uc3m | Universidad Carlos III de Madrid

Grupo de investigación:
Computer Security Lab

Table of Content

1.Introduction

Dalvik Virtual machine (DVM) is Register Architecture, designed to run on low memory, uses its own byte code and runs .dex file (Dalvik Executable File)

In fact we could say that DVM is a runtime environment. To put it in simplest terms, runtime comprises of software instructions that execute when your program is running, even if essentially they're not part of the code of that piece of software in particular. These instructions basically translate the software's own code into the code that the computer is capable of running. Therefore, all computer languages require some sort of runtime environment that can properly execute the code written in that language.

Android makes use of a virtual machine as its runtime environment in order to run the APK files that constitute an Android application. The advantage of using a virtual machine is twofold – first, the app code is isolated from the core operating system, ensuring that should something go wrong, it's contained in an isolated environment and does not affect the primary OS. Second, it allows for cross-compatibility, meaning even if an app is compiled on another platform (such as a PC, as is usually the case with developing mobile apps), they can still be executed on the mobile platform using the virtual machine.

For Android, the virtual machine-based runtime environment in use so far is known as the Dalvik Virtual Machine. Virtual machines supply an isolated environment for code execution. Hence, even if an app contains a malicious piece of code that can damage the core OS, it won't directly affect the system files, and thus the core OS will be kept from getting corrupted. The large scale benefit is more stability and reliability for the operating system.

The app APKs that are supplied through Play Store (or any other source, for that matter) are uncompiled instructions that developers rely on the virtual machine to compile before execution and run on the device. This provides more compatibility; if the developer were to provide already compiled code and it was compiled for a Snapdragon-based processor, it might not run correctly on a Tegra chip, for instance. Hence, this on-the-device compilation addresses this issue.

Hackers use a tool called a decompiler to convert VM code back into original source. Many Java decompilers are available, and because of Android's relationship to Java, any Android code compiled from Java code is open to decompilation.

When you build an Android app in Eclipse or Android Studio, it is first compiled in Java. Then a tool called "dx," which comes with the Android SDK, converts the Java jar file into a classes.dex file (see the Android build process in Figure 1.1). Decompiling an Android APK is a two-stage process: The .dex file is first converted back into .class files using a tool called dex2jar where it can then be decompiled using your favorite Java decompiler, such as JD-GUI.

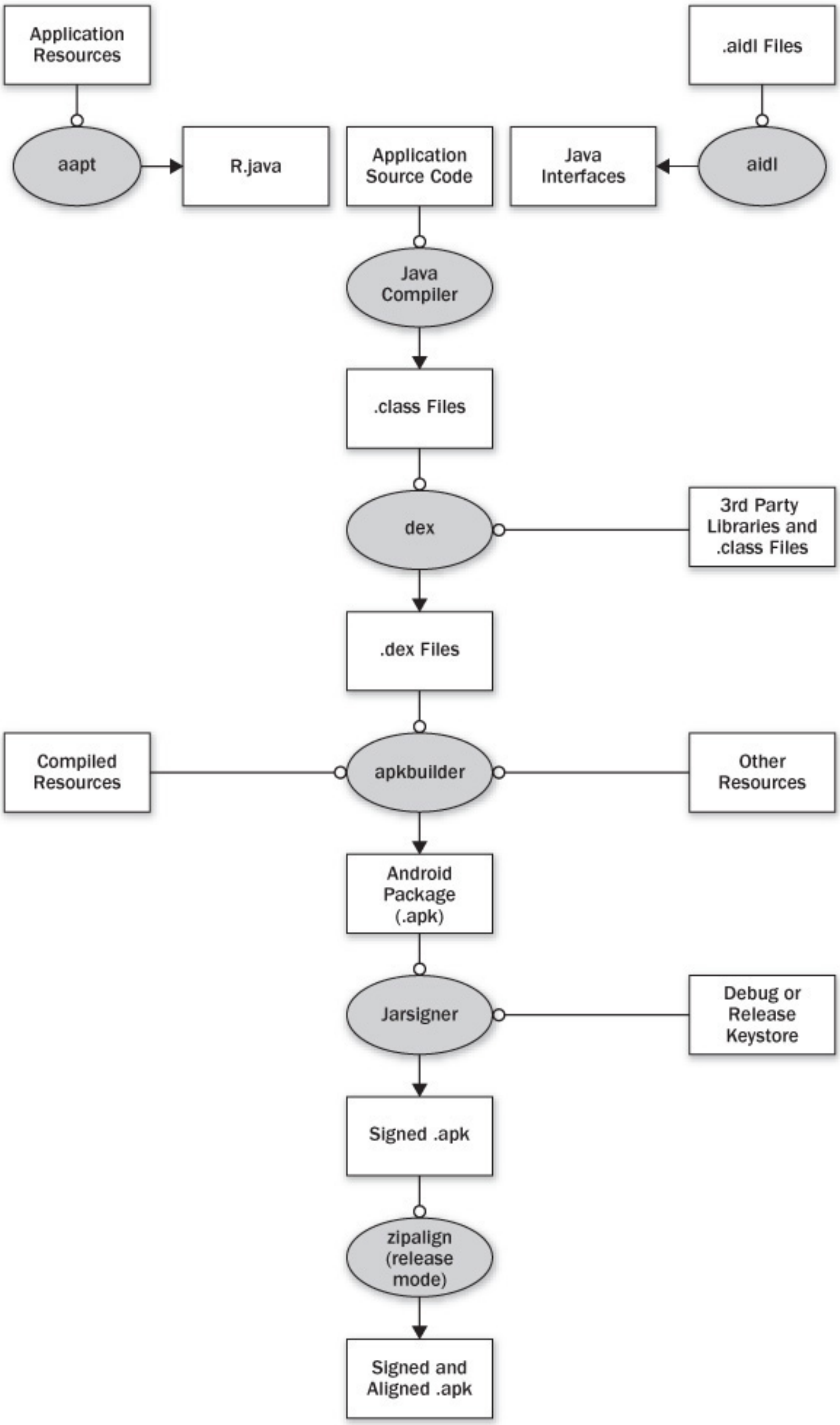

Figure 1.1. Android Build Process

2.Decompiling an APK

As demonstrated in Module I, we could use the ADB (Android debug bridge) tool that comes with the Android Developer Kit as part of the Android SDK to decompile an apk. The adb command allows you to pull a copy of the APK off the phone onto your PC for further analysis.

In this module, we will be using the canal_P_black.apk provided via AG. Once the APK is downloaded, connect your phone to your PC using your USB phone cable and then turn on the USB debugging under the developer options on your Android phone. Next, you need to know the name of the APK that you want. You don't need to root the phone if you're Android OS is anything below Android 4.3 to pull an APK off the phone because the naming convention and location follows the same basic rules. You may also use the emulator that you have been using before in Module I.
The APK is in the /data/app folder in most phones otherwise you will have to find out where it is. Now that we've found the APK, we can pull it onto the PC on most Android phones by typing the command:

adb pull /data/app/**canal_P_black.apk**

2.1 Dex2jar

Dex2jar is a tool for converting Android .dex format to Java's .class format. In other words it is just one binary format to another binary format, but not to the Java source. You still have to run a Java decompiler on the resulting jar file to view the source.

Dex2jar is available on Google code and was written by Pan Xiaobo, a graduate student at Zhejiang University in China. It is provided in AG (*"dex-tools-2.1-20171001-lanchon.zip"*).

If we unzip the **canal_P_black.apk** file (see Figure 1.2), we can see some files and folders familiar to any Android developer, such as assets and resources folders, and the AndroidManifest file.

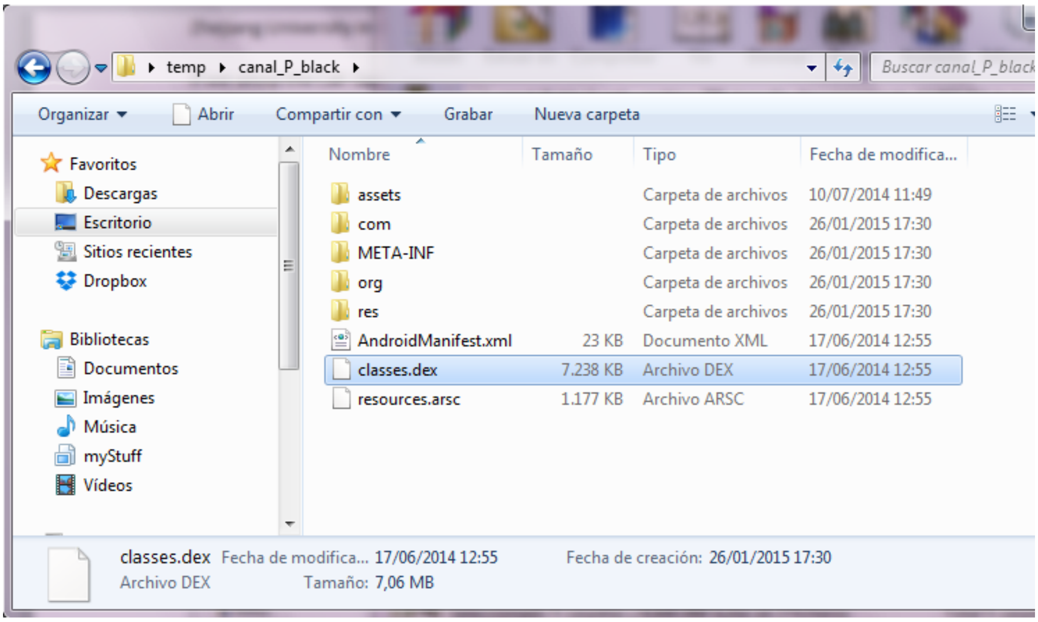

Figure 2.1. canal_P_black.apk folder

The classes.dex file contains all the programming instructions needed to run your Android app.

The Java files you write in Android Studio are first compiled into Java class files and then further compiled into a classes.dex file before being added to the APK.

We will be using Dex2jar, which can revert classes.dex back into a Java jar file. Afterwards, we can decompile them with any decompiler, such as JD-GUI. Executing the dex2jar file against the apk file from the command line will convert it into a Java jar file.

