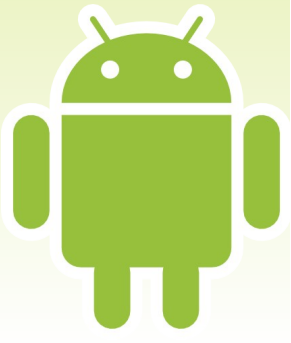
Permissions, Front to Back

Permission Theory

- Apps Require Permissions
  - To access protected components
- Apps Request Permissions
  - To access protected components or use protected framework methods
- User Prompted At Install Time
  - Exception: installation via **adb**
  - Told what permissions the app requests
  - Continue with install or abandon install



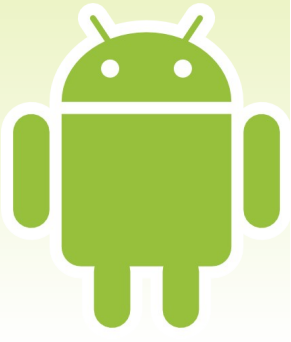
Requesting Permissions

- Step #1: Find the Permission You Need
  - Many Android-supplied permissions defined in Android documentation “List of Permissions”
  - Permissions for third-party applications hopefully documented by them
  - You need the fully-qualified name
    - E.g., `android.permission.INTERNET`
- Step #2: Add <uses-permission> to Manifest
  - `android:name=“name.of.desired.PERMISSION”`

```
<uses-permission android:name="android.permission.INTERNET"/>
```

Platform Permissions

- Defined by Framework
  - Seen in framework's manifest
  - Always exist
    - ...with variations based on API level
- `android.permission.X`
  - If you see a third party trying to define their own `android.permission.X` permission, smack 'em



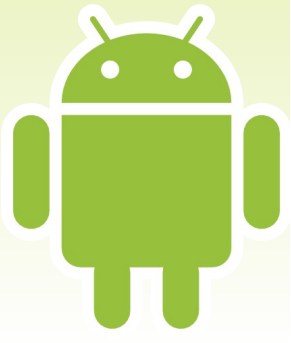
```
<permission android:name="android.permission.PROCESS_OUTGOING_CALLS"
    android:permissionGroup="android.permission-group.PHONE_CALLS"
    android:protectionLevel="dangerous"
    android:label="@string/permlab_processOutgoingCalls"
    android:description="@string/permdesc_processOutgoingCalls" />

<!-- Allows modification of the telephony state - power on, mmi, etc.
     Does not include placing calls.
     <p>Not for use by third-party applications. -->
<permission android:name="android.permission.MODIFY_PHONE_STATE"
    android:permissionGroup="android.permission-group.PHONE_CALLS"
    android:protectionLevel="signature|system"
    android:label="@string/permlab_modifyPhoneState"
    android:description="@string/permdesc_modifyPhoneState" />
```

Protection Levels

- Normal
- Dangerous
  - Presented to user higher in list than normal
- Signature
  - Consumer and defender must have matching signing keys
- System
  - Consumer must be installed on system partition

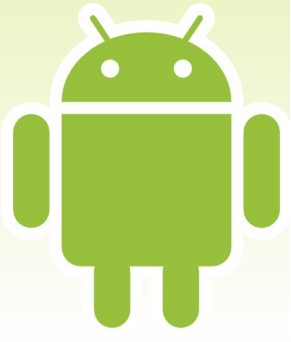
Users and Permissions, Part One

- Requested Permissions Inhibit Adoption
  - Prospective users may not like a permission, individually or in conjunction with others
    - Example: `READ_CONTACTS` and `INTERNET`
  - Long lists of permissions are scary
  - Not a barrier
    - Plenty of big brands ask for plenty of permissions
    - Some percentage of your possible audience may elect to find some alternative



