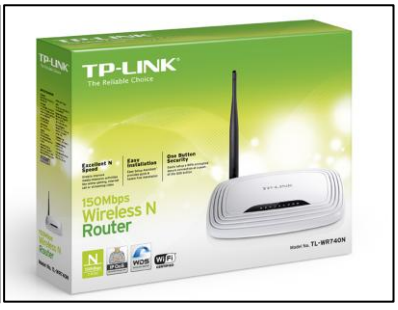
הנתב שנשכח: ניתוח מעמיק של נתב TP-Link WR740N, מחילוץ Firmware ועד גילוי חולשת אבטחה חדשה

מאת יהונתן שיאון

הקדמה

בכל בית ובכל משרד יש קופסה קטנה שמהבהבת באור ירוק בשקט, הנתב. מי ששולט בה שולט בכל מה שזורם דרכה, והרבה מתחת לרדאר, כי האנטי-וירוס רץ על המחשב שלכם, לא על הנתב. קמפיינים ברמה מדינתית כבר הפכו נתבים ביתיים לעמדות ריגול. למשל VPNFilter, שיוחס לקבוצת תקיפה של המודיעין הצבאי הרוסי והדביק יותר מחצי מיליון מכשירים, ו-Cherry Blossom של ה-CIA, שנחשף בדליפת Vault 7. מי שהסיפור הזה מסקרן אותו מוזמן לצלול: הניתוח של Cisco Talosל-VPNFilter, ומסמכי Cherry Blossom ב-WikiLeaks Vault 7, קישור למאמרים בסוף המאמר.

אבל המאמר הזה לא עוסק בהם. המאמר הוא מעשי, על נתב TP-Link TL-WR740N בבעלותי, שרכשתי דרך אתר יד 2 למטרות לימוד ומחקר בלבד. לאורך המאמר אציג כיצד ניתן לבצע הלחמות, UART, וחילוץ Firmware ישירות משבב הזיכרון. אמשיך אל פירוק וניתוח ה-Firmware, ואסיים כשהנתב כולו רץ כתוכנה על המחשב, בלי חומרה מחוברת בכלל. וכן, בדרך נשרפו כמה לוחות. הגיבור שלנו למסע הזה הוא TP-Link TL-WR740N, נתב ביתי נפוץ, זול, ומיושן מספיק כדי שנמצא בו כמעט הכל. לפני שפותחים את המכסה, מבט מהיר על הגב: דגם, גרסת חומרה, מספר סידורי.



אחת משתי דלתות הכניסה היא החומרה, ובראשה ה-UART (ראשי תיבות של Universal Asynchronous Receiver-Transmitter). זהו פרוטוקול תקשורת טורי ותיק ופשוט, שמהנדסים משתמשים בו כדי לקבל פלט Debug מהמכשיר, ולעיתים אף קונסולה אינטראקטיבית מלאה. במילים אחרות: אם נצליח להתחבר ל-UART, יש סיכוי טוב שנראה את הודעות האתחול של מערכת ההפעלה שרצה על הנתב, ואם יתמזל מזלנו גם נקבל shell.

"נתב" היא מילה מטעה. היא נשמעת כמו רכיב פסיבי שדוחף חבילות מצד לצד, אבל מבפנים נתב ביתי טיפוסי הוא מחשב לינוקס מלא לכל דבר: מעבד, זיכרון, אחסון קבוע, מערכת הפעלה ושרת Web שמנהל את הכול - פשוט בלי מסך ובלי מקלדת. אם נפתח את המכסה, נמצא כמעט בכל דגם את אותם רכיבים חוזרים:

1. ה-System on Chip: המוח. שבב אחד שדוחס לתוכו את המעבד ואת רכיבי התקשורת המרכזיים בנתב.

2. זיכרון Flash: כאן יושבת ה-Firmware, מערכת ההפעלה, הקבצים, וכל מה ששורד כיבוי. בדגמים זולים מדובר בשבב SPI-NOR קטנטן של 4 או 8 מגה-בייט בלבד.

3. זיכרון RAM: זיכרון עבודה נדיף שאליו נטענת המערכת בזמן ריצה.

4. ממשקי דיבוג: החלק שמעניין אותנו. היצרן כמעט תמיד משאיר על הלוח נקודות גישה מהפיתוח, ובראשן ממשק טורי בשם UART. הוא לא מיועד למשתמש הקצה, אבל הוא שם, פיזית, ומחכה.

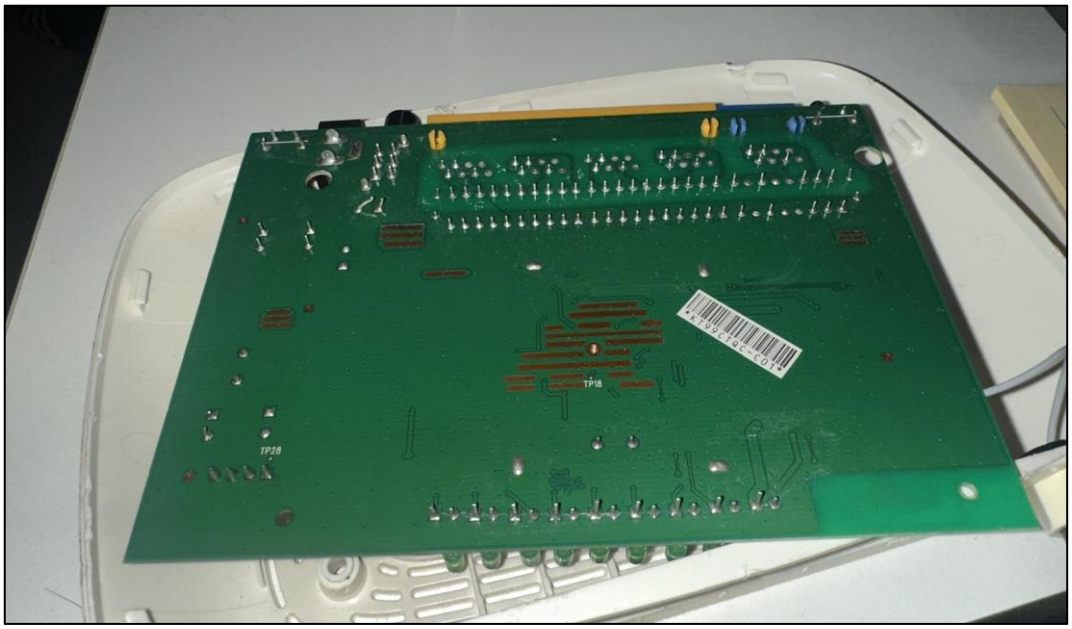
על כל הברזל הזה רץ לינוקס מוטמע (embedded Linux), לא פעם נגזרת של OpenWrt או הפצה קניינית של היצרן. ה-Firmware עצמה אינה קובץ מונוליטי אחד אלא חבילה מורכבת: bootloader (לרוב U-Boot) שמריץ את הכול באתחול, ה-kernel של לינוקס, מערכת קבצים שמכילה את כל הבינארים והסקריפטים. ברגע שנדע לפרק את החבילה הזו, נוכל לקרוא כל שורת קוד שרצה על המכשיר.

מכאן נגזרות שתי דלתות הכניסה הגדולות, ושתיהן ילוו אותנו לאורך המאמר: הדלת הראשונה, הרשת, התוקף יושב מרחוק ושולח בקשות לשרת ה-Web של הנתב, ומחפש חולשה תוכנתית. זו הדלת ה'נקייה'. הדלת השנייה, החומרה, לוקחים מברג, פותחים את הקופסה, ומתחברים פיזית אל המעבד דרך ממשק הדיבאג. וזו בדיוק נקודת ההתחלה שלנו.

ממשק ה-UART, הממשק עובד באמצעות ארבעה קווים:

- VCC – מתח (לרוב V3.3)
- GND – הארקה
- TX – שידור (הנתב משדר מידע)
- RX – קליטה (הנתב מקבל מידע)

ברוב המקרים, יצרן הנתב משאיר על גבי לוח המעגל (PCB) את ארבע נקודות החיבור הללו. לעיתים הן מופיעות כשורת פינים עם חלל, ולעיתים כרפידות חשופות. האתגר הראשון בתהליך המחקר הוא לאתר את נקודות החיבור הללו ולזהות את תפקידו של כל קו. פתחתי את הנתב ומצאתי בלוח שורת נקודות חשופות, מועמד מצוין ל-UART. אבל 'מועמד' אינו 'ודאי', וצריך לאמת. כך נראה הלוח לפני שנגעתי בו, ניתן לראות את ארבעת הפינים של ממשק ה-UART בצד שמאל למטה:



לפני שמחברים מתאם UART, יש לזהות את תפקידו של כל פין. לצורך כך השתמשתי במולטימטר וביצעתי מדידות נקודה נקודה על גבי הלוח. את קו VCC ניתן לזהות באמצעות מדידת מתח קבוע של כ־V3.3, ובכך גם לוודא שהממשק אכן פועל ברמת לוגיקה של 3.3 וולט.

את קו GND מאתרים באמצעות מצב בדיקת רציפות (Continuity), מול נקודת הארקה ידועה בלוח. לאחר זיהוי שני הקווים הללו, נותרים שני הפינים האחרונים – TX ו-RX.

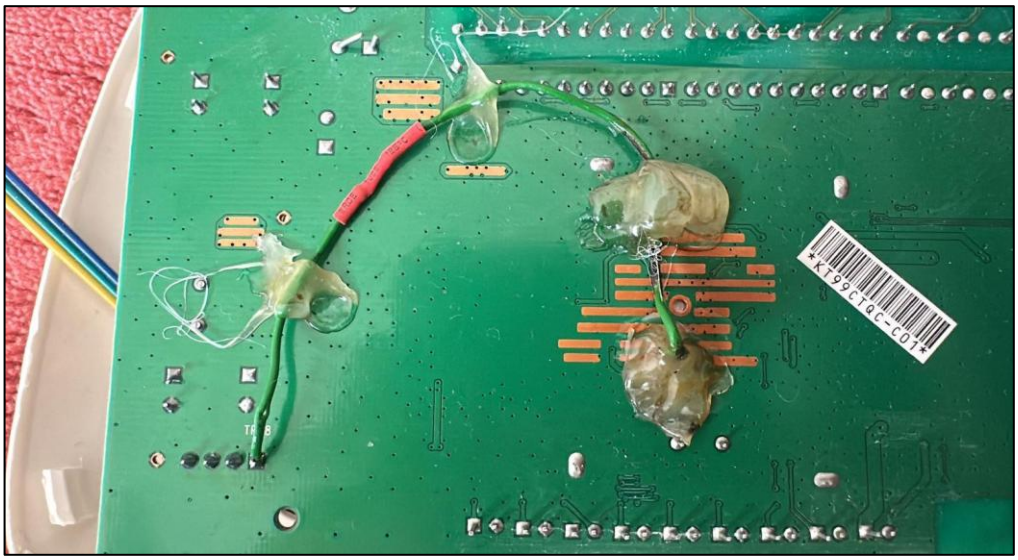
בשלב זה נתקלתי בממצא חריג, שני הקווים נמדדו במצב לוגי 0 באופן קבוע. במצב תקין, קו ה-TX

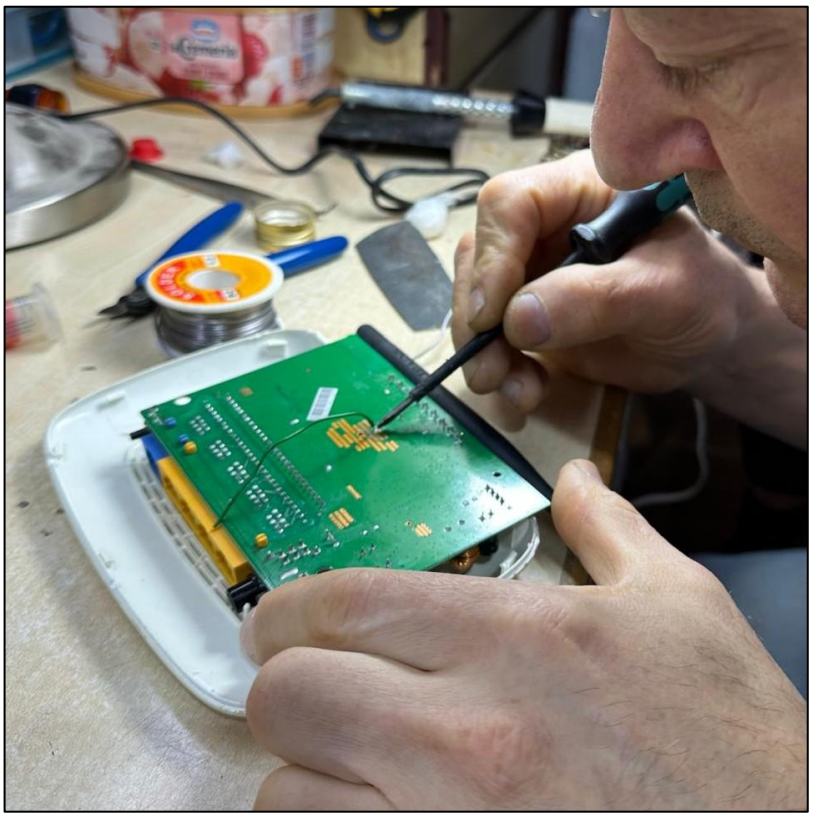
אמור לשנות את ערכו בעת הדלקת המכשיר, ולכן המדידה הייתה סימן ברור לכך שקיימת בעיה בחומרה. לאחר בדיקה מעמיקה של מסלולי המעגל, התברר כי קו ה-TX כלל אינו מחובר חשמלית לשבב הראשי, ולכן ממשק ה-UART אינו פעיל בפועל. תופעה זו לא הייתה ייחודית לנתב אחד. בשני הנתבים שחקרתי נדרש לבצע תיקון חומרה על מנת להשיב את ממשק ה-UART לפעולה. בשני המקרים קו ה-TX היה מנותק פיזית, אך אופן התיקון היה שונה בכל דגם. זוהי תופעה מוכרת בחלק ממוצרי החומרה, ובפרט במספר דגמים של TP-Link. במהלך הייצור, היצרן לעיתים משמיט רכיבים מסוימים כדי לצמצם עלויות ייצור. כתוצאה מכך, מסלולים מסוימים ובהם קו ה-TX, נותרים מנותקים חשמלית, למרות שנקודות החיבור עדיין קיימות על גבי הלוח. מבחינת המשתמש, ממשק ה-UART נראה קיים, אך למעשה אינו נגיש ללא תיקון פיזי.

בנתב TL-WR740N, הפתרון היה פשוט יחסית: יצירת גשר בדיל בין שתי נקודות בדיקה (TP18 ו-TP28). הפתרון אינו מבוסס על ניחוש, אלא מתועד בתיעוד של Openwrt. לאחר חיבור שתי הנקודות, המעגל נסגר וקו ה-TX חזר לפעול כראוי.

