Technical Report: Minimal Debugger (Minidbg) Implementation

Student Name: Jaskaran Singh
Date: 17 Dec 2025

1. Introduction

This project involves the design and implementation of a minimal debugger ("Minidbg") integrated into a custom UNIX-style shell. The debugger acts as a parent process capable of loading, controlling, and inspecting the state of a child process without relying on external tools like GDB. This implementation fulfills the requirements of **Lab 2**, demonstrating core OS-level debugging primitives on macOS.

2. System Architecture

The debugger operates on the **Parent-Child Process Model**, leveraging specific operating system API calls to bridge the two processes.

2.1 The Debug Loop

Unlike a standard shell that simply waits for a child process to finish, the debugger enters an interactive **Event Loop**:

1. **Fork & Trace:** The parent forks a new process. The child explicitly calls `ptrace(PT_TRACE_ME)` to allow monitoring before calling `execvp` to load the target program.
2. **Wait:** The parent waits for signals (like `SIGTRAP`) from the child using `waitpid`.
3. **Control:** When the child pauses, the parent reads user commands (e.g., `regs`, `continue`) and modifies the child's state accordingly.

2.2 macOS vs. Linux Primitives

While standard Linux debugging uses `ptrace` for almost all operations (including register access), macOS (based on the XNU kernel) requires a hybrid approach:

- **Process Control:** `ptrace` is used for stopping, resuming, and single-stepping (`PT_STEP`).
- **State Inspection:** The **Mach Kernel API** is required to read CPU registers and modify protected memory regions, as standard `ptrace` requests for these are deprecated or non-functional on macOS.

3. Implementation Details

3.1 Software Breakpoints

Breakpoints were implemented using the standard `INT 3` interrupt injection technique.

- **Setting a Breakpoint:**
  1. The debugger reads the 4 bytes of code at the target address.
  2. It saves this original data to a `Breakpoint` struct.
  3. It overwrites the first byte at that address with the opcode `0xCC` (INT 3).
- **Handling the Trap:** When the CPU executes `0xCC`, it raises a `SIGTRAP` signal, pausing the child process and notifying the parent.
- **Resuming Execution:** To continue past the breakpoint, the debugger temporarily restores the original instruction, single-steps the CPU (`PT_STEP`) to execute it, and then re-writes the `0xCC` trap to re-enable the breakpoint.

3.2 Register Inspection (`regs`)

To satisfy the requirement of inspecting CPU state, the debugger uses the Mach API:

1. **Task Port:** `task_for_pid` retrieves the kernel task port for the child process.
2. **Thread State:** `thread_get_state` populates an `x86_thread_state64_t` structure, allowing access to registers like `RIP` (Instruction Pointer) and `RSP` (Stack Pointer).

3.3 Memory Inspection (`peek`)

As an optional extension, the `peek` command allows the user to examine raw memory.

- **Mechanism:** It uses `ptrace(PT_READ_D, ...)` to read a word of data from the child's address space at a user-specified hexadecimal address.

