Lesson 9 – Code tampering

Introduction

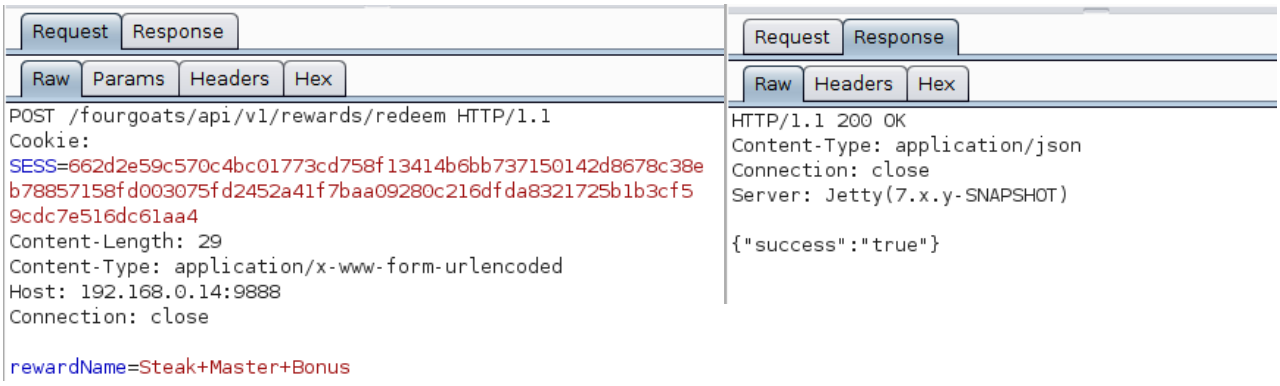
In the Reverse Engineering lesson we focused on decompilation. Now we will use this knowledge to perform a recompilation of the application in order to bypass client side business logic.

Getting started

The set up for this lesson is the same as in Lesson 8. We will be using Smali language to perform the recompilation. It is an assembly like language for Dalvik virtual machine used in Android.
There is not much reference on the language available. You can find the basics on the github page of Smali assembler/disassembler author https://github.com/JesusFreke/smali/wiki. The list of operation codes for the Dalvik machine can be useful and is available on http://pallergabor.uw.hu/androidblog/dalvik_opcodes.html.

Finding vulnerabilities

When redeeming the reward you can easily find out that the server only responds with boolean value. It means that the QR code generation happens fully on client side and you can suspect that it could be possible to bypass the check with server and use the same coupon many times:



Open the jar file generated from classes.dex with the decompiler. Take a look at the GetReward activity:

```
public void onCreate(Bundle paramBundle)
  {
    super.onCreate(paramBundle);
    setContentView(2130903058);
    this.context = getApplicationContext();
    this.mRewardname = getIntent().getStringExtra("rewardName");
    new RedeemReward(null).execute(new Void[0]);
  }

  private class RedeemReward
    extends AsyncTask<Void, Void, Boolean>
  {
    private RedeemReward() {}

    protected Boolean doInBackground(Void... paramVarArgs)
    {
      RewardsRequest localRewardsRequest = new RewardsRequest(GetReward.this.context);
      boolean bool1 = false;
      paramVarArgs = new UserInfoDBHelper(GetReward.this.context);
      try
      {
        boolean bool2 = localRewardsRequest.redeemReward(paramVarArgs.getSessionToken(),
GetReward.this.mRewardname);
        bool1 = bool2;
```

```
      }
      catch (Exception localException)
      {
        for (;;)
        {
          localException.printStackTrace();
          paramVarArgs.close();
        }
      }
      finally
      {
        paramVarArgs.close();
      }
      return Boolean.valueOf(bool1);
    }

    public void onPostExecute(Boolean paramBoolean)
    {
      if (paramBoolean.booleanValue())
      {
        GetReward.this.showCoupon();
        return;
      }
      Toast.makeText(GetReward.this.context, "Reward not available!", 1).show();
      GetReward.this.finish();
    }
  }
}
```

In the onCreate() method there is a call to the RedeemReward AsyncTask. It requests the server to redeem the coupon in the doInBackground() method. You can notice that the server response is only a boolean, it does not contain any other information. If you could manage to make it always return true and not to call the server to redeem the reward, you could possibly use the same reward many times. To do it, you could just put the return true at the beginning of the doInBackground() method.

It is not possible to perform the recompilation using Java, because the code shown by decompiler is only an attempt of interpretation of bytecode, which is more similar to assembly. To do the recompilation we will need to use the Smali language.
Use the apktool with d switch, the same way as in lesson 9.1 to decompile the apk to Smali. In the generated folder go to smali/org/owasp/goatdroid/fourgoats/activities/.

```
user@goatdroid:~$:~/app/smali/org/owasp/goatdroid/fourgoats/activities/ls |grep GetReward
GetReward$1.smali
GetReward$RedeemReward.smali
GetReward.smali
```

