Course: Operating Systems - Semester 1 of 5785

Assignment 1 with Answers

Directions

A. Due Date: 14 November 2024 at 11:55pm

B. The free response questions for this assignment must be submitted via Moodle using the dedicated assignment element.

C. There are 100 points total on this assignment.

Analyzing Executables[1]

The goal of this assignment is to understand what is really inside of a running program and what the operating system needs to deal with.

1 gdb (50 points total)

Load the `tee` executable I provided in `gdb` with a single input file command line argument, set a breakpoint at `main`, start your program, and continue one line at a time until you are in the middle of your program's execution. Take a look at the stack using where or backtrace (bt). While you are looking through `gdb`, think about the following questions and put your answers in the free response file (either TXT or DOCX).

(gdb) 1. (5 points) What is the value of `argv`? (Hint: `print argv`)

-
  ```
  (gdb) print argv
  $1 = (char **) 0x7ffff7fb4fc8
  ```

  It's a pointer to the array.

(gdb) 2. (5 points) What is pointed to by `argv`? (Hint: `print argv[0]`)

-
  ```
  (gdb) print argv[0]
  $2 = 0x7fffffffe058 "/home/mjmay/LinuxShared/tee"
  ```

  The program that is running (tee).

(gdb) 3. (5 points) What is the address of the function main?

-
  ```
  (gdb) print main
  $3 = int (int, char **) 0x555555555249 <main>
  ```

  The address is 0x555555555249

(gdb) 4. (5 points) Type `info stack`. Explain what you see.

-
  ```
  (gdb) info stack
  #0 main (argc=2, argv=0x7fffffffe058) at tee.c:15
  ```

  It shows the current state of the function main (since there are no other frames on the stack). It also shows the parameters in the function call (1). It also shows which line in the program we are at and which source code file is currently in use.

(gdb) 5. (5 points) Type `info frame`. Explain what you see.

-
  ```
  (gdb) info frame
  Stack level 0, frame at 0x7fffffffdeb0:
  rip = 0x55555555551c in main (main.c:187); saved rip = 0x7ffff7deb0b3
  source language c.
  Arglist at 0x7fffffffde68, args:  argc=2, argv=0x7fffffffdf98
  Locals at 0x7fffffffde68, Previous frame's sp is 0x7fffffffdeb0
  Saved registers:
  rbp at 0x7fffffffdea0, rip at 0x7fffffffdea8
  ```

  The program is at the outermost stack frame (no one called us).

  The program counter is 0x55555555551c (rip) in main.

  The saved rip is 0x7fffffffdf18 - this is the address to return to when the program is done.

[1]This assignment is adapted from HW0 of CS162 at UC Berkeley by Kubiatowicz from this year and previous years. The original can be found at: `https://cs162.org/static/homeworks/homework0.pdf`

The source language of the program is c.

There are is one argument in the argument list. Its address is printed.

There are local variables in the function. They are stored at 0x7fffffffde68.

The registers currently saved are listed above.

(gdb) 6. (25 points) Type `info registers`. Below are 10 registers that you should see in the results. For each register, write the data that it contains and a 1 sentence description of what its job is.

(1) rax

(2) rbx

(3) rsi

(4) rdi

(5) rbp

(6) rsp

(7) rip

(8) cs

(9) ss

(10) ds

-
  ```
  (gdb) info registers
  rax 0x555555555249 93824992236105
  rbx 0x0 0
  rcx 0x5555555554d0 93824992236752
  rdx 0x7fffffffe070 140737488347248
  rsi 0x7fffffffe058 140737488347224
  rdi 0x2 2
  rbp 0x7fffffffdf40 0x7fffffffdf40
  rsp 0x7fffffffdee0 0x7fffffffdee0
  r8 0x7ffff7f9cf10 140737353731856
  r9 0x7ffff7fc9040 140737353912384
  r10 0x7ffff7fc3908 140737353890056
  r11 0x7ffff7fde660 140737353999968
  r12 0x7fffffffe058 140737488347224
  r13 0x555555555249 93824992236105
  r14 0x0 0
  r15 0x7ffff7ffd040 140737354125376
  rip 0x55555555525d 0x55555555525d <main+20>
  eflags 0x202 [ IF ]
  cs 0x33 51
  ss 0x2b 43
  ds 0x0 0
  es 0x0 0
  fs 0x0 0
  gs 0x0 0
  ```

  There are many registers here.

  rax, rbx, rcx, rdx Accumulator registers for general purpose use

  rsi Source index register

rdi Destination index register

rbp Stack base pointer register

rsp Stack pointer register

rip Program counter

eflags x86 flags register: Parity flag (PF) and Interrupt Enable flag (IF) are set.

cs Code segment register

ss Stack segment register

ds Data segment register

fs, gs Extra segment registers

r8-r15 Extra registers for use in x86 64 bit.

See `https://wiki.cdot.senecacollege.ca/wiki/X86_64_Register_and_Instruction_Quick_Start` for more information.

2 Register Matching (10 points / 4 points each)

Correlate the following common registers with the tasks they are used for:

Register	Task
rsp	Stack pointer
rip	Instruction pointer
cs	Code Segment
ss	Stack Segment
ds	Data Segment

3 objdump (40 points / 10 points each)

There is more to the executable than meets the eye. Let's look down inside. Run `objdump -x -d tee`. You will see that the program has several segments, names of functions and variables that correspond to labels with addresses or values. And the guts of everything is chunks of stuff within segments. In the `objdump` output these segments are under the section heading. There's actually a slight nuance between these two terms which you can read more about online.

After you look through the objdump output, answer the following questions in the free response file (either TXT or DOCX).

(objd) 1. What file format is used for this binary? And what architecture is it compiled for?

-
  ```
  words:  file format elf64-x86-64
  words
  architecture:  i386:x86-64, flags 0x00000150:
  ```

  It's in ELF64 format and written for the i386 x86-64 architecture.

(objd) 2. List the names of 10 segments and sections you find.

- There are a bunch of segments and sections here:

  (a) Program Header

  (b) Dynamic Section

(c) Sections: .interp, .gnu.hash, .gnu.version, .init, .text, .eh_frame_hdr, .eh_frame, .init_array, .data, .debug_aranges, .debug_info. Some of these sections are here due to the inclusion of the gdb debug symbols.

(d) Symbol table

(e) There is a main section with the code in assembler.

(f) There is a section of libc code in assembler.

(objd) 3. What segment/section contains main (the function) and what is the address of main?

- The section is .text. Its value is `0000000000001509 <main>:` `<main>:` It's at address: 0x1509.

(objd) 4. Do you see the stack segment anywhere? What about the heap? Explain.

- No, those aren't here. They are dynamic sections which are built as the program runs. They don't show up in the executable file, only once the process begins running.