הידעוני (Diviner) - ראייה צלולה בעולם הדיגיטלי

טכניקות לניבוי מבנה הזכרון וקוד המקור של צד השרת

נכתב ע"י שי חן (CTO ב-Hackitcs ASC)

הקדמה

מאמר זה מתאר פרויקט בשם Diviner - הרחבה לכלי הבדיקה OWASP Zed Attack Proxy (ZAP) המאפשר ליועצי אבטחה "לנבא" את מבנה הזכרון וקוד המקור של השרת, לצפות בהשפעה של כל פרמטר קלט על כלל המערכת, לאתר תרחישי התקפה מורכבים ועקיפים, ואף לקבל המלצות לגבי ההתקפות אותם כדאי לבצע כנגד כל רכיב במערכת.

בנוסף לכך, המאמר מתאר סט התקפות יחודי עליו מתבסס הפרויקט - התקפות ניבוי, המאפשרות לחזות מבנה והתנהגות של רכיבים ותהליכים בצד השרת.

הפרויקט היינו פרויקט קוד פתוח המשולב במספר פרויקטים אחרים, וניתן להורדה מהכתובת הבאה:
http://code.google.com/p/diviner/

מבוא - חלוקת משאבים בלתי אפשרית

מבחינתי, להיות יועץ אבטחת מידע היה נקודת אור בקריירה. בהחלט אחת העבודות היותר מעניינות שיצא לי לעשות. תמיד יש משהו חדש ללמוד, מטרות להשיג, הבעות אימה ספונטניות של לקוחות שאפשר לאסוף לאלבום. בכנות, קשה למצוא עבודה יותר מאתגרת. אבל החסרון, לפחות מבחינתי, הוא שתמיד יש תחושה של חוסר זמן. כיועצי אבטחת מידע, רובנו המכריע חי בסביבה של עבודה בזמנים קצובים, שלא פעם אינם מספיקים לביצוע כלל המטלות הנדרשות במבדקי אבטחה. יתרה מכך, קבלת החלטות שגויה, תמחור אגרסיבי ואף חוסר מזל, עשויים להחמיר את הבעיה.

לאלו מכם שקוראים את המאמר אבל עדיין מרימים גבה אחרי ההקדמה, תנסו לענות לעצמכם **בכנות** על השאלות הבאות:

- כיצד את/ה מחליט/ה היכן וכיצד להשקיע את הזמן שמוקצה לך בבדיקה?
- מתי הפעם האחרונה בה הספקת לבצע את כל הבדיקות שרצית במערכת שגודלה מעבר למספר בודד של מודולים?
- יצא לך לבזבז שעות על גבי שעות בנסיון לאימות LEAD במערכת, רק בשביל לגלות שאין שום פגיעות?

התשובות לשאלות הללו הן כמובן אינדיבידואליות, אך לרובנו המכריע יש מכנה משותף: רובנו "שרפנו" לא מעט שעות במרדפי סרק על פוטנציאלים לא רלוונטיים, לרובנו היו מקרים בהם נאלצנו לנסות התקפות באופן מדגמי ולא יסודי מקוצר זמן, ורובנו מחליטים מה לבדוק ומה לא על סמך שילוב של נסיון, אינטואיציה, איסוף מידע, ולפעמים (מה לעשות) קצת מזל. בעיה לא פשוטה, אבל בהחלט לא בעיה ללא פתרון.

הפתרון הנוכחי - תהליכים וכלים להתמודדות עם בעיית כיסוי המערכת בבדיקה

יש מאות התקפות פוטנציאליות שעשויות להתקיים בכל מערכת, בכל מודול, ואפילו בכל פרמטר שאנו בודקים. מספיק לעבור על הרשימה של OWASP Attacks & Vulnerabilities בכדי להבין שכיסוי של כלל הבעיות באמצעים ידניים בלבד בהחלט אינה מטלה קלה.

רובנו מתמודדים מול אתגר ה-Test Coverage באמצעות שימוש בכלים שונים - סורקי אבטחה אוטומטיים, Fuzzers ותהליכי איסוף מידע; כלים המאפשרים לנו לכסות יותר בדיקות ולאתר מיקומים בהם יותר כדאי לנו להשקיע את הזמן.

למרות היתרונות הרבים של כלים אלו, לכולם יש מגבלות שונות שמונעות מהבעיה להגיע לפתרון מלא:

- סורקי אבטחה אוטומטיים (Scanners) מסוגלים לנסות ולאתר מספר רב של חשיפות ולבדוק את כלל הרכיבים באפליקציה, אך הם אינם מסוגלים לאתר חשיפות שהם אינם מכירים, אינם מסוגלים לאתר מופעי חשיפות אשר הלוגיקה שלהם אינה מתקדמת מספיק לאתר, ואינם מסוגלים לטפל במגוון רחב של מקרי קצה, כגון "התקפות לא ישירות".
- כלי Fuzzing אוספים תגובות של רכיבי האפליקציה השונים לקלטים רבים, ומאפשרים למשתמש האנושי להסיק מסקנות על תגובות אלו, אך דרך הצגת האינפורמציה קשה לניתוח ודורשת עבודה רבה, לרוב אין פעולות אוטומטיות לניתוח והסקה על הנתונים השונים, וברוב הכלים הללו, אין תמיכה בבדיקה של תרחישים מורכבים.
- תהליכי איסוף מידע מתחלקים לשני סוגים עיקריים - תהליכי איסוף מידע פסיביים, בהם נאסף מידע בצורה לא אינטרוסיבית ממקורות שונים (מנועי חיפוש, הערות HTML וכדומה), ותהליכי איסוף מידע אקטיביים, הכוללים תהליכים אוטומטיים כגון File Enumeration ו-Fingerprinting, אך גם תהליכים

ידניים המתבצעים לאיתור אינפורמציה נוספת מרכיב הנמצא בבדיקה. תהליכי האיסוף האוטומטיים מוגבלים לאיסוף אינפורמציה שנחשפת באופן ברור, או להתקפות מיפוי שונות, ואילו התהליכים הידניים מוגבלים למקומות אותם יש לנו זמן לבדוק באופן ידני.

