Detecting Integer-Overflow-to-Buffer-Overflow

Samin Islam

December 2025

1 Problem Statement

For my computer security project, I have chosen to investigate the Integer-Overflow-to-Buffer-Overflow (IO2BO) vulnerability, which is characterized by insufficient memory allocation and subsequent buffer overflows, inspired by [1]

Integer-Overflow-to-Buffer-Overflow (IO2BO) vulnerabilities are critical security risks where integer overflows lead to insufficient memory allocation and subsequent buffer overflows. Existing detection methods face significant limitations: dynamic testing often suffers from low code coverage due to reliance on specific inputs, while static analysis tools typically generate excessive false positives or fail to scale. Furthermore, state-of-the-art tools like KINT often miss vulnerabilities involving complex data structures or inter-procedural paths. Consequently, there is a pressing need for a detection framework that is both accurate enough to handle complex code paths and lightweight enough to analyze large-scale software like the Linux kernel.

2 Methodology of ELAID

The authors developed ELAID [1], a static analysis framework implemented as LLVM passes, which detects IO2BO vulnerabilities through a precise three-step pipeline.

- Global Call Graph Construction via Two-Stage Analysis: To enable inter-procedural tracking, the system builds a system-wide call graph. A key innovation is its two-stage handling of indirect calls (function pointers). First, it employs a definition-based analysis that scans the code for explicit assignments of function addresses to struct fields or global variables, tracking these targets precisely. If this method fails due to type escaping (e.g., void pointers), the system falls back to a type-based analysis, which matches function signatures to callsites to ensure no potential execution paths are missed.
- Vulnerability Identification (Taint Analysis): ELAID performs inter-procedural taint analysis on the LLVM Intermediate Representation (IR). It marks

input data as "tainted" and propagates this status through the program. Unlike previous tools, it supports implicit data flow (tracking taint through memory load/store operations) and is field-sensitive, meaning it can track tainted data stored in specific fields of complex data structures. The system flags a "candidate" vulnerability whenever a tainted arithmetic result is used as an argument for memory allocation functions like malloc. This step explicitly covers scenarios where the Integer Overflow (IO) site and the Buffer Overflow (BO) site reside in different functions.

- False Positive Filtering (Constraint Solving): To verify candidates, the system generates a mathematical formula combining two elements: the Overflow Condition (logic checking if the operation mathematically wraps around) and Path Constraints (conditions required to reach the vulnerable code from the caller function). These combined constraints are fed into the SMT solver Boolector. If the solver determines the constraints are unsatisfiable (impossible to execute), the candidate is discarded as a false positive, ensuring high report fidelity

3 Contributions of the Paper

ELAID [1] demonstrated significant improvements in accuracy and effectiveness over existing tools like KINT and LAID.

- High Accuracy: In the NIST SAMATE test suite, ELAID detected 100
- Real-World Effectiveness: When tested on seven open-source applications (including the Linux kernel), ELAID successfully detected all 14 known vulnerabilities, whereas KINT detected only 4.
- False Positive Reduction: The constraint-based filtering module successfully eliminated an average of 49.2
- Inter-procedural Detection: By extending analysis to cover inter-procedural paths (IO2IPBO), ELAID increased the number of identified vulnerable sites by 73.2
- Improved Resolution: The two-stage indirect call analysis resolved 88.7

4 My Implementation & Results

To evaluate the challenges of detecting integer overflows, I implemented an LLVM pass plugin, a static analysis and transformation tool built as an LLVM Pass. Unlike ELAID [1], which focuses on detection and verification, my pass is designed as a security diagnosis tool. Its primary goal is to identify untrusted integer arithmetic operations and inject runtime checks to prevent potential overflows dynamically.

Methodology The implementation utilizes a custom Type System to track the security state of variables through a fixpoint iteration algorithm:

- **Type 3 (Dangerous):** Represents values that are both *Untrusted* (derived from user input) and *Unchecked* (mathematically capable of overflowing).
- **Taint Tracking:** The pass hardcodes `fscanf` as a taint source. It propagates taint through arithmetic operations (`Add`, `Sub`, `Mul`, `Shl`) and memory operations (`Load`/`Store`) using LLVM's `AliasSetTracker` to handle pointer aliasing.
- **Detection Logic:** Any arithmetic instruction resolved to "Type 3" is flagged as a vulnerability candidate. Unlike ELAID, which specifically checks if these values reach memory allocation sinks (like `malloc`), my implementation flags the overflow site itself.
- **Mitigation (Patching):** For every detected "Type 3" instruction, the pass injects a Basic Block containing runtime validation logic (e.g., checking if $Pos \times Pos = Neg$ for signed multiplication). If an overflow is detected during execution, the program traps (exits with code 42).

