Intro to Securing Android Applications

William Enck

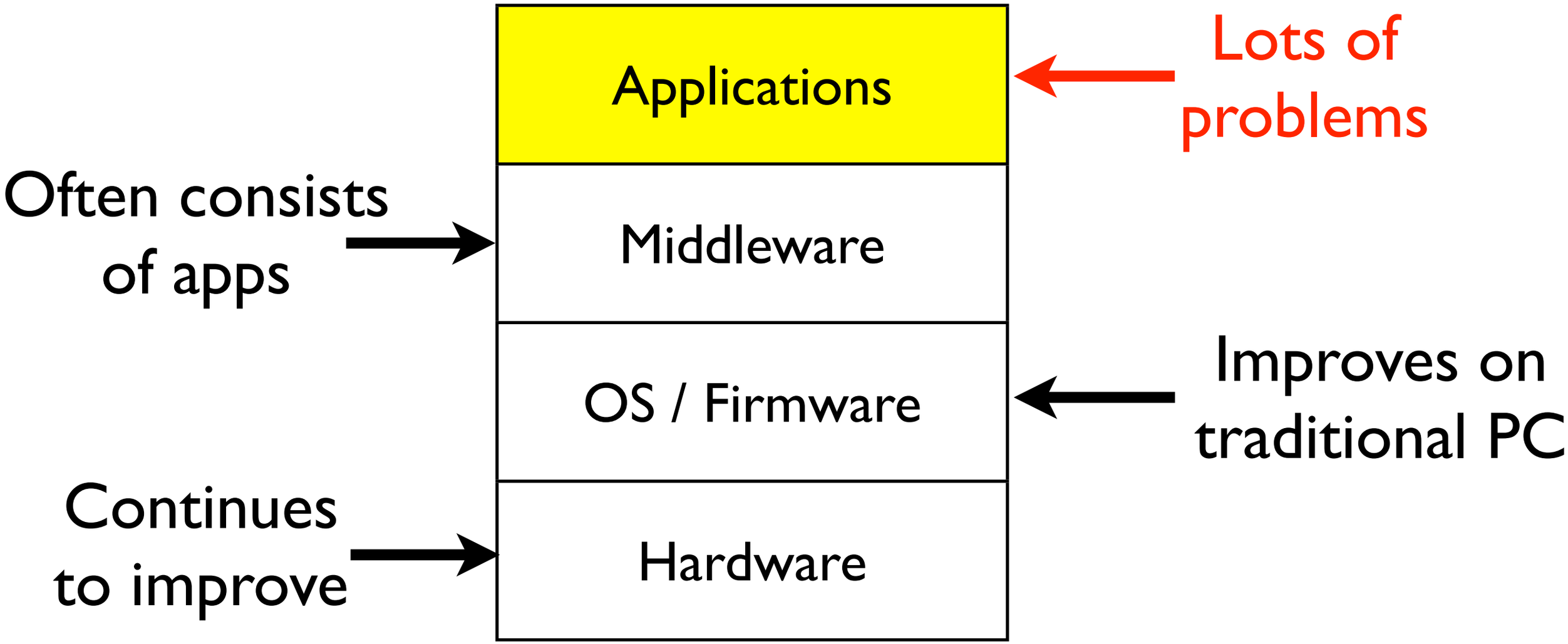
ISSA B2B - December 2014

The new “modern” OS

Mac OS

Apple

Security at Different Layers

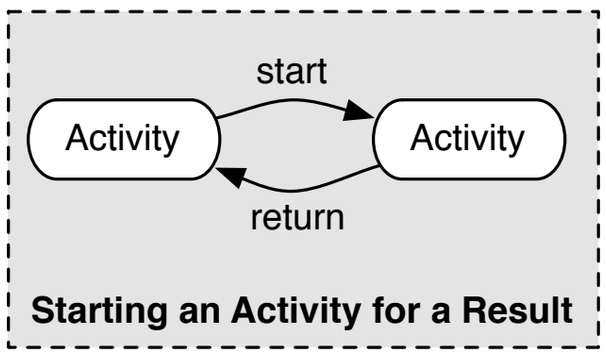


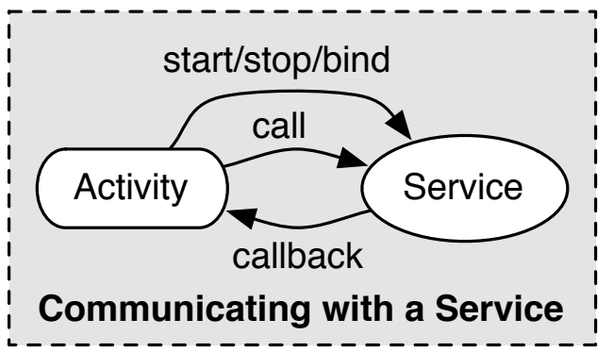
“There’s an app for that”