האבסורד - שימוש מדגמי בלבד בכלי הטוב ביותר

בזמן שאנו בודקים רכיבים ספציפיים (כגון דפים או פרמטרים), אנו למעשה מבצעים תהליכי איסוף מידע באופן אקטיבי. אנו בודקים מה התנהגותם של רכיבים בתגובה לקלטים מסוימים, אנו מנסים לגרום לשגיאות שיחשפו מידע, לקלט שיוצג חזרה, לשינוי במבנה התוכן או אפילו לעיכוב בזמן הפעולה של רכיבים שונים.

המידע שחוזר מתהליכי איסוף מידע אלו מאפשר לנו להשתמש באינטואיציה ובמוח האנושי - אולי הכלי החזק ביותר שיש ברשותנו בעת הבדיקה, בכדי להסיק האם יש פוטנציאל אותו שווה לבדוק. בניגוד לכלים אוטומטיים, האינטואיציה האנושית יכולה לאתר התנהגות חשודה שעשויה להוות פוטנציאל לפגיעות גם בלי להכיר את תבנית המתקפה, ותהליכי איסוף המידע הידניים מאפשרים לאתר מידע עליו ניתן להפעיל שיקול דעת.

הבעיה בתהליכי איסוף המידע הללו נובעים מהעובדה שהם ידניים, ומתבצעים על מספר מקומות מוגבל. כלומר, הכלי החזק ביותר שברשותנו, OUR BRAIN, משמש בפועל להסקת מסקנות על התנהגות של מספר רכיבים **מוגבל** בלבד, ועל **רוב** החלקים האחרים עוברים כלים אוטומטיים באיכויות לא ידועות, בפרט במערכות גדולות. אמת, ניתן עדיין לאתר כמות גדולה של חשיפות ולעשות עבודה מצוינת, אבל במידה ולא מדובר במערכת בסדר גודל קטן, השלמות תלויה פחות או יותר, בניסיון ומזל.

פתרון אפשרי: התקפות ניבוי - המרת מידע מאיסוף מידע אקטיבי-מסיבי

היתרון בבדיקות ידניות הוא הסקת מסקנות אנושית, אך החסרון הוא העדר היכולת להשתמש באיסוף מידע ידני על כלל המערכת. היתרון בבדיקות אוטומטיות שונות הוא היכולת לבצע בדיקות רבות על כלל המערכת, אך החסרון הוא העדר הסקת מסקנות אנושית.

השיטות הללו, יחדיו או בנפרד, אינן מהוות פתרון מלא לבעיה, ומכאן נובע שבכדי לנצל את שיקול הדעת של היועץ בצורה הטובה ביותר, יש למצוא דרך **למזג בניהם**.

במקום לבדוק האם התנהגות מסוימת קיימת במקום אחד, למה לא לבדוק האם היא קיימת במספר רכיבים בבת אחת?

מימוש ממשק שיאפשר לבודק לבצע איסוף מידע אקטיבי על **כלל הרכיבים**, ממשק שיציג לבודק את התנהגויות החשודות שהיה מחפש בעצמו, יאפשר לבודק לחסוך כמות רבה של זמן בבדיקות, לקבל

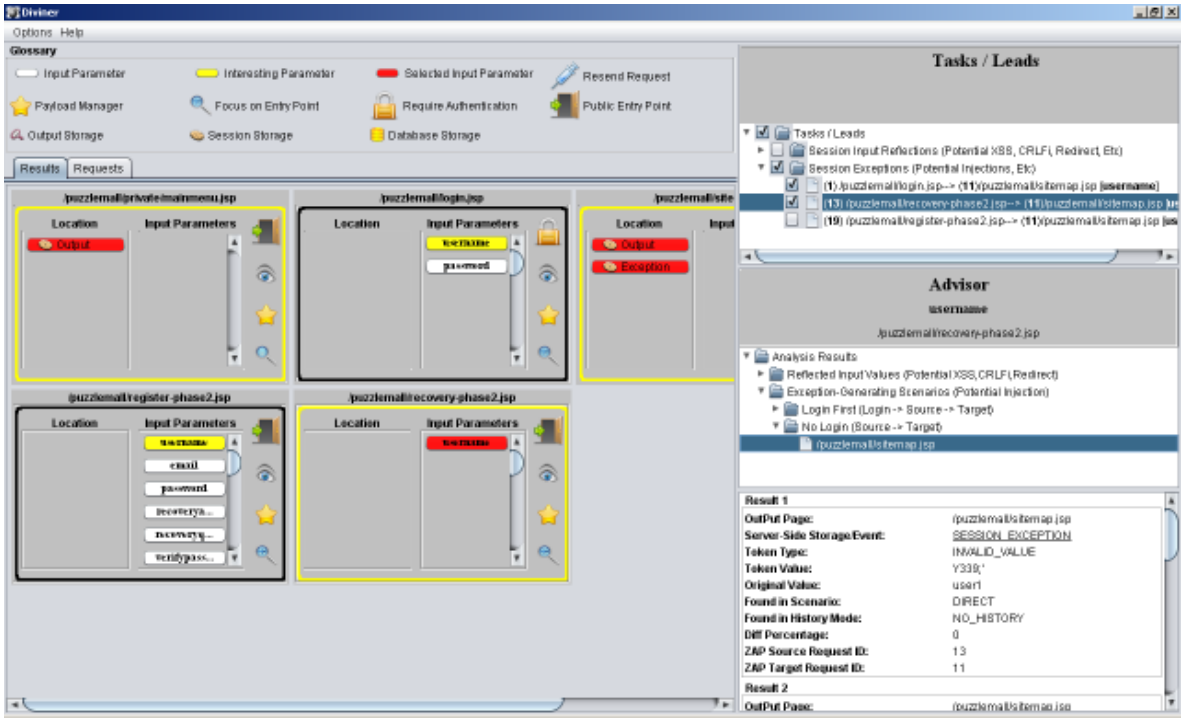
החלטות טובות יותר, ויאפשר לו לאתר תוצאות שונות מכלי אוטומטי, כל עוד יותיר בידיו את תהליך הסקת המסקנות לגבי קיום הפגיעויות.

