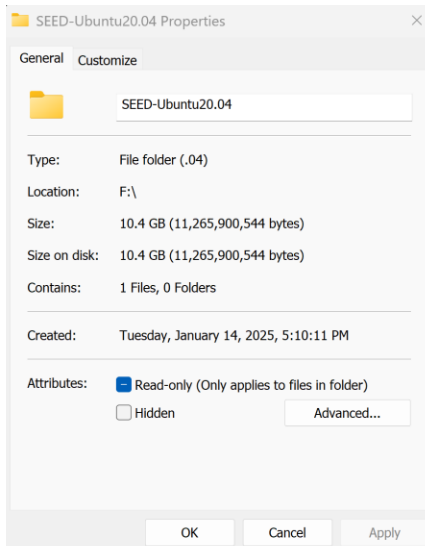
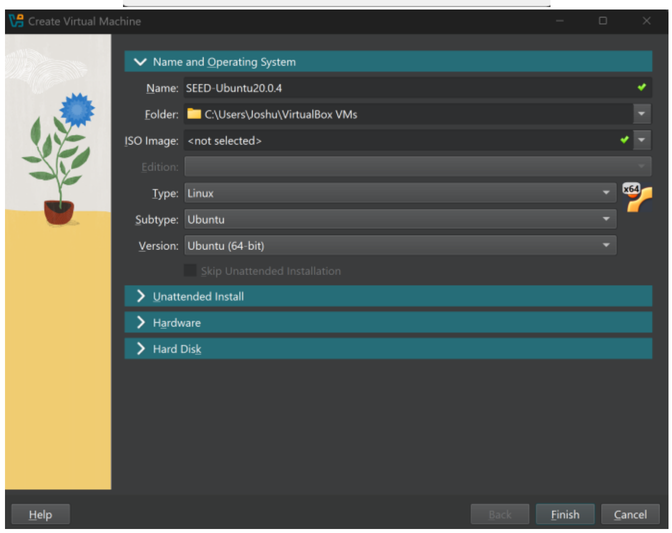
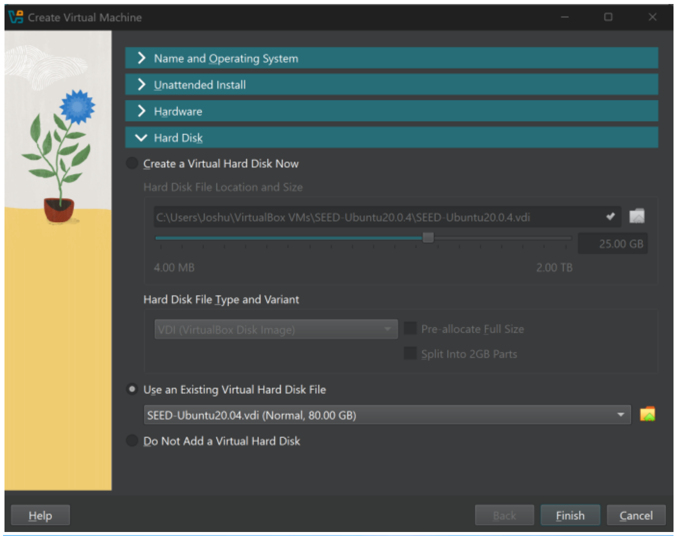
Lab 2: Using SEED Labs Virtual Image

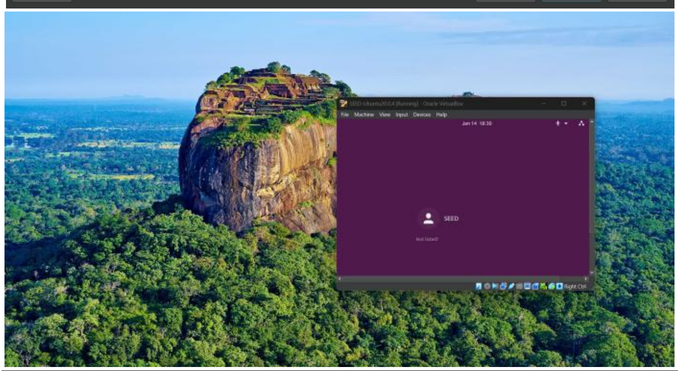
Joshua Ludolf
CSCI 4321
Computer Security

- I downloaded Ubuntu 20 image from the link for this SEED lab and future ones.

