Digital Whisper Technologic Papers

ניתוח קוד סטטי כדרך להתמודדות עם בעיות אבטחה בקוד Python

מאת יהונתן ויינברג

הקדמה

שפת התכנות Python היא שפה נפוצה מאוד בעולם, ונמצאת בשימוש ע"י חברות גדולות כמו Google, Dropbox, Reddit ועוד[1]. אך למרות השימוש הנפוץ בה, יש מספר גדול של בעיות אבטחה אפשריות שיכולות להיווצר בקוד הנכתב בה (כמו בכל שפת תכנות אחרת), ועל המפתחים המשתמשים בה לדעת על בעיות אלו ולדעת להתמודד איתן.

במאמר זה אתן מספר דוגמאות לבעיות אפשריות בקוד בשפת Python, אסביר על ניתוח קוד סטטי ועל יתרונותיו העיקריים, וכן אדבר על כלי הניתוח Bandit אשר מטרתו היא מציאת בעיות אבטחה בקוד בשפת Python בשיטת ניתוח קוד סטטי.

בעיות אבטחה בקוד בשפת Python

בעיית אבטחה בקוד מקור היא בעיה הנובעת מכתיבה לקויה של הקוד או מקרה מסוים בריצת הקוד עליו לא חשב המפתח, אשר מאפשר לגורם זדוני או לא רצוי לנצל זאת ולפגוע באופן פעולת הקוד, לגשת למידע שמפתח הקוד לא רוצה שיגיעו אליו או להשיג שליטה ברמה כלשהי על הסביבה (מחשב, שרת, מכונה וירטואלית ועוד) בה מורץ הקוד, במקרים מסוימים עד לרמה שהתוקף יכול להריץ איזה קוד שרירותי שהוא רוצה.

[1] https://realpython.com/world-class-companies-using-python/

דוגמאות לבעיות אבטחה אפשריות בקוד

להלן מספר דוגמאות לבעיות אבטחה אפשריות העלולות להיווצר בקוד בשפת Python:

שימוש ב-assertions לחסימת גישה

בעיית אבטחה אפשרית בקוד יכולה להיווצר כאשר במהלך הקוד נבדקות הרשאות גישה לקוד ע"י assert. זוהי בעיית אבטחה משום שבמהלך תהליכי ייעול הקוד של Python, כל שורות ה-assert מתבטלות, ולכן כל בדיקה שבוצעה ע"י assert פשוט לא תתבצע. לדוגמה, בקוד הבא:

```
def only_valid_users(user):
    assert user.isvalid, "user can't access this function"
    # Top Secret Stuff!!!
```

ישנה בדיקת גישה ע"י assert אשר המפתח מצפה שתעצור כל מי שאינו מקיים את הדרוש, אך במהלך ריצה רגילה של הקוד בה Python מבצע ייעול לקוד, שורת ה-assert תימחק לחלוטין ובעצם כל user יוכל לגשת לקוד המאובטח, בניגוד למה שהמפתח רצה שיקרה.

פתרון אפשרי לכך הוא שבדיקות מסוג זה יתבצעו ע"י if-else וכך לא ימחקו במהלך הייעול של Python ויוכלו לבצע את החסימה כראוי.

חולשות בחבילות צד שלישי

כידוע, ב-Python ניתן לייבא ספריות ולהוריד חבילות, ולהשתמש בכלים שהן מציעות, אך לא כל הספריות והחבילות אכן בטוחות לשימוש, ובעיקר כאשר הן מורדות או מיובאות מצד שלישי. קיימים גורמים זדוניים המעלים ל-PyPi חבילות עם שמות של חבילות נפוצות, אך במקום לעשות את מה שהחבילות הנפוצות עושות, הן מריצות קוד זדוני על המחשב של מי שמריץ אותן, וכך בעצם גורמות למשתמשים רבים להוריד חבילות מזויפות.

ניתן לפתור חולשות אלו ע"י הורדת חבילות או יבוא ספריות רק ממקורות מהימנים (או מהימנים ככל האפשר), בדיקת החתימות של הספריות והחבילות או פשוט לבצע כל שימוש בספריה או חבילה שלא בטוחה ב-100% בסביבה וירטואלית.

דריסת פונקציות או חבילות מיובאות

קיימים תרחישים בהם במהלך ריצת הקוד, מתבצעת גישה למחלקה מיובאת או לפונקציה מסוימת. גורם זדוני יכול לגשת למקום בו יושבת מחלקה זו או הפונקציה בה משתמש המפתח, ולשנותו כך שיתאים לרצונותיו ולא יבצע את מה שהמפתח מצפה שהפונקציה תבצע, או שפשוט לא יעבוד.

