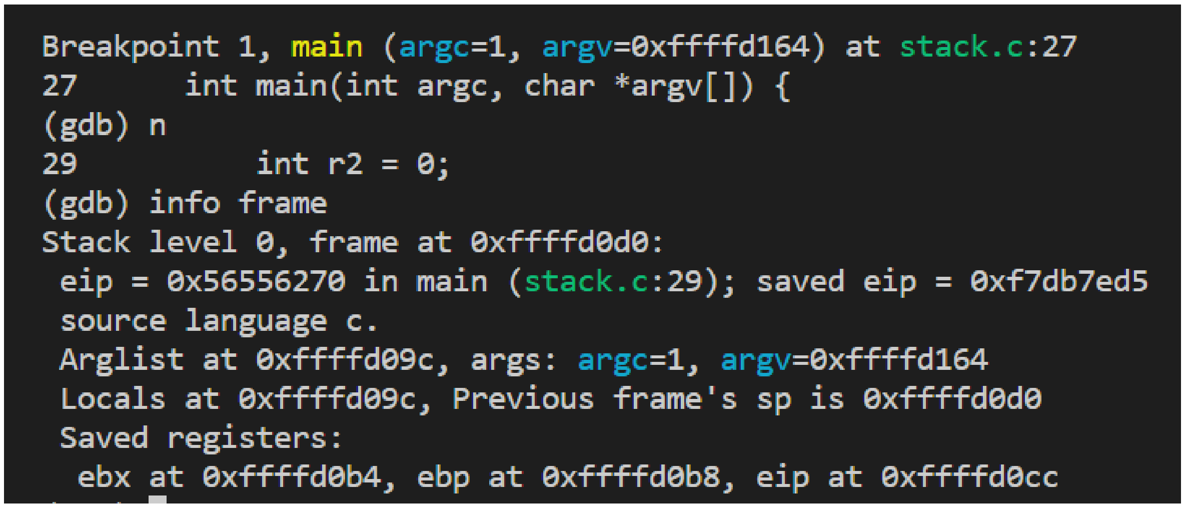
# Report

Net ID: yp315 | Yash Patel , kks107 | Kev Sharma
CS 416 - Undergraduate OS
Tested on ls.cs.rutgers.edu && man.cs.rutgers.edu

