```
        if (!node) {
                memset(&avdatum, 0, sizeof avdatum);
                /* this is used to get the node - insertion is actually unique */
                node = avtab_insert_nonunique(avtab, key, &avdatum);
                if (!node) {
                        ERR(handle, "hash table overflow");
                        return NULL;
```

```
                }
```

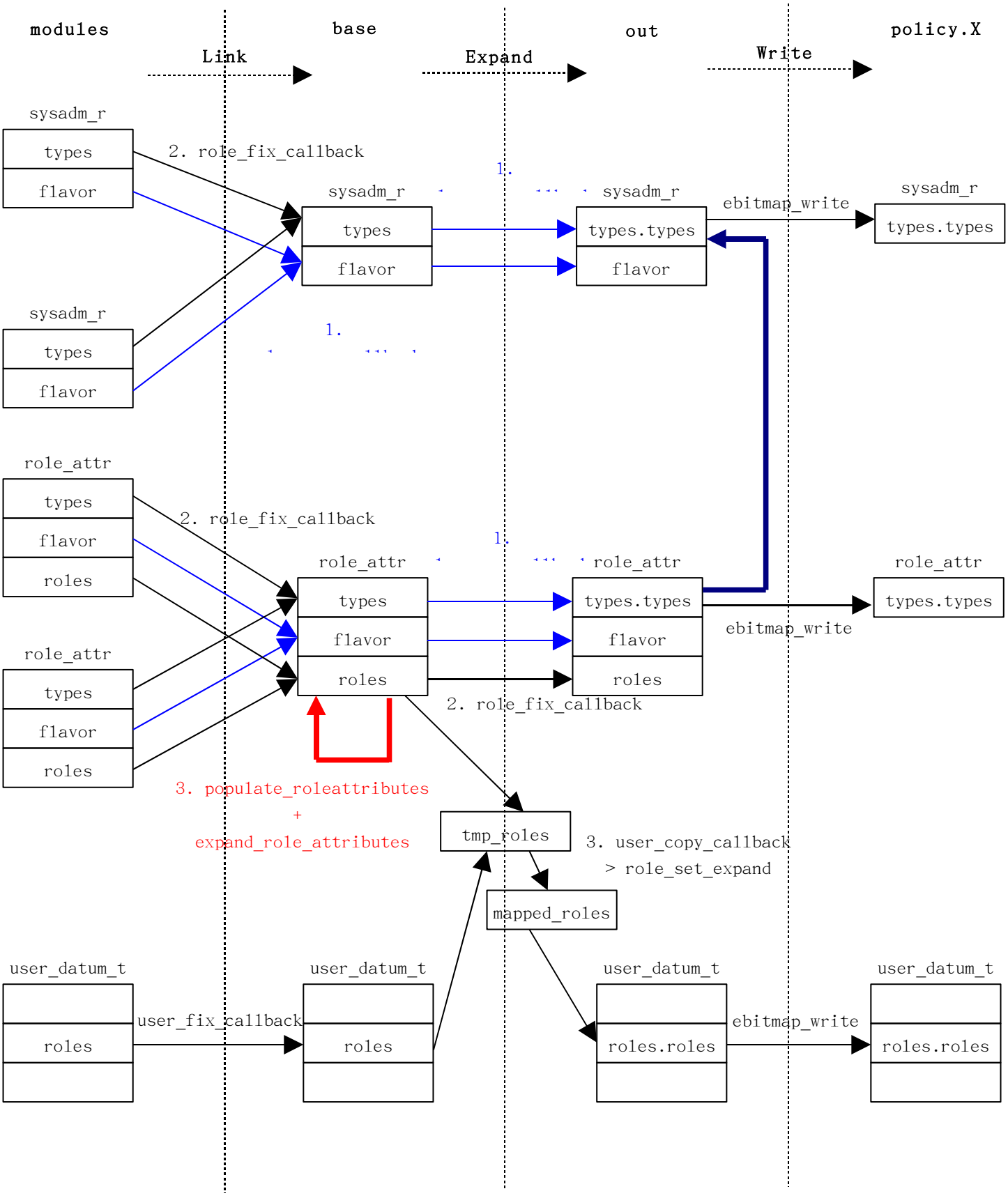
对于条件规则的情况，则进一步设置avtab_node_t.parse_context指针，指向当前true_list或false_list队首，然后创建索引cond_av_list_t数据结构，它包含指向avtab_node_t的指针。

```
                if (cond) {
                        node->parse_context = cond;
                        n1 = (cond_av_list_t *) malloc(sizeof(cond_av_list_t));
                        if (!n1) {
                                ERR(handle, "Memory error");
                                return NULL;
                        }
                        memset(n1, 0, sizeof(cond_av_list_t));
                        n1->node = node;                        # 指向当前avtab_node_t数据结构
                        n1->next = *cond;                       # 将新的索引，加入true_list/false_list队列
                        *cond = n1;
                }
        }

        return node;
}
```

12.7 link 和 expand 过程的图解（new）

至此我们已经看过了 link 和 expand 过程的部分核心源代码分析，现在让我们以 role 标识符为例，梳理一下 link 和 expand 过程中各种主要函数的调用时机和行为。

12.7.1 Role/attribute 标识符的 link 和 expand



Link 过程说明：
1，任何模块，无论定义一个标识符或者声明一个标识符，都会在自己的 p_xxx 符号表或者相应 block/dec1 的私有符号表内创建该标识符的 xxx_datum_t 数据结构。以 sysadm_r 为例，在定义或者声明时会指定它的 flavor == ROLE_ROLE。

2，模块的不同 block/dec1 或者不同的模块都会定义和 sysadm_r 相关的规则或属性，比如通过 role sysadm_r types xxx 规则指定 sysadm_r 能够和哪些 type 相结合组成合法的 SC。在 link 的过程中将 sysadm_r 的不同 role_datum_t 数据结构合并到 base.p_roles 符号表中：
1）在 role_copy_callback 中检查 flavor 是否存在冲突；
2）在 role_fix_callback > type_set_or_convert 中将 types 合并；

注意，上图中 sysadm_r 的不同 role_datum_t，有可能来自于某个模块的 p_roles 符号表（即该模块的 global block），也有可能来自于一个 block/dec1 的 avrule_dec1_t.symtab[SYM_ROLES]符号表，role_copy_callback 函数使得它无论来自于何处，都会合并到 base.p_roles 符号表中；而且如果来自于一个 block/dec1，则还会复制到 base 中新 block/dec1 的符号表中。

3，同样，一个模块的不同 block/dec1，或者不同模块都可以定义和一个 role 属性相关的规则。无论 role_attr 的 role_datum_t 定义在哪种符号表中，在 link 过程中都会合并到 base.p_roles 符号表中。

由于 role_datum_t.roles 位图记录了当前 role 属性所包含的所有普通 role，因此 roles 位图的合并必须在所有普通 role 都被 role_copy_callback 函数合并到 base.p_roles 之后、在当前模块的 policy_module_t.map[SYM_ROLES][]有效后进行。在 role_fix_callback 函数中先使用 map 数组，转换 roles 位图到 e_tmp 临时位图中，再将 e_tmp 和 base.p_roles 中已经存在的 role_datum_t 的 roles 相合并。

4，link_modules 首先针对所有模块，调用 copy_identifiers(..., NULL)函数，合并各模块的 p_xxx 符号表到 base 模块的相应 p_xxx 符号表。然后针对所有模块，调用 copy_module > ... > copy_identifier(..., dest_dec1)函数，合并个模块的**所有 block/dec1 的符号表**到 base 模块的相应 p_xxx 符号表。所以，**在 link 过程结束后，base.p_xxx 符号表就包含了所有模块中所有 block/dec1 定义的所有标识符！**

但是，尽管 link 过程结束后 base.p_xxx 符号表包含了所有模块中所有 block/dec1 所定义的标识符，某些标识符的某些属性（比如 type_datum_t.types 位图和 role_datum_t.roles 位图）却仍然保留在 block/dec1 的私有符号表中！

所以，在 expand 过程的后面还需要就 base 模块的所有 block 的当前使能的 dec1，调用 attr_convert_callback 函数，将 block/dec1 中可能的 type-attribute 关系拷贝到 out 模块中！

所以，在 link 过程的最后调用 expand_role_attributes 函数将子 role 属性的 roles 位图合并到父 role 属性之前，还必须调用 populate_roleattributes 函数将 base 模块所有 dec1 中可能存在的 role-attributes 关系，拷贝到 base.p_roles 符号表中！

即，populate_roleattributes 函数的行为，就是为了和 get_local_role 函数的行为互补（抵消）。

5，由下文可知，在 expand 过程中需要把一个 role 属性的 types.types 位图合并到每一个附属普通 role 的 types.types 位图，因此如果一个 role 属性附属于另一个 role 属性，则应该把 sub role 属性的 roles 位图合并到 parent role 属性的 roles 位图。

上述工作由 link_modules > expand_role_attributes 函数完成。

6，user_datum_t.roles 为 role_set_t 数据结构，其中 roles.roles 位图也可能记录了一个 role 属性。在 link 过程中不需要额外处理该 role 属性，通过 user_fix_callback > role_set_or_convert 函数将 roles.roles 经由 policy_module_t.map[SYM_ROLES][]数组转换为新的位图，再和 base 模块中 user_datum_t.roles.roles 相合并。另外 role_set_t.flags 也是直接被合并。

Expand 过程说明:
1，在 role_copy_callback 中将 base.p_roles 符号表复制到 out.p_roles 符号表，如果不存在则插入，此时会复制 flavor；如果存在，则调用 expand_convert_type_set 函数将 role_datum_t.types（type_set_t 数据结构）转换为 tmp_union_types 位图，再调用 ebitmap_union 函数将 tmp_union_types 位图合并到 out.p_roles 中 role_datum_t.types.types 位图。

注意，在 base 模块中 role_datum_t.types 为一个 type_set_t，其中 types/negtypes 位图都可能有效或者包含属性，其 flags 也可能包含特殊字符。而在 out 模块的 role_datum_t 中就只有 types.types 位图有效了！只有 types.types 位图会被写入 policy.X，而整个 types type_set_t 数据结构只会写入 pp 文件。

policy.X 中只会包含 ebitmap 的二进制表示，在 expand 过程中将 type_set_t 和 role_set_t 数据结构展开为 ebitmap。

2，role 属性的 roles 位图的 expand 显然需要在所有普通 role 都被合并到 out 模块后、expand_state_t.rolemap[]有效后进行。在 role_fix_callback 函数中将一个 role 属性的 roles 位图转换为临时位图，再合并到 out 模块中对应的 role_datum_t.roles。

注意，role_fix_callback 函数只需要针对 base.p_roles 符号表调用，而无须（也不应该）处理任何 block/decl 的私有 p_roles 符号表！因为后者中的 role 属性的 roles 位图，可能还记录了其他的 role 属性，这是因为 link 过程的 expand_role_attributes 函数只针对 base.p_roles 而调用而非 block/decl。

