Lesson 7 – Reverse Engineering Decompilation

Introduction

This lesson focuses on finding, exploiting and fixing security issues in Android applications that are caused by lack of Reverse Engineering protections.

Even though Reverse Engineering protections may be considered Security by obscurity by some people, it still provides vital role as one of the levels in Defense in Depth approach and in protection of intellectual property of closed-source software.

Getting started

You will need:

- a device or emulator with Android (optionally),
- computer (Windows or Linux) with dex2jar[1], apktool[2] and jd-gui[3] and Android SDK (optionally).

Firstly, we are going to get the **release** (that is important, since binary protections are not usually deployed for debug executables) *.apk (Android Package) file. You can get it by:

- generating release build in Android Studio,
- downloading from the Internet (it is not possible to do it from Google Play Store),
- getting installed app from the device using the app (many applications called APK Extractor are available in Google Play Store),
- get it on your own using adb shell (no root access needed).

Getting on your own approach will be discussed in the following paragraph.

Connect the device to the computer using USB cable and check that it is properly discovered by typing:

```
user@goatdroid:~$:adb devices
List of devices attached
000ea1a82cebaf    device
```

If you cannot see the device check that drivers are installed[4] and the device is connected. In case of `unauthorized` text displayed next to device id click on the box displayed on the device to grant the computer access to the device.

You need to know the package name of the application. It can be found in Application Manager on the device or in Google Play. Having this knowledge, you can copy the apk file from the device.

```
user@goatdroid:~$ adb shell pm path org.owasp.goatdroid.fourgoats
package:/data/app/org.owasp.goatdroid.fourgoats-2/base.apk
user@goatdroid:~$ adb pull /data/app/org.owasp.goatdroid.fourgoats-2/base.apk
3975 KB/s (2680469 bytes in 0.658s)
```

[1] https://sourceforge.net/projects/dex2jar/

[2] https://ibotpeaches.github.io/Apktool/

[3] http://jd.benow.ca/

[4] You can find drivers at https://developer.android.com/studio/run/oem-usb.html.

We managed to get the absolute path to the file using the first command (you could not find it by listing the catalog since it requires root access) and downloaded the apk to the computer using the second command.

Finding vulnerabilities

Having the apk file on the computer we can start the Reverse Engineering. It is a normal zip file, so you can extract it with Archive Manager of your choice (if it does not work, change the extension to zip).

If you try to open xml files like AndroidManifest.xml or files in the res folder, you will find out that they are illegible. It is because they are encoded and need to be decoded first.

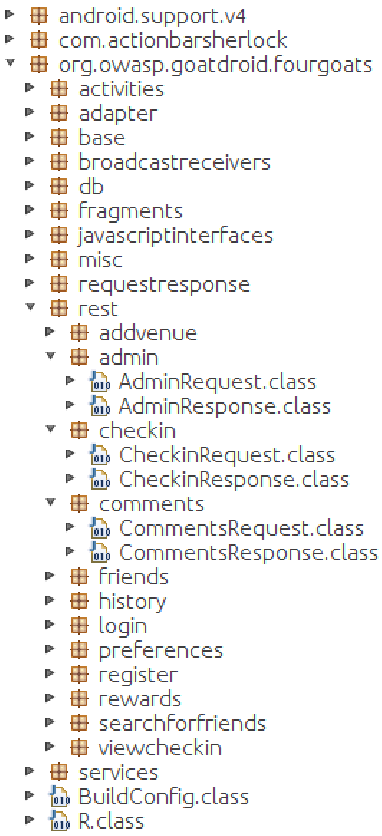
The extracted files contain one or more classes.dex files in the main root folder. These are the Dalvik Executable files containing the bytecode and can be converted to jar files using dex2jar tool in order to decompile them using Java decompiler.

Let’s start with converting classes.dex file to jar:

```
user@goatdroid:~$ d2j-dex2jar classes.dex
dex2jar classes.dex -> ./classes-dex2jar.jar
```

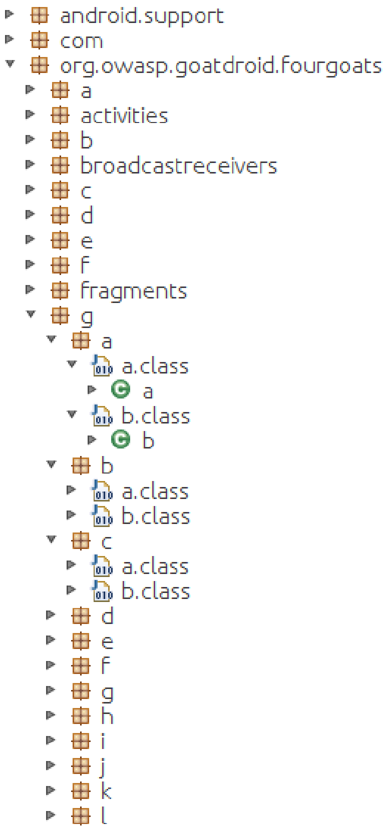
The tool created classes-dex2jar.jar file in the working directory. You can start jd-gui program and open the classes-dex2jar.jar file.

