

上篇文章描述了vtable check以及绕过vtable check的方法之一，利用vtable段中的_IO_str_jumps来进行FSOP。本篇则主要描述使用缓冲区指针来进行任意内存读写。

从前面fread以及fwrite的分析中，我们知道了FILE结构体中的缓冲区指针是用来进行输入输出的，很容易的就想到了如果能伪造这些缓冲区指针，在一定的条件下应该

本文包括两部分：

- 使用stdin标准输入缓冲区进行任意地址写。
- 使用stdout标准输出缓冲区进行任意地址读写。

接下来描述这两部分的原理以及给出相应的题目实践，原理介绍部分是基于已经拥有可以伪造IO FILE结构体的缓冲区指针漏洞的基础上进行的。在后续过程假设我们目标写的地址是write_start，写结束地址为write_end；读的目标地址为read_start，读的结束地

前几篇传送门：

- [IO FILE之fopen详解](#)
- [IO FILE之fread详解](#)
- [IO FILE之fwrite详解](#)
- [IO FILE之fclose详解](#)
- [IO FILE之劫持vtable及FSOP](#)
- [IO FILE 之vtable劫持以及绕过](#)

stdin标准输入缓冲区进行任意地址写

这一部分主要阐述的是使用stdin标准输入缓冲区指针进行任意地址写的功能。

原理分析

先通过fread回顾下通过输入缓冲区进行输入的流程：

1. 判断fp->_IO_buf_base输入缓冲区是否为空，如果为空则调用的_IO_doallocbuf去初始化输入缓冲区。
2. 在分配完输入缓冲区或输入缓冲区不为空的情况下，判断输入缓冲区是否存在数据。


```

    if (have > 0)
    {
        ...
        //memcpy

    }

    if (fp->_IO_buf_base
        && want < (size_t) (fp->_IO_buf_end - fp->_IO_buf_base))
    {
        if (__underflow (fp) == EOF) ## ███__underflow█████
        ...
    }
    ...
    return n - want;
}

```

上面贴出了一些关键代码，首先是 `_IO_file_xsgetn` 函数，函数先判断输入缓冲区 `_IO_buf_base` 是否为空，如果为空的话则调用 `_IO_doallocbuf` 初始化缓冲区，因此

接着函数中当输入缓冲区有剩余时即 `_IO_read_end - _IO_read_ptr`

>0，会将缓冲区中的数据拷贝至目标中，因此想要利用输入缓冲区实现读写，最好使 `_IO_read_end - _IO_read_ptr = 0` 即 `_IO_read_end == _IO_read_ptr`。

同时还要求读入的数据 `size` 要小于缓冲区数据的大小，否则为提高效率会调用 `read` 直接读。

`_IO_file_xsgetn` 函数中当缓冲区不能满足需求时会调用 `__underflow` 去读取数据，查看 `__underflow`。

```

int
_IO_new_file_underflow (_IO_FILE *fp)
{
    _IO_ssize_t count;
    ...
    ## █████_IO_NO_READS██████████████
    if (fp->_flags & _IO_NO_READS)
    {
        fp->_flags |= _IO_ERR_SEEN;
        __set_errno (EBADF);
        return EOF;
    }
    ## ████████████████████████████
    if (fp->_IO_read_ptr < fp->_IO_read_end)
        return *(unsigned char *) fp->_IO_read_ptr;
    ...

    ##███_IO_SYSREAD██████████████████
    count = _IO_SYSREAD (fp, fp->_IO_buf_base,
                        fp->_IO_buf_end - fp->_IO_buf_base);
    ...

}
libc_hidden_ver (_IO_new_file_underflow, _IO_file_underflow)

```

在 `_IO_new_file_underflow` 函数中先判断 `fp->_IO_read_ptr <`

`fp->_IO_read_end` 是否成立，成立则直接返回，因此再次要求伪造的结构体 `_IO_read_end == _IO_read_ptr`，绕过该条件检查。

接着函数会检查 `_flags` 是否包含 `_IO_NO_READS` 标志，包含则直接返回。标志的定义是 `#define _IO_NO_READS 4`，因此 `_flags` 不能包含 4。

最终系统调用 `_IO_SYSREAD (fp, fp->_IO_buf_base, fp->_IO_buf_end -`

`fp->_IO_buf_base)` 读取数据，因此要想利用 `stdin` 输入缓冲区需设置 `FILE` 结构体中 `_IO_buf_base` 为 `write_start`，`_IO_buf_end` 为 `write_end`。同时也需将结构体 `(fp->_fileno, buf, size)` 读取数据。

