Android Permissions Demystified

Andrienne Felt, Erika Chin, Steve Hanna, Dawn Song, David Wagner

UC Berkeley

Presented by Gang Tan
Paper Overview

• This paper is more like an empirical study paper for understanding:
  • Android’s permission system
  • What permissions do Android’s API methods require?
  • Do Android apps request more permissions than necessary
Background: Android Permission System
Permission Overview

- Android enforces a per-application permission system
- Each app comes with a manifest that declares explicitly the set of permissions it requires
- Before installation of an app, Android asks the user to approve those permissions
Kinds of Permissions

- Android 2.2: 134 permissions in three categories
  - **Normal permissions**: protect access to API calls that could annoy but not harm the user
    - SET_WALLPAPER
  - **Dangerous permissions**: control access to API calls with potential harm to users
    - READ_CONTACTS
  - **Signature/System permissions**: regulate access to most dangerous privileges
    - E.g., the ability to control the backup process; the ability to remove a package
    - Android implement rules so that these permissions are only granted to pre-installed apps
When Are Permissions Required?

- For invoking the Android API
- For getting/setting user data in databases
  - These databases are called “content providers”
  - E.g., the Contacts Content Provider
- For sending/receiving intra- and inter-application messages
  - Called “intents” in Android
Android’s Architecture for Permission Enforcement

- A privilege separated architecture
  - Unlike the traditional JVM’s architecture
- Application process is sandboxed
  - Have a low-privilege user ID; can only access its own files by default (no other accesses)
- System-critical libraries are in a separate JVM
  - App performs RPC calls to the system JVM for security-critical operations
  - All permission checks happen in the trusted system JVM
  - Some permissions are enforced by the underlying OS group permissions
What About Native Code?

- Since app’s native code is in the application process
  - It’s subject to the same permission checks
- On the other hand, native code can possibly read/write the application process
Permission Testing
Overview

- **Goal**
  - For each API method, discover what permissions it needs for a successful invocation

- Isn’t that info in Android’s documentation?
  - Sort of, but it’s incomplete and may be inaccurate
Methodology: dynamic analysis

- High-level steps
  - Modify Android 2.2 to log permission checks
  - Invoke each API method to observe what permission checks happen

- Challenges
  - There are many methods in Android’s API
  - Some classes methods may be private and hidden (need to use Java reflection to invoke them)
  - Permissions may depend on arguments and ordering of method calls
Actual Steps: two phases

- Randoop: Feedback-directed testing
  - A unit tester
  - Method-call chaining: return values of methods can be used as arguments for other methods
- A custom tool
  - Separate test-case generation from execution, which allows manual adjustment
Some Results

- Android API: 1665 classes with 16,732 public and private methods
- They achieved coverage of 85%
  - Uncovered portion: native calls; classes do not require permissions
- Discovered 1,259 methods with permission checks
  - Android 2.2 documentation: for only 78 methods
  - Also discovered 6 inconsistencies between documentation and the actual permissions discovered
Characterizing Permissions

- Number of permission checks
- Unused permissions
- Hierarchical permissions
- Permission granularity
- ... (see the paper)
Application Permission Analysis
Overview

Goal

- Using the permission map, discover what permissions an Android application actually need and how do they differ from the declared permissions
Methodology

- A static-analysis tool that analyze Java bytecode to discover what API methods an app invokes
  - Then together with the permission-map info, we can know the permissions needed by the app
Stowaway

- Stowaway is a static analyzer
  - Performs flow-sensitive, intra-procedural static analysis with inter-procedural analysis to a depth of 2 method calls
  - It deals with Java reflections by tracking the values of String, StringBuilder, Class, Method, Constructor, Field, and Object
    - Similar to what need to tracked in JNI code
Analysis Result

- Selected 940 Android apps
  - 40 for manual analysis and the rest for automated analysis
- The tool identified 323 of 900 (35.8%) as having unnecessary privileges
result table for overprivilege

- Click to edit Master text styles
  
  - Second level
    
    - Third level
      
      - Fourth level
        
        - Fifth level
          
          | Permission | Usage |
          |------------|-------|
          | ACCESS_NETWORK_STATE | 16%   |
          | READ_PHONE_STATE   | 13%   |
          | ACCESS_WIFI_STATE  | 8%    |
          | WRITE_EXTERNAL_STORAGE | 7% |
          | CALL_PHONE         | 6%    |
          | ACCESS_COARSE_LOCATION | 6% |
          | CAMERA             | 6%    |
          | WRITE_SETTINGS     | 5%    |
          | ACCESS_MOCK_LOCATION | 5% |
          | GET_TASKS          | 5%    |

  Table 2: The 10 most common unnecessary permissions and the percentage of overprivileged applications that request them.

<table>
<thead>
<tr>
<th>Apps with Warnings</th>
<th>Total Apps</th>
<th>Rate</th>
</tr>
</thead>
<tbody>
<tr>
<td>Reflection, failures</td>
<td>56</td>
<td>105</td>
</tr>
<tr>
<td>Reflection, no failures</td>
<td>151</td>
<td>440</td>
</tr>
<tr>
<td>No reflection</td>
<td>109</td>
<td>355</td>
</tr>
</tbody>
</table>

  Table 3: The rates at which Stowaway issues over-privilege warnings, by reflection status.
Reasons for Overprivilege

- Confusing permission names
- Deputy permissions
- Documentation errors
- ...

The bottom line seems that developers try to follow the principle of least privilege, but may be misled to ask for more permissions
My take

- A useful paper that gives good background about Android’s permission system
- The permission map is useful for other Android research projects
  - They used dynamic analysis, but static analysis should also be applicable
- Handling Java reflection seems critical for precise static analysis of Android apps
  - This boils down to good string analysis
  - Critical for handing JNI code as well
Not Organized