Saarland University
Information Security & Cryptography Group
Prof. Dr. Michael Backes

Android Security Lab WS 2014/15

Lab 2: Android Permission System

M.Sc. Sven Bugiel

Version 1.2 (October 6, 2014)

Version History

Version 1.1	Marked questions more clearly with leftbar Fixed error in code list for SMS broadcast receiver
Version 1.2	Improved code readability Numbered questions

Login on Lab Machines

Username:	smartseclab
Password:	android

Introduction

In this Lab you will familiarize yourself with the basics of Android's permission system. To this end, we will 1) use permissions to access the Android API, 2) extend the interfaces of the both apps App1 and App2 from the first lab session with permission-based protection, and 3) show permission inheritance in case applications share UIDs.

Additionally, we will implement a more fine-grained access control than permissions on the interfaces of App1, using the information provided by Binder and the PackageManager.

Android Permissions

Android defines roughly about 140 permissions.[1] These permissions are used to regulate an app's access to sensitive resources such as the onboard GPS sensor, the user's address book, or telephony functionality (the corresponding permissions would be `ACCESS_FINE_LOCATION`, `READ_CONTACTS`/`WRITE_CONTACTS`, and `CALL_PHONE`/`CALL_PRIVILEGED`).

Permissions are categorised in one of four protection levels:[2]

- *Normal*: Low-risk permission that will be automatically granted to an application without asking the user for consent.
- *Dangerous*: Higher-risk permission that would give a requesting application access to private user data or control over the device that can negatively impact the user. The user must confirm during app installation that the app can use this permission.
- *Signature*: A permission that the system grants only if the requesting application is signed with the same cryptographic key as the application that declared the permission. If the keys match, the system automatically grants the permission without notifying the user or asking for the user's explicit approval.
- *SignatureOrSystem*: Permissions that the system grants only to applications that are in the Android system image *or* that are signed with the same certificate as the application that declared the permission.

Most of those permissions are merely Strings associated with the UID of the application (for instance, `ACCESS_FINE_LOCATION` is a constant for the String "`android.permission.ACCESS_FINE_LOCATION`"). A permission check then consists of looking up such an association for particular permission-UID pair. However, a small subset of permissions are mapped to Groups of the underlying Linux kernel. That means, that the UID of an application that holds such a permission is member of the GID (Group) that Android maps the permission to.[3] In general, this concerns all permissions that are associated with privileged actions that are *not* proxied or provided by a system service or application, but with which the app directly interacts with the Linux kernel. This concerns primarily file-system operations such as access to the SDCard, opening a network socket, or accessing certain sysfs entries.

[1] See `http://developer.android.com/reference/android/Manifest.permission.html` for a complete list of the available permissions.

[2] `http://developer.android.com/guide/topics/manifest/permission-element.html`

[3] Check, for instance, `https://android.googlesource.com/platform/frameworks/base/+/master/data/etc/platform.xml` for a list of those mappings.

1 Accessing the Android API

In this first part of this lab exercise, you will extend App1 with permission to receive notifications about new SMS messages and to access the location information of the device.

1. First, we will add a *Listener* to App1, which we register with the system's *LocationManager* service to receive updates on the device's geolocation. To this end, extend the `MainActivity` of App1 with the following code:

```
GPSListner mGPSListener = new GPSListner();

@Override
protected void onResume()
{
  super.onResume();

  // Get a reference (Proxy) of the LocationManager service
  LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
  try {
    // Try to register our custom listener with the LocationManager service
    // Updates will be delivered as soon as the device moves (i.e., we would kill
       the battery of a real device...)
    lm.requestLocationUpdates(LocationManager.GPS_PROVIDER, 0, 0, mGPSListener);
  } catch (SecurityException e) {
    Toast.makeText(this, "Security Exception: "+e, Toast.LENGTH_LONG).show();
    Log.e("App1.MainActivity", "Security Exception", e);
  }
}

@Override
protected void onPause()
{
  super.onPause();
  // Don't leak the listener
  LocationManager lm = (LocationManager) getSystemService(Context.LOCATION_SERVICE);
  lm.removeUpdates(mGPSListener);
}

//Our custom listener class. Right now will simply log any received location updates
   .
private class GPSListner implements LocationListener {

  @Override
  public void onLocationChanged(Location location) {
    Log.d("App1.GPSListener", "New location: "+location.getLongitude()+":"+location.
       getLatitude());
  }

  @Override
  public void onProviderDisabled(String provider) {}

  @Override
  public void onProviderEnabled(String provider) {}

  @Override
  public void onStatusChanged(String provider, int status, Bundle extras) {}
}
```

2. Now try executing App1

Question 1:
What is the result of the execution?

