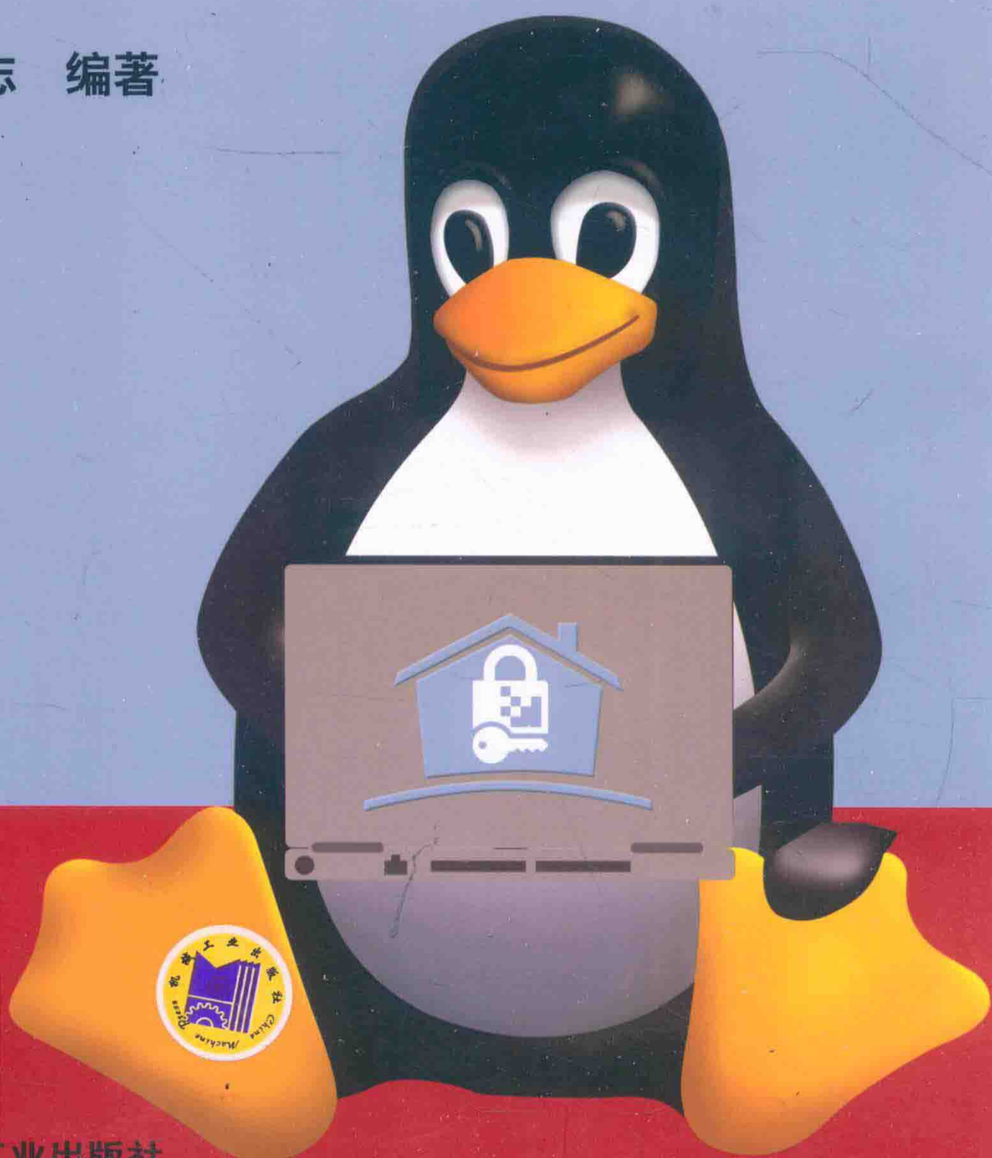
Linux
内核安全模块深入剖析

李志 编著



机械工业出版社
CHINA MACHINE PRESS

Linux 内核安全模块深入剖析

李　志　编著

机械工业出版社

本书对 Linux 内核安全子系统做了系统而深入的分析，内容包括 Linux 内核的自主访问控制、强制访问控制、完整性保护、审计日志、密钥管理与密钥使用等。

本书可供希望深入了解 Linux 安全的读者参考，也可供计算机安全专业的学生和计算机安全领域的从业人员阅读，还可用作计算机安全高级课程的教材。

图书在版编目（CIP）数据

Linux 内核安全模块深入剖析 / 李志编著. —北京：机械工业出版社，2016.9
ISBN 978-7-111-54905-5

Ⅰ. ①L… Ⅱ. ①李… Ⅲ. ①Linux 操作系统－安全技术 Ⅳ. ①TP316.85

中国版本图书馆 CIP 数据核字（2016）第 226091 号

机械工业出版社（北京市百万庄大街 22 号 邮政编码 100037）
责任编辑：车 忱 责任校对：张艳霞
责任印制：李 洋
三河市宏达印刷有限公司印刷
2016 年 11 月第 1 版 • 第 1 次印刷
184mm×260mm • 16.25 印张 • 393 千字
0001—3000 册
标准书号：ISBN 978-7-111-54905-5
定价：55.00 元