In order to get the Java jar file from our apk we do as follows:

1. Download Dex2jar folder from https://sourceforge.net/projects/dex2jar/ and unzip it in C:\ for an efficient use.
2. Unzip the apk file of our application (this was part of module I).
3. Once we have the apk unzipped, as shown in figure 2.1., copy the classes.dex file into C:\Dex2jar
4. Open a cmd window as administrator, navigate to the folder C:\Dex2jar and run the following command, as shown in the figure 2.2.

d2j-dex2jar classes.dex classes_dex2jar.jar

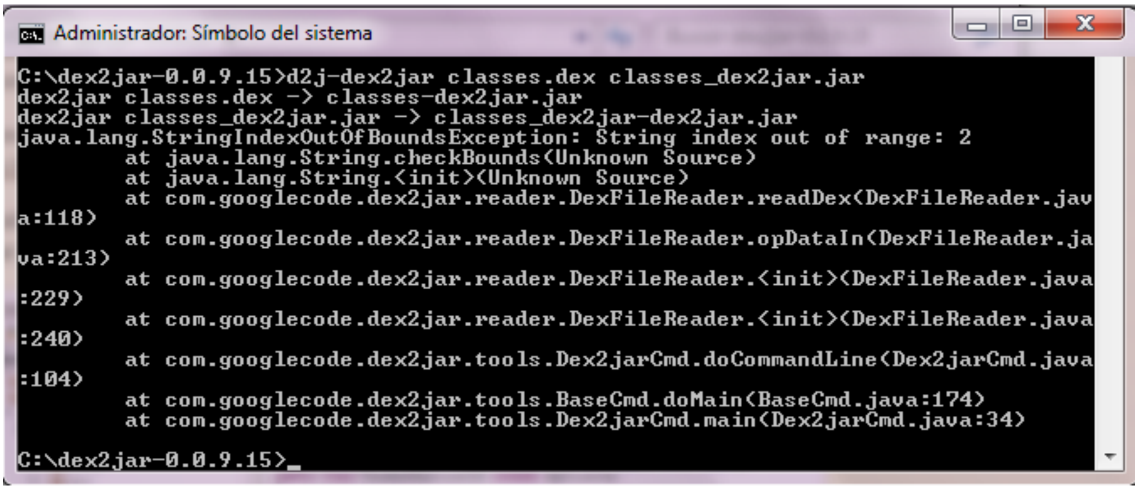

Figure 2.2. Using the Dex2jar tool to create the java jar file

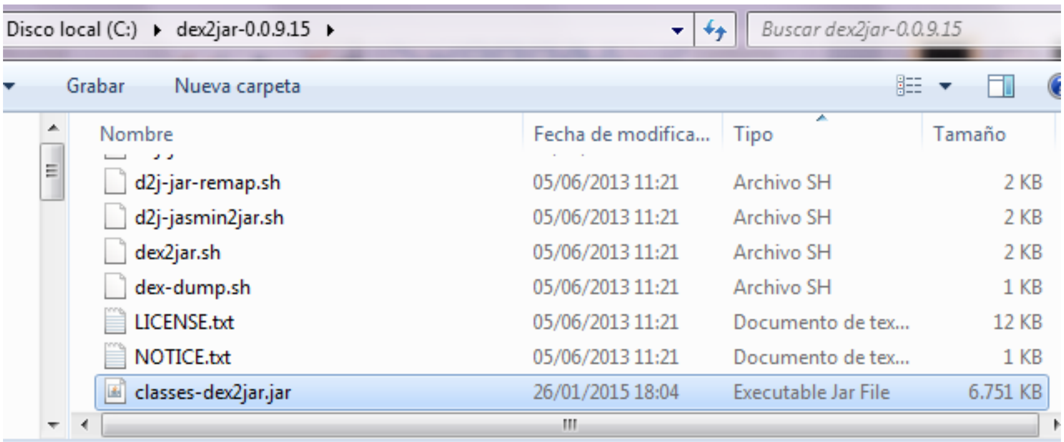

Figure 2.3. classes-dex2jar.jar file

5. The file **classes_dex2jar.jar** is created as it is shown in Figure 2.3
6. In order to decompile the jar file we need the tool JD-GUI available at http://jd.benow.ca/. Once downloaded, we can use it immediately by opening it and then going to: File -> Open File … and locating the classes-dex2jar.jar file created with the Dex2jar tool. we may also drag and drop the aforementioned file.
7. Once we open it, we will have the java files from the application as shown in figure 2.4 and figure 2.5.

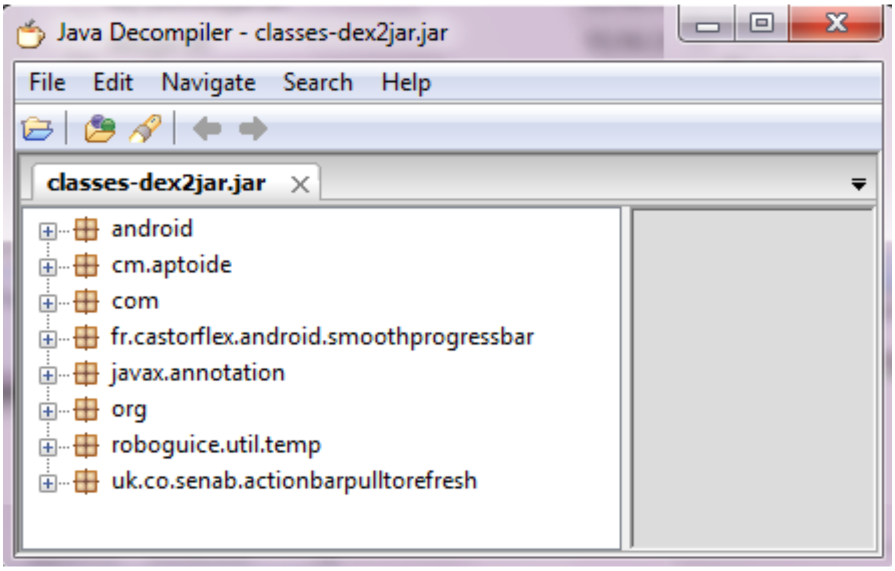

Figure 2.4. Decompiled Java files from the canal_P_black.apk application

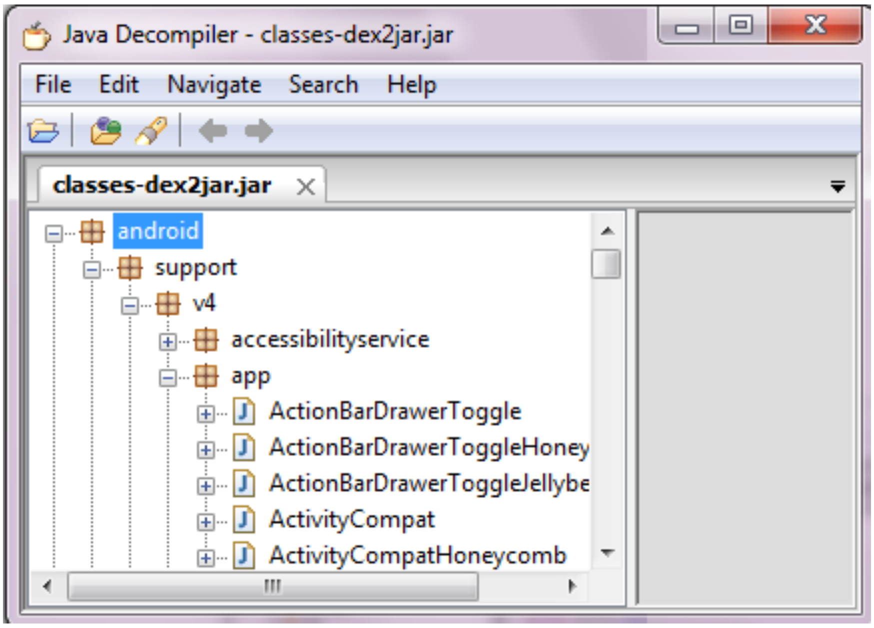

Figure 2.5. Detailed structure from the canal_P_black.apk application Java files

3. Obfuscation

If we want to protect our code we'll need to know how to hide as much information as possible from dex2jar and JD-GUI. In the next section we want to see how we can use obfuscation to make this dex2jar/JD-GUI translation a little more difficult or, at the very least, less complete.

Obfuscators protect against decompilation in a number of ways. They don't stop decompilers or dex2jar from reverse engineering the code, but they do make the decompiled code harder to understand. At the very simplest, they convert all the variables and method names and strings in an APK of one or two character strings. This takes a lot of the meaning out of the Java source and makes it more difficult to find an API key or where you're storing the user's login information for example.

Good obfuscators will also change the flow of the code and, in many cases, hide a lot of the business logic. It won't stop a determined hacker from understanding what you're doing in your code, but it will make it significantly harder.

Just like there are plenty of Java decompilers, there are also plenty of Java obfuscators, such as ProGuard, yGuard, RetroGuard, DashO, Allatori, Jshrink, Smokescreen, JODE, JavaGuard, Zelix Klassmaster, and jCloak, just to name a few. There is even an Android classes.dex obfuscator, called Apkfuscator, which is available at https://github.com/strazzere/APKfuscator, or you can try Shield4J at http://shield4j.com.

We must check at least one to see how it works; so we will be using ProGuard. There is a commercial version of ProGuard as well.

3.1 ProGuard

Before starting to get into the details about ProGuard it is important to mention that before publishing our application to any market, we need to compress the APK file as much is possible. ProGuard is an open source project and it is available at: www.proguard.sourceforge.net and it is not specific for android development, it could work for all java code. But it is so good for android that it is integrated into the Android build system, so you do not have to invoke it manually. Hence, the last URL is not for download but good for documentation.

