4: `exit->fcloseall->_IO_cleanup->_IO_flush_all->_IO_wfile_overflow->_IO_wdoallocbuf` **利用链代码审计**

本来打算照着wp先把剩下两题复现出来的,结果拼尽全力无法战胜house_of_apple.大概是超前研究惩罚太高了...

于是打算先看一下原理.

首先,让我们了解一下IO_FILE是什么.

"FILE 在 Linux 系统的标准 IO 库中是用于描述文件的结构，称为文件流。 FILE 结构在程序执行 fopen 等函数时会进行创建，并分配在堆中。我们常定义一个指向 FILE 结构的指针来接收这个返回值。"

一般来说,我们在题目里看到的那些setbuf()就和这种东西有关.

_IO_FILE的实现代码在struct_FILE.h中,具体结构如下:

```
struct _IO_FILE
{
  int _flags;//0x0
  char *_IO_read_ptr;//0x8 (由于编译器对齐,它的偏移是0x8)
  char *_IO_read_end;//0x10
  char *_IO_read_base;//0x18
  char *_IO_write_base;//0x20
  char *_IO_write_ptr;//0x28
  char *_IO_write_end;//0x30
  char *_IO_buf_base;//0x38
  char *_IO_buf_end;//0x40
  char *_IO_save_base;//0x48
  char *_IO_backup_base;//0x50
  char *_IO_save_end;//0x58
  struct _IO_marker *_markers;//0x60
  struct _IO_FILE *_chain;//0x68
  int _fileno;//0x70
  int _flags2;//0x74
  __off_t _old_offset;//0x78
```

```
  unsigned short _cur_column;//0x80
  signed char _vtable_offset;//0x82
  char _shortbuf[1];//0x83
  _IO_lock_t *_lock;//0x88
#ifdef _IO_USE_OLD_IO_FILE
};
struct _IO_FILE_complete
{
  struct _IO_FILE _file;
#endif
  __off64_t _offset;//0x90
  struct _IO_codecvt *_codecvt;//0x98
  struct _IO_wide_data *_wide_data;//0xa0
  struct _IO_FILE *_freeres_list;//0xa8
  void *_freeres_buf;//0xb0
  size_t __pad5;//0xb8
  int _mode;//0x70
  char _unused2[15 * sizeof (int) - 4 * sizeof (void *) - sizeof (size_t)];//0x74
};
```

而它的外面还会套上_IO_FILE_complete,而_IO_FILE_complete的外面还会被套上_IO_FILE_plus,所以完整的结构会是这样的:

```
struct _IO_FILE_plus
{
    _IO_FILE    file;
    IO_jump_t   *vtable;//0xd8
}
```

到了这一步,我们还是不知道调用链具体是怎样执行的.所以让我们从exit开始看:

首先, `exit` 调用了 `__run_exit_handlers`

```
void
exit (int status)
{
  __run_exit_handlers (status, &__exit_funcs, true, true);
}
libc_hidden_def (exit)
```

然后, `__run_exit_handlers` 调用了 `_IO_cleanup`

```
__run_exit_handlers (int status, struct exit_function_list **listp,//调用的代码在最后面
             bool run_list_atexit, bool run_dtors)
{
  /* First, call the TLS destructors.  */
  if (run_dtors)
    call_function_static_weak (__call_tls_dtors);

  __libc_lock_lock (__exit_funcs_lock);

  /* We do it this way to handle recursive calls to exit () made by
     the functions registered with `atexit' and `on_exit'. We call
```

```
     everyone on the list and use the status value in the last
     exit (). */
  while (true)
    {
      struct exit_function_list *cur;

    restart:
      cur = *listp;

      if (cur == NULL)
    {
      /* Exit processing complete.  We will not allow any more
         atexit/on_exit registrations.  */
      __exit_funcs_done = true;
      break;
    }

      while (cur->idx > 0)
    {
      struct exit_function *const f = &cur->fns[--cur->idx];
      const uint64_t new_exitfn_called = __new_exitfn_called;

      switch (f->flavor)
        {
          void (*atfct) (void);
          void (*onfct) (int status, void *arg);
          void (*cxafct) (void *arg, int status);
          void *arg;

        case ef_free:
        case ef_us:
          break;
        case ef_on:
          onfct = f->func.on.fn;
          arg = f->func.on.arg;
          PTR_DEMANGLE (onfct);

          /* Unlock the list while we call a foreign function.  */
          __libc_lock_unlock (__exit_funcs_lock);
          onfct (status, arg);
          __libc_lock_lock (__exit_funcs_lock);
          break;
        case ef_at:
          atfct = f->func.at;
          PTR_DEMANGLE (atfct);

          /* Unlock the list while we call a foreign function.  */
          __libc_lock_unlock (__exit_funcs_lock);
          atfct ();
          __libc_lock_lock (__exit_funcs_lock);
          break;
        case ef_cxa:
          /* To avoid dlclose/exit race calling cxafct twice (BZ 22180),
         we must mark this function as ef_free.  */
          f->flavor = ef_free;
          cxafct = f->func.cxa.fn;
```

```
            arg = f->func.cxa.arg;
            PTR_DEMANGLE (cxafct);

            /* Unlock the list while we call a foreign function.  */
            __libc_lock_unlock (__exit_funcs_lock);
            cxafct (arg, status);
            __libc_lock_lock (__exit_funcs_lock);
            break;
          }

        if (__glibc_unlikely (new_exitfn_called != __new_exitfn_called))
          /* The last exit function, or another thread, has registered
             more exit functions.  Start the loop over.  */
          goto restart;
      }

        *listp = cur->next;
        if (*listp != NULL)
      /* Don't free the last element in the chain, this is the statically
         allocate element.  */
      free (cur);
      }

  __libc_lock_unlock (__exit_funcs_lock);

  if (run_list_atexit)//一般来说,这个条件会在正常调用exit时被满足.
    call_function_static_weak (_IO_cleanup);

  _exit (status);
}
```

