

接下来的时间会通过how2heap学习堆的知识，这个系列可能会更新很多篇，因为每天学习到的东西要保证吸收消化，所以一天不会学习很多，但是又想每天记录一下。所以开个系列。

first_fit

此题的源码经过简化，如下：

```
1 #include <stdio.h>
2 #include <stdlib.h>
3 #include <string.h>
4 int main() {
5     char* a = malloc(512); //0x200
6     char* b = malloc(256); //0x100
7     char* c;
8     fprintf(stderr, "1st malloc(512): %p\n", a);
9     fprintf(stderr, "2nd malloc(256): %p\n", b);
10    strcpy(a, "AAAAAAAA");
11    strcpy(b, "BBBBBBBB");
12    fprintf(stderr, "first allocation %p points to %s\n", a, a);
13    fprintf(stderr, "Freeing the first one...\n");
14    free(a);
15    c = malloc(500);
16    fprintf(stderr, "3rd malloc(500): %p\n", c);
17    strcpy(c, "CCCCCCCC");
18    fprintf(stderr, "3rd allocation %p points to %s\n", c, c);
19    fprintf(stderr, "first allocation %p points to %s\n", a, a);
20 }
```

用gcc进行编译处理，命令：gcc -g first_fit1.c

运行一下看输出结果：

```
Thriumph@ubuntu:~/桌面/how2heap/first_fit$ ./a.out
1st malloc(512): 0x1d24010
2nd malloc(256): 0x1d24220
first allocation 0x1d24010 points to AAAAAAAA
Freeing the first one...
3rd malloc(500): 0x1d24010
3rd allocation 0x1d24010 points to CCCCCCCC
first allocation 0x1d24010 points to CCCCCCCC
```

这个程序想让我们明白的是假如我先malloc了一个比较大的堆，然后free掉，当我再申请一个小于刚刚释放的堆的时候，就会申请到刚刚free那个堆的地址。还有就是，我虽然刚刚释放了a指向的堆，但是a指针不会清零，仍然指向那个地址。这里就存在一个uaf (use_after_free)漏洞，原因是free的时候指针没有清零。

接下来再放一些学习资料上面话，比较官方，比较准确。

这第一个程序展示了 glibc 堆分配的策略，即 first-fit。在分配内存时，malloc 会先到 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 里去找合适的 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。

所以当释放一块内存后再申请一块大小略小于的空间，那么 glibc 倾向于将先前被释放的空间重新分配。

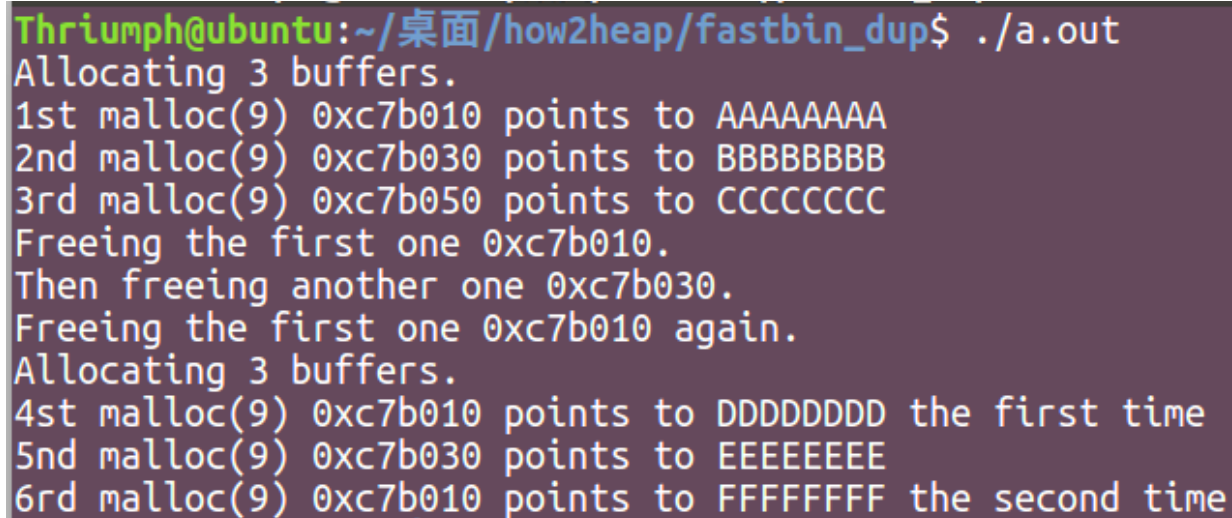
fastbin_dup

还是先放一下程序源码：

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

```
20  free(a);
21  fprintf(stderr, "Allocating 3 buffers.\n");
22  char *d = malloc(9);
23  char *e = malloc(9);
24  char *f = malloc(9);
25  strcpy(d, "DDDDDDDD");
26  fprintf(stderr, "4st malloc(9) %p points to %s the first time\n", d, d);
27  strcpy(e, "EEEEEEEE");
28  fprintf(stderr, "5nd malloc(9) %p points to %s\n", e, e);
29  strcpy(f, "FFFFFFF");
30  fprintf(stderr, "6rd malloc(9) %p points to %s the second time\n", f, f);
31 }
```