על בעית "עודף המידע" ניתן להתגבר על ידי הצגת המידע בפורמט **ויזואלי** שיאפשר ליועץ להסיק מסקנות מהר יותר, מבלי לנתח מיד כל פריט מידע קטן.

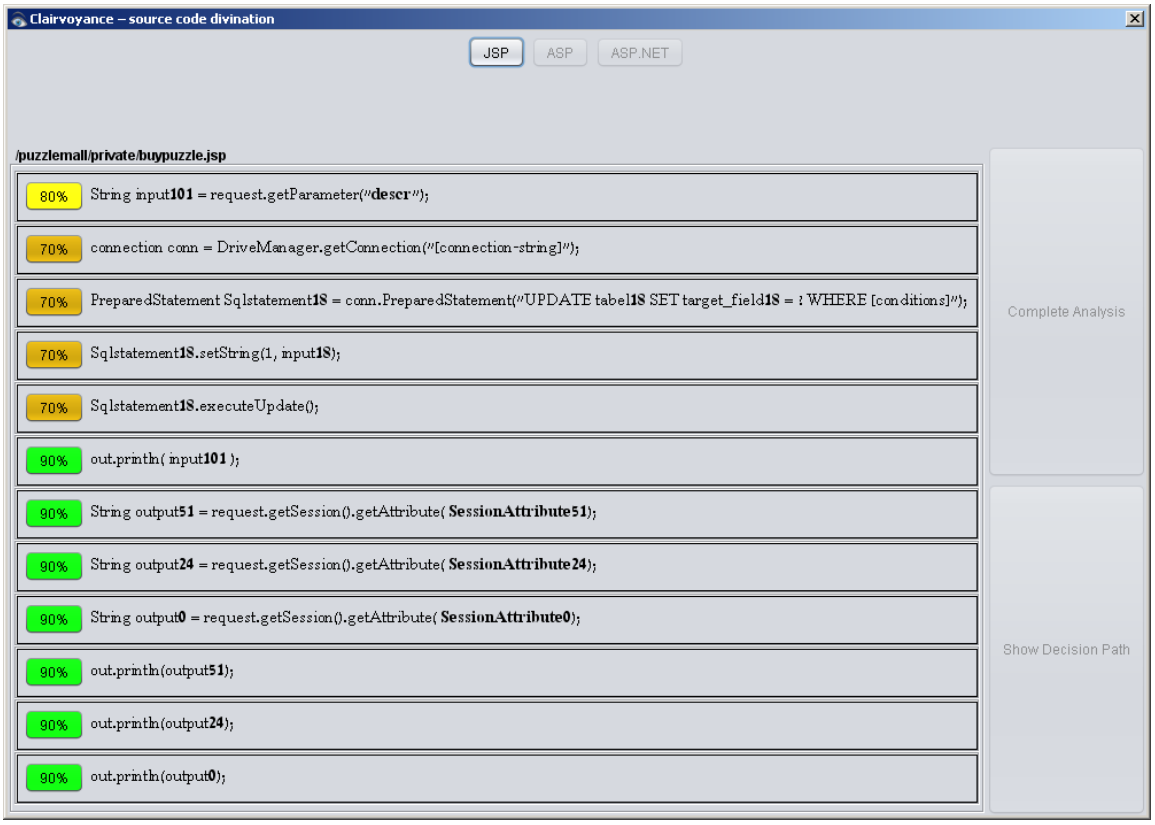
Diviner - פלטפורמה לאיסוף מסיבי של מידע באופן אקטיבי

למען הכנות, כשהרעיון הועלה בפעם הראשונה, מעטים מעמיתי האמינו שהטכנולוגיה אפשרית... אבל עם עזרתו הנאמנה של @Secure_ET (ערן תמרי), הרבה תמיכה מהאנשים שמאחורי פרויקט OWASP ZAP, ולאחר תהליך פיתוח שארך כשנה, הפרויקט קרם עור וגידים. לפני שנתחיל לדון ביתרונותיה השונים של הפלטפורמה, או במכניקה שמאחורי ההתקפות השונות, רצוי שנציג מה התרומה שלה לתהליך הבדיקה.

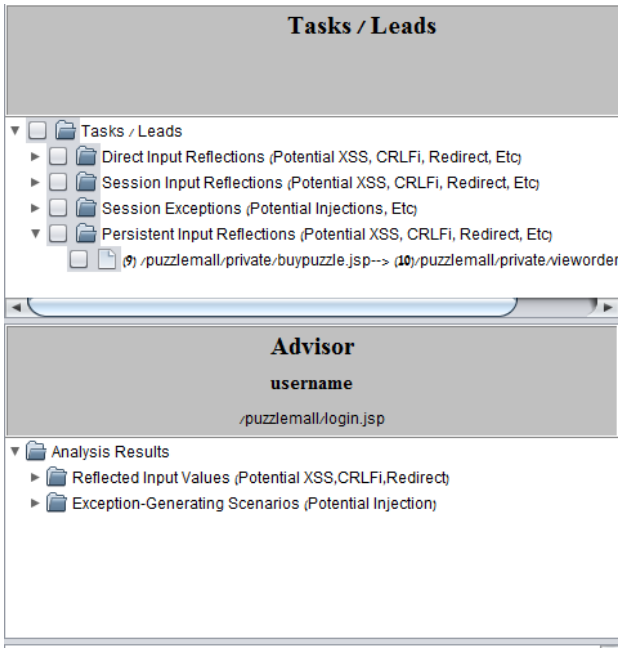
דמיינו לכם סיטואציה בה במקום לבדוק כיצד משפיע כל פרמטר על הדף אליו הוא נשלח, או על כל דף אחר באפליקציה, כל שהייתם צריכים לעשות הוא ללחוץ עליו במפה ויזואלית:



או לחילופין, תארו לכם שארגון שאתם בודקים את המערכת שלו מתעכב או מסרב לחשוף קוד מקור, אך קיים ברשותכם כלי שיכול להציג חלק ממנו בכל זאת:



ולקינוח, תחשבו על הפשטות בבחירת התנהגות חשודה לבדיקה מתוך רשימת התנהגויות של כלל רכיבי המערכת, על פני ניסיון איתור של התנהגות זו באופן אקראי וידני:



מכניקת ההמרה ורעיון הבסיס

כשתיארנו את הבעיות השונות העומדות בפני היועץ בזמן הבדיקה, הסברנו מה החשיבות של איתור התנהגויות מחשידות. כדוגמה טובה לתיאור התהליך, ניקח התנהגות של **עיכוב תגובת המערכת** באמצעות התקפת ADOS מסוג Connection Pool Consumption:

התקפה זו היא התקפה הגורמת למניעת שירות ומתבצעת על ידי פניות חוזרות ונשנות עם מספר רב של Threads לרכיב הניגש לבסיס הנתונים, המבצע את הפניה על ידי קבלת קישור לבסיס הנתונים מ"בריכת קישורים" (Connection Pool). הפניות הרבות והרצופות לרכיב מבטיחות שתמיד יהיה "תור" לקבלת קישור לבסיס הנתונים, מה שיוביל לעיכוב בזמן תגובת המערכת (או לפחות, של כלל רכיבי המערכת הניגשים לבסיס הנתונים דרך בריכת הקישורים).

הפוטנציאל לקיומה של התקפה זאת יאותר באמצעות קיומה של התנהגות מסוג **עיכוב בזמן התגובה** ברכיב הנבדק, בתגובה **לקלט או לפנייה** ספציפית.

המסקנה העיקרית היא שהמיקום הנבדק עשוי להיות פגיע להתקפה האפליקטיבית הנ"ל, אך ישנה גם **מסקנה משנית**: המסקנה המשנית היא שכנראה קיימת ברכיב הנבדק שורת קוד שניגשת לבריכת קישורים:

```
Connection conn = ConnectionPoolManager.getConnection();
```

* הדוגמאות מובאות ב-Java, אך ניתן להציג את הקוד שמאחורי ההתנהגויות בכל שפה אחרת.

כמו כן, קיימת סבירות גבוהה שאותו רכיב במערכת גם ניגש למאגר המידע אליו קיבל את הקישור, מה שעשוי להצביע על קוד פנייה לבסיס הנתונים המבצע שאילתת SELECT/UPDATE/Etc.

למעשה, ניתן להסיק מסקנות משניות אלו משלל התנהגויות מחשידות או נורמליות במערכת, ולהמיר אותם לקוד שעשוי להיות מאחריהם, בשפה פיתוח כלשהי.

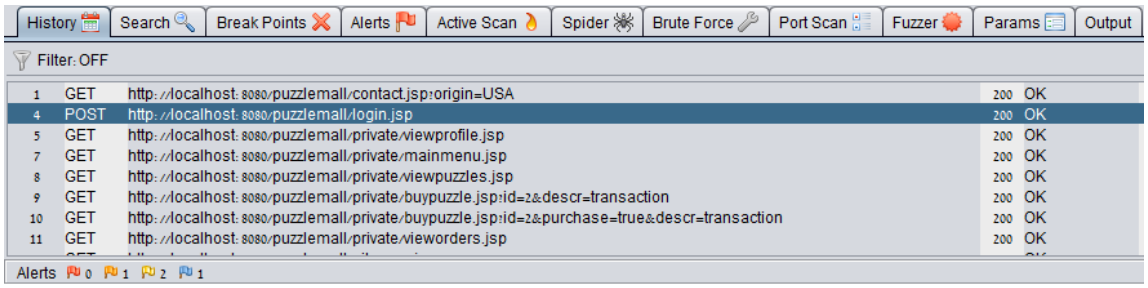
בנוסף, ניתן להמיר את ההתנהגויות לצורות שונות של תצוגה, כגון תצוגת זכרון, מפת תהליכים וצורות נוספות.

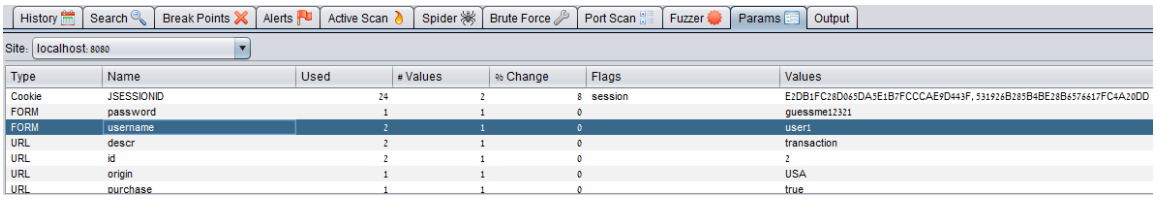
איסוף התנהגויות מסיבי מכלל רכיבי האפליקציה

ככל שיהיה בידינו יותר מידע על התנהגויות שונות של רכיבים, נוכל להציג לבודק יותר מידע על האפליקציה ככלל, ועל כל רכיב בפרט. יתרה מכך, איתור התנהגויות שמתרחשות רק **ברצפי פניות למספר רכיבים**, יאפשרו לנו להבין כיצד הרכיבים השונים **משפיעים** אחד על השני, ולהמיר הבנה זו לתצוגות קוד או לתצוגות אחרות. בכדי לאתר כמה שיותר התנהגויות, ננקוט בגישה פשוטה: נבצע כמה שיותר ניסיונות איסוף מידע - נפעיל כל רכיב במספר רב של דרכים, ונבדוק שוב ושוב כיצד רצף הפניה השפיע על רכיבים אחרים.

