Digital Whisper Technologic Papers

BOLABuster: למצוא BOLA בסקייל בעזרת LLMs

מאת רביד מזון

הקדמה

במאמר זה אציג את המחקר והפיתוח של מתודולוגיה שקראתי לה BOLABuster, המשתמשת ב-LLMs לזיהוי חולשות BOLA (או בשמם המלא: Broken Object-Level Authorization) בצורה אוטומטית ובסקייל גבוהה.

לפני שנצלול אל המתודולוגיה עצמה, אציג את המוטיבציה והאתגרים הרבים שעברתי תו"כ הפיתוח, ולבסוף את ה-Evaluation אל מול השוק והחולשות שחשפתי בשימוש ב-BOLABuster עד כה.

BOLA (או בשמה הקודם IDOR), היא חולשה נפוצה ובעלת פוטנציאל קריטי ב-APIs מודרניים וב-Web Applications. היא מדורגת במקום הראשון ברשימת ה-OWASP Top10 API, ובמקום הרביעי ברשימת החולשות המדווחות ביותר על פי HackerOne. ההשלכות של ניצול חולשה מסוג BOLA יכולות להיות הרסניות, החל מ-Data Manipulation, Sensitive Data Exposure ועד ל-Full System Compromise. בעוד שניצול ידני של חולשות BOLA הוא בדרך כלל פשוט, זיהוי אוטומטי של חולשה חדשה מהווה אתגר גדול מהסיבות הבאות:

- The complexities of application logic
- The diverse range of input parameters
- The stateful nature of modern web applications

מסיבות אלו, שיטות מסורתיות כמו Fuzzing ו-Static Analysis אינן יעילות בזיהוי BOLAs, מה שהופך את הזיהוי הידני לשיטה המקובלת כיום. על מנת להתמודד עם אתגרים אלו, אנו מנצלים את יכולות ההיגיון והגנרציה של LLMs כדי להפוך משימות שעד כה נעשו בצורה ידנית לאוטומטית. משימות אלו כוללות את הדברים הבאים:

- Understanding application logic
- Identifying endpoint dependency relationships
- Generating test cases and interpreting test results

על ידי שילוב של LLMs עם יוריסטיקות, השיטה שלי מאפשרת זיהוי אוטומטי מלא של BOLA בקנה מידה רחב. למרות שהמחקר נמצא בשלבים ראשוניים, הצלחתי לחשוף עד כה 17 חולשות BOLA חדשות, הן בפרויקטים פנימיים והן בקוד פתוח. בין החולשות שחשפתי:

- Grafana (CVE-2024-1313)
- Harbor (CVE-2024-22278)
- Easy!Appointments (CVE-2023-3285 to CVE-2023-3290, CVE-2023-38047 to CVE-2023-38055)

BOLA והמוטיבציה למחקר

חולשה מסוג BOLA מתרחשת כאשר ה-API של אפליקציה לא מצליח לאמת אם למשתמש יש את ההרשאות הנכונות לגשת, לשנות או למחוק אובייקט ביישום.

תרשים 1 מדגים דוגמה פשוטה ל-BOLA. ביישום רפואי זה, מטופלים יכולים להשתמש ב-API באופן הבא: *api.clinic[.]site/get_history?visit_id=XXXX* על מנת לגשת להערות של רופא מביקור.

כל מטופל צריך לגשת רק לרשומות הרפואיות שלו. עם זאת, אם השרת אינו מצליח לאמת את הלוגיקה הזו כראוי, מטופל זדוני עשוי לשנות את הפרמטר *visit_id* בבקשה כדי לגשת לנתונים של מטופל אחר. תרשים 1 מציג מניפולציה זו בקריאת API זדונית:

[תרשים 1: דוגמא קלאסית ל-*BOLA*]

דוגמא נוספת לחולשה מסוג BOLA יכולה להיות התרחיש שבו למשתמש קצה א' יש את היכולת לערוך או למחוק תגובה של משתמש קצה ב', באפליקציה כמו טוויטר לדוגמא.

כמובן שלמשתמש קצה א' לא אמורה להיות הרשאה לבצע פעולה זדונית זו, בה הוא מבצע מניפולציה באובייקט שלא שייך לו (במקרה הזה תגובה), במידה והיישום מנהל את הרשאות משתמשי הקצה בצורה נכונה ובטוחה.