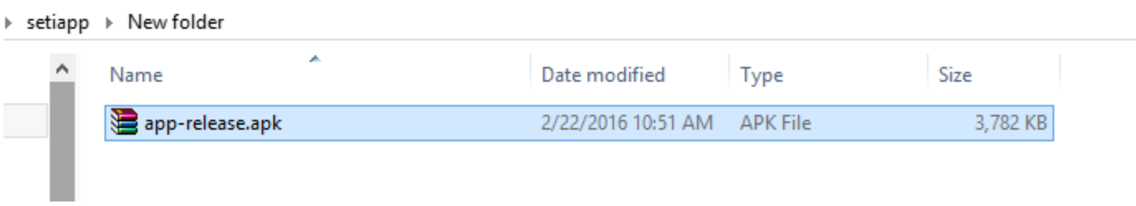

Figure 3.1. Size of SetiApp.apk file with no obfuscation

In order to use ProGuard in Android Studio, all you need to do is just a simple change in the build.gradle (Module:App) in Gradle Scripts by changing the minifyEnable to true instead of false

```
android {
 ...

  buildTypes {
      release {
          minifyEnabled true
          proguardFiles getDefaultProguardFile('proguard-android.txt'),
          'proguard-rules.pro'
      }
  }
}
```

Figure 3.2. Enabling ProGuard in Android Studio

For some situations, the default configurations in the ProGuard configuration file will suffice. However, many situations are hard for ProGuard to analyze correctly and it might remove code that it thinks is not used, but your application actually needs. Some examples include:

- a class that is referenced only in the AndroidManifest.xml file
- a method called from JNI
- dynamically referenced fields and methods

The default ProGuard configuration file tries to cover general cases, but you might encounter exceptions such as ClassNotFoundException, which happens when ProGuard strips away an entire class that your application calls.

You can fix errors when ProGuard strips away your code by adding a -keep line in the ProGuard configuration file titled 'proguard-rules.pro' under Gradle Scripts directory inside of your Android Studio project as shown in Figure 3.3 or by adding “@Keep” before the class declaration.

https://developer.android.com/reference/androidx/annotation/Keep.html?hl=en

```
-keep public class <MyClass>
```

Figure 3.3.a. Line that may be added to Proguard configuration file to stop Proguard from stripping away an entire class call (put complete name of the class, including the package)

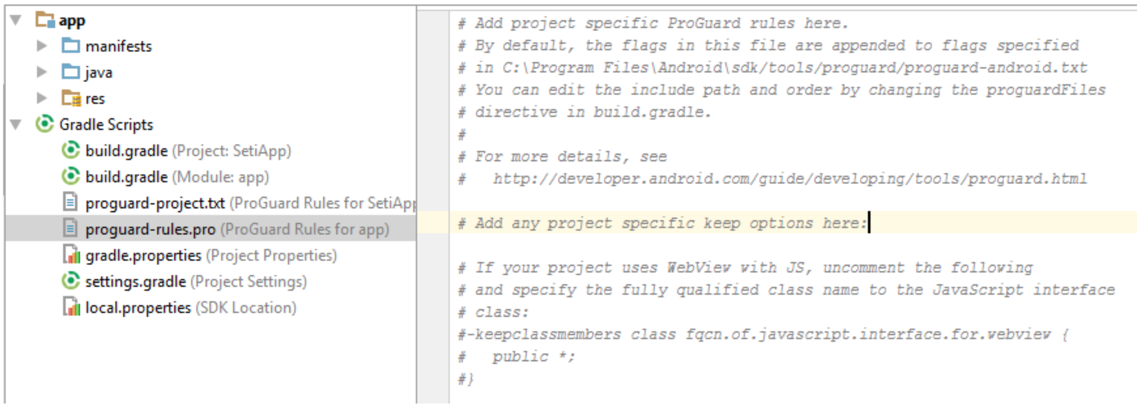

Figure 3.3.b. Location of adding extra rules to the Proguard configuration file

If by adding the keeping option we encounter a warning, it may be necessary to add the following line in the build-gradle file (Figure 3.3.c) and syncronize the project again.

```
implementation "androidx.versionedparcelable:versionedparcelable:1.1.0"
```

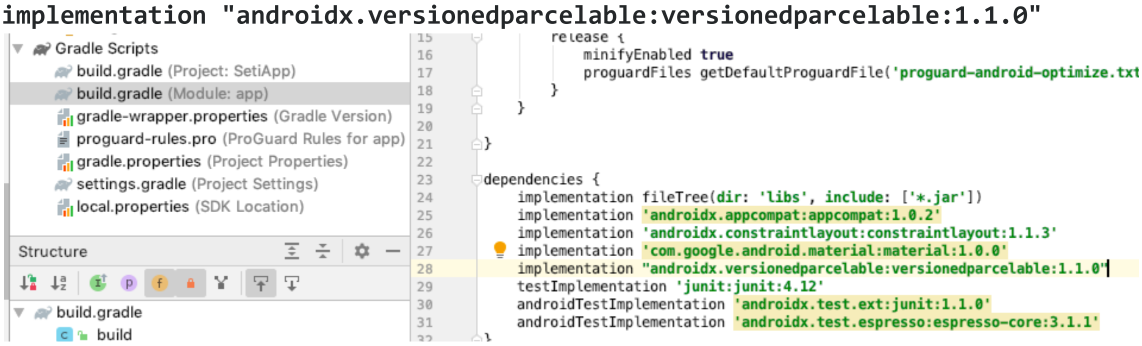

Figura 3.3.c. Line to correct the warning

Obfuscating the code will go through all the methods and variables names and will switch their descriptive name with very short single character names. Then it will remove classes and other references that are not used in the app and by doing this it will shrink down the size of the apk file. This is particularly useful when the app uses external libraries but only small part of them are used in the app, such as support libraries that use only fraction of that library.
Once we have changed our build.gradle file, we **'Rebuild Project'** then we **'Generate Signed APK...'** of our setipp.apk app. Both actions can be found in the Build menu as shown in Figure 3.4. Of course, you will follow the same steps of signing the apk that were detailed in Module I.

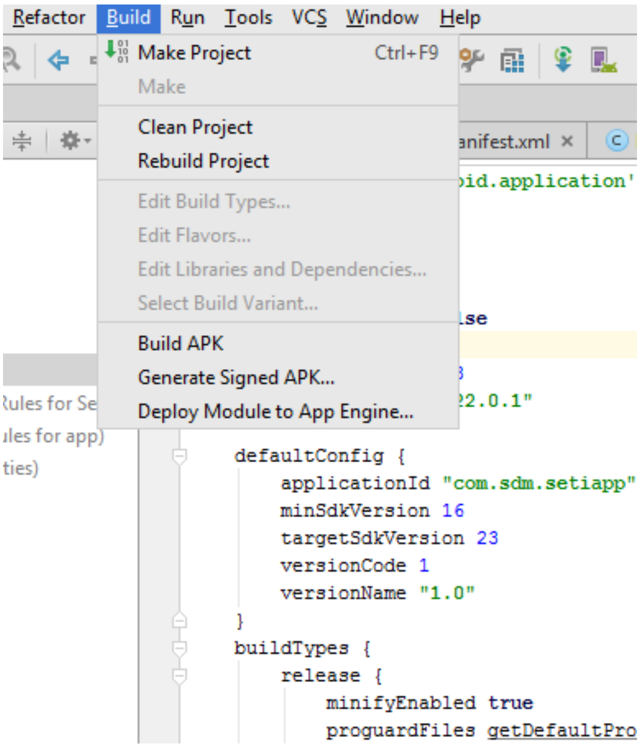

Figure 3.4. Rebuilding Project and Generate Signed APK

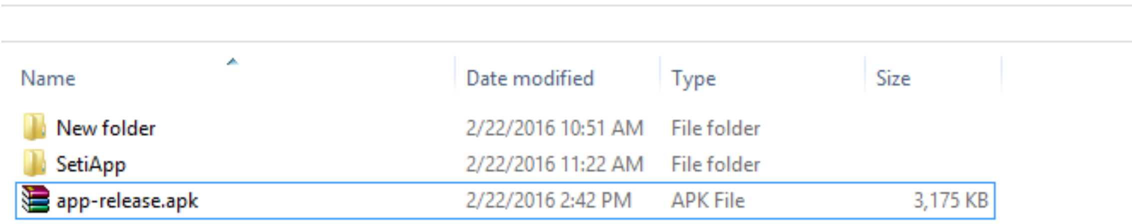

Figure 3.5. Signed SetiApp APK size after using Proguard

1. As we can see from the Figure 3.5 the size has changed. The time of the creation of our setiapp.apk file decreases as well. The size goes down from 3,782 KB to 3,175 KB, (around 16% optimization).

It is very important that we take the APK optimized file, load it into a physical device and test it. This is just to be sure that the app is working properly as it is was designed. This is because in java programming when we don´t have direct reference to classes and methods (different java techniques) ProGuard sometimes can strip out the code that it considers unnecessary as we mentioned previously.

Let´s take a look at how ProGuard would obfuscate our code. For example, just looking at the first few lines of our Inicio class in Figure 4.6, we could see the variable names contador, sumar, etc. We will compare it later with the obfuscated code that ProGuard will provide us for these lines of our original code.

Figure 3.6. A few lines from our original code

Once we regenerate our app as it was shown previously in this section, we copy our setiapp.apk and proceed as in section 2 from this Module:

1. Go to the Dex2jar folder that we located in the C:\ folder for an efficient use.
2. Create the SetiApp folder in C:\ Dex2jar\SetiApp and unzip the new setipp.apk file that you created after enabling ProGuard in it
3. Once you have the files from our regenerated apk, rename the classes.dex to seti_classes.dex so you don´t confuse it with any other classes.dex in the C:\Dex2jar\ folder (Figure 3.7)

