15213 - Recitation 2 - Bomb Lab

September 19th, 2022

Introduction

In this activity you will review the material on assembly instructions and use gdb to debug and interpret programs. This serves as practice for bomblab. The same information is provided on the slides and but have been provided as a handout in order to help people along with the process. In order to get started, login to a Shark machine and type these commands:

```
$ wget http://www.cs.cmu.edu/~213/activities/rec4.tar
$ tar xvpf rec4.tar
$ cd rec4
$ make
$ gdb act1
```

Activity 1:

In this activity you will get familiar with using gdb to interpret program behavior.

- `(gdb) break main` // tells GDB to pause right before entering main
- `(gdb) run 15213` // starts execution with the argument "15213"
  You should see GDB print out:
  `Breakpoint 1, main (argc=2, argv=[...]) at act1.c:5`
- `(gdb) continue` // this continues execution until another break point
- `(gdb) clear main` // remove the breakpoint at function main
- `(gdb) run 15213` // Q: What happens now? Program ended straight away
- `(gdb) disassemble main` // show the assembly instructions in main
- `(gdb) print (char*) [0x...]` // prints a string
- Q: Does the printed value correspond to anything in the C code?
  no
- `(gdb) break main`
- `(gdb) run (with some argument)`
- `(gdb) print argv[1]` // Q: What does this print out? The argument passed in
- `(gdb) continue` // Q: Now what does this print out? Process received signal
- `(gdb) quit` // exit GDB; agree to kill the running process

Activity 2:

In this activity you will get familiar with using gdb to step through assembly code and try to interpret what the program does

- `$ gdb act2`
- `(gdb) break main`
- `(gdb) run`
- `(gdb) print/x $rsi` // '/x' means print in hexadecimal

- `(gdb) print /x $rdi`
- Q. RDI and RSI are registers that pass the first two arguments.
  Looking at their values, which is the first argument to main (the 'argc' argument)? Why?
  RDI is the argc because it's an integer and equals 1.
  That means we didn't pass any command line args
- `(gdb) disassemble main`
- `(gdb) break stc` // main calls the stc function, so we'll study that function too
- `(gdb) continue`
- Q. How could you view the arguments that have been passed to stc?
  check %rdi and %rsi
- `(gdb) run 18213` // gdb will ask if you want to restart; choose yes
- `(gdb) continue` // Q. Which function is in execution now?
- `(gdb) disassemble`
- `(gdb) stepi`
- `(gdb) nexti` // step through a single x86 instruction
- `(gdb)` // just press enter 3 to 4 times
  (GDB will repeat your previous instruction. Useful for single- stepping.)
- `(gdb) disassemble`
- Q. Where are the "=>" characters printed on the left side?
  It's the next instruction to execute, i.e. %rip pointer

Activity 3:

Activity 3 has a Bomb Lab feel to it. It will print out "good args!" if you type in the right numbers into the command line. Use GDB to find what numbers to use.

- `$ cat act3.c` // display the source code of act3
- `$ gdb act3`
- Q. Which register holds the return value from a function? %rax
  (Hint: Use disassemble in main and look at what register is used right after the function call to compare)
- `(gdb) disassemble compare`
- Q. Where is the return value set in `compare`? %rax
- `(gdb) break compare`
- Now run act3 with two numbers
- Q. Using nexti or stepi, how does the value in register %rbx change, leading to the cmp instruction?

  %rbx was initially assigned by %rdi. Next, the value + 5 was set.
  Then %rsi was added in.

GDB Cheat Sheet:

● **quit** (q)
○ Exit gdb

● **disassemble** [any function name within the executable] (disas)
○ Show the assembly code for that function

● **set disassemble-next-line on set disassemble-next-line off show disassemble-next-line**
○ Shows the next assembly instruction after each step

● **stepi**
○ Step through 1 assembly instruction. Will follow function calls into other functions

● **nexti**
○ Same as stepi, but will not follow function calls.

● **breakpoint** [location] (b)
○ Set a breakpoint at location.
○ Execution will stop before executing the line where the breakpoint is located at.
○ The most common are [function], [*memory address], and [line number], but can be a number of things
○ See the gdb documentation for more information:
ftp://ftp.gnu.org/old-gnu/Manuals/gdb/html_node/gdb_28.html#SEC29

● **info breakpoints** (i b)
○ List all breakpoints, along with whether or not they are enabled

● **delete** [breakpoint] (d)
○ Delete a breakpoint

● **enable** [breakpoint] (en)
○ Enable a breakpoint, If none specified, enables all breakpoints

● **disable** [breakpoint] (dis)
○ Disable a breakpoint. If none specified, disables all breakpoints

● **info registers** (i reg)
○ Print all the registers, along with their contents

● **print** [any valid C expression] (p)
○ Can be used to study any kind of local variable or memory location
○ Use casting to get the right type (e.g. print *(long *)pointer)
○ Can format with things like /x (hex), /d (int), /s (string) etc.

● **x** [some format specifier] [some memory address]
○ Examines memory. “x ptr” is the same as “p *ptr”
○ Can format with things like /x (hex), /d (int), /s (string) etc.

● **layout** [next | prev | assembly | register] (la [n | p | asm | reg])
○ Show assembly code or registers in the top half of the window as you go
○ The TUI is buggy. If your screen begins glitching, you may have to restart GDB
○ Exit with <ctrl> x + <ctrl> a

● **focus** [command | assembly | register] (fo [cmd | asm | reg])
○ Change the focus window within the TUI (buggy)