Detection of Malicious Code Using Information Flow Analysis

Introduction

The Department of Defense uses a lot of software produced by various supply chains, which can be compromised by an adversary.

Examples: xz backdoor incident of 2024; SolarWinds incident of 2020

Our tool is designed to detect exfiltration of sensitive information as well as timebombs/logic bombs, remote-access Trojans, etc.

Overview of Approach

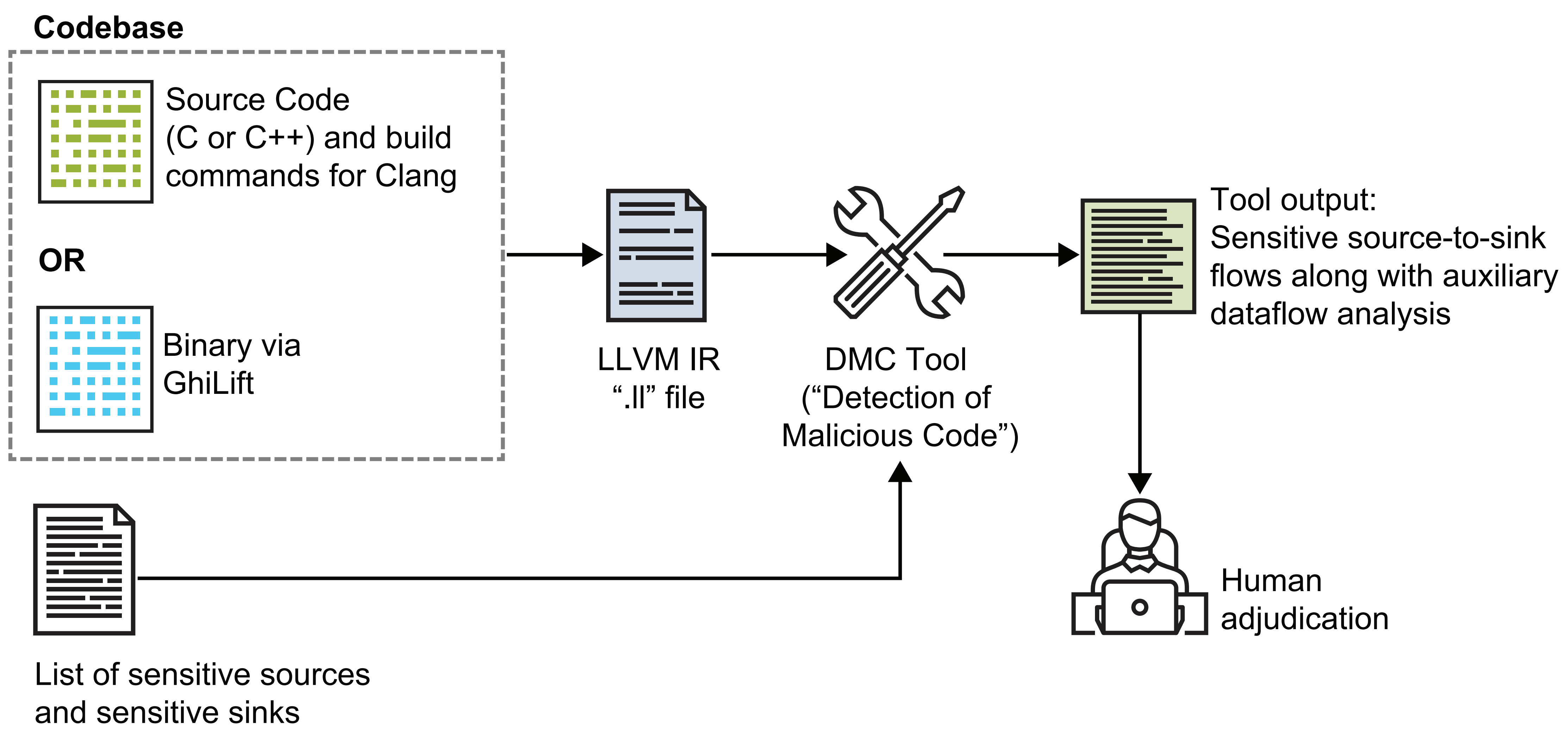
Our tool flags code as **potentially malicious**; however, further human analysis is required to determine whether the code is **actually malicious**, because whether behavior is malicious depends on what the program is supposed to do.

