23

HACKING ANDROID APPS

You've spent the entirety of this book thus far learning to hack web applications. The majority of bug bounty programs offer bounties on their web apps, so mastering web hacking is the easiest way to get started in bug bounties, as it will unlock the widest range of targets.

On the other hand, mobile hacking has a few more prerequisite skills and takes more time to get started. But because of the higher barrier to entry, fewer hackers tend to work on mobile programs. Also, the number of mobile programs is rising as companies increasingly launch complex mobile products. Mobile programs can sometimes be listed under the Mobile or IoT sections of the company's main bug bounty program. This means that if you learn to hack mobile applications, you'll likely file fewer duplicate reports and find more interesting bugs.

Despite the more involved setup, hacking mobile applications is very similar to hacking web applications. This chapter introduces the additional skills you need to learn before you begin analyzing Android apps.

Companies with mobile applications typically have both Android and iOS versions of an app. We won't cover iOS applications, and this chapter is by no means a comprehensive guide to hacking Android applications. But, along with the previous chapters, it should give you the foundation you need to start exploring the field on your own.

NOTE *One of the best resources to reference for mobile hacking is the OWASP Mobile Security Testing Guide (*https://github.com/OWASP/owasp-mstg/*).*

Setting Up Your Mobile Proxy

In the same way that you configured your web browser to work with your proxy, you'll need to set up your testing mobile device to work with a proxy. This generally involves installing the proxy's certificate on your device and adjusting your proxy's settings.

If you can afford to do so, acquire another mobile device, or use one of your old devices for testing. Mobile testing is dangerous: you might accidentally damage your device, and many of the techniques mentioned in this chapter will void the device's warranty. You can also use a mobile emulator (a program that simulates a mobile device) for testing.

First, you'll need to configure Burp's proxy to accept connections from your mobile device, because by default, Burp's proxy accepts connections only from the machine Burp is running on. Navigate to Burp's **Proxy ▸ Options** tab. In the Proxy Listeners section, click **Add**. In the pop-up window (Figure 23-1), enter a port number that is not currently in use and select **All interfaces** as the Bind to address option. Click **OK**.

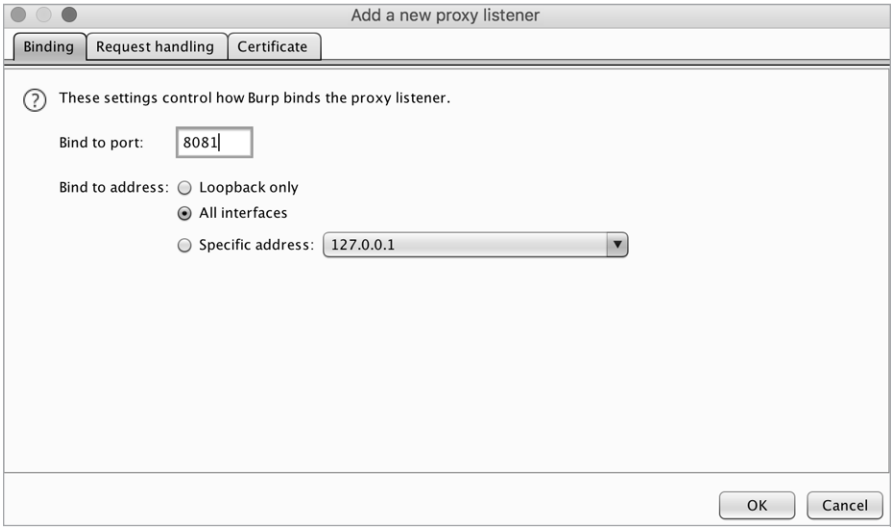

Figure 23-1: Setting up Burp to accept connections from all devices on the Wi-Fi network

Your proxy should now accept connections from any device connected to the same Wi-Fi network. As such, I do not recommend doing this on a public Wi-Fi network.

