

Lab 3

Buffer Overflow Exploits

ITSC304: Operating Systems Exploitation

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

Table of Contents

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

[Reading 3](#_Toc477907329)

[Introduction 3](#_Toc477907330)

[1.0 Linux Stack Overflow 4](#_Toc477907331)

2.0 Stack Overflow to Execute Malicious Function ....…………………………………………….7

3.0 Stack Overflow - Exploit ( Shell- Spawning) ……………………….…………………………10

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**

|  |  |  |
| --- | --- | --- |
| Linux Stack Overflow | 15 |  |
| Stack Overflow to Execute Malicious function | 10 |  |
| Stack Overflow –Exploit ( Spawn a Shell) | 15 |  |
| TOTAL MARK | 40 |  |

Lab Outcome(s)

* Linux buffer overflow exploit

Reading

References text books: “Hacking the Art of Exploitation, Jon Erickson, 2nd edition, no starch press chapter 3 and read from <https://www.exploit-db.com/papers> the following paper (PDF)

|  |  |  |
| --- | --- | --- |
| Read from 2019-06-25 |  | [Buffer Overflows, C Programming, NSA GHIDRA and More](https://www.exploit-db.com/docs/47032) |

Introduction

Most program exploits have to do with memory corruption. Most security holes come from overwriting or overflowing the memory area called stack. The boundary of the stack is defined by RSP and RBP registers. Stack- based overflows have been one of the most popular methods of software exploitation. By creating segmentation fault the system will crash generating denial of Service (DoS) but we can take advantage of stack overflow to gain control of the execution, basically gets controlled on what is loaded into instruction pointer (RIP).

The goal is to inject instructions into a buffer and overwrite the return address. Once the return address is overwritten we can have control of program execution flow. After taking control of the system. Hackers can continue exploiting the system with post-exploitation techniques

In this Lab Kali virtual machine will be used (Ubuntu VM can be used as well) to overflow the stack and take advantage of overflowing and overwriting the address that will redirect execution to malicious function such as spawn a shell.

Lab Requirements:

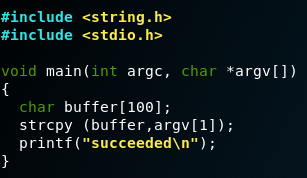
To complete this Lab you need the following:

1. Virtual-Box latest version
2. Linux –Kali operating system

1. Linux Stack Overflow \_\_\_\_/15

The objectives of this exercise are to

1. Analyze the stack using *gdb* debugger and
2. Overflow the stack
3. Analyze the stack
4. Run and compile the following code as follows:



gcc -g –fno-stack-protector -z execstack vuln.c -o vuln

1. Use the following gdb commands to analyze the stack:

gdb ./vuln

NOTE: You can use gdb , peda or GEF debuggers

1. ( 2 marks) .Why is this program vulnerable?

The buffer has a limit of 100 characters, which can cause a buffer overflow if exceeded.

1. break main
2. r $(python -c ‘print “A” \* 119’ ) It will run the program and input 119 characters in the buffer
3. n to move to next line and you will see the printf() function displaying ‘succeeded’
4. info registers Analyze the registers and its respective addresses
   1. (1 mark) What are the registers that control the stack pointers?

RSP and RBP and RIP

* 1. ( 2 marks) Why is the address of rsp lower than rbp?

The lower address is higher on the stack

1. When a function is called a stack frame is push onto the stack. The size and structure of the frame depends on the function and compiler optimization. To verify the stack frame size in bytes for this function, use the command print(p) and perform the following subtraction:

p rbp address - rsp address (replace with respective address)