בדגם TL-WR841N נדרש תיקון מעט שונה, במקרה זה היה צורך להלחים שני רכיבים קטנים אשר משלימים את מסלול האות ומחזירים את קו ה-TX לפעילות. תיעוד מלא של התהליך, כולל תמונות והסברים, מופיע במאמר נוסף שפרסמתי ב-GitHub.

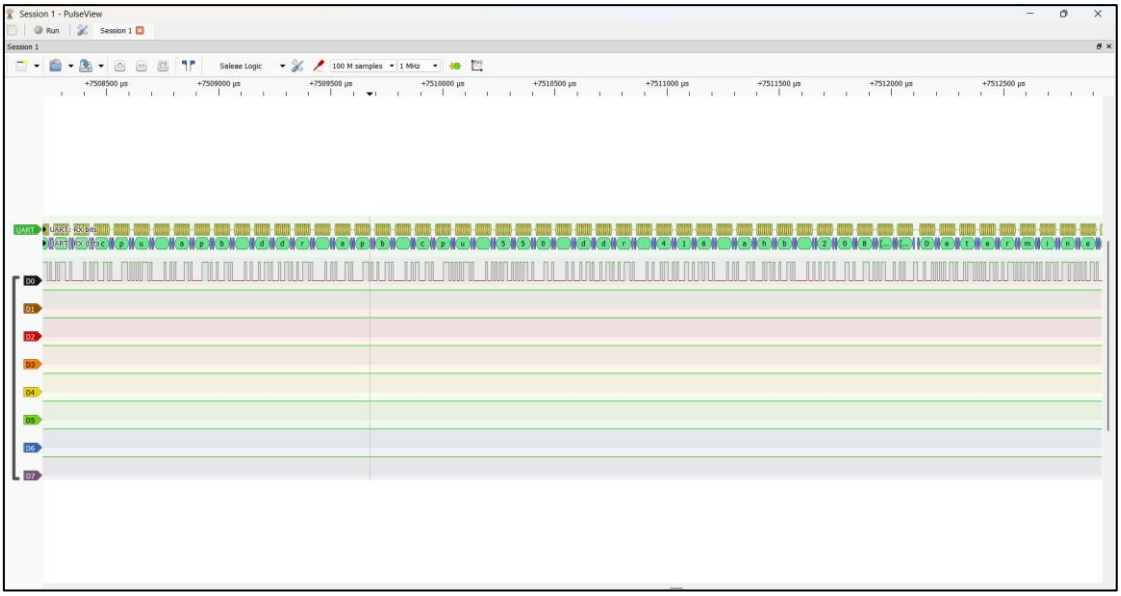
מדובר בעבודה עדינה הדורשת ציוד הלחמה מתאים וניסיון בעבודה עם רכיבים זעירים. לאחר השלמת התיקון וסגירת המעגל, ממשק ה-UART חזר לפעול באופן תקין.

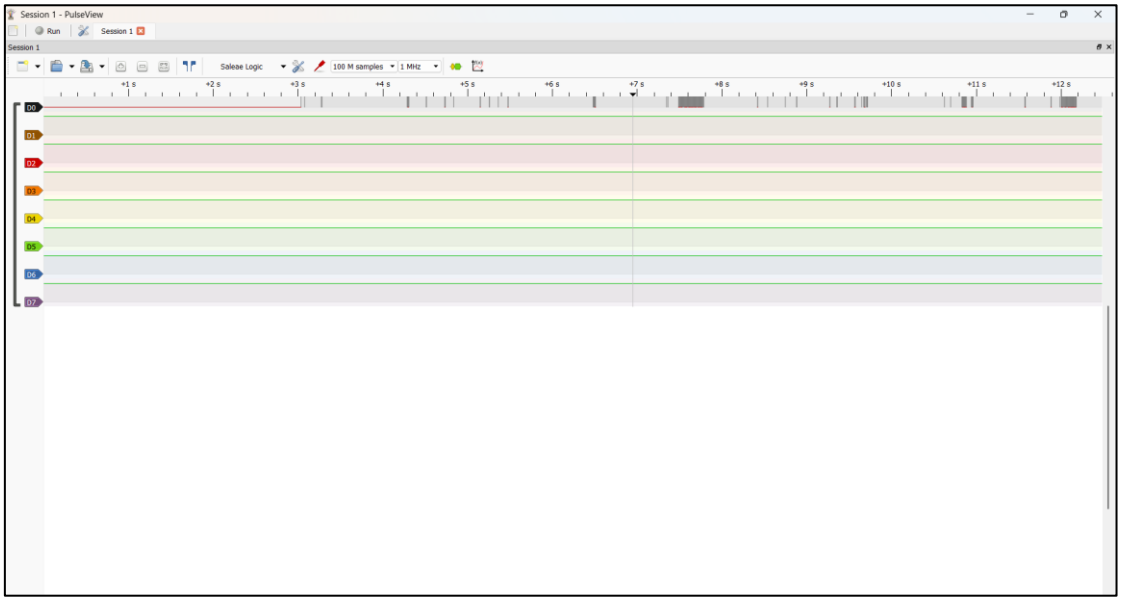




אחרי שהחזרנו את ה-UART לחיים, נותרה שאלה אחת: מהו קצב התקשורת (baud rate)? בלי הקצב הנכון נקבל ג'יבריש על המסך.

אפשר לנחש מרשימה קצרה של קצבים נפוצים (9600, 57600, 115200...), אבל יש דרך אלגנטית יותר, מנתח לוגי (logic analyzer). חיברתי מנתח לוגי זול של 8 ערוצים, בין השאר אל קו ה-TX, הקלטתי את הסיגנל בזמן אתחול, ובעזרת Sigrok PulseView מדדתי את רוחב הביט הבודד הקצר ביותר, וממנו חישבתי את הקצב:

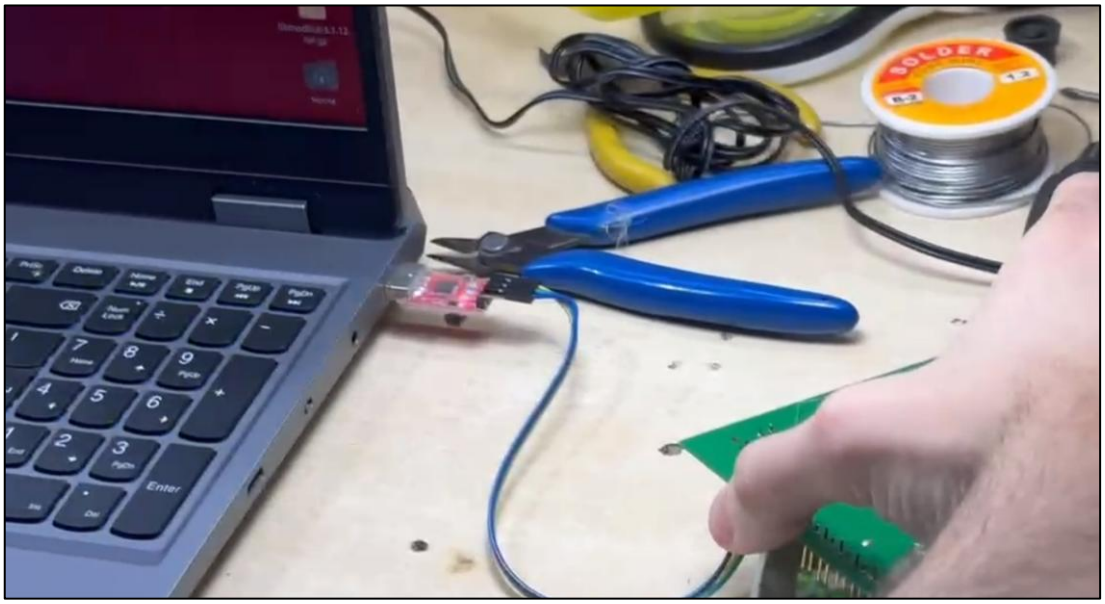


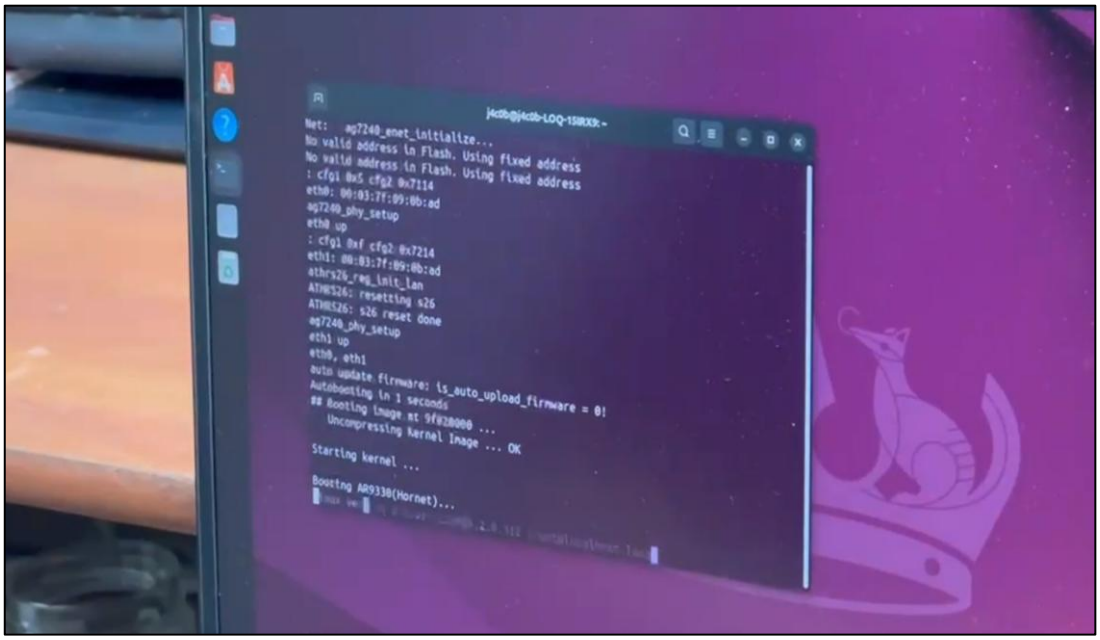


התוצאה: 115200.

עכשיו את הסיגנל הטורי צריך לתרגם ל-USB, ולשם כך משתמשים במתאם USB To UART.

מחברים את ה-TX של הנתב ל-RX של המתאם ולהיפך, עם GND משותף:





ועכשיו:

```
j4c0b@j4c0b-LOQ-15IRX9:~$ sudo screen /dev/ttyUSB0 115200
```

לאחר חיבור מתאם ה-UART למחשב, ניתן לפתוח את החיבור הסדרתי באמצעות הפקודה sudo screen עם ההתקן (למשל /dev/ttyUSB0) וקצב התקשורת המתאים. לאחר שהחיבור נפתח, מפעילים את הנתב מחדש. מרגע ההדלקה מתחיל להופיע במסוף שטף הודעות האתחול (Boot Log) בזמן אמת, המאפשר לעקוב אחר כל שלבי העלייה של המערכת, החל מה-Bootloader ועד לטעינת מערכת ההפעלה. שלב זה מספק מידע רב על תהליך האתחול. במהלך האתחול יש חשיבות רבה לתזמון הלחיצה על מקש ENTER. אם הלחיצה מתבצעת מוקדם מדי, בזמן שה-Bootloader עדיין פעיל, מתקבלת גישה לממשק של U-Boot בלבד. לעומת זאת, אם ממתינים לסיום תהליך האתחול, המערכת מגיעה למסך ההתחברות (Login Prompt) של הנתב.

במחקר שביצעתי ניסיתי להתחבר באמצעות ממשק ההתחברות של המערכת, אך ללא הצלחה מאחר והיה נדרש להקיש שם משתמש וסיסמא שלא ברשותי. ניסיון זה הדגיש עיקרון חשוב במחקר Firmware: כאשר לא ניתן לקבל גישה דרך מנגנוני ההתחברות הרגילים, ניתן להמשיך את המחקר באמצעות חילוץ וניתוח ה-Firmware עצמה. את הקונסולה החיה, לרבות תהליך ההתחברות כ-root בנתב TL-WR740N, ניתן לראות במלואה במאמר שפרסמתי ב-GitHub.

השלב הבא במחקר הוא חילוץ ה-Firmware ישירות מזיכרון ה-Flash של הנתב. בשלב זה כבר לא נעשה שימוש בממשק ה-UART, אלא בגישה ישירה לשבב הזיכרון, המאפשרת לחלץ את תוכן ה-Firmware לצורך ניתוח מעמיק.

בהרבה נתבים אפשר פשוט להוריד את קובץ ה-Firmware הרשמי מאתר היצרן ולנתח אותו בנוחות. זו הדרך הקלה, אבל היא גם מוגבלת: מה עושים כשאין Firmware זמינה להורדה, כשרוצים לוודא שמה שרץ על המכשיר זהה בדיוק למה שהיצרן מפרסם, או כשרוצים לגבות את המצב הנוכחי לפני שמשנים משהו? התשובה לכל אלה זהה: קוראים את שבב ה-Flash ישירות. וזה בדיוק מה שעשיתי.

