WILLIAM & MARY

CHARTERED 1693

CSCI 445: Mobile Application Security

Lecture 14

Prof. Adwait Nadkarni

Derived from Slides by William Enck

Announcements

- **Project Milestone 2 Graded!**
  - Most of you scored full or close to full
  - Will release grades by Thursday, along with HW3
- **HW4 will be released on Thursday**
  - Directly related to *today's* ***class*** and HW3
    - Will discuss in the next class
- **03/28 (Thursday):** Guest lecture on the ***legal implications of vulnerabilities in mobile/IoT!***
  - *Must attend!*

Intro to Static Analysis

Introduction

- Literally, analyze programs (i.e., apps in this case) without executing them
- Various *abstractions/granularities*: strings, call graphs, instruction-level, procedure-level
  - Some are more complex than the others
    - *In this class, we will study (and use) light-weight static analysis*
- Lots of analysis tools: *FlowDroid, AmanDroid, MalloDroid,...*
- Tools that enable analysis: *ded, dare, dex2Jar*

An Android app

- Is installed as an `apk`, which contains:
  - `AndroidManifest.xml`: A *binary* XML
  - `classes.dex`: Application code compiled into Dalvik Executable (dex) format.
    - Executes in a Dalvik VM (DVM) (or ART)
  - `resources.arsc` and `res/`: Application resources (e.g., UI layouts), important a few lectures later
  - `assets/`: Other assets packaged with the app
  - `lib/`: libraries compiled with the app
  - `META-INF/`: Stores the signature

Q: Can there be more than one .dex file?

Multidex support in static analysis

- Why is it important?
  - Android Studio enables multidex by default (since 2014)
  - Need to look for vulnerable code in *all .dex files*
  - *Otherwise, you may end up with significant false __________? (positives/negatives?)*

negatives?

Enabling Analysis

- Disassemble to *readable* Dalvik bytecode using `baksmali`
- *De-compilation* to source code (Java). Why?
  - Android apps are written in Java (generally)
  - To use existing tools for analyzing Java source code.
- Vast range of tools/techniques for decompiling Java applications (i.e., class files) to source code.
  - Q: Can we simply adapt these?
  - A: No; the JVM and DVM are *significantly* different
  - Solution: Retarget dex to Java class files (deep dive)
  - Tools: ded (superceded by dare), dex2Jar, recent additions to Soot.

*V*ery lightweight static analysis

- Searching for strings!
- Where would you search?
  - Lots of options, starting with, ...
- The *AndroidManifest* is a surprisingly rich source of information. ***Class Exercise!!***
  - What permissions does the app ask for?
  - What permissions does it define?
  - What kinds of components does it have?
  - Are they exported/internal?
  - What permissions are used to protect components
  - ...

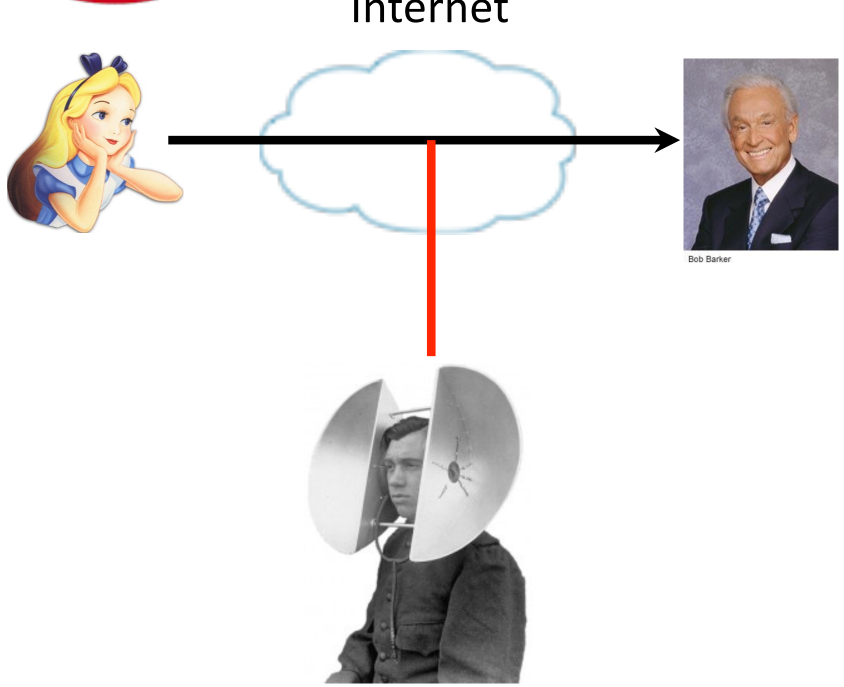
Grep is good, but use an XML parser

Lightweight static analysis

- Identify **classes/methods** of interest.
- Analyze **return values/types**.
- Used to identify ***potentially* vulnerable/malicious** target behavior.
- Analysis may raise *several false alarms*
  - Require manual effort to confirm

Example: Identifying SSL misuse

- MalloDroid (Fahl et al.)
- Target behavior: What are we looking for?
  - Trusting All certificates
  - Trusting All hostnames
  - No SSL pinning
  - No SSL use/ mixed use:
    - Recall: Why is mixed use a problem?
    - *SSL Stripping*
    - *Stealing cookies*



Locating Vulnerable code

- Parse code using existing tools (e.g., Androguard)
  - Get method definitions, class definitions, etc.
- Perform light-weight analysis based on some known signatures
  - Are there classes that override the TrustManager class?
  - In any of these classes,
    - Does the checkServerTrusted method return true?
    - Is the getInsecure() used to get the SocketFactory object?
  - If the HostnameVerifier is overridden,
    - Does it use an instance of the AllowAllHostnameVerifier?
- Identify more vulnerable custom classes, search for them again!

https://github.com/sfahl/mallodroid/blob/master/mallodroid.py

Advantages

- Fairly easy to implement, debug, and extend
- Fast
  - No call graphs, control flows, or any other complex data structures to build.
  - Allows you to quickly triage apps

Pitfalls

- Analysis may be *imprecise* (i.e., likely to have false positives)
  1. Analysis leads to *potential flaws*
     - Need manual analysis to confirm
       - Q: Why is this an issue?
       - A: Scalability (i.e., can you scale to all 10k apps?)
  2. Some flaws may be in dead code, or code that is unlikely to be executed (e.g., old libraries)
- Analysis may be *unsound* (i.e., likely to have false negatives)
  - Relies on coarse signatures, that will miss complex flaws
    - E.g., MalloDroid may not detect an app implements complex SSL verification logic, which may still be flawed.

Other (more complex) Program analysis

- Lots of Techniques: flow-sensitive, value-sensitive, context-sensitive analysis
- Can answer complex questions:
  - List of methods that may call this method
  - Potential arguments to be passed into this method (e.g., Crypto API)
  - Flows of data from source to sink methods (e.g., Location → Internet)
- Examples: FlowDroid, AmanDroid, DroidSafe, BlueSeal, ...
- Advantages: More precise and sound than lightweight analysis
- HOWEVER, *are they really as sound as they claim to be? (soon)*

General challenges for static analysis

- **Obfuscation:** For protecting IP (benign), or hiding malicious behavior.
  - Can range from simple (i.e., changing variable names to reduce readability) to very complex (e.g., modifying control flows)
- **Dynamic code loading**
- **Intricacies of Android's app model:** E.g., no "main" method, UI callbacks, lifecycle callbacks (relevant for deeper static analysis)
  - Prior work tries to overcome this with *lists*

The End