Detection of Malicious Code

This is a two-year SEI-funded project, ending in Oct 2024.

Will Klieber

January 2023

Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center.

DM23-0054

Problem

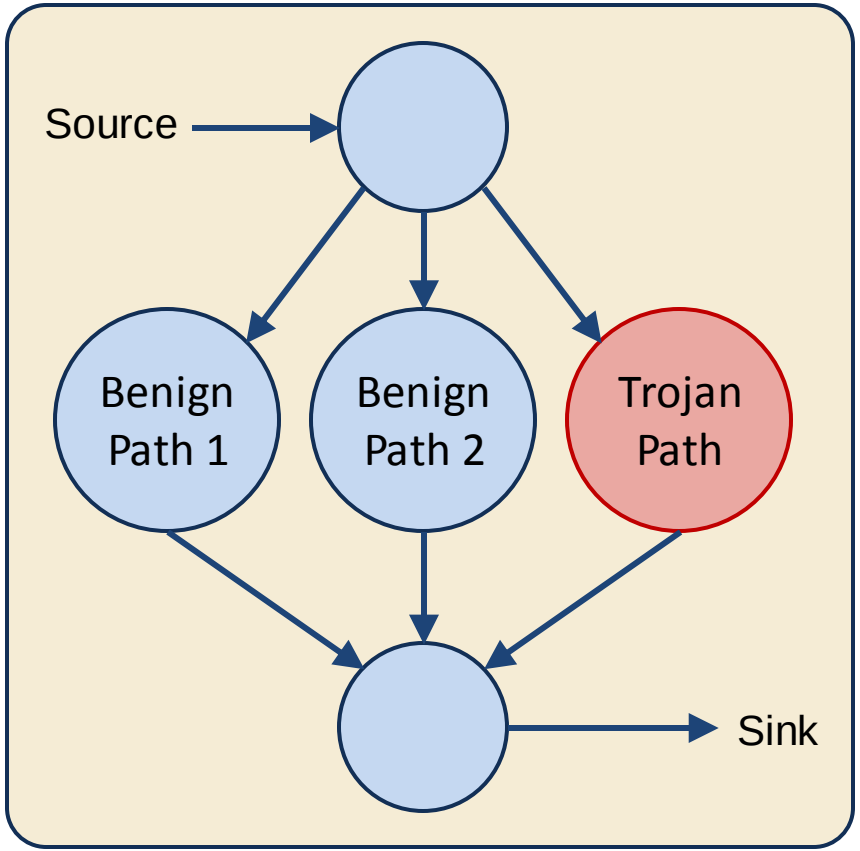
- DoD uses much software produced by various supply chains.
- These supply chains can be compromised by an adversary:
  - Network intrusion
  - Insider threat
- Failing to detect malicious code can be very costly.
- Example: SolarWinds incident of 2020.
- Detection is currently impractical.

Our solution

- We will design a static analysis to detect two types of malicious code:
  - Exfiltration of potentially sensitive information (e.g., keyloggers)
  - Calling a potentially sensitive system API call (e.g., writing to a file, starting a new process, etc.) in response to a potentially questionable trigger (e.g., on a specific date, in response to incoming network packets, etc.)
    - E.g., Remote-Access Trojan
- We will use ***information flow*** techniques, as well as other static analysis.
- Our tool will also attempt to find *witness traces* detailing the flows.
- E.g., for the malware injected in the SolarWinds Orion code, we would find:
  - Execution trace where attacker launches an arbitrary process
  - Execution trace where attacker writes arbitrary data to arbitrary files
  - Etc.

Information Flow Analysis

- Interprocedural, Finite, Distributive Subset (IFDS) algorithm
  - has successful track record, e.g., finding malicious flows of information in Android apps.
  - Sources: designated system API calls that return potentially sensitive information.
  - Sinks: designated system API calls that can be used to exfiltrate data to an attacker.
- Limitation: conflates together all flows from a given source to a given sink.
- So, a malicious flow can be 'hidden' by a benign flow.
- Our idea: Separate flows according to which parts of the code are exercised in propagating the flow.



Technical approach

- We will implement an IFDS-based analysis with an enhancement to prevent malicious flows from being masked by legit flows.
- For each LLVM IR instruction or basic block:
  - If there is a flow involving this code, ensure we generate at least one witness trace for it.
- This identifies flows in malicious code separately from flows in legit code.
- Identify restrictions on what data an attacker can pass to sensitive sinks (e.g., reading files restricted to only a single folder).
- A human analyst examines the flows to determine whether they are malicious.
  - Reduce false alarms by filtering out alarms about potentially sensitive operations that the user indicates would be expected of the program being analyzed.
  - Develop other techniques to prioritize alerts and reduce false positives.

Technical questions

1. Which platform(s) are of greatest interested? (E.g., Java, C/C++ code for x86 Windows, code for embedded systems, etc.)
2. Are you mostly interested in analyzing binaries, or is analysis of code source of interest too?
   - For C/C++ source code, we would plan to do the analysis at the LLVM IR level.
   - For binaries, we can do analysis at the LLVM IR level using a binary-to-LLVM lifter, but accuracy might be too low.
3. Are any particular system API calls (or categories of API calls, or general characteristics) of greatest interest?
   - What about direct I/O without using system calls?
4. Do you have any suggestions for malicious and benign benchmarks?
   - github.com/ytisf/theZoo
   - Examples for types of malicious code (e.g., keyloggers, timebombs) that you're most interested in?

Our transition goal

- Our goal is to develop a tool that provides enough benefit for it to be integrated into operational tools/processes.
- Question: What would be needed of the tool for it to provide this level of benefit?

Requested feedback for preliminary version of tool

- We aim to have a preliminary version of our tool ready around July or August.
- Would you be available to try running the tool and give us feedback?
  - Would you be able to run the tool on representative code that you might be unable to share with us?
  - If running on software that you cannot share with us, can you share basic summary stats (e.g., numbers of various types of flows, any errors/crashes from the tool, analysis time)?
  - What is the useful about the tool?
  - What needs to be improved for the tool to be provide enough benefit to be used in operational practice?
- In year 2 of this two-year project, we would like to engage more closely with you, aiming to improve the tool so that it can provide real benefit in operational use.

Plans for further meetings

- We plan to update you regularly by email on our progress.
- We'd also like to meet occasionally as the tool becomes more fleshed out, to ensure that we're still on track to produce a tool that would be useful to you.