לדוגמה, נסתכל על הקוד הבא:

```
f = open("1.txt", "w")
f.close()
f.write("This is a test.")
```

קוד זה, ללא הוספת קוד אחרת, אמור לזרוק שגיאה בשורה השלישית, משום שהקובץ נסגר ולכן לא ניתן לכתוב לתוכו. גורם זדוני בעל גישה לזיכרון המחשב יכול להוסיף לקובץ או לספריות Python המותקנות על המחשב את הקוד הבא:

```
def open(s1, s2):
    return F(s1, s2)  # not what the function was planned to do

class F:
    def __init__(self, s1, s2):
        j = 5  # not what the function was planned to do

    def write(self, s):
        print("Not what you expected!")  # not what the function was planned to do

    def close(self):
        i = 1  # not what the function was planned to do
```

הוספת קוד זה תגרום לדריסת המימוש המקורי של המחלקה, ולכן הקוד המקורי ירוץ ללא בעיות, בניגוד למה שהמפתח ציפה (ומה שאמור היה לקרות לולא הוספת הקוד).

ניתן להתמודד עם בעיית אבטחה זו ע"י ביצוע Hash על הפונקציות והמחלקות החשובות לפני השימוש בהן ושמירתו, ובמהלך הריצה לבצע שוב Hash על הפונקציות ולהשוות את התוצאה לזו המוקדמת ולפי זה לדעת האם אנחנו קוראים לפונקציה המצופה או לפונקציה דורסת. האמנם גם פתרון זה אינו שלם משום שקיים מצב בו התוקף יכול לדרוס את הפונקציה של חישוב ה-Hash ובכך לתת תוצאות מזויפות או לתת רק התוצאות המצופות, אך לא ארחיב על כך במאמר זה.

שימוש ב-eval או exec

הפונקציות eval ו-exec ב-Python משמשת להרצת המחרוזת אשר הין מקבלות בפרמטר כקוד רגיל, כאשר eval יכולה להריץ שורה אחת בלבד שאינה מכילה import, ו-exec יכולה להריץ מס' שורות אשר יכולות להכיל בין היתר גם import.

פונקציות זו כשלעצמן הן פונקציות לא בטוחות, מאחר ובעזרתן ניתן להריץ כל קוד על המחשב. ניתן לפתור זאת ע"י בדיקת הפרמטר שלהן לפני הרצתן, אך נראה כעת דוגמה בה הבדיקה לא עובדת:

```
def run_the_string_without_import(code):
    if "import" in code:
        print("Import is not allowed!")
    else:
        exec(code)
```

לכאורה נראה כי הקוד אכן מבצע את הדרוש, וחוסם כל שימוש ב-import בקוד, אך הנה דוגמה לשימוש שכן משתמשת ב-import ואינה נחסמת:

```
codeToRun = "exec('impo' + 'rt os')\nprint('Imported!')"
run_the_string_without_import(codeToRun)
```

בהרצת שורות אלו, אנו מתגברים על החסימה בעזרת שימוש נוסף ב-exec ובעצם לא קוראים ישירות ל-import אלא באופן שהחסימה לא יכולה לגלות, ואכן קוד זה מדפיס במהלך הרצתו !Imported ללא שגיאות, פעולת ה-import מתבצעת בניגוד לרצון מפתח הקוד.

דרך להתמודד עם זה היא לבצע חסימה נוספת של שימוש ב-eval או ב-exec וכך לבטל אפשרות של שימוש פנימי בהן.

בעיות תזמון

קיימים קטעי קוד אשר מבצעים בדיקה כלשהי מרובת חלקים או מספר בדיקות, וכל אחת מהבדיקות מוסיפה לזמן הריצה, למשל הקוד הבא:

```
def checkvalid(code):
    if code[0] == 1:
        if code[1] == 5:
            if code[2] == 3:
                print("Welcome!")
            else:
                print("Access Denied")
        else:
            print("Access Denied")
    else:
        print("Access Denied")
```

בקוד הבא ניתן לראות בדיקת רשימה המכילה קוד בן 3 ספרות, ועל מנת "להיכנס" יש להזין את הקוד הרצוי. גורם זדוני המשתמש בקוד יכול לבצע בדיקות מרובות ועבור כל בדיקה לבדוק כמה זמן היא לקחה, ולפי זה לדעת האם הוא מצא חלק מהספרות או לא.

לדוגמה, אם הוכנסה הרשימה [1,2,3] אז זמן הריצה של התוכנית יהיה ארוך יותר מאשר אם הייתה מוכנסת הרשימה [2,3,4], כי עבור הרשימה הראשונה התוכנית נכנסת אל ה-if השני לעומת הרשימה השנייה שבה הקוד יעצר כבר ב-if הראשון, ולפי זה גורם זדוני או לא רצוי מסוגל למצוא את כל הספרות לאחר כמות של ניסיונות.

