Android Framework
How to use and extend it
Lectures 7/8

- UI advanced building bricks
- Wakelocks
- Task scheduling inside UI thread and outside of it
- Manage Http connections
- Parse XML content
Lecture 9: Android Security

• Security
  ○ Threats and approaches used to enforce security
• Places where to enforce security
  ○ Layers / Distributed agents
• Android Security Model
  ○ Platform layer
  ○ Application distribution
  ○ Framework layer
Security threats \cite{1}

- **Interception**
  A malicious user could read private data (saved or streamed)

- **Interruption**
  A malicious user could crash or starve running services using some flows/bugs of them

- **Modification**
  A malicious user could alter some information (saved or streamed) without none to notice

- **Fabrication**
  A malicious user could generate fake contents
A Security policy is:

The description of the actions which system's entities can or cannot do.

We recall System's entities as:

Everyone or everything which can do an action on the system (users, Software agents etc.)
Security policies and mechanisms

How can security policies be applied?

- **Authentication**
  
  *Certify* that the user Alice is really Alice

- **Authorization**
  
  Give to Bob some *permissions* he can spend around the system.

- **Cryptography**
  
  *Scramble* the data where it is likely to have interception problems

- **Auditing**
  
  Important actions should be logged persistently
Lecture 9: Android Security

• Security
  ◦ Threats and approaches used to enforce security

• Places where to enforce security
  ◦ Layers / Distributed agents

• Android Security Model
  ◦ Platform layer
  ◦ Application distribution
  ◦ Framework layer
Security gates

Security policies must be applied in a reasonable amount of points within a system to state the system secure.
Security gates

Methods must maintain consistency of saved data all the time.
Security gates

Implement ACLs to grant only to the privileged subset of entities the permission to call potentially harmful methods.
Security gates

Implement a way to identify (authenticate) every user and grant/reject permissions to it.
Security gates into architecture

• Security gates are translated into *Interfaces* and *Protocols* inside the software architecture:
Security gates into architecture

- Security gates are translated into *Interfaces* and *Protocols* inside the software architecture.
- Every component (layer or agent) define its own security policy and adopt security mechanisms to enforce it.
- Every component makes its own choices about the *trust level* it has about other components in the system.
Lecture 9: Android Security

• Security
  ○ Threats and approaches used to enforce security

• Places where to enforce security
  ○ Layers / Distributed agents

• Android Security Model
  ○ Platform layer
  ○ Application distribution
  ○ Framework layer
Android Security Model

- Android is a complex OS where both *Layered* and *Distributed* architectures coexists locally into a device.
- Android application security start from the developer and end to the running application.
Lecture 9: Android Security

• Security
  ◦ Threats and approaches used to enforce security
• Places where to enforce security
  ◦ Layers / Distributed agents
• Android Security Model
  ◦ Platform layer
  ◦ Application distribution
  ◦ Framework layer
Platform - Android Security

Android make use of UNIX mechanisms to assure the platform security:

- Every components of a single application runs in its own process*
- Android gives a unique UID within the system to every applications at install time, private data are saved with 0XX0 permissions (0660) and owner: UID, UID

→ No ways to access other apps private data (persistent and volatile)
→ No ways to starve other applications (linux shedule)
Platform - Android Security

- Only **core** processes execute as **root** (init / Zygote ..)
  **No normal users can became root at runtime**

- System processes (system_server) run as **system** user
  *If granted by the hardware vendor, a developer can write applications with system level credentials*

- Core binaries are assigned **strict level of permissions**
  *(UID / GID / perm_mask)* so only authorized software
  can read / write / execute them

→ **No ways to normal applications to modify directly**
  the core binaries
Platform - Android Security

- Android maintains static mappings
  
  GID → hardware functionality

- Hardware functionalities are something like: *Using Camera device, Access the external storage (r/w), Access the modem device, Access the Wifi device etc..*

- All the resources bound to an hardware functionality are deployed with strict access permissions

- An application which has been granted to access the hw functionality will be part of the correct GID

  → No ways to access to hardware functionalities for not authorized applications
Platform - Android Security

At this layer:

- **Interception**
  No way to access system/other-app's resources

- **Interruption**
  No way to kill/starve system/other-app's processes

- **Modification**
  No access no modify

- **Fabrication**
  No way to place fake resources into sensible places
Lecture 9: Android Security

• Security
  ◦ Threats and approaches used to enforce security

• Places where to enforce security
  ◦ Layers / Distributed agents

• Android Security Model
  ◦ Platform layer
  ◦ Application distribution
  ◦ Framework layer
Apk - Android Security

- Android requires developers to **sign** apk files. Even development versions are signed with a default development key [2].
- To be able to publish applications to the Android Market, a developer **must use its own couple of Private/Public keys** which can be self-generated.
- **The certificate identify the developer and assures that the apk has not been modified during the deployment.**
- There is no **Certification Authority** in this process and the goodness of a certificate is user evaluated:
  