בכדי להמחיש את תהליך איסוף המידע המתבצע, נבחן אותו על הדוגמה הבאה:

ZAP שומר היסטוריה של הבקשות שנשלחו דרכו והתשובות שהתקבלו מהשרת. על מנת לבודד כמות גדולה של התנהגויות ייחודיות, מתבצע תהליך מחזורי המתחיל מחדש עבור כל בקשה בהיסטוריה של ZAP:





הסבר פשוט:

עבור כל פרמטר בדף הראשון, נשלח ערך אקראי, ונבדקת תגובת הדף הראשון בהיסטוריה, נשלח ערך אקראי נוסף, ונבדקת תגובת הדף השני בהיסטוריה, וחוזר חלילה. התהליך עצמו מתבצע שוב עבור כל הפרמטרים בדף השני, השלישי, וכן הלאה.

כל התנהגות חשודה המאותרת ברצף הפניות הנ"ל מתועדת לבסיס הנתונים, ומומרת בסופו של דבר לצורות תצוגה שונות.

הסבר מלא:

עבור כל פרמטר קלט בכל דף המופיע בהיסטוריה, מתבצעות הפעולות הבאות:

- שידור מחדש של הבקשה עם ערך אקראי לפרמטר, ובדיקת התגובה של דף המקור.
- שידור מחדש של הבקשה עם ערך אקראי לפרמטר, ובדיקת התגובה בפניה לדף אחר בהיסטוריה, וחוזר חלילה עבור כל אחד מהדפים בהיסטוריה.
- שידור מחדש של הבקשה עם ערך אקראי הכולל תווים שאינם תקינים (מיועדים לגרום לשגיאות), ובדיקה התגובה של דף המקור.
- שידור מחדש של הבקשה עם ערך אקראי הכולל תווים שאינם תקינים (מיועדים לגרום לשגיאות), בדיקת התגובה בפניה לדף אחר בהיסטוריה, וחוזר חלילה עבור כל אחד מהדפים בהיסטוריה.

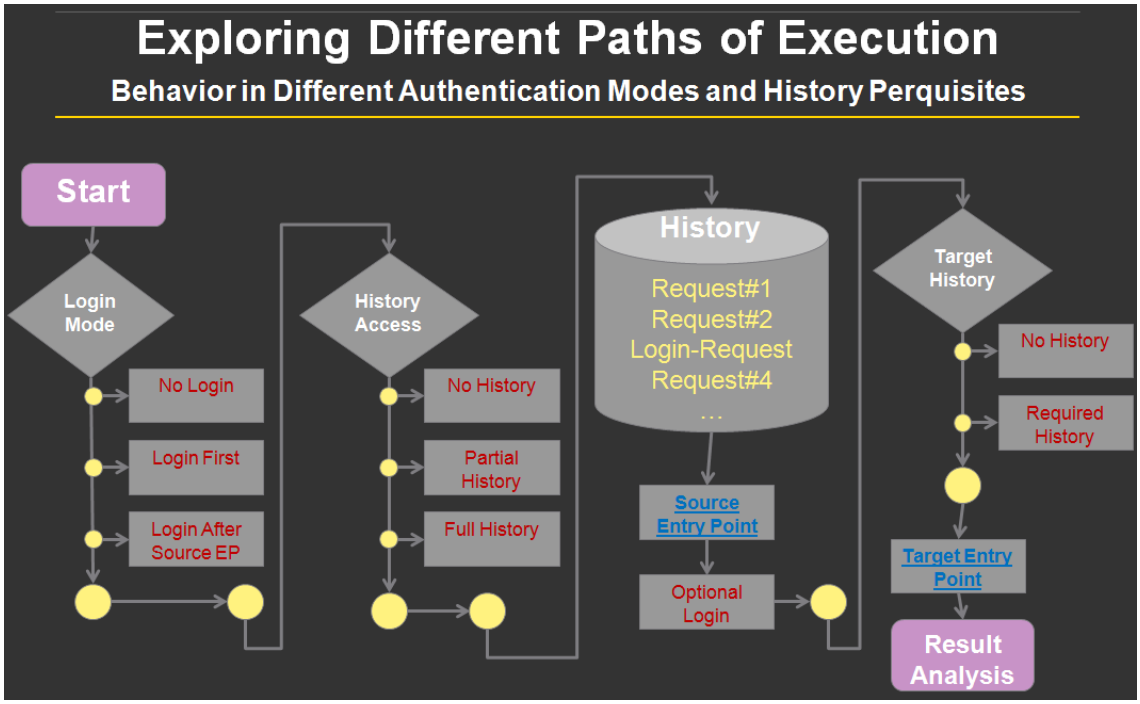
- שידור מחדש של הבקשה עם ערך תקין בעל משמעות לוגית (המצוי בהיסטוריה או מוגדר כחלק מרשימה מיוחדת על ידי המשתמש), ובדיקת התגובה של דף המקור.
- שידור מחדש של הבקשה עם ערך תקין בעל משמעות לוגית (המצוי בהיסטוריה או מוגדר כחלק מרשימה מיוחדת על ידי המשתמש), בדיקת התגובה בפניה לדף אחר בהיסטוריה, וחוזר חלילה עבור כל אחד מהדפים בהיסטוריה.

התהליך כולו חוזר על עצמו עם תהליך Login ו-Session מזוהה, ללא Login, ועם Login בין הדף בו נשלח הקלט לדף בו נבדקת השפעת הקלט.

