CSE 220: Systems Programming

Lab 04: Introduction to gdb

Introduction

The GNU Debugger, or gdb, is a powerful *symbolic debugger*. Symbolic debuggers are available for many languages and platforms, and allow you to examine your program at the source code level.

You can use gdb to debug either a running program or a program that has crashed and left a *core file*. A core file is an image of the memory that a program was using at the time that it crashed. Running processes can be debugged either by starting the process and then attaching gdb, or starting the process from within gdb. Once gdb is debugging a process (or examining a core), a variety of commands may be used to examine the current state of the program or (if it is running) manipulate that state directly.

A debugger is a powerful complement to passive, print- or log-style debugging, as it lets you halt a program at a suspected problem point and then examine a broad range of characteristics, potentially including things you may not have thought to check before learning about the execution through interactive examination. It is not a panacea, however, as it disturbs timings and is not well-suited to finding *unknown* or *unexpected* bugs that may be discovered by principled and complete logs or execution traces.

In this lab, you will use gdb to examine and modify the execution of a running process to fix bugs. In the real world, this technique is often useful to move past a specific real bug in order to continue execution and perform additional debugging or development. You will learn how to:

- Examine local variables
- Examine memory
- Stop execution at a chosen point
- Modify memory or variables
- Move through the call stack

1 Getting Started

You will need to run this lab *either on `timberlake.cse.buffalo.edu` or the course virtual machine*; no other environments are guaranteed to produce the correct results. The file you should download to complete this lab depends on which machine you wish to use, as follows.

Machine	URL
timberlake	https://www.cse.buffalo.edu/~eblanton/course/cse220-2019-2f/materials/lab04-timberlake.tar
UBCSEVM	https://www.cse.buffalo.edu/~eblanton/course/cse220-2019-2f/materials/lab04-vm.tar

Retrieve the appropriate file, then extract it on the machine where you will be performing this lab with the command `tar xf filename.tar`, where `filename.tar` is the file you downloaded. You should find an extracted directory named `gdb_project` containing the files `debug` and `public.c`. The file `debug` is an executable built from the source file `public.c` and a second source file that is not provided to you. This allows you to examine the execution of `public.c` easily, but prevents you from fixing the sources in `public.c` and simply recompiling `debug`.

Start gdb as `gdb debug` in the extracted directory. You should see something like this:

```
[eblanton@timberlake]~/work/gdb/src/gdb_project$ gdb debug
GNU gdb (GDB) Red Hat Enterprise Linux (7.2-92.el6)
Copyright (C) 2010 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.  Type "show copying"
and "show warranty" for details.
```

```
This GDB was configured as "x86_64-redhat-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/csefaculty/eblanton/work/gdb/src/gdb_project/debug...done.
(gdb)
```

2 Using gdb

As mentioned above, gdb can start a program to be debugged, attach to an already running program, or debug a program that has crashed from a core file. You can control which mode gdb runs in by how you invoke it. You can also change the mode at run time, but that will not be described in this document.

In general, if you start gdb with one argument, it assumes that that one argument is the name of an executable file that you will want to debug. It will look in the current directory for a file of that name, and if it cannot find it there, it will search your $PATH for a matching file. If you start it with two arguments, it will go through the same search for an executable to debug, but it will assume that you are either attaching to a running program or performing a *post mortem* debugging session on a core file. If the second argument is the numeric process ID of a running process, it will attempt to attach to the running process; if it is a core file on disk, it will load the image for examination.

Thus, when we run gdb debug, we are telling gdb that we want to debug the executable named debug, and that we do not want it to either attach to a running program or load a core file. It will then load the symbols present in the debug executable and prepare it to be debugged when started. The program under debug can be started and will run until it either completes or crashes by issuing the command run at the (gdb) prompt. Try it now, and you should see something like this:

```
(gdb) run
Starting program: /home/elb/work/buffalo/cse220/recitations/gdb/src/debug
This program is using 'eblanton' as your UBITName.  If that is not your
UBITName, you should set the CSE220_UBIT environment variable to reflect
your actual UBITName.

Starting tests.

Program received signal SIGSEGV, Segmentation fault.
0x0000555555554eac in initialize_array (numbers=0x7fffffffded0, len=201527) at public.c:44
44          numbers[i] = i + 1;
(gdb)
```

This tells us that the program started, but then crashed with a segmentation fault (which means it attempted to access memory that did not belong to it) in the function initialize_array(), which can be found in the file public.c, on line 38 of that file. The offending line is then printed for reference. You can see more lines around that point by typing list, or you can consult the named source file for context.

