氷菓

TUTORIAL

Advanced Heap Exploitation: File Stream Oriented Programming

- JANUARY 1, 2018
- DANGOKYO
- LEAVE A COMMENT

Introduction

In this post, I will give a detailed introduction of File Stream Oriented Programming, including the internal implementation on file structure, related file operation and corresponding exploitation techniques in CTF. This post is based on the source code of glibc-2.26. Since this post is for newbies interested in CTF challenges, I will add many implementation details based on source code. I write this post following the lecture notes given by [1].

Data Structure in File

Fist of all, we need to explain the data structure in file processing.

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

  /* The following pointers correspond to the C++ streambuf protocol. */
  /* Note:  Tk uses the _IO_read_ptr and _IO_read_end fields directly. */
  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. */

  struct _IO_marker *_markers;

  struct _IO_FILE *_chain;

  int _fileno;
#if 0
  int _blksize;
#else
  int _flags2;
#endif
  _IO_off_t _old_offset; /* This used to be _offset but it's too small.  */

#define __HAVE_COLUMN /* temporary */
  /* 1+column number of pbase(); 0 is unknown. */
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];

  /*  char* _save_gptr;  char* _save_egptr; */

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

In libc, all *_IO_FILE* structures all linked via a singly linked list. Pointer **_chain* points to the next *_IO_FILE* structure in the list. Furthermore, the head of the linked list is stored in *_IO_list_all*. A typical memory layout of the linked list is shown below:

```
0x7f0cc0434500 <_IO_list_all>:    0x00007f0cc0434520  0x0000000000000000
0x7f0cc0434510: 0x0000000000000000  0x0000000000000000
0x7f0cc0434520 <_IO_2_1_stderr_>: 0x00000000fbad2087  0x00007f0cc04345a3
0x7f0cc0434530 <_IO_2_1_stderr_+16>:  0x00007f0cc04345a3  0x00007f0cc04345a3
0x7f0cc0434540 <_IO_2_1_stderr_+32>:  0x00007f0cc04345a3  0x00007f0cc04345a3
0x7f0cc0434550 <_IO_2_1_stderr_+48>:  0x00007f0cc04345a3  0x00007f0cc04345a3
0x7f0cc0434560 <_IO_2_1_stderr_+64>:  0x00007f0cc04345a4  0x0000000000000000
0x7f0cc0434570 <_IO_2_1_stderr_+80>:  0x0000000000000000  0x0000000000000000
0x7f0cc0434580 <_IO_2_1_stderr_+96>:  0x0000000000000000  0x00007f0cc0434600
0x7f0cc0434590 <_IO_2_1_stderr_+112>: 0x0000000000000002  0xffffffffffffffff
0x7f0cc04345a0 <_IO_2_1_stderr_+128>: 0x0000000000000000  0x00007f0cc0435750
0x7f0cc04345b0 <_IO_2_1_stderr_+144>: 0xffffffffffffffff  0x0000000000000000
0x7f0cc04345c0 <_IO_2_1_stderr_+160>: 0x00007f0cc0433640  0x0000000000000000
0x7f0cc04345d0 <_IO_2_1_stderr_+176>: 0x0000000000000000  0x0000000000000000
0x7f0cc04345e0 <_IO_2_1_stderr_+192>: 0x0000000000000000  0x0000000000000000
0x7f0cc04345f0 <_IO_2_1_stderr_+208>: 0x0000000000000000  0x00007f0cc0430400
0x7f0cc0434600 <_IO_2_1_stdout_>: 0x00000000fbad2887  0x00007f0cc0434683

//And the data of _IO_2_1_stderr_ can be interpreted as:
_flags = 0xfbad2087,
_IO_read_ptr = 0x7f0cc04345a3,
_IO_read_end = 0x7f0cc04345a3,
_IO_read_base = 0x7f0cc04345a3,
_IO_write_base = 0x7f0cc04345a3,
_IO_write_ptr = 0x7f0cc04345a3,
_IO_write_end = 0x7f0cc04345a3,
_IO_buf_base = 0x7f0cc04345a3,
_IO_buf_end = 0x7f0cc04345a4,
_IO_save_base = 0x0,
_IO_backup_base = 0x0,
_IO_save_end = 0x0,
_markers = 0x0,
_chain = 0x7f0cc0434600,  //point to _IO_2_1_stdout_
_fileno = 0x2,
_flags2 = 0x0,
_old_offset = 0xffffffffffffffff,
_cur_column = 0x0,
_vtable_offset = 0x0,
_shortbuf = {0x0},
_lock = 0x7f0cc0435750,
_offset = 0xffffffffffffffff,
_codecvt = 0x0,
_wide_data = 0x7f0cc0433640,
_freeres_list = 0x0,
_freeres_buf = 0x0,
__pad5 = 0x0,
_mode = 0x0,
_unused2 = {0x0 <repeats 20 times>}
```

