

序言

本书是基于我们多年在 SELinux 上工作、开发以及帮助推进安全增强的 Linux (SELinux) 的经验写成的。我们也创建了关于 SELinux 的技术资源，在我们多年的教学经历中，我们发现要对新的听众完整介绍新的和国外关于计算机安全思想时，显得非常困难。在这本书中，我们认为在概念叙述和具体实际演练之间达到了很好的平衡。

本书的另一个挑战是 SELinux 是一门新技术；尽管它已经被集成到主流的 Linux 发行版中，但它仍然在不断发展。我们和其他 SELinux 开发者正在进行许多创新的研究和项目开发，以多种方式提高 SELinux。在这本书中，我们面临描述一个活的目标的挑战。幸运的是，SELinux 的核心概念已经明确地建立起来了，至少安全增强的核心部分是以易管理的步子在迈进。

读者群

本书主要是为那些需要使用安全增强的 Linux (SELinux) 的人准备的，他们主要对理解、编写、修改和/或管理 SELinux 策略感兴趣。如果你想使用 SELinux 增强你的应用程序、系统和网络的安全，那么你就是本书的合适读者了。

为了有效使用这本书，你应该对 Linux/Unix 非常了解，并且非常熟悉与 Linux 内核和关键服务交互工作，这有助于你快速理解 SELinux 使用的安全对象模型。不过，只要你对 Linux、它的标准、文件系统布局以及它的编程风范了解深刻，读懂本书的内容应该没什么问题。

包含了 SELinux 的系统(如 Red Hat Enterprise Linux, Fedora Core, Gentoo 和 Debian) 中的用户也会发现本书非常有用。尽管大部分用户和系统管理员都不大可能编写 SELinux 策略，但理解 SELinux 策略语言和安全模型将会使你更深入地认识到 SELinux 在给予你安全方面的强大功能。

你将会学到什么

这本书主要描写如何编写 SELinux 安全策略，有效使用 SELinux 给 Linux 带去安全增强。听起来很简单，但实际上，在你能够帮助你理解如何有效使用这些增强之前，你不得不学习新的思想和理解 SELinux 策略语言。

围绕你的学习步骤，我们将本书划分为三部分，具体如下：

第一部分

强制访问控制概述

类型增强概念和应用程序

SELinux 架构和机制

第二部分

SELinux 自然策略语言语法和语义详解

SELinux 中的对象标记

第三部分

两个主要的创建 SELinux 策略的开发方法：样例策略和引用策略

SELinux 对系统管理的影响

如何为 SELinux 编写策略模块

我们的目标是帮助你理解包括在 SELinux 中的详细信息，以便你有能力创建安全的系统。为了引出 SELinux 的新特性，我们必须向你提供初级策略语言的详细内容。不过，请记住正在进行的许多工作将会使构建安全系统变得非常容易，而且不需要知道所有的初级知识。我们将在合适的地方描述这项正在进行的工作，帮助你理解如何编写能够通过独立审查的安全策略。

每章都以该章描述的关键点摘要回顾和加深对这些关键点理解的练习收尾，从实验练习思想到实际练习，再到修改真实的安全策略。这一切将会帮助你加深对 SELinux 的理解。

章节摘要

我们将本书划分为三部分，每部分又包括多个章节：

第一部分：“SELinux 概述”。这部分描述了 SELinux 发展的背景，以及它的安全概念和架构。

第 1 章：“背景”。在这一章中，我们描述了操作系统访问控制的发展历史，访问控制机制的种类，以及 SELinux 带给 Linux 的访问控制种类。

第 2 章：“概念”。在这一章中，我们以详细教程的方式对 SELinux 安全机制的概念进行了概述。对 SELinux 带给 Linux 的安全增强做了简单明了的描述。

第 3 章：“架构”。在这一章中，我们概述了 SELinux 的架构和实现，以及策略语言。

第二部分：“SELinux 策略语言”。这部分包括了对 SELinux 策略语言语法和语义详细的描述。每章代表语言的一部分。本书的这一部分可以当作策略语言的参考。

第 4 章：“对象类和许可”。在这一章中，我们描述了 SELinux 如何使用对象类和给对象类定义的细粒度许可控制内核资源。

第 5 章：“类型增强策略”。在这一章中，我们描述了所有核心策略语言规则和编写类型增强策略的指令。类型增强是 SELinux 最重要的访问控制特性。

第 6 章：“角色和用户”。在这一章中，我们描述了 SELinux 基于角色的访问控制机制，以及策略语言中的角色和用户如何支持类型增强策略。

第 7 章：“约束”。在这一章中，我们描述了 SELinux 策略语言的约束特性，即在支持强制策略类型的策略内提供约束。

第 8 章：“多级安全”。在这一章中，我们描述了除核心强制访问控制之外的，允许非强制的多级安全访问控制的策略语言特性。

第 9 章：“条件策略”。在这一章中，我们描述了策略语言的增强，使我们可以在类型增强策略中应用布尔表达式，布尔表达式的值在生产系统上，在运行过程中可以被改变。

第 10 章：“对象标记”。在这一章中，我们结束了对策略语言的描述，同时描述了如何标记对象，以及如何在 SELinux 增强的访问控制支持下管理那些标记。

第三部分：“创建和编写 SELinux 安全策略”。在这最后一部分中，我们向你展示如何使用策略语言，同时描写了建立安全策略的方法，以及如何管理一个 SELinux 系统和调试 SELinux 策略模块。

第 11 章：“最早的样例策略”。在这一章中，我们描述了样例策略，它是一个创建 SELinux 策略的方法（源文件，构建工具和示范等），自从美国国家安全局（NSA）释放出最初的样例策略以来，已经经过多年的发展和改进。

第 12 章：“参考策略”。在这一章中，我们描述了一个新的创建 SELinux 策略的方法，它提供了所有样例策略的特性。最近发布的 Fedora Core 5 就是使用参考策略作为它的策略基础的。

第 13 章：“管理 SELinux 系统”。在这一章中，我们描述了 SELinux 如何影响 Linux 系统的管理的。

第 14 章：“编写策略模块”。在这最后一章中，我们利用在本书中学到的所有知识总结成一个向导式的指南，指导如何为样例策略和参考策略编写策略模块。

附录。本书结尾包括了几个附加的参考资料的附录：

附录 A：“获取 SELinux 样例策略”。提供了关于如何获取本书中描述过的样例策略源文件的说明。

附录 B：“参与和额外信息”。列出了关于 SELinux 的额外信息源，以及如何参与 SELinux 的开发。

附录 C: “对象类参考”。提供了一个详细的关于 SELinux 内核对象类和关联的许可的字典。

附录 D: “SELinux 命令和实用程序”。提供了一些实用程序和第三方工具，帮助开发 SELinux 策略和管理 SELinux 系统。

如何使用这本书

很少有人翻来覆去地阅读一本技术书籍。大多数人都只想理解某个特定的知识点或开始探索一下新技术。尽管反复阅读确实是可取的，这里我们也给出一种备选的方法。

透彻阅读并理解第一部分的内容(第 1-3 章)。这一部分提供了必要的背景知识和概念，对深入理解 SELinux 是很有帮助的。特别要仔细阅读和理解第 2 章。你可能想撇去第二部分(第 4-10 章)的内容，这一部分主要讲解了 SELinux 策略语言。对大多数人而言，这一部分的内容确实显得太深入了，特别是对于初次接触 SELinux 的人更是如此。因此，你可以跳过第 4 章和第 10 章，但要仔细阅读第 5 章。这些章节覆盖了几几乎所有 SELinux 策略语言元素，在编写策略时就会使用到。最后，阅读第三部分的所有章节(第 11-14 章)，描述了你感兴趣的问题。阅读这些章节时使用第二部分作参考。

侧边栏，注意，警告和提示

贯穿本书，我们广泛使用了侧边栏和注解以提供附加的信息或强调某个项目，也包括了大量的警告和提示。下面是它们在本书中的约定。

侧边栏：我们使用侧边栏主要出于两个目的。首先，使用它描述在章节主体内容中没有直接涉及到附加信息。例如：我们使用侧边栏列出不同 SELinux 版本之间的差异或深入描述读者感兴趣的某个特定的概念。在第二部分中，我们使用侧边栏来描述所有 SELinux 策略语言语句的完整语法。这些语法侧边栏为大量策略语言元素提供了快速参考。

注意：我们使用注意为某个特定知识点提供了附加的强调。通常注意是非常短的附加说明。

警告：警告与注意的使用类似，只是它更着重强调或指出需要额外小心。

提示：为如何执行一个给定的功能提供快速提示和建议，或使某事变得容易的技巧。

排版约定

所有技术书籍都必须使用某种排版约定，以便于与读者更好地沟通。我们使用斜体字表示定义概念时的一个关键概念(通常是首次使用时定义)，同时也使用斜体字表示强调。对于着重强调的地方，我们使用粗体字。

对所有 SELinux 策略语言元素 (allow)，用户命令 (ps, ls) 或你输入的内容或你在电脑显示屏上看到的内容使用固定宽度的字体。

对于显示命令及其输出的长清单，我们使用 Bourne shell 标准提示符# (root shell) 和\$ (普通用户 shell)。用户输入使用粗体加固定宽度字体。例如：

```
# ls -lZ /etc/selinux/
-rw-r--r--
root root system_u:object_r:selinux_config_t
config
drwxr-xr-x
root root system_u:object_r:selinux_config_t
strict
drwxr-xr-x
root root system_u:object_r:selinux_config_t
targeted
```

谈论到库函数或系统调用时，我们约定使用一对空括号，如 `execve()`，对策略带有参数的宏也使用这个约定，如 `domain_auto_trans()`。当参考 Linux 命令或函数的帮助手册时，对命令或函数使用斜体字，并用括弧将手册小节括起来。如 `make(1)`，`execve(2)`。

从哪里获取 SELinux

SELinux 在多个 Linux 发行版中受到支持，包括 Red Hat Enterprise Linux, Red Hat Fedora Core, Gentoo 和 Debian。Fedora Core 已经成为 SELinux 社区测试和集成大部分创新技术的主要平台。Red Hat Enterprise Linux 版本 4 (RHEL4) 是第一个完全支持 SELinux 的大型商业发行版。我们在本书中描述的所有内容都与 RHEL4 和其他发行版有关。

我们选择 Fedora Core 4(FC4)作为本书的基础，它是 RHEL4 之后释放出来的一个 Fedora Core 版本。我们描述的所有内容都可以在 FC4 系统上重现。在我们花了八个月编写此书期间，FC4 在不断发展、测试、发布。当我们写完此书时 Fedora Core 5 (FC5) 刚刚发布。FC5 集成了许多 SELinux 新的特性，FC5 的特性可能暗示了 RHEL5 将会更新的内容。实际上，我们在本书中标注了 FC4 中没有 FC5 中的新特性和功能。同样，我们也标注了 FC4 中有但 RHEL4 中没有的特性。

如果你是一个企业用户或开发人员，你很可能正在使用 RHEL4 或计划使用 RHEL5。目前我们使用 RHEL4 开发我们的企业应用产品。如果你是一名 SELinux 的开发人员或早期使用者，你可能正在使用某 Fedora Core 版本或某些其他发行版。无论你是哪种情况，本书都将会向你提供大量的关于如何使用 SELinux 和开发 SELinux 策略的信息。

如何取得本书中的样例策略

贯穿本书，我们给出了大量的 SELinux 策略样例。这些样例基于 Red Hat 发布 Fedora Core 4 附带的 strict 策略。我们在第 11 章详细地介绍了这个策略。FC4 默认使用的是 target 策略，而不是 strict 策略，因此你必须用更多的步骤来获得我们样例使用的基础策略。在第三部分中，我们扩大了视野，涵盖了其他类型的策略。在附录 A 中，我们提供了如何获得本书使用到的样例策略源文件的说明。

目录

第一部分：SELinux 摘要	1
第 1 章. 背景	1
1.1. 软件失效的必然性	1
1.2. 操作系统访问控制安全的进展	2
1.2.1. 引用监视程序原理	2
1.2.2. 任意访问控制的问题	3
1.2.3. 强制访问控制的起源	3
1.2.4. 更好的强制访问控制	4
1.2.5. SELinux 的发展	5
1.3. 小结	6
练习	6
第 2 章：概念	1
2.1. 类型强制的安全上下文	1
2.1.1. 对比 SELinux 和标准 Linux	2
2.1.2. 安全上下文	2
2.2. 类型强制访问控制	3
2.2.1. 类型强制示例	4
2.2.2. 域转变的问题	5
2.2.3. 标准 Linux 安全中的 setuid 程序	6
2.2.4. 域转变	7
2.2.5. 默认域转变：type_transition 指令	9
2.3. 角色	9
2.4. SELinux 中的多层安全	10
2.5. 精通 SELinux 特性	11
2.5.1. 重游 passwd 示例	12
2.5.2 精读策略文件	12
2.6. 小结	13
练习	14
第 3 章. 架构	1

3.1. 内核架构	1
3.1.1. LSM 框架	1
3.1.2. SELinux LSM 模块	2
3.2. 用户空间客体管理器	4
3.2.1. 用户空间客体管理器的内核支持	4
3.2.2. 策略服务器架构	5
3.3. SELinux 策略语言	6
3.3.1. 本地 SELinux 策略语言编译器	6
3.3.2. 单个策略中的源策略模块	8
3.3.3. 载入式策略模块	8
3.3.4. 构建和安装单策略	8
3.4. 小结	10

练习 10

第二部分：SELinux 策略语言 1

第4章. 客体类别和许可 1

4.1. SELinux 中客体类别的用途	1
4.2. 在 SELinux 策略中定义客体类别	2
4.2.1. 声明客体类别	2
4.2.2. 声明并连接客体类别许可	3
4.3. 有效的客体类别	5
4.3.1. 与文件相关的客体类别	5
4.3.2. 与网络有关的客体类别	7
4.3.3. system V IPC 客体类别	8
4.3.4. 其它杂类客体类别	8
4.4. 客体类别许可示例	9
4.4.1. 文件客体类别许可	9
4.4.2. 进程客体类别许可	11
4.5. 使用 Apol 研究客体类别	14
4.6. 小结	16

练习 16

第5章-类型强制 1

5.1. 类型强制	1
5.2. 类型、属性和别名	2
5.2.1. 类型声明	2
5.2.2. 类型和属性	3

5.2.3. 关联类型和属性	4
5.2.4. 别名	5
5.3. 访问向量规则	6
5.3.1. 通用 AV 规则语法	7
5.3.2. 允许 (allow) 规则	11
5.3.3 审核 (audit) 规则	11
5.3.4. neverallow 规则	13
5.4. 类型规则	14
5.4.1. 通用类型规则语法	15
5.4.2. 类型转换规则	16
5.4.3. 类型改变规则	18
5.5. 用 Apo1 研究类型强制规则	19
5.6. 小结	21

练习 22

第 6 章. 角色和用户 1

6.1. SELinux 中基于角色的访问控制	1
6.1.1. SELinux 中 RBAC 概述	2
6.1.2. 用角色管理用户权限	3
6.1.3. 客体安全上下文中的用户和角色	4
6.2. 角色和角色语句	4
6.2.1. 角色声明语句	4
6.2.2. 角色 allow 规则	5
6.2.3. 角色转换规则	5
6.2.4. 角色控制语句	6
6.3. 用户和用户语句	7
6.3.1. 声明用户及其关联的角色	7
6.3.2. 将 Linux 用户映射到 SELinux 用户	8
6.4. 用 Apo1 分析角色和用户	8
6.5. 小结	10

练习 11

第 7 章. 约束 1

7.1. 近距离查看访问决定算法	1
7.2. 约束语句	2
7.3. 标记转换约束	5
7.4. 小结	8

练习 8

第 8 章. 多层安全 1

8.1. 多层安全约束 1

8.2. 开启了 MLS 后的安全上下文 1

8.2.1. 安全级别定义 2

8.2.2. MLS 对安全上下文的扩展 4

8.3. MLS 约束 5

8.3.1. mlsconstrain 语句 5

8.3.2. mlsvalidatetrans 语句 8

8.4. MLS 的其它作用 10

8.5. 小结 11

练习 11

第 9 章. 条件策略 1

9.1. 条件策略概述 1

9.2. 布尔变量 2

9.2.1. 布尔变量定义 2

9.2.2. 在运行系统中关联布尔变量 2

9.2.3. 对布尔值的永久性改变 4

9.3. 条件语句 6

9.3.1. 条件表达式和规则列表 6

9.3.2. 条件语句限制 9

9.3.2.1. 支持的语句 9

9.3.2.2. 嵌套条件语句 10

9.4. 使用 Apol 检查布尔和条件策略 10

9.5. 小结 14

练习 14

第 10 章. 客体标记 1

10.1. 客体标记简介 1

10.2. 与文件有关的客体标记 2

10.2.1. 扩展属性的文件系统 (fs_use_xattr) 4

10.2.1.1. 扩展属性文件系统的标记行为 5

10.2.1.2. 在扩展属性文件系统中管理安全上下文 (文件上下文) 6

10.2.2. 基于任务的文件系统 (fs_use_task) 7

10.2.3. 基于转换的文件系统 (fs_use_trans) 7

10.2.4. 普通安全上下文标记 (genfscon) 8

10.2.4.1. genfscon 语句细粒度标记	8
10.2.4.2. 使用 genfscon 语句标记传统文件系统	10
10.3. 网络和套接字客体标记	10
10.3.1. 网络接口标记 (netifcon)	11
10.3.2. 网络节点标记 (nodecon)	11
10.3.3. 网络端口标记 (portcon)	12
10.3.4. 套接字标记	13
10.4. System V IPC	14
10.5. 其它客体标记	14
10.5.1. capability 客体标记	15
10.5.2. process 客体标记	15
10.5.3. system 和 security 客体标记	15
10.6. 初始安全标识符	15
10.7. 使用 Apol 研究客体标记	17
10.8. 小结	18
练习	19
第三部分：创建和编写 SELinux 安全策略	1
第 11 章. 原始示例策略	1
11.1. 管理构建过程的方法	1
11.2. strict 示例策略	2
11.2.1. 策略源文件结构概述	3
11.2.1.1. 客体类别和许可定义	4
11.2.1.2. 域类型和策略规则	4
11.2.1.3. 独立的资源类型	5
11.2.1.4. 其它顶层文件和目录	5
11.2.1.5. 安全上下文标记	6
11.2.1.6. 应用程序配置文件	7
11.2.2. 分析示例策略模块	7
11.2.2.1. 定义类型和域	9
11.2.2.2. 指定域转换规则	10
11.2.2.3. 条件策略示例	11
11.2.2.4. ping 命令的网络和其它访问	11
11.2.2.5. 审核规则	12
11.2.2.6. 文件安全上下文标记	12
11.2.3. strict 示例策略构建选项	12

11.2.3.1. 配置策略模块	12
11.2.3.2. 开启可选的 MLS 特性	13
11.2.3.3. 构建时可调选项	13
11.3. targeted 示例策略	14
11.4. 小结	15
练习	15
第 12 章 . 引用策略	1
12.1. 引用策略的目的	1
12.2. 策略源文件结构概述	2
12.2.1. 构建和支持文件	2
12.2.2. 核心策略文件	3
12.3. 设计原则	3
12.3.1. 分层	3
12.3.2. 模块化	5
12.3.2.1. 封装	5
12.3.2.2. 提取	5
12.3.2.3. 模块文件	6
12.3.2.4. 接口	6
12.3.2.4.1. 访问接口	6
12.3.2.4.2. 模板接口	9
12.4 分析引用策略模块	12
12.5. 引用策略构建选项	18
12.5.1. build.conf	18
12.5.2. modules.conf	19
12.6. 小结	20
练习	21
第 13 章. 管理 SELinux 系统	1
13.1SELinux 配置和策略管理文件	1
13.1.1. SELinux 配置文件 (/etc/selinux/config)	2
13.1.2. 策略目录	3
13.1.2.1. 安装的布尔变量文件	4
13.1.2.2 应用程序和文件安全上下文	5
13.1.2.3. SELinux 用户定义	6
13.1.2.4. SELinux 文件系统	7
13.2. SELinux 对系统管理的影响	7

13.2.1. 管理用户	8
13.2.1.1 添加普通的非特权用户	8
13.2.1.2. 添加特权用户	9
13.2.1.3. 修改用户角色	11
13.2.2. 理解审核消息	11
13.2.2.1. 常见的 SELinux 审核消息	11
13.2.2.2. AVC 消息	14
13.2.2.3. 使用 Seaudit 查看审核消息	16
13.2.3. 问题修复：与文件有关的客体标记	16
13.2.3.1. 与文件有关的客体标记命令	17
13.2.3.2. 自动重新标记	19
13.2.4. 管理多个策略	19
13.3. 小结	20
练习	20
第 14 章 . 编写策略模块	1
14.1. 策略模块编写概述	2
14.2. 准备和计划	2
14.2.1. 收集应用程序信息	2
14.2.2. 创建测试环境	3
14.2.3. 指定安全目标	7
14.3. 创建一个初始化策略模块	8
14.3.1. 创建策略模块文件	8
14.3.1.1. 示例策略	8
14.3.1.2. 引用策略	8
14.3.2. 类型声明	8
14.3.2.1. 示例策略	9
14.3.2.2. 引用策略	10
14.3.3. 最初允许的限制性访问	11
14.3.3.1. 示例策略	12
14.3.3.2. 引用策略	14
14.3.4. 允许域转换和指派角色	16
14.3.4.1. 示例策略	16
14.3.4.2. 引用策略	17
14.3.5. 整合进系统策略	17
14.3.5.1. 示例策略	17

14.3.5.2. 引用策略	18
14.3.6. 创建标记策略	19
14.3.6.1. 示例策略	19
14.3.6.2. 引用策略	20
14.3.7. 应用策略	20
14.4. 测试和分析策略	23
14.4.1. 测试策略模块	23
14.4.1.1. 评估审核消息允许额外的访问	25
14.4.1.2. 在示例策略中添加额外的访问权	26
14.4.1.3. 在引用策略中添加额外的访问权	26
14.4.1.4. 测试额外的访问权	26
14.4.2. 策略分析	27
14.5. 新兴的策略开发工具	27
14.6. 完整的 IRC 守护进程模块清单	28
14.7. 小结	36

第一部分：SELinux 摘要

第 1 章. 背景

安全增强的 Linux (SELinux) 是一项加强我们计算机网络和系统安全的令人兴奋的新技术。确切地说，它代表了操作系统安全研究近 40 年来达到的最高境界。首先，我们拥有了一个强大的、灵活的强制访问控制机制，被广泛合并到主流的操作系统发行版中。在这一章中，我们简单介绍一下操作系统安全研究的历史，以及 SELinux 给当今计算机安全挑战带来的希望。

1.1. 软件失效的必然性

这个标题源自 SELinux 主要创作者的一篇关于 SELinux 的论文[1]，写该论文时甚至 SELinux 项目都还没有开始。那篇论文的作者指出软件总是有缺陷的，太多的软件都假设应用程序无需在操作系统支持下就可以实施安全保护，因此他们做了以下注解：

[1]P. Loscocco, S. Smalley, P. Muckelbauer, R. Taylor, S. Turner, J. Farrell.
失效的必然性:现代计算机环境安全设想缺陷。21 届国家信息系统安全会议记录, pp. 303314, 1998 年十月, 在 <http://www.nsa.gov/selinux/papers/inevit-abs.cfm> 可看到全文。

操作系统安全的必要性对整个系统安全而言是不可争辩的，如果它失效了，将导致整个系统变得脆弱。

脱离操作系统支持的安全设计，就象“建造在沙子之上城堡”[2]一样没有安全基础。

[2]D. Baker, 建造在沙子之上的城堡。新安全范例工作台记录, pp. 148153, 1996 年。

自 1998 年那篇论文发表以来, 有缺陷的应用软件的问题已经变成每日新闻头条了。很少有一周不出现新病毒、计算机偷窃行为或不通告系统弱点。计算机时代生活的真实写照是应用软件总是有缺陷的或将会留下缺陷的。我们举双手赞成使软件变得更好、更可靠的努力, 但缺陷在未来会一直存在。总会有一些人会尝试利用这些缺陷, 我们的挑战是寻找到一种加强系统已知缺陷的方法, 如果不先打好地基(即操作系统), 我们就不能成功应对这个挑战。

因此我们找到了 SELinux 的目标: 明确地宣扬一种使操作系统更安全的格式程序。正如我们在本书中描述的, 操作系统的安全做得还不够好。我们作为一个计算机安全团体, 40 年前就知道了, 我们已经做了大量的研究, 但对于改善这个情况我们只取得了及其有限的成功。最后, SELinux 的出现, 我们才认为取得了长足的进步。SELinux 对 Linux 操作系统的确是一个安全增强。这个增强可以有效地缓解应用软件的缺陷问题, 包括那些还没有发现的缺陷。这个增强也实现了许多安全目标, 从数据保密性到应用程序完整性到提高的坚固性。

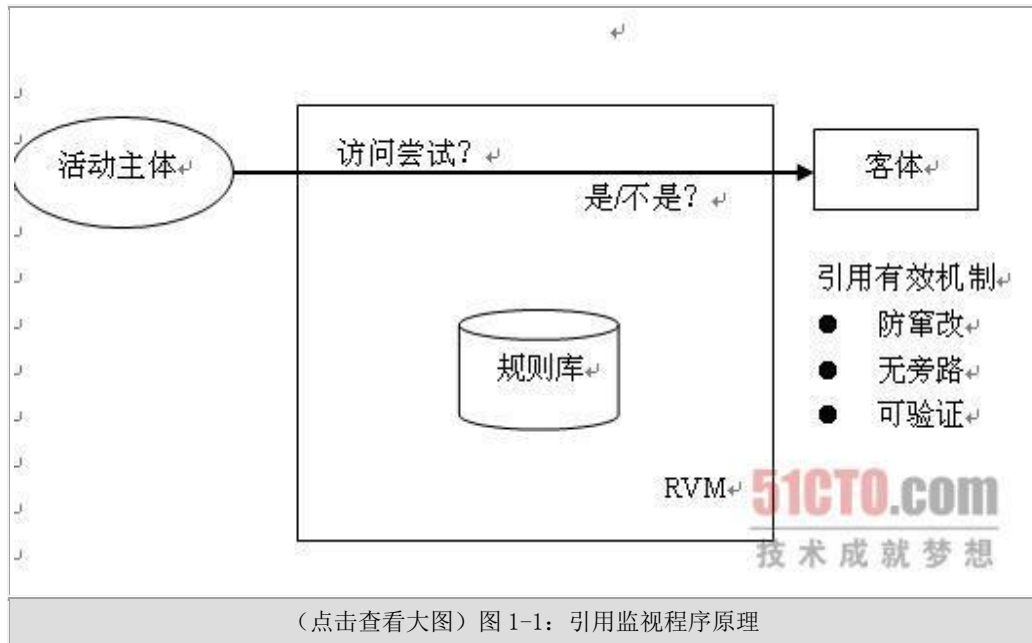
使用 SELinux, 将我们的“城堡”从目前的沙子位置超前移动了一大步。

1.2. 操作系统访问控制安全的进展

早期操作系统有一点或没有安全性, 用户可以访问所有文件或资源。幸运的是, 在访问控制机制开始出现之前这种情况并没有太长的时间。目前我们主要的访问控制类型叫做任意访问控制 (DAC), DAC 的主要特性是个别用户(通常是资源的“所有者”)可能有或没有访问资源的权限。正如你将会看到的, DAC 有一些内在的安全弱点。为了克服这些弱点, 计算机安全社区尝试开发有用的强制访问控制 (MAC) 机制, 设计 MAC 就是为了避免 DAC 的弱点, 提供需要的安全保障, 不幸的是, 创建一个有用的 MAC 安全机制是很困难的, 由于它涉及的范围非常大, 而且它很灵活, SELinux 带给 Linux 的主要价值是一个灵活的, 可配置的 MAC 机制。在本节剩余部分, 我们研究一下 DAC 和 MAC 机制的强度和弱点, 为理解 SELinux 提供的真实价值提供一个背景做参考。

1.2.1. 引用监视程序原理

要理解访问控制, 你必须对引用监视程序原理有一个了解。美国国防部门在 1970-1980 期间对操作系统安全做了一个研究。来自那次研究的一个关键报告叫做 Anderson 报告, 第一次提出在操作系统中使用访问控制的概念(参考图 1-1)。



在一个引用监视程序中，操作系统将无源的资源隔离到截然不同的客体如文件、运行中的程序活动条目，引用监视程序机制（叫做引用有效机制）将会通过应用一个嵌入在一套访问控制规则中的安全策略确认在活动主体和客体之间的访问。按照这种方法，程序访问系统资源（如文件）可以被限制到那些符合安全策略的访问，访问控制决定基于每个活动主体和客体的安全属性，安全属性指活动主体/客体与安全有关的特性。例如：在标准的 Linux 中，活动主体（即进程）有真实有效的用户标识，客体（如文件）有访问许可模式，由许可模式决定进程是否可以打开某个文件。

除实现安全策略外，引用监视程序原理最初的设计目标是：

防止篡改（不能恶意改变或修改）

无旁路（活动主体不能避免访问控制决定）

可验证（可以证明安全策略实现是正确的）

几乎所有的操作系统都实现了某种格式的引用监视，都包括活动主体、客体和安全策略规则。在标准的 Linux 中，活动主体通常是进程，客体通常是用于信息共享、存储和通讯的系统资源（如文件、目录、套接字、共享内存等）。在 Linux 中，与其他大多数流行的操作系统一样，安全策略规则通过引用监视程序（即内核）强制固定和硬编码，无论这些规则的安全属性用于验证什么（如访问模式），都是可以改变和分配的，标准 Linux 安全是 DAC 格式的安全。

1.2.2. 任意访问控制的问题

DAC 是一种允许经授权的用户（通过它们的程序如一个 shell）改变客体的访问控制属性的访问控制机制，由此明确指出其他用户是否有权访问这个客体。一个简单的 DAC 形式可

能要数文件密码了，访问该文件需要知道文件所有者创建的密码。大多数 DAC 机制都是基于用户标识的访问控制属性。几乎所有现代操作系统都有一种基于用户标识的 DAC 实现。在 Linux 中，非常流行和广为人知的是所有者-组-其它许可模式机制。同样，常见的访问控制列表机制也是通用的方法。

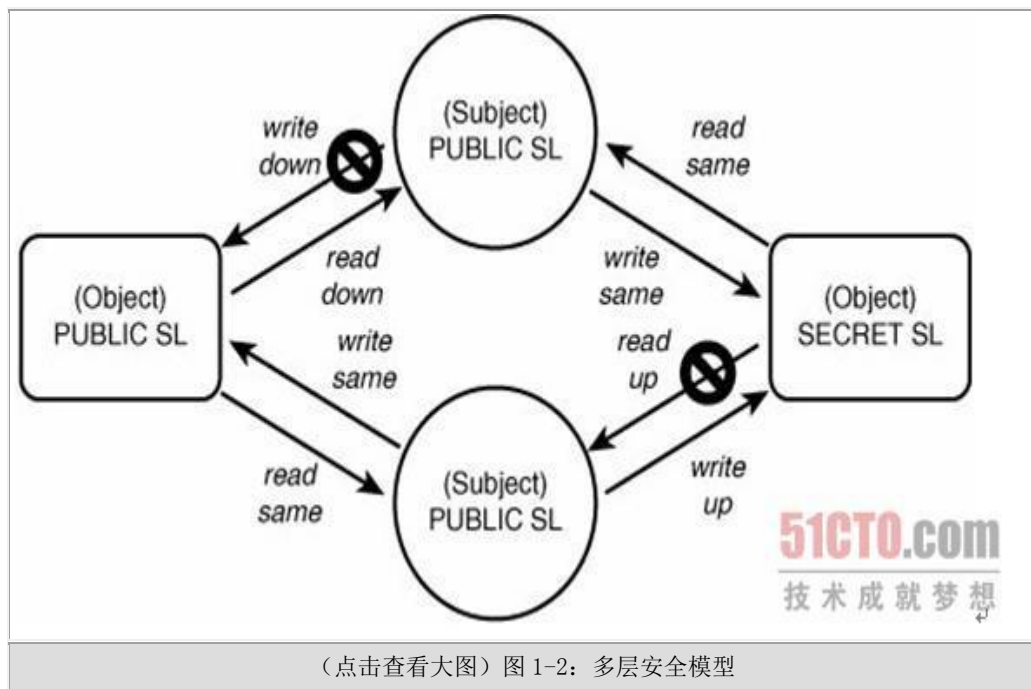
所有 DAC 机制都有一个共同的弱点，就是它们不能识别自然人与计算机程序之间最基本的区别。DAC 通常是尝试模仿所有权原理。例如：文件所有者拥有指定文件的访问权限，只授予可以信任的用户访问文件[4]。假设你可以信任某个用户（通常被认为是无效的事情），计算机工作方法不是完全模仿的真实世界，只是按照用户指定的指令在计算机上执行操作，因此，我们不能向用户提供授予和使用访问的能力，相反，我们提供软件程序实现这个功能，在 internet 时代变得特别明显，程序总是充满缺陷，或干脆就是恶意程序，这就是特洛伊木马问题，首次是在 1970 年被认定，到目前的流行病毒、蠕虫、间谍软件，都是它的变体。简单点说就是，如果一个用户被授权允许访问，意味着程序也被授权访问，如果程序被授权访问，那么恶意程序也将有同样的访问权。

[4] 所有者使用它们的辨别力判断是否授予访问权。

DAC 假设一开始所有的程序都是值得信赖且没有缺陷的，早期的计算机研究社团大都是住在高校中的一群人（教师、学生），因此目前发展的技术可能都是建立在那个假设的环境基础之上的，然而，实际上，我们知道在计算机科学历史上还没有出现过那种开始环境，人类的天性总是喜欢攻击有弱点的软件。

1. 2. 3. 强制访问控制的起源

从 1970 年到 1980 年，最主要的精力都使用在寻找解决恶意软件或有缺陷的软件的问题上，目标是实现 MAC，它的访问控制判断基础不是某个人或系统管理员的辨别力，我们想实现一种有组织的安全策略来控制对客体的访问，不受个别程序的行为影响。军方在这方面积累了大量的经验，主要是保护政府的机密数据。大多数常见的 MAC 机制实现了数据多层安全，图 1-2 显示了一个多层安全模型的组成结构。



多层安全（MLS）一般都是基于 Bell-LaPadula 模型[5]，在 MLS 模型中，所有主体和客体都标记有安全级别，在我们的例子中，我们有一个 PUBLIC 和 SECRET 安全级别，这两个级别代表了数据的敏感程度（SECRET 的敏感度比 PUBLIC 的敏感度要高），在 MLS 中，主体总是可以读写客体，除此之外，主体还可以读取底层客体（向下读取），写入高层客体（向上写入），然而，主体可能永远不会读取高层客体（无向上读取），也不写入底层客体（无向下写入），意思就是信息可以从底层流向高层，但不能反过来，因此要保护高层数据的机密性。

[5]<http://csrc.nist.gov/publications/history/bell76.pdf>

MLS 是对访问控制最根本的改变，不再是数据所有者任意地决定谁可以访问客体，此外，对大部分被认为不受信任的软件我们也有足够的安全措施，因为信息流规则阻止了不正当的数据访问，在 MLS 中，通过固定的规则决定数据如何共享，而不管用户的请求（更重要的是他们运行的程序），MLS 实现了大部分 MAC 机制，在目前流行的操作系统中普遍存在，MAC 机制与 MLS 类似，也是经过深思熟虑建立起来的，它实现了小部分固定的安全属性。

MLS 的主要弱点是它严格地，以不可改变的方式实现了单一安全目标（即保护政府文件敏感数据的机密性），不是所有的操作系统安全业务都与数据机密性有关，大多数都对保密政府文件（包括许多处理保密数据的政府系统）的强硬的简单的模型不肯顺从，为了在 MLS 中扩大这个目标，主体必须被授予安全策略特权（即违反无旁路的原则），并信任不会违反策略的目的，这个不屈性和有限的焦点使 MLS 和类似的 MAC 机制实现了充足的要求。

1. 2. 4. 更好的强制访问控制

SELinux 实现了一个灵活的 MAC 机制，叫做类型强制（TE），正如你将会看到的，类型强制提供强壮的强制安全能够适应大量的安全目标，类型强制提供一种思想，将访问控制到对某一个程序级别去，在某种程度上，它允许组织定义适合他们系统的安全策略。在类型强制下，所有主体和客体都有一个类型标识符与它们关联，要访问某个客体，主体的类型必须为客体的类型进行授权，而不管主体的用户标识符。

使 SELinux 比 MLS 解决方案更高级的是规则决定了基于类型的访问控制，这不是预先定义好的，也不是硬编码在内核中的。默认情况下，SELinux 不允许任何访问，组织可以开发任意数量的规则指定允许什么，这样使 SELinux 可以适应非常多样的安全策略。

定义允许访问一个系统收集的规则叫做 SELinux 策略，物理上，SELinux 策略是一个特殊的文件，它包括 SELinux 内核将会实施的所有规则，策略文件是从一套源文件编译而来的，正如你马上会看到的，系统与系统之间的 SELinux 策略是可以不同的，在启动过程中，策略被载入内核，然后内核就会使用它作为访问控制决定的根据。

注意：在计算机安全领域，术语策略是一个非常大的范围。在这一章中，定义一个组织的安全目标和目的时我们使用这个术语。然而，在谈到载入内核的 SELinux 规则集（和包括它们的文件）时也使用策略这个术语。我们尽力避免使用这个词时产生冲突，但不能完全避免这个问题，如果遇到有歧义的地方，我们通常会明确地编写 SELinux 策略来避免冲突。

SELinux 为 Linux 带来了灵活的类型强制和一套基于角色的访问控制，以及非强制的 MLS。这个灵活性和适应性强的 MAC 安全已经内置到主流的 Linux 操作系统中，说明 SELinux 是如此有希望的提高安全性的技术。

1.2.5. SELinux 的发展

SELinux 起源于自 1980 开始的微内核和操作系统安全的研究，这两条研究线路最后形成了一个叫做的分布式信任计算机（Distribute Trusted Mach (DTMach)）的项目，它融合了早期研究项目的成果（LOCK【锁】，它包含一组安全内核类型强制；Trusted Mach【信任计算机】，它将多层安全控制合并到计算机微内核中）。美国国家安全局的研究组织参加了 DTMach 项目，付出了巨大努力，并且继续参与了大量的后续安全微内核项目。最终，这些工作和努力导致了一个新的项目产生，那就是 Flask，它支持更丰富的动态类型的强制机制。

由于不同平台对这项技术没有广泛使用，NAS 认为需要在大量的社团中展示这个技术，以说明它的持久生命力，并收集广泛的使用支持意见。在 1999 年夏天，NSA 开始在 Linux 内核中实现 Flask 安全架构，在 2000 年十二月，NSA 发布了这项研究的第一个公共版本，叫做安全增强的 Linux。因为是在主流的操作系统中实现的，所以 SELinux 开始受到 Linux 社区的注意，SELinux 最迟是在 2.2.x 内核中以一套内核补丁的形式发布的。

随着 2001 年 Linux 内核高级会议在加拿大渥太华顺利召开, Linux 安全模型 (LSM[7]) 项目开始为 Linux 内核创建灵活的框架, 允许不同的安全扩展添加到 Linux 中。NSA 和 SELinux 社区是 SELinux 的主要贡献者, SELinux 帮助 LSM 实现了大量的需求, 为了与 LSM 一起工作, NSA 开始修改 SELinux 使用 LSM 框架。2002 年八月 LSM 核心特性被集成到 Linux 内核主线, 同时被合并到 Linux 2.6 内核。2003 年八月, NSA 在开源社区的帮助下, 完成了 SELinux 到 LSM 框架的迁移, 至此, SELinux 进入 Linux 2.6 内核主线, SELinux 已经成为一种全功能的 LSM 模块, 包括在核心 Linux 代码集中。

[7]<http://lsm.immunix.org/>

数个 Linux 发行版开始在 2.6 内核中不同程度使用 SELinux 特性, 但最主要是靠 Red Hat 发起的 Fedora Core 项目才使 SELinux 具备企业级应用能力, NSA 和 Red Hat 开始联合集成 SELinux, 将其作为 Fedora Core Linux 发行版的一部分。在 Red Hat 参与之前, SELinux 总是作为一个附加的软件包, 需要专家级任务才能进行集成。Red Hat 开始采取行动让 SELinux 成为主流发行版的一部分, 完成了用户空间工具和服务的修改, 默认开启增强的安全保护。从 Fedora Core 2 开始, SELinux 和它的支持基础架构以及工具得到改进。在 2005 年早些时候, Red Hat 发布了它的 Enterprise Linux 4 (REL 4), 在这个版本中, SELinux 默认就是完全开启的, SELinux 和强制访问控制已经进入了主流操作系统和市场。

SELinux 仍然是一个相对较新和复杂的技术, 重要的研究和开发在继续不断地改进它的功能。

1.3. 小结

应用软件是有缺陷的, 并且在可预见的未来仍然是有缺陷的, 尽管如此, 我们必须找到创建安全系统的方法, 不管这些缺陷是否存在。没有更好的底层操作系统安全的支持真正安全是实现不了的, SELinux 的目标是为主流操作系统提供改良的安全性。

引用监视原理在操作系统中通常用来描述访问控制, 在引用监视程序中, 资源被封装为明确的客体, 在主体 (即进程) 和客体之间的访问通过引用确认机制按照系统安全策略进行判决。

操作系统有两类访问控制: 任意访问控制 (DAC) 和强制访问控制 (MAC)。标准 Linux 安全是一种 DAC, SELinux 为 Linux 增加了一个灵活的和可配置的 MAC。

DAC 最根本的弱点是主体容易受到多种多样的恶意软件的攻击, MAC 就是避免这些攻击的出路, 大多数 MAC 特性组成了多层安全模型。

SELinux 实现了一个更灵活的 MAC 形式, 叫做类型强制和一个非强制的多层安全形式。

练习

1、建立一个 SELinux 系统，使用附录 A“获得 SELinux 示例策略”中的指令安装 strict 示例策略。

第 2 章：概念

SELinux 访问控制机制和策略语言的详细内容非常广泛，将在后面的章节中进行描述。但 SELinux 的基本概念和目标是相当简单的，在这一章中，我们将学习 SELinux 的安全概念，并在这些概念后进行练习。要有效使用和应用 SELinux 访问控制，理解其概念是非常必要的，这一章主要集中介绍 SELinux 的访问控制特性和类型强制（TE），同时我们也简要地介绍了一下非强制的多层安全机制。

2.1. 类型强制的安全上下文

所有操作系统访问控制都是以关联的客体和主体的某种类型的访问控制属性为基础的。在 SELinux 中，访问控制属性叫做安全上下文。所有客体（文件、进程间通讯通道、套接字、网络主机等）和主体（进程）都有与其关联的安全上下文，一个安全上下文由三部分组成：用户、角色和类型标识符。常常用下面的格式指定或显示安全上下文：

用户:角色:类型

这个字符串标识符中的每个元素使用 SELinux 策略语言进行定义，后面我们会对策略语言进行详细的描述，现在只需要理解一个有效的安全上下文必须要有一个有效的用户、角色和类型标识符，通过策略编写器定义标识符，每一个标识符的命名空间正交的，因此，用户、角色和类型的标识符可能相同，但不合理，也不建议这么定义。

安全上下文示例

SELinux 对系统中的许多命令做了修改，通过添加一个 -Z 选项显示客体和主体的安全上下文，例如：ls -Z 显示文件系统客体的安全上下文，ps -Z 显示进程的安全上下文，另一个有用的命令是 id，它显示了你的 shell 的安全上下文（即你当前的用户、角色和类型），下面的例子显示了一个运行 SELinux 的 shell 安全上下文：

```
$id -Z
joe:user_r:user_t
```

你可以使用本章中我们列出的这些命令研究你自己的 SELinux 系统。

2.1.1. 对比 SELinux 和标准 Linux

至此，比较标准 Linux 和 SELinux 之间的访问控制属性显得非常有用，为了使表述简化，我们只固定讨论文件系统客体如文件和目录。在标准 Linux 中，主体的访问控制属性是与进程通过在内核中的进程结构关联的真实有效的用户和组 ID，这些属性通过内核利用大量工具进行保护，包括登陆进程和 setuid 程序，对于客体（如文件），文件的 inode 包括一套

访问模式位、文件用户和组 ID。以前的访问控制基于读/写/执行这三个控制位，文件所有者、文件所有者所属组、其他人各一套。

在 SELinux 中，访问控制属性总是安全上下文三人组形式，所有客体和主体都有一个关联的安全上下文，标准 Linux 使用进程用户/组 ID，文件的访问模式，文件用户/组 ID 要么可以访问要么被拒绝，SELinux 使用进程和客体的安全上下文，需要特别指出的是，因为 SELinux 的主要访问控制特性是类型强制，安全上下文中的类型标识符决定了访问权。

注意:SELinux 在标准 Linux 基础上增加了类型强制,这就意味着标准 Linux 和 SELinux 访问控制都必须满足先要能访问一个客体,例如:如果我们对某个文件有 SELinux 写入权限,但我们没有该文件的 w 许可,那么我们也不能写该文件。

表 2-1 总结了标准 Linux 和 SELinux 之间访问控制属性的对比

表 2-1. 标准 Linux 和 SELinux 访问控制比较

	标准 Linux	SELinux
进程安全属性	真实有效的用户和组 ID	安全上下文
客体安全属性	访问模式和文件用户和组 ID	安全上下文
访问控制基础	进程用户/组 ID 和文件基于文件的用户/组 ID 的访问模式	在进程类型和文件类型之间允许的许可

2.1.2. 安全上下文

安全上下文是一个简单的、一致的访问控制属性，在 SELinux 中，类型标识符安全上下文的主要组成部分，由于历史原因，一个进程的类型通常被称为一个域（domain），”域”和”域类型”意思都是一个，我们不必苛刻地去区分或避免使用术语域，通常，我们认为域、域类型、主体类型和进程类型都是同义的。

安全上下文中的用户和角色标识符除了对强制有一点约束之外对类型强制访问控制策略没什么影响,我们将在第 7 章”约束”中讨论,对于进程,用户和角色标识符显得更有意义,因为它们用于控制类型和用户标识符的联合体,这样就会 Linux 用户账号关联起来;然而,对于客体,用户和角色标识符几乎很少使用,为了规范管理,客体的角色常常是 object_r,客体的用户常常是创建客体的进程的用户标识符,它们在访问控制上没什么作用。

最后，一定要清楚标准 Linux 安全中的用户 ID 和安全上下文中的用户标识符之间的区别，就技术而论，它们是正交标识符，分别用于标准的和安全增强的访问控制机制，这两者

之间的任一相互关联都是通过登陆进程按照规范严格规定的,而不是通过 SELinux 策略直接强制实施的。

2.2. 类型强制访问控制

在 SELinux 中,所有访问都必须明确授权,SELinux 默认不允许任何访问,不管 Linux 用户/组 ID 是什么。这就意味着在 SELinux 中,没有默认的超级用户了,与标准 Linux 中的 root 不一样,通过指定主体类型(即域)和客体类型使用 allow 规则授予访问权限,allow 规则由四部分组成:

源类型 (Source type(s)) 通常是尝试访问的进程的域类型

目标类型 (Target type(s)) 被进程访问的客体的类型

客体类别 (Object class(es)) 指定允许访问的客体的类型

许可 (Permission(s)) 象征目标类型允许源类型访问客体类型的访问种类

举一个例子进行说明,如:

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

这个例子显示了 TE allow 规则的基础语法,这个规则包含了两个类型标识符:源类型(或主体类型或域) user_t,目标类型(或客体类型) bin_t。标识符 file 是定义在策略中的客体类别名称(在这里,表示一个普通的文件),大括号中包括的许可是文件客体类别有效许可的一个子集,这个规则解释如下:

拥有域类型 user_t 的进程可以读/执行或获取具有 bin_t 类型的文件客体的属性。

正如我稍后会讨论的,SELinux 中的许可相对标准 Linux 而言要更细一些,在标准 Linux 中只有三种 (rwx)。在这里,read 和 execute 是十分常见的,getattr 明显要少得多,实质上,文件的 getattr 许可只不过允许调用者查看(不修改)如日期、时间等属性,以及任意访问控制 (DAC) 访问模式。在一个标准 Linux 系统中,调用者只有搜索文件所在目录的权限,但也可能看到关于文件的信息,因为他可以通过搜索目录来看到文件的某些信息,即使他没有该文件的访问权限。

假设 user_t 是一个普通的、无特权的用户进程如一个登陆 shell 进程域类型,bin_t 是与用户运行的执行文件(如/bin/bash)关联的类型,策略中的规则可能允许用户执行 shell 程序如 bash shell。

注意:在类型标识符名字中的_t 没有意义,只不过是在大多数 SELinux 策略中惯用的约定,策略编写器可以通过策略语言语法允许的任何合适的规范定义类型标识符。

在这一章中，我们使用符号圆圈代表进程，方框代表客体，箭头代表允许的访问来描绘允许的访问，例如：如 2-1 描绘了前面的 allow 规则允许的访问。

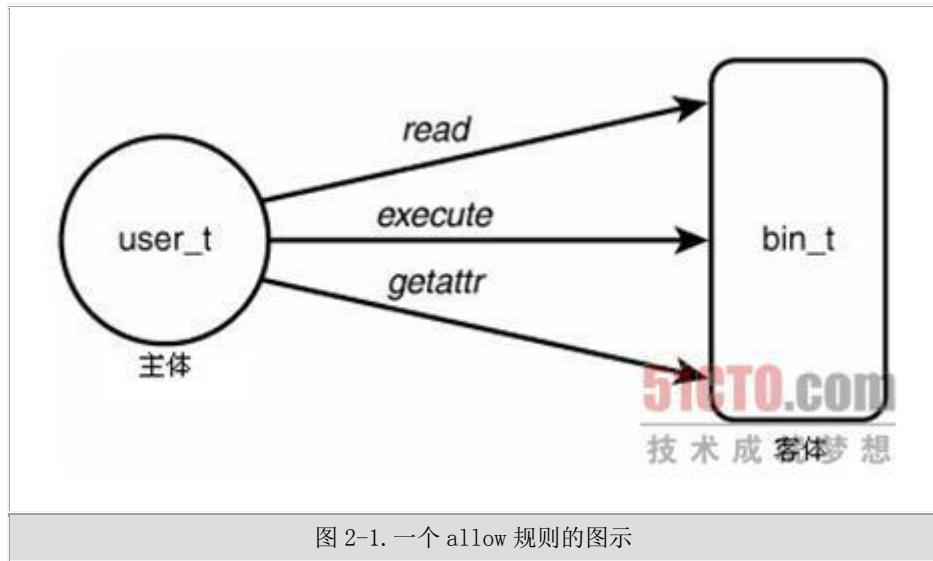


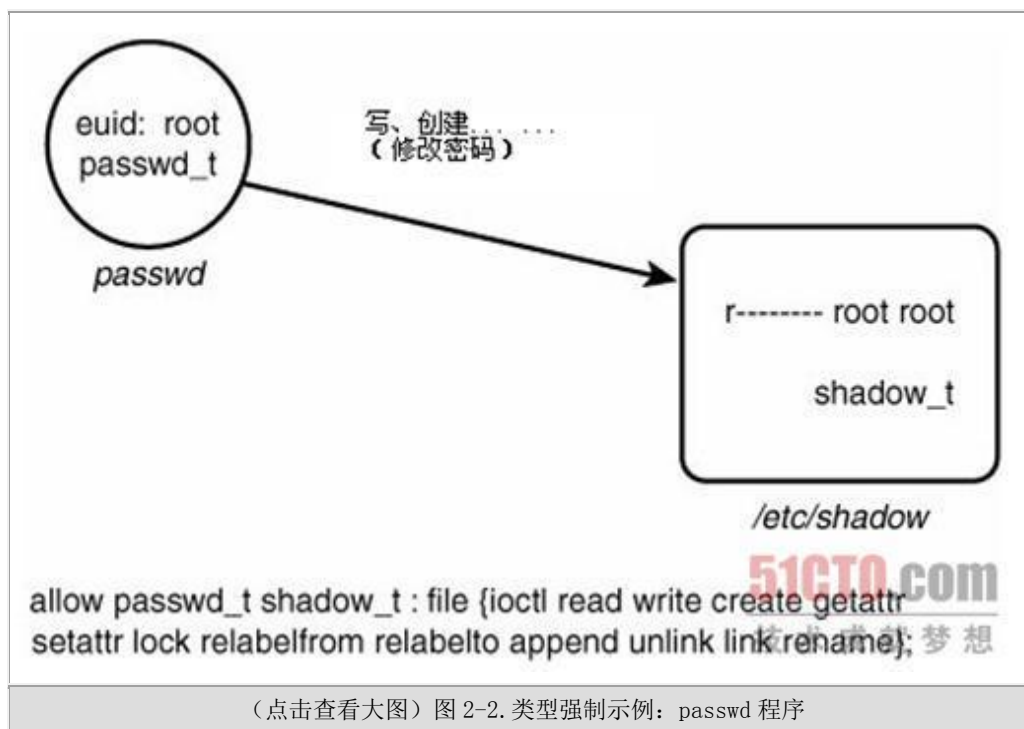
图 2-1. 一个 allow 规则的图示

2.2.1. 类型强制示例

SELinux allow 规则如之前的例子在 SELinux 中实际上都是授予访问权的，真正的挑战是如何保证数以万计的访问正确授权，只授予必须的权限，实现尽可能的安全。

为了进一步研究类型强制，我们以管理密码的程序（即 passwd）为例进行说明，在 Linux 中，passwd 程序是可信任的，修改存储经过加密的密码的影子密码文件（/etc/shadow），passwd 程序执行它自己内部的安全策略，允许普通用户修改属于他们自己的密码，同时允许 root 修改所有密码。为了执行这个受信任的作业，passwd 程序需要有移动和重新创建 shadow 文件的能力，在标准 Linux 中，它有这个特权，因为 passwd 程序可执行文件在执行时被加上了 setuid 位，它作为 root 用户（它能访问所有文件）允许，然而，许多程序都可以作为 root 允许（实际上，所有程序都有可能作为 root 允许）。这就意味着任何程序（当以 root 身份运行时）都有可能能够修改 shadow 文件。类型强制使我们能做的事情是确保只有 passwd 程序（或类似的受信任的程序）可以访问 shadow 文件，不管运行程序的用户是谁。

图 2-2 描绘了在 SELinux 系统中 passwd 程序如何使用类型强制进行工作的



在这个例子中，我们定义了两个类型，passwd_t 类型代表密码程序使用的域类型，shadow_t 类型代表 shadow 密码文件的类型。如果我们在磁盘上检查这个文件，我们将会看到：

```
# ls -Z /etc/shadow
-r----- root root system_u:object_r:shadow_t shadow
```

同样，在这个策略下检查运行密码和程序的进程将会看到：

```
# ps -aZ
joe:user_r:passwd_t 16532 pts/0 00:00:00 passwd
```

现在，你可以忽略安全上下文的用户和角色元素，只需注意类型就行了。

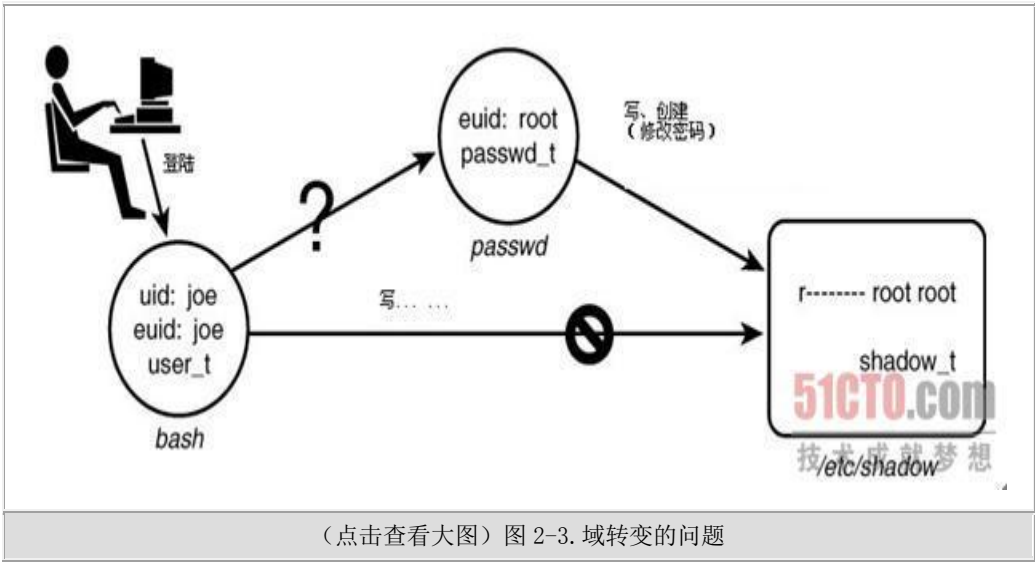
检查图 2-2 中的 allow 规则，这个规则的目的是授予 passwd 进程的域类型(passwd_t) 访问 shadow 的文件类型(shadow_t)，需要允许进程移动并创建一个新的 shadow 密码文件。重新回到图 2-2 中，我们看到描绘运行密码程序 (passwd) 的进程能够成功管理 shadow 密码文件，因为它有一个有效的 root 用户 ID (标准 Linux 访问控制)，同时因为 TE allow 规则允许它访问 shadow 密码文件的类型 (SELinux 访问控制)，两者都是必须的，但都不充分。

2.2.2. 域转变的问题

如果我们所做的所有事情是为进程提供对客体如文件的许可访问，编写一个 TE 策略应该是直截了当的事情。然而，我们不得不找出用正确的域类型安全正确地运行一个程序的方法。例如：我们不想程序由于某种原因在一个进程中用 `passwd_t` 域类型不受信任地访问 `shadow` 文件，这可能是灾难性的，这个问题带来了域转变的问题。

为了说清楚这个问题，在图 2-3 中，我们扩展了前面的密码程序示例，在一个普通的系统中，用户（如 Joe）登陆进系统，通过登陆进程会创建一个 `shell` 进程（如运行 `bash`）。在标准 Linux 安全中，真实有效的用户 ID（即 Joe）是相同的[1]，在我们的 SELinux 策略示例中，我们看到进程类型是 `user_t`，它表示一个普通的、不受信任的用户进程域类型。当 Joe 的 `shell` 程序运行其他程序时，新进程的类型将会保留 `user_t` 域类型，除非做了其他操作，那么 Joe 如何修改密码呢？

[1] 正确地说，Joe 不是用户 ID，Joe 只不过是在密码文件（`/etc/passwd`）中表示用户的字符串而已，为了使解释更浅显易懂，我跳过了中间的步骤，在我们的例子中只使用字符串标识符。



我们不应该让 Joe 的不受信任的域类型 `user_t` 能够直接读写 `shadow` 密码文件，因为这样将会允许任何程序（包括 Joe 的 `shell`）看到并修改这个关键文件的内容。正如前面讨论的，我们只想密码程序有这个访问权，然后只以 `passwd_t` 域类型运行它。那么，问题是如何提供一个安全可靠的并且比较隐蔽的方法将 Joe 的以 `user_t` 类型运行的 `shell` 转变为一个以 `passwd_t` 类型运行密码程序的进程。

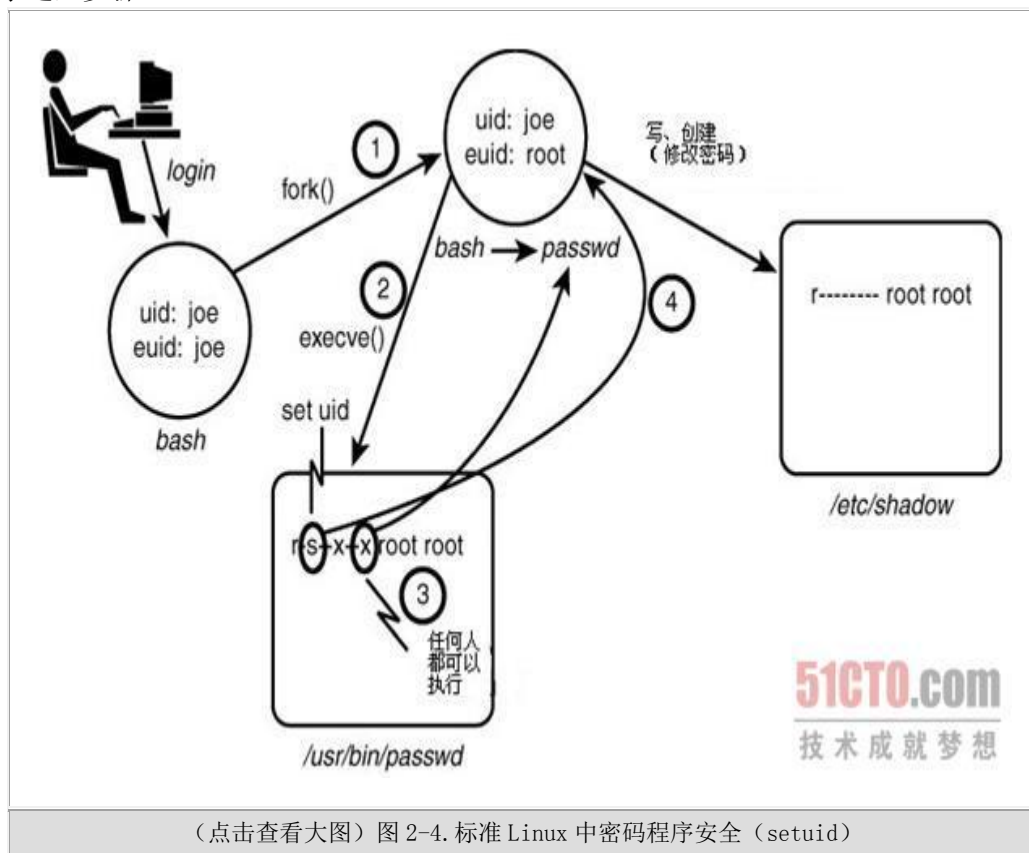
2.2.3. 标准 Linux 安全中的 `setuid` 程序

在我们讨论如何处理域转变的问题之前，首先看一下类似的问题在标准 Linux 中是如何处理的，即 Joe 想安全地修改现有的密码问题，Linux 解决这个问题的方法是通过给 `passwd`

赋一个 setuid 值，使其执行时具有 root 权限，如果你在一个普通 Linux 系统上列出密码文件，你看到的会是：

```
# ls -l /usr/bin/passwd
-r-sxx 1 root root 19336 Sep  7 04:11 /usr/bin/passwd
```

这里注意两件事，第一个是在所有者权限的 x 位置被设置为 s 了，这就是所谓的 setuid 位，意思是任何执行这个文件的进程，它的有效 UID（即用户 ID）将会被改为文件所有者。这里，root 是文件所有者，因此当执行密码程序时实际上将会以 root 用户的 ID 运行，图 2-4 显示了这些步骤。



当 Joe 运行密码程序时真正会发生的是他的 shell 将会产生一个 fork() 系统调用创建一个它自身的副本，这个复制进程仍然有真实有效的用户 ID (joe)，并且仍然运行有 shell 程序 (bash)。然而，在调用完 fork 指令后，新的进程将会产生一个 execve() 系统调用来执行密码程序。标准 Linux 安全需要调用的用户 ID (仍然是 joe) 有 x 权限，这里确实就是这样，因为所有人都有 x 权限。成功执行 execve() 调用将会发生两件非常关键的事情，第一件是运行在新进程中的 shell 程序将会被 passwd 程序替换，第二，因为 setuid 位是为所有者设的，真正的用户 ID 将从进程的原始 ID 改为文件所有者的 ID (这里是 root)，因为 root 可以访问所有文件，那么密码程序也就可以访问 shadow 密码文件了，也就可以处理 Joe 修改密码的需求了。

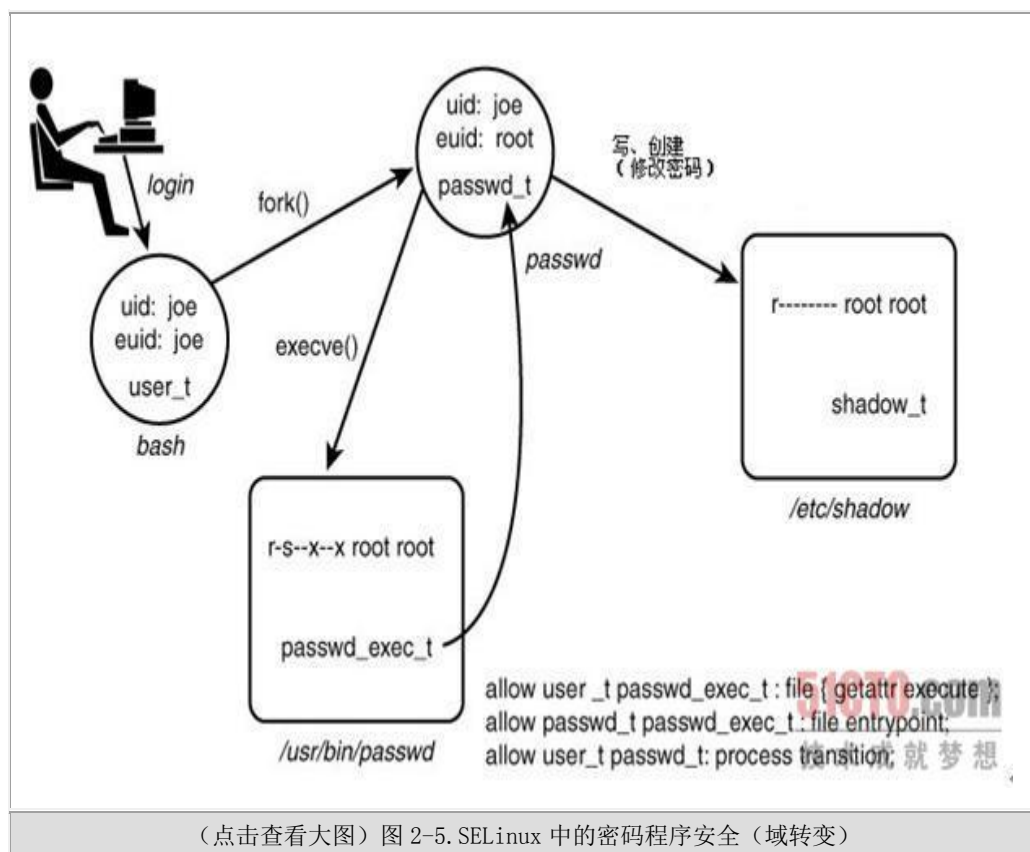
setuid 位的使用在类 Unix 操作系统中非常普遍，这是它们的一个简单但强大的特性，然而，它也成为标准 Linux 安全的主要弱点，密码程序需要以 root 身份运行访问 shadow 文件，然而，当以 root 身份运行时，密码程序可以访问所有的系统资源，这就违背了中心安全工程的最小权限原则，结果，我们必须信任密码程序，对于信任的安全应用程序，密码程序需要一个扩展的代码审核来确保它不会滥用它的特权，而且，当遇到无法预见的错误时，会影响到密码程序，这样就可能会将缺陷引入，即使密码程序相当简单且是高度受信任的，想象一下其他程序（包括登陆 shell）可能以 root 身份运行将会是多么可怕的。

我们真正要做的事情应该是确保为密码程序设置最小权限，我们认为密码程序应该只能访问 shadow 文件和其他与密码有关的文件，再加上那些必须的最小系统资源，同时我们还应该确保除了密码程序能访问 shadow 文件外，其他程序都不能访问这个文件，这样说来，我们就只需要关心密码程序它自身的角色就行了，不用管用户账号了。

这就是马上会描述的类型强制。

2.2.4. 域转变

在前面的图 2-2 中，allow 规则确保 passwd 进程域类型（passwd_t）可以访问 shadow 密码文件，然而，我们仍然有前面已经提到的域转变问题，提供一个安全的域转变与 setuid 程序的原理非常类似，为了解释清楚一点，下面我们以在 setuid 的例子中增加类型强制为例进行说明（查看图 2-5）。



现在，我们的例子变得更复杂了，让我们一起来详细看看这张图。首先，注意我们在前面的基础上增加了三个类型，即 Joe 的 shell 域 (user_t)，密码程序的域类型 (passwd_t) 和 shadow 密码文件的类型 (shadow_t)。此外，我们为 passwd 可执行文件增加了文件类型 (passwd_exec_t)。例如：列出密码程序的安全上下文时，将会看到：

```
# ls -Z /usr/bin/passwd
-r-sxx root root system_u:object_r:passwd_exec_t /usr/bin/passwd
```

现在我们有足够的信息来创建 TE 策略规则允许密码程序（大概只有密码程序）以 passwd_t 域类型运行，让我们来看一看图 2-5 中的规则，第一条规则是：

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

这条规则所做的事情是允许 Joe 的 shell (user_t) 在 passwd 可执行文件 (passwd_exec_t) 上启动 `execve()` 系统调用，SELinux `execute` 文件权限实际上与标准 Linux 中的 `x` 访问权限是一样的，尝试执行之前 shell 先“统计”文件，因此需要 `getattr` 权限，回想一下我们在前面描述的 shell 程序实际上是如何工作的，首先它派生出自身的一个副本，包括相同的安全属性，这个副本仍然保留了 Joe 的 shell 原始域类型 (user_t)，因此，必须给原始域（即 shell 的域类型）赋予执行权限，那就是为什么 user_t 是这个规则的源类型的原因了。

下面让我们来看看图 2-5 中的第二个规则：

```
allow passwd_t passwd_exec_t : file entrypoint;
```

这条规则提供了对 passwd_t 域的入口访问权，`entrypoint` 许可在 SELinux 中是一个相当有用的许可权限，这个权限所做的事情是定义哪个可执行文件（程序）可以“进入”某个特定的域，对于域转变，新的或将要进入的域（这里是 passwd_t）必须具有访问可执行文件的 `entrypoint` 许可权，以转变到新的域类型，在这个例子中，假设只有 passwd 可执行文件被标识为 passwd_exec_t，并且只有 passwd_t 类型有 `entrypoint` 权限访问 passwd_exec_t，这样我们就具备了只有密码程序才能运行在 passwd_t 域类型中的条件，这是一个强大的安全控制。

警告： `entrypoint` 许可权限的概念是相当重要的，如果你没有完全理解前面的示例，在开始行动前，请重新阅读一次。

让我们在看一看最后的规则：

```
allow user_t passwd_t : process transition;
```

这是我们第一次看到的 allow 规则没有提供对文件客体的访问，在这个例子中，客体类别是 process，意味着客体类别代表进程，回想一下前面的内容，所有的系统资源都被封装为客体类别，这个概念也包括进程，在这最后一个规则中，许可是 TTransition 访问，在允许修改进程的安全上下文的类型时，需要这个许可，原始的类型（user_t）到新的类型（passwd_t）进行域转变必须有 TTransition 许可才允许进行。

这三条规则一起提供了域转变发生时必须的访问权，对于一个成功的域转变，这三条规则都是必须的，任何单独一个都是不够充分的，因此，域转变只有在同时满足下面的三个条件时才允许进行：

- 1、进程的新域类型对可执行文件类型有 enTPoint 访问权
- 2、进程的当前（或旧的）域类型对入口文件类型有 execute 访问权
- 3、进程当前的域类型对新的域类型有 transition 访问权

当这三个许可在一个 TE 策略中都被通过后，才可能发生域转变，而且，在可执行文件上使用 enTPoint 许可，我们有能力严格控制哪个程序可以用给定的域类型运行，execve() 系统调用是修改域类型的唯一方法[2]，使得策略编写器可以很好地控制任何一个程序的访问特权，而不管调用程序的用户。

[2] 更精确地说，为 SELinux 最近对进程的修改提供了一个方法，赋予必须的特权，修改它的安全上下文，而不使用 execve() 调用，通常，这个机制将会在第 5 章“类型强制”中描述，它没有一个强健的理由，我们不应该使用它，因为它大大地削弱了类型强制的强健性。

现在的问题是 Joe 如何指出他想要的域转变，上面的规则只允许域转变，他们不需要它，程序员或用户有多种方法明确请求一个域转变（如果允许），但通常情况下，我们不想让用户明确地发出这些请求，Joe 想做的是允许密码程序，它希望系统确保他能完成这个任务，我们需要一个方法让系统默认启动域转变。

2.2.5. 默认域转变：type_transition 指令

为了支持默认的域转变，我们需要引入一个新的规则，类型转变规则(type_transition)，这个规则为 SELinux 策略提供一个方法指定默认应该尝试的转变，在没有明确指出需要的转变时，让我们在 allow 规则后面添加一个 type_transition 规则：

```
type_transition user_t passwd_exec_t : process
passwd_t;
```

这个规则的语法与 allow 规则有点不一样了，但仍然有源和目标类型（分别是 user_t 和

passwd_exec_t)，也有客体类别（process），然而，许可相反了，我们有了第三个类型，默认类型（passwd_t）。

type_transition 规则用于多中不同的与修改默认类型相关的目的，现在，我们关心将 process 作为它的客体类别的 type_transition 规则，这个规则引发默认的域转变尝试，type_transition 规则默认代表一个 execve() 系统调用，如果调用的进程域类型是 user_t，并且可执行文件的类型是 passwd_exec_t（如图 2-5 中的示例），将会尝试到一个新域类型（passwd_t）的域转变。

type_transition 规则允许策略编写器引发默认的域转变，而不需要明确的用户输入，这使得类型强制减少了用户的强制闯入，在我们的例子中，Joe 不想知道任何关于访问控制或类型的事情，他只想修改它的密码，系统和策略设计人员可以使用 type_transition 规则使这些转变对于用户而言是透明的。

注意：记住 **type_transition** 规则引发一个默认的域转变，但它不允许它执行，你还是必须为域转变成功发生提供三个需要的访问类型，无论是默认启动还是通过用户的明确请求启动。