למרות שהרעיון של BOLA הוא פשוט, אוטומציה של זיהויו מציבה אתגרים משמעותיים. בניגוד לחולשות נפוצות אחרות כגון XSS, SQL Injection או Buffer Overflow, כלים לבדיקת אבטחה כמו SAST ו-DAST אינם יכולים לזהות BOLAs ביעילות. כלים אלו מסתמכים על דפוסים והתנהגויות מוכרות של החולשות, שאינם חלים על BOLAs. ולכן אין כלי אוטומטי כרגע לזיהוי BOLAs.

בנוסף, אין מסגרת פיתוח שיכולה לסייע למפתחים במניעת BOLAs. כתוצאה מכך, צוותי אבטחה שצריכים לבצע Auditing על BOLAs חייבים ליצור מקרי בדיקה מותאמים אישית באופן ידני.

העובדה ש-BOLA מאוד פופולרית, ההשלכות הקריטיות שלה והעובדה שאין כיום כלי שמוצא חולשות אלה בצורה אוטומטית ובסקייל, הן הסיבות שהניעו את המחקר ואת התוצאה שלו - מתודולוגיה שמוצאת BOLA בצורה אמינה, אוטומטית ובסקייל גבוהה.

האתגרים באוטומציה למציאת BOLAs

מספר אתגרים טכניים תורמים לקושי באוטומציה של זיהוי BOLA, והם:

1. **מנגנוני הרשאות מורכבים:** APIs מודרניים כוללים לעתים קרובות מנגנוני הרשאות מורכבים הכוללים סוגים שונים של משתמשים (Users), תפקידים (Roles), משאבים (Resources) ופעולות (Actions) מרובים. מורכבות זו מקשה על בודקים לקבוע אילו פעולות משתמש אמור להיות מורשה לבצע על משאב מסוים ואילו פעולות אסורות.
2. **Stateful Applications:** רוב ה-Web Applications המודרניים הם Stateful, כלומר כל קריאת API יכולה לשנות את ה-state של היישום ולהשפיע על תוצאות קריאות API אחרות. במילים אחרות, התגובה לקריאה לאחת מנקודות הקצה של ה-API תלויה בתוצאות הביצוע של נקודות קצה אחרות של ה-API (זוכרים את הדוגמה של התגובה בטוויטר? אז אי אפשר למחוק תגובה לפני שיצרנו אותה, נכון?). לוגיקה מורכבת זו משולבת בדרך כלל ב-GUI מולו עובד המשתמש. הנדסה לאחור של הלוגיקה באופן אוטומטי של ה-API Spec ומעקב אחר ה-States השונים של האפליקציה זו משימה לא פשוטה בכלל.

3. **חוסר אינדיקטורים לחולשות:** BOLA היא חולשה לוגית. זה אומר שאין לה דפוסים ידועים שכלי SAST או Compilers יכולים לזהות. בזמן ריצה, BOLA אינה גורמת לשגיאות או מציגה התנהגות ספציפית שחושפת את הבעיות. הקלט והפלט של ניצול מוצלח בדרך כלל מסתיימים בבקשות מוצלחות Status Code 200 והן אינן מכילות Payloads חשודים, מה שמקשה על זיהוי החולשות.

איך יודעים שה-Endpoint שמצאנו כחשוד ל-BOLA באמת פגיע? כמו שאמרה נשיאת Harvard לשעבר: "It depends on the context".

4. **קלטים רגישים ל-Context:** בדיקת BOLA כוללת מניפולציה של פרמטרי קלט של API endpoints כדי לזהות חולשות. תהליך זה דורש זיהוי פרמטרים המתייחסים לנתונים רגישים ואספקת פרמטרים עם ערכים חוקיים לביצוע הבדיקות.