凡购本书，如有缺页、倒页、脱页，由本社发行部调换

电话服务
服务咨询热线：010－88379833
读者购书热线：010－88379649

网络服务
机 工 官 网：www.cmpbook.com
机 工 官 博：weibo.com/cmp1952
教育服务网：www.cmpedu.com
金 书 网：www.golden-book.com

封面无防伪标均为盗版

前　言

在很多数人眼中，操作系统内核是神秘的，安全是说不清的，内核安全则是子虚乌有的。如果说内核是一座险峻的高山，内核安全则是隐藏在高山之中的一个深不可测的洞窟。

即使对内核开发人员，内核安全也是陌生的。Linux 的创始人 Linus Torvalds 就曾说他不懂安全。内核开发人员不了解安全的原因主要有两个：其一，内核安全涉及领域太多了，Linux 是庞大的，很少有人能对所有领域都了解；其二，如 Linus Torvalds 所说，安全是“软”科学，没有一个简单的量化的指标去衡量安全，开发人员自然是重视实实在在的功能，而轻视“虚无缥缈”的安全。

我十二岁的儿子看到我整日为内核安全而忙碌，就问我什么是内核，什么是安全。经过我的解释，他对内核的印象就是隐藏在网络中的一个坚硬的“蛋”，内核安全就是“蛋”里面隐藏的“武林高手”。当不怀好意的攻击者攻击系统时，“蛋”中隐藏的“武林高手”就会跳出来保卫系统。基于这种理解，他画出了下面这幅画：

上面这幅童真的绘画表达了三个意思。其一是网络，在万物互联的今天，安全越来越难以忽视。其二是内核，处于底层的内核为上层的用户态应用提供支持和保障，其中也包含安全保障。其三是内核安全，今天的 Linux 内核已经拥有相当多的安全机制来应对各种安全威胁。但是 Linux 内核安全十分复杂，不易掌握。至本书成稿时，国内外还没有一本系统、全面地介绍 Linux 内核安全的书。

屈指算来，我从事内核安全相关工作已是第八个年头。当年的一个面试邀请电话让我在一个仲春的清晨伴着蒙蒙细雨步入了 Linux 内核安全领域。几年后，就在我对内核安全有了一点点理解，并沾沾自喜时，我亲耳聆听了 Linux 内核安全领域负责人 James Morris 的演讲。我突然发现他讲的大部分东西我都听不懂。好吧，不懂可以学。本书的提纲可以说是 James Morris 提供的。我将他演讲中提到的内核安全相关内容研究了一遍，呈现在这里。

本书的内容有相当一部分出自我对 Linux 内核源代码代码的分析。很多内核安全模块缺少文档，读代码是了解它们最直接、最准确的方式。代码面前，了无秘密。但是为了不让读者因为陷入代码的细节之中，而失去对软件框架的掌握，本书尽量避免简单罗列源代码，所列代码一般是关键代码，用省略号表示略去的不重要的部分。本书也列出了一些用户态工具的用例，讲解用户态工具的使用不是本书的目的，列出工具的用例是为了让读者对内核安全有些感性认识。

我很早就想写一本关于 Linux 内核安全的书，但真正提笔却源于一时兴起。兴起之时的 Linux 内核版本是 3.14-rc4，于是 3.14-rc4 就成了本书参考的 Linux 内核版本。读者可以浏览 http://lxr.linux.no/linux+v3.14-rc4/查阅此版本的内核代码。

本人才疏学浅，经验不足，书中错误在所难免，还望各位读者多多包涵。读者可以通过微博和微信同我交流，指出本书的错误与不足，我的微博昵称是“xiaoxiangfu12”，微信号也是“xiaoxiangfu12”。

作　者

试读结束：需要全本请在线购买： www.ertongbook.com

目　录

前言
第1章　导言……1
1.1　什么是安全……1
1.2　计算机系统安全的历史……1
1.3　计算机系统安全的现状……2
1.4　Linux内核安全概貌……3

第一部分　自主访问控制

第2章　主体标记与进程凭证……5
2.1　进程凭证……5
2.2　详述……6
2.2.1　uid和gid……6
2.2.2　系统调用……7
2.3　proc文件接口……9
2.4　参考资料……9
习题……9
第3章　客体标记与文件属性……10
3.1　文件的标记……10
3.2　文件属性……10
3.3　系统调用……11
3.4　其他客体……12
3.5　其他客体的系统调用……14
习题……15
第4章　操作与操作许可……16
4.1　操作……16
4.2　操作许可与操作分类……16
4.3　允许位……18
4.4　设置位……19
4.4.1　文件……19
4.4.2　目录……19
4.5　其他操作的许可……20
4.6　系统调用……20
习题……20
第5章　访问控制列表……21
5.1　简介……21
5.2　扩展属性……21
5.3　结构……21
5.4　操作许可……23
5.5　两种ACL……23
5.6　与允许位的关系……23
5.7　系统调用……23
5.8　参考资料……24
习题……24
第6章　能力（capabilities）……25
6.1　什么是能力……25
6.2　能力列举……25
6.2.1　文件……26
6.2.2　进程……27
6.2.3　网络……28
6.2.4　ipc……28
6.2.5　系统……28
6.2.6　设备……28
6.2.7　审计……28
6.2.8　强制访问控制（MAC）……28
6.3　UNIX的特权机制……29
6.4　Linux的能力集合和能力机制……29
6.4.1　能力集合……29
6.4.2　能力机制……30
6.5　向后兼容……32
6.6　打破向后兼容……33
6.7　总结……34
6.8　参考资料……34
习题……34