3. Next, we will extend the Manifest of App1 such that the app requests the `ACCES_FINE_LOCATION` permission. Either edit the Manifest code as shown in the following code listing or use the ADT's GUI as shown in Figure 1.

```
<manifest ...>
  <uses-sdk ... />

  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

  <application ...>
    ...
  </application>
</manifest>
```

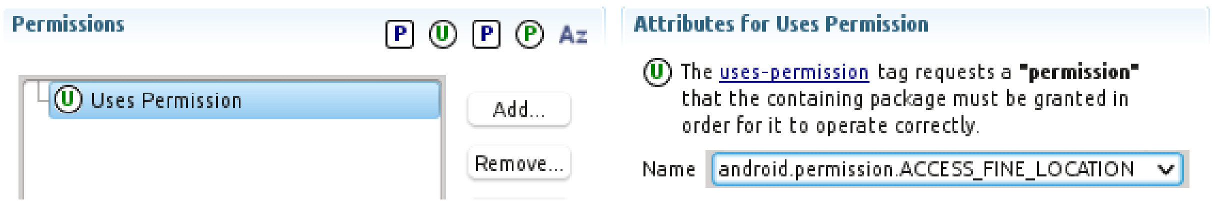

Permissions
Uses Permission
Add...
Remove...
Attributes for Uses Permission
The uses-permission tag requests a "permission" that the containing package must be granted in order for it to operate correctly.
Name
android.permission.ACCESS_FINE_LOCATION

Figure 1: Requesting permissions in the app's Manifest.

4. Now test the code again by executing App1 and then changing the location of the virtual device using either the Emulator Control (in DDMS) or using the `geo` command of the emulator (after connecting to the telnet server):

```
geo fix <longitude value> <latitude value>
```

Question 2:
What is the new result of the execution? Can you see the GPSListener log?

5. Next, we will extend the BroadcastReceiver component `Receiver.java` of App1 to receive notification of newly received SMS messages.

   (a) First add a new `IntentFilter` for the corresponding `receiver` to the Manifest, which will catch all broadcasts with Action string `android.provider.Telephony.SMS_RECEIVED`

   (b) Second, extend the BroadcastReceiver code to properly parse and log the SMS messages. All messages will be delivered as payload to the broadcast Intents. Use the following code for this purpose:

```
@Override
public void onReceive(Context context, Intent intent) {
  if(intent.getAction().equals("android.provider.Telephony.SMS_RECEIVED"))
  {
    // Get the SMS message from the Intent payload ("extras")
    Bundle extras = intent.getExtras();
    String messages = "";

    // If present, reconstruct and log the SMS message. Intent can contain more than
         one message!
    if ( extras != null )
    {
       // Get received SMS array
       Object[] smsExtra = (Object[]) extras.get( "pdus" );
```

```
      for ( int i = 0; i < smsExtra.length; ++i )
      {
        SmsMessage sms = SmsMessage.createFromPdu((byte[])smsExtra[i]);

        String body = sms.getMessageBody().toString();
        String address = sms.getOriginatingAddress();

        messages += "SMS from " + address + " :\n";
        messages += body + "\n";
      }
    }
    Log.d("App1.Receiver", messages);
  } else if (intent.getAction().equals("com.example.intent.action.BROADCAST"))
  {
    Log.d("App1.Receiver", "Called with Intent "+intent.toString());
  } else {
    Log.d("App1.Receiver", "Called with unknown Action "+intent.getAction());
  }
}
```

6. Build and run App1 again. Then send a new SMS message to the virtual device and observe the log for any entry concerning the delivery of the corresponding broadcast message to the BroadcastReceiver of App1.