More information about the crash can be gathered with various query commands provided by gdb. Try running the debug program in gdb now, and let it crash. After it crashes, try the following commands at the (gdb) prompt:

- backtrace to print the program execution stack — that is, the function calls that led it to the crash point
- backtrace full to print the backtrace along with more information
- list to see the source code around the current line
- disassemble to see the machine instructions for the current function

When you are finished, type kill to kill the crashed program so that it can be run again. Since it crashed in initialize_array(), you can keep it from crashing by halting execution when initialize_array() is called so

that you may be able to prevent the crash. (The information you learned from `backtrace full` should have told you why it crashed!) To cause it to halt when this function is called, type `break initialize_array` and then start the program again by typing `run`. You should see something similar to the following:

```
(gdb) break initialize_array
Breakpoint 1 at 0x555555554e89: file public.c, line 43.
(gdb) run
Starting program: /home/elb/work/buffalo/cse220/recitations/gdb/src/debug
This program is using 'eblanton' as your UBITName.  If that is not your
UBITName, you should set the CSE220_UBIT environment variable to reflect
your actual UBITName.

Starting tests.

Breakpoint 1, initialize_array (numbers=0x7fffffffded0, len=201527) at public.c:43
43      for (int i = 0; i < len; i++) {
(gdb)
```

The line that is printed at the bottom, just before the prompt, is the line of code that is *about to be executed, but has not yet been executed*. The command `next` will cause it to execute. Since this line of code seems unlikely to be the source of the bug we are experiencing, go ahead and run `next`:

```
(gdb) next
37      for (i = 0; i < 4; i++) {
(gdb)
```

If you consult the listing of `public.c`, you will see that although we went from line 35 to 37, 37 is the first line after 35 that actually contains executable code:

```
(gdb) list
32   * contain the digits 1-4 and 0.
33   */
34 void initialize_array() {
35      int *numbers = NULL, i = 0;
36
37      for (i = 0; i < 4; i++) {
38          numbers[i] = i + 1;
39      }
40      numbers[4] = 0;
41
(gdb)
```

In general, execution can be controlled via the following commands:

- `break` sets a *breakpoint*, which will stop execution when the program being debugged reaches a named *point in the code*. It takes an argument, which can be a source code line of the form `file:line` (or just `line` for the current file), such as `public.c:37`, in which case it breaks as soon as the line is reached (but before it executes); a function name, in which case it breaks as soon as the function is called (but, again, before it executes). It will print out the breakpoint number, which can be used to modify or delete the breakpoint later.
- `delete` removes all breakpoints if executed with no arguments, or a specified breakpoint if specified by number as an argument to the command.
- `watch` sets a *watchpoint*, which is a memory address or expression to be monitored for change. If a memory address (or variable name) is given, the program will stop when the data at that address or stored in the

variable changes. If an expression (in C syntax) is given, the program will change when the *result of the expression* changes. For example, `watch i < 4`, if executed when `i == 0`, will run without halting as long as `i` is negative or between 0 and 3, but will halt if `i` takes a value 4 or greater.

- `next` executes the program under debug until it reaches the next line of code, or a breakpoint/watchpoint (whichever comes first). The "next line of code" is the next line of code *within the current function*, so function calls will be executed as a single line of code.
- `step`, like `next`, executes the program under debug until it reaches the next line of code, but will "step into" a function if the current line of code includes a function call, halting inside the called function before its first line of code.
- `run` starts the program from the beginning, and executes it until completion, crash, or a breakpoint/watchpoint is reached. To pass arguments to the command being run, give them to the `run` command. For example, to run the program `debug` with the argument `--sort`, you would start `gdb` as `gdb debug`, and then type `run --sort` at the `(gdb)` prompt.
- `continue` continues execution where it was left off after halting for debugging, and runs until the next breakpoint/watchpoint, program completion, or a crash.
- Control-C will immediately halt the program (whatever it happens to be doing) for debugging. This is particularly useful if the program appears to have entered an infinite loop.