2.3. 角色

SELinux 也提供了一种基于角色的访问控制（RBAC），SELinux 的 RBAC 特性是依靠类型强制建立的，SELinux 中的访问控制主要是通过类型实现的，角色基于进程安全上下文中的角色标识符限制进程可以转变的类型，如此，策略编写器可以创建一个角色，允许它转变为一套域类型（假设类型强制规则允许转变），从而定义角色的限制，以我们在图 2-5 中所举的密码程序为例，按照类型强制规则，密码程序可以通过 user_t 域类型执行，然后转到新的 passwd_t 域，Joe 的角色必须允许与新的域类型建立关联以便进行域转变，为了同图进行说明，我们在图 2-6 中扩展了密码程序示例。

类型强制无疑是 SELinux 引入的最重要的强制访问控制 (MAC) 机制, 然而, 在某些情况下, 主要是保密控制应用程序的一个子集, 传统的多层安全 (MLS) MAC 与类型强制一起使用显得更有价值, 在这些情况下, SELinux 总是包括某种格式的 MLS 功能, MLS 特性是可选的, 在 SELinux 的两个 MAC 机制中, 它通常不是最重要的那个, 对大多数安全应用程序而言, 包括许多非保密数据应用程序, 类型强制是最适合的安全增强的机制, 尽管如此, MLS 对部分应用程序还是增强了安全性。

MLS 的基本概念在第 1 章“背景”中已经介绍过了, 真实实施 MLS 更佳复杂, MLS 系统使用的安全级别是层次敏感的, 是一个 (包括空集) 非层次范畴的集合。这些敏感度和范畴用于映射真实的机密信息或用户许可, 在大多数 SELinux 策略中, 敏感度 (s0, s1, ...) 和范畴 (c0, c1, ...) 使用通配名, 将它留给用户空间程序和程序库, 以指定有意义的用户名。(例如: s0 可能与 UNCLASSIFIED 关联, s1 可能与 SECRET 关联)

为了支持 MLS, 安全上下文被扩展了, 包括了安全级别, 如:

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

注意 MLS 安全上下文至少必须有一个安全级别 (它由单个敏感度和 0 个或多个范畴组成), 但可以包括两个安全级别, 这两个安全级别分别被叫做低 (或进程趋势) 和高 (或进程间隙), 如果高安全级别丢失, 它会被认为与低安全级别的值是相同的 (最常见的情况), 实际上, 对于客体和进程而言, 低和高安全级别通常都是相同的, 通常用于进程的级别范围被认为是受信任的主体 (即进程信任降级信息) 或多层客体, 如一个目录, 它又包括了不同安全级别的客体。为了使描述简单, 假设所有的进程和客体都只有一个安全级别。

访问客体的 MLS 规则与第 1 章中描述的非常相似, 除了安全级别是无层次的之外, 但又提供了一个控制关系进行管理, 不像等式那样, 级别要么比另一个要高, 或相等, 要么就比它低, 在控制关系中, 有一个叫做无可比的第四种状态 (也叫做非可比的, 查看下面列表中的 incomp), 安全级别通过控制关系而不是范畴关联, 它没有层次关系, 有四个控制操作符可以将两个 MLS 安全级别关联起来, 如:

dom: 如果 SL1 的敏感度大于或等于 SL2 的敏感度, 则 SL1 dom (优于) SL2, SL1 的范畴就是 SL2 范畴的一个超集。

domby: 如果 SL1 的敏感度小于或等于 SL2 的敏感度, 则 SL1 domby SL2, SL1 的范畴是 SL2 的子集。

eq: 如果 SL1 的敏感度等于 SL2 的敏感度, 则 SL1 eq SL2, SL1 和 SL2 的范畴也是相同的。

incomp: 如果 SL1 的范畴和 SL2 的范畴不能进行对比(即既不是子集也不是超集或其他), 那么 SL1 incomp (不可对比) SL2。

提供了域关系, 在 SELinux 中实现了多种 Bell-La Padula 模型, 如果进程当前的安全级别优于客体的安全级别, 进程则可以“读取”客体。如果低于客体的安全级别, 则可以“写入”客体, 因此, 同时读写客体只要这两个安全级别相等就可以了。

SELinux 中 MLS 的约束是附加在 TE 规则中的, 如果启用了 MLS, 要授予访问权两个检查都必须通过, 第 8 章“多层安全”将会讨论 SELinux 可选的 MLS 特性。

2.5. 精通 SELinux 特性

此时, 花点时间摆弄一下 SELinux 系统是值得的, 在我们的例子中, 我们使用带 strict (严格)策略的 Fedora Core 4(FC 4)发行版, 这些例子大部分都能工作在 Red Hat Enterprise Linux 4 (RHEL 4) 或 Fedora Core 5 (FC 5), 也可能能够工作在其他发行版上, 即使它们之间有许多不同之处。附录 A“获得 SELinux 示例策略”描述了如何获取策略文件和其他我们在这本书的示例中使用到的资源, 以及如何使用这些资源来配置你的系统。

运行在 Permissive (许可) 模式

SELinux 可以运行在 Permissive 模式, 这个模式只存在访问检查, 但不会拒绝不允许的访问, 它只是简单地进行一个审核操作, 这个模式在初次接触 SELinux 学习时非常有用, 你可能想在这个模式下研究系统, 当然, 如果你想增强 SELinux 的访问安全, 不应该在正式的系统上使用 Permissive 模式, 注意/usr/sbin 下的某些工具不是使用的常见用户路径。

检查 SELinux 当前的工作模式最简单的方法是运行 getenforce 命令, 要将系统设置为 Permissive 模式, 运行 setenforce 0, 你必须要以 root 用户在 system_t 域登陆修改系统为 Permissive 模式, 要将其修改强制模式, 运行 setenforce 1, 因为你在 Permissive 模式, 所以只需要以 root 登陆, 将系统修改为强制模式即可。

前面我们已经说过在某些系统命令上添加了 -Z 选项, 如 ls 和 ps 分别显示文件和进程的安全上下文, 请做一个练习, 运行 ps xZ 和 ls -Z /bin 命令, 检查运行中的进程和可执行文件的安全上下文。

2.5.1. 重游 passwd 示例

在这一章中, 我们都使用 shadow 密码文件和密码程序作为例子, 如果检查过这两个文件的安全上下文, 它们的类型分别应该是 shadow_t 和 passwd_exec_t, 正如前面讨论的, passwd_exec_t 是 passwd_t 域的入口类型, 为了证明进程是如何进行域转变的, 按照下面的命令集合进行操作, 你需要两个终端窗口或虚拟控制台。

在第一窗口中, 运行 passwd 命令:

```
$ passwd
Changing password for user joe.
Changing password for joe
(current) UNIX password:
```

启动密码程序，提示输入用户当前的密码，不要输入密码，此时转入第二个终端窗口，在第二个窗口中，su 到 root 用户，然后运行 ps 命令：

```
$ su
Password:
Your default context is root:sysadm_r:sysadm_t.

    Do you want to choose a different one? [n]
# ps axZ|grep passwd
user_u:user_r:passwd_t 4299 pts/1 S+ 0:00 passwd
```

正如你看到的，运行中的密码程序的类型是 passwd_t，与前面例子中描述的规则一样。

注意：在 strict 策略中，普通用户（即在 user_t 域中运行 shell 的用户）无权读取大多数 /proc/pid 条目，因此在 ps axZ 命令的输出中就看不到 passwd 程序，这就是为什么我们要 su 到 root 的原因了。

2.5.2 精读策略文件

在 FC 4 系统中，包括内核策略的二进制文件放在 /etc/selinux/ 目录下，该目录下的配置文件（config）表示策略在启动时载入和使用，你也可以在这个文件中配置系统启动到 permissive 模式，在我们的练习中，我们使用 FC 4 的 strict 策略，它应该位于（如果是按照附录 A 进行安装的）：

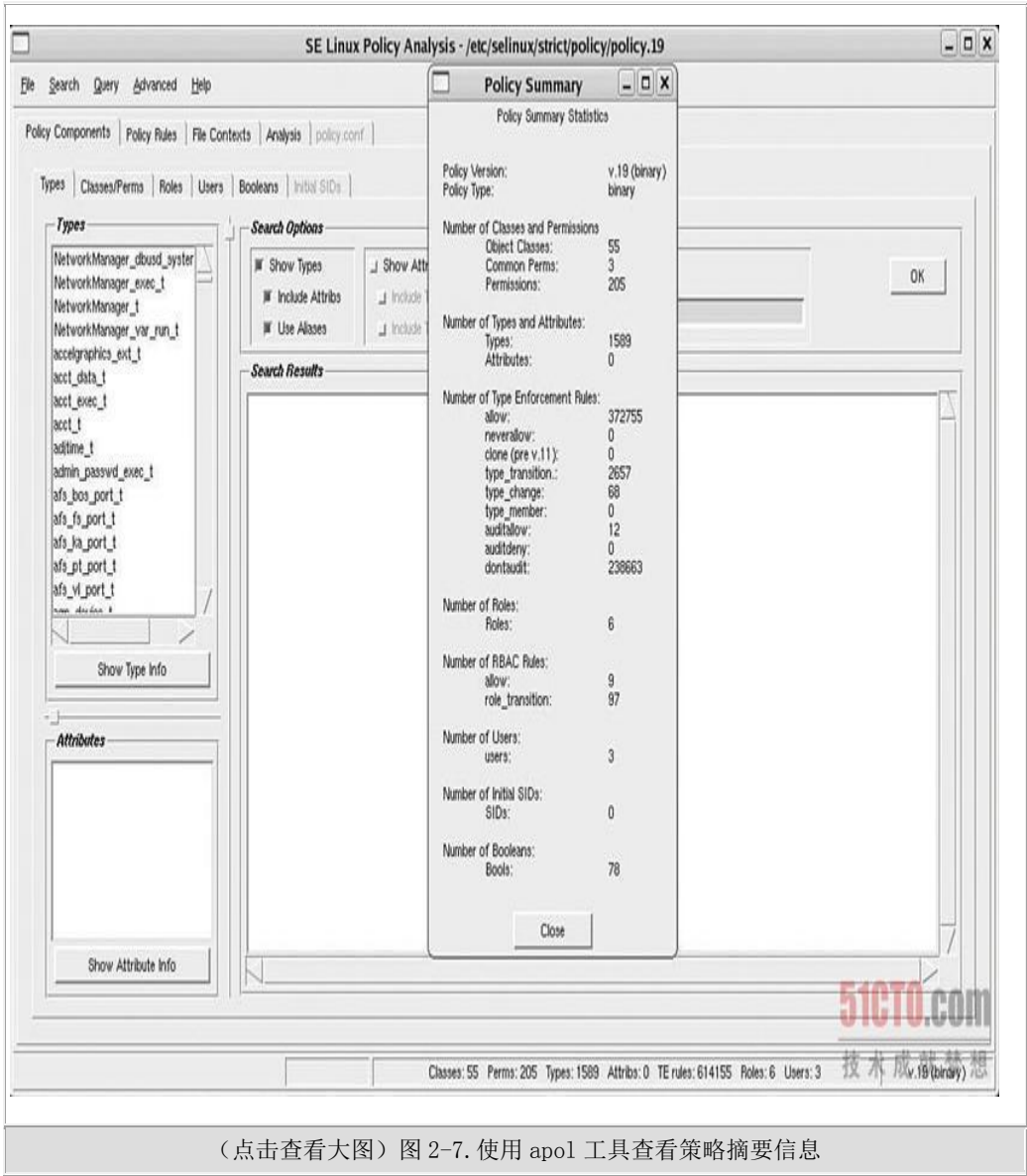
```
/etc/selinux/strict/policy/policy.[ver]
```

策略的版本反应了 SELinux 策略编译器（checkpolicy）的版本，在我们的例子中，版本是 19。第三部分“创建和编写 SELinux 安全策略”将会详细地描述配置 SELinux 系统和从策略源文件创建内核策略文件，现在，我们想深入地研究一下策略内部的东西。

一个检查策略内容非常有用的工具是有 Tresys 科技公司创建的策略分析工具 apol，它随 SELinux 工具包一起发布，叫做 SeTools（参考附录 D“SELinux 命令和实用程序”）。SeTools 软件包包括在大多数 SELinux 发行包中，运行 apol 命令确定这个工具是否已经安装到系统中，如果没有安装，附录 D 提供有关如何获得 SeTools 软件包的信息。

apol（即 analyze policy【分析策略】）工具是一个成熟的 SELinux 策略分析工具，在本书中，我们会一直使用这个工具检查 SELinux 策略，现在，我们想使用它的一些基本特

性检查策略文件,运行 apol 然后打开 strict 策略文件,在菜单 Query【查询】>Policy Summary【策略摘要】下,你可以看到策略统计的摘要信息,如图 2-7 所示。



(点击查看大图) 图 2-7. 使用 apol 工具查看策略摘要信息

apol 有一排主标签页（策略组件，策略规则，分析等），使你可以采取多种方式搜索和分析策略，多花点时间研究一下策略组件和策略规则标签页，熟悉我们在这一章中谈到的策略部分和 apol 工具本身，在第二部分“SELinux 策略语言”中，你会发现它非常有用，使用 apol 沿着例子检查你的策略。

2. 6. 小结

SELinux 访问控制是基于与所有系统资源包括进程关联的安全上下文的，安全上下文包括三个组件：用户、角色和类型标识符。类型标识符是访问控制的主要基础。

在 SELinux 中，访问控制的主要特性是类型强制，在主体（即进程）与客体之间通过指定 allow 规则（主体的类型【也叫做域类型】是源，客体的类型是目标）进行访问授权，访问被授予特定的客体类别，为每个客体类别设置细粒度的许可。

类型强制的一个关键优势是它可以控制哪个程序可能运行在给定的域类型上，因此，它允许对单个程序进行访问控制（比起用户级的安全控制要安全得多了），使程序进入另一个域（即以一个给定的进程类型运行）叫做域转变，它是通过 SELinux 的 allow 规则紧密控制的，SELinux 也允许通过 type_transition 文件使域转变自动发生。

SELinux 在访问控制安全上下文中不直接使用角色标识符，相反，所有的访问都是基于类型的，角色用于关联允许的域类型，这样可以设置类型强制允许的功能组合到一起，将用户作为一个角色进行认证。

SELinux 提供了一个可选的 MLS 访问控制机制，它提供了更多的访问限制，MLS 特性依靠 TE 机制建立起来的，MLS 扩展了安全上下文的内容，包括了一个当前的（或低）安全级别和一个可选的高安全级别。

练习

1、“域”是什么？它与类型是什么关系？或与类型有什么区别？

2、SELinux 类型强制安全使用什么访问控制属性控制访问的？属性的什么部分用于类型强制进行访问控制的？

3、让我们假设有一个名叫 datafile 的文件，它有下列安全属性：

```
-r-xr-xr-x root root system_u:object_r:data_t datafile
```

让我们假设你的 shell 进程类型是 user_t，它对类型为 data_t 的文件客体有所有访问权，你可以读或写这个文件吗？为什么或为什么不能？

4、为了 SELinux 允许域转变，有三个类型是必须要有的，它们分别代表什么？

5、在回答问题 4 时，需要 type_transition 规则吗？为什么或为什么不？

6、在 SELinux 中，角色不是访问控制的基础，但它能阻止域转变成功，它是如何以及为什么阻止的？

附记：检查 SELinux 配置文件/etc/selinux/config，它里面可能包括有哪几种状态，SELinux 可以运行在每个状态下吗？在各个状态下会做什么？在这个文件中进行设置与使用 setenforce 命令有什么不同？

第 3 章. 架构

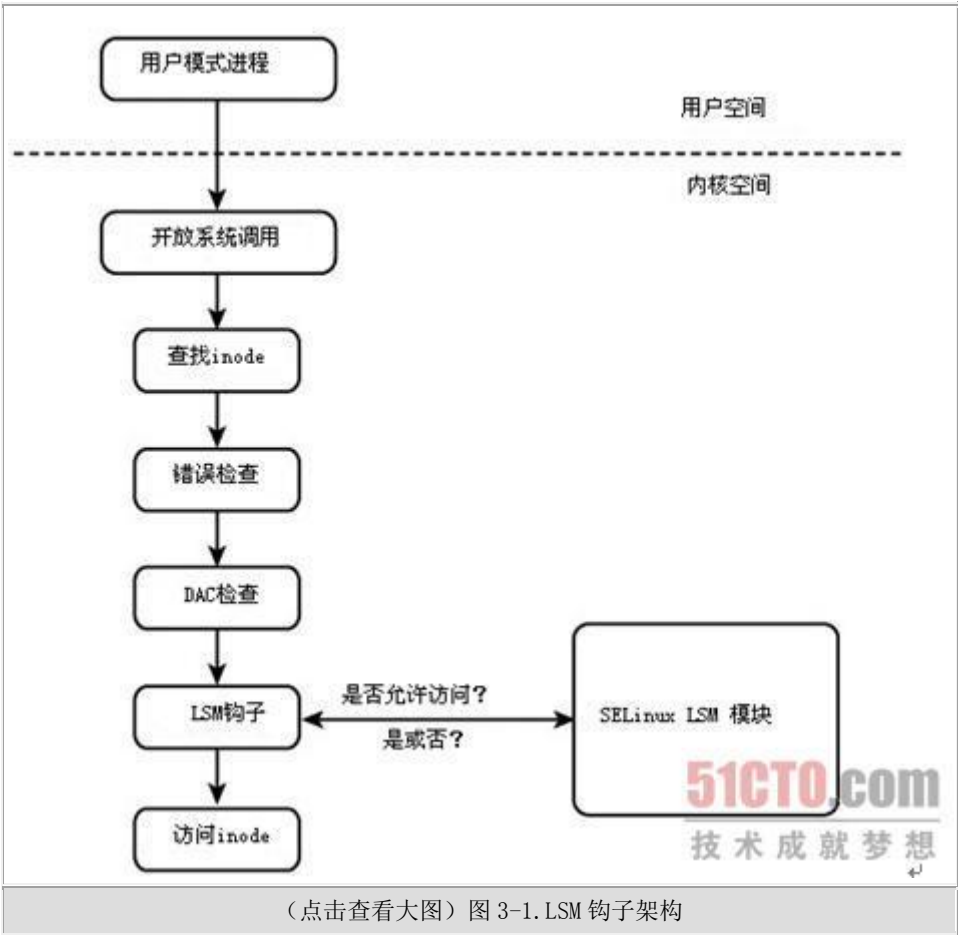
这一章对 SELinux 的设计和它的策略语言提供一个概述，SELinux 架构反应了它起源于安全微内核的研究，使用 Linux 安全模型（LSM）框架将它自身集成到内核中，这个架构也延伸到了用户空间服务器。SELinux 策略语言非常灵活，它允许组织通过强制访问控制实施多个安全目标。

3.1. 内核架构

SELinux 在所有内核资源上提供增强的访问控制，在它目前的格式下，SELinux 是通过 LSM 框架合并到内核中的。

3.1.1. LSM 框架

LSM 框架后面包含的思想是允许安全模块以插件形式进入内核，以便更严格地控制 Linux 默认的基于身份的任意访问控制（DAC）安全性。LSM 在内核系统调用逻辑中提供了一套钩子（hooks），这些钩子通常放在标准 Linux 访问检查后内核调用访问真实资源之前，图 3-1 举例说明了 LSM 框架的基础。



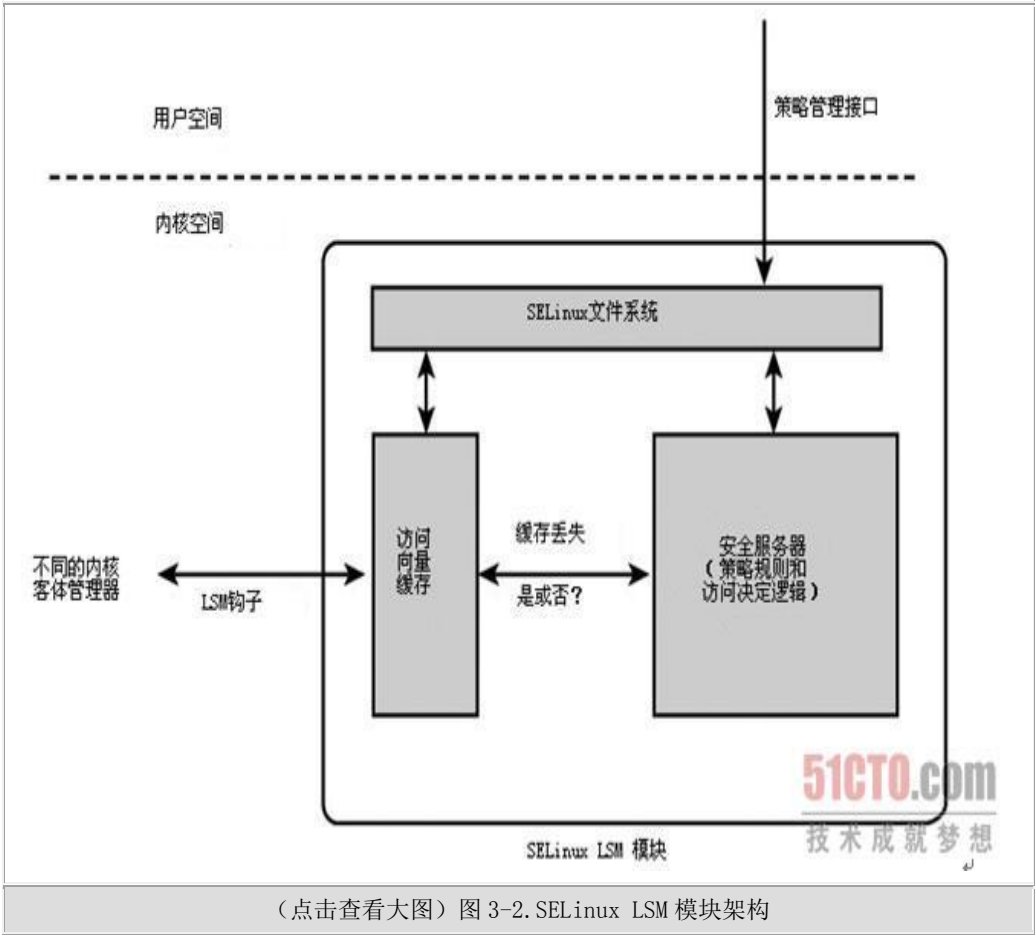
SELinux 作为一个 LSM 模块载入内核，在访问被允许之前进行额外的访问确认。

LSM 框架的一个分支是只有当标准 Linux 访问检查成功后 SELinux 才会生效。实际上，这在访问控制策略方面没有负面影响，因为 SELinux 访问控制比标准 Linux DAC 约束更多，而且它也不会覆盖 DAC 的决定。然而，LSM 框架可能影响由 SELinux 收集的审核数据。例如：如果你想使用 SELinux 审核数据来检测所有访问拒绝信息，注意大多数情况下 SELinux 不会被考虑，因为标准 Linux 安全就拒绝了，因此也不会审核。

LSM 框架很容易理解，钩子分散在内核的各个地方，每个 LSM 钩子被解释为一个或多个对一个或多个客体类别的访问许可，理解了 SELinux 中的客体访问许可再理解 LSM 钩子就容易多了，第 4 章“客体类别和许可”将会详细地讨论客体类别和许可。

3.1.2. SELinux LSM 模块

SELinux 内核架构反射出了 Flask 架构，Flask 架构是设计用于微内核环境的，它包括三个主要的组件，如图 3-2 所示：安全服务器、客体管理器和访问向量缓存。



(点击查看大图) 图 3-2. SELinux LSM 模块架构

Flask 设计使得安全策略决定和强制功能之间产生了巨大的差异，策略决定是安全服务器的工作，安全服务器反射出 SELinux 微内核的根，策略决定角色是被封装在一个用户空间服务器中的。在 Linux 中，内核对象的的安全服务器位于 SELinux LSM 模块中。安全服务器使用

的策略是潜入在一个规则集中的，通过策略管理接口进行载入。系统与系统之间这些规则可以不同，这样 SELinux 就可以满足不同组织的不同安全目标。架构是设计好的，以便安全服务器完全可以用不用修改剩下的架构实现一个全新的访问控制策略的逻辑进行替换，实际上，新的安全服务器是不需要的，因为类型强制提供了充足的灵活性，几乎可以适应任何访问控制安全策略。

客体管理器负责对它们管理的资源集强制执行安全服务器的策略决定，对于内核，你可以认为客体管理器是一个内核子系统，它创建并管理内核级客体。内核客体管理器的实例包括文件系统、进程管理和 System V 进程间通信（IPC）。在 LSM 架构中，客体管理器是通过 LSM 钩子描绘的，这些钩子分散在内核子系统各个地方，调用 SELinux LSM 模块做出访问决定。然后，LSM 钩子通过允许或拒绝对内核资源的访问强制执行这些决定。

SELinux 架构的第三方组件是访问向量缓存（AVC），AVC 缓存决定是由安全服务器为后面的访问检查准备的，目的的提升访问确认的速度。AVC 还为 LSM 钩子和内核客体管理器提供了 SELinux 接口。

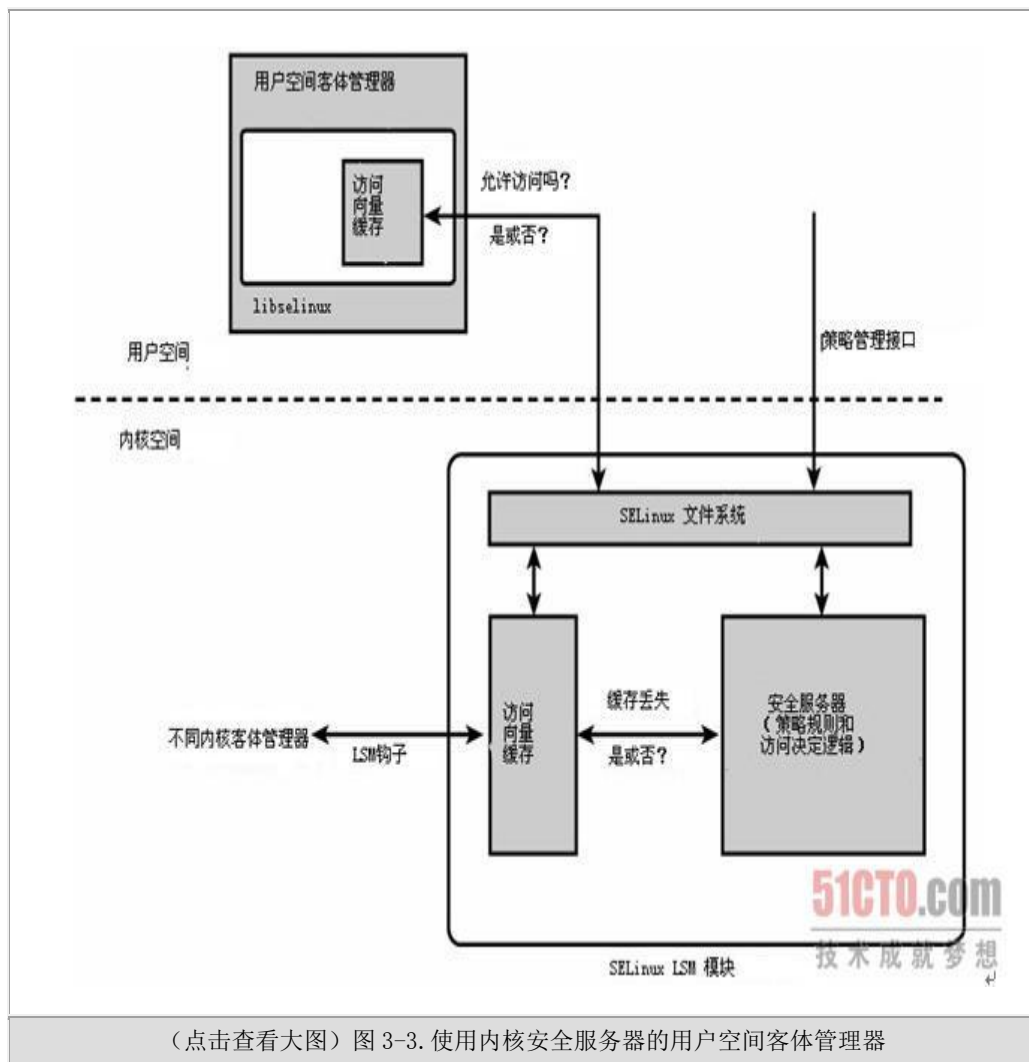
当某个策略被载入后，AVC 就是无效的了，因此要保证缓存的连续性，然而，当策略变化后，SELinux 没有完全实现访问撤销，在标准 Linux 中是不用担心这个的，因为它根本就没有访问撤销，在标准 Linux 中，如果你有一个文件描述符，你就可以访问它，不管文件访问模式是否发生了改变，在 SELinux 中，对于客体，如文件，访问是要进行确认的（例如：每一个与策略有关的系统读取调用都会被检查，但不会打开调用），访问是可以撤销的。只有一个文件描述符并不意味着对文件的访问就有权限，对于某些资源，如内存映射文件和方向连接套接字，只有当资源是初始访问并且后面没有使用时，访问才是有效的。在这种情况下，退出访问是无效的，我们希望将来在 SELinux 中加深对访问撤销的研究。

3.2. 用户空间客体管理器

SELinux 架构强大的特色之一是它可以被应用到用户空间资源和内核资源。实际上，它起源于微内核研究，大多数资源管理都是由用户空间服务器完成的，Linux 下用户空间服务器的实例可以强制在它们的资源包括 X 服务器和数据库服务上进行访问控制以提高安全性，每个服务器都提供了抽象的资源（窗口，表等），这一节检查 SELinux 架构支持用户空间服务器的两条路线。

3.2.1. 用户空间客体管理器的内核支持

SELinux 支持用户空间客体的一个简单方法是直接通过内核安全服务器，如图 3-3 所示。



(点击查看大图) 图 3-3. 使用内核安全服务器的用户空间客体管理器

在这个方法中，用户空间客体管理器行为与内核客体管理器行为非常类似，内核安全服务器包括全部安全策略，用户空间客体管理器必须查询内核访问控制决定，主要的区别是用户空间客体管理器不能使用内核 AVC，每个服务器都必须有其自身的，独立的 AVC 存储过去的请求自内核的决定，用户空间的 AVC 功能包括在 libselinux 库中。

另一个区别是用户空间客体管理器没有 LSM 钩子，LSM 钩子属于内核空间的概念，相反，客体管理器有与它的 AVC 内部接口，AVC 处理缓存丢失并代表客体管理器查询内核。

坦白地说，这个支持用户空间客体管理器的方法有许多弱点，首先，要使用类型强制，客体管理器必须定义客体类别代表它们的资源，例如：一个数据库服务器可能定义的客体类别包括数据库、表、方案、记录等，对于内核资源，客体类别是固定的，与定义在 SELinux LSM 模块头文件中的硬编码类别偏移是相符的。在策略中类别定义的关系以及在内核代码中的那些关系，在用户空间策略和代码之间产生了不恰当的依赖，特别地，两个用户空间服务器都必须注意在内核中不能使用相同的客体类别偏移，内核没有提供管理这种可能发生的冲突的解决办法。

这个方法的第二个弱点是内核安全服务器是为客体类别管理策略，由于客体管理器没有在内核中，这就增加了内核中的存储成本，并对内核策略校验 AVC 丢失成本产生了消极的影响。

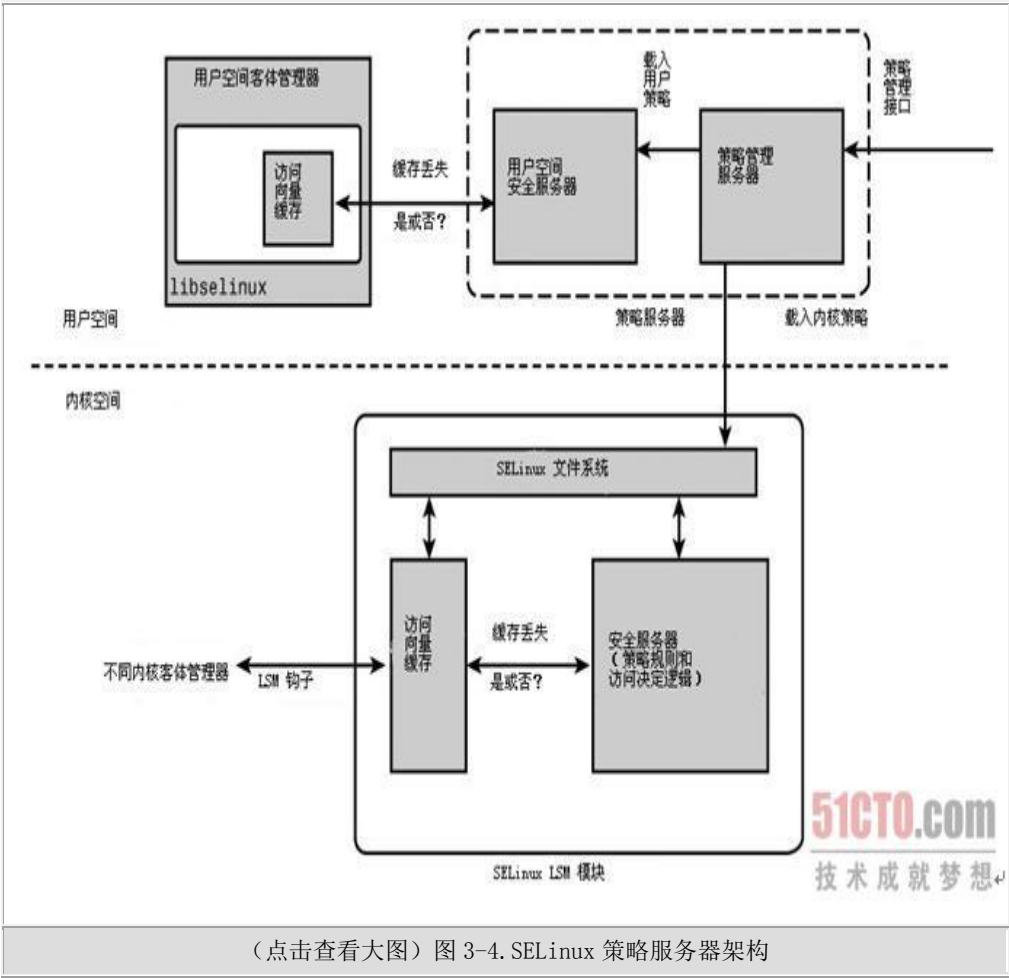
3.2.2. 策略服务器架构

为了描述为用户空间客体管理器使用内核安全服务器的弱点，并增强 SELinux 的安全能力，正在进行的一个努力是为用户空间客体管理器建立用户空间支持，这个项目有两个主要目标和大量的二级目标，主要目标是：

通过提供用户空间安全服务器为用户空间客体管理器提供更好的支持，由安全服务器产生策略中用户命运的访问决定。

通过建立一个策略管理服务器为策略本身提供细粒度访问控制，策略管理服务器是一个用户空间客体管理器，它的客体类别代表了策略的命运。

总的来说，这两个服务器都涉及到了策略服务器，图 3-4 描述了策略服务器的架构。



在策略服务器架构中，所有系统策略的操作和管理都是通过策略管理服务器（PMS）控制的，

PMS 本身是一个用户空间客体管理器，它创建代表客体资源的客体类别，并在这些资源上实施细粒度访问控制策略，这个特性为 linux 单独提供了有效的安全强化，以前，对策略的访问控制是一个非全有既全无的命题，要么能够写策略文件，要么不能写，在 PMS 中，你可以允许访问策略的一部分，并限制访问其他部分。例如：SELinux 策略可以允许用户管理工具添加用户和分配角色，但不能改变类型强制 allow 规则。另外，你还可以委托一个数据库服务器改变类型强制（TE）规则关联的客体类别和类型，但不能对内核做改动，在内部，PMS 被设计用于另一个最新的 SELinux 特性，可载入式策略模块，我们将在后面的章节中讨论它。

PMS 第二个主要功能是将系统策略分割成内核部分和用户部分，然后分别载入内核安全服务器和用户空间安全服务器（USSS），因此，内核不用知道规则和客体类别，只要用户空间客体管理器知道就可以了，用户客体管理器查询 USSS 而不是内核，在不同用户空间可以管理器中的 AVC 都注册到 USSS（不是内核）以进行策略更新和缓存连贯功能。

策略服务器架构除了移除了内核的用户空间资源的可靠性以及策略管理的细粒度访问外，它还做了大量的强化规则，因为 PMS 是一个运行中的服务器，我们可以扩展它的接口以允许远程网络访问分布式策略管理，PMS 和 USSS 被设计允许运行时注册客体类别，打破了存在于内核中的用户空间客体管理器代码依赖，这两种途径的区别通过 libselinux 屏蔽掉了，提供向后的兼容性，最后，PMS 和 USSS 设计为独立的服务，这样允许一个或两者独立使用，例如：在一个无需细粒度策略访问控制的系统中，USSS 可以独立使用以支持其他的用户空间客体服务器。

在写本书时，策略服务器已经在开发之中了，但没有完全集成到所有发行版中，你可以在 <http://sepolicy-server.sourceforge.net/> 检查这项工作的最新状态。

3.3. SELinux 策略语言

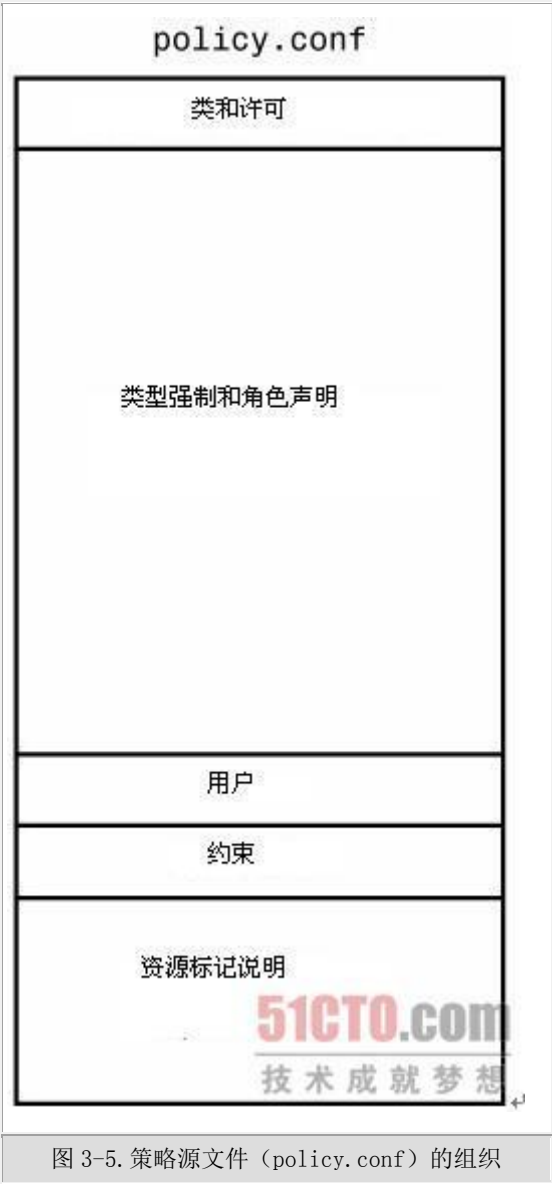
第 2 章“概念”对 SELinux 安全概念做了一个概述，并介绍了部分策略语言概念。在前一小节中，你看到了在 SELinux 架构中策略是如何被使用的，对于内核资源，策略被载入 SELinux LSM 模块安全服务器用于产生访问控制的决定，SELinux 的一个强度是它的策略规则不是静态的，相反，人们（或大多数人）必须自己编写策略并确保它达到了安全目标的要求，幸运的是，本书都是围绕如何编写 SELinux 策略而写的（以及如何确保它们是最好的策略），其实使用和应用 SELinux 就是编写和理解策略。

在本书的第二部分中，我们将讲述策略的每个主要部分并详细讨论策略语言语法和语义。在这一小节中，我们概述了一个策略是如何构建和编译的，并向你展示如何从我们在本书中使用的 strict 示例策略建立一个策略。

3.3.1. 本地 SELinux 策略语言编译器

为内核构造一个策略文件的主要方法是从一个源策略文件使用 checkpolicy 程序编译它，这个源文件本身也是经过多个步骤构造的，它的名字通常是 policy.conf，checkpolicy 检查源策略文件的语法和语义的准确性，并将结果以某种格式存储起来（叫做为二进制策略文件），由内核策略载入器（load_policy）进行读取，checkpolicy 支持的语言体系就是 SELinux 支持的本地的，简单的语言。你可以认为 checkpolicy 语言与汇编语言类似，高级语言和其他更抽象的创建策略的方法目前还在开发之中，本书后面将会做介绍，现在，我们主要集中讲解本地策略语言和策略的构造。

图 3-5 举例说明了策略源文件的主要部分。



策略源文件的第一部分定义了安全服务器的客体类别，同时，这一部分也定义了每个客体类别的许可，对于内核而言，这些类别直接关系到内核源文件，通常，作为一名 SELinux

策略编写者，你可能永远不会修改客体类别和许可定义，我们将在第 4 章讨论特定的客体类别和与之关联的许可。

第二部分包括类型强制声明，它是一个 SELinux 策略中最大的一部分，也是策略编写者花费时间最多的一部分，它包括所有的类型声明和所有的 TE 规则（包括所有的 allow，type_transition 和其他 TE 规则），我们将在第 5 章“类型强制”中详细讨论类型和核心 TE 规则，TE 部分通常包括上千的类型声明和 TE 规则，这一部分还包括规则和角色及用户的声明，角色和用户是类型强制支持的概念，我们将在第 6 章“角色和用户”中详细讨论，最近最 TE 策略部分做了一些增强，特别是有条件的策略，我们将在第 9 章“条件策略”中讨论。

策略源文件接下来的一部分是约束，约束在 TE 规则许可范围之外对 TE 策略提供了更多的限制，例如：多级安全（MLS）策略就是约束的一种实现，我们将在第 7 章“约束”和第 8 章“多级安全”章讨论约束。

策略源文件的最后一部分包括标记说明，所有客体都必须用一个安全上下文标记 SELinux 以实施访问控制，这部分告诉 SELinux 如何处理文件系统标记以及标记运行时创建的临时客体规则，另一个独立有关的机制叫做文件上下文文件，它用于在永久文件系统上初始化文件、目录和其他客体的安全上下文标记，这些以及另外的与客体标记有关的内容将在第 10 章“客体标记”中讨论。

policy.conf 文件示例

使用由 checkpolicy (policy.[ver]) 创建的二进制策略文件，你可以使用 Tresys apol 工具查看、搜索和分析 policy.conf 文件的内容，policy.conf 文件比为二进制格式文件更难以理解，二进制文件更易于分析和调试，同样，policy.conf 文件最接近最原始的源模块形式，因此，它也是最好的跟踪原始源文件 bug 的形式，在任何情况下，它们都是相等的，并且应该反应相同的安全策略。

3.3.2. 单个策略中的源策略模块

目前 SELinux 策略常见的类型是单策略，它是一个由 checkpolicy 构造的单二进制策略文件，它直接载入内核，因为 SELinux 策略通常比较大而且非常复杂，与软件类似，它们是由一些比较小的叫做模块的单元构成的，有多种方法产生策略模块，最原始也是最广泛使用的方法叫做源模块法，它支持单策略的开发，源模块通过一组 shell 脚本、m4 宏和 Makefiles 一起合并为文本文件，策略模块实际上是被集合到一块然后聚成一个大的源文件（即 policy.conf），然后由 checkpolicy 进行编译，编译完成后就成为内核可读的二进制文件了。

3.3.3. 载入式策略模块

一个新的创建模块策略的方法叫做载入式模块，它使用了最近对 `checkpolicy` 的扩展和一个模块编译器（`checkmodule`）来构造载入式策略模块，相互独立编译，不互相依赖，载入式模块也是本章前面讨论的策略服务器的基础，在载入式模块中，没有单二进制策略构造了，相反，构造的是一个叫做基础模块的策略核心子集，创建基础模块与你创建单策略非常类似，只包括了与核心操作系统有关的规则，当你安装了有关软件包时，你可以将它们的策略规则以模块化的形式进行添加。

载入式模块引入了策略语法修改，这样设计后就更容易将策略分成独立的，一个一个的分布式的策略模块，基础模块和非基础模块之间的修改是不同的，基础模块使用与单策略相同的策略语言，只有一点点的增加功能，非基础（即载入式的）模块使用标准策略语言的子集，不过添加了一些额外的语言特性，策略语言的子集包括了大部分的类型强制、角色和用户声明，额外添加的语言特性用于管理模块之间的依赖，我们将在第二部分中详细讨论载入式模块语言修改结果形式。

Fedora Core 5 (FC 5) 已经采用了载入式模块架构，在本书中，我们主要讨论单策略方法和语言，但我们将使用侧边栏的形式介绍最新的载入式模块特性。

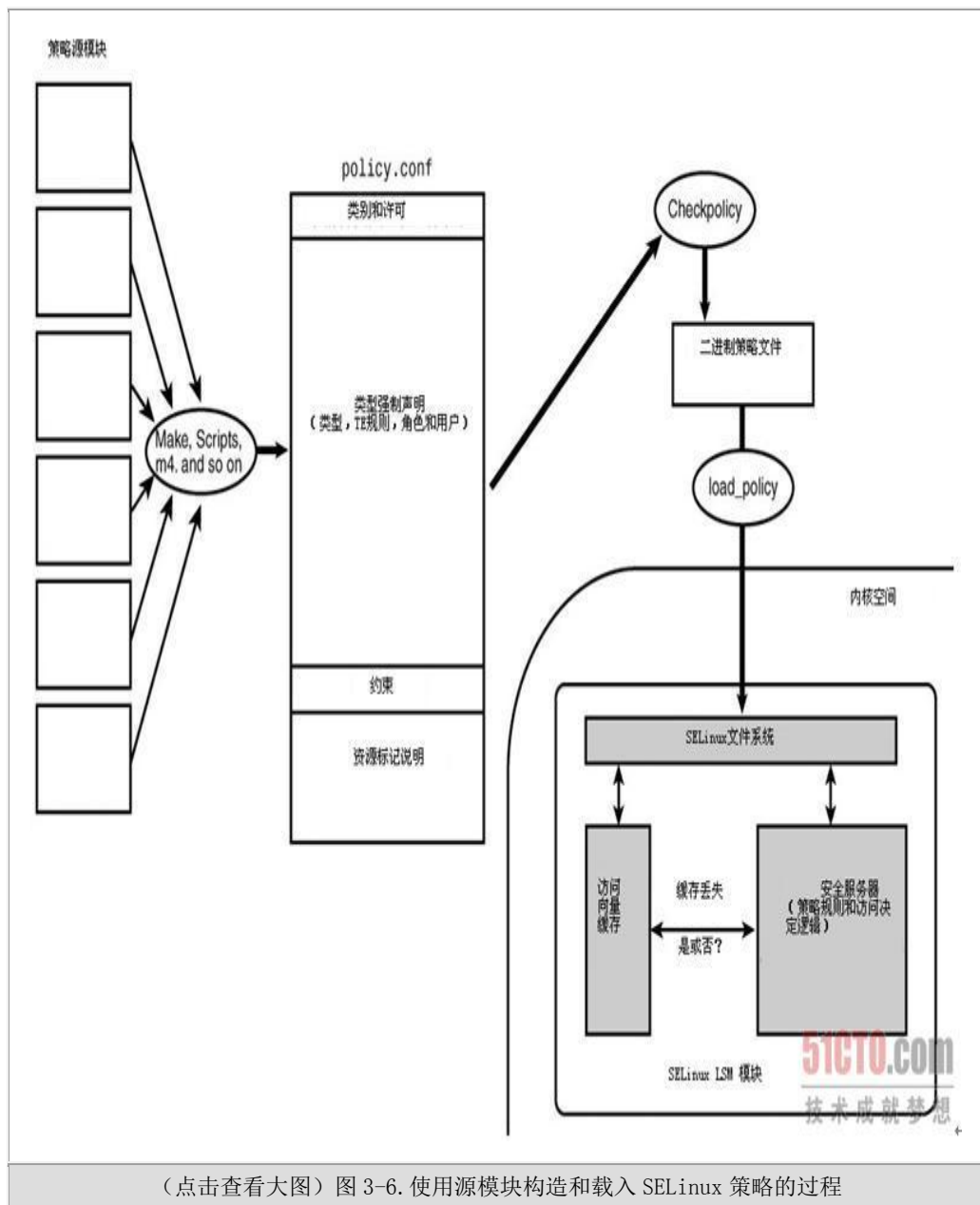
3.3.4. 构建和安装单策略

通过阅读本书剩余部分，你将会想要实验一下 SELinux 策略的编写，你需要编译你的修改以形成一个完整的策略文件，通过载入新的策略到内核对你的修改进行试验，并体验对内核的访问控制实施改变的结果，在你能够完成这些行动前，我们必须介绍一下构建和安装内核安全策略的基础知识。

提示：

记住如果你安装了你自己的策略，内核将会立即开始实施基于策略中规则的访问控制，你在学习 SELinux 和实验时，可能会出现因为无访问权而导致程序崩溃的现象，除非你对策略语言和它的分支非常熟悉，我们建议你将系统设为许可（`permissive`）模式（`setenforce 0`）进行策略编写的试验，当然，在生产系统中你总应该以强制模式（`setenforce 1`）运行。

示例策略构造方法（参考第 11 章“原始示例策略”）是构造策略常用的一个方法，图 3-6 显示了这种类型的构造。



(点击查看大图) 图 3-6. 使用源模块构造和载入 SELinux 策略的过程

从这个图的左边可以看出，策略源文件被分为许多独立的源模块，本书后面，我们将会谈到在示例策略中组织这些模块的多种约定，现在，只需要知道这些文件是通过一组脚本和宏处理形成一个 policy.conf 文件的，它是一个完整的语法正确的 SELinux 源策略语句，然后，你可以使用 checkpolicy 将这些源策略编译成二进制策略文件（假设没有错误！）供内核使用，然后，load_policy 程序载入二进制策略文件进入内核，然后内核实施基于策略规则的访问控制。

至此，你可能发现这个过程是无法抵抗的并且也让人很困惑，特别是我们讨论到的构造策略的方法和从源模块构造策略的方法，不必惊慌，我们只是想让你对整个过程有一个大致的了解，策略源目录下通常还有一个 Makefile 文件，它使得这些过程全部自动执行，在第

二部分中我们要用到的策略中，如果安装正确的话，应该在
/etc/selinux/strict/src/policy/，与之有关系的是下面三部分：

`policy` 本地产生 `policy.conf` 和 `policy.[ver]` 测试编译并检查错误。

`install` 实际上就是完成 `make policy` 做的事情，加上安装二进制策略文件，以便在启动时载入内核。

`load` 实际上就是完成 `make policy` 做的事情，并立即载入二进制策略文件进入内核，作为活动的访问控制策略，并安装 `file_contexts` 文件。

因此，例如，`make policy` 将会执行图 3-6 中所有的步骤，除了最后一步外（安装二进制策略并载入内核）。

可以使用我们的示例策略自由感受不同的 `make` 目标，只是在 `make install` 或 `make load` 时需要注意，因为这将会在你的系统上实施访问控制。

3.4. 小结

SELinux 在内核中以一个 LSM 模块的形式实现，SELinux 使用 LSM 钩子控制对内核资源的访问，访问决定由 SELinux 安全服务器产生，它是 SELinux LSM 模块的一部分，安全策略实施由安全服务器通过一个具有特权的用户空间接口载入内核，AVC 为访问确认提供性能增强。

SELinux 框架也支持通过 `libselinux` 库对用户空间客体进行管理，内核安全服务器直接提供访问确认，而程序库包括每一个进程 AVC，这个方法需要内核保留策略所有用户空间管理器以及所有用户空间客体类别。

策略服务器架构通过提供用户空间安全服务器增强支持用户空间客体管理器，用户空间安全服务器实施所有与用户空间客体有关的策略，从而减轻内核的负担，因为内核需要知道用户空间客体类别和策略规则，策略服务器对策略本身也会提供细粒度的访问控制，允许策略管理分布更广。

SELinux 策略越来越大和复杂，它们必须以模块的形式进行构造，最常用的一种方法是使用源模块，所有的模块都以一个单独的模块进行构造，这也是 Red Hat Enterprise Linux 4 和 Fedora Core 4 使用的方法。

第二个模块性方法提供了载入式模块，策略块可以不依赖其他模块进行构造，在系统上安装时才进行合并，在载入式模块中，基础模块的创建某种程度上与单策略类似，但基础模块可以更小，新添加的软件包将以独立的载入式模块形式安装它们的策略，这个方法被 FC 5 采用。

checkpolicy 是策略编译器，它获取一个完整的策略源文件（policy.conf）并进行语法和语义的校验，然后创建一个二进制策略文件，对于载入式模块而言，checkpolicy 编译基础策略模块，然后，checkmodule 程序编译每一个载入式模块。

练习

1. 在 LSM 框架中，首先检查的是什么？标准 Linux 访问检查还是 SELinux 检查？为什么？
2. 在内核中，SELinux 客体管理器与 LSM 钩子是如何关联的？
3. 当一个新的策略载入内核时，访问向量缓存（AVC）是无效的，你觉得为什么那是不可避免的？
4. 即使 SELinux 没有实现策略改变访问撤销，标准 Linux 访问控制也没有实现访问撤销，解释它们之间的差异。
5. 你认为为什么用户空间客体管理器不能象它们使用内核安全服务器那样使用内核访问确认缓存？
6. 在策略服务器架构中，没有策略管理服务器的用户空间客体管理器有意义吗？为什么或为什么没有？

结尾提示：进入示例策略源目录，执行 make policy 创建 policy.conf（源）和 policy.[ver]（二进制）策略文件，使用 apol 工具检查每个文件 allow 规则的数量，注意大的差异，对于那些差异是怎么产生的你有什么想法吗？

第二部分：SELinux 策略语言

第 4 章. 客体类别和许可

本章涉及 SELinux 中定义的客体类别和许可，我们讨论定义内核支持的客体类别和许可的策略语言指令，并对 SELinux 系统中内核客体类别标准做一个概述，附录 C：“客体类别和许可”包括了一个有关 SELinux 客体类别和与之关联的许可标准清单。

4.1. SELinux 中客体类别的用途

客体类别及其许可是 SELinux 中访问控制的基础，客体类别代表资源的范畴，如文件和套接字，许可代表对这些资源的访问权限，如读或发送。理解客体类别和许可是 SELinux 中比较困难的部分，因为这需要 SELinux 和 Linux 两方面的知识。

一个客体类别代表某个确定类型（如文件或套接字）的所有资源，一个客体类别的实例（如某个特定的文件或套接字）被称为一个客体，通常，客体类别和客体这两个术语可以交

替使用，但是理解其定义是相当重要的，客体类别指的是资源（文件）的所有范畴，客体指的是客体类别的某个特定实例（/etc/passwd）。

正如第 2 章“概念”中描述的，客体的访问权限是在策略中通过客体类别的许可表达的，我们从第 2 章引用一个 allow 规则来进行说明：

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

在这个规则中，使用类型 user_t 的进程允许读、执行和获取所有目标类型在它们的安全上下文中被标记为 bin_t 的 file 类客体的属性，客体类别 file 指定了资源的范畴，bin_t 指定了这个规则应用到的资源范畴的具体实例（也就是说，这些文件客体类型为 bin_t），但它不会应用到类型为 bin_t 但类别不是 file 的客体，也不会应用到类型不是 bin_t 的文件客体。

这个规则中的许可 read, execute 和 getattr 定义了类型为 user_t 的主体（暗指进程）对那些客体允许的访问权限，这些许可对文件客体类别必须是有效的，它们代表了对客体的访问形式，例如：使用 open(2) 系统调用打开一个文件进行读操作，在一个打开的文件上使用 read(2) 系统调用时需要 read 许可。给一个客体类别定义的许可设置代表所有可能允许对资源的访问权限。

客体类别设置的有效性依赖于 SELinux 及其 Linux 内核的版本，新的不同的客体类别对内核做了扩展，对现有的部分特性进行了改进，并加入了新的特性，例如：新版本的 Linux 内核为控制审核框[1]引入了一个新的 Netlink 套接字，对于那些支持 Netlink 套接字的内核而言，需要定义恰当许可的 SELinux 客体类别。

[1]Linux 审核框架和工具的有关信息和源代码可以在 <http://people.redhat.com/sgrubb/audit/> 找到。

4.2. 在 SELinux 策略中定义客体类别

策略中必须包括所有 SELinux 内核支持的客体类别和许可的声明，以及其他客体管理器。通常，对于策略开发者而言，我们并不关心创建一个全新的客体类别，但要想写出高效的 SELinux 策略，对客体类别是如何定义的必须要理解透彻。理解客体类别和许可声明的语法是有用的，因为它让我们有机会理解在当前使用的策略版本中支持的客体类别和许可。

添加新的客体类别和许可

添加新的客体类别和修改现有客体类别的许可是一项复杂的任务，仅当修改真实系统代码本身时应该这样做，与 SELinux 策略语言的其他方面不同，客体类别和许可依赖于 Linux

的实现细节，特别是内核。实际上，客体类别和许可被设计为尽可能准确地代表系统执行的资源，正是由于这个原因，改变客体类别或在系统中改变对应的许可才变得有意义。

保证客体类别和许可中类型改变的一个实例是一种新形式的进程间通信（IPC），在这个例子中，一个全新范畴的资源被添加，一个新的客体类别也将会是必须的，它准确地代表了这个资源的语义。

添加或修改客体类别或许可需要同时修改策略和基于新的客体类别或许可的强制访问控制系统代码。如果只添加了一个客体类别，但没有修改系统代码，除了浪费核心内存外，可能没有什么作用。

基本上，除了本书的目标听众（SELinux 策略编写者和系统管理员）外，你应该永远都不会改变客体类别和许可的定义。

4.2.1. 声明客体类别

客体类别是使用类别声明语句声明的，类别声明语句只是简单地声明了一个客体类别名字，无其他内容了，例如：我们使用下面的指令为目录声明了一个客体类别（名叫 dir）：

```
class dir
```

类别声明语句是由关键词 `class` 跟上类别名字组成的，注意类别声明语句与其他策略语句的区别，在声明语句的末尾是没有分号的，你可以在 63 页的侧边栏看到完整的类别语句语法。

客体类别名字具有独立的命名空间，但实际编写策略时，客体类别，许可，类型等通常都是使用的相同的名字。

类别声明语句语法

类别声明语句允许你声明客体类别名字，完整的类别声明语句语法如下：

```
class 类别名字
```

类别名字 客体类别的标识符，它的长度不定，可以包括 ASCII 字母或数字。

类别声明只在一个策略和基础载入模块中有效，在有条件限制的语句和非基础载入模块中都是无效的。

4.2.2. 声明并连接客体类别许可

声明许可有两种方法，第一种叫做通用许可，它允许我们创建与客体类别一起作为一个组的许可，通用许可在类似的客体类别（如文件和符号连接）共享一套访问许可时很有用；

第二种方法叫做特定类别许可，它允许我们单独为客体类别声明特定的许可，正如将会看到的，有一些客体类别只有特定的许可，有一些只有通用许可，还有一些是这两者都有。

4.2.2.1. 通用许可

通用许可语句允许创建一套与两个或更多客体类别一起构成组的许可，完整的通用许可语句语法在 64 页的侧边栏中，例如：UNIX 基本原理“一切都是文件”意味着许多与文件有关的客体类别都有一套通用许可，在 SELinux 中，通用许可声明这些与文件有关的许可的语句如下：

```
common file
{
    ioctl
    read
    write
    create
    getattr
    setattr
    lock

    relabelfrom
    relabelto
    append
    unlink
    link
    rename
    execute
    swapon
    quotaon
    mounon
}
```

这个语句声明了一套叫做 file 的通用许可集，并定义了一套许可，如 ioctl, read, write, create 等，通用许可语句本身没什么用，仅当我们要将通用许可与客体类别联合在一起时才有用。

与客体类别一样，通用许可名字是在它们自己的命名空间中声明的，如果我们不注意，这样可能会引起某些混淆。例如：如前面所有举的例子，我们的客体类别和通用许可名都是 file。虽然名字是相同的，但事实上在策略中，它们非常明显是不同的组件。

通用许可语句语法

通用许可语句允许你声明一个通用许可名字，许可可以与客体类别一起组成一个组，通用许可可以与多个客体类别进行联合，完整的通用许可语句语法如下：

common 通用名 {许可集}

通用名 通用许可标识符，该标识符长度不限，可以包括 ASCII 字母，数字，破折号 (–) 和句号 (.)。

许可集 在一个独立空间列表中的一个或多个许可标识符，标识符长度不限，可以包括 ASCII 字母，数字，破折号 (–) 和句号 (.)。

通用许可集与客体类别一起使用访问向量语句。

通用许可集只在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。

4.2.2.2. 联合许可和客体类别

我们使用访问向量语句将许可和客体类别联合起来了，访问向量语句的完整语法参见 66 页的侧边栏内容，我们使用访问向量语句联合通用和特定类别许可，例如：下面的语句将一个特定类别许可与客体类别 dir 联合起来了：

```
class dir { search }
```

从这个例子看起来，访问向量语句与类别声明语句非常类似，都使用了关键词 class，但实际上，类别声明语句与访问向量语句是截然不同的，即使都是以 class 开头，访问向量语句必须提供一个事先声明的客体类别名 (dir)，然后再提供一个或多个许可，在这个例子中，我们只定义了一个特定类别的许可 search，注意这个语句的结尾也是没有分号的。

前面的访问向量语句将使 dir 客体类别产生一个特定类别的许可：search，通常，你看到的客体类别的许可应该有多个，如：

```
class dir { search add_name remove_name }
```

这个例子将三个特定类别许可与 dir 客体类别联合到一起了，在访问向量语句中，我们也可以使用关键词 inherits 来联合通用许可，例如：dir 客体类别是多个文件类似的客体类别与其他文件类似的客类别可之间共享通用许可的客体链表之一，下面的访问向量语句是 dir 与通用许可 file 联合的完整语句，除了前面显示的内容外，还增加了几个与目录有关的特定类别许可：

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

```
search
rmdir
}
```

在这个例子中，我们使用了关键词 `inherits`，后面跟前面声明的许可集名字，并联合了所有的通用文件许可，这个语句的结果是客体类别 `dir` 的有效许可是前面定义的通用许可 `file`，以及五个为 `dir` 指定的许可。

只有通用许可的客体类别是可以存在的，如：客体类别符号链接文件 (`lnk_file`) 的访问向量语句是：

```
class lnk_file inherits file
```

这个语句导致类别 `lnk_file` 只有在通用许可 `file` 中定义的那些许可。

同样，只有特定类别许可的客体类别也是可能的（即没有通用许可），如：代表文件描述符 (`fd`) 的客体类别访问向量语句只有一个特定类别的许可允许使用一个文件描述符。

```
class fd { use }
```

访问向量语句语法

联合许可及之前声明的客体类别的访问向量语句，它的完整语法如下：

```
class 类别名 [inherits 通用许可集名] [{许可集}]
```

类别名 --- 前面声明的客体类别名

通用许可集名 --- 前面声明的通用许可集名

许可集 --- 在一个独立空间列表中的一个或多个许可标识符，标识符的长度不受限制，包括 ASCII 字母，数字，或句号 (.)。

最低情况下，至少要指定一个通用许可集名或一个许可集，但这两者是可以同时提供的，最后所得到的许可是联合了通用许可和许可集的并集。

访问向量语句只在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中是无效的。

4.3. 有效的客体类别

这一章对 Fedora Core 4 (FC 4) 中可用的内核客体类别做了一个概述。我们的目标是描述客体类别以及系统资源是如何映像到这些客体类别的。附录 C 提供了一份所有客体类别及其关联的许可的参考，编写一个良好的策略最困难的部分是理解客体类别和许可的语义，以及特定系统上应用程序上下文中那些语义的含义。

FC 4 系统有超过 40 个内核客体类别，它们代表了内核提供的所有资源。客体类别的数量说明了一个基本原理，在 SELinux 中代表尽可能完整准确的内核资源，Linux 涉及的范围广泛且非常复杂，这就决定了客体类别肯定也是及其多且复杂的，复杂得让人畏缩，但要取

得 SELinux 的灵活性，完全解决 Linux 面临的安全挑战，这是必需的。使用 SELinux 的工具和技术正在逐渐形成，以后人们将不用再担心底层的复杂性，使用工具就可以完成安全目标。

为了易于理解，我们将内核客体类别分解成四种了：文件相关的，网络相关的，System V IPC 和杂项。

4.3.1. 与文件相关的客体类别

第一个种客体类别是那些与文件及其他存储在文件系统中的资源有关的，这是大多数用户最熟悉的客体类别了，它包括了所有的与持续不变的，在磁盘上的文件系统和在内存中的文件系统，如 proc 和 sysfs 结合在一起的客体类别。

在类 UNIX 系统中，一个底层的概念是“一切皆是文件”，大多数情况下，这么说是对的，但是它隐藏了一个事实，就是不是所有的文件都是相同的。实际上，现代类 UNIX 系统，如 Linux 专门为设备，IPC 以及标准的存储数据的文件设计了专用的文件，SELinux 详细准确地表现了内核的视图，表 4-1 总结了与文件有关的客体类别。

表 4-1. 与文件有关的客体类别

客体类别	描述
blk_file	块文件
chr_file	字符文件
dir	目录
fd	文件描述符
fifo_file	命名管道
file	普通文件
filesystem	文件系统（如一个真实的分区）
lnk_file	符号链接
sock_file	UNIX 域套接字

客体类别 file 和 dir 分别代表普通文件和目录，普通文件就是那些存储数据的文件，它们是大多数系统上最常见的客体了，目录在 Linux 中也是一个特定的文件，它是独一无二的，因为它们可能还包含有其他客体。

lnk_file 客体类别代表符号链接，大多数情况下，它非常重要，它可以区别普通文件和符号链接，这样可以预防常见的攻击，恶意进程和用户可以创建符号链接，这样可能引起

某个进程访问或修改本不是它们打算要访问或修改的文件，独立的 `lnk_file` 客体类别允许编写预防这些攻击类型的策略。

客体类别 `fifo_file` 和 `sock_file` 表示用于 IPC 的特定文件，`fifo_file` 客体类别代表 `fifo` 文件，也叫做命名管道，`sock_file` 客体类别联合 UNIX 域套接字控制创建、访问等与文件有关的客体的能力，我们讨论 UNIX 域套接字客体类别及它们与下一节将要谈到的套接字文件的关系。

在 Linux 中，设备通常是通过在 `/dev/` 目录下的特定文件来进行访问的，这些文件通过主/次设备号表示块和字符设备。字符设备是程序以字节流形式读或写入数据的设备，块设备是将数据以更大块进行传递的设备，`chr_file` 和 `blk_file` 客体类别分别表示字符设备和块设备。

最后两个客体类别是文件系统和文件描述符，它们不是典型的 Linux 客体，`filesystem` 客体类别表示挂载的文件系统，这个客体类别控制全局操作如挂载或查询限额，例如：使用 `filesystem` 客体类别，我们可以只运行挂载支持存储安全上下文的文件系统。所有特定类型的文件系统（如 `ext3`）在策略中都使用相同的 `fs_use` 语句获取默认的标记定义，第 10 章“客体标记”中将会描述 `fs_use`，挂载分区时如果使用了 `context mount` 选项，默认的类型可能会被覆盖掉，也将在第 10 章中讲述。

文件描述符表示打开的与文件有关的客体，存在于进程中，即使与文件有关的客体明显不同，它们表示内核数据结构，通常认为文件描述符是文件有关的客体的基础，的确，标准 Linux 访问控制不能单独在文件描述符上提供访问控制，这种策略忽略了文件描述符是可以在进程之间进行传递的资源的事实，通常，子进程会从父进程那里继承文件描述符，这个继承并不总是有利的，在许多 Linux 编程指南中都警告最好减少文件描述符继承，特别是后台进程，为了标识这个和其它问题，我们以 `fd` 客体类别为例，在 SELinux 中，它代表文件描述符，使用这个客体类别阻止文件描述符在进程间传递或继承就成为可能，值得注意的是，有权使用文件描述符并不意味着就可以访问与文件有关的客体，进程必须对这些文件也要有访问许可才行。

4.3.2. 与网络有关的客体类别

与网络有关的客体类别代表网络资源如网络接口、不同种类的套接字和主机。目前的客体类别足以允许对单个系统上的网络进行广泛的控制，另外还有增强，如标记网络数据包，表 4-2 总结了与网络和套接字有关的客体类别。

表 4-2. 与网络有关的客体类别

客体类别	描述
<code>association</code>	IPsec 安全联盟
<code>key_socket</code>	PF_KEY 协议家族的套接字，用于管理

	IPsec 中的密钥
netif	网络接口（如 eth0）
netlink_audit_socket	用于控制审核的 Netlink 套接字
netlink_dnrt_socket	用于控制 DECnet 路由的 Netlink 套接字
netlink_firewall_socket	用于创建用户空间防火墙过滤器的 Netlink 套接字
netlink_ip6fw_socket	用于创建用户空间防火墙过滤器的 Netlink 套接字
netlink_kobject_uevent_socket	用于在用户空间接收内核事件通知的 Netlink 套接字
netlink_nflog_socket	用于接收 Netfilter 日志消息的 Netlink 套接字
netlink_route_socket	用于控制和管理网络资源如路由表和 IP 地址的 Netlink 套接字
netlink_selinux_socket	用于接收策略载入通知，强制模式切换和清空 AVC 缓存的 Netlink 套接字
netlink_tcpdiag_socket	用于监视 TCP 连接的 Netlink 套接字
netlink_socket	所有其它的 Netlink 套接字
netlink_xfrm_socket	用于获取、管理和设置 IPsec 参数的 Netlink 套接字
node	代表一个 IP 地址或一段 IP 地址的主机
packet_socket	协议在用户空间执行的原始套接字
rawip_socket	既不是 TCP 也不是 UDP 的 IP 套接字
tcp_socket	TCP 套接字
udp_socket	UDP 套接字
unix_dgram_socket	本地机器上（unix 域）的 IPC 数据报套接字
unix_stream_socket	本地机器上（unix 域）的 IPC 流套接字

node, netif, packet_socket, rawip_socket, tcp_socket, udp_socket 和 socket 客体类别控制典型的网络访问，netif 客体类别代表网络接口，每个有名字的网络接口（如 eth0, eth1）都是通过 netif 客体类别的实例进行表现的，由 IP 地址或地址段进行标识的网络上的远程主机是通过 node 客体类别表现的，使用 node 客体类别，我们可以限制主机（通过 IP 地址）上的哪个进程可以使用网络，前面列出的不同种类的套接字客体类别代表协议套接字类型，成功发送或接收网络数据需要所有有关的 netif, node, socket 客体类别实例的许可。

标准网络套接字是由协议分配的（在调用 socket(2) 系统调用时确定的），不同的 socket 客体类别允许我们限制应用程序发送或接收数据包的类型，这对限制应用程序发送原始数据包的能力特别有用。客体类别 tcp_socket 和 udp_socket 分别代表 TCP 和 UDP 套接字，

rawip_socket 客体类别代表发送原始 IP 数据包的套接字，packet_socket 客体类别代表发送其它类型的原始数据包的套接字，所有其它的套接字都由 socket 客体类别表示。

使用 IP 安全 (IPsec) 的通讯拥有额外的资源，由客体类别 association 和 key_socket 表示，IPsec 安全联盟是为通讯提供安全服务的连接，association 客体类别代表 IPsec 联盟，IPsec 需要通过密钥管理 (PF_KEY) 套接字管理密钥，它通过 key_socket 客体类别进行表现。

Linux 系统上的本地通讯可以使用 unix 域套接字 (PF_UNIX) 实现，这些套接字通常用于本地 IPC，面向连接的套接字也叫做流套接字，通过 unix_stream_socket 客体类别进行表现，数据报套接字通过 unix_dgram_socket 表现，unix 域套接字可以与文件系统上的某个特定文件关联起来，让其它应用程序很容易就连接到套接字，这个文件通过 sock_file 客体类别表现，它是一个与文件有关的客体类别，前面已经说过了。

SELinux 中最后一组套接字是 Netlink 套接字，这些套接字最初是开发用于在 Linux 最后提供一个标准意义的网络配置，现在它们常常用于在内核与用户空间之间通讯，基于协议类型的不同有多种表示 Netlink 套接字的客体类别，常见的 netlink_socket 表示无特定客体类别的保留协议。

4.3.3. system V IPC 客体类别

与 IPC 有关的客体类别代表 System V IPC 资源（参考表 4-3），msgq 和 msg 客体类别分别代表消息队列和消息队列中的消息，sem 客体类别代表信号量，shm 客体类别代表共享内存段，注意对关于所有 System V IPC 资源的全局系统信息的访问是通过 system 类别上的许可进行控制的。

表 4-3. 与 IPC 有关的客体类别

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

4.3.4. 其它杂类客体类别

表 4-4 列出了大量剩下的客体类别，这些客体类别与其它种类不容易混合在一起。

capability 客体类别代表标准 Linux 访问控制模式下的进程权利，这个客体类别允许 SELinux 控制授给“root”进程的权利，这些权利包括否决任意访问控制（许可模式）和发送原始网络数据包，这个客体类别和它的许可允许控制每个进程是否可以使用标准 Linux 授予它的权利。

剩下的两个客体类别 security 和 system 分别代表对 SELinux 安全服务器的特定资源和系统的访问，它们是唯一的，这些客体类别每一个都只有一个实例，反映出只有一个安全服务器和系统。

表 4-4. 其它杂类客体类别

客体类别	描述
capability	Linux 中表示权利的特权
processes	SELinux 中的进程
security	内核中的 SELinux 安全服务器
system	整个系统

4. 4. 客体类别许可示例

为了更好地理解许可是如何控制对系统资源的访问的，下面我们进一步讨论一下两个客体类别许可：file 和 process。附录 C 中详细描述了每个客体类别的所有许可。