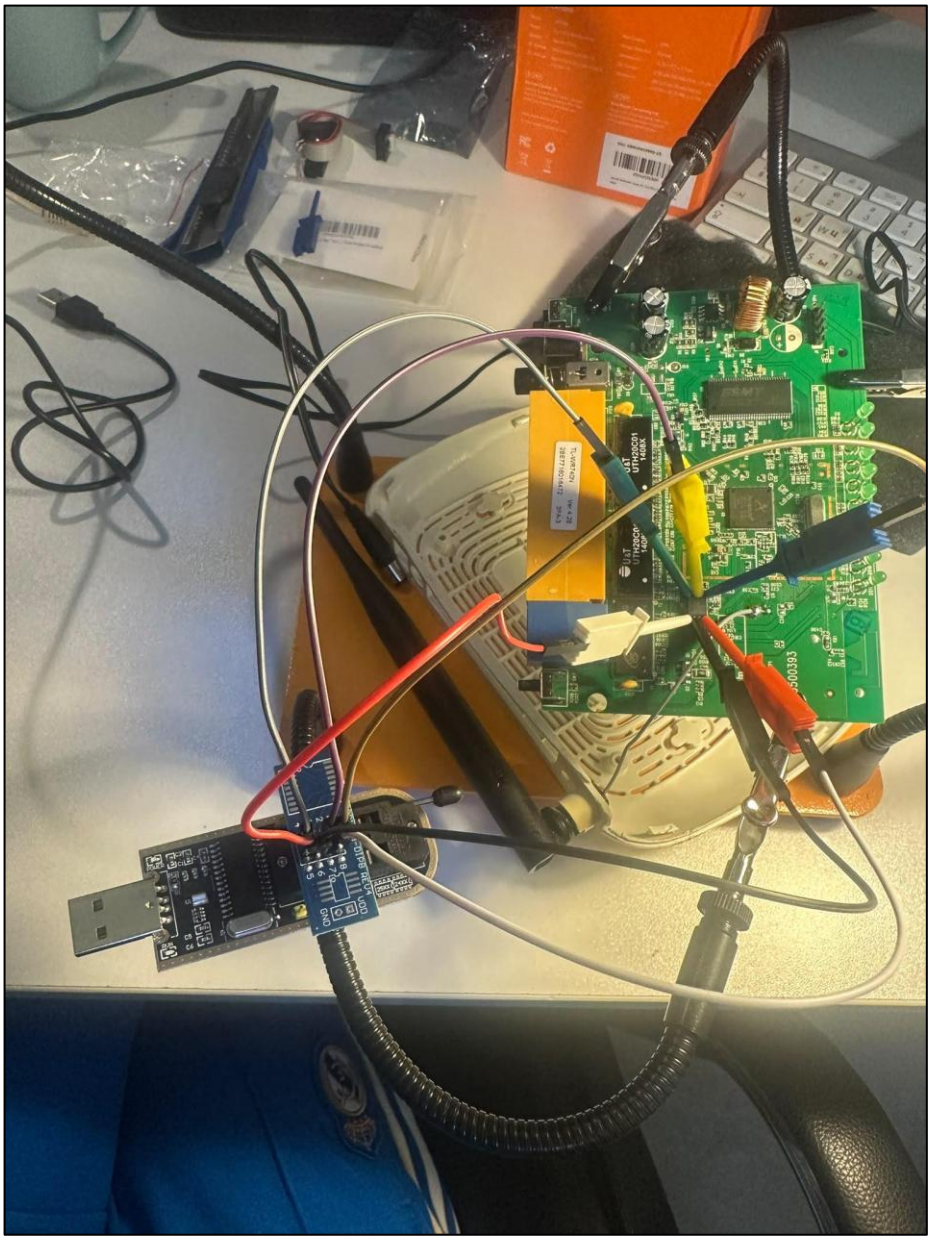
ה-Firmware יושבת בשבב SPI-NOR Flash. כדי לקרוא אותו נשתמש בקורא שבבים זול ונפוץ, ה-CH341A, שמתחבר ל-USB ומדבר SPI מול השבב:

זהירות, מתח.

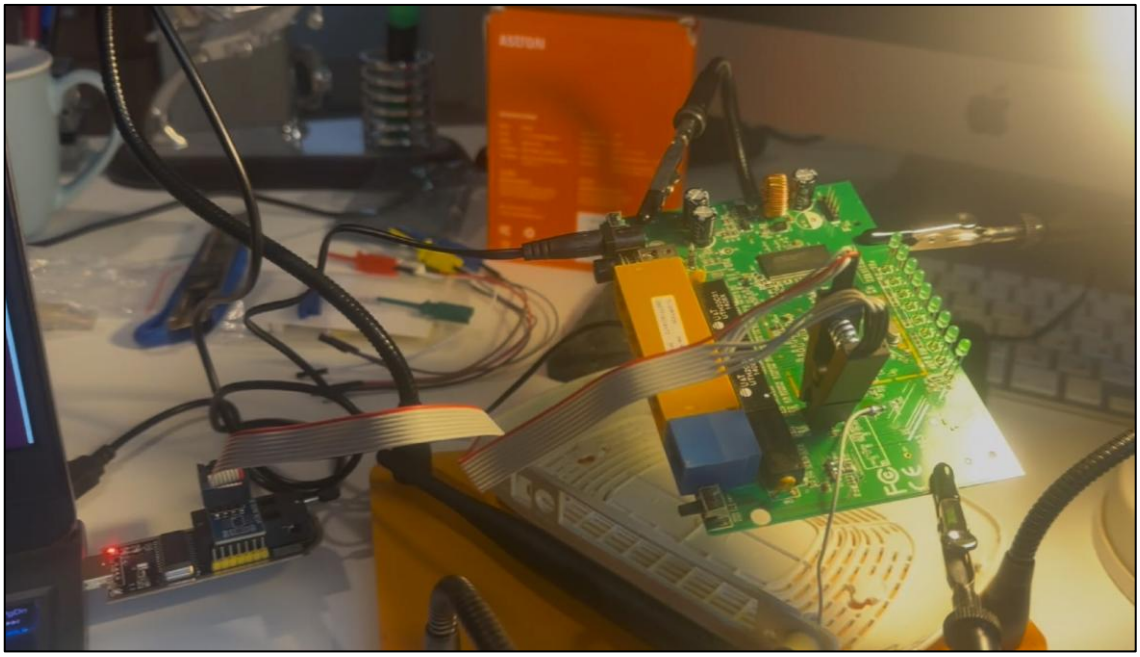
חלק גדול ממתכנתי ה-CH341A הזולים מוציאים V5 על קווי הנתונים כברירת מחדל, בעוד שבב ה-Flash מצפה ל-V3.3. חיבור ישיר כזה עלול לשרוף את השבב. מומלץ לוודא/לתקן את המתכנת ל-V3.3 לפני שמחברים אותו ליעד או לקנות את הדגם החדש - בצבע תכלת שאין לו את הבעיה של המתחים כמו לדגם הישן.

יש שתי דרכים לקרוא את השבב. הראשונה, in-circuit: מצמידים SOIC-8 clip ישירות לרגלי השבב על הלוח, בלי להלחים כלום. הדרך השנייה, האמינה יותר, היא chip-off: מלחימים את השבב מהלוח ומניחים אותו ישירות בשקע של המתכנת.

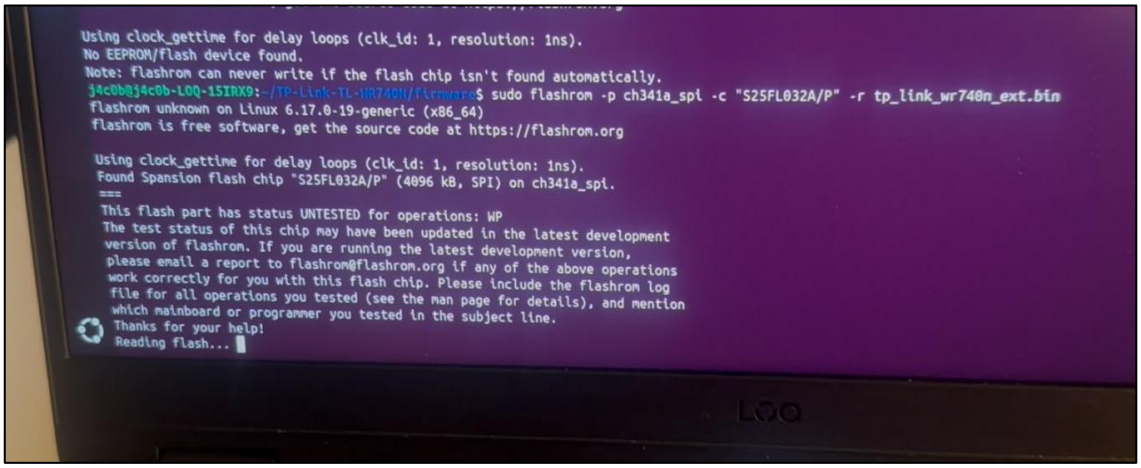
כך זה נראה כשהשבב מחובר לקריאה בדרך in-circuit:



מכאן העבודה תוכנתית. הכלי הסטנדרטי הוא flashrom: הוא מזהה את הדגם המדויק של השבב וקורא את התוכן: לאחר חיבור מתכנת ה-CH341A לשבב ה-Flash, ניתן להשתמש בכלי flashrom כדי לזהות את השבב ולחלץ את התוכן.



תחילה נזהה את שבב הזיכרון: flashrom -p ch341a_spi, לאחר מכן נקרא את כל תוכן זיכרון ה-Flash ונשמור אותו בקובץ בשם tp_link_wr740n_ext.bin:



כדי לוודא שהקריאה בוצעה בצורה תקינה וללא שגיאות, מומלץ לבצע קריאה נוספת של השבב ולשמור אותה בקובץ נוסף, ולאחר מכן להשוות בין שני הקבצים:

```
flashrom -p ch341a_spi -r verify.bin && cmp tp_link_wr740n_ext.bin verify.bin
```

אם הפקודה cmp אינה מחזירה פלט, משמעות הדבר היא ששני הקבצים זהים, ולכן ניתן להניח שחילוץ ה-Firmware בוצע באופן תקין ואמין. קריאה כפולה והשוואה היא הרגל חשוב: חיבור לא יציב יכול לתת קובץ שנראה תקין אבל מכיל שגיאות. ברגע ששתי הקריאות זהות, יש בידינו את ה-Firmware המלאה, בדיוק כפי שהיא צרובה על המכשיר. עכשיו אפשר לפרק אותה. לנתח חומרה זה כיף, אבל הממצאים האמיתיים מסתתרים בקוד. הכלי הראשון בארגז הוא binwalk, סכין הצבא השווייצרי לניתוח Firmware. הוא סורק את הקובץ הבינארי, מזהה חתימות (magic bytes) של פורמטים מוכרים, ומגלה לנו מה מסתתר בפנים ואיפה. נריץ:

```
binwalk -e firmware.bin
```

binwalk מזהה בדיוק את המבנה שדיברנו עליו. המטרה שלנו היא ה-SquashFS, בתוכה נמצאות כל הבינאריות והסקריפטים. הפעלנו את binwalk -e כדי לחלץ את הרכיבים.

הדבר הראשון שכל חוקר רץ אליו הוא קבצי המשתמשים, /etc/passwd ו-/etc/shadow. מה שמצאנו שם מלמד המון על איך נראית אבטחה (או היעדרה) בנתב מסחרי:

1. חשבון Admin עם UID 0: מלבד root, קיים חשבון שני בשם Admin שגם לו UID 0, כלומר הרשאות root מלאות. זו דלת אחורית (backdoor) לכל דבר ועניין, שחולקת בדיוק את אותו hash סיסמה כמו root.

2. חשבון עם סיסמה ריקה: חשבון שירות (חשבון דיבאג של יצרן ה-SoC) עם שדה סיסמה ריק לחלוטין. כל מי שמגיע ל-shell יכול להתחבר דרכו בלי שום סיסמה.

3. שימוש חוזר בסיסמת root: ה-hash של root התגלה כבר מפוצח בפורום של OpenWrt, אבל בהקשר של דגם נתב אחר לגמרי של TP-Link. במילים אחרות, אותה סיסמת root משמשת מחדש על פני משפחות שלמות של מכשירים.

בפרקים הקודמים ראינו איך מגיעים אל תוך ה-Firmware של הנתב, איך מחלצים ממנו את מערכת הקבצים ואיך מזהים את הבינארי ששווה עליו את המסע. המכשיר שעליו עבדנו הוא TP-Link TL-WR740N בגרסת חומרה v4, עם Firmware בגרסה 3.17.0 Build 150105. מדובר בנתב אלחוטי ביתי צנוע, מבוסס מעבד MIPS בגרסת MSB, עם ספריית uClibc הקטנה האופיינית למערכות מוטמעות. החלטנו שנתמקד בבינארי httpd, הוא שרת ה-Web של הנתב. זהו משטח התקיפה הגדול ביותר מרחוק, כי הוא מקבל קלט מכל מי שמגיע לממשק הWeb. כל בקשת HTTP שנשלחת אל הנתב עוברת דרכו, וכל שדה בבקשה כזו הוא נקודת קלט פוטנציאלית שהתוקף שולט בה במלואה. כשמדובר בקוד C שנכתב לפני שנים, בלי הגנות מודרניות, זהו בדיוק סוג הבינארי שבו מוצאים חולשות זיכרון קלאסיות.

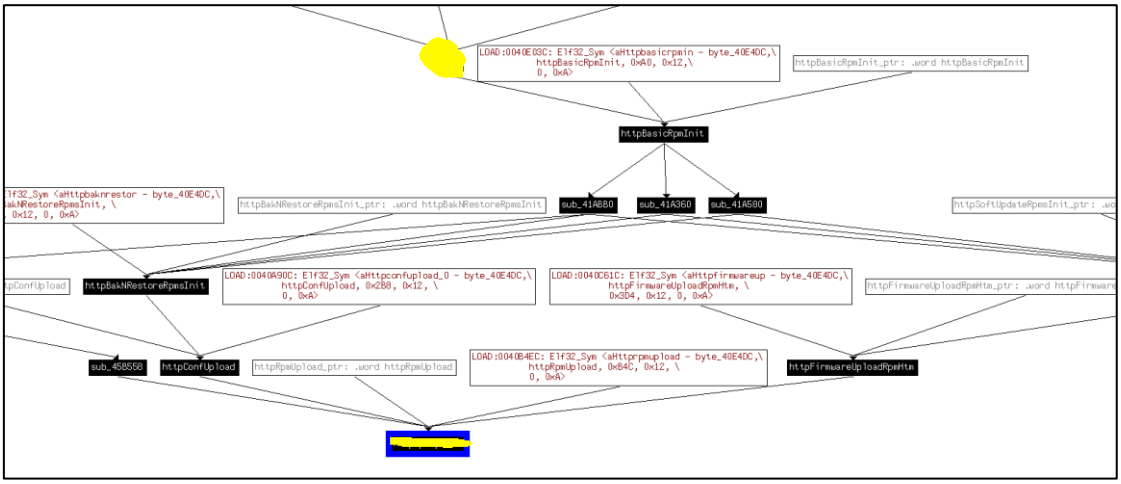
החולשה שנתאר בפרק הזה היא Heap Overflow בפרסר של העלאות הקבצים בשרת ה-Web. היא ניתנת לניצול מרחוק, ברשת המקומית, וללא כל צורך באימות או בסיסמה. התוצאה המודגמת היא מניעת שירות (Denial of Service) מלאה של ממשק הניהול של הנתב: בקשה אחת מספיקה כדי לפגוע במנגנון ניהול הזיכרון עד לאתחול ידני של המכשיר.

נעבור למחקר של החולשה, הבינארי המעניין, httpd, למה דווקא ה-httpd, לפני שנצלול אל הקוד עצמו, כדאי להבין את העיקרון. חוצץ (buffer) הוא אזור זיכרון בגודל קבוע שהוקצה מראש לצורך אחסון נתונים, למשל 2048 בתים. כל עוד התוכנית כותבת לתוך החוצץ הזה כמות נתונים שקטנה או שווה לגודלו, הכל תקין. הבעיה מתחילה כשהתוכנית כותבת יותר בתים ממה שהחוצץ יכול להכיל, בלי לבדוק את הגבול. הבתים העודפים לא נעלמים, הם נכתבים אל הזיכרון שנמצא מיד אחרי החוצץ, ודורסים את מה שהיה שם.

השאלה הגדולה היא מה בדיוק נמצא מיד אחרי החוצץ. אם שם יושבים נתונים חשובים, הרי שהתוקף, בכך שהוא שולט בבתים העודפים, שולט למעשה בערכים הרגישים האלה. זה מה שהופך גלישת חוצץ תמימה לכאורה לחולשה שעלולה להוביל להשתלטות מלאה על המכשיר.