第二部分　强制访问控制

第 7 章　SELinux ······ 36

7.1　简介 ······ 36

7.1.1　历史 ······ 36

7.1.2　工作原理 ······ 36

7.1.3　SELinux 眼中的世界 ······ 39

7.2　机制 ······ 39

7.2.1　安全上下文 ······ 39

7.2.2　客体类别和操作 ······ 40

7.2.3　安全上下文的生成和变化 ······ 59

7.3　安全策略 ······ 62

7.3.1　基本定义 ······ 63

7.3.2　安全上下文定义 ······ 68

7.3.3　安全上下文转换 ······ 72

7.3.4　访问控制 ······ 75

7.3.5　访问控制的限制和条件 ······ 75

7.4　伪文件系统的含义 ······ 78

7.5　SELinux 相关的伪文件系统 ······ 78

7.5.1　selinuxfs ······ 78

7.5.2　proc ······ 80

7.6　总结 ······ 80

7.7　参考资料 ······ 81

习题 ······ 81

第 8 章　SMACK ······ 82

8.1　历史 ······ 82

8.2　概述 ······ 83

8.3　工作机制 ······ 84

8.3.1　操作许可 ······ 84

8.3.2　类型转换 ······ 84

8.4　扩展属性 ······ 86

8.5　伪文件系统 ······ 86

8.5.1　策略相关文件 ······ 87

8.5.2　网络标签相关文件 ······ 89

8.5.3　其他文件 ······ 90

8.6　网络标签 ······ 91

8.7　总结 ······ 91

8.8　参考资料 ······ 91

习题 ······ 91

第 9 章　Tomoyo ······ 92

9.1　简介 ······ 92

9.1.1　基于路径的安全 ······ 92

9.1.2　粒度管理 ······ 93

9.1.3　用户态工具 ······ 93

9.1.4　三个分支 ······ 93

9.2　机制 ······ 93

9.2.1　操作许可 ······ 94

9.2.2　类型和域 ······ 94

9.3　策略 ······ 95

9.3.1　域策略 ······ 95

9.3.2　异常 ······ 96

9.3.3　轮廓 ······ 99

9.4　伪文件系统 ······ 102

9.5　总结 ······ 103

9.6　参考资料 ······ 103

习题 ······ 103

第 10 章　AppArmor ······ 104

10.1　简介 ······ 104

10.2　机制 ······ 104

10.2.1　操作许可 ······ 104

10.2.2　域间转换 ······ 107

10.3　策略语言 ······ 108

10.4　模式 ······ 110

10.5　伪文件系统 ······ 110

10.5.1　proc 文件系统 ······ 110

10.5.2　sys 文件系统 ······ 111

10.5.3　securityfs 文件系统 ······ 112

10.6　总结 ······ 113

10.7　参考资料 ······ 113

习题 ······ 113

第 11 章　Yama ······ 114

11.1　简介 ······ 114

11.2　机制 ······ 114

11.3　伪文件系统 ······ 116

11.4　嵌套使用 ······ 116

11.5　总结 ······ 117

11.6 参考资料 …… 117
习题 …… 117

第三部分 完整性保护

第 12 章 IMA/EVM …… 119
12.1 简介 …… 119
12.1.1 可信计算 …… 119
12.1.2 完整性 …… 120
12.1.3 哈希 …… 121
12.1.4 IMA/EVM …… 121
12.2 架构 …… 122
12.2.1 钩子 …… 122
12.2.2 策略 …… 122
12.2.3 扩展属性 …… 123
12.2.4 密钥 …… 123
12.2.5 用户态工具 …… 124
12.3 伪文件系统 …… 126
12.4 命令行参数 …… 127
12.5 总结 …… 128
12.6 参考资料 …… 128
习题 …… 128
第 13 章 dm-verity …… 129
13.1 Device Mapper …… 129
13.2 dm-verity 简介 …… 130
13.3 代码分析 …… 131
13.3.1 概况 …… 131
13.3.2 映射函数（verity_map） …… 132
13.3.3 构造函数（verity_ctr） …… 137
13.4 总结 …… 138
13.5 参考资料 …… 138
习题 …… 138

