Technologic Papers

Gita's Black Box Challange

מאת אלי כהן-נחמיה

הקדמה

נתקלתי לראשונה באתגר החומרה של Gita לפני זמן מה תוך כדי שיטוט בפורום מקומי. תחת ההכרזה המתריסה כי היא מאתגרת מומחי אבטחה ישראלים, מציעה חברת מוצרי האבטחה Gita Technologies התקן חומרה אישי לפריצה. "ההתקן שיישלח אליכם הוא שלכם" מבטיחה Gita ומבקשת בתמורה כי אותם מומחים אשר נענו לאתגר ישלחו אליה את סיכומיהם ורשמיהם מתהליך המחקר. כחוקר אבטחה אשר נמצא לרוב במרחב הנוח של ארכיטקטורת אינטל, העניין סיקרן אותי במיוחד. המחשבה על חקירת התקן חומרה לא ידוע, התחקות אחר אופי פעולת ה-Firmware ולימוד ארכיטקטורה לא מוכרת קסמה לי והחלטתי להיענות לאתגר אליו נרשמתי דרך דף האתגר.

ייעודו של המסמך וקהל היעד

מכיוון שתיעוד אינטרנטי אודות האתגר לא היה בנמצא, החלטתי לערוך מחדש את הדברים שהעליתי על הכתב תוך כדי תנועה ולפרסמם בצורה מסודרת. מסמך זה אינו מיועד להיות דו"ח רשמי או ממצה, אלא תיאור מודרך של תהליך האנליזה תוך ציון השיקולים והכיוונים השונים. למסמך התווספו מספר הערות והארות בכדי להפוך אותו קריא וידידותי לחסרי הרקע החומרתי שביננו.

רשימת קבצים המצורפים לאתגר זה:

- msp430-image.bin - קובץ הכולל את תמונת הזיכרון של ה-Firmware כפי שהורדה מההתקן עצמו.
- msp430-image.i64 - קובץ מאגר נתונים של IDA הכולל ניתוח מלא של ה-Firmware.
- decode.py - סקריפט Python שנכתב לטובת יצירת הערכים הפותרים את האתגר, עבור אתגר נתון.

אזהרת ספויילר

מסמך זה כולל פתרון מלא, שלב אחר שלב, לאתגר "Gita's Black Box Challenge". המסמך דן בפתרון האתגר, בשלבים שבוצעו על מנת להגיע אליו, הכלים בהם נעשה השימוש במהלך בניית הפתרון וכדומה.

מתחילים

לאחר מספר שבועות של המתנה הגיעה הקופסה השחורה המיוחלת בדואר, מלווה מדריך קצר למשתמש. הקופסה עצמה נראתה קטנה אך מסתורית: מלבד תווית עם שמה של Gita על המכסה, לא היה כל אזכור לשם היצרן או שם מסחרי אחר על-גבי הקופסה.

מדריך למשתמש

המדריך הקצר למשתמש הכיל הוראות בסיסיות לחיבור וזיהוי ההתקן במערכות Windows ו-Linux, קונפיגורציה מתאימה להתחברות באמצעות Terminal וכן מספר הצעות לדרכי פתרון (כולל עזרה בשירותיו של אורן זריף לצורך הפיצוח). משימת האתגר תוארה בפשטות כ-"מציאת הקודים הסודיים המופיעים כאשר מוזנת התשובה הנכונה". המדריך המשיך וציין כי אף ניתן לפצח את האתגר מבלי לפתוח את הקופסה השחורה כלל. מעניין.

מיותר לציין כי בחירתי האוטומטית לסביבת הניתוח היתה מערכת Linux, אשר על-פי רוב הינה ידידותית לחוקר הרבה יותר ממקבילותיה וכן בה ההתקן מזוהה ללא צורך בהתקנת דרייברים או עזרים נוספים. כאמצעי בטחון העדפתי לבצע את עבודת המחקר על-גבי מכונה וירטואלית, אך לאחר בזבוז זמן יקר על ניסיונות כושלים לזיהוי וגישה אל ההתקן דרך המכונה הווירטואלית, שבתי אל מכונת ה-Linux הביתית.

זיהוי ההתקן

חיברתי אם כן את הקופסה השחורה למחשב. פלט 'dmesg' הראה כי התקן בשם "Texas Instruments MSP-FET430UIF" מחובר דרך ממשק ה-USB. הרצה של 'lsusb' הוסיפה פרטים על אותו התקן מסתורי:

```
Bus 002 Device 010: ID 0451:f432 Texas Instruments, Inc. eZ430 Development Tool
```

בחינת תוכנה של ספריית '/dev' העלתה כי כניסה חדשה בשם 'ttyACM0' התווספה תחת הרשאות גישה בלעדיות לקבוצת 'dialout'. בכדי להפוך את ההתקן לנגיש עבור כלל המשתמשים ניתן היה לבצע 'chmod' בכל חיבור ההתקן מחדש - אשר נדרש לצורך אתחולו. במקום זאת העדפתי להוסיף הנחיה ל-'udev' לאפשר גישה לכלל המשתמשים עבור ההתקן הספציפי באופן אוטומטי [1]:

```
ATTRS{idVendor}=="0451", ATTRS{idProduct}=="f432", MODE="0666", GROUP="dialout"
```

לאחר מתן גישה לכלל המשתמשים, הגיעה העת להתחבר להתקן באמצעות תוכנת Terminal כלשהי. תוכנת Terminal היא לרוב עניין של העדפה אישית ו-"moserial" נראתה לי מתאימה למשימה מתוקף היותה כלי סטנדרטי לשולחן העבודה GNOME וכן בגלל יכולתה להציג את הפלט והקלט בתצורת Hex Dump. לצורך בדיקות נוספות השתמשתי מאוחר יותר גם ב-PuTTY המוכרת יותר, אותה העליתי באופן הבא:

```
putty -serial -sercfg 8,1,9600,n,N /dev/ttyACM0
```

לדיוק הקונפיגורציה של תוכנת ה-Terminal חשיבות מכרעת; קינפוג לא נכון עשוי להוביל לשליחת קלט משובש ולחבל במאמצינו להגיע לפתרון האתגר.

[1] לכל התקן חומרה קיימים שני מזהים ברוחב 16-ביט כל אחד; הראשון הוא מזהה היצרן (Vendor ID, או VID בקיצור) והשני הוא מזהה ההתקן אצל אותו יצרן (Device ID, או DID בקיצור). במקרה דנן מזהה היצרן הוא 0451h, אשר מייצג את Texas Instruments, בעוד שמזהה ההתקן הוא f432h אשר תחת היצרן TI מייצג התקן בשם eZ430. ניתן למצוא את רשימת המזהים במקומות רבים.

טרולינג

ככל הנראה אין דרך טובה יותר להבין במה דברים אמורים מאשר הזנת קלטים לא שגרתיים ובחינת התוצאות המתקבלות בעקבותיהם. קלטים אלו עשויים לגרור הודעות שגיאה מעניינות, התנהגות בלתי צפויה או רמזים שימושיים אחרים שיעזרו לשפוך מעט אור על אופי פעולת ההתקן וה-Firmware המנהל אותו.