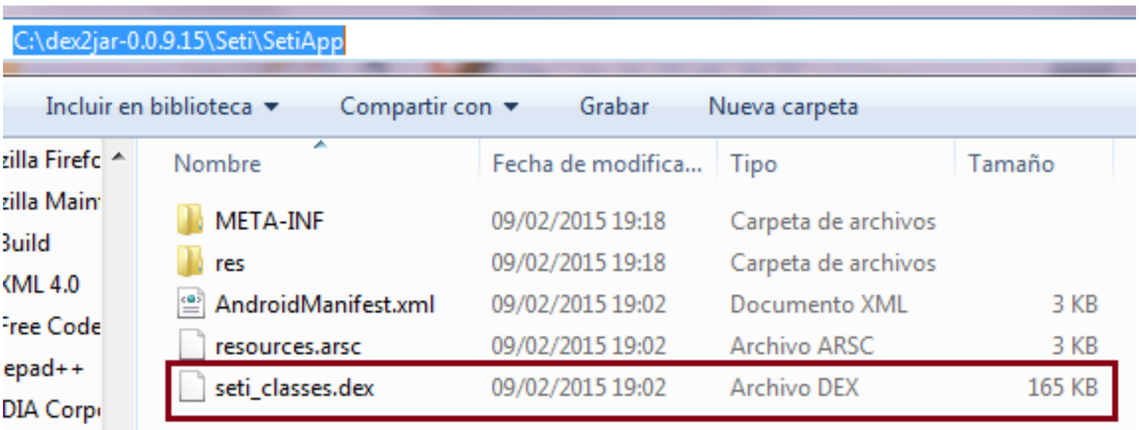

Figure 3.7. Repackage seti_classee.dex file

4. Cut the seti_classes.dex and paste it in the the C:\Dex2jar\ folder
5. Open a cmd window as administrator, and change the directory to file C:\Dex2jar\ and run the following command (as it is shown in the figure 3.8):

d2j-dex2jar seti_classes.dex seti_classes_dex2jar.jar

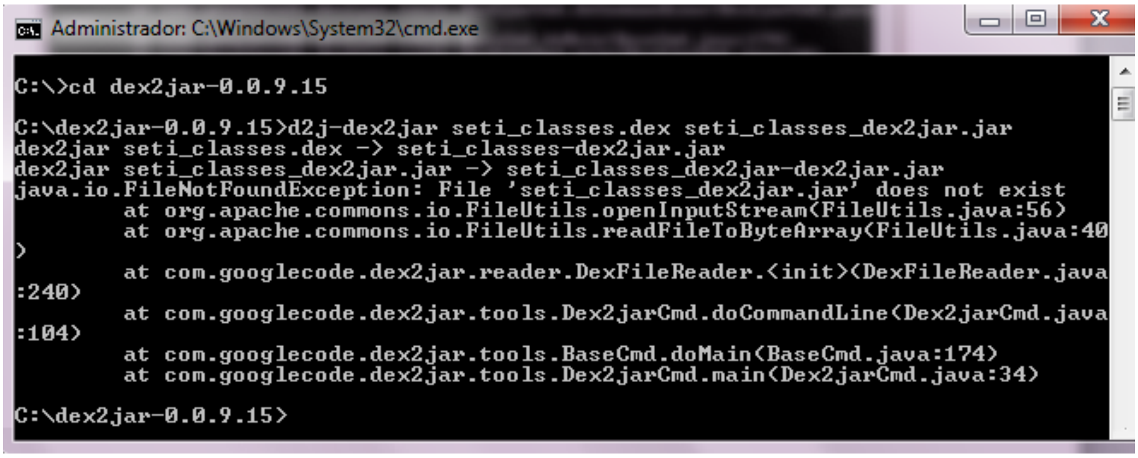

Figure 3
.8. Getting the jar file for the seti_classes.dex file

6. The file **seti_classes_dex2jar.jar** is created as it is shown in the figure 3.9.
7. In order to view the jar file we need the tool JD-GUI that you may download from http://jd.benow.ca/. Once you install it, run the JD-GUI tool and then go to: File Open File … and locate the **seti_classes_dex2jar.jar** file created with the Dex2jar tool

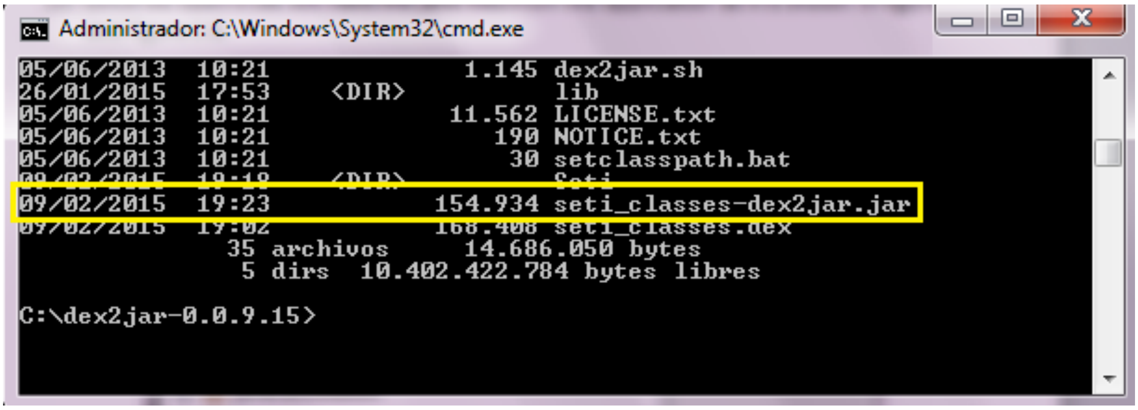

Figure 3.9. Creation of the seti_classes-dex2jar.jar file

8. Once we open it, we will have the java files from the application as it is shown in figure 3.10.

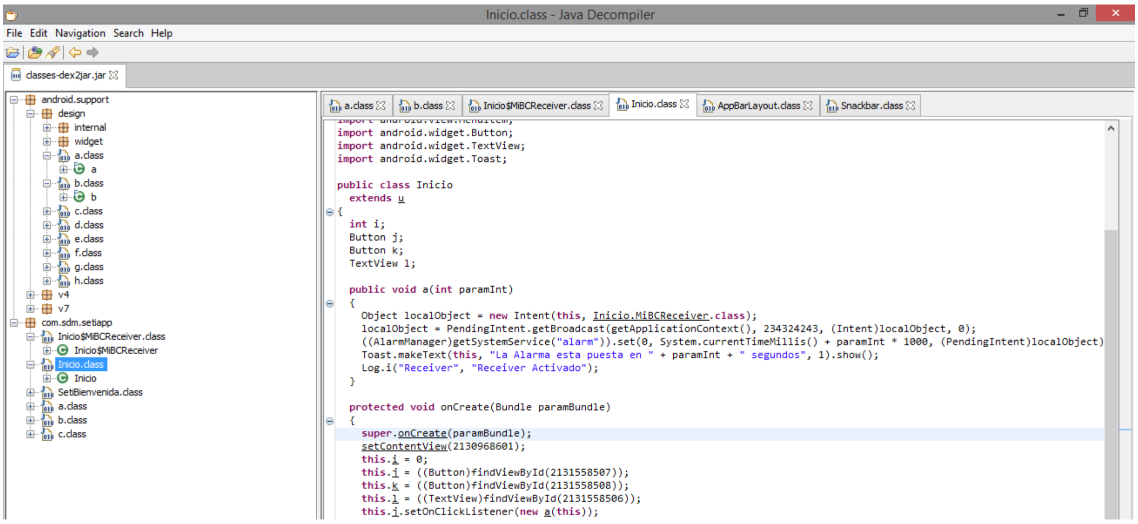

Figure 3.10.a JD-GUI showing obfuscated code from our APK

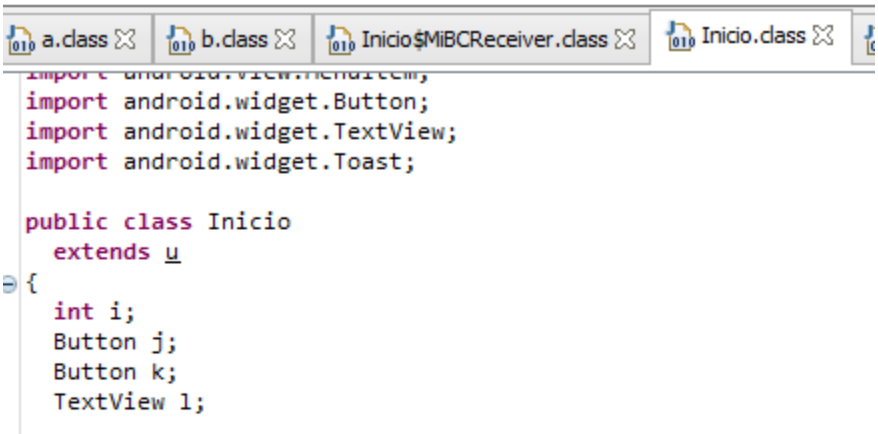

Figure 3.10.b. Variable names from the Inicio class after obfuscation

In order to have a better look of the final obfuscated code from our application and to compare it with the listing given above we give a continuation the new listing of the obfuscated code. As we can see, the variable names have been changed. In the same way the object´s name has been changed for numbers. In general the code look much more compressed as before. All the added comments introduced as “End of the public Class ..” or “End of the method ..” are suppressed. The intention here is to remove any information that would help the attacker understand how your code works.

If you ever need to get back to the original method names, ProGuard creates a mapping.txt file, which enables you to see exactly what method names and variables it transformed. We can see in Figure 4.9 what has exactly been changed. For example, we can see that the Button sumar has its name changed to -- > b, Button restar has changed name to -- > c and so on.

ProGuard generates four files. However, Android Studio only create the mapping.txt file by default in the path *app\build\outputs\mapping\release*. To force it to create the usage.txt and the seed.txt, the proguard-rule.pro file and add the following lines (Figure 3.10.c):

```
-printusage
<Path_to_Projects_folder>/SetiApp/app/build/outputs/mapping/release/usage.txt

-printseeds
<Path_to_Projects_folder>/SetiApp/app/build/outputs/mapping/release/seeds.txt
```

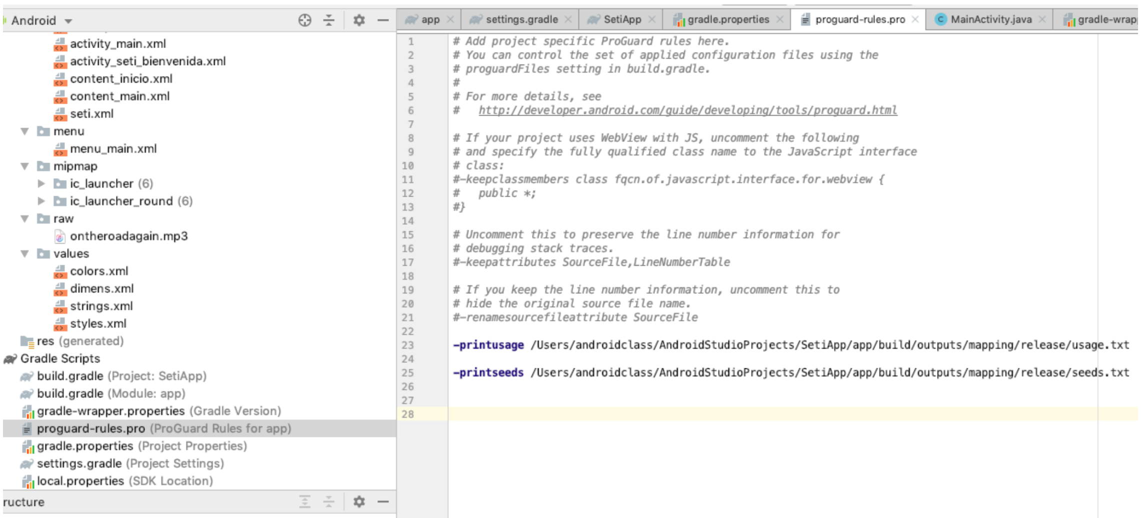

Figura 3.10.c.Líneas a añadir al archivo proguard-rules.pro

ProGuard in general creates four files to help you keep track of your original code. They are dump.txt, mapping.txt, seeds.txt, and usage.txt located in *app\build\outputs\mapping\release*

- dump.txt shows the Java Classfile information before it gets transformed into a classes.dex file (not generated bu Android Studio).
- mapping.txt shows names before and after obfuscation changes.
- seeds.txt lists the classes not obfuscated.
- usage.txt lists the code that was removed from the APK.

