Lab 2: Mini-Debugger: Technical Report

2025MCS2121 Gaurav Jain
2025MCS2122 Aman Singh

17 December 2025

1 Introduction

1.1 Problem Statement and Scope

This project entails the development of a minimal debugger for ELF binaries on the Linux operating system. The primary objective is to load, control, inspect, and step through processes using OS-level debugging primitives, specifically avoiding high-level tools like GDB. The debugger is required to support breakpoint management, single-stepping, register inspection, and status reporting.

1.2 Design Philosophy

The system is designed as a lightweight command-line interface (CLI) tool. It operates by forking a child process to execute the target binary while the parent process acts as the tracer. This architecture leverages the Linux `ptrace` API for process control and memory manipulation, ensuring strict adherence to the non-functional requirement of not corrupting the debugger's state.

2 System Architecture

The debugger is modularized into five core components, managed by a central event loop in `main.c`.

2.1 Process Control (`debugger.c`)

The debugger initiates the session using a standard `fork()` and `exec()` pattern.

- **Child Process**: Executes `ptrace(PTRACE_TRACEME, ...)` to allow the parent to trace it, then calls `execl()` to replace its memory image with the target program.
- **Parent Process**: Uses `waitpid()` to synchronize with the child. It issues `ptrace` commands to control execution (e.g., `PTRACE_CONT`) and intercepts signals (like `SIGTRAP`) generated by breakpoints.

2.2 Breakpoint Management (`breakpoints.c`)

This module manages a data structure of active breakpoints. The key challenge is safely modifying the instruction stream of the running process without permanently corrupting the binary.

- **Insertion**: The debugger reads the instruction word at the target address using `PTRACE_PEEKTEXT`. It preserves the least significant byte (LSB) in the `breakpoint` struct and replaces it with the `INT 3` opcode (`0xCC`).
- **Restoration**: To remove a breakpoint or resume execution, the original byte is restored using bitwise operations: `(data & ~0xFF) | original_byte`.

3 Technical Implementation

3.1 The Breakpoint Mechanism

The core complexity lies in handling the `SIGTRAP` signal when a breakpoint is hit.

1. When the CPU executes `0xCC`, it halts the child process and sends `SIGTRAP` to the parent.
2. The parent inspects the Instruction Pointer (`RIP`) via `PTRACE_GETREGS`.
3. Since the CPU advances `RIP` after fetching the instruction, the `RIP` will be one byte past the breakpoint address.
4. The debugger detects this condition: `if (regs.rip - 1 == bp->addr)`.
5. To resume, the debugger must:
    - Restore the original instruction byte.
    - Decrement `RIP` by 1 (`regs.rip -= 1`) so the original instruction is executed.
    - Update the child's registers via `PTRACE_SETREGS`.

```c
void handle_breakpoint(pid_t pid, struct breakpoint *bp)
{
  struct user_regs_struct regs;
  ptrace(PTRACE_GETREGS, pid, NULL, &regs);

  if (regs.rip - 1 == (unsigned long)bp->addr)
  {
    // Restore original code and reset instruction pointer
    remove_breakpoint(pid, bp);
    regs.rip -= 1;
    ptrace(PTRACE_SETREGS, pid, NULL, &regs);
    printf("Hit breakpoint at %p\n", bp->addr);
  }
}
```

Listing 1: Handling Breakpoint Hit

3.2 Single Stepping

Single-stepping is implemented via `ptrace(PTRACE_SINGLESTEP, ...)`. This places the processor in a mode where it executes one machine instruction and then raises a `SIGTRAP`, returning control to the debugger. This is essential for fine-grained analysis of control flow.

3.3 Register Inspection

Register states are retrieved using `ptrace(PTRACE_GETREGS, ...)`, which populates the `sys/user.h` defined `user_regs_struct`. The debugger outputs critical 64-bit registers (RIP, RSP, RAX, RBX, etc.) to standard output, allowing users to verify variable values and stack pointers during execution.

