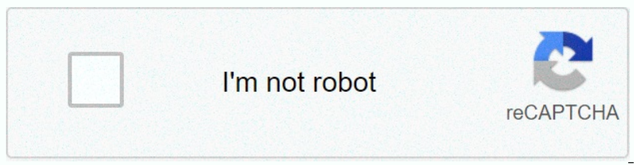

Continue

How to modify apk file in android studio

Today I would like to share with you my findings on how the .apk file can be modified. The .apk is a mobile app because it is installed on a mobile device such as a smartphone, tablet, wearable, etc. Such a file .apk is a simple archive that can be opened with any packager, such as WinRAR So you can easily open and view files - although viewing most files will not make you happy because you realize they are compiled, in binary format, etc... But this is a different story. By the way, you can open the archive and then edit any resource file and save the modification in the archive. However, if you then try to install .apk on your smartphone (or tablet or the like), you'll get an error. The following screenshot shows an error when installing the modified sample myApp.apk app on your Android device: This is because after editing the checksum and signature are no longer valid. This means that simply .apk the file cannot be used. However, there is still a valid use case to edit or replace files within an existing .apk. For example: – files that are located in the asset folder — property files containing configuration data — images that can be replaced — style information sources, and so on. My personal case was: I created an Android app using the SAP Netweaver Gateway Productivity Accelerator. I had to deliver the application to my users, .apk file. But there was a requirement that they want to modify the finished application (change the configuration data). So I had to figure out how to do this: edit the application without accessing the source code. Below, I'll share the steps you want. The description is based on the following software and versions: Android Current API 19 Java 7 Windows 7 If you are not familiar with Android but want to be, you may want to check the documents [1] and [2] All prerequisites for understanding this blog are explained there. Note: To perform the commands described below, you must have Java on your windows variable path (see [1] for clarification). Overview There are 3 steps to follow in order to edit an existing .apk file: 1. Do the actual required edits inside the .apk file 2. They shall sign the .apk 3 December 2004. Install .apk on device 1. Change the resource in the file .apk Open the .apk file with WinRAR (if it does not work, rename the file extension .apk to .zip) Change the resource in the archive as needed (packager tools allow you to change the files without having to extract the archive) Once you are done with the changes, you need to take care of the signature files that are included .apk : Inside the archive go to the META-INF folder Remove existing *. RSA and *. SF files The following screenshot shows the contents of meta-inf in .apk: Now the archive can be closed. If you previously changed the file extension, you must now change it back to .apk 2. Sign .apk Android doesn't allow you to install an app (APK) that isn't signed. When developing an application in Eclipse, ADT (Android Developer Tools, an Eclipse extension that supports development for Android) takes care of signing the application with the default certificate, before installing it on the device. This is convenient, but with the following description, everyone is able to sign the application. Signing .apk is done in 2 steps: a) creating a certificate b) sign .apk with the certificate created Both steps are done with commands at the command line a) Generate a certificate If you are working in a Java environment, you have a JDK on your file system. JDK comes with a certificate management tool: keytool. You can find it in the .../bin folder of the JDK installation. Example: On my computer is here: Now you can generate a certificate using the command below. However, before making, check the notes below, in order to customize the parameters keytool.exe -genkey -v <myKeystore>-keystore-alias <myAlias>-sigalg MD5withRSA -keyalg RSA -keysize 2048 -force 1000 Please note that you have to customize some parameters of the above command to your personal needs: keystore Here, you can provide <myKeystore>any name for your keystore. The name you provide here will be the name of the keystore-file that will be created. The file is created in the current directory. (I tried, but probably you can enter the name of the existing keystore file in order to save the new certificate there) alias <myAlias>Here also, you can enter any name for the alias. It is designed for you to recognize. An alias is the human readable name of a certificate that will be created and stored in a key store. validity of 1000 This is the number of days requested. You can enter any number. I think it should be high enough to avoid expiration problems. Note that the parameters sigalg and keyalg are required JDK 7, so there should be no need to add if you are using JDK 6 Example: keytool.exe -genkey -in -keystore mykeystore-alias myAlias-sigalg MD5withRSA -keyalg RSA -keysize 2048 -force 10000 When executing a command, You will get a few requests at the command line, asking for password, username, organization, city, etc. You can enter any data here, you will just have to make sure you remember the password. After you create the command, the current directory (from where you executed the command) displays the generated key store file in the current directory (from where you executed the command) Now you can continue signing the .apk using the newly created certificate. b) Before signing the .apk signed apk you</myAlias> </myKeystore> </myAlias> </myKeystore>make sure that no certificates are available in the .apk. This is described in step 1 above. To sign the archive, we use the jarsigner tool, which is available with JDK, and which can be found in the same place as keytool. The following command is used to sign apks. jarsigner -verbose -sigalg MD5withRSA-digestalg SHA1 -keystore Please note that you will need to customize some parameters of the above command to fit your personal <keystoreName> <appName> <alias>needs: keystore Here you need to enter the name that <keystoreName>you typed in the previous step a) In order to make the command line short, I recommend temporarily copying the key store file to the same location where you are executing the command. Here you need to enter the name of the APK you want to sign In order to be <appName>command line short, I recommend temporarily copying the .apk file to the same location where you execute the command. <alias>Here you must enter the name of the alias that you entered when you created the certificate Note that the sigalg and digestalg parameters are required by JDK 7. so it should not be necessary to add if you are using JDK 6 Example: jarsigner -verbose -sigalg MD5withRSA-digestalg SHA1 -keystore mykeystore myApp.apk myAlias After you have executed the command, you can check the result inside the file .apk: Open the archive, go to the folder .../META-INF and check the CERT files. RSA and CERT. SF were created. 