# CS155 Project 3

Spring 2013

May 23, 2013

## Introduction

Project 3 is about Android security. It consists of three parts, where the first part is a relatively easy warmup. Each part is performed on the Android emulator included in the Android SDK, which behaves almost exactly like a real device. In Part 1, you will see the danger of storing unencrypted data on the device (as many real-world apps do). In Part 2, you will learn the basics of Android app reverse engineering, and craft an attack against a vulnerable application in order to steal the contacts from a phone using an unprivileged app. In Part 3, you will perform a different kind of attack, where an Android app can simulate fake clicks to a paid advertisement library in order to cost the advertiser money.

At the end of this document there is a section called "Getting started with Android development" which you should read before beginning. You will use adb, the Android Debug Bridge tool, to get a root shell on the device for Part 1. Before beginning Part 1, you should be able to launch the provided Android Virtual Device image in the Android emulator and access the shell using adb.

This is a newly developed project, so please pay close attention to Piazza and let the teaching staff know if anything is amiss via Piazza posts. Once you discover the "trick" to completing the attacks in this project, it becomes straightforward to solve them–so please don't share the secrets with your classmates.

Good luck!

## Part I

# Warmup: A forensics treasure hunt

We have provided you an Android Virtual Device image, which you can boot up in the Android emulator. The device contains lots of juicy data! Unfortunately, the owner has locked the device with a PIN. To retrieve useful data from the device, you'll have to access it from the Android Debugger's shell.

Retrieve at least the following information from the device:

1. List of contacts
2. Recent SMS messages
3. IM credentials from the Xabber app
4. E-mail credentials
5. *B*onus browser bookmarks, call logs, or anything thing else interesting

To get started, boot up the Android Virtual device, called `Bradley.avd`. Since the device is locked, and you don't know the PIN, you'll need to poke around using `adb`. The android documentation has an extensive explanation of how to use `adb`, but to get you started, run `adb shell` while an device is running to get a Unix shell on the device.

**HINT** Look around for `.db` files. They contain a wealth of information! Also, the user doesn't use the native e-mail app, so look for alternative e-mail clients.

### Deliverables

1. **contacts.tsv** a tab-separated text file with one contact per line, listing each contact's name, e-mail and/or phone (simply omit columns that do not exist)
2. **sms.txt** a file containing the body of each SMS on a separate line.
3. **im.txt** a file containing the user's IM *username@host* on one line and their password on a second.
4. **email.txt** a file containing the user's E-mail *username@host* on one line and their password on a second.
5. **extra.txt** anything extra you find in whatever format you choose.

## Part II

## Android app security: Reverse-engineering and privilege escalation

The virtual device's user has installed an application called TrustedApp, which is supposed to send fun SMS messages to the user's friends. TrustedApp was installed with the READ_CONTACTS and SEND_SMS permissions.

However, the developers of TrustedApp did not take CS155, so they accidentally introduced a vulnerability that allows a third-party malicious app to read the list of contacts names on the device. The vulnerability is related to the inter-component communication features of Android, which determine when one application component can send a request to another application component. Your task is to discover the details and exploit this vulnerability.

You can assume that the user will install a malicious app written by you, called EvilApp. EvilApp has the INTERNET permission, but no other permissions. EvilApp will be the app which tricks TrustedApp in order to steal the contacts.

To proceed, you will need to use the Android SDK tools to reverse-engineer TrustedApp. Make sure you read through the "Getting started with Android development" section of this document and set up the SDK tools and emulator first.

1. **Find and disassemble the TrustedApp .apk**

   Android applications are distributed in Android .apk files. These files are very similar to Java JAR files: zipped archives containing everything needed to run the app. Suggested steps for reverse-engineering TrustedApp are as follows:

   - Install the Android SDK tools, and start the provided Android image in the emulator.
   - Find and copy the TrustedApp apk off the device using the Android Debug Bridge, or adb. (This comes with the SDK and is a command-line tool that lets you manipulate the Android emulator. For example, typing "adb shell" in the terminal will give you a root shell on the device, and "adb push/pull" lets you push or retrieve files.) **Make sure adb is in your path**.
   - Now that you have the .apk file, unzip it. The AndroidManifest.xml file will appear, but it is encoded. A Google search will reveal various ways to decode it. You can also obtain the information you need from the manifest file by using "aapt," the Android Asset Packaging Tool included with the SDK, on the original .apk file using the command below. The Android manifest contains essential information about what components make up an application.

   ```
   aapt l -a [.apk file name]
   ```

   - The .apk also holds a file called "classes.dex." This file contains the compiled Android application code. You can use the "dex2jar" tool to convert to a JAR file, and then extract the JAR file to find the .class files. This tool can be found here: `http://code.google.com/p/dex2jar/`.
   - Once you have the class files, a Java decompiler can be used to view the (approximate) original Java code. A good tool to use is JD-GUI, found here: `http://java.decompiler.free.fr/?q=jdgui`.

   Now that you have the code, you will use it and your understanding of how Android applications and services work to craft your attack.

2. **You will find that TrustedApp exposes a certain component that can be accessed by EvilApp. The developers of TrustedApp tried to include *some* security: you'll find that a secret key is required. Find the key in the decompiled code and write the attack code in the provided EvilApp Eclipse starter project.**

   You will need to infer the interface in order to successfully send a request to TrustedApp. Make sure you understand how an AIDL (Android Interface Definition Language) file works: `http://developer.android.com/guide/components/aidl.html`.

   In order to test your code, you should follow these steps:

   (a) Create a fresh Android emulator image in the AVD Manager and start it. (Use the latest Android SDK version.)

   (b) Import the EvilApp starter code into Eclipse.