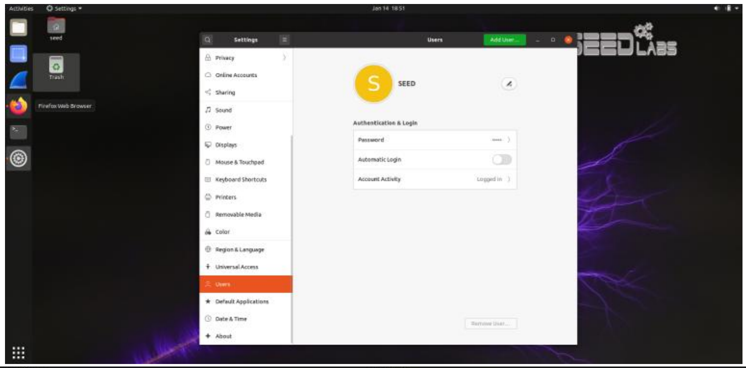


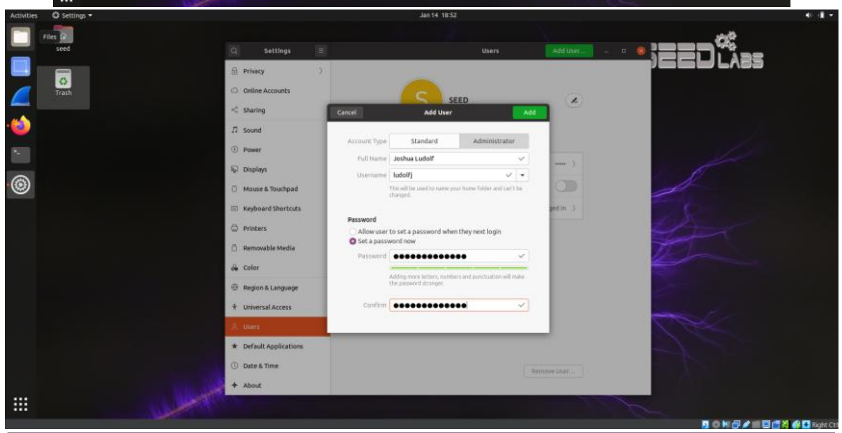


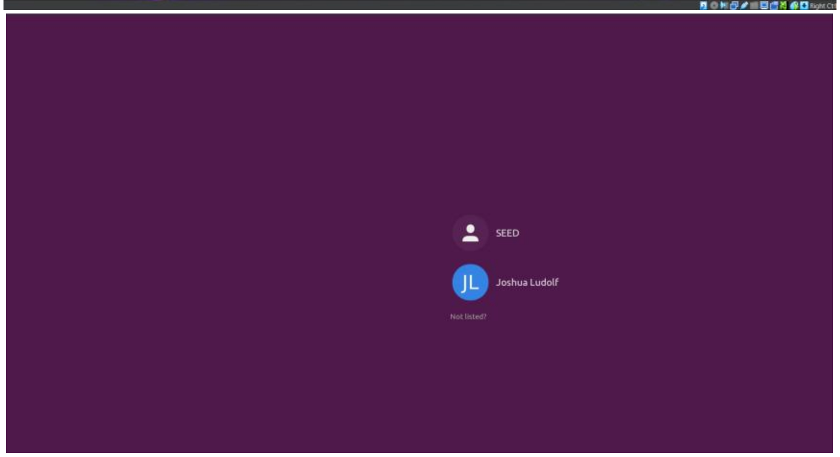


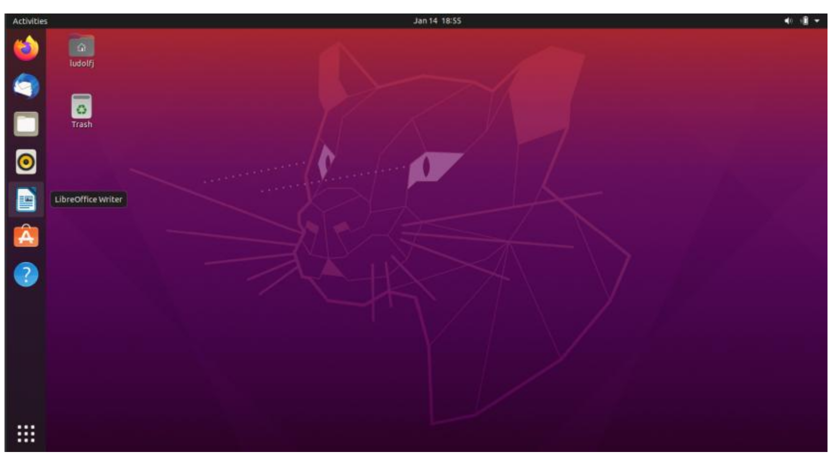



- I proceeded to create my own account by going to settings and clicked on the green Add User button:

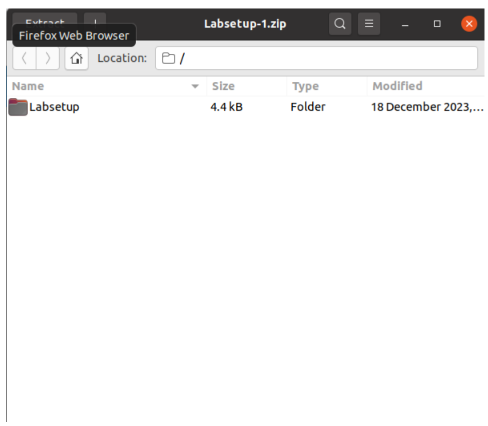

Settings
Users
Add User...
SEED
Authentication & Login
Password
Automatic Login
Account Activity
Privacy
Online Accounts
Sharing
Sound
Power
Displays
Mouse & Touchpad
Keyboard Shortcuts
Printers
Removable Media
Color
Region & Language
Universal Access
Users
Default Applications
Date & Time
About

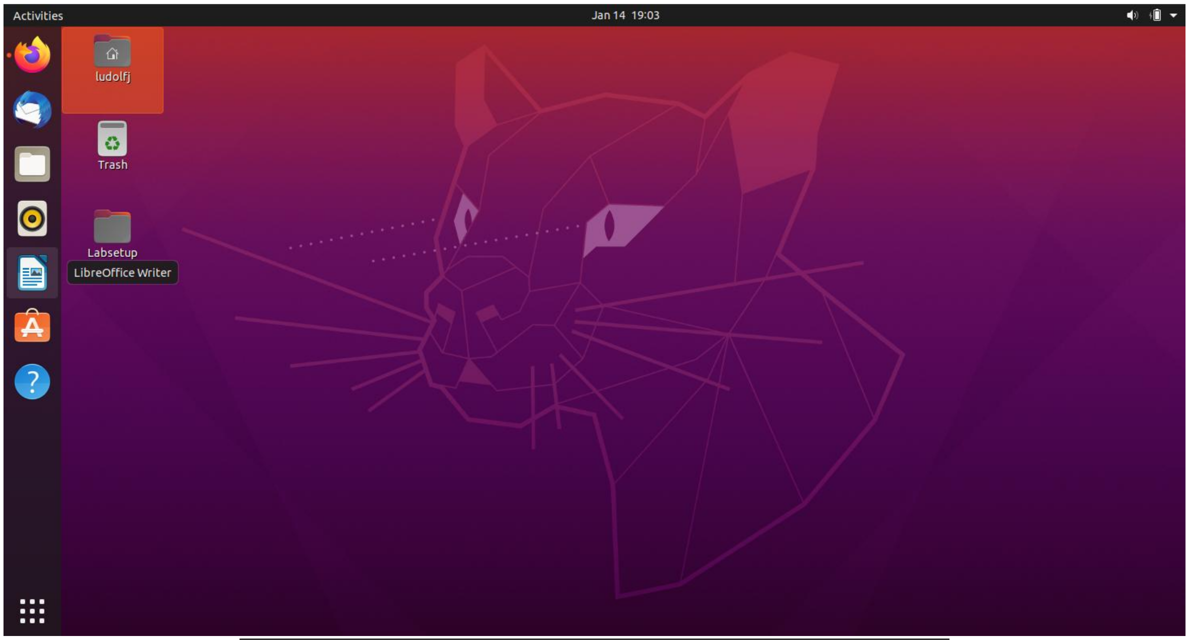

Settings
Users
Add User...
Cancel
Add User
Add
Account Type
Standard
Administrator
Full Name
Joshua Ludolf
Username
ludolfj
Password
Allow user to set a password when they next login
Set a password now
Confirm
Remove User...

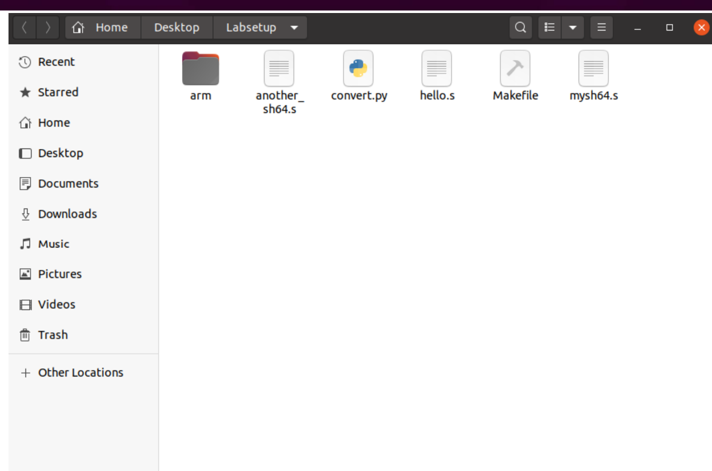

SEED
JL
Joshua Ludolf
Not listed?