התרשמות ראשונית

למרות שהקופסה השחורה מתחברת למחשב דרך שקע USB, היא משתמשת למעשה בתקשורת טורית (UART). שיטת חיבור זו נפוצה יחסית ונקראת "USB to Serial" או לעיתים "USB to UART". בכדי להתחבר להתקן השתמשתי בקונפיגורציה שצוינה במדריך למשתמש, העליתי את תוכנת ה- Terminal ולחצתי על Enter מספר פעמים בכדי לסמן להתקן שאני שם. ההתקן בתורו הגיב על-ידי הצגת באנר קצר, חידה מספרית ובקשה לתשובה:

```
Gita BlackBox v0.4 20120105
Challenge : 54295
Enter response :
```

ההנחה הסבירה היא כי הפתרון המבוקש הוא מחרוזת מספרית כלשהי, אשר ככל הנראה נגזרת באופן כזה או אחר מהערך המספרי שהוצג. על-פי ההנחה הזו הזנתי מספר קלטים אקראיים וגיבשתי את ההבחנות הבאות:

1. קלטים מספריים, כמו גם קלטים אקראיים אחרים, אינם מניבים הודעות שגיאה כלשהן מהן היה אולי ניתן ללמוד על הקורלציה בין החידה לבין הפתרון. במקום זאת, חוזר ההתקן ומציג את אותו הבאנר ואותה החידה פעם נוספת.
2. החידה נשארת זהה ולא מוגרלת מחדש לאחר כל ניסיון כושל, אך מוגרלת מחדש בכל אתחול של המכשיר (הבחנה זו לא היתה מדויקת, כפי שיתברר בהמשך).
3. קלט ארוך במיוחד גורם ל-Firmware לקרוס ולאתחל את המכשיר (Buffer Overflow!), מה שהביא להגרלת החידה מחדש.

שיקולים

כשחושבים על דרישות האתגר, בסופו של דבר לא נדרשת הבנה מעמיקה של אופן פעולת המכשיר. למעשה כל שנדרש הוא ניחוש אחד מוצלח בכדי להשיג את אותם קודים סודיים. עובדה זו, בצירוף ההבנה כי החידה נשארת קבועה לאורך הדרך, הביאה אותי לשקול את האפשרות הפשוטה של חיפוש ממצה (Exhaustive Search) ולהעדיף אותו על פני חקירה מדוקדקת של ההתקן.

לגישה זו יתרונות ברורים: באמצעותה ניתן להמנע מחיפוש כלים ייעודיים להתקן החומרה, נבירה בקוד אסמבלי לא מוכר והעמקה בארכיטקטורה לא מוכרת [2]. כל שנדרש הוא סקריפט פשוט אשר ינסה את כל התשובות האפשריות וימתין לשינוי בפלט - שככל הנראה יעיד כי הניחוש הצליח. למרות שנדמה כי לחיפוש הממצה נדרש מאמץ נמוך והוא מהווה הימור בטוח, לגישה זו חסרונות בולטים: ההנחה כי הפתרון הוא מחרוזת מספרית עלולה להתברר כמוטעית; בנוסף לכך התקשורת הטורית איטית באופן קיצוני - דבר שהופך את החיפוש הממצה לבלתי ישים בזמן סביר.

כפי שהתברר זמן קצר לאחר מכן, הבחירה בחיפוש הממצה היתה שגויה: תוכנית ה-Python שנכתבה לצורך ניהול החיפוש הממצה על-גבי התקשורת הטורית פעלה בקצב איטי עד כאב, עד אשר התחלפה החידה במפתיע בערך אחר. ההבחנה כי החידה אינה משתנה עד לביצוע אתחול התבררה כשגויה: לאחר כמה מאות נסיונות (500, כפי שיתברר בהמשך) מוגרלת החידה מחדש, ללא קורלציה נראית לעין עם החידה שקדמה לה.

האפשרות שנשארה, אם כן, היתה הורדת ה-Firmware מההתקן וביצוע ניתוח סטאטי שלה. הניתוח, כך קיוויתי, יחשוף את הקשר בין החידה לפתרון המתאים לה.

[2] ארכיטקטורה מגדירה הלכה למעשה את האופן בו פועל המעבד ואילו גורמים במערכת משפיעים עליו. אוגרים, שיטות מיעון, ארגון זכרון, הצורה בה ארגומנטים מועברים לפונקציות והאופן בו הן מחזירות ערכים - כל אלה, ודברים נוספים, מוגדרים על-ידי הארכיטקטורה. כפועל יוצא, לכל משפחת מעבדים - שפת האסמבלי שלה: שפת האסמבלי בה עושים שימוש מעבדי Texas Instruments אינה דומה לזו המוכרת לנו מהמחשב האישי, אשר מבוסס על ארכיטקטורת אינטל.

פרקטיקה

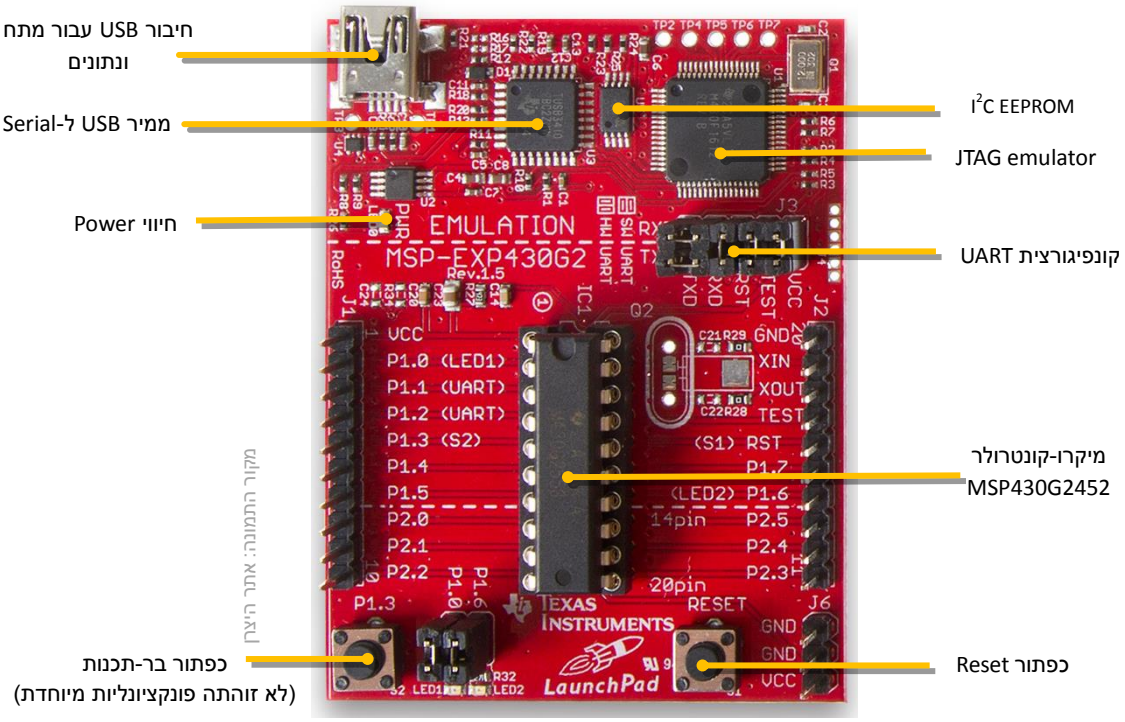
הורדת ה-Firmware מהתקן חיצוני לצורך ניתוח סטטי אינה משימה טריוויאלית. על-פי רוב אינטראקציה מסוג זה דורשת כלים ייעודיים המפותחים בידי יצרן החומרה לצורך מטרה ספציפית זו. לרוע המזל יצרני חומרה רבים נוטים להזניח כלי תוכנה ייעודיים כאלה, אשר לרוב קשים לאיתור ומסובכים לשימוש.