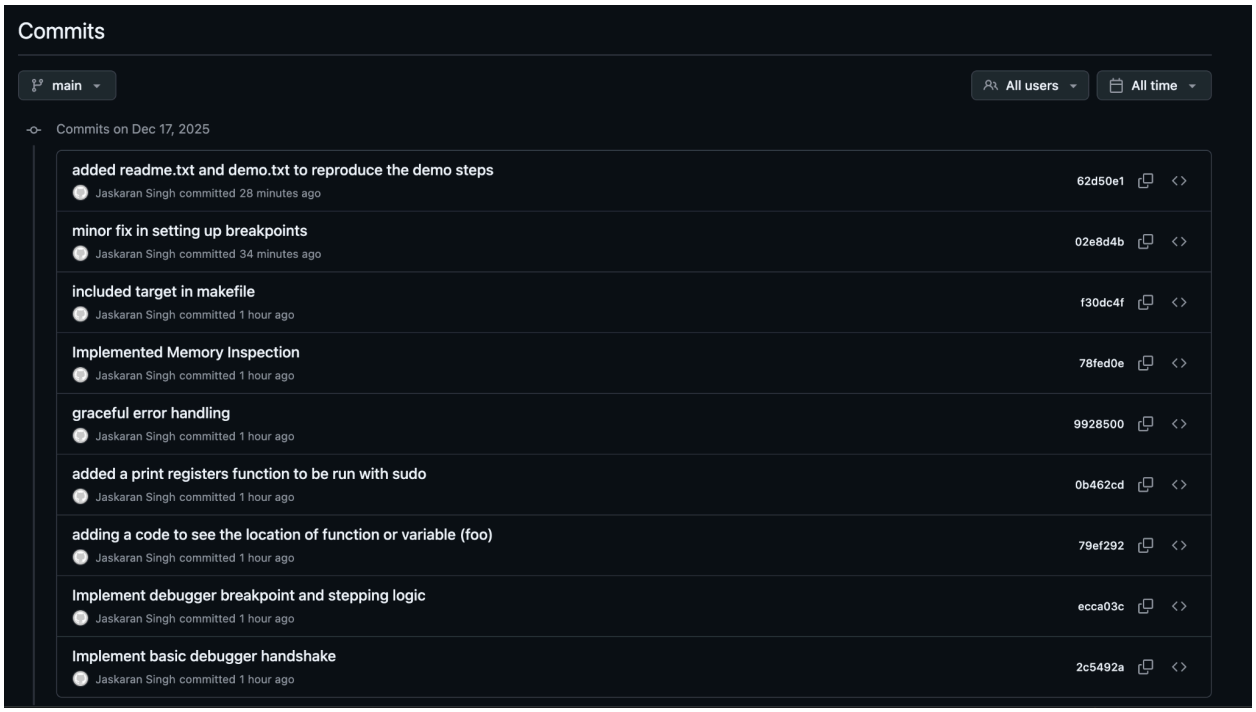
4. Development Workflow

Per project requirements, a structured Git workflow was utilized to ensure incremental development. Feature was developed on dedicated branch and merged into main only after verification.

Branching Strategy:

- feature/debug: Implementation of the debugger.

Commit History Visualization:



5. Key Challenges & Solutions

5.1 Challenge: Read-Only Text Segments

- **Problem:** On macOS, the text segment (where executable code resides) is marked as Read-Execute (RX). Attempting to write the `0xCC` trap using standard `ptrace` or `memcpy` failed with "Invalid argument" or access violations.
- **Solution:** The implementation uses **`mach_vm_protect`** to temporarily change the memory permissions of the target page to `VM_PROT_COPY` (Read/Write/Copy). This allows the debugger to write the trap using **`mach_vm_write`** before restoring the permissions to Read-Execute.

5.2 Challenge: Address Space Layout Randomization (ASLR)

- **Problem:** The operating system randomizes the load address of the binary for security, meaning static addresses (e.g., from a symbol table) are invalid at runtime.
- **Solution:** The user workflow was adapted to run the target program once to reveal its dynamic function addresses (e.g., printing `&foo`) before setting breakpoints.

5.3 Challenge: Privileged Access

- **Problem:** The call to `task_for_pid` failed with "Access Denied" because standard processes cannot control other tasks.
- **Solution:** The debugger must be executed with root privileges (`sudo`), granting it the necessary Mach capabilities to inspect and control child processes.

6. Testing & Verification

The debugger was verified against a suite of 7 specific test cases covering all functional requirements and the optional extension. The tests were executed on macOS to ensure compatibility with the Mach Kernel API and ASLR mechanisms.

1. Target Loading & Execution

- **Objective:** Confirm the debugger can launch a child process under tracing mode.
- **Verification:** Verified using `debug ./target`. Confirmed that the child process starts, prints its initial PID, and pauses immediately, waiting for debugger commands.

2. Software Breakpoint Injection

- **Objective:** Confirm the debugger can modify the text segment of a running process.
- **Verification:** Verified using `break <address>` (e.g., `break 0x100...`). Confirmed that the debugger successfully bypassed macOS memory protection (RX) using `mach_vm_protect`, wrote the `0xCC` trap, and reported "Breakpoint set" without "Invalid Argument" errors.

3. Execution Control & Trap Handling

- **Objective:** Confirm the debugger can resume execution and catch signals.
- **Verification:** Verified using `continue`. Confirmed that the target program runs until it hits the injected trap, raising `SIGTRAP`, which the debugger catches via `waitpid` to display "Hit breakpoint!".

4. Register Inspection

- **Objective:** Confirm access to the child's CPU state.
- **Verification:** Verified using `regs` while paused at a breakpoint. Confirmed that the `RIP` (Instruction Pointer) matched the breakpoint address and that valid stack pointers (`RSP`) were retrieved via the `thread_get_state` Mach API.

5. Memory Inspection (Extension)

- **Objective:** Confirm the ability to read raw data from the child's address space.
- **Verification:** Verified using `peek <address>`. Confirmed that the debugger returned the correct 4-byte hexadecimal value stored at the function address, satisfying the optional memory inspection requirement.

6. ASLR Adaptation

- **Objective:** Confirm the user workflow handles Address Space Layout Randomization.
- **Verification:** Verified by running `debug ./target` multiple times. Confirmed that `ADDRESS_OF_FOO` changed on every run, and that breakpoints only functioned correctly when the user utilized the address from the *current* session.

7. Privilege Enforcement & Error Handling

- **Objective:** Confirm graceful failure when required permissions are missing.
- **Verification:** Verified by running `./shell` without `sudo`. Confirmed that `regs` and `break` commands printed descriptive error messages (e.g., "Failed to get task for pid") instead of crashing the shell.

<---------- END OF REPORT ---------->