  Good applications → good developer → trustful key
Apk - Android Security

Two applications signed with the same key can share the same UID (components runs within the same process):

```xml
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.retis.first" >
    ...
</manifest>

<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.retis.second"
    android:sharedUserId="com.retis.first" >
    ...
</manifest>

Can be a number or pkg name

The second application will be able to read/write private files (preferences, databases ..) of the first one
Why sharing UID

- To deploy an **Updater** application which can reuse the old data.
- To share common private data without the use of a content provider (harmful)
- To gain privileges
  - If the user id to share is `android.uid.system` the application will be able to access directly system peripherals

→ **Remember**: You must sign with the same key!
Apk - Android Security

At this layer:

- **Interception**
  
  An apk can be intercepted.. but we want to spread it

- **Interruption**
  
  Android market gives valid data.

- **Modification**
  
  Apks cannot be modified without a re-sign. Not possible if installing from Android Market

- **Fabrication**
  
  Not possible if installing from Android Market
Lecture 9: Android Security

• Security
  ◦ Threats and approaches used to enforce security
• Places where to enforce security
  ◦ Layers / Distributed agents
• Android Security Model
  ◦ Platform layer
  ◦ Application distribution
  ◦ Framework layer

At this layer security is enforced using the following mechanisms:

- Component privacy
- Validation
- Fine grained Filters
- Permissions
Component privacy

The strongest way to secure the access to a component is making this **private**:

```xml
<activity android:name=".HiddenActivity"
     android:exported="false" >
    <intent-filter>
       <action android:name="com.retis.CLEAR_DB" />
    </intent-filter>
</activity>
```

Only the owner application is be able to access it and we should trust ourself.......!
Validate your input

This code is legal even if MyReceiver is not listening to FAKE_ACTION:

Intent i = new Intent("my.big.FAKE_ACTION");

i.setComponent(new ComponentName("com.retis.pkgname",
  "com.retis.problematic.MyReceiver"));

sendBroadcast(i);

Than think about the robustness of all your onReceive() methods....
Refine what you are listening on

Refine your intent filter with categories and data desc to be sure you are handling what you are really interested on

<activity android:name="BrowserActivity"
    android:label="@string/application_name"
    android:launchMode="singleTask"
    android:alwaysRetainTaskState="true"
    android:configChanges="orientation|keyboardHidden"
    android:theme="@style/BrowserTheme">
<!-- For these schemes were not particular MIME type has been supplied, we are a good candidate. -->
<intent-filter>
    <action android:name="android.intent.action.VIEW" />
    <category android:name="android.intent.category.DEFAULT" />
    <category android:name="android.intent.category.BROWSABLE" />
    <data android:scheme="http" />
    <data android:scheme="https" />
    <data android:scheme="about" />
    <data android:scheme="javascript" />
</intent-filter>
Permissions: classification

Two class of permissions:

**System level permission:**
if granted to an application, this will be part to the associated GID and will have access to the associated hardware/system-level functionality

**Manifest defined permission:**
Every application can define its set of permissions which will be used to secure actions
Permissions: protection level

Every permissions define a protection level as:

<table>
<thead>
<tr>
<th>Level</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal</td>
<td>Permissions for application's features whose consequences are minor</td>
</tr>
<tr>
<td></td>
<td>User can review but may not be explicitly warned</td>
</tr>
<tr>
<td>Dangerous</td>
<td>Permissions for harmful actions (for the system or the user pocket)</td>
</tr>
<tr>
<td></td>
<td>User is warned about granting these permissions</td>
</tr>
<tr>
<td>Signature</td>
<td>Permissions granted only to applications signed with the same key as the defining application</td>
</tr>
</tbody>
</table>
Permissions: user is prompted

---

**GO Launcher EX**
GO LAUNCHER DEV TEAM

Accept & download

---

**PERMISSIONS**

**Storage**
Modify/delete USB storage contents

**System tools**
Make application always run, modify global system settings, retrieve running applications

**Services that cost you money**
Directly call phone numbers
Permissions:

Inside your AndroidManifest.xml:

```xml
<permission android:name="com.myapp.permission.WRITE_ME"
    android:label="@string/permlab_write_me"
    android:description="@string/permdesc_write_me"
    android:protectionLevel="dangerous" />
```

Inside the requesting AndroidManifest.xml:

```xml
<uses-permission android:name="com.myapp.permission.WRITE_ME"/>
```
Require permissions

We can restrict the access to all exported Android components (Activity/Services/BroadcastReceivers/ContentProviders) with:

```
<service
    android:name=".Sender"
    android:permission="com.app.permission.SEND_DATA">
    <intent-filter>
        <action android:name="com.app.SEND_MESSAGE" />
    </intent-filter>
</service>
```

Applications which does not hold that permission will not be able to communicate with *Sender* service
Safely send Broadcasts

When sending a Broadcast we can couple it with a permission string which listeners must hold to receive the intent:

```java
Intent i = new Intent("com.app.STATE_CHANGE");
sendBroadcast(i, "com.app.permission.RECEIVE_STATE_CHANGE");
```

Make use of it otherwise malicious applications could listen freely at your own actions!
Fine grained for ContentProviders

ContentProviders can request different permissions for read/write operations using:

```xml
android:readPermission=""
android:writePermission=""
```

Make sure your ContentProvider implementation is aware of **SQL injection** threat

Never trust String arguments enough to place them directly into the WHERE statement of your database query.
Intent reflection

This happen when an application A (with a restricted set of permissions) drive an application B (owner of a larger set of permissions) to fire an Intent which A cannot fire.

PendingIntent is the way to solve this problem:
The PendingIntent object generated by the application A and fired by application B will be executed within the application A context → with app A subset of permissions.

Real cases: Alarms!
Framework - Android Security

• **Interception**
  Enforce permissions when sending broadcasts

• **Interruption**
  Validate the input carefully, use permissions and fine grained filters to filter out not useful intents

• **Modification**
  The communication channel is considered secure while the device is not rooted

• **Fabrication**
  It is possible to generate fake broadcasts → validate everything!
Links of interest

[1] Tanenbaum, Distributed Systems