WILLIAM & MARY

CHARTERED 1693

CSCI 445:
Mobile Application Security

Lecture 16

Prof. Adwait Nadkarni

Project Part 1 Grades Released!

- Great job folks!
- Average score ~94!
- *8/15 teams scored 100!*

Other Announcements

- **HW4 released on Sunday, due on April 16th, 11:59 PM**
- Directly related to *today's* ***class*** and HW3
  - Will discuss at the end of class (if possible)

Permission Analysis

Goal: Finding ***overprivileged*** apps

Recall: Principle of Least Privilege

A system should only provide those rights needed to perform the processes' function and no more.

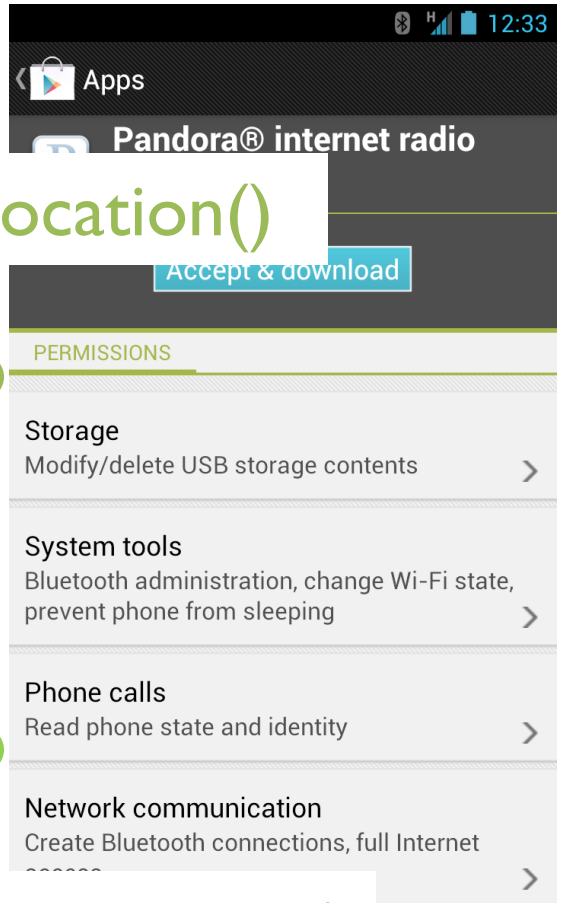
- Implication 1: you want to reduce the protection domain to the smallest possible set of objects
- Implication 2: you want to assign the minimal set of rights to each subject
- Caveat: of course, you need to provide enough rights and a large enough protection domain to get the job done.

How can we confirm that an app does not need a permission?

Recall: Android Permissions

- *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)*

```
            <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>
```

Uses API == Needs the Permission

- *Stowaway* [Felt et al. CCS'11]
- Basic approach:
  1. Analyze the manifest or code to determine the *permissions requested by the app (say $P_{request}$)*
  2. Use static analysis to determine *sensitive API calls made by the app*
  3. Build a *permission map*
     - Permissions needed for an API call
  4. Combine **2** and **3** and determine the *set of permissions needed by the app (P_{need}).*
  5. Violations = [$P_{request} - P_{need}$]

Permission Map

- *Goal:* To determine the permissions required to call an API
- Hundreds of APIs and about a hundred permissions
  - *Look in the documentation?*
- Basic approach: Empirical analysis
  - Modify the OS to *log* each permission check
    - Hook into a few methods (e.g., checkPermission(...)) in the framework
  - Execute all APIs using *automated testing*
  - Note the permission(s) checked when a test case is executed.

Challenges in building a *sound* permission map

- *What are the potential problems in automatically executing framework APIs/methods?*
  - Some APIs may expect a certain order (e.g., call something else before this)
    - Potential Solution: *Manually* adjust order
  - Some APIs may expect specific parameter values
    - Potential Solution: *Manually* add specific parameters
- Lesson: Need for *customizable*, *semi-automated* testing.

