Software Requirements Specification

Secure Language Assembly Inspector Tool (SLAIT)

Team Members
Maria Linkins-Nielsen – mlinkinsniel2022@my.fit.edu
Michael Bratcher – mbratcher2021@my.fit.edu

Client & Faculty Advisor
Dr. Marius Silaghi – msilaghi@fit.edu

---

1. Introduction

1.1 Purpose

The purpose of this document is to specify the requirements for SLAIT, a web-based application designed to securely execute MASM x86 assembly code, record register/flag states at user-defined points, and present results in a clear, organized format. SLAIT eliminates the need for students to lower system security or configure complex debuggers locally, providing a reproducible and safe learning environment for low-level code inspection.

.

1.2 Scope

SLAIT is a web application that supports:

- **MASM code input:** Users paste or upload assembly code into a built-in editor.
- **Inspection definition:** Users select lines and specify registers/flags to capture.
- **Sandboxed execution:** Code runs inside an isolated container.
- **Result visualization:** Register/flag snapshots are displayed in a timeline view.
- **Error feedback:** Compiler/runtime errors are returned to the user in a structured form.

Out-of-Scope for Phase 1:

- Other language support (Java bytecode, C/C++)
- Persistent history
- Advanced runtime profiling beyond register/flag capture

1.3 Definitions, Acronyms, and Abbreviations

Term	Definition
MASM	Microsoft Macro Assembler – used for compiling x86 assembly
Inspection	A line number specified by the user where registers/flags will be captured
Snapshot	Captured state of CPU registers/flags at an inspection point
Flags	Processor status register (e.g., ZF, CF, SF, OF)

1.4 Overview

This document describes SLAIT's high-level functionality, functional and interface requirements, and performance expectations. It will guide frontend/backend development and ensure consistent implementation of security, usability, and correctness.

---

2. Overall Description

2.1 Product Perspective

SLAIT is a client-server system with three main layers:

- **Frontend:** Angular-based UI for editing code, selecting inspections, and viewing results.
- **Backend:** API that accepts code, triggers execution, and returns results.
- **Execution Sandbox:** Windows Container with MASM toolchain and register capture logic.

2.2 Product Functions

- **Code Viewing & Editing:** Load and display MASM source code.
- **Inspection Creation:** Allow users to define line numbers and choose registers/flags.
- **Code Execution:** Run code securely in a sandbox and capture requested states.
- **Result Visualization:** Present register/flag values per inspection in execution order.
- **Error Reporting:** Return compiler/runtime errors in a user-readable format.

2.3 User Characteristics

- **Primary Users:** Undergraduate students in Computer Architecture & Assembly courses (CSE 3120) with minimal prior exposure to MASM or containerization.
- **Secondary Users:** Faculty and teaching assistants demonstrating code behavior during lectures or labs.
- Users are expected to have a browser-compatible device and basic familiarity with editing and running code snippets.

2.4 Assumptions and Dependencies

- Users have internet access to reach the hosted SLAIT service.
- The host server can run Docker containers securely and with sufficient resources.
- MASM toolchain functions reliably in the container.
- Execution timeouts prevent infinite loops from locking system resources.

---

3. Specific Requirements

3.1 Functional Requirements

3.1.1 Code Input and Inspection Management

- **3.1.1.1 Upload Code:** The system shall allow the user to upload ".asm" files.
- **3.1.1.2 View Code:** The system shall display the MASM code.
- **3.1.1.3 Create Inspection:** The system shall allow users to select a line number and specify which registers/flags to capture.
- **3.1.1.4 Delete Inspection:** The system shall allow users to remove inspections prior to execution.

3.1.2 Code Execution and Data Capture

- **3.1.2.1 Execute Code:** The system shall compile and execute MASM code in a container.
- **3.1.2.2 Capture State:** The system shall record registers (EAX, EBX, ECX, EDX, ESI, EDI, ESP, EBP) and Flags at each inspection.
- **3.1.2.3 Return Results:** The system shall format captured data into a returnable list.
- **3.1.2.4 Handle Errors:** The system shall return descriptive error messages for compilation or runtime failures.

3.1.3 Visualization and Analysis

- **3.1.3.1 Display Results:** The system shall present captured register/flag states in a timeline or table.
- **3.1.3.2 Highlight Differences:** The system should highlight changes in register values between inspections.
- **3.1.3.3 Support Multiple Runs:** The system shall allow users to re-run code and compare results.

3.2 Interface Requirements

3.2.1 Graphical User Interface

- The system shall display a syntax-highlighted code viewer.
- The system shall visually mark lines that contain inspections.
- The system shall display results in a tabular format with columns for each register/flag.
- The system shall display errors in a separate panel with clear indicators.

3.3 Performance Requirements

- The frontend shall update the UI (e.g., enable/disable “Run” button) within **100 ms** of state changes.
- The system shall prevent excessive memory/CPU use by killing execution after a configurable timeout.