CS 518 - Project 1: Stacks and Basics (Warm-up Project)

Submitted by Anmol Arora (aa2640) and Raunak Negi (rn444)

Part 1: Signal Handler and Stacks (35 Points)

Q1: What are the contents in the stack ? Feel free to describe your understanding.

Ans: The stack includes the following contents:

- EBP(Extended Base Pointer) of the previous function's stack frame.
- The Local variables in the order in which they are declared.
- Values stored in some of the registers are added on to the stack frame of that function for future references.(Ex. eip, eax, ebp, etc.)
- The arguments for the next function call. These are stored in the reverse order in which they are declared.
- Finally, the return address of the previous function. The **EIP** (Extended Instruction Pointer) register stores the next instruction to execute (also known as **Program Counter**)

Q2: Where is the program counter, and how did you use GDB to locate the PC?

Ans: The EIP register stores the address of the next instruction that needs to be executed which is exactly what the program counter does as well. In order to find the value stored in the eip register we use the command "**info registers eip**". The value of eip register will change depending on different stages of execution of the program.

For the current example we tried to find the value of PC at different stages of execution:

1. Segmentation Fault occurs:

Figure 1.

This is the address of the instruction that causes the segment fault and also the return address for the main function meaning that this is where the program execution will begin after the “signal_handler” function returns/ finishes execution(i.e **0x5655629f**).This will result in yet another segmentation fault, perpetuating an endless loop of errors.

As per the project requirements, we must modify this return address to the next instruction in the main function. This change should **prevent** a segmentation fault and allow the main function to exit gracefully.

2. After Signal Handler function is called:

Q3: What were the changes to get the desired result?

Ans: In order to get the desired results we have to identify the memory address in the stack frame that stores the return address of the main function. Once we have the memory address we can modify the value of the return address to the address of the next instruction in the main function that does **NOT** result in segmentation fault.

In order to locate the memory address in the stack frame that stores the return address we first need to know how many stack frames are there in the running program. We do this by using the "**backtrace**" or "**where**" command in the GDP.

```
Program received signal SIGSEGV, Segmentation fault.
main (argc=1, argv=0xffffd0c4) at stack2.c:39
39          r2 = *( (int *) 0 ); // This will generate segmentation fault
(gdb) layout split
(gdb) n
signal_handle (signalno=11) at stack2.c:13
13      void signal_handle(int signalno) {
(gdb) n
15          printf("handling segmentation fault!\n");
(gdb) bt
#0  signal_handle (signalno=11) at stack2.c:15
#1  <signal handler called>
#2  main (argc=1, argv=0xffffd0c4) at stack2.c:39
(gdb)
```

We get to know that there are **three** stack frames. Frame 1 corresponding to the "<signal handler called>" will have the return address of the main function. Therefore we inspect the values stored in frame 1 using the GDB command "**info frame 1**".

```
(gdb) info frame 1
Stack frame at 0xffffcfe0:
 eip = 0xf7fc4560 in __kernel_sigreturn; saved eip = 0x5655629f
 called by frame at 0xffffd010, caller of frame at 0xffffc1f0
 Arglist at unknown address.
 Locals at unknown address, Previous frame's sp is 0xffffcfe0
 Saved registers:
  eax at 0xffffc220, ecx at 0xffffc21c, edx at 0xffffc218, ebx at 0xffffc214,
  ebp at 0xffffc20c, esi at 0xffffc208, edi at 0xffffc204, eip at 0xffffc22c
```

We see that the saved eip shows the value of the return address of the main function.
This value is saved on the stack at a particular address which is mentioned in saved registers section of the output.(i.e **eip at 0xffffc22c**).

In the signal_handle function we have access to the address of the "signalno" parameter.(i.e **0xffffc1f0**)

```
16          int* ptr = &signalno;
(gdb) n
18          printf("Address of signalno: %p \n", ptr);
(gdb) n
Address of signalno: 0xffffc1f0
29      }
```

=> We need to go from address **0xffffc1f0** to **0xffffc22c** where the return address of the faulty instruction (i.e **0x5655629f**) is stored and modify it to the the address of the next instruction in the main which does **NOT** lead to segmentation fault.

