CSCI 445: Mobile Application Security

Lecture 9

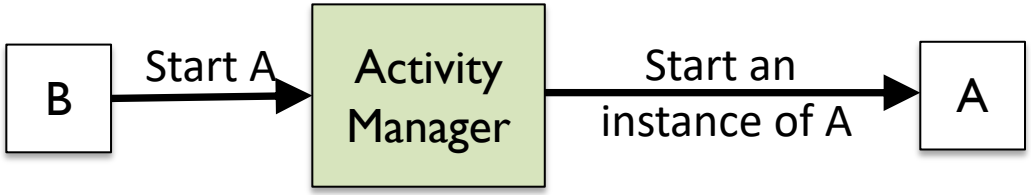
Prof. Adwait Nadkarni

Derived from Slides by William Enck

Recap:
Access control, Mobile OSes, and Intents/Intent-filters

Recall: Intents

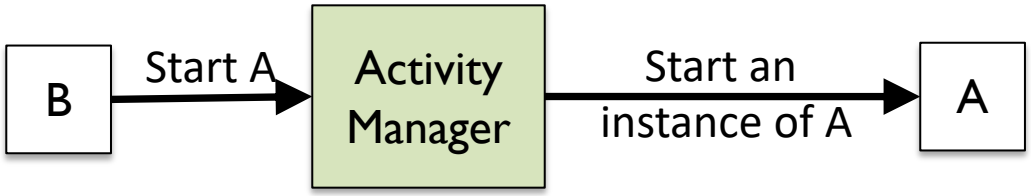
- Most common form of *Inter-component communication*



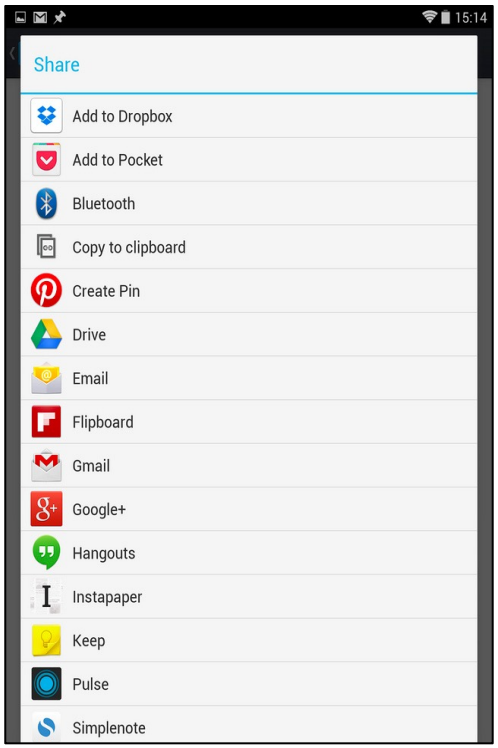
- Intents messages can be used for
  - Starting an *activity*
  - Starting a *service*
  - *Binding* to a service
- Two types: *explicit or implicit*
  - Explicit: start activity **A** from **app XYZ**"
  - Implicit: start an activity to **ACTION_VIEW** a **PDF**

Intent Filters

- Intents are an *indirect* and *asynchronous* communication mechanism.
- Intent filters describe the service provided by a component: ACTION, DATA, CATEGORY, ...
- *The system matches intents with filters*



- But, what if there is more than one match?
  - Activity: Ask the user!
  - Service: ?

