Mobile Security WS 17/18

Exercise 0: Android Application Programming

(Not graded)
Introduction

In this first exercise, we provide a simple introduction to application programming on Android, with focus on familiarizing you with the most important IDE elements and on inter-application communication based on *Intents*, *ContentProviders*, and *Services*.

1 Development environment

In this exercise we use the Android Studio IDE based on IntelliJ IDEA \(^1\) If you are familiar with Android application programming and prefer another IDE, you are free to switch the IDE and skip the following introduction of the IDE elements.

1. If you are using your private laptop/PC, please install the Android Studio IDE first, since the following steps refer to this IDE.

2. Start the Android Studio IDE

3. Create a new Android Studio project. Since we continue to use this app in the following exercises, please use *Service App* as Application name and *trust.cispa.saarland* as Company Domain, resulting in *saarland.cispa.trust.serviceapp* as package name.

4. Next, chose *Phone and Tablet* as target device with *API 23: Android 6.0 (Marshmallow)* as Minimum SDK

5. Next, chose *Empty Activity* and leave the default options (e.g., *Activity_Main* as name), then *Finish* the project setup

6. Once the setup finished, you will be presented with programming environment

   - On the left hand side you should see the project’s folder structure
     (a) *manifests*: Contains the *AndroidManifest.xml* for this app
     (b) *java*: All the app logic stays here
     (c) *res*: Contains the app’s resources and assets (e.g., layout definition for GUIs or String values)

   ![Figure 1: App project structure](http://developer.android.com/sdk/installing/studio.html)

   - On the top you see Android development specific tools such as:
     (a) Building the app project
     (b) Running the app, for instance, on an attached device or within an device emulator
     (c) Android Virtual Device (AVD) manager to create, delete, edit, start/stop Android emulators
     (d) SDK Manager to install support for other Android versions
   - On the bottom you can see the Android monitor, where debugging and log output will appear when you are executing an app
   - Lastly, via *Tools*$$\rightarrow$$*Android*$$\rightarrow$$*Android Device Monitor* to, e.g., get device system information, create screenshots, read the device log out (*logcat*), etc. (note that the emulator control in this menu is disabled by default)

---

\(^1\) [http://developer.android.com/sdk/installing/studio.html](http://developer.android.com/sdk/installing/studio.html)
2 Creating an Android Virtual Device

1. Open the AVD Manager.

2. Create a new Nexus 5 device.

3. Create a new x86 Marshmallow API Level 23 Android 6.0 device, you can leave the default AVD name and finish the creation.

4. Launch this device in a new emulator (you should see debugging output in the Android device monitor). The first boot a new device can take a little bit longer, so be patient.

5. Alternatively to the GUI tools in Android Studio, the android debugging bridge (adb) tool of the Android SDK can be used to communicate with attached devices or emulators. Open a terminal on Linux and use the adb tool to observe the device’s log output:

   \$ cd ~/Android/Sdk/platform-tools
   \$ ./adb logcat

For a full list of the adb features (e.g., emulator control), please refer to the adb help page.

6. Open the emulator control directly from the emulator menu, from where you can inject different hardware events into the device.

7. For instance, under Phone you can send an SMS message to the device.

8. The emulator control is essentially just a GUI for a telnet-based service to send commands to the emulator. Please refer to https://developer.android.com/studio/run/emulator-commandline.html#console for a description on how to control the emulator via telnet. For instance, once telnet authentication to the emulator has been done, sending an SMS message could also be accomplished from the terminal using:
$ telnet localhost 5554
$sms send 5556 "Your message"

where 5554 is the ID of the emulator (check the emulator window name) and 5556 the sender number of the SMS. Using those numbers, two or more parallel running emulators could send each other SMS messages.

9. Create and boot a second virtual device using the same configuration for SDK version as for the first device
10. Once this second device has booted, go to the SMS (“Messaging”) app and send a test SMS to the first emulator device. The phone number of each device equals the number (e.g., 5554) in the title bar of the emulator window.
11. To end emulators, simply press and hold the power button, then chose Power off device
12. Close both emulators and the AVD window.