Activities
Jan 14 18:55
ludolfj
Trash
LibreOffice Writer

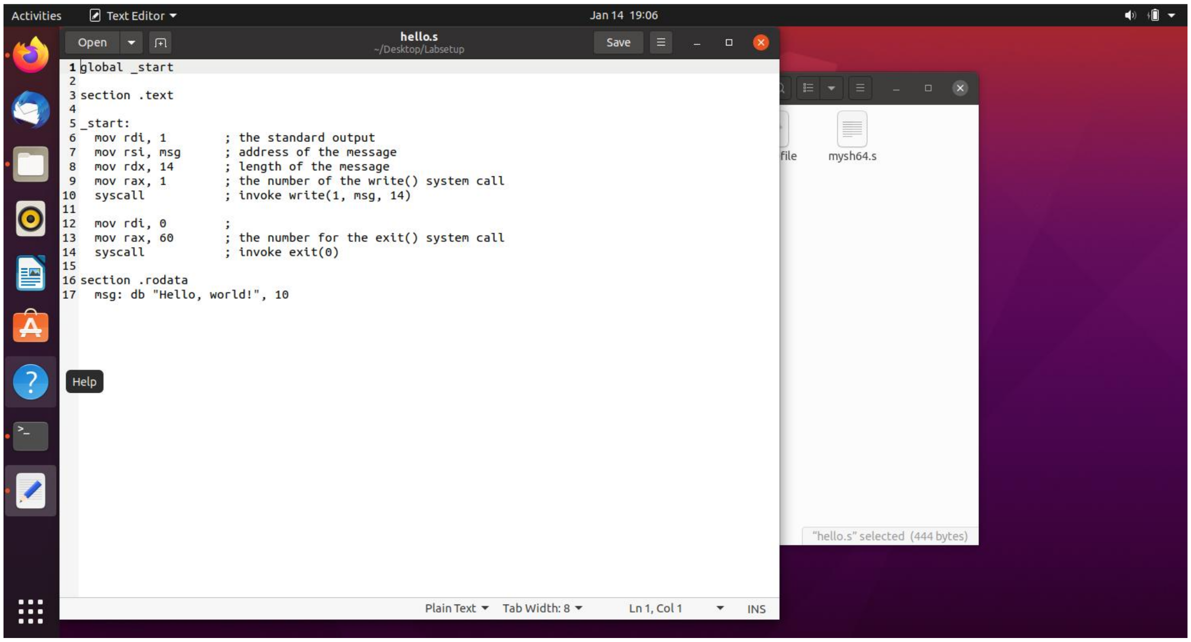
- In the SEED image I went to following link to get files to setup lab - https://seedsecuritylabs.org/Labs_20.04/Software/Shellcode/:



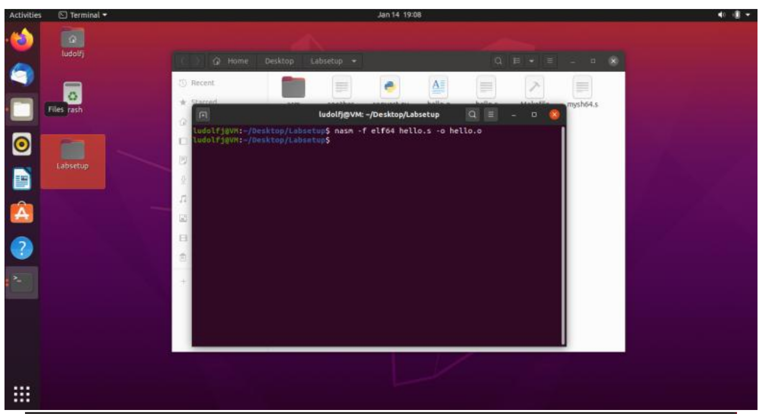


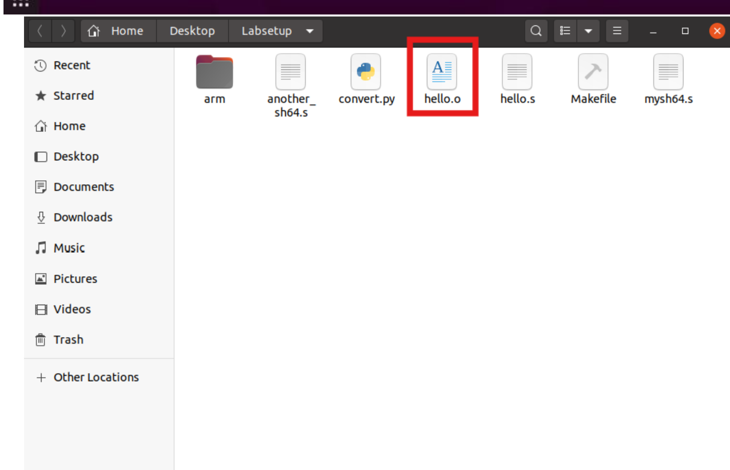


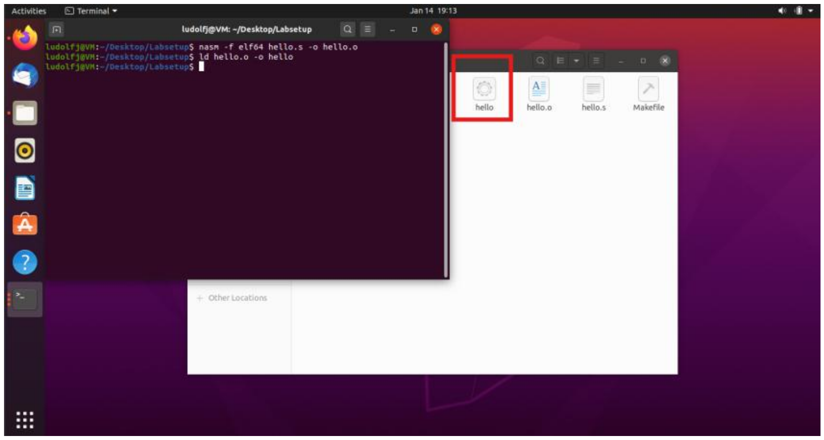
- I reviewed the contents in hello.s to ensure that they matched task 1 image of same file:



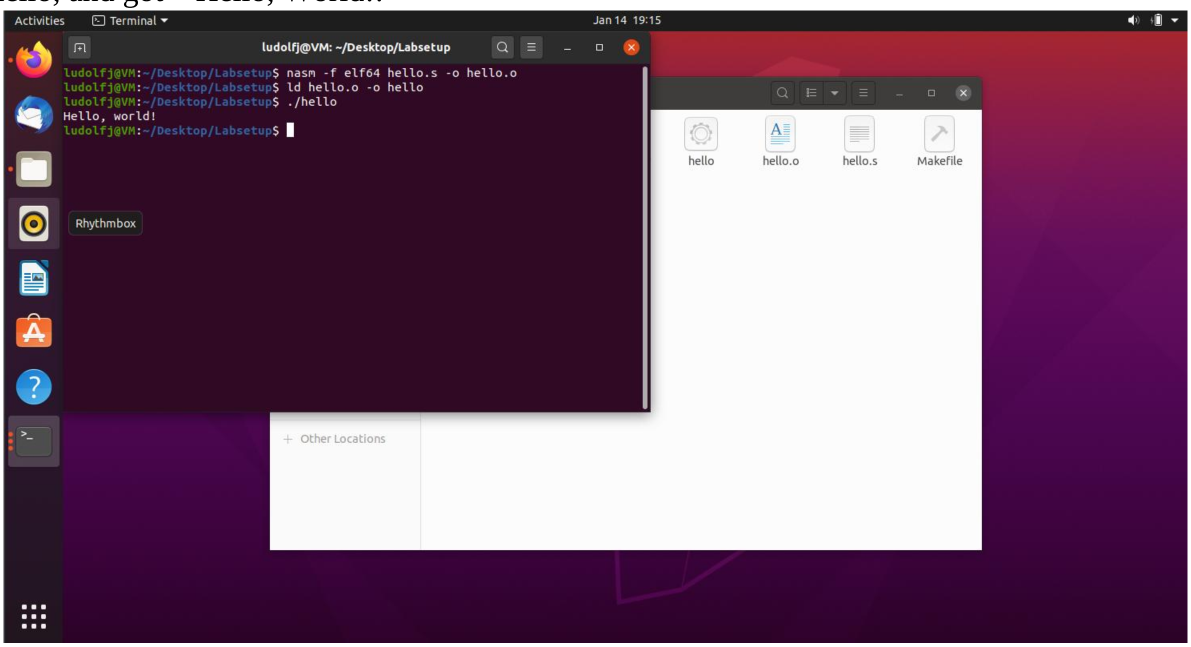
- From there, I proceeded to open terminal in the labsetup directory and ran following command – nasm -f elf64 hello.s -o hello.o and ld hello.o -o hello:



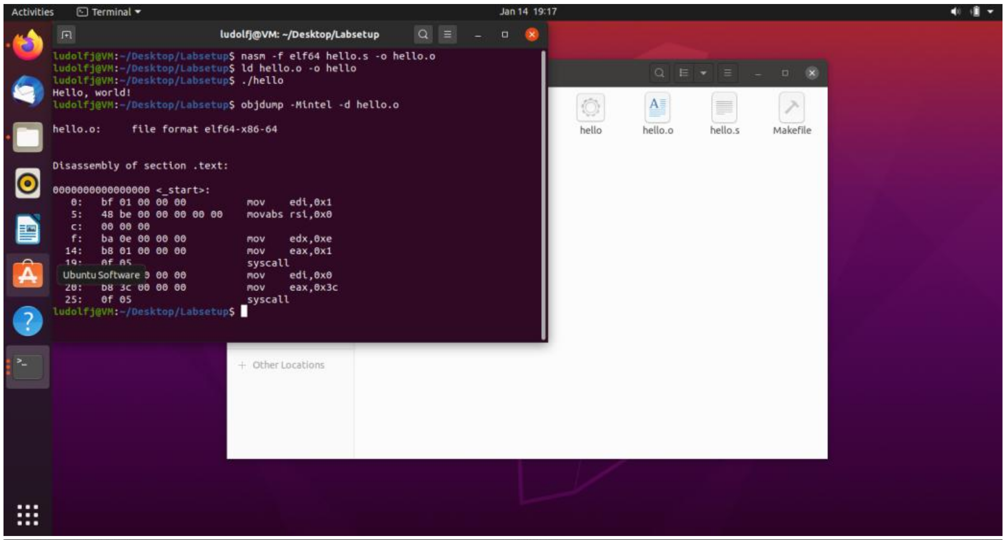


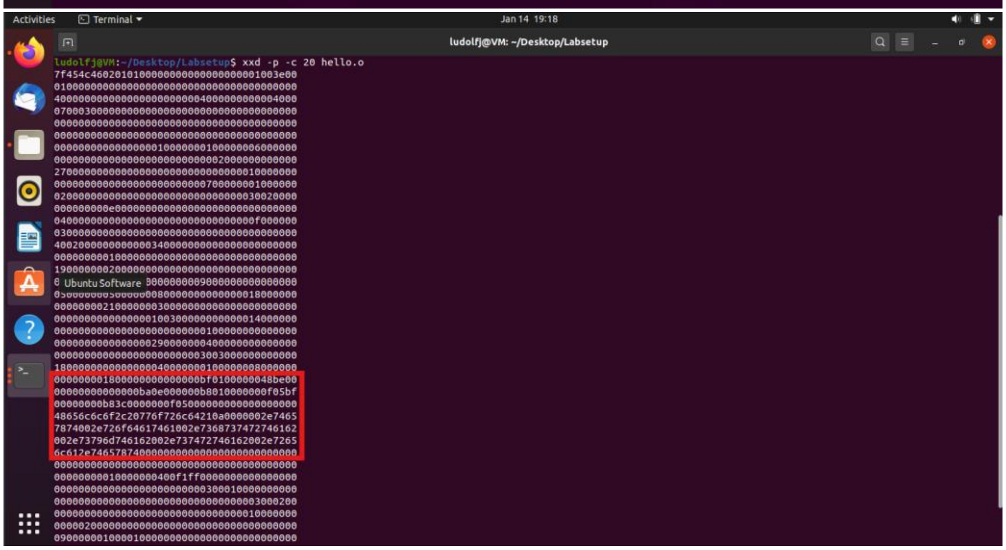


- After executing command to create executable binary, I executed it with following command - ./hello, and got – Hello, World!:



- Now, I shall obtain the machine code with following command - objdump -Mintel -d hello.o:

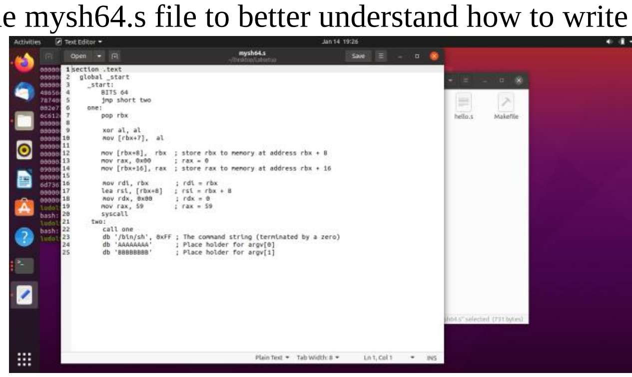




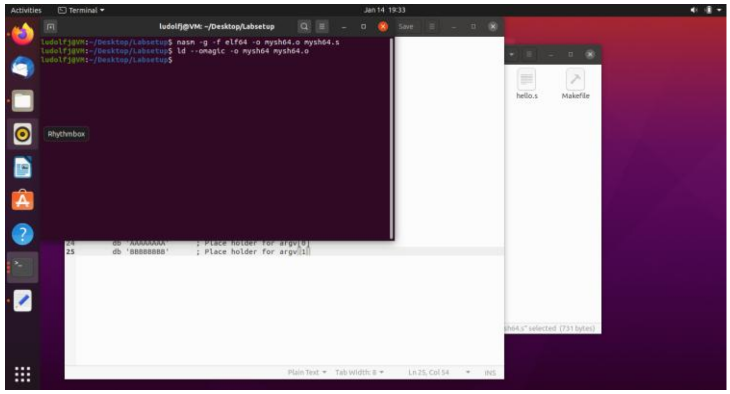
- Next, I dove to the first approach of writing shell code and executed this command - execve("/bin/sh", argv[], 0) and learned it gonna require some arguments (3 - rdi, rsi, and rdx for their registers):

```
ludolfj@VM:~/Desktop/Labsetup$ execve("/bin/sh", argv[], 0)
bash: syntax error near unexpected token `"/bin/sh",'
```

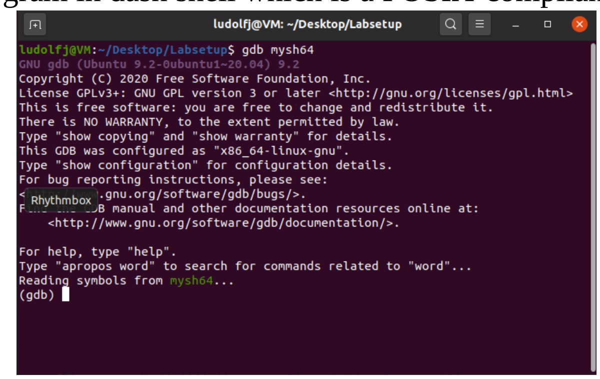
- From there, I opened the mysh64.s file to better understand how to write shell code:



- Then, I wanted to debug it to see how much insight I could gather from this shell code. I executed the following commands nasm -g -f elf64 -o mysh64.o mysh64.s and ld --omagic -o mysh64 mysh64.o:



- Next step, was to now run debug command – gdb mysh64 and then when gdb started I inputted run which started the program in dash shell which is a POSIX-compliant Unix shell:



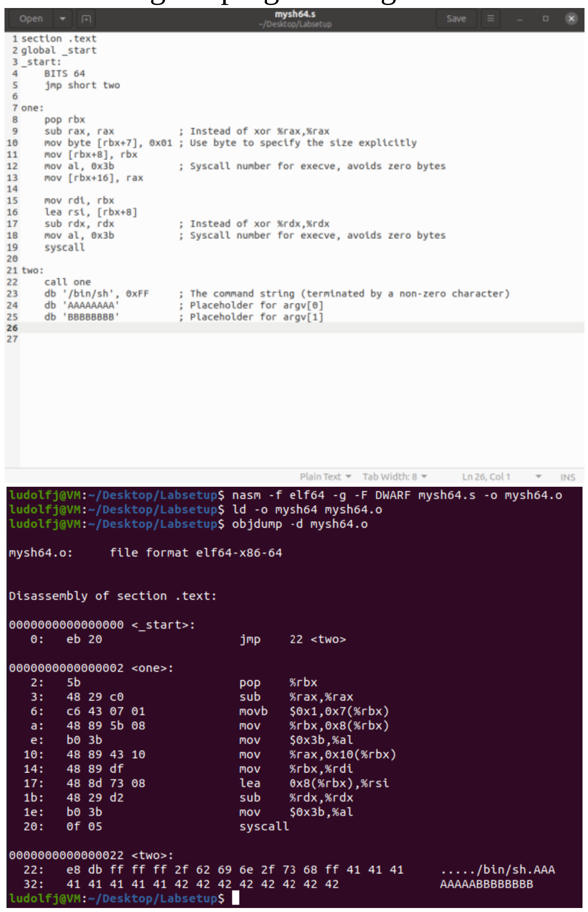
- From, there I created a breakpoint from *two (for task 2 in 3.1) and started stepping through the debug of mysh.s program:

```
ludolfj@VM:~/Desktop/Labsetup$ gdb mysh64
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
T     "show copying" and "show warranty" for details.
T      DB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mysh64...
(gdb) break *two
Breakpoint 1 at 0x401028: file mysh64.s, line 22.
(gdb) run
Starting program: /home/ludolfj/Desktop/Labsetup/mysh64

Breakpoint 1, two () at mysh64.s:22
22              call one
(gdb) info registers
rax            0x0                 0
rbx            0x0                 0
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffe090      0x7fffffffe090
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x401028            0x401028 <two>
eflags         0x202               [ IF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) x/32x $rip
0x401028 <two>: 0xffffd5e8      0x69622fff      0x68732f6e      0x414141ff
0x401038:       0x41414141      0x42424241      0x42424242      0x00002c42
0x401048:       0x00000300      0x08000000      0x00000000      0x40100000
0x401058:       0x00000000      0x00004500      0x00000000      0x00000000
0x401068:       0x00000000      0x00000000      0x00000000      0x00000e00
0x401078:       0x00000300      0x00000000      0x00000000      0x46000000
0x401088:       0x03000000      0x00000000      0x00010800      0x00004010
0x401098:       0x45000000      0x00004010      0x00000000      0x6d000000
(gdb) stepi
one () at mysh64.s:7
7               pop rbx
(gdb) stepi
o                4.s:9
9               xor al, al
(gdb) info registers
rax            0x0                 0
rbx            0x40102d            4198445
rcx            0x0                 0
rdx            0x0                 0
rsi            0x0                 0
rdi            0x0                 0
rbp            0x0                 0x0
rsp            0x7fffffffe090      0x7fffffffe090
r8             0x0                 0
r9             0x0                 0
r10            0x0                 0
r11            0x0                 0
r12            0x0                 0
r13            0x0                 0
r14            0x0                 0
r15            0x0                 0
rip            0x401003            0x401003 <one+1>
eflags         0x10202             [ IF RF ]
cs             0x33                51
ss             0x2b                43
ds             0x0                 0
es             0x0                 0
fs             0x0                 0
gs             0x0                 0
(gdb) x/s $rbx
0x40102d:       "/bin/sh\377AAAAAAAABBBBBBBB,"
```

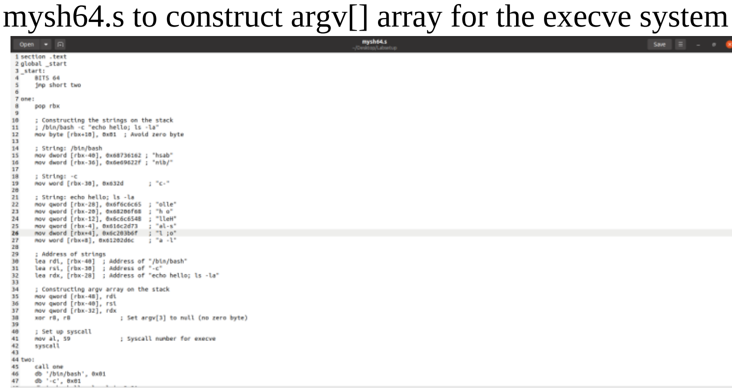
- From that I learned:
  - The pop rbx instruction retrieves the address of the /bin/sh string from the stack and stores it in the rbx register.
  - The address 0x40102d points to the beginning of the /bin/sh string in memory.
  - This string is used later in the program when calling the execve system call to start the shell.
  - For task 3 (section 3.1) I kept getting Segmentation fault and that the program no longer exists. However for task 4 (section 3.1), line 1 indicates that the following code belongs to the .text section of the executable. The .text section is typically used for storing the

executable code (i.e., the instructions that the CPU will run). Line 2 declares the _start symbol as global. In other words, it tells the assembler and linker that _start should be accessible from other object files or by the operating system loader. _start is usually the entry point of the program, where execution begins.

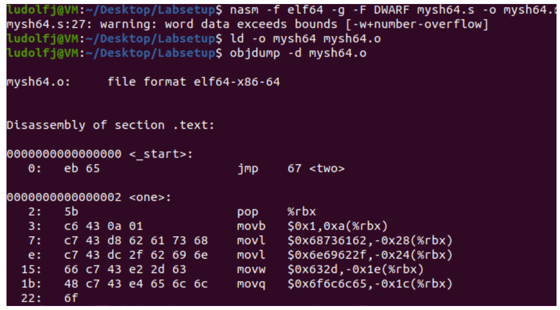
- For task 2.b, I eliminated zeros from the code by changing the termination characters from zero to non-zero values like 0x0 and loading values into registers that were adjusted to non-zero where possible, while still maintaining the program's logic.