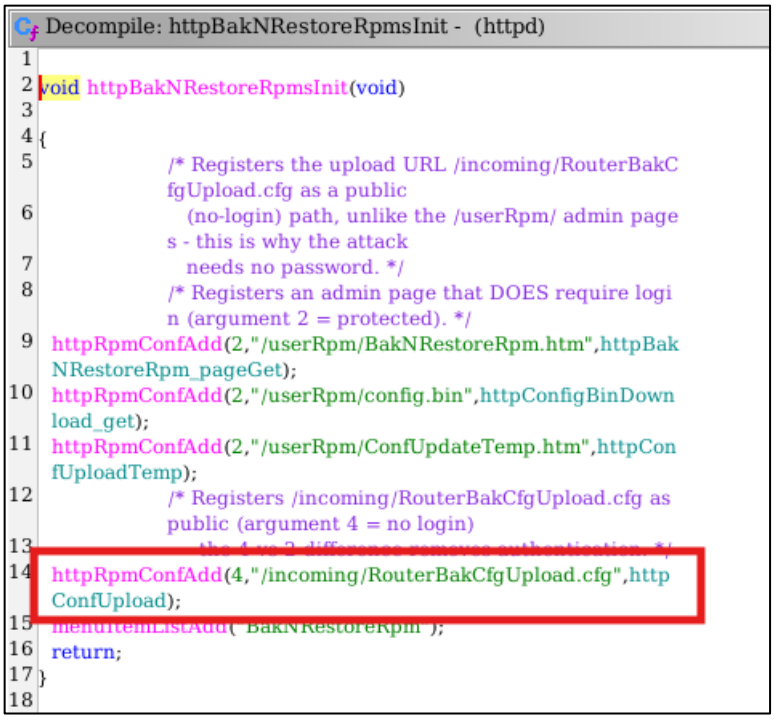
בחולשה הספציפית שלנו, מה שנמצא מיד אחרי החוצץ שנגלש הוא לא סתם נתונים, אלה מבני הבקרה של מנגנון ניהול הזיכרון עצמו. וזו בדיוק הסיבה שהגלישה כאן כל כך הרסנית, כפי שנראה בהמשך.

שורש החולשה: הפונקציה httpRpmUpload, מהmain אל הפונקציה.



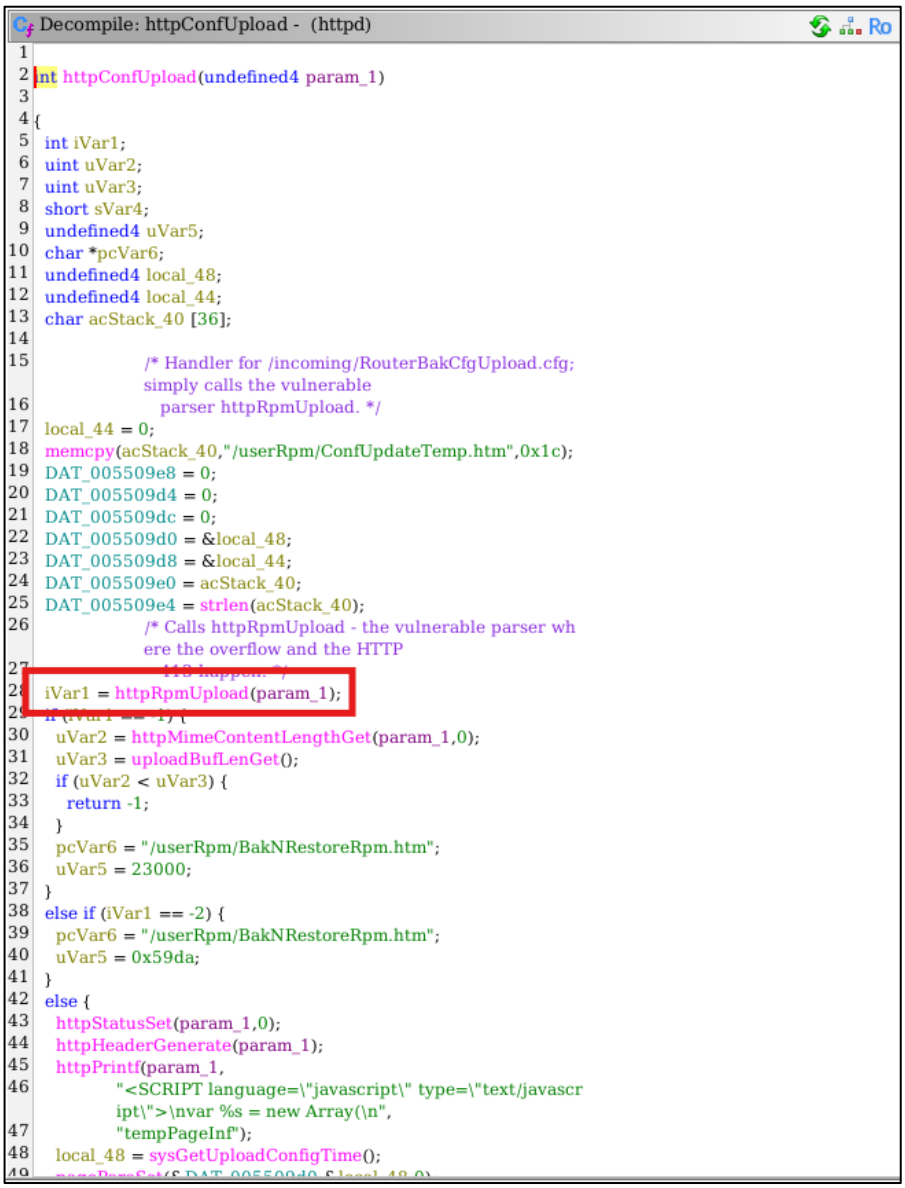
הדרך אל החולשה מתחילה בבקשת HTTP POST.

כאשר מישהו רוצה לשחזר קובץ הגדרות אל הנתב, הדפדפן שולח בקשת POST אל הכתובת שמטפלת בשחזור הגדרות. הבקשה הזו היא מסוג multipart/form-data, הפורמט הסטנדרטי שבו דפדפנים שולחים טפסים שכוללים קבצים. הבקשה מגיעה תחילה אל פונקציית הטיפול בהעלאות, שמזהה את סוג הבקשה ומעבירה אותה הלאה אל הפרסר שמפרק אותה לחלקים.



בתמונה המצורפת ניתן לראות שאנחנו ניגשים לפונקציה httpConfUpload ברגע שאנחנו מקבלים בקשת POST עם הנתיב הבא: /incoming/RouterBakCfgUpload.cfg

```
httpRpmConfAdd( 4 , "/incoming/RouterBakCfgUpload.cfg" , httpConfUpload );
                |                    |                   |
             METHOD                URL                HANDLER
           "on POST"     "for this path"     "run this function"
```



```
Decompile: httpConfUpload -  (httpd)
 1
 2 int httpConfUpload(undefined4 param_1)
 3
 4 {
 5   int iVar1;
 6   uint uVar2;
 7   uint uVar3;
 8   short sVar4;
 9   undefined4 uVar5;
10   char *pcVar6;
11   undefined4 local_48;
12   undefined4 local_44;
13   char acStack_40 [36];
14
15                 /* Handler for /incoming/RouterBakCfgUpload.cfg;
                   simply calls the vulnerable
16                   parser httpRpmUpload. */
17   local_44 = 0;
18   memcpy(acStack_40,"/userRpm/ConfUpdateTemp.htm",0x1c);
19   DAT_005509e8 = 0;
20   DAT_005509d4 = 0;
21   DAT_005509dc = 0;
22   DAT_005509d0 = &local_48;
23   DAT_005509d8 = &local_44;
24   DAT_005509e0 = acStack_40;
25   DAT_005509e4 = strlen(acStack_40);
26                 /* Calls httpRpmUpload - the vulnerable parser wh
                   ere the overflow and the HTTP
27                   ### happens. */
28   iVar1 = httpRpmUpload(param_1);
29   if (iVar1 == -1) {
30     uVar2 = httpMimeContentLengthGet(param_1,0);
31     uVar3 = uploadBufLenGet();
32     if (uVar2 < uVar3) {
33       return -1;
34     }
35     pcVar6 = "/userRpm/BakNRestoreRpm.htm";
36     uVar5 = 23000;
37   }
38   else if (iVar1 == -2) {
39     pcVar6 = "/userRpm/BakNRestoreRpm.htm";
40     uVar5 = 0x59da;
41   }
42   else {
43     httpStatusSet(param_1,0);
44     httpHeaderGenerate(param_1);
45     httpPrintf(param_1,
46             "<SCRIPT language=\"javascript\" type=\"text/javascr
                ipt\">\nvar %s = new Array(\n",
47             "tempPageInf");
48     local_48 = sysGetUploadConfigTime();
49     pageParaSet(&DAT_005509d0,&local_48,0)
```

עכשיו כשאנחנו בתוך הפונקציה httpConfUpload, הפרסר הזה - httpRpmUpload, הוא לב החולשה.

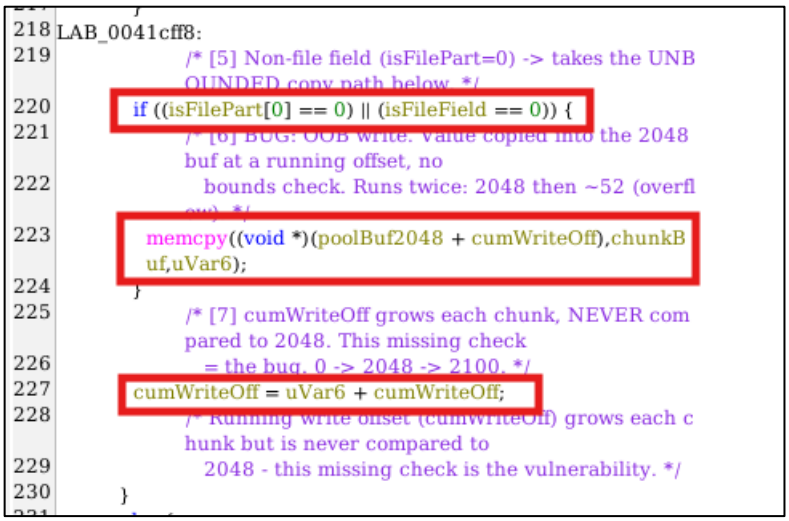
תפקידו של הפרסר לפרק את הבקשה לחלקיה. בקשת multipart בנויה מכמה חלקים, שכל אחד מהם מופרד ממשנהו על ידי מחרוזת גבול (boundary). כל חלק יכול להיות או שדה טופס רגיל, למשל שדה טקסט עם שם וערך, או קובץ שהועלה. הפרסר קורא את הבקשה חלק אחר חלק, ולכל חלק הוא צריך להחליט לאן להעתיק את הנתונים.

בתחילת הפונקציה מוקצים ארבעה חוצצים ממנגנון ניהול הזיכרון.

```
Decompile: httpRpmUpload -  (httpd)
                gin via /incoming/ (0x43c180).
50                */
51  isFilePart[0] = 1;
52  bytesRemaining = 0;
53  if ((DAT_005508b4 == (undefined1 *)0x0) &&
54     (DAT_005508b4 = (undefined1 *)uploadBufGet(), DAT_0055
       08b4 == (undefined1 *)0x0)) {
55    return -1;
56  }
57  *DAT_005508b4 = 0;
58  uploadFileSizeSet(0);
59  local_64 = DAT_005508b4;
60  pcVar3 = (char *)httpMimeContentTypeGet(param_1,0);
61                /* [1] Entry gate: only multipart/form-data is parse
                   d. The PoC matches. */
62  iVar4 = strcmp(pcVar3,"multipart/form-data");
63  uVar16 = 0xffffffff;
64  if (iVar4 != 0) goto LAB_0041d400;
65  local_58 = (char *)httpBoundaryGet(param_1);
66  local_38 = strlen(local_58);
67  memPool = httpReqMemPartIdGet(param_1);
68  uVar17 = (uint)(memPool == 0) << 3;
69  pvVar5 = (void *)memPoolAlloc(memPool,0x800);
70  if (pvVar5 == (void *)0x0) {
71    uVar17 = 8;
72  }
73  chunkBuf = (void *)memPoolAlloc(memPool,0x800);
74  if (chunkBuf == (void *)0x0) {
75    uVar17 = 8;
76  }
77  fieldNameBuf = (char *)memPoolAlloc(memPool,0x40);
78  if (fieldNameBuf == (char *)0x0) {
79    uVar17 = 8;
80  }
81                /* [2] Victim buffer: fixed 2048 bytes (0x800). Size
                   never depends on input. */
82  poolBuf2048 = memPoolAlloc(memPool,0x800);
83  if (poolBuf2048 == 0) {
84    uVar17 = 8;
85  }
86                /* [3] Content-Length -> loop counter. Only checke
                   d vs the BIG upload cap, not
87                  vs 2048. */
88  uVar6 = httpMimeContentLengthGet(param_1,0);
89  bytesRemaining = uVar6;
90  if (uVar17 == 0) {
91    uVar17 = uploadBufLenGet();
92    uVar9 = bytesRemaining;
93    if (uVar6 < uVar17) {
94      uVar6 = 0;
95      sVar7 = strlen(local_58);
96      iVar4 = httpLineRead(param_1,pvVar5,uVar7 + 4)
```

שלושה מהם בגודל 2048 בתים ואחד בגודל 64 בתים. אחד מהחוצצים בגודל 2048 מיועד לשמש כיעד שאליו מועתק הערך של שדה טופס רגיל שאינו קובץ, וזהו החוצץ שיגלוש - "poolBuf2048". חוצץ נוסף בן 64 בתים מיועד לשם השדה, ומכאן ואילך הוא לא רלוונטי לחולשה: שם השדה מועתק בעזרת פונקציה שחוסמת את האורך בבטחה, כך שהוא לעולם לא יגלוש. רק ערך השדה הוא זה שאינו חסום. וHttpMimeContentLengthGet מחזירה את הערך של Content-Length.

