

拖了好久，但是在期间做了几道pwn题目，发现堆原来也没有想象中的难。

## fastbin\_dup\_into\_stack

这个说白了，就是利用double free可以进行任意地址的写，说是任意地址不准确，这个任意地址需要先进行布局！这个例子演示的是将一个堆分配到栈上。

先上一个简化版的源码：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main() {
5     unsigned long long stack_var = 0x21;
6     fprintf(stderr, "Allocating 3 buffers.\n");
7     char *a = malloc(9);
8     char *b = malloc(9);
9     char *c = malloc(9);
10    strcpy(a, "AAAAAAAA");
11    strcpy(b, "BBBBBBBB");
12    strcpy(c, "CCCCCCCC");
13    fprintf(stderr, "1st malloc(9) %p points to %s\n", a, a);
14    fprintf(stderr, "2nd malloc(9) %p points to %s\n", b, b);
15    fprintf(stderr, "3rd malloc(9) %p points to %s\n", c, c);
16    fprintf(stderr, "Freeing the first one %p.\n", a);
17    free(a);
18    fprintf(stderr, "Then freeing another one %p.\n", b);
19    free(b);
20    fprintf(stderr, "Freeing the first one %p again.\n", a);
21    free(a);
```

```
22  fprintf(stderr, "Allocating 4 buffers.\n");
23  unsigned long long *d = malloc(9);
24  *d = (unsigned long long) (((char*)&stack_var) - sizeof(d));
25  fprintf(stderr, "4nd malloc(9) %p points to %p\n", d, &d);
26  char *e = malloc(9);
27  strcpy(e, "EEEEEEEE");
28  fprintf(stderr, "5nd malloc(9) %p points to %s\n", e, e);
29  char *f = malloc(9);
30  strcpy(f, "FFFFFFFF");
31  fprintf(stderr, "6rd malloc(9) %p points to %s\n", f, f);
32  char *g = malloc(9);
33  strcpy(g, "GGGGGGGG");
34  fprintf(stderr, "7th malloc(9) %p points to %s\n", g, g);
35 }
```

用pwndbg一步步调试看看：

在22行的地方下个断点。

```
pwndbg> fastbins
fastbins
0x20: 0x602000 → 0x602020 ← 0x602000
```

然后进行先进行

`d=malloc(9)`

`*d=栈地址`

```
pwndbg> fastbins
fastbins
0x20: 0x602020 → 0x602000 → 0x7fffffffda38 → 0x602010 ← 0x0
0x30: 0x0
```

这里的这个栈地址，不是随便的地址，而是

```
5 unsigned long long stack_var = 0x21;
```


减去0x8的位置。

这里的目的就是为了让这里的0x7fffffffda38作为chunk的prev\_size字段，然后让stack\_var这个八个字节作为chunk的size字段，因为在从fastbins中取空间的时候，在2.23的libc中是会检查size字段，需要size字段合适，如果size字段不对，就不会分配成功。

glibc 在执行分配操作时，若块的大小符合 fast bin，则会在对应的 bin 中寻找合适的块，此时 glibc 将根据候选块的 size 字段计算出 fastbin 索引，然后与对应 bin 在 fastbin 中的索引进行比较，如果二者不匹配，则说明块的 size 字段遭到破坏。所以需要 fake chunk 的 size 字段被设置为正确的值。

接下来就是再分配两个chunk，这个时候最后一个chunk就被分配到栈上面了。

```
31 fprintf(stderr, "6rd malloc(9) %p points to %s\n", f, f);  
32 char *g = malloc(9);  
33 strcpy(g, "GGGGGGGG");
```



