
אל תהיה מצחיק, זו רק נורה

מאת אייל איטקין

הקדמה

אני מניח שכולכם כבר מכירים את המונח "האינטרנט של הדברים" (IoT), אבל כמה מכם שמעו אי פעם על נורות חכמות? באמצעות שימוש פשוט באפליקציה בנייד או פקודות קוליות לעוזרת הביתית הדיגיטלית, הנורות מאפשרות שליטה חכמה באורות הפזורים בבית, עד לרמת עוצמת ואפילו צבע התאורה. נורות חכמות אלו מנוהלות לרוב באלחוט מעל רשת ה-WiFi או אפילו באמצעות פרוטוקול ה-ZigBee, פרוטוקול רדיו צר סרט.

לפני מספר שנים, צוות חוקרים ממכון ויצמן הדגים כיצד ניתן להשתלט מרחוק על נורות חכמות ואף ליצור תגובת שרשרת שתאפשר להם לדלג מנורה אחת לאחרת ובכך להרחיב את השליטה שלהם לעוד ועוד נורות ברחבי העיר. המחקר שלהם העלה שאלה מעניינת: למעט החשכה של העיר, או נסיון לעורר התקפי אפילפסיה בקרב התושבים, האם הנורות הללו מהוות איום מוחשי גם על רשת המחשבים עצמה? האם תוקפים יכולים איכשהו לגשר על הפער בין רשת הנורות הפיזית ולדלג ממנה אל מטרות מעניינות יותר כמו רשת המחשבת הביתית, הארגונית או אפילו רשת הניהול של העיר החכמה?

והתשובה הקצרה היא: כן.

בהמשך ישיר למחקר הקודם, אנחנו בחרנו ללכת ישר לליבה עצמה: הבקר החכם שמתפקד כגשר בין רשת ה-IP (רשת המחשבים) ורשת ה-ZigBee (רשת הרדיו של הנורות). באמצעות התחזות לנורות לגיטימיות ברשת, הצלחנו לבסוף לנצל חולשות שמצאנו בבקר ושאפשרו לנו לחדור אל רשת המחשבים הנכספת באמצעות תקיפה אלחוטית מרוחקת מעל פרוטוקול ה-ZigBee.

להלן סרטון ההדגמה של התקיפה: https://youtu.be/4CWU0DA_by

מחקר זה נעשה באמצעות עזרה והדרכה של מכון צ'ק פוינט לאבטחת מידע (CPIIS) באוניברסיטת תל אביב.

מתחילים

לאחר שסיימתי את המחקר הקודם ([חייכו, אכלתם אותה - איך התקנתי ransomware על המצלמה הדיגיטלית שלכם](#)) רציתי להרחיב את התמיכה של הדיבאגר שלי ([Scout](#)) לארכיטקטורות נוספות כדוגמת Mips. הצעד הטבעי כמובן היה לנסות ולהתחיל במחקר בארכיטקטורת Mips ולכן הלכתי לטוויטר כדי לשאול אנשים האם יש להם רעיון טוב למחקר חולשות שכזה.

כמו שקורה לרוב, התגובות כללו מספר כיווני מחקר מעניינים, כאשר המבטיח מכולם הגיע, איך לא, מחבר מתקופת השירות הצבאי: אייל רון ([@eyalr0](#)). זה עולם די קטן בסה"כ, וכך יצא שבצירוף מקרים משעשע אייל נושא כיום במשרת מחקר ב-CPIIS. היות והדבר עלול להוביל למספר אי הבנות בקריאת הכתוב בהמשך, אנסה להבהיר זאת מראש כעת, ומאוחר יותר אהיה פורמאלי יותר מהרגיל ואתייחס לאייל רון בשמו המלא כדי שלא לבלבל את הקוראים:

- אני (הכותב), אייל איטקין, חוקר בצוות מחקר החולשות בקבוצת המחקר של חברת צ'ק פוינט
- המחקר נעשה בהדרכתו של אייל רון, כיום ד"ר במכון המחקר על שם צ'ק פוינט באוניברסיטת תל אביב

ואחרי שסטינו מעט מהנושא והבהרנו נקודה זו, בואו נחזור חזרה לרעיון המחקר שאייל רון הציע. אייל רון הציע כי אמשך מחקר שביצע בעבר כחלק מהדוקטורט שלו במכון ויצמן ואשר מתואר בפירוט בפרק הבא: "מחקרים קודמים בנושא". היות והם הצליחו להשתלט אך ורק על הנורות עצמן, הוא מאמין כי אפשר יהיה למנף את האחיזה בנורות בכדי לתקוף את הבקר אשר מחבר את רשת ה-ZigBee ורשת ה-IP. כלומר, אפיק תקיפה חדש זה יאפשר לתוקף לחדור אל רשת המחשבים מהאוויר באמצעות רשת הרדיו המשמשת את הנורות החכמות.

מחקרים קודמים בנושא

במחקר הנושא את השם: [IoT Goes Nuclear: Creating a ZigBee Chain Reaction](#) (בתרגום חופשי: "האינטרנט של הדברים נהיה גרעיני: יצירת תגובת שרשרת ב-ZigBee"), צוות חוקרים הכולל את אייל רון ([@eyalr0](#)), ([@colinoflynn](#)) Colin O'Flynn, עדי שמיר ואחרים, ניתח את האבטחה של מנורות חכמות הנשלטות באמצעות ZigBee. החוקרים בחרו להתמקד בנורות ובבקר מדגם [Philips Hue](#), והדיגמו מספר תקיפות כנגדן:

- תוקפים יכולים "לחטוף" נורה מרשת ZigBee נתונה, ולהכריח אותה להצטרף אל הרשת שלהם. ההדגמה נעשתה באמצעות רחפן (War-Flying) ממרחק של כ-400 מטר:

<https://www.youtube.com/watch?v=Ed1OjAuRARU>

- בשל בעיית מימוש, גם נורה "רגילה" יכולה לבצע את ההתקפה שתוארה קודם לכן, ו"לחטוף" נורות מרשתות שכנות
 - תוקפים שחולקים רשת משותפת עם נורה נתונה יכולים לשלוח לה עדכון תוכנה עוין, ובכך להשתלט עליה לחלוטין
- לטענת החוקרים, 3 התקיפות הללו מובילות לכך שבאמצעות השתלטות על קומץ נורות שנבחרו בקפידה על סמך מיקומן הגיאוגרפי בעיר נתונה, הם יכולים ליצור תגובת שרשרת (הדומה לתגובה גרעינית) שתוביל להשתלטות על כל הנורות החכמות שבעיר.

בשל אופי התקיפות שהודגמו, היצרן תיקן אך ורק את התקיפה השנייה, ובכך השאיר לנו שתי יכולות:

1. היכולת "לחטוף" נורה מרשת נתונה בסמיכות גיאוגרפית (כ-400 מטר)
2. היכולת לעדכן את התוכנה לנורה הנ"ל, ובכך להשתמש בה כקרח קפיצה לשלב הבא של התקיפה

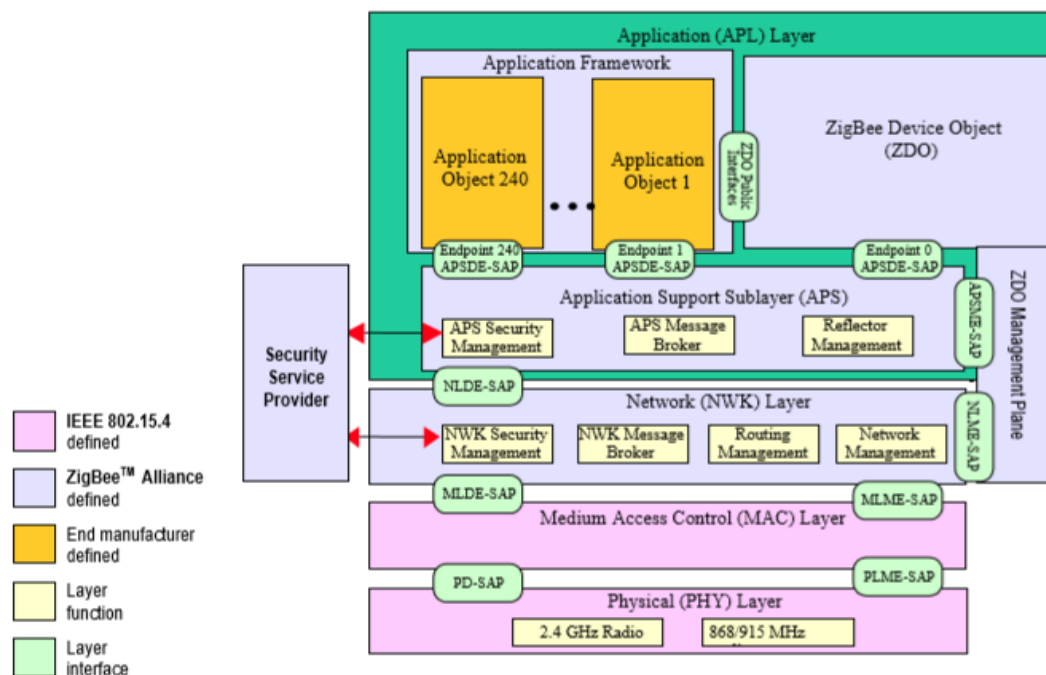
לאחר הסבר מפורט על המחקר שביצעו עוד ב-2016, ולאחר שציידו אותי בבקר Philips Hue Bridge שאיכשהו נשאר מאותו המחקר, איחלו לי בהצלחה ושלחו אותי לדרכי. זמן להתחיל.

מבוא ל-ZigBee

עמוד הויקיפדיה של [הפרוטוקול](#) מעיד כי "ZigBee הוא תקן מבוסס IEEE 802.15.4 למשפחה של פרוטוקולים המיועדים ליצור... רשתות תקשורת אד-הוק צרות סרט, בסמיכות גיאוגרפית ובעלות צריכת מתח נמוכה". שלא להתבלבל עם WiFi (IEEE 802.11), [IEEE 802.15.4](#) הוא התקן הטכני לפרוטוקול רשת מבוסס רדיו המתפקד כשכבות 1-2 במחסנית התקשורת של ZigBee, בהתאם למודל שכבות התקשורת (OSI Model).

רק כדי לקבל מושג טוב עד כמה פרוטוקול זה הוא צר-סרט, יחידת השליחה המקסימאלית (MTU) עבור הודעה בודדת מעל IEEE 802.15.4 היא **127 בתים**. אני חוזר שוב, ההודעה המקסימאלית, כולל כל כותרות התקשורת, לא תעלה על 127 בתים. מה שזה אומר בגודל, זה שבמידה ולא נעשה שימוש בפרגמנטציה של הודעות (שבירתן ל"רסיסי הודעות" והרכבתן מחדש בצד המקבל), ההודעות בפרוטוקול ה-ZigBee יהיו מוגבלות למדי בגודל. בתקווה, הגבלה זו לא תפריע יתר על המידה בהשמת החולשות שאנו מקווים למצוא במימוש עצמו.

מעל שכבת הרדיו הצרה שמגדיר התקן, ZigBee מגדיר מחסנית תקשורת הכוללת מספר שכבות תקשורת שונות, כפי שניתן לראות באיור הבא שנלקח מ**מסמכי התקן**:



בקצרה, אנחנו יכולים לחלק את השכבות בצורה גסה ל-4 קטגוריות (בסדר עולה):

1. **השכבה הפיזית (MAC)** - הודעות רדיו המוגדרות על ידי IEEE 802.15.4
2. **שכבת הרשת (NWK)** - אחראית על ניתוב, ממסור ואבטחה (הצפנה)
3. **שכבה טרום-אפליקטיבית (APS)** - מנתבת את ההודעה לשכבה האפליקטיבית הרצויה
4. **שכבת האפליקציה (ZCL, ZDP, וכו')** - השכבה האפליקטיבית הלוגית, מחולקת למספר אפליקציות בהתאם לסוג ההודעה הנכנסת

קצת מונחים:

- **ZDP - ZigBee Device Profile**
- **ZCL - ZigBee Cluster Library**

למי מכם ששמע בעבר על פרוטוקול [SNMP](#), ZCL יראה כמו קידוד שונה לאותו ממשק לוגי. שכבת ה-ZCL מאפשרת לחברים השונים ברשת לתשאל (READ_ATTRIBUTE) ולענות (WRITE_ATTRIBUTE) בנוגע לאוסף שדות קונפיגורציה הנשמרים כ"צבירים" (Clusters). על כן, שכבה זו מאפשרת למתפעל (הבקר) לשלוט בנורות השונות. שדות קונפיגורציה לדוגמה הם:

- הגבלה פיזית מינימאלית/מקסימאלית לטמפרטורת הנורה
- ערך הצבע האדום/ירוק/כחול של הנורה
- וכו'

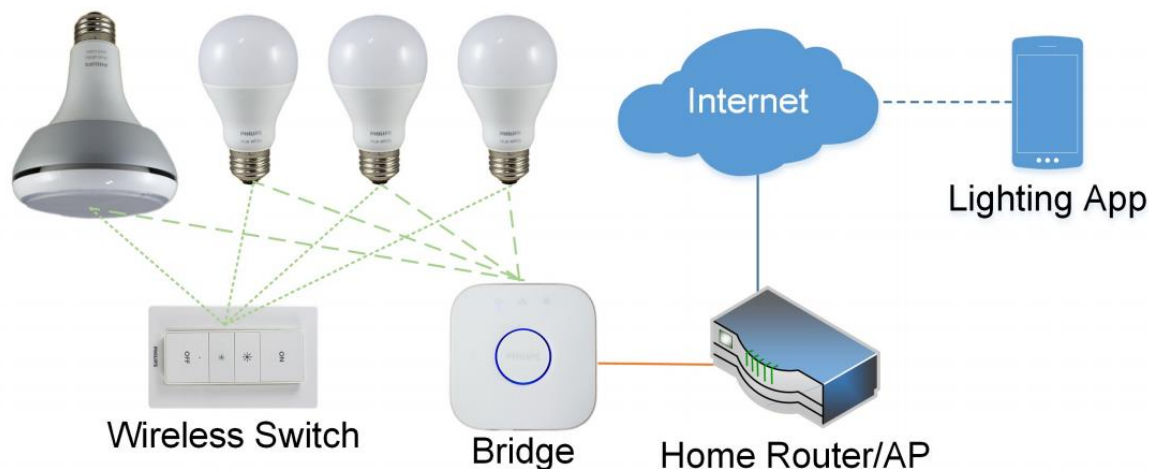
דרך הדוגמה שבדיוק ראינו ניתן לראות שלא מדובר בנורות רגילות בצבעי לבן/צהוב. נורות חכמות אלו תומכות במגוון רחב של צבעים, אשר נבחרים באמצעות קידוד RGB.

והמוצר אותו נתקוף הפעם הוא...

המוצר אותו נתקוף במחקר זה הוא סט המוצרים העונה לשם [Philips Hue](#), ובצורה יותר ספציפית, בקר הנורות [Philips Hue Bridge](#). הערת צד: סט המוצרים התחיל במקור תחת חטיבת התאורה של חברת Philips, ומכאן שמן, וכעת הן ממותגות תחת חברה-בת בשם [Signify](#). היות וזה יכול להיות מעט מבלבל, מעתה ואילך ננסה להתייחס ליצרן פשוט כ"היצרן", אבל אם בטעות נכתוב "Philips" הרי שאנחנו ככל הנראה מתכוונים ל"Signify".

