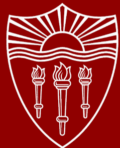
CS356: Discussion #5

Debugging with GDB

Marco Paolieri (paolieri@usc.edu)



USC University of Southern California

Schedule: Exams and Assignments

- Week 1: Binary Representation **HW0**
- Week 2: Integer Operations
- Week 3: Floating-Point Operations **Data Lab 1**
- Week 4: Assembly (Arithmetic Instruction)
- **Week 5: Assembly (Debugging with GDB)** **Data Lab 2**
- Week 6: Assembly (Function Calls)
- Week 7: **Bomb Lab** (Oct. 1), **Exam I** (Oct. 4), Security Vulnerabilities
- Week 8: Memory Organization
- Week 9: Caching **Attack Lab**
- Week 10: Virtual Memory
- Week 11: Dynamic Memory Allocation and Linking
- Week 12: Processor Organization and **Exam II** (Nov. 8) **Cache Lab**
- Week 13: Processor Organization
- Week 14: Code Optimization and **Thanksgiving**
- Week 15: Cache Coherency and Review **Allocation Lab**
- Week 16: Study Days and **Final** (Dec. 6)

Project #3

Goal: to defuse a “binary bomb” by figuring out the correct inputs.

- A sequence of 6 phases: each phase asks for an **input from stdin**.
- If the correct input is provided, the program proceeds to the **next phase**.
- If the wrong input is provided, the program **terminates** with an “explosion.”

Your goal is to complete all phases. You must figure out the correct inputs by disassembling the binary program that is **already in your GitHub repository**.

- Complete the assignment inside the VM (must have internet connection).
- No need to submit your work: the binary program pings our server.

Score (see: http://bytes.usc.edu/cs356/assignments/bomblab.pdf)

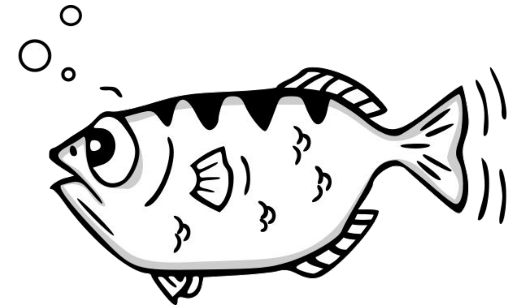
- You gain 10 points for phases 1-4) and 15 points for phases 5-6 (total: 70).
- You lose 0.5 points if you cause an explosion in an unsolved phase.
- Your score is updated with these losses only after you complete the phase.
- You have 1 free explosion for phases 1-4 and 3 for phases 5-6.
- Completing a phase always gives you **at least 40% of its points**.

gdb: The GNU Debugger

Goal: "To help you catch bugs in the act."

How?

- Start your program (specifying inputs).
- Pause it when a condition is met (breakpoints).
- Examine the current state (inspect).
- Proceed step-by-step (understand).



For a fish, the archer fish is known to shoot down bugs from low hanging plants by spitting water at them.

— Jamie Guinan | https://goo.gl/VxsgbU

Getting started

- Install gdb: `apt-get install gdb` (already present on your VM)
- Include debugging information: `gcc -g hello.c -o hello`
- Run gdb on your binary program:

```
$ gdb hello
Reading symbols from hello...done.
(gdb) _
```

User Interface

An interactive shell

- Autocomplete a command with **tab**
- Scroll history of previous commands with **up** / **down**
- Repeat the previous command with **enter**
- Commands can often be abbreviated with few letters (in **red**)
- Help about a command: `(gdb) help <command>`
- Open a file for debug: `(gdb) file <binary file>`
- Quit: `(gdb) quit`

Looking at the C code

- Show 10 lines around beginning of a function: `(gdb) list func_name`
- Show next 10 lines: `(gdb) list`
- Set how many lines to show: `(gdb) set linesize 20`

A bit tedious!