Next, you'll configure your Android device to work with the proxy. These steps will vary slightly based on the system you're using, but the process should be some version of choosing **Settings ▸ Network ▸ Wi-Fi**, selecting (usually by tapping and holding) the Wi-Fi network you're currently connected to, and selecting **Modify Network**. You should then be able to select a proxy hostname and port. Here, you should enter your computer's IP address and the port number you selected earlier. If you're using a Linux computer, you can find your computer's IP address by running this command:

```
hostname -i
```

If you are using a Mac, you can find your IP with this command:

```
ipconfig getifaddr en0
```

Your Burp proxy should now be ready to start intercepting traffic from your mobile device. The process of setting up a mobile emulator to work with your proxy is similar to this process, except that some emulators require that you add proxy details from the emulator settings menu instead of the network settings on the emulated device itself.

If you want to intercept and decode HTTPS traffic from your mobile device as well, you'll need to install Burp's certificate on your device. You can do this by visiting *http://burp/cert* in the browser on your computer that uses Burp as a proxy. Save the downloaded certificate, email it to yourself, and download it to your mobile device. Next, install the certificate on your device. This process will also depend on the specifics of the system running on your device, but it should be something like choosing **Settings ▸ Security ▸ Install Certificates from Storage**. Click the certificate you just downloaded and select **VPN and apps** for the Certificate use option. You'll now be able to audit HTTPS traffic with Burp.

Bypassing Certificate Pinning

Certificate pinning is a mechanism that limits an application to trusting predefined certificates only. Also known as *SSL pinning* or *cert pinning*, it provides an additional layer of security against *man-in-the-middle attacks*, in which an attacker secretly intercepts, reads, and alters the communications between two parties. If you want to intercept and decode the traffic of an application that uses certificate pinning, you'll have to bypass the certificate pinning first, or the application won't trust your proxy's SSL certificate and you won't be able to intercept HTTPS traffic.

It's sometimes necessary to bypass certificate pinning to intercept the traffic of better-protected apps. If you've successfully set up your mobile device to work with a proxy but still cannot see the traffic belonging to your target application, that app may have implemented certificate pinning.

The process of bypassing cert pinning will depend on how the certificate pinning is implemented for each application. For Android

applications, you have a few options for bypassing the pinning. You can use *Frida*, a tool that allows you to inject scripts into the application. You can download Frida from *https://frida.re/docs/installation/*. Then use the Universal Android SSL Pinning Bypass Frida script (*https://codeshare.frida.re/@pcipolloni/universal-android-ssl-pinning-bypass-with-frida/*). Another tool that you could use to automate this process is Objection (*https://github.com/sensepost/objection/*), which uses Frida to bypass pinning for Android or iOS. Run the Objection command `android sslpinning disable` to bypass pinning.

For most applications, you can bypass the certificate pinning by using these automated tools. But if the application implements pinning with custom code, you might need to manually bypass it. You could overwrite the packaged certificate with your custom certificate. Alternately, you could change or disable the application's certificate validation code. The process of executing these techniques is complicated and highly dependent on the application that you're targeting, so I won't go into detail. For more information on these methods, you'll have to do some independent research.

Anatomy of an APK

Before you attack Android applications, you must first understand what they are made of. Android applications are distributed and installed in a file format called *Android Package (APK)*. APKs are like ZIP files that contain everything an Android application needs to operate: the application code, the application manifest file, and the application's resources. This section describes the main components of an Android APK.

First, the *AndroidManifest.xml* file contains the application's package name, version, components, access rights, and referenced libraries, as well as other metadata. It's a good starting point for exploring the application. From this file, you can gain insights into the app's components and permissions.

Understanding the components of your target application will provide you with a good overview of how it works. There are four types of app components: Activities (declared in `<activity>` tags), Services (declared in `<service>` tags), BroadcastReceivers (declared in `<receiver>` tags), and ContentProviders (declared in `<provider>` tags).

