



הצייד והמכונה: גילוי חולשות ו-Bug Bounty בעזרת AI Agents

מאת שמוליק כהן

הקדמה

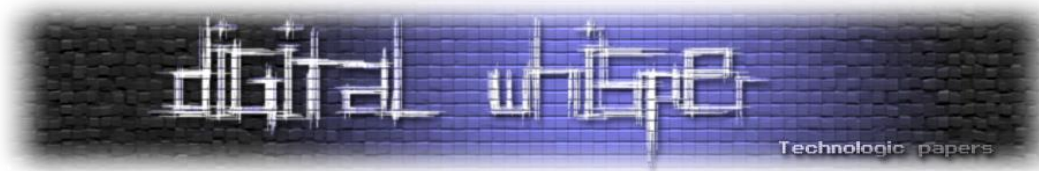
כשהתחלתי לעבוד ב-APIiro כ-API Engineer, התעסקתי בעיקר באיך למנוע ולנתח בעיות אבטחה. במקביל שמעתי לא מעט דיבורים על איך כלי AI יכולים לעשות חיים קלים לחוקרי חולשות, ופשוט רציתי לראות אם זה עובד גם בפועל.

לפני כמה חודשים פתחתי תיקייה חדשה בשם `first_cve` והתיישבתי לחפש את החולשה הראשונה שלי. נעזרתי ב-Agent כדי לסרוק את הקוד, לבדוק מחקרים קודמים ולבנות בדיקת ניצול (PoC). תוך כמה שעות מצאתי חולשה אמיתית ודיווחתי עליה. מאז המשכתי עם אותה שיטה כדי לחקור ולדווח על עוד חולשות.

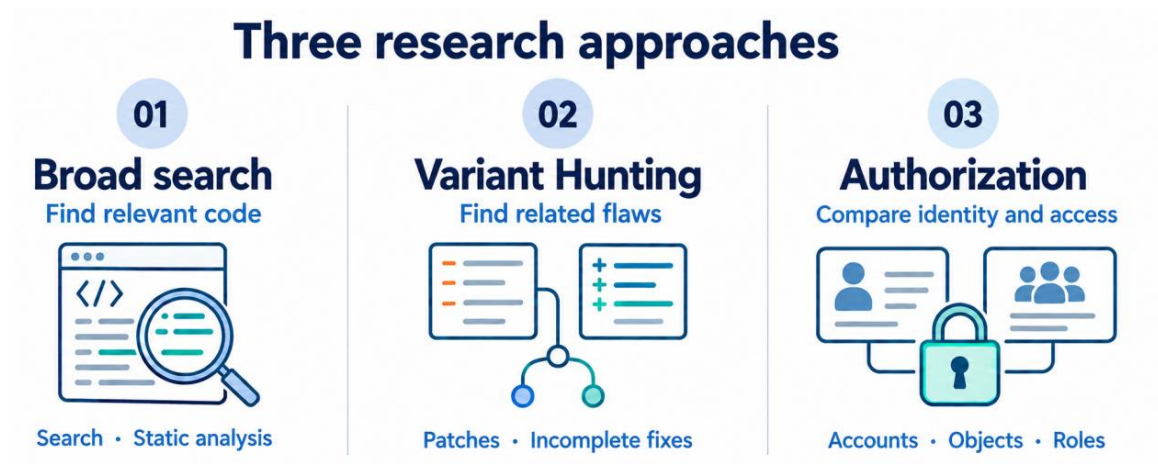
במאמר הזה אשתף איך אני עובד עם AI Agents כדי למצוא חולשות, גם בקוד פתוח וגם בתוכניות Bug Bounty, ואיך אפשר להתחיל לעשות את זה בעצמכם.

כל השיטות שאציג כאן מתבססות על שלושה דפוסי עבודה עיקריים:

1. **חיפוש וסריקה רחבים (Broad Sweeps):** איסוף כיוונים ראשוניים מחיפוש ב-GitHub ומסריקות CodeQL או Semgrep/Openpgrep. אפשר גם להתמקד בקבוצת מוצרים מסוימת, כמו שרתי MCP.
 2. **חקר תיקונים וציד וריאנטים (Variant Hunting):** הבנה של תיקון אבטחה ישן ובדיקה אם אותה הבעיה קיימת גם במקומות אחרים בקוד או במוצרים דומים.
 3. **בדיקות הרשאות:** בדיקה שיטתית של מה משתמשים עם חשבונות או תפקידים שונים מורשים לראות ולשנות, ומה נשבר כשמשנים תפקיד או בעלות על אובייקט.
- הדפוסים האלה משתלבים כל הזמן: חיפוש רחב יכול להוביל לתיקון ישן, ובדיקה של התיקון מעלה לא פעם שאלה על בדיקת הרשאות.



חלק מהממצאים שלי הובילו להודעות אבטחה רשמיות ולתגמולים כספיים (Bounties), ואחרים הסתיימו רק בתיקון של הקוד. לצד הדוגמאות אסביר איך אני מסנן רעש, מזהה טענות לא מבוססות ודיווחי AI באיכות נמוכה (AI Slop), ומחליט מה באמת שווה דיווח.