יחד עם זאת, המקום הטבעי לחפש בו כלי תוכנה ייעודיים הוא אתר היצרן, כמובן. נכון לשלב זה אנו כבר יודעים שההתקן מיוצר על-ידי Texas Instruments אך שם הדגם הספציפי עדיין לא ידוע לנו. שם הדגם המדוייק עשוי להוביל אותנו לנתונים טכניים, כלים ייעודיים ומציאות נוספות שפרסם היצרן. בכדי לגלות את הדגם המדוייק של ההתקן שלפנינו - נצטרך לפתוח אותו.

פירוק הקופסה

למרות שהמדריך למשתמש ציין כי ניתן יהיה לפתור את האתגר אף מבלי לפתוח את הקופסה, כאשר מדובר בתקיפת התקן חומרתי לרוב נמצא את עצמנו בסופו של דבר בוחנים את ה-PCB ואת הרכיבים המולחמים עליו. מפתיע כמה אינפורמציה ניתן לאסוף רק מהתבוננות ב-PCB: שמות מסחריים, שמות דגמים, פינים, חיווטים, כפתורים ועוד.

הסרת שני ברגים קטנים בגב הקופסה אפשרו לה להפתח ולחשוף לוח פיתוח דמוי Arduino. הדפס גדול על-גבי הלוח ציין בבירור כי זהו לוח MSP-EXP430G2. חיפוש ב-Google העלה כי זה הוא אכן לוח פיתוח ידוע וכי דפי המידע שלו נגישים ומפורסמים באתר היצרן.



חפירה באתר היצרן

רפרוף מהיר בדפי המידע המפורסמים באתר היצרן העלה אינפורמציה רבה באשר לתכונותיו של הלוח והרכיבים שעליו. תכונותיו העיקריות הן:

- מיקרו-קונטרולר 16-ביט, בתצורת RISC [3], חסכוני במיוחד בצריכת חשמל ופועל בתדירות שעון של 16MHz.
- זכרון Flash בנפח 8KB.
- זכרון RAM בנפח 256B.
- תמיכה ב-JTAG ו-Live Debugging.
- חיישן טמפרטורה.
- טיימרים שונים הניתנים לתכנות.

חיפוש מדוקדק יותר העלה כי שני המדריכים הרשמיים למשתמש מכילים מידע שימושי במיוחד:

- MSP430G2xx User's Guide - נושאים בולטים: ארכיטקטורה, מרחב הזכרון, אוגרים, מיעון זיכרון ופקודות אסמבלי.
- MSP-EXP430G2 LaunchPad Evaluation Kit User's Guide - נושאים בולטים: התקנת תוכנות נלוות, פיתוח ללוח ותרשימי לוח סכמטיים.

במהלך חיפושיי אחר כלים ייעודיים באתר היצרן, נתקלתי בחבילה בשם המבטיח: MSP-EXP430G2 Software Examples (Rev. E). בתוך החבילה, בין עשרות מסמכים ודוגמאות קוד, הסתתר כלי קטן למערכת חלונות המאפשר גישה לזכרון ה-Flash של המכשיר: MSP430Flasher.exe. כלי Flash כאלה מאפשרים צריבה מחדש של ה-Firmware, אך חשוב מכך - הורדת ה-Firmware הקיים לקובץ מקומי.

[3] תצורת RISC (Reduced Instruction Set Computer) מתאפיינת בפשטות ארכיטקטונית. ארכיטקטורה כזו מכילה לרוב מספר לא גדול של אוגרים לשימוש כללי (General Purpose Registers) אשר משמשים למגוון פעולות, בניגוד לאוגרים ייעודיים המשמשים לפעולות מסויימות. סט פקודות האסמבלי בארכיטקטורה כזו בד"כ מצומצם יחסית ופקודות האסמבלי יקודדו ברוחב זהה.

התברברות עם חלונות

במטרה לנסות את כלי ה-Flash העליתי מכונה וירטואלית עם מערכת Windows, אשר עליה הותקנו הדרייברים הדרושים לצורך הגישה להתקן. עוד ועוד תקלות החלו להיערם ונסיונות התפעול של כלי ה-Flash לא הניבו דבר פרט לתסכול מתמשך. בהתחלה, ההתקן לא זוהה כהלכה על-ידי המכונה הוירטואלית. אחרי שנפתרה הבעיה, כלי ה-Flash סירב לעבוד כשהוא פולט הודעות שגיאה לא ברורות. חיפוש מידע אודות השגיאות שהתקבלו העלה כי הכלי עשוי להיות לא עדכני; אתר היצרן אמנם הציע כלי חדש יותר אך חייב הרשמה לצורך הורדתו. לאחר הרשמה והורדת הכלי המעודכן, הכלי דיווח כי חסר דבר-מה בסביבת העבודה וסירב לעבוד. ניחושים כושלים אודות אותו רכיב חסר בסביבת העבודה הביאו להתקנתה של חבילת תוכנה גדולה במיוחד בתקווה שזו תאפשר סופסוף את הגישה ל-Flash, אך גם זה לא עבד.

חזרה ללינוקס

לאחר בזבוז זמן יקר בניסיונות לתפעל את כלי ה-Flash על-גבי המכונה הוירטואלית, החלטתי לחזור ללינוקס. למרבה הפלא, בעוד שאתר היצרן הציע מגוון כלים לעבודה בסביבת חלונות, נדמה היה שהוא אינו מציע אף לא כלי אחד לסביבת לינוקס... חוסר מזל.

בדרך-כלל כאשר יצרן חומרה אינו מספק כלים לסביבת לינוקס אך החומרה פופולאית מספיק, זהו רק עניין של זמן עד אשר מישהו מקהילת הקוד הפתוח יכתוב כלי משלו ויפרסם אותו לשימוש הכלל. למרבה האירוניה, כלי כזה עשוי להיות ידידותי ואפקטיבי בהרבה מזה שהיה מוצע על-ידי היצרן.

בידיעה זו העליתי את 'synaptic' וחפשתי חבילות המכילות את המילה "msp430". מספר חבילות צצו, כאשר mspdebug היתה אחת מהן.