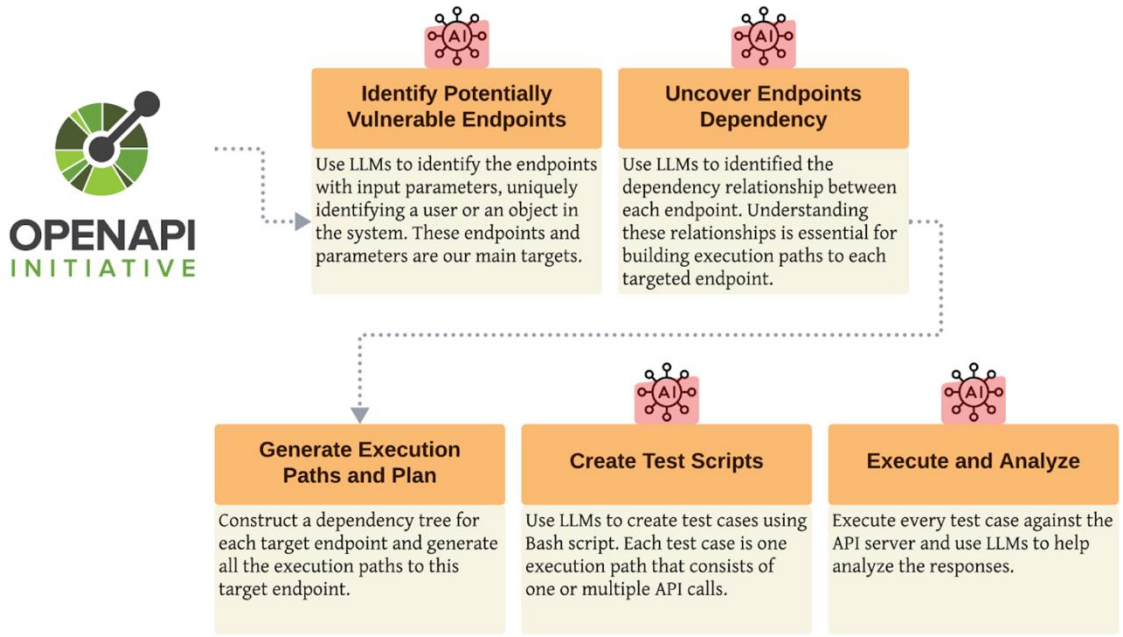
אנו מסתמכים אך ורק על מפרט ה-API (כיום נקרא API Spec, בעבר נקרא Swagger) כדי להבין את הפונקציות והפרמטרים של כל API endpoint, מה שמקשה על קביעה אם ה-Endpoint עשויה לחשוף או לשנות נתונים רגישים. זוהי בדיוק הנקודה שבה ה-LLM נכנס לתמונה.

לאחר זיהוי נקודות הקצה המיועדות והפרמטרים שלהן, השלב הבא הוא לשלוח בקשות לנקודות הקצה הללו ולבחון את התנהגותן. קביעת ערכי הפרמטרים הספציפיים לבדיקה היא קשה מכיוון שרק ערכים שממופים לאובייקטים קיימים במערכת יכולים להפעיל BOLAs. יצירת Payloads כאלו באופן אוטומטי באמצעות טכניקות Fuzzing מסורתיות היא גם מאתגרת, וגם לא יעילה.

מתודולוגיית BOLABuster - זיהוי BOLA בעזרת AI

בהתחשב בהתקדמות האחרונה בבינה מלאכותית גנרטיבית (Gen AI) ובאתגרים של אוטומציה של זיהוי BOLA, החלטתי להתמודד עם הבעיה בעזרת AI על ידי הפיתוח של BOLABuster. מתודולוגיית BOLABuster מנצלת את יכולות ההיגיון של LLMs כדי להבין יישום API ולבצע משימות זיהוי BOLA שהיו בעבר ידניות וגוזלות זמן באופן אוטומטי.

האלגוריתם של BOLABuster, כפי שמתואר בתרשים 2, דורש רק את מפרט ה-API של יישום ה-API המיועד כקלט. BOLABuster יוצר את כל מקרי הבדיקה ממפרט ה-API. הכלי תומך כיום במפרט OpenAPI 3, הפורמט הנפוץ ביותר למפרטי API.

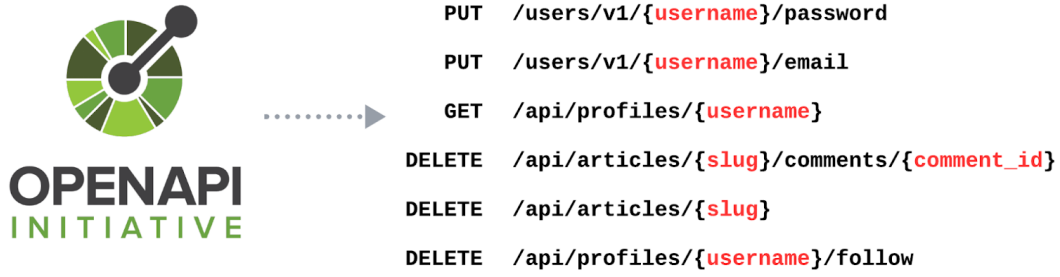


[תרשים 2: סקירה של מתודולוגיית *BOLABuster*]