3. Install the APK on your device Now that the file .apk signed, you can install it on your device. BTW: This procedure is also called lateral load. For Android apps, the installation is performed at the command line using the adb command. ADB stands for Android Debug Bridge adb.exe software that connects a PC to an Android device. Allows access to the device, allows you to run operations, transfer files, etc. To install .apk on a device, you must connect the device to your computer via a USB cable, and then perform the following installation of command adb <appName>In order for the command line to be short, you can temporarily copy the APK file to the same location where you are executing the command. Example: adb install myApp.apk the result should be message success at the command line. If not, one of the previous steps may have been unsuccessful. That's all. You can find the app in your smartphone's app folder. This procedure worked for me on WIN7 and JDK 7. There was no need to refresh the application or generate a new checksum or similar. Links You will find in the following documents for a lot of information for beginners. They also contain many other links for further reading. [1] Getting Started with GWPA: Essential [2] Getting Started with GWPA: Android Preparation:</appName> </alias> </appName> </keystoreName> </alias> </appName> The official docu can be found here: Android Studio 3.0 and above allows you to profile and debug APK without having to build from the Android Studio project. However, you must make sure that you are using an APK with debugging enabled. To start debugging your APK, click Profile or Debug APK on the Android Studio Welcome screen. If you already have a project open, on the menu bar, click > or Debug APK. In the next dialog box, select the APK you want to import into Android Studio, and then click OK. The Android app then displays the unzipped APKs, similar to Figure 1. This is not a fully decompiled set of files, even if it provides .smali files for a more readable version of .dex files. Figure 1. Import a preinstalled APK interface into Android Studio. Note: When you import an APK into Android Studio, the IDE creates a new project in your home directory under ApkProjects/, and makes a local copy of the target APK there. Viewing Android in the Project pane lets you review the following APK: APK: Double-clicking an APK opens the APK. Manifests: Contains application manifests that are extracted from the APK. java: Contains Kotlin/Java code that Android Studio disassembles (into .smali files) from your APK DEX files. Each .smali file in this directory corresponds to the Kotlin/Java class.cpp: If your application contains native code, this directory contains native APK libraries (.so files). External Libraries: Includes an Android SDK. You can use your Android profiler instantly to test your app's performance. To debug your application's kotlin/Java code, you must connect Kotlin/Java resources and add breakpoints to .kt/.java. Similarly, you must attach native debug symbols to debug native code. Connect Kotlin /Java Resources By default, Android Studio extracts Kotlin/Java code from your APK and saves it as .smali files. To debug the kotlin/Java code by using breakpoints, you must point the IDE to the .kt or .java source files that match the .smali files that you want to debug. To connect Kotlin/Java resources, follow these steps: Double-click the .smali file from the Project pane (use android view). After opening the file, the editor displays a banner asking you to select Kotlin / Java resources: Click Connect Kotlin / Java Resources from the banner at the top of the editor window. Go to the directory using the source files of the Kotlin/Java application and click Open. In the Project window, the IDE replaces the .smali files with their corresponding .kt or .java files. The IDE also includes internal classes automatically. Now you can add breakpoints and debug the app as usual. Attach native debug symbols If the APK contains native libraries (súbory.so) that debugging symbols, the IDE function shows you the similar to that shown in Figure 1. You cannot debug native APKs or use breakpoints without attaching tunavible native libraries. If you create native libraries in an APK using a report ID, Android Studio checks whether the report ID in the symbol files matches the report ID in the native libraries, and rejects the symbol files if there is a mismatch. If you do not build with build IDs, then providing incorrect symbol files can cause debugging problems. To connect a debuggable native library, follow these steps: If you haven't already, make sure you download the NDK and tools. In the cpp directory in the Project window (visible only if you selected Android view as shown in Figure 2), double-click the library's native file that does not contain debug symbols. The editor displays a table of all APIs that your APK supports. Click Add in the upper-right corner of the editor window. Go to the directory that contains the tunable native libraries that you want to connect to, and then click OK. If the APKs and tunable native libraries were created using a different workstation, you must also specify paths to the local debug symbols by following these steps: Add local paths to the missing debug symbols by editing the field in the Local Paths column under Path assignments in the editor window, as shown in Figure 2. In most cases, you just need to provide a path to the root folder, and Android Studio automatically checks subdirectories to assign additional resources. Ide also automatically maps paths to remote NDK to local NDK downloads. Click Apply Changes under Path Assignments in the editor window. Figure 2. Provide paths to local debug symbols. You should now see the native source files in the project window. Open these native files and add breakpoints and debug the app as usual. You can also remove mappings by clicking Clear under Path assignments in the editor window. Known issue: When you connect debug symbols to APKs, APKs, and debug.so files must be built by using the same workstation or build server. In Android Studio 3.6 and above, you no longer need to create a new project when the APK is updated outside the IDE. Android Studio detects changes to your APK and gives you the option to re-import it. Figure 3. ApIs updated outside of Android Studio can be re-imported. re-import.