You can find the ProGuard SDK in the sdk/tools/proguard directory of the Android SDK. It provides some extra ProGuard tools, such as retrace, which allows you to debug an obfuscated app. You may use the following command, in en *sdk/tools/proguard/lib,* to obtain a stacktrace output as if you never enabled obfuscation.

java -jar retrace.jar mapping.txt appname.trace

https://www.guardsquare.com/es/blog/debugging-applications-obfuscated-dexguard

3.3 Smali

The Icelandic theme comes up again and again in Android. The Dalvik in DVM is the name of an Icelandic fishing village where the original developer of the DVM's ancestors came from. So, continuing with the Icelandic theme, there are also Baksmali and Smali, which are the de facto tools used to disassemble and assemble classes.dex files. Smali means shepherd or assembler in Icelandic, and Baksmali means disassembler. Disassembled files are given the .smali extension, where each Smali file corresponds to the original java file.

Smali files are ASCII representation of the Dalvik opcodes and are fairly easy to read, similar in style and content to what we saw using dexdump earlier in this module. Smali is its own language, and there is no reason why you couldn't code your entire Android app in Smali if you had enough time. See http://source.android.com/devices/tech/dalvik/dalvik-bytecode.html for a good beginner's reference to Smali and the DVM. You may download smali and/or baksmali from https://bitbucket.org/JesusFreke/smali/downloads .

Depending on the developer´s knowledge, sometimes it is possible for recompiling to modify the behaviour of applications. In the next section we would like to demonstrate how an attacker can change an application's behaviour by decompiling the application, changing the smali code, and recompiling it. So our task here is to decompile and recompile Android applications without having access to the source code, thus changing the application's behaviour.

4. Modifying the Behaviour of an Application

Many applications require a registration code or serial key before being used or they can only be used for a specified trial period or show ads when being used. A user of these techniques could edit smali code and bypass these mechanisms making the application exposed to a malicious user.

In this section, we want to describe how a malicious user can reverse engineer and modify the behavior of a particular application. For this practice we will be providing the **"uc3m_banca.apk"** application as material in the Aula Global, for which a malicious user can achieve this. The intention is to describe few ways in which a malicious user can modify an application's behavior to add or remove functionality. We want to demonstrate how application behavior can be modified by decompiling it into smali code and recompiling it back and then packaging it in apk file. Once you are done with this exercise we will ask you to modify your application built in Module I. You will need to add a user and password activity as shown in this section. In order to accomplish such task you will need to change your application that will require a user to enter the correct passcode before using the application. Then to describe how a malicious user can potentially bypass this intended functionality.

Figure 4.1. Banca UC3M Login screen

The first step in analyzing or reverse engineering an application is to understand its behavior. Typically, this entails installing, using and reviewing the application's various functions.

4.1 Decompiling the application

To accomplish such task:

1. Install the application on an emulator and try to use it. Follow the instruction depicted in previous sections and/or Module I in order to install the application on the emulator
2. Once you installed the app, launch the application. It will present the user with a login user and password screen as it is shown in Figure 4.1. At this point, we don’t know the length of the password required or if the password is numeric (PIN) or alphanumeric
3. Learn (by trial and error) that the password field only accepts 4 characters
4. Learn (by trial and error) that the password field only accepts digits. We want to simulate the PIN code login of apps used in Bank applications
5. Decompile the application file (apk) by using apktool (Figure 4.2). Please read the previous sections and Module I in order to decompile the apk. See Figure 4.3 that shows uc3m_banca.apk decompiled into a uc3m_banca folder.
6. Browse through the folder (Figure 5.3), we note that there is a smali folder. Smali files are found in the uc3m_banca directory

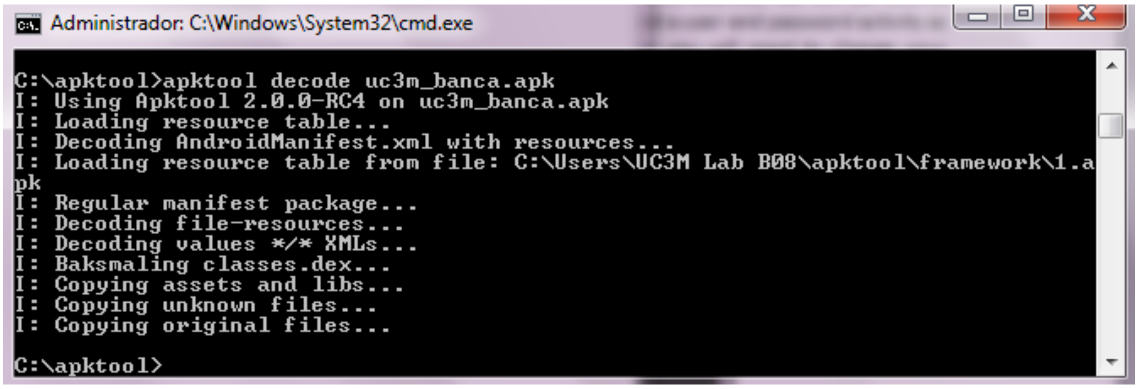

Figure 4.2. uc3m_banca.apk decompilation

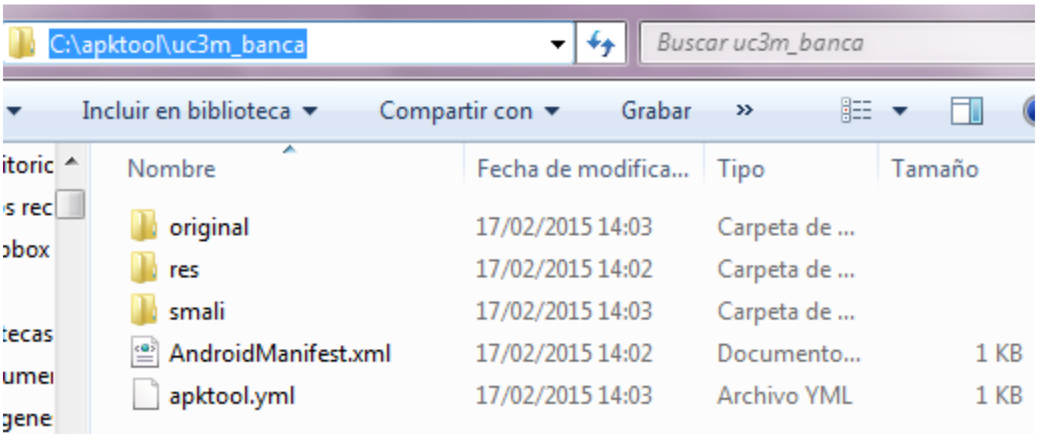

Figure 4.3. uc3m_banca decompiled folder

7. Note that there are smali files (Figure 4.4) beginning with both LoginActivity and R prefixes. We can conclude from this that those are the main application Java files—LoginActivity.java and R.java

8. Browse all the activity files in order to find which one is the main activity. Perhaps we could guide ourselves by looking where the variables' definitions are. We could also decompile another apk file for which we have the java code so we can have an idea on how smali code is organized.

