**武汉大学国家网络安全学院**

实 验 报 告

**课 程 名 称： 网络安全**

**实 验 名 称： 缓冲区溢出**

**指 导 老 师：**

**学 生 学 号：**

**学 生 姓 名：**

**完 成 日 期： 2022.04.17**

目录

[1 shellcode构造 3](#_Toc101114578)

[2 漏洞代码与攻击描述 5](#_Toc101114579)

[2.1 vul1与exploit1 5](#_Toc101114580)

[2.1.1 vul1描述 5](#_Toc101114581)

[2.1.2 exploit1.c攻击原理 6](#_Toc101114582)

[2.1.3 payload构造方式 7](#_Toc101114583)

[2.1.4攻击过程描述 9](#_Toc101114584)

[2.2 vul2与exploit2 10](#_Toc101114585)

[2.2.1 vul2描述 10](#_Toc101114586)

[2.2.2 exploit2.c攻击原理 10](#_Toc101114587)

[2.2.3 payload构造方式 11](#_Toc101114588)

[2.2.4攻击过程描述 12](#_Toc101114589)

[2.3 vul3与exploit3 13](#_Toc101114590)

[2.3.1 vul3描述 13](#_Toc101114591)

[2.3.2 exploit3.c攻击原理 13](#_Toc101114592)

[2.3.3 payload构造方式 14](#_Toc101114593)

[2.3.4攻击过程描述 15](#_Toc101114594)

[2.4 vul4与exploit4 16](#_Toc101114595)

[2.4.1 vul4描述 16](#_Toc101114596)

[2.4.2 exploit4.c攻击原理 19](#_Toc101114597)

[2.4.3 payload构造方式 20](#_Toc101114598)

[2.4.4攻击过程描述 23](#_Toc101114599)

[2.5 vul5与exploit5 24](#_Toc101114600)

[2.5.1 vul5描述 24](#_Toc101114601)

[2.5.2 exploit5.c攻击原理 24](#_Toc101114602)

[2.5.3 payload构造方式 26](#_Toc101114603)

[2.5.4攻击过程描述 27](#_Toc101114604)

[2.6 vul6与exploit6 28](#_Toc101114605)

[2.6.1 vul6描述 28](#_Toc101114606)

[2.6.2 exploit6.c攻击原理 28](#_Toc101114607)

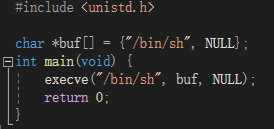
[2.6.3 payload构造方式 30](#_Toc101114608)

[2.6.4攻击过程描述 31](#_Toc101114609)

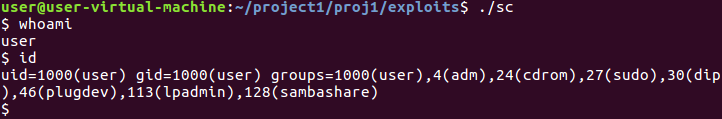
# 1 shellcode构造

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

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

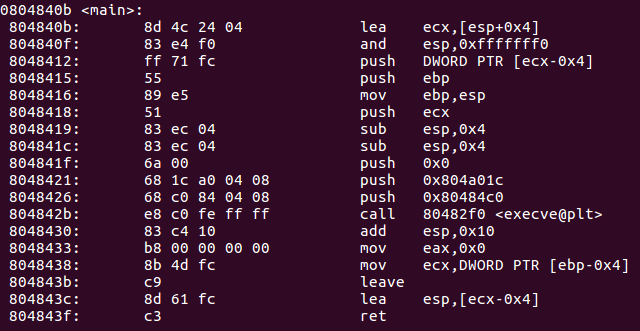


编译执行程序，成功得到shell，因为该程序并没有设置为root权限，所以得到一个普通用户的shell。

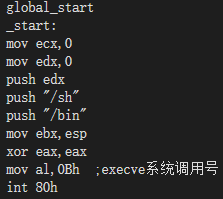


使用objdump工具反汇编程序得到main函数的汇编代码，可以参考该汇编代码来编写shellcode的汇编代码。

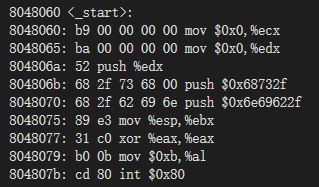
objdump -d sc -M intel



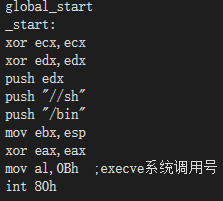
调用execve函数可以通过在汇编代码中指定execve的系统调用号实现，因此保留getshell功能且更为精炼的汇编代码如下所示。



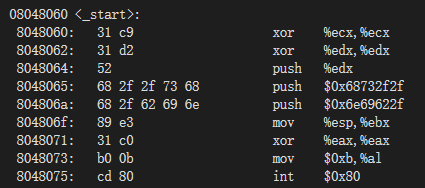
该汇编代码对应的机器码如下所示，其中可以看到机器码中存在许多\x00字节，shellcode中出现\x00字节一般会被截断，\x00字节表示NULL，输入函数检测到NULL就会返回，因此需要避免\x00字节的出现。