第四部分 审计和日志

第 14 章 审计（audit） …… 140
14.1 简介 …… 140
14.1.1 审计和日志 …… 140
14.1.2 概貌 …… 140
14.2 架构 …… 141
14.2.1 四个消息来源 …… 141
14.2.2 规则列表 …… 147
14.2.3 对文件的审计 …… 150
14.3 接口 …… 153
14.4 规则 …… 155
14.5 总结 …… 157
14.6 参考资料 …… 157
第 15 章 syslog …… 158
15.1 简介 …… 158
15.2 日志缓冲 …… 158
15.3 读取日志 …… 159
15.4 netconsole …… 160
15.5 参考资料 …… 160
习题 …… 160

第五部分 加密

第 16 章 密钥管理 …… 162
16.1 简介 …… 162
16.2 架构 …… 162
16.2.1 数据结构 …… 162
16.2.2 生命周期 …… 164
16.2.3 类型 …… 168
16.2.4 系统调用 …… 171
16.2.5 访问类型 …… 174
16.3 伪文件系统 …… 175
16.4 总结 …… 175
16.5 参考资料 …… 175
第 17 章 eCryptfs …… 176
17.1 简介 …… 176
17.2 文件格式 …… 176
17.3 挂载参数 …… 179
17.4 设备文件 …… 180

17.5 用户态工具 …… 180
17.6 总结 …… 185
17.7 参考资料 …… 185
第 18 章 dm-crypt …… 186
18.1 简介 …… 186
18.2 架构 …… 186
18.2.1 两个队列（queue） …… 186
18.2.2 五个参数 …… 189
18.3 总结 …… 193
18.4 参考资料 …… 193
第 19 章 LUKS …… 194
19.1 简介 …… 194
19.2 布局 …… 194
19.3 操作 …… 195
19.4 总结 …… 196
19.5 参考资料 …… 196

第六部分 其他

第 20 章 namespace …… 198
20.1 引言 …… 198
20.1.1 容器与监狱 …… 198
20.1.2 chroot()与 pivot_root() …… 198
20.2 机制 …… 202
20.2.1 挂载命名空间 …… 203
20.2.2 进程间通信命名空间 …… 205
20.2.3 UNIX 分时命名空间 …… 207
20.2.4 进程号命名空间 …… 208
20.2.5 网络命名空间 …… 212
20.2.6 用户命名空间 …… 212
20.2.7 进程数据结构 …… 215
20.3 伪文件系统 …… 216
20.4 系统调用 …… 216
20.4.1 clone …… 216
20.4.2 unshare …… 217
20.4.3 setns …… 217
20.5 总结 …… 217
20.6 参考资料 …… 218
习题 …… 218
第 21 章 cgroup …… 219
21.1 简介 …… 219
21.1.1 一种安全攻击 …… 219
21.1.2 对策 …… 220
21.1.3 历史 …… 220
21.1.4 用法举例 …… 221
21.2 架构 …… 222
21.2.1 设计原则 …… 222
21.2.2 代码分析 …… 223
21.3 伪文件系统 …… 229
21.4 总结 …… 231
21.5 参考资料 …… 232
第 22 章 seccomp …… 233
22.1 简介 …… 233
22.2 架构 …… 233
22.2.1 进程数据结构 …… 233
22.2.2 模式 …… 234
22.2.3 内核中的虚拟机 …… 234
22.2.4 工作原理 …… 235
22.3 内核接口 …… 236
22.3.1 系统调用 …… 236
22.3.2 伪文件 …… 236
22.4 总结 …… 237
22.5 参考资料 …… 237
第 23 章 ASLR …… 238
23.1 简介 …… 238
23.2 内存布局 …… 239
23.2.1 伪文件/proc/[pid]/maps …… 239
23.2.2 ELF 文件格式 …… 240
23.2.3 动态链接的可执行文件 …… 241
23.2.4 静态链接的可执行文件 …… 243
23.2.5 位置无关的可执行文件 …… 244
23.3 工作原理 …… 245
23.4 内核接口 …… 246
23.5 总结 …… 246
23.6 参考资料 …… 246
附录 …… 247

第1章 导　言

1.1 什么是安全

安全是一个很难说清楚的概念。我们很难说明白它到底是什么，它到底包含什么。众多国家科研项目以安全为题，其背后大半是某些开源软件，隐含的逻辑是开源等于自主可控，因为自主所以安全。国外开源界有些人的观点是，因为是开源，代码暴露给无数双眼睛，所以有安全问题一定会被早早发现。但是，近一两年来，开源代码频频曝光重大漏洞，追根溯源，一些问题代码已经存在了几年甚至十几年。在发展迅猛的计算机领域，十几年已可称作“古时候”了。还有些观点将安全与漏洞挂钩，且不说究竟什么是漏洞，漏洞和代码缺陷（bug）的区别究竟是什么。先回忆一下我们的手机，在不到十年之前，还是功能机满天飞的时代，我们并不担心手机安全。为什么到了智能机时代，手机安全反而让老百姓耳熟能详了呢？是原先的功能机漏洞或代码缺陷少吗？

