Digital Whisper
Technologic Papers

מחקר ותקיפת אפליקציות אנדרואיד באמצעות Frida

מאת בניה

הקדמה

בגיליון 154, הכותב עידן שכטר פרסם מאמר בשם: "מבוא למחקר אפליקציות" בו הראה כיצד הוא חוקר את אפליקציית Shazam Lite ומבצע עליה מניפולציות (זיוף ערכי פונקציות שמחזירות את שם מפעיל הרשת וקוד המדינה) כדי להצליח להריץ אותה.

מטרת מאמר זה הינה להציג שימושים נוספים ל-Frida כגון עקיפת מנגנוני הגנה מפני root, עקיפת מסכי הזדהות, עקיפת הגנה מ-Emulators וחשיפת סיסמאות השמורות באפליקציה.

מה זה Frida?

Frida היא כלי שמאפשר מניפולציות בזמן ריצה על אפליקציות של מגוון רחב של פלטפורמות, בין השאר על אנדרואיד. באמצעות Frida נוכל לצפות במידע פנימי רגיש של האפליקציה, לשנות התנהגות של פונקציות וערכים של משתנים, לקרוא לפונקציות שלא אמורות להיות מופעלות ועוד מגוון רחב של שימושים.

נדגים את היכולות של הכלי במאמר זה בכך שנעקוף מנגנוני הגנה באפליקציות אנדרואיד, נשנה נתונים בתוך האפליקציה, ונחשוף סיסמאות שמאוכסנות באפליקציה.

הדרך הנפוצה ביותר של עבודה עם Frida כוללת כתיבת סקריפט בשפת Javascript שמכיל את ההוראות לביצוע על ידי Frida - וטעינתו לתוך אפליקציה (בזמן ריצה או סטטית).

למידע נוסף על דרך פעולת הכלי ושימושיו אני ממליץ על המאמר "מבוא למחקר אפליקציות" ועל העמוד Android Hooking in Frida.

איך עובדים עם Frida?

לכלי יש שני מצבים עיקריים:

- **Injected** - במצב זה, מותקן על המכשיר Frida Server שהוא מעין שרת המקבל פקודות מהלקוח (המחשב שעליו אנו עובדים) ומבצע אותן. פקודות אלו יהיו למשל להזריק קוד לתוך אפליקציה, לבחון את האפליקציות המותקנות או את התהליכים שרצים על המכשיר. מצב זה הוא הנפוץ ביותר. הוא דורש ש-Frida Server ירוץ כ-root אולם הוא מאפשר חופש פעולה נרחב יחסית - למשל, נוכל להזריק קוד לאפליקציות שונות בקלות יחסית, לשנות קוד ולטעון אותו במהירות לאפליקציה וכו'... במאמר זה נשתמש במצב זה.

- **Frida Gadget** - במצב זה עלינו להכניס להכניס ספריה של Frida (גאדג'ט) לתוך קובץ אפליקציה (apk), לארוז אותם ביחד ורק לאחר מכן להתקין את האפליקציה על המכשיר. מצד אחד, מדובר בפתרון טוב למצב שבו אין לנו הרשאות root על המכשיר, אולם במצב זה חופש הפעולה נפגע ויעילות השימוש פוחתת- הואיל ואיננו יכולים להזריק בקלות לאפליקציות שונות- אנו מוגבלים אך ורק לאפליקציה אליה הכנסנו את הספרייה.

התקנת Frida

ההתקנה פשוטה יחסית. הואיל ו-Frida מבוססת על Python - נוכל להתקין אותה באמצעות pypi:

pip install frida-tools

כעת נרצה להתקין Frida Server במכשיר עליו נבחן את האפליקציות שלנו. ניתן להוריד את הבינארי מכאן.
לאחר ההורדה, נחלץ את קובץ המכווץ:

```
unxz frida-server.xz
```

נדחוף אותו למכשיר:

```
adb push frida-server /data/local/tmp/
```

נריץ adb shell כדי שנוכל להריץ פקודות מתוך המכשיר ולאחר מכן נשנה לקובץ שהכנסנו את ההרשאות ונריץ אותו:

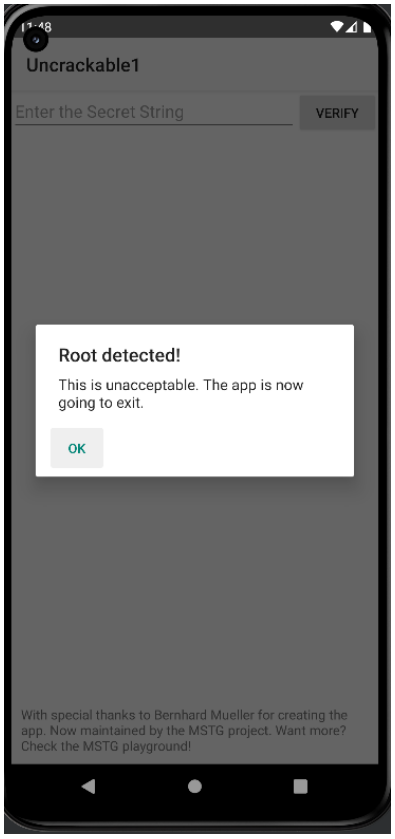
```
→  OWASP adb shell
emulator_arm64:/ $ su
emulator_arm64:/ # cd /data/local/tmp
emulator_arm64:/data/local/tmp # chmod 775 fridaserver
emulator_arm64:/data/local/tmp # ./fridaserver
```

לאחר שסיימנו את ההתקנה, הגיע הזמן ללכלך קצת הידיים. נדגים את השימוש באמצעות האפליקציה Android UnCrackable L1 מהאתר של OWASP. נתקין אותה על אותו מכשיר עליו התקנו את Frida Server:

```
→  OWASP adb install UnCrackable-Level1.apk
Performing Streamed Install
Success
```

מעקף בדיקת Root

בפתיחת האפליקציה נקבל הודעה כי זוהה שהמכשיר הוא Rooted:



לאחר לחיצה על OK והאפליקציה נסגרת. לעיתים אפליקציות לא מוכנות לרוץ כאשר המכשיר הוא rooted (או "פרוץ"). כך לדוגמה, באתר של בנק ישראלי נכתב כי ייתכן והם חוסמים את השימוש על מכשירים שהם זיהו כ-rooted:

- אנו ממליצים שלא לבצע התקנה/שימוש באפליקציה ורכיביה במכשיר פרוץ (jailbroken / rooted) עקב הסיכונים לפרטיותך ואנו שומרים את הזכות למנוע שימוש באפשרויות עקב זיהוי מכשיר פרוץ.

באפליקציה לפנינו נראה שאכן מימשו הגנה כזו.