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



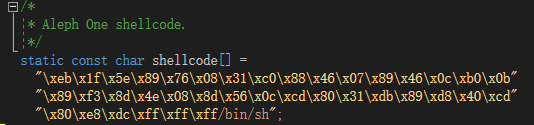
该汇编代码对应的机器码如下所示，其中已经不存在\x00字节。



最后将这些机器码提取出来，按照字节顺序构成shellcode。



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



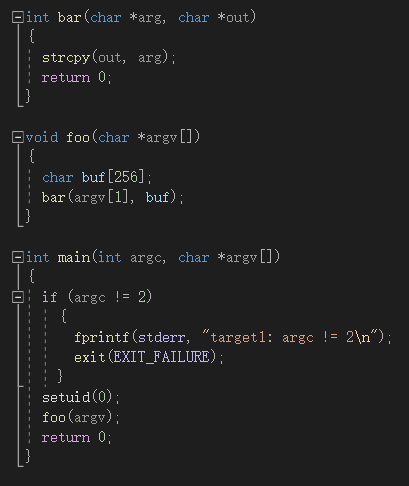
# 2 漏洞代码与攻击描述

## 2.1 vul1与exploit1

### 2.1.1 vul1描述

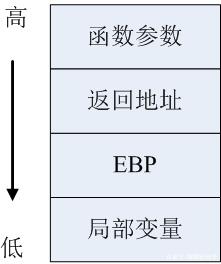
漏洞程序vul1要求argc的值等于2，即除了该程序路径外，还需要在命令行中输入一个参数。然后将该参数，也就是argv[1]的内容通过strcpy函数拷贝到缓冲区buf中。因为strcpy函数不会检查参数的长度是否符合要求，所以缓冲区buf中的数据是可以被控制的，并且缓冲区buf存在于函数foo的栈帧中，因此函数foo的返回地址可以被覆盖，该程序存在栈溢出漏洞。

漏洞代码vul1.c中主要内容如下图所示。



### 2.1.2 exploit1.c攻击原理

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



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

### 2.1.3 payload构造方式

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

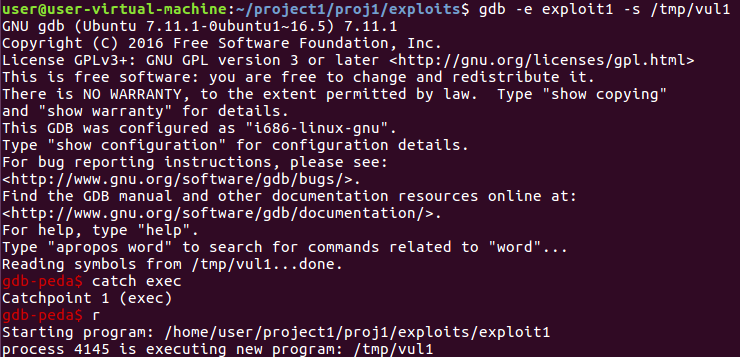
exploit1需要的payload的主要组成部分是shellcode和shellcode的入口地址，现在已经获得shellcode，而入口地址可以通过gdb工具调试得到。为了使得调试时内存的分配情况与执行exploit1程序时一致，需要使用以下命令启动gdb进行调试。

gdb -e exploit1 -s /tmp/vul1

然后使用gdb命令catch exec捕获exec事件，再开始运行程序。

gdb-peda$ catch exec

gdb-peda$ r

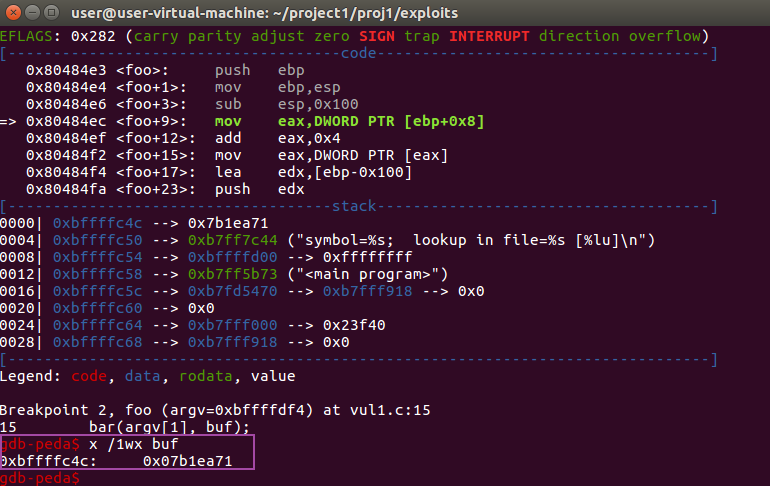


然后在foo函数处设置断点，并将程序运行至断点处。此时已经为缓冲区buf在栈中分配了内存空间，查看缓冲区buf的起始地址，值为0xbffffc4c。

gdb-peda$ b foo

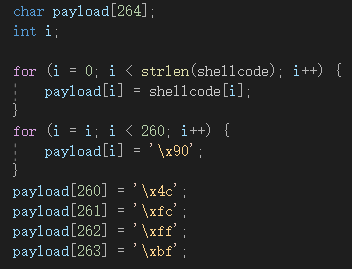
gdb-peda$ c

gdb-peda$ x /1wx buf



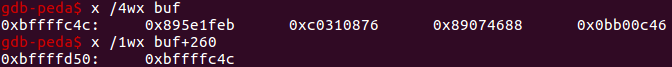
阅读vul1.c可以发现缓冲区buf的大小是256个字节，再结合栈结构可以得出返回地址存储于以buf+260为起始地址的4个字节处，buf+256为起始地址的4个字节存储的是旧寄存器EBP的值。因此我们可以将payload放入缓冲区buf中，以缓冲区buf的起始地址0xbffffc4c为payload入口地址。将shellcode全部放入至缓冲区后，再使用\x90字节进行填充，直至填充到了260个字节，再放入入口地址0xbffffc4c，覆盖原来的返回地址，从而成功构造长度为264个字节的payload。