访问撤销

撤销前面授予的访问权限是创建有关灵活和动态的安全机制非常重要的部分，当策略发生变化或客体安全上下文发生变化时，就需要撤销之前授予的权限，例如：如果某个文件的安全上下文发生了变化，打开那个文件的进程可能不再有相同的访问权，这是由新的策略决定的，系统不得不撤销现有的与变化不一致的访问权。对于复杂的操作系统，确保在所有环境下访问权的撤销是一件很困难的任务。

SELinux 支持多种环境下的撤销，比标准 Linux 支持得要多得多，例如：每次对文件进行读和写时都会检查文件的访问权，因此，如果文件的安全的上下文发生了变化，在下一次读或写时访问权就会被撤销掉。

有很多时候访问权是不会被撤销的（例如：内存映像的文件访问和未完成的异步 I/O 请求），在 SELinux 中很可能还会对撤销支持进程增强，但不可能覆盖全部范围，部分是由于 unix 应用程序编程接口（API）的性质决定的，部分是由于社区反对对内核子系统做入侵性的改变，还有部分是由于任务本身的复杂性。

通常，你可以避免大多数撤销问题，通过设计系统不重新标记客体实现，SELinux 提供许可（relabelfrom 和 relabelto）来限制这个能力。

4.4.1. 文件客体类别许可

表 4-5 列出了 file 客体类别许可，大多数对所有与文件有关的客体类别的许可都是公用的，只有 execute_no_trans, entrypoint 和 execmod 是特定给 file 客体类别的（这些许可都加了星号*）。

文件客体类别有三类许可：直接映像到标准 Linux 访问控制许可的许可，标准 Linux 许可的扩展和 SELinux 特定的许可。

表 4-5. 与文件有关的客体类别许可

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

4.4.1.1. 标准 Linux 许可

许可 read, write 和 execute 基本上与标准 Linux 学科读，写和执行（即 r, w, x）类似，但与标准许可检查有些不同，在标准 Linux 中，访问权通常是在文件打开时进行检查，

在 SELinux 中，访问权是每次使用时都会检查，read 许可包括了读取整个文件的能力，它包括以一种随即方式访问文件的许可，write 许可包括写入文件的许可，包括附加。与 read 许可类似，write 许可包括了随即访问写，当一个文件被映像到内存中时，也会检查 read 和 write 许可，例如：mmap(2) 系统调用，或使用 mprotect(2) 系统调用保护现有映像被改变。

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

4.4.1.2. 标准 Linux 访问控制的扩展

SELinux 的一个好处就是它提供了额外的许可，可以更细粒度地进行控制。

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

查看和修改文件属性的能力，包括许可模式和属主信息分别由 getattr 和 setattr 许可控制，getattr 许可控制文件属性的读取（例如：使用 stat(2) 系统调用），setattr 许可控制文件属性的写入（例如：使用 chmod(2) 系统调用）。

可以通过 flock(2) 或 fnctl(2) 系统调用锁定文件，由 lock 许可进行控制，获取一个锁不再需要其它许可，尽管实际上你需要有 read，write 或 append 许可获得文件描述符传递给有关的锁定系统调用。

通常，只允许对文件进行附加（append）访问非常有用，如：日志文件不能被覆盖，这样可以阻止攻击者清除证据，SELinux 单独提供了 append 许可，它在文件打开时强制实施了 o_append 模式，允许域类型 append 许可而无 write 许可意味着那个域类型进程只能将数据追加到文件。

由于创建是单独用 create 许可进行控制的，创建和移除硬链接是用 link 和 unlink 许可控制的，在 Linux 中，文件是可以被引用为一个或多个名字的，这就叫做硬链接，硬链接不是文件的“真实”名字，一个文件的所有硬链接都只有一个等效的有效名，Linux 文件系统的这种语义包含了多种安全含义，解除一个文件的硬链接本质上是删除一个文件（注意不是删除真正的文件，而是硬链接的名字而已），同样，硬链接一个文件本质上是对该文件创建了一个新的名字而已，可以使用系统调用 rename(2) 改变硬链接的名字，这个系统调用是由

rename 许可进行控制的，所有三种与硬链接有关的许可在有影响的目录上都需要额外的许可才能成功完成。

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

4.4.1.3. SELinux 特定许可

对于文件而言有五种 SELinux 特定许可: relabelfrom, relabelto, execute_no_trans, enTEntryPoint 和 execmod。

relabelfrom 和 relabelto 许可控制域类型将文件从一个类型改为另一个类型的能力，为了使重新标记文件成功，域类型必须要有该文件客体当前类型的 relabelfrom 许可，并且还要有新类型的 relabelto 许可，注意这些许可不允许控制确切的许可对，域可以将它有 relabelfrom 许可的任何类型改为它有 relabelto 许可的任何类型，在重新标记时可以增加约束，正如你在第 7 章“约束”中看到的 validate_trans 规则那样，重新标记客体对于系统而言是一个潜在的安全危险，应该严加控制。

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

我们在第 2 章中的域转换小节已经讨论过 enTEntryPoint 许可了，它控制使用可执行文件允许域转换的能力，execute, execute_no_trans 和 enTEntryPoint 许可允许精确控制什么代码可以执行什么域类型，SELinux 控制各个程序域类型的能力是它能够提供强壮灵活的安全的主要原因。

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

4.4.2. 进程客体类别许可

表 4-6 列出了进程客体类别许可，与文件许可不同，许多进程许可没有直接对应到标准的 Linux 访问控制，因为传统的 Linux 没有将进程作为一个正式的客体对待。

表 4-6. 进程客体类别许可

许可	描述
dyntransi tion	允许进程动态地转移到新的上下文中
execheap	产生一个堆栈可执行体
execmem	产生一个匿名的映像或可写的私有文件映像可执行体
execstack	产生进程堆栈可执行体
fork	派生两个进程
getattr	通过/proc/[pid]/attr/目录获取进程的属性
getcap	获取这个进程允许的 Linux 能力
getpgid	获取进程的组进程 ID
getsched	获取进程的优先级
getsessio n	获取进程的会话 ID
noatsecur e	禁用清除安全模式环境，允许进程在 execve(2) 上禁用 glibc 的安全模式特性
ptrace	跟踪程序执行的父进程或子进程
rlimitnh	在 execve(2) 上继承进程资源限制
setcap	为进程设置允许的 Linux 能力
setcurren t	设置当前的进程上下文，当进程试图执行一个动态域转换时，这是第一个检查的能力
setexec	下一次调用 execve(2) 时覆盖默认的上下文
setfscrea te	允许进程设置由其创建的客体的上下文
setpgid	设置进程的组进程 ID
setrlimit	改变进程硬性资源限制
setsched	设置进程的优先级
share	允许与克隆的或派生的进程共享状态
siginh	在 execve(2) 上继承信号状态
sigkill	发送 sigkill 信号
sigchld	发送 sigchld 信号
signal	发送一个非 sigkill, sigstop 或 sigchld 的信号
signull	不发送信号测试另一个进程的存在性
sigstop	发送 sigstop 信号
transitio n	在 execve(2) 上转换到一个新的上下文

4. 4. 2. 1. 创建进程

fork 许可控制进程使用 fork(2) 系统调用的能力，这个系统调用创建一个进程的副本，只是进程标识符和资源利用数据有所不同，派生进程的安全上下文不能改变，通常，执行一

个新程序的第一步就是派生，控制进程派生的能力，限制它使用系统资源的能力，可以潜在地预防某些类型的拒绝服务攻击。

三个其它的许可在进程转换时控制状态的共享，share 许可控制进程状态的共享，如在一个 `execve(2)` 系统调用上的文件描述符和内存地址空间，`siginh` 许可控制信号状态的继承，包括任意挂起的信号，最后，`rlimitnh` 许可控制从父进程那里继承的资源限制。

4.4.2.2. 进程域类型转换

与第 2 章中描述的域转换一样，`transition` 许可控制进程通过 `execve(2)` 系统调用从一个域转到另一个域的能力，如果允许，域转换或明确地请求时可能会自动产生一个 `type_transition` 规则，请求明确的域转换的能力是由 `setexec` 许可控制的，这个请求会往 `proc` 文件系统中写一个特定的文件，这个过程在 `setexeccon(3)` 库函数中被抽象出来，为下一个请求 `execve(2)` 系统调用查看当前请求的转换的能力是由 `getattr` 许可控制的。

`noatsecure` 许可使内核在进行域转换时不设置 `glibc` 的安全模式，在安全模式下，`glibc` 清除进程环境，包括相当多的环境变量，如 `LD_PRELOAD`，如果不清除环境，源域可能会控制目标域的关键部分，当域转换进入更高特权域时，允许 `noatsecure` 许可是特别危险的。

`dyntransition` 许可与转换许可类似，但它是控制进程在任何时间改变域类型的能力，不仅仅是执行程序那一刻，这个许可比 `transition` 许可更危险，因为它允许起始域在新域中执行任意的代码，由于这个原因，`dyntransition` 许可只有在目标域是起始域的一个受限的子集时可以安全地使用，否则，想要理解域改变的保护就会失败，所有授予目标域的访问权对起始域都必须能够访问。

警告： 随意使用 `dyntransition` 许可改变进程域类型会破坏标记的性质，在 `SELinux` 中，标记意味着在一个运行的系统中，创建一个客体后，它的类型将会被改变，尽管存在受信任的操作系统组件偶然改变了客体的类型的可能，`SELinux` 还是会严格控制进程的类型改变，`dyntransition` 许可的引入打破了这种固有属性，使得所有的策略安全分析都变得极为复杂，我们强烈建议你永远都不要使用这个许可，除非你在编写用户空间客体管理器或其它 `SELinux` 扩展。

`dyntransition` 许可的 `setcurrent` 许可与 `transition` 许可的 `setexec` 许可类似，它控制请求改变进程域类型的能力，成功改变域类型需要 `dyntransition` 许可，另外还要 `setcurrent` 许可，和 `setexec` 类似，请求被写入到 `proc` 文件系统的特殊文件中，这个过程抽象在 `setcon(3)` 库函数中。

4.4.2.3. 创建文件

与域转换类似，与文件有关的客体的安全上下文设置可以通过继承或 `type_transition` 规则进行自动创建，也可以明确地创建，通过在 `proc` 文件系统中写入一个特定的文件实现

与文件有关的客体的安全上下文的设置，这个过程抽象在 `setfscreatecon(3)` 库调用中，`setfscreate` 许可控制产生这个明确请求的能力，与 `setexeccon` 类似，查看文件系统客体上下文请求的当前状态是由 `getattr` 许可控制的。

4.4.2.4. 进程信号

向进程发送信号的权力非常大，因为它可能允许结束或停止进程，此外，信号可以用于在进程间传输信息，`sigchld`，`sigkill` 和 `sigstop` 许可分别控制发送 `SIGCHLD`，`SIGKILL` 和 `SIGSTOP` 信号的能力，`signull` 许可控制发送空信号的能力，例如：通过传递一个 0 字符作为一个信号参数给 `kill(2)` 系统调用，最后，`signal` 许可控制发送其它信号的能力。

为什么有的信号有明确的许可定义，而剩下的都在常见的 `signal` 许可控制之下呢？有两个原因，`SIGKILL` 和 `SIGSTOP` 这两个信号有明确的许可，因为它们不能由进程阻碍，`SIGCHLD` 信号有它自己的许可主要是因为它被正式使用的（如它经常是每个进程 `init` 时使用），剩下的安全属性都相同，因此它们都由 `signull` 许可控制。

4.4.2.5. 进程属性

查询或设置调度优先级以及进程策略的能力是由 `getsched` 和 `setched` 许可控制的，设置调度优先级和策略，特别是设置 `SCHED_FIFO` 策略，使用 `sched_setscheduler(2)` 系统调用可以允许进程不受限制占用 CPU 时间，因此，它可以用于拒绝服务攻击。

进程组和会话标识符控制大部分进程的交互，包括终端处理和信号传递，`getpgid` 和 `setpgid` 许可控制查询和设置进程组标识符，`getsession` 许可控制进程标识符的查询。

`getcap` 和 `setcap` 许可控制查询和设置进程的 Linux 许可，要成功设置一个许可，这个许可也必须被标记了域类型的 `capability` 客体类别接受。

资源限制，如核心转储的最大大小或 CPU 时间的最大大小，都是使用 `setrlimit(2)` 系统调用，`setrlimit` 许可控制设置硬性资源限制的能力。

4.4.2.6. 执行可写入内存

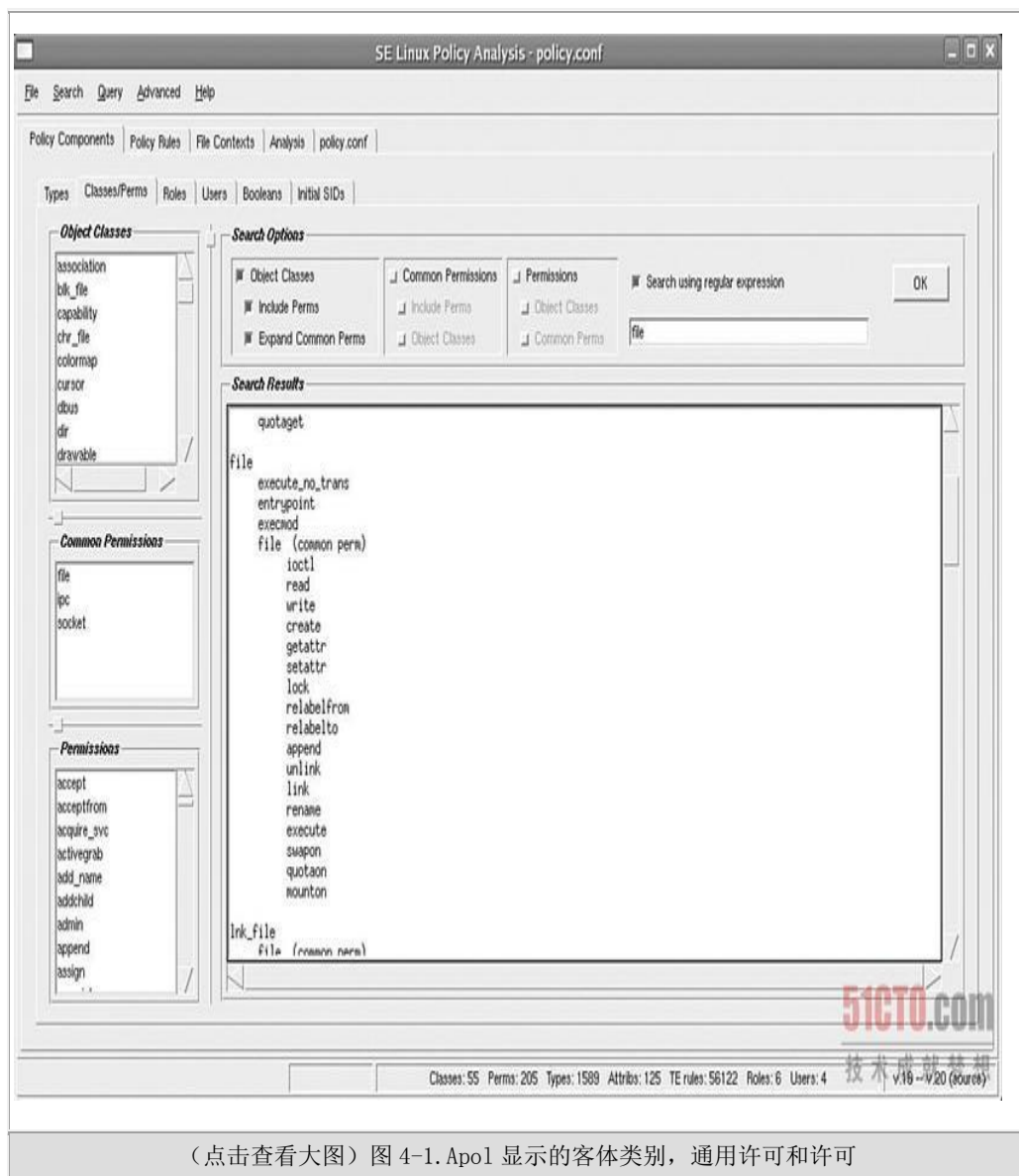
正如在 `file` 客体类别的 `execmod` 许可中讨论的那样，执行可写入内存段的能力是许多安全事件的起源，为了帮助标记出这些事件，首先创建 `execmem`，`execstack` 和 `execheap` 许可，它们分别控制可执行的匿名映像、堆栈和堆的创建，许可的执行依赖于另外的软件，如 `ExecShied`，硬件特性，如 `NX`。

`ExecShied` 是 Red Hat 开发的内核补丁，控制内存执行，并添加了其它安全特性，它包括在所有的 Fedora Core 和自 Red Hat Enterprise Linux 3 以来的版本中。

`NX` 是一个硬件设置，它实现了许多 `ExecShied` 相同的目标。

4. 5. 使用 Apol 研究客体类别

Apol 为浏览和查询客体类别和许可提供了大量的特性，在策略组件(Policy Components)标签下是类别/许可 (Classes/Perms) 标签，它可以浏览和搜索所有的客体类别，通用许可和唯一性许可。图 4-1 显示了 apol 的这个标签，在左边是所有客体类别，通用许可和许可，右边是一个查询结构，让你可以搜索客体类别或许可。



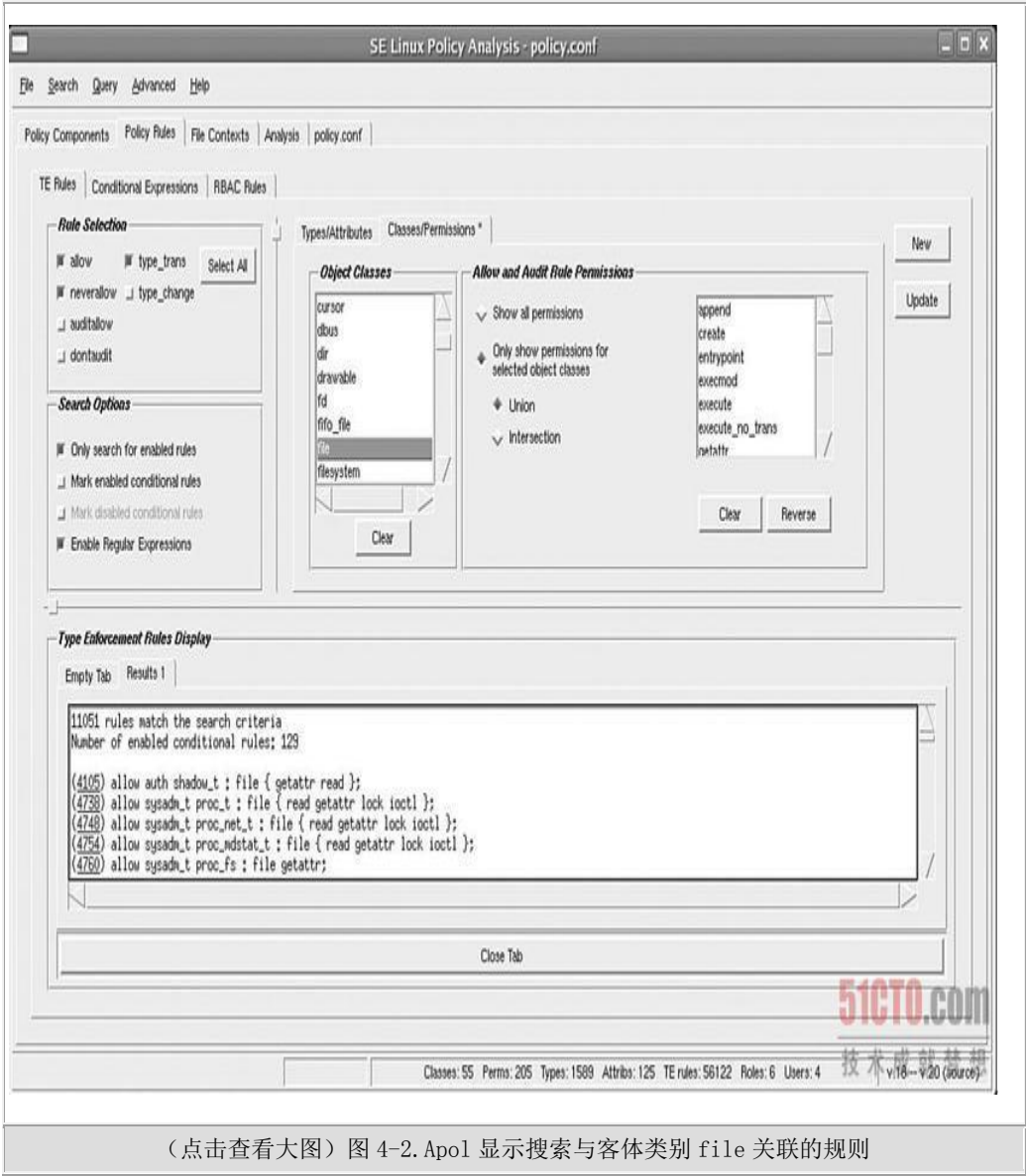
(点击查看大图) 图 4-1. Apol 显示的客体类别，通用许可和许可

双击列表中的一个策略组件，在左边就会显示关于它的详细信息，例如：双击一个客体类别，就会显示它的访问向量，双击一个许可，就会显示所有与其关联的客体类别。

搜索接口让你可以使用正则表达式搜索客体类别或许可，例如：在图 4-1 中，我们在所有客体类别中搜索名字中包括有“file”的客体类别，我们没有设置选项在结果中包括特定类别许可或扩展通用许可，但正如你在搜索结果 (Search Results) 窗口中看到的，apol 显

示了客体类别 file 包括特定类别许可和扩展的通用许可，这是一个收集与客体类别关联的完整许可列表的好方法。

apol 中许多与客体类别交互的其它特性包括规则搜索和自动分析，它允许基于客体类别和许可过滤结果，例如：图 4-2 显示了一个涉及到客体类别 file 的规则搜索。



(点击查看大图) 图 4-2. Apol 显示搜索与客体类别 file 关联的规则

4. 6. 小结

客体类别和许可是 SELinux 中访问控制的基础，它们是策略语言和内核中访问实施机制的一部分。

客体类别代表资源，如文件、进程、目录和套接字，每种系统资源都对应着一个客体类别。

许可代表对系统资源的访问权，每个客体类别都有一套定义好的许可，叫做访问向量。

客体类别使用类别声明语句（class）进行声明。

使用访问向量语句（也是 class）将许可与客体类别进行关联。

SELinux 中定义了两类许可：通用许可和特定类别许可。

通用许可是一套由多个客体类别共享的许可集，使用访问向量语句将它们与客体类别作为一个组进行关联。

SELinux 提供客体类别和许可精确地覆盖了所有的系统资源，在 FC 4 中，有超过 40 个客体类别，反映了 Linux 的丰富和复杂。

理解所有的客体类别和许可需要深入理解 SELinux 和 Linux。

在 Linux 中允许访问完成多个任务需要一个或多个客体类别的多个许可。

附录 C 列出了所有的客体类别和许可的参考。

练习

1、创建一个具有 read, write, bind, connect 和 listen 许可名叫 socket 的通用许可集。

2、将通用许可 socket 和特定类别许可 connecto, acceptfrom 与问题 2 中声明的客体类别关联起来。

3、编写一个 allow 规则允许域 httpd_t 将类型 httpd_log_t 追加到某个文件，但不能写入。

4、编写必要的 allow 规则允许域 httpd_t 执行 bin_t 类型的文件，包括请求一个明确的域转换的能力，但不是无转换执行的能力，假设该规则提供的转换和入口点在策略中已经存在。

第 5 章-类型强制

SELinux 策略大部分内容都是由多条类型强制规则构成的，这些规则控制被允许的使用权，大多数默认转换标志，审核，以及固定部分的检查。在这一章中，我们将详细地讨论类型强制规则以及这些规则使用的类型定义和声明。

5.1. 类型强制

SELinux 策略大部分都是一套声明和规则一起定义的类型强制（TE）策略，一个定义良好、严格的 TE 策略可能包括上千个 TE 规则，TE 规则数量的巨大并不令人惊奇，因为它们表达了所有由内核暴露出的允许对资源的访问权，这就意味着每个进程对每个资源的访问尝试都必须至少要有一条允许的 TE 访问规则，如果我们仔细思考一下现代 Linux 操作系统中

进程和资源数量，就明白为什么在策略中有那么多的 TE 规则了。当我们添加由 TE 规则控制的审核配置和标志时，对于具有严格限制的 SELinux 策略，常常会见到它包含有上千条规则，在第三部分中，我们将会讨论如何创建和管理这些大量的规则，现在，我们先了解一下 TE 规则是如何工作的。

TE 规则的绝对数量对理解 SELinux 策略是一个大的挑战，但是规则本身并不复杂，它们的分类相对较少，所有的规则基本上都属于两类范畴：访问向量（AV）和类型规则。我们使用 AV 规则允许或审核两个类型之间的访问权，我们在某些情况下使用类型规则控制默认的标志决定。

正如它们的名字暗示的意思，TE 规则通过安全上下文与所有资源联合一起对类型起作用，策略语言包括了另外的允许我们定义类型及其策略组件的语句。

SELinux 的一个重要概念是 TE 规则是将权限与程序的访问结合在一起，而不是结合用户。这一章我们讨论的所有 SELinux 策略语言特性都是处理主体（正常的运行中的进程）对客体（文件、目录和套接字等）的访问权的，主要集中于程序访问控制决策，这也是 SELinux 的主要益处，它允许 SELinux 策略编写者基于程序的功能和安全属性，加上用户要完成任务需要的所有访问权做出访问决策，可以将程序限制到功能合适，权限最小化的程度，因此，即使它出了故障或被攻击破坏，但整个系统的安全并不会受到威胁。例如：如果一个 Web 服务器的策略阻止修改它显示的文件，那么即使服务器被攻破了，TE 策略也能阻止被攻破的服务器修改那些文件。这样就消除了通过 Web 服务器的漏洞攻击造成对网站的威胁。只有被攻破的应用程序受到影响，并且它会被我们在策略中定义的访问权限制。

SELinux 是不会管用户的，可以给同一个程序指定多个域类型（因此有不同的特权集），这样就允许引入角色的概念，我们将在第 6 章“角色和用户”中讨论有关角色的话题，尽管如此，访问控制的标准仍然是基于程序的域类型而不是用户的特权。

将焦点从用户转移背后蕴含的意义很难理解，这样我们就不得不重新讨论计算机安全领域的基本原理，好处是很明显的，但它也要求我们思考这种新的访问控制方式，当你读完这一章时，你将会记住焦点是在程序访问权了。另外，如果你还没有完全掌握类型强制的概念，在继续阅读下面的内容前，你应该回顾一下第 2 章“概念”的内容。在这一章中，我们提供足够详细的信息使你能够编写 TE 规则，所以我们假设你已经完全理解了第 2 章中讨论的 SELinux 和类型强制的基本概念。

5.2. 类型、属性和别名

正如你从术语类型强制猜测的那样，类型是构成 TE 规则的最小单位，SELinux 主要就是使用类型来确定什么访问是被允许的；属性和别名是为减轻管理和使用类型的策略特性，我们使用属性利用单个标识符来引用一组类型。通常，策略语言允许我们在 TE 规则中类型

的适当位置使用属性，而别名允许我们为类型定义另一个名字，别名标识符和类型标识符做同等地位对待。

5.2.1. 类型声明

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

```
type httpd_t;  
type http_user_content_t;
```

声明了类型后就可以在安全上下文、TE 规则和其它策略语句中使用它们了，你可以在 92 页查看完整的类型声明语法。

类型声明语法

可以使用 `type` 语句声明类型和选项，以及关联的别名属性，下面是完整的类型声明语法：

`type 类型名称 [alias 别名集] [, 属性集];`

类型名称 类型标识符，长度任意，允许包括 ASCII 字符，数字，下划线 (`_`) 和点 (`.`)。

别名集 一个或多个别名标识符，别名标识符与类型标识符的命名限制是一致的，如果指定的不止一个别名标识符，要在一对大括号中用空格将各个别名区别开来，如：`alias {aliasa_t aliasb_t}`。

属性集 一个或多个预先声明的属性标识符，如果同时指定多个属性标识符，属性之间使用逗号进行分隔，如：`type bin_t, file_type, exec_type;`。

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

5.2.2. 类型和属性

可能你已经想到，一个大型的，复杂的策略可能包括上万个代表系统上不同资源的类型，例如：Fedora Core 4 (FC 4) 的 `targeted` 策略相对较小，但也声明了超过 800 个类型。由于默认的规则是拒绝所有的访问，所以任何一个访问都需要明确地被允许，这就导致了类型注定会很冗长，这时策略语言的属性特性就派上用场了。属性可以理解为：1) 类型的性质或属性，或二者兼得，2) 一组类型。在任何一种情况下，原理都是相同的。

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

```
type backup_t;

allow backup_t httpd_user_content_t : file read;
allow backup_t shadow_t : file read;
```

这里指定了域类型 `backup_t` 可以访问两个文件类型：前面声明的 `http_user_content_t` 和 `shadow_t` 类型，`shadow_t` 指的是 `/etc/shadow` 文件的类型，两者都是备份程序必须可读的磁盘文件。

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

使用 `attribute` 语句进行属性声明，如：

```
attribute file_type;
```

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

```
allow backup_t file_type :file read;
```

注意：尽管在所有的类型名称后面附加一个 `_t` 很常见，但对于属性名称而言，却没有这样通用的约定，因为类型和属性共享同一个命名空间，这样在编写和检验 TE 规则就能很容易地识别出谁是类型，谁是属性了。

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

下面列出了完整的属性声明的语法：

属性声明语法

使用 `attribute` 语句进行属性声明，完整的属性声明语法如下：

`attribute` 属性名称；

属性名称 属性的标识符，它的长度不受限制，可以包括 ASCII 字符，数字，下划线（`_`）和点（`.`）。属性和类型，别名都在同一个命名空间，因

此不能与其他类型或别名重名。

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

5.2.3. 关联类型和属性

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

```
type http_user_content_t, file_type;
```

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

由于 `http_user_content_t` 代表了 Web 服务器要使用的所有文件，而属性表示可以将它们一致使用，在这个例子中，我们创建的属性叫做 `file_type`，它代表所有永久存储的文件，因此，我们就可以只写一条规则来访问所有文件了，再也不用为每个文件编写一条规则了。

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

```
type httpd_user_content_t, file_type, httpdcontent;
type shadow_t, file_type;

allow backup_t file_type : file read;
allow httpd_t httpdcontent : file read;
```

现在我们已经给 `httpd_user_content_t` 添加了两个属性，`file_type`（表明这是一个在磁盘上的文件的类型）和 `httpdcontent`（表明这个类型 Web 服务器是可读的）。对于有更多特权的 `shadow_t` 类型，我们只关联了 `file_type` 属性（因为对于一个 Web 服务器而言，要是能够显示 `shadow` 密码文件，看起来可能不是一个好主意），我们也使用了两条 `allow` 规则为 Web 服务器和备份程序授予了需要的访问权，结果就是 Web 服务器（`httpd_t`）可以访问所有具有 `httpcontent` 属性的文件，但不能访问其他文件，如具有 `shadow_t` 类型的文件，换句话说，备份程序（`backup_t`）可以访问所有具有 `file_type` 属性的文件。

类型具有的属性数量没有限制，就和类型一样，我们可以合理定义相应的属性。

注意：在写本书时，我们在代码中对类型和属性的数量限制到 232 个标识符，这也是 Red Hat Enterprise Linux 版本 4 (RHEL 4) 支持的大小限制，本书发行时，其大小可能被改为 216 个标识符（由于要优化 SELinux 对内存的利用率），其实在实际环境中，我们能够定义的类型数量也就几千个，因为数量太大时，与之关联的 TE 规则可能就会变得非常笨重，难以控制，因此，我们到目前为止看到的最复杂的策略，其中也没有超过 2000 个类型和属性声明。

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

```
type httpd_user_content_t, file_type, httpdcontent;
```

分成两条语句进行表述

```
#下面是两条语句
type httpd_user_content_t;
typeattribute httpd_user_content_t file_type,
httpdcontent;
```

实际上，这两条语句的作用等于前面的那个单条语句。

提示：我们首次在策略语句中使用了注释，对于策略编译器，`#`符号表示注释，在编译时，会忽略它后面的字符。

仅从这个例子还看不出为什么需要 `typeattribute` 语句，但你阅读到后面的章节时，你会发现，使用它可以使语句变得更加清晰易读，从根本上说，这个语句允许我们在一个地方定义类型，在另一个地方关联属性，增强了语言的灵活性，在设计策略源文件时，可以考虑进行模块化设计了。

警告：属性是策略语言很方便的一个特性，但它也很危险，将属性和类型关联后，可能会扩大对类型的访问权，这个访问权可能是也可能不是恰当的，主要依赖于我们的安全目标。例如：将一个域类型关联上一个属性后，可能扩大了类型的访问权，而你可能不会完全感受到，这就跟授予一个进程强大的特权类似，你应该确定属性的访问权对于类型是恰当的，并注意 TE 规则引用属性时的影响。

下面是 `typeattribute` 语句的完整语法：

typeattribute 语句语法

`typeattribute` 语句允许你关联前面声明的类型和属性，在类型声明时，如果没有关联属性，就可以使用这个语句进行类型和属性的关联，`typeattribute`

语句完整的语法如下：

`typeattribute` 类型名 属性名；
类型名 添加到属性上的类型的名称，类型名必须事先使用 `type` 语句进行声明，而且这里只能出现一个类型名。
属性名 一个或多个事先声明的属性标识符，如果指出多个属性标识符，属性标识符之间使用逗号分隔，如 `typeattribute bin_t file_type, exec_type`；
`Typeattribute` 语句在单个策略，基础载入模块和非基础载入模块中都是有效的，只有在条件语句中无效。

5.2.4. 别名

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

有两种方式进行别名声明，第一种方式是在使用 `type` 语句声明类型的同时就声明别名，因此在 `type` 语句中，声明 `mozilla_t` 类型时，可以使用关键字 `alias` 声明一个别名 `netscape_t`。如：

```
type mozilla_t alias netscape_t, domain;
```

注意别名声明是放在属性的前面的。

我们也可以使用 `typealias` 语句独立于 `type` 语句单独声明别名，下面的语句等同于单条 `type` 语句：

```
# 这两条语句等同于
type mozilla_t, domain;
typealias mozilla_t alias netscape_t;

#下面这一条语句
type mozilla_t alias netscape_t, domain;
```

当策略的结构难以在声明类型时同时声明别名时 `typealias` 语句很有用，下面给出 `typealias` 语句的完整语法介绍：

typealias 语句语法

`typealias` 语句允许你定义一个类型别名，在使用 `type` 语句声明类型时就可以同时声明别名，完整的 `typealias` 语句语法如下：

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

别名名称	一个或多个别名标识符，它的命名约束与类型一样。如果同时指定多个别名，别名之间用空格分开，并使用大括号将所有别名括起来，如{aliasa_t aliasb_t}。 typealias 语句在单个策略，基础载入模块和非基础载入模块中都有效，只有在条件语句中无效。
域类型和其他类型	<p>在第 2 章中，你看到的类型有时叫做“域类型”，而在本书中，我们将在“类型”这个次之前加上其他形容词，如“文件类型”，“目录类型”，这些形容词只是为了描述出类型的用途方向，并没有其他含义，例如文件类型指的是安全上下文中引用的类型是与文件有关，事实上，类型也可以用于其他客体类别，在策略语言中对这些类型（不管是文件类型还是域类型）的用途并没有做限制，SELinux 中的所有类型地位相同，都可以用于标记任何客体类别实例，只要授予了适当的访问权。</p> <p>例如 httpd_t 域类型可以同时用于进程和文件，在加上一些额外的规则，照惯例，这在 SELinux 策略中是要避免出现的，主要是为了使策略更清晰，但在某些情况下，我们需要将同一个类型即用作域类型又要用作文件类型，这完全取决于策略编写者的爱好了。</p> <p>然而，就域类型而言，出于一些技术上的原因，也最好不要在文件和目录上使用它们，在 Linux 中，每个进程和文件都会在 /proc/ 文件系统中由内核创建一个，这些客体用于获取和设置这些进程的属性，在 SELinux 中，进程的类型自动用于这些文件和目录，对于类型为 httpd_t 的进程而言，如果进程 ID 是 1000，目录 /proc/1000/ 和它下面包括的所有文件及子目录都自动具有 httpd_t 类型，如果 httpd_t 也用于普通文件，这就意味着授予其他普通文件访问权时，可能同时也授予了 /proc/ 目录下的文件和目录同等的访问权，这可能不是所希望的。</p>

5.3. 访问向量规则

AV 规则就是按照对客体类别的访问许可[1]指定其具体含义的规则，SELinux 策略语言目前支持四类 AV 规则：

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

allow	表示允许主体对客体执行允许的操作。
dontaudit	表示不记录违反规则的决策信息，且违反规则不影响运行。
auditallow	表示允许操作并记录访问决策信息。
neverallow	表示不允许主体对客体执行指定的操作。

本小节剩余部分将详细讨论这些规则的语法和语义，以及一些示例。

5.3.1. 通用 AV 规则语法

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

规则名称	allow, dontaudit, auditallow 和 neverallow。
源类型	授予访问的类型，通常是进程尝试访问的域类型。
目标类型	客体的类型，它被授权可以访问源类型。
客体类别	客体的类别。
许可	表示主体对客体访问时允许的操作类型（也叫做访问向量）。

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

```
allow user_t bin_t : file execute;
```

这个 allow 规则的源类型为 user_t, 目标类型为 bin_t, 客体类别 file, 许可 execute, 这个规则可以解读为“允许 user_t 执行类型为 bin_t 的文件”。

这四个 AV 规则语法都一样，只是关键字不同，如我们可以将前面这个例子替换成 auditallow 规则，只需要替换掉规则名称即可：

```
auditallow user_t bin_t : file execute;
```

我们将在后面讨论这个规则的含义，目前重要的是理解它们的语法。

5.3.1.1. AV 规则的密钥

在内核中，所有的 AV 规则都是通过一个组源类型、目标类型和许可进行唯一性标识，这个三重组叫做一个密钥，当做哈希表使用，缓存在策略数据结构中，回忆一下第 3 章“架构”中的内容，指出了规则是靠这个密钥存储和检索的，当一个进程产生了一个访问请求时，SELinux LSM 模块被要求允许基于这个密钥进行访问。

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

```
allow user_t bin_t : file execute;
allow user_t bin_t : file read;
```

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

警告：策略中的每个子级 AV 规则与其上级 AV 规则具有相同的密钥，并将许可 adds 进最高级规则编译进策略中，不存在 removing 授予给其他角色的许可的概念，因此要当心，尽管你在策略中的某部分编写了良好的规则，但可能在策略的其他地方会授予额外的许可。

5.3.1.2. 使用 AV 规则中的属性

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

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

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

```
allow user_t exec_type : file execute;
```

与前面的例子有点不同，这里的规则没有反应出什么将被内核执行，包括属性的规则将在内核中进行扩展，与属性关联的每个类型都有一个独立的密钥，如果有 20 个文件类型与 exec_type 属性关联，内核 AVC 可能在 20 个密钥结束，每个都授予对类型为 user_t 的 file 客体类别 execute 访问权。

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

```
allow domain exec_type : file execute;
```

为了更好地解释规则扩展的原理，假设我们的策略关联了类型为 user_t 和 staff_t 的属性 domain，以及文件类型为 bin_t, local_bin_t 和/sbin_t 的属性 exec_type，那么上面那一条规则的效果就等同于下面这些规则：

```
allow user_t bin_t : file execute;
allow user_t local_bin_t : file execute;
allow user_t/sbin_t : file execute;
allow staff_t bin_t : file execute;
allow staff_t local_bin_t : file execute;
```

```
allow staff_t sbin_t : file execute;
```

5.3.1.3. AV 规则中的多类型和属性

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

```
allow user_t { bin_t sbin_t } : file execute;
```

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

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

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

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

5.3.1.4. 特殊类型 self

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

```
# 这两条规则是相等的  
allow user_t user_t : process signal;  
allow user_t self : process signal;
```

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

下面来看一个稍微复杂一点的例子：

```
allow {user_t staff_t} self : process signal;
```

在这个例子中，实际上创建了两条规则，每条规则的源类型和目标类型都是相同的，它完全等同于下面这两条规则：

```
# 这两条规则  
allow user_t user_t : process signal;
```

```
allow staff_t staff_t : process signal;

#等于下面这一条规则
allow {user_t staff_t} self : process signal;
```

注意在使用 self 时，每个类型都创建了等同的规则，需要指出的是 user_t 不能访问 staff_t，反之亦然。

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

当 AV 规则的源类型为属性或一大串类型时，self 作为目标类型显得非常有用，例如：假设我们想让每一个域都可以向它自己发送信号，那么可以编写如下规则：

```
allow domain domain : process signal; # 这不是我们真的想要的
```

尽管这条规则实现了预期的目标（每个域都可以向它自己发送信号），但也将允许每个域类型向其它域类型发送信号，这个非预期的结果可能成为一个安全灾难，通过使用 self 关键字，可以确保每个域类型都只能访问它自身，如：

```
allow domain self : process signal; # 这才是我们想要的
```

5.3.1.5. “非”特殊操作符

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

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

这个规则在展开时就好像属性 exec_type 没有包括类型 sbin_t 一样。

类型否定不依赖顺序，即使要排除的类型位于属性列表的前面，例如下面的语句在语义上与前面的例子是等同的：

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

5.3.1.6. 在 AV 规则中指定客体类别和许可

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

```
allow user_t bin_t : { file dir } { read getattr };
```

这条规则将会产生两个密钥，每个客体类别一个，这条规则等同于下面这两条规则：

```
allow user_t bin_t : file { read getattr };
allow user_t bin_t : dir { read getattr };
```

注意客体类别被展开了，但每条规则都有相同的许可列表，这意味着列表中的所有许可对所有客体类别都是有效的，有时我们不得不创建两个有相同的源和目标类型、但客体类别不一样的规则，因为许可列表不是对每个类别都有效的，例如：如果我们查看一下 file 和 dir 客体类别的许可，你会发现它们大多数都是相同的，但也有部分是不同的（许可对两者都有效说明是使用了通用许可的结果）。

例如：假设我们想编写一条规则允许对这两个客体类别都能够“read”，下面的规则就是无效的了：

```
# 无效的规则，因为 search 对于客体类别 file 是无效的
allow user_t bin_t : { file dir } { read getattr search };
```

即使 read 和 getattr 对于 file 和 dir 客体类别都是通用许可，但 search 许可只对 dir 客体类别有效，因为给 file 类别提供一个无效的许可（search），checkpolicy 不能为其创建密钥，在编译策略时就会报错，唯一的办法就是创建两条规则，如：

```
#当许可对两个客体类别不是都有效时，需要两条规则
allow user_t bin_t : file { read getattr };
allow user_t bin_t : dir { read getattr search } ;
```

5.3.1.7. AV 规则中的特殊许可操作符

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

```
allow user_t bin_t : { file dir } *;
```

这条规则扩大后将包括 file 和 dir 的所有许可。

通配符语法与列出所有的许可有点不同，使用通配符时，许可包括每个客体类别的许可，此时不会考虑其中一个许可是否对另一个客体类别是否有效，这样就可以在规则中使用多个

客体类别，即使这些客体类别有不同的许可，因此，上面这条规则就安全地处理了许可，不会像前面那条规则那样，这里只对 dir 客体类别有效的规则不会影响到 file 客体类别。

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

```
allow user_t bin_t : file ~{ write setattr ioctl };
```

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

警告：建议正确地使用这些特殊操作符（非，通配符和求补算操作符），过去几年，它们也在不断向前发展，并且也发生了一些变化，checkpolicy 版本不同，对这些特殊操作符的使用也不同，如 RHEL 4 就可以在类型上使用通配符。

最近对编译器的改进又严格限制了这些操作符的使用，主要的例外是允许在 neverallow 规则中使用通配符匹配所有类型，但其它 TE 规则中不能使用，通常，如果你像本章介绍的这样使用这些操作符是安全的。

通用访问向量规则语法

完整的 AV 规则通用语法如下：

规则名称 类型集 类型集：类别集 许可集

规则名称 访问向量规则的名称，有效的规则名称是 allow, auditallow, auditdeny, dontaudit 和 neverallow。

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

类别集 一个或多个客体类别，多个客体类别必须使用大括号括起来，如 {file lnk_file}。

许可集 一个或多个许可，所有许可对类别集列出的所有客体类别都要有效，多个许可必须用大括号括起来，如 {read create}。通配符（*）指出所有客体类别的所有许可，求补算操作符（~）用于指出所有的许可，除了明确列出的之外。

所有 AV 规则在单个策略，基础载入模块和非基础载入模块中都有效，所有 AV 规则除了 auditdeny, neverallow 规则外，其它的在条件语句中也有效。

5.3.2. 允许（allow）规则

到目前为止，你已经看到了许多的 allow 规则，allow 规则是策略中最常见的规则，它实现了 SELinux 策略的主要目的（即允许访问）。

正如前面讨论的，我们使用 allow 规则指出了所有运行时授予的许可，它们是 SELinux 策略中允许许可的唯一方法，记住，默认情况下，不允许任何访问，我们指定了两个类型列表（源和目标类型），根据列出的客体类别的许可指定访问权，如：

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

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

如果这个例子策略中只有这条 allow 规则，没有其它的访问权授予类型为 bin_t 的文件，那么 user_t 将不能往类型为 bin_t 的文件进行写入操作。

与所有 AV 规则类型，allow 规则是累加的，对于一个主体-目标-类别密钥允许的访问许可是联合了所有的 allow 规则的并集，例如下面这两套规则是等同的：

```
# 这两条规则.
allow user_t bin_t : file read;
allow user_t bin_t : file write;
# 等同于这一条规则
allow user_t bin_t : file { read write };
```

5.3.3 审核（audit）规则

SELinux 有大量的工具记录日志信息，或审核、访问尝试被策略允许或拒绝的信息。审核消息通常叫做“AVC 消息”，它提供了详细了关于访问尝试的信息，包括是允许还是拒绝，源和目标的安全上下文，以及其它一些访问尝试涉及到资源信息。AVC 消息与其它内核消息类似，都是存储在/var/log 目录下的日志文件中，它是策略开发、系统管理和系统监视不可缺少的工具。在这一章中，我们检查是哪一个访问尝试产生了审核消息，第三部分提供了更多关于使用审核消息进调试和理解策略的内容。

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

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

警告：仅当拒绝是由 SELinux 拒绝时，才会进行审核，回顾一下第 3 章中 LSM 模块钩子功能，只有当这些访问传递给标准 Linux 任意访问控制检查时才会调用它。这就意味着如果一个访问是由标准 Linux 访问检查给拒绝的，SELinux 就不会有任何动作，更不会记录任何审核消息，如果你想审核任何拒绝访问的消息，而不管它为什么被拒绝，你只有直接使用 Linux 2.6.x 系列内核的审核系统，参考 `man auditd(8)` 和 `man auditctl(8)`。

例如：假设下面这个规则

```
dontaudit httpd_t etc_t : dir search;
```

指出类型为 `httpd_t` 的进程在类型为 `etc_t` 的目录上进行 `search` 时被拒绝，这个拒绝就不应该被审核，不要理睬默认的行为，你会发现 Linux/Unix 通常会显示出这种行为，即它们不需要这些东西也能正常工作，但它们就是要去访问。

`dontaudit` 规则在我们想屏蔽掉预期的审核拒绝消息时非常有用，这个规则允许我们避免授予不需要的访问权，也不会有大量预期的审核消息填充到日志文件中，正如我们前面所谈到的，这种行为太常见了。

auditdeny 规则

早期的 SELinux 版本支持 `auditdeny` 规则，它的用途与 `dontaudit` 类似，即便是策略语言还支持这个规则，也很少使用它，我们也建议你不要使用这个规则。`dontaudit` 规则和它默认的记录审核所有拒绝的消息的行为才是我们想要的。

录允许的访问，一起来看一下下面这条规则：

```
auditallow domain shadow_t : file write;
```

这条规则指出当一个具有 `domain` 属性的类型的进程成功地获得了类型为 `shadow_t` 的文件的 `write` 访问许可，允许的访问被审核了，`auditallow` 规则在审核一个重要的安全事件时非常有用，如这个例子中的对 `shadow` 密码文件的写操作，或将一个新的策略载入内核。

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

注意：在许可模式和强制模式下审核是不一样的。运行在强制模式下时，每次允许或拒绝时都会进行审核，应该在策略中对审核频率进行限制（可以使用 `auditctl` 实现）。在许

可模式下时，只会记录第一次访问尝试，直到下一次策略载入，或固定为强制模式，在开发时通常使用的就是许可模式，这种模式可以减少日志文件的大小。

5.3.4. neverallow 规则

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

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

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

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

```
neverallow * domain : dir ~{ read getattr };
```

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

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

另一个常见的 neverallow 规则是：

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

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

载入式模块依赖性处理

载入式策略模块是 Fedora Core 5 (FC 5) 的一个新特性，它包括处理模块之间依赖性的语言特性，依赖性处理特性使得在安装策略组件时可以解决掉依赖问题，参考第 13 章“管理 SELinux 系统”了解更多关于如何安装和管理载入式策略模块的信息，可能的策略组件依赖包括客体类别，许可，用户，角色，类型或别名，属性和布尔标识符。

require 语句指出了载入式模块需要的策略组件，所有不是在模块中声明的策略组件都必须以某种形式进行申请，如下面的 require 语句：

```
require { type etc_t; }
```

上面这个例子说明了载入式模块需要类型 etc_t，etc_t 应该在策略的其它位置进行声明（即在基础模块或在其它载入式模块中声明），这个 require 语句允许类型 etc_t 显示在模块内的策略规则中，而不用在模块内进行声明，下面的例子显示了一个复杂的 require 语句，包括了类型声明，allow 规则：

```
require {  
  attribute domain;  
  type etc_t;  
  class file { read getattr };  
}  
type httpd_t, domain;  
allow httpd_t etc_t : file { read getattr };
```

正如你所看到的，这里的 allow 示例中使用到的策略组件要么是自行声明的，要么是在它前面进行请求的，如 domain 属性就是在声明类型 httpd_t 前进行请求的，很明显，越复杂的载入式模块越需要 require 语句，在第 12 章“参考策略”中，我们将会讨论参考策略是如何自动产生 require 语句的。

我们使用 require 语句指出载入式模块在安装时绝对需要的必要条件，optional 语句指出的是可要可不要的条件，这样策略作者就可以基于策略组件是否需要而添加规则了，如下面的 optional 语句：

```
optional {  
  require { type user_home_t; }  
  allow httpd_t user_home_t : file read;  
}
```

这个语句允许类型为 httpd_t 的进程读取类型为 user_home_t 的文件，如果类型 user_home_t 存在的话，正如你所看到的，optional 语句包括了标准的策略语句，包括 require 语句，无论何时增加模块到系统中或从系统中移除模块，都会检查所有的可选的依赖。

完整的 require 语句语法如下：

```
require {请求列表}
```

请求列表 一个或多个由分号隔开的请求声明，一个请求声明是由一个策略组件类型标识符跟上策略组件的名称组成的，有效的策略组件类型标识符有 class, user, role, type, attribute 和 bool。对于用户，角色，类型，属性和布尔值，只需要列出一个名字即可（如 type httpd_t），对于客体类别来说，就需要同时列出客体类别和许可集（如 class file {read write};）。

require 语句在基础模块和条件语句中都无效。

下面是完整的 optional 语句的语法：

optional {规则集}

规则集 如果 optional 语句的所有 require 语句都符合要求，它就是一个或多个启用的策略语句，有效的策略语句有 user, role, type, attribute 和 alias 声明，以及 TE 和 RBAC 规则（包括条件语句）。

optional 语句仅在基础和非基础载入式策略模块中有效，在条件语句中无效。

5. 4. 类型规则

类型规则在创建客体或在运行过程中重新标记时指定其默认类型，在第 2 章中我们已经看到一个使用 type_transition 规则进行域转换的例子，在策略语言中定义了两个类型规则：

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

type_change 使用 SELinux 的应用程序执行标记时指定其默认类型。

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

5. 4. 1. 通用类型规则语法

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

规则名称	type_transition 或 type_change
源类型	创建或拥有进程的类型
目标类型	包含新的或重新标记的客体的客体类型
客体类别	新创建的或重新标记的客体的类别
默认类型	新创建的或重新标记的客体的单个默认类型

完整的类型规则语法如下：

通用类型规则语法

完整的通用类型规则语法如下：

规则名称 类型集 类型集:类别集 默认类型；

规则名称 类型规则的名称，有效的规则名称有 type_transition, type_change 和 type_member。

类型集 一个或多个类型或属性。在规则中源和目标类型有其独立的类型集，多个类型和属性使用空格进行分隔，并用大括号将它们括起来，如 {bin_t sbin_t}，可以在类型名前放一个 (-) 符合将其排除，如 {exec_type - sbin_t}。

类别集 一个或多个客体类别，多个客体类别必须使用大括号括起来，并用空格分开，如 {file lnk_file}。

默认类型 为新创建的或重新标记的客体类别指定的单个默认类型，

这里不能使用属性和多个类型。
所有类型规则在单个策略，基础载入模块，非基础载入模块和条件语句中都有效。

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

最简单的类型规则包括一个源默认类型，一个目标默认类型和一个客体类别，如：

```
type_transition user_t passwd_exec_t : process passwd_t;
```

这条规则你应该在第 2 章中已经看到过了，它指出了当一个类型为 user_t 的进程执行一个类型为 passwd_exec_t 的文件时，进程类型将会尝试转换，除非有其它请求，默认是换到 passwd_t，当声明的客体类别是进程（process）时，隐含着目标类型要与 file 客体类别关联，声明的客体类别（process）与源和默认类型关联，这个隐藏着的关联很容易被忽略，即使你成为一个策略编写专家也容易犯这个错。

和 AV 规则一样，我们可以同时指定多个客体类别，只要使用大括号将它们括起来，并用空格进行分隔就可以了，同样，我们还可以使用属性，在类型规则中列出类型和属性集，如：

```
type_transition { user_t sysadm_t } passwd_exec_t :  
process passwd_t;
```

这条 type_transition 规则在源列表中包括两个类型：user_t 和 sysadm_t。和 AV 规则一样，这条规则将会展开为两条规则，前面这条规则与下面这两条规则的含义完全一样：

```
# 这两条规则。  
type_transition user_t passwd_exec_t : process passwd_t;  
type_transition sysadm_t passwd_exec_t : process  
passwd_t;  
  
# 等于下面这一条规则。  
type_transition { user_t sysadm_t } passwd_exec_t :  
process passwd_t;
```

属性也可以像在 AV 规则中那样工作。

和源和目标类型区域不同，属性和/或多个类型不能用于默认类型，当你理解了这条规则的目的就清楚为什么要这样限制了（即要指定的是单个默认类型），如果我们列出不止一条默认类型，规则将会产生二意性，内核就决定不了究竟使用哪个默认类型。

这个限制还指出了另一个意思：我们不能指定两条独立的规则，它们拥有相同的源类型，目标类型和客体类别，因为这样写的规则相当于有两个默认类型了，如下面这两条规则将会产生冲突：

```
# 这两条规则将会冲突，编译时会报错
type_transition user_t passwd_exec_t : process passwd_t;
type_transition user_t passwd_exec_t : process
user_passwd_t;
```

如果这两条规则在一个策略中，编译时编译器会产生一个错误消息，冲突的原因就是因为默认类型只允许有一个。

5.4.2. 类型转换规则

我们使用 `type_transition` 规则指定默认类型，目前有两种格式的 `type_transition` 规则，第一种支持默认域转换事件，第2章中介绍的就是这种格式，第二章格式指出客体转换，它允许我们指定默认的客体标记。

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

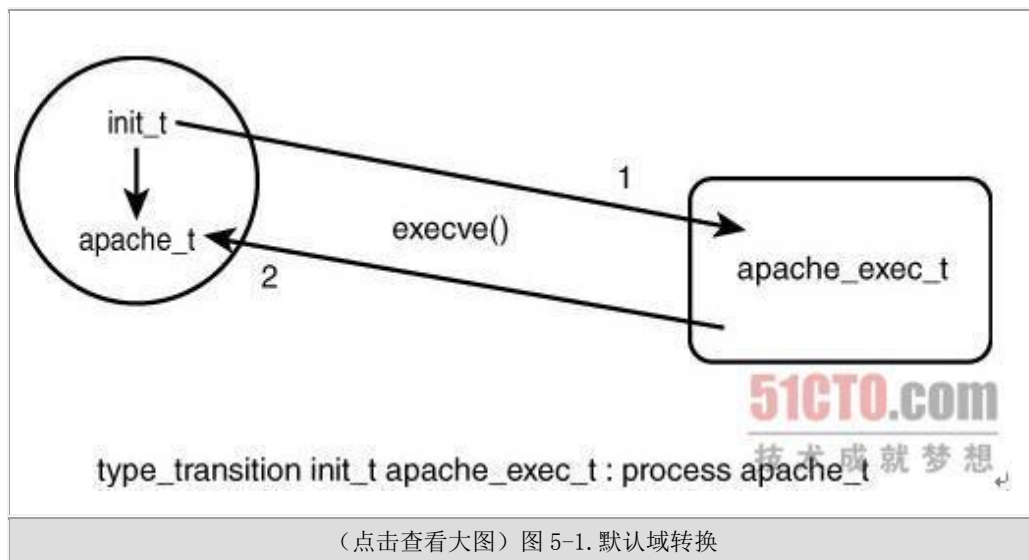
`type_transition` 规则没有 `allow` 访问权，它仅提供一个新的默认类型标记，要成功进行类型转换，也必须需要一套相关联的 `allow` 规则，以允许进程类型可以创建客体 and 标记客体。此外，默认的标记指定在 `type_transition` 规则中了，只有创建进程没有明确地覆盖默认标记行为它才有效。

5.4.2.1. 默认域转换

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

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

这条规则指出类型为 `init_t` 的进程执行一个类型为 `apache_exec_t` 的文件时，进程类型将会转换到 `apache_t`。客体类别 `process` 只表示这是一个域转换规则的格式。图 5-1 显示了一个域转换，实际上，域转换只是改变了进程现有的类型，而不是新建了一个进程，这是因为在 Linux 转换创建一个新的进程首先是要调用 `fork()` 系统调用复制一份现有的进程，如果进程类型在 `fork` 上被改变了，它就会允许域在新的域中执行任意的代码了，通过 `execve()` 系统调用执行一个新的程序时，发生域转换时就更安全些。



警告：最近的 SELinux 版本引入 `process` 客体类别许可 `dyntransition`，引入这个许可主要是为了和其它系统保持兼容，它允许进程在请求时转换它的域类型，而不是在刚刚执行时，这种进程转换不安全，因为它允许域在新的域中执行任何代码，这样就消除了两个域之间的距离，此外，相同的功能通常可以使用其它的安全机制实现，我们建议在你的策略中不要使用这种许可，除非你在建立一个用户空间客体管理器，或你确实必须要使用它。

正如前面谈到的，只有当策略允许了有关的访问权时才会发生类型转换，域转换要成功，策略必须允许下面三个访问权：

`execute` 源类型 (`init_t`) 对目标类型 (`apache_exec_t`) 文件有 `execute` 许可。

`transition` 源域 (`init_t`) 对默认类型 (`apache_t`) 必须要有 `transition` 许可。

`entrypoint` 新的（默认）类型 (`apache_t`) 对目标类型 (`apache_exec_t`) 文件必须要有 `entrypoint` 许可。

同时，上面的域转换规则要想成功，还必须要有下列的 `allow` 规则：

```
# 这条域转换规则.
type_transition init_t apache_exec_t : process apache_t;

# 至少需要下面三条 allow 规则才能成功
allow init_t apache_exec_t : file execute;
allow init_t apache_t : process transition;
allow apache_t apache_exec_t : file entrypoint;
```

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

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

5.4.2.2. 默认客体转换

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

客体转换规则由客体类别进行标记，如：

```
type_transition passwd_t tmp_t : file passwd_tmp_t;
```

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

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

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

5.4.3. 类型改变规则

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

type_transition 规则不同，type_change 规则的影响不会在内核中生效，而是依赖于用户空间应用程序，如 login 或 sshd，为了在策略基础上重新标记客体，如下面的规则：

```
type_change sysadm_t tty_device_t : chr_file
sysadm_tty_device_t;
```

这条 type_change 规则指出以 sysadm_t 名义重新标记一个类型为 tty_device_t 的字符文件时，应该使用 sysadm_tty_device_t 类型。

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

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

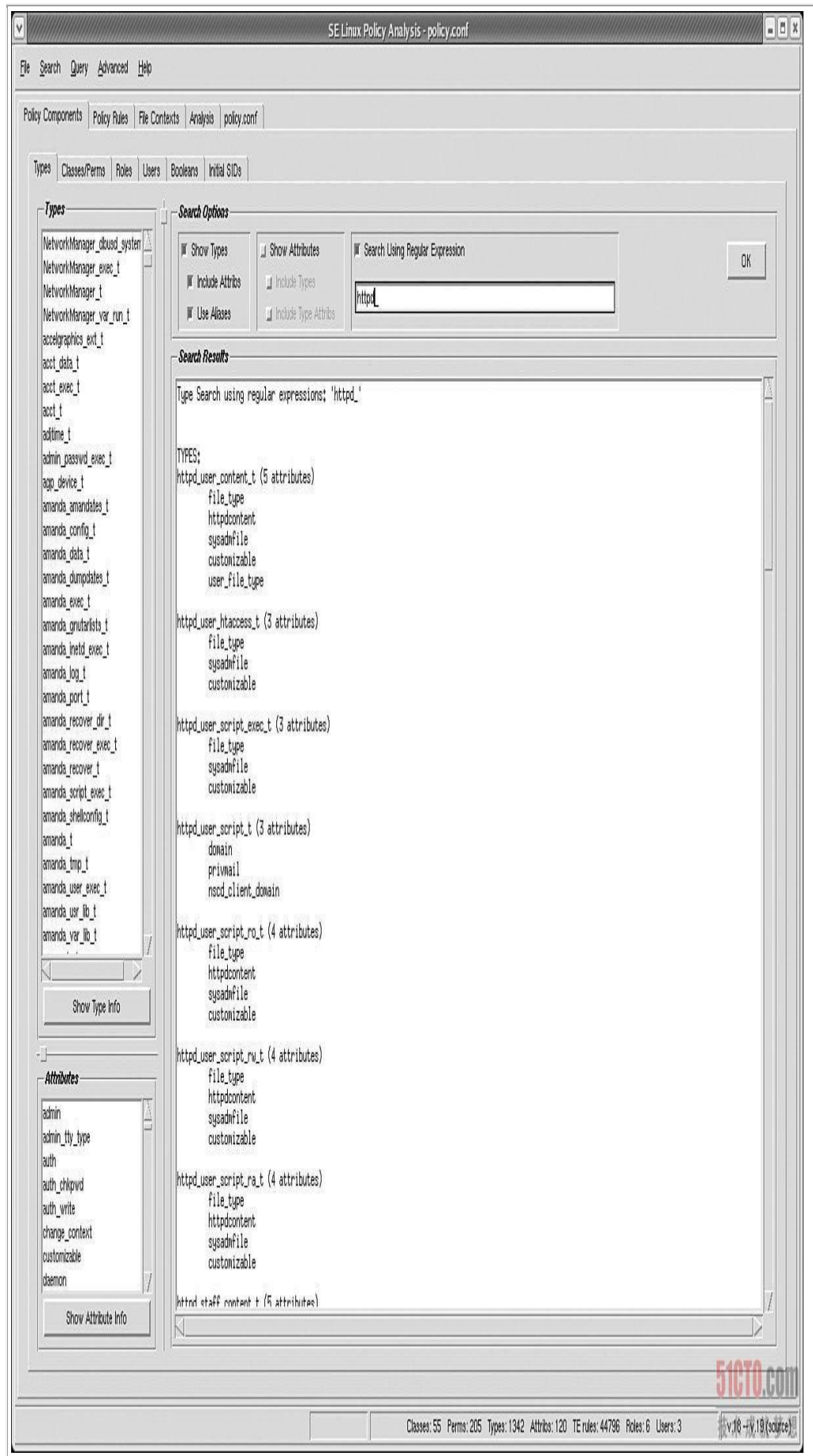
type_member 规则

策略编译器还支持第三个类型规则 type_member，目前，这个规则还没有语义含义，如果使用它不会产生任何影响，我们在这里提到它，主要是因为目前它的开发工作仍在继续，type_member 规则打算支持多个实例化客体的成员的类型，它将在有完整的语义后启用，这个规则的语法和其它两个类型规则一致。

5.5. 用 Apol 研究类型强制规则

我们已经看到要用一条规则来学习和理解所有的类型强制声明和规则很困难，例如：确定所有类型都是属性的一部分，要求在一个策略中检查所有的 type 和 typeattribute 语句，在一个大型策略中，包括成千上万的语句，横跨许多文件，这是一个艰巨的任务，为了使策略分析工作自动化，就创建了策略分析和调试工具 apol。

第一次打开 apol 载入一个策略时，如图 5-2 所示，当类型 (Types) 标签被选中时就可以看见策略组件 (Policy Component) 标签了，左边列出了所有的类型和属性，右边显示了一个检索窗口，选择一个类型然后点击显示类型信息 (Show Type Info)，会打开一个新的窗口显示与该类型有关的所有属性和别名，与此类似，选择一个属性并点击显示属性信息 (Show Attribute Info) 时，会打开一个窗口显示那个属性包括的所有类型。图 5-3 显示了关于 domain 属性的详细信息，这是 apol 最简单但最有价值的一个功能了。



(点击查看大图) 图 5-2. 使用 apol 工具检查类型和属性

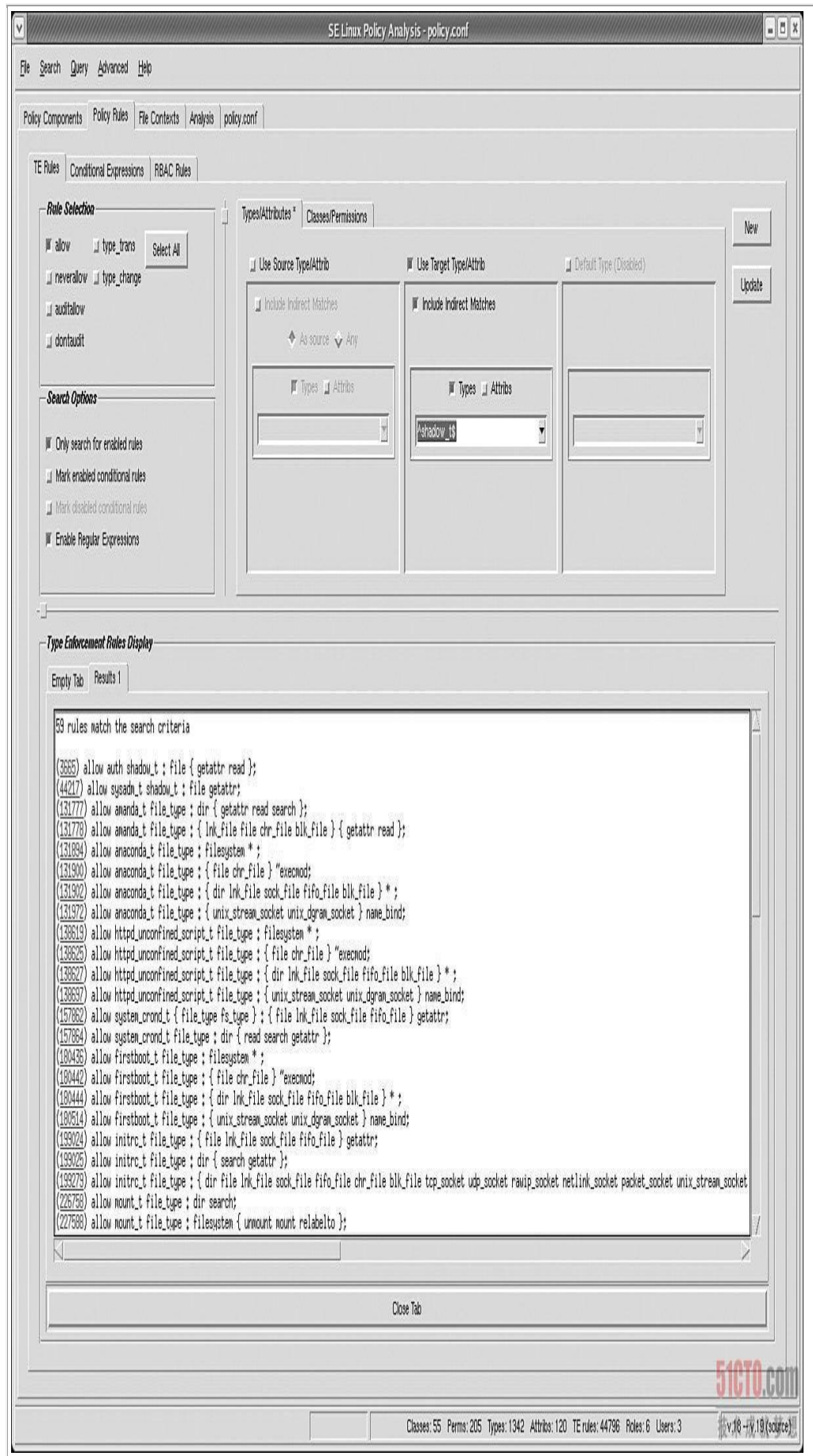


图 5-3. domain 属性的详细信息

除了显示关于类型和属性的信息之外，apol 让我们可以使用正则表达式检索类型和属性，图 5-4 显示了在所有类型中检索属性和别名中包括字符串 httpd_的类型。



apol 还可以检索策略规则，包括检索没有直接包括类型是而属性的规则，apol 的规则检索功能非常强大，但我们在这里只想介绍一下其中的部分功能，图 5-5 显示了检索目标类型为 shadow_t 的 allow 规则，注意选中“包括间接匹配 (Include indirect matches)”按钮，它意味着通过属性间接引用了 shadow_t 的规则也会包括进来，靠人工检索规则和分解属性几乎是不可能完成的任务。



(点击查看大图) 图 5-5. 检索目标类型为 shadow_t 的 allow 规则

当你阅读完本书后尝试理解一个 SELinux 策略时,你会发现 apol 是一个非常有价值的工具,利用它,你可以慢慢研究策略的内容,执行精确的查找,浏览策略组件,如类型和客体类别。尤其重要的是,你将会在策略规则 (Policy Rules) 标签下找到 TE 规则 (TE Rules),它可以帮你回答所有的问题,当你熟悉了工具和 SELinux 策略后,你应该转到分析(Analysis)标签下的工具,这些工具执行一些复杂的策略分析任务。

5.6. 小结

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

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

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

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

我们使用 allow 规则指出访问一个域类型时需要一个什么客体类型,我们根据客体类别和许可指定访问权。

默认情况下,访问被允许时不产生审核消息,而是访问被拒绝时产生。我们使用 dontaudit 规则指出被拒绝的访问不产生审核消息,我们使用 auditallow 规则指出允许的访问要产生审核消息。

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

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

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

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

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

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

练习

- 1、声明一个类型 samba_t，属性为 doamin，别名为 smbd_t。
- 2、创建一条 allow 规则，让类型为 samba_t 的进程对类型为 user_home_t 的文件有 read，write 和 getattr 的访问权。
- 3、将下面的规则尽可能转换为少的 allow 规则：

```
allow samba_t self : process *;  
allow samba_t user_homedir_t : dir { read getattr search };  
allow samba_t user_homedir_t : dir { write add_name };  
allow samba_t user_homedir_t : file { read getattr };  
allow samba_t user_home_t : file { write };
```

- 4、编写一条访问向量规则（AV 规则），当用户的 ssh 密钥文件类型为 user_ssh_key_t 时，就产生审核消息。
- 5、编写一条 type_transition 规则，当类型为 sysadm_t 的进程在类型为 tmp_t 的目录下创建文件时，该文件的类型就为 sysadm_tmp_t。
- 6、编写一条 type_transition 规则，当类型为 user_t 的进程执行类型为 games_exec_t 的文件时，域就转换到 games_t。
- 7、编写最小的 allow 规则，要求练习 6 中的 type_transition 规则成功。

第 6 章. 角色和用户

SELinux 提供了一种依赖于类型强制（TE）基于角色的访问控制（RBAC），角色用于组域类型和限制域类型与用户之间的关系，SELinux 中的用户关联一个或多个角色，使用角色和用户，RBAC 特性允许有效地定义和管理最终授予 Linux 用户的特权。

6.1. SELinux 中基于角色的访问控制

SELinux 中的角色和用户构成了它的 RBAC 特性的基础，你可能会觉得奇怪，为什么我们现在才开始讨论角色和用户，其它大部分主流操作系统的安全特性几乎都集中于将访问权授予用户，或通过用户组或角色的机制进行授权，但这与 SELinux 中的情况有所不同，SELinux 中的访问权不是直接授予用户或角色的，相反，正如第 5 章“类型强制”中讨论到的，访问权是通过 TE allow 规则授给类型的，角色扮演的是类型强制的一个支持特性，它和用户一起为 Linux 用户及其允许运行的程序提供了一种绑定基于类型的访问控制，SELinux 中的 RBAC 通过定义域类型和用户之间的关系对类型强制做了更多限制，以控制 Linux 用户的