התהליך יכול להתבצע תוך כדי הגדרת דפים שיש להריץ בכל מחזור לפני פניה לדף בו נשלח הקלט (בכדי לתמוך בתהליכים כגון שחזור סיסמא, טרנזקציות והרשמה), ובנוסף, התהליך עצמו עשוי להתפצל לתהליכים נוספים במקרי קצה מסוימים, כגון החלפת Session, הוספת ערך ל-Cookie, ערך AntiCSRF Token לא תקין, וכדומה.

כל אחת מהתגובות המתקבלות מנותחת לאיתור התנהגויות חשודות, כגון פלט המושפע מקלט, שגיאות, תבניות שהגדיר הבודק והבדלי תוכן ביחס לתשובה המקורית בהיסטוריה - כאשר כל התנהגות חשודה מתועדת לבסיס נתונים יעודי, ומקושרת לדף בו נשלח הקלט ולדף בו נגרמה ההתנהגות החשודה.

התרשים הבא מתאר **חלק** מהתהליכים המבוצעים בזמן איסוף המידע:



המרת התנהגויות לקוד מקור

בסופו של תהליך איסוף המידע, קיימת **אינפורמציה רבה** על התנהגותם של רכיבים במערכת במקרים שונים. מכיוון שמאחורי כל התנהגות עומדות שורות קוד, המערכת מנסה, על פי בסיס חוקים שהוגדר לה מראש, להמיר את ההתנהגויות לשורות קוד הגורמות להתנהגות זהה.

לדוגמה, במידה וקלט נשלח בדף A אך מודפס חזרה רק בדף B, והתנהגות זאת נשנית רק במהלך SESSION ולא באופן קבוע, ניתן להסיק בסבירות גבוהה שדף A מכיל קוד הזהה בפעולתו לקוד הבא:

```
String input1 = request.getParameter("input1");
session.setAttribute("sessionValue1", input1 );
```

[דף המקבל ערך ומאחסן אותו ב-Session]

ואילו דף B מכיל קוד הזהה בפעולתו לקוד הבא:

```
out.println(session.getAttribute("sessionValue1"));
```

[דף המדפיס ערך שמאוחסן ב-Session, המושפע על ידי דף אחר]

מכיוון שהתנהגויות מסוימות עשויות להיגרם מכמה סוגים שונים של קוד, במידה וקיימת יותר מאפשרות אחת לייצוג התנהגות, המערכת מבצעת הצלבות ואימותים שמטרתן להעלות או להוריד את הסבירות לקיומן של שורות קוד שונות, כאשר בסופו של התהליך מוצגות שורות הקוד שכנראה רצות מאחורי הקלעים. לאחר המרה של מספר רב של התנהגויות לשורות קוד, עשויה להיווצר בעיה של סידור שורות קוד ברמת התצוגה (או במילים אחרות, לא נדע איזה שורות קוד באות קודם).

ניתן להתמודד עם בעיית סידור השורות באמצעות איסוף מידע על קדימות התנהגויות באפליקציה, ובאמצעות התקפות Layer Targeted ADoS - אשר מטרתן לעכב הרצה של שורות קוד ספציפיות. לדוגמה, אם אותר קוד שניגש לבסיס הנתונים, וקוד אשר בודק קלט ב-RegEx, ביצוע התקפה מסוג ReDos (אשר תעכב הרצה של השורה הרלוונטית) על בסיס בקשת HTTP שעל בסיסה הסקנו שמדובר בקוד שניגש לבסיס הנתונים, תאפשר לדעת איזה התנהגות מתרחשת קודם - העיכוב בזמן או ההתנהגות האחרת, מה שיאפשר לסדר את שורות הקוד בצורה ראלית יותר. יש לציין שאפשרות זאת עדיין אינה ממומשת ב-Diviner, ונכון להיום, כל שורת קוד מקבלת מיקום ברירת מחדל.

המרת התנהגויות למפת זכרון

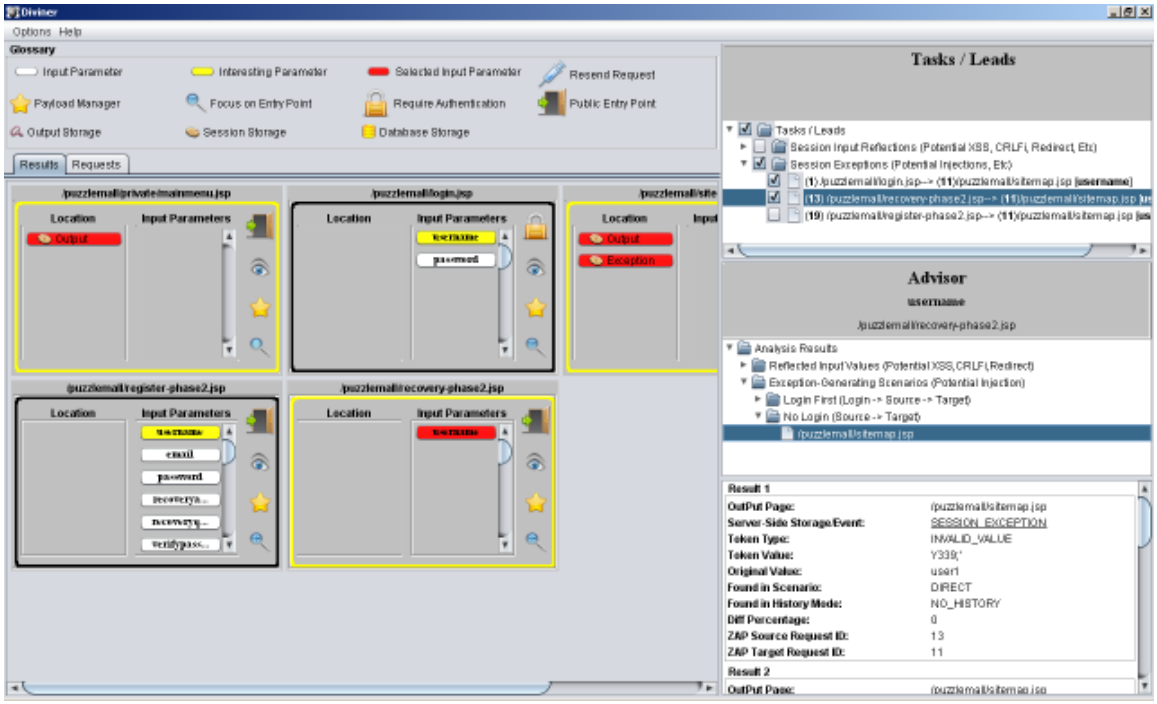
התנהגויות המצביעות על אחסון נתונים במאגר כלשהו בצד הלקוח או השרת (כגון Session, Database, Files וכדומה) יכולות להיות מתורגמות למפה של הזכרון בצד השרת. איתור מבנה הזכרון של השרת אפקטיבי במיוחד במידה ותהליך הלימוד איתר **השפעה עקיפה** של קלטים מדף אחד על דפים אחרים: מכיוון שהשפעה זו חייבת לעבור דרך מאגר משותף, נשאר רק לוודא שלא מדובר במקרה חד פעמי, ולנסות ולנתח מהו סוג מאגר המידע. ערכים שחיים רק בקונטקסט של Session יכולים להיות ערכים השמורים ב-Session Attributes, ב-Viewstate, או במקומות אחרים.