There is a more practical interface: `gdb -tui`, the “terminal user interface”

User Interface Reloaded: `gdb -tui`

```
hello.c
     1       #include <stdio.h>
     2
     3       int twice(int x) {
     4           return 2*x;
     5       }
     6
     7       int main() {
B+>  8           printf("Hello World!\n");
     9
     10          int x = 3;
     11          if (x > 0) {
     12              printf("%d\n", twice(x));
     13          }
     14      }
     15
     16
     17
native process 22823 In: main                    L8     PC: 0x40057c
(gdb) file hello
Reading symbols from hello...done.
(gdb) b 8
Breakpoint 1 at 0x40057c: file hello.c, line 8.
(gdb) run
Starting program: /home/marco/Desktop/cs356/discussion-5/hello

Breakpoint 1, main () at hello.c:8
(gdb)
```

Scroll through source code

Enter commands

A few tips

Moving the focus

- By pressing up / down / left / right, you scroll the source sub-window
- To scroll the history or move along the command line, you must set the focus on the other part of the screen: **`C-x o`** (press ctrl+x, release, press o)

Redrawing the screen

- If your program prints to stdout, it will interfere with the TUI interface
- In case, you can redraw the screen with **`C-l`**

Changing mode

- You can enable/disable the TUI mode with **`C-x a`**
- Or, you can select a mode:
  - `(gdb)` **`layout src`** Show source and commands
  - `(gdb)` **`layout asm`** Show assembly and commands
  - `(gdb)` **`layout split`** Show source, assembly, commands
  - `(gdb)` **`layout regs`** Show registers

Layouts

```
  ┌─hello.c──────────────────────────────────────┐
  │7        int main() {                          │
B+>│8            printf("Hello World!\n");         │
  │9                                              │
  │10           int x = 3;                        │
  │11           if (x > 0) {                      │
  │12               printf("%d\n", twice(x));     │
  │13           }                                 │
  │14       }                                     │
  │15                                             │
  │16                                             │
  │17                                             │
  │18                                             │
  │19                                             │
  │20                                             │
  │21                                             │
  │22                                             │
  │23                                             │
  └───────────────────────────────────────────────┘
native process 5647 In: main               L8      PC: 0x40057c
(gdb) file hello
Reading symbols from hello...done.
(gdb) layout src
(gdb) b 8
Breakpoint 1 at 0x40057c: file hello.c, line 8.
(gdb) run
Starting program: /home/marco/Desktop/cs356/discussion-5/hello

Breakpoint 1, main () at hello.c:8
(gdb)
```

```
   0x400574 <main>           push   %rbp
   0x400575 <main+1>         mov    %rsp,%rbp
   0x400578 <main+4>         sub    $0x10,%rsp
B+>0x40057c <main+8>         mov    $0x400644,%edi
   0x400581 <main+13>        callq  0x400430 <puts@plt>
   0x400586 <main+18>        movl   $0x3,-0x4(%rbp)
   0x40058d <main+25>        cmpl   $0x0,-0x4(%rbp)
   0x400591 <main+29>        jle    0x4005ae <main+58>
   0x400593 <main+31>        mov    -0x4(%rbp),%eax
   0x400596 <main+34>        mov    %eax,%edi
   0x400598 <main+36>        callq  0x400566 <twice>
   0x40059d <main+41>        mov    %eax,%esi
   0x40059f <main+43>        mov    $0x400651,%edi
   0x4005a4 <main+48>        mov    $0x0,%eax
   0x4005a9 <main+53>        callq  0x400440 <printf@plt>
   0x4005ae <main+58>        mov    $0x0,%eax
   0x4005b3 <main+63>        leaveq
native process 7334 In: main               L8      PC: 0x40057c

Reading symbols from hello...done.
(gdb) layout asm
(gdb) b 8
Breakpoint 1 at 0x40057c: file hello.c, line 8.
(gdb) run
Starting program: /home/marco/Desktop/cs356/discussion-5/hello

Breakpoint 1, main () at hello.c:8
(gdb)
```

```
  ┌─hello.c──────────────────────────────────────┐
  │7        int main() {                          │
B+>│8            printf("Hello World!\n");         │
  │9                                              │
  │10           int x = 3;                        │
  │11           if (x > 0) {                      │
  │12               printf("%d\n", twice(x));     │
  │13           }                                 │
  │14       }                                     │
  └───────────────────────────────────────────────┘
   0x400574 <main>           push   %rbp
   0x400575 <main+1>         mov    %rsp,%rbp
   0x400578 <main+4>         sub    $0x10,%rsp
B+>0x40057c <main+8>         mov    $0x400644,%edi
   0x400581 <main+13>        callq  0x400430 <puts@plt>
   0x400586 <main+18>        movl   $0x3,-0x4(%rbp)
   0x40058d <main+25>        cmpl   $0x0,-0x4(%rbp)
   0x400591 <main+29>        jle    0x4005ae <main+58>
native process 8542 In: main               L8      PC: 0x40057c

Reading symbols from hello...done.
(gdb) layout split
(gdb) b 8
Breakpoint 1 at 0x40057c: file hello.c, line 8.
(gdb) run
Starting program: /home/marco/Desktop/cs356/discussion-5/hello

Breakpoint 1, main () at hello.c:8
(gdb)
```

```
┌─Register group: general──────────────────────────┐
│rax            0x400574 4195700                    │
│rbx            0x0      0                          │
│rcx            0x0      0                          │
│rdx            0x7fffffffdce8   140737488346344    │
│rsi            0x7fffffffdcd8   140737488346328    │
│rdi            0x1      1                          │
│rbp            0x7fffffffdbf0   0x7fffffffdbf0     │
│rsp            0x7fffffffdbe0   0x7fffffffdbe0     │
└───────────────────────────────────────────────────┘
   0x400574 <main>           push   %rbp
   0x400575 <main+1>         mov    %rsp,%rbp
   0x400578 <main+4>         sub    $0x10,%rsp
B+>0x40057c <main+8>         mov    $0x400644,%edi
   0x400581 <main+13>        callq  0x400430 <puts@plt>
   0x400586 <main+18>        movl   $0x3,-0x4(%rbp)
   0x40058d <main+25>        cmpl   $0x0,-0x4(%rbp)
   0x400591 <main+29>        jle    0x4005ae <main+58>
native process 10111 In: main              L8      PC: 0x40057c
Reading symbols from hello...done.
(gdb) layout asm
(gdb) layout regs
(gdb) b 8
Breakpoint 1 at 0x40057c: file hello.c, line 8.
(gdb) run
Starting program: /home/marco/Desktop/cs356/discussion-5/hello

Breakpoint 1, main () at hello.c:8
(gdb)
```

