二、实验名称

Project1 缓冲区溢出实验

三、实验目的

Buffer over flow 漏洞利用实践

四、实验内容

编写exploits攻击漏洞程序

五、实验结果

获取具有root权限的shell

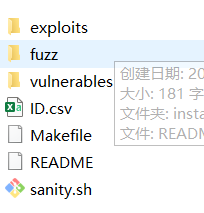
六、实验环境

Windows10

Oracle VM VirtualBox

Ubuntu16.04 i386

Sudo apt-get prelink

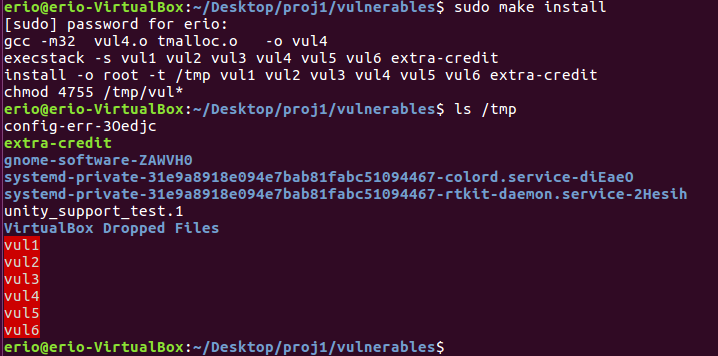


Exploits 攻击程序。Vulnerables 目标程序

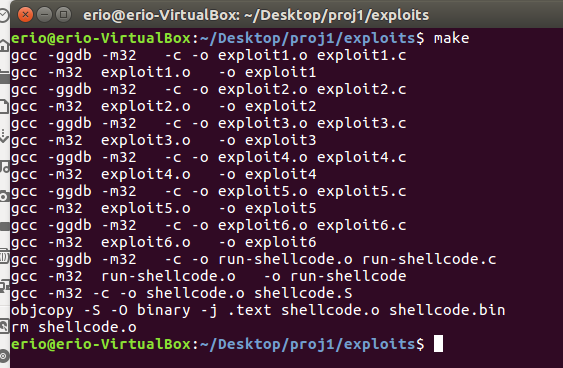
准备

Vulnerable 文件夹 编译，安装 Make Sudo make install

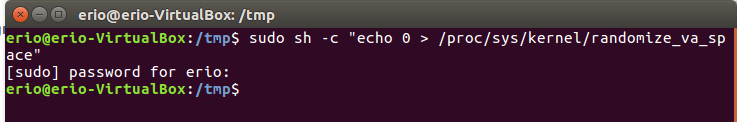
安装到了根目录 /tmp 文件夹。



Exploits 编译 Make



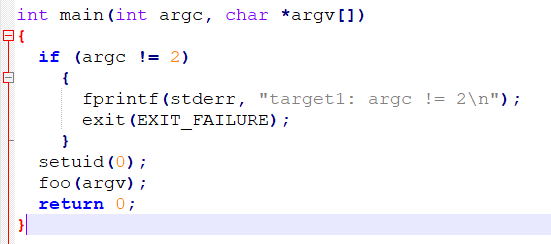
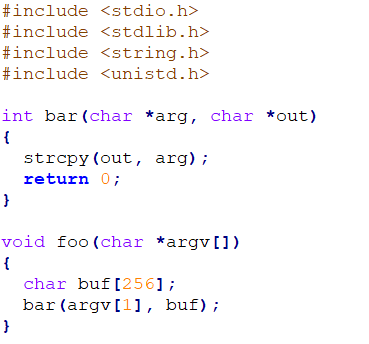
先关闭地址随机化 sudo sh -c "echo 0 > /proc/sys/kernel/randomize\_va\_space"



**七、实验步骤**

**1.vul[1-6]的简单描述**

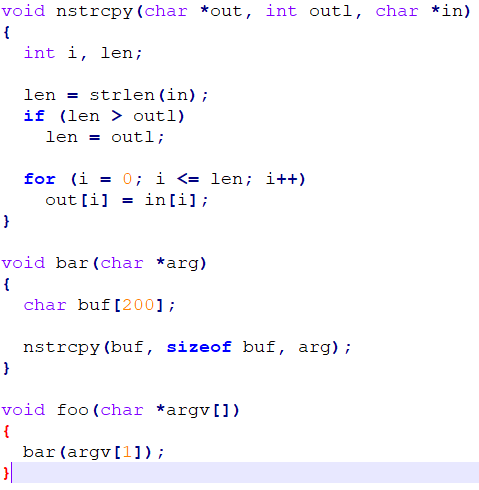
1.1 vul 1



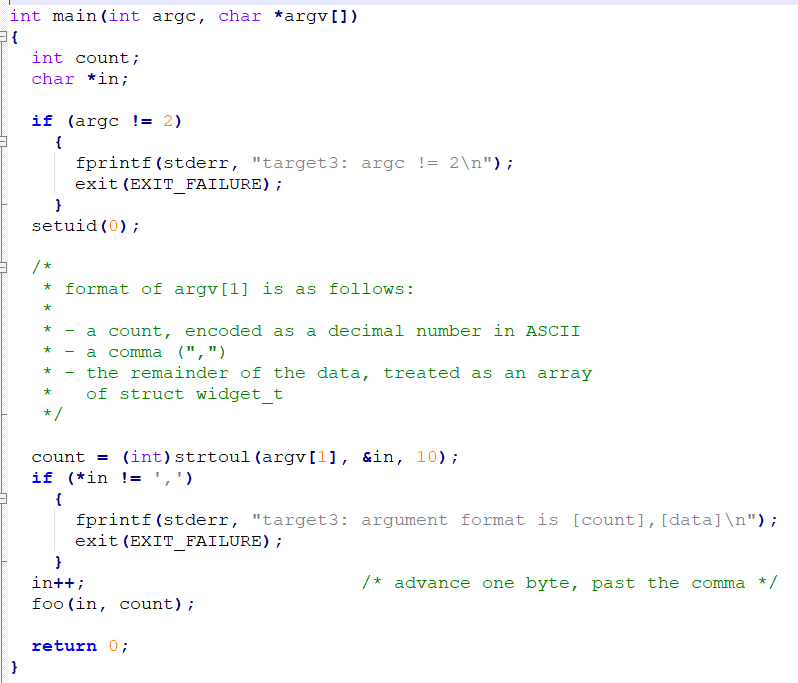
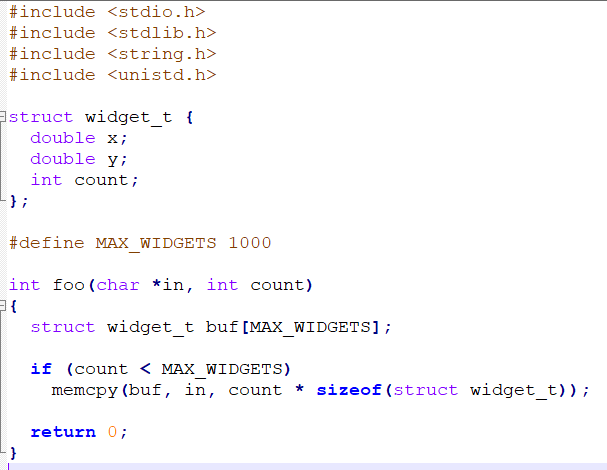
查看Vul1 源代码可知其中存在strcpy函数，当各种保护机制未开启时，容易利用strcpy实现缓冲区溢出。Vul 1 没有做边界的检查，直接拷贝argv[1]数组到buf数组，当argv[1]数组大小大于buf数组时，超出的字节会覆盖掉foo函数的ebp,返回地址等内容，当函数返回时读取返回地址，只要修改返回地址为shellcode的起始地址，就可以执行shellcode，进而获得shell。