בעוד שנורות "חכמות" הן לא כל כך פופולאריות בישראל, מצאנו שבמדינות רבות, ובעיקר באירופה, המצב שונה לחלוטין. למשל, במאמר [זו](#) משנת 2018, מצאנו ש-Philips Hue שולטים ב-31% משוק התאורה החכמה בבריטניה, עם מעל 430,000 משקי בית אשר משתמשים בנורות חכמות. למעשה, כאשר הדגמנו את המחקר לחלק מהנהלה הבכירה בחברה, הופתענו לגלות שכל התאורה בבתים שלהם היא מהמותג של Philips Hue.

באיור הבא, שנלקח מהמחקר המקורי, מוצגת ארכיטקטורת הרשת של בית או משרד המשתמשים במוצר:



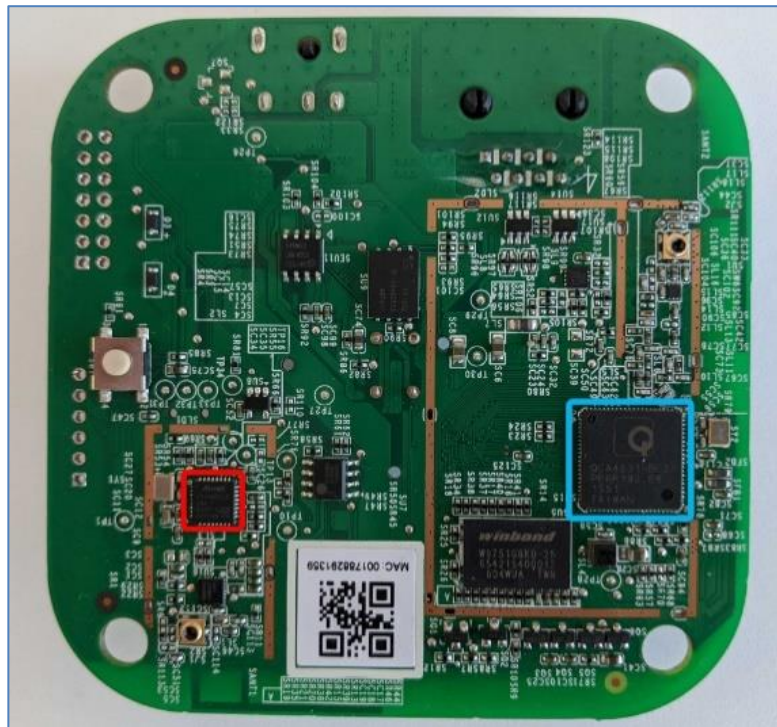
[מקור: <https://eprint.iacr.org/2016/1047.pdf>]

כאשר ZLL מתאר את שכבת ה-ZigBee שהותאמה למוצרי תאורה: ZigBee Light Link.

מצד אחד, יש לנו מוצרים תומכי ZigBee: הנורות והבקר. מהצד השני, יש לנו את רשת המחשבים ה"קלאסית" בה חברים: המחשב האישי, הפלאפון הנייד ושוב, הבקר. כפי שמשתמע משמו באנגלית "Bridge" (גשר), הבקר הוא המוצר היחיד שחבר בו-זמנית בשתי הרשתות, ותפקידו לגשר בין הרשתות ולתרגם פקודות שנשלחות מהאפליקציה של המשתמש להודעות רדיו שישלחו אל הנורות עצמן.

ארכיטקטורת הבקר

בעוד שידענו עוד במקור כי הבקר משתמש במעבד Mips, הרי זו הסיבה שבחרנו בו מלכתחילה, מסתבר כי הארכיטקטורה שלו מורכבת יותר משחשבונו תחילה. באיור הבא ניתן לראות את לוח האם של הבקר (דגם חומרה 2.0) לאחר שהוצאנו אותו ממארז הפלסטיק:



- בצד ימין (מסומן בכחול) ניתן לראות את המעבד הראשי: [QCA4531-BL3A](#).
 - בצד שמאל (מסומן באדום) ניתן לראות מעבד עזר של Atmel ([ATSAMR21E18E](#)) המממש את שכבות התקשורת התחתונות של מחסנית ה-ZigBee.
- מטבעו, אנו נתייחס למעבד ה-Atmel בתור "מודם", היות והמעבד הראשי מאציל אליו את הטיפול בשכבות התקשורת התחתונות של הודעות נכנסות/יוצאות. המשמעות של חלוקה לוגית זו היא שגם השכבה הפיסית (MAC) וגם שכבת התקשורת (NWK) יטופלו על ידי המודם, שבתורו יתשאל את המעבד הראשי כאשר יעלה הצורך בערכי קונפיגורציה שונים עבור שכבות אלו.
- להפתעתנו הרבה, המעבד הראשי מריץ את מערכת ההפעלה לינוקס, ולא מערכת הפעלה רזה כלשהיא (RTOS). שינוי זה הוא די מרענן אל מול הפרויקטים עליהם עבדנו בתקופה האחרונה. בנוסף, תכונה זו עזרה לנו בצורה משמעותית בתהליך חילוץ התוכנה (Firmware) ומאוחר יותר גם בדיבוג הקוד שרץ מעל המעבד שכאמור אחראי על לוגיקת הליבה של המוצר.

כפי שתיאר בפירוט [באתר שלו](#), Colin O'Flynn ([@colinoflynn](#)), מתאר איך להתחבר אל הממשק הסיריאלי החשוף שבלוח האם על מנת להשיג גישה בהרשאות גבוהות (root) אל המערכת. המדריך של Colin הוא מדריך מעולה ומומלץ לכל מי שמתמודד עם מכשירי Embedded המריצים את מערכת



ההפעלה לינוקס, ובייחוד מתמקד ב-Bootloader מסוג U-Boot. לצערי, לא היה לי את הציוד הדרוש בכדי להתחבר לממשק הסיריאלי, דבר שלמדתי על בשרי לאחר שעות של נסיונות כושלים לשחזר את התוצאות של Colin. למזלי, התייעצתי עם אחי הקטן (ירון איטקין) שעזר לי להבין איזה [כבלים](#) אני צריך, ומיד לאחר מכן הזמנתי אותם ברשת.

בנתיים, בזמן שלוקח לציוד שהזמנו להגיע מחו"ל, התחלתי לפרוש¹ את התוכנה הישנה (תוכנה משנת 2016) שקיבלתי מאייל רונן.

ipbridge

התהליך הראשי שרץ על המעבד הראשי נקרא "ipbridge". בדיקה בסיסית של הבינארי העלתה שמדובר במקרה קלאסי של קימפול קובץ ELF:

- הקוד עצמו מקומפל להיות תלוי מיקום - משתמש בכתובות זיכרון קבועות ולכן לא תומך בטעינה דינאמית.
- הספריות (קבצי ה-so.*), המחסנית וה-heap נטענות בצורה דינאמית אקראית (באחריות מערכת ההפעלה).
- אין שימוש בקנריות מחסנית (stack cookies/canaries) להגנה על כתובת החזרה השמורה במחסנית.
- התהליך רץ בהרשאות גבוהות ("root") - חדשות טובות מאוד עבורינו.

המצב שתיארנו הרגע הוא מצב השכיח מאוד כאשר מתמודדים עם מטרה שרצה מעל מע"ה לינוקס. מערכת ההפעלה מפעילה מספר הגנות כברירת מחדל, אבל היצרן עצמו לרוב לא מנסה להפעיל הגנות נוספות אשר נתמכות על ידי הקומפיילר. משכך, הבינארי לא קומפל לתמוך בטעינה דינאמית אקראית (ASLR), לא הוגדר כ-PIE (Position Independent Executable) וגם לא קומפל לכלול קנריות מחסנית.

מנקודת המבט שלנו כתוקפים, תהליך ההשמשה של חולשה עתידית לא יהיה טריוויאלי, משום שיש אקראיות זכרון (ASLR) בסיסית, אבל הוא גם לא יהיה מורכב יתר על המידה, משום שיש כתובות זכרון קבועות וידועות שנוכל לשלב בהשמשה שנכתוב.

לפני שנוכל להתחיל בתהליך הפרישה, נאלצנו לעצור ולנסות לשפר את איכות ניתוח התוכנה שבוצע על ידי IDA Pro, משום שהמצוקה של התוכנה בהבחנה בין קוד מעבד מסוג Mips ומסוג Mips16 בלטה לעין.

¹ קיים ויכוח ער בקרב קהילת החוקרים בארץ על האיות הנכון למונח "פרישה" המתאר את התהליך שנקרא באנגלית Reverse Engineering. היות ואני לא מתכוון להשתמש במונח "הנדסה לאחור", והיות ובעברית מדוברת קל פשוט לומר "פריסה", הרי שיש כאן קושי. במאמר זה החלטתי להכות על חטא, ולאית את המונח באמצעות "ש" ולא "ס" כפי שעשיתי עד כה במאמריי הקודמים ב-Digital Whisper. חבריי שכנעו אותי שמקור המונח הוא מפרישת ציוד על שולחן ולא מפריסת לחם.

היות ומצב זה דומה מאוד לשני המצבים של מעבדי Arm (Thumb ו-Arm), זה היה זמן טוב לבדוק את [Thumbs Up](#), תוסף IDA שכתבנו בעבר ושעד כה נבדק רק על בינארים למעבדי אינטל ו-Arm. לשמחתנו הרבה, התוסף הצליח מעל המצופה: בעוד שהתחלנו עם 2525 פונקציות, לאחר ההרצה קיבלנו בינארי נקי ומסודר ובו 3478 פונקציות שמזוהות על ידי IDA. עכשיו באמת אפשר להתחיל לעבוד על הבינארי, ללא הצורך לתקן ידנית טעויות ניתוח של התוכנה.

מיד לאחר התחלת הפרישה, נתקלנו במשהו מוזר. מסיבה לא ברורה, זה נראה כאילו התוכנה מצפה שההודעות הנכנסות יקודדו כמחרוזות!?

```
# "Bridge"
.word 1, 0
.word off_53FFE8 # "Version"
.word 4
.word aLink # "Link"
.word 1
.word 0
.word off_53FFAC # "Touchlink"
.word 5
.word aTh # "TH"
.word 1
.word 0
.word off_53FF64 # "Ready"
.word 6
.word aConnection # "Connection"
.word 2
.word 0
.word off_53FF40 # "FindFreePanDone"
.word 3
.word aZdp # "Zdp"
.word 2
.word 0
.word off_53FE80 # "SendMgmtPermitJoiningReq"
.word 0x10
.word aZcl # "Zcl"
.word 2
.word EI_zcl_main_handler #
# $a0 - input buffer (char *)
# $a1 - string name (cmd descriptor ?)
#
```

האיור שלמעלה מציג את רשימת המחרוזות שאנו מצפים לראות בתוך הודעה נכנסת, כאשר כל מחרוזת תוביל לניתוב פנימי של ההודעה לפונקצית הטיפול המתאימה. לדוגמא, ניתן לראות כי המחרוזת "Zcl" תוביל לפונקציה אותה סימנו בשם "EI_zcl_main_handler". בנקודה זו חזרנו שוב למסמכי האיפיון של תקן ה-ZigBee וזה פשוט לא היה הגיוני. הפרוטוקול הוא ממש צר-סרט, והשדות השונים בהודעה מקודדים על גבי ביטים בודדים, בטח שלא באמצעות מחרוזות ארוכות ובזבזניות... למה התוכנה שלנו מצפה לקבל מחרוזות שכאלה?

אחרי קריאה נוספת של תיעוד המחקר של אייל רונן ו-Colin, פתאום הכל נהיה ברור. המודם משחק תפקיד נוסף שממנו התעלמנו בהתחלה: הוא ממיר את ההודעה המקורית מהקידוד הבינארי לייצוג

טקסטואלי, ושולח אותה בממשק Usb-to-Serial אל המעבד הראשי. בצורה זו, כל שהמעבד הראשי צריך לעשות זה לקרוא הודעות טקסטואליות פשוטות לטיפול מממשק סיריאלי המיוחצן כקובץ על ידי מערכת ההפעלה.

Colin גם מצא עדויות לכך שהנורות עצמן עושות שימוש ב-[Atmel BitCloud SDK](#), שכיום הוא קוד סגור שצריך לקנות מ-Atmel. בהינתן ממצא זה, הגיוני להניח כי אותה התוכנה נמצאת גם במודם ומשמשת בתור שכבת "המרה" להודעות הנכנסות, בדרכן אל המעבד הראשי:

1. הודעה נכנסת עוברת פרסור ונבדקת על ידי מחסנית התקשורת של BitCloud.

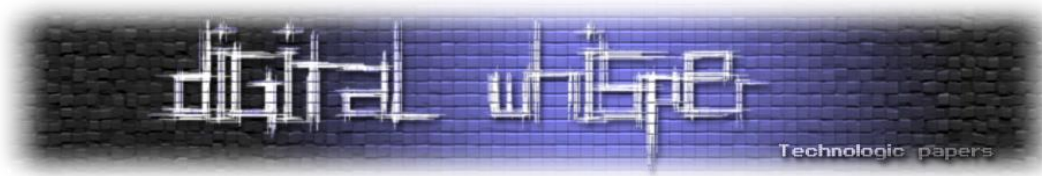
2. ההודעה מומרת לייצוג טקסטואלי נוח.

3. ההודעה בייצוג זה היא זו שנשלחת אל המעבד הראשי לטיפול.

בצורה זו, המפתחים של התוכנה במעבד הראשי לא נדרשים להכיר כל ביט וביט בתקן של ZigBee, והם יכולים להתמקד רק בלוגיקה של הטיפול בהודעות הנכנסות. הטיפול בקידוד וההצפנה המורכבים של ZigBee נעשים בצורה אוטומאטית בקוד של חברת Atmel שרץ על המעבד של Atmel, וחוסכים ליצרן הנורות את כאב הראש הזה.

אם נסתכל על זה מנקודת מבט אבטחתית, מדובר בהפרדה בריאה לתכולות שיש לה יתרונות וחסרונות. יחד עם זאת, להפרדה הנ"ל יש השפעה גדולה מאוד על תהליך המחקר שלנו. לנו יש רק את התוכנה של תהליך ה-ipbridge (ושל יתר התהליכים במעבד הראשי), אותו אנחנו גם יכולים לדבג באמצעות התקנת דיבאגר (gdbserver) בצורה נוחה בסביבת הלינוקס. התוכנה של המודם מוצפנת, וזה לא הולך להיות כל כך פשוט לשחזר את הצעדים של המחקר המקורי, לחלץ את מפתח ההצפנה הסימטרי באמצעות תקיפת ניתוח צריכת מתח (power analysis) ובאמצעותו לפענח את התוכנה.

נראה ששום דבר לא הולך להיות פשוט במחקר הזה. אנחנו פשוט נוסיף את המכשול הזה לרשימת הדברים שנצטרך להתמודד איתם בעתיד, נמשיך הלאה ונקווה לטוב.