3 Creating a first Android App

1. We will now implement the app that we created in the previous task
2. To quickly test your app, start one of your virtual devices and wait until it has booted up. Then press the Run 'app' button and chose the connected emulator from the menu as the target device to install and run the app
3.1 Adding a BroadcastReceiver component

1. Create a new BroadcastReceiver with class name Receiver in your package (i.e., saarland.cispa.trust.Receiver) via right-clicking on app or the java folder, and leave it Enabled and Exported.

2. The new component is also directly added to the AndroidManifest.xml

   `<receiver
     android:name="Receiver"
     android:enabled="true"
     android:exported="true">
   </receiver>

3. Remove the UnsupportedOperationException from the onReceive method and add the following line of code in order to log the received Intent message:

   `Log.d("ServiceApp.Receiver", "Called with Intent "+intent.toString());`

4. Next edit the receiver’s entry in the manifest to add an IntentFilter to inform the system about which kinds of Intents should be delivered to this receiver:

   `<receiver
     android:name="Receiver"
     android:enabled="true"
     android:exported="true">
     <intent-filter>
       <action android:name="saarland.cispa.trust.intent.action.BROADCAST"/>
     </intent-filter>
   </receiver>

This defines an IntentFilter, which means the BroadcastReceiver will only receive broadcasted Intents that match the action string saarland.cispa.trust.intent.action.BROADCAST.
5. To quickly test your new BroadcastReceiver, re-install your modified app by running it again on the virtual device. Start monitoring the logcat output in the device monitor. Run in a terminal the following command to instruct the ActivityManager on the device via the android debug bridge to send a broadcast that fits the IntentFilter of the receiver component:

```
$ adb shell am broadcast -a saarland.cispa.trust.intent.action.BROADCAST
```

6. If done correctly, you should be able to see debug output similar to:

```
D/ServiceApp.Receiver: Called with Intent Intent { act=saarland.cispa.trust.intent.action.BROADCAST flg=0x10 cmp=saarland.cispa.trust.serviceapp/.Receiver }
```

7. A broadcast receiver can also be registered dynamically at runtime through an application’s context. However, at this point, we refer you to the documentation for more information on this.

### 3.2 Adding a simple Service component

1. Next, we will add a simple Service component to the application to which other apps can bind and then interact with in a client-server fashion. First, define the interface of the new service through the Android Interface Definition Language (AIDL). Create in your Service App source code package a new file IRemoteService.aidl, which will be placed by Android Studio in a newly added AIDL folder:

![Image](image1.png)

Figure 7: Adding an AIDL file

```
Figure 8: Added AIDL files folder
```

2. In the auto-generated AIDL file, remove any existing method, like basicTypes, and instead add the following method:

```
String getVersion();
```

3. Now we will implement this interface in a dedicated Service component (cf. Figure 14 and Figure 15 in Appendix A for an illustration of a Service’s life-cycle). Thus, add a new component RemoteService to your package.

[http://developer.android.com/reference/android/content/BroadcastReceiver.html](http://developer.android.com/reference/android/content/BroadcastReceiver.html)
4. Next, we have to implement the Service logic, e.g., for binding, in the create Service component skeleton. Add the following method to the Service component and edit the `onBind` method:

```java
static private String SERVICE_VERSION = "1.0";

// Stub of our service, i.e., implementation of the interface
private final IRemoteService.Stub mBinder = new IRemoteService.Stub() {
  @Override
  public String getVersion() throws RemoteException {
    // We use android.os.Process
    return SERVICE_VERSION;
  }
  
  @Override
  public IBinder onBind(Intent intent) {
    return mBinder;
  }
};
```

The Stub implements the Service interface. The only remotely callable function `getVersion()` simply returns the service’s version string to the caller. Callers that bind to this service, receive an IBinder interface object for this service, which they can use for RPC (remote procedure calls).

5. Creating the Service component also registered it directly in the `AndroidManifest.xml`:

```xml
<service
  android:name=".RemoteService"
  android:enabled="true"
  android:exported="true"></service>
```

6. To make the service component better discoverable, we add an Intent filter:

```xml
<service
  android:name=".RemoteService"
  android:enabled="true"
  android:exported="true">
  <intent-filter>
    <action android:name="saarland.cispa.trust.intent.service.REMOTE_SERVICE"/>
  </intent-filter>
</service>
```
7. To test this service, we create a simple second caller application.

### 3.3 Creating a second Android App

1. Repeat the steps for the creation of Service App, however, call this new app Frontend App in the same Company Domain as Service App, same target SDK, and an empty Activity called FrontendActivity instead of the default MainActivity.

