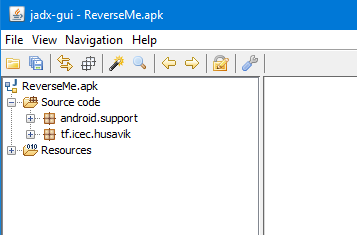
**Intro To Reverse - AndroidRE Solution**

**Introduction:** AndroidRE is designed to introduce students to the basics of reverse engineering a very simple Android application. This challenge centers mostly around basic decompilation of .apk files using a tool known as jadx followed by reading the decompiled Java code.

**Task:** Decomile the provided apk using the provided tool, and find the flag hidden in the code.

**Solving:** Download both the provided tools and the ReverseMe.apk file provided in the challenge package. You’ll notice inside of the tools archive I’ve provided two tools, APKTool, and the aforementioned Jadx tool. You can solve this problem using either tool, however I’ll be using Jadx due to the simple GUI interface provided. However, if you wish to use APKTool decompilation of an .apk file is as easy as executing the following command: ‘apktool -d PATHTOAPK’

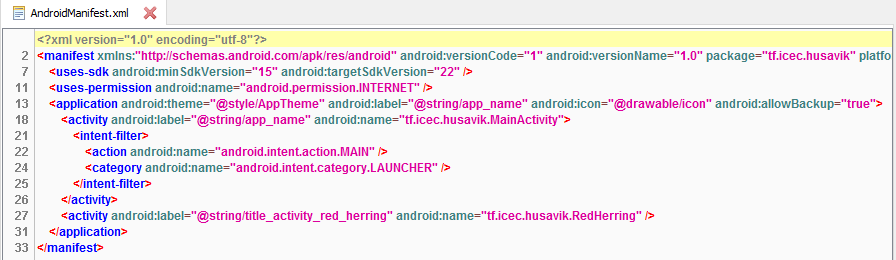
So first things first here, go ahead and launch Jadx - Gui. Decompile the provided .apk by going to file->openfile->reverseme.apk. After you specify the apk to decompile Jadx will list the decompiled packages in the lefthand side of the interface seen below.



You should see two folders named, “Source code” and “Resources”. The first stop we should make when reversing any apk is to take a look at a file called “AndroidManifest.xml”, located under the “Resources” folder. Every Android application is required to have one of these manifest files. The manifest file contains important information required by the Android system to run the application, details such as:

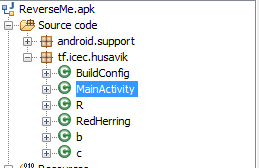
* the java package for the application, which uniquely identifies the application
* Descriptions of the components used by the application, the activities, services, broadcast receivers, and content providers that the application is composed of, as well as the permissions the application needs access to.

The complete details of the information contained inside of the manifest file can be found on Google’s developer page [here](https://developer.android.com/guide/topics/manifest/manifest-intro.html). A screenshot of this application's manifest file is below:

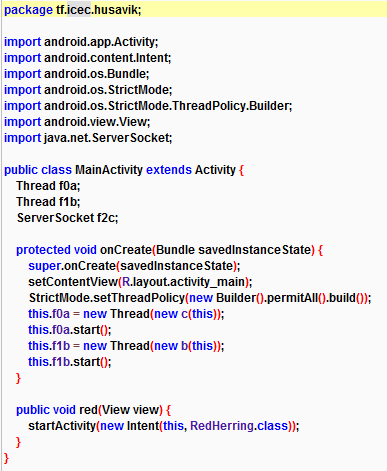


By examining the manifest file we can learn several key pieces of information. This application’s main package is “tf.icec.husavik”, this application requires internet privileges, and finally this application's main activity while running is contained in “tf.icec.husavik.MainActivity”.

Let’s shift gears now and see if we can’t find any interesting information contained in the main package “tf.icec.husavik”. This package is contained under the “Source code” tab go ahead and open it up to list the contents of the package.

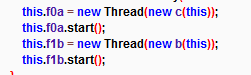


We can see from the contents of the main package that this is a relatively simple program. Think of the MainActivity class as you would a main() function in a c++ program. While every other class file could be seen similarly to their c++ counterpart if that makes sense. Our next step should be to take a look at the code executed in MainActivity for clues.



Above is a screenshot of the MainActivity class. There’s a few things to be learned by examining this code, I’ll list them out from top to bottom. First, notice that two Thread variables are created, as well as a ServerSocket variable. The declaration of those variables could indicate that at some point this application could execute additional behavior in a secondary thread while the main application continues to run. Additionally the ServerSocket variable could be used for some form of internet communication.

After the variable declarations the first few lines of code simply draw the applications interface to the screen. This snippet of code below is interesting though.



When these lines of code are executed by the application new threads will be created and the activity defined in classes ‘c’ and ‘b’ will be executed. Let’s go take a look at those classes and see if we can’t learn what’s going on.

Let’s analyse the behavior of class ‘c’ the code is below.



The only real functionality class ‘c’ provides is opening a socket on port 6464 and accepting incoming connections while the thread remains active. Let’s take a look at class ‘b’ now and hope we find something worthwhile, code snippet below.



At first glance it seems class ‘b’ provides similar functionality to that of class ‘c’. Class ‘b’ opens a socket on port 6464 using the localhost ip address 127.0.0.1. After the socket is created the application uses an objectOutputStream to write a base64 encoded string object to the socket. I wonder what the application’s trying to write. Let’s use an online base64 decoder to find out.

Decoding the base64 string reveals: flag\_wait\_wasnt\_it\_dalvik

Congratulations, you’ve officially reversed your first Android binary. I strongly suggest you poke around the other files in the decompiled binary to learn a little more.