Advanced - Debugging

EECS 201 Spring 2026

Submission Instructions

This assignment is an "online assignment" on Gradescope, where you will attach your files and answer some questions.

Presentation/Checkoff

In addition to submitting this assignment, you will need to attend a lab session after the deadline to discuss your work with the course staff. Refer to the course schedule for the window for doing so for this assignment.

As part of the discussion, be prepared to do the following:

- Explain your approach for the assignment. What documentation did you read, and how did you find it? What did you try that didn't work, and what was your process for figuring out how to fix it?
- Summarize what you learned from the assignment.
- Discuss how you plan on implementing the tools in this assignment and what you learned in your future projects.
- Apply your approach for solving the problems in this assignment to a new problem that you have not seen before.

I suggest keeping these in mind while doing the assignment, and writing down your thoughts so that you can review them before the checkoff.

Your overall score for the assignment will be a combination of your submission score and the presentation/checkoff score. Refer to the syllabus for the details.

Preface

In this assignment you'll be provided yet another zipped archive containing some starter files.

`https://www.eecs.umich.edu/courses/eecs201/sp2026/files/assignments/adv-debug.tar.gz`

1 Attaching to a running process (5)

This part may or may not work on macOS: try it on the course server. Occasionally you may want to debug an already running process with GDB or LLDB. Perhaps it's some sort of daemon process that you want to catch in the act or maybe it's some interesting setup that you have concocted *cough cough*. Regardless, GDB and LLDB have the ability to attach to already running processes when provided with the process ID ("PID") of a process you want to debug (and in the case of LLDB, it can also even attach to a process via the program's name!).

In `adv-debug/attach` there is a program called `revd` that will reverse strings provided to it and log them. It is intended to run in the background, using a special type of file called a *named pipe* or *FIFO* (like a pipe used to string commands together, but accessible as a file) as input. It uses the FIFO as a form of inter-process communication (IPC): a process that wants to communicate will write to the FIFO file and `revd` will read from it.

The Makefile has targets for building and running the application. `test-producer.sh` is a sample script that communicates with `revd`, forever providing it with a random word every second (until you `^C`). Try running the `run` target and then run `test-producer.sh`. In another terminal run `$ tail -f revd.log` to continuously print out updates to the log file. You should see that the `revd` process is chugging along (for extra fun, run some more instances of `test-producer.sh` or write another program/script that produces more input for `revd`).

Now onto the debugging part: the goal of this exercise is to *attach* GDB or LLDB to this running process. There's

multiple ways of going about this: I'll leave it up to you to review the GDB/LLDB manual or manpage or other random resources on the internet.

Notes:

- **If you are using WSL, the Windows filesystem (i.e. stuff under /mnt/c) does not support FIFOs. This can only be done on Unix filesystem side (i.e. everything not /mnt/c).**
- **Don't do this bullet on the course server: you don't have permissions to, and it's already set for you.** Newer versions of the Linux kernel now by default have security measures in place that prevent the tracing of processes, which GDB does to attach to a process. If you are unable to attach to the process and there is some warning/message about `ptrace` not being permitted, or some other permission denied,

  run `$ sudo sh -c "echo 0 > /proc/sys/kernel/yama/ptrace_scope"`. This will invoke the `sh` shell under `root` and write a 0 into the `/proc/sys/kernel/yama/ptrace_scope` file which serves as the way to configure the security level for tracing. **Don't do this on the course server! You don't have the permissions to modify system-level settings on my server ;p**
- `make run` will run `revd`.
- `make kill` will kill `revd` (since it has been backgrounded it's a tad more involved to stop it).
- If you're doing this on the course server, please be nice and kill your `revd` when you're done. The server will kill running `revd` processes that are older than 5 minutes to free up resources, so don't be alarmed if you find them terminated.
- `pgrep`, `pidof`, and `ps a` can help you find the PID of `revd`.
- If you're on the course server, `ps u` might be of greater use.

☐ What command-line commands did you run in the process of attaching GDB or LLDB to the running `revd` process? Be specific!

☐ If you look at the `revd.log` file you'll notice that it isn't reversing the string properly. Fix this issue and submit the fixed version. (By the way, string reversal is a pretty common algorithmic interview question)

2 Bisecting a Regression (5)

Sometimes a program that used to work just stops working, and you're left staring at a pile of commits with no idea which one is responsible. The commit messages won't save you (they never advertise the bug they introduced), and reading every diff by hand is slow and easy to get wrong. The good news: as long as you have one commit where things worked and one where they don't, Git can binary-search the history in between and pin down the exact commit that broke things for you, using `git bisect`.

In `adv-debug/bisect` there is `mini-calc`, a small C++ program that evaluates integer arithmetic expressions (`+ - * / %`, parentheses, and unary minus). It's a real Git repository with history: it shipped a known-good release, tagged `v1.0`, where everything worked. Many commits later something is off, and the project's own tests (`make test`) still pass, so nobody noticed when it slipped in.

Build it and take it for a spin:

```
make
./eval "2 + 3 * 4"      (should print 14)
```

The folder also contains `test.sh`, a script that checks for the regression. Keep it in the repository root and don't commit it: it's listed in `.gitignore` on purpose so that it stays put while `git bisect` swaps the tracked files around underneath you.

Now onto the actual hunt: **figure out the exact commit that introduced the bug, and then fix it.**

Notes:

- Read through `git bisect --help` and the "Debugging with Git" section of Pro Git, which walk through the whole workflow.
- `v1.0` is your known-good starting point, and the current tip (`HEAD`) is bad.

☐ What does `./eval "8 - 3 - 2"` print on the current (broken) code, and what should it print? Give both numbers.

☐ What exact sequence of commands did you run to bisect from `v1.0` to `HEAD` and arrive at the first bad commit? I should be able to run these commands as written.

☐ What is the hash of the first bad commit, and what is its commit message (the subject line)?

☐ Paste your `git bisect log`.

☐ Fix the bug and submit the fixed file. Show that `./eval "8 - 3 - 2"` now prints the right answer. (Copy the relevant working code from the last good commit to the tip and commit the fixed code.)