尽管很难说清楚，国际上对计算机安全还是勉强概括了三个特性：私密性（Confidentiality）、完整性（Integrity）、可用性（Availability），简写为 CIA。私密性概念比较直观，就是数据不被未授权的人看到，你肯定不希望你的电话号码、银行账户还有照片什么的让不认识的人看到。完整性是指存储或传输的信息不被篡改，你肯定不希望付款的时候输入 100 元，结果实际被划走了 1000 元。可用性是指，你的设备在你需要使用它的时候能够使用。你肯定不希望你的家人万分焦急之中因为拨不通你的电话而轻信了骗子的话，真的以为你发生了意外。

计算机系统应对安全挑战的办法大致有四种：隔离、控制、混淆、监视。计算机系统安全的设计者在系统的各个层级都发明了不同的技术来实现隔离，隔离的结果常常被称作“沙箱”。隔离是对外的，阻断内部和外部的交互。控制则是对内的，在计算机世界是通过系统代码内在的逻辑和安全策略来维护信息流动和信息改变。如用户 A 可不可以读取文件 a，用户 B 能不能改变文件 b 的内容，等等。混淆要达到的效果是，明明你可以接触到数据却无法还原信息。隐藏一片树叶的最好地点是在一堆树叶中。在计算机世界中，加密就是一种混淆。最后是监视，监视的作用是间接的。遍布城市街道各个角落的监控摄像头并不能阻止违法和犯罪。它们不会在你闯红灯时，播放语音提示你；它们也不会在歹徒持刀抢劫时，发射激光、电流或者别的什么。但是你可能会因为它们的存在而接到交通罚单，警察也会在破案时查阅监控录像。计算机系统中的日志和审计就是在做监视工作。

1.2 计算机系统安全的历史

现代意义的计算机诞生于第二次世界大战。那时的计算机真的就是一个巨大的计算机器，或者可以不太恭敬地叫它“超级算盘”。你会担心一个“超级算盘”的安全问题吗？当然不会。后来，技术的进步和普及让计算机不仅仅是“计算的机器”，而且是“信息处理的机器”。那么，怎样保证计算机中的信息的私密性和完整性呢？走在信息革命前面的美国首先遇到这个问题，

并试图解决它。在 1970 年前后，先后出现了两个安全模型：BLP 模型和 BIBA 模型。前者参考美国军方的保密原则，着力解决私密性；后者则着力解决完整性。BLP 模型可简化为两句话：禁止上读，禁止下写。数据被分级，下级部门不能读上级部门的数据，上级部门也不能把数据传递给下级部门。如此，某部门就只能读到本部门或级别低于本部门的数据，数据的私密性得到了保证。BIBA 模型也可简化为两句话：禁止上写，禁止下读。数据也是分级的。下级不能写上级的数据，上级不能读下级的数据。BIBA 模型是服务于完整性的。完整性的英文是 Integrity，朗文字典中的英文释义有两个：一、strength and firmness of character or principle; honesty; trustworthiness；二、a state of being whole and undivided; completeness。第一个意思更接近于汉语的“人品”，第二个意思是汉语的“完整性”。BIBA 模型反映的是 integrity 的第一个意思。怎么讲呢？禁止上写：“低贱”的人不能玷污“高贵”的人的数据；禁止下读：“高贵”的人也不要去读“低贱”的东西，降低自己的品味。这有点像印度古老的种姓制度。BLP 模型和 BIBA 模型本身都经过了数学证明，都是很严谨的，可惜的是它们的适应面有些窄，无法覆盖计算机系统信息处理的全部。

为了更有效地利用计算机，计算机操作系统步入分时多用户时代。许多人登录到一台主机（mainframe），张三是个程序员，李四是个文档管理员，王五是系统管理员。随之出现了基于角色的访问控制（Role-based Acess Control，RBAC），让用户分属于不同的角色，再基于角色赋予访问权限。当 PC 时代来临，计算机设备专属于某个人，系统中的所谓用户也背离了原有的含义。随便打开 Linux 系统上的/etc/passwd 文件，看看里面还有几个是真正的用户？因此，在 PC 中使用基于角色的访问控制就有些力不从心了。接下来诞生了另一个访问控制模型——类型增强（Type Enforcement，TE）。模型中控制的对象不再是人，或角色，而是进程。进程属于不同的类型，不同类型有不同的访问权限。

江湖中不仅有少林，还有武当。计算机系统安全的另一路人马在“可信”领域辛勤地耕耘着。他们希望计算机只做人预先定义好的工作，不会有意或无意地去做主人不希望的事情。如果做到了这一点，他们就认为计算机是“可信”的了。可信理论的背后是将人类社会的信任模型构建到计算机的世界中，甲信任乙，乙信任丙，于是甲信任丙；计算机固件信任加载器（boot loader），加载器信任操作系统，操作系统信任应用，于是应用是可信的。信任的度量是用完整性校验值和数字签名。这的确可以保证应用是由某个“正直”的人或公司开发的，但不能保证应用没有漏洞，不会被恶意利用。

