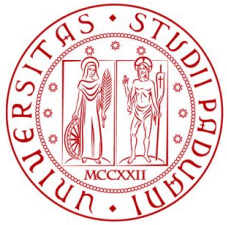
Mobile Security

Dr. Eleonora Losiouk
Department of Mathematics
University of Padua
elosiouk@math.unipd.it
https://www.math.unipd.it/~elosiouk/



UNIVERSITÀ
DEGLI STUDI
DI PADOVA

SPRITZ
SECURITY & PRIVACY
RESEARCH GROUP

DIPARTIMENTO
MATEMATICA

Possible reverse-engineering goals

- Generic understanding on what the app does?
- Find ways to attack the app?
- How does the app interact with network endpoints?
- How does it store private information?
- Check whether a specific functionality can be abused?

Generic Suggestions

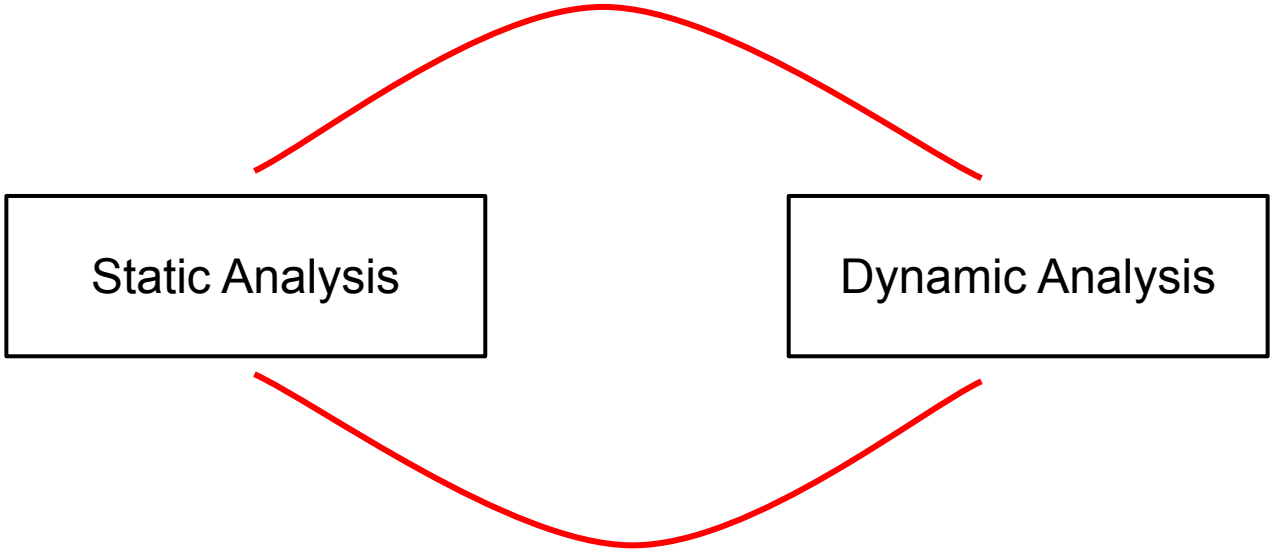
- Top-down mentality:
  - Start with high-level understanding of the app's organization/functionality, then drill down on the tech details depending on your needs
  - Start with the various entry points, explore the different functionalities, explore the app as a "user"
  - Perform attack surface analysis

- Keep a flexible mindset!

Two Main Approaches

- **Static** analysis vs. **dynamic** analysis
- Static analysis: inspect the app without running it
- Dynamic analysis: run the app and check what it does
- They are complementar: taken independently are limited
- Key skill: understand **which** one to use and **when**

Usual Workflow



Key: learn when to switch between static and dynamic analysis

Overview on Static Analysis

- Key point: you do not run the app
- You inspect it "statically"
- Take the app, unpack it, check what's inside
  - Manifest analysis, disassemble/decompile .dex, check .so files, etc.