נתחיל בלחקור את האפליקציה כדי להבין איך נראים מנגנוני ההגנה שלה וכיצד ניתן לעקוף אותם. אני משתמש ב-jadx שהוא כלי Open Source ל-reverse engineering של אפליקציות אנדרואיד.

```
→  OWASP jadx-gui UnCrackable-Level1.apk
INFO  - output directory: UnCrackable-Level1
INFO  - loading ...
INFO  - Loaded classes: 7, methods: 15,
```

הצעד הראשון הוא לבחון את המניפסט בו מפורטים רכיבי האפליקציה:

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android" android:versionCode="1" android:versionName="1.0" package="ow
    <uses-sdk android:minSdkVersion="19" android:targetSdkVersion="28"/>
    <application android:theme="@style/AppTheme" android:label="@string/app_name" android:icon="@mipmap/ic_launcher" android:allowB
        <activity android:label="@string/app_name" android:name="sg.vantagepoint.uncrackable1.MainActivity">
            <intent-filter>
                <action android:name="android.intent.action.MAIN"/>
                <category android:name="android.intent.category.LAUNCHER"/>
            </intent-filter>
        </activity>
    </application>
</manifest>
```

המנפיסט של אפליקציה קצר ופשוט. אין לאפליקציה הרשאות או רכיבים מיוחדים. נמצא את השם של ה-Activity הראשי, שמופעל בפתיחת האפליקציה. במקרה זה הוא גם ה-Activity היחיד:

```
<activity android:label="@string/app_name" android:name="sg.vantagepoint.uncrackable1.MainActivity">
    <intent-filter>
        <action android:name="android.intent.action.MAIN"/>
        <category android:name="android.intent.category.LAUNCHER"/>
```

אם כך מצאנו ש-**sg.vantagepoint.uncrackable1.MainActivity** הוא ה-Activity שמופעל מיד בפתיחה. נפתח את הקוד של **MainActivity** ונמצא את הפונ' **onCreate** שהיא הפונ' הראשונה שנקראת כש-Activity נטען.

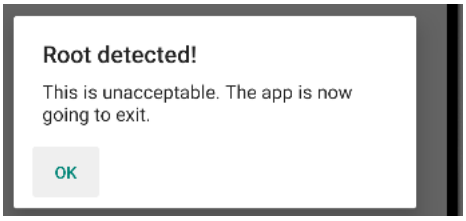
```
@Override // android.app.Activity
protected void onCreate(Bundle bundle) {
    if (c.a() || c.b() || c.c()) {
        a("Root detected!");
    }
    if (b.a(getApplicationContext())) {
        a("App is debuggable!");
    }
    super.onCreate(bundle);
    setContentView(R.layout.activity_main);
}
```

נראה שמיד כשהאפליקציה עולה היא מבצעת כמה בדיקות - איתור Root ובדיקה אם האפליקציה היא debuggable. במידה ובדיקות הללו מגלות שאכן המכשיר הוא rooted או שהאפליקציה היא debuggable - הפונ' a מופעלת.

נבחן את הפונקציה a:

```
private void a(String str) {
    AlertDialog create = new AlertDialog.Builder(this).create();
    create.setTitle(str);
    create.setMessage("This is unacceptable. The app is now going to exit.");
    create.setButton(-3, "OK", new DialogInterface.OnClickListener() { // fro
        @Override // android.content.DialogInterface.OnClickListener
        public void onClick(DialogInterface dialogInterface, int i) {
            System.exit(0);
        }
    });
    create.setCancelable(false);
    create.show();
```

היא מקבלת מחרוזת, בונה Dialog שהכותרת שלו מכילה את המחרוזת שהיא קיבלה כפרמטר. ה-Dialog גם מכיל כפתור שבלחיצה עליו האפליקציה נסגרת.



אני מריץ את האפליקציה ב-emulator שמוגדר כ-rooted device ועל כן האפליקציה לא נותנת לי להתקדם ונסגרת לאחר הצגת ה-Dialog. הדבר הראשון שנרצה לעשות יהיה לעקוף את הבדיקה הזאת.

נרצה לגרום לאפליקציה לחשוב שאנחנו לא רצים כ-rooted device. נחזור לקוד שמבצע root detection שנמצא בפונ' onCreate:

```
@Override // android.app.Activity
protected void onCreate(Bundle bundle) {
    if (c.a() || c.b() || c.c()) {
        a("Root detected!");
    }
```

למעשה מדובר ב-3 בדיקות שונות. נלך למחלקה c שמכילה את פונקציות הבדיקה:

c.a(),c.b() ,c.c()

נבחן כל אחת מהן:

הפונ' **c.a** סורקת את משתנה הסביבה PATH. בכל נתיב בו, היא בודקת אם קיים הקובץ **su**.

```
public class c {
    public static boolean a() {
        for (String str : System.getenv("PATH").split(":")) {
            if (new File(str, "su").exists()) {
                return true;
            }
        }
        return false;
    }
```

הקובץ su הוא קובץ הרצה שמאפשר לתהליך להחליף את המשתמש שהוא רץ בו. כש-su מורץ ללא פרמטרים - התהליך שמפעיל אותו יקבל הרשאות root. במילים אחרות: su מאפשר לתהליך להפוך ל-root.

הפונקציה c.b קוראת את השדה Build.TAGS:

```
public static boolean b() {
    String str = Build.TAGS;
    return str != null && str.contains("test-keys");
}
```

שנקרא מהקובץ ב-/system/build.pro :

```
emulator_arm64:/ # cat /system/build.prop | grep ro.build.tags
ro.build.tags=dev-keys
emulator_arm64:/ #
```

כאשר אנדרואיד נבנה על ידי גוגל, ה-tag יהיה release-keys. במצבים אחרים - זה יכול להיות אינדיקציה לגירסה לא רשמית.

ב-tags אחרים יכולים להיות משהו כמו test-keys או dev-keys:

```
+# The "test-keys" tag marks builds signed with the old test keys,
+# which are available in the SDK.  "dev-keys" marks builds signed with
+# non-default dev keys (usually private keys from a vendor directory).
```

[https://android.googlesource.com/platform/build/+/e17f3f2%5E%21]

הפונקציה השלישית, c.c עוברת על רשימה המכילה מספר קבצים ובודקת אם הם קיימים במכשיר:

```
public static boolean c() {
    for (String str : new String[]{"/system/app/Superuser.apk",
        if (new File(str).exists()) {
            return true;
        }
    }
    return false;
}
```

הקבצים שהיא סורקת הם:

```
{"/system/app/Superuser.apk",
"/system/xbin/daemonsu",
"/system/etc/init.d/99SuperSUDaemon",
"/system/bin/.ext/.su",
"/system/etc/.has_su_daemon",
"/system/etc/.installed_su_daemon",
"/dev/com.koushikdutta.superuser.daemon/"}
```

במידה ואחד מהקבצים הללו נמצא, הפונקציה תחזיר true. נרצה כמובן לעקוף את הבדיקות הנ"ל ולכך ניעזר ב-Frida. בפרט, נרצה להחליף את המימוש של הפונקציות c.a, c.b, c.c ולהחזיר תמיד false. בכך נגרום לבדיקה הבאה להיכשל:

```
if (c.a() || c.b() || c.c()) {
    a("Root detected!");
}
```

ניצור קובץ JS חדש. כדי לעבוד עם התשתית של Frida נצטרך להכניס את הקוד שלנו בין הסוגריים המסולסלים בקוד הבא:

```
Java.perform(function() {
   console.log("[ * ] Starting...");
 });
```

כזכור, הפונקציות c.a, c.b, c.c נמצאות במחלקה c.

מחלקה זו נמצאת ב-package בשם sg.vantagepoint.a:

```
1  package sg.vantagepoint.a;
2
3  import android.os.Build;
4  import java.io.File;
5
6  /* loaded from: classes.dex */
7  public class c {
8      public static boolean a() {
9          for (String str : System
```

נרצה לשנות את הפונקציות של המחלקה. אז כצעד ראשון נרצה לקבל גישה ל-class.

נעשה זאת באמצעות שימוש בפונקציה Java.use באופן הבא:

```
var rootCheck = Java.use("sg.vantagepoint.a.c");
console.log("[ * ] Got class rootCheck ");
```

בשלב הבא, נרצה להחליף את המימוש של הפונקציה a:

```
rootCheck.a.implementation = function(){
    console.log("[ * ] a returned false ");
    return false;
 }
```

ובאופן דומה נרצה להחליף את b ו-c. נריץ את הסקריפט שהרצנו באמצעות Frida:

```
➜  OWASP frida -U -f owasp.mstg.uncrackable1 -l Script.js
```

- הדגל **U-** הוא לחיבור לשרת דרך USB (בניגוד לחיבור מרוחק לדוגמה).
- הדגל **f-** (--file) מקבל כפרמטר את היעד אליו נכניס את הסקריפט.
- הדגל **l-** (--load) מקבל את הסקריפט שנרצה לטעון לתוך היעד.

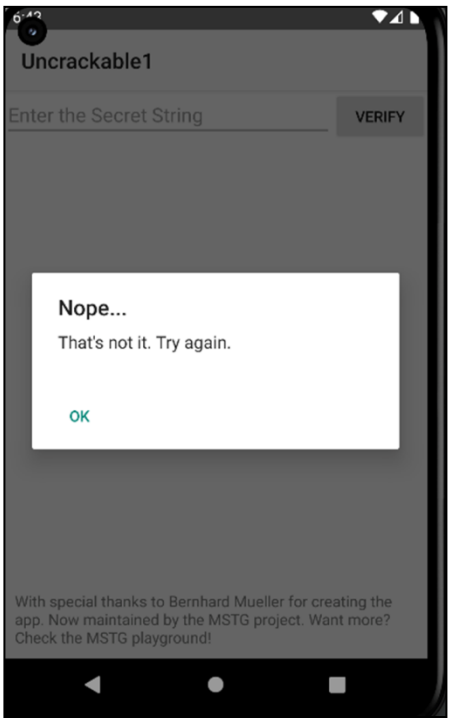
בתחתית המסך ניתן לראות ששלושת הפונקציות נקראו והוחזר false משלושתן:

```
➜  OWASP frida -U -f owasp.mstg.uncrackable1 -l Script.js
     ____
    / _  |   Frida 16.0.2 - A world-class dynamic instrumentation to
   | (_| |
    > _  |   Commands:
   /_/ |_|       help      -> Displays the help system
   . . . .       object?   -> Display information about 'object'
   . . . .       exit/quit -> Exit
   . . . .
   . . . .   More info at https://frida.re/docs/home/
   . . . .
   . . . .   Connected to Android Emulator 5554 (id=emulator-5554)
Spawned `owasp.mstg.uncrackable1`. Resuming main thread!
[Android Emulator 5554::owasp.mstg.uncrackable1 ]-> [ * ] Starting i
[ * ] Got class rootCheck
[ * ] a returned false
[ * ] b returned false
[ * ] c returned false
```

ואכן הצלחנו לעקוף את ההגנה מפני root והאפליקציה נפתחה:

עקיפת מסך הסיסמה

כעת, האפליקציה מבקשת שנזין string. בלחיצה על הכפתור נקבל שה-string secret שהזנו שגוי:



אם כך, נרצה לגלות את הקוד הנכון, או לחלופין, לשכנע את האפליקציה שמה שנזין יהיה נכון, לא משנה מה נזין. נמשיך לרוורס את האפליקציה כדי למצוא את מנגנון בדיקת הסיסמה שלה.

ה-onCreate נראה די פשוט, אבל יש פונקציה מעניינת נוספת במחלקה MainActivity.

```
41      public void verify(View view) {
42          String str;
43          String obj = ((EditText) findViewById(R.id.edit_text)).getText().toString();
44          AlertDialog create = new AlertDialog.Builder(this).create();
45          if (a.a(obj)) {
46              create.setTitle("Success!");
47              str = "This is the correct secret.";
48          } else {
49              create.setTitle("Nope...");
50              str = "That's not it. Try again.";
51          }
52          create.setMessage(str);
53          create.setButton(-3, "OK", new DialogInterface.OnClickListener() { // from class
54              @Override // android.content.DialogInterface.OnClickListener
55              public void onClick(DialogInterface dialogInterface, int i) {
56                  dialogInterface.dismiss();
57              }
58          });
59          create.show();
60      }
61  }
```

בשורה 43 נקרא טקסט שמאוכסן ב-view כלשהו. טקסט זה הוא הסיסמה שהזנו.

בשורה 45 מבוצעת קריאה לפונ' a.a, התוצאה של הפונקציה הזאת תקבע האם הסיסמה שלנו התקבלה או לא.

נבחן את הפונקציה a.a:

```
public class a {
    public static boolean a(String str) {
        byte[] bArr;
        byte[] bArr2 = new byte[0];
        try {
            bArr = sg.vantagepoint.a.a.a(b("8d127684cbc37c17616d806cf50473cc"), Base64.decode("5UJiFctbmgbI
        } catch (Exception e) {
            Log.d("CodeCheck", "AES error:" + e.getMessage());
            bArr = bArr2;
        }
        return str.equals(new String(bArr));
    }
```

היא מפעילה את הפונקציה b על מחרוזת של מספר הקסהדצימלי, ושולחת את התוצאה יחד עם פיענוח של מחרוזת ב-base64 לפונקציה a.a.a.

אנחנו יכולים לחפור מעט יותר עמוק ולגלות ש-a.a.a מפענחת מידע באמצעות AES:

```
public static byte[] a(byte[] bArr, byte[] bArr2) {
    SecretKeySpec secretKeySpec = new SecretKeySpec(bArr, "AES/ECB/PKCS7Padding");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(2, secretKeySpec);
    return cipher.doFinal(bArr2);
}
```

אם כך, אנחנו יכולים לתקוף את האפליקציה ב-2 דרכים:

1. להחליף את ערך ההחזרה של הפונקציה a.a שתמיד יחזיר true - שינוי זה יגרום לכך שכל סיסמה שנזין תתקבל.
2. להדפיס את ערך ההחזרה של a.a.a ובכך לגלות את הסיסמה.

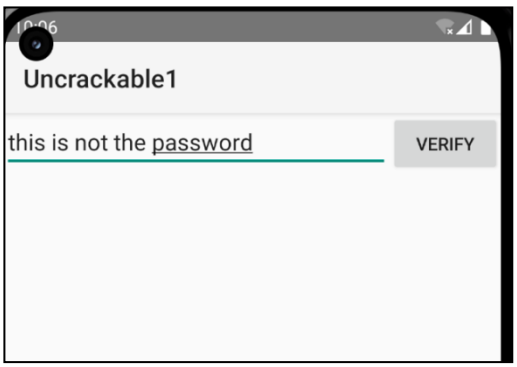
נממש את שתי הדרכים.

נתחיל בדרך הראשונה ונוסיף את הקטע הבא לסקריפט שלנו:

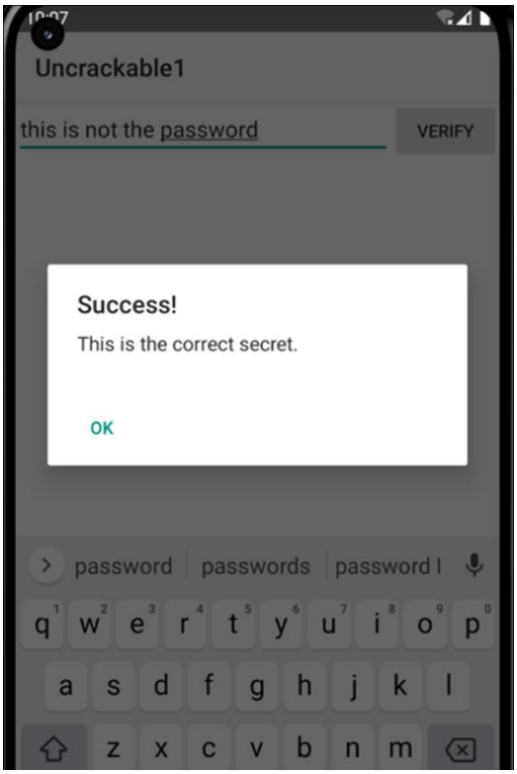
```
var secretHack = Java.use("sg.vantagepoint.uncrackable1.a");
 secretHack.a.implementation = function() {
    console.log("[ * ] Returning true from secretHack.a ");
    return true;
 }
```

1. בשורה הראשונה ניצור אובייקט שייצג את המחלקה a.
2. בשורה השנייה נחליף את המימוש של הפונקציה המקורית בפונקציה החדשה.
3. הפונקציה החדשה תדפיס לוג ותחזיר true.
4. נריץ Frida באותו אופן.

נזין טקסט כלשהו:



ובלחיצה על verify נקבל:



כלומר הצלחנו לשכנע את האפליקציה לקבל את הסיסמאות שלנו! הפלט של הסקריפט שלנו:

```
[ * ] Got class rootCheck
[ * ] a returned false
[ * ] b returned false
[ * ] c returned false
[ * ] Returning true from secretHack.a
```

חשיפת הסיסמה

כעת, נרצה לגלות את הסיסמה האמיתית. כזכור, הסיסמה האמיתית מתפענחת בפונקציה a.a.a. לשם כך, נמחק את המקטע האחרון שהוספנו ונכניס במקומו את הקטע הבא:

```
var decryptClass = Java.use("sg.vantagepoint.a.a");
 decryptClass.a.implementation = function(arg1, arg2) {
    let decryptedPass = this.a(arg1, arg2);
    console.log("[ * ] decryptedPass = " + decryptedPass);
    return decryptedPass;
 }
```

נשים לב למספר שינויים.

ראשית, שם ה-package השתנה.

שנית, הפעם הפונקציה המקורית קיבלה פרמטרים ועל כן, גם הפונקציה החדשה שאנו כותבים, שמחליפה את הפונקציה המקורית, צריכה לקבל את אותם פרמטרים. בפונקציה החדשה שכתבנו אנחנו קוראים לפונקציה המקורית (שורה 3) כדי לקבל את הסיסמה המפוענחת.

בשורה 4 נדפיס את ערך ההחזרה ובשורה 5 נחזיר אותו.

לאחר הרצת הסקריפט נקבל:

```
[ * ] decryptedPass = 73,32,119,97,110,116,32,116,111,32,98,101,108,105,101,118,101
```

מה זה? למה לא קיבלנו מחרוזת?

אם ניזכר בפונקציה המקורית:

```
public static byte[] a(byte[] bArr, byte[] bArr2) {
    SecretKeySpec secretKeySpec = new SecretKeySpec(bArr, "AES/ECB/PKCS7Padding");
    Cipher cipher = Cipher.getInstance("AES");
    cipher.init(2, secretKeySpec);
    return cipher.doFinal(bArr2);
}
```

נשים לב שהיא מחזיר אובייקט מסוג byte array ולא מסוג string וזה מסביר למה לא קיבלנו מחרוזת.

נרצה להמיר את מערך התווים למחרוזת.

ולפיכך נשנה מעט את הקוד שכתבנו:

```
var decryptClass = Java.use("sg.vantagepoint.a.a");
 var stringClass = Java.use("java.lang.String");
 decryptClass.a.implementation = function(arg1, arg2) {
    let byteArrPass = this.a(arg1, arg2);
    let stringPass = stringClass.$new(byteArrPass);
    console.log("[ * ] decryptedPass = " + stringPass);
    return byteArrPass;
 }
});
```

בשורה הראשונה אנחנו יוצרים אובייקט שמייצג את המחלקה a כמקודם.

בשורה השנייה אנחנו יוצרים אובייקט שמייצג String של Java.

בשורה השלישית אנחנו מחליפים את הפונקציה המקורית בפונקציה שלנו.

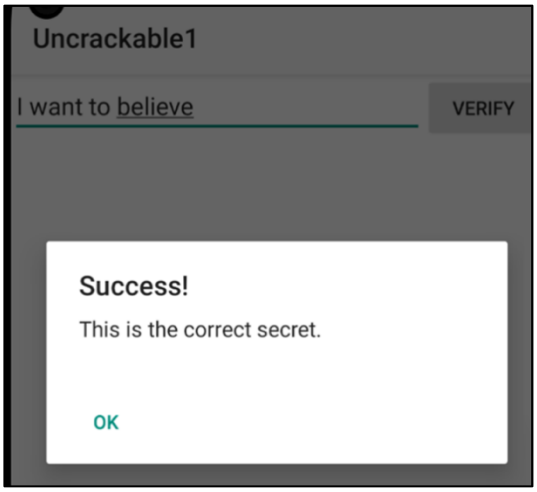
בשורה הרביעית אנחנו מפעילים את הפונקציה המקורית ומאכסנים את ערך ההחזרה במשתנה byteArrPass.

בשורה החמישית אנחנו יוצרים אובייקט String של Java שמקבל את המשתנה byteArrPass - כלומר אנחנו ממירים את מערך הבתים למחרוזת, ובשורה השישית אנחנו מדפיסים את הערך.

בשורה השביעית אנחנו מחזירים את מערך הבתים כמקודם.

נריץ עם Frida ונקבל את הסיסמה:

```
[ * ] Got class rootCheck
[ * ] a returned false
[ * ] b returned false
[ * ] c returned false
[ * ] decryptedPass = I want to believe
```



הסקריפט המלא שכתבנו:

```
Java.perform(function() {
    console.log("[ * ] Starting implementation...");
    var rootCheck = Java.use("sg.vantagepoint.a.c");
    console.log("[ * ] Got class rootCheck ");
    rootCheck.a.implementation = function(){
        console.log("[ * ] a returned false ");
        return false;
    }

    rootCheck.b.implementation = function(){
        console.log("[ * ] b returned false ");
        return false;
    }

    rootCheck.c.implementation = function(){
```

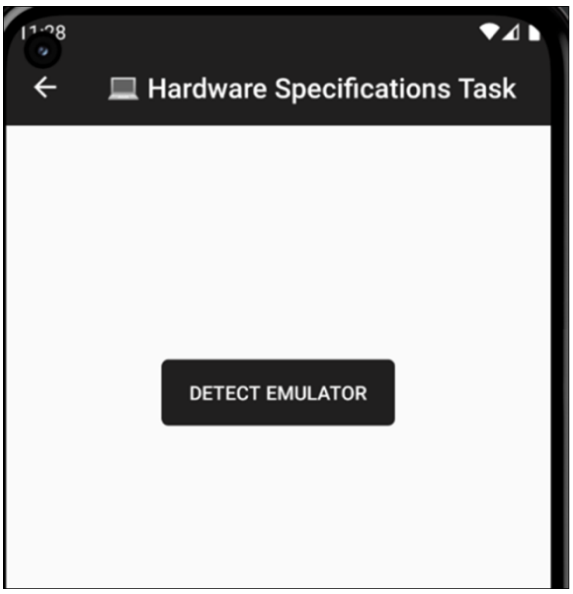
```
        console.log("[ * ] c returned false ");
        return false;
    }

    var decryptClass = Java.use("sg.vantagepoint.a.a");
    var stringClass = Java.use("java.lang.String");
    decryptClass.a.implementation = function(arg1, arg2) {
        let byteArrPass = this.a(arg1, arg2);
        let stringPass = stringClass.$new(byteArrPass);
        console.log("[ * ] decryptedPass = " + stringPass);
        return byteArrPass;
    }
});
```

מעקף Emulator Detection

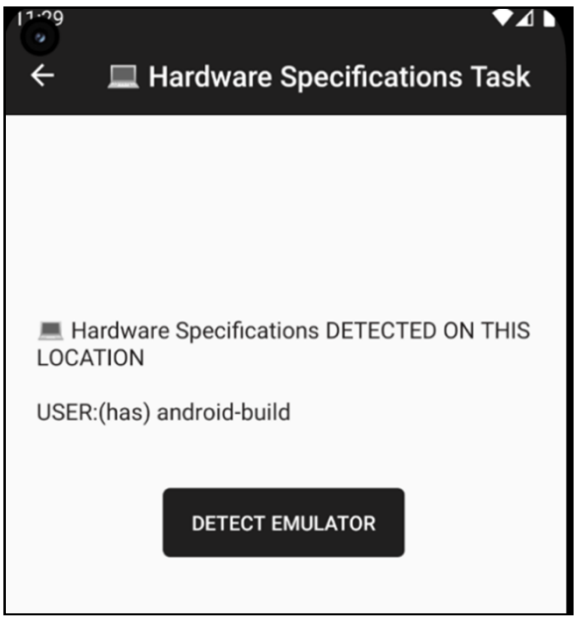
בדומה להגנה שראינו מפני ריצה על מכשירים פרוצים, לעיתים אפליקציות ירצו להימנע מלרוץ או יתנהגו באופן שונה כאשר הן רצות בסביבה וירטואלית (Emulator).

ישנן מגוון רחב של בדיקות שניתן לבצע כדי לזהות אם מכשיר הוא emulator או לא. נדגים מעקף של Emulator Detection באמצעות האפליקציה com.hpandro.androidsecurity_1.3.apk האפליקציה היא אפליקציית Kotlin שפותחה כחלק מ-CTF. היא מכילה אתגרים קטנים שבסיומו של כל אתגר מקבלים "דגל" המעיד על סיום האתגר.

אנחנו נדגים עקיפת Emulator Detection בכך שנפתור את אתגר ה-Hardware Specifications Task:



בלחיצה על הכפתור ניתן לראות שהאפליקציה זיהתה שהמכשיר אכן רץ ב-Emulator:



נתחיל שוב במחקר הקוד של ה-Activity הרלוונטי. נשים לב שהואיל והאפליקציה פותחה ב-Kotlin, הקוד שיציג Jadx עשוי להיות מעט יותר מסורבל.

ה-onCreate קורא לפונקציה init:

```
public void onCreate(Bundle bundle) {
    super.onCreate(bundle);
    setContentView(R.layout.activity_emulator_detection);
    init();
}
```

החלק שמעניין אותנו ב-init הוא החלק ששם Listener על כפתור בשם btnCheckEmulator.

```
public final void init() {
    String stringPlus = Intrinsics.stringPlus(getString(R.string.hardware_spec), " Task");
    Toolbar toolbarTask = (Toolbar) findViewById(R.id.toolbarTask);
    Intrinsics.checkNotNullExpressionValue(toolbarTask, "toolbarTask");
    init_Toolbar(toolbarTask, stringPlus);
    ((Button) findViewById(R.id.btnCheckEmulator)).setOnClickListener(new View.OnClickListener() { // from class: com.hpar
        @Override // android.view.View.OnClickListener
        public final void onClick(View view) {
            HardwareSpecificationsActivity.lambda$1pzo6QP76Oq7ubMsNkM0G12LOwU(HardwareSpecificationsActivity.this, view);
        }
    });
}
```

בכל פעם שהכפתור יילחץ, תופעל הפונקציה עם השם הלא סימפתי:

lambda$1pzo6QP76Oq7ubMsNkM0G12LOwU

שקוראת לפונקציה נוספת:

```
public static /* synthetic */ void lambda$1pzo6QP76Oq7ubMsNkM0G12LOwU(Har
    m104init$lambda0(hardwareSpecificationsActivity, view);
}
```

בתוכה נגלה את הקוד המעניין שאחראי להחלטה האם אנו רצים באמולטור או לא:

```
100     public static final void m104init$lambda0(HardwareSpecificationsActivity this$0, View view) {
101         Intrinsics.checkNotNullParameter(this$0, "this$0");
102         String string = MainApp.Companion.getSharedPrefEmulatorDetection().getString("HardwareSpecificationStr", "00");
103         boolean checkHardwareSpecifications = this$0.checkHardwareSpecifications();
104         Intrinsics.checkNotNull(string);
105         if (StringsKt.contains$default((CharSequence) string, (CharSequence) "F", false, 2, (Object) null) && !checkHardwareSpecifications) {
106             this$0.presenter = new EmulatorCheckTaskPresenterAPI(this$0);
```

ננתח את השורות הרלוונטיות:

שורה 102 - אנחנו קוראים את הערך שהמפתח שלו בור HardwareSpecificationStr מה-Shared Preferences. ה-Shared Preferences באנדרואיד הם קבצים בפורמט XML בו מאוכסנים ערכים שהאפליקציה עשויה להשתמש בהם. ערכים אלו יכולים להשתנות במהלך ריצת האפליקציה- למשל עקב הגדרות המשתמש.

דוגמה לקטע מתוך קובץ Shared Preferences של google calendar:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <boolean name="uss_sa_shipshape" value="true" />
    <boolean name="contacts_permissions_never_ask_detected" value="false" />
    <long name="contacts_permissions_request_count" value="0" />
    <boolean name="uss_mod_shipshape" value="true" />
```

בשורה 103 - אנחנו מפעילים את הפונקציה checkHardwareSpecifications שנראית מאוד מעניינת.

בשורה 105 - יש לנו תנאי שבודק שני דברים:

1. האם המחרוזת שהמפתח שלה הוא HardwareSpecificationStr שקיבלנו מה-Shared Preferences בשורה 102 מכיל את התו F.
2. האם הערך שחזר מהפונקציה checkHardwareSpecifications הוא false.