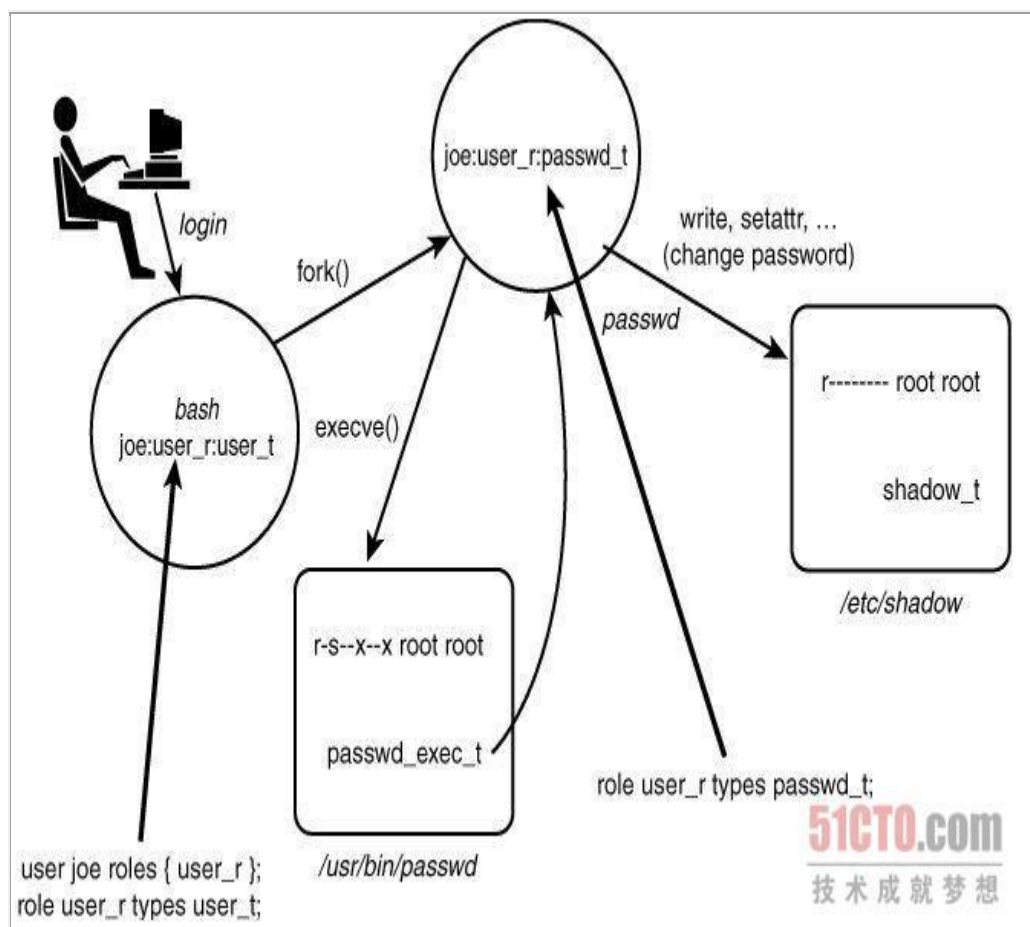
特权和访问许可，RBAC 没有允许访问权，在 SELinux 中，所有的允许访问权都是由类型强制提供的。

警告：Linux 和 SELinux 有它们自己的用户标识符，这让人难以理解，为了避免混淆，在本书写作过程中，如果指的是 `/etc/passwd` 中定义的用户，我们就用“Linux 用户”来表达，如果没有明确指出是什么用户或用户标识符，那就指的是 SELinux 策略中在安全上下文中定义的用户标识符了。

6.1.1. SELinux 中 RBAC 概述

正如前面所述，SELinux 中的 RBAC 特性依赖并支持 TE 特性，我们通过将域类型和一个或多个角色进行关联，而不是直接将权限授给用户，RBAC 通过在安全上下文中控制域类型、角色和用户的关联实现对 TE 策略更多的约束，也就是说，域转换是受用户的角色约束的，最终约束了用户的总体权限。

为了增强理解，我们还是以第 2 章“概念”中的例子进行说明，在图 6-1 中，我们进行了更详细的表达。这个例子描述了一个从域类型为 `user_t` 的 `bash` shell 进程转换到一个域类型为 `passwd_t` 运行密码程序的进程，注意，我们在进程的安全上下文中添加了用户和角色部分（`joe:user_r` 和 `joe:user_r:passwd_t`），同时，也假设策略已经包括了必要的 TE 规则许可域转换（这个没有在图中进行标识）。



这个例子显示了两类 RBAC 策略语句：一个用户声明语句（user）和两个角色声明语句（role）。这些语句在策略中创建了用户、角色和类型标识符之间的关系。在本章后面，我们将列出这些声明语句完整的语法，现在，先理解它们在域转换上所起的作用。

图 6-1 中的 user 语句将 SELinux 用户 joe 和角色 user_r 关联起来，这条语句告诉 SELinux 用户 joe 和角色 user_r 在安全上下文中是可以共存的，如果没有这条语句，图 6-1 中的用户 joe 和角色 user_r 进程安全上下文将是无效的，SELinux 将会拒绝创建它们，最后拒绝进行域转换。

那两个 role 语句将角色 user_r 和域类型 user_t 和 passwd_t 关联起来，与 user 语句类似，要使进程安全上下文有效，role 语句也是必需有的，如果没有 role 语句关联类型 passwd_t，即使 TE 策略允许，域转换也会失败。如果我们不想 user_r 角色运行密码程序，我们只需要将 role 语句移除，内核也就不会创建对应的安全上下文了，即使 TE 规则允许进行访问。

6.1.2. 用角色管理用户权限

如图 6-1 中的例子所示，我们不会直接将域类型和用户进行关联，相反，域类型是与角色进行关联的，然后再将角色与 SELinux 用户进行关联，这就间接增加了两个效果，首先，他简化了管理所有策略的复杂度，一个系统可能只有三或四个角色，但有成千上万的用户和域类型，直接将域类型与用户进行关联，将会导致管理非常困难，将域类型分配给一小撮描述类型（如普通用户域类型）特权集的角色，然后将这些角色指派给用户，这样就更容易管理了。

SELinux 中的角色也允许我们限制用户的访问，对于任何进程，同一时间只有一个角色（即进程安全上下文中的角色）是“活动的”，因为域类型是与角色进行关联的，域转换就会受到与它们关联的活动角色的限制了。

限制域转换只对当前活动的角色有效，允许用户关联更多的角色，不会获得所有角色访问权的并集，例如：我们可以将一个用户与系统管理角色和受限的普通用户角色进行关联，在后面进行交互时将会使用普通用户角色，而不是管理角色，在这个例子中，在常规使用过程中，普通用户角色将会被激活，阻止访问权授予管理域类型，仅当必需执行系统管理任务时激活管理角色（即通过域转换改变），这与标准 Linux 下使用普通账户切换到 root 账户进行系统管理非常类似，但比准 Linux 方式更精细。

关键要记住角色仅仅是一套域类型的集合，他可以方便地与用户建立联系，它们不是 SELinux 中独立的访问控制机制。

提示：

大量的实用程序，如 `newrole` 和一个经过修改的 `su` 命令提供了一种方法让用户（或用户进程）改变当前（即激活的）的角色，通过以一个不同的安全上下文进行域转换创建一个新的 shell 进程实现的（参考第 13 章“管理 SELinux 系统”），Fedora Core 5 (FC 5) 移除了 `su` 改变角色的功能，相反，它使用了 `newrole`，改变角色是由关联的用户和角色（即 `user` 语句）以及角色允许（`allow`）规则控制的，我们将在本章后面讨论。

角色 vs 用户域类型

迄今为止，大部分 SELinux 策略仅以有限的方式使用角色，这也在一定程度上说明了角色在类型强制的辅助作用，目前典型的状况是与每个角色关联的域类型都是“用户域类型”，它是初始登陆时为用户 shell 进程指派的角色类型，例如：普通的，非特权用户域类型 `user_t` 与角色 `user_r` 关联，同样，特权用户的非信任域类型是 `staff_t`，它与角色 `staff_r` 关联。

这些初始化用户域类型以及所有能转换的域类型都定义角色“`user`”和“`staff`”，例如：这两个“角色”之间最主要的不同是 `staff` 角色（和 `staff_t` 域类型）可以转换到具有特权的角色和管理角色和用户域类型（`sysadm_t`，它与 `sysadm_r` 角色关联）。

每个角色都有一个初始用户域类型的结果是可能部分程序一开始就有起源域类型，例如：对于保存的下载数据，包括程序，由角色进行隔离（减少选择管理用户的机会），我们可以为每个角色以不同域类型运行 Web 浏览器，为了实现这个目标，我们为关联的用户域类型（`user_t` 和 `staff_t`）创建了不同的域转换角色，执行 Web 浏览器可执行文件时，每个用户域类型将转换到不同的域类型（即 `user_mozilla_t` 和 `staff_mozilla_t`），而不是同一个类型（即 `mozilla_t`），因此，普通用户（`user_t`）和管理用户（`staff_t`）在 Web 浏览时会相互保护，我们应该将角色只关联到恰当的类型（例如：`user_mozilla_t` 应该只与 `user_r` 关联），为了实现这个分隔，我们应该为每个 Web 浏览器域创建一个单独的文件类型，并且只允许域类型“`write`”【即写访问】访问它们各自的文件类型，结果就是 Web 浏览器运行在一个不同的域，依赖于运行它的用户角色，而下载数据是基于角色分开的。

6.1.3. 客体安全上下文中的用户和角色

在我们的密码策略示例中（参考图 6-1），我们没有包括显示的文件客体完整的安全上下文（即可执行文件 `/etc/bin/passwd` 和 `shadow` 密码文件 `/etc/shadow`），相对客体安全上下文中的用户和角色部分而言，它们的重要性要低一些，然而客体必需要有一个完整的安全上下文，用户字段支持审核，而角色没什么用，如果我们在我们的示例系统中检查图 6-1 中的客体，我们会发现如下所示的安全上下文：

```
# ls --scontext /usr/bin/passwd /etc/shadow
system_u:object_r:shadow_t      /etc/shadow
system_u:object_r:passwd_exec_t /usr/bin/passwd
```

正如你所看到的，这两个客体都有特殊的角色 `object_r`，它是所有客体代表性的角色，这个角色是硬编码进 SELinux 的，它是不需要声明的，对于所有类型都是隐含允许的，你不要尝试声明角色 `object_r`。

客体安全上下文的用户部分通常设置为创建进程安全上下文的用户部分，这个特性有一些潜在的用途，可以跟踪是哪个用户创建的这个客体，但通常没有安全强制效果（我们将在第 9 章“条件策略”中讨论），在前面的例子中，这个两个客体的用户都是 `system_u`，它是一个特殊的用户，在许多策略中都存在，它代表系统资源和进程。

6.2. 角色和角色语句

除了 `object_r` 外，SELinux 没有内置任何的角色，与类型一样，角色也需要在策略中进行声明，与角色有关的有 4 个策略语句：角色声明语句，角色 `allow` 规则，角色转换规则和角色控制语句。

6.2.1. 角色声明语句

角色声明语句 (`role`) 声明一个角色标识符，图 6-1 中的示例包括下面两个 `role` 语句：

```
role user_r types user_t;
role user_r types passwd_t;
```

这些语句将域类型 `user_t` 和 `passwd_t` 与角色 `user_r` 关联起来了，正如你所看到的，对于相同的角色标识符，`role` 语句可以重复，第一个 `role` 语句给出了一个角色标识符，并关联了列出的类型，后面所有的 `role` 语句都是关联额外的类型，一个角色使用多个 `role` 语句的情况通常见于角色语句比较靠近它关联的类型的声明（即在相同的策略源模块下），完整的角色声明语法如下：

角色声明语句语法

角色声明语句声明角色标识符，并关联类型。类型必需要与角色进行关联，它们要共存于安全上下文中，特殊角色 `object_r` 是预先定义好的，并且是隐含与所有类型都关联了的，在所有客体的安全上下文中，它都是存在的，对于同一个角色标识符，可以有多条 `role` 语句，第一条语句声明角色，并关联一个或多个类型，后面的语句仅仅是关联类型，完整的角色声明语句如下：

`role 角色名称 [类型 类型集]`

角色名称 角色标识符，如果它是在第一条 `role` 语句中，这就声明了一个角色，标识符长度任意，可以包括 ASCII 字符，数字，点 (.) 和下划线 (_)。点 (.) 在角色标识符中有特殊的用途，它用于指出分配给角色的类型集上的约束，例如：某个角色的类型集 `apache.cgi` 必需是类型集 `apache` 的子集。

类型集 一个或多个类型或标识符属性，使用大括号 {} 将多个标识符括起来，标识符之间使用空格分开，如 {`user_t passwd_t`}，类型也是可以被排除的，只需要在类型名称前加上一个“-”即可，如 {`exec_type`

- sbin_t}，如果省略 type_set（同关键字 types 一起），角色声明时就不会关联任何类型。

角色声明在单个策略，基础载入模块和非基础载入模块中都是有效的，但在条件语句中就无效了。

6.2.3. 角色转换规则

因为程序在执行过程中角色可能发生变化，某种程度上与类型相似，在策略语言中，我们需要一个方法实现这种转换的自动化，对于类型而言，我们使用了 type_transition 规则进行自动化处理，对于角色，我们就使用角色转换规则 role_transition，这个规则在作用和语法上与 type_transition 很类似，如：

```
role_transition sysadm_r http_exec_t system_r;
```

这个规则说明，当一个角色为 sysadm_r 的进程执行一个类型为 http_exec_t 的文件时，SELinux 将会尝试将起转换为 system_r 角色。

与 type_transition 规则一样，role_transition 规则也不允许（allow）规则执行角色改变，假设这样，要使得角色改变成功，角色 allow 规则也是必须的，角色转换规则通常用于系统管理员直接执行系统后台进程时改变角色，而不是初始化进程（init），如果这种情形下不使用角色转换规则，后台进程可能有一个不同的角色，这依赖于是谁启动的它，除了这种情况外，我们不希望角色被隐含地改变，相反，我们希望用户在需要的时候明确地改变它们的角色（如 newrole 命令）。角色转换规则语法如下：

角色转换规则语法

角色转换规则指定了一个执行文件时默认的角色变化，角色转换规则没有允许访问权，要成功转换角色，角色 allow 规则也是必须的，完整的角色转换规则如下：

role_transition 角色集 类型集 角色；

角色集 一个或多个角色标识符，多个标识符之间使用空格分隔，并使用大括号将它们括起来，如 {staff_r sysadm_r}。

类型集 一个或多个类型或属性标识符，多个标识符之间使用空格进行分隔，并使用大括号将它们括起来，如 {user_t passwd_t}。可以使用前缀字符“-”来排除类型，如 {exec_type - sbin_t}。

角色 角色转换后安全上下文中的新角色。

角色转换规则在单个策略，基础载入模块和非基础载入模块中都有效，但在条件语句中无效。

6.2.4. 角色控制语句

角色控制语句（dominance）按照其他角色声明一个角色，我们可以使用这个语句创建一个角色之间的层次关系，假设这样，“高层角色”可能会自动继承所有与角色关联的类型，例如：

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

这个角色 dominance 语句声明了一个角色 super_r，如果它没有声明，并使它控制角色 sysadm_r 和 secadm_r，角色 super_r 将拥有与角色 sysadm_r 和 secadm_r 合并后的类型，如果这些“高层角色”发生了任何变化，合并内容也将为 super_r 发生变化，注意任何添加到控制语句后的高层角色的类型没有通过 dominance 语句继承下来，因此，在前面的例子中，如果在 dominance 语句后面给 secadm_r 角色添加了一个类型，super_r 角色不会继承这个新的类型，角色 dominance 语句在现有策略中也没有广泛使用，完整的角色控制语句语法如下：

角色控制语句语法

角色控制语句指出了角色之间的层次关系，角色继承所有它们控制的角色合并后的类型，角色控制语句的基本语法如下：

dominance {role 角色名称 {角色集}}

角色名称 角色标识符，标识符长度不受限制，可以包括 ASCII 字符，数字，点 (.) 和下划线 (_)。

角色集 一个或多个以“role 角色名称”形式指定的角色，多个角色之间使用空格分隔，如 {role staff_r;role sysadm_r}。

策略语言不支持更复杂的语法，这里的角色集可以包括嵌套的控制关系定义，使用大括号进行表示，如：

```
dominance { role a_r { role b_r; role c_r { role d_r; } } }
```

在这个例子中，角色定义如下：

d_r 只有它自己的类型

c_r 它的类型和 d_r 的类型

b_r 只有它自己的类型

a_r 它自己的类型 and 所有 b_r, c_r 和 d_r 的类型

角色控制语句在单个策略，基础载入模块和非基础载入模块中都是有效的，但在条件语句中无效。

6.3. 用户和用户语句

Linux 和 SELinux 用户标识符是不同的，通常也没什么关联，在 SELinux 中，进程的 Linux 用户标识符和 SELinux 用户标识符是不同的（例如：参考稍后讨论的 user_u），当初设计 SELinux 用户标识符时，希望实现一个固定不变的 SELinux 用户标识符，所以才没有共享 Linux 的用户标识符，在标准 Linux 中，用户标识符的改变反应的权限的变化（例如：改变为 root 账号），在多数情况下，这两个真实有效的用户标识符都可以改变，这使得在跟踪审核、认证登陆的用户变得困难，将 Linux 用户标识符和 SELinux 用户标识符分隔开来允许 Linux 用户标识符按需改变，而不会对 SELinux 用户标识符产生任何影响。

注意：许多 SELinux 系统，包括 Red Hat Enterprise Linux 版本 4 (RHEL 4) 和 Fedora Core 4 (FC 4)，实际上在登陆会话过程中都可以改变 SELinux 用户标识符，特别要指出的是，su 经过修改后可以设置 Linux 和 SELinux 用户标识符，这就违背了最初设计 SELinux

用户标识符要固定不变的初衷,但如果不改变 SELinux 用户标识符可能会引起用户管理混乱,需要创建更多复杂的进程来添加用户账号,此外,Linux 审核框架出于审核目的存储了一个固定不变的登陆用户标识符,这样稍微减少了 SELinux 用户标识符改变的次数,Fedora Core 5 (FC 5) 还原到了最初的行为,不允许 SELinux 用户标识符被改变。

6.3.1. 声明用户及其关联的角色

用户声明语句 (user) 声明一个用户标识符,以及与之关联的一个或多个角色, user 语句是 SELinux 策略语句唯一关联 SELinux 用户的语句,图 6-1 中的示例包括下面的用户声明:

```
user joe roles {user_r};
```

这个语句声明了一个用户 joe,如果在策略中没有进行声明,以及与之关联的角色 user_r。与 role 语句不同,它可能是混合在 TE 语句中的, user 语句必须在类型和角色语句后,约束语句前(参考第 3 章“架构”中的图 3-5)。

与角色和类型之间的联合类似,用户联合允许一个角色和一个特定的用户出现在一个安全上下文中。

注意没有用户转换或用户 allow 规则,这反应了最初的用户不能改变的设计初衷,改变用户标识符只能由约束控制,我们将在第 9 章中讨论它。

完整的 user 语句语法如下:

用户声明语句语法

用户声明语句声明一个用户标识符,完整的 user 语句语法如下:

user 用户名称 roles 角色集;

用户名称 用户标识符,如果这是该标识符的第一个 user 语句,那就会声明一个标识符,标识符长度不受限制,可以包括 ASCII 字符,数字,点 (.) 和下划线 (_)。

角色集 一个或多个角色标识符,这些角色标识符必须事先在策略中声明,多个角色标识符之间使用空格分开,并用大括号将它们括起来,如 {staff_r sysadm_r}。

用户声明语句在单个策略,基础载入模块和非基础载入模块中都有效,但在条件语句中无效。

6.3.2. 将 Linux 用户映射到 SELinux 用户

登陆程序(如 login,sshd)负责映射 Linux 用户到 SELinux 用户,在登陆时,如果 SELinux 用户标识符恰好与一个 Linux 用户标识符完全相同,匹配的 SELinux 用户标识符成为初始 shell 进程安全上下文的用户标识符,假设这样,如果一个 Linux 用户标识符在 SELinux 策

略中也作为一个用户标识符存在,所有登陆进程将设置初始 shell 进程安全上下文用户标识符为那个匹配的 Linux 标识符。

大多数时候,特别是 RHEL 4 和 FC 4 中默认策略的常规用途配置中,策略中定义的每个普通用户不尽如人意,普通用户相对于 SELinux 而言有相同的权限(即 user_r 角色和 user_t 初始用户域类型),为了解决这个问题,SELinux 有一个特殊的用户标识符 user_u,叫做普通用户,如果普通用户 user_u 定义在策略中,所有 Linux 用户在策略中如果没有匹配到 SELinux 用户,它们都将被映射到 user_u。

例如:假设在我们的策略中有下面的 user 语句:

```
user user_u roles {user_r};
```

这个语句定义了普通用户 user_u,并授予它 user_r 角色,与前面定义的就一样,不同的地方是如果 user_u 是定义在策略中的,所有明确定义在策略中的 Linux 用户都将被映射到 user_u,因此,如果 jane 是一个 Linux 用户标识符,但在 SELinux 策略中没有定义用户 jane,当 Linux 用户 jane 登陆时,在初始 shell 进程安全上下文中的用户标识符将会是 user_u,因为 jane 是定义在策略中的,它的初始 SELinux 用户标识符将是 jane,即使在策略中也定义了 user_u。

如果普通用户 user_u 没有定义在策略中,任何没有明确在 SELinux 策略中定义的 Linux 用户标识符将不能登陆,即使在许可(permissive)模式下,原因是在登陆过程中,初始 shell 进程必须要有一个有效的安全上下文,包括用户标识符,如果策略中既没有定义 user_u,也没有定义 Linux 用户标识符,登陆进程就不能创建有效的安全上下文(因为没有用户标识符可用),因此,如果你的策略中没有 user_u(它对大多数配置都有意义),你必须明确地将所有 Linux 用户添加到 SELinux 策略中。

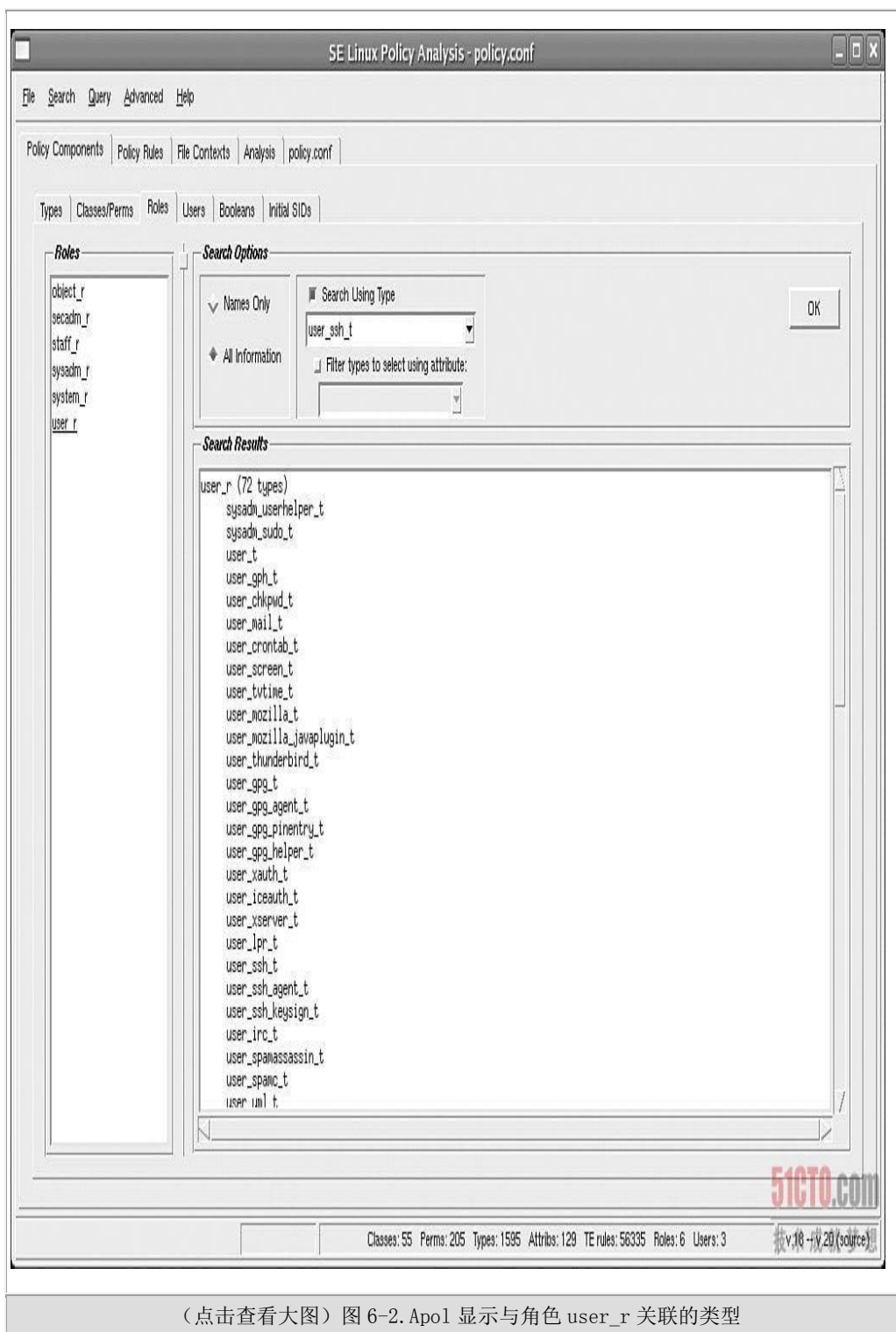
注意:在 FC 5 中,用户映射机制被增强了,允许通过一个配置文件明确地映射 Linux 用户到 SELinux 用户,这样就允许创建更多普通用户(例如 staff_u, user_u),现有的映射规则为向后兼容仍然保留,第 13 章包括了关于管理用户映射的新工具更详细的信息。

SELinux 还有第二个特殊用户,系统用户 system_u,它通常用于所有系统进程,如 init,以及由 init 启动的后台守护进程,从技术上看,用户 system_u 没有特殊的含义,在策略语言中也没有异常对待,然而,大多数现有的策略包括了这个用户,系统通常被配置为希望这个 SELinux 用户对于所有系统资源都存在,在你的策略中包括 system_u 通常被认为是一个好主意。

警告：永远不要创建一个标识符为 `system_u` 的 Linux 用户账号，如果你这样做了，那个 Linux 用户将能够以系统用户标识符身份登陆，它通常具有较高的特权（不过没有 `root` 厉害）。

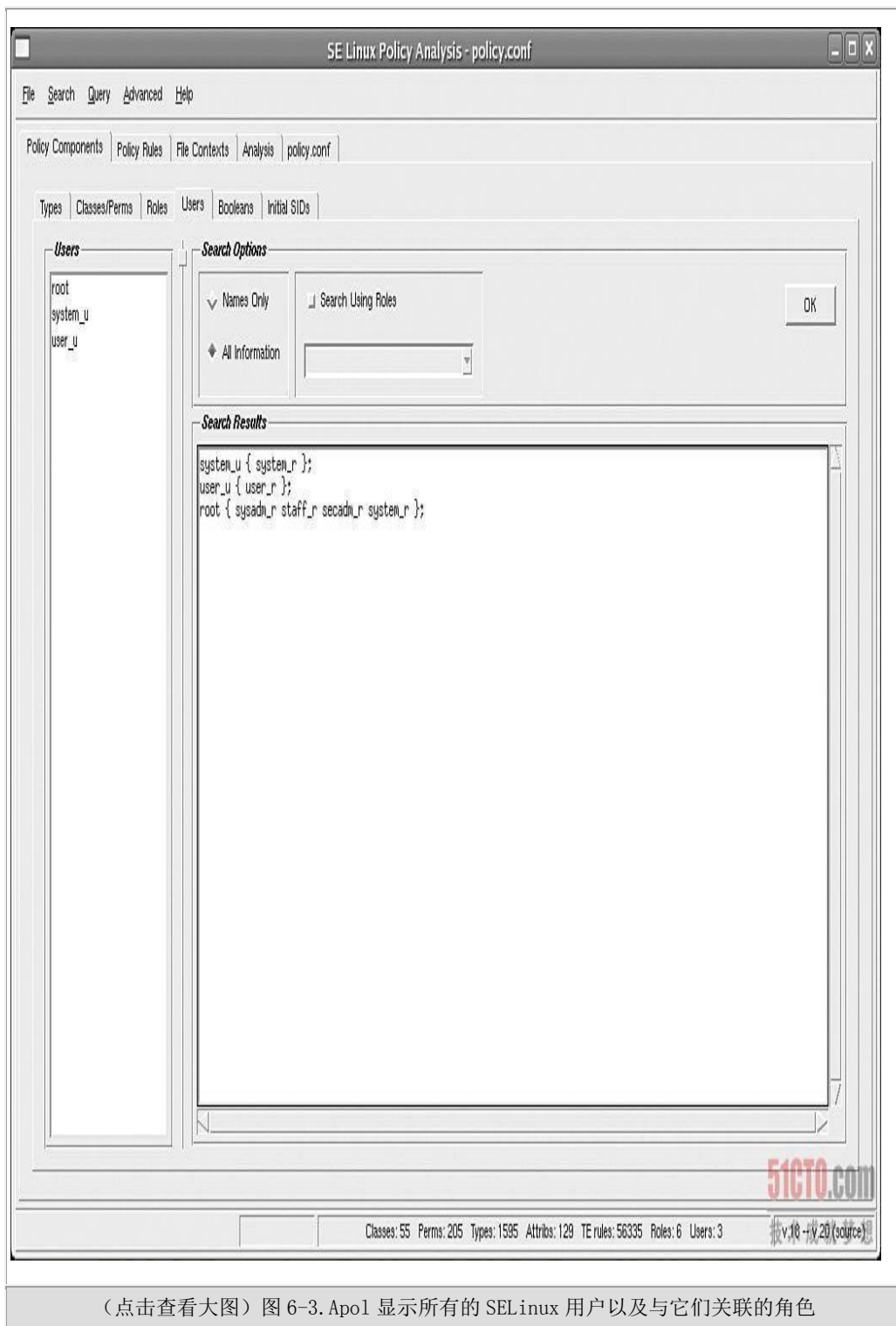
6. 4. 用 Apol 分析角色和用户

Apol 有检索和显示角色、用户的特性，图 6-2 中策略组件（Policy Components）标签上的角色（Roles）标签显示了所有的角色，并提供了检索功能，在这个例子中，我们检索关联的类型为 `user_ssh_t` 的角色，检索结果显示角色 `user_r` 与这个类型关联，因为我们以及选择了在检索结果中显示所有关于角色的信息，因此，所有与这个角色关联的类型都显示出来了，正如前面讨论到的，角色声明语句是通用的，它连接角色和类型，可以通过策略资源分发出去，apol 的这个特性使它很容易找出角色和类型之间的关系。

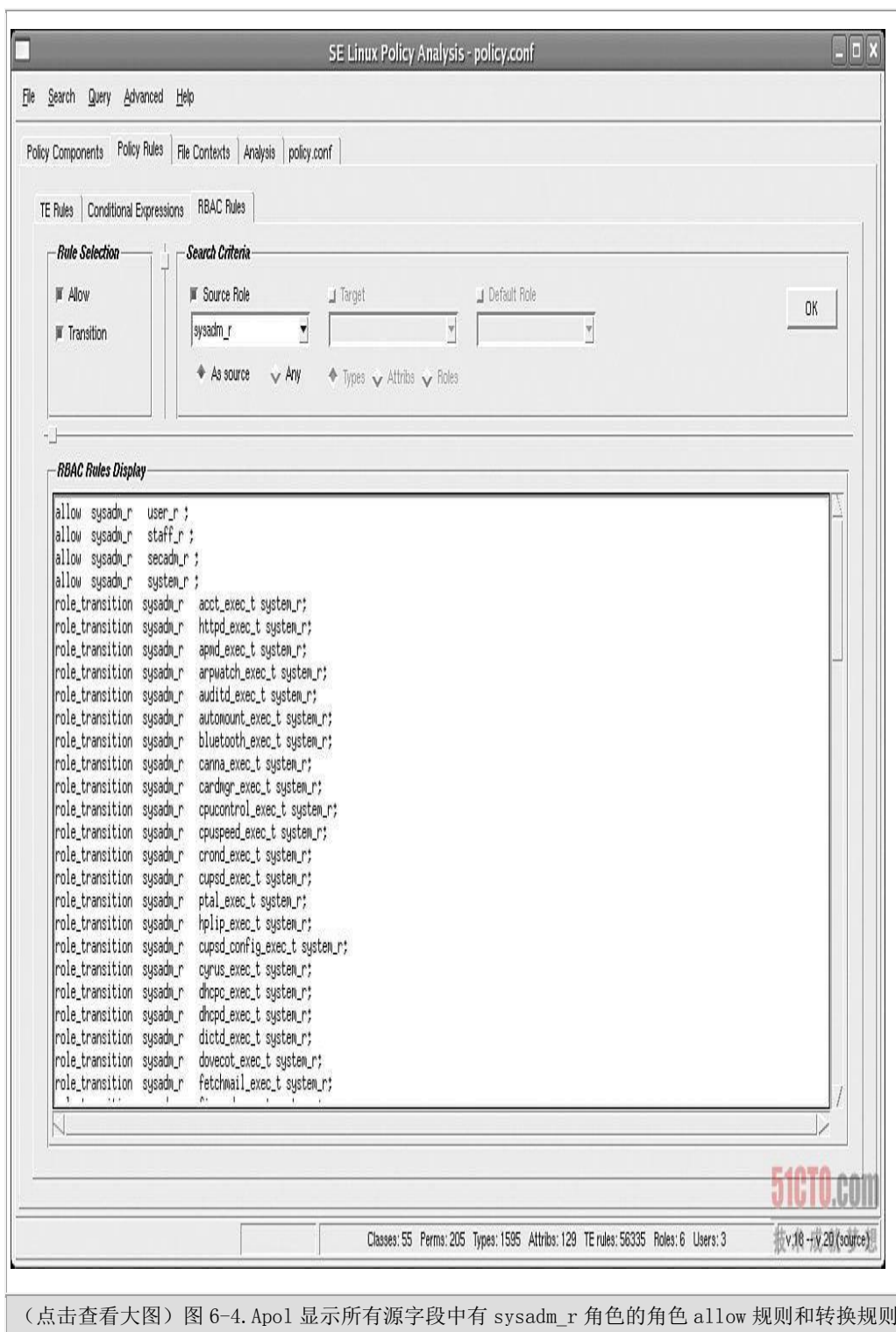


(点击查看大图) 图 6-2. Apol 显示与角色 user_r 关联的类型

策略组件 (Policy Components) 标签上的用户 (Users) 标签为用户提供了类似的功能, 图 6-3 显示了策略中所有的 SELinux 用户, 以及这些用户关联的角色, 通过关联的角色检索用户也是可能的。



此外还显示了用户和角色，apol 让我们可以检索角色 allow 和转换规则，这个特性位于策略规则（Policy Rules）标签的 RBAC 规则（RBAC Rules）标签，与 TE 规则检索特性类似，图 6-4 显示了检索所有源字段中有 sysadm_r 角色的角色 allow 规则和转换规则。



(点击查看大图) 图 6-4. Apo1 显示所有源字段中有 sysadm_r 角色的角色 allow 规则和转换规则

6. 5. 小结

在 SELinux 中，角色和用户提供了一个 RBAC 特性，与传统的 RBAC 机制不一样，SELinux 中的角色和用户依赖于类型强制而不是传统类型的访问控制。

角色是一类域类型的集合，它代表了我们分配给用户的“特权”，角色控制域转换，因为只有当新的类型被授予安全上下文的角色时，SELinux 才会创建一个安全上下文。

角色声明语句 (role) 定义一个角色标识符及其关联的一个或多个类型，在一个策略中可以对一个角色使用多个 role 语句，角色的定义是累加的，角色还可以通过较少使用的角色控制语句 (dominance) 进行声明。

角色允许 (allow) 规则控制安全上下文中的角色是否可以在 `execve()` 系统调用上进行改变，角色转换语句 (role_transition) 产生一个角色变换，默认依赖于角色的调用进程和可执行文件的类型。

SELinux 用户和 Linux 用户具有不同的用户标识符，它们之间的任何交往都会导致登陆进程转换，常见的行为是，如果 Linux 用户和 SELinux 用户标识符匹配，初始用户登陆进程安全上下文将会匹配用户标识符，除此以外，如果策略中定义了特殊用户 `user_u`，所有没有匹配的 Linux 用户都将 `user_u` 作为它们初始进程安全上下文的用户，如果没有匹配的用户，并且也没有定义 `user_u`，用户将不能登陆，即使在许可 (permissive) 模式下。

在 SELinux 中，用户提供了一个方法将 Linux 用户关联到 SELinux 角色 (另外还有与该角色关联的域类型集)，用户声明语句 (user) 指定了这个关联，除非角色通过 user 语句进行了关联，否则 SELinux 不会创建安全上下文。

练习

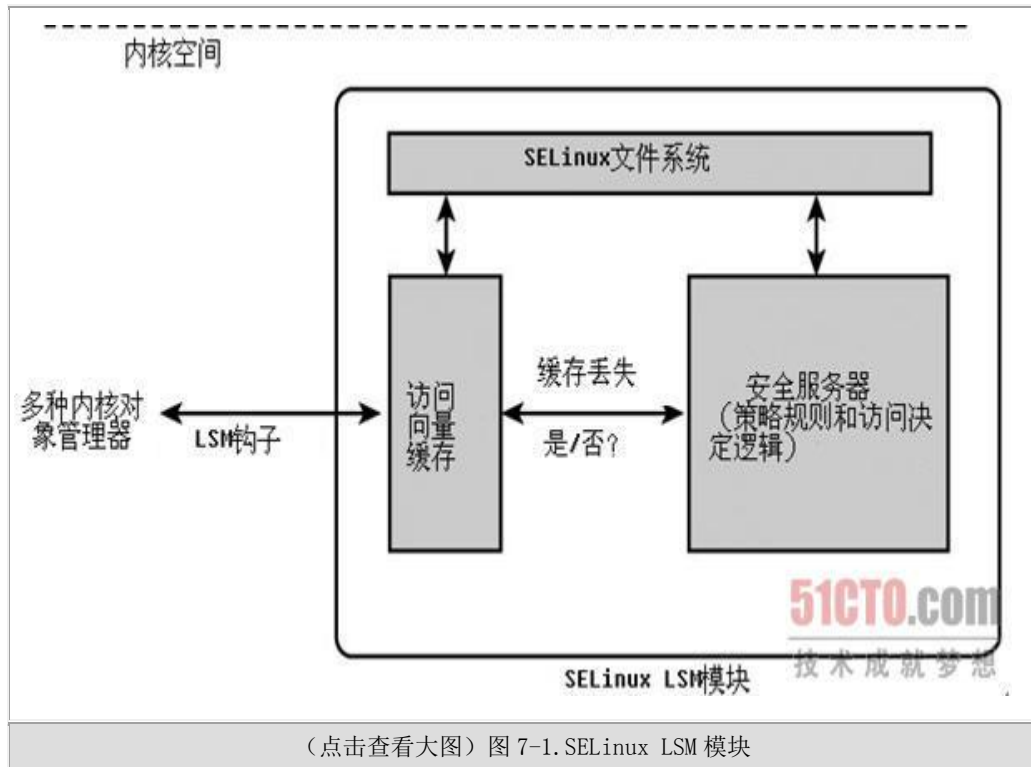
- 1、声明一个 SELinux 用户 tom，关联角色 `staff_r` 和 `sysadm_r`。
- 2、将角色 `sysadm_r` 关联到类型 `sysadm_mozilla_t`。
- 3、当角色为 `sysadm_r` 的进程执行一个类型为 `initrc_exec_t` 的文件时，写出角色转换到 `system_r` 的语句。

第 7 章 约束

SELinux 对策略允许的访问提供了更严格的约束机制，不管策略的 allow 规则如何，在这一章中，我们讨论 SELinux 的约束特性。

7.1 近距离查看访问决定算法

为了理解约束的用途，让我们再来看一下 SELinux Linux 安全模块 (LSM)，在第 3 章“架构”中讨论的 SELinux 内核架构最突出的部分，我们再次在图 7-1 中绘出。



我们想近距离看一下访问决定逻辑是如何在安全服务器内工作的，访问向量缓存(AVC)

[1]由一组源安全标识符(SID)、目标SID和客体类别锁定，SID是安全上下文内部引用的(查看下面的补充说明)。

[1]参考 `linux-2.6/security/selinux/avc.c`

安全上下文和SID

SELinux执行的是Flask安全框架，这个框架为实现增强访问控制提供了一个参考标准，但仍然保留了安全策略的中立性，这就意味着AVC和它与LSM钩子之间的接口不是特别依赖类型强制(TE)和其它的SELinux安全策略。

SELinux安全服务器通过内部与SID缔合的安全上下文应用语义含义给SID，因此SELinux可以使用SID查找类型、用户和角色标识符。

在`linux-2.6/security/selinux/*.c`和`linux-2.6/security/selinux/ss/*`源代码中可以看到SELinux LSM模块中立部分的代码，还包括AVC。

当SELinux LSM钩子[2]请求一个访问决定时，它们提供了主体(源)的SID、客体(目标)和客体类别，AVC使用一组存储为一个掩码的SID-SID-类别来查找允许的访问权。

[2]参考 `linux-2.6/security/selinux/hooks.c`。

当没有命中缓存时，AVC调用安全服务器的`security_compute_av()`[3]函数来决定允许哪些访问权，这个函数在它们的访问决定逻辑中包含了两个基本步骤：1)根据TE允许规则

为类型-类型-类别组合创建一个代表允许的客体许可的掩码，2) 从允许的许可中移除那些受约束的不允许的许可。第二个步骤意思是除了这些受约束的许可外，其它许可都是策略允许的。

[3] 参考 `linux-2.6/security/selinux/ss/services.c`。

约束的主要目的是不考虑策略中 `allow` 规则对具体的许可实现全局约束，每次调用 `security_compute_av()` 函数时，在返回允许的访问掩码给 AVC 之前都会检查约束，因此，约束在 SELinux 策略中对允许的访问权多了一层限制。

SELinux 有两种类型的约束，`constrain` 语句是最常见的约束，使得可以基于用户，角色以及源和目标安全上下文的类型做更多的访问限制，`validatetrans` 语句是最近才增加到 SELinux 中的，它使得可以基于旧的，新的和进程安全上下文对安全上下文改变事件做更多的访问限制。

注意：在编写本书时，`apol` 工具还不支持约束，因此，使用它你不能够看到这些语句，对约束的支持已经排在 `apol` 的开发计划中，相信未来一定可以实现直接利用它查看这些约束语句了。

7.2 约束语句

`constrain` 语句包含三个元素：将要应用约束的一套客体类别，为那些受约束的类别指定的一套许可和一个约束的布尔表达式。约束是由客体类别进行组织并存储在策略中的，下面是 `constrain` 语句的完整语法。

约束语句语法

`constrain` 语句使您可以通过基于源和目标安全上下文之间的关系定义约束，对特定客体类别限制特定的许可。完整的 `constrain` 语句语法如下：

`constrain` 类别集 许可集 表达式；

类别集 一个或多个客体类别，多个客体之间必须用空格分隔，并且要用大括号将它们括起来，如 `{file lnk_file}`，在这个语句中不接受特殊操作符 `*`，`~` 和 `-`。

许可集 一个或多个许可，所有许可对类别集中的客体类别必须是有效的，多个许可之间必须使用空格分隔，并且要用大括号将它们括起来，如 `{read create}`，在这个语句中也不接受特殊操作符 `*`，`~` 和 `-`。

表达式 约束的布尔表达式。

布尔表达式支持下列关键字：

`t1, r1, u1` 分别是源类型，角色和用户

`t2, r2, u2` 分别是目标类型，角色和用户

约束表达式语法还支持下列操作符：

`=` 等于或是...的成员

`!=` 不等于或不是...的成员

`eq` (仅角色关键字) 等于

dom	(仅角色关键字) 优先于
domby	(仅角色关键字) 被... 优先
incomp	(仅角色关键字) 不可比

完整的语义含义和每个操作符接受的变量列在表 7-1 中。
 constrain 语句仅在单个策略和基础载入模块中有效，在条件语句和非基础模块中无效。

constrain 语句让你可以在安全上下文的三个元素(用户, 角色和类型)中做任意限制, 约束表达式比较源(主体)进程的上下文和目标(客体)的上下文, 或直接使用明确的名字进行比较, 如类型或角色标识符。

约束表达式本身可以很复杂, 但实际上很少这么做, 如下面的例子:

```
constrain process transition (u1 == u2) ;
```

让我们仔细来研究一下这个约束语句, 首先, 它只应用到 process 客体类别, 并且只对 transition 许可做了约束, 前面的章节已经讨论过了, 在域转换过程中是需要 transition 许可的, 实际上, 这个约束对域转换做了进一步的限制。

再来看一看约束表达式(u1 == u2), u1 和 u2 分别表示安全上下文的源和目标用户标识符, 因此, 当源和目标用户标识符相同时, 表达式返回的值为 true, 在域转换的情况下, 源就是进程当前的安全上下文, 目标就是进程新的安全上下文。

再完整地看一看前面这条约束语句, 它要求对于所有的域转换, 源和目标用户标识符要保持一致。如何保持一致呢? 回顾一下前面讨论过的访问算法, 当某个进程请求转换许可时, AVC 就将请求转发给安全服务器, 由安全服务器为源-目标-类别决定访问权, 前面的约束将会生效(对于 process 客体类别), 并将会检查源和目标安全上下文中的用户标识符, 如果用户标识符不同, 在将授予的访问许可掩码返回给 AVC 前, 将会先移除代表转换许可的掩码。

再来看另一个例子:

```
constrain process transition (r1 == r2) ;
```

这个约束语句和前面的例子类似, 除了这里约束的是角色标识符外, 关键字 r1 和 r2 分别代表源和目标角色标识符, 这个约束要求在域转换过程中角色标识符不能改变, 和前面要求的不能改变用户标识符一样。

因为这两个约束都是关联到相同的客体类别和许可, constrain 表达式语法允许我们将它们合并为一条约束语句, 如下:

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


这一条约束语句等同于前面那两条约束语句，它们都能起到在域转换过程中限制用户和角色标识符不能改变的效果。

将我们的例子做进一步延伸，在某些情况下，我们想要用户和/或角色标识符在域转换过程中发生变化，如登陆进程需要改变用户和角色标识符到那些登陆的用户和角色，另一个例子是一个允许你改变你的角色的程序，在域转换过程中，它必须要能够改变角色标识符，通常，那样的程序是受信任的进程，我们需要一个方法允许它们改变用户/角色标识符，并且还要确保约束对所有其它的程序都是有效的。

为了实现这个目标，我们首先定义一个方法来识别出那些受信任的可以改变用户或角色标识符的域类型，可以通过类型属性来实现，我们假设在策略中已经定义好了两个属性：privuser 和 privrole。前者与所有能够改变用户标识符的类型关联，后者与那些允许改变角色标识符的类型关联。使用这两个属性，我们可以将前面的约束语句改为：

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

在这两条语句中，t1 指的是源类型（如果使用了 t2，那就指的是目标类型），第一条语句指出，只要源类型具有 privuser 属性，就允许在域转换过程中改变用户标识符，同样，只要源类型具有 privrole 属性，就可以在域转换过程中改变角色标识符。

确定你已经理解了属性是如何影响约束的，回顾一下内核是如何将属性扩展为一个类型列表的，对于内核而言，这些约束实际上一串类型列表而不是单个的属性，要是类型列表或属性在操作符的右边，==操作符的真实意思是“类型列表的一个成员”，同样，!=操作符就意味着“不是类型列表的一个成员”，因此，在我们前面的例子中，t1 == privuser 意味着“如果源类型在具有 privuser 属性的类型列表中”。

注意：对于约束表达式，所有操作符的左边必须是一个允许的关键字，如 u1, u2 等，而不应该是一个类型、属性、角色或用户标识符（或标识符列表），操作符的右边可以是一个关键词，或一个或多个标识符名字。

如果操作符的左边和右边分别是关键字 r1 和 r2，此时，你还可以选择更多的操作符，如 eq, dom, domby 和 incomp，但这些都很少使用，表 7-1 总结了在 constrain 语句可以使用的操作符。

表 7-1. 约束表达式中允许的参数和对应的语义

操作符	左边	右边	语义含义
<u>==</u>	t1	t2	源类型等于目标类型
	t1 (t2)	类型和/或属性名	源（目标）类型是

			由右边的名字指出的类型集的一个成员
	r1	r2	源角色等于目标角色
	r1 (r2)	角色名	源（目标）角色是由右边角色指出的角色集的一个成员
	u1	u2	源用户等于目标用户
	u1 (u2)	用户名	源（目标）用户是由右边用户名指出的用户集的一个成员
!=	t1	t2	源类型不等于目标类型
	t1 (t2)	类型和/或属性名	源（目标）类型不是由右边的名字指出的类型集的一个成员
	r1	r2	源角色不等于目标角色
	r1 (r2)	角色名	源（目标）角色不是由右边角色指出的角色集的一个成员
	u1	u2	源用户不等于目标用户
	u1 (u2)	用户名	源（目标）用户不是由右边用户名指出的用户集的一个成员
eq	r1	r2	源角色等于目标角色，和 $r1 = r2$ 的语义相同
dom	r1	r2	源角色优先于目标角色
domby	r1	r2	目标角色优先于源角色
incomp	r1	r2	源角色和目标角色不可比

7.3 标记转换约束

SELinux 支持另一个约束语句 `validatetrans`，这个语句作为我们将在下一章将要讨论的经过修改的多层安全特性的一部分，使用 `validatetrans` 语句，我们可以进一步控制改变安全上下文的能力，在编写本书时，唯一支持这种约束的客体是文件系统客体，如文件，目录，设备文件等。

和 constrain 语句不同, validate-trans 语句允许你将某个客体的新安全上下文和旧安全上下文联系起来, 和/或第三个尝试重新标记客体的进程的安全上下文联系起来, 因此, 在这个语句中增加了新的关键字, 具体地说就是 t3, r3 和 u3, 它们分别代表进程安全上下文的类型, 角色和用户, *1 关键字代表旧的安全上下文, *2 关键字代表新的安全上下文, validate-trans 语句完整的语法如下。

警告: 小心不要将 constrain 和 validate-trans 语句中关键字弄混淆了, 对于 constrain 语句, t1 代表的是源类型, t2 代表的是目标 (客体) 类型。然而, 在 validate-trans 语句中, t3 才代表的是源进程类型, t1 代表的是旧类型, t2 代表是新类型。

validate-trans 语句语法

[validate-trans](#) 语句通过在进程的新旧安全上下文之间定义基于约束的关系限制改变客体安全上下文的能力, validate-trans 语句完整的语法如下:

validate-trans 类别集 表达式;

类别集 一个或多个受支持的客体类别, 多个客体类别之间使用空格分隔, 并且用大括号将它们括起来, 如 {file lnk_file}, 目前, 还只有文件系统客体类别受支持。

表达式 约束一个布尔表达式。

布尔表达式支持下面的关键字:

t1, r1, u1 分别代表旧类型, 旧角色和旧用户

t2, r2, u2 [分别代表新类型, 新角色和新用户](#)

t3, r3, u3 分别代表进程类型, 进程角色和进程用户

约束表达式语法也支持下面的操作符:

= 是...的一个成员或等于

!= 不是...的成员或不等于

eq (角色关键字) 等于

dom [\(角色关键字\) 优先于](#)

domby (角色关键字) 被...优先

incomp (角色关键字) 不可比

每个操作符完整的语义和可接受的变量列在表 7-2 中。

每个操作符完整的语义和可接受的变量列在表 7-2 中。

validate-trans 语句在单个策略和基础载入模块中有效, 在条件语句和非基础载入模块中无效。

迄今为止, 我们还没有看到使用 validate-trans 约束的例子, 这个语句是作为第 8 章“多层安全”中讨论的多层安全 (MLS) 的姊妹添加进来的, 将来它可能非常有用, 为了帮助理解如何使用这个语句, 我们先来看一个例子, validate-trans 语句的关键特性是允许我们在标记文件客体改变时可以将新旧安全上下文关联起来。

假设在我们的策略中有一个类型叫做 user_tmp_t, 它用于普通的没有经过验证的用户程序的临时文件, 我们想要确保具有特权的域可以改变所有文件标记 (如管理员运行的标记

维护程序），而不是偶尔重新标记类型为 user_tmp_t 的文件到一个更高级的类型（如 shadow_t 类型，它是/etc/shadow 文件的类型），下面是我们的约束提供的也给限制：

```
validatetrans {file lnk_file} ( t2 != shadow_t or t1 !=
user_tmp_t );
```

注意这个约束语句的几个特性，首先，我们使用了一个普通文件和一个符号链接文件（lnk_file），因为我们不想某些人使用链接代替文件，接下来检查约束表达式，说得简单一点，约束表达了对文件和符号链接客体允许的安全上下文改变，如果旧的类型不是 user_tmp_t，新的类型可能只能是 shadow_t，换句话说，没有哪个域类型被授权可以重新标记一个用户临时文件的类型为 shadow 密码文件的类型。

为了扩展这个例子，假设有一个我们想允许重新标记 user_tmp_t 到 shadow_t 域类型的子集，我们创建一个属性 relabel_any，然后将它分配给那些我们想赋予这个特权的域类型，然后，我们可以扩展这个约束，如下：

```
validatetrans {file lnk_file}
( ( t3 == relabel_any) or
( t2 != shadow_t or t1 != user_tmp_t ) );
```

这样我们就一套域类型（具有 relabel_any 属性）不受这个约束限制了。

表 7-2. validatetrans 表达式可接受的参数及其语义

操作符	左边	右边	语义含义
=	t1	t2	旧类型等于新类型
	t1 (t2)	类型和/或属性名	旧（新）类型是由右边名字标识的类型集的一个成员
	t3	类型和/或属性名	进程类型是由右边名字标识的类型集的一个成员
	r1	r2	旧角色等于新角色
	r1 (r2)	角色名	旧（新）角色是由右边名字标识的角色集的一个成员
	r3	角色名	进程角色是由右边名字标识的角色集的一个成员
	u1	u2	旧用户等于新用户
	u1 (u2)	用户名	旧（新） 用户是由右边名字标识的用户集

			的一个成员
	u3	用户名	进程用户是由右边名字标识的用户集的一个成员
!=	t1	t2	旧类型不等于新类型
	t1 (t2)	类型和/或属性名	旧（新）类型不是由右边名字标识的类型集的一个成员
	t3	类型和/或属性名	进程类型不是由右边名字标识的类型集的一个成员
	r1	r2	旧角色不等于新角色
	r1 (r2)	角色名	旧（新）角色不是由右边名字标识的角色集的一个成员
	r3	角色名	进程角色不是由右边名字标识的角色集的一个成员
	u1	u2	旧用户不等于新用户
	u1 (u2)	用户名	旧（新）用户不是由右边名字标识的用户集的一个成员
	u3	用户名	进程用户不是由右边名字标识的用户集的一个成员
eq	r1	r2	等同于= 的语义
	r1 (r2)	角色名	等同于=的语义
dom	r1	r2	源角色优先于目标角色
domby	r1	r2	目标角色优先于源角色
incomp	r1	r2	源和目标角色不可比

7.4 小结

约束为某个特定的许可提供了全局限制，不管策略中的 allow 规则这么定义都不行。

constrain 语句使我们可以基于源和目标类型、角色和用户标识符之间的关系限制授予的许可。

validatetrans 语句使我们可以基于旧的、新的进程类型、角色和用户标识符之间的关系限制改变客体安全上下文的能力，目前只有文件系统客体支持这个语句。

7.5 练习

1、将 96 页列出的那两条约束语句使用一条约束语句写出来。

2、一个常见的 neverallow 恒定规则是：

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

编写一条约束尽可能关闭这个恒等式的等价含义。

3、回顾 93 页的 validatetrans 示例语句：

```
validatetrans {file lnk_file}

( ( t3 == relabel_any) or

  ( t2 != shadow_t or t1 != user_tmp_t ) );
```

假设你想添加大量的其它类型到那些你没有重新标记的类型为 user_tmp_t 的文件中，你要如何修改这个约束语句才能实现这个目标？

第 8 章 多层安全

在最近对 SELinux 的增强中，约束特性已经被扩展，实现了一个可选的多层安全 (multilevel security，简写 MLS) 策略，MLS 是另一种强制访问方式，它基于类型强制 (type enforcement，简称 TE)，在这一章中，我们要研究的就是这个可选的 MLS 策略特性。

8.1 多层安全约束

MLS 是另一种强制访问控制方法，特别适合于政府机密数据的访问控制，早期对计算机安全的研究大多数都是以在操作系统内实现 MLS 访问控制为驱动的，SELinux 为 MLS 提供了可选的支持，即使类型强制保留了 SELinux 的基础访问控制机制，我们也还是可以开启 MLS 特性，提供额外的 MLS 风格的强制访问控制。在 SELinux 中，MLS 是类型强制的一个可选扩展，如果没有类型强制就没有 MLS。

注意：Fedora Core 5 (FC 5) 默认开启了 MLS，在 FC 5 中，MLS 是用来实现所谓的多范畴安全而不是传统的 MLS 策略模块，这两种使用方式显示了 SELinux 的灵活性，但不管怎样，所有 MLS 的使用都是建立在 TE 安全的基础之上的。

我们可以通过创建一个表示 MLS 策略的二进制内核策略文件来开启 SELinux 中的 MLS 特性，主要的方法是在使用 checkpolicy 程序编译策略时，使用 -M 选项，使用这个选项时，

checkpolicy 将会创建一个开启 MLS 的内核策略, 当它载入内核时, 内核将会执行额外的 MLS 约束, 你将会看到一个可用的策略源构建树 (示例和参考策略可以参考第 11 章“原始示例策略”和第 12 章“参考策略”), 通过 Makefile 或配置文件来管理 MLS 特性。

注意: 在本书准备发行时, Tresys 发布了一个新版的 apol (SeTools 2.4 版), 已经可以支持检查 MLS 安全上下文和规则了, 本章我们不打算涉及那些新特性, 只要你熟悉了 apol 后, 这些新功能使用起来非常简单。

8.2 开启了 MLS 后的安全上下文

正如第 2 章“概念”中讨论的那样, 在开启了 MLS 后, 安全上下文扩展了两个字段: 低安全级别和高安全级别。每个安全级别本身有两个字段: 灵敏度和一套分类。灵敏度有着严格的分级, 它反应了一个有序的数据灵敏度模型, 如政府分类控制中的绝密, 机密和无密级。分类是无序的, 它反应的是数据划分的需要。基本思路是对于要访问的数据你同时需要足够的灵敏度和正确的分类。

警告: 不要将安全级别和灵敏度弄混了, 一个安全级别是一个灵敏度和一套分类的集合。灵敏度有着严格的分级, 可以使用等价关系符 (<, =, >) 进行比较; 安全级别是没有分级的, 可以使用控制关系运算符 (dom, domby, eq, incomp) 进行比较。

8.2.1 安全级别定义

在 SELinux 策略中, 使用 sensitivity 语句定义灵敏度, 如:

```
sensitivity s0;
sensitivity s1;
sensitivity s2;
sensitivity s3;
```

这些语句定义了四个灵敏度, 分别是 s0, s1, s2 和 s3。你可以随意给灵敏度命名, 还可以在定义灵敏度时给它起别名, 别名的待遇和正式名称一样, 如:

```
sensitivity s1 alias unclassified;
```

注意: 最近对 SELinux 做了一些改进, 包括在 FC 5 中, 已经添加一个实用程序 semanage, 它可以使你给策略灵敏度和分类起一个易读 (和可打印) 的名字, 包含可打印的映射的文件是 /etc/selinux/[policy]/setrans.conf, 这里的 [policy] 表示一个已经安装的策略。

因为灵敏度是分层的, 我们必须在策略中使用 dominance 语句指定灵敏度的分级, 如:

```
dominance { s0 s1 s2 s3 } # s0 表示低, s3 表示高
```

dominance 语句按从低到的高的序列出了灵敏度的名字，因此，在我们的例子中，s0 比 s1 低，s1 比 s2 低，以此类推。

警告：在 dominance 语句结尾是没有分号的（即使其他大部分策略语句都是以分号结尾的），在这种情况下，一个右大括弧就表示语句的结尾，这一点你要小心。

分类的定义方法与灵敏度类似，它使用 category 语句进行定义，分类也可以有别名。但与灵敏度不同的是，分类是没有分级的，因此，不需要在分类之间明确定义任何关系，如：

```
category c0 alias blue;  
category c1 alias red;  
category c2 alias green;  
category c3 alias orange;  
category c4 alias white;
```

最后一步是使用 level 语句定义安全级别，level 语句规定了灵敏度如何与分类进行关联，记住一个安全级别是由一个灵敏度加上一套分类组成的，如：

```
level s0:c0.c4;  
level s1:c0.c4;  
level s2:c0.c4;  
level s3:c0.c4;
```

在 level 语句中，可以灵活组织和搭配已经定义好的灵敏度和分类。

在前面这个例子中，我们将定义的所有灵敏度和分类都进行关联，在这个关联中还可以做更多的限制，如：

```
level s0:c0.c2;  
level s1:c0.c2,c4;
```

在这个例子中，s0 可能只与分类 c0，c1 和 c2 关联，s1 与 c0，c1，c2 和 c4 关联，但没有 c3，现在你应该已经知道这里的点（.）表示一个分类的范围了，而逗号（,）表示一个非连续的分类列表了。

警告：由于分类范围是通过范围操作符（.）来指定的，这并不意味着分类是有分级的，相反，范围操作符只不过是一个引用一套分类的简便方法，分类的顺序也仅仅表示声明时的一个先后次序，并没有其他含义。

因此，如果你按 c1，c0 和 c2 的顺序声明的分类，表达式 c0.c2 等于 c0 和 c2，并不包含 c1。

level 语句定义了灵敏度和分类的关联，共同组成一个合格的安全级别，成为 SELinux 中 MLS 的一部分。

安全级别语句语法

在 SELinux 策略中总共有四个语句定义安全级别，它们的完整语法如下：

sensitivity 语句

这个语句定义了策略灵敏度标识符和可选的别名标识符

sensitivity 标识符 [alias 别名 id1 别名 id2];

标识符 灵敏度字符串标识符

别名 id 一个或多个灵敏度别名标识符字符串

灵敏度标识符和灵敏度别名标识符在策略中可以相互代替使用。

dominance 语句

这个语句定义了所有灵敏度之间的分级关系

dominance {标识符 标识符 标识符}

标识符 由 sensitivity 语句定义的灵敏度标识符

灵敏度的顺序按从低到高的顺序指定，为了定义完整的灵敏度等级，所有定义的灵敏度必须包括在控制语句内。

category 语句

这个语句定义策略分类标识符和可选的别名标识符

category 标识符 [alias 别名 id 别名 id];

标识符 分类标识符字符串

别名 id 一个或多个分类别名标识符字符串

分类标识符及其别名标识符在策略中具有同等地位。

level 语句

这个语句定义了灵敏度和一套分类的集合

level 灵敏度[:分类集];

灵敏度 一个已经定义好的灵敏度标识符

分类集 一套已经定义好的分类标识符，分类标识符可以使用逗号分隔，也可以使用范围操作符 (.) 指定一个分类范围，如分类集 c0. c3, c5 意思是包含所有从 c0 到 c3 定义的分类，再加上 c5，注意这里没有隐含按照分类名称进行排序的意思，相反，范围操作符使用的顺序是定义标识符时的顺序。

你可以为每个灵敏度使用一条 level 语句，分类集是可选的，未指定的分类集等于“空”分类集，即该灵敏度没有关联任何分类，一个有效的安全上下文可能只关联了一个灵敏度和一套分类集。

level 语句在单个策略语句和基础载入模块中有效，在条件语句和非基础载入模块中无效。

8.2.2 MLS 对安全上下文的扩展

对于 MLS SELinux 系统，安全上下文被扩展，新加入两个安全级别：低或当前的安全级别和高安全级别。通常，低安全级别反应的是进程当前的安全级别，或包含在客体内的数据的灵敏度；高安全级别反应的是上下文中用户标识符的许可证级别或那些所谓多层客体允许的数据最大范围，当 MLS 开启时，对安全上下文的扩展格式如下：

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

注意安全级别只需要一个灵敏度，可以不要分类或搭配多个分类，即分类是可选的，此外，在指定安全上下文时不需要指定高级级别，如果未指定，高级就等于低级，对于客体而言，这很常见。

对于一个有效的安全上下文，高级级别必须优先于低级级别，此外，与灵敏度关联的分类也必须是有效的，假设我们使用前面的 level 语句：

```
level s0:c0.c2;  
level s1:c0.c2,c4;
```

和 user_u, user_r 和 user_t，这些都是有效的用户、角色和类型标识符，下面的安全上下文是无效的：

```
user_u:user_r:user_t:s0-s0:c2,c4  
(c4 is invalid for s0)  
user_u:user_r:user_t:s0:c0-s0:c2  
(high does not dominate the low)
```

8.3 MLS 约束

SELinux 支持两个 MLS 约束语句：mlsconstrain 和 mlsvalidatetrans。它们两个一起让我们可以指定可选的 MLS 访问强制规则，除了它们允许你基于安全上下文的安全级别表示约束外，这两个语句在无 MLS 的副本中是等同的。你可能只能在开启了 MLS 特性的策略中使用 MLS 约束，你可以在任何策略类型中使用无 MLS 的约束语句。

8.3.1 mlsconstrain 语句

mlsconstrain 语句是以 constrain 语句为基础的，我们可以使用第 7 章“约束”中讨论的 constrain 语句的所有语法，为了描述基于源（l1 和 h1）和目标（l2 和 H2）的低和高安全级别的约束，mlsconstrain 语句添加了新的关键词。下面是 mlsconstrain 语句的完整语法：

mlsconstrain 语句语法

[mlsconstrain](#) 语句允许你限制特定客体类别指定的许可，通过基于源和目标安全上下文之间的关系定义约束，源和目标安全上下文包括了 MLS 特性（即高和低安全级别），完整的 mlsconstrain 语句语法如下：

mlsconstrain 类别集 许可集 表达式；

类别集 一个或多个客体类别。多个客体类别必须使用空格分隔，并用一对大括号将它们括起来，如 {file lnk_file}，在这个语句中，类别集不接受*，~和-特殊操作符。

许可集 一个或多个许可。所有的许可对类别集中指定的客体类别都要有效，多个许可必须使用空格分隔，并用一对大括号将它们括起来，如 {read create}，在这个语句中，类别集不接受*，~和-特殊操作符。

表达式 一个布尔表达式。

布尔表达式语法支持下列关键字：

t1, r1, u1, l1, h1	分别表示源类型、源角色、源用户、 源低级别和源高级别 。
t2, r2, u2, l2, h2	分别表示目标类型、目标角色、目标用户、目标低级别和目标高级别。
约束表达式语法也支持下列特殊操作符：	
==	是...的一个成员或等于。
!=	不是...的成员或不等于。
eq	（只用于角色和安全级别）等于。
dom	（只用于角色和安全级别）优先于
domby	（只用于角色和安全级别）被...优先
incomp	（只用于角色和安全级别）不可比
这些操作符完整的语义含义和它们可接受的参数放在表 8-1 中了，表 8-1 是根据表 7-1 制作的。	
mlsconstrain 语句仅在非强制 MLS 策略中受到支持。	
mlsconstrain 语句仅在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。	

为了解释 mlsconstrain 语句，我们来看一个将 MLS 应用给普通文件系统对象的例子，因为是一个简单示例，我们假设那个文件对象只有一个安全级别，即高和低安全级别是相同的，我们可以使用下面这样一条约束语句来实现这个约束：

```
mlsconstrain file { create relabelto }
( l2 eq h2 );
```

假设 create 和 relabelto 是为设置文件客体安全级别需要的 file 许可，这个约束足以要求所有的文件高安全级别等于低安全级别。

接下来我们来看一看更核心的 MLS 策略约束，回顾一下第 2 章中关于 MLS 的基础前提，即阻止信息从高安全级别滑向低安全级别或无法比较的安全级别，我们通过在所有客体上执行“不能读就不能写”的规则来实现这个要求，在 SELinux 中，低安全级别通常表示进程或客体的当前安全级别，因此，对于文件我们使用以下 MLS 约束：

```
mlsconstrain file write ( l1 domby l2 );
```

在这个语句中，我们对 file 客体类别限制了 write 许可，要求客体安全级别（l2）优先源安全级别（l1），换句话说就是进程仅在它当前的安全级别上可以写文件。

遗憾的是这个约束太简单而不能保证 MLS 策略对文件客体有效，首先，我们考虑一下文件客体类别许可，除了 write 外还有许多许可允许向文件写入信息，如 append 许可也允许往文件中写入信息（追加在文件末尾），还有 rename 许可，为了充分理解，我们需要扩展我们的约束，覆盖所有的“有能力写”文件许可：

```
mlsconstrain file { write create setattr relabelfrom
```

```
append
unlink link rename mounton }
( l1 domby l2 );
```

在这个语句中除了 write 许可外，还列出了一系列的许可，它们都可以以某种形式往客体写入信息，约束表达式仍然相同。

这个约束语句仍然相当简单，我们需要指出受信任的域类型在何处，并且要给它提供特殊的许可，以跳过“不能写”规则，即使你应该避免使用这种受信任的域，但实际上，几乎所有的 MLS 系统应用程序都会使用它们，为了适应这个原理，我们需要扩展这些约束语句让它们接受这些受信任的域。

为了实现这些受信任的降级域，可以创建一个类型属性，叫做 mlsfilewritedown，它可以识别出所有受信任的域，现在，我们的约束语句如下：

```
mlsconstrain file { write create setattr relabelfrom
append
unlink link rename mounton }
( ( l1 domby l2 ) or
( t1 == mlsfilewritedown ) );
```

这个约束语句允许一个例外，就是任何属性为 mlsfilewritedown 的源域（t1），这就是所谓就信任的域了。

对于一个完整的 MLS 策略，我们也需要限制读权限，和写权限一样，除了 read 许可外，也有大量的许可是允许“读”访问权的，如 execute 许可肯定会允许进程读取可执行文件的内容的，下面是一个对文件客体的 MLS 读约束：

```
mlsconstrain file { read getattr execute }
( ( l1 dom l2 ) or
( t1 == mlsfilewritedown ) );
```

在这个约束语句中，它允许读取那些有特权的属性包含有 mlsfilereadup 的域类型的特权。

在编写一个完整的 MLS 策略时，你需要检查所有的客体类别和它们的许可，确保读和写限制是正确约束的。如在前面的读约束语句中，我们可能想在单条语句中标出所有的文件系统客体，如：

```
mlsconstrain { dir file lnk_file chr_file
blk_file sock_file fifo_file }
```

```
{ read getattr execute }
( ( l1 dom l2 ) or
( t1 == mlsfilereadup ) );
```

对于一个给定的 SELinux 策略，你会发现 MLS 约束通常是在一个源策略文件中的，一般都叫做 mls，除了这一章外，我们打算过多地谈及 SELinux 的 MLS 特性，如果你对这方面的东西感兴趣，你可以直接打开这个文件研究研究。

表 8-1. Mlsconstrain 表达式接受的参数对应的语义含义（参考表 7-1）

操作符	左边	右边	语义含义
==	l1	l2, H1, H2	源的低安全级别（当前安全级别）等于目标的低安全级别 l2 ，源的高安全级别（H1）或目标的高安全级别（H2）
	l2	h2	目标的低安全（当前）级别等于目标的高安全级别
	h1	l2, h2	源的高安全级别等于目标的低安全级别或高安全级别
!=	l1	l2, h1, H2	源的低安全（当前）级别不等于目标的低安全级别 l2，源的高安全级别（H1）或目标的高安全级别（H2）
	l2	H2	目标的低安全（当前）级别不等于目标的高安全级别
	h1	l2, h2	源的高安全级别不等于目标的低安全级别或高安全级别
eq	l1	l2, H1, H2	和== 的语义完全一样
	l2	h2	和==的语义完全一样
	h1	l2, h2	和==的语义完全一样
dom	l1	l2, H1, H2	源的低安全级别（当前安全级别）优先于目标的低安全级别 l2 ，源的高安全级别（H1）或目标的高安全级别（H2）
	l2	h2	目标的低安全（当前）级别优先于目标的高安全级别
	h1	l2, h2	源的高安全级别优先于目标的低安全级别或高安全级别
domby	l1	l2, H1, H2	目标的低安全级别（当前安全级别）优先于源的低安全级别 l2，源的高安全级别（H1）或目标的高安全级别（H2）
	l2	h2	目标的高安全级别优先于目标的低（当前）安全级别
	h1	l2, h2	目标的低安全级别或高安全级别优先于源的高安全级别
incomp	l1	l2, H1, H2	源的低安全级别（当前安全级别）与目标的低安全级别 l2，源的高安全级别（H1）或目标的高安全级别（H2）

			不可比
	l2	h2	目标的低安全（当前）级别与目标的高安全级别不可比
	h1	l2, h2	目标的低安全级别或高安全级别与源的高安全级别不可比

8.3.2 mlsvalidatetrans 语句

我们还有一个 MLS 约束语句需要讨论，第 7 章我们已经讨论了 `validatetrans` 语句的变种 `mlsvalidatetrans`，除了它新引入了几个关键字 `l1` 和 `h1`，`l2` 和 `h2`，`l3` 和 `h3` 外，它和 `validatetrans` 非常类似，这几个关键字分别代表旧的低安全级别和高安全级别，新的低安全级别和高安全级别，源进程低安全级别和高安全级别。它们之间另一个不同之处是在 TE 策略中 `mlsvalidatetrans` 语句比 `validatetrans` 语句更常用。下面是完整的 `mlsvalidatetrans` 语句语法：

mlsvalidatetrans 语句语法

`mlsvalidatetrans` 语句基于旧和新的安全上下文以及源进程的安全上下文之间的关系定义限制指定客体改变安全上下文的能力的约束。它的完整语法定义如下：

`mlsvalidatetrans` 类别集 表达式；
类别集 一个或多个受支持的客体类别，多个客体类别必须使用大括号括起来，如 `{file lnk_file}`，目前只支持永久性文件系统客体类别。