[תרשים 1: חיפוש רחב, חקר תיקונים ובדיקות הרשאות]

מונחים שכדאי להכיר לפני שמתחילים

כדי שנדבר באותה שפה, הנה כמה מונחי מפתח שילוו אותנו לאורך המאמר:

- **CVE (Common Vulnerabilities and Exposures)**: מזהה מוסכם לחולשת אבטחה, בפורמט CVE-YEAR-NUMBER. הוא מאפשר למתחזקים, לחוקרים ולכלי אבטחה להתייחס לאותה חולשה. חשוב לזכור: קבלת מספר CVE ופרסום המידע עליו בפועל הם שני שלבים נפרדים.
- **CNA (CVE Numbering Authority)**: ארגון או גוף שמוסמך לחלק מזהי CVE בתחום האחריות שלו (למשל GitHub, 'יצרניות תוכנה גדולות או פרויקטי קוד פתוח מובילים).
- **GHSA (GitHub Security Advisory)**: מנגנון הודעות האבטחה של GitHub. הוא מאפשר לדווח למתחזקים ולתאם תיקון באופן פרטי, לבקש מזהה CVE, ורק אז לפרסם את הפרטים לציבור. מזהה GHSA ומזהה CVE הם לא אותו הדבר, אם כי בדרך כלל הם מקושרים זה לזה.
- **Taint Analysis (מעקב אחר זרימת קלט)**: שיטת ניתוח קוד שבה עוקבים אחרי קלט שמגיע ממשתמש (Source) לאורך הדרך, כדי לראות אם הוא מגיע לפונקציה רגישה (Sink) בלי לעבור ניקוי או הגנה מתאימה (Sanitizer). אם קלט מגיע ל-Sink בלי הגנה, זה כיוון ששווה לבדוק לעומק - לא הוכחה אוטומטית שיש שם חולשה.
- **Bug Bounty**: תוכנית שבה חברה או ארגון מתגמלים כספית חוקרים שמדווחים על חולשות לפי הכללים שהוגדרו מראש. קבלת הדוח, תיקון הקוד, קבלת התשלום והקצאת CVE הם דברים שונים לחלוטין - אחד מהם לא מבטיח אוטומטית את האחרים.



- **IDOR ו-BOLA:** מונחים קרובים לכשלי הרשאות ברמת האובייקט: המערכת אינה מוודאת כראוי שמותר למבקש לגשת לאובייקט הספציפי. השורה התחתונה פשוטה: העובדה שמשתמש יודע את ה-ID של מסמך או רשומה לא אומרת שמותר לו לקרוא או לערוך אותם.
- **סביבת העבודה שלי:** רוב העבודה שלי נעשית על המק האישי, עם Claude ו-Codex ועם Skills שאני בונה ומשפר לאורך הדרך. ה-Skills מרכזים הנחיות למשימות חוזרות, כמו בדיקת פרסומים קודמים, אימות טענות ותיעוד ראיות, ולעיתים גם סקריפטים וכלי עזר.
- לדוגמה, ה-Skill בשם published-dedup בודק אם ממצא כבר פורסם. תמצית ההנחיות שלו היא: "השווה את הממצא להודעות אבטחה ולפרסומים קודמים של הפרויקט, צרף מקורות למסקנה, וציין במפורש אם אין מספיק מידע להכריע. היעדר תוצאה בחיפוש אינו הוכחה שהממצא חדש". כך בקשה כללית כמו "בדוק כפילויות" הופכת למשימה עם דרישות ברורות לתוצאה.
- את המעקב אני מנהל ב-Obsidian: לכל ממצא משמעותי אני שומר את התיאור, הראיות, מצב הדיווח והתשובות מהמתחזקים, וגם מתעד מדוע כיוונים אחרים נפסלו. כך העבודה לא נשארת רק בהיסטוריית השיחות. לא התחלתי עם כל הכלים האלה מוכנים מראש, הסביבה התפתחה לפי הצרכים שעלו במחקר.

חיפוש וסריקה רחבים (Broad Sweeps)

- חיפוש רחב מתחיל ממיקוד בקבוצת פרויקטים, בטכנולוגיה מסוימת או ברכיב שמעניין אותנו לבדוק.
- בסקירה שעשיתי על שרתי MCP, למשל, נקודת המוצא הייתה לבדוק מימושים שונים של אותו פרוטוקול. MCP (Model Context Protocol) מחבר יישומי AI לכלים ולמקורות מידע. כל שרת מממש פעולות אחרות לגמרי, כך שעצם העובדה שכולם משתמשים באותו פרוטוקול לא אומרת שהם מתנהגים אותו הדבר.
- השלב הראשון הוא איסוף חומרים:
- **חיפוש ב-GitHub:** איתור ריפוזיטוריים רלוונטיים, מעבר על Issues, Pull Requests והודעות אבטחה ישנות כדי להבין איפה היו בעיות בעבר.
 - **Semgrep ו-Opengrep:** בדיקה מהירה של קוד לפי כללים ותבניות (Pattern matching).
 - **CodeQL:** כלי לבניית בסיס נתונים של הקוד, שמאפשר ניתוח עמוק של זרימת נתונים (Data flow) בין מודולים ופונקציות שונות.