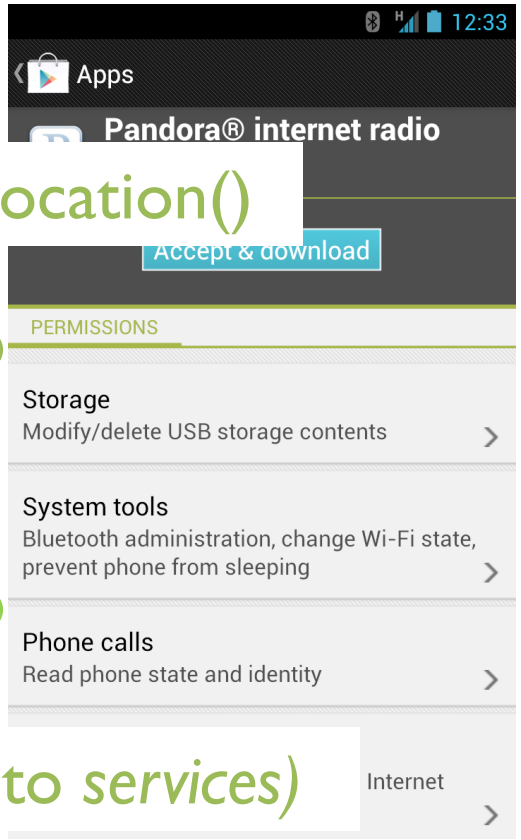


On Mobile OSes

- *Permissions define capabilities*
- For accessing objects belonging to the user/system
  - E.g., SDcard, network, phone IMEI/IMSI, contacts, calendar data, ...
- For *accessing objects belonging to other apps*:
  - E.g., Interfaces to services exposed by other apps, files/data of other apps

Call some system API; e.g., getLocation()

Use Intents(start *activities*, bind to *services*)



Assume that the permission assignment is correct.

Are we done?

Least Privilege

- Limit permissions to those required and no more
  - Restrict privilege of the process of J to prevent leaks
    - Cannot R/W O3

Does this mean we have *security?*

	O_1	O_2	O_3
J	R	RW	-
S_2	-	R	-
S_3	-	R	RW

Least Privilege

	O_1	O_2	O_3
J	R	RW	-
S_2	-	R	-
S_3	-	R	RW

A ***trojan***, or ***confused deputy*** can still append O1 to O2, which everyone can read.

Recall: An access control matrix with Least Privilege

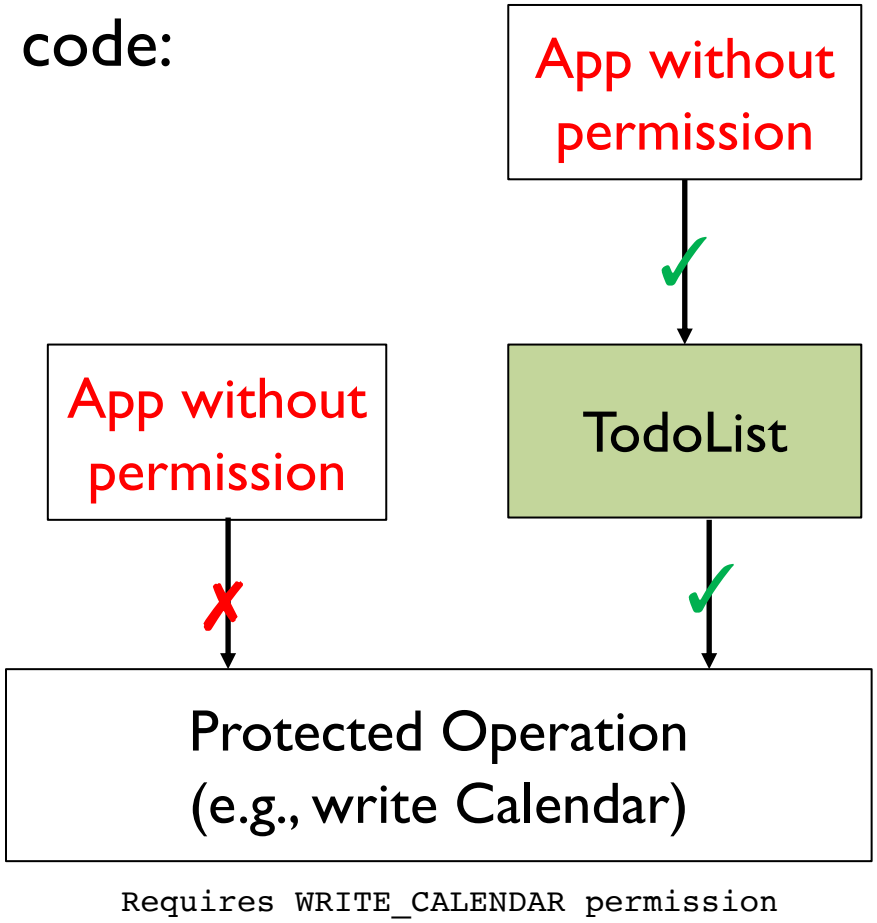
- Do we get *secrecy* if we do not trust some of J's processes?
  - *Trojan Horse*: Attacker controlled code run by J can violate secrecy.
  - *Confused Deputy*: Attacker may trick trusted code to violate integrity

