CS155: Computer Security **Spring 2016**

Project #3

Due: Wednesday, June 1 (at most one late day allowed)

Introduction

This project is about Android security and network attack detection. It consists of three parts. Part one is a warm-up that will teach you the importance of keeping sensitive data encrypted. Part two will have you learn about Android app permissions, reverse-engineer a vulnerable Android app, and craft an attack. Part three will have you read a packet dump and detect a network attack.

Setup

We get more questions about setup on this project than any other, so please start early and follow these instructions carefully before getting started! These instructions cover the setup for both part 1 and part 2.

1. Install Android Studio from https://developer.android.com/sdk/. Once your download starts, you will be redirected to a set of installation instructions. Follow them carefully! If you need to get back to the installation instructions later, they are available at https://developer.android.com/sdk/installing/.

2. Start Android Studio and select "Import project (Eclipse ADT, Gradle, etc.)" from the start-up screen or, if Android Studio skips the start-up screen, click File→New→Import project...

3. Select the "EvilApp" folder from the starter code and import the project.

4. Open the SDK Manager by selecting Tools→Android→SDK Manager or using the toolbar icon ().

   (a) Take note of the Android SDK location.

   > Common paths are `~/.android/sdk/` (on Linux), `~/Library/android/sdk/` (on Mac), and `C:\Users\<user>\AppData\Local\Android\sdk\` (on Windows).

   (b) Under the "SDK platforms" tab, install Android API Levels 23 and 17 by checking the corresponding boxes and clicking "apply."

   (c) Under the "SDK tools" tab, install the Android SDK build tools, Android SDK platform tools, Android SDK tools, and Android support repository.

5. Open the AVD Manager by selecting Tools→Android→AVD Manager or using the toolbar icon ().

   (a) Click the "create virtual device" button and select a device (e.g. Nexus 6P).

   (b) On the next screen ("System Image"), select the "X86 Images" tab and download the "Marshmallow, API level 23, x86_64 ABI, Android 6.0" image (*without* Google APIs).

   (c) After selecting Marshmallow as the system image, finish creating the AVD.

6. In the AVD manager, right click on the AVD you just created and select "show on disk." This will open the Android AVD folder. Move Bradley.avd and Bradley.ini from the BradleyAndroidImage folder of the starter code into this folder.

   > Common AVD folder paths include `~/.android/avd/` (on Linux), `~/Library/android/avd` (on Mac), or `C:\Users\<user>\AppData\Local\Android\avd\` (on Windows).

7. Open Bradley.ini in a text editor. Edit the "path" line to refer to the location of Bradley.avd (i.e. the absolute path of the place you just copied it to). Note: tildes will not work here! Use the full, absolute, spelled-out path.

   > Common paths include `/home/<user>/.android/avd/Bradley.avd` (Linux), `/Users/<user>/Library/android/avd/Bradley.avd` (Mac) or `C:\Users\<user>\AppData\Local\Android\avd\Bradley.avd` (Windows).

8. Return to the AVD manager and click the refresh button at the bottom-right. You should now see two images: the Nexus 6P and Bradley.

   (a) If the Bradley image says "failed to load" in the Actions column, you likely need to check your work editing the Bradley.ini file in the previous step (you may also want to restart Android Studio).

   (b) If the Bradley image has a play icon and a pencil icon in the actions column, try to run the image. If it starts, great! Otherwise, close and re-open the AVD manager and repeat this step.

   (c) If the Bradley image says "download" in the actions column, right-click on the image and select "edit." The "System Image" screen should appear again. Select the "x86 images" tab, download the "Jellybean, API level 17, x86 ABI, Android 4.2" image (again, without Google APIs), and use it as Bradley's system image. You should now be able to run the Bradley image.

9. Open a terminal and try to run the commands `adb` and `aapt`.

   (a) If these commands both work, you're done!

   (b) Otherwise, look in `<sdk>/platform-tools` and `<sdk>/build-tools` (or their subfolders) to find these programs, where `<sdk>` is the Android SDK folder from step 4a.

   (c) From the Android SDK folder, try running `./build-tools/23.0.3/aapt` and `./platform-tools/adb` (or whatever the equivalent paths are on your system) to make sure both programs work.

   (d) If you want to be able to run `aapt` and `adb` from any directory, you may need to add the programs to your system's PATH. See here for what that means.

If you need any help with setup, please watch the section video, come to office hours, or ask on Piazza! The point of this project is not to test your Android Studio setup skills, and we want to help you get started as painlessly as possible.

Part 1: A forensics treasure hunt

We have provided you an Android Virtual Device image (called Bradley.avd). Boot it up using instructions above, if it's not already running. On the default emulator, this will take about 3

minutes. Since the device is locked, you will need to access it using adb. Refer to the documentation for adb, or run `adb shell` to explore on your own.) 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.

Hint: Look around for .db files. Also, the user doesn't use the native email app, so look for alternative e-mail clients.

Deliverables

1. **contacts.csv** a comma-separated text file with one contact per line, listing each contact's name, phone number, and email address (leave blank any columns that do not exist, e.g. "a,,c" or ",b,").
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 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. Design EvilApp to trick TrustedApp into extracting the contact list. To proceed, you will need to use the Android SDK tools to reverse-engineer TrustedApp.