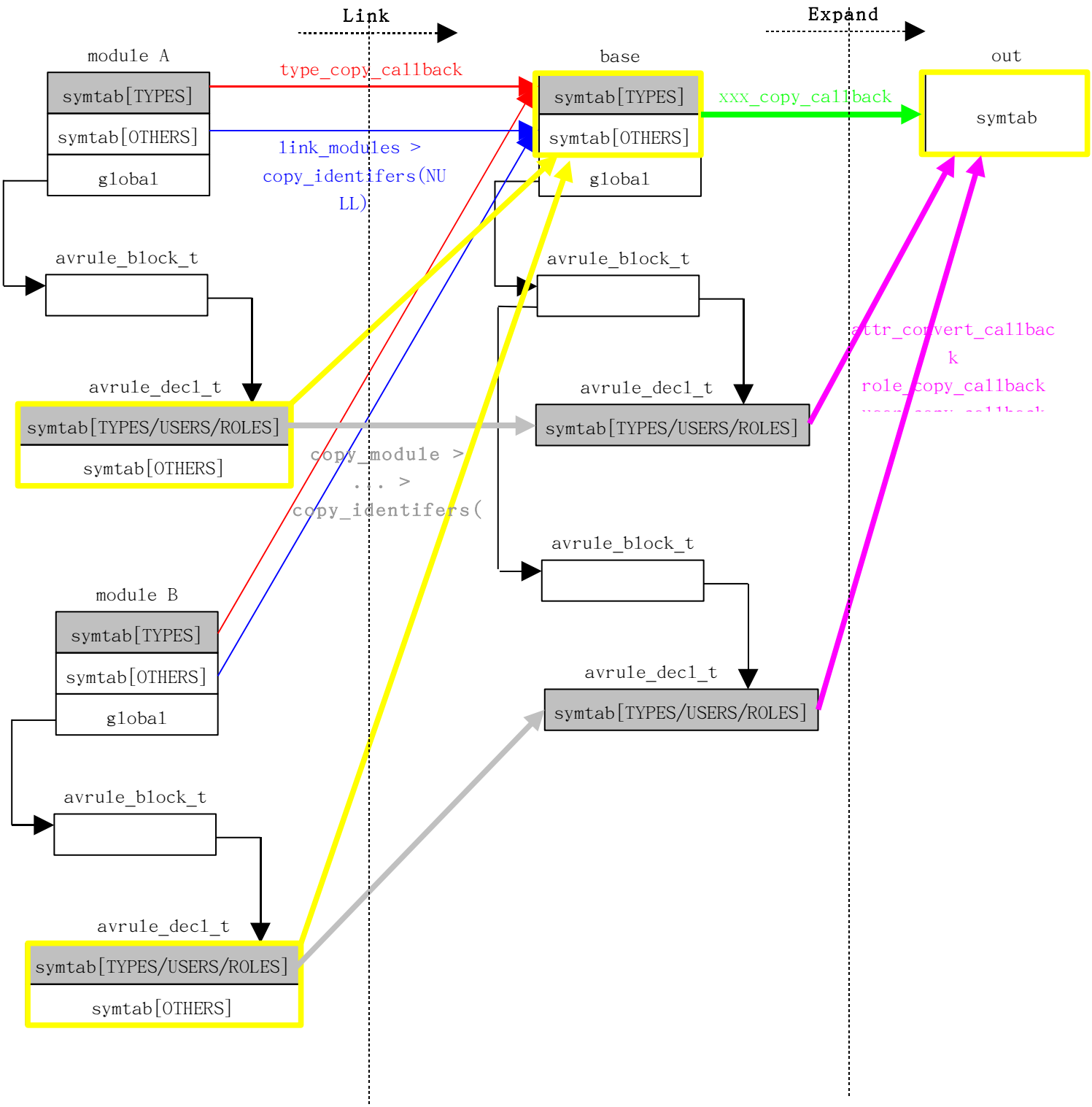
3，属性不但描述了一个集合（role_datum_t.roles 位图），而且还描述了该集合所拥有的共性（role_datum_t.types）。因此在一个 role 属性的 types type_set_t 被 expand 完成后，就需要把它合并到属于该 role 属性的所有普通 role 的 types.types 位图中！

所以在 role_fix_callback 函数中，就 base 模块中 role 属性的 roles 位图的所有非 0 位，利用 base 模块的 p_role_val_to_name[]数组得到相应普通 role 的名称字符串，以其为 key 查询 out.p_roles 符号表得到该普通 role 在 out 模块中的 role_datum_t，最后通过 ebitmap_union 合并 types.types。

4，role_set_expand 函数负责在 expand 过程中扩展一个 role_set_t.roles 位图。就其中的每一个非 0 位，利用 base 模块的 role_val_to_struct[]数组得到相应的 role_datum_t 数据结构。如果它是一个 role 属性，则将其 roles 位图合并到临时位图中，否则只将其(policy value - 1)记录到临时位图中。最后将临时位图经由 expand_state_t.rolemap[]转换后合并到 out 模块中相应数据结构的 role_set_t.roles 位图中。

注意，正是得益于在 link 过程的最后调用了 expand_role_attributes 函数，确保一个 role 属性的 roles 位图中不再含有任何 role 属性，在 expand 过程中 role_set_expand 函数才无须考虑 role 属性“嵌套”的情况，确保扩展后 roles.roles 位图中只含有普通 role。

12.7.2 symtab 的 link 和 expand



Link 过程说明：
1，link_modules 首先就所有模块的 p_types 符号表逐一调用 type_copy_callback 函数，将各个模块的 p_types 符号表合并到 base.p_types，如上图红色箭头所示；

2，待所有模块的 p_types 符号表合并完成后，再逐一针对各个模块通过 copy_identifiers 函数，合并其余下类型的所有符号表，如上图蓝色箭头所示；

注意，由于 user/role/type 标识符中直接或者间接包含 ebitmap 位图，因此在 xxx_copy_callback 函数调用完毕后，还需要调用 xxx_fix_callback 函数修正在 base 模块中的新位图。

3，然后，逐一针对各个模块调用 copy_module 函数，copy_module > copy_avrule_block > copy_avrule_decl，拷贝当前模块所有 block 的所有 decl 数据结构，包括在当前 block/decl 中定义的各种规则的字面描述，以及 block/decl 的私有符号表。

如上图黄色箭头所示，block/decl 的私有符号表也被 xxx_copy_callback 函数合并到 base 模块的对应符号表中。

注意，当前 block/decl 的所有规则的字面描述，都会被复制到 base 模块的副本中。而只有 user/role/type 类标识符，相应 xxx_copy_callback 才会将它们插入到 base 模块副本的私有符号表，如上图灰色加粗箭头所示。

Expand 过程说明：
1，expand_module 按照一定的顺序，调用 xxx_copy_callback 函数逐一拷贝 base 模块的符号表到 out 模块中；

2，然后，就 base 模块 global 链表中每个 avrule_block_t 的使能了的 avrule_decl_t，顺序地调用 attr_convert_callback，role_copy_callback 和 user_copy_callback 函数，合并 type 属性，role 和 user 到 out 模块的相应符号表中，如上图紫色箭头所示。

由于上述三个 callback 函数既要合并 base 模块的 p_xxx 符号表，也要合并 block/decl 的私有符号表，因此需要首先调用 hashtab_search 函数在 out 模块的相应符号表中查找当前 xxx_datum_t 是否存在，如果不存在则首先复制并插入，最后再合并 xxx_datum_t 中的各种数据结构。

与之相比，而其他 callback 函数就不需要调用 hashtab_search 了，直接复制并插入 out 模块的相应符号表即可。

体会：正如上文所言，正是由于某些标识符的属性（比如 type_datum_t.types 位图）仍然保留在 block/decl 的私有符号表而不是 base.p_types 符号表中（参见 get_local_type 函数），所以在 expand 过程的后面还需要就 base 模块的所有 block 的当前使能的 decl，调用 attr_convert_callback 函数将 block/decl 中可能的 type-attribute 关系拷贝到 out 模块中！

只要这样才能确保在某个 optional_policy 宏修饰的 block/decl 中定义的 type-attribute 关系生效，否则被遗漏！

另外，我觉得 expand 过程中只需针对 block/decl 的私有符号表调用 attr_convert_callback 函数（以和 get_local_type 函数的行为互补），调用 role_copy_callback 函数一定是多余的，而且一定不能调用 role_fix_callback 函数（是错误的）。

12.8 规则中的m4宏定义（new）

在编译refpolicy时首先使用m4将.te文件中使用的宏展开为checkpolicy/checkmodule能够识别的语法结构。.te文件中调用的interface和template都是refpolicy自定义的，都被视作需要展开的宏。interface和template在.if文件中定义，而其它比较低级的宏定义由policy/support/xxx.spt文件提供，比如domtrans_pattern等。

m4将若干输入文件视作一个字节流，将其划分为若干token。如果一个token存在一个定义（由define关键字声明一个宏定义），则将其展开。m4会递归地展开宏，直到无法展开为止。下面我们介绍一些在refpolicy中常见的m4宏定义，顺带介绍相关m4的使用方法。（在宏定义中每一行末尾的数字编号即为相应注释信息的编号）

```
define(`policy_module',`
        ifndef(`self_contained_policy',`                                # 2
                module $1 $2;                                           # 1

                require {
                        role system_r;
                        all_kernel_class_perms                          # 3

                        ifdef(`enable_mcs',`
                                dec1_sens(0,0)
                                dec1_cats(0,decr(mcs_num_cats))
                        ')

                        ifdef(`enable_mls',`                            # 4
                                dec1_sens(0,decr(mls_num_sens))         # 5
                                dec1_cats(0,decr(mls_num_cats))
                        ')
                }
        ')
')
```

1，policy_module宏用于声明一个模块，展开后的核心为module关键字，参数$1和$2分别为模块名称和版本号。在m4中$0总是为当前宏的名称；
2，在以模块方式编译时，才需要将policy_module宏展开。如果以Monolithic方式编译则在m4命令行定义 -D self_contained_policy变量，所以policy_module展开为空，这个也很好理解；
3，policy/flask/security_classes文件定义所有SELinux内核所支持的class类型，以及用户态SELinux Object Manager（比如X）所支持的class定义；policy/flask/access_vectors文件定义所有这些class所支持的permission。

all_kernel_class_perms宏将展开为一个字符串，包含所有SELinux内核所支持的class及其permission，比如：

```
        class system { ipc_info syslog_read syslog_mod syslog_console module_request };
```

用户态policydb_t数据结构即可以描述policy.X，也可以描述base.pp或者普通pp。所以普通pp中也必须包含所有class标识符及相关permission的定义；
4，在build.conf文件的TYPE变量决定支持的feature是否为MLS或者MCS；
5，在使能MLS时，可能需要定义若干sensitivity level，而在使能MCS时只需要一个（s0）；

```
# gen_cats(N)
#
# declares categores c0 to c(N-1)
#
define(`dec1_cats',`dn1                                                    # 1
category c$1;                                                            # 2
ifelse(`$1',`$2',,`dec1_cats(incr($1),$2)')dn1                           # 3
')

define(`gen_cats',`dec1_cats(0,decr($1))')
```

1，dec1_cats 宏用于定义一些列连续的 category，从参数$1 开始，共$2 个。比如 c0.c1023，共 1024 个；
2，dec1_cats 宏展开后的核心为 category 语法；
3，此处 m4 的 ifelse 结构有 4 个参数，如果第一个参数和第二个参数相同（逐字符完全相同），则展开为第三个参数（此处为空），否则展开为第四个参数。由于第四个参数为对 dec1_cats 宏的递归调用，只不过$1 递增 1，所以 dec1_cats(0，1024)将会被展开为

```
category c0;
category c1;
....
category c1023;
```

dec1_sens 宏的实现和 dec1_cats 相同，只是把 category 规则替换为 sensitivity 规则。

```
# For use in interfaces, to optionally insert a require block
#
define(`gen_require',`                                                   # 1
        ifdef(`self_contained_policy',`                                 # 3
                ifdef(`__in_optional_policy',`                          # 3
                        require {
                                $1                                      # 2
                        } # end require
                ')
        ',`
                require {
                        $1
                } # end require
        ')
')
```