As you can see on the left of the Java Decompiler screen the whole application structure can be restored. Try reading decompiled source code. You can quickly find out, that it differs from the original source code.



```
▸ android.support.v4
▸ com.actionbarsherlock
▾ org.owasp.goatdroid.fourgoats
  ▸ activities
  ▸ adapter
  ▸ base
  ▸ broadcastreceivers
  ▸ db
  ▸ fragments
  ▸ javascriptinterfaces
  ▸ misc
  ▸ requestresponse
  ▾ rest
    ▸ addvenue
    ▾ admin
      ▸ AdminRequest.class
      ▸ AdminResponse.class
    ▾ checkin
      ▸ CheckinRequest.class
      ▸ CheckinResponse.class
    ▾ comments
      ▸ CommentsRequest.class
      ▸ CommentsResponse.class
    ▸ friends
    ▸ history
    ▸ login
    ▸ preferences
    ▸ register
    ▸ rewards
    ▸ searchforfriends
    ▸ viewcheckin
  ▸ services
  ▸ BuildConfig.class
  ▸ R.class
```

Illustration 1: Structure of unobfuscated FourGoats application



```
▸ android.support
▸ com
▾ org.owasp.goatdroid.fourgoats
  ▸ a
  ▸ activities
  ▸ b
  ▸ broadcastreceivers
  ▸ c
  ▸ d
  ▸ e
  ▸ f
  ▸ fragments
  ▾ g
    ▾ a
      ▾ a.class
        ▸ a
      ▾ b.class
        ▸ b
    ▾ b
      ▸ a.class
      ▸ b.class
    ▾ c
      ▸ a.class
      ▸ b.class
    ▸ d
    ▸ e
    ▸ f
    ▸ g
    ▸ h
    ▸ i
    ▸ j
    ▸ k
    ▸ l
```

Illustration 2: Structure of obfuscated FourGoats application

Source code of some of the method could not be decompiled by Java Decompiler, so you can see the Java bytecode listings. It looks similar to Assembly code:

```
public class ViewCheckinResponse
{
  /* Error */
   public static java.util.HashMap<String, String> parseCheckinResponse(String
paramString)
  {
    // Byte code:
    //   0: new 15      java/util/HashMap
    //   3: dup
    //   4: invokespecial 16  java/util/HashMap:<init>      ()V
    //   7: astore 5
    //   9: ldc 18
    //   11: astore 4
    //   13: new 20     org/json/JSONObject
    //   16: dup
    //   17: aload_0
    //   18: invokespecial 23 org/json/JSONObject:<init>
      (Ljava/lang/String;)V
```

Let’s compare the source code of a method and its decompiled version. For the purpose of comparison we will use the onCreate() method from Login Activity:

The decompiled version:

```
public void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130903081);
    this.context = getApplicationContext();
    this.userNameEditText = ((EditText)findViewById(2130968646));
    this.passwordEditText = ((EditText)findViewById(2130968652));
    this.rememberMeCheckBox = ((CheckBox)findViewById(2130968653));
    paramBundle = getSharedPreferences("credentials", 1);
    try
    {
      this.previousActivity =
getIntent().getExtras().getString("previousActivity");
      this.userNameEditText.setText(paramBundle.getString("username", ""));
      this.passwordEditText.setText(paramBundle.getString("password", ""));
      if (paramBundle.getBoolean("remember", true))
      {
        this.rememberMeCheckBox.setChecked(true);
        return;
      }
    }
    catch (NullPointerException localNullPointerException)
    {
      for (;;)
      {
        this.previousActivity = "";
      }
      this.rememberMeCheckBox.setChecked(false);
    }
  }
```

And the original source code:

```
public void onCreate(Bundle savedInstanceState) {
      super.onCreate(savedInstanceState);
      setContentView(R.layout.login);
      context = this.getApplicationContext();
      userNameEditText = (EditText) findViewById(R.id.userName);
      passwordEditText = (EditText) findViewById(R.id.password);
      rememberMeCheckBox = (CheckBox) findViewById(R.id.rememberMeCheckBox);
      SharedPreferences prefs = getSharedPreferences("credentials",
      MODE_WORLD_READABLE);
      try {
            previousActivity = getIntent().getExtras().getString(
      "previousActivity");
      } catch (NullPointerException e) {
            previousActivity = "";
      }
      userNameEditText.setText(prefs.getString("username", ""));
      passwordEditText.setText(prefs.getString("password", ""));
      if (prefs.getBoolean("remember", true))
            rememberMeCheckBox.setChecked(true);
      else
            rememberMeCheckBox.setChecked(false);
}
```

Do you see the difference? The names of local variables are not preserved, the code consists of many infinite loops, the Resource identifiers from R class are converted into numbers. But the general logic of the code is preserved.