מתוצאות חיפוש למועמדים לבדיקה

תוצאות חיפוש או התראה מכלי סריקה הן בסך הכל המלצה איפה להתחיל לקרוא - ממש לא הוכחה לחולשה. בסקירות שלי לא מעט כיוונים נפסלו מהר: לפעמים בגלל בדיקת תקינות מוקדמת שחסמה את הגישה, לפעמים כי הקוד שימש לבדיקות בלבד (Mock/Test), ולפעמים כי ההתנהגות הייתה תקינה ותאמה את ההרשאות. מועמדים שלא הגעתי לבדוק פשוט נשארו לא מאומתים, ולא ספרתי אותם כהתראות שווא.

כאן גם נכנס ה-Agent לתמונה. אני מחבר אותו לכלי סריקה. ה-Agent מקבל את התוצאות, מרכז אותן וקורא את הקוד סביבן. את תוצאות הסריקה אפשר לשמור בפורמט SARIF (פורמט סטנדרטי לדוחות ניתוח סטטי) ובקובצי CSV.

ה-Agent מקבל מספר מוגבל של מועמדים לבדיקה, במסגרת זמן ותקציב טוקנים מוגדרים, ובוחן אותם לפי שלוש שאלות פשוטות:

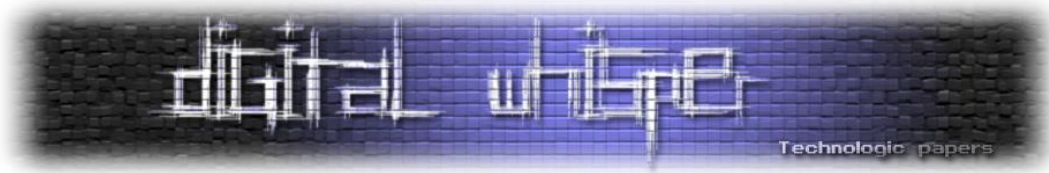
1. **מי שולט בקלט ואילו הרשאות צריך?** משתמש מחובר יכול גם הוא לחצות גבולות הרשאה; ממצא לא נפסל רק בגלל שהוא דורש התחברות לחשבון.
2. **אילו הגנות קיימות בדרך?** צריך לבדוק את כל המסלול, כולל שכבות ביניים (Middleware) ותנאים מוקדמים לפני הפונקציה הרגישה.
3. **האם יש השפעה אמיתית בשטח?** חייבים להבדיל בין קוד שנראה חשוד על הנייר לבין פגיעה שהוכחה בפועל.

מקרה בוחן: Polynomial ReDoS ב- Parse Server

במהלך סריקה של Parse Server (תשתית Backend פופולרית בקוד פתוח), קפצה התראת CodeQL בשם `js/polynomial-redos`. ה-Agent קרא את הקוד סביב ההתראה, עקב אחרי מסלול הבקשה ובנה בדיקה כדי לאמת את הבעיה. CodeQL נתן את הכיוון - וה-Agent ביצע את הבדיקה הטכנית שאימתה אותו.

חולשת ReDoS (Regular Expression Denial of Service) מתרחשת כשהתאמה של ביטוי רגולרי לוקחת זמן לא פרופורציונלי לאורך הקלט. במנועי Regex שמבוססים על Backtracking, זמן הריצה עלול לזנק ולהפוך לפולינומי או מעריכי.

הרבה מפתחים חושבים שזה קורה רק עם חזרות מקוננות (כמו `((a+)+)`), אבל זה לא תנאי הכרחי: גם חיפוש התאמה ממיקומי התחלה רבים עשוי להוביל לזמן ריצה פולינומי, בהתאם למבנה הביטוי ולקלט. ב-Node.js, עבודה סינכרונית כבדה על התהליכון הראשי חוסמת את לולאת האירועים (Event Loop), ולכן עלולה לעכב בקשות אחרות ולפגוע בזמינות השירות.



הממצא ב- Parse Server יושב בפענוח גרסת הלקוח מתוך בקשת ה-HTTP. הביטוי הבעייתי שהוסר בתיקון היה:

```
const versionRE = /([-a-zA-Z]+)([0-9\.]+)/;
```

החלק הקריטי במחקר לא היה רק מציאת הביטוי, אלא ההבנה איפה הוא פועל במסלול הבקשה: לפי הודעת האבטחה, הפענוח הזה קרה **לפני** אימות המשתמש (Authentication) ו**לפני** הגבלת קצב הבקשות (Rate Limiting). המשמעות היא פגיעה אפשרית בזמינות השירות גם ללא חשבון משתמש, לפני שההגנות האלה נכנסות לפעולה.

הממצא **פורסם כ-CVE-2026-47138 בהודעת האבטחה של Parse Server**, ובתיקון הוסר מנגנון הפענוח. במקרה הזה, CodeQL הצביע על הנקודה החשודה, וקריאת מסלול הבקשה המלא יחד עם ה-Agent הסבירה למה מדובר בבעיה אמיתית.