Besides *_IO_FILE*, another important data structure is *_IO_FILE_plus*. It maintains a vtable-like data structure *_IO_jump_t*. Each operation on a file is done via the function pointer stored in the table.

```c
struct _IO_FILE_plus
{
  _IO_FILE file;
  const struct _IO_jump_t *vtable;
};

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);
#if 0
    get_column;
    set_column;
#endif
};
```

File Operation

In [1], it gives the workflow of *fopen*, *fread*, *fwrite* and *fclose*. Here I may give a detailed explanation on *fopen*, *fread* and *fwrite*.

Function **fopen**

Function *__fopen_internal* is the internal implementation of *fopen*. In this function, it will create a *locked_FILE* object of the target file and initialise the file descriptor.

```c
#   define fopen(fname, mode) _IO_new_fopen (fname, mode)

```

```c
_IO_FILE * _IO_new_fopen (const char *filename, const char *mode)
{
  return __fopen_internal (filename, mode, 1);
}

_IO_FILE *
__fopen_internal (const char *filename, const char *mode, int is32)
{
  struct locked_FILE
  {
    struct _IO_FILE_plus fp;
#ifdef _IO_MTSAFE_IO
    _IO_lock_t lock;
#endif
    struct _IO_wide_data wd;
  }
  *new_f = (struct locked_FILE *) malloc (sizeof (struct locked_FILE));

  if (new_f == NULL)
    return NULL;
#ifdef _IO_MTSAFE_IO
  new_f->fp.file._lock = &new_f->lock;
#endif
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
  _IO_no_init (&new_f->fp.file, 0, 0, &new_f->wd, &_IO_wfile_jumps);
#else
  _IO_no_init (&new_f->fp.file, 1, 0, NULL, NULL);
#endif
  _IO_JUMPS (&new_f->fp) = &_IO_file_jumps;
  _IO_new_file_init_internal (&new_f->fp);
  if (_IO_file_fopen ((_IO_FILE *) new_f, filename, mode, is32) != NULL)
    return __fopen_maybe_mmap (&new_f->fp.file);

  _IO_un_link (&new_f->fp);
  free (new_f);
  return NULL;
}

_IO_new_file_init_internal (struct _IO_FILE_plus *fp)
{
  /* POSIX.1 allows another file handle to be used to change the position
     of our file descriptor.  Hence we actually don't know the actual
     position before we do the first fseek (and until a following fflush). */
  fp->file._offset = _IO_pos_BAD;
  fp->file._IO_file_flags |= CLOSED_FILEBUF_FLAGS;

  _IO_link_in (fp);
  fp->file._fileno = -1;
}

void _IO_link_in (struct _IO_FILE_plus *fp)
{
  if ((fp->file._flags & _IO_LINKED) == 0)
    {
      fp->file._flags |= _IO_LINKED;
```

```
      fp->file._chain = (_IO_FILE *) _IO_list_all;
      _IO_list_all = fp;
      ++_IO_list_all_stamp;
    }
}
```

(1) Allocate a *locked_file*.
(2) Invoke function *_IO_new_file_init_internal*. In this function, the newly allocated *fp* will be inserted into the singly linked list.
(3) Invoke syscall fopen to get a file descriptor of the target file and assign the file descriptor number to the *fp->fileno*.