1.2 vul2

下面只给出有漏洞部分的代码。Vul2问题为sizeof函数在计算时会考虑最后的/0占一字节而strlen不会，同时for循环中用的是i<=，导致虽然nstrcpy函数做了len和out1的比较，但是从argv拷贝到buf数组时，最多可以拷贝201字节，多拷贝的1字节覆盖掉foo函数ebp的最低字节。利用这一点可以实现溢出攻击。

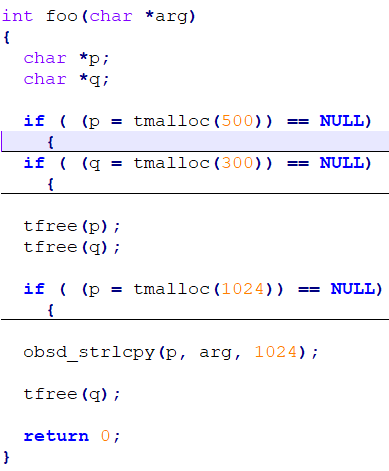
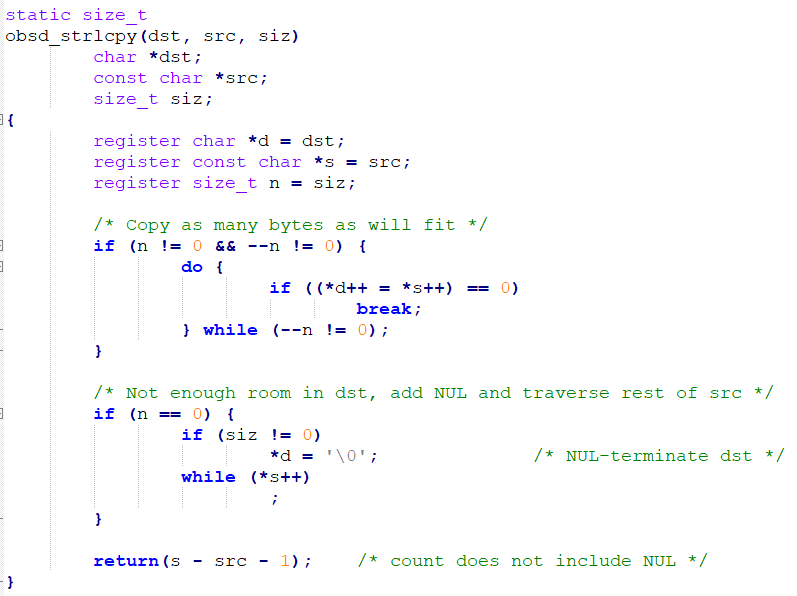


1.3 vul3



分析Vul3可知输入为整数count和字符串in。vul3中使用if判断来防溢出，问题在于一个足够小点负数可以满足 if 的条件进入memcpy函数，而负数\*sizeof(widget\_t)可能溢出产生正数。故要对vul3栈溢出攻击只需要构造合适的负数count产生溢出和字符串in覆盖buf然后覆盖ebp，ret即可。

1.4 vul4

观察foo函数，可见问题在于最后一次tfree(q)。由于之前已经tfree了q，现在相当于tfree一个不存在的空间，导致出错。

再看自定义的函数

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

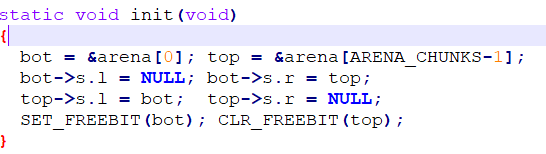
FROMCHUNK()函数为由CHUNK返回指针位置

ARENA\_CHUNKS 为CHUNK的数目

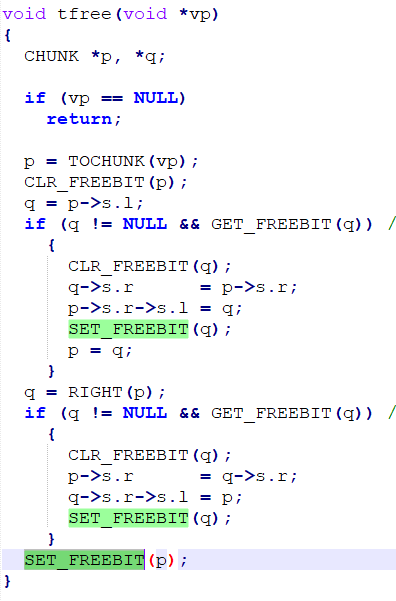
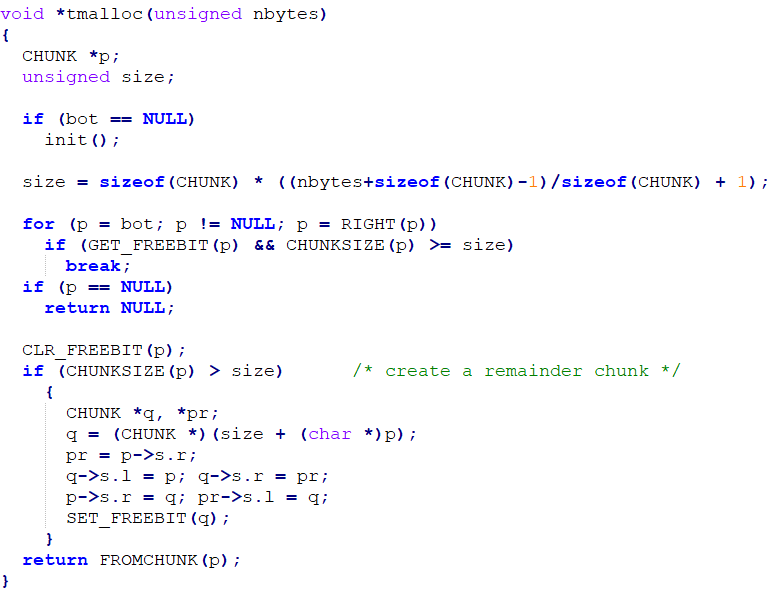
arena[]为 每个CHUNK的空间