ערכים שחיים בקונטקסט קבוע יכולים להיות ערכים השמורים ב-Database, בקבצים, במשתנים סטטיים או במאגרי מידע נוספים. כמו במקרה של המרות התנהגויות לקוד - לאחר ביצוע הצלבות שונות ניתן להגיע לערכי Session המשותפים למספר דפים, שדות בטבלאות בסיסי נתונים המשותפים למספר דפים, טבלאות בבסיס הנתונים אשר מספר דפים עושים בהם שימוש, סוגי שאילתות, ועוד.

תצוגת ההשפעה של פרמטרים על רכיבי המערכת השונים

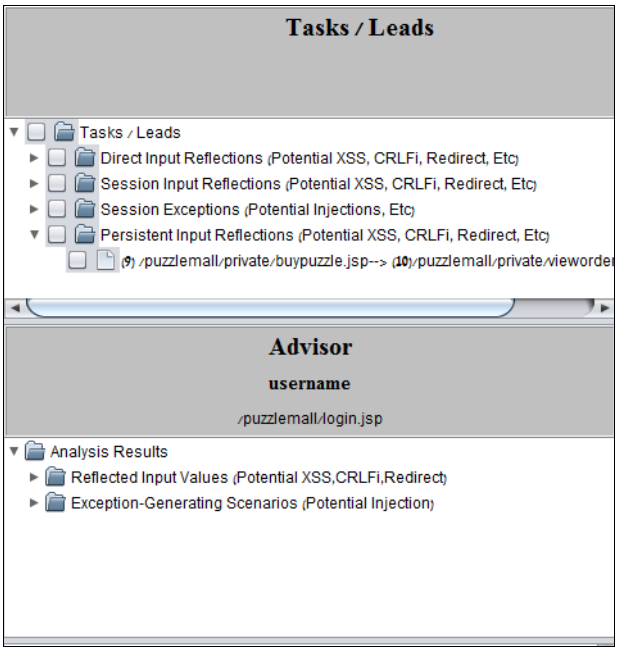
המפתח בהפקת תועלת מכמות האינפורמציה שאספנו **הוא התצוגה** (התנצלות מראש לחובבי ה-Command Line). הסקת מסקנות על נתונים שמוצגים בצורה ויזואלית קלה ומהירה הרבה יותר, ומאפשרת "פישוט" של האינפורמציה הטכנית. למטרה זו, Diviner מציג ממשק ויזואלי המכיל את כלל **הדפים המשפיעים ו/או המושפעים על ידי דפים אחרים**, בו כל לחיצה אחד הפרמטרים באחד הדפים מציגה על המפה אילו דפים מושפעים, באיזה תרחיש, ומה סוג ההשפעה (קלט חוזר, שגיאה, שינוי תוכן באחוזים, וכדומה).

לחיצה על פרמטר גם "מסננת" בתצוגה המטלות את התרחישים שרלוונטיים לפרמטר בלבד, ומציגה את הנתונים הדרושים לשחזור ההתנהגות (זיהוי, ערך ספציפי, וכדומה).



הצגת LEADS ו-TASKS באופן מרוכז, שימוש ב- Advisor לשחזור האירוע

במידה וכמות הדפים המשפיעים/מושפעים גדולה, או במידה והבודק רוצה להשקיע את המאמצים באיתור התקפות מסוג ספציפי, ההתנהגויות החשודות נאספות תחת קטגוריות התנהגות.



לחיצה על התנהגות ספציפית תציג מיד את פרטיה ברשימת המטלות, במפה הויזואלית וגם בפיצ'ר "היועץ" - פיצ'ר המכיל את כל הפרטים הדרושים לשחזור ההתנהגות.