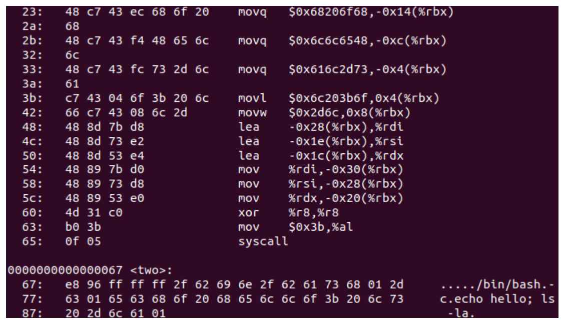
- Task 2.c, I updated mysh64.s to construct argv[] array for the execve system call.



```
48    db 'echo hello; ls -la', 0x01
49
```

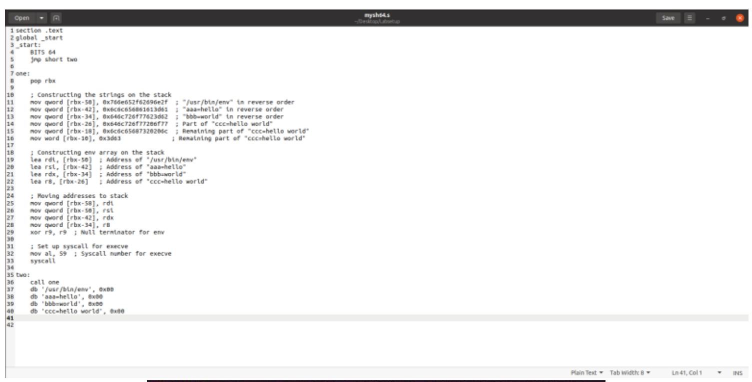


```
ludolfj@VM:~/Desktop/Labsetup$ nasm -f elf64 -g -F DWARF mysh64.s -o mysh64.o
mysh64.s:27: warning: word data exceeds bounds [-w+number-overflow]
ludolfj@VM:~/Desktop/Labsetup$ ld -o mysh64 mysh64.o
ludolfj@VM:~/Desktop/Labsetup$ objdump -d mysh64.o

mysh64.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   eb 65                   jmp    67 <two>

0000000000000002 <one>:
   2:   5b                      pop    %rbx
   3:   c6 43 0a 01             movb   $0x1,0xa(%rbx)
   7:   c7 43 d8 62 61 73 68    movl   $0x68736162,-0x28(%rbx)
   e:   c7 43 dc 2f 62 69 6e    movl   $0x6e69622f,-0x24(%rbx)
  15:   66 c7 43 e2 2d 63       movw   $0x632d,-0x1e(%rbx)
  1b:   48 c7 43 e4 65 6c 6c    movq   $0x6f6c6c65,-0x1c(%rbx)
  22:   6f
```

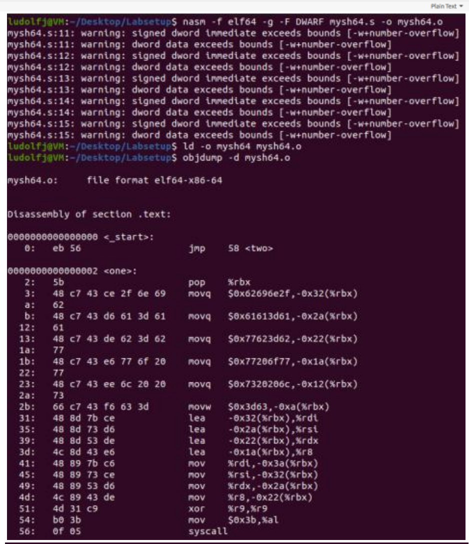


```
  23:   48 c7 43 ec 68 6f 20    movq   $0x68206f68,-0x14(%rbx)
  2a:   68
  2b:   48 c7 43 f4 48 65 6c    movq   $0x6c6c6548,-0xc(%rbx)
  32:   6c
  33:   48 c7 43 fc 73 2d 6c    movq   $0x616c2d73,-0x4(%rbx)
  3a:   61
  3b:   c7 43 04 6f 3b 20 6c    movl   $0x6c203b6f,0x4(%rbx)
  42:   66 c7 43 08 6c 2d       movw   $0x2d6c,0x8(%rbx)
  48:   48 8d 7b d8             lea    -0x28(%rbx),%rdi
  4c:   48 8d 73 e2             lea    -0x1e(%rbx),%rsi
  50:   48 8d 53 e4             lea    -0x1c(%rbx),%rdx
  54:   48 89 7b d0             mov    %rdi,-0x30(%rbx)
  58:   48 89 73 d8             mov    %rsi,-0x28(%rbx)
  5c:   48 89 53 e0             mov    %rdx,-0x20(%rbx)
  60:   4d 31 c0                xor    %r8,%r8
  63:   b0 3b                   mov    $0x3b,%al
  65:   0f 05                   syscall

0000000000000067 <two>:
  67:   e8 96 ff ff ff 2f 62 69 6e 2f 62 61 73 68 01 2d     ...../bin/bash.-
  77:   63 01 65 63 68 6f 20 68 65 6c 6c 6f 3b 20 6c 73     c.echo hello; ls
  87:   20 2d 6c 61 01                                       -la.
```

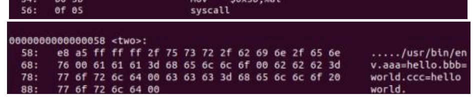
- Then in task 2.d, I had to pass environment variables to execve system call by modifying mysh64.s file.



```
section .text
global _start
_start:
    BITS 64
    jmp short two

one:
    pop rbx

    ; Constructing the strings on the stack
    mov qword [rbx-50], 0x766e652f62696e2f  ; "/usr/bin/env" in reverse order
    mov qword [rbx-42], 0x6c6c656861613d61  ; "aaa=hello" in reverse order
    mov qword [rbx-34], 0x646c726f77623d62  ; "bbb=world" in reverse order
    mov qword [rbx-26], 0x646c726f77206f77  ; Part of "ccc=hello world"
    mov qword [rbx-18], 0x6c6c65687320206c  ; Remaining part of "ccc=hello world"
    mov word [rbx-10], 0x3d63               ; Remaining part of "ccc=hello world"

    ; Constructing env array on the stack
    lea rdi, [rbx-50]  ; Address of "/usr/bin/env"
    lea rsi, [rbx-42]  ; Address of "aaa=hello"
    lea rdx, [rbx-34]  ; Address of "bbb=world"
    lea r8, [rbx-26]   ; Address of "ccc=hello world"

    ; Moving addresses to stack
    mov qword [rbx-58], rdi
    mov qword [rbx-50], rsi
    mov qword [rbx-42], rdx
    mov qword [rbx-34], r8
    xor r9, r9  ; Null terminator for env

    ; Set up syscall for execve
    mov al, 59  ; Syscall number for execve
    syscall

two:
    call one
    db '/usr/bin/env', 0x00
    db 'aaa=hello', 0x00
    db 'bbb=world', 0x00
    db 'ccc=hello world', 0x00
```