1.3 计算机系统安全的现状

计算机系统安全的可悲之处在于，任何一个被用户广泛接受的操作系统在设计之初都没有把安全作为设计系统的目标，包括 UNIX。这一点可以从 UNIX 设计者之一 Dennis Ritchie 的论文[⊖]中看到。计算机系统安全的第二个可悲之处是，安全不仅仅是一个技术问题，它和管理维护紧密联系。在安全研究人员眼中，苹果的 iOS 并不比谷歌的 Android 安全，但是后者暴露的安全问题却多得多。计算机系统安全的第三个可悲之处在于，安全性和易用性总是矛盾的。没有哪个厂商会为了安全而牺牲市场份额，而用户在免费的诱惑下，也更愿意牺牲自己的隐私。

计算机在过去的几十年里迅猛发展，但是计算机安全并没有跟上时代的脚步。四十多年前

⊖ On the Security of Unix, Dennis M. Ritchie, ftp://ftp.club-ix.farlep.net/pub/doc/security/info/unix-security.ps.gz

试读结束：需要全本请在线购买： www.ertongbook.com

的 BLP 模型是成功的，它达成了预定的目标——将美国军方的安全原则移植到信息处理系统中，但是在随后的日子里，没有一个好的安全模型能覆盖计算机应用的方方面面。这换来 BLP 的设计者之一 Bell 的一声叹息[⊖]。

病毒与反病毒，漏洞与漏洞补丁，头痛医头，脚痛医脚，乐此不疲！

1.4　Linux 内核安全概貌

内核对于系统的重要性是不言而喻的。Linux 内核安全的开发开始得还是比较早的，约始于 20 世纪 90 年代中后期。经过近二十年的开发，Linux 内核中安全相关的模块还是很全面的，有用于强制访问控制的 LSM（Linux Security Module），有用于完整性保护的 IMA（Integrity Measurement Architecture）和 EVM（Extended Verification Module），有用于加密的密钥管理模块和加密算法库，还有日志和审计模块，以及一些零碎的安全增强特性。

说起来安全功能是很多的。但是问题也有，其一是应用问题，这些安全功能还是没有被广泛地应用起来。最典型的是 Linux 内核中基于能力的特权机制，时至今日，广大应用程序的开发者不仅没用它，甚至根本不知道它的存在！其二是整合问题。攻与防的区别在于，攻击只求一点突破即可，防守则要保证整条防线。内核各个安全模块散布于内核多个子系统之中，如何整合各个安全子模块来整体加固系统的安全是一个不小的挑战。

⊖ Looking Back at the Bell-La Padula Model, David Elliott Bell, http://www.acsac.org/2005/papers/Bell.pdf

第一部分　自主访问控制

1．访问

访问是什么？看一个例子，西游记第十七回的标题是："孙行者大闹黑风山　观世音收伏熊罴怪"。浓缩一下就是，孙行者闹黑风山，观世音收熊怪。由此可见访问包括三个要素，访问发起者——孙行者、观世音，访问动作——闹、收，被访问者——黑风山、熊怪。在计算机安全领域，访问发起者被称为主体，访问动作就是具体的操作，被访问者被称为客体。比如，进程A读文件a，进程A是主体，读是操作，文件a是客体。

计算机是人发明出来的一种机器。它是为人服务的。但是计算机的世界里没有人，只有代表用户执行任务的进程。程序是静态的，进程是动态的，进程是程序的一次运行。用户甲和用户乙运行同一个程序能做的事情却可能不同，比如普通用户运行 vi /etc/passwd 不能修改文件内容，而 root 用户运行 vi /etc/passwd 就可以修改。这种区别就是访问控制造成的。

本书中提及的用户，在大部分语境下是指代表用户执行任务的进程。因为，在计算机的世界里没有作为实体的人存在。

2．访问控制

访问控制就是对访问进行控制。比如，允许进程A读文件a，不允许进程B读文件b。要实现访问控制，需要两个东西，一个是标记，标记主体和客体，这样才有控制的对象；另一个是策略，允许某主体对某客体做什么。

3．自主访问控制

自主访问控制的自主是指使用计算机的人可以决定访问策略，比如规定某文件只能读不能写，制造出一种"只读"文件，防止文件内容被不小心更改。再比如，张三有个 mp3 文件，规定李四可以读，但赵五不可以读。

自主访问控制的优点是设计简单。缺点是安全性相对较差，用户往往不清楚潜在的安全问题。在个人电脑和个人移动终端系统上，用户的含义被异化，不再表示使用计算机的人而是表示应用，这个问题很突出。允许应用读通信录，可以吗？允许应用通过 Internet 发送数据，可以吗？如果既允许读通信录，又允许通过 Internet 发送数据呢？有些软件显然在滥用自主访问控制，界面上总是弹出菜单，询问人们是否允许这个，是否允许那个，结果就是训练出不看内容快速点击确定的人。

4．UNIX 的自主访问控制

UNIX 的自主访问控制的设计是简单而有效的。它分为两个部分，第一部分可以概括为进程操作文件。操作分三种：读、写、执行。在进程操作文件时，内核会检查进程有没有对文件的相应操作许可。第二部分可以概括为：拥有特权的进程可以做任何事情，内核不限制。特权机制实际上包含了两类行为，一类是超越第一部分的操作许可控制，比如 root 用户可以读或写任何文件。另一类是无法纳入上述"进程操作文件"模型之内的行为，比如重启动系统。