```

pwndbg> Stack
00:0000 | rsp | 0x7fffffffda30 -> 0x7fffffffda90 -> 0x4008f0 (__libc_csu_i
01:0008 | | 0x7fffffffda38 -> 0x4008a1 (main+612) -> 0xe8458b48e845894
02:0010 | | 0x7fffffffda40 -> 0x21 /* '!' */ -> 0x0
03:0018 | rax rdx | 0x7fffffffda48 -> 'GGGGGGGG'
04:0020 | | 0x7fffffffda50 -> 0x602009 -> 0x0
05:0028 | | 0x7fffffffda58 -> 0x602031 -> 'EEEEEEEE'
06:0030 | | 0x7fffffffda60 -> 0x602050 -> 'CCCCCCCC'
07:0038 | | 0x7fffffffda68 -> 0x602030 -> 'EEEEEEEE'
pwndbg>

```

虽然图片中的代码是向堆写内容，但是其实我们写入的地址是栈，这样就实现了有条件的任意地址写！

### fastbin\_dup\_consolidate

在前面我们说到，在libc2.23版本下，double free是有检测，会检测表头是不是上次被free的chunk头，这个检测好像在2.27版本就没了。。。还没有学习到那里，就先不谈了，这里说一下这个例子能干嘛。这个例子也是可以绕过doublefree的检测，但是不是我们前面讲的

```
free(a)
```

```
free(b)
```

```
free(a)
```

先看源码：

```
1 #include <stdio.h>
2 #include <stdint.h>
3 #include <stdlib.h>
4 #include <string.h>
5 int main()
6 {
7     void *p1 = malloc(0x10);
8     void *p2 = malloc(0x10);
9     strcpy(p1, "AAAAAAA");
10    strcpy(p2, "BBBBBBBB");
11    fprintf(stderr, "Allocated two fastbins: p1=%p p2=%p\n", p1, p2);
12    fprintf(stderr, "Now free p1!\n");
13    free(p1);
14    void *p3 = malloc(0x400);
15    fprintf(stderr, "Allocated large bin to trigger malloc_consolidate(): p3=%p\n", p3);
16    fprintf(stderr, "In malloc_consolidate(), p1 is moved to the unsorted bin.\n");
17    free(p1);
18    fprintf(stderr, "Trigger the double free vulnerability!\n");
19    fprintf(stderr, "We can pass the check in malloc() since p1 is not fast top.\n");
20    void *p4 = malloc(0x10);
21    strcpy(p4, "CCCCCCC");
22    void *p5 = malloc(0x10);
23    strcpy(p5, "DDDDDDDD");
24    fprintf(stderr, "Now p1 is in unsorted bin and fast bin. Some will get it twice: %p %p\n", p4, p5);
25 }
```

```
Triumph@ubuntu:~/桌面/how2heap/fastbin_dup_consolidate$ ./a.out
Allocated two fastbins: p1=0x17f0010 p2=0x17f0030
Now free p1!
Allocated large bin to trigger malloc_consolidate(): p3=0x17f0050
In malloc_consolidate(), p1 is moved to theunsorted bin.
Trigger the double free vulnerability!
We can pass the check in malloc() since p1is not fast top.
Now p1 is in unsorted bin and fast bin. Sowe'll get it twice: 0x17f0010 0x17f0010
```

我们先在14行下一个断点看看：

```
6 {
7     void *p1 = malloc(0x10);
8     void *p2 = malloc(0x10);
9     strcpy(p1, "AAAAAAA");
10    strcpy(p2, "BBBBBBB");
11    fprintf(stderr, "Allocated two fastbins: p1=%p p2=%p\n", p1,p2);
12    fprintf(stderr, "Now free p1!\n");
13    free(p1);
14    void *p3 = malloc(0x400);
15    fprintf(stderr, "Allocated large bin to trigger malloc_consolidate(): p3=%p\n", p3);
16    fprintf(stderr, "In malloc_consolidate(), p1 is moved to theunsorted bin.\n");
17    free(p1);
```