Activities are application components that interact with the user. The windows of Android applications you see are made up of Activities. *Services* are long-running operations that do not directly interact with the user, such as retrieving or sending data in the background. *BroadcastReceivers* allow an app to respond to broadcast messages from the Android system and other applications. For instance, some applications download large files only when the device is connected to Wi-Fi, so they need a way to be notified when the device connects to a Wi-Fi network. *ContentProviders* provide a way to share data with other applications.

The permissions that the application uses, such as the ability to send text messages and the permissions other apps need to interact with it, are also declared in this *AndroidManifest.xml* file. This will give you a good sense

of what the application can do and how it interacts with other applications on the same device. For more about what you can find in *AndroidManifest.xml*, visit *https://developer.android.com/guide/topics/manifest/manifest-intro/*.

The *classes.dex* file contains the application source code compiled in the DEX file format. You can use the various Android hacking tools introduced later in this chapter to extract and decompile this source code for analysis. For more on conducting source code reviews for vulnerabilities, check out Chapter 22.

The *resources.arsc* file contains the application's precompiled resources, such as strings, colors, and styles. The *res* folder contains the application's resources not compiled into *resources.arsc*. In the *res* folder, the *res/values/strings.xml* file contains literal strings of the application.

The *lib* folder contains compiled code that is platform dependent. Each subdirectory in *lib* contains the specific source code used for a particular mobile architecture. Compiled kernel modules are located here and are often a source of vulnerabilities.

The *assets* folder contains the application's assets, such as video, audio, and document templates. Finally, the *META-INF* folder contains the *MANIFEST.MF* file, which stores metadata about the application. This folder also contains the certificate and signature of the APK.

Tools to Use

Now that you understand the main components of an Android application, you'll need to know how to process the APK file and extract the Android source code. Besides using a web proxy to inspect the traffic to and from your test device, you'll need some tools that are essential to analyzing Android applications. This section doesn't go into the specifics of how to use these tools, but rather when and why to use them. The rest you can easily figure out by using each tool's documentation pages.

Android Debug Bridge

The *Android Debug Bridge (ADB)* is a command line tool that lets your computer communicate with a connected Android device. This means you won't have to email application source code and resource files back and forth between your computer and your phone if you want to read or modify them on the computer. For example, you can use ADB to copy files to and from your device, or to quickly install modified versions of the application you're researching. ADB's documentation is at *https://developer.android.com/studio/command-line/adb/*.

To start using ADB, connect your device to your laptop with a USB cable. Then turn on *debugging mode* on your device. Whenever you want to use ADB on a device connected to your laptop over USB, you must enable USB debugging. This process varies based on the mobile device, but should be similar to choosing **Settings ▸ System ▸ Developer Options ▸ Debugging**. This will enable you to interact with your device from your laptop via ADB. On Android version 4.1 and lower, the developer options

screen is available by default. In versions of Android 4.2 and later, developer options need to be enabled by choosing **Settings ▸ About Phone** and then tapping the **Build number** seven times.

On your mobile device, you should see a window prompting you to allow the connection from your laptop. Make sure that your laptop is connected to the device by running this command in your laptop terminal:

```
adb devices -l
```

Now you can install APKs with this command:

```
adb install PATH_TO_APK
```

You can also download files from your device to your laptop by running the following:

```
adb pull REMOTE_PATH LOCAL_PATH
```

Or copy files on your laptop to your mobile device:

```
adb push LOCAL_PATH REMOTE_PATH
```

Android Studio

Android Studio is software used for developing Android applications, and you can use it to modify an existing application's source code. It also includes an *emulator* that lets you run applications in a virtual environment if you don't have a physical Android device. You can download and read about Android Studio at *https://developer.android.com/studio/*.

Apktool

Apktool, a tool for reverse engineering APK files, is essential for Android hacking and will probably be the tool you use most frequently during your analysis. It converts APKs into readable source code files and reconstructs an APK from these files. The Apktool's documentation is at *https://ibotpeaches.github.io/Apktool/*.

