

System Programming Project Report

Serhat Turgut b221202056 Yusuf Okur b221202045

1. Introduction

This report documents the implementation and functionality of two system programming tools developed for this project: a **Memory Analyzer** and a **Code Security Checker**. These tools are designed to provide insights into process memory usage and identify potential security vulnerabilities in C source code, respectively.

2. Part 1: Memory Analyzer (Q1)

2.1 Overview

The Memory Analyzer is a utility that inspects the memory state of a running process. It leverages the Linux /proc filesystem to gather real-time information about a process's memory mappings and status.

2.2 Design and Implementation

The tool operates by parsing specific kernel interface files found in /proc/[pid]/.

- **Memory Map Analysis (print_memory_maps):**
 - **Source:** Reads from /proc/self/maps.
 - **Parsing Strategy:** The tool reads the file line-by-line and uses sscanf with the pattern %lx-%lx %4s ... to extract the start address, end address, permissions, and pathname.
 - **Segment Identification:**
 - * **Heap/Stack/VDSO:** Identified by specific pathnames like [heap], [stack], and [vds0].
 - * **Text (Code) Segment:** Identified as regions mapping the main executable with r-x (read-execute) permissions.
 - * **Data/BSS:** Identified as regions mapping the main executable (or anonymous) with rw- (read-write) permissions.
 - **Shared Libraries:** The tool iterates through the maps again to list all regions ending in .so, calculating their size in KB.
- **Memory Status Analysis (print_memory_status):**
 - **Source:** Reads from /proc/self/status.
 - **Metrics:** parses key-value pairs to extract:
 - * **VmSize:** Total virtual memory size.
 - * **VmRSS:** Resident Set Size (actual physical memory used).
 - * **VmStk, VmData, VmExe:** Specific usage for stack, data, and executable segments.

- **Efficiency Metric:** Calculates $(\text{VmRSS} / \text{VmSize}) * 100$ to indicate how much of the allocated virtual memory is actually in physical RAM.

3. Part 2: Security Checker (Q2)

3.1 Overview

The Security Checker is a static analysis tool designed to scan C source files for common security vulnerabilities and coding errors. It helps developers identify unsafe function usage and potential specific attack vectors.

3.2 Design and Implementation

The checker employs a token-based scanning approach rather than a full Abstract Syntax Tree (AST), making it lightweight and fast.

- **Tokenization (`code_parser.c`):** The `contains_token` utility ensures that matches are actual function calls and not substrings within other words, comments, or string literals.
- **Vulnerability Detection (`security_checker.c`):**
 - **Buffer Overflow Risks:** Flags usage of functions known to lack bounds checking: `strcpy`, `strcat`, `sprintf`, `gets`, and `scanf`.
 - **Command Injection:** Detects invocations of `system()` and `popen()` which can allow arbitrary command execution if inputs are not sanitized.
 - **Format String Attacks:** Specifically checks `printf` and `fprintf` calls to ensure the format string is a string literal. If a variable is passed directly as the format argument (e.g., `printf(user_input)`), it is flagged as a vulnerability.
 - **Thread Safety:** Identifies non-reentrant time functions such as `ctime`, `asctime`, `gmtime`, and `localtime`, recommending safer `_r` alternatives.
 - **Integer Overflows:** In extended mode, currently checks `malloc` calls for simple arithmetic patterns that might result in overflow/under-allocation.

4. Usage Guide

4.1 Prerequisites

- GCC Compiler
- Linux Environment (WSL or Native)
- Make

4.2 Building the Project

Navigate to the project root and run:

`make`

This will compile both the Memory Analyzer and Security Checker and place the executables in the `bin/` directory.

4.3 Running Memory Analyzer (Q1)

`./bin/mem_analyzer`

(Note: The tool analyzes its own memory layout to demonstrate functionality on the running process)

4.4 Running Security Checker (Q2)

`./bin/sec_checker tests/samples/vulnerable1.c`

5. Conclusion

The developed tools successfully demonstrate key system programming concepts. The Memory Analyzer shows deep interaction with the Linux kernel via the `/proc` pseudo-filesystem, while the Security Checker illustrates the principles of static code analysis and secure coding practices.