רק במידה ושני התנאים הללו מתקיימים, נמשיך לשורה 106 שבה מתבצע קוד מעט מורכב שאחראי להציג לנו את הדגל:

```
this$0.presenter = new EmulatorCheckTaskPresenterAPI(this$0);
ProgressBar progress = (ProgressBar) this$0.findViewById(R.id.progress);
Intrinsics.checkNotNullExpressionValue(progress, "progress");
this$0.showProgressBar(progress);
EmulatorCheckTaskPresenterAPI emulatorCheckTaskPresenterAPI = this$0.pre
if (emulatorCheckTaskPresenterAPI != null) {
    this$0.getHardwareSpecificationFlag(emulatorCheckTaskPresenterAPI);
```

לעומת זאת, אם התנאי לא מתקיים, תוצג לנו השורה כפי שקיבלנו:

```
showData(this$0.getString(R.string.hardware_spec) + " DETECTED ON THIS LOCATION
```

נבחן את הפונקציה checkHardwareSpecifications:

```
public final boolean checkHardwareSpecifications() {
    boolean z;
    String FINGERPRINT = Build.FINGERPRINT;
    Intrinsics.checkNotNullExpressionValue(FINGERPRINT, "FINGERPRINT");
    boolean z2 = false;
    if (!StringsKt.startsWith$default(FINGERPRINT, "generic", false, 2, (Object) null)) {
        String MODEL = Build.MODEL;
        Intrinsics.checkNotNullExpressionValue(MODEL, "MODEL");
        if (!StringsKt.contains$default((CharSequence) MODEL, (CharSequence) "google_sdk", false, 2, (Object) null)) {
            String MODEL2 = Build.MODEL;
            Intrinsics.checkNotNullExpressionValue(MODEL2, "MODEL");
            String lowerCase = MODEL2.toLowerCase();
            Intrinsics.checkNotNullExpressionValue(lowerCase, "(this as java.lang.String).toLowerCase()");
            if (!StringsKt.contains$default((CharSequence) lowerCase, (CharSequence) "droid4x", false, 2, (Object) null)) {
                String MODEL3 = Build.MODEL;
                Intrinsics.checkNotNullExpressionValue(MODEL3, "MODEL");
                if (!StringsKt.contains$default((CharSequence) MODEL3, (CharSequence) "Emulator", false, 2, (Object) null)) {
                    String MODEL4 = Build.MODEL;
                    Intrinsics.checkNotNullExpressionValue(MODEL4, "MODEL");
                    if (!StringsKt.contains$default((CharSequence) MODEL4, (CharSequence) "Android SDK built for x86", false,
                        String MANUFACTURER = Build.MANUFACTURER;
                        Intrinsics.checkNotNullExpressionValue(MANUFACTURER, "MANUFACTURER");
                        if (!StringsKt.contains$default((CharSequence) MANUFACTURER, (CharSequence) "Genymotion", false, 2, (
```

אנחנו רואים שנבדקים מאפיינים רבים של ה-build כמו מודל, יצרן, מספר סריאלי וכדומה. מאפיינים אלו מושווים למאפיינים של אמולטורים. במידה ונמצאה התאמה, הפונקציה תחזיר true.

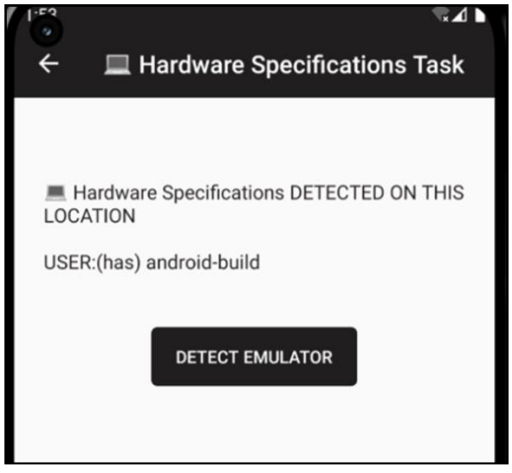
אם כך, בשלב ראשון נרצה לעקוף את הבדיקה הזאת. הדרך הפשוטה ביותר היא להחליף את הפונקציה checkHardwareSpecifications בפונקציה משלנו שתחזיר תמיד false:

```
Java.perform(function() {
  console.log("[ * ] Starting bypass emulator detection script");
  var HardwareSpecificationsActivity = Java.use("com.hpandro.androidsecurity.ui.activity.task.
                                                emulatorDetection.
                                                HardwareSpecificationsActivity");
  HardwareSpecificationsActivity.checkHardwareSpecifications.implementation = function () {
    console.log("[ * ] original checkHardwareSpecifications should be " +
                this.checkHardwareSpecifications());
    console.log("[ * ] Returning false ");
    return false;
 }
```

האם זה מספיק?

```
[ * ] Starting bypass emulator detection script
[ * ] original checkHardwareSpecifications should be true
[ * ] Returning false
```

מסתבר שלא!



כזכור ראינו שהתנאי שמחליט האם להציג לנו את הדגל מורכב משני חלקים:

```
if (StringKt.contains$default((CharSequence) strings, (CharSequence) "F",
false, 2 (Object) null) && !checkHardwareSpecifications) {
```

החלק שבו לא טיפלנו הוא הבדיקה האם המחרוזת שקיבלנו מה-Shared Preferences מכילה את התו F.

אבל מה התנאי הזה בעצם בודק? ומה המחרוזת הזו מכילה?

בנתיב /data/data/com.hpandro.androidsecurity/shared_prefs מאוכסנים ה-Shared Preferences.

```
emulator_arm64:/data/data/com.hpandro.androidsecurity/shared_prefs # ls
AndroidSecurity.xml    FirebaseAppHeartBeat.xml  WebViewChromiumPrefs.xml  com.google.android.gms.appid.xml
EmulatorDetection.xml  RootDetection.xml         admob.xml                 com.google.android.gms.measurement.prefs.xml
```

אנו מעוניינים בקובץ EmulatorDetection.xml:

```
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
  <string name="HardwareSpecificationStr">000</string>
  <boolean name="DebugFlag" value="false" />
  <string name="PackageNamesBaseStr">000</string>
  <string name="QEMUStr">00F</string>
  <string name="DeviceIDsStr">000</string>
  <string name="VirtualPhoneNumberStr">000</string>
  ...
  ...
</map>
```

אנו רואים כל מיני keys כמו HardwareSpecificationStr וPackageNamesBaseStr וערכים שרובם 0 אבל לפעמים יש גם F. אבל מה 0 אומר? ומה F אומר? ומי כותב לשם בכלל?

האמת היא שבכלל לא צריך לדעת את התשובה. הואיל ואנחנו רק מעוניינים לקבל את הדגל, כל מה שמעניין אותנו הוא שהביטוי הבא:

```
(StringKt.contains$default((CharSequence) strings, (CharSequence) "F", false, 2
(Object) null)
```

יהיה true.

ובכל זאת נתאר מה המשמעות. בשלב יחסית מוקדם בעליית האפליקציה, מבוצעות בדיקות רבות שהתוצאות שלהן נכתבות ב-shared preferences בצורה של 0 (לא אותר) או F (אותר).