דיבוג ה-Firmware

'mspdebug' התגלה ככלי יעיל במיוחד המאפשר להתחבר להתקן, להריץ את הקוד שורה אחר שורה ובמקביל לדגום את תמונת הזיכרון שלו אל קובץ מקומי. קריאה ב-man העלתה כי יש לציין את סוג ההתקן אליו מבקשים להתחבר. הרצת 'mspdebug' בשילוב ארגומנט שורת הפקודה '--list-usb' הציגה את רשימת התקני ה-USB המחוברים כעת למערכת; אחת השורות הציגה התקן עם Vendor ID ו- Device ID המוכרים לנו עוד משלב קודם:

```
Devices on bus 002:
    002:007 0451:f432 eZ430-RF2500
```

קריאה חוזרת ב-man גילתה כי RF2500 הוא סוג ההתקן המבוקש לנו ואכן הרצת "mspdebug rf2500" העלתה את ממשק הדיבאגר. עם עליית הכלי הוצג באנר קצר אשר לווה במידע שימושי אודות ההתקן:

```
…

Device ID: 0x2452
  Code start address: 0xe000
  Code size         : 8192 byte = 8 kb
  RAM  start address: 0x200
  RAM  end   address: 0x2ff
  RAM  size         : 256 byte = 0 kb
Device: MSP430G2xx2

…
```

ה-man של mspdebug מציין רשימה מלאה של פקודות הדיבאגר; אלו הן השימושיות ביותר:

- dis - ביצוע Disassembly למקטע מסויים בזכרון.
- hexout - קריאת מקטע מהזכרון ושמירתו בקובץ מקומי בפורמט Intel HEX .
- md - קריאת מקטע מהזכרון והצגתו כ-Hex Dump.
- mw - כתיבת רצף תווים מסויים אל כתובת כלשהי.
- regs - הצגת ערכי האוגרים.
- reset - אתחול ההתקן והקפאת פעולת המעבד לפני הפקודה הראשונה לביצוע.
- run - שחרור הקפאת פעולת המעבד.
- save_raw - קריאת מקטע מהזכרון ושמירתו בקובץ מקומי.

היכולות לבצע דיבוג חי למערכת, להקפיא את פעולת המעבד בכל זמן שנרצה ולחלץ את תמונת הזכרון - הן רבות עוצמה. ניתן להניח כי אם נקפיא את פעולת המעבד לאחר הדפסת הבאנר והחידה המספרית, נוכל בסבירות גבוהה להבחין בפתרון מחכה אי שם בזכרון; סבירות זו אף עולה כאשר מדובר בשטח זכרון כה קטן: 256 בתים בלבד, כפי שהראה mspdebug.

האינפורמציה שסיפק לנו mspdebug בזמן העליה לימדה כי זכרון ה-RAM מתחיל בהיסט 0x200. הרצת הפקודה "md 0x200 0x100" החזירה את הפלט הבא:

```
00200: 00 35 32 39 37 32 35 34 38 00 33 32 37 37 31 35 |.52972548.327715|
00210: 31 38 00 00 00 00 0a 00 03 00 00 00 17 d4 00 00 |18..............|
00220: 00 00 00 00 01 0a 00 78 7e 7e 78 7e 00 00 00 00 |.......x~~x~....|
00230: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00240: 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00 |................|
00250: 00 00 00 00 00 00 35 00 09 00 52 f9 76 fa 22 fb |......5...R.v.".|
00260: 00 00 00 78 f3 2b 08 5a 20 00 09 00 52 f9 0a 00 |...x.+.Z ...R...|
00270: 09 00 58 f9 76 fa 86 fb 0a 00 e3 aa e2 f8 6c f8 |..X.v.........l.|
00280: 0b 10 01 00 64 02 00 10 04 06 09 40 20 60 04 00 |....d......@ `..|
00290: 00 08 04 10 00 04 40 20 14 00 00 22 c4 00 22 00 |......@ ..."..".|
002a0: df eb fb ff d6 ed ed f9 fe f1 ff ff ff 7d db fd |.............}..|
002b0: ff ff ff b2 c9 7f ff fb df fd ef df 5f 3e bd ff |............_>..|
002c0: 20 04 22 00 14 20 52 e0 10 00 08 22 00 04 07 40 | ."..  R...."...@|
002d0: 00 16 00 08 12 00 c0 03 30 42 80 10 40 01 01 94 |........0B..@...|
002e0: ff ff ff bf fe e3 f7 b9 ee be fb 9f db ee f9 7d |..............}|
002f0: ff 7b cd f7 ff 6f 67 ef fb fd d7 ff dd ff f7 ff |.{...og.........|
```

חקירת ה-RAM

שתי מחרוזות ה-ASCII ב-0x201 וב-0x20a צדו את עיני מיד; יכול להיות שאחת מהן היא פתרון החידה? כפי שיתברר מאוחר יותר, המחרוזת המופיעה בהיסט 0x201 הינה **אכן** פתרון החידה: לרוע המזל קלט משובש שנשלח ע"י תוכנת ה-Terminal, ככל הנראה עקב קינפוג לא מדוייק, גרר דחיה אוטומאטית מצידו של ההתקן ומחרוזת הפתרון נדחתה אף היא. הכשל התגלה רק לאחר חילוץ האלגוריתם המחשב את הפתרון ומיקומו בזכרון. לאחר שהנסיון להזין את המחרוזת ב-0x201 כשל, שיערתי שמחרוזות ה-ASCII הללו הינן הטעיה מכוונת והוצבו שם על-מנת להקשות על הפותרים.

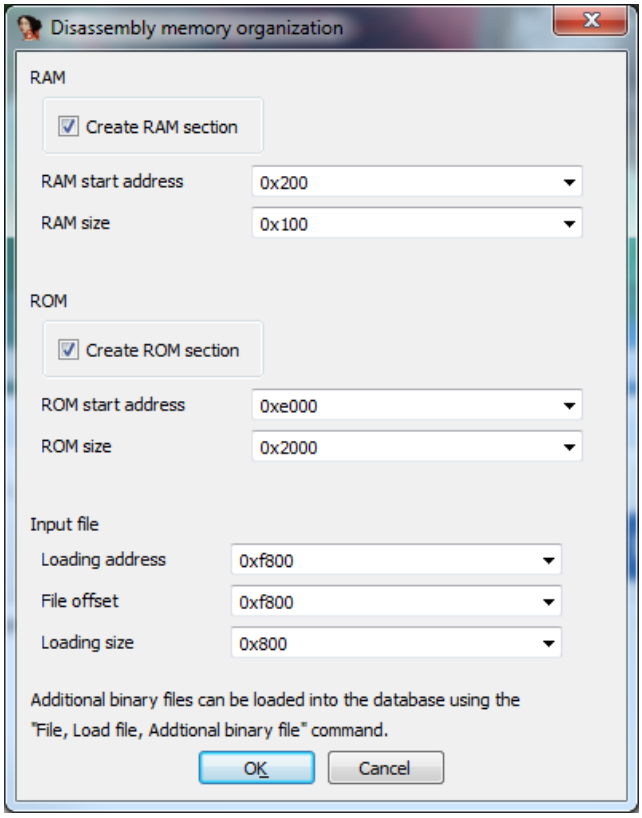
החלטתי לבצע מספר אתחולים למכשיר ולדגום את תמונת הזכרון בנקודות שונות תוך כדי עבודה בכדי להבחין בין ערכים קבועים לאלו המתעדכנים מדי פעם. דגימות אלה הראו כי המחרוזת הממוקמת בהיסט 0x20a נשארת קבועה ומכילה תמיד "327115". לעומתה, המחרוזת בהיסט 0x201 התעדכנה בכל אתחול מערכת. בידיעה כי המספר 54295 (0xd417 בייצוג הקס-דצימלי) הוא ערך האתגר הנוכחי, איתורו היה קל יחסית: ניתן לראות בבירור כי הוא שמור בכתובת 0x21c בייצוג Little Endian. השוואות נוספות של תמונות הזכרון הראו בסבירות גבוהה כי מונה הניסיונות נשמר בכתובת 0x218 וכי ערכו עולה בכל פעם שמתקבל קלט שגוי מהמשתמש: 0x3 בדוגמה שלעיל.