	O_1	O_2	O_3
J	R	RW	-
S_2	-	R	-
S_3	-	R	-

Inter-app communication: Attacks, best-practices and defenses

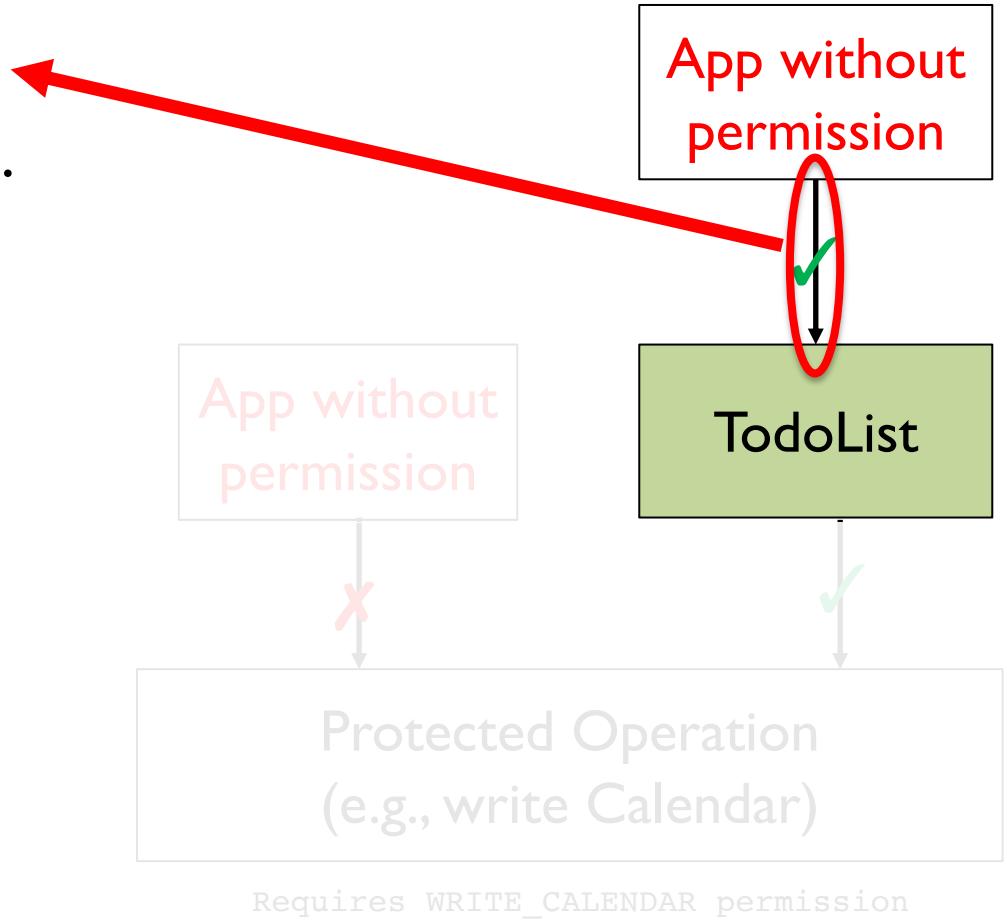
Confused Deputy

- Attacker may trick trusted code:

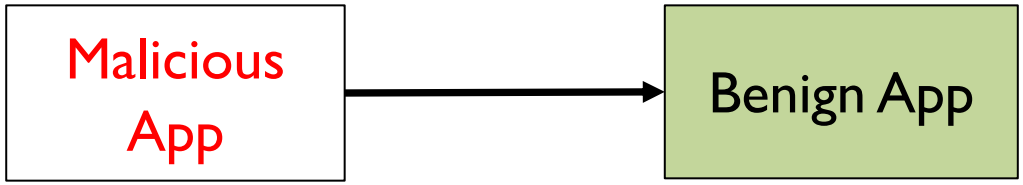


Confused Deputy

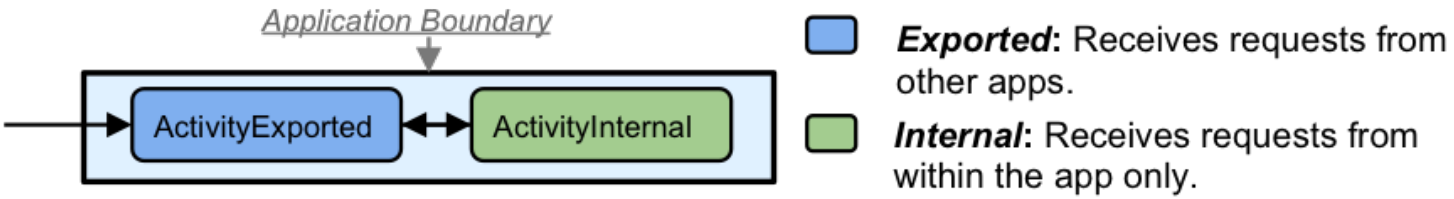
- Q: Why does this happen?
- A: Unprotected interfaces.



Receiving Intents



Internal vs Exported Components

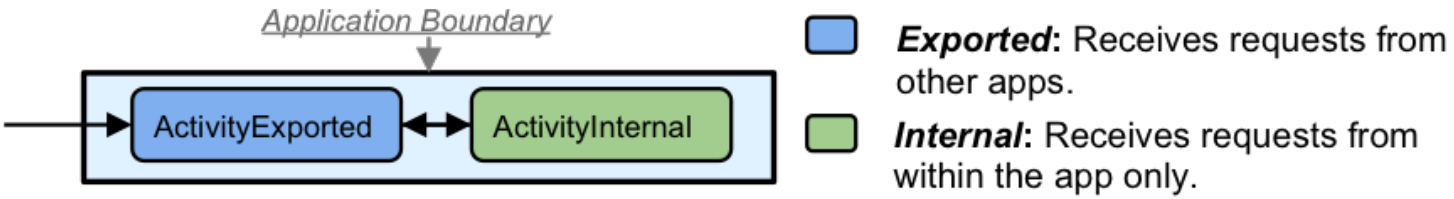


- App components can be *internal* or *exported*
  - Optional "exported" attribute in AndroidManifest.xml: "true" for exported, "false" for internal.

```
<activity android:name=".ActivityExported" android:exported="true" .../>
<activity android:name=".ActivityInternal" android:exported="false".../>
```

- *What is the default?*
  - "False"
  - CAVEAT!

Internal vs Exported Components



- App components can be *internal* or *exported*
  - Optional "exported" attribute in AndroidManifest.xml: "true" for exported, "false" for internal.

```
<activity android:name=".ActivityExported" android:exported="true" .../>
<activity android:name=".ActivityInternal" android:exported="false".../>
```

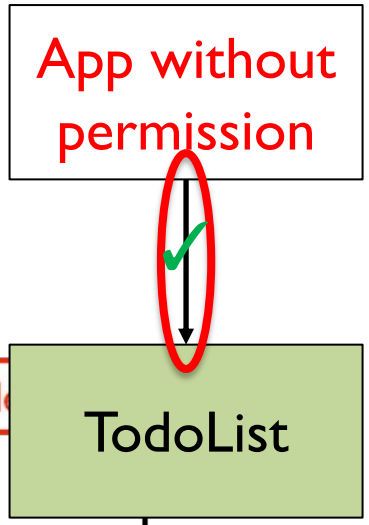
  - Default rules *export* of intent-filter defined

android:exported="true" by default!

```
<activity android:name=".ActivityInternal">
   <intent-filter>
        <action android:name="android.intent.action.SEND"/>
        <category android:name="android.intent.category.DEFAULT"/>
        <data android:mimeType="text/plain"/> </intent-filter>
</activity>
```