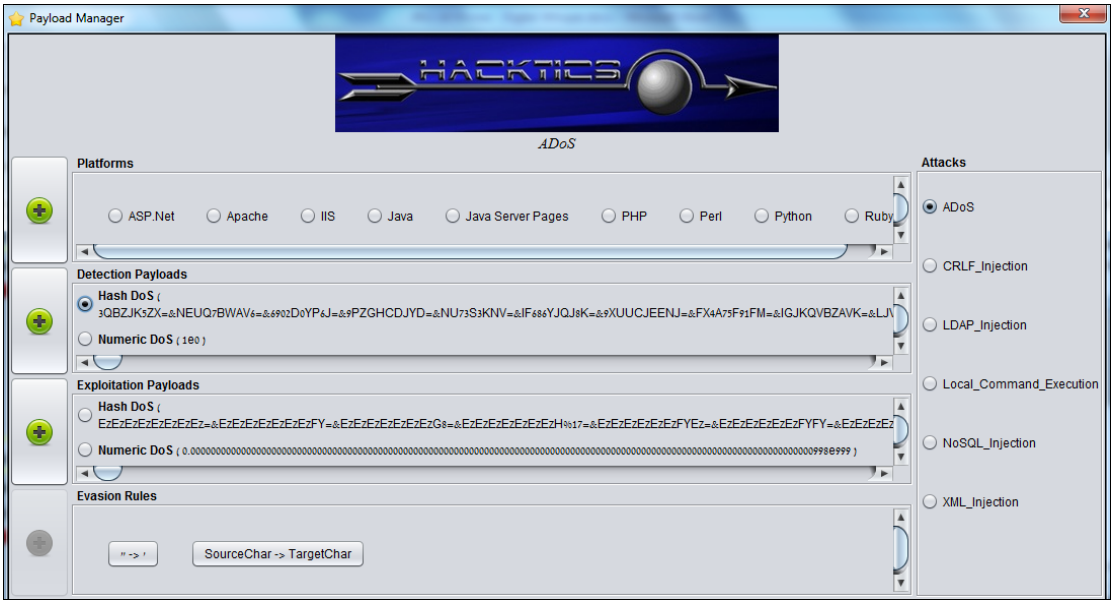
ניהול PAYLOADS והמלצות על קלטי תקיפה לפי התנהגות הרכיב

בנוסף ליתרונות הקודמים שהצגנו, ב-Diviner משולב היום פרויקט נוסף הנקרא Payload Manager. ההיגיון מאחורי Payload Manager פשוט:

יש יותר מידי התקפות ו-payloads בשביל לזכור הכל בעל פה (את כל התרחישים של LDAP Injection אתם זוכרים בעל פה? ומה עם EL Injection?). תהליך הבדיקה של התקפות אקזוטיות יותר יהפוך לפשוט עם כלי נוח עם שיאפשר לבחור את ה-Payload של התקפה שאנחנו רוצים לבצע, לערוך אותו במשק נוח ולצרף אותו לבקשה ב-Proxy לפני השליחה. יתרה מכך, במידה ולמדתם או קראתם על התקפה/פיילואד/טכניקת המרה מעניינת, תוכלו להוסיף אותה למאגר הפרטי שלכם, ולהשתמש בה שוב בלי לזכור אותה בעל פה.

הפרויקט תומך באחסון קלטי "איתור התקפות", "ניצול המרות" ובחוקי "המרה" (Evasion Rules). הוא גם מגיע מוכן עם פיילואדים למספר התקפות אקזוטיות יותר (כן, LDAP Injection נכלל), ובעתיד יכלול הרבה

יותר. ניתן להגיע למסך הבחירה על ידי בחירת פרמטר ולחצה על כפתור ה"כוכב", או על ידי הקלקה כפולה על פרמטר:



ניתן להשתמש בכלי הנ"ל על תוצאות הניתוח של Diviner, אך גם על כלל הבקשות בהיסטוריה של ZAP.

אינטגרציה עם ZAP, היורש של PAROS

לאחר הרכבת Payload ב-Payload Manager, ההרחבה תאפשר לכם לפתוח את חלון ה-"Resend" של ZAP (המקבילה של ה-Repeater של Burp), למקד אותה על הבקשה הרלוונטית, ולהעתיק אוטומטית את הקלט שהרכבתם לפרמטר המתאים.

פיצ'רי האינטגרציה ישופרו בעתיד ויאפשרו פעולות נוספות.

תכניות עתידיות

בעתיד הקרוב, יפתח באתר הבית של Diviner (http://code.google.com/p/diviner/) ויקי שיתעד את ההתנהגויות המומרות לשורות קוד, ואת הלוגיקה מאחורי ההמרה.

אנחנו מעודדים את הקהילה לתרום רעיונות הקשורים להתנהגויות מהם ניתן להסיק את קיומם של שורות קוד ספציפיות. מרגע העליה של הויקי (הודעה תופיע בטוויטר), **כל רעיון חדש שישלח ירשם על שמו של השולח.** אנחנו מעוניינים להפוך את רעיון ההמרה מכלי שימושי למדע של ממש, ולצורך העניין, שמחים לקבל עזרה מהוגי רעיונות או מפתחים (ישנה כתובת יצירת קשר באתר הפרויקט).

בנוסף לאינטגרציה עם פרויקט ה-Payload Manager, אנחנו בתהליך שיתוף פעולה עם 2 פרויקטי סורקים אוטומטיים, אשר בעתיד, נוכל לייצא להם את מאגר הרמזים שאיתר Diviner בפורמט XML, בכדי שסורקים אלו יוכלו לנסות ולסרוק את התרחישים המורכבים שהוא מאתר, ובפרט - תרחישי התקפה עקיפים הדורשים ממספר רב של תנאים להתקיים.

סיכום

פרויקט Diviner היינו פרויקט קוד פתוח המביא לשולחן יכולות לא שגרתיות שאינן נכללות היום בכלים אחרים בשוק, מסחריים או חינמיים. על אף העובדה שהפרויקט עדיין בשלבי Beta, מצבו יציב, ושימוש בו מאפשר כבר עכשיו איתור תרחישים שקשה מאוד לאתר באמצעים אחרים. עם הזמן, תרומה של מתנדבים בקהילה והשקעה שלנו תאפשר לנו להפוך את הפרויקט לכלי עזר שישפר את איכות הבדיקות של כולנו.

כשתהליך הפריצה הופך להיות ויזואלי, החזון ההוליוודי של פריצה עם צורות תלת מימדיות כבר לא כל כך רחוק...