חיפוש חולשות - סבב ראשון

עכשיו אחרי שהבנו שהמודם מעביר אלינו הודעות בייצוג טקסטואלי דרך ממשק סיריאלי, עקבנו אחרי זרימת המידע בין החוטים (Threads) השונים וניסינו לחפש חולשות בפונקציות השונות המטפלות בהודעות הנכנסות. די מהר, המאמצים שלנו התמקדו במודול ה-ZCL, היות והוא תומך בקריאה/כתיבה של סוגים רבים של ערכי קונפיגורציה:

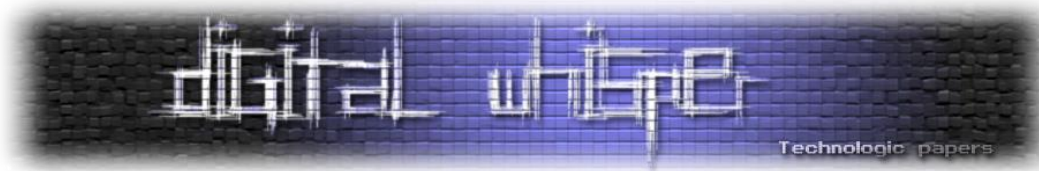
- E_ZCL_UINT32 (0x23) - מספר בייצוג של 4 בתים
- E_ZCL_UINT16 (0x21) - מספר בייצוג של 2 בתים
- E_ZCL_UINT8 (0x20) / E_ZCL_ENUM8 (0x30) - מספר בייצוג של בית / ערך enum בגודל בית
- E_ZCL_BOOL (0x10) - ביט בודד המייצג ערך בוליאני (אמת/שקר)
- E_ZCL_ARRAY (0x48) - מערך בתים בגודל משתנה, מיוצג באמצעות שדה אורך של בית בודד

כפי שבוודאי הבנתם, טיפול בטיפוסים בעלי אורך משתנה, במוצר Embedded, זה מתכון בטוח לחולשות. האירור הבא מציג את קוד האסמבלי שאחראי על הטיפול בטיפוס הנ"ל:

```
LOAD:0048B194
LOAD:0048B194
LOAD:0048B194 24 02 00 02 li $v0, 2
LOAD:0048B198 02 40 28 21 move $a1, $s2
LOAD:0048B19C 02 00 30 21 move $a2, $s0
LOAD:0048B1A0 02 60 20 21 move $a0, $s3
LOAD:0048B1A4 0C 12 E3 04 jal EI_zcl_read_1_byte # read the length field of the array
LOAD:0048B1A8 AE 22 00 0C sw $v0, zcl_attribute.one_if_data_two_if_pdata($s1) # Store Word
LOAD:0048B1AC A2 22 00 10 sb $v0, zcl_attribute.data_type_or_pdata_length($s1) # Store Byte
LOAD:0048B1B0 0C 12 FF C1 jal EI_malloc_with_mutex # malloc(0x2B) - 43 bytes
LOAD:0048B1B4 24 04 00 28 li $a0, 0x2B # '+' # Load Immediate
LOAD:0048B1B8 00 40 20 21 move $a0, $v0
LOAD:0048B1BC 00 00 28 21 move $a1, $zero
LOAD:0048B1C0 24 06 00 28 li $a2, 0x2B # '+' # Load Immediate
LOAD:0048B1C4 0C 10 1E D0 jal memset # memset(pBuffer, 0, sizeof(buffer) = 0x2B)
LOAD:0048B1C8 AE 22 00 14 sw $v0, zcl_attribute.data_or_pdata($s1) # Store Word
LOAD:0048B1CC 8E 24 00 14 lw $a0, zcl_attribute.data_or_pdata($s1) # Load Word
LOAD:0048B1D0 92 25 00 10 lbu $a1, zcl_attribute.data_type_or_pdata_length($s1) #
LOAD:0048B1D0 # EI-DBG: Controlled (1-byte) memcpy into a heap buffer
LOAD:0048B1D0 # EI-DBG: of fixed size 0x2B (43) bytes ==> Heap Buffer Overflow :)
LOAD:0048B1D4 02 60 30 21 move $a2, $s3
LOAD:0048B1D8 AF 00 00 10 sw $s0, 0x38+var_28($sp) # Store Word
LOAD:0048B1DC 0C 12 E3 7A jal EI_zcl_read_blob # read the blob content of the array into the allocated buffer
LOAD:0048B1E0 02 40 38 21 move $a3, $s2
```

הערה: קחו בחשבון שבארכיטקטורת Mips נעשה שימוש במה שנקרא [delay slot](#). כלומר, הערך 0x2B מועבר כארגומנט בקריאה לפונקציה malloc() בפקודת האסמבלי הבאה: li \$a0, 0x2B, שנמצאת לאחר הקריאה לפונקציה. זאת משום שלמעשה פקודת האסמבלי העוקבת לקריאה לפונקציה תבצע לפני שהפונקציה תיקרא בפועל. זו המשמעות של delay slot.

אוקיי, אז מה ראינו בקוד? תוקף יכול לשלוח הודעת תגובה עוינת כמענה להודעת READ_ATTRIBUTE אשר תכיל טיפוס מסוג מערך בתים.



במידה והתוקף יציין שאורך המערך הוא מעל 43 בתים (0x2B) הטיפול בהודעה יגרום לדריסת זכרון נשלטת על ה-heap, ללא שום אילוצי תווים. הגבלות אפשריות לחולשה הן:

- היות ו-ZCL היא שכבה גבוהה יחסית בפרוטוקול, אנחנו יכולים להרשות לעצמנו מערך בגודל מקסימאלי של לכל היותר כ-70 בתים, אחרת ההודעה תחרוג מגודל הודעה מקסימאלית ברדיו.
 - בדיקת מכונת מצבים יכולה לאכוף כי נוכל רק לשלוח מענה לשאילתא שנשלחה מצד הבקר עצמו.
 - בדיקת מכונת מצבים יכולה לאכוף כי נוכל רק לשלוח מענה מהטיפוס אותו ביקש הבקר בשאילתא שנשלחה.
 - המודם עלול להפיל את ההודעה שלנו במידה והוא יחשוב שהיא מפרה את הלוגיקה שלו, לוגיקה אשר מבחינתנו היא "קופסא שחורה" היות ואין לנו את קוד המודם.
- לא בדיוק החולשה הקלה בעולם לניצול, אבל בכל זאת חולשה רצינית.

בתזמון טוב זה סוף סוף הגיעו הכבלים הסיריאליים שהזמנו מחו"ל, אז עכשיו אנחנו יכולים לבדוק האם אכן מצאנו חולשה. עקבנו אחרי המדריך של Colin והשגנו גישת ssh בהרשאות root אל הבקר. לאחר מכן, קימפלנו gdbserver לארכיטקטורת Mips והעלנו אותו לבקר עצמו, כדי שנוכל לדבג את התהליך ipbridge בו מצאנו מה שנראה כמו חולשה. כעת הגענו למכשול חדש: אין לנו משדר רדיו שמסוגל לשלוח את התקיפה שלנו אל הבקר. אחרי התייעצות נוספת עם אייל רונן, הוחלט לקנות את [לוח הפיתוח של Atmel למעבד של הנורה עצמה](#), בדיוק כפי שצוות המחקר שלהם עשה במחקר ההמקורי.

בנתיים, מצאנו מעקף שיאפשר לנו לאמת את קיום החולשה גם מבלי לשלוח תקיפה מעל הרדיו. עם קצת מזל, התקיפה אכן גם תעבוד מאוחר יותר ולא תחסם על ידי המודם. התהליך ipbridge תומך בממשק דיבאג אשר מופעל באמצעות חיבור אל שני named pipes אליהם התהליך מאזין באמצעות חוט ייעודי:

- /tmp/ipbridgeio_in
- /tmp/ipbridgeio_out

בעוד שיכולות הדיבאג המובנות לא עוזרות לנו כלל, ריתכנו את הבינארי בכדי שההודעות שמגיעות על גבי החיבורים הללו יועברו ישירות אל תור ההודעות האחראי על הודעות המגיעות מהרדיו, במקום שיגיעו אל תור ההודעות לטיפול בהודעות דיבאג. כל שצריך לעשות הוא לשנות פקודה בודדת שתתייחס לכתובת (הקבועה וידועה מראש) של תור ההודעות הרצוי, במקום שתתייחס לתור ההודעות לדיבאג.

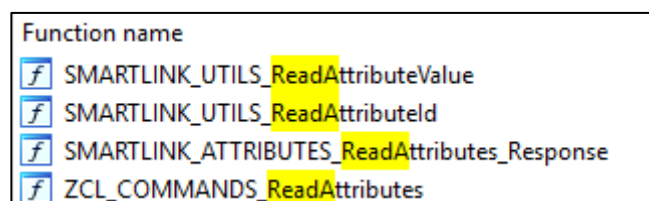
הריתוך הקטן הנ"ל איפשר לנו לקמפל תהליך משלנו שירוך לצד התהליך הראשי ויתקשר איתו מעל ה-named pipes שצינו קודם לכן. התהליך שלנו עכשיו יכול לשלוח הודעות טקסטואליות שיראו כאילו הן הגיעו ישירות מהמודם ולמעשה הרווחנו את היכולת "להזריק" הודעות תקשורת אל המעבד הראשי גם ללא משדר רדיו. אחרי מעט ניסוי וטעייה, ובאמצעות הדיבאגר שלנו, הצלחנו להטריג את החולשה שמצאנו בקוד, ולהוכיח שהבנו את הקוד כהלכה והחולשה אכן קיימת כפי שחשבנו תחילה.

ההגבלה היחידה שעדיין יכולה לחסום אותנו היא שהמודם עלול להחליט שהודעת התקיפה שלנו אינה חוקית, ולהפיל אותה. לשם כך נצטרך לשדר את ההודעה שלנו באוויר ולבדוק מה קורה בפועל.

בעודנו מחכים שלוח הפיתוח שלנו יגיע ויאפשר לנו להתחיל לשדר הודעות, הגיעה חבילה אחרת: קיט שלם של Philips Hue הכולל בקר מדגם חומרה 2.1 יחד עם 3 נורות. זה נראה כמו הזמן הנכון לחלץ את התוכנה מהבקר החדש וגם לעדכן את התוכנה בבקר הישן מדגם 2.0. אחרי הכל, עד עכשיו עבדנו על תוכנה ישנה משנת 2016, ויכול להיות שדברים השתנו מאז.

לצערנו, דברים אכן השתנו.

הדבר הראשון שמשך את העין שלנו היה גודל התוכנה החדשה: מסיבה כלשהי, קובץ ה-`ipbridge` גדל מ-1.2MB עד ל-3.2MB. כשפתחנו את הקובץ ב-IDA ההבדל היה ברור: הבינארי החדש קומפל (בטעות?) עם סמלי `debug`. כלומר, יש לנו את שמות רוב הפונקציות בבינארי, כפי שהן נקראות בקוד על ידי המפתחים. אלה חדשות מצויינות משום שמידע זה יקדם אותנו משמעותית בתהליך הפרישה שכבר התחלנו. האיור הבא מראה חלק משמות הפונקציות שמצאנו בתוכנה החדשה:



הצעד הראשון כמובן היה להצליב את שמות הפונקציות שקיבלנו אל מול השמות שנתנו לאותן הפונקציות בתוכנה הישנה כחלק מתהליך הפרישה. ניכר כי השמות שנתנו היו יחסית מדויקים עד כה, ושם הפונקציה הפגיעה הוא: `SMARTLINK_UTILS_ReadAttributeValue`.

הצעד השני, והחשוב יותר, הוא להעמיק בפונקציה הפגיעה ולבדוק שהחולשה שלנו עדיין שם ושהיא לא נסגרה. לצערנו, חיכתה לנו הפתעה לא נעימה. רשימת טיפוסים המשתנים הנתמכים עודכנה, וכעת היצרן תומך במחרוזות בתים (0x42) במקום במערכי בתים (0x48). ובעוד שמחרוזות בתים הן עדיין בגודל משתנה ובעלות פוטנציאל לחולשת מימוש, ההקצאה כעת השתנתה על מנת לשקף את אופיין של מחרוזות המסתיימות בתו '0':

1. שדה אורך של בית בודד נשלף מההודעה. נציין את האורך שהוא מייצג בתור `L`.
2. חוצץ בגודל `L + 1` מוקצה דינאמית על ה-`heap`.
3. `L` בתים נקראים מתוך ההודעה ומועתקים אל החוצץ הדינאמי שזה אך הוקצה.

זה אומר שאין יותר שימוש בחוצץ בגודל קבוע, ושהשינוי הנ"ל בדיוק סגר לנו את החולשה. זמן למצוא חולשה אחרת.

חיפוש חולשות - סבב שני

אחרי התגלית המצערת בחרנו לעזוב את מודול ה-ZCL, ובהדרגה הגענו אל מודול ה-ZDP ובתוכו אל הטיפול בהודעות נכנסות מסוג תשובות לשאליות LQI - Link Quality Indicator (סמן איכות אות שידור). הודעות אלו הן חלק מהמודול שאחראי על גילוי שכנים ברשת. אחת לכמה זמן, הבקר שולח שאליות לכל הנורות שהוא מכיר, ומתשאל אותן אודות שכנים שהן מכירות על מנת שירחיב את המידע שיש לו אודות הרשת. בעוד ששם ההודעות מتركז באיכות אות הרדיו, בפועל מבנה ההודעה מכיל ברובו את כל מזהי הרשת השונים של השכן עליו מדווחים.

המידע שנשלח אודות כל שכן, כפי שהוא מיוצג בהודעות אלו, הוא:

- **8 בתיים - כתובת מורחבת:** מזהה גלובאלי יחודי (בכל העולם), דומה לכתובות MAC ב-Ethernet.
- **2 בתיים - כתובת מקוצרת:** מזהה מקומי יחודי שתקף רק ברשת הרדיו המקומית בה חברה הנורה.
- **2 בתיים - מזהה PAN:** מזהה הרשת עצמה (Personal Area Network).
- **1 בתיים - ערך LQI:** עריך איכות אות השידור, בטווח של 0-255.
- **3 בתיים - דגלים ושונות:** מגוון דגלים שסוכמים יחד את גודל הרשומה ל-16 בתיים.

היות ונדרש להעביר מידע אודות מספר רב של שכנים (קיבולת המערך הגלובאלי שבבקר היא עד 0x41 רשומות) ההודעות הללו תומכות בפרגמנטציה. בכל הודעת תשובה הנורה עונה לבקר כי היא כרגע עונה על L רשומות, החל מהיסט X ועד $X + L - 1$, מתוך S רשומות סה"כ.

