

# DD2525 – Lab Assignment 3

## Android Security

Deadline: May 17, 2024


## 1 Introduction

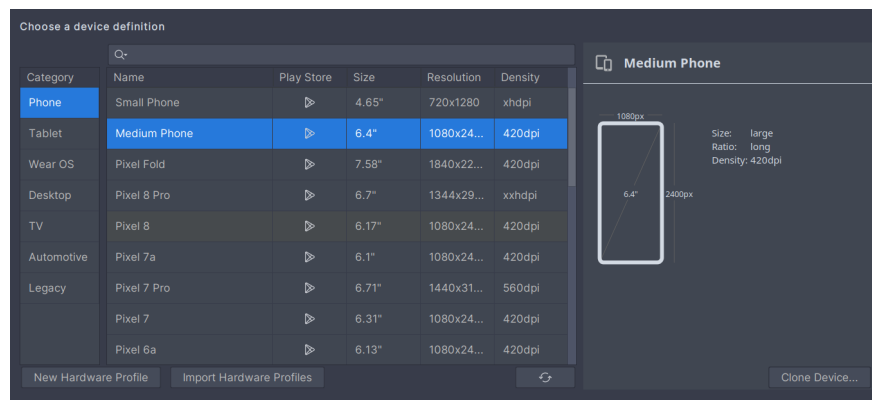
This assignment introduces you to some of the key concepts in Android application security. At its core, the Android platform relies on Linux-based access control and controlled inter-app communication via the permission system to enforce security. As this lab will show, the current security mechanisms of Android are not always sufficient to protect the privacy of users. Common malpractices in Android development may allow an attacker to access users' sensitive data. In this lab, you will investigate the root cause of some of these vulnerabilities, exploiting them by implementing attacks, and proposing fixes by suggesting and implementing countermeasures.

## 2 Getting Started

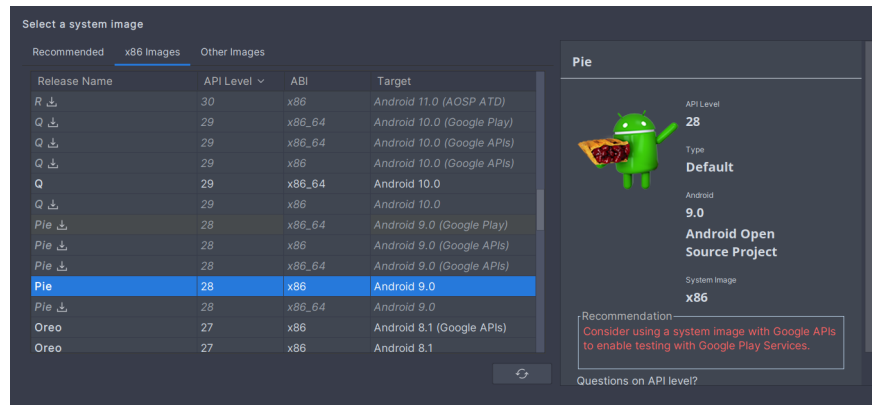
### 2.1 Install and configure Android Studio

First, make sure you have Java Development Kit (JDK) – at least – version 1.8 installed. Download Android Studio and follow the installation steps for your preferred operating system.

When running Android Studio for the first time, let it go through the standard initialization process to install the latest SDK and platform tools. Next, in order to create an emulator select  on the start screen and choose Virtual Device Manager to open the device manager window, in which you can click on the + button to create a new virtual device. For this lab you can choose the “Medium Phone” template.



After selecting the phone template, **do not** accept the recommended target, instead go to x86 Images tab and select Android API 28, ABI x86 as target:



## 2.2 Starting code for the assignment

Download and extract the zip file with the starting code from Canvas. You will find a directory for each task, which includes the vulnerable app(s) as well as a template for exploit apps which you need to extend in order to exploit the vulnerable apps. You *should* open each of these apps as an Android project.