e.g p 0x7fffffffe190 - 0x7fffffffe110

The stack frame is 128 bytes

1. Use info frame and identify the last saved address for rbp and rip registers

rbp = 0x7fffffffdb80

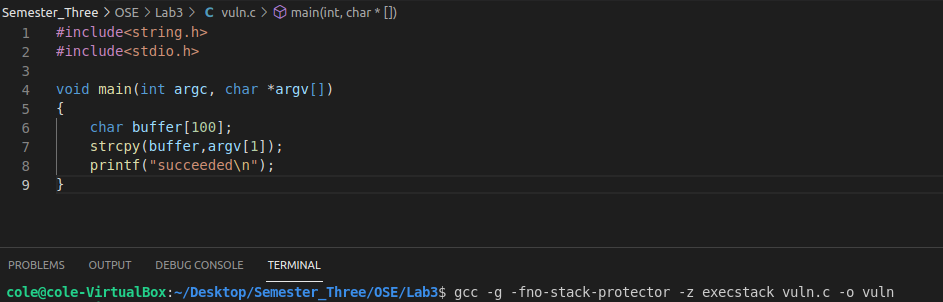
rip = 0x7fffffffdb88

1. Use x/40x $rsp
   1. Analyze the stack frame area for main() from the start of rsp register address to rbp address (stack frame for main())
   2. Identify the input of strings “A” (41) that should be followed by 00 which terminates the string ( 0x00414141).
2. Identify the return address. This should be the first word after rbp register (0x00414141)

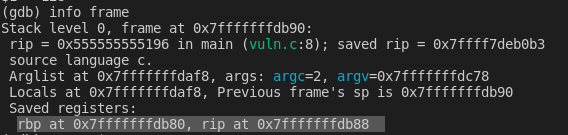
The return address is an important address because it is the next instruction to be executed after main() function returns. If the return address is known, it can be used to control and exploit the program.

The return address is = 0x7fffffffdb80

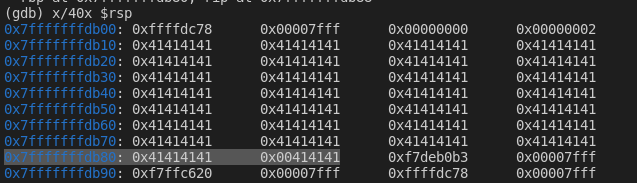
1. Use Info &buffer to verify buffer size and respective address
2. Use c to continue.
3. ( 5 marks) Attach screen captures to demo:
   1. C code created



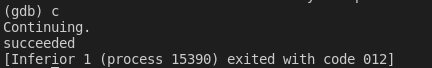
* 1. Identified last address for rbp and rip



* 1. Debug output that displays stack area with 0x41414141 values (A) and underline identified return address in the output



* 1. Final result after pressing c to continue



1. Buffer Overflow

Now that you know how to explore the stack lets overflow it as follows:

1. Create a break point for main
2. r $(python -c ‘print “A” \* 120’ )

NOTE: If 120 characters does not overflow your system keep adding till it overflows it

1. n next
2. x/40x $rsp

Analyze the stack and compare it with previous output.

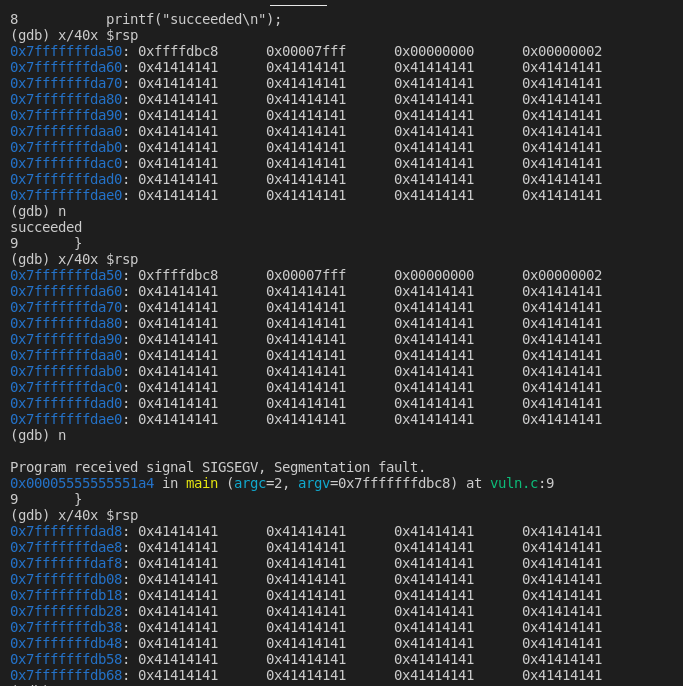
* + (1 marks) What is different?