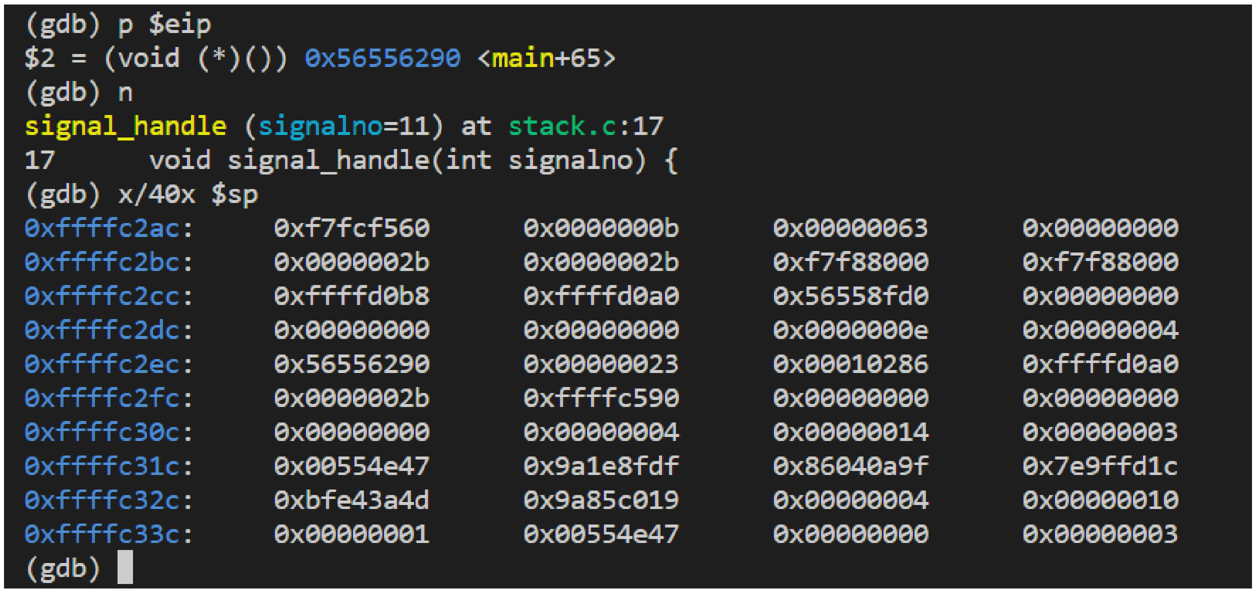
## Stack.c

After we run the first line in the main and look into the stack frame we can see what's in the stack currently



```
Breakpoint 1, main (argc=1, argv=0xffffd164) at stack.c:27
27      int main(int argc, char *argv[]) {
(gdb) n
29          int r2 = 0;
(gdb) info frame
Stack level 0, frame at 0xffffd0d0:
 eip = 0x56556270 in main (stack.c:29); saved eip = 0xf7db7ed5
 source language c.
 Arglist at 0xffffd09c, args: argc=1, argv=0xffffd164
 Locals at 0xffffd09c, Previous frame's sp is 0xffffd0d0
 Saved registers:
  ebx at 0xffffd0b4, ebp at 0xffffd0b8, eip at 0xffffd0cc
```

1. This the current stack frame after r2 is set to value to 0. We can see that we know the address of the instruction pointer (program counter) at %eip we know where the argument lists are and what registers are saved. Generally stack stores our local variables like r2 in this example and saves it to a register like %ebp in this. To know where currently our stack is we have stack pointer or **SP** if there is a call to a different function the return address is pushed to stack and we are in a different frame storing data for a different function
2. The pointer counter is stored in the %eip register so in gdb we can simply do p $eip to print out its contents. If you want to find the address you will have to examine the stack pointer we can do that by x/(# of characters)x $sp and it will print out something like this



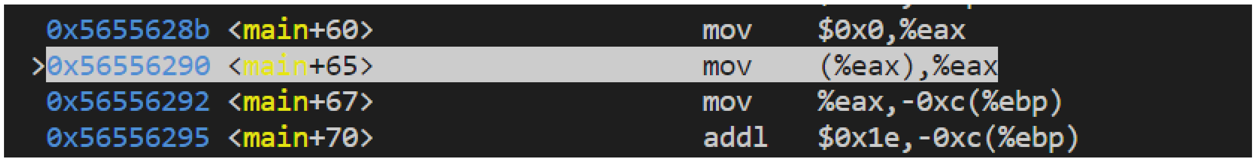
```
(gdb) p $eip
$2 = (void (*)()) 0x56556290 <main+65>
(gdb) n
signal_handle (signalno=11) at stack.c:17
17      void signal_handle(int signalno) {
(gdb) x/40x $sp
0xffffc2ac:     0xf7fcf560      0x0000000b      0x00000063      0x00000000
0xffffc2bc:     0x0000002b      0x0000002b      0xf7f88000      0xf7f88000
0xffffc2cc:     0xffffd0b8      0xffffd0a0      0x56558fd0      0x00000000
0xffffc2dc:     0x00000000      0x00000000      0x0000000e      0x00000004
0xffffc2ec:     0x56556290      0x00000023      0x00010286      0xffffd0a0
0xffffc2fc:     0x0000002b      0xffffc590      0x00000000      0x00000000
0xffffc30c:     0x00000000      0x00000004      0x00000014      0x00000003
0xffffc31c:     0x00554e47      0x9a1e8fdf      0x86040a9f      0x7e9ffd1c
0xffffc32c:     0xbfe43a4d      0x9a85c019      0x00000004      0x00000010
0xffffc33c:     0x00000001      0x00554e47      0x00000000      0x00000003
(gdb)
```

Where we can examine and print out what is in our stack with the address.

3. From the last question we know how to examine the stack. If we look into the previous picture we know that the program counter is address that is main+65 (p $eip). We will just look for that when we examine the stack when we go into our handler because we know that the counter is pushed. We examine the stack right after we call the function so its easier to find it. As we can see in the pic we can find that address **0xffffc2ec** has the value that we are looking for.

   So now its simple pointer arithmetic to get to that address. We can get signalno address by p &signalno = **0xffffc2b0** because this is a int ptr when we add 1 in gdb it will add 4 to the address so we need to find the offset and divide by 4.

   If we do hex **ec - b0** we get 60 in decimal. We noticed that we have to divide by 4 so that will be 15 so we just have to add 15 to the address of signalno. To skip the offending instruction we have to look into the assembly and look at what is causing the problem and skip it.



```
 0x5655628b <main+60>                    mov    $0x0,%eax
>0x56556290 <main+65>                    mov    (%eax),%eax
 0x56556292 <main+67>                    mov    %eax,-0xc(%ebp)
 0x56556295 <main+70>                    addl   $0x1e,-0xc(%ebp)
```

   The picture shows the instruction that causing the problem the next instruction is at main + 67 which is 2 from the current so if we add 2 to the program counter we should be to the next instruction where it will continue executing without problems. So we just have to get the int ptr by &signalno + 15 and then we can access whats inside and add by 2 by *ptr += 2.