bot 为空间的底部

top 为空间的顶部



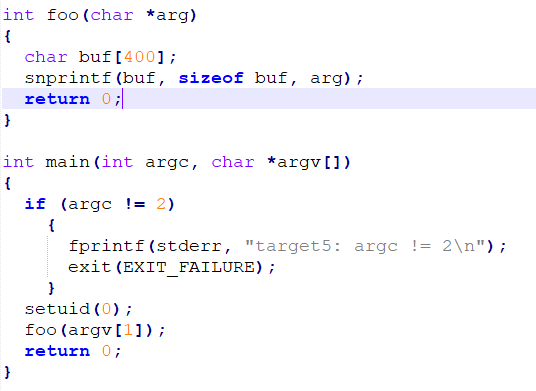
Init分配65536字节的chunk数组，bot是数组底部指针，top是数组顶部指针。



Tmalloc分配空间，遍历chunk双链表，找到free且足够大的chunk分配。返回连续的chunk的头部。先找到一个足够大的连续chunk链，chunk链头部标志占用。然后如果chunk链大于需要的空间，在中间截断（选一个块然后标志空闲）。

Tfree函数相当于先把左侧连续的占用的chunk free合并，再把右侧的free，合并。P代表了连续的空闲块的头部。

1.5 vul5



Snprintf函数，百度百科：

将可变个参数(...)按照format格式化成字符串，然后将其复制到str中。

(1) 如果格式化后的字符串长度 < size，则将此字符串全部复制到str中，并给其后添加一个字符串结束符('\0')；

(2) 如果格式化后的字符串长度 >= size，则只将其中的(size-1)个字符复制到str中，并给其后添加一个字符串结束符('\0')，返回值为欲写入的字符串长度。

查阅资料可知：

snprintf(buf, sizeof buf, arg);

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。

%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，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

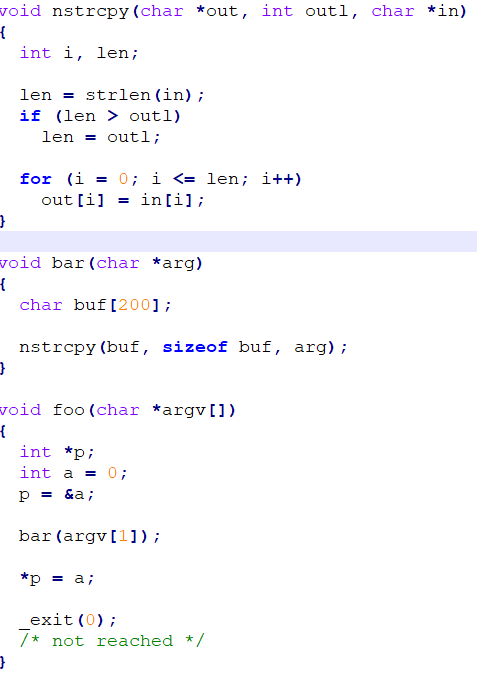
格式参考了：

<https://blog.csdn.net/u010517901/article/details/46486341>

<https://blog.csdn.net/qq_36779888/article/details/89684453>

故此次溢出的构造应该是构造合适的arg，从而构造合适的参数（%n），从而修改snprintf函数的返回地址，跳转到payload。

1.6 vul6



Vul6和vul2类似，可以覆盖foo的ebp的一字节。而这里有一个exit(0)，需要绕过。绕过思路见3.6exploit6

**2.shellcode（构造过程）**

原理是运行/bin/sh 来得到shell，构造过程是将具有运行/bin/sh的C代码转换成有相同功能的机器码。注意代码中用到 0 的地方改成用 xor eax,eax，这样可以避免复制字符串时遇到/0 中断。

下面的shellcode长度为45字节（不含/0）

/\*

\* 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";

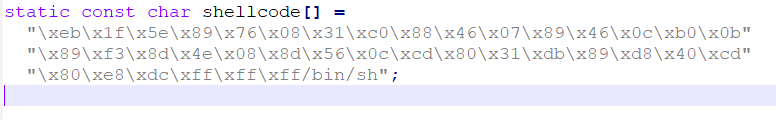
**3.exploit[1-6]攻击方式描述（攻击原理，payload构造方式，攻击过程描述）**

3.1 exploit1

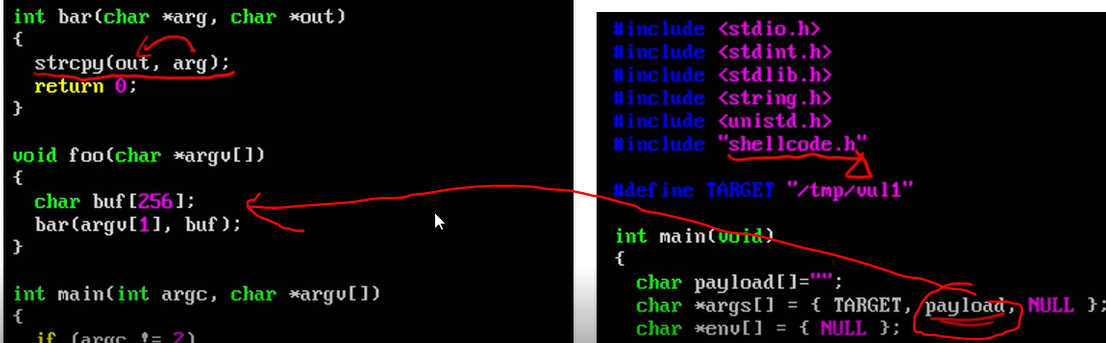
查看exploit1，框架已经搭建好，只需要替换其中的”hi there”为设计的payload

Payload由三部分组成，shellcode（获取shell的机器码），目标地址（返回地址设置为shellcode），nop（填充，使得能返回到shellcode）

下面是shellcode。执行exec(“/bin/sh”)的机器码。执行/bin/sh 可以获得root权限

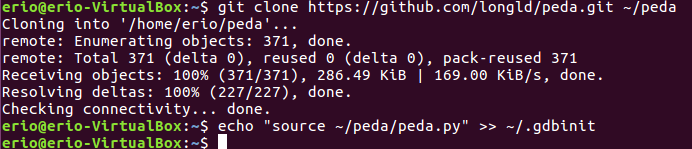


**如图，payload要覆盖调buf数组，长度要大于256，所以需要填充nop**

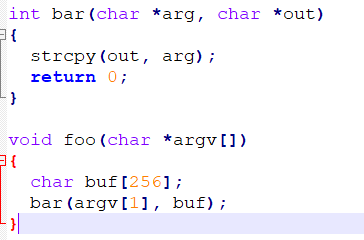


**以vul1为例介绍如何获得地址**

安装git-peda



其含有漏洞的代码如下。由源码可见Buf256字节，向上覆盖foo的返回地址还需要覆盖ebp（4字节），ret（4字节），所以payload需要256+8=264字节