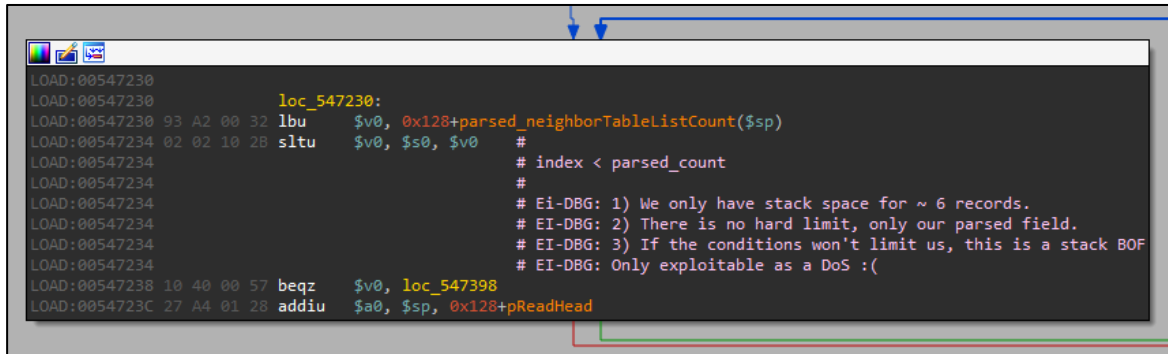
רק כתזכורת, הודעות הרדיו מוגבלות למדי בגודל. ולכן, אתם יכולים לנחש לבד שהשימוש בכמות כזו גדולה של מזהים בכל הודעה, ושליחת מספר רשומות בגודל 16 בתיים כל אחת, די מגבילים את כמות הרשומות שנכנסות בכל הודעה. ניכר שגם המפתחים של המודול הבינו את הנקודה הזו, מה שהוביל אותם לשמור את הרשומות הנכנסות על מערך מחסנית קבוע שיכול להכיל עד 6 רשומות. אולם, אין בדיקות שתפקידן לאמת כי שדה האורך אכן קטן מספיק, מה שעלול להוביל לדריסת זכרון על המחסנית.

אתם בוודאי שואלים את עצמכם איך אנחנו מתכננים לשלוח הודעה "ענקית" שכזו שתכיל מעל 6 רשומות ותדרוס זכרון על המחסנית? ואכן, זו שאלה טובה. בהינתן האילוצים הפיסיים על גדלי ההודעות מעל הרדיו, התקווה היחידה שלנו היא למצוא חולשה במודם, ואז לנצל את האחיזה במודם בכדי לשלוח הודעות גדולות שידרסו זכרון על המעבד הראשי.

אבל, מתאר תקיפה שכזה יוכל לעבוד רק במידה ונמצא חולשה במודם ונצליח להשמיש אותה, וכל זה ללא הקוד שרץ על המודם. לא התוכנית המדהימה בעולם, אבל זאת עוד אופציה במידה ולא נמצא תוכנית מוצלחת יותר.

לפני שננסה צעד הרפתקני שכזה, חזרנו שוב להזריק הודעות באמצעות המעקף שתיארנו קודם לכן, וניסינו להטריג את החולשה שמצאנו. לצערנו, כתובת החזרה השמורה על המחסנית שמורה בדיוק בהיסט עליו אנחנו לא שולטים בצורה טובה בזמן הדריסה. הדריסה שלנו נובעת מתהליך פרסור של שדות מההודעה הנכנסת, ושמירתם אחד-אחד ב-struct שמייצג כל רשומה נכנסת. רצה הגורל, והשדה שישמר בהיסט של כתובת החזרה יהיה תמיד המספר 4.

גזר דין: לא נציל. מה שגם חסך לנו את הצורך בחיפוש חולשה נוספת במודם.



```

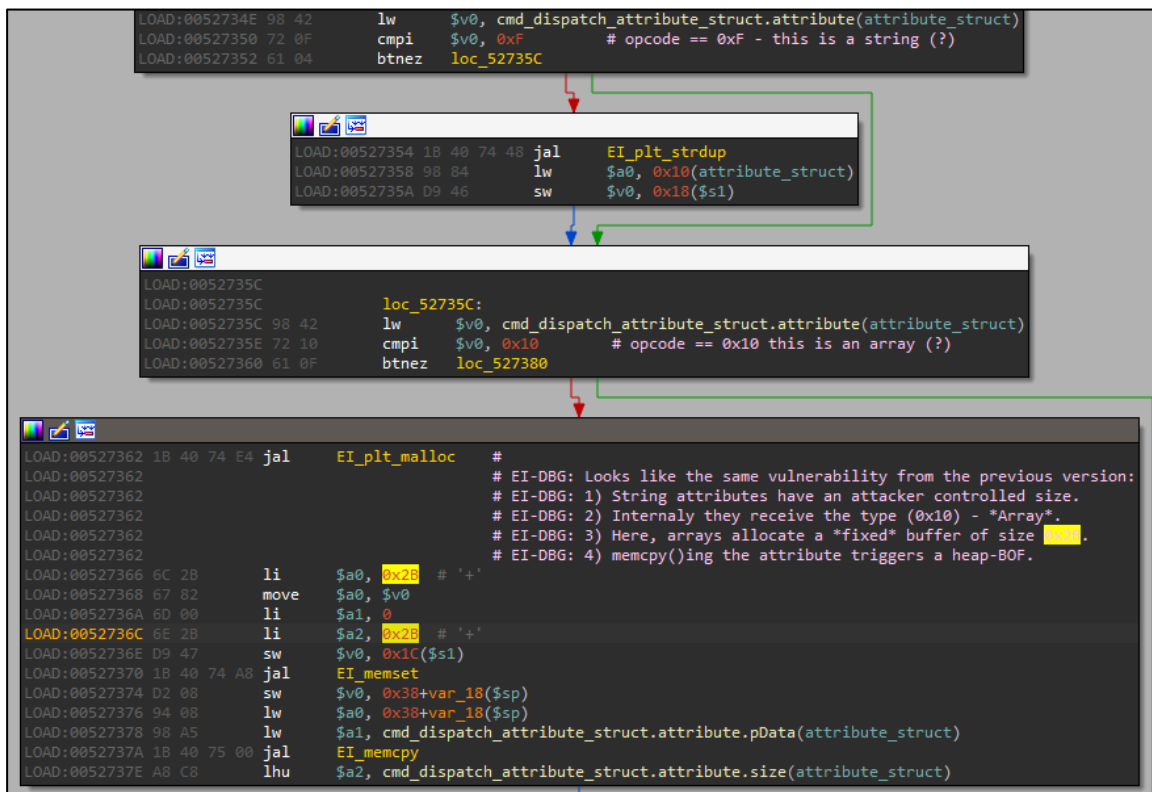
LOAD:00547230
LOAD:00547230      loc_547230:
LOAD:00547230 93 A2 00 32 lbu     $v0, 0x128+parsed_neighborTableListCount($sp)
LOAD:00547234 02 02 10 28 sltu     $v0, $s0, $v0      # index < parsed_count
LOAD:00547234                                     #
LOAD:00547234                                     # Ei-DBG: 1) We only have stack space for ~ 6 records.
LOAD:00547234                                     # EI-DBG: 2) There is no hard limit, only our parsed field.
LOAD:00547234                                     # EI-DBG: 3) If the conditions won't limit us, this is a stack BOF
LOAD:00547234                                     # EI-DBG: Only exploitable as a DoS :(
LOAD:00547238 10 40 00 57 beqz     $v0, loc_547398
LOAD:0054723C 27 A4 01 28 addiu     $a0, $sp, 0x128+pReadHead
  
```

כהערת צד, ההגבלה על כמות הרשומות המקסימאלית בהודעה, כפי שהיא מוגדרת ב-BitCloud SDK, היא: 2. פרוטוקול ה-ZigBee כולל מספר רב של מזהים בכל רסיס הודעה, על מנת לשלוח לכל היותר 2 רשומות בכל פעם. אין ספק, לא התכנון היעיל ביותר בעולם.

CVE-2020-6007 - סבב שלישי - חיפוש חולשות

פעם שלישית גלידה. אחרי שסיימנו לעבור על הקוד של כל המודולים השונים עלתה במוחנו שאלה מעניינת: מי מטפל בכל ערכי ה-ZCL שהועברו בתשובות ששלחנו לבקר? בעוד הפרסור של התשובות כבר לא פגיע, לאן מועברות ההודעות לאחר הפרסור הראשוני?

בניסיון לענות על השאלות הללו, מצאנו חוט נוסף בשם `applproc` (קיצור של `Application` ולא של `Apple`), אשר קורא את המבנה המכיל את המידע ששלחנו, בודק בדיקה לא ידועה במכונת מצבים כלשהי, ואם התמזל מזלנו גם מעביר את הטיפול לפונקציה בשם `CONFIGURE_DEVICES_ReceivedAttribute`. האיור הבא מציג את הקוד של הפונקציה הנ"ל:



```

LOAD:0052734E 98 42      lw      $v0, cmd_dispatch_attribute_struct.attribute(attribute_struct)
LOAD:00527350 72 0F      cmpi    $v0, 0xF      # opcode == 0xF - this is a string (?)
LOAD:00527352 61 04      btnez   loc_52735C

LOAD:00527354 18 40 74 48      jal     EI_plt_strdup
LOAD:00527358 98 84      lw      $a0, 0x10(attribute_struct)
LOAD:0052735A D9 46      sw      $v0, 0x18($s1)

loc_52735C:
LOAD:0052735C          lw      $v0, cmd_dispatch_attribute_struct.attribute(attribute_struct)
LOAD:0052735E 72 10      cmpi    $v0, 0x10     # opcode == 0x10 this is an array (?)
LOAD:00527360 61 0F      btnez   loc_527380

EI_plt_malloc #
# EI-DBG: Looks like the same vulnerability from the previous version:
# EI-DBG: 1) String attributes have an attacker controlled size.
# EI-DBG: 2) Internally they receive the type (0x10) - *Array*.
# EI-DBG: 3) Here, arrays allocate a *fixed* buffer of size 0x20.
# EI-DBG: 4) memcpy()ing the attribute triggers a heap-BOF.
LOAD:00527362 18 40 74 E4      jal     EI_plt_malloc
LOAD:00527362          li      $a0, 0x20 # '+'
LOAD:00527366 6C 2B      move    $a0, $v0
LOAD:00527368 67 82          li      $a1, 0
LOAD:0052736A 6D 00          li      $a2, 0x20 # '+'
LOAD:0052736C 6E 2B      sw      $v0, 0x1C($s1)
LOAD:0052736E D9 47      sw      $v0, 0x38+var_18($sp)
LOAD:00527370 18 40 74 A8      jal     EI_memset
LOAD:00527372 D2 08      sw      $a0, 0x38+var_18($sp)
LOAD:00527376 94 08      lw      $a0, cmd_dispatch_attribute_struct.attribute.pData(attribute_struct)
LOAD:00527378 98 A5      lw      $a1, cmd_dispatch_attribute_struct.attribute.pData(attribute_struct)
LOAD:0052737A 18 40 75 00      jal     EI_memcpy
LOAD:0052737E A8 C8      lhu     $a2, cmd_dispatch_attribute_struct.attribute.size(attribute_struct)
  
```

מסיבה לא ברורה, שדה מספרי מחולץ מתוך מבנה הנתונים:

- **0x0F** - ככל הנראה מייצג **מחרוזת**. המידע ישופל באמצעות קריאה לפונקציה `strcpy()`.
- **0x10** - ככל הנראה **מערך בתים**. המידע יועתק באמצעות **בדיקת** אותה לוגיקה פגיעה שראינו בגרסאות התוכנה הישנה.

חזרנו מהר לבדוק כיצד מאותחל מבנה הנתונים שנשלח אל החוט הנ"ל, ולשמחתנו ראינו את מה שמתואר באיור הבא:

```

LOAD:005C0B74 95 15      lw      $a1, 0x50+arg_4($sp)
LOAD:005C0B76 96 16      lw      $a2, 0x50+arg_8($sp)
LOAD:005C0B78 1A E0 E1 18  jal      UTIL_Fetch_uint8
LOAD:005C0B7C 94 14      lw      $a0, 0x50+arg_0($sp)
LOAD:005C0B7E 67 22      move     $s1, $v0
LOAD:005C0B80 6A 10      li      $v0, 0x10
LOAD:005C0B82 41 81      addiu   $a0, $s1, 1
LOAD:005C0B84 D8 40      sw      $v0, 0($s0)
LOAD:005C0B86 1B 00 0C 73  jal      OSA_MEMORY_Malloc
LOAD:005C0B8A C8 24      sh      $s1, 8($s0)
LOAD:005C0B8C 67 82      move     $a0, $v0
LOAD:005C0B8E 6D 00      li      $a1, 0
LOAD:005C0B90 67 D1      move     $a2, $s1
LOAD:005C0B92 1B 40 A5 58  jal      EI_plt_memset
LOAD:005C0B96 D8 43      sw      $v0, 0xC($s0)
LOAD:005C0B98 93 16      lw      $v1, 0x50+arg_8($sp)
LOAD:005C0B9A 98 83      lw      $a0, 0xC($s0)
LOAD:005C0B9C 96 14      lw      $a2, 0x50+arg_0($sp)
LOAD:005C0B9E 97 15      lw      $a3, 0x50+arg_4($sp)
LOAD:005C0BA0 D3 04      sw      $v1, 0x50+var_40($sp)
LOAD:005C0BA2 1A E0 E1 63  jal      UTIL_Fetch_Buffer
LOAD:005C0BA6 67 B1      move     $a1, $s1
LOAD:005C0BA8 10 33      b       loc_5C0C10
    
```

נראה שהתבצע מעבר לא מוצלח בין תמיכה במערכים לתמיכה במחרוזות, והמחרוזות הנכנסות מסומנות בטעות כ"מערך" (0x10) במקום "מחרוזת" (0x0F) וכך הן יטופלו בהמשך דרכן. המשמעות היא ששוב מצאנו **דריסת זכרון נשלטת על ה-heap**, ואפילו הצלחנו להטריג אותה באמצעות המעקף שבנינו להזרקת הודעות.

עכשיו כשיש לנו חולשה בגרסת התוכנה העדכנית, עדיין נשאר לנו לבדוק מה היא אותה בדיקת מכונת מצבים שקוראת לפני הקריאה לפונקציה הפגיעה. זמן טוב לפתוח את החבילה עם לוח הפיתוח שבדיוק הגיע, כדי שנוכל להתחיל לשדר הודעות משלנו אל הבקר.

מרחחים רמזים

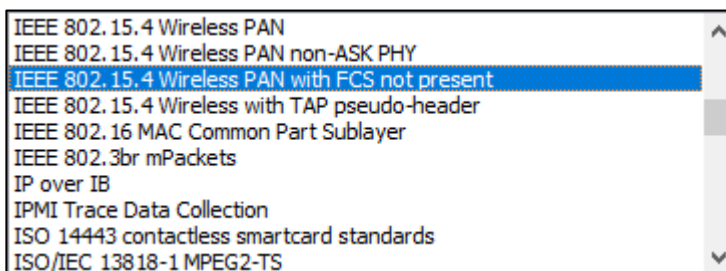
חשוב לציין שבחרנו ספציפית בלוח הפיתוח [ATMEGA256RFR2-XPRO](#) ממספר סיבות:

- במידה ונצליח לבצע את התקיפה כולה מהלוח, נוכיח שהתקיפה ישימה גם מנורה פשוטה, היות ושניהם משתמשים בדיוק באותן יכולות חומרה: מעבד, אנטנה וכו'.
 - קיוונו שנוכל להציל מעט קוד C מהמחקר המקורי של אייל רונן ו-Colin O'Flynn (תקווה שהתבדתה בסופו של דבר).
- דווקא הנקודה הראשונה התבררה כנקודה המכריעה, אבל לכך נגיע בהמשך.

