**CSE 565 Lab 4 Report**

**Buffer-Overflow Attack Lab (Set-UID Version)**

**Notes: (IMPORTANT)**

* It is **required** to use this report template.
* **Select <File> - <Make a copy> to make a copy of this report for yourself.**
* Report your work in each section. **Describe** what you have done and what you have observed. You should take screenshots to support your description. You also need to provide an explanation of the observations that are interesting or surprising. Please also list the important code snippets followed by an explanation.
* Simply attaching code or screenshots without any explanation will NOT receive credits.
* Do **NOT** claim anything you didn’t do. If you didn’t try on a certain task, leave that section blank. You will receive a ZERO for the whole assignment if we find any overclaim.
* Grading will be based on your **description** and the completion of each task.
* After you finish, export this report as a PDF file and submit it on UBLearns.

Your Full Name: Nazmus Saquib

UBITName: nsaquib2

Student Number: 50510460

I, nsaquib2 (UBITName), have read and understood the course academic integrity policy.

(Your report will not be graded without filling in the above AI statement.)

# Task 1: Getting Familiar with Shellcode

## Task: Invoking the Shellcode

(Run two compiled binaries and report your observations. Can you run any commands in the new shell? Can you type and delete your input in the new shell?)

Answer:

I ran two compiled binaries and found out that I can run any regular command (like whoami, ls, cd, etc.) in the new shell. I can type anything, but cannot delete or use backspace in the new shell.

A screenshot of a computer

Description automatically generated

# Task 2: Understanding the Vulnerable Program

Before you run “make” to compile the programs, change the corresponding **Makefile** lines in the Labsetup/code folder as follows:

L1 = 345

L2 = 190

L3 = 130

What is a Set-UID program? Explain how this Set-UID program owned by the root user can potentially be exploited by an attacker to obtain a root shell. In your explanation, make sure to reference the concepts of real UID (RUID) and effective UID (EUID).

Ans:

In Unix and Unix-like operating systems, a Set-UID (Set User ID) application allows users to execute the program with the permissions of the program's owner rather than the rights of the user who runs it. When a Set-UID program is run, it temporarily raises the process's effective user ID (EUID) to that of the program's owner.

Here's how a Set-UID program owned by the root user can potentially be exploited by an attacker to obtain a root shell:

* + - 1. Real UID (RUID) and Effective UID (EUID):

The actual user that owns the process is the real UID (RUID).

The process's effective UID (EUID) is the user whose rights it presently possesses.

* + - 1. Exploiting Set-UID for Privilege Escalation:

When a user runs a Set-UID application, the process's effective UID becomes the owner's UID.

If the Set-UID application is owned by the root user, the process briefly gets root-level privileges while running.

A screen shot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

# Task 3: Launching Attack on 32-bit Program (Level 1)

*(In the following tasks, you need to run and show the “whoami” command result* ***before*** *launching the attack and* ***after*** *getting the root shell in one image to prove that you did not run the vulnerable program under the root user.)*

(Note 2 under section 5.1 is particularly important as you need to take this into consideration when developing your exploit. A guess of 200 bytes for gdb-pushed data would be a good start.)

Carefully follow the instructions in the handout.

Ans:

I have the buffer overflow in stack.c file in the bof function. Before running the make file I changed the L1, L2, L3 value according to the instruction. Then I ran "make" to compile it and got all the executable files, converted as Set-UID. Before starting the debugging I created an empty badfile. After starting the debugger, I created a breakpoint at bof (where I have the buffer overflow) and ran it. Found the offset from the difference between ebp register and start of buffer.

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer code

Description automatically generatedA screenshot of a computer

Description automatically generated

The exploit.py that is creating the payload. I replaced the shellcode with 32 bits shellcode. In line 16 of the screenshot below, I am starting to fill the shellcode. The toal size of badfile is 517 which I chose in the python file. Out of this 517, I have put the shellcode towards the end. I have started it at 400. So, between 400 and 517 I will get the shell code. Previously, I found the difference between ebp and buffer which is offset. At this point I will find the difference between returning address and beginning of the buffer by adding 4 to that number. So 353+ 4 = 357 is the number where my return address is.

Now I need the value of the return address. The return address will help me to jump in that NOP region between the shell code and the return address. so I filled the entire space with NOP. As long as I can jump anywhere in this area I will be able to arrive at the shellcode. So the return address should be a value greater than ebp beacuse that adress is more than ebp. so added 100 to it. (In the screenshot, later when it didn’t work, I added 160 instead of 100, and it worked)

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

I got the illegal instruction error because the return address filled was not correct. So, I changed the return address to the ebp location + 160. Ran the program again and then ran stack-L1, finally got the root provilege.

# Task 4: Launching Attack without Knowing Buffer Size (Level 2)

Carefully follow the instructions in the handout.