In addition to controlling program execution, the program's local variables and memory can be modified and examined, and a running program can be caused to execute (almost) arbitrary C code. In order to examine variables and memory, two valuable commands are `print` and `x`. The `print` command accepts a C expression as an argument and prints its value:

```
(gdb) print i
$1 = 0
(gdb) print len
$2 = 201527
(gdb)
```

Note that if the expression being printed *has side effects*, those side effects will also be executed! For example:

```
(gdb) print i = 1
$3 = 1
(gdb)
```

After this command, the actual value of `i` in the executing program is 1. Program state can be conveniently modified in this way (hint, hint). You can also call functions using print:

```
(gdb) print puts("This is a debug print")
This is a debug print
$8 = 22
(gdb)
```

Be careful, as in some cases this may cause your program to execute code in a state where it is not fully prepared to do so. In particular, using standard I/O functions (such as puts, above) before the program itself has initialized the standard I/O library is likely to lead to segmentation faults.

The `x` command is somewhat more complicated to use, but very powerful; it allows an arbitrary range of memory to be examined as various native word types. Its syntax is `x/FMT`, where `FMT` is a format string describing the number of words to be examined, the type of output desired, and the size of the word being examined. For example, to analyze the first 8 bytes of the executable code for the function `main()`, you would use:

```
(gdb) x/8xb main
0x555555554d53 <main>: 0x55 0x48 0x89 0xe5 0x48 0x83 0xec 0x10
(gdb)
```

Consult the gdb documentation (see Section 2.1) for usage of the `x` command.

Throughout this course, you will find the `backtrace` and `frame` commands to be among the most useful for analyzing the flow of data through your program. The `backtrace` command displays the program's call stack as it was when execution was halted:

```
(gdb) run
Starting program: /home/elb/work/buffalo/cse220/recitations/gdb/src/debug
This program is using 'eblanton' as your UBITName.  If that is not your
UBITName, you should set the CSE220_UBIT environment variable to reflect
your actual UBITName.

Starting tests.

Breakpoint 1, initialize_array (numbers=0x7fffffffded0, len=201527) at public.c:43
43     for (int i = 0; i < len; i++) {
(gdb) backtrace
#0  initialize_array (numbers=0x7fffffffded0, len=201527) at public.c:43
#1  0x0000555555554e4b in main (argc=1, argv=0x7fffffffdfd8) at public.c:27
(gdb)
```

This backtrace indicates that the `main()` function called the `initialize_array()` function. Each of these functions is contained in what is called a *stack frame*. We will learn more about stack frames later in the semester. The number at the beginning of each line is the stack frame identifier. You can inspect a specific frame using the `frame` command followed by the identifier of the stack frame you want to view, at which point you can use any of the debugging strategies previously described:

```
(gdb) frame 1
#1  0x0000555555554e4b in main (argc=1, argv=0x7fffffffdfd8) at public.c:27
27     initialize_array(numbers, len);
(gdb) list
22     private_init(&len);
23
24     printf("Starting tests.\n");
25     fflush(stdout);
26
27     initialize_array(numbers, len);
28
29     copy_string();
30
31     bubble_sort();
```

Consult the gdb documentation (see Section 2.1) for further explanation of the `backtrace` and `frame` commands.

2.1 gdb Documentation

There are two primary sources of `gdb` documentation: the `help` command and the `info` pages. You may remember GNU Info from the `make` documentation. You can access the `gdb` info pages with the command `info gdb`. For more information on using info, see the handout for the Introduction to Make lab. The `gdb` online help via the `help` command is also very valuable. Explore `help x` and `help info` for two good examples of powerful and useful commands that are difficult to summarize.

3 Requirements

You must correct the bugs in `public.c` by stopping the program at appropriate points and using `gdb` to examine and fix the program logic in-memory. In doing so, you will receive three "tokens" (which will be printed by the

program during execution). Save these tokens to a text file and submit it to Autograder. The tokens will be emitted on lines looking something like this:

```
Array token: arircDCZiszpw
```

You may include either the entire line or only the token itself in the file that you submit to Autograder.

There are three functions with bugs in the given code: `initialize_array`, `copy_string`, and `is_valid_integer`. Each function has one logic, memory management, or initial condition bug that you must correct in order to receive a token.

So that you do not have to start over at the beginning if you make a mistake in generating a token, you can run the `debug` program with arguments to skip to a particular test. Examine the source of `public.c` to see how.

4 Grading

You will receive one point for each bug you "fix", submitting the associated token to Autograder.