טוב, אז כאמור, נרצה לוודא שהביטוי לעיל תמיד מחזיר true. אם כך, נרצה לדרוס את contains$default של המחלקה StringsKt. נוסיף את הקטע הבא:

```
var StringsKt = Java.use("kotlin.text.StringsKt");
 StringsKt.contains$default.implementation = function(arg1, arg2, arg3, arg4, arg5) {
   return true
 }
```

וסיימנו לא? אז לא כל כך מהר...

```
Error: contains$default(): has more than one overload, use .overload(<signature>) to choose from:
        .overload('java.lang.CharSequence', 'char', 'boolean', 'int', 'java.lang.Object')
        .overload('java.lang.CharSequence', 'java.lang.CharSequence', 'boolean', 'int', 'java.lang.Object')
```

Frida כועסת עלינו! ובצדק...

למעשה, בניגוד להחלפות הקודמות שביצענו (בהן החלפנו פונקציות מקומיות יחסית) כעת אנחנו מחלפים פונקציה תשתיתית חשובה! והמשמעות מבחינתנו היא שיש שיקולים נוספים שעלינו להתחשב בהם.

השיקול הראשון הוא שיש overloading לפונקציה - כלומר קיימות מספר פונקציות עם אותו שם.

אם ננווט למחלקה StringsKt נגלה כי אכן קיימות שתי גרסאות לפונקציה, שתיהן עם אותו מספר פרמטרים:

- `public static /* synthetic */ boolean contains$default(CharSequence charSequence, CharSequence charSequence2, boolean z, int i, Object obj)`
- `public static /* synthetic */ boolean contains$default(CharSequence charSequence, char c, boolean z, int i, Object obj)`

כאשר ביקשנו מ-Frida להחליף את הפונקציה, היא לא ידעה במי לבחור ולכן קיבלנו הודעת שגיאה. נפתור זאת באמצעות ציון מפורש של טיפוסי פרמטרים שהפונקציה מקבלת:

```
var StringsKt = Java.use("kotlin.text.StringsKt");
StringsKt.contains$default.overload('java.lang.CharSequence', 'java.lang.CharSequence',
'boolean', 'int', 'java.lang.Object').implementation = function(arg1, arg2, arg3, arg4,
arg5) {
   return true
}
```

ניתן לראות שהוספנו קריאה ל-overload שאומרת ל-Frida לבחור את הפונקציה שהטיפוסים שלה הם הטיפוסים בתוך הסוגריים.

האם כעת סיימנו? גם לא...

```
***
FATAL EXCEPTION: main
Process: com.hpandro.androidsecurity, PID: 17304
java.lang.ExceptionInInitializerError
```

החלפנו פונקציה תשתיתית מרכזית, שמבצעים לה קריאות בכל רחבי התוכנית - ובמקום להחזיר את הערך התקין שהיה אמור להתקבל - אנו תמיד מחזירים true. כלומר אנחנו גורמים לחוסר יציבות אדיר בתוכנית.

עלינו לשפר את הקוד כדי לגרום מינימום נזק! נדפיס את הפרמטרים שהפונקציה מקבלת.

הפרמטרים החשובים ביותר הם הראשון והשני, שבהם נמצא ה-string שעליו מחפשים, וה-substring או התו שאותו מחפשים:

```
var StringsKt = Java.use("kotlin.text.StringsKt");
 StringsKt.contains$default.overload('java.lang.CharSequence',
'java.lang.CharSequence', 'boolean', 'int', 'java.lang.Object').implementation
= function(arg1, arg2, arg3, arg4, arg5) {
    console.log("[ + ] contains$default: arg1=" + arg1 + " arg2=" + arg2);
   return true
 }
```

כשנריץ עם Frida נגלה אלפי קריאות לפונקציה contains$default:

```
[ + ] contains$default: arg1=[ro.zygote]: [zygote64_32] arg2=[1]
[ + ] contains$default: arg1=[ro.zygote.disable_gl_preload]: [1] arg2=ro.secure
[ + ] contains$default: arg1=[ro.zygote.disable_gl_preload]: [1] arg2=[0]
[ + ] contains$default: arg1=[ro.zygote.disable_gl_preload]: [1] arg2=ro.debuggable
[ + ] contains$default: arg1=[ro.zygote.disable_gl_preload]: [1] arg2=[1]
[ + ] contains$default: arg1=[security.perf_harden]: [1] arg2=ro.secure
[ + ] contains$default: arg1=[security.perf_harden]: [1] arg2=[0]
[ + ] contains$default: arg1=[security.perf_harden]: [1] arg2=ro.debuggable
[ + ] contains$default: arg1=[security.perf_harden]: [1] arg2=[1]
[ + ] contains$default: arg1=[selinux.restorecon_recursive]: [/data/misc_ce/0] arg2=ro.secure
[ + ] contains$default: arg1=[selinux.restorecon_recursive]: [/data/misc_ce/0] arg2=[0]
```

כל הקריאות הללו לא רלוונטיות מבחינתנו. אותנו מעניינת אך ורק הקריאה הבאה:

```
(StringKt.contains$default((CharSequence) strings, (CharSequence) "F", false, 2
(Object) null)
```

מה אנחנו יודעים עליה? אנחנו יודעים שהפרמטר הראשון הוא המחרוזת שנקראה מה-Shared Preferences והיא יכולה להכיל רק 0 ו-F. הפרמטר השני הוא F, השלישי הוא false וכן הלאה.

אם כך, נוסיף תנאי שרק במקרה שבו זיהינו קריאה ל-contains$default עם הפרמטרים במדויקים האלו- רק אז נחזיר תמיד true.

בשאר המצבים - נקרא לפונקציה המקורית. כלומר הקוד המתוקן שלנו יראה כך:

```
var StringsKt = Java.use("kotlin.text.StringsKt");
StringsKt.contains$default.overload('java.lang.CharSequence', 'java.lang.CharSequence',
'boolean', 'int', 'java.lang.Object').implementation =
function(arg1, arg2, arg3, arg4, arg5) {
  console.log("[ + ] contains$default: arg1=" + arg1 + " arg2=" + arg2);
  var arg1_str = "" + arg1;
  if ((arg1_str.match(/^[0F]*$/) !== null) &&  (arg2=='F') &&
      (arg3 == false) && (arg4 == 2) && (arg5 == null)) {
      console.log("[ + ] returned true");
      return true;
   }
   console.log("[ + ] returned original");
   return this.contains$default(arg1, arg2, arg3, arg4, arg5);
 }
```

נסביר את החלקים המעניינים:

- בשורה הראשונה אנו ניגשים למחלקה StringsKt.
- בשורות 2-4 אנחנו מחליפים את המימוש של הפונקציה contains$default, אנחנו מציינים את טיפוסי הפרמטרים שבמפורש כדי להחליף ה-overload הספציפי שאנו מעוניינים בו.
- בשורה 5 נדפיס את הפרמטרים הראשון והשני.
- בשורה 6 נמיר את הפרמטר הראשון מאובייקט string של java לאובייקט string של javascript.
- אנו עושים זאת כדי שבשורה הבאה נוכל להפעיל עליו פונקציות של javascript.