Unprotected Exported Components

- An exported component can be accessed by *any 3rd-party application*, even if only "useful" to the app
- If not protected, the caller can potentially:
  - Obtain confidential user or app information
  - Perform privileged actions




```
<activity android:name=".ActivityExported">
    <intent-filter>
        <action android:name="android.intent.action.SEND"/>
          <category android:name="android.intent.category.DEFAULT"/>
          <data android:mimeType="text/plain"/>
```

exported by d

Lesson 1: Set exported to false *explicitly* for internal components

Protecting Exported Components

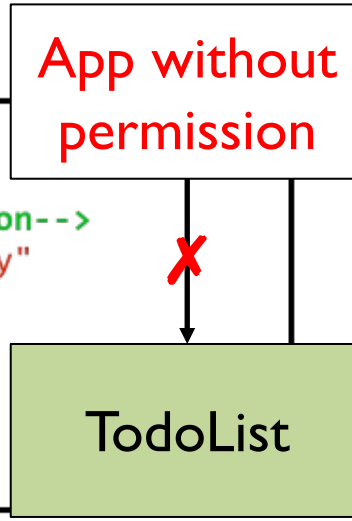
- When access needed by apps by the *same developer*, use a *signature* protection-level permission.

```
<manifest . . . >
    <!-- 1. Create signature permission-->
    <permission android:name="com.example.project.SIGNATURE_PERM"
                android:protectionLevel="signature"/>
    <application . . .>
        <!-- 2. Protect the activity with the signature permission-->
        <activity android:name="com.example.project.Activity1"
                  android:exported="true"
                  android:permission="com.example.project.SIGNATURE_PERM"
                  ... />
    </application>
</manifest>
```

Protecting Exported Components

- When access needed by apps by other *3rd-party developers*, use a *Android-defined* permission where appropriate.

```
<manifest . . . >
    <application . . .>
        <!-- 1. Protect the activity with a predefined dangerous permission-->
        <activity android:name="com.example.project.CreateContactsActivity"
            android:exported="true"
            android:permission="android.permission.WRITE_CONTACTS”
        ... />
    </application>
</manifest>
```



Requires WRITE_CALENDAR permiss

Protecting Exported Broadcast Receivers

- **Recall:** Broadcast receivers receive system-wide events (e.g., system has booted, SMS received).
- *The attacker can broadcast an intent to trick the Broadcast Receiver into believing an event occurred!*
  - i.e., broadcast intent with BROADCAST_SMS.
- Android defines "*protected broadcasts*" to mitigate (i.e., ACTIONS only the system can broadcast). Solved?
  - No! *Explicit intents* without the action! (i.e., start *Receiver A*)
- **Mitigation 1:** Use Permissions wherever possible.

```
<receiver android:name="SmsReceiver" android:permission= "android.permission.BROADCAST_SMS">
  <intent-filter>
    <action android:name="android.provider.Telephony.SMS_RECEIVED"/>
  </int
</recei
```

Lesson 2: Broadcast receivers are generally exported. Protect them with permissions!

Protecting Exported Broadcast Receivers

- **Recall:** Broadcast receivers receive system-wide events (e.g., system has booted, SMS received).
- *The attacker can broadcast an intent to trick the Broadcast Receiver into believing an event occurred!*
  - i.e., broadcast intent with BROADCAST_SMS.
- Android defines "*protected broadcasts*" to mitigate (i.e., ACTIONS only the system can broadcast). Solved?
  - No! *Explicit intents* without the action! (i.e., start *Receiver A*)
- **Mitigation 2:** Or, check the caller's identity.

```
<!-- AndroidManifest.xml -->
<receiver android:name="SystemActionReceiver" >
  <intent-filter>
    <action
      android:name="android.intent.action.BOOT_COMPLETED"
    />
  </intent-filter>
</receiver>
```

```
//In the SystemActionReceiver
public void onReceive(Context context,
                      Intent intent) {
    //Check if caller is system
    if(Binder.getCallingUid()!=1000)
       return;
    //Continue if check succeeds
        ...
}
```

