

这是本系列的第四篇文章，经过fwrite以及fread的分析，在进行fclose调试之前，已经知道IO FILE结构体包括两个堆结构，一个是保存IO FILE结构体的堆，一个是输入输出缓冲区的堆。对于fclose的分析，主要有两个关注点，一个是函数的流程，一个就是对于堆块的处理（何时释放，如何释放）。

传送门：

- [IO FILE之fopen详解](#)
- [IO FILE之fread详解](#)
- [IO FILE之fwrite详解](#)

## 总体概览

还是首先把fclose的总体的流程描述一遍，从fopen的流程中，我们知道了fopen主要是建立了FILE结构体以及将其链接进入了\_IO\_list\_all链表中，同时fread或fwrite会

fclose函数实现主要是在\_IO\_new\_fclose函数中，大致可分为三步，基本上可以与fopen相对应：

1. 调用\_IO\_un\_link将文件结构体从\_IO\_list\_all链表中取下。
2. 调用\_IO\_file\_close\_it关闭文件并释放缓冲区。
3. 释放FILE内存以及确认文件关闭。

下面进行具体的源码分析。

## 源码分析

fclose的函数原型为：

```
int close(int fd);
```

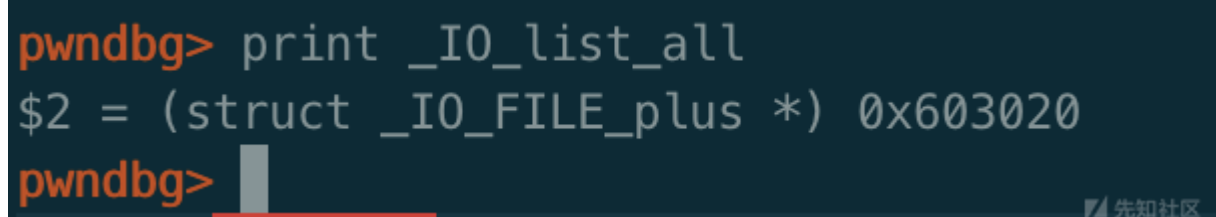
DESCRIPTION: close() closes a file descriptor, so that it no longer refers to any file and may be reused. Any record locks

demo程序如下,仍然是使用带调试符号的glibc2.23对代码进行调试：

```
#include<stdio.h>

int main(){
    char *data=malloc(0x1000);
    FILE*fp=fopen("test","wb");
    fwrite(data,1,0x60,fp);
    fclose(fp);
    return 0;
}
```

断点下在fclose函数。断下来以后以后，在调试之前将所需关注的内存结构先给出来，首先是此时的\_IO\_list\_all的值为此时的IO FILE结构体：



第二个是IO FILE结构体的值，其中需要留意的是经过fwrite的函数调用，此时输出缓冲区中是存在内容的，即\_IO\_write\_base小于\_IO\_write\_ptr：

```
pwndbg> print *_IO_list_all
$3 = {
  file = {
    _flags = 0xfbad2c84,
    _IO_read_ptr = 0x603250 "",
    _IO_read_end = 0x603250 "",
    _IO_read_base = 0x603250 "",
    _IO_write_base = 0x603250 "",
    _IO_write_ptr = 0x6032b0 "",
    _IO_write_end = 0x604250 "",
    _IO_buf_base = 0x603250 "",
    _IO_buf_end = 0x604250 "",
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x7ffff7dd5540 <_IO_2_1_stderr_>,
    _fileno = 0x3,
    _flags2 = 0x0,
    _old_offset = 0x0,
    _cur_column = 0x0,
    _vtable_offset = 0x0,
    _shortbuf = "",
    _lock = 0x603100,
    _offset = 0xffffffffffffffff,
    _codecvt = 0x0,
    _wide_data = 0x603110,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0xffffffff,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7ffff7dd36e0 <_IO_file_jumps>
}
pwndbg>
```

可以看到程序断在 `_IO_new_fclose` 函数，文件在 `/libio/iofclose.c` 中。可以看到 `_IO_new_fclose` 函数就是实现 `fclose` 的核心部分了：

```

int
_IO_new_fclose (_IO_FILE *fp)
{
    int status;

    ...

    if (fp->_IO_file_flags & _IO_IS_FILEBUF)
        _IO_unlink ((struct _IO_FILE_plus *) fp); // fp _IO_list_all

    ...
    if (fp->_IO_file_flags & _IO_IS_FILEBUF)
        status = _IO_file_close_it (fp); //
    ...
    _IO_FINISH (fp); // FILE
    ...
    if (fp != _IO_stdin && fp != _IO_stdout && fp != _IO_stderr)
    {
        fp->_IO_file_flags = 0;
        free(fp);
    }

    return status;
}