Function **File_underflow**

Before introducing the internal implementation of **fread** and **fwrite**. We need to go through function **_IO_new_file_underflow and _IO_new_file_overflow**. These two functions are important in fread/fwrite respectively. And the exploitation techniques used in Ghost In The Heap (https://dangokyo.me/2017/12/16/hitcon-2017-ctf-quals-ghost-in-the-heap/) will involve *_IO_new_file_underflow*.

```
int _IO_new_file_underflow (_IO_FILE *fp)
{
  _IO_ssize_t count;

  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 active
	 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;
}
```

(1) Check the status of fp. If the file descriptor is not readable return error.
(2) If *fp->_IO_buf_base* is NULL pointer, do allocate buffer. Set member variable of *fp*: *fp->_IO_buf_base* and *fp->_IO_buf_end*.

(3) Update the buffer pointer in current file descriptor.
(4) Invoke syscall read, copy *fp->_IO_buf_end – fp->_IO_buf_base* bytes to *fp->_IO_buf_base*. Update *fp->_IO_read_end* to *fp->_IO_read_end* + *count*.

```
int
_IO_new_file_overflow (_IO_FILE *f, int ch)
{
  if (f->_flags & _IO_NO_WRITES) /* SET ERROR */
  {
      f->_flags |= _IO_ERR_SEEN;
      __set_errno (EBADF);
      return EOF;
  }
  /* If currently reading or no buffer allocated. */
  if ((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
  {
      /* Allocate a buffer if needed. */
      if (f->_IO_write_base == NULL)
      {
      _IO_doallocbuf (f);
      _IO_setg (f, f->_IO_buf_base, f->_IO_buf_base, f->_IO_buf_base);
      }
      /* Otherwise must be currently reading.
     If _IO_read_ptr (and hence also _IO_read_end) is at the buffer end,
     logically slide the buffer forwards one block (by setting the
     read pointers to all point at the beginning of the block).  This
     makes room for subsequent output.
     Otherwise, set the read pointers to _IO_read_end (leaving that
     alone, so it can continue to correspond to the external position). */
      if (__glibc_unlikely (_IO_in_backup (f)))
      {
      size_t nbackup = f->_IO_read_end - f->_IO_read_ptr;
      _IO_free_backup_area (f);
      f->_IO_read_base -= MIN (nbackup,
                   f->_IO_read_base - f->_IO_buf_base);
      f->_IO_read_ptr = f->_IO_read_base;
      }

      if (f->_IO_read_ptr == f->_IO_buf_end)
    f->_IO_read_end = f->_IO_read_ptr = f->_IO_buf_base;
      f->_IO_write_ptr = f->_IO_read_ptr;
      f->_IO_write_base = f->_IO_write_ptr;
      f->_IO_write_end = f->_IO_buf_end;
      f->_IO_read_base = f->_IO_read_ptr = f->_IO_read_end;

      f->_flags |= _IO_CURRENTLY_PUTTING;
      if (f->_mode <= 0 && f->_flags & (_IO_LINE_BUF | _IO_UNBUFFERED))
    f->_IO_write_end = f->_IO_write_ptr;
  }
  if (ch == EOF)
    return _IO_do_write (f, f->_IO_write_base,
           f->_IO_write_ptr - f->_IO_write_base);
  if (f->_IO_write_ptr == f->_IO_buf_end ) /* Buffer is really full */
    if (_IO_do_flush (f) == EOF)
      return EOF;
```

```
  *f->_IO_write_ptr++ = ch;
  if ((f->_flags & _IO_UNBUFFERED) || ((f->_flags & _IO_LINE_BUF) && ch == '\n
    if (_IO_do_write (f, f->_IO_write_base,
       f->_IO_write_ptr - f->_IO_write_base) == EOF)
      return EOF;
return (unsigned char) ch;
}
```

(1) Check the status of fp. If the file descriptor is not writable return error.
(2) If *fp->_IO_write_base* is NULL pointer, do allocate buffer. Set member variable of *fp*: *fp->_IO_buf_base* and *fp->_IO_buf_end*.
(3) Update the buffer pointer in current file descriptor.
(4) Invoke syscall write, copy *fp->_IO_wrte_ptr – fp->_IO_write_base* bytes to *fp->_IO_write_ptr*.

