

# Lab #8 : Mobile Malware

CSE3801 : Introduction to Cyber Operations

Anthony Agatiello, James Spies, Robert Whitman

---

## Overview

### Android

In this lab, the goal was to reverse some Android packages to find one that seems like it could be malware based upon our lesson in class. Many of the APKs on the [apkpure](http://apkpure.com) website did not seem that interesting, however, I came across an application called “Super Fast Charger 2019” that was interesting at first glance. Within each APK is an `AndroidManifest.xml` file that contains a list of all the privileges granted to the app. `META-INF` contains many of the files related to certificates and different metadata. The last of interest to us is the `classes.dex` file which contains the byte code of the APK. This byte code can be roughly converted to java code by using JADX. By analyzing these files, we can determine whether or not an APK is potentially harmful in some way.

### iOS

The process of creating malware on iOS is extremely different than on Android or other operating systems. This is mainly due to Apple’s notoriously strict application review process and built-in binary encryption for all apps on the App Store, which are packaged as an IPA file. As a result of this, any malware that would normally be in an app would be discovered either with a static analyzer or by the person who tests it. With that being said and assuming that you can use an alternate method of distribution, it is still very possible for iOS malware to be written into an app, or even injected into a well-known application’s binary that’s redistributed on certain websites for which App Store approval is not possible (ie. illegal movie/music streaming apps).

## Methodology

### Android

First, I navigated to [apkpure.com](http://apkpure.com) to find an interesting APK to reverse and examine. The one I chose was “Super Fast Charger 2019”. This is the APK I decided to pick based upon its bold name and terrible application description. In addition, it is a relatively new and very unpopular app, thereby increasing the chance that it is malicious and has not yet been detected. Using [javadecompilers.com/apk](http://javadecompilers.com/apk), I was able to pass in my Super Fast Charger APK file so that I could analyze its contents. First, I looked into the

AndroidManifest.xml file, shown below.

```
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
<uses-permission android:name="android.permission.ACCESS_WIFI_STATE"/>
<uses-permission android:name="android.permission.BLUETOOTH"/>
<uses-permission android:name="android.permission.BLUETOOTH_ADMIN"/>
<uses-permission android:name="android.permission.CHANGE_WIFI_STATE"/>
<uses-permission android:name="android.permission.CHANGE_NETWORK_STATE"/>
<uses-permission android:name="android.permission.WRITE_SETTINGS"/>
<uses-permission android:name="android.permission.KILL_BACKGROUND_PROCESSES"/>
<uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"/>
<uses-permission android:name="android.permission.WAKE_LOCK"/>
<uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
<uses-permission android:name="android.permission.VIBRATE"/>
<uses-permission android:name="android.permission.BATTERY_STATS"/>
```

Many things stand out in this XMLfile. You’d expect a battery application to be able to view your battery stats, kill background processes, and be able to toggle your wifi, which it does. However, this application has the “WAKE\_LOCK” permission, which means it can [“wake up the screen or the CPU and keep it awake to complete some work”](#). The purpose of an idle Android device falling asleep is to preserve battery. This permission given to the battery saving app is in direct contradiction to its claimed mission to preserve your battery. This was a big red flag. There are additional permissions that do not make sense within this app’s advertised scope. The ability to have access to fine location and Bluetooth are two additional permissions that this app should not have. The permission ACCESS\_FINE\_LOCATION would allow the app to know the precise location of the device, which an app like this should not need at all. The BLUETOOTH and BLUETOOTH\_ADMIN permissions would allow the app to discover, pair with, and connect to bluetooth devices, which is not necessary for a battery saving app. In addition, the ability to “RECEIVE\_BOOT\_COMPLETED” means that this application can technically startup every time your device boots.