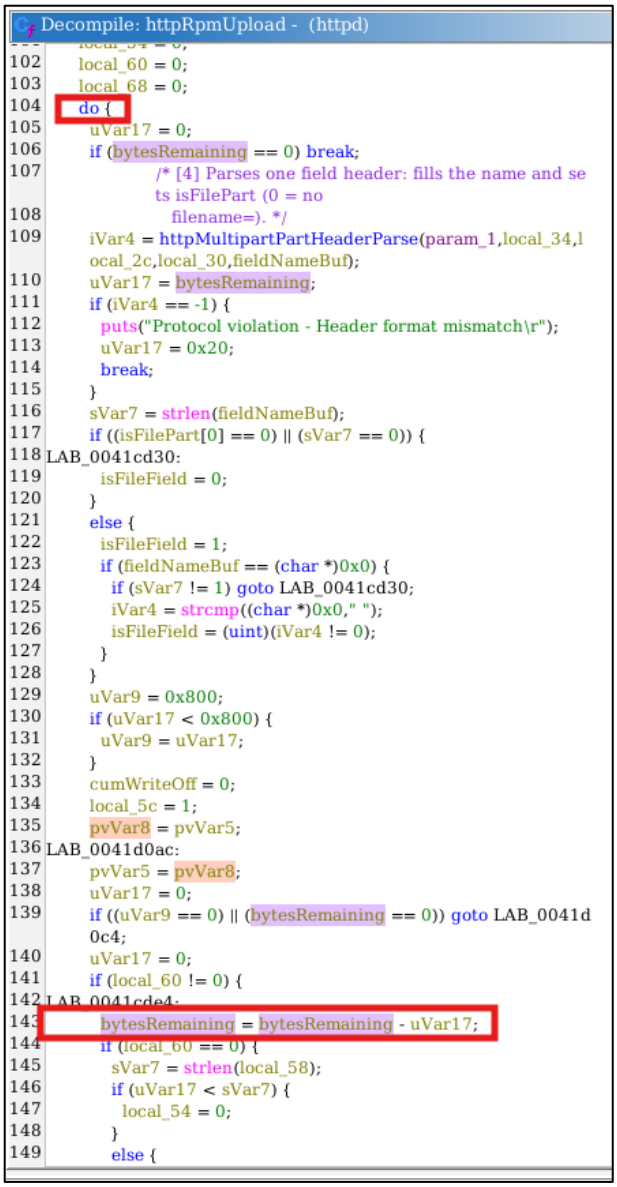
הנקודה המכרעת בכל הקוד היא ההבחנה בין קובץ לבין שדה רגיל. הפרסר מחזיק דגל פנימי שקובע האם החלק הנוכחי הוא קובץ שהועלה או שדה טופס רגיל. עבור קובץ, הערך מועתק אל חוצץ ההעלאה הגדול, חוצץ שגודלו המרבי מגיע עד ארבעה מגה - בתים, וההעתקה אליו חסומה כראוי. עבור שדה רגיל, לעומת זאת, הערך מועתק אל אותו חוצץ קטן בן 2048 הבתים וכאן, ורק כאן, אין שום בדיקת אורך.



הלולאה שגורמת לגלישה, חשוב להבין שגלישת החוצץ כאן אינה נובעת מהעתקה בודדת וענקית. כל העתקה בודדת מוגבלת בעצמה ל-2048 בתים לכל היותר, כך שאף פעולת העתקה יחידה אינה יכולה לבדה למלא את החוצץ מעבר לגבולו. מקור הבעיה הוא הלולאה.

המשתנה bytesRemaining מאותחל לערך של כותרת Content-Length (כל גוף הבקשה).

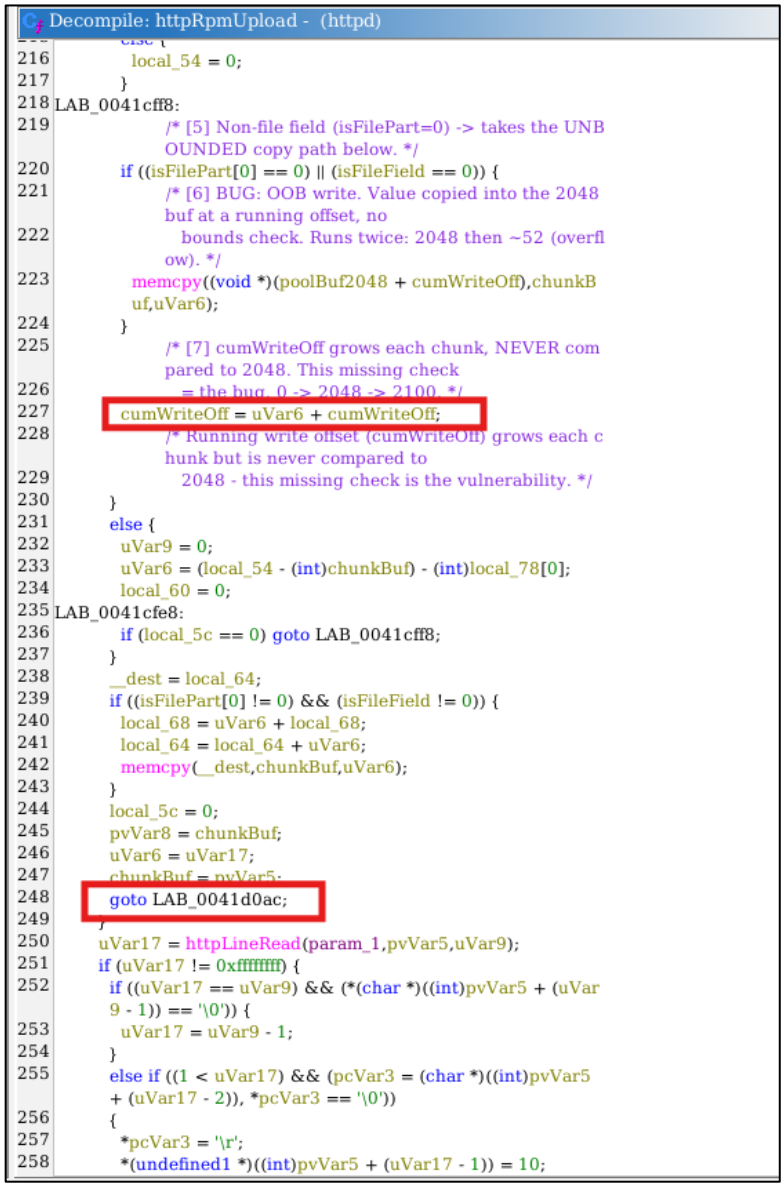
בכל איטרציה נקרא מקטע (chunkBuf) אחד אל באפר זמני, ולאחר מכן מועתק באמצעות memcpy אל הבאפר הקבוע poolBuf2048. מספר הבתים שנקראו מופחת מ־bytesRemaining, והלולאה ממשיכה לחזור על עצמה עד ש־bytesRemaining מגיע ל־0. המשתנה cumWriteOff לעולם אינו נבדק מול גודל הבאפר (2048 בתים). לכן, אם גוף הבקשה גדול מ־2048 בתים, פעולות ה־memcpy הבאות ימשיכו לכתוב מעבר לגבולות poolBuf2048.



הפרסר קורא את ערך השדה בנתחים (chunks). כל נתח הוא חתיכה בגודל של עד 2048 בתים מתוך ערך השדה. לאחר שהוא קורא נתח, הוא מעתיק אותו אל תוך החוצץ במיקום שנקבע על ידי מונה היסט (offset) מצטבר - cumWriteOff, לאחר ההעתקה, המונה המצטבר גדל בגודל הנתח שהועתק זה עתה. ואז הלולאה חוזרת אל תחילתה, קוראת את הנתח הבא, ומעתיקה אותו אל החוצץ הפעם במיקום מתקדם יותר, לפי המונה שכבר גדל.

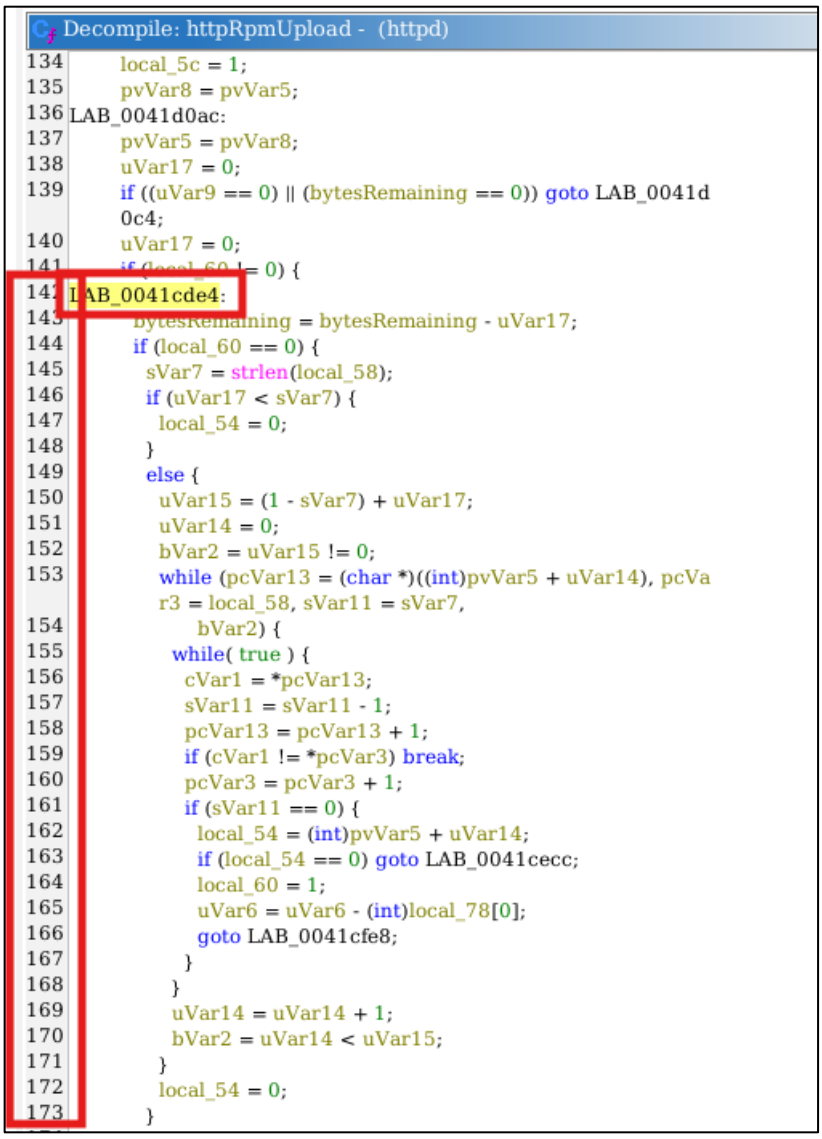
וכאן טמונה החולשה: המונה המצטבר cumWriteOff לעולם אינו נבדק מול גודל החוצץ. הוא פשוט ממשיך לגדול, נתח אחר נתח, בלי שאף שורת קוד תעצור ותשווה אותו מול הגבול של 2048. הנתח הראשון נכתב אל תחילת החוצץ ותקין. הנתח השני, אם ערך השדה ארוך מספיק, כבר נכתב סביב הבית ה-2048 כלומר בדיוק בקצה החוצץ ומעבר לו.

כל נתח נוסף אחריו נכתב עמוק יותר ויותר אל תוך הזיכרון שאחרי החוצץ.



```
Decompile: httpRpmUpload - (httpd)
216          local_54 = 0;
217        }
218 LAB_0041cff8:
219                /* [5] Non-file field (isFilePart=0) -> takes the UNB
                   OUNDED copy path below. */
220        if ((isFilePart[0] == 0) || (isFileField == 0)) {
221                /* [6] BUG: OOB write. Value copied into the 2048
                   buf at a running offset, no
222                  bounds check. Runs twice: 2048 then ~52 (overfl
                   ow). */
223          memcpy((void *)(poolBuf2048 + cumWriteOff),chunkB
             uf,uVar6);
224        }
225                /* [7] cumWriteOff grows each chunk, NEVER com
                   pared to 2048. This missing check
226                  = the bug. 0 -> 2048 -> 2100. */
227        cumWriteOff = uVar6 + cumWriteOff;
228                /* Running write offset (cumWriteOff) grows each c
                   hunk but is never compared to
229                  2048 - this missing check is the vulnerability. */
230      }
231      else {
232        uVar9 = 0;
233        uVar6 = (local_54 - (int)chunkBuf) - (int)local_78[0];
234        local_60 = 0;
235 LAB_0041cfe8:
236        if (local_5c == 0) goto LAB_0041cff8;
237      }
238      __dest = local_64;
239      if ((isFilePart[0] != 0) && (isFileField != 0)) {
240        local_68 = uVar6 + local_68;
241        local_64 = local_64 + uVar6;
242        memcpy(__dest,chunkBuf,uVar6);
243      }
244      local_5c = 0;
245      pvVar8 = chunkBuf;
246      uVar6 = uVar17;
247      chunkBuf = pvVar5;
248      goto LAB_0041d0ac;
249      }
250      uVar17 = httpLineRead(param_1,pvVar5,uVar9);
251      if (uVar17 != 0xffffffff) {
252        if ((uVar17 == uVar9) && (*(char *)((int)pvVar5 + (uVar
           9 - 1)) == '\0')) {
253          uVar17 = uVar9 - 1;
254        }
255        else if ((1 < uVar17) && (pcVar3 = (char *)((int)pvVar5
           + (uVar17 - 2)), *pcVar3 == '\0'))
256        {
257          *pcVar3 = '\r';
258          *(undefined1 *)((int)pvVar5 + (uVar17 - 1)) = 10;
```

אם תשימו לב הפונקציה LAB_0041d0ac מחזירה אותנו כמעט להתחלה, ניתן לראות את זה בתמונה הבאה ובמספר שורה.



מכיוון שהתקרה על אורך כל הבקשה מגיעה עד ארבעה מגה - בתים, התוקף יכול לשלוח ערך שדה גדול מאוד. ערך כזה יגרום ללולאה לרוץ עשרות פעמים, ובכל פעם לדחוף את מונה הכתיבה עוד ועוד הרחק אל מעבר לחוצץ. כך, בעצם שליטה באורך ערך השדה, התוקף שולט בכמה רחוק הגלישה מגיעה. מה שנראה כמו חוצץ קטן ומוגן בן 2048 בתים הופך, דרך הלולאה, לצינור שכותב כמות כמעט בלתי מוגבלת של בתים אל תוך הזיכרון.

למה הגלישה כל כך הרסנית? עד כה תיארנו שהבתים העודפים נכתבים אל הזיכרון שאחרי החוצץ. עכשיו נבין למה דווקא הזיכרון הזה כל כך רגיש.

החוצצים בפרסר אינם מוקצים ישירות ממערכת ההפעלה, אלא ממנגנון ניהול זיכרון פנימי משלו, מעין מקצה זיכרון (allocator) שמנהל בריכה (pool) של בלוקים. המנגנון הזה שומר לפני כל בלוק של נתונים כותרת בקרה בגודל 16 בתים. הכותרת הזו מכילה כמה שדות: ערך "קסם" (magic) שמזהה האם הבלוק בשימוש או פנוי, מצביע אל הבלוק הבא ברשימת הבלוקים הפנויים, גודל הבלוק, וריפוד. ערך הקסם מקבל דפוס אחד כשהבלוק בשימוש ודפוס אחר כשהוא פנוי, כך המנגנון יודע להבחין ביניהם.