构造payload使用到的主要代码如下图所示。

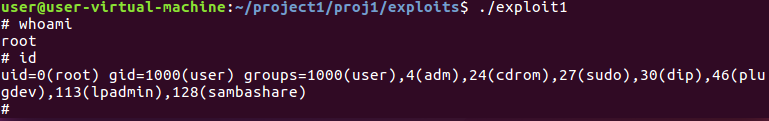


### 2.1.4攻击过程描述

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



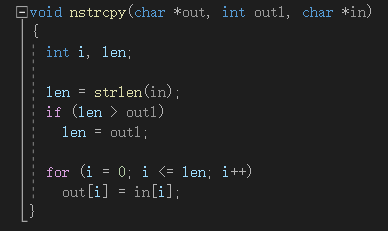
执行程序exploit1，成功获取一个具有root权限的shell，说明成功实现攻击过程。



## 2.2 vul2与exploit2

### 2.2.1 vul2描述

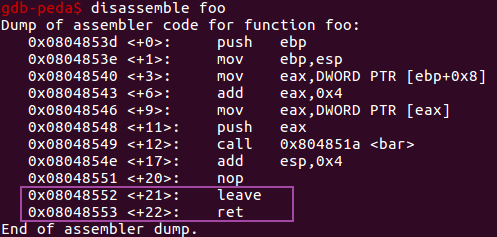
漏洞程序vul2与vul1相似，都会将输入的命令行参数拷贝到缓冲区中，不过vul2相较于vul1增加了一个检测拷贝内容长度的步骤。不过在拷贝的过程中，for语句中的判断条件i <= len使用了小于等于号，而不是小于号，这说明拷贝的过程中会比len的值多拷贝一个字符。



### 2.2.2 exploit2.c攻击原理

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

gdb-peda$ disassemble foo



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

### 2.2.3 payload构造方式

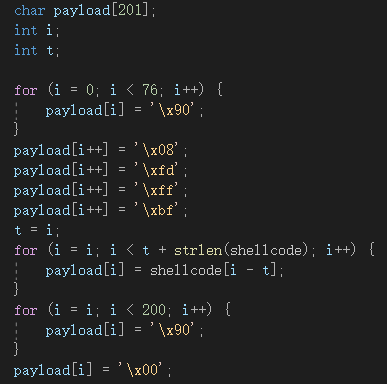
因为只能修改bar栈帧中旧EBP的值的最低位一个字节，所以需要使用gdb查看bar栈帧中旧EBP的值，该值为0xbffffd8c。这说明修改后的EBP的值最小是0xbffffd00，我们就按修改后的EBP的值是0xbffffd00继续构造payload。



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



构造paylaod使用到的主要代码如下图所示。



### 2.2.4攻击过程描述

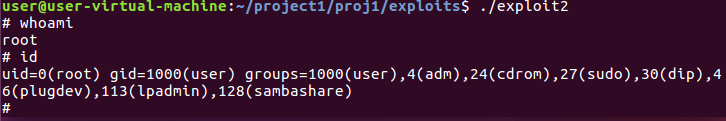
使用gdb工具调试程序，执行至函数bar准备返回时，即拷贝过程已经结束时，查看此时旧的EBP的值，已经成功被修改为0xbffffd00。



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



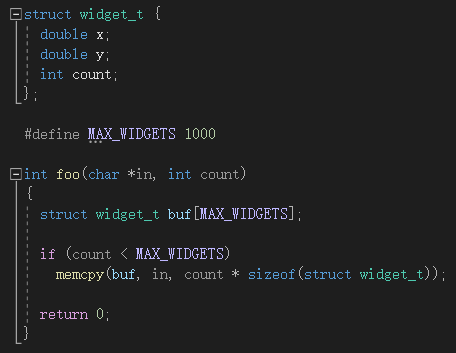
在函数bar拷贝完成后，将0xbffffd00赋给EBP并返回至函数foo的栈帧中，函数foo返回时会将ESP指向0xbffffd04地址处，指向的值0xbffffd08就是入口地址，然后程序跳转至0xbffffd08地址处，开始执行存放在此处的shellcode，从而获取具有root权限的shell。