מתודולוגיית BOLABuster כוללת חמישה שלבים עיקריים:

1. **זיהוי נקודות קצה פוטנציאליות לחשיפה (Potentially Vulnerable Endpoints: PVEs)**: בשלב הראשון שלנו BOLABuster מזהה נקודות קצה של API שעשויות להיות פגיעות ל-BOLA. אנו מתמקדים בנקודות קצה מאומתות עם פרמטרי קלט שמזהים באופן ייחודי אובייקטים במערכת, כגון שם משתמש, אימייל, invoiceId, teamId ו-visitId. נקודות קצה עם פרמטרים אלו עשויות להיות פגיעות ל-BOLA אם ה-backend אינו מצליח לאמת את לוגיקת ההרשאות. ה-AI מסייע בניתוח הפונקציות והפרמטרים של כל נקודת קצה כדי לקבוע אלו מהם מתייחסים לנתונים רגישים או מחזירים אותם.

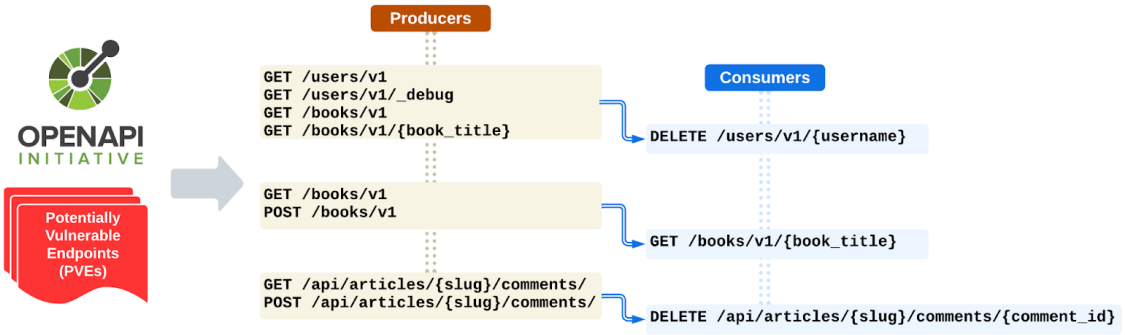
תרשים 3 מציג סט של נקודות קצה פוטנציאליות לחשיפה (PVEs):



[תרשים 3: נקודות קצה פוטנציאליות לחשיפה ל-*BOLA*]

2. **חשיפת תלות בין נקודות קצה:** בשלב זה BOLABuster מנתח את לוגיקת היישום כדי לחשוף קשרי תלות בין נקודות קצה של API. בשל הטבע ה-Stateful של יישומי Web מודרניים, הבנת התנאים המוקדמים של נקודת קצה לפני הבדיקה היא קריטית. לדוגמה, כדי לבדוק את ה-APIs של קופה של אפליקציית עגלת קניות, יש להוסיף תחילה פריטים לעגלה.

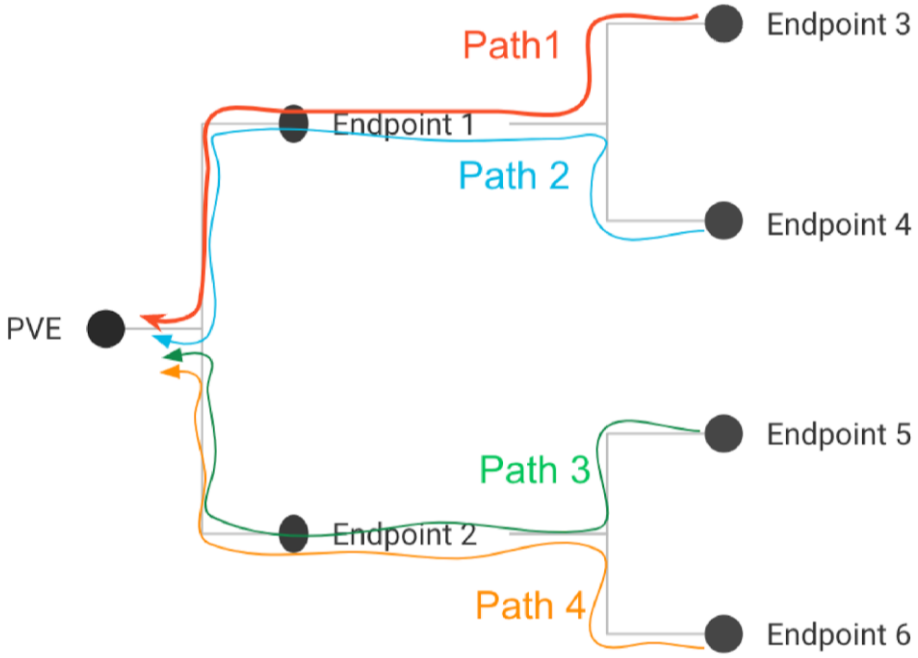
פעולה זו דורשת לדעת את ה-itemId וה-customerId. אנו מסווגים נקודות קצה שפולטות פרמטרים הנדרשים עבור נקודות קצה אחרות כיצרניות (Producers) וכאלה שמקבלות פרמטרים אלו כצרכניות (Consumers), כפי שמוצג בתרשים 4. כל נקודת קצה יכולה לתפקד הן כיצרנית והן כצרכנית. ה-AI מסייע בניתוח הפונקציות והפרמטרים של כל נקודת קצה כדי לקבוע אם נקודת קצה אחת יכולה לפלוט ערכים הנדרשים לקלט של נקודת קצה אחרת:



[תרשים 4: דוגמא לנקודות קצה יצרניות (*Producers*) ונקודות קצה צרכניות (*Consumers*)]

בדוגמא זו ניקח לדוגמא את ה-Consumer המציין נקודות קצה *DELETE /users/v1/username*, כדי לבצע את הקריאה הזאת ביישום, נהיה חייבים לבצע את אחת מקריאת הקצה מסוג היצרניות (Producers), על מנת לייצר או לחלץ ערך תקין מסוג *username*, במקרה הזה.

3. **הפקת Execution path and Test Plan**: באמצעות התוצרים משני השלבים הקודמים, כעת BOLABuster יוצר עץ תלויות עבור כל PVE. כל צומת מייצג נקודת קצה של API, וכל קצה מצומת אב לצומת בן מייצג קשר תלות שבו ההורה הוא צרכן והילד הוא יצרן. השורש של כל עץ תלות הוא PVE, והנתיב מכל עלה לשורש מייצג נתיב ביצוע שיכול להגיע ל-PVE. לאחר מכן, אנו יוצרים Test עבור כל Execution Path (נתיב ביצוע), הכוללת את כל נתיבי הביצוע של ה-PVE וקריאות ה-API שלהם. תרשים 5 מציג דוגמה לעץ תלות עם ארבעה Execution paths:

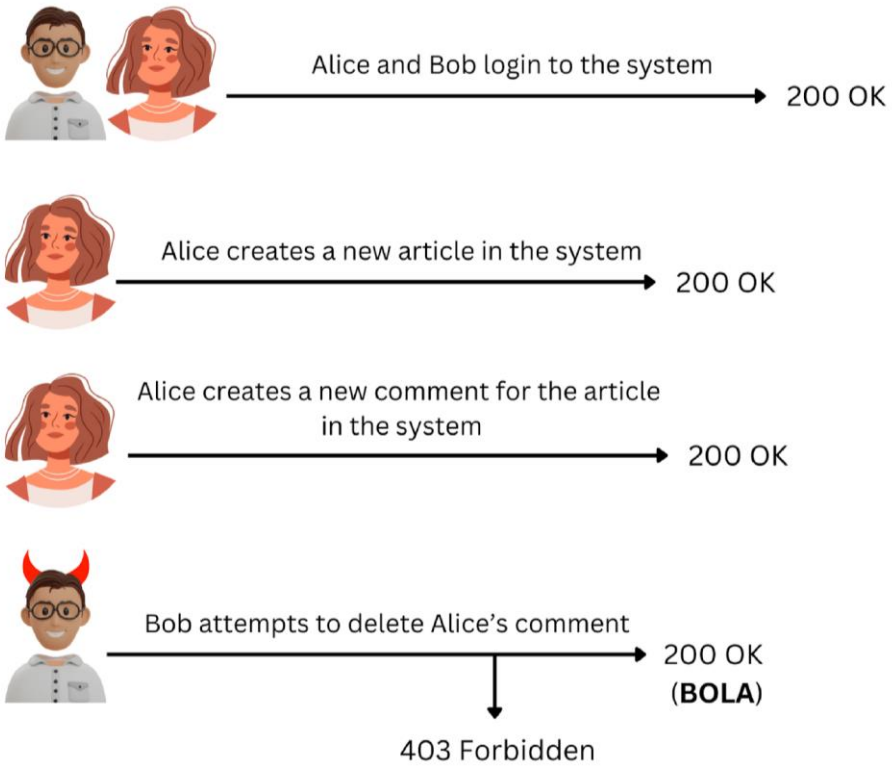


[תרשים 5: *Dependency tree with 4 execution paths*]