1，gen_require 宏通常在.if 文件的 template 或 interface 宏内部使用，用于声明当前宏定义需要引用的外部依赖（由上文 refpolicy 的编译过程可知，无论一个标识符在哪个 block 中被定义或者声明为外部依赖，最终 symtab_insert 函数都会将它注册到当前模块的全局符号表中，从而保证后来在规则中使用该符号时它是被定义过的）。gen_require 宏会被展开为一个 ifdef 结构，此处包含两个分支，由当前编译方式决定那个方式生效；
2，gen_require 宏展开后的核心为 require 关键字，而 gen_require 宏的参数（$1）将被原封不动地传递给 require 语法；
3，注意 gen_require 宏并不是总是能够被展开为 require 语法。如果以模块方式编译则总是展开，如果以 Monolithic 方式编译，则只有在 optional block 中才需要展开（此时__in_optional_block 变量被定义，参见下文），而在 global block 中则不展开。

这一点也不难理解。如果以 Monolithic 方式编译则 policy.conf 中的 global block 部分应该为自包含的，所以在 global block 中调用的所有 template 或 interface 中的 gen_require 宏都不需要展开；而 optional block 必须声明自己所依赖的外部标识符，只有在外部依赖满足时该 optional block 才会被使能。

所以，如果一个 template 或 interface 在 global block 中被调用，那么在以 Monolithic 方式编译时其中的 gen_require 宏展开为空。这样就要求该 template 或 interface 实际使用的标识符在此之前必须被定义过，否则会出现“未定义”错误。参见 9.4.6 小节。

```
# Optional policy handling
#
define(`optional_policy',`                                       # 1
        ifelse(regexp(`$1',`\W'),`-1',`                         # 4
                refpolicywarn(`deprecated use of module name ($1) as first parameter of optional_policy()
block.')
                optional_policy(shift($*))                      # 4
        ',`
                optional {``pushdef(`__in_optional_policy')     # 1/5
                        $1                                      # 2
                ifelse(`$2',``,``,`} else {                     # 3
                        $2                                      # 3
                ')}``popdef(`__in_optional_policy')``ifndef(`__in_optional_policy',` # end optional')
        ')                                                      # 6
')
```

1，optional_policy 宏用于声明一个 optional block，其展开后的核心为 optional 关键字；
2，一个 optional block 包含 1～2 个分支，第一个分支被原封不懂地传递给 optional 关键字；
3，而第二个分支如果不为空，则增加“} else {”修饰符；
4，目前 optional_policy 宏的实现不鼓励使用所在模块名称作为第一个参数。使用 regexp 语法来计算第一个参数中非单词字符串的第一个索引，如果找不到则返回-1。找不到非单词字符串即说明包含单词伺服串，所以此时进一步调用 refpolicywarn 宏打印信息，然后使用 shift 语法弹出$1（使得原来的$2 作为新的$1，原来的$3 作为现在的$2，依次类推）后重新调用 optional_policy 宏；
5，在一个 optional block 中编译时，定义“__in_optional_policy”变量。pushdef 和 popdef 使得以栈的方式维护变量的定义：如果此前没有 define 定义则 pushdef 相当于 define，否则使用当前最新的定义；而 popdef 则恢复之前的定义；
6，一个 optional block 可能嵌套调用其他 optional block。所以当前（内层）optional block 编译完成后、popdef 之后，如果返回上层 optional block 则__in_optional_block 变量仍然被定义，所以使用 ifndef 结构检查，只有当它彻底未定义时才说明从最外层 optional block 返回，则此时在输出文件中加入“# end optional”字符串；

```
# In the future interfaces should be in loadable modules
#
# interface(name,rules)
#
define(`interface',` dnl
        ifdef(`$1',                                                                     # 2
                `refpolicyerr(`duplicate definition of $1(). Original definition on '$1.)
define(`__if_error')',
                `define(`$1',__line__)') dnl
        `define(`$1',                                                                   # 1
                ` dnl
                        pushdef(`policy_call_depth',incr(policy_call_depth)) dnl
                        policy_m4_comment(policy_call_depth,begin `$1'(dollarsstar)) dnl        # 3
                        $2
                        popdef(`policy_call_depth') dnl
                        policy_m4_comment(policy_call_depth,end `$1'(dollarsstar)) dnl
                '
        ')
')
```

```
define(`policy_call_depth',0)
```

1，template 和 interface 其实为 define 的封装，利用它们以具体模板或接口的名称字符串（即$1）定义一个新的 m4 宏。所以展开后的核心为一个 define(`$1',`$2')结构，即将当前 template 或 interface 的名称字符串展开为相应的规则块；
2，在展开之前首先用 ifdef 结构检查以该名称字符串命名的宏定义是否已经存在。如果已经被定义过，则调用 rerfpolicyerr 宏打印出错信息，然后定义__if_error 变量（而在 support/iferror.m4 中定义 ifdef(`__if_error',`m4exit(1)')）
3，在展开中还调用 policy_m4_comment 宏，输出当前调用的嵌套数目。policy_call_depth 变量被初始化为数字 0，在任何一个 template 或 interface 调用期间递增 1，返回时递减 1；

比如存在如下接口定义：

```
interface(`vlock_run',`
        gen_require(`
                type vlock_t;
        ')

        vlock_domtrans($1)
        role $2 types vlock_t;
')
```

那么实际调用比如 vlock_run(sysadm_t，sysadm_r)中使用的参数将被 vlock_run 的定义（$2）继续使用，所以上述调用将被展开为：

```
        gen_require(`
                type vlock_t;
        ')

        vlock_domtrans(sysadm_t)
        role sysadm_r types vlock_t;
```

（注意 vlock_domtrans 宏会被继续展开，在此忽略）

tunable_policy 宏被展开为 if-else conditional：

```
define(`tunable_policy',`
        gen_require(`
                declare_required_tunables(`$1')                         #1
        ')
        if (`$1') {                                                     #2
                $2
        ifelse(`$3',``,``,`} else {                                     #3
                $3
        ')}
')
```

1，首先使用 declare_required_tunables 宏，来声明对相应 tunables 标识符的外部依赖；
2，整个 tunable_policy 宏所携带的代码块，被翻译为相应的 if-else 结构；
3，如果 else 块不为空，则增加“} else {”字符串定义；

```
        tunable_policy(`user_dmesg',`
                kernel_read_ring_buffer($1_t)
        ',`
                kernel_dontaudit_read_ring_buffer($1_t)
        ')
```

于是，上面的 tunable_policy 宏，将被翻译为：

```
#line 12
                require {
...
tunable user_dmesg;
...
                } # end require
#line 12

#line 12

#line 12
        if (user_dmesg) {
...
##### begin kernel_read_ring_buffer(test_t) depth: 2
...
##### end kernel_read_ring_buffer(test_t) depth: 1
...
        } else {
...
##### begin kernel_dontaudit_read_ring_buffer(test_t) depth: 2
...
##### end kernel_dontaudit_read_ring_buffer(test_t) depth: 1
...
        }
#line 12
```

由此可见，的确是一个 if-else 结构。

13. SELinux 的应用

13.1 Labeled Networking (half-baked)

13.1.1 IPsec 简介

IPsec 协议包含 AH（Authentication Header）和 ESP（Encapsulate Security Payload），前者能够提供身份认证和数据完整性服务，后者能够进一步提供数据加密服务。

IPsec 协议能够为 IP 层之上的协议（比如 TCP 等）提供安全服务，比如身份认证、数据完整性、数据加密。此时使用 IPsec 的 Transport 模式，在 IP 头和上层协议数据报之间插入 IPsec 头（ESP 或者 AH，或者两者都有）。

IPsec 协议也能够为整个 IP 报文提供这些安全服务，此时使用 IPsec 的 Tunnel 模式，在整个 IP 数据报的外面封装 IPsec 头以及新的 IP 头。

Transport 模式用于实现“点对点”的安全服务，网络拓朴上的任何两个叶子节点，只要已经能够正常访问对方，就可以使用 Transport 模式。

Tunnel 模式通常部署在两个 Gateway 上，每个 Gateway 都代表各自的子网，从而在这两个子网之间建立安全链接。

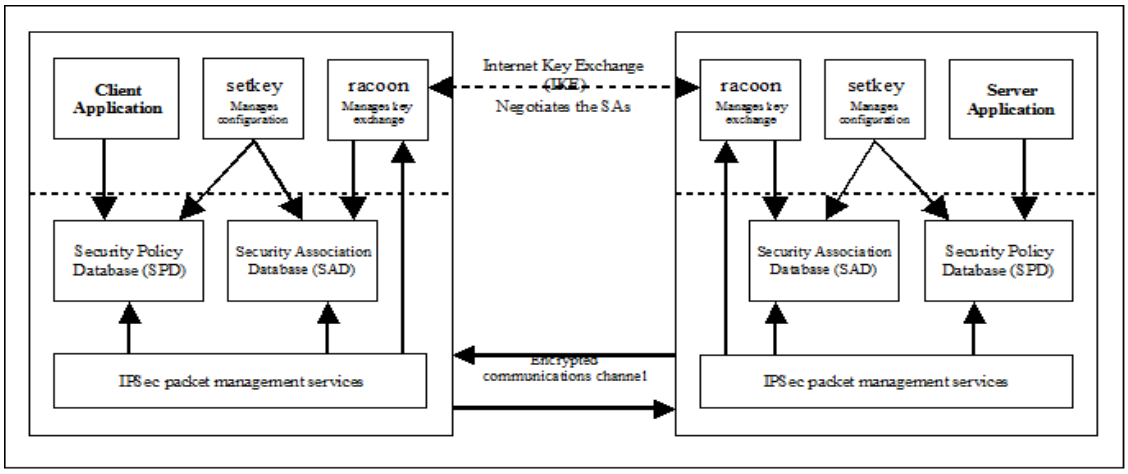
IPsec 通过 SA（Security Association）来提供安全服务，简单地说一个 SA 就是一个单向的安全连接（connection），通信的双方在验证完对方的身份、协商好密钥后，即建立了 SA。一个 SA 是单向的，所以通信的每一方都至少有两个 SA（一个发送，一个接收）。如果需要同时使用 AH 和 ESP 协议，则需要 4 个 SA。

可以将一个 SA 想象成一个“水管”，管道对在其内部流通的数据提供安全性。如果同时使用 AH 和 ESP 协议，则可以想象成有两层水管套在一起。

用户态的 ipsec-tools 包提供 racoon 后台进程和 setkey 工具。使用 setkey 工具向内核的 SPD（Security Policy Database）中注册若干 policy 条目，每个 policy 条目都指名需要使用 IPsec 安全服务的源、目的 IP 地址，上层协议编号以及端口号，以及需要使用的 IPsec 模式/协议等信息。当前已经注册的 SPD 条目可以用“setkey -DP”命令查看。

当内核发送/接收 IP 数据报时，如果它需要使用 IPsec 服务（有对应的 SPD 条目存在），则通过相应的 SA 来发送/接收它。如果此时相应的 SA 尚未建立，则 IPsec 内核驱动将会激活用户态 racoon 后台进程，由它完成身份认证以及交换密钥的工作（即建立 SA）。建立好的 SA 的具体参数（比如身份认证算法，加密算法以及密钥）记录在 SAD（Security Association Database）中，可以用“setkey -D”命令查看。

