CS155 Project 3

Spring 2014

CS 155 Staff

May 18, 2014

Due date

- This project is due on **Wednesday, June 4, 2014** at **11:59 PM**.
- **You may not use more than one late day for this project.** Hence, the hard deadline for this project is **Thursday, June 5, 2014** at **11:59 PM**.

Submission instructions

- Make sure your solutions/part**[1,2,3]** directories contain the deliverables mentioned below. Each directory should contain the deliverables as mentioned in each part; misplaced deliverables may not receive credit.
- Run the following command from the project root directory to generate the submission tarball submission.tar.gz:
  make submission
- Upload the submission tarball to a Stanford cluster machine (i.e. myth or corn).
- To submit, run the following command from the directory containing the tarball:
  /usr/class/cs155/bin/submit proj3
- Follow the on-screen instructions.
- You can check whether we have received your submission by going to /usr/class/cs155/submissions/proj3.

Introduction

Project 3 is about Android security. It consists of three parts, where the first part is a relatively easy warm-up. 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 extract the contacts from a phone using an unprivileged app. In Part 3, you will perform a different kind of attack, where you will be redirecting HTTPS requests made by an app and understand the need for certificate pinning.

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.

Part 1

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 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. The list of contacts.
2. Recent SMS messages.
3. IM credentials from the Xabber app.
4. E-mail credentials.
5. Bonus browser bookmarks, call logs, or anything else that is 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 elaborate explanation of how to use adb, but to get you started, run adb shell while a 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 email app, so look for alternative e-mail clients.

Part 1 Deliverables

1. **contacts.tsv** a tab-separated text file with one contact per line, listing each contact's name, email address and/or phone number (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 email address *username@host* on one line and their password on a second.
5. **extra.txt** anything else you find, in whatever format you choose.

Part 2

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 CS 155, so they accidentally introduced a vulnerability that allows a malicious third-party app to read the list of contact 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. 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 extract the contact list.

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. **Setup: 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. The steps suggested 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 [filename.apk]

   - 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://jd.benow.ca/`.

   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. **Programming task 1:** 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 "Target", i.e. Android 4.4.2 - API Level 19.)

   (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 a single running emulator only.) 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 extract.

   (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. **Programming task 2:** 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. **Q&A time:** 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.

   (a) What should the developers of TrustedApp have done to make their app secure against the attack performed by EvilApp?

   (b) 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.txt**, a text file containing your answers to the two questions in item 4.

Part 3

Certificate pinning

You have been provided with the source code for an app, called "Click Me", that loads an ad from an ad server. You have also been provided with two python scripts in ad-servers/: **good-server.py** and **bad-server.py**. These scripts are used to run the "good" and the "bad" ad servers locally on your computer, on ports 4443 and 4444 respectively. The ad images served by these servers are co-located in the same folder, along with the SSL credentials used by them.

When run, the "Click Me" app tries to load an ad by sending a GET request to `https://localhost:4443/` (Note that the filename of the image is not part of the request.) Although the app uses HTTPS to get the ad, it is still vulnerable to a man-in-the-middle attack.

1. **Setup: Intercept the request sent by the app to the "good" ad server, forward the request to the "bad" ad server, and load the ad provided by the "bad" ad server in the app.**
   - Import the ClickMe project into Eclipse and run the app on a fresh Android emulator image. (Use the latest "Target", i.e. Android 4.4.2 - API Level 19.)
   - Run both ad servers using the python command (i.e. python [filename.py]). You'll need to install Python 2.7 on your computer first.
   - Download and run BurpSuite from: `http://portswigger.net/burp/burpsuite_free_v1.6.jar` The "Proxy" component of this suite will help you intercept requests. Take a look at the "Options" tab under it. (Read more about these options at `http://portswigger.net/burp/help/proxy_options.html`).
   - On the emulator, navigate to the APN settings ("Settings" >"More..." >"Mobile networks" >"Access Point Names") and click on the sole tmobile APN in the emulator.
   - Change the proxy and port to the same IP address / port combination as the one being monitored by Burp Proxy. Specifically, make sure that the IP / port combination matches the running interface in your "Proxy Listeners" list under the "Options" tab in Burp Proxy.
   - All requests from your emulator should now go through the proxy. Confirm this by using the Browser app in the emulator and checking if the "HTTP History" in Burp Proxy shows any activity. It is normal to see a security warning when accessing a site via HTTPS using this proxy. (If the default IP address being used by Burp Proxy doesn't work for you, i.e. you get "ERR_PROXY_CONNECTION_FAILED" in the browser, try a different IP address from the list generated by running ifconfig / ipconfig on your computer. There should be no reason to change the port though.)
   - Make sure that "Intercept is off" in Burp Proxy. Run the unmodified "Click Me" app on the same emulator. Confirm that the ad image from the "good" server is displayed in the app. (You should see a white hat.)
   - Make sure that "Intercept is on" in Burp Proxy. Close and run the unmodified app again. Intercept and redirect its request from the "good" to the "bad" ad server. Confirm that the ad image from the "bad" server is displayed in the app. (You should see a black hat instead of a white hat.)
   - You'll notice that Burp Suite modifies all HTTPS connections, by default; this can be disabled using the "SSL Pass Through" option in the Burp Proxy settings. This will be useful for checking the behavior of your modified app in the next step.

2. **Programming task: Implement certificate pinning in the app, in order to ensure that only ads from the "good" ad server load in the app.**

   - Look through `http://developer.android.com/training/articles/security-ssl.html` to understand how certificate pinning works.
   - Modify the app source code to pin the certificate used by the "good" ad server. Your app should load the ad image only if it is able to connect to the "good" ad server without any interference ("SSL Pass Through" will need to be enabled here). In all other cases, it should display the default error image (provided in the source code). Do not modify the python ad server scripts.

Note

You should always use the proxy for this part of the project. We will run your app on an emulator configured such that all its requests go through the proxy.

Part 3 Deliverables

1. **ClickMePinned.tar.gz**, an archive of the Eclipse project of your modified app, generated via "File" >"Export..." >"Archive File" in Eclipse.
2. **Answers.txt**, a text file containing your answers to the following questions:

   (a) Why do you see a security warning in the Browser app when accessing a site via HTTPS using Burp Proxy initially?

   (b) Why does the image from the "bad" server get loaded in the unmodified app? Explain with reference to the source code of the app. Would you classify this behavior as "fail-open" or "fail-close"?

   (c) List all the steps you took to extract and embed the required SSL credentials into the source code. Start by stating where you acquired the required credentials from. End by stating where the credentials are located in your source code.

   (d) Identify a situation in which certificate pinning is not a practical solution for secure mobile app development. Why is it not practical in this situation? What would you use instead, to protect against similar attacks?

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) It might be useful 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-4.4.2/

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 app requirements" and "App 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 run it directly. 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) Import the desired Android project (e.g. EvilApp) into your Eclipse workspace via File >Import... You should see the imported project(s) 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=" field 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) Based on the contents of **Bradley.ini**, figure out if you need an older SDK platform, and install it via the "Android SDK Manager" in Eclipse, found right next to the "Android Virtual Device Manager". (You should restart Eclipse after this step.)

   (f) "Repair" the image in the device manager, if necessary, and click "Start" to run the emulator.