חקר תיקונים וציד וריאנטים

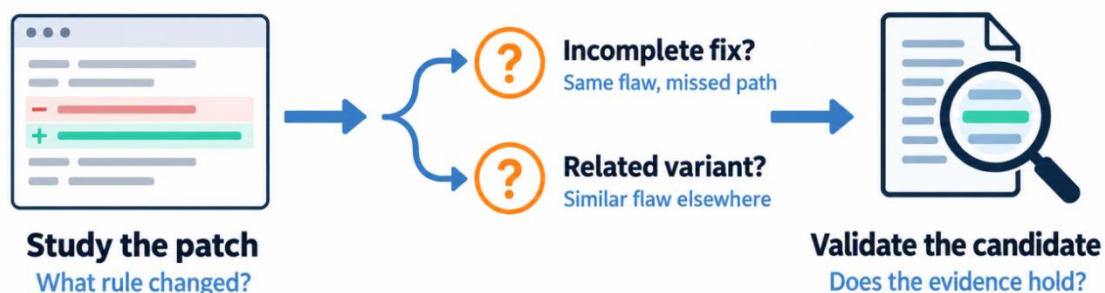
אחת הדרכים היעילות למצוא חולשות היא להתחיל דווקא מתיקון אבטחה שכבר פורסם. ה-Diff מציג בדיוק מה השתנה, אבל כדי להבין את התמונה המלאה צריך לקרוא גם את הקוד שמסביב ואת תיאור הבעיה המקורית.

משם עולה השאלה המרכזית: **אם ההגנה הזו הייתה חסרה כאן - איפה עוד עשו את אותה הטעות?**

בציד וריאנטים אנחנו מחפשים מופעים נוספים של אותו דפוס כשל:

- **בתוך אותו הפרויקט:** בדיקה אם התיקון היה חלקי (Incomplete Fix), או אם נתיבים ומודולים מקבילים סובלים מאותה בעיה.
- **בין פרויקטים שונים (Cross-Project):** דפוס שגוי שנמצא בספרייה, ב-Framework או בארכיטקטורה מסוימת מופיע לעיתים קרובות בפרויקטים נוספים שמימשו רעיון דומה או העתיקו גישה דומה.

Variant Hunting



[תרשים 2: ציד וריאנטים ובחינת תיקונים חלקיים]



מקרה בוחן 1: pyLoad והרשימה המתוחזקת ידנית

ב-pyLoad (מנהל הורדות פופולרי ב-Python), רשימה ידנית בקוד הגדירה אילו הגדרות מערכת מותר לשנות רק למשתמש מנהל (Admin). ה-Agent עבר איתי על תיקונים קודמים, סרק את כלל ההגדרות הקיימות בפרויקט והשווה אותן לרשימה. במקום לפתוח ולקרוא עשרות קבצים ידנית, המודל מיפה במהירות את הפערים והכין בדיקות אימות.

ההגבלה נועדה למשתמשים רגילים עם הרשאת **SETTINGS** בלבד. הלוגיקה נראתה בערך כך:

```
ADMIN_ONLY_CORE_OPTIONS = {
    ("general", "storage_folder"),
    ("log", "syslog_host"), ("log", "syslog_port"),
    ("proxy", "password"), ("proxy", "username"),
    ("reconnect", "script"),
    ("webui", "host"),
    ("webui", "ssl_certfile"), ("webui", "ssl_keyfile"),
    ("webui", "use_ssl"),
}

if (category, option) in ADMIN_ONLY_CORE_OPTIONS and not is_admin:
    self.pyload.log.error("Permission denied")
    return
self.pyload.config.set(category, option, value)
```

הבעיה ברשימות סגורות שמתוחזקות ידנית (Allowlist/Blocklist) היא שקל מאוד לשכוח משהו בחוץ. השוואת ההגדרות בפועל מול הרשימה העלתה שני ממצאים שבהם תיקוני העבר היו חלקיים:

1. **שינוי הגדרות Proxy ללא הרשאת מנהל (CVE-2026-42313):** חלק מהגדרות ה-Proxy נעלו, אבל אחרות נשכחו - מה שאפשר למשתמש להפנות את כל תעבורת ההורדות דרך שרת Proxy בשליטתו.
2. **ביטול אימות תעודות (CVE-2026-42312) TLS:** ההגדרה שקובעת אם לאמת תעודות SSL/TLS נשמטה, מה שאפשר לבטל הגנה קריטית על חיבורים יוצאים.

הודעות האבטחה על [שינוי הגדרות ה-Proxy](#) ועל [ביטול אימות תעודות TLS](#) מתארות תיקונים קודמים שלא כיסו את כל ההגדרות הרגישות. הן מפנות לארבעה מזהי CVE קודמים. במקרה הזה נדרשה בדיקה של כלל ההגדרות מול מדיניות ההרשאות.



מקרה בוחן 2: n8n וההבדל בין מזהים לערכים ב-SQL

ב-n8n (פלטפורמה מובילה לאוטומציה), תיקון קודם ברכיב MySQL v1 טיפל בהזרקות דרך שמות של טבלאות ועמודות. נעזרתי ב-Agent כדי לנתח את השינוי ולבדוק האם הפתרון מכסה גם ערכים שמגיעים מביטויים (Expressions) בתהליך העבודה.

במסדי נתונים יש הבדל יסודי:

- **Prepared Statements (פרמטריזציה):** מגנים על ערכים (Values), ולא על שמות של טבלאות או עמודות (Identifiers).
- **מזהים דינמיים:** דורשים הגבלה לרשימה מותרת או שימוש במנגנון ציטוט ייעודי למזהים של מסד הנתונים.

תיקון שמטפל בשמות עמודות לא מגן כשלעצמו על ערכים שמשורשרים ישירות לתוך השאילתה. בבדיקת הפונקציה `executeQuery`, גילינו שערכי ביטויים אכן שורשרו לשאילתה כמחרוזות פשוטות ללא פרמטריזציה.