דוגמה נוספת היא מתקפת ערוץ צדדי (Side Channel), התקפה קריפטוגרפית המנצלת את מנגנון ההצפנה של יעד המתקפה ולא מבצעת זאת בכוח גס (Brute force). אם בקוד קריפטוגרפי מסוים קיימת בעיית תזמון במנגנון ההצפנה, תוקף פוטנציאלי יכול לנתח את זמני ריצת התוכנית ולפי זה לקבל מידע על מפתח ההצפנה או על התוכן המוצפן. תוכלו לקרוא עוד על מתקפה זו במאמר הבא, שנכתב ע"י יובל סיני ופורסם ב-DigitalWhisper.

פתרון אפשרי לבעיית אבטחה זו הוא להשתמש ב-sleep של הספרייה המובנית time ובטיימרים של הספרייה וכך לתזמן את זמן הריצה כך שכל ריצה תיקח זמן קבוע, ללא תלות בקלט, וכך לא יהיה אפשר לפענח את הצופן ע"י תזמון.

ישנן כמובן עוד בעיות אבטחה רבות מאוד בנושאים שונים ומגוונים, ולרוב אנו מגלים רק חלק קטן מהן וגם לא בהכרח מצליחים לפתור אותן, אך עדיין יש חלק גדול שקשה מאוד למפתח ממוצע לגלות אותו לבד. על מנת למצוא כמה שיותר בעיות אבטחה בקוד ולצמצם ככל הניתן את הפתחים והפרצות, נכתבו כלים רבים אשר מבצעים סריקה של הקוד בשיטת "ניתוח קוד סטטי".

בהמשך המאמר אדבר על הכלי Bandit אשר מבצע ניתוח קוד סטטי לקוד בשפת Python ומיועד לגלות בו פרצות ובעיות אבטחה להודיע עליהן למפתח, אך לפני כן בואו נבין מהו ניתוח קוד סטטי ולמה שנבצע ניתוח קוד סטטי לקוד מקור.

ניתוח קוד סטטי

מהו ניתוח קוד סטטי?

ניתוח קוד סטטי (Static code analysis) זהו תהליך אוטומטי המיועד לסריקת ובדיקת קוד מקור הבוחן את התנהגות הקוד ומנסה למצוא בעיות ובאגים מכניים בקוד ללא הרצתו, ומודיע למתכנת על בעיות אלו ועל סיכויי ההשפעה שלהן על תקינות הקוד.

ניתוח הקוד הסטטי יכול להיות שימושי במיוחד בזכות מאפיינו העיקרי שמבדיל אותו מניתוח קוד דינאמי- הוא לא צריך להריץ את הקוד עליו הוא עובר כדי לזהות בו בעיות. זאת, בניגוד לניתוח קוד דינאמי, אשר כחלק מהסריקה של קוד המקור, הוא מבצע הרצה מלאה או חלקית של הקוד.

ניתוח הקוד הסטטי שימושי למשל בקודים הרצים על שרתים אינטרנטיים או בתוכנות גדולות במיוחד, כי אז הרצה "עולה" הרבה (זמן ריצה ארוך במיוחד במקרה הטוב) וניתוח הקוד הסטטי חוסך זמן ריצה יקר ועוזר לגלות את הבעיות ללא הרצת השרת או התוכנה הארוכה.

בחלק זה אדבר בעיקר על היכולות והיתרונות של ניתוח הקוד הסטטי, ובהמשך אראה דוגמה לכלי קיים כזה. קיים מאמר נוסף במגזין על ניתוח קוד סטטי שאני ממליץ על קריאתו.

יכולות אפשריות של ניתוח הקוד הסטטי

קיימות מספר בעיות ושגיאות שניתוח הקוד הסטטי מסוגל לגלות (כתלות בכלי הניתוח, חלק מהכלים ימצאו בעיות כאלה ואחרות שכלים אחרים לא בהכרח):

ישנם מקרים בהם הקוד מבצע שימוש מתמשך במשאב מערכת מסוים, לדוגמה ב-CPU (ביצוע פעולה מסוימת) , בזיכרון (הקצאת זיכרון לשמירת אובייקטים) או בהתקן I/O מסוים (קליטה מהמשתמש,

הדפסה וכו'), ולאחר גמר השימוש בהם הוא אינו "סוגר את המשאב" ומאפשר למשאב להתפנות למשימה הבאה, כלומר מייצר דליפה במשאבי המערכת.