You will find three files starting with GetReward. In Smali every class is in separate file, including internal classes. The class we are interested in is RedeemReward, so open the file GetReward$RedeemReward.smali. Find the method doInBackground():

```
# virtual methods
.method protected varargs doInBackground([Ljava/lang/Void;)Ljava/lang/Boolean;
    .locals 7
    .param p1, "params"    # [Ljava/lang/Void;

    .prologue
    .line 84
    new-instance v2, Lorg/owasp/goatdroid/fourgoats/rest/rewards/RewardsRequest;

    iget-object v6, p0, Lorg/owasp/goatdroid/fourgoats/activities/GetReward$RedeemReward;-
>this$0:Lorg/owasp/goatdroid/fourgoats/activities/GetReward;

    # getter for: Lorg/owasp/goatdroid/fourgoats/activities/GetReward;-
>context:Landroid/content/Context;
    invoke-static {v6}, Lorg/owasp/goatdroid/fourgoats/activities/GetReward;-
>access$100(Lorg/owasp/goatdroid/fourgoats/activities/GetReward;)Landroid/content/Context;

    move-result-object v6
```

```
    invoke-direct {v2, v6}, Lorg/owasp/goatdroid/fourgoats/rest/rewards/RewardsRequest;-
><init>(Landroid/content/Context;)V

    .line 85
    .local v2, "rest":Lorg/owasp/goatdroid/fourgoats/rest/rewards/RewardsRequest;
    const/4 v3, 0x0
```

In the first lines, you have the method signature and declaration .locals. It specifies the number of non-parameter local registers. The 7 means that the available registers will be v0 to v7. To make the method always return true you can just but the value 1 to one of the registers and return it at the beginning of the method, so the code that is after return is not going to get executed.
We need to invoke static method valueOf on Boolean class to achieve this because the method needs to return Boolean object and not boolean simple type. For more difficult code it will be easier to write the needed code snippet in Java compile it to Android platform and decompile it to Smali to get the needed fragment in Smali.

```
# virtual methods
.method protected varargs doInBackground([Ljava/lang/Void;)Ljava/lang/Boolean;
    .locals 7
    .param p1, "params"    # [Ljava/lang/Void;

    const v0, 1
    invoke-static {v0}, Ljava/lang/Boolean;->valueOf(Z)Ljava/lang/Boolean;
    move-result-object v0
    return-object v0
```

Save the file and run apktool with b switch in the main folder of the decompilation results:

```
user@goatdroid:~/app$ java -jar apktool_2.2.1.jar b
I: Using Apktool 2.2.1
I: Checking whether sources has changed...
I: Smaling smali folder into classes.dex...
I: Checking whether resources has changed...
I: Building resources...
I: Building apk file...
I: Copying unknown files/dir...
```

You can find the generated apk file in dist directory. Try to install it with the adb command:

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:

```
user@goatdroid:~/app/dist$ adb install app-release.apk
4864 KB/s (2523685 bytes in 0.506s)
       pkg: /data/local/tmp/app-release.apk
Failure [INSTALL_PARSE_FAILED_NO_CERTIFICATES]
```

You will get the error because the generated apk file is not cryptographically signed. This is a security mechanism implemented in Android that is used to protect permissions (discussed in Lesson 1), to allow different apps from the same vendor to run in the same process and to prevent a possibility of a malicious update of the application. During the update process the certificates of the application update and the installed version are compared. Only the update signed with the same certificate is able to be completed successfully.

To sign the application you need to generate your own Java keystore with the keys:

```
keytool -genkey -v -keystore my_keystore.jks -keyalg RSA -keysize 2048 -validity 10000
-alias my_alias
```

And sign the application with it:

```
jarsigner -verbose -sigalg SHA1withRSA -digestalg SHA1 -keystore my_keystore.jks app.apk
my_alias
```

Try to install the application now:

```
user@goatdroid:~/app/dist$ adb install app-release.apk
4864 KB/s (2523685 bytes in 0.506s)
       pkg: /data/local/tmp/app-release.apk
Failure [INSTALL_FAILED_UPDATE_INCOMPATIBLE]
```

This is the security mechanism that prevents from installation of a malicious update. To install the app, you need to delete the old version first.
Now you can run the application and redeem the same coupon many times. The recompilation has been completed successfully.

Fixing the vulnerability

The most obvious vulnerability here is putting the weakness in application architecture – implementing critical business logic on client side. It is not a good idea, because there will always be a way to bypass it (either through recompilation, dynamic hooking of methods or traffic interception). The application should not generate the coupon with QR code based only on data it has already. The correct way to design the reward redeeming would be to send some kind of token in the server response to reward redeem request. This token should be put in the QR code and should be needed in order to use this coupon successfully at the restaurant.

The second issue exists in the application recompilation. How easy it was to make some changes in code and recompile it. Of course recompilation protections should not be used to protect against bad design of business logic. But it can serve as additional layer of protection in Defense in Depth methodology. It protects also against malicious attackers recompiling the application to add malware to it and distribute it in unofficial application markets. This does not affect application vendor directly, but can have a bad influence on vendor's reputation.

The simplest way of implementing the recompilation would be to check the signature and compare it to the constant String embedded into the application:
public boolean tamperedWith() {

```
    try {
        PackageManager pm = context.getPackageManager();
        PackageInfo info = pm.getPackageInfo("org.owasp.goatdroid.fourgoats",
PackageManager.GET_SIGNATURES);
        Signature[] signatures = info.signatures;
        if (signatures.length != 1) {
            return true;
        } else {
            String signatureString = new String(signatures[0].toChars());
            return !signatureString.equals("PUT HERE THE APPLICATION SIGNATURE STRING");
        }
    } catch (Exception e) {
        return true;
    }
}
```

This is just a simple example of performing a check of application signature. Of course finding and bypassing that method in the decompiled code would be easy. But this would stop some of the people reaching for the low-hanging fruit. Implementing this kind of checks in many places of the application code including native libraries would make it difficult and would cost much time for an attacker to bypass.