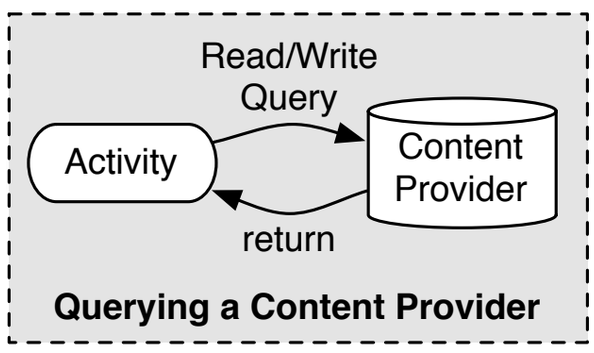
- Modern smartphone platforms revolutionized computing by creating *economies of software*.
- Software development is vastly simplified by feature rich APIs on both the device and cloud.
- Result:
  - Millions of apps, but many low quality
  - Developers looking to “make a buck”
  - Feature rich APIs easy to abuse
  - Lots of security vulnerabilities in apps

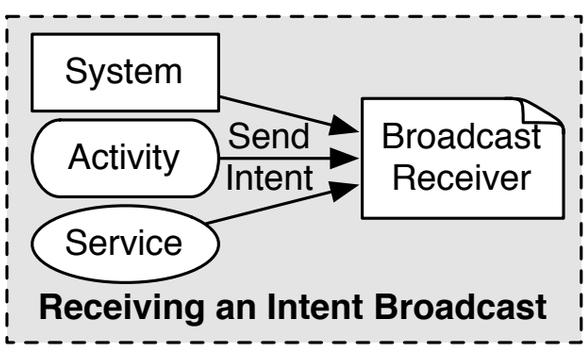
Case Study: Android

- An Android device is a collection of applications
- Each application is a collection of *components*
  - *Activity*: comprises the UI
  - *Service*: a daemon
  - *Content Provider*: a relational DB + file sharing
  - *Broadcast Receiver*: receives events









Inter Component Communication (ICC)

- ICC is built on a basic RPC primitive provided by Binder
- Services define RPCs directly (the core of the Android API is a collection of service components providing RPCs)
- ICC primarily *intent messages* specified for a component type
  - Can be automatically resolved by system (called "implicit intents") similar to MIME types for opening files on PC
  - Primarily: *action* and *data*
  - Secondarily: *category*, *type*, *component*, and *extras*.
- Content Providers have standardized RPCs
  - Equivalent to SQL select, insert, update, delete
  - Operate on URIs: `content://<authority>/<table>/[<id>]`
  - Can share both tabular content as well as files

Permissions

- Applications run as different Linux UIDs
  - The low-level Android OS is protected by Linux file permissions and SELinux policy (on newer versions of Android)
- The middleware ICC is protected by *permissions*
  - A permission is just a text string that gains semantics based on where it is used (typically, `android.permission.<name>`)
  - Android defines many permissions for protecting resources and sensitive interfaces (e.g., `android.permission.WRITE_CONTACTS`).
  - 3rd-party app developers can define additional permissions
- Four basic permission *protection-levels*:
  - *normal*, *dangerous*, *signature*, and *signatureOrSystem*