Overview on Dynamic Analysis

- Key point: you actually run the app.
- You want to know what's going on "at run-time"
  - Actual values at run-time (useful when strings are obfuscated)
  - Trace of API invocation
  - Trace of syscalls
- Two main techniques
  - Debugging
  - Instrumentation

Debugging

- You run the app and you "attach" a debugger
- You can ask debuggers a number of things
  - Stop the execution when you reach point XYZ
  - Tell me the content of this field / memory location
  - Single-step through instructions
- Helpful to understand the state / context at a given point

Instrumentation

- Run the app in an "instrumented" environment so that
  - Every APIs invocation is traced
  - Every network-related API invocation is traced + all their arguments
  - Log all strings
  - Dump additional information when specific conditions are met
  - Note: too much info is not always good!
- Many technical ways
  - Modify the Android emulator, ART environment, the app's itself
  - Manual instrumentation vs. instrumentation frameworks
  - But still, somehow it is still an open problem

"Manual" app modifications

- Bytecode injection
  - Unpack the app, add extra functionality, pack the app
- Example:
  - What's the value at run-time of variable X?
  - Add proper Log invocations
- Usual trick
  - 1) write your functionality in Java
  - 2) get the smali
  - 3) inject the smali

Frida Framework

- It can instrument the app itself
- Based on code injection
  - It actually injects a Javascript engine into a running process
- Modules are written in Javascript

Frida Framework

- By default, it requires root
  - It needs to ptrace the target app for code injection
- You can inject the "frida-server" directly into the target app
  - https://koz.io/using-frida-on-android-without-root/
- In both cases: it can be detected

Static analysis pros/cons

- Pros
  - Initial general understanding
  - What does the app do from an high-level perspective / "semantics"?
  - Determine which points are interesting
  - Answer questions such as: how can I reach a specific point?
  - Find security vulnerabilities
- Cons
  - Some values may be difficult to determine at static-analysis time!
    - String obfuscation, complex algorithms, etc.
  - Some of them may not even be available!
    - Strings coming from the network, dynamic code loading, etc.

Dynamic analysis pros/cons

- Pros
  - Dump actual values at run-time
  - Dump network traffic (both sent and **received**), no reconstruction needed
  - Stop analysis at any-point and do context/memory inspection
  - "Is this API ever invoked?", "With which arguments?"
  - Verify that a security vulnerability is actually exploitable
- Cons
  - Limited code coverage: Where should I click? Input to insert?
  - How can I reach a specific point?
  - Is what I'm seeing "bad"? Missing context!
  - It can be evaded (app can understand it is under analysis / bypass it)

Automatic Analysis Approach

- How do researchers find malware / bugs?
  - In many cases: mostly manual reverse engineering
  - The holy grail: find everything automatically
- Why is it important?

Why?

- The scale is huge
- Not enough good people to analyze all apps out there
  - Google Play Store: 3 million apps+
  - Apple Store: 1.8 million
- Researcher still find malware & vulns in very popular apps

Program Analysis

- Static analysis
  - Manual: static reverse engineering, manual unpacking & inspection
  - Automatic: detection of specific payload, signature-based matching, taint tracking, symbolic execution

- Dynamic analysis
  - Manual: manually start & interact with app in instrumented environment, instrumented app, debugging
  - Automatic: fully automated system that takes an APK, run it in an instrumented environment and track what it's doing

Static Program Analysis

- Common traits
  - The app is not actually executed (hence, "static")
  - "Scan" all possible paths (vs. "only one")
  - While scanning the code, keep track of "relevant information"
    - What "relevant info" is depends on the specific instance of static analysis
- Useful to answer
  - Can X happen (when running the app)?
  - Can I be sure that X never happens (when running the app)?
    - Useful to extract "invariants"
    - "Invariant" ⇔ a property that always holds when the app is run

Static Analysis

- Simple/scalable approaches
  - "Quick" types of analyses that do not require significant resources
  - Manifest analysis, simple API analysis / scanning, signature scanning
    - YARA rules
  - The output is presented to the analyst and/or
  - The output is fed to a classification engine based on machine learning