4. **יצירת Test scripts**: בשלב זה BOLABuster ממיר כל נתיב ביצוע ל-Executable Bash Script באמצעות LLMs. כל סקריפט מבצע סדרת קריאות API לשרת יעד, החל מ-Login כדי לקבל Tokens עבור ה-Users שלוקחים חלק ב-Test ומסתיים בקריאה ל-PVE. כל סקריפט בדיקה כולל לפחות שני משתמשים מאומתים, כאשר משתמש אחד מנסה לגשת לנתונים של משתמש אחר. אם משתמש אחד מצליח לגשת לנתונים של משתמש אחר או לשנות אותם, זהו אינדיקטור ל-BOLA.

תרשים 6 מציג דוגמה כללית לבדיקת PVE. הבדיקה כוללת שני משתמשים, אליס ובוב, שנכנסים למערכת ומקבלים Tokens, שמזהים כל אחד מהם בצורה ייחודית. אליס יוצרת Article ולאחר מכן Comment על מאמר זה במערכת. מזהי ה-Article וה-Comment נשמרים לשם שימוש בקריאות API הבאות. בוב מנסה לבצע פעולה זדונית - למחוק את התגובה של אליס.

אם הפעולה מצליחה, זה מצביע על BOLA פוטנציאלית:



[תרשים 6: בדיקת היתכנות *BOLA* ב-PVE]

5. **הרצת טסטים ואנליזה:** בשלב זה, BOLABuster מריץ את סקריפטי הבדיקה כנגד שרת ה-API המיועד, ולאחר מכן מנתח את התגובות כדי לקבוע אם ה-PVEs פגיעים ל-BOLA. תהליכים כמו Users Registration, Users Login and Tokens Refresh נעשים בצורה אוטומטית כדי להבטיח ביצוע רצוף של כל תוכנית בדיקה. BOLABuster מפעיל את מקרי הבדיקה לאותו PVE בסדר מסוים כדי למזער תלות ביניהם. לדוגמה, אנו נמנעים מ-Test שמוחק אובייקט ש-Test אחר יזדקק לו. במהות, אנו מבטיחים שכל קריאות ה-API בתוך נתיב ביצוע יצליחו למעט הקריאה ל-PVE. התוצאה של קריאה זו אמורה להצביע על כך שה-PVE פגיע ל-BOLA.

האלגוריתם לקביעת סדר מבטיח שהיישומים יתמלאו בנתונים הנחוצים לפני ביצוע כל ניסיונות גישה. BOLABuster מתזמן את סקריפטי הבדיקה הכוללים פעולות כגון עדכון או מחיקת משתמשים או משאבים בסוף סדר הביצוע כדי למנוע ניסיונות לאחזר משאבים שנמחקו או שונו. לאחר ביצוע הטסטים, התוצאות מנותחות על ידי AI. כאשר ה-AI קובע שנקודת קצה פגיעה, גורמים אנושיים מאמתים את התוצאות כדי להעריך את השפעת ה-PVE בתוך ה-Context של היישום. בעוד שאנו מאוטומטים כמה שיותר משימות בעזרת AI, אימות אנושי נותר חיוני. הניסויים שלנו מראים שמשוב אנושי משפר באופן עקבי את הדיוק והאמינות של ה-AI.

Evaluation

BOLABuster הוכיח תוצאות מבטיחות בזיהוי נקודות תורפה ידועות של BOLA במספר פרויקטים בקוד פתוח. כלים מסורתיים ל-API Fuzzing דורשים בדרך כלל שעות או אפילו ימים כדי לבדוק כל נקודת קצה של API עם עשרות אלפי בקשות אקראיות. לעומת זאת, המתודולוגיה שלנו מעצבת אסטרטגית מקרי בדיקה עבור כל נקודת קצה ומתמקדת רק באלה שעלולים להיות פגיעים ל-BOLA.

