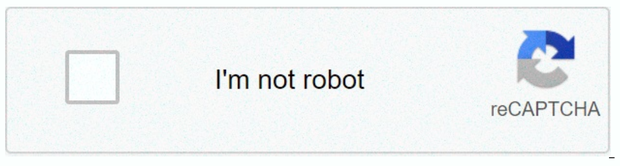

Continue

Edit apk file android studio

Today I want to share my findings with you how the existing .apk file can be modified. A .apk file represents the mobile application as it is installed on mobile devices, such as smartphones, tablets, wearables, etc. Such .apk file is a simple collection that can be opened with any packager such as WinRAR so you can easily open it and view the files – although looking at most files will not make you happy, because you will realize that they are compiled in binary format, etc... But it's a different story. Anyway, you can open the archive and then modify any resource file and save modifications to the collection. But if you try to install .apk on a smartphone (or tablet or similar) later, you'll get an error. The modified sample application on Android device displays the following screenshot error when installing myApp.apk: The reason is that after modification, the checksum and signature are no longer valid. Thus, it is not possible to simply change a .apk file. However, there is still a valid use case for modifying or changing files inside existing .apk. For example: - Files that are placed in the Assets folder - Asset files with configuration data - Images that can be changed - information resources and similar styles. My personal use was the case: I created an Android application using SAP Netver Gateway Productivity Accelerator. I had to apply my users as .apk file. But there's the requirement that they wanted to modify the ready application (change configuration data). So I had to figure out how to get it: modify the app without access to the source code. Below, I'm sharing the necessary steps with you. The details are 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 requirements to understand this blog are explained there. Note: To execute the commands described below, you must have Java on the path variable of your Windows system (see [1] for clarification). Overview To modify the existing .apk file need to follow 3 steps: 1. Make the actual desired modification inside .apk file 2. sign .apk 3. Install .apk on Device 1. Change resources in .apk Open .apk file with WinRAR (if it doesn't work, rename file extensions .apk .zip) Change the resource as desired in the collection (packager tools allow to replace files without the need to extract storage) once you've worked with your changes , you need to take care of the signature files that are part of the .apk: Inside the archive, delete the folder meta-INF existing*. RSA and *. SF files the following screenshot displays the contents of meta-INF In a .apk file: The archive can now be closed. If you changed the file extension earlier, you must now change it back to .apk 2. Sign .apk Android does not allow installing an application (apk) that is not signed. When developing an app in Eclipse, ADT (Extending Eclipse supporting Android developer tools, development for Android) takes care of signing the app with the default certificate before installing on the device. It is comfortable, but with the following details, everyone is able to sign an application. signing .apk is done in 2 steps: a) Create certificate B) Sign .apk with the certificate created Both steps are done with the command on command line A) Generate a certificate If you are working in java environment, you have JDK on your file system. JDK comes with a tool to manage the certificate: Kettool. You can find it in the .../Bin folder of your JDK installation. Example: On my machine it's here: now you can generate certificates using the command below. However, before executing it, check the notes below, to customize the parameter Kettool.exe -genkey-v-keystore-aka-sigalg <myKeystore> <myAlias>MD5withRSA-keyalg RSA-keysize 2048-validity 1000 Please note that you need to adapt some of the parameters of the above command to your individual needs: <mystoreKey>Keystore, you can provide an arbitrary name for your key here the name you will provide will be the keystore-file, which will be created. The file will be created in the current directory. (I haven't tried it, but maybe you can enter the name of an existing keystore file, so that the new certificate can be stored there) aka <myAlias>Here too, you can provide arbitrary names for the alias. It is for you to recognise it. The alias is the human readable name of the certificate that will be created and stored in the keystore. Validity 1000 it is the number of days desired. You can enter any number of your will. I think it should be high enough in order to avoid trouble with termination. Note that parameters by JDK 7 require siglag and keyg, so if you are using jdk 6 example they should not be required to add: Kettool.exe -genkey -v-keystore mystore -aka mystoreAlias-sigalg MD5withRSA -keyalg RSA -ize 2048 -Validity 10000 you will find several signals on the command line, Asking for password, username, organization, city etc, you can enter any arbitrary data here, you only have to be sure to remember the password. After executing the command, you will see the keystore file generated on your file system in the current directory (from which you executed the command) now you can proceed with signing the .apk using the newly created certificate. b) .apk sign the apk before signing the file, your </myKeystore> </myAlias> </myKeystore> </myKeystore> Make sure that no certificates are available in the .apk. This is described in step 1 above. To sign a collection, we use the Jarsigner tool, which is provided with JDK, and which can be found in a place such as Kettool. The following command is used to sign apk. Jarcyner -Verbose -Siglig MD5WRSA-Digestlg SHA1 -keystore <keystoreName> <appName><alias>Please note that you need to customize some of the parameters of the above command to meet your individual needs: keystore <keystoreName>Here you need to enter the name you gave in the previous step) To keep the command line short, I recommend temporarily copying the keystore file to the same location where you are executing the command. <appName>Here you need to enter the apk file name that you want to sign to keep the command line short, I recommend temporarily copying the file .apk the same location where you are executing the command. <alias>Here you must enter the name of the alias you provided when creating the certificate Note that jdk 7 requires parameter siglag and digestology, So if you are using jdk 6 example they should not be required to add: Jarsigner-Verbose-Siglig MD5Vidrsa-Digestig Shaa1-Keystore Mykistore MyApp.apk After executing the command, you can check the result inside the .apk file: open the archive, go to guna.../META-INF and check if RSA and CERT. SF has been created. 