# Minimal Debugger Technical Documentation

## 1. Project Overview

Minimal Debugger is a C++ application that allows users to trace and control the execution of another process. It leverages the Linux `ptrace` API to observe and control the execution of the child process, examine memory, and manipulate registers.

### Core Mechanisms

- **Ptrace API:** The foundation of the debugger, used for `PTRACE_TRACEME` (child), `PTRACE_CONT`, `PTRACE_SINGLESTEP`, `PTRACE_PEEKUSER`, and `PTRACE_POKETEXT`.
- **Software Breakpoints:** Implemented by replacing the instruction at the target address with the `int3` opcode (`0xCC`). When the CPU executes this opcode, it raises `SIGTRAP`, returning control to the debugger.
- **Register Manipulation:** Reading and writing CPU registers (`struct user_regs_struct`) to track the instruction pointer (`RIP`) and inspect program state.

## 2. Architecture

The application is structured into three main layers: the CLI/REPL, the Debugger Controller, and the Low-Level Wrapper.

### Control Flow

1. **Initialization:** `main.cpp` instantiates the `Debugger` class and starts the REPL.
2. **Command Parsing:** The `Debugger` reads user input, parses commands (like `load`, `break`, `step`), and delegates them to the `ptrace_wrapper`.
3. **Process Control:** The `ptrace_wrapper` manages the child process via `fork()` and `execv()`.
4. **Event Loop:** When `run` or `continue` is issued, the debugger waits for signals from the child (usually `SIGTRAP` from breakpoints or steps) and updates its internal state accordingly.

## 3. Module Descriptions

### 3.1. Main & CLI

- **Source:** `src/cli/main.cpp`
- **Purpose:** Entry point.
- **Functionality:** Creates the `Debugger` instance and calls its `run()` method.

### 3.2. Debugger (Runner)

- **Source:** `src/runner/debugger.h`, `src/runner/debugger.cpp`
- **Key Class:** `Debugger`
- **Purpose:** The high-level controller of the application.
- **Responsibilities:**
  - **REPL:** Implements the `repl()` loop to accept user input.
  - **Command Dispatch:** Parses string commands and calls specific methods on the `ptracer` object.
  - **UI/Output:** Handles printing prompts, help messages, and formatting register output.

### 3.3. Ptrace Wrapper

- **Source:** `src/runner/ptrace_wrapper.h`, `src/runner/ptrace_wrapper.cpp`
- **Key Class:** `ptrace_wrapper`
- **Purpose:** encapsulation of the `ptrace` system call and low-level process management.
- **Responsibilities:**
  - **Process Lifecycle:** `load_binary` forks a new process. The child calls `ptrace(PTRACE_TRACEME, ...)` and then `execv`.
  - **Register Access:** Wraps `ptrace(PTRACE_GETREGS, ...)` to fetch CPU state.
  - **Stepping Logic:** `execute_single_step` handles the complex logic of stepping over a breakpoint (disable BP -> step instruction -> re-enable BP).
  - **Memory Access:** Uses `ptrace(PTRACE_PEEKTEXT/POKETEXT)` to read/write memory (used for breakpoint injection).
  - **Composition:** Owns an instance of `BreakpointManager`.

### 3.4. Breakpoint Manager

- **Source:** `src/managers/breakpoint_manager.h`, `src/managers/breakpoint_manager.cpp`
- **Key Class:** `BreakpointManager`
- **Purpose:** Manages the collection of active breakpoints.
- **Responsibilities:**
  - **Storage:** Maintains a list (`std::vector`) of `Breakpoint` structures.
  - **Lookup:** Allows retrieving breakpoint details by memory address.
  - **CRUD:** Adds and removes breakpoints from the tracking list (note: actual memory modification is delegated back to the wrapper/runner in this architecture).

### 3.5. Utils

- **Source:** `src/utils/parsers.h`, `src/utils/parsers.cpp`

- **Purpose:** Helper functions for string and data manipulation.
- **Functionality:** Parsing hexadecimal strings (addresses) and splitting command strings.

## 4. Key Algorithms

### Breakpoint Handling (The "Int3" Technique)

1. **Set:**
    - Read 8 bytes at target `address`.
    - Save the bottom byte (original instruction) into the `Breakpoint` struct.
    - Replace the bottom byte with `0xCC` (int3).
    - Write back the modified word.
2. **Hit:**
    - Process stops with `SIGTRAP`.
    - Instruction Pointer (`RIP`) is now `address + 1`.
    - Debugger decrements `RIP` by 1 so it points back to the start of the instruction.
3. **Step Over:**
    - Restore original byte at `address`.
    - `ptrace(PTRACE_SINGLESTEP, ...)` (execute the original instruction).
    - Re-write `0xCC` at `address` (re-enable breakpoint).

## 5. Build Process

- **Compiler:** `g++` (Standard C++17)
- **Tool:** `make`
- **Artifacts:**
    - `debugger`: The main executable.
    - `dummy/*.o`: Object files for test targets.
    - `build/*.o`: Object files for the source code.

## 6. Testing

The project uses a custom integration test approach defined in `TESTS.md`. * **Mechanism:** Pipes predefined input files (`input_*.txt`) into the debugger and asserts behavior. * **Targets:** Simple assembly programs in `dummy/` (e.g., `simple.s`, `loop.s`) provide predictable execution flows for verification.