Breakpoints and Control Flow

Breakpoints

- Add at current location: `(gdb) break`
- Add at the beginning of a function: `(gdb) break func_name`
- Add at a specific line of a source file: `(gdb) break hello.c:5`
- Add at a specific line of current file: `(gdb) break 5`
- List all breakpoints: `(gdb) info breakpoints`
- Delete a breakpoint: `(gdb) delete <breakpoint #>`
- Disable/enable breakpoint: `(gdb) disable <#>` and `(gdb) enable <#>`

Controlling the execution

- Run a program from start, until first breakpoint: `(gdb) run <args>`
- Advance your program execution manually
  - Continue to the next line, **executing** subroutines: `(gdb) next`
  - Continue to the next line, **stepping into** subroutines: `(gdb) step`
- Run until the next breakpoint: `(gdb) continue`
- Run until the end of the function and print return value: `(gdb) finish`

Inspecting Data

Registers: `(gdb) info registers`
Stack: `(gdb) info stack` and `(gdb) info frame`

Memory

- Print 1 byte at `0x12345` as unsigned int: `(gdb) x/1ub 0x12345`
- Print 2 words above stack pointer as hex: `(gdb) x/2xw $sp`
- Print string at memory address contained in `%rdi`: `(gdb) x/s $rdi`

Variables

- Print an expression: `(gdb) print a/b+3.0*func_name(3)`
- In hexadecimal: `(gdb) print/x var_name`
- Display an expression after every step: `(gdb) display var_name`

Pausing on variable or condition changes

- Add a **watchpoint** for a variable (current scope): `(gdb) watch var_name`

Pausing at a line on given conditions

- Add a **conditional breakpoint**: `(gdb) break 8 if x > y`

Disassembling binary code

When source code is missing...