表达式 约束的布尔表达式。

这里的布尔表达式支持下面的关键字：

`t1`, `r1`, `u1`, `l1`, `h1` 分别代表旧类型、旧角色、旧用户、旧低级别和旧高级别。

`t2`, `r2`, `u2`, `l2`, `h2` 分别代表新类型、新角色、新用户、新低级别和新高级别。

`t3`, `r3`, `u3`, `l3`, `h3` 分别代表进程类型、进程角色、进程用户、进程低级别和进程高级别。

约束表达式也支持下列操作符：

`==` 设为...的一个成员或等于。

`!=` 设为不是...的成员或不等于。

`eq` （角色和安全级别关键字）等于

`dom` [（角色和安全级别关键字）优先于](#)

`domby` （角色和安全级别关键字）被...优先

`incomp` （角色和安全级别关键字）不可比

表 8-2 列出了这些操作符完整的语义含义和可接受的参数。

`mlsvalidatetrans` 语句仅在非强制 MLS 策略中受到支持。

`mlsvalidatetrans` 语句在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。

我们通常不想要文件的安全级别被改变，经过多年的 MLS 系统运行经验，我们已经认识到有些 MLS 应用程序需要进化成受信任的应用程序以便可以改变现有客体如文件的安全级别，为了对那些受信任的应用程序实施限制，就可以使用 mlsvalidatetrans 约束语句，如：

```
mlsvalidatetrans file
( ( l1 eq l2 ) or
  (( t3 == mlsfileupgrade ) and ( l1 domby l2 )) or
  (( t3 == mlsfiledowngrade ) and
    ( l1 dom l2 or l1 incomp l2 )) );
```

这个约束语句有许多特性，首先，当某个文件客体的安全上下文被改变时它有基本的要求，它当前的（低）安全级别和高安全级别必须是相同的（l1 等于 l2），然而，它也提供了升级（mlsfileupgrade 属性）和降级（mlsfiledowngrade 属性）特权，如果进程域类型有 mlsfileupgrade 属性，就允许升级（即新安全级别 l2 优先于旧安全级别 l1）；同样，如果进程域类型有 mlsfiledowngrade 属性，就允许降级（即旧安全级别优先或两者不可比）。

表 8-2. Mlsvalidatetrans 可接受的参数及其对应的语义含义（加上那些定义在表 7-2 中的 Validatetrans 语句）

操作符	左边	右边	语义含义
==	l1	l2, h1, h2	旧的低（当前）安全级别等于新的低安全级别、旧的高安全级别或新的高安全级别
	l2	h2	新的低（当前）安全级别等于新的高安全级别
	h1	l2, H2	旧的高安全级别等于新的低安全级别、或新的高安全级别
!=	l1	l2, h1, h2	旧的低（当前）安全级别不等于新的低安全级别、旧的高安全级别或新的高安全级别
	l2	h2	新的低（当前）安全级别不等于新的高安全级别
	h1	l2, H2	旧的高安全级别不等于新的低安全级别、或新的高安全级别
eq	l1	l2, h1, h2	旧的低（当前）安全级别等于新的低安全级别、旧的高安全级别或新的高安全级别
	l2	h2	新的低（当前）安全级别等于新的高安全级别
	h1	l2, H2	旧的高安全级别等于新的低安全级别、或新的高安全级别

dom	l1	l2, h1, h2	旧的低（当前）安全级别优先于新的低安全级别、旧的高安全级别或新的高安全级别
	l2	h2	新的低（当前）安全级别优先于新的高安全级别
	h1	l2, H2	旧的高安全级别优先于新的低安全级别、或新的高安全级别
dom by	l1	l2, h1, h2	旧的低（当前）安全级别被新的低安全级别、旧的高安全级别或新的高安全级别优先
	l2	h2	新的低（当前）安全级别被新的高安全级别优先
	h1	l2, H2	旧的高安全级别被新的低安全级别、或新的高安全级别优先
inc omp	l1	l2, h1, h2	旧的低（当前）安全级别与新的低安全级别、旧的高安全级别或新的高安全级别不可比
	l2	h2	新的低（当前）安全级别与新的高安全级别不可比
	h1	l2, H2	旧的高安全级别与新的低安全级别、或新的高安全级别不可比

注意：记住，在编写本书时，`validatetrans` 和 `mlsvalidatetrans` 约束语句只支持文件系统客体，具体是 `dir`, `file`, `lnk_file`, `chr_file`, `blk_file`, `sock_file` 和 `fifo_file` 客体类别。

8.4 MLS 的其它影响

本章描述了在 SELinux 中定义 MLS 策略的基本机制，然而，没有描述一个完整的 MLS 策略，与类型强制不一样，它更灵活，适应能力也很强，MLS 是用来严格限制和执行单条安全不变量的（即 no write down, no read up），这种方法对于保护敏感数据（如国家机密）是很重要的，然而，它也提出了许多挑战，你必须跳出本书扮演一个安全系统的设计者。

因为 SELinux 中的 MLS 特性扩展了安全上下文，无论在什么地方使用安全上下文都需要指定安全级别信息了，其中一个受影响的就是第 6 章讨论的 `user` 语句，对于一个 MLS 系统，所有用户都必须要有有一个定义好的许可证安全级别（即高安全级别），它代表最高级别进程用户，`user` 语句的语法就被改为：

`user` 用户名 `roles` 角色集 `level` 默认级别 `range` 允许的范围；

这里的用户名和角色集参数和之前的一样，但添加了两个新的关键字，一个定义用户登陆时默认的安全级别（`level`），另一个定义了用户登陆或可以运行的进程的安全级别的范

围(range)，默认级别是一个单一有效安全级别，允许的范围是从低到高的安全级别范围，如：

```
user joe roles user_r level s0 range s0 - s3:c0.c4;
```

这个语句给用户 joe 分配的默认安全级别是 s0（我们早先定义的最低的敏感度，没有分类），允许用户从无分类的安全级别 s0 到有分类 c0 到 c4 的安全级别 s3 的任何安全级别登陆，如用户可以以 s1:c1.c2 安全级别登陆，但不允许以 s4:c0 安全级别登陆，因为这个级别没有在允许的范围内。

MLS 影响的其它主要区域是使用一个安全上下文标记一个客体的地方，在第 10 章“客体标记”中，我们详细地讨论了无 MLS 的系统中的客体标记，你只需要记住，在 MLS 系统中，要扩展客体安全上下文，使其包括低和高安全级别，你会发现在使用 MLS 系统时真正的挑战是为每个客体分配合适的安全级别。

8.5 小结

SELinux 策略语言通过使用额外的约束语句和对安全上下文的扩展提供了对 MLS 的非强制支持。

对于一个 MLS 策略，你必须定义分级敏感度和无分级的分类，一个有效的安全级别是由一个敏感度和一套分类（包括空集）组成的。

对 MLS 而言，安全上下文被一个低（当前）和一个高（许可证）安全级别扩展了，高安全级别总是优先于低安全级别，这是硬编码进 MLS 的。

MLS 策略的主要用途是为了对所有客体实现“不能读就不能写”（即 no read down, no write up）恒定规则，我们可以使用 mlsconstrain 语句实现这个要求，它和 constrain 非常类似，除了它还允许基于源和目标安全级别之间的关系建立约束外，其它就没什么区别了。

mlsconstrain 语句和 validatetrans 语句也很相似，除了它还允许我们基于旧的、新的和进程安全级别限制安全上下文被改变外，其它就没什么区别了。这样我们就可以控制文件系统客体安全级别的是否可以被更改了。

对于一个完整的 MLS 安全策略，你必须在所有有关的客体类别许可上实现 MLS 约束，并对安全上下文的内容进行扩展。

8.6 练习

1、假设有下列敏感度和分类定义：

```
sensitivity s0;  
sensitivity s1;
```

```
sensitivity s2;

category c0;
category c1;
category c2;
category c3;
category c4;

level s0;
level s1:c0.c2;
level s2:c0.c4;
```

同样也假设 user_u, user_r 和 user_t 是有效的用户，角色和类型标识符，选择下列哪个安全上下文是有效的，并解释为什么或为什么不是？

- A、user_u:user_r:user_t:s0-s0:c0
- B、user_u:user_r:user_t:s0-s1
- C、user_u:user_r:user_t:s0-s1:c0.c4
- D、user_u:user_r:user_t:s1:c0.c2-s2:c0.c1
- E、user_u:user_r:user_t:s1-s2:c0,c4

2、有如下的 MLS 约束语句：

```
mlsconstrain file { write create setattr relabelfrom
append
unlink link rename mounton }
( ( l1 domby l2 ) or
( t1 == mlsfilewritedown ) );
```

这个约束语句限制“write down”，但它允许所有域“write up”，还没有 MLS 约束语句限制“write up”，因为它不会使信息安全级别降级，经常使用它建立 MLS 安全应用程序，尽管如此，还是有一些 MLS 系统开发人员喜欢提供一个特权像“write down”那样控制“write up”，作为一个练习，请将前面这个约束语句改为可以同时控制 write down 和 write up。

第 9 章 条件策略

在这一章中，我们探讨一下条件策略，它允许我们基于不同的情况定义规则启用或禁用，这一章我们主要讨论的是支持条件策略的 SELinux 策略语言，以及条件策略的使用。

9.1 条件策略概述

对条件策略的支持是 SELinux 策略语言第一次发布以来最主要的功能改进了，条件策略语句使我们可以定义仅在由条件表达式定义的环境下的策略规则集，条件表达式是一个使用变量和逻辑运算符的逻辑表达式。

让我们一起来看一个例子，假设我们有一台移动电脑，我们想定义一些可以访问某个特定程序的域类型（如 myprog_t）的策略规则，以便这台电脑接入网络时，只能通过有线以太网接口进行访问，当它断开网络时就使用无线网络接口，为了实现这个目标，我们可能需要编写一条条件语句，如：

```
bool docked true;
if (docked) {
    #允许 my_prog_y 访问有线以太网设备的规则
} else {
    #允许 my_prog_y 访问无线设备的规则
}
```

在这个例子中，我们首先声明了一个布尔变量 docked，使用这个布尔变量将设备是否接入网络的情况告诉 SELinux，作为声明的一部分，我们赋予布尔变量 docked 一个默认值 true，然后，我们创建了一条条件语句（if），它包括一个条件表达式（docked）和一个真值，以及一个可选的假值列表，这个语句允许我们为每种情况编写 allow 规则（即设备接入网络时和移除网络时），我们需要做的是在移动电脑接入网络和移除网络时改变这个布尔变量的值，如利用一个运行的服务监控这个状态，并在状态发生变化时对布尔变量的值做相应的改变。

这个简单的例子说明了条件策略的主要特征，在本节剩下的部分，我们讨论如何定义和改变布尔变量，列出条件语句的语法，以及展示一些使用条件策略的例子。

9.2 布尔变量

使条件策略起作用的是条件表达式，条件表达式是通过使用一个或多个布尔变量和逻辑运算符一起形成的，然后通过改变布尔变量的值来影响条件表达式的值，从而改变条件语句中的规则集，因此，编写条件策略的第一步是创建布尔变量。

9.2.1 布尔变量定义

我们使用 bool 语句定义布尔变量，例如：假设我们想通过配置策略让普通用户可以使用 ping 命令，但可以通过开关进行控制，对于这样的情况，我们需要定义一个布尔变量 user_ping，我们将在条件表达式中使用它，使用下面的语句定义这个变量：

```
bool user_ping false; #控制用户是否可以使用 ping 命令
```

bool 语句有两个参数：布尔变量的名称（user_ping）和它的默认值，默认值可以是 true 或 false。在这里的默认值是 false，意味着普通用户默认情况下不能使用 ping 命令。下面是 bool 语句完整的语法介绍：

bool 语句语法

bool 语句定义条件布尔变量和它的默认值，完整的 bool 语句语法如下：

bool 布尔变量名称 默认值；

布尔变量名称 布尔变量的标识符，它的长度不受限制，可以包括 ASCII 字符，数字和下划线，必须以 ASCII 字符开头。

默认值 布尔变量的默认值，要么是 true 要么是 false。

bool 语句在单个策略，基础载入模块和非基础载入模块中有效，在条件语句中无效。

9.2.2 在运行系统中关联布尔变量

在系统运行时改变布尔变量的值使我们可以改变条件表达式的值，因此就有了条件策略，因此 SELinux 内核使布尔变量的改变对运行中的进程有效是必需的，这与策略中的其它组件不同，其它组件一般是一旦载入内核就是静态的了，直到有全新的策略载入，在运行状态的系统上，布尔变量是容易获取和改变的。

内核通过 selinux 伪文件系统将布尔变量暴露出来，这个伪文件系统是用户空间和 SELinux Linux 安全模块 (LSM) 之间主要的接口，它通常挂载到 /selinux/，当前策略中定义的所有布尔变量都将在这个伪文件系统的 booleans 目录下显示为一个个的文件，因此，你可以查看上面定义的布尔变量 user_ping 的值，cat /selinux/Booleans/user_ping。

我们使用 selinux 文件系统中的布尔文件查询和布尔变量的当前值，如果你看过布尔文件的内容，你会发现它的内容总是一对数字（0 代表 false，1 代表 true），如：

```
# cat /selinux/booleans/user_ping
1 1
```

第一个值表示布尔变量的当前值，第二个值表示布尔变量待定的值，当前值是内核真正正在使用的值，也就是由它决定了条件表达式的值，待定值是布尔变量在改变并提交后的值。

我们通过改变布尔变量的待定值来改变它的当前值，然后将改动提交给内核，通过往布尔文件中写入一个 1 或一个 0 来改变待定值，如：

```
# cat /selinux/booleans/user_ping    #当前值和待定值都是 1
1 1

# echo 0 > /selinux/booleans/user_ping # 往文件中写入一个 0
# cat /selinux/booleans/user_ping      # 待定值改为 0
1 0
```

正如你所看到的，待定值以及变为 0 了，意味着 false，当前值仍然保留不变，这意味着布尔变量 user_ping 的值仍然是 true (1)，即使你将它的待定值改为 false (0) 了，原因是改变布尔值需要提交两步，第一次是改变的待定值，然后还要再提交一次，将待定值提交为当前值，这样你就可以改变多个布尔值，然后一次提交所有的改变。

文件 `/selinux/commit_pending_bools` 是提交所有布尔变量待定值为当前值的接口，当你向这个文件写入一个 1 时就会触发提交事件，如：

```
# echo 1 > /selinux/commit_pending_bools # 提交所有的待定值
# cat /selinux/booleans/user_ping
0 0
```

第一个命令是将 1 写入 `commit_pending_bools` 文件，它触发内核将所有布尔变量的当前值改为它们的待定值，你可以通过检查 `user_ping` 布尔变量来检验一下是否已经第二次提交了，这个布尔变量的当前值现在已经和待定值一样，都是 `false (0)` 了，默认的待定值总是等于当前值。

为了将这个布尔变量的值重设为 `true`，我们只需要反向做就可以了：

```
# echo 1 > /selinux/booleans/user_ping # 设置待定值为 true
# cat /selinux/booleans/user_ping      # 查看已经改变的待定值
0 1
# echo 1 > /selinux/commit_pending_bools # 提交待定值
# cat /selinux/booleans/user_ping      # 查看已经改变的当前值
1 1
```

SELinux 提供了方便的命令来查询和改变布尔值，这样就不用再记那些文件的位置了，`getsebool` 命令显示布尔值的状态为 `active (true)` 或 `inactive (false)`，如：

```
# getsebool user_ping
user_ping > active
```

注意：最近对 SELinux 的增强，已经包括 Fedora Core 5 (FC 5)，在使用 `getsebool` 命令时，返回的值发生了一点变化，从原来的 `active` 和 `inactive` 变为更直观的 `on` 和 `off` 了。

要查看当前系统中所有的布尔变量和它们的状态，可以使用 `-a` 选项，如：

```
# getsebool -a
docked > inactive
user_ping > active
...
```

我们可以使用 `setsebool` 命令来更改这些布尔变量的值，如：

```
# getsebool user_ping          # 显示当前状态
user_ping > active

# setsebool user_ping false    # 改变并提交最新状态
# getsebool user_ping          # 显示改变后的状态
user_ping > inactive
```

注意 `setsebool` 命令同时改变了挂起的状态并将更改提交为当前最新状态。我们也可以在一
条约束语句中同时改变多个布尔值，如：

```
# getsebool user_ping docked    # 显示当前状态
user_ping > active
docked > inactive

# setsebool user_ping=0 docked=1 # 同时改变两个布尔变量
的值
# getsebool user_ping docked    # 显示当前状态
user_ping > inactive
docked > active
```

警告：你系统上定义的布尔变量依赖于当前载入内核的策略，很可能你看到的布尔变量和我们这里专门设计的变量有所不同，千万不要昏倒！如果你愿意，你可以使用本书中介绍的布尔变量作为练习。

9.2.3 对布尔值的永久性改变

正如前面讨论的，布尔变量和它们的默认状态都定义在一个策略文件中，这样就引发了一个问题，如何不重建策略而改变布尔变量的默认状态，因为策略一旦写成就成为一个比较固定的实体了，于是引入了一个新的概念“固定值”，SELinux 实用程序使用的标准库提供了一个方法使得对布尔值的改变永久化，这个方法就是专门用一个文件来保存和维护永久不变的布尔值，在系统初始化期间 `init` 进程就使用这个文件覆盖掉策略中定义的默认值，于是我们就可以改变布尔变量的当前值，即使系统重启也不会丢失，就不用再修改静态的 SELinux 策略了。

在 Fedora Core 4 (CF 4) 和 Red Hat Enterprise Linux 4 (RHEL 4) 中，按照惯例，载入的 SELinux 策略是存储在 `/etc/selinux/[pol_name]/` 目录下的，这里的 `pol_name` 是包括一个 SELinux 策略及相关文件的子目录名称，在 RHEL 4 中，我们这里想讨论的一个文件是 `booleans`，它包含了布尔变量的名称和默认值，`init` 进程在载入策略进入内核后就会读这个文件，然后改变列在这个文件中的所有布尔变量的当前值，如果我们仔细研究一下这个文件，就会发现它包含了很多布尔变量，如：

```
# cat booleans      # 在策略子目录中，
如 /etc/selinux/strict/
ftpd_is_daemon=1
ftp_home_dir=1
ssh_sysadm_login=1
staff_read_sysadm_file=1
user_ping=1
```

这里可以看到前面讨论到的布尔变量 `user_ping`，因此，为了保持布尔变量的当前值一致，我们应该同时修改当前值和 `booleans` 文件，这样做了后就能确保系统重启后布尔变量的值不会变化，即固定下来，成为永久性值。

注意：在一个运行中的系统上，当策略被重新载入时，布尔变量的状态是保持当前的活动状态，而不是被重设为默认或固定状态，这样就可以确保在一个运行的系统中非永久性布尔值的改变是可以保留的。

在 FC 4 中引入一个新文件 `booleans.local`，它和 RHEL 4 上的 `booleans` 文件的使用方法是相同的，`booleans` 文件也被保留下来了，但它被用来存储分布式定义的默认布尔值，`booleans.local` 文件包括本地定义的布尔变量和默认值，它的优先级比 `booleans` 要高，这一小小的改进使得更新 `booleans` 文件中的默认状态时，就不会影响到本地定义的布尔变量了。

FC 5 包括了载入式模块结构，它就没有再供用户可编辑的用来存储永久性布尔值的文件了，管理策略的工具（包括 `setsebool` 的永久性模式）直接与载入式模块结构进行交互，实现对固定值的存储，因此，在 FC 5 中，你应该一直使用 `setsebool` 或其它系统命令改变布尔值。

`setsebool` 命令提供了一个便利的选项 `-p`，使用它可以使对布尔值的改变固定下来，这个选项可以在 RHEL 4，FC 4，FC 5 下工作。当 `setsebool` 带上这个选项时，所有改变会立即反应到活动策略中，作为优先于默认值的本地布尔值，否则，对布尔值的改变只反应在当前运行中的策略，当系统重启后就会重设为默认值，例如，在一个 RHEL 4 系统上：

```
# getsebool user_ping      # 显示当前的运行状态
user_ping > active
# cat booleans | grep user_ping  # 和固定状态
user_ping=1
# setsebool user_ping false    # 改变当前状态
# getsebool user_ping          # 当前状态已经被改变
user_ping > inactive
# cat booleans | grep user_ping  # 当固定状态没有被改变
user_ping=1
```

```
# setsebool -P user_ping false      # 使用-P 改变固定状态
# getsebool user_ping               # 当前状态仍然为 false
user_ping > inactive
# cat booleans | grep user_ping     # 现在固定状态也改变了
user_ping=0
```

注意：你没有必要将布尔变量的当前值改为固定状态，这都依赖于你怎么使用布尔变量，在某些情况下，你可能想改变或切换布尔值，系统重启后又重设为默认值，在这种情况下，你就没有必要使用固定值了。

9.3 条件语句

我们在 SELinux 策略中使用布尔变量的原因是为了可以在编写规则时可以使用条件语句 (if)，条件语句有一个条件表达式，条件表达式是由布尔变量和一个真值及假值的列表组成，如果条件表达式的值为真，真值列表规则就生效，假值列表规则就失效；如果条件表达式的值为假，情况就和前面的相反了，在一个处于运行中的系统上，我们可以通过改变表达式使用的布尔变量的当前值来改变条件表达式的值。

9.3.1 条件表达式和规则列表

最简单最常见的条件语句只有一个布尔变量和一个真值规则列表(没有假值规则列表)，我们还是延续前面的 ping 例子进行讨论，当布尔变量 user_ping 为真值时，我们可以编写规则允许用户域使用 ping，下面就是开启 user_ping 的条件语句：

```
# Example: controlling user ping via a Boolean
# Assumptions (defined elsewhere in policy):
#   unpriv_userdomain: attribute for all ordinary user
domains
#   ping_t: domain type for the ping process (which has
necessary
#           network interface access for ping to work)
#   ping_exec_t: entrypoint file type of the ping
executable

if ( user_ping ) {
# domain transition access to allow user access
allow unpriv_userdomain ping_t : process transition;
allow unpriv_userdomain ping_exec_t : file { read getattr
execute };
# entrypoint might be redundant since ping_t should already
have it
# but adding it again is not harmful
```



```
allow ping_t ping_exec_t : file entrypoint;

    # cause the transition to happen by default
type_transition unpriv_userdomain ping_exec_t: process
ping_t;
}
```

在这个例子中，我们看到对 ping 程序域类型（ping_t）的域转换访问权都给所有的普通用户域类型（假设在策略中他们都关联了 unpriv_userdomain 属性）了，无论在策略的什么位置，我们都可以为 ping 命令正常工作对必要的访问编写规则为其提供 ping 域类型，同时，我们还需要确保用户的角色也是经过 ping_t 域类型授权的（参考第 6 章“角色和用户”），在这种情况下，ping_t 域类型通常是无条件批准预期的用户角色，通过前面已经讨论过的类型转换许可来控制这个授权是否可以利用。

条件语句语法（if）

依赖条件表达式的值，条件语句（if）具体指定了策略语句是启用还是禁用（即内核是执行还是不执行），条件语句完整的语法如下：

```
if (条件表达式) {真值列表} [else{假值列表}]
```

条件表达式 一个条件表达式是由一个或多个布尔变量及逻辑运算符组成的，表 9-1 列出了受支持的逻辑运算符，布尔变量必须使用 bool 语句定义。

真/假值列表 一系列依靠条件表达式的值启用或禁用的规则，当条件列表为真时，真值列表规则就启用（假值列表规则就禁用），当假值列表为真时，情况就相反了。假值列表是可选的，内核只会执行启用的条件规则，受支持的规则包含 allow, auditallow, dontaudit, type_transition 和 type_change。

条件语句在单个策略和基础载入模块，非基础载入模块中有效。在编写本书时，条件表达式还不支持嵌套。

让我们再来看一个同时使用真值规则和假值规则的例子，假设我们想控制 ping_t 域以便 docked 布尔变量决定 ping 程序具有什么访问权（即没有接入有线网络时只能访问无线网络），下面的策略语句是实现这个目标的解决方法的一部分代码：

```
# Example: restricting ping's access based on docked state
# Assumptions (defined elsewhere in policy):
#   docked: Boolean indicating docked state
#   ping_t: domain type for the ping process
#   wired_netif: attrib for all wired netif types
#   wireless_netif: attrib for all wireless netif types

# Allowed wired access when docked, wireless otherwise
if ( docked ) {
allow ping_t wired_netif:netif { tcp_send tcp_recv udp_send
```

```

udp_rcv rawip_send rawip_rcv };
} else {
allow ping_t wireless_netif:netif { tcp_send tcp_rcv udp_send
udp_rcv rawip_send rawip_rcv };
}

# Remaining network and other access needed regardless of
interface
allow ping_t self:capability { net_raw setuid };

# etc., remaining rules not listed for simplicity

```

在这个例子中，我们使用条件语句控制了两个网络接口的访问权，包括原始访问（rawip_send 和 rawip_rcv），同时使用非条件规则（即规则不在条件语句中，不管布尔变量的值是真还是假，它们总是被启用的）向 ping 提供了它需要的其它访问权。

警告：在 SELinux 中，类型强制（TE）规则总是累加的，即它们总是向源-目标-类别组合添加许可，目前还无法使用条件语句从策略中移除许可，因为默认情况下是不接受任何许可的，这就意味着你在编写 allow 规则时要格外小心，不要在策略中的某个位置添加了它，又想在另一个地方视图对它加上约束条件，非条件规则总是优先的，因此，如果你视图在一个被非条件语句接受的条件语句中控制许可，条件规则将没有效果，条件规则会按照条件表达式启用/禁用，非条件规则将总是接受许可。

最后，让我们再来看一个更复杂一点的条件表达式，假设我们想扩展前面的 user_ping 的用途，控制 ping 程序是否对所有用户域都可以接受，而不仅仅是普通用户域，因此我们也为布尔变量使用了另一个名字 allow_ping，使用它更能代表我们的意图，而且，我们想只有计算机接入网络后 ping 才可到达，我们创建了下面的语句：

```

# Example: restricting ping based on docked state and allow
Boolean
# Assumptions (defined elsewhere in policy):
#   docked: Boolean indicating docked state
#   allow_ping: Boolean indicating whether ping is
allowed
#   ping_t: domain type for the ping process
#   ping_exec_t: entrypoint file type of the ping
executable
#   wired_netif: attrib for all wired netif types
#   userdomain: attrib for all user domains (priv &
unpriv)

# Allowed wired access when docked if allowed
if ( allow_ping && docked ) {

```

```
# domain transition permission
allow userdomain ping_t : process transition;
allow userdomain ping_exec_t : file { read getattr
execute };
allow ping_t ping_exec_t : file entrypoint;
type_transition userdomain ping_exec_t: process ping_t;

    # wired netif access for ping
allow ping_t wired_netif:netif { tcp_send tcp_recv
udp_send
udp_recv rawip_send rawip_recv };
}
```

这个例子在条件表达式中使用了两个布尔变量和一个逻辑运算符（&&），这两个布尔变量的值控制条件是否为真以及与之关联的 allow 规则是否启用，条件表达式支持类似 C 语言的逻辑运算符和使用括弧表示优先顺序的规则，表 9-1 列出了条件表达式支持的逻辑运算符。

表 9-1. 条件表达式支持的运算符

运算符	语法	语义
&&	bool_1 && bool_2	逻辑 和
	bool_1 bool_2	逻辑 或
^	bool_1 ^ bool_2	逻辑 “异或”
!	!bool_1	逻辑非
==	bool_1 == bool_2	等于
!=	bool_1 != bool_2	不等于

9.3.2 条件语句限制

条件策略语言扩展有多个重要的限制，实际上，这些限制还没有真正地限制到什么，不过你还是应该小心一点，因为这些限制很难会在将来被移除，也有可能被增强。

9.3.2.1 支持的语句

目前，在一个真或假条件列表中可以接受的策略语句如下：

- allow（类型允许规则，而不是角色允许规则）
- auditallow
- dontaudit
- type_transition
- type_change

这些是 TE 策略规则语句，出现这个限制的原因是条件策略实际上是开发用来支持有条件的 TE 策略的，因此，TE 规则也受到支持，这就很有意义了，因为我们正在讨论的启用和禁用允许访问，审核访问和设置默认访问的规则。

需要特别指出的是，不要在条件表达式内定义类型或其它的策略标识符，要想设计成根据运行时的条件定义策略组件非常困难，相反，在一个给定的策略中，策略组件标识符要么是已经定义好的，要么没有定义，它们不是根据条件要求进行定义的，这也说明了为什么在条件语句中，声明用户和角色，以及布尔变量是无效的原因。

在条件语句内某些不受支持的语句可能很有用，例如：我们发现一个联合了一个之前定义的属性和类型的 `typeattribute` 语句如果允许出现在条件真/假值列表中，那将是非常有用的，因为向类型添加属性本质上是向那个类型添加访问规则，于是有人认为在一个有条件的运行环境中包括这样的语句将非常有意义，这也是为什么在初始化条件策略时 `typeattribute` 语句不受支持的原因，简单地说就是 `typeattribute` 语句本身在策略语言中不受支持，我们希望这些语句最终能在策略编译器中得到支持。

警告：不要把策略构建时选项 (build-time options) 和有条件的运行时环境 (runtime conditionals) 搞混淆了，通常使用脚本/宏语言 (如 m4) 提供构建时选项 (参考第 11 章“原始示例策略”和第 12 章“参考策略”)，例如，你可以控制一个给定的策略中是否包括某个确定的域类型及其关联的规则，假设这样，我们应该从编译好的策略中排除所有的规则及其关联的类型声明，这是一个编译时定制特性，它与有条件的运行时环境完全不同，在后面的例子中，我们包括了所有想要的规则和类型，但允许某些规则 (不是类型) 基于由布尔变量控制的条件进行切换。

9.3.2.2 嵌套条件语句

目前，条件语句语法还不支持嵌套，因此，下面的策略语句可能会产生一个编译错误：

```
# 由于不支持嵌套，下面的语句目前将会失败
if (docked) {
# 接入语句
if (allow_ping) {
# 接入和允许语句
}
} else {
# 不接入语句
}
```

相反，我们不得不将其使用几个单独的语句表达，如：

```
if (docked && allow_ping) {
# 接入和允许语句
}
```

```
}  
  
if (docked) {  
#接入语句  
} else {  
# 不接入语句  
}
```

我们希望尽快将嵌套条件语句条件到策略语言中。

9.4 使用 Apol 检查布尔和条件策略

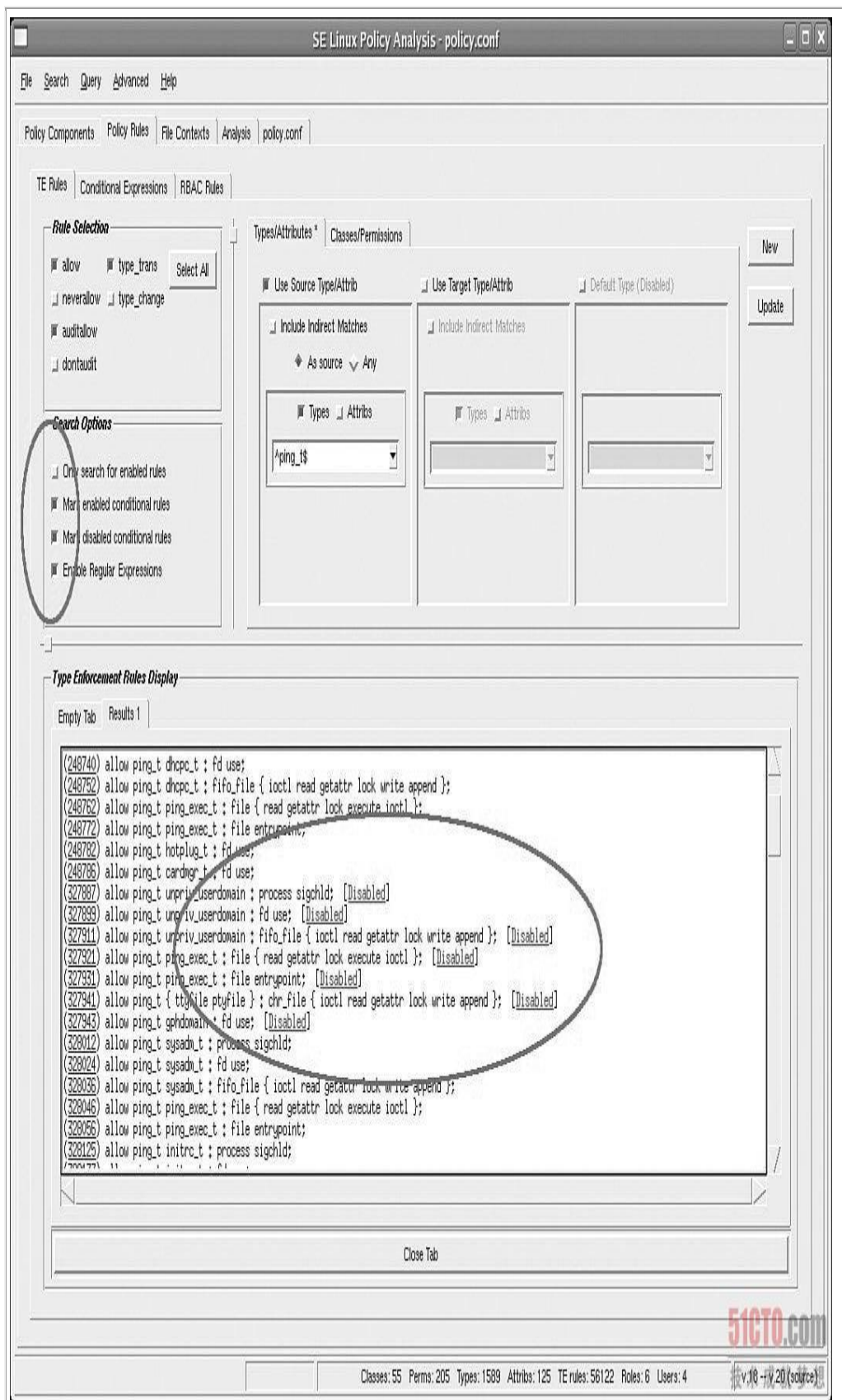
使用 apol 工具我们可以更轻松地检查条件策略语句和它们关联的布尔变量，在理解条件策略语句的作用时，apol 特别有用，特别是相同的条件在策略中出现多次时。

图 9-1 显示了如何使用 apol 在策略中检查定义好的布尔变量，在策略组件（Policy Components）标签下的布尔变量（Booleans）标签中，显示了所有的布尔变量和它们的默认值，apol 也允许你修改布尔变量的默认值，当你研究条件策略规则时它非常有用。



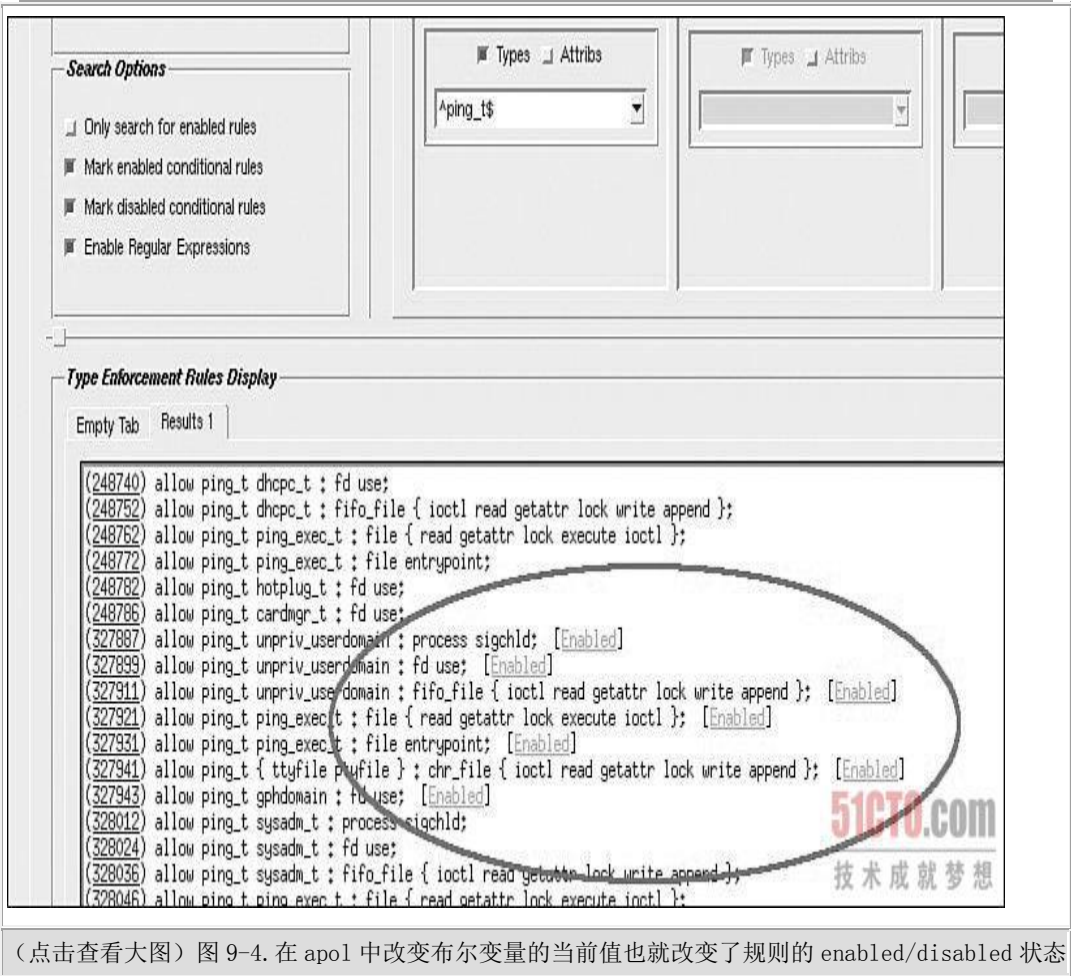
(点击查看大图) 图 9-1. 使用 ap01 检查布尔变量

更有趣的是,当你搜索策略规则时,在 TE 规则(TE Rules)标签下的策略规则(Policy Rules)标签中,你可以配置 apol 是否显示所有的规则,以及显示它们当前的状态,如图 9-2 所示。大部分不在条件语句中的规则不会显示当前的状态,但在条件语句中的规则都是显示它们的状态(enabled 或 disabled)。



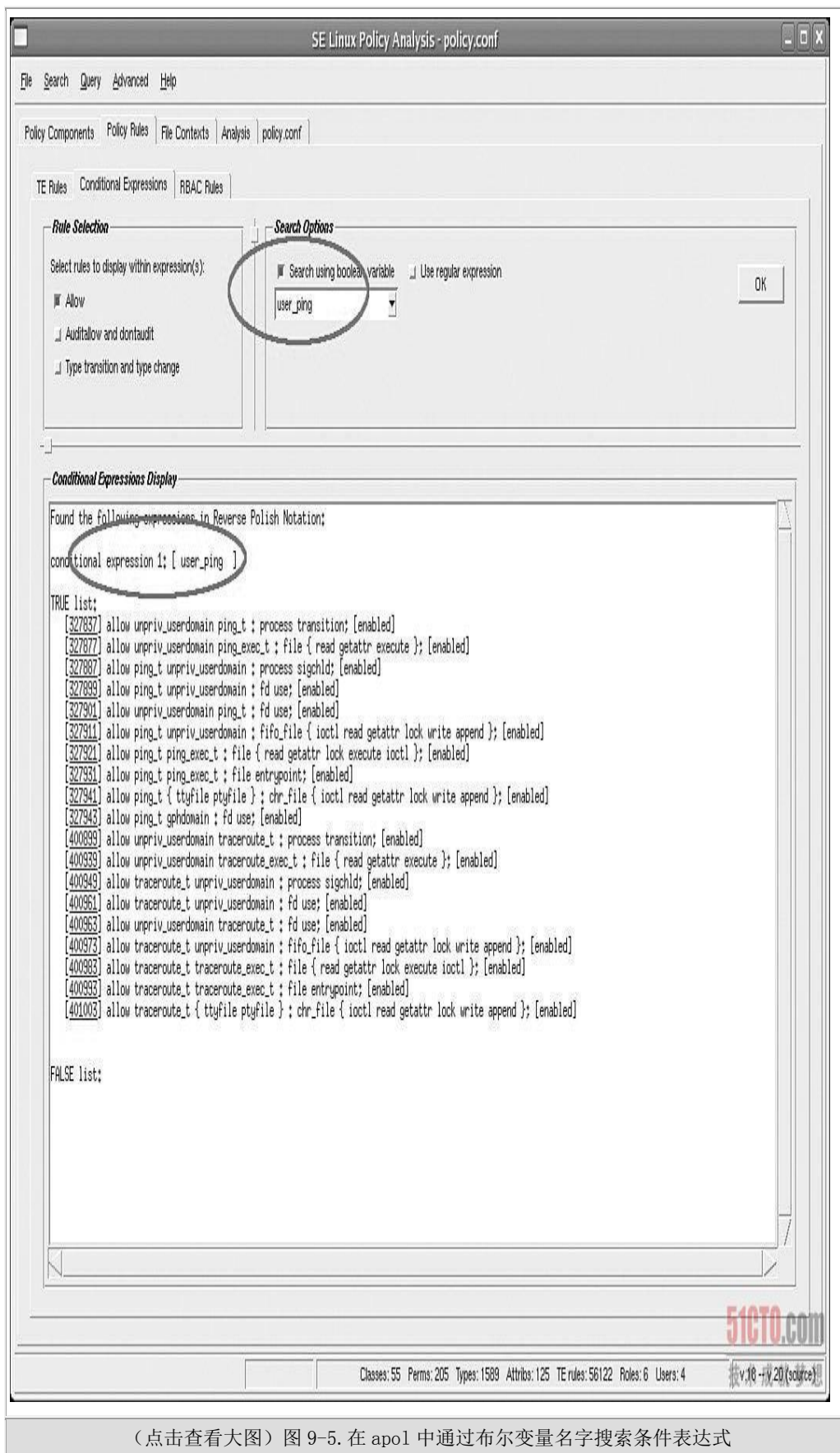
(点击查看大图) 图 9-2. 在 apo1 中查看被禁用的条件规则

你可以在布尔变量（Booleans）标签中使用 apol 修改布尔变量的默认值，查看其影响，例如，在图 9-3 中，我们将布尔变量的默认值从 false 该为 true，这将会影响到哪些规则呢，如图 9-4 所示，先前禁用的规则现在都变为启用了。



最后，通过策略规则（Policy Rules）标签下的条件表达式（Conditional Expression）标签，你可以通过布尔变量搜索所有的条件语句，如图 9-5 所示，apol 将会显示所有使用了

该布尔变量的条件表达式，以及它们的真值/假值规则列表。另外，如果使用 like 条件，apol 将会崩溃（例如，如果有五个条件，它们的条件表达式都相同，apol 将会显示一条合并的条件）。理解完整的相关的条件规则变得更为容易了，如图 9-4 中的规则搜索，布尔变量的当前状态将会影响到这个搜索的结果。



(点击查看大图) 图 9-5. 在 apo1 中通过布尔变量名字搜索条件表达式

9.5 小结

条件语句允许你创建策略规则，通过改变布尔变量的值使其启用或禁用，不在条件语句中的规则总是处于启用状态。

在策略中使用 `bool` 语句定义布尔变量，以及每个布尔变量的默认值。

在一个处于运行的系统上定义的所有布尔变量在 `selinux` 文件系统都有一个与之对应的文件名，通常挂载在 `/selinux/booleans/`，这些文件标识出了每个布尔变量的当前值和待定值，为了改变一个布尔变量的当前值，你可以在这个文件中写入新的值（1 或 0），然后将 1 写入 `/selinux/commit_pending_bools` 文件中，使修改生效，命令 `getsebool` 和 `setsebool` 为改变这些值提供了一个更为方便的方法，不用再去记那些变量文件名了。

布尔变量支持固定值，在系统重启时它会优先于默认值，使用固定值，你可以不用修改策略本身而实现对默认值的修改，最简单的方法就是使用 `setsebool -p` 命令来改变布尔变量的固定值。

条件语句 (`if`) 允许你使用一个定义好的布尔变量和一个真值规则列表或一个可选的假值规则列表来表示一个逻辑条件表达式，这些规则将由内核通过读取条件表达式的值进行启用或禁用。

目前，在条件语句真值/假值规则列表中受支持的规则包括 `allow`，`auditallow`，`dontaudit`，`type_transition` 和 `type_change`。

目前，还不能使用嵌套的条件语句，这个限制在不久的将来会得到解决。

9.6 练习

1、解释布尔变量的默认值，当前值，待定值和固定值之间的区别。

2、假设在我们的策略中定义了三个布尔变量：`bool1`，`bool2` 和 `bool3`，看一下下面的命令：

```
# cd /selinux/booleans
# cat bool1
0 1
# cat bool2
1 1
# cat bool3
1 0
```

这三个布尔变量的当前值是什么？

3、在前面的问题的注释位置，添加下面的命令：

```
# echo 1 > /selinux/commit_pending_bools
```

现在，这三个布尔变量的当前值是什么？

4、条件语句的一个用途是控制 SELinux 执行的审核级别，通过禁用或启用审核规则实现，假设我们想创建一个布尔变量（enhanced_audit）来控制访问尝试的审核（成功和拒绝事件），另外，假设有两类事件我们想进行捕获，以增强审核，这两类事件是：转换到任何域类型和使用 ping 命令访问访问网络。写出实现这个目标的策略关键部分，假设在你的策略中有两个属性：domain，它关联了所有的域类型；netif_type，它关联了网络接口客体使用的所有类型。

第 10 章 客体标记

为了让 SELinux 策略工作，所有客体实例必须用安全上下文进行标记，在这一章中，我们讨论将安全上下文应用到客体实例的不同方法，包括创建客体时如何指派安全上下文，以及之后对这些标记的修改（称为重新标记）。

10.1. 客体标记简介

SELinux 中的所有客体自它们创建时就具有一个安全上下文，直到它们被毁坏，这是 SELinux 执行访问控制的核心能力的属性，例如，我们来看一个在第 2 章“概念”中就讨论过的一个文件的安全上下文：

```
# ls -Z /etc/shadow
-r----- root root system_u:object_r:shadow_t shadow
```

在这个例子中，程序 ls 显示了/etc/shadow 文件的安全上下文。与一个客体关联的安全上下文（本例中是 system_u:object_r:shadow_t）是 SELinux 用来做访问控制决定的唯一属性，因此，为客体指派正确的安全上下文就显得非常之重要了。

迄今为止，我们讨论的客体假设都有安全上下文，但很少或根本没谈过安全上下文是如何确定以及应用的，这反应出了在使用 SELinux 系统时，不应该小觑标记的作用，普通用户和管理员可以象使用标准 Linux 那样使用 SELinux，不用担心安全上下文，所有程序在 SELinux 系统上运行时几乎不需要做任何改动就可以。

SELinux 策略语言包括的大量特性，如 type_transition 规则和域转换，使得标记决定可以自动透明地进行，然而，在某些时候，如系统维护期间、策略开发期间和系统按照期间，标记就成为一个问题了，作为一个策略编写者，我们必须小心标记策略语句，以减轻运行时标记管理的难度。

在 SELinux 系统上标记一个客体有四种方法：

策略语句：SELinux 策略语言包括大量的特性（如 `type_transition` 规则）可以指定客体标记决定的行为。

硬编码默认值：大部分客体类别都将部分类型的默认标记行为编码进客体管理器中，例如：当某个进程创建一个新的套接字时，这个新的套接字就具有创建它的进程一样的安全上下文。

程序请求标记：对于某些客体类别，SELinux 提供了许多应用程序编程接口（API）允许程序明确地请求标记，这对于新创建的和已经存在的客体实例都一样。对于那些存储在支持标记的文件系统上的与文件有关的客体，SELinux 实用程序就会使用这些 API 初始化和修复标记（如安装系统时的 rpm 包管理器）。

初始 SID：SELinux 有一套初始的安全标识符（初始 SID），当客体丢失标记或标记无效时用它作为 failsafe 标记。

对于许多客体，一组这些行为类型可能用于决定一个新客体的标记，标记决定也使用来自执行环境的信息（如进程的安全上下文和有关的客体实例）来确定新客体的安全上下文，在任何情况下，策略必须为标记行为授予允许适当的访问权，客体标记行为通常只控制客体如何标记，不管它是否允许标记。

我们在第 2 章，第 5 章“类型强制”和第 6 章“角色和用户”中已经讨论过一些支持标记决定的策略规则（即类型和角色转换规则），在这一章中，我们讨论另外的与标记网络有关的客体类别的策略语句，你将会看到，这些语句只能应用到特定类别的客体，不是所有客体都可以。

每个客体类别的默认标记行为都是硬编码进客体管理器中的，这些默认的标记行为在有关的策略标记规则不存在时使用，以及那些根本就没有关联策略标记规则的客体类别使用，大部分客体类别的默认标记都继承了创建它的进程/或包括客体的容器的安全上下文，如：与文件有关的客体就继承了包含它们的目录的类型，一个硬编码的角色（`object_r`）和创建进程的 SELinux 用户。

在本章的剩下部分，我们讨论策略编写者必须掌握的标记客体的方法。

10.2 与文件有关的客体标记

在 SELinux 中，标记通常指的是与文件有关的标记，因为这是一个普通用户或管理员可能遇到的唯一一种标记管理形式，文件标记的更多挑战来自如何将文件存储在目录结构中，Linux 可用的文件系统也有很多，这就更增加了复杂性，常见的文件系统包括传统的、本地文件系统都是用来在磁盘上或可移动介质上存储数据（如 `ext3` 和 `XFS`），非本地文件系统的存在都是为了兼容其它操作系统（如 `iso9660` 和 `vfat`），在内存中，伪文件系统用于内核和用户空间之间的通讯（如 `proc` 和 `sysfs`）。

与文件有关的客体如何标记根据文件系统的用途不同而有所差异，客体如何创建，存储和使用的特定语义也不一样，例如，长期存储在 ext3 文件系统磁盘上的文件在创建时就被标记了，安全上下文与文件一起存储，相反，proc 文件系统中的文件仅在运行时才存在，在运行时才会产生标记，这时产生的标记会覆盖在长期存储设备上的安全上下文。

SELinux 挂载选项

context 挂载选项(文件系统使用这个选项挂载时通常指的是“挂载点标记”)优先于任何文件系统的标记行为，它只会给所有与文件有关的客体应用一个安全上下文，例如，思考一下以下的 mount 命令：

```
mount -t nfs -o context=user_u:object_r:user_home_t
               gotham:/shares/homes/ /home/
```

在这个例子中，挂载选项 context=user_u:object_r:user_home_t 告诉 SELinux 应用特定安全上下文给所有与文件有关的 nfs 文件系统中挂载在/home/ 目录的客体，就在这个选项中指定了安全上下文，在这个例子中是 user_u:object_r:user_home_t，这个安全上下文会应用到现有的文件，也会应用到新创建的文件。

context 挂载选项关键字对有文件系统都可用，不管它们支持哪种标记行为，或在策略中指定标记行为，例如，文件系统正常情况下能使用扩展属性标记，如 ext3，那就可以使用 context 选项（即使用 context 选项挂载时，磁盘上新的节点（inode）也不会接受任何 SELinux 属性，现有的节点也不会发生改变），这对于从非 SELinux 系统传输到 SELinux 系统费用有用。

与文件系统挂载有关的选项有两个：fscontext 和 defcontext。它们可以一起使用也可以单独使用代替 context 选项，fscontext 选项用于设置或覆盖文件系统客体实例安全上下文（例如给 ext3 文件系统设置一个安全上下文覆盖策略中默认的为 ext3 文件系统设置的安全上下文），defcontext 选项用于覆盖一个给定的文件系统的文件安全上下文（默认情况下，文件安全上下文就是文件初始化 SID，本章后面将会做介绍）。

标准的 nosuid 挂载选项除了否决了标准 Linux 的 setuid/setgid 行为外，也改变了 SELinux 的行为，nosuid 选项禁用了类型为 enTryptoint 的文件 SELinux 安全上下文转换，使用 context 挂载选项可以实现相同的效果（即强制所有文件用一个不受信任的类型标记），但 nosuid 是另一个绝佳的选择。

SELinux 支持四类与文件有关的客体标记，以标识出文件系统的不同：扩展的属性，基于任务的，基于转换的和一般的安全上下文。这些机制之间的主要差异是 SELinux 如何决定文件系统上节点的初始标记，安全扩展的属性标记也将安全上下文长期存储在磁盘上，表 10-1 列出了文件系统标记机制，以及 Fedora Core 4（FC 4）上每个标记机制使用的文件系统。

表 10-1. 不同类型的文件系统标记机制及其关联的文件系统

标记机制	文件系统
扩展的属性	ext2, ext3, xfs, jfs, reiserfs

基于任务的	pipefs, sockfs
基于转换的	devpts, tmpfs, shm, mqueue
一般的	proc, rootfs, sysfs, selinuxfs, autofs, automount, usbdevfs, iso9660, udf, romfs, cramfs, ramfs, vfat, msdos, fat, ntfs, cifs, smbfs, nfs, nfs4, afs, debugfs, inotifyfs, hugetlbfs, capi, eventpollfs, futexfs, bdev, usbfs, nfsd, rpc_pipefs, binfmt_misc

在策略中要么使用文件系统 use 语句要么使用普通文件系统标记支持语句来指定每个文件系统的标记，通常叫做 genfscon 语句，SELinux 策略支持三种文件系统 use 语句：fs_use_xattr, fs_use_task 和 fs_use_trans，它们分别指定扩展属性，基于任务的，基于转换的标记行为。

这三个文件系统 use 语句的语法是一致的，下面是一个 fs_use_xattr 例子：

```
fs_use_xattr ext3 system_u:object_r:fs_t;
```

文件系统 use 语句语法

文件系统 use 语句指出了用于文件系统的标记机制，它以前面介绍的三个关键字任意一个开始，必须使用中括号括起来，每个文件系统只能有一个文件系统 use 语句，它的完整语法如下：

[fs_use_xattr | [fs use task](#) | fs_use_trans] fs_name [fs context](#)
fs_name 文件系统名字（如 ext3），它与内核和 mount (8) 命令理解的名字一致，列在 /proc/filesystems 文件中。

fs_context 与文件系统关联的文件系统客体实例的安全上下文。

[fs use xattr](#) 语句指出文件系统将使用扩展属性提供安全上下文信息（通过它的 getxattr(2) 方法），fs_use_task 语句指出文件系统将使用基于任务的标记行为提供安全上下文信息，使用 [fs use xattr](#) 时文件系统必须要支持标记行为才行（其它情况下，SELinux 处理标记）。

文件系统 use 语句在单个策略和基础载入模块中有效。在条件语句或非基础载入模块中无效。

10.2.1 扩展属性的文件系统（fs_use_xattr）

大多数本地的，基于磁盘的 Linux 文件系统都使用扩展属性标记，这个标记机制扩展了标准的扩展属性机制，支持设置、检索和存储所有与文件有关的客体的安全上下文。使用这种标记机制的文件系统支持程序请求标记，当存储在持久性介质上时，即使系统重启也会保留其安全上下文。

更多关于使用扩展属性的安全上下文的信息

存储在本地 Linux 文件系统上的与文件有关的资源通常都有关于资源的重要信息，如所有权和访问模式，这些信息存储在一个特殊的数据结构中，叫做

inode，在最近发布的 Linux 版本中，与 inode 有关的额外信息都以扩展属性形式存在。扩展属性存储的是附加信息，如名称/值对，用于存储系统信息，如访问扩展列表（ACL），或其它程序或服务需要的其它数据，SELinux 使用扩展属性存储所有与文件有关的客体的安全上下文。

扩展属性的名称部分分成多个命名空间，以允许不同类型的数据共存，SELinux 使用前缀为 security. 安全命名空间存储安全上下文，这个命名空间由所有的 Linux 安全模块(LSM)共享，因此 SELinux 在安全空间中使用名称 selinux 存储安全上下文，为了说明这一点，我们直接检查一下 SELinux 中文件的扩展属性：

```
# getfattr -n security.selinux /etc/shadow
# file: etc/shadow
security.selinux="system_u:object_r:shadow_t\000"
```

正如你看到的，安全上下文直接存储为一个字符串，可以使用 `ls -Z` 命令显示与文件有关的客体的安全上下文，这样也可以检查其扩展属性，我们强烈建议使用 libselinux API（如 `getfilecon(3)`）代替直接使用扩展属性 API，因为安全上下文的存储久而久之可能会发生变化。

10.2.1.1 扩展属性文件系统的标记行为

对于扩展属性文件系统的标记是由一组策略规则和安全上下文继承决定的，默认情况下，所有新的与文件有关的客体继承了包含它的目录的类型和创建进程的用户，角色总是被设为特殊客体角色 `object_r`。如果 `type_transition` 规则匹配了创建进程的类型和容纳新客体的目录的类型，在这个规则中指定的默认类型将被用于新的与文件有关的客体的类型，安全上下文的剩余部分的设置就和没有 `type_transition` 规则时一样了。

程序请求的标记允许进程使用 `setfscreatecon(3)` 库程序调用为新的文件请求一个特殊的安全上下文，在这种情况下，客体将使用请求到的安全上下文进行创建，除非进程缺少请求的访问权，正常情况下，只有那些扩展了 SELinux 的应用程序或 SELinux 实用程序使用这个特性，标准应用程序创建的文件通过自动标记决定接受正确的安全上下文。

除了在创建文件时设置安全上下文外，与文件有关的客体还可以重新标记，需要用到三个库程序调用：`setfilecon(3)`，`lsetfilecon(3)`和 `fsetfilecon(3)`。只需要适当的 `relabelfrom` 和 `relabelto` 许可就可以明确地改变一个客体的标记，这些许可应该由策略进行严密控制。

客体标记的策略控制

能够改变一个客体的安全上下文需要的权限很大，回顾一下第 4 章“客体类别和许可”中的内容，其中描述到大多数客体类别在策略中都使用 `relabelfrom` 和 `relabelto` 改变与文件有关的客体的类型，`relabelfrom` 许可控制客体的开始类型，`relabelto` 许可控制结果类型，一个域必须要同时具有这两个许可才能成功地重新标记一个客体，例如，看一下下面的 allow 规则：

```
allow user_t user\_home\_t : file { relabelfrom };
allow user_t httpd\_user\_content\_t : file { relabelto };
```

这些 allow 规则说明了一个类型为 user_t 的进程被允许重新标记一个文件，从 user_home_t 重新标记为 httpd_user_content_t。

relabelto 和 relabelfrom 许可只控制客体类型的改变，在第 7 章“约束”中谈到，改变安全上下文中的用户和角色部分可以由约束控制，如下面的语句：

```
constrain file { create relabelto relabelfrom }  
( u1 == u2 or t1 == privowner );
```

这个约束语句说明当一个进程在某个文件上请求 create、relabelto 或 relabelfrom 许可时，安全上下文的用户部分必须与进程的用户匹配，或进程类型具有 privowner 属性。

10.2.1.2 在扩展属性文件系统中管理安全上下文（文件上下文）

与文件有关的客体标记使用的扩展属性与其它文件系统和其它非文件客体类别的扩展属性不同，安全上下文使用扩展属性正常情况下在安装期间被初始化，如 rpm 包管理器安装 rpm 包时，使用运行时标记请求，运行时标记请求由一个或多个配置文件扩展，叫做文件上下文文件，它列出了路径或部分路径和安全上下文，策略中没有直接包括文件上下文文件，而是存储在文件系统上的一个标准位置（参考第 13 章“管理 SELinux 系统”），通过使用合适的文件上下文文件，与文件有关的客体安全上下文可以基于路径被正确初始化，初始化将系统推入一个已知的安全状态，初始化后，自动标记决定接管并确保后面创建的保任何文件能够被正确标记，安全状态总是可维护的。

这个标记管理策略用于隔离策略，它主要从路径名和文件名处理类型和安全上下文，它有几个优点。首先，文件系统的布局可以非常大，通过在策略中消除这方面的变化，单个策略就可以适应更多的系统。

更重要的是，在本地 Linux 文件系统中，与文件有关的客体不能简单地通过单个路径、硬链接、chroot 环境来标识，每个进程文件系统命名空间意味着单个与文件有关的客体可以由多个路径名标识，如果在内核中直接使用路径名实施策略，将无法决定这些路径名哪个是正确的，可能导致进程访问同一个客体时有所不同，依赖于访问的尝试方式不同而不同。由于这个原因，SELinux 直接关联客体的安全上下文，只使用一条路径初始化安全上下文。

文件上下文文件的每一行的格式是有规则的，包括一个或多个与文件有关的客体的路径，一个可选的客体列表规范和一个安全上下文，例如，下面是一段文件上下文文件的内容：

1	/bin(/.*)?		system_u:object_r:bin_t
2	/bin/tcsh	--	system_u:object_r:shell_exec_t
3	/bin/bash	--	system_u:object_r:shell_exec_t
4	/bin/bash2	--	system_u:object_r:shell_exec_t
5	/bin/sash	--	system_u:object_r:shell_exec_t
6	/bin/d?ash	--	system_u:object_r:shell_exec_t
7	/bin/zsh.*	--	system_u:object_r:shell_exec_t
8	/usr/sbin/sesh	--	system_u:object_r:shell_exec_t

```

9  /bin/ls          --      system_u:object_r:ls_exec_t
10 /boot(/.*)?      system_u:object_r:boot_t
11 /boot/System\map(-.*)? system_u:object_r:system_map_t
12 /dev(/.*)?       system_u:object_r:device_t
13 /dev/pts(/.*)?   <<none>>
14 /dev/cpu/. *     -c      system_u:object_r:cpu_device_t
15 /dev/microcode   -c      system_u:object_r:cpu_device_t
16 /dev/MAKEDEV     --      system_u:object_r:sbin_t
17
/dev/null          -c      system_u:object_r:null_device_t
18
/dev/full          -c      system_u:object_r:null_device_t
19
/dev/zero          -c      system_u:object_r:zero_device_t

```

这个例子指出了在/bin/，/boot/和/dev/目录下的文件应该如何标记，例如，第三行指出文件名为/bin/bash的文件应该使用安全上下文 system_u:object_r:shell_exec_t 进行标记，在这里例子中的客体类别规范是一个常规文件，客体类别规范与 find (1) 理解的一致。

当进程查询文件上下文时，文件使用 matchpathcon(3) 库程序调用匹配与文件有关的客体名字，总会使用到大多数具体项目。例如，第一行的正则表达式匹配/bin/目录下的所有文件，如果没有客体类别说明符，就拿第一行来说，它将会匹配所有与文件有关的客体类别。然而，从第二行到第九行，正则表达式就更具体了，它匹配/bin/目录下的特定文件，当请求匹配/bin/bash/时，将会使用第三行，因为它使用的是精确匹配。当请求匹配/bin/dd 时会使用第一行，因为没有适合它的精确匹配。

第 13 行使用了特殊的<<none>>语法，它指出凡是与该正则表达式匹配的与文件有关的客体不进行标记。

许多实用程序和应用程序使用了文件上下文文件，通常是在策略开发和系统管理期间使用。第 13 章描述了这些工具和相关的用法。

10.2.2 基于任务的文件系统 (fs_use_task)

使用基于任务的标记时，新的与文件有关的客体继承创建它们的进程的安全上下文。使用基于任务的标记的文件系统不支持程序请求的标记，这种类型的标记行为对于不真实存储用户数据但支持某种类型的内核资源（如未命名的管道）的伪文件系统有用。如下面的 fs_use_task 语句：

```
fs_use_task pipefs system_u:object_r:fs_t;
```

这条语句指出 pipefs 文件系统使用基于任务的标记，这个文件系统是一个伪文件系统，它用于实现未命名的管道。使用 pipe(2) 系统调用创建未命名的管道，它们与用户空间可见文件系统中的文件没有任何关联，尽管这样，管道通信还是在文件描述符上使用标准的读和写系统调用，因此，Linux 实现了一个用户空间不可见文件系统 pipefs，它只在内核中挂载和使用，使用任务标记进行标记。

10.2.3. 基于转换的文件系统 (fs_use_trans)

基于转换的文件系统标记与基于任务的文件系统标记非常类似，都使用的是伪文件系统，然而，这里使用的安全上下文不是来自创建进程，基于转换的标记设置与文件有关的客体的安全上下文是基于类型转换规则 (type_transition) 的。

用于基于转换的标记的类型转换规则和那些常见的扩展属性机制稍微有点不同，在扩展属性标记文件系统上，标记决定使用创建进程和容纳目录的安全上下文，对于基于转换的标记文件系统，type_transition 规则使用创建进程的安全上下文和关联的 filesystem 客体实例的安全上下文，不提供基于容纳目录的安全上下文，安全上下文总是基于关联的 filesystem 客体的类型，如果没有有关的 type_transition 规则，安全上下文默认就是 filesystem 客体的安全上下文。

研究一下下面的文件系统 use 语句：

```
fs_use_trans devpts system_u:object_r:devpts_t;
```

这个语句指出 devpts 文件系统使用基于转换的标记，devpts 文件系统客体的安全上下文使用 fs_use_trans devpts system_u:object_r:devpts_t;。

正如前面谈到的，基于转换标记的文件系统使用 type_transition 规则取得与文件有关的客体的类型，如下面的类型转换规则：

```
type_transition sysadm_t devpts_t : chr_file  
sysadm_devpts_t;
```

这个规则说明当类型为 sysadm_t 的进程在标记为 devpts_t 的文件系统上创建 chr_file 类别的客体时，最终创建的客体的标记为 sysadm_devpts_t，这里面暗含着这种类型转换的目标是 filesystem 而不是 dir，因为这种类型的转换是应用到基于转换的文件系统上创建的客体，而不是目录类型，如果没有合适的 type_transition 规则，在这个文件系统上创建的任何客体将只有文件系统安全上下文。

10.2.4 普通安全上下文标记 (genfscon)

普通安全上下文标记语句 (genfscon) 用于运行时标记伪文件系统，如 proc 和 sysfs，和不支持扩展属性的传统文件系统，与前面讨论过的其它文件系统标记机制不同，它需要修改内核文件系统代码，genfscon 标记至少在受限的条件下可用于未修改的文件系统。

genfscon 语句同时指出了文件系统标记机制和存储在文件系统中与文件有关的客体的标记机制，有两种类型的 genfscon 语句：适合于与文件有关的客体的细粒度标记的完整形式，适合于传统文件系统的受限形式。

10.2.4.1 genfscon 语句细粒度标记

思考一下下面的适合于 proc 文件系统的 genfscon 语句完整形式：

```
1 genfscon proc
/                               system_u:object_r:proc_t
2 genfscon proc
/kmsg                           system_u:object_r:proc_kmsg_t
3 genfscon proc
/kcore                          system_u:object_r:proc_kcore_t
4 genfscon proc
/mdstat                         system_u:object_r:proc_mdstat_t
5 genfscon proc
/mtrr                           system_u:object_r:mtrr_device_t
6 genfscon proc
/net                            system_u:object_r:proc_net_t
7 genfscon proc
/sysvipc                        system_u:object_r:proc_t
8 genfscon proc
/sys                            system_u:object_r:sysctl_t
9 genfscon proc
/sys/kernel                     system_u:object_r:sysctl_kernel_t
10 genfscon proc
/sys/net                        system_u:object_r:sysctl_net_t
11 genfscon proc
/sys/vm                         system_u:object_r:sysctl_vm_t
12 genfscon proc
/sys/dev                        system_u:object_r:sysctl_dev_t
13 genfscon proc
/net/rpc                        system_u:object_r:sysctl_rpc_t
14 genfscon proc
/irq                            system_u:object_r:sysctl_irq_t
```

正如这些示例语句显示的，genfscon 语句语法需要文件系统名字，一个完整的或部分路径名（相对于根文件系统）和一个安全上下文，完整的 genfscon 语句语法如下：

普通安全上下文语句 (genfscon)

普通安全上下文语句 (genfscon) 指定用于文件系统的标记机制和存储在该文件系统中与文件有关的客体的标记机制，每个文件系统可以有多个 genfscon 语句，完整的语法如下：

```
genfscon fs_name partial_path context;
```

fs_name 使用 genfscon 标记的文件系统名字（如 proc），文件系统名字与内核理解的和 mount(8) 命令理解的一致，列在 /proc/filesystems 文件中。

partial_path 相对于文件系统挂载点的部分路径（如文件系统挂载在 /proc，部分路径/在运行时翻译成/proc），如果一个文件系统指定了多个 genfscon 语句，与文件有关的客体的安全上下文采取就近原则获取安全上下文。

context 用于标记与文件有关的客体的安全上下文，往往是最匹配 genfscon 语句的安全上下文，这个安全上下文也用于标记与文件系统关联的文件系统客体。

genfscon 语句在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。

这些示例 genfscon 语句显示对于同一个文件系统可以有多个 genfscon 语句，对于那些支持完整格式 genfscon 语句的文件系统，使用多个 genfscon 语句细粒度指定与文件有关的客体的标记，出现多个 genfscon 语句时，与文件有关的客体的安全上下文是由最匹配 genfscon 语句决定的。

例如，假设 proc 文件系统挂载在 /proc（标准位置），使用这些示例语句，/proc/filesystems 将匹配第 1 行的语句，接受安全上下文为 system_u:object_r:proc_t，类似的，目录 /proc/sys/kernel/ 将匹配第 9 行的 genfscon 语句（使用部分路径 /sys/kernel/ 匹配），它就被标记为 system_u:object_r:sysctl_kernel_t。

所有使用 genfscon 标记的文件系统至少包括一个 genfscon 语句，使用 / 作为部分路径，在这个 genfscon 句中的安全上下文用于标记文件系统，此外它还作为存储在该文件系统下的与文件有关的客体的默认安全上下文。在前面的例子中，文件系统客体 proc 将接受安全上下文 system_u:object_r:proc_t。

在 proc 中标记 PID 文件

proc 文件系统下的文件和目录代表系统上的活动进程，这些包括在以进程 ID 命名的目录下的文件和目录，可以通过 libselinux 调用（如 getcon(3)，setcon(3)）获取或设置进程的属性，PID 目录和它包括的所有文件和子目录的安全上下文与它们代表的进程的安全上下文是一样的。

由于这个原因，下面这样的规则就很常见了：

```
allow apache_t self : dir { read getattr search };
allow apache_t self : file { read getattr write };
```

这种规则允许某个域（本例中是 apache_t）访问代表它本身的文件和目录。

10.2.4.2 使用 genfscon 语句标记传统文件系统

正如前面谈到的，genfscon 有两种使用方式。在分析适合于传统文件系统的受限的 genfscon 语句之前，我们先分析一下 proc 文件系统的某些属性，使其可以使用完整的 genfscon 语句。这将帮助你理解为什么其它文件系统不能使用这些特性。

首先，除了代表活动进程的文件和目录外，所有能够显示在 proc 中的文件和目录的名字在所有系统上都是一致的。如/proc/sys/kernel/hostname 总是用于获取或设置主机名的文件，虽然重要文件的位置在各个系统上的位置都想相对固定的（如/etc/shadow），但相对于文件系统挂载点的位置却很少有人知道，更重要的是，很少有与文件有关的客体的安全属性是通过路径来决定的。

其次，在 proc 文件系统中的与文件有关的客体是通过路径名进行唯一性标识的，内核在所有情况下都能够很容易地判断出它的绝对路径，proc 文件系统不支持硬链接，如由 /proc/sys/kernel/hostname 标识的客体永远都不可能由其它路径标识。

这些属性加在一起使得 proc 文件系统特别适合使用基于路径名的标记，只有少部分文件系统具有这些属性，这样使得通过路径名进行标记不仅很困难而且风险也很大。

对于大多数传统文件系统，我们使用受限形式的 genfscon 语句进行标记，包括那些不具有这些属性的文件系统，如 proc，为了处理那些不能使用路径名进行标记的文件系统，我们使用 genfscon 语句为这些文件系统以及存储它上面的与文件有关的客体设置默认的标记，使用一条 genfscon 语句就可以对整个文件系统进行设置，如：

```
genfscon vfat /                system_u:object_r:dosfs_t
genfscon msdos /              system_u:object_r:dosfs_t
genfscon fat /                system_u:object_r:dosfs_t
genfscon ntfs /              system_u:object_r:dosfs_t
```

这些 genfscon 语句分别设置了 vfat，msdos，fat 和 ntfs 文件系统关联的 filesystem 客体的安全上下文，以及存储在这些文件系统上的与文件有关的客体的安全上下文，都设置为 system_u:object_r:dosfs_t。

10.3 网络和套接字客体标记

网络和套接字客体使用策略语句和初始 SID 标记，没有适合于程序请求标记的机制，我们使用多条策略标记语句标记网络和套接字客体。表 10-2 列出了所有与网络和套接字有关的标记语句和相关的客体类别。

表 10-2. 网络和套接字相关的客体标记机制

SELinux 策略语	Linux 资源和 SELinux 客体类别
-------------	------------------------

句	
netifcon	网络接口: netif
nodecon	代表网络主机的 ip 地址: node
portcon	网络套接字: tcp_socket (name_bind, recv_msg 和 send_msg) , udp_socket (name_bind, recv_msg 和 send_msg) , rawip_socket (recv_msg 和 send_msg)

10.3.1 网络接口标记 (netifcon)

网络接口使用网络接口安全上下文语句(netifcon)或使用netif初始化SID进行标记, 如:

```
netifcon eth0 system_u:object_r:netif_t
system_u:object_r:packet_t
```

这个语句为网络设备 eth0 提供了安全上下文 system_u:object_r:netif_t (即第一个安全上下文), 并给这个接口接收到的数据包指定了默认安全上下文 system_u:object_r:packet_t(即第二个安全上下文),默认数据包标记不是当前所使用的, 期待每个数据包都可以单独进行标记, 完整的 netifcon 语句语法如下, 网络接口名 (本例中是 eth0) 和 ifconfig(8)理解的含义是一致的。

