Exposing Secure Coding Vulnerabilities with a Custom Static Analysis Tool

Tom Schollenberger, Landon Spitzer, Samrudhi Ravindra Yadgude
Rochester Institute of Technology, Rochester, New York, USA
tss2344@rit.edu, lbs9440@rit.edu, sy5029@rit.edu

Motivation

- Open-source software forms the backbone of critical systems, but many repositories contain vulnerabilities that have remain in the code.
- Secure coding guidelines exist but are inconsistently applied across these repositories.[2]
- Developers and reviewers tend to overlook some security concerns, leading to unnoticed vulnerabilities that can be exploited [1]

Developed Solution

- A structured method for identifying secure coding weaknesses across large C and C++ repositories.
- A classification process using Coverity Scan and CWE mappings to find vulnerability patterns typically overlooked manually.
- A lightweight static scanner for detection of high-risk coding patterns using TOML based rules.
- A foundation for patch development, using the system's classifications to guide fixes.

Common Points of Weakness

Memory Safety Errors

- CWE-119: Buffer Errors – Unbounded operations that can overwrite nearby memory
- CWE-476: Null Pointer Issues – Using pointers that were never checked for validity

Input and Data Handling Issues

- CWE-134: Format String Problems – *printf*-style calls using untrusted or non-literal format strings
- CWE-798: Hard-Coded Secrets – Credentials or tokens stored directly in the source code

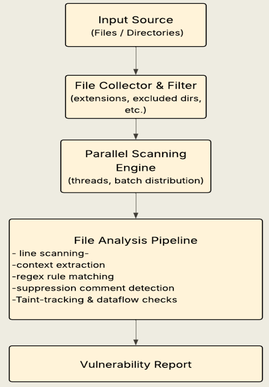
System Interaction Issues

- CWE-78: Command Injection Risks – Calls to *system*, *popen*, or *exec** with unchecked input that could execute unintended commands

System Design and Processing Steps

Step 1: System Architecture

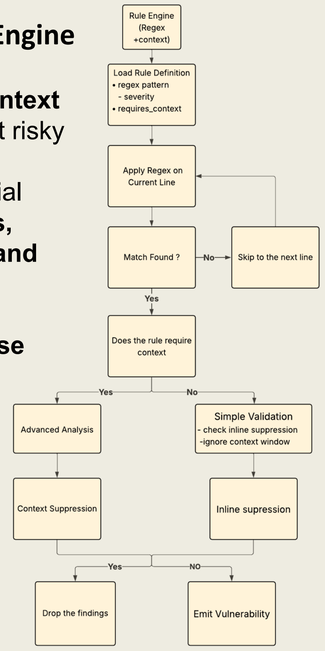
- **Detects vulnerabilities** in C/C++ programs using static analysis.
- Designed to be **lightweight**, **multithreaded**, and **extensible**



The overall system components and flow design

Step 2: Rule -Engine

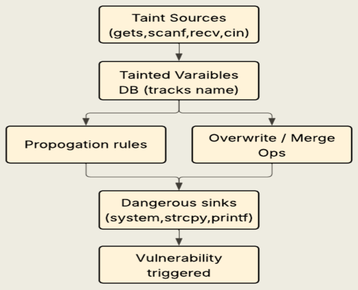
- Uses **regex + context window** to detect risky functions
- Highlights potential **buffer overflows, format strings, and memory leaks**.
- Helps evaluate **accuracy vs. false positives**



Shows how each code line is matched against regex rules and optionally analyzed

Step 3: Taint-Tracking and Analysis

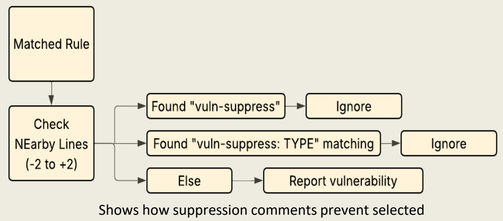
- Marks **untrusted input** from command-line arguments
- Works with rule engine for **prioritizing warnings**.



How user input is marked tainted and traced through the program

Step 4: Suppression Handling

- Allows developers to **ignore false positives**
- **Inline suppression**: per-line comments
- **Contextual suppression**: automatic suppression based on surrounding code.



Shows how suppression comments prevent selected findings from reporting

Results

Metric	Syntactic Scanner	Coverity Scan
Methodology	**Pattern Matching:** Fast, regex-based rules defined in TOML.	**Semantic Analysis:** Deep inspection of memory paths and logic.
Total Defects	15	134
Categories	2 (Overflows, Leaks)	17 (Integer Errors, Null Deref, Leaks, etc.)
Coverage	**11.8%** of total categories found.	**100%** comprehensive coverage.
Context	**Strict Enforcement:** Flags all banned patterns (e.g., wrappers) to enforce hygiene.	**Heuristic Filtering:** Ignores dangerous patterns if logic suggests safety.
Speed	**High Speed:** Immediate feedback suitable for pre-commit hooks.	**High Latency:** Requires full build integration and processing time.
Role	**Security Linting:** Enforces code hygiene, prevents unsafe primitives.	**Debugger:** Identifies runtime logic errors and crashes.

- Acts as a lightweight, lint-style checker that enforces secure coding practices early, while Coverity focuses more on deeper analysis
- Runs in seconds on large codebases making it ideal for continuous use

Status and Concluding Remarks

- Exposed recurring security issues that persist even in well-maintained open-source codebases.
- Developed a lightweight static tool that flags high-risk C/C++ patterns and reinforces secure coding practices.
- Showed that stronger OSS security requires layered tools supported by current legal and ethical practices.

References

[1] Wachiraphan Charoenwet, Patanamon Thongtanunam, Van-Thuan Pham, andChristoph Treude. 2024. Toward effective secure code reviews: an empiricalstudy of security-related coding weaknesses. Empirical Software Engineering 29,4 (2024), 88. doi:10.1007/s10664-024-10496-y

[2] Emanuele Iannone, Roberta Guadagni, Filomena Ferrucci, Andrea De Lucia, andFabio Palomba. 2023. The Secret Life of Software Vulnerabilities: A Large-ScaleEmpirical Study. IEEE Transactions on Software Engineering 49, 1 (2023), 44–63.doi:10.1109/TSE.2022.3140868