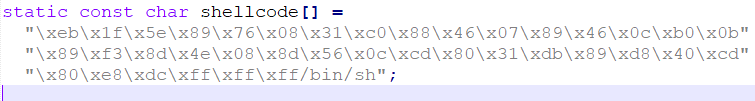


Exploits.c先填充shellcode，再填充nop，最后填充地址

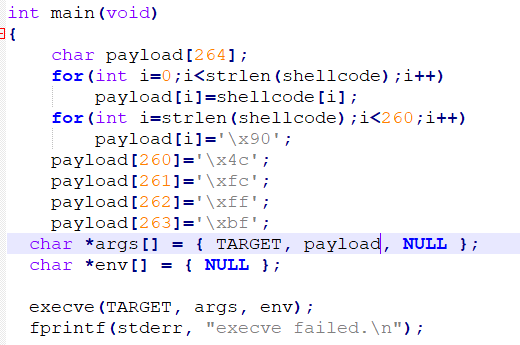
Shellcode 45字节（除最后的\0）

先填充shellcode再填充nop是因为内存中数组地址从低到高，栈方向从高到低，覆盖掉返回地址后返回到shellcode地址。

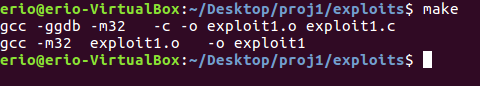
45+(256-45)\*NOP覆盖到buf，+4nop覆盖ebp，最后四字节覆盖ret



修改exploits1.c

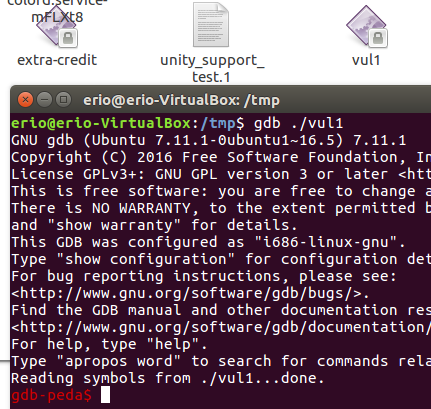


修改exploits1.c后重新make生成生成新的exploit可执行文件

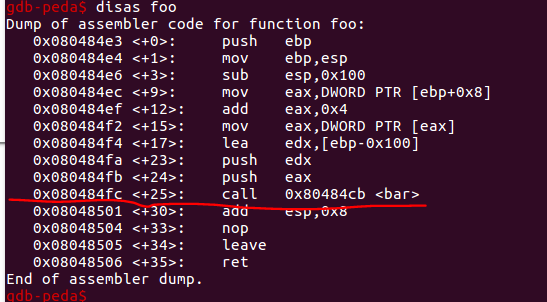


这里最后覆盖的地址0xbffffc4c获取方法见下。一开始随便填都可以。

在vul目录下，调试vul1 可执行文件。



含有strcpy的函数为bar（第一句直接调用），foo调用，先找到foo调用bar的地址

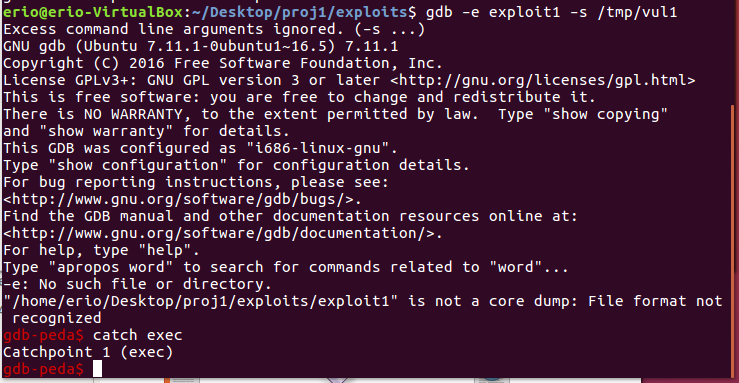


找到调用bar位置为0x080484fc

切换到exploitmulu，调试

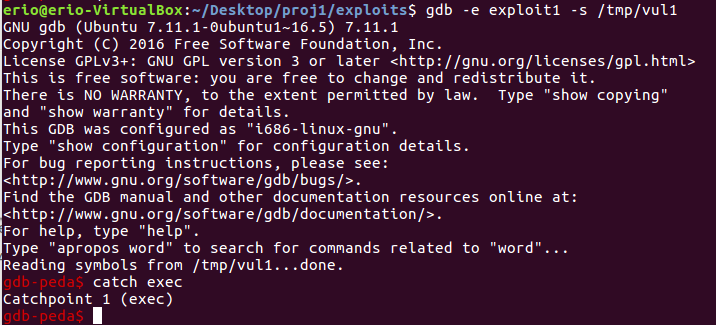
gdb –e exploit1 –s /tmp/vul1 执行exploit1，从指定文件vul1读取符号表

* catch exec 在exploit执行exec的位置设断点
* r
* b \*0x80484fc
* c
* ni
* find 0xd231c931



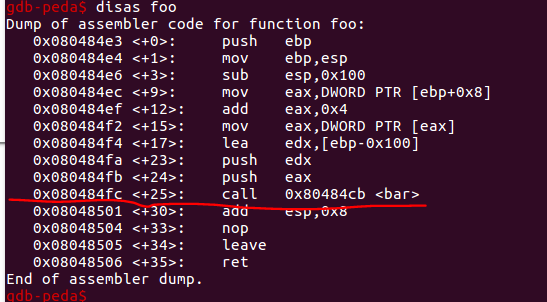
报错"/home/erio/Desktop/proj1/exploits/exploit1" is not a core dump: File format not recognized

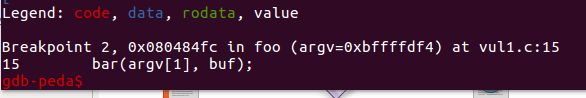
检查源文件meiwent。发现是输入的时候直接copy的，可能有干扰。手打gdb -e exploit1 -s /tmp/vul1就没问题了