## 2.3 vul3与exploit3

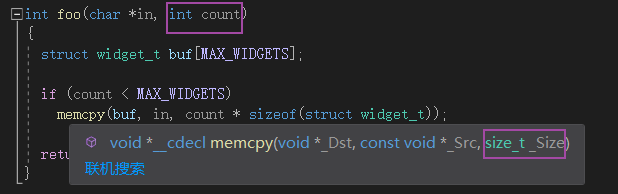
### 2.3.1 vul3描述

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



### 2.3.2 exploit3.c攻击原理

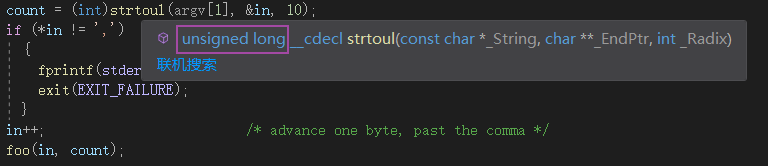
根据上述分析，想要造成缓冲区溢出，需要绕过memcpy前的if判断。可以发现传入的count参数是int类型，而memcpy接受的第三个参数，即指定拷贝的字节数参数是size\_t类型，本质上是unsigned类型。这表示存在有符号整数到无符号整数的转化过程中的漏洞。int类型的最高位为1时，会被认为是负数，因此成功通过程序中的if判断；而unsigned的最高位为1时，表示一个较大的正数，所以将count的最高位设为1且选择一个合适的值，就可以实现既通过if判断，又向缓冲区中写入比缓冲区更大的数据。



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

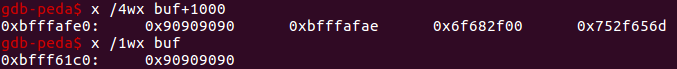
### 2.3.3 payload构造方式

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

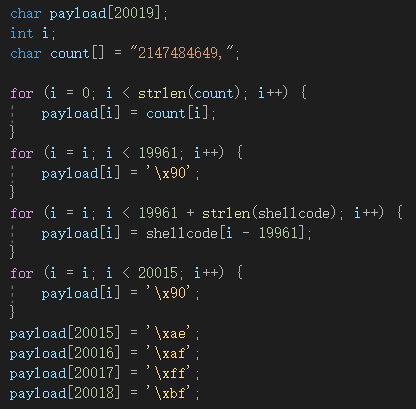


这个无符号整数我们选择2147484649这个值，即0x800003e9，符合最高位为1的条件，0x3e9十进制值为1001，比MAX\_WIDGETS的值大1，因为一个struct widget\_t类型占20个字节，覆盖返回地址只需要比缓冲区大小多8个字节的空间，因此多一个struct widget\_t类型的大小，也就是能够覆盖缓冲区后20个字节大小对我们来说是完全够用的。

然后查看返回地址所在地址的值，值为0xbfffafe4。再查看缓冲区的起始地址，值为0xbfff61c0。我选择了0xbfffafae地址作为shellcode的入口地址，这个地址距离缓冲区开头19950个字节。构造的payload还需要符合vul3的命令行参数格式[count],[data]，因此"2147484649,"不包含末尾零字节的字符串11个字节，加上缓冲区大小20000个字节，再加上覆盖返回地址需要缓冲区后8个字节的空间，payload总长度为20019个字节。在payload开头是字符串"2147484649,"的内容，然后在合适的位置写入shellcode，payload的最后是shellcode的入口地址，即用于覆盖返回地址的值，payload其余部分使用\x90字节填充。



构造payload使用到的主要代码如下图所示。

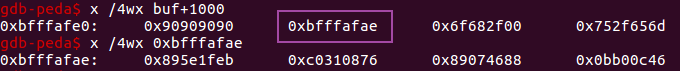


### 2.3.4攻击过程描述

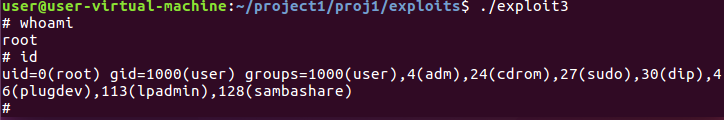
使用gdb工具调试程序，可以发现count的值为0x800003e9，与预期一致。