(上面的代码在exit.c中)

`_IO_cleanup`又调用了`_IO_flush_all ()`和`_IO_unbuffer_all ()`

```
int
_IO_cleanup (void)
{
  int result = _IO_flush_all ();
  _IO_unbuffer_all ();

  return result;
}
```

接下来让我们来看`_IO_flush_all`

```
int
_IO_flush_all (void)
{
  int result = 0;
  FILE *fp;

#ifdef _IO_MTSAFE_IO
  _IO_cleanup_region_start_noarg (flush_cleanup);
```

```
  _IO_lock_lock (list_all_lock);
#endif

  for (fp = (FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
    {
      run_fp = fp;//似乎是一个全局变量,也许接下来我们还能碰到它
      _IO_flockfile (fp);//上个线程锁

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)
    result = EOF;//一大堆判断

      _IO_funlockfile (fp);
      run_fp = NULL;
    }

#ifdef _IO_MTSAFE_IO
  _IO_lock_unlock (list_all_lock);
  _IO_cleanup_region_end (0);
#endif

  return result;
}
```

它通过一个for循环,遍历整个_IO_file链表

```
for (fp = (FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
```

通过large_bin_attack,可以将_IO_list_all覆盖为堆地址.

每个链表有一大堆判断:

```
if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)
```

首先,程序会判断这个:

```
((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base) || (_IO_vtable_offset (fp) == 0 && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr > fp->_wide_data->_IO_write_base)))
```

fp->_mode <= 0且fp->_IO_write_ptr > fp->_IO_write_base,或者 (_IO_vtable_offset (fp) == 0 && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr > fp->_wide_data->_IO_write_base)).这两个条件至少要满足一个,程序才会进行_IO_OVERFLOW()的判断.