Run

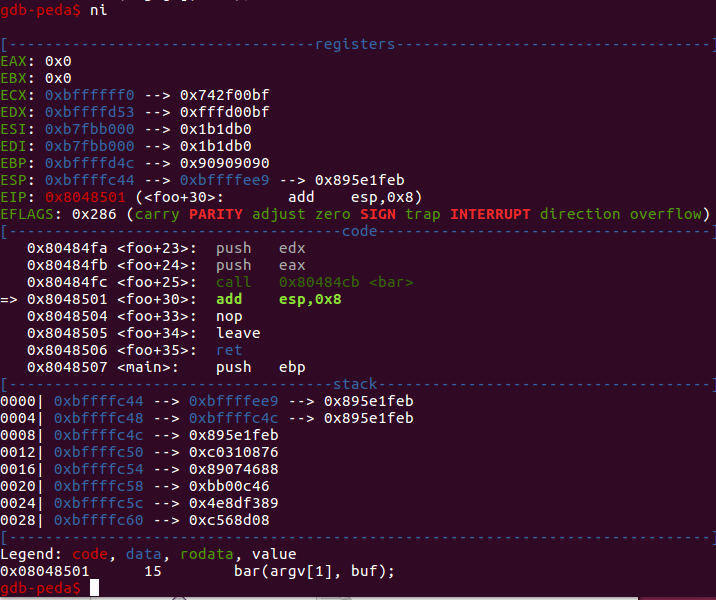
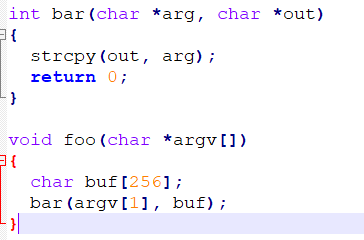
执行到exec刚刚知道bar的地址是 0x080484fc，在这里设置断点，c运行





此时暂停到foo调用bar的位置，因为strcpy就在bar第一句，只执行一步就可以得到地址。

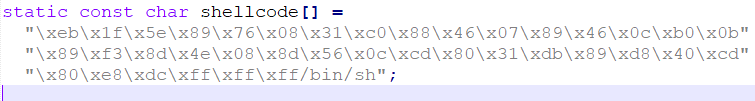
使用ni 进入bar函数内部

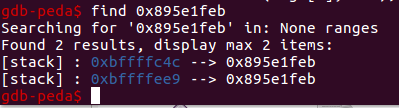


现在在strcpy内部

Payload中最开始是shellcode，shellicode前4字节为 0x895e1feb （注意小端）

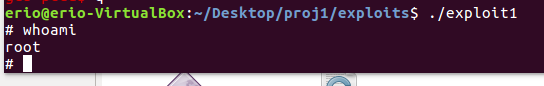
Find 0x895e1feb





找到地址在0xbffff1c4c，所以刚刚的payload最后四字节填 0xbfff1c4c。因为payload最后四字节溢出覆盖掉了foo的返回地址（不是bar的），strcpy是直接填了foo中buf。然后bar结束，栈回收只是esp，ebp改变，shellcode其实还在内存中，foo返回时eip跳到shellcode起始处，然后把shellcode的字符串当成机器码执行，eip++。

正确修改payload最后四位为0xbffff1c4c后，执行exploit1，效果如下，得到了root权限



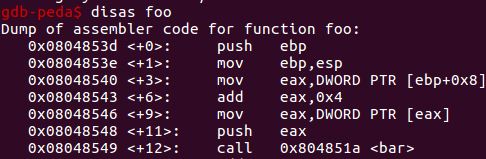
3.2 exploit2

根据1.2vul2的分析和内存栈的知识可知，内存中由高到低为foo的ebp和buf数组的200字节，构造的payload为201字节，payload[200]覆盖到ebp的低位。需要知道buf的起始地址和ebp的值。

Gdb调试vul2

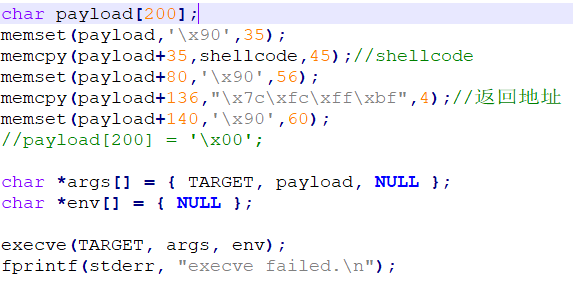
Gdb ./vul2

Disas foo



得到bar地址 0x804851a

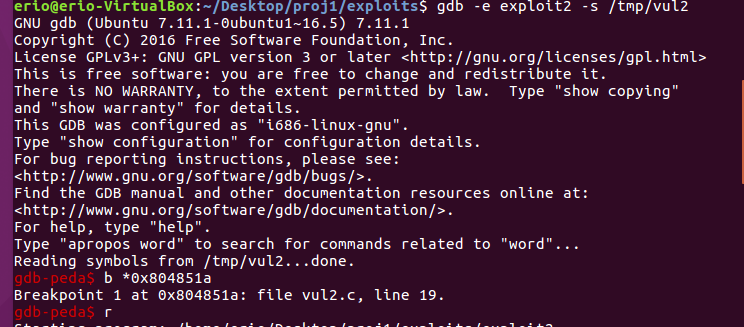
先随便填exploits的payload，先填200字节。调试exploit2

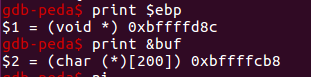


gdb -e exploit2 -s /tmp/vul2

b \*0x804851a

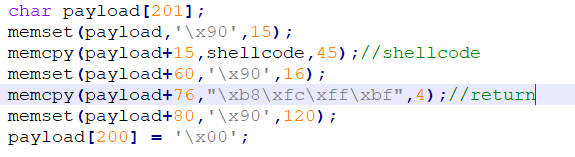
r



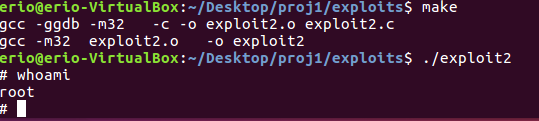


打印ebp和buf值，得到如图结果。Buf范围为 0xbffffcb8-0xbffffd80 200字节。Ebp为0xbffffd8c。如果覆盖掉ebp最后一字节为00，改变ebp为0xbffffd00，那么ret存放地址位0xbffffd04-0xbffffd08，注意小端。D04-cb8=4c=76字节。

故payload构造方式：15字节nop+45字节shellcode+16字节nop+4字节返回地址（buf起始地址）+120字节nop+0x00



运行



3.3 exploit3

由vul3可知buf数组20\*1000=20000字节

溢出部分如下，widget-t结构体20字节。Count\*20>20000即可。

**if (count < MAX\_WIDGETS)**

**memcpy(buf, in, count \* sizeof(struct widget\_t));**

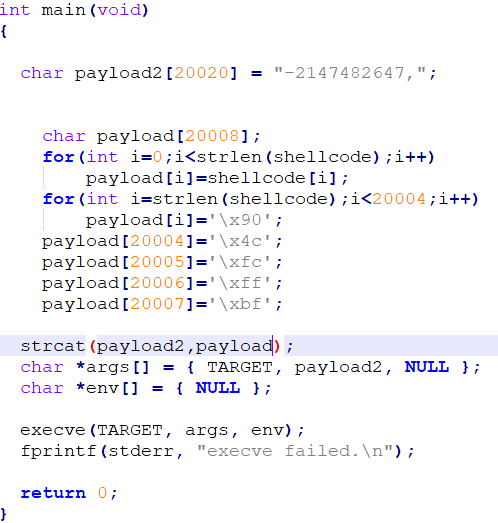
设count 为1001，符合1001\*20>20000. 1001 16进制为\X3e9。

负数符号位为1，得到\x800003e9。计算可知由于溢出，20\*\x3e9=20\*\x800003e9。\x800003e9在计算机中存储为补码2147482647.故count为 -2147482647.