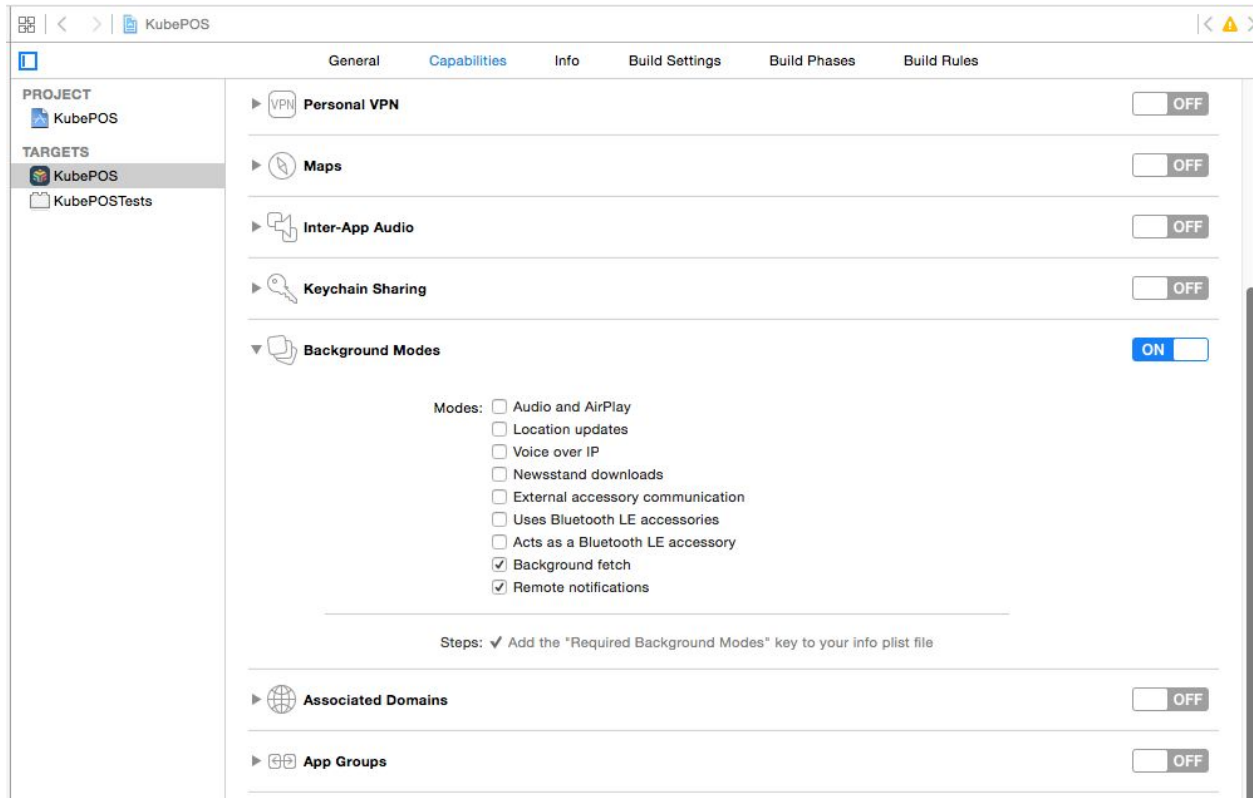
Within the javadecompilers website, I selected to decompile the dex file with JADX. There did not seem to be anything of interest in the source code of the application. The following is an example of the code that enables the application to turn on airplane mode to preserve battery life.