הבלוקים בבריכה יושבים בזיכרון בזה אחר זה, צמודים. המשמעות היא שמיד אחרי אזור הנתונים של החוצץ שלנו, אותו חוצץ בן 2048 הבתים יושבת הכותרת של הבלוק הבא. ולכן, כשהגלישה חורגת מעבר לחוצץ, הבתים העודפים דורסים בדיוק את הכותרת הזו: קודם את ערך הקסם, ומיד אחריו את המצביע אל הבלוק הבא.

זו הנקודה הקריטית. המנגנון הזה מסתמך על הכותרות האלה כדי לתפקד. כשהוא משחרר בלוק, הוא בודק את ערך הקסם שלו כדי לוודא שהבלוק תקין. וכשהוא מקצה או משחרר בלוקים, הוא מטייל לאורך רשימת הבלוקים הפנויים על ידי מעקב אחר המצביעים "הבא". אם דרסנו את ערך הקסם הבדיקה נכשלת. ואם דרסנו את המצביע "הבא", הטיול לאורך הרשימה מוביל אל כתובת שגויה שהתוקף שלט בה. הזווית שהופכת את זה לחמור: ללא אימות

עד כה תיארנו את הגלישה ואת ההשלכה שלה, מניעת שירות. אבל יש רכיב נוסף שמעלה את חומרת החולשה באופן משמעותי: הכתובת שדרכה מגיעים אל הפרסר אינה מוגנת בסיסמה כלל.

מודל האימות של שרת ה-Web בנוי בשיטת "הכל מותר כברירת מחדל". כלומר, רק נתיבים שנרשמו במפורש עם כלל אימות דורשים את סיסמת המנהל. במהלך הניתוח מיפינו את רשימת כללי האימות, וראינו ששני משפחות נתיבים מרכזיות אכן מוגנות ודורשות התחברות. אבל הנתיב שדרכו מגיעים אל פרסר ההעלאות, נתיב שחזור ההגדרות, אינו מופיע ברשימת הכללים המוגנים. הוא פשוט לא נרשם, ולכן הוא פתוח לחלוטין.

```
Decompile: httpd -  (httpd)
1
2  undefined4 httpd(void)
3
4  {
5    int iVar1;
6    undefined4 uVar2;
7    undefined1 auStack_50 [32];
8    undefined1 auStack_30 [16];
9    undefined1 auStack_20 [16];
10
11   iVar1 = httpInit();
12   if (iVar1 == 0) {
13     httpAliasConfAdd("/","/userRpm/Index.htm");
14     httpAliasConfAdd(&DAT_0051009c,"*/Index.htm");
15     httpAliasConfAdd("/images/*","/fs/images/*");
16     httpAliasConfAdd("/frames/*","/fs/frames/*");
17     httpAliasConfAdd("/dynaform/*","/fs/dynaform/*");
18     httpAliasConfAdd("/userRpm/*","/userRpm/*");
19     httpAliasConfAdd("/localiztion/*","/fs/localiztion/*");
20     httpAliasConfAdd("/help/*","/help/*");
21     swGetPasswordCfg(auStack_50);
22     httpPwdConfAdd("admin",auStack_30,auStack_20);
23     httpCtrlConfAdd("/fs/*","admin","Everywhere");
24     httpCtrlConfAdd("/userRpm/*","admin","Everywhere");
25     httpRpmConfAdd(2,&DAT_00510154,httpRpmFs);
26     httpFsConfAdd(&DAT_00510154,"/rc_filesys/doc/");
27     httpFileSetDefaultFs(1);
28     httpUploadConfAdd("/incoming/","/rc_filesys/");
29     uVar2 = httpServerStart();
30   }
31   else {
32     puts("httpInit error\r");
33     uVar2 = 0xffffffff;
34   }
35   return uVar2;
36 }
37
```

המשמעות המעשית היא שכל בקשה שמגיעה אל הנתיב הזה מגיעה אל הפרסר בלי שום בדיקת הרשאות. אישרנו זאת גם בזמן ריצה: בקשה נטולת פרטי התחברות מגיעה אל הפרסר ומקבלת תשובה עניינית, לעולם לא דף התחברות. זהו הרכיב שהופך את מה שהיה יכול להיות עוד מניעת שירות למניעת שירות שכל אחד ברשת יכול להפעיל, בלי חשבון ובלי סיסמה. אורח על רשת ה-Wi-Fi, או אפילו דף אינטרנט זדוני שמישהו ברשת גולש אליו בדפדפן שלו, יכולים לשגר את הבקשה הזו ולהשבית את ממשק הניהול של הנתב.

הפוטנציאל שלא הודגם: הגינות מחקרית מחייבת אותנו להפריד בבירור בין מה שהוכח לבין מה שרק אפשרי. מה שהדגמנו בפועל הוא מניעת שירות, השבתת ממשק הניהול. את זה ראינו בעינינו, שוב ושוב, בצורה שחוזרת על עצמה.

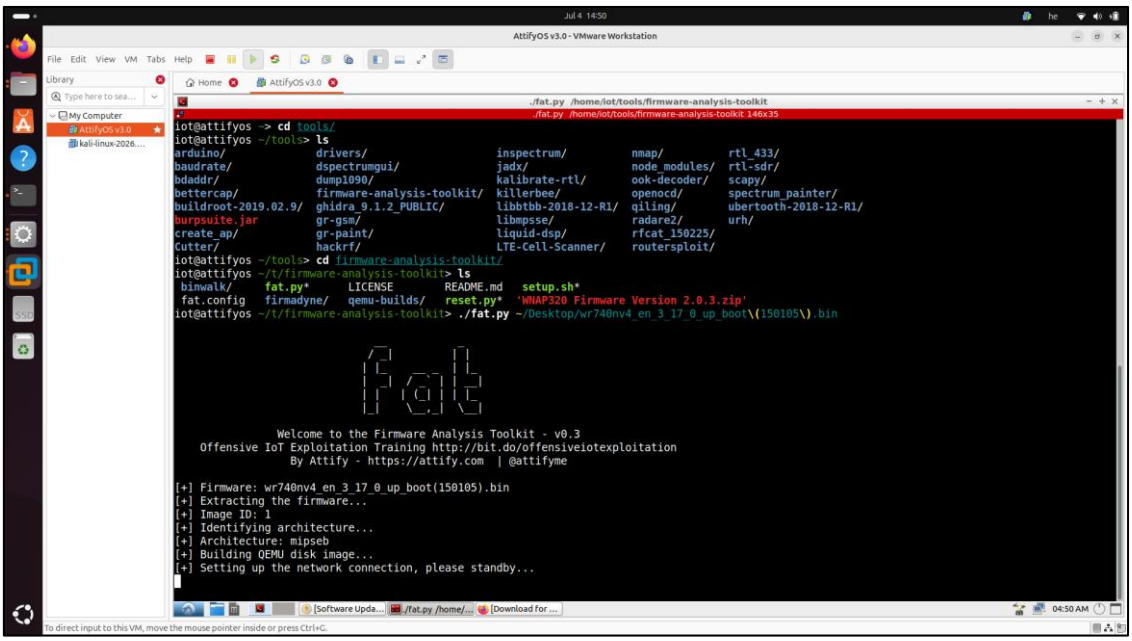
אבל יש כאן פוטנציאל חמור יותר, שלא מימשנו. כזכור, הגלישה דורסת את המצביע "הבא" ברשימת הבלוקים הפנויים בערכים שהתוקף שולט בהם. שליטה במצביע כזה היא אבן דרך קלאסית ומוכרת בעולם ניצול חולשות הזיכרון, היא יכולה, בעיקרון, להוביל לכתיבה אל כתובת זיכרון שרירותית שהתוקף בוחר, ומשם אף להרצת קוד. ואם ניקח בחשבון ששרת ה-Web רץ בהרשאות root, הרי שהרצת קוד כזו הייתה בהרשאות הגבוהות ביותר.

אנחנו לא הפכנו את הגלישה הזו לכתיבה שרירותית או להרצת קוד. כדי לעשות זאת היה צורך לשלוט בדיוק בבתים הנכתבים ובפריסת הזיכרון, עבודה שלא ביצענו. אבל הפוטנציאל קיים.

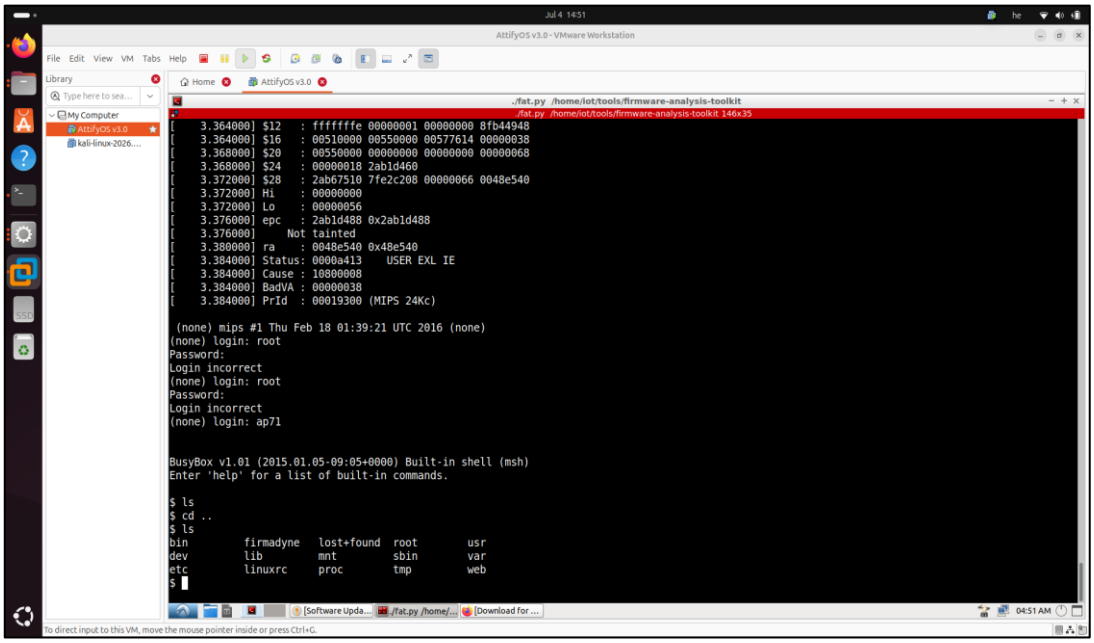
המעבדה החיה, סביבת אמולציה, עד כה עבדנו מול הנתב הפיזי, או ניתחנו את ה-Firmware באופן סטטי, כלומר קראנו את הקוד בלי להריץ אותו. אבל בין שתי הגישות האלה יש שלב ביניים עוצמתי במיוחד: אמולציה. הרעיון הוא להריץ את ה-Firmware כתוכנה על המחשב, בתוך מכונה וירטואלית, בלי שהחומרה בכלל מחוברת.

היתרון עצום. פתאום אפשר לדבר עם ממשק ה-Web של הנתב, לשלוח אליו בקשות, להריץ עליו כלי דיבוג, הכל בסביבה בטוחה לחלוטין ובלי שום חומרה. אין יותר חשש לשרוף לוח, כפי שקרה לי כשניסיתי לחקור רחפן יפני זול. אמולציה היא הדרך הבטוחה לבדוק חולשות כמו זו שתיארתי. כדי לאמת שהגלישה אכן משביתה את השרת, הייתי צריך לשגר אליו בקשות פוגעניות שוב ושוב, ובסביבה מאומלצת אפשר לעשות זאת בלי כל סיכון, ולאתחל את הכל בלחיצת כפתור בכל פעם שמשהו נתקע.

הכלי המרכזי שבו השתמשתי הוא FAT, קיצור של Firmware Analysis Toolkit.



FAT הוא למעשה עטיפה נוחה סביב פרויקט בשם Firmadyne. תפקידו של Firmadyne הוא לקחת תמונת קובץ Firmware גולמי - bin. ולחלץ ממנה את מערכת הקבצים, ולהריץ אותה בתוך אמולטור QEMU עם קרנל תואם לארכיטקטורה, במקרה שלי, קרנל של MIPS. במילים אחרות, FAT לוקח קובץ Firmware ומנסה, כמעט לבד, להפוך אותו לנתב חי שרץ על המחשב.



את כל ערכת הכלים הזו מקבלים ארוזה ומוכנה בתוך AttifyOS - הפצת לינוקס ייעודית לבדיקות חדירה של מכשירי IoT - מאוד מומלץ, ההפצה מגיעה עם כל הכלים שאנחנו צריכים כבר מותקנים: binwalk לחילוץ, Firmadyne ו-FAT לאמולציה, ו-Ghidra ו-radare2 ל-reverse engineering. ההפצה מופצת כקובץ מכונה וירטואלית שמייבאים אל תוכנת הווירטואליזציה, ונכנסים אליה עם שם משתמש וסיסמה שמגיעים כברירת מחדל.

חשוב לציין ש-FAT/Firmadyne הם לא הדרך היחידה. אפשר להקים סביבת אמולציה גם באמצעות Buildroot - בנייה של מערכת לינוקס מוטמעת בהתאמה אישית, שבתוכה מריצים את הבינארי של הנתב. חבריי הטובים כבר הסבירו את התהליך הזה לעומק, מוזמנים לקרוא כאן.

המהמורות בהתקנה, ההתקנה לא הייתה חלקה לגמרי, ושווה להכיר את המהמורות מראש כדי לחסוך זמן. אציין אותן כאן בקצרה, בלי להיכנס לכל הפרטים הטכניים. הבעיה הראשונה: נתב חי בלי רשת. לאחר שמריצים את FAT על קובץ ה-Firmware, הכלי מחלץ את הקושחה, מזהה את הארכיטקטורה, בונה תמונת אמולציה ומתחיל להריץ אותה. האמולטור עולה, הקרנל מתחיל לרוץ, ואז מגיעה האכזבה. הנתב עולה, אבל כל ממשקי הרשת שלו מתים ואין לו שום כתובת IP.