דליפה זו יכולה להפריע לפעילותו התקינה של הקוד, כלומר אם בהמשך הקוד יש שימוש נוסף במשאב, אז המשאב כבר תפוס והקוד קורס או לא מבצע את עבודתו כראוי. ניתוח הקוד הסטטי יודע לזהות את הדליפה ואת מיקומה (הזיכרון ששאר תפוס למרות שאין צורך בו עוד, שורת הקוד שפונה למשאב I/O ואינה "משחררת" אותם, וכו') ולהתריע על כך לכותב הקוד כדי שיבצע את שחרור המשאב.

בחלק מהמקרים, ניתן להתייחס לסוג בעיות זה כאל פרצת אבטחה, משום שמשתמש זדוני בקוד יכול לבצע תקיפת DoS ע"י כך שהוא ינצל את דליפת המשאבים ויבצע עוד ועוד פעולות עד אשר לא ישארו עוד משאבים, ואז המחשב עליו רץ הקוד לא יוכל לתפקד ולחלק את המשאבים לגורמים אחרים. פגיעה זו יכולה להיות קריטית אף יותר אם מדובר בשרת עליו רץ הקוד.

בנוסף לכך, לעיתים כותב הקוד מבצע טעות בכתיבת הקוד וגורם להמצאות חור אבטחה- כלומר פותח אפשרות למשתמש לבצע שימוש שלא כראוי בקוד ובכך לגשת למידע שאינו מורשה להגיע אליו ולעיתים אף לשנותו או למחקו. ניתוח הקוד הסטטי מסוגל לזהות את הפרצות לפני הרצת הקוד ולהתריע עליהן למשתמש, ובכך למנוע את מימוש האפשרות שנפתחה למשתמשים ע"י טעות הכותב.

מעבר לכך, במהלך ריצת הקוד יכולים לקרות מקרים בהם מספר תהליכים בקוד ניגשים למשאב מסוים של המחשב והחומרה (CPU, I/O, זיכרון ועוד) באותו זמן, והמשאב לא יכול לשרת את כולם אז נוצר מצב שמי שניגש אל המשאב ראשון יכול להשתמש בו ומי שאיחר את המועד צריך לחכות עד שיתפנה, וזה גורם לבזבוז זמן במהלך ריצת הקוד. ניתוח הקוד הסטטי יודע לזהות בעיה זו לפני הרצת הקוד ולהתריע עליה לכותב, ובכך למנוע את מרוץ התהליכים ואת בזבוז הזמן.

בנוסף לאלו, לעיתים כותב הקוד מבצע טעות הגורמת לשגיאת זיכרון כמו אי אתחול משתנים או חריגה מגבולות מערך נתון, ומובן שהקוד קורס בזמן הריצה ולא יכול להמשיך במקרה כזה. ניתוח הקוד הסטטי מסוגל לזהות שגיאות זיכרון אלו או חריגות אלו מבלי להריץ את הקוד ובכך לחסוך שגיאה בזמן הריצה.

יתר על כן, ישנם מקרים בהם חלק מהקוד לא נגיש, כלומר בכל מצב שהוא חלק זה של הקוד לא יורץ (לדוגמה בלוק if שהתנאי בו תמיד False) או משתנה שאין בו שימוש במהלך הקוד והוא קיים ללא סיבה. ניתוח הקוד הסטטי יודע להתריע לכותב הקוד על משתנה זה או על חלק הקוד שלא יורץ לעולם ובכך לחסוך מקום בזיכרון ולקצר את הקוד.

יש לשים לב כי בשפות אשר יש להן מהדר (Compiler) אז הוא יודע לנקות משתנים או קטעי קוד לא נגישים באופן עצמאי במהלך תהליך ההידור (Compilation) כך שבשפות אלו ניתוח הקוד הסטטי לא ימצא כנראה קטע קוד לא שמיש. לעומת זאת, בשפות סקריפטים אשר אינן משתמשות במהדר, חלקי קוד אלו לא ינוקו בנפרד ולכן אם אכן ישנם כאלה, ניתוח הקוד הסטטי יוכל למצוא אותם ולהגיד עליהם למתכנת.

מעבר לאלו, ניתוח הקוד הסטטי יודע לזהות גם חריגות (Exceptions) מסוימות בקוד לפי תבניות קבועות מראש המוגדרות בסריקה (בכל כלי לניתוח קוד סטטי יש תבניות שונות לפיהן הוא מוצא חריגות), אי עמידה בסטנדרטים של כתיבת קוד (תלוי בסטנדרטים של שפת הקוד הנסרק) ועוד.