2. Since Frontend App, as a caller to the service of Service App, has to know the Service interface (i.e., Proxy object), we have to copy the AIDL file from the package of Service App to the package of Frontend App.

   - Create a new aidl folder in the app of Frontend App

   ![Figure 10: Manually create AIDL folder](image)

   - Then create a new package saarland.cispa.trust.serviceapp in the aidl folder and copy the IRemoteService.aidl file from the Service App to the new package in the Frontend App

3. For this setup, we keep binding and calling to App1’s service very simple and integrate them strictly into the life-cycle of App2’s main activity (cf. Figure 13 in Appendix A for an illustration of the Activity life-cycle). Add the following code to the MainActivity.java of App2:

```java
private ServiceConnection mConnection = new ServiceConnection() {
    public void onServiceConnected(ComponentName className, IBinder service) {
        // This gets an instance of the IRemoteInterface, which we can use to call on the service
        mIRemoteService = IRemoteService.Stub.asInterface(service);
        String version = "";
        try {
            version = mIRemoteService.getVersion();
        } catch (RemoteException e) {
            Log.e("FrontendActivity", "Error calling getVersion()", e);
        }
        Log.d("FrontendActivity", "getVersion()=\"+
```
super.onResume();

Intent i = new Intent("saarland.cispa.trust.intent.service.REMOTE_SERVICE");
i.setPackage("saarland.cispa.trust.serviceapp");

if (bindService(i, mConnection, Context.BIND_AUTO_CREATE))
{
    Toast.makeText(this, "Binding to successful", Toast.LENGTH_LONG).show();
} else {
    Toast.makeText(this, "Binding to failed!", Toast.LENGTH_LONG).show();
}

@Override
protected void onPause()
{
    super.onPause();
    if (mIRemoteService != null)
    {
        unbindService(mConnection);
    }
}

This code will bind to the service every time the Activity comes back to the foreground on the screen (onResume()) and will directly use the IBinder object received upon binding to call the remote service. The received version string from Service App is logged. The result of the binding operation is briefly shown as a pop-up (Toast) on the screen to the user. To correctly cast the IBinder object to the interface of Service App’s service, FRONTEND APP requires the AIDL file. To avoid leaking service connections, the activity unbinds from the service as soon as the activity is not longer in the foreground on the screen (onPause()). If everything works correctly, you should see the following log entry every time the FRONTEND APP comes to foreground on screen:

D/FrontendActivity: getVersion()=1.0

4. Install first SERVICE APP and afterwards FRONTEND APP by running them as Android applications. Observe the screen for successful binding and the logcat output for the logged version string.

5. For simpler service implementations, such as IntentServices, we refer to the Android documentation.

6. Finally, let’s extend the FRONTEND APP to also send broadcasts and Intents, which the SERVICE APP can receive, but triggered via the FRONTEND APP’s GUI.

   • Open the res/layout/activity_frontend.xml in Android Studio
   • Either in the XML or in the graphical editor, remove the “Hello World!” TextView
   • Add two buttons and in their properties give the first button with ID button the label Send broadcast and the second button with ID button2 the label Start Activity

![Figure 11: Buttons added to FRONTEND APP](http://developer.android.com/reference/android/app/IntentService.html)
Lastly, in the `onCreate` method of the `FrontendActivity` find those buttons and add logic that is triggered on clicking them:

```java
private Button sndBroadcastBtn;
private Button startAcrtivityBtn;

@Override
protected void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.activity_frontend);

    sndBroadcastBtn = (Button)findViewById(R.id.button);
    sndBroadcastBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            // Implicit intent via action string
            sendBroadcast(new Intent("saarland.cispa.trust.intent.action.BROADCAST"));
        }
    });

    startAcrtivityBtn = (Button)findViewById(R.id.button2);
    startAcrtivityBtn.setOnClickListener(new View.OnClickListener() {
        @Override
        public void onClick(View view) {
            // explicit Intent via Component
            Intent i = new Intent();
            i.setComponent(new ComponentName("saarland.cispa.trust.serviceapp", "saarland.cispa.trust.serviceapp.MainActivity"));
            startActivity(i);
        }
    });
}
```

Try these two buttons: After sending broadcasts, you should see the log entries from the receiver component and after starting the Activity, you should see the `MainActivity` of the Service App come to foreground on screen.