## 3 Android App Vulnerabilities

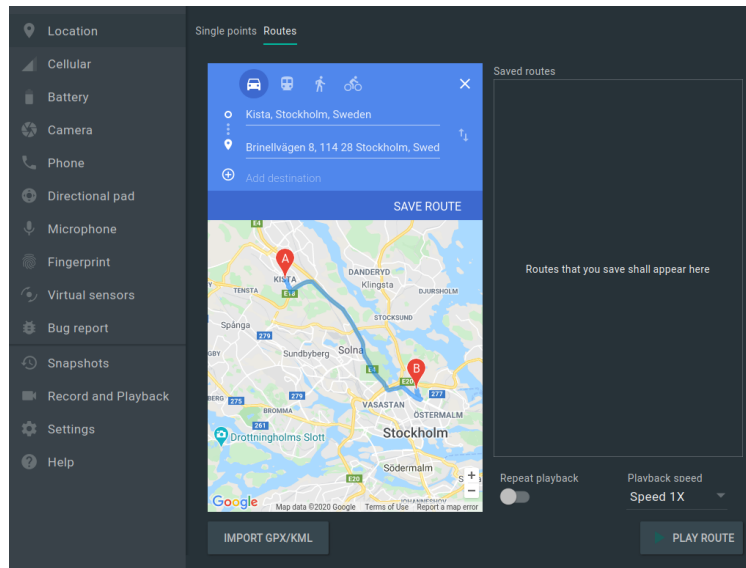
### 3.1 Intent Sniffing

Android uses Intents to facilitate inter-app communication as well as interaction between different components of a single app. However, when a component is initiated by using a broadcast intent, the data passed in the intent may be read by other apps. The scenario here is that a service in *LocationApp* reads the user's location, sends it to an activity in *LocationApp* in order to show the user's location on a map, and it also sends this data to an external app *WeatherApplication* to show the weather. In this task, you should implement an intent sniffer that captures the location data passed between the internal components of *LocationApp*, as well as the data passed between *LocationApp* and *WeatherApplication*.

#### Preparation:

1. Take a look at Broadcasts in Android and study the limitations of Implicit Broadcasts since Android Oreo.
2. Get familiar with the *LocationApp*'s `ForegroundLocationService`. Pay special attention to how it uses broadcasts for inter-app and intra-app communication.

3. For this task, you should have an active route in your emulator. Go to the emulator setting > location > Routes. Here you can search for different locations and then use directions button to create a route, just like in Google Maps application.



When working on this task, you should save a Route and Play it. This way you will get constant location updates.

### Implementation:

1. Implement the functionality in *IntentSniffer* app to capture both inter-app and intra-app broadcasts, extract the location data from both and show it in `LatitudeTextView` and `LongitudeTextView`.

### Report:

1. Describe how the broadcast system works, and the use cases of implicit broadcasts.
2. Explain your code and how you can get these broadcast intents.
3. Propose and discuss at least **two** solutions to overcome intra-app broadcast sniffing.
4. But you can also sniff inter-app broadcasts. Can you explain why?
5. What is your solution for inter-app broadcast sniffing?? Propose and discuss at least **two**.
6. Implement at least one of the solutions for each case and verify that the exploits no longer work.

## 3.2 Confused Deputy Attacks

A confused deputy attack happens whenever a benign but privileged program is tricked into mis-using its authority on the system. In this task you should implement the *NotesExploit* app in a way that it tricks the *Notes* app to modify its own database.

### Preparation:

1. Read Sections 1,2, and 3 of Felt et al.'s paper on permission re-delegation in Android.
2. Make sure you know what Intents are, specially get familiar with Explicit Intents.
3. Make sure you understand how the **Notes** app, specially the `DatabaseActivity`, works.

### Implementation:

1. There are 3 methods in the *NotesExploint* app's `MainActivity`
  - `addClicked`
  - `removeClicked`
  - `showClicked`

These are button event handlers, each corresponding to a button in the *NotesExploit* app's UI. The first two handlers should send an Intent to the *Notes* app and trick it to add or remove an item from its database. The last handler should trick the *Notes*' app to send it the text of a note in return. For remove and show, it is assumed that we know the ID of the note we want to remove or see.

### Report:

1. Describe the functionality of `DatabaseActivity`, specifically explain how it modifies the database.
2. What is the root cause of the issue? Why can you modify the database content?
3. There are several ways to fix this issue, propose at least **two** solutions for it.
4. Explain your solutions, specifically describe the pros and cons of each approach.
5. Implement at least one of the solutions and verify that the exploits no longer work.

## 3.3 Leaky Content Provider

Content Providers in Android act as a central repository to store data and facilitate sharing of this data with other apps. However, if a content provider is not implemented correctly, it can leak sensitive data. In this task we focus on leaky content providers. You should misuse the content provider of the *Notes* app in a way that allows you to read the contents of a text file in the SD-Card. Obviously, your exploit app should not have permission to read from external storage.

### Preparation:

1. Make sure you understand the basics of Content Providers – we don't use the database-related methods here – pay special attention to `openFile` method.
2. Learn about Path Traversal Attacks.
3. For this task, you have to manually create a text file inside of your emulator's SD-Card. The easiest way is to create a `txt` file in your own computer and then use `adb push` command to push it to the emulator. (`adb` is inside your Android SDK installation directory, in a sub-directory called `platform tools`).

*Note:* For this task you should directly query the content provider, avoid using URI permissions and/or the actions of `DataBaseActivity`.