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



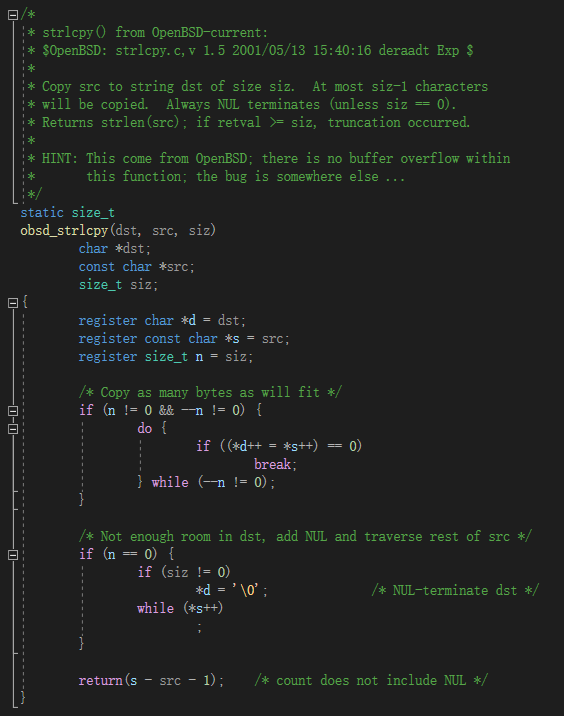
执行exploit3程序，获得一个具有root权限的shell。



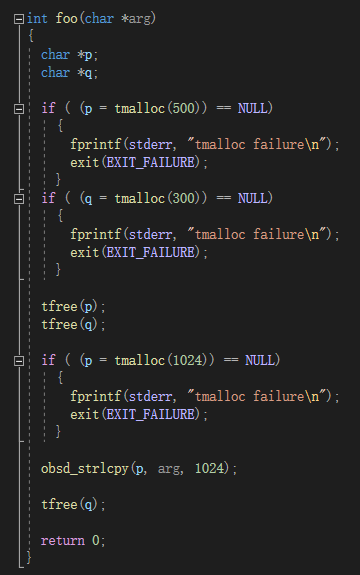
## 2.4 vul4与exploit4

### 2.4.1 vul4描述

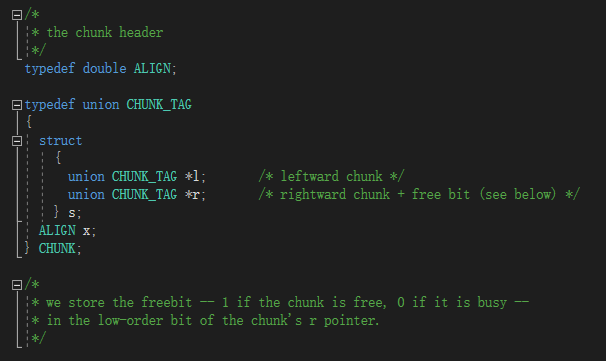
漏洞程序vul4会在堆中开辟空间，并将命令行中传入的参数拷贝至这段堆中的内存空间。使用的拷贝函数是obsd\_strlcpy，这个函数是不存在缓冲区溢出漏洞的，因此需要在程序的其他部分寻找漏洞。



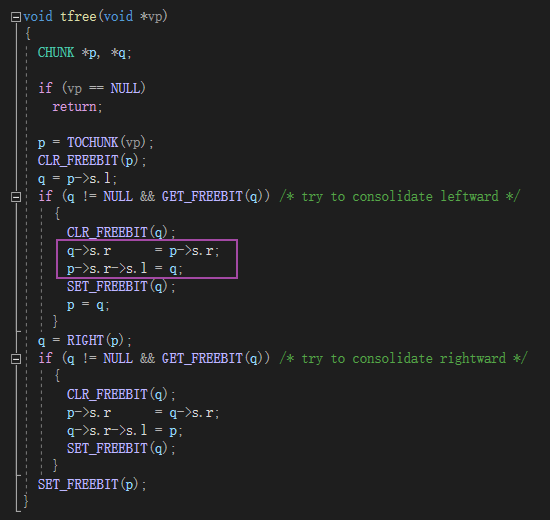
在函数foo中，先使用函数tmalloc分别为p和q分配了500字节和300字节的内存空间，然后使用函数tfree 分别释放p和q占有的内存空间。接着再为p分配了1024字节的内存空间，然后释放q占有的内存空间。因为此时q并没有被分配内存空间，却被释放了内存空间，所以这里应该是存在漏洞的。至于如何利用，需要具体分析tfree函数。



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

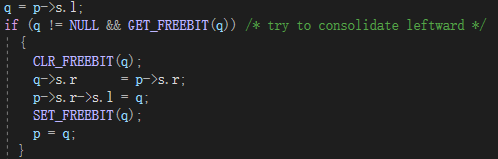


因此在tfree函数中，可以发现双向链表的删除操作，双向链表的删除是可以引起缓冲区溢出漏洞的，产生漏洞的具体代码如下图方框中所示。当待释放结点的左指针和右指针可以被我们所控制时，就能够产生缓冲区溢出漏洞。再根据之前对于foo函数的分析，可以发现p和q先被分配了内存，再释放了内存，但是p和q的值在释放内存后仍是不变的，然后p被分配了比之前p和q所分配的更大的内存，也就是说拷贝的内容可以覆盖q的左指针和右指针，因此可以实现缓冲区溢出。这个漏洞程序与前面三个漏洞程序相比，前面三个漏洞程序都属于栈溢出，这个漏洞程序属于堆溢出。

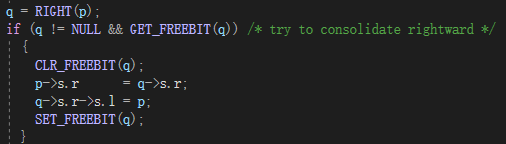


### 2.4.2 exploit4.c攻击原理

攻击该漏洞程序的关键在于下图所示代码，其中p是待释放的指针，q是p的左结点。值得一提的是，这里所提到的p和q是按照下图代码命名称呼，与函数foo中的p和q并不一致。然后会进行一个if判断，q不能为空且q是空闲的，结合tmalloc.c文件内容分析得到当某个结点的右指针的值的最低位为1，那么该结点是空闲的，若为0则表示该结点已被使用。为了能够进入if代码块，我们需要让q结点的右指针的最低位为1。再看if代码块中的内容，其中p->s.r->s.l = q;是我们覆盖返回地址的关键。只要将p->s.r，即p的右指针设置为返回地址所在地址，那么p->s.r就是返回地址的值，p->s.r->s.l就是返回地址指向的值。因为在union CHUNK\_TAG类型的定义中，s.l位于起始位置，所以p->s.r->s.l表示先取p->s.r指向的值，再取该值所指向的值。