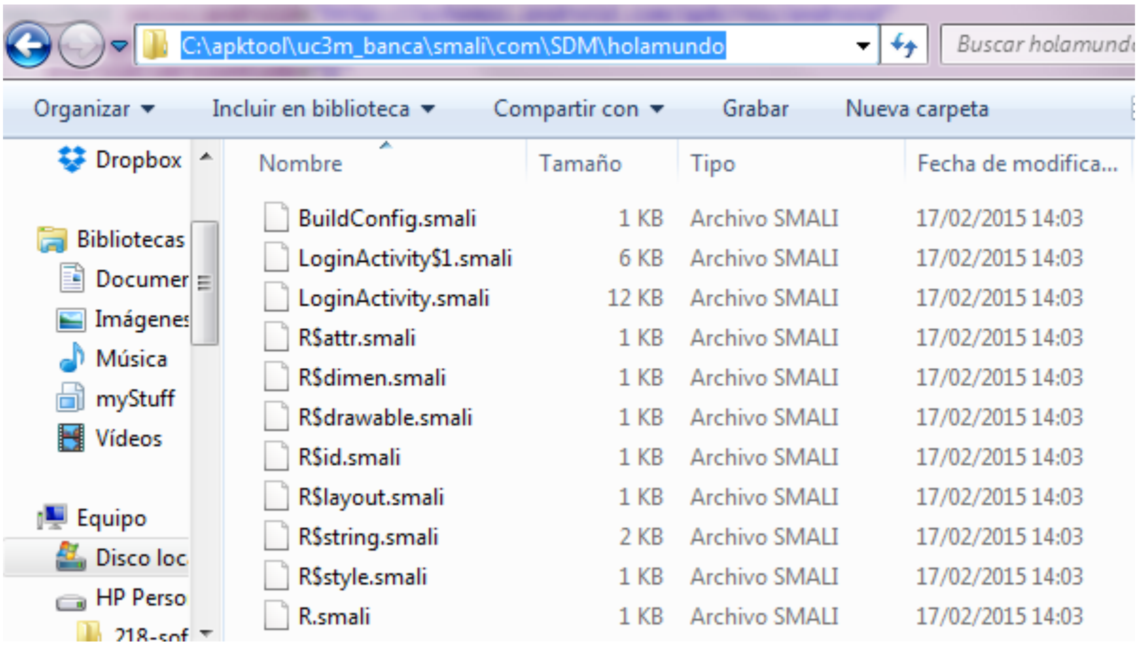

Figure 4.4. smali folder from the uc3m_banca.apk app

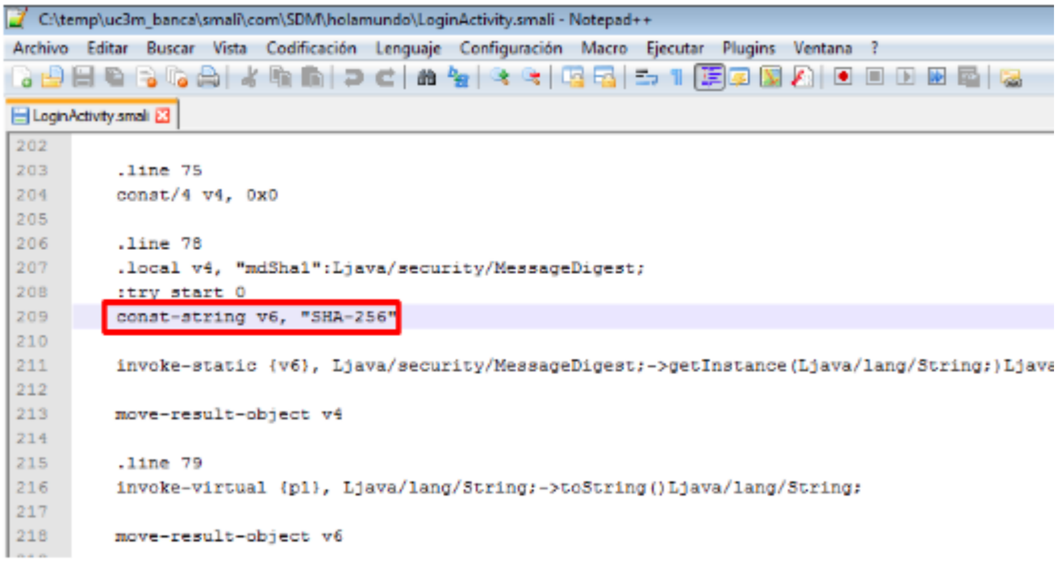

Figure 4.5. LoginActivity smali file showing the hash format used in the uc3m_banca.apk

9. After a while we find out that LoginActivity.smali is the main java file and also we can see in line 78 from the same file that SHA-256 is used to encrypt the password. So if SHA-256 is being used for hashing password user inputs from the login screen of the application, we must also think that this password is then compared against the stored password, so if they match, the user is logged into the application

10. Usually a **salt** is used in order to obfuscate the encryption and to make the password a little more robust. Remember that we are only using a 4 digit password for each user. We have to ask ourselves that the password is stored somewhere, where? To answer that question, please take a look on Figure 4.6a and Figure 4.6b

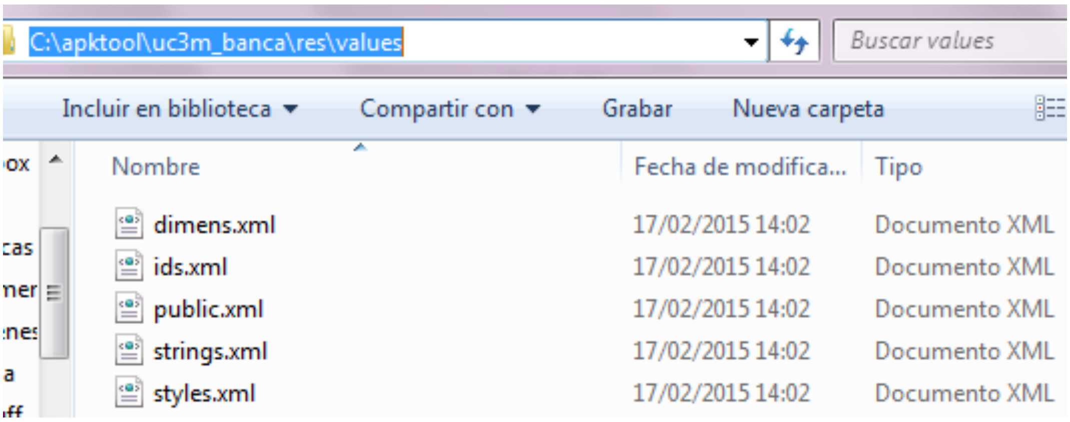

Figure 4.6.a. res folder from the uc3m_banca.apk app

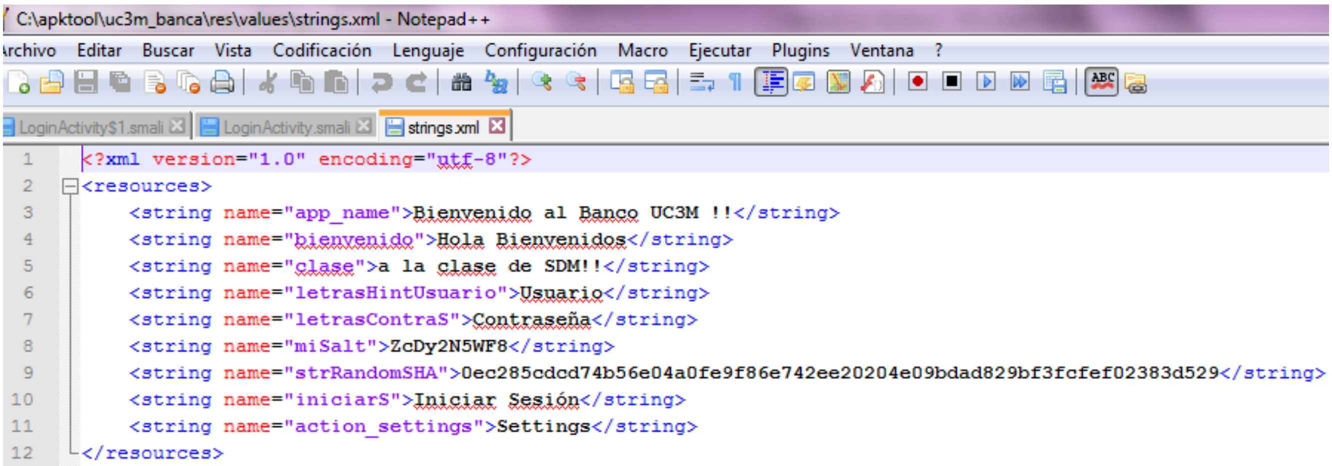

Figure 4.6.b. string.xml files from the uc3m_banca.apk app

11. Taking a look on line 87 (as it is in the original java code file) from the LoginActivity.smali file we find the code where the comparison is made from the hash stored in the device and the one that the user enters in his login screen (Figure 4.7). At this moment the password we are using to log in is composed from the **salt** part: “ZcDy2N5WF8” and the personal PIN code from the user “9999”. After compiling, the whole password will appear as “ZcDy2N5WF89999”, which after being hashed with SHA-256 will give the following string:

0ec285cdcd74b56e04a0fe9f86e742ee20204e09bdad829bf3fcfef02383d529

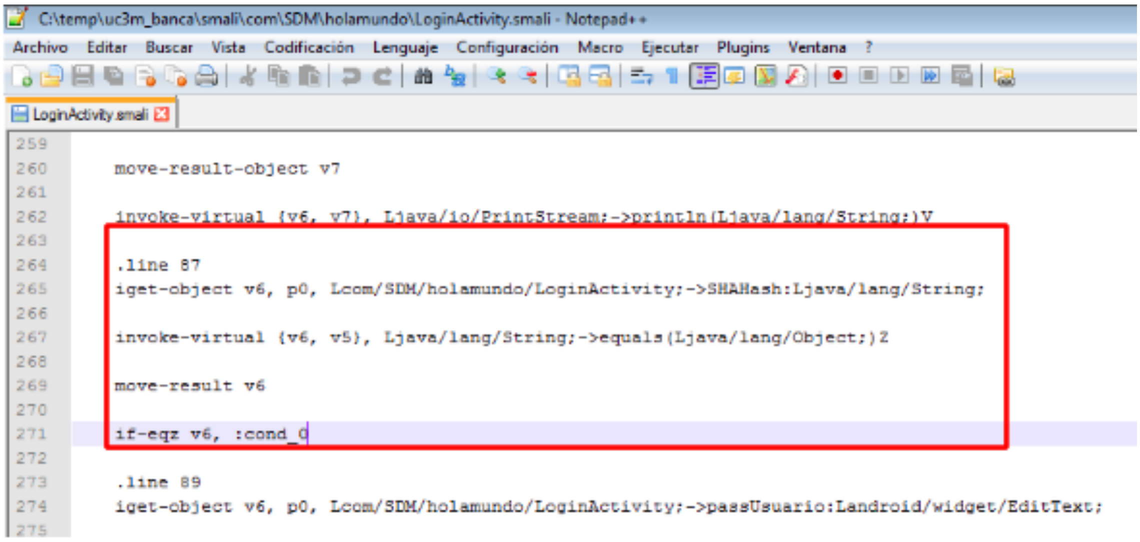

Figure 4.7. if-eqv v6 compare computed hash value with the hash values in v5

If you look further in Figure 5.8, you will find that in Line 73, the variable “pswCompleta” is declared (the variable's name in the original code), which is called v5 in the smali code. This variable contains the value that is generated when the user enters the password and is used to compare hash values as it is shown in Figure 4.7. If these values are the same (v6 and v5), the user is logged in. We can create an SHA-256 hash value and create a value for v5, thus modifying the password to our choice and bypassing authentication.

4.2 Changing your application

4.2.1 First Way

Since we are logging from the screen, the value we want to modify is the one that is stored because we know what we will be inputting as new value, mainly: the salt: “ZcDy2N5WF8” and the new PIN that the user will input; in this case PIN: 4321. Putting these two strings together we will get “ZcDy2N5WF84321” whose hash value is:

65eac95249546111f642a2c1289714fb38734729208167179b8576caafd6c34d1

You may use http://hashgenerator.de to generate a hash for any string you have.