Question 3:
What log entry can you observe?

7. If you observed a security error message in the log, fix App1 such that it can successfully receive the SMS message!

2 Defining Custom Permissions

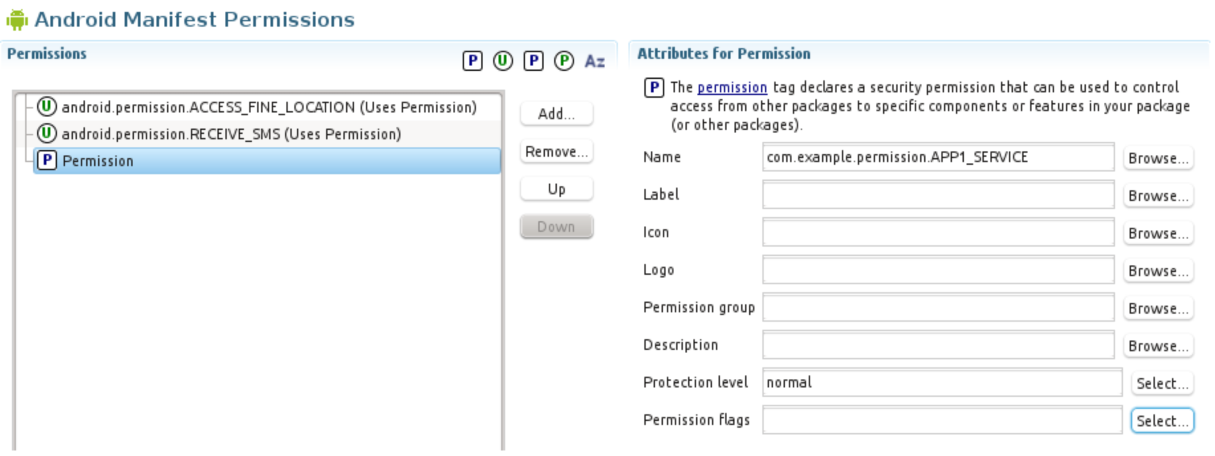
In this second part of the exercise, you will define a new custom permission for App1 to protect its service interface.

1. First, we define a custom permission with the *normal* protection level, i.e., any application can request this permission without any further restrictions and user intervention. Thus, extend the Manifest of App1 to define a new permission `com.example.permission.APP1_SERVICE` by either editing the Manifest manually as in the following listing or by using the ADT GUI as shown in Figure 2:

```
<manifest ...>
  <uses-sdk ... />

  <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>
  <uses-permission android:name="android.permission.RECEIVE_SMS"/>
  <permission android:name="com.example.permission.APP1_SERVICE" android:
      protectionLevel="normal"/>

  <application ...>
    ...
  </application>
</manifest>
```



android.permission.ACCESS_FINE_LOCATION (Uses Permission)
android.permission.RECEIVE_SMS (Uses Permission)
Permission

The permission tag declares a security permission that can be used to control access from other packages to specific components or features in your package (or other packages).

Name	com.example.permission.APP1_SERVICE
Label	
Icon	
Logo	
Permission group	
Description	
Protection level	normal
Permission flags	

Figure 2: Defining a new permission.

2. To protect the service interface of App1 with this new permission, the service's Manifest entry has to be extended with an `android:permission` attribute as follows:

```
<application ...>

  ...

  <service
    android:name="com.example.app1.RemoteService"
    android:permission="com.example.permission.APP1_SERVICE" >
    <intent-filter>
      <action android:name="com.example.app1.IRemoteService" />
    </intent-filter>
  </service>

</application>
```

3. To better illustrate the protection levels of custom permissions, we use from now on only signed applications. To this end, we have to create cryptographic keys used to sign application packages. Fortunately, ADT provides good tool support in creating and using cryptographic keystores when exporting your apps as application packages.