ההשפעה הייתה תלויה בתנאים ספציפיים (שימוש ברכיב הישן, בפעולה הזו, ובתהליך שבו קלט חיצוני מזין את הביטויים - רכיב MySQL v2 החדש לא הושפע). הממצא אושר, פורסם כ- CVE-2026-59257 (GHSA-hwmj-qg4v-cvg9) ותוגמל ב-Bounty.

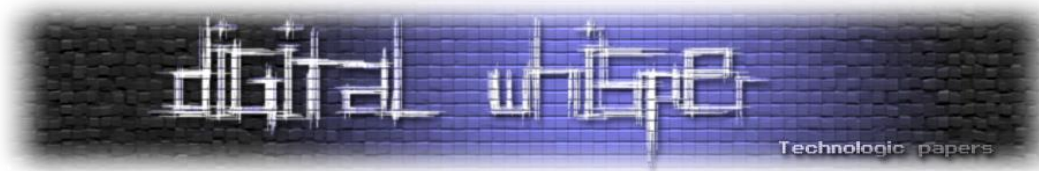
זהו מקרה קלאסי שבו הבנת הגבולות של תיקון מסוים מאפשרת למצוא וריאנט שלא כוסה.

בדיקות הרשאות בין חשבונות

בתוכניות Bug Bounty בודקים הרבה פעמים יישומי Web ושירותי ענן, לעיתים קרובות בלי גישה לקוד המקור (Blackbox). אחד הכיוונים הכי מתגמלים לבדיקה הוא כשלי הרשאות: האם משתמש רגיל יכול לקרוא מידע או לבצע פעולה ששייכת למישהו אחר? כשלי לוגיקה עסקית הם עולם רחב מאוד, שלא מסתכם רק ב-IDOR או BOLA קלאסיים.

כדי לבדוק הרשאות בצורה שיטתית, אני מגדיר שלושה דברים: מי המשתמש, לאיזה חשבון שייך האובייקט, ומה הפעולה המבוקשת. הדרך הכי פשוטה ואמינה לעשות את זה היא בעזרת שני חשבונות בדיקה בשליטתי (The Two-Account Matrix):

- **חשבון א' (התוקף):** מנסה לגשת למשאב
- **חשבון ב' (הקורבן):** המשאב שייך לו



ההשוואה הזו מאפשרת לראות מיד מה נחסם כמו שצריך ומה דולף. אם הבדיקה כוללת שינוי תפקיד (למשל קידום למנהל) או העברת בעלות, אני מתעד את ההתנהגות לפני השינוי ואחריו.

Two accounts, one access policy

	Object A	Object B	
Account A	 Baseline	 Check policy	 Inspect returned data  Compare state changes
Account B	 Check policy	 Baseline	

HTTP status alone is not enough

(תרשים 3: מטריצת בדיקות הרשאות באמצעות שני חשבונות (The Two-Account Matrix))

מקרה בוחן: כשל הרשאות בין סוכנויות ב-Revive Adserver

במהלך מחקר על Revive Adserver (מערכת ניהול פרסומות בקוד פתוח, דרך תוכנית ה-Bug Bounty שלה ב-HackerOne), ה-Agent ניתח את מנגנון בדיקת ההרשאות בקובץ:

`www/admin/website-index.php`

המערכת בנויה לעבודה עם מספר סוכנויות במקביל (Multi-Agency Tenants), כאשר כל מנהל סוכנות אמור לראות אך ורק את האתרים והקמפיינים של הסוכנות שלו. בקובץ הקונפיגורציה הראשי הופיע הקטע הבא:

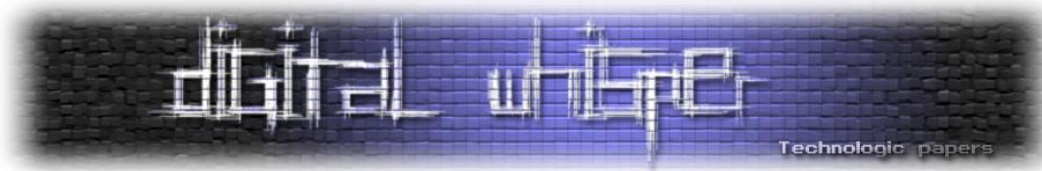
```
// www/admin/config.php
phpAds_registerGlobalUnslashed('agencyid');

// ערך ברירת המחדל נקבע רק כאשר הפרמטר אינו קיים בבקשה
if (!isset($agencyid)) {
```

המשמעות: אם המשתמש סיפק בבקשה פרמטר `agencyid` משלו, המערכת השתמשה בו ישירות ולא דרסה אותו עם ה-ID של הסוכנות שאליה הוא באמת שייך. בדף שמציג את רשימת האתרים, הבדיקה נראתה כך:

```
// www/admin/website-index.php
// ללא שום בדיקה לאיזו סוכנות מותר לו לגשת - בדיקת סוג החשבון
OA_Permission::enforceAccount(OA_ACCOUNT_MANAGER);
$aWebsitesZones = $dalAffiliates->getWebsitesAndZonesByAgencyId($agencyid,
```

כאן בדיוק ישב הכשל: הפונקציה `enforceAccount` בדקה רק את **סוג החשבון** (האם המשתמש הוא מנהל סוכנות באופן כללי), אבל לא בדקה האם יש לו הרשאה ל-`agencyid` הספציפי שהוא שלח בבקשה.