Calculating the offset manually leads to 15. (0xffffc22c - 0xffffc1f0 = 3c => 60 in decimal and 60 bits /4 = 15 Bytes) and hence we update the ptr by 15 inside the segment_handle.
As observed in figure 1 the length of the faulty instruction is 2 bits. Hence we update the return address by 2 bits.

Part 2: Bit Manipulation (35 Points)

Q: Describe how you implemented bit manipulation?

Ans: The entire code consists of 3 major functions:

1. **Get_top_bits:**
   This C function takes two arguments: an unsigned integer value and a signed integer num_bits. The function is used to extract the top ‘num_bits’ bits from the binary representation of ‘value’. To get there, we initialize an unsigned integer variable ‘rev’ to 0,which will store the reversed binary representation of ‘value’.

   Next we move into the bit reversal loop, where we iterate through the bits of ‘value’ one by one. For every iteration, we left shift the variable ‘rev’ by 1 bit to make room for the next bit. While doing this, we also check if the least significant bit of ‘value’ is 1, and if it is, then we set the least significant bit of ‘rev’ using XOR operation. Finally, we right shift the ‘value’ by 1 bit to examine the next bit.

   Next, we initialize another unsigned variable ‘ans’ to 0, which will store the extracted bits from the reversed value ‘rev’. Similar to the loop above, we iterate through the bits of reversed value ‘rev’ till we extract ‘num_bits’ from it.

   We start off by left shifting the ‘ans’ variable by 1, check if the least significant bit of ‘rev’ is 1, and finally right shift ‘rev’ by 1 bit. Crucial difference here is that we need to decrement ‘num_bits’ by 1 for every bit we examine so that we get a specified number of bits, for example 5, 10, or 4 bits in this case.

   Finally, we return the extracted bits stored in the variable ‘ans’.

   In summary, this code reverses the bits of an input unsigned integer value and then extracts the top num_bits bits from the reversed value. It uses bitwise operations to perform these operations.

2. **Set_bit_at_index:**
   This C function is designated to set a bit at a specified ‘index’ in a bitmap represented as an array of characters. For the code, we have a bitmap of size 4, and each block of 8 bits corresponded to a single character, so we had 32 bits of memory to deal with.

In order to set a bit at a specific position, we had to determine where the bit character is located in the bitmap array. We initialize an unsigned integer ‘bitmapPos’ as “index/8”, which would correspond to its position in the array.

Next, we had to figure out the position of the bit that is to be set, and initialize an unsigned integer ‘maskPos’ to be “index%8”, where the remainder indicated which bit within the byte needed to be set.

Now we needed to set this bit, and for that we declared another unsigned integer ‘setBit’ as “1<<maskPos”, this left shifted the set bit ‘maskPos’ number of times. This left us with bitmask with a 1 in the position corresponding to the target bit within the byte.

Finally, we perform the OR operation between ‘bitmap[bitmapPos]’ and ‘setBit’ such that it sets the target bit to 1 within the byte, while leaving the other bits unchanged. The results are stored back to ‘bitmap[bitmapPos]’, updating the bitmap with the modified bit.

In summary, this function takes a bitmap represented as an array of characters, calculates the byte and bit positions corresponding to the target index, and then sets the bit at that index within the bitmap using bitwise operations.

3. **Get_bit_at_index:**
   This function is designed to retrieve the value (0 or 1) of a specific bit at a given index in a bitmap represented as an array of characters.

   Similar to set_bit_at_index, we initialize ‘bitmapPos’ and ‘maskPos’ as “index/8” and “index%8” respectively. Next, we initialize ‘readBit’ as “bitmap[bitmapPos] << (8 - maskPos)”, where we retrieve the byte where the target bit is placed, and left shift it by “8-maskPos” times so that it becomes the most significant bit.

   To extract this significant bit, we reuse the ‘get_top_bits’ function to retrieve this bit and set the parameter ‘num_bits’ as 1, since our target bit is the most significant bit. Finally, we return the result.

   In summary, this function takes a bitmap represented as an array of characters, calculates the byte and bit positions corresponding to the target index, retrieves the value of the target bit using bitwise operations, and returns it.

References:
Recitation 1 & 2
https://stackoverflow.com/questions/5144727/how-to-interpret-gdb-info-frame-output
https://blog.devgenius.io/understanding-the-stack-a-precursor-to-exploiting-buffer-overflow-8c6972fdb4ac