同样的gcc编译运行后看一下运行结果：



```
Thriumph@ubuntu:~/桌面/how2heap/fastbin_dup$ ./a.out
Allocating 3 buffers.
1st malloc(9) 0xc7b010 points to AAAAAAAA
2nd malloc(9) 0xc7b030 points toBBBBBBBB
3rd malloc(9) 0xc7b050 points to CCCCCCCC
Freeing the first one 0xc7b010.
Then freeing another one 0xc7b030.
Freeing the first one 0xc7b010 again.
Allocating 3 buffers.
4st malloc(9) 0xc7b010 points to DDDDDDDD the first time
5nd malloc(9) 0xc7b030 points to EEEEEEEE
6rd malloc(9) 0xc7b010 points to FFFFFFFF the second time
```

程序做了哪些事呢？

1. malloc申请了三个堆，并赋值。

2. free了第一个堆。

3. free了第二个堆。

4. 再次free了第一个堆。

5. malloc又申请了三个堆。

发现：第五步malloc申请堆的时候，第一个堆申请到了free第一次的位置，第二个堆申请到了free第二次的位置，第三个堆又申请到了free了第一次的位置。

这个程序展示了利用 **fastbins** 的 **double-free** 攻击，可以泄漏出一块已经被分配的内存指针。**fastbins** 可以看成一个 **LIFO** 的栈，使用单链表实现，通过 **fastbin->fd** 来遍历 **fastbins**。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。

说白了就是连着free两次一个堆是不被允许的，但是假如再其中加一个free其他堆，那么就可以对一个堆free两次。这样再我们malloc再申请的时候，就可以申请到两个指针指向同一个堆块了。

为了方便理解，我们试着在pwndbg里面看看：

首先在11行的位置下了断点：

```

1 #include <string.h>
2
3 int main() {
4     fprintf(stderr, "Allocating 3 buffers.\n");
5     char *a = malloc(9);
6     char *b = malloc(9);
7     char *c = malloc(9);
8     strcpy(a, "AAAAAAAA");
9     strcpy(b, "BBBBBBBB");
10    strcpy(c, "CCCCCCCC");
11    fprintf(stderr, "1st malloc(9) %p points to %s\n", a, a);
12    fprintf(stderr, "2nd malloc(9) %p points to %s\n", b, b);
13    fprintf(stderr, "3rd malloc(9) %p points to %s\n", c, c);
14    fprintf(stderr, "Freeing the first one %p.\n", a);
15
16 }

```

```

pwndbg> heap
0x602000 FASTBIN {
  prev_size = 0,
  size = 33,
  fd = 0x4141414141414141,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x21
}
0x602020 FASTBIN {
  prev_size = 0,
  size = 33,
  fd = 0x4242424242424242,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x21
}
0x602040 FASTBIN {
  prev_size = 0,
  size = 33,
  fd = 0x4343434343434343,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x20fa1
}

```

可以看到我们申请的三个堆，接下来我们在看一下free(a)、free(b)、再次free(a)时的情况：

```

14 fprintf(stderr, "3rd malloc(9) %p point\n", a);
15 fprintf(stderr, "Freeing the first one\n");
16 free(a);
17 fprintf(stderr, "Then freeing another\n");
18 free(b);
19 fprintf(stderr, "Freeing the first one\n");
20 free(a);
21 fprintf(stderr, "Allocating 3 buffers\n");
22 char *d = malloc(9);

```

```

0x602000 FASTBIN {
  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 = 0x21
}
0x602040 FASTBIN {
  prev_size = 0,
  size = 33,
  fd = 0x4343434343434343,
  bk = 0x0,
  fd_nextsize = 0x0,
  bk_nextsize = 0x20fa1
}