Manifest File

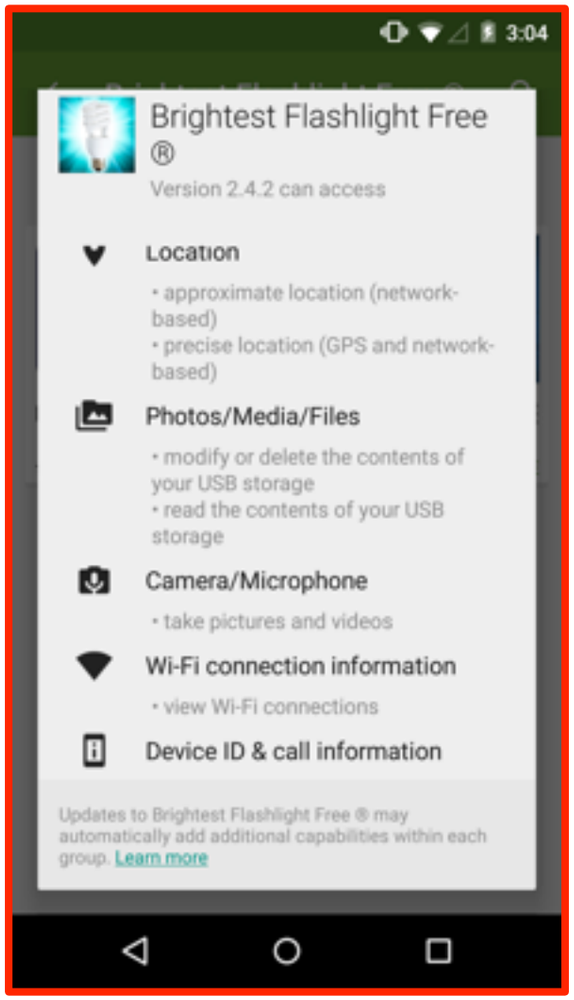
```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
      package="org.siislab.tutorial.friendtracker"
      android:versionCode="1"
      android:versionName="1.0.0">
    <application android:icon="@drawable/icon" android:label="@string/app_name">
        <activity android:name=".FriendTrackerControl"
                  android:label="@string/app_name">
            <intent-filter>
                <action android:name="android.intent.action.MAIN" />
                <category android:name="android.intent.category.LAUNCHER" />
            </intent-filter>
        </activity>
        <provider android:authorities="friends"
              android:name="FriendProvider"
              android:writePermission="org.siislab.tutorial.permission.WRITE_FRIENDS"
              android:readPermission="org.siislab.tutorial.permission.READ_FRIENDS">
        </provider>
        <service android:name="FriendTracker" android:process=":remote"
              android:permission="org.siislab.tutorial.permission.FRIEND_SERVICE">
        </service>
        <receiver android:name="BootReceiver">
            <intent-filter>
                <action android:name="android.intent.action.BOOT_COMPLETED"></action>
            </intent-filter>
        </receiver>
    </application>

    <!--  Define Permissions -->
    <permission android:name="org.siislab.tutorial.permission.READ_FRIENDS"></permission>
    <permission android:name="org.siislab.tutorial.permission.WRITE_FRIENDS"></permission>
    <permission android:name="org.siislab.tutorial.permission.FRIEND_SERVICE"></permission>

    <!--  Uses Permissions -->
    <uses-permission android:name="org.siislab.tutorial.permission.READ_FRIENDS"></uses-permission>
    <uses-permission android:name="org.siislab.tutorial.permission.WRITE_FRIENDS"></uses-permission>
    <uses-permission android:name="org.siislab.tutorial.permission.FRIEND_SERVICE"></uses-permission>

    <uses-permission android:name="android.permission.RECEIVE_BOOT_COMPLETED"></uses-permission>
    <uses-permission android:name="android.permission.READ_CONTACTS"></uses-permission>
    <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"></uses-permission>
</manifest>
```