Results I evaluated my pass against two test cases to measure its effectiveness and precision.

Test Case 1: True Positive

- An integer n is read from `stdin` via `fscanf` and multiplied by 4 to determine the size of a buffer `malloc(`$n*4$`)`. No bounds checks are performed. And the tool successfully identified the taint source (`fscanf`) and propagated the "Type 3" status to the multiplication instruction (`%19 = mul nsw i32 %18, 4`). It correctly flagged this as a dangerous operation and injected a runtime trap.

```
===============Dangerous Registers====================
[DANGER] Fn:main Inst:  %2 = alloca i32, align 4 Type:3
[DANGER] Fn:main Inst:  %11 = load i32, ptr %2, align 4 Type:3
[DANGER] Fn:main Inst:  %15 = load i32, ptr %2, align 4 Type:3
   -> Related arithmetic operations:
        %16 = mul nsw i32 %15, 4
[DANGER] Fn:main Inst:  %16 = mul nsw i32 %15, 4 Type:3
[DANGER] Fn:main Inst:  store i32 %16, ptr %3, align 4 Type:3
[DANGER] Fn:main Inst:  %17 = load i32, ptr %3, align 4 Type:3
[DANGER] Fn:main Inst:  %18 = sext i32 %17 to i64 Type:3
[DANGER] Fn:main Inst:  %26 = load i32, ptr %2, align 4 Type:3
```

Figure 1: Test Case 1; correctly flagged

Test Case 2: False Positive

- Similar to Test 1, n is read from `stdin`, where the code includes a check: `if (n <= 0 || n > INT32_MAX/4) return 1;`. This explicitly prevents the multiplication $n * 4$ from overflowing at runtime. But unfortunately my pass incorrectly flagged that as dangerous by marking "Type 3". This is the limitation of my implementation, which lacks path sensitivity and constraint solving of the program. It sees that n is tainted and used in arithmetic, but it cannot determine that the path to the overflow is mathematically impossible and falls into a False Positive.

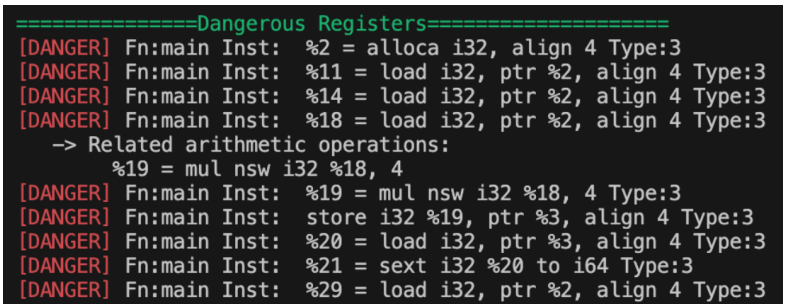

Figure 2: Test Case 2; False Positive

5 Insights and Conclusions

The experimental results confirm that my implementation effectively neutralizes vulnerabilities via runtime patching but suffers from high false positives (Test 2). To solve this problem, I should try the formal **Vulnerability Filter** from ELAID [1] would eliminate false positives by generating path constraints and using SMT solvers to prove if an overflow is mathematically unreachable. Moreover,I can also consider a modern approach, leveraging Code LLMs (e.g., Code Llama), using them as a post-processing filter for the safe patterns that static analysis misses.
The source code and test cases are available in the GitHub repository: `https://github.com/Samin-Islam0312/IntOverflow2BufferOverflow`.

References

[1] L. Xu, M. Xu, F. Li, and W. Huo, "Elaid: detecting integer-overflow-to-buffer-overflow vulnerabilities by light-weight and accurate static analysis," *Cybersecurity*, vol. 3, no. 1, pp. 1–19, 2020.