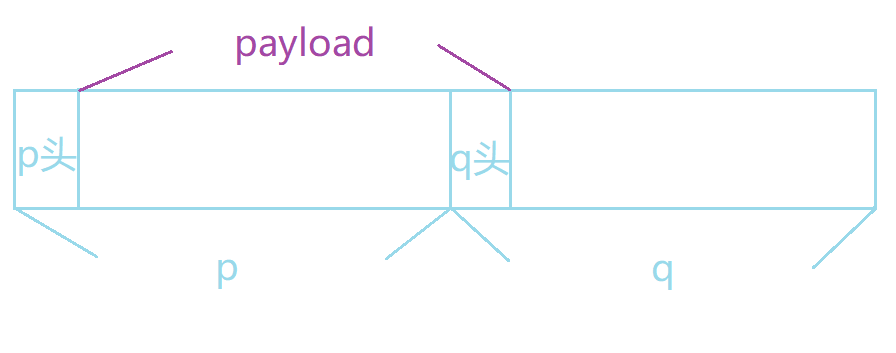


在tfree函数中，我们仅需要执行第一个if代码块就可以完成攻击，因此没有进入第二个if代码块的必要，同时也为了防止第二个if代码块的执行篡改了某些我们已经在内存中写入的payload，此处我将p的右指针的最低位设置为0，表示p的右结点已被使用，使得程序不会进入第二个if代码块。虽然之后经过调试发现第二个if代码块的执行不会影响攻击程序的执行效果，但是我还是把这个设置保存在了exploit4.c文件中，更为稳妥。如下图所示是第二个if代码块。

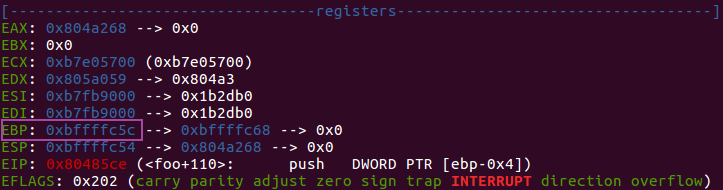


### 2.4.3 payload构造方式

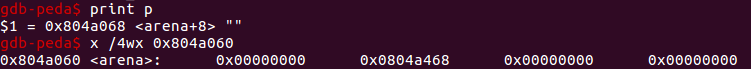
经过以上分析，可以发现只需要知道返回地址的地址、p的头结点的地址和q的头结点的地址就可以完成payload的构造。堆中数据的分布以及payload所在内存的位置如下图所示，其中p头表示p的头结点，q头表示q的头结点，整个p包括p头和数据部分，整个q包括q头和数据部分。



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

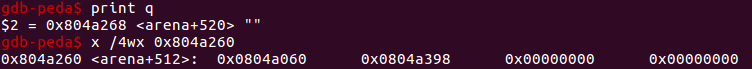


因为返回地址的值是shellcode的入口地址，同时返回地址的值是通过p->s.r->s.l = q;代码赋值得到，而这里面的q是函数tfree中的定义，在函数foo中这个值是q的左指针。所以返回地址的值，即shellcode的入口地址的值等于q的左指针。而q的左指针是指向p头的，而payload只能从p的数据部分开始，无法修改p头，因此我们还需要覆盖q的左指针，令其指向我们能够写入的地址。此处我直接选择p的数据部分的起始地址作为q的左指针的值，在gdb中查看到该值是0x804a068，p头起始地址是0x804a060。因此返回地址的值是0x804a068。



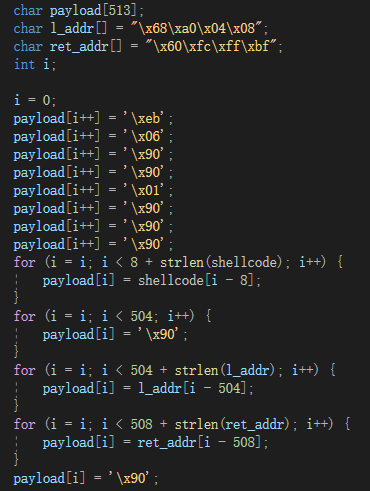
不过为了进入tfree函数中的第一个if代码块，q的左结点，也就是p必须是空闲状态，即p的右指针的最低位需要为1。此时查看我们的shellcode，发现如果将返回地址作为shellcode的入口地址，那么p的右指针所在的内存空间是shellcode的部分内容，即0xc0310876，最低位为0。因此不能直接将shellcode的入口地址设置为返回地址，而是需要构造payload使得p的右指针的最低位为1，然后将返回地址指向的指令设置为jmp短跳转指令的机器码，使其跳转到shellcode的入口地址，从而解决了需要将q的左结点设置为空闲状态的问题。这里将shellcode的入口地址设置为0x804a070，在返回地址的基础上加了8个字节，因此短跳转指令的机器码为'\xeb\x08'，\xeb表示jmp，\x08表示短跳转8个字节。

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



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

构造payload使用到的主要代码如下图所示。



### 2.4.4攻击过程描述

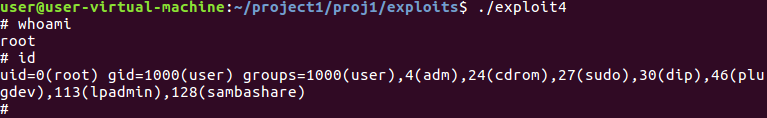
使用gdb工具调试查看拷贝后返回地址的值，可以发现已被修改为0x804a068。