Our tool detects vulnerabilities such as Log4Shell in Log4j, but vulnerabilities involving undefined behavior are out of the tool's scope.

We are using only static analysis, not dynamic analysis. We have focused on C/C++ codebases, and our tool works natively at the level of LLVM intermediate representation (IR). We also have some support for binaries by lifting to LLVM IR.

Initially, we built on PhASAR, a static-analysis framework based on LLVM. However, PhASAR had trouble scaling to real-world codebases, so we reimplemented directly on top of LLVM. This proved much more scalable but less precise.

Our tool identifies flows of information that indicate **malicious code** inserted as a **supply-chain** attack and provides detailed summaries for human adjudication.


Codebase
Source Code (C or C++) and build commands for Clang
OR
Binary via GhiLift
LLVM IR ".ll" file
DMC Tool ("Detection of Malicious Code")
Tool output: Sensitive source-to-sink flows along with auxiliary dataflow analysis
Human adjudication
List of sensitive sources and sensitive sinks

```
[
{"sink": {"func":"write", "callsite":["mal-client-3.c","main",152,21],
    "aux file": [{"func":"socket", "callsite":["mal-client-3.c","main",65,18]}]},
 "srcs": [{"func":"fread", "callsite":["mal-client-3.c","main",139,29],
    "aux file": [{"func":"fopen", "callsite":["mal-client-3.c","main",132,26],
      "aux file": [{"func":"getline", "callsite":["mal-client-3.c","main",106,26], "FILE*":"stdin"},
                   {"func":"getline", "callsite":["mal-client-3.c","main",117,30], "FILE*":"stdin"}]}]}]},
...
]
```

