Lab: Debugging C Programs Using GDB

DS2040, Satyajit Das

February 13, 2025

1 Introduction

Your task is to:

1. Use **GDB** to *debug* each binary.
2. Diagnose the underlying issues.
3. *Write a corrected version* of the program in C that compiles *and* runs correctly (no segmentation faults or incorrect results).

2 Lab Setup

Files Provided in the tar file

- `mystery1` (compiled binary; the debug information is kept intact. "layout next" will show the actual program)
- `mystery2` (compiled binary; debug information stripped)
- `mystery3` (compiled binary; debug information stripped)
- `mystery4` (compiled binary, debug information stripped)

No source files are provided. You will have to **reverse-engineer**, **debug**, and **recreate** the `.c` code from what you discover with GDB (and from observed runtime behavior).

3 Lab Outline

3.1 Part 1: Analyzing `mystery1`

Hints/Clues

- **Pointer Arithmetic Issue:** The program might be *reading* or *writing* one element too far.
- Observed runtime: might crash intermittently or produce strange output.

Instructions

1. Run `mystery1` without arguments first. See if it crashes or outputs an error.
2. Start GDB:
   `$ gdb ./mystery1`
3. **Look for main:** `break main`, `run`, `step` into suspicious calls or loops.
4. Check pointer variables with `print` commands, e.g.:

```
(gdb) print p
(gdb) print *p
```

5. Once you identify the bug, try to deduce **what the code was supposed to do**. Write a new `mystery1_fixed.c` with the corrected pointer usage.
6. Compile, run, and test your `mystery1_fixed` to ensure it produces correct results (or at least no crash).

3.2 Part 2: Analyzing `mystery2`

Hints/Clues

- This program likely has an **array out-of-bounds** issue.
- It might read from or write to an index beyond the array length.
- You could see memory corruption or a segmentation fault.

Instructions

1. Run `mystery2` in GDB.
   `$ gdb ./mystery2`
2. Set breakpoints inside loops or suspect functions (use `disassemble main` to guess the loop area or `info functions` to find function boundaries).
3. Step through carefully, watch variables that track array indices.
4. Identify the improper index or loop boundary condition.
5. Create `mystery2_fixed.c`, ensuring you handle the array within correct boundaries.
6. Recompile and check behavior.

3.3 Part 3: Analyzing `mystery3`

Hints/Clues

- Potentially a **function call mismatch**: maybe the program calls a function with the wrong number/type of parameters.
- Could also be a mismatch in **return type** or function prototype vs. definition.

Instructions

1. Execute `mystery3` to see if it prints partial results or crashes.
2. In GDB, break at `main` and step into each function call.
3. Use `info args` in each function to see the actual arguments.
4. Compare with how they're used. If, for example, a function expects three parameters but you only see two in `info args`, that's a clue.
5. Investigate the function's prototype vs. usage. Possibly they used `int` but it's returning `char*`, etc.
6. Write a corrected `mystery3_fixed.c` with a proper function signature and calls.

3.4 Part 4: Analyzing `mystery4`

Hints/Clues

- This program may combine **pointer arithmetic** and **array passing** to a function, or have nested function calls with invalid pointer usage.
- Possibly more than one bug.

Instructions

1. Again, `gdb ./mystery4`, break `main`, `run`.
2. Look for suspicious pointer increments, out-of-bounds pointer usage, or incorrectly declared function parameters.
3. Watch for loops that manipulate pointers.
4. If it segfaults, use `backtrace`, then examine local variables and memory near the address that caused the crash (`x/10x address`).
5. Re-implement the logic in `mystery4_fixed.c`.

4 Additional Tips for Debugging

- **Use GDB watchpoints:** `(gdb) watch someVariable` to break whenever `someVariable` changes.
- **Examine memory:** `x/16bx &array` to see 16 bytes in hex from `array`'s address.
- **Disassemble:** If you get stuck, do `disassemble /m main` to see assembly interleaved with (best-guess) source lines. This can reveal function calls and indexing logic.
- **Step & Next carefully:**
  - `step` goes into function calls.
  - `next` goes over them.

- `finish` runs until the current function returns.

- **Document your process:** Write notes on how you discovered each bug, which lines in your final source code fix them, etc.

By reconstructing and fixing each program, you'll practice real-world skills that blend **reverse-engineering** and **C debugging**. Have fun!