4 Testing and Validation

The project includes a custom unit testing framework defined in `tests/test_framework.h`. The framework macros (`TEST`, `ASSERT_EQ`) facilitate automated validation of:

- **Data Structures**: Verifying breakpoint struct initialization and flag management.
- **Integration**: An automated test creates a dummy C program, compiles it, and verifies that the debugger can attach and control it without crashing.
- **Status Macros**: Validates correct interpretation of `WIFEXITED` and `WIFSTOPPED` macros across different OS environments.

5 Conclusion

The Mini-Debugger successfully meets all functional requirements set forth in Lab 2A. By directly manipulating process memory and execution flow via `ptrace`, the tool provides a robust environment for binary analysis. The implementation demonstrates a solid understanding of low-level system programming concepts, including signal handling, opcode injection, and register manipulation.

Appendix

A1. Demo Screenshot

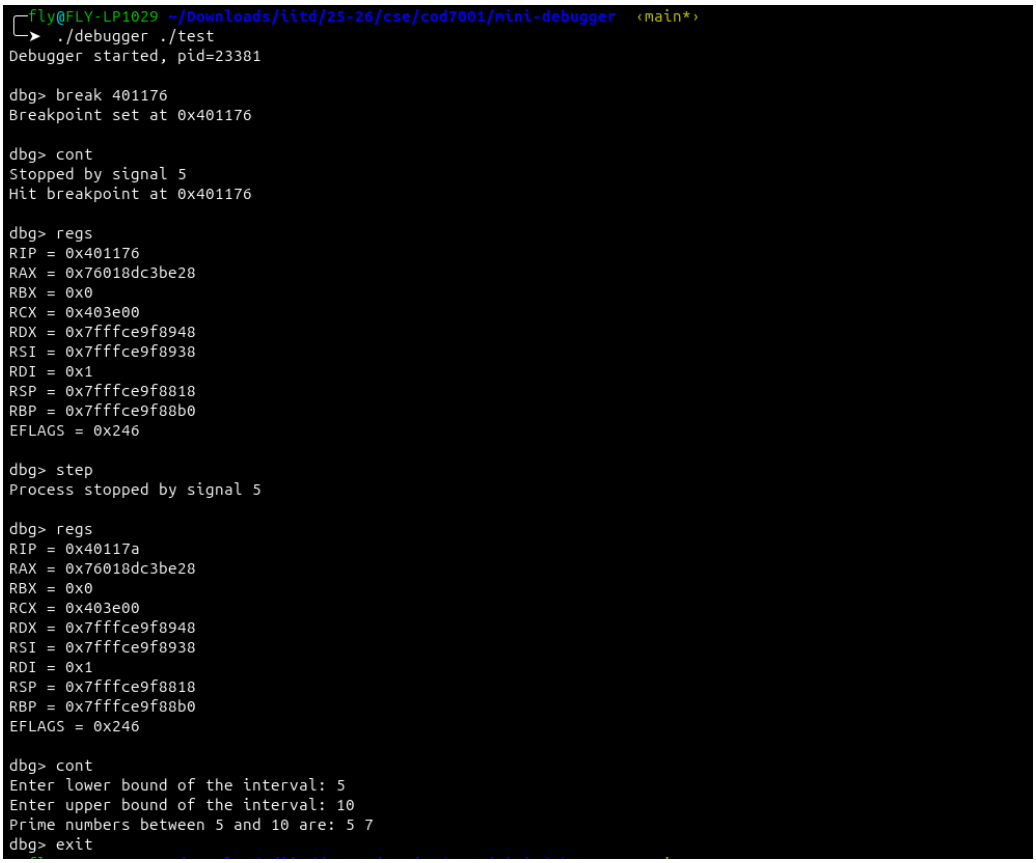

Figure 1: Debugger Interactive Session

A2. Commit History

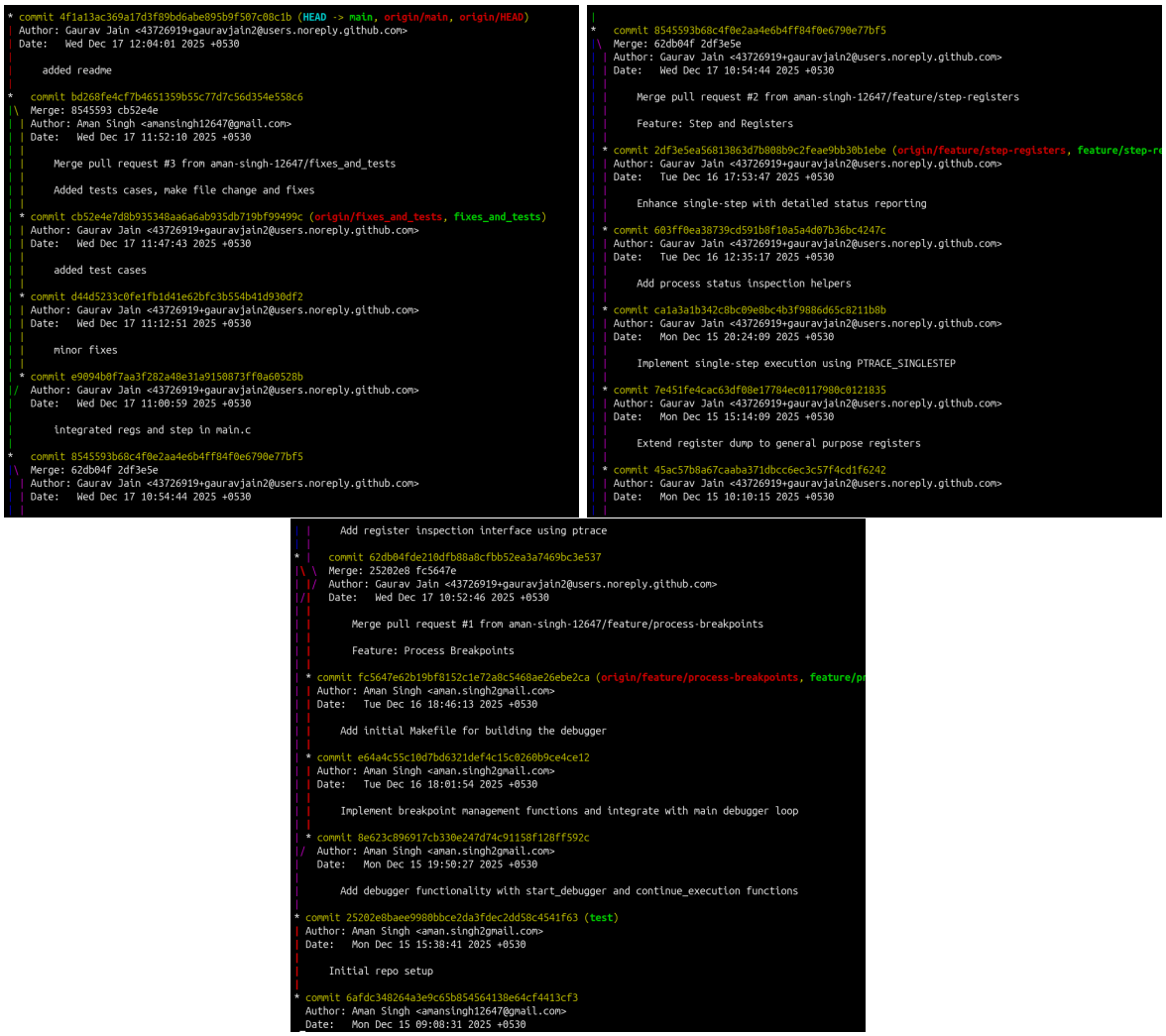

Figure 2: Project Commit History