מה שחזק את ההשערה הייתה השוואה מול דפים אחרים במערכת: בעמודים כמו `agency-edit.php` או `agency-access.php`, המתכנתים דווקא הוסיפו בדיקת הרשאה מפורשת לאובייקט. הפער הזה בין הדפים קפץ מיד לעין.

הרמתי סביבת Docker מקומית עם שני חשבונות נפרדים של מנהלי סוכנויות, ואימתתי שמשתמש אחד יכול לקרוא את רשימת האתרים של הסוכנות השנייה פשוט על ידי שינוי הפרמטר. הגשתי את הדוח יחד עם הצעת תיקון. הדיווח התקבל, החולשה תוקנה בגרסה הבאה, אך לא הוענק עליה תגמול כספי.

סינון ואימות לפני הדיווח

כדי לבסס את הדיווחים על ראיות, ריכזתי את תהליך העבודה בכלים וב-Skills:

Grepnir: כלי שפיתחתי שמשלב חיפוש קוד ב-grep.app, מידע מ-GitHub, סריקות Semgrep ומיון מועמדים בעזרת Agents. הוא יודע לקלוט גם דוחות SARIF מכלי סריקה חיצוניים, ולתעד במדויק אילו תוצאות נפסלו ואילו דורשות בדיקה נוספת.

CVE-Hunt: מרכז מקורות מידע וסריקה רחבה.

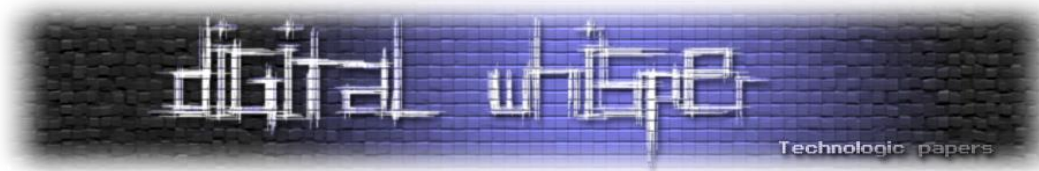
CVE-Toolkit: מרכז בדיקות ממוקדות - משלבי האימות הטכני ועד לניסוח הדוח הסופי.

From candidate to report



Stop or revise when evidence fails

[תרשים 4: מסיון מועמדים לאימות ולדיווח]



הגנה מפני דיווחי סרק (AI Slop)

פעם נתתי ל-Codex לבדוק 34 מועמדים ל-XSS, ובטעות תיארתי לו מראש איזו חולשה אני מצפה למצוא. הוא הכריז על מועמד אחד כ-HIT. אבל בבדיקה חוזרת גיליתי שהקלט רק מצא רכיב שכבר היה בדף; הוא לא יצר תוכן חדש ולא הפעיל קוד. הרמז שלי הפך תשובה שגויה למשכנעת. בעקבות זאת הוספתי ל-Skill cve-hunt שאלה פשוטה: האם הקלט באמת יכול ליצור תוכן חדש בדף, או שהוא רק מחפש משהו שכבר קיים?

כדי למנוע את זה, ב-cve-toolkit חילקתי את התהליך לשלבים נפרדים, כשכל שלב מופקד בידי Skill ייעודי:

- vuln-verify: אחראי אך ורק על אימות הטענה הטכנית ובניית PoC.
- finding-judge: משמש כביקורת עמיתים קשוחה שבודקת את הממצא בעיניים ביקורתיות.

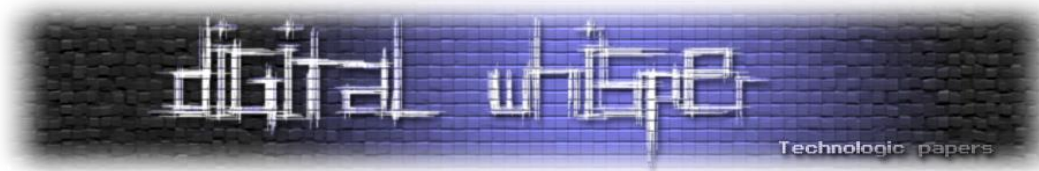
בדיקת פרסומים קודמים מתחילה עוד לפני האימות המעמיק. אחרי שני השלבים האלה מעדכנים אותה, בוחרים סיווג CWE ומחשבים ציון חומרה (CVSS). אם מתגלה כפילות או פער בראיות - עוצרים מיד או חוזרים לשולחן הבדיקות.

שני העקרונות שמובילים את האימות:

1. **מה הוכח בפועל מול מה שרק נראה חשוד:** בדיקה אמיתית חייבת לזהות גרסה מדויקת, תנאים מוקדמים, מצב התחלתי ותוצאה שנצפתה בפועל. היא חייבת לכלול גם מקרה השוואה (ביקורת), שבו הפעולה הלא מורשית אמורה להיחסם. בדיקה של פונקציה מבודדת בקוד לא מוכיחה לבדה שאפשר להגיע אליה דרך הממשק או ה-API של המוצר. גם אם המועמד סומן בסינון הראשוני כ-HIT, זה רק אומר שהוא נראה מבוסס בקוד - לא שהדוח מוכן להגשה.
2. **מה באמת מותר לטעון בדוח:** היכולת הטכנית שנצפתה, הדרך להגיע אליה וההשפעה שלה נבדקות בנפרד. התנהגות שנראית מסוכנת או חריגה היא לא בהכרח חולשה אם המוצר תוכנן לפעול כך במכוון (By Design). כשאני נעזר במודל נוסף בשלב הזה, התפקיד שלו הוא לשמש כפרקליט השטן ולנסות לערער על הטענות בעזרת העובדות, ולא להצביע אוטומטית בעד הדוח.