There is no return address, it is all covered by A’s

* + (1 mark) How many byte(s) were overwritten?

448 bytes, (note I also used 300 characters instead of 120)

1. Use c to continue the system will receive the signal SIGSEGV segmentation fault. Analyze the results
2. By overwritten the return address we can now have control of the program because we can redirect this address to perform a malicious function
3. ( 3 marks) Attach screen captures to demo:
   1. Debug output that displays stack area with 0x41414141 values (A) and underline identified return address in the output



There is no return address to identify here

* 1. Final result after pressing c to continue

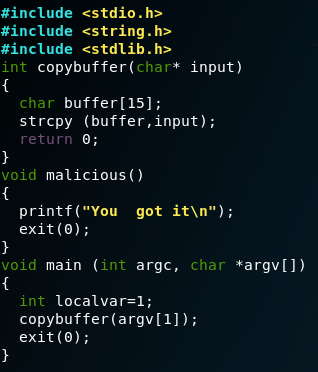


1. Stack Overflow to Execute Malicious Function \_\_\_/10

The objectives of this exercise are to:

1. Use stack overflow to overwrite RIP and take control of the program
2. Exploit the program by executing malicious function

1. Run and compile the following program. Analyze the code and note that the malicious code is never called in main() function and never executed. What if we use buffer overflow to control the execution and change the flow to execute this function?



gcc -g –fno-stack-protector -z execstack bufferover.c -o buff

1. Use the following gdb commands to analyze the stack:
2. gdb ./buff

b. list

1. break main
2. r $(python -c ‘print “a” \* 26’ )
3. next
4. c to continue
5. After segmentation fault pay attention to the error address with ?? . This means you got the return address. Verify how many bytes were injected with ‘a’ string into RIP register e.g 0x0000555500616161 here we have 3 bytes injected with string “a” (61). We need this number to subtract from the initial 26 injected). Record the number of bytes that were injected based on your output:

3 – 26 = 23

1. Use info registers to verify the address of rsp and rbp registers

rbp = 0x6161616161616161

rsp = 0x7fffffffdbc0

1. Use disassemble malicious to find the address of rbp register push into the stack for this function and record the address:

0x00005555555551b3

1. To confirm it use print command as follows:

p malicious You should get the same address as you recorded before

1. To execute the malicious function we need to know the following:
   1. Precisely the amount of strings to inject into return address in this case:

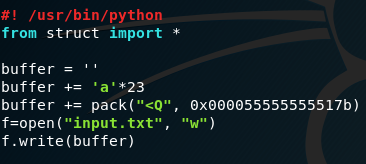
26 -3 =23

26 are the initial 61(s) injected that generated segmentation fault and 3

(61s) are the bytes overwritten the return address in this example is 3

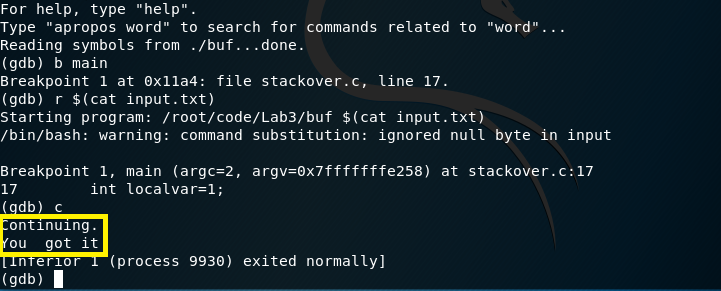
* 1. The rbp address push into the stack for the malicious function

1. Because we cannot pass the rbp register’s address directly we will use python to pack the address of the malicious function as follows:
   1. Create the following python script, save it as *exploibuf*. In this case the address of rbp register push into the stack for malicious function is 0x000055555555517b and 23 is the subtraction from 26-3 explained before. Make sure the number of strings and the address are accordingly to your outputs

