

Name and NetID:

Rohit Bellam – rsb204

Enrico Aquino - eja97

Part 1:

**1. What are the contents of the stack? Feel free to describe your understanding.**

The contents of the stack include the argument build, saved registers and local variables, the old frame pointer, and the caller frame that includes the return address and argument for the call. In the context for stack.c, there are two primary functions:

- main
- signal\_handle

Now, in order to view the contents of the stack, I used GDB (GNU debugger) which lets users debug programs that have errors or issues running.

In the case of stack.c, the content of the stack includes

- int r2
- int signalno
- function memory addresses (main and signal\_handle)

Of course, the stack also holds the program counter (return address), function arguments, saved register memory locations, and other key memory addresses.

**2. Where is the program counter, and how did you use GDB to locate the PC?**

In this case, the program counter is located within the stack frame. If we analyze the image below:

```

(gdb) b signal_handle
Breakpoint 1 at 0x11ce: file stack.c, line 22.
(gdb) run
Starting program: /common/home/rsb204/CS416/Project1/stack
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Program received signal SIGSEGV, Segmentation fault.
main (argc=1, argv=0xffffd464) at stack.c:41
41      r2 = *(int *) 0; // This will generate segmentation fault
(gdb) info frame
Stack level 0, frame at 0xffffd3b0:
  eip = 0x56556230 in main (stack.c:41); saved eip = 0xf7d76519
  source language c.
  Arglist at 0xffffd398, args: argc=1, argv=0xffffd464
  Locals at 0xffffd398, Previous frame's sp is 0xffffd3b0
  Saved registers:
    ebx at 0xffffd394, ebp at 0xffffd398, eip at 0xffffd3ac
(gdb) p $eip
$1 = (void (*)( )) 0x56556230 <main+61>
(gdb) continue
Continuing.

Breakpoint 1, signal_handle (signalno=11) at stack.c:22
22      int i = 0;
(gdb) info frame
Stack level 0, frame at 0xfffffc570:
  eip = 0x565561ce in signal_handle (stack.c:22); saved eip = 0xf7fc4560
  called by frame at 0xffffd380
  source language c.
  Arglist at 0xfffffc568, args: signalno=11
  Locals at 0xfffffc568, Previous frame's sp is 0xfffffc570
  Saved registers:
    ebx at 0xfffffc564, ebp at 0xfffffc568, eip at 0xfffffc56c
(gdb) p $eip
$2 = (void (*)( )) 0x565561ce <signal_handle+17>
(gdb) p &signalno
$3 = (int *) 0xfffffc570
(gdb) x/32x $esp
0xfffffc550: 0x00000000 0x00000000 0x00000000 0x00000000
0xfffffc560: 0x00000000 0x56558fd0 0xffffd398 0xf7fc4560
0xfffffc570: 0x0000000b 0x00000063 0x00000000 0x0000002b
0xfffffc580: 0x0000002b 0xf7fcb80 0xffffd464 0xffffd398
0xfffffc590: 0xffffd380 0x56558fd0 0x00000000 0x00000000
0xfffffc5a0: 0x00000000 0x0000000e 0x00000004 0x56556230
0xfffffc5b0: 0x00000023 0x00010282 0xffffd380 0x0000002b
0xfffffc5c0: 0xffffc850 0x00000000 0x00000000 0x00000000

```

We can see the old program counter circled in orange. We note this value and then type `continue` to run until our second breakpoint is hit (signal\_handle). Next, we then print the value of the memory address for the int signalno, marked in yellow. This represents within the stack where the function was called initially.

Now, we want to find the distance (or offset) between where the function was called (which is in &signalno) and the memory address of the faulty instruction. To do this, we print the memory addresses (locations) nearby the stack pointer using:

```
x/32x $esp
```

To find the offset value, we know that each row is separated by 4 bytes, which means from 0xfffffc5a0, 0x56556230 is 12 bytes away. Then, we go up from 0xfffffc5a0 to 0xfffffc570 by 3 bytes. In total, 12+3 = 15 bytes.

### 3. What were the changes to get to the desired result?

In the code (stack.c), one line was added inside of the signal\_handle function provided, which is the following:

```
*(&signalno + offset) += lengthofbadinstruction
```

Here, we are first retrieving the memory address of the int signalno by using ampersand (&). Next, we add an offset, which is how far the old program counter is from the new one. We then dereference this memory address (since pointers are memory addresses) and increment the length of the bad instruction. By incrementing with this length, we get to the next instruction, which will be this line:

```
r2 = r2 + 1 * 45
```

By adding this one line of code in the signal\_handle function, we have effectively changed the program counter to run the instruction after the line that causes the segmentation fault.

From question 2, we were able to find the offset to be 15 and the length of the bad instruction to be 5 bytes.