אך לעומת כל אלו, לניתוח הקוד הסטטי יש גם מגבלות- אי הרצת הקוד גורמת לבעיות רבות לא להתגלות ולכן לא ניתן להסתמך במהלך בדיקת קוד רק על ניתוח קוד סטטי, אלא יש להשתמש בכלים נוספים לניתוח קוד שכן מריצים את הקוד ומוצאים שגיאות זמן ריצה. ומכאן, שכדאי ראשית לבצע ניתוח קוד סטטי ורק לאחר סיומו ושליחת התוצאות לכותב הקוד, לבצע גם ניתוח קוד דינאמי (הכולל הרצה של הקוד ומציאת שגיאות זמן ריצה).

יתרונות וחסרונות של ניתוח הקוד הסטטי

לניתוח הקוד הסטטי ישנם מספר יתרונות נוספים, ולהלן העיקריים שבהם:

ניתוח הקוד הסטטי פועל בצורה אוטומטית והמתכנת שבודק את הקוד צריך להפעיל את הניתוח, ומאותו רגע הסריקה והסקת המסקנות ממנה מתבצעת באופן אוטומטי לחלוטין וללא התערבות המתכנת בניתוח.

בנוסף, ניתוח הקוד הסטטי מוצא באגים ובעיות בקוד באופן מהיר יותר וזול יותר מניתוחי קוד דינאמיים- ניתוחים הכוללים הרצה מלאה או חלקית של הקוד ולא רק סריקה שלו, מכיוון שהרצת הקוד דורשת משאבי מערכת וזמן מעבד יקר שניתוח הקוד הסטטי חוסך, ובכך הניתוח חוסך זמן רב ומבצע את הסריקה ואת הסקת המסקנות עם גישה מועטה למשאבי המערכת.

כמו כן, ניתוח הקוד הסטטי עובד לפי תבניות - הוא מחפש ומוצא שגיאות לפי תבניות מוגדרות מראש, כלומר הוא יכול למצוא שגיאות מסוגים מסוימים שקיימת תבנית מסוימת אשר השגיאה משתייכת אליה, וזה ממקד את הניתוח ועוזר למתכנת למצוא באגים רק מסוגים מסוימים. התאמת הניתוח לחיפוש שגיאות לפי תבניות מוגדרות מראש תלויה בכלי אשר המתכנת משתמש בו לביצוע הסריקה, משום שלא כל כלי מכיל את כל התבניות האפשריות, וגם לא כל הכלים יכולים להתאים את הניתוח לחיפוש בעיות בקוד לפי תבניות מסוימות.

מעבר לכך, ניתוח הקוד הסטטי יודע להצביע על מיקום השגיאה בקוד - הוא יודע להגיד איפה בקוד ישנה בעיה או ישנה שורת הקוד הבעייתית מבחינה מסוימת, ולא רק להתריע על השגיאה ולהגיד למתכנת על קיומה בלבד, ובכך הוא עוזר למתכנת לתקן את השגיאות והבעיות בקוד בצורה נוחה יותר.

לניתוח הקוד הסטטי ישנם גם מספר חסרונות

כידוע, ניתוח הקוד הסטטי מחפש ומוצא שגיאות לפי תבניות מסוימות קבועות מראש, אך זה גם חיסרון כי זה אומר שישנן שגיאות מסוימות שאם הן לא משתייכות לאחת התבניות של הניתוח, הוא לא יהיה מסוגל לזהות שגיאות אלו, ללא קשר לסוגן או למידת חומרתן והשפעתן על תקינות הקוד.

בנוסף לכך, מכיוון שניתוח הקוד הסטטי אינו מריץ את קוד המקור, הוא אינו יכול לזהות כלל שגיאות זמן ריצה- כלומר שגיאות ובאגים שניתן לגלותם רק במהלך שהקוד מורץ ולא לפני כן. בעקבות חיסרון זה, מומלץ להשתמש בכלי ניתוח דינאמיים בנוסף לשימוש בכלי ניתוח קוד סטטי, וזאת על מנת לגלות את כל הבאגים והבעיות הקיימים בקוד (במידת היכולת של הכלים כמובן), לפני הריצה ובמהלכה.

מעבר לכך, מכיוון שניתוח הקוד הסטטי אינו מריץ את קוד המקור אותו הוא סורק ורק מנסה לחזות בעיות אפשריות ולהתריע עליהן למתכנת, לעיתים ישנן שורות קוד או ישנם קטעי קוד אשר הניתוח מזהה בטעות ככאלה שעלולים לגרום לבעיה אפשרית או ככאלה הגורמים לסיכון בטיחותי כלשהו, אך הם לא בעייתים כלל ואינם גורמים לבעית אבטחה בקוד, והתרעות אלו עלולות להפריע למתכנת ולהטריד אותו ואף לבלבלו למרות שאין כלל בעיה בקוד.

