PermissionFlow:
Detecting permission-protected information leaks in Android applications
Overview

• Section 1: PermissionFlow
  – Android Background
  – Attacks
  – The problem
  – Solutions
  – System description
  – Results

• Section 2: Related work
Android Background (1/4)

Activities

• **Activity** = A component that
  – Provides a window
  – Enables user interaction
Intents

- **Intents = messages between Activities**
  - Intra-application
  - Inter-application (IPC)

1. `Intent i = new Intent();`
2. `i.setClassName( this, "CalleeActivity" );`
3. `startActivity(i);`
Android Background (3/4)

Intents with return values

Starting an activity expecting a return value:

1. Intent i = new Intent();
2. i.setClassName( this, "package.CalleeeActivity" );
3. startActivityForResult(i);

Returning a return value to the caller:

1. Intent intent = new Intent();
2. intent.putExtra("key", "my value");
3. this.setResult(RESULT_OK, intent);
4. finish();

Accessing the return value of the child activity:

```java
1. void onActivityResult( int request, int result, Intent data) {
2. if ((request == CREATE_REQUEST_CODE) && (result == RESULT_OK))
3. String info = intent.getStringExtra("key");
4. }}
```
Android Background (4/4) Permissions

- To access protected APIs, applications need to declare permissions.
- The same permissions are not enforced on callers.

```
<manifest package="com.android.app.myapp" sharedUid="UID">
  <uses-permission name="android.permission.VIBRATE" />
  <activity name="MyActivity">
    <intent-filter>
      <action name="com.zxing.SCAN" />
      <category name="category.DEFAULT" />
    </intent-filter>
  </activity>
</manifest>
```
Android Permissions and Intents

• Sample Android application configuration file:

```xml
<manifest package="com.android.myapp" sharedUid="UID">
  <uses-permission name="android.permission.VIBRATE" />
  <activity name="MyActivity" exported="true">
    <intent-filter>
      <action name="com.zxing.SCAN" />
      <category name="category.DEFAULT" />
    </intent-filter>
  </activity>
</manifest>
```

The presence of either of these makes the activity implicitly public (invokable by inter-application intents)
Attacks (1/2):
Colluding applications

- Example: Two colluding applications can collaborate to leak private pictures.
Attacks (2/2):
Private Activity Invocation

- Example: An attack that exposes user contacts legally obtained by application A to application B.
The problem (1/2)

• Intents are used for intra and inter-application communication.

• These two use-cases have different security implications!

• Implicitly marking activities as exported when an intent-filter is present goes against the secure-by-default principle.

• Difficult configuration increases developer confusion.
The problem (2/2)

In the absence of caller permissions enforcement, the following combinations are vulnerable:

<table>
<thead>
<tr>
<th>Activity configuration</th>
<th>Application configuration</th>
<th>Consequence (Callers accepted)</th>
<th>Risk</th>
</tr>
</thead>
<tbody>
<tr>
<td>Exported</td>
<td>Intent-filter</td>
<td>Shared UID</td>
<td></td>
</tr>
<tr>
<td>“true”</td>
<td>Present</td>
<td>don’t care</td>
<td>All</td>
</tr>
<tr>
<td>“true”</td>
<td>Absent</td>
<td>don’t care</td>
<td>All</td>
</tr>
<tr>
<td>“false”</td>
<td>Present</td>
<td>constant</td>
<td>Same developer &amp; UID</td>
</tr>
<tr>
<td>“false”</td>
<td>Absent</td>
<td>constant</td>
<td>Same developer &amp; UID</td>
</tr>
<tr>
<td>Default</td>
<td>Present</td>
<td>don’t care</td>
<td>All</td>
</tr>
<tr>
<td>Default</td>
<td>Absent</td>
<td>constant</td>
<td>Same developer &amp; UID</td>
</tr>
</tbody>
</table>
Solutions

• Android modifications:
  – Simplify configuration (no implicit export)
  – Require callers to own the permissions of the callee

• Check for the vulnerability with security tool
PermissionFlow

• Static analysis tool that detects misconfigured applications that leak permission-protected information

TelephonyManager tm = new TelephonyManager();

Intent intent = new Intent();

String imei = tm.getDeviceId();

intent.putExtra("key", imei);

this.setResult(RESULT_OK, intent);
Questions

• How to locate sources of permission-protected information?
  – Found by the Permissions Mapper

• How to identify if the information reaches `Activity.setResult`?
  – Static taint analysis

• How to ensure permissions are not enforced through configuration?
  – Application manifest analysis
Permissions Mapper

• Static analysis tool to identify which permissions are required by each API
  – Inspects library call chains
  – builds a map with
    • Keys = API functions signatures
    • Values = set of Permissions required
    • Ex (Tizen):
      Key = Tizen::Social::Addressbook::AddContact (Osp::Social::Contact&)
      Value = WRITE_ADDRESSBOOK

• Previous work:
  – Automated testing
  – Part manual process
Permissions Mapper: How it works

• Tracking of call-chains leading to APIs that enforce permissions
• Tracking of permissions strings.
Permissions Mapper: Comparison between two approaches

**Static analysis**
- ✓ Finds all permissions for Java APIs
- ✗ Does not find permissions enforced through other mechanisms (Linux users, native code, etc.)
- ✗ Cannot identify permissions as optional or alternative.
- ✓ Fully automatic

**Automated testing**
- ✗ Does not reach 100% coverage
- ✓ Finds APIs whose permissions are enforced by any mechanism
- ✓ Identifies optional or alternative permissions
- ✗ Needs manual intervention in test case generation.
Permissions Mapper results

• False negatives
  – Static analysis: no false negatives (for Java-enforced permissions APIs)
  – Automated testing: one false negative

• False positives
  – Static analysis: yes, for optional permissions
  – Automatic testing: no

• Map size (# of APIs requiring permissions)
  – Static analysis: 4361 calls
  – Automated testing: 1311 calls
Permissions Mapper in practice: Overprivileged applications

- Overprivileged applications = applications that request more permissions than they need

- Tested on 313 applications
  - Both approaches found 116 apps to be overprivileged.
  - No permission found by automated testing as not required was found as being required by static analysis.
  - 47 permissions declared used by static analysis, but unused by automated testing.
PermissionFlow Results

• Tested on 313 Android market applications
  – 56% of applications used intra-application intents

– 3 apps leak permissions-protected information
  • Adobe Photoshop Express (READ_CONTACTS)
  • Soundtracking (LOCATION)
  • Sygic GPS (CAMERA)

– 1 apps leaks private information
  • GO Locker/ GO SMS (lockscreen PIN)
Lessons learned

• Make application configuration simple

• Safe-by-default configuration (no implicit changes that affect security)

• Enforce permissions though single mechanism, enables use of static analysis for security