בשורות 7-8 אנחנו בודקים שהפרמטרים שקיבלנו הם בדיוק הפרמטרים שהופיעו כאן:

```
(StringKt.contains$default((CharSequence) strings, (CharSequence) "F", false, 2
(Object) null)
```

את הערך של הפרמטר הראשון אנחנו לא יודעים במדויק אבל הואיל והוא יכול להכיל רק 0 ו-F, נשווה אותו לביטוי רגולרי שמייצג מחרוזת של 0 ו-F בלבד.

בשורות 9-10 אנחנו מדפיסים ומחזירים true מכיוון שמצאנו את הקריאה המדויקת שחיפשנו. לעומת זאת, אם לא מצאנו (שורות 12-13) - נדפיס ונחזיר את הערך של קריאה לפונקציה המקורית.

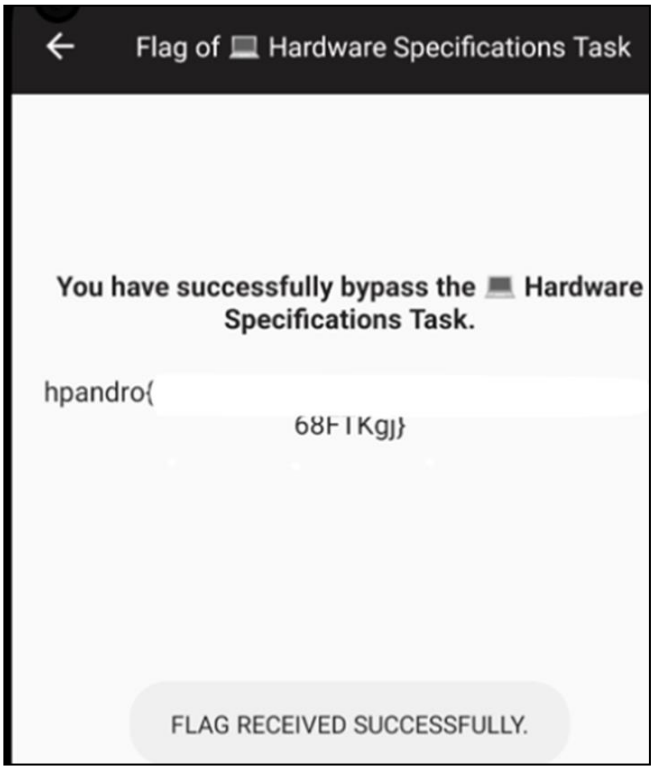
הסקריפט המלא שלנו יראה כך:

```
Java.perform(function() {
  console.log("[ * ] Starting bypass emulator detection script");
  var HardwareSpecificationsActivity =
Java.use("com.hpandro.androidsecurity.ui.activity.task.emulatorDetection.HardwareSpecif
icationsActivity");
  HardwareSpecificationsActivity.checkHardwareSpecifications.implementation = function
() {
    console.log("[ * ] original checkHardwareSpecifications should be " +
this.checkHardwareSpecifications());
    console.log("[ * ] Returning false ");
    return false;
 }

var StringsKt = Java.use("kotlin.text.StringsKt");
StringsKt.contains$default.overload('java.lang.CharSequence', 'java.lang.CharSequence',
'boolean', 'int', 'java.lang.Object').implementation =
function(arg1, arg2, arg3, arg4, arg5) {
  console.log("[ + ] contains$default: arg1=" + arg1 + " arg2=" + arg2);
  var arg1_str = "" + arg1;
  if ((arg1_str.match(/^[0F]*$/) !== null) &&  (arg2=='F') &&
      (arg3 == false) && (arg4 == 2) && (arg5 == null)) {
      console.log("[ + ] returned true");
      return true;
   }
   console.log("[ + ] returned original");
   return this.contains$default(arg1, arg2, arg3, arg4, arg5);
 }
});
```

נטען את הסקריפט ל-Frida ונסתכל בלוגים. כאשר מופעלת הפונקציה contains$default עם קריאות שלא תואמות את הפילטר שהגדרנו - חוזר הערך המקורי. לעומת זאת כאשר נקראה הפונקציה עם הפרמטרים שרצינו - חזר true:

```
[ + ] contains$default: arg1=rsr1.210722.002 arg2=frf91
[ + ] returned original
[ + ] contains$default: arg1=ranchu arg2=goldfish
[ + ] returned original
[ + ] contains$default: arg1=emulator_arm64 arg2=generic
[ + ] returned original
[ + ] contains$default: arg1=google/sdk_gphone_arm64/emulator_arm64:11/rsr1.210722.002/7602718:userdebug/d
[ + ] returned original
[ + ] contains$default: arg1=unknown arg2=null
[ + ] returned original
[ + ] contains$default: arg1=android-build arg2=android-build
[ + ] returned original
[ * ] original checkHardwareSpecifications should be true
[ * ] Returning false
[ + ] contains$default: arg1=00F arg2=F
[ + ] returned true
[ + ] contains$default: arg1=hpandro.raviramesh.info arg2=:
[ + ] returned original
```

ואכן הצלחנו לקבל את הדגל!



לסיכום

במאמר ראינו מספר שימושים חשובים ל-Frida והדגמנו אותם באמצעות ביצוע מניפולציות על האפליקציות Android UnCrackable L1 ו-com.hpandro.androidsecurity_1.3.apk.

תחילה ביצענו מחקר קוד סטטי של האפליקציה Android UnCrackable L1 והבנו אילו מנגנוני הגנה קיימים בתוכה. גילינו כי קיימים בתוכה מספר בדיקות root ובדיקה נוספת שמגלה האם האפליקציה היא debuggable. כתבנו סקריפט ב-JS שיחליף את הפונקציות שמבצעות את הבדיקות הללו, וטענו אותו ל-Frida ובכך הצלחנו לשכנע את האפליקציה שהיא רצה בסביבה לגיטימית.

בשלב השני גילינו כי קיים מסך הזדהות. חקרנו את הקוד שמנהל אותו, מצאנו את הפונקציה שמשווה בין הסיסמה שהזין המשתמש לבין סיסמה מוצפנת שנמצאת באפליקציה. שכללנו את הסקריפט שלנו והחלפנו את פונקציה ההשוואה בפונקציה משלנו שתמיד תקבל כל סיסמה שנזין לה - ובכך הצלחנו לנטרל את מסך ההזדהות.

בשלב השלישי רצינו לחשוף את הסיסמה המוצפנת. לשם כך ביצענו hook על פונקציית הפיענוח- קראנו בפונקציה שלנו לפונקציית הפיענוח המקורית והדפסנו את הערך המפוענח ובכך חשפנו את הסיסמה המוצפנת.

לאחר מכן פתרנו אתגר מ-hpandro CTF במסגרתו למדנו איך מתבצע זיהוי Emulators ומה נדרש כדי לשכנע את האפליקציה שאיננו רצים ב-Emulator ולמסור לנו את הדגל.

התמודדנו עם האתגרים טכניים הכרוכים בהחלפת פונקציה תשתיתית מרכזית והצלחנו להשיג את הדגל.

על המחבר

שמי בניה, בן 23, מהנדס תוכנה. בזמני הפנוי אני עוסק בנושאי אבטחת מידע ומחקר Malwares. מוזמנים לפנות בכל נושא Bnayayoo@gmail.com.

תודות

תודה רבה ל**אפיק קסטיאל** על העריכה!