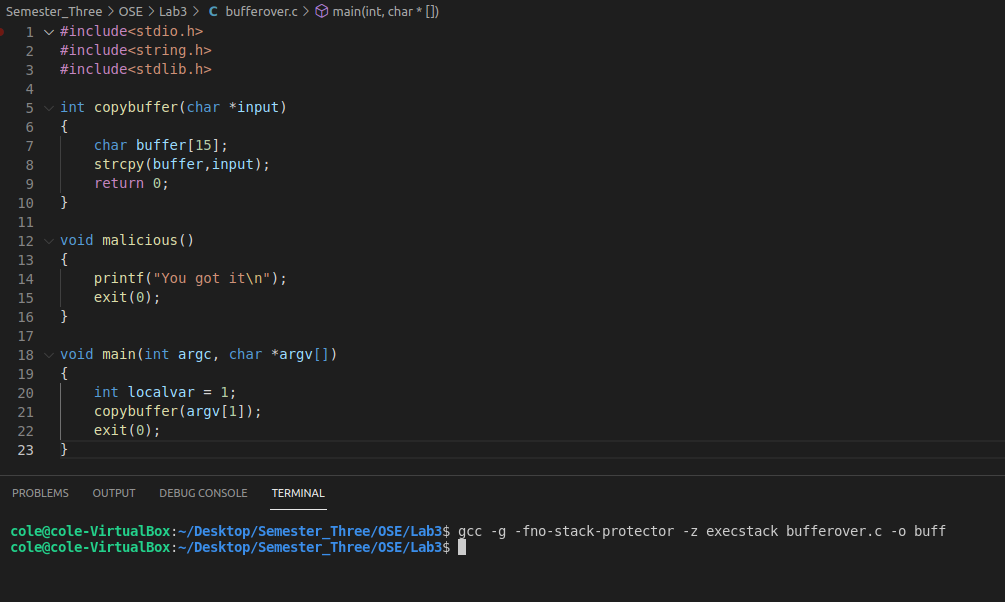


* 1. run the script to generate the input.txt file
  2. chmod 755 exploitbuf
  3. ./exploitbuf

1. Now use gdb debugger to run the exploit as follows:
   1. break main
   2. r $(cat input.txt)
   3. type c to continue
2. If the calculation and address are correct the result should be the following:



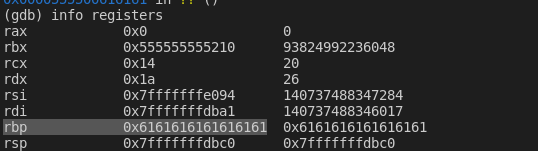
1. This means we have control of the return address and it was redirected to execute a function that should not be executed.
2. ( 10 marks) Attach screen captures to demo:
   1. C code created



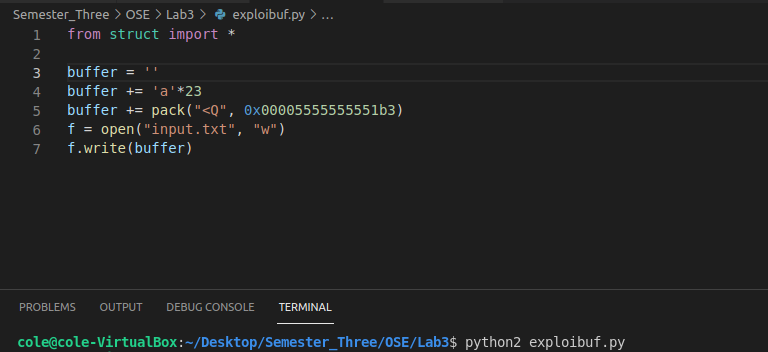
* 1. Debug output that displays stack area and underline identified return address in the output