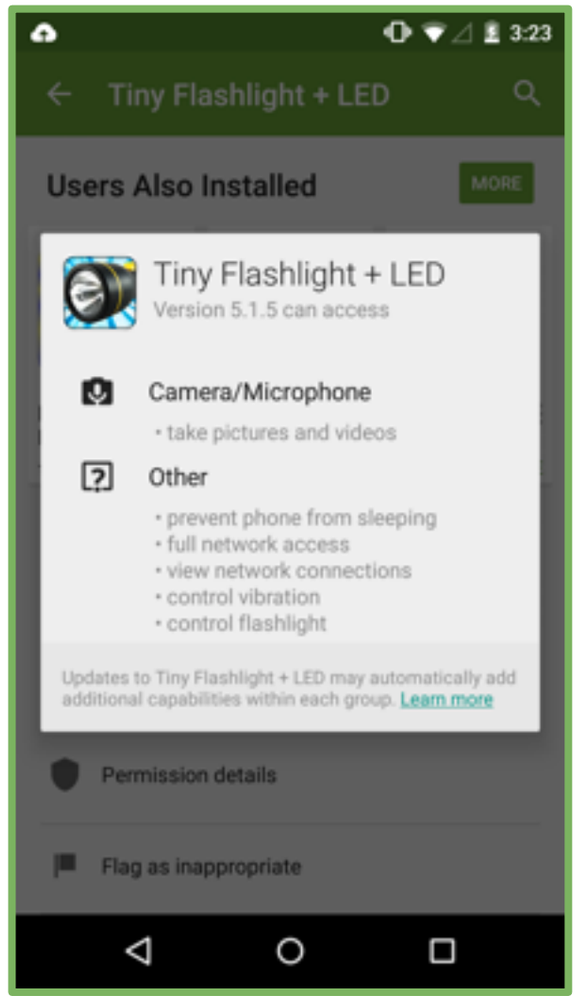
Outline

- Protecting Users
- Protecting App Interfaces
- Protecting Data at Rest
- Protecting Data in Transit

Least Privilege

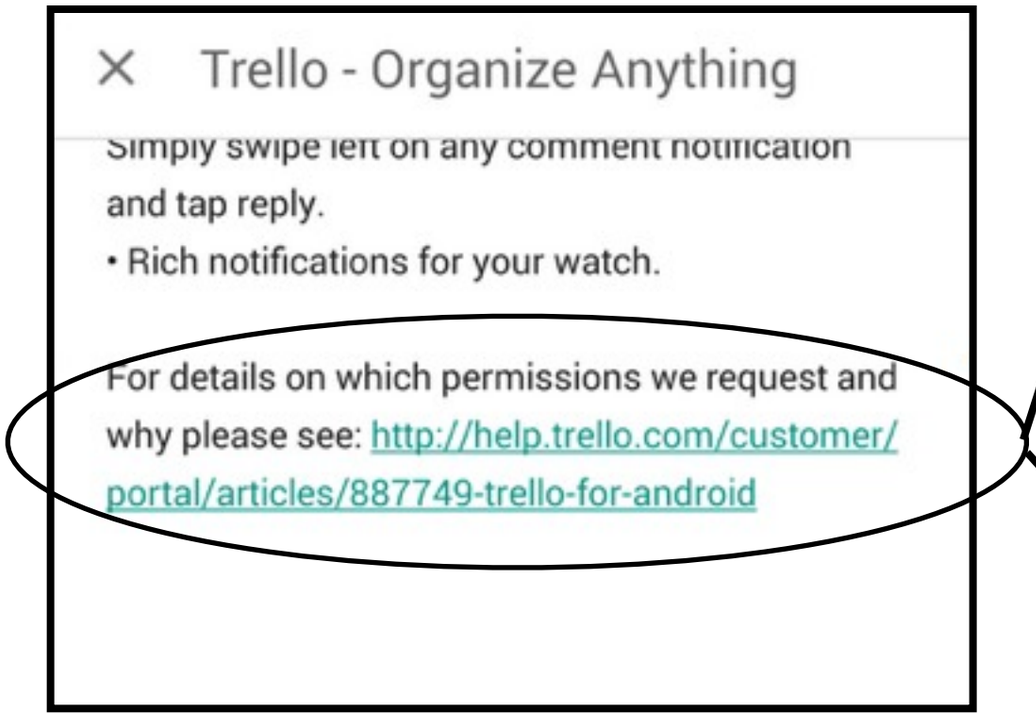
- Only request permissions for functionality *required* by your application.
  - e.g., if need approximate location, use COARSE_LOCATION