将上述条件综合表述为：

1. 设置 `_IO_read_end` 等于 `_IO_read_ptr`。
2. 设置 `_flag &~ _IO_NO_READS` 即 `_flag &~ 0x4`。
3. 设置 `_fileno` 为 0。
4. 设置 `_IO_buf_base` 为 `write_start`，`_IO_buf_end` 为 `write_end`；且使得 `_IO_buf_end - _IO_buf_base` 大于 `fread` 要读的数据。

实践

实践的题目是whctf2017的stackoverflow，这一年也是这一种利用方式的兴起之年，这一题是很经典的一题。

题目首先是输入name，并把name输出出来，由于name未进行初始化设置且读取数据后未加入\x00，可以由此泄露出libc地址。

接着进入主功能函数，漏洞在先使用temp变量保存了输入的size，但是后续最后写\x00的时候使用的是temp，而不是size，因此存在一个溢出写\x00的漏洞。

在之前的文章中，我们知道了当申请堆块大小很大时（0x200000），申请出来的堆块会紧挨着libc，因此我们可以利用这个溢出写\x00的漏洞往libc的内存中写入一个\x00

往哪里写一个\x00字节，后续改变整个内存结构而拿到shell？答案时stdin结构体中的\x00，我们先看下输入之前的stdin结构体中的数据：

```
pwndbg> print *stdin
$1 = {
  _flags = 0xfbad208b,
  _IO_read_ptr = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_read_end = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_read_base = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_write_base = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_write_ptr = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_write_end = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_buf_base = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
  _IO_buf_end = 0x7fbaa40de944 <_IO_2_1_stdin_+132> "",
  _IO_save_base = 0x0,
  _IO_backup_base = 0x0,
  _IO_save_end = 0x0,
  _markers = 0x0,
  _chain = 0x0,
  _fileno = 0x0,
  _flags2 = 0x0,
  _old_offset = 0xffffffffffffffff,
  _cur_column = 0x0,
  _vtable_offset = 0x0,
  _shortbuf = "",
  _lock = 0x7fbaa40e0770 <_IO_stdfile_0_lock>,
  _offset = 0xffffffffffffffff,
  _codecvt = 0x0,
  _wide_data = 0x7fbaa40de9a0 <_IO_wide_data_0>,
  _freeres_list = 0x0,
  _freeres_buf = 0x0,
  __pad5 = 0x0,
  _mode = 0x0,
  _unused2 = '\000' <repeats 19 times>
}
```



```

pwndbg> x/20gx &*stdin
0x7fbaa40de8c0 <_IO_2_1_stdin>:      0x00000000fbad208b      0x00007fbaa40de943
0x7fbaa40de8d0 <_IO_2_1_stdin+16>:  0x00007fbaa40de943      0x00007fbaa40de943
0x7fbaa40de8e0 <_IO_2_1_stdin+32>:  0x00007fbaa40de943      0x00007fbaa40de943
0x7fbaa40de8f0 <_IO_2_1_stdin+48>:  0x00007fbaa40de943      0x00007fbaa40de943
0x7fbaa40de900 <_IO_2_1_stdin+64>:  0x00007fbaa40de944      0x0000000000000000
0x7fbaa40de910 <_IO_2_1_stdin+80>:  0x0000000000000000      0x0000000000000000
0x7fbaa40de920 <_IO_2_1_stdin+96>:  0x0000000000000000      0x0000000000000000
0x7fbaa40de930 <_IO_2_1_stdin+112>: 0x0000000000000000      0xffffffffffffffff
0x7fbaa40de940 <_IO_2_1_stdin+128>: 0x0000000000000000      0x00007fbaa40e0770
0x7fbaa40de950 <_IO_2_1_stdin+144>: 0xffffffffffffffff      0x0000000000000000

```

可以看到在glibc

我们可将 `IO buf end` 覆盖为 `malloc hook+0x8`，则输入时最后控制写的数据为 `stdin` 中的 `IO buf end` 指针位置到 `malloc hook+0x8`，以实现控制 `malloc`

一是IO_getc函数的作用是刷新_IO_read_ptr，每次会从输入缓冲区读一个字节数据即将_IO_read_ptr加一，当_IO_read_ptr等于_IO_read_end的时候便会调用r

gadget用不了，因此在栈中找到了一个gadget，地址为0x400a23，可以读取数据形成栈溢出，从而进行ROP，拿到shell。

stdout标准输入缓冲区进行任意地址读写

任意写

想要实现上述功能，查看fwrite源码中如何才能实现该功能：

任意写功能的实现在于IO缓冲区没有满时，会先将要输出的数据复制到缓冲区中，可通过这一点来实现任意地址写的功能。可以看到任意写好像很简单，只需将 `_IO_write`