4. To export your application as a signed package, right click on your project in ADT and chose the option as depicted in Figure 3:

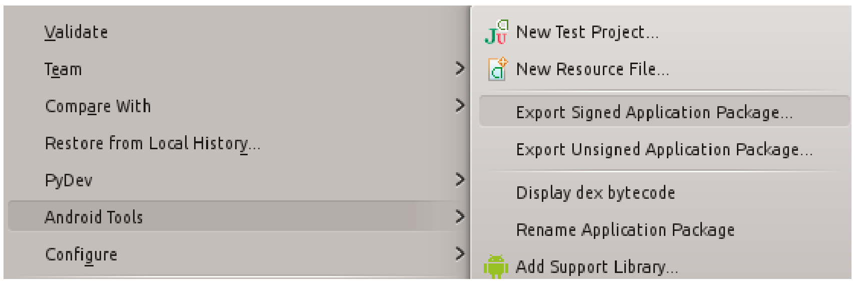
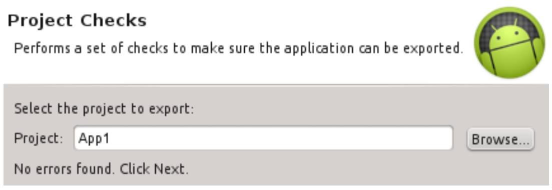


Figure 3: Export application as signed package.

5. The wizard requires you to specify a keystore file. First time you export, you probably don't have a keystore yet, thus follow the following instructions to create one and directly use it to export your signed application:


Project Checks

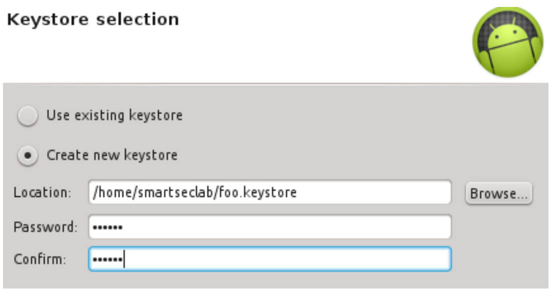
Performs a set of checks to make sure the application can be exported.

Select the project to export:

Project: App1

No errors found. Click Next.

Figure 4: Select your app project, i.e., App1.


Keystore selection

Use existing keystore

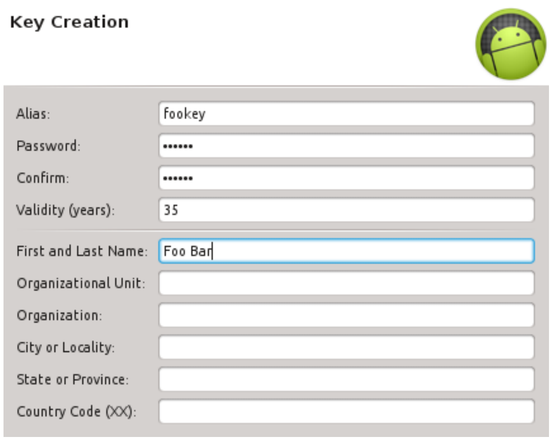
● Create new keystore

Location: /home/smartseclab/foo.keystore

Password: ••••••

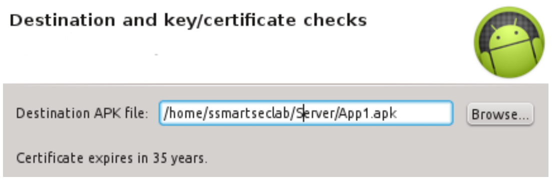
Confirm: ••••••

Figure 5: Unless you already have an existing keystore, create a new one. This password authenticates access to the keystore as a whole.


Key Creation

Alias:	fookey
Password:	••••••
Confirm:	••••••
Validity (years):	35
First and Last Name:	Foo Bar
Organizational Unit:	
Organization:	
City or Locality:	
State or Province:	
Country Code (XX):	

Figure 6: Create a new key. The filled fields in the figure are mandatory. Android requires, that the key must be valid at least 35 years.