然后构造payload。Buf大小为20000字节，故返回地址在20008字节处。覆盖该地址内存为buf的起始地址，然后buf覆盖为shellcode即可。

Payload=count（12字节）+45字节shellcode+19965nop+4nop（ebp）+ret（buf基址）。

Buf基址：先随便填payload，然后使用gdb调试得到;



Gdb vul3

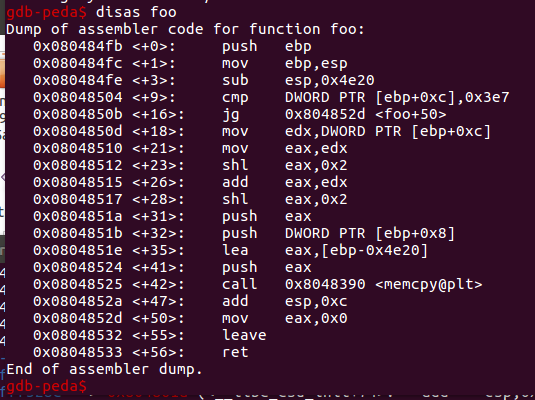
set args 3，666

disas foo

break \*0x0804850b

r

print &buf

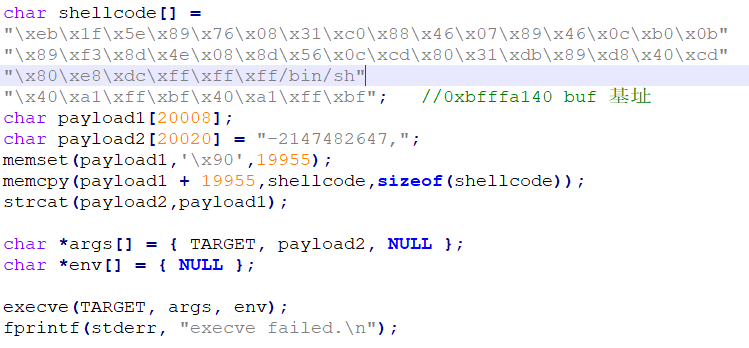




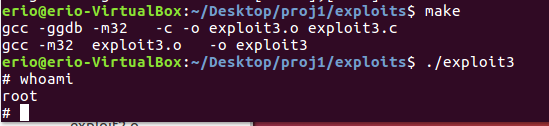


**得到buf基址 0xbfffa140**

填充payload最后4字节即可



**运行结果**



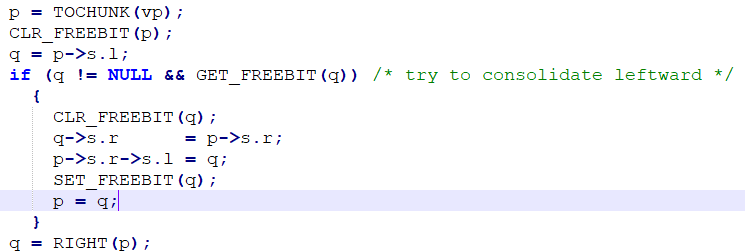
**3.4 exploit4**

先参考1.4中vul4的分析，

看foo中流程，可知先是p从65536的起始位置（BOT）分配了512字节空间（注意自定义的分配函数 size = sizeof(CHUNK) \* ((nbytes+sizeof(CHUNK)-1)/sizeof(CHUNK) + 1);），然后紧接着q分配312字节空间。然后tfree这两块。然后又从BOT分配1024给p。这样第二层分配给p的空间就覆盖第一次的p的空间。根据TOCHUNK（VP

）函数可知他获得的是VP-8字节的内容。

所以q的chunk结构（8字节）对应 obsd\_strlcpy(p, arg, 1024);的504-512字节。



利用tfree(p)的过程，将q的右节点设置为存放ret地址的地址，左节点设置为payload起始地址，这里设置为foo中p节点的地址（1024字节的起始位置）。

此时因为刚刚分配了1024空间，foo中p节点的标志位为0（占用），需要通过payload修改为1（空闲），也就是payload的4-8字节（对应foo中p的reght指针）最低位修改为1.

然后进入tfree第一个if，过程为

q->s.r = p->s.r; // 设置foo中p的s.r为foo的存放ret的地址

p->s.r->s.l = q;//将ret的地址的连续的8字节当做了chunk结构，恰好修改了ret地址位payload起始地址（foo中p的地址）

此时可以写出第一个payload=40\*nop+45shellcode+419\*nop+payload地址（foo中p的地址）+存放ret的地址（ebp+4）+512\*nop=1024字节。

然后修改foo中p的right指针标志位为1标记空闲（\*（int\*）（payload+4）=-1））设置为111111111.

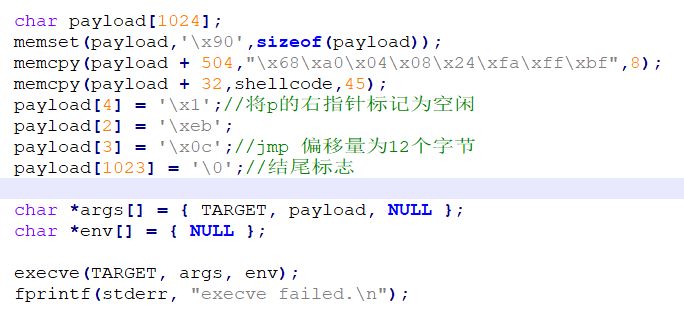
然后遇到了问题，payload跳转到其他位置。Gdb查看payload，发现有call指令。

然后修改为payload2

只修改foo中p的right的最低一位为1，即00000001.然后加一个jmp指令向后跳，跳过可能的call指令。Jmp对应机器码\xeb。注意jmp的机器码后需要有jmp的偏移量，可以设置为12。