- Actual static program analysis

Taint Analysis

- Conceptual steps
  - "Go through" the program/app you want to analyze
  - If a register contains something "sensitive" (e.g., location data), "taint it"
  - "location taint" ⇔ "value in register X may contain "info of type location"
  - Propagate taint according to the operations performed on these objects
- Common use case
  - Q: Does this app send location information to a network end-point?
  - Analysis
    - Taint objects coming from location-related APIs ("sources")
    - If a tainted object reaches network end-point APIs ("sinks"), flag the app
  - Popular "research" tool: FlowDroid

Symbolic Execution

- Conceptual steps
  - "Go through" the program/app you want to analyze
  - When possible, keep track of "real" values / "concrete"
  - When not possible, keep track of values "symbolically"
  - Keep track of each operation on these objects in form of "symbolic expressions"

- Common use cases
  - Q: which conditions should be satisfied for a branch ("if") to be taken?
  - Q: what are the possible arguments API X is invoked with?

Problems

- Inherent trade-off between scalability and precision
- Precision is (usually) characterized by FP / FN
- FP: False Positive
  - The analysis says "X can happen" while, in fact, it cannot happen
- FN: False Negative
  - The analysis says "X will never happen", while, in fact, it can happen
- The semantics of FP/FN is analysis-dependent

Another Problem: Evasion

- Malicious apps intentionally make use of code constructs to make static analysis challenging
- Reflection, dynamic code loading, string obfuscation, native code
- How to statically deal with these tricks is an open research problem
  - Current solution: use hybrid approach with dynamic analysis

Dynamic Program Analysis

- Common traits
  - The app is actually executed
  - Usually: only "one path" at the time (all values are "concrete")
  - Either the app or the environment is instrumented to track what happens
  - It needs to address the "how to interact with the UI" problem
- Useful to answer
  - Does X happen (when running the app)?
    - Cf. "can X happen?" of static analysis
    - Cf. "can I be sure that X never happens?" of static analysis
  - Collect files accessed, network connections, IP addresses / URLs, APIs invoked, access of sensitive info (location, contacts, etc.), UI usage

Dynamic Program Analysis

- We have already seen the underlying techniques
  - debugging
  - instrumentation
    - app rewriting / repackaging
    - environment instrumentation
    - xposed, frida
  - UI interaction
    - monkey runner
    - uiautomator
- Analysis usually builds on top of these blocks

Problems

- Key problem: limited code coverage
  - Dynamic analysis only sees what it "reaches"
    - If the malicious piece code is not even reached, dynamic analysis will not see it
  - In the general case: it's difficult to "explore" everything
  - Compare with static analysis that (in principle) covers "all paths"

- Common difficult questions
  - On which button should the analyzer click to reach X?
  - What are the conditions to reach X?
  - What should be the environment set to?

Problem: Evasion

- When analyzing malware: dynamic analysis evasion!
  - Malware uses "code coverage" problem to its benefit
- Intentional use of "difficult-to-satisfy" conditions
  - Time bombs: functionality X only runs at a specific time constraint
    - Rationale: this "time constraint" will likely NOT be satisfied during the tests / analysis
  - Many variants: location bombs, SMS bombs, env bombs, logic bombs

Problem: Evasion

- Malware apps attempt to determine whether they are "under analysis"
  - Emulator-related checks: if emulator ⇒ don't do anything
    - Rationale: real users don't use emulators; Google Bouncer does
  - Motion-based: no motion during the analysis (because it's all fake)
  - Is the app repackaged/instrumented? Is debugger/xposed/frida running?
  - Is the environment "too clean"? Distinguish real user vs. analysis system
    - Idea: pre-populate address book, photo album, sms messages, files, ...

Summary

- Static Program Analysis
  - Pros
    - Full code coverage
  - Cons
    - Evasion problems
- Dynamic Program Analysis
  - Pros
    - Results are usually "actionable"
  - Cons
    - Code coverage problems
    - Evasion problems