Function **fread**

Function *_IO_fread* is the internal function of *fread*. In this function, it will first calculate the total bytes to be read into buffer and then invoke *_IO_file_xsgetn* for the following steps.

```
//Internal function of fread
_IO_size_t _IO_fread (void *buf, _IO_size_t size, _IO_size_t count, _IO_FILE
{
  _IO_size_t bytes_requested = size * count;
  _IO_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;
}

_IO_size_t _IO_file_xsgetn (_IO_FILE *fp, void *data, _IO_size_t n)
{
  _IO_size_t want, have;
  _IO_ssize_t count;
  char *s = data;

  want = n;

  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);
  }
```

```c

  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)
      {
#ifdef _LIBC
          s = __mempcpy (s, fp->_IO_read_ptr, have);
#else
          memcpy (s, fp->_IO_read_ptr, have);
          s += have;
#endif
          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;

          continue;
      }

      /* These must be set before the sysread as we might longjmp out
         waiting for input. */
      _IO_setg (fp, fp->_IO_buf_base, fp->_IO_buf_base, fp->_IO_buf_base);
      _IO_setp (fp, fp->_IO_buf_base, fp->_IO_buf_base);

      /* Try to maintain alignment: read a whole number of blocks.  */
      count = want;
      if (fp->_IO_buf_base)
      {
          _IO_size_t block_size = fp->_IO_buf_end - fp->_IO_buf_base;
          if (block_size >= 128)
        count -= want % block_size;
```

```
      }

      count = _IO_SYSREAD (fp, s, count);
      if (count <= 0)
      {
          if (count == 0)
        fp->_flags |= _IO_EOF_SEEN;
          else
        fp->_flags |= _IO_ERR_SEEN;

          break;
      }

      s += count;
      want -= count;
      if (fp->_offset != _IO_pos_BAD)
        _IO_pos_adjust (fp->_offset, count);
       }
  }
  return n - want;
}
```

(1) If *fp->_IO_buf_base* is NULL pointer, do allocate buffer. Set member variable of *fp*: *fp->_IO_buf_base* and *fp->_IO_buf_end*.
(2) Set *want* to the requested size. Set *have* to *fp->_IO_read_buf – fp->_IO_read_end*.
(3) If *want* is less than *have*, read *want* bytes of data from *fp->_IO_read_buf* into target buffer. Otherwise go to step 4.
(4) If *have* is larget than zero, read *have* bytes of data from *fp->_IO_read_buf* into target buffer.
(5) If *fp->_IO_buf_end* is not null and *want* is less than *fp->_IO_buf_end – fp->_IO_buf_base*, invoke function *_IO_new_file_underflow* to read data into *fp->_IO_buf_base* and then go to step (2). Otherwise go to step (4)
(6) Invoke syscall read to read requested bytes of data into target buffer.

Function **fwrite**

Function *_IO_fwrite* is the internal implementation of *fread*. In this function, it will first calculate the total bytes to be written into file and then invoke *_IO_new_file_xsputn* for the following steps.

```
_IO_size_t _IO_fwrite (const void *buf, _IO_size_t size, _IO_size_t count, _IO
{
  _IO_size_t request = size * count;
  _IO_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;
}

_IO_size_t _IO_new_file_xsputn (_IO_FILE *f, const void *data, _IO_size_t n)
{
  const char *s = (const char *) data;
  _IO_size_t to_do = n;
  int must_flush = 0;
  _IO_size_t count = 0;

  if (n <= 0)
    return 0;
  /* This is an optimized implementation.
     If the amount to be written straddles a block boundary
     (or the filebuf is unbuffered), use sys_write directly. */

  /* 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;
          }
       }
       }
  }
  else if (f->_IO_write_end > f->_IO_write_ptr)
    count = f->_IO_write_end - f->_IO_write_ptr; /* Space available. */

  /* Then fill the buffer. */
  if (count > 0)
  {
      if (count > to_do)
    count = to_do;
#ifdef _LIBC
      f->_IO_write_ptr = __mempcpy (f->_IO_write_ptr, s, count);
#else
      memcpy (f->_IO_write_ptr, s, count);
      f->_IO_write_ptr += count;
#endif
      s += count;
      to_do -= count;
```

```
    }
    if (to_do + must_flush > 0)
    {
        _IO_size_t block_size, do_write;
        /* Next flush the (full) buffer. */
        if (_IO_OVERFLOW (f, EOF) == EOF)
      /* If nothing else has to be written we must not signal the
         caller that everything has been written.  */
      return to_do == 0 ? EOF : n - to_do;

        /* Try to maintain alignment: write a whole number of blocks.  */
        block_size = f->_IO_buf_end - f->_IO_buf_base;
        do_write = to_do - (block_size >= 128 ? to_do % block_size : 0);

        if (do_write)
        {
        count = new_do_write (f, s, do_write);
        to_do -= count;
        if (count < do_write)
          return n - to_do;
        }

        /* Now write out the remainder.  Normally, this will fit in the
       buffer, but it's somewhat messier for line-buffered files,
       so we let _IO_default_xsputn handle the general case. */
        if (to_do)
      to_do -= _IO_default_xsputn (f, s+do_write, to_do);
    }
    return n - to_do;
  }
```