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



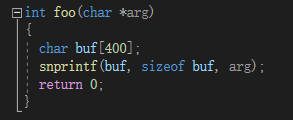
执行exploit4程序，获得一个具有root权限的shell。



## 2.5 vul5与exploit5

### 2.5.1 vul5描述

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



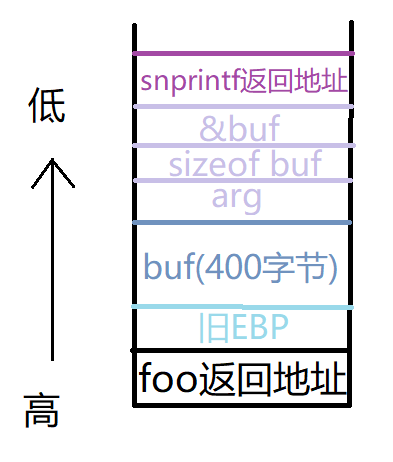
在格式化字符串参数中，存在"%n"格式，该格式会计算%n前的字符串长度，并将这个长度值写入至以这个格式所对应的值为地址的内存中。比如下图所示代码，会往指针p所指向的内存空间写入5这个值，因为%n前的字符串"12345"长度为5。



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

### 2.5.2 exploit5.c攻击原理

因为snprintf会先构造格式化字符串，再判断长度是否大于指定大小并截断字符串，所以可以使用"%n"格式进行攻击。而传入的参数仅有三个，如果使用"%n"格式，那么参数个数至少需要四个。snprintf会在栈中读取参数，并且会按照传入参数个数正常的情况读取参数，分析栈结构可以发现传入的三个参数后跟着的是buf的内存空间，因此snprintf会从buf中读取除前三个参数外的其他参数。参考的栈结构如图所示。



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

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

### 2.5.3 payload构造方式

经过上述分析可以发现我们只需要知道snprintf返回地址的地址和shellcode的入口地址就能够构造payload。首先使用gdb调试工具查看这两个地址。从下图可以发现snprintf第一个参数的地址是0xbffffc30和arg的起始地址是0xbfffff60，结合上述栈结构的分析可以得到snprintf返回地址的地址是第一个参数的地址减4，即0xbffffc2c；为了给"%n"和"%0<number>u"格式构造出能够覆盖snprintf返回地址的字符串预留空间，我选择在payload的第100个字节开始写入shellcode，也就是说shellcode的起始地址是0xbfffff60加100，即0xbfffffc4。

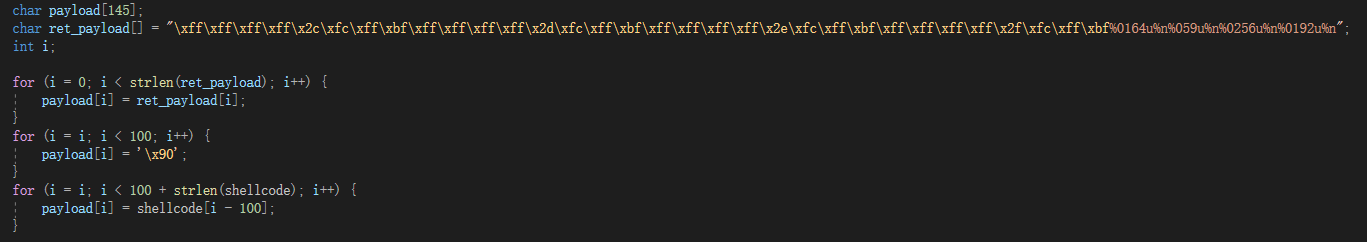


然后构造能够覆盖snprintf返回地址的payload，我选择每次使用"%n"写入一个字节的方式进行构造。具体而言，我在payload使用四次"%n"格式，四次格式的使用分别会在0xbffffc2c、0xbffffc2d、0xbffffc2e、0xbffffc2f四个地址处写入内容。然后需要用到"%0<number>u"格式，因为需要用到四次"%n"，那么一般也需要在每次"%n"前用到"%0<number>u"，所以需要四次"%0<number>u"，每次"%0<number>u"对应的值我选择是0xffffffff。从而使得snprintf总共需要11个参数，除了已经准备的3个参数，其他8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

我们需要将snprintf返回地址的值设置为0xbfffffc4，每次写入一个字节，也就是分别写入0xc4、0xff、0xff、0xbf。首先写入0xc4，因为payload开头是snprintf的后8个参数，共占32个字节，0xc4的十进制值是196，所以仅需要用"%0<number>u"格式填充164个字符；接着写入0xff，十进制值是255，前面已有字符长度是196，所以需要填充59个字符；接着再写入0xff，十进制值是255，但是因为前面已经构造好了每个"%0<number>u"格式对应的值是0xffffffff，这个值的十进制值是4294967295，占10个字符，因此在已经至少有265个字符的情况下无法减少为255个字符。

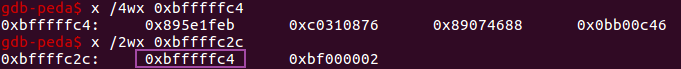
此时可以写入0x1ff，即十进制值511以解决这个问题，因为我的做法是每次只写入一个字节，所以写入的第二个字节会被后面写入的字节所覆盖，或者不在返回地址的地址范围内，从而不会造成影响。同理最后写入0x2bf，十进制值是703，完成了负责覆盖snprintf返回地址的payload的构造。再在payload的第100个字节开始写入shellcode，其余部分用\x90字节填充，payload总长为前100个字节加上shellcode的45个字节，即145个字节。