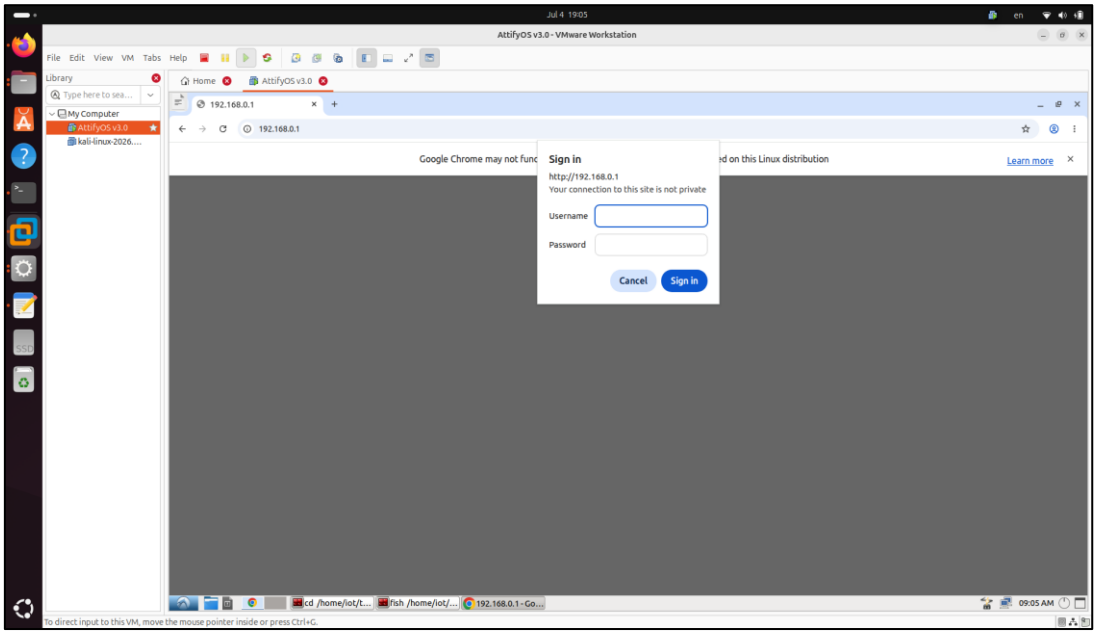
ללא רשת אין ממשק Web ואין אפשרות לתקשר עם הנתב. כל היתרון של האמולציה, היכולת לשלוח בקשות אל הנתב, פשוט נעלם.

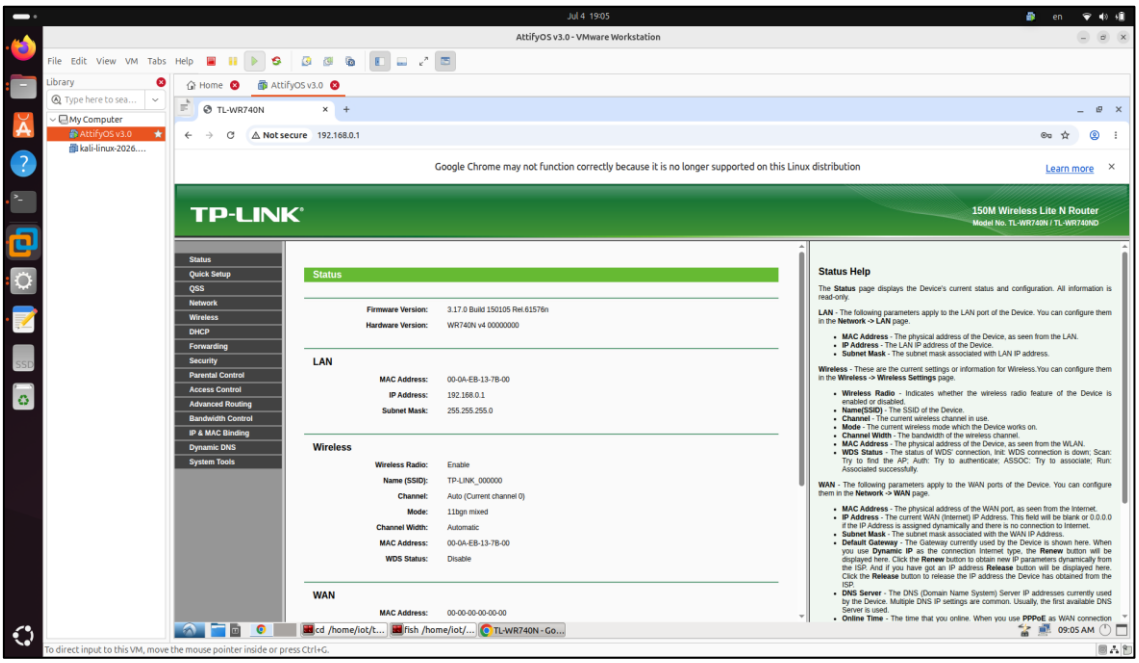
הפתרון לכך נקרא TAP. באמצעות יצירת ממשק TAP והגדרת כתובות IP הן במחשב המארח והן בנתב, ניתן לחבר ביניהם באותה רשת וירטואלית. לאחר שמקימים את החיבור ומגדירים את הכתובות בשני צדיו, מגיע הרגע שבו הכול מתחבר. מהמחשב המארח שולחים בדיקת חיות פשוטה אל כתובת הנתב ומקבלים תשובה. החיבור חי. פותחים דפדפן, ניגשים אל כתובת הנתב, ולפנינו נפרש ממשק הניהול המלא של הנתב, רץ כולו בתוך אמולציה, בלי שום חומרה מחוברת.

הבעיה השנייה: גם לאחר שהרשת פעלה, ממשק הניהול עדיין לא היה זמין. הסיבה לכך הייתה שסביבת האמולציה אינה כוללת את כל מנהלי ההתקנים (Drivers) הייעודיים של הנתב, ולכן חלק מממשקי החומרה שהקושחה ציפתה להם כלל לא התקיימו. כתוצאה מכך, שרת ה-Web של הנתב (httpd) קרס מיד עם עלייתו. לאחר ניתוח הקריסה וביצוע תיקון נקודתי (Binary Patch) לקובץ הבינארי, ניתן היה לעקוף את הקריאה הבעייתית ולאפשר לשרת להמשיך לפעול כרגיל.

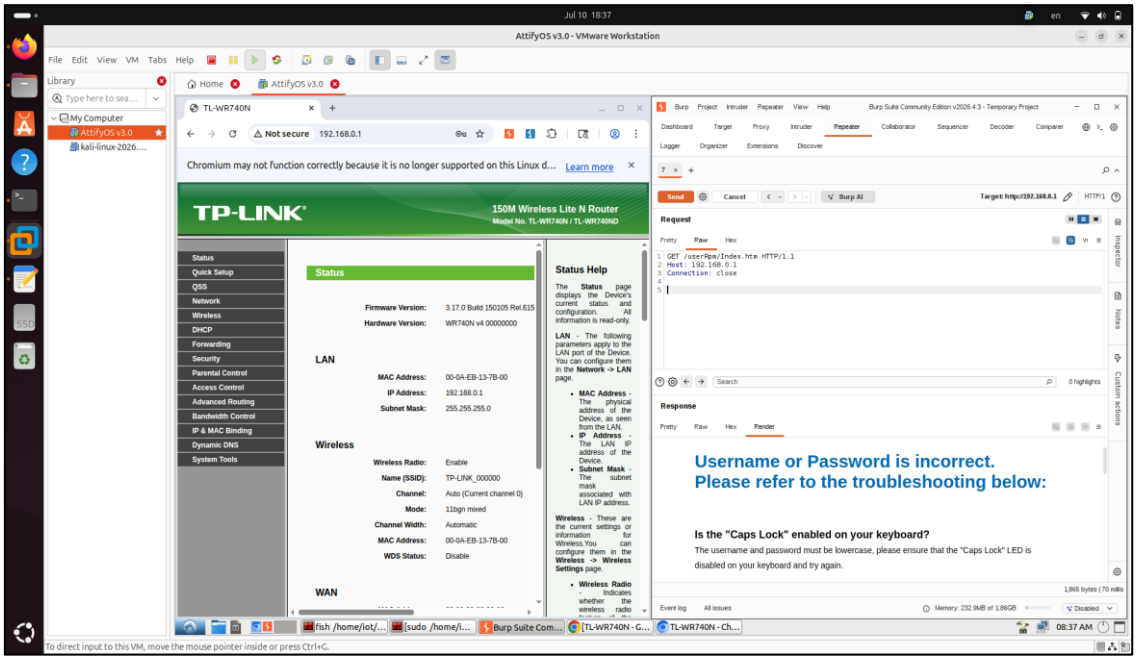
בכך השגנו משהו חזק מאוד: מעבדה חיה. נתב שלם שרץ על המחשב, שאפשר לשגר אליו בקשות בחופשיות ולחקור אותו בכלים מתקדמים, בלי לסכן שום חומרה. ובדיוק בסביבה הזו אימתנו את החולשה שתיארנו: שיגרנו את הבקשות הפוגעניות, ראינו את ממשק ה-Web מפסיק להגיב לחלוטין. הכול התרחש בסביבה וירטואלית ובטוחה, ללא צורך בנתב פיזי.

לפני הקריסה:

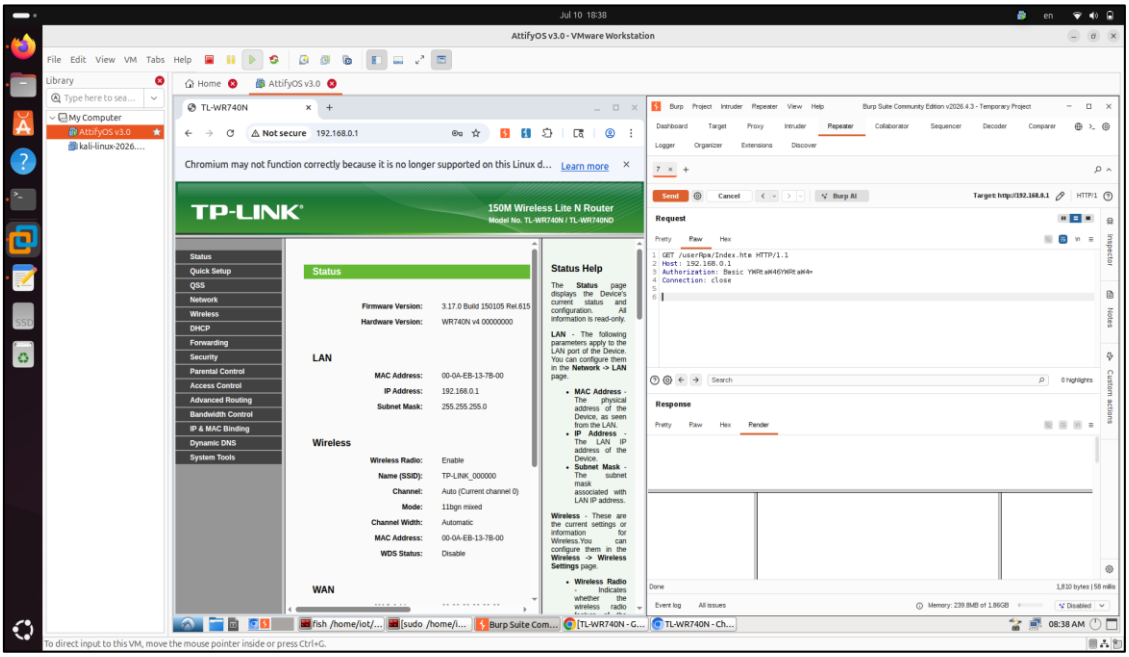




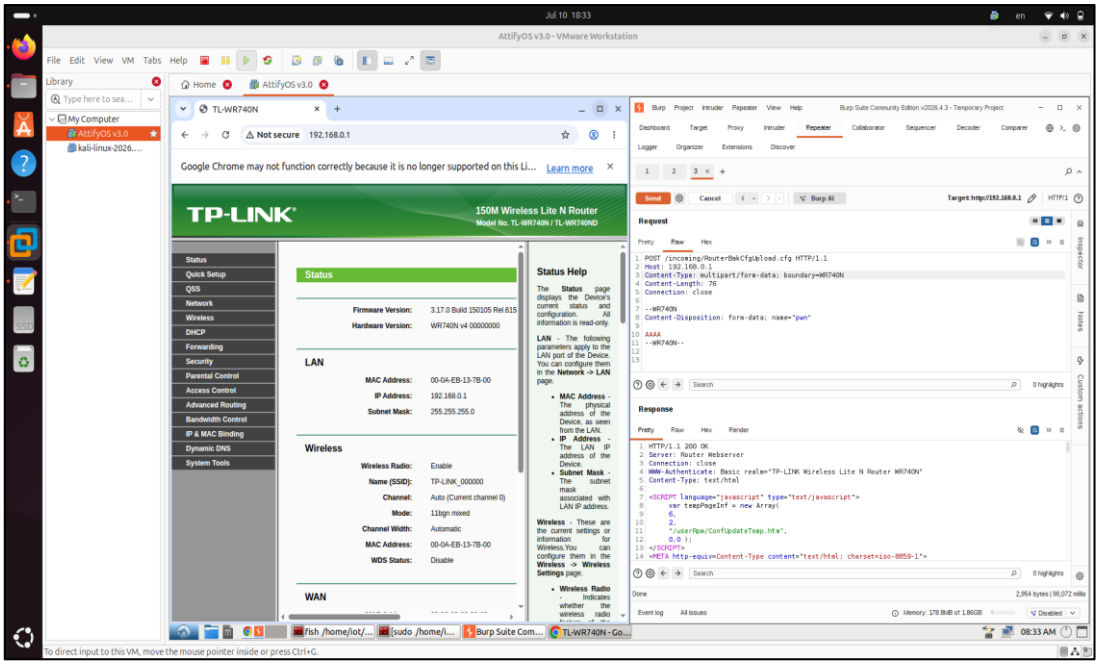
בקשה לגיטימית שמצריכה התחברות:



בקשה עם admin:admin וניתן לראות את התבנית של העמוד בצד ימין למטה:

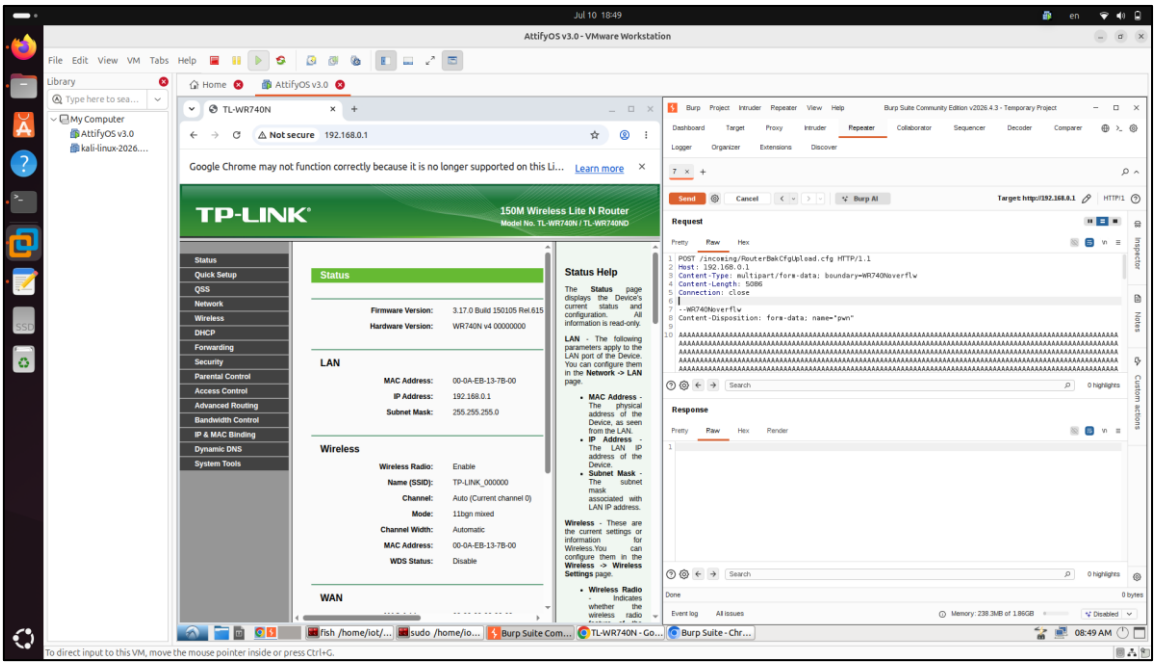


בקשת POST לממשק המדובר /incoming/RouterBakCfgUpload.cfg עם buffer קטן:

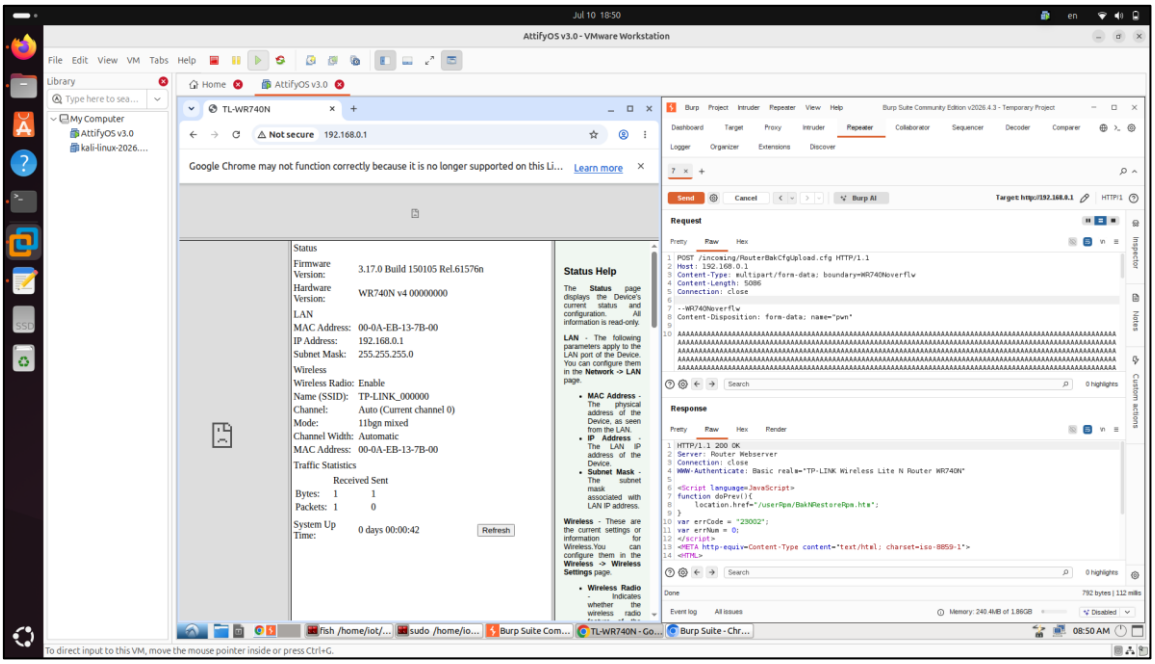


בקשת POST לממשק המדובר /incoming/RouterBakCfgUpload.cfg עם buffer גדול וניצול החולשה:

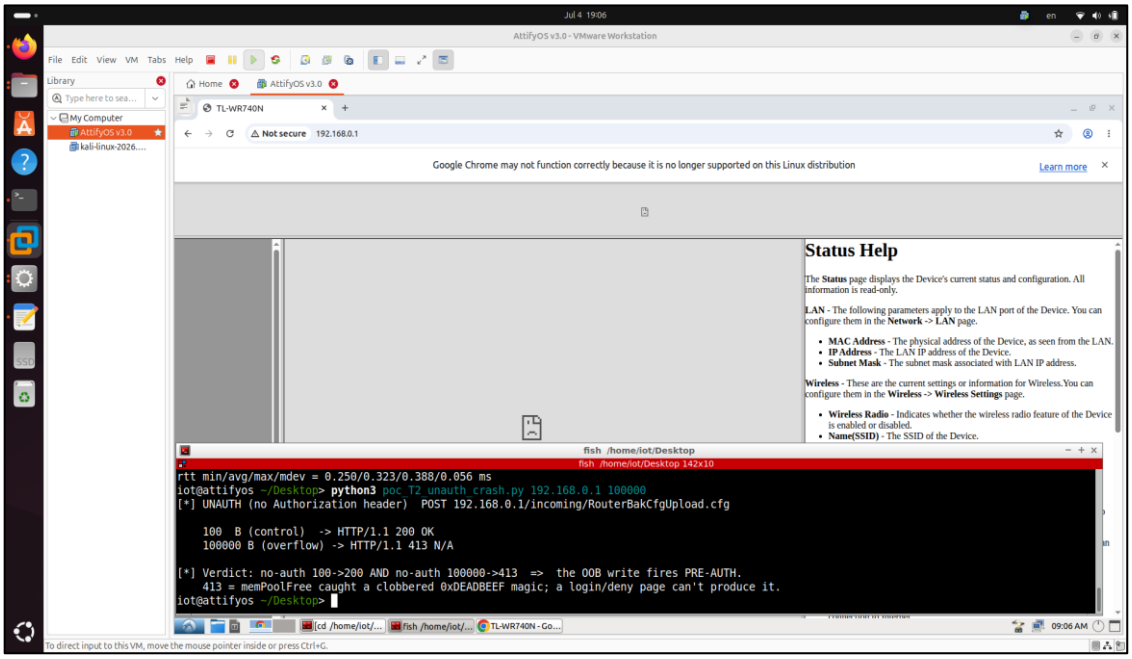
לפני:



אחרי:



POC שמדגים גם בקשה לגיטימית וגם בקשה זדונית וממשק הניהול אחרי הקריסה:



גם Refresh לעמוד לא יעזור ועדיין נקבל שגיאה.

סיכום

במאמר הזה ניסיתי להראות כיצד נראה מחקר אבטחה על נתב אמיתי, החל מהחומרה עצמה ועד לניתוח מעמיק של ה-Firmware והקוד שרץ עליו. התהליך עבר דרך זיהוי ממשק UART, חילוץ הקושחה ישירות משבב ה-Flash, פירוק וניתוח מערכת הקבצים, Reverse Engineering של שרת ה-Web עם הכלי החינמי Ghidra, מציאת חולשה ולבסוף הקמת סביבת אמולציה שאפשרה לאמת את הממצאים בצורה בטוחה וללא תלות בחומרה.

אחד המסרים המרכזיים של המחקר הוא שעולם ה-IoT דורש שילוב של מספר תחומי ידע. לעיתים צריך להחזיק מלחם ביד אחת ודיבאגר ביד השנייה. חולשות משמעותיות אינן תמיד נמצאות באמצעות סריקות אוטומטיות, אלה בזכות הבנה של החומרה, מערכת ההפעלה, מבנה ה-Firmware והקוד שרץ על המכשיר. השילוב בין כל התחומים הללו הוא שהופך מחקר מסוג זה למאתגר.

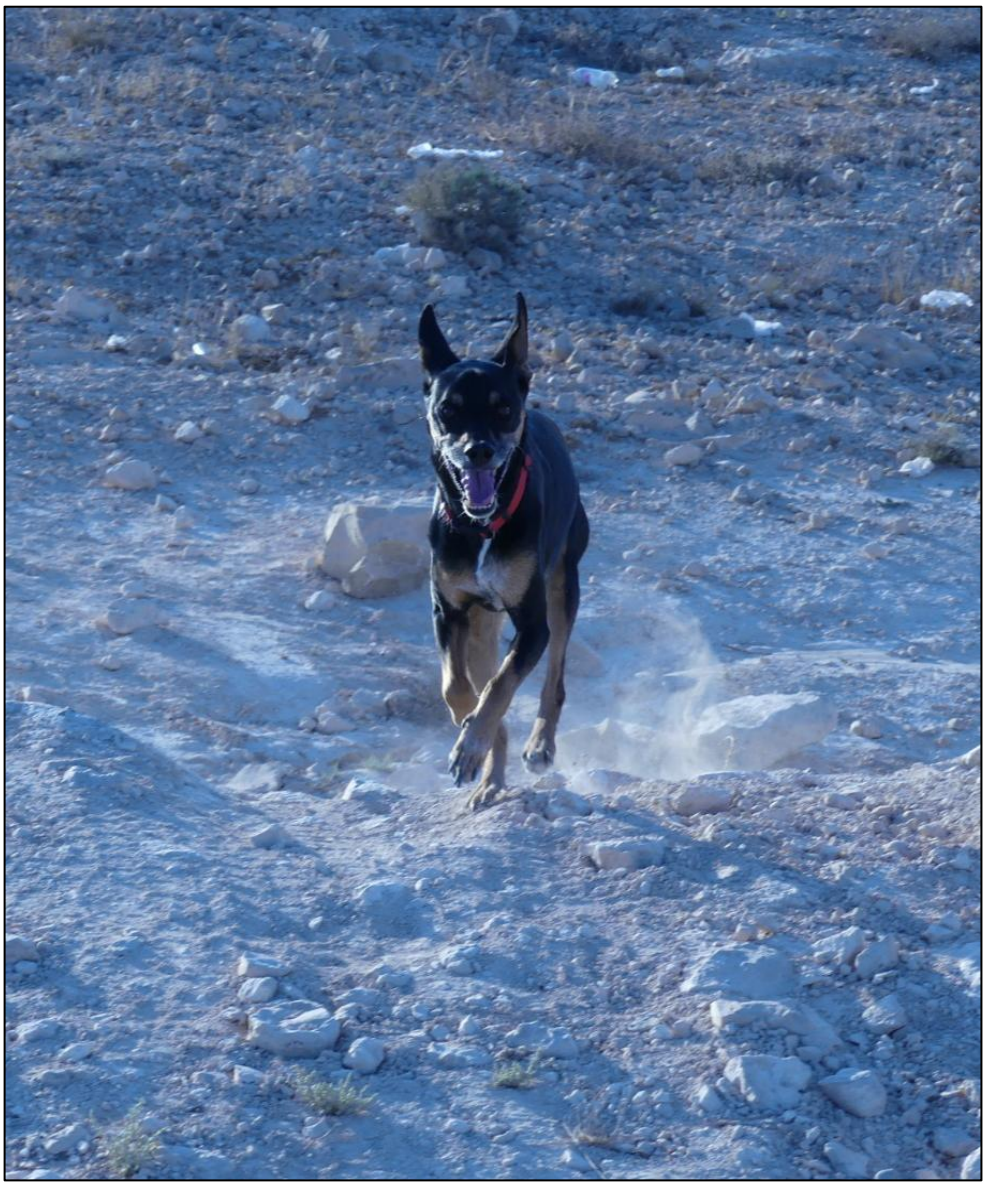
במהלך המחקר זוהתה חולשת אבטחה, ככל שהצלחתי לברר באמצעות חיפוש במקורות מידע ציבוריים, אינה מתועדת בפרסומים קודמים. עם זאת, ייתכן שקיים תיעוד שלא הצלחתי לאתר. אם מי מהקוראים מכיר פרסום המתעד את אותה חולשה לפני מועד פרסום מאמר זה, אשמח מאוד אם ישתף אותי באמצעות כתובת ה-Gmail המופיעה בסוף המאמר.

בנוסף, חשוב לציין כי מדובר במוצר אשר הגיע לסוף מחזור החיים שלו (End of Life). בעקבות כך, פניתי לחברה במטרה לשתף את הממצאים ולאפשר בחינה משותפת של החולשה שהתגלתה. עם זאת, החברה בחרה שלא לבצע תהליך בדיקה משותף ולא לחתום על קרדיט או אישור רשמי בנוגע לחולשה ולממצאים המתוארים במאמר זה.

בהתחשב בכך שלא ניתן היה לבצע תהליך אימות או תיאום פרסום מול היצרן, המחקר מתפרסם מתוך רצון לשתף ידע עם קהילת האבטחה ולהציג את תהליך העבודה כפי שבוצע בפועל. אני מקווה שהמאמר יספק לקוראים נקודת פתיחה מעשית לעולם מחקר ה-IoT, ויעודד מחקרי כחול לבן נוספים.

ברצוני להודות לדניאל דרן על הסיוע והייעוץ שנתן במהלך חלק מהמחקר. תרומתו סייעה בקידום מספר היבטים בתהליך המחקר, ועל כך תודתי.

לבסוף, אני מבקש להקדיש את המחקר הזה לזכרו של הכלב האהוב שלי, שמוליק, הוא ליווה אותי לאורך תקופת העבודה על המחקר, ובדרכו השקטה תמיד היה שם לצדי בשעות הארוכות מול שולחן העבודה. המחקר הזה מוקדש לזכרו.



על המחבר

- יהונתן שיאון
- אימייל: Yehonatan.sion@gmail.com
- לינקדין: www.linkedin.com/in/yehonatan-sion-29213b334
- מוסמך +OSCP, פתוח למחקר משותף.

מקורות מידע

- קורס מבוא של TCM Security המלמד Beginner's Guide to IoT and Hardware Hacking – באופן מעשי את יסודות פריצת התקני IoT, חומרה, Firmware ו-Reverse Engineering - קורס שביצעתי ונתן לי בסיס טוב לכל המחקר שביצעתי. Beginner's Guide to IoT and Hardware Hacking - TCM Security
- קישור למערכת ההפעלה AttifyOS adi0x90/attifyos: Attify OS - Distro for pentesting IoT devices
- קישור לתת מאמר שכתבתי במחקר שלי על נתב WR740N. J4c0b-1337x007/TL-WR740N-v4.28-hardware-research: Security research on the TP-Link TL-WR740N v4 (4.28) — hardware teardown, firmware extraction, and vulnerability analysis..
- קישור לתת מאמר שכתבתי במחקר שלי על נתב WR841N. J4c0b-1337x007/TL-WR841N-v12-hardware-research: Hardware analysis and UART access research on TP-Link TL-WR841N v12
- קישור למאמר של חברי ניר גילס וארד דוננפלד - מספרים בהרחבה על Buildroot.Digital Whisper :: הגליון המאה ושמונים ושניים של DigitalWhisper שוחרר!
- עמוד ייעודי עבור הנתב WR740N של OpenWrt [OpenWrt Wiki] TP-Link TL-WR740N
- מחקר על מתקפת VPNFilter VPNFilter Malware - Critical Update
- מחקר על מתקפת Cherry Blossom WikiLeaks - Vault 7: Projects
- קישור להורדה של הFirmware של נתב WR740N Download for TL-WR740N | TP-Link
- קישור למקום שמצאתי בו פיצוח של הסיסמאות למשתמשים: OpenWrt Forum Archive