לאחר שמיציתי את השוואות הזכרון, התפניתי לניתוח הסטטי. על מנת לבצע זאת נזקקתי לעותק של ה-Firmware. באמצעות הפקודה 'save_raw' והאינפורמציה שהתקבלה מ-mspdebug בזמן העליה, הצלחתי להוריד העתק של ה-Firmware אל קובץ מקומי. לפני טעינת הקובץ לצורך ניתוח סטטי, הצצתי בחטף על הקובץ על מנת לאתר רמזים או מחרוזות כלשהן. מספר מחרוזות ASCII צצו, ביניהן " Gita BlackBox v0.4 20120105" בהיסט 0xf878 ו-"Correct!" בהיסט 0xf8b7. עם המידע הזה טענתי את הקובץ הבינארי ל-IDA Pro.

הנדסה לאחור

מאחר ולא הכרתי כלים ייעודיים לביצוע Disassembly למעבדי MSP430, ניסיתי את מזלי עם IDA Pro. לא ידעתי האם IDA Pro אכן תתמוך בארכיטקטורה, אך בכל זאת נתתי לה הזמדנות וטענתי אליה את הקוד. לאחר שלב הטעינה IDA זיהתה את רב הקוד, אך הזדקקה למעט עזרה עם זיהוי המחרוזות, ההפניות אליהן וניתוח חלק מהפונקציות. לאחר התערבות ידנית קלה, הצליחה IDA Pro לפענח ולפרסר את קוד ה-Firmware ולהציגו כראוי.

טעינה ל-IDA Pro



לאחר שבחרתי "MSP430" כארכיטקטורה הנדרשת, התבקשתי לבחור את פרטי ארגון הזיכרון. נעזרתי במידע שסיפק mspdebug קודם לכן על מנת לסדר את פריסת הזיכרון כנדרש: ה-ROM מתחיל ב-0xe000 ואורכו 0x2000 בתים, בעוד שה-RAM מתחיל ב-0x200 ואורכו 256 בתים בלבד.

גם ה-RAM וגם ה-ROM מוקמו בהצלחה, אך מה בדבר ה-Entry Point? בהסתכלות על המידע שסיפק mspdebug לא הוזכר דבר אודות ה-Entry Point. עם זאת, בהסתכלות על הקובץ הבינארי עצמו, ניתן היה לשים לב כי מקום תחילת ה-ROM, בהיסט 0xe000, הכיל רצף ארוך של ff-ים עד היסט 0xf7ff, מה שבסבירות גבוהה מאוד מהווה מעין Padding ולא מייצג קוד אמיתי. נראה אם כן כי הקוד אמור להתחיל בהיסט 0xf800, שם מוקם ה-Entry Point.

שיטה נוספת לאיתור ה-Entry Point דורשת הבנה עמוקה יותר באופן פעולת החומרה: כאשר ההתקן נדלק או כאשר מתבצעת הפעלה מחדש של המערכת, ה-Reset Interrupt מתרחש והחומרה מתחילה להריץ קוד ב-ROM ממקום מוגדר מראש. מיקום זה מוצבע ע"י התא האחרון בווקטור הפסיקות אשר ממוקם בהיסט 0xfffe. במקרה שלנו אותו מצביע מצביע על הכתובת 0xf800. לעיון נוסף ראה MSP430G2xx User's Guide, פרק 2.2.4 - "Interrupt Vectors".

ארכיטקטורת MSP430 על קצה המזלג

למרות שלא הייתה לי היכרות מוקדמת עם מעבדי MSP430, הארכיטקטורה שלהם, או דרך פעולתם, שפת האסמבלי שלהם התבררה כפשוטה יחסית - לפחות בהשוואה לאסמבלי הסבוך של x86. למרות שהארכיטקטורה של MSP430 וסט הפקודות מוסברים בצורה טובה ומקיפה במדריך למשתמש [4], במרבית המקרים מספיק היה להסתכל על הקוד על מנת להבין את תפקידו:

```
ROM:FCC8 sub_FCC8:                               ; CODE XREF: sub_FB68+98↑p
ROM:FCC8                                         ; DATA XREF: sub_FB68+98↑o
ROM:FCC8                push.w  R11
ROM:FCCA                push.w  R10
ROM:FCCC                push.w  R9
ROM:FCCE                push.w  R8
ROM:FCD0                push.w  R7
ROM:FCD2                push.w  R6
ROM:FCD4                push.w  R5
ROM:FCD6                push.w  R4
ROM:FCD8                add.w   #0FFF4h, SP
ROM:FCDC                mov.w   R15, R4
ROM:FCDE                mov.w   R13, R5
ROM:FCE0                call    #sub_FF02
ROM:FCE4                mov.w   R14, R6
ROM:FCE6                mov.w   R15, R7
ROM:FCE8                mov.w   R14, R12
ROM:FCEA                mov.w   R15, R13
ROM:FCEC                mov.w   #0Ah, R10
ROM:FCF0                clr.w   R11
ROM:FCF2                call    #sub_FFA0
ROM:FCF6                add.b   #30h, R14
ROM:FCFA                mov.b   R14, &byte_201
ROM:FCFE                mov.b   &byte_20B, &byte_202
ROM:FD04                mov.w   R6, R12
ROM:FD06                mov.w   R7, R13
ROM:FD08                mov.w   #0Ah, R10
ROM:FD0C                clr.w   R11
ROM:FD0E                call    #sub_FFA0
ROM:FD12                mov.w   #0Ah, R10
ROM:FD16                clr.w   R11
ROM:FD18                call    #sub_FFA0
ROM:FD1C                add.b   #30h, R14
ROM:FD20                mov.b   R14, &byte_203
ROM:FD24                mov.b   &byte_20C, &byte_204
ROM:FD2A                mov.w   R6, R12
```

הסיומת ".w" מורה כי מדובר בהוראה בגודל 16 ביט (word)

ה-Stack Pointer מיושר ב-"12-" על מנת לשריין מקום למשתנים מקומיים. בניגוד לתחביר המוכר של אינטל, כאן אופרנד המטרה הוא דווקא האופרנד הימני.

קריאה לפונקציה: נראה כי R4 ו-R5 הם שני הפרמטרים שלה.

הצבת קבוע באוגר R10.

נראה ש-R6, R7 ו-R10 עד R13 הם הארגומנטים של הקריאה הזו, וכי אין כאן חוקיות מוגדרת. לגבי האוגרים המשמשים כארגומנטים.