Building the permission map

- Are we done? Is there *any* need for *manual analysis*?
  - Different method argument values/combinations may result in different permission checks
  - Different API call sequences may also result in different permission checks
- Manual analysis and confirmation can examine arguments/combinations
- Are not API calls but still need permissions:
  - Content Provider URIs
  - System Intents/ protected String constants

Analyzing apps for overprivilege

- Disassembled DEX files as input
- Inter and Intra-procedural analysis
- Identify calls to known APIs
- *What about Java reflection?*

  `java.lang.reflect. Constructor.newInstance()`

  `java. lang.reflect.Method.invoke()`

  - Static analysis: Track Class and Method names
    - Up to a depth of 2 method call

```
Method sumInstanceMethod =
Operations.class.getMethod("publicSum", int.class, double.class);
```

Findings

- Over 900 apps analyzed, >35% overprivileged
- Potential Causes:
  - Developer Confusion
  - Insufficient documentation of permission requirements
    - Official (78 APIs) vs Stowaway (1259 APIs)
  - Errors in the official documentation
  - Copy and Paste

Factors affecting the documentation of Permission Maps

- Complexity of API, as well as the absolute number, is the main factor

Phone

- 2011 study [Felt et al.], identified 1259 API with permission checks (only 78 documented!)

Home

- Study of the Google Nest platform [Kafle et al.], identified the *same number as the documentation*

- Correctness of this map remains a "policy specification" issue
  - i.e., *does this API need a permission check?*

Kafle, K., Moran, K., Manandhar, S., Nadkarni, A., & Poshyvanyk, D. (2019, March). A Study of Data Store-based Home Automation. *Proceedings of the 9th ACM Conference on Data and Application Security and Privacy (CODASPY).Best Paper Award*

Pitfalls of this approach

- Q: Does an API call *really* mean that an app *needs* a permission?
  - No. What does the app claim to do? (e.g. ,use description, UI analysis)
- Dynamic code loading
- Needs to be continuously updated:
  - *Stowaway* is outdated
  - *PScout* provides mappings up to Android 5.1
  - axplorer [USENIX'16] provides mappings up to Android 7.1
    - https://github.com/reddr/axplorer

A Permission-based security policy

- Apps ask for *dangerous* permissions: This *security policy* is specified in the Android Manifest.
- If you know only the requested permissions: What is *undesirable/ potentially harmful*?
  - An application that can start on boot
  - An application that can get Location
  - An application that can use the Internet
  - How about an app that can do *all three*?
    - Potentially, a *tracker*

Security Rules (Kirin)

https://developer.android.com/reference/android/Manifest.permission.html

- Single Permission: SYSTEM_ALERT_WINDOW (Draw over other apps)
- Multiple Permissions: *Class Exercise!*
  - RECORD_AUDIO and INTERNET (eavesdropping)
  - ACCESS_FINE_LOCATION and RECEIVE_BOOT_COMPLETE (tracking)
  - SEND_SMS and WRITE_SMS (use phone as bot for spamming and erase evidence)
- Permissions and action strings: SET_PREFERRED_APPLICATION, Intent filter with CALL action

Deriving Security Rules

- Security ***requirements engineering***
- Manual process
  - Determine assets (e.g., Location data)
  - Determine security goals, and threats, i.e., *(mis)use cases* (e.g., in terms of *confidentiality,* an attacker may *get location and export it* to a remote server).
  - Determine the permissions required to compromise an asset (e.g., FINE_LOCATION, INTERNET permissions)
  - Limit rules to what is actually enforceable.

What is the most difficult step in this process?

Advantages

- Simple and fast analysis; good for *triaging* apps
- Easily deployable without significant modifications to the OS
  - Add it to the installer

Disadvantages

- Coarse-grained analysis:
  - *False Positives*, i.e., policy violations may be triggered by legitimate apps; Manual analysis may be required
  - *False Negatives*, i.e., Inter-app communication allows apps to collude; i.e., malicious functionality may be distributed among apps.

The End