Introduction

In this micro project, you will explore the basic features of the GNU debugger, or as more commonly known, GDB. GDB is a standard debugging program that is included by default in all Linux platforms, and is a crucial tool in any C/C++ programmers arsenal. Before starting, please brush up on C concepts, such as pointers and dynamic memory allocation. We hope you have fun!

The main goal of getting you familiar with GDB is so that you can become more pro-efficient C programmers, and have a tool to debug your code before asking for help. *We expect you to have tried debugging your code before asking help in office hours for the next few assignments*

Expectations

The exercises in this guide will make you familiar with basic GDB features including

- Running a program in GDB
- Setting and modifying breakpoints
- Examining current state of the register file
- Examining the stack

you will be answering some questions along the way in the provided text file.

Make sure that you do this project **INDIVIDUALLY**. No form of collaboration is permitted. Some questions are fairly open-ended and can have many forms of correct answers.

Preparation

1. Make sure you are using either a 32-bit or 64-bit Linux distro
2. Install gdb and gcc if not already present. You can run: `sudo apt-get install build-essential` if you are running a debian based OS.
3. Make sure you have glibc installed on your computer.

What have we provided you?

- **gdb_example1.c** The buggy code file that will be used in part 1
- **gdb_example2.c** The buggy code file that will be used in part 2
- **buggyBST.c** A buggy binary tree implementation, the final exercise
- **Makefile** A makefile to compile the code
- **micro-project2.txt** The text file you will be writing your answers to
- **GDB-Intro.pdf** A short GDB introduction

PART 1

Read the GDB introduction slide before moving on. Now, open the Makefile we have provided you with, and understand what it does before moving on. You may need to search for some of the terms on the Internet.

Question 1: What does the ‘-g‘ flag present in the gcc command do?

Now take a look at the **gdb_example1.c**, make sure you understand what it does. Now compile the first example by running the following command in the terminal.

```
   $ make part1
```

The compilation should be successful. Don't worry about any warnings you might see at this point. Try running the program using the following command:

```
   $ ./gdb_example1.c
```

You should get a segmentation fault. This is a very common problem with code written in low level languages such as C. Try running the following command:

```
   $ dmesg | tail -1
```

What information can you get from the *dmesg* command? Is there enough information there to help you debug your program? We will now use GDB and see if we can do better. Run the following command:

```
   $ gdb ./gdb_example1.c
```

Since our program does not take in any command line arguments, we could use the above command. However in the case you wanted to pass command line arguments to your program, you could have used the following command inside the gdb shell:

```
   (gdb) set args <args ...  >
```

Now, that you have the program loaded into GDB, you can run it by:

```
   (gdb) run
```

You should see a message that looks something like:

```
Program received signal SIGSEGV, Segmentation fault.
0x0804842c in print_scrambled (message=0x0) at gdb_example1.c:7
7       printf("%c", (*message)+i);
```

Now, let us print the backtrace of our program flow to find exactly where the segmentation fault occurred:

```
(gdb) where
(gdb) where full
```

You could also use the backtrace command:

```
(gdb) backtrace
(gdb) backtrace full
```

The ‘full’ essentially shows you more information regarding each function call, like the local variables that were used. To get a better idea of the arguments passed into the current function, use the following commands:

```
   (gdb) info args
```

We can also look at the state of the stack and register by using the following commands:

```
(gdb) info stack
(gdb) info register
```

Question 2: What is the value of the ‘char*’ argument that was passed last into the `print_scrambled()` function? Can you guess what caused the segmentation fault based on this information?

Question 3: What is the address of the stack pointer and frame pointer before the seg fault? Based on this information, can you figure out where the ‘char*’ argument is stored on the stack before the segmentation fault occurred?

Hint: A general way of printing a value at a certain memory address is to use the following command:

```
   (gdb) x/nfu
```

Here, ‘n’ is the number of units (e.g. 4), ‘f’ is the format (e.g. ‘x’ for hex), ‘u’ is the unit (e.g. ‘w’ for 32-bit word).