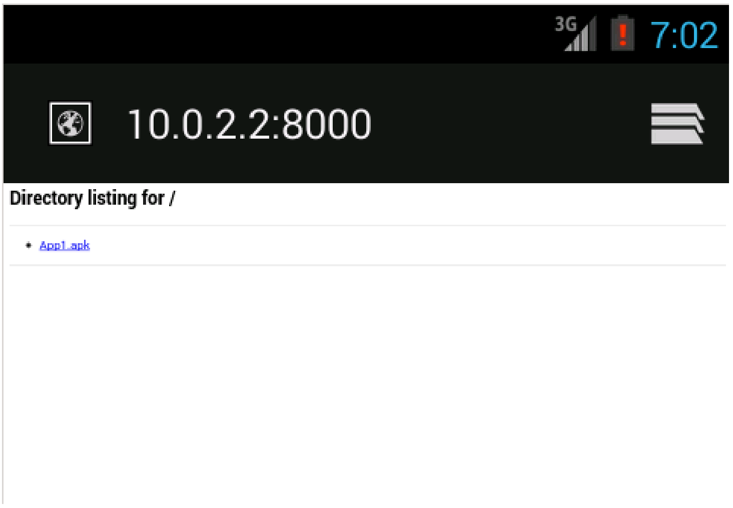

Destination and key/certificate checks

Destination APK file: /home/ssmartseclab/Server/App1.apk

Certificate expires in 35 years.

Figure 7: For the next part of this exercise, it is important that you save the application package in the *Server* directory of your home directory.

6. Next, we install the signed application package. Because Android uses a *same-origin policy* for application updates (and re-installation) based on the signature key of apps, you first have to uninstall the current version of APP1, which is signed with a different key. To do so, you have two options:
   (a) Open the *Settings* app in your AVD, go to *Apps*, chose APP1, and then *Uninstall*
   (b) Close the AVD and start with the *Wipe user data* option, which removes **any** user data including **all** 3rd party apps (i.e., APP1 *and* APP2)
7. When the uninstallation is completed, we use side-loading to install the signed version of APP1:
   (a) On your PC, open a shell and go to the *Server* directory in your home folder
   (b) Execute the command:
   ```
   $ python srv.py
   ```
   This will start a minimalistic webserver, which serves the content of this directory at port 8000 of your localhost.
   (c) Go to the *Settings* app of your AVD, go to *Security*, and check the option *Unknown sources* to allow side-loading
   (d) Open the webbrowser on your AVD and go to the URL: `http://10.0.2.2:8000/` 10.0.2.2 is the IP of your host machine and allows your AVD to communicate with your host.
   (e) You should see a website like Figure 8.
   (f) Click on the *App1.apk* link to start the download of the application package.
   (g) After the download is complete, open the APK on your AVD to start the installation. You should see a permission request as in Figure 9.



Directory listing for /

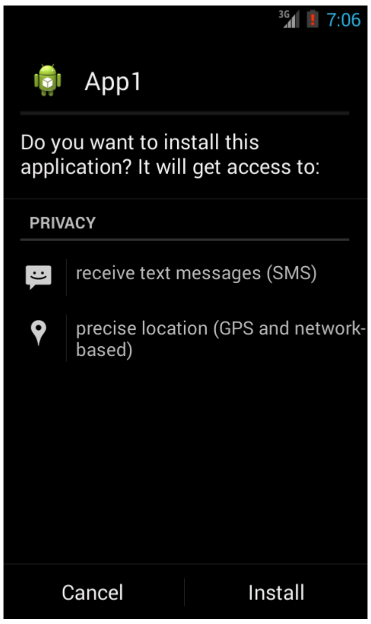
- App1.apk

Figure 8: Web browser showing the content of your *Server* directory.

8. Extend the Manifest of APP2 to request your custom permission `com.example.permission.APP1_SERVICE`.
9. Export APP2 also as a signed package and install it via side-loading (do not forget to uninstall it first, if you didn't wipe your user partition). However, follow to the following steps:
   (a) First, use a **new** signing key for APP2 which is hence **different** from the key for APP1.

   Question 4:
   Does APP2 successfully bind to APP1's service and call the function? Why/Why not?

   (b) Uninstall APP2.


7:06

App1

Do you want to install this application? It will get access to:

PRIVACY

receive text messages (SMS)

precise location (GPS and network-based)

Figure 9: Permission request by App1 during installation.