טבלה 1 מציגה את תוצאות ההערכה של BOLABuster מול שלושה פרויקטים פגיעים בכוונה המכילים נקודות תורפה של BOLA. פרויקטים אלה משמשים כמצע מבחן, המציע אמת יסוד להערכת הביצועים והדיוק של המתודולוגיה שלנו. אנו משווים את הביצועים של המתודולוגיה שלנו עם RESTLer Fuzzer, פרויקט קוד פתוח פופולרי ומתוחזק, המפותח ע"י Microsoft ונמצא בשימוש נרחב, שמטרתו מציאת חולשות על ידי Fuzzing API Services.

בבדיקות אלו, השתמשנו ב-GPT-4 של OpenAI כמודל השפה שלנו ללא כל כוונון. RESTLer שומש עם הגדרות ברירת המחדל שלו. בתוך RESLler, ה-NameSpaceRuleChecker, שמזהה גישה לא מורשית לשירותים או משאבים, היה אחראי על זיהוי BOLAs. בודק זה הופעל במהלך הבדיקות שלנו.

במהלך ה-Evaluation, הערכנו בעיקר שני מדדים:

- מספר נקודות התורפה האמיתיות של BOLA שזוהו על ידי כל כלי
- מספר קריאות ה-API שנעשו על ידי כל כלי (לא כולל בקשות Registration ו-Login)

התוצאות היו מדהימות: **BOLABuster** זיהה בהצלחה את כל נקודות הקצה הידועות של **BOLA** בשלושת הפרויקטים הללו. למרות ש-RESTLER זיהה נקודות תורפה ובאגים מרובות בכל אפליקציה, היא לא הצליחה לזהות נקודות תורפה של BOLA. בנוסף, במהלך הטסטים, **BOLABuster** שלח פחות מ-1% מבקשות ה-**API** לשרת יעד תוך חשיפת נקודות תורפה יותר מאשר **RESTler**:

	Known BOLAs	RESTLer BOLA Discovery	BOLABuster BOLA Discovery	RESTLer API Calls	BOLABuster API Calls
VAmPI	3	**0**	**3**	5,123	**39**
Capital	2	**0**	**2**	202,602	**141**
crAPI	2	**0**	**2**	156,003	**765**

[טבלה 1: תוצאות הבדיקה: השוואה בין *BOLABuster* ל-RESTLer]

טבלה 2 מציגה את ה-Confusion Matrix עבור BOLABuster. המטרה העיקרית שלנו היא לשפר את שיעור ה-True Positive תוך שמירה על כמה שפחות מקרי False Positives. אנו שואפים להגיע ליותר נקודות קצה של BOLA בפועל על ידי מיקסום כיסוי מקרי הבדיקה במחיר של עלייה במספר חיוביות כוזבות. עם זאת, מכיוון ש-BOLABuster מדווח על מספר נמוך מאוד של נקודות קצה פגיעות, עדיין ניתן לסקור ולאמת את התוצאות הללו.

ראוי לציין שהסרנו את התוצאה של RESTler מה-Confusion Matrix מכיוון שהיא לא דיווחה על נקודות קצה פגיעות ל-BOLA במהלך הבדיקה:

App	Total BOLA Detection	True Positives	False Positives
VAmPI	8	3	5
Capital	10	2	8
crAPI	8	2	6

[טבלה 2: *Confusion Matrix*]

חושפים BOLAs בעולם האמיתי

המאמצים המתמשכים שלנו בבדיקת ובחינת פרויקטים בקוד פתוח באמצעות BOLABuster הובילו לזיהוי ודיווח על מספר חולשות BOLA שלא היו ידועות קודם לכן, שחלקן עלולות לגרום ל-Privilege Escalation קריטית. סעיף זה מספק סקירה קצרה של החולשות שגילינו בשלושה פרויקטים בקוד פתוח: Grafana, Harbor ו-Easy!Appointments.

Grafana (CVE-2024-1313):

Grafana הוא כלי פופולרי לויזואליזציה של נתונים וניטור שמאפשר למשתמשים למשוך נתונים ממקורות שונים כדי לצפות ולהבין מערכות נתונים מורכבות. BOLABuster מצא את *CVE-2024-1313*, המאפשר למשתמשים בעלי הרשאות נמוכות מחוץ לארגון למחוק Dashboard Snapshot השייכת לארגון אחר באמצעות Snapshot Key.

סיקור החולשה באופן מלא מוצג בבלוג שפרסמתי.