הייתם מצפים שכאשר אתם קונים מוצר של Atmel, שמגיע עם סביבת פיתוח מבוססת Visual Studio שנושאת את השם "Atmel Studio" אז יהיה פשוט ליצור פרויקט ZigBee לדוגמא שמסניף הודעות ומדפיס אותן לסיריאל למשל. ציפיות לחוד ומציאות לחוד. אחרי מספר שיטוטים ב-Google, הצלחנו למצוא ש-Atmel שיחררו מספר סרטוני הדרכה ל-YouTube, וביניהם הסרטון [הזה](#). בסרטון הנ"ל ניתן לראות אדם המפליג על יאכטה (ברצינות, זו לא בדיחה), המדריך אנשים כיצד להשתמש במנהל ההרחבות של סביבת העבודה בכדי להוריד מארז הרחבה שיאפשר ליצור פרויקט "אלחוט" לדוגמא. בינגו, בדיוק מה שרצינו.

עכשיו כשביכולתנו להסניף הודעות, חיברנו בין נורה והבקר (תהליך שנקרא "Commissioning") והדפסנו את ההודעות שקיבלנו אל חיבור הסיריאל. בשלב זה הבנו שבעוד שיש לנו הודעות מוקלטות, אין לנו דרך הגיונית לקרוא אותן לפורמט שיהיה קריא לבני אדם.

ניסינו מגוון תוכנות קוד-פתוח המיועדות ל-ZigBee אבל כולן אכזבו. מה שכן הצלחנו לעשות, זה לטעון הודעות בייצוג hex ישירות אל Wireshark, באמצעות שימוש באופציית ההכנסה (Encapsulation) הבאה:



הערה חשובה: Wireshark יכשל בניית הודעות שכוללות שדה FCS (Frame Check Sequence) לא תקין. בנוסף, כאשר אנחנו משדרים את ההודעות, השדה הנ"ל מחושב ומתווסף בצורה אוטומאטית על ידי האנטנה עצמה (או יותר נכון מודול ה-MAC הפיזי). לכן, מומלץ להפיל את השדה הנ"ל מההודעות הנכנסות ולבחור מראש באופציה של Wireshark בה מצויין כי השדה אינו קיים (כפי שמודגש בכחול באיור שלמעלה). בחירה זו מקלה משמעותית על הצגת הודעות נכנסות / יוצאות ללא התעסקות מיותרת עם השדה שנועד לאיתור שגיאות בהודעה.

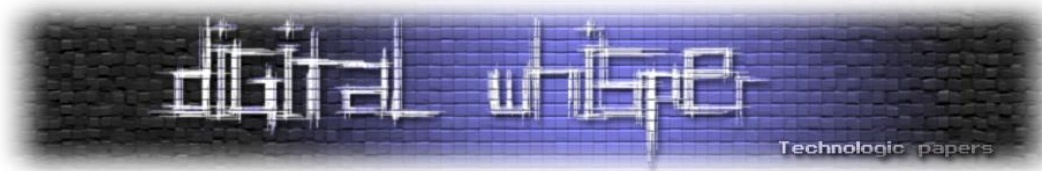
גם מבט חטוף בשיחה שהקלטנו בין הנורה והבקר מבהיר שחסרים לנו חלקים משמעותיים:

- רוב ההודעות מוצפנות, ולנו אין את המפתח.
- חלק מהודעות תהליך ה-Commissioning חסרות, היות וגם השיחה הגלויה נראית שבורה.

כפי שרמזנו קודם לכן, הבחירה שלנו בלוח הפיתוח הספציפי הייתה מכרעת. הפרוטוקול משדר את ההודעות כל כך מהר שאין לנו את היכולת להעביר אותן מעל חיבור סיריאלי איטי אל המחשב בכדי שהוא ידפיס אותן / יבצע לוגיקה כלשהי, היות והחיבור הסיריאלי יוצר עיכוב מיותר.

בקצרה, בזמן שאנחנו שולחים למחשב הודעות להדפסה, אנחנו מפספסים הודעות אחרות שנשלחות בנתיים באוויר. נראה שנצטרך לממש את התקיפה כולה על לוח הפיתוח (בשפת C) אם נרצה שיהיה לנו סיכוי כלשהו אפילו לעמוד בקצב ההודעות הגבוה של פרוטוקול ה-ZigBee.

בנתיים, אגרנו את ההודעות לחוצץ גלובאלי בזכרון, ואז שלחנו אותן למחשב רק בכל פעם שהחוצץ התמלא. בצורה זו יכלנו להקליט את הרוב המוחלט של ההודעות מהאוויר, אך עדיין פספסנו פה ושם הודעות כאשר גם הנורה וגם הבקר שידרו הודעות יחד בפרקי זמן קצרים.



מקלים את שכבת הצופן

Wireshark תומך באופציה של פענוח אוטומאטי של הודעות ZigBee במידה ומספקים לו מראש את מפתחות ההצפנה. נראה שזה זמן טוב להבין איך בכלל עובדת ההצפנה בפרוטוקול. אחרי קריאה נוספת בתקן, נראה שיש 2 מפתחות² חשובים:

- **מפתח תעבורה (Transport)** - מפתח גלובאלי (Broadcast) שמשותף לכל מכשירי ה-ZigBee בעולם(!).
- **מפתח רשת (Network)** - מפתח ייחודי לכל רשת אשר מופץ על ידי הבקר בזמן ההתחברות לרשת (כחלק מתהליך ה-Commissioning).

האיור הבא מציג תעבורת ZigBee לדוגמא כפי שהיא מוצגת באתר של Wireshark, כאשר הודעת ה-Transport Key מסומנת:

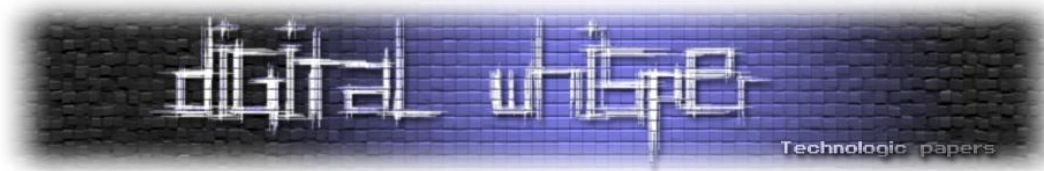
15	17.015...	00:1c:da:ff:ff:00:20:07	0x0000	IEEE 802.15.4	21 Association Request, FFD
16	17.265...			IEEE 802.15.4	5 Ack
17	17.515...	00:1c:da:ff:ff:00:20:07	0x0000	IEEE 802.15.4	18 Data Request
18	17.765...			IEEE 802.15.4	5 Ack
19	18.015...	00:0d:6f:00:00:0d:c5:58	00:1c:da:ff...	IEEE 802.15.4	27 Association Response, PAN: 0x01ff Addr: 0x2c4d
20	18.265...			IEEE 802.15.4	5 Ack
21	18.515...	0x0000	0x2c4d	ZigBee	65 APS: Command
22	18.765...			IEEE 802.15.4	5 Ack

היות ואין לנו את מפתח התעבורה, אין לנו יכולת לפענח את ההודעה הנ"ל והיא מוצגת בפשטות כהודעה "כללית" מסוג APS. חשוב להבין שהודעה זו היא ההודעה בה הבקר מפיץ לנו את מפתח הרשת, ובלעדיה אין לנו יכולת להמשיך ולפענח את יתר ההודעות.

למרות שמצאנו לא מעט מפתחות כשחיפשנו על הנושא באינטרנט, אף אחד מהמפתחות לא עבד. אבל, נראה שאנחנו לא הראשונים שהיו צריכים להתמודד עם הבעיה, משום שבסופו של דבר הצלחנו למצוא את הבלוג ה**זו** בו מתואר הפתרון לבעיה.

נראה שהמפתחות ה"רגילים" משמשים למה שנקרא "Touchlink Commissioning" בעוד שאנחנו מבצעים בפועל "Classic Commissioning" ולשם כך נעשה שימוש במפתחות שונים לחלוטין. למזלנו, המפתחות לשני התרחישים כתובים בפירוט בבלוג והם אכן עבדו במקרה שלנו.

² טכנית יש 3 מפתחות חשובים, היות והבקר אחראי לתאם מול כל נורה מפתח תקשורת ייחודי (Unicast) שישימש רק את שניהם לתקשורת מוצפנת בתוך הרשת. נראה שאף אחד לא באמת מימש את המודול המורכב הזה, ובפועל כל החברים ברשת שולחים הודעות מוצפנות אחד לשני באמצעות שימוש במפתח הרשת המשותף לכולם.



הפעם הצלחנו לפענח בהצלחה את השיחה שהקלטנו, ובאיור הבא תוכלו לראות את ההודעה שפענחנו עכשיו כשהשתמשנו במפתח התעבורה הנכון:

5	0.000004	Broadcast	IEEE 802.15.4	10 Beacon Request
6	0.000005	0x0001	ZigBee	28 Beacon, Src: 0x0001, EPID: 65:b7:92:58:d6:b7:19:f2
7	0.000006	0x0001	Broadcast	48 Permit Join Request
8	0.000007	00:17:88:01:04:16:39:8a	0x0001	IEEE 802.15.4
9	0.000008		IEEE 802.15.4	21 Association Request, FFD
10	0.000009	00:17:88:01:04:16:39:8a	0x0001	5 Ack
11	0.000010		IEEE 802.15.4	18 Data Request
12	0.000011	00:17:88:01:01:69:13:59	00:17:88:01...	5 Ack
13	0.000012		IEEE 802.15.4	27 Association Response, PAN: 0x949f Addr: 0x0016
14	0.000013	0x0001	0x0016	5 Ack
15	0.000014		ZigBee	73 Transport Key
			IEEE 802.15.4	5 Ack

```

> Frame 14: 73 bytes on wire (584 bits), 73 bytes captured (584 bits) on interface 0
> IEEE 802.15.4 Data, Dst: 0x0016, Src: 0x0001
> ZigBee Network Layer Data, Dst: 0x0016, Src: 0x0001
▼ ZigBee Application Support Layer Command
  > Frame Control Field: Command (0x21)
    Counter: 70
  > ZigBee Security Header
  ▼ Command Frame: Transport Key
    Command Identifier: Transport Key (0x05)
    Key Type: Standard Network Key (0x01)
    Key: 6b65a36b4d32892630b435b52b9f52ee
    Sequence Number: 0
    Extended Destination: PhilipsL_01:04:16:39:8a (00:17:88:01:04:16:39:8a)
    Extended Source: ff:ff:ff:ff:ff:ff:ff:ff (ff:ff:ff:ff:ff:ff:ff:ff)
  
```

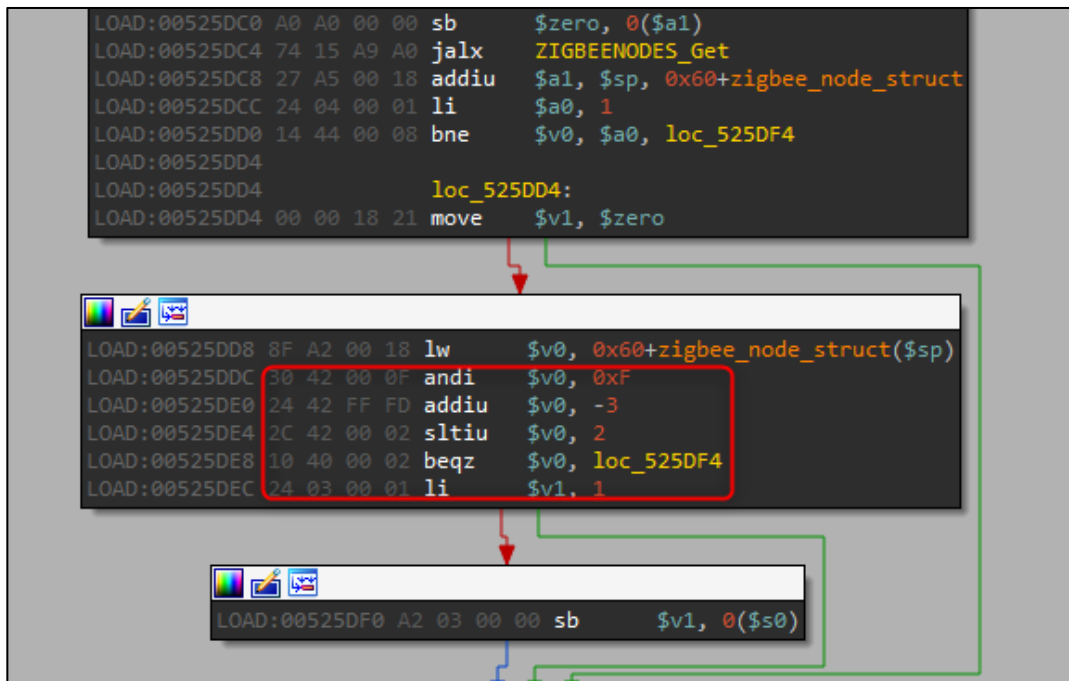
הערה: בחרנו בכוונה לצרף את מפתח הרשת האמיתי במקום לטשטש אותו מהתמונה. מאוחר יותר גם נספק קישור לקובץ הקלטה (.pcap). מלא של כל תהליך ה-Commissioning אל מול הבקר בו השתמשנו. בתהליך מימוש שכבות הצופן של הפרוטוקול בשפת C, הסתמכנו על המימוש המצוין של הפרוטוקול כפי שהוא מופיע ב-[GitHub](https://github.com) של Wireshark.

נסיון תקיפה נאיבי

אחרי שמצאנו את מפתח התעבורה, ובאמצעותו השגנו את מפתח הרשת של הבקר, נוכל לנסות כעת לבנות את הודעת ה-ZCL העוינת ש"הזרקנו" קודם לכן, ולנסות לשדר אותה מהאוויר. אחרי מספר סבבים כושלים בהם לא הצלחנו להגיע אל ה-breakpoint ששמנו בפונקציה הפגיעה, היו לנו חדשות טובות וחדשות רעות:

- **חדשות טובות:** אין בדיקות מכונת מצבים במודול ה-ZCL. הבקר יטפל בשמחה בכל הודעת תשובה שיקבל, גם אם הוא כלל לא שלח שאלה עליה אנו עונים.
- **חדשות רעות:** מכונת המצבים הלא ידועה שנמצאת ליד הפונקציה הפגיעה מנתבת את ההודעה שלנו לפונקציה אחרת.

הבדיקה שמפריעה לנו מוצגת באיור הבא:



בתחילה, זה נראה כאילו צריך שיהיו מינימום 2 או 3 נורות ברשת, אבל גם זה לא עבד. אחרי צלילה מעמיקה לקוד למדנו שהפונקציה בודקת האם הנורה ששלחה את ההודעה נמצאת כרגע במהלכו של תהליך Commissioning.

מסקנה #1: הפונקציה הפגיעה נגישה לתוקף רק במהלך צירוף נורה חדשה לרשת. חברים לגיטימיים ברשת ה-ZigBee אינם יכולים לנצל את החולשה שברצוננו להשמיש.

Classic Commissioning

”Classic Commissioning” הוא שמו של התהליך בו נורה חדשה מצורפת לרשת באמצעות השימוש באפליקציה הסטנדרטית של היצרן לפלאפון הנייד. האפליקציה בה השתמשנו היא [Philips Hue](#) כפי שהיא מופיעה בחנות האפליקציות של אנדרואיד.