   (c) Install the TrustedApp .apk file on the new device using "adb install." (It's easiest to close the emulator from Part 1 before this, so adb sees there is only a single running emulator.) Verify that you can open TrustedApp on the phone and use it to view the contacts.

   (d) **Important.** On the emulator, create some contacts using the phone's normal interface. These are the contacts you are going to steal.

   (e) Run your app using the Eclipse "Run" or "Debug" commands, and it should automatically push your compiled EvilApp onto the device and open it up.

3. **It turns out the secret key varies from version to version of TrustedApp, and you want to make an attack that works on *all* versions. Find a way to retrieve the contacts from TrustedApp using EvilApp *without* your knowledge of the secret key.**

   Hint: look for more mistakes by the TrustedApp developers in the source code.

4. **The moral of the story for an app developer is that you should never roll your own security – take the time to use the established methods of the system you're building on. What should the developers of TrustedApp have done to make their app secure against the attack performed by EvilApp? What are the real-world defenses against reverse-engineering an Android app?**

### Part 2 deliverables:

1. A copy of one of the .java files from the decompiled TrustedApp
2. **MainActivity.java**, containing your attack code using the secret key.
3. **MainActivity_no_key.java**, a separate file you create containing the same attack without using the secret key you found.
4. Answers to the two questions in item 4.

# Part III
# Click-fraud

One of the common ways apps make money is by embedding third-party advertising. But how vulnerable are these libraries to abuse? In this part, you'll implement an Android app that defrauds the advertiser by generating network traffic that's indistiguishable from user clicks, without requiring your users to actually view an advertisement.

We've given you some starter code that already links to and (legitimately) uses the ad library. We've also provided the source code for the ad library, but it's missing a secret key it shares with the server. To recover the secret key, we've also provided a network trace of legitimate clicks between an app and the ad server. Once you've obtained the secret, create an app that generates a large number of clicks without disturbing the user's experience. Specifically, the user should not even know that clicks are being generated on her behalf unless she examines the network. You are allowed to use the ad library if you wish, but you don't have to.

Each app needs a unique application id to communicate with the ad server. Get an app id by registering at `http://adlib.mappend.net`. Once you've registered, you can monitor the number of clicks you've generated at `http://adlib.mappend.net/stats/[yourappid]`.

## Setup

1. Extract the starter code for the client application from `jack.tar`. You can either import the project into Eclipse, or use an editor of your choice and build with Apache Ant.
2. Generate an application id at `http://adlib.mappend.net`. You'll need this application id to generate unique clicks. If you want to test the starter code, replace the bogus application id in `edu.stanford.cs.jack.MainActivity` with your generated one, install the app on a virtual device and run it.
3. Determine the shared secret used by the ad library by examining the network trace in `jack_trace.log`. This is a trace of a user running the starter code and clicking through to an ad. The app is running with application id 'defc505d488e0a00ff92f7b0ac1c38c7'.
4. Modify the starter code to steal clicks from the ad server without affecting the user interface. If you find it helpful to ad UI widgets, that's fine, as long as it is not apparent to a user interacting with the app (i.e. you are free to add hidden UI elements).

Finally, modify the ad library source code to improve it's resiliance to click fraud. You are free to make whatever changes you want to the ad library. While you cannot technically change the server, you may modify the network protocol as long as you document what changes, if any, are necessary on the server side.

## Hints

1. Look at the adlib source code and figure out what the protocol between the library and server is.
2. A click is registered only when the server thinks a user has viewed the advertisement in a web browser.
3. Pay special attention to HTTP headers in the network trace. Which ones might the ad server use to determine who is making requests?

## Deliverables

1. **secret.txt** a text file containing the secret you extracted from the network trace on a single line.
2. **jack.tar.gz** a gzipped tarball of your fraudulent app.
3. **adlib.tar.gz** a gzipped tarball of your modified ad library. The tarball should include a file called CHANGES that summarizes what you modified.

# Getting started with Android development

1. Get the Android SDK tools for your platform at `http://developer.android.com/sdk/index.html`. Follow the instructions for your platform at `http://developer.android.com/sdk/installing/index.html`.
   (a) You'll want to add adb and aapt to your system's path. To be safe, add the following folders:
      i. sdk/tools/
      ii. sdk/platform-tools/
      iii. sdk/build-tools/android-4.2.2/
2. Look over the first page of the Android docs: `http://developer.android.com/guide/components/fundamentals.html`. The opening section contains essential information on how Android works. You can skip the last two sections, "Declaring application requirements" and "Application Resources."
3. Start Eclipse, and import the starter code
   (a) Eclipse is found in the "eclipse" folder of the SDK package. You should be able to just run it. If you get an error about a missing package you need to run Eclipse, look at the troubleshooting page found with the SDK instructions.
   (b) Copy the **EvilApp**, **Jack**, and **Adlib** folders containing the starter code into your Eclipse workspace. You should see the projects in the "Package Explorer" pane on the left.

4. Start the Android image in the emulator

   (a) In Eclipse, click the "Android Virtual Device Manager" icon at the top. A label at the top will say "List of existing Android Virtual Devices located at [folder name]."
   (b) Copy **Bradley.avd** and **Bradley.ini** into this folder.
   (c) Edit **Bradley.ini** such that the "path=" fields contains the path to **Bradley.avd**, i.e. [folder name]/Bradley.avd.
   (d) Hit "Refresh" in the virtual device manager. You should see the device image appear.
   (e) You may need to press "Repair" in the Virtual Device Manager, but then you should be able to click "Start" to run the emulator.

5. Miscellaneous tips on how to get things up and running smoothly will be posted on a pinned Piazza post.