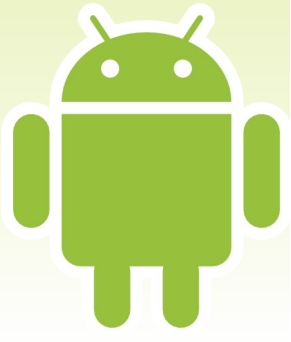
Users and Permissions, Part One

- Solution: Minimize Needed Permissions
  - Do not ship with permissions that you no longer need
    - Example: StackExchange
  - Consider whether feature X requiring new permissions will be worth the cost
    - May be a candidate for a plugin approach



Users and Permissions, Part Two

- No Optional Permissions
  - Must list all permissions up front, cannot ask for new ones at runtime
    - Avoiding “the Vista syndrome”
  - Users do not have ability to grant some permissions and deny others
    - Up front or after installation
    - Exception #1: AppOps
    - Exception #2: ROM mods
    - Exception #3: Your own optional permissions



Requiring Permissions

- In the Manifest
  - `<activity>`, `<service>`, `<receiver>`: `android:permission`
    - Single permission that other app must hold to communicate with this component
    - Can be system-defined or custom permission
  - `<provider>`
    - `android:permission`
    - `android:readPermission`
    - `android:writePermission`

```
<provider
  android:name="FileProvider"
  android:authorities="com.commonsware.cwac.security.demo.files"
  android:exported="true"
  android:grantUriPermissions="false"
  android:permission="com.commonsware.cwac.security.demo.OMG">
  <grant-uri-permission android:path="/test.pdf"/>
</provider>
```

Requiring Permissions

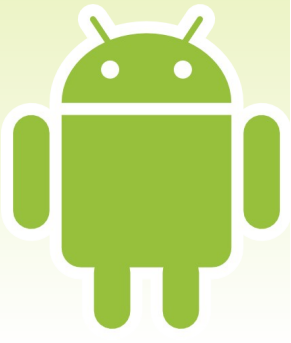
- In Java Code
  - `checkCallingPermission()`
    - Good for bound services
  - PackageManager and `checkPermission()`
    - Good for determining if given app (by PID) holds a permission

Requiring Permissions

- Scenario: System Data Leakage
  - You use some permission (e.g., `READ_CONTACTS`)
  - You expose some data through an API that came from a source secured by that permission (e.g., contact phone numbers)
  - You should require the same permission for accessing that API
  - Net: other apps cannot use you as “back door” way of getting private information without permission

Custom Permissions

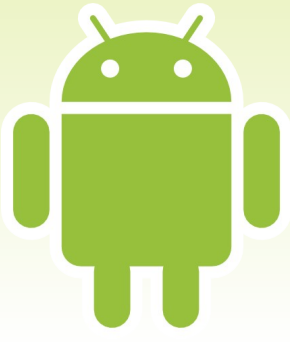
- Step #1: Add `<permission>` Element
  - Ideally to all apps tied to the permission
  - `android:name` = unique identifier
    - Do not use `android.permission` prefix, please!
  - `android:label` / `android:description` = what the user sees
    - Speak to users, not developers
    - String resources!
  - `android:protectionLevel`
    - `normal`, `dangerous`, `signature`



```
<permission
  android:name="com.commonsware.cwac.security.demo.OMG"
  android:description="@string/perm_desc"
  android:label="@string/perm_label"
  android:protectionLevel="signature"/>
```

Custom Permissions

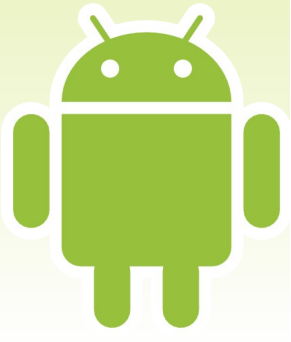
- Step #2: Add <uses-permission> As Normal
  - Putting the `<permission>` in all apps allows install order to be arbitrary
  - If `signature`, app requesting permission has to be signed with same signing key as app requiring permission
    - Great for developer-only plugins, app suites
    - Not great for plugins written by third parties



```
<uses-permission android:name="com.commonsware.cwac.security.demo.OMG"/>
```