באופן מפתיע, למרות שישנם מספר רב של מסמכים המתארים את ההודעות השונות בפרוטוקול ה-ZigBee, לא הצלחנו למצוא מסמך אחד שמתעד כהלכה את ההודעות השונות הדרושות עבור תהליך צירוף נורה חדשה לרשת. על כן, בחרנו לנקוט בגישה שונה וקיוונו שבסופו של דבר היא תעבוד:

1. הקלטנו כמה שיותר הודעות מתהליך הצירוף של נורה רגילה לרשת.
2. בצורה הדרגתית מימשנו את ההודעות השונות, שלחנו אותן לבקר, וחיכינו לקבל ממנו תשובה. בכל שלב בו הצלחנו, הבקר היה עונה חזרה עם שאלה חדשה, ואז היינו הולכים לממש את הודעת המענה וחוזר חלילה.

מסקנה #2: הבקר לא יקבל נורות חדשות לרשת אלא אם המשתמש לחץ במפורש על הכפתור באפליקציה שמורה לבקר לחפש נורות חדשות. כל לחיצה תזכה אותנו בתקופת חסד קצרה של 1-2 דקות.

בחירה מימושית זו היא החלטה מצויינת מנקודת מוצא הגנתית, היות והיא מפחיתה בצורה משמעותית את משטח התקיפה על הבקר. המשמעות עבור מתאר התקיפה שלנו היא שנצטרך בצורה כלשהיא לגרום למשתמש ללחוץ על הכפתור באפליקציה בנייד.

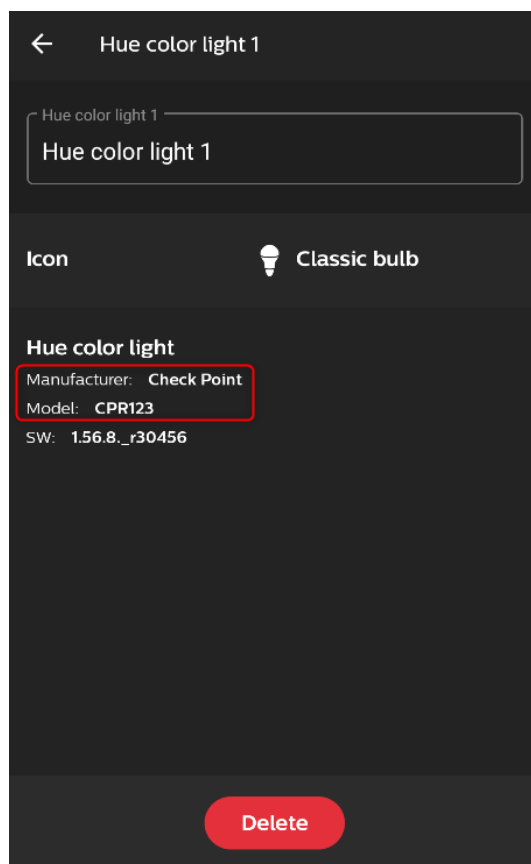
המשמעות הנוספת של ההגבלה הנ”ל היא שבתחילת כל ניסוי נצטרך ללחוץ בעצמנו על הכפתור באפליקציה, ורק לאחר מכן נוכל להריץ את התוכנית מלוח הפיתוח שלנו. זה היה המקרה כשניסינו ללמוד על ההודעות השונות בתהליך צירוף הנורה לרשת, וזה גם היה המקרה כאשר בדקנו את קוד התקיפה שלנו. לא בדיוק תהליך חלק, אבל הוא עבד בסופו של דבר.

כפי שהבטחנו קודם לכן, להלן [לינק להקלטה \(.pcap\) מלאה](#) של צירוף לוח הפיתוח שלנו אל הרשת של הבקר אותו ניסינו לתקוף, עד לשלב בו האפליקציה הציגה חיווי למשתמש כי נמצאה נורה חדשה. ההודעות אינן כוללות את שדה ה-FCS אותו הזכרנו קודם לכן, וההקלטה אינה כוללת את הודעות ה-Ack של שכבת ה-IEEE 802.15.4 היות והן נשלחות בצורה אוטומאטית כמעט לאחר כל הודעה ברשת.

הערות מימוש: יש מגוון אילוצי זמנים קשיחים למדי בפרוטוקול ה-ZigBee בכלל, ובבקר הנורות בפרט, אשר גורמים לשיחה כולה להיות רעועה להחריד במידה ולא מתזמנים את ההודעות כהלכה. בין היתר, נדרשנו לשלוח Ack על הודעות נכנסות תוך מיקרו-שניות בודדות, הגבלה שאינה ישימה בבואנו לממש את כל הפרוטוקול בתוכנה בלבד. על כן, הגדרנו את לוח הפיתוח שלנו כך שרכיב ה-MAC יענה במקומו בצורה אוטומאטית על ההודעות הנכנסות. שינוי זה שיפר משמעותית את יציבות התקיפה שלנו, אבל גרם

לחסרון אחר: אנחנו לא יכולים יותר להקליט את כל ההודעות מהאוויר, משום שלא נוכל יותר להשתמש ב-promiscuous mode.

האיור הבא מציג את ה"נורה" שלנו, כפי שהיא מוצגת למשתמש באפליקציה לאחר שסיימנו לצרף אותה בהצלחה לרשת:



כפי שאתם יכולים לראות בבירור, ישנם מספר שדות נשלטים אשר מועברים בתהליך צירוף הנורה לרשת. אנו בחרנו למתג את הנורה החדשה שלנו כנורה החדשה ביותר של Check Point Research מדגם CPR123.

את תהליך צירוף הנורה לרשת ניתן לחלק בגסות ל-4 שלבים עיקריים:

1. **שיוך:** הנורה החדשה תציג את עצמה, ותשוייך לה כתובת רשת מקוצרת.
2. **קבלה:** הנורה החדשה תקבל את מפתח הרשת ותכריז על עצמה באמצעות הודעת Device Announce.
3. **בירוקרטיה:** הבקר יתשאל את הנורה אודות מספר רב של מזהים שונים.
4. **ZCL:** הבקר ישלח מספר רב של שאלות ZCL במטרה ללמוד על המאפיינים השונים של הנורה. רק במהלך השלב האחרון, שלב ה-ZCL, אנחנו יכולים להתחיל לשלוח הודעות ZCL עוינות, בניסיון לנצל את החולשה שמצאנו קודם לכן. בעוד שאנחנו יכולים לשלוח הודעות מענה לא חוקיות גם ללא שאלות עליהן אנו כביכול עונים, אנחנו עדיין מוגבלים לביצוע התקיפה רק במהלך שלב ה-ZCL.

תקיפת ה-Heap

בחרנו להתמודד עם הבעיות שלנו אחת-אחת. תחילה ננסה לנצל כהלכה את דריסת הזכרון ב-heap עד לכדי קפיצה להרצת קוד מכתובת זכרון נבחרת כלשהי, ורק לאחר מכן ננסה להבין לאן כדי לקפוץ. בדיעבד, גישה זו התבררה כשגויה, היות ומבנה ה-heap השתנה בצורה דרסטית כתלות בהודעות ששלחנו על מנת למקם את הקוד הזדוני שלנו בזכרון.

הדבר הראשון שמומלץ לעשות כשמנצלים חולשת זכרון ב-heap הוא לבדוק באיזה מימוש heap מדובר. במקרה שלנו, הבקר עושה שימוש ב-[uClibc](#), או "מיקרו-libC", ספרייה נפוצה מאוד בעולם ה-Embedded. הגרסה המדויקת של הספרייה מצויינת בבירור בשמה של הספרייה: "libuClibc-1.0.14", ומתוך מספר מימושי heap בהם תומכת הספרייה, ניתן היה לאתר בקלות את השימוש במימוש הסטנדרטי (malloc- standard), אשר מבוסס על [dlmalloc](#).

בהיותה מימוש רזה של הספרייה הסטנדרטית, עם קהל יעד ברור של מוצרים המתמודדים עם אילוצי זכרון וכוח עיבוד די קשוחים, המימוש של הספרייה הוא די פשוט:

- Fast Bins שומרים חוצצים קטנים ברשימה מקושרת חד-כיוונית (singly-linked list).
- רשימה מקושרת דו-כיוונית (doubly-linked list) משמשת לאפסון יתר החוצצים.
- פעם בכמה זמן, כתגובה למגוון אירועים, מתבצע איחוד של החוצצים על מנת "לסדר" את ה-heap. בהיותה מימוש סטנדרטי מבוסס dlmalloc, מידע הניהול של ה-heap עבור כל הקצאה הוא המידע הבא:

```
struct malloc_chunk {
    size_t      prev_size; /* Size of previous chunk (if free). */
    size_t      size;      /* Size in bytes, including overhead. */

    struct malloc_chunk* fd; /* double links -- used only if free. */
    struct malloc_chunk* bk;

};
```

הערות:

- כאשר חוצץ מוקצה ובשימוש, שני השדות הראשונים נשמרים לפני המידע של המשתמש.
- כאשר החוצץ משוחרר וממוקם ב-Fast Bin, השדה השלישי גם הוא בשימוש, והוא מצביע על האיבר הבא ברשימת החוצצים המשוחררים. שדה זה ימוקם ב-4 הבתים הראשונים של חוצץ המידע שהיה נגיש למשתמש.
- כאשר החוצץ משוחרר ואינו ממוקם ב-Fast Bin, השדות השלישי והרביעי יהיו בשימוש כחלק מהרשימה הדו-כיוונית בה שמור החוצץ. שדות אלו ימוקמו ב-8 הבתים הראשונים של חוצץ המידע של המשתמש.

בחירת המטרה בתוך ה-heap

רשימות דו-כיווניות לרוב מציעות פרימיטיב השמשה חזק למדי, היות ובתהליך הוצאת איבר משובש מהרשימה ניתן לגרום לפעולת כתיבה נשלטת (Write-What-Where). אולם, אנחנו לא בשנות ה-2000 המוקדמות, ופרימיטיב תקיפה זה לא עובד יותר על מימושי heap נפוצים. במקום, המפתחים מימשו מנגנון הגנה הידוע בכינוי "Safe Unlinking" (או: ניתוק בטוח), אשר מוודא את שני המצביעים לפני השימוש בהם:

```
/* Take a chunk off a bin list */
#define unlink(P, BK, FD) {
    FD = P->fd;
    BK = P->bk;
    if (FD->bk != P || BK->fd != P)
        abort();
    FD->bk = BK;
    BK->fd = FD;
}
```

בשל קיומו של מנגנון הגנה זה, בחרנו לתקוף במקום את הרשימות החד-כיווניות שבמימוש ה-Fast Bins. את הרשימות הללו לא ניתן לבדוק לתקינות באותה הצורה בשל העובדה שלא ניתן ללכת קדימה ואחורה ולבדוק כי נשארו באותו המקום.

מבנה ה-Fast Bins הוא למעשה מערך של "סלים" בגדלים שונים, אשר מכילים רשימה חד-כיוונית של חוצצים עד לגודל נתון. הסל המינימאלי יכול חוצצים בגודל של עד 0x10 בתים, הסל הבא יכול חוצצים בגדלים של 0x11-0x18, וכו'. במהלך קריאת קוד המימוש של ה-heap נתקלנו בבאג מימושי מועיל הקורה בזמן הקריאה לפונקציה free():

```
/* offset 2 to use otherwise unindexable first 2 bins */
#define fastbin_index(sz) (((((unsigned int)(sz)) >> 3) - 2)
```

הקוד מסתמך על ההנחה כי גודל החוצץ המינימאלי **צריך להיות** 0x10, ועל כן המאקרו fastbin_index() מחלק את הגודל ב-8, מפחית 2, ומשתמש בתוצאה בתור אינדקס למערך הסלים. במידה ונוכל לשבש את המידע הניהולי שליד חוצץ נתון, הרי שבתהליך השחרור שלו נגרום לאינדקס להיות אחד מהערכים הלא תקינים: -1, -2:

```
struct malloc_state {
    /* The maximum chunk size to be eligible for fastbin */
    size_t max_fast; /* low 2 bits used as flags */

    /* Fastbins */
    mfastbinptr fastbins[NFASTBINS];

    /* Base of the topmost chunk -- not otherwise kept in a bin */
    mchunkptr top;
```

באמצעות השימוש בערך הלא חוקי "-1", נוכל לגרום לשמירת החוצץ על גבי השדה "max_fast" אשר אחראי על הגודל בר הקינפוג של הסל המקסימאלי. שמירת מצביע במקום הערך המספרי הנ"ל כנראה תוביל לתוהו ובוהו במרחב הזכרון של התהליך הנתקף, אבל מה בנוגע לערך הלא חוקי "-2"?

באמצעות דיבאגר, ראינו כי לפני המידע הניהולי הגלובאלי של ה-heap, לא שמור דבר. כלומר, שמירת מצביע במערך fastbins[-2] לא תהרוס שום דבר חשוב. בנוסף, malloc() לא יודע שהרגע המצאנו תא חדש במערך, ולכן החוצצים שישמרו לרשימה שבתא זה ישארו בו לעד. לכל צורך מעשי, הרגע יצרנו את סל ה-dev/null, דבר הנותן לנו פרימיטיב של איבוד זכרון מה-heap, פרימיטיב שיעזור לנו לעצב את ה-heap לכדי הצורה הרצויה עבור התקיפה שלנו.

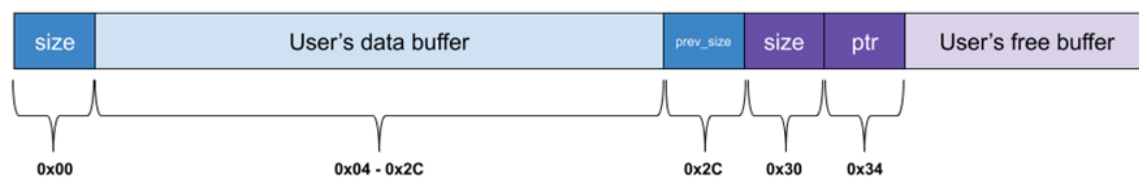
תוכנית פעולה לדריסה

החולשה שלנו מאפשרת לנו דריסת זכרון רציפה ונשלטת על ה-heap, מעבר לחוצץ בגודל 0x2B ועד למקסימום של 70 בתים. הודות ליישור טבעי של ה-heap, ככל הנראה נקבל חוצץ בגודל 0x30 בתים (אנו עלולים לקבל חוצץ גדול אף יותר במידה ויגמרו החוצצים הפנויים בגודל 0x30). בנוסף, יש פרט מימושי מעצבן במימוש הספציפי הזה של ה-heap:

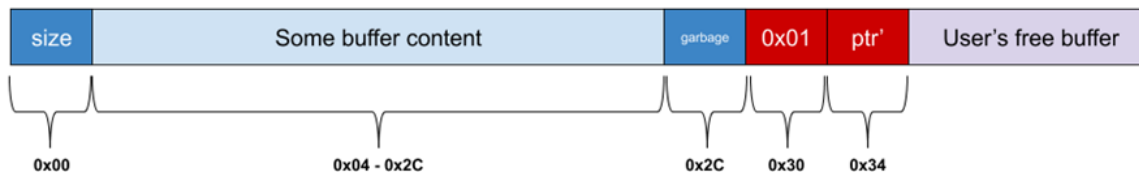
- **בתים 0x00-0x04:** שדה הגודל מתוך ה-malloc_chunk.
- **בתים 0x04-0x2C:** חוצץ המידע שנגיש למשתמש. **קצר ב-4 בתים** ממה שאנחנו באמת צריכים.
- **בתים 0x2C-0x30:** ארבעת הבתים ה"חסרים", אשר מתפקדים גם בתור שדה ה-prev_size של ה-malloc_chunk שממוקם אחריו בזכרון.