注意: 对每个数据包进行标记可以细粒度控制网络通信, 它也是 SELinux 的实现初衷, 但当 SELinux 以 LSM 模块形式合并到内核中时, 并没有包括这个特性, 主要担心入侵和性能影响, 特别是那些额外的对每个数据包的访问检查, 结果就有部分 SELinux 网络控制功能没有包括进来, 如每数据标记。

任何一个没有使用 netifcon 语句标记的网络接口就使用 netif 初始 SID 作为安全上下文。

网络接口安全上下文语句 (netifcon)

网络接口安全上下文语句用于标记 netif 客体实例, netifcon 语句完整的语法如下:

netifcon interface if_context packet_context

interface

if_context

packet_context

用于标记的网络接口名字 (如 eth1), 接口名字和 ifconfig(8)命令理解的名字是一个含义。

与特定网络接口关联的 netif 客体实例的安全上下文。

网络接口接收到的数据包的默认安全上下文, 目前未使用。

netifcon 语句在单个策略和基础载入模块中有效, 在条件语句和非基础载入模块中无效。

10.3.2 网络节点标记 (nodecon)

节点对象使用节点安全上下文语句 (nodecon) 或 node 初始 SID 进行标记, nodecon 语句通过子网和掩码对节点客体进行标记, 回顾第 4 章中的内容, node 客体类别通过 IP 地址代表网络节点, 如:

```
nodecon 127.0.0.1 255.255.255.255
system_u:object_r:node_lo_t
```

这个语句指出所有 IP 地址为 127.0.0.1 子网掩码为 255.255.255.0 的节点 (即一台主机 127.0.0.1 或 localhost) 使用上下文 system_u:object_r:node_lo_t 进行标记。nodecon 语句完整的语法如下:

节点安全上下文语句 (nodecon)

节点安全上下文语句 (nodecon) 标记 node 客体实例, 完整语法如下:

nodecon subnet netmask context

subnet 一个 IP 地址或子网 (如 127.0.0.1 或 192.168.0.0), 可以是 IPv4 地址, 也可以是 IPv6 地址。

netmask 子网掩码, 子网掩码必须匹配子网的协议版本。

context 代表特定子网和掩码的节点客体实例的安全上下文。

nodecon 语句在单个策略和基础载入模块中有效, 在条件语句和非基础载入模块中无效。

nodecon 语句支持 IPv4 地址和 IPv6 地址, 如:

```
nodecon ::1 ffff:ffff:ffff:ffff:ffff:ffff:
ffff:ffff system_u:object_r:node_lo_t
```

对于 localhost, 使用 IPv6 地址的 nodecon 语句和使用 IPv4 地址的 nodecon 语句是等效的。

除了精确匹配外, nodecon 语句还支持模糊匹配, 如下面的语句匹配整个子网:

```
nodecon 192.168.0.0 255.255.255.0
system_u:object_r:node_intranet_t
```

上面的示例语句将匹配所有子网为 192.168.0.0 的 C 类网络主机。

节点安全上下文语句由策略编译器自动排序, 越精确的语句越先匹配, 它和 genfscon 语句的工作机理类似, 这个约定允许策略包含 IP 地址范围重复的 nodecon 语句, 并自然解决掉冲突, 如:

```
nodecon 192.168.0.0 255.255.0.0
system_u:object_r:node_intranet_t
nodecon 192.168.1.0 255.255.255.0
system_u:object_r:node_webserver_t
```

在这个例子中，子网为 192.168.0.0 的 nodecon 语句比子网为 192.168.1.0 的 nodecon 语句更常见，这样在节点安全上下文自动排序时确保所有在子网 192.168.1.0 内的地址（如 192.168.1.100）匹配第二条语句，接受类型为 webserver_t，而其它在子网 192.168.0.0 内的地址（如 192.168.2.1）匹配第一条语句。

当前有效的策略并没有大量使用节点标记，通常只标记了 localhost 和所有其它节点，如下面这些 IPv4 nodecon 语句就非常常见：

```
nodecon 127.0.0.1      255.255.255.255
system_u:object_r:node_lo_t
nodecon 0.0.0.0 255.255.255.255
system_u:object_r:node_inaddr_any_t
```

这样设计的目的是为了消除基于本地网络设置进行策略自定义的需要，为特定应用程序定制的策略都倾向于重新构造网络策略，以便于更好地控制网络。

所有 nodecon 语句没有匹配到的节点将使用 node 初始 SID 进行标记。

10.3.3 网络端口标记 (portcon)

代表端口的客体使用端口安全上下文语句(portcon)或port初始SID进行标记,portcon语句基于协议和端口或端口范围进行标记，如：

```
portcon tcp 80 system_u:object_r:http_port_t
```

这个语句显示 portcon 语句语法需要协议（tcp 或 udp）、端口号或端口号范围，以及一个安全上下文。注意语句的末尾没有分号，完整的端口安全上下文语句语法如下：

端口安全上下文语句 (portcon) 语法
端口安全上下文语句 (portcon) 基于协议和端口号或端口号范围标记网络端口，完整的语法如下：
portcon protocol port_num context
protocol 网络协议 (tcp/udp)。
port_num 端口号或端口号范围（如 80, 11023），如果多个语句有重叠，用第一个匹配到的语句标记端口。
context 与端口关联的套接字客体实例的安全上下文。
portcon 语句在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。

上面的例子用安全上下文 system_u:object_r:http_port_t 标记了 80 端口。使用端口范围时，常常会出现重叠现象，如下面这两条语句：

```
portcon tcp 80 system_u:object_r:http_port_t
portcon tcp 1-1023 system_u:object_r:reserved_port_t
```

这两条语句都匹配了端口 80，在这种重叠的情况下，使用第一条匹配到的语句，在这个例子中，与端口 80 关联的套接字客体将接受安全上下文 `system_u:object_r:http_port_t`，1-1023 范围内的其它端口将接受安全上下文 `system_u:object_r:reserved_port_t`，这个解决方法使得策略维护变得相对简单，可以在一个大范围的标记语句前插入一个具体的语句。

与 `nodecon` 语句不同，`portcon` 语句按照策略指定的顺序进行匹配，这意味着按某种方式排序 `portcon` 语句时，如果有语句永远都不会匹配，在这种情况下，策略编译器会产生一个警告。

没有匹配到 `portcon` 语句的端口将使用 `port` 初始 SID 进行标记。

你可能还没有注意到从第 4 章开始，到现在都还没有出现过真正的端口客体类别，与端口有关的许可用于检查用端口类型标记的套接字客体，用于检查端口许可的套接字客体实例不同于用于进程通信的套接字客体实例，进程通信的套接字是使用创建进程的类型进行标记的。

例如，假设 `tcp` 端口 80 正常情况下用于 `http` 通信，用 `http_port_t` 进行标记，允许类型为 `httpd_t` 的进程在这个端口上接收 `tcp` 数据，需要进程类型标记的 `tcp` 套接字许可和 `http_port_t` 类型标记的 `tcp` 套接字许可，为了说明，规则只允许接收 `tcp` 数据（查看 `tcp_socket` 客体类别上的 `recv_msg` 许可），如：

```
allow httpd_t self : tcp_socket recv_msg;
allow httpd_t http_port_t : tcp_socket recv_msg;
```

在两个规则清楚地显示了两个 `tcp_socket` 客体实例，表 10-2 显示了套接字客体类别上用端口类型标记的客体上检查的许可。

10.3.4 套接字标记

由进程使用 `socket(2)` 系统调用创建的套接字继承创建进程的安全上下文，检查与端口有关的许可的套接字在前面的 `portcon` 语句中已经讨论了。

例如，一个安全上下文为 `system_u:system_r:httpd_t` 的进程创建的套接字的安全上下文也是一样的，这意味着可以使用一个 `tcp` 套接字，一个类似下面的 `allow` 规则来发送和接收这个域类型：

```
allow httpd_t self : tcp_socket { read write send_msg
recv_msg };
```

这个例子说明 tcp 套接字是如何标记的，真正使用套接字时还需要额外的许可，由用户空间进程创建的所有套接字，包括本地套接字如 Netlink 和 UNIX 域套接字，都使用这个方法进行标记，由内核创建的套接字使用内核初始 SID 的安全上下文进行标记。

10.4 System V IPC

除 msg 客体外，System V 进程间通信 (IPC) 客体使用创建进程的安全上下文进行标记，例如，如果一个安全上下文为 user_u:object_r:user_xserver_t 的进程创建了一个关系内存段，与之关联的 shm 客体就具有相同的安全上下文 user_u:object_r:user_xserver_t，这个标记行为对于 shm，sem 和 msgq 客体类别都一样。

msg 客体使用 type_transition 规则进行标记，这个规则使用消息发送进程的类型和信息队列的类型，例如：

```
type_transition user_t user_xserver_t : msg user_msg_t;
```

type_transition 规则指出当类型为 user_t 的进程在类型为 user_xserver_t 的消息队列上发送消息时，消息类型应该是 user_msg_t，与之前讨论过的 type_transition 规则不同，不为进程提供明确的消息类型的请求，如果 type_transition 规则没有匹配到，消息接受发送进程相同的类型。

无论消息是通过继承接受类型还是通过 type_transition 规则接受类型，进程都必须要有发送该类型消息的许可，例如，下面的 allow 规则可能需要上面的 type_transition 规则：

```
allow user_t user_msg_t : msg send;
```

这条 allow 规则指出类型为 user_t 的进程允许发送类型为 user_msg_t 的消息，注意这里不需要标记消息的访问权，只需要发送消息的权利，也不提供创建消息而不发送，这些条件都是基于 System V 消息和消息队列实现的%A

10.5 其它客体标记

表 10-3 列出了剩下的客体类别 (capability, process, security 和 system) 的标记机制。

表 10-3. 其它客体类别关联的标记机制

客体类别	标记机制
capability	继承来自关联 process 客体的标记
process	继承来自父进程的标记，或由域转换或动态上下

	文转换设置的标记
security	SECURITY 初始 SID
system	SYSTEM 初始 SID

10.5.1 capability 客体标记

capability 客体类别与进程客体类别紧密相关，capability 客体与它们关联的进程具有相同的安全上下文，例如：

```
allow user_t self : capability dac_override;
```

这条 allow 规则指出类型为 user_t 的进程允许保留 dac_override 权利，依靠 self 关键字，capability 客体的类型和进程的类型一样了，没有策略语句或机制设置或改变 capability 客体的安全上下文。

10.5.2 process 客体标记

进程客体的标记是由 SELinux 集中控制的，因为它是与应用程序关联的正确访问权的机制，第 2 章包括了对这个机制和域转换的大幅描述，它是进程标记最重要的方面，这里我们讨论进程标记的其它方面。

在 Linux 中，进程客体不是应用程序执行时（使用 execve(2) 系统调用）创建的，相反，新的进程是使用 fork(2) 或 clone(2) 系统调用复制另一个进程创建的，因此，新的进程客体继承创建进程的安全上下文，反应出它们具有相同的安全属性，不提供否决这种标记决定的方法，域转换和动态安全上下文转换是改变进程安全上下文的唯一方法，它们也只能改变现有进程的安全上下文。

域转换是在 execve(2) 系统调用上改变进程安全上下文的，通过使用 setexeccon(3) 系统调用，安全上下文的改变可以自动触发 type_transition 规则或程序请求。改变进程安全上下文必须在策略中明确地允许。

我们通常在域转换期间通过 execve(2) 系统调用改变进程类型，但也可能改变用户或角色，角色的改变可以通过角色转换语句自动进行，用户和角色的改变可以通过 setexeccon(3) 系统调用明确地由程序请求进行，改变进程的用户或角色是由约束和角色 allow 规则控制的，我们已经在第 6 和 7 章中讨论过了。

动态安全上下文转换是改变现有进程的安全上下文的程序请求标记，使用 sercon(3) 系统调用实现，它必须具有 dyntransition 许可，第 5 章有关于动态安全上下文转换的更多信息，包括它的危险性和我们的建议。

10.5.3 system 和 security 客体标记

system 和 security 客体类别在每个实例中是唯一的，kernel 和 security 初始 SID 分别用于标记 system 和 security 客体实例，没有提供改变这些客体安全上下文的机制。

10.6 初始安全标识符

初始 SID 提供了一种特殊的默认标记行为，初始 SID 适用于两种环境：在系统初始化策略还没有载入前的时间，以及当客体的安全上下文无效或安全上下文丢失时使用。

第 7 章介绍了 SID，相对于安全上下文，它并不透明，因为它通常只在 SELinux 内部使用，初始 SID 是一套保留的 SID，用于系统初始化期间或预定义客体时使用，与大部分 SID 不同的是，它是在运行期间按需创建的，初始 SID 在系统中总是存在的（即它们是硬编码进 SELinux LSM 模块的），表 10-4 列出了 FC 4 系统中使用到的初始 SID。

表 10-4. FC 4 中初始 SID 示例

初始 SID	描述
kernel	适用于所有由内核创建的客体（如内核创建的线程和套接字），system 客体实例，并且作为内核资源的默认标记。
security	适用于 security 客体实例。
unlabeled	适用于所有安全上下文无效的客体。
file	与文件有关的客体的默认安全上下文，如果它们的安全上下文无效，就使用 unlabeled SID。
port	与端口关联的套接字客体默认安全上下文。
netif	与网络接口关联的 netif 客体的默认安全上下文。
node	与节点关联的 node 客体的默认安全上下文。
sysctl	proc 文件系统系统客体的默认安全上下文，这些客体通常通过文件系统安全上下文语句而不是这个初始 SID 进行标记的。

在系统初始化的早期，策略还没有载入时，有一部分客体是通过初始 SID 进行标记的，这个标记行为在如标记内核安全服务器客体和根文件系统时需要用到，在第一条策略载入前，它就存在于系统中了，当策略真实载入后，初始 SID 就关联合适的安全上下文了。

初始 SID 也用于防止客体安全上下文丢失和失效，如果安全上下文丢失或失效将使 SELinux 正确实施访问控制变得不可能，相反，SELinux 就会使用 unlabeled 初始 SID 关联这些客体，unlabeled 初始 SID 具有的安全上下文只允许受限的访问，因此预防了不恰当的访问，直到客体被管理员重新标记或销毁。

无效的安全上下文通常来自新载入的移除了用户，角色或类型或改变了角色或类型授权的策略，在这种情况下，SID 代表使用这些无效名字或关联的安全上下文，在策略载入时，它们就被映射为 unlabeled SID，无效安全上下文在系统间传输客体实例时也可能出现（如使用可移动介质），而且，如果客体是在一个非 SELinux 系统上创建的，它们将没有关联安

全上下文，不管安全上下文是丢失还是无效，SELinux 在首次访问时都将会使用 unlabeled 初始 SID 作为安全上下文。

与客体类别相似，初始 SID 是由内核和其它客体管理器定义的，而且是在策略中声明的，初始 SID 声明语句在策略中声明一个 SID，在编写策略时，我们一般不会改变初始 SID 语句，为了说明其语法，下面举一个例子进行说明：

```
sid kernel
```

这个语句声明了初始 SID kernel，初始 SID 名字在它们自己的命名空间，可以与类型，客体类别或其它策略组件名称重叠，初始 SID 语句完整的语法如下：

初始 SID 声明语句语法 (sid)

初始 SID 语句 (sid) 为初始 SID 取了一个名字，初始 SID 由内核和其它客体管理器定义，这个语句使它们在策略中有效，完整的语法如下：

sid sid_name

sid_name 初始 SID 的名字，可以包括字母或数字。

初始 SID 声明语句仅在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。

初始 SID 安全上下文语句语法 (SID)

这个初始 SDI 安全上下文语句 (sid) 将前面声明的初始 SID 与上下文进行关联，完整的语法如下：

```
sid sid_name context;
```

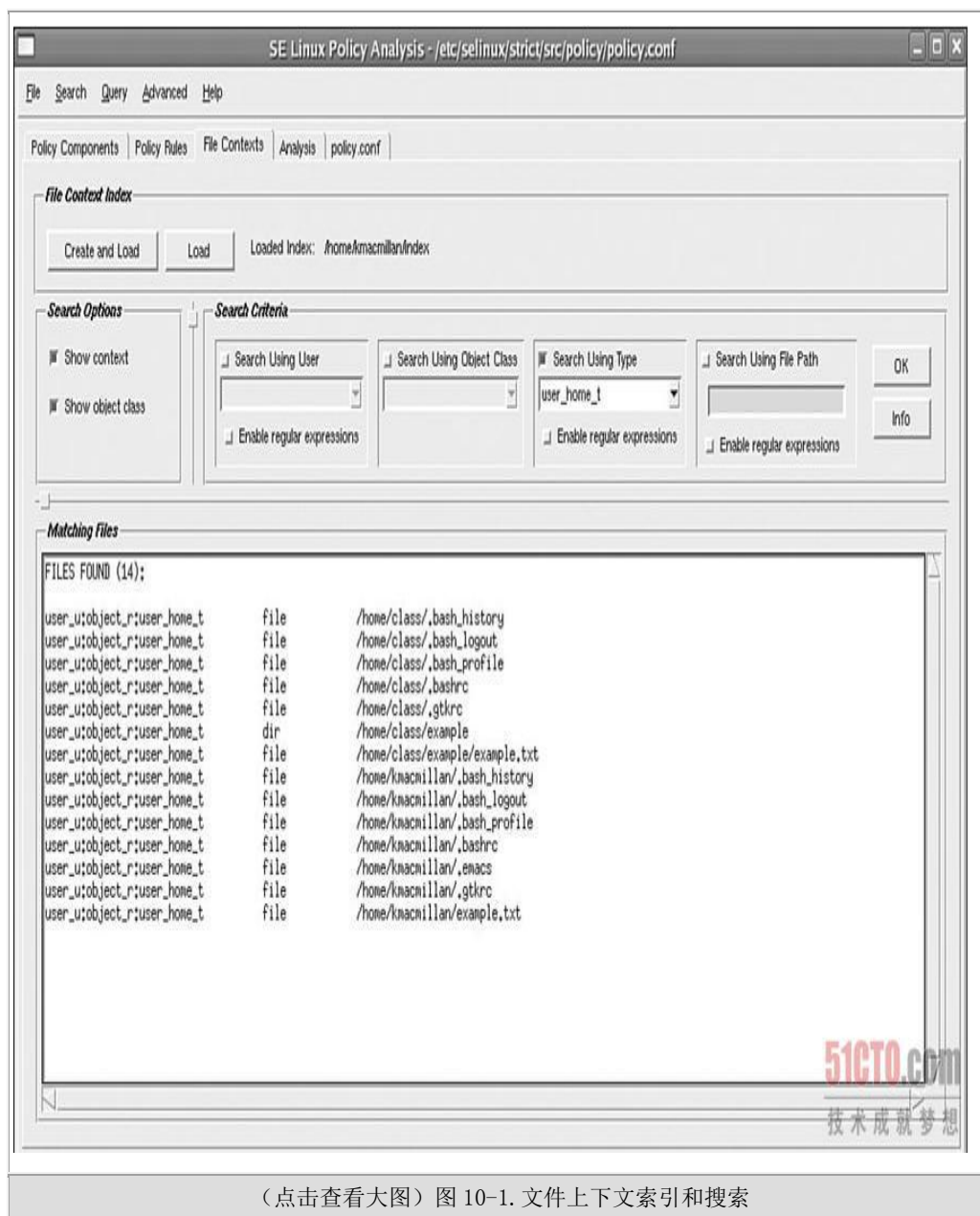
sid_name 前面声明的初始 SID 的名字。

context 与初始 SID 关联的安全上下文。

初始 SID 安全上下文语句在单个策略和基础载入模块中有效，在条件语句和非基础载入模块中无效。

10.7 使用 Apol 研究客体标记

Apol 目前在理解客体标记方面有两个特性：规则搜索和文件安全上下文索引和搜索。我们在第 5 和 6 章中已经研究过规则搜索了，图 10-1 显示了 apol 的文件上下文 (File Context) 标签，使用它创建和搜索与文件有关的客体的安全上下文的索引，允许我们检查与文件有关的客体是如何标记的，它显示了与文件有关的客体应该如何标记，在尝试理解一个策略在系统上是如何实施的时候，关于与文件有关的客体是如何标记的消息是必不可少的。



(点击查看大图) 图 10-1. 文件上下文索引和搜索

文件上下文索引是系统上所有与文件有关的客体的安全上下文的一个快照，这个索引可以由 apol 创建，使用创建（Create）和载入（Load）按钮，或使用 indexcon 命令（包括在 setools 软件包中），这两个工具都可以递归地检索所有挂载的文件系统，记录名字，客体类别和所有与文件有关的客体的安全上下文。在所以创建好后（数据存储在文件中），就可以使用 apol 或 searchcon 命令进行搜索，因此索引是存储了的，所以搜索起来性能非常高，它不搜索真正的文件系统，如图 10-1 显示的搜索类型为 user_home_t 的与文件有关的客体的结果，搜索文件上下文索引找出所有类型为此的文件速度相当快，搜索文件系统时却要花费好几分钟，此外，搜索文件上下文索引可以在非创建它的文件系统上进行。

可以使用任何类型的名字，用户，客体类别或类型的组合进行搜索，基于角色的搜索还不支持，因为所有与文件有关的客体正常情况下都有一个特殊的角色 object_r。

10.8 小结

客体可以使用下列 4 种方法之一进行标记：策略语句（如类型转换规则），硬编码进默
认的客体管理器，程序请求标记和初始 SID。

除了有关标记语句外，策略还必须包括合适的访问权才能使得标记成功。

标记决定通常使用来自执行环境的消息（如进程的安全上下文和有关的客体实例）。

与文件有关的标记行为每个系统使用文件系统 use 语句或濮阳安全上下文语句指定的。

扩展属性标记用于大部分本地 Linux 文件系统，支持程序请求的标记，并支持永久存储
安全上下文。

在扩展属性上标记的文件系统是使用文件上下文文件和读取这些文件的实用程序进行
管理的。

基于任务和基于转换的标记主要用于伪文件系统。

普通安全上下文标记主要用于标记 proc 和传统文件系统。

网络接口是通过接口名（如 eth0）使用 netifcon 语句进行标记的。

网络节点是通过 IP 地址和掩码使用 nodecon 语句进行标记的。

端口是通过端口号使用 portcon 语句进行标记的。

成功发送和接收网络数据除了有关的 node 和 netif 客体许可外，通常需要多个套接字
客体许可。

除 msg 客体外，system V IPC 客体接受创建进程的安全上下文，msg 客体基于
type_transition 规则或创建进程的安全上下文进行标记。

进程接受它们的父进程相同的安全上下文，可以通过域转换或动态上下文转换进行改变。

capability 客体与它们关联的进程具有相同的安全上下文。

security 和 system 客体分别接受 kernel 和 security 初始 SID 的安全上下文。

初始 SID 用于标记那些安全上下文丢失或失效的客体。

10.9 练习

1、假设一个文件上下文文件的内容如下，/etc/passwd，/etc/shadow 和/etc/mtab 接
受的安全上下文是什么？

```
/etc(/.*)?          system_u:object_r:etc_t
/var/db/.*/.*.db    --  system_u:object_r:etc_t
```

```
/etc/.pwd\..lock -- system_u:object_r:shadow_t
/etc/passwd\..lock -- system_u:object_r:shadow_t
/etc/group\..lock -- system_u:object_r:shadow_t
/etc/shadow.* -- system_u:object_r:shadow_t
/etc/gshadow.* -- system_u:object_r:shadow_t
/var/db/shadow.* -- system_u:object_r:shadow_t
/etc/blkid\..tab.* -- system_u:object_r:etc_runtime_t
/etc/fstab\..REVOKE -- system_u:object_r:etc_runtime_t
/etc/.fstab\..hal\..+ -- system_u:object_r:etc_runtime_t
/etc/hostname -- system_u:object_r:etc_runtime_t
/etc/ioctl\..save -- system_u:object_r:etc_runtime_t
/etc/mtab -- system_u:object_r:etc_runtime_t
/etc/motd -- system_u:object_r:etc_runtime_t
```

2、关于与文件有关的客体使用扩展属性标记时什么是唯一的？

3、编写一个 portcon 语句，使用安全上下文 system_u:object_r:sshd_t 标记 tcp 端口 22，这个语句标记的客体类别是什么？

4、编写一个 nodecon 语句，使用安全上下文 system_u:object_r:webserver_t 标记系统 192.168.1.128，这个语句标记的客体类别是什么？

第三部分：创建和编写 SELinux 安全策略

第 11 章. 原始示例策略

如果你使用第二部分介绍的自然策略语言将所有 SELinux 策略元素组合成一个完整全面的安全策略，以满足你的目标是相当困难的，在这一章在，我们讨论过去几年不断发展演变的策略开发方法之一（由原始 NSA 示例策略驱动），让开发人员可以管理策略构建过程。

11.1. 管理构建过程的方法

如果你已经完整阅读了第二部分的内容，现在你可能非常关心构建一个完整，易于理解的 SELinux 策略，不可否认的是，SELinux 策略内容是非常多且复杂，因为 SELinux 为内核和大量的用户空间应用程序交互提供了细粒度的访问控制，在这一章中，我们讨论管理整个策略构建过程的方法，以及 SELinux 社区为这个过程注入的新的活力。

构建策略的方法是正在快速地发生改变和演变，在这一章中，我们概述一个时下流行的方法，它使用基本的策略语言工具和编译器，这种低级的开发方法是目前创建和修改 SELinux 策略占主导地位的方法，高级开发方法正处于开发之中，目前还没有正式使用。

本章我们讨论的构建策略的方法是使用示例策略，就是使用随 SELinux 发布时附带的由 NSA 开发的原始示例策略，在第 12 章中我们将讨论另一种方法，叫做参考策略。这些方法都比较低级（如都是一个树状的源模块）。

示例策略通过社区开发力量多年的发展已经超出了 NSA 发布的原始示例策略，其中一个重要的增强就是可以使用两个不同的策略源树构建 strict 和 targeted 策略，这两个示例策略变种共享通用的特性，并相互关联，strict 策略是基于最接近 NSA 原始示例策略的示例策略的，正如它名字所暗示的含义，strict 策略试图为每个合理请求一个私有域的程序提供一个域类型，strict 策略经过开源社区开发人员多年的努力，已经发展成为了反应策略语句知识最丰富的策略类型。

使用 strict 策略的一大挑战是被严格的策略限制得难受，它必然会对现有 Linux 程序造成破坏，特别是那些希望得到宽松控制的应用程序，对许多用户来讲，为了增强安全性而破坏掉这些应用程序让它们无法接受，为了解决这个问题，创建了 targeted 策略的概念，Red Hat Enterprise Linux 4 (RHEL 4) 和 Fedora Core 4 (FC 4) 默认就是使用的 targeted 策略，targeted 策略的目的是让大部分没有在 SELinux 系统上运行过的程序能够正常运行，这些应用程序叫做 unconfined，通过创建一个有权访问 SELinux 策略中所有类型的 unconfined 域来实现。

在 targeted 策略中，更具限制性的策略集中于小部分边界点，好像是附加到程序上的，如面向网络的守护进程，在 strict 策略中这些程序插入受限的域中，因此，targeted 策略比 strict 策略引起问题的可能性要小得多，但使用 targeted 策略时，安全增强更少一些，然而，对于许多系统而言，targeted 策略就已经足够了，至少比目前的安全措施强多了，同时，在刚刚使用 SELinux 时，使用 targeted 策略也是一个不错的方法。

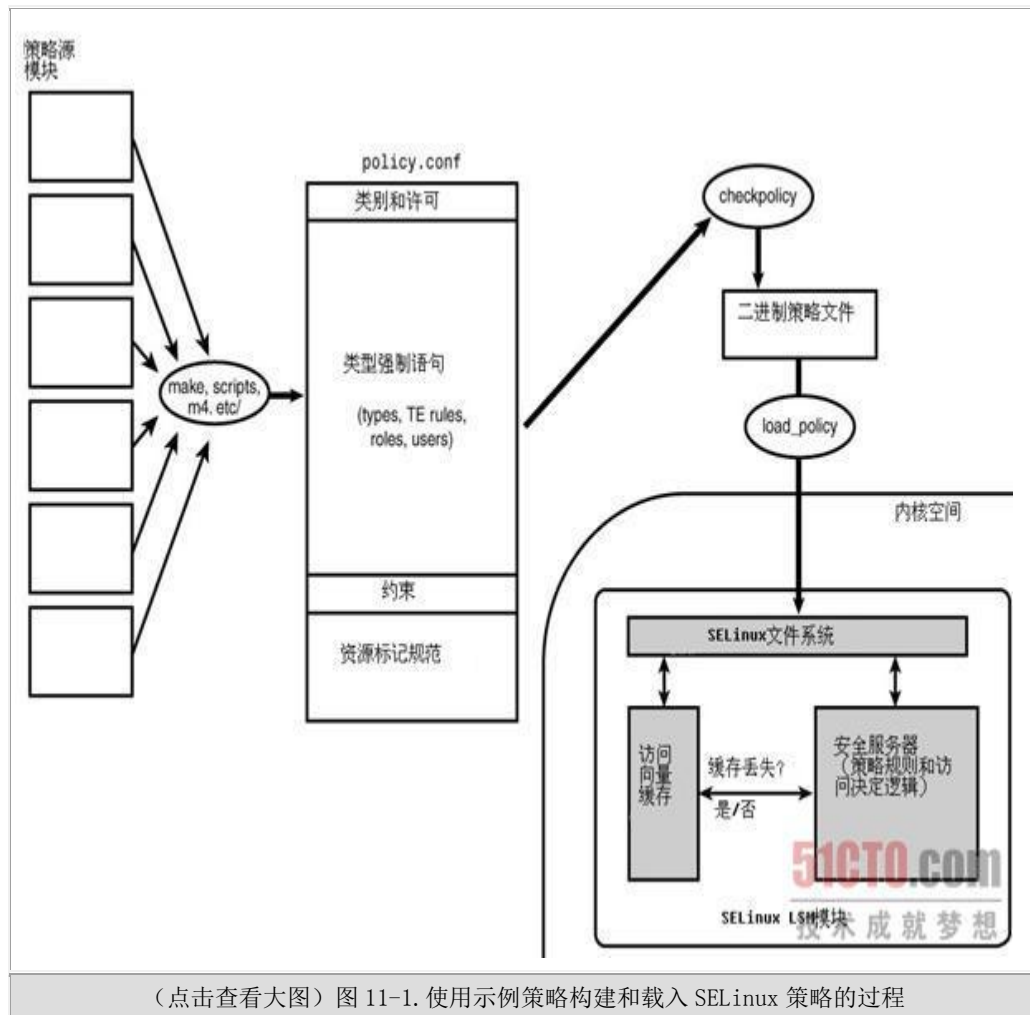
在本章的剩下部分，我们总结了 strict 策略和 targeted 示例策略源树的功能和特性。

警告：我们讨论的所有策略构建方法都处于不断发展和变化之中，注意本书中使用的策略源树在你看到时可能已经发生了变化。

11.2. strict 示例策略

strict 示例策略是最具生命力的示例策略版本，它主要通过 NSA 和 FC 邮件列表进行维护和更新，但它也随其它发行版放出。NSA 和 Red Hat 维护的示例策略本质上是相同的源树，你可以 NSA SELinux 项目网站 (<http://selinux.sourceforge.net>) 或 Red Hat FC 项目网站获得这个版本的策略，如果你的系统已经安装了 strict 示例策略（参考附录 A：获取 SELinux 示例策略），你应该可以在 /etc/selinux/strict/src/policy/ 目录下看到源代码，本书前面的示例都是来自某个版本的 strict 示例策略，我们概述的是基于 Fedora Core 4 (FC 4) 的 strict 示例策略，我们鼓励你下载最新的版本作为你策略的基线版本。

这个 strict 示例策略使用第 3 章“架构”中介绍的源模块方法构建了一个完整的策略源文件 (policy.conf)，回想一下使用一组脚本和宏创建高级结构的源模块，提供一个单一的、巨大的策略文件（如图 11-1），源模块大量使用了 m4 语言编写的宏，m4 是一个功能强大的、灵活的宏工具。



strict 策略 Makefile 支持大量的构建目标，make policy 命令编译整个策略，产生一个二进制策略文件（如 ./policy.19），这个目标对于创建一个测试策略或安装到其它系统的策略非常有用，make policy.conf 命令构造一个完整的策略源文件（./policy.conf），但它不编译策略源，make install 命令构建二进制策略文件和所有支持策略管理的文件，并进行安装，使用 strict 策略时，默认的安装目录是 /etc/selinux/strict/。这个命令只是安装新的策略但并不载入内核，可以通过重启系统或 make load 命令来载入新策略。

最后，make relabel 命令将安全上下文应用到系统中的所有文件，通常，我们在初始安装过程中和/或载入一个完整的新策略后，可能要对整个系统重新标记。

注意：重新标记整个系统不再使用 `make relabel` 命令，相反，目前重新标记整个系统时要么使用 `fixfiles relabel` 命令，要么使用 `touch ./autorelabel` 命令，或干脆重启系统。仅当文件上下文配置已经安装而不是整个策略源安装好时才可以使用这两个方法。

11.2.1.1. 客体类别和许可定义

正如在第 4 章“客体类别和许可”中讨论到的，客体类别和它们关联的许可设置是在策略语言中定义的，对于示例策略，`./flask/` 目录包括了这些定义，Flask 定义本质上是静态的，不应该被改变，内核源头文件由这些文件自动产生，因为内核和策略必须得到客体类别和它们关联的许可一致设置，这个目录中最重要的文件是：

`./flask/security_classes` 客体类别声明，参考第 4 章。

`./flask/access_vectors` 通用许可声明，以及通用许可和唯一性许可组合，参考第 4 章和附录 C：“客体类别和许可”

`./flask/initial_sids` 初始 SID 标识符声明，它用于管理默认的安全上下文标记，参考 `./initial_sid_contexts` 和第 10 章“客体标记”。

此外，这个目录也包括多个用于构造内核头文件的 shell 脚本。

警告：通常，你不应该编辑 `./flask/` 目录下的任何文件，这些文件必须与内核头文件相符，除非你确实知道你在做什么，你对内核 SELinux LSM 源代码非常熟悉，否则不要动这个目录。

11.2.1.2. 域类型和策略规则

对于一个示例策略主要的策略模块位于 `./domains/` 目录下，这个目录下有两个文件和两个子目录：

```
./domains/admin.te
./domains/user.te
```

这两个文件（.te 表示类型强制）定义了用户登陆会话的域类型，这些“用户域”与其它没有与特定程序关联的域类型不同，它们是用户类别的默认域类型，`admin.te` 文件定义了域类型 `sysadm_t`，它是最有权利的域类型，然而它在任何地方都没有特权，这个域类型是 SELinux 模拟 root 的产物。`user.te` 定义了很少特权的用户域类型 `user_t` 和 `staff_t`，这两个类型都限制了特权，适合于普通用户，它们之间最大的不同是 `staff_t` 可以转换它的角色和域类型到有特权的 `sysadm_t`。

这些文件也定义了大量的布尔变量，用于运行时提供策略配置选项。

```
./domains/program/
```

```
./domains/misc/
```

这两个目录包括了 strict 示例策略的策略源模块,大部分模块都位于 program/目录下,通常是每个域类型(或关联的域类型集)一个文件,每个策略模块包括在一个.te文件中,每个模块有一个独立的文件上下文文件,由这个文件上下文文件标识与文件有关的文件如何用客体关联的模块的类型安全上下文进行标记,本章后面我们将会介绍文件上下文。

misc/目录包括了少了的模块,通常是些不常见的域类型(如内核类型:kernel_t),也没有相关的.fc(文件上下文)文件,按功能来讲这两个目录只有少许差别,通常,你应该将新模块添加到 program/目录下。

./domains/目录被组织用来支持一个粗糙水平的策略自定义,这两个策略模块目录都有一个 unused/目录(如./domains/program/unused),在这些 unused 目录下的.te模块文件在编译时不会包括进去,与它们关联的文件上下文也不会包括进去,因此,我们可以通过将不需要的策略语句移动到 unused/目录来排除它们,本章后面我们会分析一个策略模块。

警告:示例策略不能很好地管理依赖性,因此,当你将策略模块移进/移除 unused 目录时,你可能会遇到由于有些模块要依赖这些模块而产生编译错误。

11.2.1.3. 独立的资源类型

除了前面描述的策略模块外,包含在./types/目录下的剩下大部分都是 strict 示例策略的类型声明了,这个目录下的文件通常(但不总是)只是定义了类型,而没有规则,这些类型主要是内核和关键用户空间服务的客体,它们不是活动的域类型,你可能需要改变这些文件,特别如果你想要改变与内核资源(如网络)关联的策略,这个目录下包括以下文件:

./types/device.te 这个文件定义了许多与设备文件有关的类型,包括默认设备文件类型 device_t。这个文件中定义的大部分类型都应用到/dev/目录下的客体。

./types/devpts.te 这个文件定义了 devpts 文件系统和他根目录的类型(即 Linux 下伪终端设备文件系统)。

./types/file.te 这个文件定义了常见的与文件有关的类型,包括 unlabeled_t(它常用于与文件有关的客体的类型无效时)和 file_t(它用于与文件有关的客体没有关联上下文时),这个文件声明了其它标准文件系统类型,如为/etc/目录声明的默认类型 etc_t,为/tmp/目录声明的 tmp_t 类型。

./types/network.te 这个文件定义了所有与网络有关的客体的类型(节点,网络接口,端口等),大部分保留的网络端口有它们自己的类型(ssh_port_t, dns_port_t, smtp_port_t 等),然而,你通常会发现 strict 示例策略的网络策略架构被设计为要么允许大部分或所有的网络,要么根本就禁用网络,如果你想更好地控制网络(如控制一个范围

内的节点或基于一段 IP 地址范围进行控制），或你有多个网络接口时（如一台路由器），你可能要重新改写这个文件。

`./types/nfs.te` 网络文件系统（NFS）在 SELinux 中还没有得到很好地支持，例如，网络上下文还不能在 SELinux 内核之间传递，这个文件定义了基本的 NFS 支持，`nfs_t` 类型，它用于所有的 `nfs` 文件，通常，NFS 目前还不能为类型和类型强制（TE）提供很好的支持。

`./types/procfs.te` 这个文件为 `proc` 文件系统（`/proc/`）提供类型，包括默认类型 `proc_t` 和许多特殊目的的 `proc*_t` 类型。

`./types/security.te` 这个文件定义了与 SELinux 和它的策略文件有关的各种类型，类型 `security_t` 是安全客体类别的类型，其它大部分类型都是定义来保护已经安装的策略文件，相关配置和源文件。

11.2.1.4. 其它顶层文件和目录

在顶层目录中，大部分文件都很少改动，我们在这里列出这些文件，并指明每个文件要求的规范：

`./assert.te` 这个文件与 `neverallow` 规则（即固定声明）处于同一个位置，参考第 5 章“类型强制”。

`./attrib.te` 几乎有属性声明的地方就有这个文件，策略文件中从头到尾到处都可看到属性与类型关联的语句，但约定是所有在这个文件中使用 `attribute` 语句声明的属性，所有属性都应该在这个文件中声明，并使用适当的注释解释它们的目的。

`./constraints` 这个文件位于所有非多层安全（MLS）约束定义的地方，参考第 7 章“约束”。

`./macros/` 这个目录包括了大量的 `m4` 宏文件。

`./mls` 这个文件位于所有 MLS 约束定义的地方，只有当你决定在你的策略中构建可选的 MLS 特性才会看到它，参考第 8 章“多层安全”。

`./msc` 这个文件是另一个 MLS 配置，主要使用类别而不是敏感度，这个文件与标准的 MLS 配置（`./mls`）类似，可以作为一个可选功能构建入策略。

`./rbac` 这个文件最初包括了所有的 `allow` 规则，久而久之，这些规则已经逐步转移到其它策略文件中去了。

`./users` 这个文件包括所有用户声明，通常，这个文件也会声明特殊用户 `system_u` 和 `user_u`，以及 `root` 和其它系统默认用户。

`./local.users` 这个文件是最近才增加到 SELinux 策略中的,它使系统管理员可以在没有策略源文件的情况下将本地用户添加到策略中, 这个文件将被安装在 `/etc/selinux/strict/users/local.users`, 你可以手动编辑安装后的文件, 每当策略重新载入时, 本地用户定义将被添加到内核策略中。

这些文件都相当简单, 它们的用途也非常明了, 你将会发现你有一种想修改它们的欲望。

11.2.1.5. 安全上下文标记

使用 SELinux 最大的挑战除了编写一个好的 TE 策略外, 还有一个就是确保所有的客体实例(文件, 目录, 端口, 网络接口等)使用正确的类型进行标记。当你正确编译一个 TE 策略后, 你正在调试的问题将会关联到不正确标记的客体, 在第 10 章中我们讨论了标记机制和如何标记, 接下来我们讨论如何在 `strict` 示例策略源文件中管理标记。

在策略源根目录下的很多文件都是用来标记系统的安全上下文, 特别是那些上下文标记语句:

`./initial_sid_contexts` 正如前面讨论到的, 初始 SID 是在 `./flask/initial_sids` 文件中作为 `flask` 定义的一部分定义的, 那个文件简单声明了初始 SID, `initial_sid_contexts` 文件给每个初始 SID 分配安全上下文, 例如, 初始 SID `security` 用于分配一个安全上下文给安全客体类别(它的类型应该是 `security_t`)的单个实例, 大部分初始 SID 定义没有明确标记语句的默认标记, 例如, 初始 SID `port` 为没有明确使用 `portcon` 语句定义上下文的端口客体分配默认上下文(如默认的端口类型 `port_t`)。

你可以编辑这个文件改变每个初始 SID 关联的安全上下文, 然而, 你通常会在策略中明确使用语句来定义客体实例的安全上下文(如使用 `portcon` 语句标记额外的端口)。

`./net_contexts` 这个文件包括所有与网络有关的安全上下文语句(例如: `portcon` 和 `nodecon`), 我们在 `./types/network.te` 文件中可以看到为保留端口声明的各种类型, 如为 `tcp` 端口 22 关联的类型 `ssh_port_t`,

`./genfs_contexts` 这个文件包括所有 `genfscon` 语句(即为所有不支持扩展属性的文件系统, 如 `proc`, 标记的安全上下文), 这个文件可能还包括标记 `procfs` 文件系统根目录的 `genfscon` 语句, 安全上下文包括定义在 `./types/procfs.te` 文件中的 `proc_t` 类型。

`./fs_use` 这个文件包括各种 `fs_use_*`语句, 它们为每种文件系统类型定义了如何进行客体标记。

基于磁盘的客体标记叫做文件上下文标记, 正如我们在第 10 章讨论的。

为了创建一个初始文件上下文标记策略, 我们要使用 `./file_contexts/`目录, 这个目录包括大量的文件, 它们和 `./file_contexts/misc/`目录下的文件一起用于创建一个完整的文

件上下文文件，用于标记和重新标记所有基于磁盘的文件系统，`./file_contexts/program/`目录直接与`./domains/program/`目录关联，这个目录包括`.fc`文件（即文件上下文），在构建一个文件上下文文件时，只有那些`.fc`文件关联的策略模块`.te`文件目前在使用（即不在`./domains/program/`目录下的`unused`目录中），因此，你可以使用相同的方法同时关联`TE`文件（`.TE`）和关联的安全上下文文件（`.fc`）。

文件`./file_contexts/types.fc`定义的标记语句不是具体针对哪个程序模块（如：如何标记`/etc/`目录下的文件），这个文件总是包括在`strict`示例策略构建中，文件`./file_contexts/distro.fc`与`types.fc`类似，但它多包括了具体某个发行版的配置选项。

文件`./file_contexts/home_fir_template`包括为用户`home`目录中文件和子目录标记的文件标记命令，这个文件是个临时文件，因此，用户`home`目录进行标记时要依赖于用户的角色，这个文件在管理策略时也会用到（参考第13章“管理SELinux系统”），这些文件和使用中的`.fc`文件一起在构建过程中合并成一个`file_contexts`文件，这个文件是`setfiles`程序（和其它使用`matchpathcon(3)`库的有关程序）用来设置和修复基于磁盘的客体标记。

11.2.1.6. 应用程序配置文件

`./appconfig/`目录包括一套指定当前系统中使用到的服务和应用程序的安全上下文的文件，这些文件安装在运行中的策略目录中（如`/etc/selinux/strict/contexts`），我们将在第13章中讨论这些文件的用途。

11.2.2. 分析示例策略模块

为了帮助理解`strict`示例策略，以及它是如何管理策略构建过程的，让我们分析一个示例策略模块，我们来看一个`ping`程序策略模块，清单11-1显示了这个模块的一部分内容，你应该在`./domains/program/ping.te`或`./domains/program/unused/ping.te`发现一个类似的模块。

清单 11-1. 来自`strict`示例策略（`ping.te`）为`ping`程序制定的策略模块

```
1 type ping_t, domain, privlog, nscd_client_domain;
2 role sysadm_r types ping_t;
3 role system_r types ping_t;
4 in_user_role(ping_t)
5 type ping_exec_t, file_type, sysadmfile, exec_type;
6
7 # Transition into this domain when you run this program.
8 domain_auto_trans(sysadm_t, ping_exec_t, ping_t)
9 domain_auto_trans(initrc_t, ping_exec_t, ping_t)
10 bool user_ping false;
```

```

11 if (user_ping) {
12     domain_auto_trans(unpriv_userdomain,
ping_exec_t, ping_t)
13     # allow access to the terminal
14     allow ping_t { ttyfile ptyfile }:chr_file
rw_file_perms;
15     ifdef(`gnome-pty-helper.te', `allow ping_t
gphdomain:fd use;')
16 }
17
18 uses_shlib(ping_t)
19 can_network_client(ping_t)
20 can_resolve(ping_t)
21 allow ping_t dns_port_t:tcp_socket name_connect;
22 can_yplib(ping_t)
23 allow ping_t etc_t:file { getattr read };
24 allow ping_t self:unix_stream_socket
create_socket_perms;
25
26 # Let ping create raw ICMP packets.
27 allow ping_t self:rawip_socket {create ioctl read write
bind getopt
setopt };
28
29 # Use capabilities.
30 allow ping_t self:capability { net_raw setuid };
31
32 # Access the terminal.
33 allow ping_t admin_tty_type:chr_file rw_file_perms;
34 allow ping_t privfd:fd use;
35 dontaudit ping_t fs_t:filesystem getattr;
36
37 # it tries to access /var/run
38 dontaudit ping_t var_t:dir search;
39 ifdef(`hide_broken_symptoms', `
40     dontaudit ping_t init_t:fd use;
41 ')

```

注意第 4, 8, 9, 12, 18 到 20, 和 22 行包括的宏, 它们不是策略语言语句, 使用 m4 宏处理器的宏在示例策略源文件中很常见, 在这一章中, 我们将分析其中一部分宏, 在策略编译过程中, 这些行将会包括进模块源文件中。

11.2.2.1. 定义类型和域

第 1 行和第 5 行定义了两个类型 `ping_t` 和 `ping_exec_t`，类型 `ping_t` 是为 `ping` 程序制定的域类型，类型 `ping_exec_t` 与磁盘上可执行的 `ping` 文件关联的类型，域类型和关联的可执行文件类型（通常以 `_exec_t` 结尾），正如你看到的，每个类型都关联了多个属性，例如，类型 `ping_t` 关联的 `domain` 属性，在 `strict` 示例策略中，所有域类型都有这个属性。

第 2 行和第 3 行将 `ping` 域类型与角色 `sysadm_r` 关联起来了，这个角色是一个具有特权的用户角色，它本身是为系统进程配置的，第 4 行也为 `ping` 域类型关联了一个角色，但使用的是宏 `in_user_role()`，在 `./macros/user_macros.te` 文件中，我们发现这个宏的定义如下：

```
define(`in_user_role', `  
role user_r types $1;  
role staff_r types $1;  
' )
```

正如你看到的，这个宏给域类型额外关联了两个角色：`user_r`，它本身是为普通用户定义的角色；`staff_r`，它是经过授权可以将角色改变为具有特权的 `sysadm_r` 的非特权角色。

注意：`m4` 宏使用字符串代替参数，如 `$1` 代表第一个参数，`$2` 代表第二个，以此类推，在 `ping` 模块的第 4 行，我们可以看到使用了 `in_user_role` 宏：

```
in_user_role(ping_t)
```

这个调用提供了一个参数 `ping_t`，它就代表 `$1`。

用户角色和域类型

在 `strict` 示例策略中，通过一个域类型和三个标准用户角色（`sysadm_t`，`staff_r` 和 `user_r`）一起定义了一个程序不用进行域转换就可以执行的权利，角色 `user_r` 对应一个标准域类型 `user_t`，同样，`sysadm_r` 就对应 `sysadm_t`，它是一个权限非常大的域类型（和标准 Linux 系统中的 `root` 差不多）。

你可以分析 `./domains/user.te` 中的非特权用户域类型（`user_t` 和 `staff_t`）的策略规则，以及 `./domains/admin.te` 中域类型（`sysadm_t`）的特权用户域类型策略规则。

11.2.2.2. 指定域转换规则

现在看第 8 和第 9 行，这里出现了两次调用 `domain_auto_trans()` 宏，这个宏可能是 `strict` 示例策略中最常用的一个宏了，因为它定义了我们在第 2 章“概念”中讨论到的允许

域转换的标准规则，你可以在 ./macros/core_macros.te 文件找到这个宏的定义，实际上宏通常很短，因为它要调用另一个宏 domain_trans()，这个宏的定义如下：

```
# $1 is original domain, $2 is executable file type, $3
is new domain
define(`domain_auto_trans',`
domain_trans($1,$2,$3)
type_transition $1 $2:process $3;
')

```

domain_auto_trans() 宏授予了必要的权限允许域转换（通过调用 domain_trans() 宏）默认通过 type_transition 规则使这个转换自动进行。

如果我们进一步分析 domain_trans() 宏，我们会发现更多规则，它们大部分都是标识进程间（父进程和子进程间）通信（IPC）需要的许可，然而，这个宏还包括了三个小型 allow 规则，如：

```
# 来自 domain_trans macro 的关键规则
# $1 是原始域，$2 是可执行文件类型，$3 是新域
define(`domain_trans',`
allow $1 $3:process transition; # 旧域可以转换成新域
allow $1 $2:file { read x_file_perms }; # 旧域可执行文件
类型
allow $3 $2:file entrypoint; # 新域可以送入的文件类型
# 剩下的 domain_trans 规则没有显示
')

```

注意第二个规则中许可字段中有 read 和 x_file_perms，然而 read 是针对 file 客体类别的许可，x_file_perms 却不是，相反，它是 m4 宏的另一个类型，它通常代表文件可执行许可，我们在 ./macros/core_macros.te 可以找到这个宏的定义：

```
define(`x_file_perms',`{ getattr execute }')

```

因此，我们再看 ping 模块中的第 8 行和第 9 行，我们看到具有特权的管理域类型 sysadm_t 和启动进程脚本域 initrc_t 都可以访问域 ping_t，这就清晰地表明它们可以运行 ping 程序。

11.2.2.3. 条件策略示例

从第 10 行开始，我们看到有一个条件策略块示例，第 10 行定义了布尔变量 user_ping，第 11 行到 16 行包括了使用这个布尔变量的条件子句，因此，使用了布尔变量的条件策略语

句可以控制无特权的用户域是否可以使用 ping 程序，这主要是通过第 12 行的对宏 `domain_auto_trans()` 的条件调用实现的，注意转换的原始域是一个属性 (`unpriv_userdomain`)，而不是第 8 和 9 行中的类型，这意味着所有具有该属性的类型都获得将域转换到 `ping_t` 的许可集，没有一个简单的方法确定在策略源文件中是哪些类型，通常，我们希望知道那个属性代表什么，也希望在策略源文件中没有违反这个期望的类型，唯一可行的方法就是使用 `apol` 工具来确定类型关联了哪些属性。

11.2.2.4. ping 命令的网络和其它访问

再来看一下 ping 模块，第 18 行调用了授予了 `ping_t` 域许可的宏，用来连接共享库，第 19 行到 24 行提供了大量的 ping 域将要使用到的网络和系统资源的访问许可，这些行大都调用了宏，例如：第 19 行的 `can_network_client()` 宏，它定义在 `./macros/network_macros.te` 中，注意这个宏提供了几乎所有对客户端网络的访问权，这是一个比较粗糙的许可，SELinux 允许你在网络控制方面进行更精确地控制，然而，这种类型的宏在常规用途的策略中很常见，如示例策略，正如你在第 20 到 22 行和 27 行所看到的，提供了额外的网络访问控制，花点时间仔细研究一下这些宏。

提示：

为了分析宏都做了些什么，你首先要找到它，最简单的方法就是在宏目录 `./macros` 下使用 `grep`，例如：要找 ping 模块中第 18 行中的 `uses_shlib` 宏，输入：

```
# cd /etc/selinux/strict/src/policy/macros
# grep -r uses_shlib * | grep define
global_macros.te:define('uses_shlib','
```

看 ping 模块中的第 30 行，这里有一个 `ping_t` 域类型，使用了关键字 `self` 访问它自身，即 `capability` 客体类别，这个客体类别控制 Linux 权利，对于这个客体类别自身使用域许可没什么意义，在这里，我们提供 `ping_t` 许可使用特权功能实现网络原始访问，使用 `setuid` 内核调用改变用户 ID（在这个例子中即是改成 `root`）。

第 33 行和 34 行中的 `allow` 规则提供了与显示输出终端设备交互的 `ping_t` 域许可，通常任意犯不给终端设备提供访问权的失误。

11.2.2.5. 审核规则

最后，我们有一组 `dontaudit` 规则示例，这些规则用于指出我们期望不妨碍 ping 功能的访问拒绝，对于要使用更多许可的 Linux 应用程序这并不常见，不是授予它们过度的访问权，最佳做法使用访问拒绝，但要使用 `dontaudit` 规则过滤出审核消息，因此审核日志不能弄脏了。

11.2.2.6. 文件安全上下文标记

Ping 模块最后的组件是文件上下文语句，用于正确标记与 ping 有关的文件和目录，在 `/policy/file_contexts/program/ping.fc` 中你可以找到 ping 文件上下文，如：

```
/bin/ping.* -- system_u:object_r:ping_exec_t
```

这个文件上下文说明子句使用 `setfiles` 实用程序标记 `/bin/` 目录下的任何以 ping 打头的文件(在我们的系统上包括 ping 和 ping6，它们都是标准的 ping 程序)，用包括文件可执行类型 `ping_exec_t` 的安全上下文标记。

11.2.3. strict 示例策略构建选项

Strict 示例策略源树提供了几个基础配置选项，使我们可以控制内核策略的内容，这些配置选项在不用编写策略语句的情况下允许控制最终策略的内容。

11.2.3.1. 配置策略模块

我们可以使用 `unused/` 目录来控制包括在策略中的策略模块，如果我们不想要 ping 策略模块包括在我们的策略中，我们可以将 `/domains/program/` 目录下的 `ping.te` 文件移至 `/domains/program/unused/` 目录，这样就可以阻止在策略中包括 ping.te 文件(和关联的 `ping.fc` 文件)。

你可以移除不想要的模块来定制满足你特定安装的策略，即使无关的策略模块(即与之相关的程序还没有安装)通常不会对系统的操作产生影响，但也会在内核内部增加内存的消耗，在某种情况下(如一个 Web 服务器)，缺少的策略不值得要，因为应用程序可能运行在某个用户的域下比在更多限制的浏览器域下有更多的访问权。

包括不想要的策略模块也会引发安全威胁，因为软件可能会偶然被安装而具有特权运行，如：假设我们不想任何用户运行 ping 程序，因此，我们不会安装这个软件，但我们忘记移除 ping 策略模块了，如果之后某一天，我们安装某个软件时，它包含了 ping 程序(因为它需要 ping 程序)，我们的用户突然就具备访问 ping 的权限了，如果我们在原始策略中移除了 ping 模块，当软件包安装了 ping 程序时，用户也不能使用它，因为域 `ping_t` 没有定义。

11.2.3.2. 开启可选的 MLS 特性

在本书中我们到处都可见到可选的 MLS 策略和相关的使用 MLS 可选特性的多范畴安全(MCS)配置，默认情况下，strict 示例策略配置没有开启这些 MLS 配置，为了使用 MLS，策略必须使用一个特殊的选项编译，告诉内核要使用 MLS，更重要的是，所有安全上下文都必须用请求的 MLS 敏感度进行扩展，在第 8 章有相关讨论。

Strict 示例策略 Makefile 有配置选项来自动操作这些步骤，如果你仔细看 `Makefile(/Makefile)` 文件的顶部，你会看到如下代码：

```
# Set to y if MLS is enabled in the policy.MLS=n# Set to y if MCS is enabled
in the policyMCS=n
```

设置这些标记告诉 `checkpolicy` 编译器构建策略时开启 MLS 特性，当这个策略载入内核时，将会告诉内核使用 MLS 特性进行访问控制，当它们相互排斥时你不应该开启这些选项。

只有当构建一个开启了 MLS 特性的策略文件时才使用 MLS 或 MCS 选项，不能保证所有的安全上下文规范都能够正确包括扩展的 MLS 安全上下文信息，strict 示例策略可以使用 `make` 命令对所有的安全上下文进行基本的重新配置：`make mlsconvert` 和 `make mcsconvert`。这

样将会改变策略中所有的安全上下文内容，无论如何，安全上下文的 MLS 部分对于一个真实的 MLS 系统而言都还是不够的，你可能要构建你自己的文件上下文标记所有的文件，目录，端口，网络接口等供你的 MLS 应用程序使用，如 `make mls` 将会把 `ping.te` 改成：

```
/bin/ping.* -- system_u:object_r:ping_exec_t:s0
```

查看第 8 章了解更多关于 MLS 安全上下文的信息。

警告：

无论是使用 `make mlsconvert` 还是使用 `make mcsconvert` 都会永久改变示例策略源文件中的安全上下文，目前还无法从改变后的状态还原到最初的原始状态，因此，建议你在尝试这个特性时先备份一下源策略树。

这些配置选项将会在 Makefile 中设置对应的 `MLS=y` 或 `MCS=y` 选项，如果你使用这些 `make` 目标就不需要手动设置这些选项了。

11.2.3.3. 构建时可调选项

在 `/tunables/` 目录下有两个文件：`distro.tun` 和 `tunable.tun`。这两个选项允许我们启用/禁用配置选项，`distro.tun` 文件用于特定发行版配置选项，如在我们的 FC 4 系统上，这个文件包括下面的内容：

```
define('distro_redhat')dnl define('distro_suse')dnl
define('distro_gentoo')dnl define('distro_debian')
```

这个文件指出 `distro_redhat` 选项被开启，`dnl` 是一个 `m4` 命令，意思是丢掉新行，`tunables.tun` 文件有相似的配置选项，我们可以配置与发行版无关的一些选项。

在整个策略模块中，包括在 `m4 ifdef` 子句中的语句将依赖于调试选项是启用或禁用而包括进策略或不包括进策略，如，第 39 行到 41 行的 `ping` 模块，我们在 `ifdef('hide_broken_symptoms')` 内有一个 `dontaudit` 规则，它是一个 `m4 ifdef` 语句，如果你查看 `/tunables/tunable.tun` 文件，你会看到这个选项是否启用。

11.3. targeted 示例策略

Targeted 示例策略是从 `strict` 示例策略衍生而来的，它的结构和组织几乎是一样的，但 `strict` 策略更趋向于最大化使用 SELinux 所有特性，为大部分程序提供强壮的安全保护，而 `targeted` 策略的目标是隔离高风险程序，使用 `targeted` 策略的好处是一方面可以向 Linux 系统添加大量的安全保护，同时又尽量少影响现有的用户程序，`targeted` 策略主要集中于面向网络的服务（即那些暴露在外任意遭受黑客攻击的组件），`targeted` 策略是 RHEL 和 FC 系统上标准的策略，因为它在增强安全性和减少对现有应用程序影响之间达到了一个很好的平衡。

如果安装了 `targeted` 示例策略，我们应该可以在 `/etc/selinux/targeted/src/policy/` 目录下看到它的源文件，从各个方面来看，`targeted` 示例策略源与 `strict` 示例源都非常相似，因此我们就不对 `targeted` 文件结构做详细介绍了，我们只谈一下它们之间的不同之处。

Targeted 示例策略和 `strict` 示例策略之间主要的差异是使用了无限制的域类型 `unconfined_t`，并移除了所有其它用户域类型，如 `sysadm_t` 和 `user_t`，这也意味着基本的角色结构也被移除了，所有用户都以角色 `system_r` 运行，几乎所有的用户运行的程序都以 `unconfined_t` 域类型执行。

我们可以在 `/domain/unconfined.te` 中找到 `unconfined` 域定义，注意在 `targeted` 示例策略中，`strict` 策略文件 `admin.te` 和 `user.te` 不再位于 `/domains/` 目录下，这些文件为

strict 示例策略定义了各种各样的用户域，每一个都具有受限的特权，在 targeted 示例策略中，所有程序都以 unconfined_t 域类型运行，除非它们都明确地指定了域类型(因此得名 targeted)，本质上 unconfined 域可以访问所有的 SELinux 类型，使它免除 SELinux 安全控制(因此得名 unconfined)。

在 strict 示例策略中，./domains/program/包括许多策略模块，每个模块代表一个或多个域类型和关联的类型，以及为特定程序制定的规则。在 targeted 示例策略中，这个目录包括的文件要少得多，这些就是目标。

目标示例策略模块与 strict 策略中策略模块类似，例如：我们发现 strict ping 模块和 targeted ping 模块是一致的，但有部分 targeted 模块只是简单地定义类型使域不受限制(不是 targeted)，例如：如果我们查看 crond 的 targeted 策略(crond.te)，我们会发现有一行 unconfined_domain(crond_t)。这个宏在 targeted 示例中定义

在 ./policy/macros/global_macros.te 文件中，它将 crond 域类型提供了所有 SELinux 访问权，使得它不受限制，如果我们将域 strict 版本

(/etc/selinux/strict/src/policy/domains/program/crond.te) crond 模块进行比较，会看到它们之间有很大的差异，在 targeted 策略中，crond 被认为是不受限制的域，但在这两个策略中 ping 却保留是 strict 域。

在 strict 和 targeted 示例策略间剩下的不同之处就很微小的，已经超出本书的范围，你会发现在构建目标策略(targeted)和限制策略(strict)时的选项都很相似。

11.4. 小结

Strict 策略的目标是最大化使用 SELinux 为每个程序提供独立的域类型，strict 示例策略是最直接反应 NSA 原始示例策略的策略了。

Targeted 示例策略是从 strict 示例策略衍生而来的，它的目标是使用 SELinux 隔离高风险系统服务，它将大部分程序运行在不受限制的域上，只对 targeted 服务具有增强的限制。

Strict 和 targeted 示例策略源树本质上都相似，它们都经过了很长一段时间的发展，包括了大量的文件和目录。

Strict 示例策略构建约定使用一个松散的模块化的结构，允许在每个基础域上构建策略源文件，因此，我们可以决定要包括那个程序域要剔除哪个程序域，m4 宏处理器用于提供策略源中的抽象概念。

Strict 和 targeted 策略间的主要差异是 targeted 策略限制了容易遭受攻击的服务的许可集，但没有为本地用户和程序提供额外的限制，strict 策略为所有用户和大部分应用程序以及服务定义了许可集。

FC 4 和 RHEL 4 系统使用 targeted 策略作为默认支持的策略。

第 12 章. 引用策略

引用策略是构建 SELinux 策略的另一个方法，它使得策略更容易理解、修改、维护和生效。通过现代软件工程技术，如模块和封装技术实现这一目标，利用引用策略可以从相同的源树构建 strict 和 targeted 策略变种。

12.1. 引用策略的目的

引用策略项目是为了对来源于 NSA 示例策略的策略进行重新改造，以便更容易使用、理解和维护策略。主要目的是为了给策略开发工作创建一个强健的设计原则，通过应用很好理

解的软件设计原则，保留多年从社区开发中总结出来的经验，为现有策略的开发做好基础工作，换句话说就是取其精华，去其糟粕。

现有示例策略中主要的缺陷是缺乏强有力的模块，即使向示例策略中添加了宏，实际上，所有策略标识符(类型、角色、属性等)是全局性的，编辑一个策略模块可能需要了解许多其它相互依赖的模块，同样，创建一个新的策略模块需要详细理解其它策略模块的详细实现。

引用策略的关键特性让策略开发变得更简单和更容易理解，如：

单个源树同时支持 `strict` 和 `targeted` 策略、非强制的多层安全/多范畴安全 (MLS/MCS) 扩展，单个内核策略文件(叫做单片策略)和新的可载入式模块架构。

按照强健设计原则设计的应用程序，定义精良的接口，没有全局性类型和其它标识符(因此，所有与类型相关的改变在单个模块中会完全改变)。

集成文档支持，例如：策略模块开发者使用一个接口时，不用理解该接口是如何实现的。

简化并标准化策略配置和构建选项，因此编写和自定义策略模块变得更加容易。

除了使策略开发变得更加简单外，引用策略也使得校验策略的安全属性更加简单，也增加了对高级开发工具的支持，如图形开发环境和高端策略调试器。

引用策略是个新东西，但我们希望它能普及开来，成为构建 SELinux 系统时最权威的“引用”，在编写本书时，Fedora Core 5(FC 5)已经将它支持的策略从原来的 `targeted` 示例策略改为基于引用策略的 `targeted` 策略。

警告：

在编写本书时引用策略还是一个新生事物，它的初始开发才刚刚结束，因此，在本书发行时引用策略的某些细节很可能已经发生了变化了。

在引用策略项目的网站上(<http://serefpolicy.sourceforge.net/>)有关于引用策略的更多信息和最新的策略源，如果你正在使用 FC 5 系统，你默认的 `targeted` 策略很可能就是基于引用策略构建的，如果你按照我们在附录 A“获取 SELinux 示例策略”中介绍的方法安装了引用策略，在 `/etc/selinux/refpolicy/src/policy` 中你会发现引用策略的源文件。如果你从你的发行版中获取了引用策略源树，源文件可能在 `/etc/selinux/` 目录下的不同子目录中，FC 5 将它的 `targeted` 引用策略安装在 `/etc/selinux/targeted/`，本章我们所使用的路径名都是相对于策略源根目录的。

12.2. 策略源文件结构概述

引用策略的文件结构与示例策略有些不一样，在描述引用策略的关键实现细节之前，我们先概述一下引用策略源文件的布局，熟悉一下它的文件结构。

12.2.1. 构建和支持文件

下面的文件和目录用于构建或支持引用策略的构建：

build.conf: 这个文件定义构建选项集，我们可以使用其中的选项控制构建过程，这个文件在 `make` 过程中包括在 `Makefile` 中，本章后面我们将会讨论其中一些选项的使用。

Rules.modular: 这个文件包括构建支持载入式模块的策略时使用的 `make` 规则，它支持构建基础策略模块和可载入式策略模块，它的模块作为基础模块的一部分构建，它被构建成一个可载入式模块，定义在 `policy/modules.conf` 文件中，`build.conf` 中的构建选项 `MONOLITHIC` 控制是构建一个模块还是构建一个单片策略。

Rules.monolithic: 如果正在构建一个单片策略，这个文件(不是 `Rules.modular`)被包括在 `Makefile` 中，定义构建单片策略时用到的规则。

Config/: 这个目录包括应用程序配置文件子目录, 这些配置文件与示例策略中 appconfig/目录下的文件恰好一致, 这些文件安装在可用的策略目录中, 如 /etc/selinux/refpolicy, 支持不同服务和应用程序。

Doc/: 这个目录包括的是支持文档, 要查看这些文档, 运行 make html 命令, 然后查看 doc/html 目录。

support/: 这个目录包括源代码和用于支持构建过程的工具使用的脚本。

12.2.2. 核心策略文件

在引用策略中, 用于创建策略(或可载入式模块)的主要文件放在 policy/目录下, 这些文件是我们作为策略编写者通常要修改和分析的文件:

policy/constraints: 这个文件定义了所有无 MLS 的约束, 它基本上和示例策略中的文件一致。

policy/flask/: 这个目录包括 Flask 定义, 和示例策略中的一样。

policy/mls 和 policy/mcs: 这两个文件为非强制的 MLS 特性定义了两个配置, 它们和示例策略中文件一样。

policy/global_booleans 和 policy/global_tunables: 这个两个文件目前存储定义的布尔变量和它们的默认值, 它们被合并安装在/etc/selinux/refpolicy/booleans, 管理员可以改变布尔变量的默认值, 参考第 9 章“条件策略”, global_booleans 文件包括布尔变量, global_tunables 包括构建/运行时布尔变量的配置选项。

policy/modules.conf: 这个文件配置在构建过程中要包括哪个模块, 以什么形式包括, 模块可以被构建成一个单片策略或基础模块, 而不能成为一个可载入式模块, modules.conf 文件是由 make conf 命令创建的, 本章后面我们将会讨论模块配置选项。

policy/modules/: 这个目录包括所有的策略模块, 它下面又按层分成若干个子目录, 这里面的大部分文件需要分析, 编辑和修改。

policy/support/: 这个目录包括宏, 如 policy/support/obj_perm_sets.spt 文件定义的宏定义了许可集, 我们使用这些宏简化策略编写的步骤, 创建更容易读懂的策略。

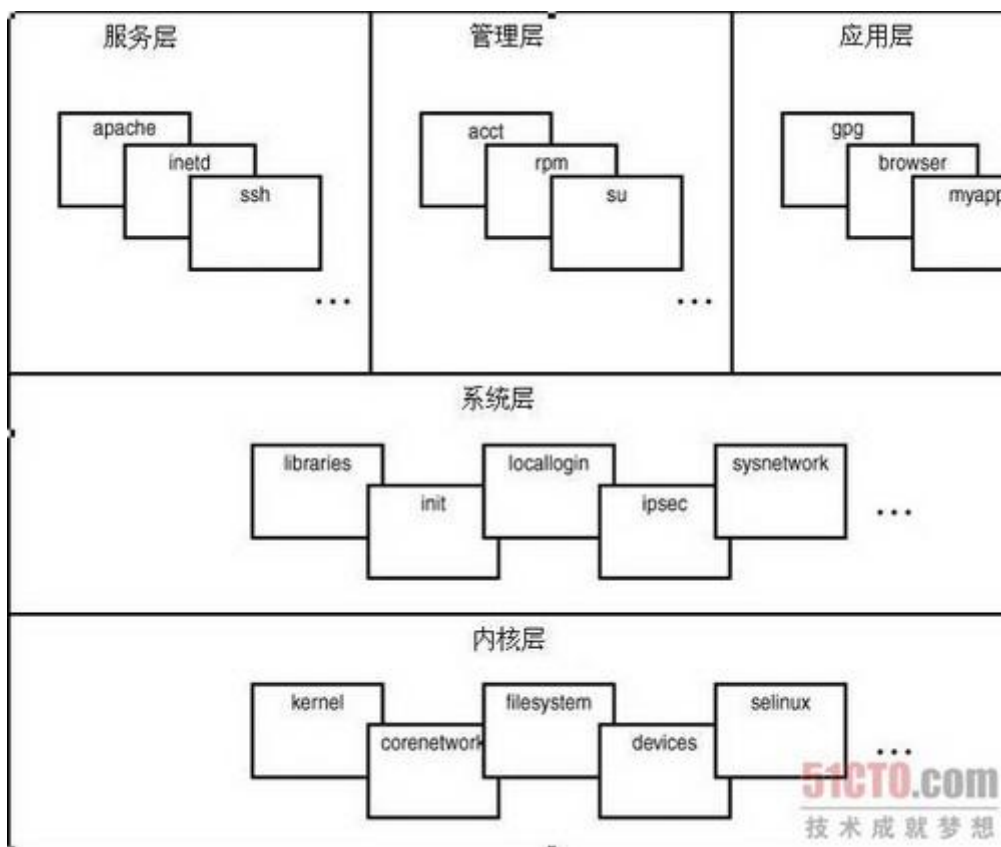
policy/users: 这个文件和示例策略中的 users 文件一样, 不过它要使用一个接口 gen_user(), 参考第 11 章中创建用户的内容。

12.3. 设计原则

引用策略是围绕几个设计原则构建的, 这些原则主要集中于实现该项目的目标, 目前, 大部分原则都是通过约定实施的, 因为高级开发环境和工具发展了引用策略, 我们希望看到这些原则在这些工具身上能够更严格地执行。

12.3.1. 分层

正如在下一节将要讨论的, 引用项目通过强健的模块组件实现了大部分它设计的目标, 但它的一个弱点也应该引起注意, 引用策略的设计原则是模块的分层, 这些层构成了一个松散的组织结构, 图 12-1 显示了目前为引用策略定义的层。



(点击查看大图)图 12-1. 引用策略层和每一层中的示例模块

通常，在一个层中，引用策略视图保持模块间的依赖性，我们可以在 `policy/modules/` 找到层的目录，它包含了每一层的模块，引用策略目前定义了下面的层：

内核层(kernel)：这一层包括了直接与 Linux 内核有关的策略模块。它是最底层的模块，在这一层的模块包括内核、设备、文件系统和基础网络策略语句，这些模块大部分都会包括在任何类型的策略中。

系统层(system)：这些策略模块通常也包括在所有策略中，但它不直接支持内核。这一层中的模块包括常用库、登陆进程和网络管理策略。

服务层(services)：这一层包括所有服务和后台进程的策略模块，这些模块从 `cron` 到 `sshd` 到 `apache`。

管理层(admin)：这一层包括管理工具和有自己域类型的管理命令的策略模块。

应用层(apps)：这一层包括所有其它有自己域类型和策略模块的程序的策略模块。

再说一次，分层不是严格实施的，主要用途是组织模块集，正如你从图 12-1 中看到的，有些层实际上是同等的组合而不是分层的(即服务层、管理层和应用程序)。

12.3.2. 模块化

模块化是引用策略最强设计原则，虽然 11 章中讨论的示例策略有模块的概念，但这些模块都比较松散(主要是由于使用了全局类型和属性名)，在引用策略中，模块要求松散地联合，它是通过两个强壮设计规范实现：封装和提取。