דרכי מימוש

קיימות כל מיני דרכים בהן ניתוח הקוד הסטטי ממומש. הניתוח יכול להיות ממומש כחלק מ-IDE, וכחלק מעורך הקוד הניתוח רץ באופן אוטומטי לאחר כל שינוי או בכל הרצה ומתריע למתכנת בזמן אמת על בעיות ובאגים אפשריים בקוד שלו (זה כמובן משתנה בין עורכי הקוד השונים, כל עורך מתנהג בצורה שונה בהתאם למוגדר אצלו).

דרך נוספת בה הניתוח יכול להיות ממומש היא ככלי נפרד מעורך הקוד- כלי אשר מצריך התקנה מיוחדת ובסיום כתיבת הקוד המתכנת צריך לתת לכלי המסוים את המיקום בו נמצא קובץ הקוד או את הקובץ עצמו, והכלי יודע לבצע את ניתוח הקוד הסטטי ולהחזיר למתכנת משוב על הקוד או להתריע על בעיות ובאגים אפשריים. כעת אדבר על כלי כזה אשר עובד על שפת Python - כלי ששמו Bandit אשר מבצע ניתוח קוד סטטי לקוד בשפת Python, ומתמקד בבעיות אבטחה אפשריות בקוד.

הכלי Bandit

הכלי Bandit הוא כלי ניתוח של קוד Python אשר מטרתו היא למצוא בעיות אבטחה בסיסיות בקוד ודיווח עליהן למשתמש. Bandit פותח בהתחלה תחת OpenStack Security Project, ולאחר מכן עבר להיות מפותח תחת PyCQA. הקוד מאחורי כלי זה הינו קוד פתוח, כלומר הקוד שלו חשוף לציבור וכל אחד יכול לגשת לקוד ולצפות בו, וקוד הפרויקט נמצא ב-Github.

קישור לפרויקט: https://github.com/PyCQA/bandit

כאשר אנו מריצים את Bandit על תיקייה כלשהי, הוא הופך כל אחד מהקבצים למבנה שנקרא -AST Abstract Syntax Tree, שזה עץ תחבירי אשר מציג בצורה מפושטת את הקוד. לאחר הפיכת הקוד ל-AST, הכלי מריץ תבניות שונות ובדיקות שונות על העץ במטרה למצוא בעיות אבטחה בקוד, כאשר

המשתמש יכול להחליט אילו תבניות יורצו על העץ ואילו לא. עבור כל בעיה ש-Bandit מוצא, קיים סיווג לפי 2 פרמטרים (Severity, Confidence) ושלוש רמות לכל פרמטר (Low, Medium, High).

קיימות 7 קטגוריות של בדיקות ש-Bandit מבצע על הקוד, להלן העיקריות שבהן:

סימון	קטגוריה	תיאור
B1xx	misc tests	בעיות מסוגים שונים שאינן משוייכות לאחת מהקטגוריות האחרות. לדוגמה, בדיקת שימוש ב-assert, בדיקת שימוש ב-exec, בדיקת שימוש ב-try-except-pass, ועוד.
B3xx	blacklists (calls)	שימוש בפונקציות מתוך רשימה של פונקציות אשר יכולות להיות לא בטוחות במקרים מסוימים. לדוגמה, שימוש ב-input ב-Python 2.x, שימוש בפונקציות מהספריה telnetlib, שימוש ב-eval, שימוש בפונקציות מהספריה ftplib ועוד.
B4xx	blacklists (imports)	יבוא ספריות הנחשבות לא בטוחות או עלולות להיות לא בטוחות במקרים מסוימים. לדוגמה, יבוא הספריה telnetlib, יבוא הספריה ftplib, יבוא הספריה pickle, יבוא חלק מתת הספריות של Crypto ועוד.
B5xx	cryptography	בעיות העלולות להיווצר במהלך עבודה עם הצפנה, לדוגמה בחירת מפתח קריפטוגרפי חלש מידי, פתיחת תקשורת ב-SSL או TLS ללא בדיקת אישורים (certificates), ועוד.
B6xx	injection	בעיה העלולה להיווצר מקלט מסוים אשר יכול לגרום לתוכנית להריצו או לשנות את אופן פעולתה, לדוגמה הכנסת שאילתת SQL לקלט שיורץ או קלט שיוכנס לתוך שאילתת SQL כשדה או שם כלשהו.