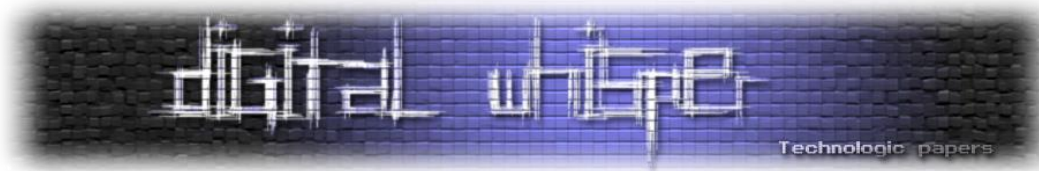
בחירה מימושית "מעניינת" זו ככל הנראה חסכה למישהו איזה ארבעה בתים עבור כל הקצאה, אבל היא בוודאי לא גרמה לקוד להיות פשוט יותר להבנה או לבדיקה.

עכשיו כשאנחנו מבינים לעומק את כל פרטי המימוש של ה-heap, התוכנית שלנו תהיה לדרוס זכרון ולשבש שדות זכרון של חוצץ ששמור אחריו ובתקווה נמצא כעת ב-Fast Bin. האירור הבא מציג את החוצץ כפי שיראה לפני שנתחיל את התקיפה:



וכעת האירור יציג את אותם שני החוצצים **לאחר** הדריסה שלנו:





באמצעות הדריסה שלנו, אנו מתכננים לשנות את השדה "size" של החוצץ שאחריו לערך 1. היות והגודל תמיד אמור להתחלק ב-4, שני הביטים התחתונים משמשים כדגלים, ולכן הערך 1 מייצג את הגודל 0 והדגל "prev_inuse" יהיה דלוק.

בנוסף, ננסה לשנות את המצביע של הרשימה החד-כיוונית בה אנו מקווים שחבר החוצץ שאחריו בזכרון. באמצעות שינוי מצביע זה לערך לשליטתנו, נוכל לעבוד על ה-heap שכעת ישנו חוצץ "משוחרר" ששמור באזור כרצוננו בזכרון. מאוחר יותר, באמצעות קריאה לרצף הקצאות זכרון מהגודל המתאים לסל הרלבנטי, נוכל לקבל פרימיטיב של "הקצאה כרצוננו" (Malloc-Where), באמצעותו אנו מתכננים להגיע לכדי הרצת קוד.

להלן תיאור קצר של ארבעת התרחישים האפשריים בהם אנו עלולים להיתקל בזמן הדריסה:

- החוצץ אחריו משוחרר ונמצא ב-Fast Bin. **הצלחה**, שיבשנו את המצביע של הרשימה ונוכל להרוויח את פרימיטיב התקיפה הנכסף.
 - החוצץ אחריו בשימוש, והוא ישוחרר מתישהו. **סבבה**, פעולת השחרור תמקם אותו ברשימת ה-"/dev/null", ולאחר מכן לא נראה אותו יותר.
 - החוצץ אחריו בשימוש, ולא ישוחרר אף פעם. **לא נורא**, בואו נקווה ששיבוש ארבעת הבתים הראשונים בחוצץ יעבור בשקט ולא יגרום נזק.
 - החוצץ אחריו משוחרר והרגע שיבשנו מצביע של רשימה דו כיוונית. **כישלון**, בואו נקווה שאף אחד לא ישים לב, כי שחרור של החוצץ מה-heap יוביל להקרסה של התוכנית.
- למעשה, אנחנו יכולים ממש להפסיד רק באחד מתוך ארבעה מקרים, וביתר המקרים או שאנחנו מצליחים, או שאנחנו יחסית מתקדמים בכיוון הנכון. בואו נקווה שהסיכויים יהיו תמיד לטובתנו, ובכל זאת ננסה לדרוס כמה שפחות פעמים בדרך לתקיפה מוצלחת.

הערת צד: אחרי שסיימתי את המחקר, חשבתי עוד על הדרך שבה התקיפה התמקדה ברשימות החד-כיווניות ב-heap, שיטה שאני משתמש בה לתקיפות מבוססות heap כבר קרוב לעשור. אחרי תהליך חשיבה בנושא, פיתחתי את מנגנון ההגנה Safe Linking להגנה על רשימות חד-כיווניות, בדגש על ה-heap. את הפתרון עצמו הצלחתי לשלב בהצלחה ב-uClibc-NG ואפילו ב-Glibc, מימוש הספרייה הסטנדרטית המשמש את הפצות הלינוקס השונות. פרטים נוספים על מנגנון ההגנה Safe Linking תוכלו למצוא ב**בלוג שכתבתי בנושא**.

עיצוב מבנה ה-heap

הרכיב המשפיע ביותר על עיצוב ה-heap לכדי הצורה שהצגנו קודם לכן, הוא העובדה כי המעבד הראשי עליו רץ התהליך הנתקף הוא מעבד די חלש. אם נשלח מספיק הודעות ומספיק מהר, אז נרעיב חלק מהחוסים במערכת בגלל שהטיפול בהודעות שלנו ימנע מהם זמן ריצה. מבחינתנו, זה אומר שלכל צורך מעשי החוסים במסלול הטיפול בהודעות שלנו הם היחידים שבאמת יתוזמנו לרוץ, מה שמצמצם משמעותית את הרעש איתו נצטרך להתמודד.

עכשיו כשלמדנו את זה, וכשאנחנו יודעים שאנחנו מקצים ומשחררים חוצצים בגודל של $0x30$ בתיים, התוכנית נהיית יחסית פשוטה:

1. נשלח מספר הודעות ZCL שיובילו להקצאות זכרון בגדלים של $0x28$ ו- $0x30$.
 2. נשלח (ממש) מעט הודעות ZCL עוינות, ונכוון לשבש מצביע Fast Bin של גודל $0x30$ כך שיצביע על טבלת המצביעים הגלובאלית (Global Offset Table - GOT).
 3. נשלח רצף נוסף של הודעות בגודל $0x30$ בתקווה להטריג את הפרימיטיב מסוג הקצאה נשלטת.
- השלב הראשון הוא האיטי ביותר, היות ואנחנו רוצים שהחוצצים שהוא מקצה יתחילו להשתחרר לפני שנתחיל לדרוס זכרון. שוב, אנחנו מכוונים לדרוס ישירות אל תוך חוצץ משוחרר.

בשלב השני, אנחנו מכוונים לדרוס את מצביע ה-Fast Bin כך שיצביע כעת על המצביע לפונקציה `free()` כפי שהיא שמורה בטבלת המצביעים הגלובאלית. בצורה זו, השלב השלישי ישלח הודעות שאחת מהן תשמר על גבי ה-GOT משום שה-heap חשב בטעות שמדובר בחוצץ זכרון זמין לשימוש. בכך למעשה הפכנו פרימיטיב של הקצאה נשלטת לכדי דריסה מלאה באמצעות הודעה מהרשת ישר על טבלה מלאה של מצביעים לפונקציות. פרימיטיב חזק לכל הדעות.

לאחר מכן, הטריגר להרצת הקוד הוא ישיר - קריאה לפונקציה `free()` עם אחת מההודעות שלנו תוביל לקפיצה אל כתובת לבחירתנו.



אכסון קוד התקיפה (shellcode)

היות ומרבית חוצצי הזכרון מוקצים בצורה דינאמית על ידי ה-heap, אשר נטען בכל פעם לכתובת אקראית על ידי מערכת ההפעלה, המטלה של מציאת מיקום קבוע לשמירת תוכן נשלט שישמש אותן כקוד להרצה היא מטלה קשה למדי. בנוסף, המודם מעביר למעבד הראשי הודעות בייצוג טקסטואלי, כך שאין לנו איזה חוצץ גלובאלי בו נוכל לשמור מידע בינארי נוח ונשלט.

לבסוף, הגענו למסקנה שלא נוכל להרשות לעצמנו להיות בררניים. אנחנו ניאלץ להשתמש במערך הגלובאלי היחידי שזמין לרשותנו: המערך בו הבקר שומר את הודעות השכנים (LQI) הנכנסות. למערך זה יש יתרונות וחסרונות:

יתרונות:

- המערך הגלובאלי ממוקם במקום קבוע וידוע מראש.
- המערך, כמו יתר המשתנים הגלובאליים הכתיבים, הוא בעל הרשאות זכרון של RWX - בר הרצה.
- המערך יחסית גדול - יכול להכיל עד 0x41 (65) רשומות של 0x10 (16) בתים.

חסרונות:

- אנחנו לא שולטים בצורה מלאה בתוכן של כל הרשומה - אין לנו באמת 0x10 בתים נשלטים רציפים. מאוחר יותר גילינו גם שאנחנו אפילו לא יכולים להשתמש בכל הקיבולת של 0x41 רשומות, אבל כשאין הרבה אופציות, נאלצים להסתפק במה שיש.

האילוצים על כל רשומת שכנים הם:

- **בתים 0x00-0x08:** כתובת מורחבת - הבתים נשלטים לחלוטין.
- **בתים 0x09-0x0A:** כתובת מקוצרת - הבתים נשלטים לחלוטין.
- **בתים 0x0A-0x10:** שונות - מחוץ לשליטתנו.

וכדי להוסיף הגבלה נוספת, אנחנו לא באמת יכולים לשלוט ב-10 הבתים הרציפים שבתחילת כל רשומה, היות והבקר בודק שהרשומות השונות **יחודיות**. כלומר, כל כתובת מורחבת צריכה להיות יחודית, לא כזה נורא היות ומדובר ב-8 בתים והסיכוי שיתנגשו הוא אפסי. אבל, גם כל כתובת מקוצרת צריכה להיות יחודית, וזה סיפור שונה לגמרי. נצטרך להיות די יצירתיים בכדי לעקוף את ההגבלה הזו על מנת להשתמש בכל הבתים שנוכל לסחוט מהמצב הבעייתי הזה.

הדרך התקינה לשלוח את רשומות השכנים הללו אל הבקר היא באמצעות הודעות מענה לשאלות LQI. אבל, הפעם המודם והמעבד הראשי החליטו לנהל מעקב מסודר אחרי מכונת המצבים של ההודעות, ואנחנו יכולים לשלוח תשובות אך ורק עבור הודעות שנשלחו קודם לכן מצד הבקר. לצערנו, הבקר מתחיל לשלוח הודעות שכאלו רק **לאחר** שמסתיים שלב ה-ZCL. כלומר, אנחנו יכולים לסדר את קוד התקיפה בזכרון רק **אחרי** שנסגר חלון ההזדמנויות שלנו לתקיפה...

בשלב הזה עברנו לבדוק את תוכן מערך הזכרון הגלובאלי, וראינו שהוא כולל בתוכו גם את כתובת הרשת שלנו, אפילו שטרם ענינו על הודעות LQI כלשהן. בדיקה נוספת העלתה שגם הודעות Device Announce, אשר נשלחות כחלק מצירוף ה"נורה" שלנו לרשת, נשמרות כרשומות בודדות באותו המערך. למעשה, מדובר במעין "ספר טלפונים" ולא במערך "שכנים".

No.	Time	Source	Destination	Protocol	Length	Info
62	0.020812	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4000, Ext Addr: Private_11:11:11:11:11
69	0.037196	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4001, Ext Addr: 22:22:22:22:22:22:22:22
73	0.053580	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4002, Ext Addr: 33:33:33:33:33:33:33:33
78	0.069964	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4003, Ext Addr: 44:44:44:44:44:44:44:44
81	0.086348	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4004, Ext Addr: 55:55:55:55:55:55:55:55
88	0.102732	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4005, Ext Addr: 66:66:66:66:66:66:66:66
94	0.119116	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4006, Ext Addr: 77:77:77:77:77:77:77:77
97	0.135500	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4007, Ext Addr: 88:88:88:88:88:88:88:88
98	0.151884	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4008, Ext Addr: 99:99:99:99:99:99:99:99
105	0.168268	0x013c	Broadcast	ZigBee ZDP	55	Device Announcement, Nwk Addr: 0x4009, Ext Addr: aa:aa:aa:aa:aa:aa:aa:aa

בשלב הזה הדברים התחילו להסתבך. עבור כל כתובת חדשה שהבקר לומד ומאחסן במערך, הוא ישלח הודעה תואמת מסוג Route Request בניסיון להבין איך מגיעים אל אותה הנורה. שליחת כמות גדולה (מעל 20) של הודעות כאלו תערער לחלוטין את יציבות הבקר, שגם כך לא היה הכי יציב עוד כשהתחלנו.

הפתרון שהעלנו לשלב זה הוא לעשות שימוש במספר נורות עבור תהליך התקיפה שלנו:

1. נורה "לגיטימית" שתופיע באפליקציה של המשתמש, ומאוחר יותר תנצל דלת אחורית אותה אנחנו מתכננים להתקין במהלך התקיפה.
2. נורה פיקטיבית שתפרסם המון "נורות" ובפועל תמקם את קוד התקיפה שלנו במערך בזכרון, כפי שניתן לראות באיור שלמעלה.
3. נורה פיקטיבית נוספת שתגיע לשלב ה-ZCL ותנצל את החולשה בכדי להריץ את הקוד שכעת נמצא בזכרון.

היות ורק הנורה הראשונה תסיים בהצלחה את כל תהליך צירוף הנורה לרשת, למשתמש לא תהיה שום אינדיקציה כי הבקר ראה נורות דמה נוספות במהלך התקיפה שלנו.

תכנון קוד תקיפה בעולם מושלם

אם נעשה שימוש בפקודות Mips16, מרביתן של הפקודות יעלו לנו 2 בתים כל אחת, ורק פקודות מורכבות יותר יעלו לנו 4 בתים לפקודה. בתיאוריה, נוכל להשתמש ב-8 הבתים הראשונים של כל רשומה כדי להריץ מעט קוד, ולאחר מכן נשתמש ב-2 הבתים הנוספים על מנת לקפוץ אל הרשומה הבאה במערך. בדיוק בשלב הזה מגבלת הייחודיות תחסום אותנו. ברוב הפעמים נקפוץ 6 בתים קדימה (אל הרשומה הבאה), וזה אומר שפקודת הקפיצה תהיה זהה בכל פעם ותפר את מגבלת הייחודיות. נוכל אולי לעקוף חלקית את ההגבלה באמצעות שימוש במגוון סוגי קפיצות במידה ונוכל לבצע קפיצות מותנות.

התוכנית הכוללת עבור קוד התקיפה שלנו היא לערוך את הקובץ הבינארי המקורי של תהליך ה-ipbridge על מנת להתקין בו דלת אחורית. סביר להניח שהשינויים שעשינו ב-heap יגרמו לתהליך להיות לא יציב,

וזאת בלשון המעטה. אם נשנה את קובץ ההרצה עצמו, אזי אחרי שנקרוס מערכת ההפעלה תריץ את הקובץ המתוקן שכעת יכיל את הדלת האחורית שהתקנו בו.