```

和fopen一样，代码的核心部分也比较少。

`_IO_unlink`将结构体从`_IO_list_all`链表中取下

第一部分，调用`_IO_unlink`函数将IO FILE结构体从`_IO_list_all`链表中取下，跟进去该函数，函数在`/libio/genops.c`中：

```

void
_IO_unlink (struct _IO_FILE_plus *fp)
{
    if (fp->file._flags & _IO_LINKED) //
    {
        ...
        if (_IO_list_all == NULL) // _IO_list_all
        ;
        else if (fp == _IO_list_all) // fp
        {
            _IO_list_all = (struct _IO_FILE_plus *) _IO_list_all->file._chain;
            ++_IO_list_all_stamp;
        }
        else // fp
        for (f = &_IO_list_all->file._chain; *f; f = &(*f)->_chain)
            if (*f == (_IO_FILE *) fp)
            {
                *f = fp->file._chain;
                ++_IO_list_all_stamp;
                break;
            }
        fp->file._flags &= ~_IO_LINKED; //
        ...
    }
}
libc_hidden_def (_IO_unlink)

```

函数先检查标志位是否包含`_IO_LINKED`标志，该标志的定义是`#define _IO_LINKED 0x80`，表示该结构体是否被链接到了`_IO_list_all`链表中。

如果没有`_IO_LINKED`标志（不在`_IO_list_all`链表中）或者`_IO_list_all`链表为空，则直接返回。

否则的话即表示结构体为`_IO_list_all`链表中某个节点，所要做的就是将这个节点取下来，接下来就是单链表的删除节点的操作，首先判断是不是`_IO_list_all`链表头

最后返回之前设置`file._flags`为`~_IO_LINKED`表示该结构体不在`_IO_list_all`链表中。

```

pwndbg> print _IO_list_all
$5 = (struct _IO_FILE_plus *) 0x7ffff7dd5540 <_IO_2_1_stderr_>
pwndbg>

```

第二部分就是调用 `_IO_file_close_it` 关闭文件，释放缓冲区，并清空缓冲区指针。跟进去该函数，文件在 `/libio/fileops.c` 中：

这个函数也做了很多事情，首先是调用 `_IO_file_is_open` 宏检查该文件是否处于打开的状态，宏的定义为 `#define _IO_file_is_open(__fp) ((__fp)->_fileno != -1)`，只是简单的判断 `_fileno`。

```
#if defined _LIBC || defined _GLIBCXX_USE_WCHAR_T
# define _IO_do_flush(_f) \
(((_f)->_mode <= 0 \
 ? _IO_do_write(_f, (_f)->_IO_write_base, \
    (_f)->_IO_write_ptr-(_f)->_IO_write_base) \
 : _IO_wdo_write(_f, (_f)->_wide_data->_IO_write_base, \
    ((_f)->_wide_data->_IO_write_ptr \
    - (_f)->_wide_data->_IO_write_base)))
```

可以看到它对应的是调用 `_IO_do_write` 函数去输出此时的输出缓冲区，`_IO_do_write` 函数已经在 `fwrite` 这篇文章中跟过了，主要的作用就是调用系统调用输出缓冲区。

```
pwndbg> print *fp
$8 = {
  _flags = 0xfbad2c04,
  _IO_read_ptr = 0x603250 "",
  _IO_read_end = 0x603250 "",
  _IO_read_base = 0x603250 "",
  _IO_write_base = 0x603250 "",
  _IO_write_ptr = 0x603250 "",
  _IO_write_end = 0x604250 "",
  _IO_buf_base = 0x603250 "",
  _IO_buf_end = 0x604250 "",
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _markers = 0x0,
  _chain = 0x7ffff7dd5540 <_IO_2_1_stderr>,
  _fileno = 0x3,
  _flags2 = 0x0,
  _old_offset = 0x0,
  _cur_column = 0x0,
  _vtable_offset = 0x0,
  _shortbuf = "",
  _lock = 0x603100,
  _offset = 0xffffffffffffffff,
  _codecvt = 0x0,
  _wide_data = 0x603110,
  _freeres_list = 0x0,
  _freeres_buf = 0x0,
  __pad5 = 0x0,
  _mode = 0xffffffff,
  _unused2 = '\000' <repeats 19 times>
}
```

回到 `_IO_new_file_close_it` 函数中，可以看到在调用了 `_IO_do_flush` 后，代码调用了 `_IO_SYSCLOSE` 函数，该函数是 `vtable` 中的 `__close` 函数，跟进去该函数，在

```

int
_IO_file_close (_IO_FILE *fp)
{
    /* Cancelling close should be avoided if possible since it leaves an
       unrecoverable state behind.  */
    return close_not_cancel (fp->_fileno);
}
libc_hidden_def (_IO_file_close)

```

close\_not\_cancel的定义为#define close\_not\_cancel(fd) \ \_\_close (fd)该函数直接调用了系统调用close关闭文件描述符。