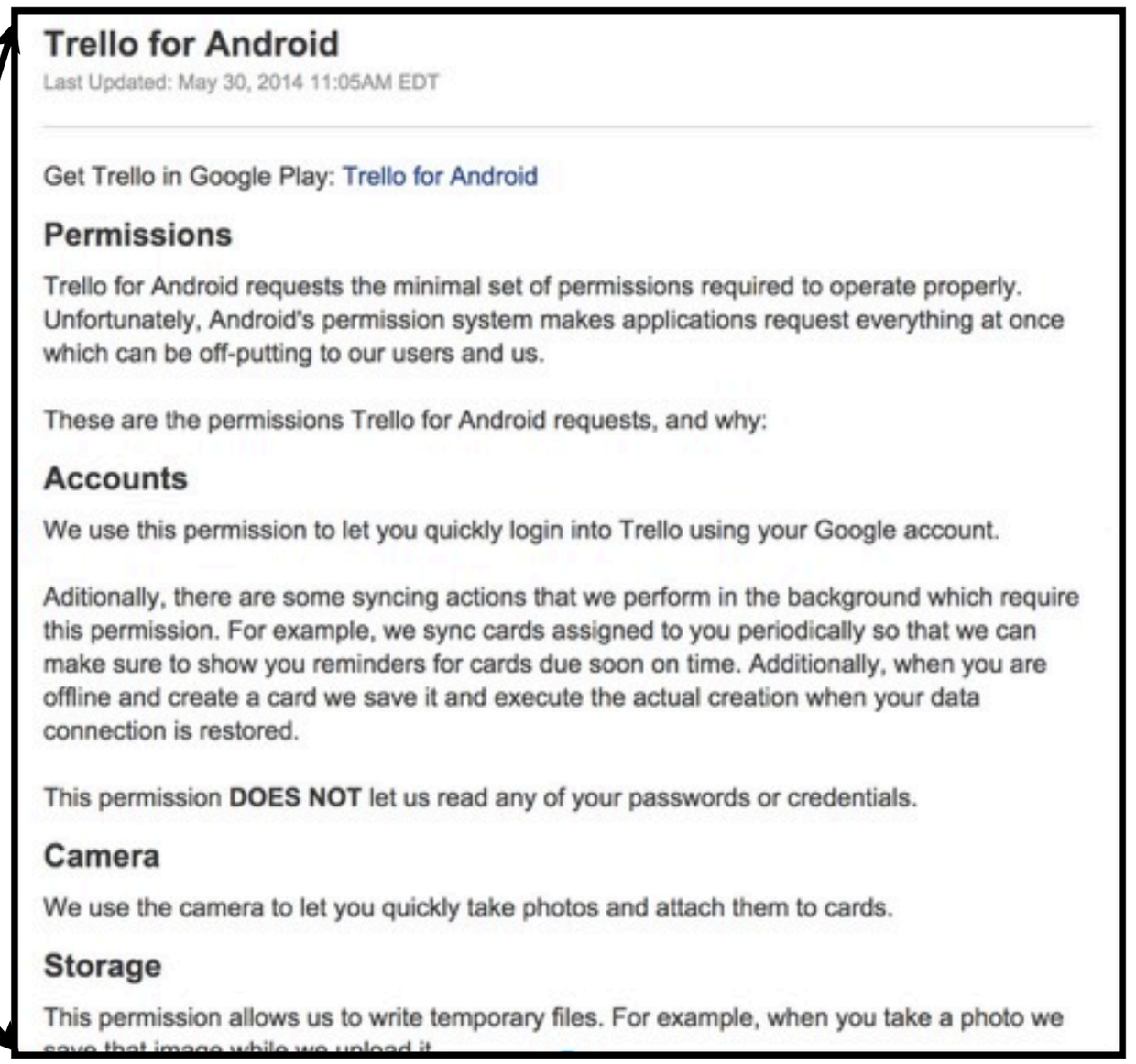




Describe Permission Use

- Build User Confidence
- Helps you reconsider the permissions requested.





Involve the User

- Often a non-permission option using a "trusted UI"
- Example: insert a contact using an *intent message* instead of requiring the *_CONTACTS permissions

```
// Creates a new Intent to insert a contact
Intent intent = new Intent(Intents.Insert.ACTION);
// Sets the MIME type to match the Contacts Provider
intent.setType(ContactsContract.RawContacts.CONTENT_TYPE);
// Inserts a phone number
intent.putExtra(Intents.Insert.PHONE, mPhoneNumber.getText());
// Inserts an email address
intent.putExtra(Intents.Insert.EMAIL, mEmailAddress.getText());
// Sends the Intent
startActivity(intent);
```

Avoid fixed device identifiers

- Device identifiers such as the IMEI are *persistent tracking cookies* and lead to privacy failure
- Alternatives include:
  - Play Store statistics (why do it yourself?)
  - ANDROID_ID - requires OEM support, still some privacy concern

```
import android.provider.Settings.Secure;
...
private String android_id = Secure.getString(getContext().getContentResolver(), Secure.ANDROID_ID);
```