(1) Set *to_do* to requested bytes. Set *count* to the available space in write buffer.
(2) If *count* is larger than *to_do*, copy *to_do* bytes of data into *f->_IO_write_ptr*
(3) Invoke function *_IO_new_file_overflow* to write data into file. If it reaches the end of file, return. Otherwise, go to step (4).
(4) Invoke syscall write to write data into file.

Exploitation Technique

In unsorted bin attack, we gain a write-something-anywhere primitive. In 0CTF 2017 Babyheap (https://dangokyo.me/2017/12/11/0ctf-2017-quals-pwn-babyheap-write-up/), we used unsorted bin attack to corrupt the *global_max_fast* and used fastbin attack to hijack control flow. What if there were limitation on the times of allocation that make fastbin attack impossible?
Here we are going to give two exploitation techniques in FSOP. The first is the attack on **_IO_list_all** used in House of Orange. The second one is the attack on **_IO_2_1_stdin_->_IO_buf_end**. In both techniques, attacker does not need to allocate multiple chunks to hijack control flow. On the contrary, both techniques try to hijack control flow in one allocation.

Attack on **_IO_list_all**

As explained in previous section, *_IO_list_all* is the head of a linked list that contains all *_IO_FILE* structures. So let's discuss what will happen if *_IO_list_all* is corrupted.

```
#define fflush(s) _IO_flush_all_lockp (0)

fp = (_IO_FILE *) _IO_list_all;
while (fp != NULL)
{
      run_fp = fp;
      if (do_lock)
    _IO_flockfile (fp);

      if (((fp->_mode <= 0 && fp->_IO_write_ptr > fp->_IO_write_base)
#if defined _LIBC || defined _GLIBCPP_USE_WCHAR_T
       || (_IO_vtable_offset (fp) == 0
           && fp->_mode > 0 && (fp->_wide_data->_IO_write_ptr
                    > fp->_wide_data->_IO_write_base))
#endif
       )
      && _IO_OVERFLOW (fp, EOF) == EOF)
    result = EOF;

      if (do_lock)
    _IO_funlockfile (fp);
      run_fp = NULL;

      if (last_stamp != _IO_list_all_stamp)
      {
      /* Something was added to the list.  Start all over again.  */
      fp = (_IO_FILE *) _IO_list_all;
      last_stamp = _IO_list_all_stamp;
      }
      else
    fp = fp->_chain;
}
```

In the abort routing of libc, function *fflush* will be invoked and function *_IO_flush_all_lockp* will be implicitly invoked. In this function, it will traverse all the **_IO_FILE_plus** objects in the linked list and trigger _IO_OVERFLOW of each object.
According to the process above, there are two potential variables that could be corrupted for exploitation. The first variable is the *vtable* pointer of the first object in the linked list. If attacker can craft a virtual table in memory, the attacker can hijack the control flow. The second variable is the *chain* pointer of the first object in the linked list. Then attacker craft a fake *_IO_FILE_plus* object in memory and craft a *vtable* and hijack control flow in *_IO_OVERFLOW* on the following object .
In unsorted bin attacker, attacker can write the address of unsorted bin into any place. When corrupting *_IO_lists*, we have to process on the smallbin in libc. Therefore we need to put some chunks in smallbin as crafted *vtable* pointer or *chain* pointer.