现在只需要获得foo中p的地址和存放ret的地址。存放ret的地址可以先print ebp然后+4就得到

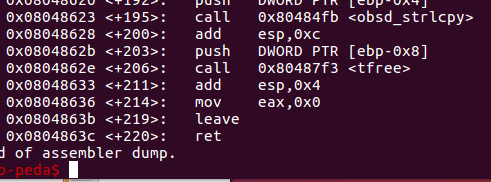
先按照上述格式填写payload，地址先随便填



Gdb /tmp/vul4

Disas foo

得到最后一个tfree的地址 0x0804862e



Gdb -e exploit4 -s /tmp/vul4

Catch exec

R

Break foo

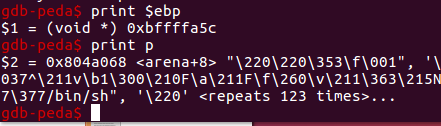
C

Ni

Break \*0x804862e

Print $ebp

Print p

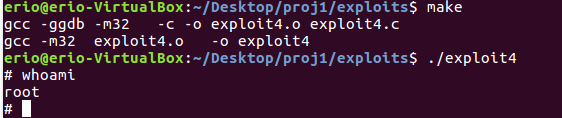


所以payload中地址：

P=0x804a068

Ret的地址=0xbffffa5c+4=0xbffffa60

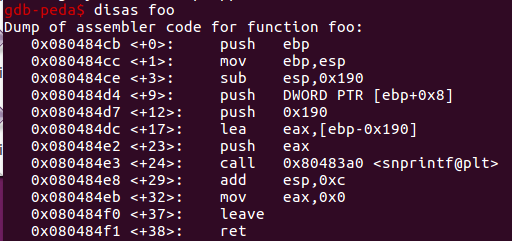
运行结果



**3.5 exploit5**

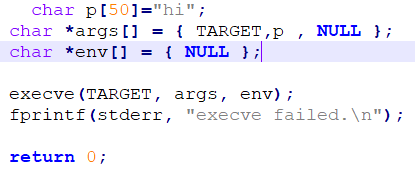
**Gdb /tmp/vul5**

**Disas foo**



**0x80484cb 0x80484e3**

**先构造一个简单的payload，用做测试**



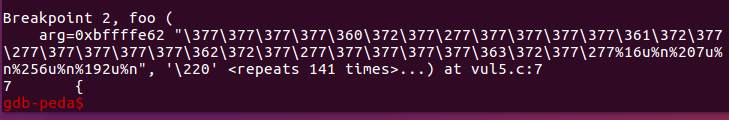
**gdb -e exploit5 -s /tmp/vul5**

**catch exec**

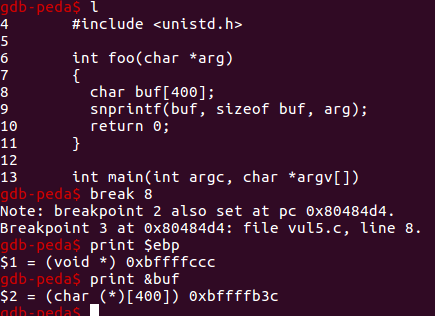
**r**

**break \*0x80484cb**

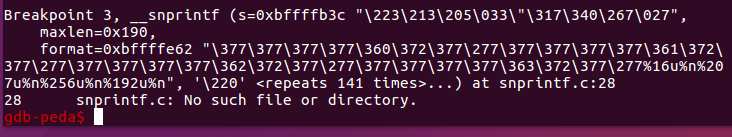
**c**



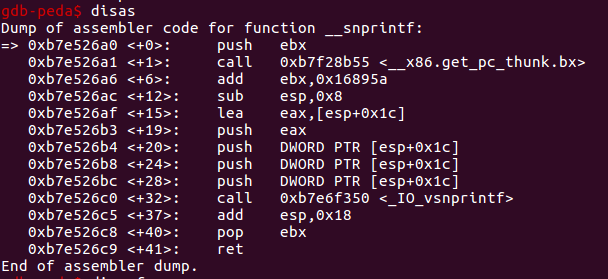
**此时刚刚进入foo，ebp，buf地址分别为**



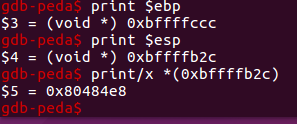
B snprintf

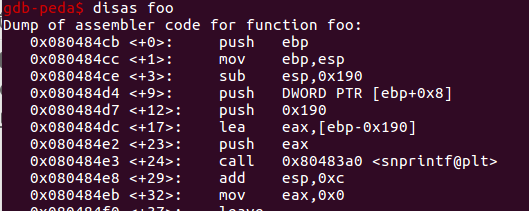


**此时disas查看snprintf的汇编**



查看此时的，ebp，esp及其内容，可见snprintf没有改变ebp，而esp（0xbffffb2c）指向的是0x80484e8，是snprintf的返回地址(下一条指令)，所以我们只要构造合适的payload，覆盖**0xbffffb2c中的返回地址**，使其指向我们的**payload**即可。



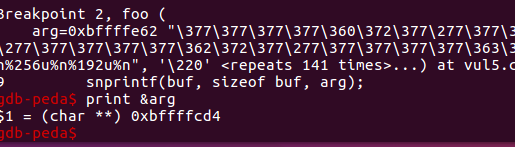
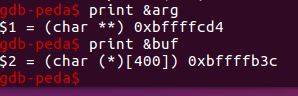


所以payload需要使用合适的参数修改esp（0xbffffb2c）的值。能修改的参数只有%n，写入的数据是已经输出的字符数，所以只需要构造合适的数目的字符就可以修改为指定地址。（比如40个字符，就写入0x28）

Vul5中，Snprintf用到%u参数时不断的修改buf，所以返回的地址不能是buf，那么只能是arg。

查看arg的地址：重新gdb调试，在foo break查看

为 0xbffffcd4，而buf地址在0xbffffb3c，相差408字节，为了不覆盖到arg，payload需要不超过408字节。（这里我觉得其实snprintf限制了不超过buf的400字节了）