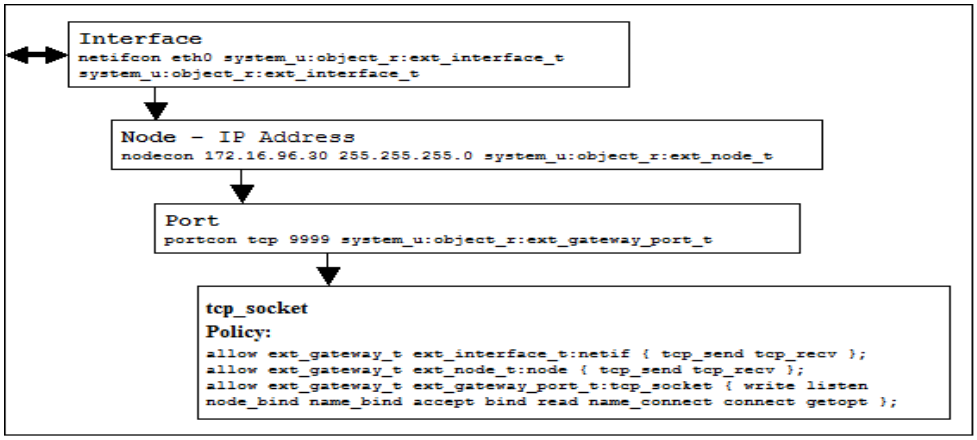
racoon 建立 SA 的过程由其配置文件 racoon.conf 来决定，部署 IPsec 双方的 racoon.conf 文件必须相同。racoon 可以使用证书（certificate，双方的证书需要由可信的 CA 颁发），pre-shared-key，Kerberos 来验证双方的身份，然后使用 Diffie Hellmann 密钥交换算法来协商密钥。参见下图。



有关 IPsec 进一步的介绍在参考文献 13 中。

13.1.2 SELinux 对本地网络的控制（compat-net）

refpolicy 已经通过 netifcon，portcon，nodecon，以及相应的 Initial SID 给本地网络设施定义了相应的安全上下文，通过 TE 规则控制哪些 domain（确切地说是这些 domain 所创建的 socket）能够以什么权限使用这些网络设施，参见下图：



在上面的例子中，使用 netifcon 规则将 eth0 的标签设置为 ext_interface_t，使用 nodecon 规则将 172.16.96.30 节点的标签设置为 ext_node_t，使用 portcon 规则将 TCP 9999 端口的标签设置为 ext_gateway_port_t。那么，相应的 allow 规则使得 ext_gateway_t 这个 domain/type 能够和 TCP 9999 端口相绑定（bind），并许可监听、接收等操作；并且能够在 eth0 接口上收发 TCP 报文，在 172.16.96.30 机器上收发 TCP 报文。

注意，ext_gateway_t 为 socket 的 type，通常等于其创建者的 domain。一个 socket 必须具备上述对 netif/port/node 的相关权限，才能正常收发报文。

但是，这种控制只局限于本地而无法跨越网络，比如无法控制两台机器上任意两个进程是否能够互联。

如果希望跨越网络，即在两个机器之间实施 MAC 控制，则必须让网路数据报携带对方的安全上下文信息。而为了保证网络数据报中的安全上下文信息是真实有效的，必须满足如下条件：
1，对方机器必须和本地机器一样，都是可信的。称为 TCB（Trusted Compute Base），比如都部署了 SELinux，从而确保数据报中的安全上下文信息和发送者进程的 domain 保持一致（而不是被伪装了的）；

2，具有一个安全的信道，确保网络数据报的内容不被篡改；

由此可见，“MAC + IPsec”是必然的，因为只有 IPsec 才能保证可信的信道，进而保证数据报头部所携带的 MAC 信息的正确性。

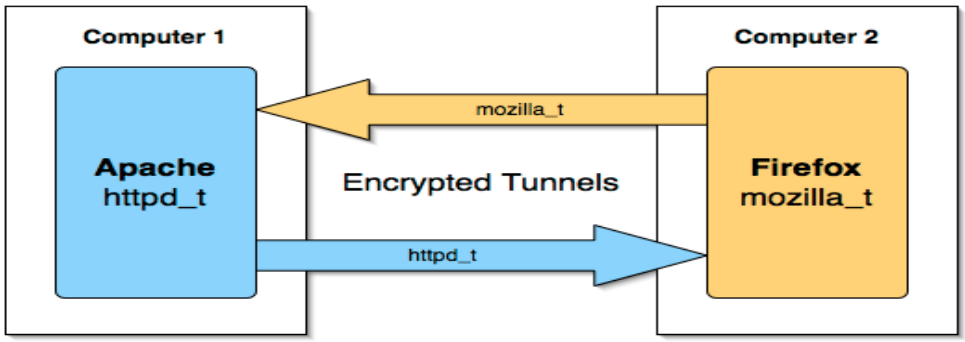
13.1.3 用 Labeled IPsec 实现分布式网络控制

分布式网络控制由两个层面构成：
1，控制本地进程对 socket 的访问（compat-net）；
2，控制 socket 对 SA 的访问；

本地 socket 的安全上下文由 sk_security_struct 数据结构描述（绝大多数情况下继承其创建者的 SC，可能被相应 type_transition 规则改变，也可能被应用程序通过 sockcreate 接口直接设置），为了控制 socket 对 SA 的访问，必须首先对 SA 也打标签，然后在 socket 访问 SA 时查询 Security Server，看 policy.X 是否许可当前的操作。

说白了就是需要把 Linux 内核 IPsec 子系统也变成一个 Object Manager，不但给 IPsec 的核心数据结构设计 SELinux 安全属性并打标签（所以才称为 ***Labeled*** IPsec），并且在 IPsec 操作中设置 LSM 回调函数并提供 SELinux 的实现，由它们访问 Security Server 并施加 MAC 控制。

对 SA 打标签的方法由下图描述：



注意，任何一方都至少存在两个 SA，SAin 的 SC 总是和远程发送者进程的 SC 一致；SAout 总是和本地发送者进程的 SC 一致。

通过在数据报中携带发送方的安全上下问信息，本地接收端就可以使用 SELinux 规则来决定本地的哪些 domain 能够接受这些报文了（确切地说，SAin 的 SC 由数据报即远程发送者确定，而本地 socket 的 SC 由其创建者即接收进程确定），即实现控制两台机器上哪些进程能够相互访问的目的。

体会：SELinux 给本地进程划分 domain，现在又借助 Labeled IPsec 得到了远程进程的 domain。将本地和远程进程区分为不同的 domain 就是为了施加相应的控制！

xinetd 可以根据数据报中携带的安全上下文信息决定新创建的 worker process 的安全上下文，从而保证请求方只能看到本地 SELinux 策略所许可它访问的本地资源，正如请求者直接在本地登录时一样。

比如，远端使用 ftp client 来访问本地服务器，数据报中携带的安全上下文信息描述 ftp client 的 domain 和 MLS level，则本地 xinetd 将创建子进程执行 ftpd，xinetd 根据数据报中的 MLS level 来设置本地 ftpd 进程的 level，使得远端客户只能访问到和其 MLS level 相匹配的那些文件，而不是本地上高安全级别的文件。

13.1.4 Linux 内核 XFM 相关数据结构

为了将一个内核子系统变成一个 Object Manager，首先需要抽象出它所管理的数据结构对象所属的分类，然后给相关对象设计描述 SELinux 安全属性的数据结构。

XFRM 子系统使用 xfrm_policy 和 xfrm_state 数据结构分别描述 SPD 和 SAD。socket->sock 数据结构在使能 XFRM 时增加了一个指向 xfrm_policy 的指针数组：

```
#ifdef CONFIG_XFRM
        struct xfrm_policy      *sk_policy[2];
#endif
```

即指向和该 socket 相关的、用于收发数据报的 IPsec policy。

xfrm_policy 和 xfrm_state 数据结构的安全属性由 xfrm_sec_ctx 数据结构描述：

```
struct xfrm_sec_ctx {
        __u8    ctx_doi;                # == XFRM_SC_DOI_LSM == 1
        __u8    ctx_alg;                # == XFRM_SC_ALG_SELINUX == 1
        __u16   ctx_len;                # ctx_str 的长度
        __u32   ctx_sid;                # 调用 security_context_to_sid 向 sidtab 注册后的返回值
        char    ctx_str[0];             # 比如 system_u:object_r:ipsec_spd_t:s0-s15:c0.c1023
};
```

ctx_doi 和 ctx_alg 域是为了扩展的需要，将来可能使用非 SELinux 的其他 LSM 实现，或者使用非 LSM 的其他安全框架。就 LSM/SELinux 而言，这两个域都设置为 1，即由 spdadd 命令的“-ctx 1 1”选项决定。

```
struct xfrm_user_sec_ctx {
        __u16                   len;
        __u16                   exttype;
        __u8                    ctx_alg;  /* LSMs: e.g., selinux == 1 */
        __u8                    ctx_doi;
        __u16                   ctx_len;
};
```

xfrm_user_sec_ctx 数据结构用于描述用户态 setkey.conf 文件中 spdadd 命令所指定的 SPD 条目的安全上下文。比如用户指定“-ctx 1 1 “system_u:object_r:ipsec_spd_t:s0””，则 ctx_doi = ctx_alg = 1，字符串“system_u:object_r:ipsec_spd_t:s0”紧随该数据结构分配，其长度记录在 ctx_len 中。

13.1.5 和 IPsec 相关的类，权限和接口

为了支持将 XFRM 子系统变成一个 Object Manager，在 refpolicy 和 SELinux 内核驱动中为 IPsec 设计了如下类及权限：

```
class association
{
        sendto
        recvfrom
        setcontext
        polmatch
}
```

这样在 refpolicy 中就可以编写关于进程 domain/socket type 对 SPD/SAD type 的规则了，并且在 XFRM 中设计若干 LSM 回调函数完成如下操作：

1，在创建 SPD/SAD 数据结构时初始化安全属性，确定其 sid;
2，在相关操作中向 Security Server 查询当前操作是否许可;
3，如果不被许可，则 LSM 回调函数返回失败，则导致当前操作被禁止（即实施 Security Server 的决定）;

association 类用于描述 SPD 和 SAD 的条目。
polmatch 描述当前 socket 的 type 是否可以使用一个 SPD/SAD 条目;
setcontext 描述当前进程 domain 是否可以设置一个 SPD/SAD 条目（在添加、删除 SPD/SAD 时检查）;
sendto/recvfrom 描述当前 socket 的 type 是否可以和一个 SA 之间发送/接收数据报。

13.1.6 LSM 中和 Labeled Ipsec 相关的回调函数

（TODO：要想进一步了解 XFRM 中 LSM 回调函数的作用，必须了解其所在 XFRM 代码的机制！由于目前对 XFRM 的实现还不了解，所以无法按照情景分析的思路分析相关源代码）

使能内核配置选项 CONFIG_SECURITY_NETWORK_XFRM，将为如下 XFRM 中的回调函数使能 SELinux 上的实现:

```
#ifdef CONFIG_SECURITY_NETWORK_XFRM
        .xfrm_policy_alloc_security =   selinux_xfrm_policy_alloc,
        .xfrm_policy_clone_security =   selinux_xfrm_policy_clone,
        .xfrm_policy_free_security =    selinux_xfrm_policy_free,       # 释放一个 SPD
        .xfrm_policy_delete_security =  selinux_xfrm_policy_delete,     # 检查当前 domain 能否删除一个 SPD
        .xfrm_state_alloc_security =    selinux_xfrm_state_alloc,
        .xfrm_state_free_security =     selinux_xfrm_state_free,        # 释放一个 SAD
        .xfrm_state_delete_security =   selinux_xfrm_state_delete,      # 检查当前 domain 能够删除一个 SAD
        .xfrm_policy_lookup =           selinux_xfrm_policy_lookup,
        .xfrm_state_pol_flow_match =    selinux_xfrm_state_pol_flow_match,
        .xfrm_decode_session =          selinux_xfrm_decode_session,
#endif
```

include/linux/security.h 中有关于 xfrm_xxx 函数型构的介绍
并且参考资料《Leveraging IPsec for Distributed Authorization》