After analysing the source code of *_IO_flush_all_lockp*, crafting *chain* requires less manipulation on the smallbin and takes smaller number of deallocations.
We show the memory layout of libc after the unsorted bin attack in House of Orange.

```
// Set one_gadget address to 0x414141414141
Program received signal SIGSEGV, Segmentation fault.
0x0000414141414141 in ?? ()
// Corrupted _IO_list_all now pointing to unsorted bin
(gdb) x/4gx &_IO_list_all
0x7fa59f770520 <_IO_list_all>:    0x00007fa59f76fb78  0x0000000000000000

(gdb) x/20gx 0x00007fa59f76fb78
0x7fa59f76fb78: 0x000056133da03010  0x000056133d9e1600
0x7fa59f76fb88: 0x000056133d9e1600  0x00007fa59f770510
0x7fa59f76fb98: 0x00007fa59f76fb88  0x00007fa59f76fb88
0x7fa59f76fba8: 0x00007fa59f76fb98  0x00007fa59f76fb98
0x7fa59f76fbb8: 0x00007fa59f76fba8  0x00007fa59f76fba8
0x7fa59f76fbc8: 0x00007fa59f76fbb8  0x00007fa59f76fbb8
0x7fa59f76fbd8: 0x000056133d9e1600     [0x000056133d9e1600]<= crafted chain
0x7fa59f76fbe8: 0x00007fa59f76fbd8  0x00007fa59f76fbd8
0x7fa59f76fbf8: 0x00007fa59f76fbe8  0x00007fa59f76fbe8
0x7fa59f76fc08: 0x00007fa59f76fbf8  0x00007fa59f76fbf8

(gdb) x/20gx 0x000056133d9e1600
0x56133d9e1600: 0x0000000000000000  0x0000000000000061
0x56133d9e1610: 0x00007fa59f76fbc8  0x00007fa59f76fbc8
0x56133d9e1620: 0x0000000000000000  0x0000000000000000
0x56133d9e1630: 0x0000000000000000  0x0000000000000000
0x56133d9e1640: 0x0000000000000000  0x0000000000000000
0x56133d9e1650: 0x0000000000000000  0x0000000000000000
0x56133d9e1660: 0x0000000000000000  0x000056133d9e17d0<= crafted chain
0x56133d9e1670: 0x0000000000000000  0x0000000000000000
0x56133d9e1680: 0x0000000000000000  0x0000000000000000
0x56133d9e1690: 0x0000000000000000  0x0000000000000000

0x56133d9e17d0: 0x0000424242424242  0x0000000000000000
0x56133d9e17e0: 0x0000000000000000  0x0000000000000000
0x56133d9e17f0: 0x0000000000000000  0x0000000000000001
0x56133d9e1800: 0x0000000000000000  0x0000000000000000
0x56133d9e1810: 0x0000000000000000  0x0000000000000000
0x56133d9e1820: 0x0000000000000000  0x0000000000000000
0x56133d9e1830: 0x0000000000000000  0x0000000000000000
0x56133d9e1840: 0x0000000000000000  0x0000000000000000
0x56133d9e1850: 0x0000000000000000  0x0000000000000000
0x56133d9e1860: 0x0000000000000000  0x0000000000000000
0x56133d9e1870: 0x0000000000000000  0x0000000000000000
0x56133d9e1880: 0x0000000000000000  0x0000000000000000
0x56133d9e1890: 0x0000000000000000  0x0000000000000000
0x56133d9e18a0: 0x0000000000000000  0x000056133d9e18b8<= crafted vtable
0x56133d9e18b0: 0x0000000000000000  0x0000000000000000
0x56133d9e18c0: 0x0000000000000000  0x0000000000000000
0x56133d9e18d0: 0x0000414141414141  0x0000000000000000<= crafted virtual funct
```

Attack on **_IO_2_1_stdin_**

In glibc-2.26, a new mitigation strategy was introduced as below:

```
/* 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_vtabl
  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;
}
```