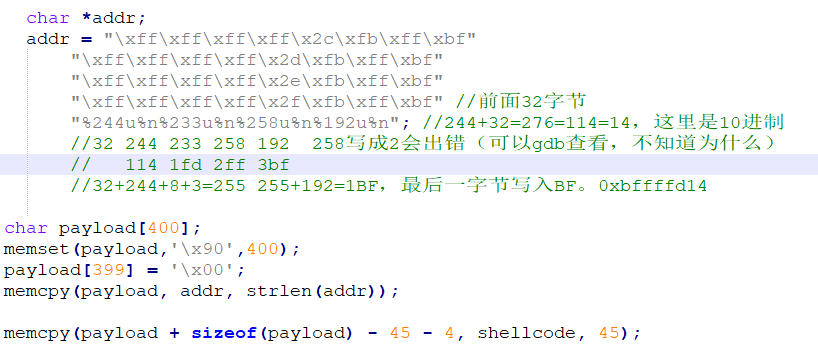


考虑到添加一些nop，最终返回地址可以是&arg+0x40=0xbffffd14

字符串长度不应该过长，所以每次修改esp指向的一字节。比如往esp（0xbffffb2c）写入0xbffffcf4的f4，需要有16\*f+4=244个字符

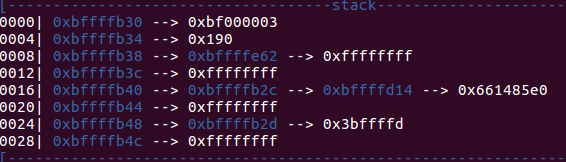
写fc需要252个字符，ff需要255字符，最后的bf，需要191个字符

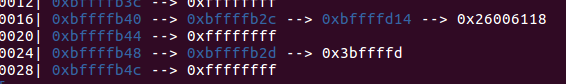
修改地址的payload部分为



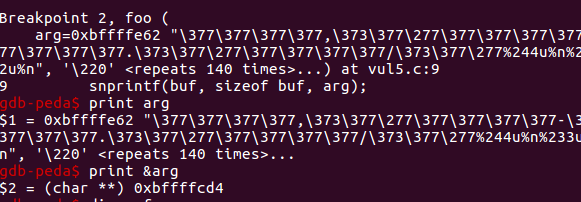
然后填nop和shellcode即可

然后这样出了问题，总是跳到一个随机的地址

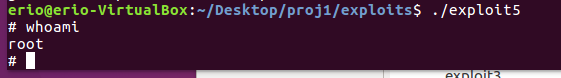




Gdb调试，发现是arg的地址出错。是0xbffffe62



目标地址改为 0xbffffea2即可



**3.6 exploit6**

由1.6vul6可知，除了覆盖ebp外，还需要绕过exit(0)。

Gdb /tmp/vul6

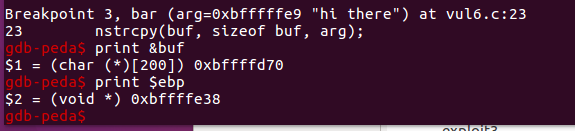
Set args 123131

B foo

C

B bar

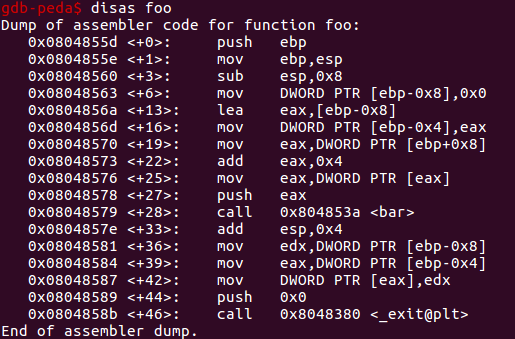
可知ebp和buf分别的值。其中buf范围为0xbffffd70-fe38，如果修改ebp最后为00，0xbffffe00在buf当中



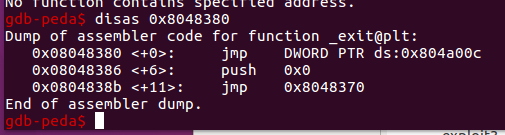
而已知p指针在ebp+4，a在ebp+8.

继续调试

Disas foo，可见 0x804858b，调用0x8048980的exit



Disas 0x8048380。



可见exit先是jmp到0x804a00c。

所以绕过的思路为：修改p，指向0x804a00c，然后修改0x804a00c的内容为shellcode地址。而shellcode地址其实就是buf地址（payload已经copy了）。

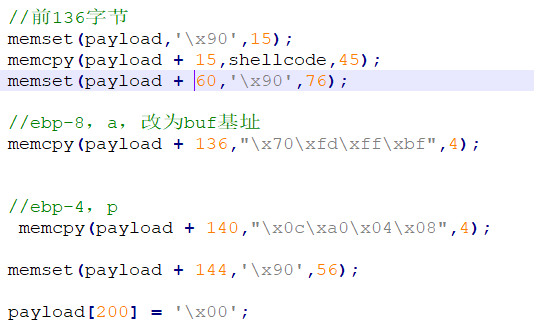
上面已知buf地址 0xbffffd70，ebp0xbffffe38，由vul2可知ebp修改为0xbfffef00后，p和a相对于ebp的位置不变。于是修改ebp-[0-3]就相当于修改p，ebp-[4-7]的位置就修改了a的值。

然后程序foo正常执行，exit->0x804a00c->a->shellcode。

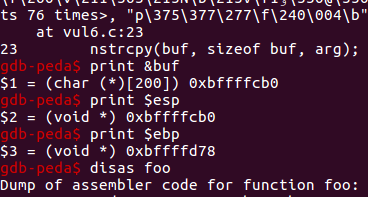
这里ebp修改后距离buf 144字节，p的位置为相对140-143字节，a为139-136字节

注意和vul2的区别在于p，a相对ebp的位置是在低地址

Payload：

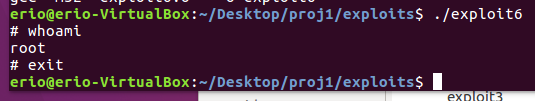


这里发现刚刚调试的时候用的payload是”hi there”，改为201字节的payload后有所更改



修改相应地址即可

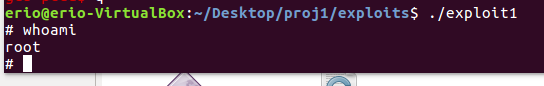




八、实验结果及分析

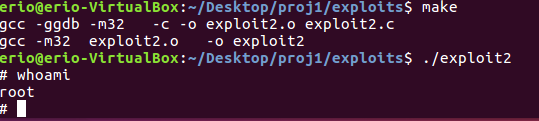
1 vul1

执行exploit1，获得root shell exit退出



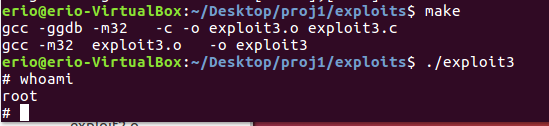
2 vul2

执行exploit2，获得root shell



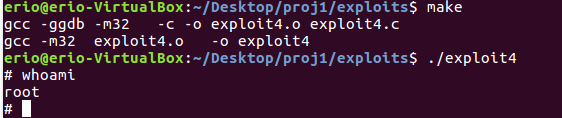
3 vul3

执行exploit3，获得root shell

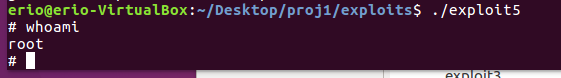


4 vul4

执行exploit4，获得root shell



5 vul5



6 vul6