Protecting Content Providers

- *Internal Content Provider*: Explicitly set the *exported* attribute to "false"
- *External Content Provider*: Protect both *read* (select) and *write* (insert, update, delete) interfaces with a permission.

```
<!-- For Content Providers, exported="true" by default for minSDKVersion and targetSDKVersion
>=16 -->
<provider android:name="com.example.project.Provider1" android:exported="false" ... />

<!-- Provider2 Stores contacts data, hence requires the same permissions. -->
<provider android:name="com.example.project.Provider2"
    android:readPermission="android.permission.READ_CONTACTS"
    android:writePermission="android.permission.WRITE_CONTACTS"
    ... />
```

Lesson 3: Protect both *read and write* interfaces of providers!

Precise provider access control

- *URI Permissions*: allow *delegation* of read/write access to specific rows/files in a Content Provider

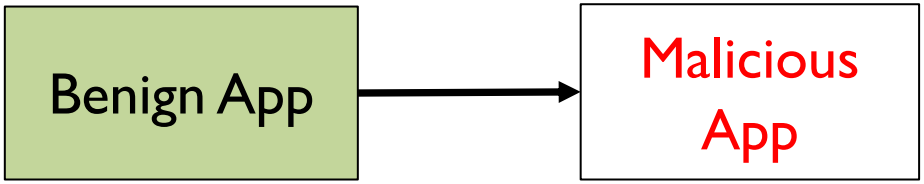
```
<!-- Two methods (use either) -->
<!-- a. Granting URI permissions through the entire provider -->
<provider android:name="com.example.project.CustomProvider"
          android:authorities="com.example.project.CustomProvider"
          android:grantUriPermission="true"
          android:readPermission= ...>

<!-- b. Granting URI permissions to a specific "public" sub-path of the provider -->
<grant-uri-permission android:pathPattern="/public/" />

</provider>
```

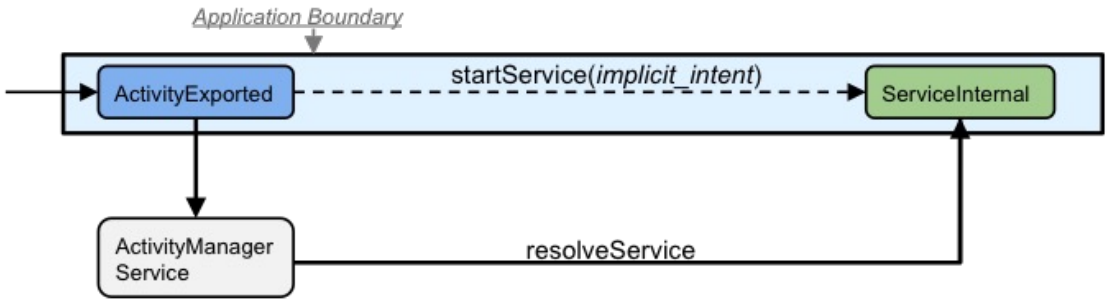
```
// Assume app has permission to read "com.example.project.CustomProvider"
// Implicit grant example
Uri uri = Uri.parse("content://com.example.project.CustomProvider/table/1");
Intent intent = new Intent(Intent.ACTION_VIEW);
intent.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
intent.setData(uri);
startActivity(intent);
// Explicit grant example
grantUriPermission("com.example.project2", uri, Intent.FLAG_GRANT_READ_URI_PERMISSION);
```