בעוד שהתוכנית נראתה טוב על הנייר, גם הנתיב אל קובץ ההרצה, וגם קוד הדלת האחורית, היו גדולים מדי ליצוג ב-10 בתים רציפים. הרעיון שחשבנו עליו כדי להתמודד עם האילוץ הנ"ל הוא לולאת קידוד פשוטה:

1. הרשומות הראשונות ירצו בלולאה שתעתיק מידע מיתר הרשומות על מנת לסדרן בזכרון כחוצץ רציף.

2. יתר הרשומות ישמשו בתור קוד התקיפה עצמו אותו תכננו מלכתחילה. בעוד שקוד התקיפה עבד טוב בסביבת הבדיקות, הוא נתקל בקשיים כשניסינו אותו על המטרה האמיתית.

ראשית, מדובר בקוד תקיפה די יקר: הוא עולה לנו 0x19 רשומות כאשר במקור תכננו לשלוח רק 0x10 רשומות כאשר ניסינו את התקיפה על ה-heap עם קוד תקיפה צעצוע. השינוי המזערי הזה של הוספת 9 רשומות היה הקש ששבר את גב הגמל: הבקר נהיה רעוע מדי ולא הצלחנו להגיע לשלב בו אנחנו מצרפים את הנורה השלישית שלנו לרשת.

אחרי הרבה חישובים הצלחנו לדחוס את קוד התקיפה היפה והקונפיגוראבילי שלנו לכדי קוד משמעותי פחות קריא שמשתמש רק ב-0x12 רשומות. הצלחנו לעבור את מגבלת הגודל בהצלחה, והתחלנו לדבג את קוד התקיפה שלנו על המטרה האמיתית.

בשלב זה מצאנו חורים נוספים בתוכניתנו המקורית. לולאת קידוד בארכיטקטורת Mips מחייבת קריאה לפונקציה Sleep() בכדי להימנע מבעיות cache. אחרת, השינוי שעשינו לקוד לא יפעל כהלכה ל-Instruction Cache של המעבד ויפיע רק ב-Data Cache. בפועל, זה אומר שנריץ זבל מוחלט במקום המידע היפה שסידרנו כפי שהמעבד רואה אותו בבואו לקרוא מידע ולא פקודות. אבל, קריאה זו לשינה של פרק זמן קצר תוביל לכך שאחרי שהשמדנו את ה-heap של הקורבן אנחנו נשאיר לחוטים אחרים להתמודד עם הבלאגן הזה בזמן שאנחנו ישנים את שנת היופי שלנו. בפועל, זה אומר שנקרוס.

אנחנו מצד אחד לא יכולים להרשות לעצמנו להגדיל את קוד התקיפה כדי גם לישון וגם לשחזר ריצה, ומהצד השני אנחנו לא יכולים להשתמש בלולאת קידוד ללא שינה ושחזור ריצה. בנוסף, הסתבר לנו שקובץ ה-ELF שרצינו לשנות כלל אינו כתיב בזמן הריצה, כך שנאלצנו לבסוף לתכנן תקיפה אחרת.

תכנון קוד תקיפה קטן ונועז

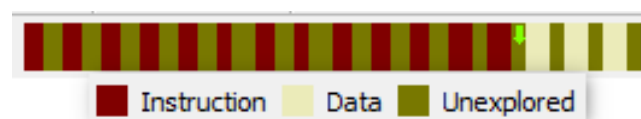
אם אנחנו גם ככה צריכים לשחזר את הריצה של התוכנית כדי שהיא לא תקרוס בזמן הקריאה לפונקציה Sleep(), אנחנו כבר יכולים לנסות לשחזר את הריצה של התוכנית ולהתקין את הדלת האחורית בזכרון התוכנית שרצה כרגע. בצורה זו לא נצטרך לכתוב לקובץ, לא נצטרך נתיב לקובץ, ואם קצת מזל זה יוריד את הצורך שבשימוש בלולאת קידוד יקרה.

אז חזרנו בחזרה אל לוח השרטוטים, ואחרי כמה ימים הצלחנו לכתוב קוד תקיפה חדש שיבצע את סט הפעולות הדרוש, על פי סדר:

1. **שחזור ריצה:** יצוב ה-heap, שחזור ה-GOT וכו'.
2. **השתקת ה-Watchdog:** עלינו לדאוג שהוא לא יכעס על כך ששלחנו הרבה הודעות והרעבנו את יתר החוטים בזמן התקיפה שלנו (לחילופין, אם אף אחד לא ישים לב שהוא כועס, זה גם טוב).
3. **התקנת דלת אחורית:** נשנה את הרשאות הזכרון באזור ספציפי להיות RWX באמצעות קריאה ל-mprotect(), ונשנה את הבתים הדרושים בכדי להתקין את הדלת האחורית במקום הרצוי.

הנקודה השנייה הייתה מעט משעשעת היות ועצם שליחת הודעות התקיפה גרמה ל-Watchdog לכעוס, גם מבלי שנרדים את החוט שלנו. כאשר היינו מסיימים את התקיפה, ה-watchdog היה מגלה שהרעבנו חלק מהחוטים והיה סוגר את התוכנית יחד עם הודעת syslog נחמדה שהוא היה שולח ליצרן. מפה לשם, אחרי שדיווחנו ליצרן על החולשה, גם ניצלנו את ההזדמנות בכדי להתנצל בפניו על זה ששלחנו לו עשרות הודעות syslog וגרמנו לו לחשוב שיש משהו לא בסדר באחד מהמוצרים שלו...

אחרי תהליך ארוך, ולא מעט בדיקות, היה לנו קוד תקיפה עובד בגודל של 0x10 רשומות. האיור הבא מציג את הזכרון של קוד התקיפה כפי שהוא מוצג ב-IDA:



כפי שניתן לראות, הרשומות הראשונות מאחסנות את הקוד אותו נריץ, ו-3 הרשומות האחרונות מכילות קונפיגורציה הכוללת בין היתר את קוד הדלת האחורית אותה נרצה להתקין. כל רשומה תריץ מעט פקודות אסמבלי, ולאחר מכן תקפוץ אל הרשומה הבא בתור. כך בשרשרת, עד שנסיים לבצע את כל המטלות ונשחזר חזרה את ריצת התוכנית כאילו דבר לא קרה.

דלת אחורית

אנחנו לא הולכים לצלול ליותר מדי פרטים טכניים בנוגע לדלת האחורית שהתקנו, היות ואנחנו לא מתכוונים לשחרר תקיפה מלאה לציבור. אבל, אנחנו כן יכולים לחשוף שהדלת האחורית העניקה לנו פרימיטיב תקיפה מסוג Write-What-Where באמצעות שליחת הודעות מיוחדות אל הבקר. את ההודעות הללו תשלח הנורה ה"לגיטימית" שהצטרפה בהצלחה לרשת הנורות. פרימיטיב תקיפה יציב זה ישמש אותנו כדי לכתוב את קוד הטעינה של [Scout](#) למערת זכרון בהרשאות RWX, ולאחר מכן ננצל את העובדה כי הקוד עדיין כתיב (בשל התקיפה שזה אך סיימנו) על מנת להפנות את הקוד אל קוד התקיפה החדש שכתבנו.

קוד הטעינה של Scout יתחבר חזרה הביתה מעל TCP ויקבל תהליך הרצה אותו הוא יטיל ויריץ על הבקר, תחת ההרשאות הגבוהות בהן רץ התהליך הנתקף. באיור הבא ניתן לראות כי הטלנו את הקובץ /tmp/exploit כתהליך שיריץ את השלב הבא של התקיפה:

```
2227 root      61968 S      /usr/sbin/ipbridge -p /home/ipbridge/var -z /dev/ttyZigbee
2278 root           0 SW      [kworker/u2:1]
2287 root      176 S      /tmp/exploit
```

באמצעות קורבן ה-Mips אותו תקפנו במחקר זה, הצלחנו להרחיב בהצלחה את התמיכה של Scout גם לארכיטקטורת Mips. בדיוק כפי שרצינו כשיצאנו לדרך בתחילת המחקר.

תיאור התקיפה המלאה

במתאר התקיפה שתיארנו, אנחנו רוצים להשתלט על בקר הנורות דרך הרדיו, ולהשתמש באחיזה בו בכדי לתקוף את רשת המחשבים הביתית / ארגונית אליה הוא מחובר. אבל קודם לכן, החולשה שלנו מחייבת אותנו לגרום למשתמש לחפש אחר נורות חדשות בכדי לצרפן לרשת, לא בדיוק שלב טריוויאלי. באמצעות יכולות התקיפה של המחקר אותו המשכנו, להלן תרחיש התקיפה שהדגמנו בפועל:

1. נשתמש ב-Touchlink Commissioning (בו השתמשו במחקר המקורי) על מנת "לחטוף" נורה מרשת הקורבן, על מנת שנוכל לשלוט בה.
2. נשנה את צבע הנורה ואת עוצמת התאורה כדי לעצבן את המשתמש ככל שנוכל. על המשתמש להבין שמשו לא בסדר עם הנורה, אבל אסור שיחשוב שהיא נשרפה כי אז היא פשוט תזרק לפח.
3. אופציונאלית: ניתן להתקין תוכנה לשליטתנו על הנורה באמצעות יכולות המחקר הקודם על מנת לבצע את יתר השלבים מהנורה עצמה. לשם פשטות, ומכיוון ואנחנו לא מפתחים נשק, אנחנו בחרנו להשתמש בלוח הפיתוח שלנו שמכיל את אותן יכולות שידור ועיבוד כמו נורה.
4. המשתמש בסופו של דבר ישים לב שמשו לא כשורה עם הנורה שלו, והיא תופיע באפליקציה כ"לא זמינה". ולכן, הוא "יאתחל אותה מחדש".
5. הדרך היחידה לאתחל נורה מחדש היא למחוק אותה מהאפליקציה, ואז לומר לבקר לחפש אותה מחדש. **בינגו! עכשיו התקיפה שלנו באמת תתחיל.**
6. הנורה שחטפנו קודם לכן נמצאת כבר ברשת אחרת (בשליטתנו) ולכן היא לא תתגלה על ידי הבקר.
7. אנחנו נתחזה לנורה לגיטימית שהמשתמש יראה באפליקציה, כך שהמשתמש יהיה מרוצה ו"יכוון מחדש" את הגדרות התאורה של "הנורה" שמצא.
8. מאחורי הקלעים, אנחנו נייצר נורות דמה **שינצלו את החולשה בבקר** על מנת להתקין בו דלת אחורית.
9. הנורה ה"לגיטימית" שצורפה לרשת תשתמש בדלת האחורית שהתקנו על מנת להריץ פוגען כרצוננו על הבקר.
10. הפוגען שלנו יצור איתנו קשר חזרה מעל האינטרנט, ויחכה לפקודות נוספות. **בזה הרגע חדרנו לרשת המחשבים של הקורבן מעל הרדיו, באמצעות ZigBee.**

לצורך ההדגמה, בחרנו להשתמש בכלי התקיפה של ה-NSA שדלף לרשת, EternalBlue, בדיוק כפי שעשינו במחקר של [הפקס](#). התקיפה תורץ מהבקר עצמו ותתקוף מחשב ביתי מסכן שידמה את רשת המחשבים הביתית / ארגונית של הנתקף. להלן סרטון של ההדגמה:

https://youtu.be/4CWU0DA_by

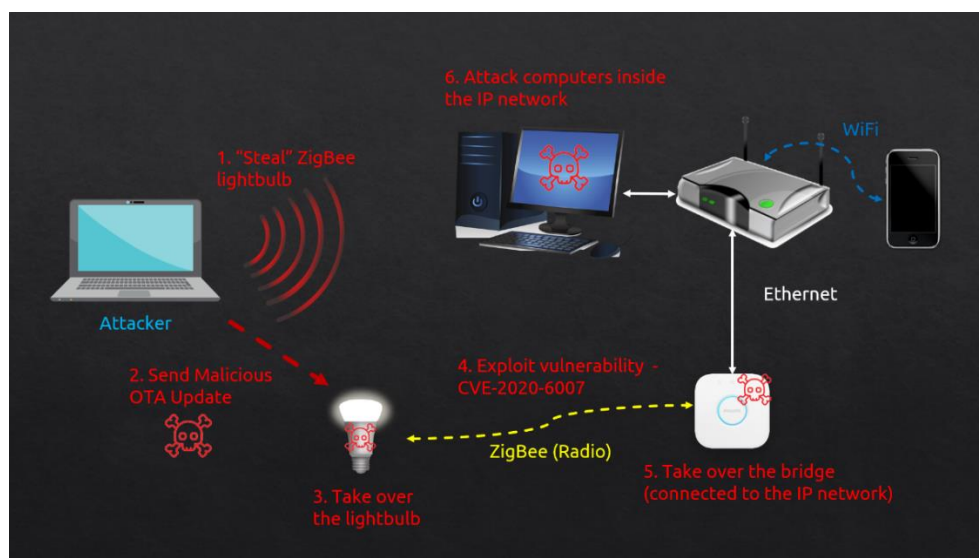
סיכום

ככל שהשנים מתקדמות, יותר ויותר מוצרים יום-יומיים נהיים "חכמים" ומאפשרים לנו לשלוט בהם מקרוב ומרחוק, לרוב באמצעות אפליקציה מהנייד. למרות שהצורך בחלק מהמוצרים החכמים הללו עדיין לא מובן לי עד הסוף, אנשים נוטים לאמץ את השימוש בהם תוך התעלמות מהסיכון הפוטנציאלי שהמוצר יכול להוסיף לרשת הביתית שלהם.

כששאלנו אנשים אם הם לא חושבים שיש סיכון בחיבור הנורות שלהם לאינטרנט, קיבלנו את התשובה הקבועה של "אל תהיו מצחיקים, זו רק נורה".

במחקר זה המשכנו עבודה קודמת שנעשתה בנוגע לתקיפת נורות חכמות והצלחנו להראות כי באמצעות השתלטות על נורה חכמה ניתן לתקוף את הבקר שמנהל אותה ומחובר במקביל גם לרשת ה-ZigBee (רדיו) וגם לרשת המחשבת הביתית. למרות אילוצי התקשורת הקשוחים למדי של ZigBee, הצלחנו להדגים תקיפה אל-חוטית שתאפשר לנו להשתלט על הבקר, ודרכו לתקוף את יתר המחשבים שברשת.

לשם המחשה, להלן איור המסכם את תהליך התקיפה שהדגמנו במהלך המחקר:



תודות מיוחדות

מחקר זה נעשה באמצעות סיוע והדרכה מצידו של אייל רונן (@eyalr0).

הפניות למידע נוסף

במקום להוסיף בכל מקום הפנייה ללינק חיצוני עם מידע נוסף, אנו מזמינים אתכם לעיין בבלוג המחקרי שפרסמנו באתר של קבוצת המחקר, והוא כולל את סרטון ההדגמה כמו גם את רשימת כל מקורות המידע בהם נעזרנו / אליהם אנחנו מפנים במהלך המאמר:

<https://research.checkpoint.com/2020/dont-be-silly-its-only-a-lightbulb/>