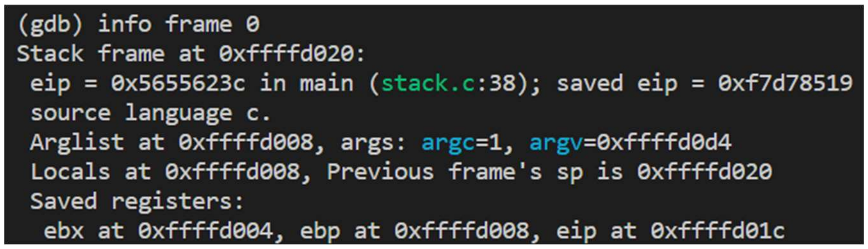
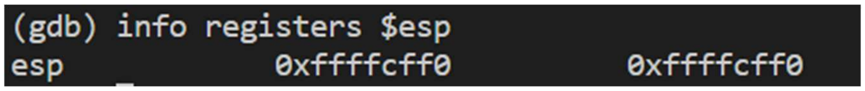
Pavitra Patel - php51
Kush Patel - kp1085
CS 416
ilab2

Report – 1.4

1. Let's understand description of the stack with following sample data:



```
(gdb) info frame 0
Stack frame at 0xffffd020:
 eip = 0x5655623c in main (stack.c:38); saved eip = 0xf7d78519
 source language c.
 Arglist at 0xffffd008, args: argc=1, argv=0xffffd0d4
 Locals at 0xffffd008, Previous frame's sp is 0xffffd020
 Saved registers:
  ebx at 0xffffd004, ebp at 0xffffd008, eip at 0xffffd01c
```



```
(gdb) info registers $esp
esp            0xffffcff0          0xffffcff0
```

- Every function has its own stack frame. Stack frame at level 0 corresponds to the function that is being executed currently.
- "Stack frame at 0xffffd020" indicates that the current stack frame is located at memory address 0xffffd020.
- "eip = 0x5655623c" in main (stack.c:38) – eip stands for Extended Instruction Pointer. eip is the program counter which points at the next instruction that will be executed.
- "saved eip = 0xf7d78519" is the value of eip saved from the previous stack frame where the control will return after the current function finishes execution.
- "Arglist at 0xffffd008, args: argc=1, argv=0xffffd0d4" – Argument list for the function is stored in Arglist. argc has a value of 1 which means that function was called with one argument and a pointer to argv which starts at memory address 0xffffd0d4.
- "Locals at 0xffffd008" – This is the memory address location where local variables for the current function are stored.
- "Previous frame's sp is 0xffffd020" – This is the memory address location of the previous stack frame.
- "Saved registers" – This section lists the saved registers in the current stack frame.
  i. ebx is saved at 0xffffd004
  ii. ebp is saved at 0xffffd008
  iii. eip is saved at 0xffffd01c
- ebp is the base pointer which points to the base of the stack frame at the beginning of the function
- esp is saved at 0xffffcff0 and points to the top of the stack and it changes as items are pushed or popped from the stack.

Additional Notes on how I approached the solution using gdb:

- There are three stack frames: main, kernel, signal_handle
- Understanding the details of stack frame above I accessed ebp and esp values of all stack frames
- I used "x/100x 0xffffcff0" command in gdb to observe values at addresses from esp's of these stack frames

- When I found the return address I calculated the offset
- In gdb I ran “layout split” and put breakpoints at the instruction causing segmentation fault and the next instruction that we wanted to jump to. By running the program in that layout with assembly and source code open, I observed the difference in bytes to jump to next instruction

2. eip is the program counter which points to the next instruction that will be executed. Suppose you are at a break point while debugging in gdb. When you run “info registers $eip” or view stack frame using “info frame” then we can see the memory address location of eip.

3. First, I got the address of where the signalno is stored and then added 60 bytes from that address. I converted int pointer to char pointer and added 60 which means adding 60 bytes. The int pointer is now pointing to the new address which is 60 bytes ahead of the location of the signalno. The value at this location is the return address of the instruction which will be executed after the signal handler finishes executing. I dereferenced the pointer to access the value stored at that location and added 5 to it which was the difference in bytes between the instruction that was causing segmentation fault and the next instruction that we wanted to jump to.

Report – 2.2

Bit Operations Implementation explanation for three methods:

1. Method – static unsigned int get_top_bits(unsigned int value, int num_bits):

For this method all we want is to access top order bits. The total bit length of the value stored is 32. Considering the below example if we want to access 4 top order bits then num_bits = 4

11110010 00101010 00110010 01001100

To simplify the output we can shift those 4 bits on the other end and the rest of the bits can be 0.

00000000 00000000 00000000 00001111

We will be able to access decimal value which is our desired output if we manage to perform this operation.

We can perform this operation by right shifting: 32 - num_bits = 28 bits. When we right shift by 28 bits we can achieve decimal value for 4 top order bits.

2. Method – static void set_bit_at_index(char *bitmap, int index):

Consider the following example:

11110000 00000000 00110010 01000000

If we want to set bit at 17th location, then we want to access 17th location from the right. The first location on right is location 0. We observe 4*8 = 32. Let’s say 4 segments or indexes of bitmap array of 8 bits. Now, we do 3 – (index/8) = 1 to get the segment or index in bitmap array.

We conclude, 17th location in the example above is located at bitmap[1]. There are 0 to 7 bits from right to left at bitmap[1]. When we do index%8 = 17%8 = 1, we can find the bit that we want to access at that index in array.

If we perform OR bitwise operation between bitmap[1] and 10, then we always get 1 for 17th location.

In the below instruction, we are left shifting bits by 1 or index%8 location for value 1 and we are performing the following operation which will give us results:

bitmap[1] = 00000000 | 10

bitmap[17] = 00000010

“bitmap[byte_index] = bitmap[byte_index] | (1 << bit_position);”

The rest of the locations remain unchanged due to the nature of OR operation.

3. Method – static int get_bit_at_index(char *bitmap, int index):

Considering same example as in 2: 11110000 00000010 00110010 01000000

Using the similar approach as in 2, we can access bitmap[i] and the bit in bitmap[i] that we want to get the value for.

int result = ((bitmap[byte_index] & (1 << bit_position)) >> bit_position);

(1 << bit_position) = (1 << 1) = 10

00000010 & 10 = 00000010 (Due to nature of AND operation, 0 & 1 = 0 and 1 & 1 = 1, hence preserving only the bit we want)

Now, we do 00000010 >> 1 = 00000001 = 1 (decimal value)

Now, we return 1 for this function.