13.1.6.1 检查一个 flow（发送或接收）能否使用一个 SPD 条目

xfrm_policy_match 函数检查当前 flow 是否能够和当前 SPD 条目相匹配，以及是否能够使用它:

```
/*
 * Find policy to apply to this flow.
 *
 * Returns 0 if policy found, else an -errno.
 */
static int xfrm_policy_match(const struct xfrm_policy *pol, const struct flowi *fl,
                             u8 type, u16 family, int dir)
{
        const struct xfrm_selector *sel = &pol->selector;
        int match, ret = -ESRCH;

        if (pol->family != family || (fl->flowi_mark & pol->mark.m) != pol->mark.v || pol->type != type)
                return ret;
```

（TODO： family 为当前 flow 的协议族（IPv4/v6），flowi_mark 和 type 是什么含义？）

```
        match = xfrm_selector_match(sel, fl, family);
        if (match)
                ret = security_xfrm_policy_lookup(pol->security, fl->flowi_secid, dir);
```

如果当前 flow 的源、目的地址，端口号等能够和当前 SPD 条目相匹配，则进一步调用 LSM 回调函数检查当前 flow 是否能够使用它。

```
        return ret;
}
```

selinux_xfrm_policy_lookup 函数
检查当前 flow 是否能否使用一个 SPD 条目。参数 ctx 等于当前 SPD 条目的 xfrm_policy 数据结构中指向 xfrm_sec_ctx 数据结构的指针，fl_secid 为 flow 的 sid。

```
/*
 * LSM hook implementation that authorizes that a flow can use
 * a xfrm policy rule.
 */
int selinux_xfrm_policy_lookup(struct xfrm_sec_ctx *ctx, u32 fl_secid, u8 dir)
{
        int rc;
        u32 sel_sid;

        /* Context sid is either set to label or ANY_ASSOC */
        if (ctx) {
                if (!selinux_authorizable_ctx(ctx))
                        return -EINVAL;
                sel_sid = ctx->ctx_sid;
```

如果 ctx 不为 NULL，首先通过 selinux_authorizable_ctx 函数检查该 xfrm_sec_ctx 数据结构是否为 LSM/SELinux 所使用，如果不是，则返回错误；否则其 ctx_sid 即为该 SPD 条目的 SID（对应于 spdadd 命令“-ctx”选项所设置的 SC 字符串，比如 ipsec_spd_t）。

```
        } else
                /*
                 * All flows should be treated as polmatch'ing an
                 * otherwise applicable "non-labeled" policy. This
                 * would prevent inadvertent "leaks".
                 */
                return 0;
```

如果 ctx 为 NULL，说明当前 SPD 条目并没有应用 SELinux 的标签（即为 non-labeled IPsec），所以默认许可当前 flow 能够使用该 SPD 条目。

```
        rc = avc_has_perm(fl_secid, sel_sid, SECCLASS_ASSOCIATION, ASSOCIATION__POLMATCH, NULL);
```

最终通过 avc_has_perm 函数检查当前 flow 是否对于该 SPD 条目具备 association 类的 polmatch 权限。如果没有，则当前 flow 无法使用该 SPD 条目。

```
        if (rc == -EACCES)
                return -ESRCH;

        return rc;
}
```

13.1.6.2 给 SPD/SAD 分配安全属性

selinux_xfrm_policy_alloc 函数给一个 xfrm_policy 数据结构分配 xfrm_sec_ctx 数据结构。参数 ctxp 为 xfrm_policy->security 指针的地址，uctx 指向的 xfrm_user_sec_ctx 数据结构描述用户态 spdadd 命令的“-ctx”选项：

```
int selinux_xfrm_policy_alloc(struct xfrm_sec_ctx **ctxp, struct xfrm_user_sec_ctx *uctx)
{
        int err;

        BUG_ON(!uctx);

        err = selinux_xfrm_sec_ctx_alloc(ctxp, uctx, 0);
        if (err == 0)
                atomic_inc(&selinux_xfrm_refcount);

        return err;
}
```

类似地，seinux_xfrm_state_alloc 函数给一个 xfrm_state 数据结构分配 xfrm_sec_ctx 数据结构：

```
int selinux_xfrm_state_alloc(struct xfrm_state *x, struct xfrm_user_sec_ctx *uctx, u32 secid)
{
        int err;

        BUG_ON(!x);

        err = selinux_xfrm_sec_ctx_alloc(&x->security, uctx, secid);
        if (err == 0)
                atomic_inc(&selinux_xfrm_refcount);
        return err;
}
```

上述两个函数都借助 selinux_xfrm_sec_ctx_alloc 函数分配 xfrm_sec_ctx 数据结构，并按照传递的参数初始化。

如果分配并初始化 xfrm_sec_ctx 成功，这两个函数还递增 selinux_xfrm_refcount 计数器（在其他 SELinux 和网络相关的回调函数中使用）。

selinux_xfrm_sec_ctx_alloc 函数分配一个 xfrm_sec_ctx 数据结构。参数 ctxp 为上层数据结构 xfrm_policy 或 xfrm_state 中指向 xfrm_seec_ctx 数据结构的指针 security 的地址，uctx 指向的数据结构及其后继字符串包含用户态 spdadd 命令“-ctx”选项的内容。sid 为该函数在被 XFRM 内核驱动自身调用时，传递的当前 SPD 条目的 SID。

```
/*
 * Security blob allocation for xfrm_policy and xfrm_state
 * CTX does not have a meaningful value on input
 */
static int selinux_xfrm_sec_ctx_alloc(struct xfrm_sec_ctx **ctxp, struct xfrm_user_sec_ctx *uctx, u32 sid)
{
        int rc = 0;
        const struct task_security_struct *tsec = current_security();
        struct xfrm_sec_ctx *ctx = NULL;
        char *ctx_str = NULL;
        u32 str_len;

        BUG_ON(uctx && sid);
```

```
if (!uctx)
        goto not_from_user;
```

如果 uctx 不为空，则说明因用户态而调用；否则被内核直接调用，转到“not_from_user”处理。无论何种情况 uctx 和 sid 参数总有一个不为 NULL。

```
if (uctx->ctx_alg != XFRM_SC_ALG_SELINUX)
        return -EINVAL;
```

用户态使用 spdadd 命令的“-ctx”选项时必须指定“-ctx 1 1 “<security context string>””，其中 doi 和 alg 都为 1，即指定 xfrm_sec_ctx 的使用者为 LSM/SELinux。在这里检查是否为 SELinux 所使用。

```
str_len = uctx->ctx_len;
if (str_len >= PAGE_SIZE)
        return -ENOMEM;

*ctxp = ctx = kmalloc(sizeof(*ctx) + str_len + 1, GFP_KERNEL);
if (!ctx)
        return -ENOMEM;
```

用户态指定的 SPD/SAD 条目的安全上下文字符串，都紧随 xfrm_user_sec_ctx 和 xfrm_sec_ctx 数据结构分配，最大长度应该不超过一个页框（否则会被丢弃）。所以首先分配足够的空间以容纳 xfrm_sec_ctx 数据结构及后继安全上下文字符串。多分配一个字节是为了容纳结束符。

```
ctx->ctx_doi = uctx->ctx_doi;
ctx->ctx_len = str_len;
ctx->ctx_alg = uctx->ctx_alg;

memcpy(ctx->ctx_str, uctx+1, str_len);
ctx->ctx_str[str_len] = 0;
```

然后复制 xfrm_user_sec_ctx 数据结构的相应域到 xfrm_sec_ctx 数据结构，以及安全上下文字符串。

```
rc = security_context_to_sid(ctx->ctx_str, str_len, &ctx->ctx_sid);
if (rc)
        goto out;
```

调用 security_context_to_sid 函数，向 sidtab 注册该字符串并返回对应的 SID，以便下面调用 avc_has_perm 接口。

```
/*
 * Does the subject have permission to set security context?
 */
rc = avc_has_perm(tsec->sid, ctx->ctx_sid, SECCLASS_ASSOCIATION, ASSOCIATION__SETCONTEXT, NULL);
if (rc)
        goto out;

return rc;
```

最后调用 avc_has_perm 函数，检查当前执行流对该 SPD/SAD 条目的安全上下文字符串是否具有 association 类的 setcontext 能力。（注意，上面把安全上下文注册到 sidtab 得到 sid，而 avc_has_perm > ... > security_compute_av，后者又会调用 sidtab_search 函数由 sid 得到安全上下文字符串，最终查询 Security Server 是否存在许可当前进程 SC 对安全上下文字符串具有相应能力的规则）

一方面，在 refpolicy 中对 setkey_t 调用了 corenet_setcontext_all_spds 接口，使得 setkey_t 对 association 类的 ipsec_spd_type 属性具有 setcontext 能力；另一方面，在 spdadd 命令中指定“-ctx

1 1“system_u:object_r:ipsec_spd_t:s0”” ，从而使得这里的检查能够满足。

```
not_from_user:
        rc = security_sid_to_context(sid, &ctx_str, &str_len);
        if (rc)
                goto out;
```

如果直接被内核调用，则参数 sid 为发送 socket 的 sid，或者接收到的数据报中所携带的安全上下文所对应的 sid，分别对应需要确定 SAout 或者 SAin 的安全属性的情况（TODO：对此情形有待验证）。

通过查询 sidtab，直接返回 sid 所对应的安全上下文字符串即可。

```
        *ctxp = ctx = kmalloc(sizeof(*ctx) + str_len, GFP_ATOMIC);
        if (!ctx) {
                rc = -ENOMEM;
                goto out;
        }

        ctx->ctx_doi = XFRM_SC_DOI_LSM;
        ctx->ctx_alg = XFRM_SC_ALG_SELINUX;
        ctx->ctx_sid = sid;
        ctx->ctx_len = str_len;
        memcpy(ctx->ctx_str, ctx_str, str_len);
```

分配 xfrm_sec_ctx 数据结构，并拷贝 ctx_str 字符串。**“SAout 的 sid 等于发送者 socket 的 sid，SAin 的 sid 等于接收数据报中所携带的安全上下文所对应的 sid”，正是在这里实现！**

```
        goto out2;

out:
        *ctxp = NULL;
        kfree(ctx);
out2:
        kfree(ctx_str);
        return rc;
}
```

注意 security_sid_to_context 函数会分配 ctx_str 字符串空间，所以在退出前需要释放。

另外，从 xfrm_state_alloc_security 函数的调用这来看，在静态创建 SA 时，参数 sid == 0；在由 racoon 协商、自动建立 SA 时，参数 sid 为相应 soekct 的 sid：

```
int security_xfrm_state_alloc(struct xfrm_state *x, struct xfrm_user_sec_ctx *sec_ctx)
{
        return security_ops->xfrm_state_alloc_security(x, sec_ctx, 0);
}
EXPORT_SYMBOL(security_xfrm_state_alloc);

int security_xfrm_state_alloc_acquire(struct xfrm_state *x, struct xfrm_sec_ctx *polsec, u32 secid)
{
        if (!polsec)
                return 0;
        /*
         * We want the context to be taken from secid which is usually from the sock.
         */
        return security_ops->xfrm_state_alloc_security(x, NULL, secid);
}
```