5．Linux 的自主访问控制

在自主访问控制上，Linux 对 UNIX 的扩展主要有两处，一是提供了访问控制列表（Access Control List），使得能够规定某一个用户或某一个组的操作许可；二是对特权操作细化，将原有属于根用户的特权细化为互不相关的三十几个能力。有些遗憾的是这两个扩展的接受程度不够理想，广大应用程序开发者和系统维护者对它们还不熟悉，还没有频繁地使用它们。

本书对于 Linux 沿袭 UNIX 的部分会叙述作 UNIX 如何如何，对于 Linux 特有的部分会叙述作 Linux 如何如何。

第 2 章　主体标记与进程凭证

2.1　进程凭证

没有标记就谈不上区分，没有区分就无从实施控制。本章介绍进程的标记，下一章介绍文件等客体的标记。

UNIX 是诞生于 20 世纪 70 年代的分时多任务多用户操作系统。当时的场景是许多用户同时登录到一台主机，运行多个各自的进程。因此，很自然的，UNIX 系统中进程的标记是基于用户的。在人类的世界中，人的标记是名字。相比字符串而言，计算机更擅长处理数字，UNIX 使用一个整数来标记运行进程的用户，这个整数被称作 user id，简写为 uid。人通常被分组，比如这几个人做研发工作，被分到研发组，那几个人做销售工作，被分到销售组。UNIX 用另一个整数来标记用户组，这个整数被称作 group id，简写为 gid。uid 和 gid 是包括 Linux 在内的所有类 UNIX 操作系统的自主访问控制的基础。

下面看一下 uid 和 gid 是如何记录在内核的进程控制结构之中的：

```
include/linux/sched.h
struct task_struct {
...
/* objective and real subjective task credentials (COW) */
const struct cred __rcu *real_cred;
/* effective (overridable) subjective task credentials (COW) */
const struct cred __rcu *cred;
...
}
```

Credential 的中文意思为凭证或通行证，本书用凭证这个词。进程的凭证中存储有和访问控制相关的成员。从上面代码可见，进程的控制结构中有两个凭证，一个叫 real_cred，另一个叫 cred。在内核代码注释中将 real_cred 称为客体（objective）凭证，将 cred 称为主体（subjective）凭证。进程是主体，在某些场景下又是客体。典型的场景是进程间发信号，进程 A 向进程 B 发送信号，进程 A 是主体，进程 B 就是客体。在大多数情况下，主体凭证和客体凭证的值是相同的，但在某些情况下内核代码会修改当前进程的主体凭证，以获得某种访问权限，待执行完任务后再将主体凭证改回原值。

下面看一下凭证的数据结构：

```
include/linux/cred.h
struct cred {
...
  kuid_t uid;/* real UID of the task */
  kgid_t gid;/* real GID of the task */
  kuid_t suid;/* saved UID of the task */
```

```
    kgid_t sgid;/* saved GID of the task */
    kuid_t euid;/* effective UID of the task */
    kgid_t egid;/* effective GID of the task */
    kuid_t fsuid;/* UID for VFS ops */
    kgid_t fsgid;/* GID for VFS ops */

    /* SUID-less security management */
  unsigned securebits;
    /* caps our children can inherit */
    kernel_cap_t cap_inheritable;
    /* caps we're permitted */
    kernel_cap_t cap_permitted;
    /* caps we can actually use */
    kernel_cap_t cap_effective;
    /* capability bounding set */
    kernel_cap_t cap_bset;


    #ifdef CONFIG_KEYS
    /* default keyring to attach requested keys to */
  unsigned char   jit_keyring;
    /* keyring inherited over fork */
  struct key __rcu *session_keyring;
    /* keyring private to this process */
  struct key      *process_keyring;
    /* keyring private to this thread */
  struct key      *thread_keyring;
    /* assumed request_key authority */
  struct key      *request_key_auth;
    #endif

    #ifdef CONFIG_SECURITY
  void          *security;     /* subjective LSM security */
    #endif

  ...
  }
```

进程凭证中不止有 id 相关的成员，还有能力集相关的成员、密钥串相关的成员和强制访问控制相关的成员。能力集相关内容在第 6 章介绍，密钥串相关内容在第 16 章介绍，强制访问控制则在第二部分介绍。

2.2 详述

2.2.1 uid 和 gid

单纯的 user id 和 group id 都好理解。不好理解的是进程凭证中有不止一个 uid 和 gid。

（1）uid

这是最早出现的 user id。有时也被称为 real uid，实际的 uid，简写为(r)uid。这个 uid 在资

源统计和资源分配中使用，比如限制某用户拥有的进程数量。

（2）euid

euid（effective uid）即有效 uid。在内核做特权判断时使用它。它的引入和提升权限有关[1]。此外，内核在做 ipc（进程间通信）和 key（密钥）的访问控制时也使用 euid。