(c) Now change the protection level of `com.example.permission.APP1_SERVICE` to *signature*, sign App1 with your first key, and re-install **both** App1 and App2 in this order (no uninstall is needed, because the same key is used).

Question 5:
Does App2 successfully bind to App1's service and call the function? Why/Why not? During the installation of App2, can you observe in the log a warning of the Package-Manager regarding App2's permissions? Did you see any warning on the UI during the installation of App2?

(d) Finally, uninstall App2, re-export it signed with the same key as App1, install it, and verify that it can successfully bind to App1's service.

3 Shared UIDs and Permission Inheritance

Android allows developers to use shared UIDs. That means, that two or more applications will be assigned the same UID in the system and thus share the same sandbox, including permissions. However, for security reasons, two apps can only share a UID if they are signed with the same key, i.e., are from the same origin (developer).

1. Add to the manifests of both App1 and App2 an `android:sharedUserId` attribute[4] for the shared UID `com.example.uid.smartseclab`.
2. In the `onResume()` functions of both apps, log their UIDs (`android.os.Process.myUID()`) using `Log.d`.
3. Export both apps signed with the same key.
4. First install App1.
5. Next install App2 and note the requested permissions during the installation.

Question 6:
Which permissions does App2 request and why? If you would install first App2 and then App1, what would change in this scenario?

[4]`http://developer.android.com/guide/topics/manifest/manifest-element.html`

4 Protected Broadcasts (Optional/Homework)

Because broadcast Intents are often used to notify apps about important system events or to carry privacy/security sensitive information to apps (e.g., newly received SMS), it is important to ensure that such broadcasts 1) are only delivered to sufficiently privileged receivers and 2) can only be send by the correct, trusted apps.

Protected by Sender

To ensure that broadcasts are only delivered to privileged receivers, the sender can define a permission that a receiving application must hold in order to get the broadcast.[5] For instance, the SMS received broadcast from earlier required the receiver to hold the permission `android:name="android.permission.RECEIVE_SMS`.

1. Extend App2 to send the broadcast with action string `com.example.action.BROADCAST` and requires the receiver to hold a new custom permission `com.example.permission.APP2_BROADCAST`.
2. Extend App1 to receive this broadcast.
3. Test this scenario for normal and signature protection level as well as different combinations of signing keys for App1 and App2.

Protected by Receiver

To ensure that a receiver receives broadcast Intents only from a trusted and sufficiently privileged sender app, Android allows developers to protect their receivers with permissions.[6] That means, that a sender of a broadcast must hold this permission in order to successfully send the Intent to this receiver. Note, however, that this enforcement is *per receiver*, i.e., receivers which do not require any permission from the sender would still receive the broadcast!

4. To test this mechanism, extend App1 or App2 to send a broadcast for a newly received SMS message, i.e., spoofing SMS messages:

```
sendBroadcast(new Intent("android.provider.Telephony.SMS_RECEIVED"));
```

Question 7:
What is the warning message in the system log and which permission is required to send broadcast Intents with this action string in order to spoof SMS messages?
Can you find the manifest declaring this receiver component in the Android source code?

5. Extend App1's receiver such that sending the broadcast with action string `com.example.action.BROADCAST` to App1's receiver requires the sender to hold the custom permission `com.example.permission.APP1_SERVICE` (for simplicity). Again, verify this scenario for normal and signature protection level as well as different combinations of signing keys for App1 and App2, in particular in conjunction with protection level for the second custom permission `com.example.permission.APP2_BROADCAST`.

Question 8:
Which combination of protection levels do you have to chose for your custom permissions, such that only apps of same origin can be the sender of the broadcasts received by App1 and only apps explicitly allowed by the end-user can receive the broadcast? How would you implement this?

[5] `http://developer.android.com/reference/android/content/Context.html#sendBroadcast(android.content.Intent,java.lang.String)`
[6] http://developer.android.com/guide/topics/manifest/receiver-element.html

5 Using Binder and PackageManager for fine-grained access control (Optional/Homework)

While permission attributes set in application manifests can protect entire component interfaces (e.g., calling *any* function of a service component requires the caller to hold the permission), this access control is sometimes to coarse-grained. For instance, the *LocationManager* service you used in the first part of this exercise actually requires the caller to hold either the `ACCESS_FINE_LOCATION` or `ACCESS_COARSE_LOCATION` permission depending on whether the GPS or the network is used for determining the location.