Sending Intents

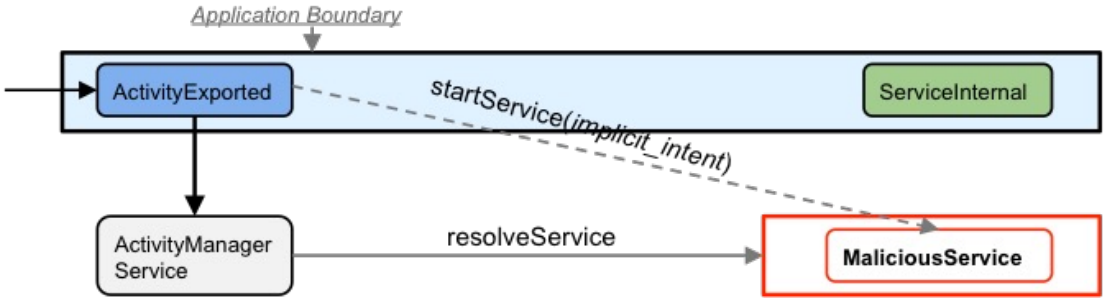


Implicit Intents, and Intent Hijacking

- Recall: an *implicit intent* is an intent message where Android's ActivityManager selects the target.

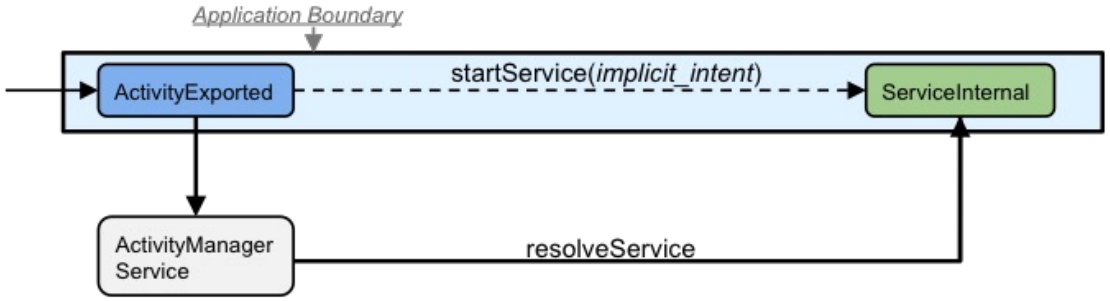


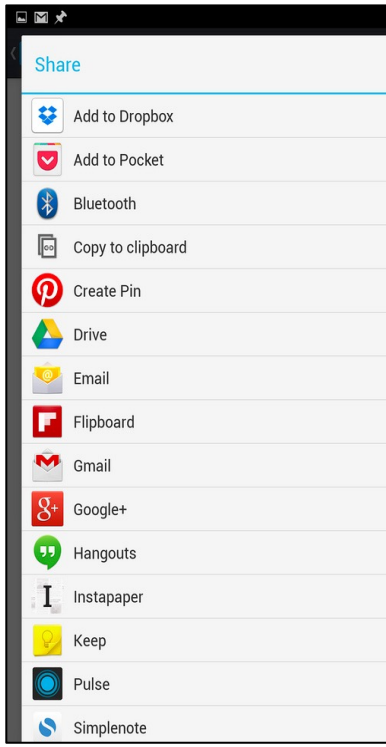
- *Intent Hijacking*: the ActivityManager is tricked into selecting a malicious target component



Implicit Intents, and Intent Hijacking

- Recall: an *implicit intent* is an intent message where Android's ActivityManager selects the target.





- But, what if there is more than one match?
  - Activity: Ask the user!
  - Service: ?
    - *Random choice*

Lesson 4: Know your defaults; especially who can receive your messages by default

Preventing Intent Hijacking

- Use *explicit intents* for communication within an app

```
<!-- AndroidManifest.xml with Activity1 and Service1-->
<activity android:name="com.example.project.Activity1" android:exported="false">
</activity>
<service android:name="com.example.project.Service1">
</service>
```

- Explicit intents specify the target component in the intent message

```
//Inside Activity1
//Explicit intent, will ONLY start Service1.
Intent intent = new Intent(this, Service1.class);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
startService(intent);
```