- List all the strings in a binary file using: **`strings`** `objfile`
- Print the symbol table: **`objdump -t`** `objfile`
  - Names of all functions and global variables in `objfile`
  - Example:
    `0000000000400ab6 g F .text 0000000000000064  riddle_2`
    Meaning: a **g**lobal **F**unction in section `.text` with name `riddle_2`
- Debugging with **gdb** (use **`layout asm`** `in gdb -tui`)
  - Print the assembly of a function: `(gdb)` **`disassemble`** `<func>`
  - Breakpoint at a given address: `(gdb)` **`break`** `*<addr>`
  - Next/step one assembly instruction at a time: `(gdb)` **`ni`** and **`si`**
  - Jump to a given address: `(gdb)` **`jump`** `*<addr>`
  - Print the string at a given address: `(gdb)` **`x/s`** `<addr>`

Getting started with the assignment

Disassemble and step through `main`

- Open **`gdb -tui`** and set **`layout asm`**
- Load the binary file: `(gdb)` **`file riddle`**
- Set a breakpoint on main: `(gdb)` **`b main`**
- Start the program: `(gdb)` **`run`**
- Look around and advance with **`ni`** and **`si`**
  - Can you find where inputs are read from stdin?
  - Can you find the calls to `riddle_1` and `riddle_2`?
  - Can you figure out their input parameters?

Remember

- Disassemble a function with `(gdb)` **`disas func_name`**
- Redraw the screen with **`Ctrl-l`**
- Print the string at the address in `%rdi` using: `(gdb)` **`x/s $rdi`**

Today: an easier problem

Download from: http://bytes.usc.edu/cs356/labs/riddle.zip

Two-Phases

- The main program reads two strings from stdin.
- The strings are validated by calling functions `riddle_1` and `riddle_2`

```
$ ./riddle
To continue, tell me:  how is an orange like a bell?
I know you can Google it, but don't.
<enter correct answer here>

Very well then.  Tell me the ages of my three children.
Hint 1:  If you multiply their ages, the product is 36.
Hint 2:  If you add up their ages, it is the number of
         my neighbor's house.
Hint 3:  The oldest one is in fourth grade.
<enter three numbers here>
Sorry, you failed to complete the riddle challenge.
```

Riddle 1

Understanding

- Which functions are called by `riddle_1`?
- Which parameters are passed?
- Which output values are used afterward?
- Jumps? Conditional jumps?

```
(gdb) disas riddle_1
Dump of assembler code for function riddle_1:
  0x0000000000400a30 <+0>:    sub    $0x8,%rsp
  0x0000000000400a34 <+4>:    mov    $0x400dd0,%esi
  0x0000000000400a39 <+9>:    callq  0x4009c9 <strings_not_equal>
  0x0000000000400a3e <+14>:   test   %eax,%eax
  0x0000000000400a40 <+16>:   je     0x400a47 <riddle_1+23>
  0x0000000000400a42 <+18>:   callq  0x400891 <explode_bomb>
  0x0000000000400a47 <+23>:   add    $0x8,%rsp
  0x0000000000400a4b <+27>:   retq
End of assembler dump.
```

Riddle 2

```
0x0000000000400a79 <+0>:    sub    $0x18,%rsp
0x0000000000400a7d <+4>:    lea    0x4(%rsp),%rsi
0x0000000000400a82 <+9>:    callq  0x400a4c <read_three_numbers>
0x0000000000400a87 <+14>:   mov    0x4(%rsp),%eax
0x0000000000400a8b <+18>:   test   %eax,%eax
0x0000000000400a8d <+20>:   jns    0x400a94 <riddle_2+27>
0x0000000000400a8f <+22>:   callq  0x400891 <explode_bomb>
0x0000000000400a94 <+27>:   cmp    $0x2,%eax
0x0000000000400a97 <+30>:   je     0x400a9e <riddle_2+37>
0x0000000000400a99 <+32>:   callq  0x400891 <explode_bomb>
0x0000000000400a9e <+37>:   cmpl   $0x2,0x8(%rsp)
0x0000000000400aa3 <+42>:   je     0x400aaa <riddle_2+49>
0x0000000000400aa5 <+44>:   callq  0x400891 <explode_bomb>
0x0000000000400aaa <+49>:   cmpl   $0x9,0xc(%rsp)
0x0000000000400aaf <+54>:   je     0x400ab6 <riddle_2+61>
0x0000000000400ab1 <+56>:   callq  0x400891 <explode_bomb>
0x0000000000400ab6 <+61>:   add    $0x18,%rsp
0x0000000000400aba <+65>:   retq
```