在调用了\_IO\_SYSCLOSE函数关闭文件描述符后，\_IO\_new\_file\_close\_it函数开始释放输入输出缓冲区并置零输入输出缓冲区。一口气调用了\_IO\_setb、\_IO\_setg

```

void
_IO_setb (_IO_FILE *f, char *b, char *eb, int a)
{
    if (f->_IO_buf_base && !(f->_flags & _IO_USER_BUF))
        free (f->_IO_buf_base); //■■■■■
    f->_IO_buf_base = b;
    f->_IO_buf_end = eb;
    if (a)
        f->_flags &= ~_IO_USER_BUF;
    else
        f->_flags |= _IO_USER_BUF;
}
libc_hidden_def (_IO_setb)

```

可以看到在 `_IO_setb` 释放的缓冲区，并置零了 `buf` 指针。找到了释放缓冲区的地方了，之前看 `fread` 和 `fwrite` 的时候都没注意到这里。执行完这一段之后，指针被清零了：

```
pwndbg> print *fp
$10 = {
  _flags = 0xfbad2c05,
  _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 = 0x7ffff7dd5540 <_IO_2_1_stderr_>,
  _fileno = 0x3,
  _flags2 = 0x0,
  _old_offset = 0x0,
  _cur_column = 0x0,
  _vtable_offset = 0x0,
  _shortbuf = "",
  _lock = 0x603100,
  _offset = 0xffffffffffffffff,
  _codecvt = 0x0,
  _wide_data = 0x603110,
  _freeres_list = 0x0,
  _freeres_buf = 0x0,
  __pad5 = 0x0,
  _mode = 0xffffffff,
  _unused2 = '\000' <repeats 19 times>
}
```

继续往下看，其调用了 `_IO_un_link` 函数，确保结构体从 `_IO_list_all` 链表中取了下来。然后将文件描述符设置为 -1。

## 释放FILE内存以及确认文件关闭

结束\_io\_file\_close\_it函数后，程序回到\_io\_new\_fclose中，开始第三部分代码，调用\_io\_finish进行最后的确认，跟进去该函数，该函数是vtable中的\_\_finish

```
void
_io_new_file_finish (_IO_FILE *fp, int dummy)
{
    if (_IO_file_is_open (fp))
    {
        _IO_do_flush (fp);
        if (!(fp->_flags & _IO_DELETE_DONT_CLOSE))
            _IO_SYSCLOSE (fp);
    }
    _IO_default_finish (fp, 0);
}
libc_hidden_ver (_IO_new_file_finish, _IO_file_finish)
```

可以看到代码首先检查了文件描述符是否打开，在第二步中已经将其设置为-1，所以不会进入该流程。如果文件打开的话则会调用\_io\_do\_flush和\_io\_sysclose刷新缓

接着调用\_io\_default\_finish确认缓冲区确实被释放，以及结构体从\_io\_list\_all中取了下来，并设置指针，函数源码在libio/genops.c中：

```
void
_io_default_finish (_IO_FILE *fp, int dummy)
{
    struct _IO_marker *mark;
    if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
    {
        free (fp->_IO_buf_base);
        fp->_IO_buf_base = fp->_IO_buf_end = NULL;
    }

    for (mark = fp->_markers; mark != NULL; mark = mark->_next)
        mark->_sbuf = NULL;

    if (fp->_IO_save_base)
    {
        free (fp->_IO_save_base);
        fp->_IO_save_base = NULL;
    }

    _IO_un_link ((struct _IO_FILE_plus *) fp);
}
libc_hidden_def (_IO_default_finish)
```

感觉\_io\_finish函数并没有做什么操作，都是之前已经进行过的，有些冗余。

程序回到\_io\_new\_fclose中，到此时已经将结构体从链表中删除，刷新了缓冲区，释放了缓冲区内内存，只剩下结构体内内存尚未释放，因此代码也剩下最后一段代码，即调

到此，源码分析结束。

## 小结

分析完成后，回头看fclose函数的功能，主要就是刷新输出缓冲区并释放缓冲区内内存、释放结构体内内存。仍然总结下调用了vtable中的函数：

- 在清空缓冲区的\_io\_do\_write函数中会调用vtable中的函数。
- 关闭文件描述符\_io\_sysclose函数为vtable中的\_\_close函数。
- \_io\_finish函数为vtable中的\_\_finish函数。

fclose函数分析完成后，对于IO

FILE源码分析的主体部分就完成了，后续会进入利用的部分。主要包括通过虚表的劫持来控制程序流、vtable的检查以及绕过、通过结构体的指针实现内存读写等技巧。

点击收藏 | 0 关注 | 1

[上一篇：内核漏洞挖掘技术系列\(5\)——Ke...](#) [下一篇：CVE-2019-0948：Mic...](#)

1. 0 条回复

- 动动手指，沙发就是你的了！

[登录](#) 后跟帖



[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](#)