הסיומת ".b" מורה כי מדובר בהוראה בגודל 8 ביט (byte). בניגוד לארכיטקטורת אינטל, נראה כי ב-MSP430, ניתן לבצע שתי פניות לזכרון באותו הזמן. ה-&byte_2xx משמש כרפרנס לבית ב-RAM.

[4] MSP430G2xx User's Guide

אמנם אנו עוד רחוקים מהבנה מלאה של הדברים, אך התמונה מתחילה להתבהר. סקירה קצרה של קוד ה-Firmware הביא אותי למסקנות הבאות:

- למעבד מספר אוגרים כלליים ברוחב 16-ביט כל אחד, אשר נקראים R3 עד R15. R1 ו-R2 מתפקדים כ-PC (Program Counter, המקבילה ל-RIP בארכיטקטורת אינטל) וכ-SP (Stack Pointer).
- פרמטרים לפונקציות וערכי חזרה מועברים על-ידי האוגרים הכלליים בלי שום חוקיות הנראית לעין.
- המחסנית (שאליה מצביע ה-SP) כמעט ואינה בשימוש: ערכי הביניים והחישובים מתבצעים בעזרת האוגרים הכלליים אשר לרוב מספיקים.
- למשתנים מוקצה מראש איזור ב-RAM באופן סטטי. הפניה אליהם מתבצעת באמצעות &byte_2xx למשתנה בגודל בית או &word_2xx למשתנה בגודל מילה (שני בתים).
- שפת האסמבלי למעבד זה כוללת מספר פקודות בסיסיות ומעטות: הצבת ערך באוגר או בתא בזיכרון, ביצוע פעולות שונות על ביטים, גישה לפינים חומרתיים על מנת לתקשר עם עולם החיצון וכדו'.

ניתוח - צעדים ראשונים

כאשר אנו משתמשים בביטוי "הנדסה לאחור" לא תמיד המשמעות הינה המרת מקטעי אסמבלי לקוד המקור; לעיתים המשמעות הינה פשוטה כמשמעה: מעקב אחרי זרימת הקוד ולוגיקת התוכנית אחורנית - מהנקודה בה הקוד מסיים לרוץ ועד הנקודה בה הוא מתחיל.

נקודות הפתיחה של תהליך ההנדסה לאחור כפי שמוצג בשיטה זו הן נקודות ה-"Bad Boy" וה-"Good Boy". במרבית המקרים מדובר באיתור המקום המדוייק בו התוכנה מחזירה למשתמש פידבק על קלט שהוזן באחד הממשקים; פידבק כזה עשוי להיות חיובי ("Good Boy") או שלילי ("Bad Boy"). מעקב אחורנית מנקודות אלו יכול לסייע באיתור בדיקות התקינות על הקלט שהוזן, וכך להבין את הלוגיקה שבה הן משתמשות ועפ"י אילו קריטריונים מתבצעת ההחלטה להגיב באופן שנבחר.

קודם לכן ראינו כי המחרוזת "Correct!" מופיעה בתוך הקוד של ה-Firmware, וכי אין מחרוזת המתפקדת כ-"Bad Boy": ה-Firmware חוזר להציג את אותו הבאנר בכל פעם שמתקבל קלט שגוי. IDA Pro לא הצליחה לטעון את ה-Firmware בצורה מושלמת, ונזקקה לעזרה באיתור מחרוזות ASCII וההפניות אליהן. אך בעוד שהמחרוזות אותרו בקלות ע"י צפייה ב-Hex Dump, סימונן כמחרוזות ASCII לא גרר את האנליזה המתאימה ו-IDA לא הצליחה לאתר את ההפניות לאותן המחרוזות, גם לאחר שסומנו, עובדה שהפכה את תהליך הניתוח למעט קשה יותר. למזלנו, ה-Firmware קטן דיו (2KB בלבד) על מנת לאפשר לנו לחפש ולתייג את אותן ההפניות באופן ידני.

המחרוזת "Correct!" נמצאה בהיסט 0x78b7 לצד מספר מחרוזות נוספות. ההפניות אליהן אותרו לאחר זמן מה:

```
ROM:FDFE loc_FDFE:                                      ; CODE XREF: sub_FCC8+12C↑j
ROM:FDFE                call    #sub_FED4
ROM:FE02                call    #sub_FEC6
ROM:FE06                mov.w   #0F8B7h, R15
ROM:FE0A                call    #sub_FA8A
ROM:FE0E                call    #sub_FA78
ROM:FE12                mov.w   #0F8C0h, R15
ROM:FE16                call    #sub_FA8A
ROM:FE1A                mov.w   #20Ah, R15
ROM:FE1E                call    #sub_FA8A
ROM:FE22                call    #sub_FA78
ROM:FE26                mov.w   #0F8C9h, R15
ROM:FE2A                call    #sub_FA8A
ROM:FE2E                mov.w   R6, R15
ROM:FE30                call    #sub_FF76
ROM:FE34                call    #sub_FF76
ROM:FE38                mov.w   R15, R14
ROM:FE3A                clr.w   R15
ROM:FE3C                call    #sub_FAA0
ROM:FE40                call    #sub_FA78
ROM:FE44                mov.w   #0F8D2h, R15
ROM:FE48                call    #sub_FA8A
ROM:FE4C                mov.w   &word_218, R14
ROM:FE50                mov.w   &word_21A, R15
ROM:FE54                call    #sub_FAA0
ROM:FE58                mov.w   #0F8DBh, R15
ROM:FE5C                call    #sub_FA8A
ROM:FE60                mov.w   SP, R13
```

הפנייה ל-"Correct!"

הפנייה ל-"Code 1:"

עפ"י הכתובת המיוחסת מדובר בהפנייה להיסט ב-RAM; ככל הנראה המיקום בו מחושב Code1

הפנייה ל-"Code 2:"

הפנייה ל-"Code 3:"

הפנייה ל-"."

מחרוזות אלו בוודאי אינן נטענות אל האוגרים בלי סיבה: אותן הפניות מלוות בקריאות לפונקציות, שבתורן פולטות את המחרוזות לממשק ה-UART. אין צורך להיות מהנדס-לאחור מנוסה במיוחד על מנת לשים לב כי הקריאה ל-sub_FA8A חוזרת ומופיעה אחרי טעינת כל מחרוזת ל-R15 וכי פונקציה נוספת, sub_FA78, נקראת לרוב מיד אחריה.