You can use Apktool to get individual files from an APK for source code analysis. For example, this command extracts files from an APK called *example.apk*:

```
$ apktool d example.apk
```

Sometimes you might want to modify an APK's source code and see if that changes the behavior of the app. You can use Apktool to repackage individual source code files after making modifications. This command packages the content of the *example* folder into the file *example.apk*:

```
$ apktool b example -o example.apk
```

Frida

Frida (*https://frida.re/*) is an amazing instrumentation toolkit that lets you inject your script into running processes of the application. You can use it to inspect functions that are called, analyze the app's network connections, and bypass certificate pinning.

Frida uses JavaScript as its language, so you will need to know JavaScript to take full advantage of it. However, you can access plenty of premade scripts shared online.

Mobile Security Framework

I also highly recommend the *Mobile Security Framework* (*https://github.com/MobSF/Mobile-Security-Framework-MobSF/*), or the *MobSF*, for all things mobile app testing. This automated mobile application testing framework for Android, iOS, and Windows can do both static and dynamic testing. It automates many of the techniques that I talk about in this chapter and is a good tool to add to your toolkit once you understand the basics of Android hacking.

Hunting for Vulnerabilities

Now that your mobile hacking environment is set up, it's time to start hunting for vulnerabilities in the mobile app. Luckily, hacking mobile applications is not that different from hacking web applications.

To start, extract the application's package contents and review the code for vulnerabilities. Compare authentication and authorization mechanisms for the mobile and web apps of the same organization. Developers may trust data coming from the mobile app, and this could lead to IDORs or broken authentication if you use a mobile endpoint. Mobile apps also tend to have issues with session management, such as reusing session tokens, using longer sessions, or using session cookies that don't expire. These issues can be chained with XSS to acquire session cookies that allow attackers to take over accounts even after users log out or change their passwords. Some applications use custom implementations for encryption or hashing. Look for insecure algorithms, weak implementations of known algorithms, and hardcoded encryption keys. After reviewing the application's source code for potential vulnerabilities, you can validate your findings by testing dynamically on an emulator or a real device.

Mobile applications are an excellent place to search for additional web vulnerabilities not present in their web application equivalent. You can hunt for these with the same methodology you used to find web vulnerabilities: using Burp Suite to intercept the traffic coming out of the mobile app during sensitive actions. Mobile apps often make use of unique endpoints that may not be as well tested as web endpoints because fewer hackers hunt on mobile apps. You can find them by looking for endpoints that you haven't seen in the organization's web applications.

I recommend testing an organization's web applications first, before you dive into its mobile applications, since a mobile application is often a simplified version of its web counterpart. Search for IDORs, SQL injections, XSS, and other common web vulnerabilities by using the skills you've already learned. You can also look for common web vulnerabilities by analyzing the source code of the mobile application.

In addition to the vulnerabilities that you look for in web applications, search for some mobile-specific vulnerabilities. *AndroidManifest.xml* contains basic information about the application and its functionalities. This file is a good starting point for your analysis. After you've unpacked the APK file, read it to gain a basic understanding of the application, including its components and the permissions it uses. Then you can dive into other files to look for other mobile-specific vulnerabilities.

The source code of mobile applications often contains hardcoded secrets or API keys that the application needs to access web services. The *res/values/strings.xml* file stores the strings in the application. It's a good place to look for hardcoded secrets, keys, endpoints, and other types of info leaks. You can also search for secrets in other files by using `grep` to search for the keywords mentioned in Chapter 22.

If you find files with the *.db* or *.sqlite* extensions, these are database files. Look inside these files to see what information gets shipped along with the application. These are also an easy source of potential secrets and sensitive information leaks. Look for things like session data, financial information, and sensitive information belonging to the user or organization.

Ultimately, looking for mobile vulnerabilities is not that different from hacking web applications. Closely examine the interactions between the client and the server, and dive into the source code. Keep in mind the special classes of vulnerabilities, like hardcoded secrets and the storage of sensitive data in database files, that tend to manifest in mobile apps more than in web applications.