```
public void setAirplane(boolean z) {
    Editor edit = activity.getSharedPreferences("AIRPLANE", 0).edit();
    edit.putBoolean("AIR", z);
    edit.apply();
}

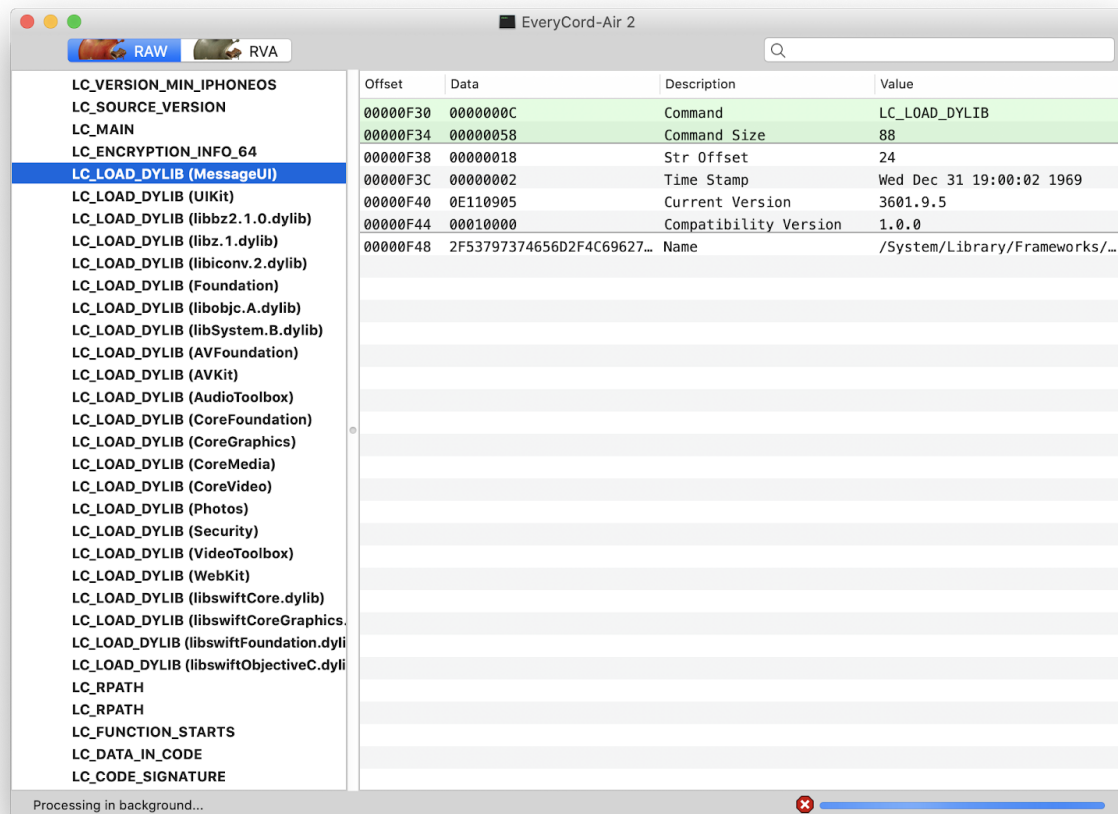
public boolean getAirplane() {
    return activity.getSharedPreferences("AIRPLANE", 0).getBoolean("AIR", false);
}
```