Let’s move on to decoding resources. We will use the apktool for this task.

```
user@goatdroid:~$ java -jar apktool_2.2.1.jar d base.apk
I: Using Apktool 2.2.1 on base.apk
I: Loading resource table...
I: Decoding AndroidManifest.xml with resources...
I: Loading resource table from file:
/home/user/.local/share/apktool/framework/1.apk
I: Regular manifest package...
I: Decoding file-resources...
I: Decoding values */* XMLs...
I: Baksmaling classes.dex...
I: Copying assets and libs...
I: Copying unknown files...
I: Copying original files...
```

The command d used with apktool decodes the resources and decompiles the Dalvik Virtual Machine bytecode to smali language. The folder names the same as the apk file appeared in the working directory. It consists of:

- AndroidManifest.xml – decoded Android manifest file. This should be the starting point of vulnerability assessment. Analyze this file in order to get information about the application structure, used components and their security protections.
- apktool.yml – file generated by apktool and used for recompiling the application. It consists of information about the build config, including minimum and targeted SDK version.
- original – folder containing original (encoded) AndroidManifest.xml and META-INF directory with Jar manifest and digital signature used for signing of apk file.
- res – directory containing the decoded resources.
- smali – directory containing classes decoded to Smali language. It is Dalvik Virtual Machine assembly language. It is easier to analyze comparing to Java assembly language. May be useful for analyzing heavily obfuscated applications. It is used for changing the application code in order to recompile it.

Let’s look at the fragment of smali version of onCreate() method in Login Activity class:

```
.method public onCreate(Landroid/os/Bundle;)V
    .locals 6
    .param p1, "savedInstanceState"    # Landroid/os/Bundle;

    .prologue
    const/4 v5, 0x1

    .line 48
    invoke-super {p0, p1}, 
Lorg/owasp/goatdroid/fourgoats/base/BaseUnauthenticatedActivity;-
>onCreate(Landroid/os/Bundle;)V

    .line 49
    const v2, 0x7f030029

    invoke-virtual {p0, v2}, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>setContentView(I)V

    .line 50
    invoke-virtual {p0}, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>getApplicationContext()Landroid/content/Context;

    move-result-object v2

    iput-object v2, p0, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>context:Landroid/content/Context;

    .line 51
    const v2, 0x7f040046

    invoke-virtual {p0, v2}, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>findViewById(I)Landroid/view/View;

    move-result-object v2

    check-cast v2, Landroid/widget/EditText;

    iput-object v2, p0, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>userNameEditText:Landroid/widget/EditText;

    .line 52
    const v2, 0x7f04004c

    invoke-virtual {p0, v2}, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>findViewById(I)Landroid/view/View;

    move-result-object v2

    check-cast v2, Landroid/widget/EditText;

    iput-object v2, p0, Lorg/owasp/goatdroid/fourgoats/activities/Login;-
>passwordEditText:Landroid/widget/EditText;

…

.end method
```

It is more difficult to analyze comparing to decompilated source code, but it can still be readable and it is much easier to analyze than native assembly code.

Fixing the vulnerability

The easiest way to obfuscate Android application is to use ProGuard – open source Java bytecode optimizer available in Android Studio.

To enable ProGuard optimization put the following lines in the android section of build.gradle file:

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

MinifyEnabled parameter turns on ProGuard optimization. Proguard-android.txt is a default rules file used by ProGuard, proguard-rules.pro is a user-defined files with additional rules, that can change the way optimization is performed, for example exclude some files from ProGuard optimization.

Let’s compare the onCreate() method from Login Activity decompiled by Java Decompiler from obfuscated apk file:

```
public void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130903082);
    this.a = getApplicationContext();
    this.b = ((EditText)findViewById(2131427413));
    this.c = ((EditText)findViewById(2131427419));
    this.d = ((CheckBox)findViewById(2131427420));
    paramBundle = getSharedPreferences("credentials", 1);
    try
    {
      this.e = getIntent().getExtras().getString("previousActivity");
      this.b.setText(paramBundle.getString("username", ""));
      this.c.setText(paramBundle.getString("password", ""));
      if (paramBundle.getBoolean("remember", true))
      {
        this.d.setChecked(true);
        return;
      }
    }
    catch (NullPointerException localNullPointerException)
    {
      for (;;)
      {
        this.e = "";
      }
      this.d.setChecked(false);
    }
  }
```

Some of the classes names and class field names are changed into letters a, b, c, etc. to make the analysis of decompiled code harder.

There are also other, more powerful obfuscation tools available on the market that handle changing of the program control flow and Strings encryption. But Strings carry information and need to be decrypted in order to use, so there are deobfuscation tools available that handle Strings decryption in apk files. The class and field names do not carry any information needed by the program in order to execute properly, so they can be safely changed to meaningless letters.