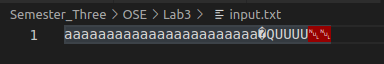


* 1. Buffer overflow results and identified rbp address

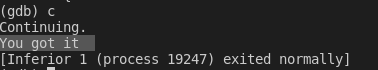


* 1. Python script and results after executing ./exploitbuf





* 1. Debugger with final results after running the exploit



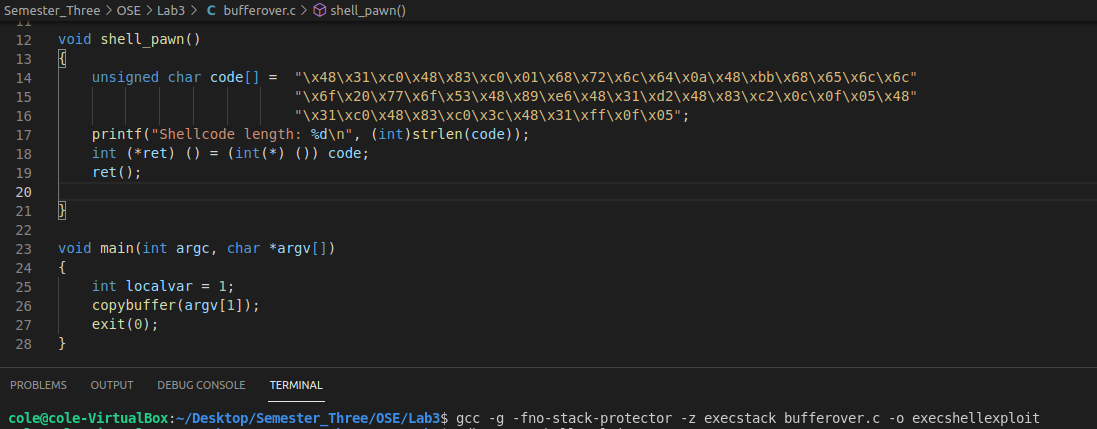
1. Stack Overflow –Exploit (Shell Spawning) \_\_\_\_/15

The objectives of this exercise is to:

1. Use stack overflow to inject execve shellcode and spawn a shell
2. Now in the program bufferover.c replace the malicious function with the function called shell\_pawn() which will spawn a shell (execve shellcode that was generated in Lab2 ) as follows:



1. Save the file as execshellexploit.c and compile it
2. Use gdb to execute it
3. Create a break point at main
4. Run the program with “A” or “B” strings and find the number of strings required to create segmentation fault
5. Analyze the error address ?? after segmentation fault and record the number of injected bytes on RIP address
6. Use info registers, info frames or print command to analyze values of registers such as rsp, rbp, rip
7. Disassemble the new function called shell\_pawn or use print command to find the rbp register address that was push into the stack
8. Create a python script with the required strings “A”s or “B”s to overwrite the return address and pack the address of shell\_pawn function( address you found before) and write the results to shellpawn.txt file
9. Execute the python script to generate the shellpawn.txt file. This file will be used to redirect RIP and execute the desired function that in this case is shell\_pawn
10. Use gdb as previous exercise to run shellpawn.txt file and inject shellcode that should spawn the shell (sh)
11. ( 15 marks) Attach screen captures to demo:
    1. (3 marks) C code created bufferover.c with malicious shellcode



* 1. (3 marks) Debug output that displays stack area and underlined identified injected bytes in RIP address in the output

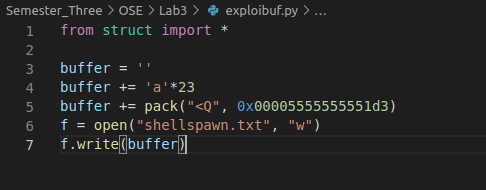


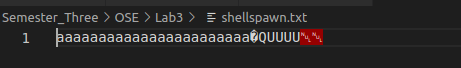
* 1. (3 marks) Buffer overflow results and underlined identified rbp address push on the stack and calculation of amount of strings to inject to spawn the shell (take control of the system).



The calculations ended up being the same as the last example.

* 1. (3 marks) Python script and results after execution





* 1. (3 marks) Debugger with final results after running the exploit