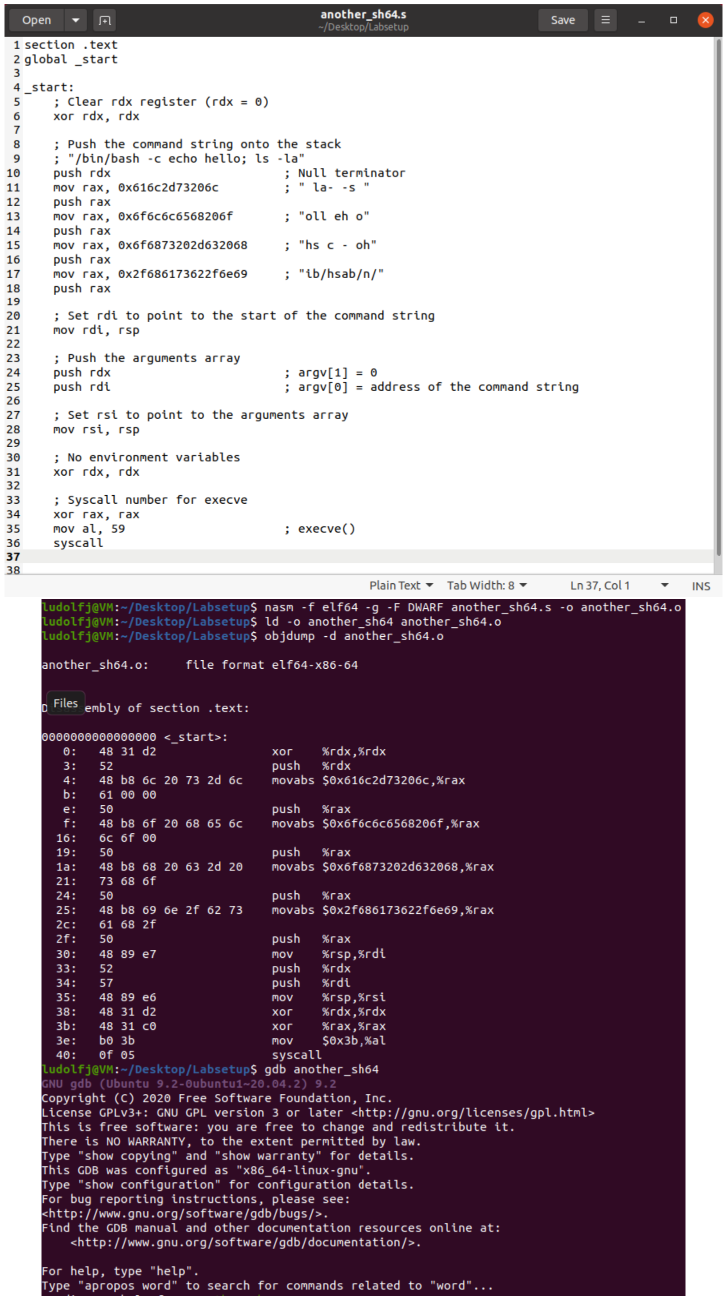
```
ludolfj@VM:~/Desktop/Labsetup$ nasm -f elf64 -g -F DWARF mysh64.s -o mysh64.o
mysh64.s:11: warning: signed dword immediate exceeds bounds [-w+number-overflow]
mysh64.s:11: warning: dword data exceeds bounds [-w+number-overflow]
mysh64.s:12: warning: signed dword immediate exceeds bounds [-w+number-overflow]
mysh64.s:12: warning: dword data exceeds bounds [-w+number-overflow]
mysh64.s:13: warning: signed dword immediate exceeds bounds [-w+number-overflow]
mysh64.s:13: warning: dword data exceeds bounds [-w+number-overflow]
mysh64.s:14: warning: signed dword immediate exceeds bounds [-w+number-overflow]
mysh64.s:14: warning: dword data exceeds bounds [-w+number-overflow]
mysh64.s:15: warning: signed dword immediate exceeds bounds [-w+number-overflow]
mysh64.s:15: warning: dword data exceeds bounds [-w+number-overflow]
ludolfj@VM:~/Desktop/Labsetup$ ld -o mysh64 mysh64.o
ludolfj@VM:~/Desktop/Labsetup$ objdump -d mysh64.o

mysh64.o:     file format elf64-x86-64


Disassembly of section .text:

0000000000000000 <_start>:
   0:   eb 56                   jmp    58 <two>

0000000000000002 <one>:
   2:   5b                      pop    %rbx
   3:   48 c7 43 ce 2f 6e 69    movq   $0x62696e2f,-0x32(%rbx)
   a:   62
   b:   48 c7 43 d6 61 3d 61    movq   $0x61613d61,-0x2a(%rbx)
  12:   61
  13:   48 c7 43 de 62 3d 62    movq   $0x77623d62,-0x22(%rbx)
  1a:   77
  1b:   48 c7 43 e6 77 6f 20    movq   $0x77206f77,-0x1a(%rbx)
  22:   77
  23:   48 c7 43 ee 6c 20 20    movq   $0x7320206c,-0x12(%rbx)
  2a:   73
  2b:   66 c7 43 f6 63 3d       movw   $0x3d63,-0xa(%rbx)
  31:   48 8d 7b ce             lea    -0x32(%rbx),%rdi
  35:   48 8d 73 d6             lea    -0x2a(%rbx),%rsi
  39:   48 8d 53 de             lea    -0x22(%rbx),%rdx
  3d:   4c 8d 43 e6             lea    -0x1a(%rbx),%r8
  41:   48 89 7b c6             mov    %rdi,-0x3a(%rbx)
  45:   48 89 73 ce             mov    %rsi,-0x32(%rbx)
  49:   48 89 53 d6             mov    %rdx,-0x2a(%rbx)
  4d:   4c 89 43 de             mov    %r8,-0x22(%rbx)
  51:   4d 31 c9                xor    %r9,%r9
  54:   b0 3b                   mov    $0x3b,%al
  56:   0f 05                   syscall
```



```
0000000000000058 <two>:
  58:   e8 a5 ff ff ff 2f 75 73 72 2f 62 69 6e 2f 65 6e     ...../usr/bin/en
  68:   76 00 61 61 61 3d 68 65 6c 6c 6f 00 62 62 62 3d     v.aaa=hello.bbb=
  78:   77 6f 72 6c 64 00 63 63 63 3d 68 65 6c 6c 6f 20     world.ccc=hello
  88:   77 6f 72 6c 64 00                                   world.
```

- From that point I was able to examine the second approach for writing the shellcode, which kinda seemed more simplified compared to approach 1.

- Task 3.a is modifying the second approach like we did for task 2.a:

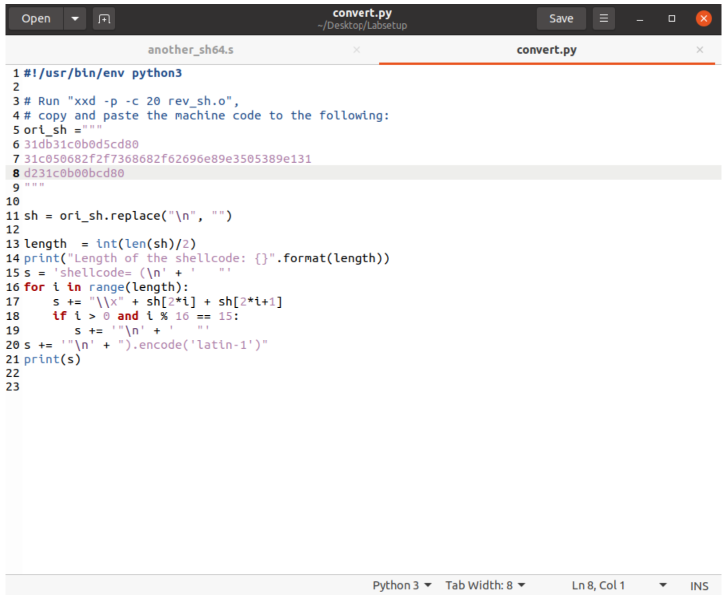


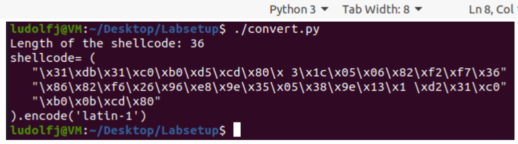
```
another_sh64.s
~/Desktop/Labsetup

section .text
global _start

_start:
    ; Clear rdx register (rdx = 0)
    xor rdx, rdx

    ; Push the command string onto the stack
    ; "/bin/bash -c echo hello; ls -la"
    push rdx                        ; Null terminator
    mov rax, 0x616c2d73206c         ; " la- -s "
    push rax
    mov rax, 0x6f6c6c6568206f       ; "oll eh o"
    push rax
    mov rax, 0x6f6873202d632068     ; "hs c - oh"
    push rax
    mov rax, 0x2f686173622f6e69     ; "ib/hsab/n/"
    push rax

    ; Set rdi to point to the start of the command string
    mov rdi, rsp

    ; Push the arguments array
    push rdx                        ; argv[1] = 0
    push rdi                        ; argv[0] = address of the command string

    ; Set rsi to point to the arguments array
    mov rsi, rsp

    ; No environment variables
    xor rdx, rdx

    ; Syscall number for execve
    xor rax, rax
    mov al, 59                      ; execve()
    syscall
```

```
ludolfj@VM:~/Desktop/Labsetup$ nasm -f elf64 -g -F DWARF another_sh64.s -o another_sh64.o
ludolfj@VM:~/Desktop/Labsetup$ ld -o another_sh64 another_sh64.o
ludolfj@VM:~/Desktop/Labsetup$ objdump -d another_sh64.o

another_sh64.o:     file format elf64-x86-64


D Files embly of section .text:

0000000000000000 <_start>:
   0:   48 31 d2                xor    %rdx,%rdx
   3:   52                      push   %rdx
   4:   48 b8 6c 20 73 2d 6c    movabs $0x616c2d73206c,%rax
   b:   61 00 00
   e:   50                      push   %rax
   f:   48 b8 6f 20 68 65 6c    movabs $0x6f6c6c6568206f,%rax
  16:   6c 6f 00
  19:   50                      push   %rax
  1a:   48 b8 68 20 63 2d 20    movabs $0x6f6873202d632068,%rax
  21:   73 68 6f
  24:   50                      push   %rax
  25:   48 b8 69 6e 2f 62 73    movabs $0x2f686173622f6e69,%rax
  2c:   61 68 2f
  2f:   50                      push   %rax
  30:   48 89 e7                mov    %rsp,%rdi
  33:   52                      push   %rdx
  34:   57                      push   %rdi
  35:   48 89 e6                mov    %rsp,%rsi
  38:   48 31 d2                xor    %rdx,%rdx
  3b:   48 31 c0                xor    %rax,%rax
  3e:   b0 3b                   mov    $0x3b,%al
  40:   0f 05                   syscall
ludolfj@VM:~/Desktop/Labsetup$ gdb another_sh64
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04.2) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
```

- After examining both approaches, I realize that approach 1 maybe the way to go as it would be nice to not have to write register information.
- Finally, using the shellcode in attacking code, I had to run xxd command to get contents of the binary file.

```
ludolfj@VM:~/Desktop/Labsetup$ xxd -p -c 20 another_sh6
7f454c4602010100000000000000000001003e00
0100000000000000000000000000000000000000
4000000000000000000000004000000000004000
0f00020000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000000000000000000000000000
0000000000000000010000000100000006000000
0000000000000000000000000004000000000000
4200000000000000000000000000000001000000
0000000000000000000000000700000003000000
0000000000000000000000000000000050040000
00000000b4000000000000000000000000000000
0100000000000000000000000000000011000000
0200000000000000000000000000000000000000
1005000000000000a80000000000000004000000
0600000008000000000000001800000000000000
1900000003000000000000000000000000000000
00000000c0050000000000001700000000000000
0000000000000000010000000000000000000000
0000000021000000010000000000000000000000
0000000000000000e00500000000000030000000
0000000000000000000000000100000000000000
0000000000000000300000000400000000000000
0000000000000000000000001006000000000000
3000000000000000030000000500000001000000
0000000018000000000000004400000001000000
0000000000000000000000000000000040060000
0000000012000000000000000000000000000000
0100000000000000000000000000000054000000
0100000000000000000000000000000000000000
6006000000000000500000000000000000000000
0000000001000000000000000000000000000000
6000000004000000000000000000000000000000
00000000b0060000000000007800000000000000
0300000008000000010000000000000018000000
0000000071000000010000000000000000000000
0000000000000000300700000000000001b000000
0000000000000000000000000100000000000000
00000000000000007f0000000100000000000000
0000000000000000000000005007000000000000
5100000000000000000000000000000001000000
0000000000000000000000008b00000004000000
00000000000000000000000000000000b0070000
0000000018000000000000000300000b000000
0100000000000000180000000000000009c000000
0100000000000000000000000000000000000000
d007000000000000040000000000000000000000
0000000008000000000000000000000000000000
a900000001000000000000000000000000000000
00000000e0070000000000001000000000000000
0000000000000000010000000000000000000000
000000004831d25248b86c20732d6c6100005048
b86f2068656c6c6f005048b86820632d2073686f
5048b8696e2f627361682f504889e752574889e6
4831d24831c0b03b0f0500000000000000000000
00000000002e74657874002e7368737472746162
002e73796d746162002e737472746162002e6465
6275675f6172616e676573002e72656c612e6465
6275675f6172616e676573002e64656275675f70
75626e616d6573002e64656275675f696e666f00
2e72656c612e64656275675f696e666f002e6465
6275675f616262726576002e64656275675f6c69
6e65002e72656c612e64656275675f6c696e6500
2e64656275675f6672616d65002e64656275675f
6c6f630000000000000000000000000000000000
0000000000000000000000000000000000000000
010000000400f1ff000000000000000000000000
0000000000000000030001000000000000000000
0000000000000000000000000300080000000000
0000000000000000000000000000000003000a00
0000000000000000000000000000000000000000
03000b0000000000000000000000000000000000
1000000010001000000000000000000000000000
00000000000000000000000000616e6f74686572
5f736836342e73005f7374617274000000000000
000000002c000000030000000000080000000000
0000000000000000042000000000000000000000
0000000000000000000000000600000000000000
0a00000003000000000000000000000001000000
0000000001000000020000000000000000000000
0e00000003000000000000000000000000000000
0000000000000000000000004c00000003000000
0000080100000000000000004200000000000000
00000000616e6f746865725f736836342e73004e
41534d20322e31342e3032000180020000000000
0000000000000000000000000600000000000000
0a000000040000000000000000000000c000000
0000000001000000020000000000000000000000
1400000000000000010000000200000000000000
000000001c000000000000000a00000005000000
00000000000000003f0000000000000001000000
0200000000000000000000000000000000000000
0111011101120110060308250813050000022e00
1101400600000000000000004d00000003002500
00000101fb0e0d00010101010000000100000100
616e6f746865725f736836342e73000000000000
09020000000000000000001740219f219f219f219f
233f21233f3f3d2f020200010100000000000000
0000000000000000000320000000000000001000000
0200000000000000000000000000000000000000
0000000000000000000000000000000000000000
000000000000000000000000
```





- From above, I learned the length of the shellcode was 36 (lines) and the sequence is displayed as a sequence of hexadecimal bytes. This particular shellcode is encoded in 'latin-1'.

- Summary of knowledge obtained from this lab:

From this lab, I learned how to write and develop shellcode, which is a small piece of code used as the payload in exploiting software vulnerabilities. I delved into assembly language, gaining direct control over the instructions executed. Disassembling binary code helped me understand its structure and functionality. Writing shellcode posed several challenges, such as ensuring there were no zeros in the binary code and finding the addresses of data used in commands. To address these challenges, I learned techniques like pushing data onto the stack or storing data in the code region to obtain addresses. Through modifying shellcode examples to accomplish various tasks, I appreciated the practical application of shellcode development and how to adapt it for different architectures, such as x86 and x64. Overall, this lab provided me with valuable hands-on experience and a deeper understanding of shellcode.