## iOS

If we wanted to write our own code in an iOS application that we did not build, we first have to decide what we want the malware to do. Similar to Android's manifest XML file, iOS also has a PLIST file that determines what permissions the app (or our injected code) can have, titled "Entitlements.plist". However, there is not usually anything in here that can determine whether or not the application is malicious, since there are fairly limited permission options to begin with.



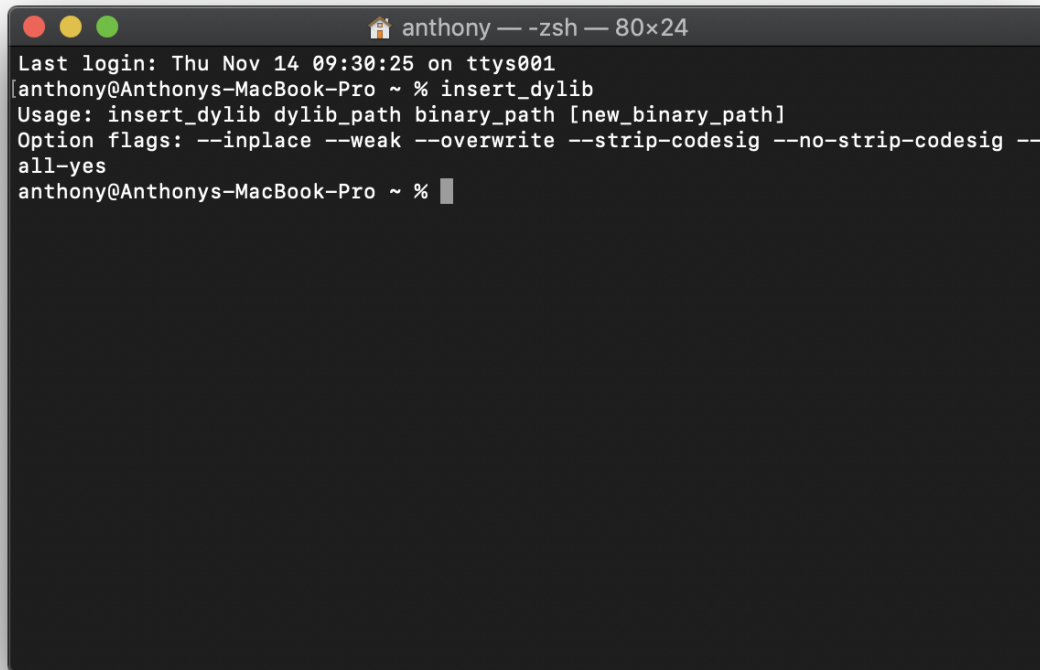
Since we have our code built, we now need to somehow load it into the binary. To start, the first step should be to load the current binary into something that can display what is currently being loaded into it at runtime:



As seen in the screenshot, the very first couple of instructions are to load in the dependencies the app needs using the `LC_LOAD_DYLIB` command. So, all we need to do is copy our DYLIB (dynamic library) to the same directory the binary itself is in, insert a `LC_LOAD_DYLIB` instruction specifying the path (see screenshot below), and lastly, code sign the entire thing.

Example:

```
$ insert_dylib "@executable_path/Malicious_Library.dylib"
```

A terminal window titled 'anthony — zsh — 80x24' with standard macOS window controls (red, yellow, green buttons). The terminal shows the following text:

```
Last login: Thu Nov 14 09:30:25 on ttys001
[anthony@Anthonys-MacBook-Pro ~ % insert_dylib
Usage: insert_dylib dylib_path binary_path [new_binary_path]
Option flags: --inplace --weak --overwrite --strip-codesig --no-strip-codesig --
all-yes
anthony@Anthonys-MacBook-Pro ~ %
```

Code signing is one of the most well-known security advantages of iOS, and it ensures that binaries only with Apple's own signature, or a developer signature, can be open or installed on the device. The most recent way people get around this is telling users to use a service which will call Apple Developer APIs that will sign the applications with their own Apple ID, which requires a computer. Since this is already a fairly convoluted process, most people would probably stop trying to get the app working on their device at this point. However, if the user wants it installed badly enough, this could essentially bypass most of Apple's "normal" restrictions, especially if they are downloading the IPA file from an untrusted source.

## Results

### Android

Though the developer of this battery saver app included a lot of application permissions, there is nothing evident in the code that points to this application acting as any type of malware. It is very important to carefully review apps that have access to so many permissions because an app can become more powerful given more permissions. An app that has permissions to send SMS messages could be an example of a potentially malicious application by having the ability to text certain numbers to pay for subscriptions. Decompiling the DEX file can allow you to analyze the java code and discover malicious code, however in this case, I did not discover anything malicious besides the abusable privacy permissions.