12.3.2.1. 封装

封装是引用策略要求类型和属性名只能在单个模块中使用的模块化规范，实际上，类型和属性名可能不能作为全局名使用，只有定义了类型/属性的模块可以直接引用这个名称，任何其它要使用类型/属性的模块必须明确定义自己拥有的模块定义的接口。

例如，在示例策略中，所有域类型都具有 domain 属性，每个策略模块都必须为它们定义的域类型明确指定 domain 属性，如果我们决定改变域在策略中是如何实现的（给每个类型授予明确的规则或重命名属性），我们不得不改变每个定义了域类型的模块。

在引用策略中，在内核层一个叫做“domain”的模块定义了域的概念，正好碰巧的是这个概念是使用 domain 属性实现的，但实现细节是域模块私有的，它的改变（如重命名）不会影响到其它模块源，那些想让它的一个类型成为域类型的模块必须调用一个定义了域模块的接口：

```
domain_type(my_type) #使一个类型成为域类型的接口
```

我们将在本章后面的小节详细讨论定义在 policy/modules/kernel/domain.if 文件中的域模块接口。

封装让我们可以将引用策略模块的实现细节对于模块不可见以实现松散组合的模块。

12.3.2.2. 提取

提取是一个预计目标，它的接口描述了它们提供了什么提取入口，以及它们不提供什么，引用策略接口的目的是为么描述提供了什么提取入口，或者是该接口开启的系统功能，要求开启入口的策略语句与接口调用程序应该无关，例如我们前面定义的让一个类型成为域类型的宏叫做 domain_type() 而不是 add_domain_attribute()。接口的目的是为了让一个类型成为域类型，通过添加 domain 属性实现这个目标只是 domain_type() 接口私有的实现细节，这个接口可以简单地为该接口提供的各个类型明确地添加规则来进行替换，如果我们选择不影响其它使用这个接口的模块我们仍然可以修改它的实现。

再看另一个例子，为了让一个目录成为一个挂载点，我们在“files”模块中调用 file_mountpoint() 宏，给所有目录类型应用 mountpoint 属性时，我们不需要知道这个接口的实现细节，然后为属性定义规则允许目录类型作为挂载点使用，作为一名策略编写者，我们所有人都要知道 file_mountpoint() 接口是如何让目录类型成为一个挂载点的。

目前，每个模块内引用策略都实现了一个低级接口，高级接口将要结合低级接口并添加额外的接口被开发出来。

12.3.2.3. 模块文件

正如前面所谈到的，引用策略源树中的所有模块都保留在 policy/modules/[layer]/中，这里的 layer 是一个目录，它的名字与前面讨论的“层”一致，每个模块必须由三个相关的文件组成，所有文件都有一个相同的根名字，即模块名：

私有策略文件(.te)：这个文件包括了模块专用的声明和规则，通常，所有模块类型和属性声明都包括在.te 文件中，以及授予这些类型和属性核心访问权的规则。

外部接口文件(.if)：这个文件包括模块接口，这些接口是其它模块访问这个模块的类型和属性。

标记策略文件(.fc)：这个文件包括与这个模块有关的文件上下文标记语句。

因为一个强壮的必要条件是类型或属性不能是全局的，只有模块中包括的.te 和.if 文件才可明确地使用模块的类型/属性名，所有其它引用该模块的类型和属性都必须通过该模块的接口。

12.3.2.4. 接口

正如前面所讨论的,在引用策略中最有效的改进之一就是使用宏接口访问定义在模块外的类型,接口提供了对模块的策略资源(如它声明的私有类型和属性)的访问入口,所有其它需要特殊访问权的模块使用同样的接口,因此,策略规则要求所有用户对接口的访问都要一致,这样在使用位于示例策略中的类型时就不同修改所有模块了。

正如上面提及的,接口都保留在模块的 .if 文件中,都是使用宏实现的,目前,引用策略支持两种类型接口:访问接口和模块接口。

我们提供的每个接口名字遵循 modname_purpose 的约定,例如我们可以告诉 domain_type() 接口定义在 domain 模块中,它的目的是让一个类型成为一个域类型(当模块名也是 purpose(目的)的一部分时,我们避免使用冗长的名字如 domain_domain_type())。

12.3.2.4.1. 访问接口

最常见的接口叫做访问接口,正如它的名字那样,访问接口的用途是为需要使用的模块私有类型和属性提供某些类型的访问,访问接口是通过宏 interface() 实现的, domain_type() 接口就是一个访问接口的例子,下面我们更详细地分析这个接口,参考清单 12-1。

清单 12-1. domain_type 访问接口部分接口清单(domain.if)

在清单 12-1 中的第 8 行,我们看到了 interface() 宏,和引用策略中的所有宏一样,它是使用一个 m4 宏实现的,我们使用 interface() 宏定义访问接口,这个宏和其它支持规范的宏和引用策略构建过程一起叫做支持宏,都放在 policy/support/目录下,interface() 宏控制接口定义的细节和调式的核心点,以及其它可以插入到结果中的构建信息,因此,我们必须使用宏 interface() 定义访问接口。

```
1 #####
2 ## <summary>
3 ## Make the specified type usable as a domain.
4 ## </summary>
5 ## <param name="type">
6 ## Type to be used as a domain type.
7 ## </param>
8 interface('domain_type', '
9     domain_base_type($1)
10
11 # 使用/dev 中信任的对象
12 dev_rw_null_dev($1)
13 dev_rw_zero_dev($1)
14 term_use_controlling_term($1)
15
16 # 读取 root 目录
17 files_list_root($1)
18
19 # 向 init 发送 sigchld 和 signull
20 init_sigchld($1)
21 init_signull($1)
```

```

22
23  ifdef('targeted_policy','
24      unconfined_use_fd($1)
25      unconfined_sigchld($1)
26  ')
27
28  tunable_policy('allow_ptrace','
29      userdom_sigchld_sysadm($1)
30  ')
31
32  # 允许任何域连接到 LDAP 服务器
33  optional_policy('ldap','
34      ldap_use($1)
35  ')
36 ')

```

domain_type() 接口的目的是允许提供的类型(这个宏唯一的参数\$1)在策略中被当做域类型使用,从第 1 行到第 7 行我们看到的是接口的描述和它的参数,引用策略使用 XML 捕获关于接口和其它策略方面的文档的消息,假如这样,我们总结了一下接口和它参数的用途,所有参数都将包括在文档中。

第 9、12 到 14、17、20 和 21 行都调用的是其它接口,接口的名字给我们一些关于模块的提示,按照规范,接口名字的第一个组件是定义接口的模块的名字,例如,接口 dev_rw_null_dev() 和 dev_rw_zero_dev() 定义在“设备”模块 policy/modules/kernel/devices.if 中,files_list_root() 接口定义在“文件”模块 policy/modules/kernel/files.if 中,我们可以分析每一个接口查看它们是如何实现的,也可以分析接口文档。

注意:命令 make html 创建的引用策略文档包括了接口的描述,用浏览器打开 doc/html/index.html 查看个究竟吧。

第 23 到 26 行显示了一个使用 m4 ifdef 的语句,尽管在示例策略中经常使用到 ifdef,但对于引用策略,使用 ifdef 按照规范要受到极大的限制,因此,如果我们构建的是 targeted 策略,我们要调用其它的接口(来自 targeted 特殊策略“unconfined”模块),符号 targeted_policy 定义在 build.conf 中,作为其构建过程的一部分,我们将在本章后面进行讨论。

第 28 行到 30 行显示了另一个支持宏 tunable_policy(),这个宏的用途是允许基于定义的可调的值额外的行为,可调的值指的是构建/安装时的策略选项,它们定义在 policy/global_tunables 中,目前,可调的选项是通过布尔变量实现的,但实际上对于可载入模块我们希望使用与条件策略布尔变量不同的方式实现可调它,因此,当真正允许管理用户域可以调式其它用户域类型时,我们就有了 allow_ptrace。

最后,分析一下第 33 到 35 行,我们有一个 optional_policy() 支持宏的例子,这个宏允许我们随意调用一个接口,依赖于模块是否包括在策略中,这个支持宏实现的功能与单片策略、基础模块或可载入模块中的有点不同,尽管如此,从一个策略编写者的视角来看,原

理是相同的，如果模块是包括在构建进程中(这里指的是 ldap 模块)，接口 ldap_use() 也会被调用。

ifdef

在引用策略中，按照规范，对于一个受限的条件的集合，可能只使用得到 m4 ifdef 语句，这些适合在策略构建过程内硬编码实现，所有其它形式的策略选项必须使用基于定义在 policy/global_booleans 中的布尔变量的活动条件策略语句(if)，或支持如定义在 policy/global_tunables 或 optional_policy() 中的 tunable_policy() 宏的引用策略，这些支持宏允许我们修改这些概念的实现，以更好地支持构建过程和将来的开发工具。

在引用策略中唯一允许使用的 ifdef 有以下定义：

targeted_policy: 当构建一个 targeted 策略时定义。

strict_policy: 当构建一个 strict 策略时定义。

enable_mls: 当构建非强制的 MLS 策略时定义。

enable_mcs: 当使用非强制的 MLS 特性构建 MCS 策略时定义。

hide_broken_symptoms: 它用来控制 dontaudit 规则，我们使用这些规则屏蔽希望拒绝的审核消息，即我们有意不允许访问，但我们希望程序去进行尝试，这些 dontaudit 规则可以帮助我们移除误报的审核消息。

direct_sysadm_daemon: 让我们决定策略是否允许系统管理员用户域直接控制后台进程，否则就是由 init 控制的，注意，如果这个选项被禁用，管理员仍然可以用 run_init 工具控制后台进程。

distro_tunable: 针对不同的 Linux 发行版有可调的参数，例如，redhat 是 Fedora Core(FC) 和 Red Hat Enterprise Linux(RHEL) 系统的可调参数，gentoo 是 Gentoo 系统的可调参数。

12.3.2.4.2. 模板接口

第二类接口是模板接口，相对访问接口，它并不常见，当两个模块共享一个或多个类型时就需要模板接口，我们调用这种类型的派生类型，派生类型的名字来源于调用的模块的类型，从逻辑视角来看，派生类型被看做是调用模块的私有类型，然而，派生类型的定义和为它们定义的访问规则用另一个模块(即调用的模块)中的模板接口实现，因为调用的模块为策略许可创建访问规则，派生类型名字在一定程度上基于调用模块的名字和被调用的模块的名字，但它们都不知道最后的名字，这样我们就可以修改模板的名字而不影响调用模块。

派生类型的例子和模板接口都可以在 ssh 模块中发现，它为 sshd 服务实现了客户端和服务端策略规则，我们可以在 policy/modules/services/ssh.* 中找到这个模块，我们想分析一下 c 模板接口，清单 12-2 列出它的一部分内容。

清单 12-2. ssh_per_userdomain_template() 模板接口的部分内容(ssh.if)

ssh_per_userdomain_template() 接口创建了一个派生类型域，允许每个域有它们私有的类型用于 ssh 会话和加密密钥，因为 ssh 模块不知道所有需要 shh 类型的域类型，它不可能直接创建它的 .te 文件需要的规则，同样，任何给定的模块想要一个 ssh 私有类型，它也不能知道如何实现 ssh 私有会话和密钥类型，因此需要一个模板接口。

```
1 #####
2 ## <summary>
```

```

3 ## The per user domain template for the ssh module.
4 ## </summary>
5 ## <desc>
6 ## <p>
7 ## This template creates a derived domains which are used
8 ## for ssh client sessions and user ssh agents. A derived
9 ## type is also created to protect the user ssh keys.
10 ## </p>
11 ## <p>
12 ## This template is invoked automatically for each user and
13 ## generally does not need to be invoked directly
14 ## by policy writers.
15 ## </p>
16 ## </desc>
17 ## <param name="userdomain_prefix">
18 ## The prefix of the user domain (for example, user
19 ## is the prefix for user_t).
20 ## </param>
21 ## <param name="user_domain">
22 ## The type of the user domain.
23 ## </param>
24 ## <param name="user_role">
25 ## The role associated with the user domain.
26 ## </param>
27 template('ssh_per_userdomain_template', '
28     #####
29     # Declarations
30
31     type $1_home_ssh_t;
32     userdom_home_file($1,$1_home_ssh_t)
33     role $3 types $1_ssh_t;
34
35     type $1_ssh_t;
36     domain_type($1_ssh_t)
37     domain_entry_file($1_ssh_t,ssh_exec_t)
38
39     type $1_ssh_agent_t;
40     domain_type($1_ssh_agent_t)
41     domain_entry_file($1_ssh_agent_t,ssh_agent_exec_t)
42     role $3 types $1_ssh_agent_t;
43
44     type $1_ssh_keysign_t; #, nscd_client_domain;
45     domain_type($1_ssh_keysign_t)
46     domain_entry_file($1_ssh_keysign_t,ssh_keysign_exec_t)

```



```

47     role $3 types $1_ssh_keysign_t;
48
49     # Private policy for each derived types not shown
50     #         see policy/modules/ssh.if
51
52     # remainder not shown...
53 ')m32

```

正如你在清单 12-2 中看到的，这个接口有三个参数：类型名前缀(如域类型 `user_t`，我们提供前缀 `user`)，用户域类型(如 `user_t`)，与用户域关联的主要角色。模板接口分为两个部分，第一部分是派生类型创建的地方，使用提供的类型名前缀，我们在清单 12-2 中第 21 到 47 行可以看到这些声明，例如，第 35 行定义了主要的派生类型，如果前缀名是 `user`，派生域类型将会是 `user_ssh_t`，它就是 `ssh` 客户端的域类型，正如你看到的，不同方向的 `ssh` 会话也创建了三个其它派生类型。

模板接口的第二部分是派生类型的私有规则，这些规则是为所有派生类型定义的，要修改的话只需要修改一个地方就可以了，本书中我们就不分析这些规则了，但我们鼓励你去分析一下，即使模板接口不常见，但它们对于策略编写时简化类型是非常有价值的。

12.4 分析引用策略模块

为了帮助深入理解引用策略是如何工作的，我们分析示例策略中为 `ping` 程序制定的策略的所有方面，尽管在示例策略中 `ping` 程序有它自己的模块，在引用策略中 `ping` 包括在一个标识所有管理网络实用程序 `netutils` 的模块中，我们可以在 `policy/modules/admin/netutils` 中发现这个模块。

注意：在引用策略中，我们尝试将策略片打包，便于安装时好理解，引用策略主要受 FC 打包规范的影响，允许我们定义模块为可载入式模块，作为安装包的一部分，这就是为什么 `ping` 是和其它网络实用程序组合在一起的缘故，这些实用程序都是 FC 中相同软件包的一部分，具体就是 `iputils` 软件包。

清单 12-3 显示了一部分 `netutils.te` 文件的内容，把这些与 `ping` 有关的组件集中在一起，回顾一下 `.te` 文件就是包括模块私有声明和规则的文件，我们通常可以在这个文件中发现 `type` 和 `attribute` 声明，首先注意在第 1 行中使用了 `policy_module()` 支持宏，所有模块都必须使用 `policy_module()` 作为它们 `.te` 文件的第一行，这个宏需要两个参数：模块名和模块版本。目前，`policy_module()` 支持宏只影响当模块作为可载入模块构建时的构建过程，尽管如此，所有模块都强制要求使用它，它的功能将会扩展，如更好的调式支持。

清单 12-3. `netutils(ping)` 私有模块文件(`netutils.te`)部分内容

从第 4 行到第 7 行，我们定义了域类型 `type_t` 和入口点类型 `ping_exec_t`，这两个类型和示例策略的用途一样，引用策略实现域类型的目标是与示例策略中的规则起相同作用，但实现方法完全不一样，注意第 6 行我们调用了来自 `init` 模块的接口，`init` 模块允许 `ping` 域在系统初始化脚本中使用，这个接口完成的任务和清单 11-1 示例策略 `ping` 模块中第 9 行调用的宏几乎一样。

```

1 policy_module(netutils,1.0)
2 #####

```

```
3 # Declarations
4 type ping_t;
5 type ping_exec_t;
6 init_system_domain(ping_t,ping_exec_t)
7 role system_r types ping_t;
8
9 #####
10 # Ping local policy
11 allow ping_t self:capability { setuid net_raw };
12 dontaudit ping_t self:capability sys_tty_config;
13
14 allow ping_t self:tcp_socket create_socket_perms;
15 allow ping_t self:udp_socket create_socket_perms;
16 allow ping_t self:rawip_socket { create ioctl read write bind
getopt setopt };
17
18 corenet_tcp_sendrecv_all_if(ping_t)
19 corenet_udp_sendrecv_all_if(ping_t)
20 corenet_raw_sendrecv_all_if(ping_t)
21 corenet_raw_sendrecv_all_nodes(ping_t)
22 corenet_tcp_sendrecv_all_nodes(ping_t)
23 corenet_udp_sendrecv_all_nodes(ping_t)
24 corenet_tcp_sendrecv_all_ports(ping_t)
25 corenet_udp_sendrecv_all_ports(ping_t)
26 corenet_udp_bind_all_nodes(ping_t)
27 corenet_tcp_bind_all_nodes(ping_t)
28
29 fs_dontaudit_getattr_xattr_fs(ping_t)
30
31 domain_use_wide_inherit_fd(ping_t)
32
33 files_read_etc_files(ping_t)
34 files_dontaudit_search_var(ping_t)
35
36 libs_use_ld_so(ping_t)
37 libs_use_shared_libs(ping_t)
38
39 sysnet_read_config(ping_t)
40 sysnet_dns_name_resolve(ping_t)
41
42 logging_send_syslog_msg(ping_t)
43
44 ifdef('hide_broken_symptoms', '
45     init_dontaudit_use_fd(ping_t)
```

```

46 ')
47
48 ifdef('targeted_policy', '
49     term_use_unallocated_tty(ping_t)
50     term_use_generic_pty(ping_t)
51     term_use_all_user_ttys(ping_t)
52     term_use_all_user_ptys(ping_t)
53 ', '
54     tunable_policy('user_ping', '
55         term_use_all_user_ttys(ping_t)
56         term_use_all_user_ptys(ping_t)
57     ')
58 ')

```

清单 12-3 中剩下的行实现了允许 ping 域类型需要的访问规则，例如，第 18 到 27 行的接口调用提供了需要的网络访问，通过使用来自核心网络模块 corenetwork 的接口实现，更多的访问也是通过接口提供的，这些期望的模块实现形式，因此访问就只需要在一个地方定义，接口可以在任何地方调用定义的访问即可，我们可以分析每个接口的用途，详细阅读各个接口的实现细节文档，和你阅读这个清单一样，你会注意到在第 48 行使用了 target_policy 条件定义只适合于 targeted 策略的规则。

下面我们来看一看适合于 ping 的接口，它定义在 netutils 接口文件 netutils.if 中，它的内容显示在清单 12-4 中，除了定义接口本身外，.if 文件还包括用于生成文档的 XML 语句，正如你在清单 12-4 中第 1 行所看到的，所有模块接口文件都必须以一个摘要语句开头，简要说明模块的用途，因为 ping 是网络实用程序模块中的一部分，我们看到的是对整个大模块的用途描述。

清单 12-4.netutils(ping)接口模块文件(netutils.if)的部分内容

从第 11 行到 16 行，41 和 43，65 到 68 以及 86 到 88 行，我们看到使用了另一个支持宏 gen_require()，这个宏是支持可载入模块结构的关键，支持开发工具需要的模块和接口依赖信息，每个模块接口文件必须要有 gen_require() 宏，用它列出该接口使用的策略标识符(类型、属性、角色、布尔变量等的名字)。对于类型和属性，这些标识符对于模块而言必须是私有的类型和属性，因为私有类型和属性可以在模块内明确地命名，gen_require() 宏将产生适当的依赖信息以支持策略构建时的不同类型，这样就允许不用整个策略源就可以链接到可载入模块了。

```

1 ## <summary>Network analysis utilities</summary>
2
3 #####
4 ## <summary>
5 ## Execute ping in the ping domain.
6 ## </summary>
7 ## <param name="domain">
8 ## The type of the process performing this action.
9 ## </param>

```

```

10 interface('netutils_domtrans_ping','
11     gen_require('
12         type ping_t, ping_exec_t;
13         class process sigchld;
14         class fd use;
15         class fifo_file rw_file_perms;
16     ')
17
18     domain_auto_trans($1,ping_exec_t,ping_t)
19
20     allow $1 ping_t:fd use;
21     allow ping_t $1:fd use;
22     allow ping_t $1:fifo_file rw_file_perms;
23     allow ping_t $1:process sigchld;
24 ')
25
26 #####
27 ## <summary>
28 ## Execute ping in the ping domain, and
29 ## allow the specified role the ping domain.
30 ## </summary>
31 ## <param name="domain">
32 ## The type of the process performing this action.
33 ## </param>
34 ## <param name="role">
35 ## The role to be allowed the ping domain.
36 ## </param>
37 ## <param name="terminal">
38 ## The type of the terminal allow the ping domain to use.
39 ## </param>
40 interface('netutils_run_ping','
41     gen_require('
42         type ping_t;
43     ')
44
45     netutils_domtrans_ping($1)
46     role $2 types ping_t;
47     allow ping_t $3:chr_file rw_term_perms;
48 ')
49
50 #####
51 ## <summary>
52 ## Conditionally execute ping in the ping domain, and
53 ## allow the specified role the ping domain.

```

```

54 ## </summary>
55 ## <param name="domain">
56 ## The type of the process performing this action.
57 ## </param>
58 ## <param name="role">
59 ## The role to be allowed the ping domain.
60 ## </param>
61 ## <param name="terminal">
62 ## The type of the terminal allow the ping domain to use.
63 ## </param>
64 interface('netutils_run_ping_cond', '
65     gen_require('
66         type ping_t;
67         bool user_ping;
68     ')
69
70     role $2 types ping_t;
71
72     if ( user_ping ) {
73         netutils_domtrans_ping($1)
74         allow ping_t $3:chr_file rw_term_perms;
75     }
76 ')
77
78 #####
79 ## <summary>
80 ## Execute ping in the caller domain.
81 ## </summary>
82 ## <param name="domain">
83 ## The type of the process performing this action.
84 ## </param>
85 interface('netutils_exec_ping', '
86     gen_require('
87         type ping_exec_t;
88     ')
89
90     can_exec($1,ping_exec_t)
91 ')

```

清单 12-4 中 .if 文件剩下的部分定义了四个与 ping 有关的接口，这些接口都是使用了 Interface() 宏的访问接口，第一个接口是 netutils_domtrans_ping()，它定义在第 3 到 24 行，满足所有规则允许域类型许可产生一个域转换到 ping 域类型，接下来两个接口是 netutils_run_ping()，定义在 26 到 48 行，netutils_run_ping_cond() 接口定义在 50 到

76 行, 调用 `netutils_domtrans_ping()` 接口但也需要一个角色确保该角色是经过 ping 域授权的, 这两个接口中的后一个接口支持使用基于 `user_ping` 布尔变量的条件表达式(第 72 到 75 行), 和第 11 章中讨论的示例策略一样。

最后一个接口是 `netutils_exec_ping()`, 定义在 78 到 91 行, 允许提供的域类型不用域转换就可以执行 ping 程序, 因此, 提供的域类型必须有足够的网络访问权, 如系统实用程序和后台进程。

最后, 我们来看一下文件标记策略, 可以在 `netutils.fc` 文件找到它, 在这个文件中, 可以看到类似下面这样的内容:

```
/bin/ping.* -- gen_context(system_u:object_r:ping_exec_t,s0)
```

这一行与 ping 的文件上下文文件有点类似, 只有一个明显的不同: 文件上下文是在另一个支持宏 `gen_context()` 内提供的, 这个宏包括一个完整的安全上下文, 包括任何非强制的 MLS 选项, `gen_context()` 宏基于构建类型可以使用 MLS 选项也可以不使用 MLS 选项产生安全上下文, 因此, 我们可以编写一个有 MLS 特性或没有 MLS 特性的策略, 而不用改变文件的内容或导致示例策略源不可撤销的改变。

12.5. 引用策略构建选项

引用策略被设计为不用理解策略的所有细节而进行策略自定义, 引用策略构建的主要目标是与示例策略中相同的名字和功能。例如, Makefile 目标, policy, policy.conf, relabel 和 load, 所有产品与第 11 章示例策略中讨论的结果都一样。

有两个引用策略构建配置文件: `build.conf` 和 `modules.conf`。

12.5.1. build.conf

本章前面我们已经讨论过一些由 `build.conf` 控制的构建选项了, 我们想讨论第一个选项是策略类型(policy type), 引用策略的目标之一就是创建与源树不同的类型, 这个构建选项控制构建什么类型, 在 `build.conf` 中具体就是用 TYPE 选项, 如果我们想构建一个 strict 策略, 我们可以为该选项指定下面这样的值:

```
TYPE = strict
```

对于 targeted 策略, 我们可以用 targeted 代替, 此外, 我们可以开启 MLS 特性, 有两种方法, 一是使用常见的 MLS 策略(strict-mls 或 targeted-mls), 二是使用 MCS 配置(strict-mcs 或 targeted-mcs), 目前引用策略只支持这六个值(strict, targeted, strict-mls, targeted-mls, strict-mcs 和 targeted-mcs)。

`build.conf` 中另一个选项是策略名字(policy name), 就是 NAME 选项, 在引用策略中它是一个很好的特性, 允许我们命名策略中除了策略类型外的其它事物, 这个名字用于决定 `/etc/selinux/` 中的策略的安装目录, 例如, 引用策略默认提供的策略名:

```
NAME = refpolicy
```

因此, 当我们安装策略时, 安装目录就是 `/etc/selinux/refpolicy/`, 如果没有提供任何值, 将会使用策略类型名, 例如, 假设我们的 `build.conf` 文件有以下两行:

```
TYPE = targetedNAME =
```

我们的安装目录将会是 `/etc/selinux/targeted/`, 如果你想使用引用策略替换你默认的 targeted 策略就可以这么做, 如果你试图对引用策略进行实验, 你可能想不要覆盖当前的, 系统提供的 targeted 策略。

另一个有意思的选项是发行调试选项 DISTRO，前面已经讨论过，引用策略支持为不同发行版本调整不同的选项，例如，对于 FC 和 RHEL 系统，这个选项被设置为：

```
DISTRO = redhat
```

我们想讨论 build.conf 的最后一个选项是策略是否是单片策略，这个选项是由 MONOLITHIC 选项控制的，如果我们构建的是一个单片策略，即一个完整的内核二进制策略，我们可以将这个选项设置为：

```
MONOLITHIC=y
```

否则，我们应该将其设置为：

```
MONOLITHIC=n
```

这里的 n 表示我们想支持可载入的模块，并且将同时构建基础模块和可载入模块，可载入模块是基础模块的一部分，只不过它是可载入的，由 modules.conf 控制，接下来我们就讨论它。

12.5.2. modules.conf

modules.conf 文件控制我们的策略中包括哪个模块，以及以什么形式包括进来，我们可以在 policy/modules.conf 发现这个文件，如果这个文件不存在，你可以在策略根目录位置使用 make conf 命令来创建它，这个命令创建一个 modules.conf 文件，为 policy/modules/ 目录下的每个模块创建一个条目，如果 modules.conf 文件已经存在，make conf 将会把新加入的模块追加到文件的末尾，对现有的模块不会有任何影响，因此，当我们添加新的模块时，我们运行 make conf 然后为这些新的模块修改 modules.conf 中的设置。

下面是 modules.conf 文件中的一个条目：

```
# Layer: admin1# Module: netutils## Network analysis utilities#netutils = module
```

这是前面讨论过的 netutils 模块产生的条目，注释行(前面是#符号)用来写明模块的用途，layer 注释来自采访模块文件的目录名，模块名注释来自模块文件的根名，描述注释来自模块.if 文件中顶层的模块摘要描述。

唯一起作用的行是 netutils = module，它告诉策略构建工具在构建过程中如何处理这个模块，module 可以有三个值，依赖于构建的类型(单片还是可载入模块)，这些值决定了如何构建模块，有以下三个值：

base：对于单片策略，所有模块都将使用 base 进行标记，对于可载入模块，基础模块中的所有模块都必须用 base 进行标记。

module：对于单片策略，策略中包括的所有标记为 module 的模块都将和 base 模块一样处理，对于可载入模块，所有标记为 module 的模块都将被构建为可载入模块。

off：对于单片和可载入模块，所有标记为 off 的模块不构建。

policy/module/ 目录中的所有模块不是都列在 modules.conf 文件中了，或者是列出但没有赋值，这些模块也不会被构建进去，和标记为 off 的作用一样。

当使用 make conf 命令创建或更新 modules.conf 文件夹时，所有模块都将被标记为 module，除非模块在模块接口文件(.if)内被要求标记，例如，下面是内核模块.if 文件的头部，内核模块总是需要的：

```
##
```

```
## Policy for kernel threads, proc filesystem, and## unlabeled processes and objects.## ## ## This module has initial SIDs.##
```

以开头的块表示这个模块和注释一起都是需要的，对于所有这样的模块而言，`modules.conf` 设置的默认值是 `base`，确保模块总是包括在单片策略或作为可载入模块策略中基础策略的一部分，因此，但我们产生 `modules.conf` 文件时，内核模块块的内容看起来如下：

```
# Layer: kernel# Module: kernel# Required in base## Policy for kernel threads, proc filesystem, and# unlabeled processes and objects.#kernel = base
```

正如你所看到的，除了设置默认的值 `base` 外，这里也有额外的注释，解释将来如何引用这个模块。

12.6. 小结

引用策略从再造示例开始，再造的目标包括现代软件工程设计原则，让策略开发和维护变得更加简单易行，并且也支持衍生技术，如可载入模块和老牌策略开发工具。

分层是引用策略一个很弱的设计原则，以分层方式组织的策略模块通常反应了我们理解的策略模块是如何关联的。

模块化是引用策略的推荐的设计原则，即使示例策略有某种形式的模块，它也定义得很弱，不能确保模块保持松散组合，引用策略模块隐藏了其它模块的实现细节，对于整个策略和分布式策略开发变得更加简单。

模块化最主要的两个属性确保引用策略模块保持松散组合：封装和提取。

封装确保模块的实现细节只由模块本身需要，这个目标是通过类型和属性名保留本地标识符，只能由定义它们的模块明确使用来实现的，其它使用这些类型和属性的模块需要通过模块的接口进行调用。

提取确保策略模块编写者可以思考策略开发逻辑而不是集中于策略的细节，这是通过模块接口实现的，接口描述接口提供了什么没有提供什么。

一个模块 由三部分组成：私有策略文件(`.te`)，外部接口文件(`.if`)和标记策略文件(`.fc`)。这三个文件在每个定义的模块中必须存在，即使是空的也要有。

引用策略模块有两种类型的接口：访问接口和模板接口。访问接口到目前为止用得最多，这些接口提供对模块的私有类型和属性的访问权。模板接口很少使用，当需要管理两个模块之间的派生类型时才使用。

引用策略引入了两个配置文件，提供策略构建时的配置选项，`build.conf` 文件控制全局策略构建选项，如策略类型和安装位置，`modules.conf` 文件控制策略模块以什么形式进行构建。

目前，引用策略可以从相同的源树构建六类策略：`strict`，`targeted`，`strict-mls`，`targeted-mls`，`strict-mcs` 和 `targeted-mcs`。

引用策略支持两种构建，单片构建创建一个内核二进制策略，单片策略是目前常用的策略类型，可载入模块构建创建一个基础模块和大量的可载入模块，我们希望将来可载入模块能够得到更多的使用。

第 13 章. 管理 SELinux 系统

SELinux 系统从许多方面看起来感觉和其它 Linux 系统一样，Red Hat Enterprise Linux(RHEL)就是一个 SELinux 系统，但它增强了安全性，由于多种原因，一些之前能够好

好工作的东西现在被破坏了，或者变得不能工作了，要修复这些问题可能需要额外的管理程序，一个正常的操作现在可能需要更多的步骤，在这一章中，我们讨论 SELinux 影响到 Linux 管理员的哪些地方，以及如何实现常见的重要的任务。

13.1 SELinux 配置和策略管理文件

SELinux 包括了允许管理 SELinux 特定增强的文件，包括策略，这包括设置当安装了多个策略时使用哪个策略，标记管理文件和 SELinux 应用程序和实用程序的配置文件。

注意：本章中涉及到的文件都是基于 Fedora Core 4(FC 4)系统的，域 RHEL 4 相比较可能稍微有所不同，而且 FC 5 中肯定又有很多增强了，因此我们会标明不同的地方。

13.1.1. SELinux 配置文件(/etc/selinux/config)

SELinux 配置文件/etc/selinux/config 控制系统下一次启动过程中载入哪个策略，以及系统运行在哪个模式下，我们可以使用 `sestatus` 命令确定当前 SELinux 的状态，清单 13-1 显示了一个 config 文件的例子：

清单 13-1. /etc/selinux/config 文件的内容

```
1 # This file controls the state of SELinux on the system.
2 # SELINUX= can take one of these three values:
3 # enforcing - SELinux security policy is enforced.
4 # permissive - SELinux prints warnings instead of enforcing.
5 # disabled - SELinux is fully disabled.
6 SELINUX=enforcing
7 # SELINUXTYPE= type of policy in use. Possible values are:
8 # targeted - Only targeted network daemons are protected.
9 # strict - Full SELinux protection.
10 SELINUXTYPE=strict
```

这个文件控制两个配置设置：SELinux 模式和活动策略。SELinux 模式(由第 6 行的 SELINUX 选项确定)可以被设置为 `enforcing`，`permissive` 或 `disabled`。在 `enforcing` 模式下，策略被完整执行，这是 SELinux 的主要模式，应该在所有要求增强 Linux 安全性的操作系统上使用。在 `permissive` 模式下，策略规则不被强制执行，相反，只是审核遭受拒绝的消息，除此之外，SELinux 不会影响系统的安全性，这个模式在调试和测试一个策略时非常有用。

在 `disabled` 模式下，SELinux 内核机制是完全关闭了的，只有系统启动时策略载入前系统才会处于 `disabled` 模式，这个模式和 `permissive` 模式有所不同，`permissive` 模式有 SELinux 内核特征操作，但不会拒绝任何访问，只是进行审核，在 `disabled` 模式下，SELinux 将不会有任何动作，只有在极端环境下才使用这个模式，例如，当策略错误阻止你登陆系统时，即使在 `permissive` 模式下也有可能发生这种事情，或我们不想使用 SELinux 时。

警告：在 `enforcing` 和 `permissive` 模式或 `disabled` 模式之间切换时要小心，当你返回 `enforcing` 模式时，通常会导致文件标记不一致，本章后面我们将会讨论如何修复文件标记问题。

SELinux 配置文件中的模式设置由 `init` 使用，在它载入初始策略前配置 SELinux 使用。

SELinux 配置文件中的 SELINUXTYPE 选项告诉 `init` 在系统启动过程中载入哪个策略，这里设置的字符串必须匹配你用来存储二进制策略版本的目录名，例如，我们使用 `strict` 策略作为例子，因此我们设置 `SELINUXTYPE=strict`，确保我们想要内核使用的策略位于 `/etc/selinux/strict/policy/`，如果我们已经创建了我们自己的自定义策略，如 `custom_policy`，我们应该将这个选项设置为 `SELINUXTYPE=custom_policy`，确保我们编译的策略位于 `/etc/selinux/custom_policy/policy/`。

FC 和 RHEL 系统提供了一个图形工具(system-config-securitylevel)，我们可以使用它来设置 SELinux 配置文件选项，这样就不需要直接编辑文件了，前面两个选择框设置 SELINUX 选项，策略类型下拉列表允许我们从安装好的策略中选择一个活动策略。

13.1.2. 策略目录

和 FC 3(RHEL 4)一样，系统上安装的每个策略在/etc/selinux/目录下都它们自己的目录，子目录的名字对应于策略的名字(如，strict，targeted，refpolicy等)，在 SELinux 配置文件中就要使用这些子目录名字，告诉内核在启动时载入哪个策略，在本章中提到的所有路径都是相对域策略目录路径/etc/selinux/[policy]/的，下面是 FC 4 系统上 /etc/selinux/目录的简单列表输出：

```
# ls -lZ /etc/selinux-rw-r--r-- root root system_u:object_r:selinux_config_t
configdrwxr-xr-x root root system_u:object_r:selinux_config_t strictdrwxr-xr-x
root root system_u:object_r:selinux_config_t targeted
```

正如你所看到的，在我们的系统上安装了两个策略目录：strict 和 targeted。注意目录和策略子目录都用 selinux_config_t 类型进行标记的，这些传统的应用给二进制策略和相关文件的类型，你可以使用 apol 分析适合于这个类型的规则，思考一下什么程序和实用工具可以修改策略文件。

FC 5 中的策略目录

FC 5 中策略子目录的布局稍微做了一下改动，它引入了可载入策略模块结构，主要的改动是引入了管理策略文件的库和工具，这以改动使得安装和移除可载入模块变得更加简单，通常，在 FC 5 中不需要直接编辑策略子目录下的文件。

semodule 和 semanage 命令管理策略的许多方面，semodule 命令管理可载入策略模块的安装、更新和移除，它对可载入策略包起作用，它包括一个可载入策略模块和文件上下文消息，semanage 工具管理添加、修改和移除用户、角色、文件上下文、多层安全(MLS)/多范畴安全(MCS)转换、端口标记和接口标记，关于这些工具的更多信息在它们的帮助手册中。

每个策略子目录包括的文件和文件如何标记必须遵守一个规范，这个规范被许多系统实用程序使用，帮助管理策略，通常，任何设计优良的策略源树都将正确安装策略文件，下面是 strict 策略目录的列表输出，它就是一个典型：

```
# ls -lZ /etc/selinux/strict-rw----- root root
system_u:object_r:selinux_config_t booleans-rw----- root root
root:object_r:selinux_config_t booleans.localdrwxr-xr-x root root
system_u:object_r:default_context_t contextsdrwxr-xr-x root root
system_u:object_r:policy_config_t policydrwx----- root root
system_u:object_r:policy_src_t srcdrwxr-xr-x root root
system_u:object_r:selinux_config_t users
```

一个正在运行的系统不需要 src/目录，它包括了安装的策略源树，要么是示例策略，要么是引用策略源树，实际上单片二进制策略文件存储在.policy/目录中的.policy.[ver]文件中，这里的[ver]就是策略二进制文件的版本号，如.policy.19，这就是系统启动时载入内核的文件。

下面的章节我们将会讨论剩下的文件和目录。

13.1.2.1. 安装的布尔变量文件

第9章“条件策略”讨论了在 SELinux 系统中如何管理布尔变量的，SELinux 策略为布尔变量定义默认的值，booleans 文件让各发行版能够修改默认值并固定下来，booleans 中的值覆盖策略默认值，booleans.local 文件提供了额外的固定值，它又会覆盖策略定义的默认值和发行版的固定值，第9章详细介绍了如何设置和控制布尔变量值，查看 man 8 booleans 也可以，这个帮助页面简要描述了在 FC 和 RHEL 系统上如何使用布尔变量。

在 FC 5 中，booleans 文件已经被取消了，但 booleans.local 文件仍然保留，发行版默认值现在由策略自身进行管理，Red Hat 在策略源中设置它们的默认值。

注意：

在 RHEL 4 中，booleans.local 文件被取消了，能够覆盖策略默认值(不是修改策略本身)的是策略目录下的 booleans 文件，使用单个文件的问题是 Red Hat 使用它作为发行版的默认值，rpm 软件包可能会覆盖它，这样就会造成对本地改变的破坏，在 FC 4 中，booleans.local 文件允许包管理器不影响本地改变。

在 Fedora Core 5 中，booleans 文件被取消，但 booleans.local 文件仍然保留，发行版默认值由策略自身进行管理，Red Hat 在策略源中设置它们的默认值。

system-config-securitylevel 工具提供了一个图形接口改变本地固定值(即 booleans.local 文件)，在这个工具的修改 SELinux 策略列表框中的项目对应定义的策略布尔变量，布尔值也可以使用 setsebool 命令行工具修改，使用 setatus 和 getsebool 命令进行查看。

)系统的，域 RHEL 4 相比较可能稍微有所不同，而且 FC 5 中肯定又有很多增强了，因此我们会标明不同的地方。

13.1.2.2 应用程序和文件安全上下文

在策略安全目录下的 contexts/子目录包括了许多帮助系统服务和实用程序管理文件安全上下文标记的大量文件，它们也包括登陆进程的默认安全上下文，通常，这些文件只应该由策略开发人员修改，但管理员可能偶尔也需要修改一下，下面列出这些文件的主要用途：

contexts/customizable_types: 包括一列当使用 restorecon 或 setfiles 实用程序修复文件标记问题时不会被重新标记的类型，这个特性在帮助保护某些文件标记时特别有用，使用 SELinux 应用程序接口(API)检查上下文是否自定义的是 is_context_customizable(3)。

contexts/default_contexts: 在初始登陆过程中，用户可能登陆会话可能不止得到一个角色/类型组授权，如管理员就可以以特权用户和非特权用户登陆，这个文件提供了一个方法决定初始登陆使用哪个登陆进程(login, ssh 等)作为默认的角色/类型组。

这个文件中的每一行都包括一对角色/类型组，代表登陆进程的安全上下文，后面跟着的就是用户初始登陆进程的默认安全上下文，例如，下面这两行代码：

```
system_r:local_login_t staff_r:staff_t user_r:user_t
sysadm_r:sysadm_t system_r:sshd_t user_r:user_t sysadm_r:sysadm_t
```

第一行代表本地登陆进程(通过它的类型 login_t 登陆)，第二个是 ssh 登陆，通过它的类型 sshd_t。登陆进程由每行的第一个角色/类型组决定的，例如，假设这个文件是登陆进程以角色 system_r，类型为 local_login_t 运行的，同一行后面的角色/类型组将作为用户登陆的默认安全上下文。

默认安全上下文的第一个经过授权的角色/类型组作为默认的安全上下文，这个文件不对用户授予角色或类型，因此，在我们的 default_contexts 文件的本地登陆案例中，如果

管理员在本地登陆(管理员通常是经过 `staff_r:staff_t` 和 `sysadm_r:sysadm_t` 授权的用户),虽然通过了 `sysadm_r:sysadm_t` 授权,但它们的默认安全上下文将是 `staff_r:staff_t`,管理员以后可以修改它们的安全上下文,如使用 `newrole` 命令,因为它们都是通过授权了的,但默认是“staff”权限集,注意,对于 `ssh` 登陆,默认是“sysadm”权限集。

注意,如果有 `contexts/users/[USER]` 文件的话,这些默认值可能会被一个特殊用户覆盖。

`contexts/users/[USER]`: 这个文件和 `default_contexts` 文件的格式一样,除了 `default_contexts` 是为一个特殊用户设计的外,如果某个用户的文件已经存在,默认的角色/类型组由该文件的第一个决定。

`contexts/failsafe_context`: 如果登陆进程不能决定用户的默认安全上下文,用户将不能登陆进系统,这和 `default_contexts` 文件被破坏会改变非常类似,这个文件提供了一个合理的安全上下文故障自动转移机制,至少允许管理员登陆,它提供最后的登陆进程失败前使用的默认安全上下文,通常是:

`sysadm_r:sysadm_t`

这样的话,至少管理员可以登陆。

`contexts/default_type`: 这个文件包括一系列实用程序(如 `newrole`)使用的角色/类型组,如果我们使用 `newrole` 修改我们的角色,但不指定类型,实用程序将会咨询这个文件以确定默认的类型和角色,如果我们使用命令 `newrole -r sysadm_r`,而且这个文件有一行的内容为 `sysadm_r:sysadm_t`,这个命令将会尝试使用 `sysadm_t` 作为我们的默认用户域类型。

`contexts/files/file_contexts`: 这个文件包括与文件有关的安全上下文标记信息,作为策略构建过程的一部分,为与文件有关的客体使用初始化安全上下文,它安装在这里帮助实用程序修复标记问题。

`contexts/files/file_contexts.home_dirs`: 这个文件是使用 `/usr/sbin/genhomedircon` 脚本自动生成的,它的格式与 `file_contexts` 一致,但它只用于标记用户 `home` 目录。

`contexts/files/homedir_template`: 这个文件包括一个 `/usr/sbin/genhomedircon` 脚本使用它产生标记块的模板。

`contexts/files/media`: 这个文件包括挂载在 `/media/` 目录下的存储设备的安全上下文,通过 `libselinux matchmediacon(3)` API 使用它。

`contexts/initrc_context`: 这个文件包括一个角色/类型组,用于 `run_init` 的安全上下文,即管理员以 `init` 相同的方式启动系统服务,这样它就可以执行由 `init` 初始化的存储在 `/etc/rc.d/` 目录下的脚本,这个角色/类型组通常和 `init` 用来启动系统服务的角色/类型组一致。

`contexts/removable_context`: 这个文件包括可移动媒体设备的默认安全上下文,这个安全上下文用于 `media` 上下文文件没有标记的设备。

13.1.2.3. SELinux 用户定义

`user/` 目录下有两个文件被添加进来支持更好的用户管理,以便不用改变策略本身,这两个文件格式相同,它们列出策略 `user` 语句。

`users/system.users`: 这个文件让发行商可以修改与明确定义在策略源中的用户关联的角色, 包管理器将会覆盖这个文件, 因此它不能做本地修改, 我们应该使用 `local.user` 文件定义本地用户。

`users/local.users`: 这个文件的功能和 `system.users` 一样, 除了它不能由发行商修改外, 因此, 我们可以在这个文件中定义本地用户。

`load_policy` 实用工具在将策略载入内核前读取这些文件并改变二进制策略, 改变仅针对内存中的策略版本, 磁盘上的二进制策略不会改变, 通常, 对于任何一个文件, 如果用户已经在策略文件中存在, 即硬编码进原始策略源中, 关联的角色会被改变, 否则, 在它载入内核前用户就会被添加进策略。

13.1.2.4. SELinux 文件系统

SELinux 伪文件系统在 SELinux 内核空间 Linux 安全模块 (LSM) 和用户空间程序之间提供了主要的控制接口, 参考第 3 章中的图 3-2。这个文件系统通常挂载在 `/selinux/`, 许多 SELinux 实用程序和 API (有 `libselinux` 库提供) 使用 SELinux 文件系统访问 LSM 模块, 在本节中, 我们分析其中一些管理员可能感兴趣的文件, 这个文件系统中存在的大部分文件支持 `libselinux` 中的 API, 这里暂时不做讨论, 使用这些文件的推荐方法是通过更稳定的 `libselinux` API 和对应的工具。

`booleans/`: 这个目录包括了策略中定义的每个布尔变量文件, 如果文件被读取, 就会返回布尔变量的当前值和待定值, 待定值就是布尔变量将要改变成的值, 参考 `commit_pending_booleans` 命令, 文件名和布尔变量的名字一致。

`commit_pending_bools`: 这个文件向内核空间安全服务器发送信号, 告知新的策略布尔变量值已经准备就绪, 这个特性允许多个策略布尔变量值以原子方式进行改变。

`disable`: 这个文件是 `init` 在初始化过程中用来禁用 SELinux 的接口, 当初始 SELinux 策略载入或 SELinux 被禁用时, 这个接口不再有效, 因此, 修改成禁用状态总需要重启, 通常, 启用/禁用 SELinux 的唯一方法是修改 `/etc/selinux/config`, 然后重启系统, 只有 `init` 能够直接通过这个文件使用该接口。

`enforce`: 这个文件是用来开启或禁用 `enforce` 模式的接口, `init` 在启动过程中用它设置模式为 `enforcing`, 我们也可以直接使用这个接口修改模式, 1 表示 `enforcing` 模式, 0 表示 `permissive` 模式, 模式的改变是立即生效的, `setenforce` 命令不能完全做到这一点。

也可以编写一个策略禁止任何域许可从 `enforcing` 模式固定成 `permissive` 模式。

`load`: 这个文件是 `load_policy` 程序用来载入新的二进制策略的。

`mls`: 这个文件是内核用来代表 MLS 是否激活。

`policyvers`: 这个文件返回内核支持的策略的最大版本号。

13.2. SELinux 对系统管理的影响

和其它 Linux 系统一样, 管理员也需要理解大量的管理 SELinux 系统的功能, 管理 SELinux 系统的大部分内容都与管理非 SELinux 系统一样, 但 SELinux 引入了几个新的管理行为, 在这一节中, 我们讨论 SELinux 管理新手在管理 SELinux 系统时经常会遇到的几个问题。

13.2.1. 管理用户

在 SELinux 系统上添加、修改和删除用户将会是一个挑战, 如果操作不当, 虽然看上去用户是创建了, 但用户却无法登陆, 通用 SELinux 用户 `user_u` 解决了许多用户管理问题,

例如，当一个用户被添加到 FC 4 系统中时，它就会被自动映射到 user_u，因为新用户策略中还没有定义，关于 user_u 的更多信息请参考第 6 章中的内容。

另一个与用户有关的挑战是如何标记用户 home 目录下的文件，存在几个问题，一个挑战是如何为不同用户域类型统一标记，如 sysadm_t，user_t。另一个挑战是当用户被添加到现有系统时，如何为用户确定正确的标记，如果用户的 home 目录和它下面的文件在初始化时没有正确标记，用户将会遇到各种各样的问题，如不能登陆系统或不能向 home 目录中写入文件，幸运的是，经过这几年的不断改进，我们可以使用 ./contexts/files/file_contexts.home_dirs 这个文件来管理 SELinux 用户了。

13.2.1.1 添加普通的非特权用户

仍然是使用 useradd 命令来添加用户，下面以在 FC 4 系统中添加 jimmy 用户的过程为例：

```
# useradd jimmy# ssh jimmy@localhostjimmy@localhost password:$ id
uid=502(jimmy) gid=502(jimmy) groups=502(jimmy) context=user_u:user_r:user_t
```

注意安全上下文中的用户标识符是通用用户 user_u，那是因为我们没有将 jimmy 添加为特定的 SELinux 用户，因此 FC 4 就使用默认丢的 user_u。

对于那些不需要定义大量用户角色系统而言使用通用用户就足够了，目前常见的策略都只有一组角色(系统,管理员和用户)，所有新用户都指派为用户角色(user_t 域类型,user_r 角色)，这是通过通用用户 user_u 实现的，和 jimmy 一样，为了向 SELinux 系统添加一个普通非特权用户，除了使用 useradd 命令外，我们不需要再做额外的事情。

FC 5 中的用户管理

在 FC 5 中，通过引入 semanage 工具使得用户管理变得更加简单了，semanage 工具可以管理 SELinux 用户、它们的角色授权和普通 Linux 用户之间的映射，关于 SELinux 用户和普通 linux 用户之间的映射请参考第 6 章中的有关内容，如：

在这个例子中，我们添加了一个新用户 staff_u，它被授予 sysadm_r 和 user_r 角色，列出所有 SELinux 用户，将用户 joe 映射到 SELinux 用户 staff_u，然后列出了所有用户映射情况。

```
# semanage user -a -R "sysadm_r user_r" staff_u
# semanage user -l=
```

SELinux User	MLS/ MCS Level	MLS/ MCS Range	SELinux Roles
root	s0	SystemLow-SystemHigh	sysadm_r user_r
system_r			
staff_u	s0	s0	sysadm_r user_r
system_u	s0	SystemLow-SystemHigh	system_r
user_u	s0	SystemLow-SystemHigh	sysadm_r user_r
system_r			

```
# semanage login -a -s staff_u joe
# semanage login -l
```

Login Name	SELinux User	MLS/MCS Range
__default__	user_u	s0
joe	staff_u	s0

```
root                                root
SystemLow-SystemHigh
```

注意，因为 FC 5 默认使用了非强制的 MLS 特性，主要是为了实现 MCS 策略。

13.2.1.2. 添加特权用户

定义在 SELinux 策略中的 root 用户通常对应具有特权的 root 用户账号，策略授予这个用户 sysadm_r 角色和 sysadm_t 用户域，这样的话它就有足够的特权管理系统，虽然 SELinux 中的 root 用户没有标准 Linux 中 root 用户那么大的权利，但它有权运行所有需要特权的程序，经授权的用户使用 root 用户账号（和它关联的特权用户域类型）通常可以获得一个不同的非特权用户域类型（staff_t），这个用户域类型和普通用户域类型 user_t 基本一样，但 staff_t 可以转换成特权用户域类型 sysadm_t，而 user_t 则不行。

添加特权用户时仍然使用 useradd 命令，但我们还需要编辑前面已经讨论过的活动策略中的 local.users 文件，在这个文件中定义了策略使用到的用户，例如，如果我们想创建一个具有特权的用户“admin”：

```
# useradd admin                                # 创建普通用户
# vi /etc/selinux/strict/users/local.users      # 将 admin 添加到特权用户
# load_policy /etc/selinux/strict/policy/policy.19 #重新载入策略
# genhomedircon                                # 修复 home 目录模板
# restorecon -R /home/admin                    #修复 admin 的 home 目录
```

使用 useradd 命令创建账号和 home 目录，但需要告诉 SELinux 这个用户不是普通用户，因此编辑 local.users 文件，添加下面这样一行内容：

```
user admin roles { staff_r sysadm_r };
```

这一行让 SELinux 策略知道用户和为该用户授予的角色，为了让这个改变生效，需要运行 load_policy 重新将策略载入内核，至此，用户就同时定义在系统和 SELinux 策略中了，当用户的 home 目录的标记却没有改变，因此需要运行 genhomedircon 工具为新用户更新 home 目录文件上下文文件（./contexts/files/file_contexts.home_dirs），然后使用 restorecon 程序基于当前的角色更新用户的 home 目录。

至此，用户账号已经创建完毕，并同时授予了非特权管理角色（staff_r）和特权管理角色（sysadm_r），但用户登陆时通过使用的是 staff_r 角色，因此管理员登陆后默认也就一非特权用户，下面以我们刚刚创建的管理员用户登陆为例进行解释：

```
# ssh admin@localhost
admin@localhost's password:
$ id
uid=506(admin) gid=506(admin) groups=506(admin)
context=admin:staff_r:staff_t
```

注意我们的角色和用户域类型分别是 **staff_r** 和 **staff_t**，它们都是非特权的用户角色和类型，通过 **ssh** 登陆方式也使用这个角色和类型作为默认的角色/类型，域类型为 **staff_t** 的进程相对其它普通用户而言本上没有啥特权，只不过它可以转换成 **sysadm_r** 角色，例如，我们在执行管理任务时：

```
$ su
Password:
Your default context is root:sysadm_r:sysadm_t.
Do you want to choose a different one? [n]
# id
uid=0(root) gid=0(root)
groups=0(root),1(bin),2(daemon),3(sys),4(adm),6(disk),10(wheel)
context=root:sysadm_r:sysadm_t
```

su 程序有它的默认安全上下文，以 **staff** 角色/类型运行的用户将会转换成 **sysadm** 角色/类型，注意 **su** 程序也会提示你是否要提供一个非默认定义的安全上下文，如果策略允许，你还可以在这里尝试其它定义的默认角色和类型。

注意：

在 FC 5 中，**su** 不会改变安全上下文，相反，用户必须使用 **newrole** 命令，例如，**newrole -r sysadm_r -t sysadm_t** 命令就和上面举的在 FC 4 和 RHEL 4 中的例子一样，修改了安全上下文。

记住，不同登陆程序的默认安全上下文都只能定义一个。

13.2.1.3. 修改用户角色

修改现有用户的角色和添加一个新的管理角色类似，只是这里不会用到 **useradd** 命令了，因为用户已经创建好了，例如，假设我们要将 **admin** 改回普通用户，我只需要做：

```
# vi /etc/selinux/strict/users/local.users
# load_policy /etc/selinux/strict/policy/policy.19
# genhomedircon
# restorecon -R /home/admin
# ssh admin@localhost
admin@localhost's password:
id
uid=506(admin) gid=506(admin) groups=506(admin) context=user_u:user_r:user_t
```

为了移除管理特权，首先从 **local.users** 文件中移除该用户，然后重新载入策略，最后修复用户的 **home** 目录安全上下文，正如你所看到的，现在我们以该用户身份登陆，我们的角色/类型就是“**user**”，我们的用户就是通用用户“**user_u**”。

13.2.2. 理解审核消息

在第 5 章“类型强制”中，我们讨论了控制由 SELinux 产生的审核消息的策略规则，这里我们讨论这些消息的格式，以及如何在生产系统上分析和管理的审核消息。

在使用内核审核框架的系统上(包括 FC 4 和 FC 5)，SELinux 审核消息同时存储在系统日志文件(**syslog**)和审核守护进程日志文件中，默认情况下，审核守护进程日志存储在

/var/log/audit/audit.log 文件中，系统日志存储在/var/log/messages 文件中。审核守护进程日志包括所有的审核消息，包括访问向量缓存(AVC)消息，AVC 消息是由 SELinux 因访问拒绝和 auditallow 规则产生的审核消息，系统日志包括的是更普通的审核消息。

注意：

RHEL 4 的原始版本和第一个升级版没有使用审核框架，这意味着所有审核消息都存储在系统日志中，通常是/var/log/messages，从 RHEL 4 Update 2 开始使用审核框架，有部分 SELinux 消息仍然存储在系统日志中，因为它们是内核消息而不是审核消息(策略载入消息)，在审核守护进程启动之前，所有 SELinux 审核消息都存储在系统日志中，在 FC 5 中，审核守护进程是可选的，这就意味着根据配置不同，审核消息要么存储在系统日志中，要么存储在审核日志中。

13.2.2.1. 常见的 SELinux 审核消息

在系统初始化、策略载入和布尔值改变时 SELinux 产生审核消息，策略本身不能控制这些消息的产生，它们是硬编码进 SELinux 的，所有常见的 SELinux 审核消息都存储在系统审核日志中。

系统初始化时，SELinux 产生关于 SELinux LSM 模块配置的有关审核消息，下面是系统启动后 SELinux 产生的第一条审核消息：

LSM 框架初始化产生第一行消息，后面的初始化过程产生第二行消息，从第三行可以看出，这个系统启动到 permissive 模式，第 4 行和第 5 行显示了 LSM 模块的功能，注册为第二个 SELinux 模块。

系统初始化后，策略第一次载入，产生类似下面这样的审核消息：

最前面的两行显示每次策略载入时产生的审核消息，正如你所看到的，这个消息显示的是关于策略载入的统计信息，这个示例策略有 3 个用户，6 个角色，1341 个类型，62 个布尔变量，55 个人客体类别以及 345260 条规则。在第 5 章中已经谈到，内核策略中的规则被扩展，这些审核消息中显示的大量规则都是扩展规则格式，它的数量比策略源文件中的规则数量要大出很多。

```
1 Jul 22 11:44:26 milton kernel: security: 3 users, 6 roles, 1341 types, 62 bools
2 Jul 22 11:44:26 milton kernel: security: 55 classes, 345260 rules
3 Jul 22 11:44:26 milton kernel: SELinux: Completing initialization.
4 Jul 22 11:44:26 milton kernel: SELinux: Setting up existing superblocks.
5 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev dm-0, type ext3), uses
xattr
6 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev tmpfs, type tmpfs),
uses
transition SIDs
7 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev selinuxfs, type
selinuxfs),
uses genfs_contexts
8 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev mqueue, type mqueue),
not
configured for labeling
```

```
9 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev hugetlbfs, type
hugetlbfs),
not configured for labeling
10 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev devpts, type devpts),
uses
transition SIDs
11 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev eventpollfs, type
eventpollfs)
, uses genfs_contexts
12 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev tmpfs, type tmpfs),
uses
transition SIDs
13 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev futexfs, type futexfs),
uses
genfs_contexts
14 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev pipefs, type pipefs),
uses
task SIDs
15 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev sockfs, type sockfs),
uses
task SIDs
16 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev proc, type proc),
uses
genfs_contexts
17 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev bdev, type bdev),
uses
genfs_contexts
18 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev rootfs, type rootfs),
uses
genfs_contexts
19 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev sysfs, type sysfs),
uses
genfs_contexts
20 Jul 22 11:44:26 milton kernel: SELinux: initialized (dev usbfs, type usbfs),
uses
genfs_context
```

这个例子也显示了第一次策略载入时产生的审核消息，第 3 和 4 行显示了完整的 SELinux 初始化，包括 SELinux 支持的文件系统客体初始化，文件系统的挂载处于策略载入之前，第 5 到 20 行显示了每个文件系统的初始化和关联的标记行为。

当布尔变量的值发生变化时也会产生 SELinux 审核消息，如：

布尔变量提交消息显示当前策略中所有布尔变量的最新状态，正如前面的例子所显示的，这可能产生很长的审核消息，会被审核框架拆分成两个独立的消息。

```
Dec 2 14:07:41 book kernel: security: committed booleans { allow_write_xshm:1,
mozilla_read_content:0, mail_read_content:0, cdrecord_read_content:0,
allow_ptrace:0, read_untrusted_content:0, write_untrusted_content:0,
user_dmesg:0, use_nfs_home_dirs:0, allow_execmem:0, allow_execstack:0,
allow_execmod:0, use_samba_home_dirs:0, user_tcp_server:0, allow_yplib:0,
allow_kerberos:1, user_rw_usb:1, user_net_control:0, user_direct_mouse:0,
user_rw_noexattrfile:1, read_default_t:1, staff_read_sysadm_file:1,
allow_httpd_user_script_anon_write:0, allow_httpd_staff_script_anon_write:0,
user_ttyfile_stat:0, httpd_unified:0, httpd_builtin_scripting:1,
httpd_enable_cgi:1, httpd_enable_homedirs:1, httpd_ssi_exec:1,
httpd_tty_comm:0,
httpd_can_network_connect:0, allow_httpd_sys_script_anon_write:0,
allow_httpd_anon_write:0, httpd_suexec_disable_trans:0, comsat_disable_trans:0,
cron_can_relabel:0, cupsd_lpd_disable_trans:0, cvs_disable_trans:0,
dbskkd_disable_trans:0, disable_evolution_trans:0, ftpd_is_daemon:1,
ftpd_home_dir
Dec 2 14:07:41 book kernel: :1, allow_ftpd_anon_write:0, disable_games_trans:0,
inetd_child_disable_trans:0, allow_java_execstack:0, ktalkd_disable_trans:0,
disable_mozilla_trans:0, allow_mplayer_execstack:0, allow_user_mysql_connect:0,
named_write_master_zones:0, secure_mode:0, user_ping:0,
allow_user_postgresql_connect:0, pppd_for_user:0, pppd_can_insmode:0,
rlogind_disable_trans:0, nfs_export_all_rw:0, nfs_export_all_ro:0,
allow_gssd_read_tmp:1, rsync_disable_trans:0, allow_rsync_anon_write:0,
allow_smbd_anon_write:0, samba_enable_home_dirs:0,
allow_saslauthd_read_shadow:0, spamassassin_can_network:0, squid_connect_any:0,
ssh_sysadm_login:1, allow_ssh_keysign:0, run_ssh_inetd:0,
stunnel_disable_trans:0, stunnel_is_daemon:0, swat_disable_trans:0,
telnetd_disable_trans:0, disable_thunderbird_trans:0, uucpd_disable_trans:0,
xdm_sysadm_login:0 }
```

```
1 Jul 22 11:44:25 milton kernel: Security Framework v1.0.0 initialized
2 Jul 22 11:44:25 milton kernel: SELinux: Initializing.
3 Jul 22 11:44:25 milton kernel: SELinux: Starting in permissive mode
4 Jul 22 11:44:25 milton kernel: selinux_register_security: Registering
secondary module capability
5 Jul 22 11:44:25 milton kernel: Capability LSM initialized as secondar
```

13.2.2.2. AVC 消息

AVC 消息是访问被拒绝时产生的消息，不是 dontaudit 消息，也不是匹配 auditallow 规则的消息，这些消息包括可以用来系统监视、管理和策略开发的宝贵信息，第 14 章“编写策略模块”解释了在策略开发时如何使用这些消息。

AVC 消息存储审核守护进程日志文件中，下面是一个 AVC 消息的典型例子：

```
type=AVC msg=audit(1135098961.471:1770): avc:
```

```
denied { read } for pid=19850
comm="cat" name="sysadm_tmp_file" dev=dm-0 ino=67482
scontext=kmacmillan:staff_r:staff_t tcontext=
kmacmillan:object_r:sysadm_tmp_t
tclass=file
```

这个 AVC 消息显示了一个安全上下文为 `kmacmillan:staff_r:staff_t` 进程在访问安全上下文为 `object_r:sysadm_tmp_t` 的文件时被拒绝，这个进程对 `/tmp/sysadm_tmp_file` 文件执行 `cat` 操作。

正如你所看到的，AVC 消息中的所有字段几乎都以“名称=值”的形式出现，如 `pid=19850`，字段的名称是 `pid`，值是 `19850`。

为了理解 AVC 消息，我们分析一下每个字段，所有 AVC 消息都有下面 6 个字段：

type: 审核守护进程产生的消息可以有多种类型，消息的类型是由“type=”的前缀和消息类型共同标识的，这个例子中的前缀是 `AVC`，标识出这条消息是一条 AVC 消息，其它消息类型包括 `USER_AUTH`，`LOGIN`，`SYSCALL` 和 `PATH`。

msg: 审核框架在所有审核消息的开始位置加上一个消息头，包括一个时间戳和由分号隔开的序列号，在这个消息中的时间戳是 `1135098961.471`，它是从基准时间以来的以纳秒为单位的时长，在这个例子中序列号是 `1770`，它用来标识由相同事件产生的多个相关的审核消息，例如，一个事件可以产生两个系统调用和 AVC 审核消息，这些消息的序列号是相同的。

avc: 这个字段再不是名称/值的形式了，它标识出审核消息是来自允许访问还是拒绝访问，以及允许或拒绝的许可，这里可以有一个或多个许可，都来自一个客体类别，关键词 `denied` 表示这个消息来自访问被拒绝，允许访问是用 `granted` 表示的。

scontext: 源或主体的安全上下文。

tcontext: 目标或客体的安全上下文。

tclass: 目标或客体的客体类别，允许或拒绝的许可来自为这个客体类别定义的访问向量。

AVC 消息中余下的部分提供了关于访问被允许或拒绝的详细信息，这些细节通常是特定的客体类别，例如，与文件有关的客体类别的审核消息通常包括客体的 `inode` 号，与网络有关的客体类别的审核消息通常包括 `IP` 地址和端口号，前面的例子有下面四个字段：

pid: 尝试访问的进程的标识符，它在一个应用程序产生多个调用时特别有用，如 `apache`。

comm: 与进程关联的可执行文件名，这个字段只包括文件名，没有文件路径信息。

dev 和 ino: 与目标关联的与文件有关的客体的设备 (`dev`) 和 `inode` 号 (`ino`)，如果在审核消息中没有完整的路径，可以使用它们来标识客体。

name: 与文件有关的客体的名称，这个字段只包括文件名，不包含文件路径。

注意：

在新的 Linux 审核框架下，每个 AVC 消息后面都跟有一个 `SYSCALL` 消息，它们的审核事件 ID 都一样，`SYSCALL` 消息有正确完整的 `exe` 和 `path` 字段（分别对应 `comm` 和 `name`）。

允许访问的 AVC 消息也类似，如：

```
type=AVC msg=audit(1135098723.344:1742): avc: granted { load_policy } for
pid=19618 comm="load_policy" scontext=root:sysadm_r:load_policy_t
tcontext=system_u:object_r:security_t tclass=security
```

上面这个 AVC 显示成功载入了策略，产生这个 AVC 消息的 auditallow 规则通常包括在策略中，因为载入的重要性。

13.2.2.3. 使用 Seaudit 查看审核消息

Seaudit 和 apol 一起都是包括在 setools 软件包中的，它用来解析和显示 SELinux 审核消息，图 13-2 显示了一个常见的 Seaudit 会话。

Hostname	Message	Date	Source Type	Target Type	Object Class	Permission	Executable	Other
	Denied	Oct 20 17:11:51	sysadm_t	shadow_t	file	read		dev=dm-0 timestamp=1129842711.180 serial=274668
	Denied	Oct 20 17:11:55	sysadm_t	shadow_t	file	read		dev=dm-0 timestamp=1129842715.034 serial=285086
	Denied	Oct 20 17:12:06	local_login_t	faillog_t	file	lock		dev=dm-0 timestamp=1129842726.617 serial=288558
	Denied	Oct 20 17:12:29	sysadm_t	shadow_t	file	read		dev=dm-0 timestamp=1129842749.882 serial=316364
	Denied	Oct 20 17:12:29	sysadm_t	shadow_t	file	read		dev=dm-0 timestamp=1129842749.907 serial=318774
	Denied	Oct 20 17:12:30	sysadm_t	shadow_t	file	read		dev=dm-0 timestamp=1129842750.003 serial=324293
	Denied	Oct 20 17:12:30	sysadm_t	shadow_t	file	read		dev=dm-0 timestamp=1129842750.150 serial=334640
	Denied	Oct 20 17:22:34	automount_t	root_t	dir	write		dev=dm-0 timestamp=1129843354.968 serial=8954
	Denied	Oct 20 17:22:36	udev_t	net_conf_t	file	read		dev=dm-0 timestamp=1129843356.665 serial=28942
	Denied	Oct 20 17:22:36	udev_t	net_conf_t	file	read		dev=dm-0 timestamp=1129843356.666 serial=28947
	Denied	Oct 20 17:22:36	udev_t	net_conf_t	file	read		dev=dm-0 timestamp=1129843356.738 serial=29819
	Denied	Oct 20 17:22:36	udev_t	net_conf_t	file	read		dev=dm-0 timestamp=1129843356.740 serial=29824
	Granted	Oct 20 17:26:55	load_policy_t	security_t	security	load_policy		timestamp=1129843615.986 serial=989973
	Granted	Oct 20 17:30:05	load_policy_t	security_t	security	load_policy		timestamp=1129843805.743 serial=1983274
	Denied	Oct 20 17:35:48	sshd_t	newrole_t	fd	use		dev=devpts timestamp=1129844148.995 serial=4005968
	Denied	Oct 20 17:35:48	sshd_t	newrole_t	fd	use		dev=devpts timestamp=1129844148.995 serial=4005968
	Denied	Oct 20 17:35:52	sshd_t	newrole_t	fd	use		dev=devpts timestamp=1129844152.370 serial=4006675
	Denied	Oct 20 17:35:52	sshd_t	newrole_t	fd	use		dev=devpts timestamp=1129844152.370 serial=4006675

图 13-2. 一个常见的 Seaudit 会话

这个工具解析日志文件，然后用一个列表显示出所有消息，每条消息有 16 个自定义字段可以显示，“Modify view”按钮允许你创建自定义过滤器，以只显示感兴趣的数据，你可以将过滤结果保存为一个报告。

13.2.3. 问题修复：与文件有关的客体标记

在正常的系统中，与文件有关的客体应该不需要标记或重新标记，在安装期间所有的操作系统文件应该都有一个正确的安全上下文，与标记有关的策略规则确保新创建的文件也具有正确的安全上下文，然而，由于策略的开发、系统设置和系统管理，文件可能需要重新标记。

警告！

重新标记客体有安全风险，包括潜在的竞争条件，不一致的访问控制应用到客体，恶意的硬连接，缺乏完整的撤销支持，出于最佳安全考虑，在生产系统上应该完全避免重新标记，当不可避免要重新标记时，系统应该处于一个良好的状态(如系统刚刚安全完毕或刚刚校验

了系统的完整性，没有出现泄密)，对于大的标记改变，如一个大的策略改变，最好是将系统从生产线撤下来先。

13.2.3.1. 与文件有关的客体标记命令

重新标记与文件有关的客体主要用到4个命令:chcon(8), restorecon(8), setfiles(8), and fixfiles(8). 这些命令都用于重新标记文件，但它们的用法都不一样，通常，chcon 和 restorecon 用于小的标记修改，fixfiles 和 setfiles 用于大的标记修改。

chcon 命令基于用户的输入为一个或多个文件设置相同的安全上下文，它是最基础的标记命令，它的用法和 chmod(1)有点类似，如：

```
$ mkdir public_html
$ ls -dZ public_html/
drwxrwxr-x  joe joe joe:object_r:user_home_dir_t public_html/
$ chcon -t httpd_user_content_t public_html/
$ ls -dZ public_html/
drwxrwxr-x  joe joe joe:object_r:httpd_user_content_t public_html/
```

restorecon 命令和 chcon 命令类似，但它是基于当前策略默认文件上下文文件设置与文件有关的客体的安全上下文，因此，用户没有指定一个安全上下文，相反，restorecon 使用文件上下文文件的条目匹配文件名，然后应用特定的安全上下文，在某些情况下，它是在还原正确的安全上下文，如：

```
$ mkdir public_html
$ ls -Zd public_html/
drwxrwxr-x  joe joe joe:object_r:user_home_dir_t public_html/
$ /sbin/restorecon public_html/
$ ls -Zd public_html/
drwxrwxr-x  joe joe user_u:object_r:httpd_user_content_t public_html/
```

/home/[^/]*/public_html(/.+)? user_u:object_r:httpd_user_content_t

文件上下文文件的条目指出了在用户 home 目录中的子目录 public_html/应该被标记为: user_u:object_r:httpd_user_content_t

我们也可以使用 restorecon 命令检查与文件有关的客体的标记与文件上下文文件中的条目是否匹配，如：

```
$ mkdir public_html
$ /sbin/restorecon -nv public_html/
/sbin/restorecon reset /home/joe/public_html context
joe:object_r:user_home_dir_t->user_u:object_r:httpd_user_content_t
```

在这个例子中好，我们使用 -n 和 -v 选项阻止 restorecon 真正地执行标记，它只输出标记改变的信息，这两个选项一起让 restorecon 输出在磁盘上的标记和文件上下文文件标记之间的差异。

最后，restorecon 命令可以用来递归标记大量的文件，-R 选项告诉 restorecon 命令下钻目录，重新标记所有的子目录和文件，如

```

$ mkdir public_html
$ scp -r gotham:public_html/*.html public_html/.
kmacmillan@gotham's password:
2005d10.html          100%  28KB  28.3KB/s   00:00
2005d11.html          100%  22KB  21.5KB/s   00:00
2005d12.html          100% 8575   8.4KB/s   00:00
2005d7.html           100%  15KB  14.9KB/s   00:00
calendar.html         100% 2839   2.8KB/s   00:00
coding_style.html     100% 1040   1.0KB/s   00:00
$ ls scontext public_html/*
joe:object_r:user_home_dir_t public_html/2005d10.html
joe:object_r:user_home_dir_t public_html/2005d11.html
joe:object_r:user_home_dir_t public_html/2005d12.html
joe:object_r:user_home_dir_t public_html/2005d7.html
joe:object_r:user_home_dir_t public_html/calendar.html
joe:object_r:user_home_dir_t public_html/coding_style.html
$ /sbin/restorecon -R public_html/
$ ls scontext public_html/*
user_u:object_r:httpd_user_content_t public_html/2005d10.html
user_u:object_r:httpd_user_content_t public_html/2005d11.html
user_u:object_r:httpd_user_content_t public_html/2005d12.html
user_u:object_r:httpd_user_content_t public_html/2005d7.html
user_u:object_r:httpd_user_content_t public_html/calendar.html
user_u:object_r:httpd_user_content_t public_html/coding_style.html

```

尽管 restorecon 命令可以递归地重新标记所有文件和目录，但平时却很少使用它标记大量的文件，因为还有 fixfiles 命令，fixfiles 命令实际上是一个 shell 脚本，依请求不同它分别可以调用 restorecon 或 setfiles 命令，和 restorecon 一样，fixfiles 使用文件上下文文件，fixfiles 在所有受支持的扩展属性标记的文件系统上都能够很好地工作，fixfiles 命令有三个模式，运行这个命令时必须指定其中一个：

check: 显示所有与文件上下文文件中安全上下文不匹配的与文件有关的客体。

restore: 重新标记那些与文件上下文文件的安全上下文不匹配的与文件有关的客体的安全上下文。

relabel: 和 restore 一样，但它也可以首先从 /tmp 目录中移除文件。

例如，下面显示了如何重新标记系统中所有与文件有关的客体：

```
# /sbin/fixfiles relabel
```

Files in the /tmp directory may be labeled incorrectly, this command can remove all files in /tmp. If you choose to remove files from /tmp, a reboot will be required after completion.

