15213 Lecture 5: Intro. to GDB & Assembly

1 Introduction

This handout will introduce and provide a reference for your use of GDB (the GNU symbolic debugger), as well as the basics of assembly. Portions of this information will be useful for bomblab, attacklab, cachelab, tshlab, malloclab, and proxylab. Which is to say, you will absolutely need to know this material to succeed in the course.

2 Getting Started

To invoke a program from the command line simply type the path to its executable file and hit enter. If your shell's working directory contains this file, simply prepend '`./`' before the file name. To pass command-line arguments to the program, just enter spaces between each argument (and after the path to the executable).

To begin this activity, login to a shark machine, then download the handout archive by typing `wget http://www.cs.cmu.edu/~213/activities/lec5.tar`
extract it with `tar xf lec5.tar`,
then `cd lec5`.
Now execute '`./act1`' and follow its instructions for rerunning it inside GDB.

3 Activity 1, Steps 1 and 2

This first activity deals with stepping though functions, examining registers, assembly syntax, disassembling, and basic use of GDB. Once you run it inside GDB with the appropriate argument, it will show you how to use GDB's `i`, `r`, and `help` commands (with more details available on the reference at the end of this writeup).

4 Activity 1, Step 3 and 4

To begin the next two steps, supply the arguments '`2`' and '`3`' to '`act1`', in turn (as you did before). Follow along with the output of these steps to learn more about disassembly, register/memory inspection, and assembly syntax. These steps introduce the following commands: `x`, `c`, `q`, and `disassemble` (see final page for additional details).

Questions posed in this section of the program, with space to write your solutions:

1. `MOV` is a common and powerful instruction. It can "move" (actually, copy) values between registers, load or store with memory. In this case, `%edi` holds the argument to the function and `%eax` the return value. Write pseudo C for this function.

2. How do the names of the registers differ between the functions?

3. Did `squareFloat` use the same registers from before?

4. What do you think it is doing? `%rdi` has the first argument and `%rsi` the second.

5 Activity 2, Step 0

Moving on to activity 2, start by launching GDB with '`gdb act2`'. To enter this step, run `act2` in GDB with no arguments (the plain `r` command). Questions posed in this section of the program, with space to write your solutions:

1. Did `whatIsThis`: compare, swap, add, or multiply two numbers?

6 Activity 2, Step 1

To enter this section, restart the program inside GDB, passing it the '`s`' command-line argument. Questions posed in this section of the program, with space to write your solutions:

1. What does this function do?

7 Activity 2, Step 2

To enter this section, pass the '`a`' argument.

8 Activity 2, Step L

To enter this section, pass the '`l`' argument. Questions posed in this section of the program, with space to write your solutions:

1. What is the value that `mx` multiplies by?

9 GDB Command Reference

This page is merely a reference for you to have on hand, please skip ahead to the actual activity if you have not yet completed it. Commands discussed in this lecture:

1. '`c`' ('`continue`') resumes program execution. Not to be used lightly.
2. '`disassemble`' outputs the assembly translation of the function currently being executed, or the translation of a target function if one is supplied.
3. '`help`' Provides a wealth of information on the following topic. Like '`i`,' enter it without a topic to see the list of possible topics.
4. '`i`' ('`info`') Provides information about the following subcommand. While '`registers`' is a common subcommand (shows the contents of primary registers), just type '`i`' for a list of all of them.
5. '`p`' ('`print`') is an extremely flexible command that lets you examine values and execute essentially arbitrary C code from within GDB, using references defined within the current scope (only works with certain compiler flags). Example usage: '`print instance.field == NULL`'
6. '`q`' ('`quit`') quits GDB session after confirmation.
7. '`r`' (or '`run`') Restarts the program currently being debugged. By default, it will use the same arguments supplied to the program during its last execution. However, you can instead pass a space-separated list of custom command-line arguments.
8. '`x`' A very powerful command that prints the memory contents of the supplied address[1]. Can be used with the format 'x/*length/format address-expression*.' Good formats to know include: `s` (string), `x` (hex), and `d` (decimal). Example usage: '`x/3x 0x42`' to dump 3 hexadecimal integers starting from memory address `0x42`

While not strictly a command, in many contexts, ^C (Ctrl+C) issues an interrupt signal to the foreground. In GDB, this returns you to a `(gdb)` prompt if the program was busy. Assorted tools that will be more useful later on in the course:

1. '`gdb -tui ./something`' will let you display the C code being stepped through alongside the standard GDB console. You may occasionally have to hit ^L to redraw.
2. '`n`' ('`next`') executes the next line of C code, then stops the program again. '`ni`' performs the equivalent for just the next assembly instruction.
3. '`b`' ('`break`') tells GDB to stop the program whenever it reaches a location, which may be a function (e.g. `foo`), source line (e.g. `file.c:80`), or address (e.g. `*0xdeadbeef`).
4. '`explore`' lets you follow chains of pointers between structs. For example, if you have a struct instance you want to inspect that is 4 levels of inheritance removed from the reference you have, you can '`explore reference`' to examine the fields of each parent struct along the pointer chain from great-grandparent to child.

[1] This is an actual expression, so `0x10 + (1 * (7 - 5))` will work, for instance.

10 FAQ

1. When the program starts, GDB reports: `(no debugging symbols found)...done` Is this a problem?

   No. While helpful for debugging, the symbols are not required. Similarly for bomblab, you will be working on a bomb that does not have any symbols provided.

2. GDB is printing out: `Program received signal SIGTRAP, Trace/breakpoint trap.` What is happening?!

   For this activity, we have hard-coded breakpoints into the program, so that you can work with GDB. This message is informing you that one of those hard-coded points was reached.

3. When I try to run the program in GDB again, GDB asks me if I want to start the program from the beginning (y or n)?

   You are currently debugging, so GDB wants to make sure you want to stop this execution and start again. For these exercises, press y then enter.