|  |
| --- |
| Graphical user interface, text, application  Description automatically generated |
| **Lab Report** |
| COURSE: Cyber 262  Lab : Buffer Overflow  Submitted BY: Jack Morgan  Date: 04/06/2022  Instructor: Hozza |
| *By submitting this report I certify that this is my work and was not copied from other sources. Any use of public materials is cited.* |

Table of Contents

[INTRODUCTION 3](#_Toc50470226)

[SCREEN CAPTURES 4](#_Toc50470227)

[REFLECTION 9](#_Toc50470228)

# INTRODUCTION

According do the introduction and deliverables, we will be experimenting with Buffer Overflow attacks and attempting to take advantage of this condition to execute malicious pieces of code. We will be using C-programming and python in order to achieve these goals and will be utilizing both prior and current knowledge of buffer overflows to complete this lab. In addition, we will be doing certain important tasks such as Set-UID privileges, learning the idea behind the buffer overflow attack and then launching an attack of our own on a target system.

# 

# SCREEN CAPTURES

***Figure (1.1):*** *Take a screen shot showing the output of “ls -ls” of the myprog file you created with the Set-UID bit set in the permissions*

*This screen shot shows a successful “ls -ls” output that displays the myprog file we created with the Set-UID.*

Text

Description automatically generated

***Figure (2.1):*** *Take a screen shot here showing the output that you received after running the stack c program.*

*After running the ./stack C program we see a successful output of Returned Properly.*

Text

Description automatically generated

***Figure (2.2):*** *Play around by adding characters to the* ***badfile****. How many characters cause the program to run into segmentation-fault and overflow the buffer? What is its significance based on the code?*

*After playing around with the badfile and adding characters, I discovered that after adding 15 characters to the badfile, it created a Segmentation Fault resulting in us overflowing the buffer.*

Graphical user interface, text

Description automatically generated

***Figure (3.1):*** *Take a screen shot of the 3 values you got in $1, $2, and $3 above.*

*Screenshot displays our 3 values we received for the $1, $2 and $3 values.*

Graphical user interface, text, chat or text message

Description automatically generated

***Figure (4.1):*** *Take a screen shot showing escalated privilege (# sign) after running the stack program with the proper payload in badfile*

*After about 4 attempts and realizing I was performing the task on the wrong server, I swapped to PLABCLIENT 1 and successfully reached root privilege as indicated by the # sign.*

Graphical user interface, text, chat or text message

Description automatically generated

***Figure (4.2):*** *Take a screen shot showing the changes you have made in your python script to force the buffer overflow and escalate privileges.*

*This screenshot shows all of the changes made to the python script, including the [D+1] and [D+2] and the D value.*

Graphical user interface, text

Description automatically generated

***Figure (4.3):*** *Do some analysis on your own of the “hexdump” command. Use this command with an option that shows the actual ASCII representation on the right side. This should provide some insight into what badfile is doing when read into the stack program that has UID root privileges. Afterwards, complete the following:*

***Figure (4.3.1):*** *Take a screen shot of the badfile with the ASCII value on the right side.*

Graphical user interface

Description automatically generated with medium confidence

***Figure (4.3.2):*** *What do the 90 values represent?*

90 is the operation code for NOP or No-Operation, meaning the program is told to do nothing or the hex dump has no data/value for that field.

***Figure (4.3.3):*** *Which location contains your ebp address?*

As shown on the right side of the screen, our ebp address is in our //ssh/bin.

***Figure (4.3.4):*** *Explain what the C program will do when it returns to the new address that you have in badfile?*

Since our file was modified, the C program will return to a new location in the stack. My best inference will be that the program will return to the top of the stack.

***Figure (5.1):*** *Take a screen shot of the 3 values you got in $1, $2, and $3 above.*

The following screenshot shows the different values we got for $1, $2 and $3 for the LUBUNTU Server.

A screenshot of a computer

Description automatically generated

***Figure (5.2):*** *Notice that the $3 value of Lubuntu Server is different from Seed server. Do some research and above a reasonable explanation as to why you think there is a difference in both the $3 values of gdb (1-2 sentences)*

Just looking at the screen shots, the $1 value varies greatly on the LUBUNTU server which is why the $3 values for the servers differ greatly. As to why these values are different is beyond my knowledge and an hour of research…My best guess is that the LUBUNTU server does not use the same form of hex that the seed server uses.

# REFLECTION

In this lab, we learned how to enact a buffer overflow attack using GDB protocols and conducted research on some minor topics including the differences between gdb values on different servers and the uses of hexdumping. Although minimal at this point in a cyber analysts carreer, this information will prove valuable in the future to understanding how to defend and ethical enact buffer overflow attacks on foreign actors. Buffer overflow is not an easy topic to understand, as there are many conflicting ideas, however learning the basics of the stack now will prove valuable in the future when we are forced to learn it at a much faster pace for a job or internship.