Do you wish to clean out the /tmp directory [N]? n

最后一个命令 `setfiles`，它需要用户指定选择哪个文件上下文文件和起始目录，此外，在下钻目录时，`setfiles` 不能跨越挂载点，这就意味着在每个文件系统上都得运行一次这个命令，通常都会使用 `fixfiles`，除非有特殊要求时才会使用 `setfiles`。

在这个例子中，我们从另一个系统中拷贝了几个 Web 页面，它们自动接受安全上下文 `joe:object_r:user_home_dir_t`，运行 `restorecon` 命令递归地重新标记所有的目录和文件为 `user_u:object_r:httpd_user_content_t`。

这个例子就其功能而言和前面的 `chcon` 一样，但只是因为文件上下文文件中有下列条目：

在这个例子中，我们修改了新创建的目录的安全上下文，创建时自动分配的安全上下文是 `joe:object_r:user_home_dir_t`，然后修改成 `joe:object_r:httpd_user_content_t`，`-t` 选项指出文件的类型应该改变而安全上下文的剩余部分保持不动。

13.2.3.2. 自动重新标记

除了使用与文件有关的客体标记命令重新标记整个系统外，在系统重启时也会自动重新标记，只需要在根文件系统下创建一个 `/.autorelabel` 文件即可，如：

```
# touch /.autorelabel
```

如果这个文件在启动过程中出现，整个系统都会重新标记，该文件也会被移除，内核将会以 `autorelabel` 参数运行，当一个 SELinux 系统在 SELinux 禁用时启动，会自动创建 `/.autorelabel` 文件。

13.2.4. 管理多个策略

通常，在生产系统上不应该存在多个策略，并在策略间进行切换，但在策略开发时就会经常有多个策略存在。

策略切换的步骤如下：

- 1、在 `/etc/selinux` 下安装策略，如一个策略叫做 `mypol`，那么它就会安装在 `/etc/selinux/mypol/`，真实的二进制策略文件应该是 `/etc/selinux/mypol/policy/policy.[ver]`。

- 2、在 SELinux 配置文件中修改策略名 (`/etc/selinux/config`)，可以使用 `system-config-securitylevel` 命令或文本编辑器来完成。

- 3、通过使用 `/.autorelabel` 文件告诉系统下次启动时自动重新标记整个系统。

- 4、重新启动系统。

在重新启动时，系统会载入新的策略，并重新标记所有与文件有关的客体。

13.3. 小结

`/etc/selinux/config` 控制激活哪个策略(即系统启动时载入哪个策略)，这个文件同时也控制系统启动时 SELinux 的默认模式：`enforcing`(正常)，`permissive` 和 `disabled`。

策略和相关文件安装在 `/etc/selinux/[policyname]/`，例如，FC 4 中 `targeted` 策略默认安装在 `/etc/selinux/targeted/`，除了真实的二进制策略文件外，这个目录还包括了大量系统实用程序用来管理策略内容(如用户)或对象标记的文件，同时还包括了策略源文件。

SELinux 为 SELinux LSM 模块提供了接口，如文件系统通常挂载在 `/selinux/`，这个目录下的大部分文件都支持 `libselinux` 库的 API。

SELinux 通用用户 `user_u` 提供了一个方法在 SELinux 系统中添加用户时不用将它们添加到策略中, `user_u` 定义了正常情况授予非特权用户的许可和角色, 要添加一个管理员用户, 必须手动编辑 `loacl.users` 文件, 然后重新载入策略。

SELinux 默认产生两种消息: 常规消息和 AVC 消息。常规审核消息记录了与系统初始化、策略载入和布尔值变化的事件; AVC 消息记录了访问被拒绝的消息。

通常, 在生产系统上不应该维护文件安全上下文标记, 但如果遇到策略更新或你使用的是一个开发/体验系统, 你可能需要手动修复或恢复客体标记, SELinux 提供了四个命令来帮助完成这个任务: `chcon(8)`, `restorecon(8)`, `setfiles(8)`, and `fixfiles(8)`。

第 14 章. 编写策略模块

这一章将我们前面所学的所有知识进行一次集中应用, 通过编写一个同时使用示例策略和引用策略的策略模块来构成一个学习指南。

14.1. 策略模块编写概述

在这一章中, 我们演练一下创建策略模块的过程, 将我们前面所学进行一个融合, 我们讨论了所有需要的步骤, 包括通过示例策略创建策略模块和通过引用策略创建策略模块, 对过程中的大部分步骤, 我们都加以了解, 并同时举了示例策略和引用策略的例子, 我们认为这种通过示例的方式对于理解这两种策略会起到事半功倍的作用。

这里我们只对这个主题进行一个概述, 要想成为一名有经验的策略编写者, 必须得尝试编写模块。我们提供的提纲向你提供了策略开发的起点, 最佳实践是你亲自通过应用 SELinux 来解决你面临的安全挑战。

在这一章中我们创建的策略模块是为 Fedora Core 4 中 IRC 守护进程准备的, 我们选择它作为例子是因为它是一个简单但有代表性的面向网络的守护进程实例。

以我的经验来看, 编写策略模块涉及三个基本步骤: 准备和计划, 初始化策略模块创建, 测试和分析。在准备和计划阶段, 我们收集关键信息, 创建测试环境, 为策略模块指定安全目标, 在初始化策略创建阶段, 我们将收集到的信息和安全目标结合起来, 创建策略模块的第一个版本, 在测试和分析阶段, 我们从功能和安全方面来决定策略模块的准确性。

在本章剩余的内容中, 我们详细描述这些步骤, 实际上, 策略编写通常是一个反复的过程, 不断地编写、测试和研究, 特别是测试和分析阶段经常会改变策略模块。

14.2. 准备和计划

在编写自己的策略模块之前, 我们需要收集一些关于应用程序的信息, 创建一个测试配置, 指定我们的安全目标, 我们还必须选择我们的目标平台和策略, 在我们的例子中, 我们的目标是一个 FC 4 系统, 为 `strict` 示例策略和 `strict` 引用策略创建策略模块。

14.2.1. 收集应用程序信息

和所有策略模块一样, 我们的 IRC 模块主要是为 IRC 守护进程创建一个域, 编写策略模块时需要很多关于该守护进程是如何设计的信息以及它都包括哪些功能, 通常, 我们越理解目标应用程序, 最终的策略就会越安全, 功能越强, 特别重要的是应用程序架构(如进程和资源的数量及目的)、管理(如配置文件的说明文档)和现有安全信息, 关于应用程序安全的现有信息, 包括硬化的指南都可以提供帮助, 安全指南经常不会提供完整的应用程序安全信息或符合你特定的安全需要。

下面是我们在 FC 4 中收集到的关于 Hybrid IRC 守护进程的信息:

守护进程由一个进程组成, 它在端口 6667 上监听如站 IRC 连接。

IRC 协议(最初是在 RFC 1459 中描述的)正常情况下是在 TCP 顶层实现的, 每个客户端只有一个连接。

默认情况下, 在/etc/ircd/目录下存储了大量的配置文件。

默认情况下, IRC 守护进程在/var/log/ircd/目录下存储了它的私有日志文件。

FC 4 RPM 为 IRC 守护进程在/var/lib/目录下创建了一个 data 目录。

和 FC 中其它守护进程类似, IRC 守护进程在/var/run/目录下创建了文件存储活动进程的 PID。

除了日志、PID、/var/lib/、配置文件和目录外, IRC 守护进程不需要任何其它的文件系统访问权。

14.2.2. 创建测试环境

编写策略模块需要测试和实验, 因此, 我们需要在用于策略开发的系统上安装一个测试服务, 和所有测试一样, 测试环境要尽可能和开发环境一致, 我们在 FC 4 上创建了一个 IRC 守护进程作为测试进程, 当然, 我们还需要在同一网络中搭建一个 IRC 客户端。

我们从一个基本安装的 FC 4 工作站开始, 我们需要添加示例和引用策略源文件和 IRC 守护进程, 附录 A“获取 SELinux 示例策略”介绍了如何获取和安装需要的 strict 示例策略, IRC 守护进程使用 yum 命令安装(需要 su 到 root 账号进行安装)。

```
#yum install ircd-hybrid
```

这个命令会安装 IRC 守护进程, 启动脚本和示例配置文件, 我们现在需要编辑配置文件 /etc/ircd/ircd.conf, 我们从文档中提供的 simple.conf 文件开始 (/usr/share/doc/ircd-hybrid-7.2.0/simple.conf), 对它稍作修改(server info sid 和 operator 选项), 如清单 14-1 所示(修改的选项已经加粗了)。

清单 14-1. 修改后的 IRC 守护进程配置文件(ircd.conf)

```
1      # Hybrid 7 minimal example configuration file
2      #
3      # $Id: simple.conf 33 2005-10-02 20:50:00Z knight $
4      #
5      # This is a basic ircd.conf that will get your server running with
6      # little modification. See the example.conf for more specific
7      # information.
8      #
9      # The serverinfo block sets your server's name. Fields that may
10     # be set are the name, description, vhost, network_name,
11     # network_desc, and hub.
12
13     serverinfo {
14         name = "irc.example.com";
15         sid = "1se";
16         description = "Test IRC Server";
17         hub = no;
18     };
19
```

```
20 # The administrator block sets up the server administrator
21 # information, that is shown when a user issues the /ADMIN
22 # command. All three fields are required.
23
24 administrator {
25     description = "Example, Inc Test IRC Server";
26     name = "John Doe";
27     email = "jdoe@example.com";
28 };
29
30 # Class blocks define the "privileges" that clients and servers
31 # get when they connect. Ping timing, sendQ size, and user
32 # limits are all controlled by classes. See example.conf for
33 # more information
34
35 class {
36     name = "users";
37     ping_time = 90;
38     number_per_ip = 0;
39     max_number = 200;
40     sendq = 100000;
41 };
42
43 class {
44     name = "opers";
45     ping_time = 90;
46     number_per_ip = 0;
47     max_number = 10;
48     sendq = 500000;
49 };
50
51 # Auth blocks define who can connect and what class they
52 # are put into.
53
54 auth {
55     user = "*@*";
56     class = "users";
57 };
58
59 # Operator blocks define who is able to use the OPER command
60 # and become IRC operators. The necessary fields are the
61 # user@host, oper nick name, and the password, encrypted with
62 # the mkpasswd program provided.
63
```

```

64     operator {
65         name = "JohnDoe";
66         user = "\*@\*.example.com";
67         # MD5 encrypted password - "selinux"
68         password = "$1$gv.dyLcq$wr2F.9AqZ/2EKxcsCexKml";
69         encrypted = yes;
70         class = "opers";
71     };
72
73     # Listen blocks define what ports your server will listen to
74     # client and server connections on. ip is an optional field
75     # (Essential for virtual hosted machines.)
76
77     listen {
78         port = 6667;
79     };
80
81     # Quarantine blocks deny certain nicknames from being used.
82
83     quarantine {
84         nick = "dcc-*";
85         reason = "DCC bots are not permitted on this server";
86     };
87
88     quarantine {
89         nick = "LamestBot";
90         reason = "You have to be kidding me!";
91     };
92
93     quarantine {
94         nick = "NickServ";
95         reason = "There are no Nick Services on this Network";
96     };
97
98     # The general block contains most of the configurable options
99     # that were once in config.h. The most important ones are below.
100    # For the rest, please see example.conf. Note that variables not
101    # mentioned here are set to the ircd defaults, which are listed in
102    # src/s_conf.c:set_default_conf.
103
104    general {
105        hide_spoof_ips = yes;
106        # Identd is commonly disabled on modern systems
107        disable_auth = yes;

```

```

108     # Control nick flooding
109     anti_nick_flood = yes;
110     max_nick_time = 20;
111     max_nick_changes = 5;
112
113     # Show extra warnings when servers connections cannot succeed
114     # because of no "N" line (a misconfigured connect block)
115     warn_no_nline = yes;
116 };

```

提示:

理解应用程序所有的文件和目录对于策略开发非常重要, 命令 `rpm -ql ircd-hybrid` 可以列出 IRC 守护进程包括的所有文件和目录。

我们对这个文件只做了三处修改, 第一处是修改服务器的唯一标识符(第 15 行), 管理密码(第 68 行)和禁用 `identd`(第 107 行), 保存为 `/etc/ircd/ircd.conf` 文件后, 我们使用下面的命令启动服务器(目前处于 SELinux permissive 模式):

```

# setenforce 0# /etc/init.d/ircd startStarting ircd: ircd: version
hybrid-7.2.0ircd: pid 9052ircd: running in background mode from /usr/lib/ircd [ OK ]

```

这些命令显示 `ircd` 服务成功启动, 一旦启动完毕, 日志文件 `/var/log/ircd/ircd.log` 就会包括下面的项目(在末尾或接近末尾的地方):

```
[2006/2/3 04.25] Server Ready
```

注意可能还会产生一些 AVC 消息, 因为我们还没有为服务器安装特定的策略, 不过目前我们可以忽略这些 AVC 消息。

14.2.3. 指定安全目标

准备工作的最后一步是指定 IRC 策略模块的安全目标, 如果不理解安全对这个应用程序意味着什么, 那我们就不能在策略模块开发过程中制定关键的安全决定, 在开始思考策略语言的细节之前应该花时间好好思考一下整体安全目标, 在创建好策略模块之后, 我们会重访这些安全目标, 以确定策略模块是否符合我们的目标。

如何正确决定和指定安全目标本身就是一个大主题, 已经超出了本书的范畴, 它需要经验和正确的思考, 下面是为我们的 IRC 守护进程策略模块指定的一些安全目标:

ircd 服务限制: 限制 `ircd` 服务, 最小化访问权, 这样可以阻止利用服务的漏洞攻击整个系统。

系统保护: 保护提供 IRC 服务的系统, 防止攻破 IRC 后的权限提升。

配置文件保护: 保护配置文件, 防止被非管理域(如除了 `sysadm_t` 之外的域)和服务自身修改。

这些安全目标仅仅是个起点, 对于 IRC 或类似的应用程序而言, 还有许多其它的安全目标。

14.3. 创建一个初始化策略模块

在下一步中, 我们在收集到的信息和指定的安全目标基础上创建一个初始化策略模块, 为了创建尽可能安全的策略模块, 在测试开始之前, 我们想创建一个策略, 只授予我们希望对 IRC 守护进程的访问权。

14.3.1. 创建策略模块文件

我们通过为示例策略和引用策略创建所有策略模块文件开始我们的策略模块开发。

14.3.1.1. 示例策略

正如我们在第 11 章中讨论的，示例策略中的策略模块由两个文件组成：策略规则文件(.te)和文件上下文文件(.fc)。因此，我们需要为 IRC 守护进程策略模块创建文件 domains/programs/ircd.te 和 file_contexts/programs/ircd.fc，开始时，这些文件都可以是空的。

注意：

所有路径名都是相对于策略源根目录的相对路径，对于示例策略就是 /etc/selinux/strict/src/policy，对于引用策略就是 /etc/selinux/refpolicy/src/policy，我们只引用了文件名(如 ircd.te 就相当于示例策略中的 /etc/selinux/strict/src/policy/domains/programs/ircd.te 文件)。

14.3.1.2. 引用策略

正如第 12 章所讨论的，引用策略模块是由三个文件组成的：私有策略文件(.te)，外部接口文件(.if)和标记策略文件(.fc)。因为 IRC 守护进程是一个系统服务，我们将它的策略模块文件放在服务层(即 policy/modules/services/ircd.te，policy/modules/services/ircd.if 和 policy/modules/services/ircd.fc)，文件 ircd.if 和 ircd.fc 开始时可以是空的，但 ircd.te 必须有下面的最小声明：

```
# Ircd policy module declarationpolicy_module(ircd, 1.0)
```

14.3.2. 类型声明

下一步是为策略模块声明适当的域和客体类型，记住，访问只能在类型间被允许，因此我们必须标识和声明正确的类型，代表应用程序的架构，从许多方面来看，这是策略模块开发中最重要的一步，如果我们没有正确地标识出需要的类型，特别是域类型，策略剩下的部分也不会是正确的。

策略模块通常声明下面的类型：

域：应用程序进程的一个或多个域类型。

入口点：每个域至少有一个入口点可执行文件类型。

应用程序资源：一个或多个应用程序控制的资源类型(如临时文件、配置文件、日志文件和套接字文件等)。

我们为我们的 IRC 守护进程策略模块声明的类型域目标应用程序的高级架构精密匹配，我们的 IRC 类型如下：

ircd_t：为 IRC 守护进程创建的域类型。

ircd_exec_t：适合于 IRC 守护进程可执行文件的入口点类型。

ircd_var_run_t：适合于存储在/var/run 目录下的 PTD 文件的文件类型。

ircd_conf_t：适合于 IRC 守护进程配置文件的文件类型。

ircd_log_t：适合于 IRC 守护进程日志文件的文件类型。

ircd_var_lib_t：适合于存储在/var/lib/ircd 目录下的文件的文件类型。

这是一个适合于简单守护进程如 IRC 的有代表性的类型集，注意除了前面两个类型外，其它的类型都是由 ircd 控制的。

14.3.2.1. 示例策略

在示例策略中，类型是直接声明的或通过宏声明的，包括关联的属性列表也是如此，清单 14-2 显示了我们为示例策略 IRC 守护进程策略模块指定的类型声明，我们直接在策略模块中声明了所有的类型，没有使用到宏，注意每个类型有不同的属性，例如，日志文件类型 `ircd_log_t` 有 `file_type`, `sysadmfile` 和 `logfile` 属性，我们基于各个类型的使用情况(如 `ircd_log_t` 计划就是一个系统管理员可以访问的日志文件类型)和可用的属性决定需要的属性。

清单 14-2. 示例策略：IRC 守护进程类型声明(`ircd.te`)

```
1 #####
2 #
3 # Type declarations
4 #
5
6 # ircd domain
7 type ircd_t, domain;
8
9 # ircd entryptpoint
10 type ircd_exec_t, file_type, exec_type;
11
12 # PID file /var/run/ircd.pid
13 type ircd_var_run_t, file_type;
14
15 # configuration files
16 type ircd_conf_t, file_type, sysadmfile;
17
18 # log files
19 type ircd_log_t, file_type, sysadmfile, logfile;
20
21 # files and directories under /var/lib/ircd
22 type ircd_var_lib_t, file_type, sysadmfile;
```

提示：

示例策略源根目录下的 `attrib.te` 文件包括所有的属性声明和如何使用它们的文档。

14.3.2.2. 引用策略

在引用策略中，类型总是直接声明的，并且不会包括属性，清单 14-3 显示我们为 IRC 引用策略私有策略模块进行的类型声明(`ircd.te`)，注意每个类型声明都和一个接口调用(调用另一个策略模块)成对出现的，其功能和清单 14-2 示例策略模块中的属性参数差不多，例如，第 24 行声明了 `ircd_conf_t` 类型，第 25 行通过调用 `files_config_file()` 接口将它标记为一个配置文件，决定为各个类型声明调用哪个接口和决定使用哪个属性有点类似，引用策略有很好的参考文档可以使用。

清单 14-3. 引用策略：IRC 守护进程私有类型声明(`ircd.te`)

```
1 # Ircd policy module declaration
2 policy_module(ircd, 1.0)
```

```

3
4 #####
5 #
6 # Type declarations
7 #
8
9 # ircd domain
10 type ircd_t;
11
12 # ircd entrypoint
13 type ircd_exec_t;
14
15 # mark ircd_t as a domain and ircd_exec_t
16 # as an entrypoint into that domain
17 init_daemon_domain(ircd_t, ircd_exec_t)
18
19 # PID file /var/run/ircd.pid
20 type ircd_var_run_t;
21 files_pid_file(ircd_var_run_t)
22
23 # configuration files
24 type ircd_conf_t;
25 files_config_file(ircd_conf_t)
26
27 # log files
28 type ircd_log_t;
29 logging_log_file(ircd_log_t)
30
31 # files and directories under /var/lib/ircd
32 type ircd_var_lib_t;
33 files_type(ircd_var_lib_t)

```

提示：记住，引用策略包括大量源自源的文档，通过查找这些文档就很容易找出上面例子中的类型声明合适的接口，你可以在引用策略网站上查看这些文档，如果你在引用策略源目录下运行了 `make html` 命令，那就在本地也可以查看了，在 FC 5 中，HTML 文档位于 `/usr/share/doc/selinux-policy-x.y.z/html/`。

14.3.3. 最初允许的限制性访问

下一步是基于我们对初始化的理解，IRC 域类型(`ircd_t`)需要的限制性访问授予访问许可，允许的访问权应该反应我们的安全目标和 IRC 守护进程需要的功能，以我们的经验来看，首先用提取的方式来规划需要的访问权，因为编写原始的 SELinux 策略规则需要关注大量的细节，通过创建一个高级的计划，更容易记住安全目标，例如，我们希望 `ircd_t` 域有下面列出的访问权，以满足我们的安全目标：

创建日志文件、读取日志文件和扩展日志文件(`ircd_log_t`)

读取配置文件(ircd_conf_t)

创建 PID 文件、读取和写入 PID 文件(ircd_var_run_t)

创建变量文件、读取和写入变量文件(ircd_var_lib_t)

网络访问权

网络接口, TCP 发送和接收

节点, TCP 发送和接收

端口, IRC 端口上的 TCP name_bind, 发送和接收

解析 DNS 名称

使用共享库

读取本地资源

读取网络应用程序常常会使用到的目录和文件, 包括设备文件/dev/null 和/proc/目录下的 sysctl 配置数据

注意:

我们最初选择了非常宽松的网络访问权, 我们没有限制能够与 IRC 守护进程通信的网络接口和主机, 实际中通常会基于本地网络设置和拓扑结构对策略进行定制, 添加一些限制。

14.3.3.1. 示例策略

在示例策略中, 我们使用一组 allow 规则和示例策略宏来定义允许的访问权, 如清单 14-4。

清单 14-4. 示例策略: IRC 守护进程最初允许的访问权(ircd.te)

```
1 #####
2 #
3 # Ircd - core access
4 #
5
6 # Log files - create, read, and append
7 append_logdir_domain(ircd)
8
9 # Configuration files - read
10 allow ircd_t ircd_conf_t : dir r_dir_perms;
11 allow ircd_t ircd_conf_t : file r_file_perms;
12 allow ircd_t ircd_conf_t : lnk_file { getattr read };
13
14 # PID file - create, read, and write
15 file_type_auto_trans(ircd_t, var_run_t,
16 ircd_var_run_t, file)
17 allow ircd_t var_t : dir search;
18
19 # /var/lib/ircd files/dirs - create, read, write
20 file_type_auto_trans(ircd_t, var_lib_t,
21 ircd_var_lib_t, file)
22 allow ircd_t ircd_var_lib_t : dir rw_dir_perms;
```

```

21
22     # Network access - the ircd daemon is allowed to send
23     # and receive network data to all nodes and ports over
24     # all network interfaces (through the
can_network_server
25     # macro). Additionally, it can name_bind to the ircd
26     # port (ircd_port_t).
27     allow ircd_t ircd_port_t:tcp_socket name_bind;
28     can_network_server(ircd_t)
29
30     # use shared libraries
31     uses_shlib(ircd_t)
32
33     # read localization data
34     read_locale(ircd_t)
35
36     # read common directories / files including
37     #     * proc
38     #     * /dev/null
39     #     * system variables
40     allow ircd_t { self proc_t }:dir r_dir_perms;
41     allow ircd_t { self proc_t }:lnk_file { getattr read };
42     allow ircd_t null_device_t:chr_file rw_file_perms;
43     allow ircd_t sysctl_type:dir r_dir_perms;
44     allow ircd_t sysctl_type:file r_file_perms;
45     allow ircd_t sysctl_t:dir search;
46     allow ircd_t sysctl_kernel_t:dir search;
47     allow ircd_t sysctl_kernel_t:file { getattr read };

```

注意，每个被注释掉的规则块就对应一个初始访问权，为了允许访问在我们的模块中声明的类型，我们主要是直接使用 `allow` 规则来实现，例如，第 10 到 12 行允许 `ircd_t` 域类型读取配置文件（即类型为 `ircd_conf_t` 的目录和文件），然而，还是有些例外，有些策略规则是通过宏添加到我们的策略中来的，例如，第 19 行的 `file_type_auto_trans()` 宏允许 `ircd_t` 域类型创建、读取和写入类型为 `ircd_var_run_t` 的文件，即 `/var/run/ircd.pid`。

对类型声明的访问除了我们策略模块外，还可以使用一组 `allow` 规则和宏来实现，例如，第 42 行允许 `ircd_t` 域读取和写类型为 `null_device_t` 的字符设备文件，即 `/dev/null`，它是通过直接使用引用了这两个类型的 `allow` 规则来实现的，这是示例策略的最大缺点，即与策略模块结合得太紧密了，因为我们的 IRC 模块必须知道其它模块中的类型声明（`null_device_t`），所以这两个的实现是相互纠缠在一起的，相比之下，使用共享库需要的访问权是由 `uses_shlib()` 宏允许的，如第 31 行所显示的。

在示例策略中，选择使用直接访问权还是使用宏主要取决于风格以及是否有合适的宏，在引用策略中没有强制的规定。

IRC 守护进程需要的网络访问权是通过 `can_network()` 宏定义的，不幸的是，这个宏允许的访问权比我们应用程序需要的访问权要多一些，除了发送和接收 TCP 包外，它还允许发送和接收原始数据包和 UDP 数据包，除了直接允许规则外，没别的办法。

提示：

在我们的策略模块中使用的大部分宏在示例策略中也很常见，阅读现有的策略模块是最快熟悉宏及如何使用宏的最佳方法，阅读 `macros/*.te` 文件中的宏也是非常有帮助的。

14.3.3.2. 引用策略

引用策略中的访问权是通过一组 `allow` 规则和调用定义在其它模块中的接口来实现的，第 12 章中谈到对任何不是在策略模块中声明的类型的访问权只通过一个接口来实现，因此，和示例策略模块不同，IRC 守护进程私有策略文件永远都不会引用来自其它模块的类型，清单 14-5 是我们最初限制 IRC 守护进程的引用策略版本。

清单 14-5. 引用策略：IRC 守护进程独自允许的访问权(`ircd.te`)

```
1      #####
2      #
3      # Ircd - core access
4      #
5
6      # Log files - create, read, and append
7      allow ircd_t ircd_log_t : dir ra_dir_perms;
8      allow ircd_t ircd_log_t : file { create
ra_file_perms };
9      logging_filetrans_log(ircd_t, ircd_log_t, file)
10     logging_search_logs(ircd_t)
11
12     # Configuration files - read
13     allow ircd_t ircd_conf_t : dir r_dir_perms;
14     allow ircd_t ircd_conf_t : file r_file_perms;
15     allow ircd_t ircd_conf_t : lnk_file { getattr read };
16
17     # PID file - create, read, and write
18     allow ircd_t ircd_var_run_t : dir rw_dir_perms;
19     allow ircd_t ircd_var_run_t : file
create_file_perms;
20     files_filetrans_pid(ircd_t, ircd_var_run_t, file)
21
22     # /var/lib/ircd files/dirs - create, read, write
23     allow ircd_t ircd_var_lib_t : dir create_dir_perms;
24     allow ircd_t ircd_var_lib_t : file
create_file_perms;
25     files_filetrans_var_lib(ircd_t, ircd_var_lib_t,
{ file, dir })
26
```

```

27     # Network access - the ircd daemon is allowed to send
28     # and receive network data to all nodes and ports
over
29     # all network interfaces. Additionally, it can
name_bind
30     # to the ircd port (ircd_port_t)
31     allow ircd_t self : tcp_socket
create_stream_socket_perms;
32     corenet_tcp_sendrecv_all_if(ircd_t)
33     corenet_tcp_sendrecv_all_nodes(ircd_t)
34     corenet_tcp_sendrecv_all_ports(ircd_t)
35     corenet_non_ipsec_sendrecv(ircd_t)
36     corenet_tcp_bind_all_nodes(ircd_t)
37     corenet_tcp_bind_ircd_port(ircd_t)
38     sysnet_dns_name_resolve(ircd_t)
39
40     # use shared libraries
41     libs_use_ld_so(ircd_t)
42     libs_use_shared_libs(ircd_t)
43
44     # read localization data
45     miscfiles_read_localization(ircd_t)
46
47     # read common directories / files including
48     #      * /etc (search)
49     # * system variables
50     files_search_etc(ircd_t)
51     kernel_read_kernel_sysctl(ircd_t)
52     kernel_read_system_state(ircd_t)
53     kernel_read_all_sysctl(ircd_t)

```

再说一次，每个注释块代表我们允许的初始访问权，选择使用直接 **allow** 规则还是引用策略中的接口需遵循一个强制规定，使用直接 **allow** 规则会更简单直接，因为在引用策略中类型都被清晰地封装起来了。

注意引用策略中使用的接口比示例策略中的宏更清晰、明确，因此接口有时会使引用策略模块更细长，这样也可以获得更细粒度的访问控制，例如，我们在引用策略中允许的网络访问权就和我们最初制定的限制访问完全匹配，因为网络访问被分成许多接口，可以对每个端口进行控制。

14.3.4. 允许域转换和指派角色

为了让我们的新域生效，我们必须允许其它域转换成我们的新域，即必须创建 `type_transition` 规则，允许域转换，并给我们的域委派合适的角色。

常规情况下，许多想转换成守护进程的域应该会被限制，IRC 守护进程包括 `init` 脚本，允许在启动过程中从 `init` 启动，或由系统管理员直接启动，为了准许这两种启动方法，我们必须确保我们的策略有以下功能：

允许 `initrc_t` 通过 `ircd_exec_t` 自动转换成 `ircd_t` (允许 `init` 启动守护进程)。
允许 `sysadm_t` 通过 `ircd_exec_t` 自动转换到 `ircd_t` (允许系统管理员启动守护进程)。
授予 `ircd_t` `system_r` 角色 (授予 `init` 的角色)。
执行 `ircd_exec_t` 自动从 `sysadm_r` 转换到 `system_r` 角色 (授予系统管理员角色)。

注意：

以 `system_r` 角色运行的 IRC 守护进程不需要使用角色转换规则，我们已经授予 `ircd_t` `sysadm_r` 角色代替了，角色转换是使用 `system_r` 的标准操作规程，但因为它产生更多类似的安全上下文，不管守护进程是否由 `init` 启动还是由系统管理员启动。

14.3.4.1. 示例策略

清单 14-6 显示了我们在示例策略中实现的域转换和指派角色。

清单 14-6. 示例策略：IRC 守护进程域和角色指派 (`ircd.te`)

```
1 #####
2 #
3 # Domain Transitions and Role Authorizations
4 #
5
6 role system_r types ircd_t;
7
8 # allow init to start ircd
9 domain_auto_trans(initrc_t, ircd_exec_t, ircd_t)
10
11 # allow sysadm_t to start ircd_t
12 domain_auto_trans(sysadm_t, ircd_exec_t, ircd_t)
13 role_transition sysadm_r ircd_exec_t system_r;
```

`domain_auto_trans()` 宏允许域转换，并且为域自动转换添加了必要的类型转换规则。

14.3.4.2. 引用策略

我们已经在引用策略中使用接口 `init_daemon_domain()` 实现了这一步，如清单 14-3 的第 17 行所示，这个接口允许所有域和角色转换。

14.3.5. 整合进系统策略

我们的初始限制访问权主要是允许 IRC 守护进程需要的访问权，我们也允许其它域类型访问我们策略模块中的资源类型，例如，日志文件只有管理工具可以读取它们时才有用，这也是我们想整合进系统策略的原因。

更多其它访问权是通过示例策略中的属性和引用策略中的接口自动处理的，例如，添加 `file_type` 属性 (示例策略) 或调用 `file_type()` 接口 (引用策略) 允许不同的域，包括 `sysadm_t` 读取文件及关联的文件类型。

我们通常需要允许特定模块访问策略资源的特定类型，我们可以授予其它域访问权，为了便于说明，让我们扩展我们的模块允许其它域读取类型为 `ircd_log_t` 的文件。

14.3.5.1. 示例策略

在示例策略中没有定义允许从其它策略模块访问的方法，规则既可以放在我们的策略模块中也可以放在其它策略中，来自两个策略模块的类型要能够直接引用，例如，清单 14-7 中的策略语句允许 `logrotate_t` 读取 IRC 守护进程日志文件，在我们的示例策略中，我们只需要将这些规则放在 `logrotate` 模块中就可以了。

清单 14-7. 示例测试：IRC 守护进程，允许 `logrotate` 域访问(`ircd.te`)

注意我们使用了 `m4 ifdef` 语句将这些规则包装起来了，防止 `logrotate` 策略模块在策略编译期间不存在，使用这个方法的挑战是很难知道所有这些规则在策略中处于什么位置，这也是开发引用策略的另一个动机(即强化模块化和封装)。

```
1  #####
2  #
3  # Integrate Into System Policy
4  #
5
6  ifdef(`logrotate.te', `
7  allow logrotate_t ircd_log_t:dir search;
8  allow logrotate_t ircd_log_t:file { getattr read };
9  ')
```

14.3.5.2. 引用策略

在引用策略中通过使用接口允许来自其它策略模块的访问看起来更结构化些，清单 14-8 显示了 IRC 策略模块的扩展接口文件(`ircd.if`)，它为读取 IRC 守护进程日志文件声明了一个接口，正如第 12 章中讨论的那样，在引用策略中，从其它模块访问一个私有类型只有使用接口。

清单 14-8. 引用策略：IRC 守护进程扩展接口示例(`ircd.if`)

```
1  ## <summary>IRC daemon</summary>
2
3  #####
4  ## <summary>
5  ##           Read IRC daemon log files.
6  ## </summary>
7  ## <param name="domain">
8  ##           Domain allowed access.
9  ## </param>
10 #
11 interface(`irc_read_log', `
12     gen_require(`
13         type ircd_log_t;
14     ')
15
16     logging_search_logs($1)
```

```

17         allow $1 ircd_log_t:dir search_dir_perms;
18         allow $1 ircd_log_t:file r_file_perms;
19     ')

```

允许其它模块访问只需要在其它模块中调用这个接口即可，例如，为了允许 **logrotate** 读取 IRC 日志文件，在 **logrotate** 策略模块中添加下面的接口调用即可：

```
irc_read_log(logrotate_t)
```

注意：

接口文件也包括了模块摘要文档信息和各个接口的摘要说明，允许我们从引用策略源文件中生产更详细的接口文档。

14.3.6. 创建标记策略

接下来我们要完成我们的初始策略模块，即以文件上下文语句形式创建并应用标记策略，正如第 10 章“客体标记”中谈到的，标记策略为文件系统客体上的文件和目录分配类型，我们使用收集到的文件和目录的位置信息来获取语句。

14.3.6.1. 示例策略

清单 14-9 显示了示例策略中的文件上下文文件(ircd.fc)，注意这个文件直接通过 **setfiles** 程序硬编码列出了 IRC 守护进程需要的文件和目录。

清单 14-9. 示例策略：IRC 守护进程文件上下文文件(ircd.fc)

```

1     # ircd labeling policy
2     # file: ircd.fc
3     /usr/bin/ircd      --      system_u:object_r:ircd_exec_t
4     /etc/ircd(/.*)?    system_u:object_r:ircd_conf_t
5     /var/log/ircd(/.*)? system_u:object_r:ircd_log_t
6     /var/lib/ircd(/.*)? system_u:object_r:ircd_var_lib_t
7     /var/run/ircd(/.*)? system_u:object_r:ircd_var_run_t

```

14.3.6.2. 引用策略

清单 14-10 显示了我们引用策略模块的标记策略文件(ircd.fc)。

清单 14-10. 引用策略：IRC 守护进程标记策略文件(ircd.fc)

本质上，引用策略中的 **ircd.fc** 和示例策略中的文件是等同的，除了使用了 **gen_context()** 模板接口宏外，这个模板接口允许引用策略透明地处理多层安全/多范畴安全 (MLS/MCS)，以及来自相同策略源的非 MLS/MCS 策略，在引用策略中，所有安全上下文必须使用 **gen_context()** 指定

```

1     # ircd labeling policy
2     # file: ircd.fc
3     /usr/bin/ircd      --      gen_context(system_u:object_r:ircd_exec_t, s0)
4     /etc/ircd(/.*)?    gen_context(system_u:object_r:ircd_conf_t, s0)
5     /var/log/ircd(/.*)? gen_context(system_u:object_r:ircd_log_t, s0)
6     /var/lib/ircd(/.*)? gen_context(system_u:object_r:ircd_var_lib_t,
s0)
7     /var/run/ircd(/.*)? gen_context(system_u:object_r:ircd_var_run_t,
s0)

```

14.3.7. 应用策略

在测试之前的最后一步是编译、安装、载入和应用策略，这个操作在示例策略和引用策略中都一样，首先是编译、安装和载入策略：

```
# make && make install && make load
```

如果这一步执行成功，你不应该看到任何错误信息，构建系统也会显示一条策略载入成功的消息，例如，对于示例策略，成功编译后会输出下面的消息，引用策略的输出将会有点不一样。

此外，策略载入情况在审核日志中也可以看到，例如，下面是由策略载入事件产生的审核消息：

```
Building file contexts files...
/usr/bin/checkpolicy -o policy.20 policy.conf
/usr/bin/checkpolicy: loading policy configuration from policy.conf
/usr/bin/checkpolicy: policy configuration loaded
/usr/bin/checkpolicy: writing binary representation (version 20) to policy.20
Compiling policy ...
/usr/bin/checkpolicy -o /etc/selinux/strict/policy/policy.20 policy.conf
/usr/bin/checkpolicy: loading policy configuration from policy.conf
/usr/bin/checkpolicy: policy configuration loaded
/usr/bin/checkpolicy: writing binary representation (version 20) to
/etc/selinux/strict/policy/policy.20
/usr/bin/checkpolicy -c 19 -o /etc/selinux/strict/policy/policy.19 policy.conf
/usr/bin/checkpolicy: loading policy configuration from policy.conf
/usr/bin/checkpolicy: policy configuration loaded
/usr/bin/checkpolicy: writing binary representation (version 19) to
/etc/selinux/strict/policy/policy.19
install -m 644 tmp/system.users /etc/selinux/strict/users/system.users
install -m 644 tmp/customizable_types
/etc/selinux/strict/contexts/customizable_types
install -m 644 tmp/port_types /etc/selinux/strict/contexts/port_types
Installing file contexts files...
install -m 644 file_contexts/homedir_template
/etc/selinux/strict/contexts/files/homedir_template
install -m 644 file_contexts/file_contexts
/etc/selinux/strict/contexts/files/file_contexts
Loading Policy ...
/usr/sbin/load_policy /etc/selinux/strict/policy/policy.19
touch tmp/load
```

```
Feb 13 23:07:48 kernel: audit(1139890068.158:15709654): avc: granted
{load_policy } for pid=1173
comm="load_policy"scontext=root:sysadm_r:load_policy_t
tcontext=system_u:object_r:security_t tclass=security
```

在 FC 5 上构建和安装策略模块

通过使用可载入策略模块和策略 rpm 安装的开发环境，在 FC 5 中构建和安装策略模块是相当简单的，为了构建我们的引用策略 IRC 模块成为一个可载入模块，我们需要 1) 创建一个新的目录，2) 拷贝我们的 IRC 源文件到这个新目录(即 `ircd.te`, `ircd.fc` 和 `ircd.if`)，3) 从 `/usr/share/selinux/devel/Makefile` 拷贝示例可载入模块到这个新目录，这样这个目录就包括下面的文件：

```
$ lsircd.fc ircd.if ircd.te Makefile
```

现在运行 `make` 命令就可以构建一个可载入的策略模块了，例如：

```
$ make
```

```
Compiling targeted ircd module
```

```
/usr/bin/checkmodule: loading policy configuration from tmp/ircd.tmp
```

```
/usr/bin/checkmodule: policy configuration loaded
```

```
/usr/bin/checkmodule: writing binary representation (version 5) to
```

```
tmp/ircd.modCreating targeted ircd.pp policy packagerm tmp/ircd.mod
```

```
tmp/ircd.mod.fc
```

这个命令创建了策略包 `ircd.pp`，示例 `Makefile` 依靠当前活动策略使用安装在 `/usr/share/selinux/devel/include` 中的引用策略接口构建策略，策略包可以使用下面的命令进行安装：`# /usr/sbin/semodule -i ircd.pp`

如果安装我们的可载入策略时没有遇到错误，`semodule` 命令将会显示已经安装好的可载入模块：

```
# /usr/sbin/semodule -lircd 1.0
```

正如你所看到的，我们已经成功安装了我们的 IRC 策略包，现在它已经作为当前运行策略的一部分。

在策略成功安装和载入后，我们可以重新标记文件系统确保我们的新文件上下文文件有效，再说一次，这个过程对于示例策略和引用策略都是一样的，下面我们使用 `restorecon` 命令重新标记我们模块在文件上下文文件中指定的所有文件和目录：

```
# restorecon /usr/bin/ircd# restorecon -R /etc/ircd/ /var/log/ircd/  
/var/lib/ircd/
```

我们可以使用 `ls` 命令来校验标记是否正确(注意你也可以使用 `ls -Z`)，如：`# ls
scontext /usr/bin/ircd /var/log/ircd/system_u:object_r:ircd_exec_t /usr/bin/ircd
/var/log/ircd/:system_u:object_r:ircd_log_t ircd.log`

提示：

使用新定义的类型标记文件系统只有在载入新的策略后才会发生，因为内核必须要认识新的类型才行。

经过这些步骤后，我们为 IRC 守护进程创建的初始策略模块已经完成，可以开始测试了。

14. 4. 测试和分析策略

在测试和策略分析这一步中，我们校验我们的策略模块功能是否正确，是否符合我们的安全目标。

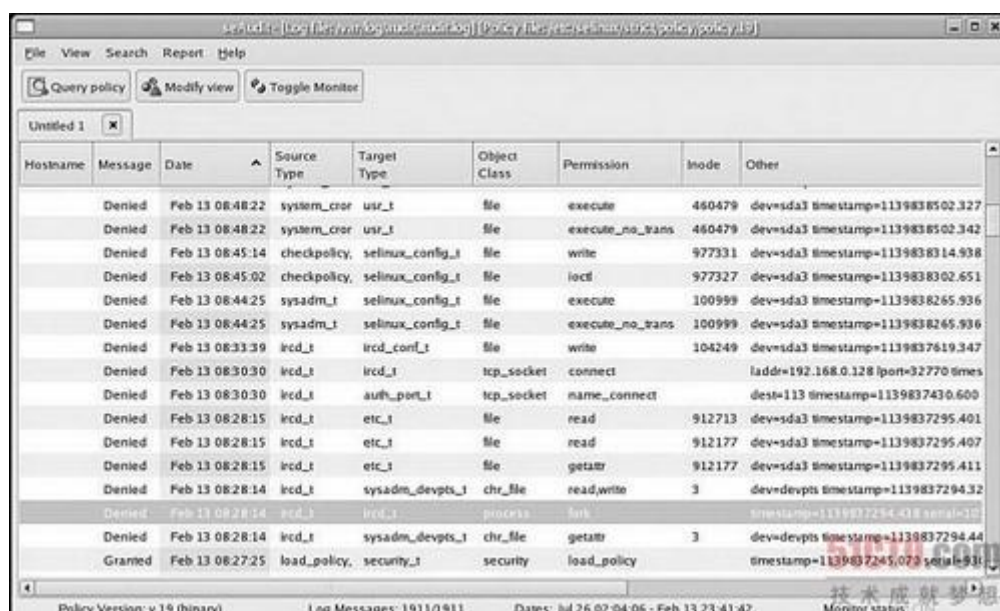
14. 4. 1. 测试策略模块

假设我们已经可以编译、安装和载入我们的新策略了，并且我们已经成功标记了文件系统，我们已经为 IRC 守护进程和策略模块功能测试做好准备，在这一步中，我们只执行最基本的功能测试，在生产环境中使用这个策略模块时应该执行更多的测试。

首先，我们校验系统处于 permissive 模式，并且守护进程处于停止状态，在 permissive 模式下运行允许 IRC 守护进程功能正确以便我们可以看到所有需要的但我们的策略模块中不允许的访问权，在 permissive 模式下，访问拒绝只是被审核而不是强制实施，切换到 permissive 模式和停止 IRC 守护进程的命令如下：

```
# setenforce 0# /etc/init.d/ircd stopStopping ircd: [ok]
```

接下来，我们需要使用来自 setools 包中的 Seaudit 实用程序，在 FC 4 系统上它应该被安装了(参考附录 D“SELinux 命令和实用程序”中对 setools 的介绍，它是一款开源软件，包括 apol 工具)，使用 Seaudit 打开审核日志，使用固定监视器(Toggle Monitor)按钮打开监视，确保在状态栏中的状态是开启的(On)，如图 14-1 所示，我们使用 Seaudit 查看审核日志以确定 IRC 守护进程是否需要我们策略模块没有的额外访问权，我们也可以使用 tail 工具来查看日志文件，如 tail -f /var/log/audit/audit.log。



Hostname	Message	Date	Source Type	Target Type	Object Class	Permission	Inode	Other
	Denied	Feb 13 08:48:22	system_cron	usr_t	file	execute	460479	dev=sd3 timestamp=1139838502.327
	Denied	Feb 13 08:48:22	system_cron	usr_t	file	execute_no_trans	460479	dev=sd3 timestamp=1139838502.342
	Denied	Feb 13 08:45:14	checkpolicy	selinux_config_t	file	write	977331	dev=sd3 timestamp=1139838314.938
	Denied	Feb 13 08:45:02	checkpolicy	selinux_config_t	file	lockf	977327	dev=sd3 timestamp=1139838302.651
	Denied	Feb 13 08:44:25	sysadm_t	selinux_config_t	file	execute	100999	dev=sd3 timestamp=1139838265.936
	Denied	Feb 13 08:44:25	sysadm_t	selinux_config_t	file	execute_no_trans	100999	dev=sd3 timestamp=1139838265.936
	Denied	Feb 13 08:33:39	ircd_t	ircd_conf_t	file	write	104249	dev=sd3 timestamp=1139837619.347
	Denied	Feb 13 08:30:30	ircd_t	ircd_t	tcp_socket	connect		laddr=192.168.0.128 lport=32770 times
	Denied	Feb 13 08:30:30	ircd_t	auth_port_t	tcp_socket	name_connect		dest=113 timestamp=1139837430.600
	Denied	Feb 13 08:28:15	ircd_t	etc_t	file	read	912713	dev=sd3 timestamp=1139837295.401
	Denied	Feb 13 08:28:15	ircd_t	etc_t	file	read	912177	dev=sd3 timestamp=1139837295.407
	Denied	Feb 13 08:28:15	ircd_t	etc_t	file	getattr	912177	dev=sd3 timestamp=1139837295.411
	Denied	Feb 13 08:28:14	ircd_t	sysadm_devpts_t	chr_file	read,write	3	dev=devpts timestamp=1139837294.32
	Denied	Feb 13 08:28:14	ircd_t	ircd_t	process	fork		timestamp=1139837294.438 serial=10
	Denied	Feb 13 08:28:14	ircd_t	sysadm_devpts_t	chr_file	getattr	3	dev=devpts timestamp=1139837294.44
	Granted	Feb 13 08:27:25	load_policy	security_t	security	load_policy		timestamp=1139837245.079 serial=931

图 14-1. Seaudit 显示测试 IRC 守护进程时产生的审核消息

启动 Seaudit 后，我们使用下面的命令启动 IRC 守护进程：

```
# setenforce 0# /etc/init.d/ircd startStarting ircd: ircd: version  
hybrid-7.2.0ircd: pid 9052ircd: running in background mode from /usr/lib/ircd [ OK ]
```

如果所有配置都是正确的，我们应该可以使用 ps 命令显示 IRC 守护进程的正确类型，如：

```
# ps axZ | grep ircdroot:system_r:ircd_t 1519 ? 00:00:00 ircd
```

我们看到 IRC 守护进程以正确的安全上下文在运行。

接下来，我们使用 IRC 客户端连接到 IRC 守护进程，例如，图 14-2 显示了 xchat 客户端成功连接到 IRC 守护进程。



图 14-2. 使用 xchat 连接到 IRC 守护进程

提示:

注意如果你在一台独立的计算机上使用 IRC 客户端, 在测试系统上确保防火墙设置允许 IRC 通讯。

14.4.1.1. 评估审核消息允许额外的访问

在简单测试了 IRC 守护进程后, 如加入一个频道进行交谈, 我们可以分析与我们的策略模块有关的拒绝审核日志, 图 14-1 显示了我们在测试 IRC 守护进程期间产生的相关审核消息, 这些消息显示 IRC 守护进程需要五个额外的访问权, 但在我们的初始策略中却不允许这些访问:

- 读取/etc/下的配置文件(etc_t)

- 派生另一个进程

- 读取和写入系统管理员所属的伪终端(sysadm_devpts_t)

- 写入配置文件(ircd_conf_t)

我们必须谨慎思考每一个需要的访问权, 确定在我们的策略模块中是否要将它们添加进来, 在评估审核消息时, 目标不是添加 allow 规则直到拒绝消息消失, 相反, 每个需要的访问权都应该仔细思考, 如果它们与我们的安全目标冲突, 如果应用程序可以正常地运行那就不要允许它。

例如, 在前面列出的审核消息中, 我们看到 IRC 守护进程尝试访问(写访问)它的配置文件(ircd_conf_t), 如果允许这个访问权就违背了我们的安全目标(即保护配置文件), 同样, 允许它读取和写系统管理员伪终端的访问权也不是必需的, 这样可能打开一个潜在的攻击向量, 其它访问需要看起来还是比较合理的, 因此我们添加 allow 规则允许这些访问, 相反, 对于允许对配置文件的写访问和对系统管理员伪终端读和写访问, 我们使用 dontaudit 规则来禁止, 然后测试 IRC 守护进程没有这些访问权时功能是否都能够正常运转。

提示:

包含意外类型的审核消息可能标志着标记问题，例如，sysadm_t 访问 IRC 守护进程有关的类型的拒绝消息可能标志着入口点 (/usr/bin/ircd) 没有标记正确(ircd_exec_t)，阻止了域转换。

14.4.1.2. 在示例策略中添加额外的访问权

在示例策略 ircd.te 文件中使用下面的策略语句允许的额外访问权：

```
allow ircd_t self : process fork;allow ircd_t etc_t : file r_file_perms;
```

审核消息中我们不允许访问的访问权使用下面的 dontaudit 规则：

```
dontaudit ircd_t ircd_conf_t : file write;dontaudit ircd_t sysadm_devpts_t :  
chr_file { getattr read write };
```

14.4.1.3. 在引用策略中添加额外的访问权

为了添加其它额外访问权，我们在 ircd.te 文件中添加下面的策略语句：

```
allow ircd_t self : process fork;files_read_etc_files(ircd_t)
```

和前面一样，审核消息中我们不允许访问的访问权使用下面的 aontaudit 规则和接口调用：

```
dontaudit ircd_t ircd_conf_t : file  
write;userdom_dontaudit_use_sysadm_ptys(ircd_t)
```

14.4.2. 策略分析

最后一步是执行策略分析校验是否符合我们的安全目标，功能测试是不够的，使用自动工具如 apol 来分析策略，确保我们的策略模块非常安全。

例如，图 14-3 显示了在 apol 中搜索所有 ircd_t 对 ircd_conf_t 的访问，包括通过属性的间接访问，这样我们可以校验 IRC 守护进程(ircd_t)不能对它的配置文件进行写访问(ircd_conf_t)。

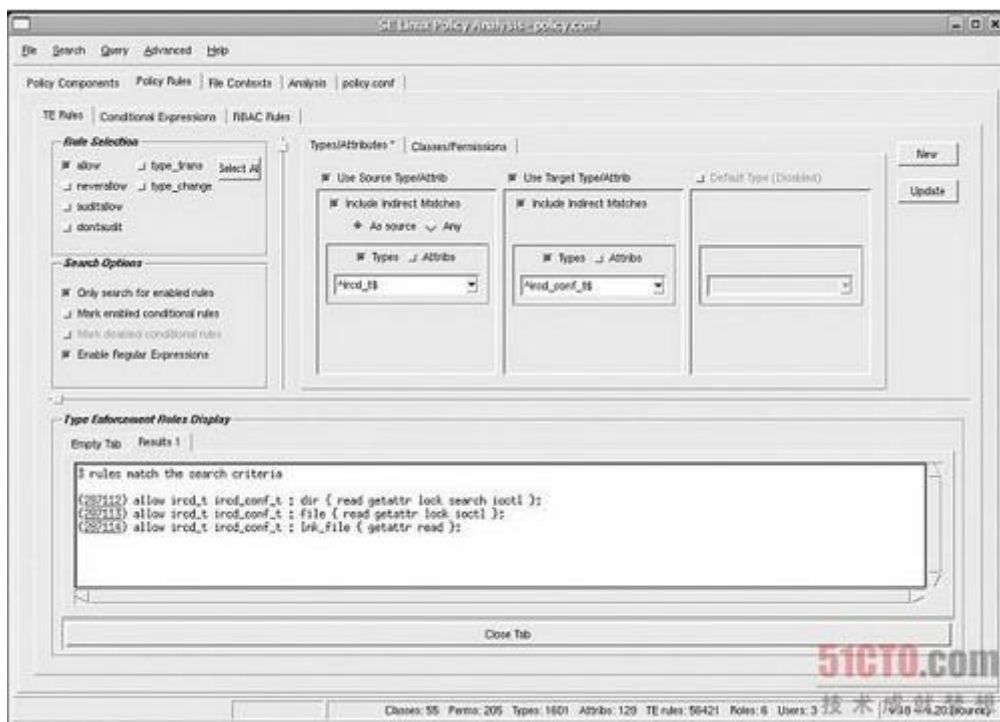


图 14-3. 显示 ircd_t 不能对 ircd_conf_t 写访问的 apol 规则搜索

14.5. 新兴的策略开发工具

最近出现了许多新兴的策略开发工具，简化了策略模块的开发过程，这些集成开发环境如 SLIDE(如图 14-4 所示)和自动策略产生工具，如 Polgen，附录 D 提供了这些工具的更多信息。

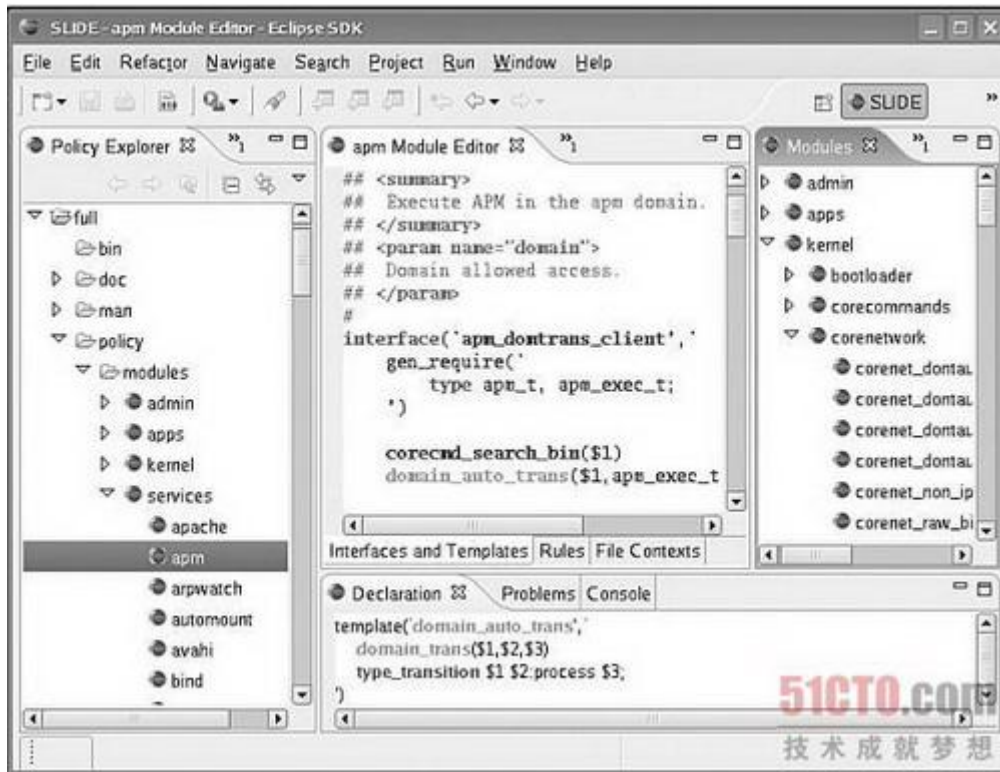


图 14-4. SLIDE 集成策略开发环境

14.6. 完整的 IRC 守护进程模块清单(1)

下面我们分别列出使用示例策略和引用策略的 IRC 守护进程模块：

清单 14-11. 示例策略：IRC 守护进程策略模块文件(ircd.te)

清单 14-12. 示例策略：IRC 守护进程文件上下文文件(ircd.fc)

```
1 #####
2 #
3 # ircd policy module
4 #
5 # file: ircd.te
6 #
7
8 #####
9 #
10 # Type declarations
11 #
```

```

12
13     # ircd domain
14     type ircd_t, domain;
15
16     # ircd entryptpoint
17     type ircd_exec_t, file_type, exec_type;
18
19     # PID file /var/run/ircd.pid
20     type ircd_var_run_t, file_type;
21
22     # configuration files
23     type ircd_conf_t, file_type, sysadmfile;
24
25     # log files
26     type ircd_log_t, file_type, sysadmfile, logfile;
27
28     # files and directories under /var/lib/ircd
29     type ircd_var_lib_t, file_type, sysadmfile;
30
31     #####
32     #
33     # Ircd - core access
34     #
35
36     # allow ircd_t to fork copies of itself
37     allow ircd_t self : process fork;
38     # Log files - create, read, and append
39     allow ircd_t var_log_t : dir ra_dir_perms;
40     allow ircd_t ircd_log_t : dir ra_dir_perms;
41     allow ircd_t ircd_log_t : file { create ra_file_perms };
42     type_transition ircd_t var_log_t : { file dir } ircd_log_t;
43
44     # Configuration files - read
45     allow ircd_t ircd_conf_t : dir r_dir_perms;
46     allow ircd_t ircd_conf_t : file r_file_perms;
47     allow ircd_t ircd_conf_t : lnk_file { getattr read };
48     dontaudit ircd_t ircd_conf_t : file write;
49
50     # PID file - create, read, and write
51     file_type_auto_trans(ircd_t, var_run_t, ircd_var_run_t, file)
52     allow ircd_t var_t : dir search;
53
54     # /var/lib/ircd files/dirs - create, read, write
55     file_type_auto_trans(ircd_t, var_lib_t, ircd_var_lib_t, file)

```

```

56     allow ircd_t ircd_var_lib_t : dir rw_dir_perms;
57
58     # Network access - the ircd daemon is allowed to send
59     # and receive network data to all nodes and ports over
60     # all network interfaces (through the can_network_server
61     # macro). Additionally, it can name_bind to the ircd
62     # port (ircd_port_t).
63     allow ircd_t ircd_port_t:tcp_socket name_bind;
64     can_network_server(ircd_t)
65
66     # use shared libraries
67     uses_shlib(ircd_t)
68
69     # read localization data
70     read_locale(ircd_t)
71
72     # read common directories / files including
73     #     * /etc/resolv.conf (etc_t)
74     #     * proc
75     #     * /dev/null
76     #     * system variables
77     allow ircd_t etc_t : file r_file_perms;
78     allow ircd_t { self proc_t }:dir r_dir_perms;
79     allow ircd_t { self proc_t }:lnk_file { getattr read };
80     allow ircd_t null_device_t:chr_file rw_file_perms;
81     allow ircd_t sysctl_type:dir r_dir_perms;
82     allow ircd_t sysctl_type:file r_file_perms;
83     allow ircd_t sysctl_t:dir search;
84     allow ircd_t sysctl_kernel_t:dir search;
85     allow ircd_t sysctl_kernel_t:file { getattr read };
86
87     #####
88     #
89     # Domain Transitions and Role Authorizations
90     #
91
92     role system_r types ircd_t;
93
94     # allow init to start ircd
95     domain_auto_trans(initrc_t, ircd_exec_t, ircd_t)
96
97     # allow sysadm_t to start ircd_t
98     domain_auto_trans(sysadm_t, ircd_exec_t, ircd_t)
99     role_transition sysadm_r ircd_exec_t system_r;

```

```

100 # dontaudit use of the sysadm_r terminal
101 dontaudit ircd_t sysadm_devpts_t : chr_file { getattr read write };
102
103 #####
104 #
105 # Integrate Into System Policy
106 #
107
108 ifdef(`logrotate.te', `
109     allow logrotate_t ircd_var_run_t:dir search;
110     allow logrotate_t ircd_var_run_t:file { getattr read };
111 ')

```

```

1 # ircd labeling policy
2 # file: ircd.fc
3 /usr/bin/ircd -- system_u:object_r:ircd_exec_t
4 /etc/ircd(/.*)? system_u:object_r:ircd_conf_t
5 /var/log/ircd(/.*)? system_u:object_r:ircd_log_t
6 /var/lib/ircd(/.*)? system_u:object_r:ircd_var_lib_t
7 /var/run/ircd(/.*)? system_u:object_r:ircd_var_run_t

```

14.6. 完整的 IRC 守护进程模块清单(1)

下面我们分别列出使用示例策略和引用策略的 IRC 守护进程模块：

清单 14-11. 示例策略：IRC 守护进程策略模块文件(ircd.te)

清单 14-12. 示例策略：IRC 守护进程文件上下文文件(ircd.fc)

```

1 #####
2 #
3 # ircd policy module
4 #
5 # file: ircd.te
6 #
7
8 #####
9 #
10 # Type declarations
11 #
12
13 # ircd domain
14 type ircd_t, domain;
15
16 # ircd entrypoint
17 type ircd_exec_t, file_type, exec_type;

```



```
18
19     # PID file /var/run/ircd.pid
20     type ircd_var_run_t, file_type;
21
22     # configuration files
23     type ircd_conf_t, file_type, sysadmfile;
24
25     # log files
26     type ircd_log_t, file_type, sysadmfile, logfile;
27
28     # files and directories under /var/lib/ircd
29     type ircd_var_lib_t, file_type, sysadmfile;
30
31     #####
32     #
33     # Ircd - core access
34     #
35
36     # allow ircd_t to fork copies of itself
37     allow ircd_t self : process fork;
38     # Log files - create, read, and append
39     allow ircd_t var_log_t : dir ra_dir_perms;
40     allow ircd_t ircd_log_t : dir ra_dir_perms;
41     allow ircd_t ircd_log_t : file { create ra_file_perms };
42     type_transition ircd_t var_log_t : { file dir } ircd_log_t;
43
44     # Configuration files - read
45     allow ircd_t ircd_conf_t : dir r_dir_perms;
46     allow ircd_t ircd_conf_t : file r_file_perms;
47     allow ircd_t ircd_conf_t : lnk_file { getattr read };
48     dontaudit ircd_t ircd_conf_t : file write;
49
50     # PID file - create, read, and write
51     file_type_auto_trans(ircd_t, var_run_t, ircd_var_run_t, file)
52     allow ircd_t var_t : dir search;
53
54     # /var/lib/ircd files/dirs - create, read, write
55     file_type_auto_trans(ircd_t, var_lib_t, ircd_var_lib_t, file)
56     allow ircd_t ircd_var_lib_t : dir rw_dir_perms;
57
58     # Network access - the ircd daemon is allowed to send
59     # and receive network data to all nodes and ports over
60     # all network interfaces (through the can_network_server
61     # macro). Additionally, it can name_bind to the ircd
```

```

62     # port (ircd_port_t).
63     allow ircd_t ircd_port_t:tcp_socket name_bind;
64     can_network_server(ircd_t)
65
66     # use shared libraries
67     uses_shlib(ircd_t)
68
69     # read localization data
70     read_locale(ircd_t)
71
72     # read common directories / files including
73     #     * /etc/resolv.conf (etc_t)
74     #     * proc
75     #     * /dev/null
76     #     * system variables
77     allow ircd_t etc_t : file r_file_perms;
78     allow ircd_t { self proc_t }:dir r_dir_perms;
79     allow ircd_t { self proc_t }:lnk_file { getattr read };
80     allow ircd_t null_device_t:chr_file rw_file_perms;
81     allow ircd_t sysctl_type:dir r_dir_perms;
82     allow ircd_t sysctl_type:file r_file_perms;
83     allow ircd_t sysctl_t:dir search;
84     allow ircd_t sysctl_kernel_t:dir search;
85     allow ircd_t sysctl_kernel_t:file { getattr read };
86
87     #####
88     #
89     # Domain Transitions and Role Authorizations
90     #
91
92     role system_r types ircd_t;
93
94     # allow init to start ircd
95     domain_auto_trans(initrc_t, ircd_exec_t, ircd_t)
96
97     # allow sysadm_t to start ircd_t
98     domain_auto_trans(sysadm_t, ircd_exec_t, ircd_t)
99     role_transition sysadm_r ircd_exec_t system_r;
100    # dontaudit use of the sysadm_r terminal
101    dontaudit ircd_t sysadm_devpts_t : chr_file { getattr read write };
102
103    #####
104    #
105    # Integrate Into System Policy

```

```

106  #
107
108  ifdef(`logrotate.te', `
109      allow logrotate_t ircd_var_run_t:dir search;
110      allow logrotate_t ircd_var_run_t:file { getattr read };
111  ')

```

```

1  # ircd labeling policy
2  # file: ircd.fc
3  /usr/bin/ircd      --      system_u:object_r:ircd_exec_t
4  /etc/ircd(/.*)?    system_u:object_r:ircd_conf_t
5  /var/log/ircd(/.*)? system_u:object_r:ircd_log_t
6  /var/lib/ircd(/.*)? system_u:object_r:ircd_var_lib_t
7  /var/run/ircd(/.*)? system_u:object_r:ircd_var_run_t

```

14.6. 完整的 IRC 守护进程模块清单(2)

清单 14-13. 引用策略: IRC 守护进程私有策略文件(ircd.te)

```

1  #####
2  #
3  # Reference Policy ircd policy module
4  #
5  # file: ircd.te
6  #
7
8  # Ircd policy module declaration
9  policy_module(ircd, 1.0)
10
11 #####
12 #
13 # Type declarations
14 #
15
16 # ircd domain
17 type ircd_t;
18
19 # ircd entrypoint
20 type ircd_exec_t;
21
22 # mark ircd_t as a domain and ircd_exec_t
23 # as an entrypoint into that domain
24 init_daemon_domain(ircd_t, ircd_exec_t)
25
26 # PID file /var/run/ircd.pid
27 type ircd_var_run_t;

```

```

28     files_pid_file(ircd_var_run_t)
29
30     # configuration files
31     type ircd_conf_t;
32     files_config_file(ircd_conf_t)
33
34     # log files
35     type ircd_log_t;
36     logging_log_file(ircd_log_t)
37
38     # files and directories under /var/lib/ircd
39     type ircd_var_lib_t;
40     files_type(ircd_var_lib_t)
41
42     #####
43     #
44     # Ircd - core access
45     #
46
47     # allow ircd_t to fork copies of itself
48     allow ircd_t self : process fork;
49
50     # Log files - create, read, and append
51     allow ircd_t ircd_log_t : dir ra_dir_perms;
52     allow ircd_t ircd_log_t : file { create ra_file_perms };
53     logging_log_filetrans(ircd_t, ircd_log_t, file)
54     logging_search_logs(ircd_t)
55
56     # Configuration files - read
57     allow ircd_t ircd_conf_t : dir r_dir_perms;
58     allow ircd_t ircd_conf_t : file r_file_perms;
59     allow ircd_t ircd_conf_t : lnk_file { getattr read };
60     dontaudit ircd_t ircd_conf_t : file write;
61
62     # PID file - create, read, and write
63     allow ircd_t ircd_var_run_t : dir rw_dir_perms;
64     allow ircd_t ircd_var_run_t : file create_file_perms;
65     files_pid_filetrans(ircd_t, ircd_var_run_t, file)
66
67     # /var/lib/ircd files/dirs - create, read, write
68     allow ircd_t ircd_var_lib_t : dir create_dir_perms;
69     allow ircd_t ircd_var_lib_t : file create_file_perms;
70     files_var_lib_filetrans(ircd_t, ircd_var_lib_t, { file dir })
71

```

```

72     # Network access - the ircd daemon is allowed to send
73     # and receive network data to all nodes and ports over
74     # all network interfaces. Additionally, it can name_bind
75     # to the ircd port (ircd_port_t)
76     allow ircd_t self : tcp_socket create_stream_socket_perms;
77     corenet_tcp_sendrecv_all_if(ircd_t)
78     corenet_tcp_sendrecv_all_nodes(ircd_t)
79     corenet_tcp_sendrecv_all_ports(ircd_t)
80     corenet_non_ipsec_sendrecv(ircd_t)
81     corenet_tcp_bind_all_nodes(ircd_t)
82     corenet_tcp_bind_ircd_port(ircd_t)
83     sysnet_dns_name_resolve(ircd_t)
84
85     # use shared libraries
86     libs_use_ld_so(ircd_t)
87     libs_use_shared_libs(ircd_t)
88
89     # read localization data
90     miscfiles_read_localization(ircd_t)
91
92     # dontaudit use of the sysadm_r terminal
93     userdom_dontaudit_use_sysadm_ptys(ircd_t)
94
95     # read common directories / files including
96     #     * /etc (search and read)
97     #     * system variables
98     files_search_etc(ircd_t)
99     files_read_etc_files(ircd_t)
100    kernel_read_kernel_sysctls(ircd_t)
101    kernel_read_system_state(ircd_t)
102    kernel_read_all_sysctls(ircd_t)

```

清单 14-14. 引用策略：IRC 守护进程标记策略文件(ircd.fc)

```

1     # ircd labeling policy
2     # file: ircd.fc
3     /usr/bin/ircd      --      gen_context(system_u:object_r:ircd_exec_t, s0)
4     /etc/ircd(/.*)?    gen_context(system_u:object_r:ircd_conf_t, s0)
5     /var/log/ircd(/.*)? gen_context(system_u:object_r:ircd_log_t, s0)
6     /var/lib/ircd(/.*)? gen_context(system_u:object_r:ircd_var_lib_t,
s0)
7     /var/run/ircd(/.*)? gen_context(system_u:object_r:ircd_var_run_t,
s0)

```

清单 14-15. 引用策略：IRC 守护进程外部接口文件(ircd.if)

```
1      ## <summary>IRC daemon</summary>
2
3      #####
4      ## <summary>
5      ##      Read IRC daemon log files.
6      ## </summary>
7      ## <param name="domain">
8      ##      Domain allowed access.
9      ## </param>
10     #
11     interface(`irc_read_log',`
12         gen_require(`
13             type ircd_log_t;
14         ')
15
16         files_search_var($1)
17         logging_search_logs($1)
18         allow $1 ircd_log_t:dir search_dir_perms;
19         allow $1 ircd_log_t:file r_file_perms;
20     ')
```

14.7. 小结

在所有现代企业一样，编写策略模块的技巧源于实践。

编写策略模块的基本步骤，不管是使用示例策略还是引用策略，都一样，如：

1、准备和计划

收集应用程序相关的信息

创建测试配置

制定安全目标

2、创建初始策略模块

创建基础模块文件

声明模块类型

允许初始限制访问权

允许域转换和角色访问

整合进系统策略

创建标记策略

应用策略

3、测试和分析策略

测试策略模块的功能

分析策略模块，是否符合我们的安全目标

通常，我们要反复执行这些步骤直到我们实现了我们想要的策略模块。