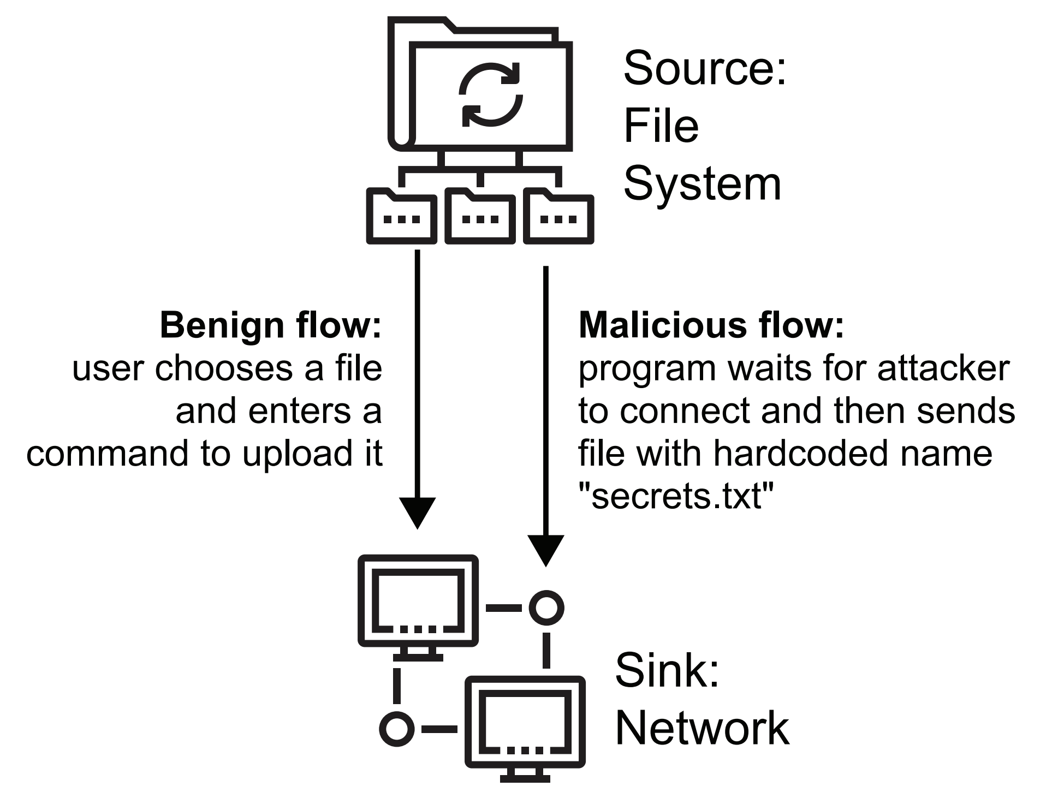
Example Tool Output

Static Taint Analysis

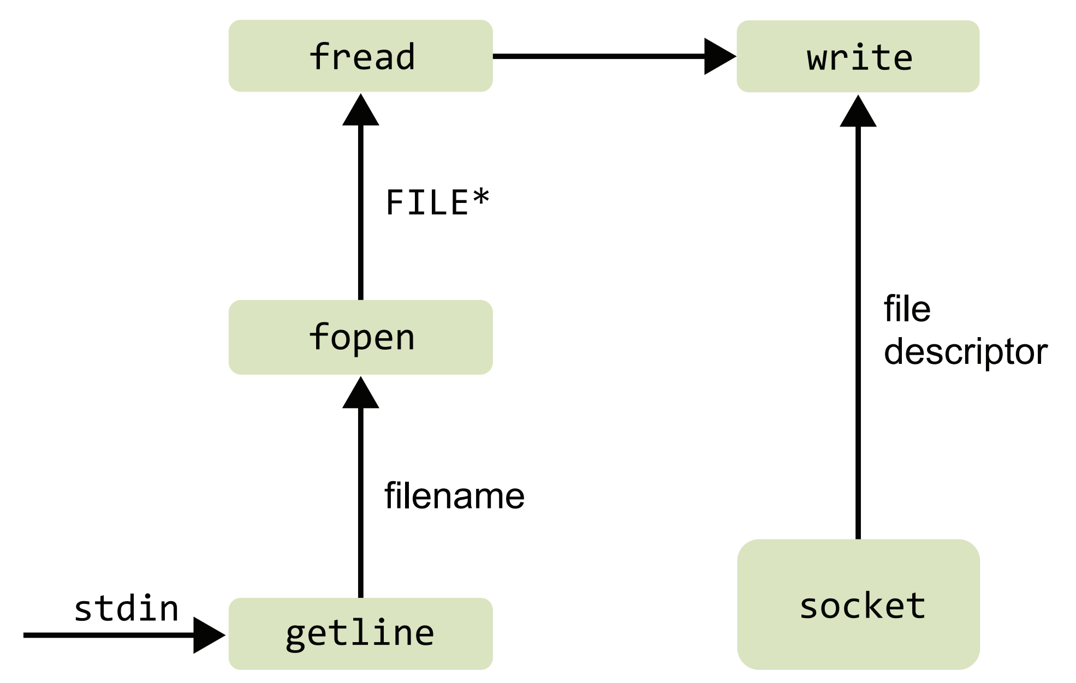
Sources are designated system application programming interface (API) functions that return potentially sensitive information.

Sinks are designated system API functions that can exfiltrate information to outside the program.

Usually, a static taint analysis conflates together all flow paths from a given source to a given sink. Therefore, a malicious flow path can be "hidden" by a benign flow path. To avoid this, we separate the flows by features relevant to detection of malicious code.


Source: File System
Benign flow: user chooses a file and enters a command to upload it
Malicious flow: program waits for attacker to connect and then sends file with hardcoded name "secrets.txt"
Sink: Network

Example with Benign and Malicious Flows



Visualization of Example Tool Output

Will Klieber| weklieber@sei.cmu.edu

[DISTRIBUTION STATEMENT A]
Approved for public release and unlimited distribution.
DM24-1460

6303_Klieber_6-23_6221_2_2.indd 1 10/31/24 11:54 AM