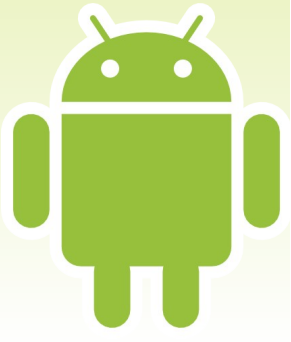
Your Own Optional Permissions

- Step #1: Isolate Permission-Requiring Code
  - Separate APK project
  - API that main (“host”) app uses
- Step #2: Secure with Signature Permission
  - Ensure that only your apps can talk to one another
- Step #3: Distribute Host and Plugin
- Net: Optional Permission
  - Only users who install the plugin need to grant you the permission required by the plugin



Permission Provider Proxy

- Wrap System Content Provider In Own
  - As a plugin APK, with the permission to access the system content provider
  - Forward all ContentProvider API calls of relevance on to the system's provider
- Host App Gets Data Via Proxy
  - Signature custom permission, so no leakage
  - No significant code changes in host
    - Mostly, use the Uri for the proxy



```
<uses-permission android:name="android.permission.READ_CONTACTS"/>
<uses-permission android:name="com.commonsware.android.cpproxy.PLUGIN"/>

<permission
  android:name="com.commonsware.android.cpproxy.PLUGIN"
  android:protectionLevel="signature">
</permission>

<application
  android:icon="@drawable/ic_launcher"
  android:label="@string/app_name">
  <provider
    android:name=".CallLogProxy"
    android:authorities="com.commonsware.android.cpproxy.CALL_LOG"
    android:permission="com.commonsware.android.cpproxy.PLUGIN">
  </provider>
</application>
```

```
public class CallLogProxy extends AbstractCPProxy {
  protected Uri convertUri(Uri uri) {
    long id=ContentUris.parseId(uri);

    if (id >= 0) {
      return(ContentUris.withAppendedId(CallLog.Calls.CONTENT_URI, id));
    }

    return(CallLog.Calls.CONTENT_URI);
  }
}
```

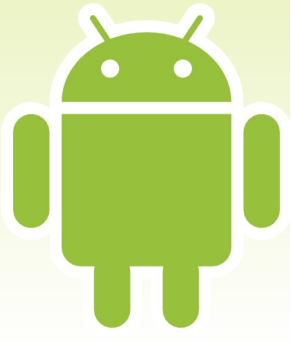
```
public abstract class AbstractCPProxy extends ContentProvider {
  abstract protected Uri convertUri(Uri uri);

  public AbstractCPProxy() {
    super();
  }

  @Override
  public boolean onCreate() {
    return(true);
  }

  @Override
  public Cursor query(Uri uri, String[] projection, String selection,
                      String[] selectionArgs, String sortOrder) {
    Cursor result=
        getContext().getContentResolver().query(convertUri(uri),
                                                projection, selection,
                                                selectionArgs,
                                                sortOrder);

    return(new CrossProcessCursorWrapper(result));
  }
```

```
<uses-permission android:name="com.commonsware.android.cpproxy.PLUGIN"/>
```

```
private static final Uri CONTENT_URI=
    Uri.parse("content://com.commonsware.android.cpproxy.CALL_LOG");
```

Custom Permission Vulnerability

- Rule: First One In Wins
  - First <permission> element for a given android:name determines behavior of that permission
  - App can define permission and have <uses-permission> and not use the permission for defense
- Net: apps installed before yours could hold your custom permissions
  - ...even signature ones, via downgrade



Custom Permission Vulnerability

- Environment
  - App A: defines and defends with custom permission
  - App B: defines and requests custom permission (attacker)
  - App C: just requests custom permission
- Scenario: App A, then App C
  - User notified about C's request, C gets access