דוגמת הפעלה והדפסה

כעת נסתכל על דוגמה פשוטה של הרצת Bandit על קובץ Python ונראה את ההדפסה בסיום ההפעלה- כיצד Bandit מדווח למשתמש על החולשות וכיצד הוא מסווג אותן. להלן קובץ Python פשוט:

```
exec("print('BandiTest')")
```

הקובץ מכיל שורת קוד אחת בה ישנה קריאה לפונקציה exec, ובמהלך הרצת הקובץ הקוד ירוץ ללא בעיות ויודפס "BandiTest".

כעת, נרצה להפעיל את Bandit על קובץ זה ולראות את ההדפסה. לאחר כתיבת הפקודה:

```
bandit pytry.py
```

יודפס הפלט הבא:

```
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.6.9
[node_visitor]  INFO    Unable to find qualified name for module: pytry.py
Run started:2020-08-04 11:36:52.897941

Test results:
>> Issue: [B102:exec_used] Use of exec detected.
   Severity: Medium   Confidence: High
   Location: pytry.py:1
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b102_exec_used.html
1       exec("print('BandiTest')")

--------------------------------------------------

Code scanned:
        Total lines of code: 1
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 0.0
                Medium: 1.0
                High: 0.0
        Total issues (by confidence):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 1.0
Files skipped (0):
```

כפי שניתן לראות בהדפסה, תחילה מודפס מידע בנוגע לריצה של Bandit, לאחר מכן מודפסות כל הבעיות שנמצאו בהרצה, מיקומן בקוד ורמת הדחיפות והחומרה שלהן (במקרה זה בשורה 1 נמצא שימוש ב-exec), ולבסוף סכימה של הבעיות לפי סיווגן.

כעת, אם נשנה את הקובץ כך שתוכנו יהיה זה:

```
exec("print('BandiTest')")
print(input("Enter something..."))
```

הרצתו מחדש תבצע את ההדפסה "BandiTest", ולאחר מכן התוכנית תקבל קלט מהמשתמש ותדפיסו (בהרצה בגרסת Python 3.x ומעלה). הפעלה מחודשת של Bandit על הקובץ תניב את הפלט הבא:

```
[main]  INFO    profile include tests: None
[main]  INFO    profile exclude tests: None
[main]  INFO    cli include tests: None
[main]  INFO    cli exclude tests: None
[main]  INFO    running on Python 3.6.9
[node_visitor]  INFO    Unable to find qualified name for module: pytry.py
Run started:2020-08-05 11:55:30.079860

Test results:
>> Issue: [B102:exec_used] Use of exec detected.
   Severity: Medium   Confidence: High
   Location: pytry.py:1
   More Info: https://bandit.readthedocs.io/en/latest/plugins/b102_exec_used.html
1       exec("print('BandiTest')")
2       print(input("Enter something..."))

--------------------------------------------------
>> Issue: [B322:blacklist] The input method in Python 2 will read from standard input, evaluate and run
the resulting string as python source code. This is similar, though in many ways worse, then using eval.
 On Python 2, use raw_input instead, input is safe in Python 3.
   Severity: High   Confidence: High
   Location: pytry.py:2
   More Info: https://bandit.readthedocs.io/en/latest/blacklists/blacklist_calls.html#b322-input
1       exec("print('BandiTest')")
2       print(input("Enter something..."))

--------------------------------------------------

Code scanned:
        Total lines of code: 2
        Total lines skipped (#nosec): 0

Run metrics:
        Total issues (by severity):
                Undefined: 0.0
                Low: 0.0
                Medium: 1.0
                High: 1.0
        Total issues (by confidence):
                Undefined: 0.0
                Low: 0.0
                Medium: 0.0
                High: 2.0
Files skipped (0):
```

כפי שניתן לראות, התווסף להדפסה דיווח על בעיה אפשרית בשורה 2, אשר אומרת שהשימוש ב-input אינו בטוח אם נריץ את הקובץ על Python 2.x, והשימוש יהיה בטוח אם נריץ את הקוד ב-Python 3.x, וכן מתווסף הסיווג של הבעיה לסיכום הבעיות.

דוגמאות לבעיות אפשריות

כעת נסתכל על שתי דוגמאות לבעיות אבטחה ש-Bandit מסוגל היה למצוא:

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2019-5729

בעיית אבטחה זו נמצאה ב-Splunk SDK For Python, סביבת עבודה של קוד Python, בה ניתן לעבוד ישירות עם Splunk- מערכת המאפשרת לעבוד עם Big Data.

בעיית האבטחה היא שבמהלך העבודה עם שרתי TLS, לא התבצעה בדיקה מספיק טובה של ה-certificates, מה שיכול היה לאפשר מתקפת man-in-the-middle.