（3）suid

suid 是“saved set user id”。euid 和特权有关，当 euid 为 0 时，进程就具有了超级用户的权限[2]，拥有了全部特权，在系统中没有做不了的事情。这有些危险。我们需要锋利的刀，但不用的时候希望把刀放入刀鞘。为了让进程不要总是具有全部特权，总能为所欲为，系统的设计者引入了 suid，用于暂存 euid 的值。euid 为 0 时做需要特权的操作，执行完操作，将 0 赋予 suid，euid 恢复为非 0 值，做普通的不需要特权的操作，需要特权时再将 suid 的值传给 euid。

（4）fsuid

fsuid 是“file system user id”。这个 uid 是 Linux 系统独有的。它用于在文件系统相关的访问控制中判断操作许可。

gid 和 uid 类似，也有(r)gid、egid、sgid、fsgid。此外，多出了一个补充组 id，Supplementary Group IDs。补充组 id 是一个数组，存有一组 group id，因为一个用户可以属于多个组。补充组 id 也用于访问控制权限检查。这点与 egid 相同。可以这样理解：在涉及权限的检查中，要判断 egid 和补充组 id 中的每一个 gid，涉及文件系统的操作还要加上 fsgid，只要有一个判断的结果是允许访问就允许访问。

gid 和 uid 的另一个区别是 group id 与特权无关。euid 为 0 的进程具备全部特权[3]。egid 或别的 gid 为 0，进程不会因此而具备特权。

2.2.2　系统调用

进程的控制结构和进程凭证都是内核中的数据结构，相应的数据对象都是被内核掌控的。用户态进程只能通过内核提供的接口来查看和修改进程凭证。Linux 内核提供了数个系统调用来查看和修改进程的凭证中的 uid 和 gid。这部分系统调用都要求进程只能修改自己的 uid 和 gid，不可以修改别的进程的。

先说和设置 user id 相关的系统调用：

```
int setuid(uid_t uid)
```

与字面意思相左，setuid 是用来设置 euid 的。如果调用进程具有 setuid 能力（一个特权），此调用会将(r)uid 和 suid 也一起设置，即(r)uid、euid、suid 的值将在系统调用后相同。

```
int seteuid(uid_t euid)
```

seteuid 也是用来设置 euid 的。seteuid 与 setuid 的区别在于 seteuid 不会设置(r)uid 和 suid。

```
int setreuid(uid_t ruid, uid_t euid)
```

[1] 参考第 6 章。

[2] 这只是一般情况，请参考第 6 章。

[3] 内核实际上判断的是 capabilities，不是 uid，请参考第 6 章。

setreuid 可以同时修改(r)uid 和 euid。提供“-1”作为参数表示维持原有值不变。在以下两个条件之一成立时，此系统调用也会修改 suid，让 suid 的值和系统调用后的 euid 值相同：

- 修改了(r)uid。
- 修改了 euid，并且 euid 的新值不等于系统调用前的(r)uid。

```
int setresuid(uid_t ruid, uid_t euid, uid_t suid)
```

setresuid 同时修改(r)uid、euid、suid。

```
int setfsuid(uid_t fsuid)
```

setfsuid 修改进程的 fsuid。在设置 euid 相关的系统调用中，内核代码会在设置 euid 的同时也设置 fsuid，让 fsuid 和 euid 的值相同。此系统调用专门设置 fsuid。

在设置 user id 的系统调用中，内核代码遵守了以下原则：

- 具备 setuid 特权的进程可以把(r)uid、euid、suid、fsuid 设置为任意值。
- 不具备 setuid 特权的进程只能将(r)uid、euid、suid 的值设置为现有的(r)uid、euid、或 suid 的值。以 euid 为例，euid 的新值只能是现在的(r)uid 的值、现在的 euid 的值或现在的 suid 的值。
- 不具备 setuid 特权的进程只能将 fsuid 的值置为现有的(r)uid、euid、suid、fsuid 的值之一。

组的 set 类系统调用和用户（user）类似，也有 setgid、setegid、setregid、setresgid、setfsgid。涉及的特权是 setgid。同 uid 类的调用非常类似，做个简单替换就可以了。比如 setgid 调用中，如果进程具有 setgid 特权，进程的(r)gid 和 egid 也被设置，隐含 fsgid 也会随着 egid 一起改变。

组 set 类系统调用还有一个：

```
int setgroups(size_t size, const gid_t *list)
```

此调用用于一次性赋值进程凭证中的补充组 id。因为补充组 id 是一个数组，而且其中的值不能限定只出现在(r)gid、egid、sgid 中，否则补充组 id 没有意义。所以这个系统调用需要特权 setgid。

与 set 类系统调用相对的是 get 类系统调用。

```
uid_t getuid(void)
```

总算和字面意思相符了，此系统调用取出进程的(r)uid。

```
uid_t geteuid(void)
```

取出进程的 euid。

```
int getresuid(uid_t *ruid, uid_t *euid, uid_t *suid)
```

取出进程的(r)uid、euid、suid。

有趣的是与 set 作比较，没有与 setreuid 和 setfsuid 相对应的 get 类系统调用。内核如此设

试读结束：需要全本请在线购买： www.ertongbook.com