All this text wasn’t written by me (Tiago a.k.a TigaxMT).

All of it was transcribed by me from HackerOne video lessons.

All credits and thanks go to them.

Common Android Bugs Pt. 1

Intents and Activities

Activities

Activities are single, focuses things for users to do. If you're thinking in terms of MVC, an Activity is a controller; it sets up views, handles incoming intents, communicates with the outside world, etc.
Everything you see in an Android app is driven by an Activity.

In order to be used under most circumstances, an Activity must be published in the application's manifest file. For example:

```
<activity
    android:name="com.example.app.MyActivity"
    android:label="@string/title_my_activity" >
</activity>
```

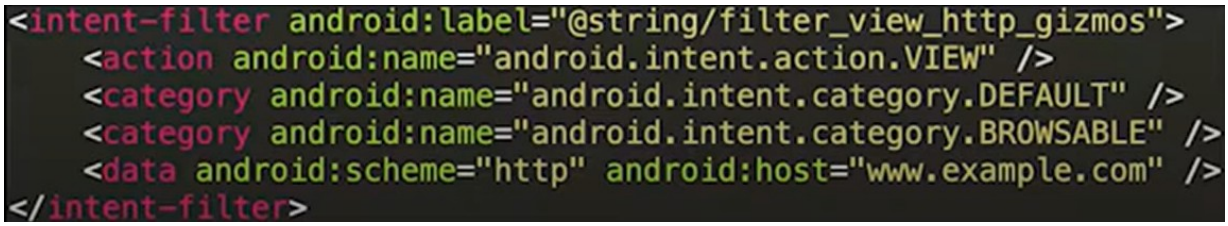
Intents

Intents are used to start activities, send messages to services, and more. Typically an intent will be sent to one specific target, but they can also be broadcast widely as below:

```
Intent call = new Intent(Intent.ACTION_DIAL);
call.setData(Uri.parse("tel:2125554240"));
startActivity(call);
```

Intent Filters

Intent filters let an activity receive intents matching certain criteria, like a URI with a specific protocol scheme.



Cross-App Scripting

This is just like cross-site scripting in a traditional web app, where input ends up in a WebView.
Depending on what the WebView can access, this can lead to compromise of private data.

Finding these instances can be tricky, but looking for instances of the loadUrl or evaluate JavaScript methods would be a good start. Trace backwards from these calls to determine if the parameters come from an insecure source.

Report #401793 is a perfect example, where a deeplink into the application (via an intent filter) allows an attacker to load arbitrary content into the WebView.
This allowed direct access to user data.

As with XSS, it’s critical to prevent unsanitized data from being directly loaded into a WebView.
URLs should be whitelisted or very strictly filtered, and if at all possible, any data passed to evaluate JavaScript should be hardcoded.

Intent Redirection

When an intent reaches an activity, it can either be handled directly or forwarded to another target. In cases of forwarding to another activity, insufficient validation on this intent may end up triggering an internal, unprotected activity.
While this can take many forms, a good first pass is to look for intents coming from getExtras() and see where they're going.

App developers should very strictly validate incoming intents and never directly send them to startActivity(). Whenever possible, construct brand new Intents and only copy over specific, safe values. If you need to use them directly, make sure to check both the class **and** package names of the target activity, to make sure it's going somewhere safe.

Intent Broadcasting

Applications can broadcast intents to the system, to be handled by any capable application. If you see a sendBroadcast call for an intent without a specified class or component, you may be able to intercept any data that is sent out.

When protecting an application, there are two approaches to take:

- When the target is known, use setClass / setClassName / setComponent to make sure it only goes to that target
- When unknown, treat the data as public and don't send anything confidential

Unprotected Activities

Publicly exported activities can be directly triggered by other applications. This is fine if it's intended, but can break application flows if not.
You can find exported activities by looking in the manifest for those with intent filters or those with the exported="true" attribute.

Developers can protect their applications by ensuring that the only activities exported are the ones intended to be directly used from the outside.
Exporting an activity should be the last resort, not a default.

Custom Permission Typos

Android apps can provide custom permissions, which allow access to certain activities from authorized applications. But if the name of the defined permission and the permission actually used differ, all apps are authorized.

```
<permission android:name="com.myapp.permission.my_permission' android:protectionLevel="dangerous"/>

<provider android:name="com.myapp.MyProvider" android:writePermission="com.myapp.permission.mypermission"
android:enabled="true" />
```

Report #440749 is a great example of this. The application created a permission called "write_contact_permission" but used "write" instead.
Any app on the system was then able to write contacts into the vulnerable app's contact database.

To prevent this, developers should check that any permission that is defined is also used. If multiple applications are sharing a set of permissions, make sure to check it in each manifest.
Each time you refer to a permission, **double-check** that you've spelled it **exactly** like you did elsewhere.
Consider documenting the permissions and their uses somewhere else.