**Bitops.c**

```
static unsigned int get_top_bits(unsigned int value,  int num_bits)
{
        //Implement your code here
    unsigned int ns, n; /* ns: number of bits (value takes up) */
    for (n = value, ns = 0; n; n >>= 1, ++ns);

    int k = ns - num_bits;
    return (((~0 << k) & value) >> k);
}
```

Consider the supplied number 4026544704. In binary this is:

$$\begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 2^{31} & 2^{30} & 2^{29} & 2^{28} & 2^{27} & 2^{26} & \dots & 2^{3} & 2^{2} & 2^{1} & 2^{0} \end{array}$$

In order to extract the first **n** bits, we create a mask (a bit vector of size 32) to bitwise AND against the supplied value and then truncate the **32-n** remaining zeros.

Our mask must align with the most significant bit of the value: 4026544704 takes up 32 bits and the MSB is the thirty-second bit at $2^{31}$. Counting left to right, we ensure our mask has **n** 1s, followed by 0s. This is to guard against extracting more bits than just the first n.

In our case, **n = 4**. Hence our mask should be:

$$\begin{array}{cccccccccccc} 1 & 1 & 1 & 1 & 0 & 0 & \dots & 0 & 0 & 0 & 0 \\ 2^{31} & 2^{30} & 2^{29} & 2^{28} & 2^{27} & 2^{26} & \dots & 2^{3} & 2^{2} & 2^{1} & 2^{0} \end{array}$$

~0 results in a bit vector where each bit in the vector is a 1. By applying consecutive logical left shifts to ~0, wherein we append our bit vector with a 0 and shift the remaining bits to the left, we can arrive at the previously shown mask.

Working with the assumption that our number shall always be at most 4 bytes (unsigned int) = 32 bits in length, we can apply **k = [32 - n] = 28** logical left shifts to ~0. This is equivalent to taking a list of 1s, appending k zeros to the list, and collecting the last 32 elements. After computing the bitwise AND between our mask (~0 << **k**) and value, we arrive at the value 15 ($2^3 + 2^2 + 2^1 + 2^0$) by applying **k** logical right shifts to our result. These right shifts truncate the zeros following the nth top bit we wanted.

It is not necessary that the supplied value is exactly 32 bits long - it can be less. Hence we apply consecutive logical right shifts to derive the variable **ns** (see code) until value becomes 0. We thus determine the length of value in binary (x such that $2^{x+1}$ > value where x <= 32).

The computation for **k** then becomes k = [ns - number of top bits] and we create the correct mask.

Get + Set a bit in a **Bitmap:**

- We shall consider a Bitmap with 4 bytes. 00000000 00000000 00000000 00000000

Counting begins with index 0. Our bitops.c get and set functions facilitate counting from left to right or from right to left but demand indexing begins at zero. The direction to begin counting is set to LEFT_TO_RIGHT by default and can be reconfigured to RIGHT_TO_LEFT by resetting the global enum *orientation* as shown below.

```
enum ORIENTATION {
    RIGHT_TO_LEFT, /* Set third bit = 0100 */
    LEFT_TO_RIGHT /* Set third bit = 0010 */
};

enum ORIENTATION orientation = LEFT_TO_RIGHT;
```

```
bitmap[byte_offset] |= (1 << shift_by);
```

**Set:** In order to set a bit, we must bitwise OR it with 1. 1 | 1 = 1 and 0 | 1 = 1 ; therefore, it doesn't matter if the bit was previously set.

```
return (bitmap[byte_offset] & (1 << shift_by)) > 0;
```

**Get:** In order to check if a bit is set, we must bitwise AND it with 1. If the bit was not set, we would get 0 & 1 = 0. The return statement would then return 0 for false. If the bit was set, we would get 1 & 1 = 1 which is greater than 0 and so the return statement yield 1 for true.

The heart of the problem lies at indexing at the correct bit in the bitmap and forming the proper mask. Let's work with index 17 for example counting from left to right.

00000000 00000000 **01000000** 00000000

To index into the proper byte {the bit vector in the bitmap}, we divide 17 by 8. Because each byte contains at most 8 bytes, computing this dividend yields the index of the group the bit is in. The first group (index 0) contains bits from 0 to 7, the second from 8 to 15, the third from 16 to 23, and the fourth group contains bits from 24 to 31. Accordingly, our byte_offset is 2.

17 modulo 8 is 1. The criteria for our mask is that only a single bit be set. Since the bit vector contains 8 bits, the highest possible value of the bit vector is $2^7$. $2^7$ = 128 = 1 << 7. Because we seek the second bit, we want $2^6$. It is easy to figure out then that **7 - (index % 8)** gives us the number of shifts needed to place the bit in the correct position. This is our shift_by value.

*To make our code less trivial, we allow you to choose whether you want the bit set when counting left to right or from right to left. The code uses the enum flag to index into the appropriate byte and form the desired mask for GET and SET.*