Finally Fix the problem in the **gdb_example1.c** file that was causing the segmentation fault. Change **ONLY** the `print_scrambled()` to make the program print `good_message` and silently ignore `bad_message`.

PART 2

We will now focus on the breakpoint portion of this tutorial. Take a look at the `gdb_example2.c` file, and compile is using the below command. Make sure to take a look at the compiler generated warnings.

```
   $ make part2-mem-protection
```

Run `gdb_example2`, and as you have probably guessed, you get another segmentation fault! If you were able to follow and understand part 1, you should be able to answer the following questions quite easily.

Question 4: Make sure you answer all the parts.

1. In which function did the segfault occur?
2. Which line of the source code (unmodified) caused the segfault?
3. Who is the caller of this function?
4. What is the PC address at which the segfault occurred?

Fix the minor bug that was causing the segfault in the `gdb_example2.c` file. Use the following commands to run re-compile the fixed source:

```
$ make clean
$ make part2
```

Let‘s move on to the next topic: **breakpoints**. The syntax to set the breakpoint in GDB is fairly simple, for example:

```
(gdb) break 20
```

Will set a break point at line 20 of the gdb_example2.c file. However, this syntax is sloppy when we are dealing with multiple files, in which case you should do the following:

```
(gdb) break gdb_example2.c:20
```

Or you can specify the address of the instruction directly:

```
(gdb) break *0xdeadbeef
```

Or use the function name:

```
(gdb) break memset
```

Lets put our newly learned knowledge to use! Say, we want to know where memset() is linked in our address space. Another powerful tool that might be of help is the ‘step‘ command.

```
(gdb) step <n>
```

This command executes ‘n‘ lines of instructions. If you don‘t give an argument, it defaults to executing one instruction. You can continue the program‘s execution by:

```
(gdb) continue
```

This will continue executing the program till the next break point or until the program end, whichever is earlier.
You can also clear breakpoints by following these instructions:

```
(gdb) info breakpoints
(gdb) delete <Breakpoints Number >
(gdb) clear
```

If you are having trouble with some of the GDB commands, try Googling them. There are many online resources describing the use of GDB commands.

Question 5: Where in memoy is memset() linked? Explain to us how you figured this out.

Question 6: In GDB, using only what you have learned, find the first 5 instructions in memset() on your machine? As a system call, does memset() use a different stack (e.g. kernel stack) than the user space stack? How did you figure this out?

Another important command without which your GDB knowledge is incomplete is the ‘print‘ command. You can use it to print the value of any variable or memory address that is currently in the scope of the program. For example, you have a variable ‘i‘ in the currently executing function, you can print its value by using the following command:

```
(gdb) print i
```

At this point, you know most of the necessary GDB commands needed in order to solve most bugs you might run into.

Fix all the bugs in `gdb_example2.c` without changing its functionality. Use GDB when necessary. Finally, answer the following question.

Question 7: Why cant the 'backtrace' command of GDB work as it is supposed to? Answer briefly.

This concludes the tutorial portion of this assignment. Before proceeding to the final exercise, make sure you have written all the answers in the provided 'micro-project2.txt' file.

PART 3

In this section, you will be fixing the bugs in a longer, more involved program. Compile **buggyBST.c**. Ignore all warnings for the sake of this exercise. Use GDB to find and fix all errors in the source code. Make any changes to the lines causing segfault. Hint: Most of the errors can be fixed by one character changes - They should seem like you are fixing a typo.

Some more GDB commands you may find helpful are: watch, rwatch and next. Find out more about them using 'help' in GDB. Type 'help' followed by the GDB command you want to know more about at the '(gdb)' prompt.

Deliverables

Here is the list of files you are to submit:

- **gdb_example1.c** - The fixed version
- **gdb_example2.c** - The bug free version
- **buggyBST.c** - The fixed version
- **Makefile** - The makefile (Either as is or with your changes)
- **micro-project2.txt** The text file with all your answers