12. From Figure 4.7 we found out the comparison is made between the variable v5 and v6, so we need to find the store v5 value in our LoginActivity.smali file and make the wanted changed for that variable (See Figure 4.8). v6 in this case is the computed variable in the program. Given what we saw in previous sections, it is not recommended to assign obvious names for sensitive variables. In this case the hash stored in the Res folder has a variable that holds the name “pswEnResFolder” meaning, password in the Res Folder. We have intentionally used this name so we can illustrate why it is important to choose meaningless variable names.

```
invoke-virtual {p0, v6}, Lcom/SDM/holamundo/LoginActivity;->getString(I)Ljava/lang/String;

move-result-object v0

.line 72
.local v0, "contraGuardada":Ljava/lang/CharSequence;
const-string v5, ""

.line 73
.local v5, "pswEnResFolder":Ljava/lang/String;
invoke-interface {v0}, Ljava/lang/CharSequence;->toString()Ljava/lang/String;

move-result-object v5

.line 74
sget-object v6, Ljava/lang/System;->out:Ljava/io/PrintStream;
```

Figure 4.8. Definition of v5. As we can see the name is obvious "pswEnResFolder"

13. Erase the part declaring the "move-result-object v5" because this is only saving the variable to a register. Erase all of line 73 as well where the v5 is somehow calculated. Then modify line 72: instead of **const-string v5, ""** write the definition of v5 as follow:

**const-string v5,
"65eac9524954611f642a2c1289714fb38734729208167179b8576caafd6e34d1"**

The new file will look as it is shown in Figure 4.9:

```
invoke-virtual {p0, v6}, Lcom/SDM/holamundo/LoginActivity;->getString(I)Ljava/lang/String;

move-result-object v0

.line 72
.local v0, "contraGuardada":Ljava/lang/CharSequence;

const-string v5, "65eac9524954611f642a2c1289714fb38734729208167179b8576caafd6e34d1"

.line 74
sget-object v6, Ljava/lang/System;->out:Ljava/io/PrintStream;

new-instance v7, Ljava/lang/StringBuilder;
```

Figure 4.9. Modification of the LoginActivity.smali file

4.2.1 Second Way

In addition to declaring constants, it is also possible to change or add the so-called "opcodes". Starting from the code without the modifications of section 4.2.1 and as seen in it (Figure 4.7), when the result of the comparison is “0” or “false” (opcode “if-eqz”) jumps to cond_0 . That is, if the hashes do not match, it jumps to that condition. Following the trace, it can be seen that it is the one in which the credentials are not valid (Figure 4.10).

```
:cond_0
:try_start_2
invoke-virtual {p0}, Lcom/SDM/holamundo/LoginActivity;->getApplicationContext()Landroid/content/Context;

move-result-object v6

const-string v7, "Sus credenciales no son validos .."

const/4 v8, 0x1
```

Figura 4.10. cond_0

Our intention is still to pass the verification process. An easy way to do it, among others, is to replace the opcode with its opposite: “if-nez”, being as follows (Figure 4.11):

```
if-nez v6, :cond_0
```

```
.line 87
iget-object v6, p0, Lcom/SDM/holamundo/LoginActivity;->SHAHash:Ljava/lang/String;

invoke-virtual {v6, v5}, Ljava/lang/String;->equals(Ljava/lang/Object;)Z

move-result v6

if-nez v6, :cond_0
```

Figura 4.11. Opcode modification in LoginActivity.smali file

In this case, the verification procedure will be passed as long as the comparison is different from 0, that is, with any password except the original.

The probability of knowing / guessing the original password of any application is low, and if successful, any other would activate the correct authentication. However, there are changes in the code that would allow access regardless of the password.

Therefore, the student is advised to play with the different possibilities of modifying the smali code and the different ways to pass said verification. Be creative!

A list of the different Dalvik opcodes can be found at the following link:

http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html

4.3 Recompiling your application

Once the apk has been modified, we need the new smali code to be reassembled and packaged into an apk file through the following command: apktool b (Figure 4.10). New Apk file will be placed in dist directory

1. Execute the command “apktool b” as it is shown in Figure 4.10 to reassembled the apk

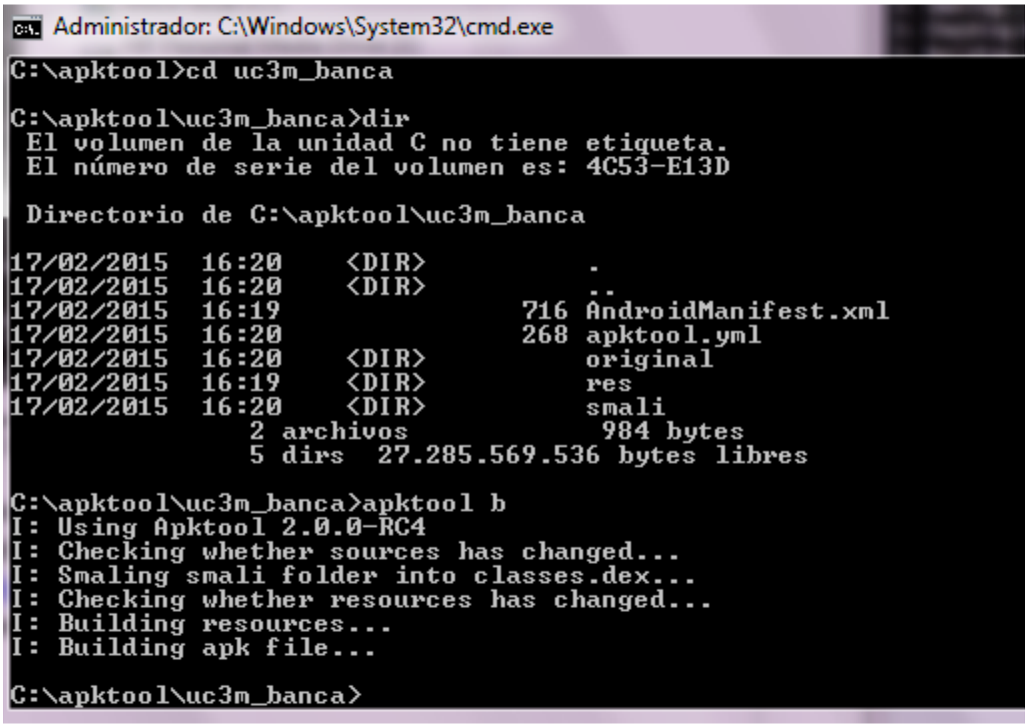

Figure 4.10. Reassembling the apk file

2. The new apk file will be located in the dist folder as it is shown in Figure 4.11.

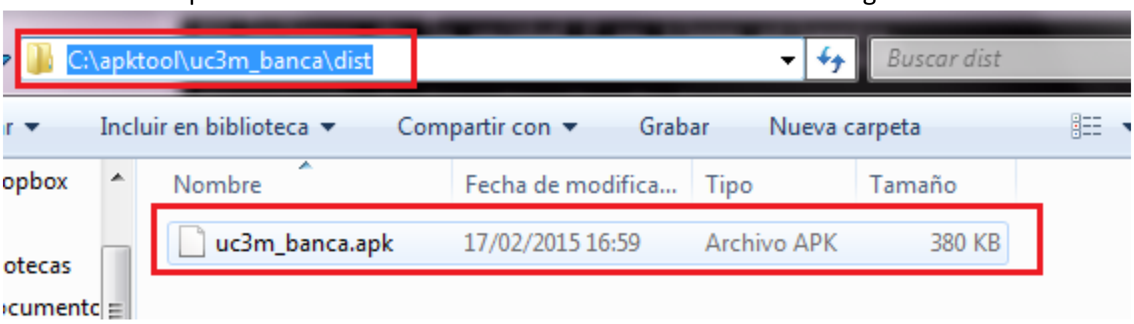

Figure 4.11. New reassembled apk file

3. The new apk needs to be signed before it can be installed on the device or emulator. The Signapk tool (Figure 4.12) is freely available tool for download. It has been uploaded on AG for convenience.

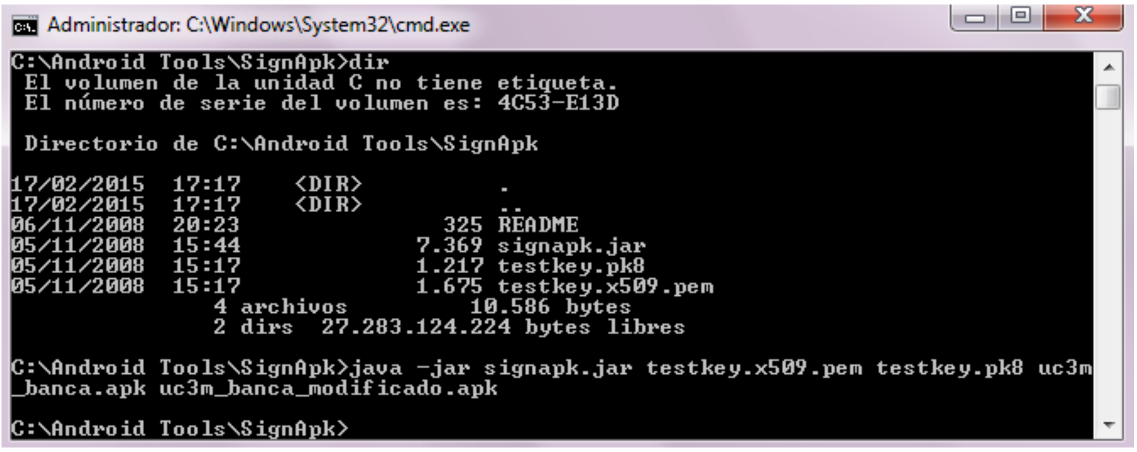

Figure 4.12. Using SignApk to resign a modified apk file

4. Signing the apk:

a. Download Signapk, create a folder with its name and put the modified .apk in it. Use the following command to sign the apk:

java -jar signapk.jar testkey.x509.pem testkey.pk8 <updated.apk> <update_signed.apk>