To implement such a more fine-grained access control, developers can use the information provided by the Binder IPC mechanism in conjunction with the *PackageManager* service.

5.1 Extending App1's service component

We will instrument App1's service component with an access control specifically for the `getPID` function.

1. First, remove the permission attribute from the service component in App1's manifest. Thus, all application are able to bind to this component and call `getPID`.
2. Next, we extend `getPID` such that it only returns the PID when
   (a) the caller package name equals to `com.example.app2`
   (b) the version of the caller package is at least 2
3. The following code snippet presents an implementation of this access control. Code comments explain the functionality in more technical detail. Note that if you would extend the interface of the service (e.g., adding a second function `getGID()`). you would have to instrument the implementation of each new function separately with access control!

```
@Override
public int getPID() throws RemoteException {
  Log.d("App1.Service", "From: "+Binder.getCallingUid());

  // Get the UID of the calling process
  int callerUID = Binder.getCallingUid();
  // Get reference to the PackageManager service
  PackageManager pm = getPackageManager();
  // Get all package names installed under this UID
  String[] packageNamesForUid = pm.getPackagesForUid(callerUID);

  for(String name : packageNamesForUid)
  {
    Log.d("App1.Service", "Checking caller package "+name);
    // We require that there must only one application installed under this UID and
        its package name equals com.example.app2. Otherwise return SecurityException
         to the caller.
    if(!name.equals("com.example.app2"))
    {
      throw new SecurityException("Only package com.example.app2 is allowed to
          access this function!");
    }

    // Retrieve more detailed information about the caller package from the
        PackageManager service
    PackageInfo pi;
    try {
      pi = pm.getPackageInfo(name, 0);
    } catch (NameNotFoundException e) {
      Log.e("App1.Service", "Error retrieving package info", e);
      throw new RemoteException("Error retrieving package info for "+name);
    }

    // Ensure that the version code of the caller is sufficient
    if(pi.versionCode < 2)
    {
      throw new SecurityException("Caller version to low. Required is at least
          version 2!");
    }
  }

  // At this point we know that the caller must com.example.app2 with at least
      version code 2
  // We use android.os.Process
  return Process.myPid();
}
```

Question 9:
The code snippet presented above does actually not ensure that `getPID` of `com.example.app1` is called by the correct/trusted `com.example.app2` that the developer of APP1 had in mind. Can you explain why? Can you implement a better approach using more/different package information?

5.2 Extending APP1's BroadcastReceiver component

Next, we examine whether the same mechanism as shown in Section 5.1 can be used in APP1's BroadcastReceiver in order to check whether APP1 can trust received broadcast messages.

1. Extend APP1's `Receiver.java` such that it checks that
   (a) Intents with action string `android.provide.telephony.SMS_RECEIVED` are always send from the `com.android.mms` application
   (b) Intents with action string `com.example.intent.action.BROADCAST` are always send from the `com.example.app2` application

Question 10:
What is the result of these new checks? Can you explain this result by analyzing the Android source code[7], in particular

- **the Binder user-space tools in**
  - `frameworks/base/core/java/android/os/Binder.java`
  - `frameworks/base/core/jni/android_util_Binder.cpp`
  - `frameworks/native/libs/binder/IPCThreadState.java`
- **and Android's message dispatching between application threads in**
  - `frameworks/base/core/java/android/os/Handler.java`
  - `frameworks/base/core/java/android/os/Looper.java`
  - `frameworks/base/core/java/android/app/ActivityThread.java`

A Code for minimalistic Python webserver

```
import SimpleHTTPServer
import SocketServer

PORT = 8000

Handler = SimpleHTTPServer.SimpleHTTPRequestHandler

httpd = SocketServer.TCPServer(("", PORT), Handler)

print "serving at port", PORT
httpd.serve_forever()
```