### 3.4 Creating a ContentProvider

Next, you will extend the Service App with a `ContentProvider` component.

1. First, create a new ContentProvider component and chose `Provider` as class name and set URI authorities to `saarland.cispa.trust.serviceapp.contentprovider`. For further information, please refer to the Android developer documentation [here](https://developer.android.com/guide/topics/providers/content-provider-creating.html).

2. Next, we define the meta data of our new ContentProvider. Create a new class `ContentProviderMetaData`:

```java
public class ContentProviderMetaData {
    public static final String AUTHORITY = "saarland.cispa.trust.serviceapp.contentprovider";
    public static final String DATABASE_NAME = "cp.db";
    public static final int DATABASE_VERSION = 1;
    public static final String TABLE_NAME = "emails";
}
```
private ContentProviderMetaData() {}

// describe the only existing table
public static final class TableMetaData implements BaseColumns {
    private TableMetaData() {}
    
    public static final String TABLE_NAME = "emails";
    public static final Uri CONTENT_URI = Uri.parse("content://" + AUTHORITY + "emails");
    public static final String CONTENT_TYPE = "vnd.android.cursor.dir/vnd.serviceapp.emails";
    public static final String CONTENT_ITEM_TYPE = "vnd.android.cursor.item/vnd.serviceapp.emails";
    public static final String NAME = "name";
    public static final String EMAIL = "email";
}

3. Next, we add necessary helper classes to the ContentProvider. Add the following code at the beginning of the Provider class:

private static final String TAG = "ServiceApp.Provider";

// Mechanism to identify the incoming URI patterns
private static final UriMatcher sUriMatcher;
private static final int DATA = 1;
private static final int DATA_ID = 2;
static {
    sUriMatcher = new UriMatcher(UriMatcher.NO_MATCH);
    sUriMatcher.addURI(ContentProviderMetaData.AUTHORITY, "emails", DATA);
    sUriMatcher.addURI(ContentProviderMetaData.AUTHORITY, "emails/#", DATA_ID);
}

private static class DatabaseHelper extends SQLiteOpenHelper {
    DatabaseHelper(Context context) {
        super(context, ContentProviderMetaData.DATABASE_NAME, null, 
             ContentProviderMetaData.DATABASE_VERSION);
    }
4. Finally, we can complete the implementation of the ContentProvider by implementing the necessary methods:

```java
@Override
public void onCreate(SQLiteDatabase sqLiteDatabase) {
    Log.d(TAG, "inner oncreate called");
    sqLiteDatabase.execSQL("CREATE TABLE " + ContentProviderMetaData.TableMetaData.TABLE_NAME + "(" +
    ContentProviderMetaData.TableMetaData._ID + " INTEGER PRIMARY KEY AUTOINCREMENT," +
    ContentProviderMetaData.TableMetaData.NAME + " TEXT," +
    ContentProviderMetaData.TableMetaData.EMAIL + " TEXT" +
    ");
}

@Override
public void onUpgrade(SQLiteDatabase sqLiteDatabase, int oldVersion, int newVersion) {
    Log.d(TAG, "inner onupgrade called");
    Log.w(TAG, "upgrading database from "+oldVersion+" to "+newVersion);
    // Here: Just drop and re-create
    sqLiteDatabase.execSQL("DROP TABLE IF EXISTS " + ContentProviderMetaData.TableMetaData.TABLE_NAME);
    onCreate(sqLiteDatabase);
}

@Override
public int delete(Uri uri, String selection, String[] selectionArgs) {
    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
    case DATA:
        count = db.delete(ContentProviderMetaData.TableMetaData.TABLE_NAME, selection, selectionArgs);
        break;
    case DATA_ID:
        String rowid = uri.getLastPathSegment();
        count = db.delete(ContentProviderMetaData.TableMetaData.TABLE_NAME, ContentProviderMetaData.TableMetaData._ID + "=" + rowid
            + (!TextUtils.isEmpty(selection) ? " AND (" + selection + ")") : ""), selectionArgs);
        break;
    default:
        throw new IllegalArgumentException("Unknown URI " +uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}

@Override
public String getType(Uri uri) {
    switch (sUriMatcher.match(uri)) {
    case DATA:
        return ContentProviderMetaData.TableMetaData.CONTENT_TYPE;
    case DATA_ID:
        return ContentProviderMetaData.TableMetaData.CONTENT_ITEM_TYPE;
    }
```
default:
    throw new IllegalArgumentException("Unknown URI "+uri);
}

