Name and NetID:

Rohit Bellam – rsb204

Enrico Aquino - eja97

Part 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.

* **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:



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 0xffffc5a0, 0x56556230 is 12 bytes away. Then, we go up from 0xffffc5a0 to 0xffffc570 by 3 bytes. In total, 12+3 = 15 bytes.

* **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.

Part 2:

* Describe how you implemented the bit operations

Getting the top bits:

To get the top bit, we take the size of an address (32 bits) and then the num of bits we want from the front and subtract the two values to get the

difference / the total number of bits we need to right shift by. This effectively removes the unwanted lower bits, leaving only the top bits in place.

Setting Bit Index:

For set bit at index, we need to find which byte we need to modify,

we do this by taking the index and dividing it by the total number of bytes

then we need to find which bit we need to modify, and we do this by taking the (index of byte to modify \* 8 (size of the byte)), this will get the amount of bits into the bitmap the start of the byte it at.

Then we take the bit index and subtract what we just calculated from it to get the amount of bits into the current byte we are.

We will then left shift by the amount of bits into the current byte.

So, we will move 1 to the bit index by using left shift and then set that bit into the bitmap at that byte.

We can then use bitwise OR to set the bit index in the current byte

EG:

bit:17

what we want:

00000000,00000000,00000010,00000000

17/8 = 2nd index = 3 bytes in

00000000,00000000,**00000000**,00000000

17-(8\*2) = 1 index into that byte = 2 bits into that byte

Now if we just left shift by 2 bits, and set that bit, we would get.

**10**

So finally we can use bitwise OR | to set the bit without modifying other bits to give us

00000000,00000000,**00000010**,00000000

Getting bit index:

To get the bit index we will use the same logic as setting the bit up until the very end where we set it.

Instead of using bitwise OR to set, we will use bitwise AND to check if the bit at that position is set or not and see if the result doesnt equal zero, if it doesn't equal zero, then it is set. if it does equal zero, then it is not set.