3. Install apk on device Now when .apk file has been signed, you can install it on your device. BTW: This process is also called side-load. Installation is done on the command line using adb command for Android applications. ADB Android Debug Bridge stands for ADB.exe is a piece of software that connects the PC to an Android device. It allows access to the device, triggering operations, transferring files, etc. To install .apk on the device, you need to connect the device to your PC via usb cable, then execute the following command ADB install <appName >to keep the command line small, you can temporarily copy the APK file to the same location where you are executing the command. Example: ADB installs myApp.apk the result must be a success message at the command prompt. If not, any of the previous steps may fail. Just. You can find the application in your smartphone's apps folder. This process worked for me on WIN7 and JDK 7. It was not necessary to rebuild the app, nor to generate new checksum or similar. Link Please refer to the following documents for a lot of information for beginners. They also contain lots of additional links for further reading. [1] Getting started with GWPA: [2] GWPA के साथ शुरू हो रही है: एंड्रॉयड तैयारी:</appName> </alias> </appName> </keystoreName> </alias> </appName> </keystoreName> </keystoreName> The official Doku can be found here: Android Studio 3.0 and higher allows you to profile and debug apks without creating them from the Android Studio project. However, you need to make sure you're using an APK that has debugging enabled. To start debugging APK, click on profile or debug APK from Android Studio Welcome screen. Or, if you already have a project open, click file > profile or debug APK from the menu bar. In the next dialog window, select the APK that you want to import into Android Studio and click OK. Android Studio then displays unpacked APK files, which is similar to figure 1. Although it is not a completely decomposed set of files, it provides .smali files for a more readable version of .dex files. Figure 1. Importing pre-made APK into Android Studio. Note: When you import APK into Android Studio, IDE creates a new project in your home directory under ApkProjects/, and there creates a local copy of the target APK. The Android view in the Project Pane allows you to inspect the following content of your APK: APK File: Double-clicking on APK opens the APK analyzer. Appears: Apps extracted from APK appear. Java: Kotlin/Java contains code that Android Studio (in .smali files) disassembled from your APK DEX files. Each .smali file in this directory corresponds to a Kotlin/Java class.cpp: If your app includes native code, this directory includes your APK's original library (.so files). External library: This includes the Android SDK. You can use Android Profiler to start testing your app's performance instantly. To debug your app's Kotlin/Java code, you must attach Kotlin/Java sources and add breakpoints to .kt/.java files. Similarly, to debug your original code, you need to attach native debug symbols. Attach Kotlin/Java formulas by default, Android Studio extracts Kotlin/Java code from your APK and saves it as .smali files. To debug the Kotlin/Java code using breakpoint, you need to point the IDE to .kt or .java source files that correspond to the .smali files you want to debug. To attach Kotlin/Java formulas, proceed as follows: Double click on the .smali file from the project pane (use Android View). After you open the file, the editor displays a banner in which you are asked to select Kotlin/Java sources: Click kotlin/Java sources attached to the banner at the top of the editor window. Navigate to the directory with the app's Kotlin/Java source files and click Open. In the Project window, the IDE replaces the .smali files with their corresponding .kt or .java files Internal classes in the IDE are also included automatically. Now you can add breakpoints and debug your app as you normally do. Attach native debug symbols If your APK includes native libraries (.so files) that don't include debug symbols, the IDE shows you a Similar to the one shown in figure 1. You can't debug apk's original code or use breakpoints without attaching debugable native libraries. If you build the original libraries in your APK with the ID, Android Studio checks if the build ID in your symbol files matches the build ID in your original libraries and rejects the symbol files if there is a mismatch. If you have not constructed with the build ID, providing incorrect symbol files may cause debugging problems. To attach debugable native libraries, proceed as follows: If you haven't already done so, be sure to download NDK and tools. Under the CPP directory in the project window (only visible if you selected android view, as shown in Figure 2), double-click a native library file that does not contain debug symbols. The editor shows a table of all AVIS supporting your APK. Click Add in the top right corner of the editor window. Navigate to the directory that includes the debugable native libraries you want to attach and click OK. If APK and debug-worthy native libraries were created using a separate workstation, you also need to specify the path for local debug symbols by following these steps: Add the local path to the missing debug symbols by editing the area under the local path column in the Path Mapping section of the editor window, as shown in Figure 2. In most cases, you only need to provide the route to the root folder, and Android Studio automatically inspects subdirectors to map additional sources. IDE automatically maps the paths of a remote NDK to download your local NDK. Click Apply changes in the Path Mapping section of the Editor window. Figure 2. Providing the path to local debug symbols. You should now view the native source files in the Project window. Open those native files to add breakpoints and debug your app as you normally would. You can also remove the mapping by clicking Clear in the Path Mapping section of the editor window. Known issue: When attaching debug symbols to APK, both APK and debugbal .so files must be created using the same workstation or build a server. In Android Studio 3.6 and above, you no longer need to create a new project when APK is updated outside the IDE. Android Studio detects changes to APK and gives you the option to import it again. Figure 3. Updated APK outside Android Studio can be re-imported. Re-imported.