אחת הבדיקות ש-Bandit מבצעת היא B501: request_with_no_cert_validation, אשר בדיוק בודקת את האפשרות לחולשה זו ומוודאת שבעבודה עם TLS Server ישנה בדיקה תקינה של ה-certificates, ולכן ככל הנראה אם על הקוד אשר מבצע את הבדיקת certificates היתה מורצת בדיקה של Bandit, הכלי היה מזהה את החולשה ומתריע עליה, והיה אפשר לתקנה.

בעיה נוספת הינה הבעיה הבאה:

https://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-1887

בעיית אבטחה זו נמצאה בפונקציה PyString_FromStringAndSize ב-Python, והחולשה הייתה שהפונקציה עשתה שימוש ב-assert, ומאחר שבאופטימיזציה של Python כל קריאות ה-assert נמחקות, הוקצב פחות מקום מהצפוי לפונקציה, ועבור ערך שלילי בקלט שלה היא גרמה לחריגת Buffer, מה שפתח פתח להשתלת קוד והרצת קוד שהתקבל כקלט על המחשב בו הורץ הקוד.

אחת הבדיקות ש-Bandit מבצעת היא B101: assert_used, אשר בודקת האם נעשה שימוש ב-assert ומדווחת על כך למפתח, ולכן אם על הפונקציה המדוברת הייתה מורצת בדיקה של Bandit, ככל הנראה הכלי היה מזהה את החולשה ומודיע עליה למפתחים, וכך היה אפשר לתקנה.

סיכום

במאמר זה סקרתי רק חלק קטן מהעולם הרחב של בעיות האבטחה בקוד בשפת Python, אך מטרת מאמר זה היא לסקור את הנושא ולהסביר אותו באופן כללי בעזרת דוגמאות לבעיות אבטחה ופתרונות מתאימים.

במאמר זה ראינו דוגמאות לבעיות אבטחה בקוד Python ע"י הצגת הבעיה ופתרון אפשרי לבעיה, למדנו מהו ניתוח קוד סטטי, על היכולות האפשריות שלו ,ועל היתרונות והחסרונות שלו, ולמדנו על כלי הניתוח Bandit אשר מטרתו למצוא חורי אבטחה אפשריים בקוד Python וראינו דוגמאות הפעלה שלו על קוד פשוט, וכן דוגמאות לבעיות אבטחה ש-Bandit יכול היה לגלות.

קצת עליי

שמי יהונתן ויינברג, בן 15.5 עולה לכיתה י"א בקריית החינוך ע"ש דוד בן גוריון בעמק חפר, לומד בנוסף לתיכון בתוכנית אודיסיאה באוניברסיטת תל אביב במסלול סייבר, בה אני עולה לשנה ג', מתעניין במחקר CyberSecurity.

ברצוני להודות לשלומי בוטנרו, ראש מסלול הסייבר באודיסיאה באוניברסיטת ת"א על העזרה, התמיכה וההערות המעשירות במהלך כתיבת המאמר.

אשמח מאוד לקבל שאלות ומשוב על המאמר במייל:

yon2005w1@gmail.com

לקריאה נוספת

- https://www.digitalwhisper.co.il/files/Zines/0x34/DW52-3-SCA.pdf
- https://www.digitalwhisper.co.il/files/Zines/0x4F/DW79-2-SideChannel.pdf

ביבליוגרפיה

- https://hackernoon.com/10-common-security-gotchas-in-python-and-how-to-avoid-them-e19fbe265e03
- https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=Python
- https://itnext.io/common-python-security-problems-ffedbae7b11c
- https://wiki.openstack.org/wiki/Security/Projects/Bandit
- https://bandit.readthedocs.io/en/latest/
- https://github.com/PyCQA/bandit
- https://pypi.org/project/bandit/
- https://en.wikipedia.org/wiki/Abstract_syntax_tree
- https://huskyci.opensource.globo.com/
- https://he.wikipedia.org/wiki/%D7%A0%D7%99%D7%AA%D7%95%D7%97_%D7%A7%D7%95%D7%93_%D7%A1%D7%98%D7%98%D7%99
- https://en.wikipedia.org/wiki/Static_program_analysis
- https://luminousmen.com/post/python-static-analysis-tools
- https://blog.codacy.com/everything-you-need-to-know-about-static-code-analysis/
- https://sdtimes.com/test/5-ways-static-code-analysis-can-save-you/
- https://he.wikipedia.org/wiki/%D7%94%D7%AA%D7%A7%D7%A4%D7%AA_%D7%A2%D7%A8%D7%95%D7%A5_%D7%A6%D7%93%D7%93%D7%99
- https://realpython.com/world-class-companies-using-python