32-bit x86 Vulnerability Check in Haskell

Team Members: Lianting Wang & Boning Li

October 2024

1. Goal

The goal of this project is to develop a tool in Haskell that identifies vulnerabilities in 32-bit x86 binaries, focusing on buffer overflows and memory mismanagement issues. This tool will help security researchers and developers analyze the compiled code and flag potential vulnerability patterns. The key focus is on detecting common vulnerabilities in legacy systems and software, ensuring that security risks are mitigated for older systems still in use. The tool will leverage functional programming principles to provide reliable and modular code for analyzing binaries.

2. Use Cases

2.1. Use Case 1: Displaying the Abstract Syntax Tree (AST)

The first use case focuses on generating and displaying the Abstract Syntax Tree (AST) from the disassembled 32-bit x86 binary code. The tool will parse the binary, convert it into human-readable x86 assembly instructions, and then construct an AST to represent the code's structure. This AST will give a hierarchical view of the program's control flow, function calls, and code blocks, making it easier to visualize the logic and structure of the binary. This use case is independent of any memory analysis, serving purely to aid in the understanding of the program's syntax and structure.

2.2. Use Case 2: Checking for Format String Vulnerabilities

Another important vulnerability the program will detect is format string vulnerabilities. These vulnerabilities occur when user-provided data is directly passed to a formatted output function (such as `printf`) without proper validation. The program will scan the disassembled code for unsafe usage of format functions and check whether the format strings are dynamically generated or come from external input. If such a pattern is found, the tool will flag the location in the code, providing developers with information on how to mitigate the risk.

2.3. Use Case 3: Inspecting Function Call Stack Integrity

The third use case is inspecting the call stack integrity in the binary, looking for anomalies such as corrupted return addresses or unusual stack frame patterns. These anomalies could indicate vulnerabilities like return-oriented programming (ROP) attacks. The program will provide a report highlighting sections of the binary where the call stack appears inconsistent, helping identify potential exploit vectors.

3. Outline of Initial Design

The project will be divided into several Haskell modules, each handling specific parts of the vulnerability check process. Here is an outline of how the modules will be structured:

3.1. BinaryParser Module

This module will be responsible for parsing 32-bit x86 binaries and extracting their disassembled code. It will use external libraries like `elf` or `pe` (depending on the file format) to load the binary, and will translate the machine code into readable x86 assembly instructions.

3.2. ASTGenerator Module

This module will generate an Abstract Syntax Tree (AST) from the disassembled binary code. The AST will provide a visual structure of the code's logic and control flow. This module focuses purely on representing the syntax of the binary, with no relation to memory or vulnerability checks.

3.3. VulnerabilityScanner Module

This module will analyze the disassembled code and identify potential vulnerabilities such as format string vulnerabilities and call stack anomalies. Higher-order functions will be used to compose complex scanning rules in a modular way.

3.4. FormatStringChecker Module

This module will focus on identifying format string vulnerabilities. It will parse the disassembled code for format string functions, checking whether they receive unvalidated external input, and flagging potential vulnerabilities.

3.5. ReportGenerator Module

After the vulnerabilities are detected, this module will generate a detailed report, highlighting the locations and types of vulnerabilities found. The output will be either a textual report in the terminal or an HTML report for easier viewing.

4. Testing

To test the program, we will generate a set of known 32-bit x86 binaries, some of which contain vulnerabilities like buffer overflows or unsafe memory access, and others that are clean and secure. We will use these binaries to evaluate the performance of the tool, ensuring that it correctly flags vulnerabilities without generating excessive false positives or false negatives. Unit tests will be written for each module, and we will perform integration testing to verify the functionality of the entire system. We will also test the tool with real-world binaries from open-source software to ensure robustness.