מה אני עושה לאורך המחקר

האמת הפשוטה היא שבמחקרים האלה כמעט שלא קראתי את קוד המקור בעצמי. ה-Agent אפשר לי לסרוק קוד בהיקפים עצומים, לעבור על מחקרים ודיווחים ישנים, ולבנות את ה-PoC שמדגים את הניצול. התפקיד שלי היה לנווט: לכוון את החיפוש, לבחון את ההסברים והתוצאות, ולהחליט מתי להעמיק, לשנות כיוון או לעצור. ה-Skills והכלים מסדרים את התהליך, אבל לא מחליפים את ההחלטות האלה.



לדעת מה לשאול

הבנה בסיסית של עולם המושגים היא קריטית כדי לנסח שאלות מדויקות. ב-Revive ההבדל המהותי היה בין סוג חשבון להרשאה לאובייקט; ב-n8, בין מזהה SQL לערך מוזרק. ה-Agent יכול להסביר כל מונח כזה, אבל בלי להבין את ההבדל ביניהם קשה מאוד לקלוט אם התשובה שהוא מחזיר רלוונטית או מומצאת.

לבחור מה לבדוק

פיצ'ר חדש, ארכיטקטורה לא שגרתית או חיבור בין שני רכיבים שונים יכולים להצדיק בדיקה נוספת - גם אם המוצר כבר נסרק בעבר. ב-NanoClaw, למשל, עניין אותי להבין אילו פעולות קבצים מתבצעות על המארח (Host) למרות הבידוד בתוך הקונטיינר. זו שאלה שנולדה מהבנת הארכיטקטורה, עוד לפני שבחרתי בכלל באיזה כלי סריקה להשתמש.

לפני ששורפים שעות על כיוון כזה, בודקים דיווחים קודמים, תיקונים והסברים בתיעוד. גם כאן ה-Agent עושה את עבודת הרקע: אני מבקש ממנו מקורות ומברר איתו אם מדובר בבעיה מוכרת או בהתנהגות מתוכננת של המערכת (By Design). בדיקה מקדימה כזו חוסכת המון זמן על כפילויות לפני שמתחילים לבנות PoC.

להחליט מתי להעמיק ומתי לעצור

Agent ישמח להציע עוד ועוד בדיקות גם כשהכיוון מוצה לחלוטין. הכלל שלי פשוט: מה הבדיקה הבאה עשויה להכריע? אם היא רק מנסחת מחדש את אותה ההנחה, אין טעם להמשיך. ראייה שסותרת את ההשערה, דיווח זהה שכבר קיים, או הבנה שהפעולה מותרת לפי מדיניות המערכת - אומרים כולם שעוצרים כאן.

מצד שני, לפעמים צריך לדעת דווקא להתעקש: אם רואים פעולת קובץ שנראית תמימה אבל מתבצעת על המארח במקום בתוך הקונטיינר המבודד, שווה להתעכב עליה. ההחלטה תלויה בחומרת הפגיעה האפשרית ובמה שעדיין חסר כדי להוכיח אותה.

לעבוד בממשק ולהכין את הכלים

חלק מהעבודה עדיין דורש ידיים על המקלדת: פתיחת חשבון בדיקה במערכת, שינוי תפקיד בממשק הניהול או השלמת תהליך התחברות (Login). אחרי כל פעולה כזו אני מעדכן את ה-Agent במצב המערכת החדש - בקשת HTTP בודדת לא תמיד מספרת לו את כל התמונה.



בנוסף, צריך להכין לו את סביבת העבודה: לכידת תעבורת רשת, ניווט יעיל בקוד והקמת סביבת הרצה מקומית (למשל ב-Docker). כאן ה-Skills חוסכים עבודה שחורה: ב-cve-toolkit, במקום להזכיר ל-Agent בכל פעם מחדש לבדוק כפילויות או להפריד בין עובדות להשערות, ההנחיות כבר צרובות מראש בכל שלב. ועדיין, חובת הבדיקה שהוא עמד בהן נשארת עליו.

הניסיון עדיין משנה

מודלים ו-Agents מאפשרים גם למתחילים לקרוא קוד מורכב ולהיעזר בניתוח ובהכנת בדיקות. היתרון של חוקר מנוסה מתבטא היום בבחירת הכיוון ובזיהוי הנחות יסוד שגויות. שני חוקרים יכולים לקבל מה-Agent את אותו הניתוח בדיוק - אבל להגיע למסקנות שונות לחלוטין לגבי מה לעשות הלאה.

זו חלוקת העבודה שלי כיום, והיא עוד תשתנה עם יכולות המודלים. אני לא מתיימר לחזות מה יישאר בלעדית בידי החוקר; אלה המקומות שבהם הניסיון חשוב כרגע.

גישה למודלים למחקר סייבר

נקודה מעשית שחשוב להכיר: חברות ה-AI מגבילות לא פעם בקשות שקשורות למחקר סייבר, גם כשהוא לגיטימי ומורשה לחלוטין.