Harbor (CVE-2024-22278):

Harbor הוא פרויקט CNCF (או בשמו המלא: Cloud Native Computing Foundation) המשמש כפלטפורמה ל-Container Registry. האפליקציה מספקת מאפיינים שליטה במשתמשים ומשאבים באמצעות RBAC (או בשמו המלא: Role-Based Access Control) וכן Vulnerability scanning ו- Images Signing.

BOLABuster זיהה את CVE-2024-22278, המאפשר למשתמש מסוג Maintainer, לעדכן ולמחוק נתוני קונפיגורציה של פרויקט. פעולות בעלות סיכון גבוה אלו צריכות להיות מוגבלות למנהלים בלבד, לפי התיעוד הרשמי של Harbor, אך מצאנו שלא כך הדבר.

סיקור החולשה באופן מלא מוצג בבלוג שפרסמתי.

Easy!Appointments - 15 חולשות חדשות (CVE-2023-3285 - 3290, CVE-2023-38047 - 38055):

Easy!Appointments הוא כלי לתזמון וניהול פגישות המשמש בעיקר עסקים קטנים. BOLABuster חשף לא פחות מ-15 נקודות קצה פגיעות המאפשרות למשתמשים בעלי הרשאות נמוכות לעקוף בקרות הרשאות, מה מאפשר לתוקף לבצע במערכת פעולות של Unauthorized Access, Data Manipulation, Sensitive Data Exposure, ואפילו ל-Full System Compromise.

מתוך 15 החולשות החדשות שמצאנו, 7 בעלות CVSS Score של 9.9, ומוגדרות שכחולשות קריטיות.

סיקור החולשות באופן מלא מוצג בבלוג שפרסמתי.

סיכום

המחקר שלי מדגים את הפוטנציאל המשמעותי של AI במהפכת זיהוי החולשות והמחקר האבטחתי. על ידי שימוש ב-LLMs, משימות שהיו בעבר ידניות וגוזלות זמן ניתנות כעת להתבצע בצורה אוטומטית. הראינו ש-AI יכול לשמש כעוזר מהימן. זה נכון לא רק לכתיבת קוד אלא גם לניפוי שגיאות וזיהוי חולשות.

למרות שהמחקר עדיין בשלבים ראשוניים, ההשלכות הן עמוקות. המתודולוגיה שפיתחנו לזיהוי BOLA עשויה להיות מורחבת לזיהוי סוגים אחרים של חולשות, ופותחת דרכים חדשות למחקר חולשות. ככל שטכנולוגיית AI תמשיך להתקדם, אנו מצפים שגישות דומות יאפשרו מגוון של יוזמות מחקר אבטחה שהיו בעבר לא מעשיות או בלתי אפשריות.

כדאי גם לציין שטכנולוגיה זו יכולה להיות חרב פיפיות. בעוד שהמגנים יכולים להשתמש ב-AI לשיפור אמצעי האבטחה שלהם, היריבים יכולים לנצל את אותה טכנולוגיה כדי לגלות חולשות Zero Days מהר יותר ולהסלים התקפות סייבר.

המושג של להילחם ב-AI עם AI מעולם לא היה רלוונטי יותר, כשאנו שואפים להתעלות על היריבים עם פתרונות AI חכמים ומדוייקים יותר. חשוב שהקהילה האבטחתית תישאר ערה ופרואקטיבית בפיתוח אסטרטגיות לנטרול איומים פוטנציאליים הנגרמים מ-AI.

כאשר אנו משתמשים במתודולוגיה זו לזיהוי חולשות BOLA במוצרים שונים, אנו מדווחים באחריות על כל מה שנמצא לספקים המתאימים.

על המחבר

שמי **רביד מזון**, אני Senior Security Researcher ב-Palo Alto Networks. ההתמחות שלי היא בעיקר באזורי ה-Web Application Security ו-API Security. מעבר ל-BOLABuster, אני היוצר של C{API}TAL (יחד עם **ליעד לוי** ו**יניב ניזרי**), פרויקט קוד פתוח שמדמה אפליקציה פגיעה לחולשות OWASP API Top 10 לשם לימוד ותקיפה בסביבה בטוחה ומבוקרת.

בנוסף, תחזקתי בעבר בלוג סייבר ב-Medium, בעיקר תחת השם CyberX. בזמני החופשי אני נהנה ממשחקי ספורט (בעיקר כדורגל), לטייל ולדאוג לכלבה שלי: מייפל.

ליצירת קשר, ניתן לפנות אלי דרך תיבת המייל:

ravid55590@gmail.com

או בלינקדין:

https://www.linkedin.com/in/ravid-mazon/