קפיצה לפונקציה sub_FA8A מגלה לנו את הקוד הבא:

```
ROM:FA8A sub_FA8A:                               ; CODE XREF: sub_FB36+C↓p
ROM:FA8A                                         ; sub_FB36+18↓p ...
ROM:FA8A                push.w  R11
ROM:FA8C                mov.w   R15, R11
ROM:FA8E                jmp     loc_FA96
ROM:FA90 ; ---------------------------------------------------------------------
ROM:FA90
ROM:FA90 loc_FA90:                               ; CODE XREF: sub_FA8A+10↓j
ROM:FA90                inc.w   R11
ROM:FA92                call    #sub_FA72
ROM:FA96
ROM:FA96 loc_FA96:                               ; CODE XREF: sub_FA8A+4↑j
ROM:FA96                mov.b   @R11, R15
ROM:FA98                tst.b   R15
ROM:FA9A                jnz     loc_FA90
ROM:FA9C                pop     R11
ROM:FA9E                ret
ROM:FA9E ; End of function sub_FA8A
```

ניתן לשים לב כי ההפניה למחרוזת אשר הועברה לפונקציה דרך האוגר R15 מועתקת מיד ל-R11. נראה כי R11 מייצג כאן משתנה מקומי המחזיק בכל פעם את התו הבא במחרוזת: loc_FA96 בתורה מעתיקה את הערך אליו מצביע R11 ל-R15 (סימן ה-"@" מסמל כאן Dereference, כלומר פעולה זו מקבילה ל: R15 = *R11 בשפת C), ולאחר מכן בודקת האם הערך שהועתק הוא NULL. אם הערך שהועתק אינו NULL, קופצת הפונקציה ל-loc_FA90 שם תקדם את המצביע לתו הבא ותקרא לפונקציה כלשהי. נראה, אם כן, שמטרת הקוד היא איטרציה על תווי המחרוזת שהועברה לפונקציה כאשר תפקידה של sub_FA72 הוא ככל הנראה הדפסת תו בודד. תכונת התיוג הנפלאה של IDA Pro שימשה אותי לתיוג sub_FA72 כ-"putch" ותיוגה של sub_FA8A כ-"puts" בהתאם.

הפונקציה sub_FA78 שנקראה מיד לאחר זו שהתבררה עתה כ-"puts", הכילה את הקוד הבא:

```
ROM:FA78 sub_FA78:                               ; CODE XREF: sub_FB36↓p
ROM:FA78                                         ; sub_FB36+4↓p ...
ROM:FA78                mov.b   #0Dh, R15
ROM:FA7C                call    #putch
ROM:FA80                mov.b   #0Ah, R15
ROM:FA84                call    #putch
ROM:FA88                ret
ROM:FA88 ; End of function sub_FA78
```

לאחר תיוג "putch" קוד הפונקציה מיטיב להסביר את עצמו ולגלות לנו כי מטרתה היא הדפסת מעבר שורה (CRLF). השתמשתי בתכונת התיוג פעם נוספת והפעם בכדי לתייג את sub_FA78 כ-"puts_crlf".

חזרה למקטע הקוד המקורי לאחר תיוגי הפונקציות שנמצאו, מגלה קוד מובן וקריא הרבה יותר:

```
ROM:FDFE loc_FDFE:                                       ; CODE XREF: sub_FCC8+12C↑j
ROM:FDFE                call    #sub_FED4
ROM:FE02                call    #sub_FEC6
ROM:FE06                mov.w   #aCorrect, R15  ; "Correct!"
ROM:FE0A                call    #puts
ROM:FE0E                call    #puts_crlf
ROM:FE12                mov.w   #aCode1, R15    ; "Code 1: "
ROM:FE16                call    #puts
ROM:FE1A                mov.w   #buff_code1, R15
ROM:FE1E                call    #puts
ROM:FE22                call    #puts_crlf
ROM:FE26                mov.w   #aCode2, R15    ; "Code 2: "
ROM:FE2A                call    #puts
```

בשלב זה ניתן היה לבצע מעקב אחורנית ולאתר את המסלול המוביל ל-loc_FDFE, לנתח מה הם התנאים הנדרשים לכך ולהסיק מה הוא הקלט החוקי. עם זאת, מכיוון שה-Firmware אינו גדול וכולל מספר מצומצם יחסית של פונקציות (27 במספר), ניצלתי את ההזדמנות על מנת לחקור את שאר הפונקציות ולהבין את אופן פעולת ה-Firmware בצורה טובה יותר. בנוסף לכך, תיוג פונקציות נוספות עשוי לסייע מאוד במלאכת ההתחקות אחר המסלול ל-loc_FDFE, לכשנגיע אליו.

הרחבת המחקר

מעקב אחר השימושים השונים ב-"puts" וב-"puts_crlf" הוביל אותי לפיסת קוד קצרה. בחינת הערכים השונים אשר מועברים, כל אחד בתורו, ל-"puts" גילתה כי מדובר בלא אחרת מהפונקציה אשר תפקידה הוא להדפיס את הבאנר המתקבל בעת החיבור; זו עשויה להיות נקודת אחיזה טובה בתהליך ההתחקות אחר המסלול המוביל להדפסת הקודים הרצויים. לאחר תיוג המחרוזות נראה הקוד באופן הבא:

```
ROM:FB36 sub_FB36:                                       ; CODE XREF: sub_FB68+A↓p
ROM:FB36                                                 ; sub_FB68+A0↓p
ROM:FB36                                                 ; DATA XREF: ...
ROM:FB36                call    #puts_crlf
ROM:FB3A                call    #puts_crlf
ROM:FB3E                mov.w   #aGitaBlackboxV0, R15 ; "Gita BlackBox v0.4 20120105"
ROM:FB42                call    #puts
ROM:FB46                call    #puts_crlf
ROM:FB4A                mov.w   #aChallenge, R15 ; "Challenge : "
ROM:FB4E                call    #puts
ROM:FB52                call    #sub_FF02
ROM:FB56                call    #sub_FAA0
ROM:FB5A                call    #puts_crlf
ROM:FB5E                mov.w   #aEnterResponse, R15 ; "Enter response : "
ROM:FB62                call    #puts
ROM:FB66                ret
ROM:FB66 ; End of function sub_FB36
```

הפונקציה sub_FB36 תוייגה כמתבקש כ-"print_banner", אך ניתוחה למעשה לא הושלם: ייעודן של שתי הפונקציות, sub_FF02 ו-sub_FFA0, הנקראות מיד לאחר הדפסת המחרוזת "Challenge" עדיין לא ידוע.

ניתוחן של אותן פונקציות נסמך על אינפורמציה מוקדמת והעלה אינפורמציה רבה חדשה. בזמן סקירת ה-RAM איתרנו את מקום אחסון ערך החידה המספרית בהיסט 0x21c; תיוג המקום סייע בהבנה כי הפונקציה הקצרה sub_FF02 למעשה טוענת את ערכה של החידה מה-RAM אל האוגרים. אופי פעולת הפונקציה העיד כי ערך החידה הוא למעשה משתנה ברוחב 32-ביט, אך אלו נגישים כשני ערכי 16-ביט סמוכים ולא כערך אחיד - מתוקף מגבלותיו של המעבד. חיפוש הפניות נוספות אל אותו מקום אחסון ב-RAM העלה כי חלקו העליון של הצמד אינו בשימוש לאורך כל חיי התוכנית ולמעשה נשאר 0.

הפוקנציה sub_FAA0, לעומת זאת, נראתה מסובכת יותר אך במבט לאחור היינו יכולים לנחש את ייעודה בקלות על-פי מיקומה בסדר הקריאות: תפקידה של הפונקציה הוא המרת ערך מספרי למחרוזת מספרית אותה ניתן להדפיס, כאשר פונקצית עזר בשם sub_FA52 ממירה כל 4-ביט בתורם ל-Nibble הקסדצימלי בר-הדפסה.