Is there any possibility to bypass the mitigation? Of course yes!
Bypass Mitigation Method 1
Let's review the code of libc and watch where IO_validate_vtable is inserted.

```
#if _IO_JUMPS_OFFSET
# define _IO_JUMPS_FUNC(THIS) \
  (IO_validate_vtable                                                   \
   (*(struct _IO_jump_t **) ((void *) &_IO_JUMPS_FILE_plus (THIS)   \
                 + (THIS)->_vtable_offset)))
# define _IO_vtable_offset(THIS) (THIS)->_vtable_offset
#else
# define _IO_JUMPS_FUNC(THIS) (IO_validate_vtable (_IO_JUMPS_FILE_plus (THIS)
# define _IO_vtable_offset(THIS) 0
#endif
#define _IO_WIDE_JUMPS_FUNC(THIS) _IO_WIDE_JUMPS(THIS)
```

It's surprising to find that *IO_validate_vtable* is only applied to _IO_JUMPS_FUNC. _IO_WIDE_JUMPS_FUNC is not taken into protection scope in *IO_validate_vtable*. So the following exploitation is to find a function to trigger _IO_WIDE_JUMPS_FUNC instead. That's the solution given in [3].

Bypass Mitigation Method 2
The official write-up given in [4] gives another bypass strategy. It finally overwrites *__malloc_hook* to hijack control flow. That is to corrupt. *_IO_stdin->_IO_buf_end*. As explained in the previous section, if *fp->_IO_buf_end – fp->_IO_buf_base* is larger than requested bytes, it will directly read requested byte of data into *fp->_IO_buf_base*. After corrupting *_IO_stdin->_IO_buf_end* to unsorted bin address, we can use function scanf to overwrite *__malloc_hook* in memory.
We show the memory layout of libc after the unsorted bin attack in Ghost in The Heap.

```
//Memory layout after unsorted bin attack
(gdb) p/x *(struct _IO_FILE*)(&_IO_2_1_stdin_)
{_flags = 0xfbad208b,
_IO_read_ptr = 0x7f59bd4989eb,
_IO_read_end = 0x7f59bd498af9,
_IO_read_base = 0x7f59bd498943,
_IO_write_base = 0x7f59bd498943,
_IO_write_ptr = 0x7f59bd498943,
_IO_write_end = 0x7f59bd498943,
_IO_buf_base = 0x7f59bd498943,
_IO_buf_end = 0x7f59bd498b58,
_IO_save_base = 0x0,
_IO_backup_base = 0x0,
_IO_save_end = 0x0,
_markers = 0x0,
_chain = 0x0,
_fileno = 0x0,
_flags2 = 0x0,
_old_offset = 0xffffffffffffffff,
_cur_column = 0x0,
_vtable_offset = 0x0,
_shortbuf = {0x0},
_lock = 0x7f59bd49a770,
_offset = 0x1b6,
_codecvt = 0x0,
_wide_data = 0x0,
_freeres_list = 0x0,
_freeres_buf = 0x0,
__pad5 = 0x0,
_mode = 0x0,
_unused2 = {0x0 <repeats 20 times>}}
```

Conclusion

In this post, we show to potential of File Stream Oriented Programming and the possibility of exploitation techniques. We can see even abort routine can also be used to exploitation. Furthermore, we demonstrate the limitation of proposed mitigation and the significance of hacking the internal implementation of common function.

Reference

[1] https://www.slideshare.net/AngelBoy1/play-with-file-structure-yet-another-binary-exploit-technique (https://www.slideshare.net/AngelBoy1/play-with-file-structure-yet-another-binary-exploit-technique)
[2] http://4ngelboy.blogspot.sg/2016/10/hitcon-ctf-qual-2016-house-of-orange.html (http://4ngelboy.blogspot.sg/2016/10/hitcon-ctf-qual-2016-house-of-orange.html)

[3] https://tradahacking.vn/hitcon-2017-ghost-in-the-heap-writeup-ee6384cd0b7 (https://tradahacking.vn/hitcon-2017-ghost-in-the-heap-writeup-ee6384cd0b7)
[4] https://github.com/scwuaptx/CTF/tree/master/2017-writeup/hitcon/ghost_in_the_heap (https://github.com/scwuaptx/CTF/tree/master/2017-writeup/hitcon/ghost_in_the_heap)