利用stdout进行任意地址读的原理为：控制输出缓冲区指针指向我们输入的地址，构造好条件，使得输出缓冲区为已经满的状态，再次调用输出函数时，程序会刷新输出缓

```

_IO_size_t
_IO_new_file_xsputn (_IO_FILE *f, const void *data, _IO_size_t n)
{

```

```

_IO_size_t count = 0;
...
## 
else if (f->_IO_write_end > f->_IO_write_ptr)
    count = f->_IO_write_end - f->_IO_write_ptr; /* Space available. */

## 
if (count > 0)
{
    ...
    //memcpy
}
if (to_do + must_flush > 0)
{
    if (_IO_OVERFLOW (f, EOF) == EOF)

```

当f->_IO_write_end > f->_IO_write_ptr时，会调用memcpy拷贝数据，因此最好构造条件f->_IO_write_end等于f->_IO_write_ptr。

接着进入_IO_OVERFLOW函数，去刷新输出缓冲区，跟进去：

```

int
_IO_new_file_overflow (_IO_FILE *f, int ch)
{
    ## 
    if (f->_flags & _IO_NO_WRITES) /* SET ERROR */
    {
        f->_flags |= _IO_ERR_SEEN;
        __set_errno (EBADF);
        return EOF;
    }

    ## 
    if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
    {
        ...
    }

    ## 
    if (ch == EOF)
        return _IO_do_write (f, f->_IO_write_base,
                             f->_IO_write_ptr - f->_IO_write_base);
    return (unsigned char) ch;
}
libc_hidden_ver (_IO_new_file_overflow, _IO_file_overflow)

```

可以看到_IO_new_file_overflow，首先判断_flags是否包含_IO_NO_WRITES，如果包含则直接返回，因此需构造_flags不包含_IO_NO_WRITES，其定义为#define _IO_NO_WRITES 8；

接着判断缓冲区是否为空以及是否不包含_IO_CURRENTLY_PUTTING标志位，如果不包含的话则做一些多余的操作，可能不可控，因此最好定义_flags包含_IO_CURRENTLY_PUTTING 0x800。

接着调用_IO_do_write去输出输出缓冲区，其传入的参数是f->_IO_write_base，大小为f->_IO_write_ptr - f->_IO_write_base。因此若想实现任意地址读，应构造_IO_write_base为read_start，构造_IO_write_ptr为read_end。

跟进去_IO_do_write，看该函数的关键代码：

```

static
_IO_size_t
new_do_write (_IO_FILE *fp, const char *data, _IO_size_t to_do)
{
    ...
    _IO_size_t count;
    if (fp->_flags & _IO_IS_APPENDING)
        fp->_offset = _IO_pos_BAD;
    else if (fp->_IO_read_end != fp->_IO_write_base)
    {
        _IO_off64_t new_pos
        = _IO_SYSSEEK (fp, fp->_IO_write_base - fp->_IO_read_end, 1);
        if (new_pos == _IO_pos_BAD)
            return 0;
    }
}

```



```
0x7f0bcf15d85b <__printf_chk+107>    mov     eax, dword ptr [rbx]
0x7f0bcf15d85d <__printf_chk+109>    and     eax, 0x8000
0x7f0bcf15d862 <__printf_chk+114>    jne     __printf_chk+202 <0x7f0bcf15d8ba>
```

小结

使用IO

FILE来进行任意内存读写真的是个很强大的功能，构造起来也比较容易。但是对于FILE结构体的伪造，个人感觉可能最容易出问题的地方还是_flags字段的构造，可能某个

至此IO FILE系列描述完毕，前四篇对IO函数fopen、fread、fwrite以及fclose的源码分析；后面三篇介绍了针对IO FILE的相关利用，包括劫持vtable、vtable引入的check机制以及相应的后续利用方式。在整个过程中为方便构造IO结构体还在pwn_debug中加入了IO_FILE_plus模块。

最后一句，阅读源码对于学习是一件很有帮助的事情。

相关文件及脚本[链接](#)

参考链接

1. [HCTF 2018 部分 PWN writeup--babyprinf_ver2](#)
2. [浅析IO_FILE结构及利用](#)
3. [教练！那根本不是IO！——从printf源码看libc的IO](#)

点击收藏 | 0 关注 | 1

[上一篇：一次Blind-XXE漏洞挖掘之旅](#) [下一篇：Linux Kernel Expl...](#)

1. 1 条回复



[Ex](#) 2019-08-10 18:47:24

很详细的文章，感谢大佬分享

0 回复Ta

[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](#)