### **Implementation:**

1. Implement `queryContentProvider_onClicked` method inside *ContentProviderExploit* app's `MainActivity`. This method should use the Content Provider to read the text file which is inside of the emulator's SD-Card. It should show the returned text in the `resultTextView`.

### **Report:**

1. Describe the usage of content providers, and why do you think we used one in this scenario?
2. What is the problem with this implementation, why can you access SD-Card through it?
3. Explain your implementation of `queryContentProvider_onClicked` method.
4. Propose and implement a solution for this issue and demonstrate that the exploit no longer works.

## **3.4 Timing Attacks on Messengers**

In this task we will focus on timing side channel attacks on a simple messenger app. You should use the information about the send and the delivery times of messages to infer the approximate location of the users.

### **Preparation:**

1. Take a look at Sections II and III of this paper to get familiar with the concept of timing attacks on messenger applications.
2. Make sure you understand the basics of Android Debug Bridge (`adb`) and Logcat on how to access and read the logs of Android applications.
3. In a realistic scenario, the process of extracting timing information from messages is difficult and error prone. In this task, we took some measures to simplify this process. Investigate the source code of the messenger app and find out how you can extract the timing data from it.

Table 1: Location of Regional CDN Servers and RTT (round trip time) to them

Region	RTT(ms)
Skyrim	803
Morrowind	384
Hammerfell	723
Valenwood	547
Elsweyr	427
Cyrodiil	1052

Table 2: Cities and their RTT (round trip time) to their Regional CDN Server

Skyrim		Morrowind		Hammerfell	
City	RTT(ms)	City	RTT(ms)	City	RTT(ms)
Riften	113	Vivec	247	Sentinel	157
Whiterun	375	Mournhold	345	Rihad	286
Windhelm	342	Balmora	307	Taneth	336
Solitude	312	Ald'ruhn	128	Elinhir	305
Markarth	149	Blacklight	387	Dragonstar	356
Falkreath	255	Narsis	289	Hegathe	220

Valenwood		Cyrodiil		Elsweyr	
City	RTT(ms)	City	RTT(ms)	City	RTT(ms)
Falinesti	340	Anvil	209	Rimmen	129
Elden Root	244	Bruma	305	Riverhold	336
Haven	128	Bravil	260	Orcrest	302
Silvenar	72	Chorrol	189	Dune	285
Arenthia	321	Leyawiin	245	Senchal	420
Southpoint	389	Cheydinhal	326	Torval	168

- The messenger app relies on a server to send and receive data. We provide this server in a docker container. To run the server you need to install Docker by following the instructions at <https://docs.docker.com/install/>. Note that you might already have Docker installed from the previous labs.
- Once docker is installed, run the bash script `launch.sh` present in the zip folder to start the server.

The goal of this task is to use the timing information you get from the messenger app, and the data provided in Table 1 and Table 2 to infer the location of users in the app.

#### Implementation:

- Implement an app or a script to calculate the time-to-server and time-to-receiver for each user.

## Report:

1. Use the timing data from your implementation and the information provided in Tables 1 and 2, to infer the location of users. Explain the process in your report.
2. What would be the difficulties of using this side channel in real-world messengers like WhatsApp or Signal?
3. Think about the possible fixes of this problem. Explain and evaluate their pros and cons in your report.
  - In this task there is not need for the suggested fixes to be Android specific.
  - You **do not** need to implement any of the suggested fixes.

*Note 1:* The timing information at each step of the way is **never** more than 1500 milliseconds. If you are getting unusually long delays, there might be issues with your machine and/or Docker. If this is the case, please inform the TAs before moving forward with the task.

## 4 Requirements and Grading

### 4.1 Report

A zip file to be submitted via Canvas that should include the following:

1. Source code to your solutions.
2. A detailed report answering the questions in the Report section of each task. Specifically, you should fully explain the issues, the exploits, and the proposed solutions.
3. Clear statement of contributions of each group member.

*Note 1:* The fixes presented for these issues should be Android specific. This means that general fixes such as using MAC, Static Fields, or Encryption are not acceptable. The proposed fixes should also rely on Android provided functionalities.

*Note 2:* The overall architecture of the apps should not change. For example, in Task 1, removing the broadcast system is **not** an acceptable fix.

### 4.2 Grading (Max 20 points)

- You get 5 points for completing each task in Section 3. Thus, you can get a total of 20 points.
- To pass the lab you should at least solve 2 of the tasks.
- This lab does not have any extra points.

In this lab special attention will be given to the **reports** and **presentation**. Please make sure that the report is satisfactory and every group member is able to explain the solution and answer questions during the live presentation of the lab. Failure to do so may result in deduction of points.