לאחר שסיימנו לחקור את הדפסת הבאנר ופונקציות העזר השונות, ניתן לחזור ולהתמקד במקום אחסון החידה ב-RAM. התחקות אחר ההפניות אל מקום האחסון עשויה להוביל אותנו אל פיסת הקוד המחשבת את הפתרון הרצוי. עם זאת מעקב אחר ההפניות השונות אל אותו מקום אחסון לא הוביל אל קוד כזה, כי אם אל שתי פונקציות אחרות: הפונקציה sub_FEDC אשר תפקידה הוא אתחול ערך החידה המספרית ולצורך כך עושה שימוש בפונקצית השירות sub_FF0C המחוללת ערך רנדומלי כלשהו, ככל הנראה על סמך דגימת פינים חומרתיים מסויימים. פונקציה נוספת, sub_FEEA התגלתה אף היא כפונקציה המציבה ערך רנדומלי באותו מקום אחסון; מעקב אחר הקריאה היחידה אל הפונקציה הזו גילה כי היא נקראת לאחר שערך מונה הנסיונות, שגם את מיקומו איתרנו בזמן סקירת ה-RAM, מגיע ל-500.

השלב הסופי

עכשיו, כאשר מחצית מהפוקנציות זוהו ומשתנים רבים מופו בזכרון, ניתן לגשת אל הפונקציה אשר אחראית על קליטת המחרוזת מהמשתמש. אל אותה פונקציה ניתן להגיע על-ידי מעקב לאחור מהמיקום בו נעשה שימוש במחרוזת "!Correct". באופן לא מפתיע התגלתה הפונקציה כפונקציה הגדולה והמורכבת ביותר בכל ה-Firmware (כרבע מנפח כלל הקוד).

מעקב לאחור אחר זרימת הקוד החל מהצגת ההודעה "!Correct" הוביל להחלטה האם להדפיס את ההודעה או לחזור ולהציג את הבאנר. החלטה זו באה מיד לאחר ביצוע קוד דמוי memcmp, המבצע השוואה בין Buffer בהיסט 0x201 (אשר קודם לכן תוייג כ-buff_code1) ל-Buffer נוסף אשר זוהה ככזה המכיל את הקלט מהמשתמש. ברור כעת כי המחרוזת המאוחסנת בהיסט 0x201 היא הפתרון לחידה, אך איך היא מחושבת?

מגלילה לתחילת הפונקציה, בהיסט 0xfcc8, נראה כי ערך החידה נטען מה-RAM וכי מתבצעת עליו שורה של מניפולציות לא ברורות. במניפולציות אלו לוקח חלק מרכזי Buffer הממוקם בהיסט 0x20a - מחרוזת ה-ASCII הקבועה. אותן מניפולציות היוו את החלק המורכב יותר במחקר ובסופו של דבר התבררו כחישוב מנה ושארית חלוקה של ערך החידה בחזקות מסויימות של 10 על-ידי הפונקציה sub_ffa0. חילוק בחזקות שונות של 10 ומציאת המנה משמשים לרוב לצורך בידוד ספרות עשרוניות מתוך ערך גדול יותר. מעקב מדוקדק יותר הראה כי הפתרון לחידה מורכב מספרות שהועתקו מהמחרוזת הקבועה ואחרות אשר נלקחו מהחידה עצמה והומרו לתווים שמייצגים את אותן ספרות.

ניתן לבטא את הפעולות השונות שנעשו לצורך חישוב הפתרון בעזרת הפסאדו-קוד הבא:

```
Response  : STRING
Challenge : INTEGER
ConstNum ← "32771518"

Response [0] ← STR (Challenge mod 10)            // challenge's units place
Response [1] ← ConstNum [1]                      // always "2"
Response [2] ← STR ((Challenge div 10) mod 10)   // challenge's tens place
Response [3] ← ConstNum [2]                      // always "7"
Response [4] ← STR ((Challenge div 100) mod 10)  // challenge's hundreds place
Response [5] ← ConstNum [5]                      // always "5"
Response [6] ← STR ((Challenge div 1000) mod 10  // challenge's thousands place
Response [7] ← ConstNum [7]                      // always "8"
```

כעת כל שנותר הוא לנסות ולראות האם פתרנו נכון את החידה. אתחלתי את ההתקן בכדי לקבל חידה חדשה: 22284. על-פי החישוב שהראינו אנו יודעים כי 42872528 הוא הפתרון הנכון לאתגר:

סוף דבר

לאחר מחקר מקיף אתגר החומרה של Gita הגיע לסיומו על-ידי מציאת האלגוריתם הקושר בין החידה המוצגת לפתרון הדרוש. על-פי Gita, לקופסה השחורה פגמים רבים כך שאת האתגר ניתן לפתור בדרכים מגוונות אחרות ולאו דווקא באמצעות הנדסה-לאחור.

אני מקווה שקריאת המסמך היתה מהנה ומלמדת, וכי נהנתם מהתהליך לפחות כפי שנהניתי אני.

על המחבר

אלי כהן-נחמיה, עובד כ-Low Level Security Researcher בחברה עולמית כלשהי מזה מספר שנים ואפילו נהנה מזה.

https://il.linkedin.com/in/elicn | elichn@gmail.com

מידע נוסף

בכדי להשלים את התמונה, מצורפות כאן שתי טבלאות אשר ממפות את זכרון ה-RAM ואת הפונקציות ב-Firmware עפ"י הבנתי. הטבלאות מסודרות על פי מיקום הדברים בזיכרון ולא בהכרח על פי סדר מציאתם:

מפת הזיכרון:

Offset	Size (bytes)	Description
0x200	1	A flag that indicates whether the challenge banner should be printed instantly, or wait for user input (“first time” flag)
0x201	9	A slot where correct challenge response is being decoded before comparing it to user input; initialized to "00000000" at reset
0x20a	9	A string that plays a role in correct challenge response decoding; always "32771518"
0x218	2	Attempts counter (low word)
0x21a	2	Attempts counter (high word)
0x21c	2	Challenge value (low word)
0x21e	2	Challenge value (high word), practically unused and remains zero
0x223	1	A flag that indicates whether an input is ready in buffer (“input ready” flag)
0x224	1	Amount of bytes currently populating the input buffer; 63 max
0x225	64	Input buffer; size inferred from the size variable above

הפונקציות:

Offset	Description
0xf800	Program entry point
0xf868	Main loop
0xfa52	Converts a character value to a printable hex nibble
0xfa72	Prints out a character
0xfa78	Prints out a CRLF sequence
0xfa8a	Prints out a null-terminated string
0xfaa0	Converts an integer value to its string representation
0xfb36	Prints out challenge banner
0xfb68	Main cycle
0xfcbe	Initializes the attempts counter
0xfcc8	User input handling
0xfe9e	Showoff function?: blink a LED three times when correct response is provided
0xfedc	Initializes challenge value
0xfeea	Re-scrambles the challenge
0xff02	Loads challenge value from memory
0xff0c	Generates the challenge value
0xffa0	Divide and modulo function