13.1.6.3 释放 SPD/SAD 中的安全属性

selinux_xfrm_policy_delete 函数和 selinux_xfrm_state_delete 函数用于检查当前进程 domain 是否有能够修改 SPD/SAD 条目，从而许可当前进程释放其中的 xfrm_sec_ctx 数据结构：

```
int selinux_xfrm_policy_delete(struct xfrm_sec_ctx *ctx)
{
        const struct task_security_struct *tsec = current_security();
        int rc = 0;

        if (ctx) {
                rc = avc_has_perm(tsec->sid, ctx->ctx_sid,
                                  SECCLASS_ASSOCIATION, ASSOCIATION__SETCONTEXT, NULL);
                if (rc == 0)
                        atomic_dec(&selinux_xfrm_refcount);
        }

        return rc;
}

int selinux_xfrm_state_delete(struct xfrm_state *x)
{
        const struct task_security_struct *tsec = current_security();
        struct xfrm_sec_ctx *ctx = x->security;
        int rc = 0;

        if (ctx) {
                rc = avc_has_perm(tsec->sid, ctx->ctx_sid,
                                  SECCLASS_ASSOCIATION, ASSOCIATION__SETCONTEXT, NULL);
                if (rc == 0)
                        atomic_dec(&selinux_xfrm_refcount);
        }

        return rc;
}
```

和分配安全属性时相同，这里也检查当前进程 domain 是否对 SAD/SPD 条目具有 association 类的 setcontext 能力。同时递减 selinux_xfrm_refcount 计数器。

13.1.6.5 逐包检查一个 socket 能够接收一个 skb

```
static int selinux_socket_sock_rcv_skb(struct sock *sk, struct sk_buff *skb)
{
        int err;
        struct sk_security_struct *sksec = sk->sk_security;
        u16 family = sk->sk_family;
        u32 sk_sid = sksec->sid;
        struct common_audit_data ad;
        char *addrp;
        u8 secmark_active;
        u8 peerlbl_active;

        if (family != PF_INET && family != PF_INET6)
                return 0;
```

```
        /* Handle mapped IPv4 packets arriving via IPv6 sockets */
        if (family == PF_INET6 && skb->protocol == htons(ETH_P_IP))
                family = PF_INET;

        /* If any sort of compatibility mode is enabled then handoff processing
         * to the selinux_sock_rcv_skb_compat() function to deal with the
         * special handling.  We do this in an attempt to keep this function
         * as fast and as clean as possible. */
        if (!selinux_policycap_netpeer)
                return selinux_sock_rcv_skb_compat(sk, skb, family);

        secmark_active = selinux_secmark_enabled();
        peerlbl_active = netlbl_enabled() || selinux_xfrm_enabled();
        if (!secmark_active && !peerlbl_active)
                return 0;

        COMMON_AUDIT_DATA_INIT(&ad, NET);
        ad.u.net.netif = skb->skb_iif;
        ad.u.net.family = family;
        err = selinux_parse_skb(skb, &ad, &addrp, 1, NULL);
        if (err)
                return err;

        if (peerlbl_active) {
                u32 peer_sid;

                err = selinux_skb_peerlbl_sid(skb, family, &peer_sid);
                if (err)
                        return err;
                err = selinux_inet_sys_rcv_skb(skb->skb_iif, addrp, family, peer_sid, &ad);
                if (err) {
                        selinux_netlbl_err(skb, err, 0);
                        return err;
                }
                err = avc_has_perm(sk_sid, peer_sid, SECCLASS_PEER, PEER__RECV, &ad);
                if (err)
                        selinux_netlbl_err(skb, err, 0);
        }

        if (secmark_active) {
                err = avc_has_perm(sk_sid, skb->secmark, SECCLASS_PACKET, PACKET__RECV, &ad);
                if (err)
                        return err;
        }

        return err;
}
```

13.1.6.4 获取发送方的安全上下文字符串

```
tcp_rcv_synsent_state_process > selinux_inet_conn_established

static void selinux_inet_conn_established(struct sock *sk, struct sk_buff *skb)
{
        u16 family = sk->sk_family;
        struct sk_security_struct *sksec = sk->sk_security;
```

```
        /* handle mapped IPv4 packets arriving via IPv6 sockets */
        if (family == PF_INET6 && skb->protocol == htons(ETH_P_IP))
                family = PF_INET;

        selinux_skb_peerlbl_sid(skb, family, &sksec->peer_sid);
}


/**
 * selinux_skb_peerlbl_sid - Determine the peer label of a packet
 * @skb: the packet
 * @family: protocol family
 * @sid: the packet's peer label SID
 *
 * Description:
 * Check the various different forms of network peer labeling and determine
 * the peer label/SID for the packet; most of the magic actually occurs in
 * the security server function security_net_peersid_cmp().  The function
 * returns zero if the value in @sid is valid (although it may be SECSID_NULL)
 * or -EACCES if @sid is invalid due to inconsistencies with the different
 * peer labels.
 *
 */
static int selinux_skb_peerlbl_sid(struct sk_buff *skb, u16 family, u32 *sid)
{
        int err;
        u32 xfrm_sid;
        u32 nlbl_sid;
        u32 nlbl_type;

        selinux_skb_xfrm_sid(skb, &xfrm_sid);
        selinux_netlbl_skbuff_getsid(skb, family, &nlbl_type, &nlbl_sid);

        err = security_net_peersid_resolve(nlbl_sid, nlbl_type, xfrm_sid, sid);
        if (unlikely(err)) {
                printk(KERN_WARNING
                       "SELinux: failure in selinux_skb_peerlbl_sid(),"
                       " unable to determine packet's peer label\n");
                return -EACCES;
        }

        return 0;
}



static inline void selinux_skb_xfrm_sid(struct sk_buff *skb, u32 *sid)
{
        int err = selinux_xfrm_decode_session(skb, sid, 0);
        BUG_ON(err);
}

/*
 * LSM hook implementation that checks and/or returns the xfrm sid for the
 * incoming packet.
 */
int selinux_xfrm_decode_session(struct sk_buff *skb, u32 *sid, int ckall)
{
        struct sec_path *sp;
```

```
        *sid = SECSID_NULL;

        if (skb == NULL)
                return 0;

        sp = skb->sp;
        if (sp) {
                int i, sid_set = 0;

                for (i = sp->len-1; i >= 0; i--) {
                        struct xfrm_state *x = sp->xvec[i];
                        if (selinux_authorizable_xfrm(x)) {
                                struct xfrm_sec_ctx *ctx = x->security;

                                if (!sid_set) {
                                        *sid = ctx->ctx_sid;
                                        sid_set = 1;

                                        if (!call)
                                                break;
                                } else if (*sid != ctx->ctx_sid)
                                        return -EINVAL;
                        }
                }
        }

        return 0;
}
```

（好像又绕回到了 ctx->ctx_sid 了！它又是如何确定的？！）

```
/**
 * security_net_peersid_resolve - Compare and resolve two network peer SIDs
 * @nlbl_sid: NetLabel SID
 * @nlbl_type: NetLabel labeling protocol type
 * @xfrm_sid: XFRM SID
 *
 * Description:
 * Compare the @nlbl_sid and @xfrm_sid values and if the two SIDs can be
 * resolved into a single SID it is returned via @peer_sid and the function
 * returns zero.  Otherwise @peer_sid is set to SECSID_NULL and the function
 * returns a negative value.  A table summarizing the behavior is below:
 *
 *                                 | function return |      @sid
 *   ------------------------------+-----------------+-----------------
 *   no peer labels                |        0        |    SECSID_NULL
 *   single peer label             |        0        |    <peer_label>
 *   multiple, consistent labels   |        0        |    <peer_label>
 *   multiple, inconsistent labels |    -<errno>     |    SECSID_NULL
 *
 */
int security_net_peersid_resolve(u32 nlbl_sid, u32 nlbl_type,
                                 u32 xfrm_sid,
                                 u32 *peer_sid)
{
        int rc;
        struct context *nlbl_ctx;
        struct context *xfrm_ctx;
```

```
        *peer_sid = SECSID_NULL;

        /* handle the common (which also happens to be the set of easy) cases
         * right away, these two if statements catch everything involving a
         * single or absent peer SID/label */
        if (xfrm_sid == SECSID_NULL) {
                *peer_sid = nlbl_sid;
                return 0;
        }

        /* NOTE: an nlbl_type == NETLBL_NLTYPE_UNLABELED is a "fallback" label
         * and is treated as if nlbl_sid == SECSID_NULL when a XFRM SID/label
         * is present */
        if (nlbl_sid == SECSID_NULL || nlbl_type == NETLBL_NLTYPE_UNLABELED) {
                *peer_sid = xfrm_sid;
                return 0;
        }

        /* we don't need to check ss_initialized here since the only way both
         * nlbl_sid and xfrm_sid are not equal to SECSID_NULL would be if the
         * security server was initialized and ss_initialized was true */
        if (!policydb.mls_enabled)
                return 0;

        read_lock(&policy_rwlock);

        rc = -EINVAL;
        nlbl_ctx = sidtab_search(&sidtab, nlbl_sid);
        if (!nlbl_ctx) {
                printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n", __func__, nlbl_sid);
                goto out;
        }

        rc = -EINVAL;
        xfrm_ctx = sidtab_search(&sidtab, xfrm_sid);
        if (!xfrm_ctx) {
                printk(KERN_ERR "SELinux: %s:  unrecognized SID %d\n", __func__, xfrm_sid);
                goto out;
        }
        rc = (mls_context_cmp(nlbl_ctx, xfrm_ctx) ? 0 : -EACCES);
        if (rc)
                goto out;

        /* at present NetLabel SIDs/labels really only carry MLS
         * information so if the MLS portion of the NetLabel SID
         * matches the MLS portion of the labeled XFRM SID/label
         * then pass along the XFRM SID as it is the most
         * expressive */
        *peer_sid = xfrm_sid;
out:
        read_unlock(&policy_rwlock);
        return rc;
}
```

而 netlabel SID 又时从何而来?