```
0x602000 FASTBIN {      consolidate.c      consolidate1.c
    prev_size = 0,
    size = 33,
    fd = 0x0,
    bk = 0x0,
    fd_nextsize = 0x0,
    bk_nextsize = 0x21
}
0x602020 FASTBIN {
    prev_size = 0,
    size = 33,
    fd = 0x4242424242424242,
    bk = 0x0,
    fd_nextsize = 0x0,
    bk_nextsize = 0x20fc1
}
```

```
pwndbg> fastbins
fastbins
0x20: 0x602000 ← 0x0
```

此时的申请的chunk0已经被放到fastbins里面了，这个时候时候我们

```
malloc(0x400)
```

```
pwndbg> bins
fastbins
0x20: 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0x20: 0x602000 → 0x7ffff7dd1b88 (main_arena+104) ← 0x602000
largebins
empty
```

这里我们发现，原来在fastbins的chunk0跑到了smallbins

malloc(0x400)，就是在分配large chunk

当分配 large chunk 时，首先根据 chunk 的大小获得对应的 large bin 的 index，接着判断当前分配区的 fast bins 中是否包含 chunk，如果有，调用 malloc\_consolidate() 函数合并 fast bins 中的 chunk，并将这些空闲 chunk 加入 unsorted bin 中。因为这里分配的是一个 large chunk，所以 unsorted bin 中的 chunk 按照大小被放回 small bins 或 large bins 中。

large bins



chunk 的指针数组，每个元素是一条双向循环链表的头部，但同一条链表中块的大小不一定相同，按照从大到小的顺序排列，每个bin保存一定大小范围的块。主要保存大小 1024 字节以上的块。

由于此时 p1 已经不在 fastbins 的顶部，可以再次释放 p1

```

pwndbg> bins
fastbins
0x20: 0x602000 ← 0x0
0x30: 0x0
0x40: 0x0
0x50: 0x0
0x60: 0x0
0x70: 0x0
0x80: 0x0
unsortedbin
all: 0x0
smallbins
0x20 [corrupted]
FD: 0x602000 ← 0x0
BK: 0x602000 → 0x7ffff7dd1b88 (main_arena+104) ← 0x602000
largebins
empty

```

这个时候我们发现chunk0也就是p1，又在fastbins里面，又在smallbins里面，当我们再次创建chunk的时候，第一次会从fastbins里面提取出chunk，第二次会从smallbins里面拿出chunk，所以这里的p4 p5都指向chunk0

```

void *p4 = malloc(0x10);
strcpy(p4, "CCCCCCC");
void *p5 = malloc(0x10);
strcpy(p5, "DDDDDDDD");
fprintf(stderr, "Now p1 is in unsorted bin and fast bin. Sowe'll get it twice: %p %p\n", p4, p5);

```

```
Now p1 is in unsorted bin and fast bin. Sowe'll get it twice: 0x17f0010 0x17f0010
```

虽然写到这里就完了，但是我还是有一点点不太明白。

当分配 large chunk 时，首先根据 chunk 的大小获得对应的 large bin 的 index，接着判断当前分配区的 fast bins 中是否包含 chunk，如果有，调用 malloc\_consolidate() 函数合并 fast bins 中的 chunk，并将这些空闲 chunk 加入 unsorted bin 中。因为这里分配的是一个 large chunk，所以 unsorted bin 中的 chunk 按照大小被放回 small bins 或 large bins 中。

说一下我现在的理解吧，就是我要申请一个 large chunk，我首先会看看 unsorted bin 中有没有合适的 chunk，我就会将 fast bins 中的 chunk 合并到 unsorted bin 中，但是由于我们申请的这个 chunk 太大了，即使把 fast bins 的 chunk 合并过来也不满足，所以我这里就按照大小，又把合并完的这个 chunk 放回到 small bins 或者 large bins。

嗯嗯，目前就这个理解。刚刚测试了一下，确实会合并 fast bins 的 chunk，合并完之后判断这个 chunk 是属于 small bins 还是 large bins，再按规则放！

这篇博客就先到这里啦！

\_\_EOF\_\_