@Override
public Uri insert(Uri uri, ContentValues values) {
    if (sUriMatcher.match(uri) != DATA) {
        throw new IllegalArgumentException("Unknown URI "+uri);
    }

    ContentValues cv;
    if (values != null) {
        cv = new ContentValues(values);
    } else {
        cv = new ContentValues();
    }

    if (cv.containsKey(ContentProviderMetaData.TableMetaData.NAME) == false) {
        throw new SQLException("Failed to insert row because name is needed");
    }
    if (cv.containsKey(ContentProviderMetaData.TableMetaData.EMAIL) == false) {
        throw new SQLException("Failed to insert row because email is needed");
    }

    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    long rowid = db.insert(ContentProviderMetaData.TableMetaData.TABLE_NAME, "name", cv);
    if (rowid > 0) {
        Uri insertedDataURI = ContentUris.withAppendedId(ContentProviderMetaData.CONTENT_URI, rowid);
        getContext().getContentResolver().notifyChange(insertedDataURI, null);
        return insertedDataURI;
    }

    throw new SQLException("Failed to insert row into "+uri);
}

@Override
public boolean onCreate() {
    Log.d(TAG, "outer oncreate called");
    mOpenHelper = new DatabaseHelper(getContext());
    return true;
}

@Override
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs, String sortOrder) {
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    switch (sUriMatcher.match(uri)) {
    case DATA:
        qb.setTables(ContentProviderMetaData.TableMetaDataTable.TABLE_NAME);
        break;
    case DATA_ID:
        qb.setTables(ContentProviderMetaDataTable.TABLE_NAME);
        qb.appendWhere(ContentProviderMetaDataTable._ID + "=" + uri.getLastPathSegment());
        break;
    default:
        throw new IllegalArgumentException("Unknown URI "+uri);
    }
    ...
String orderBy;
if (TextUtils.isEmpty(sortOrder)) {
    orderBy = "name DESC";
} else {
    orderBy = sortOrder;
}

SQLiteDatabase db = mOpenHelper.getReadableDatabase();
Cursor c = qb.query(db, projection, selection, selectionArgs, null, null, orderBy);

c.setNotificationUri(getContext().getContentResolver(), uri);
return c;

@Override
public int update(Uri uri, ContentValues values, String selection, String[] selectionArgs) {
    SQLiteDatabase db = mOpenHelper.getWritableDatabase();
    int count;
    switch (sUriMatcher.match(uri)) {
        case DATA:
            count = db.update(ContentProviderMetaData.TableMetaData.TABLE_NAME, values, selection, selectionArgs);
            break;
        case DATA_ID:
            String rowId = uri.getPathSegments().get(1);
            count = db.update(ContentProviderMetaData.TableMetaData.TABLE_NAME, values, ContentProviderMetaData.TableMetaData._ID + "=" + rowId + (!TextUtils.isEmpty(selection) ? " AND (" + selection + ")" : ""), selectionArgs);
            break;
        default:
            throw new IllegalArgumentException("Unknown URI " + uri);
    }
    getContext().getContentResolver().notifyChange(uri, null);
    return count;
}

5. As last task, extend the Frontend App to insert, query, delete entries from the provider of the Service App. For this task we let you investigate the necessary documentation. However, in the end your Frontend App should:
(a) have three more buttons: Add Random Email, Delete All Emails, and Show All Emails
(b) use a ContentResolver, which you can get from the app’s Context via getContentResolver, to add, delete, and query names and emails from the Service App’s Provider
(c) use a TextView or ListView together with CursorAdapter to display queried names and emails
A Activity and Service Life-Cycle

Figure 13: Activity lifecycle (Source: http://developer.android.com/reference/android/app/Activity.html)
Figure 14: General Service life-cycle (left: service creation with \textit{startService()}; right: service creation with \textit{bindService()}; source \url{http://developer.android.com/guide/components/services.html})
Figure 15: Bound-Services life-cycle (Source: [http://developer.android.com/guide/components/bound-services.html](http://developer.android.com/guide/components/bound-services.html)).