```

<pre> 0x602000 FASTBIN { prev_size = 0, size = 33, fd = 0x0, bk = 0x0, fd_nextsize = 0x0, bk_nextsize = 0x21 } 0x602020 FASTBIN { prev_size = 0, size = 33, fd = 0x602000, bk = 0x0, fd_nextsize = 0x0, bk_nextsize = 0x21 } 0x602040 FASTBIN { prev_size = 0, size = 33, fd = 0x4343434343434343, bk = 0x0, fd_nextsize = 0x0, bk_nextsize = 0x20fa1 } </pre>	<pre> 0x602000 FASTBIN { prev_size = 0, size = 33, fd = 0x602020, bk = 0x0, fd_nextsize = 0x0, bk_nextsize = 0x21 } 0x602020 FASTBIN { prev_size = 0, size = 33, fd = 0x602000, bk = 0x0, fd_nextsize = 0x0, bk_nextsize = 0x21 } 0x602040 FASTBIN { prev_size = 0, size = 33, fd = 0x4343434343434343, bk = 0x0, fd_nextsize = 0x0, bk_nextsize = 0x20fa1 } </pre>
--	---

可以看到fastbins形成了一个环，但是其实应该是栈的样子的，但是由于我们绕过了检测，就可以形成环。

```

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

```


|Chunk A| -> |chunk B| --> | chunk A|

大概如上个图，这样我们就成功绕过了 fastbins 的double free检查。原因如下：

fastbins 可以看成是一个 LIFO 的栈，使用单链表实现，通过 fastbin->fd 来遍历 fastbins。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。

上面的情况是在libc-2.23版本做的实验，但是好像版本不同的时候会有其他情况。这里就直接拿资料上的东西了。

-----资料-----

看一点新鲜的，在 libc-2.26 中，即使两次 free，也并没有触发 double-free 的异常检测，这与 tcache 机制有关，以后会详细讲述。这里先看个能够在该版本下触发double-free 的例子：

```
#include <stdio.h>
#include <stdlib.h>
int main() {
    int i;
    void *p = malloc(0x40);
    fprintf(stderr, "First allocate a fastbin: p=%p\n", p);
    fprintf(stderr, "Then free(p) 7 times\n");
    for (i = 0; i < 7; i++)
    {
```

```
    fprintf(stderr, "free %d: %p => %p\n", i+1, &p, p);
    free(p);
}
fprintf(stderr, "Then malloc 8 times at the same address\n");
int *a[10];
for (i = 0; i < 8; i++)
{
    a[i] = malloc(0x40);
    fprintf(stderr, "malloc %d: %p => %p\n", i+1, &a[i], a[i]);}
fprintf(stderr, "Finally trigger double-free\n");
for (i = 0; i < 2; i++)
{
    fprintf(stderr, "free %d: %p => %p\n", i+1, &a[i], a[i]);
    free(a[i]);
}
}
```

首先先malloc申请了一个堆，接着连续free了7次。

然后malloc同样大小的堆块，申请了8次。

接下来又free了申请的前两个申请的堆块。

我们看一下运行结果：

可以从输出看到，8次重新申请的堆块都指向一个我们第一次申请的地址。

```
$ gcc -g tcache_double-free.c
$ ./a.out
First allocate a fastbin: p=0x559e30950260
Then free(p) 7 times
free 1: 0x7ffc498b2958 => 0x559e30950260
free 2: 0x7ffc498b2958 => 0x559e30950260
free 3: 0x7ffc498b2958 => 0x559e30950260
free 4: 0x7ffc498b2958 => 0x559e30950260
free 5: 0x7ffc498b2958 => 0x559e30950260
free 6: 0x7ffc498b2958 => 0x559e30950260
free 7: 0x7ffc498b2958 => 0x559e30950260
Then malloc 8 times at the same address
malloc 1: 0x7ffc498b2960 => 0x559e30950260
malloc 2: 0x7ffc498b2968 => 0x559e30950260
malloc 3: 0x7ffc498b2970 => 0x559e30950260
malloc 4: 0x7ffc498b2978 => 0x559e30950260
malloc 5: 0x7ffc498b2980 => 0x559e30950260
malloc 6: 0x7ffc498b2988 => 0x559e30950260
malloc 7: 0x7ffc498b2990 => 0x559e30950260
malloc 8: 0x7ffc498b2998 => 0x559e30950260
Finally trigger double-free
free 1: 0x7ffc498b2960 => 0x559e30950260
free 2: 0x7ffc498b2968 => 0x559e30950260
double free or corruption (fasttop)
[2] 1244 abort (core dumped) ./a.out
```

后记：最后这个列子我还是没看出有啥可以学习到的。。。但是我疑惑的是，free了7次，为什么第8次

malloc的时候，还是指向了第一次malloc的地址。