Limit the receivers of a Broadcast

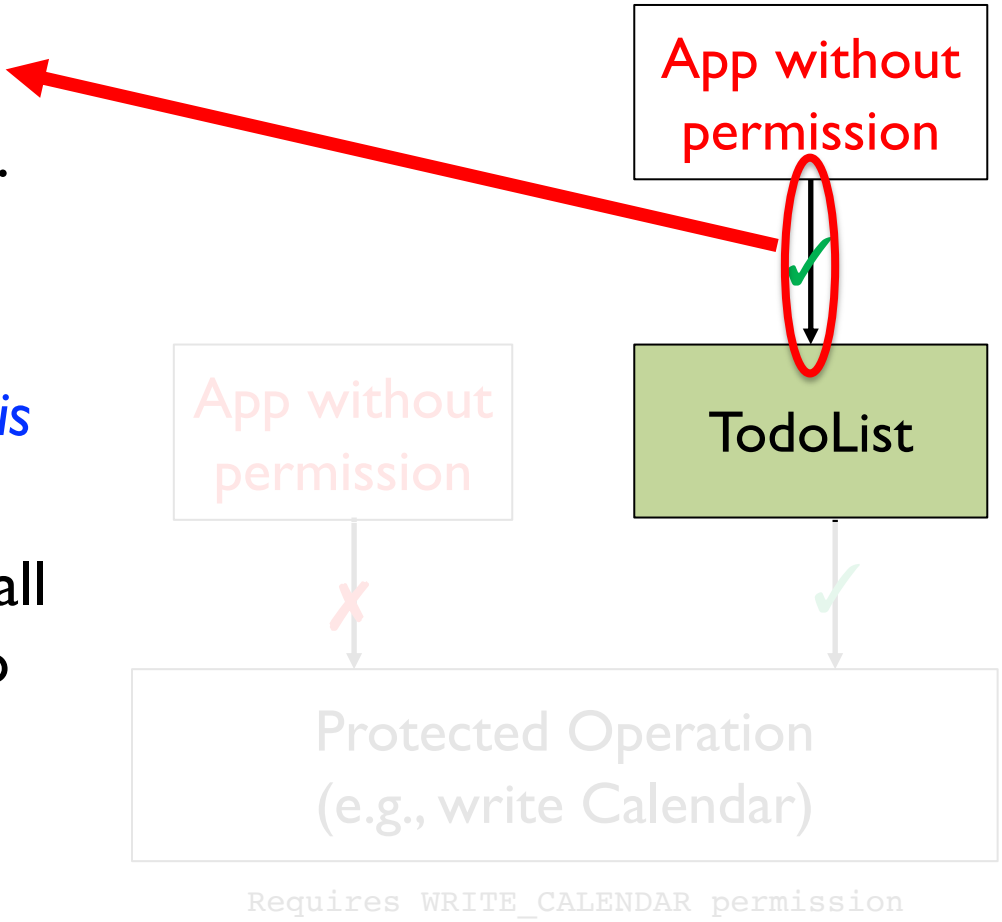
- Anyone who registers for a broadcast can receive
  - No *hijacking* necessary
- What can we use to control who receives the broadcast?
  - Permissions!

```
<!--  Declaring the permission -->
<permission android:name="com.example.project.permission.BroadcastPerm"
            android:label="broadcastPerm"
            android:protectionLevel="signature/system">
</permission>
```

```
Intent broadcast = new Intent("com.example.project.Broadcast");
//Use the API: sendBroadcast (Intent intent, String receiverPermission)
sendBroadcast(broadcast, "com.example.project.permission.BroadcastPerm");
```

Recall: Confused Deputy

- Q: Why does this happen?
- A: Unprotected interfaces.
- But, why does this *really* happen?
  - *Permission enforcement is not transitive*
  - i.e., everybody in the call chain does not need to have the permission.



Transitivity in Permissions

- *Permissions are not transitive*
  - i.e., everybody in the call chain does not need to have the permission.
- Can we add transitivity?
  - Challenges? What principle would this violate?
    - Permission bloat, i.e., overprivileged apps!