In classes.dex/sources/com/vinhashapp/fastcharger/homeactivity.java, there is code that shows why access to Bluetooth and wifi are needed.

```
public void checkBatteryState() {
    int i;
    this.turnOnConnectionCheck = true;
    try {
        i = getActivity().registerReceiver(null, new IntentFilter("android.intent.action.BATTERY_CHANGED")).getIntExtra(NotificationCompat.CATEGORY_STATUS, -1);
    } catch (NullPointerException unused) {
        i = 1;
    }
    switch (i) {
        case 2:
            checkNetworkStatus();
            this.fast.setText(C0553R.string.stop);
            Toast.makeText(getActivity(), getResources().getString(C0553R.string.thank_for_using), 1).show();
            this.thread.run();
            this.superkey = true;
            wifi();
            btDisable();
            killing();
            checkAirplane();
            turnOnDataConnection(false, getActivity());
            isCharging = true;
            return;
        case 3:
            this.buttonCounter--;
            Toast.makeText(getActivity(), getResources().getString(C0553R.string.plug_in_charger_first), 1).show();
            isCharging = false;
            return;
        case 4:
            checkNetworkStatus();
            this.fast.setText(C0553R.string.stop);
            Toast.makeText(getActivity(), getResources().getString(C0553R.string.thank_for_using), 1).show();
            this.thread.run();
            this.superkey = true;
            wifi();
            btDisable();
            killing();
            turnOnDataConnection(false, getActivity());
            checkAirplane();
            isCharging = true;
            return;
        case 5:
            checkNetworkStatus();
            this.fast.setText(C0553R.string.stop);
            Toast.makeText(getActivity(), getResources().getString(C0553R.string.thank_for_using), 1).show();
            this.thread.run();
            this.superkey = true;
            wifi();
            btDisable();
            killing();
            turnOnDataConnection(false, getActivity());
            checkAirplane();
            isCharging = true;
            return;
        default:
            this.buttonCounter--;
            return;
    }
}

public void btDisable() {
    if (this.bluetoothAdapter.isEnabled()) {
        this.bluetoothAdapter.disable();
    } else {
        this.bluetoothAdapter.disable();
    }
}

public void btEnable() {
    if (this.bluetoothAdapter.isEnabled()) {
        this.bluetoothAdapter.enable();
    } else {
        this.bluetoothAdapter.enable();
    }
}
```

As far as using Bluetooth goes, it really does what it says it does. Enabling or disabling Bluetooth to help improve battery life if the app deems it necessary.



```

public void wifi() {
    ((WifiManager) getContext().getApplicationContext().getSystemService("wifi")).setWifiEnabled(false);
}

public void wifiEnable() {
    ((WifiManager) getActivity().getApplicationContext().getSystemService("wifi")).setWifiEnabled(true);
}

```

The same can be said for access to wifi settings. The app enables or disables wifi access to help improve battery life.

## iOS

The result of having malware on an iOS device by somebody injecting unwanted code, is that the unwanted code still cannot get a lot of access over your device. Firstly, most of the permissions that are specified in the entitlements file are only enabled when the app is opened and in the foreground. For example, if a user grants camera permission, the app will only be able to access the camera API when the GUI is visible to the user. And on top of that, the permissions that are available to be granted are probably not anything that could be used to specifically make malware. There are no options similar to Android such as “Kill Background Processes”, “Change Network State”, etc. since all applications are sandboxed to their own container, meaning they cannot access anything outside of it and change system settings.

## Discussion

### Android

Analyzing the uses-permission elements in the application manifest is a great way to find potentially suspicious APKs. After going through about a dozen or so, the battery saver application seemed to have the most permissions and seemed the most interesting. Some of the other APKs I unpacked were “flappy bird”, “flappy trump” and “黑色沙漠”. At first, I thought it would be a good idea to first, unpack the original flappy bird game to understand what a game like that requires and decided to unpack the similar app, flappy trump. I was mainly looking for stark differences in those two apps as they should be relatively similar. Neither application caught my eye. Though I found some concerning privacy permission abuses, I did not discover any malicious looking API’s or hard-coded passwords.

### iOS

Despite the fact that apps with malware on Android and iOS are made and thought about in very different ways since iOS generally has more security precautions in place, they definitely can still exist outside of the App Store by injecting malicious code into the binary, redistributing the app file, and getting the user to codesign it. But because this process is time-consuming and everything happens outside of the App Store, malware that spreads on a large scale is not really a possibility for iOS.