此时,由于我们已经把*vtable指向了_IO_wfile_jumps,所以程序会去执行_IO_wfile_overflow.

```c
struct _IO_jump_t
{
    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_xsputn_t, __xsputn);
    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);
};
```

```c
wint_t
_IO_wfile_overflow (FILE *f, wint_t wch)
{
  if (f->_flags & _IO_NO_WRITES) /* SET ERROR */
    {
      f->_flags |= _IO_ERR_SEEN;
      __set_errno (EBADF);
      return WEOF;
    }
  /* If currently reading or no buffer allocated. */
  if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0
      || f->_wide_data->_IO_write_base == NULL)
    {
      /* Allocate a buffer if needed. */
      if (f->_wide_data->_IO_write_base == 0)
    {
      _IO_wdoallocbuf (f);
      _IO_free_wbackup_area (f);
      _IO_wsetg (f, f->_wide_data->_IO_buf_base,
             f->_wide_data->_IO_buf_base, f->_wide_data->_IO_buf_base);

      if (f->_IO_write_base == NULL)
        {
          _IO_doallocbuf (f);
          _IO_setg (f, f->_IO_buf_base, f->_IO_buf_base, f->_IO_buf_base);
        }
    }
```

```
      else
    {
      /* Otherwise must be currently reading.  If _IO_read_ptr
         (and hence also _IO_read_end) is at the buffer end,
         logically slide the buffer forwards one block (by setting
         the read pointers to all point at the beginning of the
         block).  This makes room for subsequent output.
         Otherwise, set the read pointers to _IO_read_end (leaving
         that alone, so it can continue to correspond to the
         external position). */
      if (f->_wide_data->_IO_read_ptr == f->_wide_data->_IO_buf_end)
        {
          f->_IO_read_end = f->_IO_read_ptr = f->_IO_buf_base;
          f->_wide_data->_IO_read_end = f->_wide_data->_IO_read_ptr =
        f->_wide_data->_IO_buf_base;
        }
    }
      f->_wide_data->_IO_write_ptr = f->_wide_data->_IO_read_ptr;
      f->_wide_data->_IO_write_base = f->_wide_data->_IO_write_ptr;
      f->_wide_data->_IO_write_end = f->_wide_data->_IO_buf_end;
      f->_wide_data->_IO_read_base = f->_wide_data->_IO_read_ptr =
    f->_wide_data->_IO_read_end;

      f->_IO_write_ptr = f->_IO_read_ptr;
      f->_IO_write_base = f->_IO_write_ptr;
      f->_IO_write_end = f->_IO_buf_end;
      f->_IO_read_base = f->_IO_read_ptr = f->_IO_read_end;

      f->_flags |= _IO_CURRENTLY_PUTTING;
      if (f->_flags & (_IO_LINE_BUF | _IO_UNBUFFERED))
    f->_wide_data->_IO_write_end = f->_wide_data->_IO_write_ptr;
    }
  if (wch == WEOF)
    return _IO_do_flush (f);
  if (f->_wide_data->_IO_write_ptr == f->_wide_data->_IO_buf_end)
    /* Buffer is really full */
    if (_IO_do_flush (f) == EOF)
      return WEOF;
  *f->_wide_data->_IO_write_ptr++ = wch;
  if ((f->_flags & _IO_UNBUFFERED)
      || ((f->_flags & _IO_LINE_BUF) && wch == L'\n'))
    if (_IO_do_flush (f) == EOF)
      return WEOF;
  return wch;
}
```

如果`f->_flags & _IO_NO_WRITES`不满足且`(f->_flags & _IO_CURRENTLY_PUTTING) == 0|| f->_wide_data->_IO_write_base == NULL`且`f->_wide_data->_IO_write_base == 0`,程序会继续调用利用链的下一环----`_IO_wdoallocbuf`

```c
void
_IO_wdoallocbuf (FILE *fp)
{
  if (fp->_wide_data->_IO_buf_base)
    return;
  if (!(fp->_flags & _IO_UNBUFFERED))
    if ((wint_t)_IO_WDOALLOCATE (fp) != WEOF)
      return;
  _IO_wsetb (fp, fp->_wide_data->_shortbuf,
             fp->_wide_data->_shortbuf + 1, 0);
}
```

我们需要使`fp->_wide_data->_IO_buf_base`不成立,且`(fp->_flags & _IO_UNBUFFERED)`为伪,让程序去执行有`_IO_WDOALLOCATE`的那个判断.

```c
#define _IO_WDOALLOCATE(FP) WJUMP0 (__doallocate, FP)
```

它会通过`_wide_data->_wide_vtable->__doallocate`调用到`__doallocate`.而我们可以通过伪造`_wide_vtable`以及它的`__doallocate`项,来实现任意函数执行.

(为什么要绕这么一大圈?是为了绕开各种各样的检查.由于这玩意的代码量比malloc大多了,我就暂时不去审计了.把利用链上的东西看完,明白大概是怎么个原理就行.)

当然,到这一步为止,我们已经可以用它执行one_gadget或system了,不过想要让它执行rop链(hgame的那题就开了沙箱,用不了one_gadget),我们还需要设法控制rsp.

既然我打算复现的题是2.39的,那就让我们先来考虑2.39版本的利用.

直接上图:

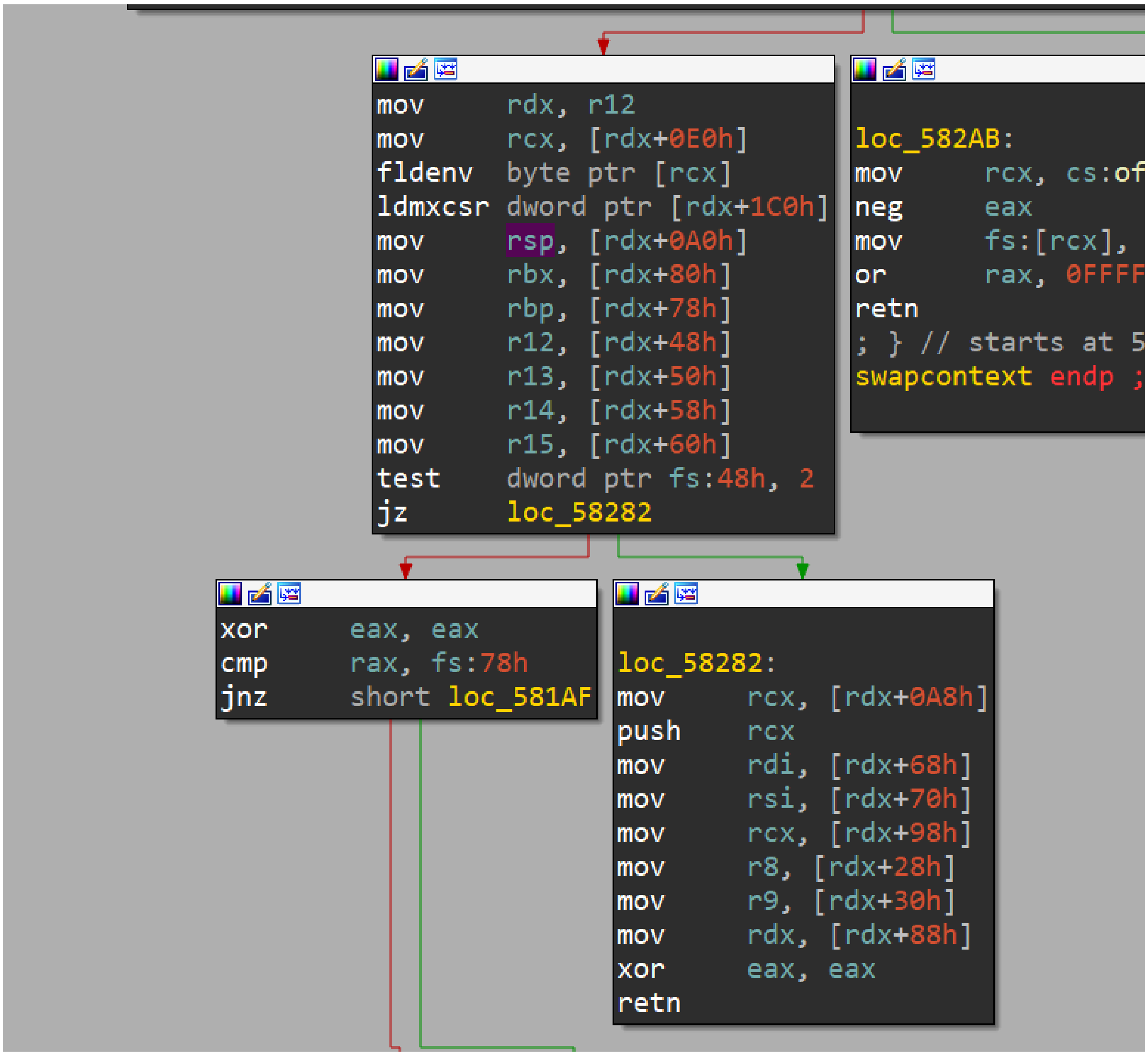
```
.text:000000000017922E 49 89 F5                    mov     r13, rsi
.text:0000000000179231 31 F6                       xor     esi, esi
.text:0000000000179233 41 54                       push    r12
.text:0000000000179235 53                          push    rbx
.text:0000000000179236 48 89 FB                    mov     rbx, rdi
.text:0000000000179239 48 83 EC 18                 sub     rsp, 18h
.text:000000000017923D 4C 8B 67 48                 mov     r12, [rdi+48h]
.text:0000000000179241 49 8B 44 24 18              mov     rax, [r12+18h]
.text:0000000000179246 4D 8D 74 24 10              lea     r14, [r12+10h]
.text:000000000017924B 41 C7 44 24 10 00 00 00 00  mov     dword ptr [r12+10h], 0
.text:0000000000179254 4C 89 F7                    mov     rdi, r14
.text:0000000000179257 FF 50 28                    call    qword ptr [rax+28h]
.text:0000000000179257
```

如图,如果我们把`__doallocate`覆写为这个地址,那么它就会实现接下来的一系列效果:

1. 将r12写为rdi+0x48(即fp+0x48)存储的值
2. 将rax写为r12+0x18内存储的值
3. 将r12+0x10写为0
4. call rax+0x28

也就是说,它将r12设为了*(fp+0x48),然后调用了*(*(*(fp+0x48)+0x18)+0x28) (好绕)

然后我们看下一张图:(图里没有给偏移,为0x5814D)



看起来很大一串,但实际上对我们有用的就三句:

```
mov rdx ,r12;
mov rsp,[rdx+0xA0]
ret
```

也就是说,我们可以将*(*(*(fp+0x48)+0x18)+0x28)设为这个地址,然后在*(*(fp+0x48)+0xe0)的位置布置rop链.

大概就这样.