构造payload使用到的主要代码如下图所示。

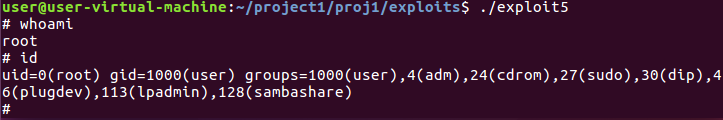


### 2.5.4攻击过程描述

使用gdb调试工具查看函数snprintf返回后0xbffffc2c存储的返回地址的值，已经成功被shellcode的入口地址0xbfffffc4覆盖，并且shellcode确实存在于这个地址指向的内存空间中。



执行exploit5程序，获得一个具有root权限的shell。

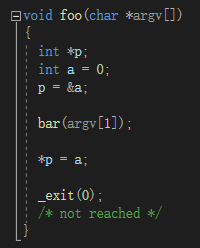


## 2.6 vul6与exploit6

### 2.6.1 vul6描述

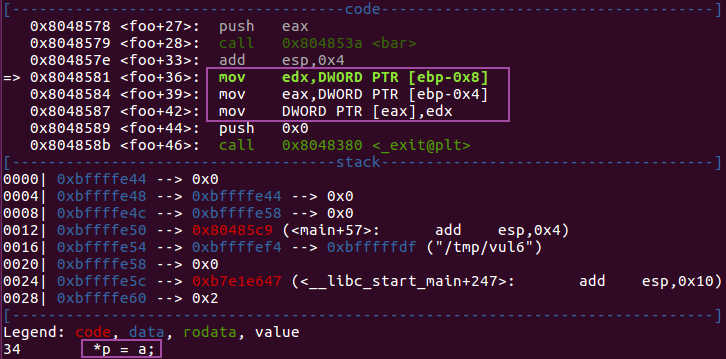
漏洞程序vul6与vul2非常相似，函数bar调用函数nstrcpy，在函数nstrcpy中最多能够比缓冲区buf大小多拷贝一个字节，从而存在缓冲区溢出。在vul2中通过这多拷贝的一个字节改变函数foo栈帧中的寄存器EBP的值，再修改返回地址的值。不过在vul6中，函数foo调用了\_exit函数提前结束进程，因此程序不会执行函数foo的leave和ret汇编指令，也就无法使用exploit2.c的攻击方式。

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

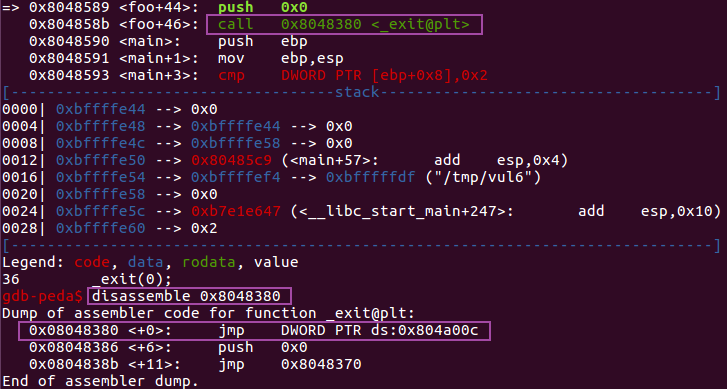


### 2.6.2 exploit6.c攻击原理

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



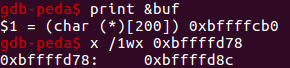
此时往返回地址的地址处写入shellcode的入口地址是行不通的，因为有函数\_exit的存在函数foo已经无法返回，所以需要寻找新的跳转指令。考虑到调用函数\_exit时程序需要跳转，因此查看\_exit(0);语句所对应的汇编指令。如下图所示，存在jmp指令，jmp的目标地址存储于0x804a00c地址处。



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

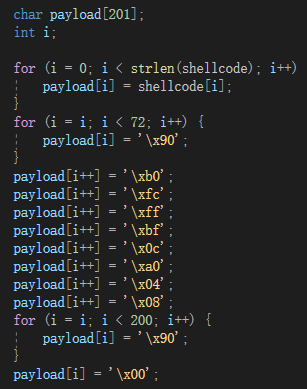
### 2.6.3 payload构造方式

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



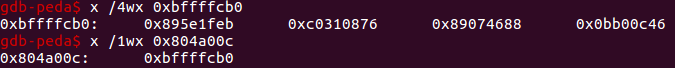
在EBP-0x4和EBP-0x8地址处需要写入预先准备的值，已知EBP为0xbffffd00，从而得到EBP-0x4为0xbffffcfc，EBP-0x8为0xbffffcf8。计算0xbffffcf8与缓冲区buf的起始地址0xbffffcb0相差72个字节，因此在payload的第72个字节开始写入p的值，即0x804a00c。在payload的第76个字节开始写入a的值，即shellcode的入口地址，值为0xbffffcb0。因为payload需要比缓冲区buf大一个字节才能够覆盖旧EBP的最低字节，所以payload大小为201个字节。除了上述关键地址外，paylaod其余部分均用\x90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

构造payload使用到的主要代码如下图所示。



### 2.6.4攻击过程描述

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



执行exploit6程序，获得一个具有root权限的shell。