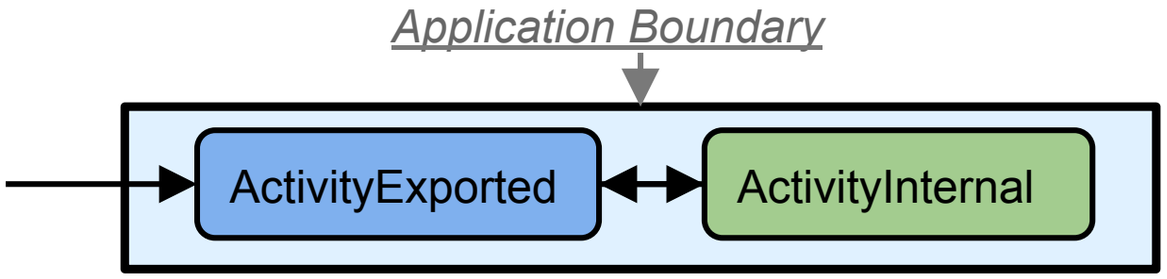
  - UUID (track the installation, not the device)

```
import java.util.UUID;
...
String id = UUID.randomUUID().toString();
```

Outline

- Protecting Users
- Protecting App Interfaces
- Protecting Data at Rest
- Protecting Data in Transit

Internal vs. Exported Components



Exported: Receives requests from other apps.

Internal: Receives requests from within the app only.

- 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

exported by default!

```
<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"/>
    </intent-filter>
</activity>
```

Protecting Exported Components (1)

- 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>
```

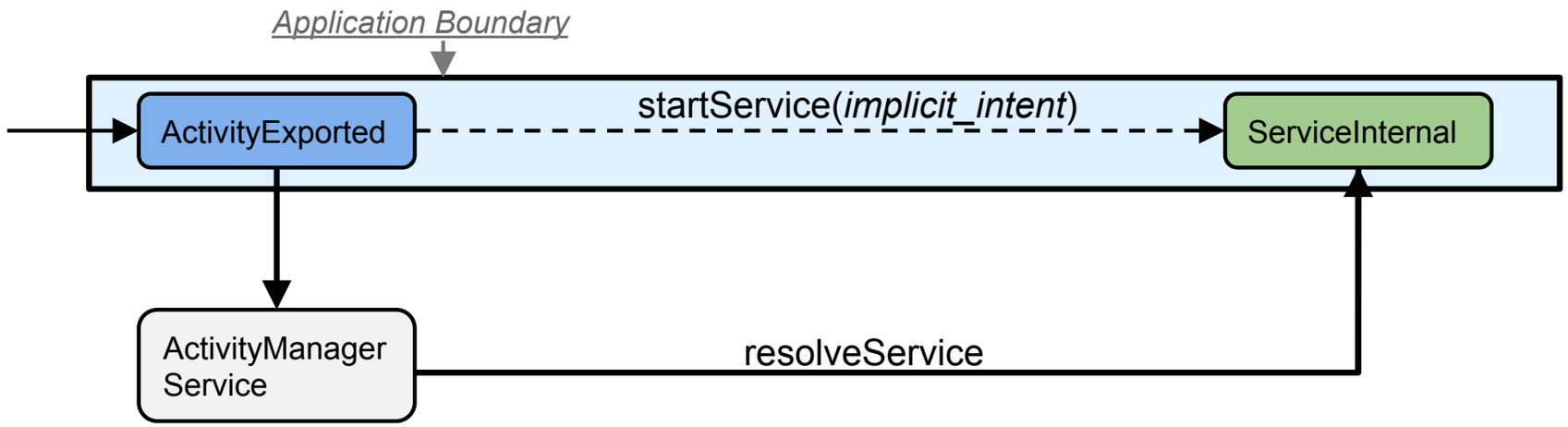
Protecting Exported Components (2)

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

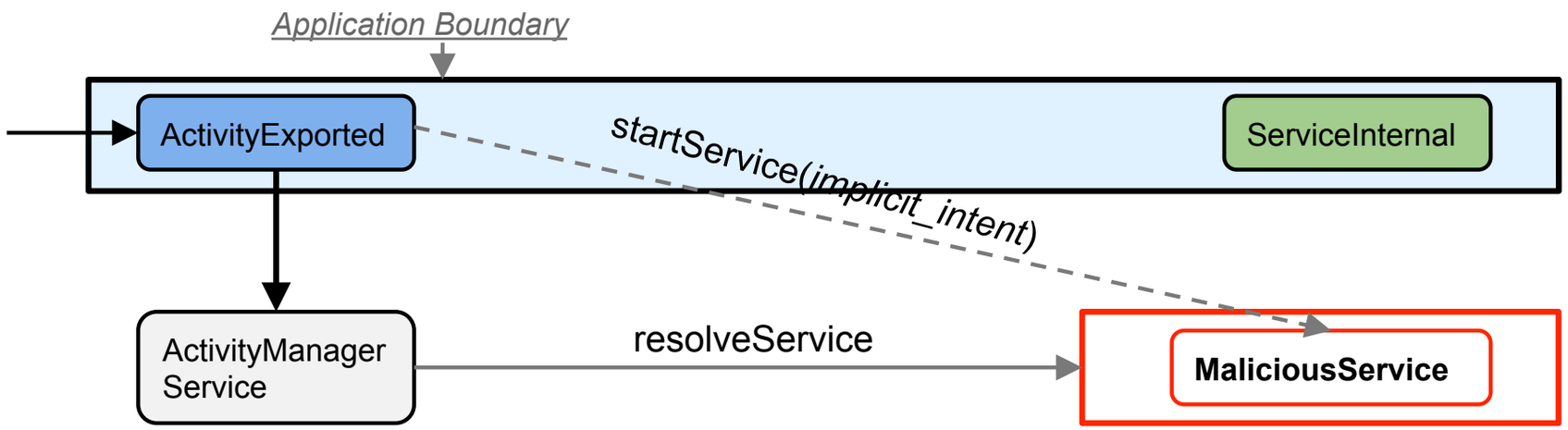
```
<manifest . . . >
    <application . . .>
        <!-- 1. Protect the activity with a predefined dangerous permission-->
        <activity android:name="com.example.project.CreateContactsActivity"
             android:exported="true"
             android:permission="android.permission.WRITE_CONTACTS”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