b. Use an existing key:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore <p
ath_to_keystore> -storepass <password_keystore> <update.apk> <alias_keystore>
```

5. Once the apk is re-signed, you must delete the original one that is in the emulator. To do that you open a cmd command and navigate until ../ platform-tools/. Once you are in the directory, run the command:

adb uninstall [nameOfThePackage]

First with the help of the adb and the command: **"adb shell pm list packages"** we obtain a list all the packages to make sure that our package is there, and then we remove it with the above command. We need to have the new modified apk that we will install in the same folder of the adb tool. We also renamed the modified package using the original name, changing it from uc3m_banca_modificada.apk to uc3m_banca.apk. Thus, we are ready to reinstall it.

1. Install the uc3m_banca.apk with the help of the command **"adb install uc3m_banca.apk"** (see Figure 4.13) and test the application with the new password; in this case, the user PIN is: 4321

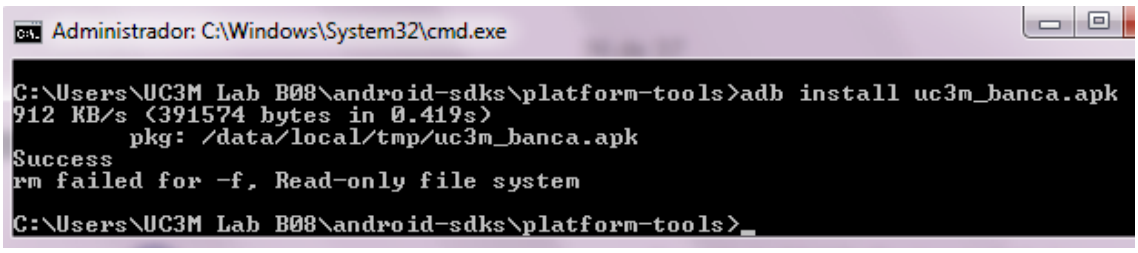

Figure 4.13. Reinstalling the modified apk on the emulator

2. Test the application on the emulator, as shown in Figure 4.14 and Figure 4.15 and Figure 4.16

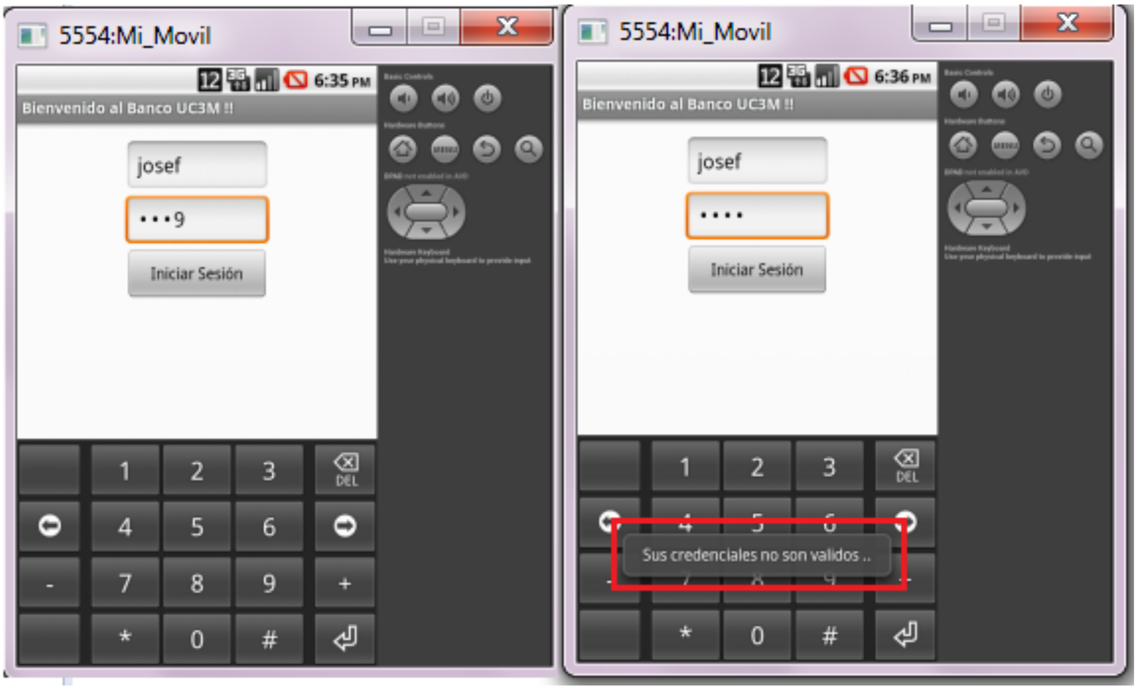

Figure 4.14. Testing the application with the older user PIN: 9999

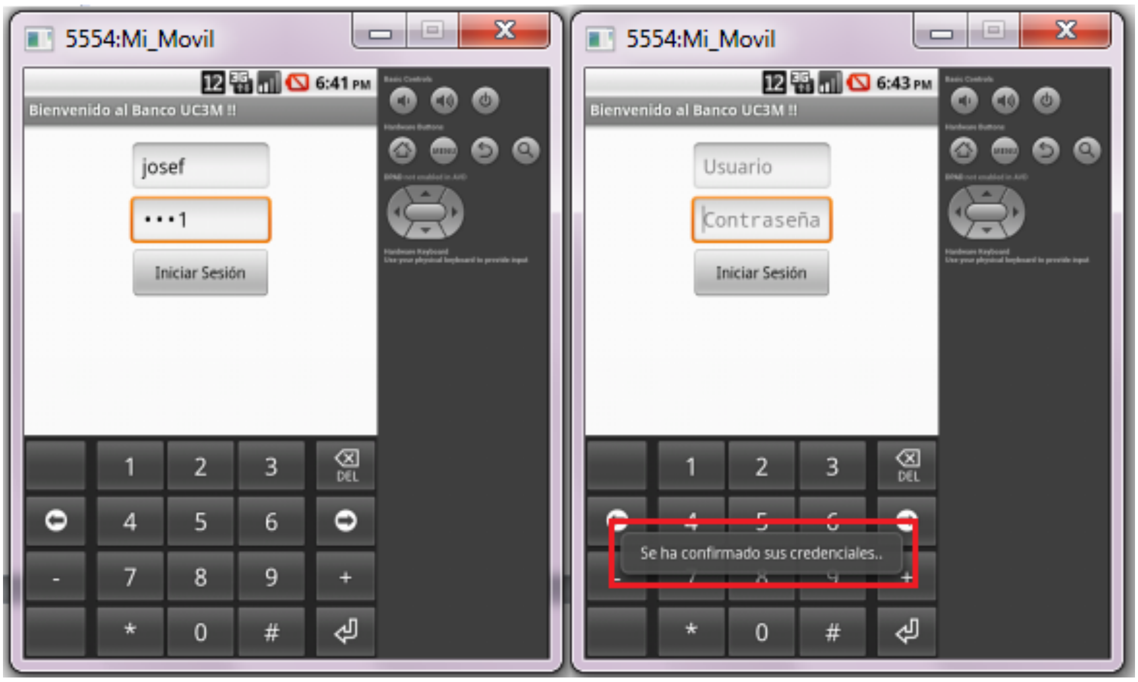

Figure 4.15. Testing the application with the new user PIN: 4321

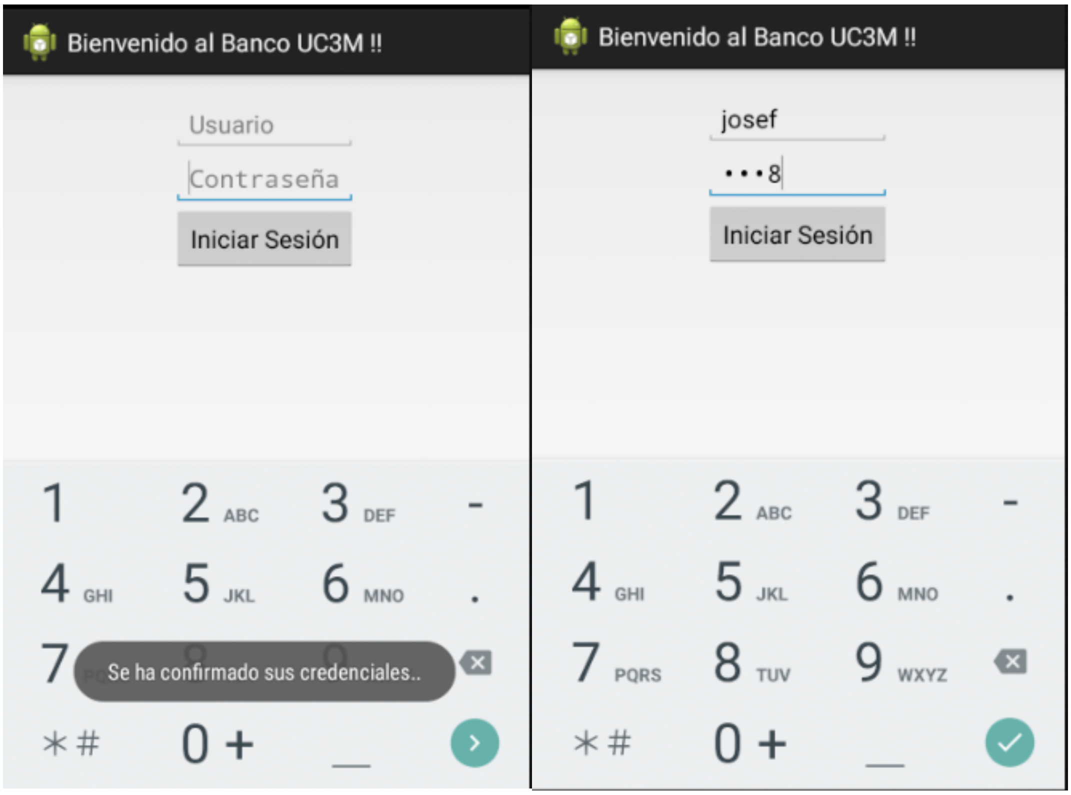

Figura 4.16. Testing the application with any PIN: 8888 (Second way)

The methodology described above can be used to analyze, decompile, and recompile an existing application. The example provided was used to describe a vulnerability that could have been exploited to bypass authentication and get access to application data or functionality. The vulnerability described here was not theoretical. There have been cases where a similar issue has resulted in compromised user data.