13.1.7 Labeled IPsec 环境的搭建

1，使能和 Labeled IPsec 相关的内核配置选项
除了使能和 ipv4/ipv6 和 IPsec 相关的若干选项之外，和必须使能如下 Labeled IPsec 相关选项：

```
CONFIG_SECURITY=y
CONFIG_SECURITY_NETWORK=y
CONFIG_SECURITY_NETWORK_XFRM=y
CONFIG_SECURITY_SELINUX=m/y
```

2，用户态 ipsec-tools 包的安装
ipsec-tools 包将提供 racoon 后台进程及其配置文件，以及 setkey 工具。所安装的包必须支持 Labeled IPsec 特性。

3，Labeled IPsec 相关的配置文件
在两个主机之间以 Transport 模式部署 IPsec，两台机器上 racoon.conf 的配置文件必须相同。比如：

```
[root/sysadm_r/s0@QtCao racoon]# cat racoon.conf
path include "/etc/racoon";
path pre_shared_key "/etc/racoon/psk.txt";
# path certificate "/etc/racoon/certs";
# path script "/etc/racoon/scripts";

remote anonymous
{
        exchange_mode main,aggressive;
        doi ipsec_doi;
        situation identity_only;

        my_identifier address;

        lifetime time 2 hours;
        initial_contact on;
        proposal_check obey;

        proposal {
                encryption_algorithm 3des;
                hash_algorithm sha1;
                authentication_method pre_shared_key;
                dh_group 2;
        }
}

sainfo anonymous
{
        pfs_group 2;
        lifetime time 1 hour;
        encryption_algorithm 3des, blowfish 448, rijndael;
        authentication_algorithm hmac_sha1, hmac_md5;
        compression_algorithm deflate;
}
```

其中 authentication_method = pre_shared_key; 即在 phase 1 中使用 pre_shared_key 文件来识别双方的身份。两个机器上的/etc/racoon/psk.txt 文件的内容必须相同，比如：

```
[root/sysadm_r/s0@QtCao racoon]# cat psk.txt
192.168.29.224 micropachycephalosaurus
192.168.29.223 micropachycephalosaurus
```

即由双方的 IP 地址，以及一个字符串组成。

在两个机器上都使用 setkey 工具来注册新的 SPD 条目。比如在 192.168.29.223 机器上，setkey.conf 文件的内容如下：

```
[root/sysadm_r/s0@QtCao racoon]# cat setkey.conf
spdflush;
flush;
```

```
spdadd  192.168.29.224 192.168.29.223 any -ctx 1 1 "system_u:object_r:ipsec_spd_t:s0" -P in ipsec esp/transport//require;
spdadd  192.168.29.223 192.168.29.224 any -ctx 1 1 "system_u:object_r:ipsec_spd_t:s0" -P out ipsec esp/transport//require;
```

setkey 的 spdadd 命令注册一个 SPD 条目，指名哪两个 IP 地址之间，什么协议（以及什么端口）上需要施加 IPsec 的什么服务，以及 IPsec 的模式等信息。比如， 对方机器的 IP 为 192.168.29.224，所以从.224 到.223 的数据报为接收；反之为发送。这两个 IP 地址之间的任何协议（的所有端口）上都施加 IPsec 服务，采用 IPsec 的 ESP 协议，IPsec 工作在 transport 模式。

注意在对方 192.168.29.224 机器上，需要把“-P in/out”对调即可。

注意，“-ctx”选项用于指明 SPD 条目（对应域 xfrm_policy 数据结构）的安全属性：doi = alg = 1，即 xfrm_sec_ctx 数据结构的使用者为 LSM/SELinux，“system_u:object_r:ipsec_spd_t:s0”即为该 SPD 对对象的安全上下文。

4，启动 Labeled IPsec
执行下面的命令安装 SPD，并重启 racoon：

```
chcon -t ipsec_conf_file_t /etc/setkey.conf            # in secadm_r
setkey -f /etc/setkey.conf                             # in secadm_r
run_init /etc/init.d/racoon restart                    # in sysadm_r
```

13.1.8 观察 Labeled IPsec 的行为

在上面执行“setkey -f /etc/setkey.conf”命令之后，可以使用“setkey -DP”命令观察注册的 SPD 条目及其 SELinux 标签：

```
192.168.29.224[any] 192.168.29.223[any] any
        out prio def ipsec
        esp/transport//require
        created: Jul 28 09:51:02 2011  lastused: Jul 28 10:01:38 2011
        lifetime: 0(s) validtime: 0(s)
        security context doi: 1
        security context algorithm: 1
        security context length: 33
        security context: system_u:object_r:ipsec_spd_t:s0
        spid=65 seq=9 pid=2834
        refcnt=2
192.168.29.223[any] 192.168.29.224[any] any
        in prio def ipsec
        esp/transport//require
        created: Jul 28 09:51:02 2011  lastused: Jul 28 10:01:47 2011
        lifetime: 0(s) validtime: 0(s)
        security context doi: 1
        security context algorithm: 1
        security context length: 33
        security context: system_u:object_r:ipsec_spd_t:s0
        spid=72 seq=10 pid=2834
        refcnt=2
192.168.29.223[any] 192.168.29.224[any] any
        fwd prio def ipsec
        esp/transport//require
        created: Jul 28 09:51:02 2011  lastused:
        lifetime: 0(s) validtime: 0(s)
        security context doi: 1
        security context algorithm: 1
        security context length: 33
        security context: system_u:object_r:ipsec_spd_t:s0
        spid=82 seq=0 pid=2834
        refcnt=1
```

注意其中“security context”的相关信息，即由 spdadd 命令的“-ctx”选项指定。此时由于双方尚未通信，所以没有 SA 被建立（“setkey -D”结果为空）。

在.224 机器上执行 server 程序，它创建一个 socket，bind 到自己的 IP 地址+3490 端口上，调用 listen

函数向内核注册为监听套接字，然后在循环中调用 accept 函数接收连接请求，在连接成功后调用 getpeercon 函数返回 server 端已连接套接字对方的安全上下文（从该套接字的 sksec->peer_sid 获得），然后 fork 子进程，将 client 端的安全上下文字符串回写给 client。（另外，fork 后子进程关闭监听套套接字，而父进程关闭和具体 client 相关的已链接套接字）

在.223 机器上执行 client 程序，它创建一个 socket 并发起对.224 机器的 IP 地址+3490 端口的连接。在连接成功后调用 recv 函数接收数据（即上面 server 端得到的 client 的套接字的安全上下文），并调用 getpeercon 函数得到 server 端的已连接套接字的安全上下问字符串，最后一并打印。

（client 和 server 程序的源码，请见 Joshua 的相关 blog:
http://securityblog.org/brindle/2007/05/28/secure-networking-with-selinux/）

在 client 发起对 server 的链接时使用 runcon 命令，使得 client 的 domain 为 newrole_t，而 server 的 domain 为 sysadm_t，以便区分。结果如下：

```
[root/sysadm_r/s0@cp3020-2-cqt racoon]# setkey -F
[root/sysadm_r/s0@cp3020-2-cqt racoon]# runcon -t newrole_t /root/client 192.168.29.224
Received: Hello, root:sysadm_r:newrole_t:s0-s15:c0.c255 from root:sysadm_r:sysadm_t:s0-s15:c0.c255

[root/sysadm_r/s0@cp3020-3 racoon]# /root/server
server: got connection from 192.168.29.223, root:sysadm_r:newrole_t:s0-s15:c0.c255
^C
[root/sysadm_r/s0@cp3020-3 racoon]#
```

红色为 server 上在建立链接后调用 getpeercon 函数得到的 client 端的 socket 的安全上下文，斜体为 server 回写给 client 的字符串，它们都和 client 实际的 SC 一样！而蓝色为 client 端在建立连接后调用 getpeercon 函数得到的 server 端的 socket 的安全上下文，也和 server 实际的 SC 一样！

getpeercon 函数的实现机制为：在 selinux_socket_getpeersec_stream 函数中即获得 sk_security_struct 数据结构的 peer_sid 域，并通过 security_sid_to_context 函数返回对应的安全上下文字符串，最终通过 copy_to_user 返回给用户态的 getpeercon 函数。

由此可见，通过 Labeled IPsec 可以获得对方的安全上下文，从而为 Distributed Authentication 提供了必要的物质基础。

进一步，可以观察 client 端执行后 SAD 的情况（server 端的类似。注意 client 的地址为.223）：

```
192.168.29.224 192.168.29.223
        esp mode=transport spi=203043989(0x0c1a3495) reqid=0(0x00000000)
        E: 3des-cbc  141747fd 6ad5876c d6b33dfb 6f6491a1 dc822ada 8add9e0b
        A: hmac-sha1  7ee60550 86007428 8b29db45 4ad4afa8 082cbc3b
        seq=0x00000000 replay=4 flags=0x00000000 state=mature
        created: Aug 15 04:32:57 2011  current: Aug 15 04:33:05 2011
        diff: 8(s)hard: 3600(s)       soft: 2880(s)
        last:                         hard: 0(s)soft: 0(s)
        current: 0(bytes)   hard: 0(bytes)      soft: 0(bytes)
        allocated: 0        hard: 0   soft: 0
        security context doi: 1
        security context algorithm: 1
        security context length: 39
        security context: root:sysadm_r:newrole_t:s0-s15:c0.c255
        sadb_seq=1 pid=21432 refcnt=0
192.168.29.223 192.168.29.224
        esp mode=transport spi=158452324(0x0971ca64) reqid=0(0x00000000)
        E: 3des-cbc  b16e80e0 02b40d82 bea8a962 c41f812d 1be22ecb e374eff8
        A: hmac-sha1  ca269b58 3745f14e a2599273 b1439563 a14ba8ef
        seq=0x00000000 replay=4 flags=0x00000000 state=mature
        created: Aug 15 04:32:57 2011  current: Aug 15 04:33:05 2011
        diff: 8(s)hard: 3600(s)       soft: 2880(s)
```

```
        last: Aug 15 04:32:57 2011    hard: 0(s) soft: 0(s)
        current: 176(bytes) hard: 0(bytes)      soft: 0(bytes)
        allocated: 5        hard: 0   soft: 0
        security context doi: 1
        security context algorithm: 1
        security context length: 39
        security context: root:sysadm_r:newrole_t:s0-s15:c0.c255                    # SAout
        sadb_seq=2 pid=21432 refcnt=0
192.168.29.224 192.168.29.223
        esp mode=transport spi=232248110(0x0dd7d32e) reqid=0(0x00000000)
        E: 3des-cbc  ccbf4231 1b5cc2e8 667a8484 a4a1e3bc 12f4c4ee 8bfe7a1b
        A: hmac-sha1  1eacf2ae 1c638c59 5a56a6c4 e815eab9 b49b8e6f
        seq=0x00000000 replay=4 flags=0x00000000 state=mature
        created: Aug 15 04:32:57 2011  current: Aug 15 04:33:05 2011
        diff: 8(s) hard: 3600(s)       soft: 2880(s)
        last: Aug 15 04:33:00 2011    hard: 0(s) soft: 0(s)
        current: 181(bytes) hard: 0(bytes)      soft: 0(bytes)
        allocated: 4        hard: 0   soft: 0
        security context doi: 1
        security context algorithm: 1
        security context length: 38
        security context: root:sysadm_r:sysadm_t:s0-s15:c0.c255                     # SAin
        sadb_seq=3 pid=21432 refcnt=0
192.168.29.223 192.168.29.224
        esp mode=transport spi=171682443(0x0a3baa8b) reqid=0(0x00000000)
        E: 3des-cbc  2464b0c4 d59edae9 c8d0e80c 20841ecb 4866f55a f510c4bd
        A: hmac-sha1  dd9fdf2a 52ae199b d3a0d964 b400be68 84058028
        seq=0x00000000 replay=4 flags=0x00000000 state=mature
        created: Aug 15 04:32:57 2011  current: Aug 15 04:33:05 2011
        diff: 8(s) hard: 3600(s)       soft: 2880(s)
        last:                          hard: 0(s) soft: 0(s)
        current: 0(bytes)   hard: 0(bytes)      soft: 0(bytes)
        allocated: 0        hard: 0   soft: 0
        security context doi: 1
        security context algorithm: 1
        security context length: 38
        security context: root:sysadm_r:sysadm_t:s0-s15:c0.c255
        sadb_seq=0 pid=21432 refcnt=0
```

就 client 而言，client 程序的 domain 为 newrole_t，所以其 SAout 也为 newrole_t；而 server 程序的 domain 为 sysadm_t，所以 client 上 SAin 就为 sysadm_t。

（问题：为什么有 4 个 SA？另外两个是怎么回事？）

13.1.9 和 Labeled IPsec 相关的 SELinux 规则

在上面的例子中，在 client 端必须使能如下和 Labeled IPsec 相关的 SELinux 规则（不包括 compat-net 所涉及的其他规则）

1, allow newrole_t ipsec_spd_t:association polmatch;
许可 client domain / socket type 能够使用 SPD 条目（标签为 ipsec_spd_t）。