Ans:

Like the previous task, created an empty bad file and ran "make" to generate all the executable files. Then ran the debugger on stack-L2-dbg file. Inside that I created a breakpoint. Printed the beginning of the buff address. I didn’t print “ebp” because I am not supposed to know that.

A screenshot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generatedA screen shot of a computer

Description automatically generatedA screenshot of a computer

Description automatically generated

Then I created the payload using the exploit.py file. Instead of putting the shell code in the start location, I put the shell code at the end of the bad file. The return address will take me to somewhere in NOP region. I must just go high enough so I can go inside NOP. As the buffer size is 100-200 bytes long, I’m trying to jump more than 200. I started with 400 bytes and set the return address to “the beginning of the buffer + 400”.

Next, I sprayed the buffer. Previously, I had one return address. Now I don’t know exactly how long the buffer is, so I sprayed the buffer. That means, I must put the return address to many places so that one of them is the actual return address. So, I created a for loop and sprayed the entire buffer with the return address. The range is 50 because 200 (highest buffer) / 4 = 50 (each address is 4 bytes long for 32 bits). Thats why I wrote the range as 50. I simply modified the content so I can put the return address across the entire buffer size.

A screenshot of a computer program

Description automatically generated

A screenshot of a computer

Description automatically generated

In the end, Finally, we got the root privilege.

# Task 5: Launching Attack on 64-bit Program (Level 3)

Carefully follow the instructions in the handout.

Ans:

The initial steps are nearly as similar as the previous ones. Just in place of ebp register in the past, I found out the offset by finding the difference between rbp register and start of buffer.

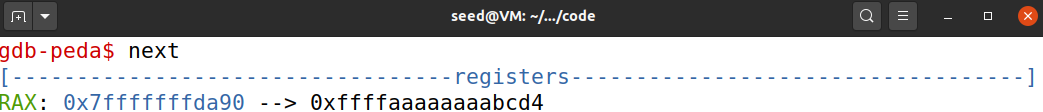
A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated  A screenshot of a computer

Description automatically generated

With the offset, I added 8 because in the 64 bit architecture the base pointer occupies 8 bytes space and the return address is after that.

I added the shellcode in the buffer itself before the return address was overwritten, because 0 values in the return address will cause the strcpy() function to terminate. Therefore, I can’t add anything after the return address like the previous tasks.

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

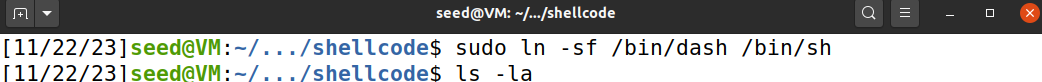
In the end, we got the root privilege.

# Task 7: Defeating dash’s Countermeasure

Carefully follow the instructions in the handout.

Ans:

initially I linked /bin/sh to /bin/zsh because dash terminal has the protection against priority escalation using set-uid file. I have changed the terminal back to dash with the following command.



With zsh terminal, the root privileged setuid binaries could be run to give access to root shell. However, since I have linked /bin/sh back to bin/dash, the protection that dash offers comes back in. Therfore, even with these files, I can't get access to the root terminals as seen in the screen shot.

The dash terminal when it finds the difference in real UID and effective UID changes the effective UID back to the real UID of that process, due to which we are given back the regular terminal access of the user.

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

The setuid(0) system call added to the shell code in call\_shellcode.c was able to bypass the protection given by the dash terminal as seen in the screen shot.

A screenshot of a computer code

Description automatically generated

A screenshot of a computer program

Description automatically generated

**A screenshot of a computer

Description automatically generated**

As the last screen shot shows, /bin/sh points to /bin/dash. This demonstrates that we have bypassed the security measures of dash and performed a level 1 attack.

# Task 9: Experimenting with Other Countermeasures

Carefully follow the instructions in the handout.

## Task 9.a: Turn on the StackGuard Protection

Answer:

At first, I repeated the Level-1 attack with the StackGuard off, and made sure that the attack is still successful.

A screenshot of a computer

Description automatically generated

I removed “-fno-stack-protector” from "Makefile" and recompiled “stack.c” to enable StackGuard protection in the program. StackGuard is a compiler-based security feature designed to protect against stack buffer overflow attacks. As a result, I observed that, the previously successful buffer overflow attack that I committed now no longer works. The StackGuard protection detected the buffer overflow and stopped the program “stack-L1”.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated

## Task 9.b: Turn on the Non-executable Stack Protection

Answer:

I removed the "-z execstack" part from the compilation flag for “call\_shellcode.c”. After removing the flag, our shellcode cannot be executed in the stack. From that I understand that the flag was responsible for suppressing the protection before removing it. It’s the non-executable stack protection that causes it so that the code in the function stack cannot be executed.

A screenshot of a computer

Description automatically generated

A screenshot of a computer program

Description automatically generated