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



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);
```

Protect Exported Broadcast Receivers

- Attacker can broadcast an intent with action to trick Broadcast Receiver into believing an event occurred
  - Android defines "protected broadcasts" to mitigate, but be careful (e.g., explicit intent without the action)
- Use permissions where possible

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

- Check caller's identity otherwise

```
<!-- 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
        ...
}
```

Limit the Receivers of a Broadcast

- Permissions can also be used to protect which Broadcast Receivers can receive a broadcast

```
<!--  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");
```

Protecting Content Providers (1)

- *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"
    ... />
```

Protecting Content Providers (2)

- *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);
```

More info: http://thinkandroid.wordpress.com/2012/08/07/granting-content-provider-uri-permissions/

Protecting WebViews

- The *WebView* class provides a UI widget that renders HTML and JavaScript content
- Be careful to prevent cross-site scripting
  - If the app does not use JavaScript, do not call `setJavaScriptEnabled()`
  - Only expose `addJavaScriptInterface()` to JavaScript contained *inside* the APK
  - If the WebView caches private data, call `clearCache()` periodically.

Outline

- Protecting Users
- Protecting App Interfaces
- Protecting Data at Rest
- Protecting Data in Transit

Storing Sensitive Data (1)

- Applications often require the use of sensitive data such as user credentials
- *Option 1*: Do not store it on the device

Storing Sensitive Data (2)

- Applications often require the use of sensitive data such as user credentials
- *Option 2*: store in app's private directory `/data/data/<app_packagename>`

```
//API to read/write files in the PRIVATE path, i.e.,/data/data/<app_packagename>/files/
FileInputStream fis = openFileInput("input.txt");

//Note: NEVER use MODE_WORLD_READABLE/WRITABLE unless there is no other option.
FileOutputStream fos = openFileOutput("output.txt", MODE_PRIVATE);
int ch;
while((ch = fis.read())!=-1){
      fos.write(ch);
}
```

Other Useful APIs:
`getFilesDir(), getDir(), deleteFile(), fileList()`

Storing Sensitive Data (3)

- Applications often require the use of sensitive data such as user credentials
- *Option 3*: Encrypt it!
  - Use the *Android KeyStore* for key generation and storage

```
KeyPair generateKeys() throws Exception {
    Calendar cal = Calendar.getInstance();
    Date now = cal.getTime();
    cal.add(Calendar.YEAR, 1);
    Date end = cal.getTime();

    KeyPairGenerator kpg = KeyPairGenerator.getInstance("RSA", "AndroidKeyStore");
    kpg.initialize(new KeyPairGeneratorSpec.Builder(context).setAlias(alias)
                        .setStartDate(now).setEndDate(end)
                        .setSerialNumber(BigInteger.valueOf(1))
                        .setSubject(new X500Principal("CN=SampleCN")).build());

    return kpg.generateKeyPair();
}
```

Prevent SQL Injection

- Use *parameterized SQL* methods for the implementation of query, insert, etc.

