CS216 Mobile Security Scribe Notes

Mobile Security

- Desktop OS (Unix, Windows) run apps using users priv; vs mobile apps are allowed specific privs by mobile OS (Android, iOS)
- **Approach 1** - Started off by building sandboxes, and static analysis of what privs an app would require. Didn't work well because static analysis might not pick up all privs an app requires.
- **Approach 2** - Dynamic analysis Is no better, since on different runs, different privs might be required.
- **Threat model** - Bening but buggy apps accessing and corrupting the system
- **Industry Approach**- Instead of inferring, its dev's responsibility to list permissions in the manifest file. Now, problem of evaluating if permissions listed are excessive/inappropriate.
- Excessive permissions are found in around 30% of the apps, by 1-2 additional permissions.
- The reason for excessive permissions might by due to automatic updates. If an app requires additional permissions during updates, it must prompt the user, instead of automatically updating.

- Another example of security model is for **Chrome browser extensions**.
- **Model:**

Page

Content script (Runs inside browser)	<->	Core Extension (Runs outside the browser)

Has a manifest file which lists permissions. The permissions are granted to the core extension, but not the content script. Also specifies which domain it runs on. Eg adblocker runs on all websites, but the gmail mail checker runs only on gmail. This ensures that if a malicious website compromises the JS interpreter on the page, it can't abuse the extension.

User-driven Access Control

- Isolating apps is great, but apps may need to access user's data. Existing security is coarse grained and inadequate.

- Approach 1: Ask for permissions on install. Eg Facebook apps
- Approach 2: Ask for permissions every time it needs permission. Eg Windows UAC
- **How to allow access to user-owned resources?**
  - System captures user-intent via authentic user actions in the context of an application
  - Implementation via access control gadgets (ACGs)
  - Users grant permissions at the time of use.
  - Minimizes burden by extracts user intent from user-app interaction behaviors.
- **System Model**
  - Apps insolated from each other
  - Access control mechanism
  - Access policies
- **Threat Model**
  - Malicious installed app
  - Attacker has null net work control and can communicate with other apps via IPC
  - Kernel works as expected.
- **Design**
  - Apps to build permission-granting Uis into their own context.
  - The system obtains user's intent though interactions with these Uis.
  - Resource monitor
    - exposes ACGs to apps
    - Device access APIs
  - Apps can invoke RM APIs only when granted permission.
  - Users can grant permissions.
- **Capturing User's Intent**
  - Ensure integrity of ACGs display
  - Display isolation, complete ACG visibility and sufficient display time.
  - Ensure authenticity of user's input.
  - Input events required to come from user's interactions with the ACG.
  - Kernel grants permissions to correct application.
- **Implementation**
  - New mechanism on top of Service OS with 2.5K lines of code.
- **Evaluation**

- Least-privilege model.
- User-driven access offers least-privilege to ~95% of apps.

Overhaul - Input-driven access control for better privacy on traditional OS

- **Problem** - Can we implement modern user-driven access control on desktops?
- **Threat Model**
  - Program runs stalthily in the b/g and access sensitive resources without user's knowledge.
  - Benign but buggy app accesses protected resources.
- **Assumptions**
  - Trusted Computing Base
  - Malicious code can execute with priv of user.
- **System Design**
  - Trusted i/p and o/p paths via display manager
  - Permission adjustment via kernel's permission, and there's a secure channel between the monitor and display manager.
  - Grants a permission based on temporal proximity i.e user clicking on the permission request button.
- **Sensitive Resource Protection Example**
  - When a request is made by the app, display manager relays the permission request with timestamp to the kernel and displays a prompt to the user. The user's acceptance timestamp is captured and sent to the kernel, which grants the permission.
- **Limitations**
  - Cannot protect against apps which mimic legitimate apps to install malware.
- **Evaluation**
  - 3% performance overhead
  - Usability tests on 46 people didn’t notice any difference in experience

Taint-droid

- Eg if an app received GPS coordinates from the OS, Taint-droid is used to find

out where that GPS info is used, if its transferred on the network etc.

- Use a indicator variable taint associated with each variable in the program. Set taint value as Y if it contains GPS info.
- Addition, concatenation operations result in the result variable being marked as tainted=Y as well.
- Implicit flows are hard to track even using dynamic analysis.