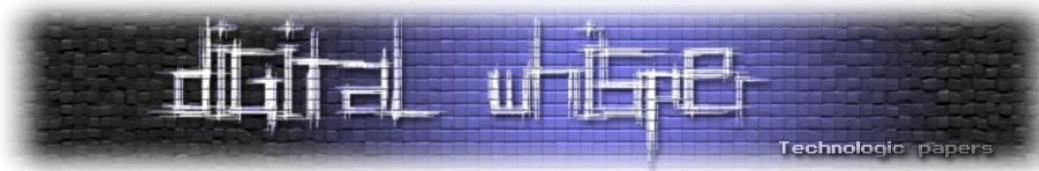
יש כיום מסלולי אישור רשמיים לחוקרים, כמו Trusted Access for Cyber במסגרת Daybreak של OpenAI, או Cyber Verification Program של Anthropic. המסלולים האלה פותחים גישה, אך אינם מהווים צ'ק פתוח ולא מבטיחים שכל בקשה תאושר אוטומטית.

מניסיוני, גם אחרי אישור רשמי לא תמיד יש גישה למודלים הכי מתקדמים למטרות מחקר. לא פעם נאלצתי לעבוד עם מודל זמין ופחות חזק במקום זה שהייתי בוחר מלכתחילה. הזכאות משתנה כל הזמן לפי המודל, סוג החשבון והמדיניות של כל חברה, ולכן כדאי להתעדכן בהגדרות העדכניות של כל ספק.

מהמצא לדיווח מתואם

אחרי שהמצא אומת וגובה בראיות, מנסחים דוח ברור: גרסת המוצר שנבדקה, ההרשאות שנדרשות כדי להפעיל את החולשה, ההתנהגות הצפויה מול מה שקרה בפועל, וההשפעה המוכחת. מצרפים ראיות (PoC, צילומי מסך ולוגים) ומציינים במפורש גם מגבלות אם יש כאלה. רגע לפני השליחה, בודקים שוב שלא פורסם דיווח זהה בינתיים.

ערוץ הדיווח נקבע לפי היעד: מנגנון הדיווח הפרטי של GitHub, פלטפורמת Bug Bounty (כמו HackerOne), או מייל האבטחה הרשמי של מפתחי הפרויקט. את מועד פרסום הפרטים לציבור מתאמים עם המתחזקים בהתאם למדיניות הגילוי של הפרויקט או התוכנית, בדרך כלל לאחר שחרור תיקון.



חשוב לעשות תיאום ציפיות: קבלת הדוח, שחרור התיקון, תשלום Bounty והקצאת מזהה CVE הם תהליכים נפרדים. במקרים של Revive ו-NanoClaw הממצאים תוקנו ללא מזהה CVE, וב-Revive גם ללא תגמול כספי. לכל תוכנית ופרויקט יש את המדיניות והקצב שלהם.

סיכום

בכל הדוגמאות, פלט הסריקה היה רק נקודת הפתיחה. ה-Agent ניתח ובדק; אני בחנתי אם הראיות מחזיקות מים.

עבורי, עבודה עם Agents הפכה רקע בפיתוח תוכנה לכרטיס כניסה מעשי למחקר חולשות. הם מאפשרים לכסות שטח עצום ולנוע במהירות בין כיוונים שונים, אבל גם מייצרים הסברים שגויים שחובה לפסול. הערך האמיתי מגיע מהאימות שקובע מה עובד באמת, ולא מכמות המועמדים שנאספה בסריקה הראשונית.

דרך טובה להתחיל היא לבחור מחקר חולשות שמעניין אתכם, להבין את שורש הבעיה ואת התיקון, ולבחון את ההסברים בסביבת לימוד מקומית ומורשית. המטרה היא ללמוד לזהות את הדפוס ולהעריך ראיות, ולא רק לקבל מה-Agent תשובה משכנעת.

על המחבר

שמוליק כהן הוא AI Engineer ב-APIiro. הוא שירת שש שנים ביחידה 8200 בצוותי פיתוח Backend ותשתיות נתונים. כיום הוא סטודנט לתואר שני (M.Sc.) במדעי המחשב באוניברסיטת תל אביב וכותב את הבלוג "AI Superhero" ב-Substack, שבו הוא מפרסם תובנות ומחקרים על החיבור שבין בינה מלאכותית, פיתוח תוכנה ואבטחת מידע.

מקורות וקריאה נוספת

- [How I Found My First Two SSRF CVEs](#) - תחילת הדרך שלי עם Agents ודפוס כשל משותף בשני פרויקטים.
- [Every CVE Unlocks Another One](#) - מחקר pyLoad, וריאנטים ותיקונים חלקיים
- [Secure by Design, Broken by a Filename](#) - NanoClaw - וגבול האמון בין הקונטיינר למארח
- [Parse Server: CVE-2026-47138](#) - הודעת האבטחה הרשמית על ReDoS והתיקון
- [pyLoad: הודעות האבטחה על שינוי הגדרות Proxy ועל ביטול אימות TLS](#)
- [n8n: MySQL v1 SQL Injection](#) - תנאי החשיפה והגרסאות המתוקנות
- מסלולי גישה למחקר סייבר: [OpenAI Trusted Access for Cyber](#) ו-[Anthropic CVP](#)