```
// Partial Implementation of the Content Provider's query method.
public Cursor query(Uri uri, String[] projection, String selection, String[] selectionArgs) {
    // Query the underlying database using the SQLiteDatabase query method,
    //Instead of calling rawQuery(String sqlQuery, String[] selectionArgs);
    SQLiteDatabase db = dbHelper.getWritableDatabase();
    SQLiteQueryBuilder qb = new SQLiteQueryBuilder();
    return qb.query(db, projection, selection, selectionArgs, null, null, orderBy);
}
```

Be careful what you log

- Log only non-sensitive information
- Ideally, never log user data
- Logs can be read
  - By system apps on a non-rooted device
  - By other 3rd-party apps on a rooted device
  - By the user via *adb logcat*

Outline

- Protecting Users
- Protecting App Interfaces
- Protecting Data at Rest
- Protecting Data in Transit

Using SSL Correctly

- Basic approach: *URLConnection* should suit the majority of needs (don't forget the "https://").

```
URL url = new URL("https://wikipedia.org");
URLConnection urlConnection = url.openConnection();
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Custom CAs (1)

- Custom CAs can be added at
  - *Device level*: use if only needed during development
  - *Application level*: allow an app to *pin* to a specific CA
    - *Step 1*: Load the custom CA and create custom KeyStore

```
// Load CAs from an InputStream
CertificateFactory cf = CertificateFactory.getInstance("X.509");
InputStream caInput = new BufferedInputStream(new FileInputStream("load-der.crt"));
Certificate customCA;
try {
    customCA = cf.generateCertificate(caInput);
} catch {caInput.close();}

// Create a KeyStore containing our trusted CAs
KeyStore keyStore = KeyStore.getInstance(KeyStore.getDefaultType());
keyStore.load(null, null);
keyStore.setCertificateEntry("ca", customCA);
```

Custom CAs (2)

- Custom CAs can *pin* an app to a specific CA
  - *Step 2*: Create a TrustManager that trusts the CAs in the new KeyStore

```
// Create a TrustManager that trusts the CAs in keyStore
String tmfAlgorithm = TrustManagerFactory.getDefaultAlgorithm();
TrustManagerFactory tmf = TrustManagerFactory.getInstance(tmfAlgorithm);
tmf.init(keyStore);
```

  - *Step 3*: Create the SSLContext

```
// Create an SSLContext that uses our TrustManager
SSLContext customSSLcontext = SSLContext.getInstance("TLS");
context.init(null, tmf.getTrustManagers(), null);
```

  - *Step 4*: Use the SocketFactory from the custom SSLContext

```
// Tell the URLConnection to use a SocketFactory from our SSLContext
URL url = new URL("https://www.example.com");
HttpsURLConnection urlConnection = (HttpsURLConnection)url.openConnection();
urlConnection.setSSLSocketFactory(customSSLcontext.getSocketFactory());
InputStream in = urlConnection.getInputStream();
copyInputStreamToOutputStream(in, System.out);
```

Nogotofail

- Google has created nogotofail as a network-level tool to identify TLS/SSL related security issues
- Tests for
  - Common SSL certificate verification issues
  - HTTPS and TLS/SSL library bugs
  - cleartext issues, etc.

Google

- https://github.com/google/nogotofail

Summary

- Modern OSes like Android make it easy to develop feature rich applications that function correctly
- Not so easy to ensure they are secure
- Lots of pieces to keep in mind
  - Protecting the user
  - Protecting app interfaces
  - Protecting data at rest
  - Protecting data in transit

Questions?

William Enck
Assistant Professor
Department of Computer Science
North Carolina State University
http://www.enck.org
enck@cs.ncsu.edu

Slides available at http://www.enck.org/teaching.html