2, allow newrole_t self:association sendto;
许可 client domain / socket type 能够从标签为 newrole_t 的 SA 上发送数据报，即 newrole_t domain 能够使用它自己创建的 SA 发送数据。

注意，sendto 权限并不能控制 newrole_t domain 能够给哪个 domain 发送数据。

3, allow newrole_t sysadm_t:association recvfrom;
许可 client domain / socket type 能够从标签为 sysadm_t 的 SA 上接收数据报。

注意，标签为 sysadm_t 的 SA，即为远端发送者的 domain。**正是通过 recvfrom 权限来控制本地的什么进程（newrole_t）能够和远程的什么进程（sysadm_t）进行通信！**

4, allow {setkey_t racoon_t} {ipsec_spd_t domain}:association setcontext;
许可 racoon 和 setkey 命令能够修改 SPD（ipsec_spd_t）和 SAD（可能是任何 domain）。

参考文献

（感谢现在能够有如此丰富的网络资源可以获得 SELinux 的相关知识，感谢 SELinux 和 refpolicy 邮件列表及其上乐于助人的前辈们，感谢 RedHat 公司和 Tresys 公司对 SELinux 发展所做的杰出贡献！）

1，《SELinux By Example》
A must-have reference about SELinux，基本概念、语法、SELinux 的编写和使用，无所不包。
注意，如果需要查找某个 permisson:class 的含义，非本书莫属！

2，http://docs.fedorapro.ject.org/en-US/Fedora/13/html/Security-Enhanced_Linux/index.html
Fedora 13 SELinux User Guide；

http://docs.fedorapro.ject.org/en-US/Fedora/13/html/Managing_Confined_Services/index.html
Fedora 13 SELinux User Guide about how to configure system services(http/samba/ftp 等)，由具体部署情况设置相应的 SELinux Boolean，或用 semanage fcontext/port 修改当前 policy。

注意，所有的 fedora documentation 都有 single-html/pdf 格式下载。

3，http://userspace.selinuxpro.ject.org/trac/wiki/Releases
refpolicy 发行包，SELinux 用户态工具/库的发行包；

4，http://www.nsa.gov/research/selinux/index.shtml
NSA SELinux main website；

5，SELinux@tycho.nsa.gov
SELinux 邮件列表，Stephan Smalley 等人很乐于助人；

refpolicy@oss.tresys.com
refpolicy 邮件列表，所有有关 reference policy 的问题都应该同时 CC 给该邮件列表。
Chris PeBenito（c.jp）是 refpolicy.git 的 maintainer，他和 Dominick Grift 等人都能够热心地回答问题

6，SELinux 相关 git tree：
用户态：
git clone http://oss.tresys.com/git/refpolicy.git
git clone http://oss.tresys.com/git/selinux.git

如果怀疑用户态包、库有问题，或者 refpolicy 有问题，可以 clone 最新的并用 git log 查看相应问题是否已经被解决。

内核态：
git clone git://git.infradead.org/users/eparis/selinux.git

当前 SELinux 内核驱动由 Eric Paris（RedHat）来维护，提交的 patch 要先集成到他的 tree 里面，然后再向 Linus 提交。因此这个 tree 里面的代码可能会比 Linux git tree 的更新。

7，Linux 系统编程（Linux System Programming）第 7 章中《Extended Attributes》关于 POSIX 文件扩展属性的小节。

8, 《Extending SELinux to meet LSPP data import/export requirements 》
有关 SELinux 上 polyinstantiation 的实现和对 cron 的修改以支持在任务提交者的 SC 中执行 cronjob

9, http://www.gnu.org/software/m4/manual/index.html
有关 m4 语法的介绍

10, 《SELinux Memory Protection Tests》
Ulrich Drepper 写的 SELinux 上内存保护相关的文章, 介绍
allow_execmod/execmem/execheap/execstack 布尔变量及相应对内存访问方式的限制。

11, 《Implementing SELinux as a Linux Security Module》
介绍 SELinux 实现的 LSM 各个回调函数的主要行为, 对核心内核数据结构的扩展等内容。

12, Labeled Networking 相关:
http://selinuxproject.org/page/NB_Networking
综述 SELinux 对网络的控制, 包含下面所有文档的引用。

《IPsec HOWTO》
介绍 IPsec 概念和部署的好资料。

《rfc2401.txt》
IETF 官方 IPsec 规范描述。

《Leveraging IPsec for Distributed Authorization》
LSM 向 XFRM 中注册的回调函数的作用

13, SELinux on Android
http://selinuxproject.org/page/SEAndroid

简化 SELinux 操作的 bash 配置方法

可以设置 bash 的 PS1 变量，在命令行提示符中显示当前 Linux 用户所扮演的 role 以及 MLS attribute:

```
[root/sysadm_r/s0@~]# vim .profile
...
if [ ! -z "$PS1" ]; then
   PS1="[\u/\$(secon -rP 2>/dev/nu11)/\$(secon -lP 2>/dev/nu11)@\W]\\$ "
   export PS1
fi

alias audhigh="newrole -r auditadm_r -l sl5:c0.cl023 -- -c"
alias seclow="newrole -r secadm_r -- -c"
alias setenforce0="newrole -r secadm_r -- -c \"setenforce 0\""
[root/sysadm_r/s0@~]#
```

另外上述 alias 有助于简化在命令行切换到其他角色及相应的安全级别的输入。

本文档各个版本的说明

作者从 2009 年 5 月份开始和 SELinux 相关的开发工作，至此已经将近 3 年。期间坚持将自己工作学习中的心得收获总结成文。特此将之前各个版本的说明文字收录如下，以献给这段颇具收获的快乐时光。衷心祝愿各位读者学习工作顺利，心想事成！

2010-6-17

最近一年多一直在做和 SELinux 有关的东西，这里把自己对《SELinux By Example》一书的学习笔记和一些实际经验总结共享给大家。希望得到同行的批评指正，欢迎相互交流:-）

由于向 linuxforum 上传文件有大小的限制，故放到了这里：
http://linux.3xin2yi.info/SELinux/SELinux_notes.pdf

距离上一次在 LinuxForum 上分享自己的文档已四年有余，真是岁月如织、人生如歌呀！

2011-1-3

把自己最新的《SELinux 学习笔记》上传，还在原来的地址上。

和 2010 年 6 月份的相比，主要变化是：

1，增强了关于 SELinux 的实践经验总结。后来把 vlock.pp 和 samhain.pp 贡献到了 refpolicy 社区里面去，在此过程中得到了 cjp 和 Dominik Grift 无私的建议，特此把这两个 pp 的开发过程详实地收入文档；

2，增加了部分关于 SELinux 内核驱动的分析，比如在父进程在 fork 子进程，后者 exec 新的可执行文件时，缺省的 type_transition 规则是如何生效的？在创建新的文件时，它的标签又是如何确定的？用户进程事先写入其/proc/<pid>/attr/fscreate 文件的内容是如何生效的？在相应 pp 中指定的关于文件的 type_transition 规则如何生效？

该文档一直处于不停的变化过程中，自己对 SELinux 的造诣很浅薄，错误难免，希望得到大家的批评指正！

2011-5-27

把自己最新的《SELinux 学习笔记》上传，还在原来的地址上。

和 2011 年 1 月份的相比，主要新增内容如下：

1．第 9.2 小节 Socket Labeling 的开发，允许一个进程创建的 socket 具有和进程 domain 不一样的 type，这样就可以支持将 socket 的 type 和进程的 domain 加入到不同的 MLS 属性；

2．第 9.3 小节给 role_transition 规则添加对 class 的支持，从而在 role_transition 规则中可以指定新创建文件的 class，使得新创建文件的 role 可以不再是默认的 object_r，而被匹配的 role_transition 规则重载。（之前 role_transition 规则默认只能对 process 类生效，用于指定一个进程在运行一个可执行文件期间的 role）

这两部分开发都已经被社区所接纳，具体的 patch 可以从相应的 git tree 上得到。具体分析，实现，测试过程都已翔实地记录在笔记中了。

3．第 12 章 refpolicy 的编译、链接、扩展。这是一个全新的章节，涵盖 50%相关核心源代码的情景分析，包括标识符的描述数据结 构，refpolicy 的编译，模块的链接和扩展，policy.X 的创建。至此对 refpolicy 的 compile/link/expand 过程有了 一个相对全面的理解。

该文档一直处于不停的变化过程中，自己对 SELinux 的造诣很浅薄，错误难免，希望得到大家的批评指正！

2012-2-2

把自己最新的《SELinux 学习笔记》上传，还在原来的地址上。

和 2011 年 1 月份的相比，主要新增内容如下：

1，重新梳理了大部分章节的文字，加入了前后章节相关内容之间的索引。对于改动较大的小节，标记以“Revisited”。对于增加的小节，标记以“new”；

2，第 9.4 小节 role attribute 的开发。之前的 SELinux toolchain 只支持就 type 定义属性，现在也支持就 role 定义属性。在开发 role attribute 的过程中能够深刻体会到“属性”所具有的两个特征：一组具有相同共性的集合；该集合所具有的共性。比如一个 role 属性能够包含许多普通 role 或其他 role 属性，在编译过程中，该 role 属性能够关联的 type 的集合将被散播给其所有成员 role（如果成员 role 为 role 属性，则进一步展开直至最末端普通 role），从而使它的所有普通成员 role 都能够和这些 type 相关联。role 属性的使用可以简化 run 接口的设计和实现。

该新特性已经合并到 SELinux 社区中，并且 cjp 也使用 role attribute 修改了许多 pp 的 run 接口的实现。

3，第 9.5 小节区分 tunable 和 boolean。之前在 refpolicy 中使用“bool”关键字来定义 boolean 和 tunable。而实际上真正需要在运行时切换状态的（从而控制某些规则是否被使能）只有 boolean。而在编译时即可根据所部署的环境的需要，确定 tunable 的逻辑值，进而控制相应 conditional policy 中哪个分支的规则被需要。对于需要的规则，可以视为 unconditional 的并永久写入 policy.X。而对于不需要的规则以及 tunable 标识符本身，在编译时即可丢弃，即不写入 policy.X。这样依赖可以进一步减少 policy.X 的大小 30%以上（这是因为在 refpolicy 所定义的数十个 tunable 中，只有少数定义为 true 大部分定义为 false，而且许多 tunable 的 conditional policy 都没有 false 分支）。

该新特性相关的 SELinux toolchain 机制已经被社区接受。截止目前 refpolicy 社区尚未开始使用“tunable”关键字来定义 tunable（仍然使用 bool 来定义），所以相应机制默认不生效。

4，第 12.7 小节 link 和 expand 过程的图解。在第 12 章已有内容的基础上，图解 link 和 expand 过程主要函数的调用时机及其行为，进一步加深了对相关源代码的理解（所以才能顺利完成 9.4 和 9.5 小节的开发任务）

5，第 12.8 小节 m4 宏的定义和展开。大致分析了 refpolicy 规则中常用的 gen_require，policy_module，interface 等宏的定义。

6，第 13.1.1 小节关于 Labeled Networking 的内容。大致描述了 Labeled Networking 的相关应用。由于自己目前对 IPsec 和内核 xfrm 框架都缺乏足够的了解，所以该小节的内容有待改进。

该文档一直处于不停的变化过程中，自己对 SELinux 的造诣很浅薄，错误难免，希望得到大家的批评指正！

（全文完）