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:

1. Start the provided Android image (Bradley.avd) in the emulator.
2. Find and copy the TrustedApp APK off the device using adb. (Hint: `adb push/pull`)
3. Now that you have the .apk file, unzip it. The AndroidManifest.xml file, which describes the components of the application, will appear in an encoded format. A Google search will reveal various ways to decode it. You can also obtain this information from the manifest file by using aapt on the original .apk file using the command `aapt l -a [filename.apk]`.
4. 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.
5. 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.

Now use the extracted code and your understanding of how Android applications and services work to craft your attack!

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 starter project. You will need to infer the interface in order to successfully send a request to TrustedApp and define that interface inside EvilApp. Make sure you understand how an AIDL (Android Interface Definition Language) file works.

Your code must call the `showContacts` function in the starter code. This function will display your stolen contacts in EvilApp, and more importantly, we will use it for grading.

Testing

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

1. Open the AVD manager and start the other emulator you created when setting up the project, the one running Android 23 on a Nexus 6P (or whatever device you chose).
2. On the emulator, create some contacts using the phone's normal interface. These are the contacts you are going to extract. Save the contacts locally on the phone, not synced with Google. If you don't have the option to save the contacts locally, you may have accidentally downloaded an Android image with Google APIs.
3. Install the TrustedApp .apk file on the new device using "adb install." (It's easiest to close the Bradley emulator from Part 1 before this, so adb sees that there is only one running emulator.) Verify that you can open TrustedApp on the phone and use it to view the contacts.
4. Run EvilApp using the "Run" or "Debug" commands in Android Studio, and Android Studio should automatically push your compiled EvilApp onto the device and open it up.

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.

Q&A time

The moral of the story for an app developer is that you should never roll your own security. Instead, use the established methods of the system you're building on.

1. What should the developers of TrustedApp have done to make their app secure against the attack performed by EvilApp?
2. What are the real-world defenses against reverse-engineering an Android app?

Deliverables

1. **ReadContactsService.java** a Java file 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. **[name censored].aidl** (optional) an AIDL file to access TrustedApp's interface.
5. **Answers.txt** a text file containing your answers to the two questions above.

Part 3: Packet sniffing

You have been given a dump of packet header information for the network that your mobile device is on (trace.txt). Your device's IP address is 10.30.22.101. Lately, there have been a number of sketchy occurrences on this network. In order to learn more about network security, your job is to sift through the packet header information and detect some anomalous behavior on the network. Since the volume of packets is large, it is expected that you will write scripts in the language(s) of your choice to do the following:

1. Your mobile device has accessed a number of websites. List 5 IP addresses of websites that it have been accessed by your mobile device. (Hint: What port is typically used for webservers? Your mobile device's IP address is 10.30.22.101. There may be more than 5 valid answers to this question.)
2. Someone sketchy has been looking for attack vectors into hosts on this network by scanning for open ports on a machine. (Hint: What kind of pattern would you expect from a host that is port scanning other hosts? How might you isolate this pattern in the packet dump?)
   (a) What is the IP address of the origin of the port scan?
   (b) What is the IP address of the host that was scanned?
   (c) What range of ports was scanned?

3. There has been an unusually high volume of syn packets going from one host on this network to another host on this network. This can be indicative of a type of Denial of Service (DoS) attack called a syn flood.

   (a) What is the IP address of the syn flooding sender?

   (b) What is the IP address of the syn flooded receiver?

   (c) How many syn packets were sent from this sender to this receiver?

4. Some malware on your phone has caused it to behave improperly and inject a malicious packet in the network. More specifically, your phone has sent a packet during a TCP connection that it should not have sent. Doing so can sometimes cause problems on the network and crash hosts that are not implemented correctly. What is the checksum value of the injected packet? Here are some helpful facts.

   - TCP retransmission may cause a client to send a packet with a sequence number that has already been sent, and possibly acknowledged.
   - A retransmitted packet should have the same content as the original packet, but may have a different timestamp, and therefore a different checksum.
   - The TCP checksum is computed over the TCP pseudo-header and data. If two packets have different checksums, and both are correct, it means that either their headers are different, the data they carry is different, or both.

Deliverables

1. **PacketSniffingAnswers.txt** A text file containing your answers to questions 1–4 above.
2. Any scripts you wrote to answer the above questions.

Submitting

1. Make sure your solutions/part[1,2,3] directories contain the deliverables mentioned in this writeup. Each directory should contain the deliverables as mentioned in each part; misplaced deliverables may not receive credit. **Many items will be auto-graded, so be careful to follow the given formats exactly.**
2. Run the following command from the project root directory to generate the submission tarball submission.tar.gz: `make submission`.
3. Upload the submission tarball to a Stanford cluster machine (myth or corn).
4. To submit, run the following command from the directory containing the tarball: `/usr/class/cs155/bin/submit proj3`. Follow the on-screen instructions.
5. You can check whether we have received your submission by checking inside `/usr/class/cs155/submissions/proj3`.