Custom Permission Vulnerability

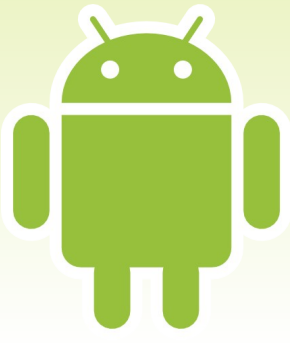
- Scenario: App C, then App A
  - User not notified, but C does not get permission, since not yet defined when it was installed
  - Problem for apps publishing SDKs
- Scenario: App A, then App B
  - Same as A → C: user informed, app gets permission

Custom Permission Vulnerability

- Scenario: App B, then App A
  - **PROBLEM**: User not notified about B's request
  - **PROBLEM**: B still gets permission
  - Downgrade Variant
    - A defines permission as signature
    - B defines permission as normal
    - B installed first, so permission is normal
    - B gets permission, despite no signature match

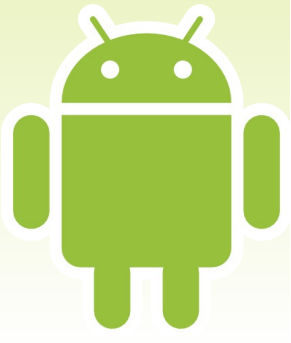
Custom Permission Vulnerability

- Not likely for bulk attacks, as normally no guarantee that B would be installed before A
  - If A installed first, user knows about B's request
  - In theory, eventually will be discovered as malware
- Bigger Risk: B Ships with Device
  - Used devices (not wiped or ROM mod)
  - “Presumed good” devices
    - Employer to employees
    - Gifts



CWAC-Security

- `PermissionUtils.checkCustomPermissions()`
  - Call on first run of your app
  - Returns details on other apps that have defined the same permissions that you have defined
    - If empty, continue as normal
    - If not empty, alert the user, send info along to your servers, etc.



```
public class FilesCPDemo extends Activity {
  private static final String PREFS_FIRST_RUN="firstRun";

  @Override
  public void onCreate(Bundle icicle) {
    super.onCreate(icicle);

    HashMap<PackageInfo, ArrayList<PermissionLint>> evildoers=
        PermissionUtils.checkCustomPermissions(this);

    if (evildoers.size() == 0 || !isFirstRun()) {
      Intent i=
          new Intent(Intent.ACTION_VIEW,
                     Uri.parse(FileProvider.CONTENT_URI + "test.pdf"));

      i.addFlags(Intent.FLAG_GRANT_READ_URI_PERMISSION);
      safelyStartActivity(i);
    }
```

```
else {
  for (Map.Entry<PackageInfo, ArrayList<PermissionLint>> entry : evildoers.entrySet()) {
    Log.e("SecurityDemoA", "This app holds the permission: "
        + entry.getKey().packageName);

    for (PermissionLint lint : entry.getValue()) {
      if (lint.wasUpgraded) {
        Log.e("SecurityDemoA",
              "...and they upgraded the protection level");
      }
      else if (lint.wasDowngraded) {
        Log.e("SecurityDemoA",
              "...and they downgraded the protection level");
      }

      if (lint.proseDiffers) {
        Log.e("SecurityDemoA",
              "...and they altered the label or description");
      }
    }
  }

  Toast.makeText(this, R.string.evil, Toast.LENGTH_LONG).show();
}
```

```
private boolean isFirstRun() {
  boolean result=false;

  SharedPreferences prefs=
      PreferenceManager.getDefaultSharedPreferences(this);

  result=prefs.getBoolean(PREFS_FIRST_RUN, true);

  prefs.edit().putBoolean(PREFS_FIRST_RUN, false).apply();

  return(result);
}
```

Slides! And Other Stuff Too!

http://commonsware.com/webinars/permissions.html